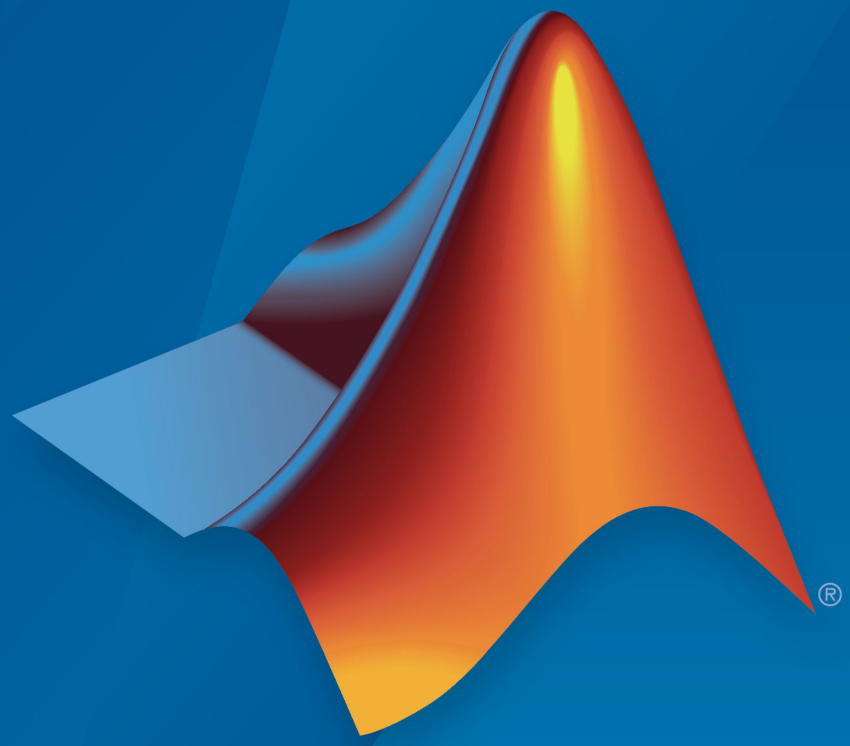
MATLAB®

Function Reference



MATLAB®

R2019b

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

MATLAB Function Reference

© COPYRIGHT 1984–2019 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See `www.mathworks.com/trademarks` for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see `www.mathworks.com/patents` for more information.

Revision History

December 1996	First printing	For MATLAB 5.0 (Release 8)
June 1997	Online only	Revised for MATLAB 5.1 (Release 9)
October 1997	Online only	Revised for MATLAB 5.2 (Release 10)
January 1999	Online only	Revised for MATLAB 5.3 (Release 11)
June 1999	Second printing	For MATLAB 5.3 (Release 11)
June 2001	Online only	Revised for MATLAB 6.1 (Release 12.1)
July 2002	Online only	Revised for 6.5 (Release 13)
June 2004	Online only	Revised for 7.0 (Release 14)
September 2006	Online only	Revised for 7.3 (Release 2006b)
March 2007	Online only	Revised for 7.4 (Release 2007a)
September 2007	Online only	Revised for Version 7.5 (Release 2007b)
March 2008	Online only	Revised for Version 7.6 (Release 2008a)
October 2008	Online only	Revised for Version 7.7 (Release 2008b)
March 2009	Online only	Revised for Version 7.8 (Release 2009a)
September 2009	Online only	Revised for Version 7.9 (Release 2009b)
March 2010	Online only	Revised for Version 7.10 (Release 2010a)
September 2010	Online only	Revised for Version 7.11 (Release 2010b)
April 2011	Online only	Revised for Version 7.12 (Release 2011a)
September 2011	Online only	Revised for Version 7.13 (Release 2011b)
March 2012	Online only	Revised for Version 7.14 (Release 2012a)
September 2012	Online only	Revised for Version 8.0 (Release 2012b)
March 2013	Online only	Revised for Version 8.1 (Release 2013a)
September 2013	Online only	Revised for Version 8.2 (Release 2013b)
March 2014	Online only	Revised for Version 8.3 (Release 2014a)
October 2014	Online only	Revised for Version 8.4 (Release 2014b)
March 2015	Online only	Revised for Version 8.5 (Release 2015a)
September 2015	Online only	Revised for Version 8.6 (Release 2015b)
October 2015	Online only	Rereleased for Version 8.5.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 9.0 (Release 2016a)
September 2016	Online only	Revised for Version 9.1 (Release 2016b)
March 2017	Online only	Revised for Version 9.2 (Release 2017a)
September 2017	Online only	Revised for Version 9.3 (Release 2017b)
March 2018	Online only	Revised for Version 9.4 (Release 2018a)
September 2018	Online only	Revised for Version 9.5 (Release 2018b)
March 2019	Online only	Revised for Version 9.6 (Release 2019a)
September 2019	Online only	Revised for Version 9.7 (Release 2019b)

Contents

1 Alphabetical List

Alphabetical List

Logical Operators: Short-Circuit && ||

Logical operations with short-circuiting

Syntax

```
expr1 && expr2
expr1 || expr2
```

Description

`expr1 && expr2` represents a logical `AND` operation that employs short-circuiting behavior on page 1-4. That is, `expr2` is not evaluated if `expr1` is logical `0` (`false`). Each expression must evaluate to a scalar logical result.

`expr1 || expr2` represents a logical `OR` operation that employs short-circuiting behavior on page 1-4. That is, `expr2` is not evaluated if `expr1` is logical `1` (`true`). Each expression must evaluate to a scalar logical result.

Examples

Use Scalar Logical Conditions

Create two vectors.

```
X = [1 0 0 1 1];
Y = [0 0 0 0 0];
```

Using the short-circuit OR operator with `X` and `Y` returns an error. The short-circuit operators operate only with scalar logical conditions.

Use the `any` and `all` functions to reduce each vector to a single logical condition.

```
any(X) || all(Y)
```

```
ans = logical
   1
```

The expression is equivalent to `1 OR 0`, so it evaluates to logical `1` (`true`) after computing only the first condition, `any(X)`.

Specify Dependent Logical Conditions

Specify a logical statement where the second condition depends on the first. In the following statement, it doesn't make sense to evaluate the relation on the right if the divisor, `b`, is zero.

```
b = 1;
a = 20;
x = (b ~= 0) && (a/b > 18.5)
```

```
x = logical
   1
```

The result is logical `1` (`true`). However, if `(b ~= 0)` evaluates to `false`, MATLAB® assumes the entire expression to be `false` and terminates its evaluation of the expression early.

Specify `b = 0` and evaluate the same expression.

```
b = 0;
x = (b ~= 0) && (a/b > 18.5)
```

```
x = logical
   0
```

The result is logical `0` (`false`). The first statement evaluates to logical `0` (`false`), so the expression short-circuits.

Change Structure Field Value

Create a structure with fields named `'File'` and `'Format'`.

```
S = struct('File',{'myGraph'},'Format',[])
```

```
S = struct with fields:
      File: 'myGraph'
    Format: []
```

Short-circuit expressions are useful in `if` statements when you want multiple conditions to be true. The conditions can build on one another in such a way that it only makes sense to evaluate the second expression if the first expression is true.

Specify an `if` statement that executes only when `S` contains an empty field named `'Format'`.

```
if isfield(S,'Format') && isempty(S.Format)
  S.Format = '.png';
end
S
```

```
S = struct with fields:
      File: 'myGraph'
    Format: '.png'
```

The first condition tests if `'Format'` is the name of a field in structure `S`. The second statement then tests whether the `Format` field is empty. The truth of the second condition depends on the first. The second condition can never be true if the first condition is not true. Since `S` has an empty field named `'Format'`, the body statement executes and assigns `S.Format` the value `'.png'`.

More About

Logical Short-Circuiting

With logical short-circuiting, the second operand, `expr2`, is evaluated only when the result is not fully determined by the first operand, `expr1`.

Due to the properties of logical AND and OR, the result of a logical expression is sometimes fully determined before evaluating all of the conditions. The logical `and` operator returns logical `0` (`false`) if even a single condition in the expression is false. The logical `or` operator returns logical `1` (`true`) if even a single condition in the expression is true. When the evaluation of a logical expression terminates early by encountering one of these values, the expression is said to have *short-circuited*.

For example, in the expression `A && B`, MATLAB does not evaluate condition `B` at all if condition `A` is false. If `A` is false, then the value of `B` does not change the outcome of the operation.

When you use the element-wise `&` and `|` operators in the context of an `if` or `while` loop expression (and *only* in that context), they use short-circuiting to evaluate expressions.

Note Always use the && and `||` operators to enable short-circuit evaluation. Using the `&` and `|` operators for short-circuiting can yield unexpected results when the expressions do not evaluate to logical scalars.

See Also

`all` | `and` | `any` | `false` | `find` | `logical` | `or` | `true` | `xor`

Topics

"Reduce Logical Arrays to Single Value"
"MATLAB Operators and Special Characters"

Introduced before R2006a

colon, :

Vector creation, array subscripting, and `for`-loop iteration

Syntax

```
x = j:k
```

```
x = j:i:k
```

```
A(:,n)
```

```
A(m,:)
```

```
A(:)
```

```
A(j:k)
```

Description

The colon is one of the most useful operators in MATLAB. It can create vectors, subscript arrays, and specify `for` iterations.

`x = j:k` creates a unit-spaced vector `x` with elements `[j,j+1,j+2,...,j+m]` where `m = fix(k-j)`. If `j` and `k` are both integers, then this is simply `[j,j+1,...,k]`.

`x = j:i:k` creates a regularly-spaced vector `x` using `i` as the increment between elements. The vector elements are roughly equal to `[j,j+i,j+2*i,...,j+m*i]` where `m = fix((k-j)/i)`. However, if `i` is not an integer, then floating point arithmetic plays a role in determining whether `colon` includes the endpoint `k` in the vector, since `k` might not be *exactly* equal to `j+m*i`. If you specify nonscalar arrays, then MATLAB interprets `j:i:k` as `j(1):i(1):k(1)`.

`x = colon(j,k)` and `x = colon(j,i,k)` are alternate ways to execute the commands `j:k` and `j:i:k`, but are rarely used. These syntaxes enable operator overloading for classes.

`A(:,n)`, `A(m,:)`, `A(:)`, and `A(j:k)` are common indexing expressions for a matrix `A` that contain a colon. When you use a colon as a subscript in an indexing expression, such

as `A(:,n)`, it acts as shorthand to include *all* subscripts in a particular array dimension. It is also common to create a vector with a colon for the purposes of indexing, such as `A(j:k)`. Some indexing expressions combine both uses of the colon, as in `A(:,j:k)`.

Common indexing expressions that contain a colon are:

- `A(:,n)` is the `n`th column of matrix `A`.
- `A(m,:)` is the `m`th row of matrix `A`.
- `A(:,:,p)` is the `p`th page of three-dimensional array `A`.
- `A(:)` reshapes all elements of `A` into a single column vector. This has no effect if `A` is already a column vector.
- `A(:,:)` reshapes all elements of `A` into a two-dimensional matrix. This has no effect if `A` is already a matrix or vector.
- `A(j:k)` uses the vector `j:k` to index into `A` and is therefore equivalent to the vector `[A(j), A(j+1), ..., A(k)]`.
- `A(:,j:k)` includes all subscripts in the first dimension but uses the vector `j:k` to index in the second dimension. This returns a matrix with columns `[A(:,j), A(:,j+1), ..., A(:,k)]`.

Examples

Create Unit-Spaced Vector

Create a unit-spaced vector of numbers between 1 and 10. The colon operator uses a default increment of +1.

```
x = 1:10
```

```
x = 1×10

     1     2     3     4     5     6     7     8     9    10
```

Create Vector with Specified Increment

Create vectors that increment or decrement by a specified value.

Create a vector whose elements increment by 0.1.

```
x = 0:0.1:1
```

```
x = 1×11

         0    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000    0.7000    0.8
```

Create a vector whose elements decrement by -2.

```
y = 10:-2:0
```

```
y = 1×6

    10     8     6     4     2     0
```

Index Matrix Rows and Columns

Examine several ways to index a matrix using a colon `:`.

Create a 3-by-3 matrix. Index the first row.

```
A = magic(3)
```

```
A = 3×3

     8     1     6
     3     5     7
     4     9     2
```

```
A(1,:)
```

```
ans = 1×3

     8     1     6
```

Index the second and third column.

```
A(:,2:3)
```

```
ans = 3×2

     1     6
     5     7
     9     2
```

Reshape the matrix into a column vector.

```
A(:)
```

```
ans = 9×1

     8
     3
     4
     1
     5
     9
     6
     7
     2
```

Specify `for`-loop Iterations

In the context of a `for`-loop, the colon specifies the loop iterations.

Write a `for`-loop that squares a number for values of `n` between 1 and 4.

```
for n = 1:4
    n^2
end
```

```
ans = 1

ans = 4

ans = 9

ans = 16
```

Input Arguments

j — Starting vector value

scalar

Starting vector value, specified as a real numeric scalar. If `j < k` so that the output vector is not empty, then `j` is the first element in the vector.

Example: `x = 0:5`

Example: `x = 0:0.5:5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `datetime` | `duration`

k — Ending vector value

scalar

Ending vector value, specified as a real numeric scalar. `k` is the last value in the vector only when the increment lines up to exactly land on `k`. For example, the vector `0:5` includes 5 as the last value, but `0:0.3:1` does not include the value 1 as the last value since the increment does not line up with the endpoint.

Example: `x = 0:5`

Example: `x = 0:0.5:5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `datetime` | `duration`

i — Increment between vector elements

1 (default) | scalar

Increment between vector elements, specified as a real numeric scalar.

Example: `x = 0:0.5:5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `datetime` | `duration`

Output Arguments

x — Regularly-spaced vector

row vector

Regularly-spaced vector, returned as a row vector. If `j > k`, then `x = j:k` is an empty matrix. More generally, the syntax `x = j:i:k` returns an empty matrix when:

- `i`, `j`, or `k` is an empty input
- `i == 0`
- `i > 0` and `j > k`
- `i < 0` and `j < k`

Tips

- The `for` reference page has a description of how to use `:` in the context of loop statements.
- `linspace` is similar to the colon operator `:`, but it gives direct control over the number of points and always includes the endpoints. The sibling function `logspace` generates logarithmically spaced values.
- When you create a vector to index into a cell array or structure array (such as *`cellName`*`{:}` or *`structName`*`(:).`*`fieldName`*), MATLAB returns multiple outputs in a comma-separated list. For more information, see “How to Use the Comma-Separated Lists”.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with some limitations. For more information, see “Index and View Tall Array Elements”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Complex inputs are not supported.
- The input `i` cannot have a logical value.
- Vector inputs are not supported.
- Inputs must be constants.
- Uses single-precision arithmetic to produce single-precision results.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- You can build the GPU array equivalent to `A:B` or `A:D:B` in this way:

  ```
  C = gpuArray.colon(A,B)
  C = gpuArray.colon(A,D,B)
  ```

  You can also pass GPU arrays directly to the colon operator (A:D:B).
- 64-bit integers are not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- The colon operator is not supported. Instead, build the distributed array equivalent to `A:B` or `A:D:B` in this way:

  ```
  C = distributed.colon(A,B)
  C = distributed.colon(A,D,B)
  ```

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`for` | `linspace` | `logspace` | `reshape` | `varargin`

Topics

"Generate Sequence of Dates and Time"
"MATLAB Operators and Special Characters"
"Array Indexing"

Introduced before R2006a

abs

Absolute value and complex magnitude

Syntax

```
Y = abs(X)
```

Description

`Y = abs(X)` returns the absolute value on page 1-16 of each element in array `X`.

If `X` is complex, `abs(X)` returns the complex magnitude on page 1-16.

Examples

Absolute Value of Scalar

```
y = abs(-5)
```

```
y = 5
```

Absolute Value of Vector

Create a numeric vector of real values.

```
x = [1.3 -3.56 8.23 -5 -0.01]'
```

```
x = 5×1

    1.3000
   -3.5600
    8.2300
   -5.0000
```

```
   -0.0100
```

Find the absolute value of the elements of the vector.

```
y = abs(x)
```

```
y = 5×1

    1.3000
    3.5600
    8.2300
    5.0000
    0.0100
```

Magnitude of Complex Number

```
y = abs(3+4i)
```

```
y = 5
```

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. If X is complex, then it must be a `single` or `double` array. The size and data type of the output array is the same as the input array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

More About

Absolute Value

The absolute value (or modulus) of a real number is the corresponding nonnegative value that disregards the sign.

For a real value, `a`, the absolute value is:

- `a`, if `a` is greater than or equal to zero
- `-a`, if `a` is less than zero

`abs(-0)` returns `0`.

Complex Magnitude

The complex magnitude (or modulus) is the length of a vector from the origin to a complex value plotted in the complex plane.

For a complex value, $|a + bi|$ is defined as $\sqrt{a^2 + b^2}$.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`angle` | `hypot` | `imag` | `norm` | `real` | `sign` | `unwrap`

Introduced before R2006a

accumarray

Construct array with accumulation

Syntax

```
A = accumarray(subs,val)
A = accumarray(subs,val,sz)
A = accumarray(subs,val,sz,fun)
A = accumarray(subs,val,sz,fun,fillval)
A = accumarray(subs,val,sz,fun,fillval,issparse)
```

Description

`A = accumarray(subs,val)` returns array `A` by accumulating elements on page 1-30 of vector `val` using the subscripts `subs`. If `subs` is a column vector, then each element defines a corresponding subscript in the output, which is also a column vector. The `accumarray` function collects all elements of `val` that have identical subscripts in `subs` and stores their sum in the location of `A` corresponding to that subscript (for index `i`, `A(i)=sum(val(subs(:)==i))`). Elements of `A` whose subscripts do not appear in `subs` are equal to 0.

For an *m*-by-*n* matrix `subs`, each row represents an *n*-dimensional subscript into output `A`. The ith row of `subs` corresponds to the ith element in the vector `val`.

`A = accumarray(subs,val,sz)` returns an array, `A`, with size `sz`. Specify `sz` as a vector of positive integers to define the size of the output, or as `[]` to let the subscripts in `subs` determine the size of the output. Use `sz` when `subs` does not reference trailing rows, columns, or dimensions that you would like to be present in the output.

`A = accumarray(subs,val,sz,fun)` applies the function `fun` to each subset of elements in `val` that have identical subscripts in `subs`. Specify `fun` using the `@` symbol (e.g., `@mean`), or as `[]` to use the default function, `@sum`.

`A = accumarray(subs,val,sz,fun,fillval)` fills all elements of `A` that are not referred to by any subscript in `subs` with the scalar value, `fillval`. The `fillval` input

must have the same class as the values returned by `fun`. Specify `fillval` as `[]` to use the default value, `0`.

`A = accumarray(subs,val,sz,fun,fillval,issparse)` returns an array, `A`, that is sparse if the scalar `issparse` is `true` or `1`, and full if `issparse` is `false` or `0`. The output, `A`, is full by default.

Examples

Find Bin Counts

Create a vector of subscripts, `subs`.

```
subs = [1; 2; 4; 2; 4]
```

```
subs = 5×1

     1
     2
     4
     2
     4
```

Use `accumarray` with `val = 1` to count the number of identical subscripts in `subs`.

```
A = accumarray(subs,1)
```

```
A = 4×1

     1
     2
     0
     2
```

The result is a vector of bin counts. You can obtain the same answer with `histcounts(subs,'BinMethod','integers')`. However, `accumarray` also can compute bin counts over higher dimensional grids.

Accumulate Data

Create a vector of data, `val`, and a vector of subscript values with the same length, `subs`.

```
val = 101:105';
subs = [1; 3; 4; 3; 4]
```

```
subs = 5×1

     1
     3
     4
     3
     4
```

Use `accumarray` to sum the values in `val` that have identical subscripts in `subs`.

```
A = accumarray(subs,val)
```

```
A = 4×1

   101
     0
   206
   208
```

The result is a vector of accumulated values. Since the second and fourth elements of `subs` are equal to 3, `A(3)` is the sum of the second and fourth elements of `val`, that is, `A(3) = 102 + 104 = 206`. Also, `A(2) = 0` because `subs` does not contain the value 2. Since `subs` is a vector, the output, `A`, is also a vector. The length of `A` is `max(subs, [],1)`.

Specify Output Size

Create a vector of data, `val`, and a matrix of subscripts, `subs`.

```
val = 101:106';
subs = [1 1; 2 2; 3 2; 1 1; 2 2; 4 1]
```

```
subs = 6×2
```

```
     1     1
     2     2
     3     2
     1     1
     2     2
     4     1
```

The subscripts in `subs` define a 4-by-2 matrix for the output.

Use `accumarray` to sum the values in `val` that have identical subscripts in `subs`.

```
A = accumarray(subs,val)
```

```
A = 4×2

   205     0
     0   207
     0   103
   106     0
```

The result is a 4-by-2 matrix of accumulated values.

Use the `sz` input of `accumarray` to return a 4-by-4 matrix. You can specify a size with each dimension equal to or greater than the default size, in this case 4-by-2, but not smaller.

```
A = accumarray(subs,val,[4 4])
```

```
A = 4×4

   205     0     0     0
     0   207     0     0
     0   103     0     0
   106     0     0     0
```

The result is a 4-by-4 matrix of accumulated values.

Use Custom Functions

Create a vector of data, `val`, and a matrix of subscripts, `subs`.

```
val = [100.1 101.2 103.4 102.8 100.9 101.5]';
subs = [1 1; 1 1; 2 2; 3 2; 2 2; 3 2]
```

```
subs = 6×2

     1     1
     1     1
     2     2
     3     2
     2     2
     3     2
```

The subscripts in `subs` define a 3-by-2 matrix for the output.

Use the `fun` input of `accumarray` to calculate the within-group variances of data in `val` that have identical subscripts in `subs`. Specify `fun` as `@var`.

```
A1 = accumarray(subs,val,[],@var)
```

```
A1 = 3×2

    0.6050         0
         0    3.1250
         0    0.8450
```

The result is a 3-by-2 matrix of variance values.

Alternatively, you can specify `fun` as an anonymous function so long as it accepts vector inputs and returns a scalar. A common situation where this is useful is when you want to pass additional parameters to a function. In this case, use the `var` function with a normalization parameter.

```
A2 = accumarray(subs,val,[],@(x) var(x,1))
```

```
A2 = 3×2

    0.3025         0
         0    1.5625
         0    0.4225
```

The result is a 3-by-2 matrix of normalized variance values.

Sum Values Natively

Create a vector of data, `val`, and a matrix of subscripts, `subs`.

```
val = int8(10:15);
subs = [1 1 1; 1 1 1; 1 1 2; 1 1 2; 2 3 1; 2 3 2]
```

```
subs = 6×3

     1     1     1
     1     1     1
     1     1     2
     1     1     2
     2     3     1
     2     3     2
```

The subscripts in `subs` define a 2-by-3-by-2 multidimensional array for the output.

Use `accumarray` to sum the data values in `val` that have identical subscripts in `subs`. You can use a function handle to sum the values in their native, `int8`, integer class by using the `'native'` option of the `sum` function.

```
A = accumarray(subs,val,[],@(x) sum(x,'native'))
```

```
A = 2x3x2 int8 array
A(:,:,1) =

   21    0    0
    0    0   14


A(:,:,2) =

   25    0    0
    0    0   15
```

The result is a 2-by-3-by-2 multidimensional array of class `int8`.

Group Values in Cell Array

Create a vector of data, `val`, and a matrix of subscripts, `subs`.

```
val = 1:10;
subs = [1 1;1 1;1 1;1 1;2 1;2 1;2 1;2 1;2 1;2 2]
```

```
subs = 10×2

     1     1
     1     1
     1     1
     1     1
     2     1
     2     1
     2     1
     2     1
     2     1
     2     2
```

The subscripts in `subs` define a 2-by-2 matrix for the output.

Use `accumarray` to group the elements of `val` into a cell array.

```
A = accumarray(subs,val,[],@(x) {x})
```

```
A=2×2 cell
    {4x1 double}    {0x0 double}
    {5x1 double}    {[      10]}
```

The result is a 2-by-2 cell array.

Verify that the vector elements are in the same order as they appear in `val`.

```
A{2,1}
```

```
ans = 5×1

     5
     6
     7
     8
     9
```

Since the subscripts in `subs` are sorted, the elements of the numeric vectors in the cell array are in the same order as they appear in `val`.

Using Functions That Depend on Data Order

Create a vector of data, `val`, and a matrix of subscripts, `subs`.

```
val = 1:5;
subs = [1 2; 1 1; 1 2; 1 1; 2 3]
```

```
subs =

     1     2
     1     1
     1     2
     1     1
     2     3
```

The subscripts in `subs` define a 2-by-3 matrix for the output, but are unsorted with respect to the linear indices in the output, `A`.

Group the values in `val` into a cell array by specifying `fun = @(x) {x}`.

```
A = accumarray(subs,val,[],@(x) {x})
```

```
A =

  2×3 cell array

    [2×1 double]    [2×1 double]     []
               []               []    [5]
```

The result is a 2-by-3 cell array.

Examine the vector in `A{1,2}`.

```
A{1,2}
```

```
ans =

     3
     1
```

The elements of the `A{1,2}` vector are in a different order than in `val`. The first element of the vector is 3 instead of 1. If the subscripts in `subs` are not sorted with respect to their linear indices, then `accumarray` might not always preserve the order of the data in `val` when it passes them to `fun`. In the unusual case that `fun` requires that its input values be in the same order as they appear in `val`, sort the indices in `subs` with respect to the linear indices of the output.

In this case, use the `sortrows` function with two inputs and two outputs to reorder `subs` and `val` concurrently with respect to the linear indices of the output.

```
[S,I] = sortrows(subs,[2,1]);
A = accumarray(S,val(I),[],@(x) {x});
A{1,2}
```

```
ans =

     1
     3
```

The elements of the `A{1,2}` vector are now in sorted order.

Fill Output with NaN Values

Create a vector of data, `val`, and a matrix of subscripts, `subs`.

```
val = 101:106';
subs = [1 1; 2 2; 3 3; 1 1; 2 2; 4 4]
```

```
subs = 6×2

     1     1
     2     2
     3     3
     1     1
```

```
     2     2
     4     4
```

The subscripts in `subs` define a 4-by-4 matrix for the output, but only reference 4 out of the 16 elements. By default, the other 12 elements are `0` in the output.

Use the `fillval` input of `accumarray` to fill in the extra output elements with NaN values.

```
A = accumarray(subs,val,[],[],NaN)
```

```
A = 4×4

   205   NaN   NaN   NaN
   NaN   207   NaN   NaN
   NaN   NaN   103   NaN
   NaN   NaN   NaN   106
```

The result is a 4-by-4 matrix padded with NaN values.

Change Output Sparsity

Create a vector of data, `val`, and a matrix of subscripts, `subs`.

```
val = [34 22 19 85 53 77 99 6];
subs = [1 1; 400 400; 80 80; 1 1; 400 400; 400 400; 80 80; 1 1]
```

```
subs = 8×2

     1     1
   400   400
    80    80
     1     1
   400   400
   400   400
    80    80
     1     1
```

The subscripts in `subs` define a 400-by-400 matrix for the output, but only reference 3 out of the 160,000 elements. When the result of an operation with `accumarray` leads to a

large output array with low density of nonzero elements, you can save storage space by storing the output as a sparse matrix.

Use the `issparse` input of `accumarray` to return a sparse matrix.

```
A = accumarray(subs,val,[],[],[],true)
```

```
A =
   (1,1)       125
  (80,80)      118
 (400,400)     152
```

The result is a sparse matrix. You can obtain the same answer with `sparse(subs(:,1),subs(:,2),val)`.

Input Arguments

`subs` — Subscript matrix

vector of indices | matrix of indices | cell array of index vectors

Subscript matrix, specified as a vector of indices, matrix of indices, or cell array of index vectors. The indices must be positive integers:

- The value in each row of the *m*-by-*n* matrix, `subs`, specifies an *n*-dimensional index into the output, `A`. For example, if `subs` is a 3-by-2 matrix, it contains three 2-D subscripts. `subs` also can be a column vector of indices, in which case the output, `A`, is also a column vector.
- The ith row in `subs` corresponds to the ith data value in `val`.
- If `subs` is empty, then the `accumarray` function errors.

Thus, `subs` determines which data in `val` to group, as well as its final destination in the output. If `subs` is a cell array of index vectors, each vector must have the same length, and the function treats the vectors as columns of a subscript matrix.

`val` — Data

vector | scalar

Data, specified as a vector or scalar:

- If `val` is a vector, it must have the same length as the number of rows in `subs`.
- If `val` is a scalar, it is scalar expanded.

In both cases, a one-to-one pairing is present between the subscripts in each row of `subs` and the data values in `val`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

`sz` — Size of output array
`[]` (default) | vector of positive integers

Size of output array, specified as a vector of positive integers or `[]` (default). When you specify `[]` for the default size, the values in `subs` determine the size of the output array, `A`.

When you specify `sz` as a vector of positive integers, it must satisfy these properties:

- If `subs` is a nonempty *m*-by-*n* matrix with `n > 1` columns, then `sz` must have *n* elements and pass the logical test `all(sz >= max(subs,[],1))`.
- If `subs` is a nonempty column vector, then `sz` must be `[m 1]` where `m >= max(subs)`.

Example: `sz = [3 3]`

`fun` — Function
`[]` (default) | function handle

Function, specified as a function handle or `[]` (default). The default function is `@sum`. The `fun` function must accept a column vector and return a numeric, `logical`, or `char` scalar, or a scalar `cell`. If the subscripts in `subs` are not sorted with respect to their linear indices, `fun` should not depend on the order of the values in its input data. For more information on function handles, see “Create Function Handle”.

Example: `fun = @max`

Data Types: `function_handle`

`fillval` — Fill value
`[]` (default) | scalar

Fill value, specified as a scalar or `[]` (default). The default value of `fillval` is `0`. If `subs` does not reference each element in the output, `accumarray` fills in the output with the

value specified by `fillval`. The class of `fillval` must be the same as the values returned by `fun`.

`issparse` — Output sparsity
`false` or `0` (default) | `true` or `1`

Output sparsity, specified as a numeric or logical `1` (`true`) or `0` (`false`). Specify `true` or `1` when you want the output array to be sparse. If `issparse` is `true` or `1`:

- `fillval` must be `0` or `[]`.
- The values in `val` and the output values of `fun` must both have type `double`.

Output Arguments

`A` — Output array
vector | matrix | multidimensional array

Output array, returned as a vector, matrix, or multidimensional array. `A` has the same class as the values returned by `fun`.

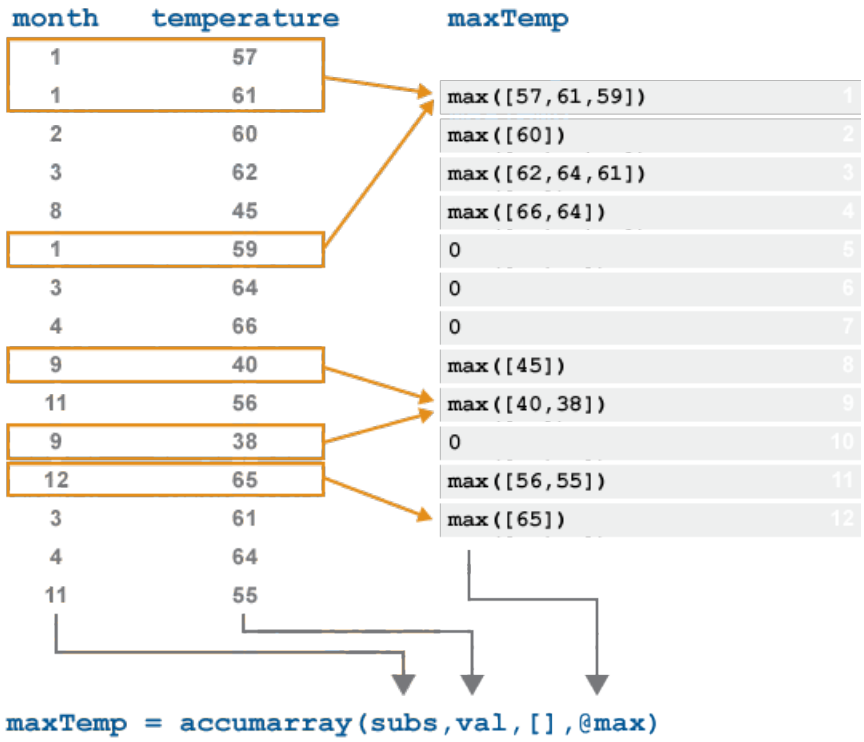
When `sz` is not specified, the size of `A` depends on the input `subs`:

- If `subs` is a nonempty matrix with `n > 1` columns, then `A` is an *n*-dimensional array of size `max(subs,[],1)`.
- If `subs` is an empty matrix with `n > 1` columns, then `A` is an *n*-dimensional empty array with size 0-by-0-by-...-by-0.
- If `subs` is a nonempty column vector, then `A` is a column vector of length `max(subs,[],1)`. The length of `A` is `0` when `subs` is empty.

More About

Accumulating Elements

The following graphic illustrates the behavior of `accumarray` on a vector of temperature data taken over a 12-month period. To find the maximum temperature reading for each month, `accumarray` applies the `max` function to each group of values in `temperature` that have identical subscripts in `month`.



No values in `month` point to the 5, 6, 7, or 10 positions of the output. These elements are `0` in the output by default, but you can specify a value to fill in using `fillval`.

Tips

- The behavior of `accumarray` is similar to that of the `histcounts` function. Both functions group data into bins.
  - `histcounts` groups continuous values into a 1-D range using bin edges. `accumarray` groups data using *n*-dimensional subscripts.
  - `histcounts` returns the bin counts and/or bin placement. However, `accumarray` can apply any function to the binned data.

  You can mimic the behavior of `histcounts` using `accumarray` with `val = 1`.

- The `sparse` function also has accumulation behavior similar to that of `accumarray`.
  - `sparse` groups data into bins using 2-D subscripts, whereas `accumarray` groups data into bins using *n*-dimensional subscripts.
  - For elements with identical subscripts, `sparse` assigns the sum of those elements to the output. `accumarray` does the same by default, but optionally can apply any function to the bins.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Unless you provide explicit bounds on the `accumarray` input values, the code generator may not be able to predetermine the size of the `accumarray` output. Therefore, variable-size arrays and dynamic memory allocation must be enabled.
- Enumeration inputs are not supported.
- The input argument `fun` must return a full (non-sparse) scalar.
- When `fun` produces a scalar 1-by-1 cell array output:
  - The `fillval` input is required, must be a scalar 1-by-1 cell array, and must contain the same type as the output of `fun` contains.
  - The order of accumulated values that `fun` receives as input may not match the order in MATLAB.
- To predetermine the output type for `fun`, the code generator may call `fun` before processing the `accumarray` input arguments. If the execution of `fun` causes side effects, for instance by modifying a global or persistent variable or printing to output, then the generated code results may differ from MATLAB results.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `fun` argument can be `@sum` (default), `@prod`, `@min`, `@max`, `@any`, or `@all`.
- `val` and `fillval` must be full (nonsparse) logical or floating-point (double or single) arrays.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`full` | `histcounts` | `sparse` | `sum`

Introduced before R2006a

acos

Inverse cosine in radians

Syntax

```
Y = acos(X)
```

Description

`Y = acos(X)` returns the "Inverse Cosine" on page 1-37 ($\cos^{-1}$) of the elements of `X` in radians. The function accepts both real and complex inputs.

- For real values of `X` in the interval [-1, 1], `acos(X)` returns values in the interval [0, π].
- For real values of `X` outside the interval [-1,1] and for complex values of `X`, `acos(X)` returns complex values.

Examples

Inverse Cosine of Value

Find the inverse cosine of a value.

```
y = acos(0)
```

```
y = 1.5708
```

Inverse Cosine of Vector of Complex Values

Find the inverse cosine of the elements of vector `x`. The `acos` function acts on `x` element-wise.

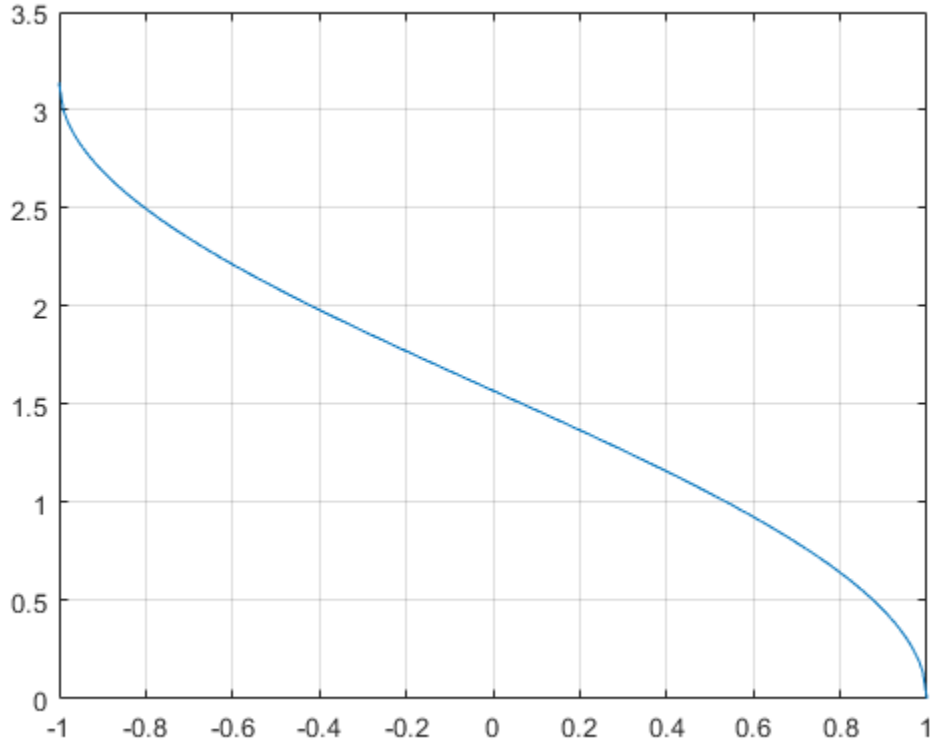
```
x = [0.5i 1+3i -2.2+i];
y = acos(x)
```

```
y = 1×3 complex

   1.5708 - 0.4812i   1.2632 - 1.8642i   2.6799 - 1.5480i
```

Plot Inverse Cosine Function

Plot the inverse cosine function over the intervals $-1 \le x \le 1$.

```
x = -1:.01:1;
plot(x,acos(x))
grid on
```



Input Arguments

X — Cosine of angle
scalar | vector | matrix | multidimensional array

Cosine of angle, specified as a scalar, vector, matrix, or multidimensional array. The `acos` operation is element-wise when X is nonscalar.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Inverse Cosine

The inverse cosine is defined as

$$\cos^{-1}(z) = -i\log\left[z + i\left(1 - z^2\right)^{1/2}\right].$$

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Generates an error during simulation and returns `NaN` in generated code when the input value `X` is real, but the output should be complex. To get the complex result, make the input value complex by passing in `complex(X)`.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Generates an error during simulation and returns `NaN` in generated code when the input value `X` is real, but the output should be complex. To get the complex result, make the input value complex by passing in `complex(X)`.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- If the output of the function running on the GPU can be complex, then you must explicitly specify its input arguments as complex. For more information, see “Work with Complex Numbers on a GPU” (Parallel Computing Toolbox).

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`acosd` | `asec` | `cos` | `cosd`

Introduced in R2006a

acosd

Inverse cosine in degrees

Syntax

```
Y = acosd(X)
```

Description

`Y = acosd(X)` returns the inverse cosine ($\cos^{-1}$) of the elements of `X` in degrees. The function accepts both real and complex inputs.

- For real values of `X` in the interval [-1, 1], `acosd(X)` returns values in the interval [0, 180].
- For values of `X` outside the interval [-1, 1] and for complex values of `X`, `acosd(X)` returns complex values.

Examples

Inverse Cosine of 0

Verify that inverse cosine of 0 is exactly 90.

```
acosd(0)
```

```
ans = 90
```

Round-Trip Calculation for Complex Angles

Show that the inverse cosine, followed by cosine, returns the original values of `X`.

```
cosd(acosd([2 3]))
```

```
ans = 1×2

     2     3
```

`acosd([2 3])` returns two complex angles, which are then passed to the `cosd` function. `cosd` returns the original values, 2 and 3.

Input Arguments

X — Cosine of angle
scalar value | vector | matrix | N-D array

Cosine of angle, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `acosd` operation is element-wise when X is non-scalar.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Angle in degrees
scalar value | vector | matrix | N-D array

Angle in degrees, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as X.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`acos` | `cos` | `cosd`

Introduced before R2006a

acosh

Inverse hyperbolic cosine

Syntax

```
Y = acosh(X)
```

Description

`Y = acosh(X)` returns the inverse hyperbolic cosine of the elements of `X`. The function accepts both real and complex inputs. All angles are in radians.

Examples

Inverse Hyperbolic Cosine of Vector

Find the inverse hyperbolic cosine of the elements of vector `X`. The `acosh` function acts on `X` element-wise.

```
X = [2 -3 1+2i];
Y = acosh(X)
```

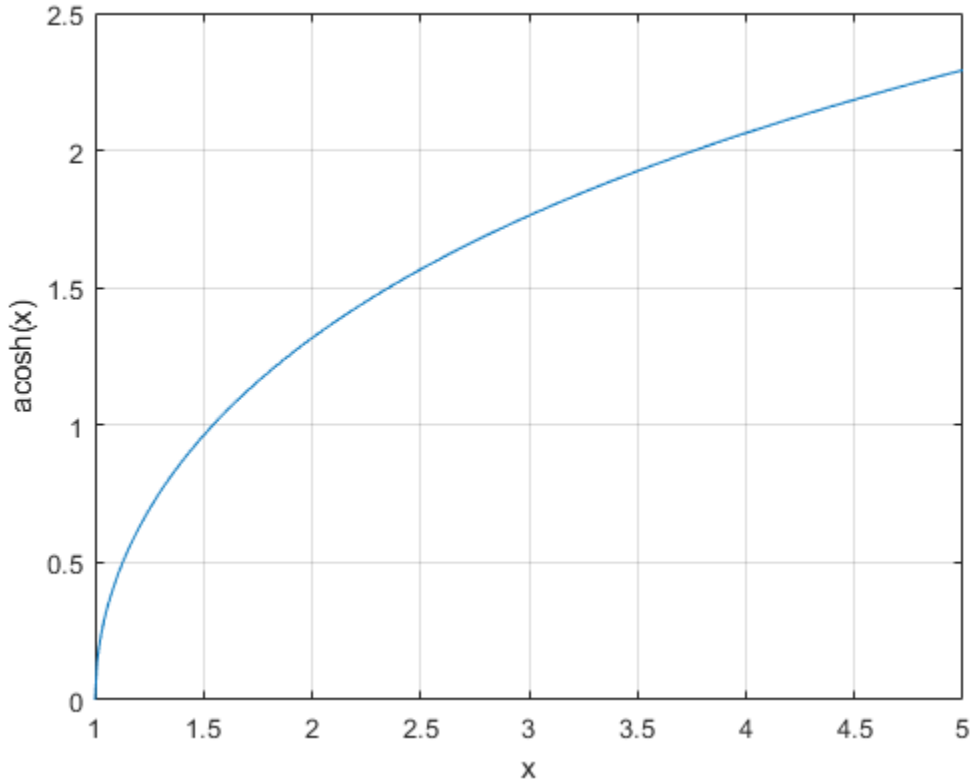
```
Y = 1×3 complex

   1.3170 + 0.0000i   1.7627 + 3.1416i   1.5286 + 1.1437i
```

Plot the Inverse Hyperbolic Cosine Function

Plot the inverse hyperbolic cosine function over the interval $1 \le x \le 5$.

```
x = 1:0.01:5;
plot(x,acosh(x))
```

```
grid on
xlabel('x')
ylabel('acosh(x)')
```



Input Arguments

X — Hyperbolic cosine of angle
scalar | vector | matrix | multidimensional array

Hyperbolic cosine of angle, specified as a scalar, vector, matrix, or multidimensional array. The `acosh` operation is element-wise when `X` is nonscalar.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Inverse Hyperbolic Cosine

For real values x in the domain $x > 1$, the inverse hyperbolic cosine satisfies

$$\cosh^{-1}(x) = \log\left(x + \sqrt{x^2 - 1}\right).$$

For complex numbers $z = x + iy$, as well as real values in the domain $-\infty < z \leq 1$, the call `acosh(z)` returns complex results.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Generates an error during simulation and returns `NaN` in generated code when the input value `x` is real, but the output should be complex. To get the complex result, make the input value complex by passing in `complex(x)`.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Generates an error during simulation and returns `NaN` in generated code when the input value `X` is real, but the output should be complex. To get the complex result, make the input value complex by passing in `complex(X)`.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- If the output of the function running on the GPU can be complex, then you must explicitly specify its input arguments as complex. For more information, see “Work with Complex Numbers on a GPU” (Parallel Computing Toolbox).

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`acos` | `asinh` | `atanh` | `cosh`

Introduced before R2006a

acot

Inverse cotangent in radians

Syntax

```
Y = acot(X)
```

Description

`Y = acot(X)` returns the "Inverse Cotangent" on page 1-49 ($\cot^{-1}$) of the elements of `X` in radians. The function accepts both real and complex inputs.

- For real values of `X`, `acot(X)` returns values in the interval [-π/2, π/2].
- For complex values of `X`, `acot(X)` returns complex values.

Examples

Inverse Cotangent of a Value

Find the inverse cotangent of a value.

```
acot(2.6)
```

```
ans = 0.3672
```

Inverse Cotangent of a Vector of Complex Values

Find the inverse cotangent of the elements of vector `x`. The `acot` function acts on `x` element-wise.

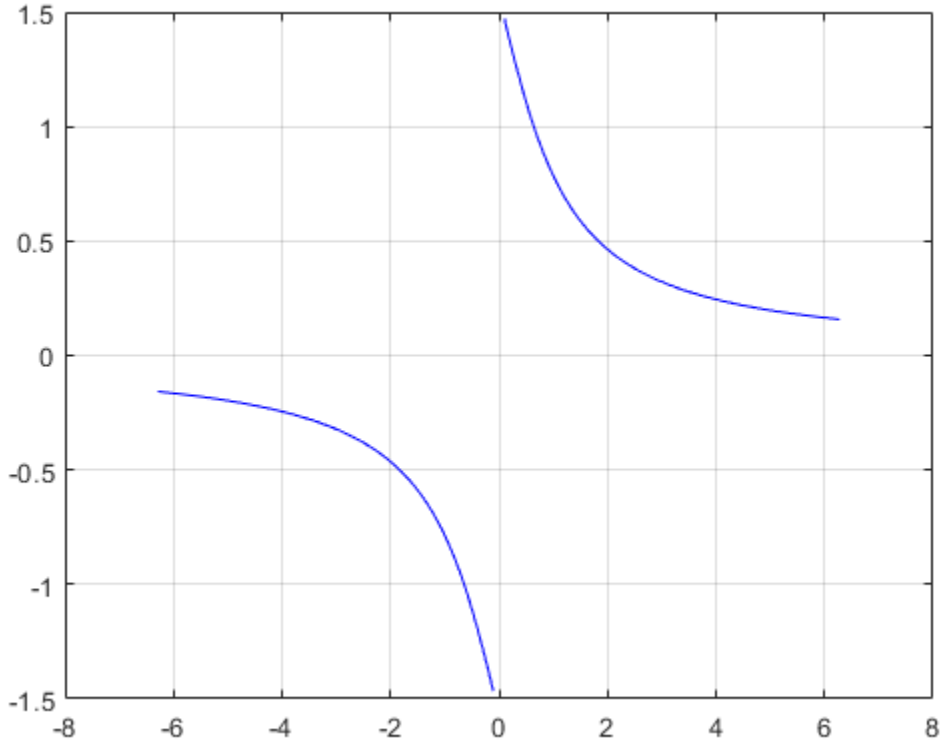
```
x = [0.5i 1+3i -2.2+i];
Y = acot(x)
```

```
Y = 1×3 complex

   1.5708 - 0.5493i   0.1093 - 0.3059i  -0.3689 - 0.1506i
```

Plot the Inverse Cotangent Function

Plot the inverse cotangent function over the intervals $-2\pi \le x < 0$ and $0 < x \le 2\pi$.

```
x1 = -2*pi:pi/30:-0.1;
x2 = 0.1:pi/30:2*pi;
plot(x1,acot(x1),'b')
hold on
plot(x2,acot(x2),'b')
grid on
```



Input Arguments

X — Cotangent of angle
scalar | vector | matrix | multidimensional array

Cotangent of angle, specified as a scalar, vector, matrix, or multidimensional array. The `acot` operation is element-wise when `X` is nonscalar.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Inverse Cotangent

The inverse cotangent is defined as

$$\cot^{-1}(z) = \tan^{-1}\left(\frac{1}{z}\right).$$

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`acotd` | `atan` | `cot` | `cotd`

Introduced before R2006a

acotd

Inverse cotangent in degrees

Syntax

```
Y = acotd(X)
```

Description

`Y = acotd(X)` returns the inverse cotangent ($\cot^{-1}$) of the elements of `X` in degrees. The function accepts both real and complex inputs.

- For real values of `X`, `acotd(X)` returns values in the range [-90, 90].
- For complex values of `X`, `acotd(X)` returns complex values.

Examples

Inverse Cotangent of Vector

```
x = [0 20 Inf];
y = acotd(x)
```

```
y = 1×3

   90.0000    2.8624         0
```

The `acotd` operation is element-wise when you pass a vector, matrix, or N-D array.

Inverse Cotangent of Complex Value

```
acotd(1+i)
```

```
ans = 31.7175 - 23.0535i
```

Input Arguments

X — Cotangent of angle
scalar value | vector | matrix | N-D array

Cotangent of angle, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `acotd` operation is element-wise when X is non-scalar.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Angle in degrees
scalar value | vector | matrix | N-D array

Angle in degrees, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as X.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`acot` | `cot` | `cotd`

Introduced before R2006a

acoth

Inverse hyperbolic cotangent

Syntax

```
Y = acoth(X)
```

Description

`Y = acoth(X)` returns the inverse hyperbolic cotangent of the elements of `X`. The function accepts both real and complex inputs. All angles are in radians.

Examples

Inverse Hyperbolic Cotangent of Vector

Find the inverse hyperbolic cotangent of the elements of vector `X`. The `acoth` function acts on `X` element-wise.

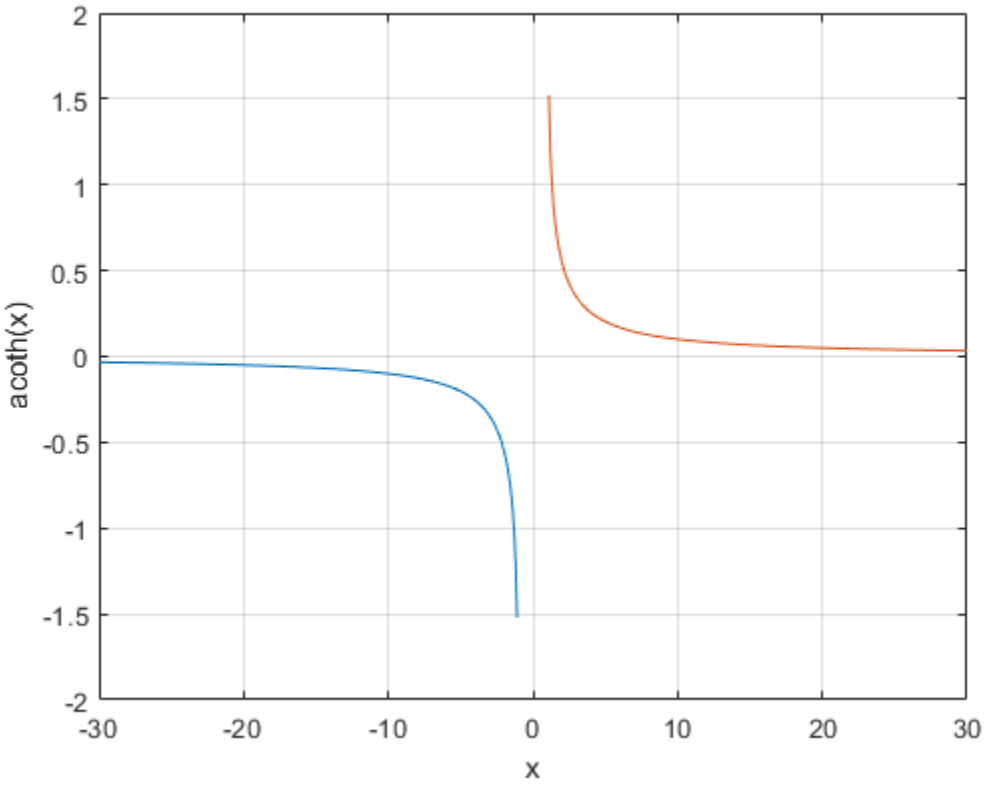
```
X = [2 -3 1+2i];
Y = acoth(X)
```

```
Y = 1×3 complex

   0.5493 + 0.0000i  -0.3466 + 0.0000i   0.1733 - 0.3927i
```

Plot the Inverse Hyperbolic Cotangent Function

Plot the inverse hyperbolic cotangent function over the intervals $-30 \le x < -1$ and $1 < x \le 30$.

```
x1 = -30:0.1:-1.1;
x2 = 1.1:0.1:30;
plot(x1,acoth(x1),x2,acoth(x2))
grid on
xlabel('x')
ylabel('acoth(x)')
```



Input Arguments

X — Hyperbolic cotangent of angle
scalar | vector | matrix | multidimensional array

Hyperbolic cotangent of angle, specified as a scalar, vector, matrix, or multidimensional array. The `acoth` operation is element-wise when `X` is nonscalar.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Inverse Hyperbolic Cotangent

For real values x in the domain $-\infty < x < -1$ and $1 < x < \infty$, the inverse hyperbolic cotangent satisfies

$$\coth^{-1}(x) = \tanh^{-1}\left(\frac{1}{x}\right) = \frac{1}{2}\log\left(\frac{x+1}{x-1}\right).$$

For complex numbers $z = x + iy$ as well as real values in the domain $-1 \le z \le 1$, the call `acoth(z)` returns complex results.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- If the output of the function running on the GPU can be complex, then you must explicitly specify its input arguments as complex. For more information, see “Work with Complex Numbers on a GPU” (Parallel Computing Toolbox).

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`acosh` | `acot` | `asinh` | `atanh` | `coth`

Introduced before R2006a

acsc

Inverse cosecant in radians

Syntax

```
Y = acsc(X)
```

Description

`Y = acsc(X)` returns the "Inverse Cosecant" on page 1-61 ($\csc^{-1}$) of the elements of `X` in radians. The function accepts both real and complex inputs.

- For real values of `X` in the intervals [-∞, -1] and [1, ∞], `acsc(X)` returns real values in the interval [-π/2, π/2].
- For real values of `X` in the interval (-1, 1) and for complex values of `X`, `acsc(X)` returns complex values.

Examples

Inverse Cosecant of a Value

Find the inverse cosecant of a value.

```
acsc(3)
```

```
ans = 0.3398
```

Inverse Cosecant of a Vector of Complex Angles

Find the inverse cosecant of the elements of vector `x`. The `acsc` function acts on `x` element-wise.

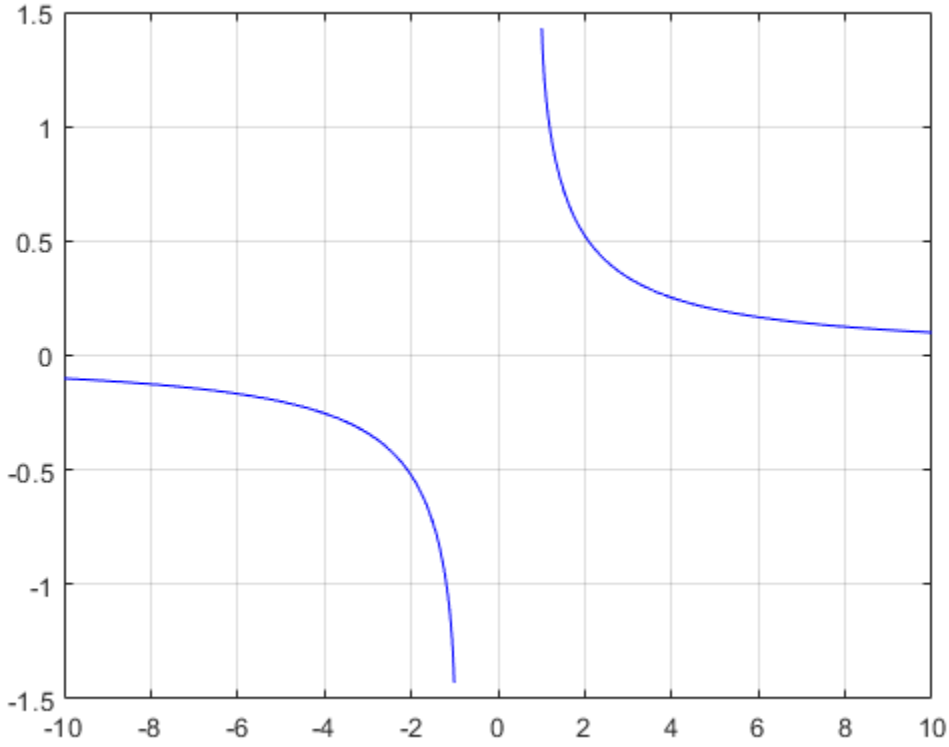
```
x = [0.5i 1+3i -2.2+i];
Y = acsc(x)
```

```
Y = 1×3 complex

   0.0000 - 1.4436i   0.0959 - 0.2970i  -0.3795 - 0.1833i
```

Plot the Inverse Cosecant Function

Plot the inverse cosecant function over the intervals $-10 \le x < -1$ and $1 < x \le 10$.

```
x1 = -10:0.01:-1.01;
x2 = 1.01:0.01:10;
plot(x1,acsc(x1),'b')
hold on
plot(x2,acsc(x2),'b')
grid on
```



Input Arguments

X — Cosecant of angle
scalar | vector | matrix | multidimensional array

Cosecant of angle, specified as a scalar, vector, matrix, or multidimensional array. The `acsc` operation is element-wise when X is nonscalar.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Inverse Cosecant

The inverse cosecant is defined as

$$\csc^{-1}(z) = \sin^{-1}\left(\frac{1}{z}\right).$$

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- If the output of the function running on the GPU can be complex, then you must explicitly specify its input arguments as complex. For more information, see “Work with Complex Numbers on a GPU” (Parallel Computing Toolbox).

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`acscd` | `asec` | `csc` | `cscd`

Introduced in R2006a

acscd

Inverse cosecant in degrees

Syntax

```
Y = acscd(X)
```

Description

`Y = acscd(X)` returns the inverse cosecant (cosec^{-1}) of the elements of `X` in degrees. The function accepts both real and complex inputs.

- For real values of `X` in the intervals [-∞, -1] and [1, ∞], `acscd(X)` returns values in the range [-90, 90].
- For real values of `X` in the interval (-1, 1) and for complex values of `X`, `acscd(X)` returns complex values.

Examples

Inverse Cosecant of Vector

```
x = [20 10 Inf];
y = acscd(x)
```

```
y = 1×3

    2.8660    5.7392         0
```

The `acscd` operation is element-wise when you pass a vector, matrix, or N-D array.

Inverse Cosecant of Complex Value

```
acscd(1+i)
```

```
ans = 25.9136 - 30.4033i
```

Input Arguments

X — Cosecant of angle
scalar value | vector | matrix | N-D array

Cosecant of angle, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `acscd` operation is element-wise when X is non-scalar.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Angle in degrees
scalar value | vector | matrix | N-D array

Angle in degrees, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as X.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`acsc` | `csc` | `cscd`

Introduced before R2006a

acsch

Inverse hyperbolic cosecant

Syntax

```
Y = acsch(X)
```

Description

`Y = acsch(X)` returns the inverse hyperbolic cosecant of the elements of `X`. The function accepts both real and complex inputs. All angles are in radians.

Examples

Inverse Hyperbolic Cosecant of Vector

Find the inverse hyperbolic cosecant of the elements of vector `X`. The `acsch` function acts on `X` element-wise.

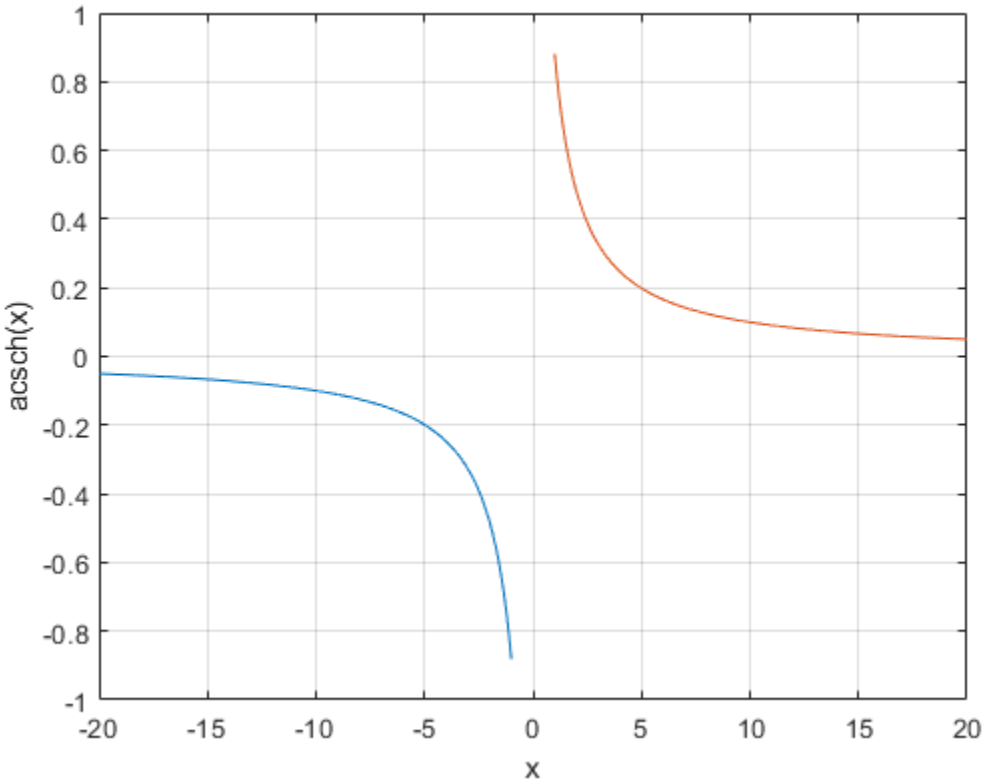
```
X = [2 -3 1+2i];
Y = acsch(X)
```

```
Y = 1×3 complex

   0.4812 + 0.0000i  -0.3275 + 0.0000i   0.2156 - 0.4016i
```

Plot the Inverse Hyperbolic Cosecant Function

Plot the inverse hyperbolic cosecant function over the intervals $-20 \le x \le -1$ and $1 \le x \le 20$.

```
x1 = -20:0.01:-1;
x2 = 1:0.01:20;
plot(x1,acsch(x1),x2,acsch(x2))
grid on
xlabel('x')
ylabel('acsch(x)')
```



Input Arguments

X — Hyperbolic cosecant of angle
scalar | vector | matrix | multidimensional array

Hyperbolic cosecant of angle, specified as a scalar, vector, matrix, or multidimensional array. The `acsch` operation is element-wise when `X` is nonscalar.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Inverse Hyperbolic Cosecant

For real values x in the domain $x < 0$ and $x > 0$, the inverse hyperbolic cosecant satisfies

$$\operatorname{csch}^{-1}(z) = \sinh^{-1}\left(\frac{1}{z}\right) = \log\left(\frac{1}{x} + \sqrt{\frac{1}{x^2} + 1}\right).$$

For complex numbers $z = x + iy$, the call `acsch(z)` returns complex results.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`acosh` | `acsc` | `asinh` | `csch`

Introduced before R2006a

actxcontrol

(To be removed) Create Microsoft ActiveX control in figure window

Note `actxcontrol` will be removed in a future release. For more information, see "Compatibility Considerations".

Syntax

```
c = actxcontrol(progid)
c = actxcontrol(progid,Name,Value)
```

Description

`c = actxcontrol(progid)` creates an ActiveX® control in a figure window. The programmatic identifier (`progid`) for the control determines the type of control created. For the value, see the documentation provided by the control vendor. The returned object `c` is the default interface for the control.

You cannot use an ActiveX server for the `progid` because MATLAB cannot insert ActiveX servers in a figure. For information about using ActiveX servers, see `actxserver`.

`c = actxcontrol(progid,Name,Value)` creates a control using name-value pair arguments.

Input Arguments

`progid` — Programmatic identifier
string | character vector

Programmatic identifier, specified as a string or a character vector. Get the Programmatic identifier from the control or server vendor documentation.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `actxcontrol('progid','position',[0 0 200 200],'parent',gcf)`

`position` — Position vector
integer vector

MATLAB position vector specifying the position of the control in an integer vector, specified as the comma-separated pair consisting of `'position'` and an integer vector. The format is [left, bottom, width, height] using pixel units.

`parent` — Parent figure, model, or Command Window
handle

Parent figure, model, or Command Window, specified as the comma-separated pair consisting of `'parent'` and a handle. This functionality is not supported for figures created with the `uifigure` function.

`callback` — Name of event handler
string | character vector | cell array of character vectors

Name of the event handler, specified as the comma-separated pair consisting of `'callback'` and a string or a character vector, or as a cell array of character vectors. To use the same handler for all events, specify a single name. To handle specific events, specify a cell array of event name/event handler pairs.

`filename` — File name
string | character vector

File name, specified as the comma-separated pair consisting of `'filename'` and a string or a character vector, containing the initial conditions of the previously saved control.

`licensekey` — License key
string | character vector

License key to create licensed ActiveX controls that require design-time licenses, specified as the comma-separated pair consisting of `'licensekey'` and a string or a character vector.

Limitations

- COM functions are available on Microsoft® Windows® systems only.

Compatibility Considerations

`actxcontrol` will be removed

Warns starting in R2019b

`actxcontrol` will be removed.

For app development using App Designer, see “actxcontrol, actxcontrollist, and actxcontrolselect functions will be removed in a future release” release notes.

See Also

`actxserver`

Introduced before R2006a

actxcontrollist

(To be removed) List currently installed Microsoft ActiveX controls

Note `actxcontrollist` will be removed in a future release.

Syntax

```
info = actxcontrollist
```

Description

`info = actxcontrollist` returns a list of controls

Output Arguments

`info` — Information
cell array of character vectors

Information specified as a 1-by-3 cell array of character vectors containing the name, programmatic identifier (ProgID), and file name for the control. Each control has one row, which MATLAB sorts by file name.

Limitations

- COM functions are available on Microsoft Windows systems only.

Compatibility Considerations

`actxcontrollist` will be removed

Warns starting in R2019b

`actxcontrollist` will be removed.

See Also

Introduced before R2006a

actxcontrolselect

(To be removed) Create Microsoft ActiveX control from UI

Note `actxcontrolselect` will be removed in a future release.

Syntax

```
c = actxcontrolselect
[c, info] = actxcontrolselect
```

Description

`c = actxcontrolselect` displays a dialog box listing all ActiveX controls installed on the system and creates the one you select from the list.

`[c, info] = actxcontrolselect` returns information about the control.

Output Arguments

c — COM object
function handle

COM object, returned as a function handle. Use the handle to identify this control when calling MATLAB COM functions.

`info` — Information
cell array of character vectors

Information specified as a 1-by-3 cell array of character vectors containing the name, programmatic identifier (ProgID), and file name for the control.

Limitations

- COM functions are available on Microsoft Windows systems only.

Compatibility Considerations

`actxcontrolselect` will be removed

Warns starting in R2019b

`actxcontrolselect` will be removed.

See Also

Introduced before R2006a

actxGetRunningServer

Handle to running instance of MATLAB Automation server

Syntax

```
c = actxGetRunningServer(progid)
```

Description

`c = actxGetRunningServer(progid)` gets a reference to a running instance of the OLE Automation server. Returns a handle to the default interface of the server.

If the server specified by `progid` is not currently running or if the server object is not registered, then the function returns an error. If multiple instances of the server are running, then the operating system controls the behavior of this function.

Examples

Create List of Excel Property Names

```
c = actxGetRunningServer('Excel.Application');
list = fieldnames(c);
```

Input Arguments

`progid` — Programmatic identifier
string | character vector

Programmatic identifier, specified as a string or a character vector.

The server vendor documentation specifies the ProgID. For MATLAB `progid` values, see “Programmatic Identifiers”.

Example: `'Matlab.Application'`

Output Arguments

c — COM object
function handle

COM object, returned as a function handle.

Limitations

- COM functions are available on Microsoft Windows systems only.

See Also

`actxserver`

Topics

"MATLAB COM Automation Server Interface"

Introduced in R2007a

actxserver

Create COM server

Syntax

```
c = actxserver(progid)
c = actxserver(progid,'machine',machineName)
```

Description

`c = actxserver(progid)` creates a local OLE Automation server, where `progid` is the programmatic identifier (ProgID) of an OLE-compliant COM server. The function returns a handle to the default interface of the server.

For components implemented in a dynamic link library (DLL), `actxserver` creates an in-process server. For components implemented as an executable (EXE), `actxserver` creates an out-of-process server. You can create out-of-process servers either on the client system or on any other system on a network that supports DCOM.

`c = actxserver(progid,'machine',machineName)` creates a server on a remote computer.

Examples

Display Excel `ActivePrinter` Property

Create a Microsoft Excel® object and display the `ActivePrinter` property. MATLAB displays the value for your system.

```
e = actxserver('Excel.Application');
get(e,'ActivePrinter')
```

```
ans = \\printers\Copy-Fl2-South on Ne04:
```

When you are finished with the application, close Excel in MATLAB and delete the server object.

```
Quit(e)
delete(e)
```

Input Arguments

`progid` — Programmatic identifier
string | character vector

Programmatic identifier, specified as a string or a character vector. The server vendor documentation specifies the ProgID. For MATLAB `progid` values, see “Programmatic Identifiers”.

Example: `'Matlab.Application'`

`machineName` — Remote computer
string | character vector

Name of remote computer on which to start the server, specified as a string or a character vector.

Example: `'machine','mymachine'`

Limitations

- COM functions are available on Microsoft Windows systems only.
- 64-bit MATLAB does not support custom interfaces.
- The MATLAB COM Interface does not support invoking functions with optional parameters.

Tips

- You can register events for COM servers. For more information, see “COM Events”.

See Also

`actxGetRunningServer`

Topics

"Write Data to Excel Spreadsheet"
"Read Spreadsheet Data Using Excel as Automation Server"

Introduced before R2006a

add

Package: `matlab.mapreduce`

Add single key-value pair to KeyValueStore

Syntax

```
add(KVStore,key,value)
```

Description

`add(KVStore,key,value)` adds a single key-value pair to `KVStore`, which is a `KeyValueStore` created during `mapreduce` execution. Use `add` in a map or reduce function written for use with `mapreduce` to store intermediate or final key-value pair information.

Examples

Add Key/Value Pairs to KeyValueStore

Use `add` in map and reduce functions to pass data into the intermediate and final `KeyValueStore`. This simple example uses identity map and reduce functions that pass the inputs straight through to the output.

```
inds = datastore('airlinesmall.csv','SelectedVariableNames','ArrDelay','TreatAsMissing
preview(inds)
```

```
ans=8×1 table
    ArrDelay
    ________

        8
        8
       21
```

```
       13
        4
       59
        3
       11
```

```
outds = mapreduce(inds,@myMapper,@myReducer,mapreducer(0));
```

```
*********************************
*      MAPREDUCE PROGRESS       *
*********************************
Map   0% Reduce   0%
Map  16% Reduce   0%
Map  32% Reduce   0%
Map  48% Reduce   0%
Map  65% Reduce   0%
Map  81% Reduce   0%
Map  97% Reduce   0%
Map 100% Reduce   0%
Map 100% Reduce 100%
```

```
readall(outds)
```

```
ans=1×2 table
        Key                Value
    ____________    __________________

    {'ArrDelay'}    {123523x1 double}
```

```
function myMapper(data,info,intermKV)
    add(intermKV, 'ArrDelay',data.ArrDelay);
end

function myReducer(key,intermValIter,outKV)
    data = getnext(intermValIter);
    while hasnext(intermValIter)
        data = [data; getnext(intermValIter)];
    end
    add(outKV,key,data);
end
```

Input Arguments

KVStore — Key-value pair storage object
KeyValueStore object

Key-value pair storage object, specified as a KeyValueStore object. The mapreduce function automatically creates the KeyValueStore object during execution:

- In the map function, the name of the intermediate KeyValueStore object is the third input argument to the map function, myMapper(data, info, intermKVStore). Use that same variable name to add intermediate key-value pairs with add or addmulti in the map function.
- In the reduce function, the name of the final KeyValueStore object is the third input argument to the reduce function, myReducer(intermKey, intermValIter, outKVStore). Use that same variable name to add final key-value pairs with add or addmulti in the reduce function.

For more information, see KeyValueStore.

key — Key
numeric scalar | character vector | string

Key, specified as a numeric scalar, character vector, or string.

All of the keys added by the map function must have the same class. The keys added by the reduce function also must have the same class, but that class can differ from the class of the keys added by the map function.

Numeric keys cannot be NaN, complex, logical, or sparse.

Example: add(intermKVStore,'Sum',sum(X)) adds a key-value pair to an intermediate KeyValueStore object (named intermKVStore) in a map function.

Example: add(outKVStore,'Stats',[mean(X) max(X) min(X) var(X) std(X)]) adds a key-value pair to a final KeyValueStore object (named outKVStore) in a reduce function.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | char | string

value — Value
any MATLAB object

Value, specified as any MATLAB object. This includes all valid MATLAB data types.

The `OutputType` argument of `mapreduce` affects the type of values that the reduce function can add:

- If the `OutputType` is `'Binary'` (the default), then a value added by the reduce function can be any MATLAB object.
- If the `OutputType` is `'TabularText'`, then a value added by the reduce function can be a numeric scalar, character vector, or string scalar when using the `add` function. Additionally, you can use the `addmulti` function to add multiple values with a numeric vector, cell vector of character vectors, cell vector of numeric scalars, or string array. In each case, the numeric values cannot be `NaN`, complex, logical, or sparse.

Note The above key-value pair requirements may differ when using other products with mapreduce. See the documentation for the appropriate product to get product-specific key-value pair requirements.

Example: `add(intermKVStore,'Sum',sum(X))` specifies a single scalar value to pair with a key.

Example: `add(outKVStore,'Stats',[mean(X) max(X) min(X) var(X) std(X)])` specifies a numeric array as the value to pair with a key.

Tips

- Avoid using `add` in a loop, as it can negatively affect `mapreduce` execution time. Instead, use cell arrays to collect multiple values (using vectorized operations if possible) and use a single call to `addmulti`.

See Also

`addmulti`

Topics

`KeyValueStore`

"Build Effective Algorithms with MapReduce"

Introduced in R2014b

addGroup

Package: `matlab.settings`

Add new settings group

Syntax

```
s = addGroup(parentgroup,name)
s = addGroup( ___ ,Name,Value)
```

Description

`s = addGroup(parentgroup,name)` adds a new settings group to the specified parent settings group, and returns a `SettingsGroup` object containing the new settings group. By default, settings groups are not hidden, which means that they display in the parent settings group.

`s = addGroup( ___ ,Name,Value)` specifies settings group properties using one or more name-value pair arguments. For example, `'Hidden',true` adds a hidden settings group. Specify name-value pairs after all other input arguments.

Examples

Add Settings Group

Use the `settings` function to access the root of the settings tree and then create the settings group `mysettings`.

```
s = settings;
newGroup = addGroup(s,'mysettings');
s
```

```
s = 
  SettingsGroup with properties:
```

```
                    matlab: [1×1 SettingsGroup]
                mysettings: [1×1 SettingsGroup]
    mldrivetripwireaccess: [1×1 SettingsGroup]
```

Add Hidden Settings Group

Use the `settings` function to access the root of the settings tree and then create the hidden settings group `myhiddensettings`. Notice that the new group does not appear when you display the parent settings group.

```
s = settings;
addGroup(s,'myhiddensettings','Hidden',true);
s
```

```
s = 
  SettingsGroup with properties:
                    matlab: [1×1 SettingsGroup]
    mldrivetripwireaccess: [1×1 SettingsGroup]
```

Although `myhiddensettings` does not appear in the settings tree, it is accessible. For instance, create the settings group `myveryhiddensettings` inside `myhiddensettings`.

```
addGroup(s.myhiddensettings,'myveryhiddensettings');
s.myhiddensettings
```

```
s = 
  SettingsGroup 'matlab.myhiddensettings' with properties:
    myveryhiddensettings: [1×1 SettingsGroup]
```

Validate Settings in Group

Create a settings group and specify a default validation function. This function validates the values of all settings within the group that do not have their own validation functions defined.

First, create a validation function `numericValidationFcn` that throws an error when the input is not numeric.

```
function numericValidationFcn(x)
    errorMsg = 'Value must be numeric.';
```

```
    assert(isnumeric(x),errorMsg);
end
```

Use the `settings` function to access the root of the settings tree and then create the settings group `mynumericsettings`. Specify the validation function `numericValidationFcn`.

```
s = settings;
newNumericGroup = addGroup(s,'mynumericsettings','ValidationFcn',@numericValidationFcn);
```

Now, test whether the validation function works. Create the setting `MyNonNumericSetting` within the `mynumericsettings` group and set the value of the setting to a non-numeric value. As expected, MATLAB throws an error.

```
addSetting(newvalidatedGroup,'MyNonNumericSetting','PersonalValue','Hello')
```

```
Error using matlab.settings.SettingsGroup/addSettingHelper
Unable to validate settings data. Error using numericValidationFcn (line 3)
Value must be numeric.

Error in matlab.settings.SettingsGroup/addSetting (line 74)
    out = obj.addSettingHelper(results,defaultsUsed);

Error in matlab.settings.SettingsGroup/addSetting (line 74)
    out = obj.addSettingHelper(results,defaultsUsed);
```

Input Arguments

`parentgroup` — Parent settings group

`SettingsGroup` object

Parent settings group to add the group to, specified as a `SettingsGroup` object. Use the `settings` function to access the root settings group object and all the available settings groups.

`name` — Name of settings group to add

character vector | string scalar

Name of settings group to add, specified as a character vector or string scalar. If `name` already exists in the specified parent settings group, MATLAB throws an error.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `addGroup(parentgroup,'myGroup','Hidden',true)` adds a new hidden settings group to the specified parent settings group.

`Hidden` — Hidden state
`false` (default) | `true`

Hidden state, specified as `true` or `false`.

When set to `true`, the settings groups and settings within the group do not display, although they remain accessible.

`ValidationFcn` — Function to validate settings in group
function handle

Function to validate settings in the group, specified as a function handle. When specified, the function is used to validate the values of all settings within the group that do not have their own validation functions defined.

The function handle must be associated with a function that accepts the potential setting value as an input argument, has no output arguments, and throws an error if the validation fails.

The function handle must point to a function on the MATLAB path. Anonymous or nested function handles are not supported.

See Also

`addSetting` | `hasGroup` | `removeGroup` | `settings`

Topics

"Access and Modify Settings"
"Create Custom Settings"

Introduced in R2019b

addSetting

Package: `matlab.settings`

Add new setting

Syntax

```
s = addSetting(parentgroup,name)
s = addSetting( ___ ,Name,Value)
```

Description

`s = addSetting(parentgroup,name)` adds a new setting to the specified parent settings group, and returns a `Setting` object containing the new setting. By default, settings are not hidden, which means that they display in the parent settings group.

`s = addSetting( ___ ,Name,Value)` specifies setting properties using one or more name-value pair arguments. For example, `'PersonalValue',10` adds a new setting with a personal value of `10`. Specify name-value pairs after all other input arguments.

Examples

Add New Setting

Create a settings group and add a new setting to the group. Then, use the value of the setting in your code.

Create the settings group `mysettings`.

```
s = settings;
addGroup(s,'mysettings');
```

Add the setting `MyWorkAddress` to `mysettings` and give it a value.

```
addSetting(s.mysettings,'MyWorkAddress');
s.mysettings.MyWorkAddress.PersonalValue = '3 Apple Hill Drive';
s.mysettings.MyWorkAddress
```

```
ans =
  Setting 'mysettings.MyWorkAddress' with properties:
       ActiveValue: '3 Apple Hill Drive'
     TemporaryValue: <no value>
      PersonalValue: '3 Apple Hill Drive'
       FactoryValue: <no value>
```

Display the value of the setting.

```
fprintf("I work at %s.\n", s.mysettings.MyWorkAddress.ActiveValue)
```

```
I work at 3 Apple Hill Drive.
```

Add Hidden Setting

Use the `settings` function to access the root of the settings tree. Then, create a settings group, add a new hidden setting to the group, and use the value of the setting in your code.

Create the hidden settings group `myhiddensettings`.

```
s = settings;
newHiddenGroup = addGroup(s,'myhiddensettings','Hidden',true);
```

Add the setting `MyHiddenWorkAddress` to `myhiddensettings` and give it a value. Notice that the new setting does not appear when you display the parent settings group.

```
addSetting(newHiddenGroup,'MyHiddenWorkAddress','Hidden',true, ...
    'PersonalValue','1 Lakeside Campus Drive');
s.myhiddensettings
```

```
ans =
  SettingsGroup 'myhiddensettings' with no properties.
```

Display the value of the hidden setting.

```
fprintf("I work at %s.\n", newHiddenGroup.MyHiddenWorkAddress.ActiveValue)
```

```
I work at 1 Lakeside Campus Drive.
```

Add Setting with Validation Function

Create a setting and specify a function to validate its value.

First, create a validation function `numericValidationFcn` that throws an error when the input is not numeric.

```
function numericValidationFcn(x)
    errorMsg = 'Value must be numeric.';
    assert(isnumeric(x),errorMsg);
end
```

Create the settings group `mysettings`.

```
s = settings;
addGroup(s,'mysettings');
```

Add the setting `MyNumericSetting` to `mysettings` and specify the validation function `numericValidationFcn`.

```
addSetting(s.mysettings,'MyNumericSetting','ValidationFcn',@numericValidationFcn);
```

Now, test whether the validation function works. Set the value of `MyNumericSetting` to a non-numeric value. As expected, MATLAB throws an error.

```
s.mysettings.MyNumericSetting.PersonalValue = 'Hello';
```

```
Unable to validate settings data. Error using numericValidationFcn (line 3)
Value must be numeric.
```

Input Arguments

`parentgroup` — Parent settings group
`SettingsGroup` object

Parent settings group to add setting to, specified as a `SettingsGroup` object. Use the `settings` function to access the root settings group object and all the available settings groups.

`name` — Name of setting to add
character vector | string scalar

Name of setting to add, specified as a character vector or string scalar. If `name` already exists in the specified settings group, MATLAB throws an error.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `addSetting(a,'mySetting','PersonalValue',10,'Hidden',true)` adds a new hidden setting with a personal value of `10` to the specified parent settings group.

`PersonalValue` — Personal value of setting
MATLAB data

Personal value of setting, specified as MATLAB data of any type except for handle types. Data containers such as cell arrays, structs, and objects that include handles are also not supported. This argument is required when creating read-only settings.

`Hidden` — Hidden state
`false` (default) | `true`

Hidden state, specified as `true` or `false`.

When set to `true`, the settings groups and settings within the group do not display, although they remain accessible.

`ReadOnly` — Read-only state
`false` (default) | `true`

Read-only state, specified as `true` or `false`. When `true`, the personal or temporary value of the setting cannot be changed. The `PersonalValue` argument is required when creating read-only settings.

`ValidationFcn` — Function to validate setting
function handle

Function to validate setting, specified as a function handle. When specified, the function is used to validate the value of the setting.

The function handle must be associated with a function that accepts the potential setting value as an input argument, has no output arguments, and throws an error if the validation fails.

The function handle must point to a function on the MATLAB path. Anonymous or nested function handles are not supported.

See Also

`addGroup` | `hasSetting` | `removeSetting` | `settings`

Topics

"Access and Modify Settings"
"Create Custom Settings"

Introduced in R2019b

matlab.apputil.create

Create or modify app project file for packaging app into `.mlappinstall` file using interactive dialog box

Syntax

```
matlab.apputil.create
matlab.apputil.create(prjfile)
```

Description

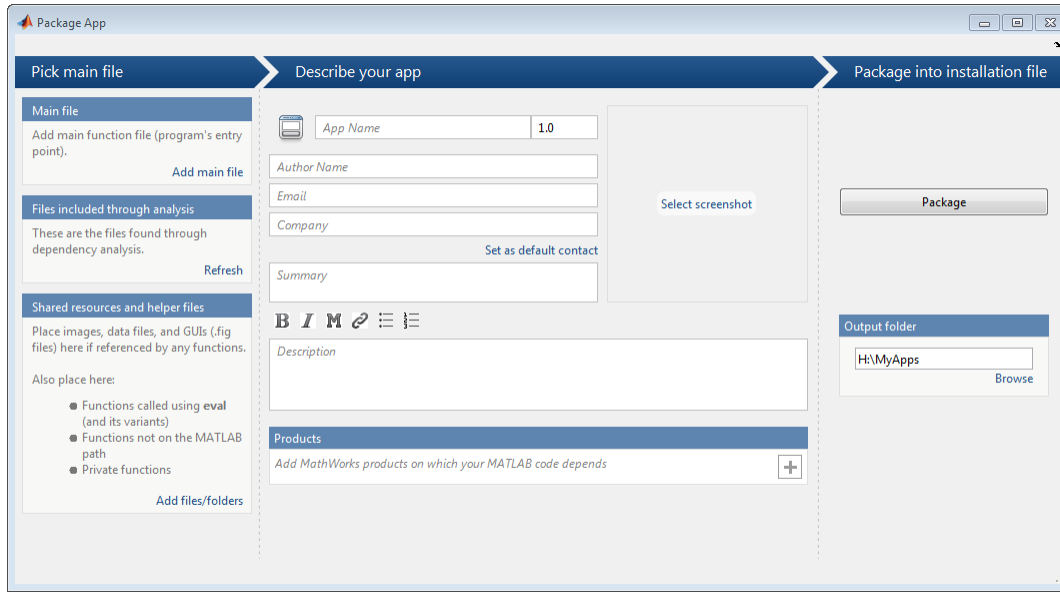
`matlab.apputil.create` opens the Package App dialog box that steps you through the process of creating an `.mlappinstall` file.

`matlab.apputil.create(prjfile)` loads the specified `.prj` file and populates the Package App dialog box with the information from the specified project file. Use this option if you need to update an existing app.

Examples

Open Dialog Box for Creating an App Package

```
matlab.apputil.create
```



Minimally, add a main file, specify an app name, and indicate the required products. MATLAB creates and continuously saves a `.prj` file, regardless of whether you click **Package**. However, MATLAB does not create a `.mlappinstall` file if you do not click **Package**.

Update Existing App Package

Assume you have an existing project file, `myapp.prj`. You want to add a file and update the description.

Open the Package App dialog box, specifying the previously created `.prj` file:

```
matlab.apputil.create('myapp.prj')
```

The dialog box opens populated with the data you previously specified for `myapp`. Adjust the information in the dialog box, as needed.

Input Arguments

`prjfile` — Full or partial path to the `.prj` file
character vector | string scalar

Full or partial path to the `.prj` file you created previously with the Package App dialog box, specified as a character vector or string scalar.

Example: `'C:\myapp.prj'`

Example: `"C:\myapp.prj"`

See Also

`matlab.apputil.package`

Topics

"MATLAB App Installer File — mlappinstall"

Introduced in R2012b

matlab.apputil.getInstalledAppInfo

List installed app information

Syntax

```
matlab.apputil.getInstalledAppInfo
```

```
appinfo = matlab.apputil.getInstalledAppInfo
```

Description

`matlab.apputil.getInstalledAppInfo` displays the ID and name of all installed custom apps. It does not display this information for apps packaged with MathWorks® products.

`appinfo = matlab.apputil.getInstalledAppInfo` returns structure to `appinfo`, which includes the status, ID, location, and name of all installed custom apps. It does not return this information for apps packaged with MathWorks products.

Examples

Display Installed Apps Information in the Command Window

Assume you installed two apps, `LinePlotter` and `PlotRandNumbers`. Display the app information in the Command Window.

```
matlab.apputil.getInstalledAppInfo
```

```
ID                    Name
------------------    ---------------
LinePlotterAPP        LinePlotter
PlotRandNumbersAPP    PlotRandNumbers
```

Store Installed App Information in a Variable

Assume you installed an app, `ColorPalette`. Get the app information and store it in a variable, `myappinfo`.

```
myappinfo = matlab.apputil.getInstalledAppInfo;
```

Store Installed App Information in a Variable and Display IDs

Assume you installed two apps, `LinePlotter` and `PlotRandNumbers`. Get and store the app information for both installed apps in a variable, `myappinfo`. Then, get the `id` for each app.

```
myappinfo = matlab.apputil.getInstalledAppInfo
```

```
myappinfo =

1x2 struct array with fields:
    id
    name
    status
    location
```

Get the id of each installed app:

```
appids={myappinfo.id}
```

```
appids =

    'LinePlotterAPP'    'PlotRandNumbersAPP'
```

Output Arguments

`appinfo` — Information about installed apps
structure array

Information about the installed app, returned as a structure array, with one element for each installed app. Each element of the structure array has the following fields:

`status` — Installation status
`'installed'`

Status of the installation, returned as `'installed'`.

`id` — Unique identifier for the installed app
character vector

Unique identifier for the installed app, returned as a character vector.

The ID is for use when running or uninstalling the app programmatically.

`location` — Folder where the app is installed
character vector

Folder where the app is installed, returned as a character vector.

`name` — Name of the installed app
character vector

Name of the installed app as it appears in the apps gallery, returned as a character vector.

See Also

`matlab.apputil.install` | `matlab.apputil.run` | `matlab.apputil.uninstall`

Topics

"MATLAB App Installer File — mlappinstall"

Introduced in R2012b

matlab.apputil.install

Install app from a `.mlappinstall` file

Syntax

```
appinfo = matlab.apputil.install(appfile)
```

Description

`appinfo = matlab.apputil.install(appfile)` installs the specified app file and returns information about the app.

Examples

Install App and Display Information About the Installation

Assume you have downloaded an app from File Exchange named `EmployeeData`. Install it and return information about the installation to the variable `appinfo`. Later, if you decide to deinstall the app programmatically, you have the app id required to do so.

```
appinfo = matlab.apputil.install...
   ('C:\myguis\myapps\EmployeeData.mlappinstall')
```

```
appinfo = 

          id: 'EmployeeDataApp'
        name: 'EmployeeData'
```

```
      status: 'installed'
    location: 'C:\myguis\myapps\EmployeeData.mlappinstall'
```

Input Arguments

`appfile` — Full or partial path to `.mlappinstall` file
character vector | string scalar

Full or partial path of the app file you want to install, specified as a character vector or string scalar.

Example: `'C:\myguis\myapps\myapp.mlappinstall'`

Example: `"C:\myguis\myapps\myapp.mlappinstall"`

Output Arguments

`appinfo` — Information about installed app
structure

Information about the installed app, returned as a structure with the fields:

`status` — Installation status
`'installed'` | `'updated'`

Installation status, returned as one of the following:

- `'installed'` — New app is installed.
- `'updated'` — Previously installed app is updated.

`id` — Unique identifier
character vector

Unique identifier for the installed app, returned as a character vector.

The ID is for use when running or uninstalling the app programmatically.

`location` — Folder where app is installed
character vector

Folder where app is installed, returned as a character vector.

`name` — Name of installed app
character vector

Name of installed app as it appears in the apps gallery, returned as a character vector.

See Also

`matlab.apputil.getInstalledAppInfo` | `matlab.apputil.package` | `matlab.apputil.uninstall`

Topics

"MATLAB App Installer File — mlappinstall"

Introduced in R2012b

matlab.apputil.package

Package app files into `.mlappinstall` file

Syntax

```
matlab.apputil.package(prjfile)
```

Description

`matlab.apputil.package(prjfile)` creates a `.mlappinstall` file based on the information in the specified `prjfile`.

Examples

Create mlappinstall File for Previously Created Project File

Assume you previously created `myprjfile.prj` using `matlab.apputil.create`. The following command creates the corresponding `.mlappinstall` file.

```
matlab.apputil.package('myprjfile.prj')
```

Input Arguments

`prjfile` — Full or partial path to app project (`.prj`) file
character vector | string scalar

Full or partial path to app project (`.prj`) file, specified as a character vector or string scalar.

Example: `'plotdata.prj'`

Example: `"plotdata.prj"`

Tips

- To create a `.prj` file, use `matlab.apputil.create`.

See Also

`matlab.apputil.create` | `matlab.apputil.install` | `matlab.apputil.run`

Introduced in R2012b

matlab.apputil.run

Run app programmatically

Syntax

```
matlab.apputil.run(appid)
```

Description

`matlab.apputil.run(appid)` runs the custom app specified by the unique identifier, `appid`.

Examples

Run Previously Installed App

Assume you installed two apps, `PlotData` and `setslider`. Run `PlotData` programmatically, using its ID.

Get IDs of all installed apps.

```
matlab.apputil.getInstalledAppInfo
```

```
ID                      Name
-------------------     ----------------
setsliderAPP            setslider
PlotDataAPP             PlotData
```

Run `PlotData`.

```
matlab.apputil.run('PlotDataAPP')
```

Input Arguments

appid — ID of custom app
character vector | string scalar

ID of custom app you want to run, specified as a character vector or string scalar.

Example: `'DataExplorationAPP'`

Example: `"DataExplorationAPP"`

Tips

- The ID of a custom app is returned when you install it. You can use `matlab.apputil.getInstalledAppInfo` to get the ID after you have installed an app.
- When a custom app runs, MATLAB adds any folders it needs to have added to the path, as identified when the app was packaged. When the app exits, MATLAB removes those folders from the path.
- You can run multiple, different custom apps concurrently. However, you cannot run two instances of the same app concurrently.

See Also

`matlab.apputil.create` | `matlab.apputil.getInstalledAppInfo` | `matlab.apputil.install`

Introduced in R2012b

matlab.apputil.uninstall

Uninstall app

Syntax

```
matlab.apputil.uninstall(appid)
```

Description

`matlab.apputil.uninstall(appid)` removes the app specified by the unique identifier, `appid`. MATLAB removes all files corresponding to the app and removes the app from the app gallery.

Examples

Uninstall App

Assume you previously installed two apps, `setslider` and `simplegui`. Get the IDs of all installed apps, and then use the ID for `simplegui` to uninstall it.

View the IDs of all apps

```
matlab.apputil.getInstalledAppInfo
```

```
ID                      Name
--------------------    ----------------
setsliderAPP            setslider
simpleguiAPP            simplegui
```

Uninstall the `simplegui` app.

```
matlab.apputil.uninstall('simpleguiAPP')
```

Confirm the app was removed, by running `matlab.apputil.getInstalledAppInfo` again.

```
matlab.apputil.getInstalledAppInfo

ID                      Name
-------------------     ----------------
setsliderAPP            setslider
```

Input Arguments

appid — ID of app
character vector | string scalar

ID of app to be uninstalled, specified as a character vector or string scalar.

Example: `'DataExplorationAPP'`

Example: `"DataExplorationAPP"`

Tips

- To determine the appid of an installed app, preserve the value returned when you install the app programmatically with `matlab.apputil.install`, or use `matlab.apputil.getInstalledAppInfo`.

See Also

`matlab.apputil.getInstalledAppInfo` | `matlab.apputil.install`

Introduced in R2012b

addevent

Add event to `timeseries`

Syntax

```
tsout = addevent(tsin,tsevent)
tsout = addevent(tsin,eventname,eventtime)
```

Description

`tsout = addevent(tsin,tsevent)` adds a `tsdata.event` object `tsevent` to the `timeseries` object `tsin`. The event `tsevent` can be a single `tsdata.event` object or an array of `tsdata.event` objects.

`tsout = addevent(tsin,eventname,eventtime)` creates a `tsdata.event` object and adds it to the `Events` property of `tsin`. The argument `eventname` can be a single event name character vector or a cell array of event name character vectors. `eventtime` is the corresponding event time or a cell array of event times.

Examples

Add Event

Create a `timeseries` object and add an event to it.

```
tsin = timeseries((1:5)');
tsout = addevent(tsin,'Event1',0);
tsout.Events
```

```
    EventData: []
         Name: 'Event1'
         Time: 0
        Units: 'seconds'
    StartDate: ''
```

Input Arguments

`tsin` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

`tsevent` — Event
scalar | array

Event, specified as a scalar `tsdata.event` object or an array of `tsdata.event` objects.

`eventname` — Event name
character vector | cell array

Event name, specified as a character vector containing the name of a `tsdata.event` object or a cell array containing multiple event names.

Data Types: `char` | `cell`

`eventtime` — Event time
scalar | cell array

Event time, specified as a scalar time or a cell array of times corresponding to the events in `eventname`.

See Also

`delevent` | `timeseries` | `tsdata.event`

Introduced before R2006a

addGroup

Package: `matlab.settings`

Add new factory settings group

Syntax

```
s = addGroup(parentgroup,name)
s = addGroup( ___ ,Name,Value)
```

Description

`s = addGroup(parentgroup,name)` adds the factory settings group `name` to the specified parent factory settings group and returns the new group as a `FactoryGroup` object. By default, factory settings groups are hidden, which means that they do not display in the parent settings group.

`s = addGroup( ___ ,Name,Value)` specifies the factory group properties using one or more name-value pair arguments. For example, `'Hidden',false` adds a group that is visible in the factory settings tree. Specify name-value pairs after all other input arguments.

Examples

Add Factory Settings Group

Create the root factory group for the toolbox `mytoolbox` and then add a group to the tree.

Create the root factory group `mytoolbox`.

```
myToolboxFactoryTree = matlab.settings.FactoryGroup.createToolboxGroup('mytoolbox', ...
    'Hidden',false);
```

Add the `font` group to the tree.

```
toolboxFontGroup = addGroup(myToolboxFactoryTree,'font','Hidden',false);
```

Validate Settings in Group

Add a settings group and specify a default validation function. This function validates the values of all settings within the group that do not specify their own validation functions. This includes settings in subgroups, as long as the subgroup or settings do not specify their own validation functions.

First, create a validation function `numericValidationFcn` that throws an error when the input is not numeric.

```
function numericValidationFcn(x)
    errorMsg = 'Value must be numeric.';
    assert(isnumeric(x),errorMsg);
end
```

Create the root factory group `mytoolbox` and then add the group `mynumericsettings` to the tree. Specify the validation function `numericValidationFcn`.

```
myToolboxFactoryTree = matlab.settings.FactoryGroup.createToolboxGroup('mytoolbox', ...
    'Hidden',false);
toolboxFontGroup = addGroup(myToolboxFactoryTree,'mynumericsettings','Hidden',false, ...
    'ValidationFcn',@numericValidationFcn);
```

MATLAB throws an error whenever a setting within the group is set to a nonnumeric value.

Input Arguments

`parentgroup` — Parent factory group
`FactoryGroup` object

Parent factory group to add the group to, specified as a `FactoryGroup` object. Use the `matlab.settings.FactoryGroup.createToolboxGroup` function to create the root factory group object.

Example: `addGroup(s.mytoolbox,'newGroup')`

`name` — Name of factory group to add
character vector | string

Name of factory group to add, specified as a character vector or string. If the factory group `name` exists in the specified parent factory group, MATLAB displays an error.

Example: `addGroup(s.mytoolbox,'newGroup')`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `addGroup(parentGroup,'newGroup','Hidden',false)` creates a visible factory group.

`Hidden` — Hidden state
`true` (default) | `false`

Hidden state, specified as `true` or `false`.

When set to `true`, the factory group, including all factory groups and factory settings within the group do not display, although they remain accessible.

`ValidationFcn` — Function to validate settings in group
function handle

Function to validate factory settings in group, specified as a function handle. When specified, the function is used to validate the values of all factory settings within the group that do not specify their own validation functions. This includes settings in subgroups, as long as the subgroup or settings do not specify their own validation functions.

The function handle must be associated with a function that accepts the potential setting value as an input argument, has no output arguments, and throws an error if the validation fails.

The function handle must point to a function on the MATLAB path. Anonymous or nested function handles are not supported.

See Also

`addSetting` | `matlab.settings.FactoryGroup.createToolboxGroup`

Topics

"Create Factory Settings for Toolboxes"
"Access and Modify Settings"

Introduced in R2019b

addmulti

Package: `matlab.mapreduce`

Add multiple key-value pairs to KeyValueStore

Syntax

```
addmulti(KVStore,keys,values)
```

Description

`addmulti(KVStore,keys,values)` adds multiple key-value pairs to `KVStore`, which is a `KeyValueStore` created during `mapreduce` execution. Use `addmulti` in a map or reduce function written for use with `mapreduce` to store intermediate or final key-value pair information.

Examples

Add Multiple Key/Value Pairs to KeyValueStore

Use `add` and `addmulti` in map and reduce functions to pass data into the intermediate and final `KeyValueStore`. This simple example uses identity map and reduce functions that pass the inputs straight through to the output.

```
inds = datastore('airlinesmall.csv','SelectedVariableNames',...
    {'ArrDelay','DepDelay'},'TreatAsMissing','NA');
preview(inds)
```

```
ans=8×2 table
    ArrDelay    DepDelay
    ________    ________

        8          12
        8           1
```

```
       21          20
       13          12
        4          -1
       59          63
        3          -2
       11          -1
```

```
outds = mapreduce(inds,@myMapper,@myReducer,mapreducer(0));
```

```
********************************
*      MAPREDUCE PROGRESS      *
********************************
Map   0% Reduce   0%
Map  16% Reduce   0%
Map  32% Reduce   0%
Map  48% Reduce   0%
Map  65% Reduce   0%
Map  81% Reduce   0%
Map  97% Reduce   0%
Map 100% Reduce   0%
Map 100% Reduce  50%
Map 100% Reduce 100%
```

```
readall(outds)
```

```
ans=2×2 table
        Key                Value
    ____________    __________________

    {'ArrDelay'}    {123523x1 double}
    {'DepDelay'}    {123523x1 double}
```

```
function myMapper(data,info,intermKV)
    addmulti(intermKV,{'ArrDelay' 'DepDelay'},{data.ArrDelay data.DepDelay});
end

function myReducer(key,intermValIter,outKV)
    data = getnext(intermValIter);
    while hasnext(intermValIter)
        data = [data; getnext(intermValIter)];
    end
    add(outKV,key,data);
end
```

Input Arguments

KVStore — Key-value pair storage object
KeyValueStore object

Key-value pair storage object, specified as a KeyValueStore object. The mapreduce function automatically creates the KeyValueStore object during execution:

- In the map function, the name of the intermediate KeyValueStore object is the third input argument to the map function, myMapper(data, info, intermKVStore). Use that same variable name to add intermediate key-value pairs with add or addmulti in the map function.
- In the reduce function, the name of the final KeyValueStore object is the third input argument to the reduce function, myReducer(intermKey, intermValIter, outKVStore). Use that same variable name to add final key-value pairs with add or addmulti in the reduce function.

For more information, see KeyValueStore.

keys — Keys
numeric scalar | numeric vector | character vector | string array | cell vector of character vectors | cell vector of numeric scalars

Keys, specified as a numeric scalar, numeric vector, character vector, string array, cell vector of character vectors, or cell vector of numeric scalars. If the keys are a numeric vector, cell vector, or string array, then each entry specifies a different key.

All of the keys added by the map function must have the same class. The keys added by the reduce function must also have the same class, but that class can differ from the class of the keys added by the map function.

Numeric keys cannot be NaN, complex, logical, or sparse.

Example: addmulti(intermKVStore,{'Sum'; 'Count'; 'Variance'},{sum(X); numel(X); var(X)}) adds three key-value pairs to an intermediate KeyValueStore object (named intermKVStore) using a cell vector to specify the keys.

Example: addmulti(intermKVStore,[1 2 3 4],{sum(X); mean(X); max(X); min(X)}) adds four key-value pairs to an intermediate KeyValueStore object using a numeric vector to specify the keys.

Example: `addmulti(outKVStore,'Stats',{[mean(X) max(X) min(X) var(X) std(X)]})` adds a single key-value pair to a final `KeyValueStore` object (named `outKVStore`) using a character vector as the key.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`values` — Values
cell array

Values, specified as a cell array. Each entry in the cell array specifies the value in a key-value pair, so `numel(values)` must be equal to the number of keys. The entries in the cell array can be any MATLAB object, including all valid MATLAB data types.

The `OutputType` argument of `mapreduce` affects the type of values that the reduce function can add:

- If the `OutputType` is `'Binary'` (the default), then a value added by the reduce function can be any MATLAB object.
- If the `OutputType` is `'TabularText'`, then a value added by the reduce function can be a numeric scalar, character vector, or string scalar when using the `add` function. Additionally, you can use the `addmulti` function to add multiple values with a numeric vector, cell vector of character vectors, cell vector of numeric scalars, or string array. In each case, the numeric values cannot be `NaN`, complex, logical, or sparse.

Note The above key-value pair requirements may differ when using other products with mapreduce. See the documentation for the appropriate product to get product-specific key-value pair requirements.

Example: `addmulti(intermKVStore,{'Sum'; 'Count'; 'Variance'},{sum(X); numel(X); var(X)})` adds three key-value pairs to an intermediate `KeyValueStore` object named `intermKVStore`.

Example: `addmulti(intermKVStore,[1 2 3 4],{sum(X); mean(X); max(X); min(X)})` adds four key-value pairs to an intermediate `KeyValueStore` object using a cell vector.

Example: `addmulti(outKVStore,'Stats',{[mean(X) max(X) min(X) var(X) std(X)]})` adds a single key-value pair to a final `KeyValueStore` object named `outKVStore`.

Example: `addmulti(outKVStore,{'Distance' 'Time'},{table.Distance table.Time})` adds two key-value pairs using variables in a table to specify the values.

Tips

- Avoid using `add` in a loop, as it can negatively affect `mapreduce` execution time. Instead, use cell arrays to collect multiple values (using vectorized operations if possible) and use a single call to `addmulti`.

See Also

`add`

Topics

`KeyValueStore`
"Build Effective Algorithms with MapReduce"

Introduced in R2014b

addSetting

Package: `matlab.settings`

Add new factory setting

Syntax

```
s = addSetting(parentgroup,name)
s = addSetting( ___ ,Name,Value)
```

Description

`s = addSetting(parentgroup,name)` adds the factory setting `name` to the specified parent factory group and returns the new setting as a `FactorySetting`. By default, factory settings are hidden, which means that they do not display in the parent settings group.

`s = addSetting( ___ ,Name,Value)` specifies the factory setting properties using one or more name-value pair arguments. For example, `'Hidden',false` creates a factory setting that is visible in the factory settings tree. Specify name-value pairs after all other input arguments.

Examples

Add New Factory Setting

Create the root factory group for the toolbox `mytoolbox` and then add a new setting to the tree.

Create the root factory group `mytoolbox`.

```
myToolboxFactoryTree = matlab.settings.FactoryGroup.createToolboxGroup('mytoolbox', ...
    'Hidden',false);
```

Add the setting `FontSize` and give it a value.

```
fontSizeSetting = addSetting(myToolboxFactoryTree,'FontSize','FactoryValue',11,'Hidden',false);
```

Add Factory Setting with Validation Function

Create a setting and specify a function to validate its value.

First, create a validation function `numericValidationFcn` that throws an error when the input is not numeric.

```
function numericValidationFcn(x)
    errorMsg = 'Value must be numeric.';
    assert(isnumeric(x),errorMsg);
end
```

Create the root factory group `mytoolbox`.

```
myToolboxFactoryTree = matlab.settings.FactoryGroup.createToolboxGroup('mytoolbox', ...
    'Hidden',false);
```

Add the setting `FontSize` and specify the validation function `numericValidationFcn`.

```
addSetting(myToolboxFactoryTree,'FontSize','FactoryValue',11,'Hidden',false, ...
    'ValidationFcn',@numericValidationFcn);
```

MATLAB throws an error whenever the setting is set to a nonnumeric value.

Input Arguments

`parentgroup` — Parent factory group
`FactoryGroup` object

Parent factory group to add the setting to, specified as a `FactoryGroup` object. Use the `matlab.settings.FactoryGroup.createToolboxGroup` function to create the root factory group object.

Example: `addSetting(s.mytoolbox,'newSetting')`

`name` — Name of factory setting to add
character vector | string

Name of factory setting to add, specified as a character vector or string. If the factory setting `name` exists in the specified parent group, MATLAB displays an error.

Example: `addGroup(parentGroup,'newSetting')`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `addSetting(parentGroup,'newGroup','Hidden',false)` creates a visible factory group.

`FactoryValue` — Factory value of setting
MATLAB data

Factory value of setting, specified as MATLAB data of any type except for handle types. Data containers such as cell arrays, structs, and objects that include handles are also not supported. You must specify either a factory value or a factory value function for the setting, but not both.

`FactoryValueFcn` — Function to set the setting factory value
function handle

Function to set the setting factory value, specified as a function handle. The factory value of the setting is set to the output of the specified function.

The function handle must point to a function on the MATLAB path. Anonymous or nested function handles are not supported.

You can specify either a factory value or a factory value function for the setting, but not both.

`Hidden` — Hidden state
`true` (default) | `false`

Hidden state, specified as `true` or `false`.

When set to `true`, the factory setting does not display, although it remains accessible.

`ReadOnly` — Read-only state
`false` (default) | `true`

Read-only state, specified as `true` or `false`. When `true`, the personal or temporary value of the setting cannot be specified.

`ValidationFcn` — Function to validate setting
function handle

Function to validate setting, specified as a function handle. When specified, the function is used to validate the value of the factory setting.

The function handle must be associated with a function that accepts the potential setting value as an input argument, has no output arguments, and throws an error if the validation fails.

The function handle must point to a function on the MATLAB path. Anonymous or nested function handles are not supported.

See Also

`addGroup` | `matlab.settings.FactoryGroup.createToolboxGroup`

Topics

"Create Factory Settings for Toolboxes"
"Access and Modify Settings"

Introduced in R2019b

addStyle

Package: `matlab.ui.control`

Add style to table UI component

Syntax

```
addStyle(uit,s)
addStyle(uit,s,target,targetIndex)
```

Description

`addStyle(uit,s)` adds a style created with the `uistyle` function to the specified table UI component. The style is applied to the whole table. The table must be parented to a figure created with the `uifigure` function, or to one of its child containers.

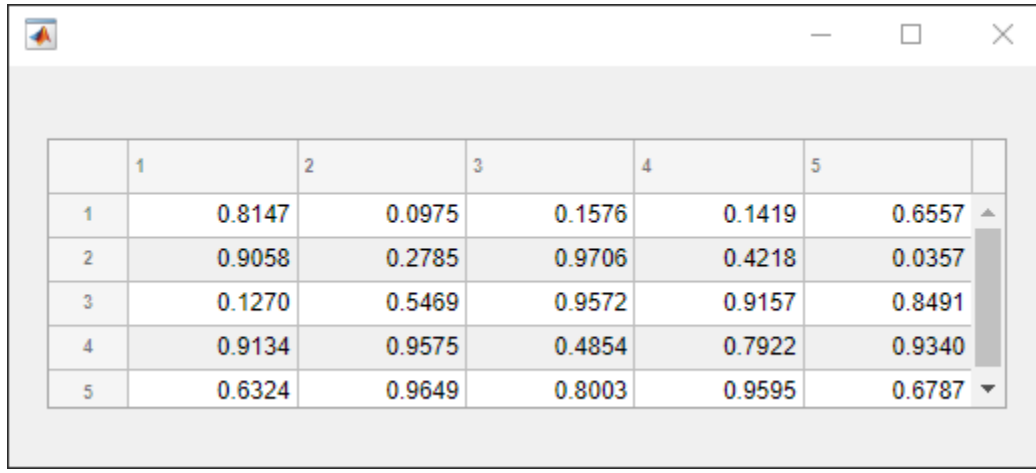
`addStyle(uit,s,target,targetIndex)` adds the style to a specific row, column, or cell. For example, `addStyle(uit,s,'column',3)` adds the style to the third column of the specified table.

Examples

Add Style to Entire Table

Create a table UI component in a figure.

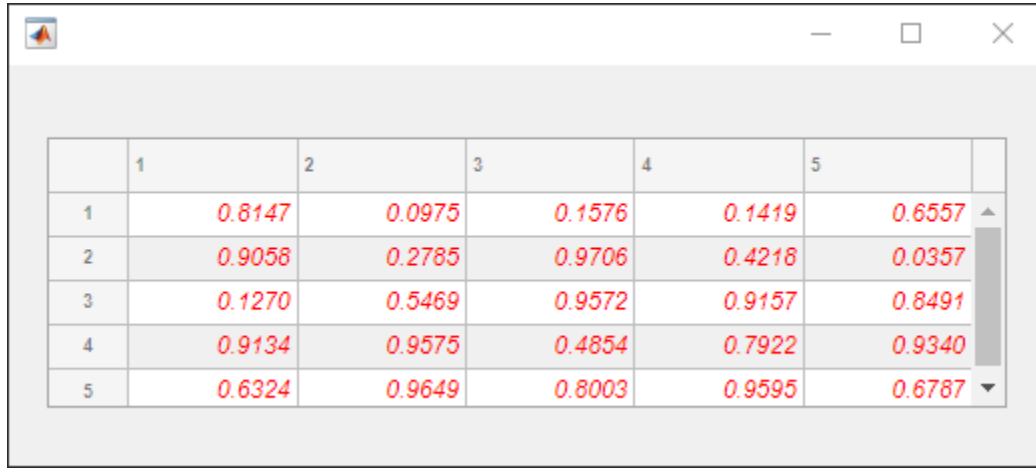
```
fig = uifigure;
fig.Position = [500 500 520 200];

uit = uitable(fig);
uit.Data = rand(5);
uit.Position = [20 30 480 135];
```



	1	2	3	4	5
1	0.8147	0.0975	0.1576	0.1419	0.6557
2	0.9058	0.2785	0.9706	0.4218	0.0357
3	0.1270	0.5469	0.9572	0.9157	0.8491
4	0.9134	0.9575	0.4854	0.7922	0.9340
5	0.6324	0.9649	0.8003	0.9595	0.6787

Then, create a red italic font style using the `uistyle` function. Add it to the whole table using the `addStyle` function.

```
s = uistyle('FontAngle','italic','FontColor','r');
addStyle(uit,s)
```



	1	2	3	4	5
1	*0.8147*	*0.0975*	*0.1576*	*0.1419*	*0.6557*
2	*0.9058*	*0.2785*	*0.9706*	*0.4218*	*0.0357*
3	*0.1270*	*0.5469*	*0.9572*	*0.9157*	*0.8491*
4	*0.9134*	*0.9575*	*0.4854*	*0.7922*	*0.9340*
5	*0.6324*	*0.9649*	*0.8003*	*0.9595*	*0.6787*

Add Style to Specific Cells

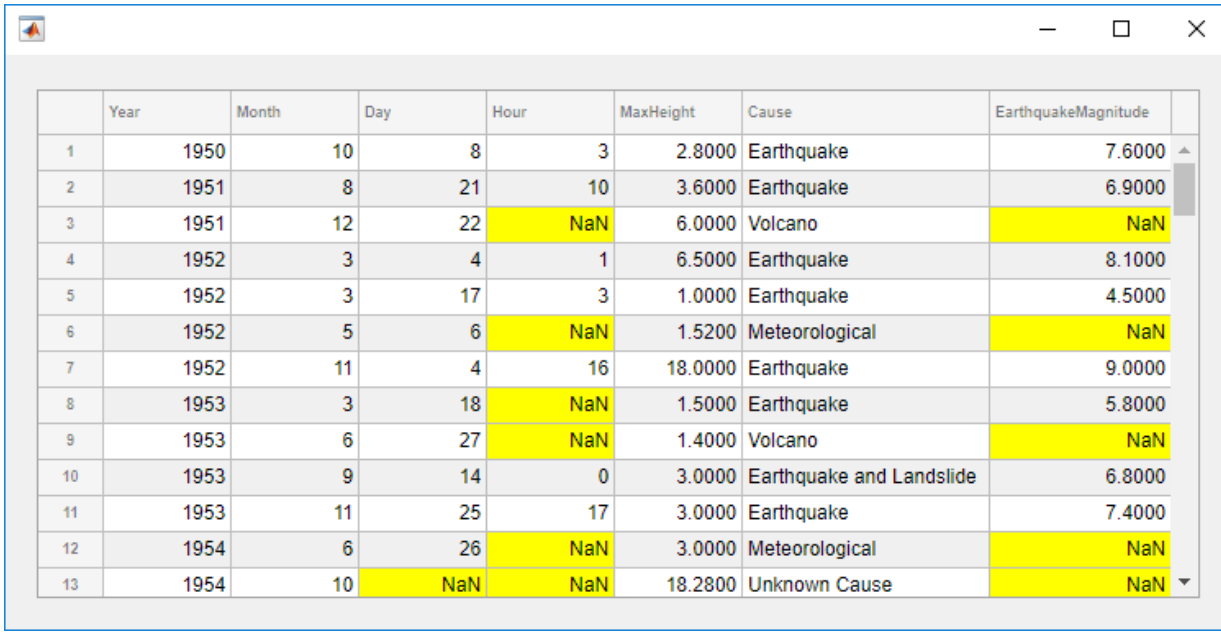
Style cells in a table UI component that contain missing values. In this case, add a yellow background color style to cells that have `NaN` values.

Read tsunami sample data into the workspace as a table array. Then create a table UI component to display the data.

```
tdata = readtable('tsunamis.xlsx');
vars = {'Year','Month','Day','Hour','MaxHeight','Cause','EarthquakeMagnitude'};
tdata = tdata(1:100,vars);

fig = uifigure('Position',[500 500 750 350]);
uit = uitable(fig);
uit.Position = [20 20 710 310];
uit.Data = tdata;
uit.RowName = 'numbered';
```

Use the `ismissing` function to get a logical array of the table elements that contain missing values. Find the row and column subscripts for the elements that have `NaN` values. Finally, create a yellow background color style and add it to the cells with `NaN` values in the table UI component.

```
styleIndices = ismissing(tdata);
[row,col] = find(styleIndices);

s = uistyle('BackgroundColor','yellow');
addStyle(uit,s,'cell',[row,col]);
```



	Year	Month	Day	Hour	MaxHeight	Cause	EarthquakeMagnitude
1	1950	10	8	3	2.8000	Earthquake	7.6000
2	1951	8	21	10	3.6000	Earthquake	6.9000
3	1951	12	22	NaN	6.0000	Volcano	NaN
4	1952	3	4	1	6.5000	Earthquake	8.1000
5	1952	3	17	3	1.0000	Earthquake	4.5000
6	1952	5	6	NaN	1.5200	Meteorological	NaN
7	1952	11	4	16	18.0000	Earthquake	9.0000
8	1953	3	18	NaN	1.5000	Earthquake	5.8000
9	1953	6	27	NaN	1.4000	Volcano	NaN
10	1953	9	14	0	3.0000	Earthquake and Landslide	6.8000
11	1953	11	25	17	3.0000	Earthquake	7.4000
12	1954	6	26	NaN	3.0000	Meteorological	NaN
13	1954	10	NaN	NaN	18.2800	Unknown Cause	NaN

Emphasize Different Parts of Table Using Multiple Styles

Create multiple styles and add them to different parts of a table UI component.

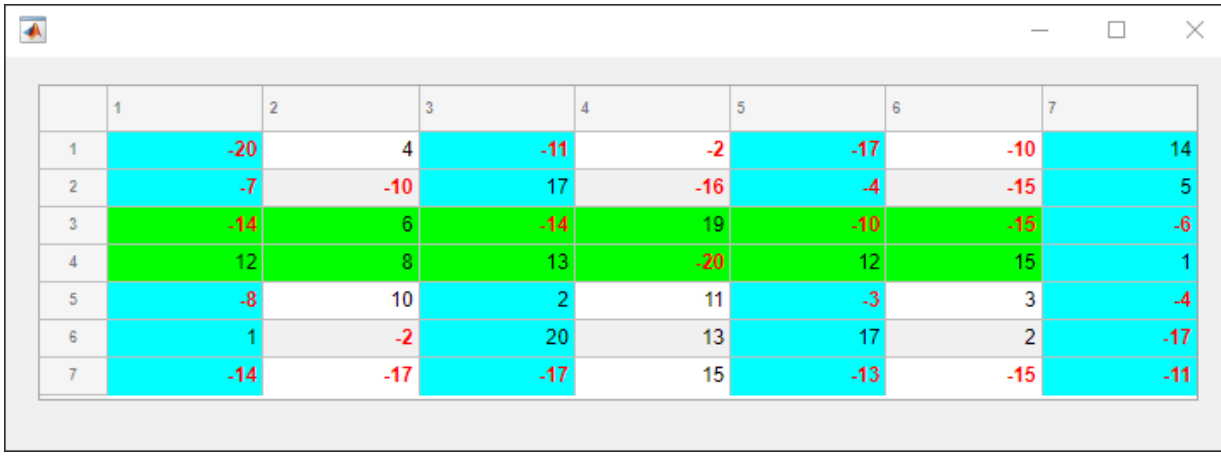
Create a figure with a table UI component in it and display numeric data in the table. Find the row and column subscripts for elements in the table with a value less than zero so you can style these cells later.

```
fig = uifigure;
fig.Position = [500 500 720 230];

uit = uitable(fig);
uit.Data = randi([-20,20],7);
uit.Position = [20 30 680 185];

[row,col] = find(uit.Data < 0);
```

Create two background color styles and one style that specifies font color and weight. Add a cyan background color to columns 1, 3, and 5. Emphasize the cells with negative

values by making their font red and bold. Then, style rows 3 and 4 with a green background color. Finally, reuse the cyan background color style and add it to column 7. For cells where multiple styles of the same type are added, the style that is added last is the one that displays in the cell.

```
s1 = uistyle;
s1.BackgroundColor = 'cyan';
addStyle(uit,s1,'column',[1 3 5])

s2 = uistyle;
s2.FontColor = 'red';
s2.FontWeight = 'bold';
addStyle(uit,s2,'cell',[row,col])

s3 = uistyle;
s3.BackgroundColor = 'green';
addStyle(uit,s3,'row',[3 4])

addStyle(uit,s1,'column',7)
```



	1	2	3	4	5	6	7
1	-20	4	-11	-2	-17	-10	14
2	-7	-10	17	-16	-4	-15	5
3	-14	6	-14	19	-10	-15	-6
4	12	8	13	-20	12	15	1
5	-8	10	2	11	-3	3	-4
6	1	-2	20	13	17	2	-17
7	-14	-17	-17	15	-13	-15	-11

Input Arguments

`uit` — Table component
`Table` object

Table component, specified as a `Table` object created with the `uitable` function. The `Table` object must be parented to a figure created with the `uifigure` function, or one of its child containers.

`s` — Style object
`Style` object

Style object created with the `uistyle` function.

`target` — Style target
`'table'` (default) | `'row'` | `'column'` | `'cell'`

Style target, specified as `'row'`, `'column'`, `'cell'`, or `'table'`. Use this argument to indicate the type of table part to apply the style to.

Example: `addStyle(uit,s,'column',5)` adds the style to the fifth column of `uit`.

Example: `addStyle(uit,s,'cell',[3 7])` adds the style to the cell in row 3, column 7.

`targetIndex` — Style target index
`''` (default) | positive integer | vector of positive integers | n-by-2 array of positive integers | ...

Style target index, specified as one of the values listed in the tables. The types of values that you can specify depend on the target and the type of data used in the table UI component.

Target	Supported Values	Examples	Result
`'row'`	Positive integer.	4	Adds a style to the row with the corresponding row index.
	Vector of positive integers	`[3 8 9 12]`	Adds a style to the rows with the corresponding row indices.

Target	Supported Values	Examples	Result
`'column'`	Positive integer.	3	Adds a style to the column with the corresponding column index.
	Vector of positive integers.	`[1 2 7 14]`	Adds a style to the columns with the corresponding column indices.
`'cell'`	n-by-2 array of positive integers.	`[2 4;5 9;13 27]`	Adds a style to the cells with the corresponding row and column subscripts.
`'table'`	Empty character vector.	`''`	Adds a style to the entire table.

In table UI components in which the underlying data is a `table` array, there are additional options when the target is specified as `'column'`. Specify these values for the `targetIndex` if you want to apply a style to a column based on the variable names in the table array.

Target	Supported Values	Examples
`'column'`	String scalar. Adds a style to the column with the corresponding variable name.	`"Torque"`
	String array. Adds a style to the columns with the corresponding variable names.	`["Torque" "Mass"]`

Target	Supported Values	Examples
	Character vector. Adds a style to the column with the corresponding variable name.	`'Revenue'`
	1-D cell array of character vectors Adds a style to the columns with the corresponding variable names.	`{'Year','Expenses','Revenue'}`

If you are styling cells based on whether the value of the cells meets a specific condition, and your cells are editable, then use a `CellEditCallback` function to recompute the style target indices that meet the specified condition, and add a new style to the table that sets these new cells as the `targetIndex`.

Tips

- To see a list of the styles that have been added to a table, query the value of the `StyleConfigurations` property.

See Also

Functions
`removeStyle` | `uistyle` | `uitable`

Properties
Style Properties | Table Properties

Introduced in R2019b

audioinfo

Information about audio file

Syntax

```
info = audioinfo(filename)
```

Description

`info = audioinfo(filename)` returns information about the contents of the audio file specified by `filename`.

Examples

Get Information About Audio File

Create a WAVE file from the example file `handel.mat`, and get information about the file.

Create a WAVE (.wav) file in the current folder.

```
load handel.mat
filename = 'C:\Temp\handel.wav';
audiowrite(filename,y,Fs);
clear y Fs
```

Use `audioinfo` to return information about the WAVE file.

```
info = audioinfo(filename)
```

```
info =

  struct with fields:

             Filename: 'C:\Temp\handel.wav'
```

```
    CompressionMethod: 'Uncompressed'
          NumChannels: 1
           SampleRate: 8192
         TotalSamples: 73113
             Duration: 8.9249
                Title: []
              Comment: []
               Artist: []
        BitsPerSample: 16
```

Input Arguments

`filename` — Name of file
character vector | string scalar

Name of file, specified as a character vector or string scalar that includes the file extension. If a path is specified, it can be absolute, relative, or partial.

Example: `'myFile.mp3'`

Example: `'../myFile.mp3'`

Example: `'C:\temp\myFile.mp3'`

`audioinfo` supports the following file formats.

Platform Support	File Format
All platforms	WAVE (`.wav`)
	OGG (`.ogg`)
	FLAC (`.flac`)
	AU (`.au`)
	AIFF (`.aiff`, `.aif`)
	AIFC (`.aifc`)
Windows 7 (or later), Macintosh, and Linux®	MP3 (`.mp3`)
	MPEG-4 AAC (`.m4a`, `.mp4`)

On Windows 7 platforms (or later), `audioinfo` might also return information about the contents of any files supported by Windows Media® Foundation.

On Linux platforms, `audioinfo` might also return information about the contents of any files supported by GStreamer.

`audioinfo` can extract audio metadata from MPEG-4 (`.mp4`, `.m4v`) video files on Windows 7 or later, macOS 10.7 Lion or higher, and Linux, and from Windows Media Video (`.wmv`) and AVI (`.avi`) files on Windows 7 (or later) and Linux platforms.

Data Types: `char` | `string`

Output Arguments

`info` — Information about audio file
structure

Information about audio file, returned as a structure. `info` can contain the following fields.

Field Name	Description	Data Type
`Filename`	Filename including the absolute path to the file and the file extension.	`character vector`
`CompressionMethod`	Compression method used.	`character vector`
`NumChannels`	Number of audio channels encoded in the audio file.	`double`
`SampleRate`	Sample rate of the audio data in the file, in hertz.	`double`
`TotalSamples`	Total number of audio samples in the file.	`double`
`Duration`	Duration of the file, in seconds.	`double`
`BitsPerSample`	Number of bits per sample encoded in the audio file. Only valid for WAVE (`.wav`) and FLAC (`.flac`) files.	`double`

Field Name	Description	Data Type
`BitRate`	Number of kilobits per second (kbit/s) used for compressed audio files. Only valid for MP3 (`.mp3`) and MPEG-4 Audio (`.m4a`, `.mp4`) files.	`double`
`Title`	Value of `'Title'`, if any.	`character vector`
`Artist`	Value of `'Artist'`, if any.	`character vector`
`Comment`	Value of `'Comment'`, if any.	`character vector`

Note The `BitRate` property returns the actual bit rate on Mac platforms, and not the encoded bit rate. This means that bit rate values might be lower than specified at the time of the encoding, depending on the source data.

Note On Mac platforms, `audioinfo` returns metadata from `.m4a` and `.mp4` files only on macOS 10.7 Lion or higher. Previous versions of macOS will not read the `'Title'`, `'Author'`, or `'Comment'` fields.

Limitations

- For MP3 and MPEG-4 AAC audio files on Windows 7 or later and Linux platforms, `audioinfo` might report fewer samples than expected. On Linux platforms, this is due to a limitation in the underlying GStreamer framework.
- On Linux platforms, `audioinfo` interprets single channel data in MPEG-4 AAC files as stereo data.

See Also

`audioread` | `audiowrite`

Introduced in R2012b

audioread

Read audio file

Syntax

```
[y,Fs] = audioread(filename)
[y,Fs] = audioread(filename,samples)

[y,Fs] = audioread( ___ ,dataType)
```

Description

`[y,Fs] = audioread(filename)` reads data from the file named `filename`, and returns sampled data, `y`, and a sample rate for that data, `Fs`.

`[y,Fs] = audioread(filename,samples)` reads the selected range of audio samples in the file, where `samples` is a vector of the form `[start,finish]`.

`[y,Fs] = audioread( ___ ,dataType)` returns sampled data in the data range corresponding to the `dataType` of `'native'` or `'double'`, and can include any of the input arguments in previous syntaxes.

Examples

Read Complete Audio File

Create a WAVE file from the example file `handel.mat`, and read the file back into MATLAB®.

Create a WAVE (`.wav`) file in the current folder.

```
load handel.mat

filename = 'handel.wav';
```

```
audiowrite(filename,y,Fs);
clear y Fs
```

Read the data back into MATLAB using `audioread`.

```
[y,Fs] = audioread('handel.wav');
```

Play the audio.

```
sound(y,Fs);
```

Read Portion of Audio File

Create a FLAC file from the example file `handel.mat`, and then read only the first 2 seconds.

Create a FLAC (`.flac`) file in the current folder.

```
load handel.mat

filename = 'handel.flac';
audiowrite(filename,y,Fs);
```

Read only the first 2 seconds.

```
samples = [1,2*Fs];
clear y Fs
[y,Fs] = audioread(filename,samples);
```

Play the samples.

```
sound(y,Fs);
```

Return Audio in Native Integer Format

Create a `.flac` file, read the first 2 seconds of the file and then return audio in the native integer format.

Create a FLAC (`.flac`) file in the current folder.

```
load handel.mat
filename = 'handel.flac';
audiowrite(filename,y,Fs);
```

Read only the first 2 seconds and specify the data and view the datatype of the sampled data y. The data type of `y` is `double`.

```
samples = [1,2*Fs];
clear y Fs
[y,Fs] = audioread(filename,samples);
whos y
```

```
  Name          Size             Bytes  Class     Attributes

  y         16384x1             131072  double
```

Request audio data in the native format of the file, and then view the data type of the sampled data `y`. Note the new data type of `y`.

```
[y,Fs] = audioread(filename,'native');
whos y
```

```
  Name          Size             Bytes  Class    Attributes

  y         73113x1             146226  int16
```

Input Arguments

`filename` — Name of file to read
character vector | string scalar

Name of file to read, specified as a character vector or string scalar that includes the file extension. If a path is specified, it can be absolute, relative or partial.

Example: `'myFile.mp3'`

Example: `'../myFile.mp3'`

Example: `'C:\temp\myFile.mp3'`

`audioread` supports the following file formats.

Platform Support	File Format
All platforms	WAVE (`.wav`)
	OGG (`.ogg`)
	FLAC (`.flac`)
	AU (`.au`)
	AIFF (`.aiff`, `.aif`)
	AIFC (`.aifc`)
Windows 7 (or later), Macintosh, and Linux	MP3 (`.mp3`)
	MPEG-4 AAC (`.m4a`, `.mp4`)

On Windows platforms prior to Windows 7, `audioread` does not read WAVE files with MP3 encoded data.

On Windows 7 (or later) platforms, `audioread` might also read any files supported by Windows Media Foundation.

On Linux platforms, `audioread` might also read any files supported by GStreamer.

`audioread` can extract audio from MPEG-4 (`.mp4`, `.m4v`) video files on Windows 7 or later, Macintosh, and Linux, and from Windows Media Video (`.wmv`) and AVI (`.avi`) files on Windows 7 (or later) and Linux platforms.

Data Types: `char` | `string`

`samples` — Audio samples to read
`[1,inf]` (default) | two-element vector of positive scalar integers

Audio samples to read, specified as a two-element vector of the form `[start,finish]`, where `start` and `finish` are the first and last samples to read, and are positive scalar integers.

- `start` must be less than or equal to `finish`.
- `start` and `finish` must be less than the number of audio samples in the file,
- You can use `inf` to indicate the last sample in the file.

Note When reading a portion of some MP3 files on Windows 7 platforms, `audioread` might read a shifted range of samples. This is due to a limitation in the underlying Windows Media Foundation framework.

When reading a portion of MP3 and M4A files on Linux platforms, `audioread` might read a shifted range of samples. This is due to a limitation in the underlying GStreamer framework.

Example: `[1,100]`

Data Types: `double`

`dataType` — Data format of audio data, `y`
`'double'` (default) | `'native'`

Data format of audio data,`y`, specified as one of the following:

`'double'`	Double-precision normalized samples.
`'native'`	Samples in the native format found in the file.

For compressed audio formats, such as MP3 and MPEG-4 AAC that do not store data in integer form, `'native'` defaults to `'single'`.

Data Types: `char` | `string`

Output Arguments

`y` — Audio data
matrix

Audio data in the file, returned as an `m`-by-`n` matrix, where `m` is the number of audio samples read and `n` is the number of audio channels in the file.

- If you do not specify `dataType`, or `dataType` is `'double'`, then `y` is of type `double`, and matrix elements are normalized values between −1.0 and 1.0.
- If `dataType` is `'native'`, then `y` can be one of several MATLAB data types, depending on the file format and the `BitsPerSample` value of the input file. Call `audioinfo` to determine the `BitsPerSample` value of the file.

File Format	BitsPerSample	Data Type of y	Data Range of y
WAVE (`.wav`)	8	`uint8`	$0 \le y \le 255$

File Format	BitsPerSample	Data Type of y	Data Range of y
	16	`int16`	-32768 ≤ y ≤ +32767
	24	`int32`	-2^31 ≤ y ≤ 2^31–1
	32	`int32`	-2^31 ≤ y ≤ 2^31–1
	32	`single`	-1.0 ≤ y ≤ +1.0
	64	`double`	-1.0 ≤ y ≤ +1.0
WAVE (`.wav`) (u-law)	8	`int16`	-32124 ≤ y ≤ +32124
WAVE (`.wav`) (A-law)	8	`int16`	-32256 ≤ y ≤ +32256
FLAC (`.flac`)	8	`uint8`	0 ≤ y ≤ 255
	16	`int16`	-32768 ≤ y ≤ +32767
	24	`int32`	-2^31 ≤ y ≤ 2^31–1
MP3 (`.mp3`), MPEG-4 AAC (`.m4a`, `.mp4`), OGG (`.ogg`), and certain compressed WAVE files	N/A	`single`	-1.0 ≤ y ≤ +1.0

Note Where `y` is `single` or `double` and the `BitsPerSample` is 32 or 64, values in `y` might exceed −1.0 or +1.0.

Fs — Sample rate

positive scalar

Sample rate, in hertz, of audio data `y`, returned as a positive scalar.

Limitations

- For MP3, MPEG-4 AAC, and AVI audio files on Windows 7 or later and Linux platforms, `audioread` might read fewer samples than expected. On Windows 7 platforms, this is due to a limitation in the underlying Media Foundation framework. On Linux platforms, this is due to a limitation in the underlying GStreamer framework. If you require sample-accurate reading, work with WAV or FLAC files.
- On Linux platforms, `audioread` reads MPEG-4 AAC files that contain single-channel data as stereo data.

See Also

`audioinfo` | `audiowrite`

Topics

"Read and Write Audio Files"

Introduced in R2012b

audiowrite

Write audio file

Syntax

```
audiowrite(filename,y,Fs)
audiowrite(filename,y,Fs,Name,Value)
```

Description

`audiowrite(filename,y,Fs)` writes a matrix of audio data, `y`, with sample rate `Fs` to a file called `filename`. The `filename` input also specifies the output file format. The output data type on page 1-150 depends on the output file format and the data type of the audio data, `y`.

`audiowrite(filename,y,Fs,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Examples

Write an Audio File

Create a WAVE file from the example file `handel.mat`, and read the file back into MATLAB®.

Write a WAVE (`.wav`) file in the current folder.

```
load handel.mat

filename = 'handel.wav';
audiowrite(filename,y,Fs);
clear y Fs
```

Read the data back into MATLAB using `audioread`.

```
[y,Fs] = audioread(filename);
```

Listen to the audio.

```
sound(y,Fs);
```

Specify Bits Per Sample and Metadata

Create a FLAC file from the example file `handel.mat` and specify the number of output bits per sample and a comment.

```
load handel.mat

filename = 'handel.flac';
audiowrite(filename,y,Fs,'BitsPerSample',24,...
'Comment','This is my new audio file.');
clear y Fs
```

View information about the new FLAC file by using the `audioinfo` function

```
info = audioinfo(filename) ;
```

The `info` structure contains the following information fields: `Filename`, `CompressionMethod`, `NumChannels`, `SampleRate`, `TotalSamples`, `Duration`, `Title`, `Comment`, `Artist`, and `BitsPerSample`.

Input Arguments

`filename` — Name of file to write
character vector | string scalar

Name of file to write, or the full path to the file, specified as a character vector or string scalar that includes the file extension. If a path is specified, it can be absolute or relative. If you do not specify the path, then the destination directory is the current working directory.

`audiowrite` supports the following file formats.

Platform Support	File Format
All platforms	WAVE (`.wav`)
	OGG (`.ogg`)
	FLAC (`.flac`)
Windows and Mac	MPEG-4 AAC (`.m4a`, `.mp4`)

Example: `'myFile.m4a'`

Example: `'../myFile.m4a'`

Example: `'C:\temp\myFile.m4a'`

When writing AAC files on Windows, `audiowrite` pads the front and back of the output signal with extra samples of silence. The Windows AAC encoder also places a very sharp fade-in and fade-out on the audio. This results in audio with an increased number of samples after being written to disk.

Data Types: `char` | `string`

y — Audio data to write
real matrix

Audio data to write, specified as an `m`-by-`n` real matrix, where `m` is the number of audio samples to write and `n` is the number of audio channels to write.

If either `m` or `n` is 1, then `audiowrite` assumes that this dimension specifies the number of audio channels, and the other dimension specifies the number of audio samples.

The maximum number of channels depends on the file format.

File Format	Maximum Number of Channels
WAVE (`.wav`)	1024
OGG (`.ogg`)	255
FLAC (`.flac`)	8
MPEG-4 AAC (`.m4a`, `.mp4`)	2

The valid range for the data in `y` depends on the data type of `y`.

Data Type of y	Valid Range for y
uint8	0 ≤ y ≤ 255
int16	-32768 ≤ y ≤ +32767
int32	-2^31 ≤ y ≤ 2^31–1
single	-1.0 ≤ y ≤ +1.0
double	-1.0 ≤ y ≤ +1.0

Data beyond the valid range is clipped.

If `y` is `single` or `double`, then audio data in `y` should be normalized to values in the range −1.0 and 1.0, inclusive.

Data Types: `single` | `double` | `int16` | `int32` | `uint8`

Fs — Sample rate
positive integer scalar

Sample rate, in hertz, of audio data `y`, specified as a positive integer scalar greater than 0. When writing to `.m4a` or `.mp4` files on Windows platforms, `audiowrite` supports only samples rates of 44100 and 48000.

Example: `44100`

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Title','Symphony No. 9','Artist','My Orchestra'` instructs `audiowrite` to write an audio file with the title “Symphony No. 9” and the artist information “My Orchestra.”

`BitsPerSample` — Number of output bits per sample
16 (default) | 8 | 24 | 32 | 64

Number of output bits per sample, specified as the comma-separated pair consisting of `'BitsPerSample'` and an integer.

Only available for WAVE (`.wav`) and FLAC (`.flac`) files. For FLAC files, only 8, 16, or 24 bits per sample are supported.

Example: `'BitsPerSample',32`

`BitRate` — Kilobits per second (kbit/s)
128 (default) | 64 | 96 | 160 | 192 | 256 | 320

Number of kilobits per second (kbit/s) used for compressed audio files, specified as the comma-separated pair consisting of `'BitRate'` and an integer. On Windows 7 or later, the only valid values are 96, 128, 160, and 192.

In general, a larger `BitRate` value results in higher compression quality.

Only available for MPEG-4 (`.m4a`, `.mp4`) files.

Example: `'BitRate',96`

`Quality` — Quality setting for the Ogg Vorbis Compressor
75 (default) | value in the range [0 100]

Quality setting for the Ogg Vorbis Compressor, specified as the comma-separated pair consisting of `'Quality'` and a number in the range [0 100], where 0 is lower quality and higher compression, and 100 is higher quality and lower compression.

Only available for OGG (`.ogg`) files.

Example: `'Quality',25`

`Title` — Title information
`[]` (default) | character vector | string scalar

Title information, specified as the comma-separated pair consisting of `'Title'` and a character vector or string scalar.

Data Types: `char` | `string`

`Artist` — Artist information
`[]` (default) | character vector | string scalar

Artist information, specified as the comma-separated pair consisting of `'Artist'` and a character vector or string scalar.

Data Types: `char` | `string`

`Comment` — Additional information
[ ] (default) | character vector | string scalar

Additional information, specified as the comma-separated pair consisting of `'Comment'` and a character vector or string scalar.

Data Types: `char` | `string`

Note On Mac platforms, `audiowrite` writes metadata to WAVE, OGG, and FLAC files only, and will not write the `'Title'`, `'Author'`, or `'Comment'` fields to MPEG-4 AAC files.

Algorithms

The output data type is determined by the file format, the data type of `y`, and the specified output `BitsPerSample`.

File Formats	Data Type of y	Output BitsPerSample	Output Data Type
WAVE (`.wav`),	`uint8`, `int16`, `int32`, `single`, `double`	8	`uint8`
		16	`int16`
		24	`int32`
	`uint8`, `int16`, `int32`	32	`int32`
	`single`, `double`	32	`single`
	`single`, `double`	64	`double`
FLAC (`.flac`)	`uint8`, `int16`, `int32`, `single`, `double`	8	`int8`
		16	`int16`
		24	`int32`
MPEG-4 (`.m4a`, `.mp4`), OGG (`.ogg`)	`uint8`, `int16`, `int32`, `single`, `double`	N/A	`single`

See Also

`audioinfo` | `audioread`

Topics

"Read and Write Audio Files"

Introduced in R2012b

autumn

Autumn colormap array

Syntax

```
c = autumn
c = autumn(m)
```

Description

`c = autumn` returns the autumn colormap as a three-column array with the same number of rows as the colormap for the current figure. If no figure exists, then the number of rows is equal to the default length of 256. Each row in the array contains the red, green, and blue intensities for a specific color. The intensities are in the range [0,1], and the color scheme looks like this image.

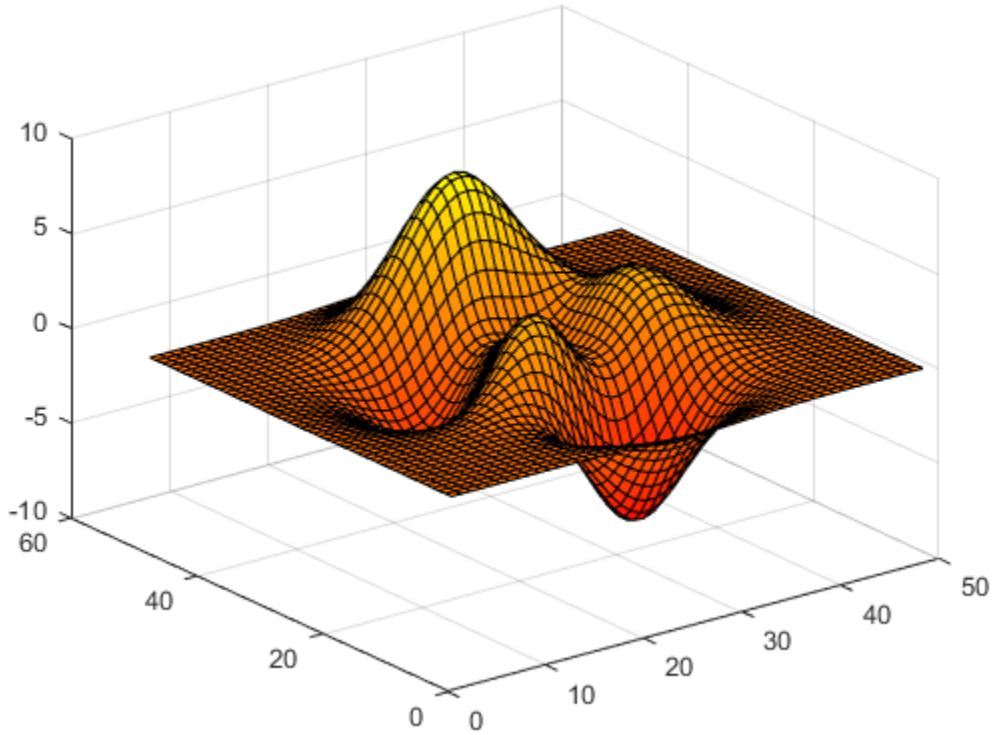
`c = autumn(m)` returns the colormap with `m` colors.

Examples

Reverse the Colormap

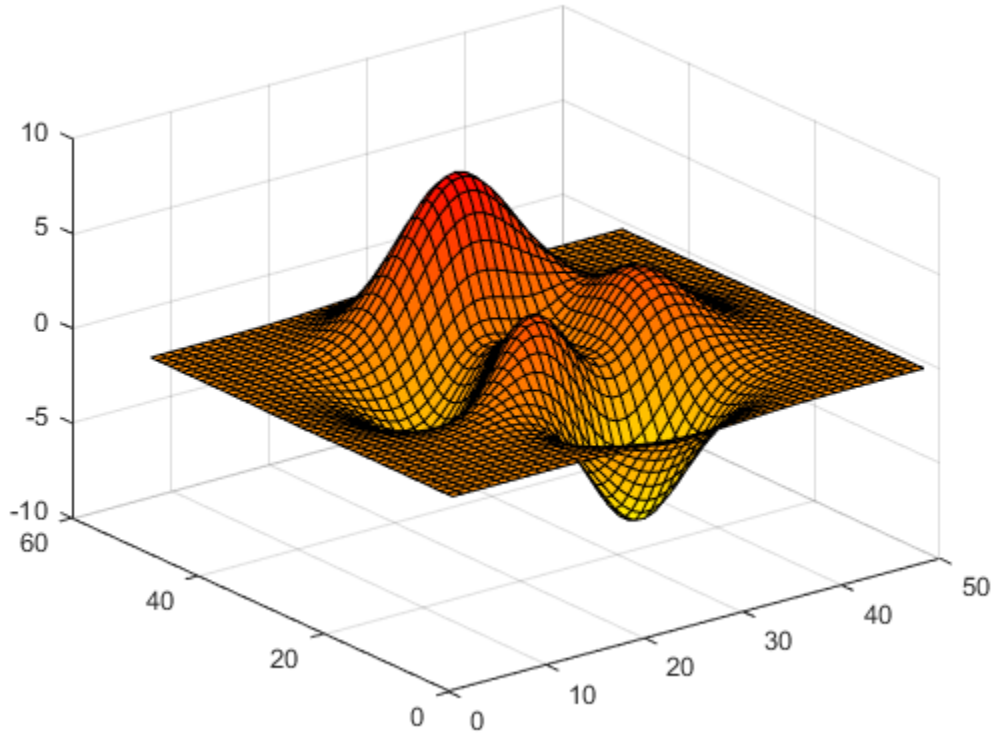
Plot a surface and assign the autumn colormap.

```
surf(peaks);
colormap('autumn');
```



Get the autumn colormap array and reverse the order. Then apply the modified colormap to the surface.

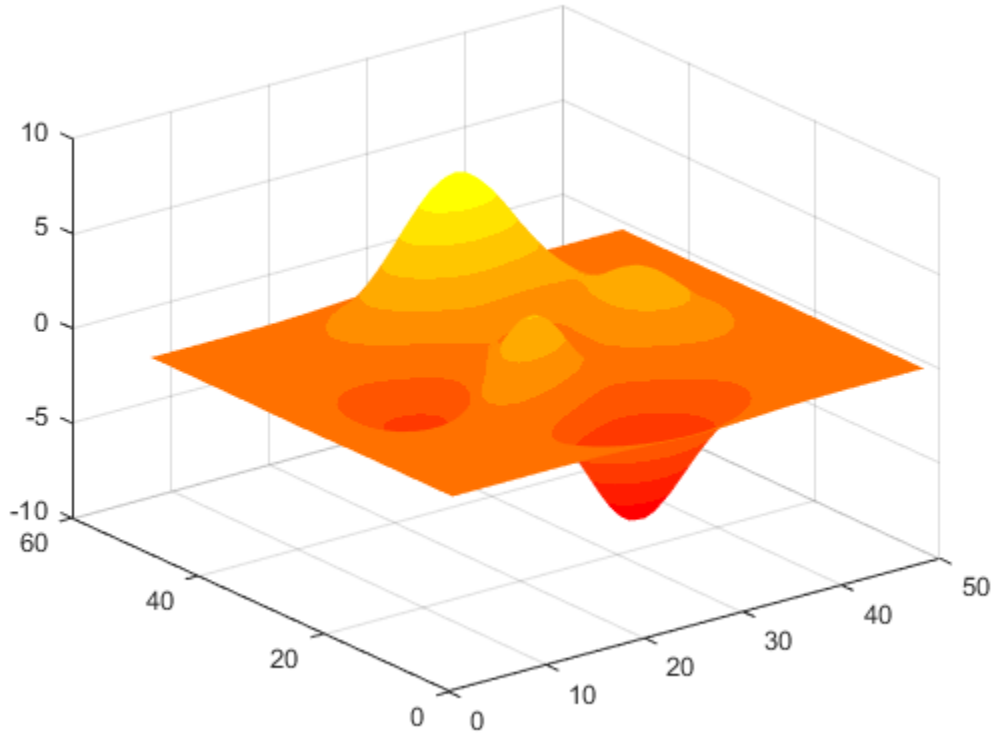
```
c = autumn;
c = flipud(c);
colormap(c);
```



Downsample the Autumn Colormap

Get a downsampled version of the autumn colormap containing only ten colors. Then display the contours of the peaks function by applying the colormap and interpolated shading.

```
c = autumn(10);
surf(peaks);
colormap(c);
shading interp;
```



Input Arguments

`m` — Number of colors

256 (default) | scalar integer value

Number of colors, specified as a scalar integer value. The default value of `m` is equal to the length of the colormap for the current figure. If no figure exists, the default value is 256.

Data Types: `single` | `double`

Compatibility Considerations

Colormaps Have 256 Colors by Default

Behavior changed in R2019b

Starting in R2019b, colormaps have 256 colors by default.

In R2019a and previous releases, the default size is 64. If you have code that depends on a colormap having 64 colors, specify the number of colors when you set the colormap for the figure, axes, or chart. For example, `colormap(autumn(64))` sets the figure's colormap to the 64-color `autumn` colormap.

Alternatively, you can change the default colormap for all figures within your MATLAB session:

```
set(groot,'defaultFigureColormap',autumn(64))
```

See Also

`colormap`

Topics

"Change Color Scheme Using a Colormap"

Introduced before R2006a

addcats

Add categories to categorical array

Syntax

```
B = addcats(A,newcats)
B = addcats(A,newcats,'Before',beforewhere)
B = addcats(A,newcats,'After',afterwhere)
```

Description

`B = addcats(A,newcats)` adds categories to the end of the category list for the input categorical array, `A`. The output categorical array, `B`, contains the same values as `A`. The output, `B`, does not contain any elements equal to the new categories until you assign values from `newcats` to elements in `B`.

If `A` is an ordinal categorical array, you must specify the `'Before',beforewhere` or `'After',afterwhere` input arguments.

`B = addcats(A,newcats,'Before',beforewhere)` adds categories before the category specified by `beforewhere`.

`B = addcats(A,newcats,'After',afterwhere)` adds categories after the category specified by `afterwhere`.

Examples

Add Categories at End

Create a nonordinal categorical array.

```
A = categorical({'republican' 'democrat' 'republican';...
    'democrat' 'republican' 'democrat'})
```

```
A = 2x3 categorical array
     republican      democrat        republican
     democrat        republican      democrat
```

Display the categories of A.

```
categories(A)
```

```
ans = 2x1 cell array
    {'democrat'  }
    {'republican'}
```

A is a 2-by-3 categorical array with two categories.

Add the categories, `independent` and `undeclared`, to the end of the category list.

```
B = addcats(A,{'independent' 'undeclared'})
```

```
B = 2x3 categorical array
     republican      democrat        republican
     democrat        republican      democrat
```

B contains the same values as A.

Display the categories of B.

```
categories(B)
```

```
ans = 4x1 cell array
    {'democrat'   }
    {'republican' }
    {'independent'}
    {'undeclared' }
```

B is a 2-by-3 categorical array with four categories.

Add Categories and Specify Category to Precede

Create an ordinal categorical array.

```
A = categorical({'medium' 'large'; 'small' 'xlarge'; 'large' 'medium'},...
    {'small' 'medium' 'large' 'xlarge'},'Ordinal',true)
```

```
A = 3x2 categorical array
     medium      large
     small       xlarge
     large       medium
```

Display the categories of A.

```
categories(A)
```

```
ans = 4x1 cell array
    {'small' }
    {'medium'}
    {'large' }
    {'xlarge'}
```

Since A is ordinal, the categories have the mathematical ordering `small < medium < large < xlarge`.

Add the category `xsmall` before `small`.

```
B = addcats(A,'xsmall','Before','small')
```

```
B = 3x2 categorical array
     medium      large
     small       xlarge
     large       medium
```

B contains the same values as A.

Display the categories of B.

```
categories(B)
```

```
ans = 5x1 cell array
    {'xsmall'}
    {'small' }
    {'medium'}
    {'large' }
    {'xlarge'}
```

The categories have the mathematical ordering `xsmall < small < medium < large < xlarge`.

Input Arguments

`A` — Categorical array
vector | matrix | multidimensional array

Categorical array, specified as a vector, matrix, or multidimensional array.

`newcats` — New categories
character vector | cell array of character vectors | string array

New categories, specified as a character vector, a cell array of character vectors, or a string array.

`beforewhere` — Category to precede
character vector | string scalar

Category to precede, specified as a character vector or a string scalar.

`afterwhere` — Category to follow
character vector | string scalar

Category to follow, specified as a character vector or a string scalar.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`categories` | `iscategory` | `mergecats` | `removecats` | `renamecats` | `reordercats` | `setcats`

Introduced in R2013b

addpath

Add folders to search path

Syntax

```
addpath(folderName1,...,folderNameN)
addpath(folderName1,...,folderNameN,position)

addpath( ___ ,'-frozen')

oldpath = addpath( ___ )
```

Description

`addpath(folderName1,...,folderNameN)` adds the specified folders to the top of the search path for the current MATLAB session.

`addpath(folderName1,...,folderNameN,position)` adds the specified folders to the top or bottom of the search path, as specified by `position`.

`addpath( ___ ,'-frozen')` additionally disables folder change detection for the folders being added. When folder change detection is disabled for a folder, MATLAB does not detect changes made to the folder from outside of MATLAB.

Use this syntax with any of the arguments in previous syntaxes. You can specify `'-frozen'` and `position` in either order.

`oldpath = addpath( ___ )` additionally returns the path prior to adding the specified folders.

Examples

Add Folder to Top of Search Path

Create a folder, add it to the top of your search path, and then save the search path for future MATLAB® sessions.

```
mkdir('matlab/myfiles')
addpath('matlab/myfiles')
savepath matlab/myfiles/pathdef.m
```

Add Folder to End of Search Path

Create the folder `matlab/myfiles` and add it to the end of the search path.

```
mkdir('matlab/myfiles')
addpath('matlab/myfiles','-end')
```

Add Folder and Its Subfolders to Search Path

Add `matlab/myfiles` and its subfolders to the search path.

Create the folder `matlab/myfiles` and call `genpath` inside of `addpath` to add all subfolders of `matlab/myfiles` to the search path.

```
mkdir('matlab/myfiles')
addpath(genpath('matlab/myfiles'))
```

Add Folder to Search Path and Disable Folder Change Notification

Create the folder `matlab/myfiles`. Then, add it to the top of the search path, disable folder change notification, and return the search path before adding the folder.

```
mkdir('matlab/myfiles')
oldpath = addpath('matlab/myfiles','-frozen');
```

Disabling folder change notification is not supported in MATLAB® Online™.

Input Arguments

`folderName1,...,folderNameN` — Folder names to add to search path
character vectors | string scalars

Folder names to add to the search path, specified as one or more character vectors or string scalars. Use the full path name for each folder. Use `genpath` with `addpath` to add all subfolders of `folderName`.

Example: `'c:\matlab\work'`

Example: `'/home/user/matlab'`

Example: `'/home/user/matlab','/home/user/matlab/test'`

MATLAB resolves all path names containing `'.'`, `'..'`, and symbolic links to their target location before adding them to the path. This ensures that each entry in the MATLAB path represents a unique folder location. For example, if you specify `c:\matlab\..\work`, MATLAB adds the folder `c:\work` to the path.

Data Types: `char` | `string`

`position` — Position on search path
`'-begin'` (default) | `'-end'`

Position on the search path, specified as one of the following:

Value of `position`	Description
`'-begin'`	Add specified folders to the top of the search path.
`'-end'`	Add specified folders to the bottom of the search path.

Output Arguments

`oldpath` — Path prior to addition of folders
character vector

Path prior to the addition of folders, returned as a character vector.

Tips

- To save the newly modified search path for future MATLAB sessions, use the `savepath` function.
- To modify the search path programmatically at startup, use `addpath` statements in a `startup.m` file. For more information, see "Add Folders to the MATLAB Search Path at Startup".

Algorithms

If you use `addpath` within a local function, the path change persists after program control returns from the function. That is, the scope of the path change is global.

See Also

`genpath` | `path` | `pathsep` | `rmpath` | `savepath`

Topics

"What Is the MATLAB Search Path?"
"Files and Folders that MATLAB Accesses"
"Specify File Names"
"Add Folders to the MATLAB Search Path at Startup"

Introduced before R2006a

addpref

Add custom preference

Syntax

```
addpref(group,pref,value)
```

Description

`addpref(group,pref,value)` creates the specified preference in the specified group and sets its value to `value`. If the specified preference exists, MATLAB returns an error.

Preferences are persistent and maintain their values between MATLAB sessions.

Examples

Add New Preference

Add a preference called `version` to the `mytoolbox` group of preferences. Set the value of `version` to the cell array `{'1.0','beta'}`.

```
addpref('mytoolbox','version',{'1.0','beta'})
```

Add Multiple New Preferences

Add two preferences to the `mytoolbox` group of preferences and set their values.

```
addpref('mytoolbox',{'modifieddate','docpath'},{'1/9/2019','C:\mytoolbox\documentation'})
```

Input Arguments

`group` — Custom preference group name

character vector | string scalar

Preference group name, specified as a character vector or a string scalar. `group` must be a valid variable name. For more information, see “Variable Names”.

Example: `'mytoolbox'`

Data Types: `char` | `string`

`pref` — Custom preference name

character vector | cell array of character vectors | string array

Custom preference name, specified as a character vector, a cell array of character vectors, or a string array. `pref` must be a valid variable name. For more information, see “Variable Names”.

If `pref` is a cell array of character vectors or a non-scalar string, `value` must specify a value for each preference specified in `pref`.

Example: `'version'`

Example: `{'version','modifieddate','docpath'}`

Data Types: `char` | `string`

`value` — Custom preference value

any MATLAB data type

Custom preference value, specified as any MATLAB data type, including numeric types, character vectors, cell arrays, structures, and objects. If `pref` is a cell array of character vectors or a nonscalar string array, `value` must specify a cell array that includes a value for each preference specified in `pref`.

Example: `1.1`

Example: `{{1.1,'beta'},datetime(2018,1,9),'C:\mytoolbox\documentation'}`

See Also

`getpref` | `ispref` | `rmpref` | `setpref` | `uigetpref` | `uisetpref`

Topics

"Preferences"

Introduced before R2006a

addpoints

Add points to animated line

Syntax

```
addpoints(an,x,y)
addpoints(an,x,y,z)
```

Description

`addpoints(an,x,y)` adds points defined by `x` and `y` to the animated line specified by `an`. Create an animated line with the `animatedline` function. To display the updates on the screen, use `drawnow` or `drawnow limitrate`. New points automatically connect to previous points.

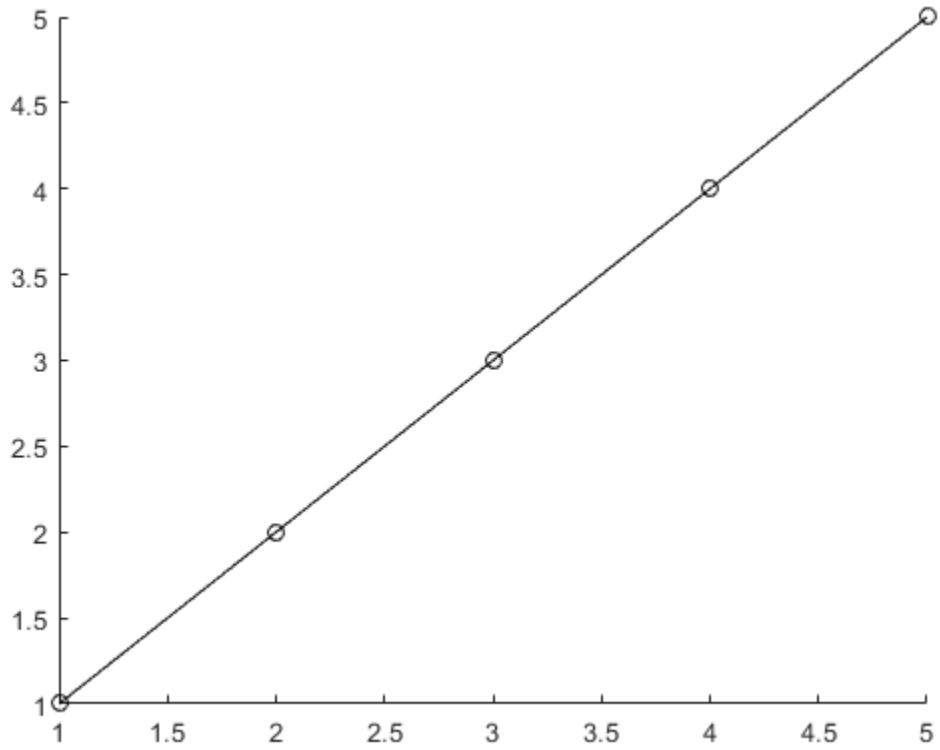
`addpoints(an,x,y,z)` adds points defined by `x`, `y`, and `z` to the 3-D animated line specified by `an`.

Examples

Add Five Points to Animated Line

Create an animated line object with no data. Then, add five points to the line. Use a circle to mark each point.

```
h = animatedline('Marker','o');
x = 1:5;
y = 1:5;
addpoints(h,x,y)
```

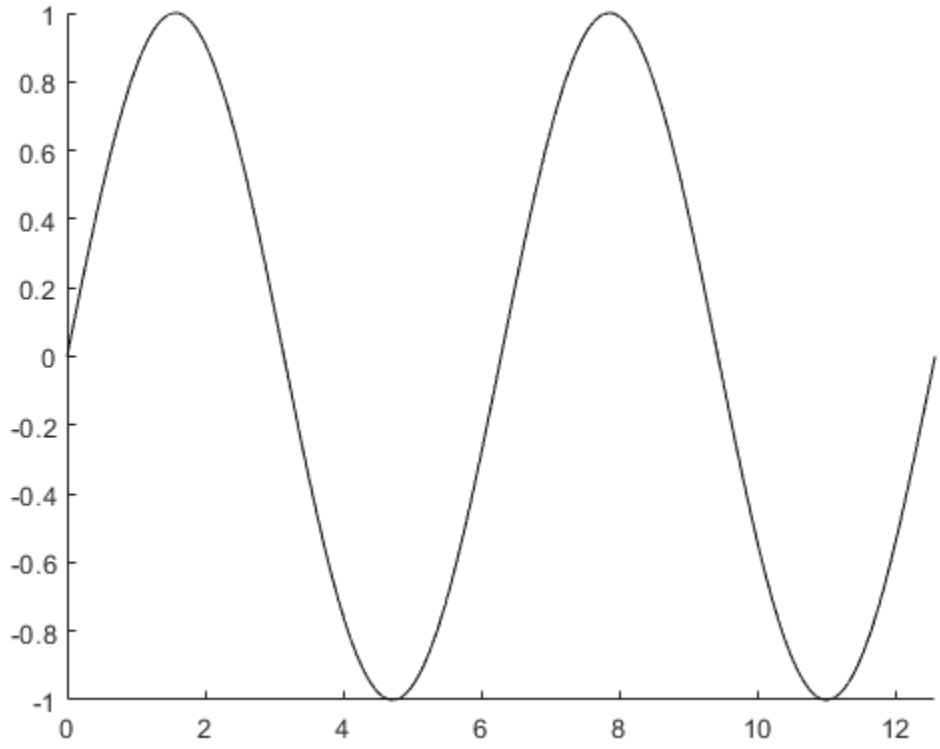


Add Points Within Loop to Animated Line

Create an animated line using the `animatedline` function. Then, add points to the line within a loop to create an animation. Set the axis limits before the loop to prevent the limits from changing.

```
figure
h = animatedline;
axis([0 4*pi -1 1])

for x = linspace(0,4*pi,10000)
```

```
    y = sin(x);
    addpoints(h,x,y)
    drawnow limitrate
end
```



Input Arguments

an — AnimatedLine object
AnimatedLine object

AnimatedLine object. Create an AnimatedLine object using the animatedline function.

x — First coordinate values
scalar | vector

First coordinate values, specified as a scalar or a vector. For Cartesian axes, the first coordinate is *x*-axis position. For geographic axes, the first coordinate is latitude in degrees.

The length of `x` must equal the length of `y`.

Example: `11:20`

Data Types: `double`

y — Second coordinate values
scalar | vector

Second coordinate values, specified as a scalar or a vector. For Cartesian axes, the second coordinate is *y*-axis position. For geographic axes, the second coordinate is longitude in degrees.

The length of `y` must equal the length of `x`.

Example: `11:20`

Data Types: `double`

z — Third coordinate values
scalar | vector

Third coordinate values, specified as a scalar or a vector. For Cartesian axes, the third coordinate is *z*-axis position.

The length of `z` must equal the length of `x` and `y`.

Example: `11:20`

Data Types: `double`

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions
`animatedline` | `clearpoints` | `getpoints`

Properties
AnimatedLine

Introduced in R2014b

addprop

Add custom properties to table or timetable

Syntax

```
T = addprop(T,propertyNames,propertyTypes)
```

Description

`T = addprop(T,propertyNames,propertyTypes)` adds properties that contain custom metadata to the table or timetable `T`. The input argument `propertyNames` specifies the names of the properties. For each custom property, `propertyTypes` specifies whether the metadata values contained in the property apply to `T` as a whole, or to the variables of `T`.

After you add properties using `addprop`, you can assign metadata values to the properties using dot syntax.

Examples

Add Custom Properties

Read data into a table. Then add properties to contain custom metadata.

First, read measurements of humidity and air quality into a table. Display the first three rows.

```
T = readtable('indoors.csv');
head(T,3)
```

```
ans=3×3 table
           Time            Humidity    AirQuality
    ___________________    ________    __________
```

```
    2015-11-15 00:00:24       36           80
    2015-11-15 01:13:35       36           80
    2015-11-15 02:26:47       37           79
```

Display the properties of the table. The properties object, `T.Properties`, stores metadata such as the names of the two dimensions of the table and the names of the table variables. All tables have such objects with the same properties. (Timetables also have similar objects that include additional, time-specific properties.)

```
T.Properties
```

```
ans = 
  TableProperties with properties:

             Description: ''
                UserData: []
          DimensionNames: {'Row'  'Variables'}
           VariableNames: {'Time'  'Humidity'  'AirQuality'}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                RowNames: {}
        CustomProperties: No custom properties are set.
      Use addprop and rmprop to modify CustomProperties.
```

In addition, you can specify your own properties to store custom metadata. For example, use the `addprop` function to add properties to the table `T` for the instrument name, measurement precision, and the name of the source file. For properties that have one metadata value per variable, specify `'variable'` as the property type. For properties that have one value that applies to the whole table, specify `'table'`.

```
T = addprop(T,{'Instrument','Precision','SourceFile'},{'variable','variable','table'});
T.Properties
```

```
ans = 
  TableProperties with properties:

             Description: ''
                UserData: []
          DimensionNames: {'Row'  'Variables'}
           VariableNames: {'Time'  'Humidity'  'AirQuality'}
    VariableDescriptions: {}
```

```
           VariableUnits: {}
      VariableContinuity: []
                 RowNames: {}

   Custom Properties (access using t.Properties.CustomProperties.<name>):
             SourceFile: []
             Instrument: []
              Precision: []
```

When you create custom properties using `addprop`, the properties are empty. To store metadata values in the custom properties, assign them using dot syntax.

```
T.Properties.CustomProperties.Instrument = ["clock" "hygrometer" "air quality meter"];
T.Properties.CustomProperties.Precision = [NaN 0.5 0.1];
T.Properties.CustomProperties.SourceFile = 'indoors.csv';
T.Properties
```

```
ans =
  TableProperties with properties:

            Description: ''
               UserData: []
          DimensionNames: {'Row'  'Variables'}
           VariableNames: {'Time'  'Humidity'  'AirQuality'}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                 RowNames: {}

   Custom Properties (access using t.Properties.CustomProperties.<name>):
             SourceFile: 'indoors.csv'
             Instrument: ["clock"    "hygrometer"    "air quality meter"]
              Precision: [NaN 0.5000 0.1000]
```

When you assign an array of text values to custom properties, the best practice is to use a string array, not a cell array of character vectors. If you use a cell array of character vectors, then there is no mechanism to prevent you from later assigning nontext values as elements of the cell array.

Input Arguments

T — Input table
table | timetable

Input table, specified as a table or timetable.

`propertyNames` — Names of custom properties
character vector | cell array of character vectors | string array

Names of the custom properties, specified as a character vector, cell array of character vectors, or string array.

`propertyTypes` — Property types
character vector | cell array of character vectors | string array

Property types, specified as a character vector, cell array of character vectors, or string array. For each property name specified by `propertyNames`, specify the corresponding property type as either `'table'` or `'variable'`. The number of property types must equal the number of property names.

The table describes the two property types.

Property Type	Description
`'table'`	The property contains a single value of arbitrary size. The value applies as metadata to the table or timetable as a whole.

Property Type	Description
`'variable'`	The property contains an array that has one value for each variable in the table or timetable. The values are metadata for the variables. The number of values in the array must match the number of variables. The values stored by the property are synchronized with the variables. They respond when you take one of these actions: • Move variables — The corresponding values in the property are reordered. • Add variables — Default values are added as corresponding values in the property. • Remove variables — The corresponding values are removed from the property.

See Also

`rmprop` | `summary` | `table` | `timetable`

Topics

"Add Custom Properties to Tables and Timetables"
"Access Data in Tables"
"Modify Units, Descriptions, and Table Variable Names"

Introduced in R2018b

addprop

Class: `dynamicprops`

Add dynamic property

Syntax

```
P = addprop(A,PropertyName)
```

Description

`P = addprop(A,PropertyName)` adds a property named `PropName` to each object in array `A`. The output argument `P` is an array of `meta.DynamicProperty` objects that is the same size as `A`.

Dynamic properties exist only on the specific instance for which they are defined. Dynamic properties do not become part of the class definition.

You can add dynamic properties only to objects derived from the `dynamicprops` class. Access the data in dynamic properties using the instance variable and the property name (`obj.PropertyName`).

Input Arguments

`A` — Input array
object array

Input array, specified as an object array of a class that is derived from `dynamicprops`.

`PropertyName` — Name of dynamic property
character vector | string

Name of dynamic property, specified as a char vector.

Example: `'DynoProp'`

Data Types: `char` | `string`

Output Arguments

P — Output array
`meta.DynamicProperty`

Output array returned as an array of `meta.DynamicProperty` objects. Use `meta.DynamicProperty` objects to define access methods for dynamic properties, to set property attributes, and to remove dynamic properties.

Attributes

`Access`	`Public`

To learn about attributes of methods, see Method Attributes.

Examples

Add Dynamic Property

Add a dynamic property to an object of the `Dyno` class.

```
classdef Dyno < dynamicprops
   properties
      Prop1
   end
end
```

Create an object

```
o = Dyno;
```

Add a property called `DynoProp`

```
p = addprop(o,'DynoProp');
```

Set the `AbortSet` attribute of the dynamic property

```
p.AbortSet = 1;
```

Remove the dynamic property

```
delete(p)
```

See Also

`dynamicprops` | `meta.DynamicProperty`

Topics

"Dynamic Properties — Adding Properties to an Instance"
"Set and Get Methods for Dynamic Properties"

Introduced in R2008a

addproperty

Add custom property to COM object

Syntax

```
addproperty(c,name)
```

Description

`addproperty(c,name)` adds custom property `name` to object or interface `c`.

Input Arguments

`c` — COM object
function handle

COM object, specified as a function handle.

`name` — Property name
character vector

Property name, specified as a character vector.

Example: `'Position'`

Limitations

- COM functions are available on Microsoft Windows systems only.

See Also

Property Inspector | `deleteproperty` | `get` | `set`

Topics

"COM Object Properties"

Introduced before R2006a

addsampletocollection

Add sample to `tscollection`

Syntax

```
tscout = addsampletocollection(tscin,'Time',timevals,tsname,tsdata)
```

Description

`tscout = addsampletocollection(tscin,'Time',timevals,tsname,tsdata)` adds data samples to the `tscollection` member specified by the name `tsname` for one or more time values. `tsdata` contains the sample data.

To add samples to more than one `tscollection` member at a time, continue to list each `tscollection` member name followed by the corresponding data separated by commas. For example, the command `tscout = addsampletocollection(tscin,'Time',timevals,ts1name,ts1data,ts2name, ts2data)` adds samples to two `timeseries` in `tscin` simultaneously.

Examples

Add `tscollection` Sample

Create a `tscollection` object from two `timeseries` objects and add a sample to the `tscollection`.

```
ts1 = timeseries([1.1 2.9 3.7 4.0 3.0]',1:5,'Name','Acceleration');
ts2 = timeseries([3.2 4.2 6.2 8.5 1.1]',1:5,'Name','Speed');
tscin = tscollection({ts1;ts2});
tscout = addsampletocollection(tscin,'Time',3.5,'Acceleration',10,'Speed',4.9);
tscout.Acceleration.Data
```

```
ans = 6×1
```

```
    1.1000
    2.9000
    3.7000
   10.0000
    4.0000
    3.0000
```

```
tscout.Speed.Data
```

```
ans = 6×1

    3.2000
    4.2000
    6.2000
    4.9000
    8.5000
    1.1000
```

Input Arguments

`tscin` — Input `tscollection`

scalar

Input `tscollection`, specified as a scalar.

`timevals` — Sample times

scalar | vector

Sample times, specified as a numeric scalar or vector, or a cell array of date character vectors. Valid date character vectors and strings can have the following forms:

Format	Example
dd-mmm-yyyy HH:MM:SS	01-Mar-2000 15:45:17
dd-mmm-yyyy	01-Mar-2000
mm/dd/yy	03/01/00
mm/dd	03/01

Format	Example
HH:MM:SS	15:45:17
HH:MM:SS PM	3:45:17 PM
HH:MM	15:45
HH:MM PM	3:45 PM
mmm.dd,yyyy HH:MM:SS	Mar.01,2000 15:45:17
mmm.dd,yyyy	Mar.01,2000
mm/dd/yyyy	03/01/2000

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | cell

tsname — timeseries name
character vector

timeseries name, specified as a character vector.

Data Types: char

tsdata — timeseries data
scalar | vector | multidimensional array

timeseries data, specified as a numeric or logical scalar, vector, or multidimensional array.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

Tips

- If you do not specify data samples for a tscollection member, that timeseries will contain NaN values for the times specified in timevals for numeric data. For logical data, the timeseries will contain false values.
- When a tscollection member requires Quality values, you can specify data quality codes together with the data samples by using the following syntax:

  ```
  tsc = addsampletocollection(tsc,'Time',timevals,...
  ts1name,ts1cellarray,ts2name,ts2cellarray,...)
  ```

Specify the data in the first cell array element and `Quality` in the second cell array element.

If a `tscollection` member already has `Quality` values, but you only provide data samples, then `0` is added to the existing `Quality` array at the times specified in `timevals`.

See Also

`delsamplefromcollection` | `timeseries` | `tscollection`

Introduced before R2006a

addtodate

Modify date number by field

Syntax

```
R = addtodate(D, Q, F)
```

Description

`R = addtodate(D, Q, F)` adds quantity `Q` to the indicated date field `F` of a scalar serial date number `D`, returning the updated date number `R`.

The quantity `Q` to be added can be a positive or negative integer. The absolute value of `Q` must be less than or equal to 1e16. The date field `F` must be a character vector or string scalar equal to one of the following: `'year'`, `'month'`, `'day'`, `'hour'`, `'minute'`, `'second'`, or `'millisecond'`.

If the addition to the date field causes the field to roll over, the MATLAB software adjusts the next more significant fields accordingly. Adding a negative quantity to the indicated date field rolls back the calendar on the indicated field. If the addition causes the field to roll back, MATLAB adjusts the next less significant fields accordingly.

Examples

Modify the hours, days, and minutes of a given date:

```
t = datenum('07-Apr-2008 23:00:00');
datestr(t)
ans =
   07-Apr-2008 23:00:00

t= addtodate(t, 2, 'hour');
datestr(t)
ans =
   08-Apr-2008 01:00:00
```

```
t= addtodate(t, -7, 'day');
datestr(t)
ans =
   01-Apr-2008 01:00:00

t= addtodate(t, 59, 'minute');
datestr(t)
ans =
   01-Apr-2008 01:59:00
```

Adding 20 days to the given date in late December causes the calendar to roll over to January of the next year:

```
R = addtodate(datenum('12/24/2007 12:45'), 20, 'day');

datestr(R)
ans =
   13-Jan-2008 12:45:00
```

See Also

date | datenum | datestr | datevec

Introduced before R2006a

addToolbarExplorationButtons

Add data exploration buttons to figure toolbar

Syntax

```
addToolbarExplorationButtons(fig)
```

Description

`addToolbarExplorationButtons(fig)` adds the data exploration buttons on page 1-191 to the figure toolbar for the specified figure.

In R2018b, the data exploration buttons were moved from the figure toolbar to the axes toolbar. In most cases, you do not need to use this function. However, if you have code that relies on the buttons appearing in the figure toolbar, you can use the `addToolbarExplorationButtons` and `removeToolbarExplorationButtons` functions to control the appearance of the buttons in the figure toolbar.

Examples

Add and Remove Data Exploration Buttons

Create a figure with a surface plot. Then add the data exploration buttons to the figure toolbar.

```
fig = figure;
surf(peaks);
addToolbarExplorationButtons(fig)
```

Notice that the figure toolbar now includes buttons to zoom in, zoom out, and so on.

Remove the buttons from the figure toolbar.

```
removeToolbarExplorationButtons(fig)
```

Input Arguments

`fig` — Target figure
single `Figure` object | vector of `Figure` objects

Target figure, specified as a single `Figure` object or a vector of `Figure` objects.

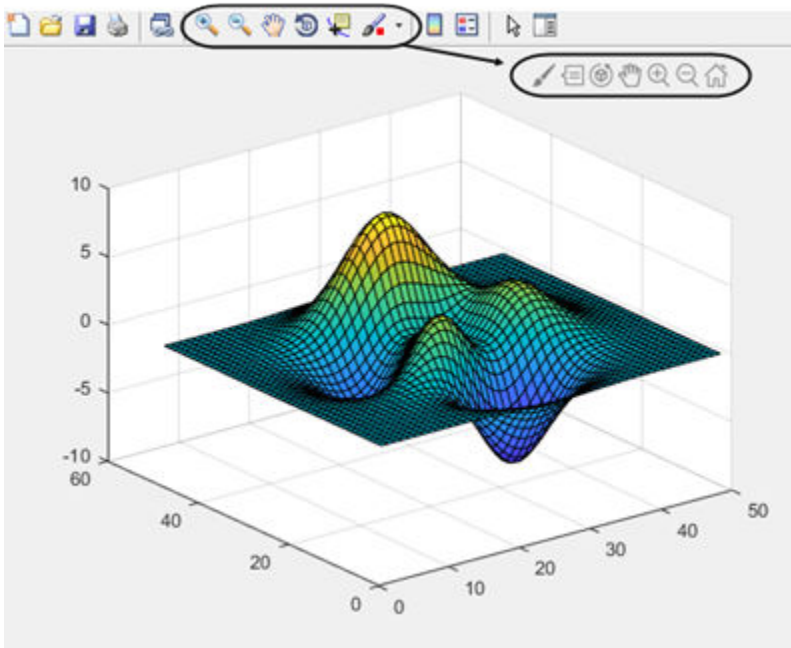
More About

Data Exploration Buttons

The standard data exploration buttons include options to:

- Zoom in or out of the axes view
- Pan the axes view
- Rotate the axes view
- Show data tips
- Brush data

In R2018b, the data exploration buttons were moved from the figure toolbar to the axes toolbar. This figure illustrates the relocation of the buttons.



See Also

`axtoolbar` | `removeToolbarExplorationButtons`

Introduced in R2018a

addts

Add `timeseries` to `tscollection`

Syntax

```
tscout = addts(tscin,ts)
tscout = addts(tscin,ts,tsname)
tscout = addts(tscin,tsdata,tsname)
```

Description

`tscout = addts(tscin,ts)` adds a `timeseries` object `ts` to the `tscollection` object `tscin`. The input `timeseries` can be a single `timeseries` object or a cell array of `timeseries` objects.

`tscout = addts(tscin,ts,tsname)` specifies the names of the input `timeseries` objects. `tsname` can be a single character vector or a cell array of character vectors defining the names for each `timeseries` in `ts`.

`tscout = addts(tscin,tsdata,tsname)` adds a `timeseries` object with data `tsdata` and name `tsname` to the input `tscollection`.

Examples

Add `timeseries` to Collection

Create a `tscollection` object containing one `timeseries` object. Then, add a second `timeseries` to the collection.

```
ts1 = timeseries([1.1 2.9 3.7 4.0 3.0]',1:5,...
                 'Name','Acceleration');
ts2 = timeseries([3.2 4.2 6.2 8.5 1.1]',1:5,...
                 'Name','Speed');
```

```
tscin = tscollection(ts1);
tscout = addts(tscin,ts2)
```

```
Time Series Collection Object: unnamed

Time vector characteristics

      Start time            1 seconds
      End time              5 seconds

Member Time Series Objects:

      Acceleration
      Speed
```

Input Arguments

`tscin` — Input `tscollection`
scalar

Input `tscollection`, specified as a scalar.

`ts` — Input `timeseries`
scalar | cell array

Input `timeseries`, specified as a scalar `timeseries` object or a cell array of `timeseries` objects.

`tsname` — `timeseries` name
character vector | cell array of character vectors

`timeseries` name, specified as a character vector or a cell array of character vectors.

`tsdata` — `timeseries` data
scalar | vector | multidimensional array

`timeseries` data, specified as a numeric or `logical` scalar, vector, or multidimensional array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Tips

- The `timeseries` objects that you add to the collection must have the same time vector as the collection. That is, the time vectors must have the same time values and units.
- Suppose that the time vector of a `timeseries` object is associated with calendar dates. When you add the `timeseries` to a collection with a time vector without calendar dates, the time vectors are compared based on the units and the values relative to the `StartDate` property.

See Also

`removets` | `timeseries` | `tscollection`

Introduced before R2006a

addvars

Add variables to table or timetable

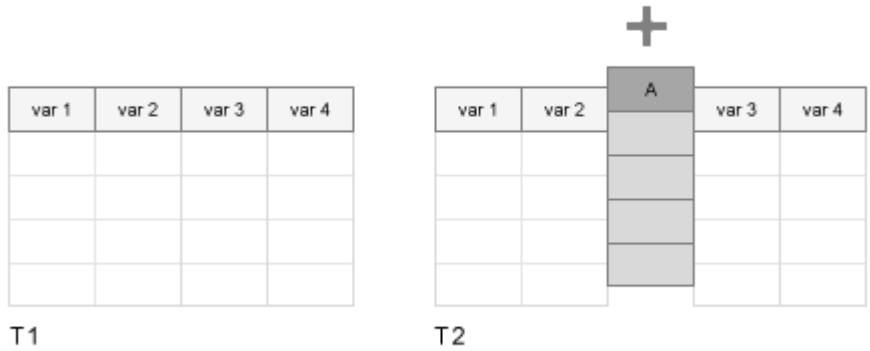
Syntax

```
T2 = addvars(T1,var1,...,varN)
T2 = addvars(T1,var1,...,varN,'Before',location)
T2 = addvars(T1,var1,...,varN,'After',location)
T2 = addvars( ___ ,'NewVariableNames',newNames)
```

Description

`T2 = addvars(T1,var1,...,varN)` adds the variables specified by `var1,…,varN` to the right of the last variable of `T1`. The input arguments `var1,…,varN` can include arrays of any type, tables, and timetables. All input arguments must have the same number of rows.

`T2 = addvars(T1,var1,...,varN,'Before',location)` inserts the variables to the left of the table variable indicated by `location` (see diagram). You can specify `location` as a variable name, or a numeric or logical index.



`T2 = addvars(T1,var1,...,varN,'After',location)` inserts the variables to the right of the table variable indicated by `location`.

`T2 = addvars( ___ ,'NewVariableNames',newNames)` renames the added variables in `T2` using the names specified by `newNames`. The number of names in `newNames` must be the same as the number of added variables. You can use this syntax with any of the input arguments of the previous syntaxes.

Examples

Add Variables

Create a table. Then add variables from the workspace to the table.

Load arrays from the `patients.mat` file. Create a table that contains the names, ages, heights, and weights of patients. Then display the first three rows.

```
load patients
T1 = table(LastName,Age,Height,Weight);
head(T1,3)
```

```
ans=3×4 table
      LastName      Age    Height    Weight
```

```
    ____________    ___    ______    ______

    {'Smith'   }    38       71       176
    {'Johnson' }    43       69       163
    {'Williams'}    38       64       131
```

Add the workspace variables, `Gender` and `Smoker`, to the table.

```
T2 = addvars(T1,Gender,Smoker);
head(T2,3)
```

```
ans=3×6 table
      LastName      Age    Height    Weight      Gender      Smoker
    ____________    ___    ______    ______    __________    ______

    {'Smith'   }    38       71       176      {'Male'  }    true
    {'Johnson' }    43       69       163      {'Male'  }    false
    {'Williams'}    38       64       131      {'Female'}    false
```

Insert Variables at Specified Locations

Create a table. Then insert variables before and after specified locations in the table.

Load arrays from the `patients.mat` file. Create a table that contains the names and genders of patients. Then display the first three rows.

```
load patients
T1 = table(LastName,Gender);
head(T1,3)
```

```
ans=3×2 table
      LastName        Gender
    ____________    __________

    {'Smith'   }    {'Male'  }
    {'Johnson' }    {'Male'  }
    {'Williams'}    {'Female'}
```

Insert the workspace variable, `Age`, before the table variable, `Gender`. To refer to a table variable by name, specify its name as a character vector.

```
T2 = addvars(T1,Age,'Before','Gender');
head(T2,3)
```

```
ans=3×3 table
      LastName      Age      Gender
    ____________    ___    __________

    {'Smith'   }    38     {'Male'  }
    {'Johnson' }    43     {'Male'  }
    {'Williams'}    38     {'Female'}
```

Insert more variables after `Age`. Since Age is a table variable in T2, specify its name as a character vector.

```
T3 = addvars(T2,Height,Weight,'After','Age');
head(T3,3)
```

```
ans=3×5 table
      LastName      Age    Height    Weight      Gender
    ____________    ___    ______    ______    __________

    {'Smith'   }    38       71       176      {'Male'  }
    {'Johnson' }    43       69       163      {'Male'  }
    {'Williams'}    38       64       131      {'Female'}
```

Insert `Smoker` after the first table variable. You can specify variables by position in the table instead of by name.

```
T4 = addvars(T3,Smoker,'After',1);
head(T4,3)
```

```
ans=3×6 table
      LastName      Smoker    Age    Height    Weight      Gender
    ____________    ______    ___    ______    ______    __________

    {'Smith'   }    true      38       71       176      {'Male'  }
    {'Johnson' }    false     43       69       163      {'Male'  }
    {'Williams'}    false     38       64       131      {'Female'}
```

Rename Variables

Create a table. Add variables and give them new names in the table.

First, create a table from workspace variables.

```
load patients
T1 = table(LastName,Age,Gender,Smoker);
head(T1,3)
```

```
ans=3×4 table
      LastName      Age      Gender      Smoker
    ____________    ___    __________    ______

    {'Smith'   }    38     {'Male'  }    true
    {'Johnson' }    43     {'Male'  }    false
    {'Williams'}    38     {'Female'}    false
```

Combine `Diastolic` and `Systolic` into one matrix with two columns. Name the new table variable `BloodPressure`.

```
T2 = addvars(T1,[Diastolic Systolic],'NewVariableNames','BloodPressure');
head(T2,3)
```

```
ans=3×5 table
      LastName      Age      Gender      Smoker    BloodPressure
    ____________    ___    __________    ______    _____________

    {'Smith'   }    38     {'Male'  }    true        93    124
    {'Johnson' }    43     {'Male'  }    false       77    109
    {'Williams'}    38     {'Female'}    false       83    125
```

Add `Height` and `Weight` as new table variables. Rename them `Inches` and `Pounds`.

```
T3 = addvars(T2,Height,Weight,'Before','Smoker','NewVariableNames',{'Inches','Pounds'})
head(T3,3)
```

```
ans=3×7 table
      LastName      Age      Gender      Inches    Pounds    Smoker    BloodPressure
    ____________    ___    __________    ______    ______    ______    _____________

    {'Smith'   }    38     {'Male'  }      71       176      true        93    124
    {'Johnson' }    43     {'Male'  }      69       163      false       77    109
```

```
    {'Williams'}    38     {'Female'}      64       131      false       83    125
```

Input Arguments

T1 — Input table
table | timetable

Input table, specified as a table or timetable.

var1,...,varN — Variables to add to output table
arrays, tables, and timetables

Variables to add to the output table, specified as arrays, tables, and timetables. The variables specified by `var1,...,varN` all must have the same number of rows as the input table `T1`.

Example: `T2 = addvars(T1,A)` inserts the workspace variables `A` to the right of the last table variable.

Example: `T2 = addvars(T1,X,Y,Z)` inserts the workspace variables `X`, `Y`, and `Z`.

location — Location to insert added variables
character vector | string scalar | integer | logical array

Location to insert added variables, specified as a character vector, string scalar, integer, or logical array.

- If `location` is a character vector or string scalar, then it is the name of a variable in the input table `T1`.
- If `location` is the integer `n`, then it specifies the `n`th variable in `T1`.
- If `location` is a logical array, whose `n`th element is `1` (`true`), then it specifies the `n`th variable in `T1`. All other elements of `location` must be `0` (`false`).

Example: `T2 = addvars(T1,Latitude,'Before','Longitude')` insert the workspace variable `Latitude` to the left of the table variable named `Longitude`.

Example: `T2 = addvars(T1,Y,Z,'After','X')` inserts the workspace variables `Y` and `Z` to the right of the table variable named `X`.

newNames — Names of added variables
character vector | cell array of character vectors | string array

Names of the added variables, specified as a character vector, cell array of character vectors, or string array.

Example: `T2 = addvars(T1,lat,lon,'NewVariableNames', {'Latitude','Longitude'})` inserts the workspace variables `lat` and `lon` and names the corresponding table variables `'Latitude'` and `'Longitude'`.

Limitations

- Use single quotes for the input names `'Before'`, `'After'`, and `'NewVariableNames'`. To avoid confusion with variable inputs, do not use double-quoted string scalars (such as `"Before"`) for these names.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also
`mergevars` | `movevars` | `removevars` | `splitvars`

Topics

"Add, Delete, and Rearrange Table Variables"
"Add and Delete Table Rows"
"Access Data in Tables"
"Modify Units, Descriptions, and Table Variable Names"
"Clean Messy and Missing Data in Tables"

Introduced in R2018a

arguments

Declare function argument validation

Syntax

```
arguments ... end
arguments (Repeating) ... end
```

Description

`arguments ... end` delimits a code block used at the beginning of a function to declare input argument restrictions. Inside an arguments block, you can impose restrictions on the sizes and classes of input arguments, and can test for specific values using validation functions.

`arguments (Repeating) ... end` delimits a block of repeating arguments.

Examples

Restrict Size and Class of Input

Use an `arguments` block to restrict the size and class of the input argument.

Write a function that restricts the size and class of the input argument to a matrix of numeric values. For more information on using `arguments` blocks, see “arguments Block Syntax”.

```
function [m,s] = Stat(x)
    arguments
        x (:,:) {mustBeNumeric}
    end
    m = mean(x,'all');
    s = std(x,1,'all');
end
```

Call the function on an input of the wrong size. An error results.

```
[m,s] = Stat(rand(10,10,10))
```

```
Invalid input argument at position 1. Value must be a matrix.
```

Repeating Arguments

Define repeating arguments.

Repeating arguments are single or groups of arguments that can be repeated zero or more times in a function call. The `fRepeat` function accepts repeating groups of arguments `x`, `y`, and `style`. `x` and `y` must be vectors of doubles or convertible to doubles. `style` must be the character string `'--'` or `':'` .

For more information on repeating arguments, see “Repeating Arguments”.

```
function fRepeat(x,y,style)
    arguments (Repeating)
        x (1,:) double
        y (1,:) double
        style {mustBeMember(style,{'--',':'})}
    end

    % Interleave inputs and call plot function
    z = reshape([x;y;style],1,[]);
    if ~isempty(z)
        plot(z{:});
    end
end
```

Call `fRepeat` with two groups of inputs.

```
x1 = 1:10; x2 = 1:5;
y1 = 1:10; y2 = 1:5;
```

```
s1 = ':'; s2 = '--';
fRepeat(x1,y1,s1,x2,y2,s2)
```

See Also

Topics

"Function Argument Validation"

Introduced in R2019b

airy

Airy Functions

Syntax

```
W = airy(Z)
W = airy(k,Z)
W = airy(k,Z,scale)
```

Description

`W = airy(Z)` returns the Airy function, $Ai(Z)$, for each element of Z.

`W = airy(k,Z)` returns any of four different Airy functions, depending on the value of `k`, such as the Airy function of the second kind or the first derivative of an Airy function.

`W = airy(k,Z,scale)` scales the resulting Airy function. `airy` applies a specific scaling function to `W` depending on your choice of `k` and `scale`.

Examples

Airy Function of Real-Valued x

Define `x`.

```
x = -10:0.01:1;
```

Calculate *Ai(x)*

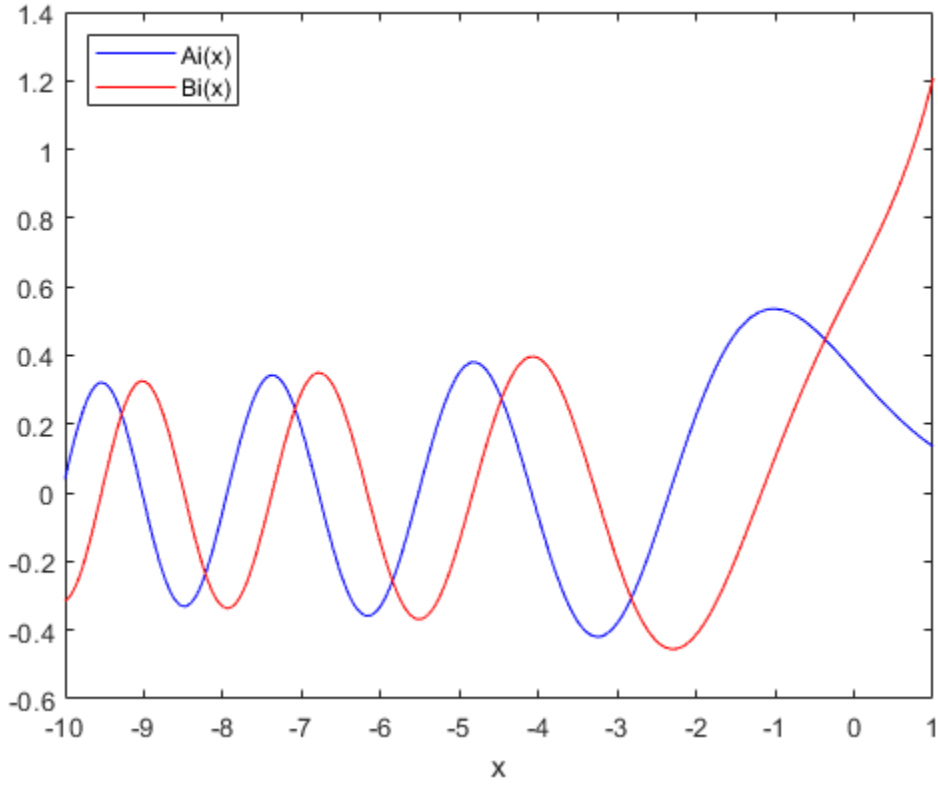
```
ai = airy(x);
```

Calculate *Bi(x)* using $k = 2$.

```
bi = airy(2,x);
```

Plot both results together on the same axes.

```
figure
plot(x,ai,'-b',x,bi,'-r')
axis([-10 1 -0.6 1.4])
xlabel('x')
legend('Ai(x)','Bi(x)','Location','NorthWest')
```



Airy Function of Complex-Valued x

Compute the Airy function at a slice through the complex plane at $x + i$.

Take a slice through the complex plane.

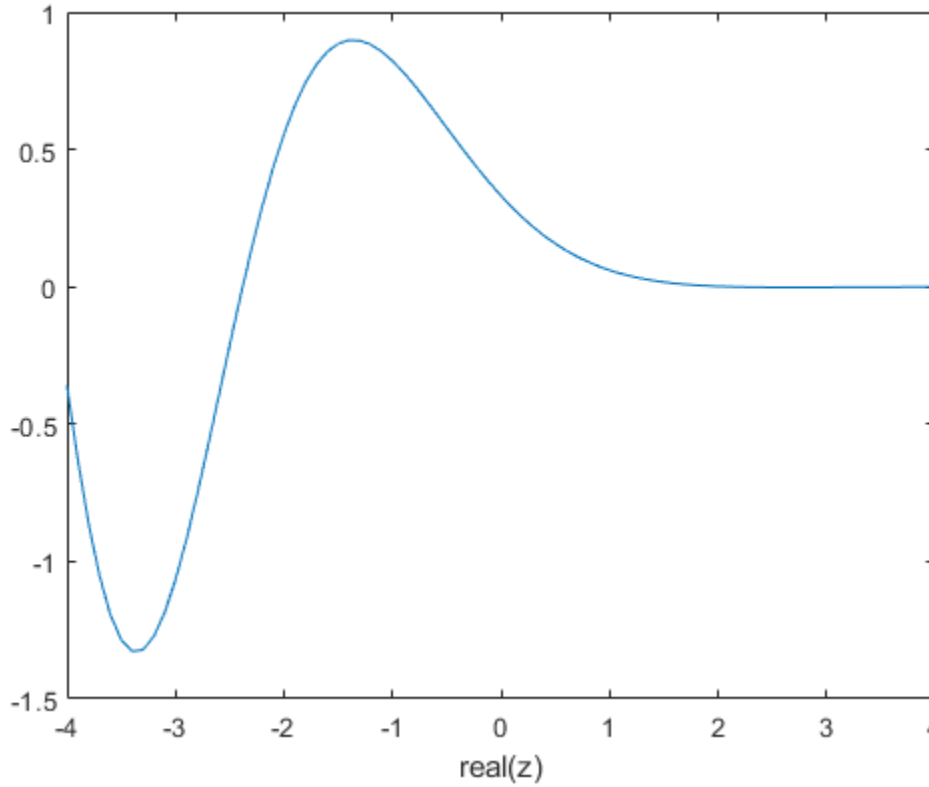
```
x = -4:0.1:4;
z = x+1i;
```

Calculate *Ai(z)*.

```
w = airy(z);
```

Plot the real part of the result.

```
figure
plot(x, real(w))
axis([-4 4 -1.5 1])
xlabel('real(z)')
```



Scaled Airy Function

Define x.

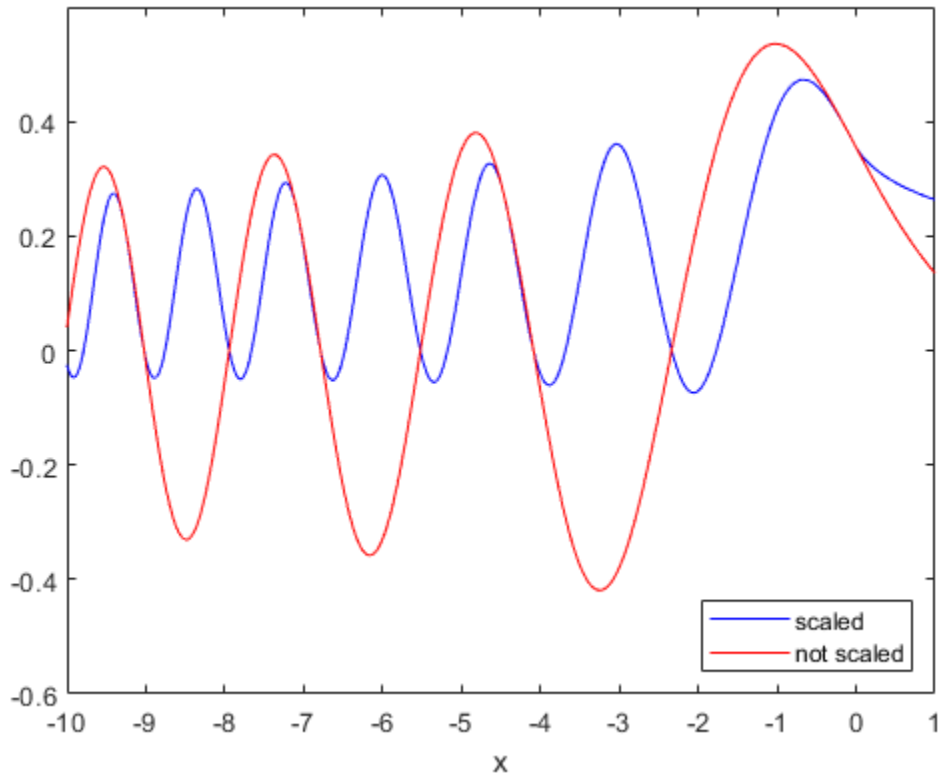
```
x = -10:0.01:1;
```

Calculate the scaled and unscaled Airy function.

```
scaledAi = airy(0,x,1);
noscaleAi =  airy(0,x,0);
```

Plot the real part of each result.

```
rscaled = real(scaledAi);
rnoscale = real(noscaleAi);
figure
plot(x,rscaled,'-b',x,rnoscale,'-r')
axis([-10 1 -0.60 0.60])
xlabel('x')
legend('scaled','not scaled','Location','SouthEast')
```



Input Arguments

Z — System variable
vector | matrix | N-D Array

System variable, specified as a real or complex vector, matrix, or N-D array.

Data Types: `single` | `double`
Complex Number Support: Yes

k — Type of Airy function
`0` (default) | `1` | `2` | `3`

Type of Airy function, specified as one of four values.

k	Returns
0	Airy function, $Ai(Z)$, which is the same as `airy(Z)`.
1	First derivative of Airy function, $Ai'(Z)$.
2	Airy function of the second kind, $Bi(Z)$
3	First derivative of Airy function of the second kind, $Bi'(Z)$

Data Types: `single` | `double`

`scale` — Scaling option
`0` (default) | `1`

Scaling option, specified as `0` or `1`. Use `scale = 1` to enable the scaling of `Z`. The values you specify for `k` and `scale` determine the scaling function `airy` applies to `Z`.

scale	k	Scaling applied to output
0	Any	None
1	0 or 1	$e^{\frac{2}{3}Z^{(3/2)}}$
1	2 or 3	$e^{-\left\lvert\frac{2}{3}Re(Z^{(3/2)})\right\rvert}$

Data Types: `single` | `double`

Output Arguments

`W` — Airy function of `Z`
vector | matrix | N-D Array

Airy function of `Z`, returned as an array the same size as `Z`.

More About

Airy Functions

The Airy functions form a pair of linearly independent solutions to

$$\frac{d^2W}{dZ^2} - ZW = 0.$$

The relationship between the Airy and modified Bessel functions is

$$Ai(Z) = \left[\frac{1}{\pi}\sqrt{\frac{Z}{3}}\right]K_{1/3}(\zeta)$$

$$Bi(Z) = \sqrt{\frac{Z}{3}}\ \left[I_{-1/3}(\zeta) + I_{1/3}(\zeta)\right],$$

where

$$\zeta = \frac{2}{3}Z^{3/2}.$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Always returns a complex result.
- Strict single-precision calculations are not supported. In the generated code, single-precision inputs produce single-precision outputs. However, variables inside the function might be double-precision.

See Also

`besselh` | `besseli` | `besselj` | `besselk` | `bessely`

Introduced before R2006a

align

Align UI components and graphics objects

Note

Use this function only with GUIDE, or with apps created using the `figure` function.

Syntax

```
align(HandleList,'HorizontalAlignment','VerticalAlignment')
Positions = align(HandleList, 'HorizontalAlignment',
'VerticalAlignment')
Positions = align(CurPositions, 'HorizontalAlignment',
'VerticalAlignment')
```

Description

`align(HandleList,'HorizontalAlignment','VerticalAlignment')` aligns the `uicontrol` and `axes` objects in `HandleList`, a vector of handles, according to the options `HorizontalAlignment` and `VerticalAlignment`. The following tables show the possible values for `HorizontalAlignment` and `VerticalAlignment`.

`HorizontalAlignment`	Definition
`'None'`	No horizontal alignment
`'Left'`	Aligns the left edges of the objects with the left edge of the bounding box that encloses the objects
`'Center'`	Shifts objects to center their positions to the average of the extreme *x*-values of the group
`'Right'`	Aligns the right edges of the objects with the right edge of the bounding box that encloses the objects

HorizontalAlignment	Definition
'Distribute'	Equalizes *x*-distances between all objects within the span of the extreme x-values
'Fixed'	Spaces objects to have a specified number of points between them in the *x*-direction

VerticalAlignment	Definition
'None'	No vertical alignment
'Top'	Aligns the top edges of the objects with the top edge of the bounding box that encloses the objects
'Middle'	Shifts objects to center their positions to the average of the extreme *y*-values of the group
'Bottom'	Aligns the bottom edges of the objects with the bottom edge of the bounding box that encloses the objects
'Distribute'	Equalizes *y*-distances between all objects within the span of the extreme *y*-values
'Fixed'	Spaces objects to have a specified number of points between them in the *y*-direction

Aligning objects does not change their absolute sizes. All alignment options align the objects within the bounding box that encloses the objects. `Distribute` and `Fixed` align objects to the bottom left of the bounding box. `Distribute` evenly distributes the objects while `Fixed` distributes the objects with a fixed distance (in points) between them. When you specify both horizontal and vertical distance together, the keywords `'HorizontalAlignment'` and `'VerticalAlignment'` are not necessary.

If you use `Fixed` for `HorizontalAlignment` or `VerticalAlignment`, you must also specify the distance, in points, where 72 points equals 1 inch. For example:

```
align(HandleList,'Fixed',Distance,'VerticalAlignment')
```

distributes the specified components `Distance` points horizontally and aligns them vertically as specified.

```
align(HandleList,'HorizontalAlignment','Fixed',Distance)
```

aligns the specified components horizontally as specified and distributes them `Distance` points vertically.

```
align(HandleList,'Fixed',HorizontalDistance,...
      'Fixed',VerticalDistance)
```

distributes the specified components `HorizontalDistance` points horizontally and distributes them `VerticalDistance` points vertically.

`Positions = align(HandleList, 'HorizontalAlignment', 'VerticalAlignment')` returns updated positions for the specified objects as a vector of `Position` vectors. The position of the objects on the figure does not change.

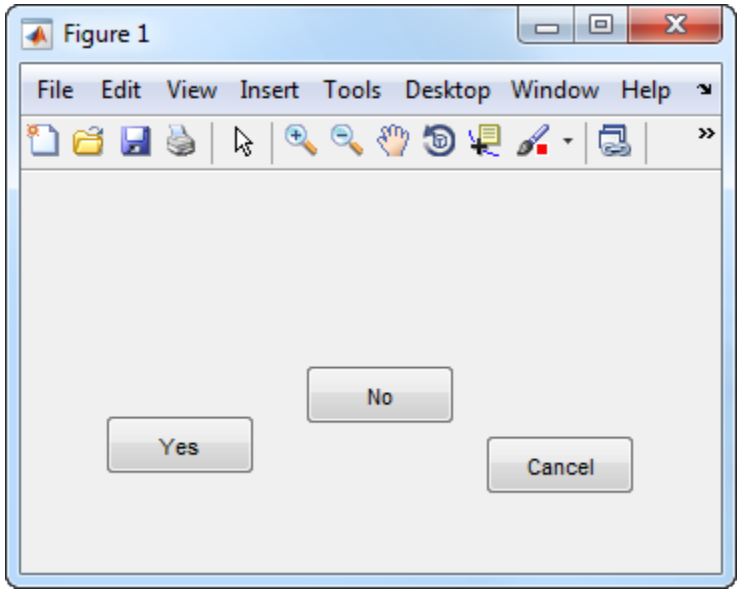
`Positions = align(CurPositions, 'HorizontalAlignment', 'VerticalAlignment')` returns updated positions for the objects whose positions are contained in `CurPositions`, where `CurPositions` is a vector of `Position` vectors. The position of the objects on the figure does not change.

Examples

Align a Row of Buttons

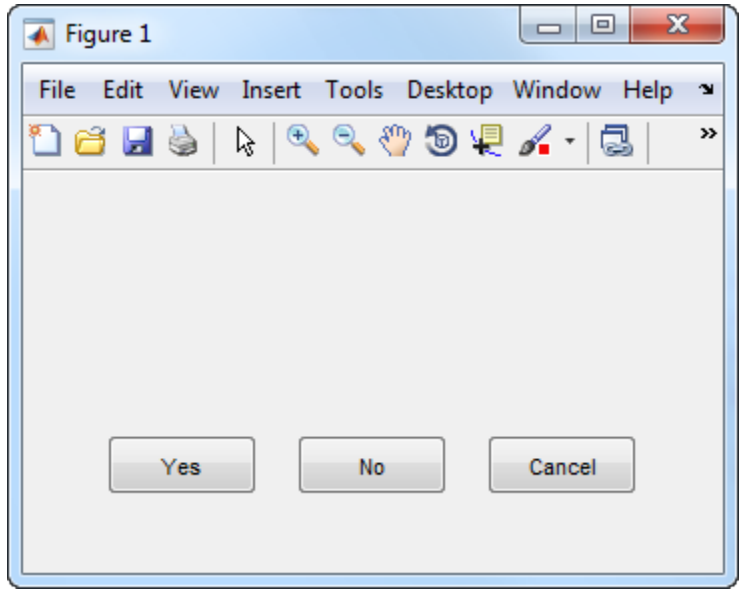
Create a UI window containing three buttons that are roughly in a row.

```
f = figure('Position',[100 100 350 200]);
u1 = uicontrol('Parent',f,'Position',[43 50 75 30],'String','Yes');
u2 = uicontrol('Parent',f,'Position',[143 75 75 30],'String','No');
u3 = uicontrol('Parent',f,'Position',[233 40 75 30],'String','Cancel');
```



Align the bottom edges of the buttons, and equalize the horizontal spacing between the buttons.

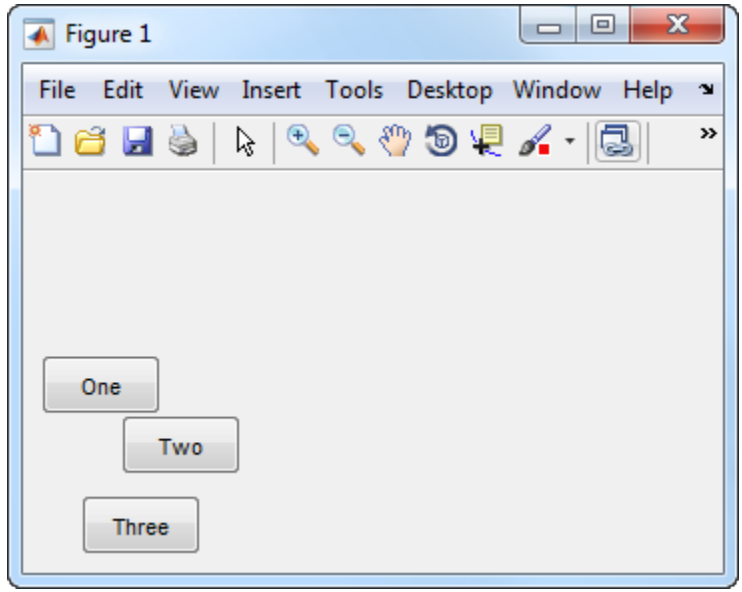
```
align([u1 u2 u3],'distribute','bottom');
```



Align a Vertical Stack of Buttons

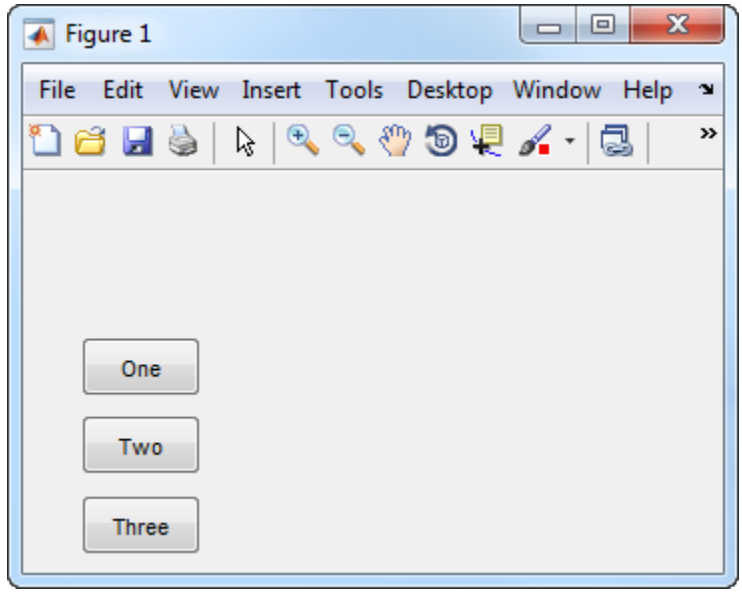
Create a UI window containing a vertical stack of buttons.

```
f = figure('Position',[100 100 350 200])
u1 = uicontrol(f,'Position',[10 80 60 30],'String','One');
u2 = uicontrol(f,'Position',[50 50 60 30],'String','Two');
u3 = uicontrol(f,'Position',[30 10 60 30],'String','Three');
```



Align the button centers and set the space between buttons to seven points.

```
align([u1 u2 u3],'Center','Fixed',7);
```



See Also

`figure` | `uicontrol` | `uistack`

alim

Set or query axes alpha limits

Syntax

```
alpha_limits = alim
alim([amin amax])
alim_mode = alim('mode')
alim('alim_mode')
alim(ax,...)
```

Description

`alpha_limits = alim` returns the alpha limits (`ALim` property) of the current axes.

`alim([amin amax])` sets the alpha limits to the specified values. `amin` is the value of the data mapped to the first alpha value in the alphamap, and `amax` is the value of the data mapped to the last alpha value in the alphamap. Data values in between are linearly interpolated across the alphamap, while data values outside are clamped to either the first or last alphamap value, whichever is closest.

`alim_mode = alim('mode')` returns the alpha limits mode (`ALimMode` property) of the current axes.

`alim('alim_mode')` sets the alpha limits mode on the current axes. `alim_mode` can be

- `auto` — MATLAB automatically sets the alpha limits based on the alpha data of the objects in the axes.
- `manual` — MATLAB does not change the alpha limits.

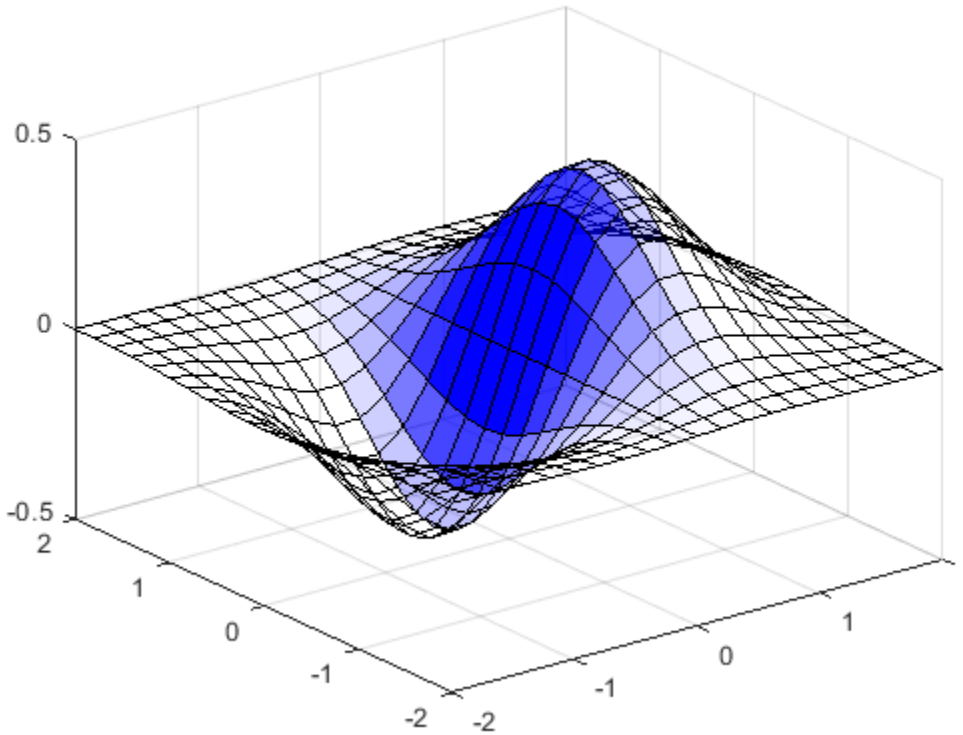
`alim(ax,...)` operates on the specified `Axes` object or `GeographicAxes` object.

Examples

Set Alpha Limits

Plot a surface using the gradient of z as the alphamap. Adjust the alpha limits to see only where the gradient is between 0 and 0.15.

```
[x,y] = meshgrid(-2:.2:2);
z = x.*exp(-x.^2-y.^2);
surf(x,y,z+.001,'FaceAlpha','flat',...
    'AlphaDataMapping','scaled',...
    'AlphaData',gradient(z),...
    'FaceColor','blue');
alim([0 .15])
```



See Also

Functions
`alpha` | `alphamap` | `caxis`

Properties
Axes

Introduced before R2006a

all

Determine if all array elements are nonzero or `true`

Syntax

```
B = all(A)
B = all(A,'all')
B = all(A,dim)
B = all(A,vecdim)
```

Description

`B = all(A)` tests along the first array dimension of `A` whose size does not equal 1, and determines if the elements are all nonzero or logical `1` (`true`). In practice, `all` is a natural extension of the logical AND operator.

- If `A` is a vector, then `all(A)` returns logical `1` (`true`) if all the elements are nonzero and returns logical `0` (`false`) if one or more elements are zero.
- If `A` is a nonempty matrix, then `all(A)` treats the columns of `A` as vectors and returns a row vector of logical `1`s and `0`s.
- If `A` is an empty 0-by-0 matrix, then `all(A)` returns logical `1` (`true`).
- If `A` is a multidimensional array, then `all(A)` acts along the first array dimension whose size does not equal 1 and returns an array of logical values. The size of this dimension becomes `1`, while the sizes of all other dimensions remain the same.

`B = all(A,'all')` tests over all elements of `A`. This syntax is valid for MATLAB versions R2018b and later.

`B = all(A,dim)` tests elements along dimension `dim`. The `dim` input is a positive integer scalar.

`B = all(A,vecdim)` tests elements based on the dimensions specified in the vector `vecdim`. For example, if `A` is a matrix, then `all(A,[1 2])` tests over all elements in `A`, since every element of a matrix is contained in the array slice defined by dimensions 1 and 2.

Examples

Test Matrix Columns

Create a 3-by-3 matrix, and then test each column for all nonzero elements.

```
A = [0 0 3;0 0 3;0 0 3]
```

```
A = 3×3

     0     0     3
     0     0     3
     0     0     3
```

```
B = all(A)
```

```
B = 1x3 logical array

   0   0   1
```

Reduce a Logical Vector to a Single Condition

Create a vector of decimal values and test which values are less than 0.5.

```
A = [0.53 0.67 0.01 0.38 0.07 0.42 0.69];
B = (A < 0.5)
```

```
B = 1x7 logical array

   0   0   1   1   1   1   0
```

The output is a vector of logical values. The `all` function reduces such a vector of logical values to a single condition. In this case, `B = all(A < 0.5)` yields logical `0`.

This makes `all` particularly useful in `if` statements.

```
if all(A < 0.5)
```

```
%do something

else

%do something else

end
```

The code is executed depending on a single condition, rather than a vector of possibly conflicting conditions.

Test Arrays of Any Dimension

Create a 3-by-7-by-5 multidimensional array and test to see if all of its elements are less than 3.

```
A = rand(3,7,5) * 5;
B = all(A(:) < 3)
```

```
B = logical
   0
```

You can also test the array for elements that are greater than zero.

```
B = all(A(:) > 0)
```

```
B = logical
   1
```

The syntax `A(:)` turns the elements of `A` into a single column vector, so you can use this type of statement on an array of any size.

Test Matrix Rows

Create a 3-by-3 matrix.

```
A = [0 0 3;0 0 3;0 0 3]
```

```
A = 3×3

     0     0     3
     0     0     3
     0     0     3
```

Test the rows of A for all nonzero elements by specifying `dim = 2`.

```
B = all(A,2)
```

```
B = 3x1 logical array

   0
   0
   0
```

Nonzero Elements in Array Page

Create a 3-D array and determine if all elements in each page of data (rows and columns) are zero.

```
A(:,:,1) = [2 1; 3 5];
A(:,:,2) = [0 0; 0 0];
A(:,:,3) = [-2 9; 4 1];
B = all(A,[1 2])
```

```
B = 1x1x3 logical array
B(:,:,1) =

   1


B(:,:,2) =

   0


B(:,:,3) =

   1
```

Input Arguments

A — Input Array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

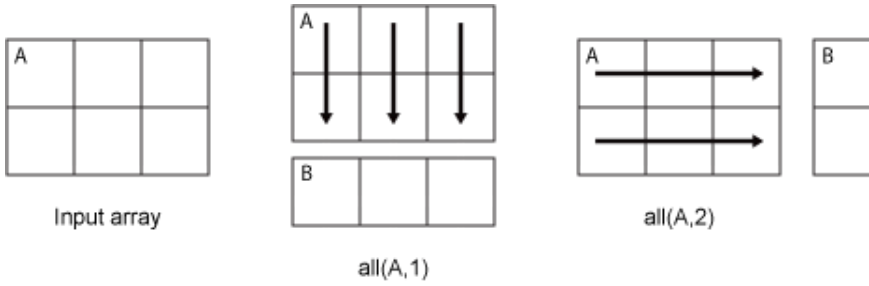
dim — Dimension to operate along

positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a two-dimensional input array, `A`:

- `all(A,1)` works on successive elements in the columns of `A` and returns a row vector of logical values.
- `all(A,2)` works on successive elements in the rows of `A` and returns a column vector of logical values.



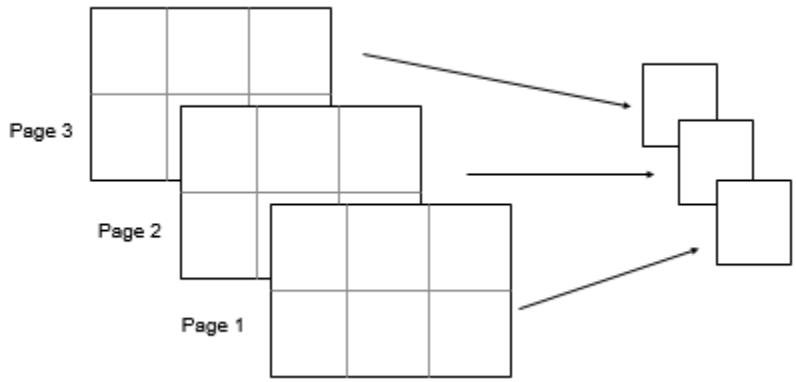
Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

vecdim — Vector of dimensions

vector of positive integers

Vector of dimensions, specified as a vector of positive integers. Each element represents a dimension of the input array. The lengths of the output in the specified operating dimensions are 1, while the others remain the same.

Consider a 2-by-3-by-3 input array, `A`. Then `all(A,[1 2])` returns a 1-by-1-by-3 array whose elements indicate nonzero values for each page of `A`.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

B — Logical array
scalar | vector | matrix | multidimensional array

Logical array, returned as a scalar, vector, matrix, or multidimensional array. The dimension of `A` acted on by `all` has size `1` in `B`.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`and` | `any` | `colon` | `prod` | `sum`

Topics

“Reduce Logical Arrays to Single Value”

Introduced before R2006a

allchild

Find all children of specified objects

Syntax

```
child_handles = allchild(handle_list)
```

Description

`child_handles = allchild(handle_list)` returns the list of all children (including ones with hidden handles) for each handle. If `handle_list` is a single element, `allchild` returns the output in a vector. If `handle_list` is a vector of handles, the output is a cell array.

Examples

Compare the results these two statements return:

```
axes
get(gca,'Children')
allchild(gca)
```

See Also

`findall` | `findobj`

Introduced before R2006a

alpha

Add transparency to objects in axes

Syntax

```
alpha value
alpha alphadata
alpha facealpha
alpha alphadatamapping

alpha(obj, ___ )
alpha(ax, ___ )
```

Description

`alpha value` sets the face transparency for graphics objects in the current axes that support transparency. Specify `value` as a scalar value between 0 (transparent) and 1 (no transparency), `'clear'`, or `'opaque'`.

`alpha alphadata` varies the transparency across all image, patch, chart surface, and primitive surface objects in the axes. This option sets the corresponding alpha data properties for the objects.

`alpha facealpha` specifies whether to use flat or interpolated transparency across the faces of all patch, chart surface, and primitive surface objects in the axes. This option sets the `FaceAlpha` property for the objects.

`alpha alphadatamapping` controls the interpretation of the alpha data values for all image, patch, chart surface, and primitive surface objects in the axes. This option sets the `AlphaDataMapping` property for the objects.

`alpha(obj, ___ )` sets the face transparency for the objects specified by `obj`. The option, `obj`, can precede any of the input arguments in the previous syntaxes. Use single quotes around input arguments that are character arrays, for example, `alpha(obj,'opaque')`.

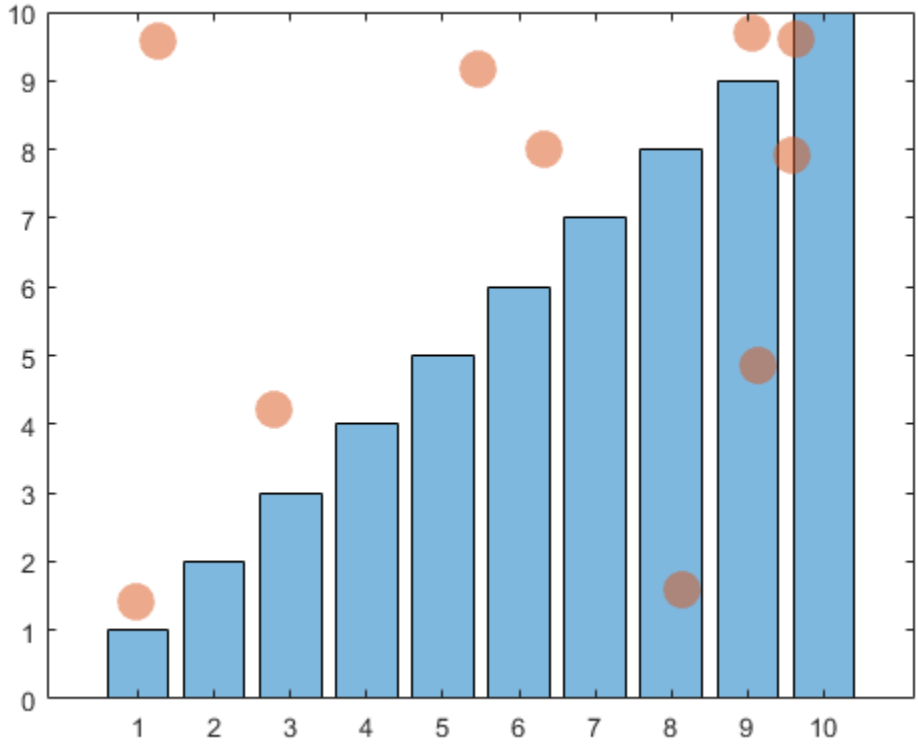
`alpha(ax, ___ )` sets the face transparency for the graphics objects in the axes specified by `ax`, instead of the current axes. Specify `ax` as the first input argument. You cannot specify both the `ax` and `obj` inputs.

Examples

Set Transparency for All Graphics Objects in Axes

Create a bar chart and a scatter chart. Set the face transparency for both the bar series and scatter series object to `0.5`.

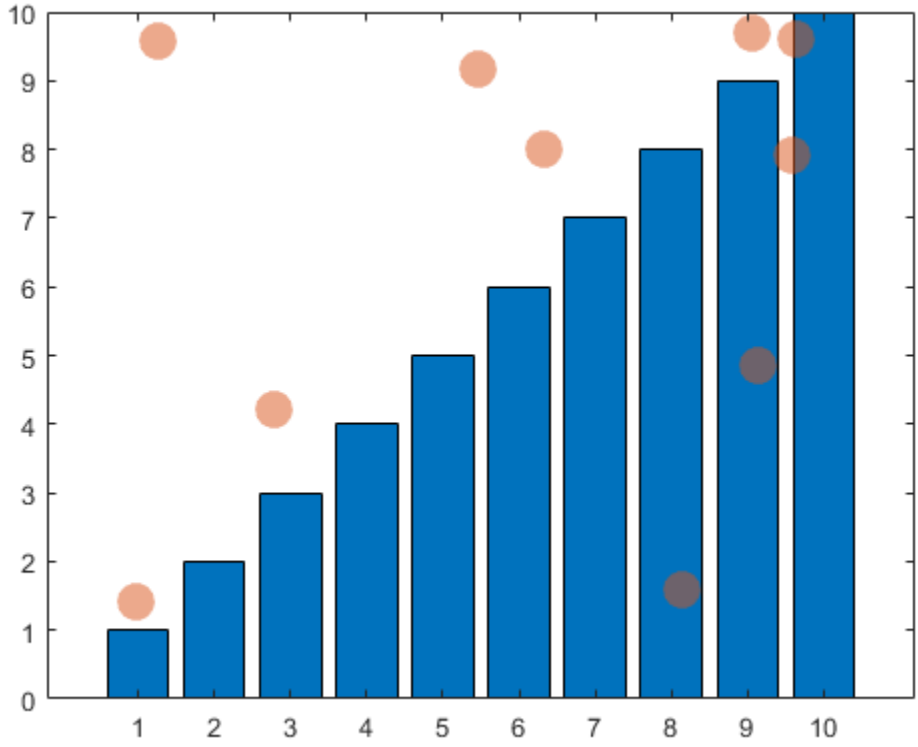
```
bar(1:10)
hold on
scatter(10*rand(10,1),10*rand(10,1),'filled','SizeData',200)
hold off
alpha(.5)
```



Set Transparency for Specific Graphics Object

Create a bar chart and a scatter chart. Set the transparency for the scatter series object to `0.5` without affecting the bar series object.

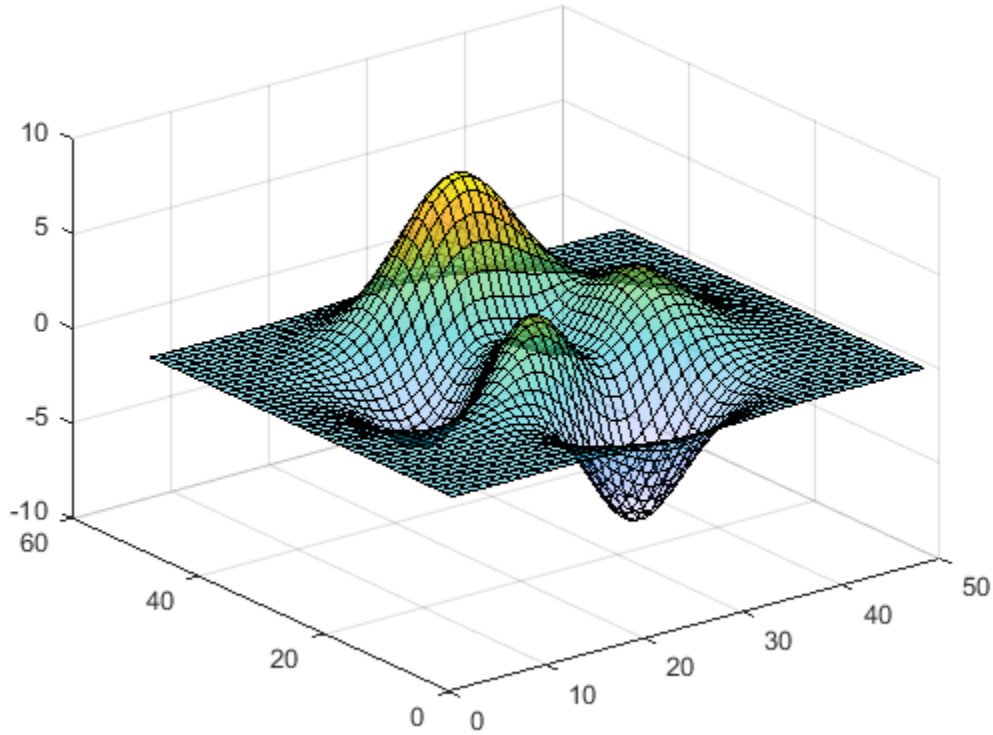
```
b = bar(1:10);
hold on
s = scatter(10*rand(10,1),10*rand(10,1),'filled','SizeData',200);
hold off
alpha(s,.5)
```



Set Surface Transparency Using *z* Values

Create a surface chart. Vary the transparency across the surface based on the *z* values.

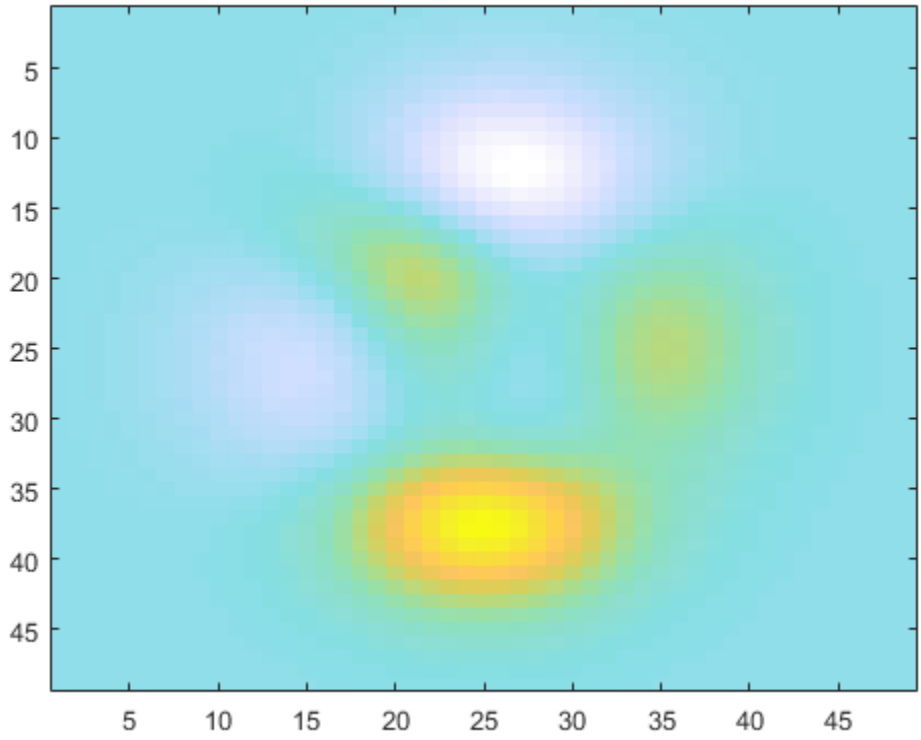
```
s = surf(peaks);
alpha(s,'z')
```



Set Mapping of Alpha Data into Alphamap

Create an image. Vary the transparency across the image based on the color values. By default, the image clamps the transparency values between 0 and 1. Values less than 0 are completely transparency and values greater than 1 are completely opaque. Instead, transform the transparency values to span the alphamap using the `alpha scaled` command.

```
imagesc(peaks)
alpha color
alpha scaled
```



Input Arguments

`value` — Face transparency value
scalar from 0 through 1 | `'clear'` | `'opaque'`

Face transparency value, specified as one of these values:

- Scalar from 0 through 1 — Set the face transparency to the specified value. A value of 0 means transparent. A value of 1 means opaque. Values between 0 and 1 are semitransparent.

- `'clear'` — Set the face transparency to 0 (transparent).
- `'opaque'` — Set the face transparency to 1 (opaque).

If the object does not have a face or if the face has no color, then the `alpha` function does not have a visual effect on the object. For example, `alpha` does not have a visual effect on scatter charts that use unfilled markers or that use markers without a face (`'*'`, `'+'`, `'.'`, and `'x'`).

This table lists the graphics objects that support transparency and the associated face transparency property that updates to the value specified.

Graphics Object	Face Transparency Property
Area objects	`FaceAlpha`
Bar objects	`FaceAlpha`
Scatter objects	`MarkerFaceAlpha`
Histogram and histogram2 objects	`FaceAlpha`
Image objects	`AlphaData`
Chart and primitive surface objects	`FaceAlpha`
Patch objects	`FaceAlpha`

`alphadata` — Option to vary transparency across object
`'x'` | `'y'` | `'z'` | `'color'` | `'rand'` | matrix

Option to vary the transparency across the object, specified as one of the values in this table. The table columns indicate how each option affects the different types of objects, depending on what objects are contained in the axes.

Option	Chart or Primitive Surface Objects	Patch Objects	Image Objects
`'x'`	Set the `AlphaData` property to be the same as the `XData` property and set the `FaceAlpha` property to `'flat'`.	Set the `FaceVertexAlphaData` property to be the same as the `XData` property and set the `FaceAlpha` property to `'flat'`.	No effect.

Option	Chart or Primitive Surface Objects	Patch Objects	Image Objects
`'y'`	Set the `AlphaData` property to be the same as the `YData` property and set the `FaceAlpha` property to `'flat'`.	Set the `FaceVertexAlphaData` property to be the same as the `YData` property and set the `FaceAlpha` property to `'flat'`.	No effect.
`'z'`	Set the `AlphaData` property to be the same as the `ZData` property and set the `FaceAlpha` property to `'flat'`.	Set the `FaceVertexAlphaData` property to be the same as the `ZData` property and set the `FaceAlpha` property to `'flat'`.	No effect.
`'color'`	Set the `AlphaData` property to be the same as the `CData` property and set the `FaceAlpha` property to `'flat'`.	Set the `FaceVertexAlphaData` property to be the same as the `FaceVertexCData` property and set the `FaceAlpha` property to `'flat'`.	Set the `AlphaData` data property to be the same as the `CData` property.
`'rand'`	Set the `AlphaData` property to a matrix of random numbers the same size as the `ZData` property and set the `FaceAlpha` property to `'flat'`.	Set the `FaceVertexAlphaData` property to a matrix of random numbers the same size as the `ZData` property and set the `FaceAlpha` property to `'flat'`.	Set the `AlphaData` data property to a matrix of random numbers the same size as the `CData` property.

Option	Chart or Primitive Surface Objects	Patch Objects	Image Objects
Matrix Use the function form of the command, for example, `alpha([.1 .2; .3 .4])`.	Set the `AlphaData` property to the matrix and set the `FaceAlpha` property to `'flat'`. The matrix must be the same size as the `CData` property of the surface.	Set the `FaceVertexAlphaData` property to the matrix and set the `FaceAlpha` property to `'flat'`. The matrix must be the same size as the `FaceVertexCData` property of the patch.	Set the `AlphaData` data property to the matrix. The matrix must be the same size as the `CData` property of the image.

`facealpha` — Option for flat or interpolated face transparency
`'flat'` | `'interp'` | `'texture'`

Option for flat or interpolated face transparency, specified as one of the values in this table. The table columns indicate how each option affects the different types of objects, depending on what objects are contained in the axes.

Option	Chart or Primitive Surface Objects	Patch Objects
`'flat'`	Set the `FaceAlpha` property to `'flat'`. The `AlphaData` property must be an array the same size as the `CData` property. The `FaceColor` property must be set to `'flat'`.	Set the `FaceAlpha` property to `'flat'`. The `FaceVertexAlphaData` property must be a column vector with length equal to the number of faces in the `Faces` property.
`'interp'`	Set the `FaceAlpha` property to `'interp'`. The `AlphaData` property must be an array the same size as the `CData` property. The `FaceColor` property must be set to `'interp'`.	Set the `FaceAlpha` property to `'interp'`. The `FaceVertexAlphaData` property must be a column vector with length equal to the number of vertices in the `Vertices` property.

Option	Chart or Primitive Surface Objects	Patch Objects
`'texture'`	Set the `FaceAlpha` property to `'texturemap'`. The `FaceColor` property must be set to `'texturemap'`.	No effect.

`alphadatamapping` — Interpretation of alpha data values
`'none'` | `'direct'` | `'scaled'`

Interpretation of alpha data values, specified as `'none'`, `'direct'`, or `'scaled'`. This option sets the `AlphaDataMapping` property for image, patch, and surface objects in the axes.

- `'none'` — Interpret alpha data values as transparency values. A value of 0 or less is completely transparent. A value of 1 or greater is opaque. Values between 0 and 1 are semitransparent.
- `'direct'` — Interpret the values as indices into the figure's alphamap. Fix values with a decimal portion to the nearest lower integer. The `Alphamap` property of the figure contains the alphamap.
- `'scaled'` — Transform the alpha data values to span the portion of the figure's alphamap indicated by the `ALim` property of the axes. Linearly map the data values to alpha values.

`obj` — Graphics objects that support transparency
scalar | vector

Graphics objects that support transparency, specified as a scalar or vector. Use this option if you want to control which objects `alpha` affects. Otherwise, `alpha` affects all objects in the axes that support transparency.

Specify one or more of these types of objects:

- Area object
- Bar series object
- Scatter series object
- Histogram or histogram2 object

- Image object
- Patch object
- Chart or primitive surface object

Note Not all objects support the `alphadata`, `facealpha`, and `alphadatamapping` input arguments. See the syntax descriptions for a list of objects that each option supports.

---

ax — Axes object
axes object

Axes object. If you do not specify the axes, then `alpha` sets the transparency for graphics objects in the current axes.

See Also

`alim` | `alphamap`

Topics

"Add Transparency to Graphics Objects"

Introduced before R2006a

alphamap

Specify figure alphamap (transparency)

Syntax

```
alphamap(alpha_map)
alphamap('parameter')
alphamap('parameter',length)
alphamap('parameter',delta)
alphamap(fig,...)
alphamap(ax,...)
alpha_map = alphamap
alpha_map = alphamap(fig)
alpha_map = alphamap(ax)
alpha_map = alphamap('parameter')
```

Description

`alphamap(alpha_map)` sets the `AlphaMap` of the current figure to the specified `m`-by-1 array of alpha values, `alpha_map`. If you set the alphamap for the figure, then axes and charts in the figure use the same alphamap.

`alphamap('parameter')` creates a new alphamap or modifies the current alphamap. You can specify the following parameters:

- `'default'` — Set the `AlphaMap` property to the figure's default alphamap.
- `'rampup'` — Create a linear alphamap with increasing opacity (default `length` equals the current alphamap length).
- `'rampdown'` — Create a linear alphamap with decreasing opacity (default `length` equals the current alphamap length).
- `'vup'` — Create an alphamap that is opaque in the center and becomes more transparent linearly towards the beginning and end (default `length` equals the current alphamap length).

- `'vdown'` — Create an alphamap that is transparent in the center and becomes more opaque linearly towards the beginning and end (default `length` equals the current alphamap length).
- `'increase'` — Modify the alphamap making it more opaque (default `delta` is `.1`, added to the current values).
- `'decrease'` — Modify the alphamap making it more transparent (default `delta` is `.1`, subtracted from the current values).
- `'spin'` — Rotate the current alphamap (default `delta` is 1; `delta` must be an integer).

`alphamap('parameter',length)` creates a new alphamap with the length specified by the integer `length` (used with parameters `'rampup'`, `'rampdown'`, `'vup'`, `'vdown'`).

`alphamap('parameter',delta)` modifies the existing alphamap using the value specified by the integer `delta` (used with parameters `'increase'`, `'decrease'`, `'spin'`).

`alphamap(fig,...)` performs the operation on the alphamap of the figure identified by `fig`.

`alphamap(ax,...)` performs the operation on the alphamap of the `Axes` object, `PolarAxes` object, or `GeographicAxes` object identified by `ax`.

`alpha_map = alphamap` returns the current alphamap.

`alpha_map = alphamap(fig)` returns the current alphamap from the figure identified by `fig`.

`alpha_map = alphamap(ax)` returns the current alphamap from the `Axes` object, `PolarAxes` object, or `GeographicAxes` object identified by `ax`.

`alpha_map = alphamap('parameter')` returns the alphamap modified by the `parameter`, but does not set the `AlphaMap` property.

Examples

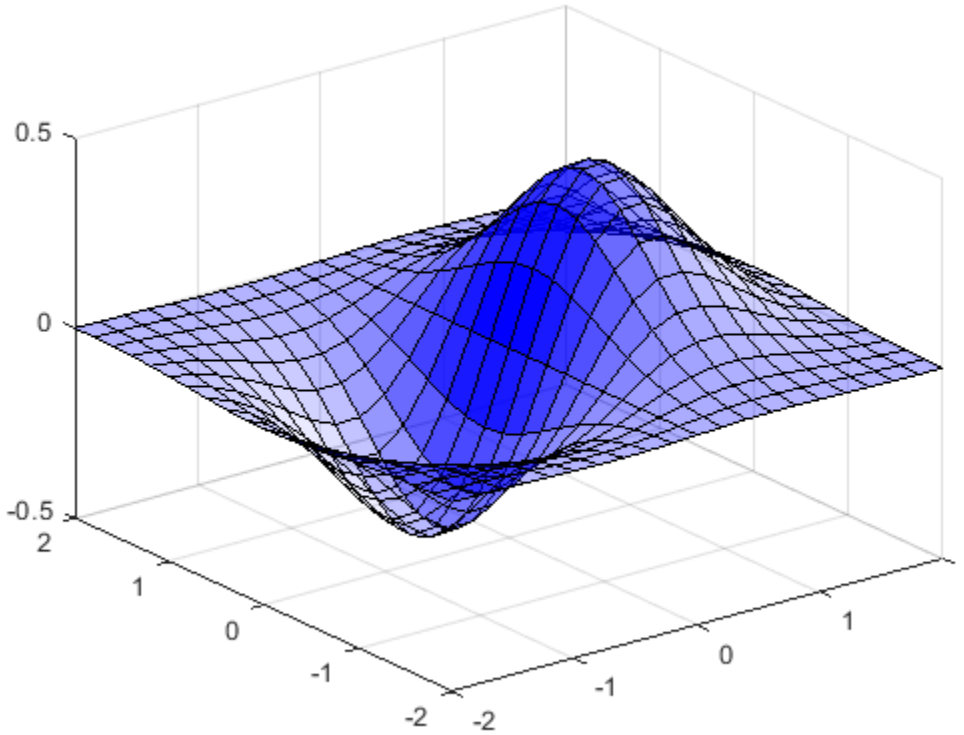
Change Alphamap for Surface Plot

Create a surface plot and change the alphamap.

```
[x,y] = meshgrid([-2:.2:2]);
z = x.*exp(-x.^2-y.^2);
```

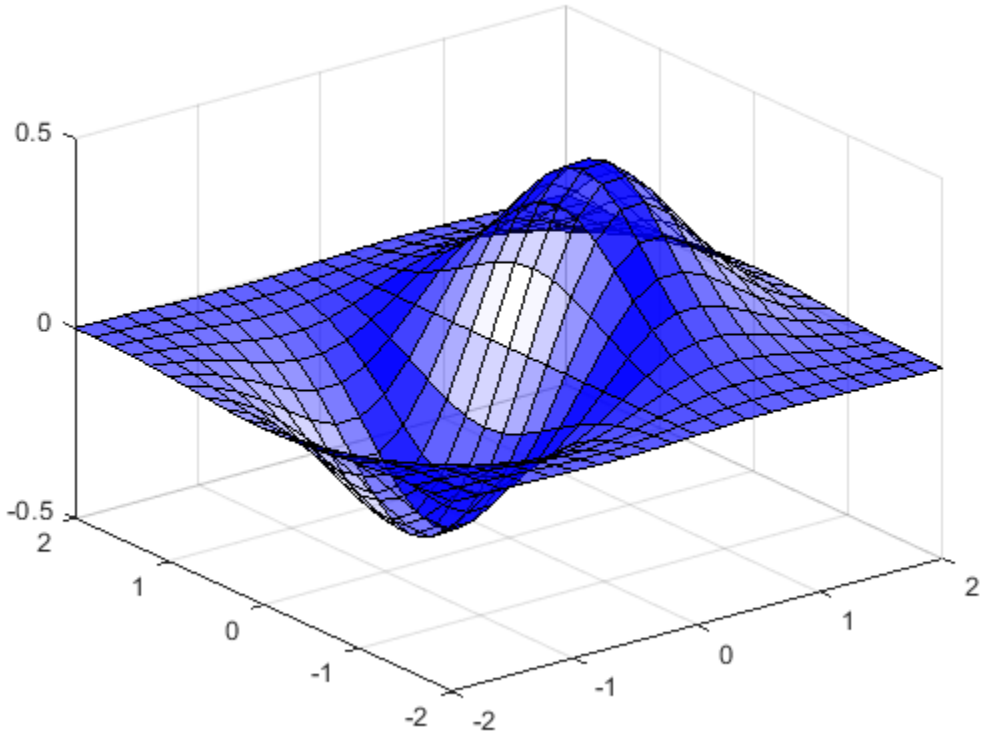
Plot the data, using the gradient of z as the alphamap.

```
figure
surf(x,y,z+.001,'FaceAlpha','flat',...
    'AlphaDataMapping','scaled',...
    'AlphaData',gradient(z),...
    'FaceColor','blue')
```



Change the alphamap to be opaque at the middle and transparent towards the ends.

```
alphamap('vup')
```



More About

Compatibility Considerations

Starting in R2018a, if you set the alphamap for a figure, then axes and charts in the figure use the same alphamap. Previously, any axes or chart that you set the alphamap for explicitly were unaffected when you set the figure alphamap. If you want an `Axes` object to use a different alphamap than the figure, then set the axes alphamap after setting the figure alphamap.

See Also

`alim` | `alpha`

Introduced before R2006a

alphaShape

Polygons and polyhedra from points in 2-D and 3-D

Description

An `alphaShape` creates a bounding area or volume that envelops a set of 2-D or 3-D points. You can manipulate the `alphaShape` object to tighten or loosen the fit around the points to create a nonconvex region. You also can add or remove points or suppress holes or regions.

After you create an `alphaShape` object, you can perform geometric queries. For example, you can determine if a point is inside the shape or you can find the number of regions that make up the shape. You also can calculate useful quantities like area, perimeter, surface area, or volume, and plot the shape for visual inspection.

Creation

To create an `alphaShape` object, use the `alphaShape` function with input arguments that define the shape's vertices. You also can specify an alpha radius and hole or region thresholds when you create the `alphaShape`.

Syntax

```
shp = alphaShape(x,y)
shp = alphaShape(x,y,z)
shp = alphaShape(P)
shp = alphaShape( ___ ,a)
shp = alphaShape( ___ ,Name,Value)
```

Description

`shp = alphaShape(x,y)` creates a 2-D alpha shape of the points `(x,y)` using the default alpha radius. The default alpha radius produces the tightest fitting alpha shape, which encloses all of the points.

`shp` represents a polygon. The polygon has no isolated points or edges, nor does it have dangling edges.

`shp = alphaShape(x,y,z)` creates a 3-D alpha shape of the points `(x,y,z)` using the default alpha radius.

`shp` represents a polyhedron. The polyhedron has the previously stated polygon traits, but it additionally does not have isolated faces or dangling faces.

`shp = alphaShape(P)` specifies points `(x,y)` or `(x,y,z)` in the columns of matrix `P`.

`shp = alphaShape( ___ ,a)` creates an alpha shape with alpha radius `a` using any of the arguments in the previous syntaxes.

`shp = alphaShape( ___ ,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. For example, you can suppress interior holes or voids using `'HoleThreshold'`.

Input Arguments

x — x-coordinates
column vector

x-coordinates of points, specified as a column vector.

Data Types: `double`

y — y-coordinates
column vector

y-coordinates of points, specified as a column vector.

Data Types: `double`

z — z-coordinates
column vector

z-coordinates of points, specified as a column vector.

Data Types: `double`

P — Point coordinates
matrix with two columns | matrix with three columns

Point coordinates, specified as a matrix with two columns (for a 2-D alpha shape) or a matrix with three columns (for a 3-D alpha shape).

- For 2-D, the columns of `P` represent `x` and `y` coordinates, respectively.
- For 3-D, the columns of `P` represent `x`, `y`, and `z` coordinates, respectively.

Data Types: `double`

a — Alpha radius
nonnegative scalar

Alpha radius, specified as a nonnegative scalar. The default alpha radius is `a = criticalAlpha(shp,'all-points')`, which is the smallest alpha radius that produces an alpha shape that encloses all points.

Specify `a = criticalAlpha(shp,'one-region')` to use the smallest alpha radius that produces an alpha shape with only one region.

The extreme values of `a` are

- `Inf`, where `alphaShape` produces the convex hull
- `0`, where `alphaShape` produces an empty alpha shape

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `shp = alphaShape(...,'HoleThreshold',10)`

`HoleThreshold` — Maximum interior holes
0 (default) | finite, nonnegative scalar

Maximum area or volume of interior holes or voids to fill in, specified as a finite, nonnegative scalar.

- For 2-D, `HoleThreshold` specifies the maximum area of interior holes to fill in.
- For 3-D, `HoleThreshold` specifies the maximum volume of interior voids to fill in. Holes extending completely through the alpha shape cannot be filled in.

When you specify both a `'HoleThreshold'` and a `'RegionThreshold'`, the application of the thresholds is order dependent. `alphaShape` fills in holes before suppressing regions.

Data Types: `double`

`RegionThreshold` — Maximum regions
0 (default) | finite, nonnegative scalar

Maximum area (2-D) or volume (3-D) of regions to suppress, specified as a finite, nonnegative scalar.

When you specify a `'HoleThreshold'` and a `'RegionThreshold'`, the application of the thresholds is order dependent. `alphaShape` fills in holes before suppressing regions.

Data Types: `double`

Properties

`Points` — Coordinates of points
matrix

Coordinates of points, specified as a matrix with two or three columns (for 2-D or 3-D point sets). These points are initially used to create the alpha shape, excluding duplicates.

Data Types: `double`

`Alpha` — Alpha radius
nonnegative scalar

Alpha radius, specified as a nonnegative scalar. The alpha radius is the radius of the alpha disk or sphere that sweeps over the points to create the alpha shape.

The default alpha radius is `a = criticalAlpha(shp,'all-points')`, which is the smallest alpha radius that produces an alpha shape enclosing all points. Specify `a = criticalAlpha(shp,'one-region')` to use the smallest alpha radius that produces an alpha shape with only one region.

The extreme values of `Alpha` have the following conditions:

- If `Alpha` is `Inf`, then `alphaShape` produces the convex hull.

- If `Alpha` is `0`, then the resulting `alphaShape` is empty.

Data Types: `double`

`HoleThreshold` — Maximum interior holes
`0` (default) | finite nonnegative scalar

Maximum area or volume of interior holes or voids to fill in, specified as a finite nonnegative scalar.

- For 2-D, `HoleThreshold` specifies the maximum area of interior holes to fill in.
- For 3-D, `HoleThreshold` specifies the maximum volume of interior voids to fill in. Holes extending completely through the 3-D alpha shape cannot be filled in.

The default value is `0`, so that `alphaShape` does not suppress any holes or voids. The application of the `HoleThreshold` and `RegionThreshold` properties is order-dependent. `alphaShape` fills in holes before suppressing regions.

Data Types: `double`

`RegionThreshold` — Maximum regions
`0` (default) | finite nonnegative scalar

Maximum area (2-D) or volume (3-D) of regions to suppress, specified as a finite nonnegative scalar.

The default value is `0`, so that `alphaShape` does not suppress any regions. The application of the `HoleThreshold` and `RegionThreshold` properties is order-dependent. `alphaShape` fills in holes before suppressing regions.

Data Types: `double`

Object Functions

alphaSpectrum	Alpha values giving distinct alpha shapes
criticalAlpha	Alpha radius defining critical transition in shape
numRegions	Number of regions in alpha shape
inShape	Determine if point is inside alpha shape
alphaTriangulation	Triangulation that fills alpha shape
boundaryFacets	Boundary facets of alpha shape
perimeter	Perimeter of 2-D alpha shape

area	Area of 2-D alpha shape
surfaceArea	Surface area of 3-D alpha shape
volume	Volume of 3-D alpha shape
plot	Plot alpha shape
nearestNeighbor	Determine nearest alpha shape boundary point

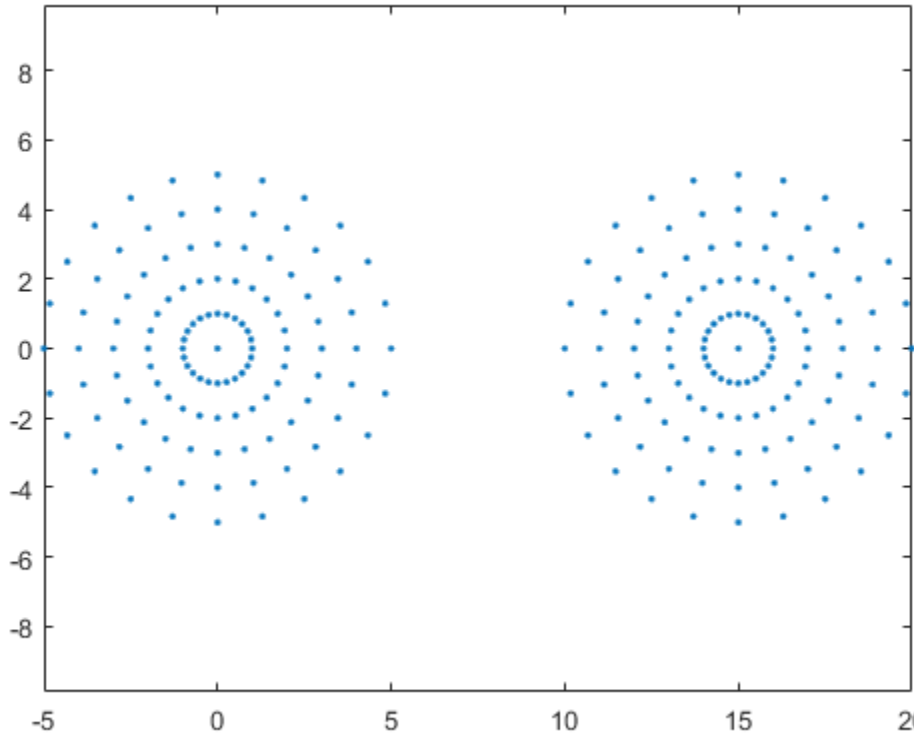
Examples

Alpha Shape from 2-D Point Cloud

Find the shape of a 2-D point cloud of data.

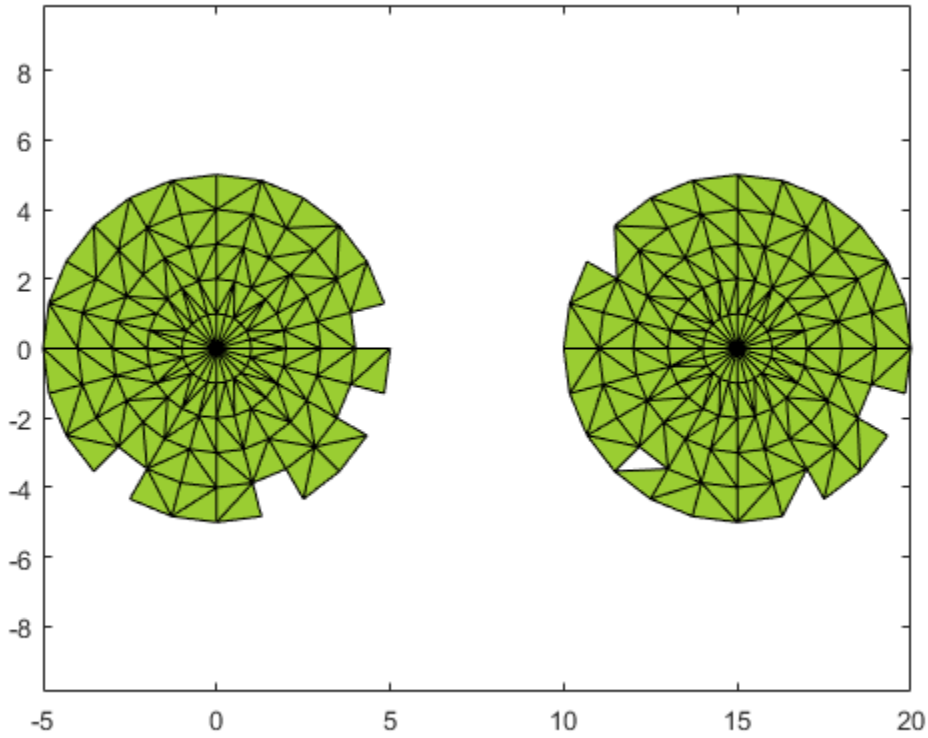
Create and plot a set of 2-D points.

```
th = (pi/12:pi/12:2*pi)';
x1 = [reshape(cos(th)*(1:5), numel(cos(th)*(1:5)),1); 0];
y1 = [reshape(sin(th)*(1:5), numel(sin(th)*(1:5)),1); 0];
x = [x1; x1+15];
y = [y1; y1];
plot(x,y,'.')
axis equal
```



Compute an alpha shape for the point set using the default alpha radius.

```
shp = alphaShape(x,y);
plot(shp)
```



Check the value of the default alpha radius.

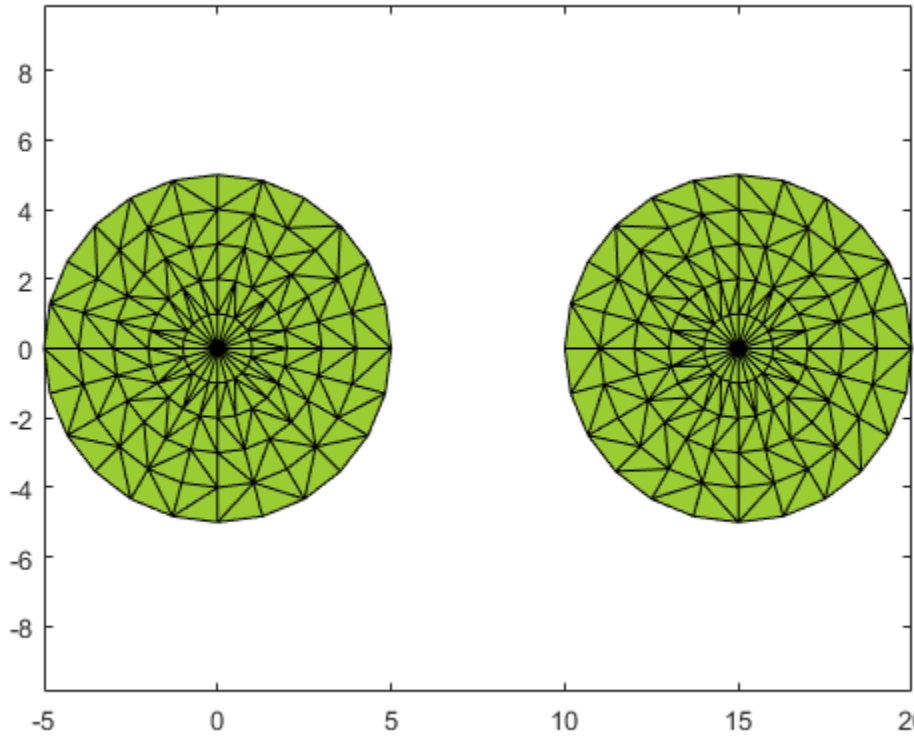
```
shp.Alpha
```

```
ans = 0.7752
```

The default alpha radius results in an alpha shape with a jagged boundary. To better capture the boundary of the point set, try a larger alpha radius.

Compute an alpha shape using an alpha value of 2.5.

```
shp.Alpha = 2.5;
plot(shp)
```



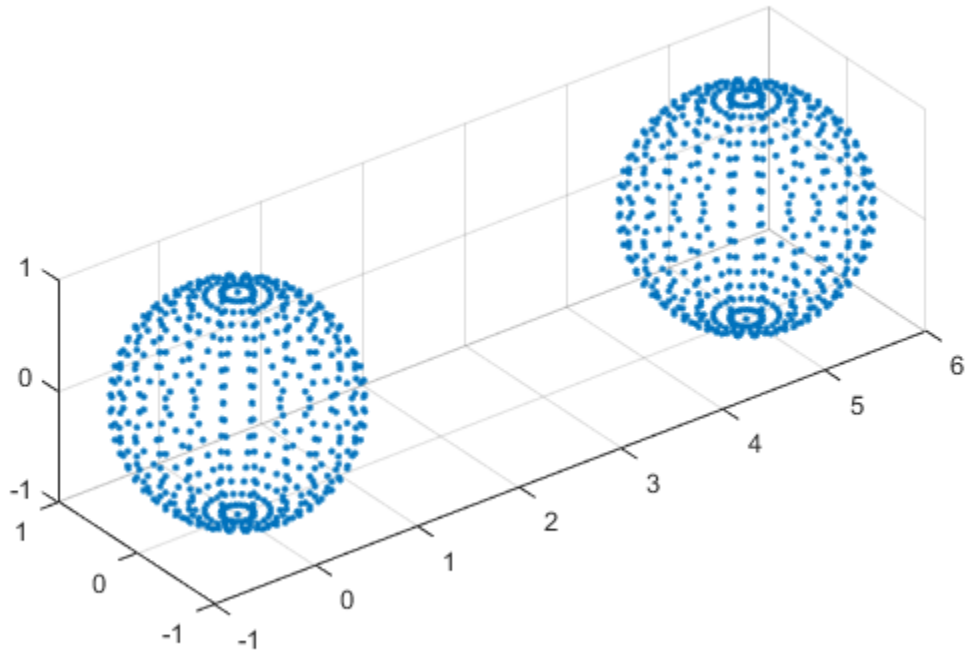
Alpha Shape from 3-D Point Cloud

Find the shape of a 3-D point cloud of data.

Create and plot a set of 3-D points.

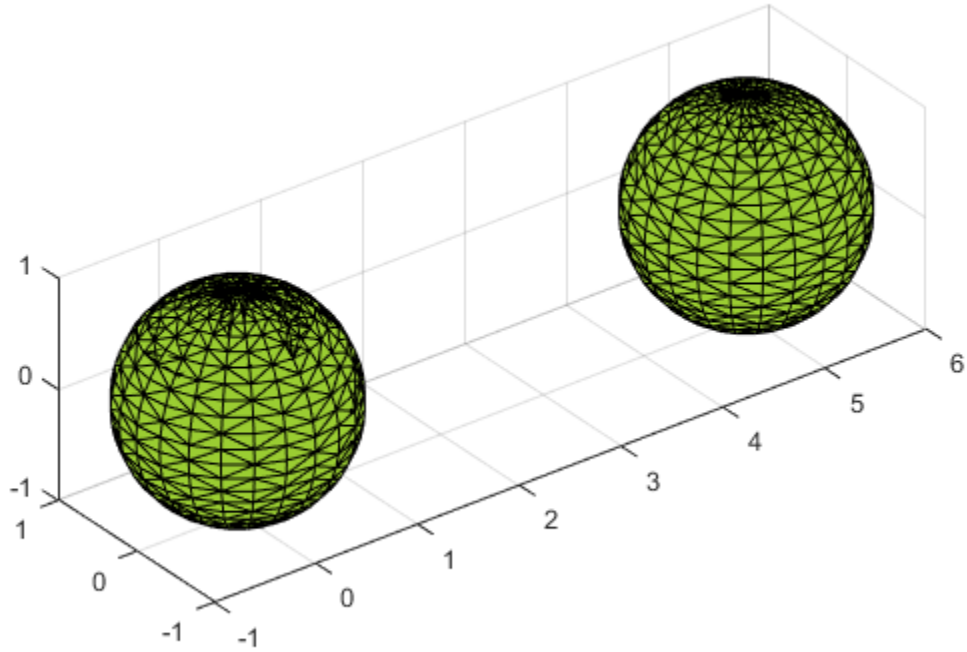
```
[x1,y1,z1] = sphere(24);
x1 = x1(:);
y1 = y1(:);
z1 = z1(:);
x2 = x1+5;
```

```
P = [x1 y1 z1; x2 y1 z1];
P = unique(P,'rows');
plot3(P(:,1),P(:,2),P(:,3),'.')
axis equal
grid on
```



Compute a 3-D alpha shape using an alpha radius of 1.

```
shp = alphaShape(P(:,1),P(:,2),P(:,3),1);
plot(shp)
axis equal
```



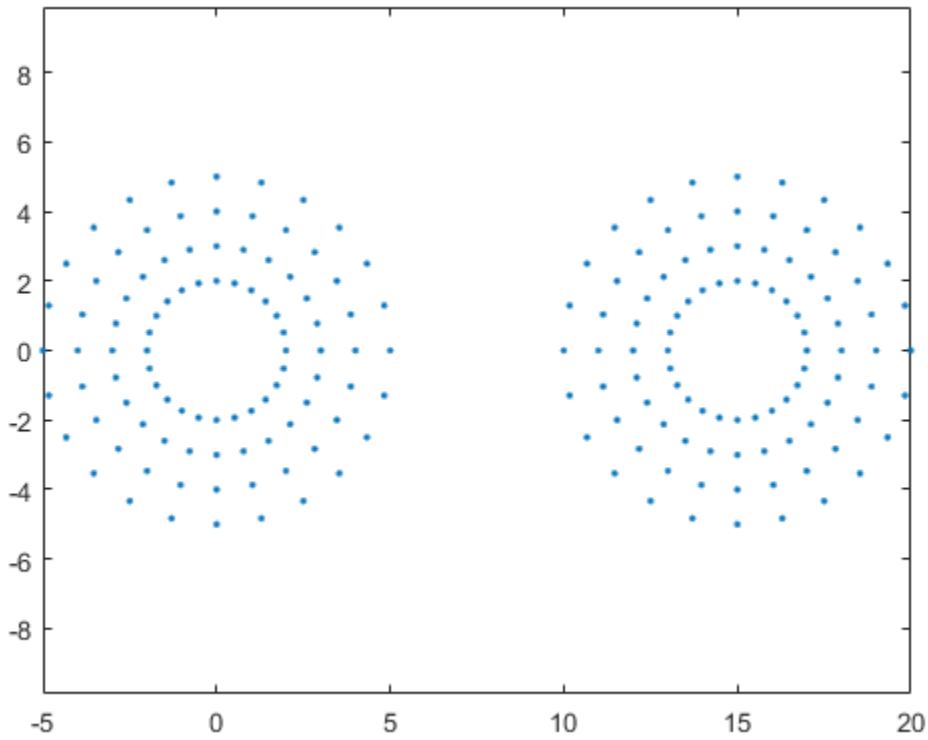
Fill Holes in 2-D Alpha Shape

Create an alpha shape by specifying its alpha radius, and fill the holes in an alpha shape.

Create and plot a 2-D set of points.

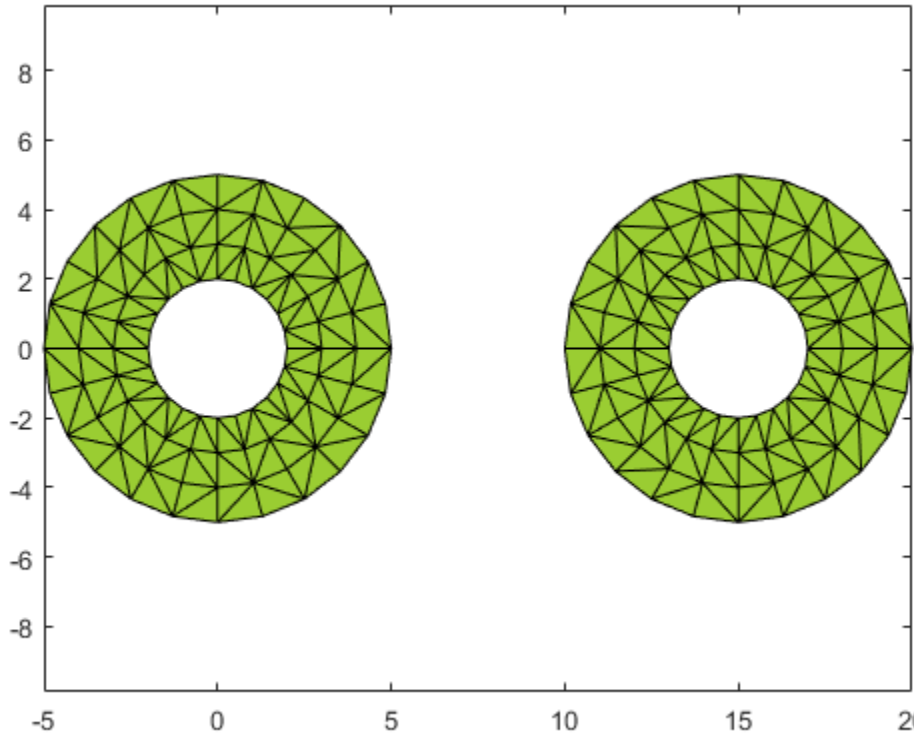
```
th = (pi/12:pi/12:2*pi)';
x1 = [reshape(cos(th)*(2:5), numel(cos(th)*(2:5)),1);];
y1 = [reshape(sin(th)*(2:5), numel(sin(th)*(2:5)),1);];
x = [x1; x1+15;];
y = [y1; y1];
```

```
plot(x,y,'.')
axis equal
```



Compute an alpha shape for the point set using an alpha radius of 1.

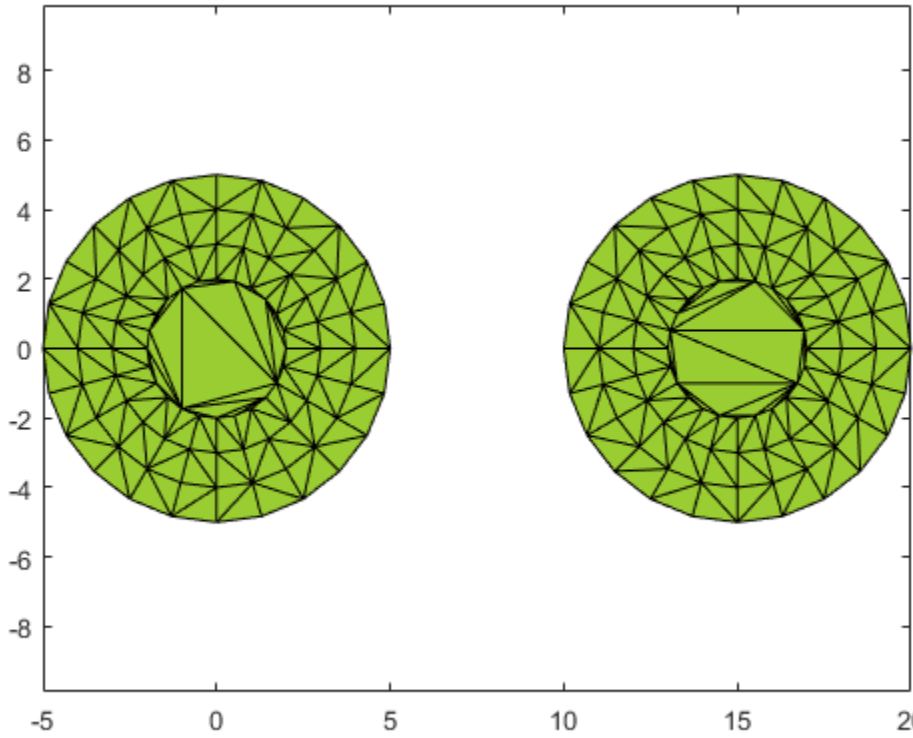
```
shp = alphaShape(x,y,1);
plot(shp)
```



An alpha radius of 1 results in an alpha shape with two regions containing holes. To suppress the small holes in the alpha shape, you can specify a `HoleThreshold` by estimating the area of the largest hole to fill. To fill all holes in the shape, you can assign an arbitrarily large value to `HoleThreshold`.

Create a new alpha shape that suppresses the holes by specifying a `HoleThreshold` of 15.

```
shp = alphaShape(x,y,1,'HoleThreshold',15);
plot(shp)
```



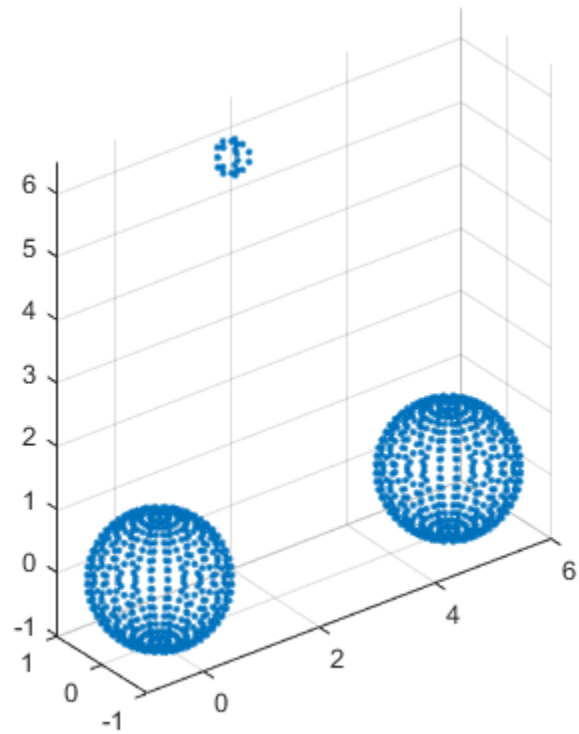
Discard Small Regions of 3-D Alpha Shape

Control the number of regions of an alpha shape by setting a region threshold.

Create and plot a set of 3-D points.

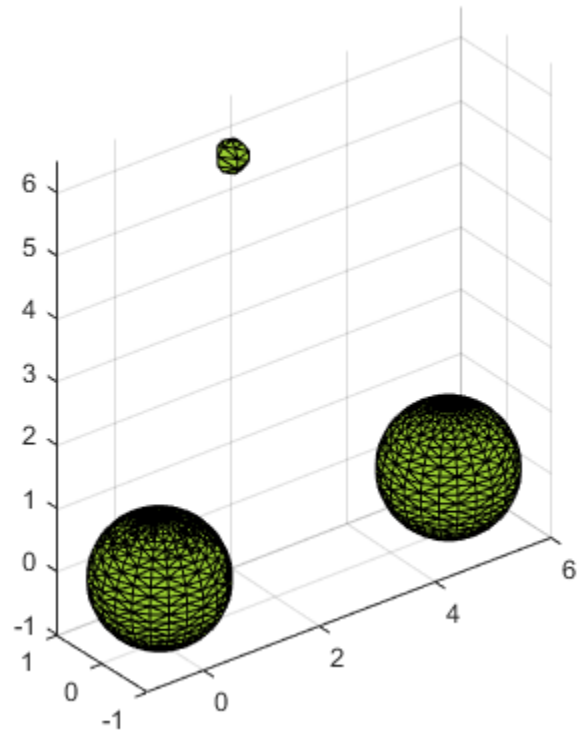
```
[x1,y1,z1] = sphere(24);
x1 = x1(:);
y1 = y1(:);
z1 = z1(:);
x2 = x1+5;
```

```
[x3,y3,z3] = sphere(5);
x3 = x3(:)+5;
y3 = y3(:);
z3 = z3(:)+25;
P = [x1 y1 z1; x2 y1 z1; 0.25*x3 0.25*y3 0.25*z3];
P = unique(P,'rows');
plot3(P(:,1),P(:,2),P(:,3),'.')
axis equal
grid on
```



Compute an alpha shape for the point set using an alpha radius of 1.

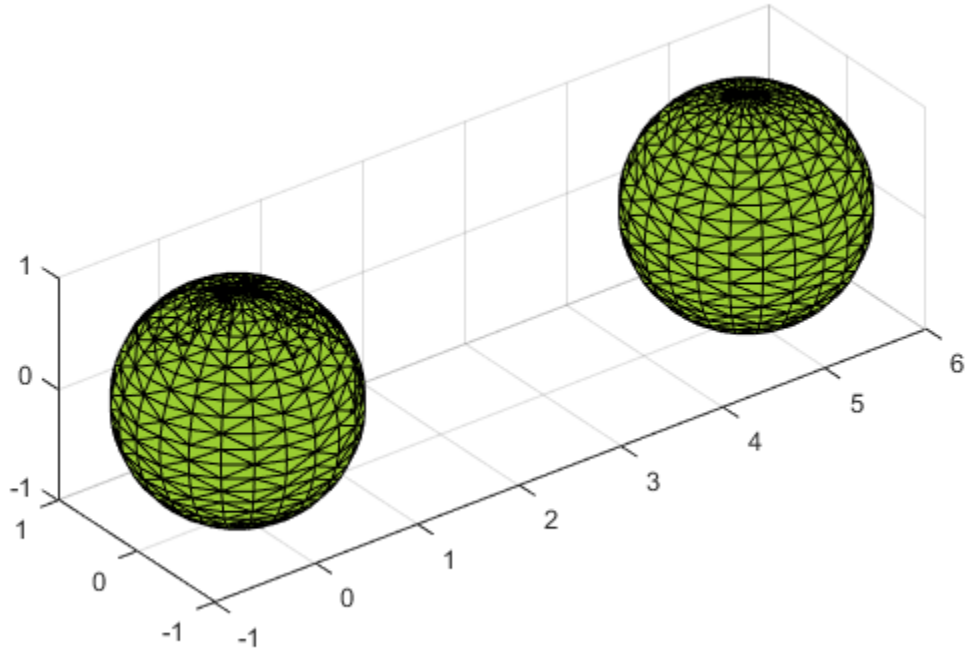
```
shp = alphaShape(P,1);
plot(shp)
axis equal
```



In this case, the alpha shape produces a small region above the two equal-sized spheres. To suppress this region, you can specify a `RegionThreshold` by estimating its volume.

Specify a `RegionThreshold` of 2. The resulting shape contains only the two larger regions.

```
shp.RegionThreshold = 2;
plot(shp)
axis equal
```



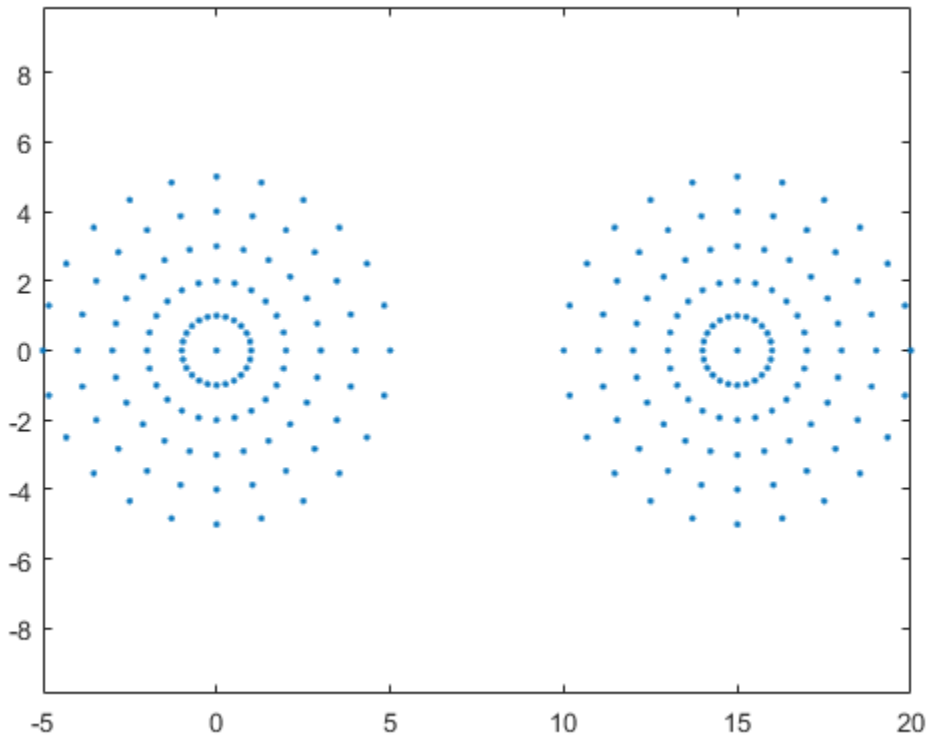
Modify Points of 2-D Alpha Shape

Add points to an existing alpha shape.

Create and plot a 2-D set of points.

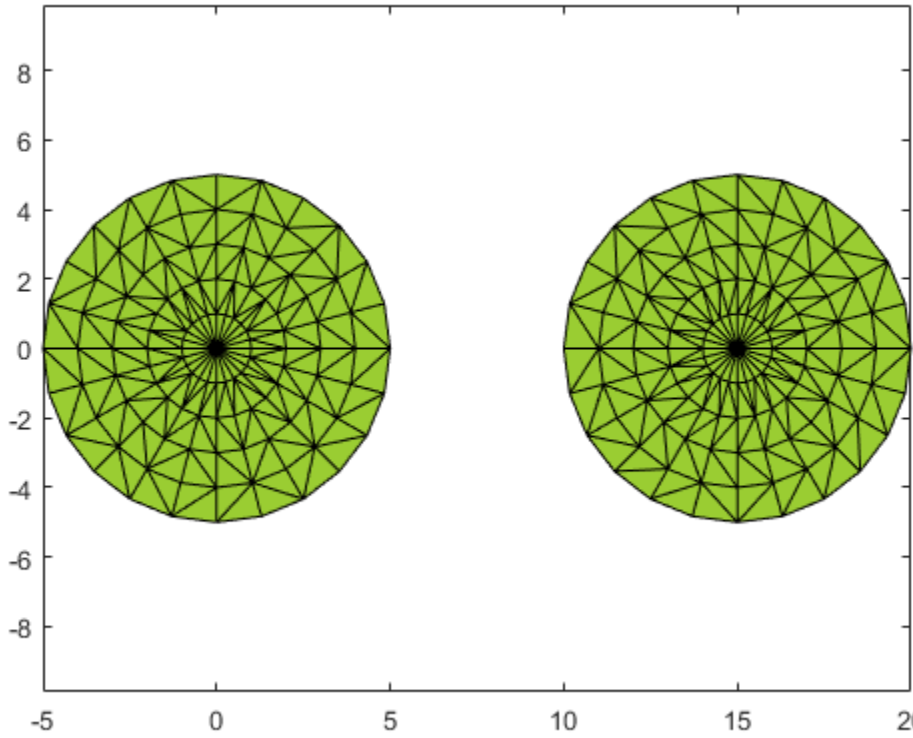
```
th = (pi/12:pi/12:2*pi)';
x1 = [reshape(cos(th)*(1:5), numel(cos(th)*(1:5)),1); 0];
y1 = [reshape(sin(th)*(1:5), numel(sin(th)*(1:5)),1); 0];
x = [x1; x1+15;];
y = [y1; y1];
```

```
plot(x,y,'.')
axis equal
```



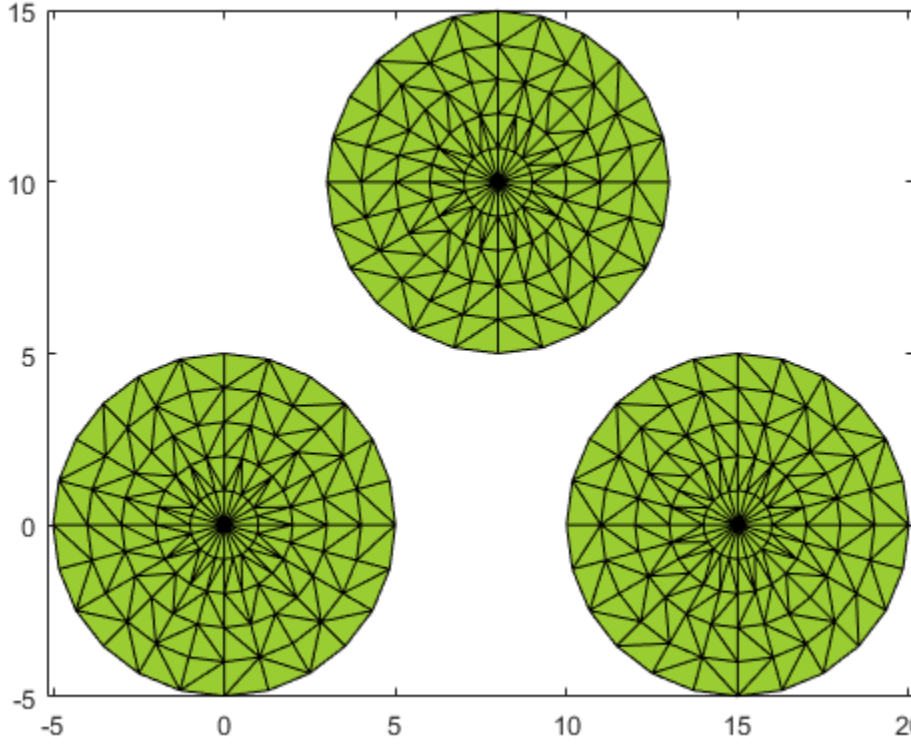
Compute an alpha shape for the point set using an alpha radius of 1. The resulting alpha shape has two regions.

```
shp = alphaShape(x,y,1);
plot(shp)
```



Now add a third region to the alpha shape by adding new points directly to the `shp.Points` matrix.

```
x3 = x1+8;
y3 = y1+10;
shp.Points(end+1,:) = [x3 y3];
plot(shp)
```



See Also

`boundary` | `convhull` | `criticalAlpha` | `delaunayTriangulation` | `triangulation` | `trisurf`

Topics

"Types of Region Boundaries"

Introduced in R2014b

alphaSpectrum

Alpha values giving distinct alpha shapes

Syntax

```
a = alphaSpectrum(shp)
```

Description

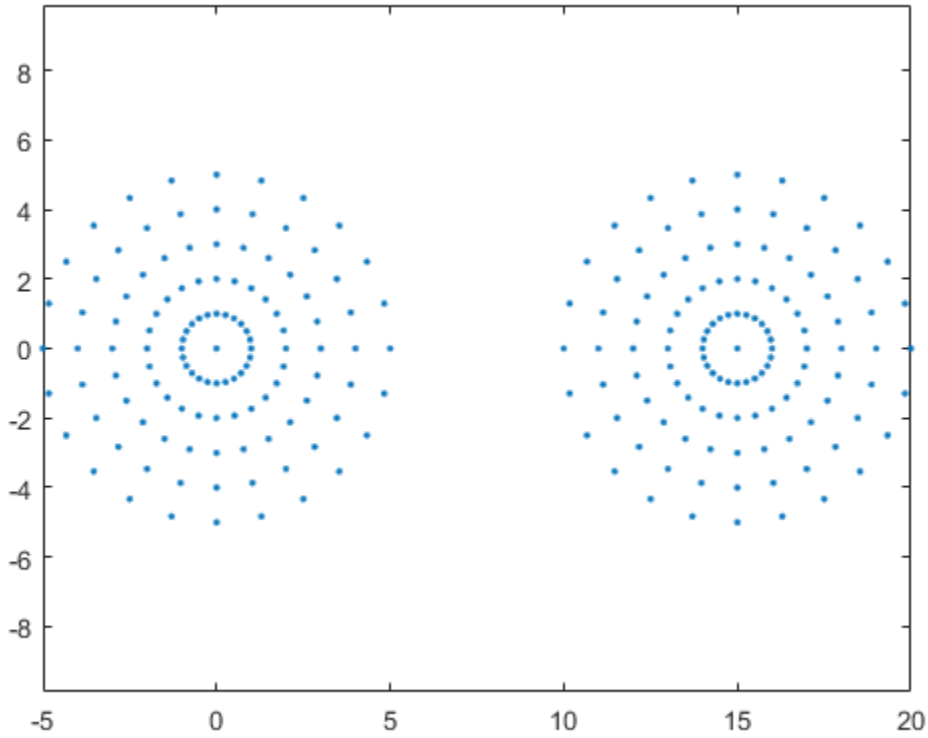
`a = alphaSpectrum(shp)` returns the values of the alpha radius that produce distinct alpha shapes. `a` is in descending sorted order. Each element in `a` represents a value of the alpha radius that results in a distinct shape. The length of `a` is equal to the number of unique shapes. Values of alpha that lie between the values in `a` do not produce unique alpha shapes.

Examples

Find Alpha Spectrum for 2-D Point Cloud

Create and plot a set of 2-D points.

```
th = (pi/12:pi/12:2*pi)';
x1 = [reshape(cos(th)*(1:5), numel(cos(th)*(1:5)),1); 0];
y1 = [reshape(sin(th)*(1:5), numel(sin(th)*(1:5)),1); 0];
x = [x1; x1+15;];
y = [y1; y1];
plot(x,y,'.')
axis equal
```



Create an alpha shape for the point cloud using the default alpha radius.

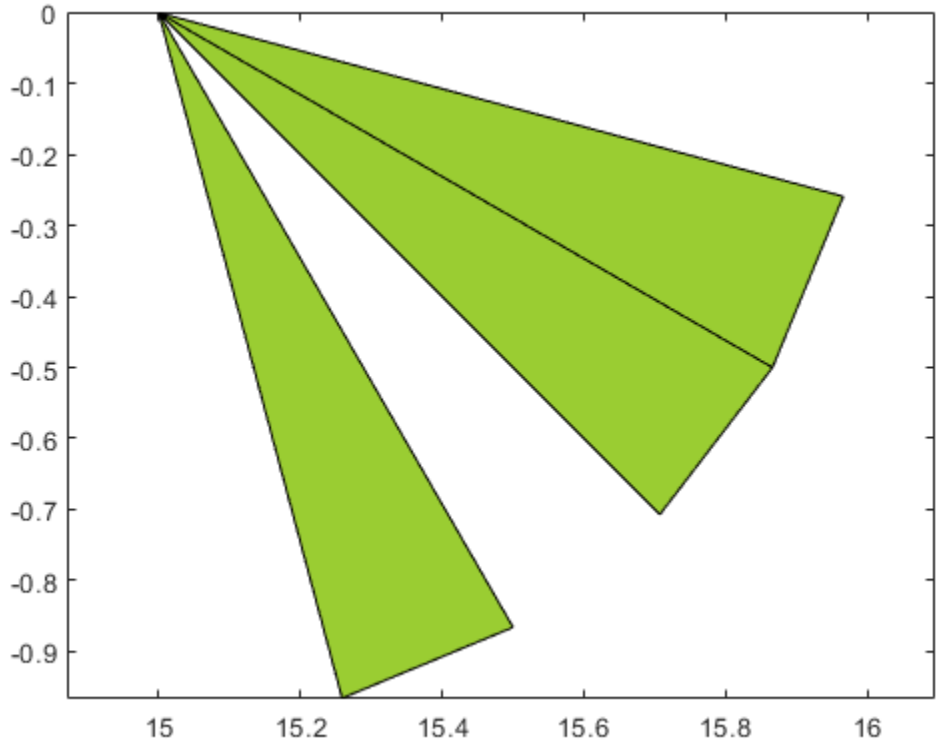
```
shp = alphaShape(x,y);
```

Find the spectrum of critical alpha values that produces unique alpha shapes for the point cloud.

```
alphaspec = alphaSpectrum(shp);
```

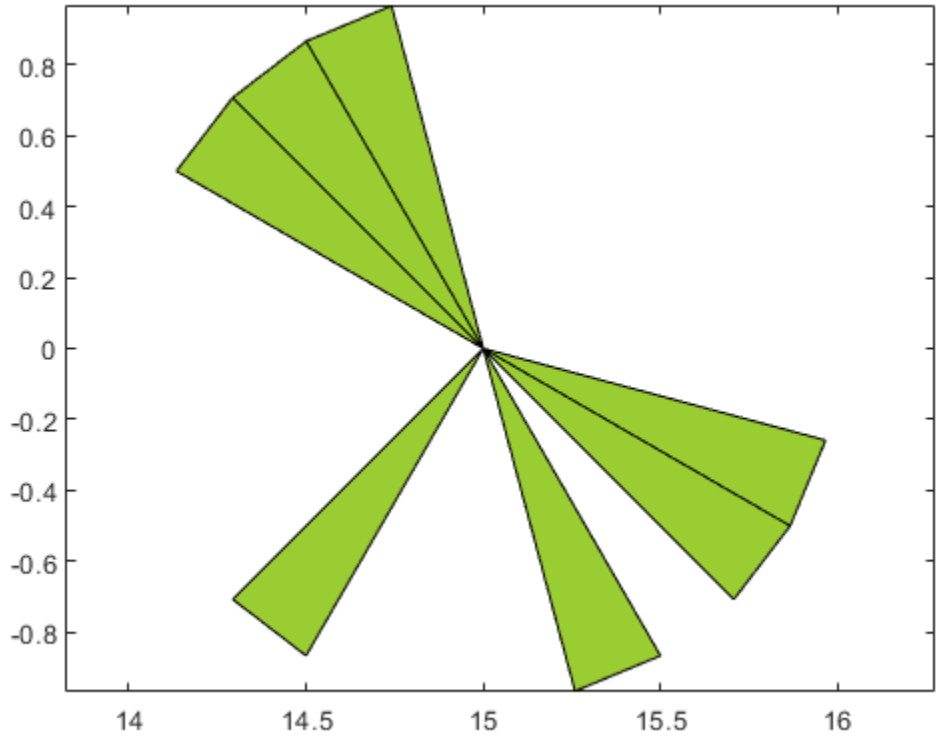
Plot the alpha shape corresponding to the smallest alpha value in the spectrum.

```
shp.Alpha = alphaspec(length(alphaspec));
plot(shp)
```



Compare this alpha shape to the one produced by the next smallest critical alpha value. The alpha shapes are unique.

```
shp.Alpha = alphaspec(length(alphaspec)-1);
plot(shp)
```



Input Arguments

shp — Alpha shape
`alphaShape` object

Alpha shape, specified as an `alphaShape` object. For more information, see `alphaShape`.

Example: `shp = alphaShape(x,y)` creates a 2-D `alphaShape` object from the `(x,y)` point coordinates.

Output Arguments

a — Alpha values for distinct alpha shapes
column vector

Alpha values for distinct alpha shapes, returned as a column vector in descending sorted order.

See Also

`alphaShape` | `plot`

Introduced in R2014b

alphaTriangulation

Triangulation that fills alpha shape

Syntax

```
tri = alphaTriangulation(shp)
tri = alphaTriangulation(shp,RegionID)
```

```
[tri,P] = alphaTriangulation( ___ )
```

Description

`tri = alphaTriangulation(shp)` returns a triangulation that defines the domain of the alpha shape. Each row in `tri` specifies a triangle or tetrahedron defined by vertex IDs (the row numbers of the `shp.Points` matrix).

`tri = alphaTriangulation(shp,RegionID)` returns a triangulation for a region of the alpha shape. `RegionID` is the ID for the region and `1` ≤ `RegionID` ≤ `numRegions(shp)`.

`[tri,P] = alphaTriangulation( ___ )` also returns a matrix of vertex coordinates, `P`, using any of the previous syntaxes.

Examples

Compute Triangulation for 3-D Point Cloud

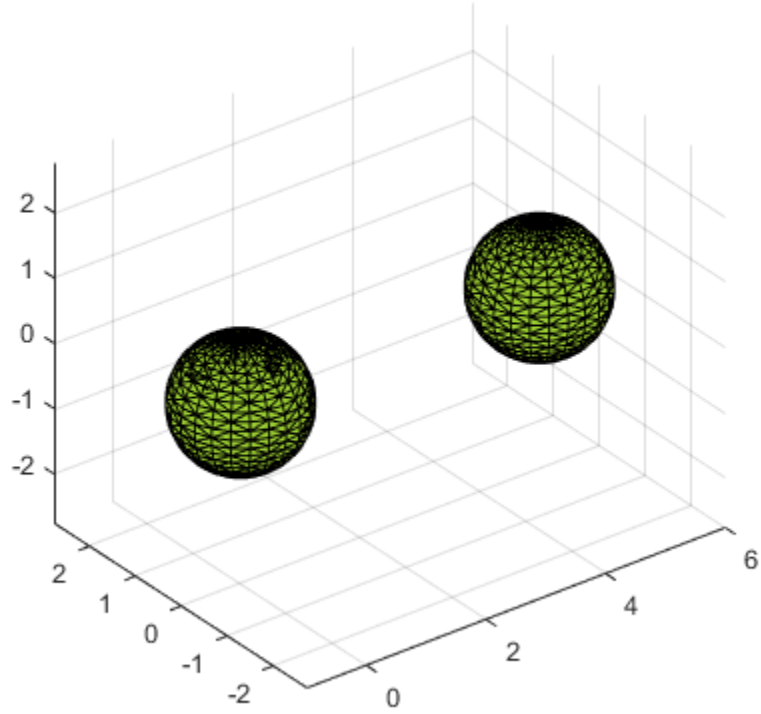
Create a set of 3-D points.

```
[x1, y1, z1] = sphere(24);
x1 = x1(:);
y1 = y1(:);
z1 = z1(:);
x2 = x1+5;
```

```
P = [x1 y1 z1; x2 y1 z1];
P = unique(P,'rows');
```

Create and plot an alpha shape for the point cloud using an alpha radius of 1.

```
shp = alphaShape(P,1);
plot(shp)
```



Use `alphaTriangulation` to recover the triangulation that defines the domain of the alpha shape.

```
tri = alphaTriangulation(shp);
```

Find the total number of tetrahedra that make up the alpha shape.

```
numtetrahedra = size(tri,1)
```

```
numtetrahedra = 3729
```

Input Arguments

shp — Alpha shape
alphaShape object

Alpha shape, specified as an alphaShape object. For more information, see alphaShape.

Example: shp = alphaShape(x,y) creates a 2-D alphaShape object from the (x,y) point coordinates.

RegionID — ID number for a region in the alpha shape
positive integer scalar

ID number for region in alpha shape, specified as a positive integer scalar between 1 and numRegions(shp).

An alpha shape can contain several smaller regions, depending on the point set and parameters. Each of these smaller regions is assigned a unique RegionID, which numbers the regions from the largest area or volume to the smallest. For example, consider a 3-D alpha shape with two regions. The region with the largest volume has a RegionID of 1, and the smaller region has a RegionID of 2.

Example: shp.RegionThreshold = area(shp,numRegions(shp)-2); suppresses the two smallest regions in 2-D alpha shape shp.

Data Types: double

Output Arguments

tri — Triangulation
matrix

Triangulation, returned as a matrix. tri is of size mtri-by-nv, where mtri is the number of triangles or tetrahedra in the alpha shape and nv is the number of vertices. The value of nv is 3 for 2-D alpha shapes and 4 for 3-D alpha shapes.

P — Vertex coordinates
matrix

Vertex coordinates, returned as a matrix. `P` is of size `N`-by-`dim`, where `N` is the number of points in the alpha shape and `dim` is either `2` or `3` (for either a 2-D or 3-D alpha shape).

See Also

`alphaShape` | `plot` | `triangulation` | `triplot`

Introduced in R2014b

area

Area of 2-D alpha shape

Syntax

```
A = area(shp)
A = area(shp,RegionID)
```

Description

`A = area(shp)` returns the area of 2-D alpha shape `shp`.

`A = area(shp,RegionID)` returns the area of a region of the alpha shape. `RegionID` is the ID for the region and `1` ≤ `RegionID` ≤ `numRegions(shp)`.

Examples

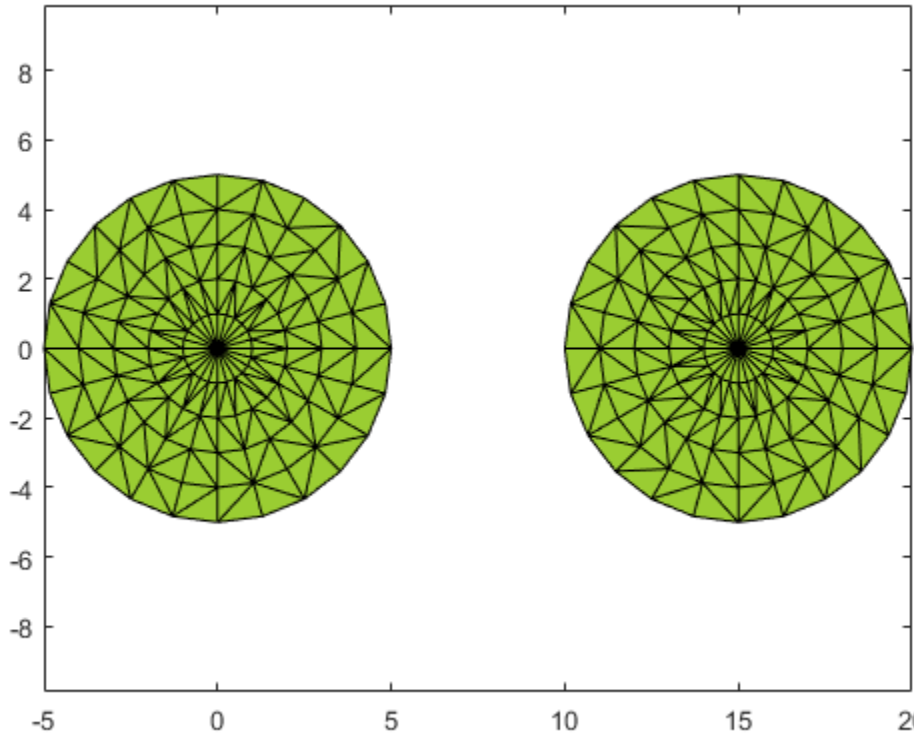
Find Area of 2-D Alpha Shape

Create a set of 2-D points.

```
th = (pi/12:pi/12:2*pi)';
x1 = [reshape(cos(th)*(1:5), numel(cos(th)*(1:5)),1); 0];
y1 = [reshape(sin(th)*(1:5), numel(sin(th)*(1:5)),1); 0];
x = [x1; x1+15;];
y = [y1; y1];
```

Create and plot an alpha shape using an alpha radius of 2.5.

```
shp = alphaShape(x,y,2.5);
plot(shp)
```



Compute the area of the alpha shape.

```
totalarea = area(shp)
```

```
totalarea = 155.2914
```

Compute the areas of each of the two regions separately.

```
regionareas = area(shp, 1:numRegions(shp))
```

```
regionareas = 1×2

   77.6457   77.6457
```

Input Arguments

shp — Alpha shape
alphaShape object

Alpha shape, specified as an alphaShape object. For more information, see alphaShape.

Example: shp = alphaShape(x,y) creates a 2-D alphaShape object from the (x,y) point coordinates.

RegionID — ID number for a region in the alpha shape
positive integer scalar

ID number for region in alpha shape, specified as a positive integer scalar between 1 and numRegions(shp).

An alpha shape can contain several smaller regions, depending on the point set and parameters. Each of these smaller regions is assigned a unique RegionID, which numbers the regions from the largest area or volume to the smallest. For example, consider a 3-D alpha shape with two regions. The region with the largest volume has a RegionID of 1, and the smaller region has a RegionID of 2.

Example: shp.RegionThreshold = area(shp,numRegions(shp)-2); suppresses the two smallest regions in 2-D alpha shape shp.

Data Types: double

See Also

alphaShape | criticalAlpha | perimeter | volume

Introduced in R2014b

boundaryFacets

Boundary facets of alpha shape

Syntax

```
bf = boundaryFacets(shp)
bf = boundaryFacets(shp,RegionID)
```

```
[bf,P] = boundaryFacets( ___ )
```

Description

`bf = boundaryFacets(shp)` returns a matrix representing the facets that make up the boundary of the alpha shape. The facets represent edge segments in 2-D and triangles in 3-D. The vertices of the facets index into the `shp.Points` matrix.

`bf = boundaryFacets(shp,RegionID)` returns the boundary facets for a region of the alpha shape. `RegionID` is the ID for the region and `1 ≤ RegionID ≤ numRegions(shp)`.

`[bf,P] = boundaryFacets( ___ )` also returns a matrix of vertex coordinates, `P`, using any of the previous syntaxes.

Examples

Find Boundary of 3-D Alpha Shape

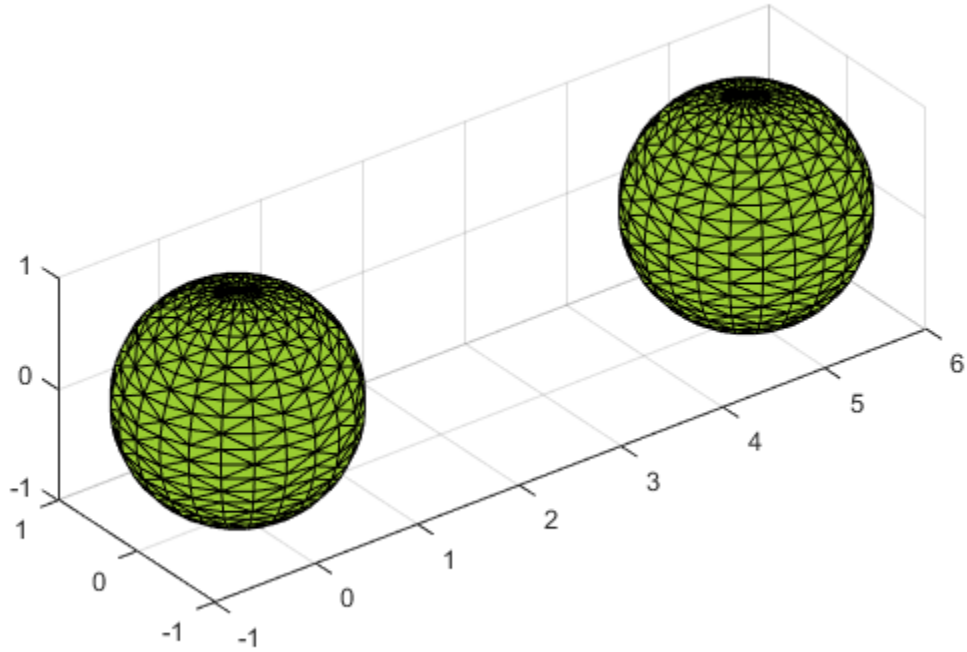
Create a set of 3-D points.

```
[x1, y1, z1] = sphere(24);
x1 = x1(:);
y1 = y1(:);
z1 = z1(:);
x2 = x1+5;
```

```
P = [x1 y1 z1; x2 y1 z1];
P = unique(P,'rows');
```

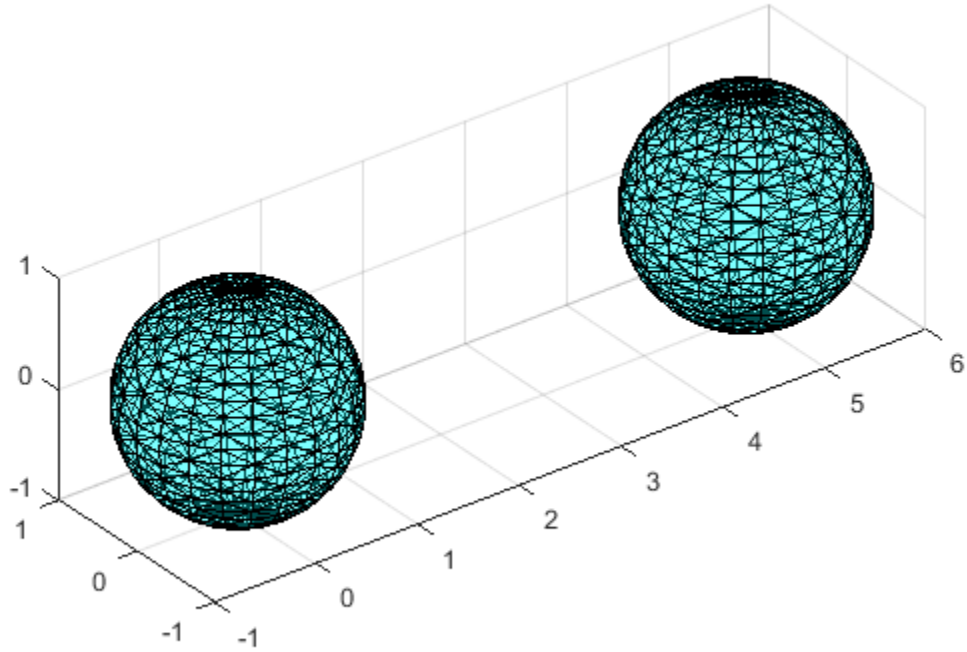
Create and plot an alpha shape using an alpha radius of 1.5.

```
shp = alphaShape(P,1.5);
plot(shp)
axis equal
```



Compute and plot only the boundary of the alpha shape.

```
[tri, xyz] = boundaryFacets(shp);
trisurf(tri,xyz(:,1),xyz(:,2),xyz(:,3),...
    'FaceColor','cyan','FaceAlpha',0.3)
axis equal
```



Input Arguments

shp — Alpha shape
alphaShape object

Alpha shape, specified as an alphaShape object. For more information, see alphaShape.

Example: shp = alphaShape(x,y) creates a 2-D alphaShape object from the (x,y) point coordinates.

RegionID — ID number for a region in the alpha shape
positive integer scalar

ID number for region in alpha shape, specified as a positive integer scalar between `1` and `numRegions(shp)`.

An alpha shape can contain several smaller regions, depending on the point set and parameters. Each of these smaller regions is assigned a unique `RegionID`, which numbers the regions from the largest area or volume to the smallest. For example, consider a 3-D alpha shape with two regions. The region with the largest volume has a `RegionID` of 1, and the smaller region has a `RegionID` of 2.

Example: `shp.RegionThreshold = area(shp,numRegions(shp)-2);` suppresses the two smallest regions in 2-D alpha shape `shp`.

Data Types: `double`

Output Arguments

`bf` — Boundary facets
matrix

Boundary facets, returned as a matrix. `bf` is of size `m`-by-`n`, where `m` is the number of boundary facets and `n` is the number of vertices per facet.

`P` — Vertex coordinates
matrix

Vertex coordinates, returned as a matrix. `P` is of size `N`-by-`dim`, where `N` is the number of points on the boundary of the alpha shape and `dim` is either `2` or `3` (for either a 2-D or 3-D alpha shape).

See Also

`alphaShape` | `plot` | `triangulation`

Introduced in R2014b

criticalAlpha

Alpha radius defining critical transition in shape

Syntax

```
a = criticalAlpha(shp,type)
```

Description

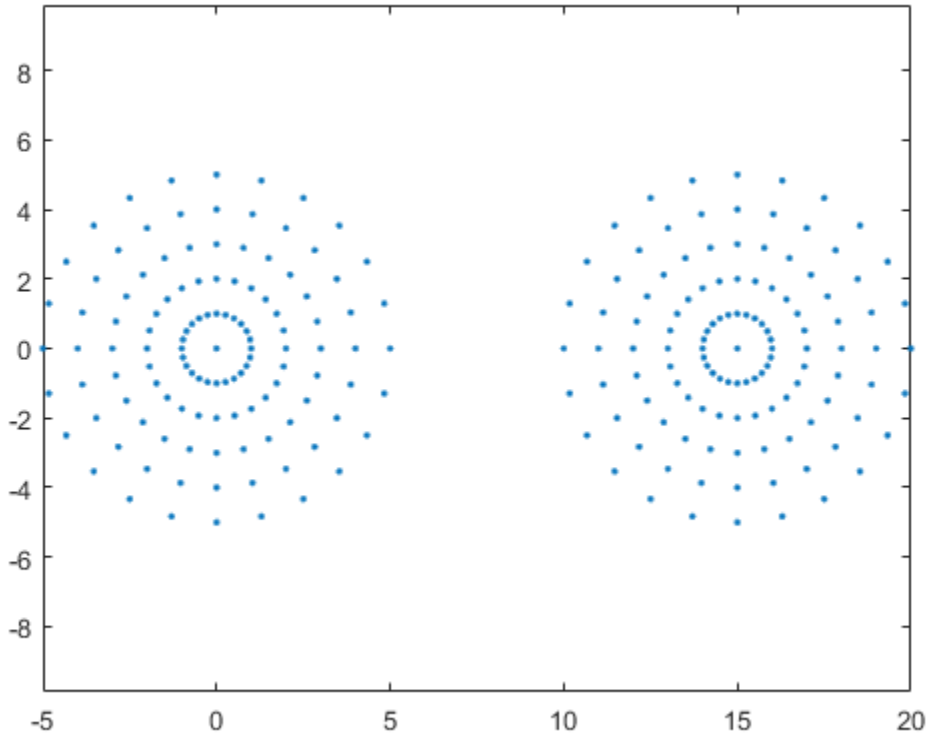
`a = criticalAlpha(shp,type)` returns the critical alpha radius that produces a notable transition in the alpha shape. Specifying `type` as `'all-points'` returns the smallest alpha radius producing an alpha shape that encloses all points. Specifying `type` as `'one-region'` returns the smallest alpha radius producing an alpha shape that encloses all points *and* has only one region.

Examples

Find Critical Alpha Values of 2-D Point Cloud

Create and plot a set of 2-D points.

```
th = (pi/12:pi/12:2*pi)';
x1 = [reshape(cos(th)*(1:5), numel(cos(th)*(1:5)),1); 0];
y1 = [reshape(sin(th)*(1:5), numel(sin(th)*(1:5)),1); 0];
x = [x1; x1+15;];
y = [y1; y1];
plot(x,y,'.')
axis equal
```

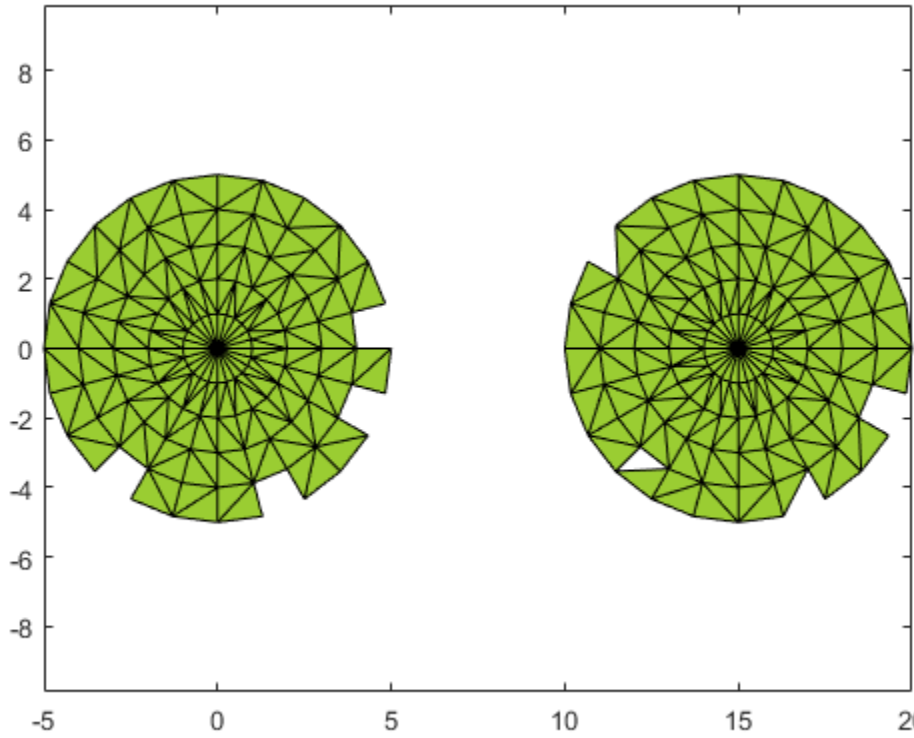


Create an alpha shape using the default alpha radius.

```
shp = alphaShape(x,y);
```

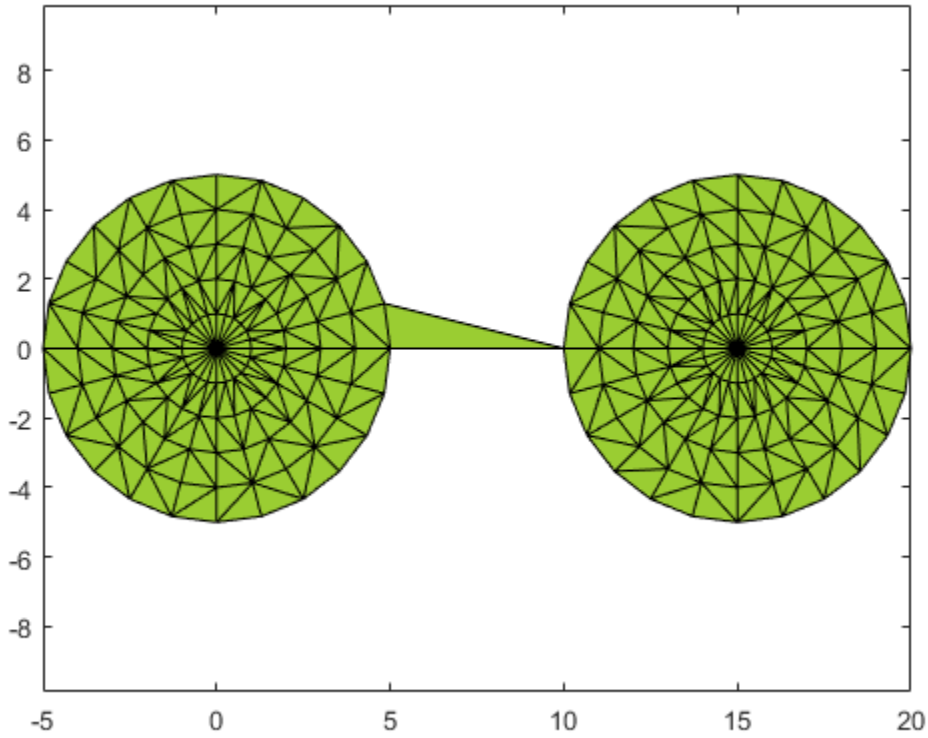
Compute the smallest alpha radius that produces an alpha shape enclosing all of the points and plot the corresponding alpha shape.

```
pc = criticalAlpha(shp,'all-points');
shp.Alpha = pc;
plot(shp)
```



Compute the smallest alpha radius that produces an alpha shape enclosing all of the points and having only one region.

```
shp = alphaShape(x,y);
pc = criticalAlpha(shp,'one-region');
shp.Alpha = pc;
plot(shp)
```



Input Arguments

shp — Alpha shape
`alphaShape` object

Alpha shape, specified as an `alphaShape` object. For more information, see `alphaShape`.

Example: `shp = alphaShape(x,y)` creates a 2-D `alphaShape` object from the `(x,y)` point coordinates.

type — Type of critical transition
`'all-points'` | `'one-region'`

Type of critical transition, specified as either `'all-points'` or `'one-region'`.

- `'all-points'` corresponds to the smallest alpha radius producing an alpha shape that encloses all points.
- `'one-region'` corresponds to the smallest alpha radius producing an alpha shape that encloses all points *and* has only one region.

Data Types: `char`

Output Arguments

a — Critical alpha radius
scalar

Critical alpha radius, returned as a scalar. `a` is the value of the alpha radius that produces an alpha shape, which either encloses all points (if `type` is `'all-points'`), or encloses all points within a single region (if `type` is `'one-region'`).

After using `criticalAlpha` to find `a`, you can make the alpha radius of `shp` equal to `a` by typing `shp.Alpha = a`.

See Also

`alphaShape` | `alphaSpectrum`

Introduced in R2014b

inShape

Determine if point is inside alpha shape

Syntax

```
tf = inShape(shp,qx,qy)
tf = inShape(shp,qx,qy,qz)
tf = inShape(shp,QP)
tf = inShape( ___ ,RegionID)
[tf,ID] = inShape( ___ )
```

Description

`tf = inShape(shp,qx,qy)` returns logical `1` (`true`) values for the 2-D query points `(qx,qy)` that are within 2-D alpha shape `shp`. Otherwise, `inShape` returns values of logical `0` (`false`). The `qx` and `qy` arguments are numeric arrays whose corresponding elements specify the `(x,y)` query point coordinates.

`tf = inShape(shp,qx,qy,qz)` tests whether the 3-D query points `(qx,qy,qz)` are within 3-D alpha shape `shp`.

`tf = inShape(shp,QP)` specifies the 2-D or 3-D query point coordinates in a matrix with 2 or 3 columns.

`tf = inShape( ___ ,RegionID)` tests whether the query points are within a specific region of the alpha shape, using any of the previous syntaxes. `RegionID` is the ID for the region and `1` ≤ `RegionID` ≤ `numRegions(shp)`.

`[tf,ID] = inShape( ___ )` also returns the IDs for the regions in the alpha shape that contain the query points. `ID` is `NaN` for query points that are not in the alpha shape.

Examples

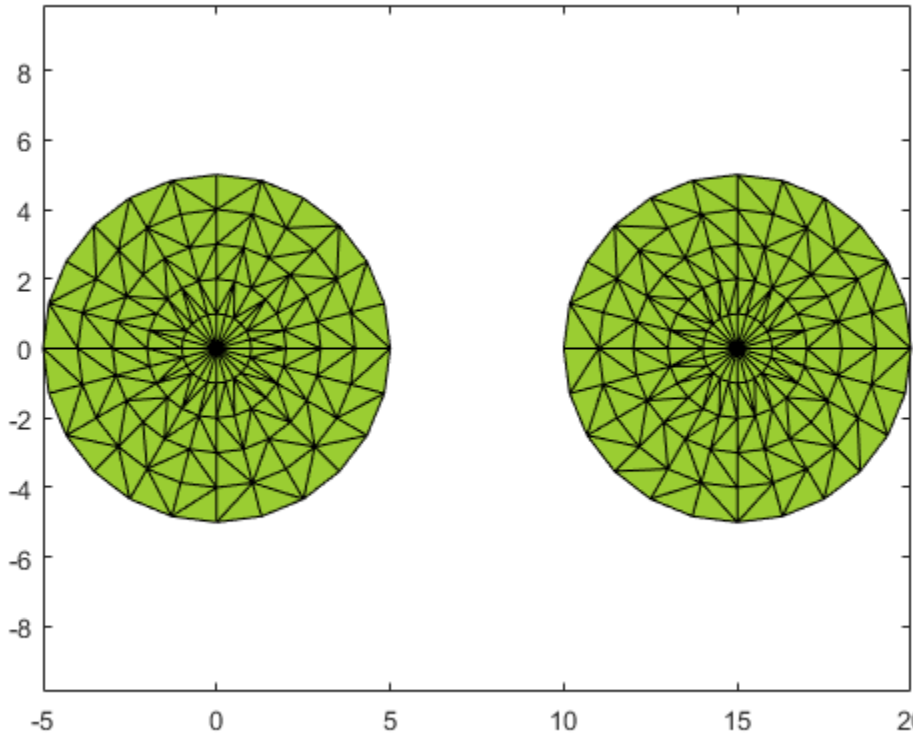
Query Points Inside and Outside of 2-D Alpha Shape

Create a set of 2-D points.

```
th = (pi/12:pi/12:2*pi)';
x1 = [reshape(cos(th)*(1:5), numel(cos(th)*(1:5)),1); 0];
y1 = [reshape(sin(th)*(1:5), numel(sin(th)*(1:5)),1); 0];
x = [x1; x1+15;];
y = [y1; y1];
```

Create and plot an alpha shape using an alpha radius of 2.5.

```
shp = alphaShape(x,y,2.5);
plot(shp)
```

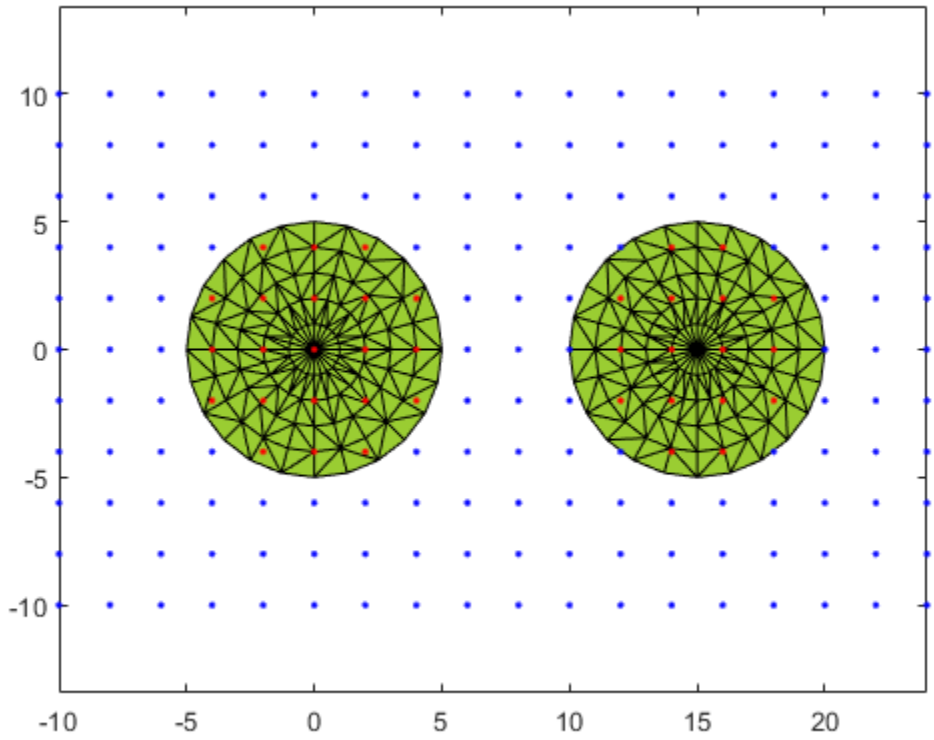


Create a Cartesian grid of query points near the alpha shape.

```
[qx, qy] = meshgrid(-10:2:25, -10:2:10);
```

Check if the query points are inside of the alpha shape, and if so, plot them red. Plot the query points that lie outside of the alpha shape in blue.

```
in = inShape(shp,qx,qy);
plot(shp)
hold on
plot(qx(in),qy(in),'r.')
plot(qx(~in),qy(~in),'b.')
```



Input Arguments

shp — Alpha shape
alphaShape object

Alpha shape, specified as an alphaShape object. For more information, see alphaShape.

Example: shp = alphaShape(x,y) creates a 2-D alphaShape object from the (x,y) point coordinates.

qx — Query point x-coordinates
numeric array

Query point x-coordinates, specified as a numeric array.

Data Types: double

qy — Query point y-coordinates
numeric array

Query point y-coordinates, specified as a numeric array.

Data Types: double

qz — Query point z-coordinates
numeric array

Query point z-coordinates, specified as a numeric array.

Data Types: double

QP — Query point coordinates
matrix with two columns | matrix with three columns

Query point coordinates, specified as a matrix with two columns (2-D) or a matrix with three columns (3-D).

- For 2-D, the columns of QP represent x and y coordinates, respectively.
- For 3-D, the columns of QP represent x, y, and z coordinates, respectively.

Data Types: double

RegionID — ID number for a region in the alpha shape
positive integer scalar

ID number for region in alpha shape, specified as a positive integer scalar between `1` and `numRegions(shp)`.

An alpha shape can contain several smaller regions, depending on the point set and parameters. Each of these smaller regions is assigned a unique `RegionID`, which numbers the regions from the largest area or volume to the smallest. For example, consider a 3-D alpha shape with two regions. The region with the largest volume has a `RegionID` of 1, and the smaller region has a `RegionID` of 2.

Example: `shp.RegionThreshold = area(shp,numRegions(shp)-2);` suppresses the two smallest regions in 2-D alpha shape `shp`.

Data Types: `double`

Output Arguments

`tf` — Containment status of query points
logical array

Status of the query points, returned as a logical array. The size of `tf` is equal to the size of the inputs that specify the query points (`qx`, `qy`, `qz`, or `QP`).

`inShape` returns logical `1` (`true`) values for points that are within the alpha shape or exactly on the boundary.

`ID` — IDs of the regions containing the query points
numeric array

IDs of regions containing query points, returned as a numeric array. `ID` is the same size as `tf`.

See Also

`alphaShape` | `plot`

Introduced in R2014b

nearestNeighbor

Determine nearest alpha shape boundary point

Syntax

```
I = nearestNeighbor(shp,qx,qy)
I = nearestNeighbor(shp,qx,qy,qz)
I = nearestNeighbor(shp,QP)

I = nearestNeighbor( ___ ,RegionID)

[I,D] = nearestNeighbor( ___ )
```

Description

`I = nearestNeighbor(shp,qx,qy)`, for a 2-D alpha shape `shp`, returns the indices of points on the boundary of `shp` closest to the query points. `I` is the array of nearest neighbor indices where each index corresponds to the row index in `shp.Points`. The `qx` and `qy` query coordinates must be the same size.

`I = nearestNeighbor(shp,qx,qy,qz)`, for a 3-D alpha shape, returns the indices of the boundary points of `shp` closest to `(qx,qy,qz)` and corresponds to the row indices in `shp.Points`. The `qx`, `qy`, and `qz` query coordinates must be the same size.

`I = nearestNeighbor(shp,QP)` specifies the query points as a matrix `QP`. For a 2-D alpha shape, `QP` is a matrix with two columns representing the `qx` and `qy` coordinates. For a 3-D alpha shape, `QP` has three columns representing the `qx`, `qy`, and `qz` coordinates.

`I = nearestNeighbor( ___ ,RegionID)` returns the index of the nearest point that lies on the boundary of the region specified by `RegionID`, where `1 ≤ RegionID ≤ numRegions(shp)`. You can include any of the input arguments in the previous syntaxes.

`[I,D] = nearestNeighbor( ___ )` additionally returns the Euclidean distance between the query point and its nearest neighbor. `D` has the same size as `I`.

Examples

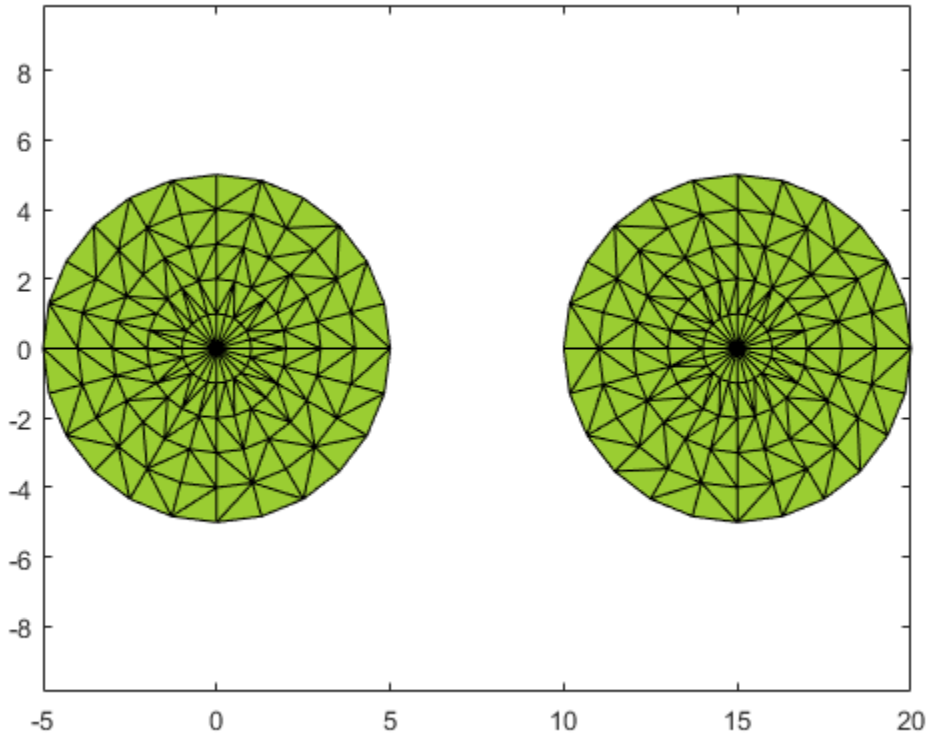
Nearest Alpha Shape Boundary Point

Create a set of 2-D points.

```
th = (pi/12:pi/12:2*pi)';
x1 = [reshape(cos(th)*(1:5), numel(cos(th)*(1:5)),1); 0];
y1 = [reshape(sin(th)*(1:5), numel(sin(th)*(1:5)),1); 0];
x = [x1; x1+15];
y = [y1; y1];
```

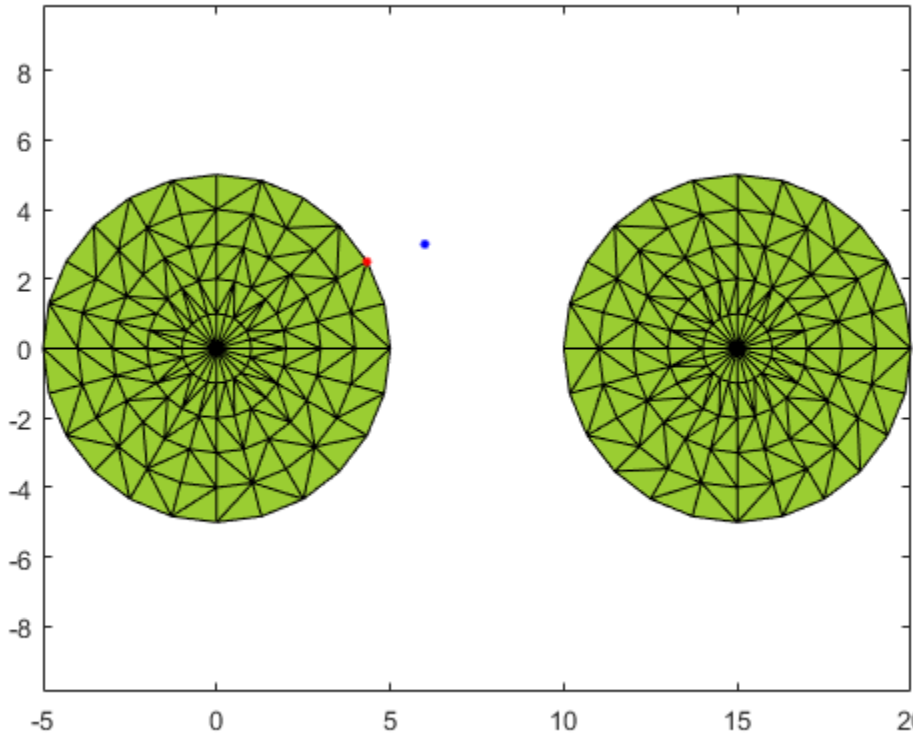
Create and plot an alpha shape with alpha radius equal to 1.

```
shp = alphaShape(x,y,1);
plot(shp)
hold on
```



Compute the nearest `shp` boundary point to the query point `QP`. Plot the query point in blue and the nearest boundary neighbor in red.

```
QP = [6 3];
plot(QP(1),QP(2),'b.','MarkerSize',10)
hold on
I = nearestNeighbor(shp, QP);
plot(shp.Points(I,1),shp.Points(I,2),'r.','MarkerSize',10)
```



Input Arguments

shp — Alpha shape
`alphaShape` object

Alpha shape, specified as an `alphaShape` object. For more information, see `alphaShape`.

Example: `shp = alphaShape(x,y)` creates a 2-D `alphaShape` object from the `(x,y)` point coordinates.

qx — Query point x-coordinates
numeric array

Query point x-coordinates, specified as a numeric array.

Data Types: `double`

`qy` — Query point y-coordinates
numeric array

Query point y-coordinates, specified as a numeric array.

Data Types: `double`

`qz` — Query point z-coordinates
numeric array

Query point z-coordinates, specified as a numeric array.

Data Types: `double`

`QP` — Query point coordinates
two-column matrix | three-column matrix

Query point coordinates, specified as a two-column matrix or a three-column matrix.

- For 2-D, the columns of `P` represent `qx` and `qy` coordinates, respectively.
- For 3-D, the columns of `P` represent `qx`, `qy`, and `qz` coordinates, respectively.

Data Types: `double`

`RegionID` — ID number for a region in the alpha shape
positive integer scalar

ID number for region in alpha shape, specified as a positive integer scalar between `1` and `numRegions(shp)`.

An alpha shape can contain several smaller regions, depending on the point set and parameters. Each of these smaller regions is assigned a unique `RegionID`, which numbers the regions from the largest area or volume to the smallest. For example, consider a 3-D alpha shape with two regions. The region with the largest volume has a `RegionID` of 1, and the smaller region has a `RegionID` of 2.

Example: `shp.RegionThreshold = area(shp,numRegions(shp)-2);` suppresses the two smallest regions in 2-D alpha shape `shp`.

Data Types: `double`

Output Arguments

`I` — Nearest neighbor indices
integer-valued array

Nearest neighbor indices, returned as an integer-valued array. The indices correspond to the row index of `shp.Points` and indicate the points on the boundary of `shp` that are closest to the given query points.

`D` — Distance from query points to nearest neighbors
numeric array

Distance from query points to nearest neighbors, returned as a numeric array. `D` is the 2-D or 3-D Euclidean distance and is the same size as `I`.

See Also

`alphaShape`

Introduced in R2015a

numRegions

Number of regions in alpha shape

Syntax

```
N = numRegions(shp)
```

Description

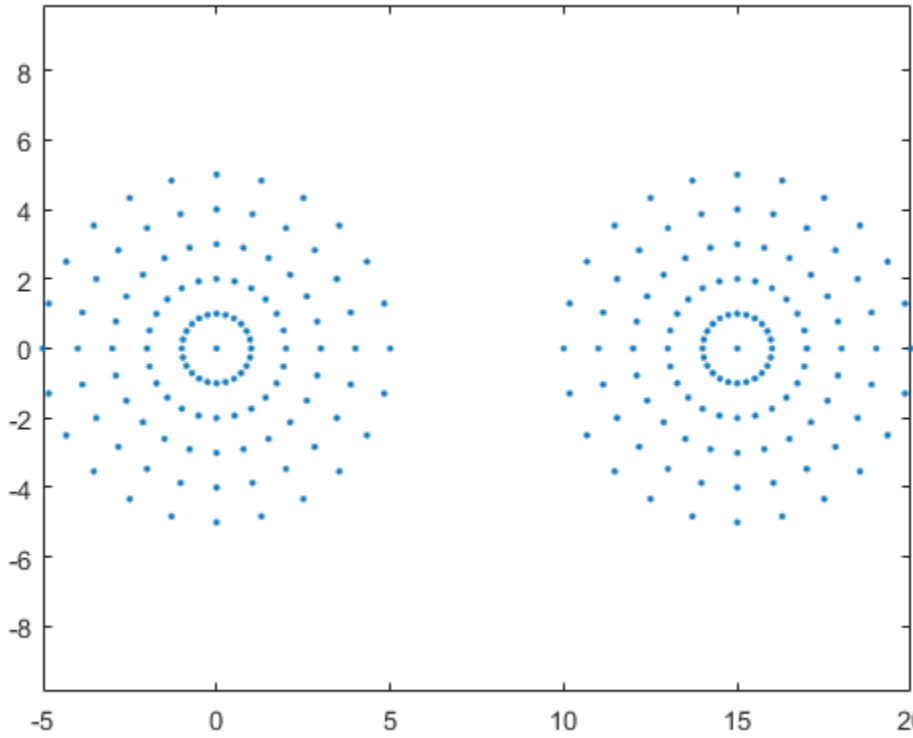
`N = numRegions(shp)` returns the number of distinct regions that make up the alpha shape. For an alpha radius of `Inf`, the alpha shape is the convex hull, and the number of regions is one. As the value of the alpha radius decreases the shape can break into separate regions, depending on the point set.

Examples

Find Number of Regions in 2-D Alpha Shape

Create and plot a set of 2-D points.

```
th = (pi/12:pi/12:2*pi)';
x1 = [reshape(cos(th)*(1:5), numel(cos(th)*(1:5)),1); 0];
y1 = [reshape(sin(th)*(1:5), numel(sin(th)*(1:5)),1); 0];
x = [x1; x1+15;];
y = [y1; y1];
plot(x,y,'.')
axis equal
```



Create an alpha shape using an alpha radius of 7 and query the number of distinct regions in the shape.

```
shp = alphaShape(x,y,7);
nregions = numRegions(shp)
```

```
nregions = 1
```

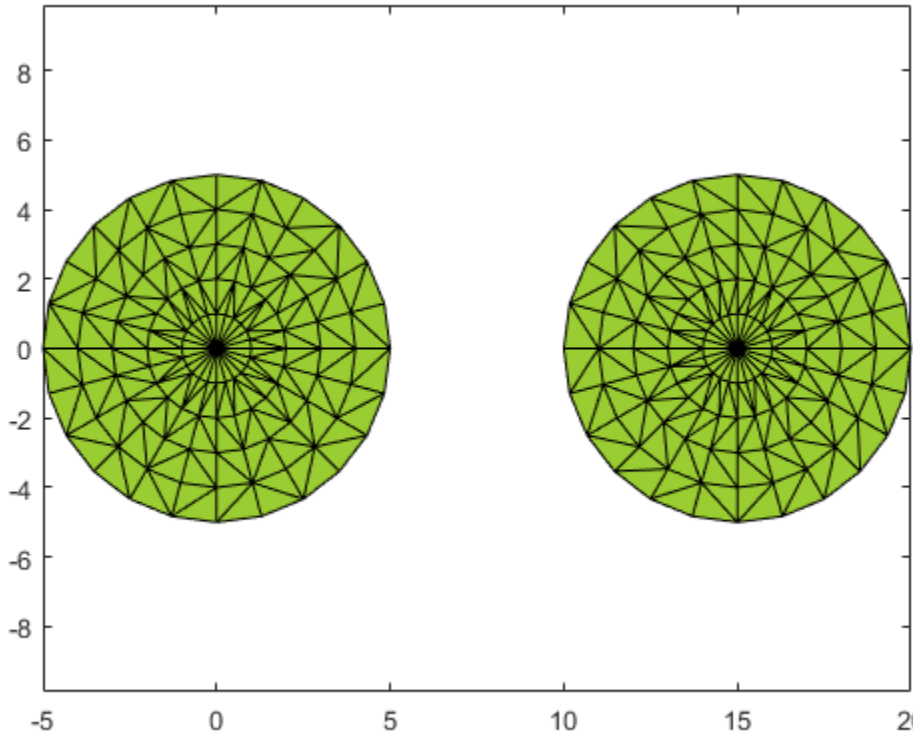
Use a smaller alpha radius of 2.5 to better capture the boundary and then retrieve the new number of distinct regions.

```
shp.Alpha = 2.5;
nregions = numRegions(shp)
```

```
nregions = 2
```

Plot the alpha shape to check the boundary quality.

```
plot(shp)
```



Input Arguments

shp — Alpha shape
`alphaShape` object

Alpha shape, specified as an `alphaShape` object. For more information, see `alphaShape`.

Example: `shp = alphaShape(x,y)` creates a 2-D `alphaShape` object from the `(x,y)` point coordinates.

See Also

`alphaShape`

Introduced in R2014b

perimeter

Perimeter of 2-D alpha shape

Syntax

```
L = perimeter(shp)
L = perimeter(shp,RegionID)
```

Description

`L = perimeter(shp)` returns the total perimeter of 2-D alpha shape `shp`, including the perimeter of any interior holes in the alpha shape.

`L = perimeter(shp,RegionID)` returns the perimeter of a region within the alpha shape. `RegionID` is the ID for the region and `1` ≤ `RegionID` ≤ `numRegions(shp)`.

Examples

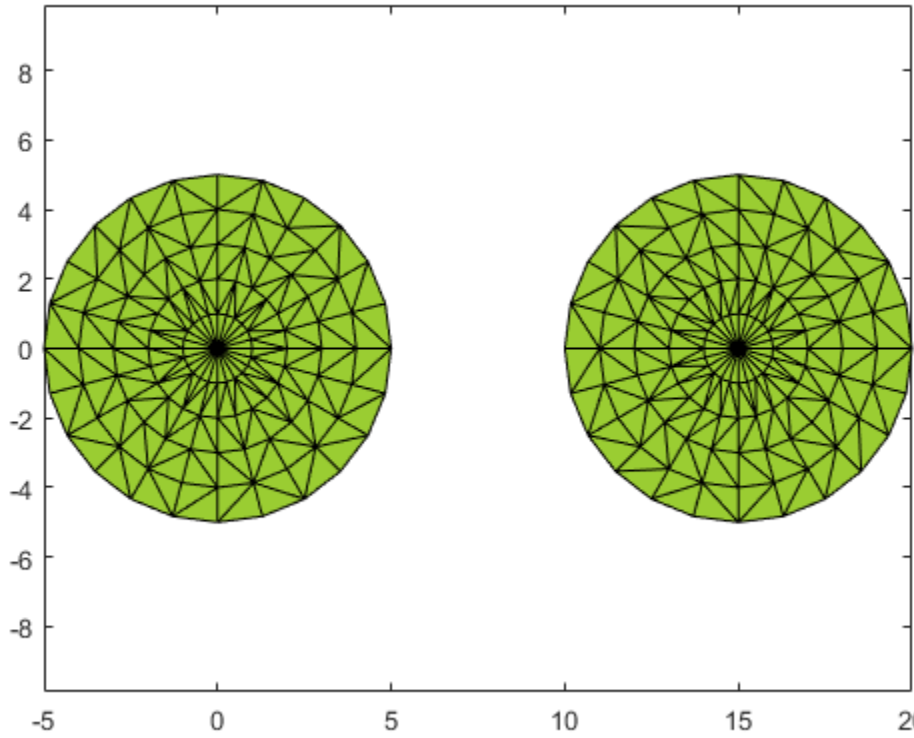
Find Perimeter of 2-D Alpha Shape

Create a set of 2-D points.

```
th = (pi/12:pi/12:2*pi)';
x1 = [reshape(cos(th)*(1:5), numel(cos(th)*(1:5)),1); 0];
y1 = [reshape(sin(th)*(1:5), numel(sin(th)*(1:5)),1); 0];
x = [x1; x1+15;];
y = [y1; y1];
```

Create and plot an alpha shape using an alpha radius of 2.5.

```
shp = alphaShape(x,y,2.5);
plot(shp)
```



Compute the perimeter of the alpha shape.

```
totalperim = perimeter(shp)
```

```
totalperim = 62.6526
```

Compute the perimeters of each of the two regions separately.

```
regionperims = perimeter(shp, 1:numRegions(shp))
```

```
regionperims = 1×2

   31.3263   31.3263
```

Input Arguments

shp — 2-D alpha shape
alphaShape object

2-D alpha shape, specified as an alphaShape object. For more information, see alphaShape.

Example: shp = alphaShape(x,y) creates a 2-D alphaShape object from the (x,y) point coordinates.

RegionID — ID number for a region in the alpha shape
positive integer scalar

ID number for region in alpha shape, specified as a positive integer scalar between 1 and numRegions(shp).

An alpha shape can contain several smaller regions, depending on the point set and parameters. Each of these smaller regions is assigned a unique RegionID, which numbers the regions from the largest area or volume to the smallest. For example, consider a 3-D alpha shape with two regions. The region with the largest volume has a RegionID of 1, and the smaller region has a RegionID of 2.

Example: shp.RegionThreshold = area(shp,numRegions(shp)-2); suppresses the two smallest regions in 2-D alpha shape shp.

Data Types: double

See Also

alphaShape | area | criticalAlpha

Introduced in R2014b

plot

Plot alpha shape

Syntax

```
plot(shp)
plot(shp,Name,Value)
h = plot( ___ )
```

Description

`plot(shp)` plots alpha shape `shp` in a figure window.

`plot(shp,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. For a complete list of allowed `Name,Value` pairs, see Patch.

`h = plot( ___ )` returns a handle to a `Patch` object using any of the previous syntaxes.

Examples

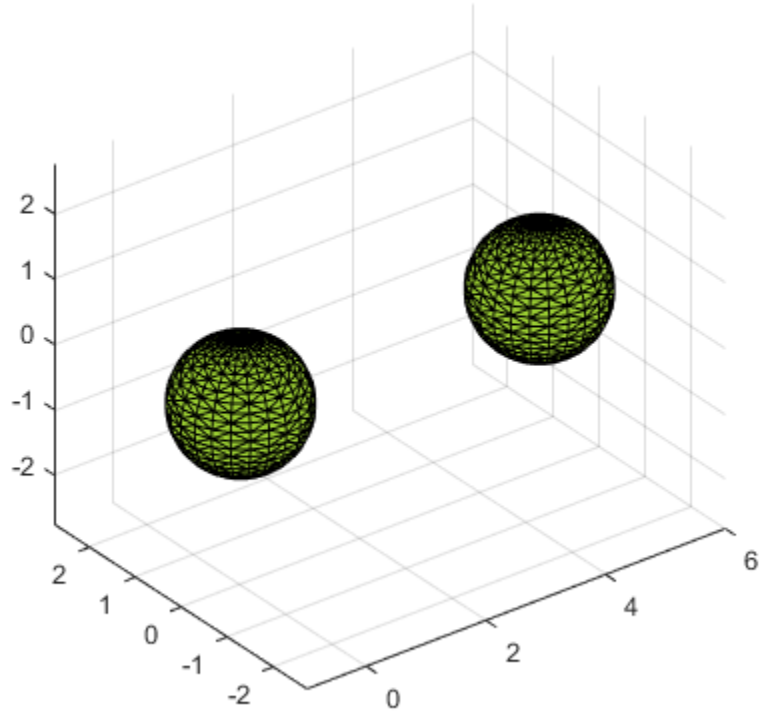
Plot 3-D Alpha Shape and Modify Patch Properties

Create a set of 3-D points.

```
[x1, y1, z1] = sphere(24);
x1 = x1(:);
y1 = y1(:);
z1 = z1(:);
x2 = x1+5;
P = [x1 y1 z1; x2 y1 z1];
P = unique(P,'rows');
```

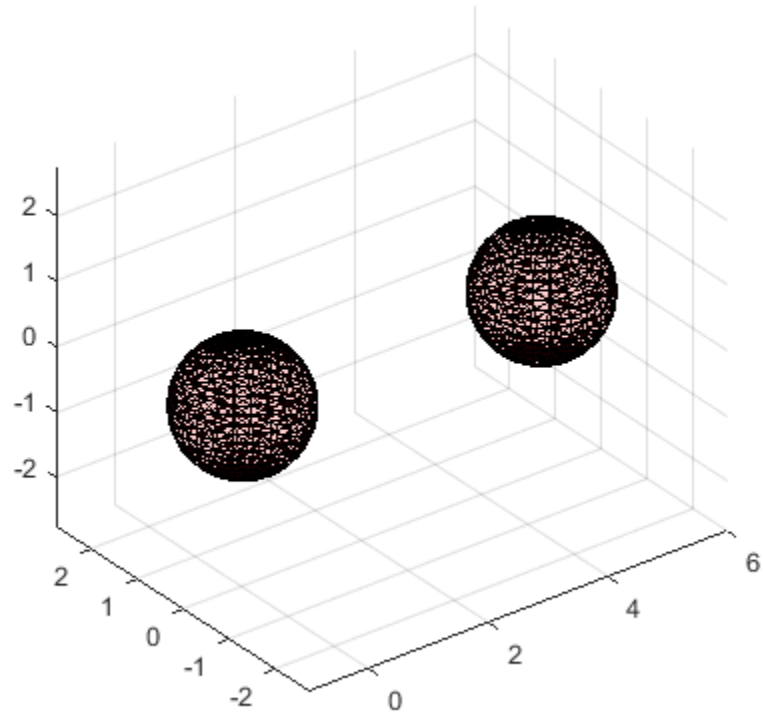
Create and plot an alpha shape using an alpha radius of 1.5.

```
shp = alphaShape(P,1.5);
plot(shp)
```



Plot the alpha shape with a specified color and transparency factor.

```
plot(shp,'FaceColor','red','FaceAlpha',0.1)
```



Input Arguments

shp — Alpha shape
`alphaShape` object

Alpha shape, specified as an `alphaShape` object. For more information, see `alphaShape`.

Example: `shp = alphaShape(x,y)` creates a 2-D `alphaShape` object from the `(x,y)` point coordinates.

Output Arguments

h — Handle to `Patch` object
handle

Handle to `Patch` object, returned as a handle. For more information, see “Introduction to Patch Objects”.

See Also

`alphaShape` | `patch`

Introduced in R2014b

surfaceArea

Surface area of 3-D alpha shape

Syntax

```
A = surfaceArea(shp)
A = surfaceArea(shp,RegionID)
```

Description

`A = surfaceArea(shp)` returns the total surface area of 3-D alpha shape `shp`, including the surface area of any interior voids in the alpha shape.

`A = surfaceArea(shp,RegionID)` returns the surface area of a region of the alpha shape. `RegionID` is the ID for the region and `1` ≤ `RegionID` ≤ `numRegions(shp)`.

Examples

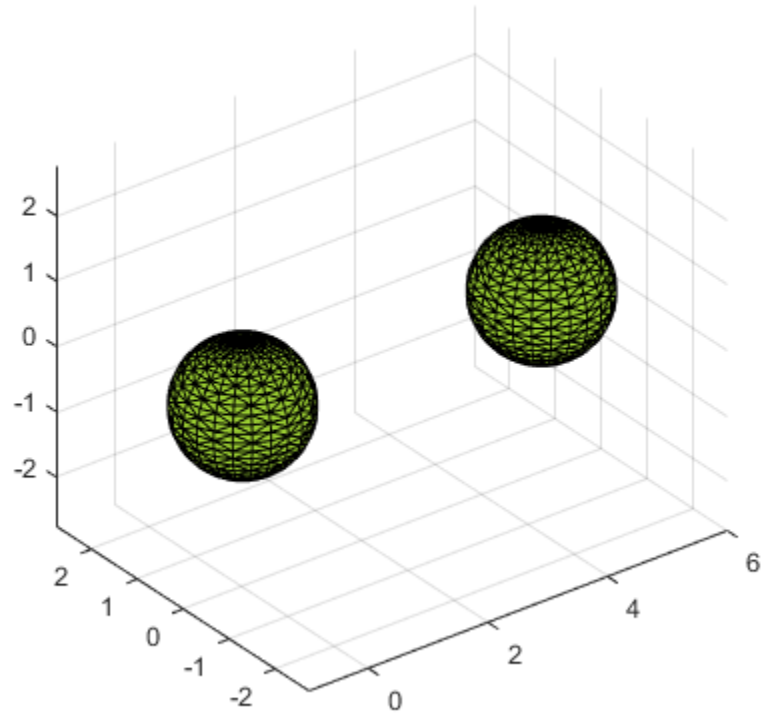
Find Surface Area of 3-D Alpha Shape

Create a set of 3-D points.

```
[x1, y1, z1] = sphere(24);
x1 = x1(:);
y1 = y1(:);
z1 = z1(:);
x2 = x1+5;
P = [x1 y1 z1; x2 y1 z1];
P = unique(P,'rows');
```

Create and plot an alpha shape using an alpha radius of 1.5.

```
shp = alphaShape(P,1.5);
plot(shp)
```



Compute the surface area of the alpha shape.

```
totalsurfarea = surfaceArea(shp)
```

```
totalsurfarea = 24.9361
```

Compute the surface area of each region separately.

```
regionsurfareas = surfaceArea(shp, 1:numRegions(shp))
```

```
regionsurfareas = 1×2

   12.4680   12.4680
```

Input Arguments

shp — 3-D alpha shape
alphaShape object

3-D alpha shape, specified as an alphaShape object. For more information, see alphaShape

Example: shp = alphaShape(x,y,z) creates a 3-D alphaShape object from the (x,y,z) point coordinates.

RegionID — ID number for a region in the alpha shape
positive integer scalar

ID number for region in alpha shape, specified as a positive integer scalar between 1 and numRegions(shp).

An alpha shape can contain several smaller regions, depending on the point set and parameters. Each of these smaller regions is assigned a unique RegionID, which numbers the regions from the largest area or volume to the smallest. For example, consider a 3-D alpha shape with two regions. The region with the largest volume has a RegionID of 1, and the smaller region has a RegionID of 2.

Example: shp.RegionThreshold = area(shp,numRegions(shp)-2); suppresses the two smallest regions in 2-D alpha shape shp.

Data Types: double

See Also

alphaShape | area | volume

Introduced in R2014b

volume

Volume of 3-D alpha shape

Syntax

```
V = volume(shp)
V = volume(shp,RegionID)
```

Description

`V = volume(shp)` returns the volume of 3-D alpha shape `shp`.

`V = volume(shp,RegionID)` returns the volume of a region of the alpha shape. `RegionID` is the ID for the region and `1` ≤ `RegionID` ≤ `numRegions(shp)`.

Examples

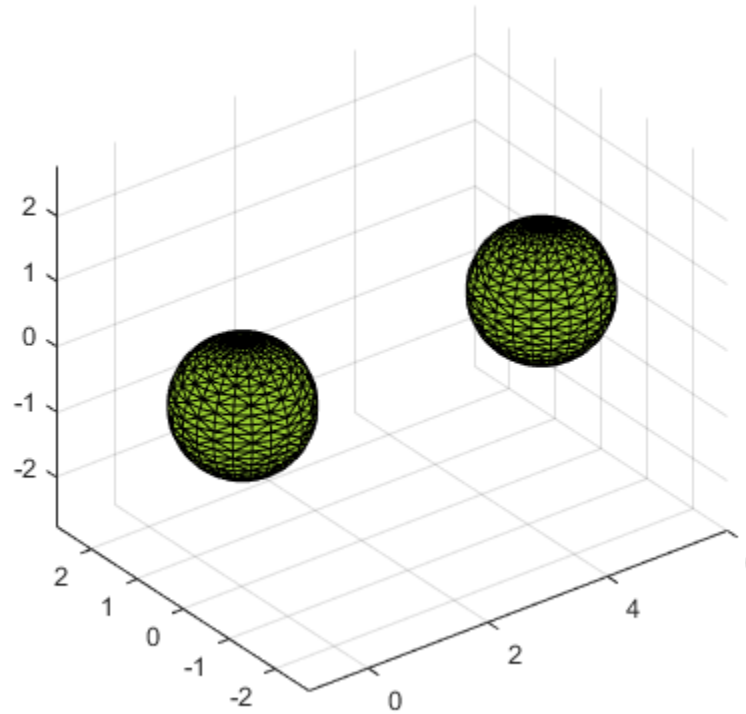
Find Volume of 3-D Alpha Shape

Create a set of 3-D points.

```
[x1, y1, z1] = sphere(24);
x1 = x1(:);
y1 = y1(:);
z1 = z1(:);
x2 = x1+5;
P = [x1 y1 z1; x2 y1 z1];
P = unique(P,'rows');
```

Create and plot an alpha shape using an alpha radius of 1.5.

```
shp = alphaShape(P,1.5);
plot(shp)
```



Compute the volume of the alpha shape.

```
totalvol = volume(shp)
```

```
totalvol = 8.2468
```

Compute the volumes of each of the two regions separately.

```
regionvols = volume(shp, 1:numRegions(shp))
```

```
regionvols = 1×2

    4.1234    4.1234
```

Input Arguments

shp — 3-D alpha shape
alphaShape object

3-D alpha shape, specified as an alphaShape object. For more information, see alphaShape

Example: shp = alphaShape(x,y,z) creates a 3-D alphaShape object from the (x,y,z) point coordinates.

RegionID — ID number for a region in the alpha shape
positive integer scalar

ID number for region in alpha shape, specified as a positive integer scalar between 1 and numRegions(shp).

An alpha shape can contain several smaller regions, depending on the point set and parameters. Each of these smaller regions is assigned a unique RegionID, which numbers the regions from the largest area or volume to the smallest. For example, consider a 3-D alpha shape with two regions. The region with the largest volume has a RegionID of 1, and the smaller region has a RegionID of 2.

Example: shp.RegionThreshold = area(shp,numRegions(shp)-2); suppresses the two smallest regions in 2-D alpha shape shp.

Data Types: double

See Also

alphaShape | area | surfaceArea

Introduced in R2014b

amd

Approximate minimum degree permutation

Syntax

```
P = amd(A)
P = amd(A,opts)
```

Description

`P = amd(A)` returns the approximate minimum degree permutation vector for the sparse matrix `C = A + A'`. The Cholesky factorization of `C(P,P)` or `A(P,P)` tends to be sparser than that of `C` or `A`. The `amd` function tends to be faster than `symamd`, and also tends to return better orderings than `symamd`. Matrix `A` must be square. If `A` is a full matrix, then `amd(A)` is equivalent to `amd(sparse(A))`.

`P = amd(A,opts)` allows additional options for the reordering. The `opts` input is a structure with the two fields shown below. You only need to set the fields of interest:

- **dense** — A nonnegative scalar value that indicates what is considered to be dense. If A is n-by-n, then rows and columns with more than `max(16,(dense*sqrt(n)))` entries in `A + A'` are considered to be "dense" and are ignored during the ordering. MATLAB software places these rows and columns last in the output permutation. The default value for this field is 10.0 if this option is not present.
- **aggressive** — A scalar value controlling aggressive absorption. If this field is set to a nonzero value, then aggressive absorption is performed. This is the default if this option is not present.

MATLAB software performs an assembly tree post-ordering, which is typically the same as an elimination tree post-ordering. It is not always identical because of the approximate degree update used, and because “dense” rows and columns do not take part in the post-order. It well-suited for a subsequent `chol` operation, however, If you require a precise elimination tree post-ordering, you can use the following code:

```
P = amd(S);
C = spones(S)+spones(S');
```

```
[ignore, Q] = etree(C(P,P));
P = P(Q);
```

If S is already symmetric, omit the second line, `C = spones(S)+spones(S')`.

Examples

Effect of Preordering on Sparsity of Cholesky Factors

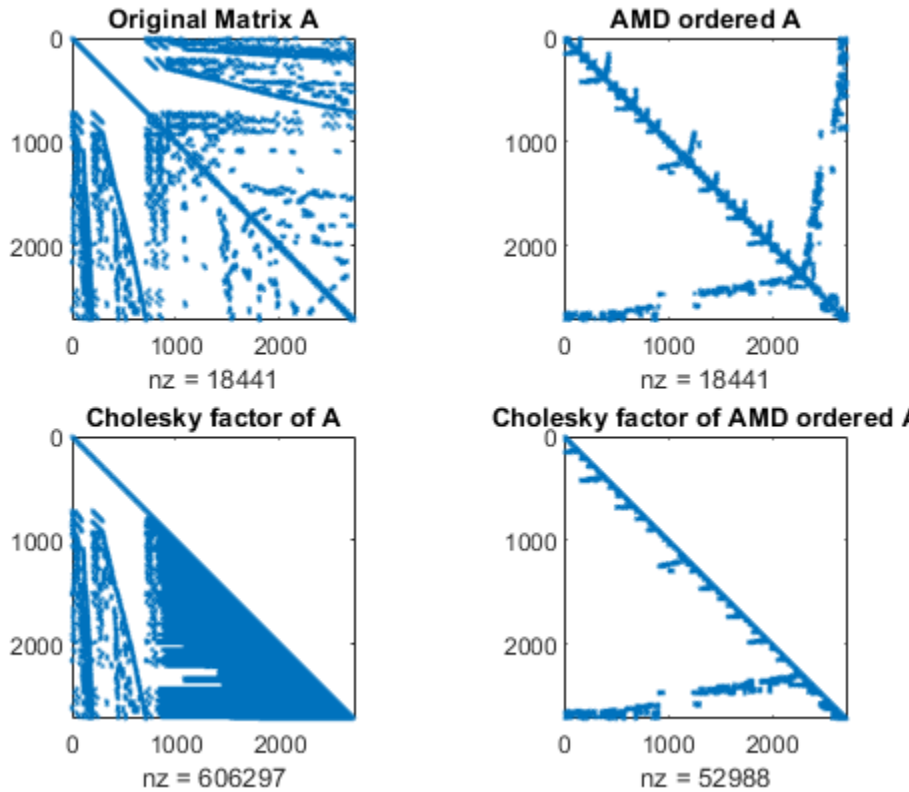
Compute the Cholesky factor of a matrix before and after it is ordered using `amd` to examine the effect on sparsity.

Load the barbell graph sparse matrix and add a sparse identity matrix to it to ensure it is positive definite. Compute two Cholesky factors: one of the original matrix and one of the original matrix preordered with `amd`.

```
load barbellgraph.mat
A = A+speye(size(A));
p = amd(A);
L = chol(A,'lower');
Lp = chol(A(p,p),'lower');
```

Plot the sparsity patterns of all four matrices. The Cholesky factor obtained from the preordered matrix is much sparser compared to the factor of the matrix in its natural ordering.

```
figure
subplot(2,2,1)
spy(A)
title('Original Matrix A')
subplot(2,2,2)
spy(A(p,p))
title('AMD ordered A')
subplot(2,2,3)
spy(L)
title('Cholesky factor of A')
subplot(2,2,4)
spy(Lp)
title('Cholesky factor of AMD ordered A')
```



See Also

`colamd` | `colperm` | `dissect` | `symamd` | `symrcm`

analyzeCodeCompatibility

Create code compatibility analysis results

Syntax

```
r = analyzeCodeCompatibility
r = analyzeCodeCompatibility(names)
r = analyzeCodeCompatibility( ___ ,'IncludeSubfolders',tf)
```

Description

`r = analyzeCodeCompatibility` creates code compatibility analysis results for the current working folder and subfolders and returns it as a `CodeCompatibilityAnalysis` object.

`r = analyzeCodeCompatibility(names)` analyzes the files or folders specified by `names`.

`r = analyzeCodeCompatibility( ___ ,'IncludeSubfolders',tf)` specifies whether to include subfolders in the analysis. By default, subfolders are included in the analysis. Use this syntax with any of the arguments in previous syntaxes.

Examples

Analyze Code in Current Folder

Create code compatibility analysis results for code in your current folder and subfolders.

```
r = analyzeCodeCompatibility
```

```
r = 

  CodeCompatibilityAnalysis with properties:

               Date: 20-Apr-2017 15:06:06
```

```
      MATLABVersion: "R2017b"
              Files: [92×1 string]
    ChecksPerformed: [299×6 table]
    Recommendations: [28×7 table]
```

Analyze the code again and omit subfolders from the analysis.

```
r = analyzeCodeCompatibility('IncludeSubfolders',false)
```

```
r = 

  CodeCompatibilityAnalysis with properties:

               Date: 20-Apr-2017 15:06:56
      MATLABVersion: "R2017b"
              Files: [5×1 string]
    ChecksPerformed: [299×6 table]
    Recommendations: [1×7 table]
```

Input Arguments

names — Name of files or folders to analyze
character vector | cell array of character vectors | string scalar | string array

Name of files or folders to analyze, specified as a character vector, cell array of character vectors, string scalar, or string array.

The name of a file must be a valid MATLAB code or App file (`.m`, `.mlx`, or `.mlapp`).

Example: `'../thisFile.m'`

Example: `{'folderA','folderB','fileA'}`

Data Types: `char` | `string`

`tf` — Subfolder analysis indicator
`true` (default) | `false` | 0 | 1

Subfolder analysis indicator, specified as `true`, `false`, 0, or 1. By default, `analyzeCodeCompatibility` includes subfolders in the analysis. To omit subfolders, set `tf` to `false` or 0.

See Also

`CodeCompatibilityAnalysis` | `codeCompatibilityReport`

Topics

"MATLAB Code Compatibility Report"

Introduced in R2017b

ancestor

Ancestor of graphics object

Syntax

```
p = ancestor(h,type)
p = ancestor(h,type,'toplevel')
```

Description

`p = ancestor(h,type)` returns the handle of the closest ancestor of `h`, if the ancestor is one of the types of graphics objects specified by `type`. `type` can be:

- a character vector with the name of a single type of object, for example, `'figure'`.
- a cell array containing the names of multiple objects, for example, `{'hgtransform','hggroup','axes'}`.

If MATLAB cannot find an ancestor of `h` that is one of the specified types, then `ancestor` returns `p` as empty. When `ancestor` searches the hierarchy, it includes the object itself in the search. Therefore, if the object with handle `h` is of one of the types listed in `type`, `ancestor` will return object `h`.

`ancestor` returns `p` as empty but does not issue an error if `h` is not a graphics object.

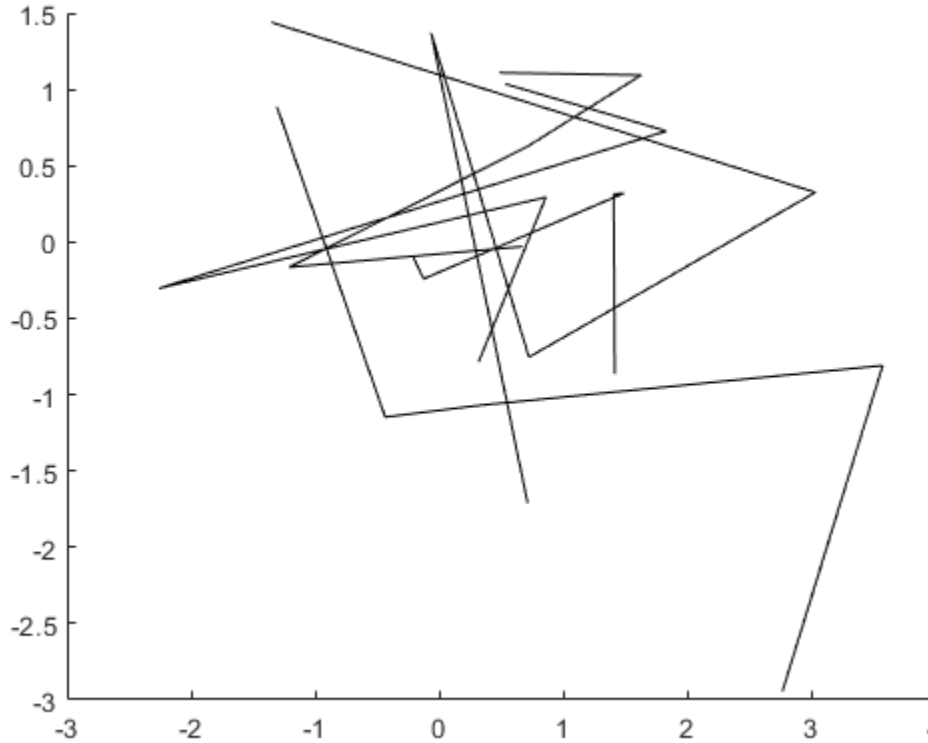
`p = ancestor(h,type,'toplevel')` returns the highest-level ancestor of `h`, if this type appears in the `type` argument.

Examples

Return Top-Level Ancestor

Create `Line` objects and parent them to a `Group` object. Then return the top-level ancestor.

```
g = hggroup;
ln = line(randn(5),randn(5),'Parent',g);
```




```
tp = ancestor(g,{'figure','axes','hggroup'},'toplevel')
```

```
tp = 
  Figure (LiveEditorRunTimeFigure) with properties:

      Number: 1
        Name: ''
       Color: [0.9400 0.9400 0.9400]
    Position: [360 502 560 420]
       Units: 'pixels'
```

Show all properties

See Also

findobj

Introduced before R2006a

and, &

Find logical AND

Syntax

```
A & B
and(A,B)
```

Description

`A & B` performs a logical AND of arrays `A` and `B` and returns an array containing elements set to either logical 1 (`true`) or logical 0 (`false`). An element of the output array is set to logical 1 (`true`) if both `A` and `B` contain a nonzero element at that same array location. Otherwise, the array element is set to 0.

`and(A,B)` is an alternate way to execute `A & B`, but is rarely used. It enables operator overloading for classes.

Examples

Locate Nonzero Values

Find the logical AND of two matrices. The result contains logical `1` (`true`) only where both matrices contain nonzero values.

```
A = [5 7 0; 0 2 9; 5 0 0]
```

```
A = 3×3

     5     7     0
     0     2     9
     5     0     0
```

```
B = [6 6 0; 1 3 5; -1 0 0]
```

```
B = 3×3

     6     6     0
     1     3     5
    -1     0     0
```

```
A & B
```

```
ans = 3x3 logical array

   1   1   0
   0   1   1
   1   0   0
```

Truth Table for Logical AND

Create a truth table for `and`.

```
A = [true false]
```

```
A = 1x2 logical array

   1   0
```

```
B = [true; false]
```

```
B = 2x1 logical array

   1
   0
```

```
C = A&B
```

```
C = 2x2 logical array

   1   0
   0   0
```

Input Arguments

A, B — Operands
scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs `A` and `B` must either be the same size or have sizes that are compatible (for example, `A` is an `M`-by-`N` matrix and `B` is a scalar or `1`-by-`N` row vector). For more information, see "Compatible Array Sizes for Basic Operations".

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Tips

- You can chain together several logical operations, for example, `A & B | C`.
- The symbols & and && perform different operations in MATLAB. The element-wise AND operator described here is &. The short-circuit AND operator is &&.
- When you use the element-wise & and `|` operators in the context of an `if` or `while` loop expression (and only in that context), they use short-circuiting to evaluate expressions. Otherwise, you must specify && or `||` to opt-in to short-circuiting behavior. See `Logical Operators: Short Circuit` for more information.

Compatibility Considerations

Implicit expansion change affects arguments for operators

Behavior changed in R2016b

Starting in R2016b with the addition of implicit expansion, some combinations of arguments for basic operations that previously returned errors now produce results. For example, you previously could not add a row and a column vector, but those operands are now valid for addition. In other words, an expression like `[1 2] + [1; 2]` previously returned a size mismatch error, but now it executes.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a `try`/`catch` block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see "Compatible Array Sizes for Basic Operations".

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see "Tall Arrays".

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see "Run MATLAB Functions on a GPU" (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see "Run MATLAB Functions with Distributed Arrays" (Parallel Computing Toolbox).

See Also

`all` | `any` | `bitand` | `not` | `or` | `xor`

Topics

"Find Array Elements That Meet a Condition"
"MATLAB Operators and Special Characters"

Introduced before R2006a

angle

Phase angle

Syntax

```
theta = angle(z)
```

Description

`theta = angle(z)` returns the phase angle in the interval [-π,π] for each element of a complex array `z`. The angles in `theta` are such that `z = abs(z).*exp(i*theta)`.

Examples

Magnitude and Phase of Complex Number

Create a complex number, and compute its magnitude and phase.

```
z = 2*exp(i*0.5)
```

```
z = 1.7552 + 0.9589i
```

```
r = abs(z)
```

```
r = 2
```

```
theta = angle(z)
```

```
theta = 0.5000
```

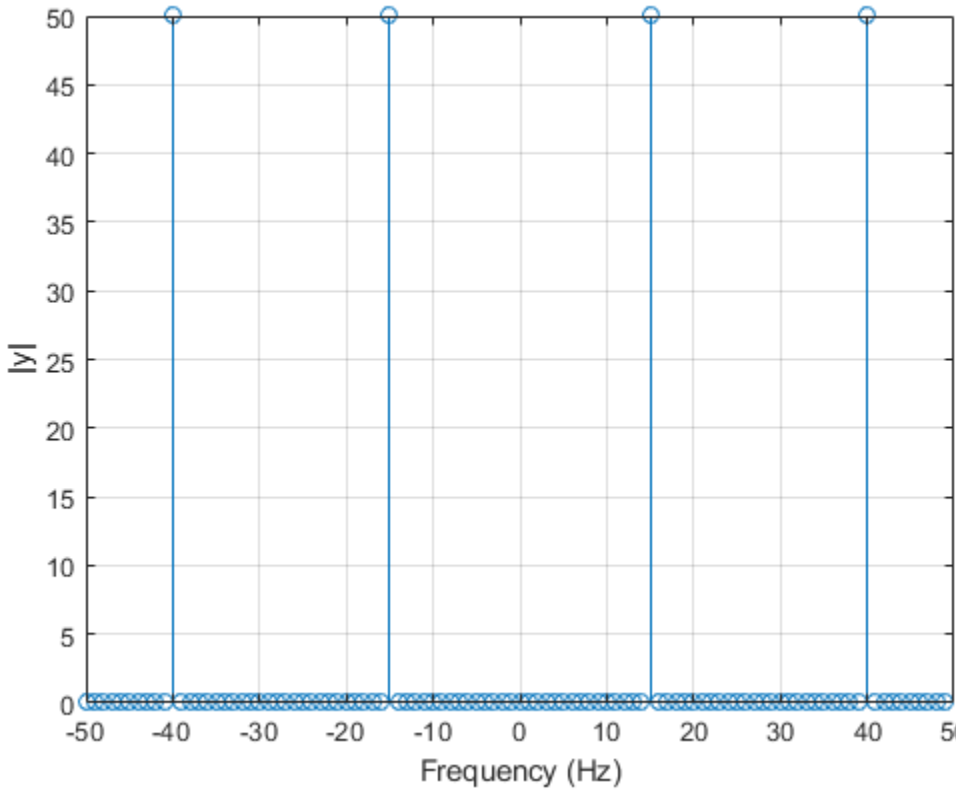
FFT Phase

Create a signal that consists of two sinusoids of frequencies 15 Hz and 40 Hz. The first sinusoid has a phase of $-\pi/4$, and the second has a phase of $\pi/2$. Sample the signal at 100 Hz for one second.

```
fs = 100;
t = 0:1/fs:1-1/fs;
x = cos(2*pi*15*t - pi/4) - sin(2*pi*40*t);
```

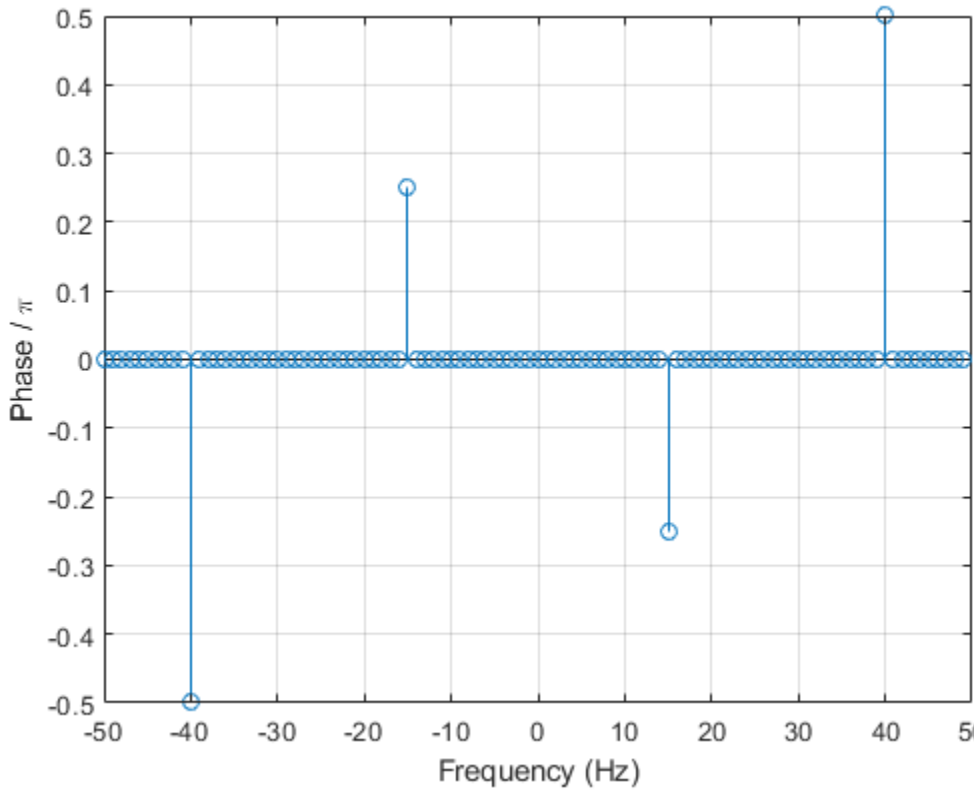
Compute the Fourier transform of the signal. Plot the magnitude of the transform as a function of frequency.

```
y = fft(x);
z = fftshift(y);

ly = length(y);
f = (-ly/2:ly/2-1)/ly*fs;

stem(f,abs(z))
xlabel 'Frequency (Hz)'
ylabel '|y|'
grid
```



Compute the phase of the transform, removing small-magnitude transform values. Plot the phase as a function of frequency.

```
tol = 1e-6;
z(abs(z) < tol) = 0;

theta = angle(z);

stem(f,theta/pi)
xlabel 'Frequency (Hz)'
ylabel 'Phase / \pi'
grid
```



Input Arguments

z — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. When the elements of `z` are non-negative real numbers, `angle` returns 0. When the elements of `z` are negative real numbers, `angle` returns π.

Data Types: `double` | `single`
Complex Number Support: Yes

Algorithms

`angle` takes a complex number $z = x + iy$ and uses the `atan2` function to compute the angle between the positive x-axis and a ray from the origin to the point (x,y) in the xy-plane.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`abs` | `atan2` | `unwrap`

Introduced before R2006a

animatedline

Create animated line

Syntax

```
an = animatedline
an = animatedline(x,y)
an = animatedline(x,y,z)
```

```
an = animatedline( ___ ,Name,Value)
an = animatedline(ax, ___ )
```

Description

`an = animatedline` creates an animated line that has no data and adds it to the current axes. Create an animation by adding points to the line in a loop using the `addpoints` function.

`an = animatedline(x,y)` creates an animated line with initial data points defined by `x` and `y`.

`an = animatedline(x,y,z)` creates an animated line with initial data points defined by `x`, `y`, and `z`.

`an = animatedline( ___ ,Name,Value)` specifies animated line properties using one or more name-value pair arguments. For example, `'Color','r'` sets the line color to red. Use this option after any of the input argument combinations in the previous syntaxes.

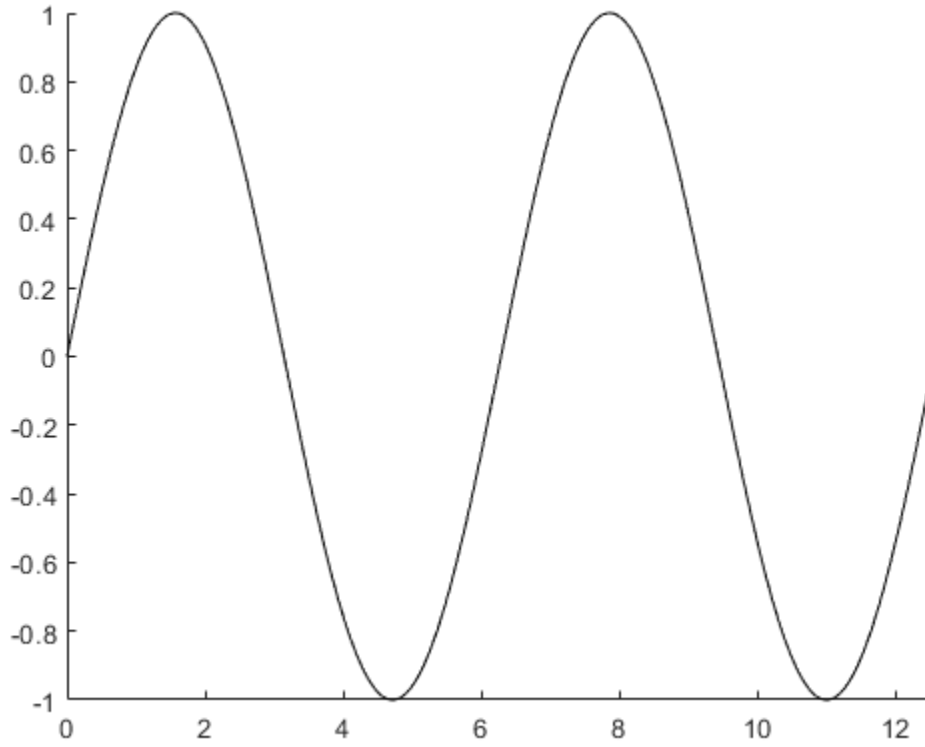
`an = animatedline(ax, ___ )` creates the line in the axes or geographic axes specified by `ax` instead of in the current axes (`gca`). The option `ax` can precede any of the input argument combinations in the previous syntaxes.

Examples

Display Line Animation

Create the initial animated line object. Then, use a loop to add 1,000 points to the line. After adding each new point, use `drawnow` to display the new point on the screen.

```
h = animatedline;
axis([0,4*pi,-1,1])

x = linspace(0,4*pi,1000);
y = sin(x);
for k = 1:length(x)
    addpoints(h,x(k),y(k));
    drawnow
end
```



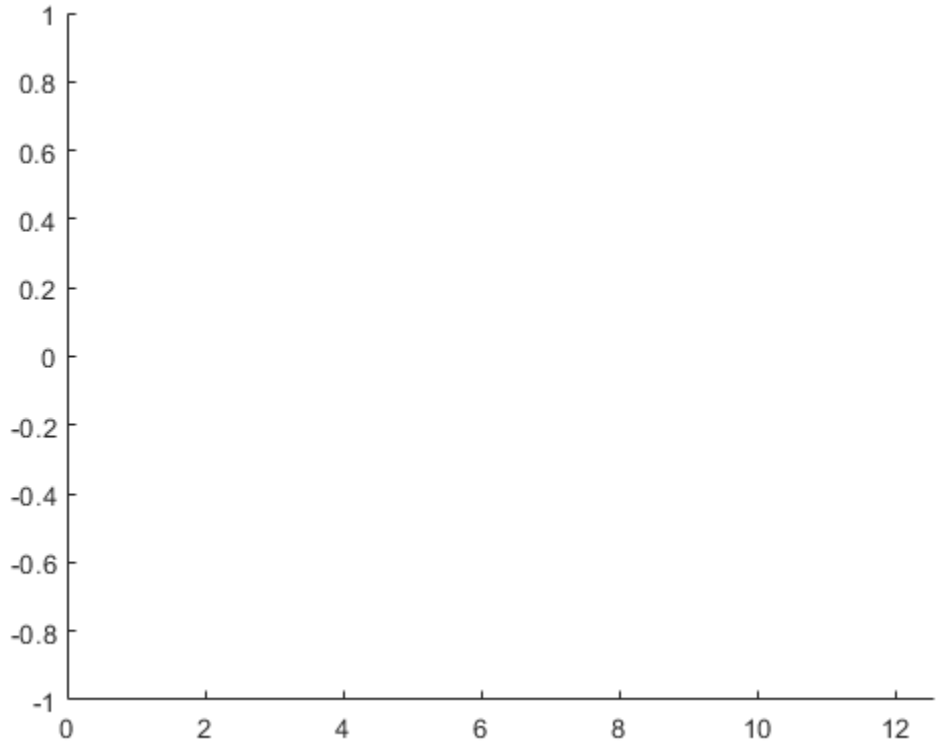
For faster rendering, add more than one point to the line each time through the loop or use `drawnow limitrate`.

Query the points of the line.

```
[xdata,ydata] = getpoints(h);
```

Clear the points from the line.

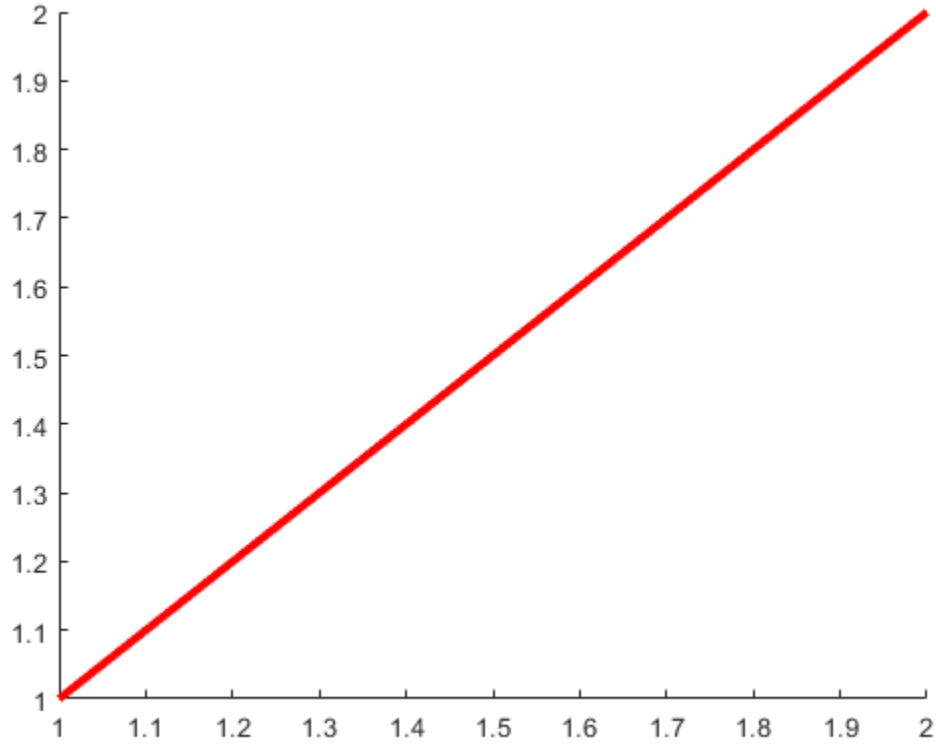
```
clearpoints(h)
drawnow
```



Specify Animated Line Color

Set the color of the animated line to red and set its line width to 3 points.

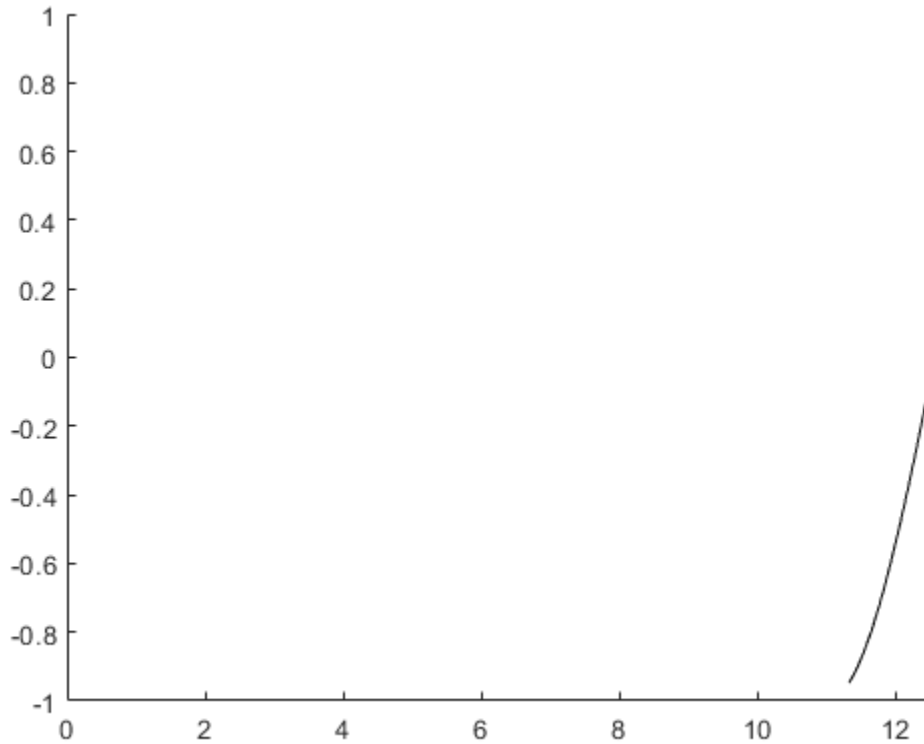
```
x = [1 2];
y = [1 2];
h = animatedline(x,y,'Color','r','LineWidth',3);
```



Set Maximum Number of Points

Limit the number of points in the animated line to 100. Use a loop to add one point to the line at a time. When the line contains 100 points, adding a new point to the line deletes the oldest point.

```
h = animatedline('MaximumNumPoints',100);
axis([0,4*pi,-1,1])

x = linspace(0,4*pi,1000);
y = sin(x);
for k = 1:length(x)
    addpoints(h,x(k),y(k));
    drawnow
end
```

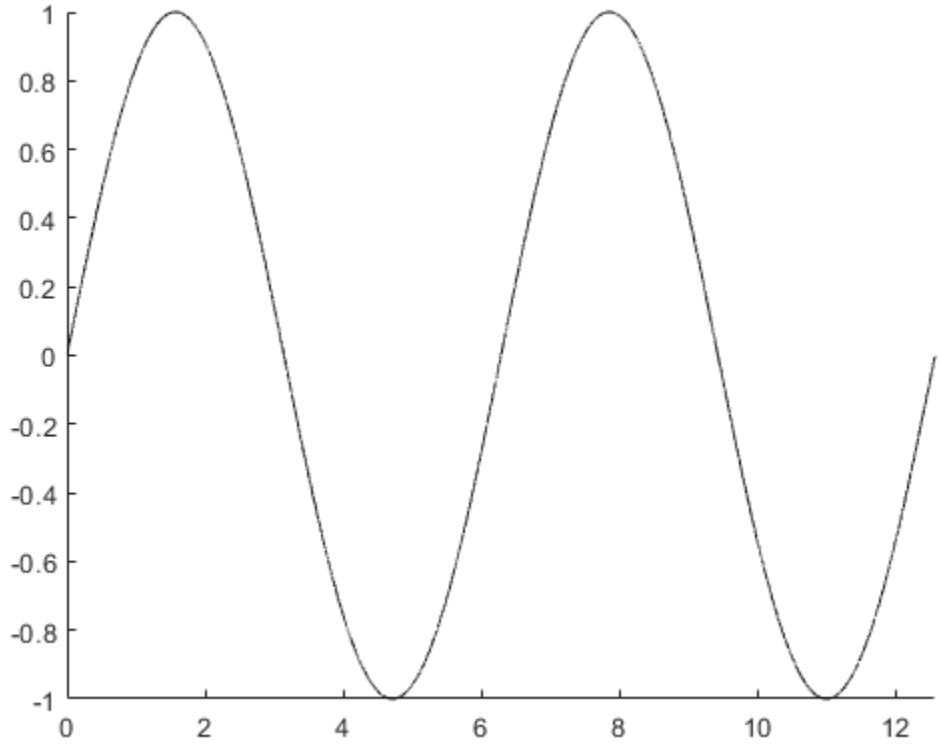


Add Points in Sets for Fast Animation

Use a loop to add 100,000 points to an animated line. Since the number of points is large, adding one point to the line each time through the loop might be slow. Instead, add 100 points to the line each time through the loop for a faster animation.

```
h = animatedline;
axis([0,4*pi,-1,1])

numpoints = 100000;
x = linspace(0,4*pi,numpoints);
y = sin(x);
```

```
for k = 1:100:numpoints-99
    xvec = x(k:k+99);
    yvec = y(k:k+99);
    addpoints(h,xvec,yvec)
    drawnow
end
```

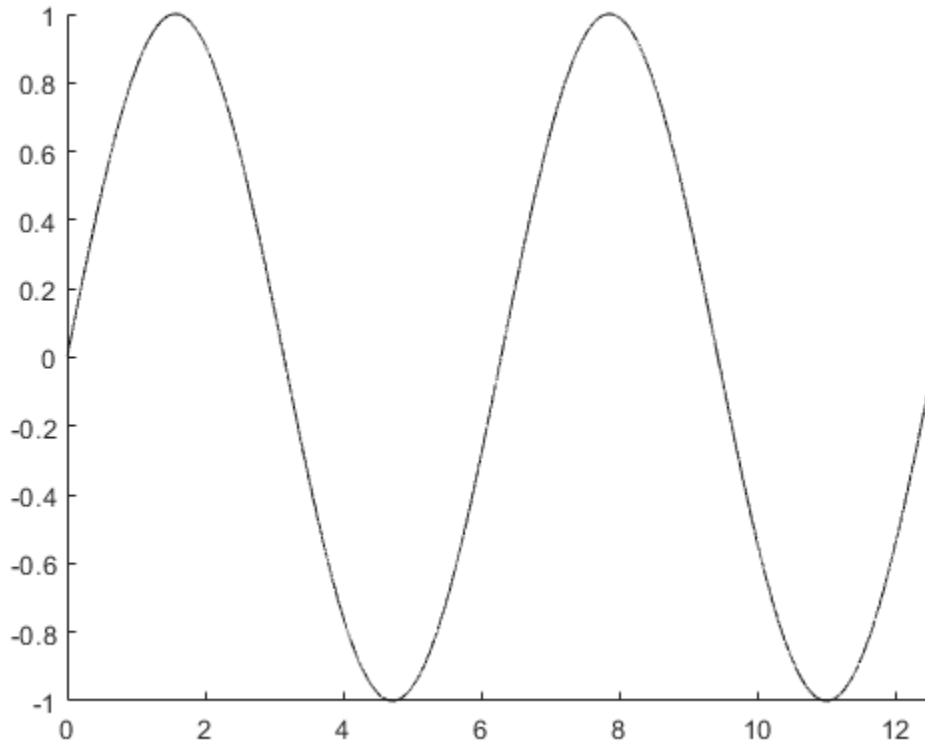


Another technique for creating faster animations is to use `drawnow limitrate` instead of `drawnow`.

Use drawnow limitrate for Fast Animation

Use a loop to add 100,000 points to an animated line. Since the number of points is large, using `drawnow` to display the changes might be slow. Instead, use `drawnow limitrate` for a faster animation.

```
h = animatedline;
axis([0,4*pi,-1,1])

numpoints = 100000;
x = linspace(0,4*pi,numpoints);
y = sin(x);
for k = 1:numpoints
    addpoints(h,x(k),y(k))
    drawnow limitrate
end
```



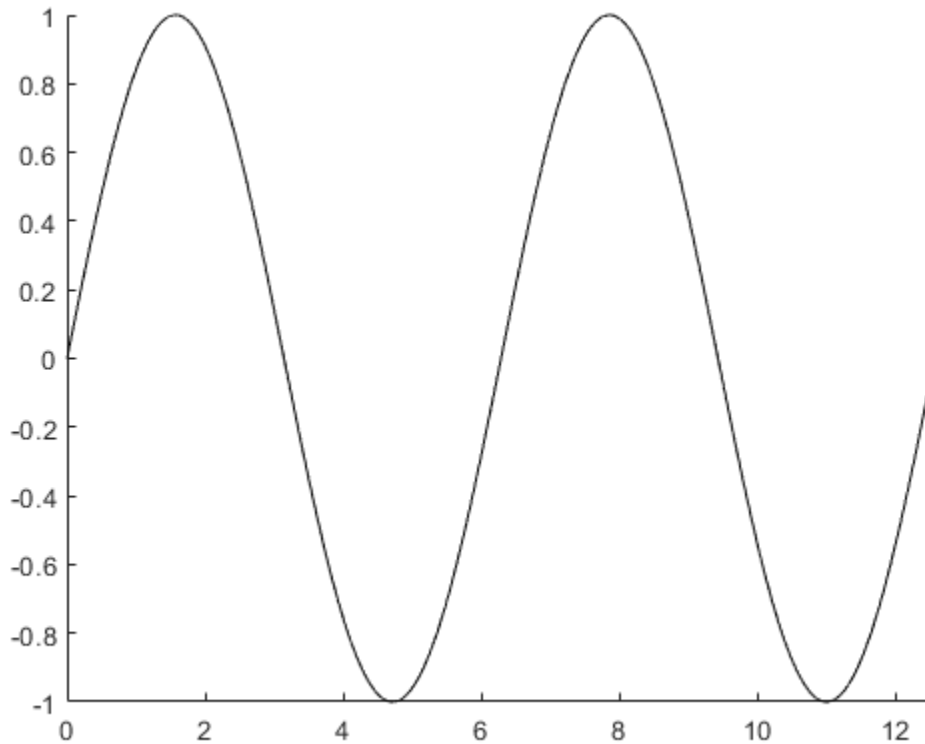
Control Animation Speed

Control the animation speed by running through several iterations of the animation loop before drawing the updates on the screen. Use this technique when `drawnow` is too slow and `drawnow limitrate` is too fast.

For example, update the screen every 1/30 seconds. Use the `tic` and `toc` commands to keep track of how much time passes between screen updates.

```
h = animatedline;
axis([0,4*pi,-1,1])
numpoints = 10000;
```

```
x = linspace(0,4*pi,numpoints);
y = sin(x);
a = tic; % start timer
for k = 1:numpoints
    addpoints(h,x(k),y(k))
    b = toc(a); % check timer
    if b > (1/30)
        drawnow % update screen every 1/30 seconds
        a = tic; % reset timer after updating
    end
end
drawnow % draw final frame
```



A smaller interval updates the screen more often and results in a slower animation. For example, use `b > (1/1000)` to slow down the animation.

Input Arguments

x — Initial first coordinate values
`[]` (default) | scalar or vector

Initial first coordinate values for the animated line, specified as a scalar or vector. For Cartesian axes, the first coordinate is *x*-axis position. For geographic axes, the first coordinate is latitude in degrees.

The input `x` must be equal in size to `y`.

Example: `1:10`

Data Types: `double`

y — Initial second coordinate values
`[]` (default) | scalar or vector

Initial second coordinate values for the animated line, specified as a scalar or vector. For Cartesian axes, the first coordinate is *y*-axis position. For geographic axes, the second coordinate is longitude in degrees.

The input `y` must be equal in size to `x`.

Example: `1:10`

Data Types: `double`

z — Initial third coordinate values
`[]` (default) | scalar or vector

Initial third coordinate values for the animated line, specified as a scalar or vector. For Cartesian axes, the third coordinate is *z*-axis position.

Example: `1:10`

Data Types: `double`

ax — Target axes
`Axes` object | `GeographicAxes` object

Target axes, specified as an `Axes` object or `GeographicAxes` object. If you do not specify an axes object, then `animatedline` uses the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The animated line properties listed here are only a subset. For a complete list, see AnimatedLine.

Example: `'Color','red','Marker','o'`

`Color` — Line color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
'-'	Solid line	
'--'	Dashed line	
':'	Dotted line	

Line Style	Description	Resulting Line
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

`Marker` — Marker symbol
`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | ...

Marker symbol, specified as one of the marker symbols listed in this table. By default, the animated line object does not have markers. Specifying a marker symbol adds markers at each data point or vertex.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerSize` — Marker size

6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

`MarkerEdgeColor` — Marker outline color

`'auto'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'`

Marker outline color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'auto'` uses the same color as the `Color` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'`

Marker fill color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` option uses the same color as the `Color` property of the parent axes. If you specify `'auto'` and the axes plot box is invisible, the marker fill color is the color of the figure.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`MaximumNumPoints` — Maximum number of points stored and displayed
`1000000` (default) | positive value | `Inf`

Maximum number of points stored and displayed as part of the line, specified as a positive value or `Inf`. By default, the value is one million points. If the number of points exceeds the maximum value permitted, then the animated line keeps the most recently added points and drops points from the beginning of the line. These dropped points no longer display on the screen and are not returned when using `getpoints`.

Use this property to limit the number of points appearing on the screen at any given time or to limit the amount of memory used. If you specify the value as `Inf`, then the animated

line does not drop any points, but the number of points stored is limited by the amount of memory available.

Example: `10`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`an` — `AnimatedLine` object
`AnimatedLine` object

`AnimatedLine` object. Use `an` to modify the `AnimatedLine` object after its been created, such as changing property values or adding points to the line. For a list of properties, see AnimatedLine.

Tips

- Animated lines do not support data tips.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions
`addpoints` | `clearpoints` | `getpoints`

Properties
AnimatedLine

Introduced in R2014b

AnimatedLine Properties

Line animation appearance and behavior

Description

`AnimatedLine` properties control the appearance and behavior of an `AnimatedLine` object. By changing property values, you can modify certain aspects of the line. Use dot notation to refer to a particular object and property:

```
an = animatedline;
c = an.Color;
an.Color = 'red';
```

Properties

Color and Styling

`Color` — Line color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
'-'	Solid line	

Line Style	Description	Resulting Line
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

`MaximumNumPoints` — Maximum number of points stored and displayed
`1000000` (default) | positive value | `Inf`

Maximum number of points stored and displayed as part of the line, specified as a positive value or `Inf`. By default, the value is one million points. If the number of points exceeds the maximum value permitted, then the animated line keeps the most recently added points and drops points from the beginning of the line. These dropped points no longer display on the screen and are not returned when using `getpoints`.

Use this property to limit the number of points appearing on the screen at any given time or to limit the amount of memory used. If you specify the value as `Inf`, then the animated line does not drop any points, but the number of points stored is limited by the amount of memory available.

Example: `10`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`AlignVertexCenters` — Sharp vertical and horizontal lines
`'off'` (default) | `'on'`

Sharp vertical and horizontal lines, specified as `'off'` or `'on'`.

If the associated figure has a `GraphicsSmoothing` property set to `'on'` and a `Renderer` property set to `'opengl'`, then the figure applies a smoothing technique to plots. In some cases, this smoothing technique can cause vertical and horizontal lines to

appear uneven in thickness or color. Use the `AlignVertexCenters` property to eliminate the uneven appearance.

- `'off'` — Do not sharpen vertical or horizontal lines. The lines might appear uneven in thickness or color.
- `'on'` — Sharpen vertical and horizontal lines to eliminate an uneven appearance.

Note You must have a graphics card that supports this feature. To see if the feature is supported, call the `rendererinfo` function. If it is supported, `rendererinfo` returns value of `1` for `info.Details.SupportsAlignVertexCenters`.

Markers

`Marker` — Marker symbol

`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | ...

Marker symbol, specified as one of the values listed in this table. By default, the object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)

Value	Description
`'none'`	No markers

`MarkerSize` — Marker size

6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

`MarkerEdgeColor` — Marker outline color

`'auto'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'auto'` uses the same color as the `Color` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` option uses the same color as the `Color` property of the parent axes. If you specify `'auto'` and the axes plot box is invisible, the marker fill color is the color of the figure.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Legend

`DisplayName` — Legend label

`''` (default) | character vector | string scalar

Legend label, specified as a character vector or string scalar. The legend does not display until you call the `legend` command. If you do not specify the text, then `legend` sets the label using the form `'dataN'`.

Annotation — Control for including or excluding object from legend

`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

Visible — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

UIContextMenu — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

`Clipping` — Clipping of object to axes limits

`'on'` (default) | `'off'`

Clipping of the object to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the object that are outside the axes limits.
- `'off'` — Display the entire object, even if parts of it appear outside the axes limits. Parts of the object might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the object so that it is larger than the original plot.

The `Clipping` property of the axes that contains the object must be set to `'on'`. Otherwise, this property has no effect. For more information about the clipping behavior, see the `Clipping` property of the axes.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see "Callback Definition".

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you

do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks when visible. The `Visible` property must be set to `'on'` and you must click a part of the `AnimatedLine` object that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The `HitTest` property determines if the `AnimatedLine` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `AnimatedLine` object passes the click to the object below it in the current view of the figure window. The `HitTest` property of the `AnimatedLine` object has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `AnimatedLine` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `AnimatedLine` object that has one of these:

- `HitTest` property set to `'on'`
- `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

Note The `PickableParts` property determines if the `AnimatedLine` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent

`Axes` object | `GeographicAxes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `GeographicAxes`, `Group`, or `Transform` object.

`Children` — Children

empty `GraphicsPlaceholder` array

The object has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.

- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'animatedline'`

This property is read-only.

Type of graphics object, returned as `'animatedline'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`animatedline`

Topics
"Access Property Values"
"Graphics Object Properties"

Introduced in R2014b

annotation

Create annotations

Syntax

```
annotation(lineType,x,y)
annotation(lineType)

annotation(shapeType,dim)
annotation(shapeType)

annotation( ___ ,Name,Value)
annotation(container, ___ )

an = annotation( ___ )
```

Description

`annotation(lineType,x,y)` creates a line or arrow annotation extending between two points in the current figure. Specify `lineType` as `'line'`, `'arrow'`, `'doublearrow'`, or `'textarrow'`. Specify `x` and `y` as two-element vectors of the form `[x_begin x_end]` and `[y_begin y_end]`, respectively.

`annotation(lineType)` creates the annotation in the default position between the points `(0.3,0.3)` and `(0.4,0.4)`.

`annotation(shapeType,dim)` creates a rectangle, ellipse, or text box annotation with a particular size and location in the current figure. Specify `shapeType` as `'rectangle'`, `'ellipse'`, or `'textbox'`. Specify `dim` as a four-element vector of the form `[x y w h]`. The `x` and `y` elements determine the position and the `w` and `h` elements determine the size.

`annotation(shapeType)` creates the annotation in the default position so that the lower left corner is at `(0.3,0.3)` and the width and height are both `0.1`.

`annotation( ___ ,Name,Value)` creates the annotation and specifies properties as name-value pair arguments. Different types of annotations support different properties.

You can specify properties with any of the input argument combinations in the previous syntaxes.

`annotation(container, ___ )` creates the annotation in the figure, uipanel, or uitab specified by `container`, instead of in the current figure.

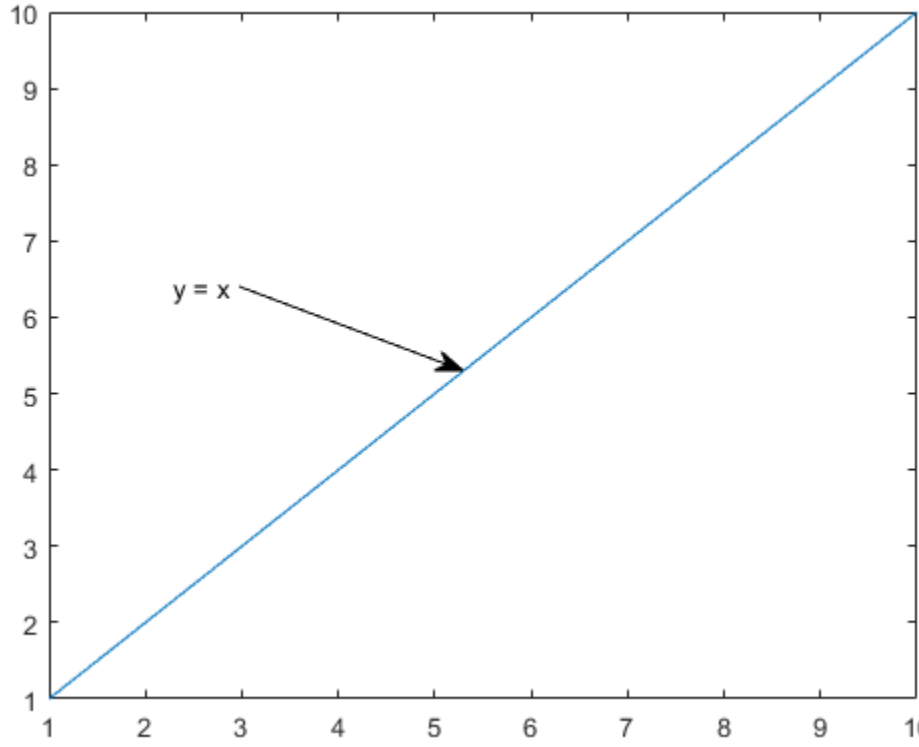
`an = annotation( ___ )` returns the annotation object. The type of object returned depends on first input argument. Use `an` to modify properties of the object after it is created. You can specify an output argument with any of the previous syntaxes.

Examples

Create Text Arrow Annotation

Create a simple line plot and add a text arrow to the figure. Specify the text arrow location in normalized figure coordinates, starting at the point `(0.3,0.6)` and ending at `(0.5,0.5)`. Specify the text description by setting the `String` property.

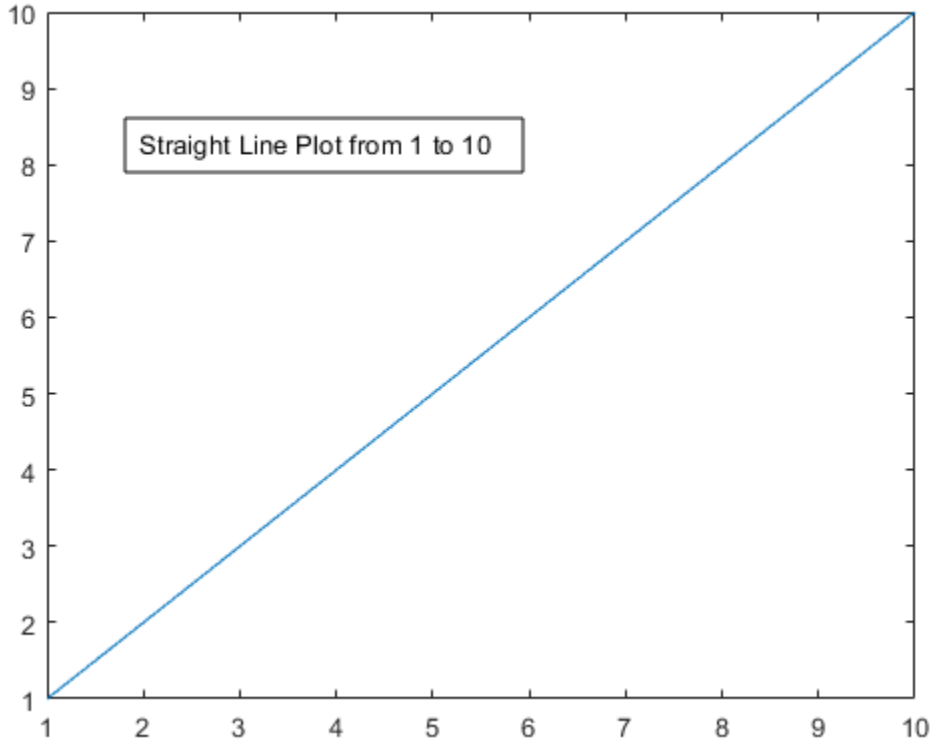
```
figure
plot(1:10)
x = [0.3 0.5];
y = [0.6 0.5];
annotation('textarrow',x,y,'String','y = x ')
```



Create Text Box Annotation

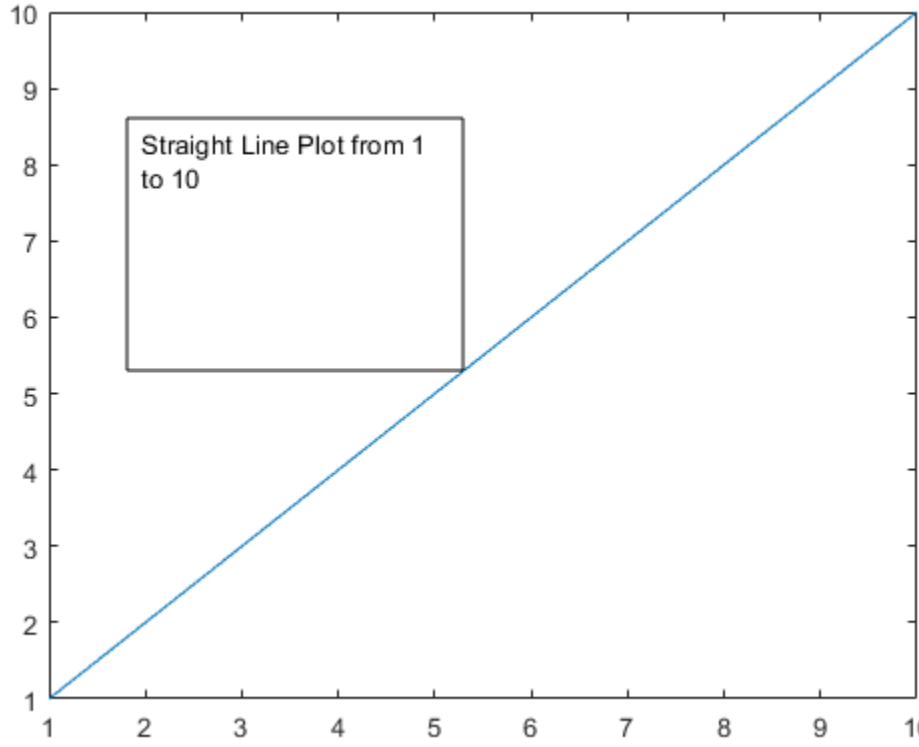
Create a simple line plot and add a text box annotation to the figure. Specify the text description by setting the `String` property. Force the box to fit tightly around the text by setting the `FitBoxToText` property to `'on'`.

```
figure
plot(1:10)
dim = [.2 .5 .3 .3];
str = 'Straight Line Plot from 1 to 10';
annotation('textbox',dim,'String',str,'FitBoxToText','on');
```



Create a text box annotation without setting the `FitBoxToText` property. The text box uses the specified width and height and wraps text as needed.

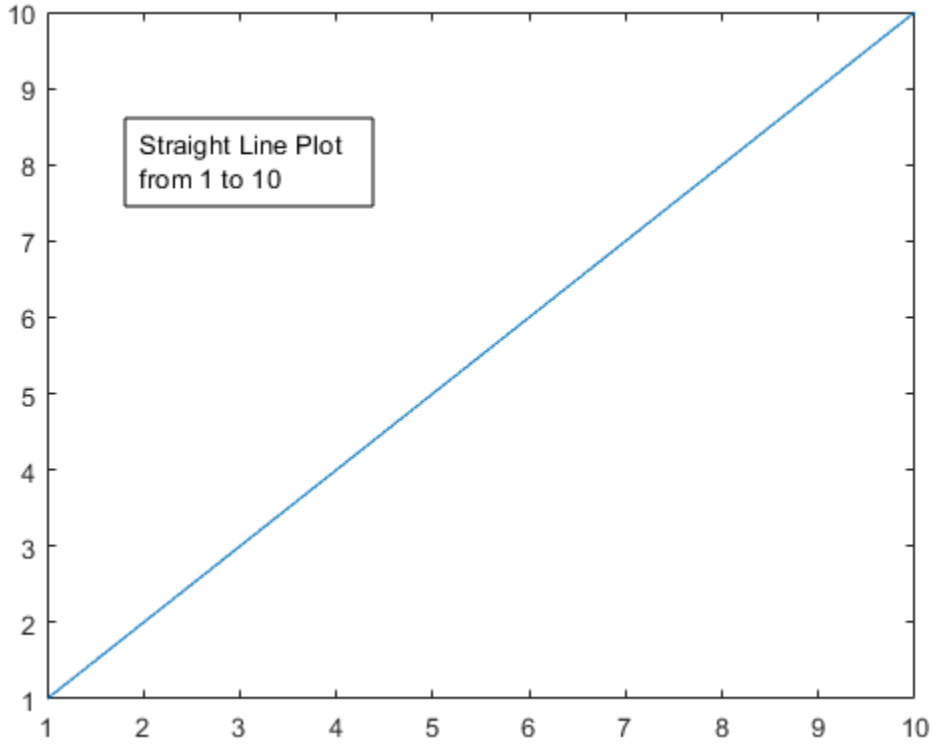
```
figure
plot(1:10)
dim = [.2 .5 .3 .3];
str = 'Straight Line Plot from 1 to 10';
annotation('textbox',dim,'String',str)
```



Create Text Box Annotation with Multiline Text

Create a text box annotation with multiline text by setting the `String` property to a cell array. Each element of the cell array displays on a separate line. Force the box to fit tightly around the text by setting the `FitBoxToText` property to `'on'`.

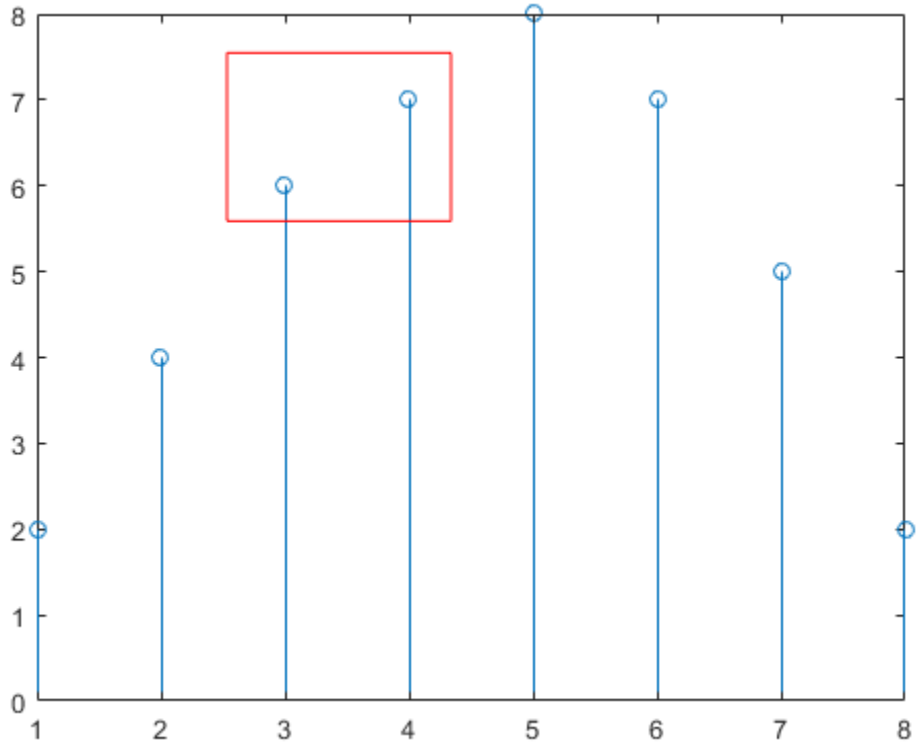
```
figure
plot(1:10)
dim = [0.2 0.5 0.3 0.3];
str = {'Straight Line Plot','from 1 to 10'};
annotation('textbox',dim,'String',str,'FitBoxToText','on');
```



Create Rectangle Annotation

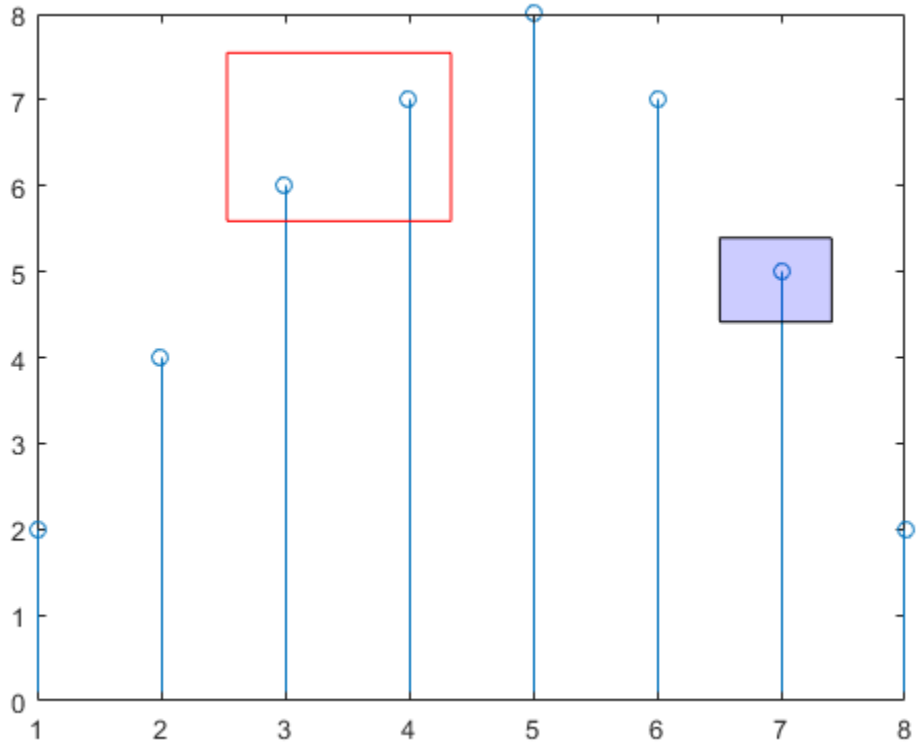
Create a stem plot and add a rectangle annotation to the figure. Change the color of the rectangle outline by specifying the `Color` property.

```
figure
data = [2 4 6 7 8 7 5 2];
stem(data)
dim = [.3 .68 .2 .2];
annotation('rectangle',dim,'Color','red')
```



Add a second rectangle annotation to the figure. Specify the fill color by setting the `FaceColor` property. Add transparency by setting the `FaceAlpha` property to a value between 0 (completely transparent) and 1 (completely opaque).

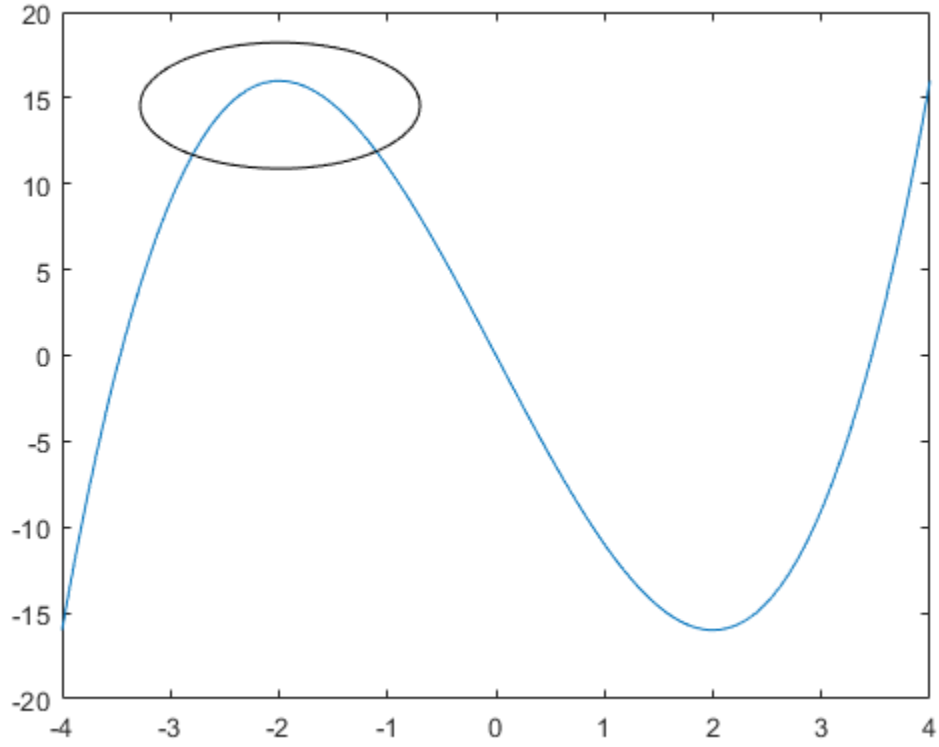
```
dim2 = [.74 .56 .1 .1];
annotation('rectangle',dim2,'FaceColor','blue','FaceAlpha',.2)
```



Create Ellipse Annotation

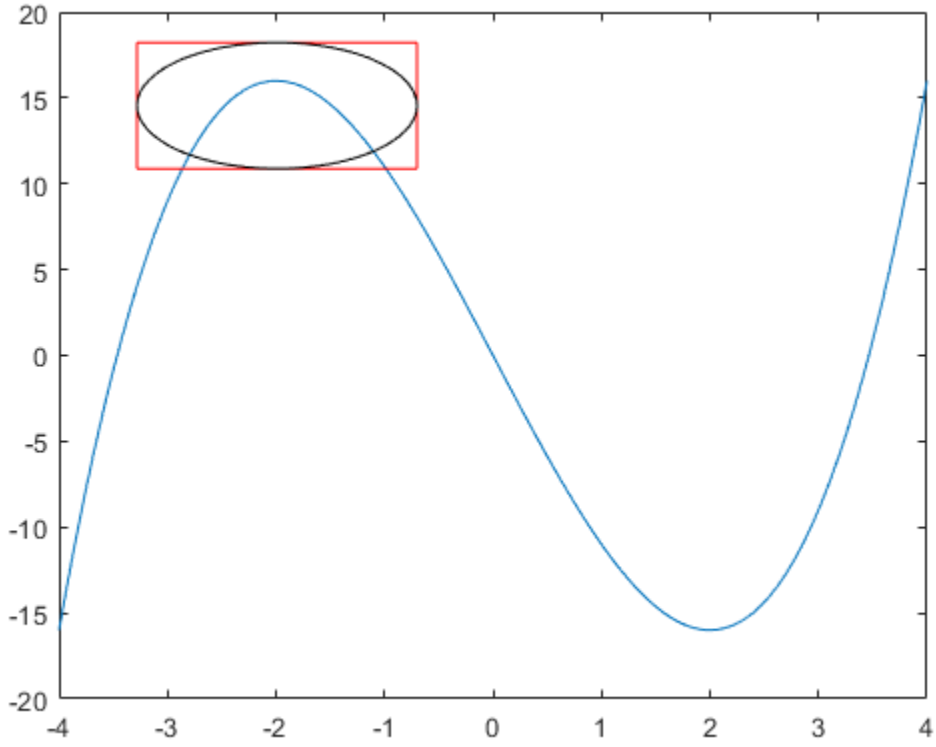
Create a simple line plot and add an ellipse annotation to the figure. Specify `dim` as the size and location of the smallest rectangle that encloses the ellipse.

```
figure
x = linspace(-4,4);
y = x.^3 - 12*x;
plot(x,y)

dim = [.2 .74 .25 .15];
annotation('ellipse',dim)
```



Draw a red rectangle using the same dimensions to show how the ellipse fills the rectangular area.

```
annotation('rectangle',dim,'Color','red')
```

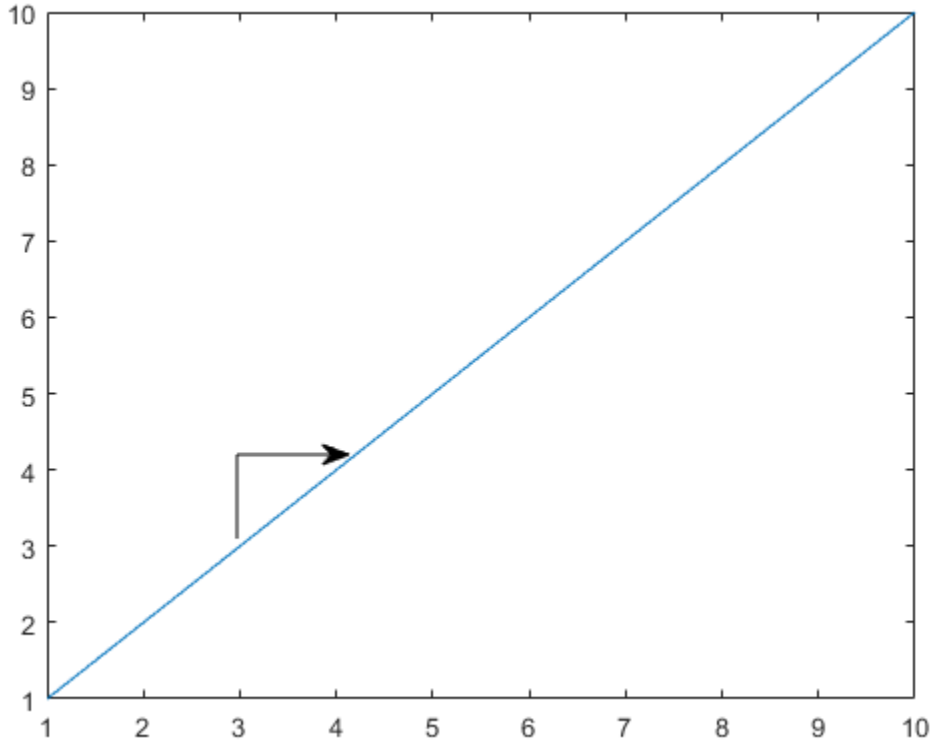


Combine Two Types of Annotations

Create a simple line plot. Then, add a bent arrow to the graph by combining a line and an arrow annotation.

```
figure
plot(1:10)

xl = [.3 .3];
yl = [.3 .4];
annotation('line',xl,yl)
```

```
xa = [.3 .4];
ya = [.4 .4];
annotation('arrow',xa,ya)
```

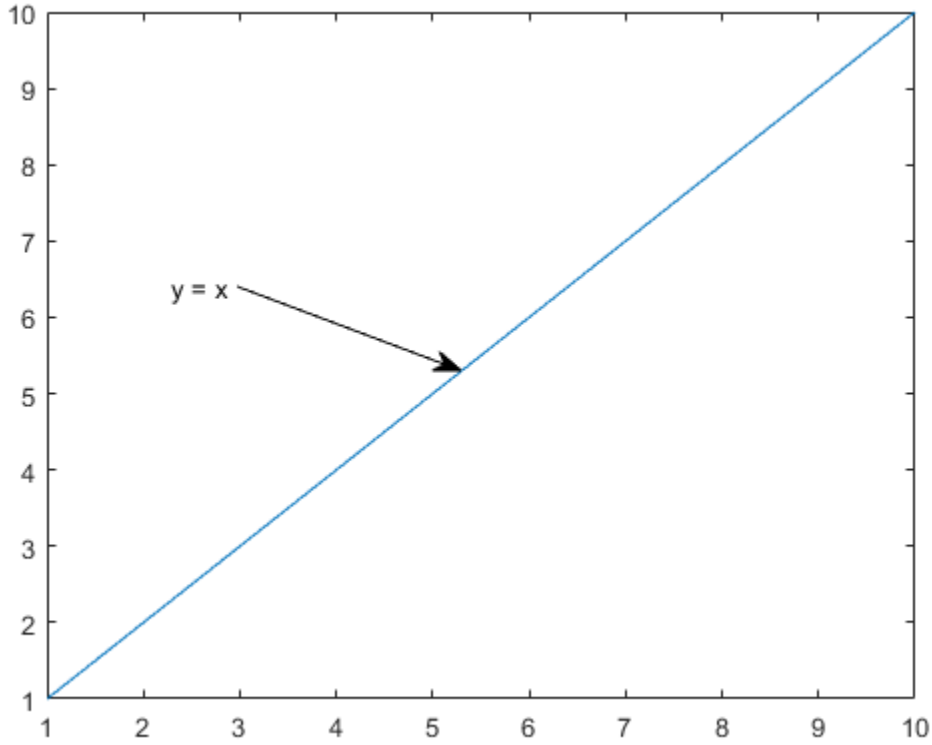


Modify Annotation After Creation

Add a text arrow to a figure and return the annotation text arrow object, a.

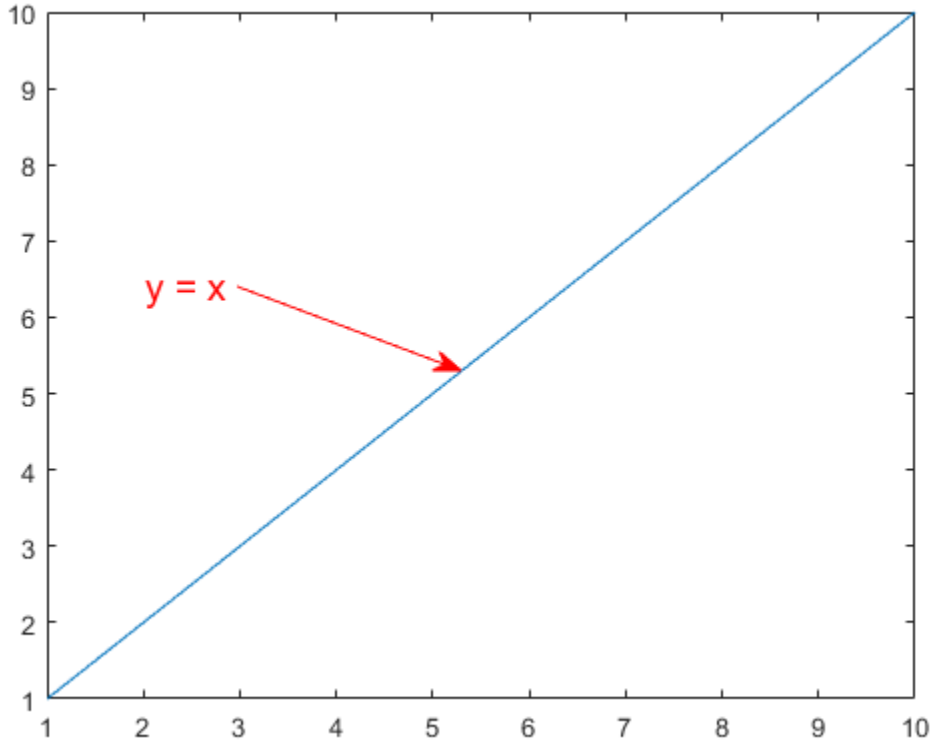
```
figure
plot(1:10)
x = [0.3,0.5];
```

```
y = [0.6,0.5];
a = annotation('textarrow',x,y,'String','y = x ');
```



Modify properties of the annotation text arrow using `a`. For example, change the color to red and the font size to 14 points.

```
a.Color = 'red';
a.FontSize = 14;
```



See the annotation property pages for a list of properties for each type of annotation.

Input Arguments

`lineType` — Type of line annotation
`'line'` | `'arrow'` | `'doublearrow'` | `'textarrow'`

Type of line annotation, specified as one of these values.

Value	Type of Object	Example
`'line'`	Annotation line	`annotation('line',[.1 .2],[.1 .2])`
`'arrow'`	Annotation arrow	`annotation('arrow',[.1 .2],[.1 .2])`
`'doublearrow'`	Annotation double arrow	`annotation('doublearrow',[.1 .2],[.1 .2])`
`'textarrow'`	Annotation text arrow. To add text to the tail end of the text arrow, use the `String` property.	`annotation('textarrow',[.1 .2],[.1 .2],'String','my text')`

shapeType — Type of shape annotation
`'rectangle'` | `'ellipse'` | `'textbox'`

Type of shape annotation, specified as one of these values.

Value	Type of Object	Example
`'rectangle'`	Annotation rectangle	`annotation('rectangle',[.2 .3 .4 .5])`
`'ellipse'`	Annotation ellipse	`annotation('ellipse',[.2 .3 .4 .5])`
`'textbox'`	Annotation text box. To specify the text, set the `String` property. To automatically adjust the dimensions of the box to fit closely around the text, set the `FitBoxToText` property to `'on'`.	`annotation('textbox',[.2 .3 .4 .5],'String','my text','FitBoxToText','on')`

x — Beginning and ending *x*-coordinates
two-element vector of the form `[x_begin x_end]`

Beginning and ending *x*-coordinates, specified as a two-element vector of the form `[x_begin x_end]`. Together the `x` and `y` input arguments determine the endpoints of the line, arrow, double arrow, or text arrow annotation. The annotation extends from the point (`x_begin`, `y_begin`) to (`x_end`, `y_end`).

By default, the units are normalized to the figure. The lower left corner of the figure maps to `(0,0)` and the upper right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `x = [.3 .5]`

`y` — Beginning and ending *y*-coordinates
two-element vector of the form `[x_begin x_end]`

Beginning and ending *y*-coordinates, specified as a two-element vector of the form `[y_begin y_end]`. Together the `x` and `y` input arguments determine the endpoints of the line, arrow, double arrow, or text arrow annotation. The annotation extends from the point (`x_begin`, `y_begin`) to (`x_end`, `y_end`).

By default, the units are normalized to the figure. The lower left corner of the figure maps to `(0,0)` and the upper right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `y = [.3 .5]`

`dim` — Size and location
four-element vector of the form `[x y w h]`

Size and location, specified as a four-element vector of the form `[x y w h]`. The first two elements specify the coordinates of the lower left corner of the text box, rectangle, or ellipse with respect to the lower left corner of the figure. The second two elements specify the width and height of the annotation, respectively.

If you are creating an ellipse, then `dim` is the size and location of the smallest rectangle that encloses the ellipse.

By default, the units are normalized to the figure. The lower left corner of the figure maps to `(0,0)` and the upper right corner maps to `(1,1)`. To change the units, use the `Units` property

Example: `dim = [.3 .4 .5 .6]`

`container` — Target for annotation
figure object | uipanel object | uitab object

Target for annotation, specified as a figure, uipanel, or uitab object. For example, to add an annotation to a specific figure, specify the figure object as the first input argument to the function.

```
f = figure;
annotation(f,'line',[.1 .2],[.2 .3])
```

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `annotation('rectangle',[.5 .5 .1 .1],'EdgeColor','r')` creates a rectangle annotation with a red outline.

Each type of annotation object supports a different set of properties. For a full list of properties and descriptions for each type, see the associated property page.

- Line
- Arrow
- DoubleEndArrow
- TextArrow
- TextBox
- Rectangle
- Ellipse

Output Arguments

an — Annotation object
scalar

Annotation object, returned as a scalar. The type of annotation object returned depends on the first input argument.

First Input Argument	Type of Object Returned	Property Page
`'line'`	Line	Annotation Line
`'arrow'`	Arrow	Arrow
`'doublearrow'`	Double arrow	DoubleEndArrow

First Input Argument	Type of Object Returned	Property Page
`'textarrow'`	Text arrow	TextArrow
`'textbox'`	Text box	TextBox
`'rectangle'`	Rectangle	Annotation Rectangle
`'ellipse'`	Ellipse	Ellipse

More About

Compatibility Considerations

Starting in R2014b, annotations cannot cross uipanel boundaries. Instead, they clip at the boundaries. Previous versions of MATLAB allow annotations to extend into (or out of) the boundaries. To display an annotation within a specific figure, uipanel, or uitab, use the `container` input argument.

See Also

Functions
`rectangle` | `text`

Properties
Arrow | DoubleEndArrow | Ellipse | Line | Rectangle | TextArrow | TextBox

Topics
"Add Text to Chart"

Introduced before R2006a

Arrow Properties

Arrow appearance and behavior

Description

`Arrow` properties control the appearance and behavior of an `Arrow` object. By changing property values, you can modify certain aspects of the arrow.

Starting in R2014b, you can use dot notation to query and set properties.

```
ar = annotation('arrow');
c = ar.Color;
ar.Color = 'red';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Color and Styling

`Color` — Arrow color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Arrow color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default RGB triplet value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'blue'`

Example: `[0 0 1]`

`LineStyle` — Style of arrow stem

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Style of arrow stem, specified as one of the line styles listed in this table.

Line Style	Description	Result
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No stem	No stem

`LineWidth` — Width of arrow stem
`0.5` (default) | positive value

Width of arrow stem, specified as a positive value in point units. One point equals 1/72 inch.

Example: `0.75`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`HeadStyle` — Style of arrowhead
`'vback2'` (default) | `'plain'` | `'ellipse'` | `'vback1'` | `'vback3'` | `'cback1'` | `'cback2'` | ...

Style of the arrowhead, specified as one of the styles in this table.

Style	Result	Style	Result
`'plain'`		`'fourstar'`	
`'ellipse'`		`'rectangle'`	
`'vback1'`		`'diamond'`	
`'vback2'` (default)		`'rose'`	

Style	Result	Style	Result
`'vback3'`		`'hypocycloid'`	
`'cback1'`		`'astroid'`	
`'cback2'`		`'deltoid'`	
`'cback3'`		`'none'`	No arrowhead

`HeadLength` — Length of arrowhead

`10` (default) | scalar numeric value

Length of the arrowhead, specified as a scalar numeric value in point units. One point equals 1/72 inch. The arrowhead extends backwards from the point `(x_end,y_end)` determined by the `X` and `Y` properties.

Example: `15`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`HeadWidth` — Width of arrowhead

`10` (default) | scalar numeric value

Width of the arrowhead, specified as a scalar numeric value in point units. One point equals 1/72 inch.

Example: `15`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Position

`X` — Beginning and ending x-coordinates

`[0.3 0.4]` (default) | two-element vector

Beginning and ending *x*-coordinates, specified as a two-element vector of the form `[x_begin x_end]`.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)`, and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `[0.2 0.5]`

Y — Beginning and ending y-coordinates
`[0.3 0.4]` (default) | two-element vector

Beginning and ending *y*-coordinates, specified as a two-element vector of the form `[y_begin y_end]`.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)`, and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `[0.2 0.5]`

`Position` — Size and location
`[0.3 0.3 0.1 0.1]` (default) | four-element vector

Size and location, specified as a four-element vector of the form `[x_begin y_begin dx dy]`. The first two elements specify the coordinates for the beginning of the arrow. The second two elements specify the slope of the arrow.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)`, and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `[0.5 0.5 0.2 0.3]`

`Units` — Position units
`'normalized'` (default) | `'inches'` | `'centimeters'` | `'characters'` | `'points'` | `'pixels'`

Position units, specified as one of the values in this table.

Units	Description
`'normalized'` (default)	Normalized with respect to the figure, uipanel, or uitab that contains the annotation. The lower-left corner of the container maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the default system font character size. • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.
`'points'`	Points. One point equals 1/72 inch.
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.

All units are measured from the lower-left corner of the figure window.

This property affects the `Position` property. If you change the units, then it is good practice to return it to the default value after completing your computation to prevent affecting other functions that assume `Units` is set to the default value.

If you specify the `Position` and `Units` properties as `Name,Value` pairs when creating the object, then the order of specification matters. If you want to define the position with particular units, then you must set the `Units` property before the `Position` property.

See Also

`annotation`

Topics

"Access Property Values"
"Graphics Object Properties"

Introduced before R2006a

DoubleEndArrow Properties

Double end arrow appearance and behavior

Description

`DoubleEndArrow` properties control the appearance and behavior of a `DoubleEndArrow` object. By changing property values, you can modify certain aspects of the double arrow.

Starting in R2014b, you can use dot notation to query and set properties.

```
an = annotation('doublearrow');
c = an.Color;
an.Color = 'red';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Color and Styling

`Color` — Arrow color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Arrow color, specified as an RGB triplet, hexadecimal color code, a color name, or a short name. The default RGB triplet value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`LineStyle` — Style of arrow stem

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Style of arrow stem, specified as one of the line styles listed in this table.

Line Style	Description	Result
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No stem	No stem

`LineWidth` — Width of arrow stem

`0.5` (default) | positive value

Width of arrow stem, specified as a positive value in point units. One point equals 1/72 inch.

Example: `0.75`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Head1Style` — Style of first arrowhead

`'vback2'` (default) | `'plain'` | `'ellipse'` | `'vback1'` | `'vback3'` | `'cback1'` | ...

Style of the first arrowhead, specified as one of the head style options in this table. The first arrowhead is located at the point `(x_begin,y_begin)` determined by the `X` and `Y` properties.

Style	Result	Style	Result
`'plain'`		`'fourstar'`	
`'ellipse'`		`'rectangle'`	
`'vback1'`		`'diamond'`	

Style	Result	Style	Result
`'vback2'` (default)		`'rose'`	
`'vback3'`		`'hypocycloid'`	
`'cback1'`		`'astroid'`	
`'cback2'`		`'deltoid'`	
`'cback3'`		`'none'`	No arrowhead

`Head2Style` — Style of second arrowhead

`'vback2'` (default) | `'plain'` | `'ellipse'` | `'vback1'` | `'vback3'` | `'cback1'` | ...

Style of the second arrowhead, specified as one of the head style options in this table. The second arrowhead is located at the point `(x_end,y_end)` determined by the `X` and `Y` properties.

Style	Result	Style	Result
`'plain'`		`'fourstar'`	
`'ellipse'`		`'rectangle'`	
`'vback1'`		`'diamond'`	
`'vback2'` (default)		`'rose'`	
`'vback3'`		`'hypocycloid'`	
`'cback1'`		`'astroid'`	
`'cback2'`		`'deltoid'`	
`'cback3'`		`'none'`	No arrowhead

`Head1Length` — Length of first arrowhead

`10` (default) | scalar numeric value

Length of the first arrowhead, specified as a scalar numeric value in point units. One point equals 1/72 inch. The first arrowhead extends backwards from the point `(x_begin,y_begin)` determined by the `X` and `Y` properties.

Example: `15`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Head2Length` — Length of second arrowhead

`10` (default) | scalar numeric value

Length of the second arrowhead, specified as a scalar numeric value in point units. One point equals 1/72 inch. The second arrowhead extends backwards from the point `(x_end,y_end)` determined by the `X` and `Y` properties.

Example: `15`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Head1Width` — Width of first arrowhead

`10` (default) | scalar numeric value

Width of the first arrowhead, specified as a scalar numeric value in point units. One point equals 1/72 inch. The first arrowhead is located at the point `(x_begin,y_begin)` determined by the `X` and `Y` properties.

Example: `15`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Head2Width` — Width of second arrowhead

`10` (default) | scalar numeric value

Width of the second arrowhead, specified as a scalar numeric value in point units. One point equals 1/72 inch. The second arrowhead is located at the point `(x_end,y_end)` determined by the `X` and `Y` properties.

Example: `15`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Position

X — Beginning and ending x-coordinates
`[0.3 0.4]` (default) | two-element vector

Beginning and ending *x*-coordinates, specified as a two-element vector of the form `[x_begin x_end]`.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)` and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `[0.2 0.3]`

Y — Beginning and ending y-coordinates
`[0.3 0.4]` (default) | two-element vector

Beginning and ending *y*-coordinates, specified as a two-element vector of the form `[y_begin y_end]`.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)` and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `[0.2 0.3]`

`Position` — Size and location
`[0.3 0.3 0.1 0.1]` (default) | four-element vector

Size and location, specified as a four-element vector of the form `[x_begin y_begin dx dy]`. The first two elements specify the coordinates for the beginning of the arrow. The second two elements specify the slope of the arrow.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)` and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `[0.5 0.5 0.2 0.3]`

Units — Position units
`'normalized'` (default) | `'inches'` | `'centimeters'` | `'characters'` | `'points'` | `'pixels'`

Position units, specified as one of the values in this table.

Units	Description
`'normalized'` (default)	Normalized with respect to the figure, uipanel, or uitab that contains the annotation. The lower-left corner of the container maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the default system font character size. • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.
`'points'`	Points. One point equals 1/72 inch.
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.

All units are measured from the lower-left corner of the figure window.

This property affects the `Position` property. If you change the units, then it is good practice to return it to the default value after completing your computation to prevent affecting other functions that assume `Units` is set to the default value.

If you specify the `Position` and `Units` properties as `Name,Value` pairs when creating the object, then the order of specification matters. If you want to define the position with particular units, then you must set the `Units` property before the `Position` property.

See Also

`annotation`

Topics

"Access Property Values"
"Graphics Object Properties"

Introduced before R2006a

Ellipse Properties

Ellipse appearance and behavior

Description

`Ellipse` properties control the appearance and behavior of an `Ellipse` object. By changing property values, you can modify certain aspects of the ellipse.

Starting in R2014b, you can use dot notation to query and set properties.

```
an = annotation('ellipse');
c = an.Color;
an.Color = 'red';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Color and Styling

`Color` — Outline color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Outline color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`FaceColor` — Fill color

`'none'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Fill color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width

`0.5` (default) | positive value

Line width of ellipse outline, specified as a positive value in point units. One point equals 1/72 inch.

Example: `0.75`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Position

Position — Size and location

`[0.3 0.3 0.1 0.1]` (default) | four-element vector

Size and location, specified as a four-element vector of the form `[x y length height]`. The first two elements specify the coordinates of the lower-left corner of the smallest rectangle that enclose the ellipse. The second two elements specify length and height of the rectangle.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)` and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `[0.2 0.3 0.4 0.5]`

Units — Position units

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'characters'` | `'points'` | `'pixels'`

Position units, specified as one of the values in this table.

Units	Description
`'normalized'` (default)	Normalized with respect to the figure, uipanel, or uitab that contains the annotation. The lower-left corner of the container maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the default system font character size. • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.
`'points'`	Points. One point equals 1/72 inch.

Units	Description
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.

All units are measured from the lower-left corner of the figure window.

This property affects the `Position` property. If you change the units, then it is good practice to return it to the default value after completing your computation to prevent affecting other functions that assume `Units` is set to the default value.

If you specify the `Position` and `Units` properties as `Name,Value` pairs when creating the object, then the order of specification matters. If you want to define the position with particular units, then you must set the `Units` property before the `Position` property.

See Also

`annotation`

Topics

"Access Property Values"
"Graphics Object Properties"

Introduced before R2006a

Line Properties

Annotation line appearance and behavior

Description

`Line` properties control the appearance and behavior of an `Line` object. By changing property values, you can modify certain aspects of the line.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = annotation('line');
c = h.Color;
h.Color = 'red';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Color and Styling

`Color` — Line color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in point units. One point equals 1/72 inch.

Example: `0.75`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Position

X — Beginning and ending x-coordinates
`[0.3 0.4]` (default) | two-element vector

Beginning and ending *x*-coordinates, specified as a two-element vector of the form `[x_begin x_end]`.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)` and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `[0.2 0.3]`

Y — Beginning and ending y-coordinates
`[0.3 0.4]` (default) | two-element vector

Beginning and ending *y*-coordinates, specified as a two-element vector of the form `[y_begin y_end]`.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)` and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `[0.2 0.3]`

`Position` — Size and location
`[0.3 0.3 0.1 0.1]` (default) | four-element vector

Size and location, specified as a four-element vector of the form `[x_begin y_begin dx dy]`. The first two elements specify the coordinates of the starting point of the line. The second two elements specify the slope of the line.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)` and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property. To change the units, use the `Units` property.

Example: `[0.2 0.2 0.3 0.3]`

`Units` — Position units
`'normalized'` (default) | `'inches'` | `'centimeters'` | `'characters'` | `'points'` | `'pixels'`

Position units, specified as one of the values in this table.

Units	Description
`'normalized'` (default)	Normalized with respect to the figure, uipanel, or uitab that contains the annotation. The lower-left corner of the container maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the default system font character size. • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.

Units	Description
`'points'`	Points. One point equals 1/72 inch.
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.

All units are measured from the lower-left corner of the figure window.

This property affects the `Position` property. If you change the units, then it is good practice to return it to the default value after completing your computation to prevent affecting other functions that assume `Units` is set to the default value.

If you specify the `Position` and `Units` properties as `Name,Value` pairs when creating the object, then the order of specification matters. If you want to define the position with particular units, then you must set the `Units` property before the `Position` property.

See Also

`annotation`

Topics

"Access Property Values"
"Graphics Object Properties"

Introduced before R2006a

Rectangle Properties

Annotation rectangle appearance and behavior

Description

`Rectangle` properties control the appearance and behavior of a `Rectangle` object. By changing property values, you can modify certain aspects of the rectangle.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = annotation('rectangle');
c = h.Color;
h.Color = 'red';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Color and Styling

`Color` — Outline color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Outline color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`FaceColor` — Fill color

`'none'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Fill color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`FaceAlpha` — Fill transparency
1 (default) | value in range `[0,1]`

Fill transparency, specified as a scalar value in the range `[0,1]`. A value of `1` is opaque and `0` is completely transparent. Values between `0` and `1` are semitransparent.

Example: `0.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

LineWidth — Line width

`0.5` (default) | positive value

Line width of rectangle outline, specified as a positive value in point units. One point equals 1/72 inch.

Example: `0.75`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Position

Position — Size and location

`[0.3 0.3 0.1 0.1]` (default) | four-element vector

Size and location, specified as a four-element vector of the form `[x y length height]`. The first two elements specify the coordinates of the lower left corner of the rectangle. The second two elements specify the length and height of the rectangle.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)` and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `[0.2 0.3 0.4 0.5]`

Units — Position units

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'characters'` | `'points'` | `'pixels'`

Position units, specified as one of the values in this table.

Units	Description
`'normalized'` (default)	Normalized with respect to the figure, uipanel, or uitab that contains the annotation. The lower-left corner of the container maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.

Units	Description
`'characters'`	Based on the default system font character size. • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.
`'points'`	Points. One point equals 1/72 inch.
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.

All units are measured from the lower-left corner of the figure window.

This property affects the `Position` property. If you change the units, then it is good practice to return it to the default value after completing your computation to prevent affecting other functions that assume `Units` is set to the default value.

If you specify the `Position` and `Units` properties as `Name,Value` pairs when creating the object, then the order of specification matters. If you want to define the position with particular units, then you must set the `Units` property before the `Position` property.

See Also

`annotation`

Topics

“Access Property Values”

"Graphics Object Properties"

Introduced before R2006a

TextArrow Properties

Text arrow appearance and behavior

Description

`TextArrow` properties control the appearance and behavior of a `TextArrow` object. By changing property values, you can modify certain aspects of the text arrow.

Starting in R2014b, you can use dot notation to query and set properties.

```
ta = annotation('textarrow');
s = ta.FontSize;
ta.FontSize = 12;
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Text

`String` — Text to display

`''` (default) | character vector | cell array of character vectors | string array | categorical array | numeric value

Text to display, specified as a character vector, cell array of character vectors, string array, categorical array, or numeric value. Specify a value according to how you want the text to appear. This table lists the most common situations.

Appearance of Text	Description of Value	Example

One line of text	Character vector or 1-by-1 string array.	`str = 'My Text'`
		or
		`str = "My Text"`

Appearance of Text	Description of Value	Example

Multiple lines of text	Cell array of character vectors or a string array.	`str = {'First line','Second line'};`
		or
		`str = ["First line", "Second line"];`

Text that includes a numeric variable	Array that includes the number converted to a `char` array. Use `num2str` to convert the value.	`x = 42;` `str = ['The value is ',num2str(x)];`
Text that includes special characters such as Greek letters or mathematical symbols	Array that includes TeX markup. For a list of supported markup, see the `Interpreter` property.	`str = 'x ranges from 0 to 2\pi'`

Note

- The words `default`, `factory`, and `remove` are reserved words that do not appear in text when quoted as normal characters. To display any of these words individually, precede them with a backslash, such as `'\default'` or `'\remove'`.
- If you specify this property as a categorical array, MATLAB uses the values in the array, not the categories.
- If you specify text that contains only a numeric value, the value is converted using `sprintf('%g',value)`. For example, `12345678` displays as `1.23457e+07`.

`TextRotation` — Text rotation angle in degrees

`0` (default) | scalar numeric value

Text rotation angle in degrees, specified as a scalar numeric value. Set this property to a positive value to rotate the text counterclockwise. Angles are absolute and not relative to previous rotations. A rotation of 0 degrees is always horizontal.

Example: `90`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`TextColor` — Text color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Text color, specified as a three-element RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[0 0 0]` corresponds to black.

Note Setting the `Color` property changes the `TextColor` property to the same value, unless you explicitly set the `TextColor` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: 'blue'

Example: [0 0 1]

Example: '#0000FF'

TextEdgeColor — Color of text box outline

'none' (default) | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...

Color of text box outline, specified as a three-element RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`TextBackgroundColor` — Color of text box background

`'none'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of text box background, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`TextLineWidth` — Width of text box outline
`0.5` (default) | scalar numeric value

Width of text box outline, specified as a scalar numeric value in point units. One point equals `1/72` inch.

Example: `1.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`TextMargin` — Space around text within text box
2 (default) | scalar numeric value

Space around the text within the text box, specified as a scalar numeric value in pixel units.

Example: `10`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`Interpreter` — Interpretation of text characters
`'tex'` (default) | `'latex'` | `'none'`

Interpretation of text characters, specified as one of these values:

- `'tex'` — Interpret characters using a subset of TeX markup.
- `'latex'` — Interpret characters using LaTeX markup.

- `'none'` — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text.

Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they modify only the next character or the characters within the curly braces. When you set the `Interpreter` property to `'tex'`, the supported modifiers are as follows.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{`*`specifier`*`}`	Font name — Replace *`specifier`* with the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{`*`specifier`*`}`	Font size —Replace *`specifier`* with a numeric scalar value in point units.	`'\fontsize{15} text'`
`\color{`*`specifier`*`}`	Font color — Replace *`specifier`* with one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`

Modifier	Description	Example
`\color[rgb] {specifier}`	Custom font color — Replace *specifier* with a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	β	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ϵ	`\Delta`	Δ	`\spadesuit`	♠
`\zeta`	ζ	`\Theta`	Θ	`\leftrightarrow`	↔
`\eta`	η	`\Lambda`	Λ	`\leftarrow`	←
`\theta`	θ	`\Xi`	Ξ	`\Leftarrow`	⇐
`\vartheta`	ϑ	`\Pi`	Π	`\uparrow`	↑
`\iota`	ι	`\Sigma`	Σ	`\rightarrow`	→
`\kappa`	κ	`\Upsilon`	ϒ	`\Rightarrow`	⇒
`\lambda`	λ	`\Phi`	Φ	`\downarrow`	↓
`\mu`	µ	`\Psi`	Ψ	`\circ`	º
`\nu`	ν	`\Omega`	Ω	`\pm`	±
`\xi`	ξ	`\forall`	∀	`\geq`	≥
`\pi`	π	`\exists`	∃	`\propto`	∝

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. Use dollar symbols around the text, for example, use `'$\int_1^{20} x^2 dx$'` for inline mode or `'$$ \int_1^{20} x^2 dx$$'` for display mode.

The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup.

The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters. For multiline text, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project website at https://www.latex-project.org/.

Font

`FontName` — Font name

supported font name | `'FixedWidth'`

Font name, specified as a supported font name or `'FixedWidth'`. To display and print text properly, you must choose a font that your system supports. The default font depends on your operating system and locale.

To use a fixed-width font that looks good in any locale, use `'FixedWidth'`. The fixed-width font relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`FontSize` — Font size

scalar value greater than 0

Font size, specified as a scalar value greater than 0 in point units. The default font size depends on the specific operating system and locale. One point equals 1/72 inch. To change the font units, use the `FontUnits` property.

Example: 12

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`FontWeight` — Character thickness

`'normal'` (default) | `'bold'`

Character thickness, specified as `'normal'` or `'bold'`.

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.

`FontAngle` — Character slant

`'normal'` (default) | `'italic'`

Character slant, specified as `'normal'` or `'italic'`.

Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.

FontUnits — Font size units

`'points'` (default) | `'inches'` | `'centimeters'` | `'normalized'` | `'pixels'`

Font size units, specified as one of the values in this table.

Units	Description
`'points'`	Points. One point equals 1/72 inch.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'normalized'`	Interpret font size as a fraction of the parent container height, typically a figure. If you resize the container, the font size modifies accordingly. For example, if the `FontSize` is `0.1` in normalized units, then the text is 1/10 of the container height.
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.

If you set both the font size and the font units in one function call, you must set the `FontUnits` property first so that the axes correctly interprets the specified font size.

Arrow

Color — Arrow color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Arrow color, specified as a three-element RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[0 0 0]` corresponds to black.

Note Setting this property also changes the text color if you have not explicitly set the text color using the `TextColor` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Width of arrow stem

`0.5` (default) | scalar numeric value

Width of arrow stem, specified as a scalar numeric value greater than zero in point units. One point equals `1/72` inch. The default value is `0.5` points.

Example: `0.75`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`HeadStyle` — Style of arrowhead
`'vback2'` (default) | `'plain'` | `'ellipse'` | `'vback1'` | `'vback3'` | `'cback1'` | ...

Style of the arrowhead, specified as one of the head style options in this table.

Style	Result	Style	Result
`'plain'`		`'fourstar'`	
`'ellipse'`		`'rectangle'`	
`'vback1'`		`'diamond'`	
`'vback2'` (default)		`'rose'`	
`'vback3'`		`'hypocycloid'`	
`'cback1'`		`'astroid'`	
`'cback2'`		`'deltoid'`	
`'cback3'`		`'none'`	No arrowhead

`HeadLength` — Length of arrowhead
`10` (default) | scalar numeric value

Length of the arrowhead, specified as a scalar numeric value in point units. One point equals `1/72` inch.

Example: `15`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`HeadWidth` — Width of arrowhead
`10` (default) | scalar numeric value

Width of the arrowhead, specified as a scalar numeric value in point units. One point equals `1/72` inch.

Example: `15`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Position

X — Beginning and ending x-coordinates
`[0.3 0.4]` (default) | two-element vector

Beginning and ending *x*-coordinates for the arrow, specified as a two-element vector of the form `[x_begin x_end]`.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)`, and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `[0.2 0.3]`

Y — Beginning and ending y-coordinates
`[0.3 0.4]` (default) | two-element vector

Beginning and ending *y*-coordinates for the arrow, specified as a two-element vector of the form `[y_begin y_end]`.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)`, and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `[0.2 0.3]`

`Position` — Size and location
`[0.3 0.3 0.1 0.1]` (default) | four-element vector

Size and location, specified as a four-element vector of the form `[x_begin y_begin length height]`. The first two elements specify the coordinates of the beginning of the arrow. The second two elements specify the length and height of the arrow. The text box extends from the beginning of the arrow.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)`, and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Example: `[0.2 0.2 0.3 0.1]`

Units — Position units
`'normalized'` (default) | `'inches'` | `'centimeters'` | `'characters'` | `'points'` | `'pixels'`

Position units, specified as one of the values in this table.

Units	Description
`'normalized'` (default)	Normalized with respect to the figure, uipanel, or uitab that contains the annotation. The lower-left corner of the container maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the default system font character size. • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.
`'points'`	Points. One point equals 1/72 inch.
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.

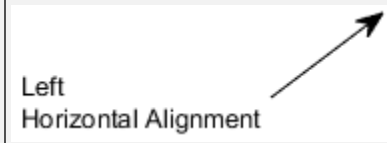
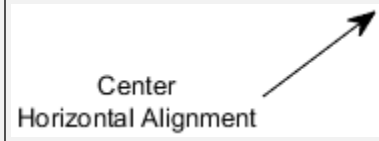
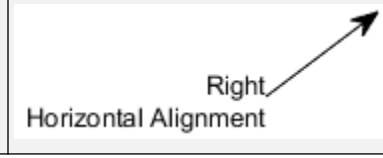
All units are measured from the lower-left corner of the figure window.

This property affects the `Position` property. If you change the units, then it is good practice to return it to the default value after completing your computation to prevent affecting other functions that assume `Units` is set to the default value.

If you specify the `Position` and `Units` properties as `Name,Value` pairs when creating the object, then the order of specification matters. If you want to define the position with particular units, then you must set the `Units` property before the `Position` property.

`HorizontalAlignment` — Horizontal alignment of text
`'left'` (default) | `'center'` | `'right'`

Horizontal alignment of the text, specified as one of the values in this table. This property is useful when aligning multiple lines of text.

Value	Result
`'left'`	
`'center'`	
`'right'`	

`VerticalAlignment` — Vertical alignment of text with respect to arrow
`'top'` (default) | `'cap'` | `'middle'` | `'baseline'` | `'bottom'`

Vertical alignment of the text with respect to the end of the arrow, specified as `'top'`, `'cap'`, `'middle'`, `'baseline'`, or `'bottom'`.

See Also

`annotation`

Topics

"Access Property Values"
"Graphics Object Properties"

Introduced before R2006a

TextBox Properties

Text box appearance and behavior

Description

`TextBox` properties control the appearance and behavior of a `TextBox` object. By changing property values, you can modify certain aspects of the text box.

Starting in R2014b, you can use dot notation to query and set properties.

```
t = annotation('textbox');
sz = t.FontSize;
t.FontSize = 12;
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Text

`String` — Text to display

`''` (default) | character vector | cell array of character vectors | string array | categorical array | numeric value

Text to display, specified as a character vector, cell array of character vectors, string array, categorical array, or numeric value. Specify a value according to how you want the text to appear. This table lists the most common situations.

Appearance of Text	Description of Value	Example

One line of text	Character vector or 1-by-1 string array.	`str = 'My Text'`
		or
		`str = "My Text"`

Appearance of Text	Description of Value	Example

Multiple lines of text	Cell array of character vectors or a string array.	`str = {'First line','Second line'};`
		or
		`str = ["First line", "Second line"];`

Text that includes a numeric variable	Array that includes the number converted to a `char` array. Use `num2str` to convert the value.	`x = 42;` `str = ['The value is ',num2str(x)];`
Text that includes special characters such as Greek letters or mathematical symbols	Array that includes TeX markup. For a list of supported markup, see the `Interpreter` property.	`str = 'x ranges from 0 to 2\pi'`

Note

- The words `default`, `factory`, and `remove` are reserved words that do not appear in text when quoted as normal characters. To display any of these words individually, precede them with a backslash, such as `'\default'` or `'\remove'`.
- If you specify this property as a categorical array, MATLAB uses the values in the array, not the categories.
- If you specify text that contains only a numeric value, the value is converted using `sprintf('%g',value)`. For example, `12345678` displays as `1.23457e+07`.

`Color` — Text color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Text color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: 'blue'

Example: [0 0 1]

Example: '#0000FF'

Interpreter — Interpretation of text characters

'tex' (default) | 'latex' | 'none'

Interpretation of text characters, specified as one of these values:

- 'tex' — Interpret characters using a subset of TeX markup.
- 'latex' — Interpret characters using LaTeX markup.
- 'none' — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text.

Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they modify only the next character or the characters within the curly braces. When you set the Interpreter property to 'tex', the supported modifiers are as follows.

Modifier	Description	Example
^{ }	Superscript	'text^{superscript}'
_{ }	Subscript	'text_{subscript}'
\bf	Bold font	'\bf text'
\it	Italic font	'\it text'
\sl	Oblique font (usually the same as italic font)	'\sl text'
\rm	Normal font	'\rm text'

Modifier	Description	Example
`\fontname{`*`specifier`*`}`	Font name — Replace *`specifier`* with the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{`*`specifier`*`}`	Font size —Replace *`specifier`* with a numeric scalar value in point units.	`'\fontsize{15} text'`
`\color{`*`specifier`*`}`	Font color — Replace *`specifier`* with one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Replace *`specifier`* with a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	β	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ϵ	`\Delta`	Δ	`\spadesuit`	♠
`\zeta`	ζ	`\Theta`	Θ	`\leftrightarrow`	↔

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	µ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. Use dollar symbols around the text, for example, use `'$\int_1^{20} x^2 dx$'` for inline mode or `'$$ \int_1^{20} x^2 dx$$'` for display mode.

The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup.

The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters. For multiline text, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project website at https://www.latex-project.org/.

Font

`FontName` — Font name

supported font name | `'FixedWidth'`

Font name, specified as a supported font name or `'FixedWidth'`. To display and print text properly, you must choose a font that your system supports. The default font depends on your operating system and locale.

To use a fixed-width font that looks good in any locale, use `'FixedWidth'`. The fixed-width font relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`FontSize` — Font size

scalar value greater than 0

Font size, specified as a scalar value greater than 0 in point units. The default font size depends on the specific operating system and locale. One point equals 1/72 inch. To change the font units, use the `FontUnits` property.

Example: 12

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`FontWeight` — Character thickness

`'normal'` (default) | `'bold'`

Character thickness, specified as `'normal'` or `'bold'`.

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.

`FontAngle` — Character slant
`'normal'` (default) | `'italic'`

Character slant, specified as `'normal'` or `'italic'`.

Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.

`FontUnits` — Font size units
`'points'` (default) | `'inches'` | `'centimeters'` | `'normalized'` | `'pixels'`

Font size units, specified as one of the values in this table.

Units	Description
`'points'`	Points. One point equals 1/72 inch.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'normalized'`	Interpret font size as a fraction of the parent container height, typically a figure. If you resize the container, the font size modifies accordingly. For example, if the `FontSize` is `0.1` in normalized units, then the text is 1/10 of the container height.

Units	Description
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.

If you set both the font size and the font units in one function call, you must set the `FontUnits` property first so that the axes correctly interprets the specified font size.

Text Box

`FitBoxToText` — Option to fit box width and height to text

`'on'` (default) | `'off'`

Option to fit the box width and height to the text, specified as one of these values:

- `'on'` — Resize the text box to fit the text.
- `'off'` — Wrap the text to fit the width of the text box. Wrapping can cause some of the text to extend below the text box.

If you resize a text box when in plot edit mode, or if you change the `Position` property, then the `FitBoxToText` property changes to `'off'`.

`EdgeColor` — Color of box outline

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of box outline, specified as a three-element RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: 'blue'

Example: [0 0 1]

Example: '#0000FF'

BackgroundColor — Color of text box background
'none' (default) | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...

Color of text box background, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`FaceAlpha` — Transparency of background color
1 (default) | scalar value between 0 and 1 inclusive

Transparency of the background color, specified as a scalar value between 0 and 1. If the value is 1, then the color is opaque. To add transparency, set the property to a value closer to 0, where 0 is completely transparent.

`LineStyle` — Line style of box outline
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style of box outline, specified as one of the options listed in this table.

Line Style	Description
`'-'`	Solid line

Line Style	Description
`'--'`	Dashed line
`':'`	Dotted line
`'-.'`	Dash-dotted line
`'none'`	Box outline is invisible

`LineWidth` — Width of box outline

`0.5` (default) | scalar numeric value

Width of box outline, specified as a scalar numeric value in point units. One point equals 1/72 inch.

Example: `1.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Margin` — Space around text within the text box

5 (default) | scalar numeric value

The space around the text within the text box, specified as a scalar numeric value in pixel units.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Position

`Position` — Size and location

`[0.3 0.3 0.1 0.1]` (default) | four-element vector

Size and location, specified as a four-element vector of the form `[x_begin y_begin length height]`. The first two elements specify the coordinates for the lower-left corner of the text box. The second two elements specify the length and height of the text box.

By default, the units are normalized to the figure. The lower-left corner of the figure maps to `(0,0)`, and the upper-right corner maps to `(1,1)`. To change the units, use the `Units` property.

Note If the `FitBoxToText` property is set to `'on'` and you change the `String` property, then the `Position` property might not reflect the latest changes until the next time the screen refreshes. To ensure that the position value reflects the latest changes, call `drawnow` before querying the position when working in a script or function.

Example: `[0.2 0.3 0.4 0.5]`

`Units` — Position units

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'characters'` | `'points'` | `'pixels'`

Position units, specified as one of the values in this table.

Units	Description
`'normalized'` (default)	Normalized with respect to the figure, uipanel, or uitab that contains the annotation. The lower-left corner of the container maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the default system font character size. • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.
`'points'`	Points. One point equals 1/72 inch.

Units	Description
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.

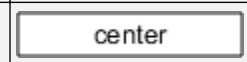
All units are measured from the lower-left corner of the figure window.

This property affects the `Position` property. If you change the units, then it is good practice to return it to the default value after completing your computation to prevent affecting other functions that assume `Units` is set to the default value.

If you specify the `Position` and `Units` properties as `Name,Value` pairs when creating the object, then the order of specification matters. If you want to define the position with particular units, then you must set the `Units` property before the `Position` property.

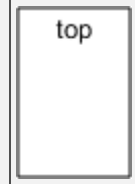
`HorizontalAlignment` — Horizontal alignment of text within text box
`'left'` (default) | `'center'` | `'right'`

Horizontal alignment of the text within the text box, specified as one of the values in this table.

Value	Result
`'left'`	
`'center'`	
`'right'`	

`VerticalAlignment` — Vertical alignment of text within text box
`'top'` (default) | `'middle'` | `'bottom'`

Vertical alignment of the text within the text box, specified as one of the values in this table.

Value	Result
`'top'`	
`'middle'`	
`'bottom'`	

Note The `'cap'` and `'baseline'` values are not recommended. Use the `'top'` and `'bottom'` values, respectively, instead.

See Also

`annotation`

Topics

“Access Property Values”
“Graphics Object Properties”

Introduced before R2006a

ans

Most recent answer

Syntax

```
ans
```

Description

When you run MATLAB code that returns an output without specifying an output argument, MATLAB creates the `ans` variable and stores the output there. Changing or using the value of `ans` in a script or function is not recommended, as the value can change frequently.

`ans` is specific to the current workspace. The base workspace and each function workspace can have its own instance of `ans`. For more information, see “Base and Function Workspaces”.

Examples

Result of a Simple Calculation

Perform a simple calculation in the Command Window without assigning the result to a variable. MATLAB stores the result in the `ans` variable.

```
2 + 2
```

```
ans = 4
```

Perform a simple calculation in the Command Window and assign the result to the variable `result`.

```
result = 4 + 4
```

```
result = 8
```

Display the values of `result` and then `ans`. MATLAB displays the value of `result` without returning an output. Therefore, the value of `ans` remains unchanged.

```
result
```

```
result = 8
```

```
ans
```

```
ans = 4
```

Call a Function That Returns Output

Suppose you have a function `testFunc`, which returns an output, without specifying an output variable.

```
function a = testFunc
a = 75;
end
```

Call `testFunc`. MATLAB stores the returned result in `ans`.

```
testFunc
```

```
ans = 75
```

See Also

`display`

Topics

"Enter Statements in Command Window"

Introduced before R2006a

any

Determine if any array elements are nonzero

Syntax

```
B = any(A)
B = any(A,'all')
B = any(A,dim)
B = any(A,vecdim)
```

Description

`B = any(A)` tests along the first array dimension of `A` whose size does not equal 1, and determines if any element is a nonzero number or logical `1` (`true`). In practice, `any` is a natural extension of the logical OR operator.

- If `A` is a vector, then `B = any(A)` returns logical `1` (`true`) if any of the elements of `A` is a nonzero number or is logical `1`, and returns logical `0` (`false`) if all the elements are zero.
- If `A` is a nonempty, nonvector matrix, then `B = any(A)` treats the columns of `A` as vectors, returning a row vector of logical `1`s and `0`s.
- If `A` is an empty 0-by-0 matrix, `any(A)` returns logical `0` (`false`).
- If `A` is a multidimensional array, `any(A)` acts along the first array dimension whose size does not equal 1 and returns an array of logical values. The size of this dimension becomes `1`, while the sizes of all other dimensions remain the same.

`B = any(A,'all')` tests over all elements of `A`. This syntax is valid for MATLAB versions R2018b and later.

`B = any(A,dim)` tests elements along dimension `dim`. The dim input is a positive integer scalar.

`B = any(A,vecdim)` tests elements based on the dimensions specified in the vector `vecdim`. For example, if `A` is a matrix, then `any(A,[1 2])` tests over all elements in `A`,

since every element of a matrix is contained in the array slice defined by dimensions 1 and 2.

Examples

Test Matrix Columns

Create a 3-by-3 matrix.

```
A = [0 0 3;0 0 3;0 0 3]
```

```
A = 3×3

     0     0     3
     0     0     3
     0     0     3
```

Test each column for nonzero elements.

```
B = any(A)
```

```
B = 1x3 logical array

   0   0   1
```

Reduce a Logical Vector to a Single Condition

Create a vector of decimal values and test which values are less than 0.5.

```
A = [0.53 0.67 0.01 0.38 0.07 0.42 0.69];
B = (A < 0.5)
```

```
B = 1x7 logical array

   0   0   1   1   1   1   0
```

The output is a vector of logical values. The `any` function reduces such a vector of logical values to a single condition. In this case, `B = any(A < 0.5)` yields logical `1`.

This makes `any` particularly useful in `if` statements.

```
if any(A < 0.5)

%do something

else

%do something else

end
```

The code is executed depending on a single condition, rather than a vector of possibly conflicting conditions.

Test Arrays of Any Dimension

Create a 3-by-7-by-5 multidimensional array and test to see if any of its elements are greater than 3.

```
A = rand(3,7,5) * 5;
B = any(A(:) > 3)
```

```
B = logical
   1
```

You can also test the array for elements that are less than zero.

```
B = any(A(:) < 0)
```

```
B = logical
   0
```

The syntax `A(:)` turns the elements of `A` into a single column vector, so you can use this type of statement on an array of any size.

Test Matrix Rows

Create a 3-by-3 matrix.

```
A = [0 0 3;0 0 3;0 0 3]
```

```
A = 3×3

     0     0     3
     0     0     3
     0     0     3
```

Test the rows of A for nonzero elements by specifying `dim = 2`.

```
B = any(A,2)
```

```
B = 3x1 logical array

   1
   1
   1
```

Nonzero Elements in Array Page

Create a 3-D array and determine if there are nonzero elements in each page of data (rows and columns).

```
A(:,:,1) = [2 0; 0 0];
A(:,:,2) = [0 0; 0 0];
A(:,:,3) = [0 0; 0 1];
B = any(A,[1 2])
```

```
B = 1x1x3 logical array
B(:,:,1) =

   1
```

```
B(:,:,2) =

   0


B(:,:,3) =

   1
```

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. The `any` function ignores elements of `A` that are `NaN` (Not a Number).

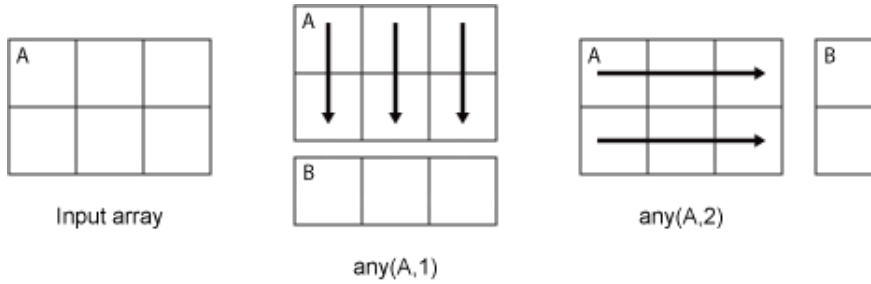
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a two-dimensional input array, `A`:

- `any(A,1)` works on successive elements in the columns of `A` and returns a row vector of logical values.
- `any(A,2)` works on successive elements in the rows of `A` and returns a column vector of logical values.

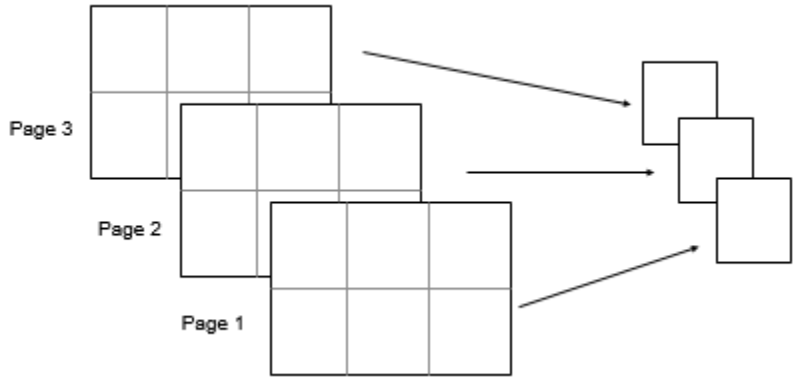


Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`vecdim` — Vector of dimensions
vector of positive integers

Vector of dimensions, specified as a vector of positive integers. Each element represents a dimension of the input array. The lengths of the output in the specified operating dimensions are 1, while the others remain the same.

Consider a 2-by-3-by-3 input array, `A`. Then `any(A,[1 2])` returns a 1-by-1-by-3 array whose elements indicate nonzero values for each page of `A`.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

B — Logical array
scalar | vector | matrix | multidimensional array

Logical array, returned as a scalar, vector, matrix, or multidimensional array. The dimension of `A` acted on by `any` has size `1` in `B`.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`all` | `colon` | `nnz` | `or` | `prod` | `sum`

Topics

“Reduce Logical Arrays to Single Value”

Introduced before R2006a

matlab.appdesigner Settings

App Designer settings, including canvas editor and component browser settings

You can customize App Designer using the `matlab.appdesigner` settings. Access `matlab.appdesigner` settings using the root `SettingsGroup` object returned by the `settings` function. For example, set the temporary value for resizing hints to `1` (where the default is `0`) to show resizing hints in the canvas.

```
s = settings;
s.matlab.appdesigner.designview.ShowResizingHints.TemporaryValue = 1
```

For more information about settings, see “Access and Modify Settings”.

Settings

matlab.appdesigner

`AutoSaveOnClickAway` — Autosave app on click away

`0` (default) | `1`

Autosave app on click away, specified as a `1` or `0`.

Set to `1` to automatically saves changes to an app when you click away from it to switch between apps or to bring another window into focus. If an app has not already been saved at least once, autosave has no effect.

Example:
`s.matlab.appdesigner.compbrowser.AutoSaveOnClickAway.TemporaryValue = 1`

matlab.appdesigner.compbrowser

`IncludeComponentLabels` — Include component labels

`0` (default) | `1`

Include component labels in the Component Browser, specified as a `1` or `0`.

Set to `1` to display labels that are grouped with components as separate items in the Component Browser. Otherwise, those labels do not appear in the Component Browser.

Example:
`s.matlab.appdesigner.compbrowser.IncludeComponentLabels.TemporaryValue = 1`

matlab.appdesigner.history

`MaxFileListSize` — Number of recent apps to display

`8` (default) | positive integer

Number of recent apps to display in the **Designer** tab **Open** menu, specified as a positive integer between 0 and 12. The default value is `8`.

Example:
`s.matlab.appdesigner.compbrowser.MaxFileListSize.TemporaryValue = 6`

matlab.appdesigner.designview

`ShowGrid` — Show the grid

`0` (default) | `1`

Show the grid, specified as `1` or `0`.

Set to `1` to overlay a grid onto the canvas as an alignment aid.

Example: `s.matlab.appdesigner.designview.ShowGrid.TemporaryValue = 1`

`GridInterval` — Grid spacing interval

`10` (default) | positive integer

Grid spacing interval in pixels, specified as a positive integer between 10 and 50.

`ShowGrid` must be set to `1` to see the change in grid spacing.

Example: `s.matlab.appdesigner.designview.GridInterval.TemporaryValue = 6`

`SnapToGrid` — Snap components to the grid

`1` (default) | `0`

Snap components to the grid, specified as `1` or `0`.

Set to `1` to have the upper left corner of a component always snap to the intersection of two grid lines whenever you resize or move the component on the canvas. `ShowGrid` must be set to 1 in order for components to snap to the grid.

Example: `s.matlab.appdesigner.designview.SnapToGrid.TemporaryValue = 0`

`ShowAlignmentHints` — Show alignment hints
`1` (default) | `0`

Show alignment hints, specified as `1` or `0`.

Set to `1` to display alignment hints as you resize or move a component on the canvas.

Example:
`s.matlab.appdesigner.designview.ShowAlignmentHints.TemporaryValue = 0`

`ShowResizingHints` — Show resizing hints
`0` (default) | `1`

Show resizing hints, specified as `1` or `0`.

Set to `1` to display the size of a component as you resize it on the canvas.

Example:
`s.matlab.appdesigner.designview.ShowResizingHints.TemporaryValue = 1`

See Also

`settings`

Topics

"Access and Modify Settings"
"App Designer Preferences"

Introduced in R2018a

append

Combine strings

Syntax

```
str = append(str1,...,strN)
```

Description

`str = append(str1,...,strN)` combines the text from `str1,...,strN`. Each input argument can be a string array, a character vector, or a cell array of character vectors.

- If any input is a string array, then the output is a string array.
- If any input is a cell array, and none are string arrays, then the output is a cell array of character vectors.
- If all inputs are character vectors, then the output is a character vector.

Unlike the `strcat` function, `append` preserves trailing whitespace characters from input arguments of all data types.

Examples

Combine Strings

Create two strings.

```
str1 = "Good";
str2 = "Morning";
```

Combine them using the `append` function.

```
str = append(str1,str2)
```

```
str =
"GoodMorning"
```

To add a space between the input strings, specify a space character as another input argument.

```
str = append(str1,' ',str2)
```

```
str =
"Good Morning"
```

As an alternative, you can use the `plus` operator to combine strings.

```
str = str1 + ' ' + str2
```

```
str =
"Good Morning"
```

However, the best practice is to use `append` when you do not know whether the input arguments are strings, character vectors, or cell arrays of character vectors.

Combine Character Vectors

Create two character vectors, with the first character vector having a trailing whitespace character.

```
chr1 = 'Hello ';
chr2 = 'World';
```

Combine them into one character vector.

```
chr3 = append(chr1,chr2)
```

```
chr3 =
'Hello World'
```

The `append` function always preserves trailing whitespace characters, unlike the `strcat` function. (`strcat` removes trailing whitespace characters from character vectors.)

```
chr4 = strcat(chr1,chr2)
```

```
chr4 =
'HelloWorld'
```

Combine Arrays of Text

You can combine string arrays or cell arrays of character vectors, element by element. Also, you can append a single piece of text to the elements of an input array.

Create an array of file names.

```
names = ["data" "report" "slides"]
```

```
names = 1x3 string array
    "data"    "report"    "slides"
```

Create an array of file extension names, with a different extension for each element of `names`.

```
ext = [".xlsx" ".docx" ".pptx"]
```

```
ext = 1x3 string array
    ".xlsx"    ".docx"    ".pptx"
```

Combine the file names and extensions.

```
str1 = append(names,ext)
```

```
str1 = 1x3 string array
    "data.xlsx"    "report.docx"    "slides.pptx"
```

To append the same extension to each name, use a character vector or a string scalar.

```
str2 = append(names,'.mat')
```

```
str2 = 1x3 string array
    "data.mat"    "report.mat"    "slides.mat"
```

Implicit Expansion

The `append` function supports implicit expansion of arrays. For example, you can combine strings from a column vector and a row vector to form a two-dimensional string array.

Create a column vector of strings. Then create a row vector.

```
str1 = ["A";"B";"C"]
```

```
str1 = 3x1 string array
    "A"
    "B"
    "C"
```

```
str2 = ["1" "2" "3" "4"]
```

```
str2 = 1x4 string array
    "1"    "2"    "3"    "4"
```

Combine `str1` and `str2`.

```
str = append(str1,str2)
```

```
str = 3x4 string array
    "A1"    "A2"    "A3"    "A4"
    "B1"    "B2"    "B3"    "B4"
    "C1"    "C2"    "C3"    "C4"
```

Input Arguments

`str1,...,strN` — Input text
string arrays | character vectors | cell arrays of character vectors

Input text, specified as string arrays, character vectors, or cell arrays of character vectors.

The `append` function supports input arguments that have *compatible sizes*.

String arrays and cell arrays of character vectors have compatible sizes if, for each dimension, one of these conditions is true:

- The lengths of that dimension are equal for all arrays.
- For one or more arrays, the length of that dimension is equal to 1. For the other arrays, the lengths are not equal to 1 but are equal to each other.

Character vectors are always compatible with all other input arguments. You can always append a character vector to another character vector, or to the elements of a string array or cell array of character vectors.

For more information on combining arrays with compatible sizes, see “Compatible Array Sizes for Basic Operations”.

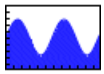
See Also

`cat` | `cellstr` | `horzcat` | `join` | `plus` | `strcat` | `strjoin` | `vertcat`

Introduced in R2019a

area

Filled area 2-D plot



Syntax

```
area(Y)
area(X,Y)
area(...,basevalue)
area(...,Name,Value)
area(ax,...)
ar = area(...)
```

Description

An area graph displays elements in `Y` as one or more curves and fills the area beneath each curve. When `Y` is a matrix, the curves are stacked showing the relative contribution of each row element to the total height of the curve at each x interval.

`area(Y)` plots the vector `Y` or plots each column in matrix `Y` as a separate curve and stacks the curves. The *x*-axis automatically scales to `1:size(Y,1)`. The values in `Y` can be numeric or duration values.

`area(X,Y)` plots `Y` versus `X` and fills the area between `0` and `Y`. The values in `X` can be numeric, datetime, duration or categorical values.

- If `Y` is a vector, then specify `X` as a vector of increasing values with length equal to `Y`. If the values in `X` are not increasing, then `area` sorts the values before plotting.
- If `Y` is a matrix, then specify `X` as a vector of increasing values with length equal to the number of rows in `Y`. `area` plots the columns of `Y` as filled areas. For each `X`, the net result is the sum of corresponding values from the rows of `Y`. You also can specify `X` as

a matrix with size equal to `Y`. To avoid unexpected output when `X` is a matrix, specify `X` so that the columns repeat.

`area(...,basevalue)` specifies the base value for the area fill. The default `basevalue` is `0`. Specify the base value as a numeric value.

`area(...,Name,Value)` modifies the area chart using one or more name-value pair arguments.

`area(ax,...)` plots into the axes `ax` instead of into the current axes (`gca`).

`ar = area(...)` returns one or more `Area` objects. The `area` function creates one `Area` object for vector input arguments. It creates one object per column for matrix input arguments.

Creating an area graph of an *m*-by-*n* matrix creates *n* area objects (that is, one per column), whereas a 1-by-*n* vector creates one area object.

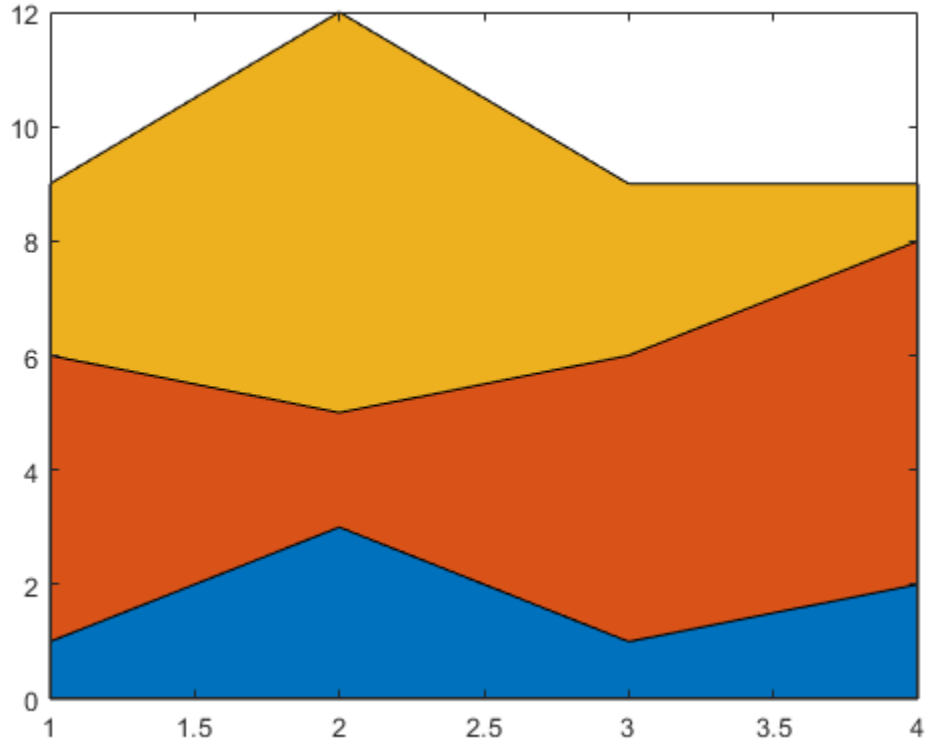
Some area object properties that you set on an individual area object set the values for all area objects in the graph. See Area for information on specific properties.

Examples

Create Area Graph

Plot the data in matrix Y as an area graph.

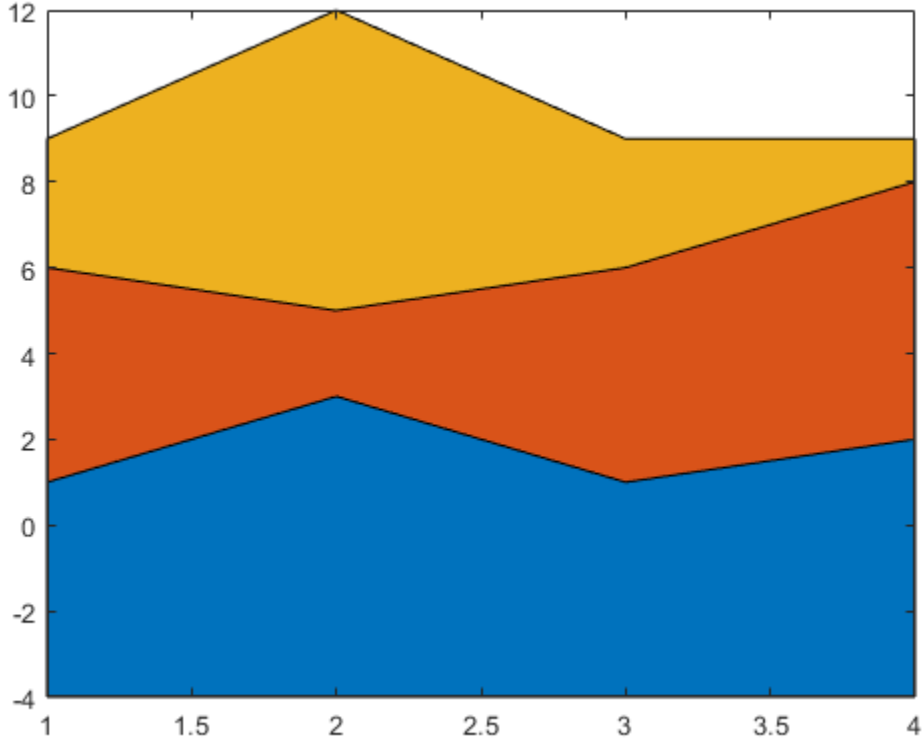
```
Y = [1, 5, 3;
     3, 2, 7;
     1, 5, 3;
     2, 6, 1];
figure
area(Y)
```



Adjust Base Value of Area Graph

By default, `area` uses the *y*-axis as the base value. Change the base value by setting the `basevalue` input argument to -4.

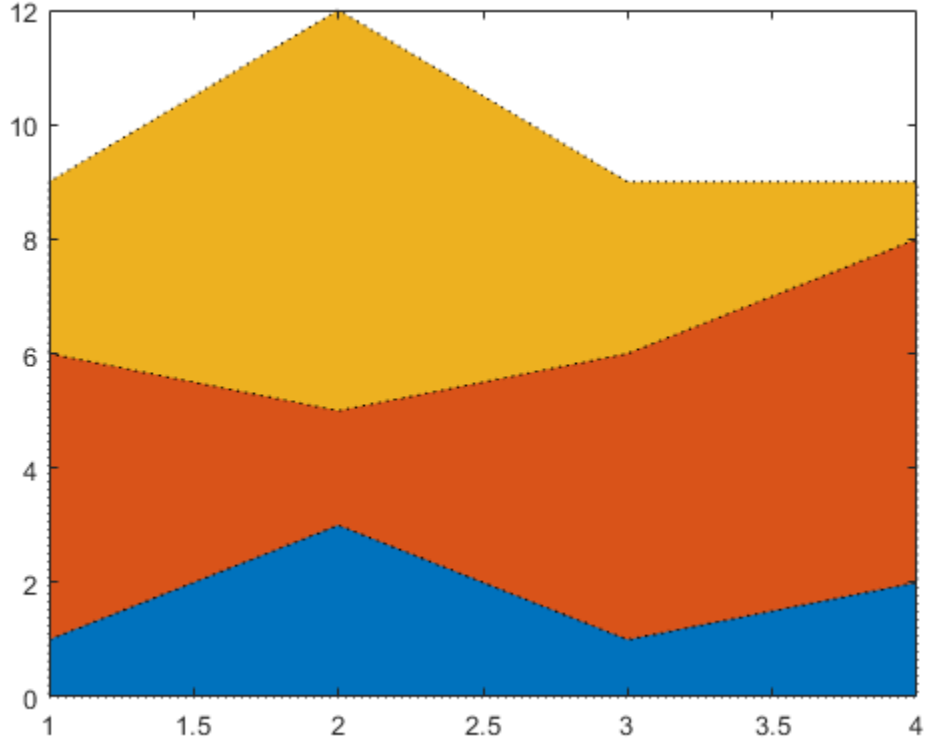
```
Y = [1, 5, 3;
     3, 2, 7;
     1, 5, 3;
     2, 6, 1];
figure
basevalue = -4;
area(Y,basevalue)
```



Specify Color and Line Style for Area Plot

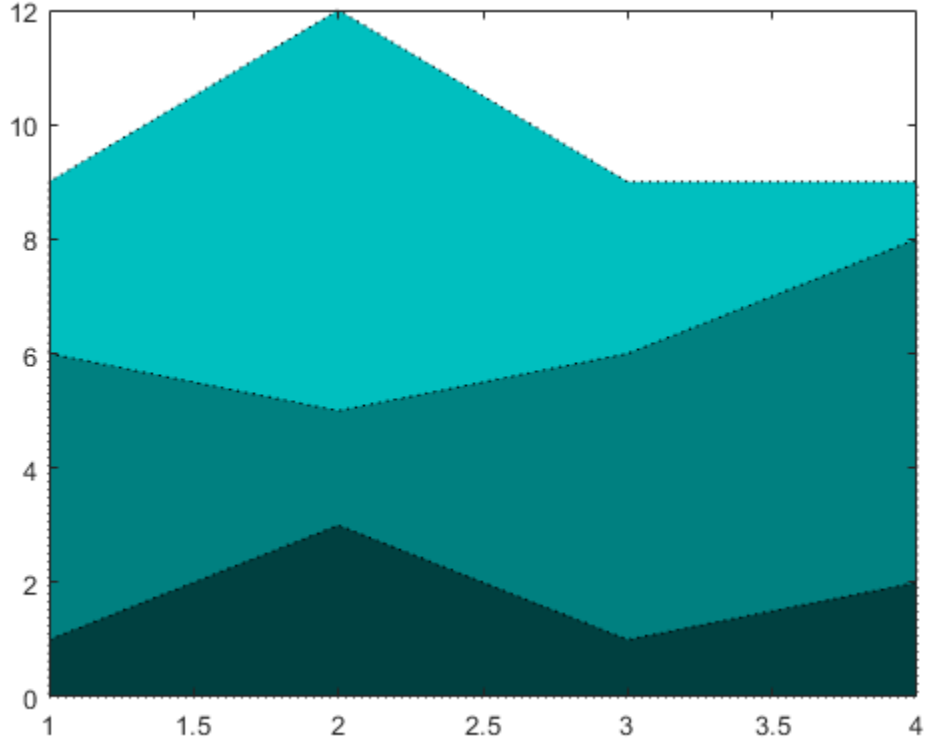
Create an area plot of `Y` and use a dotted line style. Return the three area objects in array `h`. The `area` function creates one area object for each column in `Y`.

```
Y = [1, 5, 3;
     3, 2, 7;
     1, 5, 3;
     2, 6, 1];
h = area(Y,'LineStyle',':');
```



Change the area colors using RGB triplet color values. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead.

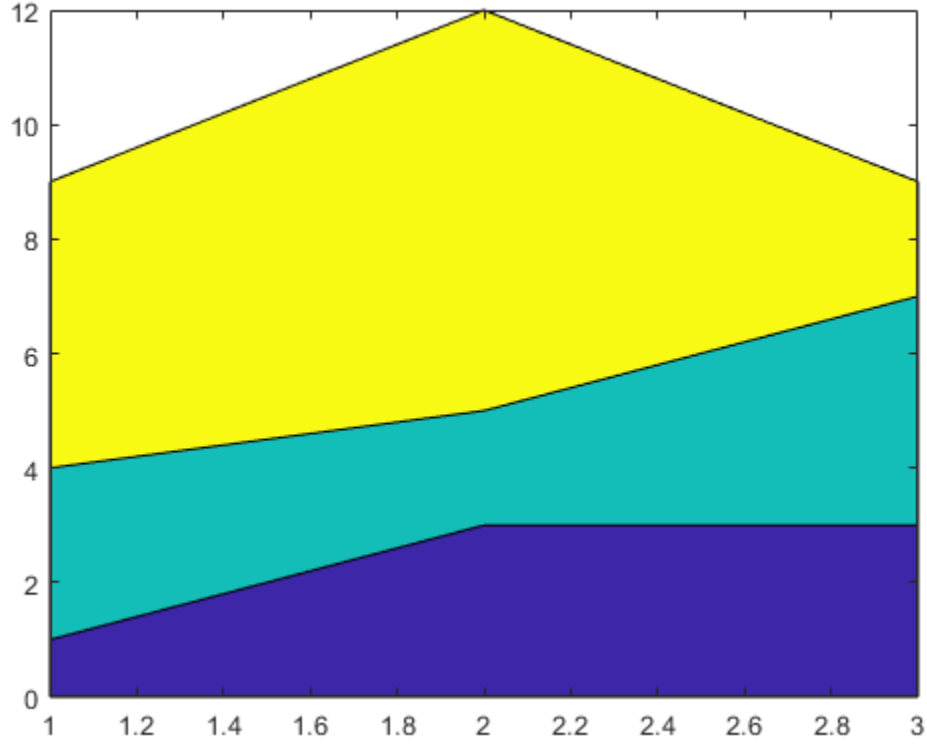
```
h(1).FaceColor = [0 0.25 0.25];
h(2).FaceColor = [0 0.5 0.5];
h(3).FaceColor = [0 0.75 0.75];
```



Area Chart with Colormap Colors

Create an area chart that uses colormap colors by setting the `FaceColor` property to `'flat'`.

```
y = [1 3 5; 3 2 7; 3 4 2];
area(y,'FaceColor','flat')
```



Tips

- If `Y` contains `NaN` values, then the `area` function breaks up the area around the missing values.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions
`bar` | `plot` | `sort`

Properties
Area

Topics

“Plot Dates and Durations”
“Plot Categorical Data”
“Compare Data Sets Using Overlayed Area Graphs”

Introduced before R2006a

Area Properties

Area chart appearance and behavior

Description

`Area` properties control the appearance and behavior of an `Area` object. By changing property values, you can modify certain aspects of the area chart.

Starting in R2014b, you can use dot notation to query and set properties.

```
ar = area(1:10);
ec = ar.EdgeColor;
ar.EdgeColor = 'red';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Color and Styling

`FaceColor` — Area fill color

RGB triplet | `'r'` | `'g'` | `'b'` | `'flat'` | ...

Area fill color, specified as an RGB triplet or one of the color options in this table.

Starting in R2017b, the default value is an RGB triplet from the `ColorOrder` property of the axes. In previous releases, the default value was `'flat'` and the colors were based on the colormap.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`
`'none'`	No color	Not applicable
`'flat'`	Colors from the axes colormap	Not applicable

`FaceColorMode` — Control how `FaceColor` is set
`'auto'` (default) | `'manual'`

Control how the `FaceColor` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the `FaceColor` property by selecting a color from the `ColorOrder` property of the axes.
- `'manual'` — You control the value of the `FaceColor` property manually, either by specifying a color when you call a charting function that creates the object, or by setting the `FaceColor` property on the object after creating it.

If you change the value of the `FaceColor` property manually, MATLAB changes the value of the `FaceColorMode` property to `'manual'`.

`EdgeColor` — Area outline color
`[0 0 0]` (default) | RGB triplet | `'r'` | `'g'` | `'b'` | `'flat'` | ...

Area outline color, specified as an RGB triplet or one of the color options in the table. The default value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`
`'none'`	No color	Not applicable
`'flat'`	Colors from the axes colormap	Not applicable

`FaceAlpha` — Face transparency
1 (default) | scalar in range `[0,1]`

Face transparency, specified as a scalar in the range `[0,1]`. A value of 1 is opaque and 0 is completely transparent. Values between 0 and 1 are semitransparent.

Example: `a.FaceAlpha = 0.5;`

`EdgeAlpha` — Edge transparency
1 (default) | scalar in range `[0,1]`

Edge transparency, specified as a scalar in the range `[0,1]`. A value of 1 is opaque and 0 is completely transparent. Values between 0 and 1 are semitransparent.

Example: `a.EdgeAlpha = 0.5;`

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Area outline width
`0.5` (default) | scalar numeric value

Area outline width, specified as a scalar numeric value in point units. One point equals 1/72 inch.

Example: `1.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`AlignVertexCenters` — Sharp vertical and horizontal lines
`'off'` (default) | `'on'`

Sharp vertical and horizontal lines, specified as `'off'` or `'on'`.

If the associated figure has a `GraphicsSmoothing` property set to `'on'` and a `Renderer` property set to `'opengl'`, then the figure applies a smoothing technique to plots. In some cases, this smoothing technique can cause vertical and horizontal lines to appear uneven in thickness or color. Use the `AlignVertexCenters` property to eliminate the uneven appearance.

- `'off'` — Do not sharpen vertical or horizontal lines. The lines might appear uneven in thickness or color.
- `'on'` — Sharpen vertical and horizontal lines to eliminate an uneven appearance.

Note You must have a graphics card that supports this feature. To see if the feature is supported, call the `rendererinfo` function. If it is supported, `rendererinfo` returns value of `1` for `info.Details.SupportsAlignVertexCenters`.

Baseline

`BaseValue` — Baseline value

`0` (default) | numeric scalar value

Baseline value, specified as a numeric scalar value.

The baseline value that you specify applies to either the *x*-axis or the *y*-axis depending on the bar chart orientation. If you change the orientation of the bar chart from vertical to horizontal, or vice versa, the baseline value might change. Set the `BaseValue` property after setting the `Horizontal` property.

`ShowBaseLine` — Baseline visibility

`'on'` (default) | `'off'`

Baseline visibility, specified as one of these values:

- `'on'` — Show the baseline.
- `'off'` — Hide the baseline.

`BaseLine` — Baseline

baseline object

This property is read-only.

Baseline object. For a list of baseline properties, see Baseline.

Data

`XData` — *x*-coordinates

`[ ]` (default) | vector

x-coordinates, specified as a vector. Alternatively, specify the *x*-coordinates using the input argument `X` to the `area` function. If you do not specify `X`, then `area` uses the indices of the values in `YData` as the *x*-coordinates.

`XData` and `YData` must have equal lengths.

Example: `1:10`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`XDataMode` — Selection mode for `XData`
`'auto'` (default) | `'manual'`

Selection mode for `XData`, specified as one of these values:

- `'auto'` — Use the indices of the values in `YData`.
- `'manual'` — Use manually specified values. To specify the values, set the `XData` property or pass the input argument `X` to the `area` function.

`XDataSource` — Variable linked to `XData`
`''` (default) | character vector | string

Variable linked to `XData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `XData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `XData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'x'`

`YData` — *y*-coordinates
`[]` (default) | vector

y-coordinates, specified as a vector. Alternatively, specify the *y*-coordinates using the input argument `Y` to the `area` function.

`XData` and `YData` must have equal lengths.

Example: `1:10`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`YDataSource` — Variable linked to `YData`
`''` (default) | character vector | string

Variable linked to `YData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `YData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `YData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'y'`

Legend

`DisplayName` — Legend label
`''` (default) | character vector | string scalar

Legend label, specified as a character vector or string scalar. The legend does not display until you call the `legend` command. If you do not specify the text, then `legend` sets the label using the form `'dataN'`.

`Annotation` — Control for including or excluding object from legend
`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`DataTipTemplate` — Data tip content

`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

`Clipping` — Clipping of object to axes limits

`'on'` (default) | `'off'`

Clipping of the object to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the object that are outside the axes limits.
- `'off'` — Display the entire object, even if parts of it appear outside the axes limits. Parts of the object might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the object so that it is larger than the original plot.

The `Clipping` property of the axes that contains the object must be set to `'on'`. Otherwise, this property has no effect. For more information about the clipping behavior, see the `Clipping` property of the axes.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see "Callback Definition".

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you

do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks only when visible. The `Visible` property must be set to `'on'`. The `HitTest` property determines if the `Area` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `Area` object passes the click to the object behind it in the current view of the figure window. The `HitTest` property of the `Area` object has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `Area` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `Area` object that has one of these:
  - `HitTest` property set to `'on'`

- `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

Note The `PickableParts` property determines if the `Area` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent

`Axes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `Group`, or `Transform` object.

`Children` — Children

empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the chart object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'area'`

This property is read-only.

Type of graphics object, returned as `'area'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`area`

Topics

“Access Property Values”
“Graphics Object Properties”

Introduced before R2006a

array2table

Convert homogeneous array to table

Syntax

```
T = array2table(A)
T = array2table(A,Name,Value)
```

Description

`T = array2table(A)` converts the `m`-by-`n` array, `A`, to an `m`-by-`n` table, `T`. Each column of `A` becomes a variable in `T`.

`array2table` uses the input array name appended with the column number for the variable names in the table. If these names are not valid MATLAB identifiers, `array2table` uses names of the form `'Var1',...,'VarN'`, where *N* is the number of columns in `A`.

`T = array2table(A,Name,Value)` creates a table from an array, `A`, with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify row names or variable names to include in the table.

Examples

Convert Numeric Array to Table

Create an array of numeric data.

```
A = [1 4 7; 2 5 8; 3 6 9]
```

```
A = 3×3

     1     4     7
```

```
     2     5     8
     3     6     9
```

Convert the array, A, to a table.

```
T = array2table(A)
```

```
T=3×3 table
    A1    A2    A3
    __    __    __

    1     4     7
    2     5     8
    3     6     9
```

The table has variable names that append the column number to the input array name, A.

Convert Array to Table Including Variable Names

Create an array of numeric data.

```
A = [1 12 30.48; 2 24 60.96; 3 36 91.44]
```

```
A = 3×3

    1.0000   12.0000   30.4800
    2.0000   24.0000   60.9600
    3.0000   36.0000   91.4400
```

Convert the array, A, to a table and include variable names.

```
T = array2table(A,...
    'VariableNames',{'Feet','Inches','Centimeters'})
```

```
T=3×3 table
    Feet    Inches    Centimeters
    ____    ______    ___________

     1        12         30.48
     2        24         60.96
```

```
    3        36         91.44
```

Input Arguments

A — Input array
matrix

Input array, specified as a matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `cell`
Complex Number Support: Yes

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'RowNames',{'row1','row2','row3'}` uses the row names, `row1`, `row2`, and `row3` for the table, `T`.

`RowNames` — Row names for `T`
`{}` (default) | cell array of character vectors | string array

Row names for `T`, specified as the comma-separated pair consisting of `'RowNames'` and a cell array of character vectors or string array, whose elements are nonempty and distinct. The number of names must equal the number of rows, `size(A,1)`.

`VariableNames` — Variable names for `T`
cell array of character vectors | string array

Variable names for `T`, specified as the comma-separated pair consisting of `'VariableNames'` and a cell array of character vectors or a string array, whose elements are nonempty and distinct. The number of names must equal the number of variables, `size(A,2)`.

Variable names can have any Unicode® characters, including spaces and non-ASCII characters.

Output Arguments

T — Output table
table

Output table, returned as a table. The table can store metadata such as descriptions, variable units, variable names, and row names. For more information, see the Properties section of `table`.

Tips

- If `A` is a cell array, use `cell2table(A)` to create a table from the contents of the cells in `A`. Each variable in the table is numeric or a cell array of character vectors. `array2table(A)` creates a table where each variable is a column of cells.

Compatibility Considerations

Variable names can contain leading and trailing whitespace characters

Behavior changed in R2019b

Table and timetable variable names with leading or trailing whitespace characters are not modified.

In previous releases, leading and trailing whitespace characters were deleted from variable names when you specified them using the `'VariableNames'` name-value pair argument, or assigned them to the `VariableNames` property.

To manually remove such characters, first use the `strtrim` function on the names, then assign them as variable names to the table or timetable.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

The `'RowNames'` name-value pair is not supported.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- In generated code, you must specify the `'VariableNames'` name-value pair argument when using this function. For more information, see “Code Generation for Tables” (MATLAB Coder) and “Table Limitations for Code Generation” (MATLAB Coder).

See Also

`cell2table` | `isvarname` | `struct2table` | `table` | `table2array`

Topics

“Access Data in Tables”

Introduced in R2013b

array2timetable

Convert homogeneous array to timetable

Syntax

```
TT = array2timetable(X,'RowTimes',rowTimes)
TT = array2timetable(X,'SampleRate',Fs)
TT = array2timetable(X,'TimeStep',dt)
TT = array2timetable( ___ ,'StartTime',t0)
TT = array2timetable( ___ ,'VariableNames',varNames)
```

Description

`TT = array2timetable(X,'RowTimes',rowTimes)` converts `X` and `rowTimes` to a timetable. `X` is an `M`-by-`N` array and `rowTimes` is an `M`-by-`1` vector of datetime or duration values. Each column of `X` becomes a variable of `TT`. The time values in `rowTimes` label the rows of the timetable. `TT` is an `M`-by-`N` timetable.

For more information on creating and using timetables, see “Timetables”.

`TT = array2timetable(X,'SampleRate',Fs)` uses the sample rate `Fs` to calculate regularly spaced row times. `Fs` is a positive numeric scalar that specifies the number of samples per second (Hz). The first row time is zero seconds.

`TT = array2timetable(X,'TimeStep',dt)` uses the time step `dt` to calculate regularly spaced row times. `dt` is a duration or calendar duration value that specifies the length of time between consecutive row times. The first row time is zero seconds.

`TT = array2timetable( ___ ,'StartTime',t0)` specifies start time `t0`, instead of zero seconds, as the first row time. You can use this syntax when you create a regular timetable using either the `'SampleRate'` or `'TimeStep'` name-value pair arguments from either of the previous two syntaxes.

`TT = array2timetable( ___ ,'VariableNames',varNames)` specifies the names of the variables in the output timetable. You can use this syntax with any of the input arguments of the previous syntaxes.

Examples

Convert Array to Timetable

Convert an array to a timetable. Add a vector of durations as the row times.

```
X = rand(5,3);
Time = seconds(1:5);
TT = array2timetable(X,'RowTimes',Time)
```

```
TT=5×4 timetable
    Time       X1         X2         X3
    _____    _______    _______    _______

    1 sec    0.81472    0.09754    0.15761
    2 sec    0.90579     0.2785    0.97059
    3 sec    0.12699    0.54688    0.95717
    4 sec    0.91338    0.95751    0.48538
    5 sec    0.63236    0.96489    0.80028
```

Return the size of the timetable. TT contains five rows and three variables. The set of row times is a table property. The row times are not contained in a table variable.

```
size(TT)
```

```
ans = 1×2

     5     3
```

Specify Sample Rate

Specify a sample rate of 500 Hz, and convert an array to a timetable.

```
X = rand(5,3);
TT = array2timetable(X,'SampleRate',500)
```

```
TT=5×4 timetable
      Time         X1         X2         X3
    _________    _______    _______    _______
```

```
    0 sec        0.81472    0.09754    0.15761
    0.002 sec    0.90579     0.2785    0.97059
    0.004 sec    0.12699    0.54688    0.95717
    0.006 sec    0.91338    0.95751    0.48538
    0.008 sec    0.63236    0.96489    0.80028
```

Specify Time Step

Specify a time step of 0.5 seconds, and convert an array to a timetable.

```
X = rand(5,3);
TT = array2timetable(X,'TimeStep',seconds(0.5))
```

```
TT=5×4 timetable
     Time        X1         X2         X3
    _______    _______    _______    _______

    0 sec      0.81472    0.09754    0.15761
    0.5 sec    0.90579     0.2785    0.97059
    1 sec      0.12699    0.54688    0.95717
    1.5 sec    0.91338    0.95751    0.48538
    2 sec      0.63236    0.96489    0.80028
```

Specify Start Time

Specify a time step of 15 seconds, and a start time of 5 minutes.

```
X = rand(5,3);
TT = array2timetable(X,'TimeStep',seconds(15),'StartTime',minutes(5))
```

```
TT=5×4 timetable
      Time        X1         X2         X3
    ________    _______    _______    _______

    5 min       0.81472    0.09754    0.15761
    5.25 min    0.90579     0.2785    0.97059
    5.5 min     0.12699    0.54688    0.95717
```

```
    5.75 min    0.91338    0.95751    0.48538
    6 min       0.63236    0.96489    0.80028
```

Specify Variable Names

Convert an array to a timetable. Specify the row times as a datetime vector. Specify names for the timetable variables.

```
X = randi([70 90],5,3);
Time = datetime(2016,7,1:5);
varNames = {'Temp1','Temp2','Temp3'};
TT = array2timetable(X,'RowTimes',Time,'VariableNames',varNames)
```

```
TT=5×4 timetable
       Time        Temp1    Temp2    Temp3
    ___________    _____    _____    _____

    01-Jul-2016     87       72       73
    02-Jul-2016     89       75       90
    03-Jul-2016     72       81       90
    04-Jul-2016     89       90       80
    05-Jul-2016     83       90       86
```

Input Arguments

X — Input matrix
matrix

Input matrix.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `categorical` | `datetime` | `duration` | `calendarDuration`
Complex Number Support: Yes

`rowTimes` — Times associated with rows of timetable
datetime vector | duration vector

Times associated with rows of a timetable, specified as a datetime vector or a duration vector. Each time labels a row in the output timetable. The time values in `rowTimes` do not need to be unique, sorted, or regular.

`Fs` — Sample rate
positive numeric scalar

Sample rate, specified as a positive numeric scalar. `Fs` specifies the number of samples per second (Hz).

`dt` — Time step
datetime scalar | duration scalar

Time step, specified as a datetime scalar or duration scalar.

Data Types: `datetime` | `duration` | `calendarDuration`

`t0` — Start time
datetime scalar | duration scalar

Start time, specified as a datetime scalar or duration scalar.

- If `t0` is a datetime value, then the row times of `TT` are datetime values.
- If `t0` is a duration, then the row times are durations.

If the time step `dt` is a calendar duration value, then `t0` must be a datetime value.

Data Types: `datetime` | `duration`

`varNames` — Names of variables
cell array of character vectors | string array

Names of the variables in the output timetable, specified as a cell array of character vectors or a string array. The number of names in `varNames` must equal the number of variables.

Variable names can have any Unicode characters, including spaces and non-ASCII characters.

Output Arguments

TT — Output timetable
timetable

Output timetable. The timetable can store metadata such as descriptions, variable units, variable names, and row times. For more information, see the Properties section of `timetable`.

Tips

- In certain cases, you can call `array2timetable` with a syntax that specifies a regular time step between row times, and yet `array2timetable` returns an irregular timetable. This result occurs when you specify the time step using a calendar unit of time and there is a row time that introduces an irregular step. For example, if you create a timetable with a time step of one calendar month, starting on January 31, 2019, then it is irregular with respect to months.

```
stime = datetime(2019,1,31);
tstep = calmonths(1);
X = [1:3]';
TT = array2timetable(X,'TimeStep',tstep,'StartTime',stime)
```

```
TT =

  3×1 timetable

       Time        X
    ___________    _

    31-Jan-2019    1
    28-Feb-2019    2
    31-Mar-2019    3
```

In addition, there are other cases where irregularities are due to shifts from Daylight Saving Time (DST) or to row times that are leap seconds. This table specifies the row time values and time steps that can produce irregular timetables unexpectedly.

Row Time Value	Time Step
Start time specified as the 29th, 30th, or 31st day of the month	Number of calendar months or quarters
Start time specified as February 29	Number of calendar years
Any row time occurring between 1:00 a.m. and 2:00 a.m. on a day shifting from DST to standard time (when row times are specified as datetime values whose time zone observes DST)	Number of calendar days or months
Any row time that is a leap second (when row times are specified as datetime values whose time zone is the `UTCLeapSeconds` time zone)	Time step specified in any calendar unit (days, weeks, months, quarters, or years)

Compatibility Considerations

Variable names can contain leading and trailing whitespace characters

Behavior changed in R2019b

Table and timetable variable names with leading or trailing whitespace characters are not modified.

In previous releases, leading and trailing whitespace characters were deleted from variable names when you specified them using the `'VariableNames'` name-value pair argument, or assigned them to the `VariableNames` property.

To manually remove such characters, first use the `strtrim` function on the names, then assign them as variable names to the table or timetable.

`'SamplingRate'` is not recommended

Not recommended starting in R2018b

The `'SamplingRate'` name-value pair argument is not recommended. Use `'SampleRate'` instead. The corresponding timetable property is also named `SampleRate`.

For backward compatibility, you still can specify `'SamplingRate'` as the name of the name-value pair. However, the value is assigned to the `SampleRate` property.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

See Also

`isvarname` | `summary` | `table2timetable` | `timetable` | `uitable`

Topics

“Create Timetables”
“Clean Timetable with Missing, Duplicate, or Nonuniform Times”
“Resample and Aggregate Data in Timetable”
“Combine Timetables and Synchronize Their Data”
“Select Timetable Data by Row Time and Variable Type”
“Access Data in Tables”
“Tables”
“Represent Dates and Times in MATLAB”

Introduced in R2016b

arrayfun

Apply function to each element of array

Syntax

```
B = arrayfun(func,A)
B = arrayfun(func,A1,...,An)
B = arrayfun( ___ ,Name,Value)
[B1,...,Bm] = arrayfun( ___ )
```

Description

`B = arrayfun(func,A)` applies the function `func` to the elements of `A`, one element at a time. `arrayfun` then concatenates the outputs from `func` into the output array `B`, so that for the `i`th element of `A`, `B(i) = func(A(i))`. The input argument `func` is a function handle to a function that takes one input argument and returns a scalar. The output from `func` can have any data type, so long as objects of that type can be concatenated. The arrays `A` and `B` have the same size.

You cannot specify the order in which `arrayfun` calculates the elements of `B` or rely on them being done in any particular order.

`B = arrayfun(func,A1,...,An)` applies `func` to the elements of the arrays `A1,...,An`, so that `B(i) = func(A1(i),...,An(i))`. The function `func` must take `n` input arguments and return a scalar. The arrays `A1,...,An` all must have the same size.

`B = arrayfun( ___ ,Name,Value)` applies `func` with additional options specified by one or more `Name,Value` pair arguments. For example, to return output values in a cell array, specify `'UniformOutput',false`. You can return `B` as a cell array when `func` returns values that cannot be concatenated into an array. You can use `Name,Value` pair arguments with the input arguments of either of the previous syntaxes.

`[B1,...,Bm] = arrayfun( ___ )` returns multiple output arrays `B1,...,Bm` when `func` returns `m` output values. `func` can return output arguments that have different data types, but the data type of each output must be the same each time `func` is called. You can use this syntax with any of the input arguments of the previous syntaxes.

The number of output arguments from `func` need not be the same as the number of input arguments specified by `A1,...,An`.

Examples

Apply Function to Field of Structure Array

Create a nonscalar structure array. Each structure has a field that contains a vector of random numbers. The vectors have different sizes.

```
S(1).f1 = rand(1,5);
S(2).f1 = rand(1,10);
S(3).f1 = rand(1,15)
```

```
S=3×1 struct
    f1
```

Calculate the mean for each field in `S` by using the `arrayfun` function. You cannot use `structfun` for this calculation because the input argument to `structfun` must be a scalar structure.

```
A = arrayfun(@(x) mean(x.f1),S)
```

```
A = 1×3

    0.6786    0.6216    0.6069
```

Return Object Array

Create a structure array in which each structure has two fields containing numeric arrays.

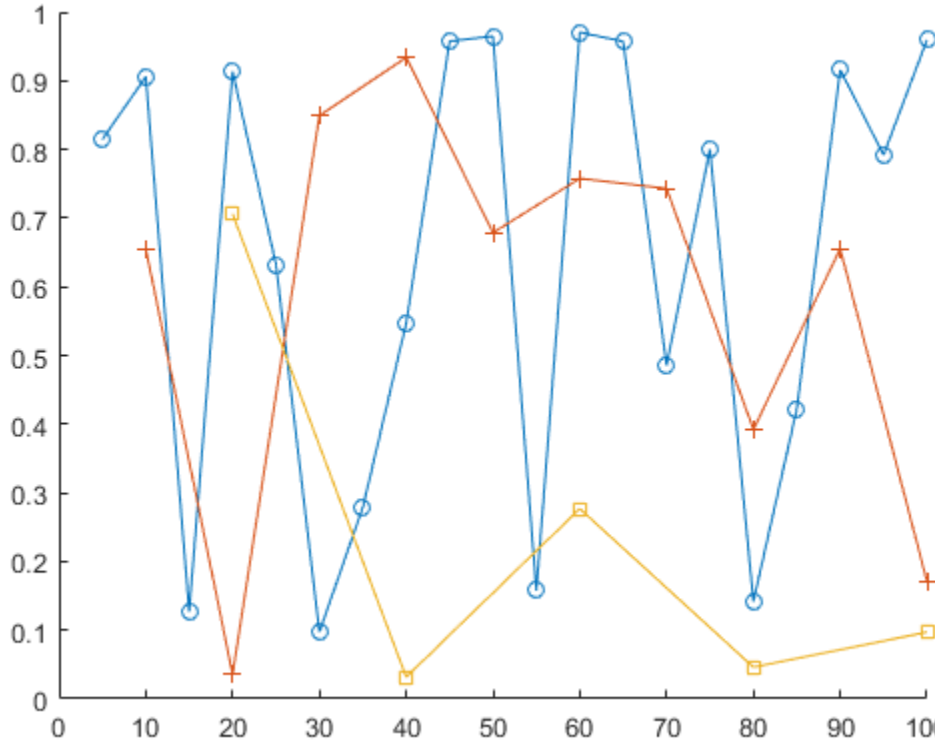
```
S(1).X = 5:5:100; S(1).Y = rand(1,20);
S(2).X = 10:10:100; S(2).Y = rand(1,10);
S(3).X = 20:20:100; S(3).Y = rand(1,5)
```

```
S=3×2 struct
    X
```

Y

Plot the numeric arrays. Return an array of chart line objects from the `plot` function and use them to add different markers to each set of data points. `arrayfun` can return arrays of any data type so long as objects of that data type can be concatenated.

```
figure
hold on
p = arrayfun(@(a) plot(a.X,a.Y),S);
p(1).Marker = 'o';
p(2).Marker = '+';
p(3).Marker = 's';
hold off
```



Return Outputs in Cell Array

Create a nonscalar structure array. Each structure has a field that contains numeric matrices.

```
S(1).f1 = rand(3,5);
S(2).f1 = rand(6,10);
S(3).f1 = rand(4,2)
```

```
S=3×1 struct
    f1
```

Calculate the mean for each field in `S` by using the `arrayfun` function. `mean` returns vectors containing the mean of each column, so the means cannot be returned as an array. To return the means in a cell array, specify the `'UniformOutput',false` name-value pair.

```
A = arrayfun(@(x) mean(x.f1),S,'UniformOutput',false)
```

```
A=1×3 cell
    {1x5 double}    {1x10 double}    {1x2 double}
```

Return Multiple Output Arrays

Create a nonscalar structure array.

```
S(1).f1 = 1:10;
S(2).f1 = [2; 4; 6];
S(3).f1 = []
```

```
S=3×1 struct
    f1
```

Calculate the sizes of each field of `S` by using the `arrayfun` function. The number of rows and columns are each in 1-by-3 numeric arrays.

```
[nrows,ncols] = arrayfun(@(x) size(x.f1),S)
```

```
nrows = 1×3

     1     3     0


ncols = 1×3

    10     1     0
```

Input Arguments

func — Function to apply
function handle

Function to apply to the elements of the input arrays, specified as a function handle.

`func` can correspond to more than one function file and therefore can represent a set of overloaded functions. In these cases, MATLAB determines which function to call based on the class of the input arguments.

Example: `B = arrayfun(@round,A)` returns the integer part of each element of `A`.

A — Input array
array

Input array. `A` can have any data type.

If you define the class that `A` belongs to, and you also overload the `subsref` or `size` methods of `A`, then `arrayfun` places these requirements on `A`:

- The `size` method of `A` must return an array of doubles.
- `A` must support linear indexing.
- The product of the sizes returned by the `size` method must not exceed the limit of `A`, as defined by linear indexing into `A`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `A = arrayfun(@(x) mean(x.f1),S,'UniformOutput',false)` returns the means in a cell array. `S` is a structure array in which each structure has a field named `f1`.

`UniformOutput` — True or false
`true` (default) | `false`

True or false, specified as the comma-separated pair consisting of `'UniformOutput'` and either `true` (`1`) or `false` (`0`).

Value of `'UniformOutput'`	Description
`true` (`1`)	`func` must return scalars that `arrayfun` concatenates into arrays.
`false` (`0`)	`arrayfun` returns the outputs of `func` in cell arrays. The outputs of `func` can have any sizes and different data types.

`ErrorHandler` — Function to catch errors
function handle

Function to catch errors, specified as the comma-separated pair consisting of `'ErrorHandler'` and a function handle. If `func` throws an error, then the error handler specified by `'ErrorHandler'` catches the error and takes the action specified in the function. The error handler either must throw an error or return the same number of outputs as `func`. If the value of `'UniformOutput'` is true, then the output arguments of the error handler must be scalars and have the same data type as the outputs of `func`.

The first input argument of the error handler is a structure with these fields:

- `identifier` — Error identifier
- `message` — Error message text
- `index` — Linear index into the input arrays at which `func` threw the error

The remaining input arguments to the error handler are the input arguments for the call to `func` that made `func` throw the error.

Suppose `func` returns two doubles as output arguments. You can specify the error handler as `'ErrorHandler',@errorFunc`, where `errorFunc` is a function that raises a warning and returns two output arguments.

```
function [A,B] = errorFunc(S,varargin)
    warning(S.identifier, S.message);
    A = NaN;
    B = NaN;
end
```

If you do not specify `'ErrorHandler'`, then `arrayfun` rethrows the error thrown by `func`.

Output Arguments

B — Output array
array of any data type | cell array

Output array, returned as an array of any data type or as a cell array.

By default, `arrayfun` concatenates the outputs from `func` into an array. `func` must return scalars. If `func` returns objects, then the class that the objects belong to must meet these requirements.

- Support assignment by linear indexing into the object array
- Have a `reshape` method that returns an array that has the same size as the input

If the value of the `'UniformOutput'` name-value pair argument is `false` (`0`), then `arrayfun` returns outputs in a cell array. In that case, the outputs from `func` can have any sizes and different data types.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with these limitations:

- The specified function must not rely on `persistent` variables.
- The `'ErrorHandler'` name-value pair is not supported.
- With the `'UniformOutput'` name-value pair set to `true` (default), the outputs from the specified function must be numeric, logical, characters, or cell arrays.

For more information, see “Tall Arrays for Out-of-Memory Data”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support the `'ErrorHandler'` option.
- Code generation does not support the `'UniformOutput'` option. The input function to `arrayfun` must return a scalar or scalars that can be concatenated into an array.
- Code generation does not support cell array inputs to `func`.
- To predetermine the output type for `func`, the code generator can call `func` before processing the `accumarray` input arguments.
  - If the execution of `func` causes side-effects, for instance by modifying a global or persistent variable or printing to output, then the generated code results can differ from MATLAB results.
  - If the input array to `accumarray` is empty, then the code generator can use zero-valued inputs to predetermine output types. `func` must not error when its inputs are zero, or the generated code can produce unexpected errors.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `arrayfun` in the Parallel Computing Toolbox™ documentation.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cell2mat` | `cellfun` | `spfun` | `splitapply` | `structfun`

Topics

“Anonymous Functions”
“Create Function Handle”

Introduced before R2006a

ascii

Set FTP transfer mode to ASCII

Syntax

```
ascii(ftpobj)
```

Description

`ascii(ftpobj)` sets the FTP transfer mode for the server associated with `ftpobj` to ASCII. Use ASCII mode for text files, such as HTML pages and Rich Text Format (RTF) files.

Examples

Set Transfer Mode to ASCII

Download a text file from an FTP server, using the ASCII transfer mode.

First, connect to the National Centers for Environmental Information (NCEI) FTP server.

```
ftpobj = ftp('ftp.ngdc.noaa.gov')
```

```
ftpobj = 

  FTP Object
     host: ftp.ngdc.noaa.gov
     user: anonymous
      dir: /
     mode: binary
```

The default transfer mode is binary. When you download a text file, you can use ASCII mode instead so that newline characters are converted correctly.

Set the transfer mode to ASCII. To show the current transfer mode, display the FTP object.

```
ascii(ftpobj)
ftpobj
```

```
ftpobj =

  FTP Object
     host: ftp.ngdc.noaa.gov
     user: anonymous
      dir: /
     mode: ascii
```

Download a text file and close the connection.

```
mget(ftpobj,'README.txt');
close(ftpobj)
```

Display the beginning of `README.txt`. To read the copy of `README.txt` downloaded to your computer, use the `fileread` function.

```
readme = fileread('README.txt');
readme(1:95)
```

```
ans =
    '                   Welcome to the
         NOAA/National Centers for Environmental Information (NCEI)'
```

FTP service courtesy of the National Centers for Environmental Information (NCEI). See the NCEI Privacy Policy, Disclaimer, and Copyright for NCEI terms of service.

Input Arguments

`ftpobj` — Connection to FTP server
FTP object

Connection to an FTP server, specified as an FTP object.

See Also

`binary` | `ftp`

Introduced before R2006a

asec

Inverse secant in radians

Syntax

```
Y = asec(X)
```

Description

`Y = asec(X)` returns the "Inverse Secant" on page 1-529 ($\sec^{-1}$) of the elements of `X` in radians. The function accepts both real and complex inputs.

- For real values of `X` in the interval [-∞, -1] and [1, ∞], `asec(X)` returns values in the interval [0, π].
- For real values of `X` in the interval (-1, 1) and for complex values of `X`, `asec(X)` returns complex values.

Examples

Inverse Secant of a Value

Find the inverse secant of a value.

```
asec(-2.8)
```

```
ans = 1.9360
```

Inverse Secant of a Vector of Complex Values

Find the inverse secant of the elements of vector `x`. The `asec` function acts on `x` element-wise.

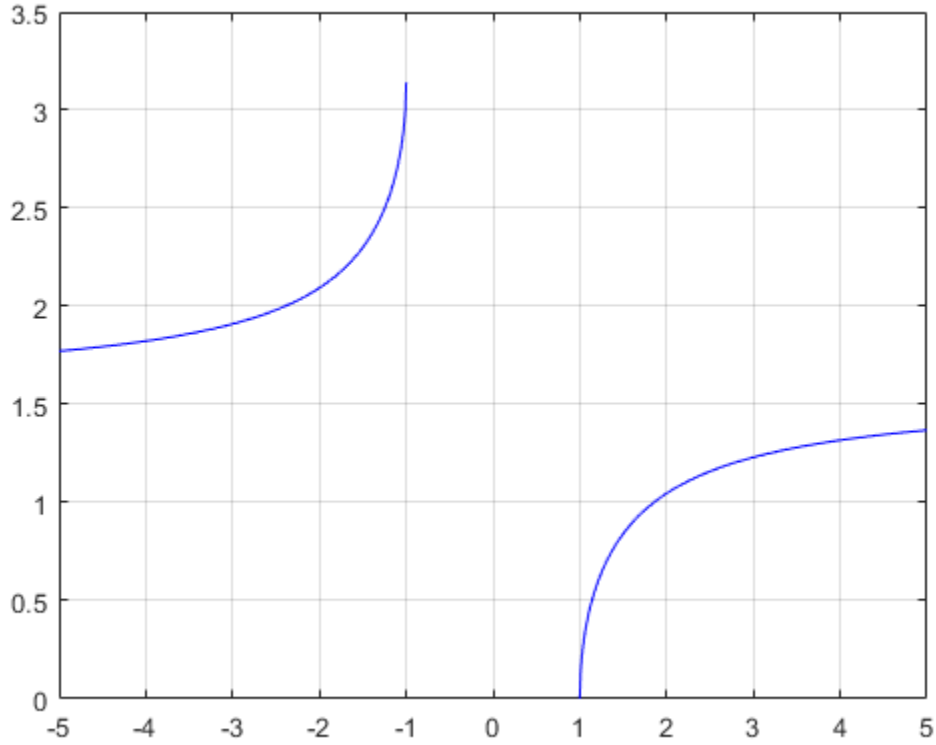
```
x = [0.5i 1+3i -2.2+i];
Y = asec(x)
```

```
Y = 1×3 complex

   1.5708 + 1.4436i   1.4749 + 0.2970i   1.9503 + 0.1833i
```

Plot the Inverse Secant Function

Plot the inverse secant function over the intervals $-5 \le x \le -1$ and $1 \le x \le 5$.

```
x1 = -5:0.01:-1;
x2 = 1:0.01:5;
plot(x1,asec(x1),'b')
hold on
plot(x2,asec(x2),'b')
grid on
```



Input Arguments

X — Secant of angle
scalar | vector | matrix | multidimensional array

Secant of angle, specified as a scalar, vector, matrix, or multidimensional array. The `asec` operation is element-wise when X is nonscalar.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Inverse Secant

The inverse secant is defined as

$$\sec^{-1}(z) = \cos^{-1}\left(\frac{1}{z}\right).$$

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- If the output of the function running on the GPU can be complex, then you must explicitly specify its input arguments as complex. For more information, see “Work with Complex Numbers on a GPU” (Parallel Computing Toolbox).

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`acos` | `asecd` | `sec` | `secd`

Introduced before R2006a

asecd

Inverse secant in degrees

Syntax

```
Y = asecd(X)
```

Description

`Y = asecd(X)` returns the inverse secant ($\sec^{-1}$) of the elements of `X` in degrees. The function accepts both real and complex inputs.

- For real values of `X` in the intervals [-∞, -1] and [1, ∞], `asecd(X)` returns values in the interval [0, 180].
- For real values of `X` in the interval (-1, 1) and for complex values of `X`, `asecd(X)` returns complex values.

Examples

Inverse Secant of Vector

```
x = [10 1 Inf];
y = asecd(x)
```

```
y = 1×3

   84.2608         0   90.0000
```

The `asecd` operation is element-wise when you pass a vector, matrix, or N-D array.

Inverse Secant of Complex Value

```
asecd(1+i)
```

```
ans = 64.0864 + 30.4033i
```

Input Arguments

X — Secant of angle
scalar value | vector | matrix | N-D array

Secant of angle, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `asecd` operation is element-wise when X is non-scalar.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Angle in degrees
scalar value | vector | matrix | N-D array

Angle in degrees, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as X.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`asec` | `sec` | `secd`

Introduced before R2006a

asech

Inverse hyperbolic secant

Syntax

```
Y = asech(X)
```

Description

`Y = asech(X)` returns the inverse hyperbolic secant of the elements of `X`. The function accepts both real and complex inputs. All angles are in radians.

Examples

Inverse Hyperbolic Secant of Vector

Find the inverse hyperbolic secant of the elements of vector `X`. The `asech` function acts on `X` element-wise.

```
X = [2 -3 1+2i];
Y = asech(X)
```

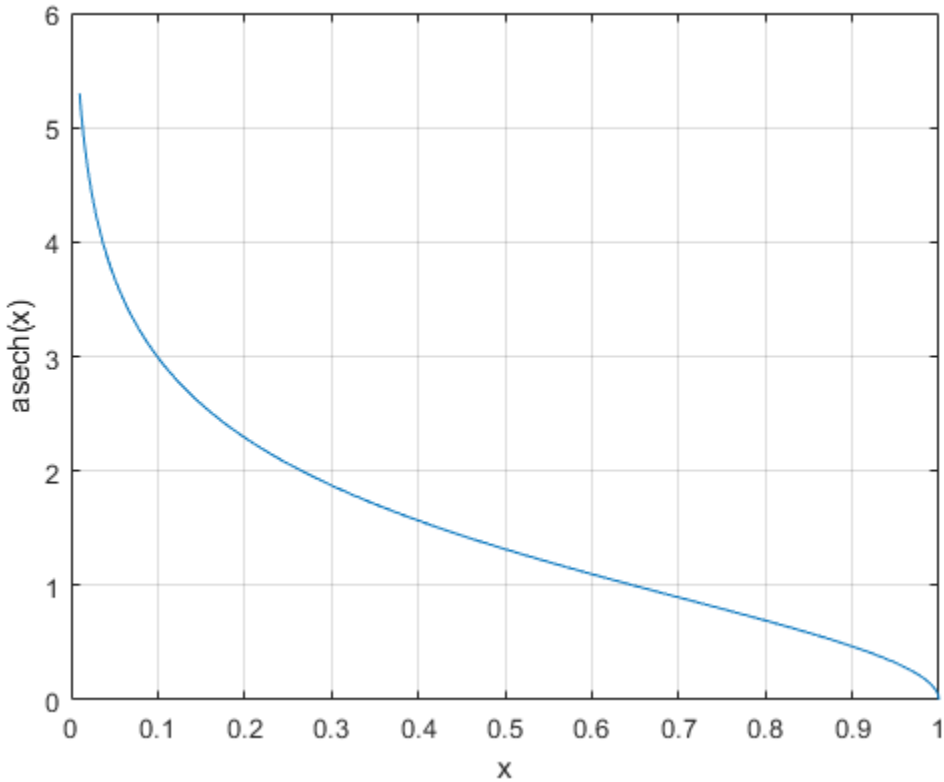
```
Y = 1×3 complex

   0.0000 + 1.0472i   0.0000 + 1.9106i   0.3966 - 1.3845i
```

Plot the Inverse Hyperbolic Secant Function

Plot the inverse hyperbolic secant function over the interval $0.01 \le x \le 1$.

```
x = 0.01:0.001:1;
plot(x,asech(x))
```

```
grid on
xlabel('x')
ylabel('asech(x)')
```



Input Arguments

X — Hyperbolic secant of angle
scalar | vector | matrix | multidimensional array

Hyperbolic secant of angle, specified as a scalar, vector, matrix, or multidimensional array. The `asech` operation is element-wise when `X` is nonscalar.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Inverse Hyperbolic Secant

For real values x in the domain $0 < x \leq 1$, the inverse hyperbolic secant satisfies

$$\operatorname{sech}^{-1}(x) = \cosh^{-1}\left(\frac{1}{x}\right) = \log\left(\frac{1}{x} + \sqrt{\frac{1}{x^2} - 1}\right).$$

For complex numbers $z = x + iy$ as well as real values in the regions $-\infty < z \leq 0$ and $1 \leq z < \infty$, the call `asech(z)` returns complex results.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- If the output of the function running on the GPU can be complex, then you must explicitly specify its input arguments as complex. For more information, see “Work with Complex Numbers on a GPU” (Parallel Computing Toolbox).

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`acosh` | `asec` | `asinh` | `sech`

Introduced before R2006a

asin

Inverse sine in radians

Syntax

```
Y = asin(X)
```

Description

`Y = asin(X)` returns the "Inverse Sine" on page 1-541 ($\sin^{-1}$) of the elements of `X` in radians. The function accepts both real and complex inputs.

- For real values of `X` in the interval [-1, 1], `asin(X)` returns values in the interval [-π/2, π/2].
- For real values of `X` outside the interval [-1, 1] and for complex values of `X`, `asin(X)` returns complex values.

Examples

Inverse Sine of Value

Find the inverse sine of a value.

```
y = asin(1)
```

```
y = 1.5708
```

Inverse Sine of Vector of Complex Values

Find the inverse sine of the elements of vector `x`. The `asin` function acts on `x` element-wise.

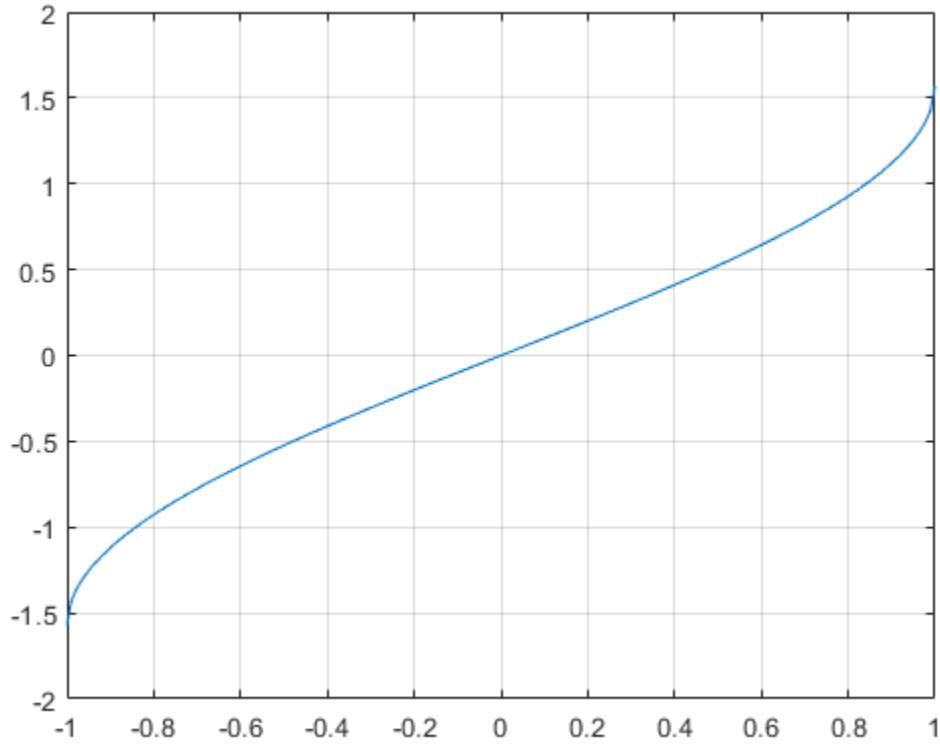
```
x = [0.5i 1+3i -2.2+i];
y = asin(x)
```

```
y = 1×3 complex

   0.0000 + 0.4812i   0.3076 + 1.8642i  -1.1091 + 1.5480i
```

Plot Inverse Sine Function

Plot the inverse sine function over the intervals $-1 \le x \le 1$.

```
x = -1:.01:1;
plot(x,asin(x))
grid on
```



Input Arguments

X — Sine of angle
scalar | vector | matrix | multidimensional array

Sine of angle, specified as a scalar, vector, matrix, or multidimensional array. The `asin` operation is element-wise when X is nonscalar.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Inverse Sine

The inverse sine is defined as

$$\sin^{-1}(z) = -i\log\left[iz + (1 - z^2)^{1/2}\right].$$

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Generates an error during simulation and returns `NaN` in generated code when the input value `X` is real, but the output should be complex. To get the complex result, make the input value complex by passing in `complex(X)`.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Generates an error during simulation and returns `NaN` in generated code when the input value `X` is real, but the output should be complex. To get the complex result, make the input value complex by passing in `complex(X)`.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- If the output of the function running on the GPU can be complex, then you must explicitly specify its input arguments as complex. For more information, see “Work with Complex Numbers on a GPU” (Parallel Computing Toolbox).

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`acsc` | `asind` | `sin` | `sind`

Introduced in R2006a

asind

Inverse sine in degrees

Syntax

```
Y = asind(X)
```

Description

`Y = asind(X)` returns the inverse sine ($\sin^{-1}$) of the elements of `X` in degrees. The function accepts both real and complex inputs.

- For real values of `X` in the interval [-1, 1], `asind(X)` returns values in the interval [-90, 90].
- For real values of `X` outside the interval [-1, 1] and for complex values of `X`, `asind(X)` returns complex values.

Examples

Inverse Sine of Scalar

Show that the inverse sine of 1 is exactly 90°.

```
asind(1)
```

```
ans = 90
```

Round-Trip Calculation for Complex Angles

Show that the inverse sine, followed by sine, returns the original values of `X`.

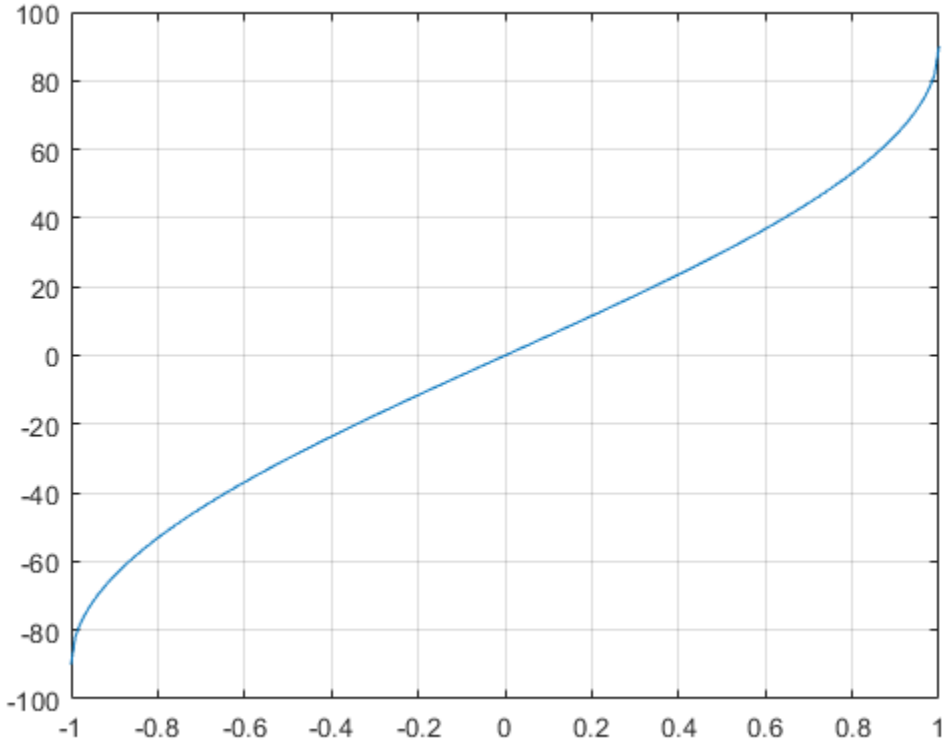
```
sind(asind([2 3]))
```

```
ans = 1×2

     2     3
```

Graph of Inverse Sine Function

Plot the inverse sine function over the domain $-1 \le x \le 1$.

```
x = -1:.01:1;
plot(x,asind(x))
grid on
```



Input Arguments

X — Sine of angle
scalar value | vector | matrix | N-D array

Sine of angle, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `asind` operation is element-wise when X is nonscalar.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Angle in degrees
scalar value | vector | matrix | N-D array

Angle in degrees, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as X.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`asin` | `sin` | `sind`

Introduced before R2006a

asinh

Inverse hyperbolic sine

Syntax

```
Y = asinh(X)
```

Description

`Y = asinh(X)` returns the inverse hyperbolic sine of the elements of `X`. The function accepts both real and complex inputs. All angles are in radians.

Examples

Inverse Hyperbolic Sine of Vector

Find the inverse hyperbolic sine of the elements of vector `X`. The `asinh` function acts on `X` element-wise.

```
X = [2 -3 1+2i];
Y = asinh(X)
```

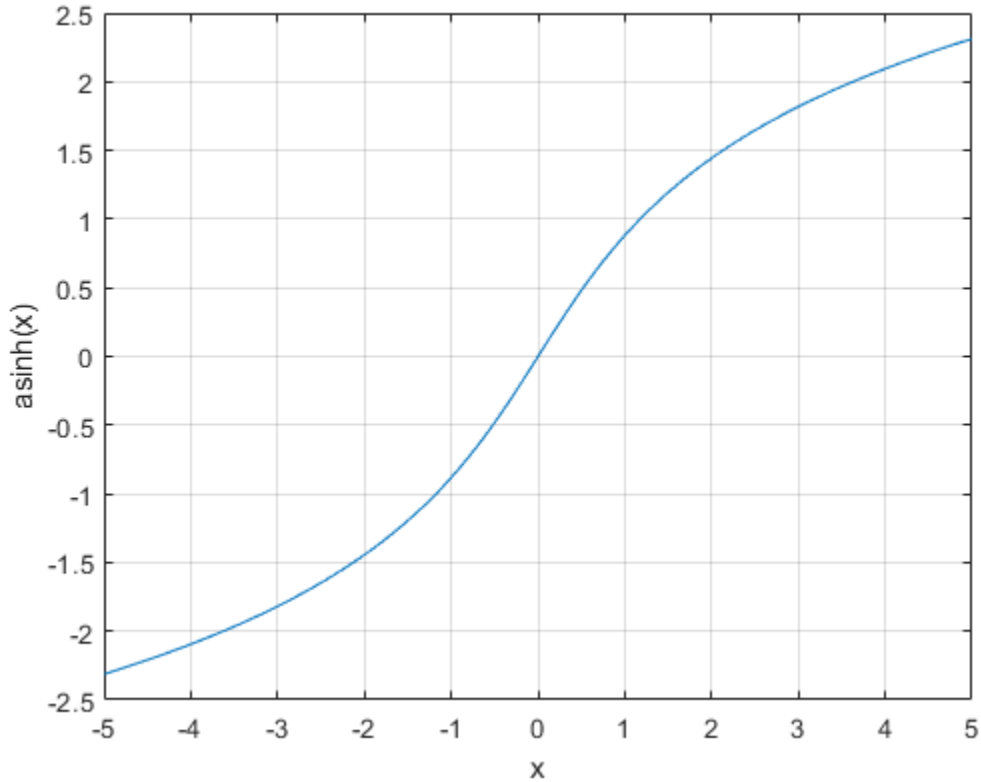
```
Y = 1×3 complex

   1.4436 + 0.0000i  -1.8184 + 0.0000i   1.4694 + 1.0634i
```

Plot the Inverse Hyperbolic Sine Function

Plot the inverse hyperbolic sine function over the interval $-5 \le x \le 5$.

```
x = -5:.01:5;
plot(x,asinh(x))
```

```
grid on
xlabel('x')
ylabel('asinh(x)')
```



Input Arguments

X — Hyperbolic sine of angle
scalar | vector | matrix | multidimensional array

Hyperbolic sine of angle, specified as a scalar, vector, matrix, or multidimensional array. The `asinh` operation is element-wise when `X` is nonscalar.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Inverse Hyperbolic Sine

For real values x in the domain of all real numbers, the inverse hyperbolic sine satisfies

$$\sinh^{-1}(x) = \log\left(x + \sqrt{x^2 + 1}\right).$$

For complex numbers $z = x + iy$, the call `asinh(z)` returns complex results.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`acosh` | `asin` | `atanh` | `sinh`

Introduced before R2006a

assert

Throw error if condition false

Syntax

```
assert(cond)

assert(cond,msg)
assert(cond,msg,A1,...,An)

assert(cond,msgID,msg)
assert(cond,msgID,msg,A1,...,An)
```

Description

`assert(cond)` throws an error if `cond` is false.

`assert(cond,msg)` throws an error and displays the error message, `msg`, if `cond` is false.

`assert(cond,msg,A1,...,An)` displays an error message that contains formatting conversion characters, such as those used with the MATLAB `sprintf` function, if `cond` is false. Each conversion character in `msg` is converted to one of the values `A1,...,An`.

`assert(cond,msgID,msg)` throws an error, displays the error message, `msg`, and includes an error identifier on the exception, if `cond` is false. The identifier enables you to distinguish errors and to control what happens when MATLAB encounters the errors.

`assert(cond,msgID,msg,A1,...,An)` includes an error identifier on the exception and displays a formatted error message.

Examples

Value in Expected Range

Assert that the value, x, is greater than a specified minimum value.

```
minVal = 7;
x = 26;

assert(minVal < x)
```

The expression evaluates as true, and the assertion passes.

Assert that the value of x is between the specified minimum and maximum values.

```
maxVal = 13;

assert((minVal < x) && (x < maxVal))
```

```
Assertion failed.
```

The expression evaluates as false. The assertion fails and MATLAB throws an error.

Expected Data Type

Assert that the product of two numbers is a double-precision number.

```
a = 13;
b = single(42);
c = a*b;

assert(isa(c,'double'),'Product is not type double.')
```

```
Product is not type double.
```

Enhance the error message to display the data type of c.

```
assert(isa(c,'double'),'Product is type %s, not double.',class(c))
```

```
Product is type single, not double.
```

Expected Code Conditions

Use the `assert` function to test for conditions that should not happen in normal code execution. If the coefficients are numeric, the computed roots should be numeric. A quadratic equation using the specified coefficients and computed roots should be zero.

```
function x = quadraticSolver(C)

validateattributes(C,{'numeric'},{'size',[1 3]})

a = C(1);
b = C(2);
c = C(3);

x(1) = (-b+sqrt(b^2-4*a*c))/(2*a);
x(2) = (-b-sqrt(b^2-4*a*c))/(2*a);
assert(isnumeric(x),'quadraticSolver:nonnumericRoots',...
    'Computed roots are not numeric')

y1 = a*x(1)^2+b*x(1)+c;
y2 = a*x(2)^2+b*x(2)+c;
assert(y1 == 0,'quadraticSolver:root1Error','Error in first root')
assert(isequal(y2,0),'quadraticSolver:root2Error','Error in second root')

end
```

Input Arguments

`cond` — Condition to assert
MATLAB expression

Condition to assert, specified as a valid MATLAB expression. If `cond` is false, the `assert` function throws an error. `cond` can include relational operators (such as `<` or `==`) and logical operators (such as `&&`, `||`, or `~`). Use the logical operators `and` and `or` to create compound expressions. MATLAB evaluates compound expressions from left to right, adhering to operator precedence rules.

Example: `a<0`

Example: `exist('myfunction.m','file')`

`msg` — Information about assertion failure
character vector | string scalar

Information about the assertion failure, specified as a character vector or string scalar. This message displays as the error message. To format the message, use escape sequences, such as `\t` or `\n`. You also can use any format specifiers supported by the `sprintf` function, such as `%s` or `%d`. Specify values for the conversion specifiers via the `A1,...,An` input arguments. For more information, see "Formatting Text".

Note You must specify more than one input argument with `assert` if you want MATLAB to convert special characters (such as `\t`, `\n`, `%s`, and `%d`) in the error message.

Example: `'Assertion condition failed.'`

`A1,...,An` — Numeric, character, or string arrays
arrays

Numeric, character, or string arrays. This input argument provides the values that correspond to and replace the conversion specifiers in `msg`.

`msgID` — Identifier for assertion failure
character vector | string scalar

Identifier for the assertion failure, specified as a character vector or string scalar. Use the message identifier to help identify the source of the error or to control a selected subset of the errors in your program.

The message identifier includes a `component` and `mnemonic`. The identifier must always contain a colon and follows this simple format: `component:mnemonic`. The `component` and `mnemonic` fields must each begin with a letter. The remaining characters can be alphanumerics (A–Z, a–z, 0–9) and underscores. No whitespace characters can appear anywhere in `msgID`. For more information, see "Message Identifiers".

Example: `'MATLAB:singularMatrix'`

Example: `'MATLAB:narginchk:notEnoughInputs'`

Tips

- When you issue an error, MATLAB captures information about it and stores it in a data structure that is an object of the `MException` class. You can access information in the exception object by using `try/catch`. Or, if your program terminates because of an

exception and returns control to the Command Prompt, you can use `MException.last`.

- If an assertion failure occurs within a `try` block, MATLAB does not cease execution of the program. In this case, MATLAB passes control to the `catch` block.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Generates specified error messages at compile time only if all input arguments are constants or depend on constants. Otherwise, generates specified error messages at run time.
- If called with more than 1 argument, has no effect in standalone code even when run-time error detection is enabled. See “Run-Time Error Detection and Reporting in Standalone C/C++ Code” (MATLAB Coder).
- See “Rules for Using assert Function” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`MException` | `error`

Topics

“Capture Information About Exceptions”

Introduced in R2007a

assignin

Assign value to variable in specified workspace

Syntax

`assignin(ws,var,val)`

Description

`assignin(ws,var,val)` assigns the value `val` to the variable `var` in the workspace `ws`. For example, `assignin('base','x',42)` assigns the value 42 to the variable `x` in the MATLAB base workspace.

If `val` requires evaluation, MATLAB evaluates it in the function that calls `assignin`, not in the workspace specified by `ws`. If `val` is a function handle, it must be evaluable in the function that calls `assignin`.

The `assignin` function is useful for these tasks:

- Exporting data from a function to the base workspace.
- From within a function, changing the value of a variable that is defined in the workspace of the caller function. For example, you can change the value of a variable in the calling-function argument list.

Examples

Update Base Workspace Variable from Function

In a file in your current working folder, create a function that adds two numbers and then assigns a value to a variable `fcnStatus` in the base workspace.

```
function c = myAdd(a,b)
    c = a+b;
```

```
    str = sprintf('%s called with %d,%d (%s)',mfilename,a,b,char(datetime));
    assignin('base','fcnStatus',str)
end
```

At the command prompt, call the function.

```
n = myAdd(2,3)
```

```
n =

     5
```

View the value of the `fcnStatus` variable that the `myAdd` function assigned in the base workspace.

```
fcnStatus
```

```
fcnStatus =

    'myAdd called with 2,3 (17-Nov-2017 14:56:14)'
```

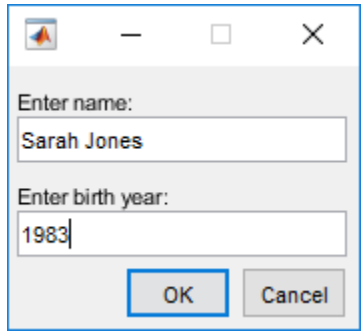
Save Data from Dialog Box to Base Workspace

In a file in your current working folder, create a function that displays a dialog box to input a name and a birth year and computes age in the year 2050. The `assignin` function exports the values to the MATLAB workspace variables `name` and `age2050`.

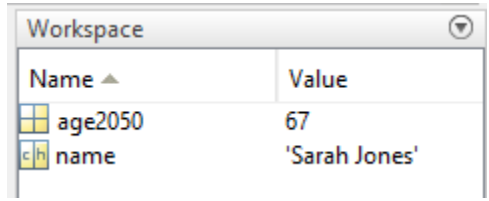
```
function mydialog
    prompt = {'Enter name:','Enter birth year:'};
    answer = inputdlg(prompt);

    n = answer{1};
    birthyear = str2double(answer{2});
    a = 2050-birthyear;

    assignin('base','name',n);
    assignin('base','age2050',a);
end
```

At the command prompt, run the function, enter data, and click OK.

```
mydialog
```



View the exported values in the Workspace browser.



Change Value of Variable in Calling Function

Create a function that changes an input age to 42. The call to `assignin` in `localfcn` changes the value of `a` in the workspace of the main function, `updateAge`.

```
function updateAge(a)
    validateattributes(a,{'numeric'},{'scalar'})
    fprintf('\tYour age: %d\n',a)
    localfcn
    fprintf('\tYour updated age: %d\n',a)
end

function localfcn
    assignin('caller','a',42)
end
```

At the command prompt, call the main function.

```
updateAge(37)
```

```
    Your age: 37
    Your updated age: 42
```

While this example describes how to assign a variable into the caller workspace, the best practice is to have the local function `localfcn` return the updated age as an output argument.

Assign Function Handle to Variable in Specified Workspace

In a file in your current working folder, create a function that finds the minimum value of a random array. The `assignfh` local function assigns the function handle `fh` into the workspace of `minRand`. The `minRand` function evaluates `fh` with the input `n`.

```
function m = minRand(n)
    assignfh

    A = fh(n)
    m = min(A(:));
end

function assignfh
    fh = @(dim)rand(dim);
    assignin('caller','fh',fh)
end
```

Call the function with an input value of 2.

```
m = minRand(2)
```

```
A =

    0.3486    0.1423
    0.0419    0.0766


m =

    0.0419
```

The function handle evaluates to a 2-by-2 array of random numbers.

Create another version of the function, called `minRand2`, in which the local function overrides the `rand` function in the function handle definition.

Similar to the `minRand` example, the `assignfh2` local function assigns `fh` into the workspace of `minRand2`. The `assignfh2` function overrides the `rand` function in its workspace with a variable named `rand` and creates the function handle. This behavior is

consistent with anonymous functions – the function handle is created using variables available at the time you create it. Therefore, the function handle evaluation in `minRand2` results in `n` indexing into the `rand` array defined in `assignfh2`.

```
function m = minRand2(n)
    assignfh2(n)

    A = fh(n)
    m = min(A(:));
end

function assignfh2(n)
    rand = 13*ones(n);
    fh = @(dim)rand(dim);
    assignin('caller','fh',fh)
end
```

Call the function with an input value of 2.

```
m = minRand2(2)
```

```
A =

    13


m =

    13
```

When assigning an anonymous function to a caller workspace, MATLAB puts the definition of the function handle in a variable in the caller workspace. The function with the call to `assignin` evaluates the function handle. While this example describes how to assign a variable into the caller workspace, the best practice is to have the local function `assignfh` return the function handle as an output argument.

Input Arguments

`ws` — Workspace
`'base'` | `'caller'`

Workspace, specified as `'base'` or `'caller'`.

To assign values in the MATLAB base workspace, use `'base'`. The base workspace stores variables that you create at the MATLAB command prompt, including any variables that scripts create, assuming that you run the script from the command line or from the Editor.

To assign variables in the workspace of the caller function, use `'caller'`. The caller workspace is the workspace of the function that called the currently running function. For example, assume that `funA` calls `funB`. The caller workspace of `funB` is `funA`. Therefore, from `funB`, you can assign a value to a variable in `funA` using `assignin` and specifying the workspace as `'caller'`.

Note Assigning to variables in the caller workspace can make code more difficult to understand, give surprising results to the user (unexpected or redefined variables in their workspace), and have a negative performance impact. The best practice is to have the function return the variables as output arguments.

The base and caller workspaces are equivalent in the following cases:

- You call a function at the command prompt and the main function calls `assignin`.
- You call `assignin` at the command prompt.

Data Types: `char` | `string`

`var` — Variable name
character vector | string scalar

Variable name, specified as a character vector or string scalar. If `var` does not exist in the specified workspace, the `assignin` function creates it.

Data Types: `char` | `string`

`val` — Value
scalar | array

Value of variable, specified as a scalar or array value. `val` can have any data type, and can include MATLAB expressions.

If the value of the variable requires evaluation, MATLAB evaluates the expression in the function that contains the call to `assignin`, not in the workspace specified by `ws`. If `val` is a function handle, it must be evaluable in the function that calls `assignin`.

Example: `5`

Example: `'hello'`

Example: `rand(3,7)`

Example: `@cos`

Tips

- The `assignin` function does not assign values to specific elements of an array. Therefore `var` cannot contain array indices. This code results in an error.

  ```
  X = 1:8;
  assignin('base','X(3:5)',-1);
  ```

  To assign values to specific elements of an array, use the `evalin` function.

  ```
  evalin('base','X(3:5) = -1')
  ```

See Also

`evalin`

Topics

"Base and Function Workspaces"

Introduced before R2006a

atan

Inverse tangent in radians

Syntax

```
Y = atan(X)
```

Description

`Y = atan(X)` returns the "Inverse Tangent" on page 1-566 ($\tan^{-1}$) of the elements of `X` in radians. The function accepts both real and complex inputs.

- For real values of `X`, `atan(X)` returns values in the interval [-π/2, π/2].
- For complex values of `X`, `atan(X)` returns complex values.

Examples

Inverse Tangent of a Value

Find the inverse tangent of a value.

```
atan(0.8)
```

```
ans = 0.6747
```

Inverse Tangent of a Vector of Complex Values

Find the inverse tangent of the elements of vector `x`. The `atan` function acts on `x` element-wise.

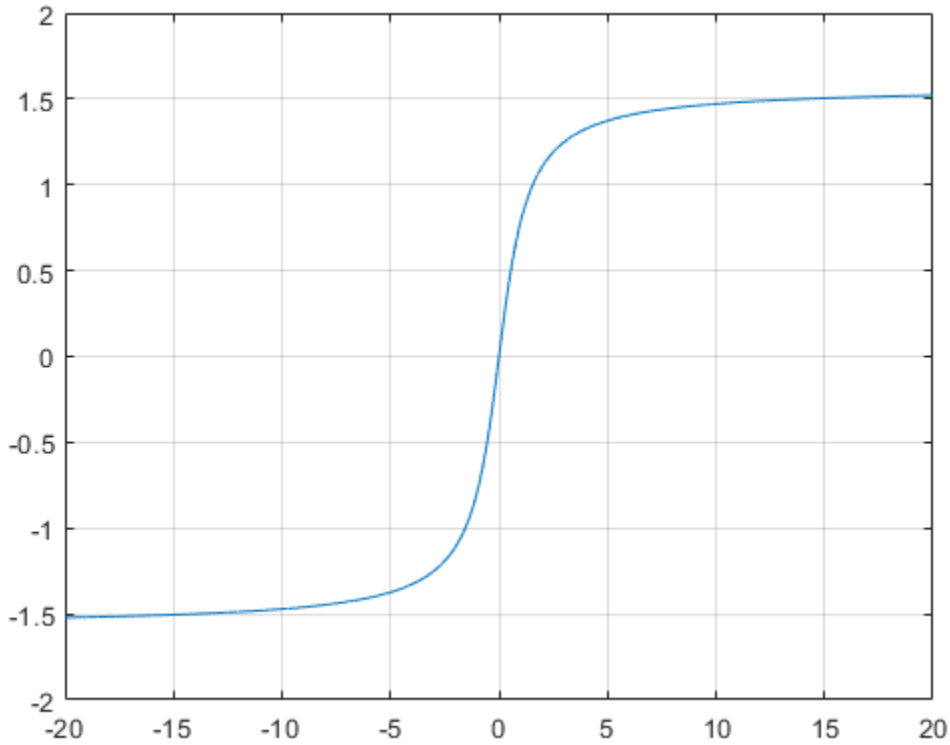
```
x = [0.5i 1+3i -2.2+i];
Y = atan(x)
```

```
Y = 1×3 complex

   0.0000 + 0.5493i   1.4615 + 0.3059i  -1.2019 + 0.1506i
```

Plot the Inverse Tangent Function

Plot the inverse tangent function over the interval $-20 \le x \le 20$.

```
x = -20:0.01:20;
plot(x,atan(x))
grid on
```



Input Arguments

X — Tangent of angle
scalar | vector | matrix | multidimensional array

Tangent of angle, specified as a scalar, vector, matrix, or multidimensional array. The `atan` operation is element-wise when `X` is nonscalar.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Inverse Tangent

The inverse tangent is defined as

$$\tan^{-1}(z) = \frac{i}{2}\log\left(\frac{i+z}{i-z}\right).$$

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`acot` | `atan2` | `atand` | `tan`

Introduced before R2006a

atan2

Four-quadrant inverse tangent

Syntax

```
P = atan2(Y,X)
```

Description

`P = atan2(Y,X)` returns the four-quadrant inverse tangent on page 1-571 ($\tan^{-1}$) of `Y` and `X`, which must be real. The `atan2` function follows the convention that `atan2(x,x)` returns `0` when `x` is mathematically zero (either `0` or `-0`).

Examples

Find Four-Quadrant Inverse Tangent of a Point

Find the four-quadrant inverse tangent of the point `y = 4`, `x = -3`.

```
atan2(4,-3)
```

```
ans = 2.2143
```

Convert Complex Number to Polar Coordinates

Convert `4 + 3i` into polar coordinates.

```
z = 4 + 3i;
r = abs(z)
```

```
r = 5
```

```
theta = atan2(imag(z),real(z))
```

```
theta = 0.6435
```

The radius `r` and the angle `theta` are the polar coordinate representation of `4 + 3i`.

Alternatively, use `angle` to calculate `theta`.

```
theta = angle(z)
```

```
theta = 0.6435
```

Convert `r` and `theta` back into the original complex number.

```
z = r*exp(i*theta)
```

```
z = 4.0000 + 3.0000i
```

Plot Four-Quadrant Inverse Tangent

Plot `atan2(Y,X)` for `-4<Y<4` and `-4<X<4`.

Define the interval to plot over.

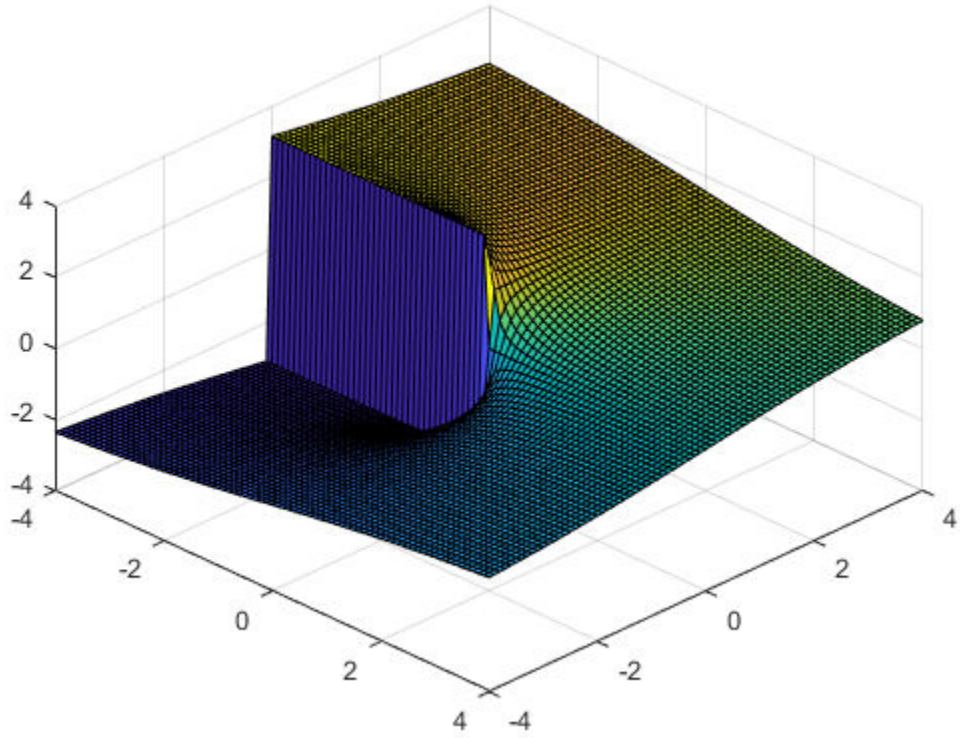
```
[X,Y] = meshgrid(-4:0.1:4,-4:0.1:4);
```

Find `atan2(Y,X)` over the interval.

```
P = atan2(Y,X);
```

Use `surf` to generate a surface plot of the function. Note that `plot` plots the discontinuity that exists at `Y=0` for all `X<0`.

```
surf(X,Y,P);
view(45,45);
```



Input Arguments

Y — *y*-coordinates
scalar | vector | matrix | multidimensional array

y-coordinates, specified as a scalar, vector, matrix, or multidimensional array. Inputs `Y` and `X` must either be the same size or have sizes that are compatible (for example, `Y` is an `M`-by-`N` matrix and `X` is a scalar or `1`-by-`N` row vector). For more information, see "Compatible Array Sizes for Basic Operations".

Data Types: `single` | `double`

X — *x*-coordinates
scalar | vector | matrix | multidimensional array

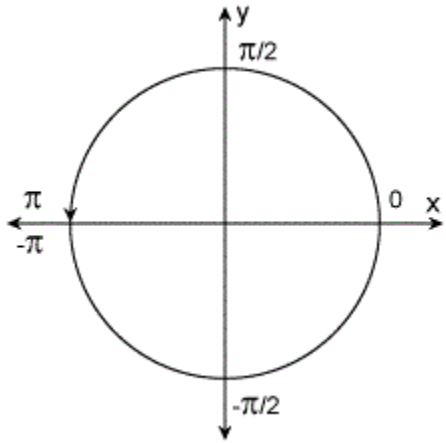
x-coordinates, specified as a scalar, vector, matrix, or multidimensional array. Inputs `Y` and `X` must either be the same size or have sizes that are compatible (for example, `Y` is an `M`-by-`N` matrix and `X` is a scalar or `1`-by-`N` row vector). For more information, see "Compatible Array Sizes for Basic Operations".

Data Types: `single` | `double`

More About

Four-Quadrant Inverse Tangent

The four-quadrant inverse tangent, `atan2(Y,X)`, returns values in the closed interval `[-pi,pi]` based on the values of `Y` and `X`, as shown in the graphic.



In contrast, `atan(Y/X)` returns results that are limited to the interval `[-pi/2,pi/2]`, shown on the right side of the diagram.

IEEE Compliance

For real inputs, `atan2` has a few behaviors that differ from those recommended in the IEEE®-754 Standard.

	MATLAB	IEEE
`atan2(0,-0)`	`0`	`pi`
`atan2(-0,-0)`	`0`	`-pi`

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`angle` | `atan` | `atan2d` | `atanh` | `tan`

Introduced before R2006a

atan2d

Four-quadrant inverse tangent in degrees

Syntax

```
D = atan2d(Y,X)
```

Description

`D = atan2d(Y,X)` returns the four-quadrant inverse tangent on page 1-575 ($\tan^{-1}$) of `Y` and `X`, which must be real. The result, `D`, is expressed in degrees.

Examples

Inverse Tangent of Four Points on the Unit Circle

```
x = [1 0 -1 0];
y = [0 1 0 -1];
d = atan2d(y,x)
```

```
d = 1×4

     0    90   180   -90
```

Input Arguments

Y — *y*-coordinates
scalar | vector | matrix | multidimensional array

y-coordinates, specified as a scalar, vector, matrix, or multidimensional array. Inputs `Y` and `X` must either be the same size or have sizes that are compatible (for example, `Y` is an `M`-

by-N matrix and X is a scalar or 1-by-N row vector). For more information, see "Compatible Array Sizes for Basic Operations".

Data Types: `single` | `double`

X — *x*-coordinates
scalar | vector | matrix | multidimensional array

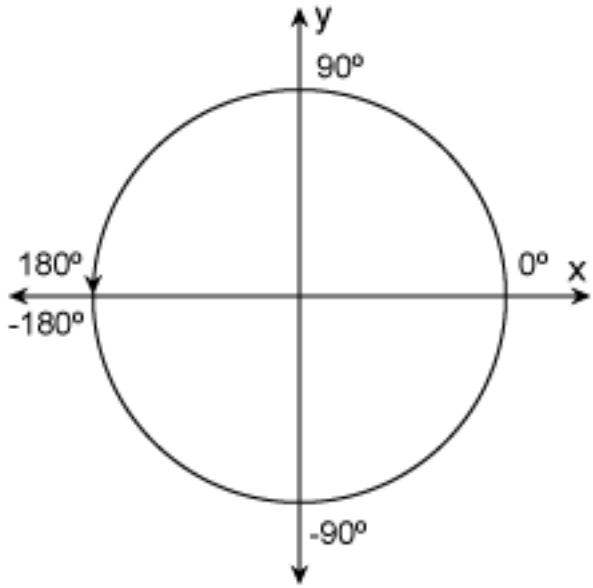
x-coordinates, specified as a scalar, vector, matrix, or multidimensional array. Inputs Y and X must either be the same size or have sizes that are compatible (for example, Y is an M-by-N matrix and X is a scalar or 1-by-N row vector). For more information, see "Compatible Array Sizes for Basic Operations".

Data Types: `single` | `double`

More About

Four-Quadrant Inverse Tangent

The four-quadrant inverse tangent, `atan2d(Y,X)`, returns values in the closed interval `[-180,180]` based on the values of Y and X as shown in the graphic.



In contrast, `atand(Y/X)` returns results that are limited to the interval `[-90,90]`, shown on the right side of the diagram.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`atan` | `atan2` | `atand` | `tan` | `tand`

Introduced in R2012b

atand

Inverse tangent in degrees

Syntax

```
Y = atand(X)
```

Description

`Y = atand(X)` returns the inverse tangent ($\tan^{-1}$) of the elements of `X` in degrees. The function accepts both real and complex inputs.

- For real values of `X`, `atand(X)` returns values in the interval [-90, 90].
- For complex values of `X`, `atand(X)` returns complex values.

Examples

Inverse Tangent of Vector

```
x = [-50 -20 0 20 50];
y = atand(x)
```

```
y = 1×5

  -88.8542  -87.1376         0   87.1376   88.8542
```

The `atand` operation is element-wise when you pass a vector, matrix, or N-D array.

Inverse Tangent of Complex Value

```
atand(10+i)
```

```
ans = 84.3450 + 0.5618i
```

Input Arguments

X — Tangent of angle
scalar value | vector | matrix | N-D array

Tangent of angle, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `atand` operation is element-wise when X is non-scalar.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Angle in degrees
scalar value | vector | matrix | N-D array

Angle in degrees, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as X.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`atan` | `atan2d` | `tan` | `tand`

Introduced before R2006a

atanh

Inverse hyperbolic tangent

Syntax

```
Y = atanh(X)
```

Description

`Y = atanh(X)` returns the inverse hyperbolic tangent of the elements of `X`. The function accepts both real and complex inputs. All angles are in radians.

Examples

Inverse Hyperbolic Tangent of Vector

Find the inverse hyperbolic tangent of the elements of vector `X`. The `atanh` function acts on `X` element-wise.

```
X = [2 -3 1+2i];
Y = atanh(X)
```

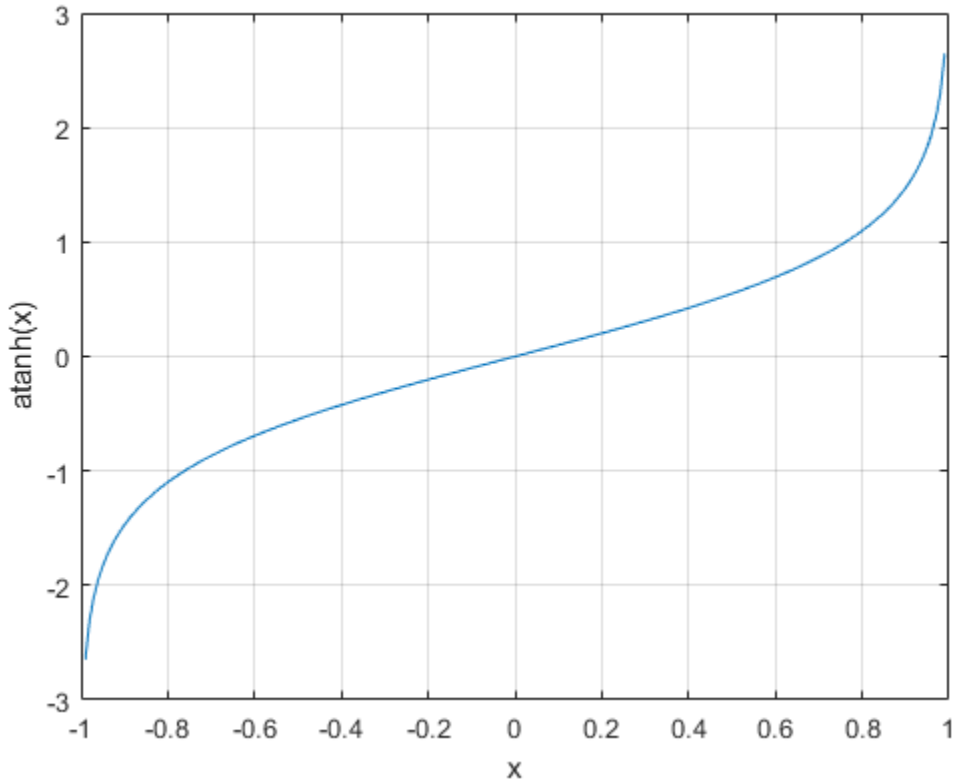
```
Y = 1×3 complex

   0.5493 + 1.5708i  -0.3466 - 1.5708i   0.1733 + 1.1781i
```

Plot the Inverse Hyperbolic Tangent Function

Plot the inverse hyperbolic tangent function over the interval $-1 < x < 1$.

```
x = -0.99:0.01:0.99;
plot(x,atanh(x))
```

```
grid on
xlabel('x')
ylabel('atanh(x)')
```



Input Arguments

X — Hyperbolic tangent of angle
scalar | vector | matrix | multidimensional array

Hyperbolic tangent of angle, specified as a scalar, vector, matrix, or multidimensional array. The `atanh` operation is element-wise when `X` is nonscalar.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Inverse Hyperbolic Tangent

For real values x in the domain $-1 < x < 1$, the inverse hyperbolic tangent satisfies

$$\tanh^{-1}(x) = \frac{1}{2}\log\left(\frac{1+x}{1-x}\right).$$

For complex numbers $z = x + iy$ as well as real values in the regions $-\infty < z < -1$ and $1 < z < \infty$, the call `atanh(z)` returns complex results.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Generates an error during simulation and returns `NaN` in generated code when the input value `x` is real, but the output should be complex. To get the complex result, make the input value complex by passing in `complex(x)`.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- If the output of the function running on the GPU can be complex, then you must explicitly specify its input arguments as complex. For more information, see “Work with Complex Numbers on a GPU” (Parallel Computing Toolbox).

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`acosh` | `asinh` | `atan` | `atan2` | `tanh`

Introduced before R2006a

audiodevinfo

Information about audio device

Syntax

```
info = audiodevinfo
nDevices = audiodevinfo(IO)
name = audiodevinfo(IO,ID)
DriverVersion = audiodevinfo(IO,ID,'DriverVersion')
support = audiodevinfo(IO,ID,Fs,nBits,nChannels)

ID = audiodevinfo(IO,name)
ID = audiodevinfo(IO,Fs,nBits,nChannels)
```

Description

`info = audiodevinfo` returns information about the input and output audio devices on the system.

`nDevices = audiodevinfo(IO)` returns the number of input devices on the system if `IO` is `1`, and returns the number of output devices on the system if `IO` is `0`.

`name = audiodevinfo(IO,ID)` returns the name of the audio device specified by the device identifier, `ID`.

`DriverVersion = audiodevinfo(IO,ID,'DriverVersion')` returns the name of the driver for the audio device specified by `ID`.

`support = audiodevinfo(IO,ID,Fs,nBits,nChannels)` returns `1` if the input or output audio device specified by `ID` supports the sample rate, number of bits, and number of channels specified by the values of `Fs`, `nBits`, and `nChannels`, respectively. Otherwise, `support` is `0`.

`ID = audiodevinfo(IO,name)` returns the device identifier of the input or output audio device identified by the device name, `name`. If no device is found with the specified name, then `audiodevinfo` returns an error.

`ID = audiodevinfo(IO,Fs,nBits,nChannels)` returns the device identifier of the first input or output device that supports the sample rate, number of bits, and the number of channels specified by the values of `Fs`, `nBits`, and `nChannels`, respectively. If no supporting device is found, then `ID` is `-1`.

Examples

View Information About Audio Devices

Call `audiodevinfo` with no inputs to view information about the input and output audio devices on a system. `audiodevinfo` returns a structure containing two fields, `input` and `output`.

```
info = audiodevinfo
```

```
info =

  struct with fields:

     input: [1×2 struct]
    output: [1×4 struct]
```

The input field contains audio device names, driver used, and device identifiers.

```
info.input
```

```
ans =

  1×2 struct array with fields:

    Name
    DriverVersion
    ID
```

Display the information on the first input device.

```
info.input(1)
```

```
ans =

  struct with fields:

             Name: 'Primary Sound Capture Driver  Windows DirectSound '
    DriverVersion: 'Windows DirectSound'
               ID: 0
```

Display the information on the second input device.

```
info.input(2)
```

```
ans =

  struct with fields:

             Name: 'HP 4120 Microphone  HP 4120   Windows DirectSound '
    DriverVersion: 'Windows DirectSound'
               ID: 1
```

The values on your system might differ from this example.

View Number of Output Devices

View the number of output audio devices on the system, using an IO value of 0 to indicate output.

```
nDevices = audiodevinfo(0)
```

```
nDevices =

     4
```

This example shows three output devices, but your system might vary.

Check Support for Input Device

Check if the input audio device identified by the `ID` value, `0`, supports a sample rate of `44100` hertz, with `16` bits per sample, and two channels.

```
support = audiodevinfo(1,0,44100,16,2)
```

```
support =

  logical

   1
```

The input device supports the specified sample rate, number of bits and number of channels. Note that results on your system might vary.

Input Arguments

`IO` — Input or output device
`1` | `0`

Input or output device, specified as `1` to indicate input, or `0` to indicate output.

`ID` — Audio device identifier
integer

Audio device identifier, specified as an integer. The device can be an input or output audio device.

`Fs` — Sample rate
scalar

Sample rate, in hertz, specified as a positive scalar.

Example: `44100`

Data Types: `single` | `double`

`nBits` — Number of bits per sample
scalar

Number of bits per sample, specified as a scalar.

Example: 16

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`nChannels` — Number of audio channels
`1` | `2`

Number of audio channels, specified as 1 (mono) or 2 (stereo).

`name` — Name of input or output device
character vector

Name of the input or output audio device, specified as a character vector.

Data Types: `char`

Output Arguments

`info` — Information about audio devices
structure array

Information about audio devices, returned as a structure array containing two fields, `input` and `output`. Each field is an array of structures, with each structure containing information about one of the audio input or output devices on the system. The individual device structure fields are:

- `Name` — Name of the device, returned as a character vector.
- `DriverVersion` — Name of the driver used to communicate with the device, returned as a character vector.
- `ID` — Device identifier, returned as a scalar.

See Also

`audioplayer` | `audiorecorder`

Introduced before R2006a

audioplayer

Object for playing audio

Description

Use an `audioplayer` object to play audio data. The object contains properties that enable additional flexibility during playback. For example, you can pause, resume, or define callbacks using the `audioplayer` object functions.

Creation

Syntax

```
player = audioplayer(Y,Fs)
player = audioplayer(Y,Fs,nBits)
player = audioplayer(Y,Fs,nBits,ID)

player = audioplayer(recorder)
player = audioplayer(recorder,ID)
```

Description

`player = audioplayer(Y,Fs)` creates an `audioplayer` object for signal `Y`, using sample rate `Fs`. The function returns the audio player object, `player`.

`player = audioplayer(Y,Fs,nBits)` uses `nBits` bits per sample for signal `Y`.

`player = audioplayer(Y,Fs,nBits,ID)` uses the audio device identified by `ID` for output.

`player = audioplayer(recorder)` creates an `audioplayer` object using audio recorder object `recorder`.

`player = audioplayer(recorder,ID)` creates an object from `recorder` that uses the audio device identified by `ID` for output.

Input Arguments

Y — Audio signal

vector | array

Audio signal, specified as a vector or two-dimensional array of numeric data.

The value range of the input sample depends on the data type.

Data Type	Sample Value Range
`int8`	-128 to 127
`uint8`	0 to 255
`int16`	-32768 to 32767
`single`	-1 to 1
`double`	-1 to 1

Data Types: `single` | `double` | `int8` | `int16` | `uint8`

Fs — Sampling frequency

numeric scalar

Sampling frequency in hertz (Hz), specified as a numeric scalar.

Valid values of the sampling rate depend on both the sample rates permitted by MATLAB and the specific audio hardware on your system. MATLAB has a hard restriction of 1000 Hz <= `Fs` <= 384000 Hz, although further hardware-dependent restrictions apply. Typical values supported by most sound cards are 8000, 11025, 22050, 44100, 48000, and 96000 hertz.

Data Types: `single` | `double`

nBits — Bits per sample

16 (default) | 8 | 24

Bits per sample, specified as `8`, `16`, or `24`.

Specify `nBits` only when the signal `Y` contains floating-point values. Valid values of `nBits` depend on the audio hardware. For example, depending on your audio hardware, `nBits` can be one of these values: `8`, `16`, or `24`.

`ID` — Device identifier
`-1` (default) | integer

Device identifier, specified as an integer.

To obtain the ID of a device, use the `audiodevinfo` function.

`recorder` — Audio recorder object
`audiorecorder` object

Audio recorder object, specified as an `audiorecorder` object. Use the `audiorecorder` function to create the object.

Properties

`BitsPerSample` — Bits per sample
positive integer

This property is read-only.

Bits per sample, returned as a positive integer.

`CurrentSample` — Sample currently playing
positive integer

This property is read-only.

Sample currently playing on the audio output device, returned as a positive integer.

If the device is not playing, then `CurrentSample` is the next sample to play using the `play` or `resume` methods.

`DeviceID` — Audio device identifier
integer

This property is read-only.

Audio device identifier, returned as an integer.

`NumChannels` — Number of audio channels
`1` | `2`

This property is read-only.

Number of audio channels, returned as `1` or `2`.

`Running` — Audio player status
`on` | `off`

This property is read-only.

Audio player status, returned as `on` or `off`.

`SampleRate` — Sampling frequency
numeric scalar

Sampling frequency in hertz (Hz), returned as a numeric scalar.

To set the `SampleRate`, use the `Fs` input argument when constructing the `audioplayer` object.

`TotalSamples` — Total length of audio data
integer

This property is read-only.

Total length of the audio data in samples, returned as an integer.

`Tag` — Label
character vector

Label, specified as a character vector.

`Type` — Object class name
`'audioplayer'`

This property is read-only.

Object class name, returned as `'audioplayer'`.

`UserData` — User-defined data
`[ ]` (default) | any data type

User-defined data, specified as a value of any data type. Use this property to store any additional data with the object.

`StartFcn` — Function to execute at start
character vector | string scalar | function handle

Function to execute at start of playback, specified as a character vector or string scalar containing the name of the function, or a function handle.

The first two inputs to your callback function must be the `audioplayer` object and an *`event`* structure. For more information, see callback functions.

`StopFcn` — Function to execute at end
character vector | string scalar | function handle

Function to execute at the end of playback, specified as a character vector or string scalar containing the name of the function, or a function handle.

The first two inputs to your callback function must be the `audioplayer` object and an *`event`* structure. For more information, see callback functions.

`TimerFcn` — Function to execute repeatedly
character vector | string scalar | function handle

Function to execute repeatedly during playback, specified as a character vector or string scalar containing the name of the function, or a function handle. To specify time intervals for the repetitions, use the `TimerPeriod` property.

The first two inputs to your callback function must be the `audioplayer` object and an *`event`* structure. For more information, see callback functions.

`TimerPeriod` — Timer period
0.05 (default) | numeric scalar

Timer period, specified as numeric scalar.

Timer period is the time in seconds between `TimerFcn` callbacks.

Object Functions

get	Query property values for audioplayer object

isplaying	Determine if playback is in progress
pause	Pause playback or recording
play	Play audio from audioplayer object
playblocking	Play audio from audioplayer object, hold control until playback completes
resume	Resume playback or recording from paused state
set	Set property values for audioplayer object
stop	Stop playback or recording

Examples

Play Audio File

Load and play a sample audio file.

Load `handel.mat` into the workspace. The file contains a sample audio data array `y` and the sampling rate `Fs`.

```
load('handel.mat')
whos y Fs
```

```
  Name          Size             Bytes  Class     Attributes

  Fs            1x1                  8  double
  y         73113x1             584904  double
```

Create an `audioplayer` object to play the file.

```
player = audioplayer(y,Fs);
```

Play the audio object on the default audio device.

```
play(player);
```

Record and Play Audio Sample

Record audio data from a microphone and then play the recorded audio.

Create an `audiorecorder` object with default property values.

```
recObj = audiorecorder;
```

Record a five second sample of your speech with your microphone.

```
disp('Start speaking.')
```

```
Start speaking.
```

```
recordblocking(recObj,5);
disp('End of Recording.');
```

```
End of Recording.
```

Create an audio player object from the recording and then play the recorded sample.

```
playerObj = audioplayer(recObj);
play(playerObj);
```

See Also

`audiodevinfo` | `audiorecorder` | `sound`

Topics

"Record and Play Audio"
"Supported Video and Audio File Formats"

Introduced before R2006a

audiorecorder

Object for recording audio

Description

Use an `audiorecorder` object to record audio data from an input device such as a microphone for processing in MATLAB. The `audiorecorder` object contains properties that enable additional flexibility during recording. For example, you can pause, resume, or define callbacks using the `audiorecorder` object functions.

Creation

Syntax

```
recorder = audiorecorder
recorder = audiorecorder(Fs,nBits,NumChannels)
recorder = audiorecorder(Fs,nBits,NumChannels,ID)
```

Description

`recorder = audiorecorder` creates and returns an `audiorecorder` object with these properties:

- Sampling frequency `Fs` = 8000 hertz
- Bits per sample `nBits` = 8
- Number of channels `nChannels` = 1

`recorder = audiorecorder(Fs,nBits,NumChannels)` sets the sample rate `Fs` (in hertz), the bits per sample `nBits`, and the number of channels `nChannels`.

`recorder = audiorecorder(Fs,nBits,NumChannels,ID)` sets the audio input device to the device specified by `ID`.

Input Arguments

`Fs` — Sampling frequency

numeric scalar

Sampling frequency in hertz (Hz), specified as a numeric scalar.

Valid values of the sampling rate depend on both the sample rates permitted by MATLAB and the specific audio hardware on your system. MATLAB has a hard restriction of 1000 Hz <= `Fs` <= 384000 Hz, although further hardware-dependent restrictions apply. Typical values supported by most sound cards are 8000, 11025, 22050, 44100, 48000, and 96000 hertz.

Data Types: `single` | `double`

`nBits` — Bits per sample

8 (default) | 16 | 24

Bits per sample, specified as `8`, `16`, or `24`.

Specify `nBits` only when the signal `Y` contains floating-point values. Valid values of `nBits` depend on the audio hardware. For example, depending on your audio hardware, `nBits` can be one of these values: `8`, `16`, or `24`.

`NumChannels` — Number of channels

`1`(mono) (default) | `2` (stereo)

Number of channels, specified as `1` (mono) or `2` (stereo).

`ID` — Device identifier

`-1` (default) | integer

Device identifier, specified as an integer.

To obtain the ID of a device, use the `audiodevinfo` function.

Properties

`BitsPerSample` — Bits per sample

positive integer

This property is read-only.

Bits per sample, returned as a positive integer.

`CurrentSample` — Sample currently recording
positive integer

This property is read-only.

Sample currently recording on the audio input device, returned as a positive integer.

If the device is not recording, `CurrentSample` is the next sample to record using the `record` or `resume` methods.

`DeviceID` — Audio device identifier
integer

This property is read-only.

Audio device identifier, returned as an integer.

`Running` — Audio recorder status
`off` (default) | `on`

This property is read-only.

Audio recorder status, returned as `on` or `off`.

`SampleRate` — Sampling frequency
`8000` (default) | numeric scalar

Sampling frequency in hertz (Hz), returned as a numeric scalar.

To set the `SampleRate`, use the `Fs` input argument when constructing the `audiorecorder` object.

`TotalSamples` — Total length of audio data
integer

This property is read-only.

Total length of the audio data in samples, returned as an integer.

`Tag` — Label
character vector

Label, specified as a character vector.

Type — Object class name
`'audiorecorder'`

This property is read-only.

Object class name, returned as `'audiorecorder'`.

`UserData` — User-defined data
`[]` (default) | any data type

User-defined data, specified as a value of any data type. Use this property to store any additional data with the object.

`StartFcn` — Function to execute at start
character vector | string scalar | function handle

Function to execute at start of recording, specified as a character vector or string scalar containing the name of the function, or a function handle.

The first two inputs to your callback function must be the `audiorecorder` object and an *`event`* structure. For more information, see callback functions.

`StopFcn` — Function to execute at end
character vector | string scalar | function handle

Function to execute at end of recording, specified as a character vector or string scalar containing the name of the function, or a function handle.

The first two inputs to your callback function must be the `audiorecorder` object and an *`event`* structure. For more information, see callback functions.

`TimerFcn` — Function to execute repeatedly
character vector | string scalar | function handle

Function to execute repeatedly during recording, specified as a character vector or string scalar containing the name of the function, or a function handle. To specify time intervals for the repetitions, use the `TimerPeriod` property.

The first two inputs to your callback function must be the `audiorecorder` object and an *`event`* structure. For more information, see callback functions.

`TimerPeriod` — Timer period
0.05 (default) | numeric scalar

Timer period, specified as numeric scalar.

Timer period is the time in seconds between `TimerFcn` callbacks.

Object Functions

get	Query property values for audiorecorder object
getaudiodata	Store recorded audio signal in numeric array
getplayer	Creates associated audioplayer object
isrecording	Determine if recording is in progress
pause	Pause playback or recording
play	Play audio from audiorecorder object
record	Record audio to audiorecorder object
recordblocking	Record audio to audiorecorder object, hold control until recording completes
resume	Resume playback or recording from paused state
set	Set property values for audiorecorder object
stop	Stop playback or recording

Examples

Record Audio from Input Device

Record audio data from a microphone and then play the recorded audio.

Create an `audiorecorder` object with default property values.

```
recObj = audiorecorder;
```

Alternatively, create an `audiorecorder` object with the desired properties. For a CD-quality audio in stereo, define these properties: sampling frequency (`Fs`), number of bits per sample (`nBits`), the number of channels (`nChannels`), and input device identifier (`ID`).

```
Fs = 44100 ;
nBits = 16 ;
```

```
nChannels = 2 ;
ID = -1; % default audio input device
recObj = audiorecorder(Fs,nBits,nChannels,ID);
```

Collect a five second sample of your speech with your microphone.

```
disp('Start speaking.')
```

```
Start speaking.
```

```
recordblocking(recObj,5);
disp('End of Recording.');
```

```
End of Recording.
```

Play back the recording.

```
play(recObj);
```

See Also

`audiodevinfo` | `audioplayer` | `sound`

Topics

"Record and Play Audio"
"Supported Video and Audio File Formats"

Introduced before R2006a

aviinfo

Information about Audio/Video Interleaved (AVI) file

Note `aviinfo` will be removed in a future release. Use `VideoReader` instead.

Syntax

`fileinfo` `= aviinfo(`*`filename`*`)`

Description

`fileinfo` `= aviinfo(`*`filename`*`)` returns a structure whose fields contain information about the AVI file in *`filename`*. Specify *`filename`* as a character vector or string scalar. If *`filename`* does not include an extension, then `.avi` is used. The file must be in the current working directory or in a directory on the MATLAB path.

The set of fields in the *`fileinfo`* structure is shown below.

Field Name	Description
`AudioFormat`	Character vector containing the name of the format used to store the audio data, if audio data is present
`AudioRate`	Integer indicating the sample rate in Hertz of the audio stream, if audio data is present
`Filename`	Character vector specifying the name of the file
`FileModDate`	Character vector containing the modification date of the file
`FileSize`	Integer indicating the size of the file in bytes
`FramesPerSecond`	Integer indicating the desired frames per second
`Height`	Integer indicating the height of the AVI movie in pixels

Field Name	Description
`ImageType`	Character vector indicating the type of image. Either `'truecolor'` for a truecolor (RGB) image, or `'indexed'` for an indexed image.
`NumAudioChannels`	Integer indicating the number of channels in the audio stream, if audio data is present
`NumFrames`	Integer indicating the total number of frames in the movie
`NumColormapEntries`	Integer specifying the number of colormap entries. For a truecolor image, this value is 0 (zero).
`Quality`	Number between 0 and 100 indicating the video quality in the AVI file. Higher quality numbers indicate higher video quality; lower quality numbers indicate lower video quality. This value is not always set in AVI files and therefore can be inaccurate.
`VideoCompression`	Character vector containing the compressor used to compress the AVI file. If the compressor is not Microsoft Video 1, Run Length Encoding (RLE), Cinepak, or Intel® Indeo, `aviinfo` returns the four-character code that identifies the compressor.
`Width`	Integer indicating the width of the AVI movie in pixels

See Also

`VideoReader` | `VideoWriter` | `mmfileinfo`

Introduced before R2006a

axes

Create Cartesian axes

Syntax

```
axes
axes(Name,Value)
axes(parent,Name,Value)
ax = axes( ___ )
```

```
axes(cax)
```

Description

`axes` creates the default Cartesian axes in the current figure and makes it the current axes. Typically, you do not need to create axes before plotting since graphics functions automatically create axes when plotting if they do not exist.

`axes(Name,Value)` modifies the axes appearance or controls the way data displays using one or more name-value pair arguments. For example, `'FontSize',14` sets the font size for the axes text. For a list of properties, see Axes.

`axes(parent,Name,Value)` creates the axes in the figure, panel, or tab specified by `parent`, instead of in the current figure.

`ax = axes( ___ )` returns the `Axes` object created. Use `ax` to query and modify properties of the `Axes` object after it is created. For a list of properties, see Axes.

`axes(cax)` makes the axes or chart specified by `cax` the current axes on page 1-614 and brings the parent figure into focus. This command also makes `cax` the first object listed in the `Children` property of the figure and sets the `CurrentAxes` property of the figure to `cax`.

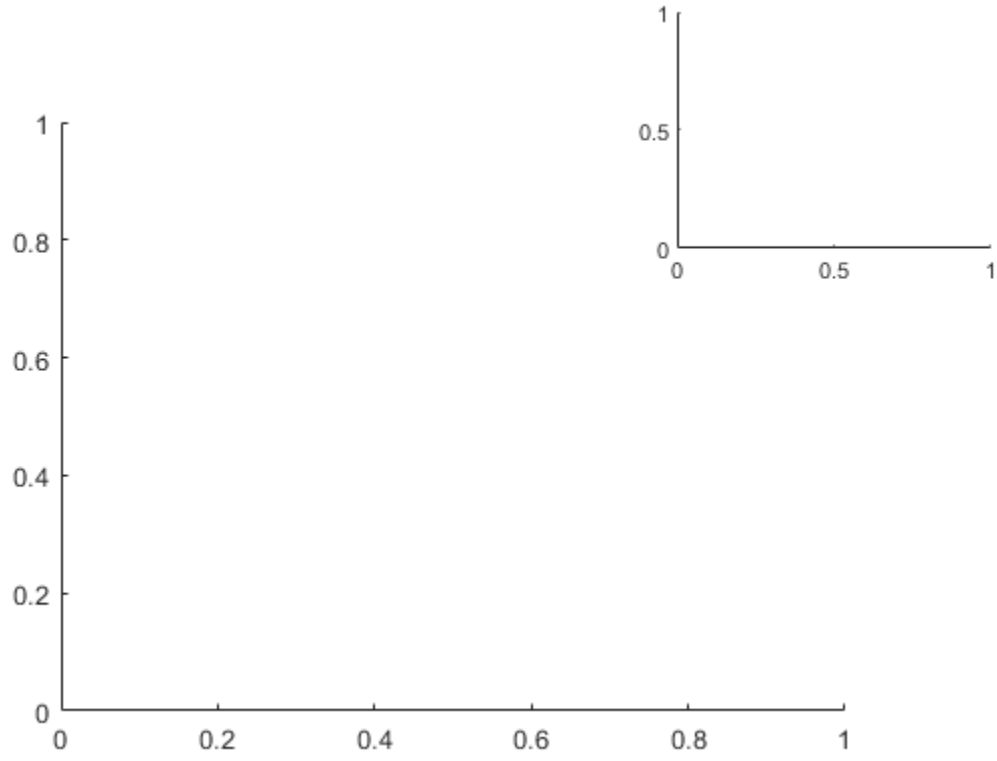
Examples

Position Multiple Axes in Figure

Position two `Axes` objects in a figure and add a plot to each one.

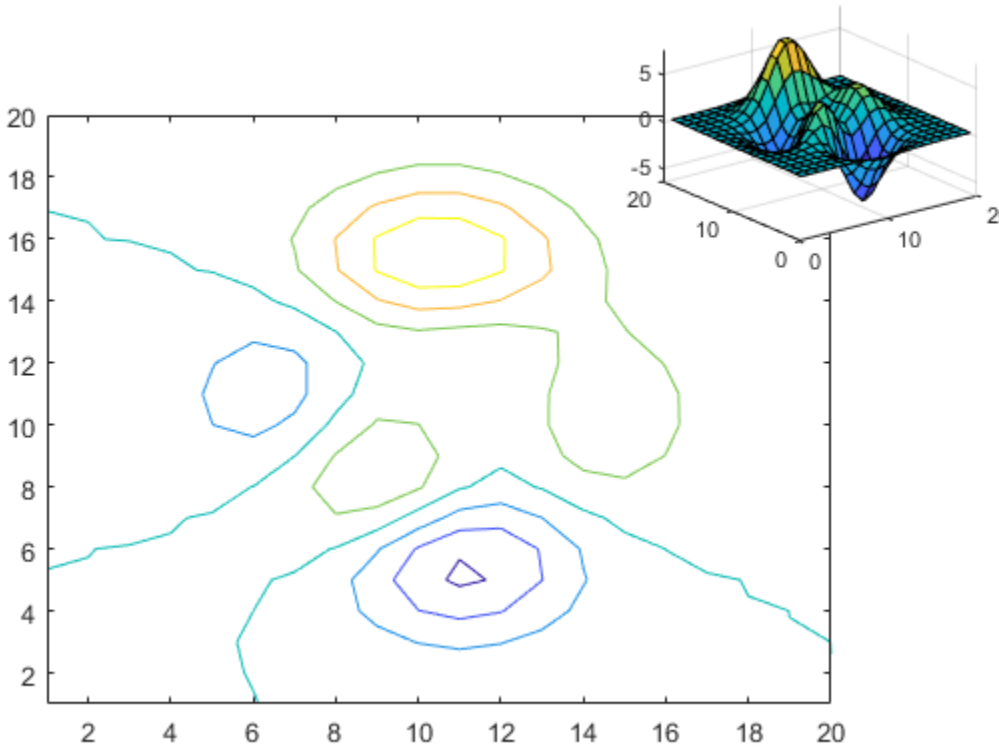
Specify the position of the first `Axes` object so that it has a lower left corner at the point (0.1 0.1) with a width and height of 0.7. Specify the position of the second `Axes` object so that it has a lower left corner at the point (0.65 0.65) with a width and height of 0.28. By default, the values are normalized to the figure. Return the `Axes` objects as `ax1` and `ax2`.

```
figure
ax1 = axes('Position',[0.1 0.1 0.7 0.7]);
ax2 = axes('Position',[0.65 0.65 0.28 0.28]);
```



Add a plot to each `Axes` object. Specify the axes by passing it as the first input argument to the graphics function. Most graphics functions reset some axes properties, such as the tick values and labels. However, they do not reset the axes position.

```
contour(ax1,peaks(20))
surf(ax2,peaks(20))
```

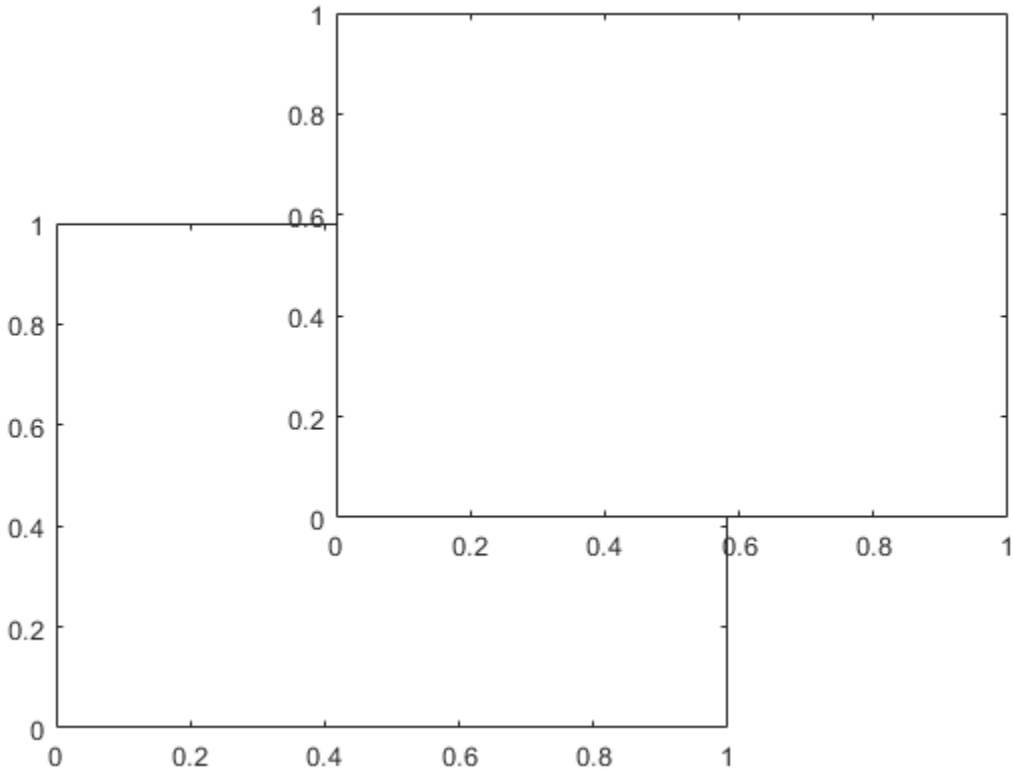


Make Axes the Current Axes

Create two overlayed `Axes` objects. Then, specify the current axes and add a plot.

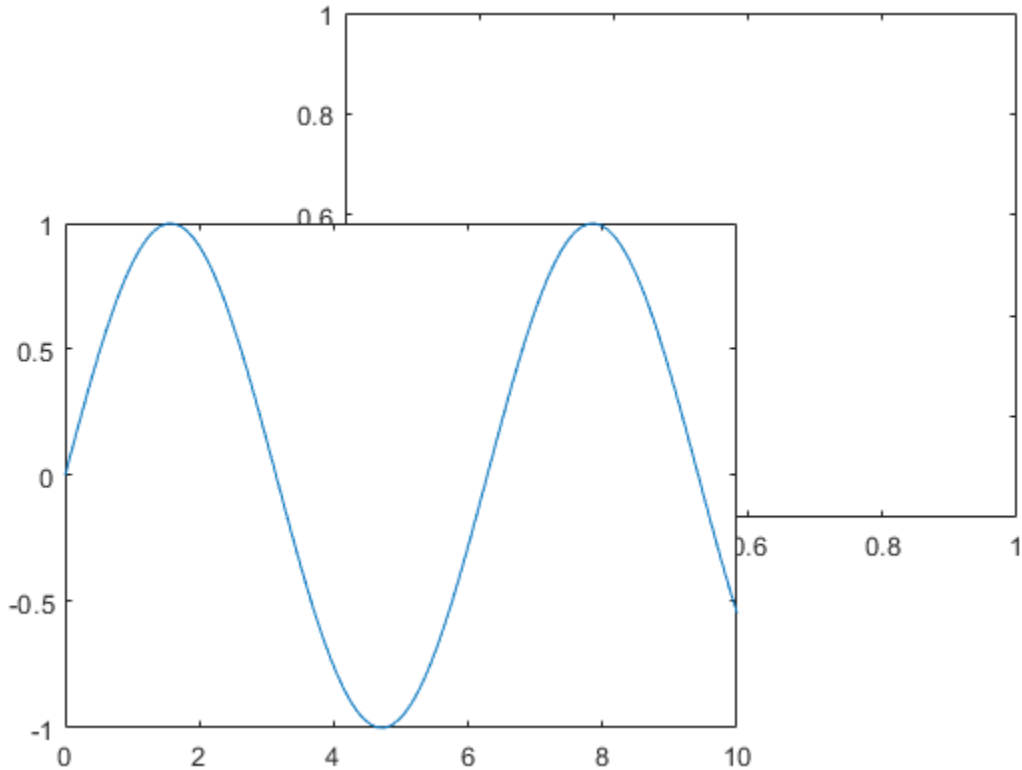
First create two `Axes` objects and specify the positions. Display the box outline around each axes. Return the `Axes` objects as `ax1` and `ax2`.

```
figure
ax1 = axes('Position',[0.1 0.1 .6 .6],'Box','on');
ax2 = axes('Position',[.35 .35 .6 .6],'Box','on');
```



Make `ax1` the current axes. This action brings the axes to the front of the display and makes it the target for subsequent graphics functions. Add a line plot to the axes.

```
axes(ax1)
x = linspace(0,10);
y = sin(x);
plot(x,y)
```

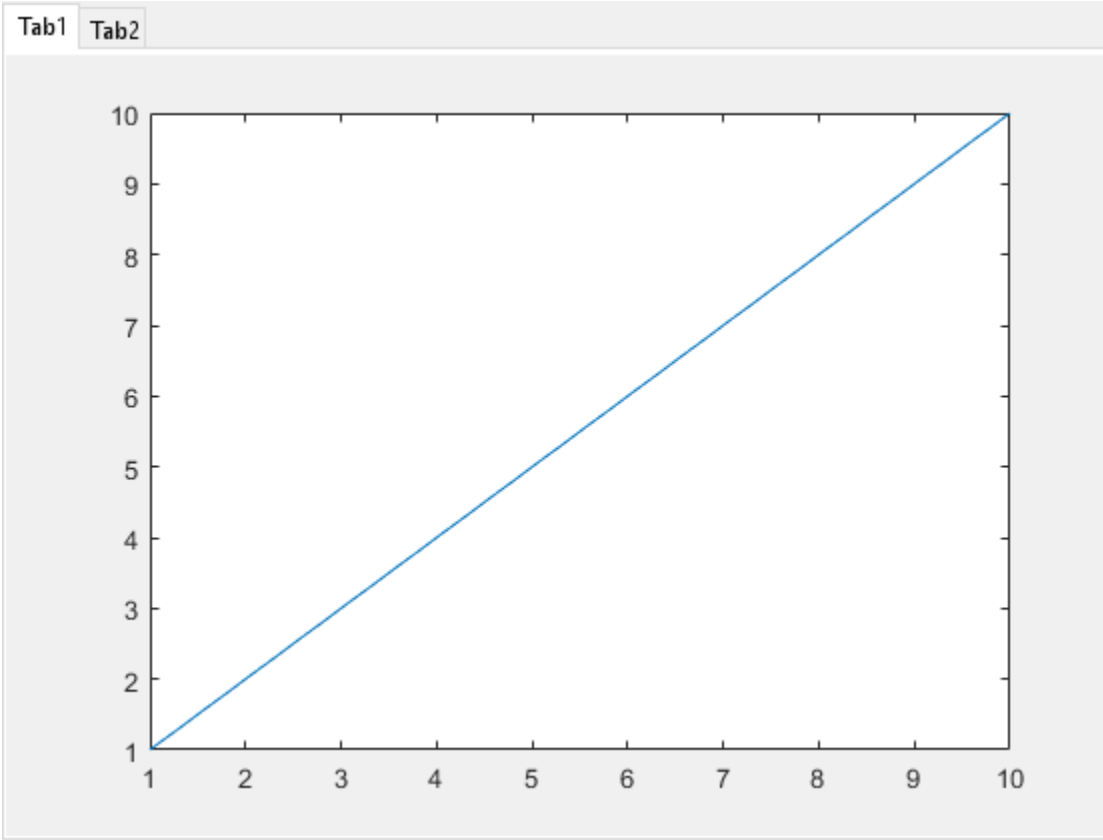


Create Axes in Tabs

Create a figure with two tabs. Add axes to each tab by specifying the parent container for each one. Plot a line in the first tab and a surface in the second tab.

```
figure
tab1 = uitab('Title','Tab1');
ax1 = axes(tab1);
plot(ax1,1:10)

tab2 = uitab('Title','Tab2');
```

```
ax2 = axes(tab2);
surf(ax2,peaks)
```



Input Arguments

parent — Parent container
`Figure` object | `Panel` object | `Tab` object | `TiledChartLayout` object

Parent container, specified as a `Figure`, `Panel`, `Tab`, or `TiledChartLayout` object.

cax — Axes to make current
`Axes` object | `PolarAxes` object | `GeographicAxes` object | graphics object

Axes to make current, specified as an `Axes` object, a `PolarAxes` object, a `GeographicAxes` object, or a graphics object such as a `HeatmapChart`.

If you want to make an object the current axes without changing the state of the figure, set the `CurrentAxes` property of the figure containing the axes; for example:

```
fig = gcf;
fig.CurrentAxes = cax;
```

This approach is useful if you want a figure to remain minimized or stacked below other figures, but want to specify the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments as `Name1,Value1,...,NameN,ValueN`.

Example: `axes('Position',[.3 .3 .5 .5])` sets the position.

Some graphics functions change axes property values when plotting, such as the axis limits or tick values. Set axes properties after plotting.

Note The properties listed here are only a subset. For a full list, see Axes.

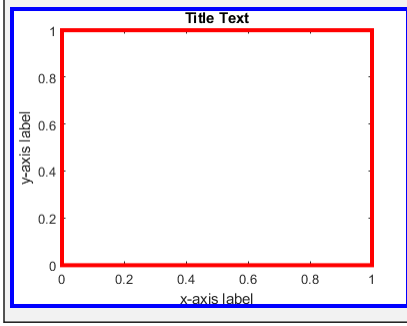
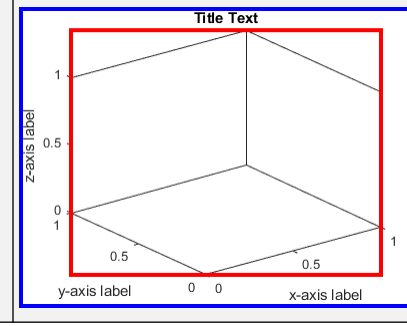
`Position` — Size and location, excluding margin for labels

`[0.1300 0.1100 0.7750 0.8150]` (default) | four-element vector of form `[left bottom width height]`

Size and location, excluding a margin for the labels, specified as a four-element vector of the form `[left bottom width height]`. By default, MATLAB measures the values in units normalized to the container. To change the units, set the `Units` property.

- The `left` and `bottom` elements define the distance from the lower left corner of the container (typically a figure, panel, or tab) to the lower left corner of the position boundary.
- The `width` and `height` elements are the position boundary dimensions. For axes in a 3-D view, the `Position` property is the smallest rectangle that encloses the axes.

If you want to specify the position and account for the text around the axes, then set the `OuterPosition` property instead. These figures show the areas defined by the `OuterPosition` values (blue) and the `Position` values (red).

2-D View of Axes	3-D View of Axes
	

For more information on the axes position, see “Control Axes Layout”.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

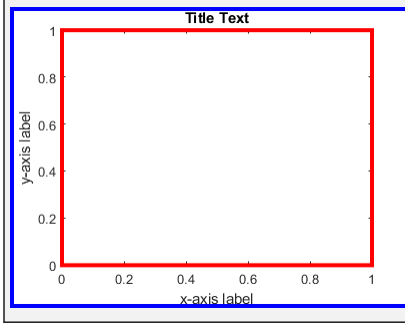
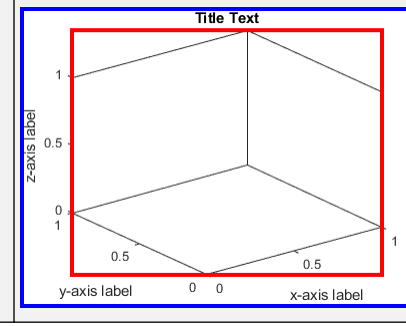
`OuterPosition` — Size and location, including labels and margin

`[0 0 1 1]` (default) | four-element vector of the form `[left bottom width height]`

Size and location, including the labels and a margin, specified as a four-element vector of the form `[left bottom width height]`. By default, MATLAB measures the values in units normalized to the container. To change the units, set the `Units` property. The default value of `[0 0 1 1]` includes the whole interior of the container.

- The `left` and `bottom` elements define the distance from the lower left corner of the container (typically a figure, panel, or tab) to the lower left corner of the outer position boundary.
- The `width` and `height` elements are the outer position boundary dimensions.

These figures show the areas defined by the `OuterPosition` values (blue) and the `Position` values (red).

2-D View of Axes	3-D View of Axes
	

For more information on the axes position, see “Control Axes Layout”.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`Units` — Position units

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'points'` | `'pixels'` | `'characters'`

Position units, specified as one of these values.

Units	Description
`'normalized'` (default)	Normalized with respect to the container, which is typically the figure or a panel. The lower left corner of the container maps to `(0,0)` and the upper right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.

Units	Description
`'characters'`	Based on the default uicontrol font of the graphics root object: • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.
`'points'`	Typography points. One point equals 1/72 inch.
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems. • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. • On Linux systems, the size of a pixel is determined by your system resolution.

When specifying the units as a `Name,Value` pair during object creation, you must set the `Units` property before specifying the properties that you want to use these units, such as `Position`.

More About

Current Axes

The current axes is the target for graphics output. By default, graphics functions such as `plot`, `text`, and `surf` draw their results in the current axes. The current axes object is typically the last `Axes` object created or the last one you clicked with the mouse. Changing the current figure also changes the current axes. The `gca` command returns the current axes.

See Also

Properties
Axes

Functions
`axis` | `cla` | `clf` | `figure` | `gca` | `nexttile` | `polaraxes` | `tiledlayout`

Topics
"Control Ratio of Axis Lengths and Data Unit Lengths"
"Control Axes Layout"

Introduced before R2006a

Axes Properties

Axes appearance and behavior

Description

`Axes` properties control the appearance and behavior of an `Axes` object. By changing property values, you can modify certain aspects of the axes.

Starting in R2014b, you can use dot notation to query and set properties.

```
ax = gca;
c = ax.Color;
ax.Color = 'blue';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Font

`FontName` — Font name
supported font name | `'FixedWidth'`

Font name, specified as a supported font name or `'FixedWidth'`. To display and print text properly, you must choose a font that your system supports. The default font depends on your operating system and locale.

To use a fixed-width font that looks good in any locale, use `'FixedWidth'`. The fixed-width font relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`FontWeight` — Character thickness
`'normal'` (default) | `'bold'`

Character thickness, specified as `'normal'` or `'bold'`.

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.

`FontSize` — Font size

scalar numeric value

Font size, specified as a scalar numeric value. The font size affects the title, axis labels, and tick labels. It also affects any legends or colorbars associated with the axes. The default font size depends on the specific operating system and locale. By default, the font size is measured in points. To change the units, set the `FontUnits` property.

MATLAB automatically scales some of the text to a percentage of the axes font size.

- Titles and axis labels — 110% of the axes font size by default. To control the scaling, use the `TitleFontSizeMultiplier` and `LabelFontSizeMultiplier` properties.
- Legends and colorbars — 90% of the axes font size by default. To specify a different font size, set the `FontSize` property for the `Legend` or `Colorbar` object instead.

Example: `ax.FontSize = 12`

`FontSizeMode` — Selection mode for font size

`'auto'` (default) | `'manual'`

Selection mode for the font size, specified as one of these values:

- `'auto'` — Font size specified by MATLAB. If you resize the axes to be smaller than the default size, the font size might scale down to improve readability and layout.
- `'manual'` — Font size specified manually. Do not scale the font size as the axes size changes. To specify the font size, set the `FontSize` property.

`FontAngle` — Character slant

`'normal'` (default) | `'italic'`

Character slant, specified as `'normal'` or `'italic'`.

Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.

`LabelFontSizeMultiplier` — Scale factor for label font size

`1.1` (default) | numeric value greater than 0

Scale factor for the label font size, specified as a numeric value greater than 0. The scale factor is applied to the value of the `FontSize` property to determine the font size for the *x*-axis, *y*-axis, and *z*-axis labels.

Example: `ax.LabelFontSizeMultiplier = 1.5`

`TitleFontSizeMultiplier` — Scale factor for title font size
`1.1` (default) | numeric value greater than 0

Scale factor for the title font size, specified as a numeric value greater than 0. The scale factor is applied to the value of the `FontSize` property to determine the font size for the title.

Example: `ax.TitleFontSizeMultiplier = 1.75`

`TitleFontWeight` — Title character thickness
`'bold'` (default) | `'normal'`

Title character thickness, specified as one of these values:

- `'bold'` — Thicker characters outlines than normal
- `'normal'` — Default weight as defined by the particular font

Example: `ax.TitleFontWeight = 'normal'`

`FontUnits` — Font size units
`'points'` (default) | `'inches'` | `'centimeters'` | `'normalized'` | `'pixels'`

Font size units, specified as one of these values.

Units	Description
`'points'`	Points. One point equals 1/72 inch.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'normalized'`	Interpret font size as a fraction of the axes height. If you resize the axes, the font size modifies accordingly. For example, if the `FontSize` is `0.1` in normalized units, then the text is 1/10 of the height value stored in the axes `Position` property.

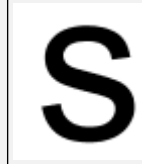
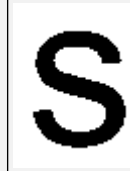
Units	Description
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems. • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. • On Linux systems, the size of a pixel is determined by your system resolution.

To set both the font size and the font units in a single function call, you first must set the `FontUnits` property so that the `Axes` object correctly interprets the specified font size.

`FontSmoothing` — Character smoothing

`'on'` (default) | `'off'`

Character smoothing, specified as `'on'` or `'off'`.

Value	Description	Result
`'on'`	Use antialiasing to make text appear smoother on the screen. **Example:** `ax.FontSmoothing = 'on'`	
`'off'`	Do not use antialiasing. Use this setting if the text seems blurry. **Example:** `ax.FontSmoothing = 'off'`	

Ticks

XTick, YTick, ZTick — Tick values

[] (default) | vector of increasing values

Tick values, specified as a vector of increasing values. If you do not want tick marks along the axis, then specify an empty vector `[]`. The tick values are the locations along the axis where the tick marks appear. The tick labels are the labels that you see next to each tick mark. Use the `XTickLabels`, `YTickLabels`, and `ZTickLabels` properties to specify the associated labels.

Example: `ax.XTick = [2 4 6 8 10]`

Example: `ax.YTick = 0:10:100`

Alternatively, use the `xticks`, `yticks`, and `zticks` functions to specify the tick values. For an example, see "Specify Axis Tick Values and Labels".

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

XTickMode, YTickMode, ZTickMode — Selection mode for tick values

`'auto'` (default) | `'manual'`

Selection mode for the tick values, specified as one of these values:

- `'auto'` — Automatically select the tick values based on the range of data for the axis.
- `'manual'` — Manually specify the tick values. To specify the values, set the `XTick`, `YTick`, or `ZTick` property.

Example: `ax.XTickMode = 'auto'`

XTickLabel, YTickLabel, ZTickLabel — Tick labels

`''` (default) | cell array of character vectors | string array | categorical array

Tick labels, specified as a cell array of character vectors, string array, or categorical array. If you do not want tick labels to show, then specify an empty cell array `{}`. If you do not specify enough labels for all the ticks values, then the labels repeat.

Tick labels support TeX and LaTeX markup. See the `TickLabelInterpreter` property for more information.

If you specify this property as a categorical array, MATLAB uses the values in the array, not the categories.

As an alternative to setting this property, you can use the `xticklabels`, `yticklabels`, and `zticklabels` functions. For an example, see “Specify Axis Tick Values and Labels”.

Example: `ax.XTickLabel = {'Jan','Feb','Mar','Apr'}`

XTickLabelMode, YTickLabelMode, ZTickLabelMode — Selection mode for tick labels

`'auto'` (default) | `'manual'`

Selection mode for the tick labels, specified as one of these values:

- `'auto'` — Automatically select the tick labels.
- `'manual'` — Manually specify the tick labels. To specify the labels, set the `XTickLabel`, `YTickLabel`, or `ZTickLabel` property.

Example: `ax.XTickLabelMode = 'auto'`

TickLabelInterpreter — Tick label interpretation

`'tex'` (default) | `'latex'` | `'none'` | x

Tick label interpretation, specified as one of these values:

- `'tex'` — Interpret labels using a subset of TeX markup.
- `'latex'` — Interpret labels using a subset of LaTeX markup.
- `'none'` — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the text type and color, and include special characters in the text.

The table that follows lists the supported modifiers when the `TickLabelInterpreter` property is set to `'tex'`, which is the default value. Modifiers remain in effect until the end of the text, except for superscripts and subscripts which only modify the next character or the text within the curly braces `{}`.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`

Modifier	Description	Example
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (rarely available)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{specifier}`	Set `specifier` as the name of a font family to change the font style. You can use this with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{specifier}`	Set `specifier` as a scalar numeric value to change the font size.	`'\fontsize{15} text'`
`\color{specifier}`	Set `specifer` as one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Set `specifier` as a three-element RGB triplet to change the font color.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	β	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\delta	δ	\Gamma	Γ	\heartsuit	♥
\epsilon	ϵ	\Delta	Δ	\spadesuit	♠
\zeta	ζ	\Theta	Θ	\leftrightarrow	↔
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	µ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `TickLabelInterpreter` property to `'latex'`. Use dollar symbols around the text, for example, use `'$\int_1^{20} x^2 dx$'` for inline mode or `'$$\int_1^{20} x^2 dx$$'` for display mode.

The displayed text uses the default LaTeX font style. To change the font style, use LaTeX markup within the text. The `FontName`, `FontWeight`, and `FontAngle` properties have no effect.

The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters. For multiline text, this limit reduces by about 10 characters per line. For more information about the LaTeX system, see The LaTeX Project website at www.latex-project.org.

`XTickLabelRotation`, `YTickLabelRotation`, `ZTickLabelRotation` — Tick label rotation

`0` (default) | numeric value in degrees

Tick label rotation, specified as a numeric value in degrees. Positive values give counterclockwise rotation. Negative values give clockwise rotation.

Example: `ax.XTickLabelRotation = 45`

Example: `ax.YTickLabelRotation = 90`

Alternatively, use the `xtickangle`, `ytickangle`, and `ztickangle` functions.

`XMinorTick`, `YMinorTick`, `ZMinorTick` — Minor tick marks

`'off'` | `'on'`

Minor tick marks, specified as one of these values:

- `'off'` — Do not display minor tick marks. This value is the default for an axis with a linear scale.

- `'on'` — Display minor tick marks between the major tick marks on the axis. The space between the major tick marks determines the number of minor tick marks. This value is the default for an axis with a log scale.

Example: `ax.XMinorTick = 'on'`

`TickDir` — Tick mark direction
`'in'` (default) | `'out'` | `'both'`

Tick mark direction, specified as one of these values:

- `'in'` — Direct the tick marks inward from the axis lines. (Default for 2-D views)
- `'out'` — Direct the tick marks outward from the axis lines. (Default for 3-D views)
- `'both'` — Center the tick marks over the axis lines.

Example: `ax.TickDir = 'out'`

`TickDirMode` — Selection mode for `TickDir`
`'auto'` (default) | `'manual'`

Selection mode for the `TickDir` property, specified as one of these values:

- `'auto'` — Automatically select the tick direction based on the current view.
- `'manual'` — Manually specify the tick direction. To specify the tick direction, set the `TickDir` property.

Example: `ax.TickDirMode = 'auto'`

`TickLength` — Tick mark length
`[0.01 0.025]` (default) | two-element vector

Tick mark length, specified as a two-element vector of the form `[2Dlength 3Dlength]`. The first element is the tick mark length in 2-D views and the second element is the tick mark length in 3-D views. Specify the values in units normalized relative to the longest of the visible *x*-axis, *y*-axis, or *z*-axis lines.

Example: `ax.TickLength = [0.02 0.035]`

Rulers

`XLim`, `YLim`, `ZLim` — Minimum and maximum axis limits
`[0 1]` (default) | two-element vector of the form `[min max]`

Minimum and maximum limits, specified as a two-element vector of the form `[min max]`, where `max` is greater than `min`. You can specify the limits as numeric, categorical, datetime, or duration values. However, the type of values that you specify must match the type of values along the axis.

You can specify both limits or you can specify one limit and let the axes automatically calculate the other. For an automatically calculated minimum or maximum limit, use `-inf` or `inf`, respectively.

Example: `ax.XLim = [0 10]`

Example: `ax.YLim = [-inf 10]`

Example: `ax.ZLim = [0 inf]`

Alternatively, use the `xlim`, `ylim`, and `zlim` functions to set the limits. For an example, see "Specify Axis Limits".

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `datetime` | `duration`

`XLimMode`, `YLimMode`, `ZLimMode` — Selection mode for axis limits
`'auto'` (default) | `'manual'`

Selection mode for the axis limits, specified as one of these values:

- `'auto'` — Automatically select the axis limits based on the data plotted, which is, the total span of the `XData`, `YData`, or `ZData` of all the objects displayed in the axes.
- `'manual'` — Manually specify the axis limits. To specify the axis limits, set the `XLim`, `YLim`, or `ZLim` property.

Example: `ax.XLimMode = 'auto'`

`XAxis`, `YAxis`, `ZAxis` — Axis ruler
ruler object

Axis ruler, returned as a ruler object. The ruler controls the appearance and behavior of the *x*-axis, *y*-axis, or *z*-axis. Modify the appearance and behavior of a particular axis by accessing the associated ruler and setting ruler properties. The type of ruler that MATLAB creates for each axis depends on the plotted data. For a list of ruler properties that `Axes` objects support, see:

- NumericRuler

- DatetimeRuler
- DurationRuler
- CategoricalRuler

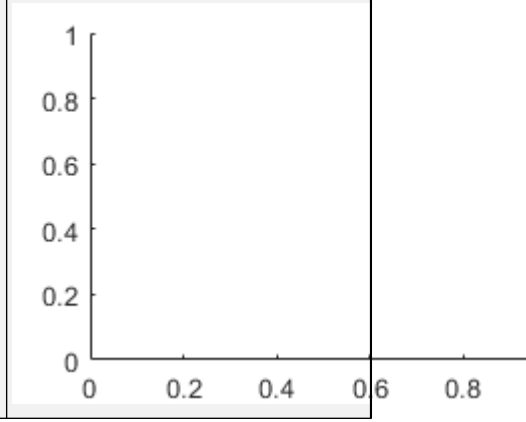
For example, access the ruler for the *x*-axis through the `XAxis` property. Then, change the `Color` property of the ruler, and thus the color of the *x*-axis, to red. Similarly, change the color of the *y*-axis to green.

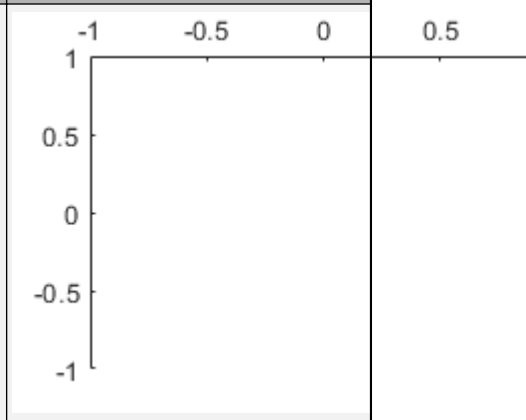
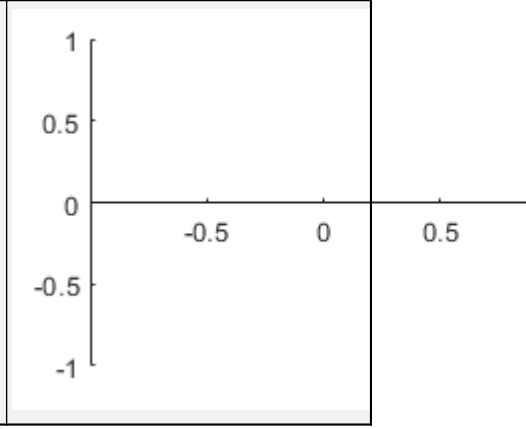
```
ax = gca;
ax.XAxis.Color = 'r';
ax.YAxis.Color = 'g';
```

If the `Axes` object has two *y*-axes, then the `YAxis` property stores two ruler objects.

`XAxisLocation` — *x*-axis location
`'bottom'` (default) | `'top'` | `'origin'`

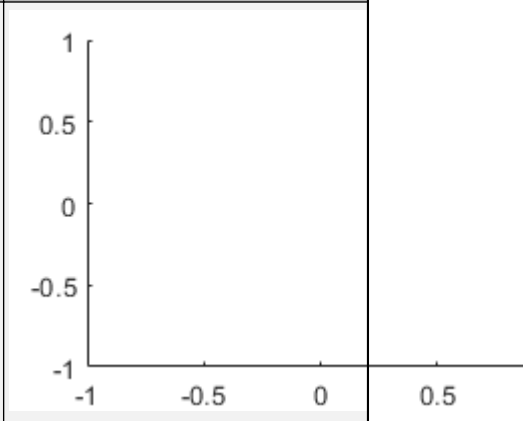
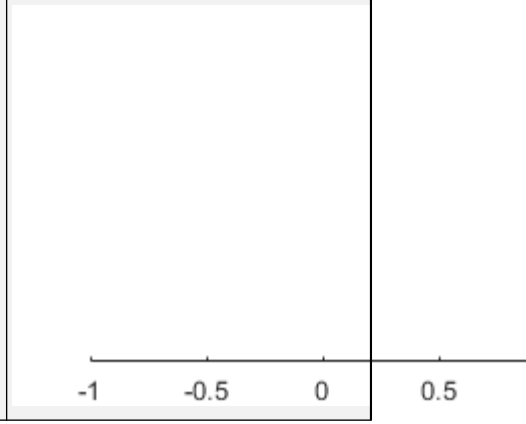
x-axis location, specified as one of the values in this table. This property applies only to 2-D views.

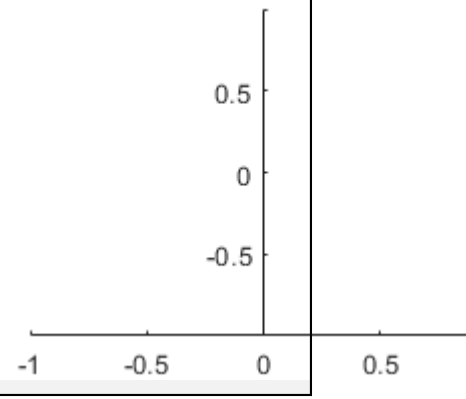
Value	Description	Result
`'bottom'`	Bottom of the axes. **Example:** `ax.XAxisLocation = 'bottom'`	

Value	Description	Result
`'top'`	Top of the axes. **Example:** `ax.XAxisLocation = 'top'`	
`'origin'`	Through the origin point (0,0). **Example:** `ax.XAxisLocation = 'origin'`	

`YAxisLocation` — *y*-axis location
`'left'` (default) | `'right'` | `'origin'`

y-axis location, specified as one of the values in this table. This property applies only to 2-D views.

Value	Description	Result
`'left'`	Left side of the axes. **Example:** `ax.YAxisLocation = 'left'`	
`'right'`	Right side of the axes. **Example:** `ax.YAxisLocation = 'right'`	

Value	Description	Result
`'origin'`	Through the origin point (0,0). **Example:** `ax.YAxisLocation = 'origin'`	

`XColor`, `YColor`, `ZColor` — Color of axis line, tick values, and labels
`[0.15 0.15 0.15]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of the axis line, tick values, and labels in the *x*, *y*, or *z* direction, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The color you specify also affects the grid lines, unless you specify the grid line color using the `GridColor` or `MinorGridColor` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `ax.XColor = [1 1 0]`

Example: `ax.YColor = 'y'`

Example: `ax.ZColor = 'yellow'`

Example: `ax.ZColor = '#FFFF00'`

`XColorMode` — Property for setting *x*-axis grid color

`'auto'` (default) | `'manual'`

Property for setting the *x*-axis grid color, specified as `'auto'` or `'manual'`. The mode value only affects the *x*-axis grid color. The *x*-axis line, tick values, and labels always use the `XColor` value, regardless of the mode.

The *x*-axis grid color depends on both the `XColorMode` property and the `GridColorMode` property, as shown here.

XColorMode	GridColorMode	x-Axis Grid Color
`'auto'`	`'auto'`	`GridColor` property
	`'manual'`	`GridColor` property
`'manual'`	`'auto'`	`XColor` property
	`'manual'`	`GridColor` property

The *x*-axis minor grid color depends on both the `XColorMode` property and the `MinorGridColorMode` property, as shown here.

XColorMode	MinorGridColorMode	x-Axis Minor Grid Color
`'auto'`	`'auto'`	`MinorGridColor` property
	`'manual'`	`MinorGridColor` property
`'manual'`	`'auto'`	`XColor` property
	`'manual'`	`MinorGridColor` property

`YColorMode` — Property for setting *y*-axis grid color
`'auto'` (default) | `'manual'`

Property for setting the *y*-axis grid color, specified as `'auto'` or `'manual'`. The mode value only affects the *y*-axis grid color. The *y*-axis line, tick values, and labels always use the `YColor` value, regardless of the mode.

The *y*-axis grid color depends on both the `YColorMode` property and the `GridColorMode` property, as shown here.

YColorMode	GridColorMode	y-Axis Grid Color
`'auto'`	`'auto'`	`GridColor` property
	`'manual'`	`GridColor` property
`'manual'`	`'auto'`	`YColor` property

YColorMode	GridColorMode	y-Axis Grid Color
	`'manual'`	`GridColor` property

The *y*-axis minor grid color depends on both the `YColorMode` property and the `MinorGridColorMode` property, as shown here.

YColorMode	MinorGridColorMode	y-Axis Minor Grid Color
`'auto'`	`'auto'`	`MinorGridColor` property
	`'manual'`	`MinorGridColor` property
`'manual'`	`'auto'`	`YColor` property
	`'manual'`	`MinorGridColor` property

`ZColorMode` — Property for setting *z*-axis grid color

`'auto'` (default) | `'manual'`

Property for setting the *z*-axis grid color, specified as `'auto'` or `'manual'`. The mode value only affects the *z*-axis grid color. The *z*-axis line, tick values, and labels always use the `ZColor` value, regardless of the mode.

The *z*-axis grid color depends on both the `ZColorMode` property and the `GridColorMode` property, as shown here.

ZColorMode	GridColorMode	z-Axis Grid Color
`'auto'`	`'auto'`	`GridColor` property
	`'manual'`	`GridColor` property
`'manual'`	`'auto'`	`ZColor` property
	`'manual'`	`GridColor` property

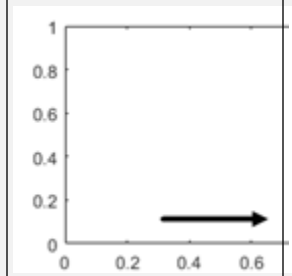
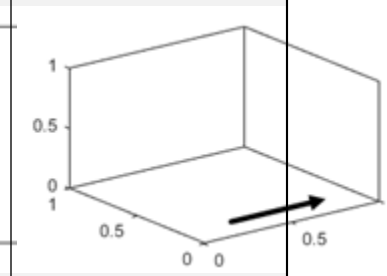
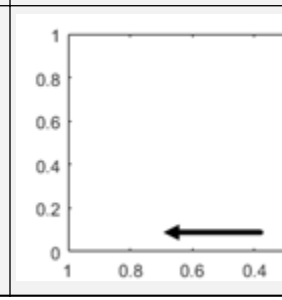
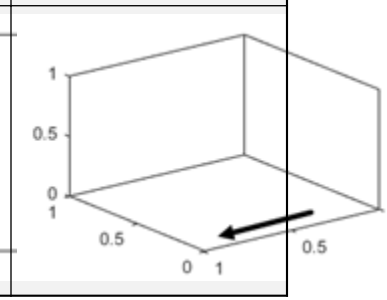
The *z*-axis minor grid color depends on both the `ZColorMode` property and the `MinorGridColorMode` property, as shown here.

ZColorMode	MinorGridColorMode	z-Axis Minor Grid Color
`'auto'`	`'auto'`	`MinorGridColor` property
	`'manual'`	`MinorGridColor` property
`'manual'`	`'auto'`	`ZColor` property

ZColorMode	MinorGridColorMode	z-Axis Minor Grid Color
	`'manual'`	`MinorGridColor` property

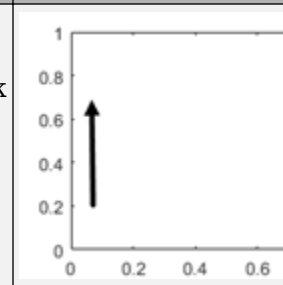
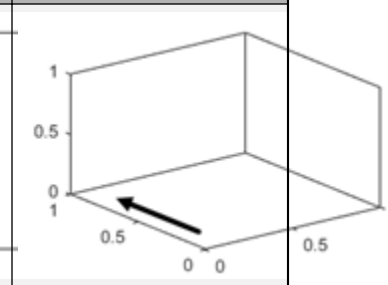
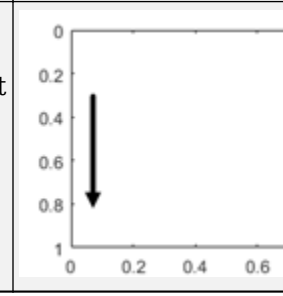
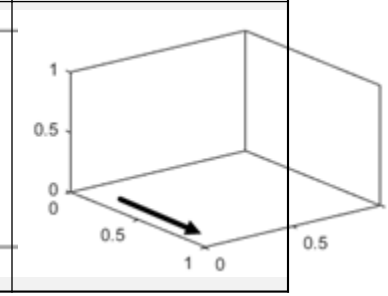
`XDir` — *x*-axis direction
`'normal'` (default) | `'reverse'`

x-axis direction, specified as one of these values.

Value	Description	Result in 2-D	Result in 3-D
`'normal'`	Values increase from left to right. **Example:** `ax.XDir = 'normal'`		
`'reverse'`	Values increase from right to left. **Example:** `ax.XDir = 'reverse'`		

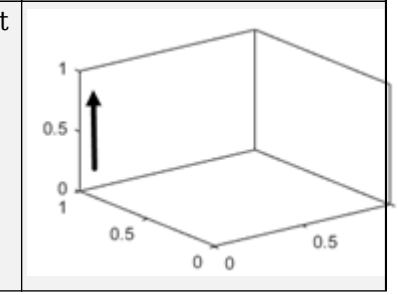
`YDir` — *y*-axis direction
`'normal'` (default) | `'reverse'`

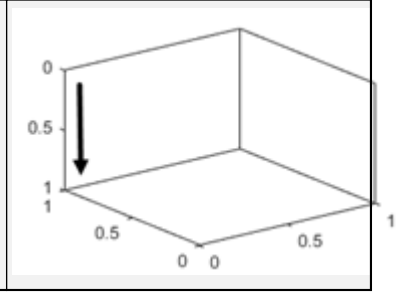
y-axis direction, specified as one of these values.

Value	Description	Result in 2-D	Result in 3-D
`'normal'`	Values increase from bottom to top (2-D view) or front to back (3-D view). **Example:** `ax.YDir = 'normal'`		
`'reverse'`	Values increase from top to bottom (2-D view) or back to front (3-D view). **Example:** `ax.YDir = 'reverse'`		

`ZDir` — *z*-axis direction
`'normal'` (default) | `'reverse'`

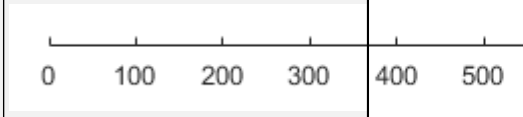
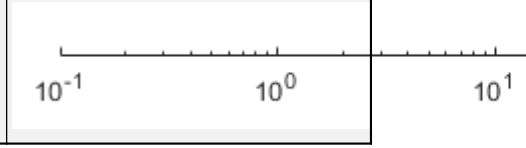
z-axis direction, specified as one of these values.

Value	Description	Result in 3-D
`'normal'`	Values increase pointing out of the screen (2-D view) or from bottom to top (3-D view). **Example:** `ax.ZDir = 'normal'`	

Value	Description	Result in 3-D
`'reverse'`	Values increase pointing into the screen (2-D view) or from top to bottom (3-D view). **Example:** `ax.ZDir = 'reverse'`	

`XScale`, `YScale`, `ZScale` — Scale of values along axis
`'linear'` (default) | `'log'`

Axis scale, specified as one of these values.

Value	Description	Result
`'linear'`	Linear scale **Example:** `ax.XScale = 'linear'`	
`'log'`	Log scale **Example:** `ax.XScale = 'log'`	

Grids

`XGrid`, `YGrid`, `ZGrid` — Grid lines
`'off'` (default) | `'on'`

Grid lines, specified as one of these values:

- `'off'` — Do not display the grid lines.
- `'on'` — Display grid lines perpendicular to the axis; for example, along lines of constant *x*, *y*, or *z* values.

Alternatively, use the `grid on` or `grid off` command to set all three properties to `'on'` or `'off'`, respectively. For more information, see `grid`.

Example: `ax.XGrid = 'on'`

Layer — Placement of grid lines and tick marks

`'bottom'` (default) | `'top'`

Placement of grid lines and tick marks in relation to graphic objects, specified as one of these values:

- `'bottom'` — Display tick marks and grid lines under graphics objects.
- `'top'` — Display tick marks and grid lines over graphics objects.

This property affects only 2-D views.

Example: `ax.Layer = 'top'`

GridLineStyle — Line style for grid lines

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style for grid lines, specified as one of the line styles in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

To display the grid lines, use the `grid on` command or set the `XGrid`, `YGrid`, or `ZGrid` property to `'on'`.

Example: `ax.GridLineStyle = '--'`

GridColor — Color of grid lines

`[0.15 0.15 0.15]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of grid lines, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

To set the colors for the axes box outline, use the `XColor`, `YColor`, and `ZColor` properties.

To display the grid lines, use the `grid on` command or set the `XGrid`, `YGrid`, or `ZGrid` property to `'on'`.

Example: `ax.GridColor = [0 0 1]`

Example: `ax.GridColor = 'b'`

Example: `ax.GridColor = 'blue'`

Example: `ax.GridColor = '#0000FF'`

`GridColorMode` — Property for setting grid color
`'auto'` (default) | `'manual'`

Property for setting the grid color, specified as one of these values:

- `'auto'` — Check the values of the `XColorMode`, `YColorMode`, and `ZColorMode` properties to determine the grid line colors for the *x*, *y*, and *z* directions.
- `'manual'` — Use `GridColor` to set the grid line color for all directions.

`GridAlpha` — Grid-line transparency
`0.15` (default) | value in the range `[0,1]`

Grid-line transparency, specified as a value in the range `[0,1]`. A value of `1` means opaque and a value of `0` means completely transparent.

Example: `ax.GridAlpha = 0.5`

`GridAlphaMode` — Selection mode for `GridAlpha`
`'auto'` (default) | `'manual'`

Selection mode for the `GridAlpha` property, specified as one of these values:

- `'auto'` — Default transparency value of `0.15`.
- `'manual'` — Manually specify the transparency value. To specify the value, set the `GridAlpha` property.

Example: `ax.GridAlphaMode = 'auto'`

XMinorGrid, YMinorGrid, ZMinorGrid — Minor grid lines
`'off'` (default) | `'on'`

Minor grid lines, specified as one of these values:

- `'off'` — Do not display grid lines.
- `'on'` — Display grid lines aligned with the minor tick marks of the axis. You do not need to enable minor ticks to display minor grid lines.

Alternatively, use the `grid minor` command to toggle the visibility of the minor grid lines.

Example: `ax.XMinorGrid = 'on'`

MinorGridLineStyle — Line style for minor grid lines
`':'` (default) | `'-'` | `'--'` | `'-.'` | `'none'`

Line style for minor grid lines, specified as one of the line styles shown in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

To display minor grid lines, use the `grid minor` command or set the `XMinorGrid`, `YMinorGrid`, or `ZMinorGrid` property to `'on'`.

Example: `ax.MinorGridLineStyle = '-.'`

MinorGridColor — Color of minor grid lines
`[0.1 0.1 0.1]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of minor grid lines, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

To display minor grid lines, use the `grid minor` command or set the `XMinorGrid`, `YMinorGrid`, or `ZMinorGrid` property to `'on'`.

Example: `ax.MinorGridColor = [0 0 1]`

Example: `ax.MinorGridColor = 'b'`

Example: `ax.MinorGridColor = 'blue'`

Example: `ax.MinorGridColor = '#0000FF'`

`MinorGridColorMode` — Property for setting minor grid color
`'auto'` (default) | `'manual'`

Property for setting the minor grid color, specified as one of these values:

- `'auto'` — Check the values of the `XColorMode`, `YColorMode`, and `ZColorMode` properties to determine the grid line colors for the *x*, *y*, and *z* directions.
- `'manual'` — Use `MinorGridColor` to set the minor grid line color for all directions.

`MinorGridAlpha` — Minor grid line transparency
`0.25` (default) | value in the range `[0,1]`

Minor grid line transparency, specified as a value in the range `[0,1]`. A value of `1` means opaque and a value of `0` means completely transparent.

Example: `ax.MinorGridAlpha = 0.5`

`MinorGridAlphaMode` — Selection mode for `MinorGridAlpha`
`'auto'` (default) | `'manual'`

Selection mode for the `MinorGridAlpha` property, specified as one of these values:

- `'auto'` — Default transparency value of `0.25`.
- `'manual'` — Manually specify the transparency value. To specify the value, set the `MinorGridAlpha` property.

Example: `ax.MinorGridAlphaMode = 'auto'`

Labels

`Title` — Text object for axes title

text object

Text object for axes title. To add a title, set the `String` property of the text object. To change the title appearance, such as the font style or color, set other properties. For a complete list, see Text.

```
ax = gca;
ax.Title.String = 'My Title';
ax.Title.FontWeight = 'normal';
```

Alternatively, use the `title` function to add a title and control the appearance.

```
title('My Title','FontWeight','normal')
```

Note This text object is not contained in the axes `Children` property, cannot be returned by `findobj`, and does not use default values defined for text objects.

`XLabel`, `YLabel`, `ZLabel` — Text object for axis label

text object

Text object for axis label. To add an axis label, set the `String` property of the text object. To change the label appearance, such as the font size, set other properties. For a complete list, see Text.

```
ax = gca;
ax.YLabel.String = 'My y-Axis Label';
ax.YLabel.FontSize = 12;
```

Alternatively, use the `xlabel`, `ylabel`, and `zlabel` functions to add an axis label and control the appearance.

```
ylabel('My y-Axis Label','FontSize',12)
```

Note These text objects are not contained in the axes `Children` property, cannot be returned by `findobj`, and do not use default values defined for text objects.

Legend — Legend associated with axes

`empty GraphicsPlaceholder` (default) | `Legend` object

This property is read-only.

Legend associated with the `Axes` object, specified as a `Legend` object. To add a legend to the axes, use the `legend` function. Then, you can use this property to modify the legend. For a complete list of properties, see Legend.

```
plot(rand(3))
legend({'Line 1','Line 2','Line 3'},'FontSize',12)
ax = gca;
ax.Legend.TextColor = 'red';
```

You also can use this property to determine if the axes has a legend.

```
ax = gca;
lgd = ax.Legend
if ~isempty(lgd)
    disp('Legend Exists')
end
```

Multiple Plots

`ColorOrder` — Color order

seven predefined colors (default) | three-column matrix of RGB triplets

Color order, specified as a three-column matrix of RGB triplets. This property defines the palette of colors MATLAB uses to create plot objects such as `Line`, `Scatter`, and `Bar` objects. Each row of the array is an RGB triplet. An RGB triplet is a three-element vector whose elements specify the intensities of the red, green, and blue components of a color. The intensities must be in the range [0, 1]. This table lists the default colors.

Colors	ColorOrder Matrix
	[0 0.4470 0.7410 0.8500 0.3250 0.0980 0.9290 0.6940 0.1250 0.4940 0.1840 0.5560 0.4660 0.6740 0.1880 0.3010 0.7450 0.9330 0.6350 0.0780 0.1840]

MATLAB assigns colors to objects according to their order of creation. For example, when plotting lines, the first line uses the first color, the second line uses the second color, and so on. If there are more lines than colors, then the cycle repeats.

Changing the Color Order Before or After Plotting

Starting in R2019b, you can change the color order in either of the following ways:

- Call the `colororder` function to change the color order for all the axes in a figure. The colors of existing plots in the figure update immediately. If you place additional axes into the figure, those axes also use the new color order. If you continue to call plotting commands, those commands also use the new colors.
- Set the `ColorOrder` property on the axes, call the `hold` function to set the axes hold state to `'on'`, and then call the desired plotting functions. This is like calling the `colororder` function, but in this case you are setting the color order for the specific axes, not the entire figure. Setting the `hold` state to `'on'` is necessary to ensure that subsequent plotting commands do not reset the axes to use the default color order.

If you are using R2019a or an earlier release, changing the color order matrix does not affect existing plots. To change colors in an existing plot, you must set the `ColorOrder` property, and then set the axes hold state to `'on'` before calling any plotting functions.

The behavior of this property changed in R2019b. For more information, see:

- “Changing ColorOrder or LineStyleOrder on the axes affects existing plots immediately” on page 1-676
- “Indexing scheme for ColorOrder and LineStyleOrder might change plot colors and line styles” on page 1-676

`ColorOrderIndex` — Color order index

1 (default) | positive integer

Color order index, specified as a positive integer. This property specifies the next color MATLAB selects from the axes `ColorOrder` property when it creates the next plot object such as a `Line`, `Scatter`, or `Bar` object. For example, if the color order index value is `1`, then the next object added to the axes uses the first color in the `ColorOrder` matrix. If the index value exceeds the number of colors in the `ColorOrder` matrix, then the index value modulo of the number of colors in the `ColorOrder` matrix determines the color of the next object.

When the `NextPlot` property of the axes is set to `'add'`, then the color order index value increases every time you add a new plot to the axes. To start again with first color, set the `ColorOrderIndex` property to `1`.

The behavior of this property changed in R2019b. For more information, see “Indexing scheme for ColorOrder and LineStyleOrder might change plot colors and line styles” on page 1-676.

`LineStyleOrder` — Line style order

`'-'` solid line (default) | character vector | cell array of character vectors | string array

Line style order, specified as a character vector, a cell array of character vectors, or a string array. This property lists the line styles that MATLAB uses to display multiple plot lines in the axes. MATLAB assigns styles to lines according to their order of creation. It changes to the next line style only after cycling through all the colors in the `ColorOrder` property with the current line style. The default `LineStyleOrder` has only one line style, `'-'`.

To customize the line style order, create a cell array of character vectors or a string array. Specify each element of the array as a line specifier or marker specifier from the following tables. You can combine a line and a marker specifier into a single element, such as `'-*'`.

Line Specifier	Description
`'-'` (default)	Solid line
`'--'`	Dashed line
`':'`	Dotted line
`'-.'`	Dash-dotted line

Marker Specifier	Description
`'+'`	Plus sign markers
`'o'`	Circle markers
`'*'`	Star markers
`'.'`	Point markers
`'x'`	Cross markers
`'s'`	Square markers

Marker Specifier	Description
`'d'`	Diamond markers
`'^'`	Upward-pointing triangle markers
`'v'`	Downward-pointing triangle markers
`'>'`	Right-pointing triangle markers
`'<'`	Left-pointing triangle markers
`'p'`	Five-pointed star (pentagram) markers
`'h'`	Six-pointed star (hexagram) markers

Changing Line Style Order Before or After Plotting

Starting in R2019b, you can change the line style order before or after plotting into the axes. When you set the `LineStyleOrder` property to a new value, MATLAB updates the styles of any lines that are in the axes. If you continue plotting into the axes, your plotting commands continue using the line styles from the updated list.

If you are using R2019a or an earlier release, you must change the line style order before plotting. Set the value of the `LineStyleOrder` property, and then call the `hold` function to set the axes hold state to `'on'` before calling any plotting functions.

The behavior of this property changed in R2019b. For more information, see:

- “Changing ColorOrder or LineStyleOrder on the axes affects existing plots immediately” on page 1-676
- “Indexing scheme for ColorOrder and LineStyleOrder might change plot colors and line styles” on page 1-676

`LineStyleOrderIndex` — Line style order index

1 (default) | positive integer

Line style order index, specified as a positive integer. This property specifies the next line style MATLAB selects from the axes `LineStyleOrder` property to create the next plot line. For example, if this property is set to `1`, then the next plot line you add to the axes uses the first item in the `LineStyleOrder` property. If the index value exceeds the number of line styles in the `LineStyleOrder` array, then the index value modulo of the number of elements in the `LineStyleOrder` array determines the style of the next line.

When the `NextPlot` property of the axes is set to `'add'`, MATLAB increments the index value after cycling through all the colors in the `ColorOrder` property with the current

line style. To start again with first line style, set the `LineStyleOrderIndex` property to `1`.

The behavior of this property changed in R2019b. For more information, see “Indexing scheme for ColorOrder and LineStyleOrder might change plot colors and line styles” on page 1-676.

`NextPlot` — Properties to reset
`'replace'` (default) | `'add'` | `'replacechildren'` | `'replaceall'`

Properties to reset when adding a new plot to the axes, specified as one of these values:

- `'add'` — Add new plots to the existing axes. Do not delete existing plots or reset axes properties before displaying the new plot.
- `'replacechildren'` — Delete existing plots before displaying the new plot. Reset the `ColorOrderIndex` and `LineStyleOrderIndex` properties to `1`, but do not reset other axes properties. The next plot added to the axes uses the first color and line style based on the `ColorOrder` and `LineStyle` order properties. This value is similar to using `cla` before every new plot.
- `'replace'` — Delete existing plots and reset axes properties, except `Position` and `Units`, to their default values before displaying the new plot.
- `'replaceall'` — Delete existing plots and reset axes properties, except `Position` and `Units`, to their default values before displaying the new plot. This value is similar to using `cla reset` before every new plot.

Note For `Axes` objects with only one *y*-axis, the `'replace'` and `'replaceall'` property values are equivalent. For `Axes` objects with two *y*-axes, the `'replace'` value affects only the active side while the `'replaceall'` value affects both sides.

Figures also have a `NextPlot` property. Alternatively, you can use the `newplot` function to prepare figures and axes for subsequent graphics commands.

`SortMethod` — Order for rendering objects
`'depth'` | `'childorder'`

Order for rendering objects, specified as one of these values:

- `'depth'` — Draw objects in back-to-front order based on the current view. Use this value to ensure that objects in front of other objects are drawn correctly.

- `'childorder'` — Draw objects in the order in which they are created by graphics functions, without considering the relationship of the objects in three dimensions. This value can result in faster rendering, particularly if the figure is very large, but also can result in improper depth sorting of the objects displayed.

Color and Transparency Maps

`Colormap` — Color map

parula (default) | `m`-by-`3` array of RGB triplets

Color map, specified as an `m`-by-`3` array of RGB (red, green, blue) triplets that define `m` individual colors.

Example: `ax.Colormap = [1 0 1; 0 0 1; 1 1 0]` sets the color map to three colors: magenta, blue, and yellow.

MATLAB accesses these colors by their row number.

Alternatively, use the `colormap` function to change the color map.

`ColorScale` — Scale for color mapping

`'linear'` (default) | `'log'`

Scale for color mapping, specified as one of these values:

- `'linear'` — Linear scale. The tick values along the colorbar also use a linear scale.
- `'log'` — Log scale. The tick values along the colorbar also use a log scale.

Example: `ax.ColorScale = 'log'`

`CLim` — Color limits

`[0 1]` (default) | two-element vector of the form `[cmin cmax]`

Color limits for objects in axes that use the colormap, specified as a two-element vector of the form `[cmin cmax]`. This property determines how data values map to the colors in the colormap where:

- `cmin` specifies the data value that maps to the first color in the colormap.
- `cmax` specifies the data value that maps to the last color in the colormap.

The `Axes` object interpolates data values between `cmin` and `cmax` across the colormap. Values outside this range use either the first or last color, whichever is closest.

CLimMode — Selection mode for CLim
'auto' (default) | 'manual'

Selection mode for the CLim property, specified as one of these values:

- 'auto' — Automatically select the limits based on the color data of the graphics objects contained in the axes.
- 'manual' — Manually specify the values. To specify the values, set the CLim property. The values do not change when the limits of the axes children change.

Alphamap — Transparency map
array of 64 values from 0 to 1 (default) | array of finite alpha values from 0 to 1

Transparency map, specified as an array of finite alpha values that progress linearly from 0 to 1. The size of the array can be m-by-1 or 1-by-m. MATLAB accesses alpha values by their index in the array. Alphamaps can be any length.

AlphaScale — Scale for transparency mapping
'linear' (default) | 'log'

Scale for transparency mapping, specified as one of these values:

- 'linear' — Linear scale
- 'log' — Log scale

Example: ax.AlphaScale = 'log'

ALim — Alpha limits
[0 1] (default) | two-element vector of the form [amin amax]

Alpha limits, specified as a two-element vector of the form [amin amax]. This property affects the AlphaData values of graphics objects, such as surface, image, and patch objects. This property determines how the AlphaData values map to the figure alpha map, where:

- amin specifies the data value that maps to the first alpha value in the figure alpha map.
- amax specifies the data value that maps to the last alpha value in the figure alpha map.

The `Axes` object interpolates data values between `amin` and `amax` across the figure alpha map. Values outside this range use either the first or last alpha map value, whichever is closest.

The `Alphamap` property of the figure contains the alpha map. For more information, see the `alpha` function.

`ALimMode` — Selection mode for `ALim`

`'auto'` (default) | `'manual'`

Selection mode for the `ALim` property, specified as one of these values:

- `'auto'` — Automatically select the limits based on the `AlphaData` values of the graphics objects contained in the axes.
- `'manual'` — Manually specify the alpha limits. To specify the alpha limits, set the `ALim` property.

Box Styling

`Color` — Background color

`[1 1 1]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `ax.Color = [0 0 1];`

Example: `ax.Color = 'b';`

Example: `ax.Color = 'blue';`

Example: `ax.Color = '#0000FF';`

`LineWidth` — Line width

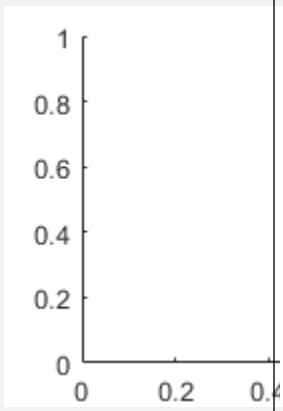
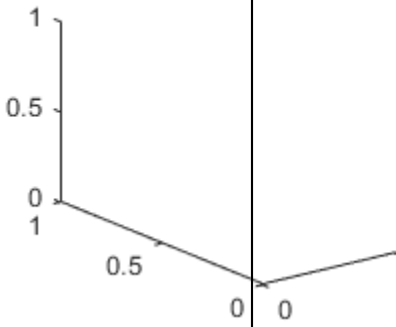
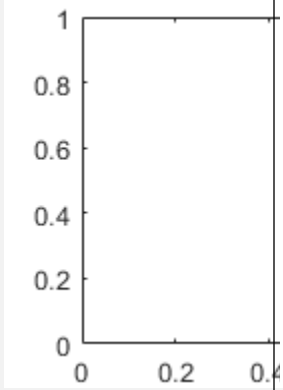
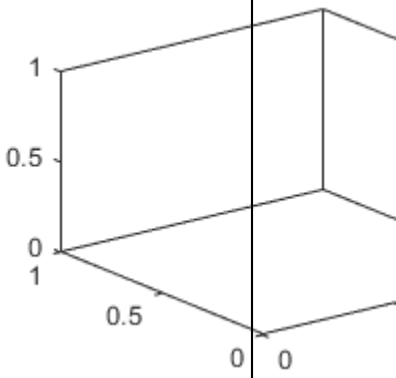
`0.5` (default) | positive numeric value

Line width of axes outline, tick marks, and grid lines, specified as a positive numeric value in point units. One point equals 1/72 inch.

Example: `ax.LineWidth = 1.5`

Box — Box outline
`'off'` (default) | `'on'`

Box outline, specified as `'off'` or `'on'`.

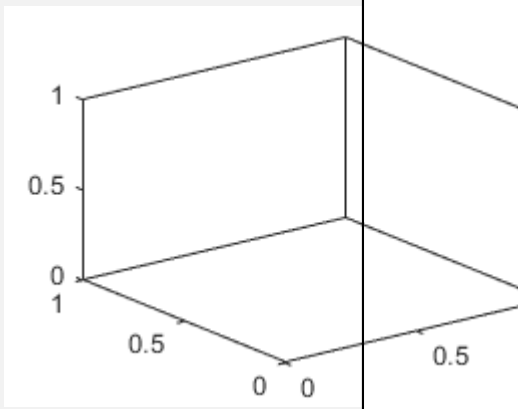
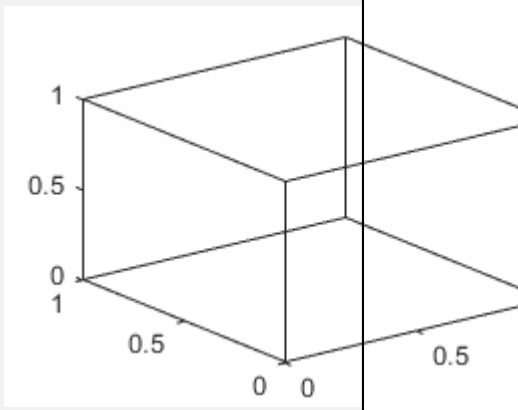
Value	Description	2-D Result	3-D Result
`'off'`	Do not display the box outline around the axes. **Example:** `ax.Box = 'off'`		
`'on'`	Display the box outline around the axes. For 3-D views, use the `BoxStyle` property to change extent of the outline. **Example:** `ax.Box = 'on'`		

The `XColor`, `YColor`, and `ZColor` properties control the color of the outline.

Example: `ax.Box = 'on'`

`BoxStyle` — Box outline style
`'back'` (default) | `'full'`

Box outline style, specified as `'back'` or `'full'`. This property affects only 3-D views.

Value	Description	Result
`'back'`	Outline the back planes of the 3-D box. **Example:** `ax.BoxStyle = 'back'`	
`'full'`	Outline the entire 3-D box. **Example:** `ax.BoxStyle = 'full'`	

`Clipping` — Clipping of objects to axes limits
`'on'` (default) | `'off'`

Clipping of objects to the axes limits, specified as either `'on'` or `'off'`. The clipping behavior of an object within the `Axes` object depends on both the `Clipping` property of the `Axes` object and the `Clipping` property of the individual object. The property value of the `Axes` object has these effects:

- `'on'` — Enable each individual object within the axes to control its own clipping behavior based on the `Clipping` property value for the object.
- `'off'` — Disable clipping for all objects within the axes, regardless of the `Clipping` property value for the individual objects. Parts of objects can appear outside of the axes limits. For example, parts can appear outside the limits if you create a plot, use the `hold on` command, freeze the axis scaling, and then add a plot that is larger than the original plot.

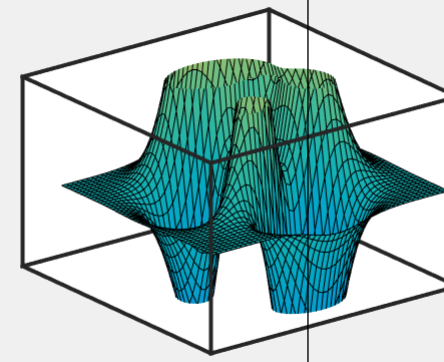
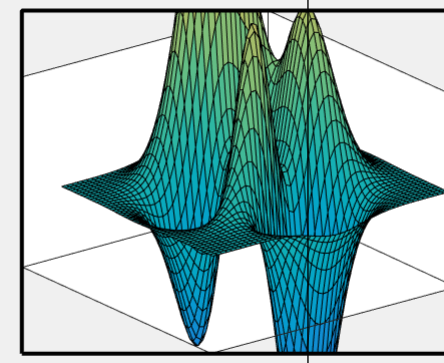
This table lists the results for different combinations of `Clipping` property values.

Clipping Property for Axes Object	Clipping Property for Individual Object	Result
`'on'`	`'on'`	Individual object is clipped. Others might or might not be.
`'on'`	`'off'`	Individual object is not clipped. Others might or might not be.
`'off'`	`'on'`	All objects are unclipped.
`'off'`	`'off'`	All objects are unclipped.

`ClippingStyle` — Clipping boundaries
`'3dbox'` (default) | `'rectangle'`

Clipping boundaries, specified as one of the values in this table. If a plot contains markers, then as long as the data point lies within the axes limits, MATLAB draws the entire marker.

The `ClippingStyle` property has no effect if the `Clipping` property is set to `'off'`.

Value	Descriptions	Illustration of Boundary Region
`'3dbox'`	Clip plotted objects to the six sides of the axes box defined by the axis limits. Thick lines might display outside the axes limits.	
`'rectangle'`	Clip plotted objects to a rectangular boundary enclosing the axes in any given view. Clip thick lines at the axes limits.	

`AmbientLightColor` — Background light color

`[1 1 1]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background light color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The background light is a directionless light that shines uniformly on all objects in the axes. To add light, use the `light` function.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `ax.AmbientLightColor = [1 0 1]`

Example: `ax.AmbientLightColor = 'm'`

Example: `ax.AmbientLightColor = 'magenta'`

Example: `ax.AmbientLightColor = '#FF00FF'`

Position

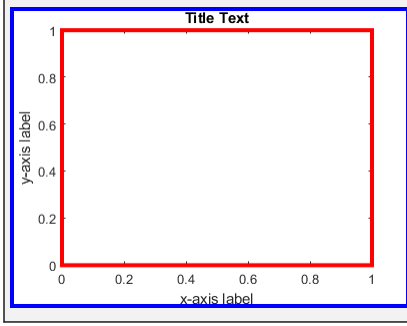
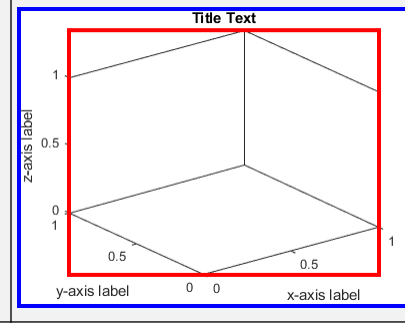
`OuterPosition` — Size and location, including labels and margin

`[0 0 1 1]` (default) | four-element vector of the form `[left bottom width height]`

Size and location, including the labels and a margin, specified as a four-element vector of the form `[left bottom width height]`. By default, MATLAB measures the values in units normalized to the container. To change the units, set the `Units` property. The default value of `[0 0 1 1]` includes the whole interior of the container.

- The `left` and `bottom` elements define the distance from the lower left corner of the container (typically a figure, panel, or tab) to the lower left corner of the outer position boundary.
- The `width` and `height` elements are the outer position boundary dimensions.

These figures show the areas defined by the `OuterPosition` values (blue) and the `Position` values (red).

2-D View of Axes	3-D View of Axes
	

For more information on the axes position, see “Control Axes Layout”.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

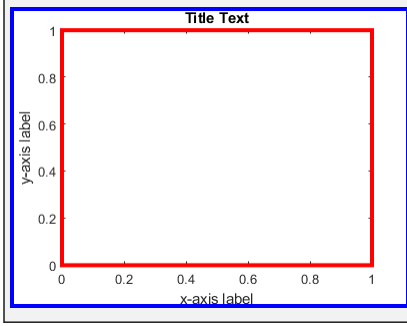
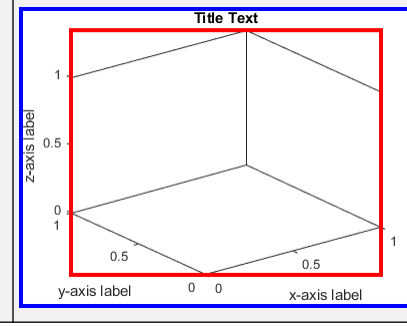
`Position` — Size and location, excluding margin for labels

`[0.1300 0.1100 0.7750 0.8150]` (default) | four-element vector of form `[left bottom width height]`

Size and location, excluding a margin for the labels, specified as a four-element vector of the form `[left bottom width height]`. By default, MATLAB measures the values in units normalized to the container. To change the units, set the `Units` property.

- The `left` and `bottom` elements define the distance from the lower left corner of the container (typically a figure, panel, or tab) to the lower left corner of the position boundary.
- The `width` and `height` elements are the position boundary dimensions. For axes in a 3-D view, the `Position` property is the smallest rectangle that encloses the axes.

If you want to specify the position and account for the text around the axes, then set the `OuterPosition` property instead. These figures show the areas defined by the `OuterPosition` values (blue) and the `Position` values (red).

2-D View of Axes	3-D View of Axes
	

For more information on the axes position, see “Control Axes Layout”.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`TightInset` — Margins for text labels

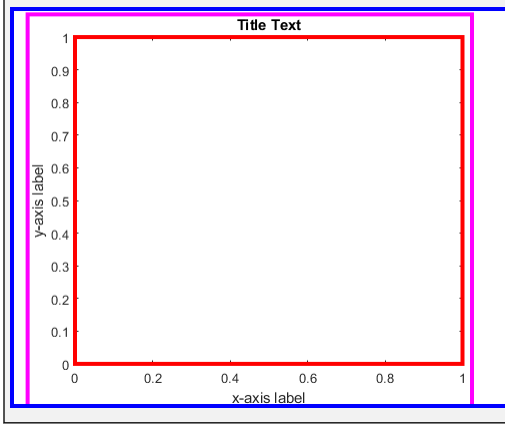
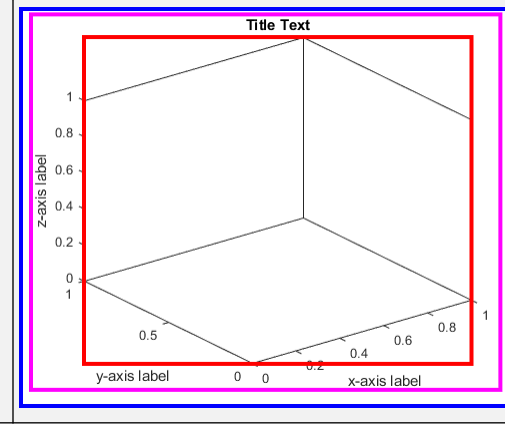
four-element vector of the form `[left bottom right top]`

This property is read-only.

Margins for the text labels, specified as a four-element vector of the form `[left bottom right top]`. By default, MATLAB measures the values in units normalized to the container. To change the units, set the `Units` property.

The elements define the distances between the bounds of the `Position` property and the extent of the surrounding text. The `Position` values combined with the `TightInset` values define the tightest bounding box that encloses the axes and the surrounding text.

These figures show the areas defined by the `OuterPosition` values (blue), the `Position` values (red), and the `Position` expanded by the `TightInset` values (magenta).

2-D View of Axes	3-D View of Axes
	

For more information, see “Control Axes Layout”.

`ActivePositionProperty` — Active position property

`'outerposition'` | `'position'`

Position property to hold constant when adding, removing, or changing decorations, specified as one of the following values:

- `'outerposition'` — The `OuterPosition` property remains constant when you add, remove, or change decorations such as a title or an axis label. If any positional adjustments are needed, MATLAB adjusts the `Position` property.
- `'position'` — The `Position` property remains constant when you add, remove, or change decorations such as a title or an axis label. If any positional adjustments are needed, MATLAB adjusts the `OuterPosition` property.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`Units` — Position units

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'points'` | `'pixels'` | `'characters'`

Position units, specified as one of these values.

Units	Description
`'normalized'` (default)	Normalized with respect to the container, which is typically the figure or a panel. The lower left corner of the container maps to `(0,0)` and the upper right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the default uicontrol font of the graphics root object: • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.
`'points'`	Typography points. One point equals 1/72 inch.

Units	Description
'pixels'	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems. • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. • On Linux systems, the size of a pixel is determined by your system resolution.

When specifying the units as a `Name,Value` pair during object creation, you must set the `Units` property before specifying the properties that you want to use these units, such as `Position`.

`DataAspectRatio` — Relative length of data units
`[1 1 1]` (default) | three-element vector of the form `[dx dy dz]`

Relative length of data units along each axis, specified as a three-element vector of the form `[dx dy dz]`. This vector defines the relative *x*, *y*, and *z* data scale factors. For example, specifying this property as `[1 2 1]` sets the length of one unit of data in the *x*-direction to be the same length as two units of data in the *y*-direction and one unit of data in the *z*-direction.

Alternatively, use the `daspect` function to change the data aspect ratio.

Example: `ax.DataAspectRatio = [1 1 1]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`DataAspectRatioMode` — Data aspect ratio mode
`'auto'` (default) | `'manual'`

Data aspect ratio mode, specified as one of these values:

- `'auto'` — Automatically select values that make best use of the available space. If `PlotBoxAspectRatioMode` and `CameraViewAngleMode` are also set to `'auto'`,

then enable "stretch-to-fill" behavior. Stretch the axes so that it fills the available space as defined by the `Position` property.

- `'manual'` — Disable the "stretch-to-fill" behavior and use the manually specified data aspect ratio. To specify the values, set the `DataAspectRatio` property.

`PlotBoxAspectRatio` — Relative length of each axis
`[1 1 1]` (default) | three-element vector of the form `[px py pz]`

Relative length of each axis, specified as a three-element vector of the form `[px py pz]` defining the relative *x*-axis, *y*-axis, and *z*-axis scale factors. The plot box is a box enclosing the axes data region as defined by the axis limits.

Alternatively, use the `pbaspect` function to change the data aspect ratio.

If you specify the axis limits, data aspect ratio, and plot box aspect ratio, then MATLAB ignores the plot box aspect ratio. It adheres to the axis limits and data aspect ratio.

Example: `ax.PlotBoxAspectRatio = [1 0.75 0.75]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`PlotBoxAspectRatioMode` — Selection mode for `PlotBoxAspectRatio`
`'auto'` (default) | `'manual'`

Selection mode for the `PlotBoxAspectRatio` property, specified as one of these values:

- `'auto'` — Automatically select values that make best use of the available space. If `DataAspectRatioMode` and `CameraViewAngleMode` also are set to `'auto'`, then enable "stretch-to-fill" behavior. Stretch the `Axes` object so that it fills the available space as defined by the `Position` property.
- `'manual'` — Disable the "stretch-to-fill" behavior and use the manually specified plot box aspect ratio. To specify the values, set the `PlotBoxAspectRatio` property.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `TiledChartLayoutOptions` object

Layout options, specified as a `TiledChartLayoutOptions` object. This property specifies options when the axes is a child of a tiled chart layout. Use this property to position the axes within the layout by setting the `Tile` and `TileSpan` properties on the `TiledChartLayoutOptions` object.

For example, this code places axes `ax` in the third tile of a tiled chart layout.

```
ax.Layout.Tile = 3;
```

To make the axes span multiple tiles, specify the `TileSpan` property as a two-element vector. For example, this axes spans 2 rows and 3 columns of tiles.

```
ax.Layout.TileSpan = [2 3];
```

If the axes is not a child of a tiled chart layout (for example, if it is a child of a figure or panel), then this property is empty and has no effect.

View

`View` — Azimuth and elevation of view

`[0 90]` (default) | two-element vector of the form `[azimuth elevation]`

Azimuth and elevation of view, specified as a two-element vector of the form `[azimuth elevation]` defined in degree units. Alternatively, use the `view` function to set the view.

Example: `ax.View = [45 45]`

`Projection` — Type of projection onto 2-D screen

`'orthographic'` (default) | `'perspective'`

Type of projection onto a 2-D screen, specified as one of these values:

- `'orthographic'` — Maintain the correct relative dimensions of graphics objects regarding the distance of a given point from the viewer, and draw lines that are parallel in the data parallel on the screen.
- `'perspective'` — Incorporate foreshortening, which enables you to perceive depth in 2-D representations of 3-D objects. Perspective projection does not preserve the relative dimensions of objects. Instead, it displays a distant line segment smaller than a nearer line segment of the same length. Lines that are parallel in the data might not appear parallel on screen.

`CameraPosition` — Camera location

three-element vector of the form `[x y z]`

Camera location, or the viewpoint, specified as a three-element vector of the form `[x y z]`. This vector defines the axes coordinates of the camera location, which is the point from which you view the axes. The camera is oriented along the view axis, which is a straight line that connects the camera position and the camera target. For an illustration, see “Camera Graphics Terminology”.

If the `Projection` property is set to `'perspective'`, then as you change the `CameraPosition` setting, the amount of perspective also changes.

Alternatively, use the `campos` function to set the camera location.

Example: `ax.CameraPosition = [0.5 0.5 9]`

Data Types: `single` | `double`

`CameraPositionMode` — Selection mode for `CameraPosition`
`'auto'` (default) | `'manual'`

Selection mode for the `CameraPosition` property, specified as one of these values:

- `'auto'` — Automatically set `CameraPosition` along the view axis. Calculate the position so that the camera lies a fixed distance from the target along the azimuth and elevation specified by the current view, as returned by the `view` function. Functions like `rotate3d`, `zoom`, and `pan`, change this mode to `'auto'` to perform their actions.
- `'manual'` — Manually specify the value. To specify the value, set the `CameraPosition` property.

`CameraTarget` — Camera target point
three-element vector of the form `[x y z]`

Camera target point, specified as a three-element vector of the form `[x y z]`. This vector defines the axes coordinates of the point. The camera is oriented along the view axis, which is a straight line that connects the camera position and the camera target. For an illustration, see “Camera Graphics Terminology”.

Alternatively, use the `camtarget` function to set the camera target.

Example: `ax.CameraTarget = [0.5 0.5 0.5]`

Data Types: `single` | `double`

`CameraTargetMode` — Selection mode for `CameraTarget`
`'auto'` (default) | `'manual'`

Selection mode for the `CameraTarget` property, specified as one of these values:

- `'auto'` — Position the camera target at the centroid of the axes plot box.
- `'manual'` — Use the manually specified camera target value. To specify a value, set the `CameraTarget` property.

`CameraUpVector` — Vector defining upwards direction
three-element direction vector of the form `[x y z]`

Vector defining upwards direction, specified as a three-element direction vector of the form `[x y z]`. For 2-D views, the default value is `[0 1 0]`. For 3-D views, the default value is `[0 0 1]`. For an illustration, see "Camera Graphics Terminology".

Alternatively, use the `camup` function to set the upwards direction.

Example: `ax.CameraUpVector = [sin(45) cos(45) 1]`

`CameraUpVectorMode` — Selection mode for `CameraUpVector`
`'auto'` (default) | `'manual'`

Selection mode for the `CameraUpVector` property, specified as one of these values:

- `'auto'` — Automatically set the value to `[0 0 1]` for 3-D views so that the positive *z*-direction is up. Set the value to `[0 1 0]` for 2-D views so that the positive *y*-direction is up.
- `'manual'` — Manually specify the vector defining the upwards direction. To specify a value, set the `CameraUpVector` property.

`CameraViewAngle` — Field of view
`6.6086` (default) | scalar angle in range [0,180)

Field of view, specified as a scalar angle greater than 0 and less than or equal to 180. Changing the camera view angle affects the size of graphics objects displayed in the axes, but does not affect the degree of perspective distortion. The greater the angle, the larger the field of view and the smaller objects appear in the scene. For an illustration, see "Camera Graphics Terminology".

Example: `ax.CameraViewAngle = 15`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`CameraViewAngleMode` — Selection mode for `CameraViewAngle`
`'auto'` (default) | `'manual'`

Selection mode for the `CameraViewAngle` property, specified as one of these values:

- `'auto'` — Automatically select the field of view as the minimum angle that captures the entire scene, up to 180 degrees.

- `'manual'` — Manually specify the field of view. To specify a value, set the `CameraViewAngle` property.

Interactivity

`Toolbar` — Data exploration toolbar

`AxesToolbar` object (default)

Data exploration toolbar, which is an `AxesToolbar` object. The toolbar appears at the top-right corner of the axes when you hover over it.

The toolbar buttons depend on the contents of the axes, but typically include zooming, panning, rotating, data tips, data brushing, and restoring the original view. You can customize the toolbar buttons using the `axtoolbar` and `axtoolbarbtn` functions.

If you do not want the toolbar to appear when you hover over the axes, set the `Visible` property of the `AxesToolbar` object to `'off'`.

```
ax = gca;
ax.Toolbar.Visible = 'off';
```

For more information, see AxesToolbar.

`Interactions` — Interactions

array of interaction objects | [ ]

Interactions, specified as an array of interaction objects or an empty array. The interactions you specify are available within your chart through gestures. You do not have to select any axes toolbar buttons to use them. For example, a `panInteraction` object enables dragging to pan within a chart. For a list of interaction objects, see “Control Chart Interactivity”.

The default set of interactions depends on the type of chart you are displaying. You can replace the default set with a new set of interactions, but you cannot access or modify any of the interactions in the default set. For example, this code replaces the default set of interactions with the `panInteraction` and `zoomInteraction` objects.

```
ax = gca;
ax.Interactions = [panInteraction zoomInteraction];
```

To remove all interactions from the axes, set this property to an empty array. To temporarily disable the current set of interactions, call the `disableDefaultInteractivity` function. You can reenable them by calling the `enableDefaultInteractivity` function.

Note

- Setting this property is not supported in the Live Editor.
- Interaction objects are not returned by `findobj` or `findall`, and they are not copied by `copyobj`.

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`CurrentPoint` — Location of mouse pointer

2-by-3 array

Location of mouse pointer, specified as a 2-by-3 array. The `CurrentPoint` property contains the (*x*,*y*,*z*) coordinates of the mouse pointer with respect to the axes. The returned array is of the form:

```
[xfront yfront zfront
 xback  yback  zback]
```

The two points indicate the location of the last mouse click. However, if the figure has a `WindowButtonMotionFcn` callback defined, then the points indicate the last location of the mouse pointer. The figure also has a `CurrentPoint` property.

The values of the current point when using perspective projection can be different from the same point in orthographic projection because the shape of the axes volume can be different.

Orthogonal Projection

When using orthogonal projection, the values depend on whether the click is within the axes or outside the axes.

- If the click is inside the axes, the two points lie on the line that is perpendicular to the plane of the screen and that passes through the pointer. The coordinates are the points where this line intersects the front and back surfaces of the axes volume (which is defined by the axes x, y, and z limits). The first row is the point nearest to the camera position. The second row is the point farthest from the camera position. This is true for both 2-D and 3-D views.
- If the click is outside the axes, but within the figure, then the points lie on a line that passes through the pointer and is perpendicular to the camera target and camera position planes. The first row is the point in the camera position plane. The second row is the point in the plane of the camera target.

Perspective Projection

Clicking outside of the `Axes` object in perspective projection returns the front point as the current camera position. Only the back point updates with the coordinates of a point that lies on a line extending from the camera position through the pointer and intersecting the camera target at that point.

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.

- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.

- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks only when visible. The `Visible` property must be set to `'on'`. The `HitTest` property determines if the `Axes` object responds to the click or if an ancestor does.
- `'all'` — Capture mouse clicks regardless of visibility. The `Visible` property can be set to `'on'` or `'off'`. The `HitTest` property determines if the `Axes` object responds to the click or if an ancestor does.

- `'none'` — Cannot capture mouse clicks. Clicking the `Axes` object passes the click to the object below it in the current view of the figure window, which is typically the axes or the figure. The `HitTest` property has no effect.

If you want an object to be clickable when it is underneath other objects that you do not want to be clickable, then set the `PickableParts` property of the other objects to `'none'` so that the click passes through them.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `Axes` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `Axes` object that has one of these:
  - `HitTest` property set to `'on'`
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

---

Note The `PickableParts` property determines if the `Axes` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container
`Figure` object | `Panel` object | `Tab` object | `TiledChartLayout` object

Parent container, specified as a `Figure`, `Panel`, `Tab`, or `TiledChartLayout` object.

`Children` — Children

empty `GraphicsPlaceholder` array | array of graphics objects

Children, returned as an array of graphics objects. Use this property to view a list of the children or to reorder the children by setting the property to a permutation of itself.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the child graphics object to the `Axes` object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'axes'`

This property is read-only.

Type of graphics object returned as `'axes'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

Compatibility Considerations

Changing ColorOrder or LineStyleOrder on the axes affects existing plots immediately

Behavior changed in R2019b

Starting in R2019b, if you change the axes `ColorOrder` or `LineStyleOrder` properties after plotting into the axes, the colors and line styles in your plot update immediately. In R2019a and previous releases, the new colors and line styles affect only subsequent plots, not the existing plots.

To preserve the original behavior, set the axes `ColorOrderIndex` or `LineStyleOrderIndex` property to any value (such as its current value) before changing the `ColorOrder` or `LineStyleOrder` property.

Indexing scheme for ColorOrder and LineStyleOrder might change plot colors and line styles

Behavior changed in R2019b

Starting in R2019b, there is a new indexing scheme that enables you to change the colors and line styles of existing plots by setting the `ColorOrder` or `LineStyleOrder` properties. MATLAB applies this indexing scheme to all objects that have a `ColorMode`, `FaceColorMode`, `MarkerFaceColorMode`, or `CDataMode`. As a result, your code might produce plots that cycle though the colors and line styles differently than in previous releases.

In R2019a and earlier releases, MATLAB uses a different indexing scheme which does not allow you to change the colors of existing plots.

To preserve the way your plots cycle through colors and line styles, set the axes `ColorOrderIndex` or `LineStyleOrderIndex` property to any value (such as its current value) before plotting into the axes.

See Also

`axes` | `axis` | `box` | `caxis` | `cla` | `gca` | `grid`

Topics

"Access Property Values"
"Graphics Object Properties"

Introduced before R2006a

axis

Set axis limits and aspect ratios

Syntax

```
axis(limits)

axis style
axis mode
axis ydirection
axis visibility

lim = axis
[m,v,d] = axis('state')

___ = axis(ax,___)
```

Description

`axis(limits)` specifies the limits for the current axes. Specify the limits as vector of four, six, or eight elements.

`axis style` uses a predefined style to set the limits and scaling. For example, specify the style as `equal` to use equal data unit lengths along each axis.

`axis mode` sets whether MATLAB automatically chooses the limits or not. Specify the mode as `manual`, `auto`, or one of the semiautomatic options, such as `'auto x'`.

`axis ydirection`, where `ydirection` is `ij`, places the origin at the upper left corner of the axes. The *y* values increase from top to bottom. The default for `ydirection` is `xy`, which places the origin at the lower left corner. The *y* values increase from bottom to top.

`axis visibility`, where `visibility` is `off`, turns off the display of the axes background. Plots in the axes still display. The default for `visibility` is `on`, which displays the axes background.

`lim = axis` returns the *x*-axis and *y*-axis limits for the current axes. For 3-D axes, it also returns the *z*-axis limits. For polar axes, it returns the *theta*-axis and *r*-axis limits.

`[m,v,d] = axis('state')` returns the current settings for the axis limit selection, the axes visibility, and the *y*-axis direction. This syntax will be removed in a future release. Use the `XLimMode`, `YLimMode`, `ZLimMode`, `Visible`, and `YDir` properties of the axes to get the values instead.

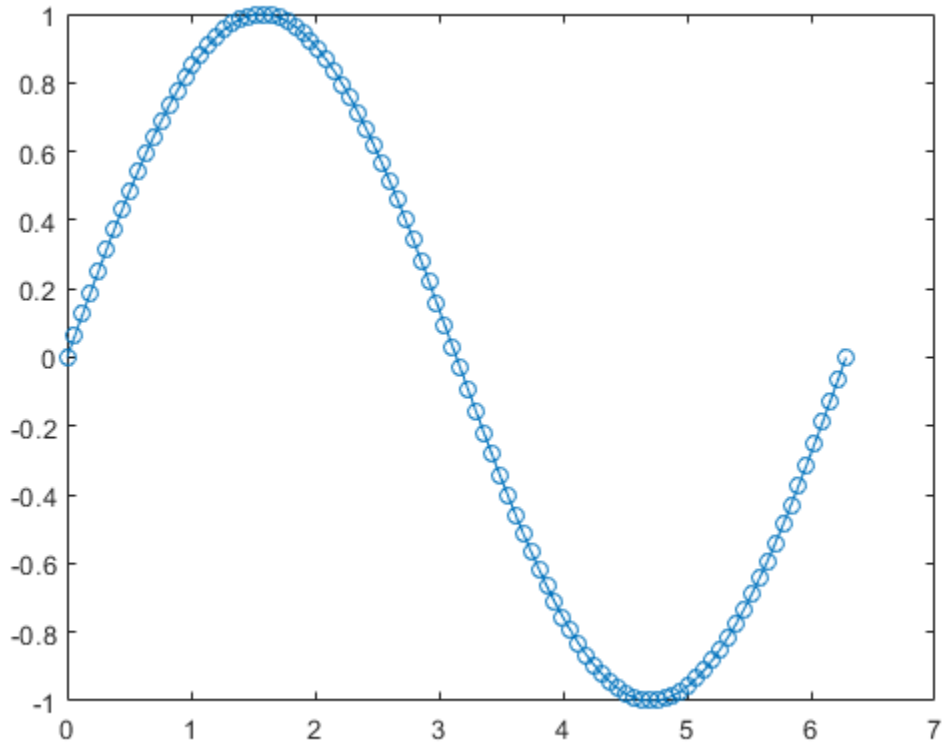
`___ = axis(ax, ___)` uses the axes or polar axes specified by `ax` instead of the current axes. Specify `ax` as the first input argument for any of the previous syntaxes. Use single quotes around input arguments that are character vectors, such as `axis(ax,'equal')`.

Examples

Set Axis Limits

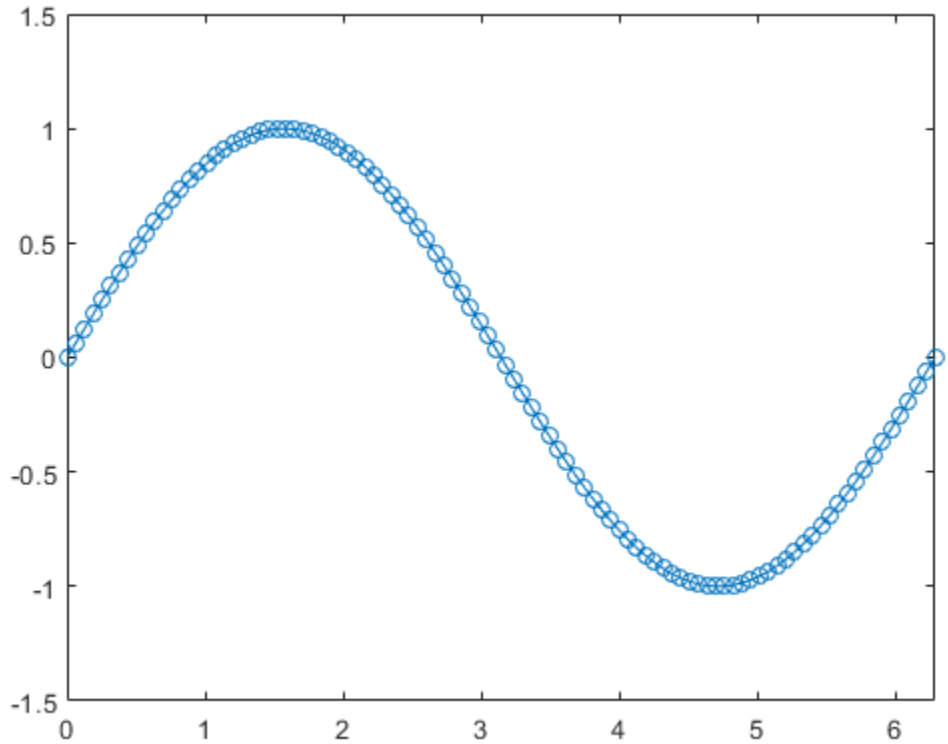
Plot the sine function.

```
x = linspace(0,2*pi);
y = sin(x);
plot(x,y,'-o')
```



Change the axis limits so that the *x*-axis ranges from 0 to 2π and the *y*-axis ranges from -1.5 to 1.5.

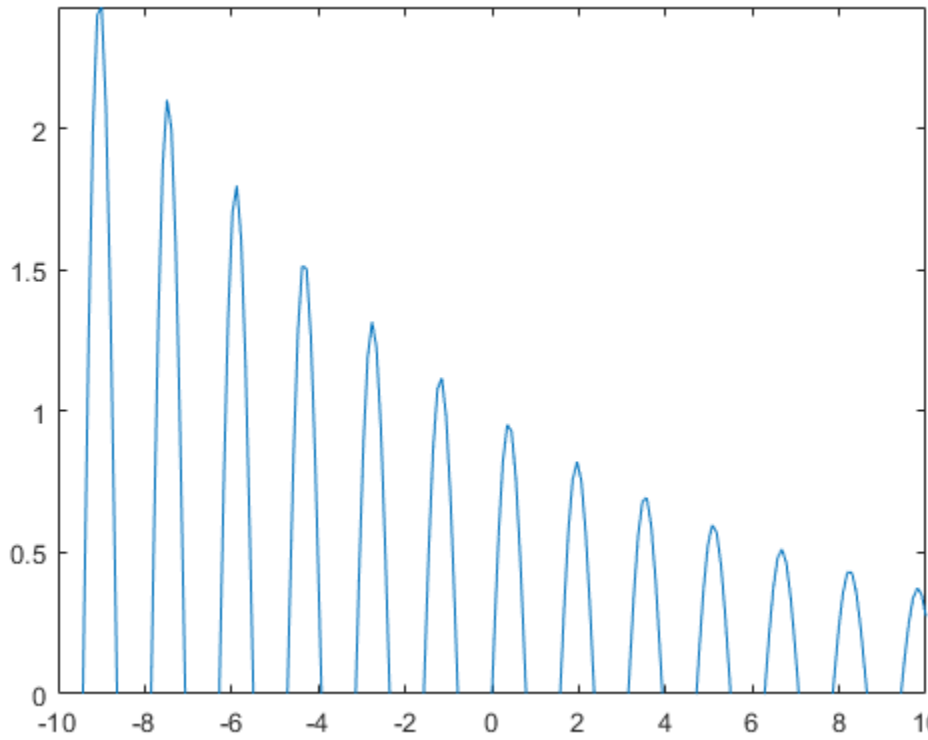
```
axis([0 2*pi -1.5 1.5])
```



Use Semiautomatic Axis Limits

Create a plot. Set the limits for the *x*-axis and set the minimum *y*-axis limit. Use an automatically calculated value for the maximum *y*-axis limit.

```
x = linspace(-10,10,200);
y = sin(4*x)./exp(.1*x);
plot(x,y)
axis([-10 10 0 inf])
```

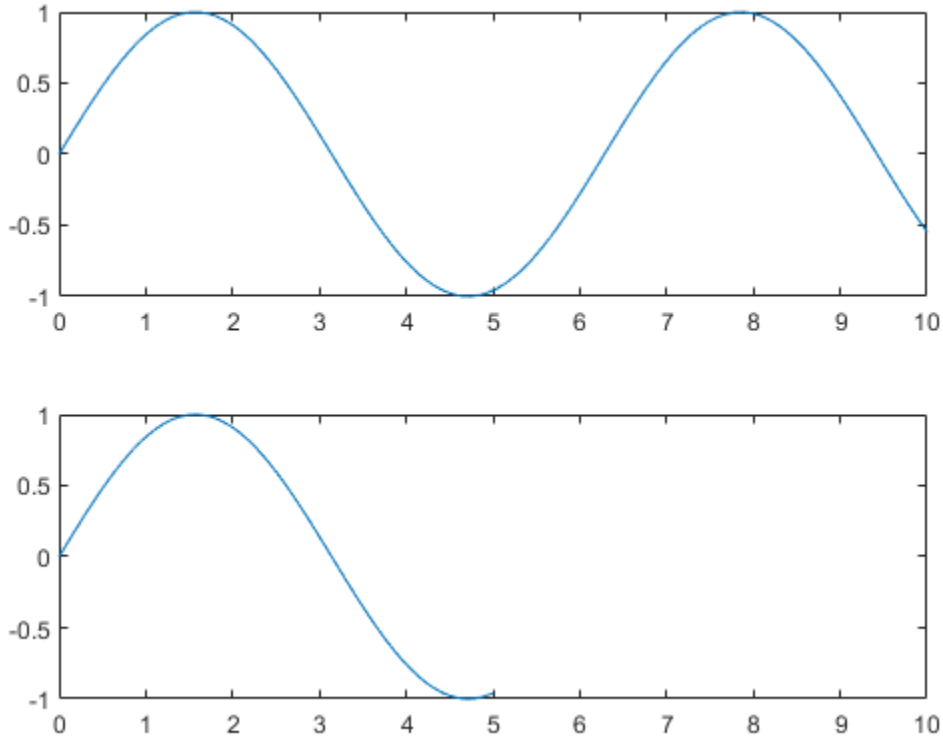


Set Axis Limits for Multiple Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Plot data in each axes. Then set the axis limits for both axes to the same values.

```
tiledlayout(2,1)
x1 = linspace(0,10,100);
y1 = sin(x1);
ax1 = nexttile;
```

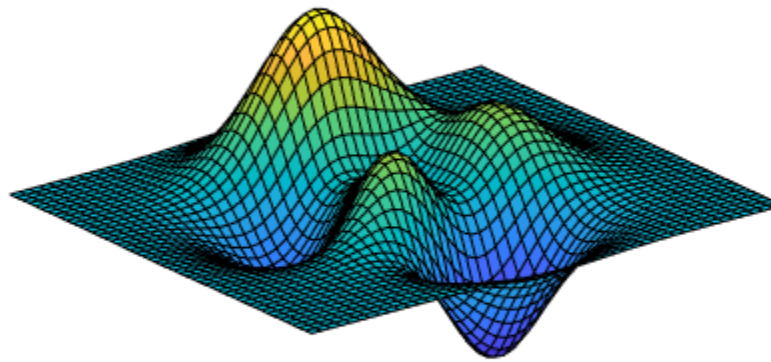
```
plot(ax1,x1,y1)

x2 = linspace(0,5,100);
y2 = sin(x2);
ax2 = nexttile;
plot(ax2,x2,y2)

axis([ax1 ax2],[0 10 -1 1])
```



Display Plot Without Axes Background

Plot a surface without displaying the axes lines and background.

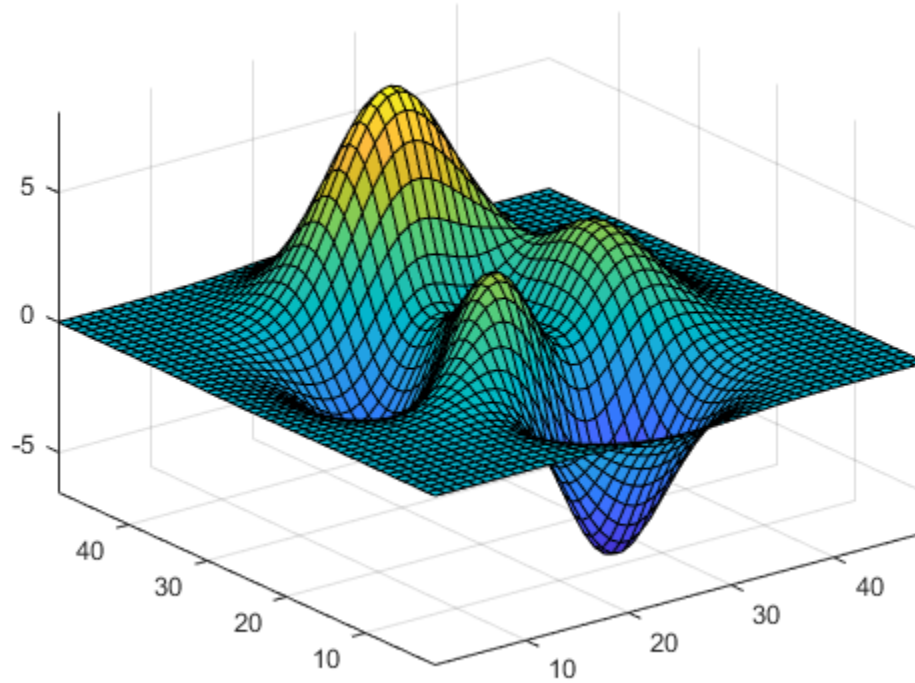
```
surf(peaks)
axis off
```



Use Tight Axis Limits and Return Values

Plot a surface. Set the axis limits to equal the range of the data so that the plot extends to the edges of the axes.

```
surf(peaks)
axis tight
```



Return the values of the current axis limits.

```
l = axis
```

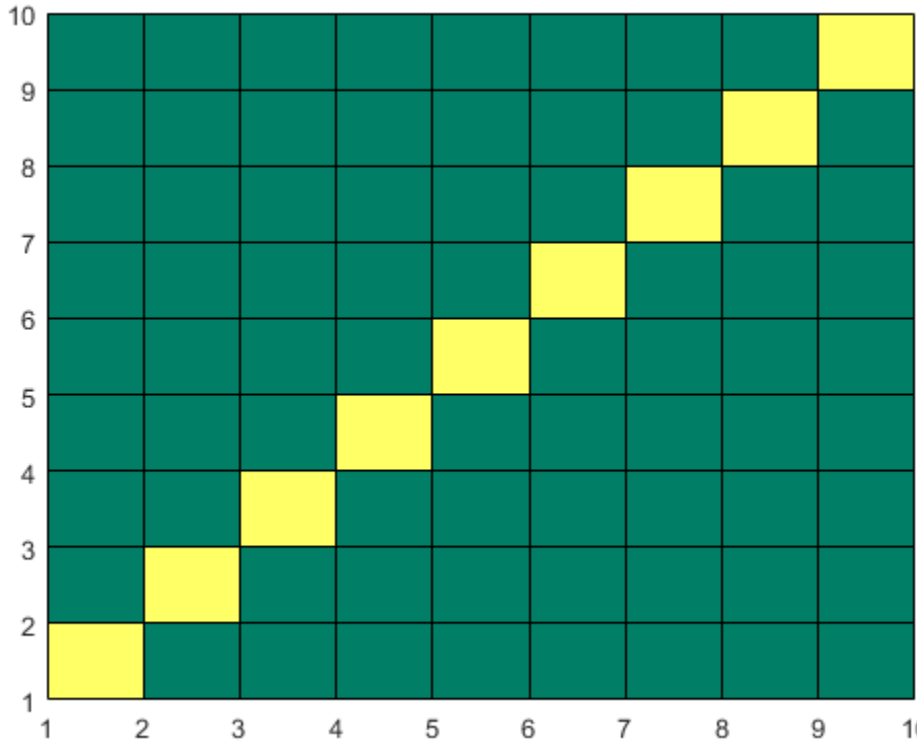
```
l = 1×6

    1.0000   49.0000    1.0000   49.0000   -6.5466    8.0752
```

Change Direction of Coordinate System

Create a checkerboard plot and change the direction of the coordinate system.

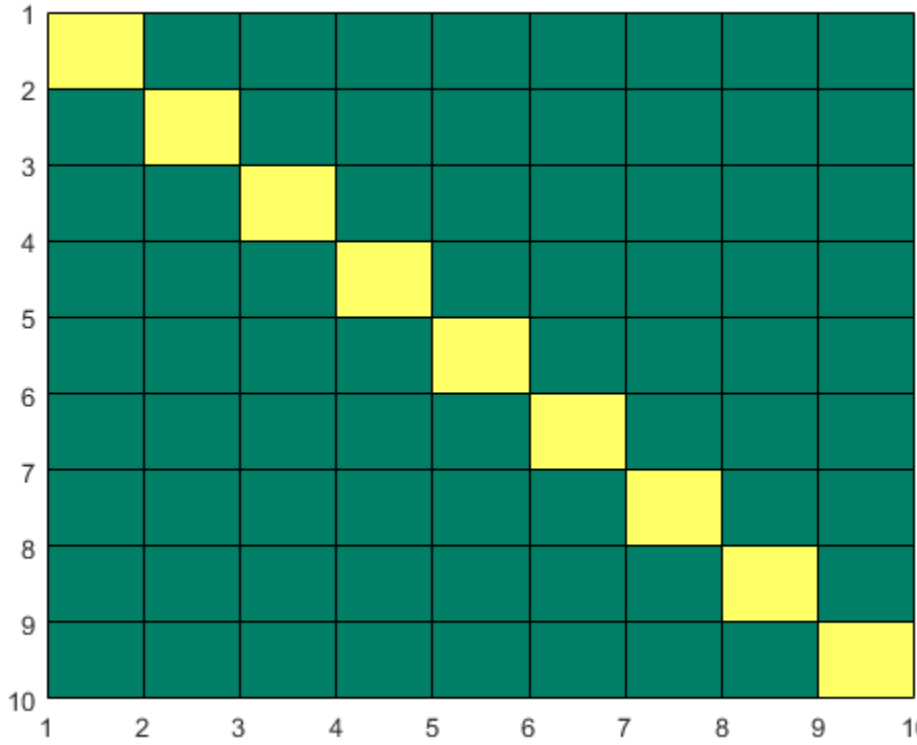
First, create the plot using the `summer` colormap. By default, the *x* values increase from left to right and the *y* values increase from bottom to top.

```
C = eye(10);
pcolor(C)
colormap summer
```



Reverse the coordinate system so that the *y* values increase from top to bottom.

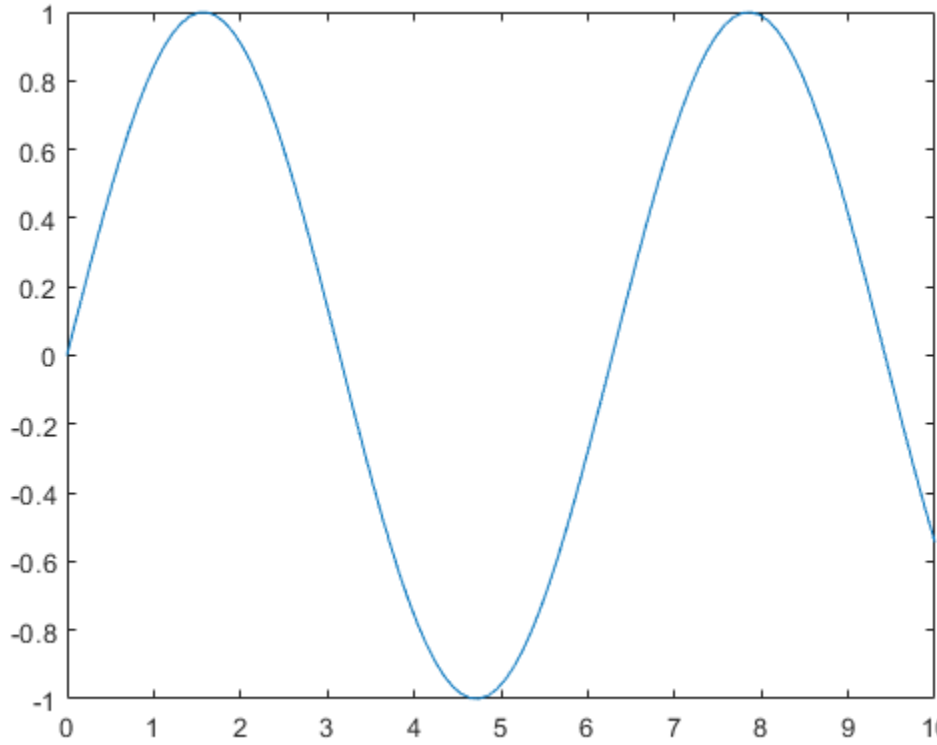
```
axis ij
```



Retain Current Axis Limits When Adding New Plots

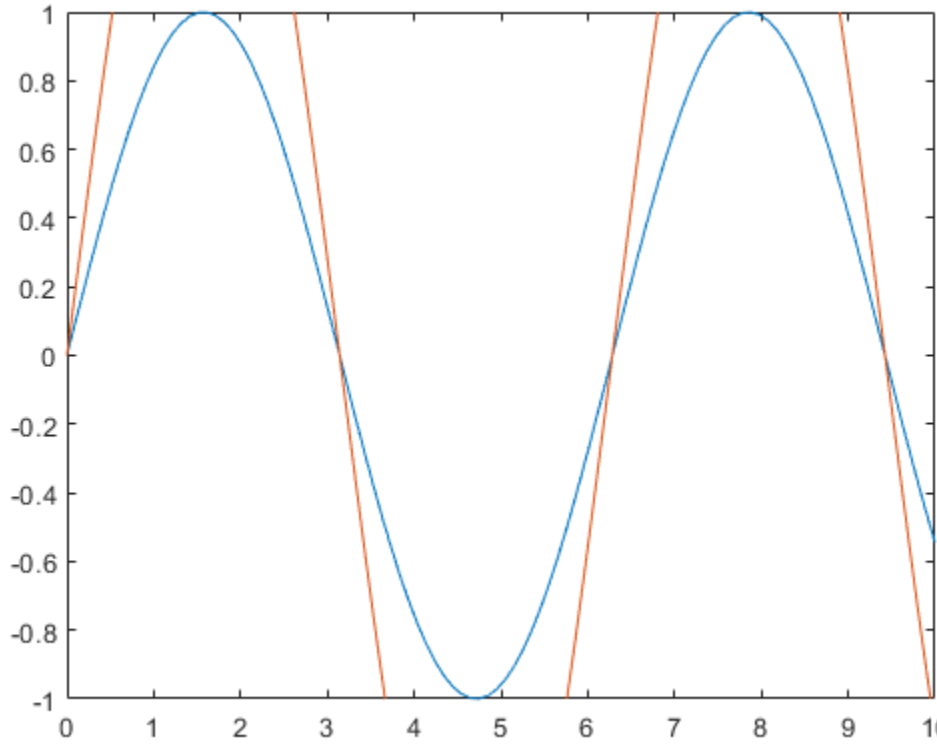
Plot a sine wave.

```
x = linspace(0,10);
y = sin(x);
plot(x,y)
```



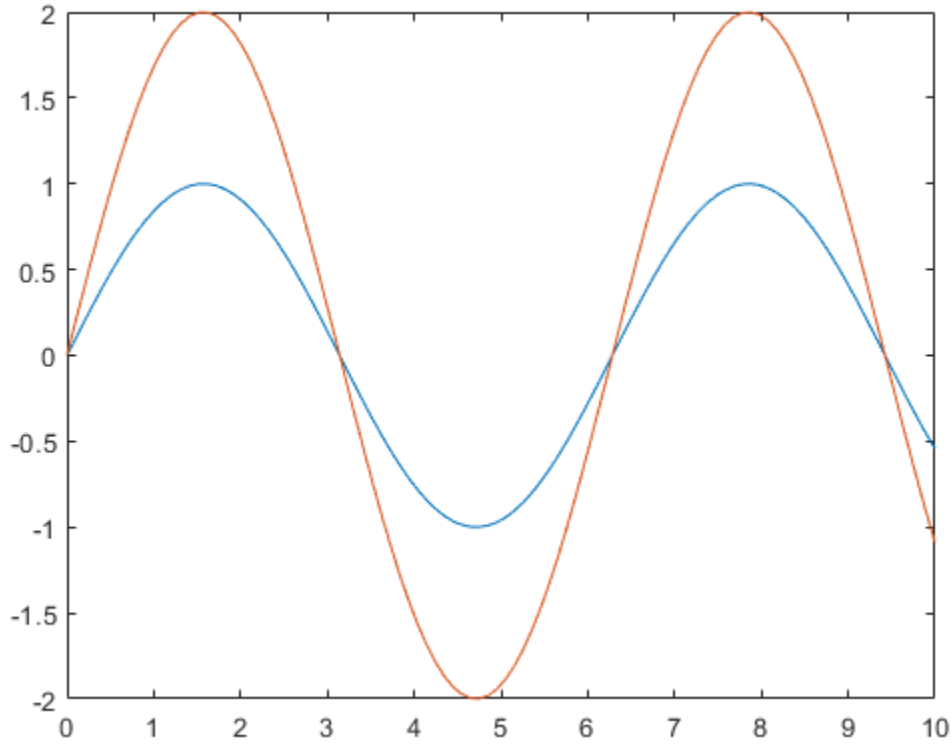
Add another sine wave to the axes using `hold on`. Keep the current axis limits by setting the limits mode to manual.

```
y2 = 2*sin(x);
hold on
axis manual
plot(x,y2)
hold off
```



If you want the axes to choose the appropriate limits, set the limits mode back to automatic.

```
axis auto
```



Input Arguments

`limits` — Axis limits

four-element vector | six-element vector | eight-element vector

Axis limits, specified as a vector of four, six, or eight elements.

For Cartesian axes, specify the limits in one of these forms:

- `[xmin xmax ymin ymax]` — Set the *x*-axis limits to range from `xmin` to `xmax`. Set the *y*-axis limits to range from `ymin` to `ymax`.

- `[xmin xmax ymin ymax zmin zmax]` — Also set the *z*-axis limits to range from `zmin` to `zmax`.
- `[xmin xmax ymin ymax zmin zmax cmin cmax]` — Also set the color limits. `cmin` is the data value that corresponds to the first color in the colormap. `cmax` is the data value that corresponds to the last color in the colormap.

The `XLim`, `YLim`, `ZLim`, and `CLim` properties for the `Axes` object store the limit values.

For polar axes, specify the limits in this form:

- `[thetamin thetamax rmin rmax]` — Set the theta-axis limits to range from `thetamin` to `thetamax`. Set the *r*-axis limits to range from `rmin` to `rmax`.

The `ThetaLim` and `RLim` properties for the `PolarAxes` object store the limit values.

For partially automatic limits, use `inf` or `-inf` for the limits you want the axes to choose automatically. For example, `axis([-inf 10 0 inf])` lets the axes choose the appropriate minimum *x*-axis limit and maximum *y*-axis limit. It uses the specified values for the maximum *x*-axis limit and minimum *y*-axis limit.

Note If the *x*-axis, *y*-axis, or *z*-axis displays categorical, datetime, or duration values, then use the `xlim`, `ylim`, and `zlim` functions to set the limits instead.

Example: `axis([0 1 0 1])`

Example: `axis([0 1 0 1 0 1])`

Example: `axis([0 inf 0 inf])`

`mode` — Manual, automatic, or semiautomatic selection of axis limits
`manual` | `auto` | `'auto x'` | `'auto y'` | `'auto z'` | `'auto xy'` | `'auto xz'` | `'auto yz'`

Manual, automatic, or semiautomatic selection of axis limits, specified as one of the values in this table.

Value	Description	Axes Properties That Change
`manual`	Freeze all axis limits at their current values.	Sets `XLimMode`, `YLimMode`, and `ZLimMode` to `'manual'`. If you are working with polar axes, then this option sets `ThetaLimMode` and `RLimMode` to `'manual'`.
`auto`	Automatically choose all axis limits.	Sets `XLimMode`, `YLimMode`, and `ZLimMode` to `'auto'`. If you are working with polar axes, then this option sets `ThetaLimMode` and `RLimMode` to `'auto'`.
`'auto x'`	Automatically choose the *x*-axis limits.	Sets `XLimMode` to `'auto'`.
`'auto y'`	Automatically choose the *y*-axis limits.	Sets `YLimMode` to `'auto'`.
`'auto z'`	Automatically choose the *z*-axis limits.	Sets `ZLimMode` to `'auto'`.
`'auto xy'`	Automatically choose the *x*-axis and *y*-axis limits.	Sets `XLimMode` and `YLimMode` to `'auto'`.
`'auto xz'`	Automatically choose the *x*-axis and *z*-axis limits.	Sets `XLimMode` and `ZLimMode` to `'auto'`.
`'auto yz'`	Automatically choose the *y*-axis and *z*-axis limits.	Sets `YLimMode` and `ZLimMode` to `'auto'`.

Note You cannot use these options with polar axes.

`style` — Axis limits and scaling

`tight` | `fill` | `equal` | `image` | `square` | `vis3d` | `normal`

Axis limits and scaling, specified as one of these values.

Value	Description	Axes Properties That Change
`tight`	Fit the axes box tightly around the data by setting the axis limits equal to the range of the data.	`XLimMode`, `YLimMode`, and `ZLimMode` change to `'auto'`. If you are working with polar axes, then `ThetaLimMode` and `RLimMode` change. The limits automatically update to incorporate new data added to the axes. To keep the limits from changing when using `hold on`, use `axis tight manual`.
`equal`	Use the same length for the data units along each axis.	Sets `DataAspectRatio` to `[1 1 1]`, sets `PlotBoxAspectRatio` to `[3 4 4]`, and sets the associated mode properties to manual. Disables the “stretch-to-fill” behavior.
`image`	Use the same length for the data units along each axis and fit the axes box tightly around the data.	Sets `DataAspectRatio` to `[1 1 1]`and sets the associated mode property to manual. Disables the “stretch-to-fill” behavior.
`square`	Use axis lines with equal lengths. Adjust the increments between data units accordingly.	Sets `PlotBoxAspectRatio` to `[1 1 1]` and sets the associated mode property to manual. Disables the “stretch-to-fill” behavior.
`fill`	Enable the “stretch-to-fill” behavior (the default). The lengths of each axis line fill the position rectangle defined in the `Position` property of the axes.	Sets the plot box aspect ratio mode and data aspect ratio mode properties to auto.

Value	Description	Axes Properties That Change
`vis3d`	Freeze the aspect ratio properties.	Sets the plot box aspect ratio mode and data aspect ratio mode properties to manual.
`normal`	Restore the default behavior.	Sets the plot box aspect ratio mode and data aspect ratio mode properties to auto.

For more information on the plot box aspect ratio and the data aspect ratio, see the `PlotBoxAspectRatio` and `DataAspectRatio` properties.

Note You cannot use these options with polar axes, except for the `axis tight` and `axis normal` commands.

`ydirection` — y-axis direction

`xy` (default) | `ij`

y-axis direction, specified as one of these values:

- `xy` — Default direction. For axes in a 2-D view, the *y*-axis is vertical with values increasing from bottom to top.
- `ij` — Reverse direction. For axes in a 2-D view, the *y*-axis is vertical with values increasing from top to bottom.

Note You cannot use these options with polar axes.

`visibility` — Axes lines and background visibility

`on` (default) | `off`

Axes lines and background visibility, specified as either `on` or `off`. Specifying the visibility sets the `Visible` property of the `Axes` object or `PolarAxes` object to the specified value.

ax — Target axes

one or more axes

Target axes, specified as one or more axes. You can specify `Axes` objects or `PolarAxes` objects. If you do not specify the axes, then `axis` sets the limits for the current axes (`gca`).

When you specify the axes, use single quotes around other input arguments that are character vectors.

Example: `axis(ax,'tight')`

Example: `axis(ax,limits)`

Example: `axis(ax,'manual')`

Output Arguments

`lim` — Current limit values
four-element vector | six-element vector

Current limit values, returned as a four-element or six-element vector.

- For Cartesian axes in a 2-D view, `lim` is of the form `[xmin xmax ymin ymax]`. For axes in a 3-D view, `lim` is of the form `[xmin xmax ymin ymax zmin zmax]`. The `XLim`, `YLim`, and `ZLim` properties for the `Axes` object store the limit values.
- For polar axes, `lim` is of the form `[thetamin thetamax rmin rmax]`. The `ThetaLim` and `RLim` properties for the `PolarAxes` object store the limit values.

Tips

- You can combine multiple input arguments together, for example, `axis image ij` . The options are evaluated from left to right. Subsequent options can overwrite properties set by prior ones.
- If axes do not exist, the `axis` function creates them.
- Use `hold on` to keep plotting functions from overriding preset axis limits.

See Also

Functions
`caxis` | `grid` | `nexttile` | `tiledlayout` | `title` | `xlim` | `ylim` | `zlim`

Properties
Axes | PolarAxes

Topics

"Specify Axis Limits"
"Control Ratio of Axis Lengths and Data Unit Lengths"

Introduced before R2006a

axtoolbar

Create axes toolbar

Syntax

```
tb = axtoolbar(buttons)
tb = axtoolbar(ax,buttons)

tb = axtoolbar
tb = axtoolbar(ax)

tb = axtoolbar( ___ ,Name,Value)
[tb,btns] = axtoolbar( ___ )
```

Description

`tb = axtoolbar(buttons)` replaces the default toolbar that appears above the top-right corner of the current axes with a toolbar that contains only the specified buttons. For example, `axtoolbar({'pan','restoreview'})` specifies a button to pan and a button to restore the original view. The function returns the `AxesToolbar` object created.

`tb = axtoolbar(ax,buttons)` replaces the toolbar for the axes or tiled chart layout specified by `ax`, instead of the current axes.

`tb = axtoolbar` replaces the toolbar for the current axes with an empty toolbar.

`tb = axtoolbar(ax)` replaces the toolbar for the specified axes or tiled chart layout with an empty toolbar, instead of the current axes.

`tb = axtoolbar( ___ ,Name,Value)` specifies toolbar properties using one or more name-value pair arguments.

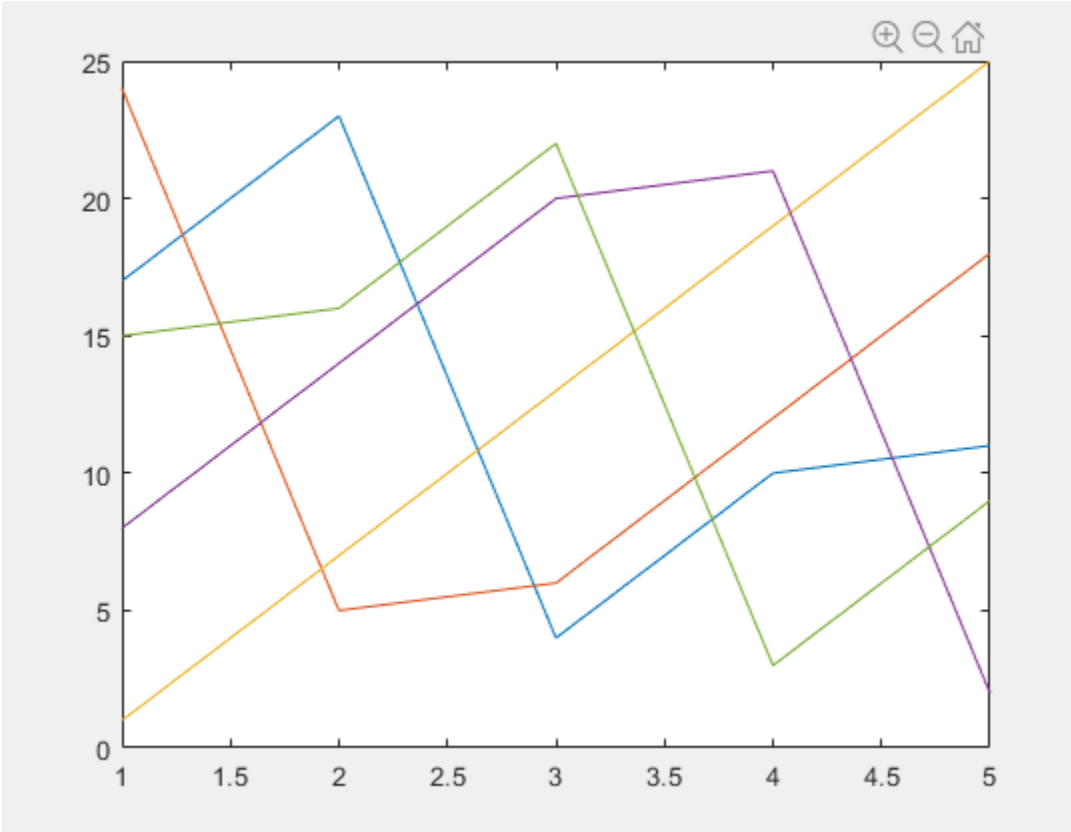
`[tb,btns] = axtoolbar( ___ )` also returns the toolbar button objects created, which are either `ToolbarStateButton`, `ToolbarPushButton`, or `ToolbarDropdown` objects. You can use the objects to modify the toolbar and toolbar buttons after you create them.

Examples

Create Toolbar with Subset of Default Buttons

Create a plot. Replace the standard axes toolbar with a custom toolbar that includes buttons to zoom in, zoom out, and restore the view. Return the `AxesToolbar` object and the button objects created as output arguments.

```
plot(magic(5))
[tb,btns] = axtoolbar({'zoomin','zoomout','restoreview'});
```

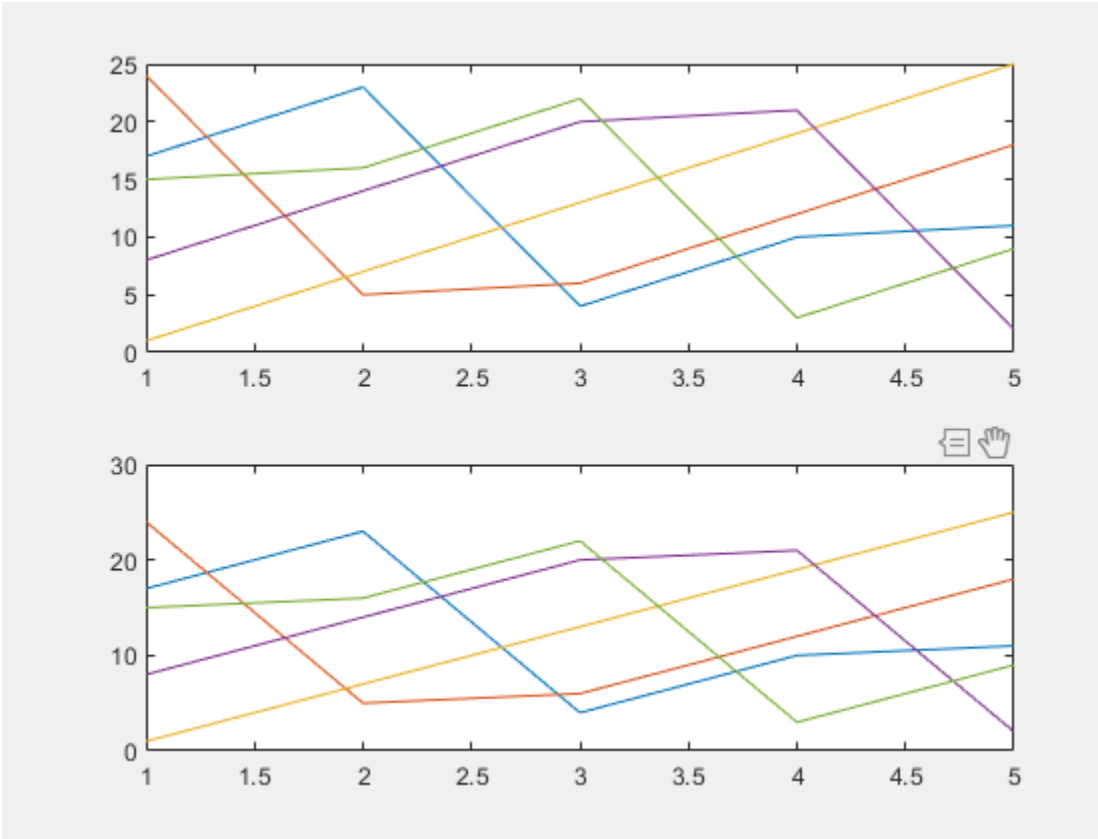


Create Toolbar for Specific Axes Object

Create two subplots with a custom toolbar for each one.

First, create two subplots and assign the `Axes` objects to the variables `ax1` and `ax2`. Replace the toolbar for the upper subplot with a custom toolbar by specifying `ax1` as the first input argument to the `axtoolbar` function. Then, replace the toolbar for the lower subplot. Hover over each subplot to see its toolbar.

```
ax1 = subplot(2,1,1);
plot(ax1,magic(5))
[tb1,btns1] = axtoolbar(ax1,{'zoomin','zoomout','restoreview'});

ax2 = subplot(2,1,2);
plot(ax2,magic(5))
[tb2,btns2] = axtoolbar(ax2,{'pan','datacursor'});
```



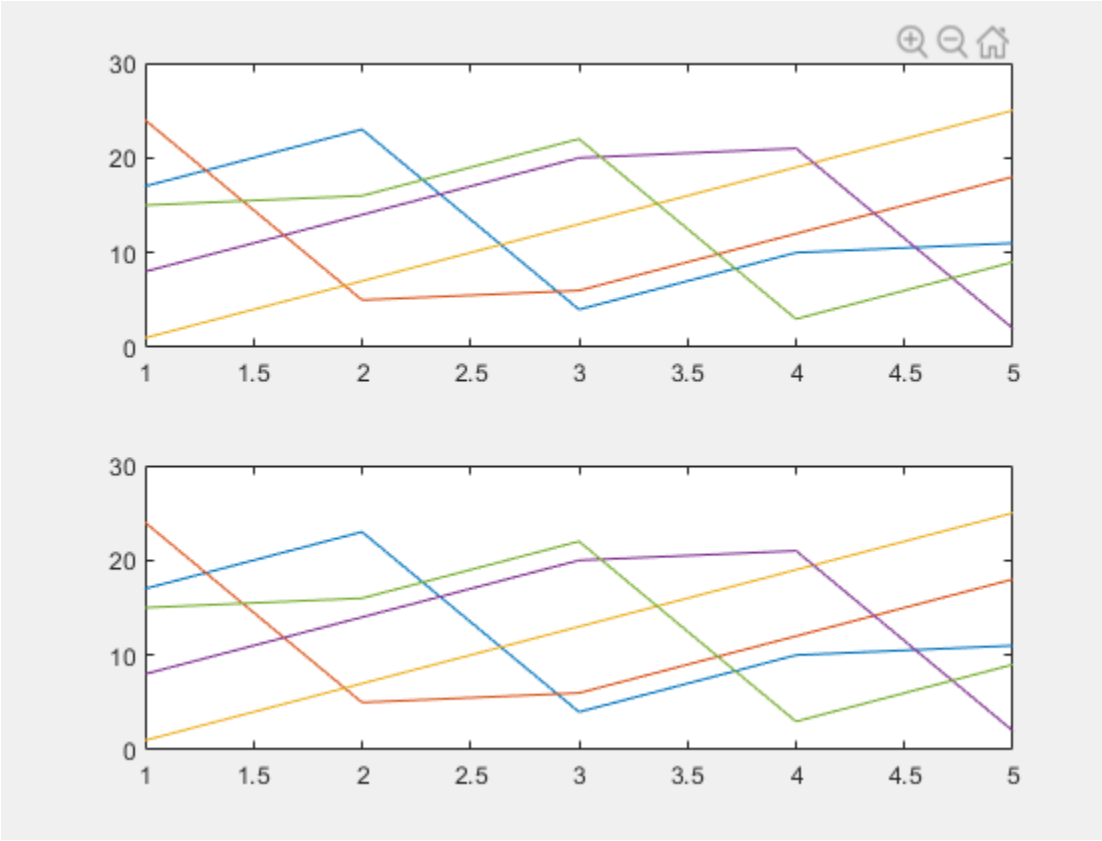
Create Toolbar for Tiled Chart Layout

Create a tiled chart layout with one axes toolbar.

First, create a 2-by-1 layout and display a different chart in each tile. Then, create a custom axes toolbar for the tiled chart layout. Display the toolbar by hovering over the layout.

```
t = tiledlayout(2,1);
nexttile
plot(magic(5));
nexttile
```

```
plot(magic(5));

tb = axtoolbar(t,{'zoomin','zoomout','restoreview'});
```



Input Arguments

ax — Target axes
`Axes` object | `TiledChartLayout` object

Target axes, specified as one of the following:

- `Axes` object – Create a custom axes toolbar for the specified set of axes.

- `TiledChartLayout` object – Create a single axes toolbar that applies to all axes in the specified tiled chart layout.

`buttons` — Toolbar buttons
`'default'` | cell array with one or more button names

Toolbar buttons, specified as `'default'` for the default set of buttons or a cell array containing one or more button names listed in this table. The buttons appear in a standard order on the toolbar regardless of the order in which you specify them. Each button can appear only once in the toolbar.

Button Name	Icon	Description
`'export'`		Display menu of export options.
`'brush'`		Toggle data brushing mode.
`'datacursor'`		Toggle data cursor mode.
`'rotate'`		Toggle rotate mode.
`'pan'`		Toggle pan mode.
`'zoomin'`		Toggle zoom-in mode.
`'zoomout'`		Toggle zoom-out mode.
`'restoreview'`		Restore original view of axes or tiled chart layout.

Hovering over the `'export'` button reveals a drop-down menu with options for exporting the axes or tiled chart layout content:

- : Save the content as a tightly cropped image or PDF.
- : Copy the content as a PNG image.
- : Copy the content as a vector graphic.

Example: `axtoolbar({'zoomin','zoomout','restoreview'})`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
`axtoolbar({'zoomin','zoomout'},'SelectionChangedFcn',@mycallback)`

Note The properties listed here are only a subset. For a full list, see AxesToolbar.

`SelectionChangedFcn` — Callback for selection changes

`''` (default) | function handle | cell array | character vector

Callback for selection changes, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when you click a state button. It does not execute if a state button `Value` property changes programmatically.

This callback function can access specific information about interaction with the buttons. MATLAB passes this information in a `SelectionChangedEventData` object as the second argument to your callback function. You can query the object properties using dot notation. For example, `event.Selection` returns the currently selected button. The `SelectionChangedEventData` object is not available to callback functions specified as character vectors.

This table lists the properties of the `SelectionChangedEventData` object.

Property	Description
`Axes`	Array of `Axes` objects associated with the toolbar
`Selection`	Currently selected button

Property	Description
`PreviousSelection`	Previously selected button
`Source`	`AxesToolbar` object
`EventName`	`'SelectionChanged'`

`Visible` — State of visibility
`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

Output Arguments

`tb` — Toolbar
`AxesToolbar` object

Toolbar, returned as an `AxesToolbar` object. Use `tb` to modify the toolbar after you create it. For a list of properties, see AxesToolbar.

`btns` — Toolbar buttons
graphics array

Toolbar buttons, returned as a graphics array containing one or more `ToolbarPushButton`, `ToolbarStateButton`, or `ToolbarDropdown` objects. Use the elements in the `btns` array to modify the buttons after you create them. For a list of properties, see ToolbarStateButton, ToolbarPushButton, and ToolbarDropdown Properties.

Limitations

- Custom toolbars do not appear in figures in the Live Editor. To see the custom toolbar, open the figure in a separate figure window by clicking the button in the upper-right corner of the figure.

See Also

Functions
`axtoolbarbtn`

Properties
AxesToolbar | ToolbarDropdown Properties | ToolbarPushButton | ToolbarStateButton

Introduced in R2018b

axtoolbarbtn

Add buttons to axes toolbar

Syntax

```
btn = axtoolbarbtn(tb)
btn = axtoolbarbtn(tb,style)
btn = axtoolbarbtn(tb,style,Name,Value)
```

Description

`btn = axtoolbarbtn(tb)` adds a push button to the axes toolbar specified by `tb` and returns the `ToolbarPushButton` object.

`btn = axtoolbarbtn(tb,style)` adds either a push button or a state button to the toolbar specified by `tb` and returns the button object. For a push button, set the style to `'push'`. For a state button with two states, set the style to `'state'`.

`btn = axtoolbarbtn(tb,style,Name,Value)` specifies button properties using one or more name-value pair arguments.

Examples

Add State Button to Toolbar

Add a custom state button for the axes toolbar that turns on and off the axes grid lines.

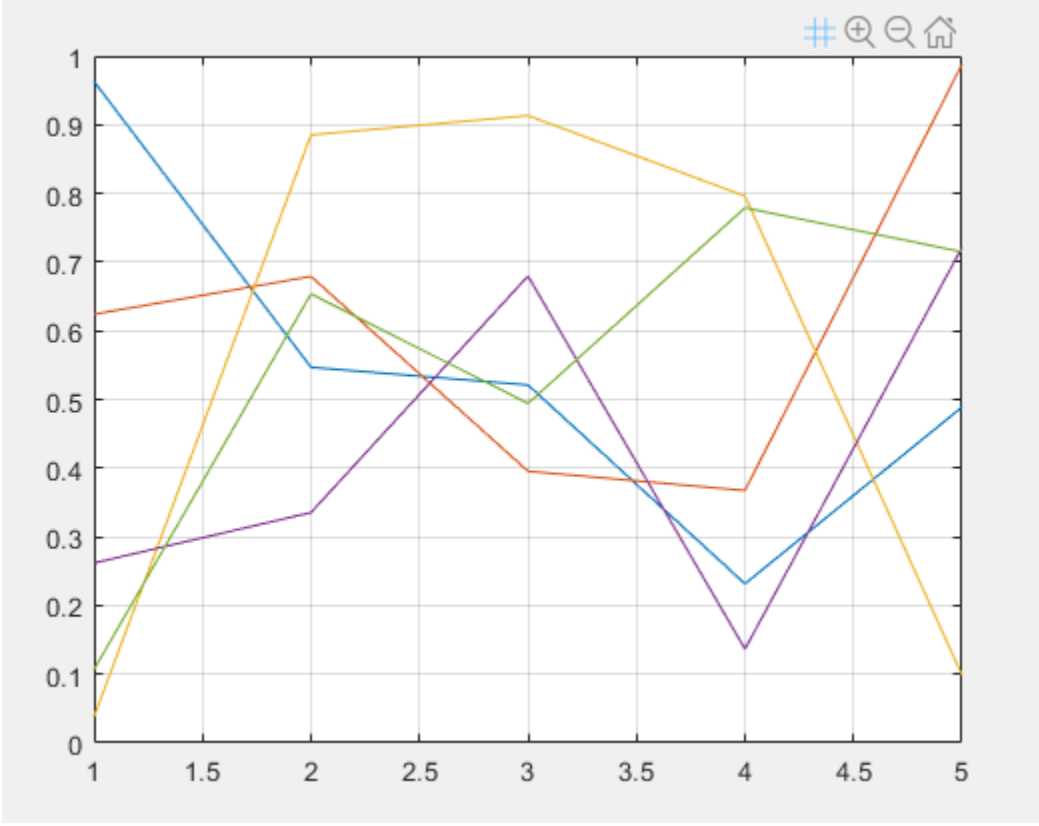
First, create a program file called `mycustomstatebutton.m`. Within the program file:

- Plot random data.
- Create a toolbar for the axes with options to zoom in, zoom out, and restore the view using the `axtoolbar` function.

- Add an empty state button to the toolbar using the `axtoolbarbtn` function. Return the `ToolbarStateButton` object.
- Specify the icon, tool tip, and callback function for the state button by setting the `Icon`, `Tooltip`, and `ValueChangedFcn` properties. This example uses the # icon, which you must first save as an image file called `mygridicon.png` on your path.

When you run the program file, click the icon to turn on and off the grid lines.

```
function mycustomstatebutton

plot(rand(5))
ax = gca;
tb = axtoolbar(ax,{'zoomin','zoomout','restoreview'});

btn = axtoolbarbtn(tb,'state');
btn.Icon = 'mygridicon.png';
btn.Tooltip = 'Grid Lines';
btn.ValueChangedFcn = @customcallback;

    function customcallback(src,event)
        switch src.Value
            case 'off'
                event.Axes.XGrid = 'off';
                event.Axes.YGrid = 'off';
                event.Axes.ZGrid = 'off';
            case 'on'
                event.Axes.XGrid = 'on';
                event.Axes.YGrid = 'on';
                event.Axes.ZGrid = 'on';
        end
    end

end
```



Add Push Button to Toolbar

Create a custom push button for the axes toolbar that snaps the view of the axes to a 2-D view.

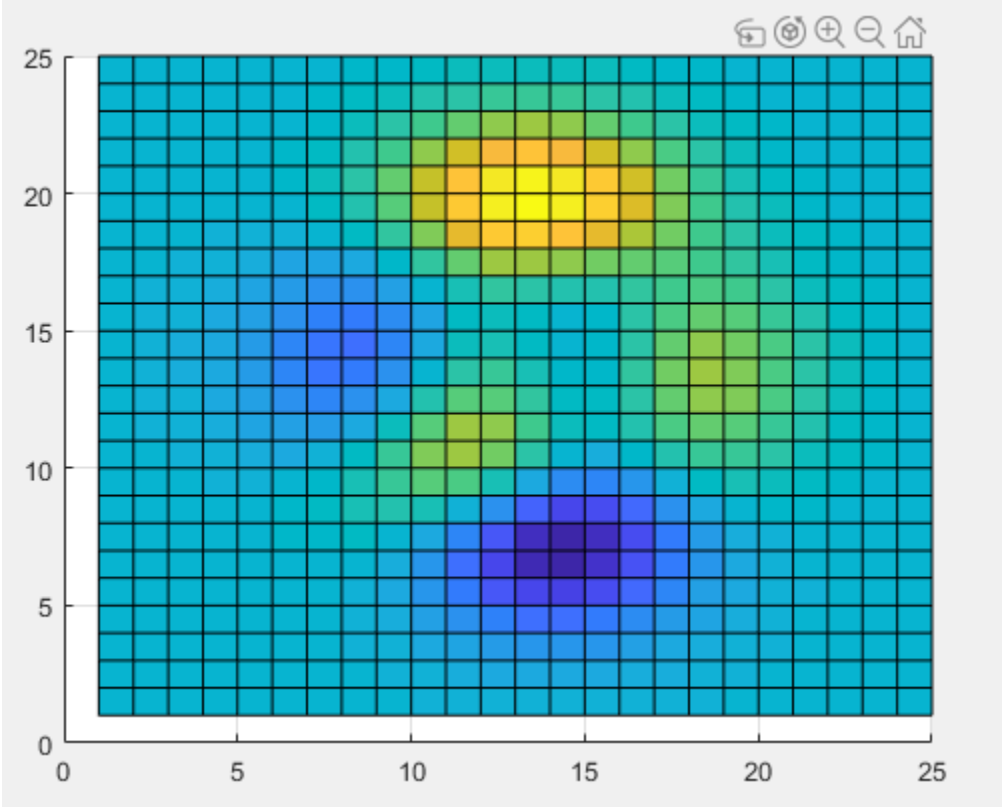
First, create a program file called `mycustompushbutton.m`. Within the program file:

- Plot a surface.
- Create a toolbar for the axes with options to zoom in, zoom out, rotate, and restore the view using the `axtoolbar` function.
- Add an empty push button to the toolbar using the `axtoolbarbtn` function. Return the `ToolbarPushButton` object.

- Specify the icon, tool tip, and callback function for the push button by setting the `Icon`, `Tooltip`, and `ButtonPressedFcn` properties, respectively. This example uses the icon, which you must first save as an image file called `my2dicon.png` on your path.

When you run the program file, click the icon to snap the view of the axes to a 2-D view.

```
function mycustompushbutton

surf(peaks(25))
ax = gca;
tb = axtoolbar(ax,{'zoomin','zoomout','rotate','restoreview'});

btn = axtoolbarbtn(tb,'push');
btn.Icon = 'my2dicon.png';
btn.Tooltip = 'Snap to 2-D view';
btn.ButtonPushedFcn = @customcallback;

    function customcallback(src,event)
        view(event.Axes,2)
    end

end
```



Input Arguments

`tb` — Toolbar
`AxesToolbar` object

Toolbar, specified as an `AxesToolbar` object. Create the toolbar using the `axtoolbar` function, such as `tb = axestoolbar`.

`style` — Button style
`'push'` | `'state'`

Button style, specified as one of these options.

Style	Description
`'push'`	Push button. When clicked once, the button appears to press and release.
`'state'`	State button with two states. When clicked once, the button remains in the pressed or released state until it is clicked again.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `axtoolbarbtn(tb,'push','Icon','myimage.png','Tooltip','My Icon Tooltip')`

Note The properties listed here are only a subset. Push buttons and state buttons support a different set of properties. For a full list of properties and descriptions for each type, see the associated property page.

- ToolbarPushButton
- ToolbarStateButton

`Icon` — Button icon

`'none'` (default) | file name | m-by-n-by-3 array | m-by-n matrix | predefined icon

Button icon, specified as one of these values. The axes toolbar displays icons as grayscale images. MATLAB scales down the image to fit, if necessary.

- File name — Specify the file name as a character vector or a string scalar. The file name can be preceded by a path. The image file must be in JPEG, GIF, or PNG format.
- Array — Specify an m-by-n-by-3 array of RGB triplets that represent pixels in an image.
- Matrix — Specify an m-by-n matrix of numeric values that represent pixels in an image.

- If the values are of an integer type, then specify values between `0` and `63`. A value of `0` is opaque and a value of `63` is transparent. `NaN` is also transparent.
- If the values are of type `double`, then specify values between `1` and `64`. A value of `1` is opaque and a value of `64` is transparent. `NaN` is also transparent.

- Predefined icon — Specify one of the names in this table.

Icon Name	Result
`'brush'`	
`'datacursor'`	
`'export'`	
`'rotate'`	
`'pan'`	
`'zoomin'`	
`'zoomout'`	
`'restoreview'`	
`'none'`	No icon

Example: `btn.Icon = 'icon.png'`

Example: `btn.Icon = 'C:\Documents\icon.png'`

Example: `btn.Icon = 'rotate'`

`Tooltip` — Button tool tip

`''` (default) | character vector | cell array of character vectors | string array

Button tool tip, specified as a character vector, cell array of character vectors, or a string array. Use this property to display a message when you hover the pointer over the button. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array displays a separate line of text.

Example: `btn.Tooltip = 'My Tooltip'`

See Also

Functions
`axtoolbar`

Properties
AxesToolbar | ToolbarPushButton | ToolbarStateButton

Introduced in R2018b

AxesToolbar Properties

Axes toolbar appearance and behavior

Description

`AxesToolbar` properties control the appearance and behavior of the `AxesToolbar` object. By changing property values, you can modify certain aspects of the toolbar.

```
tb = axtoolbar('default');
tb.Visible = 'off';
```

Properties

Interactivity

`Visible` — Visibility

`'on'` (default) | `'off'`

Visibility the axes toolbar when you hover over the axes, specified as one of these values:

- `'on'` — Display the toolbar above the top-right corner of the axes when you hover over the axes.
- `'off'` — Do not display the toolbar. You still can access the properties of an invisible toolbar.

Callbacks

`SelectionChangedFcn` — Callback for selection changes

`''` (default) | function handle | cell array | character vector

Callback for selection changes, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when you click a state button. It does not execute if a state button `Value` property changes programmatically.

This callback function can access specific information about interaction with the buttons. MATLAB passes this information in a `SelectionChangedEventData` object as the second argument to your callback function. You can query the object properties using dot notation. For example, `event.Selection` returns the currently selected button. The `SelectionChangedEventData` object is not available to callback functions specified as character vectors.

This table lists the properties of the `SelectionChangedEventData` object.

Property	Description
`Axes`	Array of `Axes` objects associated with the toolbar
`Selection`	Currently selected button
`PreviousSelection`	Previously selected button
`Source`	`AxesToolbar` object
`EventName`	`'SelectionChanged'`

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running

callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Children` — Child buttons
graphics array

Child buttons, specified as a graphics array of `ToolbarPushButton`, `ToolbarStateButton`, and `ToolbarDropdown` objects. If you are using the default axes toolbar, then this property does not contain any children. However, if you create a custom toolbar using the `axtoolbar` function, then this property contains the child buttons.

`Parent` — Parent container
`Axes` object | `PolarAxes` object | `GeographicAxes` object

Parent container, specified as an `Axes`, `PolarAxes`, or `GeographicAxes` object.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'axestoolbar'` (default)

This property is read-only.

Type of graphics object, returned as `'axestoolbar'`. Use this property to find all objects of a given type within a plotting hierarchy.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[ ]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

Unused Properties

`UIContextMenu` — Context menu (not used)
`uicontextmenu` object

Context menu, specified as a `ContextMenu` object.

Note The axes toolbar does not use this property.

`ButtonDownFcn` — Mouse-click callback (not used)
`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as a function handle, a cell array, or a character vector.

Note The axes toolbar does not use this property.

See Also

`axtoolbar`

Topics

“Access Property Values”
“Graphics Object Properties”

Introduced in R2018b

balance

Diagonal scaling to improve eigenvalue accuracy

Syntax

```
[T,B] = balance(A)
[S,P,B] = balance(A)
B = balance(A)
B = balance(A,'noperm')
```

Description

`[T,B] = balance(A)` returns a similarity transformation `T` such that `B = T\A*T`, and `B` has, as nearly as possible, approximately equal row and column norms. `T` is a permutation of a diagonal matrix whose elements are integer powers of two to prevent the introduction of roundoff error. If `A` is symmetric, then `B == A` and `T` is the identity matrix.

`[S,P,B] = balance(A)` returns the scaling vector `S` and the permutation vector `P` separately. The transformation `T` and balanced matrix `B` are obtained from `A`, `S`, and `P` by `T(:,P) = diag(S)` and `B(P,P) = diag(1./S)*A*diag(S)`.

`B = balance(A)` returns just the balanced matrix `B`.

`B = balance(A,'noperm')` scales `A` without permuting its rows and columns.

Examples

This example shows the basic idea. The matrix `A` has large elements in the upper right and small elements in the lower left. It is far from being symmetric.

```
A = [1  100  10000; .01  1  100; .0001  .01  1]
A =
   1.0e+04 *
    0.0001    0.0100    1.0000
```

```
    0.0000    0.0001    0.0100
    0.0000    0.0000    0.0001
```

Balancing produces a diagonal matrix T with elements that are powers of two and a balanced matrix B that is closer to symmetric than A.

```
[T,B] = balance(A)
T =
   1.0e+03 *
    2.0480         0         0
         0    0.0320         0
         0         0    0.0003
B =
    1.0000    1.5625    1.2207
    0.6400    1.0000    0.7813
    0.8192    1.2800    1.0000
```

To see the effect on eigenvectors, first compute the eigenvectors of A, shown here as the columns of V.

```
[V,E] = eig(A); V
V =
0.9999        -0.9999            -0.9999
0.0100         0.0059 + 0.0085i   0.0059 - 0.0085i
0.0001         0.0000 - 0.0001i   0.0000 + 0.0001i
```

Note that all three vectors have the first component the largest. This indicates V is badly conditioned; in fact `cond(V)` is `8.7766e+003`. Next, look at the eigenvectors of B.

```
[V,E] = eig(B); V
V =
0.6933        -0.6993            -0.6993
0.4437         0.2619 + 0.3825i   0.2619 - 0.3825i
0.5679         0.2376 - 0.4896i   0.2376 + 0.4896i
```

Now the eigenvectors are well behaved and `cond(V)` is `1.4421`. The ill conditioning is concentrated in the scaling matrix; `cond(T)` is `8192`.

This example is small and not really badly scaled, so the computed eigenvalues of A and B agree within roundoff error; balancing has little effect on the computed results.

Limitations

Balancing can destroy the properties of certain matrices; use it with some care. If a matrix contains small elements that are due to roundoff error, balancing might scale them up to make them as significant as the other elements of the original matrix.

Tips

- Nonsymmetric matrices can have poorly conditioned eigenvalues. Small perturbations in the matrix, such as roundoff errors, can lead to large perturbations in the eigenvalues. The condition number of the eigenvector matrix,

  ```
  cond(V) = norm(V)*norm(inv(V))
  ```

  where

  ```
  [V,T] = eig(A)
  ```

  relates the size of the matrix perturbation to the size of the eigenvalue perturbation. Note that the condition number of `A` itself is irrelevant to the eigenvalue problem.

  Balancing is an attempt to concentrate any ill conditioning of the eigenvector matrix into a diagonal scaling. Balancing usually cannot turn a nonsymmetric matrix into a symmetric matrix; it only attempts to make the norm of each row equal to the norm of the corresponding column.

  Note The MATLAB eigenvalue function, `eig(A)`, automatically balances `A` before computing its eigenvalues. Turn off the balancing with `eig(A,'nobalance')`.

See Also

`eig` | `equilibrate`

bandwidth

Lower and upper matrix bandwidth

Syntax

```
B = bandwidth(A,type)
```

```
[lower,upper] = bandwidth(A)
```

Description

`B = bandwidth(A,type)` returns the bandwidth on page 1-727 of matrix `A` specified by `type`. Specify `type` as `'lower'` for the lower bandwidth, or `'upper'` for the upper bandwidth.

`[lower,upper] = bandwidth(A)` returns the lower bandwidth, `lower`, and upper bandwidth, `upper`, of matrix `A`.

Examples

Find Bandwidth of Triangular Matrix

Create a 6-by-6 lower triangular matrix.

```
A = tril(magic(6))
```

```
A = 6×6

    35     0     0     0     0     0
     3    32     0     0     0     0
    31     9     2     0     0     0
     8    28    33    17     0     0
    30     5    34    12    14     0
     4    36    29    13    18    11
```

Find the lower bandwidth of A by specifying `type` as `'lower'`. The result is 5 since every diagonal below the main diagonal has nonzero elements.

```
B = bandwidth(A,'lower')
```

```
B = 5
```

Find the upper bandwidth of A by specifying `type` as `'upper'`. The result is 0 since there are no nonzero elements above the main diagonal.

```
B = bandwidth(A,'upper')
```

```
B = 0
```

Find Bandwidth of Sparse Block Matrix

Create a 100-by-100 sparse block matrix.

```
B = kron(speye(25),ones(4));
```

View a 10-by-10 section of elements from the top left of B.

```
full(B(1:10,1:10))
```

```
ans = 10×10

     1     1     1     1     0     0     0     0     0     0
     1     1     1     1     0     0     0     0     0     0
     1     1     1     1     0     0     0     0     0     0
     1     1     1     1     0     0     0     0     0     0
     0     0     0     0     1     1     1     1     0     0
     0     0     0     0     1     1     1     1     0     0
     0     0     0     0     1     1     1     1     0     0
     0     0     0     0     1     1     1     1     0     0
     0     0     0     0     0     0     0     0     1     1
     0     0     0     0     0     0     0     0     1     1
```

B has 4-by-4 blocks of ones centered on the main diagonal.

Find both the lower and upper bandwidths of B by specifying two output arguments.

```
[lower,upper] = bandwidth(B)
```

```
lower = 3

upper = 3
```

Input Arguments

A — Input matrix
2-D numeric matrix

Input matrix, specified as a 2-D numeric matrix. A can be either full or sparse.

Data Types: `single` | `double`
Complex Number Support: Yes

type — Bandwidth type
`'lower'` | `'upper'`

Bandwidth type, specified as `'lower'` or `'upper'`.

- Specify `'lower'` for the lower bandwidth (below the main diagonal).
- Specify `'upper'` for the upper bandwidth (above the main diagonal).

Output Arguments

B — Lower or upper bandwidth
nonnegative integer scalar

Lower or upper bandwidth, returned as a nonnegative integer scalar.

- If `type` is `'lower'`, then `0` ≤ B ≤ `size(A,1)-1`.
- If `type` is `'upper'`, then `0` ≤ B ≤ `size(A,2)-1`.

lower — Lower bandwidth
nonnegative integer scalar

Lower bandwidth, returned as a nonnegative integer scalar. `lower` is in the range `0` ≤ `lower` ≤ `size(A,1)-1`.

`upper` — Upper bandwidth
nonnegative integer scalar

Upper bandwidth, returned as a nonnegative integer scalar. `upper` is in the range `0` ≤ `upper` ≤ `size(A,2)-1`.

More About

Upper and Lower Bandwidth

The upper and lower bandwidths of a matrix are measured by finding the last diagonal (above or below the main diagonal, respectively) that contains nonzero values.

That is, for a matrix A with elements A_{ij}:

- The upper bandwidth B_1 is the smallest number such that $A_{ij} = 0$ whenever $j - i > B_1$.
- The lower bandwidth B_2 is the smallest number such that $A_{ij} = 0$ whenever $i - j > B_2$.

Note that this measurement does not disallow intermediate diagonals in a band from being all zero, but instead focuses on the location of the last diagonal containing nonzeros. By convention, the upper and lower bandwidths of an empty matrix are both zero.

Tips

- Use the `isbanded` function to test if a matrix is within a specific lower and upper bandwidth.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`diag` | `isbanded` | `isdiag` | `istril` | `istriu`

Introduced in R2014a

bar

Bar graph

Syntax

```
bar(y)
bar(x,y)

bar( ___ ,width)
bar( ___ ,style)
bar( ___ ,color)
bar( ___ ,Name,Value)

bar(ax, ___ )

b = bar( ___ )
```

Description

`bar(y)` creates a bar graph with one bar for each element in `y`. If `y` is an *m*-by-*n* matrix, then `bar` creates *m* groups on page 1-758 of *n* bars.

`bar(x,y)` draws the bars at the locations specified by `x`.

`bar( ___ ,width)` sets the relative bar width, which controls the separation of bars within a group. Specify `width` as a scalar value. Use this option with any of the input argument combinations in the previous syntaxes.

`bar( ___ ,style)` specifies the style of the bar groups. For example, use `'stacked'` to display each group as one multicolored bar.

`bar( ___ ,color)` sets the color for all the bars. For example, use `'r'` for red bars.

`bar( ___ ,Name,Value)` specifies properties of the bar graph using one or more name-value pair arguments. Only bar graphs that use the default `'grouped'` or `'stacked'` style support setting bar properties. Specify the name-value pair arguments after all other input arguments. For a list of properties, see Bar.

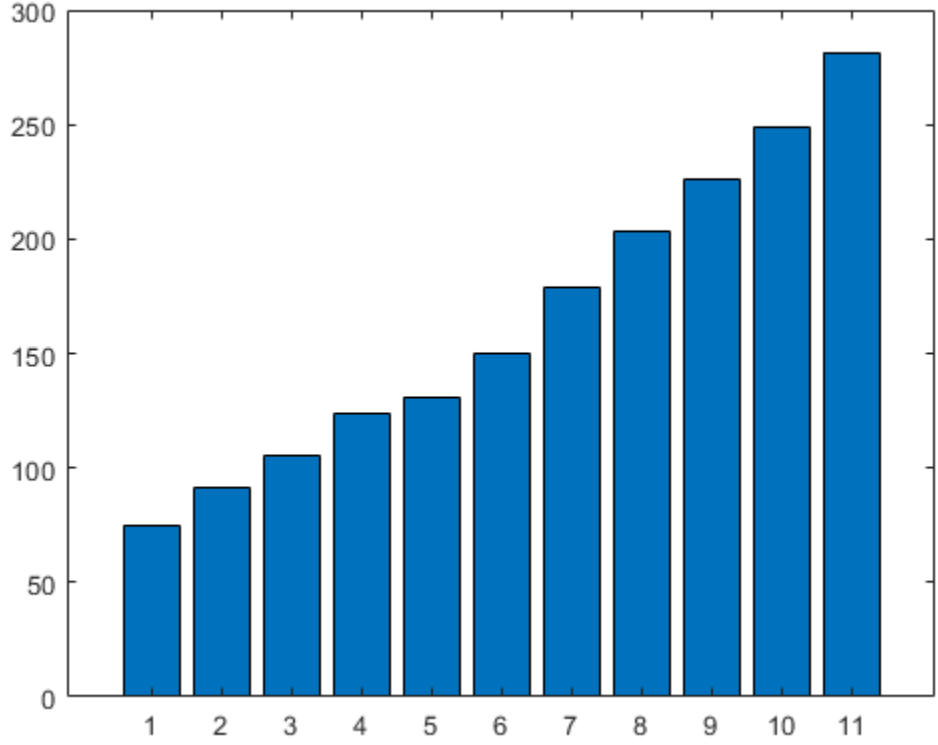
`bar(ax, ___ )` plots into the axes specified by `ax` instead of into the current axes (`gca`). The option `ax` can precede any of the input argument combinations in the previous syntaxes.

`b = bar( ___ )` returns one or more `Bar` objects. If `y` is a vector, then `bar` creates one `Bar` object. If `y` is a matrix, then `bar` returns a `Bar` object for each series on page 1-758. Use `b` to set properties of the bars after displaying the bar graph.

Examples

Create Bar Graph

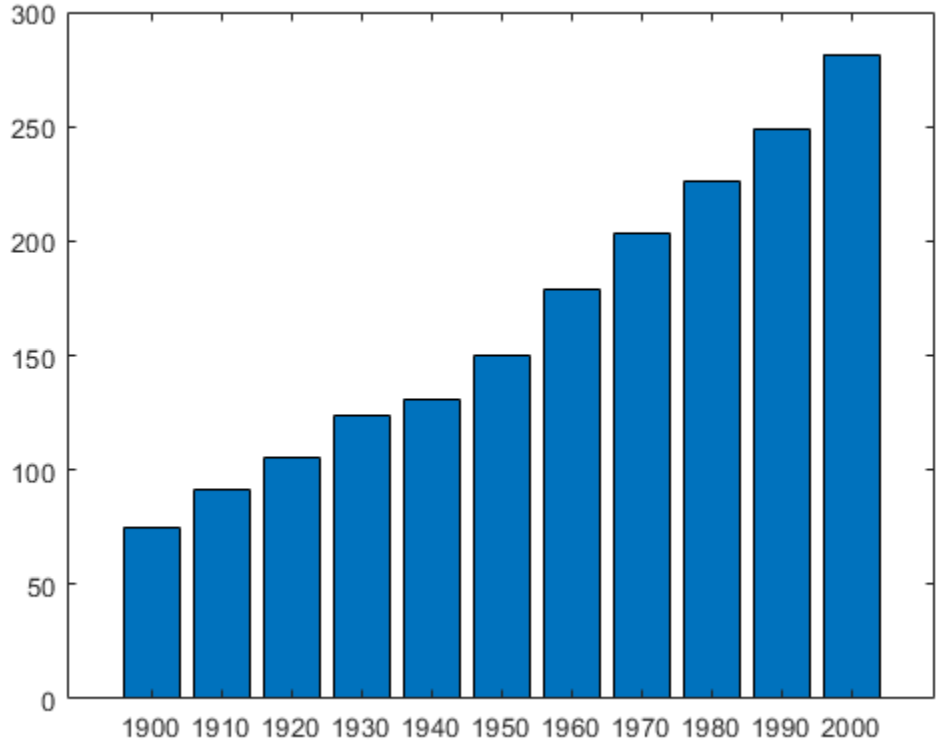
```
y = [75 91 105 123.5 131 150 179 203 226 249 281.5];
bar(y)
```



Specify Bar Locations

Specify the bar locations along the *x*-axis.

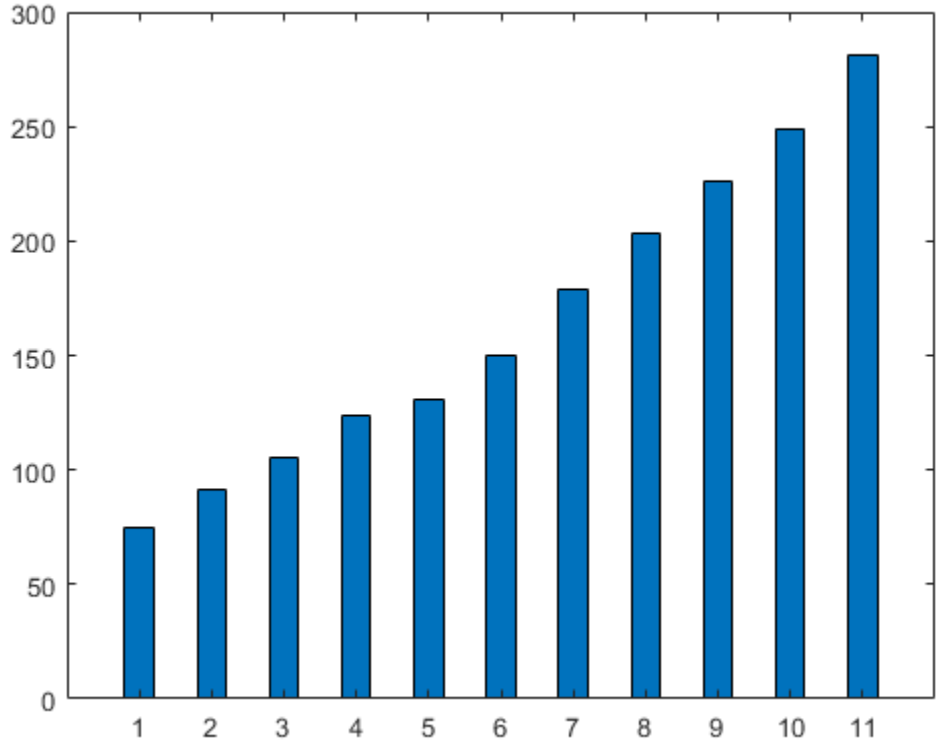
```
x = 1900:10:2000;
y = [75 91 105 123.5 131 150 179 203 226 249 281.5];
bar(x,y)
```



Specify Bar Width

Set the width of each bar to 40 percent of the total space available for each bar.

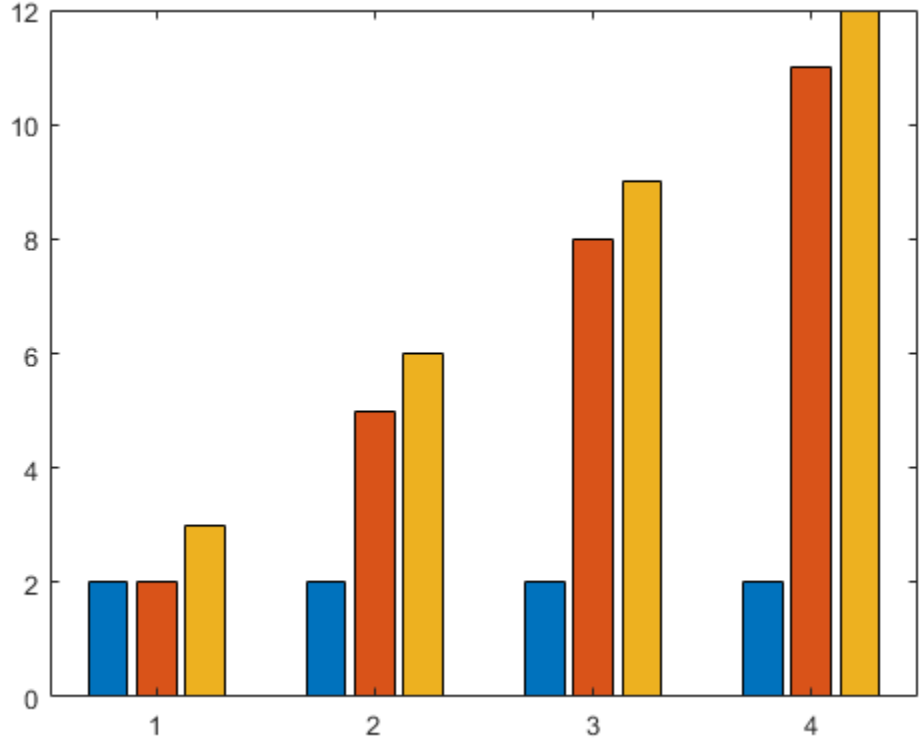
```
y = [75 91 105 123.5 131 150 179 203 226 249 281.5];
bar(y,0.4)
```



Display Groups of Bars

Display four groups of three bars.

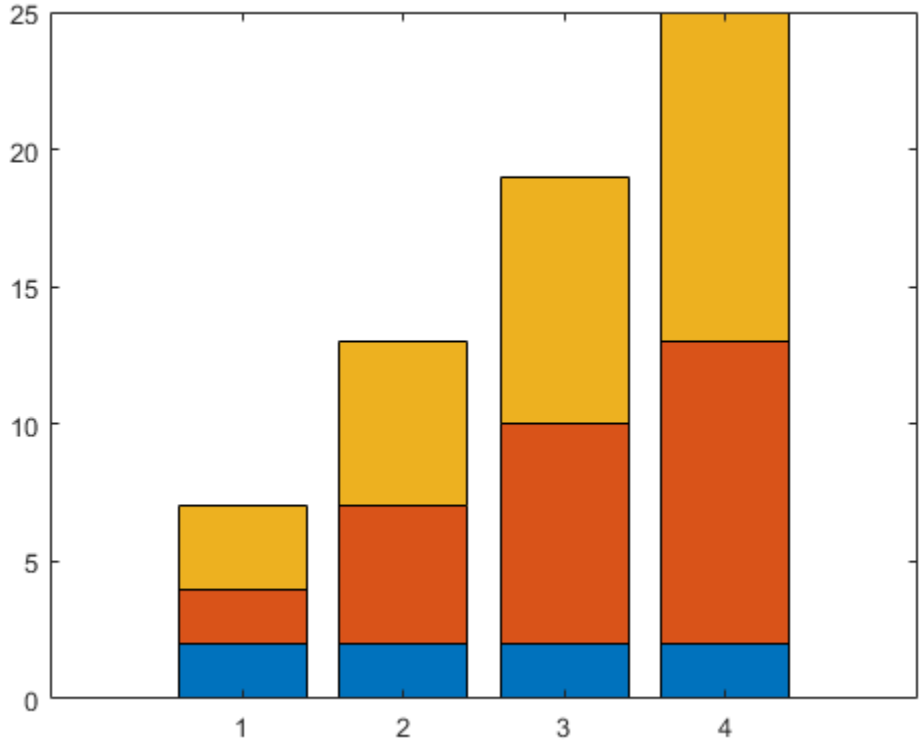
```
y = [2 2 3; 2 5 6; 2 8 9; 2 11 12];
bar(y)
```



Display Stacked Bars

Display one bar for each row of the matrix. The height of each bar is the sum of the elements in the row.

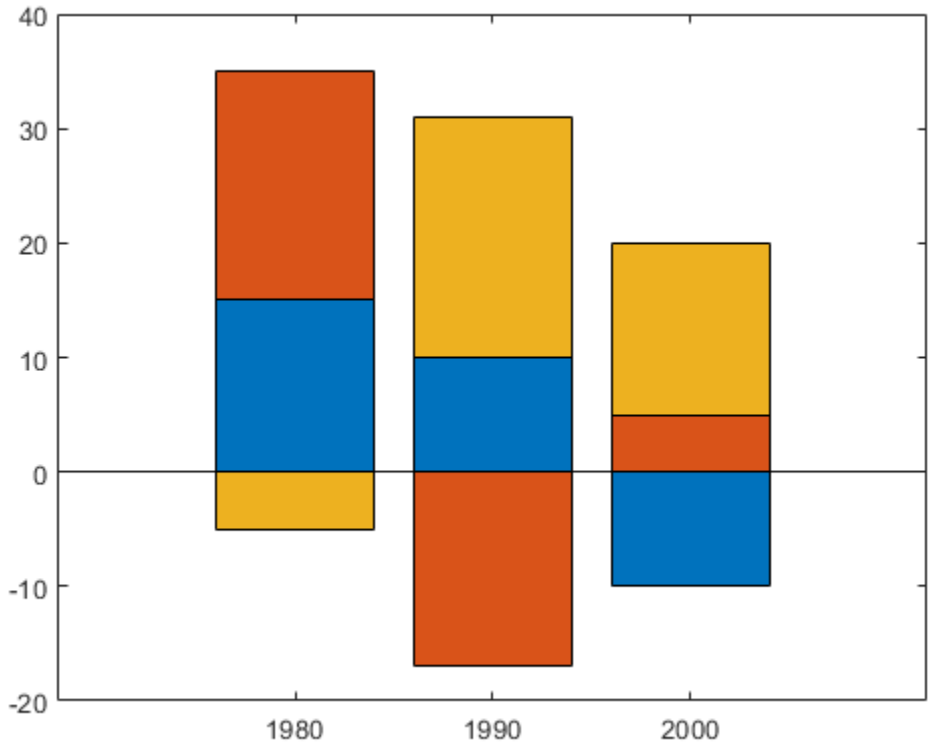
```
y = [2 2 3; 2 5 6; 2 8 9; 2 11 12];
bar(y,'stacked')
```



Display Stacked Bars with Negative Data

Define x as a vector of three year values. Define y as a matrix that contains a combination of negative and positive values. Display the values in a bar graph.

```
x = [1980 1990 2000];
y = [15 20 -5; 10 -17 21; -10 5 15];
bar(x,y,'stacked')
```



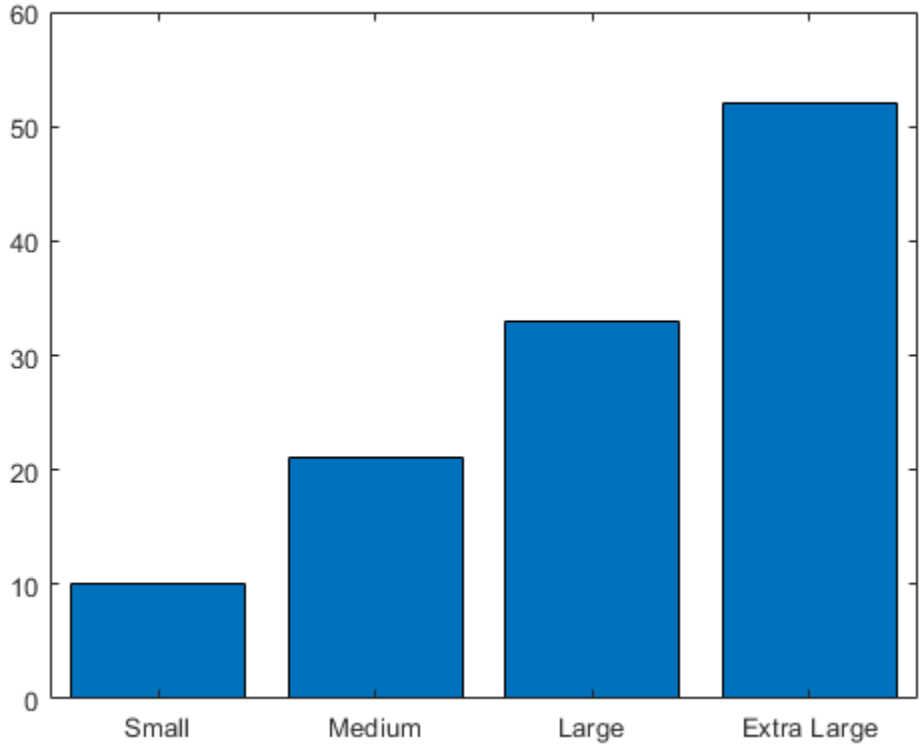
Specify Categorical Data

One way to indicate categories for your bars is to specify X as a categorical array. The `bar` function uses a sorted list of the categories, so the bars might display in a different order than you expect. To preserve the order, call the `reordercats` function.

Define X as categorical array, and call the `reordercats` function to specify the order for the bars. Then define Y as a vector of bar heights and display the bar graph.

```
X = categorical({'Small','Medium','Large','Extra Large'});
X = reordercats(X,{'Small','Medium','Large','Extra Large'});
```

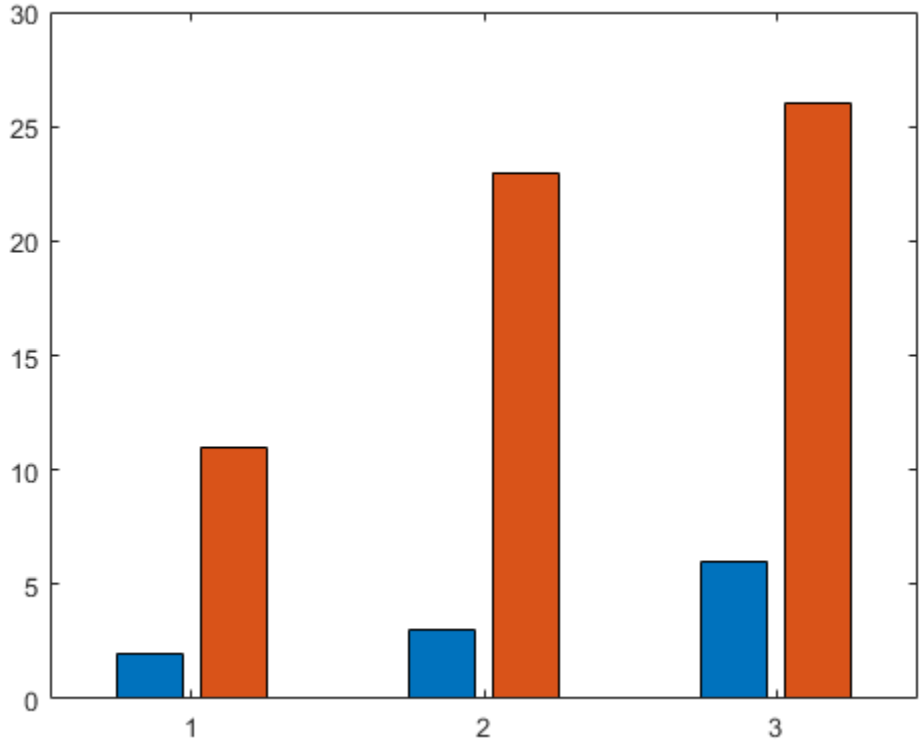
```
Y = [10 21 33 52];
bar(X,Y)
```



Specify Labels at the Ends of Bars

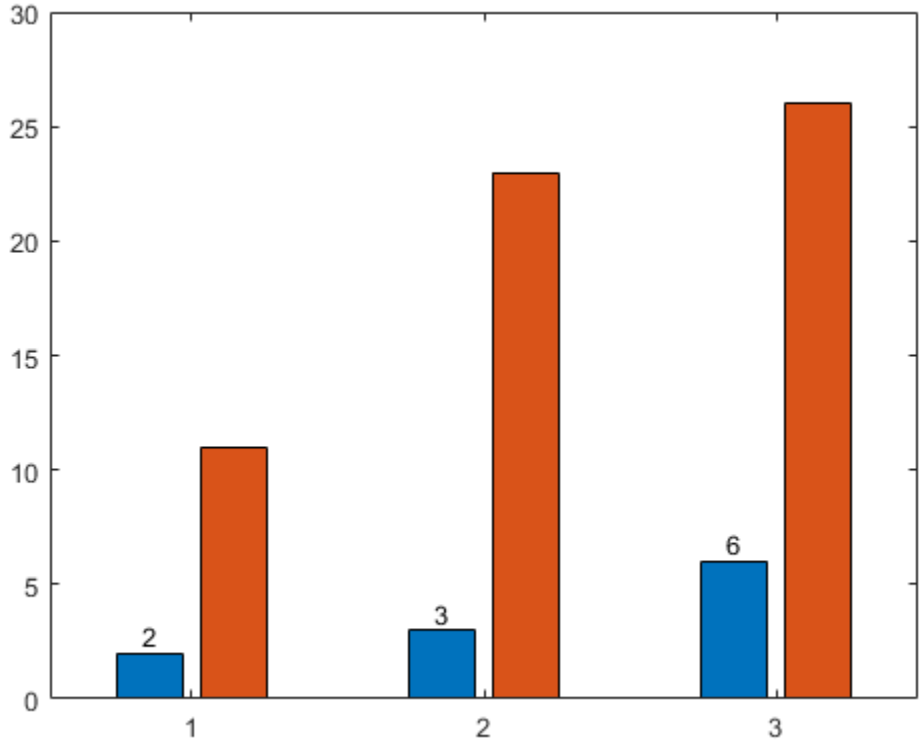
Define `vals` as a matrix containing the values of two data sets. Display the values in a bar graph and specify an output argument. Since there are two data sets, `bar` returns a vector containing two `Bar` objects.

```
x = [1 2 3];
vals = [2 3 6; 11 23 26];
b = bar(x,vals);
```



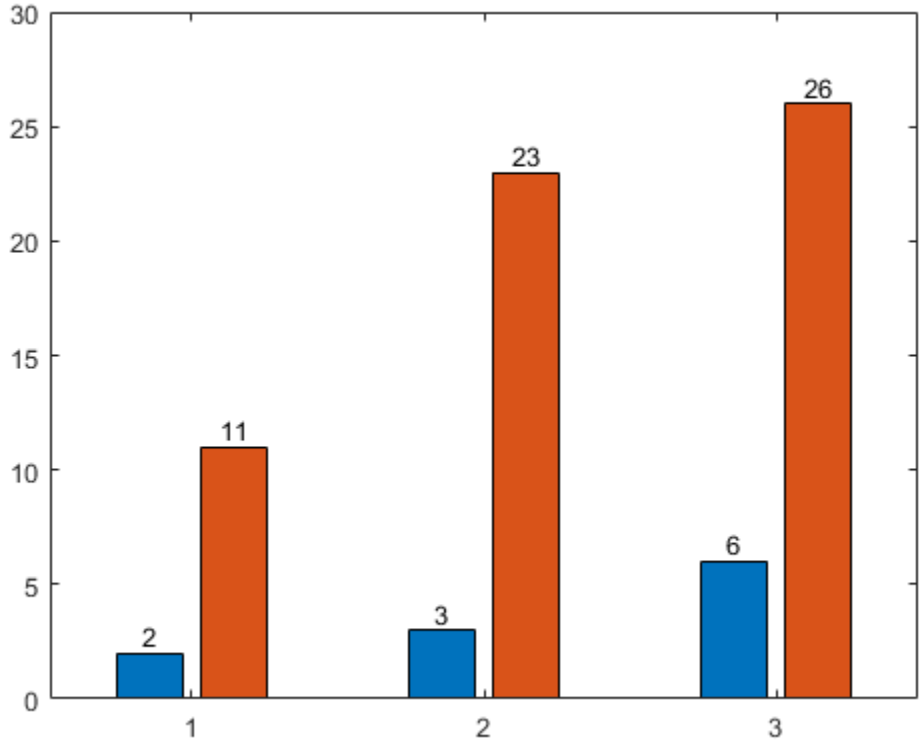
Display the values at the tips of the first series of bars. Get the coordinates of the tips of the bars by getting the `XEndPoints` and `YEndPoints` properties of the first `Bar` object. Pass those coordinates to the `text` function, and specify the vertical and horizontal alignment so that the values are centered above the tips of the bars.

```
xtips1 = b(1).XEndPoints;
ytips1 = b(1).YEndPoints;
labels1 = string(b(1).YData);
text(xtips1,ytips1,labels1,'HorizontalAlignment','center',...
    'VerticalAlignment','bottom')
```



Next, display the values above the tips of the second series of bars.

```
xtips2 = b(2).XEndPoints;
ytips2 = b(2).YEndPoints;
labels2 = string(b(2).YData);
text(xtips2,ytips2,labels2,'HorizontalAlignment','center',...
    'VerticalAlignment','bottom')
```

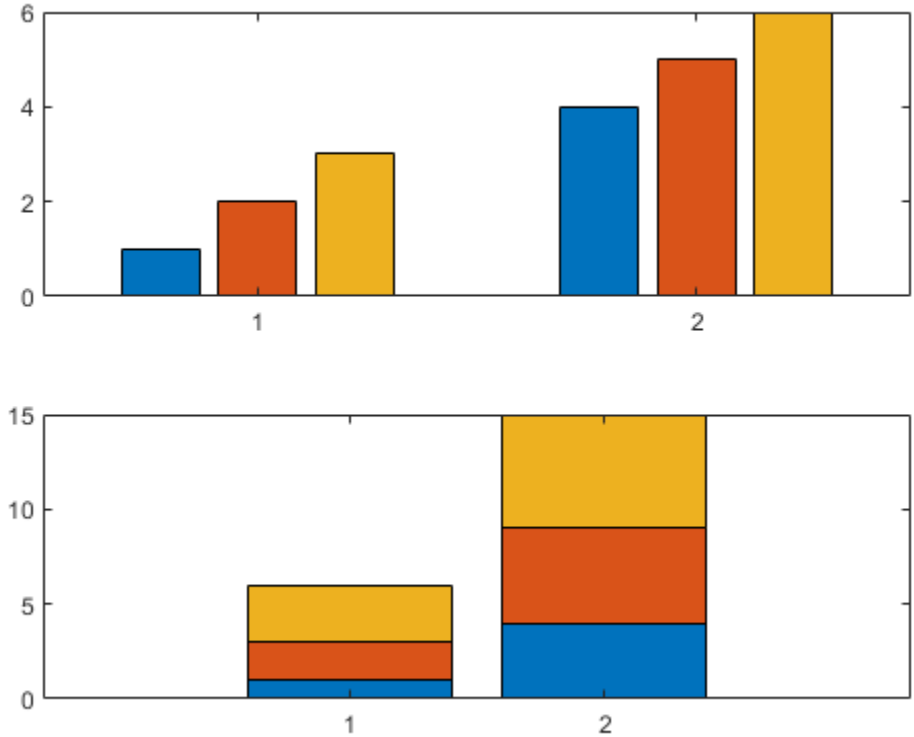


Specify Axes for Bar Graph

Starting in R2019b, you can display a tiling of bar graphs using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Display a bar graph in the top axes. In the bottom axes, display a stacked bar graph of the same data.

```
y = [1 2 3; 4 5 6];
tiledlayout(2,1)

% Top bar graph
```

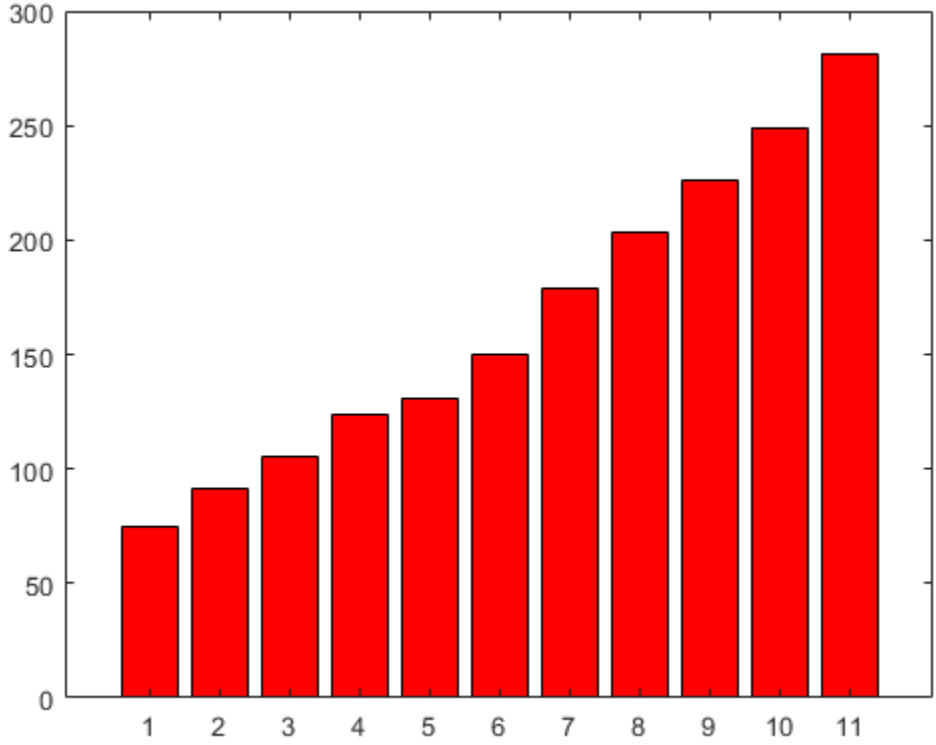
```
ax1 = nexttile;
bar(ax1,y)

% Bottom bar graph
ax2 = nexttile;
bar(ax2,y,'stacked')
```



Specify Bar Color

Create a bar graph using red bars.

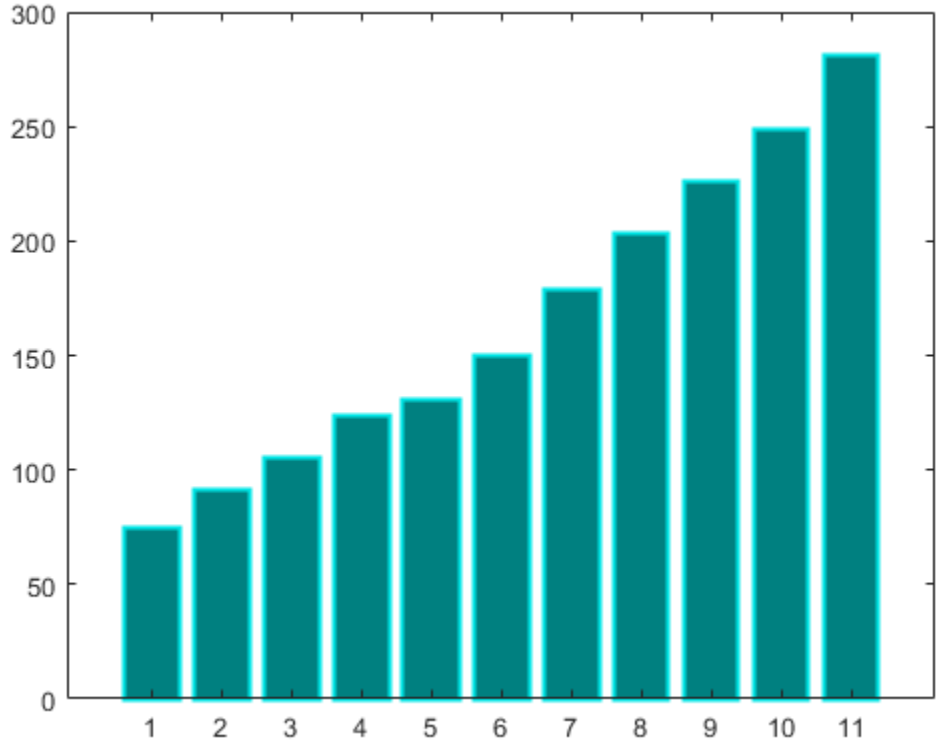
```
y = [75 91 105 123.5 131 150 179 203 226 249 281.5];
bar(y,'r')
```



Specify Bar and Outline Colors

Set the bar interior color and outline color using RGB triplets. Set the width of the bar outline.

```
y = [75 91 105 123.5 131 150 179 203 226 249 281.5];
bar(y,'FaceColor',[0 .5 .5],'EdgeColor',[0 .9 .9],'LineWidth',1.5)
```

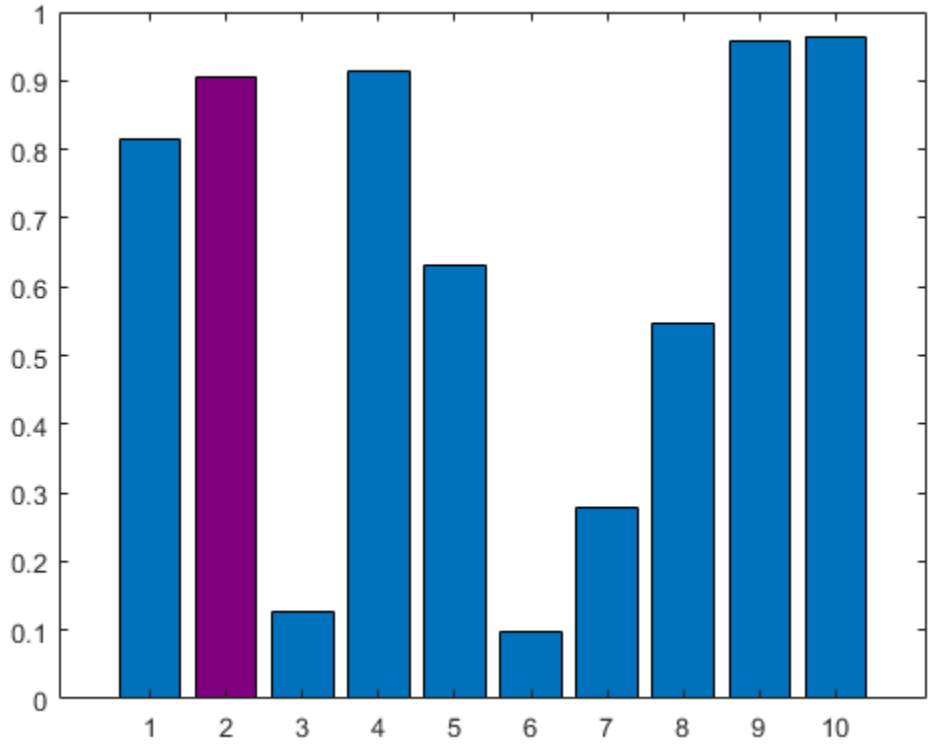


Control Individual Bar Colors

Control individual bar colors using the `CData` property of the `Bar` object.

Create a bar chart and assign the `Bar` object to a variable. Set the `FaceColor` property of the `Bar` object to `'flat'` so that the chart uses the colors defined in the `CData` property. By default, the `CData` property is prepopulated with a matrix of the default RGB color values. To change a particular color, change the corresponding row in the matrix. For example, change the color of the second bar.

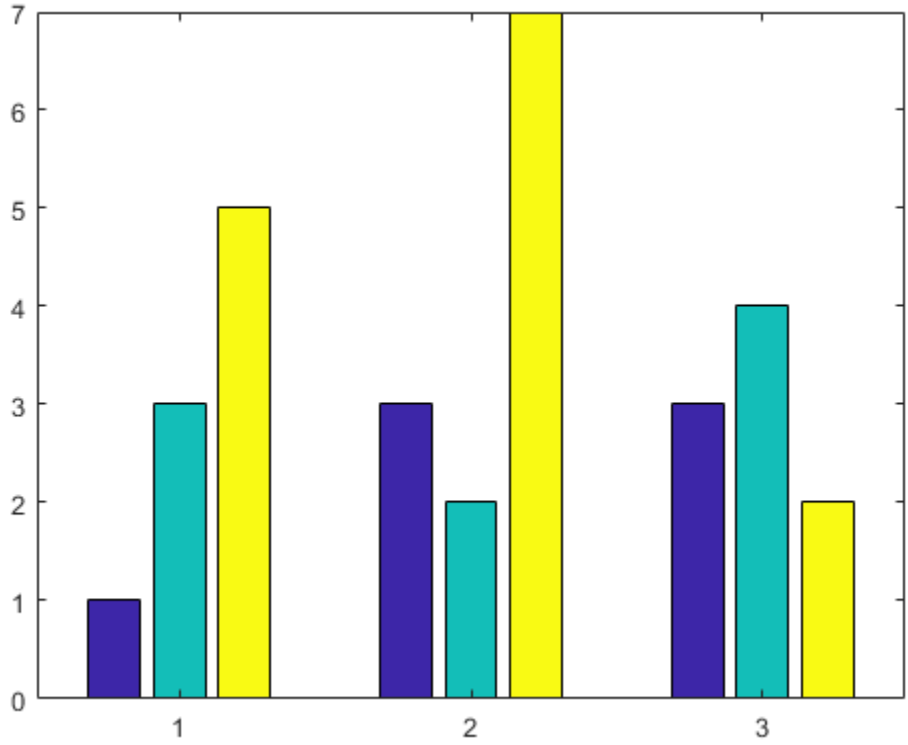
```
b = bar(rand(10,1));
b.FaceColor = 'flat';
b.CData(2,:) = [.5 0 .5];
```



Bar Chart with Colormap Colors

Create a bar chart that uses colormap colors by setting the `FaceColor` property to `'flat'`. Then set the `CData` property for each `Bar` object to an integer.

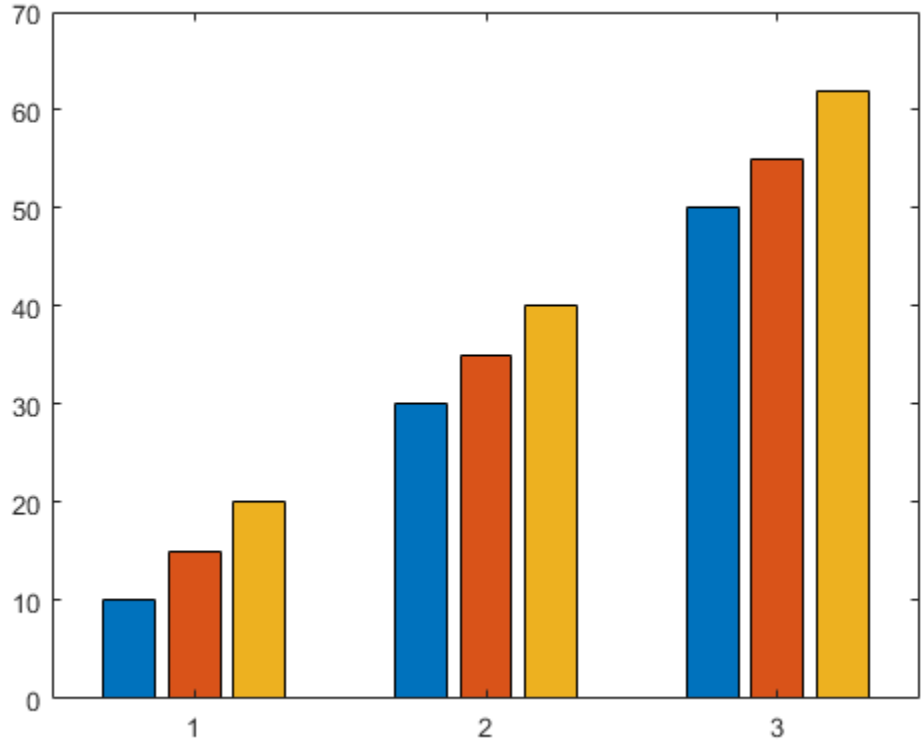
```
y = [1 3 5; 3 2 7; 3 4 2];
b = bar(y,'FaceColor','flat');
for k = 1:size(y,2)
    b(k).CData = k;
end
```



Customize One Series in Grouped or Stacked Bars

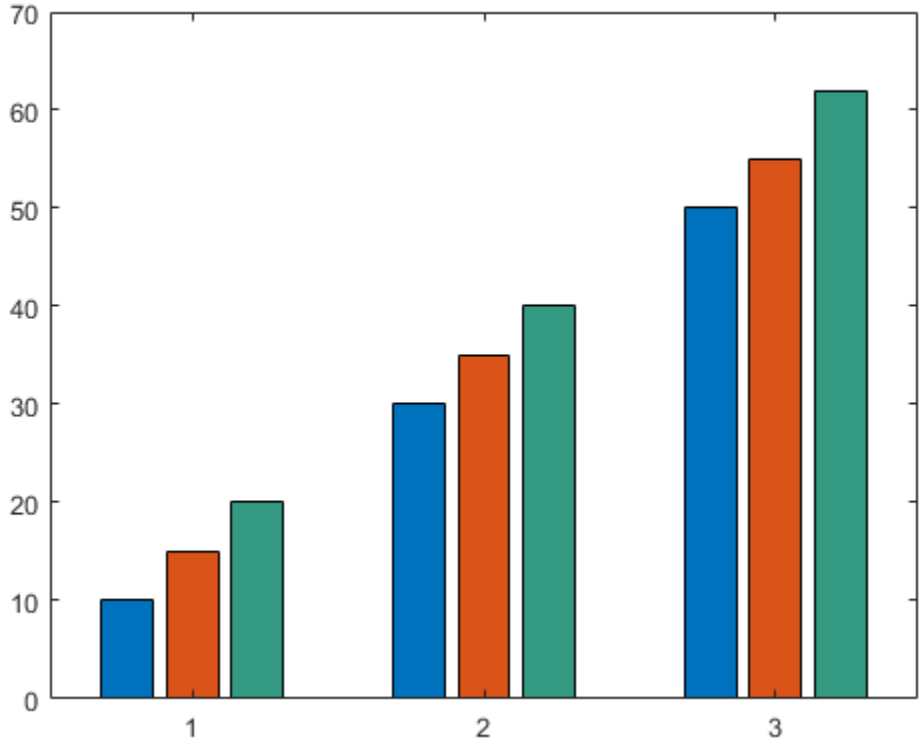
Create matrix `y`, where each column is a series of data. Call the `bar` function to display the data in a bar graph, and specify an output argument. The output is a vector of three `Bar` objects, where each object corresponds to a different series. This is true whether the bars are grouped or stacked.

```
y = [10 15 20; 30 35 40; 50 55 62];
b = bar(y);
```



Make the third series of bars green.

```
b(3).FaceColor = [.2 .6 .5];
```

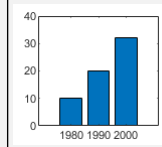


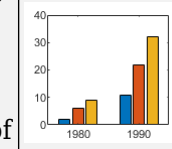
Input Arguments

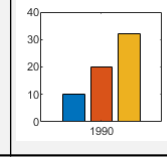
x — *x*-coordinates
scalar | vector | matrix

x-coordinates, specified as a scalar, vector, or matrix. The values of x do not need to be in order, but the size of x depends on the size of y and how you want to display your data. This table describes the most common situations.

Presentation	How to Specify X and Y	Example

Display one series of bars.	Specify x and y as vectors that are the same length. The values in x must be unique, but the values in y do not need to be unique.	`x = [1980 1990 2000];`
		`y = [10 20 30];`
		`bar(x,y)`
		 

Display multiple series of bars in groups.	Specify either of these combinations:
	• Specify x and y as matrices of equal size. Each column of y corresponds to a series of bars. By default, each series is a different color. To ensure consistent placement of the groups, specify the columns of x as identical vectors. The values within a column must be unique, even though the columns are repeated.
	• Specify x as a vector of unique values, and specify y as a matrix. The length of x must equal the length of at least one dimension of y. The other dimension of y contains values for the different series of bars.
		`x = [1980 1980 1980`
		`  1990 1990 1990];`
		`y = [2 6 9`
		`  11 22 32];`
		`bar(x,y)`
		Or
		`x = [1980 1990];`
		`y = [2 6 9`
		`  11 22 32];`
		`bar(x,y)`
		 

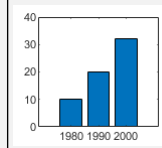
Display one group of bars centered at one x value.	Specify x as a scalar and y as a vector.	`x = 1990;`
		`y = [10 20 30];`
		`bar(x,y)`
		 

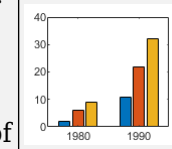
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

y — *y*-coordinates
scalar | vector | matrix

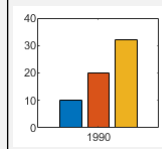
y-coordinates, specified as a scalar, vector, or matrix. The size of `y` depends on the size of `x` and how you want to display your data. This table describes the most common situations.

Presentation	How to Specify X and Y	Example

Display one series of bars.	Specify `x` and `y` as vectors that are the same length. The values in `x` must be unique, but the values in `y` do not need to be unique.	`x = [1980 1990 2000];`
		`y = [10 20 30];`
		`bar(x,y)`
		 

Display multiple series of bars in groups.	Specify either of these combinations:
	• Specify `x` and `y` as matrices of equal size. Each column of `y` corresponds to a series of bars. By default, each series is a different color. To ensure consistent placement of the groups, specify the columns of `x` as identical vectors. The values within a column must be unique, even though the columns are repeated.
	• Specify `x` as a vector of unique values, and specify `y` as a matrix. The length of `x` must equal the length of at least one dimension of `y`. The other dimension of `y` contains values for the different series of bars.
		`x = [1980 1980 1980`
		`  1990 1990 1990];`
		`y = [2 6 9`
		`  11 22 32];`
		`bar(x,y)`
		Or
		`x = [1980 1990];`
		`y = [2 6 9`
		`  11 22 32];`
		`bar(x,y)`
		 

Presentation	How to Specify X and Y	Example

Display one group of bars centered at one x value.	Specify x as a scalar and y as a vector.	`x = 1990;`
		`y = [10 20 30];`
		`bar(x,y)`
		 

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`width` — Bar width
`0.8` (default) | scalar

Bar width, specified as a fraction of the total space available for each bar. The default of `0.8` means the bar width is 80% of the space from the previous bar to the next bar, with 10% of that space on each side.

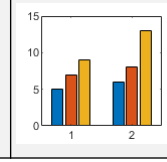
If the width is `1`, then the bars within a group touch one another.

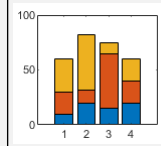
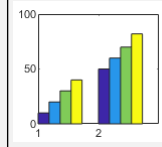
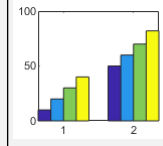
Example: `bar([1 2 3],0.5)` creates bars that use 50% of the available space.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`style` — Group style
`'grouped'` (default) | `'stacked'` | `'hist'` | `'histc'`

Group style, specified by one of these values.

Style	Result	Example
`'grouped'`	Display each group as adjacent bars that are centered around their corresponding x value.	

Style	Result	Example
`'stacked'`	Display each group as one multicolored bar. The length of a bar is the sum of the elements in the group. If `y` is a vector, then the result is the same as `'grouped'`.	
`'histc'`	Display the bars in histogram format, in which the bars in a group touch one another. The trailing edge of each group is aligned with the corresponding `x` value. **Note** A better way to display a histogram is to call the `histogram` function.	
`'hist'`	Display the bars in histogram format. Each group is centered at the corresponding `x` value. **Note** A better way to display a histogram is to call the `histogram` function.	

`color` — Bar color

`'b'` | `'r'` | `'g'` | `'c'` | `'m'` | `'y'` | `'k'` | `'w'`

Bar color, specified as one of the colors in this table.

Color	Color
`'b'`	Blue

Color	Color
`'r'`	Red
`'g'`	Green
`'c'`	Cyan
`'m'`	Magenta
`'y'`	Yellow
`'k'`	Black
`'w'`	White

ax — Axes object
axes object

Axes object. If you do not specify an axes, then `bar` uses the current axes for the bar graph.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The `Bar` properties listed here are only a subset. For a complete list, see Bar.

Example: `bar([10 20 30],'EdgeColor','g')` specifies a green outline around the bars.

Note

- The properties listed here are only a subset. For a complete list, see Bar.
- You can set these properties only on bar graphs that use the default `'grouped'` or `'stacked'` style.

`EdgeColor` — Outline color
`'flat'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Outline color, specified as `'flat'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. If there are 150 bars or fewer, the default value is `[0 0 0]`, which corresponds to black. If there are more than 150 adjacent bars, the default value is `'none'`.

Starting in R2017b, the `'flat'` option uses the `CData` values to color the edges. In previous releases, the `'flat'` option colored the edges using colors from the colormap.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `b = bar(1:10,'EdgeColor','red')`

Example: `b.EdgeColor = [0 0.5 0.5];`

Example: `b.EdgeColor = 'flat';`

Example: `b.EdgeColor = '#D2F9A7';`

`FaceColor` — Fill color

`'flat'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Fill color, specified as `'flat'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'flat'` option uses the `CData` property value of the `Bar` object to color the faces.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Starting in R2017b, the default value is an RGB triplet from the `ColorOrder` property of the axes. In previous releases, the default value was `'flat'` and the colors were based on the colormap.

Example: `b = bar(1:10,'FaceColor','red')`

Example: `b.FaceColor = [0 0.5 0.5];`

Example: `b.FaceColor = 'flat';`

Example: `b.FaceColor = '#D2F9A7';`

CData — Color data

RGB triplet | three-column matrix | scalar | vector

Color data, specified as one of these values:

- RGB triplet — Single RGB color value applies to all bars.
- Three-column matrix — One color per bar. Each row in the matrix specifies an RGB triplet for a particular bar.
- Scalar — Single color applies to all bars, where the color comes from the colormap.
- Vector — One color per bar. The colors come from the colormap.

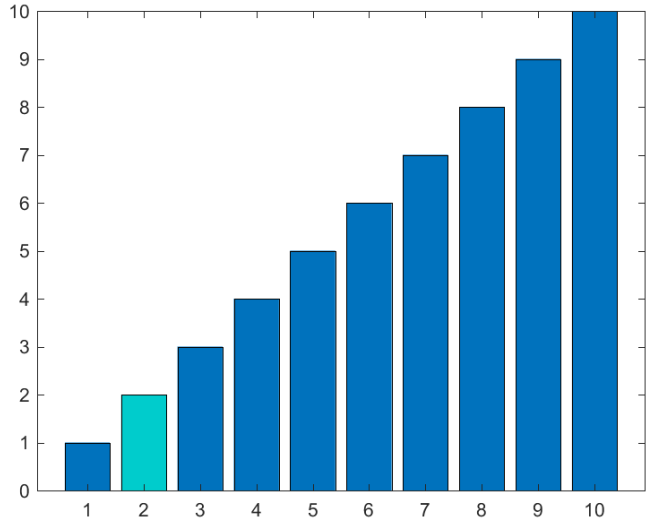
By default, when you create a bar chart, the `CData` property contains a three-column matrix of RGB triplets. You can change the color for a particular bar by changing the corresponding row in the matrix.

This property applies only when the `FaceColor` or `EdgeColor` property is set to `'flat'`.

Example

Change the color for a particular bar by setting the `FaceColor` property to `'flat'`. Then change the corresponding row in the `CData` matrix to the new RGB triplet. For example, change the color of the second bar.

```
b = bar(1:10,'FaceColor','flat');
b.CData(2,:) = [0 0.8 0.8];
```



`BaseValue` — Baseline value
`0` (default) | numeric scalar value

Baseline value, specified as a numeric scalar value.

The baseline value that you specify applies to either the *x*-axis or the *y*-axis depending on the bar chart orientation. If you change the orientation of the bar chart from vertical to horizontal, or vice versa, the baseline value might change. Set the `BaseValue` property after setting the `Horizontal` property.

`LineStyle` — Line style of bar outlines
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style of bar outlines, specified as one of the line styles in this table.

Line Style	Line Style	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	

Line Style	Line Style	Resulting Line
`':'`	Dotted line	.....
`'-.'`	Dash-dotted line	_._._._.-
`'none'`	No line	No line

`LineWidth` — Width of bar outlines
`0.5` (default) | positive value

Width of bar outlines, specified as a positive value in point units. One point equals 1/72 inch.

Example: `1.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`b` — `Bar` objects
`Bar` objects

`Bar` objects. Use the elements in `b` to access and modify properties of a specific `Bar` object after it has been created. The number of `Bar` objects depends on the size of `y`. If `y` is a vector, then `b` is one `Bar` object. If `y` is a matrix, then `b` is a vector containing a `Bar` object for each series in `y`.

More About

Series of Bars

A series consists of the bars at all locations in `X` for a particular set of data. By default, each series of bars is indicated by a different color.

Group of Bars

A group consists of all the bars at a particular location in `X`.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions

`bar3` | `bar3h` | `barh` | `histogram` | `hold` | `stairs`

Properties

Bar

Topics

“Modify Baseline of Bar Graph”
“Overlay Bar Graphs”
“Plot Dates and Durations”

"Plot Categorical Data"

External Websites

MATLAB Plot Gallery

Introduced before R2006a

barh

Horizontal bar graph

Syntax

```
barh(y)
barh(x,y)
barh( ___ ,width)
barh( ___ ,style)
barh( ___ ,color)
barh( ___ ,Name,Value)
barh(ax, ___ )
b = barh( ___ )
```

Description

`barh(y)` creates a horizontal bar graph with one bar for each element in `y`. If `y` is an *m*-by-*n* matrix, then `barh` creates *m* groups on page 1-785 of *n* bars.

`barh(x,y)` draws the bars along the vertical axis at the locations specified by `x`.

`barh( ___ ,width)` specifies the fraction of available space occupied by each bar. For example, `barh(y,1)` makes the bars in each group touch each other. Specify `width` as the last argument in any of the previous syntaxes.

`barh( ___ ,style)` specifies the style of the bar groups. For example, `barh(y,'stacked')` stacks the bars within each group into a multicolored bar.

`barh( ___ ,color)` specifies a single color for all the bars. For example, `barh(y,'red')` displays red bars.

`barh( ___ ,Name,Value)` specifies properties of the bar graph using one or more name-value pair arguments. Only bar graphs that use the default `'grouped'` or `'stacked'` style support setting bar properties. Specify the name-value pair arguments after all other input arguments. For a list of properties, see Bar.

`barh(ax, ___ )` displays the bar graph in the target axes. Specify the axes as the first argument in any of the previous syntaxes.

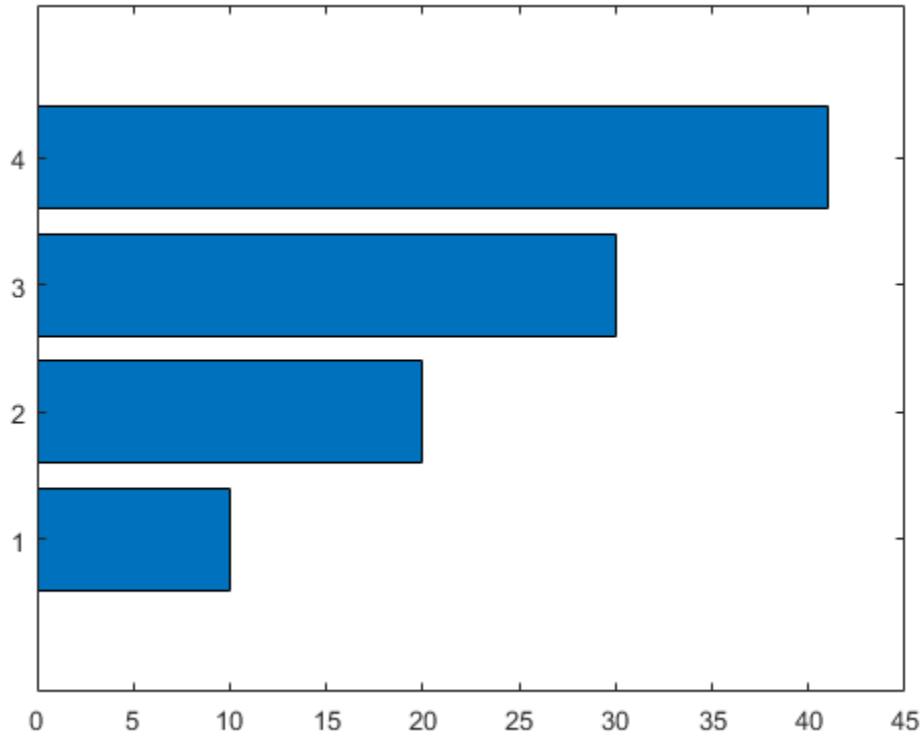
`b = barh( ___ )` returns one or more `Bar` objects. If `y` is a vector, then `barh` returns one `Bar` object. If `y` is a matrix, then `barh` returns a `Bar` object for each series on page 1-785. Use `b` to set properties of the bars after displaying the bar graph.

Examples

Display One Series of Bars

Create a vector of four values. Display the values in a bar graph with one horizontal bar for each value.

```
y = [10 20 30 41];
barh(y)
```

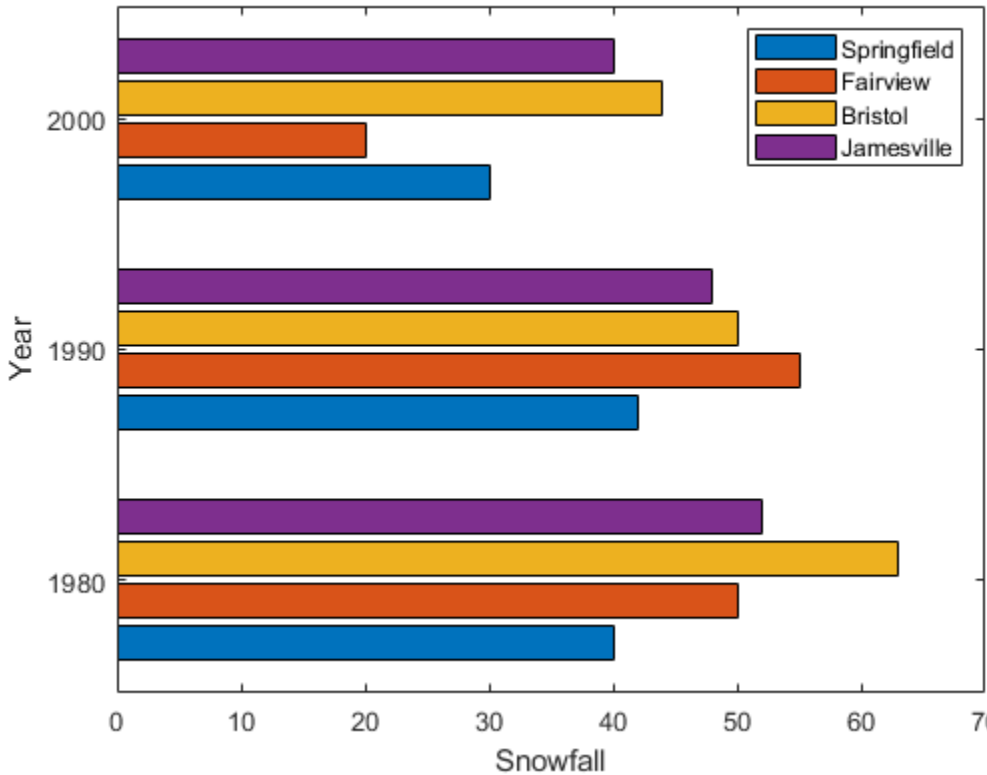


Display Four Series of Bars with Axis Labels and Legend

Define x as a matrix of three year values. Define y as a matrix containing snowfall data for four cities. Display the four series of bars in groups for each year. Then add the axis labels and a legend.

```
x = [1980 1990 2000];
y = [40 50 63 52; 42 55 50 48; 30 20 44 40];
barh(x,y)
xlabel('Snowfall')
```

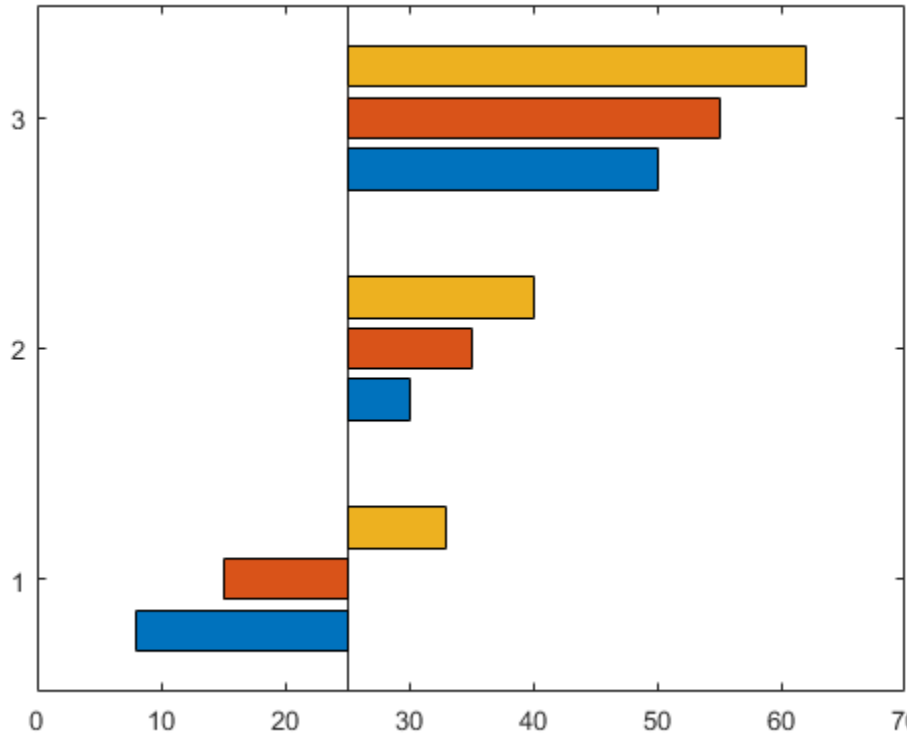
```
ylabel('Year')
legend({'Springfield','Fairview','Bristol','Jamesville'})
```



Change Baseline Value

Create matrix y. Then display the values of y in a bar graph with a baseline value of 25. Values that are less than 25 display on the left side of the baseline.

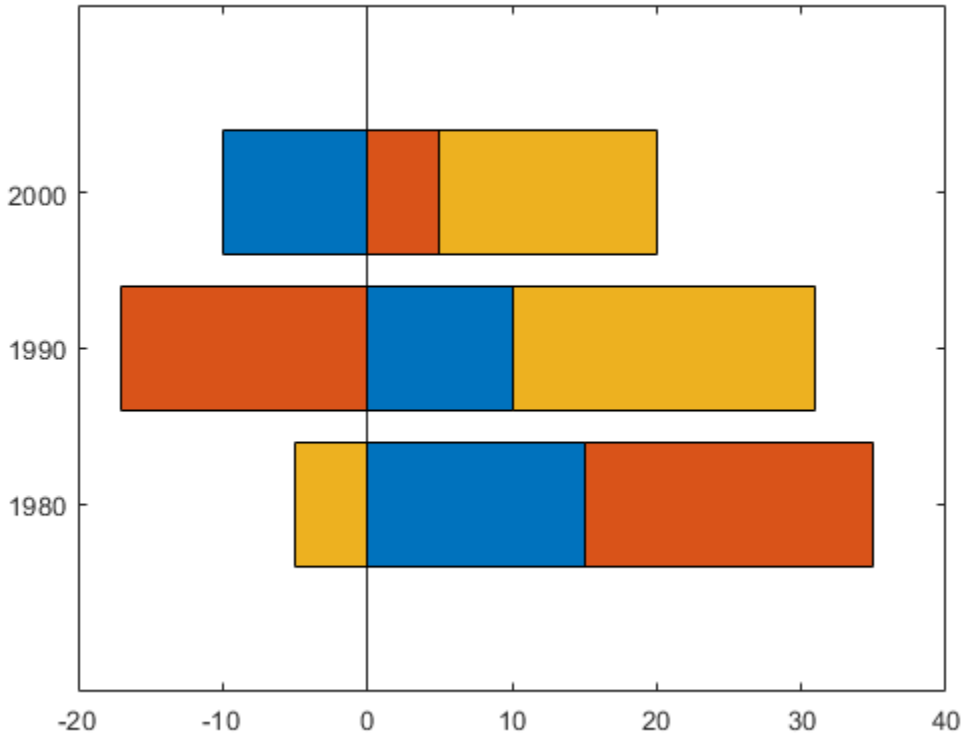
```
y = [8 15 33; 30 35 40; 50 55 62];
barh(y,'BaseValue',25)
```



Display Stacked Bars with Negative Data

Define x as a vector of three year values. Define y as a matrix that contains a combination of negative and positive values. Display the values in a stacked horizontal bar graph.

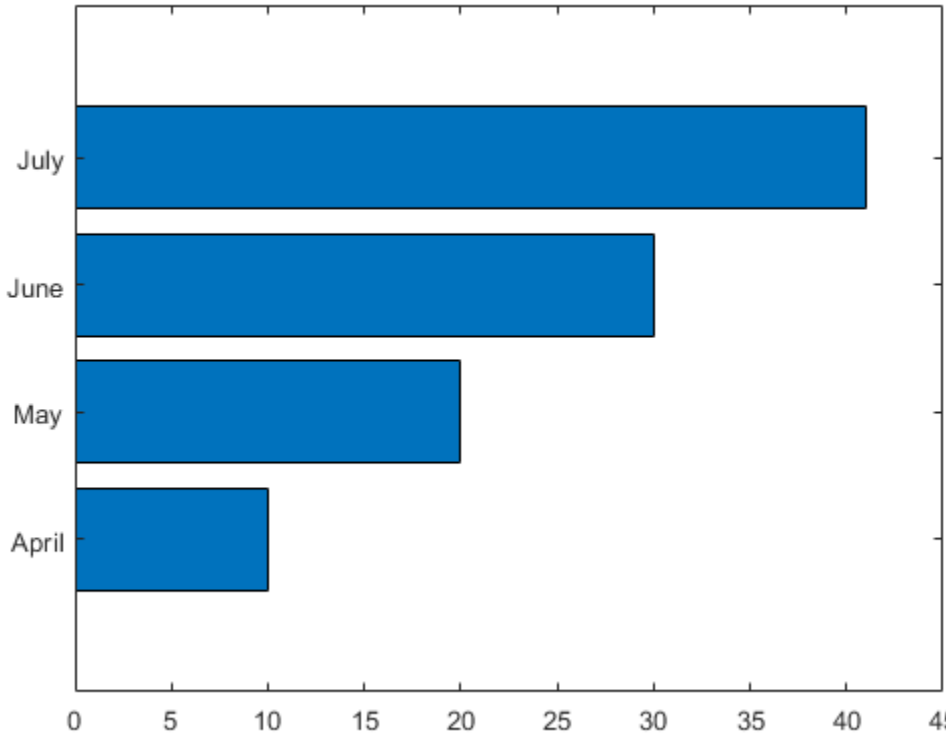
```
x = [1980 1990 2000];
y = [15 20 -5; 10 -17 21; -10 5 15];
barh(x,y,'stacked')
```



Customize Vertical Axis Tick Labels

Define `y` as a vector of four values, and display the values in a horizontal bar graph. Then call the `yticklabels` function to change the tick labels on the vertical axis.

```
y = [10 20 30 41];
barh(y)
yticklabels({'April','May','June','July'})
```



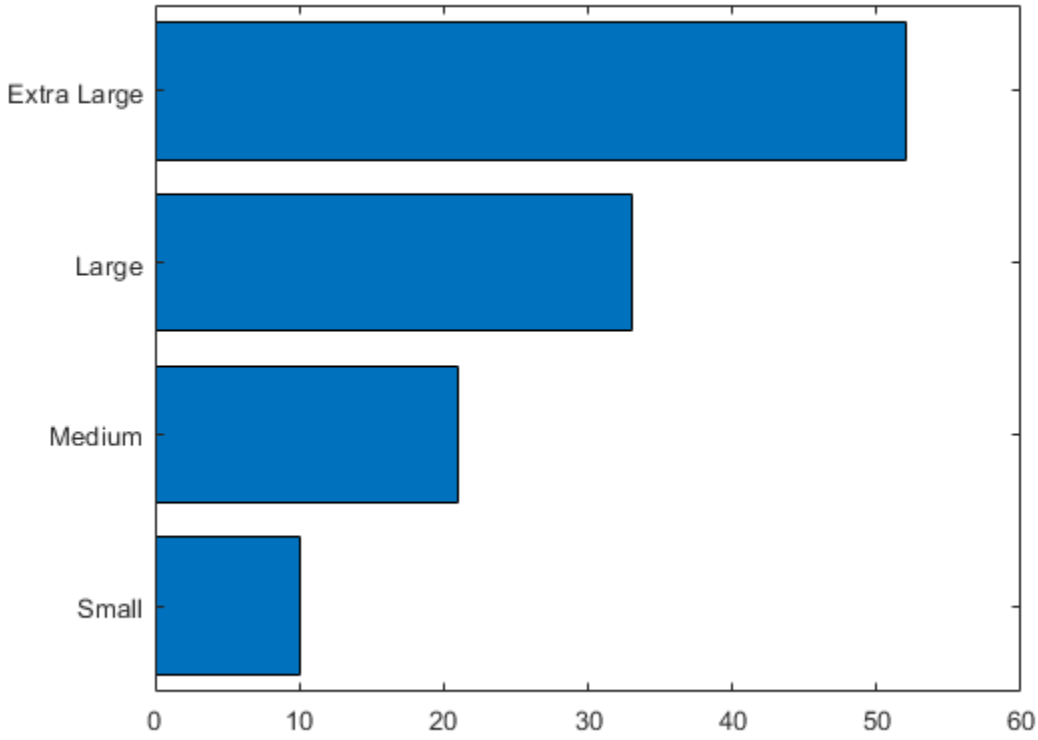
Specify Categorical Data

One way to indicate categories for your bars is to specify X as a categorical array. The `barh` function uses a sorted list of the categories, so the bars might display in a different order than you expect. To preserve the order, call the `reordercats` function.

Define X as categorical array, and call the `reordercats` function to specify the order for the bars. Then define Y as a vector of bar lengths and display the bar graph.

```
X = categorical({'Small','Medium','Large','Extra Large'});
X = reordercats(X,{'Small','Medium','Large','Extra Large'});
```

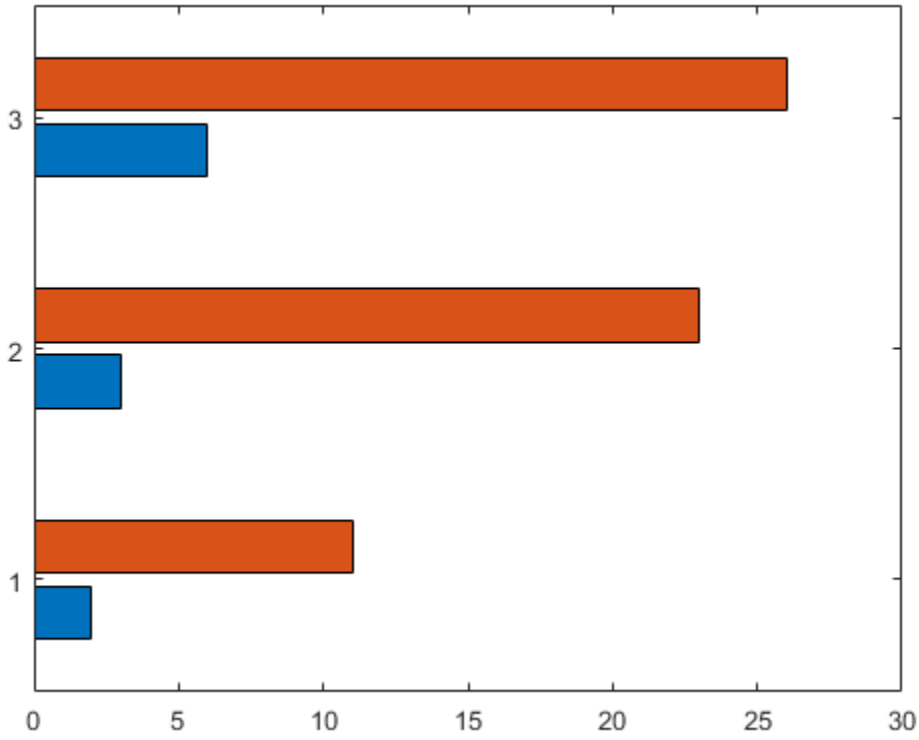
```
Y = [10 21 33 52];
barh(X,Y)
```



Add Labels to the Ends of Bars

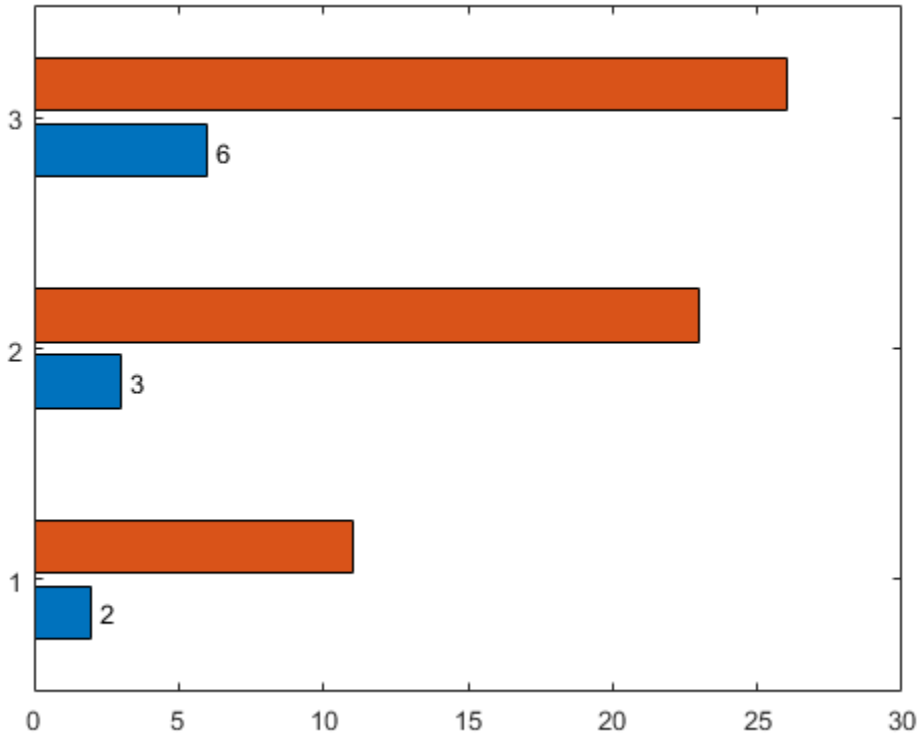
Define `vals` as a matrix containing two series of data. Display the data in a horizontal bar graph and specify an output argument. Since there are two series, `barh` returns a vector of two `Bar` objects.

```
x = [1 2 3];
vals = [2 3 6; 11 23 26];
b = barh(x,vals);
```



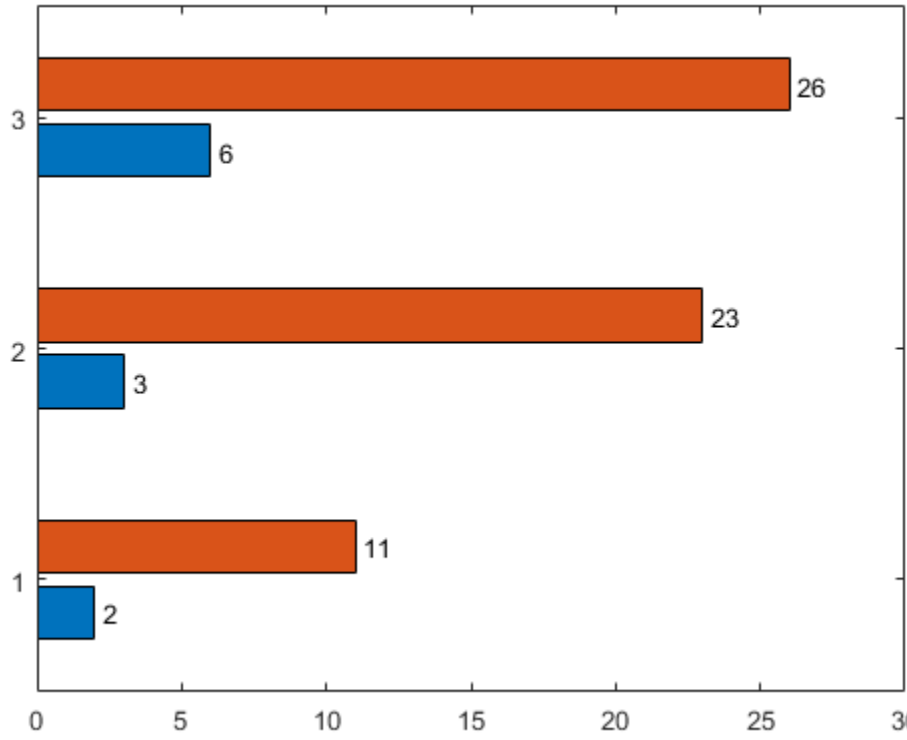
Display the values as labels at the tips of the first series of bars. To do this, get the coordinates of the tips of the bars by getting the `XEndPoints` and `YEndPoints` properties of the first `Bar` object. Since horizontal bar graphs have rotated axes, you must switch the values of `XEndPoints` and `YEndPoints` before passing them to the `text` function. Add a padding value of `0.3` to `YEndpoints` so that the text does not touch the edges of the bars. Then call the `text` function to display the labels.

```
xtips1 = b(1).YEndPoints + 0.3;
ytips1 = b(1).XEndPoints;
labels1 = string(b(1).YData);
text(xtips1,ytips1,labels1,'VerticalAlignment','middle')
```



Next, follow the same steps to display the labels at the tips of the second series of bars.

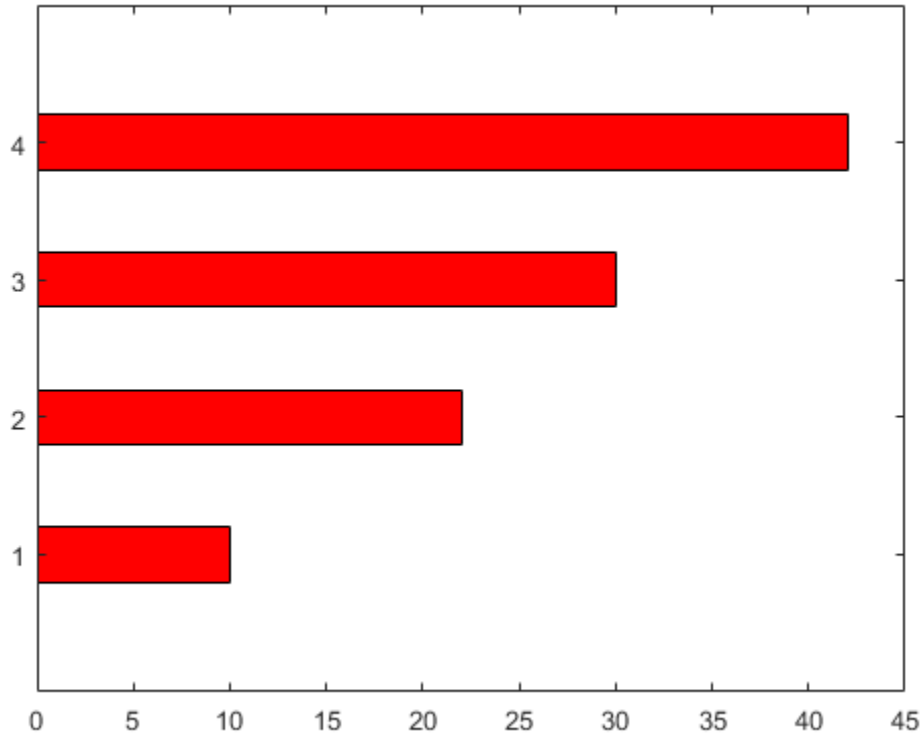
```
xtips2 = b(2).YEndPoints + 0.3;
ytips2 = b(2).XEndPoints;
labels2 = string(b(2).YData);
text(xtips2,ytips2,labels2,'VerticalAlignment','middle')
```



Specify Bar Thickness and Color

Create and display data in a horizontal bar graph that has red bars with a thickness of `0.4`.

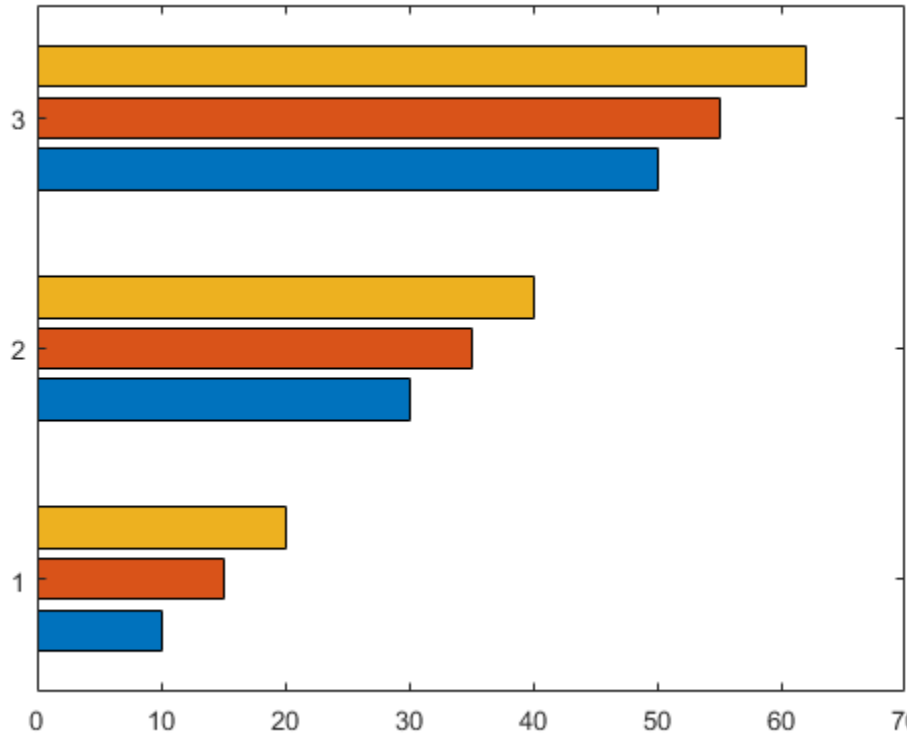
```
y = [10 22 30 42];
width = 0.4;
barh(y,width,'red');
```



Customize One Series of Bars

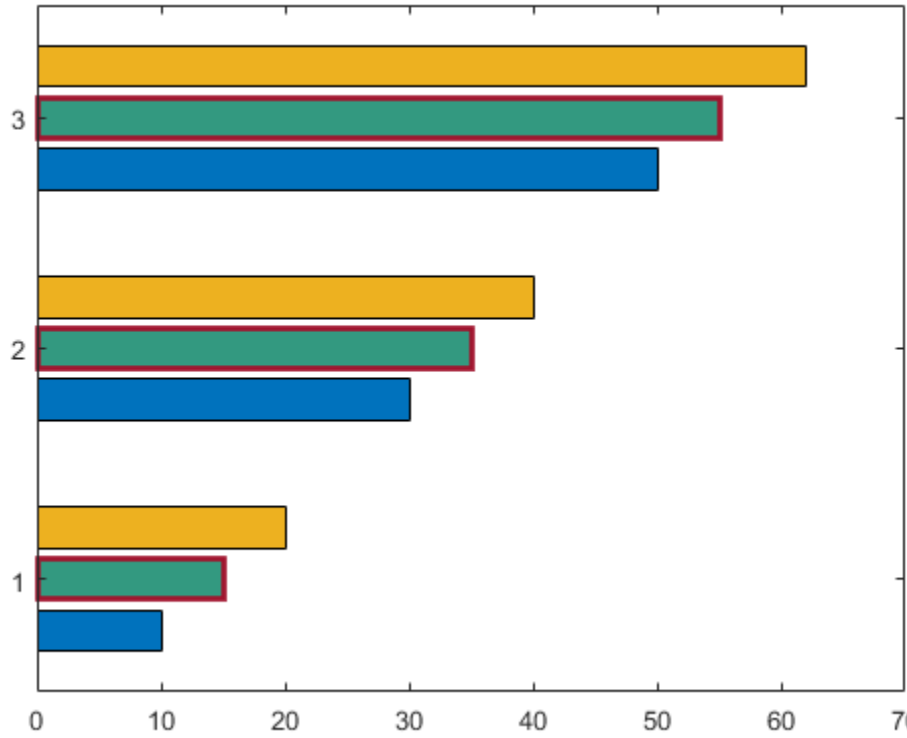
Create matrix `y` in which each column is a series of data. Then display the data in a bar graph, specifying an output argument when calling the `barh` function. In this case, `barh` returns a vector of three `Bar` objects. Each object corresponds to a different series.

```
y = [10 15 20; 30 35 40; 50 55 62];
b = barh(y);
```



Modify the second series of bars to be green with thick red edges.

```
b(2).FaceColor = [.2 .6 .5];
b(2).EdgeColor = [.63 .08 .18];
b(2).LineWidth = 2;
```

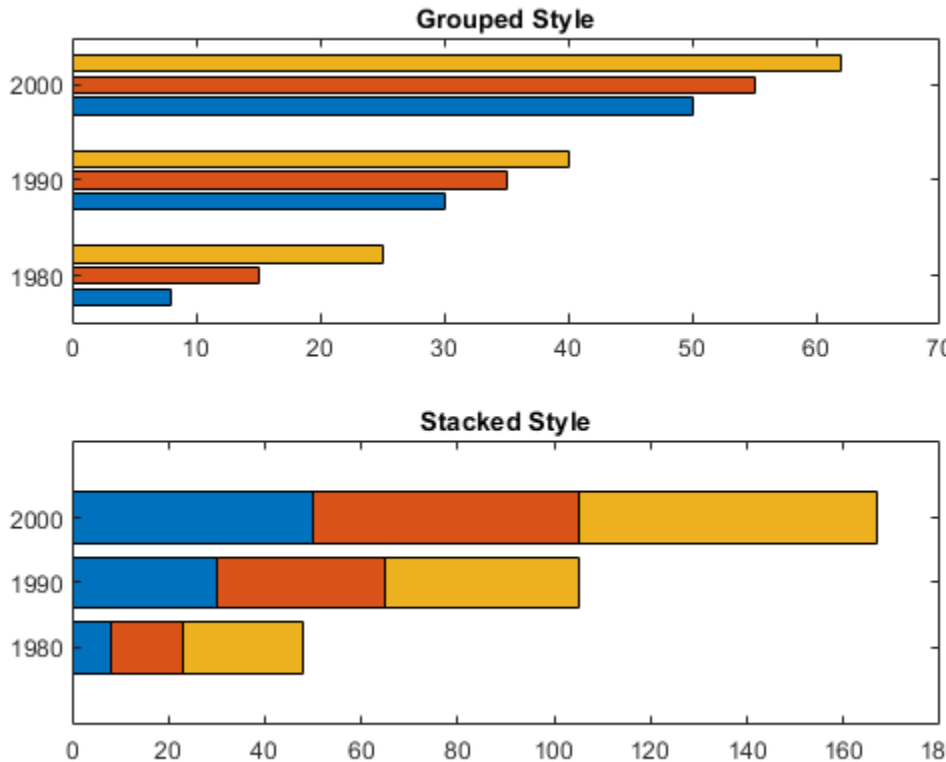


Compare Different Bar Styles

Create and display data in two different bar graphs using the default `'grouped'` style and the `'stacked'` style.

```
x = [1980 1990 2000];
y = [8 15 25; 30 35 40; 50 55 62];

% Grouped
tiledlayout(2,1);
ax1 = nexttile;
```

```
barh(ax1,x,y)
title('Grouped Style')

% Stacked
ax2 = nexttile;
barh(ax2,x,y,'stacked')
title('Stacked Style')
```

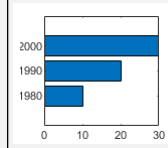


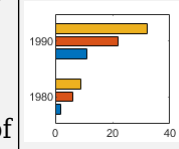
Input Arguments

x — Vertical axis coordinates
scalar | vector | matrix

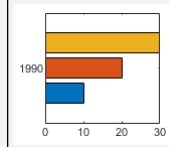
Vertical axis coordinates, specified as a scalar, vector, or matrix. The values of x do not need to be in order, but the size of x depends on the size of y and how you want to display your data. This table describes the most common situations.

Presentation	How to Specify X and Y	Example

Display one series of bars.	Specify x and y as vectors that are the same length. The values in x must be unique, but the values in y do not need to be unique.	`x = [1980 1990 2000];`
		`y = [10 20 30];`
		`barh(x,y)`
		 

Display multiple series of bars in groups.	Specify either of these combinations:
	• Specify x and y as matrices of equal size. Each column of y corresponds to a series of bars. By default, each series is a different color. To ensure consistent placement of the groups, specify the columns of x as identical vectors. The values within a column must be unique, even though the columns are repeated.
	• Specify x as a vector of unique values, and specify y as a matrix. The length of x must equal the length of at least one dimension of y. The other dimension of y contains values for the different series of bars.
		`x = [1980 1980 1980`
		`  1990 1990 1990];`
		`y = [2 6 9`
		`  11 22 32];`
		`barh(x,y)`
		Or
		`x = [1980 1990];`
		`y = [2 6 9`
		`  11 22 32];`
		`barh(x,y)`
		 

Presentation	How to Specify X and Y	Example

Display one group of bars centered at one `x` value.	Specify `x` as a scalar and `y` as a vector.	`x = 1990;`
		`y = [10 20 30];`
		`barh(x,y)`
		 

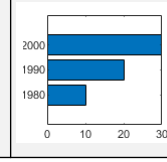
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

y — Bar lengths

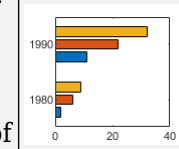
scalar | vector | matrix

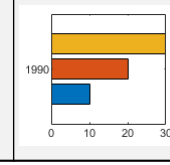
Bar lengths, specified as a scalar, vector, or matrix. The size of `y` depends on the size of `x` and how you want to display your data. This table describes the most common situations.

Presentation	How to Specify X and Y	Example

Display one series of bars.	Specify `x` and `y` as vectors that are the same length. The values in `x` must be unique, but the values in `y` do not need to be unique.	`x = [1980 1990 2000];`
		`y = [10 20 30];`
		`barh(x,y)`
		 

Presentation	How to Specify X and Y	Example

Display multiple series of bars in groups.	Specify either of these combinations:
	• Specify `x` and `y` as matrices of equal size. Each column of `y` corresponds to a series of bars. By default, each series is a different color. To ensure consistent placement of the groups, specify the columns of `x` as identical vectors. The values within a column must be unique, even though the columns are repeated.
	• Specify `x` as a vector of unique values, and specify `y` as a matrix. The length of `x` must equal the length of at least one dimension of `y`. The other dimension of `y` contains values for the different series of bars.
		`x = [1980 1980 1980`
		`  1990 1990 1990];`
		`y = [2 6 9`
		`  11 22 32];`
		`barh(x,y)`
		Or
		`x = [1980 1990];`
		`y = [2 6 9`
		`  11 22 32];`
		`barh(x,y)`
		 

Display one group of bars centered at one `x` value.	Specify `x` as a scalar and `y` as a vector.	`x = 1990;`
		`y = [10 20 30];`
		`barh(x,y)`
		 

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`width` — Bar thickness

`0.8` (default) | scalar

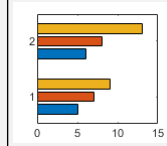
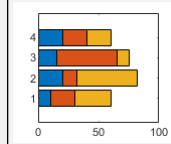
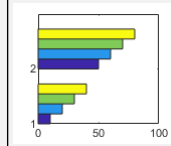
Bar thickness, specified as a fraction of the total space available for each bar. The default of `0.8` means each bar uses 80% of the space from the previous bar to the next bar, with 10% of empty space on either side.

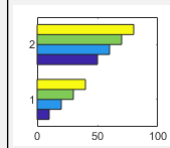
If the thickness is `1`, then the bars within a group touch one another.

Example: `barh([1 2 3],0.5)` creates bars that use 50% of the available space.

`style` — Group style
`'grouped'` (default) | `'stacked'` | `'hist'` | `'histc'`

Group style, specified by one of these values.

Style	Result	Example
`'grouped'`	Display each group as adjacent bars that are centered around the corresponding `x` value.	
`'stacked'`	Display each group as one multicolored bar. The length of a bar is the sum of the elements in the group. If `y` is a vector, then the result is the same as `'grouped'`.	
`'histc'`	Display the bars in histogram format, in which the bars in a group touch one another. The trailing edge of each group is aligned with the corresponding `x` value. **Note** A better way to display a horizontal histogram is to call the `histogram` function and specify the `Orientation` name-value pair argument.	

Style	Result	Example
`'hist'`	Display the bars in histogram format. Each group is centered at the corresponding `x` value. **Note** A better way to display a horizontal histogram is to call the `histogram` function and specify the `Orientation` name-value pair argument.	

`color` — Bar color
`'red'` | `'green'` | `'blue'` | `'r'` | `'g'` | `'b'` | ...

Bar color, specified as one of the options in this table.

Options	Description
`'blue'` or `'b'`	Blue
`'red'` or `'r'`	Red
`'green'` or `'g'`	Green
`'cyan'` or `'c'`	Cyan
`'magenta'` or `'m'`	Magenta
`'yellow'` or `'y'`	Yellow
`'black'` or `'k'`	Black
`'white'` or `'w'`	White

Example: `barh([10 20 30],'r')` creates a set of red bars.

`ax` — Target axes
`Axes` object

Target axes, specified as an `Axes` object. If you do not specify the axes, then the bar graph displays in the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `barh([10 20 30],'EdgeColor','g')` specifies a green outline around the bars.

Note

- The properties listed here are only a subset. For a complete list, see Bar.
- You can set these properties only on bar graphs that use the default `'grouped'` or `'stacked'` style.

`EdgeColor` — Outline color

`'flat'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Outline color, specified as `'flat'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. If there are 150 bars or fewer, the default value is `[0 0 0]`, which corresponds to black. If there are more than 150 adjacent bars, the default value is `'none'`.

Starting in R2017b, the `'flat'` option uses the `CData` values to color the edges. In previous releases, the `'flat'` option colored the edges using colors from the colormap.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `b = bar(1:10,'EdgeColor','red')`

Example: `b.EdgeColor = [0 0.5 0.5];`

Example: `b.EdgeColor = 'flat';`

Example: `b.EdgeColor = '#D2F9A7';`

FaceColor — Fill color

`'flat'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Fill color, specified as `'flat'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'flat'` option uses the `CData` property value of the `Bar` object to color the faces.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Starting in R2017b, the default value is an RGB triplet from the `ColorOrder` property of the axes. In previous releases, the default value was `'flat'` and the colors were based on the colormap.

Example: `b = bar(1:10,'FaceColor','red')`

Example: `b.FaceColor = [0 0.5 0.5];`

Example: `b.FaceColor = 'flat';`

Example: `b.FaceColor = '#D2F9A7';`

`LineWidth` — Width of bar outlines
`0.5` (default) | positive value

Width of bar outlines, specified as a positive value in point units. One point equals 1/72 inch.

Example: `1.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`BaseValue` — Baseline value
`0` (default) | numeric scalar value

Baseline value, specified as a numeric scalar value.

The baseline value that you specify applies to either the *x*-axis or the *y*-axis depending on the bar chart orientation. If you change the orientation of the bar chart from vertical to horizontal, or vice versa, the baseline value might change. Set the `BaseValue` property after setting the `Horizontal` property.

Output Arguments

b — Bar objects
Bar objects

Bar objects. Use the elements in b to access and modify properties of a specific Bar object after it has been created. The number of Bar objects depends on the size of y. If y is a vector, then b is one Bar object. If y is a matrix, then b is a vector containing a Bar object for each series in y.

More About

Series of Bars

A series consists of the bars at all locations in X for a particular set of data. By default, each series of bars is indicated by a different color.

Group of Bars

A group consists of all the bars at a particular location in X.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions
`bar` | `bar3` | `bar3h` | `histogram` | `stairs`

Properties
Bar

Topics

“Modify Baseline of Bar Graph”
“Overlay Bar Graphs”
“Plot Dates and Durations”
“Plot Categorical Data”

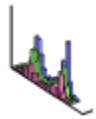
External Websites

MATLAB Plot Gallery

Introduced before R2006a

bar3

Plot 3-D bar graph



Syntax

```
bar3(Z)
bar3(Y,Z)
bar3(...,width)
bar3(...,style)
bar3(...,color)
bar3(ax,...)
h = bar3(...)
```

Description

`bar3` draws a three-dimensional bar graph.

`bar3(Z)` draws a three-dimensional bar chart, where each element in `Z` corresponds to one bar. When `Z` is a vector, the *y*-axis scale ranges from `1` to `length(Z)`. When `Z` is a matrix, the *y*-axis scale ranges from `1` to the number of rows in `Z`.

`bar3(Y,Z)` draws a bar chart of the elements in `Z` at the locations specified in `Y`, where `Y` is a vector defining the *y* values for the vertical bars. The *y* values can be nonmonotonic, but cannot contain duplicate values. If `Z` is a matrix, elements from the same row in `Z` appear at the same location along the `y`-axis.

`bar3(...,width)` sets the width of the bars and controls the separation of bars within a group. The default `width` is `0.8` and the bars have a slight separation. If `width` is `1`, the bars within a group touch one another.

`bar3(...,style)` specifies the style of the bars. `style` is `'detached'`, `'grouped'`, or `'stacked'`. Default mode of display is `'detached'`.

- `'detached'` displays the elements of each row in Z as separate blocks behind one another in the *x* direction.
- `'grouped'` displays *n* groups of *m* vertical bars, where *n* is the number of rows and *m* is the number of columns in Z. Each group contains one bar per column in Z.
- `'stacked'` displays one bar for each row in Z. The bar height is the sum of the elements in the row. Each bar is multicolored, with colors corresponding to distinct elements and showing the relative contribution each row element makes to the total sum.

`bar3(...,color)` displays all bars using the color specified by `color`. For example, use `'r'` for red bars. Specify `color` as one of these values: `'r'`, `'g'`, `'b'`, `'c'`, `'m'`, `'y'`, `'k'`, or `'w'`.

`bar3(ax,...)` plots into the axes `ax` instead of into the current axes (`gca`).

`h = bar3(...)` returns a vector of `Surface` objects. When Z is a matrix, `bar3` creates one `Surface` object per column in Z.

Examples

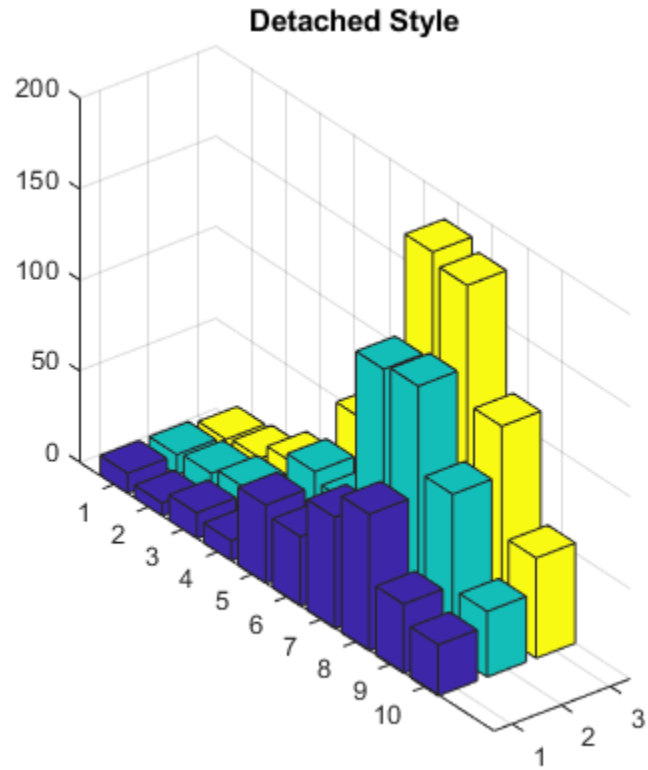
Create 3-D Bar Graph

Load the data set `count.dat`, which returns a three-column matrix, `count`. Store Z as the first 10 rows of `count`.

```
load count.dat
Z = count(1:10,:);
```

Create a 3-D bar graph of Z. By default, the style is `detached`.

```
figure
bar3(Z)
title('Detached Style')
```



Specify Bar Width for 3-D Bar Graph

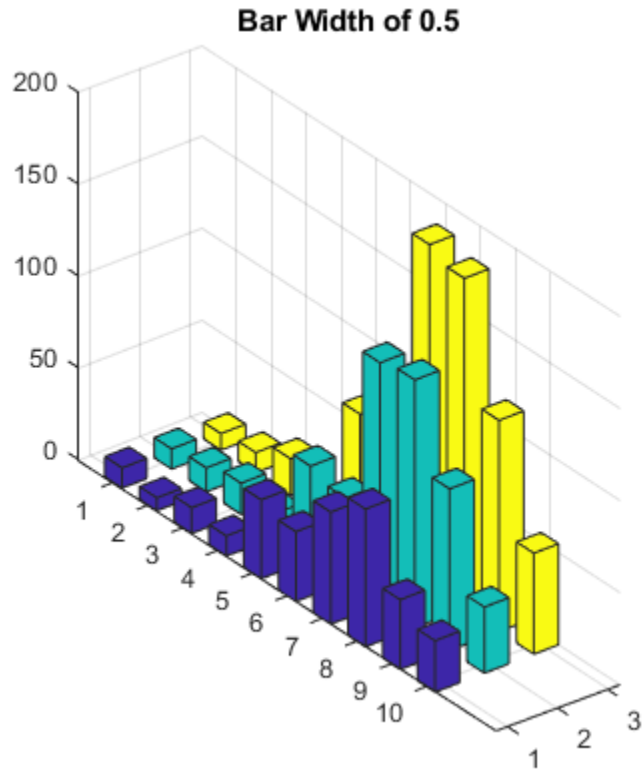
Load the data set `count.dat`, which returns a three-column matrix, `count`. Store Z as the first 10 rows of `count`.

```
load count.dat
Z = count(1:10,:);
```

Create a 3-D bar graph of Z and set the bar width to 0.5.

```
width = 0.5;
```

```
figure
bar3(Z,width)
title('Bar Width of 0.5')
```



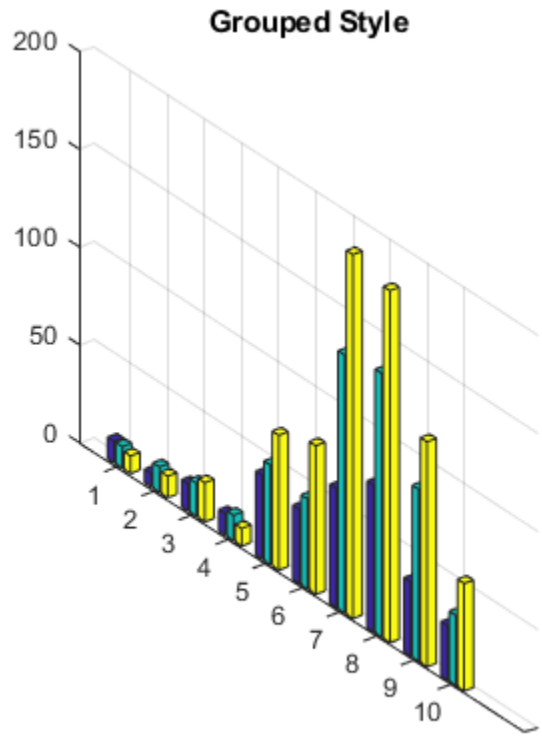
3-D Bar Graph with Grouped Style

Load the data set `count.dat`, which returns a three-column matrix, `count`. Store `Z` as the first 10 rows of `count`.

```
load count.dat
Z = count(1:10,:);
```

Create a 3-D bar graph of Z. Group the elements in each row of Z by specifying the style option as `grouped`.

```
figure
bar3(Z,'grouped')
title('Grouped Style')
```



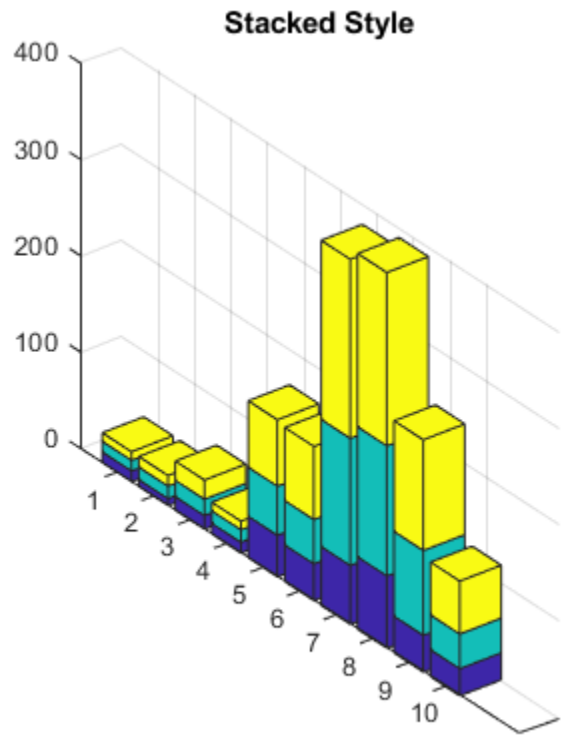
3-D Bar Graph with Stacked Style

Load the data set `count.dat`, which returns a three-column matrix, `count`. Store Z as the first 10 rows of `count`.

```
load count.dat
Z = count(1:10,:);
```

Create a 3-D bar graph of Z. Stack the elements in each row of Z by specifying the style option as `stacked`.

```
figure
bar3(Z,'stacked')
title('Stacked Style')
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`LineSpec` | `bar` | `bar3h` | `barh`

Topics

“Color 3-D Bars by Height”

Introduced before R2006a

bar3h

Plot horizontal 3-D bar graph

Syntax

```
bar3h(Y)
bar3h(Z,Y)
bar3h(...,width)
bar3h(...,style)
bar3h(...,color)
bar3h(ax,...)
h = bar3h(...)
```

Description

`bar3h` draws three-dimensional horizontal bar charts.

`bar3h(Y)` draws a three-dimensional bar chart, where each element in `Y` corresponds to one bar. When `Y` is a vector, the *z*-axis scale ranges from `1` to `length(Y)`. When `Y` is a matrix, the *z*-axis scale ranges from `1` to the number of rows in `Y`.

`bar3h(Z,Y)` draws a bar chart of the elements in `Y` at the locations specified in `Z`, where `Z` is a vector defining the *z* values for the horizontal bars. The *z* values can be nonmonotonic, but cannot contain duplicate values. If `Y` is a matrix, elements from the same row in `Y` appear at the same location along the `z`-axis.

`bar3h(...,width)` sets the width of the bars and controls the separation of bars within a group. The default `width` is `0.8` and the bars have a slight separation. If `width` is `1`, then the bars within a group touch one another.

`bar3h(...,style)` specifies the style of the bars. `style` is `'detached'`, `'grouped'`, or `'stacked'`. Default mode of display is `'detached'`.

- `'detached'` displays the elements of each row in `Y` as separate blocks behind one another in the *x* direction.
- `'grouped'` displays *n* groups of *m* horizontal bars, where *n* is the number of rows and *m* is the number of columns in `Y`. Each group contains one bar per column in `Y`.
- `'stacked'` displays one bar for each row in `Y`. The bar length is the sum of the elements in the row. Each bar is multicolored, with colors corresponding to distinct elements and showing the relative contribution each row element makes to the total sum.

`bar3h(...,color)` displays all bars using the color specified by `color`. For example, use `'r'` for red bars. Specify `color` as one of these values: `'r'`, `'g'`, `'b'`, `'c'`, `'m'`, `'y'`, `'k'`, or `'w'`.

`bar3h(ax,...)` plots into the axes `ax` instead of into the current axes (`gca`).

`h = bar3h(...)` returns a vector `Surface` objects. When `Y` is a matrix, `bar3h` creates one `Surface` object per column in `Y`.

Examples

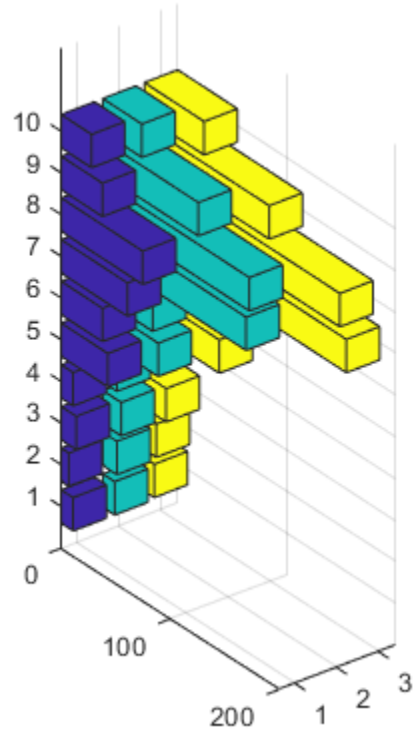
Create 3-D Horizontal Bar Graph

Load the data set `count.dat`, which returns a three-column matrix, `count`. Store `Y` as the first ten rows of `count`.

```
load count.dat
Y = count(1:10,:);
```

Create a 3-D horizontal bar graph of `Y`. By default, the style is `detached`.

```
figure
bar3h(Y)
```



Specify Bar Width for 3-D Horizontal Bar Graph

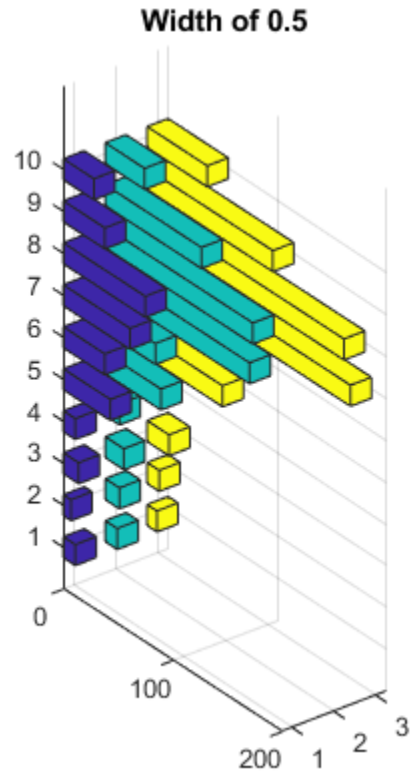
Load the data set `count.dat`, which returns a three-column matrix, `count`. Store Y as the first ten rows of `count`.

```
load count.dat;
Y = count(1:10,:);
```

Create a 3-D horizontal bar graph of Y and set the bar width to 0.5.

```
width = 0.5;
```

```
figure
bar3h(Y,width)
title('Width of 0.5')
```



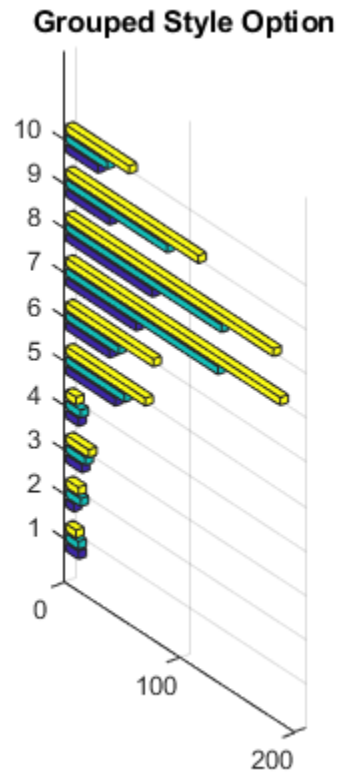
3-D Horizontal Bar Graph with Grouped Style

Load the data set `count.dat`, which returns a three-column matrix, `count`. Store `Y` as the first ten rows of `count`.

```
load count.dat
Y = count(1:10,:);
```

Create a 3-D horizontal bar graph of Y and specify the style option as `grouped`.

```
figure
bar3h(Y,'grouped')
title('Grouped Style Option')
```



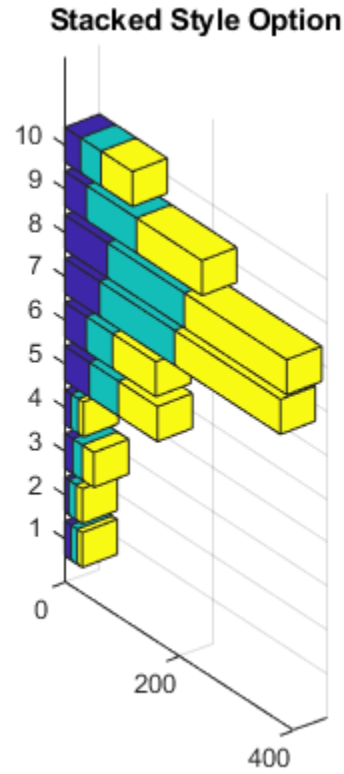
3-D Horizontal Bar Graph with Stacked Option

Load the data set `count.dat`, which returns a three-column matrix, `count`. Store Y as the first ten rows of `count`.

```
load count.dat
Y = count(1:10,:);
```

Create a 3-D horizontal bar graph of Y and specify the style option as `stacked`.

```
figure
bar3h(Y,'stacked')
title('Stacked Style Option')
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`LineSpec` | `bar` | `bar3` | `barh` | `patch`

Introduced before R2006a

Bar Properties

Bar chart appearance and behavior

Description

`Bar` properties control the appearance and behavior of a `Bar` object. By changing property values, you can modify certain aspects of the bar chart.

Starting in R2014b, you can use dot notation to query and set properties.

```
b = bar(1:10);
c = b.FaceColor
b.FaceColor = [0 0.5 0.5];
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Color and Styling

`FaceColor` — Fill color

`'flat'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Fill color, specified as `'flat'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'flat'` option uses the `CData` property value of the `Bar` object to color the faces.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Starting in R2017b, the default value is an RGB triplet from the `ColorOrder` property of the axes. In previous releases, the default value was `'flat'` and the colors were based on the colormap.

Example: `b = bar(1:10,'FaceColor','red')`

Example: `b.FaceColor = [0 0.5 0.5];`

Example: `b.FaceColor = 'flat';`

Example: `b.FaceColor = '#D2F9A7';`

`FaceColorMode` — Control how `FaceColor` is set
`'auto'` (default) | `'manual'`

Control how the `FaceColor` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the `FaceColor` property by selecting a color from the `ColorOrder` property of the axes.
- `'manual'` — You control the value of the `FaceColor` property manually, either by specifying a color when you call a charting function that creates the object, or by setting the `FaceColor` property on the object after creating it.

If you change the value of the `FaceColor` property manually, MATLAB changes the value of the `FaceColorMode` property to `'manual'`.

`EdgeColor` — Outline color
`'flat'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Outline color, specified as `'flat'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. If there are 150 bars or fewer, the default value is `[0 0 0]`, which corresponds to black. If there are more than 150 adjacent bars, the default value is `'none'`.

Starting in R2017b, the `'flat'` option uses the `CData` values to color the edges. In previous releases, the `'flat'` option colored the edges using colors from the colormap.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `b = bar(1:10,'EdgeColor','red')`

Example: `b.EdgeColor = [0 0.5 0.5];`

Example: `b.EdgeColor = 'flat';`

Example: `b.EdgeColor = '#D2F9A7';`

FaceAlpha — Face transparency

1 (default) | scalar in range [0,1]

Face transparency, specified as a scalar in the range `[0,1]`. A value of 1 is opaque and 0 is completely transparent. Values between 0 and 1 are semitransparent.

Example: `b = bar(1:10,'FaceAlpha',0.5)`

Example: `b.FaceAlpha = 0.5;`

`EdgeAlpha` — Edge transparency

`1` (default) | scalar in range `[0,1]`

Edge transparency, specified as a scalar in the range `[0,1]`. A value of 1 is opaque and 0 is completely transparent. Values between 0 and 1 are semitransparent.

Example: `b = bar(1:10,'EdgeAlpha',0.5)`

Example: `b.EdgeAlpha = 0.5;`

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Width of bar outlines

`0.5` (default) | positive value

Width of bar outlines, specified as a positive value in point units. One point equals 1/72 inch.

Example: `1.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Bar Graph Type

`BarLayout` — Arrangement of bars
`'grouped'` (default) | `'stacked'`

Arrangement of bars, specified as one of these values:

- `'grouped'` — Group bars by rows in `Y`, where `Y` is the input argument to the `bar` or `barh` function that created the bar chart.
- `'stacked'` — Display one bar for each row in `Y`. The bar height is the sum of the elements in the row. Each bar is multicolored. Colors correspond to distinct elements and show the relative contribution each row element makes to the total sum.

`BarWidth` — Relative width of individual bars
`0.8` (default) | scalar in range `[0,1]`

Relative width of individual bars, specified as a scalar value in the range `[0,1]`. Use this property to control the separation of bars within a group. The default value is `0.8`, which means that MATLAB separates the bars slightly. If you set this property to `1`, then adjacent bars touch.

Example: `0.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Horizontal` — Horizontal bar chart
`'off'` (default) | `'on'`

Horizontal bar chart, specified as one of these values:

- `'on'` — Display bars horizontally. If you create a graph with `barh`, then the `Horizontal` property is set to `'on'`.
- `'off'` — Display bars vertically. If you create the chart with `bar`, then the `Horizontal` property is set to `'off'`.

Baseline

`BaseValue` — Baseline value
`0` (default) | numeric scalar value

Baseline value, specified as a numeric scalar value.

The baseline value that you specify applies to either the *x*-axis or the *y*-axis depending on the bar chart orientation. If you change the orientation of the bar chart from vertical to horizontal, or vice versa, the baseline value might change. Set the `BaseValue` property after setting the `Horizontal` property.

`ShowBaseLine` — Baseline visibility

`'on'` (default) | `'off'`

Baseline visibility, specified as one of these values:

- `'on'` — Show the baseline.
- `'off'` — Hide the baseline.

`BaseLine` — Baseline

baseline object

This property is read-only.

Baseline object. For a list of baseline properties, see Baseline.

Data

`CData` — Color data

RGB triplet | three-column matrix | scalar | vector

Color data, specified as one of these values:

- RGB triplet — Single RGB color value applies to all bars.
- Three-column matrix — One color per bar. Each row in the matrix specifies an RGB triplet for a particular bar.
- Scalar — Single color applies to all bars, where the color comes from the colormap.
- Vector — One color per bar. The colors come from the colormap.

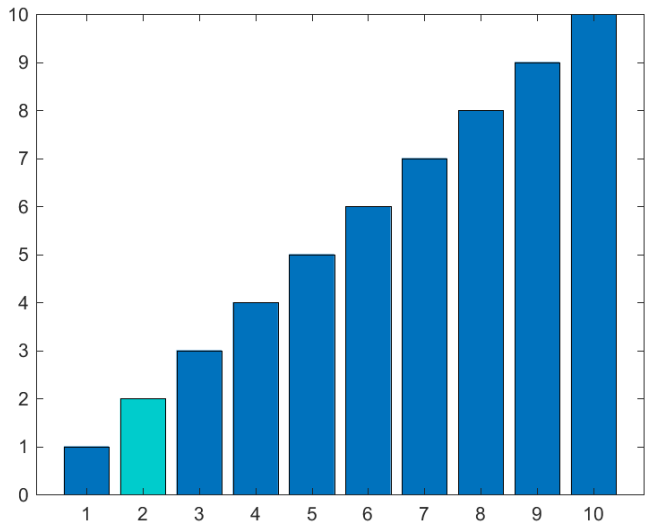
By default, when you create a bar chart, the `CData` property contains a three-column matrix of RGB triplets. You can change the color for a particular bar by changing the corresponding row in the matrix.

This property applies only when the `FaceColor` or `EdgeColor` property is set to `'flat'`.

Example

Change the color for a particular bar by setting the `FaceColor` property to `'flat'`. Then change the corresponding row in the `CData` matrix to the new RGB triplet. For example, change the color of the second bar.

```
b = bar(1:10,'FaceColor','flat');
b.CData(2,:) = [0 0.8 0.8];
```



`XData` — Bar locations
vector

Bar locations, specified as a vector with no repeating values.

- For vertical bar charts, the values are the bar locations along the *x*-axis.
- For horizontal bar charts, the values are the bar locations along the *y*-axis.

Alternatively, specify the bar locations using the input argument `X` to the `bar` or `barh` function. If you do not specify `X`, then the indices of the values in `YData` determine the bar locations.

`XData` and `YData` must have equal lengths.

Example: `1:10`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`XDataMode` — Selection mode for `XData`
`'auto'` (default) | `'manual'`

Selection mode for `XData`, specified as one of these values:

- `'auto'` — Use the indices of the values in `YData` (or `ZData` for 3-D plots).
- `'manual'` — Use manually specified values. To specify the values, set the `XData` property or specify the input argument `X` to the plotting function.

`XDataSource` — Variable linked to `XData`
`''` (default) | character vector | string

Variable linked to `XData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `XData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `XData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'x'`

`YData` — Bar lengths
vector

Bar lengths, specified as a vector. Alternatively, specify the bar lengths using the input argument `Y` to the `bar` or `barh` function.

`XData` and `YData` must have equal lengths.

Example: `1:10`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`YDataSource` — Variable linked to `YData`
`''` (default) | character vector | string

Variable linked to `YData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `YData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `YData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'y'`

`XEndPoints` — *x*-coordinates of bar tips
vector

This property is read-only.

x-coordinates of the tips of the bars, returned as a vector. These coordinates are useful when you want to add text, error bars, or other objects to the tips of the bars. For example, you can pass the value of this property to the `text` function when you want to add text to the tips of the bars.

`YEndPoints` — *y*-coordinates of bar tips
vector

This property is read-only.

y-coordinates of the tips of the bars, returned as a vector. These coordinates are useful when you want to add text, error bars, or other objects to the tips of the bars. For example, you can pass the value of this property to the `text` function when you want to add text to the tips of the bars.

Legend

`DisplayName` — Legend label

`''` (default) | character vector | string scalar

Legend label, specified as a character vector or string scalar. The legend does not display until you call the `legend` command. If you do not specify the text, then `legend` sets the label using the form `'dataN'`.

`Annotation` — Control for including or excluding object from legend

`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

DataTipTemplate — Data tip content

DataTipTemplate object

Data tip content, specified as a DataTipTemplate object. You can control the content that appears in a data tip by modifying the properties of the underlying DataTipTemplate object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

Note The DataTipTemplate object is not returned by findobj or findall, and it is not copied by copyobj.

UIContextMenu — Context menu

empty GraphicsPlaceholder array (default) | ContextMenu object

Context menu, specified as a ContextMenu object. Use this property to display a context menu when you right-click the object. Create the context menu using the uicontextmenu function.

Note If the PickableParts property is set to 'none' or if the HitTest property is set to 'off', then the context menu does not appear.

Selected — Selection state

'off' (default) | 'on'

Selection state, specified as one of these values:

- 'on' — Selected. If you click the object when in plot edit mode, then MATLAB sets its Selected property to 'on'. If the SelectionHighlight property also is set to 'on', then MATLAB displays selection handles around the object.
- 'off' — Not selected.

SelectionHighlight — Display of selection handles

'on' (default) | 'off'

Display of selection handles when selected, specified as one of these values:

- 'on' — Display selection handles when the Selected property is set to 'on'.

- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

`Clipping` — Clipping of object to axes limits

`'on'` (default) | `'off'`

Clipping of the object to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the object that are outside the axes limits.
- `'off'` — Display the entire object, even if parts of it appear outside the axes limits. Parts of the object might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the object so that it is larger than the original plot.

The `Clipping` property of the axes that contains the object must be set to `'on'`. Otherwise, this property has no effect. For more information about the clipping behavior, see the `Clipping` property of the axes.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.

- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

PickableParts — Ability to capture mouse clicks

`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks when visible. The `Visible` property must be set to `'on'` and you must click a part of the `Bar` object that has a defined color. You cannot click a part that has an associated color property set to `'none'`. The `HitTest` property determines if the `Bar` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `Bar` object passes the click to the object below it in the current view of the figure window. The `HitTest` property of the `Bar` object has no effect.

HitTest — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `Bar` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `Bar` object that has one of these:
  - `HitTest` property set to `'on'`
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

---

Note The `PickableParts` property determines if the `Bar` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

BeingDeleted — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent
`Axes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `Group`, or `Transform` object.

`Children` — Children
empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the chart object.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object
`'bar'` (default)

This property is read-only.

Type of graphics object, returned as `'bar'`. Use this property to find all objects of a given type within a plotting hierarchy, such as searching for the type using `findobj`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[ ]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`bar` | `barh` | `pareto`

Topics

“Access Property Values”
“Graphics Object Properties”

Introduced before R2006a

baryToCart

Class: `TriRep`

(Not recommended) Convert point coordinates from barycentric to Cartesian

---

Note `baryToCart(TriRep)` is not recommended. Use `barycentricToCartesian(triangulation)` instead.

`TriRep` is not recommended. Use `triangulation` instead.

Syntax

```
XC = baryToCart(TR, SI, B)
```

Description

`XC = baryToCart(TR, SI, B)` returns the Cartesian coordinates `XC` of each point in `B` that represents the barycentric coordinates with respect to its associated simplex `SI`.

Input Arguments

`TR`	Triangulation representation.
`SI`	Column vector of simplex indices that index into the triangulation matrix `TR.Triangulation`
`B`	`B` is a matrix that represents the barycentric coordinates of the points to convert with respect to the simplices `SI`. `B` is of size `m`-by-`k`, where `m = length(SI)`, the number of points to convert, and `k` is the number of vertices per simplex.

Output Arguments

`XC`	Matrix of Cartesian coordinates of the converted points. `XC` is of size `m`-by-`n`, where `n` is the dimension of the space where the triangulation resides. That is, the Cartesian coordinates of the point `B(j)` with respect to simplex `SI(j)` is `XC(j)`.

Examples

Compute the Delaunay triangulation of a set of points.

```
x = [0 4 8 12 0 4 8 12]';
y = [0 0 0 0 8 8 8 8]';
dt = DelaunayTri(x,y)
```

Compute the barycentric coordinates of the incenters.

```
cc = incenters(dt);
tri = dt(:,:);
```

Plot the original triangulation and reference points.

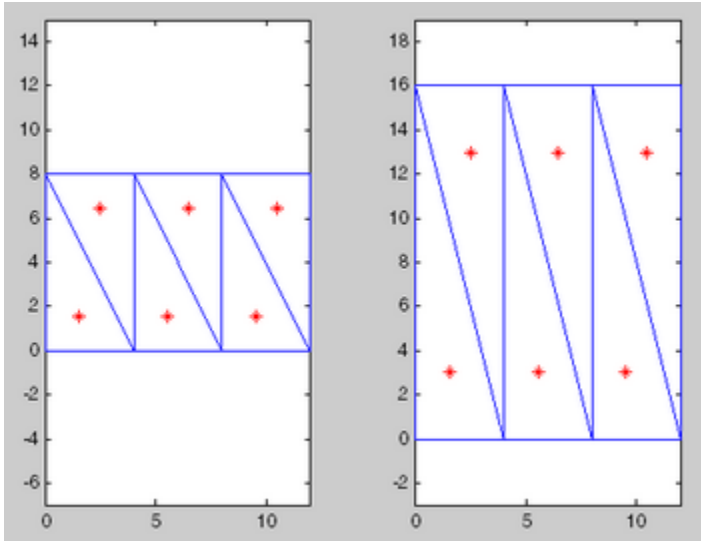
```
figure
subplot(1,2,1);
triplot(dt); hold on;
plot(cc(:,1), cc(:,2), '*r'); hold off;
axis equal;
```

Stretch the triangulation and compute the mapped locations of the incenters on the deformed triangulation.

```
b = cartToBary(dt,[1:length(tri)]',cc);
y = [0 0 0 0 16 16 16 16]';
tr = TriRep(tri,x,y)
xc = baryToCart(tr, [1:length(tri)]', b);
```

Plot the deformed triangulation and mapped locations of the reference points.

```
subplot(1,2,2);
triplot(tr); hold on;
plot(xc(:,1), xc(:,2), '*r'); hold off;
axis equal;
```



More About

Simplex

A simplex is a triangle/tetrahedron or higher-dimensional equivalent.

See Also

`cartesianToBarycentric` | `delaunayTriangulation` | `pointLocation` | `triangulation`

base2dec

Convert text representing number in base N to decimal number

Syntax

```
D = base2dec(baseStr,n)
```

Description

`D = base2dec(baseStr,n)` converts `baseStr` to a decimal number and returns it. The input argument `baseStr` represents digits of a base-`n` number using numeric characters and, when `n` is greater than 10, letters. For example, if `n` is 12, then `baseStr` represents the numbers 9, 10, and 11 using the characters `'9'`, `'A'`, and `'B'`, and represents the number 12 as the character sequence `'10'`. Letters can be either uppercase or lowercase.

Examples

Convert Text Representing Base-12 Value

Convert a character vector that represents a base-12 value to a decimal number.

```
baseStr = '1B';
D = base2dec(baseStr,12)
```

```
D = 23
```

Return Numeric Array

Create a string array that represents multiple octal, or base-8, values.

```
baseStr = ["1777" "172" "16"]
```

```
baseStr = 1x3 string array
    "1777"    "172"    "16"
```

Convert the octal values and return a numeric array.

```
D = base2dec(baseStr,8)
```

```
D = 1×3

        1023         122          14
```

Input Arguments

baseStr — Text representing base-n numbers
character array | cell array of character vectors | string array

Text representing base-n numbers, specified as a character array, cell array of character vectors, or string array.

- If `baseStr` is a character array with multiple rows or a cell array of character vectors, then the output is a numeric column vector.
- If `baseStr` is a string array, then the output is a numeric array that has the same dimensions.

`baseStr` cannot represent a negative number.

n — Base of input representation
integer between 2 and 36

Base of input representation, specified as an integer between 2 and 36.

Tips

- If the input argument `baseStr` represents a value greater than the value returned by `flintmax`, then `base2dec` might not return an exact conversion.

See Also

`bin2dec` | `dec2base` | `flintmax` | `hex2dec`

Topics

"Hexadecimal and Binary Values"

Introduced before R2006a

Baseline Properties

Baseline appearance and behavior

Description

`Baseline` objects are created as part of bar charts, area charts, and stem charts. `Baseline` properties control the appearance and behavior of a `Baseline` object. By changing property values, you can modify certain aspects of the baseline. Use dot notation to refer to a particular object and property:

```
b = bar(1:10);
bl = b.BaseLine;
c = bl.Color;
bl.Color = 'red';
```

Properties

Appearance

`Color` — Line color
`'none'` | RGB triplet | hexadecimal color codes | `'r'` | `'g''b'` | ...

Line color, specified as `'none'`, an RGB triplet, hexadecimal color code, a color name, or a short name. If you specify the color as `'none'`, then the baseline is invisible.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`LineStyle` — Line style
`'-'` | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the line styles listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width
positive value

Line width, specified as a positive value in point units.

Example: `0.75`

Location and Visibility

`BaseValue` — Value of baseline
scalar

Value of the baseline, specified as a scalar.

Typically, baselines are associated with bar series, stem series, or area objects. The `BaseValue` property for the associated object and the `BaseValue` property for the baseline object always have the same value. Setting one property also sets the other property. The `BaseLine` property for the associated object contains the baseline object.

Example: `0.75`

`Visible` — Visibility
`'on'|'off'`

Visibility, specified as one of these values:

- `'on'` — Display the baseline.
- `'off'` — Hide the baseline without deleting it. You can access the properties of an invisible baseline.

Typically, baselines are associated with bar series, stem series, or area objects. The `ShowBaseline` property for the associated object and the `Visible` property for the

baseline object always have the same value. Setting one property also sets the other property. The `BaseLine` property for the associated object contains the baseline object.

Parent/Child

`Parent` — Parent
`Axes` object

Parent, returned as an `Axes` object. However, `Baseline` objects are not listed as children of the axes.

`Children` — Children
empty `GraphicsPlaceholder` array

The baseline has no children. You cannot set this property.

See Also

`area` | `bar` | `barh` | `stem`

Topics

"Access Property Values"
"Graphics Object Properties"

Introduced in R2014b

batchStartupOptionUsed

Determine if MATLAB started with `-batch` option

Syntax

```
batchStartupOptionUsed
```

Description

`batchStartupOptionUsed` returns `true` when MATLAB starts with `-batch` option. Otherwise, the function returns `false`. Use the `-batch` option to start MATLAB non-interactively.

Use this function to guard code from being executed when MATLAB is running non-interactively with the `-batch` startup option and user input is either not desired or not supported in this mode.

Examples

Provide Default Value for Automated Test

Suppose that you have a test that requires user input. You can automate this test to run using the `matlab -batch` startup option.

Prompt the user for input during interactive testing or specify a default value for automated testing.

```
if ~batchStartupOptionUsed
    answer = inputdlg("Enter a number");
else
```

```
    answer = 10;
end
```

See Also

isdeployed | ismcc | matlab (Linux) | matlab (Windows) | matlab (macOS)

Introduced in R2019a

beep

Produce operating system beep sound

Syntax

```
beep
beep on
beep off
```

```
status = beep
```

Description

`beep` produces your computer's default beep sound, if it is enabled.

`beep on` enables the beep sound.

`beep off` disables the beep sound.

`status = beep` returns the current beep mode (`on` or `off`).

Examples

Produce Beep Sound

Produce your system’s default beep sound after a period of silence.

Pause for 5 seconds of silence, and then produce your system’s default beep sound.

```
pause(5)
beep
```

Tips

- If you have configured your system not to produce any sound, then `beep` is silent.
- `beep` produces the operating system's default beep sound. To produce a sound and specify its pitch and duration in MATLAB, use the `sound` function.

See Also

`sound`

Introduced before R2006a

BeginInvoke

Initiate asynchronous .NET delegate call

Syntax

```
result = BeginInvoke(arg1,...,argN,callback,object)
```

Description

`result = BeginInvoke(arg1,...,argN,callback,object)` initiates asynchronous call to a .NET delegate. You must call `EndInvoke` to complete the asynchronous call.

Input Arguments

`arg1,...,argN`

Input arguments for delegate. The type and number of arguments must agree with the delegate signature.

`callback`

.NET System.AsyncCallback delegate, or `[]` null value.

`object`

User-defined object, or `[]` null value.

Output Arguments

`result`

.NET System.IAsyncResult object. Used to monitor the progress of the asynchronous call. Input argument to `EndInvoke`.

See Also

`EndInvoke`

Topics

"Calling .NET Methods Asynchronously"

External Websites

MSDN Calling Synchronous Methods Asynchronously

Introduced in R2011a

bench

MATLAB benchmark

Syntax

```
bench
bench(N)
bench(0)
t = bench(N)
```

Description

`bench` measures the execution speed of six different MATLAB tasks and compares it to the speed of several other computers. The six tasks are:

Test	Description	Performance Factors
LU	Perform `lu` of a full matrix	Floating-point, regular memory access
FFT	Perform `fft` of a full vector	Floating-point, irregular memory access
ODE	Solve van der Pol equation with `ode45`	Data structures and MATLAB function files
Sparse	Solve a symmetric sparse linear system	Mixed integer and floating-point
2-D	Plot Lissajous curves	2-D line drawing graphics
3-D	Display colormapped `peaks` with clipping and transforms	3-D animated OpenGL graphics

A final bar chart shows speed, which is inversely proportional to time. The longer bars represent faster machines, and the shorter bars represent the slower ones.

`bench(N)` runs each of the six tasks N times.

`bench(0)` just displays the results from other machines.

`t = bench(N)` returns an N-by-6 array with the execution times.

Tips

Note A benchmark is intended to compare performance of one particular version of MATLAB on different machines. It does not offer direct comparisons between different versions of MATLAB because tasks and problem sizes change from version to version.

The LU and FFT tasks involve large matrices and long vectors.

The 2-D and 3-D tasks measure graphics performance, including support for hardware-accelerated graphics. The `rendererinfo` function provides information about the graphics renderer implementation MATLAB is using. For example, this command gets the information for the current axes and stores it in a structure called `info`.

```
info = rendererinfo(gca)
```

Fluctuations of five or ten percent in the measured times of repeated runs on a single machine are normal.

See Also

`cputime` | `memory` | `mlint` | `mlintrpt` | `pack` | `profile` | `profsave` | `rehash` | `tic`

Introduced in R2008a

besselh

Bessel function of third kind (Hankel function)

Syntax

```
H = besselh(nu,Z)
H = besselh(nu,K,Z)
H = besselh(nu,K,Z,scale)
```

Description

`H = besselh(nu,Z)` computes the Hankel function on page 1-846 of the first kind $H_\nu^{(1)}(z) = J_\nu(z) + iY_\nu(z)$ for each element in array `Z`.

`H = besselh(nu,K,Z)` computes the Hankel function on page 1-846 of the first or second kind $H_\nu{}^{(K)}(z)$, where `K` is `1` or `2`, for each element of array `Z`.

`H = besselh(nu,K,Z,scale)` specifies whether to scale the Hankel function to avoid overflow or loss of accuracy. If `scale` is `1`, then Hankel functions of the first kind $H_\nu^{(1)}(z)$ are scaled by e^{-iZ}, and Hankel functions of the second kind $H_\nu^{(2)}(z)$ are scaled by e^{+iZ}.

Examples

Modulus and Phase of Hankel Function

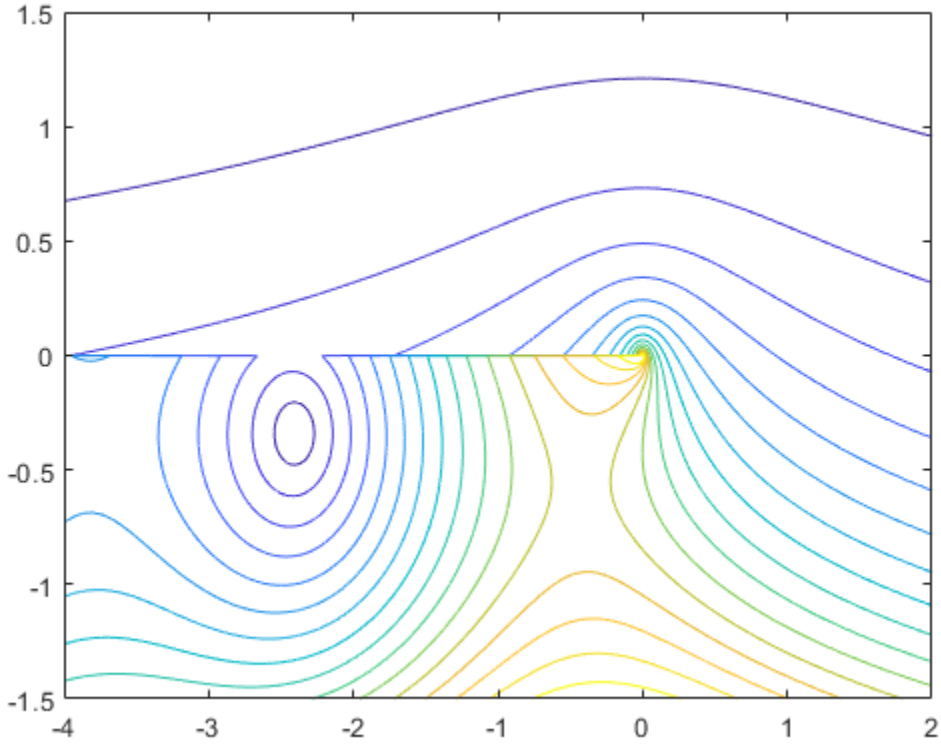
Generate the contour plots of the modulus and phase of the Hankel function $H_0^{(1)}(z)$ [1].

Create a grid of values for the domain.

```
[X,Y] = meshgrid(-4:0.002:2,-1.5:0.002:1.5);
```

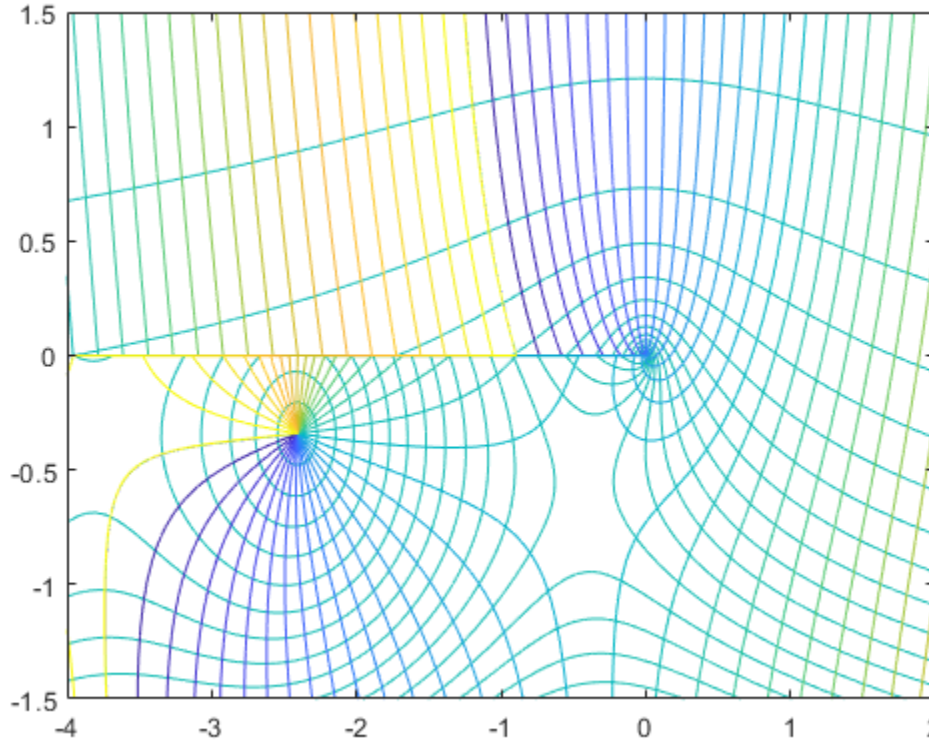
Calculate the Hankel function over this domain and generate the modulus contour plot.

```
H = besselh(0,X+1i*Y);
contour(X,Y,abs(H),0:0.2:3.2)
hold on
```



In the same figure, add the contour plot of the phase.

```
contour(X,Y,rad2deg(angle(H)),-180:10:180)
hold off
```



Asymptotic Behavior

Plot the real and imaginary parts of the Hankel function of the second kind and examine their asymptotic behavior.

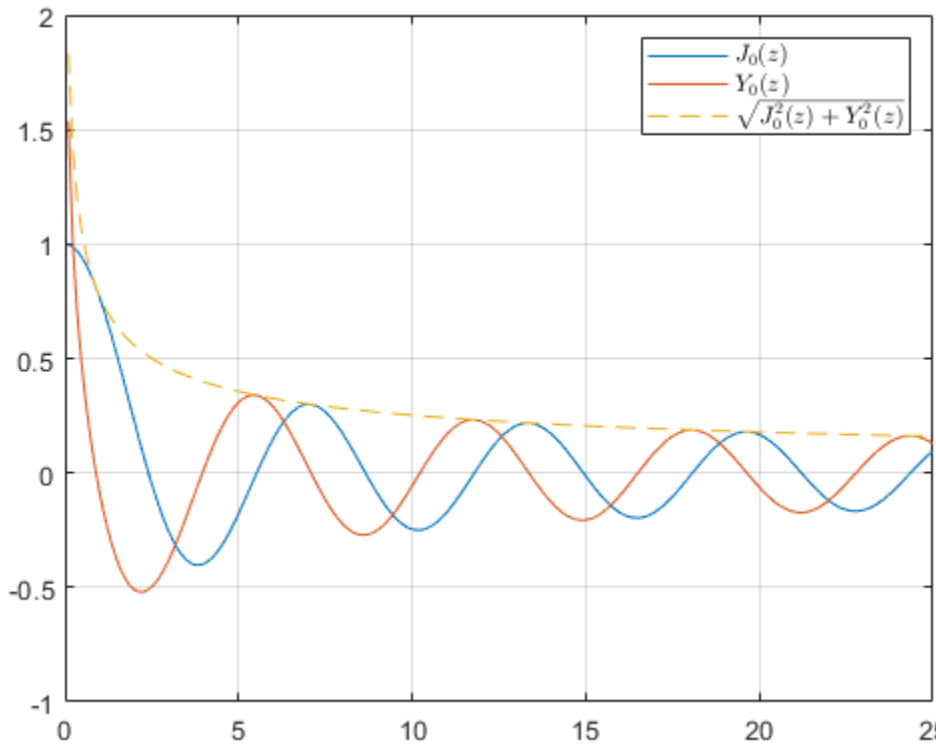
Calculate the Hankel function of the second kind $H_0^{(2)}(z) = J_0(z) - iY_0(z)$ in the interval [0.1, 25].

```
k = 2;
nu = 0;
```

```
z = linspace(0.1,25,200);
H = besselh(nu,k,z);
```

Plot the real and imaginary parts of the function. In the same figure, plot the linear combination $\sqrt{J_0^2(z) + Y_0^2(z)}$, which reveals the asymptotic behavior of the magnitudes of the real and imaginary parts.

```
plot(z,real(H),z,imag(H))
grid on
hold on
M = sqrt(real(H).^2 + imag(H).^2);
plot(z,M,'--')
legend('$J_0(z)$', '$Y_0(z)$', '$\sqrt{J_0^2 (z) + Y_0^2 (z)}$','interpreter','latex')
```

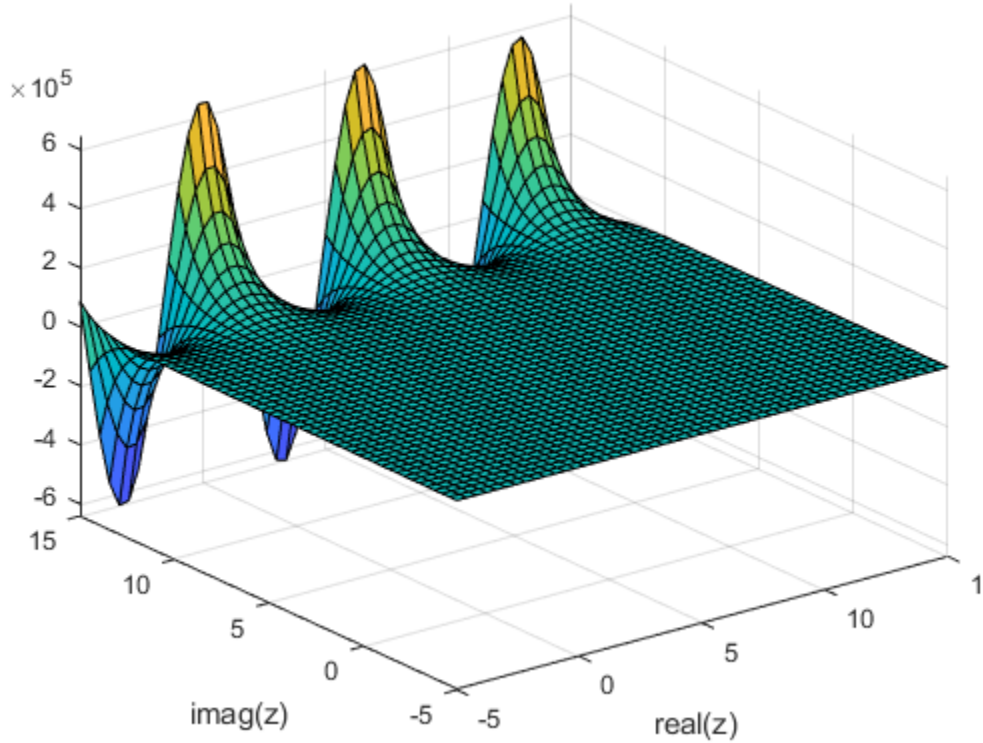


Exponentially Scaled Hankel Function

Calculate the exponentially scaled Hankel function $H_1^{(2)}(z) \cdot e^{iz}$ on the complex plane and compare it to the unscaled function.

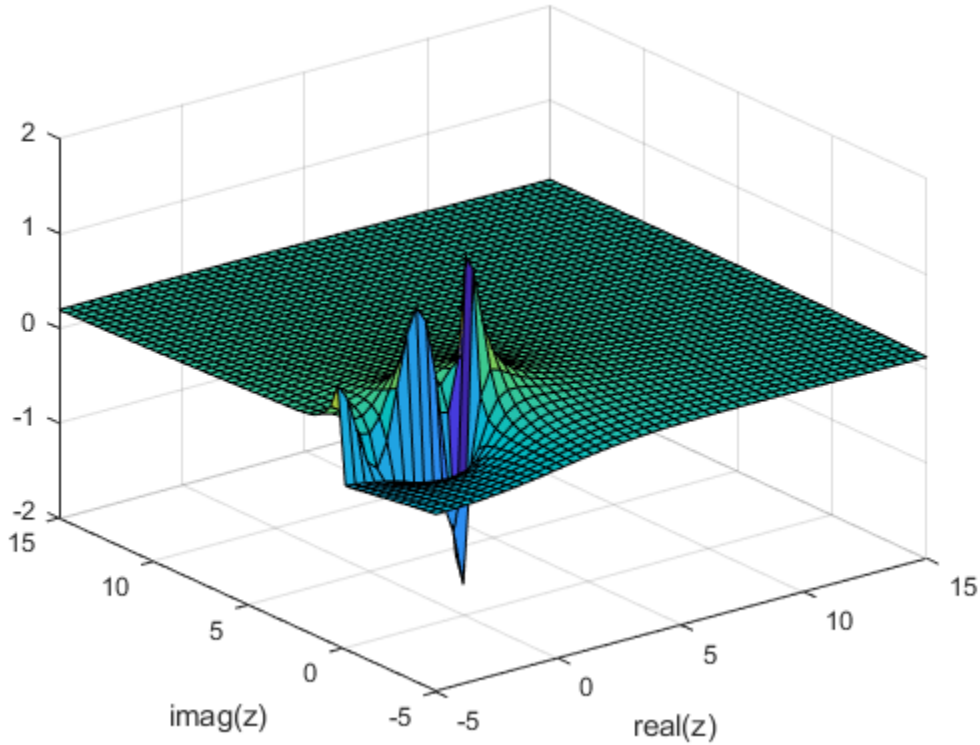
Calculate the unscaled Hankel function of the second order on the complex plane. When z has a large positive imaginary part, the value of the function quickly diverges. This phenomenon limits the range of computable values.

```
k = 2;
nu = 1;
x = -5:0.4:15;
y = x';
z = x + 1i*y;
scaled = 1;
H = besselh(nu,k,z);
surf(x,y,imag(H))
xlabel('real(z)')
ylabel('imag(z)')
```



Now, calculate $H_1^{(2)}(z) \cdot e^{iz}$ on the complex plane and compare it to the unscaled function. The scaled function increases the range of computable values by avoiding overflow and loss of accuracy when `z` has a large positive imaginary part.

```
Hs = besselh(nu,k,z,scaled);
surf(x,y,imag(Hs))
xlabel('real(z)')
ylabel('imag(z)')
```



Input Arguments

nu — Equation order
scalar | vector | matrix | multidimensional array

Equation order, specified as a scalar, vector, matrix, or multidimensional array. nu specifies the order of the Hankel function on page 1-846. nu and Z must be the same size, or one of them can be scalar.

Example: `besselh(3,Z)`

Data Types: `single` | `double`

K — Kind of Hankel function

1 (default) | 2

Kind of Hankel function, specified as `1` or `2`.

- If `K = 1`, then `besselh` computes the Hankel function of the first kind $H_\nu^{(1)}(z) = J_\nu(z) + iY_\nu(z)$.
- If `K = 2`, then `besselh` computes the Hankel function of the second kind $H_\nu^{(2)}(z) = J_\nu(z) - iY_\nu(z)$.

Example: `besselh(nu,2,Z)`

Z — Functional domain

scalar | vector | matrix | multidimensional array

Functional domain, specified as a scalar, vector, matrix, or multidimensional array. `nu` and `Z` must be the same size, or one of them can be scalar.

Example: `besselh(nu,[1-1i 1+0i 1+1i])`

Data Types: `single` | `double`
Complex Number Support: Yes

`scale` — Toggle to scale function

`0` (default) | `1`

Toggle to scale function, specified as one of these values:

- `0` (default) — No scaling
- `1` — Scale the output of `besselh`, depending on the value of `K`:
  - If `K = 1`, then scale the Hankel function of the first kind $H_\nu^{(1)}(z)$ by e^{-iZ}.
  - If `K = 2`, then scale Hankel function of the second kind $H_\nu^{(2)}(z)$ by e^{+iZ}.

  On the complex plane, $H_\nu^{(1)}(z)$ overflows when `imag(Z)` is large and negative. Similarly, $H_\nu^{(2)}(z)$ overflows when `imag(Z)` is large and positive. Exponentially scaling the output of `besselh` is useful in these two cases since the function otherwise quickly loses accuracy or overflows the limits of double precision.

Example: `besselh(nu,K,Z,1)`

More About

Hankel Functions and Bessel's Equation

This differential equation, where ν is a real constant, is called *Bessel's equation*:

$$z^2\frac{d^2y}{dz^2} + z\frac{dy}{dz} + \left(z^2 - \nu^2\right)y = 0.$$

Its solutions are known as *Bessel functions*.

The *Bessel functions of the first kind*, denoted $J_\nu(z)$ and $J_{-\nu}(z)$, form a fundamental set of solutions of Bessel's equation for noninteger ν. The *Bessel functions of the second kind*, denoted $Y_\nu(z)$, form a second solution of Bessel's equation—linearly independent of $J_\nu(z)$—defined by

$$Y_\nu(z) = \frac{J_\nu(z)\cos(\nu\pi) - J_{-\nu}(z)}{\sin(\nu\pi)}.$$

The *Bessel functions of the third kind*, also called *Hankel functions of the first and second kind*, are defined by linear combinations of the Bessel functions, where $J_\nu(z)$ is `besselj`, and $Y_\nu(z)$ is `bessely`:

$$H_\nu^{(1)}(z) = J_\nu(z) + iY_\nu(z)$$

$$H_\nu^{(2)}(z) = J_\nu(z) - iY_\nu(z).$$

References

[1] Abramowitz, M., and I.A. Stegun. *Handbook of Mathematical Functions*. National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`besseli` | `besselj` | `besselk` | `bessely`

Introduced before R2006a

besseli

Modified Bessel function of first kind

Syntax

```
I = besseli(nu,Z)
I = besseli(nu,Z,scale)
```

Description

`I = besseli(nu,Z)` computes the modified Bessel function of the first kind on page 1-852 $I_\nu(z)$ for each element in array `Z`.

`I = besseli(nu,Z,scale)` specifies whether to exponentially scale the modified Bessel function of the first kind to avoid overflow or loss of accuracy. If `scale` is `1`, then the output of `besseli` is scaled by the factor `exp(-abs(real(Z)))`.

Examples

Plot Modified Bessel Functions of First Kind

Define the domain.

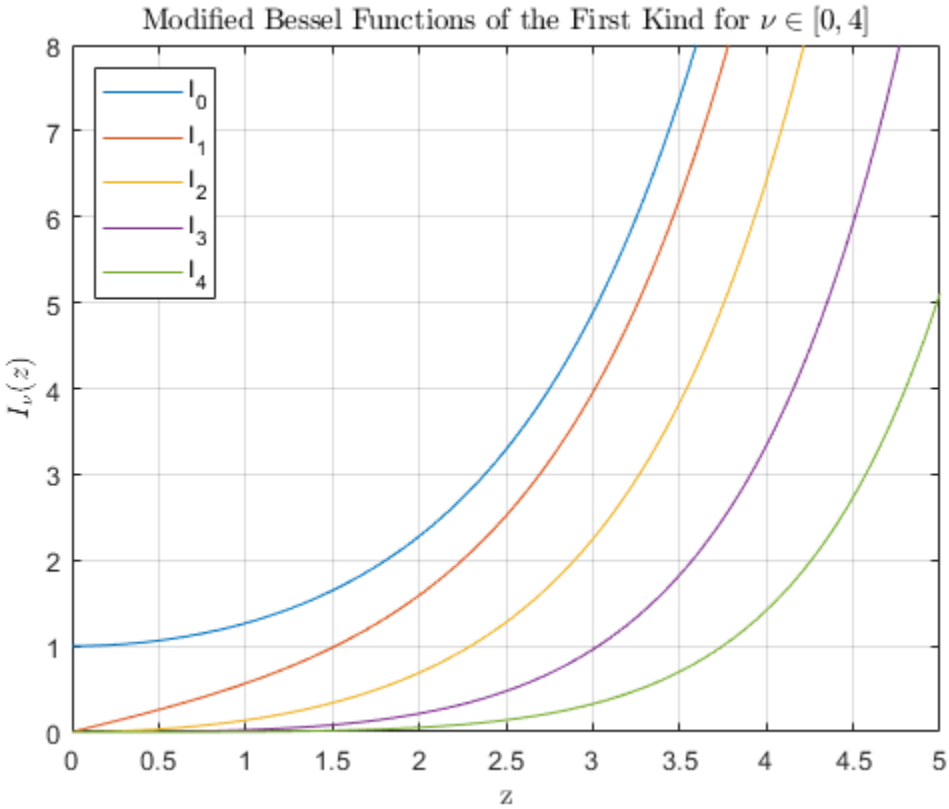
```
z = 0:0.01:5;
```

Calculate the first five modified Bessel functions of the first kind. Each row of `I` contains the values of one order of the function evaluated at the points in `z`.

```
I = zeros(5,501);
for nu = 0:4
    I(nu+1,:) = besseli(nu,z);
end
```

Plot all of the functions in the same figure.

```
plot(z,I)
axis([0 5 0 8])
grid on
legend('I_0','I_1','I_2','I_3','I_4','Location','NorthWest')
title('Modified Bessel Functions of the First Kind for $\nu \in [0,4]$','interpreter',
xlabel('z','interpreter','latex')
ylabel('$I_\nu(z)$','interpreter','latex')
```



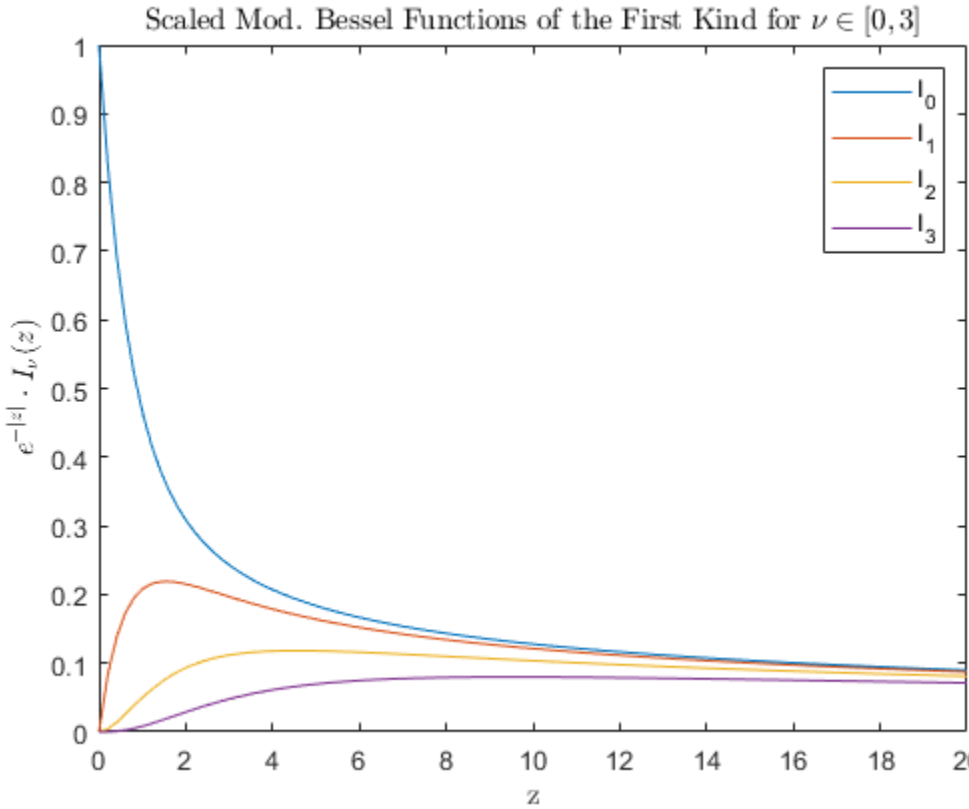
Calculate Exponentially Scaled Modified Bessel Functions

Calculate the scaled modified Bessel function of the first kind $I_\nu(z) \cdot e^{-|\mathrm{Re}[Z]|}$ for values of z in the interval $[0, 20]$ and for orders ν between 0 and 3.

```
z = linspace(0,20);
scale = 1;
Is = zeros(4,100);
for nu = 0:3
  Is(nu+1,:) = besseli(nu,z,scale);
end
```

Plot all of the functions in the same figure. For large values of z, the scaled functions do not overflow the limits of double precision, extending their range of computability compared to the unscaled functions.

```
plot(z,Is)
legend('I_0','I_1','I_2','I_3')
title('Scaled Mod. Bessel Functions of the First Kind for $\nu \in \left[0, 3 \right]$
xlabel('z','interpreter','latex')
ylabel('$e^{-|{z}|} \cdot I_\nu(z)$','interpreter','latex')
```



Input Arguments

nu — Equation order

scalar | vector | matrix | multidimensional array

Equation order, specified as a scalar, vector, matrix, or multidimensional array. nu is a real number that specifies the order of the modified Bessel function of the first kind on page 1-852. nu and Z must be the same size, or one of them can be scalar.

Example: `besseli(3,Z)`

Data Types: `single` | `double`

Z — Functional domain
scalar | vector | matrix | multidimensional array

Functional domain, specified as a scalar, vector, matrix, or multidimensional array. `besseli` is real-valued where `Z` is positive. `nu` and `Z` must be the same size, or one of them can be scalar.

Example: `besseli(nu,[1-1i 1+0i 1+1i])`

Data Types: `single` | `double`
Complex Number Support: Yes

`scale` — Toggle to scale function
`0` (default) | `1`

Toggle to scale function, specified as one of these values:

- `0` (default) — No scaling
- `1` — Scale the output of `besseli` by `exp(-abs(real(Z)))`

The magnitude of `besseli` grows rapidly as the value of `abs(real(Z))` increases, so exponentially scaling the output is useful for large values of `abs(real(Z))`, where the results otherwise quickly lose accuracy or overflow the limits of double precision.

Example: `besseli(nu,Z,1)`

More About

Modified Bessel Functions

This differential equation, where ν is a real constant, is called the *modified Bessel's equation*:

$$z^2\frac{d^2y}{dz^2} + z\frac{dy}{dz} - \left(z^2 + \nu^2\right)y = 0.$$

Its solutions are known as *modified Bessel functions*.

The *modified Bessel functions of the first kind*, denoted $I_\nu(z)$ and $I_{-\nu}(z)$, form a fundamental set of solutions of the modified Bessel's equation. $I_\nu(z)$ is defined by

$$I_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{(k=0)}^{\infty} \frac{\left(\frac{z^2}{4}\right)^k}{k!\Gamma(\nu+k+1)}.$$

The *modified Bessel functions of the second kind*, denoted $K_\nu(z)$, form a second solution independent of $I_\nu(z)$ given by

$$K_\nu(z) = \left(\frac{\pi}{2}\right)\frac{I_{-\nu}(z) - I_\nu(z)}{\sin(\nu\pi)}.$$

You can compute the modified Bessel functions of the second kind using `besselk`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If the order `nu` is less than `0`, then it must be an integer.
- Always returns a complex result.
- Strict single-precision calculations are not supported. In the generated code, single-precision inputs produce single-precision outputs. However, variables inside the function might be double-precision.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`airy` | `besselh` | `besselj` | `besselk` | `bessely`

Introduced before R2006a

besselj

Bessel function of first kind

Syntax

```
J = besselj(nu,Z)
J = besselj(nu,Z,scale)
```

Description

`J = besselj(nu,Z)` computes the Bessel function of the first kind on page 1-860 $J_\nu(z)$ for each element in array `Z`.

`J = besselj(nu,Z,scale)` specifies whether to exponentially scale the Bessel function of the first kind to avoid overflow or loss of accuracy. If `scale` is `1`, then the output of `besselj` is scaled by the factor `exp(-abs(imag(Z)))`.

Examples

Plot Bessel Functions of First Kind

Define the domain.

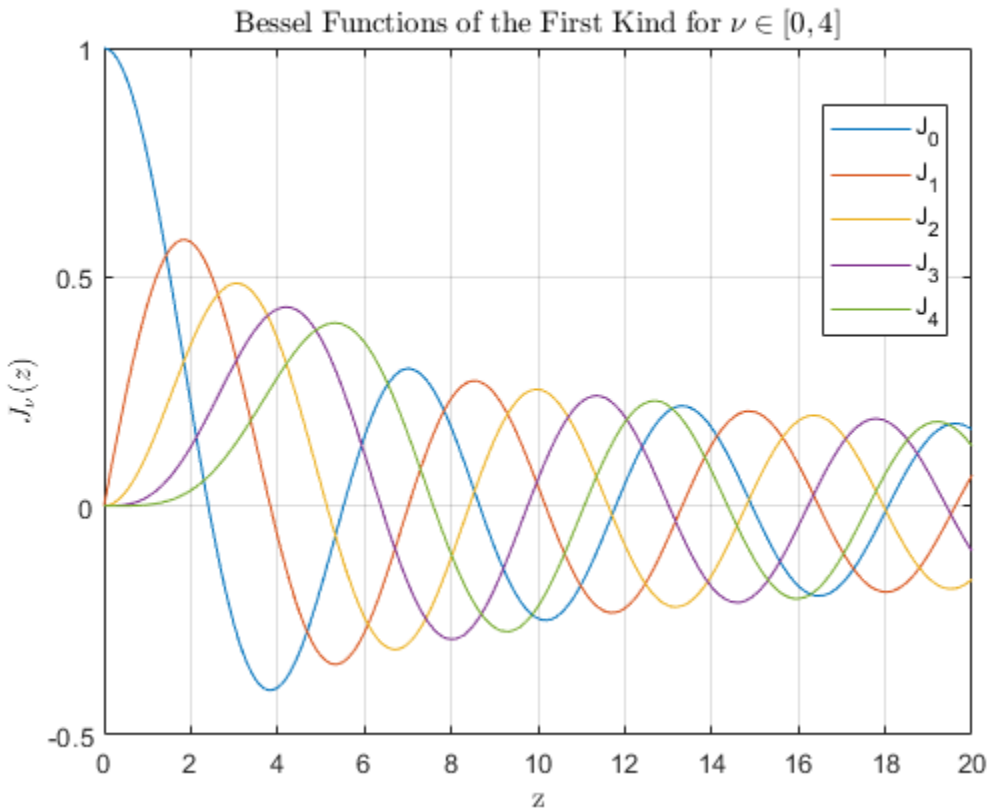
```
z = 0:0.1:20;
```

Calculate the first five Bessel functions of the first kind. Each row of `J` contains the values of one order of the function evaluated at the points in `z`.

```
J = zeros(5,201);
for i = 0:4
    J(i+1,:) = besselj(i,z);
end
```

Plot all of the functions in the same figure.

```
plot(z,J)
grid on
legend('J_0','J_1','J_2','J_3','J_4','Location','Best')
title('Bessel Functions of the First Kind for $\nu \in [0, 4]$','interpreter','latex')
xlabel('z','interpreter','latex')
ylabel('$J_\nu(z)$','interpreter','latex')
```



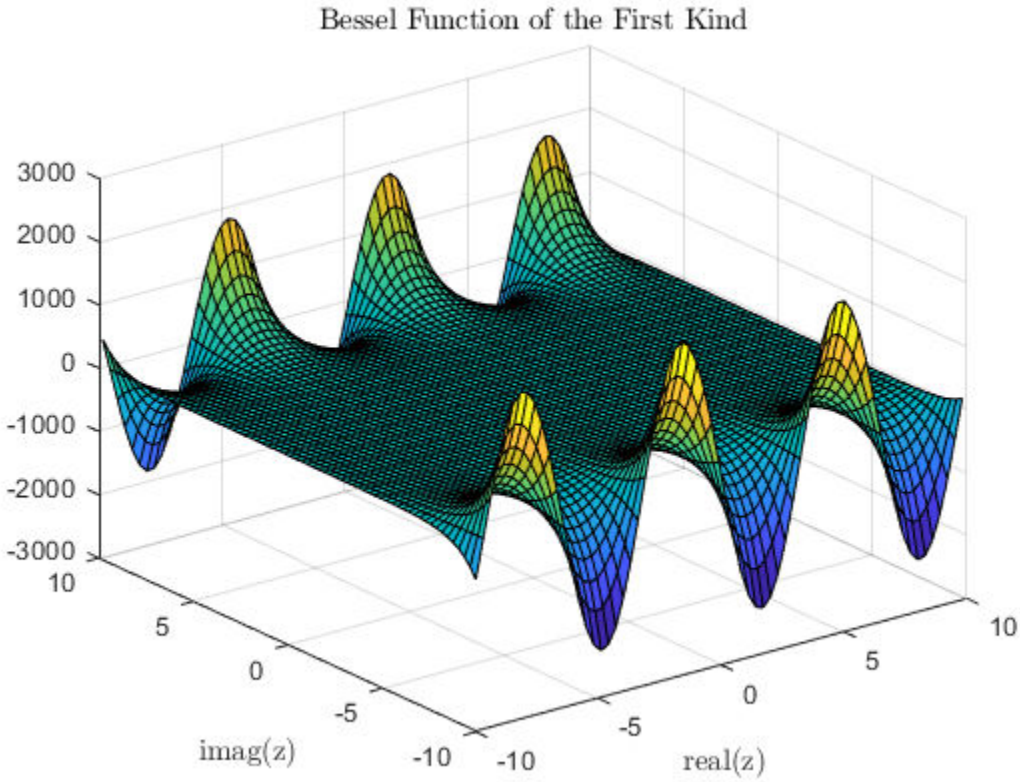
Calculate Exponentially Scaled Bessel Function

Calculate the unscaled (`J`) and scaled (`Js`) Bessel function of the first kind $J_2(z)$ for complex values of z.

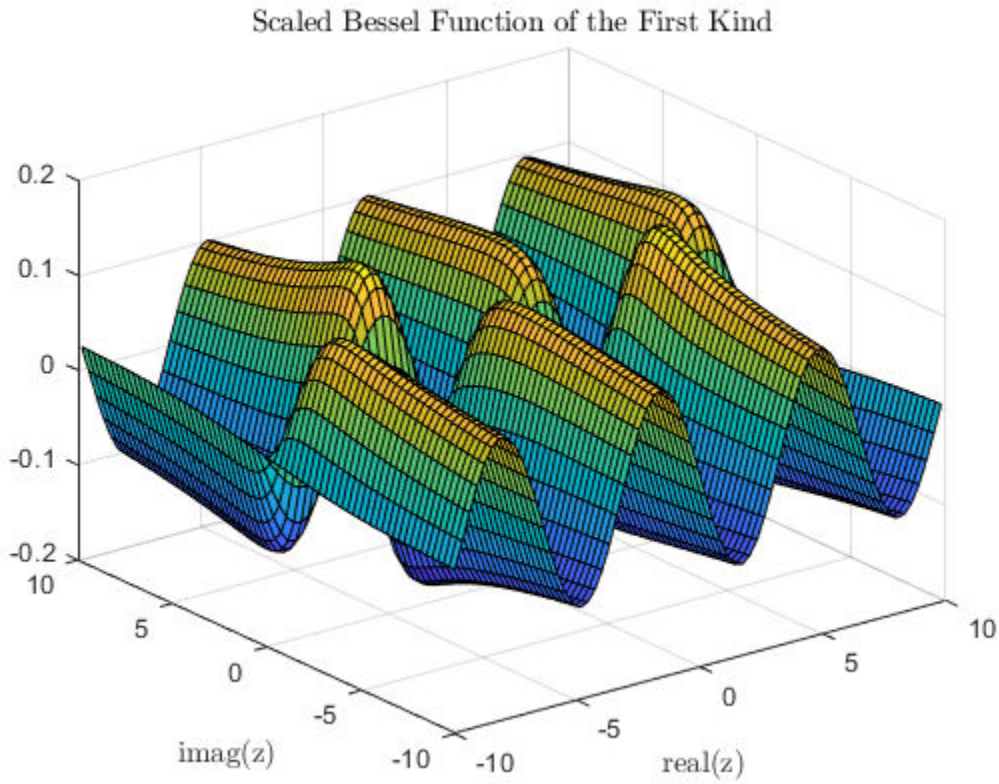
```
x = -10:0.3:10;
y = x';
z = x + 1i*y;
scale = 1;
J = besselj(2,z);
Js = besselj(2,z,scale);
```

Compare the plots of the imaginary part of the scaled and unscaled functions. For large values of `abs(imag(z))`, the unscaled function quickly overflows the limits of double precision and stops being computable. The scaled function removes this dominant exponential behavior from the calculation, and thus has a larger range of computability compared to the unscaled function.

```
surf(x,y,imag(J))
title('Bessel Function of the First Kind','interpreter','latex')
xlabel('real(z)','interpreter','latex')
ylabel('imag(z)','interpreter','latex')
```




```
surf(x,y,imag(Js))
title('Scaled Bessel Function of the First Kind','interpreter','latex')
xlabel('real(z)','interpreter','latex')
ylabel('imag(z)','interpreter','latex')
```



Input Arguments

nu — Equation order
scalar | vector | matrix | multidimensional array

Equation order, specified as a scalar, vector, matrix, or multidimensional array. nu is a real number that specifies the order of the Bessel function of the first kind on page 1-860. nu and Z must be the same size, or one of them can be scalar.

Example: `besselj(3,0:5)`

Data Types: `single` | `double`

Z — Functional domain
scalar | vector | matrix | multidimensional array

Functional domain, specified as a scalar, vector, matrix, or multidimensional array. `besselj` is real-valued where `Z` is positive. `nu` and `Z` must be the same size, or one of them can be scalar.

Example: `besselj(1,[1-1i 1+0i 1+1i])`

Data Types: `single` | `double`
Complex Number Support: Yes

`scale` — Toggle to scale function
`0` (default) | `1`

Toggle to scale function, specified as one of these values:

- `0` (default) — No scaling
- `1` — Scale the output of `besselj` by `exp(-abs(imag(Z)))`

On the complex plane, the magnitude of `besselj` grows rapidly as the value of `abs(imag(Z))` increases, so exponentially scaling the output is useful for large values of `abs(imag(Z))` where the results otherwise quickly lose accuracy or overflow the limits of double precision.

Example: `besselj(3,0:5,1)`

More About

Bessel Functions

This differential equation, where ν is a real constant, is called *Bessel's equation*:

$$z^2\frac{d^2y}{dz^2} + z\frac{dy}{dz} + \left(z^2 - \nu^2\right)y = 0.$$

Its solutions are known as *Bessel functions*.

The *Bessel functions of the first kind*, denoted $J_\nu(z)$ and $J_{-\nu}(z)$, form a fundamental set of solutions of Bessel's equation for noninteger ν. $J_\nu(z)$ is defined by

$$J_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{(k=0)}^{\infty} \frac{\left(\frac{-z^2}{4}\right)^k}{k!\Gamma(\nu+k+1)} .$$

The *Bessel functions of the second kind*, denoted $Y_\nu(z)$, form a second solution of Bessel's equation that is linearly independent of $J_\nu(z)$. $Y_\nu(z)$ is defined by

$$Y_\nu(z) = \frac{J_\nu(z)\cos(\nu\pi) - J_{-\nu}(z)}{\sin(\nu\pi)} .$$

You can calculate Bessel functions of the second kind using `bessely`.

Tips

The Bessel functions are related to the Hankel functions, also called Bessel functions of the third kind:

$$H_\nu^{(1)}(z) = J_\nu(z) + i\,Y_\nu(z)$$
$$H_\nu^{(2)}(z) = J_\nu(z) - i\,Y_\nu(z) .$$

$H_\nu^{(K)}(z)$ is `besselh`, $J_\nu(z)$ is `besselj`, and $Y_\nu(z)$ is `bessely`. The Hankel functions also form a fundamental set of solutions to Bessel's equation (see `besselh`).

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If the order `nu` is less than `0`, then it must be an integer.
- Always returns a complex result.
- Strict single-precision calculations are not supported. In the generated code, single-precision inputs produce single-precision outputs. However, variables inside the function might be double-precision.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The order `nu` must be a positive, real, integer.
- The argument `Z` must be a real value.
- The three-input syntax `J = besselj(nu,Z,scale)` is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`besselh` | `besseli` | `besselk` | `bessely`

Introduced before R2006a

besselk

Modified Bessel function of second kind

Syntax

```
K = besselk(nu,Z)
K = besselk(nu,Z,scale)
```

Description

`K = besselk(nu,Z)` computes the modified Bessel function of the second kind on page 1-867 $K_\nu(z)$ for each element in array `Z`.

`K = besselk(nu,Z,scale)` specifies whether to exponentially scale the modified Bessel function of the second kind to avoid underflow or loss of accuracy. If `scale` is `1`, then the output of `besselk` is scaled by a factor `exp(Z)`.

Examples

Plot Modified Bessel Functions of Second Kind

Define the domain.

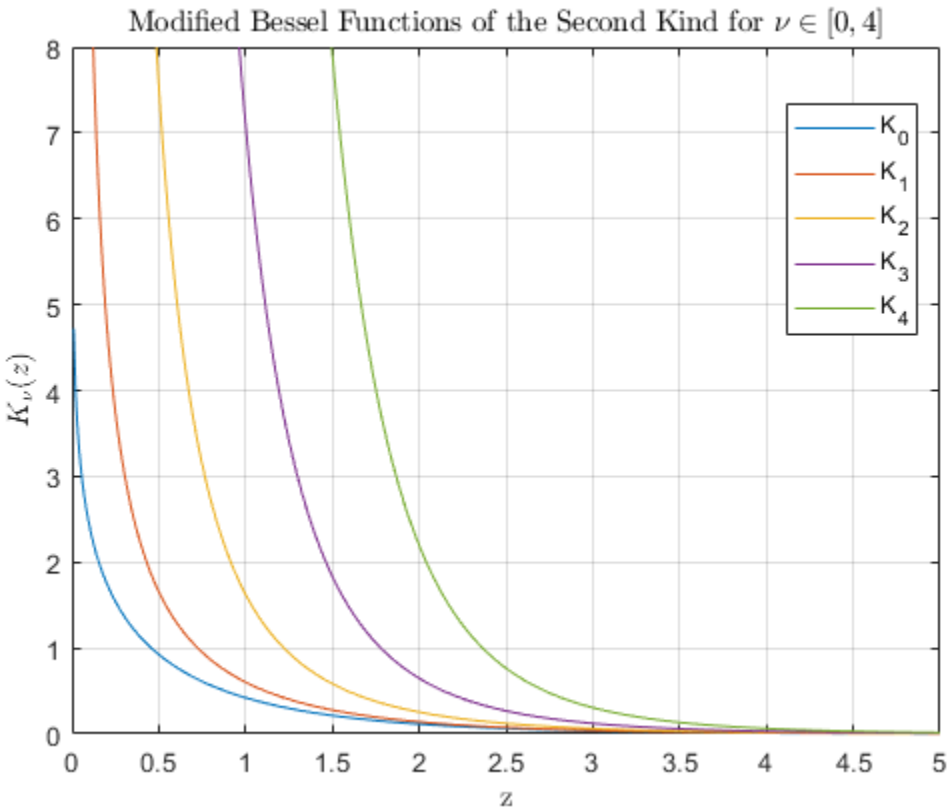
```
z = 0:0.01:5;
```

Calculate the first five modified Bessel functions of the second kind. Each row of `K` contains the values of one order of the function evaluated at the points in `z`.

```
K = zeros(5,501);
for i = 0:4
    K(i+1,:) = besselk(i,z);
end
```

Plot all of the functions in the same figure.

```
plot(z,K)
axis([0 5 0 8])
grid on
legend('K_0','K_1','K_2','K_3','K_4','Location','Best')
title('Modified Bessel Functions of the Second Kind for $\nu \in [0,4]$','interpreter'
xlabel('z','interpreter','latex')
ylabel('$K_\nu(z)$','interpreter','latex')
```



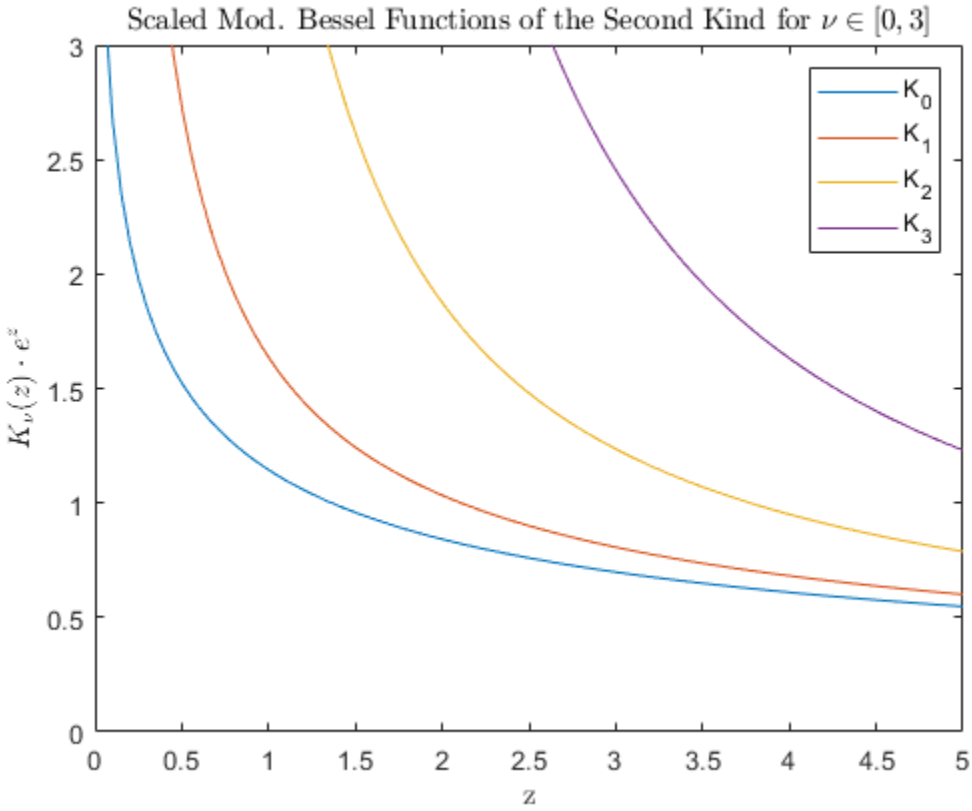
Calculate Exponentially Scaled Modified Bessel Functions

Calculate the scaled modified Bessel functions of the second kind $K_\nu(z) \cdot e^z$ for values of z in the interval $[0, 5]$ and for orders of ν between 0 and 3.

```
z = linspace(0,5);
scale = 1;
Ks = zeros(4,100);
for nu = 0:3
  Ks(nu+1,:) = besselk(nu,z,scale);
end
```

Plot all of the functions in the same figure. For large values of z, the scaled functions do not underflow the limits of double precision as quickly as the unscaled functions, extending their range of computability.

```
plot(z,Ks)
ylim([0 3])
legend('K_0','K_1','K_2','K_3')
title('Scaled Mod. Bessel Functions of the Second Kind for $\nu \in \left[0, 3 \right]$
xlabel('z','interpreter','latex')
ylabel('$K_\nu(z) \cdot e^{z}$','interpreter','latex')
```



Input Arguments

nu — Equation order

scalar | vector | matrix | multidimensional array

Equation order, specified as a scalar, vector, matrix, or multidimensional array. nu is a real number that specifies the order of the modified Bessel function of the second kind on page 1-867. nu and Z must be the same size, or one of them can be scalar.

Example: `besselk(3,Z)`

Data Types: `single` | `double`

Z — Functional domain
scalar | vector | matrix | multidimensional array

Functional domain, specified as a scalar, vector, matrix, or multidimensional array. `besselk` is real-valued where `Z` is positive. `nu` and `Z` must be the same size, or one of them can be scalar.

Example: `besselk(nu,0:3)`

Data Types: `single` | `double`
Complex Number Support: Yes

`scale` — Toggle to scale function
`0` (default) | `1`

Toggle to scale function, specified as one of these values:

- `0` (default) — No scaling
- `1` — Scale the output of `besselk` by `exp(Z)`

The value of `besselk` decreases rapidly as the value of `Z` increases, so exponentially scaling the output is useful for large values of `Z` where the results otherwise quickly lose accuracy or underflow the limits of double precision.

Example: `besselk(nu,Z,1)`

More About

Modified Bessel Functions

This differential equation, where ν is a real constant, is called the *modified Bessel's equation*:

$$z^2\frac{d^2y}{dz^2} + z\frac{dy}{dz} - \left(z^2 + \nu^2\right)y = 0.$$

Its solutions are known as *modified Bessel functions*.

The *modified Bessel functions of the first kind*, denoted $I_\nu(z)$ and $I_{-\nu}(z)$, form a fundamental set of solutions of the modified Bessel's equation. $I_\nu(z)$ is defined by

$$I_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{(k=0)}^{\infty} \frac{\left(\frac{z^2}{4}\right)^k}{k!\Gamma(\nu+k+1)}.$$

You can compute the modified Bessel functions of the first kind using `besseli`.

The *modified Bessel functions of the second kind*, denoted $K_\nu(z)$, form a second solution independent of $I_\nu(z)$ given by

$$K_\nu(z) = \left(\frac{\pi}{2}\right)\frac{I_{-\nu}(z) - I_\nu(z)}{\sin(\nu\pi)}.$$

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`airy` | `besselh` | `besseli` | `besselj` | `bessely`

Introduced before R2006a

bessely

Bessel function of second kind

Syntax

```
Y = bessely(nu,Z)
Y = bessely(nu,Z,scale)
```

Description

`Y = bessely(nu,Z)` computes the Bessel function of the second kind on page 1-874 $Y_\nu(z)$ for each element in array `Z`.

`Y = bessely(nu,Z,scale)` specifies whether to exponentially scale the Bessel function of the second kind to avoid overflow or loss of accuracy. If `scale` is `1`, then the output of `bessely` is scaled by the factor `exp(-abs(imag(Z)))`.

Examples

Plot Bessel Functions of Second Kind

Define the domain.

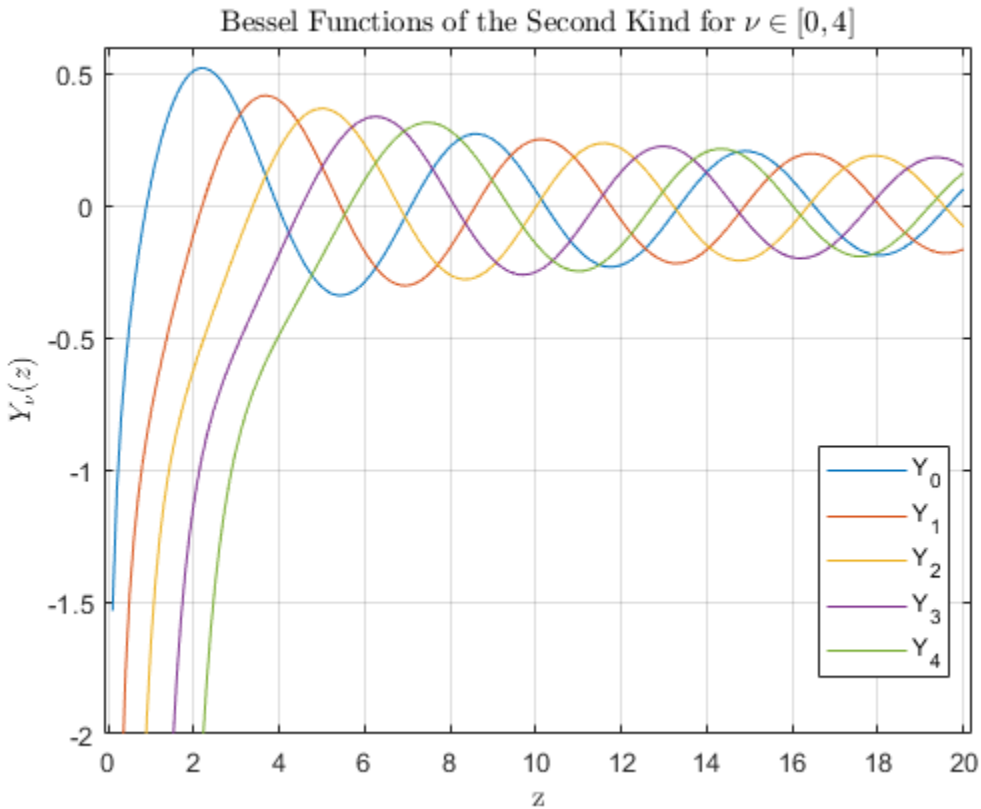
```
z = 0:0.1:20;
```

Calculate the first five Bessel functions of the second kind. Each row of `Y` contains the values of one order of the function evaluated at the points in `z`.

```
Y = zeros(5,201);
for i = 0:4
    Y(i+1,:) = bessely(i,z);
end
```

Plot all of the functions in the same figure.

```
plot(z,Y)
axis([-0.1 20.2 -2 0.6])
grid on
legend('Y_0','Y_1','Y_2','Y_3','Y_4','Location','Best')
title('Bessel Functions of the Second Kind for $\nu \in [0, 4]$','interpreter','latex')
xlabel('z','interpreter','latex')
ylabel('$Y_\nu(z)$','interpreter','latex')
```



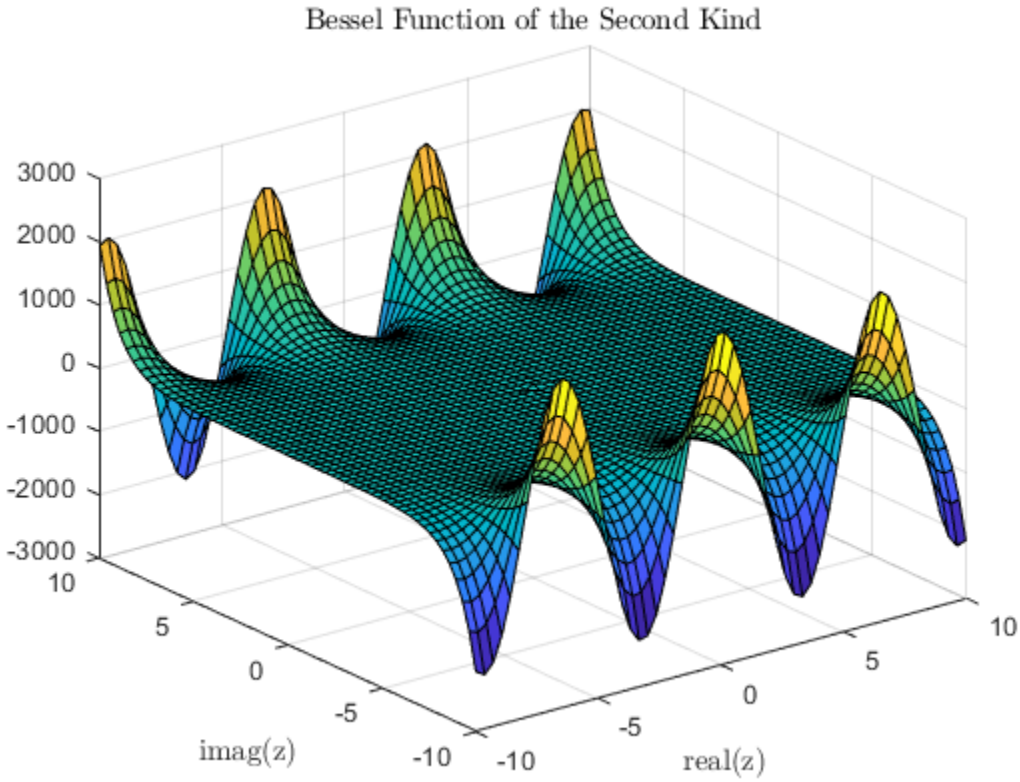
Calculate Exponentially Scaled Bessel Function

Calculate the unscaled (`Y`) and scaled (`Ys`) Bessel function of the second kind $Y_2(z)$ for complex values of z.

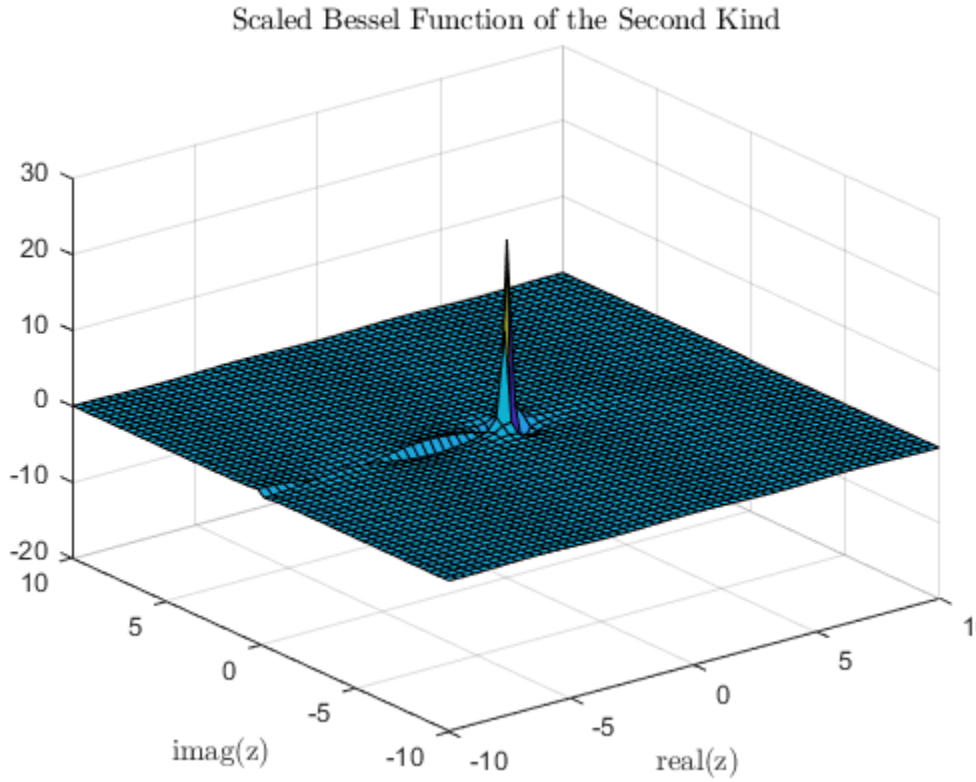
```
x = -10:0.35:10;
y = x';
z = x + 1i*y;
scale = 1;
Y = bessely(2,z);
Ys = bessely(2,z,scale);
```

Compare the plots of the imaginary part of the scaled and unscaled functions. For large values of `abs(imag(z))`, the unscaled function quickly overflows the limits of double precision and stops being computable. The scaled function removes this dominant exponential behavior from the calculation and thus has a larger range of computability compared to the unscaled function.

```
surf(x,y,imag(Y))
title('Bessel Function of the Second Kind','interpreter','latex')
xlabel('real(z)','interpreter','latex')
ylabel('imag(z)','interpreter','latex')
```




```
surf(x,y,imag(Ys))
title('Scaled Bessel Function of the Second Kind','interpreter','latex')
xlabel('real(z)','interpreter','latex')
ylabel('imag(z)','interpreter','latex')
```



Input Arguments

nu — Equation order

scalar | vector | matrix | multidimensional array

Equation order, specified as a scalar, vector, matrix, or multidimensional array. nu is a real number that specifies the order of the Bessel function of the second kind on page 1-874. nu and Z must be the same size, or one of them can be scalar.

Example: `bessely(3,0:5)`

Data Types: `single` | `double`

Z — Functional domain
scalar | vector | matrix | multidimensional array

Functional domain, specified as a scalar, vector, matrix, or multidimensional array. `bessely` is real-valued where `Z` is positive. `nu` and `Z` must be the same size, or one of them can be scalar.

Example: `bessely(1,[1-1i 1+0i 1+1i])`

Data Types: `single` | `double`
Complex Number Support: Yes

`scale` — Toggle to scale function
`0` (default) | `1`

Toggle to scale function, specified as one of these values:

- `0` (default) — No scaling
- `1` — Scale the output of `bessely` by `exp(-abs(imag(Z)))`

On the complex plane, the magnitude of `bessely` grows rapidly as the value of `abs(imag(Z))` increases, so exponentially scaling the output is useful for large values of `abs(imag(Z))` where the results otherwise quickly lose accuracy or overflow the limits of double precision.

Example: `bessely(3,0:5,1)`

More About

Bessel Functions

This differential equation, where ν is a real constant, is called *Bessel's equation*:

$$z^2\frac{d^2y}{dz^2} + z\frac{dy}{dz} + \left(z^2 - \nu^2\right)y = 0.$$

Its solutions are known as *Bessel functions*.

The *Bessel functions of the first kind*, denoted $J_\nu(z)$ and $J_{-\nu}(z)$, form a fundamental set of solutions of Bessel's equation for noninteger ν. $J_\nu(z)$ is defined by

$$J_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{(k=0)}^{\infty} \frac{\left(\frac{-z^2}{4}\right)^k}{k!\Gamma(\nu+k+1)} .$$

You can calculate Bessel functions of the first kind using `besselj`.

The *Bessel functions of the second kind*, denoted $Y_\nu(z)$, form a second solution of Bessel's equation that is linearly independent of $J_\nu(z)$. $Y_\nu(z)$ is defined by

$$Y_\nu(z) = \frac{J_\nu(z)\cos(\nu\pi) - J_{-\nu}(z)}{\sin(\nu\pi)} .$$

Tips

The Bessel functions are related to the Hankel functions, also called Bessel functions of the third kind:

$$H_\nu^{(1)}(z) = J_\nu(z) + i\,Y_\nu(z)$$
$$H_\nu^{(2)}(z) = J_\nu(z) - i\,Y_\nu(z) .$$

$H_\nu^{(K)}(z)$ is `besselh`, $J_\nu(z)$ is `besselj`, and $Y_\nu(z)$ is `bessely`. The Hankel functions also form a fundamental set of solutions to Bessel's equation (see `besselh`).

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The order `nu` must be a positive, real, integer value.
- The argument `Z` must be a positive real value.
- The three-input syntax `J = bessely(nu,Z,scale)` is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`besselh` | `besseli` | `besselj` | `besselk`

Introduced before R2006a

beta

Beta function

Syntax

```
B = beta(Z,W)
```

Description

`B = beta(Z,W)` returns the beta function on page 1-880 evaluated at the elements of `Z` and `W`. Both `Z` and `W` must be real and nonnegative.

Examples

Compute Beta Function for Integer Arguments

Compute the beta function for integer arguments $w = 3$ and $z = 1, \ldots, 10$. Based on the definition, the beta function can be calculated as

$$B(z,3) = \frac{\Gamma(z)\Gamma(3)}{\Gamma(z+3)} = \frac{(z-1)!\,2!}{(z+2)!} = \frac{2}{z(z+1)(z+2)}.$$

Set the output format to rational to show the results as ratios of integers.

```
format rat
B = beta((1:10)',3)
```

```
B =
       1/3
       1/12
       1/30
       1/60
       1/105
       1/168
       1/252
```

```
1/360
1/495
1/660
```

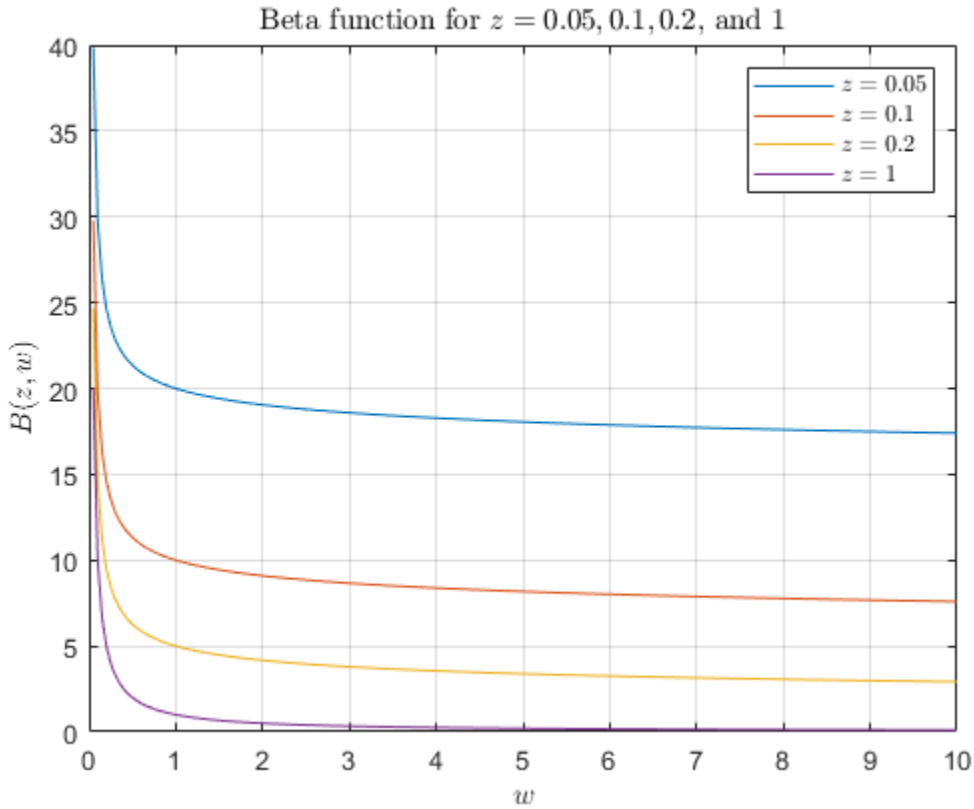
Plot Beta Function

Calculate the beta function for $z = 0.05$, 0.1, 0.2, and 1 within the interval $0 \le w \le 10$. Loop over values of z, evaluate the function at each one, and assign each result to a row of B.

```
Z = [0.05 0.1 0.2 1];
W = 0:0.05:10;
B = zeros(4,201);
for i = 1:4
    B(i,:) = beta(Z(i),W);
end
```

Plot all of the beta functions in the same figure.

```
plot(W,B)
grid on
legend('$z = 0.05$','$z = 0.1$','$z = 0.2$','$z = 1$','interpreter','latex')
title('Beta function for $z = 0.05, 0.1, 0.2$, and $1$','interpreter','latex')
xlabel('$w$','interpreter','latex')
ylabel('$B(z,w)$','interpreter','latex')
```



Input Arguments

Z — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. The elements of `Z` must be real and nonnegative. `Z` and `W` must be the same size, or else one of them must be a scalar.

Data Types: `single` | `double`

W — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. The elements of W must be real and nonnegative. Z and W must be the same size, or else one of them must be a scalar.

- If Z or W is equal to 0, the beta function returns Inf.
- If Z and W are both 0, the beta function returns NaN.

Data Types: single | double

More About

Beta Function

The beta function is defined by

$$B(z,w) = \int_0^1 t^{z-1}(1-t)^{w-1}dt = \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)}.$$

The $\Gamma(z)$ term is the gamma function

$$\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}dt.$$

References

[1] Olver, F. W. J., A. B. Olde Daalhuis, D. W. Lozier, B. I. Schneider, R. F. Boisvert, C. W. Clark, B. R. Miller, and B. V. Saunders, eds., Chapter 5.12 Beta Function, *NIST Digital Library of Mathematical Functions*, Release 1.0.22, Mar. 15, 2018.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`betainc` | `betaln` | `gamma` | `gammaln`

Introduced before R2006a

betainc

Incomplete beta function

Syntax

```
I = betainc(X,Z,W)
I = betainc(X,Z,W,tail)
```

Description

`I = betainc(X,Z,W)` computes the incomplete beta function for corresponding elements of the arrays `X`, `Z`, and `W`. The elements of `X` must be in the closed interval [0,1]. The arrays `Z` and `W` must be nonnegative and real. All arrays must be the same size, or any of them can be scalar.

`I = betainc(X,Z,W,tail)` specifies the tail of the incomplete beta function. Choices are:

`'lower'` (the default)	Computes the integral from `0` to `x`
`'upper'`	Computes the integral from `x` to `1`

These functions are related as follows:

```
1-betainc(X,Z,W) = betainc(X,Z,W,'upper')
```

Note that especially when the upper tail value is close to `0`, it is more accurate to use the `'upper'` option than to subtract the `'lower'` value from `1`.

Examples

Compute Incomplete Beta Function

Compute the incomplete beta function corresponding to the elements of `Z` according to the parameters `X` and `W`.

```
format longG
X = 0.5;
Z = (1:10)';
W = 3;
I = betainc(X,Z,W)
```

```
I = 10×1

                     0.875
                    0.6875
                       0.5
                   0.34375
                 0.2265625
               0.14453125
               0.08984375
                0.0546875
            0.03271484375
           0.019287109375
```

More About

Incomplete Beta Function

The incomplete beta function is

$$I_x(z,w) = \frac{1}{B(z,w)} \int_0^x t^{z-1}(1-t)^{w-1} dt$$

where $B(z,w)$, the beta function, is defined as

$$B(z,w) = \int_0^1 t^{z-1}(1-t)^{w-1} dt = \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)}$$

and $\Gamma(z)$ is the gamma function.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Always returns a complex result.
- Strict single-precision calculations are not supported. In the generated code, single-precision inputs produce single-precision outputs. However, variables inside the function might be double-precision.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`beta` | `betaln`

Introduced before R2006a

betaincinv

Beta inverse cumulative distribution function

Syntax

```
x = betaincinv(y,z,w)
x = betaincinv(y,z,w,tail)
```

Description

`x = betaincinv(y,z,w)` computes the inverse incomplete beta function for corresponding elements of `y`, `z`, and `w`, such that `y = betainc(x,z,w)`. The elements of `y` must be in the closed interval [0,1], and those of `z` and `w` must be nonnegative. `y`, `z`, and `w` must all be real and the same size (or any of them can be scalar).

`x = betaincinv(y,z,w,tail)` specifies the tail of the incomplete beta function. Choices are `'lower'` (the default) to use the integral from 0 to `x`, or `'upper'` to use the integral from `x` to 1. These two choices are related as follows: `betaincinv(y,z,w,'upper') = betaincinv(1-y,z,w,'lower')`. When `y` is close to 0, the `'upper'` option provides a way to compute `x` more accurately than by subtracting `y` from 1.

More About

Inverse Incomplete Beta Function

The incomplete beta function is defined as

$$I_x(z,w) = \frac{1}{\beta(z,w)} \int_0^x t^{(z-1)}(1-t)^{(w-1)} dt$$

`betaincinv` computes the inverse of the incomplete beta function with respect to the integration limit `x` using Newton's method.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Always returns a complex result.
- Strict single-precision calculations are not supported. In the generated code, single-precision inputs produce single-precision outputs. However, variables inside the function might be double-precision.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`beta` | `betainc` | `betaln`

betaln

Logarithm of beta function

Syntax

```
L = betaln(Z,W)
```

Description

`L = betaln(Z,W)` computes the natural logarithm of the `beta` function `log(beta(Z,W))`, for corresponding elements of arrays `Z` and `W`, without computing `beta(Z,W)`. Since the beta function can range over very large or very small values, its logarithm is sometimes more useful.

`Z` and `W` must be real and nonnegative. They must be the same size, or either can be scalar.

Examples

Compute Natural Log of Beta Function

Compute the natural logarithm of the beta function according to the value of `X` without directly computing the beta function. `beta(X,X)` results in floating point arithmetic underflow.

```
X = 510;
betaln(X,X)
```

```
ans = -708.8616
```

Algorithms

```
betaln(z,w) = gammaln(z)+gammaln(w)-gammaln(z+w)
```

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`beta` | `betainc` | `gammaln`

Introduced before R2006a

between

Calendar math differences

Syntax

```
dt = between(t1,t2)
dt = between(t1,t2,components)
```

Description

`dt = between(t1,t2)` returns the differences between the datetime values in `t1` and `t2`. The `dt` output is a `calendarDuration` array in terms of the calendar components years, months, days, and time, such that `t2 = t1+dt`.

`dt = between(t1,t2,components)` returns the differences between datetime values in terms of the specified calendar or time components.

Examples

Differences Between Two Datetime Arrays

Create two `datetime` arrays.

```
t1 = datetime('now')
```

```
t1 = datetime
   27-Aug-2019 00:09:18
```

```
t2 = datetime('tomorrow','Format','dd-MMM-yyyy HH:mm:ss') + caldays(0:2)
```

```
t2 = 1x3 datetime array
   28-Aug-2019 00:00:00   29-Aug-2019 00:00:00   30-Aug-2019 00:00:00
```

Find the difference between the two arrays.

```
dt = between(t1,t2)
```

```
dt = 1x3 calendarDuration array
      23h 50m 41.883s   1d 23h 50m 41.883s   2d 23h 50m 41.883s
```

`between` returns a `calendarDuration` array containing differences in terms of days, hours, minutes, and seconds.

Difference Between Datetime Values in Calendar Days

Create a sequence of datetimes over a 6-month period. Then, find the number of days between the first date and each of the dates in the sequence.

```
t1 = datetime(2013,1,1);
t2 = dateshift(t1,'end','month',0:4)
```

```
t2 = 1x5 datetime array
   31-Jan-2013   28-Feb-2013   31-Mar-2013   30-Apr-2013   31-May-2013
```

```
dt = between(t1,t2,'Days')
```

```
dt = 1x5 calendarDuration array
    30d    58d    89d   119d   150d
```

Input Arguments

`t1` — Input date and time
`datetime` array | character vector | cell array of character vectors | string array

Input date and time, specified as a `datetime` array, character vector, cell array of character vectors, or string array. At least one of inputs `t1` and `t2` must be a `datetime` array. `t1` and `t2` must be the same size unless one is a scalar.

`t2` — Input date and time
`datetime` array | character vector | cell array of character vectors | string array

Input date and time, specified as a `datetime` array, character vector, cell array of character vectors, or string array. At least one of inputs `t1` and `t2` must be a `datetime` array. `t1` and `t2` must be the same size unless one is a scalar.

`components` — Calendar or time components
`'years'` | `'quarters'` | `'months'` | `'weeks'` | `'days'` | `'time'` | cell array of character vectors | string array

Calendar or time components, specified as one of the following character vectors, or a cell array or string array containing one or more of these values:

- `'years'`
- `'quarters'`
- `'months'`
- `'weeks'`
- `'days'`
- `'time'`

Except for `'time'`, the above components are flexible lengths of time. For example, one month represents a different length of time when added to a datetime in January than when added to a datetime in February.

`between` operates on the calendar or time components in decreasing order, starting with the largest component.

In general, `t2` is not equal to `t1 + dt`, unless you include `'time'` in `components`.

Example: `{'years','quarters'}`

Data Types: `char` | `cell` | `string`

Output Arguments

`dt` — Difference array
`calendarDuration` array

Difference array, returned as a `calendarDuration` array.

Tips

- To compute differences between datetime values in `t1` and `t2` as exact, fixed-length durations, use `t2-t1`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`caldiff` | `calendarDuration` | `diff` | `minus`

Introduced in R2014b

bicg

Biconjugate gradients method

Syntax

```
x = bicg(A,b)
bicg(A,b,tol)
bicg(A,b,tol,maxit)
bicg(A,b,tol,maxit,M)
bicg(A,b,tol,maxit,M1,M2)
bicg(A,b,tol,maxit,M1,M2,x0)
[x,flag] = bicg(A,b,...)
[x,flag,relres] = bicg(A,b,...)
[x,flag,relres,iter] = bicg(A,b,...)
[x,flag,relres,iter,resvec] = bicg(A,b,...)
```

Description

`x = bicg(A,b)` attempts to solve the system of linear equations `A*x = b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and should be large and sparse. The column vector `b` must have length `n`. `A` can be a function handle, `afun`, such that `afun(x,'notransp')` returns `A*x` and `afun(x,'transp')` returns `A'*x`.

“Parameterizing Functions” explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `bicg` converges, it displays a message to that effect. If `bicg` fails to converge after the maximum number of iterations or halts for any reason, it prints a warning message that includes the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`bicg(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `bicg` uses the default, `1e-6`.

`bicg(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `bicg` uses the default, `min(n,20)`.

`bicg(A,b,tol,maxit,M)` and `bicg(A,b,tol,maxit,M1,M2)` use the preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `bicg` applies no preconditioner. `M` can be a function handle `mfun`, such that `mfun(x,'notransp')` returns `M\x` and `mfun(x,'transp')` returns `M'\x`.

`bicg(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is `[]`, then `bicg` uses the default, an all-zero vector.

`[x,flag] = bicg(A,b,...)` also returns a convergence flag.

Flag	Convergence
`0`	`bicg` converged to the desired tolerance `tol` within `maxit` iterations.
`1`	`bicg` iterated `maxit` times but did not converge.
`2`	Preconditioner `M` was ill-conditioned.
`3`	`bicg` stagnated. (Two consecutive iterates were the same.)
`4`	One of the scalar quantities calculated during `bicg` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = bicg(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = bicg(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = bicg(A,b,...)` also returns a vector of the residual norms at each iteration including `norm(b-A*x0)`.

Examples

Using bicg with a Matrix Input

This example shows how to use `bicg` with a matrix input. `bicg`. The following code:

```
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);

x = bicg(A,b,tol,maxit,M1,M2);
```

displays this message:

```
bicg converged at iteration 9 to a solution with relative
residual 5.3e-009
```

Using bicg with a Function Handle

This example replaces the matrix `A` in the previous example with a handle to a matrix-vector product function `afun`. The example is contained in a file `run_bicg` that

- Calls `bicg` with the `@afun` function handle as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_bicg` are available to `afun`.

Place the following into a file called `run_bicg`:

```
function x1 = run_bicg
n = 100;
on = ones(n,1);
b = afun(on,'notransp');
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x1 = bicg(@afun,b,tol,maxit,M1,M2);

    function y = afun(x,transp_flag)
       if strcmp(transp_flag,'transp')      % y = A'*x
          y = 4 * x;
          y(1:n-1) = y(1:n-1) - 2 * x(2:n);
          y(2:n) = y(2:n) - x(1:n-1);
       elseif strcmp(transp_flag,'notransp') % y = A*x
```

```
            y = 4 * x;
            y(2:n) = y(2:n) - 2 * x(1:n-1);
            y(1:n-1) = y(1:n-1) - x(2:n);
        end
    end
end
```

When you enter

```
x1 = run_bicg;
```

MATLAB software displays the message

```
bicg converged at iteration 9 to a solution with ...
relative residual
5.3e-009
```

Using bicg with a Preconditioner

This example demonstrates the use of a preconditioner.

Load `A = west0479`, a real 479-by-479 nonsymmetric sparse matrix.

```
load west0479;
A = west0479;
```

Define `b` so that the true solution is a vector of all ones.

```
b = full(sum(A,2));
```

Set the tolerance and maximum number of iterations.

```
tol = 1e-12;
maxit = 20;
```

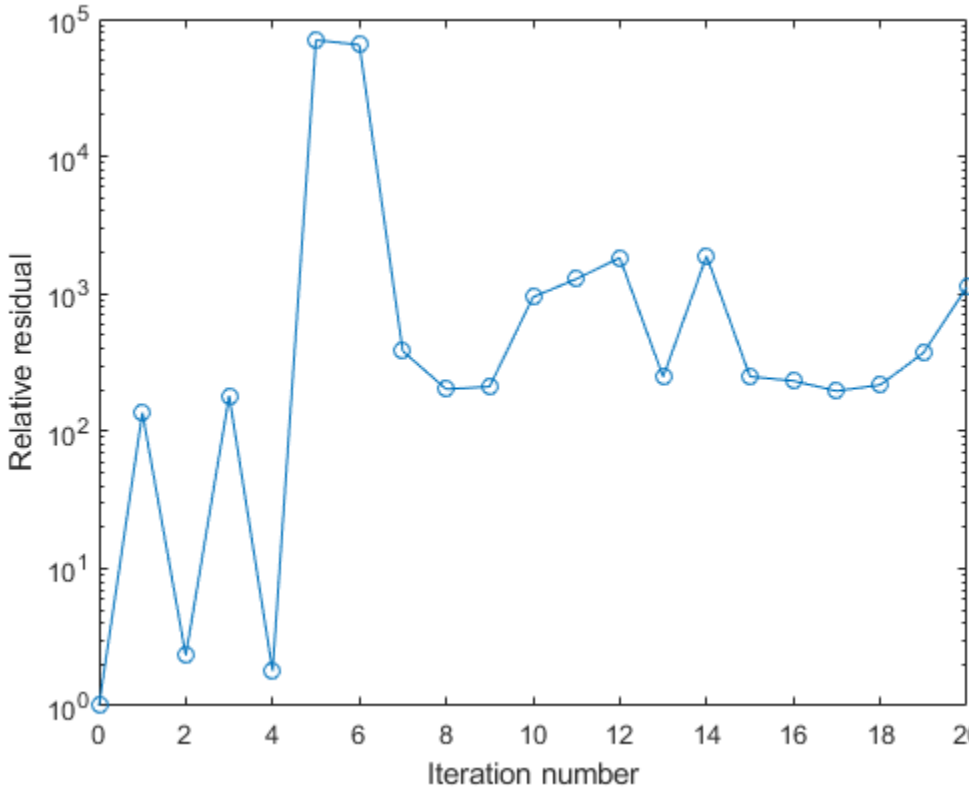
Use `bicg` to find a solution at the requested tolerance and number of iterations.

```
[x0,fl0,rr0,it0,rv0] = bicg(A,b,tol,maxit);
```

`fl0` is 1 because `bicg` does not converge to the requested tolerance `1e-12` within the requested 20 iterations. In fact, the behavior of `bicg` is so poor that the initial guess (`x0 = zeros(size(A,2),1)`) is the best solution and is returned as indicated by `it0 = 0`. MATLAB® stores the residual history in `rv0`.

Plot the behavior of `bicg`.

```
semilogy(0:maxit,rv0/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



The plot shows that the solution does not converge. You can use a preconditioner to improve the outcome.

Create the preconditioner with `ilu`, since the matrix `A` is nonsymmetric.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-5));
```

```
Error using ilu
There is a pivot equal to zero. Consider decreasing
the drop tolerance or consider using the 'udiag' option.
```

MATLAB cannot construct the incomplete LU as it would result in a singular factor, which is useless as a preconditioner.

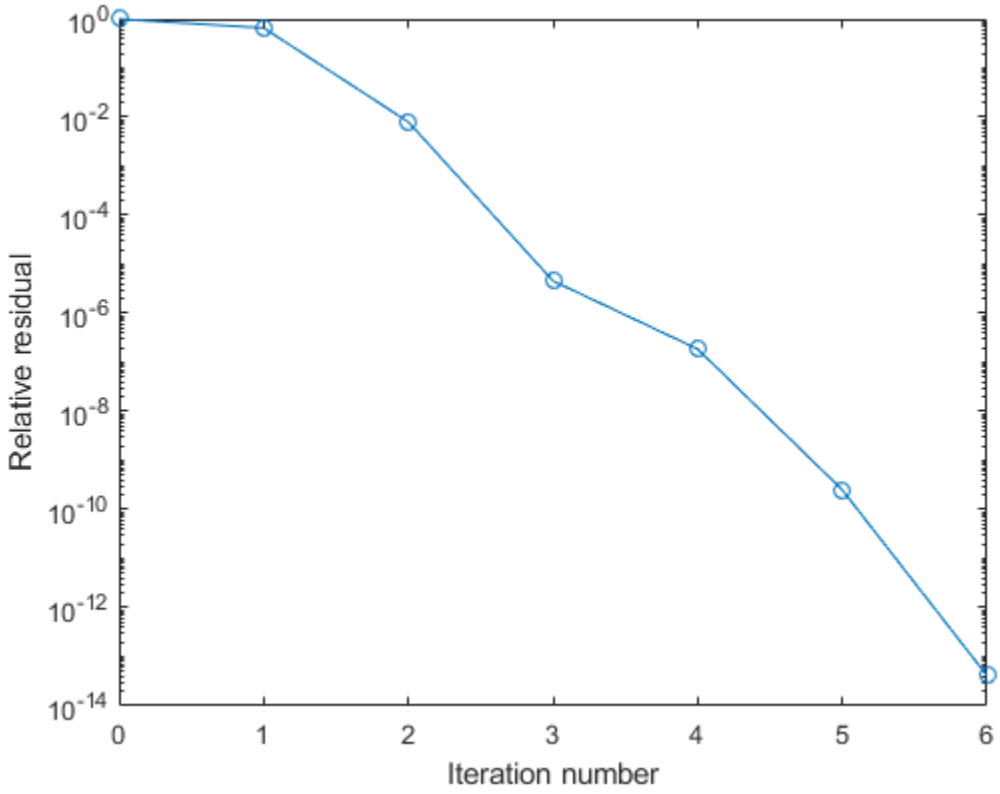
You can try again with a reduced drop tolerance, as indicated by the error message.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
[x1,fl1,rr1,it1,rv1] = bicg(A,b,tol,maxit,L,U);
```

`fl1` is 0 because `bicg` drives the relative residual to `4.1410e-014` (the value of `rr1`). The relative residual is less than the prescribed tolerance of `1e-12` at the sixth iteration (the value of `it1`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. The output `rv1(1)` is `norm(b)`, and the output `rv1(7)` is `norm(b-A*x2)`.

You can follow the progress of `bicg` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0).

```
semilogy(0:it1,rv1/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



References

[1] Barrett, R., M. Berry, T.F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When input `A` is a sparse matrix:
  - Only one sparse matrix preconditioner `M` is supported.
  - If you use two preconditioners, `M1` and `M2`, then both of them must be functions.
  - For GPU arrays, `bigc` does not detect stagnation (Flag 3). Instead, it reports failure to converge (Flag 1).

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- If `M1` is a function, then it is applied independently to each row.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bicgstab` | `cgs` | `gmres` | `ilu` | `lsqr` | `minres` | `mldivide` | `pcg` | `qmr` | `symmlq`

Topics

“Create Function Handle”

Introduced before R2006a

bicgstab

Biconjugate gradients stabilized method

Syntax

```
x = bicgstab(A,b)
bicgstab(A,b,tol)
bicgstab(A,b,tol,maxit)
bicgstab(A,b,tol,maxit,M)
bicgstab(A,b,tol,maxit,M1,M2)
bicgstab(A,b,tol,maxit,M1,M2,x0)
[x,flag] = bicgstab(A,b,...)
[x,flag,relres] = bicgstab(A,b,...)
[x,flag,relres,iter] = bicgstab(A,b,...)
[x,flag,relres,iter,resvec] = bicgstab(A,b,...)
```

Description

`x = bicgstab(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and should be large and sparse. The column vector `b` must have length `n`. `A` can be a function handle, `afun`, such that `afun(x)` returns `A*x`.

“Parameterizing Functions” explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `bicgstab` converges, a message to that effect is displayed. If `bicgstab` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`bicgstab(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `bicgstab` uses the default, `1e-6`.

`bicgstab(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `bicgstab` uses the default, `min(n,20)`.

`bicgstab(A,b,tol,maxit,M)` and `bicgstab(A,b,tol,maxit,M1,M2)` use preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `bicgstab` applies no preconditioner. `M` can be a function handle `mfun`, such that `mfun(x)` returns `M\x`.

`bicgstab(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is `[]`, then `bicgstab` uses the default, an all zero vector.

`[x,flag] = bicgstab(A,b,...)` also returns a convergence flag.

Flag	Convergence
`0`	`bicgstab` converged to the desired tolerance `tol` within `maxit` iterations.
`1`	`bicgstab` iterated `maxit` times but did not converge.
`2`	Preconditioner `M` was ill-conditioned.
`3`	`bicgstab` stagnated. (Two consecutive iterates were the same.)
`4`	One of the scalar quantities calculated during `bicgstab` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = bicgstab(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = bicgstab(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`. `iter` can be an integer + 0.5, indicating convergence halfway through an iteration.

`[x,flag,relres,iter,resvec] = bicgstab(A,b,...)` also returns a vector of the residual norms at each half iteration, including `norm(b-A*x0)`.

Examples

Using bicgstab with a Matrix Input

This example first solves `Ax = b` by providing `A` and the preconditioner `M1` directly as arguments.

The code:

```
A = gallery('wilk',21);
b = sum(A,2);
tol = 1e-12;
maxit = 15;
M1 = diag([10:-1:1 1 1:10]);

x = bicgstab(A,b,tol,maxit,M1);
```

displays the message:

```
bicgstab converged at iteration 12.5 to a solution with relative
residual 2e-014.
```

Using bicgstab with a Function Handle

This example replaces the matrix `A` in the previous example with a handle to a matrix-vector product function `afun`, and the preconditioner `M1` with a handle to a backsolve function `mfun`. The example is contained in a file `run_bicgstab` that

- Calls `bicgstab` with the function handle `@afun` as its first argument.
- Contains `afun` and `mfun` as nested functions, so that all variables in `run_bicgstab` are available to `afun` and `mfun`.

The following shows the code for `run_bicgstab`:

```
function x1 = run_bicgstab
n = 21;
b = afun(ones(n,1));
tol = 1e-12;
maxit = 15;
x1 = bicgstab(@afun,b,tol,maxit,@mfun);

    function y = afun(x)
```

```
        y = [0; x(1:n-1)] + ...
            [((n-1)/2:-1:0)'; (1:(n-1)/2)'].*x + ...
            [x(2:n); 0];
    end

    function y = mfun(r)
        y = r ./ [((n-1)/2:-1:1)'; 1; (1:(n-1)/2)'];
    end
end
```

When you enter

```
x1 = run_bicgstab;
```

MATLAB software displays the message

```
bicgstab converged at iteration 12.5 to a solution with relative
residual 2e-014.
```

Using bicgstab with a Preconditioner

This example demonstrates the use of a preconditioner.

Load `west0479`, a real 479-by-479 nonsymmetric sparse matrix.

```
load west0479;
A = west0479;
```

Define `b` so that the true solution is a vector of all ones.

```
b = full(sum(A,2));
```

Set the tolerance and maximum number of iterations.

```
tol = 1e-12;
maxit = 20;
```

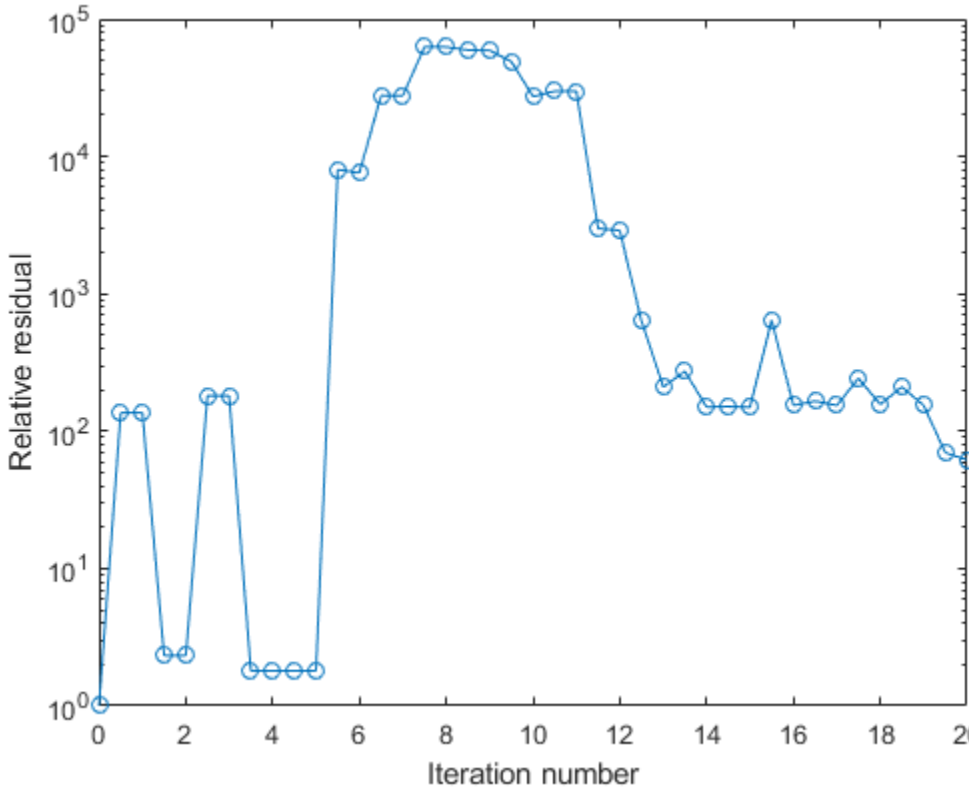
Use `bicgstab` to find a solution at the requested tolerance and number of iterations.

```
[x0,fl0,rr0,it0,rv0] = bicgstab(A,b,tol,maxit);
```

`fl0` is 1 because `bicgstab` does not converge to the requested tolerance `1e-12` within the requested 20 iterations. In fact, the behavior of `bicgstab` is so bad that the initial guess (`x0 = zeros(size(A,2),1)`) is the best solution and is returned as indicated by `it0 = 0`. MATLAB® stores the residual history in `rv0`.

Plot the behavior of `bicgstab`.

```
semilogy(0:0.5:maxit,rv0/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



The plot shows that the solution does not converge. You can use a preconditioner to improve the outcome.

Create a preconditioner with `ilu`, since `A` is nonsymmetric.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-5));
```

```
Error using ilu
There is a pivot equal to zero. Consider decreasing
the drop tolerance or consider using the 'udiag' option.
```

MATLAB cannot construct the incomplete LU as it would result in a singular factor, which is useless as a preconditioner.

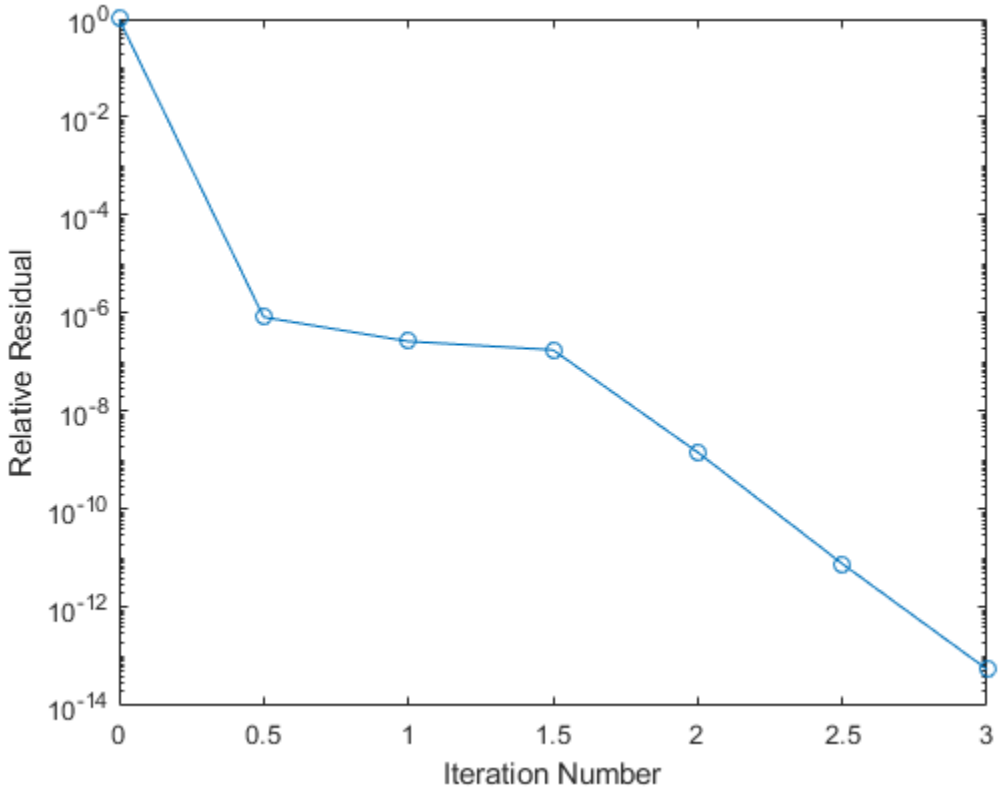
You can try again with a reduced drop tolerance, as indicated by the error message.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
[x1,fl1,rr1,it1,rv1] = bicgstab(A,b,tol,maxit,L,U);
```

`fl1` is 0 because `bicgstab` drives the relative residual to `5.9829e-014` (the value of `rr1`). The relative residual is less than the prescribed tolerance of `1e-12` at the third iteration (the value of `it1`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. The output `rv1(1)` is `norm(b)` and the output `rv1(7)` is `norm(b-A*x2)` since `bicgstab` uses half iterations.

You can follow the progress of `bicgstab` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0).

```
semilogy(0:0.5:it1,rv1/norm(b),'-o');
xlabel('Iteration Number');
ylabel('Relative Residual');
```



References

[1] Barrett, R., M. Berry, T.F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] van der Vorst, H.A., "BI-CGSTAB: A fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, March 1992, Vol. 13, No. 2, pp. 631–644.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When input `A` is a sparse matrix:
  - Only one sparse matrix preconditioner `M` is supported.
  - If you use two preconditioners, `M1` and `M2`, then both preconditioners must be functions.
  - The output parameters `iter` and `resvec` record only whole iterations.
  - `bigcstab` does not detect stagnation (Flag 3). Instead, it reports failure to converge (Flag 1).

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- If `M1` is a function, then it is applied independently to each row.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bicg` | `cgs` | `gmres` | `ilu` | `lsqr` | `minres` | `mldivide` | `pcg` | `qmr` | `symmlq`

Topics

"Create Function Handle"

Introduced before R2006a

bicgstabl

Biconjugate gradients stabilized (l) method

Syntax

```
x = bicgstabl(A,b)
x = bicgstabl(afun,b)
x = bicgstabl(A,b,tol)
x = bicgstabl(A,b,tol,maxit)
x = bicgstabl(A,b,tol,maxit,M)
x = bicgstabl(A,b,tol,maxit,M1,M2)
x = bicgstabl(A,b,tol,maxit,M1,M2,x0)
[x,flag] = bicgstabl(A,b,...)
[x,flag,relres] = bicgstabl(A,b,...)
[x,flag,relres,iter] = bicgstabl(A,b,...)
[x,flag,relres,iter,resvec] = bicgstabl(A,b,...)
```

Description

`x = bicgstabl(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and the right-hand side column vector `b` must have length `n`.

`x = bicgstabl(afun,b)` accepts a function handle `afun` instead of the matrix `A`. `afun(x)` accepts a vector input `x` and returns the matrix-vector product `A*x`. In all of the following syntaxes, you can replace `A` by `afun`.

`x = bicgstabl(A,b,tol)` specifies the tolerance of the method. If `tol` is [] then `bicgstabl` uses the default, 1e-6.

`x = bicgstabl(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is [] then `bicgstabl` uses the default, `min(N,20)`.

`x = bicgstabl(A,b,tol,maxit,M)` and `x = bicgstabl(A,b,tol,maxit,M1,M2)` use preconditioner `M` or `M=M1*M2` and effectively solve the system `A*inv(M)*x = b` for x. If `M` is [] then a preconditioner is not applied. `M` may be a function handle returning `M\x`.

`x = bicgstabl(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is [] then `bicgstabl` uses the default, an all zero vector.

`[x,flag] = bicgstabl(A,b,...)` also returns a convergence `flag`:

Flag	Convergence
0	`bicgstabl` converged to the desired tolerance `tol` within `maxit` iterations.
1	`bicgstabl` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	`bicgstabl` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `bicgstabl` became too small or too large to continue computing.

`[x,flag,relres] = bicgstabl(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = bicgstabl(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`. `iter` can be `k/4` where `k` is some integer, indicating convergence at a given quarter iteration.

`[x,flag,relres,iter,resvec] = bicgstabl(A,b,...)` also returns a vector of the residual norms at each quarter iteration, including `norm(b-A*x0)`.

Examples

Using bicgstabl with Inputs or with a Function

You can pass inputs directly to `bicgstabl`:

```
n = 21;
A = gallery('wilk',n);
b = sum(A,2);
tol = 1e-12;
maxit = 15;
M = diag([10:-1:1 1 1:10]);
x = bicgstabl(A,b,tol,maxit,M);
```

You can also use a matrix-vector product function:

```
function y = afun(x,n)
y = [0; x(1:n-1)] + [((n-1)/2:-1:0)';
(1:(n-1)/2)'].*x+[x(2:n); 0];
```

and a preconditioner backsolve function:

```
function y = mfun(r,n)
y = r ./ [((n-1)/2:-1:1)';
1;
(1:(n-1)/2)'];
```

as inputs to `bicgstabl`:

```
x1 = bicgstabl(@(x)afun(x,n),b,tol,maxit,@(x)mfun(x,n));
```

Using bicgstabl with a Preconditioner

This example demonstrates the use of a preconditioner.

Load `west0479`, a real 479-by-479 nonsymmetric sparse matrix.

```
load west0479;
A = west0479;
```

Define `b` so that the true solution is a vector of all ones.

```
b = full(sum(A,2));
```

Set the tolerance and maximum number of iterations.

```
tol = 1e-12;
maxit = 20;
```

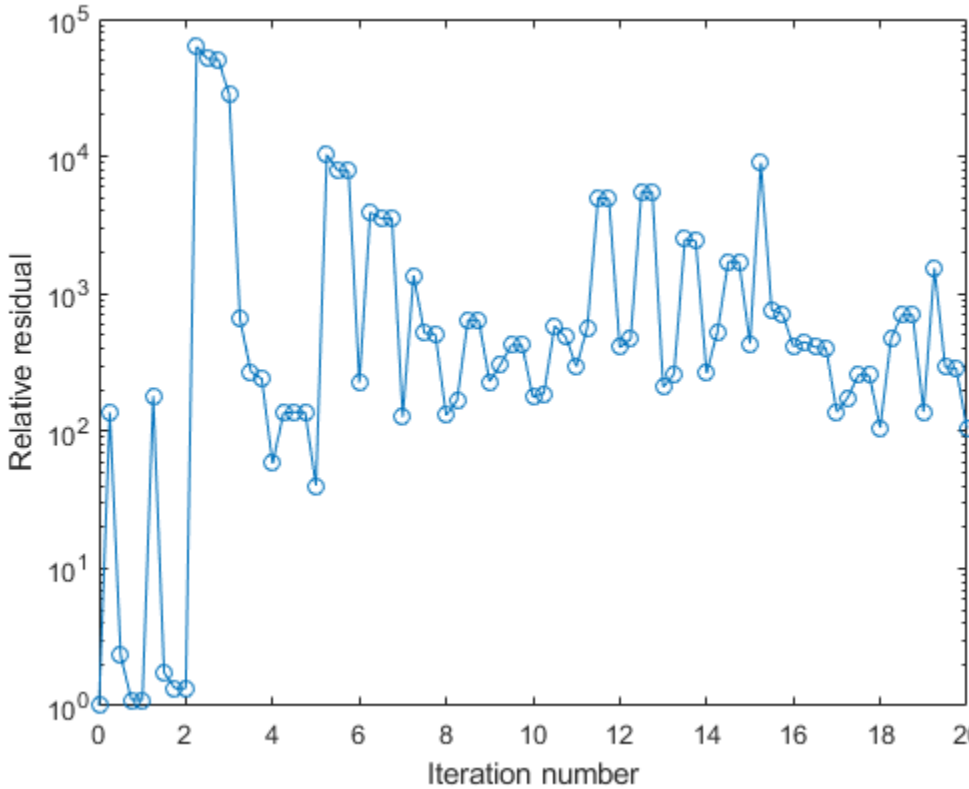
Use `bicgstabl` to find a solution at the requested tolerance and number of iterations.

```
[x0,fl0,rr0,it0,rv0] = bicgstabl(A,b,tol,maxit);
```

`fl0` is 1 because `bicgstabl` does not converge to the requested tolerance `1e-12` within the requested 20 iterations. In fact, the behavior of `bicgstabl` is so poor that the initial guess (`x0 = zeros(size(A,2),1)`) is the best solution and is returned as indicated by `it0 = 0`. MATLAB® stores the residual history in `rv0`.

Plot the behavior of `bicgstabl`.

```
semilogy(0:0.25:maxit,rv0/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



The plot shows that the solution does not converge. You can use a preconditioner to improve the outcome.

Create a preconditioner with `ilu`, since `A` is nonsymmetric.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-5));
```

```
Error using ilu
There is a pivot equal to zero.  Consider decreasing the
drop tolerance or consider using the 'udiag' option.
```

MATLAB cannot construct the incomplete LU as it would result in a singular factor, which is useless as a preconditioner.

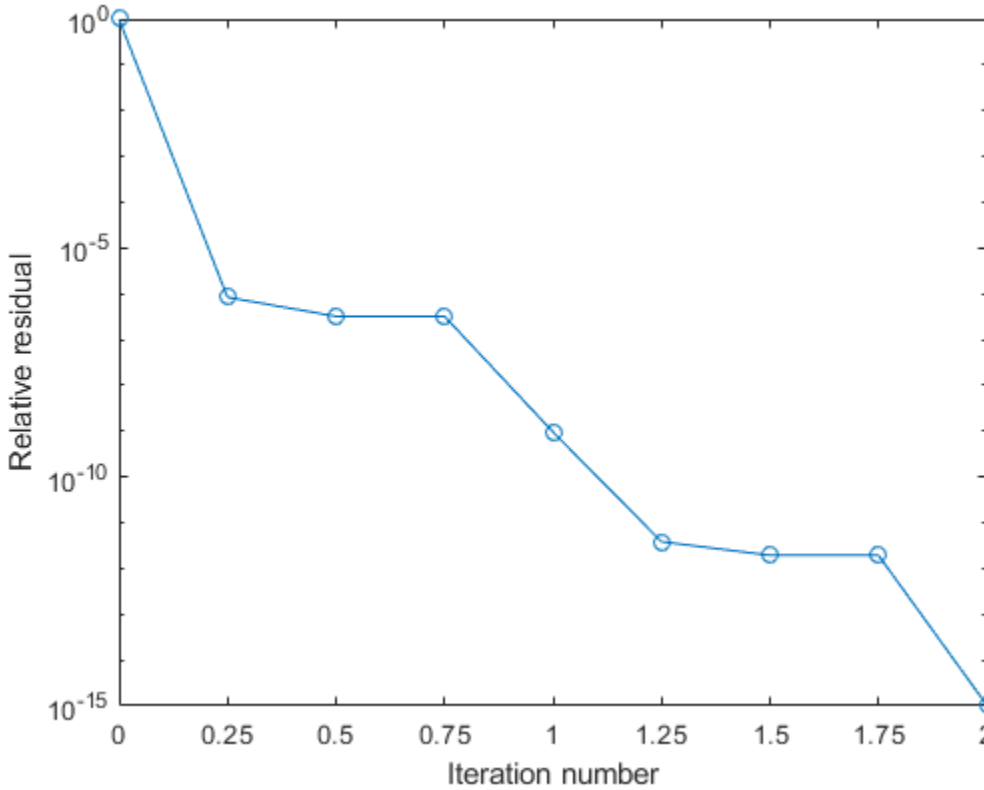
You can try again with a reduced drop tolerance, as indicated by the error message.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
[x1,fl1,rr1,it1,rv1] = bicgstabl(A,b,tol,maxit,L,U);
```

`fl1` is 0 because `bicgstabl` drives the relative residual to `1.0257e-015` (the value of `rr1`). The relative residual is less than the prescribed tolerance of `1e-12` at the sixth iteration (the value of `it1`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. The output `rv1(1)` is `norm(b)`, and the output `rv1(9)` is `norm(b-A*x2)` since `bicgstabl` uses quarter iterations.

You can follow the progress of `bicgstabl` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0).

```
semilogy(0:0.25:it1,rv1/norm(b),'-o');

h = gca;
h.XTick = 0:0.25:it1;

xlabel('Iteration number');
ylabel('Relative residual');
```



Extended Capabilities

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- If `M1` is a function, then it is applied independently to each row.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bicg` | `bicgstab` | `cgs` | `gmres` | `ilu` | `lsqr` | `minres` | `mldivide` | `pcg` | `qmr` | `symmlq`

Topics

“Create Function Handle”

bin2dec

Convert text representation of binary number to decimal number

Syntax

```
D = bin2dec(binStr)
```

Description

`D = bin2dec(binStr)` converts `binStr` to a decimal number and returns it. The input argument `binStr` represents digits of a binary number using the characters `'0'` and `'1'`.

Examples

Convert Text Representing Binary Value

Convert a character vector that represents a binary value to a decimal number.

```
binStr = '10111';
D = bin2dec(binStr)
```

```
D = 23
```

Starting in R2019b, you can write values in binary format directly without using `bin2dec`. Use the `0b` prefix and do not use quotation marks. MATLAB® stores the value as an integer, not as text.

```
D = 0b10111
```

```
D = uint8
    23
```

Return Numeric Array

Create a string array that represents multiple binary values.

```
binStr = ["1111111111" "1111010" "1110"]
```

```
binStr = 1x3 string array
    "1111111111"    "1111010"    "1110"
```

Convert the binary values and return a numeric array.

```
D = bin2dec(binStr)
```

```
D = 1×3

        1023         122          14
```

Starting in R2019b, it is recommended that you create a numeric array using binary literals instead of converting text with `bin2dec`.

```
D = [0b1111111111 0b1111010 0b1110]
```

```
D = 1x3 uint16 row vector

   1023    122     14
```

Input Arguments

`binStr` — Text representing binary numbers
character array | cell array of character vectors | string array

Text representing binary numbers, specified as a character array, cell array of character vectors, or string array.

- If `binStr` is a character array with multiple rows or a cell array of character vectors, then the output is a numeric column vector.
- If `binStr` is a string array, then the output is a numeric array that has the same dimensions.

A binary number represented by `binStr` can have no more than 53 digits. Also, `binStr` cannot represent a negative number.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Input text must be specified as a character array or a string scalar. Cell arrays are not supported.
- When the input is empty, the answer does not match the answer in MATLAB.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Input text must be specified as a character array. Cell arrays are not supported.
- When the input is empty, the answer does not match the answer in MATLAB.

See Also

`base2dec` | `dec2bin` | `flintmax` | `hex2dec`

Topics

"Hexadecimal and Binary Values"

Introduced before R2006a

binscatter

Binned scatter plot

Syntax

```
binscatter(x,y)
binscatter(x,y,N)
binscatter( ___ ,Name,Value)
binscatter(ax, ___ )
h = binscatter( ___ )
```

Description

`binscatter(x,y)` displays a binned scatter plot for vectors `x` and `y`. A binned scatter plot partitions the data space into rectangular bins and displays the count of data points in each bin using different colors. When zooming into the plot, the bin sizes automatically adjust to show finer resolution.

`binscatter(x,y,N)` specifies the number of bins to use. `N` can be a scalar or a two-element vector `[Nx Ny]`. If `N` is a scalar, then `Nx` and `Ny` are both set to the scalar value. The maximum number of bins in each dimension is 250.

`binscatter( ___ ,Name,Value)` specifies property values with one or more name-value pair arguments. For example, you can specify `'ShowEmptyBins'` as `'on'` to color areas with no data points. For a full list of properties, see Binscatter Properties. Use this option with any of the input argument combinations in previous syntaxes.

`binscatter(ax, ___ )` plots into the axes specified by `ax` instead of into the current axes (`gca`). The `ax` input can precede any of the input argument combinations in previous syntaxes.

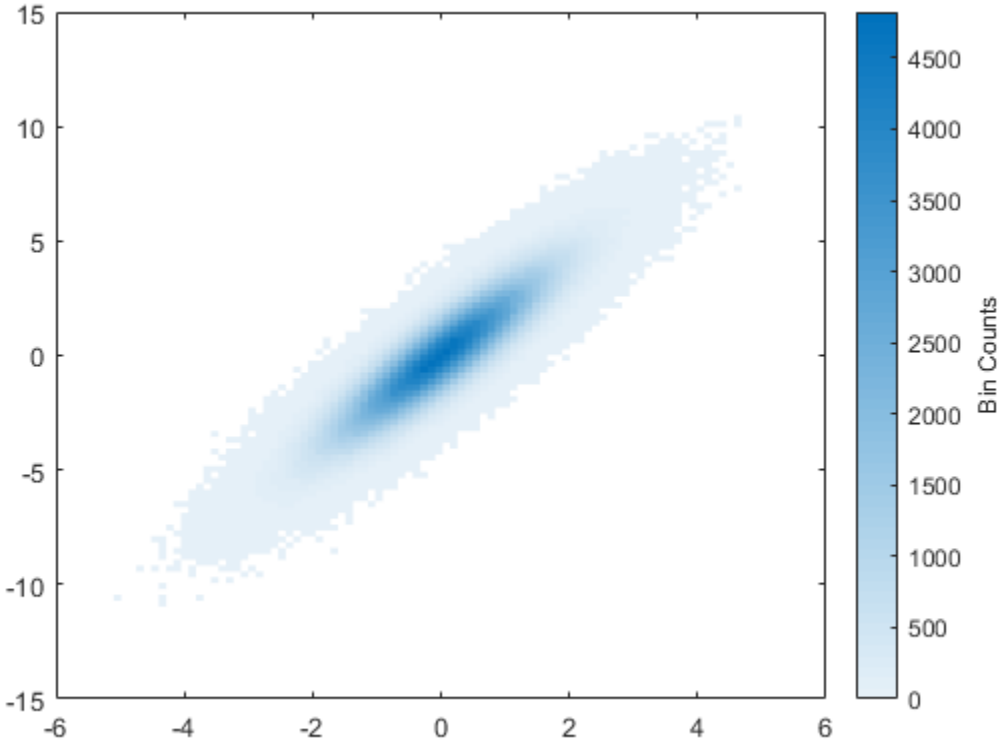
`h = binscatter( ___ )` returns a `Binscatter` object. Use this object to inspect and adjust the properties of the binned scatter plot.

Examples

Binned Scatter Plot of Vectors

Generate random numbers in both the *x* and *y* dimensions and create a binned scatter plot. The `binscatter` function automatically chooses an appropriate number of bins to cover the range of values in the data.

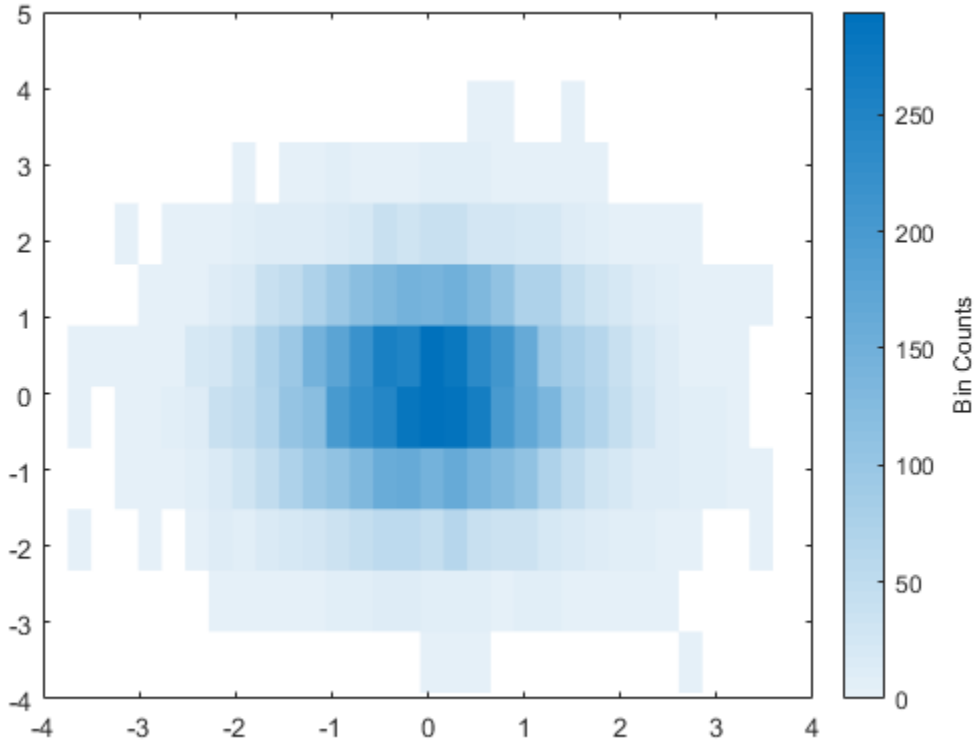
```
x = randn(1e6,1);
y = 2*x + randn(1e6,1);
binscatter(x,y)
```



Specify Number of Bins

Plot a binned scatter plot of 10,000 random numbers sorted into 30 bins in the *x* dimension and 10 bins in the *y* dimension.

```
rng default % for reproducibility
x = randn(1e4,1);
y = randn(1e4,1);
h = binscatter(x,y,[30 10]);
```



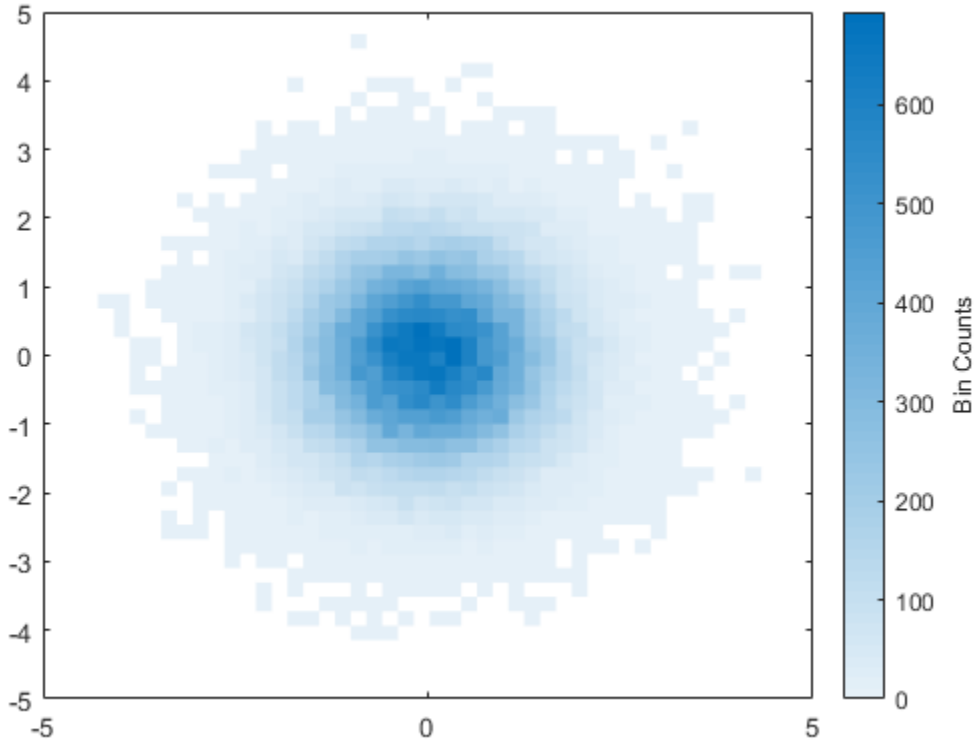
Find the bin counts. The result is a matrix with the top left element corresponding to the bin count of the bottom left bin in the plot. The *x* bins are in the rows of the matrix and the *y* bins are in the columns.

```
counts = h.Values;
```

Change Color Map of Binned Scatter Plot

Create a binned scatter plot of some random data points.

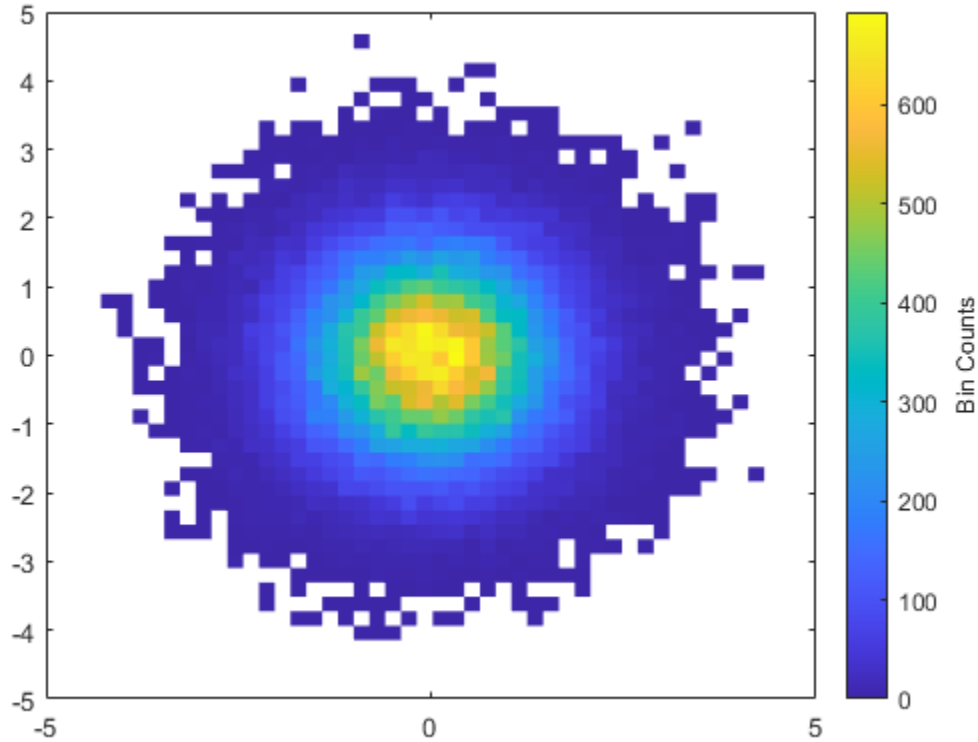
```
x = randn(1e5,1);
y = randn(1e5,1);
binscatter(x,y)
```



The default color map ranges from light colors (for small values) to dark colors (for large values). Switching to a color map that uses dark colors for small values can make it easier to spot outliers.

Use the `colormap` function to change the colors in the plot. Pass in the current axes handle using `gca`.

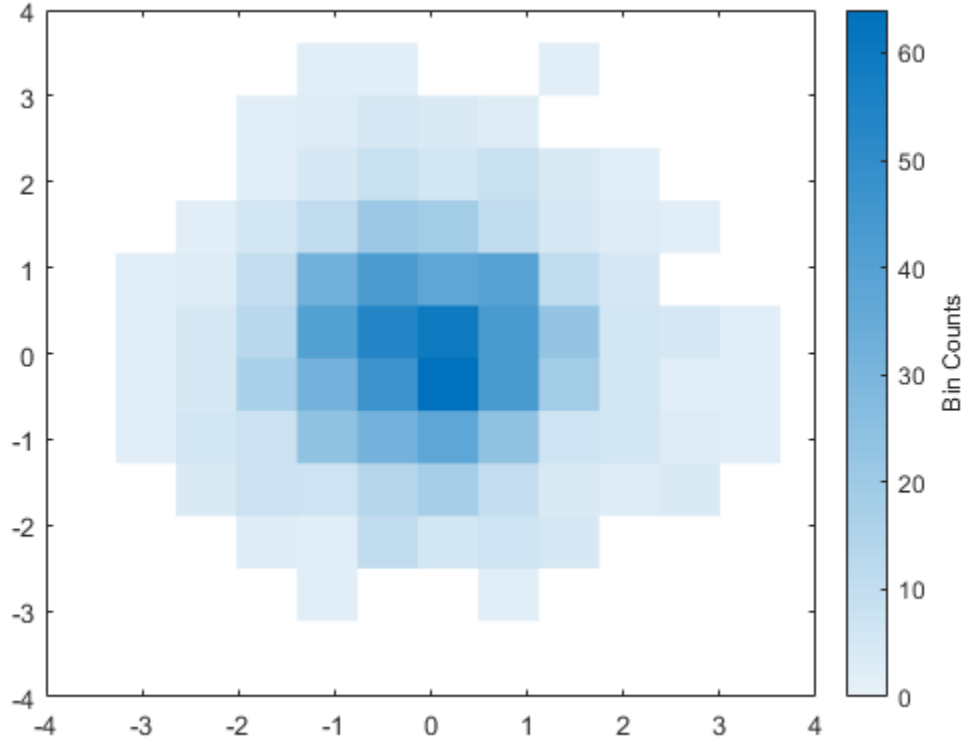
```
colormap(gca,'parula')
```



Adjust Binned Scatter Plot Properties

Generate 1,000 random numbers and create a binned scatter plot. Return the `Binscatter` object to adjust properties of the plot without recreating the entire plot.

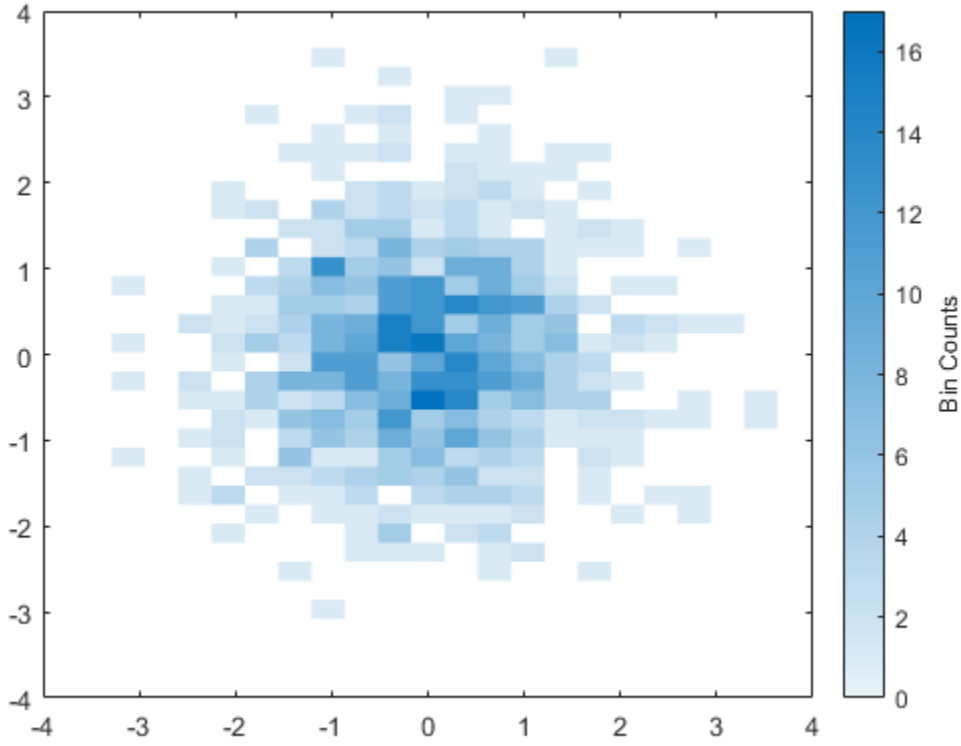
```
x = randn(1000,1);
y = randn(1000,1);
h = binscatter(x,y)
```




```
h =
  Binscatter with properties:

      NumBins: [11 11]
    XBinEdges: [1x12 double]
    YBinEdges: [1x12 double]
       Values: [11x11 double]
      XLimits: [-3.2764 3.6305]
      YLimits: [-3.1155 3.6168]
    FaceAlpha: 1

  Show all properties
```

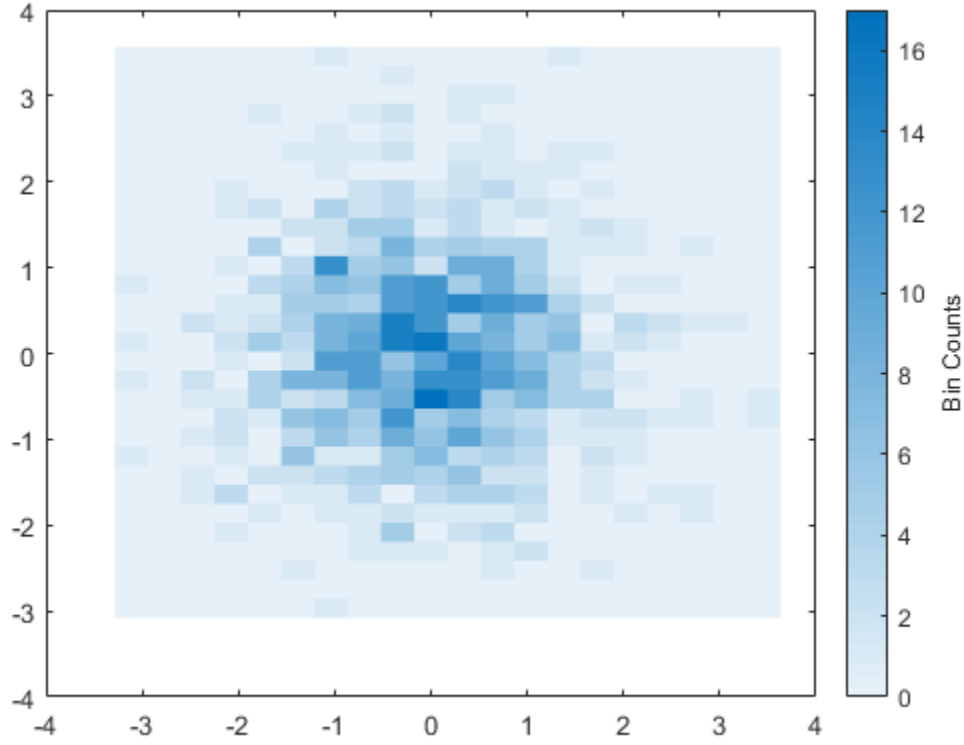
Specify exactly how many bins to use in each direction.

```
h.NumBins = [20 30];
```



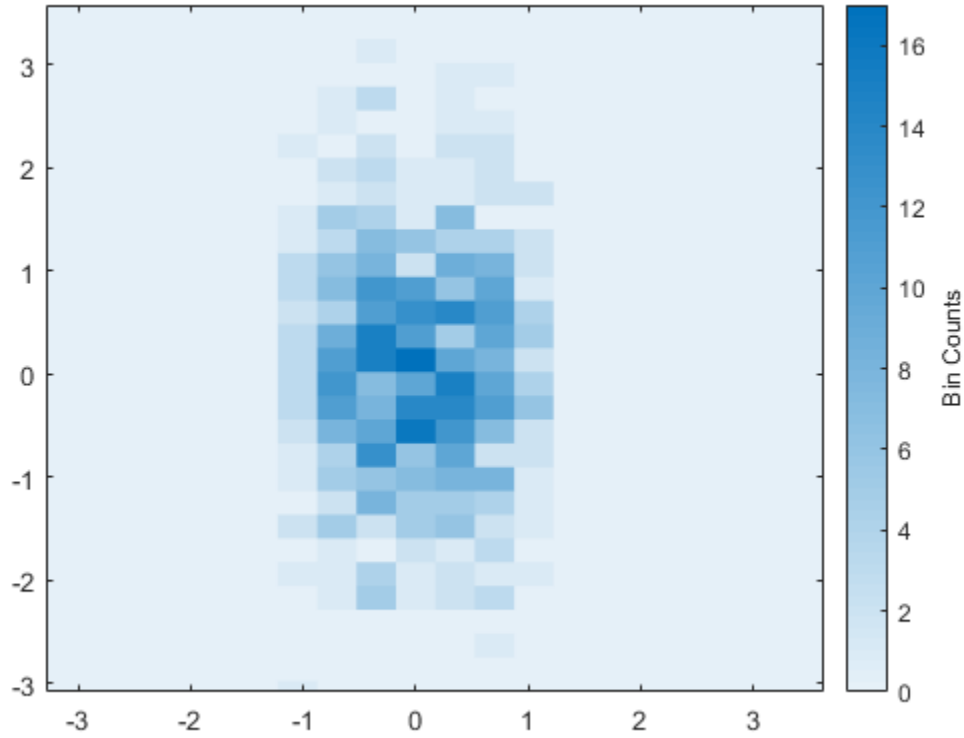
Turn on the display of empty bins in the plot.

```
h.ShowEmptyBins = 'on';
```



Specify the extent of the axes with the `XLimits` and `YLimits` properties. Then limit the bin limits in the *x* direction with a vector.

```
xlim(gca,h.XLimits);
ylim(gca,h.YLimits);
h.XLimits = [-1 1];
```



Input Arguments

x, y — Input vectors
vectors

Input vectors, specified as real vectors of the same length.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `datetime` | `duration`

N — Number of bins
scalar | two-element vector

Number of bins, specified as a scalar or two-element vector `[Nx Ny]`.

- If `N` is a two-element vector `[Nx Ny]`, then `binscatter` uses `Nx` bins in the *x* dimension and `Ny` bins in the *y* dimension.
- If `N` is a scalar, then `Nx` and `Ny` are both set to the scalar value.

`binscatter` uses `Nx` and `Ny` bins along the *x* and *y* dimensions in the initial plot, when the axes are not zoomed in. (The axes are not zoomed in when the `XLimMode` and `YLimMode` properties are both `'auto'`.) When zooming, `binscatter` adjusts the number of bins to maintain a bin size such that the visible portion of the plot is approximately divided into `Nx`-by-`Ny` bins.

The maximum number of bins in each dimension is 250. The default number of bins is computed based on the data size and standard deviation and does not exceed 100.

Example: `[10 20]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

ax — Target axes
`gca` (default) | `Axes` object

Target axes, specified as an `Axes` object. If you do not specify the axes, then `binscatter` uses the current axes (`gca`).

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `binscatter(x,y,'ShowEmptyBins','on')` turns on the display of empty bins in the plot.

The properties listed here are only a subset. For a complete list, see Binscatter Properties.

`XLimits` — Data limits in *x*-dimension
vector

Data limits in *x*-dimension, specified as a two-element vector `[Xmin Xmax]`.

`binscatter` only displays data points that fall within the specified data limits inclusively, $X_{\min} \le X \le X_{\max}$.

Example: `[0 10]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `datetime` | `duration`

`YLimits` — Data limits in *y*-dimension
vector

Data limits in *y*-dimension, specified as a two-element vector `[Ymin Ymax]`.

`binscatter` only displays data points that fall within the specified data limits inclusively, $Y_{\min} \le Y \le Y_{\max}$.

Example: `[0 10]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `datetime` | `duration`

`ShowEmptyBins` — Toggle to show empty bins
`'off'` (default) | `'on'`

Toggle to show empty bins, specified as either `'off'` or `'on'`. Specify `'on'` to color tiles in the plot that fall within the bin limits, but have no data points.

Output Arguments

`h` — `Binscatter` object
`Binscatter` object

`Binscatter` object. Use this object to inspect and adjust properties of the plot. For a full listing of properties, see Binscatter Properties.

Tips

- Change the `ColorScale` property of the axes to `'log'` to produce better bin coloring when a few bins dominate the plot.

```
ax = gca;
ax.ColorScale = 'log';
```

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

With tall arrays, the `binscatter` function plots in iterations, progressively adding to the plot as more data is read. During the updates, a progress indicator shows the proportion of data that has been plotted. Zooming and panning is supported during the update process, before the plot is complete. To stop the update process, press the pause button in the progress indicator.

For more information, see “Tall Arrays”.

See Also

`histcounts` | `histcounts2` | `histogram` | `histogram2` | `scatter`

Topics

“Visualization of Tall Arrays”

Introduced in R2017b

Binscatter Properties

Binscatter appearance and behavior

Description

`Binscatter` properties control the appearance and behavior of binned scatter plots. By changing property values, you can modify aspects of the display. Use dot notation to refer to a particular object and property:

```
h = binscatter(randn(1,100),randn(1,100));
N = h.NumBins
h.NumBins = [3 3]
```

Properties

Bins

`NumBins` — Number of bins

scalar | two-element vector

Number of bins, specified as a scalar or two-element vector `[Nx Ny]`.

- If `NumBins` is specified as a two-element vector `[Nx Ny]`, then `binscatter` uses `Nx` bins in the *x* dimension and `Ny` bins in the *y* dimension.
- If `NumBins` is specified as a scalar, then `Nx` and `Ny` are both set to the scalar value.

`binscatter` uses `Nx` and `Ny` bins along the *x* and *y* dimensions in the initial plot, when the axes are not zoomed in. (The axes are not zoomed in when the `XLimMode` and `YLimMode` properties are both `'auto'`.) When zooming, `binscatter` adjusts the number of bins to maintain a bin size such that the visible portion of the plot is approximately divided into `Nx`-by-`Ny` bins.

The maximum number of bins in each dimension is 250. The default number of bins is computed based on the data size and standard deviation and does not exceed 100.

Example: `[10 20]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`NumBinsMode` — Selection mode for number of bins
`'auto'` (default) | `'manual'`

Selection mode for number of bins, specified as either `'auto'` or `'manual'`. With the default value of `'auto'`, the number of bins is computed from the data according to Scott's rule, `[3.5*std(X(:))*numel(X)^(-1/4), 3.5*std(Y(:))*numel(Y)^(-1/4)]`.

If you specify the number of bins, then the value of `'NumBinsMode'` is set to `'manual'`.

`ShowEmptyBins` — Toggle to show empty bins
`'off'` (default) | `'on'`

Toggle to show empty bins, specified as either `'off'` or `'on'`. Specify `'on'` to color tiles in the plot that fall within the bin limits, but have no data points.

`XBinEdges` — Bin edges in *x*-dimension
vector

This property is read-only.

Bin edges in *x*-dimension, returned as a vector.

Data Types: `single` | `double` | `datetime` | `duration`

`YBinEdges` — Bin edges in *y* dimension
vector

This property is read-only.

Bin edges in *y* dimension, returned as a vector.

Data Types: `single` | `double` | `datetime` | `duration`

`XLimits` — Data limits in *x*-dimension
vector

Data limits in *x*-dimension, specified as a two-element vector `[Xmin Xmax]`.

`binscatter` only displays data points that fall within the specified data limits inclusively, $X_{\min} \le X \le X_{\max}$.

Example: `[0 10]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `datetime` | `duration`

`XLimitsMode` — Selection mode for data limits in *x*-dimension
`'auto'` (default) | `'manual'`

Selection mode for data limits in *x*-dimension, specified as `'auto'` or `'manual'`. The default value is `'auto'`, so that the bin limits automatically adjust to the data along the *x*-axis.

If you explicitly specify `XLimits`, then `XLimitsMode` is automatically set to `'manual'`. In that case, specify `XLimitsMode` as `'auto'` to rescale the bin limits to the data.

`YLimits` — Data limits in *y*-dimension
vector

Data limits in *y*-dimension, specified as a two-element vector `[Ymin Ymax]`.

`binscatter` only displays data points that fall within the specified data limits inclusively, $Y_{\min} \le Y \le Y_{\max}$.

Example: `[0 10]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `datetime` | `duration`

`YLimitsMode` — Selection mode for data limits in *y*-dimension
`'auto'` (default) | `'manual'`

Selection mode for data limits in *y*-dimension, specified as `'auto'` or `'manual'`. The default value is `'auto'`, so that the bin limits automatically adjust to the data along the *y*-axis.

If you explicitly specify `YLimits`, then `YLimitsMode` is automatically set to `'manual'`. In that case, specify `YLimitsMode` as `'auto'` to rescale the bin limits to the data.

Data

`XData` — *x* coordinates of data
vector

x coordinates of data, specified as a vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `datetime` | `duration`

`YData` — *y* coordinates of data
vector

y coordinates of data, specified as a vector.

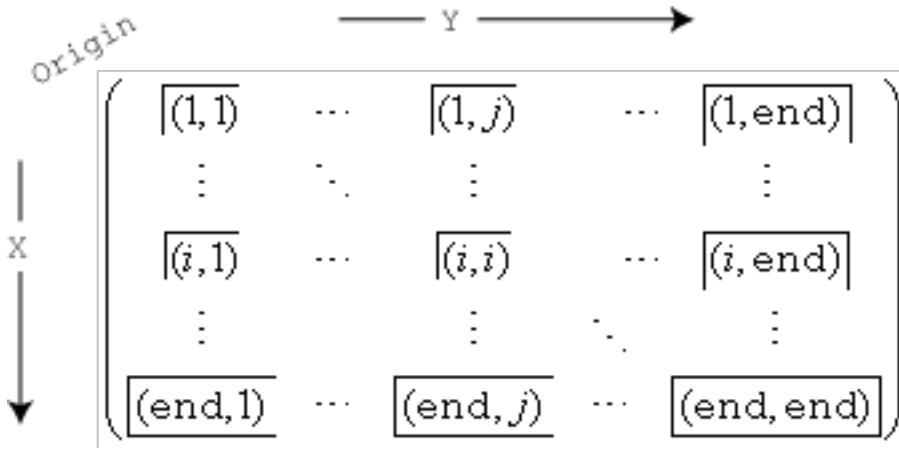
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `datetime` | `duration`

`Values` — Bin values
double matrix vector

This property is read-only.

Bin values, returned as a double matrix. The `(i,j)`th entry in `Values` specifies the bin count for the bin whose *x* edges are `[XBinEdges(i), XBinEdges(i+1)]` and whose *y* edges are `[YBinEdges(j), YBinEdges(j+1)]`.

The bin inclusion scheme for the different numbered bins in `Values`, as well as their relative orientation to the *x*-axis and *y*-axis, is



For example, the `(1,1)` bin includes values that fall on the first edge in each dimension. The last bin in the bottom right includes values that fall on any of its edges.

Transparency

FaceAlpha — Transparency of tiles

1 (default) | scalar value between 0 and 1 inclusive

Transparency of tiles, specified as a scalar value between 0 and 1 inclusive. binscatter uses the same transparency for all the tiles. A value of 1 means fully opaque and 0 means completely transparent (invisible).

Example: binscatter(X,Y,'FaceAlpha',0.5) creates a binned scatter plot with semitransparent bins.

Legend

DisplayName — Text used by legend

variable name of Data or '' (default) | character vector

Text used by the legend, specified as a character vector. The text appears next to an icon of the binscatter.

Example: 'Text Description'

For multiline text, create the character vector using sprintf with the new line character \n.

Example: sprintf('line one\nline two')

Alternatively, you can specify the legend text using the legend function.

- If you specify the text as an input argument to the legend function, then the legend uses the specified text and sets the DisplayName property to the same value.
- If you do not specify the text as an input argument to the legend function, then the legend uses the text in the DisplayName property. The default value of DisplayName is one of these values.
  - For numeric inputs, DisplayName is a character vector representing the variable name of the input data used to construct the histogram. If the input data does not have a variable name, then DisplayName is empty, ''.
  - For categorical array inputs, DisplayName is empty, ''.

If the DisplayName property does not contain any text, then the legend generates a character vector. The character vector has the form 'dataN', where N is the number assigned to the binscatter object based on its location in the list of legend entries.

If you edit interactively the character vector in an existing legend, then MATLAB updates the `DisplayName` property to the edited character vector.

`Annotation` — Control for including or excluding object from legend
`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility
`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`DataTipTemplate` — Data tip content
`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see "Callback Definition".

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you

do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks only when visible. The `Visible` property must be set to `'on'`. The `HitTest` property determines if the `Binscatter` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `Binscatter` object passes the click to the object behind it in the current view of the figure window. The `HitTest` property of the `Binscatter` object has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `Binscatter` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `Binscatter` object that has one of these:
  - `HitTest` property set to `'on'`

- `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

---

Note The `PickableParts` property determines if the `Binscatter` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent

`Axes` object

Parent, specified as an `Axes` object.

`Children` — Children

empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the chart object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'binscatter'`

This property is read-only.

Type of graphics object, returned as `'binscatter'`. Use this property to find all objects of a given type within a plotting hierarchy, such as searching for the type using `findobj`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`binscatter`

Introduced in R2017b

binary

Set FTP transfer mode to binary

Syntax

```
binary(ftpobj)
```

Description

`binary(ftpobj)` sets the FTP transfer mode for the server associated with `ftpobj` to binary, which is the default for FTP objects. If you previously called the `ascii` function, then call `binary` before transferring a nontext file, such as an executable file or zip archive.

Examples

Set Transfer Mode to Binary

Download a text file from an FTP server, using the ASCII transfer mode. After you download the file, set the transfer mode back to binary.

First, connect to the National Centers for Environmental Information (NCEI) FTP server.

```
ftpobj = ftp('ftp.ngdc.noaa.gov')
```

```
ftpobj = 

  FTP Object
     host: ftp.ngdc.noaa.gov
     user: anonymous
      dir: /
     mode: binary
```

The default transfer mode is binary. When you download a text file, you can use ASCII mode instead so that newline characters are converted correctly.

Set the transfer mode to ASCII. To show the current transfer mode, display the FTP object.

```
ascii(ftpobj)
ftpobj
```

```
ftpobj =

  FTP Object
     host: ftp.ngdc.noaa.gov
     user: anonymous
      dir: /
     mode: ascii
```

Download a text file.

```
mget(ftpobj,'README.txt');
```

Use the ASCII transfer mode only for text files. To download nontext files, such as images or ZIP archives, use the binary transfer mode.

Set the transfer mode back to binary. To check that the mode is now binary, display the FTP object again.

```
binary(ftpobj)
ftpobj
```

```
ftpobj =

  FTP Object
     host: ftp.ngdc.noaa.gov
     user: anonymous
      dir: /
     mode: binary
```

FTP service courtesy of the National Centers for Environmental Information (NCEI). See the NCEI Privacy Policy, Disclaimer, and Copyright for NCEI terms of service.

Input Arguments

`ftpobj` — Connection to FTP server

FTP object

Connection to an FTP server, specified as an FTP object.

See Also

`ascii` | `ftp`

Introduced before R2006a

bitand

Bit-wise AND

Syntax

```
C = bitand(A,B)
C = bitand(A,B,assumedtype)

objout = bitand(netobj1,netobj2)
```

Description

`C = bitand(A,B)` returns the bit-wise AND of `A` and `B`.

`C = bitand(A,B,assumedtype)` assumes that `A` and `B` are of `assumedtype`.

`objout = bitand(netobj1,netobj2)` returns the bit-wise AND of the .NET enumeration objects `netobj1` and `netobj2`.

Examples

Truth Table

Create a truth table for the logical AND operation.

```
A = uint8([0 1; 0 1]);
B = uint8([0 0; 1 1]);
TTable = bitand(A, B)
```

```
TTable = 2x2 uint8 matrix

   0   0
   0   1
```

`bitand` returns 1 only if both bit-wise inputs are 1.

Negative Values

MATLAB® encodes negative integers using two's complement. For example, to find the two's complement representation of -5, you take the bit pattern of the positive version of the number (`00000101`), swap each bit (`11111010`), and then add 1 to the result (`11111011`).

Therefore, the bit-wise AND of -5 (`11111011`) and 6 (`00000110`) is 2 (`00000010`).

```
a = -5;
bitget(a,8:-1:1,'int8')
```

```
ans = 1×8

     1     1     1     1     1     0     1     1
```

```
b = 6;
bitget(b,8:-1:1,'int8')
```

```
ans = 1×8

     0     0     0     0     0     1     1     0
```

```
c = bitand(a,b,'int8')
```

```
c = 2
```

```
bitget(c,8:-1:1,'int8')
```

```
ans = 1×8

     0     0     0     0     0     0     1     0
```

Input Arguments

A,B — Input values

scalars | vectors | matrices | multidimensional arrays

Input values, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs A and B must either be the same size or have sizes that are compatible (for example, A is an M-by-N matrix and B is a scalar or 1-by-N row vector). For more information, see "Compatible Array Sizes for Basic Operations". A and B also must be the same data type unless one is a scalar double.

- If A and B are double arrays, and `assumedtype` is not specified, then MATLAB treats A and B as unsigned 64-bit integers.
- If `assumedtype` is specified, then all elements in A and B must have integer values within the range of `assumedtype`.

Data Types: `double` | `logical` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

assumedtype — Assumed data type of integ1 and integ2

`'uint64'` | `'uint32'` | `'uint16'` | `'uint8'` | `'int64'` | `'int32'` | `'int16'` | `'int8'`

Assumed data type of A and B, specified as `'uint64'`, `'uint32'`, `'uint16'`, `'uint8'`, `'int64'`, `'int32'`, `'int16'`, or `'int8'`.

- If A and B are double arrays, then `assumedtype` can specify any valid integer type, but defaults to `'uint64'`.
- If A and B are integer type arrays, then `assumedtype` must specify that same integer type.

Data Types: `char` | `string`

netobj1, netobj2 — Input values

.NET enumeration objects

Input values, specified as .NET enumeration objects. You must be running a version of Windows to use .NET enumeration objects as input arguments.

`bitand` is an instance method for MATLAB enumeration objects created from a .NET enumeration.

Output Arguments

`C` — Bit-wise AND result
array

Bit-wise AND result, returned as an array. `C` is the same data type as `A` and `B`.

- If either `A` or `B` is a scalar double, and the other is an integer type, then `C` is the integer type.

`objout` — Bit-wise AND result
.NET enumeration object

Bit-wise AND result, returned as a .NET enumeration objects.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Both inputs can be unsigned integer arrays, or one input can be an unsigned integer array and the other input can be a scalar double.
- 64-bit integers are not supported.
- The `assumedtype` argument is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `assumedtype` argument is not supported.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bitcmp` | `bitget` | `bitnot` | `bitor` | `bitset` | `bitshift` | `bitxor` | `intmax`

Topics

“Creating .NET Enumeration Bit Flags”

Introduced before R2006a

bitcmp

Bit-wise complement

Syntax

```
cmp = bitcmp(A)
cmp = bitcmp(A,assumedtype)
```

Description

`cmp = bitcmp(A)` returns the bit-wise complement of `A`.

`cmp = bitcmp(A,assumedtype)` assumes that `A` is of `assumedtype`.

Examples

Complement of a Negative Integer

```
A = int8(-11);
cmp = bitcmp(A)
```

```
cmp = int8
    10
```

You can see the complement operation when the numbers are shown in binary.

```
original = bitget(A,8:-1:1)
```

```
original = 1x8 int8 row vector

   1   1   1   1   0   1   0   1
```

```
complement = bitget(bitcmp(A),8:-1:1)
```

```
complement = 1x8 int8 row vector
```

```
   0   0   0   0   1   0   1   0
```

Complement of Unsigned Integers

```
cmp = bitcmp(64,'uint8')
```

```
cmp = 191
```

```
maxint = intmax('uint8') - 64
```

```
maxint = uint8
    191
```

The complement of an unsigned integer is equal to itself subtracted from the maximum integer of its data type.

Input Arguments

A — Input value
signed integer array | unsigned integer array | double array

Input value, specified as a signed integer array, unsigned integer array, or double array.

- If A is a double array, and `assumedtype` is not specified, then MATLAB treats A as an unsigned 64-bit integer.
- If `assumedtype` is specified, then all elements in A must have integer values within the range of `assumedtype`.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`assumedtype` — Assumed data type of A
`'uint64'` | `'uint32'` | `'uint16'` | `'uint8'` | `'int64'` | `'int32'` | `'int16'` | `'int8'`

Assumed data type of A, specified as `'uint64'`, `'uint32'`, `'uint16'`, `'uint8'`, `'int64'`, `'int32'`, `'int16'`, or `'int8'`.

- If `A` is a double array, then `assumedtype` can specify any valid integer type, but defaults to `'uint64'`.
- If `A` is an integer type array, then `assumedtype` must specify that same integer type.

Data Types: `char` | `string`

Output Arguments

`cmp` — Bit-wise complement
signed integer array | unsigned integer array | double array

Bit-wise complement, returned as a signed integer array, unsigned integer array, or double array. `cmp` is the same size and type as `A`.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The input argument `A` must be a `gpuArray` of unsigned integers.
- 64-bit integers are not supported.
- The `assumedtype` argument is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`bitand` | `bitget` | `bitor` | `bitset` | `bitshift` | `bitxor` | `intmax`

Introduced before R2006a

bitget

Get bit at specified position

Syntax

```
b = bitget(A,bit)
b = bitget(A,bit,assumedtype)
```

Description

`b = bitget(A,bit)` returns the bit value at position `bit` in integer array `A`.

`b = bitget(A,bit,assumedtype)` assumes that `A` is of `assumedtype`.

Examples

Maximum Integer

Find the difference in the binary representation between the maximum integer of signed and unsigned integers.

```
a1 = intmax('int8');
a2 = intmax('uint8');
b1 = bitget(a1,8:-1:1)
```

```
b1 = 1x8 int8 row vector

   0   1   1   1   1   1   1   1
```

```
b2 = bitget(a2,8:-1:1)
```

```
b2 = 1x8 uint8 row vector

   1   1   1   1   1   1   1   1
```

The signed integers require a bit to accommodate negative integers.

Negative Numbers Using Two's Complement

Find the 8-bit representation of a negative number.

```
A = -29;
b = bitget(A,8:-1:1,'int8')
```

```
b = 1×8

     1     1     1     0     0     0     1     1
```

Input Arguments

A — Input values
array

Input values, specified as an array. `A` can be a scalar or an array of the same size as `bit`.

- If `A` is a double array, and `assumedtype` is not specified, then MATLAB treats `A` as an unsigned 64-bit integer.
- If `assumedtype` is specified, then all elements in `A` must have integer values within the range of `assumedtype`.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`bit` — Bit position
integer | integer array

Bit position, specified as an integer or integer array. `bit` can be a scalar or an array of the same size as `A`. `bit` must be between 1 (the least-significant bit) and the number of bits in the integer class of `A`.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

assumedtype — Assumed data type of A
`'uint64'` | `'uint32'` | `'uint16'` | `'uint8'` | `'int64'` | `'int32'` | `'int16'` | `'int8'`

Assumed data type of `A`, specified as `'uint64'`, `'uint32'`, `'uint16'`, `'uint8'`, `'int64'`, `'int32'`, `'int16'`, or `'int8'`.

- If `A` is a double array, then `assumedtype` can specify any valid integer type, but defaults to `'uint64'`.
- If `A` is an integer type array, then `assumedtype` must specify that same integer type.

Data Types: `char` | `string`

Output Arguments

b — Bit value at bit
array

Bit value at `bit`, returned as an array of `0`s and `1`s. `b` is the same data type as `A`.

- If `A` and `bit` are scalars, then `b` is also a scalar.
- If either `A` or `bit` is an array, then `b` is the same size as that array.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Both inputs can be unsigned integer arrays, or one input can be an unsigned integer array and the other input can be a scalar double.
- 64-bit integers are not supported.
- The `assumedtype` argument is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`bitand` | `bitcmp` | `bitor` | `bitset` | `bitshift` | `bitxor` | `intmax`

Introduced before R2006a

bitnot

.NET enumeration object bit-wise NOT instance method

Syntax

```
objout = bitnot(netobj)
```

Description

`objout = bitnot(netobj)` reverses all bits of the .NET enumeration objects `netobj`.

Input Arguments

`netobj` — Input value
.NET enumeration objects

Input value, specified as .NET enumeration object. You must be running a version of Windows to use .NET enumeration objects as input arguments.

Output Arguments

`objout` — Bit-wise NOT result
.NET enumeration object

Bit-wise NOT result, returned as a .NET enumeration object.

Limitations

- The method is an instance method for MATLAB enumeration objects created from a .NET enumeration. This method does not have an equivalent MATLAB function.

See Also

`bitand` | `bitor` | `bitxor`

Topics

"Creating .NET Enumeration Bit Flags"

Introduced in R2011a

bitor

Bit-wise OR

Syntax

```
C = bitor(A,B)
C = bitor(A,B,assumedtype)

objout = bitor(netobj1,netobj2)
```

Description

`C = bitor(A,B)` returns the bit-wise OR of `A` and `B`.

`C = bitor(A,B,assumedtype)` assumes that `A` and `B` are of `assumedtype`.

`objout = bitor(netobj1,netobj2)` returns the bit-wise OR of the .NET enumeration objects `netobj1` and `netobj2`.

Examples

Truth Table

Create a truth table for the logical OR operation.

```
A = uint8([0 1; 0 1]);
B = uint8([0 0; 1 1]);
TTable = bitor(A, B)
```

```
TTable = 2x2 uint8 matrix

   0   1
   1   1
```

`bitor` returns 1 if either bit-wise input is 1.

Negative Values

MATLAB® encodes negative integers using two's complement. For example, to find the two's complement representation of -5, you take the bit pattern of the positive version of the number (`00000101`), swap each bit (`11111010`), and then add 1 to the result (`11111011`).

Therefore, the bit-wise OR of -5 (`11111011`) and 6 (`00000110`) is -1 (`11111111`).

```
a = -5;
bitget(a,8:-1:1,'int8')
```

```
ans = 1×8

     1     1     1     1     1     0     1     1
```

```
b = 6;
bitget(b,8:-1:1,'int8')
```

```
ans = 1×8

     0     0     0     0     0     1     1     0
```

```
c = bitor(a,b,'int8')
```

```
c = -1
```

```
bitget(c,8:-1:1,'int8')
```

```
ans = 1×8

     1     1     1     1     1     1     1     1
```

Input Arguments

A,B — Input values

scalars | vectors | matrices | multidimensional arrays

Input values, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs A and B must either be the same size or have sizes that are compatible (for example, A is an M-by-N matrix and B is a scalar or 1-by-N row vector). For more information, see "Compatible Array Sizes for Basic Operations". A and B also must be the same data type unless one is a scalar double.

- If A and B are double arrays, and `assumedtype` is not specified, then MATLAB treats A and B as unsigned 64-bit integers.
- If `assumedtype` is specified, then all elements in A and B must have integer values within the range of `assumedtype`.

Data Types: `double` | `logical` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

assumedtype — Assumed data type of integ1 and integ2

`'uint64'` | `'uint32'` | `'uint16'` | `'uint8'` | `'int64'` | `'int32'` | `'int16'` | `'int8'`

Assumed data type of A and B, specified as `'uint64'`, `'uint32'`, `'uint16'`, `'uint8'`, `'int64'`, `'int32'`, `'int16'`, or `'int8'`.

- If A and B are double arrays, then `assumedtype` can specify any valid integer type, but defaults to `'uint64'`.
- If A and B are integer type arrays, then `assumedtype` must specify that same integer type.

Data Types: `char` | `string`

netobj1, netobj2 — Input values

.NET enumeration objects

Input values, specified as .NET enumeration objects. You must be running a version of Windows to use .NET enumeration objects as input arguments.

`bitor` is an instance method for MATLAB enumeration objects created from a .NET enumeration.

Output Arguments

`C` — Bit-wise OR result
array

Bit-wise OR result, returned as an array. `C` is the same data type as `A` and `B`.

- If either `A` or `B` is a scalar double, and the other is an integer type, then `C` is the integer type.

`objout` — Bit-wise OR result
.NET enumeration object

Bit-wise OR result, returned as a .NET enumeration objects.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Both inputs can be unsigned integer arrays, or one input can be an unsigned integer array and the other input can be a scalar double.
- 64-bit integers are not supported.
- The `assumedtype` argument is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `assumedtype` argument is not supported.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bitand` | `bitcmp` | `bitget` | `bitnot` | `bitset` | `bitshift` | `bitxor` | `intmax`

Topics

“Creating .NET Enumeration Bit Flags”

Introduced before R2006a

bitset

Set bit at specific location

Syntax

```
intout = bitset(A,bit)
intout = bitset(A,bit,assumedtype)

intout = bitset(A,bit,V)
intout = bitset(A,bit,V,assumedtype)
```

Description

`intout = bitset(A,bit)` returns the value of `A` with position `bit` set to 1 (on).

`intout = bitset(A,bit,assumedtype)` assumes `A` is of type `assumedtype`.

`intout = bitset(A,bit,V)` returns `A` with position `bit` set to the value of `V`.

- If `V` is zero, then the bit position `bit` is set to 0 (off).
- If `V` is nonzero, then the bit position `bit` is set to 1 (on).

`intout = bitset(A,bit,V,assumedtype)` assumes `A` is of type `assumedtype`.

Examples

Set Bits to On

Add powers of 2 onto a number.

```
A = 4;
intout = bitset(A,4:6)
```

```
intout = 1×3
```

```
    12    20    36
```

You can see that `bitset` sequentially turns on bits 4 through 6.

```
c = dec2bin(intout)
```

```
c = 3x6 char array
    '001100'
    '010100'
    '100100'
```

Out of Range of Integer Type

MATLAB® throws an error if you specify an integer outside the range of `assumedtype`. For instance, `bitset(300,5,'int8')` returns an error since the maximum value of an `int8` integer is 127.

Avoid this error by limiting your input to the range of the specified data type.

```
intout = bitset(75,5,'int8')
```

```
intout = 91
```

Set Bits to Off

Repeatedly subtract powers of 2 from a number.

```
a = intmax('uint8')
```

```
a = uint8
    255
```

```
for k = 0:7
   a = bitset(a, 8-k, 0);
   b(1,k+1) = a;
end
b
```

```
b = 1x8 uint8 row vector

   127    63    31    15     7     3     1     0
```

Set Multiple Bits

Set multiple bits to different values

```
bits = 2:6;
val = [1 0 0 1 1];
intout = bitset(0,bits,val,'int8')
```

```
intout = 1×5

     2     0     0    16    32
```

Input Arguments

A — Input values
array

Input values, specified as an array. `A`, `bit`, and `V` can each be scalars or arrays of the same size.

- If `A` is a double array, and `assumedtype` is not specified, then MATLAB treats `A` as an unsigned 64-bit integer.
- If `assumedtype` is specified, then all elements in `A` must have integer values within the range of `assumedtype`.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`bit` — Bit position
integer | integer array

Bit position, specified as an integer or integer array. `A`, `bit`, and `V` can each be scalars or arrays of the same size. The values of `bit` must be between 1 (the least significant bit) and the number of bits in the integer class of `A`.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`assumedtype` — Assumed data type of `A`
`'uint64'` | `'uint32'` | `'uint16'` | `'uint8'` | `'int64'` | `'int32'` | `'int16'` | `'int8'`

Assumed data type of `A`, specified as `'uint64'`, `'uint32'`, `'uint16'`, `'uint8'`, `'int64'`, `'int32'`, `'int16'`, or `'int8'`.

- If `A` is a double array, then `assumedtype` can specify any valid integer type, but defaults to `'uint64'`.
- If `A` is an integer type array, then `assumedtype` must specify that same integer type.

Data Types: `char` | `string`

`V` — bit value
scalar | numeric array

Bit value, specified as a scalar or a numeric array. `A`, `bit`, and `V` can each be scalars or arrays of the same size.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Output Arguments

`intout` — Adjusted integers
array

Adjusted integers, returned as an array. `intout` is the same data type as `A`.

- If `A`, `bit`, and `V` are all scalars, then `intout` is also a scalar.
- If any of `A`, `bit`, or `V` is an array, then `intout` is the same size as that array.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Both inputs can be unsigned integer arrays, or one input can be an unsigned integer array and the other input can be a scalar double.
- 64-bit integers are not supported.
- The `assumedtype` argument is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`bitand` | `bitcmp` | `bitget` | `bitor` | `bitshift` | `bitxor` | `intmax`

Introduced before R2006a

bitshift

Shift bits specified number of places

Syntax

```
intout = bitshift(A,k)
intout = bitshift(A,k,assumedtype)
```

Description

`intout = bitshift(A,k)` returns `A` shifted to the left by `k` bits, equivalent to multiplying by 2^k. Negative values of `k` correspond to shifting bits right or dividing by $2^{|k|}$ and rounding to the nearest integer towards negative infinity. Any overflow bits are truncated.

- If `A` is an array of signed integers, then `bitshift` returns the arithmetic shift results, preserving the signed bit when `k` is negative, and not preserving the signed bit when `k` is positive.
- If `k` is positive, MATLAB shifts the bits to the left and inserts `k` 0-bits on the right.
- If `k` is negative and `A` is nonnegative, then MATLAB shifts the bits to the right and inserts `|k|` 0-bits on the left.
- If `k` is negative and `A` is negative, then MATLAB shifts the bits to the right and inserts `|k|` 1-bits on the left.

`intout = bitshift(A,k,assumedtype)` assumes `A` is of type `assumedtype`.

Examples

Shifted 8-bit Integer

Repeatedly shift the bits of an unsigned 8-bit value to the left until all the nonzero bits overflow.

```
a = intmax('uint8');
s1 = 'Initial uint8 value %5d is %08s in binary\n';
s2 = 'Shifted uint8 value %5d is %08s in binary\n';
fprintf(s1,a,dec2bin(a))
```

```
Initial uint8 value   255 is 11111111 in binary
```

```
 for i = 1:8
    a = bitshift(a,1);
    fprintf(s2,a,dec2bin(a))
 end
```

```
Shifted uint8 value   254 is 11111110 in binary
Shifted uint8 value   252 is 11111100 in binary
Shifted uint8 value   248 is 11111000 in binary
Shifted uint8 value   240 is 11110000 in binary
Shifted uint8 value   224 is 11100000 in binary
Shifted uint8 value   192 is 11000000 in binary
Shifted uint8 value   128 is 10000000 in binary
Shifted uint8 value     0 is 00000000 in binary
```

Different Results for Different Integer Types

Find the shift for a number using different assumed integer types.

```
uintout = bitshift(6,5:7,'uint8')
```

```
uintout = 1×3

   192   128     0
```

```
intout = bitshift(6,5:7,'int8')
```

```
intout = 1×3

   -64  -128     0
```

Input Arguments

A — Input values

array

Input values, specified as an array. A can be a scalar or an array of the same size as k.

- If A is a double array, and assumedtype is not specified, then MATLAB treats A as an unsigned 64-bit integer.
- If assumedtype is specified, then all elements in A must have integer values within the range of assumedtype.

Data Types: double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

k — Number of switched bits

integer | integer array

Number of switched bits, specified as an integer or integer array. k can be a scalar or an array of the same size as A.

Data Types: double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

assumedtype — Assumed data type of A

'uint64' | 'uint32' | 'uint16' | 'uint8' | 'int64' | 'int32' | 'int16' | 'int8'

Assumed data type of A, specified as 'uint64', 'uint32', 'uint16', 'uint8', 'int64', 'int32', 'int16', or 'int8'.

- If A is an integer type array, then assumedtype must specify that same integer type.
- If A is a double array, then assumedtype can specify any valid integer type.

Data Types: char | string

Output Arguments

intout — Shifted values

array

Shifted values, returned as an array. intout is the same data type as A.

- If `A` and `k` are scalars, then `intout` is also a scalar.
- If either `A` or `k` is an array, then `intout` is the same size as that array.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- At least one of the inputs, `A` or `k`, must be an integer array.
- Input `A` cannot be a signed integer array.
- 64-bit integers are not supported.
- The `assumedtype` argument is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`bitand` | `bitcmp` | `bitget` | `bitor` | `bitset` | `bitxor` | `intmax`

Introduced before R2006a

bitxor

Bit-wise XOR

Syntax

```
C = bitxor(A,B)
C = bitxor(A,B,assumedtype)

objout = bitxor(netobj1,netobj2)
```

Description

`C = bitxor(A,B)` returns the bit-wise XOR of `A` and `B`.

`C = bitxor(A,B,assumedtype)` assumes that `A` and `B` are of `assumedtype`.

`objout = bitxor(netobj1,netobj2)` returns the bit-wise XOR of the .NET enumeration objects `netobj1` and `netobj2`.

Examples

Truth Table

Create a truth table for the logical XOR operation.

```
A = uint8([0 1; 0 1]);
B = uint8([0 0; 1 1]);
TTable = bitxor(A, B)
```

```
TTable = 2x2 uint8 matrix

   0   1
   1   0
```

`bitxor` returns 0 if both bit-wise inputs are equal.

Negative Values

MATLAB® encodes negative integers using two's complement. For example, to find the two's complement representation of -5, you take the bit pattern of the positive version of the number (`00000101`), swap each bit (`11111010`), and then add 1 to the result (`11111011`).

Therefore, the bit-wise XOR of -5 (`11111011`) and 6 (`00000110`) is -3 (`11111101`).

```
a = -5;
bitget(a,8:-1:1,'int8')
```

```
ans = 1×8

     1     1     1     1     1     0     1     1
```

```
b = 6;
bitget(b,8:-1:1,'int8')
```

```
ans = 1×8

     0     0     0     0     0     1     1     0
```

```
c = bitxor(a,b,'int8')
```

```
c = -3
```

```
bitget(c,8:-1:1,'int8')
```

```
ans = 1×8

     1     1     1     1     1     1     0     1
```

Input Arguments

A,B — Input values

scalars | vectors | matrices | multidimensional arrays

Input values, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs A and B must either be the same size or have sizes that are compatible (for example, A is an M-by-N matrix and B is a scalar or 1-by-N row vector). For more information, see "Compatible Array Sizes for Basic Operations". A and B also must be the same data type unless one is a scalar double.

- If A and B are double arrays, and `assumedtype` is not specified, then MATLAB treats A and B as unsigned 64-bit integers.
- If `assumedtype` is specified, then all elements in A and B must have integer values within the range of `assumedtype`.

Data Types: `double` | `logical` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

assumedtype — Assumed data type of A and B

`'uint64'` | `'uint32'` | `'uint16'` | `'uint8'` | `'int64'` | `'int32'` | `'int16'` | `'int8'`

Assumed data type of A and B, specified as `'uint64'`, `'uint32'`, `'uint16'`, `'uint8'`, `'int64'`, `'int32'`, `'int16'`, or `'int8'`.

- If A and B are double arrays, then `assumedtype` can specify any valid integer type, but defaults to `'uint64'`.
- If A and B are integer type arrays, then `assumedtype` must specify that same integer type.

Data Types: `char` | `string`

netobj1, netobj2 — Input values

.NET enumeration objects

Input values, specified as .NET enumeration objects. You must be running a version of Windows to use .NET enumeration objects as input arguments.

`bitxor` is an instance method for MATLAB enumeration objects created from a .NET enumeration.

Output Arguments

`C` — Bit-wise XOR result
array

Bit-wise XOR result, returned as an array. `C` is the same data type as `A` and `B`.

- If either `A` or `B` is a scalar double, and the other is an integer type, then `C` is the integer type.

`objout` — Bit-wise XOR result
.NET enumeration object

Bit-wise XOR result, returned as a .NET enumeration objects.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Both inputs can be unsigned integer arrays, or one input can be an unsigned integer array and the other input can be a scalar double.
- 64-bit integers are not supported.
- The `assumedtype` argument is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `assumedtype` argument is not supported.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bitand` | `bitcmp` | `bitget` | `bitnot` | `bitor` | `bitset` | `bitshift` | `intmax`

Topics

“Creating .NET Enumeration Bit Flags”

Introduced before R2006a

blanks

Create character array of blanks

Syntax

```
chr = blanks(n)
```

Description

`chr = blanks(n)` returns a `1`-by-`n` array of space characters.

Examples

Create Array of Blanks

Create an array of five blanks. To display it, embed it in a character array that starts and ends with a visible character.

```
b = blanks(5);
chr = ['|' b '|']
```

```
chr =
'|     |'
```

Embed the blanks in a string and display the string. Starting in R2017a, you can create strings using double quotes.

```
str = "Due Date:" + b + "2017-01-13"
```

```
str =
"Due Date:     2017-01-13"
```

Input Arguments

n — Number of space characters
integer

Number of space characters, specified as an integer.

- If `n` is `0`, then `blanks` returns a `1`-by-`0` character array.
- If `n` is less than `0`, then `blanks` treats it as `0`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`char` | `deblank` | `string`

Topics
"Text in String and Character Arrays"
"Formatting Text"

Introduced before R2006a

characteristic

Access a characteristic on Bluetooth Low Energy peripheral device

Description

A `characteristic` object represents a characteristic of a Bluetooth® Low Energy peripheral device. If read or write are supported in the object `Attributes` property, you can read characteristic values using `read` or you can write characteristic values using `write`. If the characteristic has any descriptors, you can also access them using `descriptor`.

Creation

Syntax

```
c = characteristic(b,serviceName,characteristicName)
c = characteristic(b,serviceUUID,characteristicUUID)
```

Description

`c = characteristic(b,serviceName,characteristicName)` creates an object that represents a characteristic on the peripheral device `b` using the service name and characteristic name. Identify `serviceName` and `characteristicName` by viewing the `Characteristics` property of the `ble` object `b`.

`c = characteristic(b,serviceUUID,characteristicUUID)` creates an object for a characteristic using its UUID and the service UUID. Identify `serviceUUID` and `characteristicUUID` by viewing the `Characteristics` property of the `ble` object `b`.

Input Arguments

b — Bluetooth Low Energy peripheral device connection
`ble` object

Bluetooth Low Energy peripheral device connection, specified as a `ble` object.

`serviceName` — Service name
character vector | string scalar

Service name of the characteristic, specified as a character vector or string scalar. Identify `serviceName` in the `Characteristics` table of the `ble` object that represents your peripheral device. The `ServiceName` column lists the valid values.

For a full list of standard services and how to interpret them, see the Bluetooth SIG website. All custom services in the `Characteristics` table have the name `"Custom"` and are differentiated by their UUID values. If the service name is `"Custom"`, then use the service UUID to access it.

Example: `c = characteristic(b,"Current Time Service","Current Time")` creates an object that represents the `"Current Time"` characteristic on the `"Current Time Service"`.

Data Types: `char` | `string`

`characteristicName` — Characteristic name
character vector | string scalar

Characteristic name, specified as a character vector or string scalar. Identify `characteristicName` in the `Characteristics` table of the `ble` object that represents your peripheral device. The `CharacteristicName` column lists the valid values.

For a full list of standard characteristics and how to interpret them, see the Bluetooth SIG website. All custom characteristics in the `Characteristics` table have the name `"Custom"` and are differentiated by their UUID values. If the characteristic name is `"Custom"`, then use the characteristic UUID to access it.

Example: `c = characteristic(b,"Current Time Service","Current Time")` creates an object that represents the `"Current Time"` characteristic on the `"Current Time Service"`.

Data Types: `char` | `string`

`serviceUUID` — Service UUID
character vector | string scalar

Service UUID of the characteristic, specified as a character vector or string scalar. Identify `serviceUUID` in the `Characteristics` table of the `ble` object that represents your peripheral device. The `ServiceUUID` column lists the valid values.

For a full list of standard services and how to interpret them, see the Bluetooth SIG website. If the service name is `"Custom"`, then use the service UUID to access it.

Example: `c = characteristic(b,"1805","2A2B")` creates an object that represents the characteristic with UUID `"2A2B"` on the service with UUID `"1805"`.

Data Types: `char` | `string`

`characteristicUUID` — Characteristic UUID
character vector | string scalar

Characteristic UUID, specified as a character vector or scalar string. Identify `characteristicUUID` in the `Characteristics` table of the `ble` object that represents your peripheral device. Valid values are those listed under the `CharacteristicUUID` column.

For a full list of standard characteristics and how to interpret them, see the Bluetooth SIG website. If the characteristic name is `"Custom"`, then use the characteristic UUID to access it.

Example: `c = characteristic(b,"1805","2A2B")` creates an object that represents the characteristic with UUID `"2A2B"` on the service with UUID `"1805"`.

Data Types: `char` | `string`

Properties

`Name` — Characteristic name
string scalar

This property is read-only.

Characteristic name, returned as a string scalar.

Data Types: `string`

`UUID` — Characteristic UUID
string scalar

This property is read-only.

Characteristic UUID, returned as a string scalar. This value is specific to each characteristic.

Data Types: `string`

`Attributes` — Characteristic attributes
string array

This property is read-only.

Characteristic attributes, returned as a string array. The value of this property determines the behavior of the `read`, `write`, `subscribe`, and `unsubscribe` functions. Possible values follow.

Value	Description
`"Read"`	Characteristic is readable
`"Write"`	Characteristic is writable
`"WriteWithoutResponse"`	Characteristic is writable with unacknowledged packets
`"AuthenticatedSignedWrites"`	Characteristic supports signed writes
`"Notify"`	Characteristic supports notification
`"Indicate"`	Characteristic supports indication
`"Broadcast"`	Characteristic supports broadcasting with advertising packets
`"ExtendedProperties"`	Characteristic contains `ExtendedProperties` descriptor with additional attributes
`"NotifyEncryptionRequired"` (macOS only)	Characteristic supports notification only from trusted devices
`"IndicateEncryptionRequired"` (macOS only)	Characteristic supports indication only from trusted devices

Data Types: `string`

`DataAvailableFcn` — Data available function
function handle

Data available function, returned as a function handle. This property is an empty array until you assign a function handle. Use this property for characteristics that support notification and/or indication.

For an example showing how to assign a callback to this property, see “Read Characteristic Data from a Bluetooth Low Energy Peripheral Device Using a Callback Function” on page 1-1004.

Example:

Write a function `displayHeartRate` that reads and displays data from a characteristic, create a handle, and assign the handle to this property.

```
c.DataAvailableFcn = @displayHeartRate;
```

The function is called whenever a new notification or indication is received. Stop notification or indication using `unsubscribe`.

Data Types: `function_handle`

`Descriptors` — Characteristic descriptors
table

Characteristic descriptors, returned as a table. To access a specific descriptor listed in the table, create an object using `descriptor`. The table contains the following columns.

Column	Description
`DescriptorName`	Name of descriptor, returned as a string
`DescriptorUUID`	UUID of descriptor, returned as a string
`Attributes`	Read or write permissions on descriptor, returned as a cell array

Data Types: `table`

Object Functions

read	Read characteristic or descriptor data on a Bluetooth Low Energy peripheral device
write	Write data to a characteristic or descriptor on a Bluetooth Low Energy peripheral device
subscribe	Subscribe to notification or indication from a characteristic
unsubscribe	Unsubscribe from notification and indication from a characteristic
descriptor	Access a descriptor on Bluetooth Low Energy peripheral device

Examples

Access a Characteristic on a Bluetooth Low Energy Peripheral Device

Connect to a nearby Bluetooth Low Energy peripheral device.

```
b = ble("DemoDev")
```

```
b = 
  ble with properties:

               Name: "DemoDev"
            Address: "FF548EA5658F"
          Connected: 1
           Services: [5×2 table]
    Characteristics: [10×5 table]

Show services and characteristics
```

List the available characteristics on this device.

```
b.Characteristics
```

```
ans=10×5 table
        ServiceName                     ServiceUUID                                Chara
    ___________________    ________________________________________    ___________________

    "Generic Access"       "1800"                                     "Device Name"
    "Generic Access"       "1800"                                     "Appearance"
    "Generic Access"       "1800"                                     "Peripheral Prefe
    "Generic Access"       "1800"                                     "Central Address F
    "Generic Attribute"    "1801"                                     "Service Changed"
    "Heart Rate"           "180D"                                     "Heart Rate Measu
    "Heart Rate"           "180D"                                     "Body Sensor Locat
    "Battery Service"      "180F"                                     "Battery Level"
    "Custom"               "03B80E5A-EDE8-4B33-A751-6CE34EC4C700"     "Custom"
    "Custom"               "03B80E5A-EDE8-4B33-A751-6CE34EC4C700"     "Custom"
```

This list shows all characteristics and their respective services. Some services have multiple characteristics. The list also shows the attributes for each characteristic.

Create a characteristic object that represents the `"Battery Level"` characteristic.

```
c = characteristic(b,"Battery Service","Battery Level")
```

```
c 
  Characteristic with properties:

             Name: "Battery Level"
             UUID: "2A19"
       Attributes: "Read"
      Descriptors: []
```

Alternatively, you can use the UUIDs instead of the names to create the object.

```
c = characteristic(b,"180F","2A19")
```

```
c 
  Characteristic with properties:

             Name: "Battery Level"
             UUID: "2A19"
       Attributes: "Read"
      Descriptors: []
```

This object represents the `"Battery Level"` characteristic. Since it has the `Read` attribute, you can use `read` on page 1-1002 to get the characteristic value.

See Also

`ble` | `descriptor` | `read` | `subscribe` | `unsubscribe` | `write`

Topics

"Bluetooth Low Energy Communication Overview"
"Work with Device Characteristics and Descriptors"
"Collect Data from Fitness Monitoring Devices"
"Track Orientation of Bluetooth Low Energy Device"

Introduced in R2019b

descriptor

Access a descriptor on Bluetooth Low Energy peripheral device

Description

A `descriptor` object represents a descriptor of a Bluetooth Low Energy peripheral device. If read or write are supported in the object `Attributes` property, you can read data using `read` or write data using `write`.

Creation

Syntax

```
d = descriptor(c,descriptorName)
d = descriptor(c,descriptorUUID)
```

Description

`d = descriptor(c,descriptorName)` creates an object that represents the descriptor specified by its name. Identify `descriptorName` by viewing the `Descriptors` property of the `characteristic` object `c`.

`d = descriptor(c,descriptorUUID)` creates an object that represents the descriptor specified by its UUID. Identify `descriptorUUID` by viewing the `Descriptors` property of the `characteristic` object `c`.

Input Arguments

c — Characteristic of Bluetooth Low Energy peripheral device
characteristic object

Characteristic of Bluetooth Low Energy peripheral device, specified as a `characteristic` object.

descriptorName — Descriptor name
character vector | string scalar

Descriptor name, specified as a character vector or string scalar. Identify `descriptorName` in the `Descriptors` table of the `characteristic` object. The `DescriptorName` column lists the valid values.

Example: `d = descriptor(c,"Client Characteristic Configuration")` creates an object that represents the `"Client Characteristic Configuration"` descriptor on the characteristic `c`.

Data Types: `char` | `string`

descriptorUUID — Descriptor UUID
character vector | string scalar

Descriptor UUID, specified as a character vector or string scalar. Identify `descriptorUUID` in the `Descriptors` table of the `characteristic` object. The `DescriptorUUID` column lists the valid values.

Example: `d = descriptor(c,"2902")` creates an object that represents the descriptor with UUID `"2902"` on the characteristic `c`.

Data Types: `char` | `string`

Properties

Name — Descriptor name
string scalar

This property is read-only.

Descriptor name, returned as a string scalar.

Data Types: `string`

UUID — Descriptor UUID
string scalar

This property is read-only.

Descriptor UUID, returned as a string scalar. This value is specific to each descriptor.

Data Types: `string`

`Attributes` — Descriptor attributes
string array

This property is read-only.

Descriptor attributes that describe the read and write permissions, returned as a string array. Possible values follow.

Value	Description
`"Read"`	Descriptor is readable using `read`
`"Write"`	Descriptor is writable using `write`

Data Types: `string`

Object Functions

read — Read characteristic or descriptor data on a Bluetooth Low Energy peripheral device
write — Write data to a characteristic or descriptor on a Bluetooth Low Energy peripheral device

Examples

Access a Descriptor on a Bluetooth Low Energy Peripheral Device

Create a connection to a nearby Bluetooth Low Energy peripheral device.

```
b = ble("Thingy")
```

```
b = 
  ble with properties:

              Name: "Thingy"
           Address: "F2DF635320F6"
         Connected: 1
          Services: [9×2 table]
    Characteristics: [38×5 table]
```

```
Show services and characteristics
```

Create a characteristic object that represents the `"Temperature"` characteristic.

```
c = characteristic(b,"Weather Station Service","Temperature")
```

```
c = 
  Characteristic with properties:

             Name: "Temperature"
             UUID: "EF680201-9B35-4933-9B10-52FFA9740042"
       Attributes: "Notify"
      Descriptors: [1x3 table]
 DataAvailableFcn: []

Show descriptors
```

List the descriptors for the characteristic.

```
c.Descriptors
```

```
ans=1×3 table
               DescriptorName                DescriptorUUID     Attributes
    ____________________________________    ______________    ___________

    "Client Characteristic Configuration"        "2902"        {1×2 string}
```

Create a descriptor object that represents `"Client Characteristic Configuration"`.

```
d = descriptor(c,"Client Characteristic Configuration")
```

```
d = 
  Descriptor with properties:

          Name: "Client Characteristic Configuration"
          UUID: "2902"
    Attributes: ["Read"    "Write"]
```

Alternatively, you can use the UUID instead of the name to create the object.

```
d = descriptor(c,"2902")
```

```
d = 
  Descriptor with properties:

          Name: "Client Characteristic Configuration"
          UUID: "2902"
    Attributes: ["Read"    "Write"]
```

This object represents the `"Client Characteristic Configuration"` descriptor. Since it has both `Read` and `Write` attributes, you can use `read` on page 1-1002 or `write` on page 1-1018.

See Also

`characteristic` | `read` | `write`

Topics

"Bluetooth Low Energy Communication Overview"
"Work with Device Characteristics and Descriptors"

Introduced in R2019b

read

Package: `matlabshared.blelib`

Read characteristic or descriptor data on a Bluetooth Low Energy peripheral device

Syntax

```
characteristicData = read(c)
characteristicData = read(c,mode)
[characteristicData,timestamp] = read( ___ )

descriptorData = read(d)
```

Description

Read Characteristic Values

`characteristicData = read(c)` reads the characteristic value from a Bluetooth Low Energy peripheral device. The data read depends on the `Attributes` property of the input characteristic object `c`. For more information about all the possible behaviors of `read`, see `characteristicData` on page 1-0 .

`characteristicData = read(c,mode)` specifies `mode` as the read mode.

`[characteristicData,timestamp] = read( ___ )` reads the timestamp for any of the previous syntaxes.

Read Descriptor Values

`descriptorData = read(d)` reads the descriptor value from a Bluetooth Low Energy peripheral device.

Examples

Read Characteristic Data from a Bluetooth Low Energy Peripheral Device

Access a characteristic on your peripheral device and read its data.

Create a connection to a nearby Bluetooth Low Energy peripheral device.

```
b = ble("Thingy")
```

```
b = 
  ble with properties:

               Name: "Thingy"
            Address: "F2DF635320F6"
          Connected: 1
           Services: [9×2 table]
    Characteristics: [38×5 table]

Show services and characteristics
```

Create a characteristic object that represents the `"Temperature"` characteristic.

```
c = characteristic(b,"Weather Station Service","Temperature")
```

```
c = 
  Characteristic with properties:

             Name: "Temperature"
             UUID: "EF680201-9B35-4933-9B10-52FFA9740042"
       Attributes: "Notify"
      Descriptors: [1x3 table]
 DataAvailableFcn: []

Show descriptors
```

Because this characteristic supports `"Notify"`, you can use `read` on page 1-1002 to get the latest data.

```
data = read(c)
```

```
data = 1×2

    23    75
```

You can also return the timestamp of the latest data.

```
[data,timestamp] = read(c)
```

```
data = 1×2

    23    73


timestamp = datetime
   16-May-2019 16:20:00
```

Interpret the data in Celsius. The first byte represents the integer part of the temperature and the second byte represents the decimal part with a resolution of 0.01.

```
temperature = data(1) + data(2)*0.01
```

```
temperature = 23.7300
```

Read Characteristic Data from a Bluetooth Low Energy Peripheral Device Using a Callback Function

Access a characteristic on your peripheral device and create a callback function to read its data.

Create a connection to a nearby Bluetooth Low Energy peripheral device.

```
b = ble("Thingy")
```

```
b = 
  ble with properties:

              Name: "Thingy"
           Address: "F2DF635320F6"
         Connected: 1
          Services: [9×2 table]
    Characteristics: [38×5 table]

Show services and characteristics
```

Create a characteristic object that represents the `"Temperature"` characteristic.

```
c = characteristic(b,"Weather Station Service","Temperature")
```

```
c = 
  Characteristic with properties:

             Name: "Temperature"
             UUID: "EF680201-9B35-4933-9B10-52FFA9740042"
       Attributes: "Notify"
      Descriptors: [1x3 table]
 DataAvailableFcn: []

Show descriptors
```

Because this characteristic supports `"Notify"`, you can create a callback function. Name the function `displayCharacteristicData` and define it as follows. Specify the read mode as `'oldest'` instead of `'latest'`. Calling the `'latest'` data may lead to errors in the callback function caused by the flushing of previous data.

```
function displayCharacteristicData(src,evt)
    [data,timestamp] = read(src,'oldest');
    disp(data);
    disp(timestamp);
end
```

Use the `@` operator to assign the function handle to the `DataAvailableFcn` property of the characteristic. When a new notification is available, the data appears in your command window.

```
c.DataAvailableFcn = @displayCharacteristicData
```

```
c = 
  Characteristic with properties:

             Name: "Temperature"
             UUID: "EF680201-9B35-4933-9B10-52FFA9740042"
       Attributes: "Notify"
      Descriptors: [1x3 table]
 DataAvailableFcn: displayCharacteristicData

Show descriptors
```

After you finish working with the characteristic, disable notifications using `unsubscribe` on page 1-1015.

```
unsubscribe(c)
```

Read Descriptor Data from a Bluetooth Low Energy Peripheral Device

Access a descriptor on your peripheral device and read its data.

Create a connection to a nearby Bluetooth Low Energy peripheral device.

```
b = ble("DemoDev")
```

```
b = 
  ble with properties:

               Name: "DemoDev"
            Address: "FF548EA5658F"
          Connected: 1
           Services: [5×2 table]
    Characteristics: [10×5 table]

Show services and characteristics
```

Create a characteristic object that represents the `"Heart Rate Measurement"` characteristic.

```
c = characteristic(b,"Heart Rate","Heart Rate Measurement")
```

```
c = 
  Characteristic with properties:

             Name: "Heart Rate Measurement"
             UUID: "2A37"
       Attributes: "Notify"
      Descriptors: [1x3 table]
 DataAvailableFcn: []

Show descriptors
```

Create a descriptor object that represents the `"Client Characteristic Configuration"` descriptor.

```
d = descriptor(c,"Client Characteristic Configuration")
```

```
d = 
  Descriptor with properties:
```

```
          Name: "Client Characteristic Configuration"
          UUID: "2902"
    Attributes: ["Read"    "Write"]
```

This descriptor contains information about whether notification and indication are enabled or disabled. You can use `read` to get the current data.

```
data = read(d)
```

```
data = 1×2

     0     0
```

Interpret this data by referring to the specification for this descriptor on the Bluetooth SIG website.

This value changes when the notification or indication status changes. For example, subscribe to notifcation using `subscribe` on page 1-1011. Then, observe the change in the value by reading the descriptor again.

```
subscribe(c,'notification');
data = read(d)
```

```
data = 1×2

     1     0
```

Input Arguments

c — Characteristic of Bluetooth Low Energy peripheral device
characteristic object

Characteristic of Bluetooth Low Energy peripheral device, specified as a `characteristic` object.

The `Attributes` property of the characteristic object must include `"Read"`, `"Notify"`, or `"Indicate"` to read data.

Example: `data = read(c)` reads the value of the characteristic object `c`.

`mode` — Read mode
`'latest'` (default) | `'oldest'`

Read mode, specified as `'latest'` or `'oldest'`. Using `'latest'` returns the most recent data and flushes the previous data. Using `'oldest'` returns the oldest data since the last read.

Note Use `'oldest'` inside the `DataAvailableFcn` callback function to avoid errors caused by the flushing of previous data.

Example: `data = read(c,'oldest')` reads the oldest value since the last read on the characteristic object `c`.

Data Types: `char` | `string`

`d` — Descriptor of Bluetooth Low Energy peripheral device
descriptor object

Descriptor of Bluetooth Low Energy peripheral device, specified as a `descriptor` object.

The `Attributes` property of the descriptor object must include `"Read"` to read data.

Example: `read(d)` reads the value of the descriptor object `d`.

Output Arguments

`characteristicData` — Characteristic data
numeric

Characteristic data from peripheral device, returned as a number or array of numbers.

The data read depends on the `Attributes` property of the characteristic object and the specified read “mode” on page 1-0 .

`c.Attributes`	`read(c)` or `read(c,'latest')`	`read(c,'oldest')`
• `"Read"`	Current data.	Not supported.

c.Attributes	read(c) or read(c,'latest')	read(c,'oldest')
• "Notify", "Indicate", or both	Latest notification or indication data. • If notification or indication is not enabled and this is the first time calling read, then notification or indication automatically starts. • Previous data is flushed.	Oldest notification or indication data since last read. • If notification or indication is not enabled and this is the first time calling read, then notification or indication automatically starts.
• "Read" and • "Notify", "Indicate", or both	• If notification or indication is not enabled, then characteristicData is current data. • If notification or indication is enabled, then characteristicData is the latest notification or indication.	• If notification or indication is not enabled, then read(c,'oldest') is not supported. • If notification or indication is enabled, then characteristicData is the oldest notification or indication data since the last read and previous data is flushed.

Data Types: double

timestamp — Timestamp
datetime

Timestamp indicating the receipt of characteristic or descriptor data on the computer, returned as a datetime array.

Data Types: datetime

descriptorData — Descriptor data
numeric

Descriptor data from peripheral device, returned as a number.

Data Types: `double`

See Also

`characteristic` | `descriptor` | `subscribe` | `unsubscribe` | `write`

Topics

"Work with Device Characteristics and Descriptors"
"Collect Data from Fitness Monitoring Devices"
"Track Orientation of Bluetooth Low Energy Device"

Introduced in R2019b

subscribe

Package: `matlabshared.blelib`

Subscribe to notification or indication from a characteristic

Syntax

```
subscribe(c)
subscribe(c,type)
```

Description

`subscribe(c)` subscribes to a characteristic notification or indication of a Bluetooth Low Energy peripheral device. Notification or indication depends on the `Attributes` property of the characteristic `c`. The property must contain `"Notify"`, `"Indicate"`, or both. If the property contains both `"Notify"` and `"Indicate"`, the function subscribes to notification.

`subscribe(c,type)` subscribes to a characteristic of a Bluetooth Low Energy peripheral device and specifies whether the subscription is for notification or indication. The `Attributes` property of the characteristic `c` must contain the specified `type`.

Examples

Subscribe and Unsubscribe to a Characteristic on a Bluetooth Low Energy Peripheral Device

Subscribe to a characteristic that supports notification or indication. Unsubscribe after you finish working with the characteristic.

Create a connection to a nearby Bluetooth Low Energy peripheral device.

```
b = ble("DemoDev")
```

```
b = 
  ble with properties:

               Name: "DemoDev"
            Address: "FF548EA5658F"
          Connected: 1
           Services: [5×2 table]
    Characteristics: [10×5 table]

Show services and characteristics
```

Create a characteristic object that represents the `"Heart Rate Measurement"` characteristic.

```
c = characteristic(b,"Heart Rate","Heart Rate Measurement")
```

```
c = 
  Characteristic with properties:

             Name: "Heart Rate Measurement"
             UUID: "2A37"
       Attributes: "Notify"
      Descriptors: [1x3 table]
 DataAvailableFcn: []

Show descriptors
```

This characteristic supports `"Notify"`.

Start receiving notifications using `subscribe`.

```
subscribe(c)
```

Read from the characteristic to check that you are receiving data.

```
read(c)
```

```
ans = 1×19

    23    14     1   187     1   186     1   185     1   184     1   183     1   182
```

After you finish working with the characteristic, disable notifcations using `unsubscribe`.

```
unsubscribe(c)
```

Input Arguments

c — Characteristic of Bluetooth Low Energy peripheral device
characteristic object

Characteristic of Bluetooth Low Energy peripheral device, specified as a `characteristic` object.

Example: `subscribe(c)` subscribes to notification or indication on the characteristic object `c`, depending on the `Attributes` property of `c`.

type — Subscription type
`"notification"` | `"indication"`

Subscription type, specified as `"notification"` or `"indication"`. If a characteristic supports both and you do not specify type, then the default value is `"notification"`.

Possible behaviors of `subscribe` follow.

`c.Attributes` includes	Notification supported?	Indication supported?
`"Notify"`	Yes, using • `subscribe(c)` or • `subscribe(c,"notification")`	No
`"Indicate"`	No	Yes, using • `subscribe(c)` or • `subscribe(c,"indication")`
`"Notify"` and `"Indicate"`	Yes, using • `subscribe(c)` or • `subscribe(c,"notification")`	Yes, using • `subscribe(c,"indication")`

Example: `subscribe(c,"indication")` subscribes to indication.

See Also

`characteristic` | `read` | `unsubscribe` | `write`

Topics

"Work with Device Characteristics and Descriptors"

Introduced in R2019b

unsubscribe

Package: `matlabshared.blelib`

Unsubscribe from notification and indication from a characteristic

Syntax

```
unsubscribe(c)
```

Description

`unsubscribe(c)` unsubscribes from both notification and indication of a characteristic of a Bluetooth Low Energy peripheral device.

Examples

Subscribe and Unsubscribe to a Characteristic on a Bluetooth Low Energy Peripheral Device

Subscribe to a characteristic that supports notification or indication. Unsubscribe after you finish working with the characteristic.

Create a connection to a nearby Bluetooth Low Energy peripheral device.

```
b = ble("DemoDev")
```

```
b 
  ble with properties:

              Name: "DemoDev"
           Address: "FF548EA5658F"
         Connected: 1
          Services: [5×2 table]
    Characteristics: [10×5 table]
```

```
Show services and characteristics
```

Create a characteristic object that represents the `"Heart Rate Measurement"` characteristic.

```
c = characteristic(b,"Heart Rate","Heart Rate Measurement")
```

```
c 
  Characteristic with properties:

             Name: "Heart Rate Measurement"
             UUID: "2A37"
       Attributes: "Notify"
      Descriptors: [1x3 table]
 DataAvailableFcn: []

Show descriptors
```

This characteristic supports `"Notify"`.

Start receiving notifications using `subscribe`.

```
subscribe(c)
```

Read from the characteristic to check that you are receiving data.

```
read(c)
```

```
ans = 1×19

    23    14     1   187     1   186     1   185     1   184     1   183     1   182
```

After you finish working with the characteristic, disable notifcations using `unsubscribe`.

```
unsubscribe(c)
```

Input Arguments

c — Characteristic of Bluetooth Low Energy peripheral device
characteristic object

Characteristic of Bluetooth Low Energy peripheral device, specified as a `characteristic` object.

Example: `unsubscribe(c)` unsubscribes from notification and indication on the characteristic object `c`.

See Also

`characteristic` | `read` | `subscribe` | `write`

Topics

"Work with Device Characteristics and Descriptors"

Introduced in R2019b

write

Package: `matlabshared.blelib`

Write data to a characteristic or descriptor on a Bluetooth Low Energy peripheral device

Syntax

```
write(c,data)
write(c,data,type)
write(c,data,precision)
write(c,data,precision,type)

write(d,data)
write(d,data,precision)
```

Description

Write Characteristic Values

`write(c,data)` writes the specified `data` to a characteristic of a Bluetooth Low Energy peripheral device. The `Attributes` property of the input characteristic object `c` must be `"Write"` and/or `"WriteWithoutResponse"`.

`write(c,data,type)` specifies whether the device expects a response back using `type`.

`write(c,data,precision)` specifies the precision of the data written.

`write(c,data,precision,type)` specifies both the response type and the data precision.

Write Descriptor Values

`write(d,data)` writes the specified `data` to a descriptor of a Bluetooth Low Energy peripheral device. The descriptor `d` must have `"Write"` in its `Attributes` property.

`write(d,data,precision)` specifies the precision of the data written.

Examples

Write to a Characteristic on a Bluetooth Low Energy Peripheral Device

Write data to a characteristic on a Bluetooth Low Energy Peripheral Device.

Create a connection to a nearby Bluetooth Low Energy peripheral device.

```
b = ble("DemoDevice")
```

```
b = 
  ble with properties:

              Name: "DemoDevice"
           Address: "5A0B858BC07C"
         Connected: 1
          Services: [5×2 table]
    Characteristics: [12×5 table]

Show services and characteristics
```

Create a characteristic object that represents the `"Gender"` characteristic.

```
c = characteristic(b,"User Data","Gender")
```

```
c = 
  Characteristic with properties:

            Name: "Gender"
            UUID: "2A8C"
      Attributes: "Read" "Write"
     Descriptors: []
```

Because this characteristic is readable and writable, you can write data to it and verify the change in values. Use `read` on page 1-1002 to get the latest data.

```
data = read(c)
```

```
data = 0
```

Interpret the data by referring to the specification for this characteristic on the Bluetooth SIG website. `0` represents male and `1` represents female. Write `1` to the characteristic to indicate female.

```
write(c,1)
```

You can read from the characteristic again to observe the change in the data.

```
data = read(c)
```

```
data = 1
```

Write to a Descriptor on a Bluetooth Low Energy Peripheral Device

Write data to a descriptor on a Bluetooth Low Energy Peripheral Device.

Create a connection to a nearby Bluetooth Low Energy peripheral device.

```
b = ble("DemoDev")
```

```
b = 
  ble with properties:

               Name: "DemoDev"
            Address: "FF548EA5658F"
          Connected: 1
           Services: [5×2 table]
    Characteristics: [10×5 table]

Show services and characteristics
```

Create a characteristic object that represents the `"Heart Rate Measurement"` characteristic.

```
c = characteristic(b,"Heart Rate","Heart Rate Measurement")
```

```
c = 
  Characteristic with properties:

             Name: "Heart Rate Measurement"
             UUID: "2A37"
       Attributes: "Notify"
      Descriptors: [1x3 table]
 DataAvailableFcn: []

Show descriptors
```

Create a descriptor object that respresents the `"Client Characteristic Configuration"` descriptor.

```
d = descriptor(c,"Client Characteristic Configuration")
```

```
d = 
  Descriptor with properties:

          Name: "Client Characteristic Configuration"
          UUID: "2902"
    Attributes: ["Read"    "Write"]
```

This descriptor contains information about whether notification or indication are enabled or disabled. You can use `read` on page 1-1002 to get the current data.

```
data = read(d)
```

```
data = 1×2

     0     0
```

Interpret this data by referring to the specification for this descriptor on the Bluetooth SIG website.

This value changes when the notification or indication status changes. For example, write to this value to enable notification for the `"Heart Rate Measurement"` characteristic. Then, observe the change in values by reading the descriptor again.

```
write(d,[1 0])
data = read(d)
```

```
data = 1×2

     1     0
```

Input Arguments

c — Characteristic of Bluetooth Low Energy peripheral device
characteristic object

Characteristic of Bluetooth Low Energy peripheral device, specified as a `characteristic` object.

`data` — Data to write
numeric

Data to write to the characteristic or descriptor, specified as a scalar or an array of numbers. Refer to your characteristic or descriptor specifications to determine what kind of data to write.

Example: `write(c,[1 0])` writes an array of numbers to the characteristic `c`.

Data Types: `double` | `uint8` | `uint16` | `uint32` | `uint64`

`type` — Response option type
`"withresponse"` | `"withoutresponse"`

Response option type for the characteristic, specified as `"withresponse"` or `"withoutresponse"`. If you specify `"withresponse"`, the peripheral device expects a response from the device indicating that the write is successful. If you specify `"withoutresponse"`, the peripheral device expects no response. The default value depends on the `Attributes` property of the characteristic.

`c.Attributes`	Default type
`"Write"`	`"withresponse"`
`"WriteWithoutResponse"`	`"withoutresponse"`
`"Write"` and `"WriteWithoutResponse"`	`"withresponse"`

Example: `write(c,5,"withoutresponse")` writes data to the characteristic without receiving a response back.

`precision` — Data precision
`"uint8"` (default) | `"uint16"` | `"uint32"` | `"uint64"`

Data precision, specified as `"uint8"`, `"uint16"`, `"uint32"`, or `"uint64"`.

Example: `write(d,300,"uint16")` writes data to the characteristic as an unsigned 16-bit integer.

`d` — Descriptor of Bluetooth Low Energy peripheral device
descriptor object

Descriptor of Bluetooth Low Energy peripheral device, specified as a `descriptor` object.

See Also

`characteristic` | `descriptor` | `read` | `subscribe` | `unsubscribe`

Topics

"Work with Device Characteristics and Descriptors"

Introduced in R2019b

ble

Connect to Bluetooth Low Energy peripheral device

Description

A `ble` object represents a connection to a Bluetooth Low Energy peripheral device from your computer. Find nearby connectable devices using `blelist`.

Creation

Syntax

```
b = ble(name)
b = ble(address)
```

Description

`b = ble(name)` connects to a Bluetooth Low Energy device using its name.

`b = ble(address)` connects to a Bluetooth Low Energy device using its address.

Input Arguments

name — Device name
character vector | string scalar

Device name, specified as a character vector or string scalar. Identify the name of the peripheral device you want to connect to using the information returned by `blelist`. If a device does not have a `name`, then use its address to connect to it.

Example: `b = ble("Tile")` connects to the peripheral device with the name "`Tile`".

Data Types: `char` | `string`

`address` — Device address
character vector | string scalar

Device address, specified as a character vector or string scalar. Identify the address of the peripheral device you want to connect to using the information returned by `blelist`. If a device does not have a `name`, then use its address to connect to it.

Example: `b = ble("5A8769A09DA6")` connects to the peripheral device with the MAC address "`5A8769A09DA6`" on a Windows computer.

Example: `b = ble("121B0BAC-0DE7-4FB1-909A-EEA6877DBC5C")` connects to the peripheral device with the random OS-assigned address "`121B0BAC-0DE7-4FB1-909A-EEA6877DBC5C`" on a macOS computer.

Data Types: `char` | `string`

Properties

`Name` — Peripheral device name
string scalar

This property is read-only.

Peripheral device name, returned as a string scalar.

Data Types: `string`

`Address` — Peripheral device address
string scalar

This property is read-only.

Peripheral device address, returned as a string scalar. On Windows, this value is the 48-bit Bluetooth MAC address of the device. On macOS, this value is a random 128-bit OS-assigned address.

Data Types: `string`

`Connected` — Peripheral device connection status
`true` or `1` | `false` or `0`

This property is read-only.

Peripheral device connection status, returned as numeric or logical `1` (`true`) or `0` (`false`).

Data Types: `logical`

`Services` — Services included on peripheral device
table

This property is read-only.

Services included on the peripheral device, returned as a table. The table has the following columns.

Column	Description
`ServiceName`	Name of the service included on the peripheral device, returned as a string
`ServiceUUID`	UUID of the service included on the peripheral device, returned as a string

For a full list of standard services and their UUIDs, see the Bluetooth SIG website. All custom services have the named "`Custom`" and are differentiated by their UUID values.

Note On macOS, the "`Generic Access`" and "`Generic Attributes`" services and their characteristics are hidden. The `Services` and `Characteristics` properties do not show them.

Data Types: `table`

`Characteristics` — Characteristics included on peripheral device
table

This property is read-only.

Characteristics included on the peripheral device, returned as a table. To access a specific characteristic listed in the table, create an object using `characteristic`. This table has the following columns.

Column	Description
`ServiceName`	Name of the service included on the peripheral device, returned as a string
`ServiceUUID`	UUID of the service included on the peripheral device, returned as a string
`CharacteristicName`	Name of the characteristic included on the peripheral device, returned as a string
`CharacteristicUUID`	UUID of the characteristic included on the peripheral device, returned as a string
`Attributes`	Supported operations on a characteristic, returned as a cell array. See the “Attributes” on page 1-0 property for possible values

For a full list of standard characteristics and how to interpret them, see the Bluetooth SIG website. All custom characteristics have the name "`Custom`" in the `Characteristics` table and are differentiated by their UUID values.

Note On macOS, the "`Generic Access`" and "`Generic Attributes`" services and their characteristics are hidden. The `Services` and `Characteristics` properties do not show them.

Data Types: `table`

Object Functions

characteristic Access a characteristic on Bluetooth Low Energy peripheral device

Examples

Connect to a Bluetooth Low Energy Peripheral Device Using Its Name

Search for and establish a connection to your Bluetooth Low Energy peripheral device.

Scan for nearby peripheral devices that have the name `"Gear Fit2 (E16A)"` using `blelist`.

```
blelist("Name","Gear Fit2 (E16A)")
```

```
ans=1×5 table
    Index           Name              Address        RSSI    Advertisement
    _____    __________________    _____________    ____    _____________

      1      "Gear Fit2 (E16A)"    "8498663EE16A"    -61     [1×1 struct]
```

Create a connection to the device.

```
b = ble("Gear Fit2 (E16A)")
```

```
b = 
  ble with properties:

               Name: "Gear Fit2 (E16A)"
            Address: "8498663EE16A"
          Connected: 1
           Services: [2×2 table]
    Characteristics: [3×5 table]

Show services and characteristics
```

Connect to a Bluetooth Low Energy Peripheral Device Using Its Address on Windows

Search for and establish a connection to your Bluetooth Low Energy peripheral device.

Scan for nearby peripheral devices that have the name `"Gear Fit2 (E16A)"` using `blelist`.

```
blelist("Name","Gear Fit2 (E16A)")
```

```
ans=1×5 table
    Index           Name              Address        RSSI    Advertisement
    _____    __________________    _____________    ____    _____________

      1      "Gear Fit2 (E16A)"    "8498663EE16A"    -58     [1×1 struct]
```

Create a connection to the device with address `"8498663EE16A"`. On a Windows computer, this address represents the 48-bit Bluetooth MAC address of the peripheral device.

```
b = ble("8498663EE16A")
```

```
b = 
  ble with properties:

               Name: "Gear Fit2 (E16A)"
            Address: "8498663EE16A"
          Connected: 1
           Services: [2×2 table]
    Characteristics: [3×5 table]

Show services and characteristics
```

Connect to a Bluetooth Low Energy Peripheral Device Using Its Address on macOS

Search for and establish a connection to your Bluetooth Low Energy peripheral device.

Scan for nearby peripheral devices that have the name `"Gear Fit2 (E16A)"` using `blelist`.

```
blelist("Name","Gear Fit2 (E16A)")
```

```
ans=1×5 table
    Index           Name                          Address                    RSSI    A
    _____    ___________________    ____________________________________    ____    _

      1      "Gear Fit2 (E16A)"    "2F4AFADA-1F14-4BBD-A1F1-E7E64C0CF5BA"    -49     [
```

Create a connection to the device with address `"2F4AFADA-1F14-4BBD-A1F1-E7E64C0CF5BA"`. On a macOS computer, this address represents the random 128-bit OS-assigned address of the peripheral device.

```
b = ble("2F4AFADA-1F14-4BBD-A1F1-E7E64C0CF5BA")
```

```
b = 
  ble with properties:
```

```
               Name: "Gear Fit2 (E16A)"
            Address: "2F4AFADA-1F14-4BBD-A1F1-E7E64C0CF5BA"
          Connected: 1
           Services: []
    Characteristics: []

Show services and characteristics
```

See Also

`blelist` | `characteristic`

Topics

"Bluetooth Low Energy Communication Overview"
"Find Your Bluetooth Low Energy Peripheral Devices"

Introduced in R2019b

blelist

Scan nearby Bluetooth Low Energy peripheral devices

Syntax

```
list = blelist
list = blelist(Name,Value)
```

Description

`list = blelist` scans nearby advertising Bluetooth Low Energy peripheral devices detected by your computer. Use `ble` to connect to a peripheral device.

`list = blelist(Name,Value)` scans using optional name-value pair arguments. Use name-value pair arguments to scan for specific devices or to scan for a longer time.

Examples

Scan for Nearby Bluetooth Low Energy Peripheral Devices

Scan for nearby Bluetooth Low Energy peripheral devices and look at the device advertisement data.

Use `blelist` to list all nearby peripheral devices that are currently advertising data. If your peripheral device is already connected to another device or application, it may not appear in this list. Disconnect your peripheral device from any other devices or applications first for MATLAB® to discover it.

```
list = blelist
```

```
list=14×5 table
    Index           Name              Address        RSSI    Advertisement
    _____    ___________________    _______________    ____    _____________
```

```
     1      "Gear Fit2 (E16A)"    "8498663EE16A"    -54     [1×1 struct]
     2      ""                    "2C4D2724754D"    -69     [1×1 struct]
     3      ""                    "1B75E09FD18F"    -70     [1×1 struct]
     4      ""                    "4F7D6DAF9FCE"    -75     [1×1 struct]
     5      ""                    "7B8ADB5851BD"    -76     [1×1 struct]
     6      ""                    "1838D92EC312"    -78     [1×1 struct]
     7      ""                    "6A71507635AC"    -86     [1×1 struct]
     8      ""                    "68CA4203A060"    -86     [1×1 struct]
     9      ""                    "65C057AD64E5"    -88     [1×1 struct]
    10      ""                    "399A1B26F47D"    -88     [1×1 struct]
    11      ""                    "78BDCC3784C0"    -89     [1×1 struct]
    12      ""                    "5446498F593E"    -91     [1×1 struct]
    13      ""                    "67992F048DDB"    -92     [1×1 struct]
    14      ""                    "3A0DAC7C7DCE"    -94     [1×1 struct]
```

The RSSI values indicate signal strength. The table lists devices with a stronger signal first. These devices are closer.

View advertisement data of the first device.

```
list.Advertisement(1)
```

```
ans = struct with fields:
                            Type: ["Connectable Undirected"    "Scan Response"]
                      Appearance: "Generic Watch"
              ShortenedLocalName: []
               CompleteLocalName: "Gear Fit2 (E16A)"
                     TxPowerLevel: []
    SlaveConnectionIntervalRange: []
        ManufacturerSpecificData: [0 117 1 0 2 0 1 3 0]
                     ServiceData: []
           CompleteServiceUUIDs: []
         IncompleteServiceUUIDs: []
        ServiceSolicitationUUIDs: []
```

If your device has a larger advertising interval than the default timeout value of three seconds, you can specify a larger timeout value.

```
list = blelist('Timeout',10);
```

Scan for Specific Bluetooth Low Energy Peripheral Devices

You can use name-value pair arguments to scan for Bluetooth Low Energy peripheral devices with a specified name prefix or advertising specified services.

Scan for peripheral devices with names starting with the string `"UA"`.

```
list = blelist("Name","UA")
```

```
list=2×5 table
    Index           Name                Address        RSSI    Advertisement
    _____    ____________________    _____________    ____    _____________

      1      "UA E39 MODULE"         "84DD20E39AB6"    -84     [1×1 struct]
      2      "UA Footpod 239AE2"     "0CF3EE239AE2"    -87     [1×1 struct]
```

You can also scan for peripheral devices by their services. Scan for devices that advertise the `Heart Rate` service.

```
list = blelist("Services","Heart Rate")
```

```
list=1×5 table
    Index    Name       Address        RSSI    Advertisement
    _____    ____    _____________    ____    _____________

      1       ""     "84DD20E39AB6"    -85     [1×1 struct]
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `list = blelist("Name","UA")` returns all peripheral devices with names that start with `"UA"`.

Name — Peripheral device name
character vector | string scalar

Peripheral device name for filtering results, specified as the comma-separated pair consisting of `'Name'` and a character vector or string scalar. Using this name-value pair argument lists only peripheral devices that start with the specified value in their names.

Example: `list = blelist("Name","Pebble")` returns all peripheral devices with names that start with "`Pebble`".

Data Types: `char` | `string`

Services — Service name or UUID
character vector | string scalar | cell array of character vectors | string array

Service name or UUID for filtering results, specified as the comma-separated pair consisting of `'Services'` and a character vector, string scalar, cell array of character vectors, or string array. Using this name-value pair argument produces a list of only peripheral devices that advertise all the specified services.

Example: `list = blelist("Services",["180a","180d"])` returns all peripheral devices that advertise both services with the UUIDs `180a` and `180d`.

Example: `list = blelist("Services","Heart Rate")` returns all peripheral devices that advertise the service "`Heart Rate`".

Data Types: `char` | `string` | `cell`

Timeout — Amount of time to scan for devices
3 (default) | numeric

Amount of time to scan for in seconds, specified as the comma-separated pair consisting of `'Timeout'` and a number. The value must be less than 10,485. If you are unable to find your device using `blelist`, try a larger value for this parameter.

Example: `list = blelist("Timeout",20)` searches for nearby peripheral devices for 20 seconds.

Data Types: `double`

Output Arguments

list — List of nearby peripheral devices
table

List of nearby peripheral devices, returned as a table. The table has the following columns.

`Name` — Peripheral device name
string scalar

Peripheral device name, returned as a string scalar.

Data Types: `string`

`Address` — Peripheral device address
string scalar

Peripheral device identifier, returned as a string scalar. On Windows, this value is the 48-bit Bluetooth MAC address of the device. On macOS, this value is a random 128-bit OS-assigned address.

Data Types: `string`

`RSSI` — Signal strength
double

Signal strength in dB, returned as a double. RSSI is the received signal strength indicator. The table lists devices with a stronger signal first. These devices are closer to your computer.

Data Types: `double`

`Advertisement` — Advertisement data
structure

Advertisement data of a peripheral device, returned as a structure. `Advertisement` contains connection information about each peripheral device.

On Windows, the `Type` field shows the connection status, returned as a string. The possible values follow.

Value	Peripheral device description
`"Connectable Undirected"`	Can connect to your computer
`"Connectable Directed"`	Can connect to your computer only if it is programmed to do so

Value	Peripheral device description
`"Nonconnectable Undirected"`	Cannot connect to your computer but sends advertising information
`"Scannable Undirected"`	Cannot connect to your computer but is scannable
`"Scan Response"`	Sends a scan response to a scan request

On macOS, the `Connectable` field shows the connection status, returned as a number. `1` indicates true, or connectable, and `0` or any other number indicates false, or nonconnectable.

The rest of the fields in `Advertisement` are other advertisement data defined in the Generic Access Profile (GAP) by Bluetooth SIG. For a full list of advertising data types and references to their specifications, see the Bluetooth SIG website.

Data Types: `struct`

See Also

`ble`

Topics

"Bluetooth Low Energy Communication Overview"
"Find Your Bluetooth Low Energy Peripheral Devices"

Introduced in R2019b

blkdiag

Block diagonal matrix

Syntax

```
B = blkdiag(A1,...,AN)
```

Description

`B = blkdiag(A1,...,AN)` returns the block diagonal matrix on page 1-1038 created by aligning the input matrices `A1,...,AN` along the diagonal of `B`.

Examples

Diagonal of Three Matrices

Create a block diagonal matrix from three matrices of different sizes.

```
A1 = ones(2,2);
A2 = 2*ones(3,2);
A3 = 3*ones(2,3);
B = blkdiag(A1,A2,A3)
```

```
B = 7×7

     1     1     0     0     0     0     0
     1     1     0     0     0     0     0
     0     0     2     2     0     0     0
     0     0     2     2     0     0     0
     0     0     2     2     0     0     0
     0     0     0     0     3     3     3
     0     0     0     0     3     3     3
```

Input Arguments

`A1,...,AN` — Input matrices
matrices

Input matrices, specified as a comma-separated list of matrices. The matrices can be either square or rectangular and can differ in size.

If any of the input matrices are sparse, then the output block diagonal matrix is also sparse.

More About

Block Diagonal Matrix

A block diagonal matrix is a matrix whose diagonal contains blocks of smaller matrices, in contrast to a regular diagonal matrix with single elements along the diagonal. A block diagonal matrix takes on the following form, where *A1*, *A2*,..., *AN* are each matrices that can differ in size:

$$\begin{bmatrix} A1 & 0 & 0 & 0 \\ 0 & A2 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & AN \end{bmatrix}$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`diag`

Introduced before R2006a

bone

Bone colormap array

Syntax

```
c = bone
c = bone(m)
```

Description

`c = bone` returns the bone colormap as a three-column array with the same number of rows as the colormap for the current figure. If no figure exists, then the number of rows is equal to the default length of 256. Each row in the array contains the red, green, and blue intensities for a specific color. The intensities are in the range [0,1], and the color scheme looks like this image.

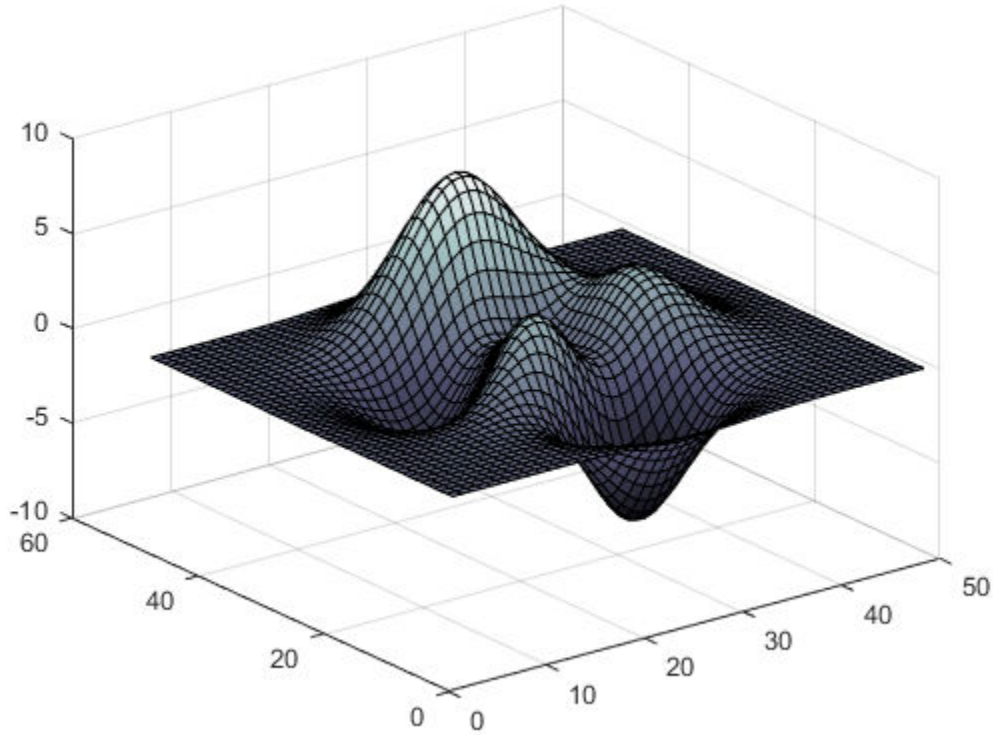
`c = bone(m)` returns the colormap with `m` colors.

Examples

Reverse the Colormap

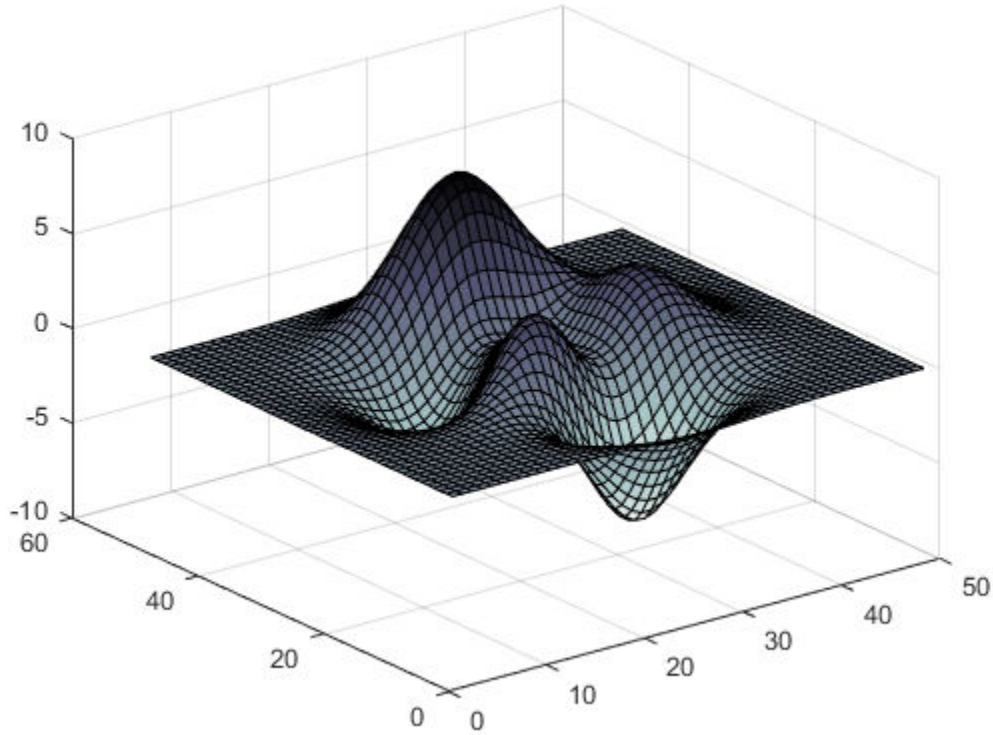
Plot a surface and assign the bone colormap.

```
surf(peaks);
colormap('bone');
```



Get the bone colormap array and reverse the order. Then apply the modified colormap to the surface.

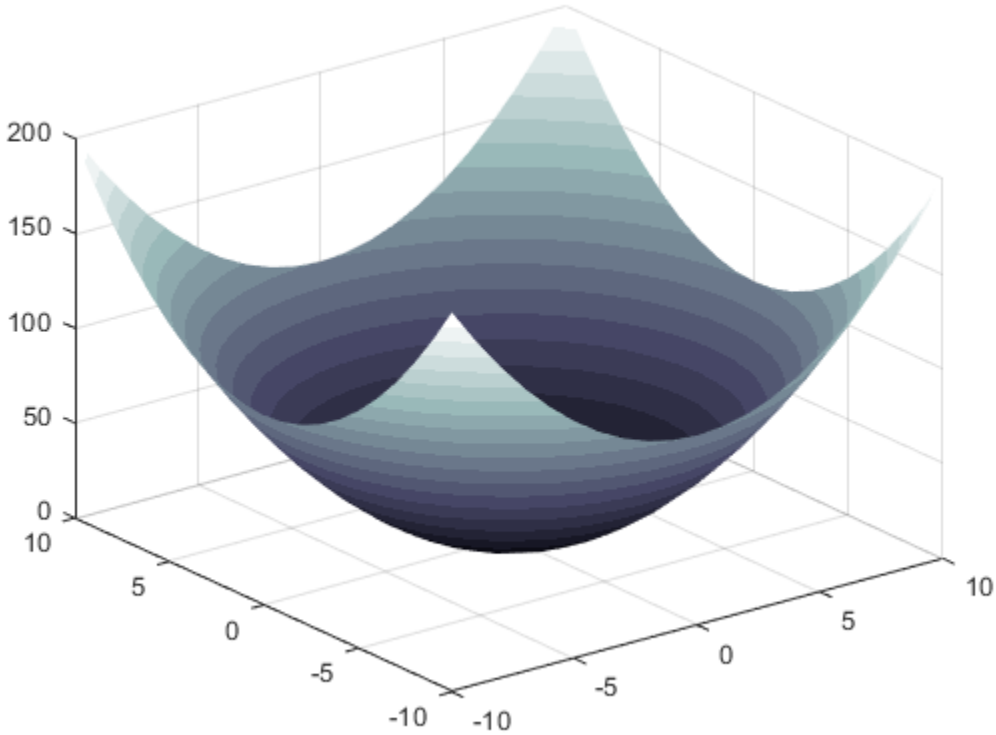
```
c = bone;
c = flipud(c);
colormap(c);
```



Downsample the Bone Colormap

Get a downsampled version of the bone colormap containing only twenty colors. Then display the contours of a paraboloid by applying the colormap and interpolated shading.

```
c = bone(20);
[X,Y] = meshgrid(-10:1:10);
Z = X.^2 + Y.^2;
surf(X,Y,Z);
colormap(c);
shading interp;
```



Input Arguments

`m` — Number of colors

256 (default) | scalar integer value

Number of colors, specified as a scalar integer value. The default value of `m` is equal to the length of the colormap for the current figure. If no figure exists, the default value is 256.

Data Types: `single` | `double`

Compatibility Considerations

Colormaps Have 256 Colors by Default

Behavior changed in R2019b

Starting in R2019b, colormaps have 256 colors by default.

In R2019a and previous releases, the default size is 64. If you have code that depends on a colormap having 64 colors, specify the number of colors when you set the colormap for the figure, axes, or chart. For example, `colormap(bone(64))` sets the figure's colormap to the 64-color `bone` colormap.

Alternatively, you can change the default colormap for all figures within your MATLAB session:

```
set(groot,'defaultFigureColormap',bone(64))
```

See Also

`colormap`

Topics

"Change Color Scheme Using a Colormap"

Introduced before R2006a

boundary

Boundary of a set of points in 2-D or 3-D

Syntax

```
k = boundary(x,y)
k = boundary(x,y,z)
k = boundary(P)
k = boundary( ___ ,s)
[k,v] = boundary( ___ )
```

Description

`k = boundary(x,y)` returns a vector of point indices representing a single conforming 2-D boundary around the points `(x,y)`. The points `(x(k),y(k))` form the boundary. Unlike the convex hull, the boundary can shrink towards the interior of the hull to envelop the points.

`k = boundary(x,y,z)` returns a triangulation representing a single conforming 3-D boundary around the points `(x,y,z)`. Each row of `k` is a triangle defined in terms of the point indices.

`k = boundary(P)` specifies points `(x,y)` or `(x,y,z)` in the columns of matrix `P`.

`k = boundary( ___ ,s)` specifies shrink factor `s` using any of the previous syntaxes. `s` is a scalar between `0` and `1`. Setting `s` to `0` gives the convex hull, and setting `s` to `1` gives a compact boundary that envelops the points. The default shrink factor is `0.5`.

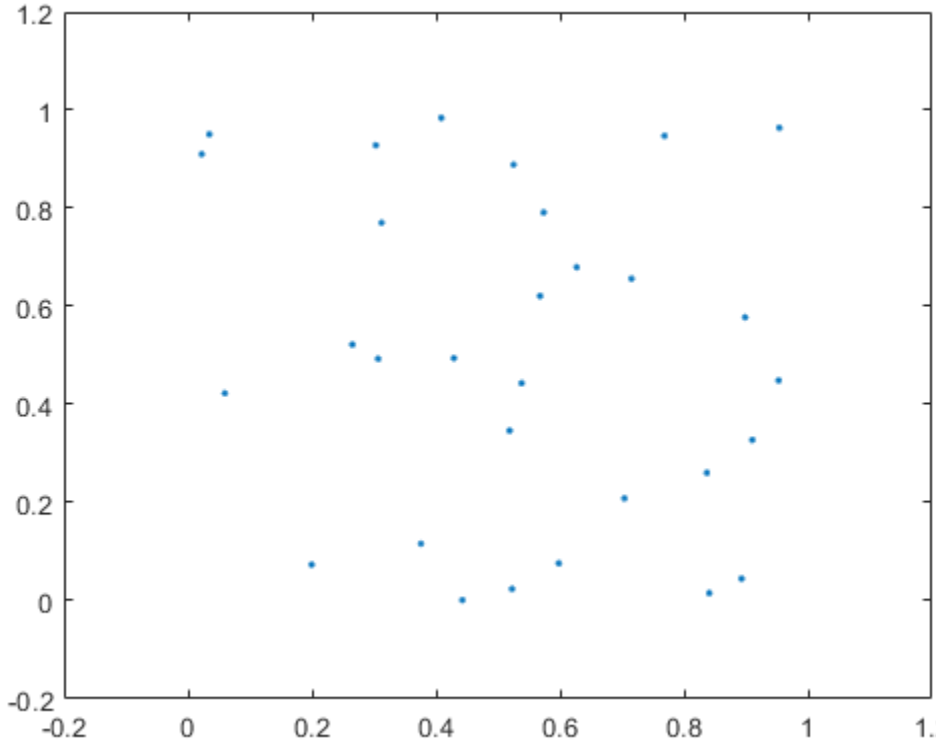
`[k,v] = boundary( ___ )` also returns a scalar `v`, which is the area (2-D) or volume (3-D) which boundary `k` encloses.

Examples

Boundary of 2-D Point Cloud

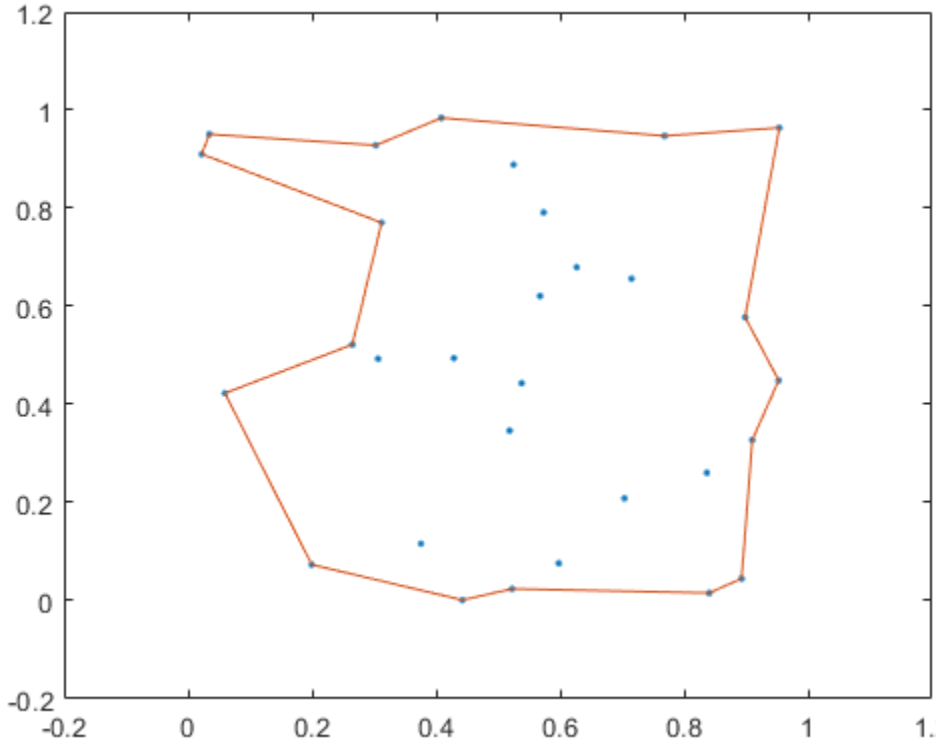
Create and plot a set of random 2-D points.

```
x = gallery('uniformdata',30,1,1);
y = gallery('uniformdata',30,1,10);
plot(x,y,'.')
xlim([-0.2 1.2])
ylim([-0.2 1.2])
```



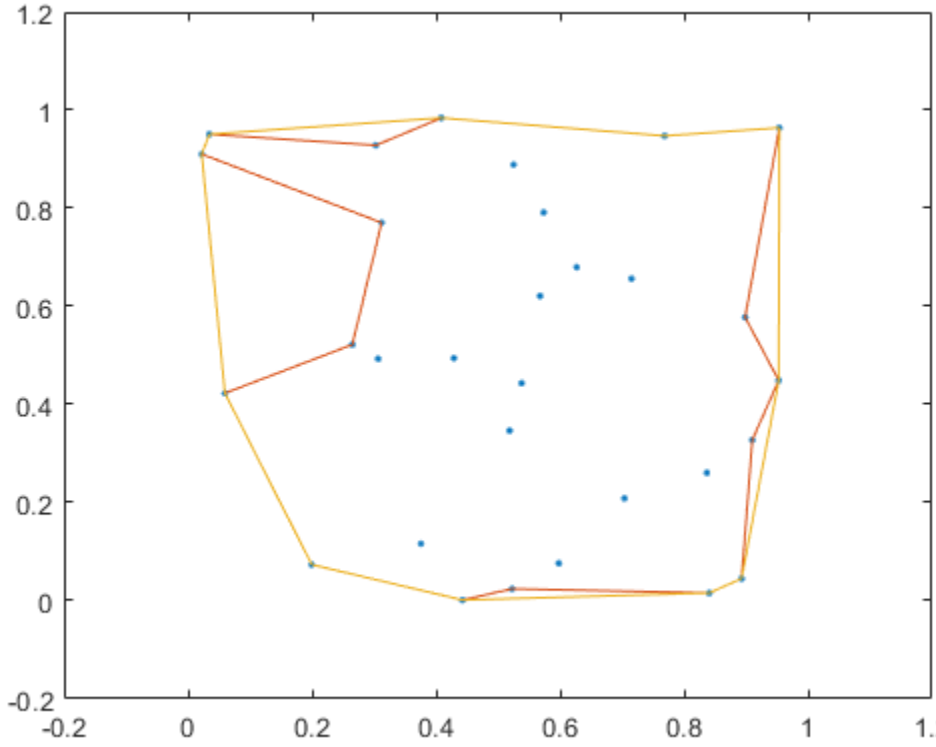
Compute a boundary around the points using the default shrink factor.

```
k = boundary(x,y);
hold on;
plot(x(k),y(k));
```



Create a new boundary around the points using a shrink factor of 0.1. The result is a less compact boundary enveloping the points.

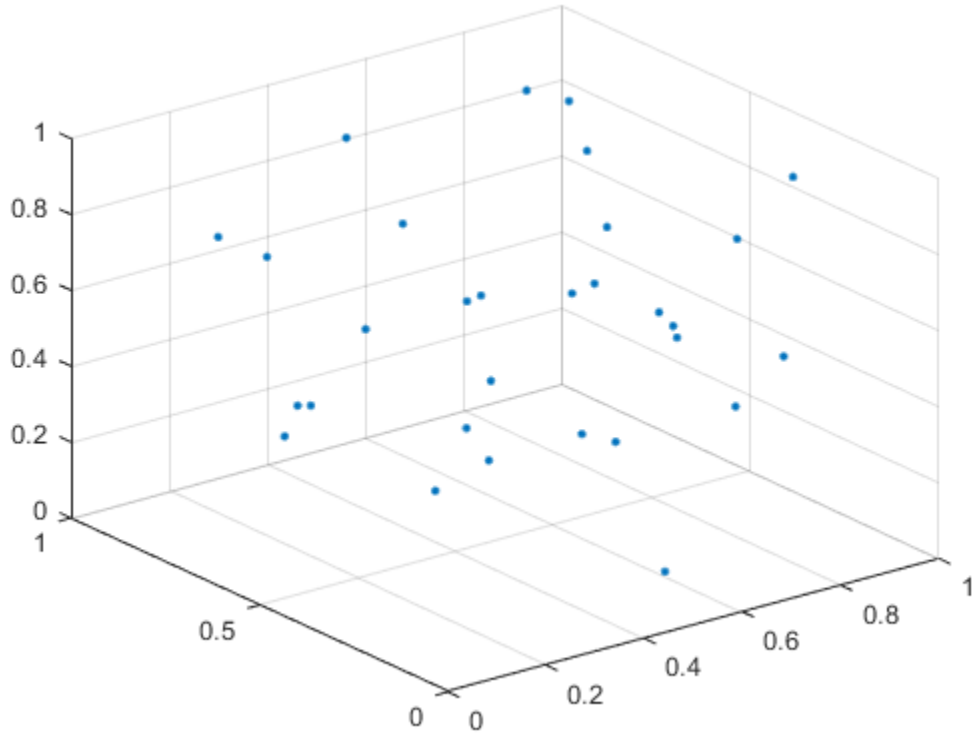
```
j = boundary(x,y,0.1);
hold on;
plot(x(j),y(j));
```



Boundary of 3-D Point Cloud

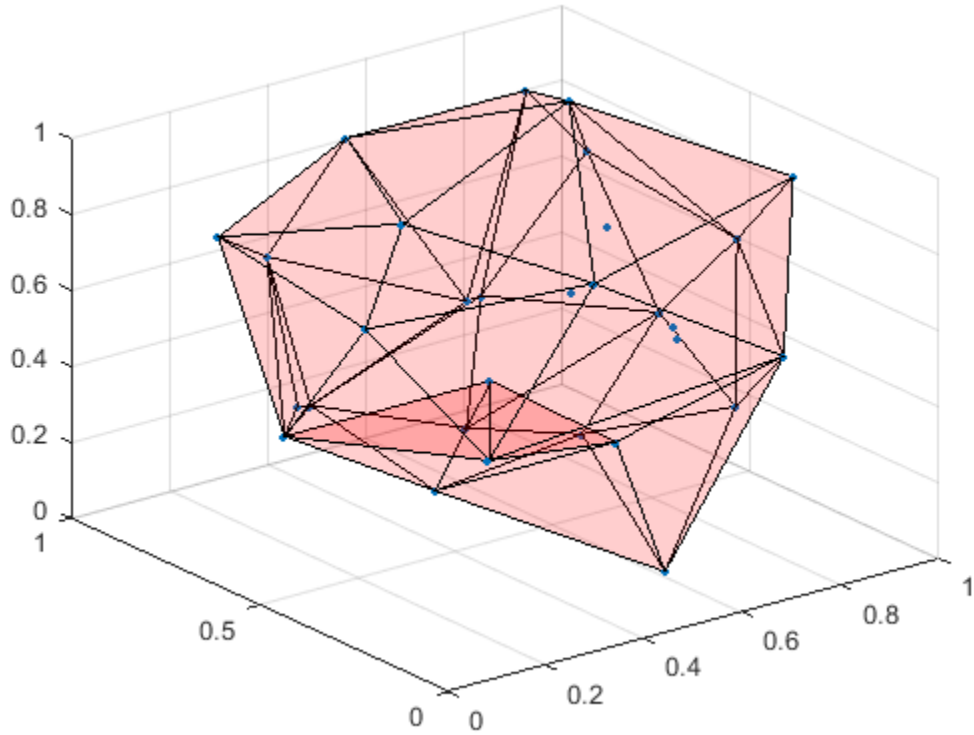
Create and plot a set of random 3-D points.

```
P = gallery('uniformdata',30,3,5);
plot3(P(:,1),P(:,2),P(:,3),'.','MarkerSize',10)
grid on
```



Plot the boundary using the default shrink factor.

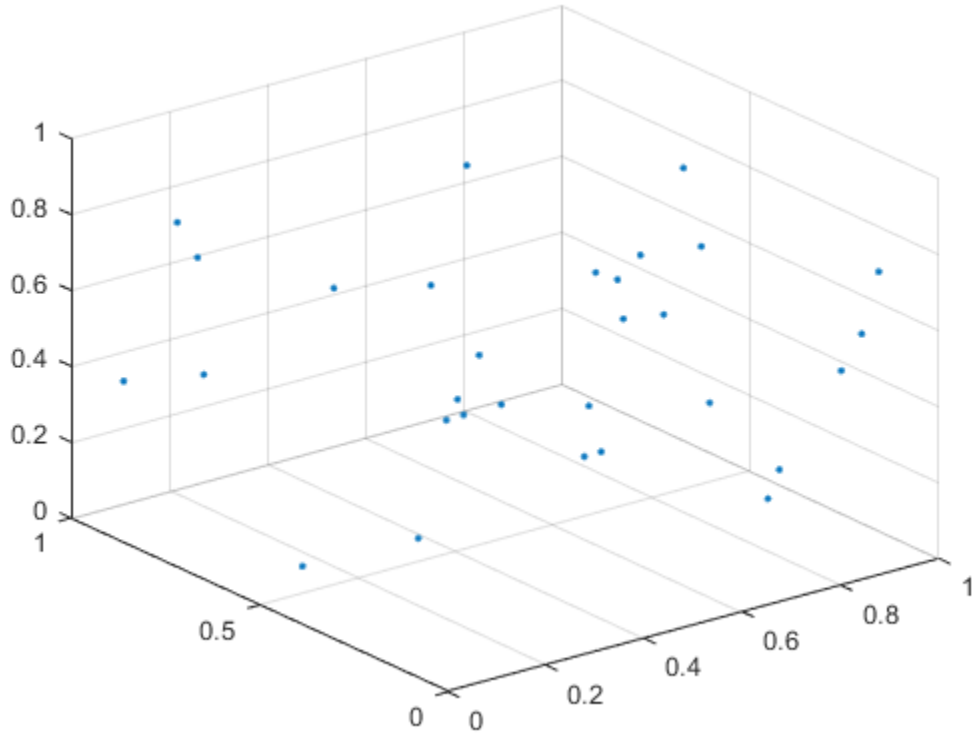
```
k = boundary(P);
hold on
trisurf(k,P(:,1),P(:,2),P(:,3),'Facecolor','red','FaceAlpha',0.1)
```



Shrink Factor Effect on 3-D Boundary

Create and plot a set of random 3-D points.

```
P = gallery('uniformdata',30,3,8);
plot3(P(:,1),P(:,2),P(:,3),'.')
grid on
```



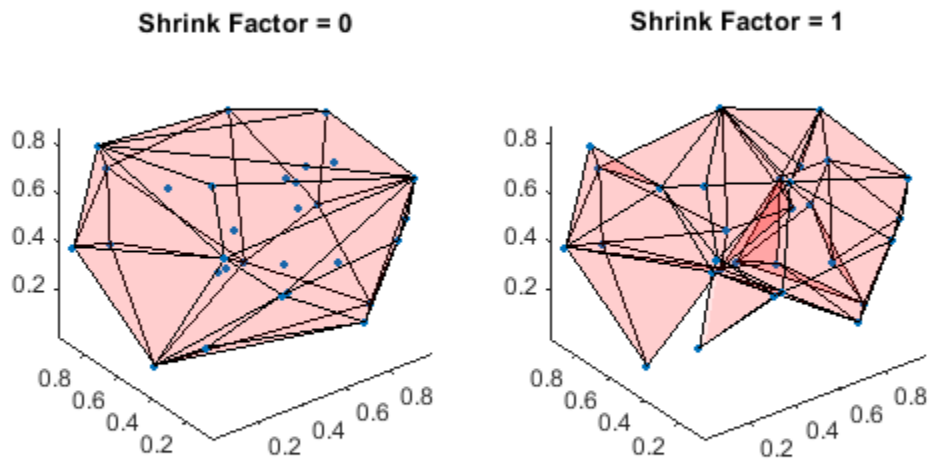
Compute two boundaries: one with a shrink factor of 0 and the other with a shrink factor of 1.

```
k = boundary(P,0);
j = boundary(P,1);
```

Compare the shrink factors by plotting the original points and the two boundaries side-by-side.

```
subplot(1,2,1);
plot3(P(:,1),P(:,2),P(:,3),'.','MarkerSize',10)
hold on
trisurf(k,P(:,1),P(:,2),P(:,3),'FaceColor','red','FaceAlpha',0.1)
axis equal
```

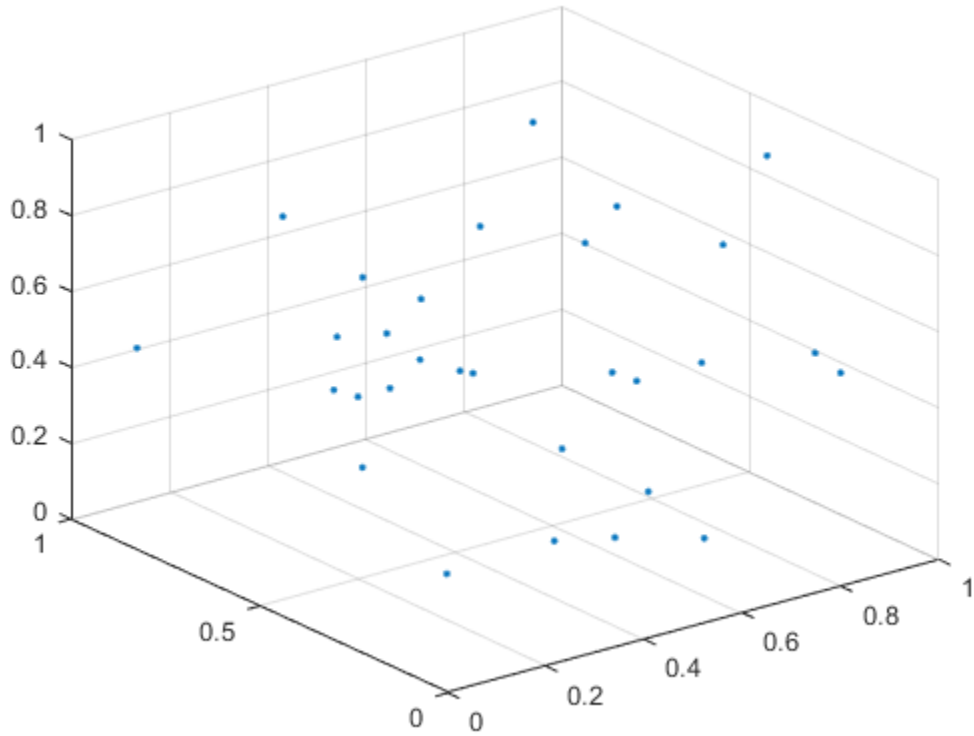
```
title('Shrink Factor = 0')

subplot(1,2,2);
plot3(P(:,1),P(:,2),P(:,3),'.','MarkerSize',10)
hold on
trisurf(j,P(:,1),P(:,2),P(:,3),'FaceColor','red','FaceAlpha',0.1)
axis equal
title('Shrink Factor = 1')
```



Volume of 3-D Boundary

Create and plot a set of random 3-D points.

```
P = gallery('uniformdata',30,3,1);
plot3(P(:,1),P(:,2),P(:,3),'.')
grid on
```



Use the `boundary` function to compute a boundary around the points, and find the volume of the resulting shape.

```
[~, vol] = boundary(P);
vol
```

```
vol = 0.3012
```

Input Arguments

x — x-coordinates of points
column vector

x-coordinates of points, specified as a column vector.

Data Types: `double`

y — y-coordinates of points
column vector

y-coordinates of points, specified as a column vector.

Data Types: `double`

z — z-coordinates of points
column vector

z-coordinates of points, specified as a column vector.

Data Types: `double`

P — Point coordinates
matrix with two columns | matrix with three columns

Point coordinates, specified as a matrix with two columns (for a 2-D alpha shape) or a matrix with three columns (for a 3-D alpha shape).

- For 2-D, the columns of `P` represent `x` and `y` coordinates, respectively.
- For 3-D, the columns of `P` represent `x`, `y`, and `z` coordinates, respectively.

Data Types: `double`

s — Shrink factor
`0.5` (default) | scalar in range `[0,1]`

Shrink factor, specified as a scalar in the range of `[0,1]`.

- `s = 0` corresponds to the convex hull of the points.
- `s = 1` corresponds to the tightest single-region boundary around the points.

The default shrink factor is `0.5`. Specify a larger or smaller shrink factor to tighten or loosen the boundary around the points, respectively.

Example: `k = boundary(x,y,0.76)` specifies a shrink factor of `0.76`, producing a tighter boundary than the default.

Output Arguments

k — Boundary point indices
vector | matrix

Boundary point indices, returned as a vector or matrix. `k` contains the indices of the input points that lie on the boundary.

- For 2-D problems, `k` is a column vector of point indices representing the sequence of points around the boundary, which is a polygon.
- For 3-D problems, `k` is a triangulation matrix of size `mtri`-by-`3`, where `mtri` is the number of triangular facets on the boundary. Each row of `k` defines a triangle in terms of the point indices, and the triangles collectively form a bounding polyhedron.

v — Area or volume enclosed by boundary
scalar

Area or volume enclosed by boundary, returned as a scalar.

- For 2-D problems, `v` is the area enclosed by boundary `k`.
- For 3-D problems, `v` is the volume enclosed by boundary `k`.

See Also

`alphaShape` | `convhull` | `delaunayTriangulation` | `triangulation` | `trisurf`

Introduced in R2014b

boundaryshape

Create `polyshape` from 2-D triangulation

Syntax

```
polyout = boundaryshape(TR)
[polyout,vertexID] = boundaryshape(TR)
```

Description

`polyout = boundaryshape(TR)` creates a `polyshape` object from the boundary of a 2-D triangulation. `TR` can be either a `triangulation` object or a `delaunayTriangulation` object.

Once you create the `polyshape` object `polyout`, you can analyze its properties or perform additional computations using `polyshape` functions. For example, you can access the vertices that define the boundary with the property `polyout.Vertices`, and you can plot the shape using the command `plot(polyout)`.

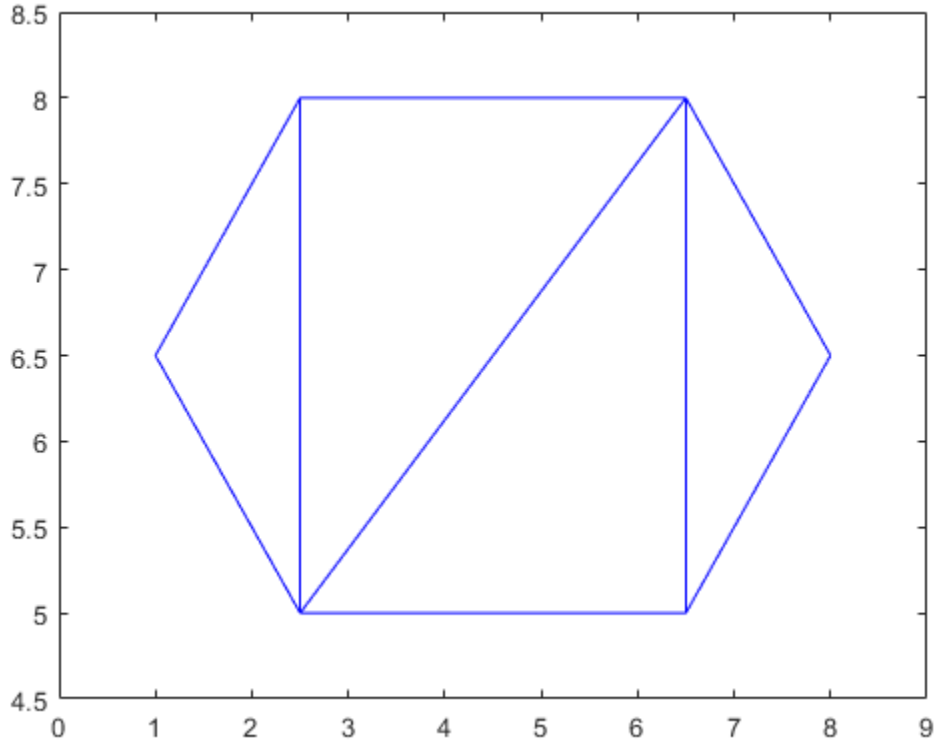
`[polyout,vertexID] = boundaryshape(TR)` also returns a vector `vertexID` that maps the vertices of the `polyshape` to the vertices of the triangulation.

Examples

Simple Boundary

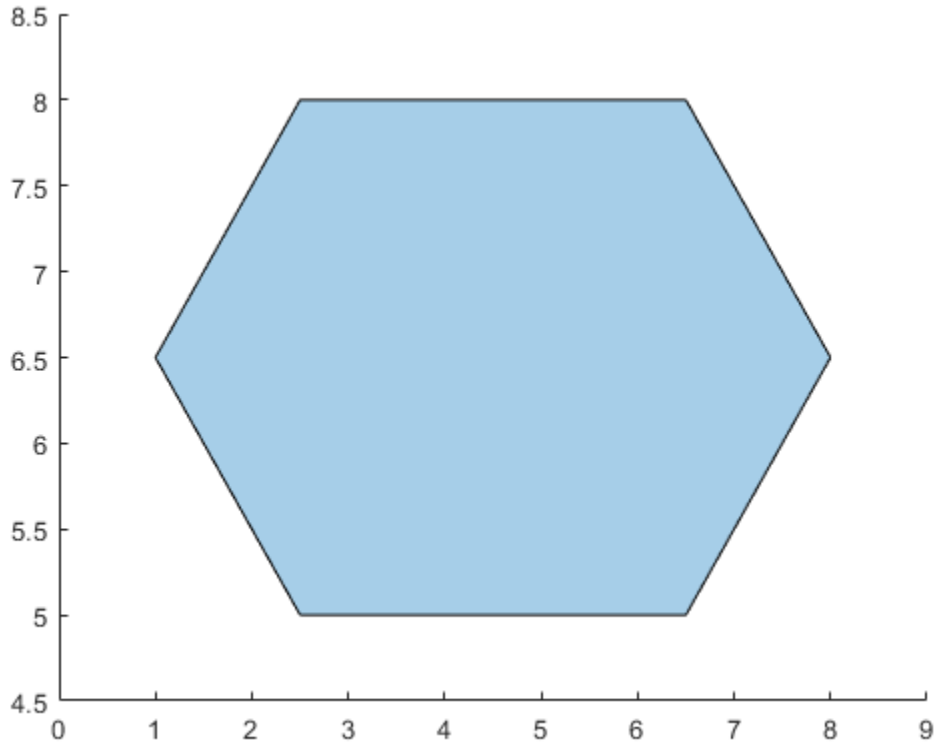
Create and plot a simple `triangulation` object.

```
P = [2.5 8.0; 6.5 8.0; 2.5 5.0; 6.5 5.0; 1.0 6.5; 8.0 6.5];
T = [5 3 1; 3 2 1; 3 4 2; 4 6 2];
TR = triangulation(T,P);
triplot(TR)
xlim([0 9])
ylim([4.5 8.5])
```



Create and plot a `polyshape` object whose boundary is equal to the boundary of the triangulation.

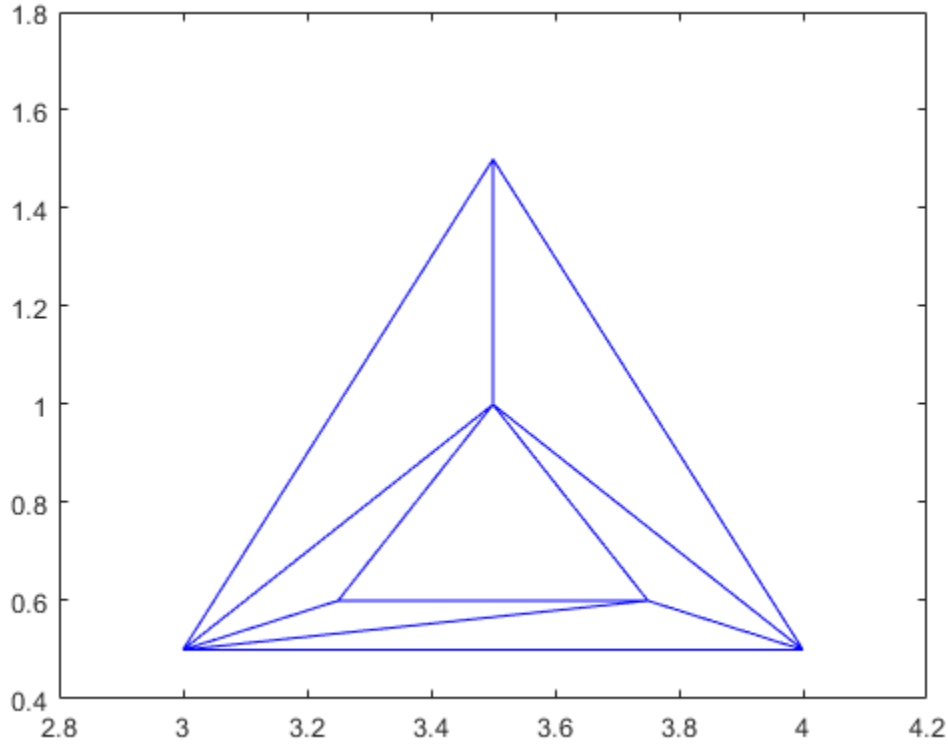
```
polyout = boundaryshape(TR);
plot(polyout)
```



Solid and Hole Boundaries

Create and plot a `triangulation` object.

```
T = [5 1 3; 6 1 4; 5 4 1; 2 6 3; 3 6 5; 2 1 6];
P = [3 0.5; 3.5 1.5; 4 0.5; 3.25 0.6; 3.75 0.6; 3.5 1];
TR = triangulation(T,P);
triplot(TR)
xlim([2.8 4.2])
ylim([0.4 1.8])
```

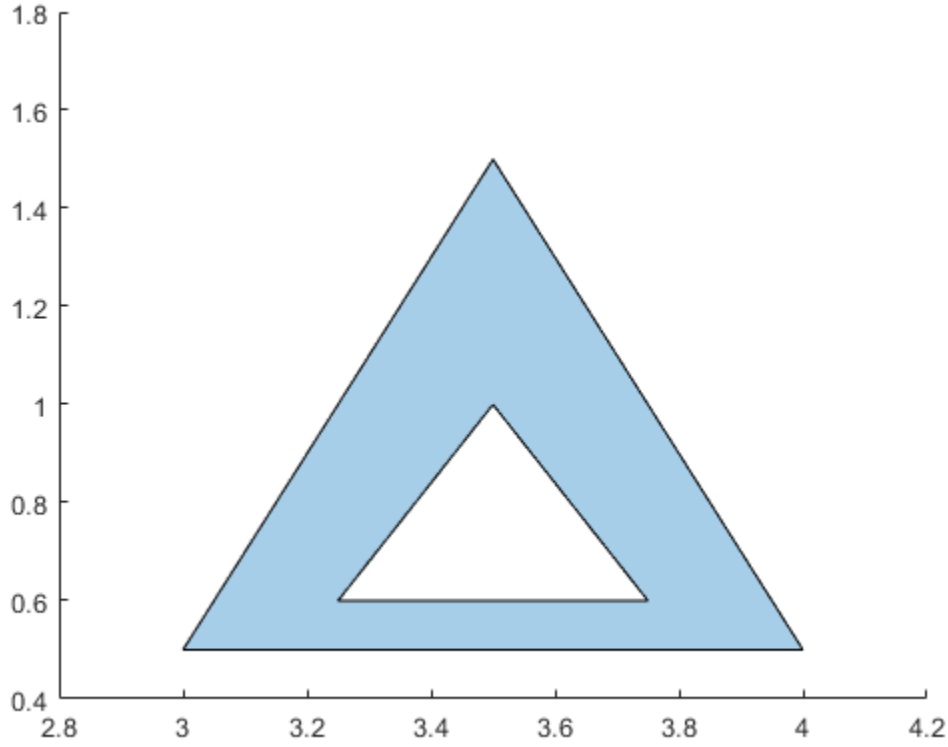


Create and plot a `polyshape` object whose boundaries are equal to the boundaries of the triangulation. Due to the inner and outer closed boundaries of the triangulation, `polyout` contains both a hole boundary and a solid boundary.

```
polyout = boundaryshape(TR)
```

```
polyout 
  polyshape with properties:

      Vertices: [7x2 double]
    NumRegions: 1
      NumHoles: 1
```

```
plot(polyout)
```



Input Arguments

TR — Triangulation
2-D `triangulation` object | 2-D `delaunayTriangulation` object

Triangulation, specified as a 2-D `triangulation` object or a 2-D `delaunayTriangulation` object.

Output Arguments

`polyout` — Output `polyshape`
`polyshape` object

Output `polyshape`, returned as a `polyshape` object.

`vertexID` — Vertex identification
vector

Vertex identification, returned as a vector of integers. Each element in `vertexID` corresponds to the row number of the triangulation vertex in the `Points` property. The length of `vertexID` is equal to the number of rows in the `Vertices` property of the `polyshape` object.

Data Types: `double`

See Also

`delaunayTriangulation` | `polyshape` | `triangulation`

Introduced in R2018b

bounds

Smallest and largest elements

Syntax

```
[S,L] = bounds(A)
[S,L] = bounds(A,'all')
[S,L] = bounds(A,dim)
[S,L] = bounds(A,vecdim)
[S,L] = bounds( ___ ,nanflag)
```

Description

`[S,L] = bounds(A)` returns the smallest element `S` and largest element `L` of an array. `S` is equivalent to `min(A)` and `L` is equivalent to `max(A)`.

`[S,L] = bounds(A,'all')` computes the smallest and largest values over all elements of `A`. This syntax is valid for MATLAB versions R2018b and later.

`[S,L] = bounds(A,dim)` operates along the dimension `dim` of `A`. For example, if `A` is a matrix, then `bounds(A,2)` returns column vectors `S` and `L` containing the smallest and largest elements of each row.

`[S,L] = bounds(A,vecdim)` computes the smallest and largest values based on the dimensions specified in the vector `vecdim`. For example, if `A` is a matrix, then `bounds(A, [1 2])` returns the smallest and largest values over all elements in `A`, since every element of a matrix is contained in the array slice defined by dimensions 1 and 2.

`[S,L] = bounds( ___ ,nanflag)` specifies whether to include or omit `NaN` values when determining the smallest and largest elements. `bounds(A,'omitnan')` ignores `NaN` values. If any element of `A` is `NaN`, then `bounds(A,'includenan')` returns `NaN` for both `S` and `L`. The default behavior is `'omitnan'`.

Examples

Smallest and Largest Elements of Vector

Simultaneously compute the smallest and largest values of a vector.

```
A = [2 4 -1 10 6 3 0 -16];
[S,L] = bounds(A)
```

```
S = -16

L = 10
```

Smallest and Largest Elements of Matrix Rows

Compute the smallest and largest elements of each row of a matrix.

```
A = magic(4)
```

```
A = 4×4

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

```
[S,L] = bounds(A,2)
```

```
S = 4×1

     2
     5
     6
     1


L = 4×1

    16
    11
    12
    15
```

Bounds of Array Page

Create a 3-D array and compute the smallest and largest values in each page of data (rows and columns).

```
A(:,:,1) = [2 4; -2 1];
A(:,:,2) = [9 13; -5 7];
A(:,:,3) = [4 4; 8 -3];
[S1,L1] = bounds(A,[1 2]);
S1
```

```
S1 =
S1(:,:,1) =

    -2


S1(:,:,2) =

    -5


S1(:,:,3) =

    -3
```

```
L1
```

```
L1 =
L1(:,:,1) =

     4


L1(:,:,2) =

    13


L1(:,:,3) =

     8
```

Starting in R2018b, to compute the bounds over all dimensions of an array, you can either specify each dimension in the vector dimension argument, or use the `'all'` option.

```
[S2,L2] = bounds(A,[1 2 3])
```

```
S2 = -5
```

```
L2 = 13
```

```
[Sall,Lall] = bounds(A,'all')
```

```
Sall = -5
```

```
Lall = 13
```

Vector with NaN Values

Include and ignore NaN elements of a vector when computing its smallest and largest values.

Ignore `NaN` values when computing the largest and smallest values of a vector, which is the default.

```
A = [2 NaN 6 -5 0 NaN 10];
[S,L] = bounds(A)
```

```
S = -5
```

```
L = 10
```

Use the `'includenan'` option to include `NaN` values, which causes `bounds` to return `NaN` for both the smallest and largest values of `A`.

```
[S,L] = bounds(A,'includenan')
```

```
S = NaN
```

```
L = NaN
```

Input Arguments

A — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

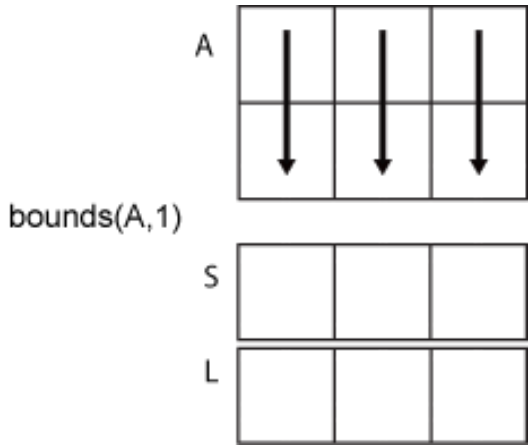
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `categorical` | `datetime` | `duration`
Complex Number Support: Yes

dim — Dimension to operate along
positive integer scalar

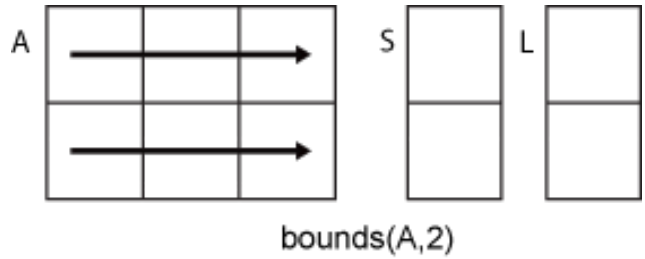
Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a matrix `A`.

- `bounds(A,1)` computes the smallest and largest values of each column.



- `bounds(A,2)` computes the smallest and largest values of each row.

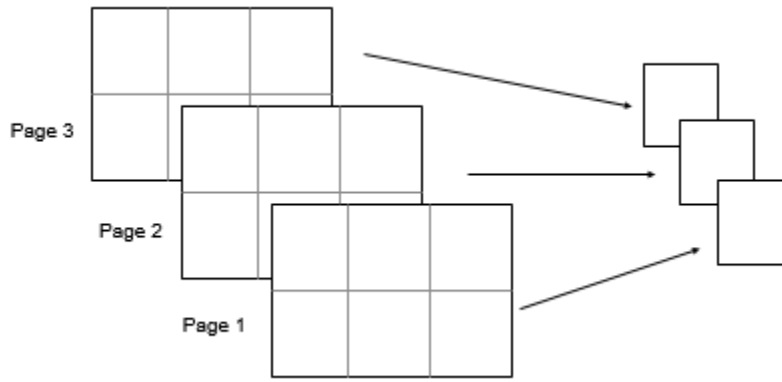


Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`vecdim` — Vector of dimensions
vector of positive integers

Vector of dimensions, specified as a vector of positive integers. Each element represents a dimension of the input array. The lengths of the output in the specified operating dimensions are 1, while the others remain the same.

Consider a 2-by-3-by-3 input array, `A`. Then `[S,L] = bounds(A,[1 2])` returns a 1-by-1-by-3 array for both `S` and `L`. The elements of `S` and `L` are the smallest and largest values in the corresponding page of `A`, respectively.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

nanflag — NaN condition
`'omitnan'` (default) | `'includenan'`

`NaN` condition, specified as one of these values:

- `'omitnan'` — Ignore all `NaN` values in the input. If the input contains only `NaN` values, then `bounds` returns `NaN` for both `S` and `L`.
- `'includenan'` — Include `NaN` values. If any element of the input is `NaN`, then `bounds` returns `NaN` for both `S` and `L`.

Output Arguments

S — Smallest element
vector | matrix | multidimensional array

Smallest element, specified as a vector, matrix, or multidimensional array.

L — Largest element
vector | matrix | multidimensional array

Largest element, specified as a vector, matrix, or multidimensional array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`max` | `min`

Introduced in R2017a

box

Display axes outline

Syntax

```
box on
box off
box
```

```
box(ax, ___ )
```

Description

`box on` displays the box outline around the current axes by setting their `Box` property to `'on'`. This property value is the default for `GeographicAxes` objects.

`box off` removes the box outline around the current axes by setting their `Box` property to `'off'`. This property value is the default for `Axes` and `PolarAxes` objects.

`box` toggles the display of the box outline.

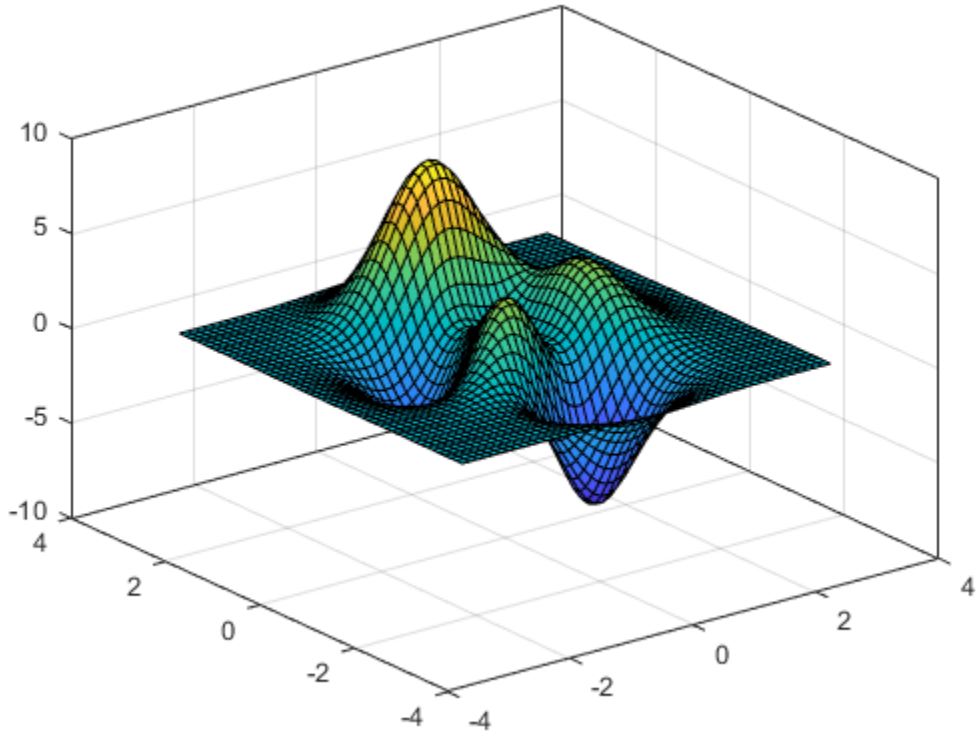
`box(ax, ___ )` uses the axes specified by `ax` instead of the current axes. Specify the axes as the first input argument for any of the previous syntaxes. Use single quotes around the `'on'` and `'off'` inputs, such as `box(ax,'on')`.

Examples

Display Box Outline Around Axes

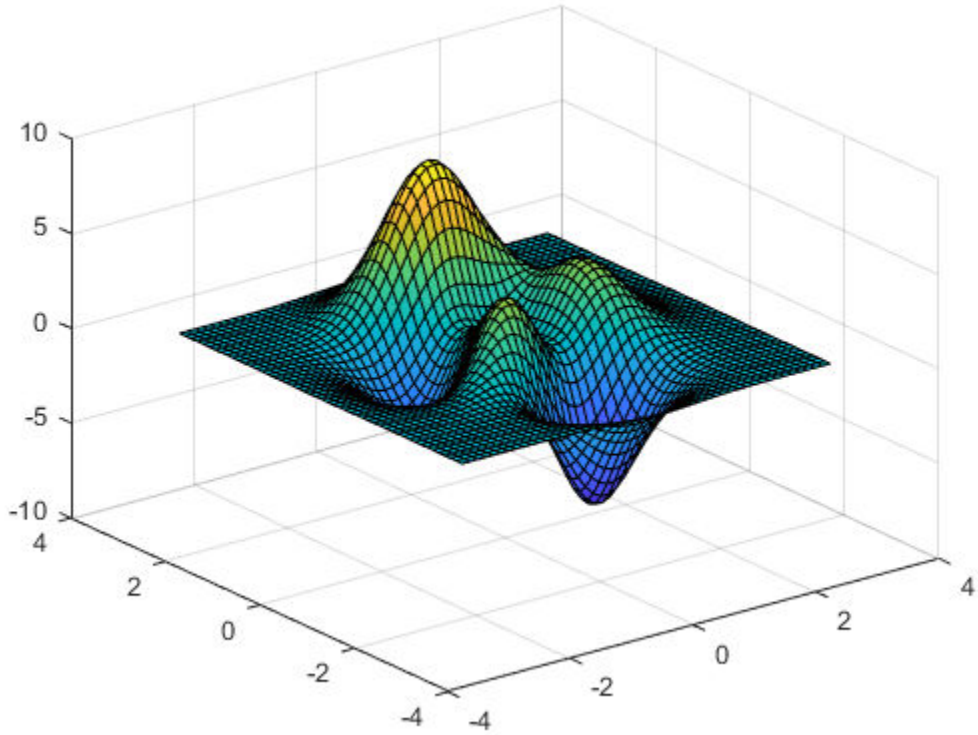
Plot a surface and display the box outline around the axes.

```
[X,Y,Z] = peaks;
surf(X,Y,Z)
box on
```



Turn off the display of the box outline.

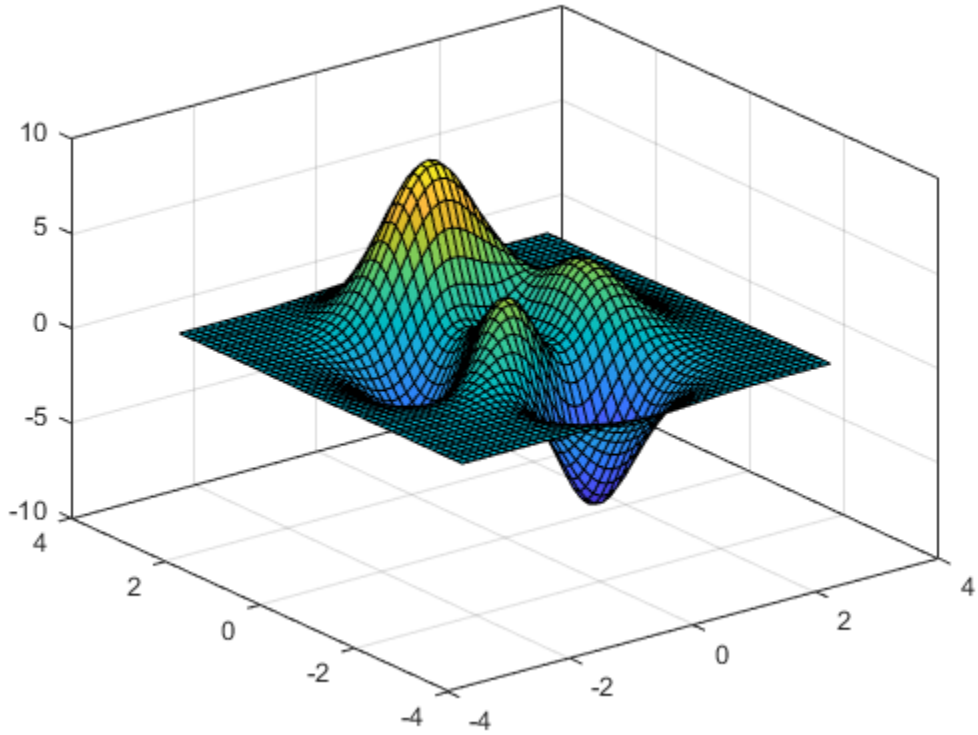
```
box off
```



Change Style of Box Outline

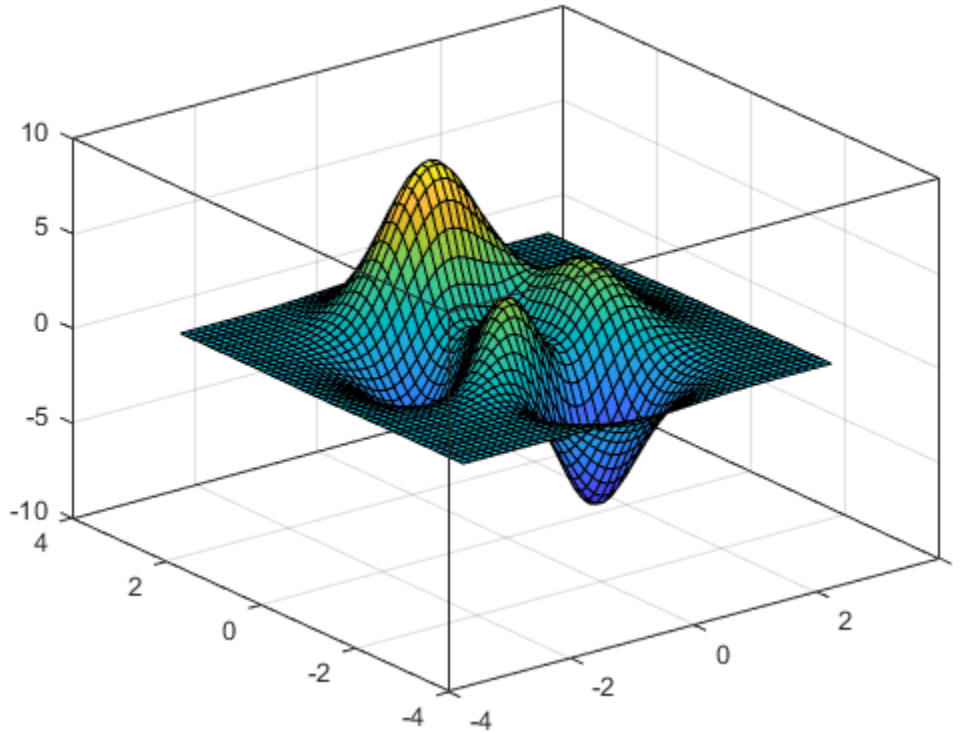
First, plot a surface and display the box outline around the axes. By default, the outline appears around the back planes of the axes because the `BoxStyle` property of the axes is set to `'back'`.

```
[X,Y,Z] = peaks;
surf(X,Y,Z)
box on
```



Next, display the outline around the entire axes by setting the `BoxStyle` property to `'full'`. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead.

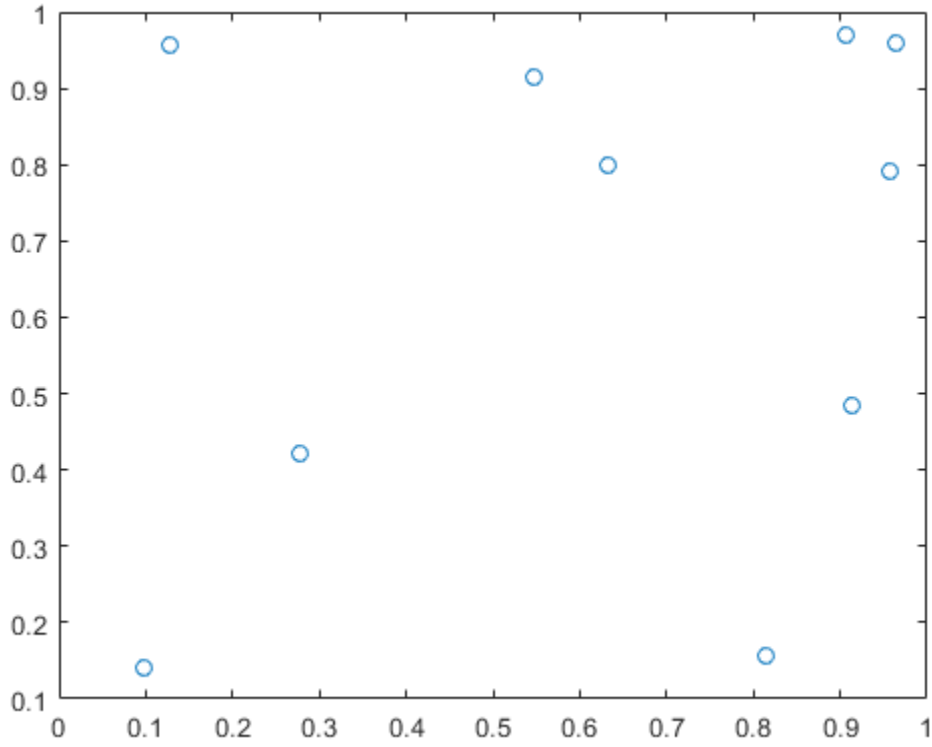
```
ax = gca;
ax.BoxStyle = 'full';
```



Change Color of Box Outline

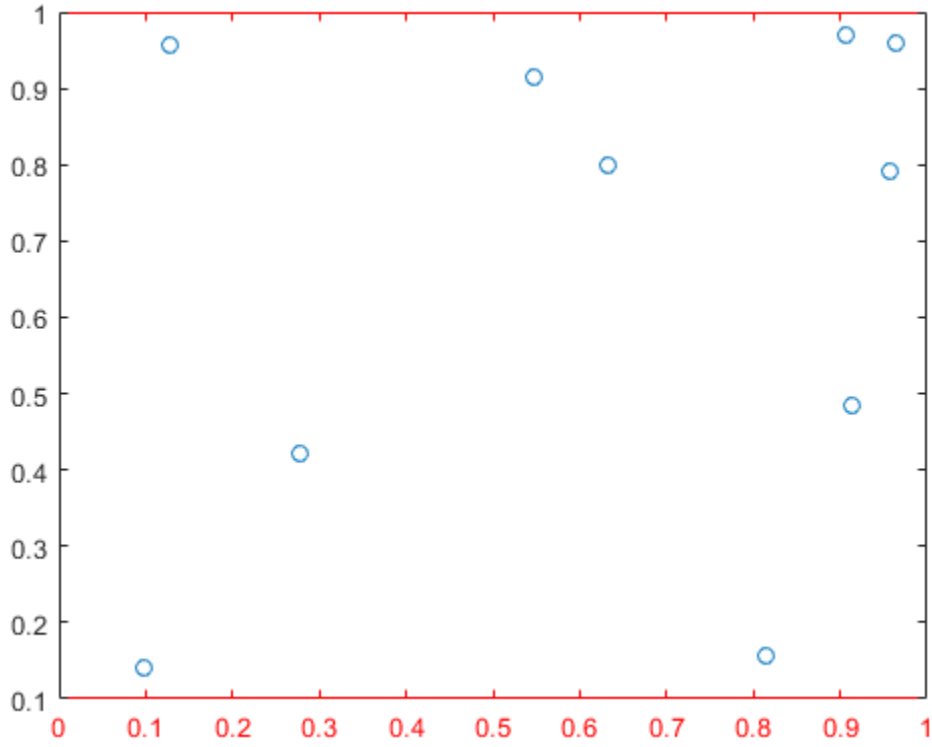
Create a scatter plot and display the box outline around the axes.

```
x = rand(10,1);
y = rand(10,1);
scatter(x,y)
box on
```



Change the color of the box outline in the *x*-axis direction by setting the `XColor` property of the axes. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead.

```
ax = gca;
ax.XColor = 'red';
```

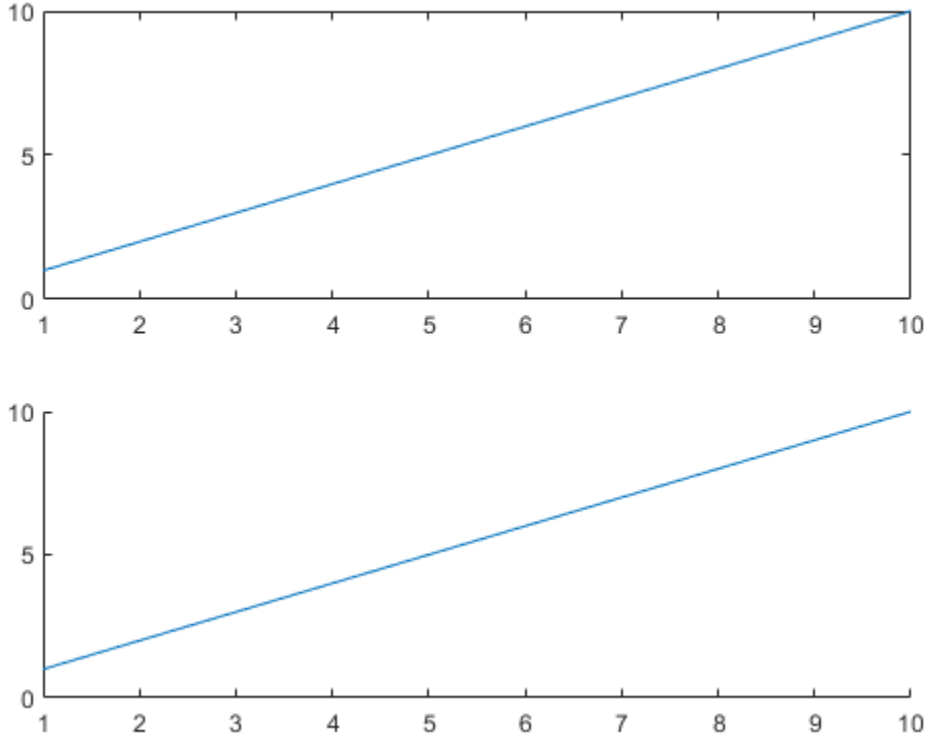


Turn Off Box Outline for Specific Axes

Create a figure with two subplots and assign the `Axes` objects to the variables `ax1` and `ax2`. Plot a line in each subplot. Remove the box outline around the lower subplot by specifying `ax2` as the first input argument to `box`.

```
ax1 = subplot(2,1,1);
plot(ax1,1:10)

ax2 = subplot(2,1,2);
```

```
plot(ax2,1:10)
box(ax2,'off')
```



Input Arguments

ax — Target axes

`Axes` object | `GeographicAxes` object | `PolarAxes` object

Target axes, specified as an `Axes` object, a `GeographicAxes` object, or a `PolarAxes` object. If you do not specify the axes, then `box` affects the current axes.

Tips

- Some Cartesian axes properties affect the appearance of the box outline. This table lists a subset of axes properties related to the box outline.

Axes Property	Description
`Box`	Display of box outline
`BoxStyle`	Style of box outline
`XColor`, `YColor`, `ZColor`	Box outline color in the *x*-axis, *y*-axis, and *z*-axis directions
`LineWidth`	Width of box outline, tick marks, and grid lines

- Some polar axes properties affect the appearance of the outline around the polar axes. If you are working with polar axes, then the `box` command controls the outline display when the *theta*-axis limits do not span 360 degrees. This table lists a subset of polar axes properties related to the outline.

PolarAxes Property	Description
`Box`	Display of full outline
`RColor`, `ThetaColor`	Outline color
`LineWidth`	Width of outline, tick marks, and grid lines

- Some geographic axes properties affect the appearance of the box outline. This table lists a subset of geographic axes properties related to the box outline.

GeographicAxes Property	Description
`Box`	Display of box outline
`AxisColor`	Color of outline, tick values, and labels
`LineWidth`	Width of box outline, tick marks, and grid lines

Algorithms

The `box` function sets the `Box` property of the `Axes`, `PolarAxes`, or `GeographicAxes` object to either `'on'` or `'off'`.

See Also

Functions
`axis` | `gca` | `grid`

Properties
Axes | GeographicAxes | PolarAxes

Introduced before R2006a

break

Terminate execution of for or while loop

Syntax

```
break
```

Description

`break` terminates the execution of a `for` or `while` loop. Statements in the loop after the `break` statement do not execute.

In nested loops, `break` exits only from the loop in which it occurs. Control passes to the statement that follows the `end` of that loop.

Examples

Exit Loop Before Expression Is False

Sum a sequence of random numbers until the next random number is greater than an upper limit. Then, exit the loop using a `break` statement.

```
limit = 0.8;
s = 0;

while 1
    tmp = rand;
    if tmp > limit
        break
    end
    s = s + tmp;
end
```

Tips

- The `break` statement exits a `for` or `while` loop completely. To skip the rest of the instructions in the loop and begin the next iteration, use a `continue` statement.
- `break` is not defined outside a `for` or `while` loop. To exit a function, use `return`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`continue` | `end` | `for` | `return` | `while`

Introduced before R2006a

brighten

Brighten or darken colormap

Syntax

```
brighten(beta)
brighten(map,beta)
newmap = brighten( ___ )

brighten(f,beta)
```

Description

`brighten(beta)` shifts the intensities of all colors in the current colormap in the same direction. The colors brighten when `beta` is between `0` and `1`, and they darken when `beta` is between `-1` and `0`. The magnitude of the change is proportional to the magnitude of `beta`. Use this syntax to adjust colors for all graphics objects in the current figure that use a colormap.

`brighten(map,beta)` shifts the intensities of the colormap specified as `map`.

`newmap = brighten( ___ )` returns the adjusted colormap for any of the input argument combinations in the previous syntaxes. When you specify the output argument, the current figure is not affected.

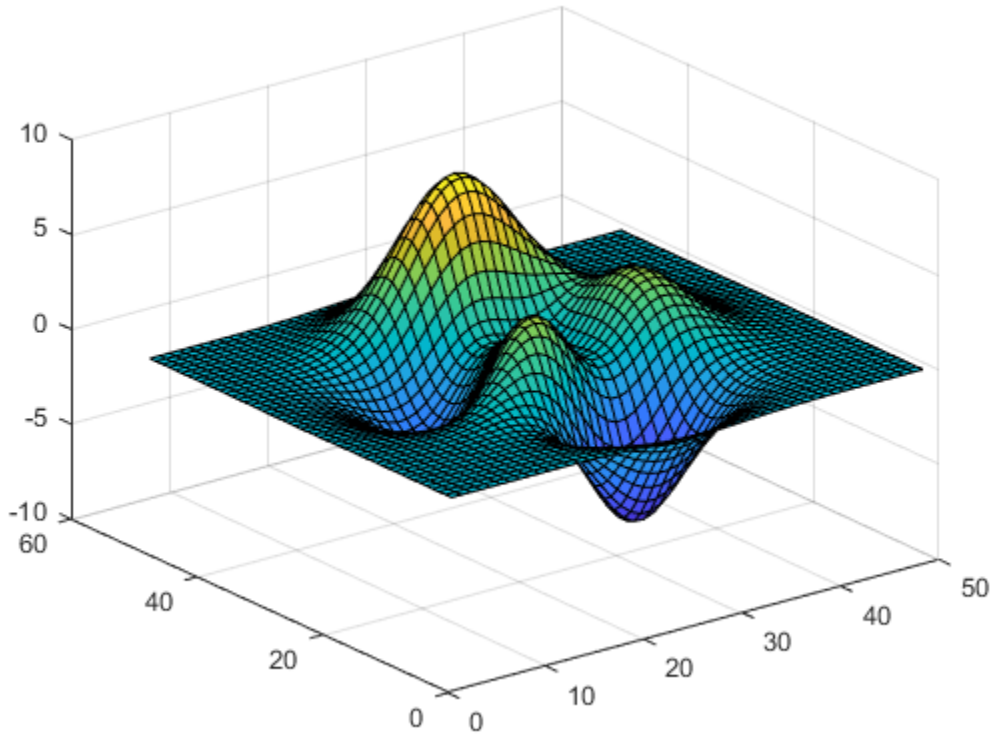
`brighten(f,beta)` shifts the intensities of the colormap assigned to figure `f`. The colors of other graphics objects are affected, such as the axes, axes labels, and ticks.

Examples

Brighten Current Colormap

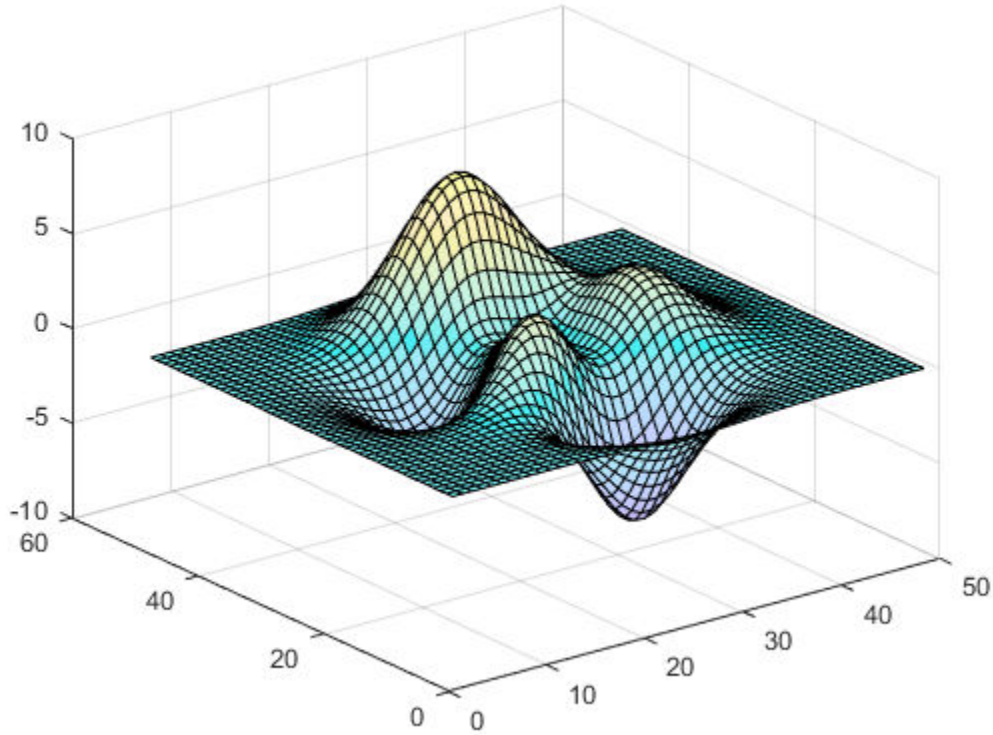
Create a surface plot with the default colormap.

```
surf(peaks);
```



Brighten the colormap for current figure.

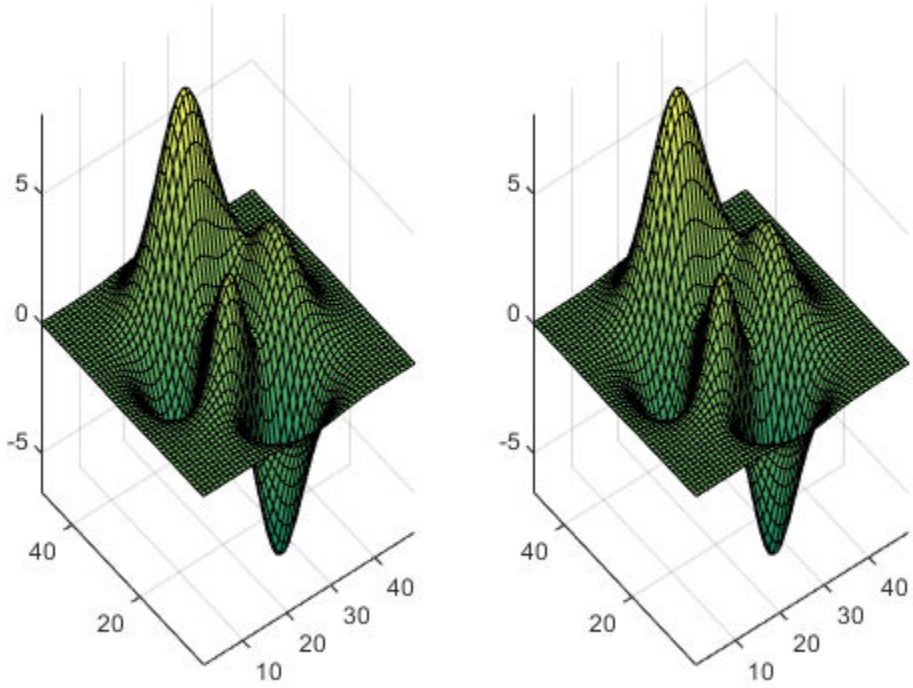
```
brighten(.8);
```



Darken a Colormap in a Subplot

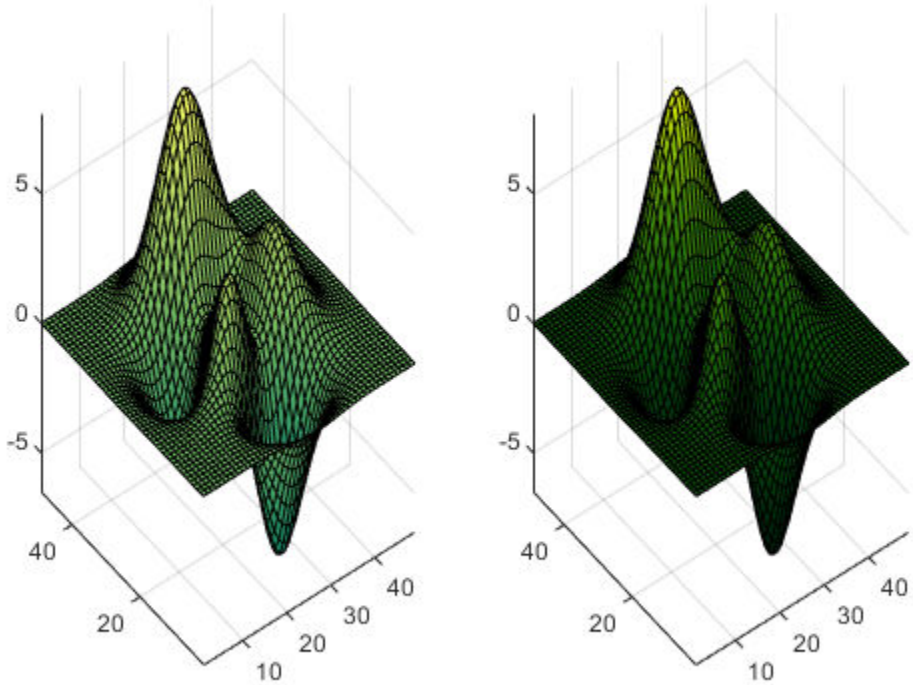
Display two surface plots using the `subplot` function and the `summer` colormap.

```
ax1 = subplot(1,2,1);
surf(ax1,peaks);
axis tight
ax2 = subplot(1,2,2);
surf(ax2,peaks);
axis tight
colormap(summer)
```



Darken the `summer` colormap and apply it to the second surface.

```
newmap = brighten(summer,-.7);
colormap(ax2,newmap)
```



Input Arguments

beta — Brightness adjustment
numeric scalar value

Brightness adjustment parameter, specified as a numeric scalar value. The `brighten` function raises each value in the colormap to the power of γ, which is defined as:

$$\gamma = \begin{cases} 1 - \beta, & \beta > 0 \\ \dfrac{1}{1 + \beta}, & \beta \le 0 \end{cases}$$

Data Types: `single` | `double`

`f` — Figure
`Figure` object

Figure to adjust, specified as a `Figure` object. When you specify this argument, the colors of other graphics objects within the figure are affected in addition to the objects that use the colormap. For example, color of the axes, axes labels, and ticks changes.

`map` — Colormap
three-column matrix of RGB triplets

Colormap to adjust, specified as a three-column matrix of RGB triplets. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of a color. The intensities must be in the range [0, 1]. For example, here is a colormap that contains five colors:

```
map = [0.2 0.1 0.5
    0.1 0.5 0.8
    0.2 0.7 0.6
    0.8 0.7 0.3
    0.9 1 0];
```

This table lists the RGB triplet values for common colors.

Color	RGB Triplet
yellow	`[1 1 0]`
magenta	`[1 0 1]`
cyan	`[0 1 1]`
red	`[1 0 0]`
green	`[0 1 0]`
blue	`[0 0 1]`
white	`[1 1 1]`
black	`[0 0 0]`

Alternatively, you can create the matrix by calling one of the predefined colormap functions. Call the function as an input argument to the `brighten` function. For example, this command brightens the `parula` colormap.

```
brighten(parula,.8)
```

Data Types: `double` | `single`

Output Arguments

`newmap` — Adjusted colormap
three-column matrix of RGB triplets

Adjusted colormap, returned as a three-column matrix of RGB triplets.

See Also

`colormap` | `rgbplot`

Introduced before R2006a

brush

Interactively mark data values in a chart

Description

Data brushing allows you to mark data values in a chart using a color of your choice. You can mark individual data values, or select many values to mark using a selection rectangle. Once you have values marked, you can delete them or save them to a variable in the workspace. Brush marks will persist until you remove them, even if you pan, zoom, add data tips, or edit the plot.

Most charts support data brushing, including line, bar, area, histogram, and surface charts. Charts that support brushing typically display the brush icon in the axes toolbar.

Creation

Syntax

```
brush on
brush off
brush
brush(color)
brush(fig, ___ )
```

```
b = brush( ___ )
```

Description

`brush on` turns on brushing in the current figure. With brushing on, mark a single data value by clicking on it. Mark multiple data values by dragging a selection rectangle around them. To add additional marks or remove unwanted marks, hold the **Shift** key while brushing the values.

`brush off` turns off brushing. If there are existing brush marks on the chart, this command will not remove them.

`brush` toggles the brushing mode between `'on'` and `'off'`.

`brush(color)` sets the color for subsequent brush marks. Setting the brush color does not affect current marks unless you brush data, change the brush color, and then add or remove marks using the **Shift** key. In this case, all marks update to the new color.

`brush(fig, ___ )` sets brush mode or color options for the specified figure. For example, `brush(fig,'on')`.

`b = brush( ___ )` creates a `brush` object. This syntax is useful for controlling the brush mode, color, and callback properties.

Input Arguments

`fig` — Target figure

`Figure` object

Target figure, specified as a `Figure` object. If you do not specify the figure, then brushing affects the current figure.

`color` — Brush mark color

`[1 0 0]` or red (default) | RGB triplet | color name

Brush mark color, specified as an RGB triplet or a color name. To create brush marks of a certain color, set the color property before brushing your data values. Setting the brush color does not affect current marks unless you brush data, change the brush color, and then add or remove marks using the **Shift** key. In this case, all marks update to the new color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and their equivalent RGB triplet values.

Color Name	Short Name	RGB Triplet	Appearance
`'red'`	`'r'`	`[1 0 0]`	

Color Name	Short Name	RGB Triplet	Appearance
'green'	'g'	[0 1 0]	
'blue'	'b'	[0 0 1]	
'cyan'	'c'	[0 1 1]	
'magenta'	'm'	[1 0 1]	
'yellow'	'y'	[1 1 0]	
'black'	'k'	[0 0 0]	
'white'	'w'	[1 1 1]	

Here are the RGB triplets for the default colors MATLAB uses in many types of plots.

RGB Triplet	Appearance
[0 0.4470 0.7410]	
[0.8500 0.3250 0.0980]	
[0.9290 0.6940 0.1250]	
[0.4940 0.1840 0.5560]	
[0.4660 0.6740 0.1880]	
[0.3010 0.7450 0.9330]	
[0.6350 0.0780 0.1840]	

Properties

`Color` — Brush mark color

`[1 0 0]` or red (default) | RGB triplet | color name

Brush mark color, specified as an RGB triplet or a color name. To create brush marks of a certain color, set the color property before brushing your data values. Setting the brush color does not affect current marks unless you brush data, change the brush color, and then add or remove marks using the **Shift** key. In this case, all marks update to the new color.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by

name. This table lists the long and short color name options and their equivalent RGB triplet values.

Color Name	Short Name	RGB Triplet	Appearance
'red'	'r'	[1 0 0]	
'green'	'g'	[0 1 0]	
'blue'	'b'	[0 0 1]	
'cyan'	'c'	[0 1 1]	
'magenta'	'm'	[1 0 1]	
'yellow'	'y'	[1 1 0]	
'black'	'k'	[0 0 0]	
'white'	'w'	[1 1 1]	

Here are the RGB triplets for the default colors MATLAB uses in many types of plots.

RGB Triplet	Appearance
[0 0.4470 0.7410]	
[0.8500 0.3250 0.0980]	
[0.9290 0.6940 0.1250]	
[0.4940 0.1840 0.5560]	
[0.4660 0.6740 0.1880]	
[0.3010 0.7450 0.9330]	
[0.6350 0.0780 0.1840]	

`Enable` — Brushing mode
`'off'` (default) | `'on'`

Brushing mode, specified as `'off'` or `'on'`.

`ActionPreCallback` — Function to execute before brushing
[ ] (default) | function handle | cell array | character vector

Function to execute before brushing, specified as one of these values:

- Function handle

- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code as you start brushing a graph. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- `figure` — Figure object you are brushing.
- `axesStruct` — Structure that contains the axes you are brushing. This structure has one field, `Axes`, the axes object you are brushing. If you do not use this argument in your callback function, then replace it with the tilde character (`~`).

For more information about how to use function handles to define callback definitions, see "Callback Definition".

`ActionPostCallback` — Function to execute after brushing
[ ] (default) | function handle | cell array | character vector

Function to execute after brushing, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code after you finish brushing a graph. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- `figure` — Figure object you are brushing.
- `axesStruct` — Structure that contains the axes you are brushing. This structure has one field, `Axes`, the axes object you are brushing. If you do not use this argument in your callback function, then replace it with the tilde character (`~`).

For more information about how to use function handles to define callback definitions, see "Callback Definition".

`ButtonDownFilter` — Brush suppression callback
[ ] (default) | function handle | cell array | character vector

Brush suppression callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to suppress brushing under conditions that you define. A numeric or logical output of 1 (`true`) suppresses the brush action, and a numeric or logical output of 0 (`false`) allows the brush action to proceed. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- `axes` — The axes object you are brushing.
- `eventData` — Empty argument. Replace it with the tilde character (`~`) in the function definition to indicate that this argument is not used.

If you specify this property using a function handle, then you must assign the function's output to a variable.

For more information about how to use function handles to define callback definitions, see “Callback Definition”.

`FigureHandle` — Figure object
`Figure` object

This property is read-only.

`Figure` object in which brush marks appear.

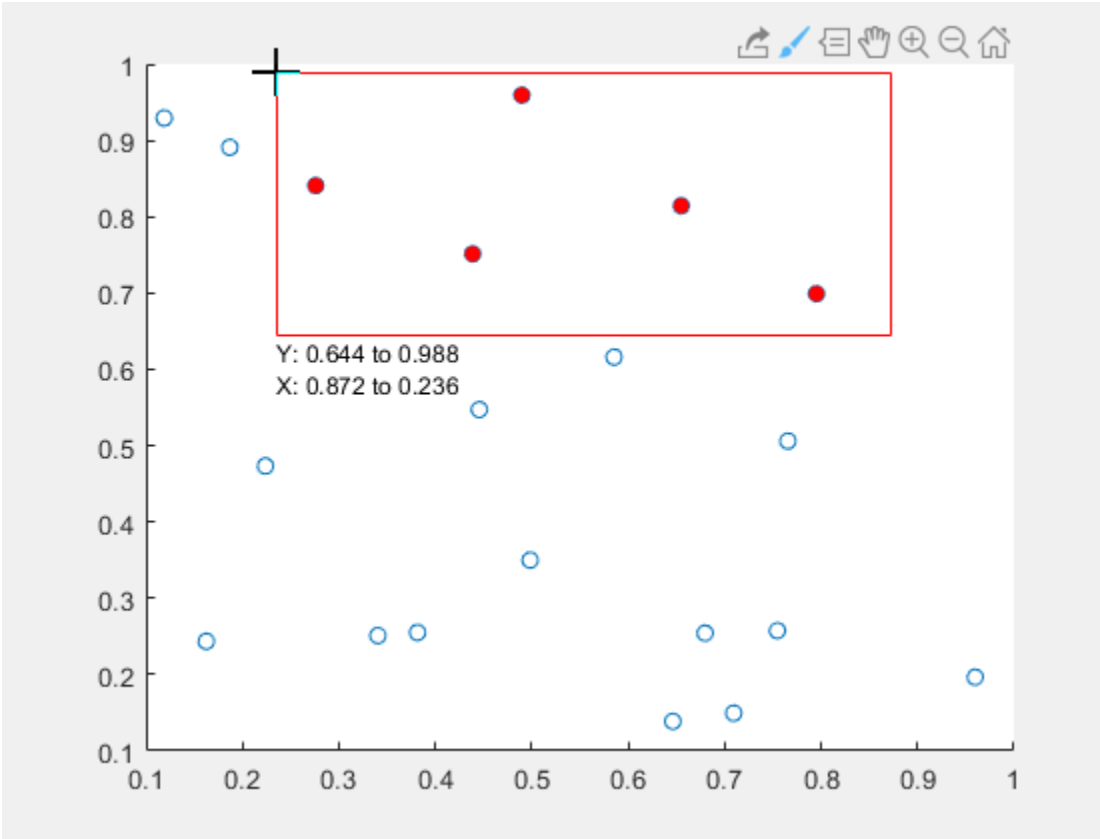
Examples

Mark Data Values in a Chart

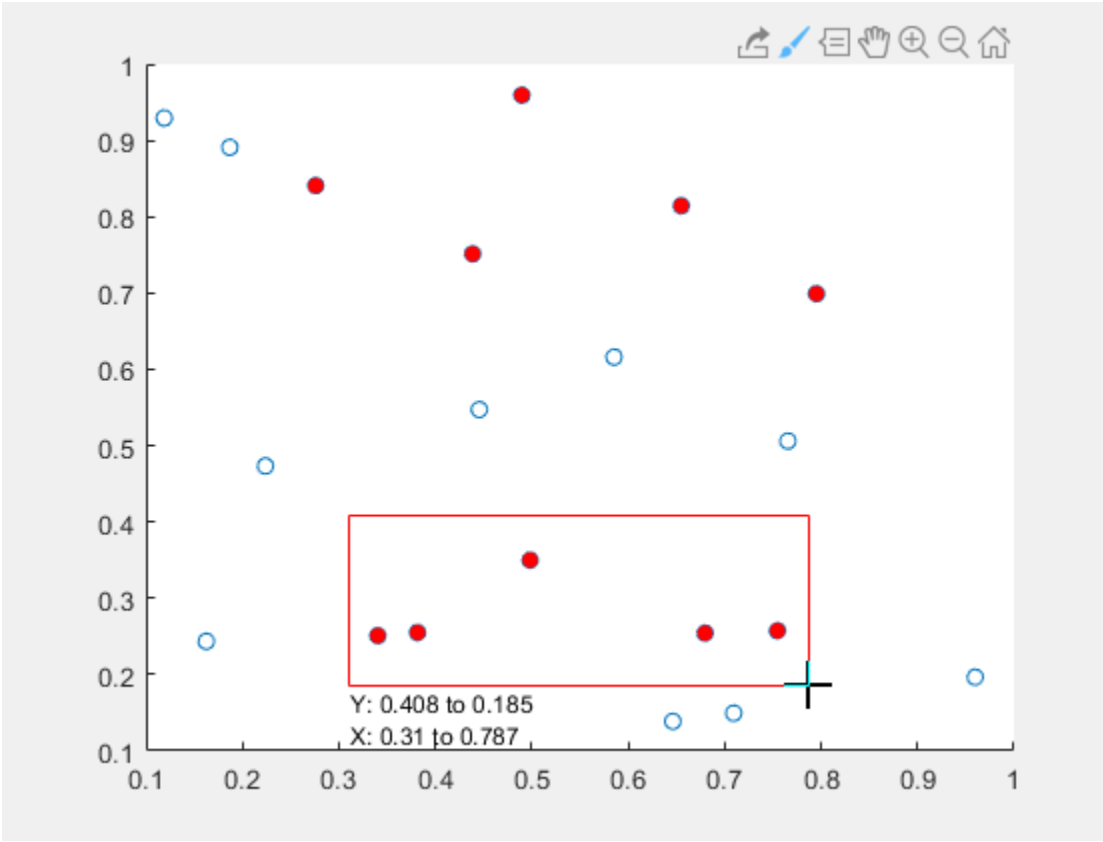
Use brushing to mark data values in a scatter plot. First, plot random `x` and `y` values. Then, turn on brushing and use the mouse to drag a rectangle around some points.

```
x = rand(20,1);
y = rand(20,1);
```

```
scatter(x,y);
brush('on')
```



To modify which points you have marked without discarding the entire set, hold down the **Shift** key while brushing. Add or remove a single point by clicking on it. Add or remove several points by dragging a selection rectangle.



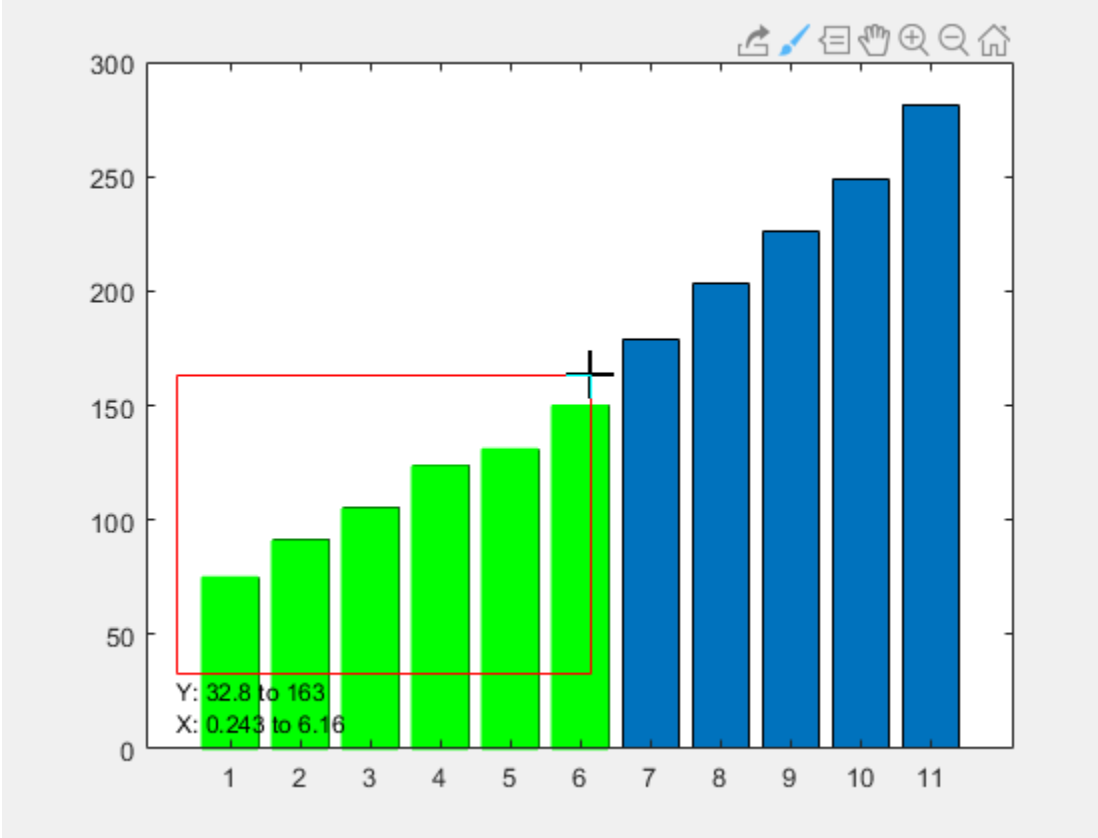
Clear all brush marks by clicking or dragging in an empty area of the axes.

Control Brush Mode and Color Using Brush Object

Plot data values in a bar chart and create a brush object. Then, enable brushing and set the brush color to `'green'` by updating the object's properties. Finally, mark data values by including the maximum value of each bar in the selection rectangle.

```
y = [75 91 105 123.5 131 150 179 203 226 249 281.5];
bar(y)
b = brush;
```

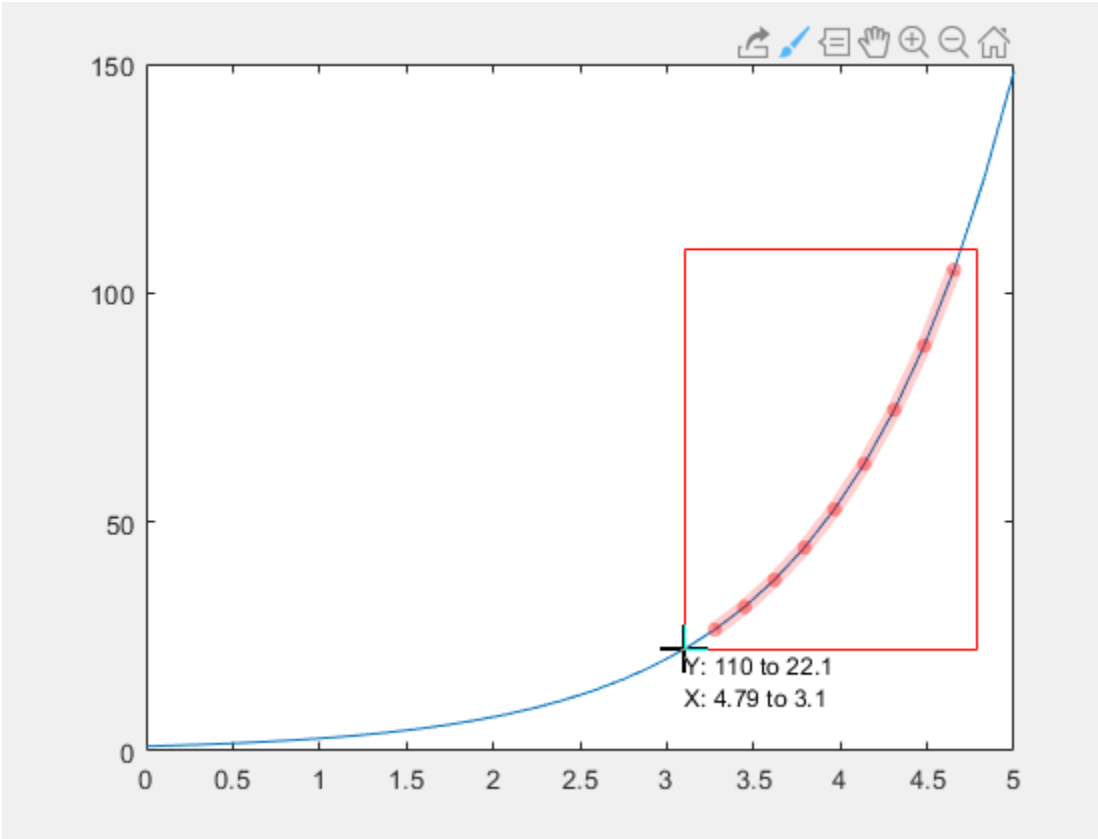
```
b.Enable = 'on';
b.Color = 'green';
```



Save Brushed Data in a Variable

Save brushed data in a variable by using the context menu. First, plot some x and y values. Turn on brushing, and then mark some data values.

```
x = linspace(0,5,30);
y = exp(x);
plot(x,y)
brush('on')
```



To store the brushed data in a variable, right-click on a brushed data value and select **Create Variable** from the context menu. When prompted, enter the variable name. In this case, use the name `bData`. Finally, view the contents of `bData`.

```
bData
```

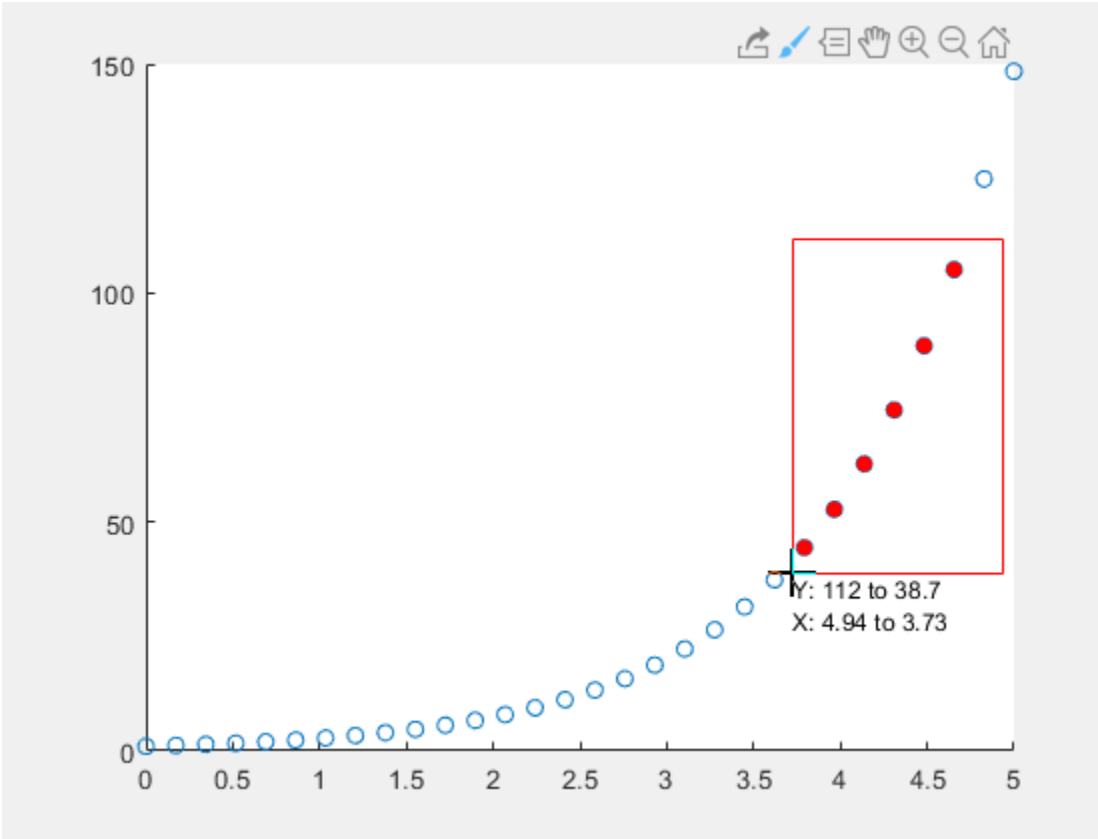
```
bData =

    3.2759   26.4660
    3.4483   31.4461
    3.6207   37.3633
    3.7931   44.3940
    3.9655   52.7475
    4.1379   62.6730
```

```
    4.3103   74.4662
    4.4828   88.4784
    4.6552  105.1273
```

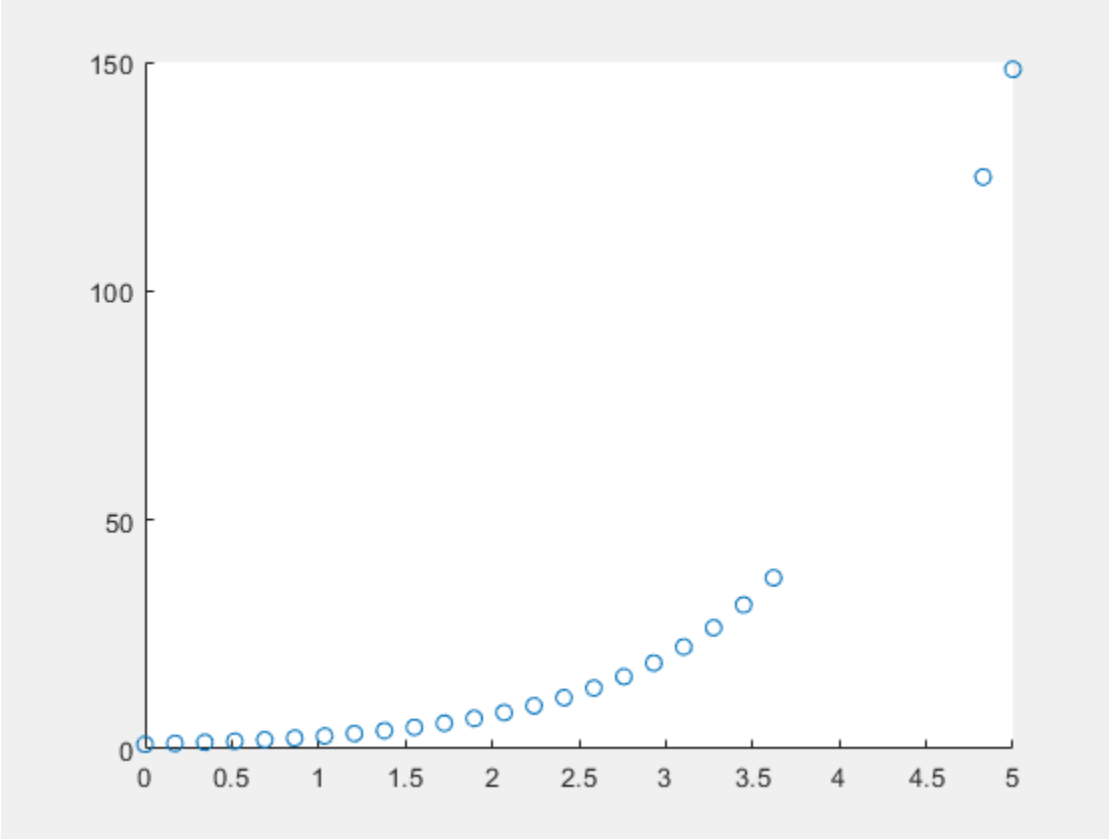
Remove Data Values by Brushing

Remove brushed data from a chart by using the context menu. First, plot some x and y values. Turn on brushing, and then mark some data values.

```
x = linspace(0,5,30);
y = exp(x);
scatter(x,y)
brush('on')
```



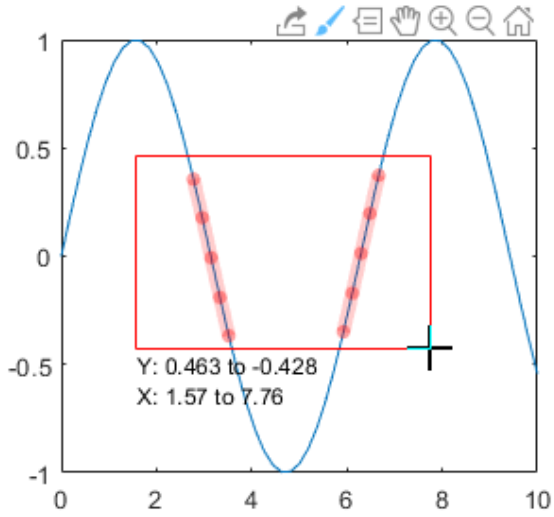
To remove the brushed data from the chart, right-click on a brushed data value and select **Remove** from the context menu.

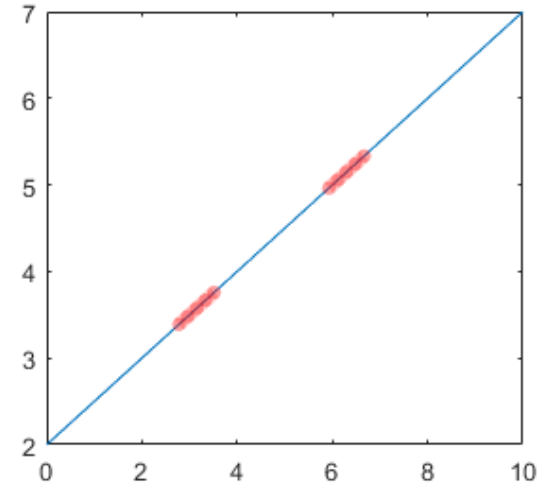


More About

How Data Linking Affects Data Brushing

Data linking automatically updates figures as plotted variables change. Enable data linking for a figure using the command `linkdata on`. Brush marks created on a linked graph will appear on other linked graphs that display the same variables. This occurs even if other plots are not in brushing mode.



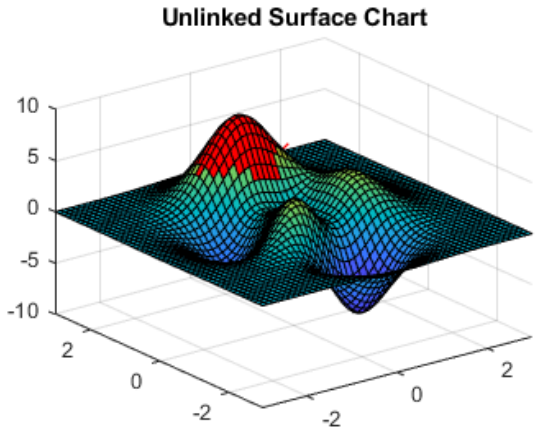


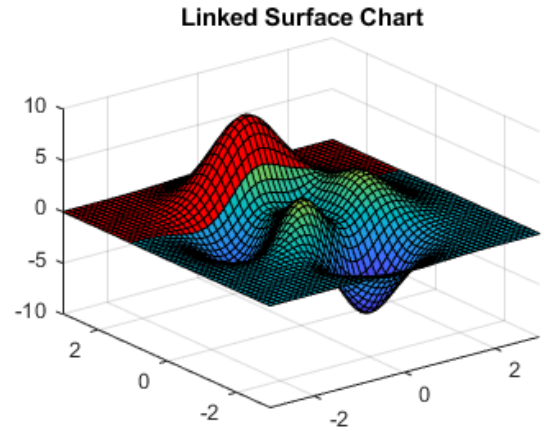
When you turn on data linking for a figure, existing brush marks are replaced by the marks in other linked figures. If you turn on data linking and there are no other linked figures, then existing brush marks disappear. If you brush a graph and then turn off data linking, the brush marks remain.

The following conditions apply:

- Brush marks appear in the same color in all linked charts. The brush color that displays in linked charts is the brush color associated with the graph you are brushing.
- The linked variable must not be complex. You can brush complex data values, but brush marks will not appear in the other linked charts. For more information about linking complex variables, see `linkdata`.

Brush marks on a surface chart display differently when the surface chart has linked data. When you brush one face of a linked surface chart, all faces with the same `yData` are also brushed. This behavior allows you to investigate relationships between columns of data distributed among multiple rows. The following images illustrate the difference between brushing unlinked and linked surface charts.





Tips

- To brush a histogram chart, you must first turn on data linking.

See Also

`linkdata` | `pan` | `rotate3d` | `zoom`

Topics

"Interactively Explore Plotted Data"

Introduced in R2008a

bsxfun

Apply element-wise operation to two arrays with implicit expansion enabled

Syntax

```
C = bsxfun(fun,A,B)
```

Description

`C = bsxfun(fun,A,B)` applies the element-wise binary operation specified by the function handle `fun` to arrays `A` and `B`.

Examples

Deviation of Matrix Elements from Column Mean

Subtract the column mean from the corresponding column elements of a matrix `A`. Then normalize by the standard deviation.

```
A = [1 2 10; 3 4 20; 9 6 15];
C = bsxfun(@minus, A, mean(A));
D = bsxfun(@rdivide, C, std(A))
```

```
D = 3×3

   -0.8006   -1.0000   -1.0000
   -0.3203         0    1.0000
    1.1209    1.0000         0
```

In MATLAB® R2016b and later, you can directly use operators instead of `bsxfun`, since the operators independently support implicit expansion of arrays with compatible sizes.

```
(A - mean(A))./std(A)
```

```
ans = 3×3

   -0.8006   -1.0000   -1.0000
   -0.3203         0    1.0000
    1.1209    1.0000         0
```

Compare Vector Elements

Compare the elements in a column vector and a row vector. The result is a matrix containing the comparison of each combination of elements from the vectors. An equivalent way to execute this operation is with `A > B`.

```
A = [8; 17; 20; 24]
```

```
A = 4×1

     8
    17
    20
    24
```

```
B = [0 10 21]
```

```
B = 1×3

     0    10    21
```

```
C = bsxfun(@gt,A,B)
```

```
C = 4x3 logical array

   1   0   0
   1   1   0
   1   1   0
   1   1   1
```

Expansion with Custom Function

Create a function handle that represents the function $f(a,b) = a - e^b$.

```
fun = @(a,b) a - exp(b);
```

Use `bsxfun` to apply the function to vectors `a` and `b`. The `bsxfun` function expands the vectors into matrices of the same size, which is an efficient way to evaluate `fun` for many combinations of the inputs.

```
a = 1:7;
b = pi*[0 1/4 1/3 1/2 2/3 3/4 1].';
C = bsxfun(fun,a,b)
```

```
C = 7×7

         0    1.0000    2.0000    3.0000    4.0000    5.0000    6.0000
   -1.1933   -0.1933    0.8067    1.8067    2.8067    3.8067    4.8067
   -1.8497   -0.8497    0.1503    1.1503    2.1503    3.1503    4.1503
   -3.8105   -2.8105   -1.8105   -0.8105    0.1895    1.1895    2.1895
   -7.1205   -6.1205   -5.1205   -4.1205   -3.1205   -2.1205   -1.1205
   -9.5507   -8.5507   -7.5507   -6.5507   -5.5507   -4.5507   -3.5507
  -22.1407  -21.1407  -20.1407  -19.1407  -18.1407  -17.1407  -16.1407
```

Input Arguments

`fun` — Binary function to apply
function handle

Binary function to apply, specified as a function handle. `fun` must be a binary (two-input) element-wise function of the form `C = fun(A,B)` that accepts arrays `A` and `B` with compatible sizes. For more information, see “Compatible Array Sizes for Basic Operations”. `fun` must support scalar expansion, such that if `A` or `B` is a scalar, then `C` is the result of applying the scalar to every element in the other input array.

In MATLAB R2016b and later, the built-in binary functions listed in this table independently support implicit expansion. With these functions, you can call the function or operator directly instead of using `bsxfun`. For example, you can replace `C = bsxfun(@plus,A,B)` with `A+B`.

Function	Symbol	Description
plus	+	Plus
minus	-	Minus
times	.*	Array multiply
rdivide	./	Right array divide
ldivide	.\	Left array divide
power	.^	Array power
eq	==	Equal
ne	~=	Not equal
gt	>	Greater than
ge	>=	Greater than or equal to
lt	<	Less than
le	<=	Less than or equal to
and	&	Element-wise logical AND
or	|	Element-wise logical OR
xor	N/A	Logical exclusive OR
max	N/A	Binary maximum
min	N/A	Binary minimum
mod	N/A	Modulus after division
rem	N/A	Remainder after division
atan2	N/A	Four-quadrant inverse tangent; result in radians
atan2d	N/A	Four-quadrant inverse tangent; result in degrees
hypot	N/A	Square root of sum of squares

Example: `C = bsxfun(@plus,[1 2],[2; 3])`

Data Types: `function_handle`

A,B — Input arrays
scalars | vectors | matrices | multidimensional arrays

Input arrays, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs A and B must have compatible sizes. For more information, see "Compatible Array Sizes for Basic Operations". Whenever a dimension of A or B is singleton (equal to one), bsxfun virtually replicates the array along that dimension to match the other array. In the case where a dimension of A or B is singleton, and the corresponding dimension in the other array is zero, bsxfun virtually diminishes the singleton dimension to zero.

Data Types: single | double | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64 | char | logical
Complex Number Support: Yes

Tips

- It is recommended that you replace most uses of bsxfun with direct calls to the functions and operators that support implicit expansion. Compared to using bsxfun, implicit expansion offers faster speed of execution, better memory usage, and improved readability of code. For more information, see "Compatible Array Sizes for Basic Operations".

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

The specified function must not rely on persistent variables.

For more information, see "Tall Arrays".

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `bsxfun` in the Parallel Computing Toolbox documentation.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`arrayfun` | `repmat`

Topics

“Array vs. Matrix Operations”

Introduced in R2007a

builddocsearchdb

Build searchable documentation database

Syntax

```
builddocsearchdb(folder)
```

Description

`builddocsearchdb(folder)` builds a searchable database, also referred to as a search index, from HTML files in the specified folder.

The `builddocsearchdb` function creates a subfolder, `helpsearch-v3`, to contain the database files. The database enables MATLAB to search for content within the HTML files assuming the MATLAB version is the same version used to create the database.

Beginning with MATLAB R2014b, you can maintain search indexes side by side. For instance, if you already have a search index for MATLAB R2014a or earlier, run `builddocsearchdb` against your help files using MATLAB R2014b or later. Then, when you run any MATLAB release, the help browser automatically uses the appropriate index for searching your documentation database.

Examples

Search Custom Help Files

Build a search database for custom help files.

MATLAB includes a set of sample files to demonstrate how to create a custom toolbox and supporting documentation. This sample toolbox is called the Upslope Area Toolbox. The `upslope` folder includes a file named `info.xml`, which is required to display custom documentation, and a subfolder named `html`, which contains HTML documentation and supporting files.

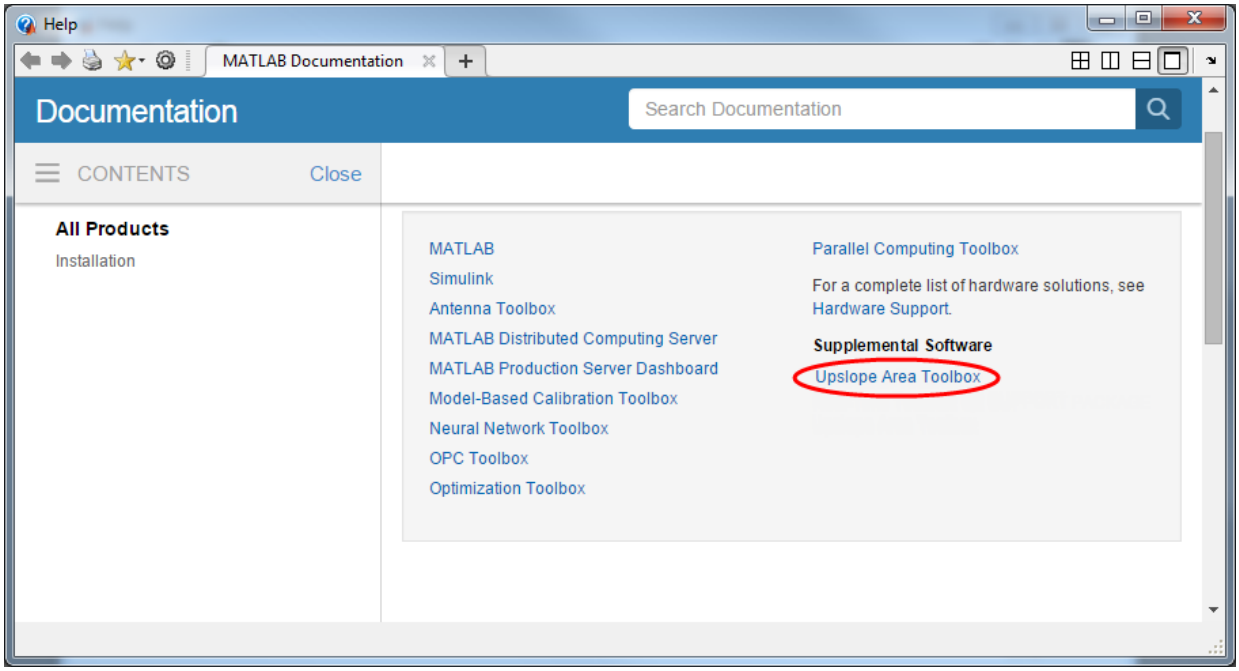
Copy the sample files to a temporary folder, and add the copied files to the path.

```
sample = fullfile(...
         matlabroot,'help','techdoc','matlab_env',...
         'examples','upslope');
tmp = tempname;
mkdir(tmp);
copyfile(sample,tmp);
addpath(tmp);
```

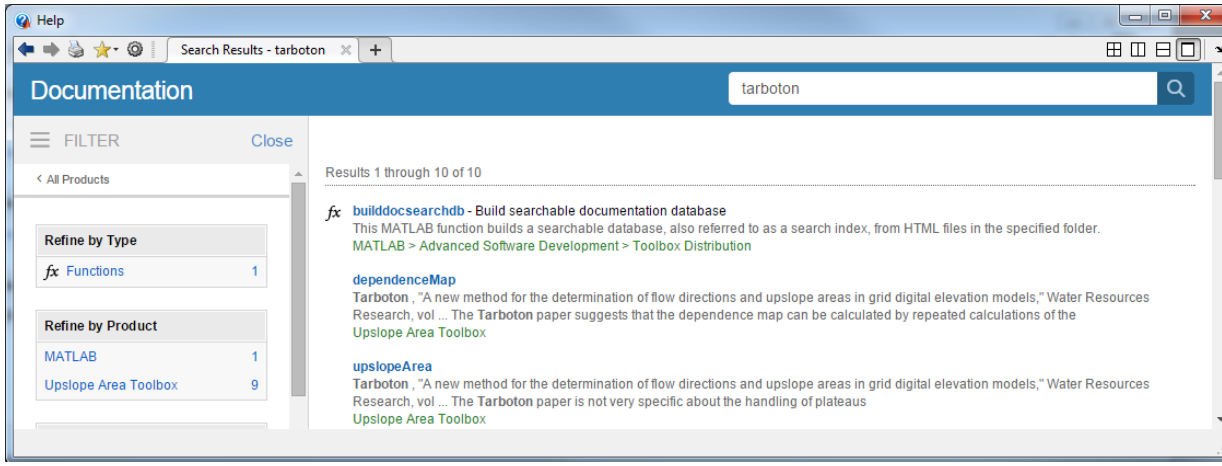
Create a search database.

```
folder = fullfile(tmp,'html');
builddocsearchdb(folder)
```

Under the Supplemental Software heading, click **Upslope Area Toolbox**.



Search the supplemental documentation for the term *tarboton*, which appears in several of the example help files. The search returns several results from the Upslope Area Toolbox.



Remove the temporary example files.

```
rmpath(tmp)
rmdir(tmp,'s')
```

Note Your help appears alongside the MathWorks documentation only when you are viewing installed documentation. If you are viewing documentation on the web, then your documentation displays in a help browser separate from the MathWorks documentation.

Input Arguments

`folder` — Full path to folder with HTML files

character vector

Full path to a folder with HTML files, specified as a character vector. The folder must be:

- On the MATLAB search path
- Outside the *`matlabroot`* folder
- Outside any installed Hardware Support Package help folder

To include a particular HTML document in the search database, the `builddocsearchdb` function requires that:

- The document has a title.
- The content is different from the title.

Example: `builddocsearchdb('c:\myfiles\html')`

See Also

`doc` | `help`

Topics

"Display Custom Documentation"

Introduced in R2007a

builtin

Execute built-in function from overloaded method

Syntax

```
builtin(function,x1,...,xn)
[y1,...,yn] = builtin(function,x1,...,xn)
```

Description

`builtin(function,x1,...,xn)` executes the built-in `function` with the input arguments `x1` through `xn`. Use `builtin` to execute the original built-in from within a method that overloads the function. To work properly, you must never overload `builtin`.

`[y1,...,yn] = builtin(function,x1,...,xn)` stores any output from `function` in `y1` through `yn`.

Examples

Run an Overloaded Function within a Class Definition

Execute the built-in functionality from within an overloaded method.

Create a simple class describing the speed of a particle and providing a `disp` method by pasting the following code into a file called `MyParticle.m`.

```
classdef MyParticle
    properties
        velocity;
    end
    methods
        function p = MyParticle(x,y,z)
            p.velocity.x = x;
            p.velocity.y = y;
```

```
            p.velocity.z = z;
        end
        function disp(p)
            builtin('disp',p) % call builtin
            if isscalar(p)
                disp('  Velocity')
                disp(['  x: ',num2str(p.velocity.x)])
                disp(['  y: ',num2str(p.velocity.y)])
                disp(['  z: ',num2str(p.velocity.z)])
            end
        end
    end
end
```

Create an instance `MyParticle`.

```
p = MyParticle(1,2,4)
```

```
p = 

  MyParticle

  Properties:
    velocity: [1x1 struct]

  Methods

  Velocity
  x: 1
  y: 2
  z: 4
```

Input Arguments

`function` — Built-in function name
character vector | string scalar

Built-in function name in the MATLAB path, specified as a character vector or string scalar. `function` cannot be a function handle.

`x1,...,xn` — Valid input arguments for `function`
supported data types

Valid input arguments for `function`, specified by supported data types.

More About

built-in function

A built-in function is part of the MATLAB executable. MATLAB does not implement these functions in the MATLAB language. Although most built-in functions have a `.m` file associated with them, this file only supplies documentation for the function.

You can use the syntax `which` *`function`* to check whether a function is built-in.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`feval` | `which`

Introduced before R2006a

bvp4c

Solve boundary value problem — fourth-order method

Syntax

```
sol = bvp4c(odefun,bcfun,solinit)
sol = bvp4c(odefun,bcfun,solinit,options)
```

Description

`sol = bvp4c(odefun,bcfun,solinit)` integrates a system of differential equations of the form $y' = f(x,y)$ specified by `odefun`, subject to the boundary conditions described by `bcfun` and the initial solution guess `solinit`. Use the `bvpinit` function to create the initial guess `solinit`, which also defines the points at which the boundary conditions in `bcfun` are enforced.

`sol = bvp4c(odefun,bcfun,solinit,options)` also uses the integration settings defined by `options`, which is an argument created using the `bvpset` function. For example, use the `AbsTol` and `RelTol` options to specify absolute and relative error tolerances, or the `FJacobian` option to provide the analytical partial derivatives of `odefun`.

Examples

Solve Second-Order BVP

Solve a second-order BVP in MATLAB® using functions. For this example, use the second-order equation

$y'' + y = 0.$

The equation is defined on the interval $[0, \pi/2]$ subject to the boundary conditions

$y(0) = 0,$

$y(\pi/2) = 2$.

To solve this equation in MATLAB, you need to write a function that represents the equation as a system of first-order equations, a function for the boundary conditions, and a function for the initial guess. Then the BVP solver uses these three inputs to solve the equation.

Code Equation

Write a function that codes the equation. Use the substitutions $y_1 = y$ and $y_2 = y'$ to rewrite the equation as a system of first-order equations.

$y_1' = y_2$,

$y_2' = -y_1$.

The corresponding function is

```
function dydx = bvpfcn(x,y)
dydx = zeros(2,1);
dydx = [y(2)
       -y(1)];
end
```

Note: All functions are included at the end of the example as local functions.

Code Boundary Conditions

Write a function that codes the boundary conditions in the form $g(y(a), y(b)) = 0$. In this form the boundary conditions are

$y(0) = 0$,

$y(\pi/2) - 2 = 0$.

The corresponding function is

```
function res = bcfcn(ya,yb)
res = [ya(1)
       yb(1)-2];
end
```

Create Initial Guess

Use the `bvpinit` function to create an initial guess for the solution of the equation. Since the equation relates y'' to y, a reasonable guess is that the solution involves trigonometric functions. Use a mesh of five points in the interval of integration. The first and last values in the mesh are where the solver applies the boundary conditions.

The function for the initial guess accepts x as an input and returns a guess for the value of y_1 and y_2. The function is

```
function g = guess(x)
g = [sin(x)
     cos(x)];
end
```

```
xmesh = linspace(0,pi/2,5);
solinit = bvpinit(xmesh, @guess);
```

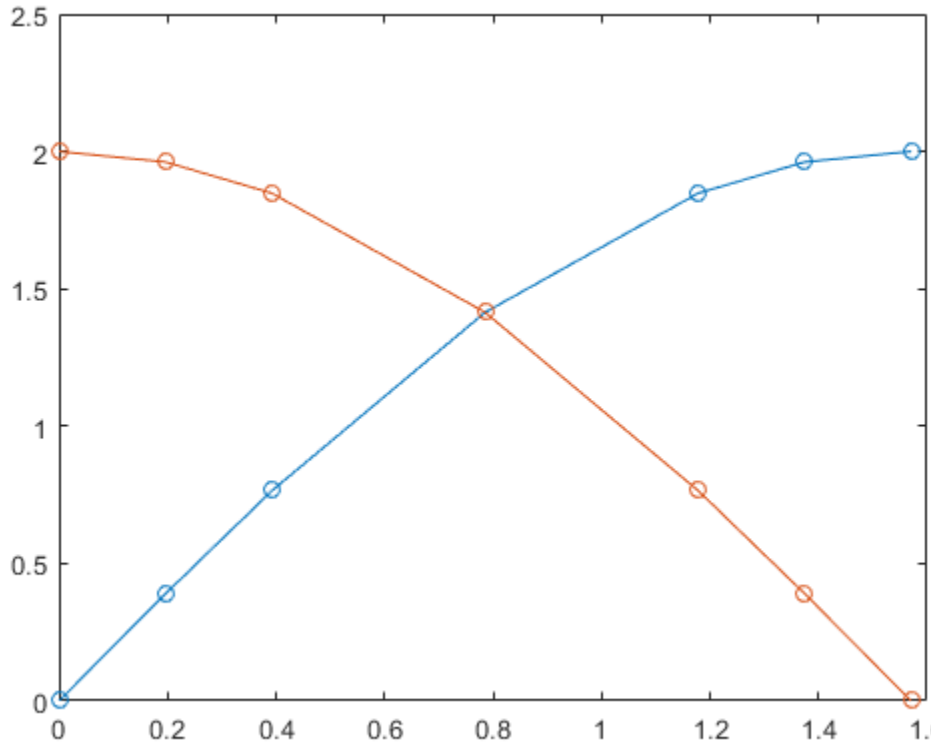
Solve Equation

Use `bvp4c` with the derivative function, boundary condition function, and initial guess to solve the problem.

```
sol = bvp4c(@bvpfcn, @bcfcn, solinit);
```

Plot Solution

```
plot(sol.x, sol.y, '-o')
```



Local Functions

Listed here are the local functions that `bvp4c` uses to solve the equation.

```
function dydx = bvpfcn(x,y) % equation to solve
dydx = zeros(2,1);
dydx = [y(2)
       -y(1)];
end
%--------------------------------
function res = bcfcn(ya,yb) % boundary conditions
res = [ya(1)
       yb(1)-2];
end
```

```
%--------------------------------
function g = guess(x) % initial guess for y and y'
g = [sin(x)
     cos(x)];
end
%--------------------------------
```

Compare bvp4c and bvp5c Solvers

Solve a BVP at a crude error tolerance with two different solvers and compare the results.

Consider the second-order ODE

$$y'' + \frac{2}{x}y' + \frac{1}{x^4}y = 0.$$

The equation is defined on the interval $\left[\frac{1}{3\pi}, 1\right]$ subject to the boundary conditions

$$y\left(\frac{1}{3\pi}\right) = 0,$$

$$y(1) = \sin(1).$$

To solve this equation in MATLAB®, you need to write a function that represents the equation as a system of first-order equations, write a function for the boundary conditions, set some option values, and create an initial guess. Then the BVP solver uses these four inputs to solve the equation.

Code Equation

With the substitutions $y_1 = y$ and $y_2 = y'$, you can rewrite the ODE as a system of first-order equations

$$y_1' = y_2,$$

$$y_2' = -\frac{2}{x}y_2 - \frac{1}{x^4}y_1.$$

The corresponding function is

```
function dydx = bvpfcn(x,y)
dydx = [y(2)
```

```
        -2*y(2)/x - y(1)/x^4];
end
```

Note: All functions are included at the end of the example as local functions.

Code Boundary Conditions

The boundary condition function requires that the boundary conditions are in the form $g(y(a), y(b)) = 0$. In this form, the boundary conditions are

$$y\left(\frac{1}{3\pi}\right) = 0,$$

$$y(1) - \sin(1) = 0.$$

The corresponding function is

```
function res = bcfcn(ya,yb)
res = [ya(1)
       yb(1)-sin(1)];
end
```

Set Options

Use `bvpset` to turn on the display of solver statistics, and specify crude error tolerances to highlight the difference in error control between the solvers. Also, for efficiency, specify the analytical Jacobian

$$J = \frac{\partial f_i}{\partial y} = \begin{bmatrix} \frac{\partial f_1}{\partial y_1} & \frac{\partial f_1}{\partial y_2} \\ \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial y_2} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{1}{x^4} & -\frac{2}{x} \end{bmatrix}.$$

The corresponding function that returns the value of the Jacobian is

```
function dfdy = jac(x,y)
dfdy = [0      1
       -1/x^4 -2/x];
end

opts = bvpset('FJacobian',@jac,'RelTol',0.1,'AbsTol',0.1,'Stats','on');
```

Create Initial Guess

Use `bvpinit` to create an initial guess of the solution. Specify a constant function as the initial guess with an initial mesh of 10 points in the interval $[1/3\pi, 1]$.

```
xmesh = linspace(1/(3*pi), 1, 10);
solinit = bvpinit(xmesh, [1; 1]);
```

Solve Equation

Solve the equation with both `bvp4c` and `bvp5c`.

```
sol4c = bvp4c(@bvpfcn, @bcfcn, solinit, opts);
```

```
The solution was obtained on a mesh of 9 points.
The maximum residual is  9.794e-02.
There were 157 calls to the ODE function.
There were 28 calls to the BC function.
```

```
sol5c = bvp5c(@bvpfcn, @bcfcn, solinit, opts);
```

```
The solution was obtained on a mesh of 11 points.
The maximum error is  6.742e-02.
There were 244 calls to the ODE function.
There were 29 calls to the BC function.
```

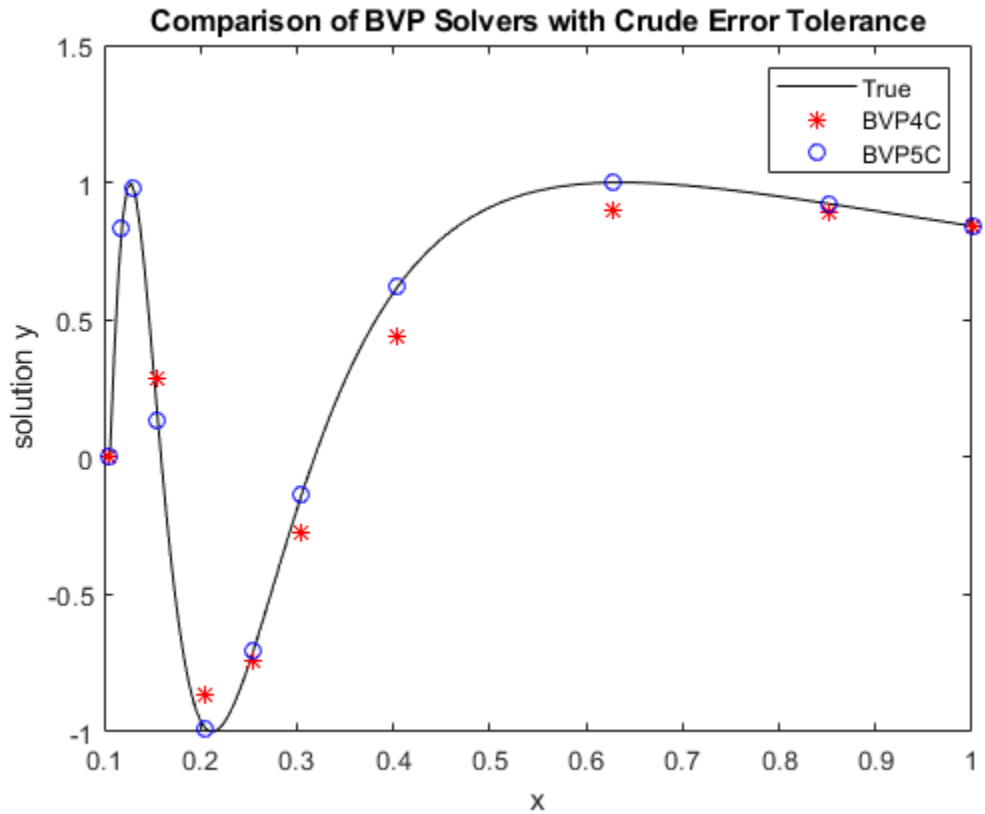
Plot Results

Plot the results of the two calculations for y_1 with the analytic solution for comparison. The analytic solution is

$$y_1 = \sin\left(\frac{1}{x}\right),$$

$$y_2 = -\frac{1}{x^2}\cos\left(\frac{1}{x}\right).$$

```
xplot = linspace(1/(3*pi),1,200);
yplot = [sin(1./xplot); -cos(1./xplot)./xplot.^2];

plot(xplot,yplot(1,:),'k',sol4c.x,sol4c.y(1,:),'r*',sol5c.x,sol5c.y(1,:),'bo')
title('Comparison of BVP Solvers with Crude Error Tolerance')
legend('True','BVP4C','BVP5C')
xlabel('x')
ylabel('solution y')
```



The plot confirms that `bvp5c` directly controls the true error in the calculation, while `bvp4c` controls it only indirectly. At more stringent error tolerances, this difference between the solvers is not as apparent.

Local Functions

Listed here are the local functions that the BVP solvers use to solve the problem.

```
function dydx = bvpfcn(x,y) % equation to solve
dydx = [y(2)
       -2*y(2)/x - y(1)/x^4];
end
%--------------------------------
function res = bcfcn(ya,yb) % boundary conditions
```

```
res = [ya(1)
       yb(1)-sin(1)];
end
%---------------------------------
function dfdy = jac(x,y) % analytical jacobian for f
dfdy = [0      1
       -1/x^4 -2/x];
end
%---------------------------------
```

Input Arguments

odefun — Functions to solve
function handle

Functions to solve, specified as a function handle that defines the functions to be integrated. `odefun` and `bcfun` must accept the same number of input arguments.

To code `odefun`, use the functional signature `dydx = odefun(x,y)` for a scalar `x` and column vector `y`. The return value `dydt` is a column vector of data type `single` or `double` that corresponds to $f(x,y)$. `odefun` must accept both input arguments `x` and `y`, even if one of the arguments is not used in the function.

For example, to solve $y' = 5y - 3$, use the function:

```
function dydt = odefun(t,y)
dydt = 5*y-3;
end
```

For a system of equations, the output of `odefun` is a vector. Each element in the vector is the solution to one equation. For example, to solve

$$y'_1 = y_1 + 2y_2$$
$$y'_2 = 3y_1 + 2y_2$$

use the function:

```
function dydt = odefun(t,y)
dydt = zeros(2,1);
dydt(1) = y(1)+2*y(2);
dydt(2) = 3*y(1)+2*y(2);
end
```

`bvp4c` also can solve problems with singularities in the solution on page 1-1128 or multipoint boundary conditions on page 1-1127.

Example: `sol = bvp4c(@odefun, @bcfun, solinit)`

Unknown Parameters

If the BVP being solved includes unknown parameters, you instead can use the functional signature `dydx = odefun(x,y,p)`, where `p` is a vector of parameter values. When you use this functional signature the BVP solver calculates the values of the unknown parameters starting from the initial guess for the parameter values provided in `solinit`.

Data Types: `function_handle`

`bcfun` — Boundary conditions

function handle

Boundary conditions, specified as a function handle that computes the residual error in the boundary conditions. `odefun` and `bcfun` must accept the same number of input arguments.

To code `bcfun`, use the functional signature `res = bcfun(ya,yb)` for column vectors `ya` and `yb`. The return value `res` is a column vector of data type `single` or `double` that corresponds to the residual value of the boundary conditions at the boundary points.

For example, if *y(a) = 1* and *y(b) = 0*, then the boundary condition function is

```
function res = bcfun(ya,yb)
res = [ya(1)-1
       yb(1)];
end
```

Since *y(a) = 1*, the residual value of `ya(1)-1` should be `0` at the point $x = a$. Similarly, since *y(b) = 0*, the residual value of `yb(1)` should be `0` at the point $x = b$.

The boundary points $x = a$ and $x = b$ where the boundary conditions are enforced are defined in the initial guess structure `solinit`. For two-point boundary value problems, `a = solinit.x(1)` and `b = solinit.x(end)`.

Example: `sol = bvp4c(@odefun, @bcfun, solinit)`

Unknown Parameters

If the BVP being solved includes unknown parameters, you instead can use the functional signature `res = bcfun(ya,yb,p)`, where `p` is a vector of parameter values. When you

use this functional signature the BVP solver calculates the values of the unknown parameters starting from the initial guess for the parameter values provided in `solinit`.

Data Types: `function_handle`

`solinit` — Initial guess of solution
structure

Initial guess of solution, specified as a structure. Use `bvpinit` to create `solinit`.

Unlike initial value problems, a boundary value problem can have no solution, a finite number of solutions, or infinitely many solutions. An important part of the process of solving a BVP is providing a guess for the required solution. The quality of this guess can be critical for the solver performance and even for a successful computation. For some guidelines on creating a good initial guess, see “Initial Guess of Solution”.

Example: `sol = bvp4c(@odefun, @bcfun, solinit)`

Data Types: `struct`

`options` — Option structure
structure

Option structure. Use the `bvpset` function to create or modify the options structure.

Example: `options = bvpset('RelTol',1e-5,'Stats','on')` specifies a relative error tolerance of `1e-5` and turns on the display of solver statistics.

Data Types: `struct`

Output Arguments

`sol` — Solution structure
structure

Solution structure. You can access the fields in `sol` with dot-indexing, such as `sol.field1`. The solution (`sol.x`,`sol.y`) is continuous on the interval of integration defined in the initial mesh `solinit.x` and has a continuous first derivative there. You can use `sol` with the `deval` function to evaluate the solution at other points in the interval.

The structure `sol` has these fields.

Field	Description
`x`	Mesh selected by `bvp4c`. This mesh typically contains different points than the initial mesh `solinit.x`.
`y`	Approximation to $y(x)$ at the mesh points of `sol.x`.
`yp`	Approximation to $y'(x)$ at the mesh points of `sol.x`.
`parameters`	Final values for the unknown parameters specified in `solinit.parameters`.
`solver`	`'bvp4c'`
`stats`	Computational cost statistics related to the solution: the number of mesh points, residual error, and number of calls to `odefun` and `bcfun`.

More About

Multipoint Boundary Value Problems

For multipoint boundary value problems, the boundary conditions are enforced at several points in the interval of integration.

`bvp4c` can solve multipoint boundary value problems where $a = a_0 < a_1 < a_2 < ... < a_n = b$ are boundary points in the interval $[a,b]$. The points $a_1,a_2,...,a_{n-1}$ represent interfaces that divide $[a,b]$ into regions. `bvp4c` enumerates the regions from left to right (from a to b), with indices starting from 1. In region k, $[a_{k-1},a_k]$, `bvp4c` evaluates the derivative as

```
yp = odefun(x,y,k)
```

In the boundary conditions function `bcfun(yleft,yright)`, `yleft(:,k)` is the solution at the left boundary of $[a_{k-1},a_k]$. Similarly, `yright(:,k)` is the solution at the right boundary of region k. In particular, `yleft(:,1) = y(a)` and `yright(:,end) = y(b)`.

When you create an initial guess with `bvpinit`, use double entries in `xinit` for each interface point. See the reference page for `bvpinit` for more information.

If `yinit` is a function, `bvpinit` calls `y = yinit(x,k)` to get an initial guess for the solution at `x` in region `k`. In the solution structure `sol` returned by `bpv4c`, `sol.x` has double entries for each interface point. The corresponding columns of `sol.y` contain the left and right solution at the interface, respectively.

See “Solve BVP with Multiple Boundary Conditions” for an example that solves a three-point boundary value problem.

Singular Boundary Value Problems

`bvp4c` solves a class of singular boundary value problems, including problems with unknown parameters `p`, of the form

$$y' = S\frac{y}{x} + f(x, y, p),$$
$$0 = bc(y(0), y(b), p).$$

The interval is required to be $[0, b]$ with $b > 0$. Often such problems arise when computing a smooth solution of ODEs that result from partial differential equations (PDEs) due to cylindrical or spherical symmetry. For singular problems, you specify the (constant) matrix `S` as the value of the `'SingularTerm'` option of `bvpset`, and `odefun` evaluates only $f(x,y,p)$. The boundary conditions and initial guess must be consistent with the necessary condition for smoothness $S \cdot y(0) = 0$.

See “Solve BVP with Singular Term” for an example that solves a singular boundary value problem.

Algorithms

`bvp4c` is a finite difference code that implements the three-stage Lobatto IIIa formula [1], [2]. This is a collocation formula and the collocation polynomial provides a C^1-continuous solution that is fourth-order accurate uniformly in the interval of integration. Mesh selection and error control are based on the residual of the continuous solution.

The collocation technique uses a mesh of points to divide the interval of integration into subintervals. The solver determines a numerical solution by solving a global system of algebraic equations resulting from the boundary conditions, and the collocation conditions imposed on all the subintervals. The solver then estimates the error of the numerical solution on each subinterval. If the solution does not satisfy the tolerance

criteria, the solver adapts the mesh and repeats the process. You *must* provide the points of the initial mesh, as well as an initial approximation of the solution at the mesh points.

References

[1] Shampine, L.F., and J. Kierzenka. "A BVP Solver based on residual control and the MATLAB PSE." *ACM Trans. Math. Softw.* Vol. 27, Number 3, 2001, pp. 299–316.

[2] Shampine, L.F., M.W. Reichelt, and J. Kierzenka. "Solving Boundary Value Problems for Ordinary Differential Equations in MATLAB with bvp4c." *MATLAB File Exchange*, 2004.

See Also

`bvp5c` | `bvpget` | `bvpinit` | `bvpset` | `bvpxtend` | `deval`

Topics

“Solving Boundary Value Problems”

Introduced before R2006a

bvp5c

Solve boundary value problem — fifth-order method

Syntax

```
sol = bvp5c(odefun,bcfun,solinit)
sol = bvp5c(odefun,bcfun,solinit,options)
```

Description

`sol = bvp5c(odefun,bcfun,solinit)` integrates a system of differential equations of the form $y' = f(x,y)$ specified by `odefun`, subject to the boundary conditions described by `bcfun` and the initial solution guess `solinit`. Use the `bvpinit` function to create the initial guess `solinit`, which also defines the points at which the boundary conditions in `bcfun` are enforced.

`sol = bvp5c(odefun,bcfun,solinit,options)` also uses the integration settings defined by `options`, which is an argument created using the `bvpset` function. For example, use the `AbsTol` and `RelTol` options to specify absolute and relative error tolerances, or the `FJacobian` option to provide the analytical partial derivatives of `odefun`.

Examples

Solve Second-Order BVP

Solve a second-order BVP in MATLAB® using functions. For this example, use the second-order equation

$y'' + y = 0.$

The equation is defined on the interval $[0, \pi/2]$ subject to the boundary conditions

$y(0) = 0,$

$y(\pi/2) = 2$.

To solve this equation in MATLAB, you need to write a function that represents the equation as a system of first-order equations, a function for the boundary conditions, and a function for the initial guess. Then the BVP solver uses these three inputs to solve the equation.

Code Equation

Write a function that codes the equation. Use the substitutions $y_1 = y$ and $y_2 = y'$ to rewrite the equation as a system of first-order equations.

$y_1' = y_2$,

$y_2' = -y_1$.

The corresponding function is

```
function dydx = bvpfcn(x,y)
dydx = zeros(2,1);
dydx = [y(2)
       -y(1)];
end
```

Note: All functions are included at the end of the example as local functions.

Code Boundary Conditions

Write a function that codes the boundary conditions in the form $g(y(a), y(b)) = 0$. In this form the boundary conditions are

$y(0) = 0$,

$y(\pi/2) - 2 = 0$.

The corresponding function is

```
function res = bcfcn(ya,yb)
res = [ya(1)
       yb(1)-2];
end
```

Create Initial Guess

Use the `bvpinit` function to create an initial guess for the solution of the equation. Since the equation relates y'' to y, a reasonable guess is that the solution involves trigonometric functions. Use a mesh of five points in the interval of integration. The first and last values in the mesh are where the solver applies the boundary conditions.

The function for the initial guess accepts x as an input and returns a guess for the value of y_1 and y_2. The function is

```
function g = guess(x)
g = [sin(x)
     cos(x)];
end
```

```
xmesh = linspace(0,pi/2,5);
solinit = bvpinit(xmesh, @guess);
```

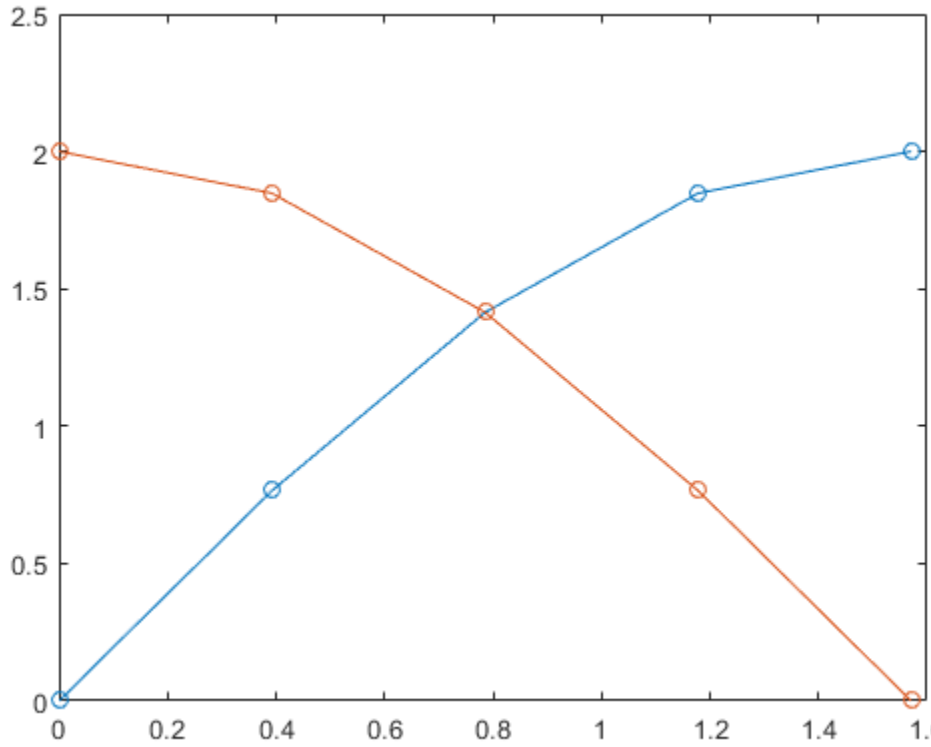
Solve Equation

Use `bvp5c` with the derivative function, boundary condition function, and initial guess to solve the problem.

```
sol = bvp5c(@bvpfcn, @bcfcn, solinit);
```

Plot Solution

```
plot(sol.x, sol.y, '-o')
```



Local Functions

Listed here are the local functions that `bvp5c` uses to solve the equation.

```
function dydx = bvpfcn(x,y) % equation to solve
dydx = zeros(2,1);
dydx = [y(2)
       -y(1)];
end
%--------------------------------
function res = bcfcn(ya,yb) % boundary conditions
res = [ya(1)
       yb(1)-2];
end
```

```
%--------------------------------
function g = guess(x) % initial guess for y and y'
g = [sin(x)
     cos(x)];
end
%--------------------------------
```

Compare bvp4c and bvp5c Solvers

Solve a BVP at a crude error tolerance with two different solvers and compare the results.

Consider the second-order ODE

$$y'' + \frac{2}{x}y' + \frac{1}{x^4}y = 0.$$

The equation is defined on the interval $\left[\frac{1}{3\pi}, 1\right]$ subject to the boundary conditions

$$y\left(\frac{1}{3\pi}\right) = 0,$$

$$y(1) = \sin(1).$$

To solve this equation in MATLAB®, you need to write a function that represents the equation as a system of first-order equations, write a function for the boundary conditions, set some option values, and create an initial guess. Then the BVP solver uses these four inputs to solve the equation.

Code Equation

With the substitutions $y_1 = y$ and $y_2 = y'$, you can rewrite the ODE as a system of first-order equations

$$y_1' = y_2,$$

$$y_2' = -\frac{2}{x}y_2 - \frac{1}{x^4}y_1.$$

The corresponding function is

```
function dydx = bvpfcn(x,y)
dydx = [y(2)
```

```
        -2*y(2)/x - y(1)/x^4];
end
```

Note: All functions are included at the end of the example as local functions.

Code Boundary Conditions

The boundary condition function requires that the boundary conditions are in the form $g(y(a), y(b)) = 0$. In this form, the boundary conditions are

$$y\left(\frac{1}{3\pi}\right) = 0,$$

$$y(1) - \sin(1) = 0.$$

The corresponding function is

```
function res = bcfcn(ya,yb)
res = [ya(1)
       yb(1)-sin(1)];
end
```

Set Options

Use `bvpset` to turn on the display of solver statistics, and specify crude error tolerances to highlight the difference in error control between the solvers. Also, for efficiency, specify the analytical Jacobian

$$J = \frac{\partial f_i}{\partial y} = \begin{bmatrix} \frac{\partial f_1}{\partial y_1} & \frac{\partial f_1}{\partial y_2} \\ \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial y_2} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{1}{x^4} & -\frac{2}{x} \end{bmatrix}.$$

The corresponding function that returns the value of the Jacobian is

```
function dfdy = jac(x,y)
dfdy = [0      1
       -1/x^4 -2/x];
end

opts = bvpset('FJacobian',@jac,'RelTol',0.1,'AbsTol',0.1,'Stats','on');
```

Create Initial Guess

Use `bvpinit` to create an initial guess of the solution. Specify a constant function as the initial guess with an initial mesh of 10 points in the interval $[1/3\pi, 1]$.

```
xmesh = linspace(1/(3*pi), 1, 10);
solinit = bvpinit(xmesh, [1; 1]);
```

Solve Equation

Solve the equation with both `bvp4c` and `bvp5c`.

```
sol4c = bvp4c(@bvpfcn, @bcfcn, solinit, opts);
```

```
The solution was obtained on a mesh of 9 points.
The maximum residual is  9.794e-02.
There were 157 calls to the ODE function.
There were 28 calls to the BC function.
```

```
sol5c = bvp5c(@bvpfcn, @bcfcn, solinit, opts);
```

```
The solution was obtained on a mesh of 11 points.
The maximum error is  6.742e-02.
There were 244 calls to the ODE function.
There were 29 calls to the BC function.
```

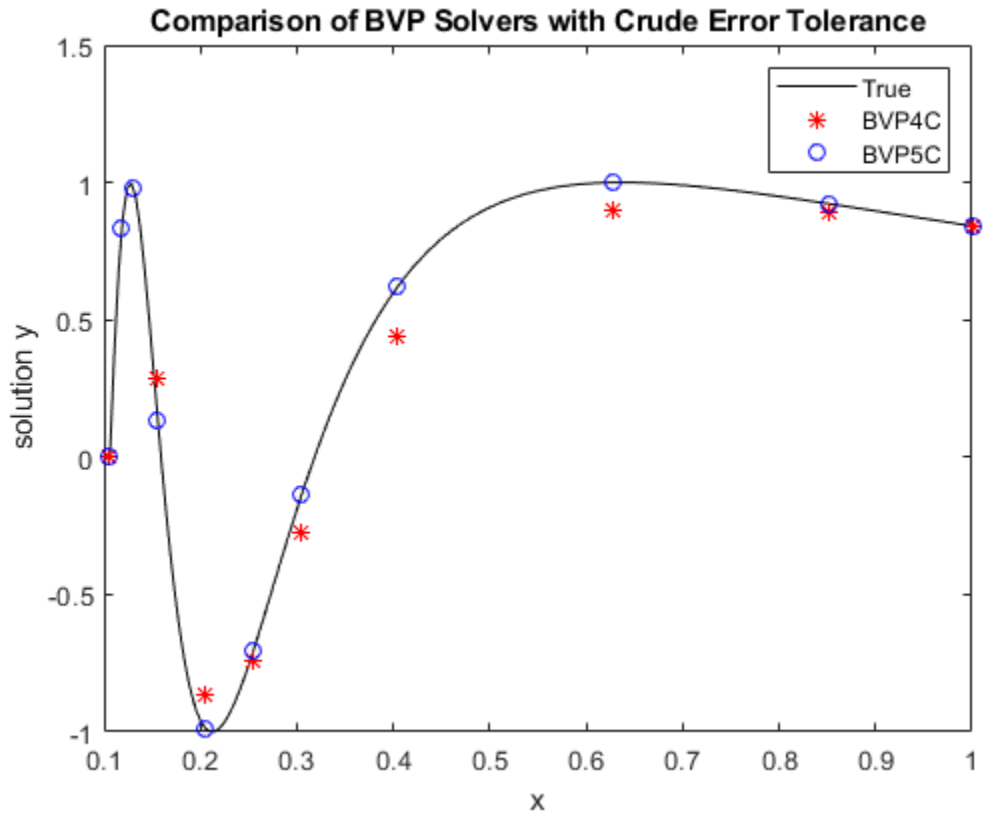
Plot Results

Plot the results of the two calculations for y_1 with the analytic solution for comparison. The analytic solution is

$$y_1 = \sin\left(\frac{1}{x}\right),$$

$$y_2 = -\frac{1}{x^2}\cos\left(\frac{1}{x}\right).$$

```
xplot = linspace(1/(3*pi),1,200);
yplot = [sin(1./xplot); -cos(1./xplot)./xplot.^2];

plot(xplot,yplot(1,:),'k',sol4c.x,sol4c.y(1,:),'r*',sol5c.x,sol5c.y(1,:),'bo')
title('Comparison of BVP Solvers with Crude Error Tolerance')
legend('True','BVP4C','BVP5C')
xlabel('x')
ylabel('solution y')
```



The plot confirms that `bvp5c` directly controls the true error in the calculation, while `bvp4c` controls it only indirectly. At more stringent error tolerances, this difference between the solvers is not as apparent.

Local Functions

Listed here are the local functions that the BVP solvers use to solve the problem.

```
function dydx = bvpfcn(x,y) % equation to solve
dydx = [y(2)
       -2*y(2)/x - y(1)/x^4];
end
%--------------------------------
function res = bcfcn(ya,yb) % boundary conditions
```

```
res = [ya(1)
       yb(1)-sin(1)];
end
%---------------------------------
function dfdy = jac(x,y) % analytical jacobian for f
dfdy = [0      1
       -1/x^4 -2/x];
end
%---------------------------------
```

Input Arguments

odefun — Functions to solve
function handle

Functions to solve, specified as a function handle that defines the functions to be integrated. `odefun` and `bcfun` must accept the same number of input arguments.

To code `odefun`, use the functional signature `dydx = odefun(x,y)` for a scalar `x` and column vector `y`. The return value `dydt` is a column vector of data type `single` or `double` that corresponds to $f(x,y)$. `odefun` must accept both input arguments `x` and `y`, even if one of the arguments is not used in the function.

For example, to solve $y' = 5y - 3$, use the function:

```
function dydt = odefun(t,y)
dydt = 5*y-3;
end
```

For a system of equations, the output of `odefun` is a vector. Each element in the vector is the solution to one equation. For example, to solve

$$y'_1 = y_1 + 2y_2$$
$$y'_2 = 3y_1 + 2y_2$$

use the function:

```
function dydt = odefun(t,y)
dydt = zeros(2,1);
dydt(1) = y(1)+2*y(2);
dydt(2) = 3*y(1)+2*y(2);
end
```

`bvp5c` also can solve problems with singularities in the solution on page 1-1128 or multipoint boundary conditions on page 1-1127.

Example: `sol = bvp5c(@odefun, @bcfun, solinit)`

Unknown Parameters

If the BVP being solved includes unknown parameters, you instead can use the functional signature `dydx = odefun(x,y,p)`, where `p` is a vector of parameter values. When you use this functional signature the BVP solver calculates the values of the unknown parameters starting from the initial guess for the parameter values provided in `solinit`.

Data Types: `function_handle`

`bcfun` — Boundary conditions
function handle

Boundary conditions, specified as a function handle that computes the residual error in the boundary conditions. `odefun` and `bcfun` must accept the same number of input arguments.

To code `bcfun`, use the functional signature `res = bcfun(ya,yb)` for column vectors `ya` and `yb`. The return value `res` is a column vector of data type `single` or `double` that corresponds to the residual value of the boundary conditions at the boundary points.

For example, if *y(a) = 1* and *y(b) = 0*, then the boundary condition function is

```
function res = bcfun(ya,yb)
res = [ya(1)-1
       yb(1)];
end
```

Since *y(a) = 1*, the residual value of `ya(1)-1` should be `0` at the point *x = a*. Similarly, since *y(b) = 0*, the residual value of `yb(1)` should be `0` at the point *x = b*.

The boundary points *x = a* and *x = b* where the boundary conditions are enforced are defined in the initial guess structure `solinit`. For two-point boundary value problems, `a = solinit.x(1)` and `b = solinit.x(end)`.

Example: `sol = bvp5c(@odefun, @bcfun, solinit)`

Unknown Parameters

If the BVP being solved includes unknown parameters, you instead can use the functional signature `res = bcfun(ya,yb,p)`, where `p` is a vector of parameter values. When you

use this functional signature the BVP solver calculates the values of the unknown parameters starting from the initial guess for the parameter values provided in `solinit`.

Data Types: `function_handle`

`solinit` — Initial guess of solution
structure

Initial guess of solution, specified as a structure. Use `bvpinit` to create `solinit`.

Unlike initial value problems, a boundary value problem can have no solution, a finite number of solutions, or infinitely many solutions. An important part of the process of solving a BVP is providing a guess for the required solution. The quality of this guess can be critical for the solver performance and even for a successful computation. For some guidelines on creating a good initial guess, see “Initial Guess of Solution”.

Example: `sol = bvp5c(@odefun, @bcfun, solinit)`

Data Types: `struct`

`options` — Option structure
structure

Option structure. Use the `bvpset` function to create or modify the options structure.

Example: `options = bvpset('RelTol',1e-5,'Stats','on')` specifies a relative error tolerance of `1e-5` and turns on the display of solver statistics.

Data Types: `struct`

Output Arguments

`sol` — Solution structure
structure

Solution structure. You can access the fields in `sol` with dot-indexing, such as `sol.field1`. The solution (`sol.x`,`sol.y`) is continuous on the interval of integration defined in the initial mesh `solinit.x` and has a continuous first derivative there. You can use `sol` with the `deval` function to evaluate the solution at other points in the interval.

The structure `sol` has these fields.

Field	Description
`x`	Mesh selected by `bvp5c`. This mesh typically contains different points than the initial mesh `solinit.x`.
`y`	Approximation to $y(x)$ at the mesh points of `sol.x`.
`yp`	Approximation to $y'(x)$ at the mesh points of `sol.x`.
`parameters`	Final values for the unknown parameters specified in `solinit.parameters`.
`solver`	`'bvp5c'`
`stats`	Computational cost statistics related to the solution: the number of mesh points, residual error, and number of calls to `odefun` and `bcfun`.

More About

Multipoint Boundary Value Problems

For multipoint boundary value problems, the boundary conditions are enforced at several points in the interval of integration.

`bvp5c` can solve multipoint boundary value problems where $a = a_0 < a_1 < a_2 < ... < a_n = b$ are boundary points in the interval $[a,b]$. The points $a_1,a_2,...,a_{n-1}$ represent interfaces that divide $[a,b]$ into regions. `bvp5c` enumerates the regions from left to right (from a to b), with indices starting from 1. In region k, $[a_{k-1},a_k]$, `bvp5c` evaluates the derivative as

```
yp = odefun(x,y,k)
```

In the boundary conditions function `bcfun(yleft,yright)`, `yleft(:,k)` is the solution at the left boundary of $[a_{k-1},a_k]$. Similarly, `yright(:,k)` is the solution at the right boundary of region k. In particular, `yleft(:,1) = y(a)` and `yright(:,end) = y(b)`.

When you create an initial guess with `bvpinit`, use double entries in `xinit` for each interface point. See the reference page for `bvpinit` for more information.

If `yinit` is a function, `bvpinit` calls `y = yinit(x,k)` to get an initial guess for the solution at `x` in region `k`. In the solution structure `sol` returned by `bpv4c`, `sol.x` has double entries for each interface point. The corresponding columns of `sol.y` contain the left and right solution at the interface, respectively.

See “Solve BVP with Multiple Boundary Conditions” for an example that solves a three-point boundary value problem.

Singular Boundary Value Problems

`bvp5c` solves a class of singular boundary value problems, including problems with unknown parameters `p`, of the form

$$y' = S\frac{y}{x} + f(x, y, p),$$
$$0 = bc(y(0), y(b), p).$$

The interval is required to be $[0, b]$ with $b > 0$. Often such problems arise when computing a smooth solution of ODEs that result from partial differential equations (PDEs) due to cylindrical or spherical symmetry. For singular problems, you specify the (constant) matrix `S` as the value of the `'SingularTerm'` option of `bvpset`, and `odefun` evaluates only $f(x,y,p)$. The boundary conditions and initial guess must be consistent with the necessary condition for smoothness $S \cdot y(0) = 0$.

See “Solve BVP with Singular Term” for an example that solves a singular boundary value problem.

Algorithms

`bvp5c` is a finite difference code that implements the four-stage Lobatto IIIa formula [1]. This is a collocation formula and the collocation polynomial provides a C^1-continuous solution that is fifth-order accurate uniformly in `[a,b]`. The formula is implemented as an implicit Runge-Kutta formula. Some of the differences between `bvp5c` and `bvp4c` are:

- `bvp5c` solves the algebraic equations directly. `bvp4c` uses analytical condensation.
- `bvp4c` handles unknown parameters directly. `bvp5c` augments the system with trivial differential equations for the unknown parameters.

References

[1] Shampine, L.F., and J. Kierzenka. "A BVP Solver that Controls Residual and Error." *J. Numer. Anal. Ind. Appl. Math.* Vol. 3(1-2), 2008, pp. 27–41.

See Also

`bvp4c` | `bvpget` | `bvpinit` | `bvpset` | `bvpxtend` | `deval`

Topics

"Solving Boundary Value Problems"

Introduced in R2006b

bvpget

Extract properties from options structure created with bvpset

Syntax

```
val = bvpget(options,'name')
val = bvpget(options,'name',default)
```

Description

`val = bvpget(options,'name')` extracts the value of the named property from the structure `options`, returning an empty matrix if the property value is not specified in `options`. It is sufficient to type only the leading characters that uniquely identify the property. Case is ignored for property names. `[]` is a valid `options` argument.

`val = bvpget(options,'name',default)` extracts the named property as above, but returns `val = default` if the named property is not specified in `options`. For example,

```
val = bvpget(options,'RelTol',1e-4);
```

returns `val = 1e-4` if the `RelTol` is not specified in `options`.

See Also

`bvp4c` | `bvp5c` | `bvpinit` | `bvpset` | `deval`

Introduced before R2006a

bvpinit

Form initial guess for boundary value problem solver

Syntax

```
solinit = bvpinit(x,yinit)
solinit = bvpinit(sol,[anew bnew])
solinit = bvpinit( ___ ,parameters)
```

Description

`solinit = bvpinit(x,yinit)` uses the initial mesh `x` and initial solution guess `yinit` to form an initial guess of the solution for a boundary value problem. You then can use the initial guess `solinit` as one of the inputs to `bvp4c` or `bvp5c` to solve the boundary value problem.

`solinit = bvpinit(sol,[anew bnew])` forms an initial guess for the solution on the interval `[anew bnew]`, where `sol` is a solution structure obtained from `bvp4c` or `bvp5c`. The new interval `[anew bnew]` must be larger than the previous interval on which `sol` is defined. The previous solution `sol` is extrapolated to the new interval.

`solinit = bvpinit( ___ ,parameters)` specifies a vector of initial guesses for parameters with unknown values in the boundary value problem. You can use this syntax with either of the previous input argument combinations.

Examples

Initial Guess of BVP Solution

Create an initial guess of the solution to a BVP, solve the BVP with `bvp4c`, and then extend the solution to a new domain.

Forming a good initial guess of the solution to a BVP problem is perhaps the most difficult part of solving the problem. BVP solutions are not necessarily unique, so the initial guess

can be the deciding factor in which of many solutions the solver returns. The initial guess should satisfy the boundary conditions, and the behavior inbetween should reflect your general expectations about the problem (whether the solution oscillates, is a simple linear function, and so on...).

Consider the differential equation

$$y'' = -y.$$

The equation is subject to the boundary conditions

$$y(0) = y(\pi) = 0.$$

The function that encodes the equation as a first-order system is

```
function dydx = bvpfun(x,y)
dydx = [y(2)
       -y(1)];
end
```

Similarly, the function that encodes the boundary conditions is

```
function res = bcfun(ya,yb)
res = [ya(1)
       yb(1)];
end
```

You either can include the required functions as local functions at the end of a file (as done here), or you can save them as separate, named files in a directory on the MATLAB path.

Initial Guess with Function Handle

You reasonably can expect the solution to the equation to be oscillatory, so sine and cosine functions are a good initial guess of the behavior of the solution and its derivative between the fixed boundary points.

```
function y = guess(x)
y = [sin(x)
     cos(x)];
end
```

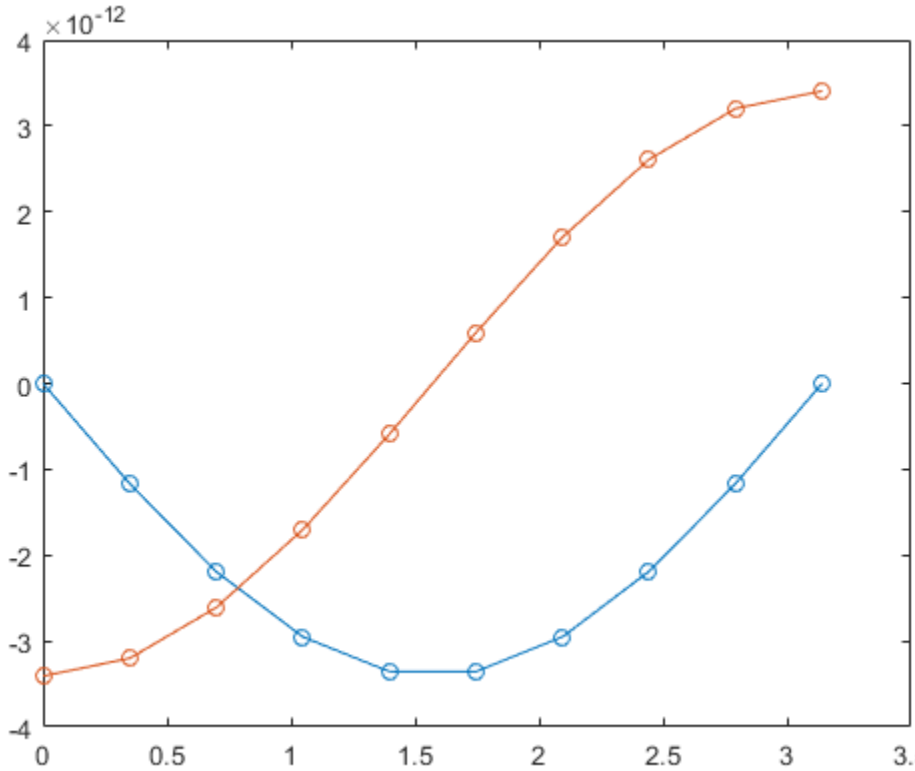
Create a solution structure using 10 equally spaced mesh points in the domain $[0, \pi]$ and the initial guess function.

```
xmesh = linspace(0,pi,10);
solinit = bvpinit(xmesh,@guess);
```

Solve BVP

Call `bvp4c` with the ode function, boundary conditions, and solution guess. Plot the result.

```
sol = bvp4c(@bvpfun, @bcfun, solinit);
plot(sol.x,sol.y,'-o')
```



Local Functions

Listed here are the local helper functions that the BVP solver `bvp4c` calls to calculate the solution. Alternatively, you can save these functions as their own files in a directory on the MATLAB path.

```
function dydx = bvpfun(x,y) % equation being solved
dydx = [y(2)
       -y(1)];
end
%--------------------------------------------
function res = bcfun(ya,yb) % boundary conditions
res = [ya(1)
       yb(1)];
end
%--------------------------------------------
function y = guess(x) % guess at solution behavior
y = [sin(x)
     cos(x)];
end
%--------------------------------------------
```

Extend BVP Solution with Extrapolation

Solve a BVP over an initial interval, and then iteratively extend the interval using each solution as the initial guess for the next interval.

Consider the equation

$y'' = y$.

As a first-order system, the equation becomes a system of two equations

$y_1' = y_2$,

$y_2' = y_1$.

The equation is initially defined on the interval $[0, 3]$ and is subject to the boundary conditions

$y(0) = 0$,

$y(3) = 1$.

The function that encodes the equation as a first-order system is

```
function dydx = bvpfun(x,y)
dydx = [y(2)
        y(1)];
end
```

Similarly, the function that encodes the boundary conditions is

```
function res = bcfun(ya,yb)
res = [ya(1)
       yb(1)-1];
end
```

You either can include the required functions as local functions at the end of a file (as done here), or you can save them as separate, named files in a directory on the MATLAB path.

Initial Guess

Use an exponential function as the initial guess for the solution. Since the equation has two solution components, write an initial guess function of the form `y = guess(x)` that returns a vector.

```
function y = guess(x)
y = [exp(x)
     exp(x)];
end
```

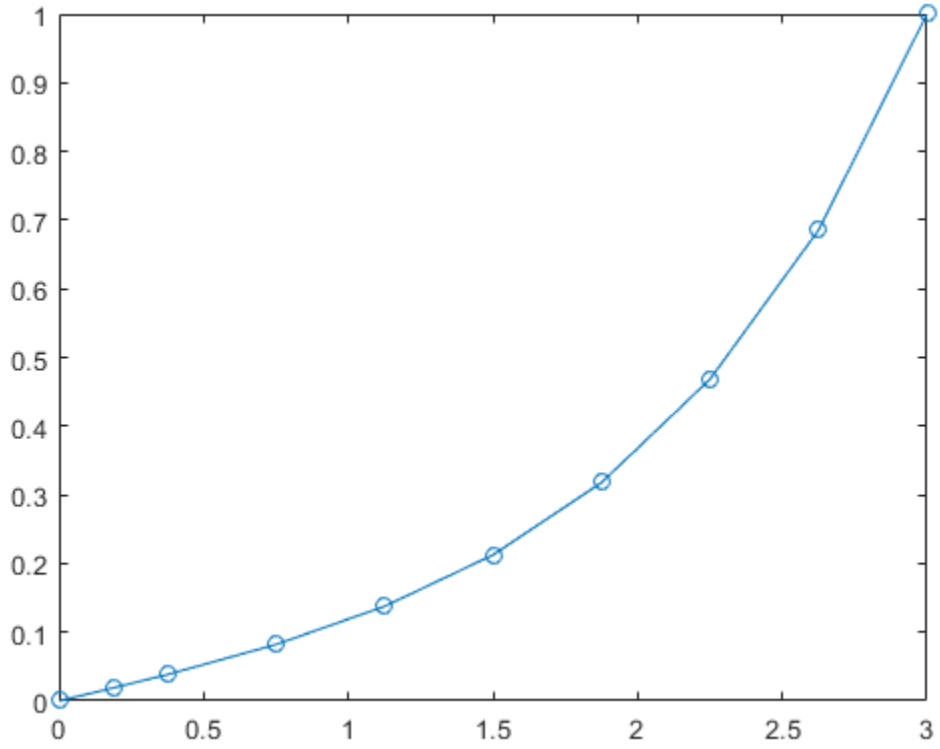
A mesh of five points is sufficient to capture the behavior of the guess function.

```
xmesh = linspace(0,3,5);
solinit = bvpinit(xmesh,@guess);
```

Solve Equation

Solve the equation in the initial interval $[0, 3]$ and plot the results for y_1.

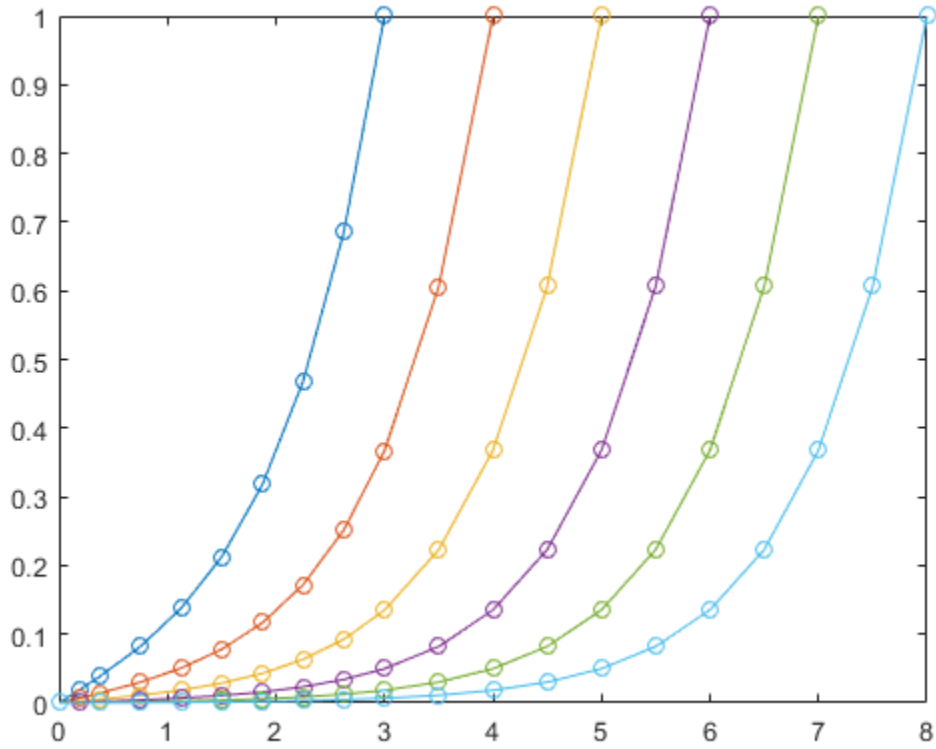
```
sol = bvp4c(@bvpfun, @bcfun, solinit);
plot(sol.x(1,:),sol.y(1,:),'-o')
```



Extend Interval

Now, use `bvpinit` to extend the interval of integration in a loop, solving and plotting each new problem. In each iteration, form the initial guess using the previous solution `sol` extrapolated to the new interval `[0 k]`. In each new problem, `bvp4c` enforces the boundary conditions at the new boundaries `[0 k]`.

```
hold on
for k = 4:8
    solinit = bvpinit(sol,[0 k]);
    sol = bvp4c(@bvpfun, @bcfun, solinit);
    plot(sol.x(1,:),sol.y(1,:),'-o')
end
```



This example shows a simplified version of *continuation*, a useful technique to solve BVPs by breaking the problem down into smaller intervals or simpler problems. For more examples of this technique, see:

- “Solve BVP Using Continuation”
- “Verify BVP Consistency Using Continuation”

Local Functions

Listed here are the local helper functions that the BVP solver `bvp4c` calls to calculate the solution. Alternatively, you can save these functions as their own files in a directory on the MATLAB path.

```
function dydx = bvpfun(x,y) % equation being solved
dydx = [y(2)
        y(1)];
end
%-------------------------------------------
function res = bcfun(ya,yb) % boundary conditions
res = [ya(1)
       yb(1)-1];
end
%-------------------------------------------
function y = guess(x) % guess at solution behavior
y = [exp(x)
     exp(x)];
end
%-------------------------------------------
```

Input Arguments

x — Initial mesh
vector

Initial mesh, specified as a vector. To solve the problem on the interval [*a*,*b*], specify `x(1)` as *a* and `x(end)` as *b*. The entries of `x` must be in increasing order (if *a* < *b*) or decreasing order (if *a* > *b*). The solver adapts this mesh to the solution (by adding, removing, and moving the mesh points), so a guess like `x = linspace(a,b,10)` often suffices. To handle difficult problems, place some mesh points where the solution changes rapidly.

- For *two-point* boundary value problems, the entries of `x` must be unique. That is, if *a* < *b*, the entries must satisfy `x(1)` < `x(2)` < ... < `x(end)`. If *a* > *b*, the entries must satisfy `x(1)` > `x(2)` > ... > `x(end)`.
- For *multipoint* boundary value problems, you can specify the points in [*a*,*b*] at which the boundary conditions apply, other than the endpoints *a* and *b*, by repeating their entries in `x`. For example, consider the vector

  ```
  x = [0 0.5 1 1 1.5 2];
  ```

  For this mesh, the boundary conditions apply at three points: the endpoints `0` and `2`, and the repeated entry `1`. In general, repeated entries represent boundary points between regions in [*a*,*b*]. The repeated entry `1` divides the interval `[0 2]` into two regions: `[0 1]` and `[1 2]`.

Example: `solinit = bvpinit(linspace(a,b,10),yinit)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

`yinit` — Initial guess of solution
vector | function

Initial guess of solution, specified as a vector or a function.

- **Vector** – For each component of the solution, `bvpinit` replicates the corresponding element of the vector as a constant guess across all mesh points. That is, `yinit(i)` is a constant guess for the `i`th component `yinit(i,:)` of the solution at all the mesh points in `x`.
- **Function** – For a given mesh point, the guess function must return a vector whose elements are guesses for the corresponding components of the solution. The function must be of the form

  ```
  y = guess(x)
  ```

  `x` is a mesh point and `y` is a vector whose length is the same as the number of components in the solution. For example, if `yinit` is a function, then at each mesh point `bvpinit` calls

  ```
  y(:,j) = guess(x(j))
  ```

  For multipoint boundary value problems, the guess function must be of the form

  ```
  y = guess(x, k)
  ```

  `y` is an initial guess for the solution at `x` in region `k`. The function must accept the input argument `k`, which is provided for flexibility in writing the guess function. However, the function is not required to use `k`.

Example: `solinit = bvpinit(x,)`

Example: `solinit = bvpinit(x,@guess)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `function_handle`
Complex Number Support: Yes

`parameters` — Initial guess for unknown parameter values
scalar | vector

Initial guess for unknown parameter values, specified as a scalar or vector.

Example: `solinit = bvpinit(x, yinit, [0 1 sqrt(2)])` specifies a vector of guesses for three unknown parameters.

Example: Type `edit mat4bvp` to see an example of a BVP with an unknown parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

`sol` — Prior solution
structure

Prior solution, specified as a solution structure returned by `bvp4c` or `bvp5c`. If `sol` contains parameters, they are copied to `solinit`.

Data Types: `struct`

Output Arguments

`solinit` — Initial guess of solution
structure

Initial guess of solution, returned as a structure. Use this structure as the third input to `bvp4c` or `bvp5c` to solve the boundary value problem.

See Also

`bvp4c` | `bvp5c` | `bvpget` | `bvpset` | `bvpxtend` | `deval`

Introduced before R2006a

bvpset

Create or alter options structure of boundary value problem

Syntax

```
options = bvpset('name1',value1,'name2',value2,...)
options = bvpset(oldopts,'name1',value1,...)
options = bvpset(oldopts,newopts)
bvpset
```

Description

`options = bvpset('name1',value1,'name2',value2,...)` creates a structure `options` that you can supply to the boundary value problem solver `bvp4c`, in which the named properties have the specified values. Any unspecified properties retain their default values. For all properties, it is sufficient to type only the leading characters that uniquely identify the property. `bvpset` ignores case for property names.

`options = bvpset(oldopts,'name1',value1,...)` alters an existing options structure `oldopts`. This overwrites any values in `oldopts` that are specified using name/value pairs and returns the modified structure as the output argument.

`options = bvpset(oldopts,newopts)` combines an existing options structure `oldopts` with a new options structure `newopts`. Any values set in `newopts` overwrite the corresponding values in `oldopts`.

`bvpset` with no input arguments displays all property names and their possible values, indicating defaults with braces `{}`.

You can use the function `bvpget` to query the `options` structure for the value of a specific property.

BVP Properties

`bvpset` enables you to specify properties for the boundary value problem solver `bvp4c`. There are several categories of properties that you can set:

- "Error Tolerance Properties" on page 1-1156
- "Vectorization" on page 1-1157
- "Analytical Partial Derivatives" on page 1-1158
- "Singular BVPs" on page 1-1159
- "Mesh Size Property" on page 1-1160
- "Solution Statistic Property" on page 1-1160

Error Tolerance Properties

Because `bvp4c` uses a collocation formula, the numerical solution is based on a mesh of points at which the collocation equations are satisfied. Mesh selection and error control are based on the residual of this solution, such that the computed solution $S(x)$ is the exact solution of a perturbed problem $S'(x) = f(x,S(x)) + res(x)$. On each subinterval of the mesh, a norm of the residual in the `i`th component of the solution, `res(i)`, is estimated and is required to be less than or equal to a tolerance. This tolerance is a function of the relative and absolute tolerances, `RelTol` and `AbsTol`, defined by the user.

$$\| (\mathrm{res}(i)/\max(\mathrm{abs}(f(i)), \mathrm{AbsTol}(i)/\mathrm{RelTol})) \| \leq \mathrm{RelTol}$$

The following table describes the error tolerance properties.

BVP Error Tolerance Properties

Property	Value	Description
`RelTol`	Positive scalar {`1e-3`}	A relative error tolerance that applies to all components of the residual vector. It is a measure of the residual relative to the size of $f(x,y)$. The default, `1e-3`, corresponds to 0.1% accuracy. The computed solution $S(x)$ is the exact solution of $S'(x) = F(x,S(x)) + \mathrm{res}(x)$. On each subinterval of the mesh, the residual $\mathrm{res}(x)$ satisfies $\| (\mathrm{res}(i)/\max(\mathrm{abs}(F(i)), \mathrm{AbsTol}(i)/\mathrm{RelTol})) \| \leq \mathrm{RelTol}$
`AbsTol`	Positive scalar or vector {`1e-6`}	Absolute error tolerances that apply to the corresponding components of the residual vector. `AbsTol(i)` is a threshold below which the values of the corresponding components are unimportant. If a scalar value is specified, it applies to all components.

Vectorization

The following table describes the BVP vectorization property. Vectorization of the ODE function used by `bvp4c` differs from the vectorization used by the ODE solvers:

- For `bvp4c`, the ODE function must be vectorized with respect to the first argument as well as the second one, so that `F([x1 x2 ...],[y1 y2 ...])` returns `[F(x1,y1) F(x2,y2)...]`.
- `bvp4c` benefits from vectorization even when analytical Jacobians are provided. For stiff ODE solvers, vectorization is ignored when analytical Jacobians are used.

Vectorization Properties

Property	Value	Description

`Vectorized`	`on` | `{off}`	Set on to inform `bvp4c` that you have coded the ODE function F so that `F([x1 x2 ...],[y1 y2 ...])` returns `[F(x1,y1) F(x2,y2) ...]`. That is, your ODE function can pass to the solver a whole array of column vectors at once. This enables the solver to reduce the number of function evaluations and may significantly reduce solution time.
		
		With the MATLAB array notation, it is typically an easy matter to vectorize an ODE function. In the `shockbvp` example shown previously, the shockODE function has been vectorized using colon notation into the subscripts and by using the array multiplication (`.*`) operator.
		
		`function dydx = shockODE(x,y,e)`
		`pix = pi*x;`
		`dydx = [ y(2,:)...`
		`-x/e.*y(2,:)-pi^2*cos(pix)-`
		`pix/e.*sin(pix)];`

Analytical Partial Derivatives

By default, the `bvp4c` solver approximates all partial derivatives with finite differences. `bvp4c` can be more efficient if you provide analytical partial derivatives $\partial f/\partial y$ of the differential equations, and analytical partial derivatives, $\partial bc/\partial ya$ and $\partial bc/\partial yb$, of the boundary conditions. If the problem involves unknown parameters, you must also provide partial derivatives, $\partial f/\partial p$ and $\partial bc/\partial p$, with respect to the parameters.

The following table describes the analytical partial derivatives properties.

BVP Analytical Partial Derivative Properties

Property	Value	Description
`FJacobian`	Function handle	A function handle that computes the analytical partial derivatives of $f(x,y)$. When solving $y' = f(x,y)$, set this property to `@fjac` if `dfdy = fjac(x,y)` evaluates the Jacobian $\partial f/\partial y$. If the problem involves unknown parameters p, `[dfdy,dfdp] = fjac(x,y,p)` must also return the partial derivative $\partial f/\partial p$. For problems with constant partial derivatives, set this property to the value of `dfdy` or to a cell array `{dfdy,dfdp}`.
`BCJacobian`	Function handle	A function handle that computes the analytical partial derivatives of $bc(ya,yb)$. For boundary conditions $bc(ya,yb)$, set this property to `@bcjac` if `[dbcdya,dbcdyb] = bcjac(ya,yb)` evaluates the partial derivatives $\partial bc/\partial ya$, and $\partial bc/\partial yb$. If the problem involves unknown parameters p, `[dbcdya,dbcdyb,dbcdp] = bcjac(ya,yb,p)` must also return the partial derivative $\partial bc/\partial p$. For problems with constant partial derivatives, set this property to a cell array `{dbcdya,dbcdyb}` or `{dbcdya,dbcdyb,dbcdp}`.

Singular BVPs

`bvp4c` can solve singular problems of the form

$$y' = S\frac{y}{x} + f(x, y, p)$$

posed on the interval $[0,b]$ where $b > 0$. For such problems, specify the constant matrix S as the value of `SingularTerm`. For equations of this form, `odefun` evaluates only the $f(x,y,p)$ term, where p represents unknown parameters, if any.

Singular BVP Property

Property	Value	Description
`SingularTerm`	Constant matrix	Singular term of singular BVPs. Set to the constant matrix S for equations of the form $$y' = S\frac{y}{x} + f(x, y, p)$$ posed on the interval $[0,b]$ where $b > 0$.

Mesh Size Property

`bvp4c` solves a system of algebraic equations to determine the numerical solution to a BVP at each of the mesh points. The size of the algebraic system depends on the number of differential equations (`n`) and the number of mesh points in the current mesh (`N`). When the allowed number of mesh points is exhausted, the computation stops, `bvp4c` displays a warning message and returns the solution it found so far. This solution does not satisfy the error tolerance, but it may provide an excellent initial guess for computations restarted with relaxed error tolerances or an increased value of `NMax`.

The following table describes the mesh size property.

BVP Mesh Size Property

Property	Value	Description
`NMax`	positive integer `{floor(1000/n)}`	Maximum number of mesh points allowed when solving the BVP, where `n` is the number of differential equations in the problem. The default value of `NMax` limits the size of the algebraic system to about 1000 equations. For systems of a few differential equations, the default value of `NMax` should be sufficient to obtain an accurate solution.

Solution Statistic Property

The `Stats` property lets you view solution statistics.

The following table describes the solution statistics property.

BVP Solution Statistic Property

Property	Value	Description
`Stats`	`on` | `{off}`	Specifies whether statistics about the computations are displayed. If the `stats` property is `on`, after solving the problem, `bvp4c` displays: • The number of points in the mesh • The maximum residual of the solution • The number of times it called the differential equation function `odefun` to evaluate $f(x,y)$ • The number of times it called the boundary condition function `bcfun` to evaluate $bc(y(a),y(b))$

Examples

To create an options structure that changes the relative error tolerance of `bvp4c` from the default value of `1e-3` to `1e-4`, enter

```
options = bvpset('RelTol',1e-4);
```

To recover the value of `'RelTol'` from `options`, enter

```
bvpget(options,'RelTol')

ans =

  1.0000e-004
```

See Also

`bvp4c` | `bvp5c` | `bvpget` | `bvpinit` | `deval`

Topics

"Create Function Handle"

Introduced before R2006a

bvpxtend

Form guess structure for extending boundary value solutions

Syntax

```
solinit = bvpxtend(sol,xnew,ynew)
solinit = bvpxtend(sol,xnew,extrap)
solinit = bvpxtend(sol,xnew)
solinit = bvpxtend(sol,xnew,ynew,pnew)
solinit = bvpxtend(sol,xnew,extrap,pnew)
```

Description

`solinit = bvpxtend(sol,xnew,ynew)` uses solution `sol` computed on `[a,b]` to form a solution guess for the interval extended to `xnew`. The extension point `xnew` must be outside the interval `[a,b]`, but on either side. The vector `ynew` provides an initial guess for the solution at `xnew`.

`solinit = bvpxtend(sol,xnew,extrap)` forms the guess at `xnew` by extrapolating the solution `sol`. `extrap` has three possible values:

- `'constant'` — `ynew` is a value nearer to end point of solution in `sol`.
- `'linear'` — `ynew` is a value at `xnew` of linear interpolant to the value and slope at the nearer end point of solution in `sol`.
- `'solution'` — `ynew` is the value of (cubic) solution in `sol` at `xnew`.

The value of `extrap` also can be a string.

`solinit = bvpxtend(sol,xnew)` uses the extrapolating solution where `extrap` is `'constant'`. If there are unknown parameters, values present in `sol` are used as the initial guess for parameters in `solinit`.

`solinit = bvpxtend(sol,xnew,ynew,pnew)` specifies a different guess `pnew`. `pnew` can be used with extrapolation, using the syntax `solinit = bvpxtend(sol,xnew,extrap,pnew)`. To modify parameters without changing the interval, use `[]` as place holder for `xnew` and `ynew`.

See Also

bvp4c | bvp5c | bvpinit

caldays

Calendar duration in days

Syntax

```
D = caldays(X)
```

Description

`D = caldays(X)` returns an array representing calendar days equivalent to the values in array `X`.

- If `X` is a numeric array, then `D` is a `calendarDuration` array with each element equal to the number of calendar days in the corresponding element of `X`. Calendar days account for Daylight Saving Time shifts when used in calendar calculations.
- If `X` is a `calendarDuration` array, then `D` is a `double` array with each element equal to the number of whole calendar days in the corresponding element of `X`.

Examples

Create Array of Calendar Days

```
X = magic(5);
D = caldays(X)
```

```
D = 5x5 calendarDuration array
   17d   24d    1d    8d   15d
   23d    5d    7d   14d   16d
    4d    6d   13d   20d   22d
   10d   12d   19d   21d    3d
   11d   18d   25d    2d    9d
```

Convert Calendar Durations to Calendar Days

Create an array of calendar durations. Then, convert each value to the equivalent number of whole calendar days.

```
X = caldays(8:10) + hours(1.2345)
```

```
X = 1x3 calendarDuration array
    8d 1h 14m 4.2s    9d 1h 14m 4.2s   10d 1h 14m 4.2s
```

```
D = caldays(X)
```

```
D = 1×3

     8     9    10
```

`caldays` returns a numeric array.

Current Time at Future Date

Add two calendar days to the current date and time.

```
t = datetime('now') + caldays(2)
```

```
t = datetime
   28-Aug-2019 23:52:19
```

Create Sequence of Dates

Create a sequence of consecutive dates beginning on March 18, 2014.

```
T = datetime([2014,03,18]) + caldays(0:4)
```

```
T = 1x5 datetime array
   18-Mar-2014   19-Mar-2014   20-Mar-2014   21-Mar-2014   22-Mar-2014
```

Create a sequence of dates beginning on March 18, 2014, spaced 2 days apart.

```
T = datetime([2014,03,18]) + caldays(0:2:8)
```

```
T = 1x5 datetime array
   18-Mar-2014   20-Mar-2014   22-Mar-2014   24-Mar-2014   26-Mar-2014
```

Input Arguments

X — Input array
numeric array | calendar duration array | logical array

Input array, specified as a numeric array, calendar duration array, or logical array. If X is a numeric array, it must contain only integer values. That is, you cannot create fractional calendar units.

Tips

- `caldays` creates days that account for Daylight Saving Time shifts when used in calendar calculations. To create exact fixed-length (24 hour) days, use the `days` function.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`calendarDuration` | `days` | `hours`

Introduced in R2014b

caldiff

Calendar math successive differences

Syntax

```
dt = caldiff(t)
dt = caldiff(t,components)
dt = caldiff(t,components,dim)
```

Description

`dt = caldiff(t)` calculates time differences between adjacent datetime values in `t` in terms of the calendar components years, months, days, and time. `caldiff` calculates differences along the first array dimension whose size does not equal 1.

- If `t` is a vector of length `m`, then `dt = caldiff(t)` returns a vector of length `m-1`. The elements of `dt` are the differences between adjacent elements of `t`.

  ```
  dt = [between(t(1),t(2)), between(t(2),t(3)),..., between(t(m-1),t(m))]
  ```

- If `t` is a nonvector `p`-by-`m` matrix, then `dt = caldiff(t)` returns a matrix of size `(p-1)`-by-`m`, whose elements are the differences between the rows of `t`.

  ```
  dt(:,I) = [between(t(1,I),t(2,I), between(t(2,I),t(3,I)), ...,
       between(t(p-1,I),t(p,I))]
  ```

`dt = caldiff(t,components)` finds the differences between successive datetimes in `t` in terms of the specified calendar or time components.

`dt = caldiff(t,components,dim)` finds the differences between successive datetimes along the dimension specified by `dim`.

Examples

Calendar Differences Between Datetime Values

Create a `datetime` array and then compute the differences between the values in terms of calendar components.

```
t = [datetime('yesterday');datetime('today');datetime('tomorrow')]
```

```
t = 3x1 datetime array
   26-Aug-2019
   27-Aug-2019
   28-Aug-2019
```

```
D = caldiff(t)
```

```
D = 2x1 calendarDuration array
   1d
   1d
```

Differences Using Specific Calendar Components

Create a `datetime` array and then compute the differences between the values in terms of days.

```
t = datetime('now') + calmonths(0:3)
```

```
t = 1x4 datetime array
Columns 1 through 3

   26-Aug-2019 23:50:24   26-Sep-2019 23:50:24   26-Oct-2019 23:50:24

Column 4

   26-Nov-2019 23:50:24
```

```
D = caldiff(t,'days')
```

```
D = 1x3 calendarDuration array
   31d   30d   31d
```

Computer the differences between the datetime values in terms of weeks and days.

```
D = caldiff(t,{'weeks','days'})
```

```
D = 1x3 calendarDuration array
   4w 3d   4w 2d   4w 3d
```

Input Arguments

t — Input date and time
`datetime` array

Input date and time, specified as a `datetime` array.

components — Calendar or time components
`'years'` | `'quarters'` | `'months'` | `'weeks'` | `'days'` | `'time'` | cell array of character vectors | string array

Calendar or time components, specified as one of the following character vectors, or a cell array or string array containing one or more of these values:

- `'years'`
- `'quarters'`
- `'months'`
- `'weeks'`
- `'days'`
- `'time'`

Except for `'time'`, the above components are flexible lengths of time. For example, one month represents a different length of time when added to a datetime in January than when added to a datetime in February.

`caldiff` operates on the calendar or time components in decreasing order, starting with the largest component.

In general, `t(2:m)` is not equal to `t(1:m-1) + dt`, unless you include `'time'` in `components`.

Example: `{'years','quarters'}`

Data Types: `char` | `cell` | `string`

`dim` — Dimension to operate along
positive integer

Dimension to operate along, specified as a positive integer. If no value is specified, the default is the first array dimension whose size does not equal 1.

Output Arguments

`dt` — Difference array
scalar | vector | matrix | multidimensional array

Difference array, returned as a scalar, vector, matrix, or multidimensional `calendarDuration` array.

Tips

- To compute successive differences between datetimes in `t1` and `t2` as exact, fixed-length units of hours, minutes, and seconds, use `diff(t)`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- The `dim` input must be specified. Use `caldiff(t,'',dim)` to use the default components.

For more information, see “Tall Arrays”.

See Also

`between` | `calendarDuration` | `diff` | `minus`

Introduced in R2014b

calendar

Calendar for specified month

Syntax

```
c = calendar
c = calendar(d)
c = calendar(y, m)
```

Description

`c = calendar` returns a 6-by-7 matrix containing a calendar for the current month. The calendar runs Sunday (first column) to Saturday.

`c = calendar(d)`, where `d` is a serial date number or text representing a date and time, returns a calendar for the specified month.

`c = calendar(y, m)`, where `y` and `m` are integers, returns a calendar for the specified month of the specified year.

If you do not specify an output argument, then `calendar` displays a calendar in the Command Window but does not return a value.

Examples

The command

```
calendar(1957,10)
```

reveals that the Space Age began on a Friday (on October 4, 1957, when Sputnik 1 was launched).

```
                    Oct 1957
     S     M    Tu     W    Th     F     S
     0     0     1     2     3     4     5
```

```
     6     7     8     9    10    11    12
    13    14    15    16    17    18    19
    20    21    22    23    24    25    26
    27    28    29    30    31     0     0
     0     0     0     0     0     0     0
```

Limitations

- MATLAB Online determines the current month from Coordinated Universal Time (UTC) rather than from local time.

See Also

`datenum`

Introduced before R2006a

calendarDuration

Lengths of time in variable-length calendar units

Description

The values in *calendar duration arrays* represent elapsed time in calendar units of variable length. For example, the number of days in a month depends on the particular month of the year. Calculations with calendar durations account for daylight saving time changes and leap years. Use calendar duration arrays to simplify calculations on datetime arrays that involve calendar units, such as days and months.

Creation

You can create calendar duration arrays that have specified time units using the `calyears`, `calquarters`, `calmonths`, `calweeks`, and `caldays` functions. For example, to create an array that has elapsed times of 1, 2, and 3 calendar months, use the `calmonths` function.

```
M = calmonths(1:3)
```

```
M =

  1×3 calendarDuration array

   1mo   2mo   3mo
```

You also can create a calendar duration array using the `calendarDuration` function, described below.

Syntax

```
L = calendarDuration(Y,M,D)
L = calendarDuration(Y,M,D,H,MI,S)
L = calendarDuration(Y,M,D,T)
L = calendarDuration(X)
```

```
L = calendarDuration( ___ ,'Format',displayFormat)
```

Description

`L = calendarDuration(Y,M,D)` creates an array of calendar durations from numeric arrays `Y`, `M`, and `D`, containing the number of years, months, and days, respectively.

`L = calendarDuration(Y,M,D,H,MI,S)` also includes hours, minutes, and seconds specified by `H`, `MI`, and `S`, respectively.

`L = calendarDuration(Y,M,D,T)` creates an array of calendar durations from numeric arrays containing the number of years, months, and days, and a duration array `T` containing elapsed times.

`L = calendarDuration(X)` creates an array of calendar durations from a numeric matrix.

`L = calendarDuration( ___ ,'Format',displayFormat)` additionally specifies `displayFormat` as the value of the `Format` property of `L`. The `Format` property changes the display of `L` but not its values. You can use this syntax with any of the arguments from the previous syntaxes.

Input Arguments

`Y,M,D` — Years, months, and days (as separate arguments)
numeric arrays

Years, months, and days, specified as numeric arrays. These arrays either must be the same size, or any can be a scalar. `Y,M,D` must contain only integer values.

Specifying month values greater than 12 is equivalent to specifying a number of years plus a number of months. For example, 25 months are equal to 2 years and 1 month. However, day values are not equivalent to a number of months because the number of days in a month is not fixed, and cannot be determined until you add the calendar duration to a specific datetime.

Example: `L = calendarDuration(2,10,24)` returns a calendar duration of 2 years, 10 months, and 24 days.

`H,MI,S` — Hours, minutes, and seconds (as separate arguments)
numeric arrays

Hours, minutes, and seconds, specified as numeric arrays. These arrays either must be the same size, or any can be a scalar. Specify fractional seconds as part of `S`. The `H` and `MI` arrays must contain only integer values.

Specifying month values greater than 12 is equivalent to specifying a number of years plus a number of months. For example, 25 months are equal to 2 years and 1 month. Minute values greater than 60 carry over to a number of hours. Second values greater than 60 carry over to a number of minutes. However, day values are not equivalent to a number of months because the number of days in a month is not fixed and cannot be determined until you add the calendar duration to a specific datetime. Similarly, hour values are not equivalent to a number of calendar days.

Example: `L = calendarDuration(2,10,24,12,45,07.451)` returns a calendar duration of 2 years, 10 months, 24 days, 12 hours, 45 minutes, and 7.451 seconds.

`T` — Elapsed times
duration array

Elapsed times, specified as a duration array. `T` either must be the same size as the `Y`, `M`, and `D` input arguments or be a scalar.

Example: `T = hours(5); L = calendarDuration(2,10,24,T)` adds a duration of 5 hours to `L`.

`X` — Years, months, days, and optionally, times
numeric matrix

Years, months, days, and optionally, times, specified as a numeric matrix. `X` must have either three or six columns. The first three columns contain the number of years, months, and days, respectively. If `X` has six columns, then the last three columns contain the number of hours, minutes, and seconds, respectively.

All columns must contain integer values, except for the sixth column. You can specify fractional seconds in the sixth column.

Example: `L = calendarDuration([2 10 24])` returns a calendar duration of 2 years, 10 months, and 24 days.

Example: `L = calendarDuration([2 10 24 12 45 07.451])` returns a calendar duration of 2 years, 10 months, 24 days, 12 hours, 45 minutes, and 7.451 seconds.

Properties

`Format` — Display format

`'ymdt'` (default) | character vector | string scalar

Display format for calendar durations, specified as a combination of the characters `y`, `q`, `m`, `w`, `d`, and `t`, in that order. The format must include `m`, `d`, and `t`.

Character	Unit	Required?
`y`	Years	no
`q`	Quarters (multiples of 3 months)	no
`m`	Months	yes
`w`	Weeks	no
`d`	Days	yes
`t`	Time (hours, minutes, and seconds)	yes

To specify the number of digits displayed for fractional seconds, use the `format` function.

If the value of a date or time component is zero, then it is not displayed.

Example: `L.Format = 'yqmdt'` displays each value in `L` as the number of calendar years, quarters, months, and days, along with its time component.

Examples

Use Calendar Durations with Datetime Values

Create a datetime value.

```
D = datetime(2017,7,1)
```

```
D = datetime
   01-Jul-2017
```

Create a datetime array in which each value starts on the first day of a different month. One convenient way to create such an array is to add an array of calendar months to `D`.

First, create an array of calendar months using the `calmonths` function.

```
C = calmonths(0:3)
```

```
C = 1x4 calendarDuration array
   0mo   1mo   2mo   3mo
```

Then, add `D` and `C`. Since `C` is a calendar duration array, this operation accounts for the fact that months can have different numbers of days.

```
M = D + C
```

```
M = 1x4 datetime array
   01-Jul-2017   01-Aug-2017   01-Sep-2017   01-Oct-2017
```

Due to leap years and Daylight Saving Time, calendar years, months, and days can have varying lengths. To accommodate these varying lengths of time, use calendar duration arrays for arithmetic operations on datetime arrays.

Create Calendar Duration

```
L = calendarDuration(1,3,15)
```

```
L = calendarDuration
   1y 3mo 15d
```

Specify Numeric and Duration Inputs That Overflow

Create a numeric array representing numbers of days.

```
D = [1 3;4 2]
```

```
D = 2×2
```

```
     1     3
     4     2
```

Create a duration array representing elapsed times in hours. One element specifies 25 hours, which is longer than one day.

```
T = hours([1 2; 25 12])
```

```
T = 2x2 duration array
    1 hr    2 hr
   25 hr   12 hr
```

Create a calendar duration array. Specify input arguments `D`, `T`, and scalar values for the year and month. The second input, 13, specifies more months than there are in one year.

```
L = calendarDuration(1,13,D,T)
```

```
L = 2x2 calendarDuration array
    2y 1mo 1d 1h 0m 0s    2y 1mo 3d 2h 0m 0s
   2y 1mo 4d 25h 0m 0s   2y 1mo 2d 12h 0m 0s
```

Month values greater than 12 carry over to years in the display. However, hour values greater than 24 do not carry over to days in the display. Due to Daylight Saving Time, the number of hours in a calendar day is not necessarily 24 hours.

Specify Display Format

Create a calendar duration array and specify a format that displays the values in terms of months, weeks, days, and time.

```
L = calendarDuration(1,1,5:9,'Format','mwdt')
```

```
L = 1x5 calendarDuration array
      13mo 5d      13mo 6d      13mo 1w   13mo 1w 1d   13mo 1w 2d
```

Since the format does not include `'y'` for years, the input values of 1 year and 1 month display as their sum, `13mo`. While `'t'` must be specified, the time component is not displayed if the hours, minutes, and seconds are all zero.

Tips

- For more information on functions that accept or return calendar duration arrays, see "Dates and Time".
- When you add a `calendarDuration` array that contains more than one unit to a datetime, MATLAB always adds the larger units first. If `t` is a datetime, then this command:

  ```
  t + calendarDuration(1,2,3)
  ```

  is the same as:

  ```
  t + calyears(1) + calmonths(2) + caldays(3)
  ```

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see "Tall Arrays".

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see "Run MATLAB Functions with Distributed Arrays" (Parallel Computing Toolbox).

See Also

`between` | `caldiff` | `cellstr` | `char` | `datetime` | `duration` | `iscalendarduration` | `split` | `string` | `time`

Topics

"Represent Dates and Times in MATLAB"
"Generate Sequence of Dates and Time"
"Compare Dates and Time"
"Date and Time Arithmetic"

Introduced in R2014b

split

Split calendar duration into numeric and duration units

Syntax

```
[X1,X2,...] = split(t,units)
```

Description

`[X1,X2,...] = split(t,units)` returns the calendar duration values specified in `t` as separate numeric arrays, one for each of the date or time units specified by `units`. The number of date and time units specified by `units` determines the number of output arguments.

Examples

Split Calendar Duration Array

Create a `calendarDuration` array.

```
T = calmonths(15:17) + caldays(8) + hours(1.2345)
```

```
T = 1x3 calendarDuration array
   1y 3mo 8d 1h 14m 4.2s   1y 4mo 8d 1h 14m 4.2s   1y 5mo 8d 1h 14m 4.2s
```

Get the month, day, and time.

```
[m,d,t] = split(T,{'months','days','time'})
```

```
m = 1×3

    15    16    17
```

```
d = 1×3

     8     8     8


t = 1x3 duration array
   01:14:04   01:14:04   01:14:04
```

Get the year, month, day, and time.

```
[y,m,d,t] = split(T,{'years','months','days','time'})
```

```
y = 1×3

     1     1     1


m = 1×3

     3     4     5


d = 1×3

     8     8     8


t = 1x3 duration array
   01:14:04   01:14:04   01:14:04
```

When you request both the year and month, `split` carries over month values greater than 12 to the year value.

Input Arguments

`t` — Input calendar duration
`calendarDuration` array

Input calendar duration, specified as a `calendarDuration` array.

Data Types: `calendarDuration`

units — Date and time units
character vector | cell array of character vectors | string array

Date and time units, specified as a character vector, a cell array of character vectors, or a string array. Specify `units` only when the first input argument is the `calendarDuration` array, `t`.

The units can be one or more of the values in the table.

Value	Units t Is Split Into
`'years'`	years
`'quarters'`	quarters
`'months'`	months
`'weeks'`	weeks
`'days'`	days
`'time'`	time, in the format hours:minutes:seconds

You must specify date and time units from largest to smallest. For example, `{'years','months'}` is valid, but `{'months','years'}` is not.

Example: `split(t,{'years','months','days'})`

Data Types: `char` | `cell` | `string`

Output Arguments

X1,X2,... — Output numeric and duration values
arrays

Output numeric and `duration` values, returned as arrays. `split` returns year, month, and day values in numeric arrays and time values in `duration` arrays.

Data Types: `double` | `duration`

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

See Also

`caldays` | `calmonths` | `calquarters` | `calweeks` | `calyears` | `time`

Introduced in R2014b

calllib

Call function in C shared library

Syntax

```
[x1,...,xN] = calllib(libname,funcname,arg1,...,argN)
```

Description

`[x1,...,xN] = calllib(libname,funcname,arg1,...,argN)` calls function `funcname` in C library `libname`, passing input arguments `arg1,...,argN`. The `calllib` function returns output values obtained from `funcname` in `x1,...,xN`.

Examples

Call `addStructByRef` Function

Load the library.

```
if ~libisloaded('shrlibsample')
   addpath(fullfile(matlabroot,'extern','examples','shrlib'))
   loadlibrary('shrlibsample')
end
```

Display function signature.

```
libfunctionsview shrlibsample
```

```
[double, c_structPtr] addStructByRef(c_structPtr)
```

The input argument is a pointer to a `c_struct` data type.

Create a MATLAB structure, `struct`:

```
struct.p1 = 4; struct.p2 = 7.3; struct.p3 = -290;
```

Call the function.

```
[res,st] = calllib('shrlibsample','addStructByRef',struct);
```

Display the results.

```
res
```

```
res =
         -279
```

Cleanup.

```
unloadlibrary shrlibsample
```

Input Arguments

`libname` — Name of shared library
character vector

Name of shared library, specified as a character vector. Do not include the path or file extension in `libname`.

If you call `loadlibrary` using the `alias` option, then you must use the alias name for the `libname` argument.

Data Types: `char`

`funcname` — Name of function in library
character vector

Name of function in library, specified as a character vector.

Data Types: `char`

`arg1,...,argN` — Input arguments
any type

Input arguments, 1 through N, required by `funcname` (if any), specified by any type. The `funcname` argument list specifies the argument type.

Output Arguments

`x1,...,xN` — Output arguments
any type

Output arguments, 1 through N, from `funcname` (if any), returned as any type. The `funcname` argument list specifies the argument type.

Limitations

- Use with libraries that are loaded using the `loadlibrary` function.

Tips

- MATLAB validates input argument types before calling `funcname`. If MATLAB displays error messages about data types, check the MATLAB function signature. For example, if `funcname` is in library `mylib` type:

  ```
  libfunctions('mylib','-full')
  ```

  To find `funcname`, scroll through the output. For more information, refer to your library documentation.

  When you call `funcname`, that function might display errors. For information about error messages, refer to your library documentation.

See Also

`libfunctionsview` | `loadlibrary`

Topics

Passing Arguments
"MATLAB Terminates Unexpectedly When Calling Function in Shared Library"
"No Matching Signature Error"

Introduced before R2006a

callSoapService

Send SOAP (Simple Object Access Protocol) message to endpoint

Note `callSoapService` will be removed in a future release. Use `matlab.wsdl.createWSDLClient` instead.

Syntax

```
response = callSoapService(endpoint,soapAction,message)
```

Description

`response = callSoapService(endpoint,soapAction,message)` sends `message` to the `soapAction` service at `endpoint`.

Examples

Retrieve Book Information from Library Database

This example assumes the library is on a local intranet and does not use an actual endpoint; therefore, you cannot run it.

Retrieve the name of the author of a book titled “In the Fall.” The relative path of the library service is `urn:LibraryCatalog`. To get the author's name, use the `getAuthor` function, which takes the book name as the input value. The `getAuthor` parameter is `nameToLookUp`. The XML data type for title is `{http://www.w3.org/2001/XMLSchema}string`. The SOAP message style is `rpc` by default.

Create the SOAP message.

```
message = createSoapMessage(...
  'urn:LibraryCatalog',...
  'getAuthor',...
```

```
  {'In the Fall'},...
  {'nameToLookUp'},...
  {'{http://www.w3.org/2001/XMLSchema}string'})
```

```
message =

[#document: null]
```

This response does not necessarily indicate that the message is valid, although certain input problems produce error messages.

Send the message to the server for processing, and get the author's name back. The server endpoint is `http://test/soap/services/LibraryCatalog`. The server method is `urn:LibraryCatalog#getAuthor`.

```
response = callSoapService(...
  'http://test/soap/services/LibraryCatalog',...
  'urn:LibraryCatalog#getAuthor',...
  message)
```

```
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body>
<getAuthorResponse xmlns="urn:LibraryCatalog">
<ns1:getAuthorReturn xmlns:ns1="http://latestversion.soap.test">
Kate Alvin
</ns1:getAuthorReturn>
</getAuthorResponse>
</soapenv:Body>
</soapenv:Envelope>
```

MATLAB returns the message in a single line, displayed here on separate lines for legibility.

Extract the author's name.

```
author = parseSoapResponse(response)
```

```
author = Kate Alvin
```

MATLAB automatically converted the XML string data type to `char`.

Input Arguments

`endpoint` — URL identifying a built-in HTTP service
string

URL identifying a built-in HTTP service, specified as a string.

Example: `'http://test/soap/services/LibraryCatalog'`

`soapAction` — Name of service
string

Name of service, specified as a string.

Example: `'urn:LibraryCatalog#getAuthor'`

`message` — Java® document object model (DOM)
string

Java document object model (DOM), specified as a string. Use `createSoapMessage` to create `message`.

Output Arguments

`response` — Result of `soapAction`
string

Result of `soapAction`, returned as a string. To read information in `response`, use the `parseSoapResponse` function.

See Also

`createSoapMessage` | `matlab.wsdl.createWSDLClient` | `parseSoapResponse` | `urlread` | `xmlread`

Introduced before R2006a

calmonths

Calendar duration in months

Syntax

```
M = calmonths(X)
```

Description

`M = calmonths(X)` returns an array representing calendar months equivalent to the values in `X`.

- If `X` is a numeric array, then `M` is a `calendarDuration` array with each element equal to the number of calendar months in the corresponding element of `X`.
- If `X` is a `calendarDuration` array, then `M` is a `double` array with each element equal to the number of whole calendar months in the corresponding element of `X`.

Examples

Create Array of Calendar Months

```
X = magic(4);
M = calmonths(X)
```

```
M = 4x4 calendarDuration array
   1y 4mo      2mo      3mo   1y 1mo
      5mo     11mo     10mo      8mo
      9mo      7mo      6mo       1y
      4mo   1y 2mo   1y 3mo      1mo
```

Convert Calendar Durations to Calendar Months

Create an array of calendar durations. Then, convert each value to the equivalent number of whole calendar months.

```
X = calmonths(15:17) + caldays(8) + hours(1.2345)
```

```
X = 1x3 calendarDuration array
   1y 3mo 8d 1h 14m 4.2s   1y 4mo 8d 1h 14m 4.2s   1y 5mo 8d 1h 14m 4.2s
```

```
M = calmonths(X)
```

```
M = 1×3

    15    16    17
```

Input Arguments

X — Input array
numeric array | calendar duration array | logical array

Input array, specified as a numeric array, calendar duration array, or logical array. If X is a numeric array, it must contain only integer values. That is, you cannot create fractional calendar units.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`calendarDuration` | `calweeks`

Introduced in R2014b

calquarters

Calendar duration in quarters

Syntax

```
Q = calquarters(X)
```

Description

`Q = calquarters(X)` returns an array representing calendar quarters equivalent to the values in array `X`.

- If `X` is a numeric array, then `Q` is a `calendarDuration` array with each element equal to the number of calendar quarters in the corresponding element of `X`.
- If `X` is a `calendarDuration` array, then `Q` is a `double` array with each element equal to the number of whole calendar quarters in the corresponding element of `X`.

  ```
  Q = fix(calmonths(t)/3)
  ```

Examples

Create Array of Calendar Quarters

```
X = magic(4);
Q = calquarters(X)
```

```
Q = 4x4 calendarDuration array
   16q    2q    3q   13q
    5q   11q   10q    8q
    9q    7q    6q   12q
    4q   14q   15q    1q
```

Convert Calendar Durations to Quarters

Create an array of calendar durations. Then, convert each value to the equivalent number of whole calendar quarters.

```
X = calmonths(2:2:6) + caldays(8)
```

```
X = 1x3 calendarDuration array
   2mo 8d   4mo 8d   6mo 8d
```

```
Q = calquarters(X)
```

```
Q = 1×3

     0     1     2
```

Input Arguments

X — Input array
numeric array | calendar duration array | logical array

Input array, specified as a numeric array, calendar duration array, or logical array. If X is a numeric array, it must contain only integer values. That is, you cannot create fractional calendar units.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`calendarDuration` | `calmonths`

Introduced in R2014b

calweeks

Calendar duration in weeks

Syntax

```
W = calweeks(X)
```

Description

`W = calweeks(X)` returns an array representing calendar weeks equivalent to the values in `X`.

- If `X` is a numeric array, then `W` is a `calendarDuration` array with each element equal to the number of calendar weeks in the corresponding element of `X`.
- If `X` is a `calendarDuration` array, then `calweeks` returns the number of whole weeks equivalent to each calendar duration in `X`.

Examples

Create Array of Calendar Weeks

```
X = magic(4);
W = calweeks(X)
```

```
W = 4x4 calendarDuration array
   16w    2w    3w   13w
    5w   11w   10w    8w
    9w    7w    6w   12w
    4w   14w   15w    1w
```

Convert Calendar Durations to Calendar Weeks

Create an array of calendar durations. Then, convert each value to the equivalent number of whole calendar weeks.

```
X = caldays(15:17) + hours(1.2345)
```

```
X = 1x3 calendarDuration array
   15d 1h 14m 4.2s   16d 1h 14m 4.2s   17d 1h 14m 4.2s
```

```
W = calweeks(X)
```

```
W = 1×3

     2     2     2
```

Input Arguments

X — Input array
numeric array | calendar duration array | logical array

Input array, specified as a numeric array, calendar duration array, or logical array. If X is a numeric array, it must contain only integer values. That is, you cannot create fractional calendar units.

Output Arguments

W — Calendar weeks
scalar | vector | matrix | multidimensional array

Calendar weeks, returned as a scalar, vector, matrix, or multidimensional array. W is the same size as X. The data type of W depends on X.

- If X is a numeric array, then W is an array of calendar durations in units of equivalent flexible-length calendar weeks.
- If X is a `calendarDuration` array, then W is a `double` array of integer values representing whole calendar weeks.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`caldays` | `calendarDuration`

Introduced in R2014b

calyears

Calendar duration in years

Syntax

```
Y = calyears(X)
```

Description

`Y = calyears(X)` returns an array representing calendar years equivalent to the values in `X`. Calendar years account for leap days when used in calendar calculations.

- If `X` is a numeric array, then `Y` is a `calendarDuration` array with each element equal to the number of calendar years in the corresponding element of `X`.
- If `X` is a `calendarDuration` array, then `calyears` returns the number of whole years equivalent to each calendar duration in `X`.

Examples

Create Array of Calendar Years

```
X = magic(4);
Y = calyears(X)
```

```
Y = 4x4 calendarDuration array
   16y    2y    3y   13y
    5y   11y   10y    8y
    9y    7y    6y   12y
    4y   14y   15y    1y
```

Convert Calendar Durations to Calendar Years

Create an array of calendar durations. Then, convert each value to the equivalent number of whole calendar years.

```
X = calmonths(21:25) + caldays(8)
```

```
X = 1x5 calendarDuration array
    1y 9mo 8d   1y 10mo 8d   1y 11mo 8d        2y 8d    2y 1mo 8d
```

```
Y = calyears(X)
```

```
Y = 1×5

     1     1     1     2     2
```

Input Arguments

X — Input array
numeric array | calendar duration array | logical array

Input array, specified as a numeric array, calendar duration array, or logical array. If X is a numeric array, it must contain only integer values. That is, you cannot create fractional calendar units.

Output Arguments

Y — Calendar years
scalar | vector | matrix | multidimensional array

Calendar years, returned as a scalar, vector, matrix, or multidimensional array. Y is the same size as X. The data type of Y depends on X.

- If X is a numeric array, then Y is an array of calendar durations in units of equivalent flexible-length calendar years.
- If X is a `calendarDuration` array, then Y is a `double` array of integer values representing whole calendar years.

Tips

- `calyears` creates years that account for leap days when used in calendar calculations. To create exact fixed-length (365.2425 day) years, use the `years` function.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`calendarDuration` | `calmonths` | `years`

Introduced in R2014b

camdolly

Move camera position and target

Syntax

```
camdolly(dx,dy,dz)
camdolly(dx,dy,dz,'targetmode')
camdolly(dx,dy,dz,targetmode,coordsys)
camdolly(ax,...)
```

Description

`camdolly(dx,dy,dz)` moves the camera position and the camera target by the specified amounts `dx`, `dy`, and `dz`.

`camdolly(dx,dy,dz,'targetmode')` uses the `targetmode` argument to determine how the camera moves:

- `'movetarget'` (default) — Move both the camera and the target.
- `'fixtarget'` — Move only the camera.

`camdolly(dx,dy,dz,targetmode,coordsys)` uses the `coordsys` argument to determine how MATLAB interprets `dx`, `dy`, and `dz`:

- `'camera'` (default) — Move in the coordinate system of the camera. `dx` moves left/right, `dy` moves down/up, and `dz` moves along the viewing axis. MATLAB normalizes the units to the scene.

  For example, setting `dx` to 1 moves the camera to the right, which pushes the scene to the left edge of the box formed by the axes position rectangle. A negative value moves the scene in the other direction. Setting `dz` to 0.5 moves the camera to a position halfway between the camera position and the camera target.
- `'pixels'` — Interpret `dx` and `dy` as pixel offsets. This option ignores `dz`.
- `'data'` — Interpret `dx`, `dy`, and `dz` as offsets in axes data coordinates.

`camdolly(ax,...)` operates on the axes identified by the first argument, `ax`. When you do not specify an axes object, `camdolly` operates on the current axes.

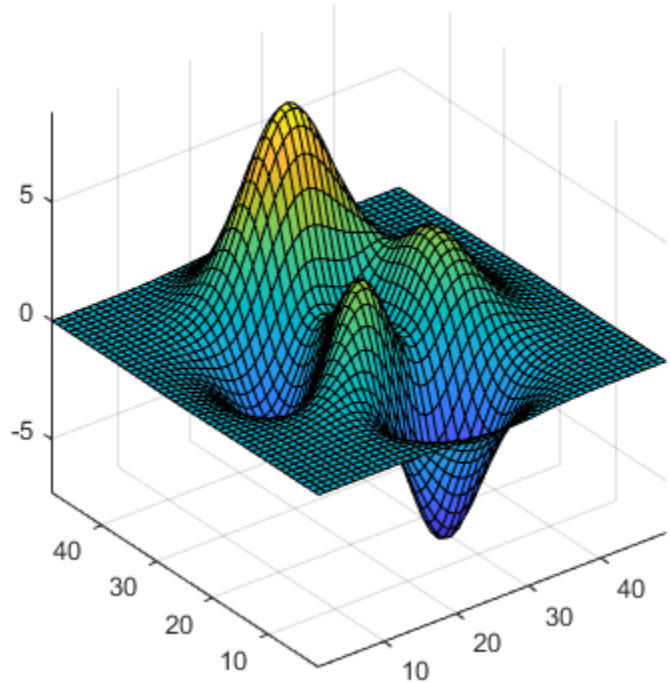
`camdolly` sets the axes `CameraPosition` and `CameraTarget` properties, which in turn sets the `CameraPositionMode` and `CameraTargetMode` properties to `manual`.

Examples

Move Camera Position and Target

Move the camera along the *x*-axis and *y*-axis in a series of steps.

```
surf(peaks)
axis vis3d
t = 0:pi/20:4*pi;
dx = sin(t)./40;
dy = cos(t)./40;
for i = 1:length(t)
    camdolly(dx(i),dy(i),0)
    drawnow
end
```



See Also

axes | campos | camproj | camtarget | camup | camva

Topics

"Camera Graphics Terminology"

Introduced before R2006a

cameratoolbar

Control camera toolbar programmatically

Syntax

```
cameratoolbar
cameratoolbar('NoReset')
cameratoolbar('SetMode',mode)
cameratoolbar('SetCoordSys',coordsys)
cameratoolbar('Show')
cameratoolbar('Hide')
cameratoolbar('Toggle')
cameratoolbar('ResetCameraAndSceneLight')
cameratoolbar('ResetCamera')
cameratoolbar('ResetSceneLight')
cameratoolbar('ResetTarget')
h = cameratoolbar(...)
mode = cameratoolbar('GetMode')
paxis = cameratoolbar('GetCoordsys')
tf = cameratoolbar('GetVisible')
cameratoolbar('Close')
cameratoolbar(fig,...)
```

Description

`cameratoolbar` creates a toolbar that enables interactive manipulation of the axes camera and light when you drag the mouse on the figure window. Several axes camera properties are set when the toolbar is initialized.

`cameratoolbar('NoReset')` creates the toolbar without setting any camera properties.

`cameratoolbar('SetMode',mode)` sets the toolbar mode (depressed button). `mode` can be `'orbit'`, `'orbitscenelight'`, `'pan'`, `'dollyhv'`, `'dollyfb'`, `'zoom'`, `'roll'`, `'nomode'`. For descriptions of the various modes, see “Camera Toolbar”. You can also set these modes using the toolbar, by clicking the respective buttons.

`cameratoolbar('SetCoordSys',coordsys)` sets the principal axis of the camera motion. `coordsys` can be: `'x'`, `'y'`, `'z'`, `'none'`.

`cameratoolbar('Show')` shows the toolbar on the current figure.

`cameratoolbar('Hide')` hides the toolbar on the current figure.

`cameratoolbar('Toggle')` toggles the visibility of the toolbar.

`cameratoolbar('ResetCameraAndSceneLight')` resets the current camera and scenelight.

`cameratoolbar('ResetCamera')` resets the current camera.

`cameratoolbar('ResetSceneLight')` resets the current scenelight.

`cameratoolbar('ResetTarget')` resets the current camera target.

`h = cameratoolbar(...)` returns the handle to the toolbar.

`mode = cameratoolbar('GetMode')` returns the current mode.

`paxis = cameratoolbar('GetCoordsys')` returns the current principal axis.

`tf = cameratoolbar('GetVisible')` returns the visibility of the toolbar (`1` if visible, `0` if not visible).

`cameratoolbar('Close')` removes the toolbar from the current figure.

`cameratoolbar(fig,...)` specifies the figure to operate on by passing the figure handle as the first argument.

In general, the use of OpenGL hardware improves rendering performance.

Alternatives

Display the toolbar by selecting **Camera Toolbar** from the figure window's **View** menu.

See Also

`rotate3d` | `zoom`

Topics

"Camera Toolbar"

Introduced before R2006a

camlight

Create or move light object in camera coordinates

Syntax

```
camlight('headlight')
camlight('right')
camlight('left')
camlight
camlight(az,el)
camlight(...,'style')
camlight(lgt,...)
camlight(ax,...)
lgt = camlight(...)
```

Description

`camlight('headlight')` creates a light at the camera position.

`camlight('right')` creates a light right and up from camera.

`camlight('left')` creates a light left and up from camera.

`camlight` with no arguments is the same as `camlight('right')`.

`camlight(az,el)` creates a light at the specified azimuth (`az`) and elevation (`el`) with respect to the camera position. The camera target is the center of rotation and `az` and `el` are in degrees.

`camlight(...,'style')` defines the style argument using one of two values:

- `local` (default) — The light is a point source that radiates from the location in all directions.
- `infinite` — The light shines in parallel rays.

`camlight(lgt,...)` uses the light specified by `lgt`.

`camlight(ax,...)` uses the axes specified by `ax` instead of the current axes.

`lgt = camlight(...)` returns the light object.

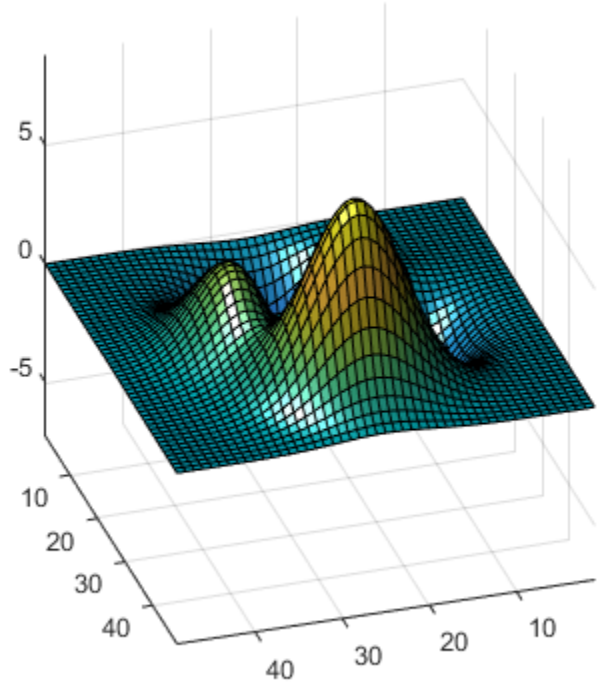
`camlight` sets the light object `Position` and `Style` properties. A light created with `camlight` does not track the camera. In order for the light to stay in a constant position relative to the camera, call `camlight` whenever you move the camera.

Examples

Move Light Object

Create a light positioned to the left of the camera and then reposition the light each time the camera moves.

```
surf(peaks)
axis vis3d
h = camlight('left');
for i = 1:20
   camorbit(10,0)
   camlight(h,'left')
   pause(.1)
end
```



See Also

`light` | `lightangle`

Topics

"Lighting Overview"

Introduced before R2006a

camlookat

Position camera to view object or group of objects

Syntax

```
camlookat(object_handles)
camlookat(axes_handle)
camlookat
```

Description

`camlookat(object_handles)` views the objects identified in the vector `object_handles`. The vector can contain the handles of axes `Children`.

`camlookat(axes_handle)` views the objects that are children of the axes identified by `axes_handle`.

`camlookat` views the objects that are in the current axes by moving the camera position and camera target while preserving the relative view direction and camera view angle. The viewed object (or objects) roughly fill the axes position rectangle. To change the view, `camlookat` sets the axes `CameraPosition` and `CameraTarget` properties.

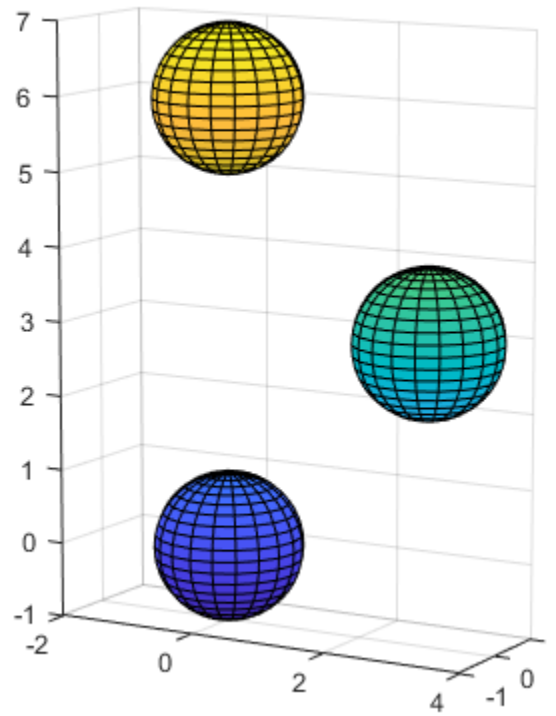
Examples

Position Camera to View Object

Create three spheres. Then set the data aspect ratio, the view, and the projection type.

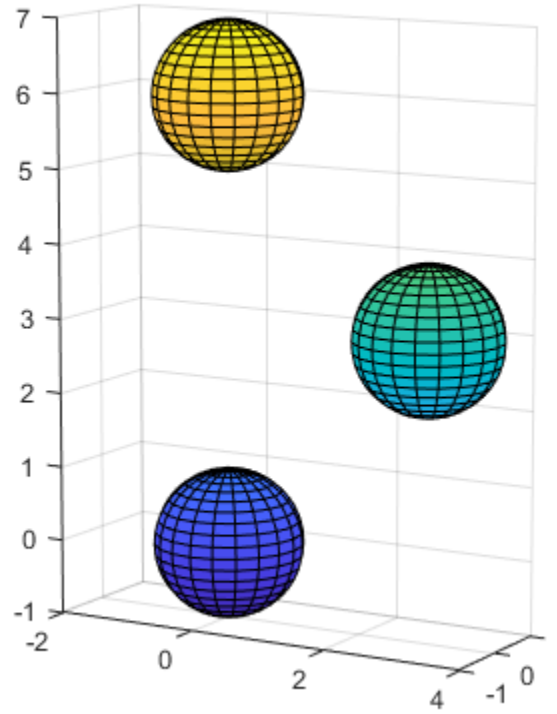
```
[x,y,z] = sphere;
s1 = surf(x,y,z);
hold on
s2 = surf(x+3,y,z+3);
s3 = surf(x,y,z+6);
```

```
daspect([1 1 1])
view(30,10)
camproj perspective
```



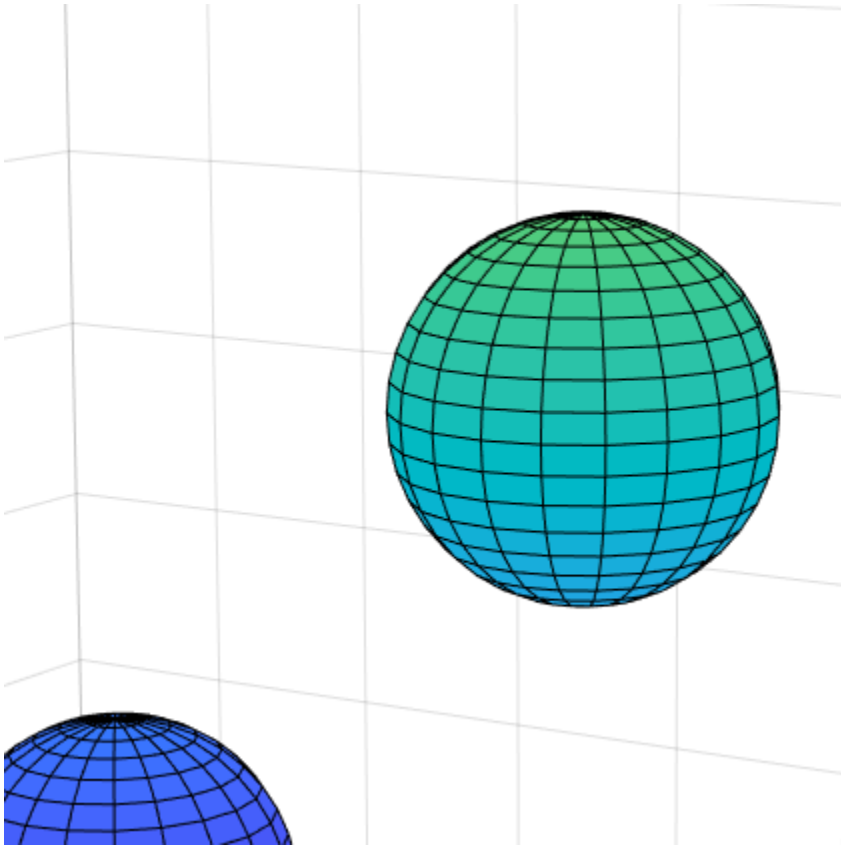
Compose the scene around sphere `s1`.

```
camlookat(s1)
```



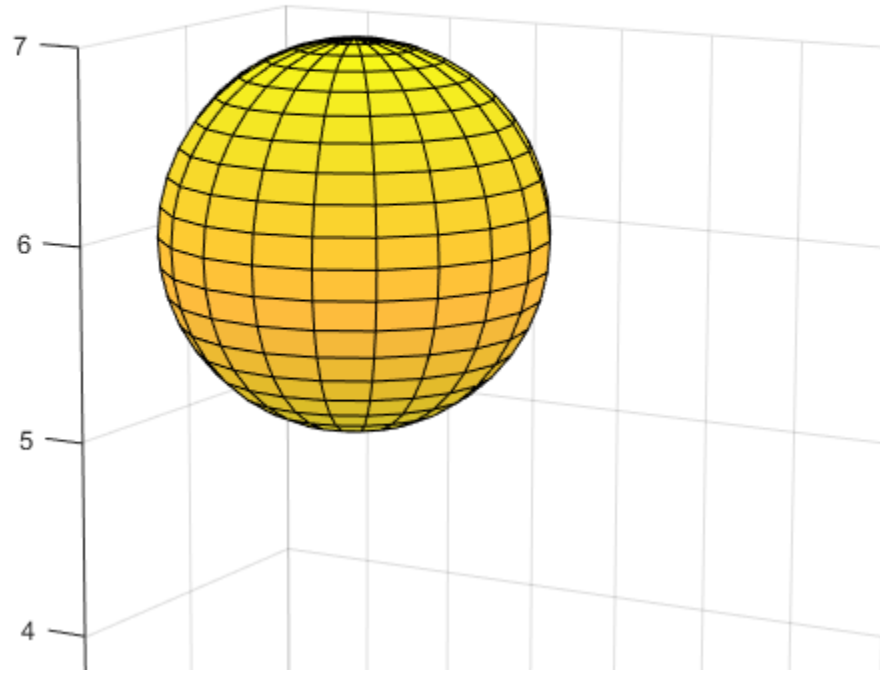
Compose the scene around sphere `s2`.

```
camlookat(s2)
```



Compose the scene around sphere `s3`.

```
camlookat(s3)
```



See Also

`campos` | `camtarget`

Topics

"Camera Graphics Terminology"

Introduced before R2006a

camorbit

Rotate camera position around camera target

Syntax

```
camorbit(dtheta,dphi)
camorbit(dtheta,dphi,'coordsys')
camorbit(dtheta,dphi,'coordsys','direction')
camorbit(axes_handle,...)
```

Description

`camorbit(dtheta,dphi)` rotates the camera position around the camera target by the amounts specified in `dtheta` and `dphi` (both in degrees). `dtheta` is the horizontal rotation and `dphi` is the vertical rotation.

`camorbit(dtheta,dphi,'coordsys')` rotates the camera position around the camera target, using the *`coordsys`* argument to determine the center of rotation. *`coordsys`* can take on two values:

- `data` (default) — Rotate the camera around an axis defined by the camera target and the `direction` (default is the positive *z* direction).
- `camera` — Rotate the camera about the point defined by the camera target.

`camorbit(dtheta,dphi,'coordsys','direction')` defines the axis of rotation for the data coordinate system using the `direction` argument in conjunction with the camera target. Specify `direction` as a three-element vector containing the *x*-, *y*-, and *z*-components of the direction or one of the options, `x`, `y`, or `z`, to indicate `[1 0 0]`, `[0 1 0]`, or `[0 0 1]` respectively.

`camorbit(axes_handle,...)` operates on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camorbit` operates on the current axes.

The behavior of `camorbit` differs from the `rotate3d` function in that while the `rotate3d` tool modifies the `View` property of the axes, the `camorbit` function fixes the

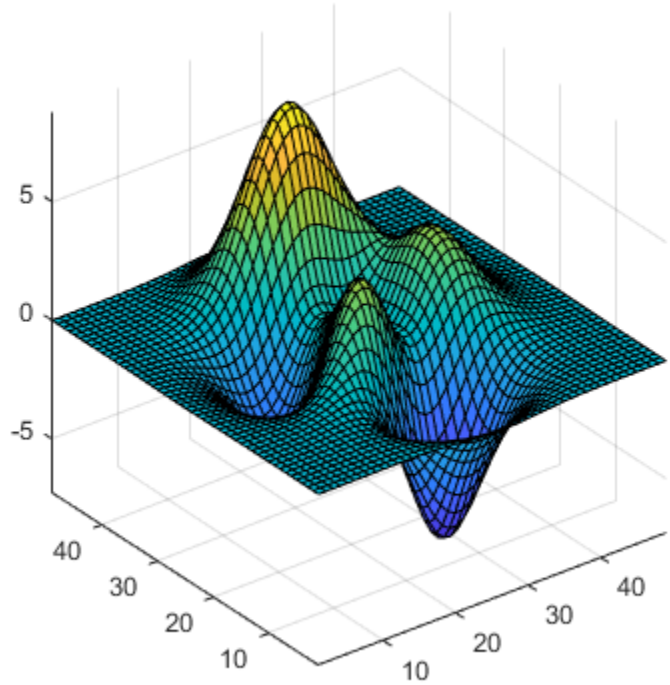
aspect ratio and modifies the `CameraTarget`, `CameraPosition`, and `CameraUpVector` properties of the axes. See Axes for more information on all axes properties.

Examples

Rotate Camera Horizontally

Rotate the camera horizontally about a line defined by the camera target point and a direction that is parallel to the *y*-axis. Visualize this rotation as a cone formed with the camera target at the apex and the camera position forming the base.

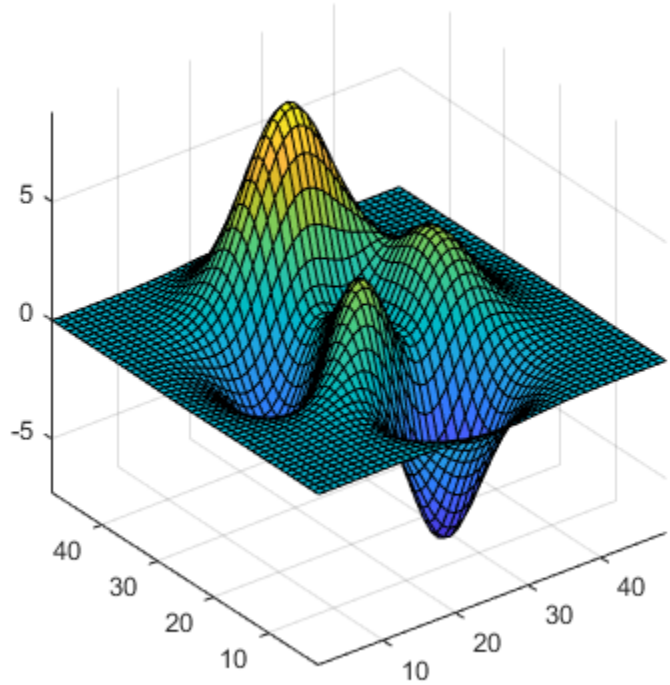
```
surf(peaks)
axis vis3d

for i = 1:36
   camorbit(10,0,'data',[0 1 0])
   drawnow
end
```



Rotate Camera Along Circle Around Axes

Rotate in the camera coordinate system to orbit the camera around the axes along a circle while keeping the center of a circle at the camera target.

```
surf(peaks)
axis vis3d

for i=1:36
   camorbit(10,0,'camera')
   drawnow
end
```



Alternatives

Enable 3-D rotation from the figure **Tools** menu or the figure toolbar.

See Also

`axes` | `axis` | `camdolly` | `campan` | `camroll` | `camzoom`

Topics

"Camera Graphics Terminology"

Introduced before R2006a

campan

Rotate camera target around camera position

Syntax

```
campan(dtheta,dphi)
campan(dtheta,dphi,coordsys)
campan(dtheta,dphi,coordsys,direction)
campan(ax,...)
```

Description

`campan(dtheta,dphi)` rotates the camera target of the current axes around the camera position by the amounts specified in `dtheta` and `dphi` (both in degrees). `dtheta` is the horizontal rotation and `dphi` is the vertical rotation.

`campan(dtheta,dphi,coordsys)` determine the center of rotation using the `coordsys` argument. It can take on two values:

- `'data'` (default) — Rotate the camera target around an axis defined by the camera position and the `direction` (default is the positive *z* direction)
- `'camera'` — Rotate the camera about the point defined by the camera target.

`campan(dtheta,dphi,coordsys,direction)` defines the axis of rotation for the data coordinate system using the `direction` argument with the camera position. Specify `direction` as a three-element vector containing the *x*-, *y*-, and *z*-components of the direction or one of the options, `'x'`, `'y'`, or `'z'`, to indicate `[1 0 0]`, `[0 1 0]`, or `[0 0 1]` respectively.

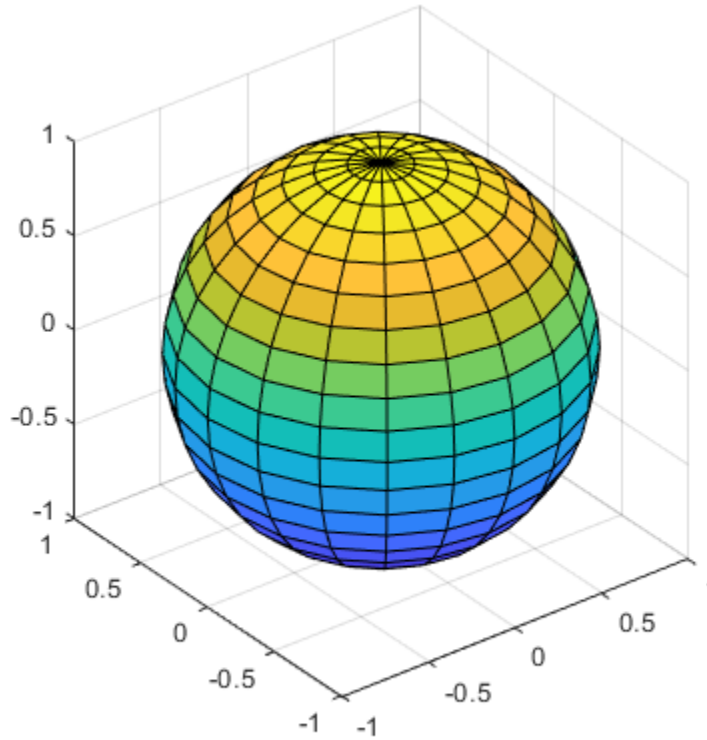
`campan(ax,...)` operates on the axes identified by the first argument, `ax`. When you do not specify an axes object, `campan` operates on the current axes.

Examples

Rotate Camera Target

Move the camera target to pan the object in a circular motion.

```
sphere;
axis vis3d
hPan = sin(-pi:1:pi);
vPan = cos(-pi:1:pi);
for k = 1:length(hPan)
   campan(hPan(k),vPan(k))
   pause(.1)
end
```



See Also

axes | camdolly | camorbit | camroll | camtarget | camzoom

Topics

"Camera Graphics Terminology"

Introduced before R2006a

campos

Set or query camera position

Syntax

```
campos
campos([camera_position])
campos('mode')
campos('auto')
campos('manual')
campos(ax,...)
```

Description

`campos` returns the camera position in the current axes.

`campos([camera_position])` sets the position of the camera in the current axes to the specified value. Specify the position as a three-element vector containing the *x*-, *y*-, and *z*-coordinates of the desired location in the data units of the axes.

`campos('mode')` returns the value of the camera position mode, which can be either `auto` (the default) or `manual`.

`campos('auto')` sets the camera position mode to `auto`.

`campos('manual')` sets the camera position mode to `manual`.

`campos(ax,...)` performs the set or query on the axes identified by the first argument, `ax`. When you do not specify an axes object, `campos` operates on the current axes.

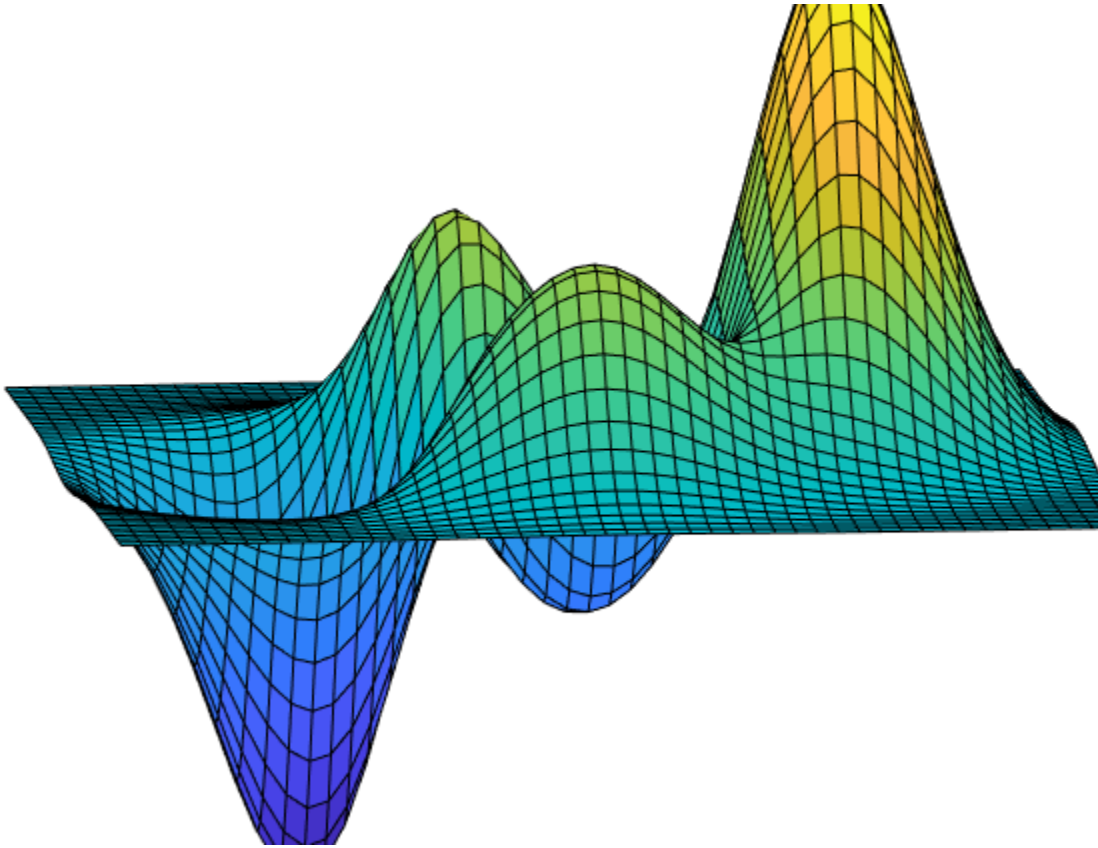
`campos` sets or queries values of the axes `CameraPosition` and `CameraPositionMode` properties. The camera position is the point in the Cartesian coordinate system of the axes from which you view the scene.

Examples

Move Camera Position

Move the camera along the *x*-axis in a series of steps.

```
surf(peaks)
axis vis3d off
for x = -200:5:200
    campos([x,5,10])
    drawnow
end
```



See Also

`axis` | `camproj` | `camtarget` | `camup` | `camva`

Topics

"Camera Graphics Terminology"

Introduced before R2006a

camproj

Set or query projection type

Syntax

```
camproj
camproj('projection_type')
camproj(axes_handle,...)
```

Description

`camproj` returns the projection type setting in the current axes. The projection type determines whether MATLAB 3-D views use a perspective or orthographic projection.

`camproj('projection_type')` sets the projection type in the current axes to the specified value. Possible values for *`projection_type`* are `orthographic` and `perspective`.

`camproj(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camproj` operates on the current axes.

`camproj` sets or queries values of the axes object `Projection` property.

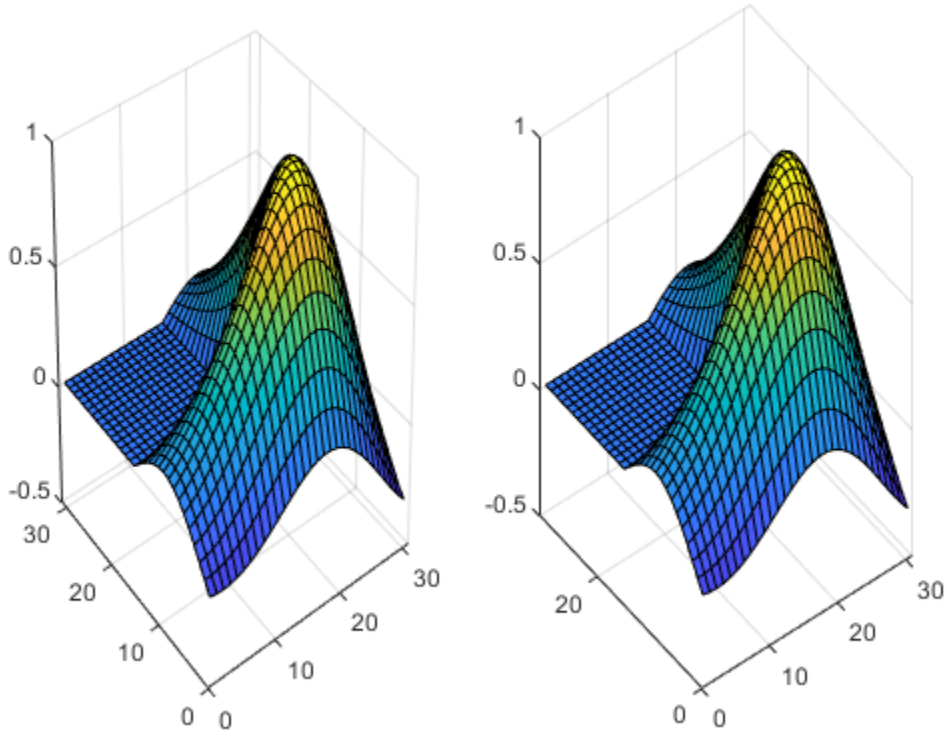
Examples

Set Projection Type

Plot two surfaces and compare the different projection settings.

```
subplot(1,2,1)
surf(membrane)
camproj('perspective')
```

```
subplot(1,2,2)
surf(membrane)
```




```
camproj('orthographic')
```

See Also

axis | campos | camtarget | camup | camva

Topics

"Camera Graphics Terminology"

Introduced before R2006a

camroll

Rotate camera about view axis

Syntax

```
camroll(dtheta)
camroll(ax,dtheta)
```

Description

`camroll(dtheta)` rotates the camera around the camera viewing axis by the amounts specified in `dtheta` (in degrees). The viewing axis is the line passing through the camera position and the camera target.

`camroll(ax,dtheta)` operates on the axes identified by the first argument, `ax`. When you do not specify an axes object, `camroll` operates on the current axes.

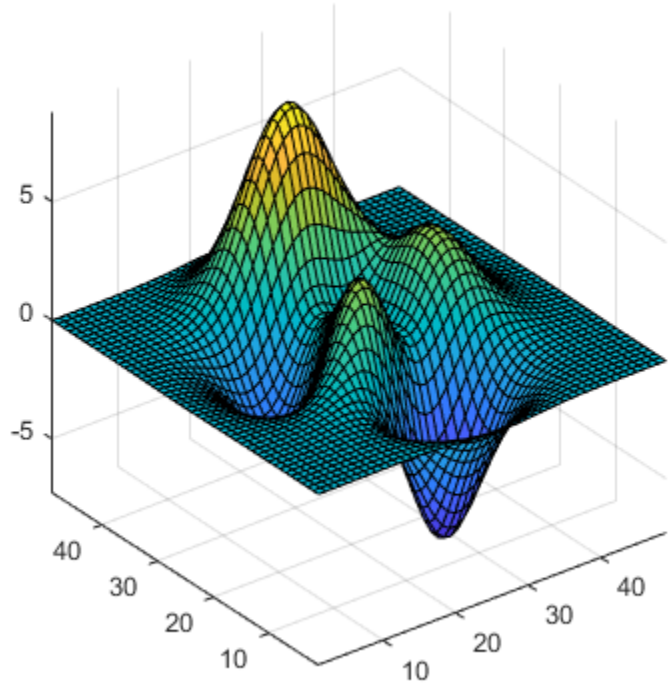
`camroll` sets the axes `CameraUpVector` property and also sets the `CameraUpVectorMode` property to `manual`.

Examples

Rotate Camera About View Axis

Rotate the camera around the viewing axis.

```
surf(peaks)
axis vis3d
for i = 1:36
    camroll(10)
    drawnow
end
```



See Also

axes | axis | camdolly | camorbit | campan | camzoom

Topics

"Camera Graphics Terminology"

Introduced before R2006a

camtarget

Set or query location of camera target

Syntax

```
camtarget
camtarget([camera_target])
camtarget('mode')
camtarget('auto')
camtarget('manual')
camtarget(axes_handle,...)
```

Description

`camtarget` returns the location of the camera target in the current axes. The camera target is the location in the axes that the camera points to. The camera remains oriented toward this point regardless of its position.

`camtarget([camera_target])` sets the camera target in the current axes to the specified value. Specify the target as a three-element vector containing the *x*-, *y*-, and *z*-coordinates of the desired location in the data units of the axes.

`camtarget('mode')` returns the value of the camera target mode, which can be either `auto` (default) or `manual`.

`camtarget('auto')` sets the camera target mode to `auto`. When the camera target mode is `auto`, the camera target is the center of the axes plot box.

`camtarget('manual')` sets the camera target mode to `manual`.

`camtarget(axes_handle,...)` performs the set or query on the axes identified by `axes_handle`. When you do not specify an axes handle, `camtarget` operates on the current axes.

`camtarget` sets or queries values of the axes object `CameraTarget` and `CameraTargetMode` properties.

Examples

Move Camera Target

Move the camera position and the camera target along the *x*-axis in a series of steps.

```
surf(peaks);
axis vis3d
xp = linspace(-150,40,50);
xt = linspace(25,50,50);
for i = 1:50
     campos([xp(i),25,5]);
     camtarget([xt(i),30,0])
     drawnow
end
```

See Also

`axis` | `campos` | `camup` | `camva`

Topics

"Camera Graphics Terminology"

Introduced before R2006a

camup

Set or query camera up vector

Syntax

```
camup
camup([up_vector])
camup('mode')
camup('auto')
camup('manual')
camup(axes_handle,...)
```

Description

`camup` returns the camera up vector setting in the current axes. The camera up vector specifies the direction that is oriented up in the scene.

`camup([up_vector])` sets the up vector in the current axes to the specified value. Specify the up vector as *x*, *y*, and *z* components.

`camup('mode')` returns the current value of the camera up vector mode, which can be either `auto` (default) or `manual`.

`camup('auto')` sets the camera up vector mode to `auto`. In `auto` mode, `[0 1 0]` is the up vector of for 2-D views. This means the *y*-axis points up. For 3-D views, the up vector is `[0 0 1]`, meaning the *z*-axis points up.

`camup('manual')` sets the camera up vector mode to `manual`. In `manual` mode, the value of the camera up vector does not change unless you set it.

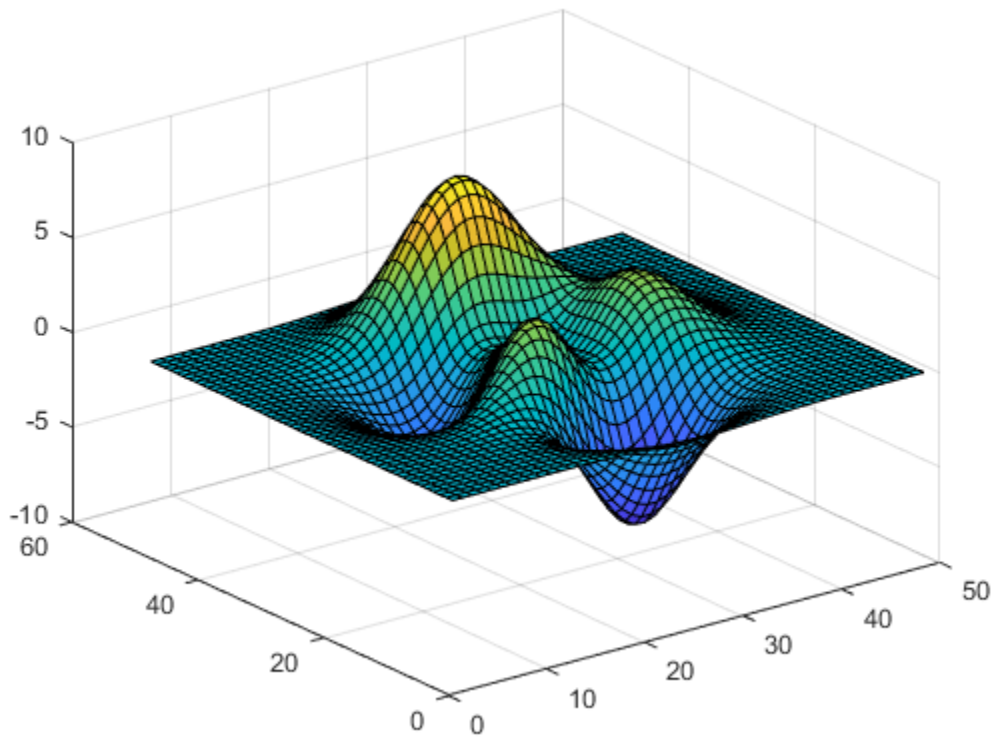
`camup(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camup` operates on the current axes.

Examples

Set Camera Up Vector

Set the *x*-axis to be the up axis.

```
surf(peaks)
camup([1 0 0]);
```



See Also

`axis` | `campos` | `camtarget`

Topics

"Camera Graphics Terminology"

Introduced before R2006a

camva

Set or query camera view angle

Syntax

```
camva
camva(view_angle)
camva('mode')
camva('auto')
camva('manual')
camva(axes_handle,...)
```

Description

`camva` returns the camera view angle setting in the current axes. The camera view angle determines the field of view of the camera. Larger angles produce a smaller view of the scene. Implement zooming by changing the camera view angle.

`camva(view_angle)` sets the view angle in the current axes to the specified value. Specify the view angle in degrees.

`camva('mode')` returns the current value of the camera view angle mode, which can be either `auto` (the default) or `manual`.

`camva('auto')` sets the camera view angle mode to `auto`.

`camva('manual')` sets the camera view angle mode to `manual`.

`camva(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camva` operates on the current axes.

Tips

The `camva` function sets or queries values of the axes object `CameraViewAngle` and `CameraViewAngleMode` properties.

When the camera view angle mode is `auto`, the camera view angle adjusts so that the scene fills the available space in the window. If you move the camera to a different position, the camera view angle changes to maintain a view of the scene that fills the available area in the window.

Setting a camera view angle or setting the camera view angle to `manual` disables the MATLAB stretch-to-fill feature (stretching of the axes to fit the window). This means setting the camera view angle to its current value,

```
camva(camva)
```

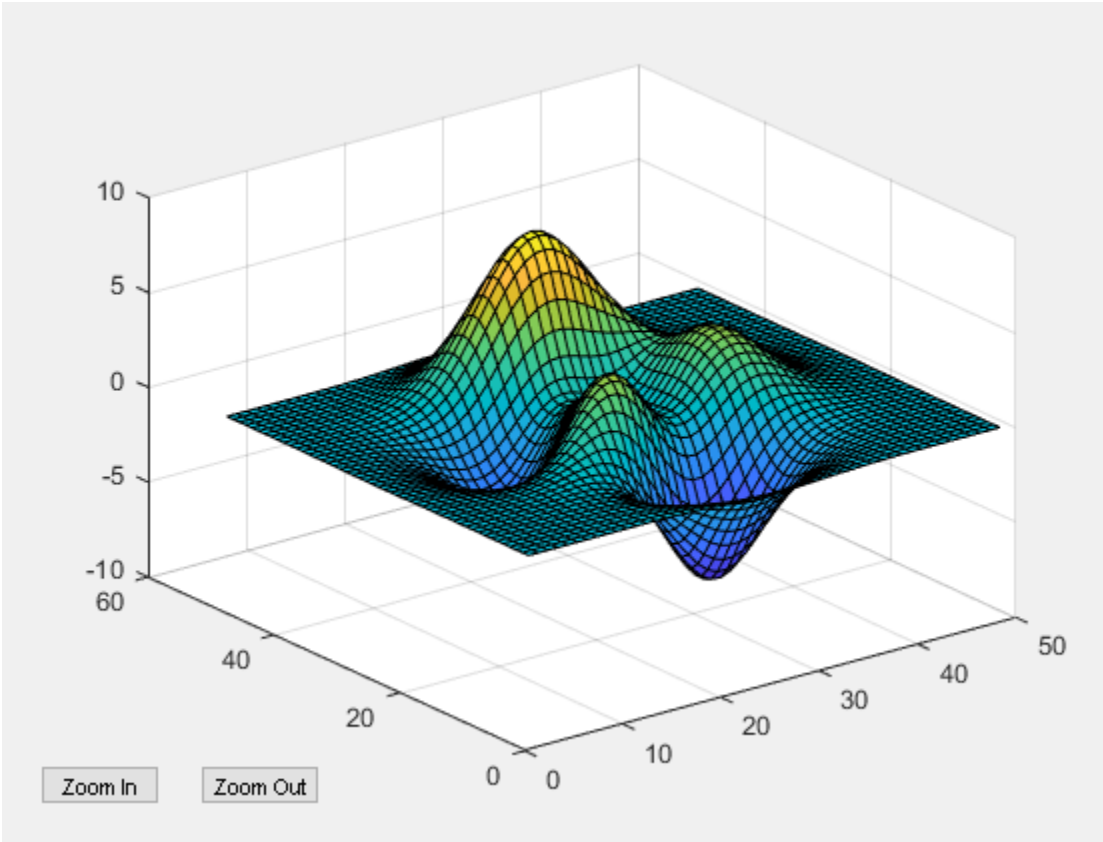
can cause a change in the way the graph looks. See `axes` for more information.

Examples

Set Camera View Angle

Create two push buttons, one that zooms in and another that zooms out. Set the range checking in the callback statements to keep the values for the camera view angle in the range greater than zero and less than 180. Then create a surface plot to zoom in and out on.

```
uicontrol('Style','pushbutton',...
  'String','Zoom In',...
  'Position',[20 20 60 20],...
  'Callback','if camva <= 1; return; else; camva(camva-1); end');

uicontrol('Style','pushbutton',...
  'String','Zoom Out',...
  'Position',[100 20 60 20],...
  'Callback',...
  'if camva >= 179; return; else; camva(camva+1); end');

surf(peaks);
```



See Also

`axis` | `campos` | `camtarget` | `camup`

Topics

"Camera Graphics Terminology"

Introduced before R2006a

camzoom

Zoom in and out on scene

Syntax

```
camzoom(zoom_factor)
camzoom(axes_handle,...)
```

Description

`camzoom(zoom_factor)` zooms in or out on the scene depending on the value specified by `zoom_factor`. If `zoom_factor` is greater than 1, the scene appears larger; if `zoom_factor` is greater than zero and less than 1, the scene appears smaller.

`camzoom(axes_handle,...)` operates on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camzoom` operates on the current axes.

Tips

`camzoom` sets the axes `CameraViewAngle` property, which in turn causes the `CameraViewAngleMode` property to be set to `manual`. Note that setting the `CameraViewAngle` property disables the MATLAB stretch-to-fill feature (stretching of the axes to fit the window). This may result in a change to the aspect ratio of your graph. See the `axes` function for more information on this behavior.

See Also

`axes` | `camdolly` | `camorbit` | `campan` | `camroll` | `camva`

Topics

"Camera Graphics Terminology"

Introduced before R2006a

cartToBary

Class: `TriRep`

(Not recommended) Convert point coordinates from Cartesian to barycentric

---

Note `cartToBary(TriRep)` is not recommended. Use `cartesianToBarycentric(triangulation)` instead.

`TriRep` is not recommended. Use `triangulation` instead.

Syntax

```
B = cartToBary(TR, SI, XC)
```

Description

`B = cartToBary(TR, SI, XC)` returns the barycentric coordinates of each point in `XC` with respect to its associated simplex `SI`.

Input Arguments

`TR`	Triangulation representation.
`SI`	Column vector of simplex indices that index into the triangulation matrix `TR.Triangulation`.
`XC`	Matrix that represents the Cartesian coordinates of the points to be converted. `XC` is of size `m`-by-`n`, where `m` is of `length(SI)`, the number of points to convert, and `n` is the dimension of the space where the triangulation resides.

Output Arguments

B	Matrix of dimension m-by-k where k is the number of vertices per simplex.

Examples

Compute the Delaunay triangulation of a set of points.

```
x = [0 4 8 12 0 4 8 12]';
y = [0 0 0 0 8 8 8 8]';
dt = DelaunayTri(x,y)
```

Compute the barycentric coordinates of the incenters.

```
cc = incenters(dt);
tri = dt(:,:);
```

Plot the original triangulation and reference points.

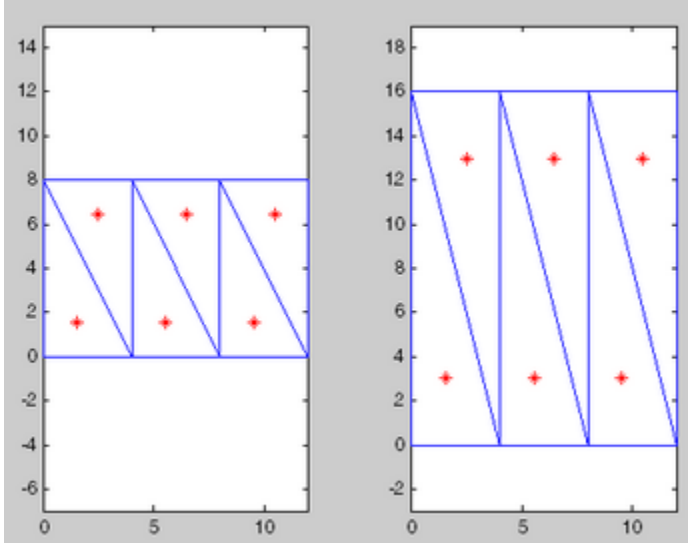
```
figure
subplot(1,2,1);
triplot(dt); hold on;
plot(cc(:,1), cc(:,2), '*r');
hold off;
axis equal;
```

Stretch the triangulation and compute the mapped locations of the incenters on the deformed triangulation.

```
b = cartToBary(dt,[1:length(tri)]',cc);
y = [0 0 0 0 16 16 16 16]';
tr = TriRep(tri,x,y)
xc = baryToCart(tr, [1:length(tri)]', b);
```

Plot the deformed triangulation and mapped locations of the reference points.

```
subplot(1,2,2);
triplot(tr);
hold on;
plot(xc(:,1), xc(:,2), '*r');
hold off;
axis equal;
```



More About

Simplex

A simplex is a triangle/tetrahedron or higher dimensional equivalent.

See Also

`barycentricToCartesian` | `delaunayTriangulation` | `pointLocation` | `triangulation`

cart2pol

Transform Cartesian coordinates to polar or cylindrical

Syntax

```
[theta,rho] = cart2pol(x,y)
[theta,rho,z] = cart2pol(x,y,z)
```

Description

`[theta,rho] = cart2pol(x,y)` transforms corresponding elements of the two-dimensional Cartesian coordinate arrays `x` and `y` into polar coordinates `theta` and `rho`.

`[theta,rho,z] = cart2pol(x,y,z)` transforms three-dimensional Cartesian coordinate arrays `x`, `y`, and `z` into cylindrical coordinates `theta`, `rho`, and `z`.

Examples

Cartesian to Polar Coordinates

Convert the Cartesian coordinates defined by corresponding entries in matrices `x` and `y` to polar coordinates `theta` and `rho`.

```
x = [5 3.5355 0 -10]
```

```
x = 1×4

    5.0000    3.5355         0  -10.0000
```

```
y = [0 3.5355 10 0]
```

```
y = 1×4
```

```
         0    3.5355   10.0000         0
```

```
[theta,rho] = cart2pol(x,y)
```

```
theta = 1×4

         0    0.7854    1.5708    3.1416


rho = 1×4

    5.0000    5.0000   10.0000   10.0000
```

Cartesian to Cylindrical Coordinates

Convert the three-dimensional Cartesian coordinates defined by corresponding entries in the matrices `x`, `y`, and `z` to cylindrical coordinates `theta`, `rho`, and `z`.

```
x = [1 2.1213 0 -5]'
```

```
x = 4×1

    1.0000
    2.1213
         0
   -5.0000


y = [0 2.1213 4 0]'
```

```
y = 4×1

         0
    2.1213
    4.0000
         0


z = [7 8 9 10]'
```

```
z = 4×1
```

```
     7
     8
     9
    10
```

```
[theta,rho,z] = cart2pol(x,y,z)
```

```
theta = 4×1

         0
    0.7854
    1.5708
    3.1416


rho = 4×1

    1.0000
    3.0000
    4.0000
    5.0000


z = 4×1

     7
     8
     9
    10
```

Input Arguments

x, y, z — Cartesian coordinates
scalars | vectors | matrices | multidimensional arrays

Cartesian coordinates, specified as scalars, vectors, matrices, or multidimensional arrays. `x`, `y`, and `z` must be the same size, or any of them can be scalar.

Data Types: `single` | `double`

Output Arguments

`theta` — Angular coordinate
array

Angular coordinate, returned as an array. `theta` is the counterclockwise angle in the *x*-*y* plane measured in radians from the positive *x*-axis. The value of the angle is in the range `[-pi pi]`.

`rho` — Radial coordinate
array

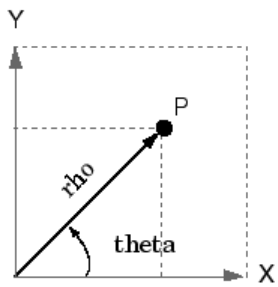
Radial coordinate, returned as an array. `rho` is the distance from the origin to a point in the *x*-*y* plane.

`z` — Elevation coordinate
array

Elevation coordinate, returned as an array. `z` is the height above the *x*-*y* plane.

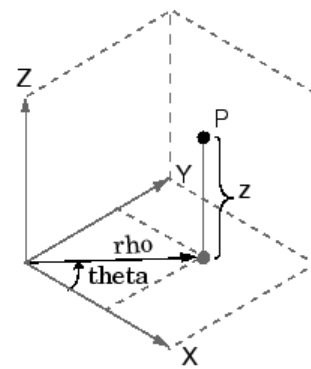
Algorithms

The mapping from two-dimensional Cartesian coordinates to polar coordinates, and from three-dimensional Cartesian coordinates to cylindrical coordinates is



Two-Dimensional Mapping

```
theta = atan2(y,x)
rho = sqrt(x.^2 + y.^2)
```



Three-Dimensional Mapping

```
theta = atan2(y,x)
rho = sqrt(x.^2 + y.^2)
z = z
```

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cart2sph` | `pol2cart` | `sph2cart`

Introduced before R2006a

cart2sph

Transform Cartesian coordinates to spherical

Syntax

```
[azimuth,elevation,r] = cart2sph(x,y,z)
```

Description

`[azimuth,elevation,r] = cart2sph(x,y,z)` transforms corresponding elements of the Cartesian coordinate arrays `x`, `y`, and `z` to spherical coordinates `azimuth`, `elevation`, and `r`.

Examples

Cartesian to Spherical Coordinates

Convert the Cartesian coordinates defined by corresponding entries in the matrices `x`, `y`, and `z` to spherical coordinates `az`, `el`, and `r`. These points correspond to the eight vertices of a cube.

```
x = [1 1 1 1; -1 -1 -1 -1]
```

```
x = 2×4

     1     1     1     1
    -1    -1    -1    -1
```

```
y = [1 1 -1 -1; 1 1 -1 -1]
```

```
y = 2×4

     1     1    -1    -1
```

```
     1     1    -1    -1
```

```
z = [1 -1 1 -1; 1 -1 1 -1]
```

```
z = 2×4

     1    -1     1    -1
     1    -1     1    -1
```

```
[az,el,r] = cart2sph(x,y,z)
```

```
az = 2×4

    0.7854    0.7854   -0.7854   -0.7854
    2.3562    2.3562   -2.3562   -2.3562


el = 2×4

    0.6155   -0.6155    0.6155   -0.6155
    0.6155   -0.6155    0.6155   -0.6155


r = 2×4

    1.7321    1.7321    1.7321    1.7321
    1.7321    1.7321    1.7321    1.7321
```

Input Arguments

x,y,z — Cartesian coordinates
scalars | vectors | matrices | multidimensional arrays

Cartesian coordinates, specified as scalars, vectors, matrices, or multidimensional arrays. x, y, and z must be the same size, or any of them can be scalar.

Data Types: single | double

Output Arguments

`azimuth` — Azimuth angle
array

Azimuth angle, returned as an array. `azimuth` is the counterclockwise angle in the *x*-*y* plane measured in radians from the positive *x*-axis. The value of the angle is in the range `[-pi pi]`.

`elevation` — Elevation angle
array

Elevation angle, returned as an array. `elevation` is the elevation angle in radians from the *x*-*y* plane. The value of the angle is in the range `[-pi/2, pi/2]`.

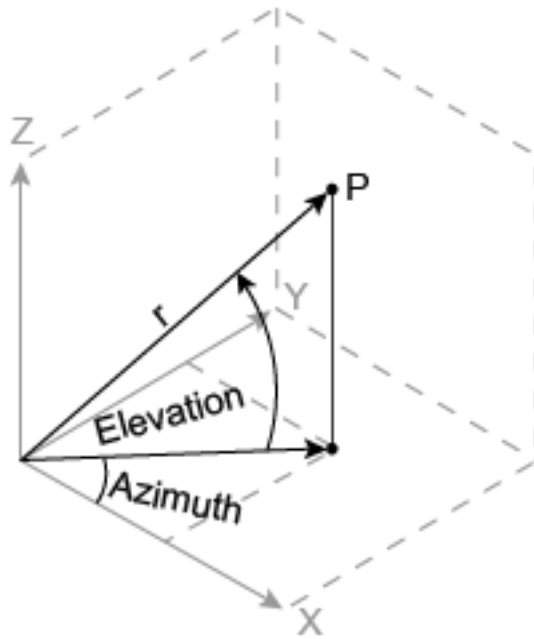
`r` — Radius
array

Radius, returned as an array. `r` is the distance from the origin to a point. The length units of `r` are arbitrary, matching the units of the input arrays `x`, `y`, and `z`.

Algorithms

The mapping from three-dimensional Cartesian coordinates to spherical coordinates is

```
azimuth = atan2(y,x)
elevation = atan2(z,sqrt(x.^2 + y.^2))
r = sqrt(x.^2 + y.^2 + z.^2)
```



The notation for spherical coordinates is not standard. For the `cart2sph` function, `elevation` is measured from the *x*-*y* plane. Notice that if `elevation = 0`, the point is in the *x*-*y* plane. If `elevation = pi/2`, then the point is on the positive *z*-axis.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cart2pol` | `pol2cart` | `sph2cart`

Introduced before R2006a

cast

Convert variable to different data type

Syntax

```
B = cast(A,newclass)
B = cast(A,'like',p)
```

Description

`B = cast(A,newclass)` converts `A` to the data type (class) `newclass`, where `newclass` is the name of a built-in data type compatible with `A`. The `cast` function truncates any values in `A` that are outside the range of `newclass` to the nearest endpoint.

When converting a floating-point number to an integer, the `cast` function rounds the number to the nearest integer. If the floating-point number has a fractional part of exactly 0.5, then it rounds away from zero to the integer with larger magnitude.

`B = cast(A,'like',p)` converts `A` to the same data type, sparsity, and complexity (real or complex) as the variable `p`. If `A` and `p` are both real, then `B` is also real. Otherwise, `B` is complex.

Examples

Convert Numeric Data Type

Convert `int8` values to `uint8`.

Define a vector of 8-bit integers.

```
a = int8([-5 5]);
```

Convert `a` to unsigned 8-bit integers. The –5 value outside the range of `uint8` is truncated to 0.

```
b = cast(a,'uint8')
```

```
b = 1x2 uint8 row vector

   0   5
```

Convert Vector to Existing Data Type

Create a 1-by-3 vector of 32-bit signed integers.

```
A = int32([-12 34 56])
```

```
A = 1x3 int32 row vector

   -12    34    56
```

Create a complex number of the data type `double`.

```
p = 1+2i
```

```
p = 1.0000 + 2.0000i
```

Convert `A` to the same data type and complexity as the variable `p`.

```
B = cast(A,'like',p)
```

```
B = 1×3 complex

 -12.0000 + 0.0000i  34.0000 + 0.0000i  56.0000 + 0.0000i
```

Convert Matrix to Existing Data Type

Create a 2-by-3 matrix of zeros whose elements are 32-bit unsigned integers.

```
A = zeros(2,3,'uint32')
```

```
A = 2x3 uint32 matrix
```

```
   0   0   0
   0   0   0
```

Create a 2-by-2 sparse matrix of the data type `double`.

```
p = sparse(2,2,pi)
```

```
p =
   (2,2)        3.1416
```

Convert `A` to the same data type and sparsity as the variable `p`.

```
B = cast(A,'like',p)
```

```
B =
   All zero sparse: 2x3
```

Input Arguments

`A` — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `fi`
Complex Number Support: Yes

`newclass` — New class
`'single'` | `'double'` | `'int8'` | `'int16'` | `'int32'` | `'int64'` | `'uint8'` | `'uint16'` | `'uint32'` | `'uint64'` | `'logical'` | `'char'`

New class, specified as `'single'`, `'double'`, `'int8'`, `'int16'`, `'int32'`, `'int64'`, `'uint8'`, `'uint16'`, `'uint32'`, `'uint64'`, `'logical'`, or `'char'`.

`p` — Prototype
scalar | vector | matrix | multidimensional array

Prototype, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `fi`
Complex Number Support: Yes

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Enumeration inputs must be scalar valued at compile time. Arrays of enumerations are not supported.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Enumeration inputs must be scalar valued at compile time. Arrays of enumerations are not supported.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- For the syntax `B = cast(A,'like',p)`, the output `B` is a `gpuArray` if and only if `p` is a `gpuArray`.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- For the syntax `B = cast(A,'like',p)`, the output `B` is a distributed array if and only if `p` is a distributed array.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`class` | `typecast`

Topics

“Class Support for Array-Creation Functions”

Introduced before R2006a

cat

Concatenate arrays

Syntax

```
C = cat(dim,A,B)
C = cat(dim,A1,A2,…,An)
```

Description

`C = cat(dim,A,B)` concatenates `B` to the end of `A` along dimension `dim` when `A` and `B` have compatible sizes (the lengths of the dimensions match except for the operating dimension `dim`).

`C = cat(dim,A1,A2,…,An)` concatenates `A1`, `A2`, … , `An` along dimension `dim`.

You can use the square bracket operator `[]` to concatenate. For example, `[A,B]` or `[A B]` concatenates arrays `A` and `B` horizontally, and `[A; B]` concatenates them vertically.

Examples

Two Matrices

Create and concatenate two matrices vertically, then horizontally.

```
A = ones(3)
```

```
A = 3×3

     1     1     1
     1     1     1
     1     1     1
```

```
B = zeros(3)
```

```
B = 3×3

     0     0     0
     0     0     0
     0     0     0
```

```
C1 = cat(1,A,B)
```

```
C1 = 6×3

     1     1     1
     1     1     1
     1     1     1
     0     0     0
     0     0     0
     0     0     0
```

```
C2 = cat(2,A,B)
```

```
C2 = 3×6

     1     1     1     0     0     0
     1     1     1     0     0     0
     1     1     1     0     0     0
```

3-D Arrays

Create two 3-D arrays and concatenate them along the third dimension. The lengths of the first and second dimensions in the resulting array matches the corresponding lengths in the input arrays, while the third dimension expands.

```
A = rand(2,3,4);
B = rand(2,3,5);
C = cat(3,A,B);
szC = size(C)
```

```
szC = 1×3

     2     3     9
```

Expand Tables

Create a table and add a row using a cell array.

```
LastName = {'Sanchez';'Johnson';'Li';'Diaz'};
Age = [38;43;38;40];
T1 = table(LastName,Age)
```

```
T1=4×2 table
     LastName      Age
    ___________    ___

    {'Sanchez'}    38
    {'Johnson'}    43
    {'Li'     }    38
    {'Diaz'   }    40
```

```
Trow = {'Brown',49};
T2 = cat(1,T1,Trow)
```

```
T2=5×2 table
     LastName      Age
    ___________    ___

    {'Sanchez'}    38
    {'Johnson'}    43
    {'Li'     }    38
    {'Diaz'   }    40
    {'Brown'  }    49
```

Dates with Different Types

Concatenate a date character vector, a string date, and a datetime into a single column of dates. The result is a datetime vector.

```
chardate = '2016-03-24';
strdate = "2016-04-19";
```

```
t = datetime('2016-05-10','InputFormat','yyyy-MM-dd');
C = cat(1,chardate,strdate,t)
```

```
C = 3x1 datetime array
   24-Mar-2016
   19-Apr-2016
   10-May-2016
```

Matrices in a Cell Array

Create a cell array containing two matrices, and concatenate the matrices both vertically and horizontally.

```
M1 = [1 2; 3 4];
M2 = [5 6; 7 8];
A1 = {M1,M2};
Cvert = cat(1,A1{:})
```

```
Cvert = 4×2

     1     2
     3     4
     5     6
     7     8
```

```
Chorz = cat(2,A1{:})
```

```
Chorz = 2×4

     1     2     5     6
     3     4     7     8
```

Input Arguments

`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. For example, if `A` and `B` are both 2-by-2 matrices, then `cat(1,A,B)` concatenates vertically creating a 4-by-2 matrix. `cat(2,A,B)` concatenates horizontally creating a 2-by-4 matrix.

`dim` must be either 1 or 2 for table or timetable input.

`A` — First input

scalar | vector | matrix | multidimensional array | table | timetable

First input, specified as a scalar, vector, matrix, multidimensional array, table, or timetable.

`B` — Second input

scalar | vector | matrix | multidimensional array | table | timetable

Second input, specified as a scalar, vector, matrix, multidimensional array, table, or timetable.

- The elements of `B` are concatenated to the end of the first input along the operating dimension. The sizes of the input arguments must be compatible. For example, if the first input is a matrix of size 3-by-2, then `B` must have 2 columns to concatenate vertically, and 3 rows to concatenate horizontally.
- When concatenating horizontally, all table inputs must have unique variable names. When present, row names must be identical, except for order. Similarly, all timetable inputs must have the same row times and all columns must have different names.
- You can concatenate valid combinations of different types. For more information, see “Valid Combinations of Unlike Classes”.

`A1,A2,…,An` — List of inputs

comma-separated list

List of inputs, specified as a comma-separated list of arrays to concatenate in the order they are specified.

- The inputs must have compatible sizes. For example, if `A1` is a row vector of length *m*, then the remaining inputs must each have *m* columns to concatenate vertically.
- When concatenating horizontally, all table inputs must have unique variable names. When present, row names must be identical, except for order. Similarly, all timetable inputs must have the same row times and all columns must have different names.
- You can concatenate valid combinations of different types. For more information, see “Valid Combinations of Unlike Classes”.

Tips

- To construct text by horizontally concatenating strings, character vectors, or cell arrays of character vectors, use the `strcat` function.
- To construct a single piece of delimited text from a cell array of character vectors or a string array, use the `strjoin` function.

Algorithms

When concatenating an empty array to a nonempty array, `cat` omits the empty array in the output. For example, `cat(2,[1 2],[])` returns the row vector `[1 2]`.

If all input arguments are empty and have compatible sizes, then `cat` returns an empty array whose size is equal to the output size as when the inputs are nonempty. For example, `cat(2,zeros(0,1),zeros(0,2))` returns a 0-by-3 empty array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitation:

- Vertical concatenation of character arrays is not supported.
- Concatenation in any dimension other than 1 requires all input arguments to be tall arrays.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Does not support concatenation of cell arrays.
- If supplied, `dim` must be a constant.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`horzcat` | `strcat` | `strjoin` | `vertcat`

Topics

“Creating, Concatenating, and Expanding Matrices”
“Valid Combinations of Unlike Classes”
“Combine Categorical Arrays”
“Concatenating Objects of Different Classes”
“Concatenation Methods”

Introduced before R2006a

categorical

Array that contains values assigned to categories

Description

`categorical` is a data type that assigns values to a finite set of discrete categories, such as `High`, `Med`, and `Low`. These categories can have a mathematical ordering that you specify, such as `High > Med > Low`, but it is not required. A categorical array provides efficient storage and convenient manipulation of nonnumeric data, while also maintaining meaningful names for the values. A common use of categorical arrays is to specify groups of rows in a table.

Creation

Syntax

```
B = categorical(A)
B = categorical(A,valueset)
B = categorical(A,valueset,catnames)
B = categorical(A, ___ ,Name,Value)
```

Description

`B = categorical(A)` creates a categorical array from the array `A`. The categories of `B` are the sorted unique values from `A`.

`B = categorical(A,valueset)` creates one category for each value in `valueset`. The categories of `B` are in the same order as the values of `valueset`.

You can use `valueset` to include categories for values not present in `A`. Conversely, if `A` contains any values not present in `valueset`, then the corresponding elements of `B` are undefined.

`B = categorical(A,valueset,catnames)` names the categories in `B` by matching the category values in `valueset` with the names in `catnames`.

`B = categorical(A, ___ ,Name,Value)` creates a categorical array with additional options specified by one or more `Name,Value` pair arguments. You can include any of the input arguments in previous syntaxes.

For example, to indicate that the categories have a mathematical ordering, specify `'Ordinal',true`.

Input Arguments

A — Input array

numeric array | logical array | categorical array | datetime array | duration array | string array | cell array of character vectors

Input array, specified as a numeric array, logical array, categorical array, datetime array, duration array, string array, or cell array of character vectors.

`categorical` removes leading and trailing spaces from input values that are strings or character vectors.

If `A` contains missing values, then the corresponding element of `B` is undefined and displays as `<undefined>`. The `categorical` function converts the following values to undefined categorical values:

- `NaN` in numeric and duration arrays
- The missing string (`<missing>`) or the empty string (`""`) in string arrays
- The empty character vector (`''`) in cell arrays of character vectors
- `NaT` in datetime arrays
- Undefined values (`<undefined>`) in categorical arrays

`B` does not have a category for undefined values. To create an explicit category for missing or undefined values, you must include the desired category name in `catnames`, and a missing value as the corresponding value in `valueset`.

`A` also can be an array of objects with the following class methods:

- `unique`

- `eq`

`valueset` — Categories
`unique(A)` (default) | vector of unique values

Categories, specified as a vector of unique values. The data type of `valueset` and the data type of `A` must be the same, except when `A` is a string array. In that case, `valueset` either can be a string array or a cell array of character vectors.

`categorical` removes leading and trailing spaces from elements of `valueset` that are strings or character vectors.

`catnames` — Category names
cell array of character vectors | string array

Category names, specified as a cell array of character vectors or a string array. If you do not specify the `catnames` input argument, then `categorical` uses the values in `valueset` as category names.

To merge multiple distinct values in `A` into a single category in `B`, include duplicate names corresponding to those values.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Ordinal',true` specifies that the categories have a mathematical ordering.

`Ordinal` — Ordinal variable indicator
`false` (default) | `true`

Ordinal variable indicator, specified as the comma-separated pair consisting of `'Ordinal'` and either `false` (`0`) or `true` (`1`).

`false` (`0`)	`categorical` creates a categorical array that is not ordinal, which is the default behavior. The categories of `B` have no mathematical ordering. Therefore, you can compare the values in `B` for equality only. You cannot compare the values using any other relational operator.
`true` (`1`)	`categorical` creates an ordinal categorical array. The categories of `B` have a mathematical ordering, such that the first category specified is the smallest and the last category is the largest. You can compare the values in `B` using relational operators, such as less than and greater than, in addition to comparing the values for equality. You also can use the `min` and `max` functions on an ordinal categorical array.

For more information, see “Ordinal Categorical Arrays”.

`Protected` — Protected categories indicator
`false` | `true`

Protected categories indicator, specified as the comma-separated pair consisting of `'Protected'` and either `false` (`0`) or `true` (`1`). The categories of ordinal categorical arrays are always protected. The default value is `true` when you specify `'Ordinal',true`. Otherwise, the value is `false`.

`false` (`0`)	When you assign new values to `B`, the categories update automatically. Therefore, you can combine (nonordinal) categorical arrays that have different categories. The categories can update accordingly to include the categories from both arrays.
`true` (`1`)	When you assign new values to `B`, the values must belong to one of the existing categories. Therefore, you can only combine arrays that have the same categories. To add new categories to `B`, you must use the function `addcats`.

Examples

Convert Array and Select Data by Category

Create a categorical array that has weather station labels. Add it to a table of temperature readings. Then use the categories to select temperature readings by station.

First, create arrays containing temperature readings, dates, and station labels.

```
Temps = [58; 72; 56; 90; 76];
Dates = {'2017-04-17';'2017-04-18';'2017-04-30';'2017-05-01';'2017-04-27'};
Stations = {'S1';'S2';'S1';'S3';'S2'};
```

Convert `Stations` to a categorical array.

```
Stations = categorical(Stations)
```

```
Stations = 5x1 categorical array
     S1
     S2
     S1
     S3
     S2
```

Display the categories. The three stations labels are categories.

```
categories(Stations)
```

```
ans = 3x1 cell array
    {'S1'}
    {'S2'}
    {'S3'}
```

Create a table that contains the temperatures, dates, and station labels.

```
T = table(Temps,Dates,Stations)
```

```
T=5×3 table
    Temps        Dates         Stations
    _____    _______________    ________

     58      {'2017-04-17'}       S1
     72      {'2017-04-18'}       S2
     56      {'2017-04-30'}       S1
     90      {'2017-05-01'}       S3
```

```
     76      {'2017-04-27'}       S2
```

Display the readings taken from station S2. You can use the == operator to find the values of Station that equal S2. Then use logical indexing to select the table rows that have data from station S2.

```
TF = (T.Stations == 'S2');
T(TF,:)
```

```
ans=2×3 table
    Temps        Dates         Stations
    _____    _____________    ________

     72      {'2017-04-18'}       S2
     76      {'2017-04-27'}       S2
```

Specify Categories Not Present in Input Array

Convert the cell array of character vectors A to a categorical array. Specify a list of categories that includes values that are not present in A.

Create a cell array of character vectors.

```
A = {'republican' 'democrat'; 'democrat' 'democrat'; 'democrat' 'republican'};
```

Convert A to a categorical array. Add a category for independent.

```
valueset = {'democrat' 'republican' 'independent'};
B = categorical(A,valueset)
```

```
B = 3x2 categorical array
     republican      democrat
     democrat        democrat
     democrat        republican
```

Display the categories of B.

```
categories(B)
```

```
ans = 3x1 cell array
    {'democrat'   }
    {'republican' }
    {'independent'}
```

Specify Category Names for Integers

Create a numeric array.

```
A = [1 3 2; 2 1 3; 3 1 2]
```

```
A = 3×3

     1     3     2
     2     1     3
     3     1     2
```

Convert A to categorical array B and specify category names.

```
B = categorical(A,[1 2 3],{'red' 'green' 'blue'})
```

```
B = 3x3 categorical array
     red        blue      green
     green      red       blue
     blue       red       green
```

Display the categories of B.

```
categories(B)
```

```
ans = 3x1 cell array
    {'red'  }
    {'green'}
    {'blue' }
```

B is not an ordinal categorical array. Therefore, you can compare the values in B only using the equality operators, == and ~=.

Find the elements that belong to the category `'red'`. Access those elements using logical indexing.

```
TF = (B == 'red');
B(TF)
```

```
ans = 3x1 categorical array
     red
     red
     red
```

Create Ordinal Categorical Array

Create a 5-by-2 numeric array.

```
A = [3 2;3 3;3 2;2 1;3 2]
```

```
A = 5×2

     3     2
     3     3
     3     2
     2     1
     3     2
```

Convert `A` to an ordinal categorical array where `1`, `2`, and `3` represent categories `child`, `adult`, and `senior` respectively.

```
valueset = [1:3];
catnames = {'child' 'adult' 'senior'};

B = categorical(A,valueset,catnames,'Ordinal',true)
```

```
B = 5x2 categorical array
     senior      adult
     senior      senior
     senior      adult
     adult       child
     senior      adult
```

Since B is ordinal, the categories of B have a mathematical ordering, `child < adult < senior`.

Convert String Array

Starting in R2017a, you can create string arrays using double quotes. Also, a string array can have missing values, displayed as `<missing>`, without quotation marks.

```
str = ["plane","jet","plane","helicopter",missing,"jet"]
```

```
str = 1x6 string array
    "plane"    "jet"    "plane"    "helicopter"    <missing>    "jet"
```

Convert string array `str` to a categorical array. The `categorical` function converts missing strings to undefined categorical values, displayed as `<undefined>`.

```
C = categorical(str)
```

```
C = 1x6 categorical array
     plane      jet      plane      helicopter      <undefined>      jet
```

Bin Numeric Data into Categories

Use the `discretize` function (instead of `categorical`) to bin 100 random numbers into three categories.

```
x = rand(100,1);
y = discretize(x,[0 .25 .75 1],'categorical',{'small','medium','large'});
summary(y)
```

```
     small       22
     medium      46
     large       32
```

Tips

- For a list of functions that accept or return categorical arrays, see “Categorical Arrays”.
- If the input array has numeric, datetime, or duration values that are too close together, then the `categorical` function truncates them to duplicate values. For example, `categorical([1 1.00001])` truncates the second element of the input array. To create categories from numeric data, use the `discretize` function.

Alternatives

You also can group numeric data into categories using `discretize`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitation:

With the syntax `B = categorical(A)`, the order of categories is undefined. Use `valueset` and `catnames` to enforce the order.

For more information, see “Tall Arrays for Out-of-Memory Data”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Starting in R2019a, you can use categorical arrays in MATLAB code intended for code generation. For more information, see “Code Generation for Categorical Arrays” (MATLAB Coder) and “Categorical Array Limitations for Code Generation” (MATLAB Coder).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- For the one input syntax `B = categorical(A)`, the order of the categories is undefined. To enforce the order, use `valueset` and `catnames`.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`categories` | `discretize` | `iscategorical`

Topics

“Access Data Using Categorical Arrays”
“Advantages of Using Categorical Arrays”
“Core Functions Supporting Categorical Arrays”

Introduced in R2013b

CategoricalRuler Properties

Control axis with categorical values

Description

`CategoricalRuler` properties control the appearance and behavior of an *x*-axis, *y*-axis, or *z*-axis that shows categorical values. Each individual axis has its own ruler object. By changing property values of the ruler, you can modify certain aspects of a specific axis.

Use dot notation to refer to a particular ruler and property. Access the ruler objects through the `XAxis`, `YAxis`, and `ZAxis` properties of the `Axes` object. For example:

```
ax = gca;
co = ax.XAxis.Color;
ax.XAxis.Color = 'blue';
```

Properties

Appearance

`Color` — Color of axis line and labels
RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of the axis line and labels, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Note The `Color` property for the ruler and the associated `XColor`, `YColor`, or `ZColor` property for the parent axes always have the same value. Setting one also sets the other.

`LineWidth` — Width of axis line and tick marks

`0.5` (default) | positive value

Width of axis line and tick marks, specified as a positive value in point units. One point equals 1/72 inch.

Example: `ax.XAxis.LineWidth = 2;`

Note Setting the `LineWidth` property for the parent axes sets the `LineWidth` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

`Label` — Axis label

text object (default)

Axis label, which is a text object. To display text or change existing text, set the `String` property for the text object. Use other properties to change the text appearance, such as the font style or color.

```
ax = gca;
ax.XAxis.Label.String = 'X Axis';
ax.XAxis.Label.FontSize = 12;
```

For a full list of options, see Text.

Alternatively, add or modify the axis labels using the `xlabel`, `ylabel`, and `zlabel` functions.

Note The text object is not a child of the ruler object, so it cannot be returned by `findobj` and it does not use the default text property values.

`Visible` — Axis visibility

`'on'` (default) | `'off'`

Axis visibility, specified as one of these values:

- `'on'` — Display the axis.
- `'off'` — Hide the axis without deleting it. You still can access properties of an invisible axis using the ruler object.

Example: `ax.XAxis.Visible = 'off';`

Scale and Direction

`Scale` — Scale of values along axis

`'linear'` (default)

Scale of values along axis, specified as `'linear'`.

`Direction` — Direction of increasing values

`'normal'` (default) | `'reverse'`

Direction of increasing values, specified as one of these values:

- `'normal'` — Values increase from left to right or bottom to top.
- `'reverse'` — Values increase from right to left or top to bottom.

Note The `Direction` property for the ruler and the associated `XDir`, `YDir`, or `ZDir` property for the parent axes always have the same value. Setting one also sets the other.

Example: `ax.XAxis.Direction = 'reverse';`

Tick Values and Labels

`Categories` — Categories plotted along axis

cell array of character vectors | categorical array | string array

Categories plotted along the axis, specified as a cell array of character vectors, a categorical array, or a string array. Querying this property returns a cell array of character vectors. Set this property to exclude certain categories from appearing in the chart.

Example: `ax.XAxis.Categories = categorical({'small','medium','large'})` specifies a categorical array.

Example: `ax.XAxis.Categories = {'small','medium','large'}` specifies a cell array.

Example: `ax.XAxis.Categories = string({'small','medium','large'})` specifies a string array.

Example

Create a histogram with categories along the x-axis and query the `Categories` property. Then set the `Categories` property so that only a subset of categories appear along the axis.

```
A =  categorical({'r' 'b' 'g' 'g' 'r' 'b' 'b' 'r' 'g' 'y' 'g' 'm'});
h = histogram(A);
ax = gca;
ax.XAxis.Categories
```

```
ans =

  1×5 categorical array

     b      g      m      r      y
```

```
ax.XAxis.Categories = categorical({'b','m','y'});
```

Data Types: `cell` | `categorical` | `string`

`Limits` — First and last categories displayed along axis
two-element vector

First and last categories displayed along the axis, specified as a two-element vector indicating two category names. The categories must be two of the categories in the `Categories` property, and must appear in the same relative order that they appear in the `Categories` property.

Example: `ax.XAxis.Categories = categorical({'small','medium'})` specifies a two-element categorical array.

Example: `ax.XAxis.Categories = {'small','medium'}` specifies a two-element cell array.

Example: `ax.XAxis.Categories = string({'small','medium'})` specifies a two-element string array.

If you assign a value to this property, then MATLAB changes the associated `LimitsMode` property to `'manual'`.

Alternatively, set the limits using the `xlim`, `ylim`, and `zlim` functions.

Note The `Limits` property for the ruler and the associated `XLim`, `YLim`, or `ZLim` property for the parent axes always have the same value. Setting one also sets the other.

Example

Set the `Limits` property to only show the categories between `'b'` and `'m'` along the axis.

```
A =  categorical({'r' 'b' 'g' 'g' 'r' 'b' 'b' 'r' 'g' 'y' 'g' 'm'});
h = histogram(A);
ax = gca;
ax.XAxis.Limits = categorical({'b','m'});
```

Data Types: `categorical` | `string` | `cell`

`LimitsMode` — Selection mode for `Limits` property

`'auto'` (default) | `'manual'`

Selection mode for the `Limits` property, specified as one of these values:

- `'auto'` — Automatically select the axis limits based on the data plotted.
- `'manual'` — Use axis limit values that you specify. To specify the axis limits, set the `Limits` property.

Note The `LimitsMode` property for the ruler and the associated `XLimMode`, `YLimMode`, or `ZLimMode` property for the parent axes always have the same value. Setting one also sets the other.

`TickValues` — Tick mark locations along the axis

cell array of character vectors | categorical array | string array

Tick mark locations along the axis, specified as a cell array of character vectors, a categorical array, or a string array indicating category names. The categories must be a subset of the categories in the `Categories` property and must appear in the same relative order that they appear in the `Categories` property. To remove all tick marks and labels along the axis, set this property to an empty array `[]`.

Example: `ax.XAxis.Categories = categorical({'small','medium','large'})` specifies a categorical array.

Example: `ax.XAxis.Categories = {'small','medium','large'}` specifies a cell array.

Example: `ax.XAxis.Categories = string({'small','medium','large'})` specifies a string array.

If you assign a value to this property, then MATLAB sets the `TickValuesMode` property to `'manual'`.

Alternatively, use the `xticks`, `yticks`, and `zticks` functions.

Note The `TickValues` property for the ruler and the associated `XTick`, `YTick`, or `ZTick` property for the parent axes always have the same value. Setting one also sets the other.

Example

Label a subset of categories along the axis by setting the `TickValues` property to a subset of categories.

```
A =  categorical({'r' 'b' 'g' 'g' 'r' 'b' 'b' 'r' 'g' 'y' 'g' 'm'});
h = histogram(A);
ax = gca;
ax.XAxis.TickValues = categorical({'b','m','y'});
```

Data Types: `categorical` | `string` | `cell`

`TickValuesMode` — Selection mode for `TickValues` property

`'auto'` (default) | `'manual'`

Selection mode for the `TickValues` property, specified as one of these values:

- `'auto'` — Automatically select the tick values based on the data plotted.
- `'manual'` — Use tick values that you specify. To specify the values, set the `TickValues` property.

Note The `TickValuesMode` property for the ruler and the associated `XTickMode`, `YTickMode`, or `ZTickMode` property for the parent axes always have the same value. Setting one also sets the other.

TickLabels — Tick mark labels

`''` (default) | cell array of character vectors | string array | categorical array

Tick mark labels, specified as a cell array of character vectors, string array, or categorical array. Use this property to relabel the tick marks without changing the category names. If you do not specify a label for every value, then the labels repeat. The labels support TeX and LaTeX markup. See the `TickLabelInterpreter` property for more information.

If you set this property, then MATLAB sets the `TickLabelsMode` property to `'manual'`.

Alternatively, specify the tick labels using the `xticklabels`, `yticklabels`, and `zticklabels` functions.

Note

- If you specify this property as a categorical array, MATLAB uses the values in the array, not the categories.
- The `TickLabels` property for the ruler and the associated `XTickLabel`, `YTickLabel`, or `ZTickLabel` property for the parent axes always have the same value. Setting one also sets the other.

Example

Relabel the categories along the x-axis.

```
A =  categorical({'r' 'b' 'g' 'g' 'r' 'b' 'b' 'r' 'g' 'y' 'g'});
h = histogram(A);
ax = gca;
ax.XAxis.TickLabels = {'Blue','Green','Red','Yellow'}
```

Data Types: `char` | `string` | `categorical`

TickLabelsMode — Selection mode for TickLabels property

`'auto'` (default) | `'manual'`

Selection mode for the `TickLabels` property, specified as one of these values:

- `'auto'` — Automatically select the tick labels.
- `'manual'` — Use tick labels that you specify. To specify the labels, set the `TickLabels` property.

Note The `TickLabelsMode` property for the ruler and the associated `XTickLabelMode`, `YTickLabelMode`, or `ZTickLabelMode` property for the parent axes always have the same value. Setting one also sets the other.

`TickLabelInterpreter` — Interpretation of tick label characters

`'tex'` (default) | `'latex'` | `'none'`

Interpretation of tick label characters, specified as one of these values:

- `'tex'` — Interpret characters using a subset of TeX markup.
- `'latex'` — Interpret characters using LaTeX markup.
- `'none'` — Display literal characters.

Note Setting the `TickLabelInterpreter` property for the parent axes sets the `TickLabelInterpreter` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.XAxis.TickLabelInterpreter = 'latex';`

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the text type and color, and include special characters in the text.

This table lists the supported modifiers with the `TickLabelInterpreter` property set to `'tex'`. Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they only modify the next character or the text within the curly braces `{}`.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`

Modifier	Description	Example
`\sl`	Oblique font (rarely available)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{specifier}`	Set `specifier` as the name of a font family to change the font style. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{specifier}`	Set `specifier` as a scalar numeric value to change the font size.	`'\fontsize{15} text'`
`\color{specifier}`	Set `specifer` as one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Set `specifier` as a three-element RGB triplet to change the font color.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	β	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\epsilon	ϵ	\Delta	Δ	\spadesuit	♠
\zeta	ζ	\Theta	Θ	\leftrightarrow	↔
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	μ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `TickLabelInterpreter` property to `'latex'`. Use dollar symbols around the labels, for example, use `'$\int_1^{20} x^2 dx$'` for inline mode or `'$$\int_1^{20} x^2 dx$$'` for display mode.

The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup within the text. The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters. For multiline text, the maximum size of the text reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project website at https://www.latex-project.org/.

`TickLabelRotation` — Rotation of tick labels

scalar value in degrees

Rotation of tick labels, specified as a scalar value in degrees. Positive values give counterclockwise rotation. Negative values give clockwise rotation.

Alternatively, you can rotate the tick labels using the `xtickangle`, `ytickangle`, and `ztickangle` functions.

Note The `TickLabelRotation` property for the ruler and the associated `XTickLabelRotation`, `YTickLabelRotation`, or `ZTickLabelRotation` property for the parent axes always have the same value. Setting one also sets the other.

Example: `ax.XAxis.TickLabelRotation = 45;`

Example: `ax.YAxis.TickLabelRotation = -45;`

`TickLabelRotationMode` — Selection mode for `TickLabelRotation` property

`'auto'` (default) | `'manual'`

Selection mode for the `TickLabelRotation` property, specified as one of these values:

- `'auto'` — Automatically select the tick label rotation.
- `'manual'` — Use a tick label rotation that you specify. To specify the rotation, set the `TickLabelRotation` property.

`TickDirection` — Tick mark direction
`'in'` (default) | `'out'` | `'both'`

Tick mark direction, specified as one of these values:

- `'in'` — Direct the tick marks inward from the axis lines. This is the default for 2-D views.
- `'out'` — Direct the tick marks outward from the axis lines. This is the default for 3-D views.
- `'both'` — Center the tick marks over the axis lines.

If you assign a value to this property, then MATLAB sets the `TickDirectionMode` property to `'manual'`.

Note Setting the `TickDir` property for the parent axes sets the `TickDirection` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.YAxis.TickDirection = 'out';`

`TickDirectionMode` — Selection mode for `TickDirection` property
`'auto'` (default) | `'manual'`

Selection mode for the `TickDirection` property, specified as one of these values:

- `'auto'` — Automatically select the tick direction.
- `'manual'` — Use a tick direction that you specify. To specify the tick direction, set the `TickDirection` property.

`TickLength` — Tick mark length
two-element vector

Tick mark length, specified as a two-element vector of the form `[2Dlength 3Dlength]`. The first element is the tick mark length in 2-D views. The second element is the tick mark length in 3-D views. Specify the values in units normalized relative to the longest axes dimension.

Note Setting the `TickLength` property for the parent axes sets the `TickLength` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.YAxis.TickLength = [0.02 0.035];`

`MinorTick` — Minor tick mark display

`'off'` (default) | `'on'`

Minor tick mark display, specified as one of these values:

- `'off'` — Do not display minor tick marks. This is the default value for an axis with a linear scale.
- `'on'` — Display minor tick marks between the major tick marks on the axis. This is the default value for an axis with a log scale. The space between the major tick marks determines the number of minor tick marks. If the `MinorTickValues` property is set to empty `[]`, then no minor tick marks appear. Specify the tick mark locations by setting the `MinorTickValues` property.

Note The `MinorTick` property for the ruler and the associated `XMinorTick`, `YMinorTick`, or `ZMinorTick` property for the parent axes always have the same value. Setting one also sets the other.

Example: `ax.XAxis.MinorTick = 'on';`

`MinorTickValues` — Minor tick mark locations

cell array of character vectors | categorical array | string array

Minor tick mark locations, specified as a cell array of character vectors, a categorical array, or a string array indicating category names. The categories must be a subset of the categories in the `Categories` property and must appear in the same relative order that they appear in the `Categories` property.

If you assign values to this property, then MATLAB sets the `MinorTickValuesMode` property to `'manual'`.

Data Types: `categorical` | `string` | `cell`

`MinorTickValuesMode` — Selection mode for `MinorTickValues` property
`'auto'` (default) | `'manual'`

Selection mode for the `MinorTickValues` property, specified as one of these values:

- `'auto'` — Use automatically calculated minor tick values.
- `'manual'` — Use minor tick values that you specify. To specify the values, set the `MinorTickValues` property.

Font

`FontName` — Font name
system supported font name | `'FixedWidth'`

Font name, specified as a system supported font name or `'FixedWidth'`. The default font depends on the specific system and locale. To use a fixed-width font that renders well, specify `'FixedWidth'`. The actual fixed-width font used depends on the `FixedWidthFontName` property of the root object.

Note Setting the `FontName` property for the parent axes sets the `FontName` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.XAxis.FontName = 'Cambria';`

`FontSize` — Font size
scalar numeric value

Font size, specified as a scalar numeric value. The default font size depends on the specific operating system and locale.

Note Setting the `FontSize` property for the parent axes sets the `FontSize` property for the ruler to the same value. However, setting the ruler property does not set the axes

property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.XAxis.FontSize = 12;`

`FontWeight` — Thickness of text characters
`'normal'` (default) | `'bold'`

Thickness of the text characters, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font.
- `'bold'` — Thicker character outlines than normal.

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold font weight. Therefore, specifying a bold font weight still can result in the normal font weight.

Note Setting the `FontWeight` property for the parent axes sets the `FontWeight` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.XAxis.FontWeight = 'bold';`

`FontAngle` — Text character slant
`'normal'` (default) | `'italic'`

Character slant, specified as `'normal'` or `'italic'`. Not all fonts have both font styles. Therefore, the italic font can look the same as the normal font.

Note Setting the `FontAngle` property for the parent axes sets the `FontAngle` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.XAxis.FontAngle = 'italic';`

FontSmoothing — Text antialiasing

'on' (default) | 'off'

Text smoothing, specified as one of these values:

- 'on' — Enable text antialiasing to reduce the jagged appearance of text characters and make the text easier to read. In certain cases, smoothed text blends against the background color and can make the text appear blurry.
- 'off' — Disable text antialiasing. Use this setting if the text seems blurry.

Note Setting the FontSmoothing property for the parent axes sets the FontSmoothing property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: ax.XAxis.FontSmoothing = 'off';

Parent/Child

Parent — Ruler parent

Axes object

Ruler parent, specified as an Axes object.

Note Ruler objects are not listed in the Children property of the parent Axes object.

Children — Ruler children

empty GraphicsPlaceholder array

The ruler has no children. You cannot set this property.

See Also

Axes

Introduced in R2017a

categories

Categories of categorical array

Syntax

```
C = categories(A)
```

Description

`C = categories(A)` returns a cell array of character vectors containing the categories of the categorical array, `A`.

Examples

List Categories in Categorical Array

Create a categorical array, `A`.

```
A = categorical({'plane' 'car' 'train' 'car' 'plane'})
```

```
A = 1x5 categorical array
     plane      car      train      car      plane
```

`A` is a 1-by-5 categorical array.

Display the categories of `A`.

```
C = categories(A)
```

```
C = 3x1 cell array
    {'car'  }
    {'plane'}
    {'train'}
```

Since you created A by specifying only an input array, the categories appear in alphabetical order.

List Categories in Ordinal Categorical Array

Create an ordinal categorical array.

```
A = categorical({'medium' 'large'; 'small' 'xlarge'; 'large' 'medium'},...
    {'small' 'medium' 'large' 'xlarge'},'Ordinal',true)
```

```
A = 3x2 categorical array
     medium      large
     small       xlarge
     large       medium
```

A is a 3-by-2 ordinal categorical array.

Display the categories of A.

```
C = categories(A)
```

```
C = 4x1 cell array
    {'small' }
    {'medium'}
    {'large' }
    {'xlarge'}
```

The categories appear in the order in which you specified them. Since A is ordinal, the categories have the mathematical ordering `small < medium < large < xlarge`.

Input Arguments

A — Categorical array
vector | matrix | multidimensional array

Categorical array, specified as a vector, matrix, or multidimensional array.

Tips

- `C` includes all categories in `A`, even if `A` does not contain any data from a category. To see the unique values in `A`, use `unique(A)`.
- The order of the categories listed in `C` is the same order used by functions, such as `summary` and `histogram`. To change the order of the categories, use `reordercats`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This function fully supports code generation. For more information, see “Code Generation for Categorical Arrays” (MATLAB Coder).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`addcats` | `categorical` | `histogram` | `iscategory` | `mergecats` | `removecats` | `renamecats` | `reordercats` | `setcats` | `unique`

Introduced in R2013b

caxis

Set colormap limits

Syntax

```
caxis(limits)
caxis('auto')
caxis('manual')
caxis(target, ___ )
```

```
cl = caxis
```

Description

`caxis(limits)` sets the colormap limits for the current axes. `limits` is a two-element vector of the form `[cmin cmax]`. All values in the colormap indexing array on page 1-1312 that are less than or equal to `cmin` map to the first row in the colormap. All values that are greater than or equal to `cmax` map to the last row in the colormap. All values between `cmin` and `cmax` map linearly to the intermediate rows of the colormap.

Note The `caxis` function only affects graphics objects that have the `CDataMapping` property set to `'scaled'`. It does not affect graphics objects that use truecolor or have the `CDataMapping` set to `'direct'`.

`caxis('auto')` enables automatic limit updates when values in the colormap indexing array change. This is the default behavior. The `caxis auto` command is an alternative form of this syntax.

`caxis('manual')` disables automatic limit updates. The `caxis manual` command is an alternative form of this syntax.

`caxis(target, ___ )` sets the colormap limits for a specific axes or chart. Specify `target` as the first input argument in any of the previous syntaxes.

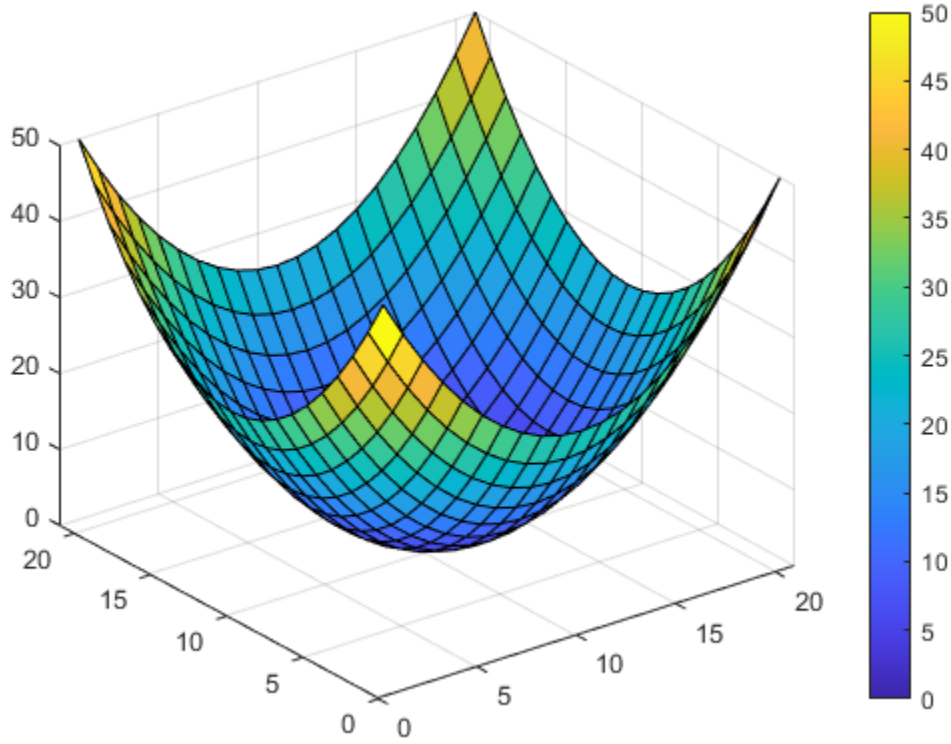
`cl = caxis` returns the current colormap limits for the current axes or chart.

Examples

Raise the Lower Limit

Plot a paraboloid with a colorbar.

```
[X,Y] = meshgrid(-5:.5:5);
Z = X.^2 + Y.^2;
surf(Z);
colorbar
```



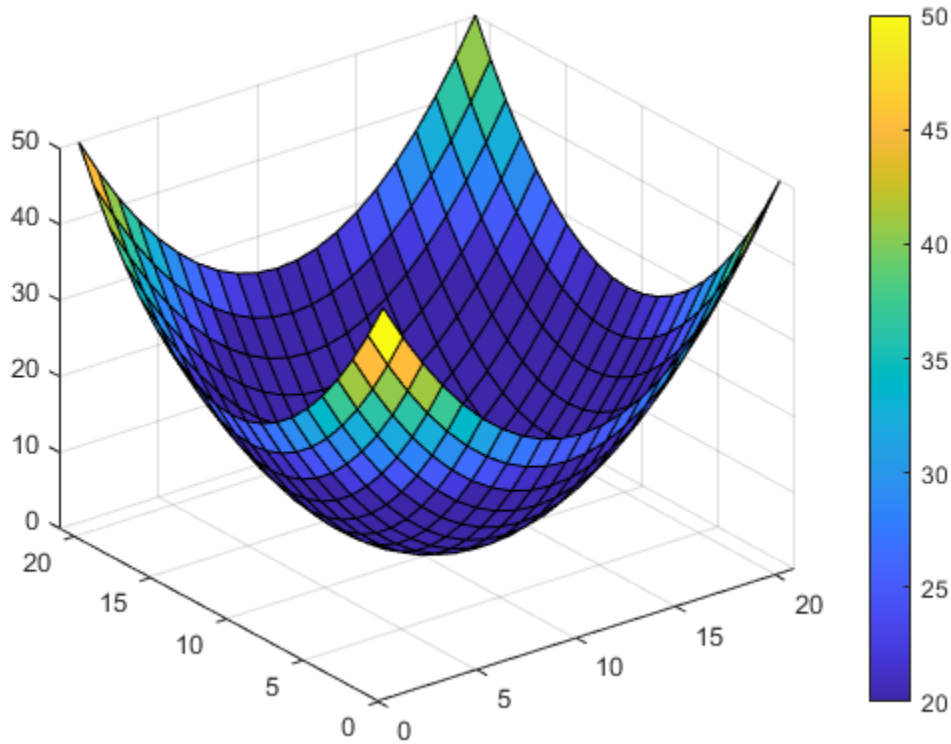
Get the current color limits.

```
lim = caxis
```

```
lim = 1×2

     0    50
```

Raise the lower limit to `20`. Notice that all values of `Z` that are less than or equal to `20` map to the first color.

```
caxis([20 50])
```



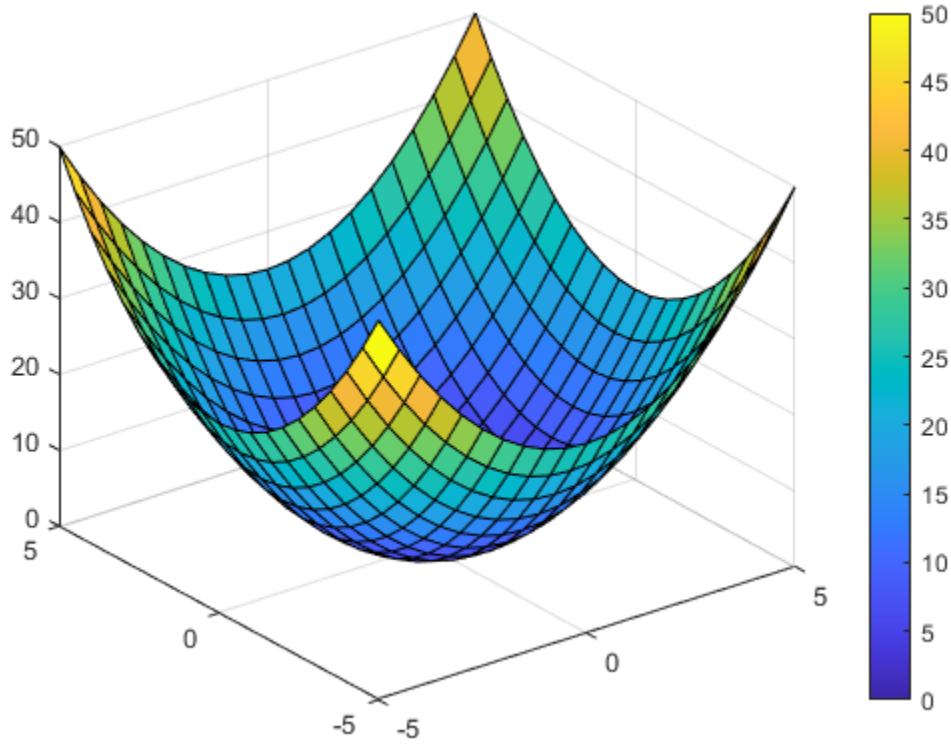
Hold Color Limits for Multiple Surface Plots

Create two paraboloid surfaces that are vertically offset.

```
[X,Y] = meshgrid(-5:.5:5);
Z1 = X.^2 + Y.^2;
Z2 = Z1 + 50;
```

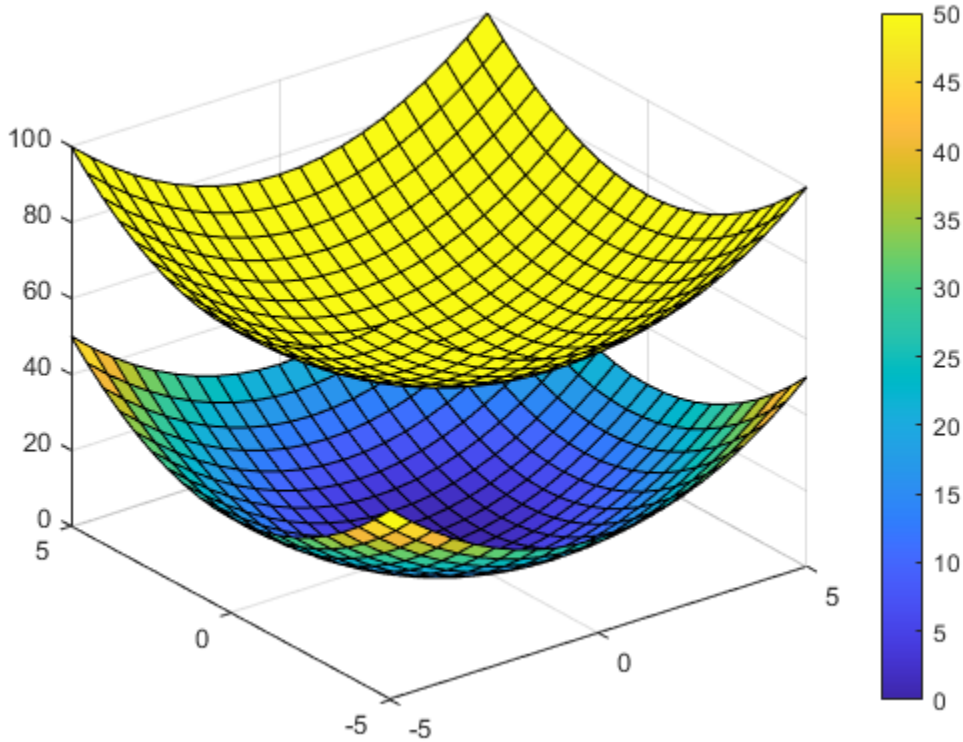
Plot the first paraboloid. Add a colorbar, and hold the axes for the second paraboloid.

```
surf(X,Y,Z1);
colorbar
hold on
```



Hold the current color limits using the `'manual'` option. Then plot the second paraboloid.

```
caxis('manual');
surf(X,Y,Z2);
```

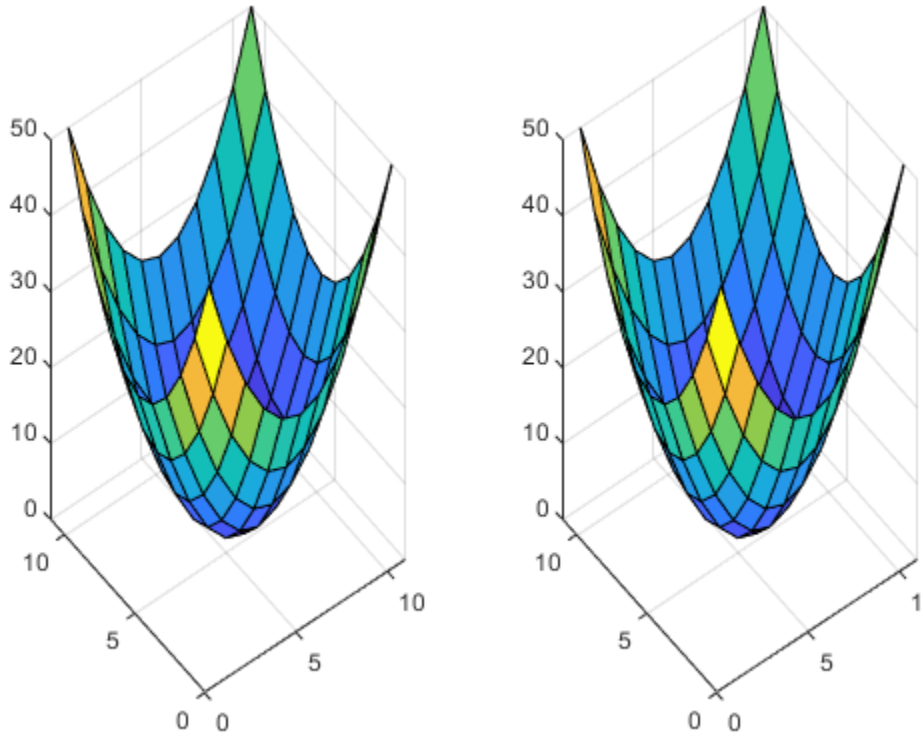


Set Color Limits on a Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 1-by-2 tiled chart layout.

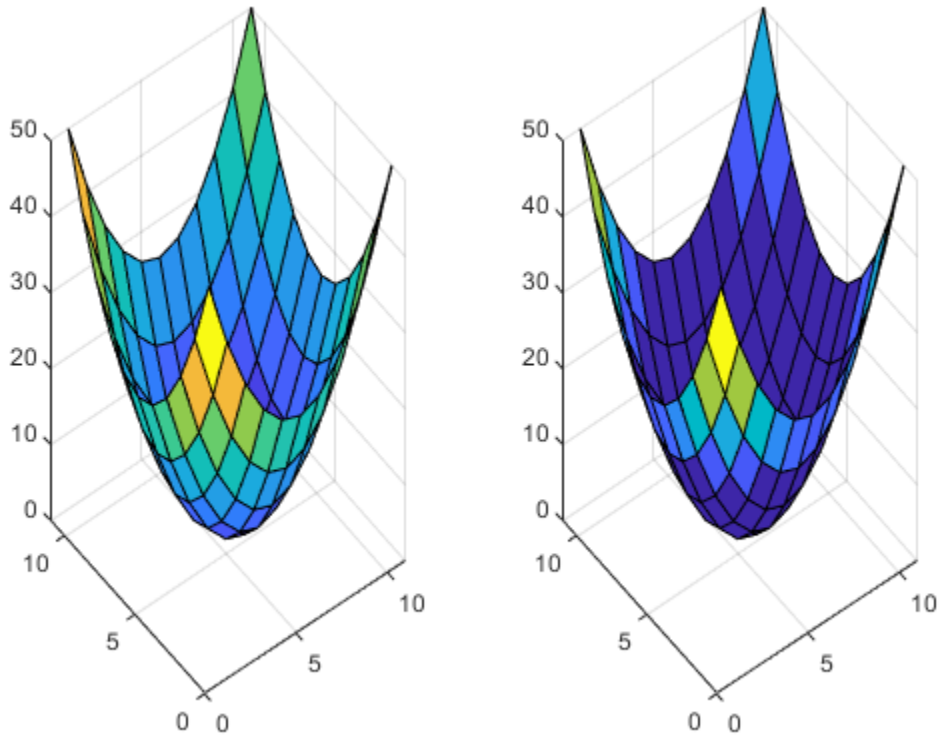
Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Plot a paraboloid into each axes.

```
[X,Y] = meshgrid(-5:1:5);
Z = X.^2 + Y.^2;
tiledlayout(1,2)

% Left plot
ax1 = nexttile;
surf(ax1,Z);

% Right plot
ax2 = nexttile;
surf(ax2,Z);
```



Raise the lower color limit of the right axes to `20`.

```
caxis(ax2,[20 50])
```



Input Arguments

`limits` — New limits
vector of the form `[cmin cmax]`

New limits, specified as a vector of the form `[cmin cmax]`. The value of `cmin` must be less than `cmax`.

Data Types: `single` | `double`

`target` — Target
current axes (default) | `Axes` object | `GeographicAxes` object | graphics object

Target axes or chart, specified as an `Axes` object, a `GeographicAxes` object, or a graphics object that has a `Colormap` property (such as a `HeatMapChart` object).

Output Arguments

`cl` — Current limits
vector of the form `[cmin cmax]`

Current limits of the current axes or chart, returned as a vector of the form `[cmin cmax]`.

More About

Colormap Indexing Array

An array that maps data elements in a chart to specific rows in the colormap. MATLAB stores the indexing array as a property on the graphics object.

For example, the `CData` property of a `Surface` object is an indexing array that maps grid points on the surface to specific rows in the colormap.

See Also

`colormap`

Topics

"Control Colormap Limits"

Introduced before R2006a

cd

Change current folder

Syntax

```
cd
cd newFolder
oldFolder = cd(newFolder)
```

Description

`cd` displays the current folder.

`cd newFolder` changes the current folder to `newFolder`. Folder changes are global. Therefore, if you use `cd` within a function, the folder change persists after MATLAB finishes executing the function.

`oldFolder = cd(newFolder)` returns the existing current folder to `oldFolder`, and then it changes the current folder to `newFolder`.

Examples

Change Current Folder Using Full and Relative Paths

Change the current folder to the featured examples folder for MATLAB R2017a, assuming that version is installed on your `C:` drive.

```
cd 'C:\Program Files\MATLAB\R2017a\examples\matlab_featured'
```

Using the full path, change the current folder from `C:\Program Files\MATLAB\R2017a\examples\matlab_featured` to `C:\Program Files\MATLAB\R2017a`.

```
cd ..\..
```

Use a relative path to change the current folder from `C:\Program Files\MATLAB\R2017a` back to `C:\Program Files\MATLAB\R2017a\examples\matlab_featured`.

```
cd examples\matlab_featured
```

Change, and then Restore Current Folder

Change the current folder to the featured examples folder for MATLAB R2017a, assuming that version is installed on your `C:` drive. Then restore the current folder to its original location.

Change the current folder to the featured examples folder for MATLAB R2017a.

```
cd 'C:\Program Files\MATLAB\R2017a\examples\matlab_featured'
```

Change the current folder to `C:\Program Files`, saving the folder path before changing it.

```
oldFolder = cd('C:\Program Files')
```

```
oldFolder =

    'C:\Program Files\MATLAB\R2017a\examples\matlab_featured'
```

Use the `cd` command to display the new current folder.

```
cd
```

```
C:\Program Files
```

Change the current folder back to the original folder, using the stored path. Use the `cd` command to display the new current folder.

```
cd(oldFolder)
cd
```

```
C:\Program Files\MATLAB\R2017a\examples\matlab_featured
```

Change Current Folder to User Home Folder on UNIX®

Change the current folder to the featured examples folder for MATLAB R2016b, assuming that version is installed in your user home folder on a UNIX platform.

```
cd ~/MATLAB/R2016b/examples/matlab_featured
```

Input Arguments

`newFolder` — New folder path
character vector | string scalar

New folder path to which you want to change the current folder, specified as a character vector or string scalar. If `newFolder` is a string, enclose it in parentheses. For example, `cd("FolderName")`.

Valid values include a full or relative path or one of these values.

`../`	One level up from the current folder.
Multiples of `../`	Multiple levels up from the current folder.
`./`	A path relative to the current folder. `cd` assumes that the path is relative to the current folder even if the `./` is not present.

If `newFolder` contains spaces, enclose it in single quotation marks. For example, `cd 'Folder Name'`.

On UNIX® systems, you can use the ~ (tilde) character to represent the user home folder. For example, `~/` or `~username/`.

MATLAB resolves folder path names containing `'.'`, `'..'`, and symbolic links to their target location before changing the current folder. For example, if you specify `c:\matlab\..\work`, MATLAB resolves the path name to `c:\work` before changing the current folder.

Output Arguments

`oldFolder` — Previous current folder
character vector

Previous current folder that was in place before you issued the `cd` command, returned as a character vector.

More About

Current Folder

The current folder is a reference location that MATLAB uses to find files. This folder is sometimes referred to as the *current directory*, *current working folder*, or *present working directory*.

See Also

`dir` | `pwd` | `what`

Topics

"Specify File Names"
"Files and Folders that MATLAB Accesses"

Introduced before R2006a

convexHull

Class: `DelaunayTri`

(Not recommended) Convex hull

Note `convexHull(DelaunayTri)` is not recommended. Use `convexHull(delaunayTriangulation)` instead.

`DelaunayTri` is not recommended. Use `delaunayTriangulation` instead.

Syntax

```
K = convexHull(DT)
[K AV] = convexHull(DT)
```

Description

`K = convexHull(DT)` returns the indices into the array of points `DT.X` that correspond to the vertices of the convex hull.

`[K AV] = convexHull(DT)` returns the convex hull and the area or volume bounded by the convex hull.

Input Arguments

`DT`	Delaunay triangulation.

Output Arguments

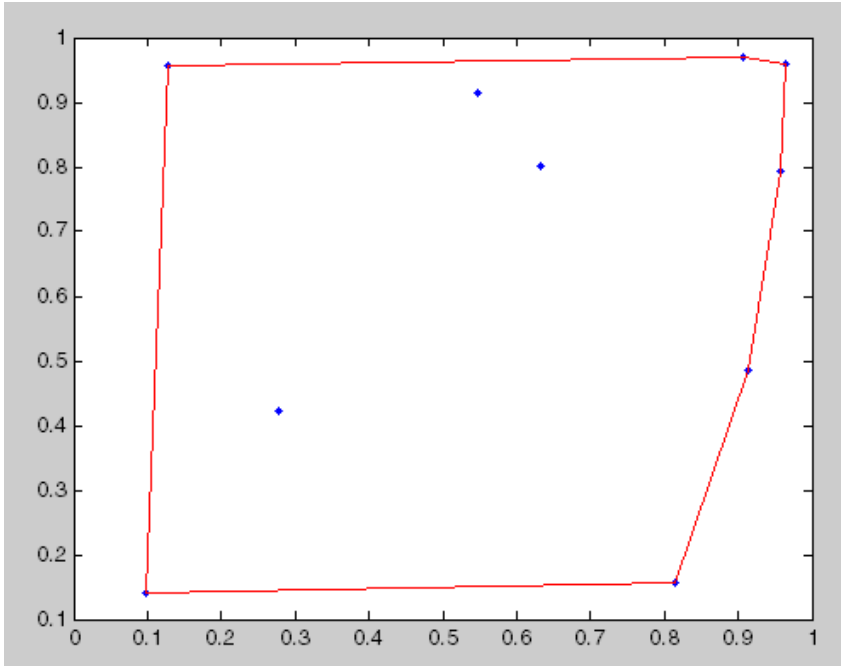
K	If the points lie in 2-D space, K is a column vector of length numf. Otherwise K is a matrix of size numf-by-ndim, numf being the number of facets in the convex hull, and ndim the dimension of the space where the points reside.
AV	The area or volume of the convex hull.

Examples

Example 1

Compute the convex hull of a set of random points located within a unit square in 2-D space.

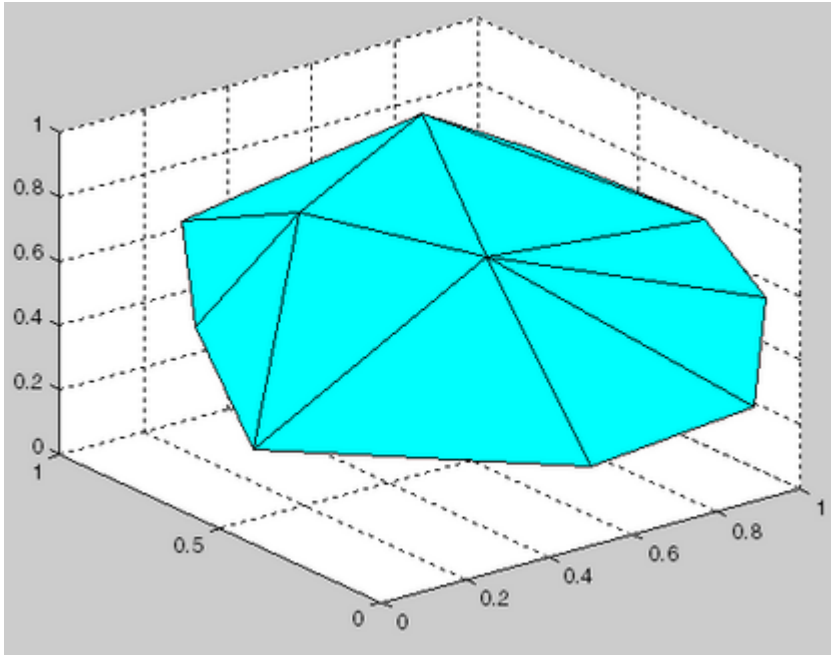
```
x = rand(10,1)
y = rand(10,1)
dt = DelaunayTri(x,y)
k = convexHull(dt)
plot(dt.X(:,1),dt.X(:,2), '.', 'markersize',10); hold on;
plot(dt.X(k,1),dt.X(k,2), 'r'); hold off;
```



Example 2

Compute the convex hull of a set of random points located within a unit cube in 3-D space, and the volume bounded by the convex hull.

```
X = rand(25,3)
dt = DelaunayTri(X)
[ch v] = convexHull(dt)
trisurf(ch, dt.X(:,1),dt.X(:,2),dt.X(:,3), 'FaceColor', 'cyan')
```



More About

Convex Hull

The convex hull of a set of points X is the smallest convex region containing all of the points of X.

See Also

`convhull` | `convhulln` | `delaunayTriangulation` | `triangulation` | `voronoiDiagram`

cd

Change or view current folder on FTP server

Syntax

```
cd(ftpobj,folder)
cd(ftpobj)
newFolder = cd( ___ )
```

Description

`cd(ftpobj,folder)` changes the current folder on the FTP server associated with `ftpobj`.

`cd(ftpobj)` displays the path to the current folder on the server.

`newFolder = cd( ___ )` also returns the path as a character vector. You can use the input arguments from either of the previous syntaxes. If you use the first syntax, then `newFolder` is the path after you change the folder.

Examples

Change Folder

Connect to an FTP server and navigate to a folder on that server.

First, connect to the National Centers for Environmental Information (NCEI) FTP server.

```
ftpobj = ftp('ftp.ngdc.noaa.gov')
```

```
ftpobj = 

  FTP Object
     host: ftp.ngdc.noaa.gov
```

```
     user: anonymous
      dir: /
     mode: binary
```

Display the contents of the top-level folder. It contains another folder named pub.

```
dir(ftpobj)
```

```
DMSP                          Solid_Earth                   google12c4c939d7b90761.html
INDEX.txt                     coastwatch                    hazards
README.txt                    dmsp4alan                     index.html
STP                           ftp.html                      international
Snow_Ice                      geomag                        ionosonde
```

Change folders to the pub folder. Then display its contents.

```
cd(ftpobj,'pub')
```

```
ans =
'/pub'
```

```
dir(ftpobj)
```

```
WebCD     coast     glac_lib  krm       outgoing  results   rgon
```

FTP service courtesy of the National Centers for Environmental Information (NCEI). See the NCEI Privacy Policy, Disclaimer, and Copyright for NCEI terms of service.

Input Arguments

ftpobj — Connection to FTP server
FTP object

Connection to an FTP server, specified as an FTP object.

folder — Name of target folder
character vector | string scalar

Name of the target folder on the FTP server, specified as a character vector or string scalar. To specify the folder above the current one, use `'..'`.

See Also

`dir` | `ftp`

Introduced before R2006a

cdf2rdf

Convert complex diagonal form to real block diagonal form

Syntax

```
[V,D] = cdf2rdf(V,D)
```

Description

If the eigensystem `[V,D] = eig(X)` has complex eigenvalues appearing in complex-conjugate pairs, `cdf2rdf` transforms the system so `D` is in real diagonal form, with 2-by-2 real blocks along the diagonal replacing the complex pairs originally there. The eigenvectors are transformed so that

```
X = V*D/V
```

continues to hold. The individual columns of `V` are no longer eigenvectors, but each pair of vectors associated with a 2-by-2 block in `D` spans the corresponding invariant vectors.

Examples

The matrix

```
X =
    1     2     3
    0     4     5
    0    -5     4
```

has a pair of complex eigenvalues.

```
[V,D] = eig(X)

V =

    1.0000      -0.0191 - 0.4002i     -0.0191 + 0.4002i
         0            0 - 0.6479i           0 + 0.6479i
```

```
         0       0.6479                0.6479

D =

    1.0000            0                     0
         0       4.0000 + 5.0000i           0
         0            0                4.0000 - 5.0000i
```

Converting this to real block diagonal form produces

```
[V,D] = cdf2rdf(V,D)

V =

    1.0000    -0.0191     -0.4002
         0          0     -0.6479
         0     0.6479           0

D =

    1.0000          0           0
         0     4.0000      5.0000
         0    -5.0000      4.0000
```

Algorithms

The real diagonal form for the eigenvalues is obtained from the complex form using a specially constructed similarity transformation.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`eig` | `rsf2csf`

Introduced before R2006a

cdfepoch

Convert date text or serial date number to CDF formatted dates

Syntax

```
E = cdfepoch(date)
```

Description

`E = cdfepoch(date)` converts the date, specified by `date`, into a `cdfepoch` object. `date` must be valid date value represented by text, as returned by `datestr`, or a serial date number, returned by `datenum`. `date` can also be a `cdfepoch` object.

When writing data to a CDF file using `cdfwrite`, use `cdfepoch` to convert MATLAB dates or serial date numbers to CDF formatted dates. The MATLAB `cdfepoch` object simulates the `CDFEPOCH` data type in CDF files.

To convert a `cdfepoch` object into a MATLAB serial date number, use the `todatenum` function.

Examples

Convert the current time in serial date number format into a CDF epoch object.

```
% NOW function returns current time as serial date number
dateobj = cdfepoch(now)

dateobj =

     cdfepoch object:
     11-Mar-2009 15:09:25
```

Convert the current time, which is returned by `datestr` in text form, into a CDF epoch object.

```
% DATESTR function returns text representing a date
dateobj2 = cdfepoch(datestr(now))

dateobj2 =

     cdfepoch object:
     11-Mar-2009 15:09:25
```

Convert the CDF epoch object into a serial date number.

```
dateobj = cdfepoch(now);
mydatenum = todatenum(dateobj)

mydatenum =

     7.3384e+005
```

More About

MATLAB Serial Date Number

A MATLAB serial date number represents the whole and fractional number of days from 0-Jan-0000 to a specific date. The year 0000 is merely a reference point and is not intended to be interpreted as a real year in time. The MATLAB serial date number calculates dates differently than CDF epochs.

CDF Epoch

A CDF epoch is the number of milliseconds since 1-Jan-0000.

See Also

`cdfinfo` | `cdfread` | `datenum` | `datestr` | `datetime` | `todatenum`

Introduced before R2006a

cdfinfo

Information about Common Data Format (CDF) file

Syntax

```
info = cdfinfo(filename)
```

Description

`info = cdfinfo(filename)` returns information about the Common Data Format (CDF) file specified by `filename`. Specify `filename` as a character vector or string scalar.

Note Because `cdfinfo` creates temporary files, the current working directory must be writeable.

The following table lists the fields returned in the structure, `info`. The table lists the fields in the order that they appear in the structure.

Field	Description
`Filename`	Character vector specifying the name of the file
`FileModDate`	Character vector indicating the date the file was last modified
`FileSize`	Double scalar specifying the size of the file, in bytes
`Format`	Character vector specifying the file format
`FormatVersion`	Character vector specifying the version of the CDF library used to create the file
`FileSettings`	Structure array containing library settings used to create the file

Field	Description	
`Subfiles`	Filenames containing the CDF file's data, if it is a multi-file format CDF	
`Variables`	N-by-6 cell array, where N is the number of variables, containing information about the variables in the file. The columns present the following information:	
	Column 1	Character vector specifying name of variable
	Column 2	Double array specifying the dimensions of the variable, as returned by the `size` function
	Column 3	Double scalar specifying the number of records assigned for the variable
	Column 4	Character vector specifying the data type of the variable, as stored in the CDF file

	Column 5	Character vector specifying the record and dimension variance settings for the variable. The single T or F to the left of the slash designates whether values vary by record. The zero or more T or F letters to the right of the slash designate whether values vary at each dimension. Here are some examples.
		`T/  (scalar variable`
		`F/T (one-dimensional variable)`
		T/TFF (three-dimensional variable)

`GlobalAttributes`	Structure array that contains one field for each global attribute. The name of each field corresponds to the name of an attribute. The data in each field, contained in a cell array, represents the entry values for that attribute.	
`VariableAttributes`	Structure array that contains one field for each variable attribute. The name of each field corresponds to the name of an attribute. The data in each field is contained in a *n*-by-2 cell array, where *n* is the number of variables. The first column of this cell array contains the variable names associated with the entries. The second column contains the entry values.	

Note Attribute names returned by `cdfinfo` might not match the names of the attributes in the CDF file exactly. Attribute names can contain characters that are illegal in MATLAB field names. `cdfinfo` removes illegal characters that appear at the beginning of attributes and replaces other illegal characters with underscores ('_'). When `cdfinfo` modifies an attribute name, it appends the attribute's internal number to the end of the field name. For example, the attribute name `Variable%Attribute` becomes `Variable_Attribute_013`.

Note To improve performance, turn off the file validation which the CDF library does by default when opening files. For more information, see `cdflib.setValidate`.

Examples

Get Information About CDF File

Get information about the sample file, `example.cdf`.

```
info = cdfinfo('example.cdf')
```

```
info = struct with fields:
                Filename: 'example.cdf'
             FileModDate: '10-May-2010 21:35:01'
                FileSize: 1310
                  Format: 'CDF'
           FormatVersion: '2.7.0'
            FileSettings: [1x1 struct]
                Subfiles: {}
               Variables: {6x6 cell}
        GlobalAttributes: [1x1 struct]
      VariableAttributes: [1x1 struct]
```

View information about the variables in the file.

```
info.Variables
```

```
ans=6×6 cell
  Columns 1 through 5

    {'Time'            }    {1x2 double}    {[24]}    {'epoch' }    {'T/'    }
```

```
    {'Longitude'       }    {1x2 double}    {[ 1]}    {'int8'  }    {'F/FT'  }
    {'Latitude'        }    {1x2 double}    {[ 1]}    {'int8'  }    {'F/TF'  }
    {'Data'            }    {1x3 double}    {[ 1]}    {'double'}    {'T/TTT' }
    {'multidimensional'}    {1x4 double}    {[ 1]}    {'uint8' }    {'T/TTTT'}
    {'Temperature'     }    {1x2 double}    {[10]}    {'int16' }    {'T/TT'  }

  Column 6

    {'Full'}
    {'Full'}
    {'Full'}
    {'Full'}
    {'Full'}
    {'Full'}
```

See Also

`cdflib.setValidate` | `cdfread`

Introduced before R2006a

cdflib

Interact directly with CDF library

Description

MATLAB provides direct access to dozens of functions in the CDF library. Using these functions, you can read and write data, create variables, attributes, and entries, and take advantage of other features of the CDF library. To use these functions, you must be familiar with the CDF C interface. Documentation about CDF is available at the CDF website.

The MATLAB functions correspond to functions in the CDF library new Standard Interface. In most cases, the syntax of a MATLAB function is similar to the syntax of the corresponding CDF library function. To use these functions, you must prefix the function name with the package name, `cdflib`. For example, to use the CDF library function to open an existing CDF file, use this syntax:

```
cdfid = cdflib.open('example.cdf');
```

MATLAB supports CDF version 3.6.1. For copyright information, see the `cdfcopyright.txt` file.

The following tables list all of the functions in the MATLAB CDF library package, grouped by category.

Library Information

cdflib.getConstantNames	Names of Common Data Format (CDF) library constants
cdflib.getConstantValue	Numeric value corresponding to Common Data Format (CDF) library constant
cdflib.getFileBackward	Return current backward compatibility mode setting
cdflib.getLibraryCopyright	Copyright notice of Common Data Format (CDF) library
cdflib.getLibraryVersion	Library version and release information
cdflib.getValidate	Library validation mode
cdflib.setFileBackward	Set backward compatibility mode
cdflib.setValidate	Specify library validation mode

File Operations

cdflib.close	Close Common Data Format (CDF) file
cdflib.create	Create Common Data Format (CDF) file
cdflib.delete	Delete existing Common Data Format (CDF) file
cdflib.getCacheSize	Number of cache buffers used
cdflib.getChecksum	Checksum mode
cdflib.getCompression	Compression settings
cdflib.getCompressionCacheSize	Number of compression cache buffers
cdflib.getCopyright	Copyright notice in Common Data Format (CDF) file
cdflib.getFormat	Format of Common Data Format (CDF) file
cdflib.getMajority	Majority of variables
cdflib.getName	Name of Common Data Format (CDF) file
cdflib.getReadOnlyMode	Read-only mode
cdflib.getStageCacheSize	Number of cache buffers for staging
cdflib.getVersion	Common Data Format (CDF) library version and release information
cdflib.inquire	Basic characteristics of Common Data Format (CDF) file
cdflib.open	Open existing Common Data Format (CDF) file
cdflib.setCacheSize	Specify number of dotCDF cache buffers
cdflib.setChecksum	Specify checksum mode
cdflib.setCompression	Specify compression settings
cdflib.setCompressionCacheSize	Specify number of compression cache buffers
cdflib.setFormat	Specify format of Common Data Format (CDF) file
cdflib.setMajority	Specify majority of variables
cdflib.setReadOnlyMode	Specify read-only mode
cdflib.setStageCacheSize	Specify number of staging cache buffers for Common Data Format (CDF) file

Variables

cdflib.closeVar	Close specified variable from multifile format Common Data Format (CDF) file
cdflib.createVar	Create new variable
cdflib.deleteVar	Delete variable
cdflib.deleteVarRecords	Delete range of records from variable
cdflib.getVarAllocRecords	Number of records allocated for variable
cdflib.getVarBlockingFactor	Blocking factor for variable
cdflib.getVarCacheSize	Number of multifile cache buffers
cdflib.getVarCompression	Information about compression used by variable
cdflib.getVarData	Single value from record in variable
cdflib.getVarMaxAllocRecNum	Maximum allocated record number for variable
cdflib.getVarMaxWrittenRecNum	Maximum written record number for variable
cdflib.getVarsMaxWrittenRecNum	Maximum written record number for CDF file
cdflib.getVarName	Variable name, given variable number
cdflib.getVarNum	Variable number, given variable name
cdflib.getVarNumRecsWritten	Number of records written to variable
cdflib.getVarPadValue	Pad value for variable
cdflib.getVarRecordData	Entire record for variable
cdflib.getVarReservePercent	Compression reserve percentage for variable
cdflib.getVarSparseRecords	Information about how variable handles sparse records
cdflib.hyperGetVarData	Read hyperslab of data from variable
cdflib.hyperPutVarData	Write hyperslab of data to variable
cdflib.inquireVar	Information about variable
cdflib.putVarData	Write single value to variable
cdflib.putVarRecordData	Write entire record to variable
cdflib.renameVar	Rename existing variable
cdflib.setVarAllocBlockRecords	Specify range of records to be allocated for variable
cdflib.setVarBlockingFactor	Specify blocking factor for variable
cdflib.setVarCacheSize	Specify number of multi-file cache buffers for variable
cdflib.setVarCompression	Specify compression settings used with variable
cdflib.setVarInitialRecs	Specify initial number of records written to variable
cdflib.setVarPadValue	Specify pad value used with variable
cdflib.SetVarReservePercent	Specify reserve percentage for variable

Attributes

cdflib.createAttr	Create attribute
cdflib.deleteAttr	Delete attribute
cdflib.deleteAttrEntry	Delete attribute entry
cdflib.deleteAttrgEntry	Delete entry in global attribute
cdflib.getAttrEntry	Value of entry in attribute with variable scope
cdflib.getAttrgEntry	Value of entry in global attribute
cdflib.getAttrMaxEntry	Number of last entry for variable attribute
cdflib.getAttrMaxgEntry	Number of last entry for global attribute
cdflib.getAttrName	Name of attribute, given attribute number
cdflib.getAttrNum	Attribute number, given attribute name
cdflib.getAttrScope	Scope of attribute
cdflib.getNumAttrEntries	Number of entries for attribute with variable scope
cdflib.getNumAttrgEntries	Number of entries for attribute with global scope
cdflib.getNumAttributes	Number of attributes with variable scope
cdflib.getNumgAttributes	Number of attributes with global scope
cdflib.inquireAttr	Information about attribute
cdflib.inquireAttrEntry	Information about entry in attribute with variable scope
cdflib.inquireAttrgEntry	Information about entry in attribute with global scope
cdflib.putAttrEntry	Write value to entry in attribute with variable scope
cdflib.putAttrgEntry	Write value to entry in attribute with global scope
cdflib.renameAttr	Rename existing attribute

Utility Functions

cdflib.computeEpoch	Convert time value to `CDF_EPOCH` value
cdflib.computeEpoch16	Convert time value to `CDF_EPOCH16` value
cdflib.epoch16Breakdown	Convert `CDF_EPOCH16` value to time value
cdflib.epochBreakdown	Convert `CDF_EPOCH` value into time value

Limitations

- CDF files do not support non-ASCII encoded inputs. Variable names, attributes names, variable values, and attribute values must have 7-bit ASCII encoding. Attempting to write non-ASCII encoded inputs results in an error or data with corrupted characters.
- The CDF library version 3.6.1 does not support the data types `CDF_TIME_TT2000` and `CDF_INT8`.

See Also

`cdfinfo` | `cdfread`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.close

Close Common Data Format (CDF) file

Syntax

```
cdflib.close(cdfId)
```

Description

`cdflib.close(cdfId)` closes the specified CDF file. `cdfId` identifies the CDF file.

You must close a CDF to guarantee that all modifications you made since opening the CDF are actually written to the file.

Examples

Open the example CDF file and then close it.

```
cdfid = cdflib.open('example.cdf');
cdflib.close(cdfid)
```

References

This function corresponds to the CDF library C API routine `CDFcloseCDF`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.create` | `cdflib.open`

Topics

“Import CDF Files Using Low-Level Functions”
“Export to CDF Files”

cdflib.closeVar

Close specified variable from multifile format Common Data Format (CDF) file

Syntax

```
cdflib.closeVar(cdfId,varNum)
```

Description

`cdflib.closeVar(cdfId,varNum)` closes a variable in a multifile format CDF.

`cdfId` identifies the CDF file and `varNum` is a numeric value that specifies the variable. Variable identifiers (variable numbers) are zero-based.

For multifile CDFs, you must close all open variable files to guarantee that all modifications you have made are actually written to the CDF file(s). You do not need to call this function for variables in a single-file format CDF.

Examples

Create a multifile CDF, create a variable, and then close the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_multifile.cdf');

% Make it a multifile format CDF
cdflib.setFormat(cdfid,'MULTI_FILE')

% Create a variable in the CDF.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

%  Close the variable.
cdflib.closeVar(cdfid, varnum)

% Clean up
```

```
cdflib.delete(cdfid)
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFclosezVar`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getFormat` | `cdflib.getVarNum` | `cdflib.setFormat`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.computeEpoch

Convert time value to `CDF_EPOCH` value

Syntax

```
epoch = cdflib.computeEpoch(timeval)
```

Description

`epoch = cdflib.computeEpoch(timeval)` converts the time value specified by `timeval` into a `CDF_EPOCH` value.

Input Arguments

`timeval`

7-by-1 time vector. The following table describes the time components.

Component	Description
year	AD e.g. 1994
month	1–12
day	1–31
hour	0–23
minute	0–59
second	0–59
millisecond	0–999

Output Arguments

epoch

MATLAB `double` representing a `CDF_EPOCH` time value.

Examples

Convert a time value into a `CDF_EPOCH` value.

```
timeval = [1999 12 31 23 59 59 0];
epoch = cdflib.computeEpoch(timeval);
```

References

This function corresponds to the CDF library C API routine `computeEPOCH`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.computeEpoch16` | `cdflib.epoch16Breakdown` | `cdflib.epochBreakdown`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.computeEpoch16

Convert time value to `CDF_EPOCH16` value

Syntax

```
epoch16 = cdflib.computeEpoch16(timeval)
```

Description

`epoch16 = cdflib.computeEpoch16(timeval)` converts the time value specified by `timeval` into a `CDF_EPOCH16` value.

Input Arguments

`timeval`

10-by-1 time vector. The following table describes the time components. To specify multiple time values, use additional columns.

Component	Description
year	AD e.g. 1994
month	1–12
day	1–31
hour	0–23
minute	0–59
second	0–59
millisecond	0–999
microsecond	0–999
nanosecond	0–999

Component	Description
picosecond	0–999

Output Arguments

epoch16

CDF Epoch16 time value. If the input argument `timeval` has `m-by-10` elements, the return value `epoch16` will have size `2-by-m`

Examples

Convert the time value into an `CDF_EPOCH16` value:

```
timeval = [1999; 12; 31; 23; 59; 59; 50; 100; 500; 999];
epoch16 = cdflib.computeEpoch16(timeval);
```

References

This function corresponds to the CDF library C API routine `computeEPOCH16`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.computeEpoch` | `cdflib.epoch16Breakdown` | `cdflib.epochBreakdown`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.create

Create Common Data Format (CDF) file

Syntax

`cdfId = cdflib.create(filename)`

Description

`cdfId = cdflib.create(filename)` creates a new CDF file with the name contained in *`filename`*. Specify *`filename`* as a character vector or string scalar. Returns the CDF file identifier `cdfId`.

Examples

Create a CDF file. To run this example, you must have write permission in your current directory.

```
cdfId = cdflib.create('myfile.cdf');

% Clean up
cdflib.delete(cdfId);

clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFcreateCDF`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.close` | `cdflib.open`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.createAttr

Create attribute

Syntax

```
attrnum = cdflib.createAttr(cdfId,attrName,scope)
```

Description

`attrnum = cdflib.createAttr(cdfId,attrName,scope)` creates an attribute in a CDF file with the specified scope.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

attrName

Name you want to assign to the attribute, specified as a character vector or string scalar.

scope

Scope of the attribute, specified as one of these character vectors or string scalars, or its numeric equivalent.

Value	Description
`'global_scope'`	Attribute applies to the CDF as a whole.
`'variable_scope'`	Attribute applies only to the variable

To get the numeric equivalent of these constants, use the `cdflib.getConstantValue` function.

Output Arguments

attrNum

Numeric value identifying the attribute. Attribute numbers are zero-based.

Examples

Create a CDF, and then create an attribute in the CDF. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create attribute
attrNum = cdflib.createAttr(cdfId,'Purpose','global_scope');

% Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFcreateAttr`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.deleteAttr` | `cdflib.getAttrNum` | `cdflib.getConstantNames` | `cdflib.getConstantValue`

Topics

"Import CDF Files Using Low-Level Functions"

“Export to CDF Files”

matlab.settings.FactoryGroup.createToolboxGroup

Package: `matlab.settings`

Create `FactoryGroup` root object for toolbox

Syntax

```
s = matlab.settings.FactoryGroup.createToolboxGroup(name)
s = matlab.settings.FactoryGroup.createToolboxGroup( ___ ,Name,Value)
```

Description

`s = matlab.settings.FactoryGroup.createToolboxGroup(name)` creates the root factory group for a toolbox and returns the new group as a `FactoryGroup` object. By default,the root factory group is hidden, which means that it does not display in the parent settings group.

`s = matlab.settings.FactoryGroup.createToolboxGroup( ___ ,Name,Value)` specifies the root factory group properties using one or more name-value pair arguments. For example, `'Hidden',false` creates a visible `FactoryGroup` root object. Specify name-value pairs after all other input arguments.

Examples

Create Root Factory Settings Group

Create the root factory group for the toolbox `mytoolbox` and make it visible.

```
s = matlab.settings.FactoryGroup.createToolboxGroup('mytoolbox', ...
    'Hidden',false);
```

Validate Settings in Root Factory Settings Group

Create a root factory group and specify a default validation function. This function validates the values of all the factory settings within the group that do not specify their own validation functions. This includes settings in subgroups, as long as the subgroup or settings do not specify their own validation functions.

First, create a validation function `numericValidationFcn` that throws an error when the input is not numeric.

```
function numericValidationFcn(x)
    errorMsg = 'Value must be numeric.';
    assert(isnumeric(x),errorMsg);
end
```

Create the root factory group `mytoolbox`. Specify the validation function `numericValidationFcn`.

```
s = matlab.settings.FactoryGroup.createToolboxGroup('mytoolbox', ...
    'Hidden',false,'ValidationFcn',@numericValidationFcn);
```

MATLAB throws an error whenever a setting within the root factory settings group is set to a nonnumeric value.

Input Arguments

`name` — Name of toolbox root factory group
character vector | string

Name of toolbox root factory group, specified as a character vector or string. If the root factory group `name` exists, MATLAB displays an error.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `matlab.settings.FactoryGroup.createToolboxGroup('myToolbox','Hidden',false)` creates a visible root factory group.

`Hidden` — Hidden state
`true` (default) | `false`

Hidden state, specified as `true` or `false`.

When set to `true`, the factory group, including all factory groups and factory settings within the group do not display, although they remain accessible.

`ValidationFcn` — Function to validate settings in group
function handle

Function to validate factory settings in group, specified as a function handle. When specified, the function is used to validate the values of all factory settings within the group that do not specify their own validation functions. This includes settings in subgroups, as long as the subgroup or settings do not specify their own validation functions.

The function handle must be associated with a function that accepts the potential setting value as an input argument, has no output arguments, and throws an error if the validation fails.

The function handle must point to a function on the MATLAB path. Anonymous or nested function handles are not supported.

See Also

`addGroup` | `addSetting`

Topics

"Create Factory Settings for Toolboxes"
"Access and Modify Settings"

Introduced in R2019b

cdflib.createVar

Create new variable

Syntax

```
varnum = cdflib.createVar(cdfId, varname, datatype, numElements,
dims, recVariance, dimVariance)
```

Description

`varnum = cdflib.createVar(cdfId, varname, datatype, numElements, dims, recVariance, dimVariance)` creates a new variable in the Common Data Format (CDF) file with the specified characteristics.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

varname

Character vector or string scalar that specifies the name you want to assign to the variable.

datatype

Data type of the variable, specified as one of the following character vectors or string scalars containing a valid CDF data type, or its numeric equivalent.

CDF Data Type	Description
'CDF_BYTE	1-byte, signed integer

CDF Data Type	Description
`'CDF_CHAR'`	1 byte, signed character data type that maps to the MATLAB `char` or `string` class
`'CDF_INT1'`	1-byte, signed integer
`'CDF_UCHAR'`	1 byte, unsigned character data type that maps to the MATLAB `uint8` class
`'CDF_UINT1'`	1-byte, unsigned integer
`'CDF_INT2'`	2-byte, signed integer
`'CDF_UINT2'`	2-byte, unsigned integer
`'CDF_INT4'`	4-byte, signed integer
`'CDF_UINT4'`	4-byte, unsigned integer
`'CDF_FLOAT'`	4-byte, floating point
`'CDF_REAL4'`	4-byte, floating point
`'CDF_REAL8'`	8-byte, floating point.
`'CDF_DOUBLE'`	8-byte, floating point
`'CDF_EPOCH'`	8-byte, floating point
`'CDF_EPOCH16'`	two 8-byte, floating point

`numElements`

Number of elements per datum. Value should be 1 for all data types, except for `'CDF_CHAR'` and `'CDF_UCHAR'`.

`dims`

A vector of the dimensions extents; empty if there are no dimension extents.

`recVariance`

Specifies record variance: `true` or `false`.

`dimVariance`

A vector of logicals; empty if there are no dimensions.

Output Arguments

varNum

The numeric identifier for the variable. Variable numbers are zero-based.

Examples

Create a CDF file and then create a variable named `'Time'` in the CDF. The variable has no dimensions and varies across records. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Initially the file contains no variables.
info = cdflib.inquire(cdfid)

info =

     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: -1
      numVars: 0
    numvAttrs: 0
    numgAttrs: 0

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Retrieve info about the file again to verify variable was created.
% Note value of numVars field is now 1.
info = cdflib.inquire(cdfid)

info =

     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: -1
      numVars: 1
    numvAttrs: 0
    numgAttrs: 0
```

```
% Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFcreatezVar`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.closeVar` | `cdflib.deleteVar`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.delete

Delete existing Common Data Format (CDF) file

Syntax

```
cdflib.delete(cdfId)
```

Description

`cdflib.delete(cdfId)` deletes the existing CDF file specified by the identifier `cdfId`. If the CDF file is a multi-file format CDF, the `cdflib.delete` function also deletes the variable files (having file extensions of `.z0`, `.z1`, etc.).

Examples

Create a CDF file, and then delete it. To run this example, you must be in a writable folder.

```
cdfId = cdflib.create('mytempfile.cdf');

% Verify that the file was created.
ls *.cdf

mytempfile.cdf

% Delete the file.
cdflib.delete(cdfId)

% Verify that the file no longer exists.
ls *.cdf
```

References

This function corresponds to the CDF library C API routine `CDFdeleteCDF`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.create` | `cdflib.setFormat`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.deleteAttr

Delete attribute

Syntax

```
cdflib.deleteAttr(cdfId,attrNum)
```

Description

`cdflib.deleteAttr(cdfId,attrNum)` deletes the specified attribute from the CDF file.

`cdfId` identifies the Common Data Format (CDF) file.`attrNum` is a numeric identifier that specifies the attribute. Attribute numbers are zero-based.

Examples

Create a CDF file, and then create an attribute in the file. Then delete the attribute. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create attribute.
attrNum = cdflib.createAttr(cdfId,'Purpose','global_scope');

% Prove it exists.
anum = cdflib.getAttrNum(cdfid,'Purpose')

anum =

     0

% Delete the attribute.
cdflib.deleteAttr(cdfid,attrNum);
```

```
% Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFdeleteAttr`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.createAttr` | `cdflib.getAttrNum`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.deleteAttrEntry

Delete attribute entry

Syntax

```
cdflib.deleteAttrEntry(cdfId,attrNum,entryNum)
```

Description

`cdflib.deleteAttrEntry(cdfId,attrNum,entryNum)` deletes an entry from an attribute in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

attrNum

Numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have variable scope.

entryNum

Numeric value that specifies the entry in the attribute. Entry numbers are zero-based.

Examples

Create a CDF, and then create an attribute in the file. Write a value to an entry for the attribute, and then delete the entry. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');
```

```
% Initially the file contains no attributes, global or variable.
info = cdflib.inquire(cdfid)

info =

     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: -1
      numVars: 0
    numvAttrs: 0
    numgAttrs: 0

% Create an attribute with variable scope in the file.
attrNum = cdflib.createAttr(cdfid,'my_var_scope_attr','variable_scope');

% Write a value to an entry for the attribute
cdflib.putAttrEntry(cdfid,attrNum,0,'CDF_CHAR','My attr value');

% Get the value of the attribute entry
value = cdflib.getAttrEntry(cdfid,attrNum,0)

value =

My attr value

% Delete the entry
cdflib.deleteAttrEntry(cdfid,attrNum,0);

% Now try to view the value of the entry
% Should return NO_SUCH_ENTRY failure.
value = cdflib.getAttrEntry(cdfid,attrNum,0)  % Should fail

% Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFdeleteAttrzEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.deleteAttr`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.deleteAttrgEntry

Delete entry in global attribute

Syntax

```
cdflib.deleteAttrgEntry(cdfId,attrNum,entryNum)
```

Description

`cdflib.deleteAttrgEntry(cdfId,attrNum,entryNum)` deletes an entry from a global attribute in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrNum`

Numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have global scope.

`entryNum`

Numeric value that specifies the entry in the attribute. Entry numbers are zero-based.

Examples

Create a CDF and create a global attribute in the file. Write a value to an entry for the attribute and then delete the entry. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');
```

```
% Initially the file contains no attributes, global or variable.
info = cdflib.inquire(cdfid)

info =

     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: -1
      numVars: 0
    numvAttrs: 0
    numgAttrs: 0

% Create an attribute with global scope in the file.
attrNum = cdflib.createAttr(cdfid,'my_global_attr','global_scope');

% Write a value to an entry for the attribute
cdflib.putAttrgEntry(cdfid,attrNum,0,'CDF_CHAR','My global attr');

% Get the value of the global attribute entry
value = cdflib.getAttrgEntry(cdfid,attrNum,0)

value =

My global attr

% Delete the entry
cdflib.deleteAttrgEntry(cdfid,attrNum,0);

% Now try to view the value of the entry
% Should return NO_SUCH_ENTRY failure.
value = cdflib.getAttrgEntry(cdfid,attrNum,0)  % Should fail

% Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFdeleteAttrgEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.deleteAttr` | `cdflib.deleteAttrEntry`

Topics

“Import CDF Files Using Low-Level Functions”
“Export to CDF Files”

cdflib.deleteVar

Delete variable

Syntax

```
cdflib.deleteVar(cdfId,varNum)
```

Description

`cdflib.deleteVar(cdfId,varNum)` deletes a variable from a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that specifies the variable. Variable numbers are zero-based.

Examples

Create a CDF, create a variable in the CDF, and then delete it.

```
cdfid = cdflib.create('mycdf.cdf');

% Initially the file contains no variables.
info = cdflib.inquire(cdfid)

info =

     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: -1
      numVars: 0
    numvAttrs: 0
    numgAttrs: 0

% Create a variable in the CDF.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);
```

```
% Retrieve info about the variable in the CDF.

info = cdflib.inquireVar(cdfid, 0)

info =

           name: 'Time'
       datatype: 'cdf_int1'
    numElements: 1
           dims: []
    recVariance: 1
    dimVariance: []

% Delete the variable from the CDF

cdflib.deleteVar(cdfid,0);

% Check to see if the variable was deleted from the CDF.
info = cdflib.inquire(cdfid)

info =

     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: -1
      numVars: 0
    numvAttrs: 0
    numgAttrs: 0

% Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFdeletezVar`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.createVar`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.deleteVarRecords

Delete range of records from variable

Syntax

```
cdflib.deleteVarRecords(cdfId,varNum,startRec,endRec)
```

Description

`cdflib.deleteVarRecords(cdfId,varNum,startRec,endRec)` deletes a range of records from a variable in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value that identifies the variable. Variable numbers are zero-based.

`startRec`

Numeric value that specifies the record at which to start deleting records. Record numbers are zero-based.

`endRec`

Numeric value that specifies the record at which to stop deleting records. Record numbers are zero-based.

Examples

Make a writable copy of the example CDF, get the number of a variable in the CDF, and delete specific records in the variable. To run this example, you must be in a writable folder.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.cdf');
copyfile(srcFile,'myfile.cdf');
fileattrib('myfile.cdf','+w');
cdfid = cdflib.open('myfile.cdf');
varnum = cdflib.getVarNum(cdfid,'Temperature');
cdflib.deleteVarRecords(cdfid,varnum,1,2);
cdflib.close(cdfid);
```

References

This function corresponds to the CDF library C API routine `CDFdeletezVarRecords`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarNumRecsWritten` | `cdflib.putVarRecordData`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.epoch16Breakdown

Convert `CDF_EPOCH16` value to time value

Syntax

```
timeVec = cdflib.epoch16Breakdown(epoch16Time)
```

Description

`timeVec = cdflib.epoch16Breakdown(epoch16Time)` convert a `CDF_EPOCH16` value into a time vector. `timeVec` will have 10-by-*n* elements, where *n* is the number of CDF_EPOCH16 values.

The following table describes the time value components.

timeVec Element	Description	Valid Values
`timeVec(1,:)`	Year AD	e.g. 1994
`timeVec(2,:)`	Month	1–12
`timeVec(3,:)`	Day	1–31
`timeVec(4,:)`	Hour	0–23
`timeVec(5,:)`	Minute	0–59
`timeVec(6,:)`	Second	0–59
`timeVec(7,:)`	Millisecond	0–999
`timeVec(8,:)`	Microsecond	0–999
`timeVec(9,:)`	Nanosecond	0–999
`timeVec(10,:)`	Picosecond	0–999

Examples

Convert `CDF_EPOCH16` value into time value.

```
timeval = [1999; 12; 31; 23; 59; 59; 50; 100; 500; 999];
epoch16 = cdflib.computeEpoch16(timeval);

timevec = cdflib.epoch16Breakdown(epoch16)

timevec =

        1999
          12
          31
          23
          59
          59
          50
         100
         500
         999
```

References

This function corresponds to the CDF library C API routine `EPOCH16breakdown`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.computeEpoch16`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.epochBreakdown

Convert CDF_EPOCH value into time value

Syntax

```
timeVec = cdflib.epochBreakdown(epochTime)
```

Description

`timeVec = cdflib.epochBreakdown(epochTime)` decomposes the CDF_EPOCH value, epochTime value into individual time components. timeVec will have 7-by-*n* elements, where *n* is the number of CDF_EPOCH values in epochTime.

The return value timeVec has the following elements:

timeVec Element	Description	Valid Values
timeVec(1,:)	Year AD	e.g. 1994
timeVec(2,:)	Month	1–12
timeVec(3,:)	Day	1–31
timeVec(4,:)	Hour	0–23
timeVec(5,:)	Minute	0–59
timeVec(6,:)	Second	0–59
timeVec(7,:)	Millisecond	0–999

Examples

Convert a CDP_EPOCH value into a time vector.

```
% First convert a time vector into a CDF_EPOCH value
timeval = [1999 12 31 23 59 59 0];
epoch = cdflib.computeEpoch(timeval);
```

```
% Convert the CDF_EPOCH value into a time vector
timevec = cdflib.epochBreakdown(epoch)

timevec =

        1999
          12
          31
          23
          59
          59
           0
```

References

This function corresponds to the CDF library C API routine `EPOCHbreakdown`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.computeEpoch` | `cdflib.epoch16Breakdown` | `cdflib.epochBreakdown`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getAttrEntry

Value of entry in attribute with variable scope

Syntax

```
value = cdflib.getAttrEntry(cdfId,attrNum,entryNum)
```

Description

`value = cdflib.getAttrEntry(cdfId,attrNum,entryNum)` returns the value of an attribute entry in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrNum`

Numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have variable scope.

`entryNum`

Numeric value that specifies the entry in the attribute. Entry numbers are zero-based.

Output Arguments

`Value`

Value of the entry.

Examples

Open the example CDF and get the value of an entry associated with an attribute with variable scope in the file.

```
cdfid = cdflib.open('example.cdf');

% The fourth attribute is of variable scope.
attrscope = cdflib.getAttrScope(cdfid,3)

attrscope =

VARIABLE_SCOPE

% Get information about the first entry for this attribute
[dtype numel] = cdflib.inquireAttrEntry(cdfid,3,0)

dtype =

cdf_char


numel =

    10

% Get the value of the entry for this attribute.
% Note that it is a character vector of length 10
value = cdflib.getAttrEntry(cdfid,3,0)

value =

Time value

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetAttrzEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getAttrgEntry` | `cdflib.putAttrEntry` | `cdflib.putAttrgEntry`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getAttrgEntry

Value of entry in global attribute

Syntax

```
value = cdflib.getAttrgEntry(cdfId,attrNum,entryNum)
```

Description

`value = cdflib.getAttrgEntry(cdfId,attrNum,entryNum)` returns the value of a global attribute entry in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrNum`

Numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have global scope.

`entryNum`

Numeric value that specifies the entry in the attribute. Entry numbers are zero-based.

Output Arguments

`Value`

Value of the entry.

Examples

Open the example CDF, and then get the value of an entry associated with a global attribute in the file:

```
cdfid = cdflib.open('example.cdf');

% Any of the first three attributes have global scope.
attrscope = cdflib.getAttrScope(cdfid,0)

attrscope =

GLOBAL_SCOPE

% Get information about the first entry for global attribute
[dtype numel] = cdflib.inquireAttrgEntry(cdfid,0,0)

dtype =

cdf_char


numel =

    23

% Get the value of the first entry for this global attribute.
value = cdflib.getAttrgEntry(cdfid,0,0)

value =

This is a sample entry.

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetAttrgEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getAttrEntry` | `cdflib.putAttrEntry` | `cdflib.putAttrgEntry`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getAttrMaxEntry

Number of last entry for variable attribute

Syntax

```
maxEntry = cdflib.getAttrMaxEntry(cdfId,attrNum)
```

Description

`maxEntry = cdflib.getAttrMaxEntry(cdfId,attrNum)` returns the number of the last entry for an attribute in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file.

`attrNum` is a numeric value that specifies the attribute. Attribute numbers are zero-based. The attribute must have variable scope.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrNum`

Numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have variable scope.

Output Arguments

`maxEntry`

Entry number of the last entry in the attribute. Entry numbers are zero-based.

Examples

Open the example CDF and get the number of the last entry associated with an attribute with variable scope in the file:

```
cdfid = cdflib.open('example.cdf');

% The fourth attribute is of variable scope.
attrscope = cdflib.getAttrScope(cdfid,3)

attrscope =

VARIABLE_SCOPE


% Get the number of the last entry for this attribute.
entrynum = cdflib.getAttrMaxEntry(cdfid,3)

entrynum =

    3


% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetAttrMaxzEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getAttrMaxgEntry`

Topics

“Import CDF Files Using Low-Level Functions”
“Export to CDF Files”

cdflib.getAttrMaxgEntry

Number of last entry for global attribute

Syntax

```
maxEntry = cdflib.getAttrMaxgEntry(cdfId,attrNum)
```

Description

`maxEntry = cdflib.getAttrMaxgEntry(cdfId,attrNum)` returns the last entry number of a global attribute in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrNum`

Numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have global scope.

Output Arguments

`maxEntry`

Entry number of the last entry in the attribute. Entry numbers are zero-based.

Examples

Open the example CDF and get the number of the last entry associated with a global attribute in the file:

```
cdfid = cdflib.open('example.cdf');

% Any of the first three attribute are of global scope.
attrscope = cdflib.getAttrScope(cdfid,0)

attrscope =

GLOBAL_SCOPE


% Get the number of the last entry for this attribute.
entrynum = cdflib.getAttrMaxgEntry(cdfid,0)

entrynum =

     4

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetAttrMaxgEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getAttrMaxEntry`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getAttrName

Name of attribute, given attribute number

Syntax

```
name = cdflib.getAttrName(cdfId,attrNum)
```

Description

`name = cdflib.getAttrName(cdfId,attrNum)` returns the name of an attribute in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrNum`

Numeric value that identifies the attribute. Attribute numbers are zero-based.

Output Arguments

`name`

Character vector specifying the name of the attribute.

Examples

Open the example CDF and get name of an attribute.

```
cdfid = cdflib.open('example.cdf');

% Get name of the first attribute in the file.
attrName = cdflib.getAttrName(cdfId,0)

attrName =

SampleAttribute

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetAttrName`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.createAttr`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getAttrNum

Attribute number, given attribute name

Syntax

```
attrNum = cdflib.getAttrNum(cdfId,attrName)
```

Description

`attrNum = cdflib.getAttrNum(cdfId,attrName)` returns the number of an attribute in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrName`

Character vector or string scalar specifying the name of an attribute.

Output Arguments

`attrNum`

Numeric value that identifies the attribute. Attribute numbers are zero-based.

Examples

Open the example CDF and get the attribute number associated with the `SampleAttribute` attribute.

```
cdfid = cdflib.open('example.cdf');

attrNum = cdflib.getAttrNum(cdfid,'SampleAttribute')

attrNum =

    0

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetAttrNum`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.createAttr` | `cdflib.getAttrName`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getAttrScope

Scope of attribute

Syntax

```
scope = cdflib.getAttrScope(cdfId,attrNum)
```

Description

`scope = cdflib.getAttrScope(cdfId,attrNum)` returns the scope of an attribute in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrNum`

Numeric value that specifies the attribute. Attribute numbers are zero-based.

Output Arguments

`scope`

One of the following character vectors, or its numeric equivalent.

Value	Description
`'GLOBAL_SCOPE'`	Attribute applies to the CDF as a whole.
`'VARIABLE_SCOPE'`	Attribute applies only to the variable.

To get the numeric equivalent of these constants, use the `cdflib.getConstantValue` function.

Examples

Open example CDF and get the scope of the first attribute in the file:

```
cdfid = cdflib.open('example.cdf');

attrScope = cdflib.getAttrScope(cdfid,0)

attrScope =

GLOBAL_SCOPE

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetAttrScope`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.createAttr` | `cdflib.getAttrName` | `cdflib.getConstantValue`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getCacheSize

Number of cache buffers used

Syntax

```
numBuffers = cdflib.getCacheSize(cdfId)
```

Description

`numBuffers = cdflib.getCacheSize(cdfId)` returns the number of cache buffers used for the Common Data Format (CDF) file identified by `cdfId`. For a discussion of cache schemes, see the CDF User's Guide.

Examples

Open the example CDF file and get the cache size:

```
cdfid = cdflib.open('example.cdf');

numBuf = cdflib.getCacheSize(cdfid)

numBuf =

   300

% Clean up
cdflib.close(cdfid)
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setCacheSize`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getChecksum

Checksum mode

Syntax

```
mode = cdflib.getChecksum(cdfId)
```

Description

`mode = cdflib.getChecksum(cdfId)` returns the checksum mode of the Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

Output Arguments

mode

One of the following character vectors or its numeric equivalent.

`'MD5_CHECKSUM'`	File uses MD5 checksum.
`'NO_CHECKSUM'`	File does not use a checksum.

To get the numeric equivalent of the constants specified by these character vectors, use `cdflib.getConstantValue`.

Examples

Open the example CDF file, and then get the checksum mode of the file:

```
cdfid = cdflib.open('example.cdf');

checksummode = cdflib.getChecksum(cdfid)


checksummode =

NO_CHECKSUM

% Clean up
cdflib.close(cdfid);
clear cdfid;
```

References

This function corresponds to the CDF library C API routine `CDFgetChecksum`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getConstantValue` | `cdflib.setChecksum`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getCompression

Compression settings

Syntax

```
[ctype,cparms,cpercentage] = cdflib.getCompression(cdfId)
```

Description

`[ctype,cparms,cpercentage] = cdflib.getCompression(cdfId)` returns information about the compression settings of a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

Output Arguments

ctype

Character vector specifying compression type, such as `'HUFF_COMPRESSION'`. If the CDF does not use compression, the function returns `'NO_COMPRESSION'`. For a list of supported compression types, see `cdflib.setCompression`.

cparms

The value of the parameter associated with the type of compression. For example, for the `'RLE_COMPRESSION'` compression type, the parameter specifies the style of run-length encoding. For a list of parameters supported by each compression type, see `cdflib.setCompression`.

cpercentage

The rate of compression, expressed as a percentage.

Examples

Open the example CDF file and check the compression settings in the file.

```
cdfId = cdflib.open('example.cdf');

[ctype, cparms, cpercentage] = cdflib.getCompression(cdfId)

ctype =

GZIP_COMPRESSION

cparms =

     7

cper =

    26

% Clean up
cdflib.close(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetCompression`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarCompression` | `cdflib.setCompression` | `cdflib.setVarCompression`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getCompressionCacheSize

Number of compression cache buffers

Syntax

```
numBuffers = cdflib.getCompressionCacheSize(cdfId)
```

Description

`numBuffers = cdflib.getCompressionCacheSize(cdfId)` returns the number of cache buffers used for the compression scratch Common Data Format (CDF) file. `cdfId` identifies the CDF file. For a discussion of cache schemes, see the CDF User's Guide.

Examples

Open the example CDF file and check the compression cache size of the file:

```
cdfId = cdflib.open('example.cdf');

numBuf = cdflib.getCompressionCacheSize(cdfId)

numBuf =

    80

% Clean up
cdflib.close(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetCompressionCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setCompressionCacheSize`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getConstantNames

Names of Common Data Format (CDF) library constants

Syntax

```
names = cdflib.getConstantNames()
```

Description

`names = cdflib.getConstantNames()` returns a cell array of character vectors, where each character vector is the name of a constant known to the Common Data Format (CDF) library.

Examples

Get a list of the names of CDF library constants.

```
names  = cdflib.getConstantNames()

names =

    'AHUFF_COMPRESSION'
    'ALPHAMVSD_ENCODING'
    'ALPHAMVSG_ENCODING'
    'ALPHAMVSI_ENCODING'
    'ALPHAOSF1_ENCODING'
    'CDF_BYTE'
    'CDF_CHAR'
     .
     .
     .
```

References

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getConstantValue`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getConstantValue

Numeric value corresponding to Common Data Format (CDF) library constant

Syntax

`value = cdflib.getConstantValue(constantName)`

Description

`value = cdflib.getConstantValue(constantName)` returns the numeric value of the CDF library constant specified in *`constantName`*. Specify *`constantName`* as a character vector or string scalar. To see a list of constant names, use `cdflib.getConstantNames`.

Examples

View the list of CDF library constants and get the numeric value corresponding to one of the constants.

```
% Retrieve a list of library constants
names = cdflib.getConstantNames();

value  = cdflib.getConstantValue(names{1})

value =

   3
```

References

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getConstantNames`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getCopyright

Copyright notice in Common Data Format (CDF) file

Syntax

```
copyright = cdflib.getCopyright(cdfId)
```

Description

`copyright = cdflib.getCopyright(cdfId)` returns the copyright notice in the CDF file identified by `cdfId`.

Examples

Create a CDF file, and then get the copyright notice in the file. To run this example, you must be in a writable folder.

```
cdfId = cdflib.create('your_file.cdf');

copyright  = cdflib.getCopyright(cdfId)

copyright =

Common Data Format (CDF)
(C) Copyright 1990-2009 NASA/GSFC
Space Physics Data Facility
NASA/Goddard Space Flight Center
Greenbelt, Maryland 20771 USA
(Internet -- CDFSUPPORT@LISTSERV.GSFC.NASA.GOV)

% Clean up.
cdflib.delete(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetCopyright`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getLibraryCopyright`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getFileBackward

Return current backward compatibility mode setting

Syntax

```
mode = cdflib.getFileBackward()
```

Description

`mode = cdflib.getFileBackward()` returns the backward compatibility mode.

Output Arguments

mode

One of the following character vectors:

`'BACKWARDFILEon'`	Backward compatibility mode is on.
`'BACKWARDFILEoff'`	Backward compatibility mode is off.

For more information about backward compatibility mode, see `cdflib.setFileBackward`.

Examples

```
mode = cdflib.getFileBackward
```

```
mode =

BACKWARDFILEoff
```

References

This function corresponds to the CDF library C API routine `CDFgetFileBackward`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getConstantValue` | `cdflib.setFileBackward`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getFormat

Format of Common Data Format (CDF) file

Syntax

format = cdflib.setFormat(cdfId)

Description

format = cdflib.setFormat(cdfId) returns the format of the CDF file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to cdflib.create or cdflib.open.

Output Arguments

format

One of the following character vectors, or its numeric equivalent.

'SINGLE_FILE'	The CDF is stored in a single file.
'MULTI_FILE'	The CDF is made up of multiple files.

To get the numeric equivalent of the constants specified by these character vectors, use cdflib.getConstantValue.

Examples

Open the example CDF file and determine its file format:

```
cdfId = cdflib.open('example.cdf');

format = cdflib.getFormat(cdfId)

format =

      'SINGLE_FILE'

% Clean up.
cdflib.close(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetFormat`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getConstantValue` | `cdflib.setFormat`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getLibraryCopyright

Copyright notice of Common Data Format (CDF) library

Syntax

```
copyright = cdflib.getLibraryCopyright()
```

Description

`copyright = cdflib.getLibraryCopyright()` returns a character vector containing the copyright notice of the CDF library.

Examples

Get the copyright of the CDF library.

```
copyright  = cdflib.getLibraryCopyright()
```

```
copyright =

Common Data Format (CDF)
(C) Copyright 1990-2008 NASA/GSFC
Space Physics Data Facility
NASA/Goddard Space Flight Center
Greenbelt, Maryland 20771 USA
(Internet -- CDFSUPPORT@LISTSERV.GSFC.NASA.GOV)
```

References

This function corresponds to the CDF library C API routine `CDFgetLibraryCopyright`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getCopyright`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getLibraryVersion

Library version and release information

Syntax

```
[version,release,increment] = cdflib.getLibraryVersion()
```

Description

`[version,release,increment] = cdflib.getLibraryVersion()` returns information about the Common Data Format (CDF) library.

Output Arguments

`version`

Numeric value indicating the version number of the CDF library.

`release`

Numeric value indicating the release number of the CDF library.

`increment`

Numeric value indicating the increment number of the CDF library.

Examples

Get the version information of the CDF library:

```
[version, release, increment] = cdflib.getLibraryVersion()
```

```
version =
```

```
     3

release =

     6

increment =

     1
```

References

This function corresponds to the CDF library C API routine `CDFgetLibraryVersion`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVersion`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getMajority

Majority of variables

Syntax

majority = cdflib.getMajority(cdfId)

Description

majority = cdflib.getMajority(cdfId) returns the majority of variables in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to cdflib.create or cdflib.open.

Output Arguments

majority

One of the following character vectors or its numeric equivalent.

'ROW_MAJOR'	C-like array ordering for variable storage. The first dimension in each variable array varies the slowest. This is the default.
'COLUMN_MAJOR'	Fortran-like array ordering for variable storage. The first dimension in each variable array varies the fastest.

To get the numeric equivalent of the constants specified by these character vectors, use cdflib.getConstantValue.

Examples

Open the example CDF file, and then determine the majority of variables in the file:

```
cdfId = cdflib.open('example.cdf');

majority = cdflib.getMajority(cdfId)

majority =

ROW_MAJOR

% Clean up
cdflib.close(cdfId)

clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetMajority`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getConstantValue` | `cdflib.setMajority`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getName

Name of Common Data Format (CDF) file

Syntax

```
name = cdflib.getName(cdfId)
```

Description

`name = cdflib.getName(cdfId)` returns the name of the CDF file identified by `cdfId`.

Examples

Open the example CDF file and get the name of the file. The path name returned for your installation will be different.

```
cdfId = cdflib.open('example.cdf');

name = cdflib.getName(cdfId)

name =

yourinstallation\matlab\toolbox\matlab\demos\example

% Clean up
cdflib.close(cdfId)

clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetName`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.create` | `cdflib.open`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getNumAttrEntries

Number of entries for attribute with variable scope

Syntax

```
nentries = cdflib.getNumAttrEntries(cdfId,attrNum)
```

Description

`nentries = cdflib.getNumAttrEntries(cdfId,attrNum)` returns the number of entries for the specified attribute in the Common Data Format (CDF) file.

`cdfId` identifies the CDF file.

`attrNum` is a numeric value that specifies the attribute. Attribute numbers are zero-based. The attribute must have variable scope.

Examples

Open the example CDF, find an attribute with variable scope, and determine how many entries are associated with the attribute:

```
cdfid = cdflib.open('example.cdf');

% Get the number of an attribute
% with variable scope
attrNum = cdflib.getAttrNum(cdfid,'Description');

% Check that scope of attribute is variable
attrScope = cdflib.getAttrScope(cdfid,attrNum)

VARIABLE_SCOPE

% Determine the number of entries for the attribute
attrEntries = cdflib.getNumAttrEntries(cdfid,attrNum)
```

```
attrEntries =

    4

% Clean up
cdflib.close(cdfid);
```

References

This function corresponds to the CDF library C API routine `CDFgetNumAttrzEntries`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getAttrScope`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getNumAttrgEntries

Number of entries for attribute with global scope

Syntax

```
nentries = cdflib.getNumAttrgEntries(cdfId,attrNum)
```

Description

`nentries = cdflib.getNumAttrgEntries(cdfId,attrNum)` returns the number of entries written for the specified global attribute in the Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `attrNum` is a numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have global scope.

Examples

Open the example CDF and find out how many entries are associated with a global attribute in the file.

```
cdfid = cdflib.open('example.cdf');

% The first attribute is a global attribute.
attrgEntries = cdflib.getNumAttrgEntries(cdfid,0)

attrgEntries =

    3

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetNumAttrgEntries`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getNumAttrEntries`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getNumAttributes

Number of attributes with variable scope

Syntax

```
numAtts = cdflib.getNumAttributes(cdfId)
```

Description

`numAtts = cdflib.getNumAttributes(cdfId)` returns the total number of attributes with variable scope in a Common Data Format (CDF) file. `cdfId` identifies the CDF file.

Examples

Open the example CDF and find out how many attributes in the file have variable scope:

```
cdfid = cdflib.open('example.cdf');

% Determine the number of attributes with variable scope
numAttrs = cdflib.getNumAttributes(cdfid)

numAttrs =

    1

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetNumvAttributes`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getNumgAttributes`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getNumgAttributes

Number of attributes with global scope

Syntax

```
ngatts = cdflib.getNumgAttributes(cdfId)
```

Description

`ngatts = cdflib.getNumgAttributes(cdfId)` returns the total number of global attributes in a Common Data Format (CDF) file. `cdfId` identifies the CDF file.

Examples

Open the example CDF and find out how many global attributes are in the file:

```
cdfid = cdflib.open('example.cdf');

% Determine the number of global attributes in the file.
numgAttrs = cdflib.getNumgAttributes(cdfid)

numgAttrs =

    3

% Clean up
cdflib.close(cdfid);
```

References

This function corresponds to the CDF library C API routine `CDFgetNumgAttributes`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getNumAttributes`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getReadOnlyMode

Read-only mode

Syntax

```
mode = cdflib.getReadOnlyMode(cdfId)
```

Description

`mode = cdflib.getReadOnlyMode(cdfId)` returns the read-only mode of a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

Output Arguments

mode

One of the following character vectors or its numeric equivalent.

`'READONLYon'`	CDF is in read-only mode
`'READONLYoff'`	CDF can be modified.

To get the numeric equivalent of the constants specified by these character vectors, use `cdflib.getConstantValue`.

Examples

Open the example CDF file and determine its current read-only status:

```
cdfId = cdflib.open('example.cdf');

mode = cdflib.getReadOnlyMode(cdfId)

mode =

READONLYoff

% Clean up.
cdflib.close(cdfId);
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetReadOnlyMode`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getConstantValue` | `cdflib.setReadOnlyMode`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getStageCacheSize

Number of cache buffers for staging

Syntax

```
numBuffers = cdflib.getStageCacheSize(cdfId)
```

Description

`numBuffers = cdflib.getStageCacheSize(cdfId)` returns the number of cache buffers used for the staging scratch file of the Common Data Format (CDF) file. For more information about cache buffers, see the CDF User's Guide.

`cdfId` identifies the CDF file.

Examples

Open the example CDF file and determine the number of cache buffers used for staging:

```
cdfId = cdflib.open('example.cdf');

numBuf = cdflib.getStageCacheSize(cdfId)

numBuf =

      125

% Clean up
cdflib.close(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetStageCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setStageCacheSize`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getValidate

Library validation mode

Syntax

```
mode = cdflib.getValidate()
```

Description

`mode = cdflib.getValidate()` returns the validation mode of the Common Data Format (CDF) library.

Output Arguments

`mode`

One of the following character vectors or its numeric equivalent.

`'VALIDATEFILEon'`	Validation mode is on. For information about validation mode, see `cdflib.setValidate`.
`'VALIDATEFILEoff'`	Validation mode is off.

To get the numeric equivalent of the constants specified by these character vectors, use `cdflib.getConstantValue`.

Examples

Determine the current validation mode of the CDF library.

```
mode = cdflib.getValidate()
```

```
mode =
```

```
'VALIDATEFILEon'
```

References

This function corresponds to the CDF library C API routine `CDFgetValidate`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getConstantValue` | `cdflib.setValidate`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getVarAllocRecords

Number of records allocated for variable

Syntax

```
numrecs = cdflib.getVarAllocRecords(cdfId, varNum)
```

Description

`numrecs = cdflib.getVarAllocRecords(cdfId, varNum)` returns the number of records allocated for a variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers are zero-based.

Examples

Open example CDF and get the number of records allocated for a variable:

```
cdfid = cdflib.open('example.cdf');

% Determine the number of records allocated for the
% first variable in the file.
numrecs = cdflib.getVarAllocRecords(cdfid,0)

numrecs =

    64


%  Clean up
cdflib.close(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarAllocRecords`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setVarAllocBlockRecords`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getVarBlockingFactor

Blocking factor for variable

Syntax

```
blockingFactor = cdflib.getVarBlockingFactor(cdfId,varNum)
```

Description

`blockingFactor = cdflib.getVarBlockingFactor(cdfId,varNum)` returns the blocking factor for a variable in a Common Data Format (CDF) file. A variable's blocking factor specifies the minimum number of records the library allocates when you write to an unallocated record.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers are zero-based.

Examples

Open the example CDF and determine the blocking factor of a variable.

```
cdfid = cdflib.open('example.cdf');

cdflib.getVarBlockingFactor(cdfid,0)

ans =

     0

%  Clean up
cdflib.close(cdfid)
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarBlockingFactor`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setVarBlockingFactor`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getVarCacheSize

Number of multifile cache buffers

Syntax

```
numBuffers = cdflib.getVarCacheSize(cdfId,varNum)
```

Description

`numBuffers = cdflib.getVarCacheSize(cdfId,varNum)` returns the number of cache buffers used for a variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable identifiers are zero-based.

This function applies only to multifile format CDFs. For more information about caching, see the CDF User's Guide.

Examples

Create a multifile CDF and retrieve the number of buffers being used for a variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf')

% Set the format of the file to be multi-file
cdflib.setFormat(cdfid,'MULTI_FILE');

% Create a variable in the file
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Note how the library creates a separate file for the variable
ls your_file.*

your_file.cdf  your_file.z0
```

```
% Determine the number of cache buffers used with the variable
numBuf = cdflib.getVarCacheSize(cdfid,varNum)

numBuf =

     1

%  Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setVarCacheSize`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getVarCompression

Information about compression used by variable

Syntax

```
[ctype,cparams,percent] = cdflib.getVarCompression(cdfId, varNum)
```

Description

`[ctype,cparams,percent] = cdflib.getVarCompression(cdfId, varNum)` returns information about the compression used for a variable in a Common Data Format (CDF) File.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

Output Arguments

`ctype`

Character vector identifying the type of compression. For a list of compression types, see `cdflib.setCompression`.

`cparams`

Any additional parameter required by the compression type.

`percent`

Numeric value indicating the level of compression, expressed as a percentage.

Examples

Open the example CDF file and check the compression settings of any variable.

```
cdfid = cdflib.open('example.cdf');

% Check the compression setting of any variable in the file
% The example checks the first variable (variable numbers are zero-based).
[ctype params percent] = cdflib.getVarCompression(cdfid,0)

ctype =

NO_COMPRESSION


params =

     []


percent =

   100

% Clean up
cdflib.close(cdfid);
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarCompression`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setCompression` | `cdflib.setVarCompression`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getVarData

Single value from record in variable

Syntax

```
datum = cdflib.getVarData(cdfId,varNum,recNum,indices)
datum = cdflib.getVarData(cdfId,varNum,recNum)
```

Description

`datum = cdflib.getVarData(cdfId,varNum,recNum,indices)` returns a single value from a variable in a Common Data Format (CDF) file.

`datum = cdflib.getVarData(cdfId,varNum,recNum)` returns a single value from a variable with no dimensions in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value identifying the variable containing the datum. Variable numbers are zero-based.

`recNum`

Numeric value identifying the location of the datum in the variable. In CDF terminology, this is called the record number. Record numbers are zero-based.

`indices`

Array of dimension indices within the record. Dimension indices are zero-based. If the variable has no dimensions, you can omit this parameter.

Output Arguments

datum

Value of the specified record.

Examples

Open the example CDF file and retrieve data associated with a variable:

```
cdfid = cdflib.open('example.cdf');

% Determine how many variables are in the file.
info = cdflib.inquire(cdfid);

info.numVars

ans =

  5

% Determine if the first variable has dimensions.
varinfo = cdflib.inquireVar(cdfid,0);
vardims = varinfo.dims
vardims =

  []

% Get data from variable, without specifying dimensions.
datum = cdflib.getVarData(cdfid, varnum, recnum)

datum =

  6.3146e+013

% Get dimensions of another variable in file.
varinfo = cdflib.inquireVar(cdfid,3);
vardims = varinfo.dims
vardims =

  [4 2 2]
```

```
% Retrieve the first datum in the record. Indices are zero-based.
datum = cdflib.getVarData(cdfId,3,0,[0 0 0])

info =

    30

% Clean up.
cdflib.close(cdfid);
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarData`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarRecordData` | `cdflib.hyperGetVarData` | `cdflib.putVarData`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getVarMaxAllocRecNum

Maximum allocated record number for variable

Syntax

```
maxrec = cdflib.getVarMaxAllocRecNum(cdfId,varNum)
```

Description

`maxrec = cdflib.getVarMaxAllocRecNum(cdfId,varNum)` returns the record number of the maximum allocated record for a variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers and record numbers are zero-based.

Examples

Open example CDF and get the maximum allocated record number for a variable:

```
cdfid = cdflib.open('example.cdf');

% Determine maximum record number for variable in file.
maxRecNum = cdflib.getVarMaxAllocRecNum(cdfid,0)

maxRecNum =

    63

%  Clean up
cdflib.close(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarMaxAllocRecNum`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarMaxWrittenRecNum`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getVarMaxWrittenRecNum

Maximum written record number for variable

Syntax

```
maxrec = cdflib.getVarMaxwrittenRecNum(cdfId,varNum)
```

Description

`maxrec = cdflib.getVarMaxwrittenRecNum(cdfId,varNum)` returns the record number of the maximum record written for a variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers and record numbers are zero-based.

Examples

Open the example CDF, and then determine the maximum number of records written to a variable:

```
cdfid = cdflib.open('example.cdf');

% Determine the number records written to variable.
numRecs = cdflib.getVarNumRecsWritten(cdfid,0)

numRecs =

    24


% Determine the maximum record number of the records written
maxRecNum = cdflib.getVarMaxWrittenRecNum(cdfid,0)

maxRecNum =
```

```
    23

%  Clean up
cdflib.close(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarMaxWrittenRecNum`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarMaxAllocRecNum` | `cdflib.getVarNumRecsWritten`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getVarsMaxWrittenRecNum

Maximum written record number for CDF file

Syntax

```
maxrec = cdflib.getVarsMaxwrittenRecNum(cdfId)
```

Description

`maxrec = cdflib.getVarsMaxwrittenRecNum(cdfId)` returns the maximum record number written for all variables in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. Record numbers are zero-based.

Examples

Open the example CDF, and then determine the maximum number of records written to the file:

```
cdfid = cdflib.open('example.cdf');

% Determine the maximum record number of the records written
maxRecNum = cdflib.getVarsMaxWrittenRecNum(cdfid)

maxRecNum =

    23

%  Clean up
cdflib.close(cdfid)
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarsMaxWrittenRecNum`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarMaxWrittenRecNum`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getVarName

Variable name, given variable number

Syntax

```
name = cdflib.getVarName(cdfId,varNum)
```

Description

`name = cdflib.getVarName(cdfId,varNum)` returns the name of the variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers are zero-based. `name` is a character vector specifying the name.

Examples

Open the example CDF, and then get the name of a variable in the file:

```
cdfid = cdflib.open('example.cdf');

name = cdflib.getVarName(cdfid,1)

name =

Longitude

%  Clean up
cdflib.close(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarName`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.inquireVar`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getVarNum

Variable number, given variable name

Syntax

`varNum = cdflib.getVarNum(cdfId,varname)`

Description

`varNum = cdflib.getVarNum(cdfId,varname)` returns the identifier (variable number) for a variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. *`varname`* is a character vector or string scalar that identifies the variable. Variable names are case-sensitive.

Examples

Open example CDF, and then get the number of a variable named `Longitude`:

```
cdfid = cdflib.open('example.cdf');

varNum = cdflib.getVarNum(cdfid,'Longitude')

varNum =

     1

%  Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarNum`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarName`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getVarNumRecsWritten

Number of records written to variable

Syntax

```
numrecs = cdflib.getVarNumRecsWritten(cdfId,varNum)
```

Description

`numrecs = cdflib.getVarNumRecsWritten(cdfId,varNum)` returns the total number of records written to a variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers are zero-based.

Examples

Open the example CDF, and then determine the number of records written to a variable:

```
cdfid = cdflib.open('example.cdf');

% Determine the number of records written to the variable.
numRecs = cdflib.getVarNumRecsWritten(cdfid,0)

numRecs =

    24

%  Clean up
cdflib.close(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarNumRecsWritten`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarMaxWrittenRecNum`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getVarPadValue

Pad value for variable

Syntax

```
padvalue = cdflib.getVarPadValue(cdfId,varNum)
```

Description

`padvalue = cdflib.getVarPadValue(cdfId,varNum)` returns the pad value used with a variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers are zero-based.

Examples

Open the example CDF, and then determine the pad value for a variable:

```
cdfid = cdflib.open('example.cdf');

% Check pad value of variable in the file.
padval = cdflib.getVarPadValue(cdfid,0)

padval =

     0

%  Clean up.
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarPadValue`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setVarPadValue`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getVarRecordData

Entire record for variable

Syntax

```
data = cdflib.getVarRecordData(cdfId,varNum,recNum)
```

Description

`data = cdflib.getVarRecordData(cdfId,varNum,recNum)` returns the data in a record associated with a variable in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value that identifies the variable in the CDF file. Variable numbers are zero-based.

`recNum`

Numeric value that identifies the record in the variable. Record numbers are zero-based.

Output Arguments

`data`

Data in the record.

Examples

Open the example CDF, and then get the data associated with a record in a variable:

```
cdfid = cdflib.open('example.cdf');

% Get data in first record in first variable in file.
recData = cdflib.getVarRecordData(cdfid,0,0)

recData =

    6.3146e+013

%  Clean up
cdflib.close(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarRecordData`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarData` | `cdflib.hyperGetVarData` | `cdflib.putVarRecordData`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getVarReservePercent

Compression reserve percentage for variable

Syntax

```
percent = cdflib.getVarReservePercent(cdfId,varNum)
```

Description

`percent = cdflib.getVarReservePercent(cdfId,varNum)` returns the compression reserve percentage for a variable in a Common Data Format (CDF) file. This operation only applies to compressed variables.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers are zero-based.

Examples

Open the example CDF file, get the number of a compressed variable, and then determine the reserve percent for the variable.

```
cdfid = cdflib.open('example.cdf');
varnum = cdflib.getVarNum(cdfid,'Temperature');
percent = cdflib.getVarReservePercent(cdfid,varnum);
cdflib.close(cdfid);
```

More About

reserve percentage

Specifies how much extra space to allocate for a compressed variable. This extra space allows the variable to expand when you write additional records to the variable. If you do not specify this room for growth, the library has to move the variable to the end of the file

when the size expands and the space at the original location of the variable becomes wasted space.

By default, the reserve percent is `0` (no extra space is reserved). You can specify any percentage between `1` and `100` and values greater than `100`. The value specifies the percentage of the uncompressed size of the variable.

References

This function corresponds to the CDF library C API routine `CDFgetzVarReservePercent`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setVarReservePercent`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getVarSparseRecords

Information about how variable handles sparse records

Syntax

```
stype = cdflib.getVarSparseRecords(cdfId,varNum)
```

Description

`stype = cdflib.getVarSparseRecords(cdfId,varNum)` returns information about how a variable in the Common Data Format (CDF) file handles sparse records.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

varNum

Numeric value that identifies the variable. Variable numbers are zero-based.

Output Arguments

stype

One of the following character vectors, or its numeric equivalent, that specifies how the variable handles sparse records.

Value	Description
`'NO_SPARSERECORDS'`	No sparse records.

Value	Description
'PAD_SPARSERECORDS'	For sparse records, the library uses the variable's pad value when reading values from a missing record.
'PREV_SPARSERECORDS'	For sparse records, the library uses values from the previous existing record when reading values from a missing record. If there is no previous existing record, the library uses the variable's pad value.

To get the numeric equivalent of the constants specified by these character vectors, use `cdflib.getConstantValue`.

Examples

Open the example CDF, and then get the sparse record type of a variable in the file:

```
cdfid = cdflib.open('example.cdf');

stype = cdflib.getVarSparseRecords(cdfid,0)

stype =

NO_SPARSERECORDS

%Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarSparseRecords`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setVarSparseRecords`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.getVersion

Common Data Format (CDF) library version and release information

Syntax

```
[version,release,increment] = cdflib.getVersion(cdfId)
```

Description

`[version,release,increment] = cdflib.getVersion(cdfId)` returns information about the version of the Common Data Format (CDF) library used to create a CDF file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

Output Arguments

`version`

Numeric value indicating the version number of the CDF library.

`release`

Numeric value indicating the release number of the CDF library.

`increment`

Numeric value indicating the increment number of the CDF library.

Examples

Open the example CDF file, and then find out the version of the CDF library used to create it:

```
cdfId = cdflib.open('example.cdf');

[version, release, increment] = cdflib.getVersion(cdfId)

version =

     2

release =

     7

increment =

     8

% Clean up
cdflib.close(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetVersion`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getLibraryVersion`

Topics

"Import CDF Files Using Low-Level Functions"

“Export to CDF Files”

cdflib.hyperGetVarData

Read hyperslab of data from variable

Syntax

```
data = cdflib.hyperGetVarData(cdfId,varNum,recSpec,dimSpec)
data = cdflib.hyperGetVarData(cdfId,varNum,recSpec)
```

Description

`data = cdflib.hyperGetVarData(cdfId,varNum,recSpec,dimSpec)` reads a hyperslab of data from a variable in the Common Data Format (CDF) file. Hyper access allows more than one value to be read from or written to a variable with a single call to the CDF library.

`data = cdflib.hyperGetVarData(cdfId,varNum,recSpec)` reads a hyperslab of data for a zero-dimensional variable in the Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Number identifying the variable containing the datum.

`recSpec`

Three-element array, `[RSTART RCOUNT RSTRIDE]`, where `RSTART`, `RCOUNT`, and `RSTRIDE` are scalar values specifying the starting record, number of records to read, and the sampling interval or stride between records. Record numbers are zero-based.

dimSpec

Three-element cell array, `{DSTART DCOUNT DSTRIDE}`, where `DSTART`, `DCOUNT`, and `DSTRIDE` are *n*-element vectors that describe the start, number of values along each dimension, and sampling interval along each dimension. If the hyperslab has zero dimensions, you can omit this parameter. Dimension indices are zero-based.

Examples

Open the example CDF file, and then get all the data associated with a variable:

```
cdfid = cdflib.open('example.cdf');

% Determine the number of records allocated for the first variable in the file.
maxRecNum = cdflib.getVarMaxWrittenRecNum(cdfid,0);

% Retrieve all data in records for variable.
data = cdflib.hyperGetVarData(cdfid,0,[0 maxRecNum 1]);

%  Clean up
cdflib.close(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFhyperGetzVarData`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.hyperPutVarData`

Topics

"Import CDF Files Using Low-Level Functions"

“Export to CDF Files”

cdflib.hyperPutVarData

Write hyperslab of data to variable

Syntax

```
cdflib.hyperPutVarData(cdfId,varNum,recSpec,dimSpec,data)
```

Description

`cdflib.hyperPutVarData(cdfId,varNum,recSpec,dimSpec,data)` writes a hyperslab of data to a variable in a Common Data Format (CDF) file. Hyper access allows more than one value to be read from or written to a variable with a single call to the CDF library.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Specifies the variable containing the datum.

`recSpec`

Three-element array described by `[RSTART RCOUNT RSTRIDE]`, where `RSTART`, `RCOUNT`, and `RSTRIDE` are scalar values giving the start, number of records, and sampling interval (or stride) between records. Record indices are zero-based.

`dimSpec`

Three-element cell array described by `{DSTART DCOUNT DSTRIDE}`, where `DSTART`, `DCOUNT`, and `DSTRIDE` are n-element vectors that describe the start, number of values

along each dimension, and sampling interval along each dimension. If the hyperslab has zero dimensions, you can omit this parameter. Dimension indices are zero-based.

`data`

Data to write to the variable.

Examples

Create a CDF, create a variable, and then write a slab of data to the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Grades','cdf_int1',1,[],true,[]);

% Write data to the variable
cdflib.hyperPutVarData(cdfid,varNum,0,[],int8(98))


%Clean up
cdflib.delete(cdfid);
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFhyperzPutVarData`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.hyperGetVarData`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.inquire

Basic characteristics of Common Data Format (CDF) file

Syntax

```
info = cdflib.inquire(cdfId)
```

Description

`info = cdflib.inquire(cdfId)` returns basic information about a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

Output Arguments

info

A structure containing the following fields:

Field	Description
`encoding`	Encoding of the variable data and attribute entry data
`majority`	Majority of the variable data
`maxRec`	Maximum record number written to a CDF variable
`numVars`	Number of CDF variables
`numvAttrs`	Number of attributes with variable scope

Field	Description
numgAttrs	Number of attributes with global scope

Examples

Open the example CDF file, and then get basic information about the file:

```
cdfId = cdflib.open('example.cdf');

info = cdflib.inquire(cdfId)

info =

     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: 23
      numVars: 5
    numvAttrs: 1
    numgAttrs: 3
```

References

This function corresponds to the CDF library C API routines `CDFinquireCDF` and `CDFgetNumgAttributes`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.inquireVar`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.inquireAttr

Information about attribute

Syntax

```
info = cdflib.inquireAttr(cdfId,attrNum)
```

Description

`info = cdflib.inquireAttr(cdfId,attrNum)` returns information about an attribute in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

attrNum

Numeric value that identifies the attribute in the file. Attribute numbers are zero-based.

Output Arguments

info

Structure containing the following fields.

Field	Description
`name`	Attribute's name
`scope`	Either `'GLOBAL_SCOPE'` or `'VARIABLE_SCOPE'`

Field	Description
`maxgEntry`	The maximum entry number used for global attributes.
`maxEntry`	The maximum entry number used for attributes with variable scope.

Examples

Open the example CDF, and then get information about the first attribute in the file.

```
cdfid = cdflib.open('example.cdf');

% Get information about an attribute
info = cdflib.inquireAttr(cdfid,0)

info =

         name: 'SampleAttribute'
        scope: 'GLOBAL_SCOPE'
    maxgEntry: 4
     maxEntry: -1

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFinquireAttr`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.inquireAttrEntry` | `cdflib.inquireAttrgEntry`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.inquireAttrEntry

Information about entry in attribute with variable scope

Syntax

```
[datatype,numElements] =
cdflib.inquireAttrEntry(cdfId,attrNum,entryNum)
```

Description

`[datatype,numElements] = cdflib.inquireAttrEntry(cdfId,attrNum,entryNum)` returns the data type and the number of elements for an attribute entry in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrNum`

Numeric value identifying an attribute in the file. Attribute numbers are zero-based. The attribute must have variable scope.

`entryNum`

Numeric value identifying the entry in the attribute. Entry number are zero-based.

Output Arguments

datatype

Character vector identifying a CDF data type. For a list of CDF data types, see `cdflib.putAttrEntry`

numElements

Numeric value indicating the number of elements in the entry.

Examples

Open example CDF, and then get information about entries associated with an attribute in the file:

```
cdfid = cdflib.open('example.cdf');

% The fourth attribute is of variable scope.
attrscope = cdflib.getAttrScope(cdfid,3)

attrscope =

VARIABLE_SCOPE

% Get information about the first entry for this attribute
[dtype numel] = cdflib.inquireAttrEntry(cdfid,3,0)

dtype =

cdf_char


numel =

    10


% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFinquireAttrzEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getAttrScope` | `cdflib.inquireAttr`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.inquireAttrgEntry

Information about entry in attribute with global scope

Syntax

```
[datatype,numElements] =
cdflib.inquireAttrgEntry(cdfId,attrNum,entryNum)
```

Description

`[datatype,numElements] = cdflib.inquireAttrgEntry(cdfId,attrNum,entryNum)` returns the data type and the number of elements for a global attribute entry in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrNum`

Numeric value identifying an attribute in the file. Attribute numbers are zero-based. The attribute must have global scope.

`entryNum`

Numeric value identifying the entry in the attribute. Entry number are zero-based.

Output Arguments

datatype

Character vector identifying a CDF data type. For a list of CDF data types, see `cdflib.putAttrgEntry`

numElements

Numeric value indicating the number of elements in the entry.

Examples

Open the example CDF, and then get information about entries associated with a global attribute in the file.

```
cdfid = cdflib.open('example.cdf');

% Any of the first three attributes have global scope.
attrscope = cdflib.getAttrScope(cdfid,0)

attrscope =

GLOBAL_SCOPE

% Get information about the first entry for this attribute
[dtype numel] = cdflib.inquireAttrgEntry(cdfid,0,0)

dtype =

cdf_char


numel =

    23


% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFinquireAttrgEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.inquireAttr` | `cdflib.inquireAttrEntry`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.inquireVar

Information about variable

Syntax

```
info = cdflib.inquireVar(cdfId,varNum)
```

Description

`info = cdflib.inquireVar(cdfId,varNum)` returns information about a variable in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value that identifies the variable. Variable numbers are zero-based.

Output Arguments

`info`

Structure containing the following fields.

Field	Description
`name`	Name of the variable
`datatype`	Data type

Field	Description
numElements	Number of elements of the datatype
dims	Sizes of the dimensions
recVariance	Record variance
dimVariance	Dimension variances

Record and dimension variances affect how the library physically stores variable data. For example, if a variable has a record variance of VARY, the library physically stores each record. If the record variance is NOVARY, the library only stores one record.

Examples

Open the example CDF file and get information about a variable.

```
cdfid = cdflib.open('example.cdf');

% Determine if the file contains variables
info  = cdflib.inquireVar(cdfid,1)

info =

           name: 'Longitude'
       datatype: 'cdf_int1'
    numElements: 1
           dims: [2 2]
    recVariance: 0
    dimVariance: [1 0]
```

References

This function corresponds to the CDF library C API routine CDFinquirezVar.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the CDF website.

For copyright information, see the cdfcopyright.txt file.

See Also

`cdflib.inquire`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.open

Open existing Common Data Format (CDF) file

Syntax

`cdfId = cdflib.open(filename)`

Description

`cdfId = cdflib.open(filename)` opens an existing Common Data Format (CDF) file. *`filename`* is a character vector or string scalar that identifies the file.

This function returns a CDF file identifier, `cdfId`.

All CDF files opened this way have the `zMode` set to `zModeon2`. Refer to the CDF User's Guide for information about `zModes`.

Examples

Open the example CDF file:

```
cdfId = cdflib.open('example.cdf');

% Clean up
cdflib.close(cdfId)

clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFopenCDF`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.close` | `cdflib.create`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.putAttrEntry

Write value to entry in attribute with variable scope

Syntax

```
cdflib.putAttrEntry(cdfId,attrNum,entryNum,CDFDataType,entryVal)
```

Description

`cdflib.putAttrEntry(cdfId,attrNum,entryNum,CDFDataType,entryVal)` writes a value to an attribute entry in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrNum`

Number identifying attribute. The attribute must have variable scope. Attribute numbers are zero-based.

`entryNum`

Number identifying entry. Entry numbers are zero-based.

`CDFdatatype`

Data type of the attribute entry, specified as one of the following character vectors or string scalars, or its numeric equivalent.

CDF Data Type	MATLAB Equivalent
`'CDF_BYTE'`	1-byte, signed integer

CDF Data Type	MATLAB Equivalent
'CDF_CHAR'	1 byte, signed character data type that maps to the MATLAB `char` or `string` class
'CDF_INT1'	1-byte, signed integer.
'CDF_UCHAR'	1 byte, unsigned character data type that maps to the MATLAB `uint8` class
'CDF_UINT1'	1-byte, unsigned integer
'CDF_INT2'	2-byte, signed integer
'CDF_UINT2'	2-byte, unsigned integer.
'CDF_INT4'	4-byte, signed integer
'CDF_UINT4'	4-byte, unsigned integer
'CDF_FLOAT'	4-byte, floating point
'CDF_REAL4'	4-byte, floating point
'CDF_REAL8'	8-byte, floating point.
'CDF_DOUBLE'	8-byte, floating point
'CDF_EPOCH'	8-byte, floating point
'CDF_EPOCH16'	two 8-byte, floating point

`entryVal`

Data to be written to attribute entry.

Examples

Create a CDF and create an attribute with variable scope in the file. Write a value to an entry in the attribute. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Initially the file contains no attributes, global or variable.
info = cdflib.inquire(cdfid)

info =

     encoding: 'IBMPC_ENCODING'
```

```
     majority: 'ROW_MAJOR'
       maxRec: -1
      numVars: 0
    numvAttrs: 0
    numgAttrs: 0

% Create an attribute of variable scope in the file.
attrNum = cdflib.createAttr(cdfid,'Another Attribute','variable_scope');

% Write a value to an entry for the attribute
cdflib.putAttrEntry(cdfid,attrNum,0,'CDF_CHAR','My Variable Attribute Test');

% Get the value of the global attribute entry
value = cdflib.getAttrEntry(cdfid,attrNum,0)

value =

My Variable Attribute Test


% Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFputAttrzEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getAttrEntry` | `cdflib.getAttrgEntry` | `cdflib.getConstantValue` | `cdflib.putAttrgEntry`

Topics

"Import CDF Files Using Low-Level Functions"

“Export to CDF Files”

cdflib.putAttrgEntry

Write value to entry in attribute with global scope

Syntax

```
cdflib.putAttrgEntry(cdfId,attrNum,entryNum,cdfDataType,entryVal)
```

Description

`cdflib.putAttrgEntry(cdfId,attrNum,entryNum,cdfDataType,entryVal)` writes a value to a global attribute entry in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`attrNum`

Number identifying attribute. Attribute numbers are zero-based. The attribute must have global scope.

`entryNum`

Number identifying entry. Entry numbers are zero-based.

`CDFdatatype`

Data type of the attribute entry, specified as one of the following character vectors or string scalars, or its numeric equivalent.

CDF Data Type	MATLAB Equivalent
`'CDF_BYTE'`	1-byte, signed integer

CDF Data Type	MATLAB Equivalent
'CDF_CHAR'	1 byte, signed character data type that maps to the MATLAB `char` or `string` class
'CDF_INT1'	1-byte, signed integer.
'CDF_UCHAR'	1 byte, unsigned character data type that maps to the MATLAB `uint8` class
'CDF_UINT1'	1-byte, unsigned integer
'CDF_INT2'	2-byte, signed integer
'CDF_UINT2'	2-byte, unsigned integer.
'CDF_INT4'	4-byte, signed integer
'CDF_UINT4'	4-byte, unsigned integer
'CDF_FLOAT'	4-byte, floating point
'CDF_REAL4'	4-byte, floating point
'CDF_REAL8'	8-byte, floating point.
'CDF_DOUBLE'	8-byte, floating point
'CDF_EPOCH'	8-byte, floating point
'CDF_EPOCH16'	two 8-byte, floating point

`entryVal`

Data to be written to global attribute entry.

Examples

Create a CDF and create a global attribute in the file. Write a value to an entry in the attribute. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Initially the file contains no attributes, global or variable.
info = cdflib.inquire(cdfid)

info =

     encoding: 'IBMPC_ENCODING'
```

```
     majority: 'ROW_MAJOR'
       maxRec: -1
      numVars: 0
    numvAttrs: 0
    numgAttrs: 0

% Create a global attribute in the file.
attrNum = cdflib.createAttr(cdfid,'Purpose','global_scope');

% Write a value to an entry for the global attribute
cdflib.putAttrgEntry(cdfid,attrNum,0,'CDF_CHAR','My Test');

% Get the value of the global attribute entry
value = cdflib.getAttrgEntry(cdfid,attrNum,0)

value =

My Test

% Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFputAttrgEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getAttrEntry` | `cdflib.getAttrgEntry` | `cdflib.getConstantValue` | `cdflib.putAttrEntry`

Topics

"Import CDF Files Using Low-Level Functions"

“Export to CDF Files”

cdflib.putVarData

Write single value to variable

Syntax

```
cdflib.putVarData(cdfId,varNum,recNum,indices,datum)
```

Description

`cdflib.putVarData(cdfId,varNum,recNum,indices,datum)` writes a single value to a variable in a Common Data File (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value that identifies the variable to which you want to write the datum. Variable numbers are zero-based.

`recNum`

Numeric value that identifies the record to which you want to write the datum. Record numbers are zero-based.

`dims`

Dimension indices within the record. Dimension indices are zero-based.

`datum`

Data to be written to the variable.

Examples

Create a CDF, create a variable in the CDF and write data to the variable. To run this example, you must have write permission in the current folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Grades','cdf_int1',1,[],true,[]);

% Write some data to the variable
cdflib.putVarData(cdfid,varNum,0,[],int8(98))

% Read the value from the variable.
datum = cdflib.getVarData(cdfid,varNum,0)

datum =

   98

%Clean up
cdflib.delete(cdfid);
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFputzVarData`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarData` | `cdflib.getVarRecordData` | `cdflib.hyperGetVarData`

Topics

"Import CDF Files Using Low-Level Functions"

“Export to CDF Files”

cdflib.putVarRecordData

Write entire record to variable

Syntax

```
cdflib.putVarRecordData(cdfId,varNum,recNum,recordData)
```

Description

`cdflib.putVarRecordData(cdfId,varNum,recNum,recordData)` writes data to a record in a variable in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value that identifies the variable to which you want to write the datum. Variable numbers are zero-based.

`recNum`

Numeric value identifying the location of the datum in the variable. Record numbers are zero-based.

`recordData`

Data to be written to the variable.

Examples

Create a CDF, create a variable, and write an entire record of data to the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Grades','cdf_int1',1,[],true,[]);

% Write some data to the variable
cdflib.putVarRecordData(cdfid,varNum,0,int8(98))

% Read the value from the variable.
datum = cdflib.getVarData(cdfid,varNum,0)

datum =

   98

%Clean up
cdflib.delete(cdfid);
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFputzVarRecordData`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarRecordData` | `cdflib.hyperPutVarData` | `cdflib.putVarData`

Topics

"Import CDF Files Using Low-Level Functions"

“Export to CDF Files”

cdflib.renameAttr

Rename existing attribute

Syntax

```
cdflib.renameAttr(cdfId,attrNum,newName)
```

Description

`cdflib.renameAttr(cdfId,attrNum,newName)` renames an attribute in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `attrNum` is a numeric value that identifies the attribute. Attribute numbers are zero-based. `newName` is a character vector or string scalar that specifies the name you want to assign to the attribute.

Examples

Create a CDF, create an attribute in the CDF, and then rename the attribute. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create an attribute
attrNum = cdflib.createAttr(cdfid,'Purpose','global_scope');

% Rename the attribute
cdflib.renameAttr(cdfid, attrNum,'NewPurpose');

% Check the name of the attribute
attrName = cdflib.getAttrName(cdfid,anum)

attrName =

NewPurpose
```

```
% Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFrenameAttr`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.createAttr`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.renameVar

Rename existing variable

Syntax

```
cdflib.renameVar(cdfId,varNum,newName)
```

Description

`cdflib.renameVar(cdfId,varNum,newName)` renames a variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers are zero-based. `newName` is a character vector or string scalar that specifies the name you want to assign to the variable.

Examples

Create a CDF, create a variable in the CDF, and then rename the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Get the name of the variable.
name = cdflib.getVarName(cdfid,varNum)

name =

Time

% Rename the variable
cdflib.renameVar(cdfid,varNum,'NewName');
```

```
% Check the new name.
name = cdflib.getVarName(cdfid,varNum)

name =

NewName

%  Clean up
cdflib.delete(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFrenamezVar`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.createVar`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.setCacheSize

Specify number of dotCDF cache buffers

Syntax

```
cdflib.setCacheSize(cdfId,numBuffers)
```

Description

`cdflib.setCacheSize(cdfId,numBuffers)` specifies the number of cache buffers the CDF library uses for an open dotCDF file. A dotCDF file is a file with the `.cdf` file extension.

`cdfId` identifies an open CDF file. `numBuffers` is a numeric value that specifies the number of buffers.

For information about cache schemes, see the CDF User's Guide.

Examples

Create a CDF file and set the cache size. To run this example, you must have write permission in your current folder.

```
cdfId = cdflib.create('your_file.cdf');

% Get the default cache size
numBuf = cdflib.getCacheSize(cdfid)

numBuf =

   300

% Specify a cache size
cdflib.setCacheSize(cdfid,150)

% Check the cache size again
```

```
numBuf =  cdflib.getCacheSize(cdfid)

numBuf =

   150

% Clean up
cdflib.delete(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFsetCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getCacheSize`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.setChecksum

Specify checksum mode

Syntax

`cdflib.setChecksum(cdfId,mode)`

Description

`cdflib.setChecksum(cdfId,mode)` specifies the checksum mode of a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`mode`

One of the following character vectors or string scalars, or its numeric equivalent.

`'MD5_CHECKSUM'`	Sets file checksum to MD5 checksum.
`'NO_CHECKSUM'`	File does not use a checksum.

To get the numeric equivalent of these constants, use `cdflib.getConstantValue`.

Examples

Create a CDF file and set the checksum mode. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('mycdf.cdf');
```

```
% Check initial value of checksum.
mode = cdflib.getChecksum(cdfid)

NO_CHECKSUM

cdflib.setChecksum(cdfid,'MD5_CHECKSUM')

% Verify the setting
mode = cdflib.getChecksum(cdfid)

MD5_CHECKSUM
```

References

This function corresponds to the CDF library C API routine `CDFsetChecksum`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getChecksum` | `cdflib.getConstantValue`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.setCompression

Specify compression settings

Syntax

```
cdflib.setCompression(cdfId,ctype,cparms)
```

Description

`cdflib.setCompression(cdfId,ctype,cparms)` specifies compression settings of a Common Data Format (CDF) file.

This function sets the compression for the CDF file itself, not that of any variables in the file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

ctype

One of the following character vectors or string scalars specifying the compression type, or its numeric equivalent.

Value	Compression Type
`'NO_COMPRESSION'`	No compression
`'RLE_COMPRESSION'`	Run-length encoding compression
`'HUFF_COMPRESSION'`	Huffman compression
`'AHUFF_COMPRESSION'`	Adaptive Huffman compression
`'GZIP_COMPRESSION'`	GNU's zip compression

To get the numeric equivalent of these constants, use `cdflib.getConstantValue`.

`cparms`

Optional parameter specifying any additional parameters required by the compression type. Currently, the only compression type that uses this parameter is `'GZIP_COMPRESSION'`. For this compression type, use `cparms` to specify the level of compression as a numeric value between 1 and 9.

Examples

Create a CDF file and set the compression setting of the file. To run this example, your current folder must be writable.

```
cdfId = cdflib.create('your_file.cdf');

% Determine the file's default compression setting
[ctype, cparms, cpercent ] = cdflib.getCompression(cdfId)

ctype =

NO_COMPRESSION

cparms =

     []

cpercent =

   100

% Specify new compression setting
cdflib.setCompression(cdfId,'HUFF_COMPRESSION');

% Check the file's compression setting.
[ctype, cparms, cpercent ] = cdflib.getCompression(cdfId)

ctype =

HUFF_COMPRESSION

cparms =
```

```
OPTIMAL_ENCODING_TREES


cpercent =

     0

% Clean up
cdflib.delete(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFsetCompression`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getCompression` | `cdflib.getConstantValue`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.setCompressionCacheSize

Specify number of compression cache buffers

Syntax

```
cdflib.setCompressionCacheSize(cdfId,numBuffers)
```

Description

`cdflib.setCompressionCacheSize(cdfId,numBuffers)` specifies the number of cache buffers used for the compression scratch CDF file. For more information about CDF cache schemes, see the CDF User's Guide.

`cdfId` identifies the CDF file. `numBuffers` specifies the number of buffers.

Examples

Create a CDF file and specify the number of compression cache buffers used. To run this example you must be in a writable folder.

```
cdfId = cdflib.create('your_file.cdf');

% Get the current number of compression cache buffers
numBuf = cdflib.getCompressionCacheSize(cdfId)

numBuf =

    80

% Set a new value
cdflib.setCompressionCacheSize(cdfId,100)

% Check the new value
numBuf = cdflib.getCompressionCacheSize(cdfId)

numBuf =
```

```
   100

% Clean up
cdflib.delete(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFsetCompressionCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getCompressionCacheSize`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.setFileBackward

Set backward compatibility mode

Syntax

```
cdflib.setFileBackward(mode)
```

Description

`cdflib.setFileBackward(mode)` sets the backward compatibility mode to the value specified by `mode`.

Input Arguments

`mode`

Compatibility mode, specified as one of these character vectors or string scalars.

`'BACKWARDFILEon'`	Set backward compatibility mode on.
`'BACKWARDFILEoff'`	Set backward compatibility mode off.

Default: `'BACKWARDFILEoff'`

Examples

Set backward compatibility mode and then check the value.

```
cdflib.setFileBackward('BACKWARDFILEon');

mode = cdflib.getFileBackward

mode =

BACKWARDFILEon
```

More About

backward compatibility mode

When specified, ensures that any new CDF file created using CDF V3.0 (or later) will be readable by clients using version 2.7 of the CDF library. CDF 3.0 and later releases use a 64-bit file offset to allow for files greater than 2G bytes in size. CDF library versions released before CDF 3.0 use a 32-bit file offset.

Tips

- Setting backward compatibility mode affects only your current MATLAB session, or until you call `cdflib.setFileBackward` again.

References

This function corresponds to the CDF library C API routine `CDFsetFileBackward`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getConstantValue` | `cdflib.getFileBackward`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.setFormat

Specify format of Common Data Format (CDF) file

Syntax

`cdflib.setFormat(cdfId,format)`

Description

`cdflib.setFormat(cdfId,format)` specifies the format of a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

format

Format of CDF file, specified as a character vector or string scalar containing one of these format values.

`'SINGLE_FILE'`	The CDF consists of only one file. This is the default file format
`'MULTI_FILE'`	The CDF consists of one header file for control and attribute data and one additional file for each variable in the CDF.

You can also specify `format` using a numeric value. To get the numeric equivalent of the formats, use `cdflib.getConstantValue`.

Examples

Create a CDF file and specify its format. To run this example, you must have write permission in your current folder.

```
cdfId = cdflib.create('mycdffile.cdf');

% Specify multifile format.
cdflib.setFormat(cdfId, 'MULTI_FILE');

% Check format.
 format = cdflib.getFormat(cdfId)

format =

MULTI_FILE

% Clean up
cdflib.delete(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFsetFormat`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getConstantValue` | `cdflib.getFormat`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.setMajority

Specify majority of variables

Syntax

`cdflib.setMajority(cdfId,majority)`

Description

`cdflib.setMajority(cdfId,majority)` specifies the majority of variables in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

majority

One of the following format values specified as a character vector or string scalar, or its numeric equivalent.

`'ROW_MAJOR'`	C-like array ordering for variable storage. The first dimension in each variable array varies the slowest. This is the default.
`'COLUMN_MAJOR'`	Fortran-like array ordering for variable storage. The first dimension in each variable array varies the fastest.

To get the numeric equivalent of these format values, use `cdflib.getConstantValue`.

Examples

Create a CDF file and specify the majority used by variables in the file. To run this example, you must have write permission in your current folder.

```
cdfId = cdflib.create('your_file.cdf')

% Specify the majority used by variables in the file
cdflib.setMajority(cdfId,'COLUMN_MAJOR');

% Check the majority value
majority = cdflib.getMajority(cdfId)

majority =

COLUMN_MAJOR

% Clean up
cdflib.delete(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFsetMajority`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getMajority`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.setReadOnlyMode

Specify read-only mode

Syntax

`cdflib.setReadOnlyMode(cdfId,mode)`

Description

`cdflib.setReadOnlyMode(cdfId,mode)` specifies the read-only mode of a Common Data Format (CDF) file.

After you open a CDF file, you can put the file into read-only mode to prevent accidental modification.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`mode`

Mode value, specified as one of these character vectors or string scalars, or its numeric equivalent.

`'READONLYon'`	CDF file is read-only
`'READONLYoff'`	CDF file is modifiable.

To get the numeric equivalent of these mode values, use `cdflib.getConstantValue`.

Examples

Open the example CDF file and set the file to read-only mode.

```
cdfId = cdflib.open('example.cdf');

% Set the  file to READONLY mode
cdflib.setReadOnlyMode(cdfId,'READONLYon')

% Check read-only status of file again.
mode = cdflib.getReadOnlyMode(cdfId)

mode =

READONLYon

% Clean up
cdflib.close(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFsetReadOnlyMode`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getConstantValue` | `cdflib.getReadOnlyMode`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.setStageCacheSize

Specify number of staging cache buffers for Common Data Format (CDF) file

Syntax

```
cdflib.setStageCacheSize(cdfId,numBuffers)
```

Description

`cdflib.setStageCacheSize(cdfId,numBuffers)` specifies the number of staging cache buffers for a Common Data Format (CDF) file. For information about CDF cache schemes, see the CDF User's Guide.

`cdfId` identifies the CDF file. `numBuffers` is a numeric value that specifies the number of buffers.

Examples

Open the example CDF file and specify the number of cache buffers used.

```
cdfId = cdflib.open('example.cdf');

% Get current number of staging cache buffers
size = cdflib.getStageCacheSize(cdfId)

size =

   125

% Specify new cache size value.
cdflib.setStageCacheSize(cdfId, 200)

% Get size again.
size = cdflib.getStageCacheSize(cdfId)

size =
```

```
   200

% Clean up
cdflib.close(cdfId)

clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFsetStageCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getStageCacheSize`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.setValidate

Specify library validation mode

Syntax

`cdflib.setValidate(mode)`

Description

`cdflib.setValidate(mode)` specifies the validation mode of the Common Data Format (CDF) library. Specify the validation mode before opening any files.

Input Arguments

`mode`

One of the following character vectors or string scalars, or its numeric equivalent:

`'VALIDATEFILEon'`	Turns validation mode on. With validation mode on, the library performs sanity checks on the data fields in the CDF' file's internal data structures to make sure that the values are within valid ranges and consistent with the defined values/types/entries. This mode also ensures that variable and attribute associations within the file are valid. Note, however, that enabling this mode will, in most cases, slow down the file opening process, especially for large or very fragmented files.
`'VALIDATEFILEoff'`	Turns validation mode off.

To get the numeric equivalent of these values, use `cdflib.getConstantValue`.

Examples

Set the validation mode of the CDF library.

```
cdflib.setValidate('VALIDATEFILEon');
```

References

This function corresponds to the CDF library C API routine `CDFsetValidate`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getConstantValue` | `cdflib.getValidate`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.setVarAllocBlockRecords

Specify range of records to be allocated for variable

Syntax

```
cdflib.setVarAllocBlockRecords(cdfId,varNum,firstrec,lastrec)
```

Description

`cdflib.setVarAllocBlockRecords(cdfId,varNum,firstrec,lastrec)` specifies a range of records you want to allocate (but not write) for a variable in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value identifying a variable in the file. Variable identifiers (variable numbers) are zero-based.

`firstRec`

Numeric value identifying the record at which to start allocating. Record numbers are zero-based.

`lastRec`

Numeric value identifying the record at which to stop allocating. Record numbers are zero-based.

Examples

Create a CDF, create a variable in the CDF, and then specify the number of records to allocate for the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Grades','cdf_int1',1,[],true,[]);

% Specify the number of records to allocate.
cdflib.setVarAllocBlockRecords(cdfid,varNum,1,10);


%  Clean up
cdflib.delete(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFsetzVarAllocBlockRecords`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarAllocRecords`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.setVarBlockingFactor

Specify blocking factor for variable

Syntax

```
cdflib.setVarBlockingFactor(cdfId,varNum,blockingFactor)
```

Description

`cdflib.setVarBlockingFactor(cdfId,varNum,blockingFactor)` specifies the blocking factor for a variable in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value identifying a variable in the file. Variable numbers are zero-based.

`blockingFactor`

Numeric value that specifies the number of records to allocate when writing to an unallocated record.

Examples

Create a CDF, create a variable in the CDF, and then set the blocking factor used with the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');
```

```
% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Get the current blocking factor used with the variable
bFactor = cdflib.getVarBlockingFactor(cdfid,varNum)

bFactor =

    0

% Change the blocking factor for the variable
cdflib.setVarBlockingFactor(cdfid,varNum,10);

% Check the new blocking factor .
bFactor = cdflib.getVarBlockingFactor(cdfid,varNum)

bFactor =

    10

%  Clean up
cdflib.delete(cdfid)

clear cdfid
```

More About

blocking factor

A variable's blocking factor specifies the minimum number of records the library allocates when you write to an unallocated record. If you specify a fractional blocking factor, the library rounds the value down.

References

This function corresponds to the CDF library C API routine `CDFsetzVarBlockingFactor`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarBlockingFactor`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.setVarCacheSize

Specify number of multi-file cache buffers for variable

Syntax

```
cdflib.setVarCacheSize(cdfId,varNum,numBuffers)
```

Description

`cdflib.setVarCacheSize(cdfId,varNum,numBuffers)` specifies the number of cache buffers the CDF library uses for a variable in a Common Data Format (CDF) file.

This function is only used with multifile format CDF files. It does not apply to single-file format CDFs. For more information about caching, see the CDF User's Guide.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value identifying a variable in the file. Variable identifiers (variable numbers) are zero-based.

`numBuffers`

Numeric value identifying the number of cache buffers to use.

Examples

Create a multifile CDF, and then retrieve the number of buffers being used for a variable:

```
cdfid = cdflib.create('your_file.cdf')

% Set the format of the file to be multi-file
cdflib.setFormat(cdfid,'MULTI_FILE');

% Create a variable in the file
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Note how the library creates a separate file for the variable
ls your_file.*

your_file.cdf  your_file.z0

% Determine the number of cache buffers used with the variable
numBuf = cdflib.getVarCacheSize(cdfid,varNum)

numBuf =

     1

% Increase the number of cache buffers used.
cdflib.setVarCacheSize(cdfid,varNum,5)

% Check the number of cache buffers used with the variable.
numBuf = cdflib.getVarCacheSize(cdfid,varNum)

numBuf =

     5

%  Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFsetzVarCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarCacheSize` | `cdflib.setVarsCacheSize`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.setVarCompression

Specify compression settings used with variable

Syntax

```
cdflib.setVarCompression(cdfId,varNum,ctype,cparams)
```

Description

`cdflib.setVarCompression(cdfId,varNum,ctype,cparams)` configures the compression setting for a variable in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value identifying a variable in the file. Variable identifiers (variable numbers) are zero-based.

`ctype`

One of the following character vectors or string scalars specifying the compression type, or its numeric equivalent.

Value	Compression Type
`'NO_COMPRESSION'`	No compression.
`'RLE_COMPRESSION'`	Run-length encoding compression
`'HUFF_COMPRESSION'`	Huffman compression

Value	Compression Type
`'AHUFF_COMPRESSION'`	Adaptive Huffman compression
`'GZIP_COMPRESSION'`	GNU's zip compression

To get the numeric equivalent of these compression type values, use `cdflib.getConstantValue`.

`cparams`

Optional parameter specifying any additional parameters required by the compression type. Currently, the only compression type that uses this parameter is `'GZIP_COMPRESSION'`. For this compression type, you use `cparms` to specify the level of compression as a numeric value between 1 and 9.

Examples

Create a CDF, create a variable, and then set the compression used by the variable. To run this example, you must be in a folder with execute permission.

```
cdfid = cdflib.create('mycdf.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Specify the compression used by the variable.
cdflib.setVarCompression(cdfid,0,'GZIP_COMPRESSION',8)

% Check the compression setting of the variable
[ctype params percent] = cdflib.getVarCompression(cdfid,0)

ctype =

GZIP_COMPRESSION

params =

     8

percent =

     0
```

```
% Clean up
cdflib.delete(cdfid);
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFsetzVarCompression`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarCompression` | `cdflib.setCompression`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.setVarInitialRecs

Specify initial number of records written to variable

Syntax

```
cdflib.setVarInitialRecs(cdfId,varNum,numrecs)
```

Description

`cdflib.setVarInitialRecs(cdfId,varNum,numrecs)` specifies the initial number of records to write to a variable in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value identifying a variable in the file. Variable numbers are zero-based.

`numRecs`

Numeric value specifying the number of records to write.

Examples

Create a CDF, create a variable, and then specify the number of records to write for the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
```

```
varNum = cdflib.createVar(cdfid,'Grades','cdf_int1',1,[],true,[]);

% Specify the number of records to write for the variable
cdflib.setVarInitialRecs(cdfid,varNum,100);

recsWritten = cdflib.getVarNumRecsWritten(cdfid,varNum)

recsWritten =

   100

%  Clean up
cdflib.delete(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFsetzVarInitialRecs`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.createVar`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.setVarPadValue

Specify pad value used with variable

Syntax

```
cdflib.setVarPadValue(cdfId,varNum,padvalue)
```

Description

`cdflib.setVarPadValue(cdfId,varNum,padvalue)` specifies the pad value used with a variable in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value identifying a variable in the file. Variable numbers are zero-based.

`padValue`

Value to use a pad value for the variable. The data type of the pad value must match the data type of the variable.

Examples

Create a CDF, create a variable in the CDF, and then set the pad value used with the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');
```

```
% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Get the current pad value used with the variable
padval = cdflib.getVarPadValue(cdfid,varNum)

padval =

    0

% Change the pad value for the variable
cdflib.setVarPadValue(cdfid,varNum,int8(1));

% Check the new pad value.
padval = cdflib.getVarPadValue(cdfid,varNum)

padval =

    1

%  Clean up
cdflib.delete(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFsetzVarPadValue`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarPadValue`

Topics

"Import CDF Files Using Low-Level Functions"

“Export to CDF Files”

cdflib.SetVarReservePercent

Specify reserve percentage for variable

Syntax

```
cdflib.setVarReservePercent(cdfId,varNum,percent)
```

Description

`cdflib.setVarReservePercent(cdfId,varNum,percent)` specifies the compression reserve percentage for a variable in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value identifying a variable in the file. Variable identifiers (variable numbers) are zero-based.

`percent`

Numeric value specifying the amount of extra space to allocate for a compressed variable, expressed as a percentage. You can specify values between`0` (no extra space is reserved) and `100`, or values greater than `100`. The value specifies the percentage of the uncompressed size of the variable. If you specify a fractional reserve percentages, the library rounds the value down.

Examples

Create a CDF, create a variable, set the compression of the variable, and then set the reserve percent for the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Set the compression of the variable.
cdflib.setVarCompression(cdfid,varNum,'GZIP_COMPRESSION',8);

% Set the compression reserver percentage
cdflib.setVarReservePercent(cdfid,varNum, 80);

cdflib.close(cdfid);
```

More About

reserve percentage

Specifies how much extra space to allocate for a compressed variable. This extra space allows the variable to expand when you write additional records to the variable. If you do not specify this room for growth, the library has to move the variable to the end of the file when the size expands and the space at the original location of the variable becomes wasted space.

References

This function corresponds to the CDF library C API routine `CDFsetzVarReservePercent`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarCompression` | `cdflib.getVarReservePercent` | `cdflib.setVarCompression`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.setVarsCacheSize

Specify number of cache buffers used for all variables

Syntax

```
cdflib.setVarsCacheSize(cdfId,varNum,numBuffers)
```

Description

`cdflib.setVarsCacheSize(cdfId,varNum,numBuffers)` specifies the number of cache buffers the CDF library uses for all the variables in the multifile format Common Data Format (CDF) file.

This function is not applicable to single-file CDFs. For more information about caching, see the CDF User's Guide.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Numeric value identifying a variable in the file. Variable identifiers (variable numbers) are zero-based.

`numBuffers`

Numeric value specifying the cache buffers.

Examples

Create a multifile CDF and specify the number of buffers used for all variables. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf')

% Set the format of the file to be multi-file
cdflib.setFormat(cdfid,'MULTI_FILE');

% Create a variable in the file
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Note how the library creates a separate file for the variable
ls your_file.*

your_file.cdf  your_file.z0

% Determine the number of cache buffers used with the variable
numBuf = cdflib.getVarCacheSize(cdfid,varNum)

numBuf =

     1

% Specify the number of cache buffers used by all variables in CDF.
cdflib.setVarsCacheSize(cdfid,6)

% Check the number of cache buffers used with the variable.
numBuf = cdflib.getVarCacheSize(cdfid,varNum)

numBuf =

     6

%  Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFsetzVarsCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarCacheSize` | `cdflib.setVarCacheSize`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdflib.setVarSparseRecords

Specify how variable handles sparse records

Syntax

```
cdflib.getVarSparseRecords(cdfId,varNum,stype)
```

Description

`cdflib.getVarSparseRecords(cdfId,varNum,stype)` specifies the sparse records type of a variable in a Common Data Format (CDF) file.

Input Arguments

`cdfId`

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

`varNum`

Number that identifies the variable to be set. Variable numbers are zero-based.

`stype`

One of the following character vectors or string scalars that specifies how the variable handles sparse records, or its numeric equivalent.

Value	Description
`'NO_SPARSERECORDS'`	No sparse records
`'PAD_SPARSERECORDS'`	For sparse records, the library uses the variable's pad value when reading values from a missing record.

Value	Description
'PREV_SPARSERECORDS'	For sparse records, the library uses values from the previous existing record when reading values from a missing record. If there is no previous existing record, the library uses the variable's pad value.

To get the numeric equivalent of these values, use the `cdflib.getConstantValue` function.

Examples

Open a multifile CDF and close a variable.

Create a CDF, create a variable, and set the sparse records type of the variable. To run this example you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Set the sparse records type of the variable
cdflib.setVarSparseRecords(cdfid,varNum,'PAD_SPARSERECORDS');

% Check the sparse records type of the variable
stype = cdflib.getVarSparseRecords(cdfid,varNum)

stype =

PAD_SPARSERECORDS


%Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFsetzVarSparseRecords`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF website`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getConstantValue` | `cdflib.getVarSparseRecords`

Topics

"Import CDF Files Using Low-Level Functions"
"Export to CDF Files"

cdfread

Read data from Common Data Format (CDF) file

Syntax

```
data = cdfread(filename)
data = cdfread(filename, param1, val1, param2, val2, ...)
[data, info] = cdfread(filename, ...)
```

Description

`data = cdfread(filename)` reads all the data from the Common Data Format (CDF) file specified `filename`. Specify `filename` as a character vector or string scalar. CDF data sets typically contain a set of variables, of a specific data type, each with an associated set of records. The variable might represent time values with each record representing a specific time that an observation was recorded. `cdfread` returns all the data in a cell array where each column represents a variable and each row represents a record associated with a variable. If the variables have varying numbers of associated records, `cdfread` pads the rows to create a rectangular cell array, using pad values defined in the CDF file.

Note Because `cdfread` creates temporary files, the current working directory must be writeable.

`data = cdfread(filename, param1, val1, param2, val2, ...)` reads data from the file, where *`param1`*, *`param2`*, and so on, can be any of the parameters listed in the following table.

`[data, info] = cdfread(filename, ...)` returns details about the CDF file in the `info` structure.

Parameter	Value
`'Records'`	A vector specifying which records to read. Record numbers are zero-based. `cdfread` returns a cell array with the same number of rows as the number of records read and as many columns as there are variables.
`'Variables'`	A 1-by-*n* or *n*-by-1 cell array specifying the names of the variables to read from the file. *n* must be less than or equal to the total number of variables in the file. `cdfread` returns a cell array with the same number of columns as the number of variables read, and a row for each record read.
`'Slices'`	An *m*-by-3 array, where each row specifies where to start reading along a particular dimension of a variable, the skip interval to use on that dimension (every item, every other item, etc.), and the total number of values to read on that dimension. *m* must be less than or equal to the number of dimensions of the variable. If *m* is less than the total number of dimensions, `cdfread` reads every value from the unspecified dimensions (`[0 1 n]`, where *n* is the total number of elements in the dimension. Note: Because the `'Slices'` parameter describes how to process a single variable, it must be used in conjunction with the `'Variables'` parameter.
`'ConvertEpochToDatenum'`	A Boolean value that determines whether `cdfread` automatically converts CDF epoch data types to MATLAB serial date numbers. If set to `false` (the default), `cdfread` wraps epoch values in MATLAB `cdfepoch` objects. Note: For better performance when reading large data sets, set this parameter to `true`.

Parameter	Value
`'CombineRecords'`	A Boolean value that determines how `cdfread` returns the CDF data sets read from the file. If set to `false` (the default), `cdfread` stores the data in an *m*-by-*n* cell array, where *m* is the number of records and *n* is the number of variables requested. If set to `true`, `cdfread` combines all records for a particular variable into one cell in the output cell array. In this cell, `cdfread` stores scalar data as a column array. `cdfread` extends the dimensionality of nonscalar and string data. For example, instead of creating 1000 elements containing 20-by-30 arrays for each record, `cdfread` stores all the records in one cell as a 1000-by-20-by-30 array Note: If you use the `'Records'` parameter to specify which records to read, you cannot use the `'CombineRecords'` parameter. Note: When using the `'Variable'` parameter to read one variable, if the `'CombineRecords'` parameter is `true`, `cdfread` returns the data as an M-by-N numeric or character array; it does not put the data into a cell array.

Note To improve performance when working with large data files, use the `'ConvertEpochToDatenum'` and `'CombineRecords'` options.

Note To improve performance, turn off the file validation which the CDF library does by default when opening files. For more information, see `cdflib.setValidate`.

Examples

Read all the data from a CDF file.

```
data = cdfread('example.cdf');
```

Read the data from the variable `'Time'`.

```
data = cdfread('example.cdf', 'Variable', {'Time'});
```

Read the first value in the first dimension, the second value in the second dimension, the first and third values in the third dimension, and all values in the remaining dimension of the variable `'multidimensional'`.

```
data = cdfread('example.cdf', ...
               'Variable', {'multidimensional'}, ...
               'Slices', [0 1 1; 1 1 1; 0 2 2]);
```

This is similar to reading the whole variable into `data` and then using matrix indexing, as in the following.

```
data{1}(1, 2, [1 3], :)
```

Collapse the records from a data set and convert CDF epoch data types to MATLAB serial date numbers.

```
data = cdfread('example.cdf', ...
               'CombineRecords', true, ...
               'ConvertEpochToDatenum', true);
```

Limitations

- The `cdfread` function does not support non-ASCII encoded data. All the variable names, attributes names, variable values, and attribute values in the CDF file must have 7-bit ASCII encoding. Attempting to read non-ASCII encoded files results in errors or data with corrupted characters.

See Also

`cdfepoch` | `cdfinfo` | `cdflib.setValidate`

Topics

"Import CDF Files Using Low-Level Functions"

Introduced before R2006a

cdfwrite

Write data to Common Data Format (CDF) file

Note `cdfwrite` is not recommended. Use the `cdflib` low-level functions instead.

Syntax

```
cdfwrite(filename,variablelist)
cdfwrite(...,'PadValues',padvals)
cdfwrite(...,'GlobalAttributes',gattrib)
cdfwrite(..., 'VariableAttributes', vattrib)
cdfwrite(...,'WriteMode',mode)
cdfwrite(...,'Format',format)
```

Description

`cdfwrite(filename,variablelist)` writes out a Common Data Format (CDF) file, to the file name indicated in `filename`. Specify `filename` as a character vector or string scalar. The `variablelist` argument is a cell array of ordered pairs, each of which comprises a CDF variable name (specified as a character vector or string scalar) and the corresponding CDF variable value. To write out multiple records for a variable, put the values in a cell array where each element in the cell array represents a record.

Note Because `cdfwrite` creates temporary files, both the destination directory for the file and the current working directory must be writeable.

`cdfwrite(...,'PadValues',padvals)` writes out pad values for given variable names. `padvals` is a cell array of ordered pairs, each of which comprises a variable name (specified as a character vector or string scalar) and a corresponding pad value. Pad values are the default values associated with the variable when an out-of-bounds record is accessed. Variable names that appear in `padvals` must appear in `variablelist`.

`cdfwrite(...,'GlobalAttributes',gattrib)` writes the structure `gattrib` as global metadata for the CDF file. Each field of the structure is the name of a global attribute. The value of each field contains the value of the attribute. To write out multiple values for an attribute, put the values in a cell array where each element in the cell array represents a record.

Note To specify a global attribute name that is invalid in your MATLAB application, create a field called `'CDFAttributeRename'` in the attribute structure. The value of this field must have a value that is a cell array of ordered pairs. The ordered pair consists of the name of the original attribute, as listed in the `GlobalAttributes` structure, and the corresponding name of the attribute to be written to the CDF file.

`cdfwrite(..., 'VariableAttributes', vattrib)` writes the structure `vattrib` as variable metadata for the CDF. Each field of the struct is the name of a variable attribute. The value of each field should be an M-by-2 cell array where M is the number of variables with attributes. The first element in the cell array should be the name of the variable and the second element should be the value of the attribute for that variable.

Note To specify a variable attribute name that is illegal in MATLAB, create a field called `'CDFAttributeRename'` in the attribute structure. The value of this field must have a value that is a cell array of ordered pairs. The ordered pair consists of the name of the original attribute, as listed in the `VariableAttributes` struct, and the corresponding name of the attribute to be written to the CDF file. If you are specifying a variable attribute of a CDF variable that you are renaming, the name of the variable in the `VariableAttributes` structure must be the same as the renamed variable.

`cdfwrite(...,'WriteMode',`*`mode`*`)`, where *`mode`* is either `'overwrite'` or `'append'`, indicates whether or not the specified variables should be appended to the CDF file if the file already exists. By default, `cdfwrite` overwrites existing variables and attributes.

`cdfwrite(...,'Format',`*`format`*`)`, where *`format`* is either `'multifile'` or `'singlefile'`, indicates whether or not the data is written out as a multifile CDF. In a multifile CDF, each variable is stored in a separate file with the name `*.vN`, where `N` is the number of the variable that is written out to the CDF. By default, `cdfwrite` writes out a single file CDF. When `'WriteMode'` is set to `'Append'`, the `'Format'` option is ignored, and the format of the preexisting CDF is used.

Examples

Write out a file `'example.cdf'` containing a variable `'Longitude'` with the value `[0:360]`.

```
cdfwrite('example', {'Longitude', 0:360});
```

Write out a file `'example.cdf'` containing variables `'Longitude'` and `'Latitude'` with the variable `'Latitude'` having a pad value of 10 for all out-of-bounds records that are accessed.

```
cdfwrite('example', {'Longitude', 0:360, 'Latitude', 10:20}, ...
         'PadValues', {'Latitude', 10});
```

Write out a file `'example.cdf'`, containing a variable `'Longitude'` with the value `[0:360]`, and with a variable attribute of `'validmin'` with the value 10.

```
varAttribStruct.validmin = {'Longitude' [10]};
cdfwrite('example', {'Longitude' 0:360}, 'VariableAttributes', ...
         varAttribStruct);
```

Limitations

- CDF files do not support non-ASCII encoded inputs. Variable names, attributes names, variable values, and attribute values must have 7-bit ASCII encoding. Attempting to write non-ASCII encoded inputs results in errors or data with corrupted characters.

See Also

`cdfepoch` | `cdfinfo` | `cdfread`

Introduced before R2006a

ceil

Round toward positive infinity

Syntax

```
Y = ceil(X)
```

```
Y = ceil(t)
Y = ceil(t,unit)
```

Description

`Y = ceil(X)` rounds each element of `X` to the nearest integer greater than or equal to that element.

`Y = ceil(t)` rounds each element of the `duration` array `t` to the nearest number of seconds greater than or equal to that element.

`Y = ceil(t,unit)` rounds each element of `t` to the nearest number of the specified unit of time greater than or equal to that element.

Examples

Round Matrix Elements Toward Positive Infinity

```
X = [-1.9 -0.2 3.4; 5.6 7 2.4+3.6i];
Y = ceil(X)
```

```
Y = 2×3 complex

  -1.0000 + 0.0000i   0.0000 + 0.0000i   4.0000 + 0.0000i
   6.0000 + 0.0000i   7.0000 + 0.0000i   3.0000 + 4.0000i
```

Round Duration Values Toward Positive Infinity

Round each value in a duration array to the nearest number of seconds greater than or equal to that value.

```
t = hours(8) + minutes(29:31) + seconds(1.23);
t.Format = 'hh:mm:ss.SS'
```

```
t = 1x3 duration array
   08:29:01.23   08:30:01.23   08:31:01.23
```

```
Y1 = ceil(t)
```

```
Y1 = 1x3 duration array
   08:29:02.00   08:30:02.00   08:31:02.00
```

Round each value in t to the nearest number of hours greater than or equal to that value.

```
Y2 = ceil(t,'hours')
```

```
Y2 = 1x3 duration array
   09:00:00.00   09:00:00.00   09:00:00.00
```

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. For complex `X`, `ceil` treats the real and imaginary parts independently.

`ceil` converts logical and `char` elements of `X` into `double` values.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `logical`
Complex Number Support: Yes

`t` — Input duration
`duration` array

Input duration, specified as a `duration` array.

`unit` — Unit of time
`'seconds'` (default) | `'minutes'` | `'hours'` | `'days'` | `'years'`

Unit of time, specified as `'seconds'`, `'minutes'`, `'hours'`, `'days'`, or `'years'`. A duration of 1 year is equal to exactly 365.2425 24-hour days.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support `char` or `logical` data types for `X`.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Code generation does not support `char` or `logical` data types for `X`.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`fix` | `floor` | `round`

Topics

“Integers”
“Floating-Point Numbers”

Introduced before R2006a

cell

Cell array

Description

A *cell array* is a data type with indexed data containers called *cells*, where each cell can contain any type of data. Cell arrays commonly contain either lists of text, combinations of text and numbers, or numeric arrays of different sizes. Refer to sets of cells by enclosing indices in smooth parentheses, `()`. Access the contents of cells by indexing with curly braces, `{}`.

Creation

When you have data to put into a cell array, create the array using the cell array construction operator, `{}`.

```
C = {1,2,3;
     'text',rand(5,10,2),{11; 22; 33}}
```

```
C=2×3 cell
    {[   1]}    {[           2]}    {[     3]}
    {'text'}    {5x10x2 double}    {3x1 cell}
```

You also can use `{}` to create an empty 0-by-0 cell array.

```
C = {}
```

```
C =

  0x0 empty cell array
```

To create a cell array with a specified size, use the `cell` function, described below.

You can use `cell` to preallocate a cell array to which you assign data later. `cell` also converts certain types of Java, .NET, and Python® data structures to cell arrays of equivalent MATLAB objects.

Syntax

```
C = cell(n)
C = cell(sz1,...,szN)
C = cell(sz)

D = cell(obj)
```

Description

`C = cell(n)` returns an `n`-by-`n` cell array of empty matrices.

`C = cell(sz1,...,szN)` returns a `sz1`-by-...-by-`szN` cell array of empty matrices where `sz1,...,szN` indicate the size of each dimension. For example, `cell(2,3)` returns a 2-by-3 cell array.

`C = cell(sz)` returns a cell array of empty matrices where size vector `sz` defines `size(C)`. For example, `cell([2 3])` returns a 2-by-3 cell array.

`D = cell(obj)` converts a Java array, .NET `System.String` or `System.Object` array, or Python sequence into a MATLAB cell array.

Input Arguments

`n` — Size of square cell array

integer value

Size of a square cell array, specified as an integer value.

- If `n` is `0`, then `C` is an empty cell array.
- If `n` is negative, then it is treated as `0`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz1,...,szN` — Sizes of dimensions

integer values

Sizes of the dimensions of the cell array, specified as integer values.

- If the size of any dimension is `0`, then `C` is an empty cell array.
- If the size of any dimension is negative, then it is treated as `0`.
- Beyond the second dimension, `cell` ignores trailing dimensions with a size of 1. For example, `cell(3,1,1,1)` produces a 3-by-1 cell array of empty matrices.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz` — Size
row vector of integers

Size, specified as a row vector of integers. Each element of `sz` indicates the size of the corresponding dimension.

- If the size of any dimension is `0`, then `C` is an empty cell array.
- If the size of any dimension is negative, then it is treated as `0`.
- Beyond the second dimension, `cell` ignores trailing dimensions with a size of 1. For example, `cell([3 1 1 1])` produces a 3-by-1 cell array of empty matrices.

Example: `sz = [2 3 4]` creates a 2-by-3-by-4 cell array of empty matrices.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`obj` — Input array
Java array or object | .NET array of type `System.String` or `System.Object` | Python sequence type

Input array, specified as:

- Java array or object
- .NET array of type `System.String` or `System.Object`
- Python sequence type

Output Arguments

`C` — Output array
cell array

Output array, returned as a cell array. Each cell contains an empty, 0-by-0 array of type `double`.

D — Converted array
cell array

Converted array, returned as a cell array.

Each cell contains a MATLAB object that has a type closest to the corresponding Java, .NET, or Python type. For more information, see:

- "Handle Data Returned from Java Methods"
- "Handle Data Returned from Python"
- ".NET Type to MATLAB Type Mapping"

Examples

Create Cell Array

When related pieces of data have different data types, you can keep them together in a cell array. Each cell contains a piece of data. To refer to elements of a cell array, use array indexing. You can index into a cell array using smooth parentheses, `()`, and into the contents of cells using curly braces, `{}`.

Create a cell array that contains several temperature readings taken on a given date. Specify a date as a character vector, and temperatures as an array of doubles. To store these pieces of data in a cell array, enclose them in curly braces.

```
C = {'2017-08-16',[56 67 78]}
```

```
C=1×2 cell
    {'2017-08-16'}    {1x3 double}
```

Add readings for different dates to the cell array. One way to add more cells is to expand the cell array by assignment, just as you can expand an ordinary array.

```
C(2,:) = {'2017-08-17',[58 69 79]};
C(3,:) = {'2017-08-18',[60 68 81]}
```

```
C=3×2 cell
    {'2017-08-16'}    {1x3 double}
    {'2017-08-17'}    {1x3 double}
```

```
    {'2017-08-18'}    {1x3 double}
```

Index into the first row of `C`. When you index with smooth parentheses, `()`, the result is a cell array that is a subset of the cell array.

```
C(1,:)
```

```
ans=1×2 cell
    {'2017-08-16'}    {1x3 double}
```

Index into the contents of a cell. When you index with curly braces, `{}`, the result is the piece of data that is contained in the specified cell.

```
C{1,2}
```

```
ans = 1×3

    56    67    78
```

Square Cell Array

Create a 3-by-3 cell array of empty matrices.

```
C = cell(3)
```

```
C=3×3 cell
    {0x0 double}    {0x0 double}    {0x0 double}
    {0x0 double}    {0x0 double}    {0x0 double}
    {0x0 double}    {0x0 double}    {0x0 double}
```

3-D Cell Array

Create a 3-by-4-by-2 cell array of empty matrices.

```
C = cell(3,4,2);
size(C)
```

```
ans = 1×3

     3     4     2
```

Clone Size from Existing Array

Create a cell array of empty matrices that is the same size as an existing array.

```
A = [7 9; 2 1; 8 3];
sz = size(A);
C = cell(sz)
```

```
C=3×2 cell
    {0x0 double}    {0x0 double}
    {0x0 double}    {0x0 double}
    {0x0 double}    {0x0 double}
```

It is a common pattern to combine the previous two lines of code into a single line.

```
C = cell(size(A));
```

Tips

- Creating a cell array of empty matrices with the `cell` function is equivalent to assigning an empty matrix to the last index of a new cell array. For example, these two statements are equivalent:

  ```
  C = cell(3,4,2);
  C{3,4,2} = [];
  ```

Compatibility Considerations

Cell array expansion is consistent with general array expansion

Behavior changed in R2019a

Starting in R2019a, the dimensions of an expanded cell array are consistent whether you use curly braces or parentheses for indices. Previously, the output dimensions were different when you did not specify indices for all dimensions. Indexing with curly braces now matches the previous behavior for indexing with parentheses, which is consistent with general array expansion. For example:

- If `C` is 1-by-2-by-3-by-4, then `C(5,2)={3}` and `C{5,2}=3` both expand `C` to 5-by-2-by-3-by-4. Previously, `C{5,2}=3` resulted in a 5-by-24 array.
- If `C` is 0-by-5 or 5-by-0, then `C(3)={2}` and `C{3}=2` expand `C` to 1-by-3. Previously, `C{3}=2` resulted in a 1-by-5 array when `C` was 0-by-5 and in a 3-by-1 array when `C` was 5-by-0.
- If `C` is 0-by-2-by-3, then `C(3)={2}` and `C{3}=2` produce errors because it is ambiguous which dimension to expand. Previously, `C{3}=2` resulted in a 1-by-6 array.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “Cell Array Limitations for Code Generation” (MATLAB Coder).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `distributed.cell` in the Parallel Computing Toolbox documentation.

See Also

`cell2mat` | `cell2struct` | `cell2table` | `iscell` | `mat2cell` | `num2cell` | `struct2cell` | `table2cell`

Topics

"Create Cell Array"
"Preallocate Memory for Cell Array"
"Access Data in Cell Array"
"Cell vs. Structure Arrays"

Introduced before R2006a

cell2mat

Convert cell array to ordinary array of the underlying data type

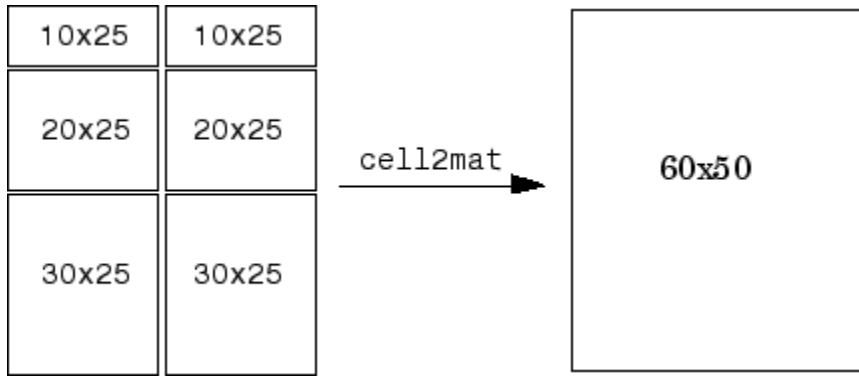
Syntax

```
A = cell2mat(C)
```

Description

`A = cell2mat(C)` converts a cell array into an ordinary array. The elements of the cell array must all contain the same data type, and the resulting array is of that data type.

The contents of `C` must support concatenation into an N-dimensional rectangle. Otherwise, the results are undefined. For example, the contents of cells in the same column must have the same number of columns, although they need not have the same number of rows (see figure).



Examples

Convert Cell Array to Numeric Array

Convert numeric arrays in four cells of a cell array into one numeric array.

```
C = {[1],    [2 3 4];
     [5; 9], [6 7 8; 10 11 12]}
```

```
C=2×2 cell
    {[       1]}    {1x3 double}
    {2x1 double}    {2x3 double}
```

```
A = cell2mat(C)
```

```
A = 3×4

     1     2     3     4
     5     6     7     8
     9    10    11    12
```

Convert Cell Array of Structures to Array

Convert structures in a cell array into one structure array. The structures must have the same fields.

```
s1.a = [1 2 3 4];
s1.b = 'Good';
s2.a = [5 6; 7 8];
s2.b = 'Morning';
c = {s1,s2};
d = cell2mat(c)
```

```
d=2×2 struct
    a
    b
```

Display the first field of structure `d(1)`.

```
d(1).a
```

```
ans = 1×4
```

```
     1     2     3     4
```

Display the second field of `d(2)`.

```
d(2).b
```

```
ans =
'Morning'
```

Input Arguments

`C` — Input cell array
cell array

Input cell array, in which all cells contain the same data type. `cell2mat` accepts numeric, logical, or character data within cells of `C`, or structures with the same field names and data types. `cell2mat` does not accept objects or nested cells within `C`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cell` | `cell2struct` | `cell2table` | `iscell` | `mat2cell` | `num2cell` | `struct2cell` | `table2cell`

Introduced before R2006a

cell2struct

Convert cell array to structure array

Syntax

```
structArray = cell2struct(cellArray, fields, dim)
```

Description

`structArray = cell2struct(cellArray, fields, dim)` creates a structure array, `structArray`, from the information contained within cell array `cellArray`.

The `fields` argument specifies field names for the structure array. This argument is a character array, a cell array of character vectors, or a string array.

The `dim` argument tells MATLAB which axis of the cell array to use in creating the structure array. Use a numeric `double` to specify `dim`.

To create a structure array with fields derived from `N` rows of a cell array, specify `N` field names in the `fields` argument, and the number 1 in the `dim` argument. To create a structure array with fields derived from `M` columns of a cell array, specify `M` field names in the `fields` argument and the number 2 in the `dim` argument.

The `structArray` output is a structure array with `N` fields, where `N` is equal to the number of fields in the `fields` input argument. The number of fields in the resulting structure must equal the number of cells along dimension `dim` that you want to convert.

Examples

Create the following table for use with the examples in this section. The table lists information about the employees of a small Engineering company. Reading the table by rows shows the names of employees by department. Reading the table by columns shows the number of years each employee has worked at the company.

	5 Years	10 Years	15 Years
Development	Lee, Reed, Hill	Dean, Frye	Lane, Fox, King
Sales	Howe, Burns	Kirby, Ford	Hall
Management	Price	Clark, Shea	Sims
Quality	Bates, Gray	Nash	Kay, Chase
Documentation	Lloyd, Young	Ryan, Hart, Roy	Marsh

Enter the following commands to create the initial cell array `employees`:

```
devel = {{'Lee','Reed','Hill'}, {'Dean','Frye'}, ...
   {'Lane','Fox','King'}};
sales = {{'Howe','Burns'}, {'Kirby','Ford'}, {'Hall'}};
mgmt = {{'Price'}, {'Clark','Shea'}, {'Sims'}};
qual = {{'Bates','Gray'}, {'Nash'}, {'Kay','Chase'}};
docu = {{'Lloyd','Young'}, {'Ryan','Hart','Roy'}, {'Marsh'}};

employees = [devel; sales; mgmt; qual; docu]
employees =

    {1x3 cell}    {1x2 cell}    {1x3 cell}
    {1x2 cell}    {1x2 cell}    {1x1 cell}
    {1x1 cell}    {1x2 cell}    {1x1 cell}
    {1x2 cell}    {1x1 cell}    {1x2 cell}
    {1x2 cell}    {1x3 cell}    {1x1 cell}
```

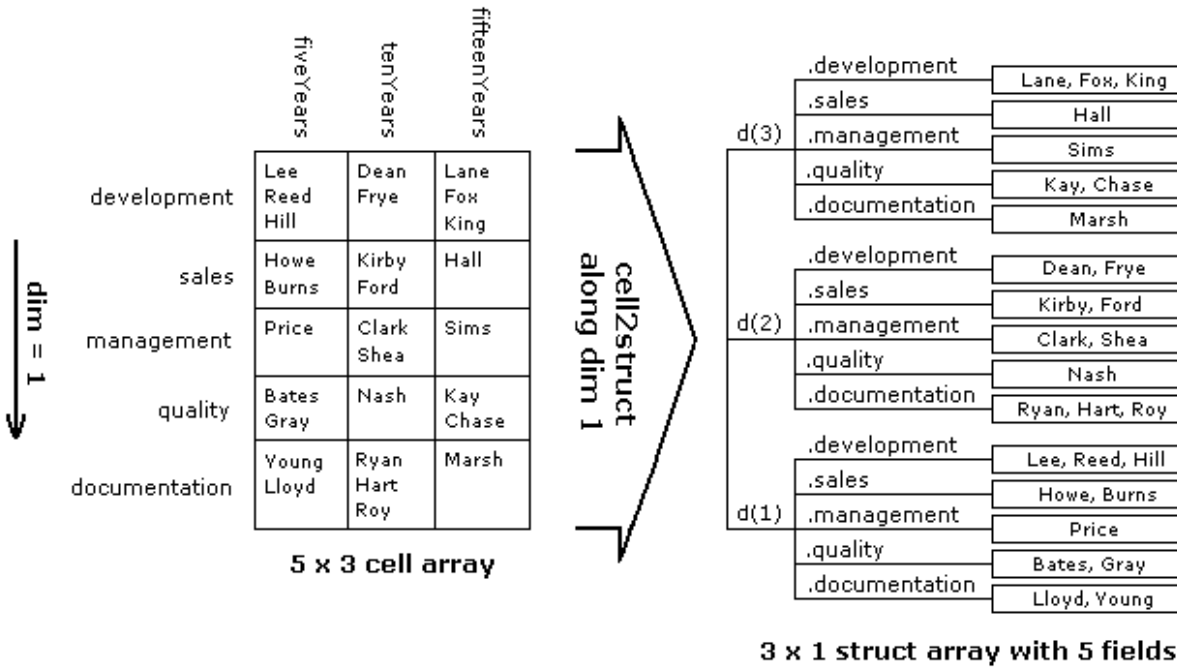
This is the resulting cell array:

	fiveYears	tenYears	fifteenYears
development	Lee Reed Hill	Dean Frye	Lane Fox King
sales	Howe Burns	Kirby Ford	Hall
management	Price	Clark Shea	Sims
quality	Bates Gray	Nash	Kay Chase
documentation	Young Lloyd	Ryan Hart Roy	Marsh

5 x 3 cell array

Convert the cell array to a struct along dimension 1:

1. Convert the 5-by-3 cell array along its first dimension to construct a 3-by-1 struct array with 5 fields. Each of the rows along dimension 1 of the cell array becomes a field in the struct array:



Traversing the first (i.e., vertical) dimension, there are 5 rows with row headings that read as follows:

```
rowHeadings = {'development', 'sales', 'management', ...
   'quality', 'documentation'};
```

2 Convert the cell array to a struct array, `depts`, in reference to this dimension:

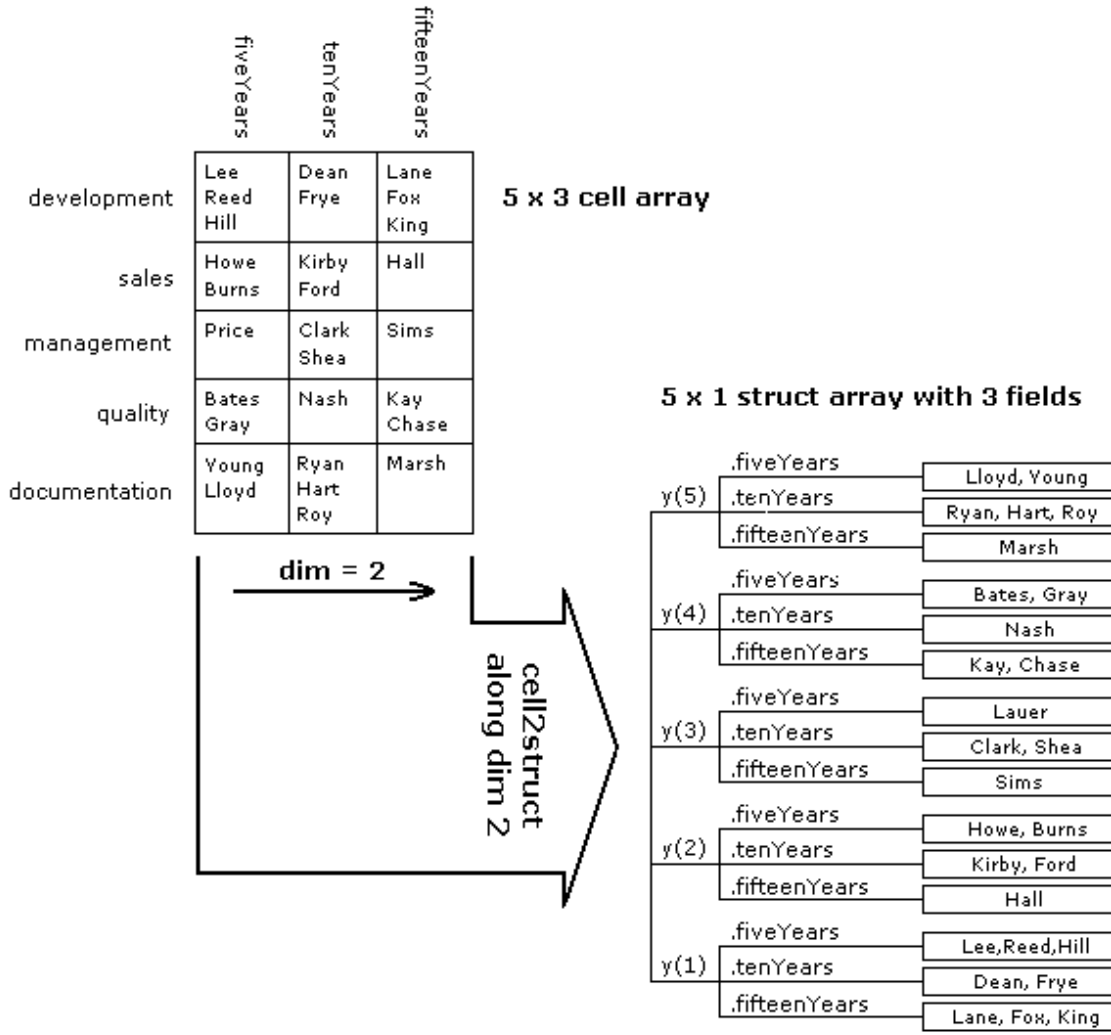
```
depts = cell2struct(employees, rowHeadings, 1)
depts =
3x1 struct array with fields:
    development
    sales
    management
    quality
    documentation
```

3 Use this row-oriented structure to find the names of the Development staff who have been with the company for up to 10 years:

```
depts(1:2).development
ans =
    'Lee'    'Reed'    'Hill'
ans =
    'Dean'    'Frye'
```

Convert the same cell array to a struct along dimension 2:

1. Convert the 5-by-3 cell array along its second dimension to construct a 5-by-1 struct array with 3 fields. Each of the columns along dimension 2 of the cell array becomes a field in the struct array:



2 Traverse the cell array along the second (or horizontal) dimension. The column headings become fields of the resulting structure:

```
colHeadings = {'fiveYears' 'tenYears' 'fifteenYears'};

years = cell2struct(employees, colHeadings, 2)
```

```
years =
5x1 struct array with fields:
    fiveYears
    tenYears
    fifteenYears
```

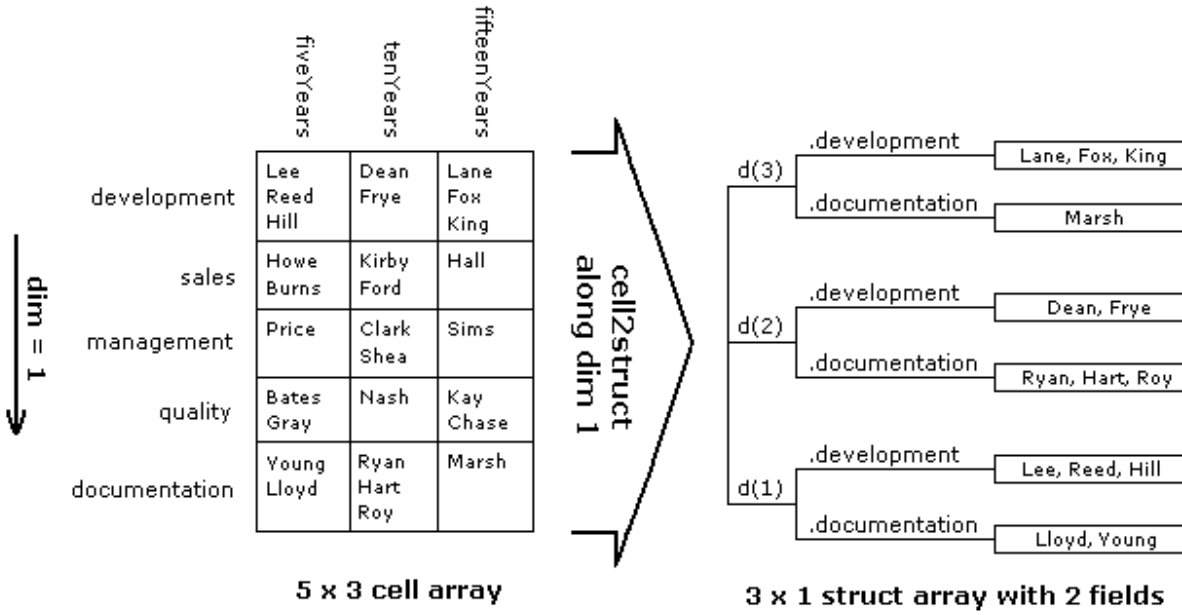
3 Using the column-oriented structure, show how many employees from the Sales and Documentation departments have worked for the company for at least 5 years:

```
[~, sales_5years, ~, ~, docu_5years] = years.fiveYears
sales_5years =
    'Howe'    'Burns'
docu_5years =
    'Lloyd'    'Young'
```

Convert only part of the cell array to a struct:

1 Convert only the first and last rows of the cell array. This results in a 3-by-1 struct array with 2 fields:

```
rowHeadings = {'development', 'documentation'};

depts = cell2struct(employees([1,5],:), rowHeadings, 1)
depts =
3x1 struct array with fields:
    development
    documentation
```



2 Display those employees who belong to these departments for all three periods of time:

```
for k=1:3
   depts(k,:)
end

ans =
      development: {'Lee'  'Reed'  'Hill'}
    documentation: {'Lloyd'  'Young'}
ans =
      development: {'Dean'  'Frye'}
    documentation: {'Ryan'  'Hart'  'Roy'}
ans =
      development: {'Lane'  'Fox'  'King'}
    documentation: {'Marsh'}
```

Extended Capabilities

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cell` | `cell2table` | `fieldnames` | `iscell` | `isstruct` | `struct` | `struct2cell` | `table2struct`

Topics

“Generate Field Names from Variables”

Introduced before R2006a

cell2table

Convert cell array to table

Syntax

```
T = cell2table(C)
T = cell2table(C,Name,Value)
```

Description

`T = cell2table(C)` converts the contents of an `m`-by-`n` cell array, `C`, to an `m`-by-`n` table, `T`. Each column of `C` provides the data contained in a variable of `T`.

To create variable names in the output table, `cell2table` appends column numbers to the input array name. If the input array has no name, then `cell2table` creates variable names of the form `'Var1',...,'Var`*`N`*`'`, where *`N`* is the number of columns in `C`.

`T = cell2table(C,Name,Value)` creates a table from a cell array, `C`, with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify row names or variable names to include in the table.

Examples

Convert Cell Array to Table

Create a cell array containing character vectors and numeric data.

```
C = {5 'cereal' 110 'C+'; 12 'pizza' 140 'B';...
    23 'salmon' 367 'A'; 2 'cookies' 160 'D'}
```

```
C=4×4 cell
    {[ 5]}    {'cereal' }    {[110]}    {'C+'}
    {[12]}    {'pizza'  }    {[140]}    {'B' }
```

```
    {[23]}    {'salmon' }    {[367]}    {'A' }
    {[ 2]}    {'cookies'}    {[160]}    {'D' }
```

Convert the cell array, `C`, to a table and specify variable names.

```
T = cell2table(C,...
    'VariableNames',{'Age' 'FavoriteFood' 'Calories' 'NutritionGrade'})
```

```
T=4×4 table
    Age    FavoriteFood    Calories    NutritionGrade
    ___    ____________    ________    ______________

     5     {'cereal' }       110           {'C+'}
    12     {'pizza'  }       140           {'B' }
    23     {'salmon' }       367           {'A' }
     2     {'cookies'}       160           {'D' }
```

The variables `T.Age` and `T.Calories` are numeric while the variables `T.FavoriteFood` and `T.NutritionGrade` are cell arrays of character vectors.

Convert Column Headings to Variable Names

Convert a cell array to a table, and then include the first row from the cell array as variable names for the table.

Create a cell array where the first row contains character vectors to identify column headings.

```
Patients = {'Gender' 'Age' 'Height' 'Weight' 'Smoker';...
    'M' 38 71 176 true;...
    'M' 43 69 163 false;...
    'M' 38 64 131 false;...
    'F' 38 64 131 false;...
    'F' 40 67 133 false;...
    'F' 49 64 119 false}
```

```
Patients=7×5 cell
    {'Gender'}    {'Age'}    {'Height'}    {'Weight'}    {'Smoker'}
    {'M'     }    {[ 38]}    {[    71]}    {[   176]}    {[     1]}
    {'M'     }    {[ 43]}    {[    69]}    {[   163]}    {[     0]}
```

```
    {'M'     }    {[ 38]}    {[    64]}    {[   131]}    {[     0]}
    {'F'     }    {[ 38]}    {[    64]}    {[   131]}    {[     0]}
    {'F'     }    {[ 40]}    {[    67]}    {[   133]}    {[     0]}
    {'F'     }    {[ 49]}    {[    64]}    {[   119]}    {[     0]}
```

Exclude the columns headings and convert the contents of the cell array to a table.

```
C = Patients(2:end,:);
T = cell2table(C)
```

```
T=6×5 table
     C1      C2    C3    C4      C5
    _____    __    __    ___    _____

    {'M'}    38    71    176    true
    {'M'}    43    69    163    false
    {'M'}    38    64    131    false
    {'F'}    38    64    131    false
    {'F'}    40    67    133    false
    {'F'}    49    64    119    false
```

The table, `T`, has variable names `C1,...,C5`.

Change the variable names by setting the table property, `T.Properties.VariableNames`, to the first row of the cell array.

```
T.Properties.VariableNames = Patients(1,:)
```

```
T=6×5 table
    Gender    Age    Height    Weight    Smoker
    ______    ___    ______    ______    ______

    {'M'}     38       71       176      true
    {'M'}     43       69       163      false
    {'M'}     38       64       131      false
    {'F'}     38       64       131      false
    {'F'}     40       67       133      false
    {'F'}     49       64       119      false
```

Input Arguments

C — Input cell array

2-D cell array

Input cell array, specified as a 2-D cell array. Each column of C provides data for a table variable.

- If the contents of the cells in a column of C have compatible sizes and types, then the corresponding table variable is the vertical concatenation of those contents into an array.
- If the contents of the cells in a column have different sizes and types, then the corresponding table variable is a cell array.
  - If the contents of the cells in a column are all character vectors, then the corresponding table variable is a cell array of character vectors.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'RowNames',{'row1','row2','row3'} uses the row names, row1, row2, and row3 for the table, T.

RowNames — Row names for T

{} (default) | cell array of character vectors | string array

Row names for T, specified as the comma-separated pair consisting of 'RowNames' and a cell array of character vectors or a string array, whose elements are nonempty and distinct. The number of names must equal the number of rows, size(C,1).

VariableNames — Variable names for T

cell array of character vectors | string array

Variable names for T, specified as the comma-separated pair consisting of 'VariableNames' and a cell array of character vectors or a string array, whose elements are nonempty and distinct. The number of names must equal the number of variables, size(C,2).

Variable names can have any Unicode characters, including spaces and non-ASCII characters.

Output Arguments

T — Output table
table

Output table, returned as a table. The table can store metadata such as descriptions, variable units, variable names, and row names. For more information, see the Properties section of `table`.

Compatibility Considerations

Variable names can contain leading and trailing whitespace characters

Behavior changed in R2019b

Table and timetable variable names with leading or trailing whitespace characters are not modified.

In previous releases, leading and trailing whitespace characters were deleted from variable names when you specified them using the `'VariableNames'` name-value pair argument, or assigned them to the `VariableNames` property.

To manually remove such characters, first use the `strtrim` function on the names, then assign them as variable names to the table or timetable.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- In generated code, you must specify the `'VariableNames'` name-value pair argument when using this function. For more information, see “Code Generation for Tables” (MATLAB Coder) and “Table Limitations for Code Generation” (MATLAB Coder).

See Also

`array2table` | `isvarname` | `struct2table` | `table` | `table2cell`

Topics

“Access Data in Tables”

Introduced in R2013b

celldisp

Display cell array contents

Syntax

```
celldisp(C)
celldisp(C,displayName)
```

Description

`celldisp(C)` recursively displays the contents of a cell array.

The `celldisp` function also displays the name of the cell array. If there is no name to display, then `celldisp` displays `ans` instead. For example, if `C` is an expression that creates an array, then there is no name to display.

`celldisp(C,displayName)` uses the specified display name instead of the default name described in the previous syntax.

Examples

Display Contents of Each Cell

Create a cell array.

```
C = {'row1',[1 2 3],3+4i;
     'row2',[2 4;1 3],{'innercells',42}}
```

```
C=2×3 cell
    {'row1'}    {1x3 double}    {[3.0000 + 4.0000i]}
    {'row2'}    {2x2 double}    {1x2 cell          }
```

Display the contents of each cell using the `celldisp` function. `celldisp` also displays the curly-brace indexing you can use to index into cells.

```
celldisp(C)
```

```
C{1,1} =

row1


C{2,1} =

row2


C{1,2} =

     1     2     3



C{2,2} =

     2     4
     1     3



C{1,3} =

   3.0000 + 4.0000i



C{2,3}{1} =

innercells


C{2,3}{2} =

    42
```

The last cell contains a cell array. `celldisp` also displays the contents of each cell from the inner cell array.

Display Different Names

Call `celldisp` and specify a cell array as its first input argument. Since the first argument is not a workspace variable, and so does not have a name of its own, specify a name as the second argument. `celldisp` displays the cell array using this name.

```
celldisp({'row1',5,10},'myCells')
```

```
myCells{1} =

row1


myCells{2} =

     5



myCells{3} =

    10
```

Call `celldisp` without specifying a name. When there is no variable name or second input argument, `celldisp` displays `ans` as the name of the cell array.

```
celldisp({'row1',5,10})
```

```
ans{1} =

row1


ans{2} =

     5
```

```
ans{3} =

    10
```

Input Arguments

`C` — Input cell array
cell array

Input cell array.

`displayName` — Displayed name of cell array
character vector | string scalar

Displayed name of the cell array, specified as a character vector or string scalar.

See Also

`cellplot`

Topics

"Export Cell Array to Text File"

Introduced before R2006a

cellfun

Apply function to each cell in cell array

Syntax

```
A = cellfun(func,C)
A = cellfun(func,C1,...,Cn)
A = cellfun( ___ ,Name,Value)
[A1,...,Am] = cellfun( ___ )
```

Description

`A = cellfun(func,C)` applies the function `func` to the contents of each cell of cell array `C`, one cell at a time. `cellfun` then concatenates the outputs from `func` into the output array `A`, so that for the `i`th element of `C`, `A(i) = func(C{i})`. The input argument `func` is a function handle to a function that takes one input argument and returns a scalar. The output from `func` can have any data type, so long as objects of that type can be concatenated. The array `A` and cell array `C` have the same size.

You cannot specify the order in which `cellfun` calculates the elements of `A` or rely on them being done in any particular order.

`A = cellfun(func,C1,...,Cn)` applies `func` to the contents of the cells of `C1,...,Cn`, so that `A(i) = func(C1{i},...,Cn{i})`. The function `func` must take `n` input arguments and return a scalar. The cell arrays `C1,...,Cn` all must have the same size.

`A = cellfun( ___ ,Name,Value)` applies `func` with additional options specified by one or more `Name,Value` pair arguments. For example, to return output values in a cell array, specify `'UniformOutput',false`. You can return `A` as a cell array when `func` returns values that cannot be concatenated into an array. You can use `Name,Value` pair arguments with the input arguments of either of the previous syntaxes.

`[A1,...,Am] = cellfun( ___ )` returns multiple output arrays `A1,...,Am` when `func` returns `m` output values. `func` can return output arguments that have different data types,

but the data type of each output must be the same each time `func` is called. You can use this syntax with any of the input arguments of the previous syntaxes.

The number of output arguments from `func` need not be the same as the number of input arguments specified by `C1,...,Cn`.

Examples

Apply Function to Contents of Cell Array

Create a cell array that contains numeric arrays of different sizes.

```
C = {1:10, [2; 4; 6], []}
```

```
C=1×3 cell
    {1x10 double}    {3x1 double}    {0x0 double}
```

Calculate the mean of each numeric array, and return the means in an array.

```
A = cellfun(@mean,C)
```

```
A = 1×3

    5.5000    4.0000       NaN
```

Return Object Array

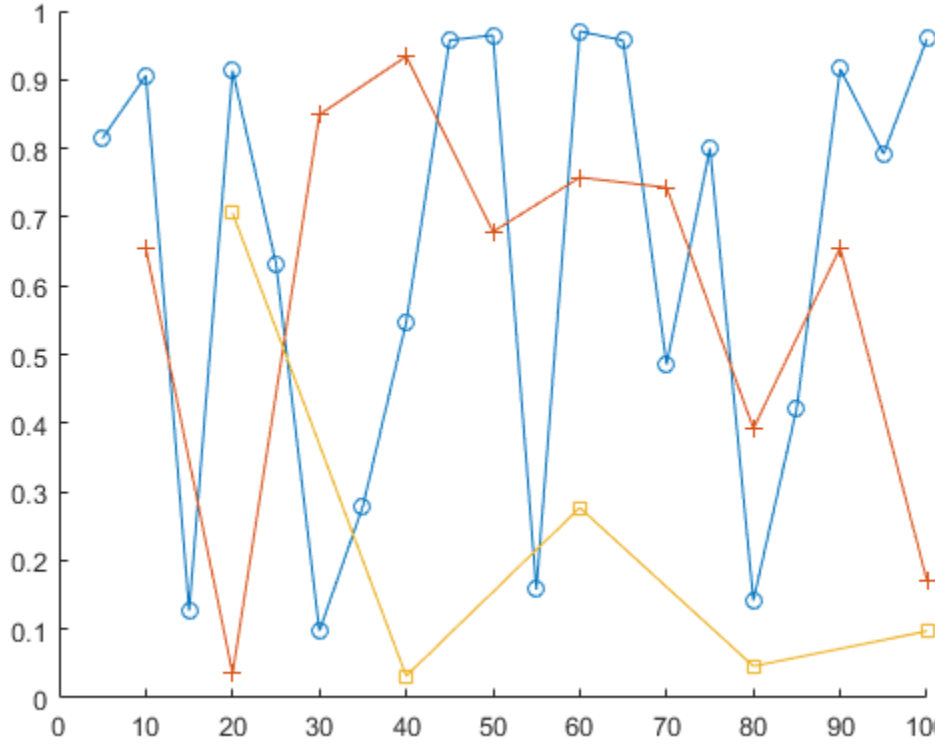
Create two cell arrays that contain numeric arrays of different sizes.

```
X = {5:5:100, 10:10:100, 20:20:100};
Y = {rand(1,20), rand(1,10), rand(1,5)};
```

Plot the arrays. Return an array of chart line objects from the `plot` function and use them to add different markers to each set of data points. `cellfun` can return arrays of any data type, so long as objects of that data type can be concatenated.

```
figure
hold on
```

```
p = cellfun(@plot,X,Y);
p(1).Marker = 'o';
p(2).Marker = '+';
p(3).Marker = 's';
hold off
```



Return Multiple Output Arrays

Create a cell array that contains numeric arrays of different sizes.

```
C = {1:10, [2; 4; 6], []}
```

```
C=1×3 cell
    {1x10 double}    {3x1 double}    {0x0 double}
```

Calculate the sizes of each array in `C`. The number of rows and columns are each in 1-by-3 numeric arrays.

```
[nrows,ncols] = cellfun(@size,C)
```

```
nrows = 1×3

     1     3     0


ncols = 1×3

    10     1     0
```

Apply Function to Characters in Cell or String Array

You can use `cellfun` to apply functions to cell arrays of character vectors and to string arrays. `cellfun` treats the two kinds of arrays identically.

Create a cell array of character vectors that contains weekday names.

```
C = {'Monday','Tuesday','Wednesday','Thursday','Friday'}
```

```
C = 1x5 cell array
    {'Monday'}    {'Tuesday'}    {'Wednesday'}    {'Thursday'}    {'Friday'}
```

Create three-letter abbreviations for the names using the `cellfun` function. Specify a function that extracts the first three characters and returns them as a character vector. To return the abbreviations in a cell array, specify the `'UniformOutput',false` name-value pair.

```
A = cellfun(@(x) x(1:3),C,'UniformOutput',false)
```

```
A = 1x5 cell array
    {'Mon'}    {'Tue'}    {'Wed'}    {'Thu'}    {'Fri'}
```

You also can call `cellfun` on a string array. For compatibility, `cellfun` treats each element of a string array as though it were a character vector. If you specify a function that returns text, then `cellfun` returns it as a cell array of character vectors, not as a string array.

Create abbreviations for names in a string array using `cellfun`.

```
str = ["Saturday","Sunday"]
```

```
str = 1x2 string array
    "Saturday"    "Sunday"
```

```
B = cellfun(@(x) x(1:3),str,'UniformOutput',false)
```

```
B = 1x2 cell array
    {'Sat'}    {'Sun'}
```

Input Arguments

`func` — Function to apply

function handle | character vector | string scalar

Function to apply to the contents of the cells of the input cell arrays, specified as a function handle, character vector, or string scalar.

`func` can correspond to more than one function file and therefore can represent a set of overloaded functions. In these cases, MATLAB determines which function to call based on the class of the input arguments.

Backward Compatibility

You can specify `func` as a character vector or string scalar, rather than a function handle, but only for a limited set of function names. `func` can be: `'isempty'`, `'islogical'`, `'isreal'`, `'length'`, `'ndims'`, `'prodofsize'`, `'size'`, or `'isclass'`.

If you specify a function name rather than a function handle:

- `cellfun` does not call any overloaded versions of the function.

- The `size` and `isclass` functions require additional inputs to the `cellfun` function:

  `A = cellfun('size',C,k)` returns the size along the `k`th dimension of each element of `C`.

  `A = cellfun('isclass',C,classname)` returns logical `1` (`true`) for each element of `C` that matches the `classname` argument. This syntax returns logical 0 (`false`) for objects that are a subclass of `classname`.

Example: `A = cellfun(@mean,C)` returns the means of the elements of `C`.

`C` — Input array
cell array | string array

Input array, specified as a cell array or a string array. If `C` is a string array, then `cellfun` treats each element of `C` as though it were a character vector, not a string.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `A = cellfun(@mean,C,'UniformOutput',false)` returns the outputs from `mean` in a cell array. Use the `'UniformOutput',false` name-value pair if `C` contains numeric matrices and `mean` returns vectors.

`UniformOutput` — True or false
`true` (default) | `false`

True or false, specified as the comma-separated pair consisting of `'UniformOutput'` and either `true` (`1`) or `false` (`0`).

Value of `'UniformOutput'`	Description
`true` (`1`)	`func` must return scalars that `cellfun` concatenates into arrays.
`false` (`0`)	`cellfun` returns the outputs of `func` in cell arrays. The outputs of `func` can have any sizes and different data types.

`ErrorHandler` — Function to catch errors
function handle

Function to catch errors, specified as the comma-separated pair consisting of `'ErrorHandler'` and a function handle. If `func` throws an error, then the error handler specified by `'ErrorHandler'` catches the error and takes the action specified in the function. The error handler either must throw an error or return the same number of outputs as `func`. If the value of `'UniformOutput'` is true, then the output arguments of the error handler must be scalars and have the same data type as the outputs of `func`.

The first input argument of the error handler is a structure with these fields:

- `identifier` — Error identifier
- `message` — Error message text
- `index` — Linear index into the input arrays at which `func` threw the error

The remaining input arguments to the error handler are the input arguments for the call to `func` that made `func` throw the error.

Suppose `func` returns two doubles as output arguments. You can specify the error handler as `'ErrorHandler',@errorFunc`, where `errorFunc` is a function that raises a warning and returns two output arguments.

```
function [A,B] = errorFunc(S,varargin)
    warning(S.identifier, S.message);
    A = NaN;
    B = NaN;
end
```

If you do not specify `'ErrorHandler'`, then `cellfun` rethrows the error thrown by `func`.

Output Arguments

`A` — Output array
array of any data type | cell array

Output array, returned as an array of any data type or as a cell array.

By default, `cellfun` concatenates the outputs from `func` into an array. `func` must return scalars. If `func` returns objects, then the class that the objects belong to must meet these requirements.

- Support assignment by linear indexing into the object array
- Have a `reshape` method that returns an array that has the same size as the input

If the value of the `'UniformOutput'` name-value pair argument is `false` (`0`), then `cellfun` returns outputs in a cell array. In that case, the outputs from `func` can have any sizes and different data types.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with these limitations:

- The input function must be a function handle.
- The input function must not rely on `persistent` variables.
- The `'ErrorHandler'` name-value pair is not supported.
- With the `'UniformOutput'` name-value pair set to `true` (default), the outputs from the specified function must be numeric, logical, characters, or cell arrays.

For more information, see “Tall Arrays for Out-of-Memory Data”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`arrayfun` | `cell2mat` | `spfun` | `splitapply` | `structfun`

Topics

"Anonymous Functions"
"Create Function Handle"

Introduced before R2006a

cellplot

Graphically display structure of cell array

Syntax

```
cellplot(c)
cellplot(c, 'legend')
handles = cellplot(c)
```

Description

`cellplot(c)` displays a figure window that graphically represents the contents of `c`. Filled rectangles represent elements of vectors and arrays, while scalars and short character vectors are displayed as text.

`cellplot(c, 'legend')` places a colorbar next to the plot labelled to identify the data types in `c`.

`handles = cellplot(c)` displays a figure window and returns a vector of surface handles.

Limitations

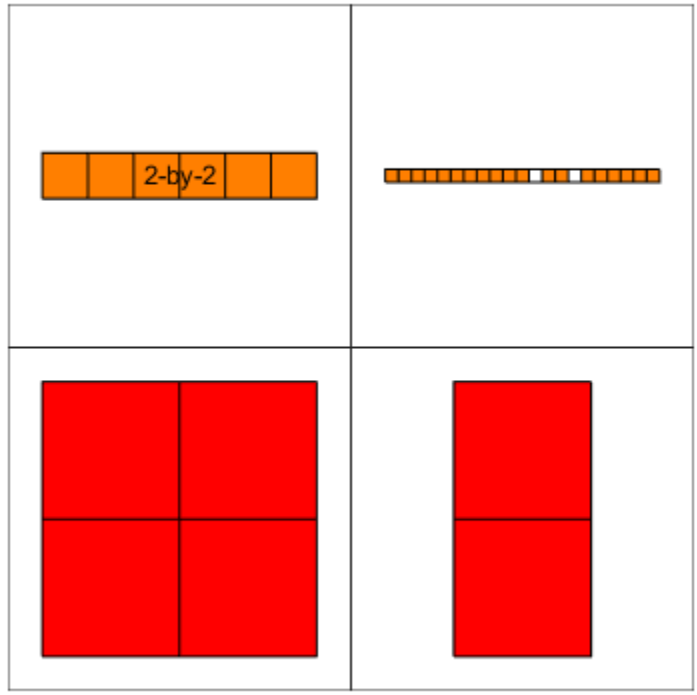
The `cellplot` function can display only two-dimensional cell arrays.

Examples

Consider a 2-by-2 cell array containing a matrix, a vector, and two character vectors:

```
c{1,1} = '2-by-2';
c{1,2} = 'eigenvalues of eye(2)';
c{2,1} = eye(2);
c{2,2} = eig(eye(2));
```

The command `cellplot(c)` produces



See Also

`celldisp`

Topics

"Export Cell Array to Text File"

Introduced before R2006a

cellstr

Convert to cell array of character vectors

Syntax

```
C = cellstr(A)
```

```
C = cellstr(D)
C = cellstr(D,fmt)
C = cellstr(D,fmt,locale)
```

Description

`C = cellstr(A)` converts `A` to a cell array of character vectors. The input array `A` can be a character array, a categorical array, or, starting in R2016b, a string array.

`C = cellstr(D)` converts a datetime, duration, or calendar duration array into a cell array of character vectors in the format specified by the `Format` property of `D`. The output has the same dimensions as `D`.

`C = cellstr(D,fmt)` represents dates or durations in the specified format. For example, `cellstr(D,'HH:mm:ss')` represents the times associated with each element of `D`.

`C = cellstr(D,fmt,locale)` represents dates or durations in the specified locale. For example, `cellstr(D,'dd-MMM-yyyy','en_US')` represents the dates associated with each element of `D` using the `en_US` locale. The locale affects the language used to represent character vectors such as month and day names.

Examples

Convert String Array to Cell Array

You can create string arrays to contain multiple pieces of text. However, you might need to use functions that accept cell arrays of character vectors as input arguments, and that

do not accept string arrays. To pass data from a string array to such functions, use the `cellstr` function to convert the string array to a cell array of character vectors.

Create a string array. Starting in R2017a, you can create strings using double quotes.

```
A = ["Past","Present","Future"]
```

```
A = 1x3 string array
    "Past"    "Present"    "Future"
```

Convert the string array to a 1-by-3 cell array of character vectors.

```
C = cellstr(A)
```

```
C = 1x3 cell array
    {'Past'}    {'Present'}    {'Future'}
```

Convert Character Array to Cell Array

Create a character array. Include trailing spaces so that each row has the same length, resulting in a 3-by-4 array.

```
A = ['abc ';'defg';'hi  ']
```

```
A = 3x4 char array
    'abc '
    'defg'
    'hi  '
```

```
class(A)
```

```
ans =
'char'
```

Convert the character array to a 3-by-1 cell array of character vectors.

```
C = cellstr(A)
```

```
C = 3x1 cell array
    {'abc' }
```

```
    {'defg'}
    {'hi'  }
```

```
class(C)
```

```
ans =
'cell'
```

Convert Calendar Duration Array to Cell Array

Create a `calendarDuration` array.

```
D = calmonths(15:17) + caldays(8) + hours(1.2345)
```

```
D = 1x3 calendarDuration array
   1y 3mo 8d 1h 14m 4.2s   1y 4mo 8d 1h 14m 4.2s   1y 5mo 8d 1h 14m 4.2s
```

Convert the array to a cell array of character vectors.

```
C = cellstr(D)
```

```
C = 1x3 cell array
    {'1y 3mo 8d 1h 14...'}    {'1y 4mo 8d 1h 14...'}    {'1y 5mo 8d 1h 14...'}
```

```
class(C)
```

```
ans =
'cell'
```

Input Arguments

A — Input array
character array | cell array of character vectors | categorical array | string array

Input array, specified as a character array, a cell array of character vectors, a categorical array, or a string array.

- If A is a character array, then each row of A is a cell of C. The cellstr function removes trailing whitespace characters in the rows of A, except for significant whitespace such as nonbreaking space characters. Use the char function to convert back into a character array.
- If A is a cell array of character vectors, then cellstr returns A unaltered.
- If A is a categorical array, then cellstr converts each element to a character vector and assigns it to a cell. The size of the cell array is the same as the size of A.
- Starting in R2016b, if A is a string array, then cellstr converts each element to a character vector and assigns it to a cell. The size of the cell array is the same as the size of A. Use the string function to convert back into a string array.
  - If any element of A is a missing string (displayed as <missing>), then cellstr assigns an empty character array to the corresponding cell of the output C.

Data Types: char | cell | categorical | string

D — Input date and time
date or duration array

Input date and time, specified as a date or duration array.

Data Types: datetime | duration | calendarDuration

fmt — Date and time format
[] (default) | character vector | string scalar

Date and time format, specified as [], a character vector, or a string scalar. If you specify [], then cellstr represents input D in the format specified by the Format property of D.

The supported formats depend on the data type of D.

- datetime formats can include combinations of units and delimiters, such as 'yyyy-MMM-dd HH:mm:ss.SSS'. For details, see the Format property for datetime arrays.
- duration formats are either single characters ('y', 'd', 'h', 'm', or 's') or one of these combinations:
  - 'dd:hh:mm:ss'
  - 'hh:mm:ss'
  - 'mm:ss'
  - 'hh:mm'

  - Any of the above, with up to nine `S` characters to indicate fractional second digits, such as `'hh:mm:ss.SSSS'`
- `calendarDuration` formats can include combinations of the characters `'y'`, `'q'`, `'m'`, `'w'`, `'d'`, and `'t'` in order from largest to smallest unit of time, such as `'ym'`.

For more information on the `duration` and `calendarDuration` formats, see “Set Date and Time Display Format”.

`locale` — Locale represented in output
character vector | string scalar

Locale represented in the output, specified as a character vector or a string scalar. The locale affects the language used to represent certain components of dates and times, such as month names.

`locale` can be:

- `'system'`, to specify your system locale.
- A character vector in the form *xx_YY*, where *xx* is a lowercase ISO 639-1 two-letter code that specifies a language, and *YY* is an uppercase ISO 3166-1 alpha-2 code that specifies a country.

The `locale` input argument can be any of the values accepted by the `'Locale'` name-value pair argument for the `datetime` function.

Example: `'en_US'`

Example: `'ja_JP'`

Algorithms

`cellstr` does not remove trailing whitespace characters from character arrays when the characters are significant whitespace, such as nonbreaking space characters.

This table shows the most common characters that are significant whitespace characters and their descriptions. For more information, see Whitespace character.

Significant Whitespace Character	Description
`char(133)`	Next line

Significant Whitespace Character	Description
`char(160)`	Nonbreaking space
`char(8199)`	Figure space
`char(8239)`	Narrow no-break space

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- In generated code, this function supports categorical arrays only. For more information, see “Code Generation for Categorical Arrays” (MATLAB Coder) and “Categorical Array Limitations for Code Generation” (MATLAB Coder).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`char` | `iscellstr` | `isstrprop` | `string` | `strsplit`

Introduced before R2006a

cgs

Conjugate gradients squared method

Syntax

```
x = cgs(A,b)
cgs(A,b,tol)
cgs(A,b,tol,maxit)
cgs(A,b,tol,maxit,M)
cgs(A,b,tol,maxit,M1,M2)
cgs(A,b,tol,maxit,M1,M2,x0)
[x,flag] = cgs(A,b,...)
[x,flag,relres] = cgs(A,b,...)
[x,flag,relres,iter] = cgs(A,b,...)
[x,flag,relres,iter,resvec] = cgs(A,b,...)
```

Description

`x = cgs(A,b)` attempts to solve the system of linear equations `A*x = b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and should be large and sparse. The column vector `b` must have length `n`. You can specify `A` as a function handle, `afun`, such that `afun(x)` returns `A*x`.

"Parameterizing Functions" explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `cgs` converges, a message to that effect is displayed. If `cgs` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`cgs(A,b,tol)` specifies the tolerance of the method, `tol`. If `tol` is `[]`, then `cgs` uses the default, `1e-6`.

`cgs(A,b,tol,maxit)` specifies the maximum number of iterations, `maxit`. If `maxit` is `[]` then `cgs` uses the default, `min(n,20)`.

`cgs(A,b,tol,maxit,M)` and `cgs(A,b,tol,maxit,M1,M2)` use the preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `cgs` applies no preconditioner. `M` can be a function handle `mfun` such that `mfun(x)` returns `M\x`.

`cgs(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess `x0`. If `x0` is `[]`, then `cgs` uses the default, an all-zero vector.

`[x,flag] = cgs(A,b,...)` returns a solution `x` and a flag that describes the convergence of `cgs`.

Flag	Convergence
`0`	`cgs` converged to the desired tolerance `tol` within `maxit`iterations.
`1`	`cgs` iterated `maxit` times but did not converge.
`2`	Preconditioner `M` was ill-conditioned.
`3`	`cgs` stagnated. (Two consecutive iterates were the same.)
`4`	One of the scalar quantities calculated during `cgs` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = cgs(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, then `relres <= tol`.

`[x,flag,relres,iter] = cgs(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = cgs(A,b,...)` also returns a vector of the residual norms at each iteration, including `norm(b-A*x0)`.

Examples

Using cgs with a Matrix Input

```
A = gallery('wilk',21);
b = sum(A,2);
```

```
tol = 1e-12;  maxit = 15;
M1 = diag([10:-1:1 1 1:10]);
x = cgs(A,b,tol,maxit,M1);
```

displays the message

```
cgs converged at iteration 13 to a solution with
relative residual 2.4e-016.
```

Using cgs with a Function Handle

This example replaces the matrix `A` in the previous example with a handle to a matrix-vector product function `afun`, and the preconditioner `M1` with a handle to a backsolve function `mfun`. The example is contained in the file `run_cgs` that

- Calls `cgs` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_cgs` are available to `afun` and `myfun`.

The following shows the code for `run_cgs`:

```
function x1 = run_cgs
n = 21;
b = afun(ones(n,1));
tol = 1e-12;  maxit = 15;
x1 = cgs(@afun,b,tol,maxit,@mfun);

    function y = afun(x)
        y = [0; x(1:n-1)] + ...
              [((n-1)/2:-1:0)'; (1:(n-1)/2)'].*x + ...
              [x(2:n); 0];
    end

    function y = mfun(r)
        y = r ./ [((n-1)/2:-1:1)'; 1; (1:(n-1)/2)'];
    end
end
```

When you enter

```
x1 = run_cgs
```

MATLAB software returns

```
cgs converged at iteration 13 to a solution with
relative residual 2.4e-016.
```

Using cgs with a Preconditioner.

This example demonstrates the use of a preconditioner.

Load `west0479`, a real 479-by-479 nonsymmetric sparse matrix.

```
load west0479;
A = west0479;
```

Define `b` so that the true solution is a vector of all ones.

```
b = full(sum(A,2));
```

Set the tolerance and maximum number of iterations.

```
tol = 1e-12;
maxit = 20;
```

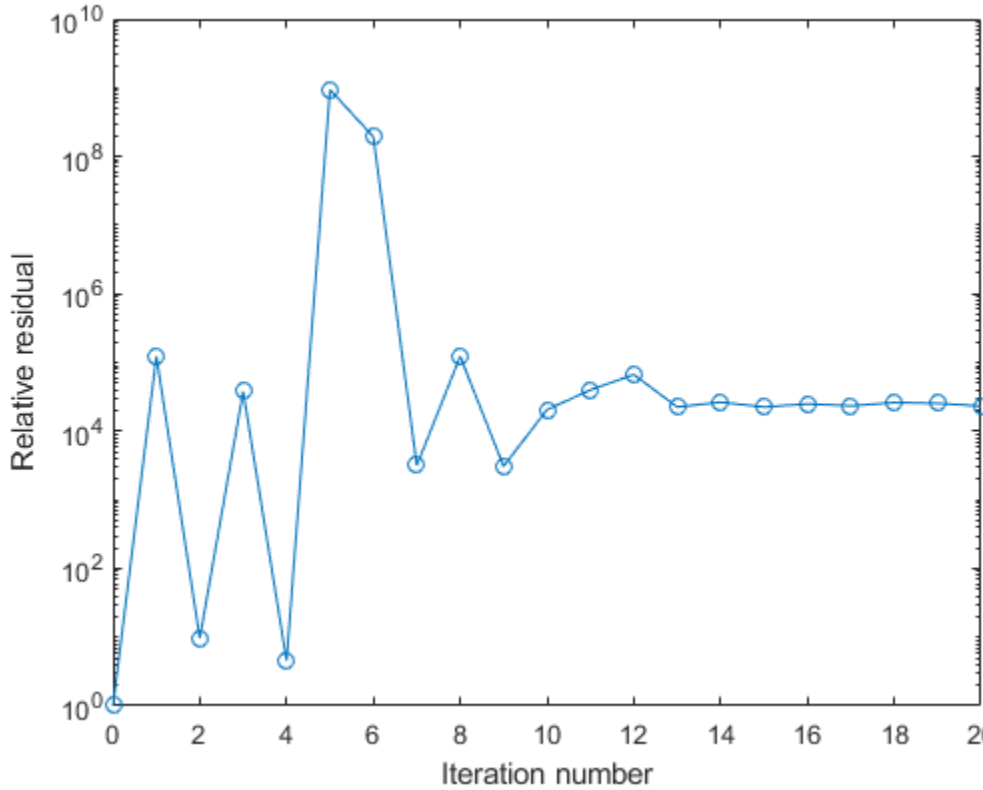
Use `cgs` to find a solution at the requested tolerance and number of iterations.

```
[x0,fl0,rr0,it0,rv0] = cgs(A,b,tol,maxit);
```

`fl0` is 1 because `cgs` does not converge to the requested tolerance `1e-12` within the requested 20 iterations. In fact, the behavior of `cgs` is so poor that the initial guess (`x0 = zeros(size(A,2),1)`) is the best solution and is returned as indicated by `it0 = 0`. MATLAB stores the residual history in `rv0`.

Plot the behavior of `cgs`.

```
semilogy(0:maxit,rv0/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



The plot shows that the solution does not converge. You can use a preconditioner to improve the outcome.

Create a preconditioner with `ilu`, since `A` is nonsymmetric.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-5));
```

```
Error using ilu
There is a pivot equal to zero.  Consider decreasing the
drop tolerance or consider using the 'udiag' option.
```

MATLAB cannot construct the incomplete LU as it would result in a singular factor, which is useless as a preconditioner.

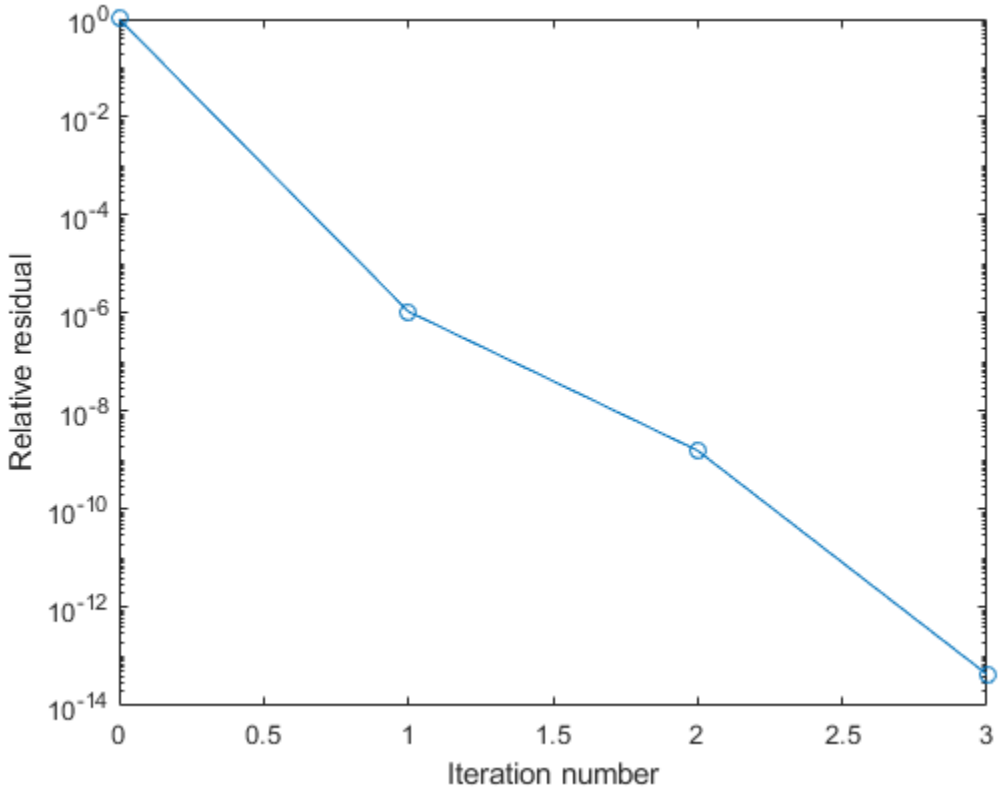
You can try again with a reduced drop tolerance, as indicated by the error message.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
[x1,fl1,rr1,it1,rv1] = cgs(A,b,tol,maxit,L,U);
```

`fl1` is 0 because `cgs` drives the relative residual to `4.3851e-014` (the value of `rr1`). The relative residual is less than the prescribed tolerance of `1e-12` at the third iteration (the value of `it1`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. The output `rv1(1)` is `norm(b)` and the output `rv1(14)` is `norm(b-A*x2)`.

You can follow the progress of `cgs` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0).

```
semilogy(0:it1,rv1/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Sonneveld, Peter, "CGS: A fast Lanczos-type solver for nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, January 1989, Vol. 10, No. 1, pp. 36–52.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- `b` must be a nonsparse column vector.
- When input `A` is a sparse matrix:
  - Only one sparse matrix preconditioner `M` is supported.
  - If you use two preconditioners, `M1` and `M2`, then both of them must be functions.
  - `cgs` does not detect stagnation (Flag 3). Instead, it reports failure to converge (Flag 1).

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- If `M1` is a function, then it is applied independently to each row.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bicg` | `bicgstab` | `gmres` | `ilu` | `lsqr` | `minres` | `mldivide` | `pcg` | `qmr` | `symmlq`

Topics

“Create Function Handle”

Introduced before R2006a

Find Change Points

Find abrupt changes in data in the Live Editor

Description

The **Find Change Points** task lets you interactively find abrupt changes in the mean, variance, or slope and intercept of data. The task automatically generates MATLAB code for your live script.

Using this task, you can:

- Specify the change point type and locate the change points from data in a workspace variable.
- Adjust the number of detected change points.
- Automatically visualize change point locations and the segments of data between them.

Open the Task

To add the **Find Change Points** task to a live script in the MATLAB Editor:

- On the **Live Editor** tab, select **Task > Find Change Points**.
- In a code block in the script, type a relevant keyword, such as `change` or `find`. Select **Find Change Points** from the suggested command completions.

See Also

Functions
`ischange`

Live Editor Tasks
Clean Missing Data | Clean Outlier Data | Find Local Extrema | Remove Trends | Smooth Data

Topics

"Add Interactive Tasks to a Live Script"
Clean Messy Data and Locate Extrema Using Live Editor Tasks

Introduced in R2019b

char

Character array

Description

A character array is a sequence of characters, just as a numeric array is a sequence of numbers. A typical use is to store a short piece of text as a row of characters in a *character vector*.

Creation

You can create a character vector using single quotation marks.

```
C = 'Hello, world'
```

```
C =

    'Hello, world'
```

If you have an array of a different data type, you can convert it to a character array using the `char` function, described below.

Syntax

```
C = char(A)
C = char(A1,...,An)

C = char(D)
C = char(D,fmt)
C = char(D,fmt,locale)
```

Description

`C = char(A)` converts array `A` into a character array.

`C = char(A1,...,An)` converts the arrays `A1,...,An` into a single character array. After conversion to characters, the input arrays become rows in `C`. The `char` function pads rows with blank spaces as needed. If any input array is an empty character array, then the corresponding row in `C` is a row of blank spaces.

The input arrays `A1,...,An` cannot be string arrays, cell arrays, or categorical arrays.

`A1,...,An` can be of different sizes and shapes.

`C = char(D)` converts a datetime, duration, or calendar duration array into a character array in the format specified by the `Format` property of `D`. The output contains one date or duration in each row.

`C = char(D,fmt)` represents dates or durations in the specified format, such as `'HH:mm:ss'`.

`C = char(D,fmt,locale)` represents dates or durations in the specified locale, such as `'en_US'`. The locale affects the language used to represent character vectors such as month and day names.

Input Arguments

`A` — Input array

numeric array | character array | cell array of character arrays | categorical array | string array

Input array, specified as a numeric array, a character array, a cell array of character arrays, a categorical array, or a string array.

- If `A` is a numeric array, then `char` converts numbers into characters. Valid numeric values range from 0 to 65535 and correspond to Unicode code units. Values from 0 to 127 also correspond to 7-bit ASCII characters. The `char` function:
  - Rounds nonintegers toward zero.
  - Treats values less than 0 as 0.
  - Treats values greater than 65535 as 65535.
- If `A` is a character array, then `char` returns `A` unaltered.
- If `A` is a cell array of character arrays, then `char` converts the cell array to a character array. Each row from each character array in the cell array becomes a row in `C`, automatically padded with blank spaces as needed.

  - If A is a multidimensional cell array, then char collapses the output into a two-dimensional character array. For example, if A is a 2-by-2-by-2-by-2 cell array, then the output character array C has 16 rows.
- If A is a categorical array, then char converts each element of A to a row of a character array, in column order.
- If A is a string array, then char converts the string array to a character array. char converts each string element of A to a character vector, and then concatenates the vectors to produce a character array, automatically padded with blank spaces as needed. Since char converts each string to a character vector, the size of the output character array is different from the size of the string array.

Example: char(65) converts the integer 65 to the character A.

D — Input date and time
datetime or duration array

Input date and time, specified as a datetime or duration array.

Data Types: datetime | duration | calendarDuration

fmt — Date and time format
[] (default) | character vector | string scalar

Date and time format, specified as [], a character vector, or a string scalar. If you specify [], then char represents input D in the format specified by the Format property of D.

The supported formats depend on the data type of D.

- datetime formats can include combinations of units and delimiters, such as 'yyyy-MMM-dd HH:mm:ss.SSS'. For details, see the Format property for datetime arrays.
- duration formats are either single characters ('y', 'd', 'h', 'm', or 's') or one of these combinations:
  - 'dd:hh:mm:ss'
  - 'hh:mm:ss'
  - 'mm:ss'
  - 'hh:mm'
  - Any of the above, with up to nine S characters to indicate fractional second digits, such as 'hh:mm:ss.SSSS'

- `calendarDuration` formats can include combinations of the characters `'y'`, `'q'`, `'m'`, `'w'`, `'d'`, and `'t'` in order from the largest to the smallest unit of time, such as `'ym'`.

For more information on the `duration` and `calendarDuration` formats, see “Set Date and Time Display Format”.

`locale` — Locale represented in output
character vector | string scalar

Locale represented in the output, specified as a character vector or a string scalar. The locale affects the language used to represent certain components of dates and times, such as month names.

`locale` can be:

- `'system'`, to specify your system locale.
- A character vector in the form *xx_YY*, where *xx* is a lowercase ISO 639-1 two-letter code that specifies a language, and *YY* is an uppercase ISO 3166-1 alpha-2 code that specifies a country.

The `locale` input argument can be any of the values accepted by the `'Locale'` name-value pair argument for the `datetime` function.

Example: `'en_US'`

Example: `'ja_JP'`

Output Arguments

`C` — Output array
character array

Output array, returned as a character array. Character arrays can have any size, but their most typical use is for storing pieces of text as character vectors.

MATLAB stores all characters as Unicode characters using the UTF-16 encoding. For more information on Unicode, see Unicode.

Examples

Convert Integers to Characters

Convert a numeric array to a character array.

```
A = [77 65 84 76 65 66];
C = char(A)
```

```
C =
'MATLAB'
```

The integers from 32 to 127 correspond to printable ASCII characters. However, the integers from 0 to 65535 also correspond to Unicode® characters. You can convert integers to their corresponding Unicode representations using the `char` function.

For example, the number 8451 corresponds to the symbol for degrees Celsius. Convert 8451 using `char`.

```
C = char(8451)
```

```
C =
'℃'
```

Convert Multiple Arrays to Character Array

Convert multiple arrays into a single character array. The input arrays need not have the same shape.

```
A1 = [65 66; 67 68];
A2 = 'abcd';
C = char(A1,A2)
```

```
C = 3x4 char array
    'AB  '
    'CD  '
    'abcd'
```

Because the input arrays do not have the same number of columns, `char` pads the rows from `A1` with blanks.

```
whos C
```

```
  Name      Size            Bytes  Class    Attributes

  C         3x4                24  char
```

Convert String to Character Vector

Create a string scalar. Starting in R2017a, you can create string scalars using double quotes. MATLAB® also displays strings with double quotes.

```
A = "Pythagoras"
```

```
A =
"Pythagoras"
```

Convert `A` to a character vector using the `char` function. MATLAB displays character vectors with single quotes.

```
C = char(A)
```

```
C =
'Pythagoras'
```

Convert Duration Array to Character Array

Create a `duration` array.

```
D = hours(23:25) + minutes(8) + seconds(1.2345)
```

```
D = 1x3 duration array
   23.134 hr   24.134 hr   25.134 hr
```

Convert `D` to a character array.

```
C = char(D)
```

```
C = 3x9 char array
    '23.134 hr'
    '24.134 hr'
    '25.134 hr'
```

C is a character array that represents one duration value per row.

Specify the format of the duration values represented by C.

```
C = char(D,'hh:mm')
```

```
C = 3x5 char array
    '23:08'
    '24:08'
    '25:08'
```

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

- For the syntax `C = char(A)`, the input A must be a tall numeric column vector.
- Syntaxes with more than one input are not supported.

For more information, see “Tall Arrays for Out-of-Memory Data”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For the syntax `C = char(A)`, the input A must be a string scalar, a numeric array, or a character array.
- Enumeration inputs must be scalar at compile time. Arrays of enumerations are not supported.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- The multiple input syntax `C = char(A1,...,An)` is not supported.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cellstr` | `convertCharsToStrings` | `convertStringsToChars` | `ischar` | `string` | `text`

Topics

“Text in String and Character Arrays”
“Locale Settings for MATLAB Process”

Introduced before R2006a

checkcode

Check MATLAB code files for possible problems

Syntax

```
checkcode(filename)
checkcode(filename1,...,filenameN)

checkcode( ___ ,option1,...,optionN)

info = checkcode( ___ ,'-struct')
msg = checkcode( ___ ,'-string')
[ ___ , filepaths] = checkcode( ___ )
```

Description

`checkcode(filename)` displays messages about `filename` that report potential problems and opportunities for code improvement. These messages are sometimes referred to as Code Analyzer messages. The line number in the message is a hyperlink that you can click to go directly to that line in the Editor. The exact text of the `checkcode` messages is subject to some change between versions.

`checkcode(filename1,...,filenameN)` displays messages for each specified `filename`.

`checkcode( ___ ,option1,...,optionN)` modifies the returned messages based on the specified option flags. For example, specify `'-modcyc'` to request the modified cyclomatic complexity to be returned with each message. You can specify options with any of the input arguments in the previous syntaxes.

`info = checkcode( ___ ,'-struct')` returns the information as an `n`-by-`1` structure array, where `n` is the number of messages found.

`msg = checkcode( ___ ,'-string')` returns the information as a character vector.

If you omit the `'-struct'` or `'-string'` argument and you specify an output argument, the default behavior is `'-struct'`.

`[ ___ , filepaths] = checkcode( ___ )` also returns `filepaths`, the absolute paths to the file names. You can specify `filepaths` with either the `'-struct'` or `'-string'` options.

Examples

Check for Potential Problems in File

Run `checkcode` on the example file `lengthofline.m`. MATLAB® displays the Code Analyzer messages for `lengthofline.m` in the Command Window.

```
checkcode('lengthofline')
```

```
L 21 (C 1-9): The value assigned to variable 'nothandle' might be unused.
L 22 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 23 (C 5-11): The variable 'notline' appears to change size on every loop iteration. C
L 23 (C 44-49): Use STRCMPI(str1,str2) instead of using UPPER/LOWER in a call to STRCMP
L 27 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 33 (C 13-16): The variable 'data' appears to change size on every loop iteration. Con
L 33 (C 24-31): Use dynamic fieldnames with structures instead of GETFIELD.
L 37 (C 29): When both arguments are numeric scalars, consider replacing | with || for
L 38 (C 47): When both arguments are numeric scalars, consider replacing | with || for
L 39 (C 47): When both arguments are numeric scalars, consider replacing | with || for
L 42 (C 13-15): The variable 'dim' appears to change size on every loop iteration. Cons
L 44 (C 13-15): The variable 'dim' appears to change size on every loop iteration. Cons
L 47 (C 52): Invalid syntax at ';'. A '(' might be missing a closing ')'.
L 47 (C 53): Invalid syntax at ')'. A '[' might be missing a closing ']'.
L 47 (C 54): Parse error at ']': usage might be invalid MATLAB syntax.
L 48 (C 17): Terminate statement with semicolon to suppress output (in functions).
L 48 (C 23): If you intend to specify expression precedence, use parentheses () instead
```

Store List of Potential Problems

Run `checkcode` on the example file `lengthofline.m`. Include message IDs and store the results in a structure.

```
info = checkcode('lengthofline', '-id')
```

```
info=17×5 struct
    id
```

```
    message
    fix
    line
    column
```

View the values for the first message

```
info(1)
```

```
ans = struct with fields:
         id: 'NASGU'
    message: 'The value assigned to variable 'nothandle' might be unused.'
        fix: 0
       line: 21
     column: [1 9]
```

Display the Modified Cyclomatic Complexity of File

Run `checkcode` on the example file `lengthofline.m` using the `'-modcyc'` option. MATLAB® displays the modified cyclomatic complexity of the file, followed by the Code Analyzer messages for `lengthofline.m`.

```
checkcode('lengthofline', '-modcyc')
```

```
L 1 (C 23-34): The modified cyclomatic complexity of 'lengthofline' is 12.
L 21 (C 1-9): The value assigned to variable 'nothandle' might be unused.
L 22 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 23 (C 5-11): The variable 'notline' appears to change size on every loop iteration. C
L 23 (C 44-49): Use STRCMPI(str1,str2) instead of using UPPER/LOWER in a call to STRCMP
L 27 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 33 (C 13-16): The variable 'data' appears to change size on every loop iteration. Con
L 33 (C 24-31): Use dynamic fieldnames with structures instead of GETFIELD.
L 37 (C 29): When both arguments are numeric scalars, consider replacing | with || for
L 38 (C 47): When both arguments are numeric scalars, consider replacing | with || for
L 39 (C 47): When both arguments are numeric scalars, consider replacing | with || for
L 42 (C 13-15): The variable 'dim' appears to change size on every loop iteration. Cons
L 44 (C 13-15): The variable 'dim' appears to change size on every loop iteration. Cons
L 47 (C 52): Invalid syntax at ';'. A '(' might be missing a closing ')'.
L 47 (C 53): Invalid syntax at ')'. A '[' might be missing a closing ']'.
L 47 (C 54): Parse error at ']': usage might be invalid MATLAB syntax.
```

```
L 48 (C 17): Terminate statement with semicolon to suppress output (in functions).
L 48 (C 23): If you intend to specify expression precedence, use parentheses () instead
```

Suppress Code Analyzer Messages

Suppress specific messages by creating and specifying a settings file. For example, the file `lengthofline.m` includes several lines that use | instead of || as the `OR` operator. By default, `checkcode` flags these lines.

```
checkcode('lengthofline')
```

```
L 21 (C 1-9): The value assigned to variable 'nothandle' might be unused.
L 22 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 23 (C 5-11): The variable 'notline' appears to change size on every loop iteration. C
L 23 (C 44-49): Use STRCMPI(str1,str2) instead of using UPPER/LOWER in a call to STRCMP
L 27 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 33 (C 13-16): The variable 'data' appears to change size on every loop iteration. Con
L 33 (C 24-31): Use dynamic fieldnames with structures instead of GETFIELD.
L 37 (C 29): When both arguments are numeric scalars, consider replacing | with || for
L 38 (C 47): When both arguments are numeric scalars, consider replacing | with || for
L 39 (C 47): When both arguments are numeric scalars, consider replacing | with || for
L 42 (C 13-15): The variable 'dim' appears to change size on every loop iteration. Cons
L 44 (C 13-15): The variable 'dim' appears to change size on every loop iteration. Cons
L 47 (C 52): Invalid syntax at ';'. A '(' might be missing a closing ')'.
L 47 (C 53): Invalid syntax at ')'. A '[' might be missing a closing ']'.
L 47 (C 54): Parse error at ']': usage might be invalid MATLAB syntax.
L 48 (C 17): Terminate statement with semicolon to suppress output (in functions).
L 48 (C 23): If you intend to specify expression precedence, use parentheses () instead
```

Create a settings file that suppresses the message flagging the use of | as the `OR` operator.

1. On the **Home** tab, in the **Environment** section, click the **Preferences** button.
2. Select **Code Analyzer** in the left pane.
3. Under **Default Settings**, in the **Aesthetics and Readability** section, clear the message **Use instead of | as the OR operator in (scalar) conditional statements**.
4. Enter `mysettings.txt` as the file name and save it to your current folder.
5. Press the **Cancel** button to exit out of the preference panel without changing the active settings.

Run `checkcode` on the example file using the custom settings file `mysettings.txt`. The message **Use instead of | as the OR operator in (scalar) conditional statements** is suppressed and is no longer visible in the list of messages.

```
checkcode('lengthofline','-config=mysettings.txt')
```

```
L 21 (C 1-9): The value assigned to variable 'nothandle' might be unused.
L 22 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 23 (C 5-11): The variable 'notline' appears to change size on every loop iteration. (
L 23 (C 44-49): Use STRCMPI(str1,str2) instead of using UPPER/LOWER in a call to STRCMP
L 27 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 33 (C 13-16): The variable 'data' appears to change size on every loop iteration. Con
L 33 (C 24-31): Use dynamic fieldnames with structures instead of GETFIELD.
L 42 (C 13-15): The variable 'dim' appears to change size on every loop iteration. Cons
L 44 (C 13-15): The variable 'dim' appears to change size on every loop iteration. Cons
L 47 (C 52): Invalid syntax at ';'. A '(' might be missing a closing ')'.
L 47 (C 53): Invalid syntax at ')'. A '[' might be missing a closing ']'.
L 47 (C 54): Parse error at ']': usage might be invalid MATLAB syntax.
L 48 (C 17): Terminate statement with semicolon to suppress output (in functions).
L 48 (C 23): If you intend to specify expression precedence, use parentheses () instead
```

Input Arguments

`filename` — File name

character vector | string array | cell array of character vectors

File name, specified as a character vector, a string array, or a cell array of character vectors. The file name can include a partial path, but must be in a folder on the search path or in the current folder.

If `filename` is a nonscalar string array or a cell array of character vectors, MATLAB displays information for each file.

Note You cannot combine cell arrays and character arrays of file names. For example, you cannot have `{'lengthofline', 'buggy'}, 'collatz'` as an input.

Example: `'lengthofline'`

Example: `{'lengthofline', 'buggy'}`

Data Types: `char` | `string`

`option` — Display option
`'-id'` | `'-fullpath'` | `'-notok'` | `'-cyc'` | `'-modcyc'` | `'-config'`

Display option, specified as one of these values. Options can appear in any order.

Option	Description
`'-id'`	Request the message ID, where ID is a character vector. When returned to a structure, the output also has the `id` field, which is the ID associated with the message.
`'-fullpath'`	Assume that the input file names are absolute paths, so that `checkcode` does not try to locate them.
`'-notok'`	Run `checkcode` for all lines in `filename`, even those lines that end with the `checkcode` suppression directive, `%#ok`. For information on `%#ok` and suppressing messages from within your program, see "Adjust Code Analyzer Message Indicators and Messages".
`'-cyc'`	Display the McCabe cyclomatic complexity of each function in the file. In general, lower complexity values indicate programs that are easier to understand and modify. Evidence suggests that programs with higher complexity values are more likely to contain errors. Frequently, you can lower the complexity of a function by dividing it into smaller, simpler functions. Some people advocate splitting up programs that have a complexity value over 10.

Option	Description
`'-modcyc'`	Displays the modified cyclomatic complexity of each function in the file. The modified cyclomatic complexity for a function is equal to the McCabe cyclomatic complexity except for one difference. McCabe cyclomatic complexity counts each individual `case` within a `switch` statement as 1, while modified cyclomatic complexity counts the entire `switch` statement as 1. In general, `switch` statements are simpler than nested `if-elseif-else` statements and therefore, the modified cyclomatic complexity is often considered a better measure of code complexity.

`'-config=`*`settingsfile`*`'`
 	
`'-config=factory'`
	Override the default active settings file with the specified settings file. If the specified file is not in the current folder, provide the full path to the file.
	
	For information about creating a settings file, see “Save and Reuse Code Analyzer Message Settings”. If you specify an invalid file, `checkcode` returns a message indicating that it cannot open or read the file you specified. In that case, `checkcode` uses the factory default settings.
	
	To ignore all settings files and use the factory default preference settings, specify `'-config=factory'`.

Output Arguments

`info` — Message information
structure array | cell array

Message information, returned as a `n`-by-`1` structure array, where `n` is the number of messages returned by the `checkcode` command. If you specify multiple file names as input, or if you specify a cell array as input, `info` contains a cell array of structures.

Field	Description
`message`	Message describing the suspicious construct that code analysis caught.
`line`	Vector of line numbers, indicating which lines of the file the message applies to.
`column`	Two-column array of columns numbers (column extents), indicating which columns of the file the message applies to. The first column of the array specifies the column in the Editor where the message begins. The second column of the array specifies the column in the Editor where the message ends. In the two-column array, each occurrence of a message has a row.

`msg` — Message information
character vector

Message information, returned as a character vector. If you specify multiple file names as input, or if you specify a cell array as input, `msg` contains a character vector where the information for each file is separated by 10 equal sign characters, a space, the file name, a space, and 10 equal sign characters.

Example: `========== C:\MyMatlabFiles\buggy.m ==========`

`filepaths` — Absolute paths of files
cell array of character vectors

Absolute paths of files, specified as a cell array of character vectors. MATLAB lists the `filepaths` in the same order as the specified input files.

Tips

To force the Code Analyzer to ignore a line of code, use `%#ok` at the end of the line. You can add comments after the tag.

```
unsuppressed1 = 10     % This line will get caught
suppressed2 = 20        %#ok This line will not get caught
suppressed3 = 30        %#ok This line will not get caught
```

See Also

`mlintrpt` | `profile`

Topics

"Check Code for Errors and Warnings"

Introduced in R2011b

checkin

(Has been removed) Check files into source control system (UNIX platforms)

Note `checkin` has been removed.

Syntax

```
checkin('filename','comments','comment_text')
checkin({'filename1','filename2'},'comments','comment_text')
checkin('filename','comments', 'comment_text','option','value')
```

Description

`checkin('filename','comments','comment_text')` checks in the file named `filename` to the source control system. Use the full path for `filename` and include the file extension. You must save the file before checking it in, but the file can be open or closed. The `comment_text` is a MATLAB character vector containing checkin comments for the source control system. You must supply **`comments`** and `comment_text`.

`checkin({'filename1','filename2'},'comments','comment_text')` checks in the files `filename1` through `filenamen` to the source control system. Use the full paths for the files and include file extensions. Comments apply to all files checked in.

`checkin('filename','comments', 'comment_text','option','value')` provides additional `checkin` options. For multiple file names, use an array of character vectors instead of `filename`, that is, `{'filename1','filename2',...}`. Options apply to all file names. The *`option`* and *`value`* arguments are shown in the following table.

option Argument	value Argument	Purpose
`'force'`	`'on'`	`filename` is checked in even if the file has not changed since it was checked out.

option Argument	value Argument	Purpose
`'force'`	`'off'` (default)	`filename` is not checked in if there were no changes since checkout.
`'lock'`	`'on'`	`filename` is checked in with comments, and is automatically checked out.
`'lock'`	`'off'` (default)	`filename` is checked in with comments but does not remain checked out.

Examples

Check In a File

Check the file `/myserver/myfiles/clock.m` into the source control system, with the comment `Adjustment for leapyear`:

```
checkin('/myserver/myfiles/clock.m','comments',...
'Adjustment for leapyear')
```

Check In Multiple Files

Check two files into the source control system, using the same comment for each:

```
checkin({'/myserver/myfiles/clock.m', ...
'/myserver/myfiles/calendar.m'},'comments',...
'Adjustment for leapyear')
```

Check In a File and Keep It Checked Out

Check the file `/myserver/myfiles/clock.m` into the source control system and keep the file checked out:

```
checkin('/myserver/myfiles/clock.m','comments',...
'Adjustment for leapyear','lock','on')
```

Introduced before R2006a

checkout

(Has been removed) Check files out of source control system (UNIX platforms)

Note `checkout` has been removed.

Syntax

```
checkout('filename')
checkout({'filename1','filename2', ...})
checkout('filename','option','value',...)
```

Description

`checkout('filename')` checks out the file named `filename` from the source control system. Use the full path for `filename` and include the file extension. The file can be open or closed when you use `checkout`.

`checkout({'filename1','filename2', ...})` checks out the files named `filename1` through `filenamen` from the source control system. Use the full paths for the files and include the file extensions.

`checkout('filename','option','value',...)` provides additional `checkout` options. For multiple file names, use a cell array of character vectors instead of `filename`, that is, `{'filename1','filename2', ...}`. Options apply to all file names. The *`option`* and *`value`* arguments are shown in the following table.

option Argument	value Argument	Purpose
`'force'`	`'on'`	The checkout is forced, even if you already have the file checked out. This is effectively an `undocheckout` followed by a `checkout`.
`'force'`	`'off'` (default)	Prevents you from checking out the file if you already have it checked out.
`'lock'`	`'on'` (default)	The checkout gets the file, allows you to write to it, and locks the file so that access to the file for others is read only.
`'lock'`	`'off'`	The checkout gets a read-only version of the file, allowing another user to check out the file for updating. You do not have to check the file in after checking it out with this option.
`'revision'`	`'version_num'`	Checks out the specified revision of the file.

If you end the MATLAB session, the file remains checked out. You can check in the file from within the MATLAB desktop during a later session, or directly from your source control system.

Examples

Check Out a File

Check out the file `/myserver/myfiles/clock.m` from the source control system:

```
checkout('/myserver/myfiles/clock.m')
```

Check Out Multiple Files

Check out `/matlab/myfiles/clock.m` and `/matlab/myfiles/calendar.m` from the source control system:

```
checkout({'/myserver/myfiles/clock.m',...
'/myserver/myfiles/calendar.m'})
```

Force a Checkout, Even If File Is Already Checked Out

Check out `/matlab/myfiles/clock.m` even if `clock.m` is already checked out to you:

```
checkout('/myserver/myfiles/clock.m','force','on')
```

Check Out Specified Revision of File

Check out revision `1.1` of `clock.m`:

```
checkout('/matlab/myfiles/clock.m','revision','1.1')
```

Introduced before R2006a

chol

Cholesky factorization

Syntax

```
R = chol(A)
R = chol(A,triangle)
[R,flag] = chol( ___ )
[R,flag,P] = chol(S)
[R,flag,P] = chol( ___ ,outputForm)
```

Description

`R = chol(A)` factorizes symmetric positive definite matrix `A` into an upper triangular `R` that satisfies `A = R'*R`. If `A` is nonsymmetric , then `chol` treats the matrix as symmetric and uses only the diagonal and upper triangle of `A`.

`R = chol(A,triangle)` specifies which triangular factor of `A` to use in computing the factorization. For example, if `triangle` is `'lower'`, then `chol` uses only the diagonal and lower triangular portion of `A` to produce a lower triangular matrix `R` that satisfies `A = R*R'`. The default value of `triangle` is `'upper'`.

`[R,flag] = chol( ___ )` also returns the output `flag` indicating whether `A` is symmetric positive definite on page 1-1662. You can use any of the input argument combinations in previous syntaxes. When you specify the `flag` output, `chol` does not generate an error if the input matrix is not symmetric positive definite.

- If `flag = 0` then the input matrix is symmetric positive definite and the factorization was successful.
- If `flag` is not zero, then the input matrix is *not* symmetric positive definite and `flag` is an integer indicating the index of the pivot position where the factorization failed.

`[R,flag,P] = chol(S)` additionally returns a permutation matrix `P`, which is a preordering of sparse matrix `S` obtained by `amd`. If `flag = 0`, then `S` is symmetric positive definite and `R` is an upper triangular matrix satisfying `R'*R = P'*S*P`.

`[R,flag,P] = chol( ___ ,outputForm)` specifies whether to return the permutation information `P` as a matrix or vector, using any of the input argument combinations in previous syntaxes. This option is only available for sparse matrix inputs. For example, if `outputForm` is `'vector'` and `flag = 0`, then `S(p,p) = R'*R`. The default value of `outputForm` is `'matrix'` such that `R'*R = P'*S*P`.

Examples

Solve Linear System with Symmetric Positive Definite Matrix

Use `chol` to factorize a symmetric coefficient matrix, and then solve a linear system using the Cholesky factor.

Create a symmetric matrix with positive values on the diagonal.

```
A = [1 0 1; 0 2 0; 1 0 3]
```

```
A = 3×3

     1     0     1
     0     2     0
     1     0     3
```

Calculate the Cholesky factor of the matrix.

```
R = chol(A)
```

```
R = 3×3

    1.0000         0    1.0000
         0    1.4142         0
         0         0    1.4142
```

Create a vector for the right-hand side of the equation Ax = b.

```
b = sum(A,2);
```

Since $A = R^T R$ with the Cholesky decomposition, the linear equation becomes $R^T R\, x = b$. Solve for `x` using the backslash operator.

```
x = R\(R'\b)
```

```
x = 3×1

    1.0000
    1.0000
    1.0000
```

Cholesky Factorization of Matrix

Calculate the upper and lower Cholesky factorizations of a matrix and verify the results.

Create a 6-by-6 symmetric positive definite test matrix using the `gallery` function.

```
A = gallery('lehmer',6);
```

Calculate the Cholesky factor using the upper triangle of `A`.

```
R = chol(A)
```

```
R = 6×6

    1.0000    0.5000    0.3333    0.2500    0.2000    0.1667
         0    0.8660    0.5774    0.4330    0.3464    0.2887
         0         0    0.7454    0.5590    0.4472    0.3727
         0         0         0    0.6614    0.5292    0.4410
         0         0         0         0    0.6000    0.5000
         0         0         0         0         0    0.5528
```

Verify that the upper triangular factor satisfies `R'*R - A = 0`, within roundoff error.

```
norm(R'*R - A)
```

```
ans = 2.5801e-16
```

Now, specify the `'lower'` option to calculate the Cholesky factor using the lower triangle of `A`.

```
L = chol(A,'lower')
```

```
L = 6×6
```

```
    1.0000         0         0         0         0         0
    0.5000    0.8660         0         0         0         0
    0.3333    0.5774    0.7454         0         0         0
    0.2500    0.4330    0.5590    0.6614         0         0
    0.2000    0.3464    0.4472    0.5292    0.6000         0
    0.1667    0.2887    0.3727    0.4410    0.5000    0.5528
```

Verify that the lower triangular factor satisfies `L*L' - A = 0`, within roundoff error.

```
norm(L*L' - A)
```

```
ans = 2.5801e-16
```

Suppress Errors for Nonsymmetric Positive Definite Matrices

Use `chol` with two outputs to suppress errors when the input matrix is not symmetric positive definite.

Create a 5-by-5 matrix of binomial coefficients. This matrix is symmetric positive definite, so subtract 1 from the last element to ensure it is no longer positive definite.

```
A = pascal(5);
A(end) = A(end) - 1
```

```
A = 5×5

     1     1     1     1     1
     1     2     3     4     5
     1     3     6    10    15
     1     4    10    20    35
     1     5    15    35    69
```

Calculate the Cholesky factor for `A`. Specify two outputs to avoid generating an error if `A` is not symmetric positive definite.

```
[R,flag] = chol(A)
```

```
R = 4×4

     1     1     1     1
```

```
     0     1     2     3
     0     0     1     3
     0     0     0     1


flag = 5
```

Since `flag` is nonzero, it gives the pivot index where the factorization fails. `chol` is able to calculate `q = flag-1 = 4` rows and columns correctly before failing when it encounters the part of the matrix that changed.

Verify that `R'*R` returns four rows and columns that agree with `A(1:q,1:q)`.

```
q = flag-1;
R'*R

ans = 4×4

     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    20


A(1:q,1:q)

ans = 4×4

     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    20
```

Cholesky Factor of Sparse Matrix

Calculate the Cholesky factor of a sparse matrix, and use the permutation output to create a Cholesky factor with fewer nonzeros.

Create a sparse positive definite matrix based on the `west0479` matrix.

```
load west0479
A = west0479;
S = A'*A;
```

Calculate the Cholesky factor of the matrix two different ways. First specify two outputs, and then specify three outputs to enable row and column reordering.

```
[R,flag] = chol(S);
[RP,flagP,P] = chol(S);
```

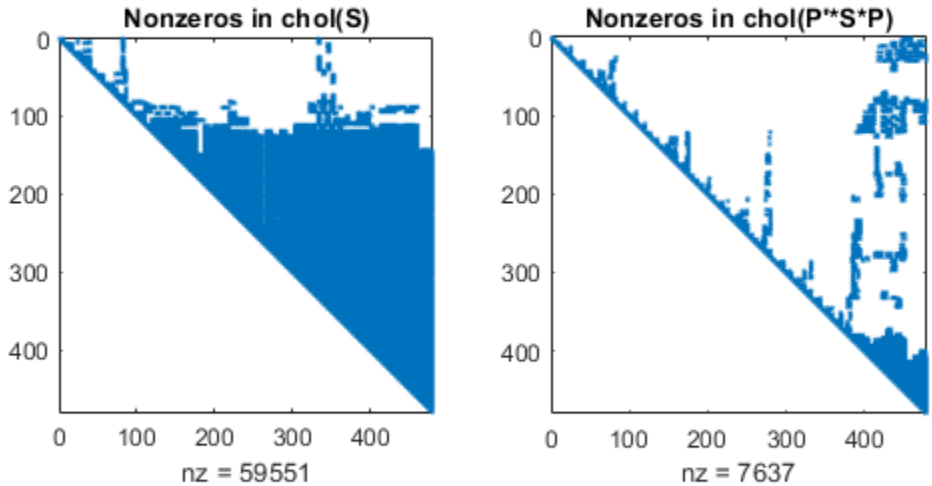
For each calculation, check that `flag = 0` to confirm the calculation is successful.

```
if ~flag && ~flagP
    disp('Factorizations successful.')
else
    disp('Factorizations failed.')
end
```

```
Factorizations successful.
```

Compare the number of nonzeros in `chol(S)` vs. the reordered matrix `chol(P'*S*P)`. Best practice is to use the three output syntax of `chol` with sparse matrices, since reordering the rows and columns can greatly reduce the number of nonzeros in the Cholesky factor.

```
subplot(1,2,1)
spy(R)
title('Nonzeros in chol(S)')
subplot(1,2,2)
spy(RP)
title('Nonzeros in chol(P''*S*P)')
```

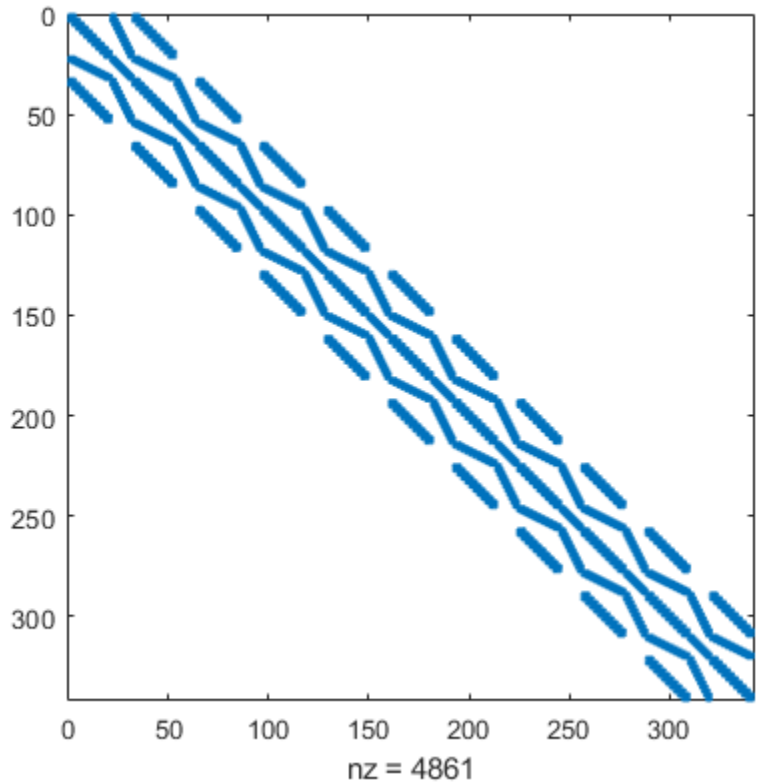


Reorder Sparse Matrix with Permutation Vector

Use the `'vector'` option of `chol` to return the permutation information as a vector rather than a matrix.

Create a sparse finite element matrix.

```
S = gallery('wathen',10,10);
spy(S)
```



Calculate the Cholesky factor for the matrix, and specify the `'vector'` option to return a permutation vector `p`.

```
[R,flag,p] = chol(S,'vector');
```

Verify that `flag = 0`, indicating the calculation is successful.

```
if ~flag
    disp('Factorization successful.')
else
    disp('Factorization failed.')
end
```

```
Factorization successful.
```

Verify that `S(p,p) = R'*R`, within roundoff error.

```
norm(S(p,p) - R'*R,'fro')
```

```
ans = 2.1161e-13
```

Input Arguments

A — Input matrix

matrix

Input matrix. Argument `A` can use full or sparse storage, but must be square and symmetric positive definite.

`chol` assumes that `A` is symmetric for real matrices or Hermitian for complex matrices. `chol` uses only the upper or lower triangle of `A` to perform its computations, depending on the value of `triangle`.

Data Types: `single` | `double`
Complex Number Support: Yes

S — Sparse input matrix

sparse matrix

Sparse input matrix. `S` must be square and symmetric positive definite.

`chol` assumes that `S` is symmetric for real matrices or Hermitian for complex matrices. `chol` uses only the upper or lower triangle of `S` to perform its computations, depending on the value of `triangle`.

Data Types: `double`
Complex Number Support: Yes

triangle — Triangular factor of input matrix

`'upper'` (default) | `'lower'`

Triangular factor of input matrix, specified as `'upper'` or `'lower'`. Use this option to specify that `chol` should use the upper or lower triangle of the input matrix to compute the factorization. `chol` assumes that the input matrix is symmetric for real matrices or Hermitian for complex matrices. `chol` uses only the upper or lower triangle to perform its computations.

Using the `'lower'` option is equivalent to calling `chol` with the `'upper'` option and the transpose of the input matrix, and then transposing the output `R`.

Example: `R = chol(A,'lower')`

`outputForm` — Shape of permutation output
`'matrix'` (default) | `'vector'`

Shape of permutation output, specified as `'matrix'` or `'vector'`. This flag controls whether the permutation output `P` is returned as a permutation matrix or permutation vector.

- If `flag = 0`, then `S` is symmetric positive definite and `P'*S*P = R'*R` (if `P` is a matrix) or `S(p,p) = R'*R` (if `p` is a vector).
- If `flag` is not zero, then `S` is not symmetric positive definite. `R` is an upper triangular matrix of size `q`-by-`n`, where `q = flag-1`. The L-shaped region of the first `q` rows and first `q` columns of `R'*R` agree with those of `P'*S*P` (if `P` is a matrix) or `S(p,p)` (if `p` is a vector).
- If the `'lower'` option is specified, then `R` is a lower triangular matrix and you can replace `R'*R` with `R*R'` in the previous identities.

The Cholesky factor of `P'*S*P` (if `P` is a matrix) or `S(p,p)` (if `p` is a vector) tends to be sparser than the Cholesky factor of `S`.

Example: `[R,flag,p] = chol(S,'vector')`

Output Arguments

`R` — Cholesky factor
matrix

Cholesky factor, returned as a matrix.

- If `R` is upper triangular, then `A = R'*R`. If you specify the `P` output for sparse matrices, then `P'*S*P = R'*R` or `S(p,p) = R'*R`, depending on the value of `outputForm`.
- If `R` is lower triangular, then `A = R*R'`. If you specify the `P` output for sparse matrices, then `P'*S*P = R*R'` or `S(p,p) = R*R'`, depending on the value of `outputForm`..

- Whenever `flag` is not zero, `R` contains only partial results. `flag` indicates the pivot position where the factorization failed, and `R` contains the partially completed factorization.

`flag` — Symmetric positive definite flag
scalar

Symmetric positive definite flag, returned as a scalar.

- If `flag = 0`, then the input matrix is symmetric positive definite. `R` is an upper triangular matrix such that `R'*R = A`.
- If `A` is not symmetric positive definite, then `flag` is a positive integer indicating the pivot position where the factorization failed, and MATLAB does not generate an error. `R` is an upper triangular matrix of size `q = flag-1` such that `R'*R = A(1:q,1:q)`.
- If `A` is sparse, then `R` is an upper triangular matrix of size `q`-by-`n` such that the L-shaped region of the first `q` rows and first `q` columns of `R'*R` agree with those of `A` or `S`.
- If the `'lower'` option is specified, then `R` is a lower triangular matrix and you can replace `R'*R` with `R*R'` in the previous identities.

`P` — Permutation for sparse matrices
matrix | vector

Permutation for sparse matrices, returned as a matrix or vector depending on the value of `outputForm`. See `outputForm` for a description of the identities that this output satisfies.

This permutation matrix is based on the approximate minimum degree ordering computed by `amd`. However, this preordering can differ from the one obtained directly by `amd` since `chol` slightly changes the ordering for increased performance.

More About

Symmetric Positive Definite Matrix

A *symmetric positive definite matrix* is a symmetric matrix with all positive eigenvalues.

For any real invertible matrix `A`, you can construct a symmetric positive definite matrix with the product `B = A'*A`. The Cholesky factorization reverses this formula by saying that any symmetric positive definite matrix `B` can be factored into the product `R'*R`.

A *symmetric positive semi-definite* matrix is defined in a similar manner, except that the eigenvalues must all be positive or zero.

The line between positive definite and positive semi-definite matrices is blurred in the context of numeric computation. It is rare for eigenvalues to be exactly equal to zero, but they can be numerically zero (on the order of machine precision). For this reason, `chol` might be able to factorize one positive semi-definite matrix, but could fail with another matrix that has very similar eigenvalues.

Tips

- Use `chol` (instead of `eig`) to efficiently determine whether a matrix is symmetric positive definite. See “Determine Whether Matrix Is Symmetric Positive Definite” for more information.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For sparse matrix inputs, the standard math library must be C99 or later. Only the first two syntaxes `chol(A)` and `chol(A,triangle)` with one output argument are supported.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Only the first two syntaxes `chol(A)` and `chol(A,triangle)` with one output argument are supported.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The input `A` must be nonsparse.
- The `'vector'` option is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- The input `A` must be nonsparse.
- The `'vector'` option is not supported.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cholupdate` | `ichol` | `ldl` | `qr`

Topics

“Factorizations”

Introduced before R2006a

cholupdate

Rank 1 update to Cholesky factorization

Syntax

```
R1 = cholupdate(R,x)
R1 = cholupdate(R,x,'+')
R1 = cholupdate(R,x,'-')
[R1,p] = cholupdate(R,x,'-')
```

Description

`R1 = cholupdate(R,x)` where `R = chol(A)` is the original Cholesky factorization of `A`, returns the upper triangular Cholesky factor of `A + x*x'`, where `x` is a column vector of appropriate length. `cholupdate` uses only the diagonal and upper triangle of `R`. The lower triangle of `R` is ignored.

`R1 = cholupdate(R,x,'+')` is the same as `R1 = cholupdate(R,x)`.

`R1 = cholupdate(R,x,'-')` returns the Cholesky factor of `A - x*x'`. An error message reports when R is not a valid Cholesky factor or when the downdated matrix is not positive definite and so does not have a Cholesky factorization.

`[R1,p] = cholupdate(R,x,'-')` will not return an error message. If `p` is `0`, `R1` is the Cholesky factor of `A - x*x'`. If `p` is greater than `0`, `R1` is the Cholesky factor of the original `A`. If `p` is `1`, `cholupdate` failed because the downdated matrix is not positive definite. If `p` is `2`, `cholupdate` failed because the upper triangle of `R` was not a valid Cholesky factor.

Examples

```
A = pascal(4)
A =

     1     1     1     1
```

```
     1     2     3     4
     1     3     6    10
     1     4    10    20

R = chol(A)
R =

     1     1     1     1
     0     1     2     3
     0     0     1     3
     0     0     0     1
x = [0 0 0 1]';
```

This is called a rank one update to `A` since `rank(x*x')` is `1`:

```
A + x*x'
ans =

     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    21
```

Instead of computing the Cholesky factor with `R1 = chol(A + x*x')`, we can use `cholupdate`:

```
R1 = cholupdate(R,x)
R1 =

    1.0000    1.0000    1.0000    1.0000
         0    1.0000    2.0000    3.0000
         0         0    1.0000    3.0000
         0         0         0    1.4142
```

Next destroy the positive definiteness (and actually make the matrix singular) by subtracting `1` from the last element of `A`. The downdated matrix is:

```
A - x*x'
ans =

     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    19
```

Compare `chol` with `cholupdate`:

```
R1 = chol(A-x*x')
Error using chol
Matrix must be positive definite.
R1 = cholupdate(R,x,'-')
Error using cholupdate
Downdated matrix must be positive definite.
```

However, subtracting `0.5` from the last element of `A` produces a positive definite matrix, and we can use `cholupdate` to compute its Cholesky factor:

```
x = [0 0 0 1/sqrt(2)]';
R1 = cholupdate(R,x,'-')
R1 =
    1.0000    1.0000    1.0000    1.0000
         0    1.0000    2.0000    3.0000
         0         0    1.0000    3.0000
         0         0         0    0.7071
```

Tips

`cholupdate` works only for full matrices.

Algorithms

`cholupdate` uses the algorithms from the LINPACK subroutines `ZCHUD` and `ZCHDD`. `cholupdate` is useful since computing the new Cholesky factor from scratch is an $O(N^3)$ algorithm, while simply updating the existing factor in this way is an $O(N^2)$ algorithm.

References

[1] Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

See Also

`chol` | `qrupdate`

Introduced before R2006a

circshift

Shift array circularly

Syntax

```
Y = circshift(A,K)
Y = circshift(A,K,dim)
```

Description

`Y = circshift(A,K)` circularly shifts the elements in array `A` by `K` positions. If `K` is an integer, then `circshift` shifts along the first dimension of `A` whose size does not equal 1. If `K` is a vector of integers, then each element of `K` indicates the shift amount in the corresponding dimension of `A`.

Note The default behavior of `circshift(A,K)` where `K` is a scalar changed in R2016b. To preserve the behavior of R2016a and previous releases, use `circshift(A,K,1)`. This syntax specifies 1 as the dimension to operate along.

`Y = circshift(A,K,dim)` circularly shifts the values in array `A` by `K` positions along dimension `dim`. Inputs `K` and `dim` must be scalars.

Examples

Shift Column Vector Elements

Create a numeric column vector.

```
A = (1:10)'
```

```
A = 10×1
```

```
     1
     2
     3
     4
     5
     6
     7
     8
     9
    10
```

Use `circshift` to shift the elements by three positions.

```
Y = circshift(A,3)
```

```
Y = 10×1

     8
     9
    10
     1
     2
     3
     4
     5
     6
     7
```

The result, `Y`, has the same elements as `A` but they are in a different order.

Shift Characters in Array

Create an array of characters and use `circshift` to shift the characters by 3 positions. The characters are in a different order in `Y`.

```
A = 'racecar';
Y = circshift(A,3)
```

```
Y =
'carrace'
```

Shift Matrix Elements

Create a numeric array with a cluster of ones in the top left.

```
A = [1 1 0 0; 1 1 0 0; 0 0 0 0; 0 0 0 0]
```

```
A = 4×4

     1     1     0     0
     1     1     0     0
     0     0     0     0
     0     0     0     0
```

Use `circshift` to shift each row of `A` one position to the right.

```
Y = circshift(A,1,2)
```

```
Y = 4×4

     0     1     1     0
     0     1     1     0
     0     0     0     0
     0     0     0     0
```

Shift the elements of `A` by one position in each dimension. The cluster of ones is now in the center of the matrix.

```
Y = circshift(A,[1 1])
```

```
Y = 4×4

     0     0     0     0
     0     1     1     0
     0     1     1     0
     0     0     0     0
```

To move the cluster back to its original position, use `circshift` on `Y` with negative shift values. The matrix `X` is equivalent to `A`.

```
X = circshift(Y,[-1 -1])
```

```
X = 4×4

     1     1     0     0
     1     1     0     0
     0     0     0     0
     0     0     0     0
```

Input Arguments

A — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell`
Complex Number Support: Yes

K — Shift amount

integer scalar | vector of integers

Shift amount, specified as an integer scalar or vector of integers.

- If you specify K as an integer and do not specify `dim`, then `circshift` shifts along the first dimension whose size does not equal 1. Positive K shifts toward the end of the dimension and negative K shifts toward the beginning.
- If you specify K as a vector of integers, then the *N*th element in K specifies the shift amount for the *N*th dimension in A. If the *N*th element in K is positive, then the values of A shift toward the end of the *N*th dimension. If the *N*th element is negative, then the values shift toward the beginning.

If the shift amount is greater than the length of the corresponding dimension in A, then the shift circularly wraps to the beginning of that dimension. For example, shifting a 3-element vector by +3 positions brings its elements back to their original positions.

dim — Dimension to operate along

positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, the default is the first dimension whose size does not equal 1. If you specify `dim`, then K

must be an integer scalar. In general, specify `dim = 1` to exchange rows, `dim = 2` to exchange columns, and so on.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`fftshift` | `permute` | `reshape` | `shiftdim`

Introduced before R2006a

circumcenters

Class: `TriRep`

(Not recommended) Circumcenters of specified simplices

Note `circumcenters(TriRep)` is not recommended. Use `circumcenter(triangulation)` instead.

`TriRep` is not recommended. Use `triangulation` instead.

Syntax

```
CC = circumcenters(TR, SI)
[CC RCC] = circumcenters(TR, SI)
```

Description

`CC = circumcenters(TR, SI)` returns the coordinates of the circumcenter of each specified simplex `SI`. `CC` is an `m`-by-`n` matrix, where `m` is of length `length(SI)`, the number of specified simplices, and `n` is the dimension of the space where the triangulation resides.

`[CC RCC] = circumcenters(TR, SI)` returns the circumcenters and the corresponding radii of the circumscribed circles or spheres.

Input Arguments

TR	Triangulation object.

`SI`	Column vector of simplex indices that index into the triangulation matrix `TR.Triangulation`. If `SI` is not specified the circumcenter information for the entire triangulation is returned, where the circumcenter associated with simplex `i` is the `i`'th row of `CC`.

Output Arguments

`CC`	`m`-by-`n` matrix. `m` is the number of specified simplices and `n` is the dimension of the space where the triangulation resides. Each row `CC(i,:)` represents the coordinates of the circumcenter of simplex `SI(i)`.
`RCC`	Vector of length `length(SI)`, the number of specified simplices containing radii of the circumscribed circles or spheres.

Examples

Example 1

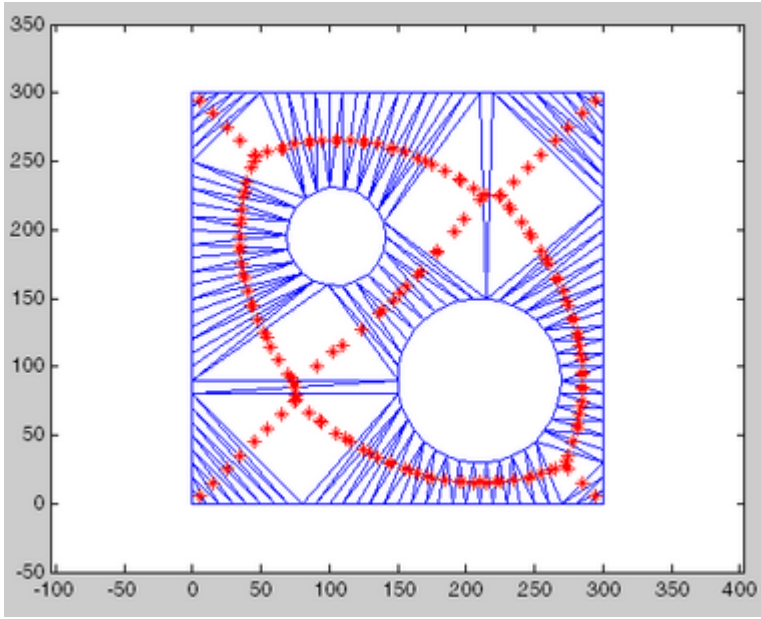
Load a 2-D triangulation.

```
load trimesh2d
trep = TriRep(tri, x,y)
```

Compute the circumcenters.

```
cc = circumcenters(trep);
triplot(trep);
axis([-50 350 -50 350]);
axis equal;
hold on;
plot(cc(:,1),cc(:,2),'*r');
hold off;
```

The circumcenters represent points on the medial axis of the polygon.



Example 2

Query a 3-D triangulation created with `DelaunayTri`. Compute the circumcenters of the first five tetrahedra.

```
X = rand(10,3);
dt = DelaunayTri(X);
cc = circumcenters(dt, [1:5]')
```

More About

Simplex

A simplex is a triangle/tetrahedron or higher-dimensional equivalent.

See Also

`delaunayTriangulation` | `incenter` | `triangulation`

cla

Clear axes

Syntax

```
cla
cla(ax)

cla reset
cla(ax,'reset')
```

Description

`cla` deletes all graphics objects that have visible handles from the current axes. The handle is visible if the `HandleVisibility` property of the object is set to `'on'`. The next plot added to the axes uses the first color and line style based on the `ColorOrder` and `LineStyleOrder` properties of the axes. If axes do not exist, then this command creates one.

`cla(ax)` deletes graphics objects from the axes, polar axes, or geographic axes specified by `ax` instead of the current axes.

`cla reset` deletes graphics objects from the current axes regardless of their handle visibility. It also resets axes properties to their default values, except for the `Position` and `Units` properties.

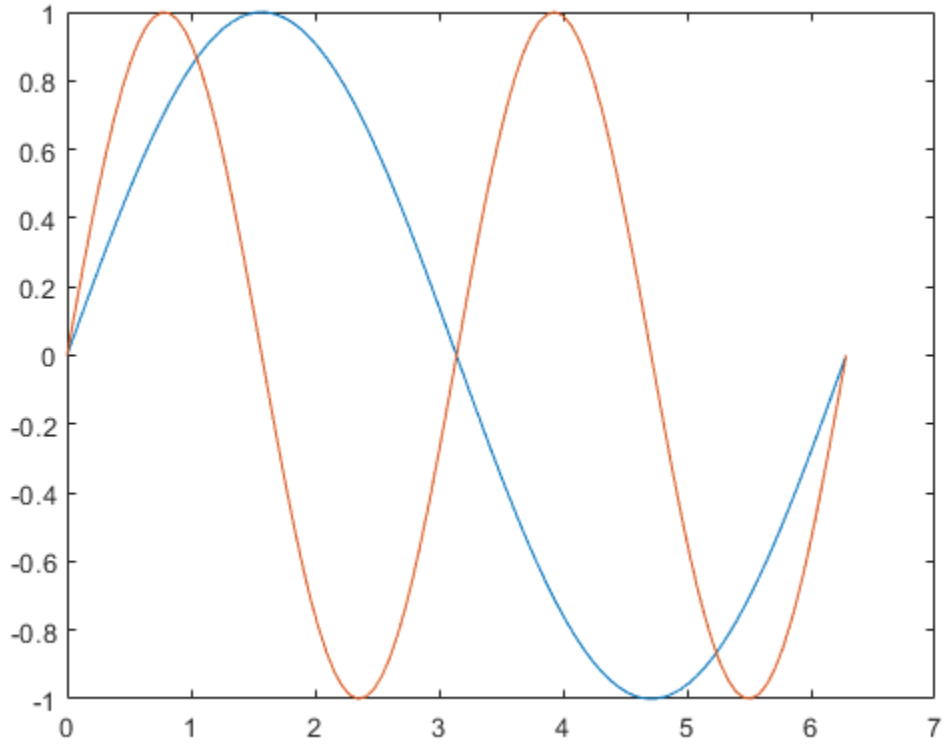
`cla(ax,'reset')` resets properties for the specified axes.

Examples

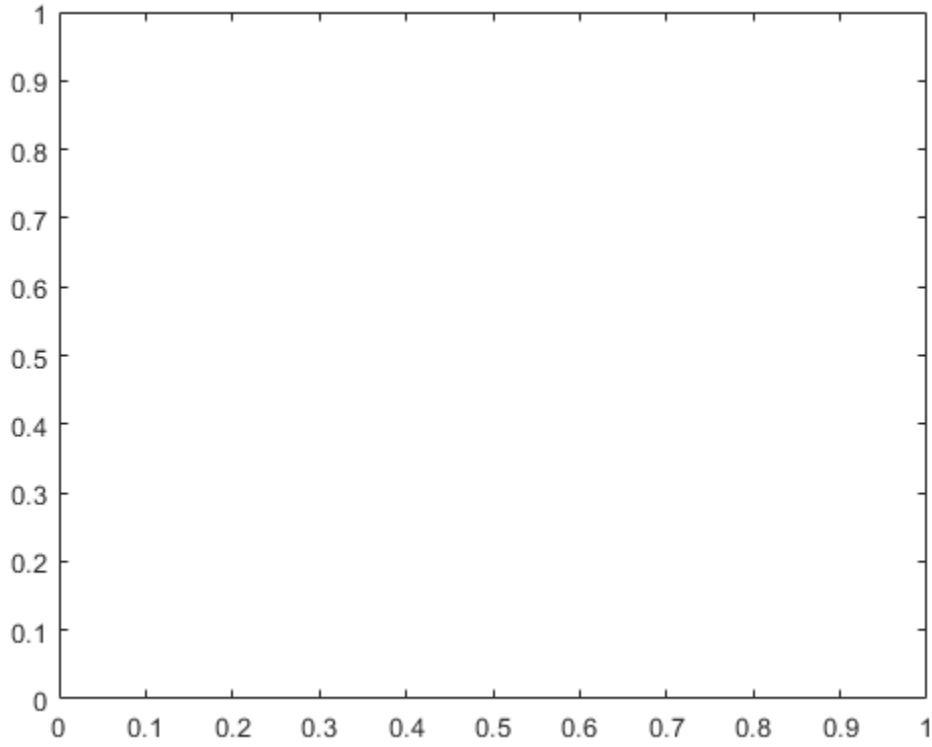
Clear Current Axes

Plot two sine waves. Then, clear the line plots from the axes.

```
x = linspace(0,2*pi);
y1 = sin(x);
plot(x,y1)

hold on
y2 = sin(2*x);
plot(x,y2)
```

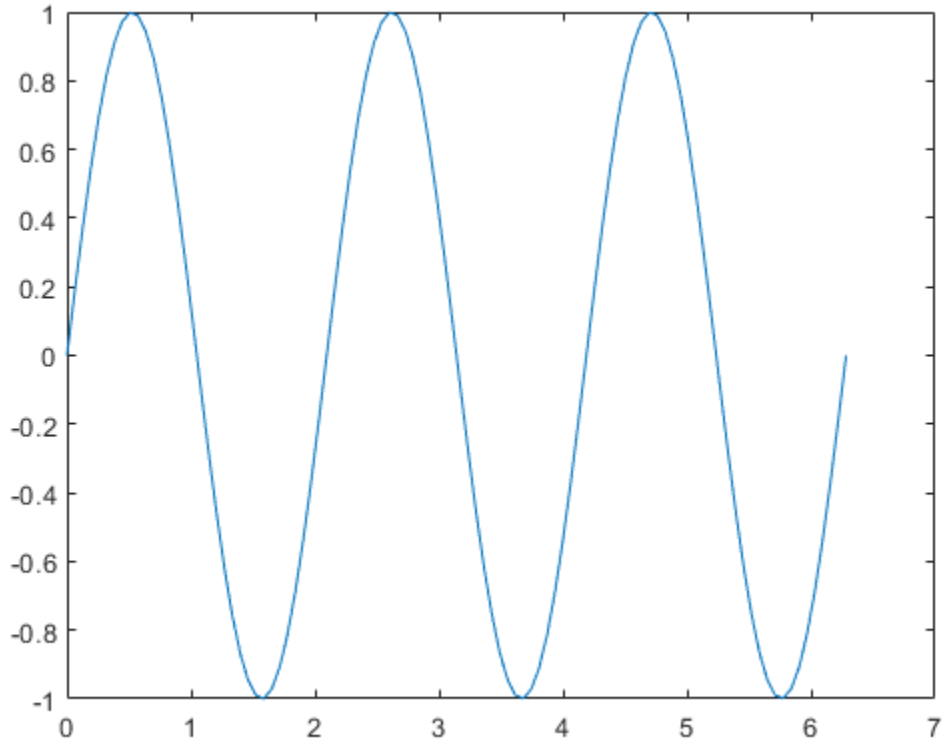



```
cla
```



`cla` clears the line plots and resets the `ColorIndex` and `LineStyleIndex` properties of the axes to 1. Subsequent plots start from the beginning of the color order and line style order. For example, plot another sine wave.

```
y3 = sin(3*x);
plot(x,y3)
hold off
```

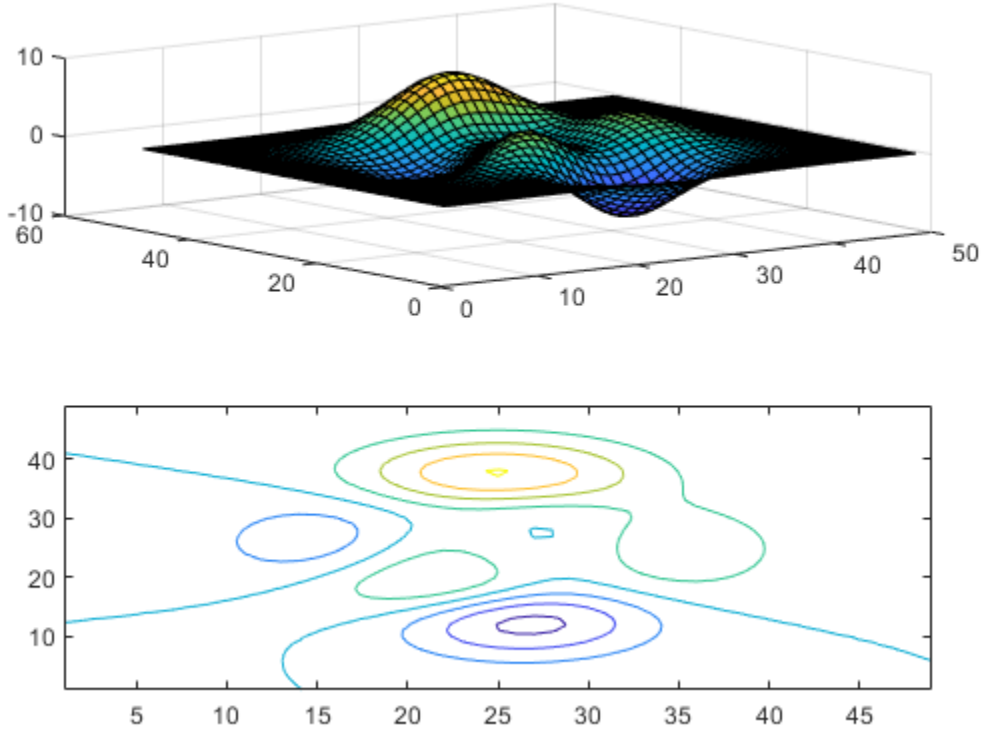


Clear Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Add plots to both axes.

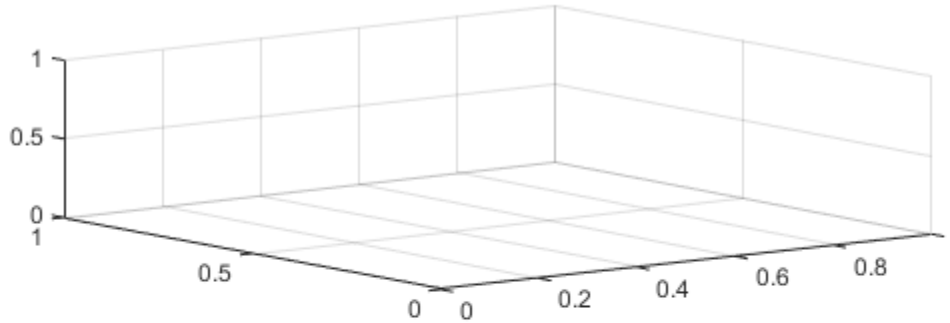
```
tiledlayout(2,1)
ax1 = nexttile;
surf(ax1,peaks)
```

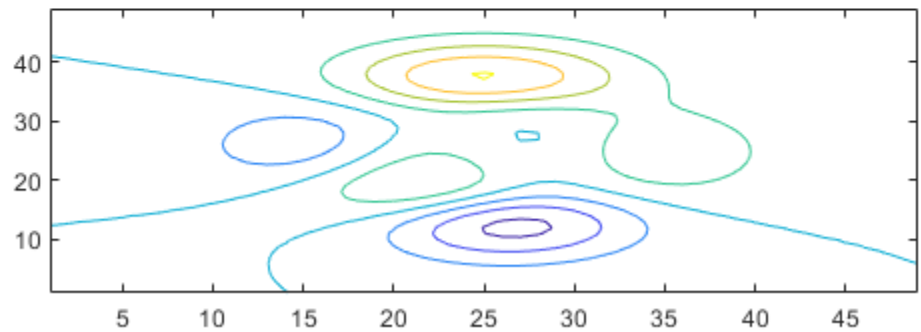
```
ax2 = nexttile;
contour(ax2,peaks)
```



Clear the surface plot from the upper axes by specifying `ax1` as an input argument to `cla`.

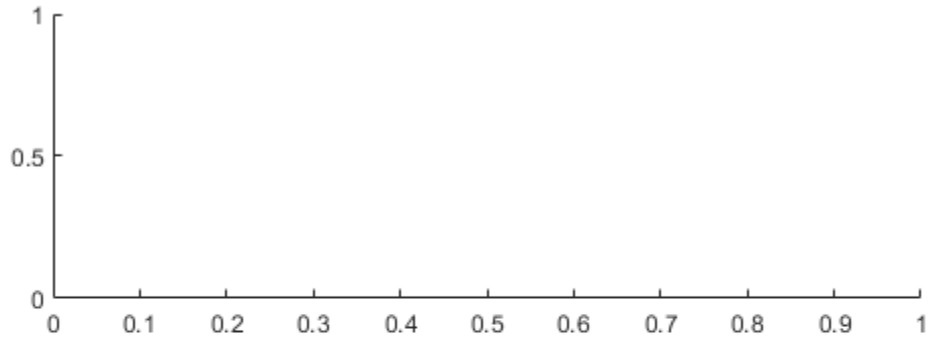
```
cla(ax1)
```

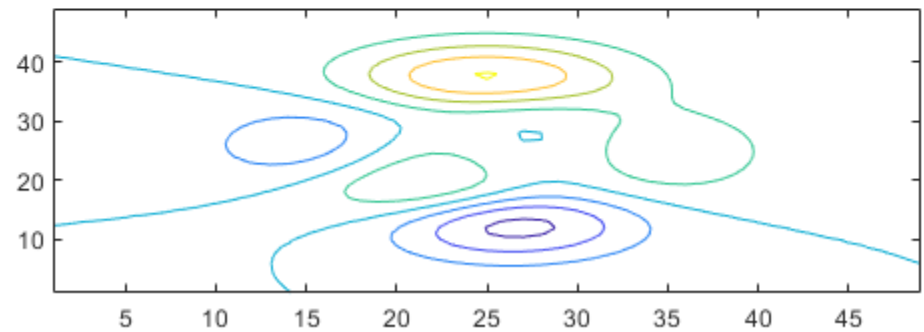




Now, reset all properties for the axes, including the camera properties that control the view, by using the optional input argument `'reset'`.

```
cla(ax1,'reset')
```

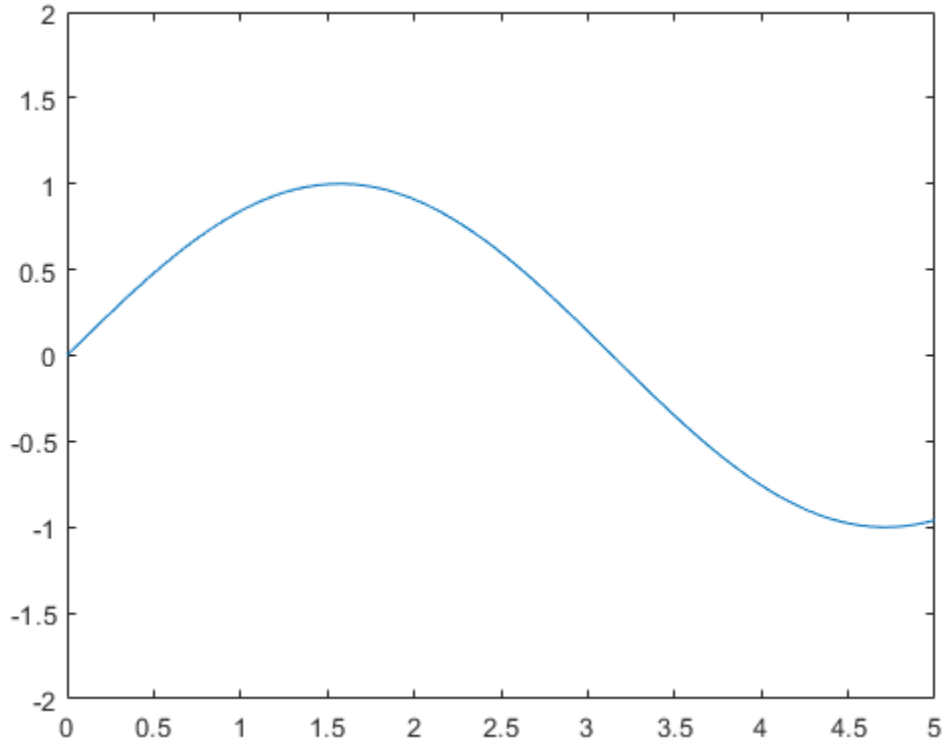




Clear Axes and Reset All Axes Properties

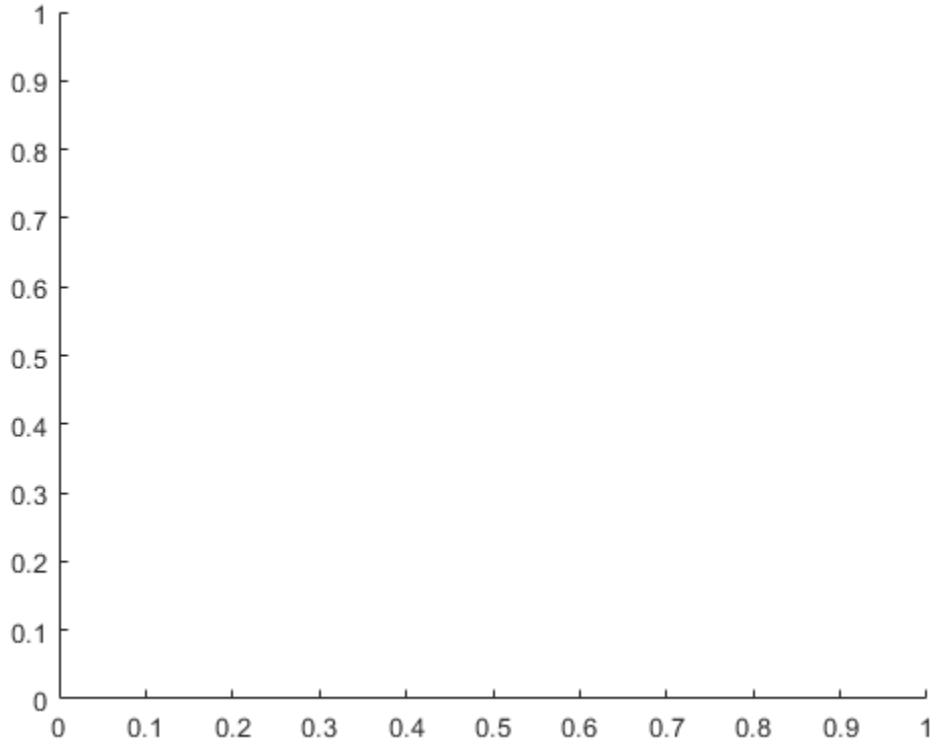
Create a line plot and set the axis limits.

```
x = linspace(0,2*pi);
y = sin(x);
plot(x,y)
axis([0 5 -2 2])
```



Clear the line plot from the axes and reset all the axes properties to their default values. `cla reset` resets all properties of the current axes, except for the `Position` and `Units` properties.

```
cla reset
```



Input Arguments

ax — Target axes
`Axes` object | `PolarAxes` object | `GeographicAxes` object

Target axes, specified as an `Axes` object, a `PolarAxes` object, or a `GeographicAxes` object. Use `ax` to clear specific axes, instead of the current axes.

Algorithms

The `cla` command resets the `ColorOrderIndex` and `LineStyleOrderIndex` properties of the current axes to `1`.

See Also

Functions
`clf` | `hold` | `newplot` | `reset`

Properties
Axes

Introduced before R2006a

clabel

Label contour plot elevation

Syntax

```
clabel(C,h)
clabel(C,h,v)
clabel(C,h,'manual')
t = clabel(C,h,'manual')
```

```
clabel(C)
clabel(C,v)
clabel(C,'manual')
tl = clabel( ___ )
```

```
clabel( ___ ,Name,Value)
```

Description

`clabel(C,h)` labels the current contour plot with rotated text inserted into each contour line. The contour lines must be long enough to fit the label, otherwise `clabel` does not insert a label. If you do not have the contour matrix `C`, then replace `C` with `[]`.

`clabel(C,h,v)` labels the contour levels specified by vector `v`.

`clabel(C,h,'manual')` labels the locations you select with the mouse. Click the mouse or press the space bar to label the contour closest to the center of the crosshair. Press the **Return** key while the cursor is within the figure window to terminate labeling.

`t = clabel(C,h,'manual')` returns the text objects created.

`clabel(C)` labels contours with `'+'` symbols and upright text.

`clabel(C,v)` adds upright labels to the contour levels specified by the vector, `v`.

`clabel(C,'manual')` adds upright labels at the locations you select with the mouse. Click the mouse or press the space bar to label the contour closest to the center of the

crosshair. Press the **Return** key while the cursor is within the figure window to terminate labeling.

`tl = clabel( ___ )` returns the text and line objects created. Use any of the input argument combinations in the previous syntaxes that do not include the contour object `h`.

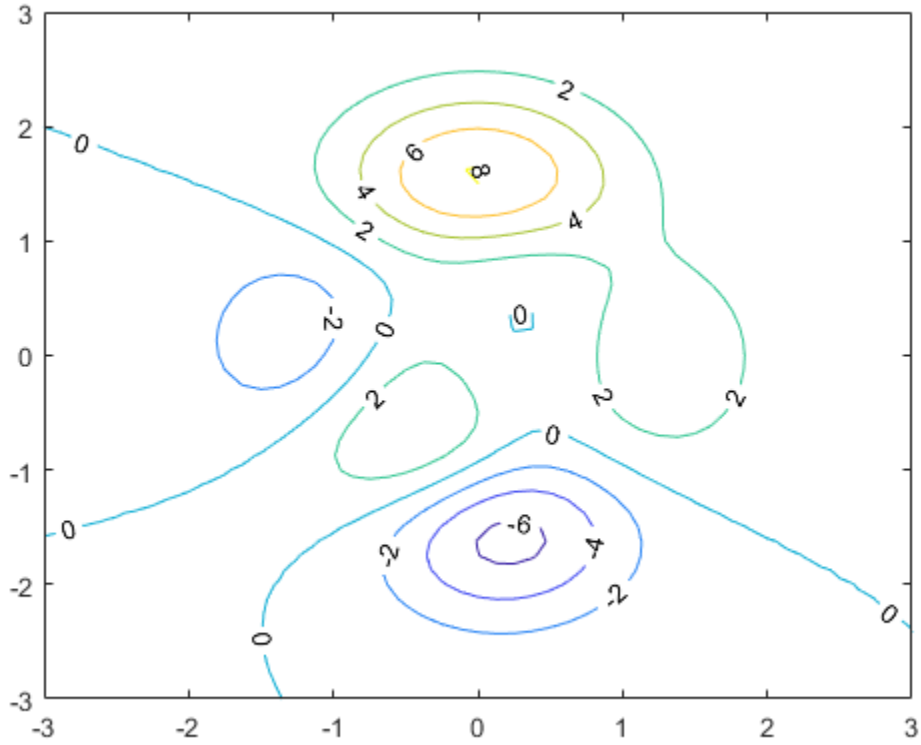
`clabel( ___ ,Name,Value)` modifies the label appearance using one or more `Name,Value` pair arguments. Use any of the input argument combinations in the previous syntaxes. For example, `'FontSize',14` sets the font size to 14 points.

Examples

Label Contour Plot Levels

Create a contour plot and obtain the contour matrix, `C`, and the contour object, `h`. Then, label the contour plot.

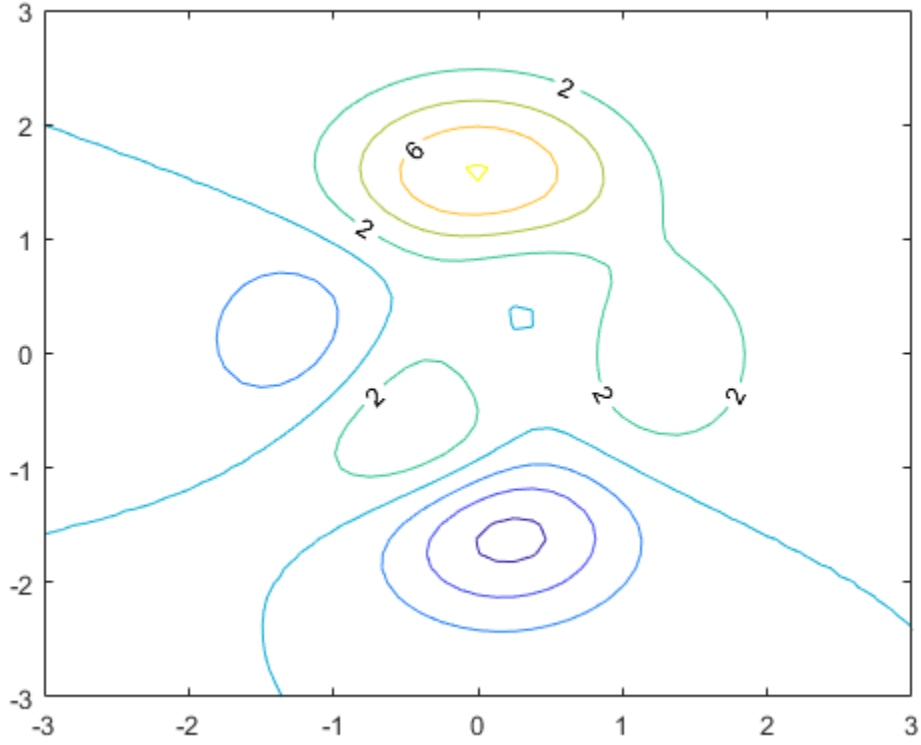
```
[x,y,z] = peaks;
[C,h] = contour(x,y,z);
clabel(C,h)
```



Label Specific Contour Levels

Label only the contours with contour levels 2 or 6.

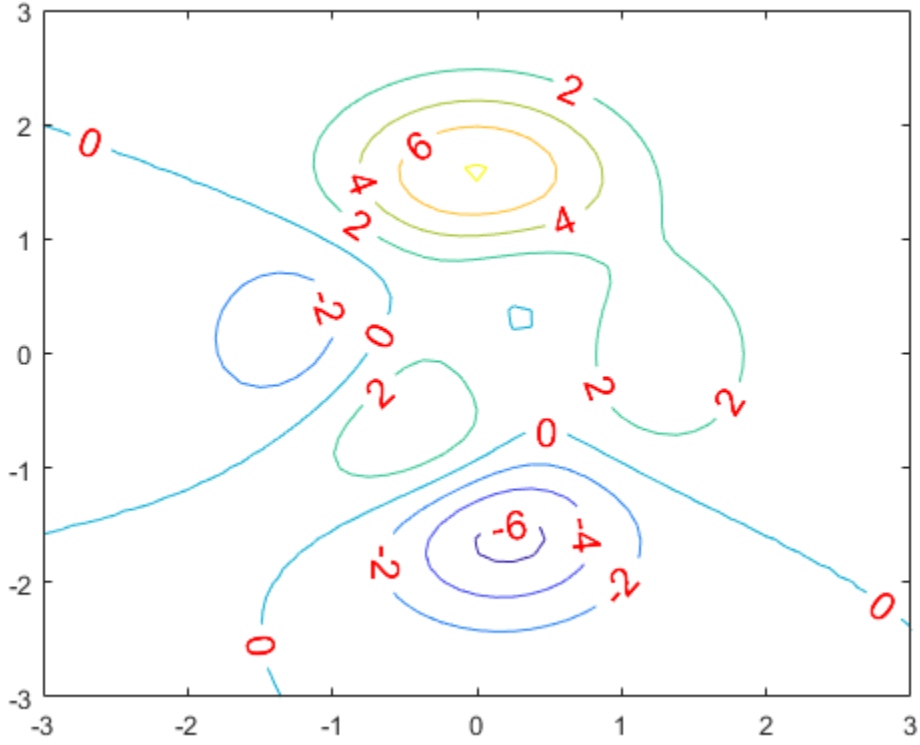
```
[x,y,z] = peaks;
[C,h] = contour(x,y,z);
v = [2,6];
clabel(C,h,v)
```



Set Contour Label Properties

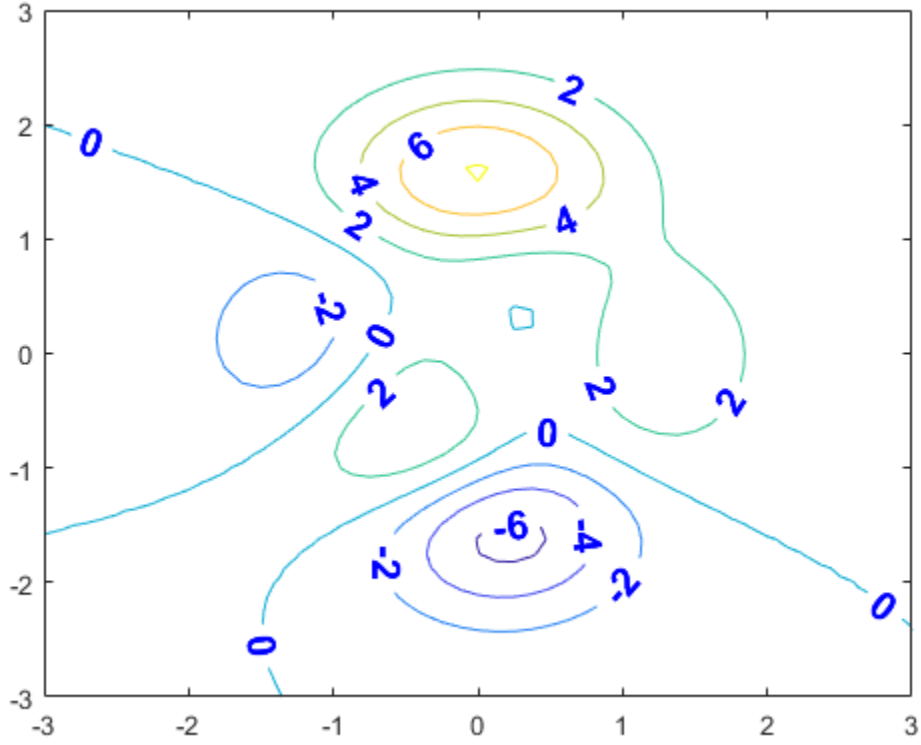
Set the font size of the labels to 15 points and set the color to red using `Name,Value` pair arguments.

```
[x,y,z] = peaks;
[C,h] = contour(x,y,z);
clabel(C,h,'FontSize',15,'Color','red')
```



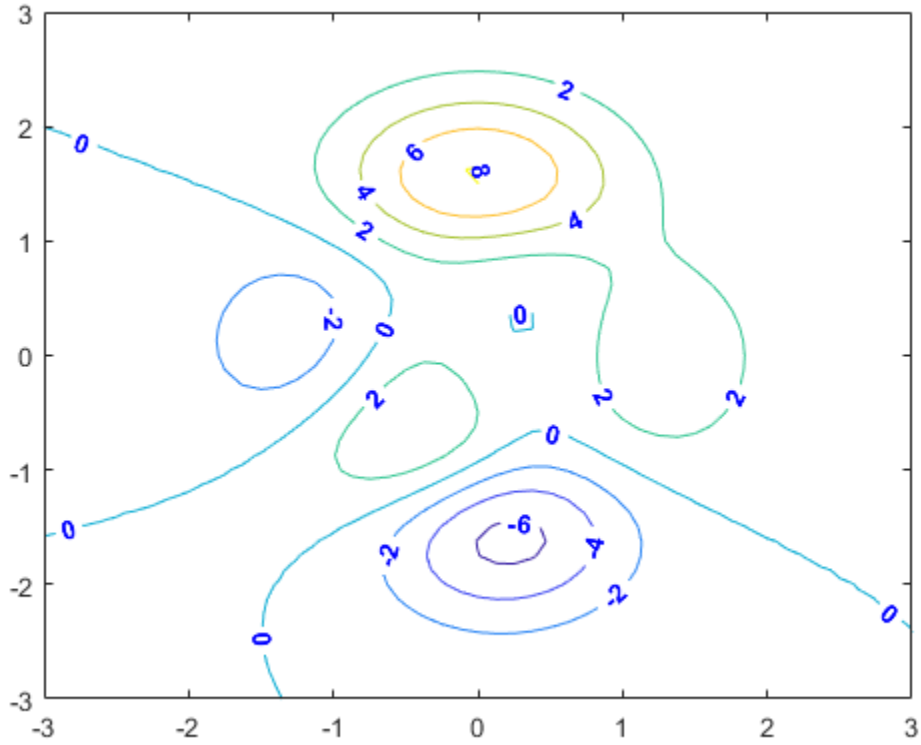
Set additional properties by reissuing the `clabel` command. For example, set the font weight to bold and change the color to blue.

```
clabel(C,h,'FontWeight','bold','Color','blue')
```



Set the font size back to the default size using the `'default'` keyword.

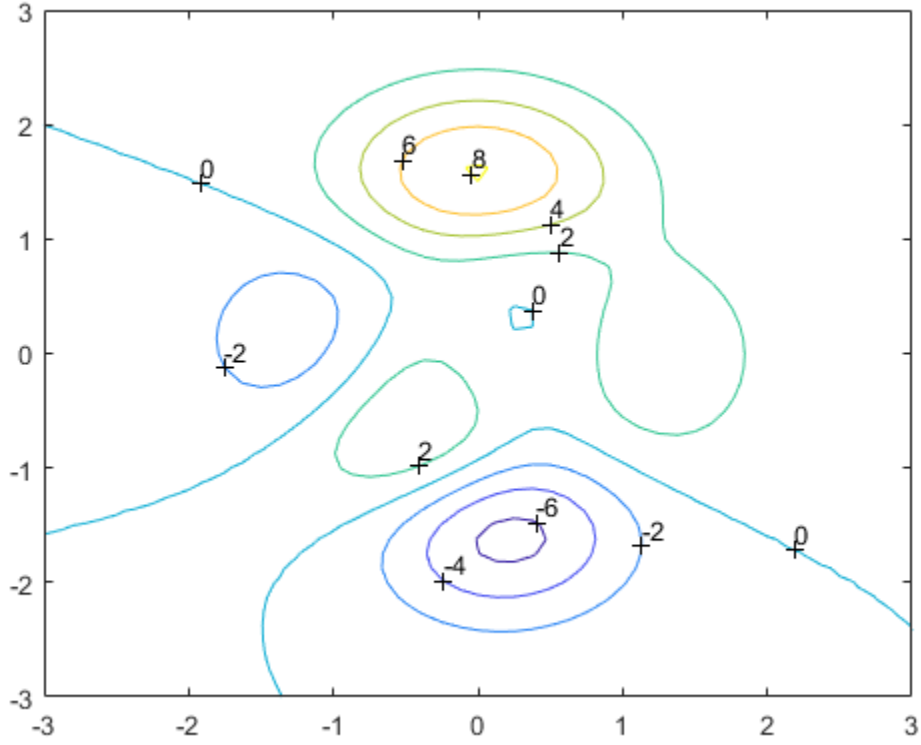
```
clabel(C,h,'FontSize','default')
```



Label Contour Plot with Vertical Text

Create a contour plot and return the contour matrix, C. Then, label the contours.

```
[x,y,z] = peaks;
C = contour(x,y,z);
clabel(C)
```



Input Arguments

C — Contour matrix
two-row matrix

Contour matrix returned by the `contour`, `contour3`, or `contourf` function. `C` contains the data that defines the contour lines. For more information on the contour matrix, see `ContourMatrix`.

Note If you pass the contour object `h` to the `clabel` function, then you can replace `C` with `[]`. For example, use `clabel([],h)`.

`h` — Contour object
contour object

Contour object returned by the `contour`, `contour3`, or `contourf` function.

`v` — Contour level values
vector

Contour level values, specified as a row or column vector of individual values.

Example: `[0 10 20]`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`clabel` supports the `LabelSpacing` property plus a subset of text properties.

Example: `clabel(C,h,'Color','red','FontSize',12)` specifies red, 12-point labels.

`LabelSpacing` — Space between labels
scalar

Space between labels, specified as a scalar value in point units.

Note Use this option with either the `clabel(C,h)` or `clabel(C,h,v)` syntax. Other syntaxes do not support this option.

Example: `clabel(C,h,'LabelSpacing',100)`

`Color` — Text color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Text color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: 'blue'

Example: [0 0 1]

Example: '#0000FF'

EdgeColor — Color of text box outline
'none' (default) | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...

Color of text box outline, specified as 'none', an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `clabel(C,h,'EdgeColor','k')`

`BackgroundColor` — Background color

`'none'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of text box background, specified as `'none'`, an RGB triplet, hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to

F. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `clabel(C,h,'BackgroundColor','g')`

`FontSize` — Font size

`10` (default) | scalar value greater than 0

Font size, specified as a scalar value greater than 0. By default, the font size uses point units. One point equals 1/72 inch. However, some syntaxes allow you to change the font units using the `FontUnits` property.

Example: `clabel(C,h,'FontSize',15)`

`FontName` — Font name
supported font name | `'FixedWidth'`

Font name, specified as a supported font name or `'FixedWidth'`. To display and print text properly, you must choose a font that your system supports. The default font depends on your operating system and locale.

To use a fixed-width font that looks good in any locale, use `'FixedWidth'`. The fixed-width font relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`FontAngle` — Character slant
`'normal'` (default) | `'italic'`

Character slant, specified as `'normal'` or `'italic'`.

Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.

`FontWeight` — Character thickness
`'normal'` (default) | `'bold'`

Character thickness, specified as `'normal'` or `'bold'`.

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.

`FontSmoothing` — Smooth font character appearance
`'on'` (default) | `'off'`

Smooth font character appearance, specified as one of these values:

- `'on'` — Apply font smoothing. Reduce the appearance of jaggedness in the text characters to make the text easier to read.

- `'off'` — Do not apply font smoothing.

Example: `clabel(C,h,'FontSmoothing','on')`

`FontUnits` — Font size units
`'points'` (default) | `'inches'` | `'centimeters'` | `'characters'` | `'normalized'` | `'pixels'`

Font size units, specified as one of the values in this table.

Units	Description
`'points'`	Points. One point equals 1/72 inch.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the default system font character size. • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.
`'normalized'`	Interpret font size as a fraction of the axes height. If you resize the axes, the font size modifies accordingly. For example, if the `FontSize` is `0.1` in normalized units, then the text is 1/10 of the axes height.
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.

If you set both the font size and the font units in one function call, you must set the `FontUnits` property first so that the axes correctly interprets the specified font size.

Note The `clabel(C,h)` and `clabel(C,h,v)` syntaxes do not support this option. They always use the default value of points.

Example: `clabel(C,'FontUnits','normalized')`

`Rotation` — Text orientation
`0` (default) | scalar

Text orientation, specified as a scalar value in degrees. The default rotation of 0 degrees makes the text horizontal. For vertical text, set this property to `90` or `-90`. Positive values rotate the text counterclockwise. Negative values rotate the text clockwise.

Note The `clabel(C,h)` and `clabel(C,h,v)` syntaxes do not support this option. Instead, they insert rotated text into the contour lines, with each label rotated to match the local orientation of the corresponding line.

Example: `clabel(C,'Rotation',90)`

`Interpreter` — Interpretation of text characters
`'tex'` (default) | `'latex'` | `'none'`

Interpretation of text characters, specified as one of these values:

- `'tex'` — Display text using TeX markup.
- `'latex'` — Display text using LaTeX markup.
- `'none'` — Display literal characters.

Since the labels are numeric text, the effect of this property is limited to subtle changes in the font style and weight.

Example: `clabel(C,h,'Interpreter','latex')`

`LineStyle` — Line style of text box outline
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style of text box outline, specified as one of the line styles in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

Example: `clabel(C,h,'LineStyle','--')`

`LineWidth` — Width of text box outline
`0.5` (default) | scalar

Width of text box outline, specified as a scalar value in point units. One point equals 1/72 inch.

Example: `clabel(C,h,'LineWidth',1)`

`Margin` — Space around text within the text box
`3` (default) | scalar

The space around the text within the text box, specified as scalar value in point units.

MATLAB uses the `Extent` property value plus the `Margin` property value to determine the size of the text box.

Example: `clabel(C,h,'Margin',4)`

Output Arguments

`t` — Text objects
vector

Text objects, returned as a vector. The `String` properties of the text objects contain the contour values displayed.

`tl` — Text and line objects
vector

Text and line objects, returned as a vector. The `String` properties of the text objects contain the contour values displayed. The line objects correspond to the `'+'` symbols.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions

`contour` | `contour3` | `contourc` | `contourf`

Properties

Text

Introduced before R2006a

class

Class of object

Syntax

```
className = class(obj)
```

Description

`className = class(obj)` returns the name of the class of `obj`.

Examples

Class of Java Object

Get the class name of a Java object.

```
jObject = java.lang.String('Java string');
className = class(jObject);
disp(className)
```

```
java.lang.String
```

Class of a Variable

Determine the class of a variable.

```
h = @sin;
className = class(h);
disp(className)
```

```
function_handle
```

Input Arguments

`obj` — Object or literal
variable or expression

Object or literal, specified as a variable or an expression.

Example: class(date)

Output Arguments

`className` — Name of the class
character vector

Name of the class, returned as a character vector.

More About

Obsolete Class Definition Syntax

In class definition before MATLAB 7.6 (classes defined without a `classdef` statement), class constructors called the `class` function to create the object. The following `class` function syntaxes apply only within classes defined before Version 7.6.

`obj = class(s,ClassName)` creates an array of objects of the specified class using the `struct` `s` as a pattern to determine the size of `obj`.

`obj = class(s,ClassName,parent1,parent2,...)` inherits the methods and fields of the specified parent objects. The size of the parent objects must match the size of `s` or be a scalar (1-by-1). If they are scalar, MATLAB performs scalar expansion.

`obj = class(struct([]),ClassName,parent1,parent2,...)` constructs an object containing only fields that it inherits from the parent objects. All parents must have the same nonzero size, which determines the size of the returned object.

`objStruct = class(structArray,ClassName,parentArray)` maps every element of the `parentArray` to a corresponding element in the `structArray` to produce the output array of objects, `objStruct`.

All arrays must be of the same size. If either the `structArray` or the `parentArray` is of size 1-by-1, then MATLAB performs scalar expansion to match the array sizes.

To create an object array of size 0-by-0, set the size of the `structArray` and `parentArray` to 0-by-0.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`isa` | `isobject` | `metaclass`

Topics

"Class Components"

Introduced before R2006a

classdef

Class definition keywords

Syntax

```
classdef ... end
```

Description

`classdef ... end` encloses a class definition.

Examples

Class Syntax

Use these keywords to define classes. For complete examples, see “Sample Class Implementations”.

```
classdef (Attributes) ClassName < SuperclassName
   properties (Attributes)
      PropertyName
      PropertyName size class {validation functions}
   end
   methods (Attributes)
      function obj = methodName(obj,arg2,...)
         ...
      end
   end
   events (Attributes)
      EventName
   end
end
```

```
classdef (Attributes) ClassName < SuperclassName
   enumeration
```

```
        EnumName
    end
end
```

`properties`, `methods`, `events`, and `enumeration` are also the names of MATLAB functions used to query the respective class members for a given object or class name.

More About

classdef

`classdef` is a keyword used to define MATLAB classes.

`classdef` *`ClassName`* begins the class definition and an `end` keyword terminates the `classdef` block. Only blank lines and comments can precede `classdef`. Enter a class definition in a file having the same name as the class, with a filename extension of `.m`. Valid class names begin with an alphabetic character, and can contain letters, numbers, or underscores.

`classdef` *`ClassName < SuperclassName1`* & *`SuperclassName2 ...`* begins the class definition and specifies one or more superclasses. For more information on deriving classes from other classes, see “Subclass Definition”.

`classdef` *`(AttributeName1 = attributevalue, AttributeName2 = attributevalue, ...) ClassName`* begins the class definition and specifies optional class attributes. For a list of class attributes, see “Class Attributes”.

Class definition files can be in folders on the MATLAB path or in class folders whose parent folder is on the MATLAB path. Class folder names begin with the `'@'` character followed by the class name (for example, `@MyClass`). For more information on class folders, see “Class Files and Folders” .

For more information on classes, see “Classdef Block” and “Class Definition”.

Properties

`properties` begins a property definition block; an `end` keyword terminates the `properties` block. Class definitions can contain multiple property definition blocks, each specifying different attribute settings that apply to the properties in that particular block.

For more information on properties, see “Property Syntax”.

Note Properties cannot have the same name as the class.

Methods

`methods` begins a methods definition block; an `end` keyword terminates the `methods` block. This block contains functions that implement class methods. Class definitions can contain multiple method blocks, each specifying different attribute settings that apply to the methods in that particular block. It is possible to define method functions in separate files.

For more information on methods, see “Methods in Class Design”.

Events

`events` begins an events definition block; an `end` keyword terminates the `events` block. This block contains event names defined by the class. Class definitions can contain multiple event blocks, each specifying different attribute settings that apply to the events in that particular block.

For more information on events, see “Events and Listeners Syntax”.

Enumeration

`enumeration` begins an enumeration definition block; an `end` keyword terminates the `enumeration` block.

For more information on enumerations, see “Enumerations”.

See Also

`events` | `methods` | `properties`

Topics

“Class Components”
“Attribute Specification”

“Sample Class Implementations”
“Methods in Separate Files”
“Validate Property Values”

Introduced in R2008a

classUnderlying

Class of underlying data in tall array

Syntax

```
C = classUnderlying(X)
```

Description

`C = classUnderlying(X)` returns the class of the data stored inside tall array `X`.

Examples

Determine Underlying Class of Tall Arrays

All tall tables and arrays belong to the `tall` class. However, the *underlying* data type of a tall array can vary.

Create a datastore for the `airlinesmall.csv` data set. Select a subset of the variables to work with, and treat `'NA'` values as missing data so that `datastore` replaces them with `NaN` values. Convert the datastore into a tall table.

```
varnames = {'Year', 'UniqueCarrier'};
ds = datastore('airlinesmall.csv','TreatAsMissing','NA',...
    'SelectedVariableNames',varnames);
tt = tall(ds)
```

```
tt =

  Mx2 tall table

    Year    UniqueCarrier
    ____    _____________

    1987       {'PS'}
```

```
    1987       {'PS'}
    1987       {'PS'}
    1987       {'PS'}
    1987       {'PS'}
    1987       {'PS'}
    1987       {'PS'}
    1987       {'PS'}
     :             :
     :             :
```

Determine the class of the tall table `tt` and the first table variable `Year`.

```
class(tt)
```

```
ans =
'tall'
```

```
class(tt.Year)
```

```
ans =
'tall'
```

Determine the underlying data types of `tt`, as well as the `Year` and `UniqueCarrier` table variables.

```
classUnderlying(tt)
```

```
ans =

  1x5 tall char array

    'table'
```

```
classUnderlying(tt.Year)
```

```
ans =

  1x6 tall char array

    'double'
```

```
classUnderlying(tt.UniqueCarrier)
```

```
ans =

  1x4 tall char array
```

```
'cell'
```

In some cases, the result returned by `classUnderlying` is an unevaluated tall array. Unevaluated tall arrays can be evaluated using the `gather` function to bring the result into memory.

Input Arguments

X — Input array
tall array

Input array, specified as a tall array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `table` | `cell` | `categorical` | `datetime` | `duration` | `calendarDuration`

Output Arguments

C — Underlying class
tall array

Underlying class, returned as a tall array.

Tips

- Use `isaUnderlying` to test whether a tall array has a particular underlying data type.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `classUnderlying` in the Parallel Computing Toolbox documentation.

See Also

`isaUnderlying` | `istall` | `tall`

Topics

"Tall Arrays for Out-of-Memory Data"

Introduced in R2016b

clc

Clear Command Window

Syntax

```
clc
```

Description

`clc` clears all the text from the Command Window, resulting in a clear screen. After running `clc`, you cannot use the scroll bar in the Command Window to see previously displayed text. You can, however, use the up-arrow key ↑ in the Command Window to recall statements from the command history.

Use `clc` in a MATLAB code file to always display output in the same starting position on the screen.

Examples

Clear All Output

Clear all text from the Command Window before running a command.

Create a 50-by-50 matrix of random numbers.

```
rand(50)
```

```
ans = 50×50

    0.8147    0.2760    0.1622    0.4173    0.6443    0.9631    0.0596    0.3015    0.4
    0.9058    0.6797    0.7943    0.0497    0.3786    0.5468    0.6820    0.7011    0.0
    0.1270    0.6551    0.3112    0.9027    0.8116    0.5211    0.0424    0.6663    0.5
    0.9134    0.1626    0.5285    0.9448    0.5328    0.2316    0.0714    0.5391    0.4
    0.6324    0.1190    0.1656    0.4909    0.3507    0.4889    0.5216    0.6981    0.6
    0.0975    0.4984    0.6020    0.4893    0.9390    0.6241    0.0967    0.6665    0.6
```

```
    0.2785    0.9597    0.2630    0.3377    0.8759    0.6791    0.8181    0.1781    0.6
    0.5469    0.3404    0.6541    0.9001    0.5502    0.3955    0.8175    0.1280    0.0
    0.9575    0.5853    0.6892    0.3692    0.6225    0.3674    0.7224    0.9991    0.0
    0.9649    0.2238    0.7482    0.1112    0.5870    0.9880    0.1499    0.1711    0.3
       ⋮
```

Clear all text from the Command Window.

```
clc
```

Create a 10-by-10 matrix of random numbers.

```
rand(10)
```

```
ans = 10×10

    0.8558    0.9390    0.9262    0.6279    0.4374    0.8884    0.8964    0.2442    0.8
    0.6708    0.8154    0.2985    0.4504    0.7462    0.2332    0.4822    0.4290    0.1
```

```
    0.5236    0.0014    0.3381    0.4736    0.4679    0.8616    0.0141    0.0102    0.8
    0.2988    0.0031    0.8595    0.9497    0.8608    0.7117    0.6229    0.6088    0.0
    0.7040    0.0875    0.3405    0.0835    0.4665    0.8728    0.2311    0.9580    0.3
    0.3816    0.2607    0.1381    0.2798    0.4981    0.9380    0.5274    0.0954    0.5
    0.5677    0.0228    0.5078    0.4470    0.4874    0.1397    0.7250    0.0356    0.5
    0.8879    0.4241    0.8567    0.5876    0.2295    0.3939    0.6074    0.8862    0.6
    0.8429    0.3411    0.3843    0.8776    0.0856    0.9806    0.5884    0.2469    0.6
    0.8988    0.5414    0.6957    0.4691    0.0674    0.6448    0.4334    0.0089    0.4
```

See Also

`clear` | `clf` | `close` | `home`

Topics

"Format Output"

Introduced before R2006a

clear

Remove items from workspace, freeing up system memory

Syntax

```
clear
clear name1 ... nameN
clear -regexp expr1 ... exprN
clear ItemType
```

Description

`clear` removes all variables from the current workspace, releasing them from system memory.

`clear name1 ... nameN` removes the variables, scripts, functions, or MEX-functions `name1 ... nameN` from memory.

`clear -regexp expr1 ... exprN` removes all variables that match any of the regular expressions listed. This option only removes variables.

`clear ItemType` removes the types of items indicated by `ItemType`, such as `all`, `functions`, or `classes`.

Examples

Clear a Single Variable

Define two variables `a` and `b`, and then clear `a`.

```
a = 1;
b = 2;
clear a
```

Only variable b remains in the workspace.

```
whos
```

```
  Name      Size            Bytes  Class     Attributes

  b         1x1                 8  double
```

Clear Specific Variables by Name

Using regular expressions, clear those variables with names that begin with Mon, Tue, or Wed.

```
clear -regexp ^Mon ^Tue ^Wed;
```

Clear Set of Variables

Create a cell array, vars, that contains the names of variables to clear. Then, clear those variables.

```
vars = {'v1','v2','time'};
clear(vars{:})
```

Clear All Compiled Scripts, Functions, and MEX-functions

If a function is locked or currently running, it is not cleared from memory.

```
clear functions
```

Input Arguments

name1 ... nameN — Names of variables, scripts, functions, or MEX-functions to clear

character vectors | string scalars

Names of variables, scripts, functions, or MEX-functions to clear, specified as one or more character vectors or string scalars.

If `name` is a:

- Function, then `clear` reinitializes any persistent variables in the function. Specifying a local or nested function is not supported.
- Script or function that is currently executing or a function locked by `mlock`, then `clear` does not remove it.
- Global variable, then `clear` removes it from the current workspace, but it remains in the global workspace. To remove a global variable from all workspaces, use `clear global` *`variable`*.

Use a partial path to distinguish between different overloaded versions of a function. For example, `clear polynom/display` clears only the `display` method for `polynom` objects, leaving any other implementations in memory.

`expr1 ... exprN` — Regular expressions matching names of variables to clear
character vectors | string scalars

Regular expressions matching names of variables to clear, specified as one or more character vectors or string scalars.

`ItemType` — Type of items to clear
`all` | `classes` | `functions` | `global` | `import` | `java` | `mex` | `variables`

Type of items to clear, specified as one of the following.

Value of `ItemType`	Items Cleared								Notes
	Variables in scope	Scripts and functions	Class definitions	Persistent variables	MEX functions	Global variables	Import list	Java classes on the dynamic path	
`all`	✓	✓		✓	✓	✓	From command prompt only		Calling `clear all` decreases code performance, and is usually unnecessary. For more information, see the “Tips” on page 1-1728 section.

Value of `ItemType`	Items Cleared								Notes
	Variables in scope	Scripts and functions	Class definitions	Persistent variables	MEX functions	Global variables	Import list	Java classes on the dynamic path	
`classes`	✓	✓	✓	✓	✓	✓	✓		Calling `clear classes` decreases code performance, and is usually unnecessary. For more information, see the “Tips” on page 1-1728 section. To instantiate an object with an updated class definition, it is not necessary to call `clear classes`. When you modify a class definition, MATLAB updates it automatically. For more information, see “Automatic Updates for Modified Classes”. `clear classes` issues a warning and does not clear a class of objects if any of those objects still exists after the workspace is cleared. For example, objects can still exist in persistent variables of functions or figure windows. `clear classes` does not clear a class if its file is locked using the `mlock`

Value of `ItemType`	Items Cleared								Notes
	Variables in scope	Scripts and functions	Class definitions	Persistent variables	MEX functions	Global variables	Import list	Java classes on the dynamic path	
									command. No warning is issued in this case.
`functions`		✓		✓	✓				Calling `clear functions` decreases code performance, and is usually unnecessary. For more information, see the "Tips" on page 1-1728 section.
`global`						✓			`clear global` removes all global variables in the current and global workspaces. For example, when called from a function, `clear global` removes all global variables in the function and global workspaces, but not in the base workspace.
`import`							✓		Call `clear import` only from the command prompt. Calling `clear import` in a function or a script returns an error.

Value of `ItemType`	Items Cleared								Notes
	Variables in scope	Scripts and functions	Class definitions	Persistent variables	MEX functions	Global variables	Import list	Java classes on the dynamic path	
`java`	✓	✓		✓	✓	✓		✓	`clear java` issues a warning and does not remove the Java class definition if any of its Java objects exist outside the workspace (for example, in user data or persistent variables in a locked code file). Issue a `clear java` command after modifying any files on the dynamic Java path.
`mex`					✓				`clear mex` does not clear locked MEX functions or functions that are currently in use.
`variables`	✓								

Note If the name of a variable is a value of `ItemType`, then calling `clear` followed by that name deletes the variable with that name. `clear` does not interpret the name as a keyword in this context. For example, if the workspace contains variables `a`, `all`, `b`, and `ball`, `clear all` removes the variable `all` only.

Tips

- Calling `clear all`, `clear classes`, and `clear functions` decreases code performance, and is usually unnecessary.
  - To clear one or more specific variables from the current workspace, use `clear name1 ... nameN`.
  - To clear all variables from the current workspace, use `clear` or `clearvars`.
  - To clear all global variables, use `clear global` or `clearvars -global`.
  - To clear a particular class, use `clear myClass`.
  - To clear a particular function or script, use `clear functionName`.
  - To clear all MEX functions, use `clear mex`.
- The `clear` function can remove variables that you specify. To remove all except a few specified variables, use `clearvars` instead.
- If you clear the handle of a figure or graphics object, the object itself is not removed. Use `delete` to remove objects. On the other hand, deleting an object does not remove the variable (if any) used for storing its handle.
- The `clear` function does not clear Simulink® models. Use `bdclose` instead.
- The `clear` function does not clear persistent variables in local or nested functions.
- On UNIX systems, `clear` does not affect the amount of memory allocated to the MATLAB process.

See Also

`clc` | `clearvars` | `delete` | `import` | `inmem` | `load` | `mlock` | `whos`

Topics

"Base and Function Workspaces"
"Strategies for Efficient Use of Memory"
"Automatic Updates for Modified Classes"
"Java Class Path"
"Regular Expressions"

Introduced before R2006a

clearAllMemoizedCaches

Clear caches for all `MemoizedFunction` objects

Syntax

```
clearAllMemoizedCaches
```

Description

`clearAllMemoizedCaches` clears caches for all `MemoizedFunction` objects.

Examples

Clear All Memoized Function Caches

Create two memoized functions, and call them several times.

```
mf1 = memoize(@sin);
mf2 = memoize(@cos);

a = [mf1(0) mf1(pi/2) mf1(pi)];
b = mf2(pi/2);
```

View the statistics for the memoized function objects.

```
mf1.stats.Cache
```

```
ans = struct with fields:
         Inputs: {{1x1 cell}  {1x1 cell}  {1x1 cell}}
        Nargout: [1 1 1]
        Outputs: {{1x1 cell}  {1x1 cell}  {1x1 cell}}
       HitCount: [0 0 0]
      TotalHits: 0
    TotalMisses: 3
```

```
mf2.stats.Cache
```

```
ans = struct with fields:
         Inputs: {{1x1 cell}}
        Nargout: 1
        Outputs: {{1x1 cell}}
       HitCount: 0
      TotalHits: 0
    TotalMisses: 1
```

Clear all memoized function caches, and view the statistics again.

```
clearAllMemoizedCaches
mf1.stats.Cache
```

```
ans = struct with fields:
         Inputs: {}
        Nargout: []
        Outputs: {}
       HitCount: []
      TotalHits: 0
    TotalMisses: 0
```

```
mf2.stats.Cache
```

```
ans = struct with fields:
         Inputs: {}
        Nargout: []
        Outputs: {}
       HitCount: []
      TotalHits: 0
    TotalMisses: 0
```

See Also

Functions
`clearCache` | `memoize`

Objects
`MemoizedFunction`

Introduced in R2017a

clearCache

Clear cache for `MemoizedFunction` object

Syntax

```
clearCache(mfcn)
```

Description

`clearCache(mfcn)` clears the cache for a `MemoizedFunction` object.

The memoization of a function is associated with the input function and not with the `MemoizedFunction` object. Therefore, clearing a variable does not clear the cache associated with the input function.

A `MemoizedFunction` object is not aware of updates to the underlying function. If you modify the function associated with the memoized function, use `clearCache` to clear the cache.

Examples

Clear Memoized Function Caches

Create a memoized function object for the `sin` function, and call it several times.

```
mf = memoize(@sin);
a = mf(0);
b = mf(pi/2);
c = mf(0);
```

View the statistics for the object.

```
mf.stats.Cache
```

```
ans = struct with fields:
         Inputs: {{1x1 cell}  {1x1 cell}}
```

```
        Nargout: [1 1]
        Outputs: {{1x1 cell}  {1x1 cell}}
       HitCount: [1 0]
      TotalHits: 1
    TotalMisses: 2
```

Clear the memoized function object cache, and view the statistics.

```
mf.clearCache
mf.stats.Cache
```

```
ans = struct with fields:
         Inputs: {}
        Nargout: []
        Outputs: {}
       HitCount: []
      TotalHits: 0
    TotalMisses: 0
```

Input Arguments

`mfcn` — Function with memoization semantics
`MemoizedFunction` object

Function with memoization semantics, specified as a `MemoizedFunction` object.

Tips

- Two variables that memoize the same function share a cache and object property values, such as cache size. In the following example, the variables `a` and `b` share a cache and have the same value for cache size.

  ```
  a = memoize(@svd);
  b = memoize(@svd);
  ```

  Clearing the cache for `b` (`b.clearCache`) also clears the cache for `a`, and any other variables that memoize the `svd` function.

- Clearing a variable does not clear the cache associated with the input function. To clear the cache for a `MemoizedFunction` object that no longer exists in the workspace, create a new `MemoizedFunction` object to the same function, and use the `clearCache` function on the new object. Alternatively, you can clear caches for all `MemoizedFunction` objects using the `clearAllMemoizedCaches` function.

See Also

Functions
`clearAllMemoizedCaches` | `memoize`

Objects
`MemoizedFunction`

Introduced in R2017a

clearPersonalValue

Package: `matlab.settings`

Clear the personal value for a setting

Syntax

```
clearPersonalValue(s)
```

Description

`clearPersonalValue(s)` clears the personal value for the specified setting. If the personal value is not set or not writeable, MATLAB throws an error.

Examples

Clear Setting Personal Value

Set and then clear the personal value for the code font size in MATLAB.

Get the root `SettingsGroup` object and set the personal value for the maximum column width for comments in MATLAB.

```
s = settings
s.matlab.editor.language.matlab.comments.MaxWidth.PersonalValue = 80;
```

Display the current values for the maximum column width for comments in MATLAB. Since the personal value is set and the temporary value is not set, the active value for the setting is the personal value. For more information on how the active value is determined, see “Access and Modify Settings”.

```
s.matlab.editor.language.matlab.comments.MaxWidth
```

```
ans = 
 Setting 'matlab.editor.language.matlab.comments.MaxWidth' with properties.
```

```
       ActiveValue: 80
    TemporaryValue: <no value>
     PersonalValue: 80
      FactoryValue: 75
```

Clear the personal value. The active value for the setting is now set to the factory value.

```
clearPersonalValue(s.matlab.editor.language.matlab.comments.MaxWidth)
```

```
ans = 
 Setting 'matlab.editor.language.matlab.comments.MaxWidth' with properties.

       ActiveValue: 75
    TemporaryValue: <no value>
     PersonalValue: <no value>
      FactoryValue: 75
```

Input Arguments

s — Setting
`setting` object

Setting, specified as a `setting` object. Use the `settings` function to access the root settings group object and all the available settings in the tree.

See Also

`addSetting` | `clearTemporaryValue` | `hasPersonalValue` | `settings`

Topics

"Access and Modify Settings"

Introduced in R2018a

clearpoints

Clear points from animated line

Syntax

```
clearpoints(an)
```

Description

`clearpoints(an)` clears all points from the animated line specified by `an`. Create an animated line with the `animatedline` function. If you want to display the update on the screen, use `drawnow` after using `clearpoints`.

Examples

Clear Points from Animated Line

Create an animated line with 10 points. Then, clear the points stored in the animated line.

```
an = animatedline(1:10,1:10);
clearpoints(an)
```

The animated line still exists, but has no data.

Input Arguments

`an` — `AnimatedLine` object
`AnimatedLine` object

`AnimatedLine` object. Create an `AnimatedLine` object using the `animatedline` function.

See Also

Functions
`addpoints` | `animatedline` | `getpoints`

Properties
AnimatedLine

Introduced in R2014b

clearTemporaryValue

Package: `matlab.settings`

Clear the temporary value for a setting

Syntax

```
clearTemporaryValue(s)
```

Description

`clearTemporaryValue(s)` clears the temporary value for the specified setting. If the temporary value is not set or not writeable, MATLAB throws an error.

Examples

Clear Setting Temporary Value

Set and then clear the temporary value for the code font size in MATLAB.

Get the root `SettingsGroup` object and set the temporary value for the maximum column width for comments in MATLAB.

```
s = settings
s.matlab.editor.language.matlab.comments.MaxWidth.TemporaryValue = 80;
```

Display the current values for the maximum column width for comments in MATLAB. Since the temporary value is set, the active value for the setting is the temporary value. For more information on how the active value is determined, see “Access and Modify Settings”.

```
s.matlab.editor.language.matlab.comments.MaxWidth
```

```
ans = 
 Setting 'matlab.editor.language.matlab.comments.MaxWidth' with properties.
```

```
       ActiveValue: 80
    TemporaryValue: 80
     PersonalValue: <no value>
      FactoryValue: 75
```

Clear the temporary value. The active value for the setting is now set to the factory value.

```
clearTemporaryValue(s.matlab.editor.language.matlab.comments.MaxWidth)
```

```
ans = 
 Setting 'matlab.editor.language.matlab.comments.MaxWidth' with properties.

       ActiveValue: 75
    TemporaryValue: <no value>
     PersonalValue: <no value>
      FactoryValue: 75
```

Input Arguments

s — Setting
`setting` object

Setting, specified as a `setting` object. Use the `settings` function to access the root settings group object and all the available settings in the tree.

See Also

`addSetting` | `clearPersonalValue` | `hasTemporaryValue` | `settings`

Topics

"Access and Modify Settings"

Introduced in R2018a

clearvars

Clear variables from memory

Syntax

```
clearvars
clearvars variables
clearvars -except keepVariables
clearvars variables -except keepVariables

clearvars -global ___
```

Description

`clearvars` removes all variables from the currently active workspace.

`clearvars variables` removes the variables specified by `variables`. If any of the variables are global, `clearvars` removes these variables from the current workspace only, leaving them accessible to any functions that declare them as global.

`clearvars -except keepVariables` removes all variables, except for those specified by `keepVariables`. Use this syntax to keep specific variables and remove all others.

`clearvars variables -except keepVariables` removes the variables specified by `variables`, and does not remove the variables specified by `keepVariables`. This syntax allows you to use a combination of variable names, wild card characters, or regular expressions to specify variables to remove or keep.

`clearvars -global ___` removes the specified global variables from the workspace, including those made global within functions, using any of the input arguments in the preceding syntaxes. The `-global` flag must be first in the argument list.

Examples

Clear Named Variables

Define three variables, a, b, and c. Then, clear a and c.

```
a = 1;
b = 2;
c = 3;
clearvars a c
whos
```

```
  Name      Size            Bytes  Class     Attributes

  b         1x1                 8  double
```

Only variable b remains in the workspace.

Clear All Variables Except Specified

Remove all variables from the workspace except for the variables C and D.

```
clearvars -except C D
```

Clear Variables Using Regular Expressions and Name Variables to Exclude

Clear variables with names that start with b and are followed by 3 digits, except for the variable b106.

```
clearvars -regexp ^b\d{3}$ -except b106
```

Name Variables to Clear and Preserve Variables Using Regular Expressions

Clear variables with names that start with a and do not end with a.

```
clearvars a* -except -regexp a$
```

Clear Global Variables Except Specified

Clear all global variables, except those with names that start with `x`.

```
clearvars -global -except x*
```

Clear List of Variables

Clear a list of variables used for intermediate calculations.

Create two variables in the workspace.

```
cashOnHand = 20;
cost = 12.99;
```

Store a list of the names of all the variables currently in the workspace.

```
initialVars = who;
```

Specify or calculate additional variables, `taxRate` and `tax`.

```
taxRate = 0.0625;
tax = round(100*cost*taxRate)/100;
```

Update the initial variables, `cost` and `cashOnHand`.

```
cost = cost + tax;
cashOnHand = cashOnHand - cost;
```

Clear all variables except the initial variables, using the function form of `clearvars`. When using the function form of a syntax, enclose input character vectors in single quotes, and separate them with commas.

```
clearvars('-except',initialVars{:})
```

`clearvars` clears the variables, `initialVars`, `taxRate`, and `tax`.

Input Arguments

`variables` — Names of variables to remove

character vectors | string scalars

Names of variables to remove, specified as one or more character vectors or string scalars in one of these forms.

Form of Variables Input	Variables to Remove
var1 ... varN	Named variables. Use the `'*'` wildcard to match patterns. For example, `clearvars A*` clears all variables in the workspace with names that start with `A`.
`-regexp` *expr1 ... exprN*	Variables with names that match the regular expressions. For example, `clearvars -regexp ^Mon ^Tues` clears only the variables in the workspace with names that begin with `Mon` or `Tues`.

Data Types: `char` | `string`

`keepVariables` — Names of variables to keep

character vectors | string scalar

Names of variables to keep, specified as one or more character vectors or string scalars, in one of these forms.

Form of Variables Input	Variables to Keep
var1 ... varN	Named variables. Use the `'*'` wildcard to match patterns. For example, `clearvars -except A*` clears all variables in the workspace, except those with names that start with `A`.
`-regexp` *expr1 ... exprN*	Variables with names that match the regular expressions. For example, `clearvars -except -regexp ^Mon ^Tues` clears all the variables in the workspace, except those with names that begin with `Mon` or `Tues`.

Data Types: `char` | `string`

See Also

`clear` | `exist` | `global` | `persistent` | `save` | `who` | `whos`

Topics

"Create and Edit Variables"
"Regular Expressions"
"Command vs. Function Syntax"

Introduced in R2008a

clf

Clear current figure window

Syntax

```
clf
clf('reset')
clf(fig)
clf(fig,'reset')
figure_handle = clf(...)
```

Description

`clf` deletes from the current figure all graphics objects whose handles are not hidden (i.e., their `HandleVisibility` property is set to `on`).

`clf('reset')` deletes from the current figure all graphics objects regardless of the setting of their `HandleVisibility` property and resets all figure properties except `Position`, `Units`, `PaperPosition`, and `PaperUnits` to their default values.

`clf(fig)` or `clf(fig,'reset')` clears the single figure with handle `fig`.

`figure_handle = clf(...)` returns the handle of the figure. This is useful when the figure `IntegerHandle` property is `off` because the noninteger handle becomes invalid when the reset option is used (i.e., `IntegerHandle` is reset to `on`, which is the default).

Alternatives

Use **Clear Figure** from the figure window's **Edit** menu to clear the contents of a figure. You can also create a *desktop shortcut* to clear the current figure with one mouse click. See “Rerun Favorite Commands”.

Tips

The `clf` command behaves the same way when issued on the command line as it does in callback routines — it does not recognize the `HandleVisibility` setting of `callback`. This means that when issued from within a callback routine, `clf` deletes only those objects whose `HandleVisibility` property is set to `on`.

See Also

`cla` | `clc` | `hold` | `reset`

Introduced before R2006a

clipboard

Copy and paste text to and from system clipboard

Syntax

```
clipboard('copy',data)
txt = clipboard('paste')
data = clipboard('pastespecial')
```

Description

`clipboard('copy',data)` copies `data` to the clipboard. If `data` is not a character array, it is converted using `mat2str`.

Note The `clipboard` function requires Oracle® Java software.

`txt = clipboard('paste')` returns the current contents of the clipboard as a character vector. If `clipboard` cannot convert the contents, `txt` is empty (`''`).

`data = clipboard('pastespecial')` imports the clipboard contents into an array using `uiimport`.

See Also

`Import Tool` | `load` | `mat2str`

Introduced before R2006a

clock

Current date and time as date vector

Syntax

```
c = clock
[c tf] = clock
```

Description

`c = clock` returns a six-element date vector containing the current date and time in decimal form:

```
[year month day hour minute seconds]
```

The `clock` function calculates the current date and time from the system time.

`[c tf] = clock` returns a second output argument that is `1` (`true`) if the current date and time occur during Daylight Saving Time (DST) in your system's time zone, and `0` (`false`) otherwise.

Examples

Round clock Output to Integer Display

To return the current date and time, use the `clock` function. Set the output format so that floating-point values display with up to five digits.

```
format shortg
c = clock
```

```
c = 1×6

        2019           8          27           3           5      48.553
```

The sixth element of the date vector output (seconds) is accurate to several digits beyond the decimal point.

To round to integer display format, use the `fix` function.

```
fix(c)
```

```
ans = 1×6

        2019           8          27           3           5          48
```

Test Whether Current Time Occurs During DST

To test whether the current date and time occur during Daylight Saving Time (DST), use the second output of the `clock` function. `tf` is `1` (`true`) if `c` occurs during DST, and `0` (`false`) otherwise.

```
format shortg
[c tf] = clock
```

```
c = 1×6

         2019           8          26          23          40       6.523


tf = logical
   1
```

Limitations

- MATLAB Online returns current Coordinated Universal Time (UTC) rather than local time.

Tips

- To time the duration of an event, use the `timeit` or `tic` and `toc` functions instead of `clock` and `etime`. The `clock` function is based on the system time, which can be adjusted periodically by the operating system, and thus might not be reliable in time comparison operations.
- To return a datetime scalar representing the current date and time, type:

```
t = datetime('now')
```

See Also

`cputime` | `date` | `datetime` | `etime` | `fix` | `now` | `tic` | `timeit` | `toc`

Introduced before R2006a

close

Remove specified figure

Syntax

```
close
close(h)
close name
close all
close all hidden
close all force
status = close(...)
```

Description

`close` deletes the current figure or the specified figure(s). It optionally returns the status of the close operation.

`close` deletes the current figure (equivalent to `close(gcf)`).

`close(h)` deletes the figure identified by `h`. If `h` is an array, `close` deletes all figures identified by `h`. `h` can also be the figure `Number`.

`close name` deletes the figure with the specified name.

`close all` deletes all figures whose handles are not hidden.

`close all hidden` deletes all figures including those with hidden handles.

`close all force` deletes all figures, including GUIs for which `CloseRequestFcn` has been altered to not close the window.

`status = close(...)` returns `1` if the specified windows have been deleted and `0` otherwise.

Algorithms

The `close` function works by evaluating the specified figure's `CloseRequestFcn` property with the statement

```
eval(get(h,'CloseRequestFcn'))
```

The default `CloseRequestFcn`, `closereq`, deletes the current figure using `delete(get(groot,'CurrentFigure'))`. If you specify an array of figure handles, `close` executes each figure's `CloseRequestFcn` in turn. If an error that terminates the execution of a `CloseRequestFcn` occurs, the figure is not deleted. Note that using your computer's window manager (i.e., the **Close** menu item) also calls the figure's `CloseRequestFcn`.

If a figure's handle is hidden (i.e., the figure's `HandleVisibility` property is set to `callback` or `off` and the root `ShowHiddenHandles` property is set to `on`), you must specify the `hidden` option when trying to access a figure using the `all` option.

To delete all figures unconditionally, use the statements

```
set(groot,'ShowHiddenHandles','on')
c = get(groot,'Children');
delete(c)
```

The figure `CloseRequestFcn` allows you to either delay or abort the closing of a figure once the `close` function has been issued. For example, you can display a dialog box to see if the user really wants to delete the figure or save and clean up before closing.

When coding a `CloseRequestFcn` callback, make sure that it does not call `close`, because this sets up a recursion that results in a MATLAB warning. Instead, the callback should destroy the figure with `delete`. The `delete` function does not execute the figure's `CloseRequestFcn`; it deletes the specified figure.

See Also

`delete` | `figure` | `gcf`

Introduced before R2006a

close

Close Tiff object

Syntax

```
close(t)
```

Description

`close(t)` closes a `Tiff` object.

Examples

Close Tiff Object

Create a `Tiff` object for a file, and then close it.

```
t = Tiff('example.tif','r');
close(t);
```

Input Arguments

t — Tiff object
`Tiff` object

`Tiff` object representing a TIFF file. Use the `Tiff` function to create the object.

Algorithms

References

This function corresponds to the `TIFFClose` function in the LibTIFF C API. To use this function, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff`

Topics

"Importing Images"
"Exporting to Images"

Introduced in R2009b

close

Close connection to FTP server

Syntax

```
close(ftpobj)
```

Description

`close(ftpobj)` closes the connection to the FTP server associated with `ftpobj`.

Examples

Close Connection

Open a connection to an FTP server. Then close it when your session is finished.

First, connect to the National Centers for Environmental Information (NCEI) FTP server.

```
ftpobj = ftp('ftp.ngdc.noaa.gov')
```

```
ftpobj = 

  FTP Object
     host: ftp.ngdc.noaa.gov
     user: anonymous
      dir: /
     mode: binary
```

Show the contents on the server, and retrieve the `README.txt` file.

```
dir(ftpobj)
```

```
DMSP                           Solid_Earth                  google12c4c939d7b90761.html
```

```
INDEX.txt                    coastwatch                   hazards
README.txt                   dmsp4alan                    index.html
STP                          ftp.html                     international
Snow_Ice                     geomag                       ionosonde
```

```
mget(ftpobj,'README.txt');
```

Close the connection to the FTP server.

```
close(ftpobj)
```

FTP service courtesy of the National Centers for Environmental Information (NCEI). See the NCEI Privacy Policy, Disclaimer, and Copyright for NCEI terms of service.

Input Arguments

`ftpobj` — Connection to FTP server
FTP object

Connection to an FTP server, specified as an FTP object.

Tips

- If you do not run `close` at the end of your session, then the connection either times out automatically or terminates when you exit MATLAB.
- After calling `close`, calling any other FTP function on the same object automatically reopens the connection.
- `close` does not return any output to indicate success or failure.

See Also

`ftp`

Introduced before R2006a

clibIsNull

Determine if C++ object is null

Syntax

```
tf = clibIsNull(cppObj)
```

Description

`tf = clibIsNull(cppObj)` returns logical `1` (`true`) if `cppObj` is `nullptr`. Otherwise, it returns logical `0` (`false`).

Examples

Test For `nullptr` Object

Suppose that you create an interface to library `nullptr`, built from this `nullptr.hpp` header file, containing a function that returns NULL.

```
class A {
public:
    double val;
};

// Function returning nullptr object
A* returnObjectNullptr() {
    return nullptr;
}
```

Display `nullptr` for an object of class `A`. The MATLAB code is used as an illustration and does not execute as is, unless you build the interface.

```
ret = clib.nullptr.returnObjectNullptr
```

```
ret =
    null A
```

Test for `nullptr`.

```
if clibIsNull(ret)
% Code to process nullptr
end;
```

Input Arguments

`cppObj` — C++ object created in MATLAB
handle

C++ object created in MATLAB, specified as a handle

See Also

Introduced in R2019b

clibIsReadOnly

Determine if C++ object is read-only

Syntax

```
tf = clibIsReadOnly(cppObj)
```

Description

`tf = clibIsReadOnly(cppObj)` returns logical `1` (`true`) if `cppObj` is read-only. Otherwise, it returns logical `0` (`false`).

Input Arguments

`cppObj` — C++ object created in MATLAB
handle

C++ object created in MATLAB, specified as a handle

See Also

Introduced in R2019b

clibRelease

Release C++ object from MATLAB

Syntax

```
clibRelease(cppObj)
```

Description

`clibRelease(cppObj)` releases C++ object from MATLAB, making it inaccessible.

Input Arguments

`cppObj` — C++ object created in MATLAB
handle

C++ object created in MATLAB, specified as a handle. Once released, the underlying C++ object is no longer accessible in MATLAB, and the MATLAB handle `cppObj` becomes invalid. If the object is returned again from the library, then MATLAB creates a handle.

See Also

Introduced in R2019a

clibgen Package

Summary of packages and classes to support calling C++ library functionality from MATLAB

Description

Use these classes to publish a MATLAB interface to a C++ shared library. If you already have a MATLAB interface, then see “Call Functions in C++ Shared Library”.

Classes

clibgen.ClassDefinition	Definition for MATLAB class corresponding to C++ class
clibgen.ConstructorDefinition	Definition for MATLAB function corresponding to C++ class constructor
clibgen.EnumDefinition	Definition for MATLAB enumeration corresponding to C++ enumeration
clibgen.FunctionDefinition	Definition for MATLAB function corresponding to C++ package function
clibgen.LibraryDefinition	MATLAB definition for C++ library
clibgen.MethodDefinition	Definition for MATLAB function corresponding to C++ class method
clibgen.PropertyDefinition	Definition for MATLAB property corresponding to public property of C++ class

Functions

clibgen.buildInterface	Create interface to C++ library without definition file
clibgen.generateLibraryDefinition	Create definition file for C++ library

See Also

Topics

“Steps to Publish a MATLAB C++ Library Interface”

clibgen.ClassDefinition class

Package: `clibgen`

Definition for MATLAB class corresponding to C++ class

Description

The `clibgen.ClassDefinition` class contains the MATLAB definition for a class defined in the C++ header.

The `clibgen.ClassDefinition` class is a `handle` class.

Creation

MATLAB constructs a `ClassDefinition` object when you call `addClass` on a `clibgen.LibraryDefinition` object.

Properties

`Description` — Help text
string | character vector

Help text for the end user describing the class, specified as a string or a character vector. The default text is

`help clib.`*`packageName`*`.`*`fullMLClassName`*

`Representation of a MATLAB C++ object clib.`*`packageName`*`.`*`fullMLClassName`*

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`MATLABName` — C++ class name in MATLAB
string | character vector

C++ class name in MATLAB, specified as a string or a character vector.

Attributes:

GetAccess	public
SetAccess	private

CPPName — Corresponding C++ class name
string | character vector

Corresponding C++ class name, specified as a string or a character vector.

Attributes:

GetAccess	public
SetAccess	private

Methods — Class methods
clibgen.MethodDefinition

Class methods, specified as one or more `clibgen.MethodDefinition` objects.

Attributes:

GetAccess	public
SetAccess	private

Constructors — Class constructors
clibgen.ConstructorDefinition

Class constructors, specified as one or more `clibgen.ConstructorDefinition` objects.

Attributes:

GetAccess	public
SetAccess	private

Properties — Class properties
clibgen.PropertyDefinition

Class properties, specified as one or more `clibgen.PropertyDefinition` objects.

Attributes:

GetAccess	public
SetAccess	private

DefiningLibrary — Library defining class
clibgen.LibraryDefinition

Library defining the class, specified as a `clibgen.LibraryDefinition` object.

Attributes:

GetAccess	public
SetAccess	private

Methods

Public Methods

addConstructor	Add constructor to C++ class definition
addMethod	Add method to C++ class definition
addProperty	Add property to C++ class definition

See Also

`addClass` | `clibgen.ConstructorDefinition` | `clibgen.LibraryDefinition` | `clibgen.MethodDefinition` | `clibgen.PropertyDefinition`

Introduced in R2019a

addConstructor

Class: `clibgen.ClassDefinition`
Package: `clibgen`

Add constructor to C++ class definition

Syntax

```
definition = addConstructor(classDef,cppSignature)
definition = addConstructor(classDef,cppSignature,'Description',
text)
```

Description

`definition = addConstructor(classDef,cppSignature)` adds a `clibgen.ConstructorDefinition` object with C++ signature `cppSignature` to the class definition.

`definition = addConstructor(classDef,cppSignature,'Description', text)` adds help text to the constructor definition.

Input Arguments

`classDef` — Class definition
`clibgen.ClassDefinition`

Class definition, specified as a `clibgen.ClassDefinition` object.

`cppSignature` — C++ constructor signature
string scalar | character vector

C++ constructor signature, specified as a string scalar or a character vector.

`text` — Help text
string scalar | character vector

Help text for the end user describing the constructor, specified as a string scalar or a character vector.

See Also

`clibgen.ClassDefinition` | `clibgen.ConstructorDefinition`

Introduced in R2019a

addMethod

Class: `clibgen.ClassDefinition`
Package: `clibgen`

Add method to C++ class definition

Syntax

```
definition = addMethod(classDef,cppSignature)
definition = addMethod(classDef,cppSignature,'Description',text)
```

Description

`definition = addMethod(classDef,cppSignature)` adds a `clibgen.MethodDefinition` object with C++ signature `cppSignature` to the class definition.

`definition = addMethod(classDef,cppSignature,'Description',text)` adds help text to the method definition.

Input Arguments

`classDef` — Class definition
`clibgen.ClassDefinition`

Class definition, specified as a `clibgen.ClassDefinition` object.

`cppSignature` — C++ method signature
string scalar | character vector

C++ method signature, specified as a string scalar or a character vector.

`text` — Help text
string scalar | character vector

Help text for the end user describing the method, specified as a string scalar or a character vector.

See Also

`clibgen.ClassDefinition` | `clibgen.MethodDefinition`

Introduced in R2019a

addProperty

Class: `clibgen.ClassDefinition`
Package: `clibgen`

Add property to C++ class definition

Syntax

```
definition = addProperty(classDef,cppName,MATLABType)
definition = addProperty(classDef,cppName,'Description',text)
```

Description

`definition = addProperty(classDef,cppName,MATLABType)` adds a `clibgen.PropertyDefinition` object to the class definition.

`definition = addProperty(classDef,cppName,'Description',text)` adds help text to the method definition.

Input Arguments

`classDef` — Class definition
`clibgen.ClassDefinition`

Class definition, specified as a `clibgen.ClassDefinition` object.

`cppName` — C++ property name
string scalar | character vector

C++ property name, specified as a string scalar or a character vector.

`MATLABType` — MATLAB type
valid data type

MATLAB type, specified as a valid data type for the property.

`text` — Help text
string scalar | character vector

Help text for the end user describing the property, specified as a string scalar or a character vector.

See Also

`clibgen.ClassDefinition` | `clibgen.PropertyDefinition`

Introduced in R2019a

clibgen.ConstructorDefinition class

Package: `clibgen`

Definition for MATLAB function corresponding to C++ class constructor

Description

The `clibgen.ConstructorDefinition` class contains the MATLAB definition for a constructor defined in the header of a C++ class.

The `clibgen.ConstructorDefinition` class is a `handle` class.

Creation

MATLAB constructs a `ConstructorDefinition` object when you call `addConstructor` on a `clibgen.ClassDefinition` object.

Properties

`Description` — Help text
string | character vector

Help text for the end user describing the constructor, specified as a string or a character vector. The default text is

```
help clib.packageName.fullMLClassName
```

```
Constructor of class clib.packageName.fullMLClassName
```

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Arguments` — Definition for input arguments
struct

Definition for input arguments, specified as a struct.

Attributes:

```
GetAccess                                public
SetAccess                                private
```

`CPPSignature` — C++ constructor signature

string | character vector

C++ constructor signature, specified as a string or a character vector.

Attributes:

```
GetAccess                                public
SetAccess                                private
```

`DefiningClass` — Class containing constructor

`clibgen.ClassDefinition`

Class containing the constructor, specified as a `clibgen.ClassDefinition` object.

Attributes:

```
GetAccess                                public
SetAccess                                private
```

`MATLABSignature` — Constructor signature in MATLAB

string | character vector

Constructor signature in MATLAB, specified as a string or a character vector. MATLAB displays this signature when you load the library.

Attributes:

```
GetAccess                                public
SetAccess                                private
Dependent                                true
```

`Valid` — Definition complete indicator

`false` (default) | `true`

Indicate if constructor definition is complete, specified as `true` or `false`.

Attributes:

`GetAccess`	`public`
`SetAccess`	`private`

Methods

Public Methods

defineArgument	Add input argument definition for C++ constructor
validate	Validate C++ constructor definition

See Also

`addConstructor` | `clibgen.ClassDefinition`

Introduced in R2019a

defineArgument

Class: `clibgen.ConstructorDefinition`
Package: `clibgen`

Add input argument definition for C++ constructor

Syntax

```
defineArgument(constructDef,argName,mltype)
defineArgument(constructDef,argName,mltype,direction)
defineArgument(constructDef,argName,mltype,direction,shape)
```

Description

`defineArgument(constructDef,argName,mltype)` adds an input argument definition.

`defineArgument(constructDef,argName,mltype,direction)` defines whether argument is input, output, or both.

`defineArgument(constructDef,argName,mltype,direction,shape)` provides information about data dimensions.

Input Arguments

`constructDef` — Constructor definition
`clibgen.ConstructorDefinition`

Constructor definition, specified as a `clibgen.ConstructorDefinition` object.

`argName` — C++ argument name
string scalar | character vector

C++ argument name, specified as a string scalar or a character vector.

`mltype` — MATLAB type
string scalar | character vector

MATLAB type, specified as a string scalar or a character vector.

`direction` — Argument type
`"input"` (default) | `"output"` | `"inputoutput"`

Argument type, specified as `"input"` for an input argument, `"output"` for an output argument, or `"inputoutput"` for both an input and an output argument.

`shape` — Information for defining data dimensions
string vector | scalar text | positive integer vector | `"nullTerminated"` | cell array

Dimension definition used by MATLAB to define data dimensions, specified as a string vector, scalar text, positive integer vector, `"nullTerminated"`, or a cell array. For a cell array, the elements are a combination of scalar text and scalar positive integers.

If you can define the argument dimension as a fixed scalar value, then enter a number, such as `5`.

If the dimension is defined by another argument, then enter the argument name as a string. For example, consider the following C++ signature. If argument `len` defines the length of `data`, then the value of `shape` is `"len"`.

```
myFunc(const int *data, int len)
```

If the size of an argument is defined by an array, then the value of `shape` is one of the following:

- `1`
- Fixed dimensions: Enter a numerical array, such as `[5,2]`.
- Variable dimensions: Enter a string array of parameter names, such as `["row","col"]`

If the C++ type for the argument is a string, then use these options to choose values for the `mltype` and `shape` arguments.

C++ Type	`mltype`	Options for `shape`
`char *`	`int8`	Scalar value Array of scalar values

C++ Type	mltype	Options for shape
`const char *`	`char`	MATLAB `char` Character vector
	`string`	`"nullTerminated"`

For more information, see “MATLAB to C++ Data Type Mapping”.

See Also

Introduced in R2019a

validate

Class: `clibgen.ConstructorDefinition`
Package: `clibgen`

Validate C++ constructor definition

Syntax

```
validate(constructDef)
```

Description

`validate(constructDef)` validates the number of arguments and the outputs of the function to ensure that they match the C++ signature for a constructor definition.

Input Arguments

`constructDef` — Constructor definition
`clibgen.ConstructorDefinition`

Constructor definition, specified as a `clibgen.ConstructorDefinition` object.

See Also

Introduced in R2019a

clibgen.EnumDefinition class

Package: `clibgen`

Definition for MATLAB enumeration corresponding to C++ enumeration

Description

The `clibgen.EnumDefinition` class contains the MATLAB definition for an enumeration defined in the C++ header.

The `clibgen.EnumDefinition` class is a `handle` class.

Creation

MATLAB constructs an `EnumDefinition` object when you call `addEnumeration` on a `clibgen.LibraryDefinition` object.

Properties

`Description` — Help text

string | character vector

Help text for the end user describing the enumeration, specified as a string or a character vector.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`CPPName` — C++ enumeration name

string | character vector

C++ enumeration name, specified as a string or a character vector.

Attributes:

```
GetAccess                                   public
SetAccess                                   private
```

`DefiningLibrary` — Library containing enumeration
`clibgen.LibraryDefinition`

Library containing the enumeration, specified as a `clibgen.LibraryDefinition` object.

Attributes:

```
GetAccess                                   public
SetAccess                                   private
```

`Entries` — Enumeration values
nested cell array

Enumeration values, specified as a nested cell array.

Attributes:

```
GetAccess                                   public
SetAccess                                   private
```

`MATLABName` — Value of `MATLABType`
string | character vector

Value of `MATLABType` specified in the XML, specified as a string or a character vector.

Attributes:

```
GetAccess                                   public
SetAccess                                   private
```

`MATLABType` — MATLAB type
`int32`

MATLAB type for enumeration values, specified as `int32`.

Attributes:

GetAccess	public
SetAccess	private

`Valid` — Definition complete indicator
`false` (default) | `true`

Indicate if enumeration definition is complete, specified as `true` or `false`.

Attributes:

GetAccess	public
SetAccess	private

Tips

- You might need to create an enumerated value in MATLAB, but the name of that value might not be a valid MATLAB name. For example, the enumerant name might begin with an underscore. To derive a value from this name at run time, use this MATLAB syntax, where *`enumMember`* is a string scalar or character vector that, when evaluated, returns an instance of an enumeration.

  ```
  clib.libName.enumName.(enumMember)
  ```

  For example, suppose that you have `clib.enums.keywords` with these properties:

  ```
  EnumDefinition with properties:

        Description: "clib.enums.keywords    Representation of C++ enumeration keywords"
  DefiningLibrary: [1×1 clibgen.LibraryDefinition]
          CPPName: "keywords"
        MATLABType: "int32"
             Valid: 1
        MATLABName: "clib.enums.keywords"
           Entries: ["_for"    "_while"    "_class"    "_enums"    "_template"    "_typename"]
  ```

  To assign entry `_class` to a variable, type:

  ```
  var = clib.enums.keywords.('_class');
  ```

  This syntax is valid for names less than the maximum identifier length `namelengthmax`.

See Also

`clibgen.LibraryDefinition`

Introduced in R2019a

clibgen.FunctionDefinition class

Package: `clibgen`

Definition for MATLAB function corresponding to C++ package function

Description

The `clibgen.FunctionDefinition` class contains the MATLAB definition for a function defined in the C++ header.

The `clibgen.FunctionDefinition` class is a `handle` class.

Creation

MATLAB constructs a `FunctionDefinition` object when you call `addFunction` on a `clibgen.LibraryDefinition` object.

Properties

`Description` — Help text
string | character vector

Help text for the end user describing the package function, specified as a string or a character vector. The default text is

`help clib.`*`packageName`*`.`*`fullMLFuncName`*

`C++ Function clib.`*`packageName`*`.`*`fullMLFuncName`*

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`CPPSignature` — C++ function signature
string | character vector

C++ function signature, specified as a string or a character vector.

Attributes:

GetAccess	public
SetAccess	private

DefiningLibrary — Library containing function
clibgen.LibraryDefinition

Library containing the function, specified as a `clibgen.LibraryDefinition` object.

Attributes:

GetAccess	public
SetAccess	private

MATLABName — C++ function name in MATLAB
string | character vector

C++ function name in MATLAB, specified as a string or a character vector.

Attributes:

GetAccess	public
SetAccess	private

MATLABSignature — Function signature in MATLAB
string | character vector

Function signature in MATLAB, specified as a string or a character vector. MATLAB displays this signature when you load the library.

Attributes:

GetAccess	public
SetAccess	private
Dependent	true

Valid — Definition complete indicator
`false` (default) | `true`

Indicate if function definition is complete, specified as `true` or `false`.

Attributes:

`GetAccess`	`public`
`SetAccess`	`private`

Methods

Public Methods

defineArgument	Add input argument definition for C++ package function
defineOutput	Add output argument definition for C++ package function
validate	Validate function definition

See Also

`addFunction` | `clibgen.LibraryDefinition`

Introduced in R2019a

defineArgument

Class: `clibgen.FunctionDefinition`
Package: `clibgen`

Add input argument definition for C++ package function

Syntax

```
defineArgument(funcDef,argName,mltype)
defineArgument(funcDef,argName,mltype,direction)
defineArgument(funcDef,argName,mltype,direction,shape)
```

Description

`defineArgument(funcDef,argName,mltype)` adds an input argument definition.

`defineArgument(funcDef,argName,mltype,direction)` defines whether argument is input, output, or both.

`defineArgument(funcDef,argName,mltype,direction,shape)` provides information about data dimensions.

Input Arguments

`funcDef` — Function definition
`clibgen.FunctionDefinition`

Function definition, specified as a `clibgen.FunctionDefinition` object.

`argName` — C++ argument name
string scalar | character vector

C++ argument name, specified as a string scalar or a character vector.

`mltype` — MATLAB type
string scalar | character vector

MATLAB type, specified as a string scalar or a character vector.

`direction` — Argument type
`"input"` (default) | `"output"` | `"inputoutput"`

Argument type, specified as `"input"` for an input argument, `"output"` for an output argument, or `"inputoutput"` for both an input and an output argument.

`shape` — Dimension definition
string vector | scalar text | positive integer vector | `"nullTerminated"` | cell array

Dimension definition used by MATLAB to define data dimensions, specified as a string vector, scalar text, positive integer vector, `"nullTerminated"`, or a cell array. For a cell array, the elements are a combination of scalar text and scalar positive integers.

If you can define the argument dimension as a fixed scalar value, then enter a number, such as `5`.

If the dimension is defined by another argument, then enter the argument name as a string. For example, consider the following C++ signature. If argument `len` defines the length of `data`, then the value of `shape` is `"len"`.

```
myFunc(const int *data, int len)
```

If the size of an argument is defined by an array, then the value of `shape` is one of the following:

- `1`
- Fixed dimensions: Enter a numerical array, such as `[5,2]`.
- Variable dimensions: Enter a string array of parameter names, such as `["row","col"]`

If the C++ type for the argument is a string, then use these options to choose values for the `mltype` and `shape` arguments.

C++ Type	`mltype`	Options for `shape`
`char *`	`int8`	Scalar value Array of scalar values

C++ Type	mltype	Options for shape
`const char *`	`char`	MATLAB `char` Character vector
	`string`	`"nullTerminated"`

For more information, see “MATLAB to C++ Data Type Mapping”.

See Also

`defineOutput`

Introduced in R2019a

defineOutput

Class: `clibgen.FunctionDefinition`
Package: `clibgen`

Add output argument definition for C++ package function

Syntax

```
defineOutput(funcDef,argName,mltype)
defineOutput(funcDef,argName,mltype,shape)
```

Description

`defineOutput(funcDef,argName,mltype)` adds an output argument definition.

`defineOutput(funcDef,argName,mltype,shape)` provides information about data dimensions.

Input Arguments

`funcDef` — Function definition
`clibgen.FunctionDefinition`

Function definition, specified as a `clibgen.FunctionDefinition` object.

`argName` — C++ output argument name
`"RetVal"` (default) | string scalar | character vector

C++ output argument name, specified as a string scalar or a character vector.

`mltype` — MATLAB type
string scalar | character vector

MATLAB type, specified as a string scalar or a character vector.

shape — Dimension definition

string vector | scalar text | positive integer vector | `"nullTerminated"` | cell array

Dimension definition used by MATLAB to define data dimensions, specified as a string vector, scalar text, positive integer vector, `"nullTerminated"`, or a cell array. For a cell array, the elements are a combination of scalar text and scalar positive integers.

If you can define the argument dimension as a fixed scalar value, then enter a number, such as `5`.

If the dimension is defined by another argument, then enter the argument name as a string. For example, consider the following C++ signature. If argument `len` defines the length of `data`, then the value of `shape` is `"len"`.

```
myFunc(const int *data, int len)
```

If the size of an argument is defined by an array, then the value of `shape` is one of the following:

- `1`
- Fixed dimensions: Enter a numerical array, such as `[5,2]`.
- Variable dimensions: Enter a string array of parameter names, such as `["row","col"]`

If the C++ type for the argument is a string, then use these options to choose values for the `mltype` and `shape` arguments.

C++ Type	mltype	Options for shape
`char *`	`int8`	Scalar value Array of scalar values
`const char *`	`char`	MATLAB `char` Character vector
	`string`	`"nullTerminated"`

For more information, see “MATLAB to C++ Data Type Mapping”.

See Also

`defineArgument`

Introduced in R2019a

validate

Class: `clibgen.FunctionDefinition`
Package: `clibgen`

Validate function definition

Syntax

```
validate(funcDef)
```

Description

`validate(funcDef)` validates the number of arguments and the outputs of the function to ensure that they match the C++ signature.

Input Arguments

`funcDef` — Function definition
`clibgen.FunctionDefinition`

Function definition, specified as a `clibgen.FunctionDefinition` object.

See Also

Introduced in R2019a

clibgen.LibraryDefinition class

Package: `clibgen`

MATLAB definition for C++ library

Description

A `clibgen.LibraryDefinition` object contains information about a C++ library, such as the classes, functions, methods, and properties present in the library. The definitions help MATLAB identify information about parameters, such as pointers.

The `clibgen.LibraryDefinition` class is a `handle` class.

Creation

`libdef = LibraryDefinition(dataFile)` creates a library definition object from `dataFile.xml`, the name of the data file created by the `clibgen.generateLibraryDefinition` command. The constructor validates that the `dataFile` file is both syntactically and semantically valid. If the file is not valid, then the constructor throws an error.

Input Arguments

`dataFile` — XML file name
string | character vector

XML file name, specified as a string or a character vector, created by `clibgen.generateLibraryDefinition`. The name includes the extension `.xml`.

Properties

`OutputFolder` — Folder for interface files
string | character vector

Folder for interface files, specified as a string or a character vector.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Libraries` — Library names

string | character vector

Library names, specified as a string or a character vector.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`IncludePath` — Path for include folders

string | character vector

Path for include folders, specified as a string or a character vector.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Classes` — Class definitions

`clibgen.ClassDefinition`

Class definitions in the library, specified as one or more `clibgen.ClassDefinition` objects.

Attributes:

```
GetAccess                                 public
SetAccess                                 private
```

`Enumerations` — Enumeration definitions

`clibgen.EnumDefinition`

Enumeration definitions in the library, specified as one or more `clibgen.EnumDefinition` objects.

Attributes:

```
GetAccess                                   public
SetAccess                                   private
```

`Functions` — Nonmember function definitions

`clibgen.FunctionDefinition`

Nonmember function definitions in the library, specified as one or more `clibgen.FunctionDefinition` objects.

Attributes:

```
GetAccess                                   public
SetAccess                                   private
```

`HeaderFiles` — Header file names

string | character vector

Header file names to use in the library interface, specified as a string or a character vector.

Attributes:

```
GetAccess                                   public
SetAccess                                   private
```

`PackageName` — MATLAB package name

string | character vector

MATLAB package name for the library, specified as a string or a character vector.

Attributes:

```
GetAccess                                   public
SetAccess                                   private
```

`DefinedMacros` — Macro definitions

string | character vector

List of macro definitions to use while parsing header files, specified as a string or a character vector.

Attributes:

GetAccess	public
SetAccess	private

UndefinedMacros — Macro cancellations
string | character vector

List of macro cancellations to use while parsing header files, specified as a string or a character vector.

Attributes:

GetAccess	public
SetAccess	private

Methods

Public Methods

addClass	Add class to library definition
addFunction	Add function to library definition
build	Build library interface
summary	Display C++ classes, functions and enums in MATLAB interface
validate	Validate library definition

See Also

clibgen.ClassDefinition | clibgen.EnumDefinition | clibgen.FunctionDefinition | clibgen.generateLibraryDefinition

Introduced in R2019a

addClass

Class: `clibgen.LibraryDefinition`
Package: `clibgen`

Add class to library definition

Syntax

```
ClassDef = addClass(libDef,CPPName)
```

Description

`ClassDef = addClass(libDef,CPPName)` adds a `ClassDefinition` object to the library definition, where `CPPName` is the C++ class name.

Input Arguments

`libDef` — Library definition
`clibgen.LibraryDefinition`

Library definition, specified as a `clibgen.LibraryDefinition` object.

`CPPName` — C++ class name
string scalar | character vector

Fully qualified C++ class name, specified as a string scalar or a character vector.

See Also

Introduced in R2019a

addFunction

Class: `clibgen.LibraryDefinition`
Package: `clibgen`

Add function to library definition

Syntax

```
FunctionDef = addFunction(libDef,CPPSignature)
```

Description

`FunctionDef = addFunction(libDef,CPPSignature)` adds a `FunctionDefinition` object to the library definition, where `CPPSignature` is the C++ function signature.

Input Arguments

`libDef` — Library definition
`clibgen.LibraryDefinition`

Library definition, specified as a `clibgen.LibraryDefinition` object.

`CPPSignature` — C++ function signature
string scalar | character vector

Fully qualified C++ function signature, specified as a string scalar or a character vector.

See Also

Introduced in R2019a

build

Class: `clibgen.LibraryDefinition`
Package: `clibgen`

Build library interface

Syntax

```
build(libDef)
```

Description

`build(libDef)` validates and builds a library interface. If successful, the method prints an output to the Command Window. The output contains information about the interface file that is generated and any functions dropped because of validation failure.

Input Arguments

`libDef` — Library definition
`clibgen.LibraryDefinition`

Library definition, specified as a `clibgen.LibraryDefinition` object.

See Also

Introduced in R2019a

summary

Class: `clibgen.LibraryDefinition`
Package: `clibgen`

Display C++ classes, functions and enums in MATLAB interface

Syntax

```
summary(libDef)
summary(libDef,'mapping')
```

Description

`summary(libDef)` displays a summary of the C++ classes, functions, and enums defined in the MATLAB library definition.

`summary(libDef,'mapping')` displays the mapping between C++ functionality and MATLAB functionality.

Input Arguments

`libDef` — Library definition
`clibgen.LibraryDefinition`

Library definition, specified as a `clibgen.LibraryDefinition` object.

See Also

Introduced in R2019a

validate

Class: `clibgen.LibraryDefinition`
Package: `clibgen`

Validate library definition

Syntax

```
validate(libDef)
```

Description

`validate(libDef)` validates a library definition.

Input Arguments

`libDef` — Library definition
`clibgen.LibraryDefinition`

Library definition, specified as a `clibgen.LibraryDefinition` object.

See Also

Introduced in R2019a

clibgen.MethodDefinition class

Package: `clibgen`

Definition for MATLAB function corresponding to C++ class method

Description

The `clibgen.MethodDefinition` class contains the MATLAB definition for a member function defined in the header of a C++ class.

The `clibgen.MethodDefinition` class is a `handle` class.

Creation

MATLAB constructs a `MethodDefinition` object when you call `addMethod` on a `clibgen.ClassDefinition` object.

Properties

`Description` — Help text
string | character vector

Help text for the end user describing the method, specified as a string or a character vector. The default text is

`help clib.`*`packageName`*`.`*`fullMLClassName`*

`Method of class clib.`*`packageName`*`.`*`fullMLClassName`*

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`CPPSignature` — C++ method signature
string | character vector

C++ method signature, specified as a string or a character vector.

Attributes:

```
GetAccess                                   public
SetAccess                                   private
```

`DefiningClass` — Class containing method

`clibgen.ClassDefinition`

Class containing the method, specified as a `clibgen.ClassDefinition` object.

Attributes:

```
GetAccess                                   public
SetAccess                                   private
```

`MATLABSignature` — Method signature in MATLAB

string | character vector

Method signature in MATLAB, specified as a string or a character vector. MATLAB displays this signature when you load the library.

Attributes:

```
GetAccess                                   public
SetAccess                                   private
Dependent                                   true
```

`Valid` — Definition complete indicator

`false` (default) | `true`

Indicate if method definition is complete, specified as `true` or `false`.

Attributes:

```
GetAccess                                   public
SetAccess                                   private
```

Methods

Public Methods

defineArgument	Add input argument definition for C++ method
defineOutput	Add output argument definition for C++ method
validate	Validate method definition

See Also

`addMethod` | `clibgen.ClassDefinition`

Introduced in R2019a

defineArgument

Class: `clibgen.MethodDefinition`
Package: `clibgen`

Add input argument definition for C++ method

Syntax

```
defineArgument(methodDef,argName,mltype)
defineArgument(methodDef,argName,mltype,direction)
defineArgument(methodDef,argName,mltype,direction,shape)
```

Description

`defineArgument(methodDef,argName,mltype)` adds an input argument definition.

`defineArgument(methodDef,argName,mltype,direction)` defines whether argument is input, output, or both.

`defineArgument(methodDef,argName,mltype,direction,shape)` provides information about data dimensions.

Input Arguments

`methodDef` — Method definition
`clibgen.MethodDefinition`

Method definition, specified as a `clibgen.MethodDefinition` object.

`argName` — C++ argument name
string scalar | character vector

C++ argument name, specified as a string scalar or a character vector.

`mltype` — MATLAB type
string scalar | character vector

MATLAB type, specified as a string scalar or a character vector.

`direction` — Argument type
`"input"` (default) | `"output"` | `"inputoutput"`

Argument type, specified as `"input"` for an input argument, `"output"` for an output argument, or `"inputoutput"` for both an input and an output argument.

`shape` — Dimension definition
string vector | scalar text | positive integer vector | `"nullTerminated"` | cell array

Dimension definition used by MATLAB to define data dimensions, specified as a string vector, scalar text, positive integer vector, `"nullTerminated"`, or a cell array. For a cell array, the elements are a combination of scalar text and scalar positive integers.

If you can define the argument dimension as a fixed scalar value, then enter a number, such as `5`.

If the dimension is defined by another argument, then enter the argument name as a string. For example, consider the following C++ signature. If argument `len` defines the length of `data`, then the value of `shape` is `"len"`.

```
myFunc(const int *data, int len)
```

If the size of an argument is defined by an array, then the value of `shape` is one of the following:

- `1`
- Fixed dimensions: Enter a numerical array, such as `[5,2]`.
- Variable dimensions: Enter a string array of parameter names, such as `["row","col"]`

If the C++ type for the argument is a string, then use these options to choose values for the `mltype` and `shape` arguments.

C++ Type	`mltype`	Options for `shape`
`char *`	`int8`	Scalar value Array of scalar values

C++ Type	mltype	Options for shape
`const char *`	`char`	MATLAB `char` Character vector
	`string`	`"nullTerminated"`

For more information, see “MATLAB to C++ Data Type Mapping”.

See Also

`defineOutput`

Introduced in R2019a

defineOutput

Class: `clibgen.MethodDefinition`
Package: `clibgen`

Add output argument definition for C++ method

Syntax

```
defineOutput(methodDef,argName,mltype)
defineOutput(methodDef,argName,mltype,shape)
```

Description

`defineOutput(methodDef,argName,mltype)` adds an output argument definition.

`defineOutput(methodDef,argName,mltype,shape)` provides information about data dimensions.

Input Arguments

`methodDef` — Method definition
`clibgen.MethodDefinition`

Method definition, specified as a `clibgen.MethodDefinition` object.

`argName` — C++ output argument name
`"RetVal"` (default) | string scalar | character vector

C++ output argument name, specified as a string scalar or a character vector.

`mltype` — MATLAB type
string scalar | character vector

MATLAB type, specified as a string scalar or a character vector.

shape — Dimension definition
string vector | scalar text | positive integer vector | `"nullTerminated"` | cell array

Dimension definition used by MATLAB to define data dimensions, specified as a string vector, scalar text, positive integer vector, `"nullTerminated"`, or a cell array. For a cell array, the elements are a combination of scalar text and scalar positive integers.

If you can define the argument dimension as a fixed scalar value, then enter a number, such as `5`.

If the dimension is defined by another argument, then enter the argument name as a string. For example, consider the following C++ signature. If argument `len` defines the length of `data`, then the value of `shape` is `"len"`.

```
myFunc(const int *data, int len)
```

If the size of an argument is defined by an array, then the value of `shape` is one of the following:

- `1`
- Fixed dimensions: Enter a numerical array, such as `[5,2]`.
- Variable dimensions: Enter a string array of parameter names, such as `["row","col"]`

If the C++ type for the argument is a string, then use these options to choose values for the `mltype` and `shape` arguments.

C++ Type	mltype	Options for shape
`char *`	`int8`	Scalar value Array of scalar values
`const char *`	`char`	MATLAB `char` Character vector
	`string`	`"nullTerminated"`

For more information, see “MATLAB to C++ Data Type Mapping”.

See Also

`defineArgument`

Introduced in R2019a

validate

Class: `clibgen.MethodDefinition`
Package: `clibgen`

Validate method definition

Syntax

```
validate(methodDef)
```

Description

`validate(methodDef)` validates the number of arguments and the outputs of the method to ensure that they match the C++ signature.

Input Arguments

`methodDef` — Method definition
`clibgen.MethodDefinition`

Method definition, specified as a `clibgen.MethodDefinition` object.

See Also

Introduced in R2019a

clibgen.PropertyDefinition class

Package: `clibgen`

Definition for MATLAB property corresponding to public property of C++ class

Description

The `clibgen.PropertyDefinition` class contains the MATLAB definition for a public property defined in the header of a C++ class.

The `clibgen.PropertyDefinition` class is a `handle` class.

Creation

MATLAB constructs a `PropertyDefinition` object when you call `addProperty` on a `clibgen.ClassDefinition` object.

Properties

`Description` — Help text
string | character vector

Help text for the end user describing the property, specified as a string or a character vector. The default text is

`help clib.`*`packageName.fullMLClassName.ClassPropertyName`*

`C++ data member`

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`CPPName` — C++ property name
string | character vector

C++ property name, specified as a string or a character vector.

Attributes:

GetAccess	public
SetAccess	private

DefiningClass — Class containing property
clibgen.ClassDefinition

Class containing the property, specified as a `clibgen.ClassDefinition` object.

Attributes:

GetAccess	public
SetAccess	private

MATLABType — Type in MATLAB
string | character vector

Type in MATLAB, specified as a string or a character vector.

Attributes:

GetAccess	public
SetAccess	private

See Also

`addProperty` | `clibgen.ClassDefinition`

Introduced in R2019a

clibgen.buildInterface

Package: `clibgen`

Create interface to C++ library without definition file

Syntax

```
clibgen.buildInterface(HeaderFiles)
clibgen.buildInterface(HeaderFiles,Name,Value)
```

Description

`clibgen.buildInterface(HeaderFiles)` generates a MATLAB interface to the C++ library defined by `HeaderFiles`. Class constructors, methods, and functions that MATLAB cannot automatically define are not included in the interface. To include missing functionality, use the `clibgen.generateLibraryDefinition` function.

`clibgen.buildInterface(HeaderFiles,Name,Value)` generates the interface using one or more name-value pair arguments.

Examples

Create Interface for Library `school`

Copy the `school.hpp` header file to writable folder `H:\Documents\MATLAB\publisher`.

```
copyfile(fullfile(matlabroot,'extern','examples','cpp_interface','school.hpp'),'.','f')
```

Build the interface.

```
clibgen.buildInterface('school.hpp')
```

```
Warning: Only header files specified; assuming this is a header-only library. If library
files also need to be specified, use the 'Libraries' option.
```

```
Building interface file 'schoolInterface.dll'.
Interface file 'schoolInterface.dll' built in folder 'H:\Documents\MATLAB\publisher\school'.
To use the library, add the interface file folder to the MATLAB path.
1 constructs need further definition to be included in the interface.
Use clibgen.generateLibraryDefinition to provide definitions.
```

Click the `interface file folder` link to add the interface to the path.

Display the classes in the library.

```
clib.school.Tab
```

Display help for the library.

```
help clib.school
```

```
Classes contained in clib.school:
Person                          - clib.school.Person    Representation of C++ class Person
Teacher                         - clib.school.Teacher    Representation of C++ class Teacher
Student                         - clib.school.Student    Representation of C++ class Student
```

Compare this functionality with the output of the `summary(defineschool)` command in "Publish Interface to Header-Only C++ Library". This interface does not include:

```
Functions
  string clib.school.getName(clib.school.Person)
```

Input Arguments

`HeaderFiles` — Header files

string array | character vector | cell array of character vectors

One or more header files, specified as a string array, character vector, or cell array of character vectors. Supported file extensions are `.h`, `.hpp`, and `.hxx`. A header file without an extension is also supported. If needed, the name includes the full or relative path to the file.

If the main header file contains `#include` statements for header files in different folders, use the `IncludePath` argument to specify these paths.

MATLAB writes the interface files in a subfolder in the current folder, unless `OutputFolder` is specified. The name of the subfolder is the name of the first header file without a file extension. For example, the following statement creates the interface library file in subfolder `myHeader` in the current working folder.

```
clibgen.buildInterface('myHeader.hpp')
```

Data Types: `char` | `string` | `cell`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
```
clibgen.buildInterface('myHeader.hpp','Libraries','myHeader.lib','OutputFolder','C:\work');
```

`Libraries` — Library path

empty (default) | string array | character vector | cell array of character vectors

One or more library paths, specified as a string array, character vector, or cell array of character vectors. Each `Libraries` value is a full path name to a library. For example, running this statement on Windows creates `sampleInterface.dll` using `myLib.lib` in `C:\myLib\` and writes it to subfolder `sample` in the current working folder.

```
clibgen.buildInterface('sample.hpp','Libraries','C:\myLib\myLib.lib')
```

Data Types: `char` | `string` | `cell`

`IncludePath` — Folders for included header files

empty (default) | string array | character vector | cell array of character vectors

Folders for included header files, specified as a string array, character vector, or cell array of character vectors. `IncludePath` is the full path name to folders to include during compilation of the header files.

Data Types: `char` | `string` | `cell`

`OutputFolder` — Folder name

current folder (default) | string scalar | character vector

Folder name used to generate the interface library, specified as a string scalar or a character vector. For example, this statement creates `myHeaderInterface.dll` in `C:\work\myHeader` on Windows.

```
clibgen.buildInterface('myHeader.hpp','OutputFolder','C:\work')
```

Data Types: `char` | `string` | `cell`

`PackageName` — Interface package name
header file name (default) | string scalar | character vector

Interface package name, specified as a string scalar or a character vector. For interfaces created from a single header file, the default value is the name of header. For multiple header files, specify the package name as a valid MATLAB name. For example, this statement creates a `myPackageInterface` library file in subfolder `myPackage` in the current working folder.

```
clibgen.buildInterface({'h1.hpp','h2.hpp'},'PackageName','myPackage')
```

Data Types: `char` | `string` | `cell`

`Verbose` — Option to display generation messages
`false` (default) | `true`

Option to display generation messages, specified as `true` or `false`. When `true`, `clibgen.buildInterface` displays generation messages to the command window while building the interface. For example, this statement creates an `h1Interface` library file in subfolder `h1` and displays messages in the command window.

```
clibgen.buildInterface('h1.hpp','Verbose',true)
```

For more information, see “Messages About Unsupported Types”.

Data Types: `logical`

`TreatObjectPointerAsScalar` — Shape for object pointer type
`false` (default) | `true`

Shape for object pointer type, specified as `true` or `false`. If `TreatObjectPointerAsScalar` is true, then MATLAB treats all object pointer types in the library functions as scalar. Otherwise, the shape of the type is unknown.

Data Types: `logical`

`TreatConstCharPointerAsCString` — Shape and MATLAB type for `const char *` pointer type
`false` (default) | `true`

Shape and MATLAB type for `const char *` pointer type, specified as `true` or `false`. If `TreatConstCharPointerAsCString` is true, then MATLAB treats all `const char *` types in the library functions as C strings by specifying MATLAB Type as `string` and shape as `nullTerminated`. Otherwise, the shape of the type is unknown.

Data Types: `logical`

`DefinedMacros` — List of macro definitions
empty (default) | scalar string | row vector of scalar strings

List of macro definitions to use while parsing header files, specified as empty, a scalar string, or a row vector of scalar strings. The name contains characters 1-9, a-z, A-Z and '_' and cannot begin with a numeral.

Data Types: `string`

`UndefinedMacros` — List of macro cancellations
empty (default) | scalar string | row vector of scalar strings

List of macro cancellations to use while parsing header files, specified as empty, a scalar string, or a row vector of scalar strings. The name contains characters 1-9, a-z, A-Z and '_' and cannot begin with a numeral.

Data Types: `string`

See Also

`clibgen.generateLibraryDefinition`

Introduced in R2019a

clibgen.generateLibraryDefinition

Package: `clibgen`

Create definition file for C++ library

Syntax

```
clibgen.generateLibraryDefinition(HeaderFiles)
clibgen.generateLibraryDefinition(HeaderFiles,Name,Value)
```

Description

`clibgen.generateLibraryDefinition(HeaderFiles)` creates a MATLAB Live Code definition file used to generate a MATLAB interface to the C++ library defined by `HeaderFiles`.

The name of the definition file is `define`*`libName`*`.mlx`. For more information about using this file, see "Define MATLAB Interface to C++ Library".

Use the `build` method to create the interface.

`clibgen.generateLibraryDefinition(HeaderFiles,Name,Value)` creates the file using one or more name-value pair arguments.

Examples

Create Definition File for Library `school`

Copy the `school.hpp` header file to a writable folder.

```
copyfile(fullfile(matlabroot,"extern","examples","cpp_interface","school.hpp"),".","f")
```

Generate the library definition file `defineschool.mlx`.

```
clibgen.generateLibraryDefinition("school.hpp")
```

```
Warning: Only header files specified; assuming this is a header-only
library. If library files also need to be specified, use the 'Libraries' option.

Using MinGW64 Compiler (C++) compiler.
Generated definition file defineschool.mlx and data file 'schoolData.xml' contain definitions for
21 constructs supported by MATLAB.
1 constructs require additional definition. To include these constructs in the interface,
edit the definitions in defineschool.mlx.
Build using build(defineschool).
```

Input Arguments

`HeaderFiles` — Header file

string array | character vector | cell array of character vectors

One or more header files, specified as a string array, character vector, or cell array of character vectors. Supported file extensions are `.h`, `.hpp`, and `.hxx`. A header file without an extension is also supported. If needed, the name includes the full or relative path to the file.

If the main header file contains `#include` statements for header files in different folders, use the `IncludePath` argument to specify these paths.

MATLAB writes the interface files in a subfolder in the current folder, unless `OutputFolder` is specified. The name of the subfolder is the name of the first header file without a file extension. For example, the following statement creates the interface library file in subfolder `myHeader` in the current working folder.

```
clibgen.generateLibraryDefinition('myHeader.hpp')
```

Example: `"sample.hpp"`

Data Types: `char` | `string` | `cell`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
`clibgen.generateLibraryDefinition({"hfile1.hpp","hfile2.hpp"},"Libraries","mylib.lib","IncludePath","C:\mylib`

```
\include","PackageName","mypackage","OutputFolder","C:\work","Define
dMacros",["mymacro1","mymacro2=0"],"UndefinedMacros","mymacro3");
```

`Libraries` — Library path

empty (default) | string array | character vector | cell array of character vectors

Full path name to library, specified as a string array, character vector, or cell array of character vectors. For example,

```
clibgen.generateLibraryDefinition("sample.hpp","Libraries","C:\myLib\myLib.lib")
```

creates `definesample.mlx` using `myLib.lib` in `C:\myLib\` and writes it in the current working folder.

Data Types: `char` | `string` | `cell`

`IncludePath` — Folders for included header files

empty (default) | string array | character vector | cell array of character vectors

Folders for included header files, specified as a string array, character vector, or cell array of character vectors. `IncludePath` is the full path name to folders to include during compilation of the header files.

Data Types: `char` | `string` | `cell`

`OutputFolder` — Folder name

current folder (default) | string scalar | character vector

Folder name used to generate the definition file, specified as a string scalar or a character vector. For example,

```
clibgen.generateLibraryDefinition("myHeader.hpp","OutputFolder","C:\work")
```

creates `definemyHeader.mlx` in `C:\work`.

Data Types: `char` | `string` | `cell`

`PackageName` — Interface package name

header file name (default) | string scalar | character vector

Name of generated interface package, specified as a string scalar or a character vector. For interfaces created from a single header file, the default value is the name of header. For multiple header files, specify the package name as a valid MATLAB name. For example,

```
clibgen.generateLibraryDefinition({"h1.hpp","h2.hpp"},"PackageName","myPackage")
```

creates `definemyPackage.mlx` in subfolder `myPackage` in the current working folder.

Data Types: `char` | `string` | `cell`

`Verbose` — Option to display generation messages
`false` (default) | `true`

Option to display generation messages, specified as `true` or `false`. If `Verbose` is true, then MATLAB displays generation messages to the command window while creating the definition file. For example,

```
clibgen.generateLibraryDefinition("h1.hpp","Verbose",true)
```

creates `defineh1.mlx` and displays messages to the command window.

For more information, see “Messages About Unsupported Types”.

Data Types: `logical`

`TreatObjectPointerAsScalar` — Shape for object pointer type
`false` (default) | `true`

Shape for object pointer type, specified as `true` or `false`. If `TreatObjectPointerAsScalar` is true, then MATLAB treats all object pointer types in the library functions as scalar. Otherwise, the shape of the type is unknown.

Data Types: `logical`

`TreatConstCharPointerAsCString` — Shape and MATLAB type for `const char *` pointer type
`false` (default) | `true`

Shape and MATLAB type for `const char *` pointer type, specified as `true` or `false`. If `TreatConstCharPointerAsCString` is true, then MATLAB treats all `const char *` types in the library functions as C strings by specifying MATLAB Type as `string` and shape as `nullTerminated`. Otherwise, the shape of the type is unknown.

Data Types: `logical`

`DefinedMacros` — List of macro definitions
empty (default) | scalar string | row vector of scalar strings

List of macro definitions to use while parsing header files, specified as empty, a scalar string, or a row vector of scalar strings. The name contains characters 1-9, a-z, A-Z and '_' and cannot begin with a numeral.

Data Types: `string`

`UndefinedMacros` — List of macro cancellations
empty (default) | scalar string | row vector of scalar strings

List of macro cancellations to use while parsing header files, specified as empty, a scalar string, or a row vector of scalar strings. The name contains characters 1-9, a-z, A-Z and '_' and cannot begin with a numeral.

Data Types: `string`

Limitations

- Avoid non-ASCII characters in folder and file names, as some locale settings might not support those characters.

Tips

- To recreate a library definition file, first delete the `define`*`libName`*`.mlx` file, then call `clibgen.generateLibraryDefinition`.

See Also

`build` | `clibgen.buildInterface`

Topics

"Define MATLAB Interface to C++ Library"
"Errors Parsing Header Files on macOS"

Introduced in R2019a

matlab.graphics.chartcontainer.ChartContainer class

Package: `matlab.graphics.chartcontainer`

Base class for developing chart objects

Description

`matlab.graphics.chartcontainer.ChartContainer` is a class for defining your own class of chart objects. When you define a subclass of this base class, you can use the `ChartContainer` properties and methods to build your chart objects. Specify this syntax on the first line of your class definition file to directly inherit from the `ChartContainer` class, where `MyChart` is the name of your object:

```
classdef MyChart < matlab.graphics.chartcontainer.ChartContainer
```

The `matlab.graphics.chartcontainer.ChartContainer` class is a `handle` class.

Class Attributes

`Abstract`	`true`
`HandleCompatible`	`true`

For information on class attributes, see “Class Attributes”.

Properties

`InnerPosition` — Chart size and location, excluding margins
`[left bottom width height]`

Chart size and location, excluding the margins for decorations such as axis labels and tick marks. Specify this property as a vector of form `[left bottom width height]`. The values are in the units specified by the `Units` property.

Note Setting this property has no effect when the parent of the chart is a `TiledChartLayout`.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true
NonCopyable	true

`OuterPosition` — Chart size and location, including margins

`[left bottom width height]`

Chart size and location, including the margins for decorations such as axis labels and tick marks. Specify this property as a vector of form `[left bottom width height]`. The values are in the units specified by the `Units` property.

Note Setting this property has no effect when the parent of the chart is a `TiledChartLayout`.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true
NonCopyable	true

`Position` — Chart size and location, excluding margins

`[left bottom width height]`

Chart size and location, excluding the margins for decorations such as axis labels and tick marks. Specify this property as a vector of form `[left bottom width height]`. This property is equivalent to the `InnerPosition` property.

Note Setting this property has no effect when the parent of the chart is a `TiledChartLayout`.

Attributes:

```
GetAccess                                       public
SetAccess                                       public
Dependent                                       true
NonCopyable                                     true
```

`ActivePositionProperty` — Position property to hold constant
`'outerposition'` | `'innerposition'`

Position property to hold constant when adding, removing, or changing decorations, specified as one of the following values:

Note Setting this property has no effect when the parent of the chart is a `TiledChartLayout`.

Attributes:

```
GetAccess                                       public
SetAccess                                       public
Dependent                                       true
NonCopyable                                     true
```

`Units` — Position units
`'normalized'` (default) | `'inches'` | `'centimeters'` | `'characters'` | `'points'` | `'pixels'`

Position units, specified as a value from the following table. To change the position of the chart in specific units, set the `Units` property before specifying the `Position` property. If you specify the `Units` and `Position` properties in a single command (using name-value pairs), be sure to specify `Units` before `Position`.

Units	Description
`'normalized'` (default)	Normalized with respect to the parent container, which is typically the figure, panel, or tab. The lower left corner of the container maps to `(0,0)`, and the upper right corner maps to `(1,1)`.

Units	Description
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the default font of the graphics root object:
`'points'`	Typography points. One point equals 1/72 inch.
`'pixels'`	Distances in pixels are independent of your system resolution on Windows and Macintosh systems: On Linux systems, the size of a pixel is determined by your system resolution.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`
`Dependent`	`true`
`NonCopyable`	`true`

`Layout` — Layout options

empty `LayoutOptions` array (default) | `TiledChartLayoutOptions`

Layout options, specified as a `TiledChartLayoutOptions` object. This property specifies options when an instance of your chart is a child of a tiled chart layout. If the instance is not a child of a tiled chart layout (for example, it is a child of a figure or panel), then this property is empty and has no effect. Otherwise, you can position the chart within the layout by setting the `Tile` and `TileSpan` properties on the `TiledChartLayoutOptions` object.

For example, this code places chart object `c` into the third tile of a tiled chart layout.

```
c.Layout.Tile = 3;
```

To make the chart span multiple tiles, specify the `TileSpan` property as a two-element vector. For example, this chart spans 2 rows and 3 columns of tiles.

```
c.Layout.TileSpan = [2 3];
```

Note Tiled chart layouts are not supported for the axes returned by the `getAxes` method. Instead, you can place an instance of your chart into a tiled chart layout.

Attributes:

GetAccess	public
SetAccess	public

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

Attributes:

GetAccess	public
SetAccess	public
Dependent	true
GetObservable	true
SetObservable	true

`Parent` — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `TiledChartLayout` object

Parent container of the chart, specified as a `Figure`, `Panel`, `Tab`, or `TiledChartLayout` object.

Attributes:

GetAccess	public
SetAccess	public
Transient	true
Dependent	true
GetObservable	true
SetObservable	true
NonCopyable	true

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'`

Visibility of the chart object handle in its parent's list of children, specified as one of these values:

If the object is not listed in the `Children` property of the chart's parent object, then functions that get object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles, regardless of their `HandleVisibility` property setting.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`
`Dependent`	`true`
`GetObservable`	`true`
`SetObservable`	`true`

Methods

Protected Methods

<subgroup> Concrete Methods </subgroup>
<subgroup> Abstract Methods </subgroup>

Examples

Smooth Line Plot

Define a class called `SmoothPlot` that plots a set of data using a dotted blue line with a smoothed version of the line.

To define the class, create a file called `SmoothPlot.m` that contains the following class definition with these features:

- Four public properties:
  - `XData` and `YData` store the *x*- and *y*-coordinate data for the original line.
  - `SmoothColor` and `SmoothWidth` control the color and width of the smooth line.

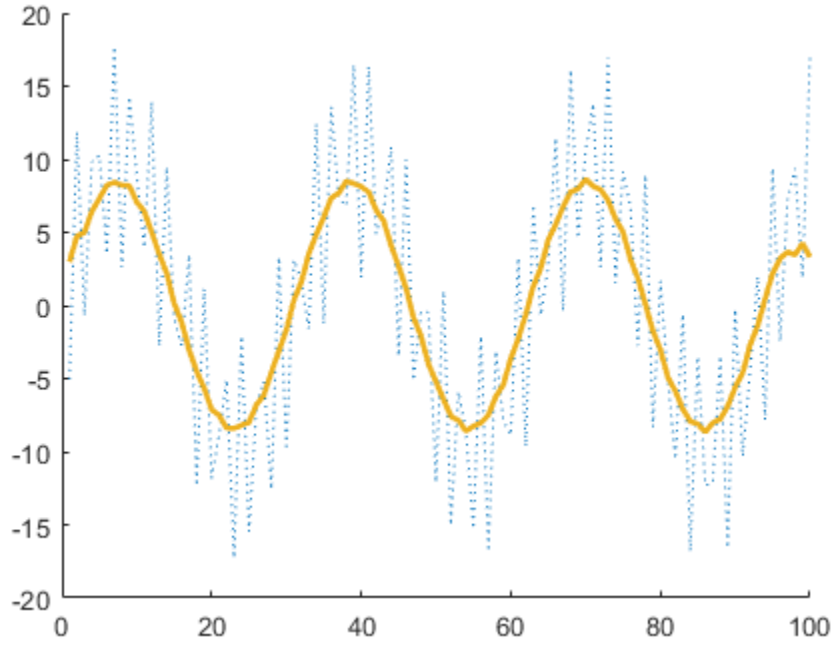
- Two private properties called `OriginalLine` and `SmoothLine` that store the `Line` objects for original and smoothed data.
- A `setup` method that initializes `OriginalLine` and `SmoothLine`.
- An `update` method that updates the plot when the user changes the value of a property.

```
classdef SmoothPlot < matlab.graphics.chartcontainer.ChartContainer
    properties
        XData (1,:) double = NaN
        YData (1,:) double = NaN
        SmoothColor (1,3) double {mustBeGreaterThanOrEqual(SmoothColor,0),...
            mustBeLessThanOrEqual(SmoothColor,1)} = [0.9290 0.6940 0.1250]
        SmoothWidth (1,1) double = 2
    end
    properties(Access = private,Transient,NonCopyable)
        OriginalLine (1,1) matlab.graphics.chart.primitive.Line
        SmoothLine (1,1) matlab.graphics.chart.primitive.Line
    end

    methods(Access = protected)
        function setup(obj)
            % Get the axes
            ax = getAxes(obj);

            % Create the orginal and smooth lines
            obj.OriginalLine = plot(ax,obj.XData,obj.YData,'LineStyle',':');
            hold(ax,'on')
            obj.SmoothLine = plot(ax,NaN,NaN,'LineStyle','-',...
                'Color',obj.SmoothColor,'LineWidth',obj.SmoothWidth);
            hold(ax,'off')
        end
        function update(obj)
            % Update line data
            obj.OriginalLine.XData = obj.XData;
            obj.OriginalLine.YData = obj.YData;
            obj.SmoothLine.XData = obj.XData;
            obj.SmoothLine.YData = createSmoothData(obj);

            % Update line color and width
            obj.SmoothLine.Color = obj.SmoothColor;
            obj.SmoothLine.LineWidth = obj.SmoothWidth;
        end
        function sm = createSmoothData(obj)
```

```
            % Calculate smoothed data
            v = ones(1,10)*0.1;
            sm = conv(obj.YData,v,'same');
        end
    end
end
```

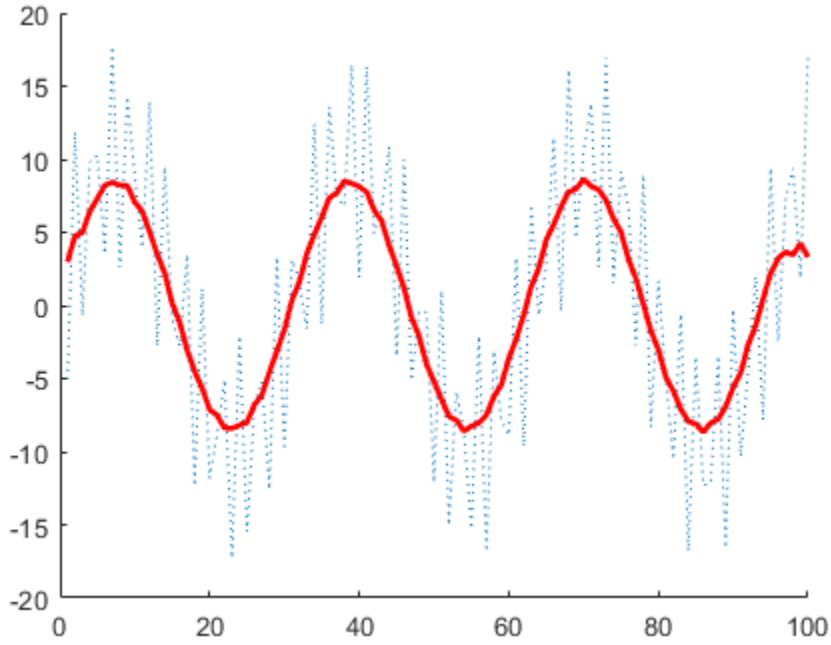
Next, create vectors `x` and `y`. Plot `x` and `y` by calling the `SmoothPlot` constructor method, which is provided by the `ChartContainer` class. Specify the `'XData'` and `'YData'` name-value pair arguments and return the object as `c`.

```
x = 1:1:100;
y = 10*sin(x./5) + 8*sin(10.*x + 0.5);
c = SmoothPlot('XData',x,'YData',y);
```



Use `c` to change the color of the smooth line to red.

```
c.SmoothColor = [1 0 0];
```



See Also

`matlab.graphics.chartcontainer.mixin.Colorbar` | `matlab.graphics.chartcontainer.mixin.Legend`

Topics

"Chart Development Overview"

Introduced in R2019b

matlab.graphics.chartcontainer.mixin.Legend class

Package: `matlab.graphics.chartcontainer.mixin`

Add legend support to chart container subclass

Description

`matlab.graphics.chartcontainer.mixin.Legend` is a class for adding legend support to charts that inherit from `matlab.graphics.chartcontainer.ChartContainer`. By inheriting from this class, you can display a legend in your chart. Your users can select options in the figure menu bar or the figure toolbar to show or hide that legend.

Use this syntax to enable legend support for your chart:

```
classdef MyChart < matlab.graphics.chartcontainer.ChartContainer & ...
                   matlab.graphics.chartcontainer.mixin.Legend
  ...
end
```

The `matlab.graphics.chartcontainer.mixin.Legend` class is a `handle` class.

Class Attributes

`Abstract`	`true`
`HandleCompatible`	`true`

For information on class attributes, see “Class Attributes”.

Properties

`LegendVisible` — Legend visibility
`'off'` (default) | `'on'`

Legend visibility, specified as `'off'` or `'on'`. When this property is set to `'on'`, the legend is visible in the chart.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true
NonCopyable	true

Methods

Protected Methods

getLegend Get legend object for legend mixin subclass

Examples

Smooth Line Plot with Legend

Define a `SmoothLegendPlot` class that plots a set of data using a dotted blue line with a smoothed version of the line.

To define the class, create a file named `SmoothLegendPlot.m` that contains the following code. The `setup` method for this class performs these tasks:

- Creates two `Line` objects with the appropriate `DisplayName` values to show in the legend
- Makes the legend visible by setting the `LegendVisible` property to `'on'`
- Gets the `Legend` object and customizes the text color, box edge color, and box line width

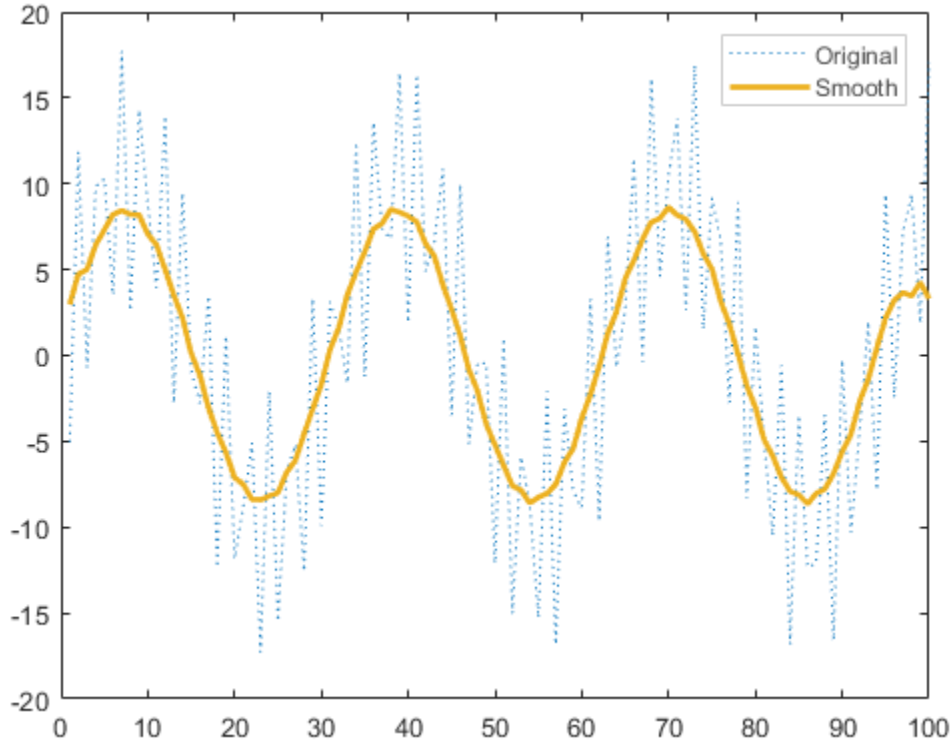
```
classdef SmoothLegendPlot < matlab.graphics.chartcontainer.ChartContainer & ...
        matlab.graphics.chartcontainer.mixin.Legend

    properties
        XData (1,:) double = NaN
        YData (1,:) double = NaN
```

```
        SmoothColor (1,3) double {mustBeGreaterThanOrEqual(SmoothColor,0),...
            mustBeLessThanOrEqual(SmoothColor,1)} = [0.9290 0.6940 0.1250]
        SmoothWidth (1,1) double = 2
    end
    properties(Access = private,Transient,NonCopyable)
        OriginalLine (1,1) matlab.graphics.chart.primitive.Line
        SmoothLine (1,1) matlab.graphics.chart.primitive.Line
    end

    methods(Access = protected)
        function setup(obj)
            ax = getAxes(obj);

            % Create line objects. Define line styles and legend names.
            obj.OriginalLine = plot(ax,obj.XData,obj.YData,'LineStyle',':',...
                'DisplayName','Original');
            hold(ax,'on')
            obj.SmoothLine = plot(ax,NaN,NaN,'LineStyle','-',...
                'Color',obj.SmoothColor,...
                'LineWidth',obj.SmoothWidth,...
                'DisplayName','Smooth');

            % Make legend visible
            obj.LegendVisible = 'on';

            % Get legend and set text color, edge color, and line width
            lgd = getLegend(obj);
            lgd.TextColor = [.3 .3 .3];
            lgd.EdgeColor = [.8 .8 .8];
            lgd.LineWidth = .7;
            hold(ax,'off')
        end
        function update(obj)
            % Update line data
            obj.OriginalLine.XData = obj.XData;
            obj.OriginalLine.YData = obj.YData;
            obj.SmoothLine.XData = obj.XData;
            obj.SmoothLine.YData = createSmoothData(obj);

            % Adjust smooth line appearance
            obj.SmoothLine.LineWidth = obj.SmoothWidth;
            obj.SmoothLine.Color = obj.SmoothColor;
        end
        function sm = createSmoothData(obj)
```

```
            % Calculate smoothed data
            v = ones(1,10)*0.1;
            sm = conv(obj.YData,v,'same');
        end
    end
end
```

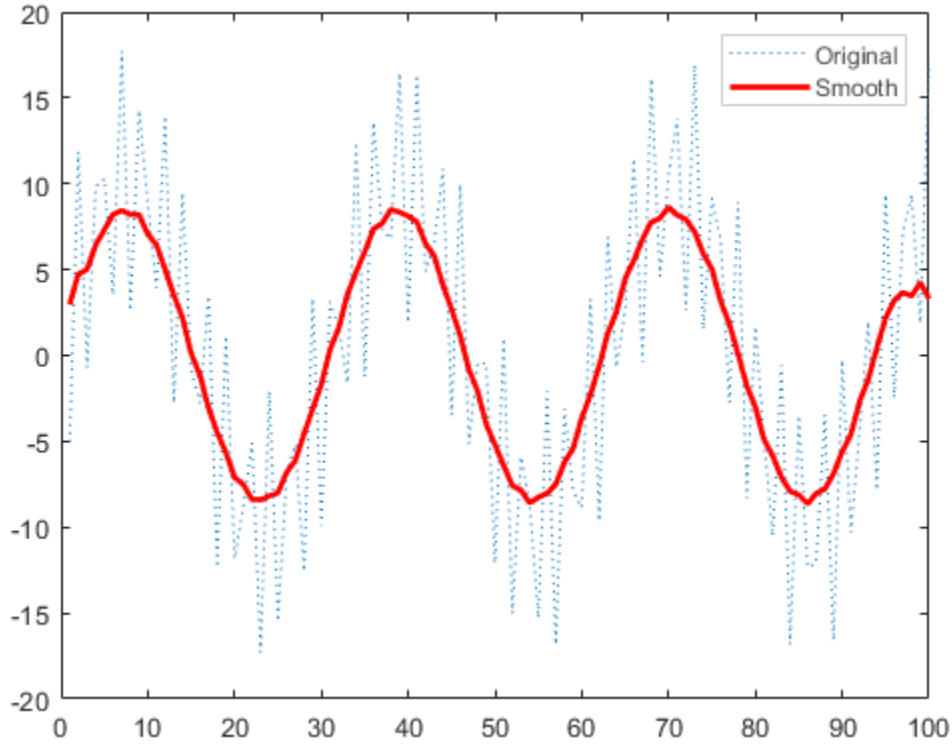
Next, create vectors `x` and `y`. Plot `x` and `y` by calling the `SmoothLegendPlot` constructor method, which is inherited from the `ChartContainer` class. Specify the `'XData'` and `'YData'` name-value pair arguments and return the chart object as `c`.

```
x = 1:1:100;
y = 10*sin(x./5)+8*sin(10.*x+0.5);
c = SmoothLegendPlot('XData',x,'YData',y);
```



Use `c` to change the color of the smooth line to red. Notice that the legend also shows the new line color.

```
c.SmoothColor = [1 0 0];
```



See Also

Classes
`matlab.graphics.chartcontainer.ChartContainer` | `matlab.graphics.chartcontainer.mixin.Colorbar`

Properties
Legend

Topics

"Chart Development Overview"

Introduced in R2019b

matlab.graphics.chartcontainer.mixin.Colorbar class

Package: `matlab.graphics.chartcontainer.mixin`

Add colorbar support to chart container subclass

Description

`matlab.graphics.chartcontainer.mixin.Colorbar` is a class for adding colorbar support charts that inherit from `matlab.graphics.chartcontainer.ChartContainer`. By inheriting from this class, you can display a colorbar in your chart. Your users can select options in the figure menu bar or the figure toolbar to show or hide that colorbar.

Use this syntax to enable colorbar support for your chart:

```
classdef MyChart < matlab.graphics.chartcontainer.ChartContainer & ...
                   matlab.graphics.chartcontainer.mixin.Colorbar
  ...
end
```

The `matlab.graphics.chartcontainer.mixin.Colorbar` class is a `handle` class.

Class Attributes

`Abstract`	`true`
`HandleCompatible`	`true`

For information on class attributes, see “Class Attributes”.

Properties

`ColorbarVisible` — Colorbar visibility
`'off'` (default) | `'on'`

Colorbar visibility, specified as `'off'` or `'on'`. When this property is set to `'on'`, the colorbar is visible in the chart.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true
NonCopyable	true

Methods

Protected Methods

getColorbar Get colorbar object for colorbar mixin subclass

Examples

Surface Plot with Image and Colorbar

Define a class named `SurfImagePlot` that displays a surface with an image underneath it.

To define the class, create a file named `SurfImagePlot.m` that contains the following code. The `setup` method for this class performs these tasks:

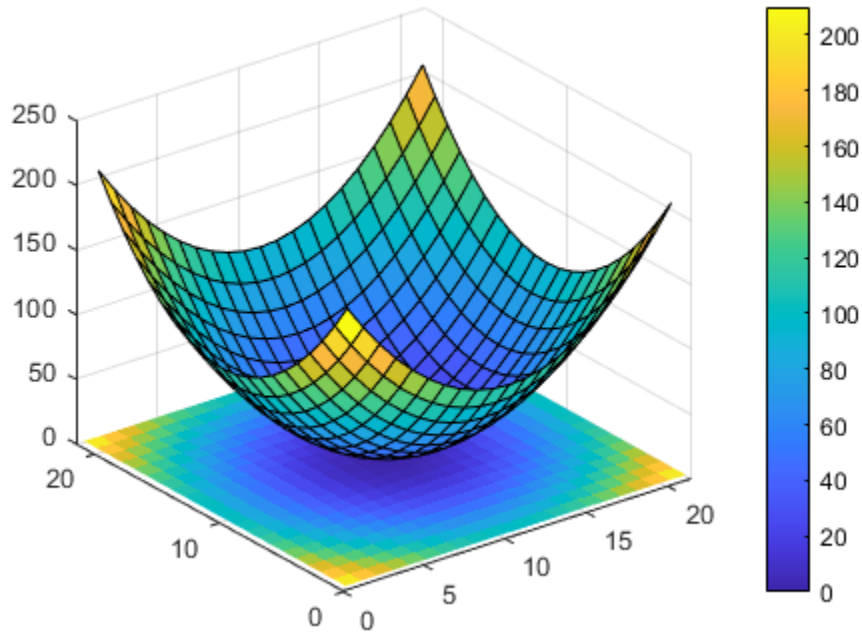
- Creates a `Surface` object with an offset on the `ZData` to provide enough space to display the image
- Creates an `Image` object
- Configures the axes
- Makes the colorbar visible by setting the `ColorbarVisible` property to `'on'`

```
classdef SurfImagePlot < matlab.graphics.chartcontainer.ChartContainer & ...
        matlab.graphics.chartcontainer.mixin.Colorbar

    properties
        ZData (:,:) double = []
        Offset (1,1) double = 10
```

```
        Colormap (:,3) double {mustBeGreaterThanOrEqual(Colormap,0),...
            mustBeLessThanOrEqual(Colormap,1)} = parula
    end
    properties(Access = private,Transient,NonCopyable)
        Surface (1,1) matlab.graphics.chart.primitive.Surface
        Image (1,1) matlab.graphics.primitive.Image
    end

    methods(Access = protected)
        function setup(obj)
            % Get the axes
            ax = getAxes(obj);

            % Create surface and image objects
            obj.Surface = surf(ax,[],[],[]);
            hold(ax,'on')
            obj.Image = imagesc(ax,[]);

            % Configure axes, make colorbar visible
            view(ax,3)
            axis(ax,'tight')
            grid(ax,'on')
            obj.ColorbarVisible = 'on';
            hold(ax,'off')
        end
        function update(obj)
            % Update Data and Colormap
            ax = getAxes(obj);
            [r,c] = size(obj.ZData);
            [X,Y] = meshgrid(1:c,1:r);
            obj.Surface.XData = X;
            obj.Surface.YData = Y;
            obj.Surface.ZData = obj.ZData + obj.Offset;
            obj.Image.CData = obj.ZData;
            colormap(ax,obj.Colormap)
        end
    end
end
```

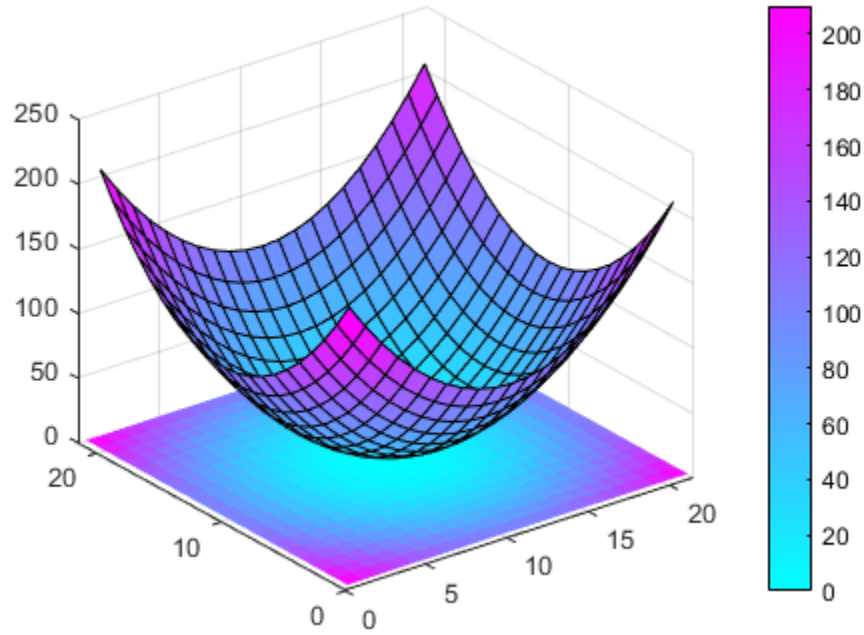
Next, define matrix `Z` as the z-coordinates of a surface. Plot `Z` by calling the `SurfImagePlot` constructor method, which is provided by the `ChartContainer` class. Specify the `'ZData'` name-value pair argument and return the object as `c`.

```
[X,Y] = meshgrid(-10:1:10);
Z = X.^2 + Y.^2;
c = SurfImagePlot('ZData',Z)
```



Use `c` to change the colormap to `cool`.

```
c.Colormap = cool;
```



See Also

Classes
`matlab.graphics.chartcontainer.ChartContainer` | `matlab.graphics.chartcontainer.mixin.Legend`

Properties
Colorbar

Topics
"Chart Development Overview"

Introduced in R2019b

getAxes

Class: `matlab.graphics.chartcontainer.ChartContainer`
Package: `matlab.graphics.chartcontainer`

Get axes for chart container subclass

Syntax

```
ax = getAxes(obj)
```

Description

`ax = getAxes(obj)` returns the current axes for a chart object that inherits from the `matlab.graphics.chartcontainer.ChartContainer` base class.

Input Arguments

`obj` — Object of the class
chart object

Object of the class that inherits from the `matlab.graphics.chartcontainer.ChartContainer` base class.

Output Arguments

ax — Axes object
`Axes` object

`Axes` object for the chart. Specify this object as the first input argument when you call a plotting function within your class definition. You can also use the object to set properties on the axes.

Attributes

Protected	true

To learn about attributes of methods, see Method Attributes.

Examples

Passing Axes to Plotting Functions

The `setup` method is a common place to call plotting functions and set the axes hold state. In both cases, you must specify the target axes.

Create a `setup` method in your class definition file. Within that method, call `getAxes` to get the axes object `ax`. Then plot two lines by passing `ax` as the first argument to the `plot` and `hold` functions. Call `hold(ax,'off')` at the end of the method.

```
classdef TwoLinesPlot < matlab.graphics.chartcontainer.ChartContainer

    properties
        % ...
    end

    methods(Access = protected)
        function setup(obj)
            % Get the axes
            ax = getAxes(obj);

            % Plot two lines in the axes
            line1 = plot(ax,[1 2 3 4 5],[3 5 1 4 9]);
            hold(ax,'on')
            line2 = plot(ax,[1 2 3 4 5],[30 52 21 9 18]);

            % Turn off hold state
            hold(ax,'off')
        end
        function update(obj)
            % ...
        end
```

```
    end
end
```

Change Axes Properties

Define a `setup` method in your class definition file. Within that method, call `getAxes` to get the axes object `ax`. Then set the *x*-axis color and the font angle for the axes. Call `hold(ax,'on')` before calling any plotting functions. Then call `hold(ax,'off')` at the end of the method.

```
classdef RedAxisPlot < matlab.graphics.chartcontainer.ChartContainer

    properties
        % ...
    end

    methods(Access = protected)
        function setup(obj)
            ax = getAxes(obj);
            ax.XColor = [1 0 0];
            ax.FontAngle = 'italic';
            hold(ax,'on')

            % Call plotting functions
            % ...

            hold(ax,'off')
        end
        function update(obj)
            % ...
        end
    end
end
```

Limitations

The `getAxes` method can return a Cartesian axes only. It cannot return other types of axes such as polar or geographic axes. Also, the axes provides limited support for setting the position and specifying the parent object.

- Setting the `OuterPositon`, `Position`, `ActivePositionProperty`, or `TightInset` properties on the axes might produce unexpected results. Instead, configure the position on an instance of your chart.
- Placing the axes into a tiled chart layout is not supported. However, you can place an instance of your chart into a tiled chart layout. For more information about tiled chart layouts, see `tiledlayout`.
- Changing the `Parent` property of the axes is not recommended. Instead, specify the `Parent` property on an instance of your chart.

See Also

Classes
`matlab.graphics.chartcontainer.ChartContainer`

Properties
Axes

Topics

"Chart Development Overview"

Introduced in R2019b

getLegend

Class: `matlab.graphics.chartcontainer.mixin.Legend`
Package: `matlab.graphics.chartcontainer.mixin`

Get legend object for legend mixin subclass

Syntax

```
lgd = getLegend(obj)
```

Description

`lgd = getLegend(obj)` returns the legend for a chart object that inherits from the `matlab.graphics.chartcontainer.mixin.Legend` class.

Input Arguments

`obj` — Object of the class
chart object

Object of the class that inherits from the `matlab.graphics.chartcontainer.mixin.Legend` class.

Output Arguments

`lgd` — Legend object
`Legend` object

`Legend` object for the chart.

Attributes

Protected	true

To learn about attributes of methods, see Method Attributes.

Examples

Modify Legend Properties

Create a `setup` method in your class definition file. Within that method, call `getLegend` to get the legend object `lgd`. Then change the text color, box edge color, and box line width.

```
classdef ConfigLegendChart < matlab.graphics.chartcontainer.ChartContainer & ...
        matlab.graphics.chartcontainer.mixin.Legend

    properties
        % ...
    end

    methods(Access = protected)
        function setup(obj)
            % Make legend visible
            obj.LegendVisible = 'on';

            % Get legend and set text color, edge color, and line width
            lgd = getLegend(obj);
            lgd.TextColor = [.3 .3 .3];
            lgd.EdgeColor = [.8 .8 .8];
            lgd.LineWidth = .7;
        end
        function update(obj)
            % ...
        end
```

```
    end
end
```

See Also

Classes
`matlab.graphics.chartcontainer.ChartContainer` | `matlab.graphics.chartcontainer.mixin.Legend`

Properties
Legend

Topics

"Chart Development Overview"

Introduced in R2019b

getColorbar

Class: `matlab.graphics.chartcontainer.mixin.Colorbar`
Package: `matlab.graphics.chartcontainer.mixin`

Get colorbar object for colorbar mixin subclass

Syntax

```
cb = getColorbar(obj)
```

Description

`cb = getColorbar(obj)` returns the colorbar for a chart object that inherits from the `matlab.graphics.chartcontainer.mixin.Colorbar` class.

Input Arguments

`obj` — Object of the class
chart object

Object of the class that inherits from the `matlab.graphics.chartcontainer.mixin.Colorbar` class.

Output Arguments

`cb` — Colorbar object
`Colorbar` object

`Colorbar` object for the chart.

Attributes

`Protected`	`true`

To learn about attributes of methods, see Method Attributes.

Examples

Modify Colorbar Properties

Create a `setup` method in your class definition file. Within that method, call `getColorbar` to get the colorbar object `c`. Then remove the ticks from the colorbar, and set its location to `'southoutside'`.

```
classdef ColorbarChart < matlab.graphics.chartcontainer.ChartContainer & ...
        matlab.graphics.chartcontainer.mixin.Colorbar

    properties
        % ...
    end

    methods(Access = protected)
        function setup(obj)
            % Make colorbar visible
            obj.ColorbarVisible = 'on';

            % Get the colorbar, remove the ticks, and set the location
            cb = getColorbar(obj);
            cb.Ticks = [];
            cb.Location = 'southoutside';
        end
        function update(obj)
            % ...
        end
```

```
    end
end
```

See Also

Classes
`matlab.graphics.chartcontainer.ChartContainer` | `matlab.graphics.chartcontainer.mixin.Colorbar`

Properties
Colorbar

Topics

"Chart Development Overview"

Introduced in R2019b

setup

Class: `matlab.graphics.chartcontainer.ChartContainer`
Package: `matlab.graphics.chartcontainer`

Set up instance of chart container subclass

Syntax

```
setup(obj)
```

Description

`setup(obj)` sets the initial state of the chart. It executes once when the chart object is created. Define this method to execute initialization code for each new instance of your class. For example, you can use this method to create the underlying graphics objects and set initial property values on those objects.

Input Arguments

`obj` — Object of the class
chart object

Object of the class that inherits from the `matlab.graphics.chartcontainer.ChartContainer` base class.

Attributes

`Abstract`	`true`
`Protected`	`true`

To learn about attributes of methods, see Method Attributes.

Examples

Smooth Line Plot

Define a `SmoothPlot` class that plots a set of data using a dotted blue line with a smoothed version of the line. To define the class, create a file named `SmoothPlot.m` that contains the following class definition with these features:

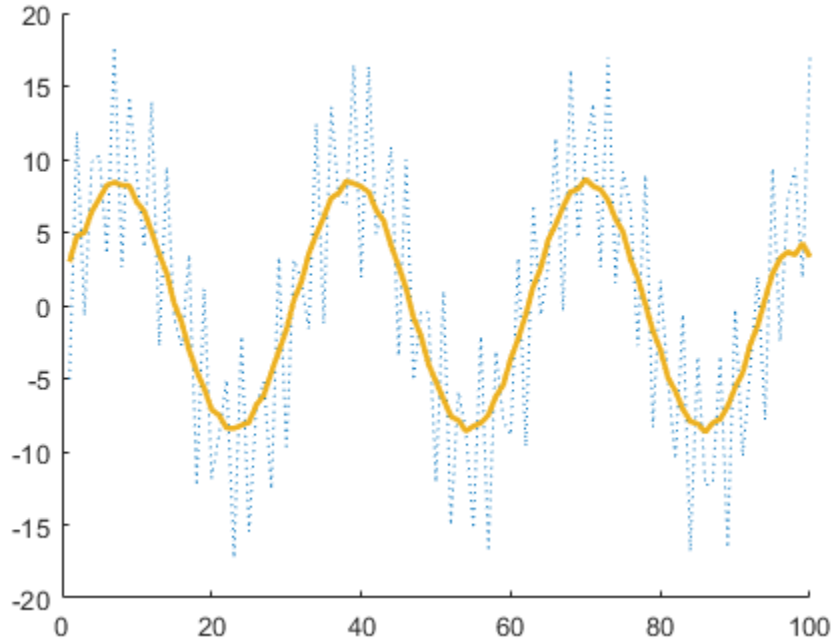
- Four public properties: `XData`, `YData`, `SmoothColor`, and `SmoothWidth`.
- Two private properties: `OriginalLine` and `SmoothLine`.
- A `setup` method that performs these tasks:
  - Gets the axes for the chart by calling the `getAxes` method.
  - Creates the `Line` object `OriginalLine` with the dotted line style. The color is not specified for this line, so MATLAB uses the default color, which is blue.
  - Creates the `Line` object `SmoothLine` with a solid line style, and sets the color and width of the line according to the class properties `SmoothColor` and `SmoothWidth`.
  - Sets the axes hold state to `'off'`.
- An `update` method that updates the plot when the user changes the value of a property.
- A `createSmoothData` method that calculates a smoothed version of the `YData` values.

```
classdef SmoothPlot < matlab.graphics.chartcontainer.ChartContainer
    properties
        XData (1,:) double = NaN
        YData (1,:) double = NaN
        SmoothColor (1,3) double {mustBeGreaterThanOrEqual(SmoothColor,0),...
            mustBeLessThanOrEqual(SmoothColor,1)} = [0.9290 0.6940 0.1250]
        SmoothWidth (1,1) double = 2
    end
    properties(Access = private,Transient,NonCopyable)
        OriginalLine (1,1) matlab.graphics.chart.primitive.Line
        SmoothLine (1,1) matlab.graphics.chart.primitive.Line
    end

    methods(Access = protected)
        function setup(obj)
```

```
            % Get the axes
            ax = getAxes(obj);

            % Create the original and smooth lines
            obj.OriginalLine = plot(ax,obj.XData,obj.YData,'LineStyle',':');
            hold(ax,'on')
            obj.SmoothLine = plot(ax,NaN,NaN,'LineStyle','-',...
                'Color',obj.SmoothColor,'LineWidth',obj.SmoothWidth);
            hold(ax,'off')
        end
        function update(obj)
            % Update line data
            obj.OriginalLine.XData = obj.XData;
            obj.OriginalLine.YData = obj.YData;
            obj.SmoothLine.XData = obj.XData;
            obj.SmoothLine.YData = createSmoothData(obj);

            % Update line color and width
            obj.SmoothLine.Color = obj.SmoothColor;
            obj.SmoothLine.LineWidth = obj.SmoothWidth;
        end
        function sm = createSmoothData(obj)
            % Calculate smoothed data
            v = ones(1,10)*0.1;
            sm = conv(obj.YData,v,'same');
        end
    end
end
```

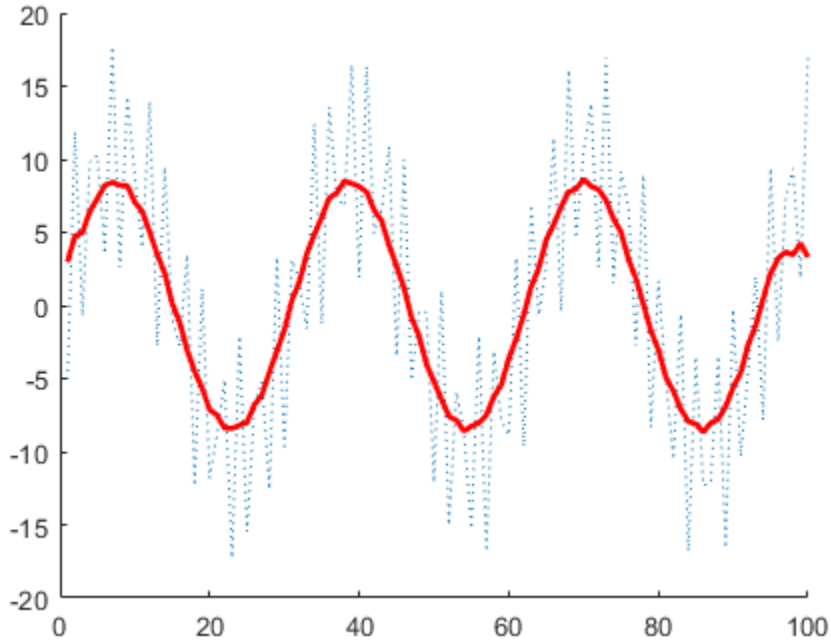
Next, create vectors `x` and `y`. Plot `x` and `y` by calling the `SmoothPlot` constructor method, which is provided by the `ChartContainer` class. Specify the `'XData'` and `'YData'` name-value pair arguments and return the object as `c`.

```
x = 1:1:100;
y = 10*sin(x./5) + 8*sin(10.*x+0.5);
c = SmoothPlot('XData',x,'YData',y);
```



Use `c` to change the color of the smooth line to red.

```
c.SmoothColor = [1 0 0];
```



See Also

`matlab.graphics.chartcontainer.ChartContainer`

Topics

"Chart Development Overview"

Introduced in R2019b

update

Class: `matlab.graphics.chartcontainer.ChartContainer`
Package: `matlab.graphics.chartcontainer`

Update instance of chart container subclass after setting properties

Syntax

```
update(obj)
```

Description

`update(obj)` updates the contents of the chart after one or more public property values change. Define this method to update the underlying graphics objects in the chart using the new property values. This method executes under the following conditions:

- During the next `drawnow` execution, after the user changes one or more property values on the chart.
- When an aspect of the user's graphics environment changes (such as the figure size).

Input Arguments

`obj` — Object of the class
chart object

Object of the class that inherits from the `matlab.graphics.chartcontainer.ChartContainer` base class.

Attributes

`Abstract`	`true`
`Protected`	`true`

To learn about attributes of methods, see Method Attributes.

Examples

Smooth Line Plot

Define a `SmoothPlot` class that plots a set of data using a dashed and dotted blue line with a smoothed version of the line.

To define the class, create a file named `SmoothPlot.m` that contains the following class definition with these features:

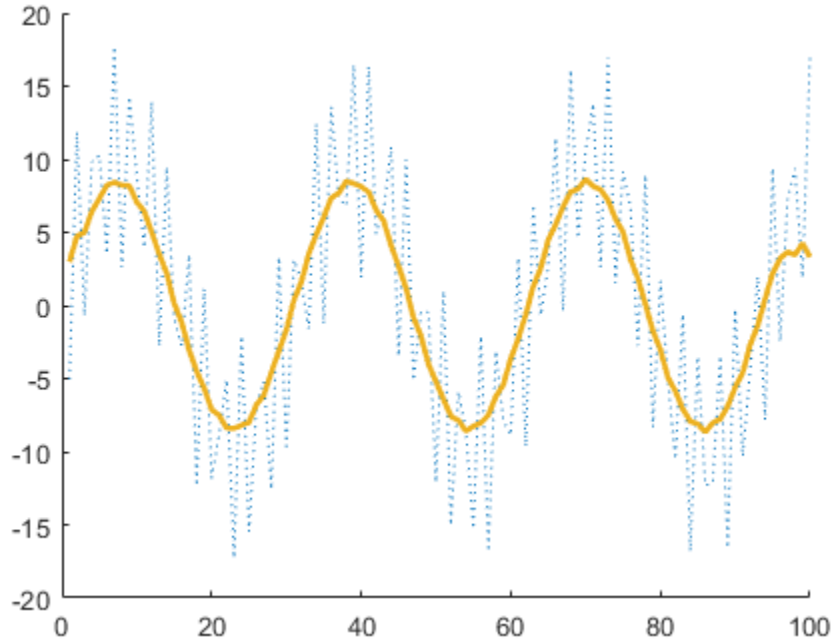
- Four public properties: `XData`, `YData`, `SmoothColor`, and `SmoothWidth`
- Two private properties: `OriginalLine` and `SmoothLine`
- A `setup` method that initializes `OriginalLine` and `SmoothLine`
- An `update` method that performs these tasks:
  - Updates the `XData` and `YData` properties of `OriginalLine` with the values stored in the `XData` and `YData` properties on the chart object
  - Updates the `XData` property of `SmoothLine` with the value stored in the `XData` property on the chart object
  - Calculates the value of the `YData` property of `SmoothLine` by calling the `createSmoothData` method.
  - Updates the `Color` and `LineWidth` properties of `SmoothLine` with the values stored in the `SmoothColor` and `SmoothWidth` properties on the chart object
- A `createSmoothData` method that calculates a smoothed version of `YData`

```
classdef SmoothPlot < matlab.graphics.chartcontainer.ChartContainer
    properties
        XData (1,:) double = NaN
        YData (1,:) double = NaN
        SmoothColor (1,3) double {mustBeGreaterThanOrEqual(SmoothColor,0),...
            mustBeLessThanOrEqual(SmoothColor,1)} = [0.9290 0.6940 0.1250]
        SmoothWidth (1,1) double = 2
    end
    properties(Access = private,Transient,NonCopyable)
        OriginalLine (1,1) matlab.graphics.chart.primitive.Line
        SmoothLine (1,1) matlab.graphics.chart.primitive.Line
```

```
    end

    methods(Access = protected)
        function setup(obj)
            % Get the axes
            ax = getAxes(obj);

            % Create the original and smooth lines
            obj.OriginalLine = plot(ax,obj.XData,obj.YData,'LineStyle',':');
            hold(ax,'on')
            obj.SmoothLine = plot(ax,NaN,NaN,'LineStyle','-',...
                'Color',obj.SmoothColor,'LineWidth',obj.SmoothWidth);
            hold(ax,'off')
        end
        function update(obj)
            % Update line data
            obj.OriginalLine.XData = obj.XData;
            obj.OriginalLine.YData = obj.YData;
            obj.SmoothLine.XData = obj.XData;
            obj.SmoothLine.YData = createSmoothData(obj);

            % Update line color and width
            obj.SmoothLine.Color = obj.SmoothColor;
            obj.SmoothLine.LineWidth = obj.SmoothWidth;
        end
        function sm = createSmoothData(obj)
            % Calculate smoothed data
            v = ones(1,10)*0.1;
            sm = conv(obj.YData,v,'same');
        end
    end
end
```

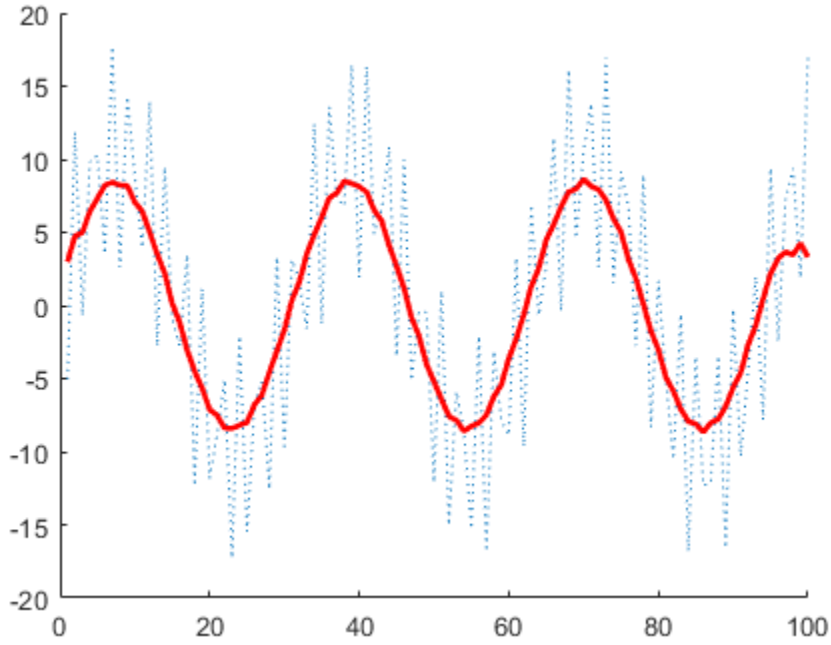
Next, create vectors `x` and `y`. Plot `x` and `y` by calling the `SmoothPlot` constructor method, which is provided by the `ChartContainer` class. Specify the `'XData'` and `'YData'` name-value pair arguments and return the object as `c`.

```
x = 1:1:100;
y = 10*sin(x./5)+8*sin(10.*x+0.5);
c = SmoothPlot('XData',x,'YData',y);
```



Use `c` to change the color of the smooth line to red.

```
c.SmoothColor = [1 0 0];
```



See Also

`matlab.graphics.chartcontainer.ChartContainer`

Topics

"Chart Development Overview"

Introduced in R2019b

closereq

Default figure close request function

Syntax

```
closereq
```

Description

`closereq` deletes the current figure. For more information, see the `CloseRequestFcn` figure property.

Introduced before R2006a

cmopts

(Has been removed) Name of source control system

Note `cmopts` has been removed. View the currently selected source control system through **Preferences** instead.

Syntax

```
cmopts
```

Description

`cmopts` returns the name of your version control system.

Output Arguments

Value Returned by cmopts	Description	Platform Supported On
`clearcase`	ClearCase® software from IBM® Rational®	UNIX platforms
`customverctrl`	Custom interface created using `customverctrl` function	UNIX platforms
`cvs`	Concurrent Version System (CVS)	UNIX platforms
`none`	No source control system selected	
`pvcs`	PVCS® and ChangeMan® software	UNIX platforms
`rcs`	Revision Control System (RCS)	UNIX platforms

Value Returned by cmopts	Description	Platform Supported On
Any SCC-compliant source control system, for example, `Microsoft Visual SourceSafe`	Varies	Windows platforms

Alternatives

To view the currently selected source control system, click the **Preferences** button on the **Home** tab, and select **General** > **Source Control**.

Introduced before R2006a

cmpermute

Rearrange colors in colormap

Syntax

```
[Y,newmap] = cmpermute(X,map)
[Y,newmap] = cmpermute(X,map,index)
```

Description

`[Y,newmap] = cmpermute(X,map)` randomly reorders the colors in colormap `map` to produce a new colormap, `newmap`. The `cmpermute` function also modifies the values in indexed image `X` to maintain correspondence between the indices and the colormap, and returns the result in `Y`. The image `Y` and associated colormap, `newmap`, produce the same image as `X` and `map`.

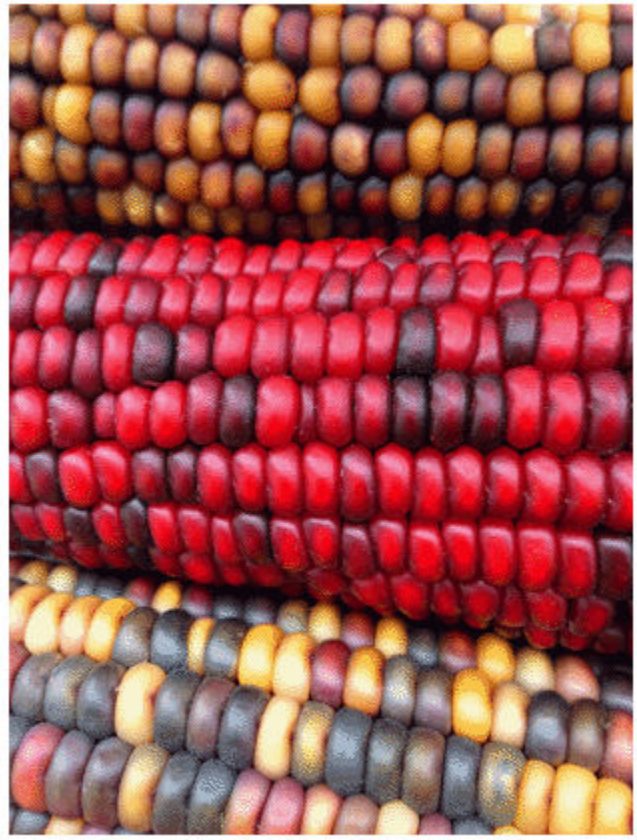
`[Y,newmap] = cmpermute(X,map,index)` uses an ordering matrix (such as the second output of `sort`) to define the order of colors in the new colormap.

Examples

Randomly Reorder Colormap and Display Image

Read the first image, `X`, and its associated colormap, `map`, from the sample indexed image file, `corn.tif`. Display the image.

```
[X,map] = imread('corn.tif');
imshow(X,map)
```



Randomly reorder the colormap to get a new colormap, `newmap`. Display image `X` with the new colormap.

```
[Y,newmap] = cmpermute(X,map);
colormap(newmap)
```

Input Arguments

X — Indexed image
m-by-*n* matrix of integers

Indexed image, specified as an *m*-by-*n* matrix of integers.

Data Types: `double` | `uint8`

`map` — Colormap
c-by-3 matrix

Colormap associated with indexed image `X`, specified as a *c*-by-3 matrix with values in the range [0, 1]. Each row of `map` is a three-element RGB triplet that specifies the red, green, and blue components of a single color of the colormap.

Data Types: `double`

`index` — Sort index
c-element vector of positive integers

Sort index, specified as a *c*-element vector of positive integers.

Data Types: `double`

Output Arguments

`Y` — Indexed image
m-by-*n* matrix of integers

Indexed image, returned as an *m*-by-*n* matrix of integers. `Y` has the same data type as input indexed image `X`.

Data Types: `double` | `uint8`

`newmap` — Colormap with reduced colors
c-by-3 matrix

Colormap with reduced colors associated with the output indexed image `Y`, returned as a *c*-by-3 matrix with values in the range [0, 1]. Each row of `newmap` is a three-element RGB triplet that specifies the red, green, and blue components of a single color of the colormap.

Data Types: `double`

See Also

`randperm` | `sort`

Topics

"Image Types"

Introduced before R2006a

cmunique

Eliminate duplicate colors in colormap; convert grayscale or truecolor image to indexed image

Syntax

```
[Y,newmap] = cmunique(X,map)
[Y,newmap] = cmunique(RGB)
[Y,newmap] = cmunique(I)
```

Description

`[Y,newmap] = cmunique(X,map)` removes duplicate rows from the colormap `map` to produce a new colormap, `newmap`. The function also adjusts the indices in intensity image `X` to maintain correspondence between the indices and the colormap, and returns the result in `Y`. The image `Y` and associated colormap `newmap` produce the same image as `X` and `map` but with the smallest possible colormap.

`[Y,newmap] = cmunique(RGB)` converts the truecolor image `RGB` to the indexed image `Y` and its associated colormap, `newmap`. The returned colormap is the smallest possible colormap for the image, containing one entry for each unique color in `RGB`.

Note `newmap` might be very large, because the number of entries can be as many as the number of pixels in `RGB`.

`[Y,newmap] = cmunique(I)` converts the grayscale image `I` to an indexed image `Y` and its associated colormap, `newmap`. The returned colormap is the smallest possible colormap for the image, containing one entry for each unique intensity level in `I`.

Examples

Eliminate Duplicate Entries in Colormap

Use the `magic` function to define `X` as a 4-by-4 array that uses every value in the range between 1 and 16.

```
X = magic(4);
```

Use the `gray` function to create an eight-entry colormap. Then, concatenate the two eight-entry colormaps to create a colormap with 16 entries, `map`. In `map`, entries 9 through 16 are duplicates of entries 1 through 8.

```
map = [gray(8); gray(8)];
size(map)
```

```
ans = 1×2

    16     3
```

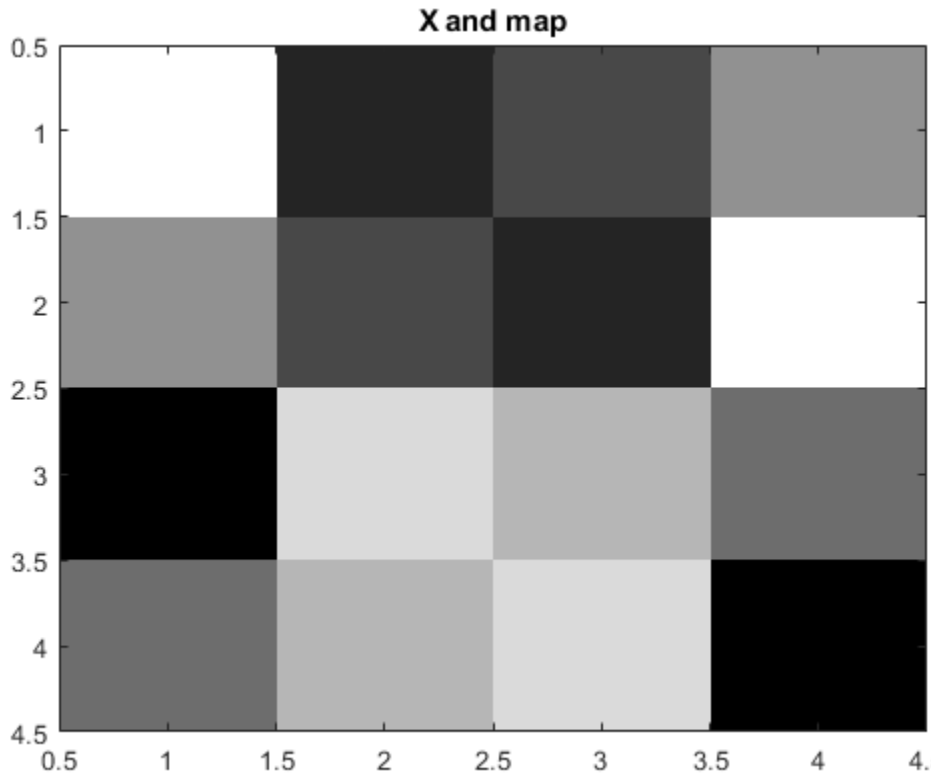
Use `cmunique` to eliminate duplicate entries in the colormap.

```
[Y, newmap] = cmunique(X, map);
size(newmap)
```

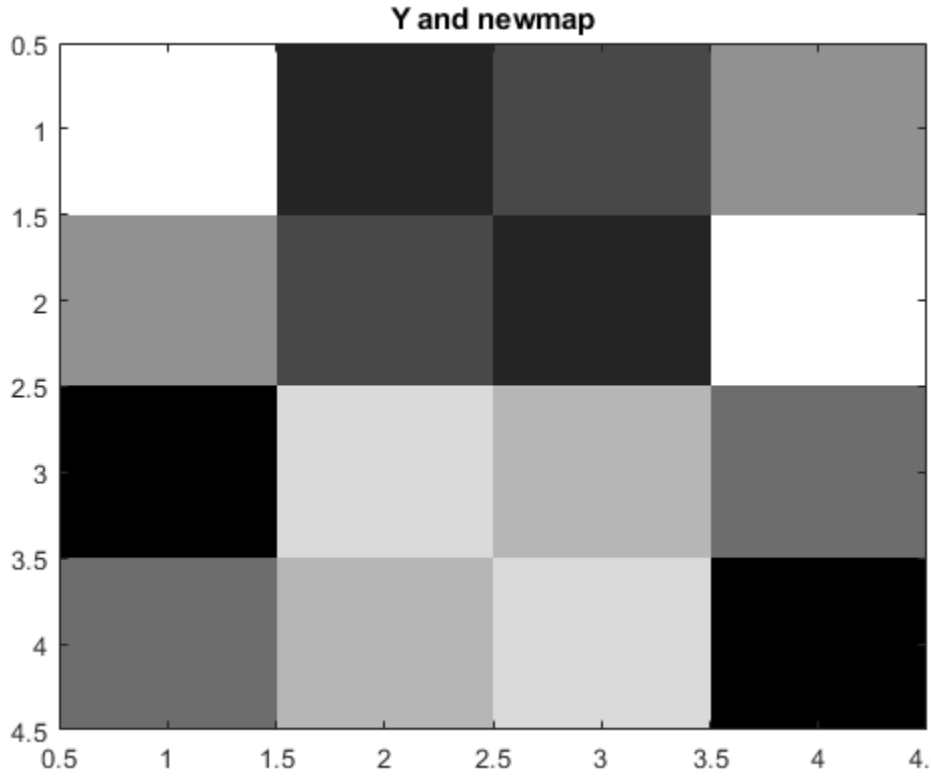
```
ans = 1×2

     8     3
```

`cmunique` adjusts the values in the original image `X` so that `Y` and `newmap` produce the same image as `X` and `map`.

```
figure
image(X)
colormap(map)
title('X and map')
```




```
figure
image(Y)
colormap(newmap)
title('Y and newmap')
```



Input Arguments

X — Indexed image with duplicate colors
m-by-*n* matrix of integers

Indexed image with duplicate colors, specified as an *m*-by-*n* matrix of integers.

Data Types: `double` | `uint8` | `uint16`

`map` — Colormap with duplicate colors
c_1-by-3 matrix

Colormap with duplicate colors associated with indexed image `X`, specified as a c_1-by-3 matrix with values in the range [0, 1]. Each row of `map` is a three-element RGB triplet that specifies the red, green, and blue components of a single color of the colormap.

Data Types: `double`

RGB — RGB image
m-by-*n*-by-3 array of nonnegative numbers

RGB image, specified as an *m*-by-*n*-by-3 array of nonnegative numbers.

Data Types: `double` | `uint8` | `uint16`

I — Grayscale image
m-by-*n* numeric matrix

Grayscale image, specified as an *m*-by-*n* numeric matrix.

Data Types: `double` | `uint8` | `uint16`

Output Arguments

Y — Indexed image with unique colors
m-by-*n* matrix of integers

Indexed image with unique colors, returned as an *m*-by-*n* matrix of integers. If the length of `newmap` is less than or equal to 256, then the output image is of class `uint8`. Otherwise, the output image is of class `double`.

Data Types: `double` | `uint8`

newmap — Colormap with unique colors
c_2-by-3 matrix

Colormap with unique colors associated with the output indexed image `Y`, returned as a c_2-by-3 matrix with values in the range [0, 1]. Each row of `newmap` is a three-element RGB triplet that specifies the red, green, and blue components of a single color of the colormap.

Data Types: `double`

See Also

`rgb2ind`

Topics

"Image Types"

Introduced before R2006a

CodeCompatibilityAnalysis

Code compatibility analysis results

Description

Use the `CodeCompatibilityAnalysis` object to save or report results from a code compatibility analysis.

Creation

Create a code compatibility analysis object using the `analyzeCodeCompatibility` function.

Properties

`Date` — Date of code compatibility analysis
datetime scalar

This property is read-only.

Date of code compatibility analysis, returned as a datetime scalar.

Data Types: `datetime`

`MATLABVersion` — Version of MATLAB
string scalar

This property is read-only.

Version of MATLAB used for code compatibility analysis, returned as a string scalar.

Data Types: `string`

`Files` — List of files analyzed
string array

This property is read-only.

List of files analyzed by for code compatibility, returned as a string array.

Data Types: `string`

`ChecksPerformed` — List of checks performed and frequency of occurrence
table

This property is read-only.

List of the checks performed and the frequency of occurrence in the analyzed files, returned as a table with these columns.

Table Column	Description
Identifier	Identifier for MATLAB code analyzer messages.
Description	Description of check. For example, *Use a newline, semicolon, or comma before this statement* or *STRMATCH is not recommended. Use STRNCMP or VALIDATESTRING instead*. Description is same as code analyzer message.
Documentation	Command to open more information in documentation.
Severity	Severity of check. For example, `Error` or `Warning`. Errors might indicate syntax errors or use of functionality that is no longer supported. Warnings might indicate opportunities to improve your code.
NumOccurrences	In analyzed code, total number of occurrences where a particular check is flagged.
NumFiles	In analyzed code, number of files where a particular check is flagged.

Data Types: `table`

`Recommendations` — Recommendations to update code
table

This property is read-only.

Recommendations to update code based on the analysis, returned as a table with the following columns.

Table Column	Description
Identifier	Identifier for MATLAB code analyzer messages.
Description	Description of check. For example, *Use a newline, semicolon, or comma before this statement or STRMATCH is not recommended. Use STRNCMP or VALIDATESTRING instead*. Description is same as code analyzer message.
Documentation	Command to open more information in documentation.
Severity	Severity of check. For example, `Error` or `Warning`. Errors might indicate syntax errors or use of functionality that is no longer supported. Warnings might indicate opportunities to improve your code.
File	Name of file that contains flagged check.
LineNumber	Line number indicating location in file of flagged check.
ColumnRange	Column range indicating location in file of flagged check.

Data Types: `table`

Examples

Analyze Code in Current Folder

Create code compatibility analysis results for code in your current folder and subfolders.

```
r = analyzeCodeCompatibility
```

```
r = 

  CodeCompatibilityAnalysis with properties:

               Date: 20-Apr-2017 15:06:06
      MATLABVersion: "R2017b"
              Files: [92×1 string]
    ChecksPerformed: [299×6 table]
    Recommendations: [28×7 table]
```

Analyze the code again and omit subfolders from the analysis.

```
r = analyzeCodeCompatibility('IncludeSubfolders',false)
```

```
r =

  CodeCompatibilityAnalysis with properties:

              Date: 20-Apr-2017 15:06:56
     MATLABVersion: "R2017b"
             Files: [5×1 string]
    ChecksPerformed: [299×6 table]
    Recommendations: [1×7 table]
```

See Also

analyzeCodeCompatibility | codeCompatibilityReport

Topics

"MATLAB Code Compatibility Report"

Introduced in R2017b

codeCompatibilityReport

Create code compatibility report

Syntax

```
codeCompatibilityReport
codeCompatibilityReport(names)
codeCompatibilityReport( ___ ,'IncludeSubfolders',tf)
```

```
codeCompatibilityReport(cca)
```

Description

`codeCompatibilityReport` creates a code compatibility report for the current working folder and subfolders. The code compatibility report on page 1-1884 opens in the MATLAB Web Browser. After you upgrade to a newer version of MATLAB, you can use this report to identify potential compatibility issues in your existing code.

`codeCompatibilityReport(names)` creates a report for the files or folders specified by `names`.

`codeCompatibilityReport( ___ ,'IncludeSubfolders',tf)` specifies whether to report on subfolders. By default, subfolders are included in the analysis. Use this syntax with any of the arguments in previous syntaxes.

`codeCompatibilityReport(cca)` creates a report from an existing code compatibility result, specified as a `CodeCompatibilityAnalysis` object.

Examples

Create Report for Code in Current Folder

Create a code compatibility report for code in your current folder and subfolders. The report opens in the MATLAB Web Browser.

```
codeCompatibilityReport
```

Create another report, but omit subfolders from the analysis.

```
codeCompatibilityReport('IncludeSubfolders',false)
```

Create Report for Existing Analysis Results

Generate compatibility analysis results for code in your current working folder and subfolders.

```
r = analyzeCodeCompatibility;
```

Create a report from the results. The report opens in the MATLAB Web Browser.

```
codeCompatibilityReport(r)
```

Input Arguments

`names` — Name of files or folders to analyze
character vector | cell array of character vectors | string scalar | string array

Name of files or folders to analyze, specified as a character vector, cell array of character vectors, string scalar, or string array.

The name of a file must be a valid MATLAB code or App file (`.m`, `.mlx`, or `.mlapp`).

Example: `'../thisFile.m'`

Example: `{'folderA','folderB','fileA'}`

Data Types: `char` | `string`

`tf` — Subfolder analysis indicator
`true` (default) | `false` | 0 | 1

Subfolder analysis indicator, specified as `true`, `false`, 0, or 1. By default, `analyzeCodeCompatibility` includes subfolders in the analysis. To omit subfolders, set `tf` to `false` or 0.

`cca` — Code compatibility analysis results
`CodeCompatibilityAnalysis` object

Code compatibility analysis results, specified as a `CodeCompatibilityAnalysis` object.

More About

Code Compatibility Report

The code compatibility report contains information to help you upgrade your code to a new version of MATLAB. It contains these sections.

- **Syntax Errors** — Table with details about syntax errors. For example, *Using ~ to ignore a value is not permitted in this context*. Syntax errors result in nonrunnable code and, while they are not introduced with a new version of MATLAB, syntax errors impact compatibility analysis.

  The table includes this information.

Report Column	Description
Occurrences	Total number of syntax errors in file.
Filename	Name of file that contains syntax errors.

- **Functionality that has been removed** — Table with details about functionality that is being used in the analyzed code and that has been removed. For example, '*wavfinfo*' *has been removed. Use* '*AUDIOINFO*' *instead*.

  The table includes this information.

Report Column	Description
Description	Description of check. Description is same as code analyzer message.
Documentation	Link to more information in documentation.
Removed In	The release in which the functionality was removed.
Filename	Name of file that contains flagged check.
Line	Line number indicating location in file of flagged check.

- **Functionality that has changed behavior** — Table with details about functionality that is being used in the analyzed code and that has changed behavior. For example, '*legend*' *has changed and might interpret the name of an argument as a legend*

property instead of a label. To include a label with the same name as a legend property, specify the labels using a cell array or string array. Refer to the documentation for a list of affected property names.

The table includes this information.

Report Column	Description
Description	Description of check. Description is same as code analyzer message.
Documentation	Link to more information in documentation.
Affects Code Written Before	The release in which the functionality was changed.
Filename	Name of file that contains flagged check.
Line	Line number indicating location in file of flagged check.

- **Functionality that will be removed** — Table with details about functionality that is being used in the analyzed code and that will be removed in a future release. For example, '*aviinfo*' *will be removed in a future release. Use* '*VideoReader*' *instead.*

  The table includes this information.

Report Column	Description
Description	Description of check. Description is same as code analyzer message.
Documentation	Link to more information in documentation.
Filename	Name of file that contains flagged check.
Line	Line number indicating location in file of flagged check.

- **Functionality that will change behavior** — Table with details about functionality that is being used in the analyzed code and that will change behavior in a future release. For example, '*interp1(...,'cubic')*' *will change in a future release to perform cubic convolution. To continue using shape-preserving piecewise cubic interpolation, use* '*interp1(...,'pchip')*' *instead.*

  The table includes the same columns as the **Functionality that will be removed** section.

- **Functionality that is not recommended** — Table with opportunities to improve your code. For example, *Programmatic use of DISPLAY is not recommended. Use DISP or FPRINTF instead.*

  The table includes this information.

Report Column	Description
Description	Description of check. Description is same as code analyzer message.
Documentation	Link to more information in documentation.
Filename	Name of file that contains flagged check.
Line	Line number indicating location in file of flagged check.
Suppression	Whether and how this check is suppressed in the Editor. A value of *None* indicates that the check is not suppressed.

- **Full list of checks performed** — Table with information about the checks performed on the specified code. It includes the following information.

Report Column	Description
Occurrences	In analyzed code, total number of occurrences where a particular check is flagged.
Files	In the analyzed code, number of files where a particular check is flagged.
Description	Description of check. For example, *Use a newline, semicolon, or comma before this statement* or *STRMATCH is not recommended. Use STRNCMP or VALIDATESTRING instead*. Description is same as code analyzer message.
Severity	Severity of check. For example, `Error` or `Warning`. Errors might indicate syntax errors or use of functionality that is no longer supported. Warnings might indicate opportunities to improve your code.
Identifier	Identifier for MATLAB code analyzer messages.
Documentation	Link to more information in documentation.

- **Full list of files analyzed** — List of files that MATLAB analyzed for code compatibility.

Tips

- To save a report, instead of generating a report with the `codeCompatibilityReport` function, you can create a `CodeCompatibilityAnalysis` object and save it. Then, you can generate a report from those saved results. For more information, see “Create Report for Existing Analysis Results” on page 1-1883.

Alternatives

Generate a Code Compatibility Report for code in the current working folder and subfolders in the Current Folder browser. In the Current Folder browser, either click or right-click the white space. Then select **Reports** > **Code Compatibility Report**. See “MATLAB Code Compatibility Report”.

See Also

`CodeCompatibilityAnalysis` | `analyzeCodeCompatibility`

Topics

“MATLAB Code Compatibility Report”
“Check Code for Errors and Warnings”

Introduced in R2017b

colamd

Column approximate minimum degree permutation

Syntax

```
p = colamd(S)
```

Description

`p = colamd(S)` returns the column approximate minimum degree permutation vector for the sparse matrix `S`. For a non-symmetric matrix `S`, `S(:,p)` tends to have sparser LU factors than `S`. The Cholesky factorization of `S(:,p)' * S(:,p)` also tends to be sparser than that of `S'*S`.

`knobs` is a two-element vector. If S is `m`-by-`n`, then rows with more than `(knobs(1))*n` entries are ignored. Columns with more than `(knobs(2))*m` entries are removed prior to ordering, and ordered last in the output permutation `p`. If the `knobs` parameter is not present, then `knobs(1) = knobs(2) = spparms('wh_frac')`.

`stats` is an optional vector that provides data about the ordering and the validity of the matrix `S`.

`stats(1)`	Number of dense or empty rows ignored by `colamd`
`stats(2)`	Number of dense or empty columns ignored by `colamd`
`stats(3)`	Number of garbage collections performed on the internal data structure used by `colamd` (roughly of size `2.2*nnz(S) + 4*m + 7*n` integers)
`stats(4)`	`0` if the matrix is valid, or `1` if invalid
`stats(5)`	Rightmost column index that is unsorted or contains duplicate entries, or `0` if no such column exists
`stats(6)`	Last seen duplicate or out-of-order row index in the column index given by `stats(5)`, or `0` if no such row index exists

`stats(7)`	Number of duplicate and out-of-order row indices

Although MATLAB built-in functions generate valid sparse matrices, a user may construct an invalid sparse matrix using the MATLAB C or Fortran APIs and pass it to `colamd`. For this reason, `colamd` verifies that `S` is valid:

- If a row index appears two or more times in the same column, `colamd` ignores the duplicate entries, continues processing, and provides information about the duplicate entries in `stats(4:7)`.
- If row indices in a column are out of order, `colamd` sorts each column of its internal copy of the matrix `S` (but does not repair the input matrix `S`), continues processing, and provides information about the out-of-order entries in `stats(4:7)`.
- If `S` is invalid in any other way, `colamd` cannot continue. It prints an error message, and returns no output arguments (`p` or `stats`) .

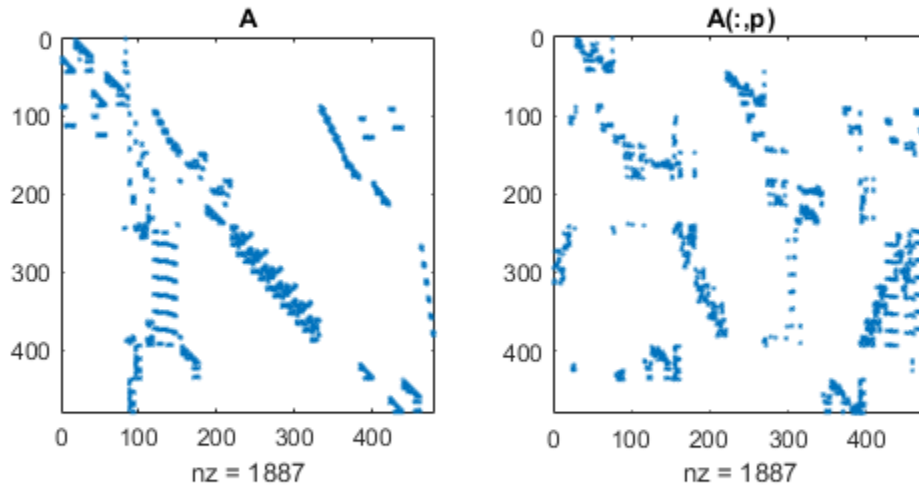
The ordering is followed by a column elimination tree post-ordering.

Examples

Compare Sparse Matrix and LU Factorization

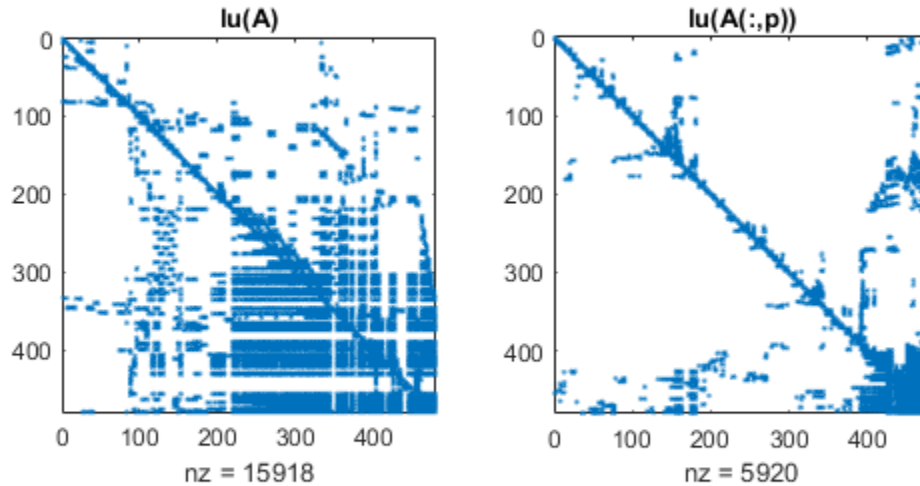
The Harwell-Boeing collection of sparse matrices and the MATLAB® demos directory include a test matrix `west0479`. It is a matrix of order 479 resulting from a model due to Westerberg of an eight-stage chemical distillation column. The spy plot shows evidence of the eight stages. The `colamd` ordering scrambles this structure.

```
load west0479
A = west0479;
p = colamd(A);

figure()
subplot(1,2,1), spy(A,4), title('A')
subplot(1,2,2), spy(A(:,p),4), title('A(:,p)')
```



Comparing the spy plot of the LU factorization of the original matrix with that of the reordered matrix shows that minimum degree reduces the time and storage requirements by better than a factor of 2.8. The nonzero counts are 15918 and 5920, respectively.

```
figure()
subplot(1,2,1), spy(lu(A),4), title('lu(A)')
subplot(1,2,2), spy(lu(A(:,p)),4), title('lu(A(:,p))')
```



References

[1] The authors of the code for `colamd` are Stefan I. Larimore and Timothy A. Davis. The algorithm was developed in collaboration with John Gilbert, Xerox PARC, and Esmond Ng, Oak Ridge National Laboratory. Sparse Matrix Algorithms Research: http://faculty.cse.tamu.edu/davis/research.html

See Also

`colperm` | `dissect` | `spparms` | `symamd` | `symrcm`

Introduced before R2006a

colorbar

Colorbar showing color scale

Syntax

```
colorbar
colorbar(location)
colorbar( ___ ,Name,Value)
colorbar(target, ___ )
c = colorbar( ___ )

colorbar('off')
colorbar(target,'off')
```

Description

`colorbar` displays a vertical colorbar to the right of the current axes or chart. Colorbars display the current colormap and indicate the mapping of data values into the colormap.

`colorbar(location)` displays the colorbar in a specific location such as `'northoutside'`. Not all types of charts support modifying the colorbar location.

`colorbar( ___ ,Name,Value)` modifies the colorbar appearance using one or more name-value pair arguments. For example, `'Direction','reverse'` reverses the color scale. Specify `Name,Value` as the last pair of arguments in any of the previous syntaxes. Not all types of charts support modifying the colorbar appearance.

`colorbar(target, ___ )` adds a colorbar to the axes or chart specified by `target`. Specify the target axes or chart as the first argument in any of the previous syntaxes.

`c = colorbar( ___ )` returns the `ColorBar` object. You can use this object to set properties after creating the colorbar. Specify the return argument `c` with any of the previous syntaxes.

`colorbar('off')` deletes all colorbars associated with the current axes or chart.

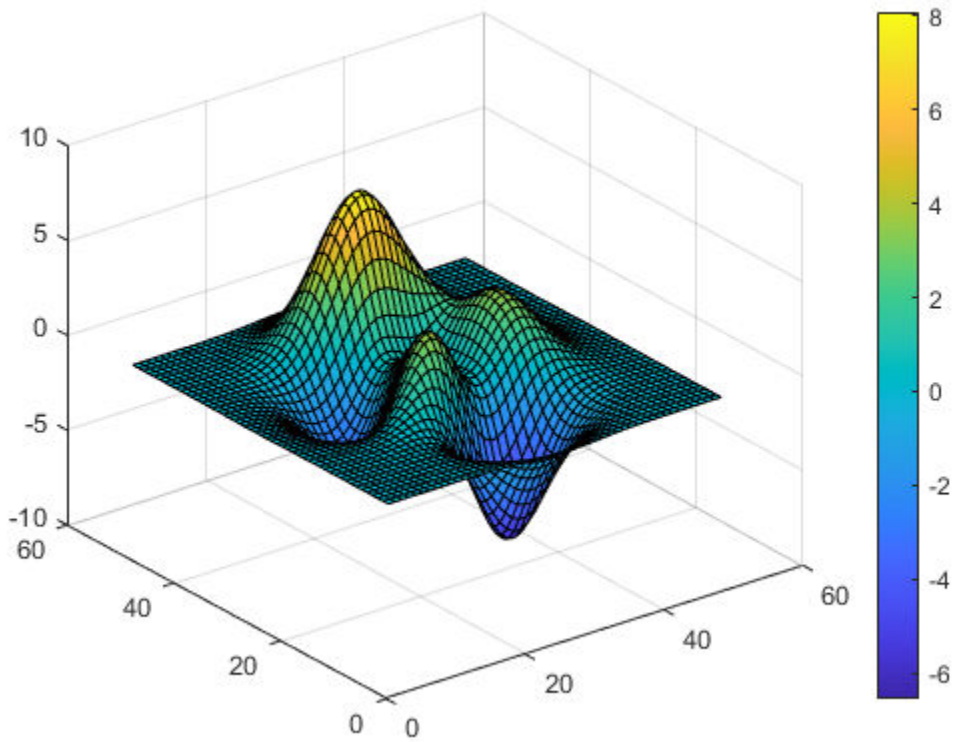
`colorbar(target,'off')` deletes all colorbars associated with the target axes or chart. Alternatively, you can specify a `ColorBar` object as the target.

Examples

Add Colorbar to Graph

Add a colorbar to a surface plot indicating the color scale.

```
surf(peaks)
colorbar
```

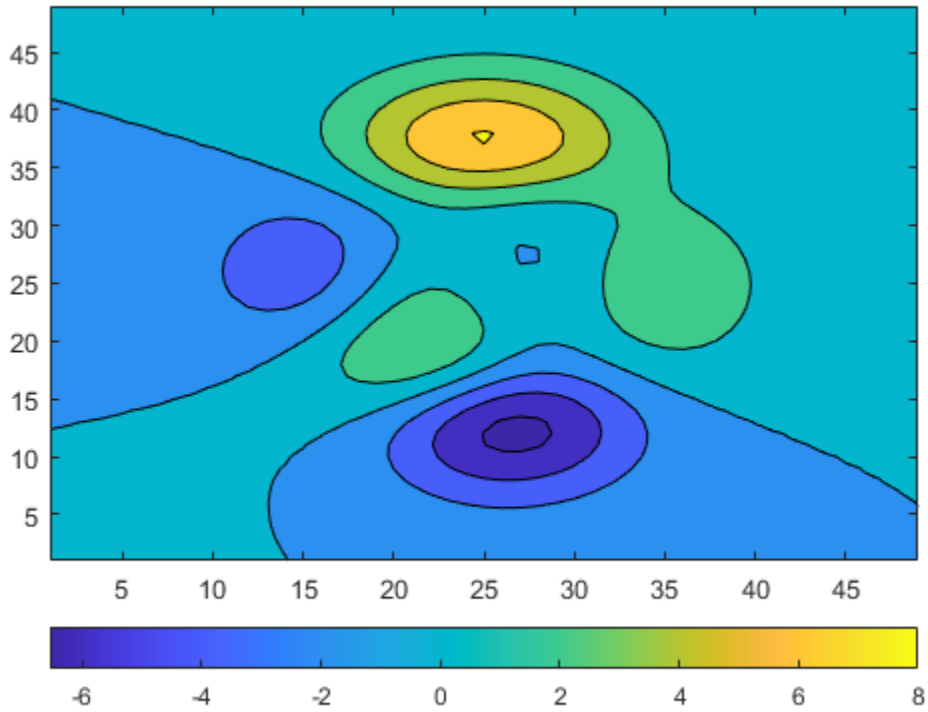


By default, the `colorbar` function adds a vertical colorbar to the right side of the graph.

Add Horizontal Colorbar to Graph

Add a horizontal colorbar below a plot by specifying the colorbar location as `'southoutside'`.

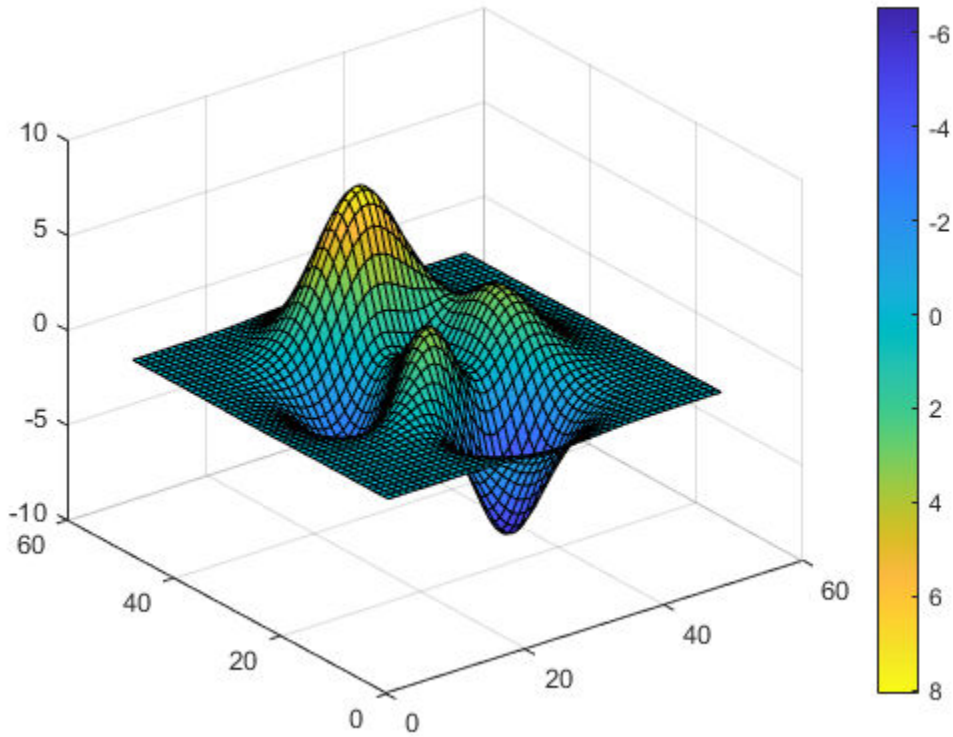
```
contourf(peaks)
colorbar('southoutside')
```



Reverse Colorbar Direction

Reverse the direction of values in a colorbar on a graph by setting the `'Direction'` property of the colorbar to `'reverse'`.

```
surf(peaks)
colorbar('Direction','reverse')
```

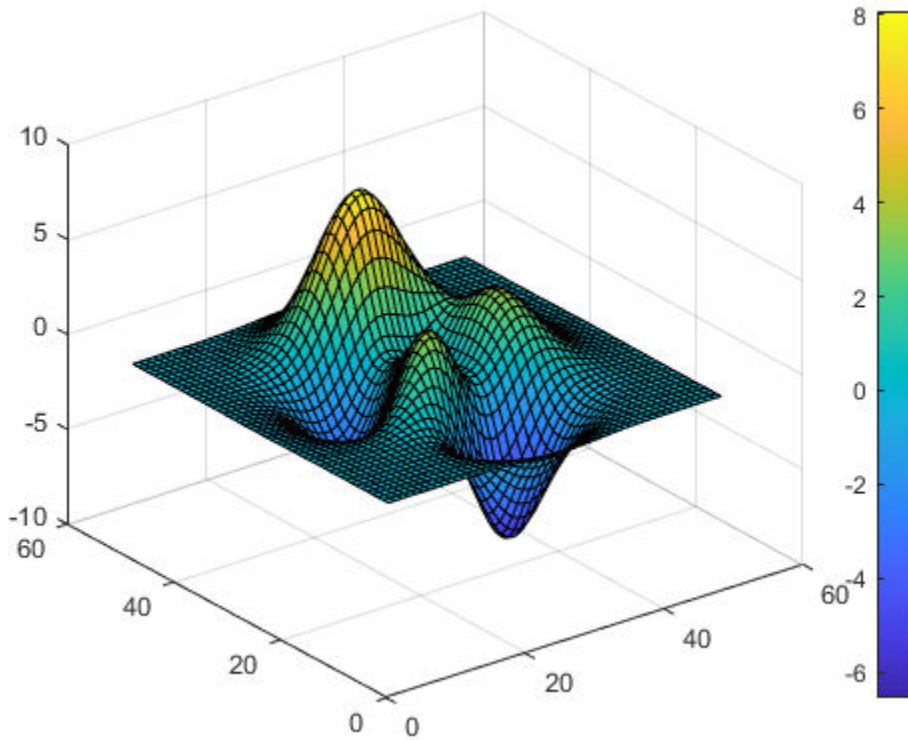


The colorbar values ascend from top to bottom instead of ascending from bottom to top.

Display Colorbar Ticks on Opposite Side

Display the colorbar tick marks and tick labels on the side of a colorbar facing the surface plot.

```
surf(peaks)
colorbar('AxisLocation','in')
```

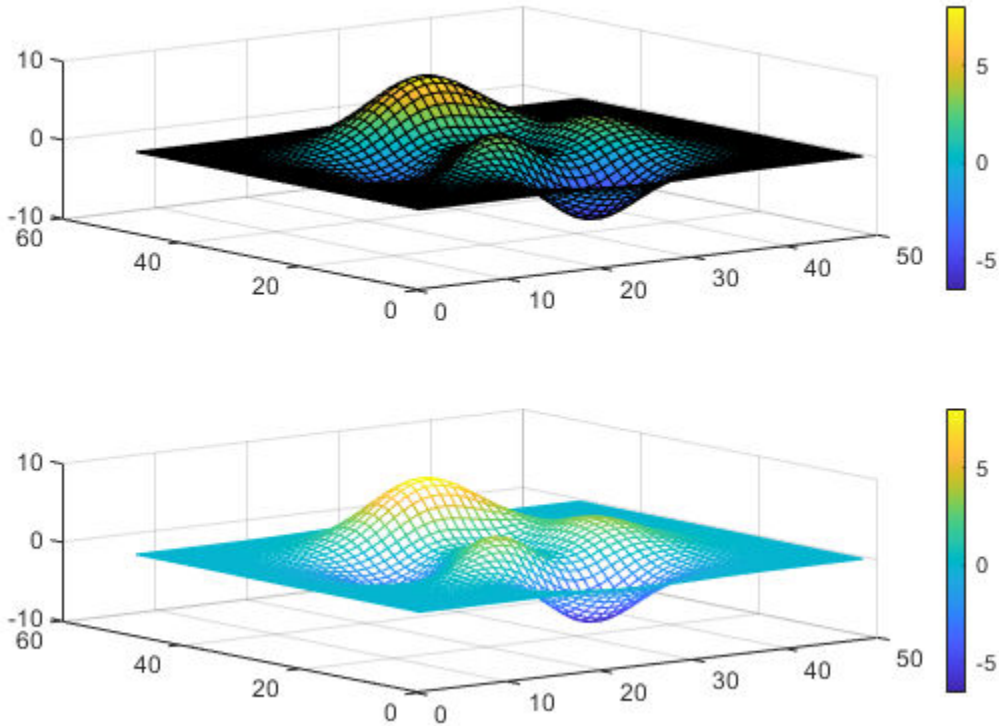


Add Colorbars to Tiled Chart Layout

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout.

Call the `nexttile` function to create the axes. Then display a surface plot in each axes with a colorbar.

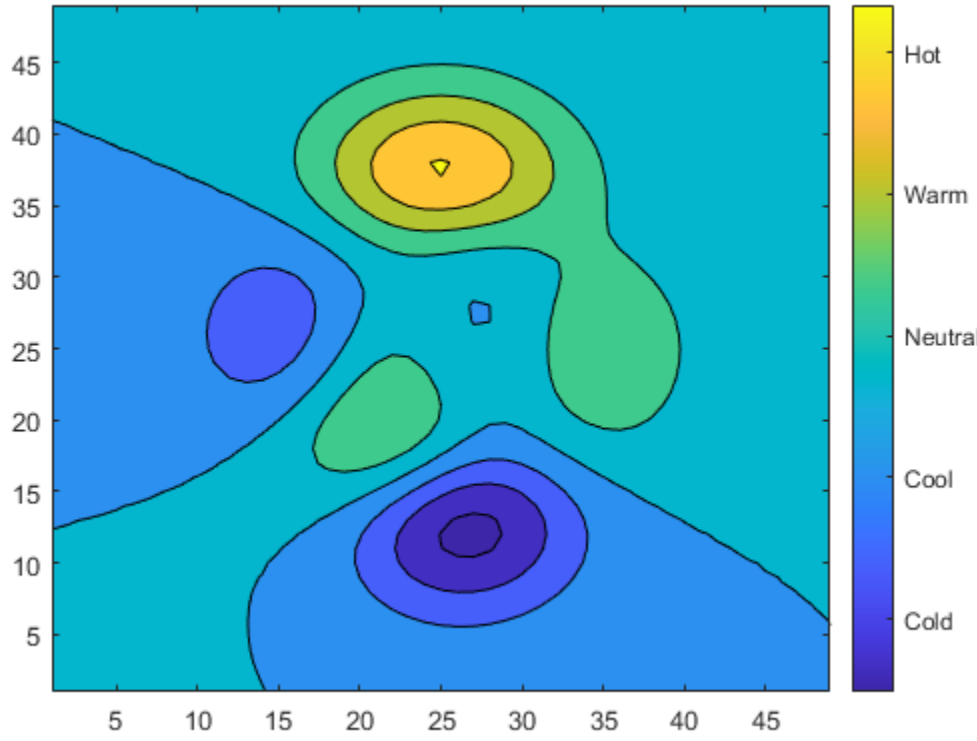
```
tiledlayout(2,1)

% Top plot
nexttile
surf(peaks)
colorbar

% Bottom plot
nexttile
mesh(peaks)
colorbar
```



Specify Colorbar Ticks and Tick Labels

Add a colorbar to a plot and specify the colorbar tick marks and tick labels. Specify the same number of tick labels as tick marks. If you do not specify enough tick labels, then the `colorbar` function repeats the labels.

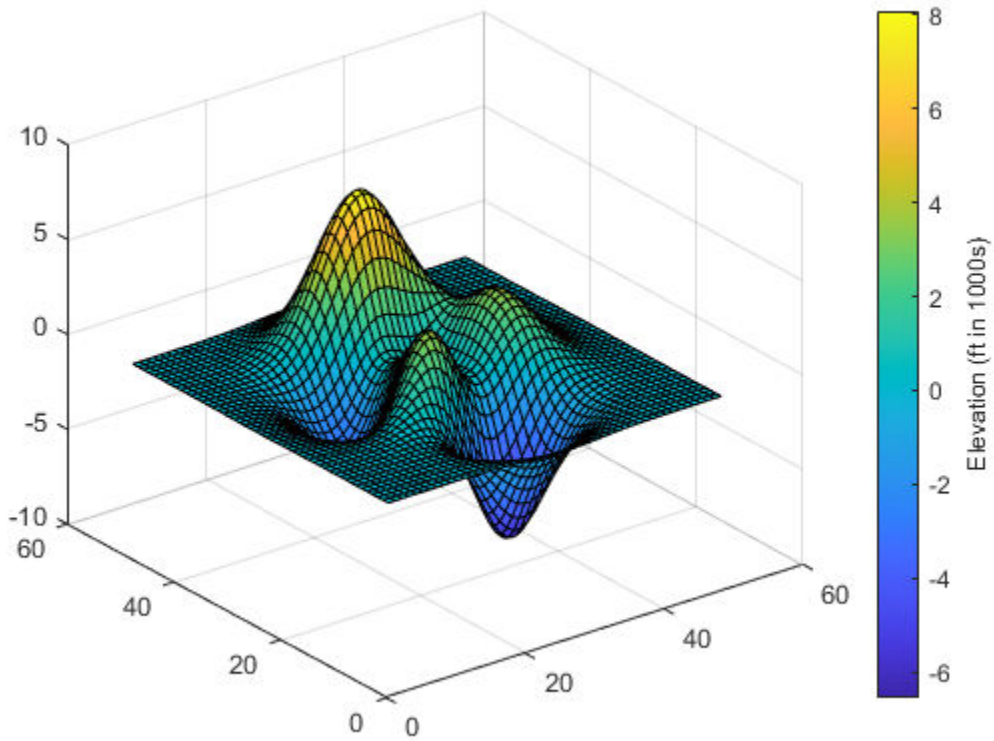
```
contourf(peaks)
colorbar('Ticks',[-5,-2,1,4,7],...
         'TickLabels',{'Cold','Cool','Neutral','Warm','Hot'})
```



Label Colorbar

Add a text label along a colorbar.

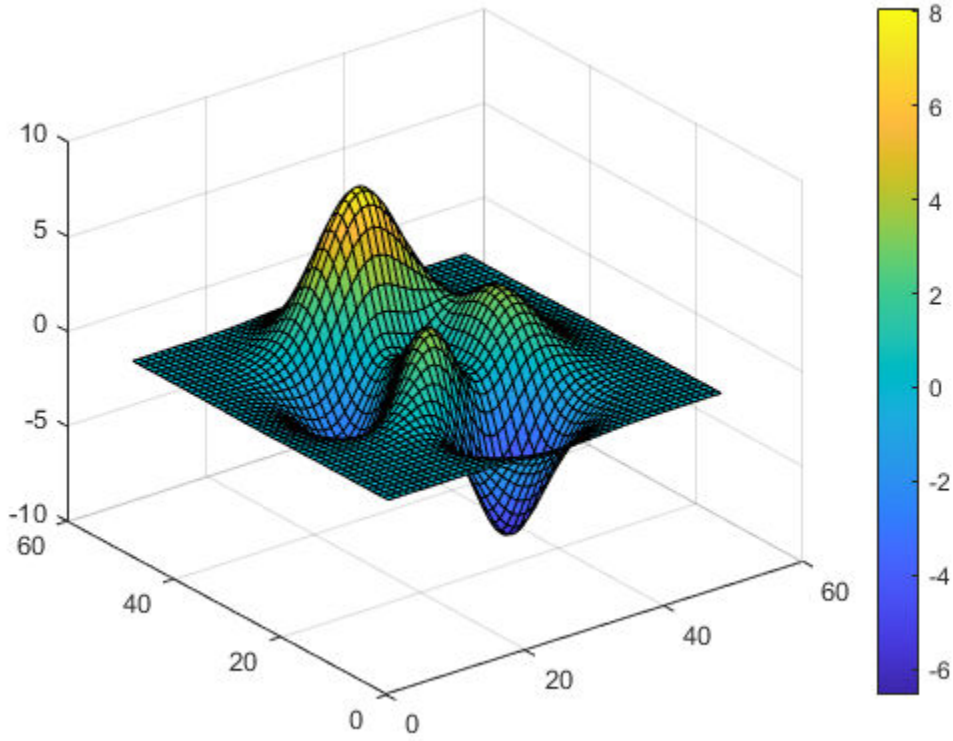
```
surf(peaks)
c = colorbar;
c.Label.String = 'Elevation (ft in 1000s)';
```



Delete Colorbar

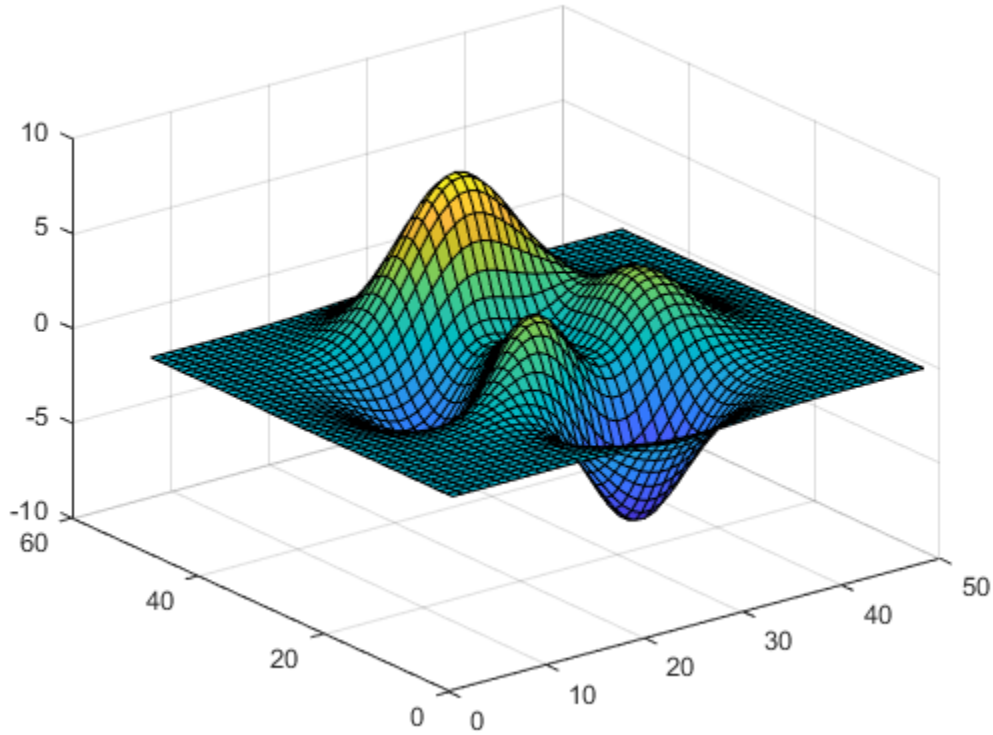
Add a colorbar to a surface plot.

```
surf(peaks)
colorbar
```



Delete the colorbar from the surface plot.

```
colorbar('off')
```



Input Arguments

`location` — Location

`'eastoutside'` (default) | `'north'` | `'south'` | `'east'` | `'west'` | `'northoutside'` | `...`

Location of the colorbar with respect to the axes, specified as one of the values in this table.

Value	Resulting Location	Resulting Orientation
`'north'`	Top of axes	Horizontal
`'south'`	Bottom of axes	Horizontal
`'east'`	Right side of axes	Vertical
`'west'`	Left side of axes	Vertical
`'northoutside'`	Top outside of axes	Horizontal
`'southoutside'`	Bottom outside of axes	Horizontal
`'eastoutside'`	Right outside of axes (default)	Vertical
`'westoutside'`	Left outside of axes	Vertical

If a colorbar already exists in the specified location, then an updated colorbar replaces the existing one. To ensure that the colorbar does not overlap the graph, specify a location with the suffix, `outside`.

You also can set the colorbar location using its `Location` property. For example, `colorbar('Location','northoutside')` is the same as `colorbar('northoutside')`.

Example: `colorbar('westoutside')`

`target` — Target

`Axes` object | `PolarAxes` object | `GeographicAxes` object | graphics object

Target object that contains the associated data visualization, specified as an `Axes` object, a `PolarAxes` object, a `GeographicAxes` object, or a graphics object that has a `ColorbarVisible` property. For example, a `HeatmapChart` object has the `ColorbarVisible` property.

If you do not specify the target, then the `colorbar` command affects the current axes (the object returned by `gca`).

Note Some charts do not support modifying the colorbar appearance, such as the location, or returning the `ColorBar` object as an output argument.

Name-Value Pair Arguments

The colorbar properties listed here are only a subset. For a complete list see Colorbar.

Example: `colorbar('FontSize',12,'Direction','reverse')` sets the font size of the colorbar to 12 points and reverses the orientation of the colorbar.

`Location` — Location with respect to the axes

`'eastoutside'` (default) | `'north'` | `'south'` | `'east'` | `'west'` | `'northoutside'` | `...`

Location with respect to the axes, specified as one of the values listed in this table.

Value	Resulting Location	Resulting Orientation
`'north'`	Top of axes	Horizontal
`'south'`	Bottom of axes	Horizontal
`'east'`	Right side of axes	Vertical
`'west'`	Left side of axes	Vertical
`'northoutside'`	Top outside of axes	Horizontal
`'southoutside'`	Bottom outside of axes	Horizontal
`'eastoutside'`	Right outside of axes (default)	Vertical
`'westoutside'`	Left outside of axes	Vertical
`'manual'`	Determined by `Position` property	Vertical

To display the colorbar in a location that does not appear in the table, use the `Position` property to specify a custom location. If you set the `Position` property, then MATLAB sets the `Location` property to `'manual'`. The associated axes does not resize to accommodate the colorbar when the `Location` property is set to `'manual'`.

`TickLabels` — Tick mark labels

cell array of character vectors | string array | numeric array | character vector | categorical array

Tick mark labels, specified as a cell array of character vectors, a string array, a numeric array, a character vector, or a categorical array. By default, the colorbar labels the tick

marks with numeric values. If you specify labels and do not specify enough labels for all the tick marks, then MATLAB cycles through the labels.

If you specify this property as a categorical array, MATLAB uses the values in the array, not the categories.

Example: `{'cold','warm','hot'}`

`TickLabelInterpreter` — Interpretation of characters in tick labels

`'tex'` (default) | `'latex'` | `'none'`

Interpretation of tick label characters, specified as one of these values:

- `'tex'` — Interpret labels using a subset of the TeX markup.
- `'latex'` — Interpret labels using a subset of LaTeX markup.
- `'none'` — Display literal characters

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the text type and color, and include special characters in the labels.

This table lists the supported modifiers when the `TickLabelInterpreter` property is set to `'tex'`, which is the default value. Most modifiers remain in effect until the end of the text. Superscripts and subscripts modify only the next character or the text within the curly braces `{}`.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (rarely available)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`

Modifier	Description	Example
`\fontname{specifier}`	Set `specifier` as the name of a font family to change the font style. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{specifier}`	Set `specifier` as a scalar numeric value to change the font size.	`'\fontsize{15} text'`
`\color{specifier}`	Set `specifer` as one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Set `specifier` as a three-element RGB triplet to change the font color.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	β	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ϵ	`\Delta`	Δ	`\spadesuit`	♠
`\zeta`	ζ	`\Theta`	Θ	`\leftrightarrow`	↔

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	µ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `TickLabelInterpreter` property to `'latex'`. The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup within the text.

The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters. For multiline text, the maximum size of the text reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project Web site at https://www.latex-project.org/.

`Ticks` — Tick mark locations

vector of monotonically increasing numeric values

Tick mark locations, specified as a vector of monotonically increasing numeric values. The values do not need to be equally spaced. If you do not want tick marks displayed, then set the property to the empty vector, `[]`.

Example: `[-1,0,1,2,3,4,5]`

Data Types: `single` | `double`

`Direction` — Direction of color scale

`'normal'` (default) | `'reverse'`

Direction of color scale, specified as one of these values:

- `'normal'` — Display the colormap and labels ascending from bottom to top for a vertical colorbar, and ascending from left to right for a horizontal colorbar.
- `'reverse'` — Display the colormap and labels descending from bottom to top for a vertical colorbar, and descending from left to right for a horizontal colorbar.

`FontSize` — Font size

scalar value greater than zero

Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale.

If you change the axes font size, then MATLAB automatically sets the font size of the colorbar to 90% of the axes font size. If you manually set the font size of the colorbar, then changing the axes font size does not affect the colorbar font.

Tips

- To add a text description along the colorbar, access the underlying text object using the `Label` property of the colorbar.

  ```
  c.Label.String = 'My Colorbar Label';
  ```

  To change the label appearance, such as the font style or color, set other text properties. For a list of properties, see Text. For example, this code changes the font size.

  ```
  c.Label.FontSize = 12;
  ```
- Adding a colorbar might resize the axes to accommodate the colorbar.
- If an axes does not exist, then the `colorbar` function creates a blank axes and displays a colorbar with the default colormap.
- You can use `colorbar('delete')` or `colorbar('hide')` instead of `colorbar('off')` to delete all colorbars in the current axes. All of these commands are equivalent.

Compatibility Considerations

`'peer'` input is not recommended

Not recommended starting in R2014b

Starting in R2014b, `colorbar('peer',target)` is not recommended and might be removed in a future release. Use `colorbar(target)` instead.

See Also

Functions

`caxis` | `colormap`

Properties
Colorbar

Topics

"Creating Colorbars"
"Control Colormap Limits"

Introduced before R2006a

ColorBar Properties

Colorbar appearance and behavior

Description

`ColorBar` properties control the appearance and behavior of a `ColorBar` object. By changing property values, you can modify certain aspects of the colorbar. Use dot notation to refer to a particular object and property:

```
c = colorbar;
w = c.LineWidth;
c.LineWidth = 1.5;
```

Properties

Ticks and Labels

`Ticks` — Tick mark locations

vector of monotonically increasing numeric values

Tick mark locations, specified as a vector of monotonically increasing numeric values. The values do not need to be equally spaced. If you do not want tick marks displayed, then set the property to the empty vector, `[]`.

Example: `[-1,0,1,2,3,4,5]`

Data Types: `single` | `double`

`TicksMode` — Selection mode for `Ticks`

`'auto'` (default) | `'manual'`

Selection mode for `Ticks`, specified as one of these values:

- `'auto'` — Automatically choose tick values.
- `'manual'` — Manually specify tick values. To specify the tick values, set the `Ticks` property.

TickLabels — Tick mark labels

cell array of character vectors | string array | numeric array | character vector | categorical array

Tick mark labels, specified as a cell array of character vectors, a string array, a numeric array, a character vector, or a categorical array. By default, the colorbar labels the tick marks with numeric values. If you specify labels and do not specify enough labels for all the tick marks, then MATLAB cycles through the labels.

If you specify this property as a categorical array, MATLAB uses the values in the array, not the categories.

Example: `{'cold','warm','hot'}`

TickLabelsMode — Selection mode for TickLabels

`'auto'` (default) | `'manual'`

Selection mode for `TickLabels`, specified as one of these values:

- `'auto'` — Automatically choose the tick labels.
- `'manual'` — Manually specify tick labels. To specify the tick labels, set the `TickLabels` property.

TickLabelInterpreter — Interpretation of characters in tick labels

`'tex'` (default) | `'latex'` | `'none'`

Interpretation of tick label characters, specified as one of these values:

- `'tex'` — Interpret labels using a subset of the TeX markup.
- `'latex'` — Interpret labels using a subset of LaTeX markup.
- `'none'` — Display literal characters

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the text type and color, and include special characters in the labels.

This table lists the supported modifiers when the `TickLabelInterpreter` property is set to `'tex'`, which is the default value. Most modifiers remain in effect until the end of the text. Superscripts and subscripts modify only the next character or the text within the curly braces `{}`.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (rarely available)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{specifier}`	Set `specifier` as the name of a font family to change the font style. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{specifier}`	Set `specifier` as a scalar numeric value to change the font size.	`'\fontsize{15} text'`
`\color{specifier}`	Set `specifer` as one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Set `specifier` as a three-element RGB triplet to change the font color.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\beta	β	\psi	ψ	\clubsuit	♣
\gamma	γ	\omega	ω	\diamondsuit	♦
\delta	δ	\Gamma	Γ	\heartsuit	♥
\epsilon	ϵ	\Delta	Δ	\spadesuit	♠
\zeta	ζ	\Theta	Θ	\leftrightarrow	↔
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	µ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	…
\perp	⊥	\surd	√	\prime	′
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `TickLabelInterpreter` property to `'latex'`. The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup within the text.

The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters. For multiline text, the maximum size of the text reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project Web site at https://www.latex-project.org/.

`Limits` — Minimum and maximum tick mark values

two-element vector

The minimum and maximum tick mark values, specified as a two-element vector. The second vector element must be greater than the first element.

Example: `[0 1]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`LimitsMode` — Selection mode for limits

`'auto'` (default) | `'manual'`

Selection mode for limits, specified as one of these values:

- `'auto'` — Automatically choose the limits.
- `'manual'` — Use manually specified limits. To specify the limits, set the `Limits` property.

`Label` — Label
text object

Label that displays along the colorbar, returned as a text object. This text object contains properties that control the label appearance and the text that displays. Use the `Label` property to access the text object, for example:

```
c = colorbar;
c.Label
```

```
ans =

  Text with properties:

                String: ''
              FontSize: 10
            FontWeight: 'normal'
              FontName: 'Helvetica'
                 Color: [0.1500 0.1500 0.1500]
    HorizontalAlignment: 'left'
              Position: [0 0 0]
                 Units: 'data'

  Show all properties
```

To add a label, or change a label, set the `String` property for the text object, for example:

```
c.Label.String = 'Label Text Goes Here';
```

To change the label appearance, such as the font style or color, set other text properties. For example, this code changes the font size.

```
c.Label.FontSize = 12;
```

For a full list of options, see Text.

`Direction` — Direction of color scale
`'normal'` (default) | `'reverse'`

Direction of color scale, specified as one of these values:

- `'normal'` — Display the colormap and labels ascending from bottom to top for a vertical colorbar, and ascending from left to right for a horizontal colorbar.
- `'reverse'` — Display the colormap and labels descending from bottom to top for a vertical colorbar, and descending from left to right for a horizontal colorbar.

`TickLength` — Tick mark length

`0.01` (default) | scalar

Tick mark length, specified as a scalar. Specify the tick length as a fraction of the colorbar axis length.

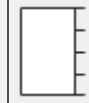
Example: `0.05`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`TickDirection` — Tick mark direction

`'in'` (default) | `'out'`

Tick mark direction, specified as one of these values:

Value	Description	Result
`'in'`	Display the tick marks inside colorbar box.	
`'out'`	Display the tick marks outside the colorbar box.	

Font

`FontName` — Font name

supported font name | `'FixedWidth'`

Font name, specified as a supported font name or `'FixedWidth'`. To display and print text properly, you must choose a font that your system supports. The default font depends on your operating system and locale.

To use a fixed-width font that looks good in any locale, use `'FixedWidth'`. The fixed-width font relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`FontSize` — Font size
scalar value greater than zero

Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale.

If you change the axes font size, then MATLAB automatically sets the font size of the colorbar to 90% of the axes font size. If you manually set the font size of the colorbar, then changing the axes font size does not affect the colorbar font.

`FontAngle` — Character slant
`'normal'` (default) | `'italic'`

Character slant, specified as `'normal'` or `'italic'`.

Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.

`FontWeight` — Character thickness
`'normal'` (default) | `'bold'`

Character thickness, specified as `'normal'` or `'bold'`.

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.

Position

`Location` — Location with respect to the axes
`'eastoutside'` (default) | `'north'` | `'south'` | `'east'` | `'west'` | `'northoutside'` | `...`

Location with respect to the axes, specified as one of the values listed in this table.

Value	Resulting Location	Resulting Orientation
`'north'`	Top of axes	Horizontal
`'south'`	Bottom of axes	Horizontal
`'east'`	Right side of axes	Vertical
`'west'`	Left side of axes	Vertical
`'northoutside'`	Top outside of axes	Horizontal
`'southoutside'`	Bottom outside of axes	Horizontal
`'eastoutside'`	Right outside of axes (default)	Vertical
`'westoutside'`	Left outside of axes	Vertical
`'manual'`	Determined by `Position` property	Vertical

To display the colorbar in a location that does not appear in the table, use the `Position` property to specify a custom location. If you set the `Position` property, then MATLAB sets the `Location` property to `'manual'`. The associated axes does not resize to accommodate the colorbar when the `Location` property is set to `'manual'`.

`AxisLocation` — Axis location
`'out'` (default) | `'in'`

Axis location, specified as one of the following values. Use this property to specify the location of the tick marks, tick labels, and colorbar label.

- `'out'` — Display the tick marks and labels on the side of the colorbar towards the outside of the figure. This is the default value.
- `'in'` — Display the tick marks and labels on the side of the colorbar towards the inside of the figure.

`AxisLocationMode` — Selection mode for `AxisLocation`
`'auto'` (default) | `'manual'`

Selection mode for `AxisLocation`, specified as one of these values:

- `'auto'` — Automatically choose the location.
- `'manual'` — Use a manually specified location. To specify the location, set the `AxisLocation` property.

Position — Custom location and size

four-element vector

Custom location and size, specified as a four-element vector of the form `[left, bottom, width, height]`. The `left` and `bottom` elements specify the distance from the lower-left corner of the figure or to the lower-left corner of the colorbar. The `width` and `height` elements specify the dimensions of the colorbar. The `Units` property determines the position units.

If you specify the `Position` property, then MATLAB changes the `Location` property to `'manual'`. The associated axes does not resize to accommodate the colorbar when the `Location` property is `'manual'`.

Example: `[0.1 0.1 0.3 0.7]`

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

Units — Position units

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'characters'` | `'points'` | `'pixels'`

Position units, specified as one of the values in this table.

Units	Description
`'normalized'` (default)	Normalized with respect to the container, which is usually the figure. The lower left corner of the figure maps to `(0,0)` and the upper right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the default system font character size. • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.

Units	Description
`'points'`	Points. One point equals 1/72 inch.
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems. • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. • On Linux systems, the size of a pixel is determined by your system resolution.

All units are measured from the lower-left corner of the container window.

This property affects the `Position` property. If you change the units, then it is good practice to return it to its default value after completing your computation to prevent affecting other functions that assume `Units` is the default value.

If you specify the `Position` and `Units` properties as `Name,Value` pairs when creating the object, then the order of specification matters. If you want to define the position with particular units, then you must set the `Units` property before the `Position` property.

Color and Styling

`Color` — Color of tick marks, text, and box outline

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of the tick marks, text, and box outline, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to

F. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `[0 1 0]`

Example: `'green'`

Example: `'#00FF00'`

Box — Box outline

`'on'` (default) | `'off'`

Box outline, specified as one of these values:

- `'on'` — Display the box outline around the colorbar.
- `'off'` — Do not display the box outline around the colorbar.

`LineWidth` — Width of box outline

`0.5` (default) | positive value

Width of box outline, specified as a positive value in point units. One point equals 1/72 inch.

Example: `1.5`

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`UIContextMenu` — Context menu

`ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles
`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

Callbacks

`ButtonDownFcn` — Mouse-click callback
`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object.

If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'off'` (default) | `'on'`

Callback interruption, specified as `'off'` or `'on'`. The `Interruptible` property determines if a running callback can be interrupted.

Consider these callback states where:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. The `Interruptible` property has two possible values:

- `'off'` — The running callback cannot be interrupted. MATLAB finishes executing the running callback without any interruptions. The `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.
- `'on'` — The running callback can be interrupted. Interruption occurs at the next point where MATLAB processes the queue. For example, when you have a command such as `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks only when visible. The `Visible` property must be set to `'on'`. The `HitTest` property determines if the `ColorBar` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `ColorBar` object passes the click to the object behind it in the current view of the figure window. The `HitTest` property of the `ColorBar` object has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `ColorBar` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `ColorBar` object that has one of these:
  - `HitTest` property set to `'on'`

- `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

---

Note The `PickableParts` property determines if the `ColorBar` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container

`Figure` object | `Panel` object | `Tab` object | `TiledChartLayout` object

Parent container, specified as a `Figure` object, `Panel` object, `Tab` object, or a `TiledChartLayout` object.

The `ColorBar` object must have the same parent as the associated axes. If you change the parent of the associated axes, then the `ColorBar` object automatically updates to use the same parent.

`Children` — Children

empty `GraphicsPlaceholder` array

The object has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'colorbar'`

This property is read-only.

Type of graphics object, returned as `'colorbar'`.

Use this property to find objects in a hierarchy. For example, you can use the `findobj` function to find objects that have a specific `Type` property value.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[ ]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`colorbar`

Topics

“Access Property Values”
“Graphics Object Properties”

Introduced in R2014b

colorcube

Colorcube colormap array

Syntax

```
c = colorcube
c = colorcube(m)
```

Description

`c = colorcube` returns the colorcube colormap as a three-column array with the same number of rows as the colormap for the current figure. If no figure exists, then the number of rows is equal to the default length of 256. Each row in the array contains the red, green, and blue intensities for a specific color. The intensities are in the range [0,1], and the color scheme looks like this image.

`c = colorcube(m)` returns the colormap with `m` colors.

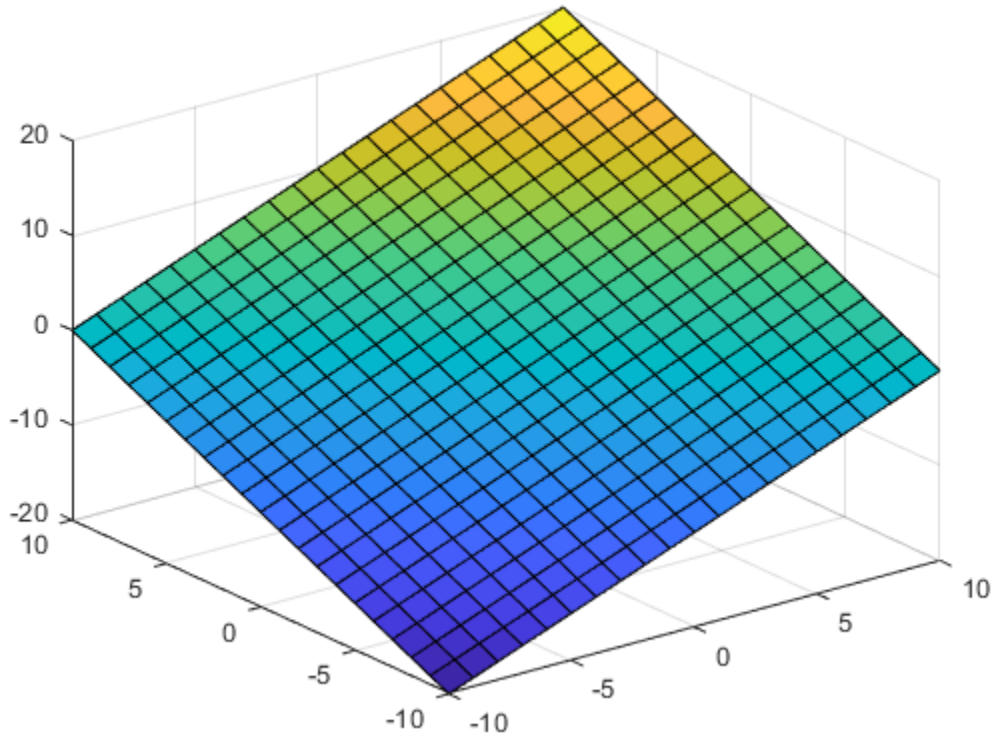
Examples

Use a Section of the Colormap

After getting the colormap array, you can isolate a section of it to use as the color scheme.

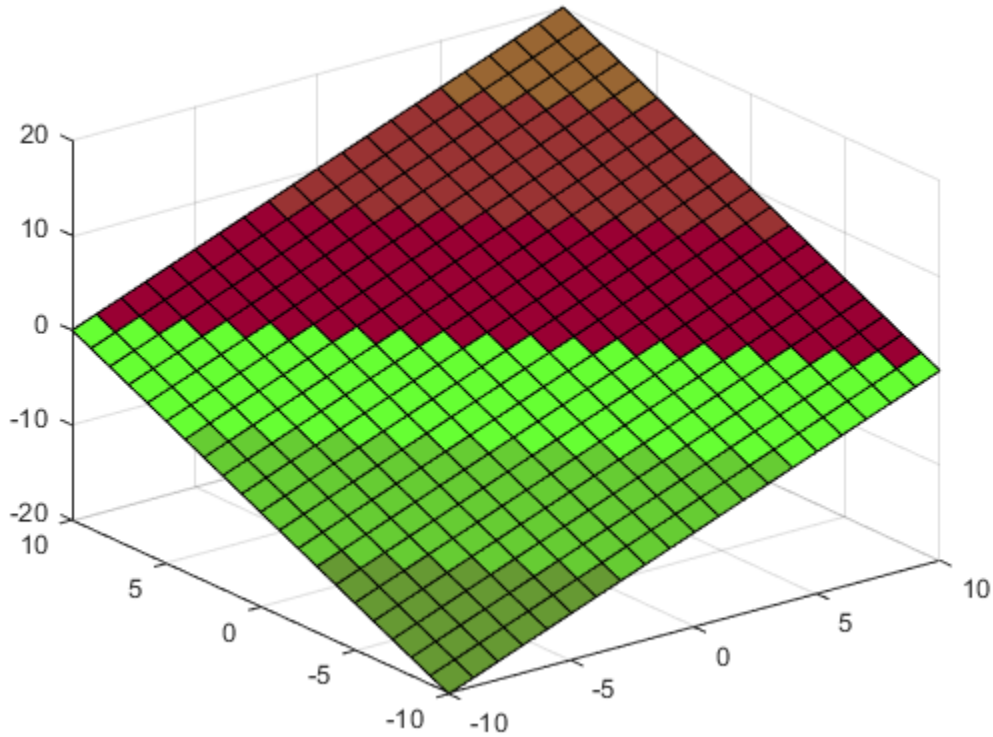
Create a surface plot of a plane.

```
[X,Y] = meshgrid(-10:1:10);
Z = X + Y;
surf(X,Y,Z);
```



Get the colorcube colormap array, and isolate the red section (entries 39 to 44). Then use this section to color the whole surface.

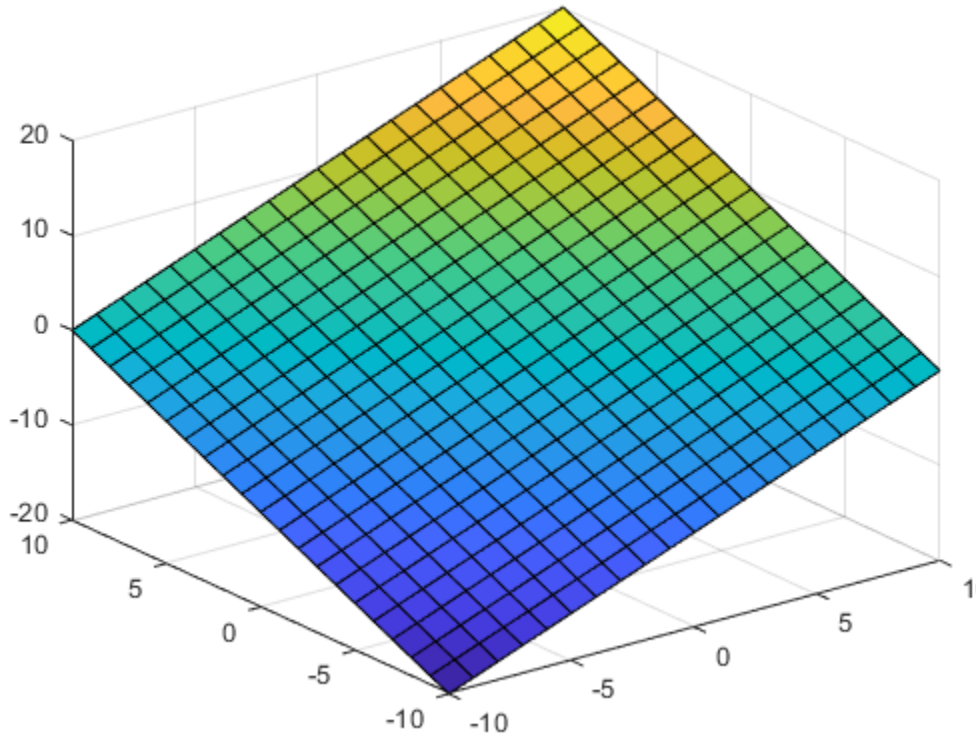
```
c = colorcube;
c = c(39:44,:);
colormap(c);
```



Color a Plane with the Colorcube Colormap

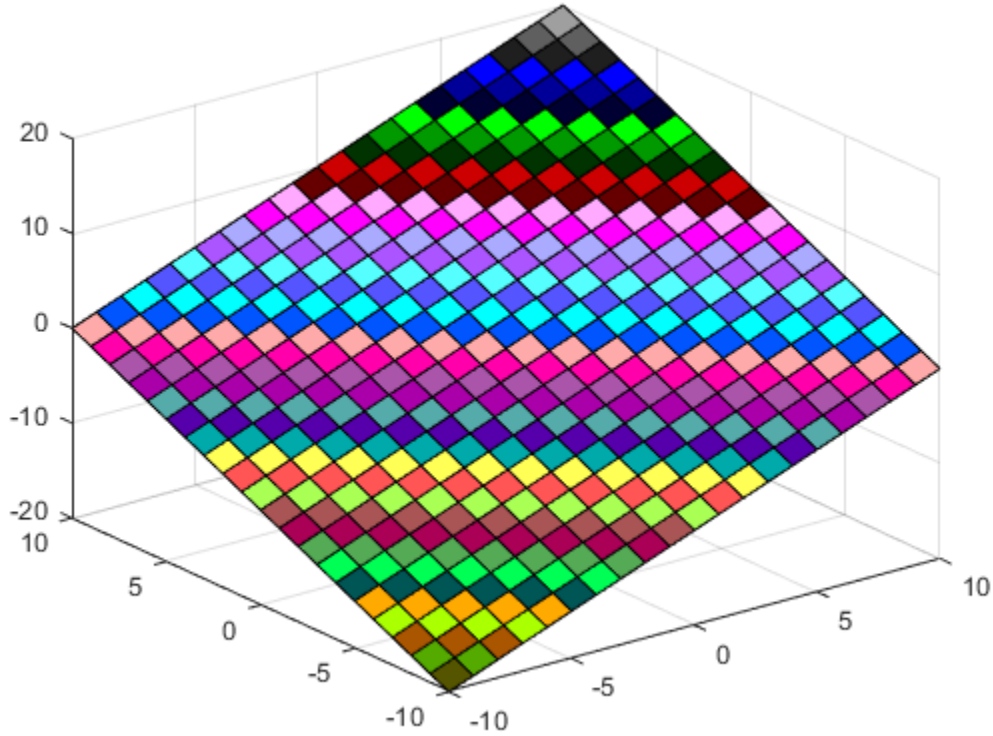
Create a surface plot of a plane.

```
[X,Y] = meshgrid(-10:1:10);
Z = X + Y;
surf(X,Y,Z);
```



Get the colorcube colormap with 75 entries. Then use it to color the plane.

```
c = colorcube(75);
colormap(c);
```



Input Arguments

`m` — Number of colors

`256` (default) | scalar integer value

Number of colors, specified as a scalar integer value. The default value of `m` is equal to the length of the colormap for the current figure. If no figure exists, the default value is 256.

Data Types: `single` | `double`

Compatibility Considerations

Colormaps Have 256 Colors by Default

Behavior changed in R2019b

Starting in R2019b, colormaps have 256 colors by default.

In R2019a and previous releases, the default size is 64. If you have code that depends on a colormap having 64 colors, specify the number of colors when you set the colormap for the figure, axes, or chart. For example, `colormap(colorcube(64))` sets the figure's colormap to the 64-color `colorcube` colormap.

Alternatively, you can change the default colormap for all figures within your MATLAB session:

```
set(groot,'defaultFigureColormap',colorcube(64))
```

See Also

`colormap`

Topics

"Change Color Scheme Using a Colormap"

Introduced before R2006a

colordef

Set default property values to display different color schemes

Note `colordef` is not recommended.

Syntax

```
colordef white
colordef black
colordef none
colordef(fig,color_option)
h = colordef('new',color_option)
```

Description

`colordef` enables you to select either a white or black background for graphics display. It sets axis lines and labels so that they contrast with the background color.

`colordef white` sets the axis background, axis lines and labels, and the figure background to the default system colors.

`colordef black` sets the axis background color to black, the axis lines and labels to white, and the figure background color to dark gray.

`colordef none` sets the figure coloring to that used by MATLAB Version 4. The most noticeable difference is that the axis background is set to `'none'`, making the axis background and figure background colors the same. The figure background color is set to black.

`colordef(fig,color_option)` sets the color scheme of the figure identified by the handle `fig` to one of the color options `'white'`, `'black'`, or `'none'`. When you use this syntax to apply `colordef` to an existing figure, the figure must have no graphic content. If it does, you should first clear it (via `clf`) before using this form of the command.

`h = colordef('new',color_option)` returns the handle to a new figure created with the specified color options (i.e., `'white'`, `'black'`, or `'none'`). This form of the

command is useful for creating GUIs when you may want to control the default environment. The figure is created with `'visible','off'` to prevent flashing.

Tips

`colordef` affects only subsequently drawn figures, not those currently on the display. This is because `colordef` works by setting default property values (on the root or figure level). You can list the currently set default values on the root level with the statement

```
get(groot,'Default')
```

You can remove all default values using the `reset` command:

```
reset(groot)
```

See the `get` and `reset` references pages for more information.

See Also

`clf`

Introduced before R2006a

colormap

View and set current colormap

Syntax

```
colormap map
colormap(map)
colormap(target,map)

cmap = colormap
cmap = colormap(target)
```

Description

`colormap map` sets the colormap for the current figure to one of the predefined colormaps. If you set the colormap for the figure, then axes and charts in the figure use the same colormap. The new colormap is the same length (number of colors) as the current colormap. When you use this syntax, you cannot specify a custom length for the colormap. To learn more about colormaps, see “What Is a Colormap?” on page 1-1952

`colormap(map)` sets the colormap for the current figure to the colormap specified by `map`.

`colormap(target,map)` sets the colormap for the figure, axes, or chart specified by `target`, instead of for the current figure.

`cmap = colormap` returns the colormap for the current figure as a three-column matrix of RGB triplets.

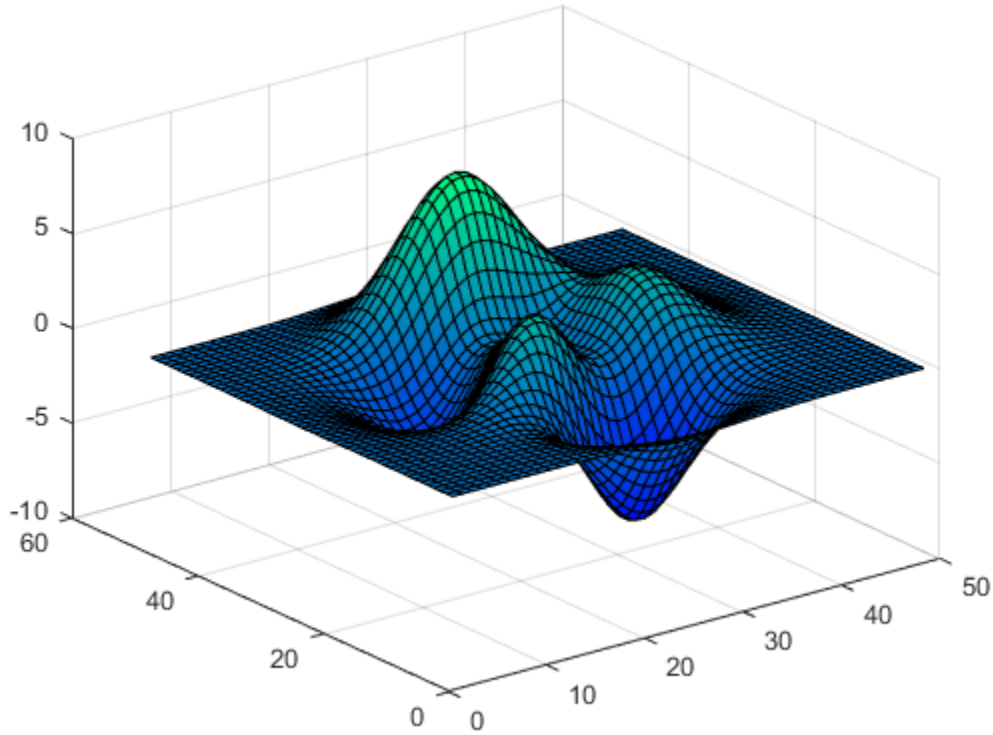
`cmap = colormap(target)` returns the colormap for the figure, axes, or chart specified by `target`.

Examples

Change Colormap for Figure

Create a surface plot and set the colormap to `winter`.

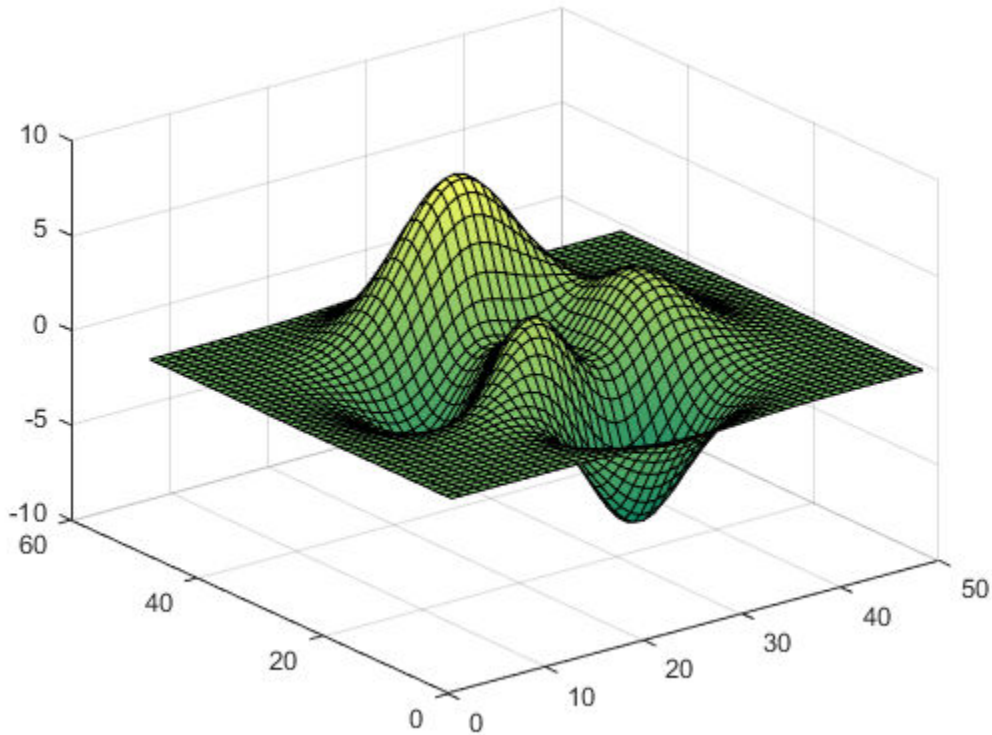
```
surf(peaks)
colormap winter
```



Set Colormap Back to Default

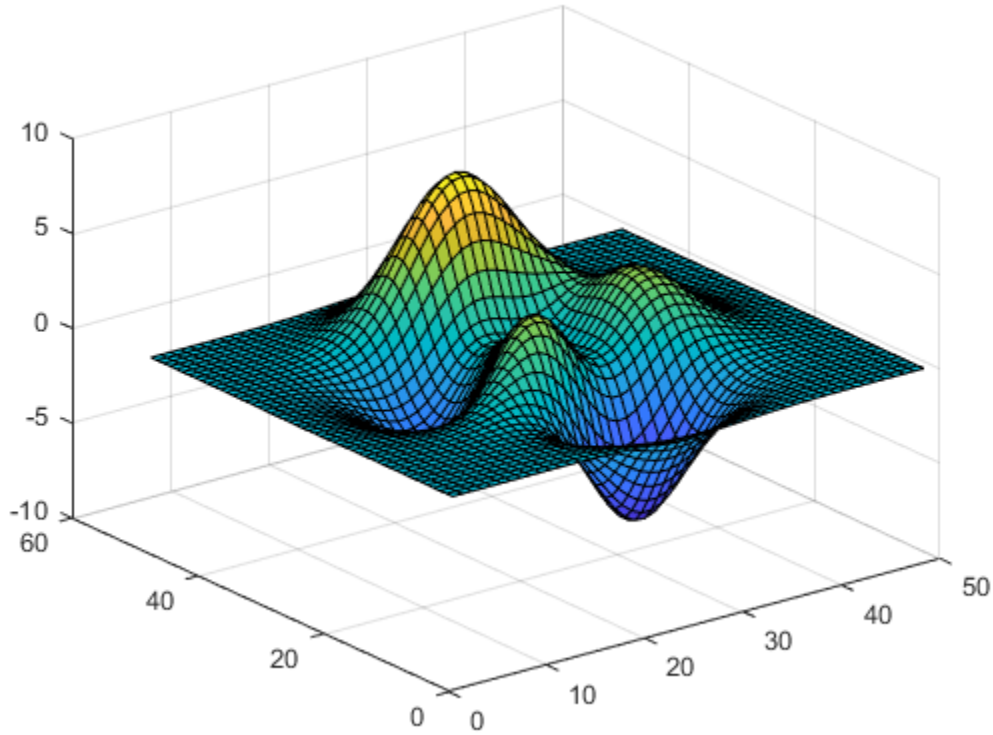
First, change the colormap for the current figure to `summer`.

```
surf(peaks)
colormap summer
```



Now set the colormap back to your system's default value. If you have not specified a different default value, then the default colormap is `parula`.

```
colormap default
```

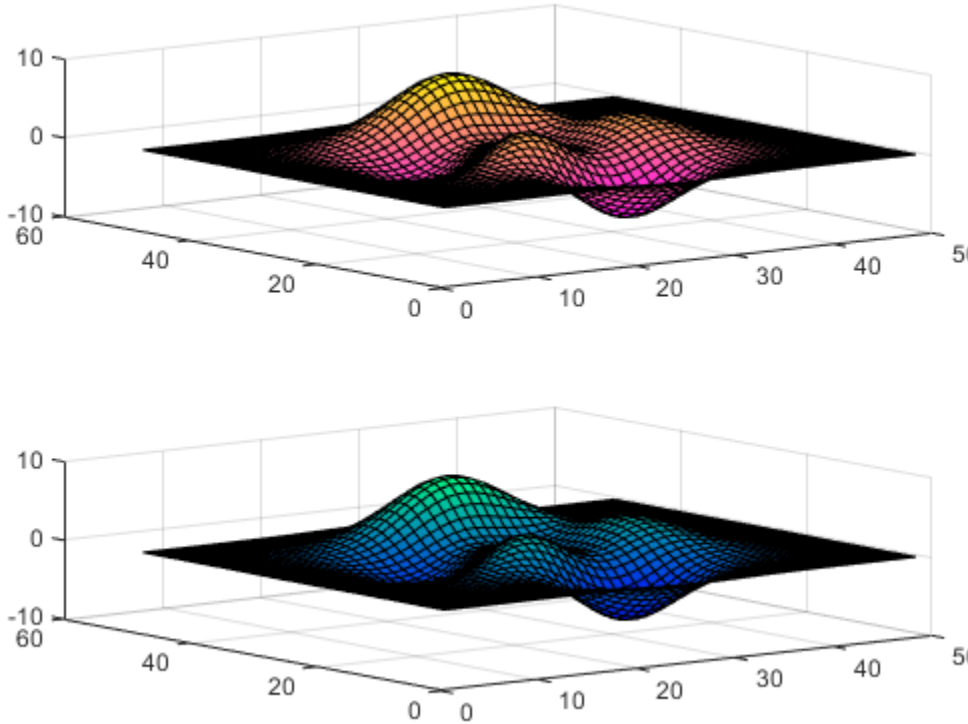


Use Different Colormaps for Each Axes in Figure

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Specify a different colormap for each axes by passing the axes object to the `colormap` function. In the upper axes, create a surface plot using the `spring` colormap. In the lower axes, create a surface plot using the `winter` colormap.

```
tiledlayout(2,1)
ax1 = nexttile;
```

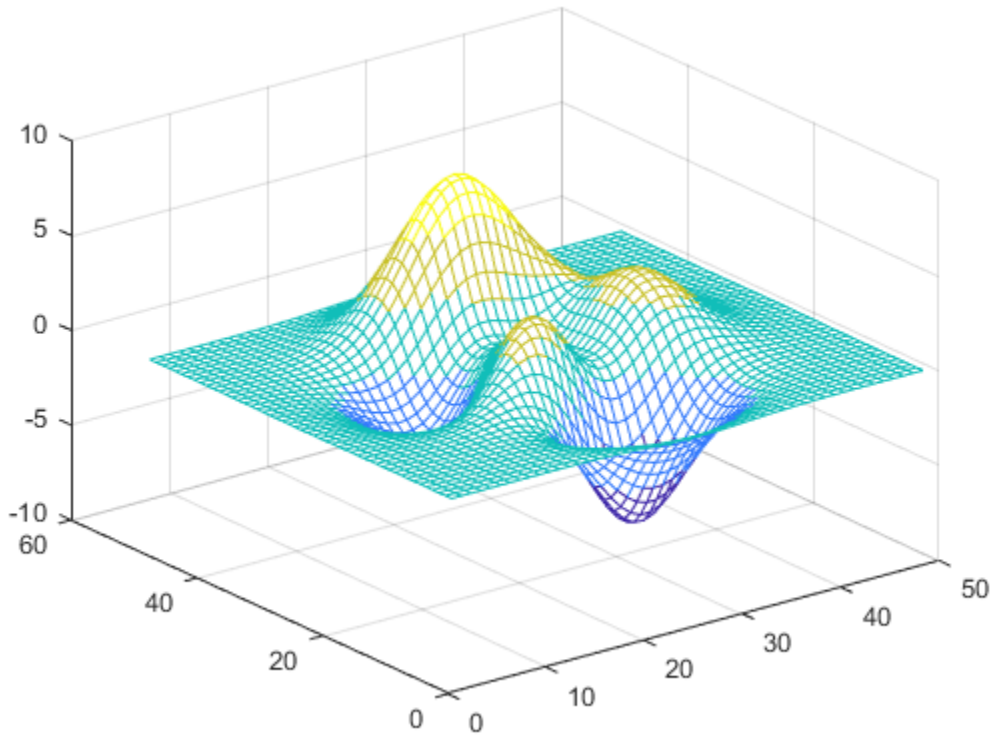
```
surf(peaks)
colormap(ax1,spring)

ax2 = nexttile;
surf(peaks)
colormap(ax2,winter)
```



Specify Number of Colors for Colormap

Specify the number of colors used in a colormap by passing an integer as an input argument to the built-in colormap. Use five colors from the parula colormap.

```
mesh(peaks)
colormap(parula(5))
```

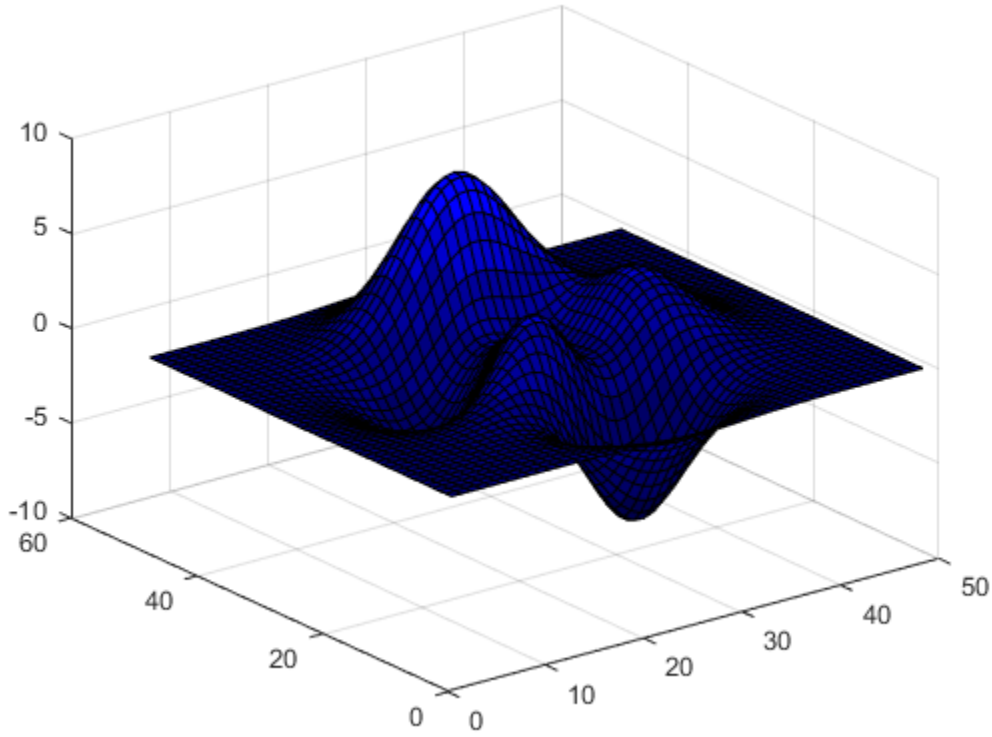


Create Custom Colormap

Create a custom colormap by defining a three-column matrix of values between 0.0 and 1.0. Each row defines a three-element RGB triplet. The first column specifies the red intensities. The second column specifies the green intensities. The third column specifies the blue intensities.

Use a colormap of blue values by setting the first two columns to zeros.

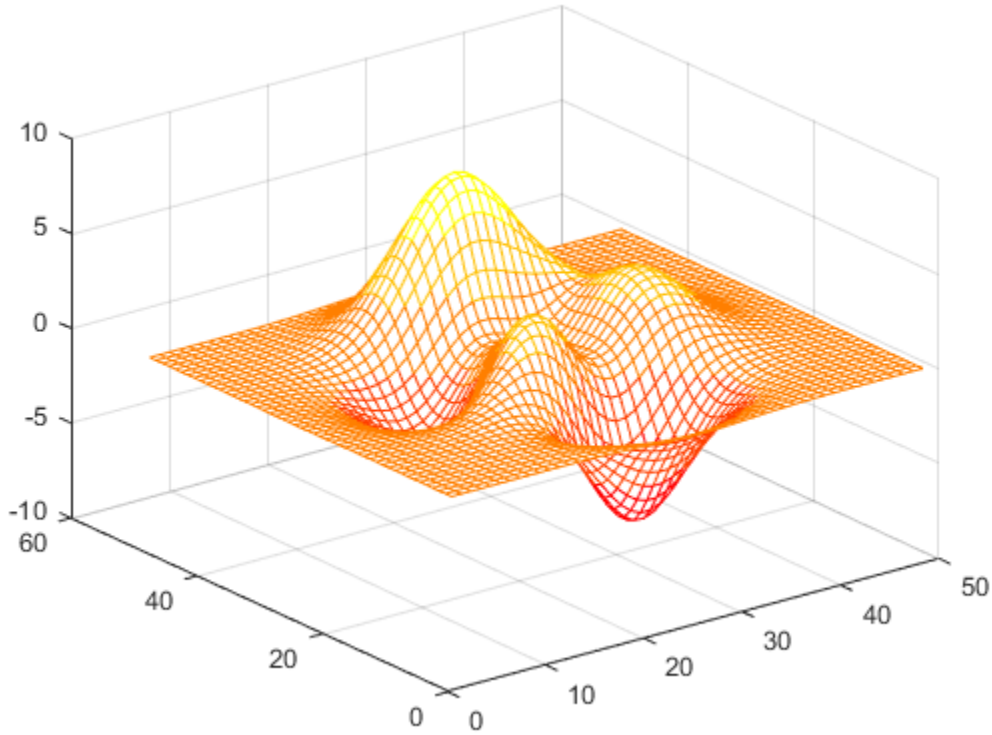
```
map = [0 0 0.3
    0 0 0.4
    0 0 0.5
    0 0 0.6
    0 0 0.8
    0 0 1.0];

surf(peaks)
colormap(map)
```



Return Colormap Values Used in Plot

Create a surface plot of the `peaks` function and specify a colormap.

```
mesh(peaks)
colormap(autumn(5))
```



Return the three-column matrix of values that define the colors used in the plot. Each row is an RGB triplet color value that specifies one color of the colormap.

```
cmap = colormap
```

```
cmap = 5×3
```

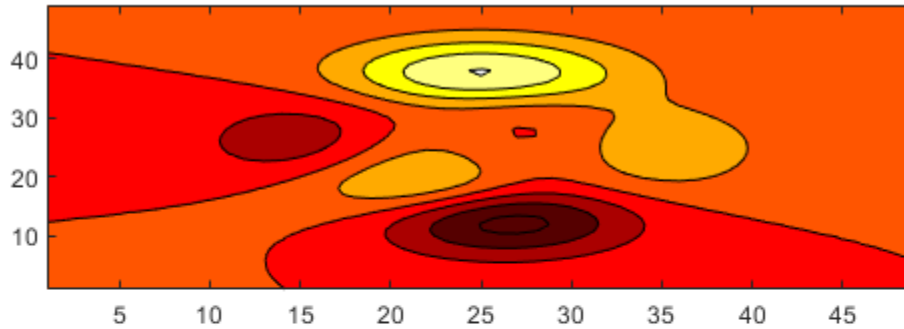
```
    1.0000         0         0
    1.0000    0.2500         0
    1.0000    0.5000         0
    1.0000    0.7500         0
    1.0000    1.0000         0
```

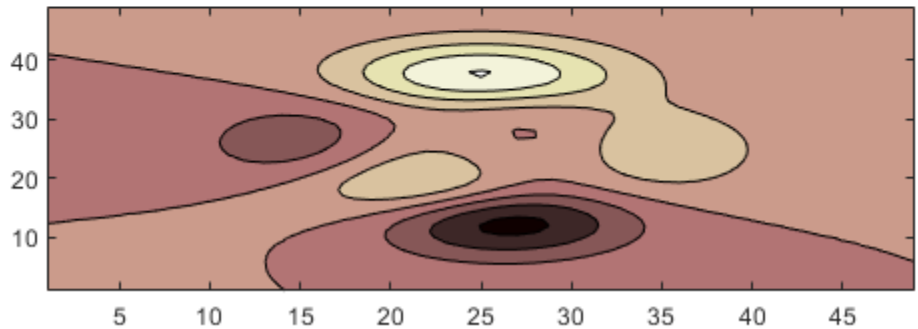
Return Colormap Values for Specific Axes

Return the colormap values for a specific axes by passing the axes object to the `colormap` function.

Create a tiling of two plots using the `tiledlayout` and `nexttile` functions, which are new functions starting in R2019b. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Then display two filled contour plots with different colormaps.

```
tiledlayout(2,1)
ax1 = nexttile;
contourf(peaks)
colormap(ax1,hot(8))

ax2 = nexttile;
contourf(peaks)
colormap(ax2,pink)
```





Return the colormap values used in the upper plot by passing `ax1` to the `colormap` function. Each row is an RGB triplet color value that specifies one color of the colormap.

```
cmap = colormap(ax1)
```

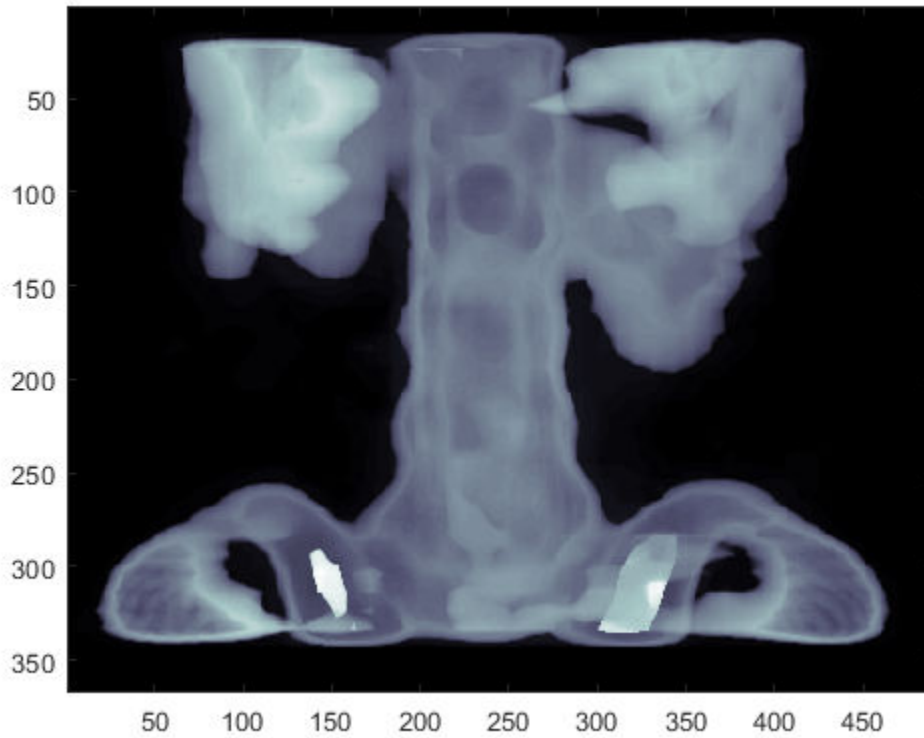
```
cmap = 8×3

    0.3333         0         0
    0.6667         0         0
    1.0000         0         0
    1.0000    0.3333         0
    1.0000    0.6667         0
    1.0000    1.0000         0
    1.0000    1.0000    0.5000
```

```
    1.0000    1.0000    1.0000
```

Change Colormap for Figure with Image

Load the `spine` data set that returns the image `X` and its associated colormap `map`. Display `X` using the `image` function and set the colormap to `map`.

```
load spine
image(X)
colormap(map)
```



Input Arguments

map — Colormap for new color scheme

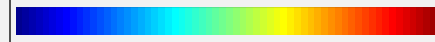
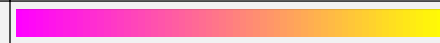
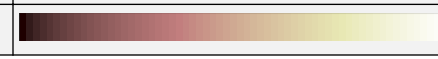
colormap name | three-column matrix of RGB triplets | `'default'`

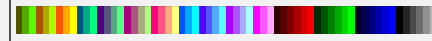
Colormap for the new color scheme, specified as a colormap name, a three-column matrix of RGB triplets, or `'default'`. A colormap name specifies a predefined colormap with the same number of colors as the current colormap. A three-column matrix of RGB triplets specifies a custom colormap. You can create the matrix yourself, or you can call one of the predefined colormap functions to create the matrix. For example, `colormap(parula(10))` sets the colormap of the current figure to a selection of 10 colors from the `parula` colormap.

A value of `'default'` sets the colormap to the default colormap for the target object.

Colormap Name

The following table lists the predefined colormaps.

Colormap Name	Color Scale
`parula`	
`jet`	
`hsv`	
`hot`	
`cool`	
`spring`	
`summer`	
`autumn`	
`winter`	
`gray`	
`bone`	
`copper`	
`pink`	

Colormap Name	Color Scale
`lines`	
`colorcube`	
`prism`	
`flag`	
`white`	

Three-Column Matrix

To create a custom colormap, specify `map` as a three-column matrix of RGB triplets where each row defines one color. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`. For example, this matrix defines a colormap containing five colors.

```
map = [0.2 0.1 0.5
    0.1 0.5 0.8
    0.2 0.7 0.6
    0.8 0.7 0.3
    0.9 1 0];
```

This table lists the RGB triplet values for common colors.

Color	RGB Triplet
yellow	`[1 1 0]`
magenta	`[1 0 1]`
cyan	`[0 1 1]`
red	`[1 0 0]`
green	`[0 1 0]`
blue	`[0 0 1]`
white	`[1 1 1]`
black	`[0 0 0]`

Data Types: `char` | `double`

target — Target
`Figure` object | `Axes` object | `PolarAxes` object | `GeographicAxes` object | graphics object

Target, specified as one of these values:

- `Figure` object. The figure colormap affects plots for all axes within the figure.
- `Axes` object, `PolarAxes` object, or `GeographicAxes` object. You can define a unique colormap for the different axes within a figure.
- Graphics object that has a `Colormap` property. For example, you can change or query the colormap for a `HeatmapChart` object.

Output Arguments

cmap — Colormap values
three-column matrix of RGB triplets

Colormap values, returned as a three-column matrix of RGB triplets. Each row of the matrix defines one RGB triplet that specifies one color of the colormap. The values are in the range [0, 1].

More About

What Is a Colormap?

A colormap is matrix of values between `0` and `1` that define the colors for graphics objects such as surface, image, and patch objects. MATLAB draws the objects by mapping data values to colors in the colormap.

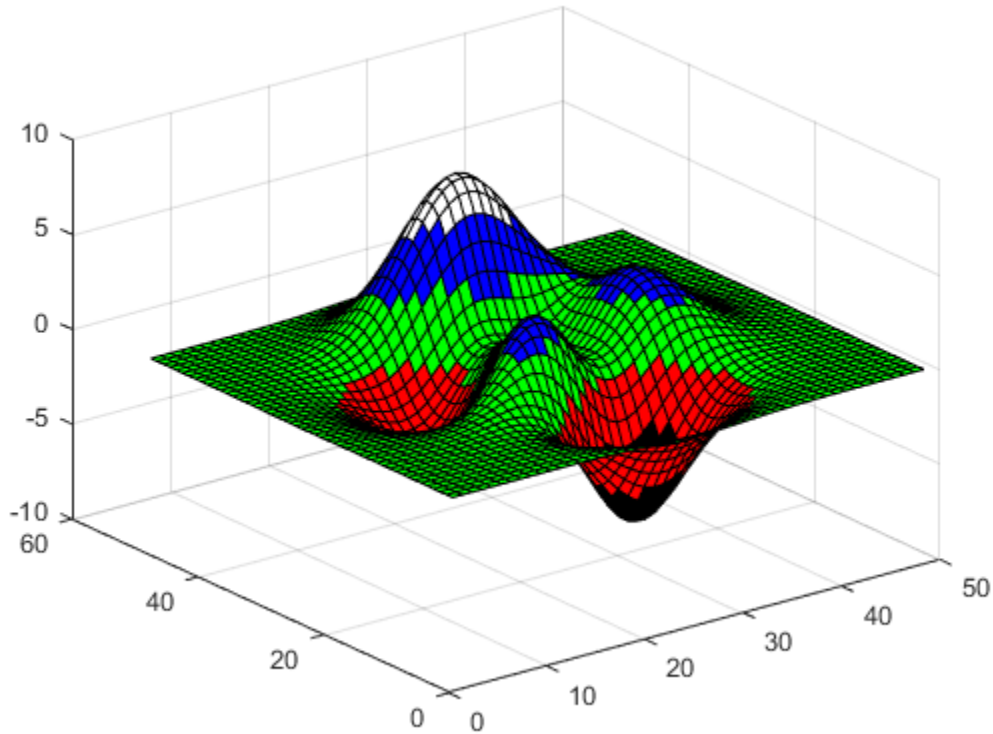
Colormaps can be any length, but must be three columns wide. Each row in the matrix defines one color using an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0, 1]. A value of `0` indicates no color and a value of `1` indicates full intensity. For example, this command creates a colormap that has five colors: black, red, green, blue, and white.

```
mymap = [0 0 0
    1 0 0
```

```
0 1 0
0 0 1
1 1 1];
```

To change the color scheme of a visualization, call the `colormap` function to change the colormap of the containing axes or figure. For example, the following commands create a surface plot and set the colormap of the figure to `mymap`.

```
surf(peaks)
colormap(mymap)
```



Compatibility Considerations

- Starting in R2018a, if you set the colormap for a figure, then axes and charts in the figure use the same colormap. Previously, any axes or chart that you set the colormap for explicitly were unaffected when you set the figure colormap. If you want an `Axes` object to use a different colormap than the figure, then set the axes colormap after setting the figure colormap.
- Starting in R2014b, the default colormap is `parula`. In previous releases, the default colormap was `jet`.

Tips

- To control the limits of the colormap, and how those limits relate to the range of your data, use the `caxis` function.

See Also

`caxis` | `colorbar` | `hsv2rgb` | `ind2rgb` | `rgbplot`

Topics

"Control Colormap Limits"

Introduced before R2006a

Colormap Editor

Open colormap editor

Description

The Colormap Editor allows you to customize the colormap of the current figure. Dragging a color marker along the displayed colormap changes the position of a color transition. Double-clicking a marker allows you to change the color at that position. MATLAB applies all your changes to the colormap of the current figure.

Open the Colormap Editor

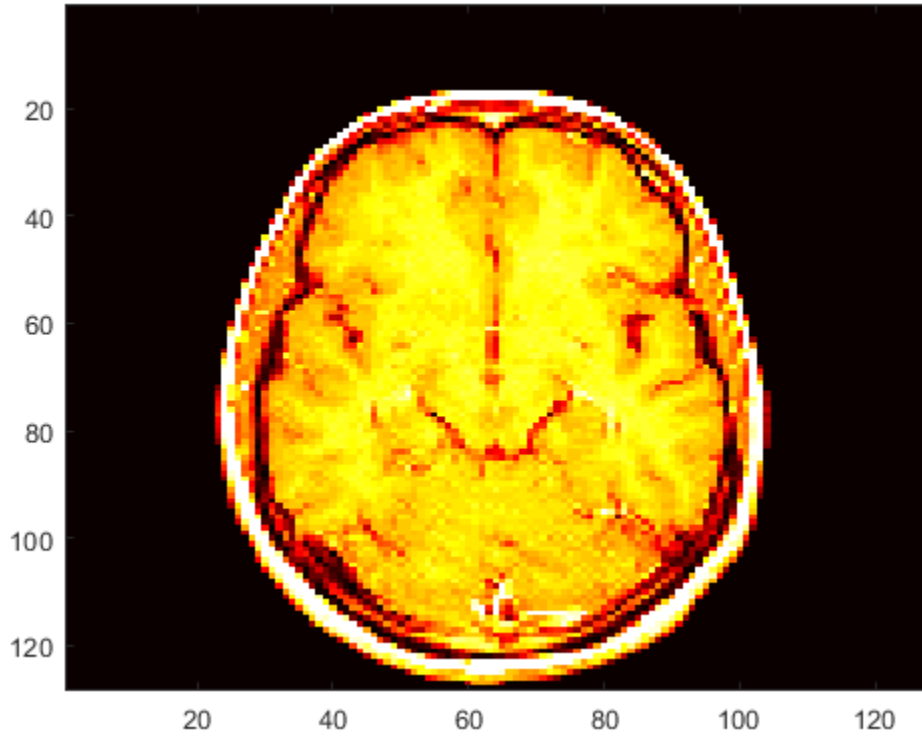
MATLAB command prompt: Enter `colormapeditor`.

Examples

Edit the Hot Colormap to Improve Image Detail

Load the `mri` dataset, and display one of image slices using the `hot` colormap.

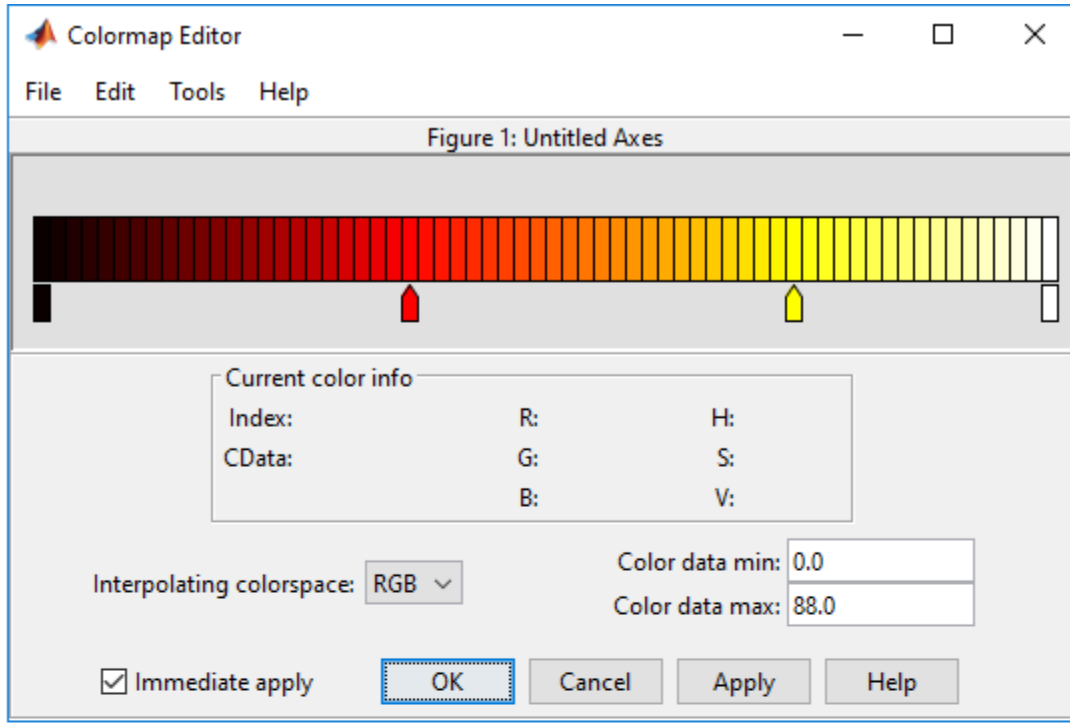
```
load mri
imagesc(D(:,:,1,11))
colormap hot
```

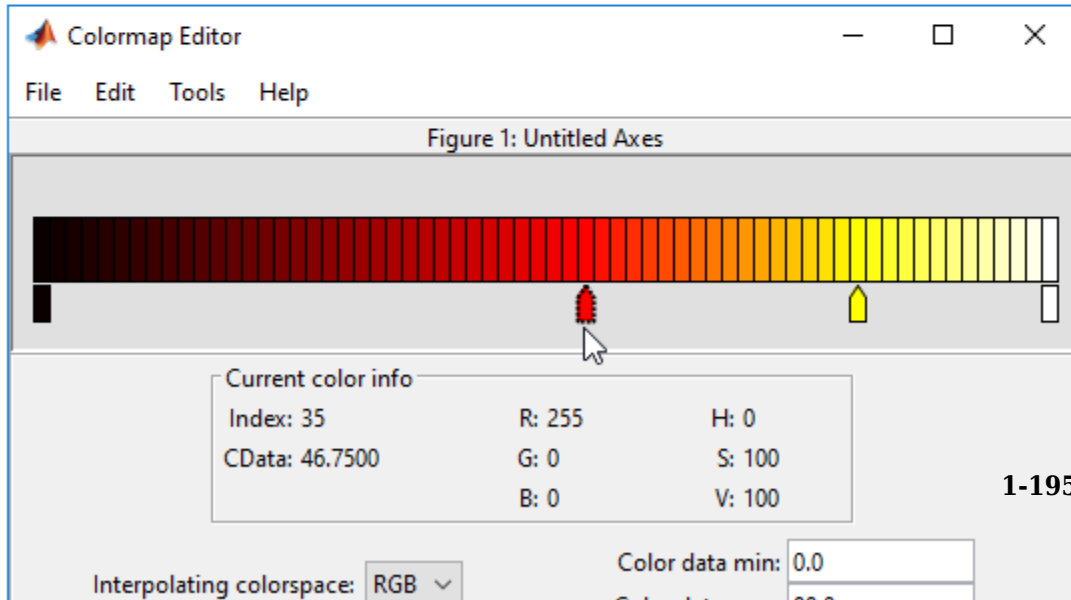


Open the Colormap Editor.

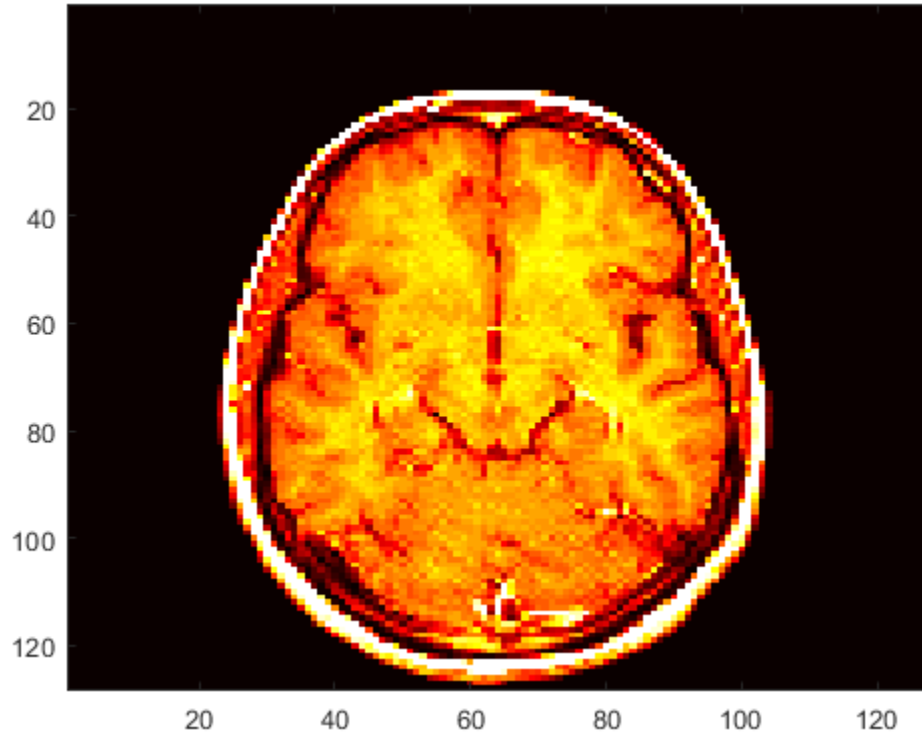
```
colormapeditor
```

Move the yellow marker to index 52, and move the red marker to index 35.


Colormap Editor
File
Edit
Tools
Help
Figure 1: Untitled Axes
Current color info
Index:
R:
H:
CData:
G:
S:
B:
V:
Interpolating colorspace:
RGB
Color data min:
0.0
Color data max:
88.0
Immediate apply
OK
Cancel
Apply
Help


Colormap Editor
File
Edit
Tools
Help
Figure 1: Untitled Axes
Current color info
Index: 35
R: 255
H: 0
CData: 46.7500
G: 0
S: 100
B: 0
V: 100
Interpolating colorspace:
RGB
Color data min:
0.0

The new colormap provides more detail near the center of the image.



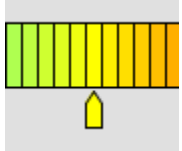
Save the colormap in the variable `c`. You can use `c` to apply the new colormap to different plots.

```
ax = gca;
c = colormap(ax);
```

Parameters

`Color markers` — Color markers
color markers placed along the displayed colormap

Color markers allow you to change the color and length of transitions in the colormap.



This table describes the adjustments you can make.

Adjustment	Adjustment Instructions
Change the colormap	Select **Tools** > **Standard Colormaps**
Add a color marker	Click below the corresponding cell in the displayed colormap.
Select a marker	Left-click the marker.
Select multiple markers	Adjacent markers: left-click the first marker. Then press **Shift** before clicking the last marker. Nonadjacent markers: left-click first marker. Then hold the **Ctrl** key as you click the other markers.
Move a marker	Select and drag the marker. You can also press the left and right arrow keys instead of dragging the marker.
Move multiple markers	Select multiple markers, and use the left and right arrow keys to move markers as a group. The movement stops when one of the selected markers encounters unselected marker or an end marker.
Delete a marker	Select the marker, and perform one of the following actions: • Press the **Delete** key • Select **Edit** > **Delete** • Press **Ctrl+X**
Delete multiple markers	Select the markers, and then press the **Delete** key, or select **Delete** from the **Edit** menu, or type **Ctrl +X**.
Change marker color	Double-click the marker.

`Interpolating colorspace` — Interpolating color space
RGB (default) | HSV

Interpolating color space, specified as one of these values:

- RGB — MATLAB calculates values in the new colormap by linearly interpolating the red, green, and blue components of color.
- HSV — MATLAB calculates values in the new colormap by linearly interpolating the hue, saturation, and value coordinates. Hue corresponds to the values on a color wheel that transition from red to orange, yellow, green, cyan, blue, magenta, and finally back to red. Hue interpolation is calculated as the shortest distance between adjacent colors on the color wheel.

When you change the **Interpolating colorspace**, the Colormap Editor preserves the number, color, and location of markers, which can cause the colormap to change.

`Color data min` — Colormap lower limit
number

Colormap lower limit, specified as a number. All values in your plot data that are less than or equal to the value of **Color data min** map to the first color in the colormap. All values between **Color data min** and **Color data max** map linearly to the intermediate colors of the colormap.

When you change **Color data min** and **Color data max**, those changes are reflected in the `CLim` property of the plotting axes.

`Color data max` — Colormap upper limit
number

Colormap upper limit, specified as a number. All values in your plot data that are greater than or equal to the value of **Color data max** map to the last color in the colormap. All values between **Color data min** and **Color data max** map linearly to the intermediate colors of the colormap.

When you change **Color data min** and **Color data max**, those changes are reflected in the `CLim` property of the plotting axes.

Tips

- To save a colormap after making changes, call the `colormap` function. Specify the current axes as the input argument, and specify an output argument to store the colormap.

  ```
  ax = gca;
  mymap = colormap(ax);
  ```

  Use the `save` function to save the colormap as a MAT-file for use in future sessions.

  ```
  save('MyColormap','mymap')
  ```

- When you move the cursor over the displayed colormap or any of its markers, the following information displays in the **Current color info** section:

  - Index — The row in the colormap array that corresponds to the current color.
  - CData — The corresponding value in the `CData` property of the graphics object that is using the colormap.
  - RGB — The RGB values of the current color.
  - HSV — The HSV values of the current color.

See Also

Functions
`caxis` | `colormap`

Introduced before R2006a

colororder

Set color order for visualizing multiple data series

Syntax

```
colororder(newcolors)
colororder(target,newcolors)
```

```
C = colororder
C = colororder(target)
```

Description

`colororder(newcolors)` sets the color order on page 1-1977 for the current figure. If a figure does not exist, MATLAB creates a figure and sets the color order for that figure. When you set the color order for a figure, you set the color order for all the axes within that figure.

`colororder(target,newcolors)` sets the color order for the target axes or figure instead of the current figure.

`C = colororder` returns the color order matrix for the current figure.

`C = colororder(target)` returns the color order matrix for the target figure or axes.

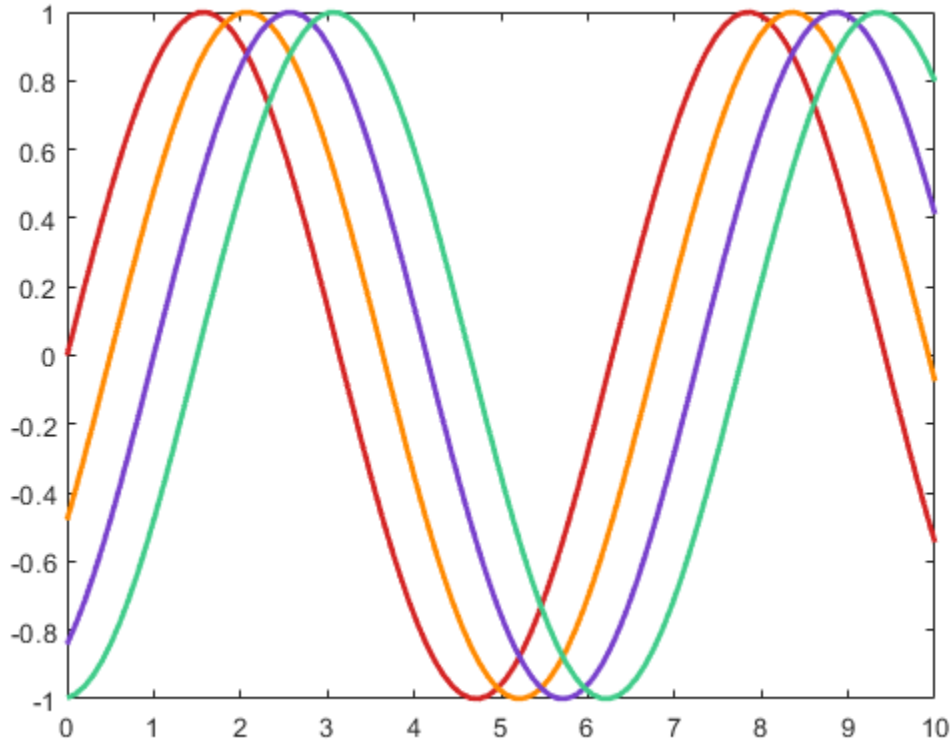
Examples

Set Colors Before Plotting

Set the color order for the figure to four colors. Define an *x*-coordinate vector and four *y*-coordinate vectors. Then plot each set of coordinates.

```
newcolors = [0.83 0.14 0.14
             1.00 0.54 0.00
```

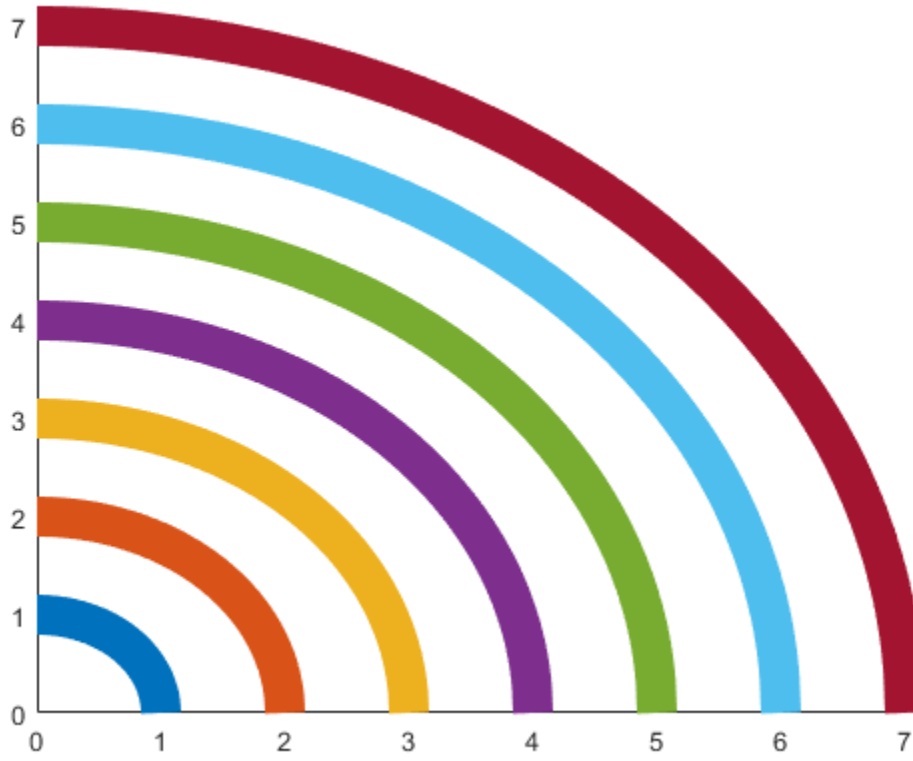
```
             0.47 0.25 0.80
             0.25 0.80 0.54];

colororder(newcolors)

% Define coordinates
x = linspace(0,10);
y1 = sin(x);
y2 = sin(x-0.5);
y3 = sin(x-1);
y4 = sin(x-1.5);

% Plot coordinates
plot(x,y1,'LineWidth',2)
hold on
plot(x,y2,'LineWidth',2)
plot(x,y3,'LineWidth',2)
plot(x,y4,'LineWidth',2)
hold off
```



Set Colors After Plotting

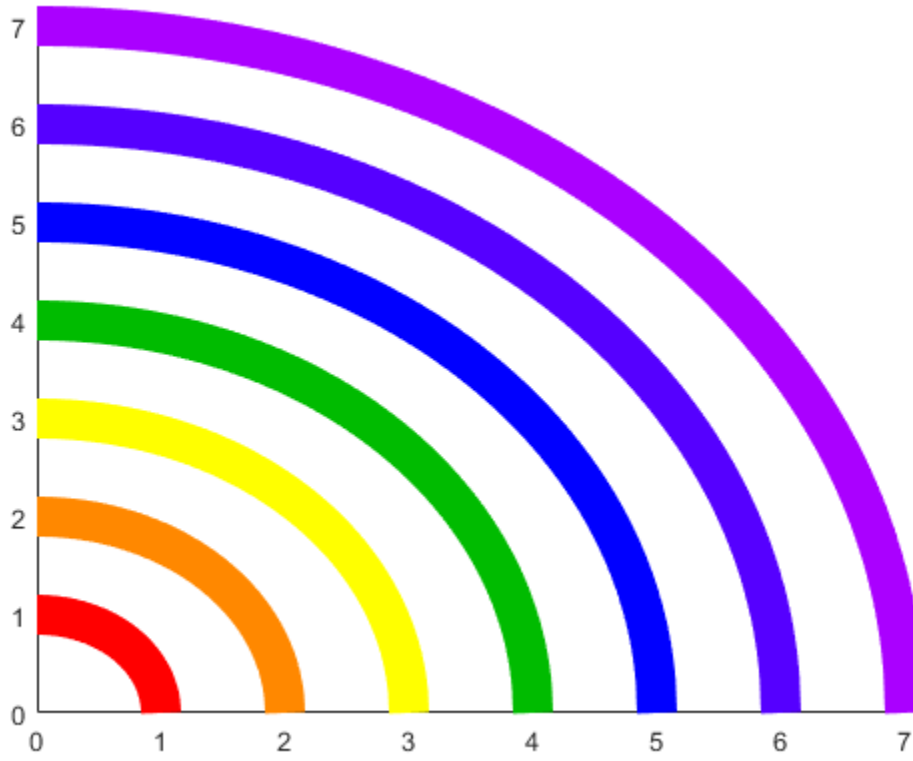
Plot seven concentric quarter circles.

```
hold on
for r=1:7
    x = linspace(0,r,500);
    y = sqrt(r.^2-x.^2);
    plot(x,y,'LineWidth',15)
end
```



Change the color order to seven hexadecimal color codes.

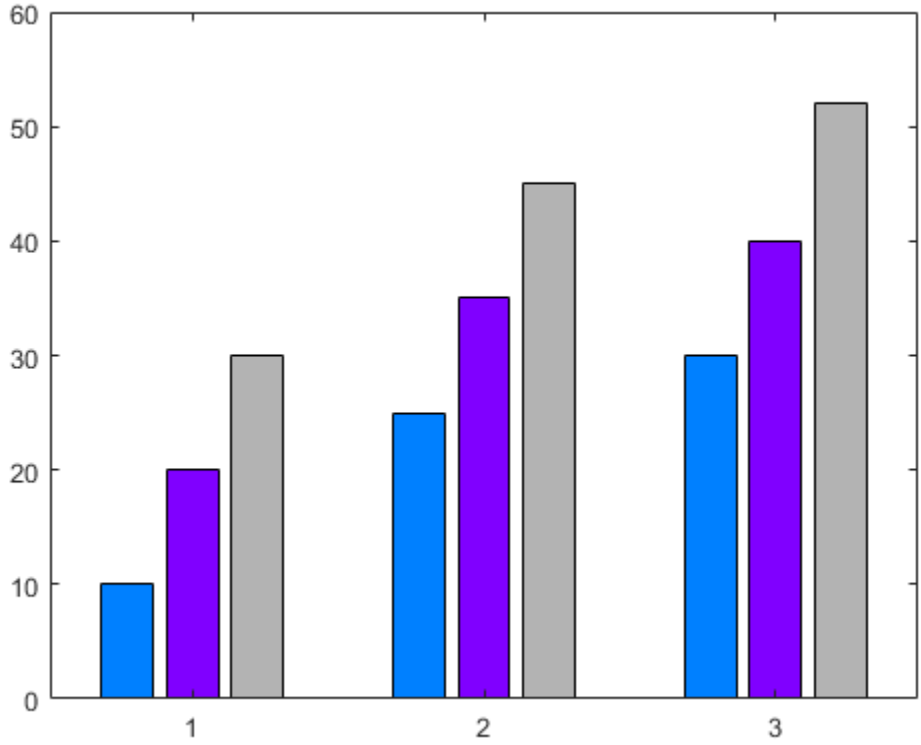
```
newcolors = {'#F00','#F80','#FF0','#0B0','#00F','#50F','#A0F'};
colororder(newcolors)
```



Set Colors in a Bar Graph

Display three series of bars. Then set the color order to blue, purple, and gray.

```
bar([10 20 30; 25 35 45; 30 40 52])
newcolors = [0 0.5 1; 0.5 0 1; 0.7 0.7 0.7];
colororder(newcolors)
```



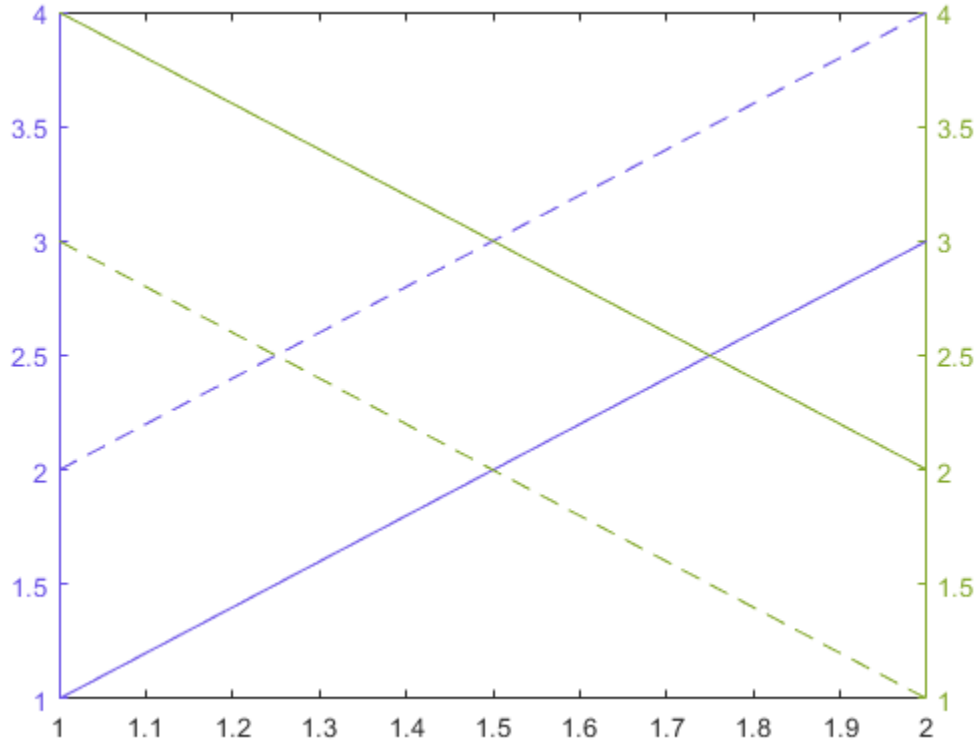
Set Colors for Both Sides of Plot with Two y-Axes

Setting the color order for the figure before calling `yyaxis` sets the color for each *y*-axis. The left side uses the first color, and the right side uses the second color. If you specify more than two colors, the additonal colors are not used by either side.

Define `newcolors` as a matrix containing two RGB triplets. Set the color order for the figure, and plot two lines against the left side. Then plot two lines against the right side.

```
newcolors = [0.40 0.30 0.90; 0.50 0.65 0.15];
colororder(newcolors)
```

```
% Left side
yyaxis left
plot([1 2; 3 4])

% Right side
yyaxis right
plot([4 3; 2 1])
```

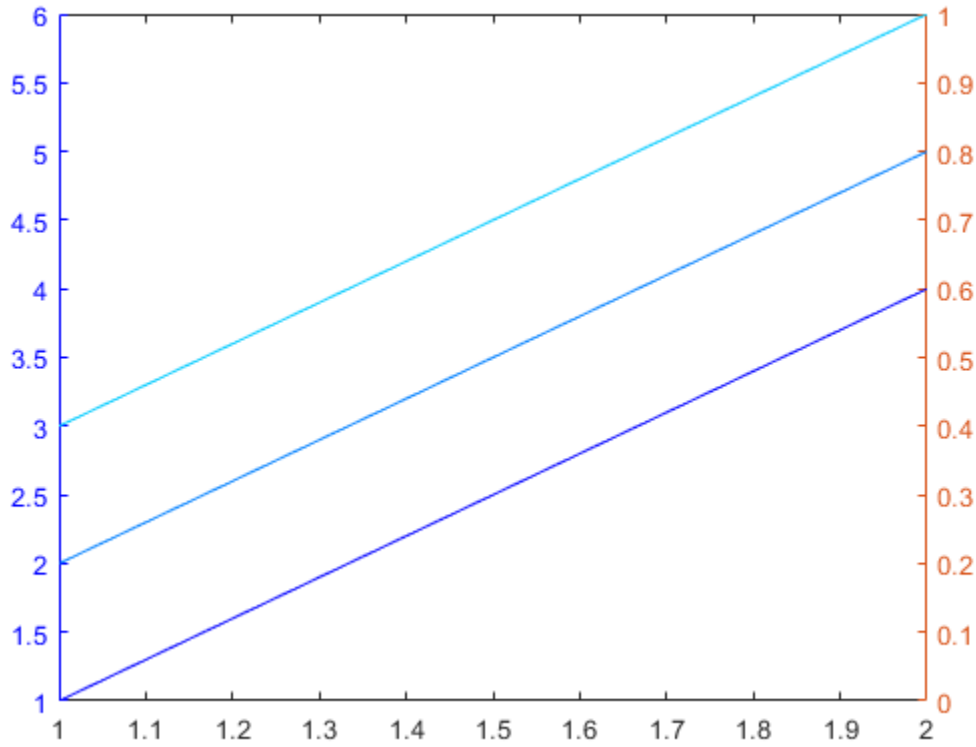


Set Colors for Active Side of Plot with Two y-Axes

Setting the color order for the figure after calling `yyaxis` sets the color for the active side.

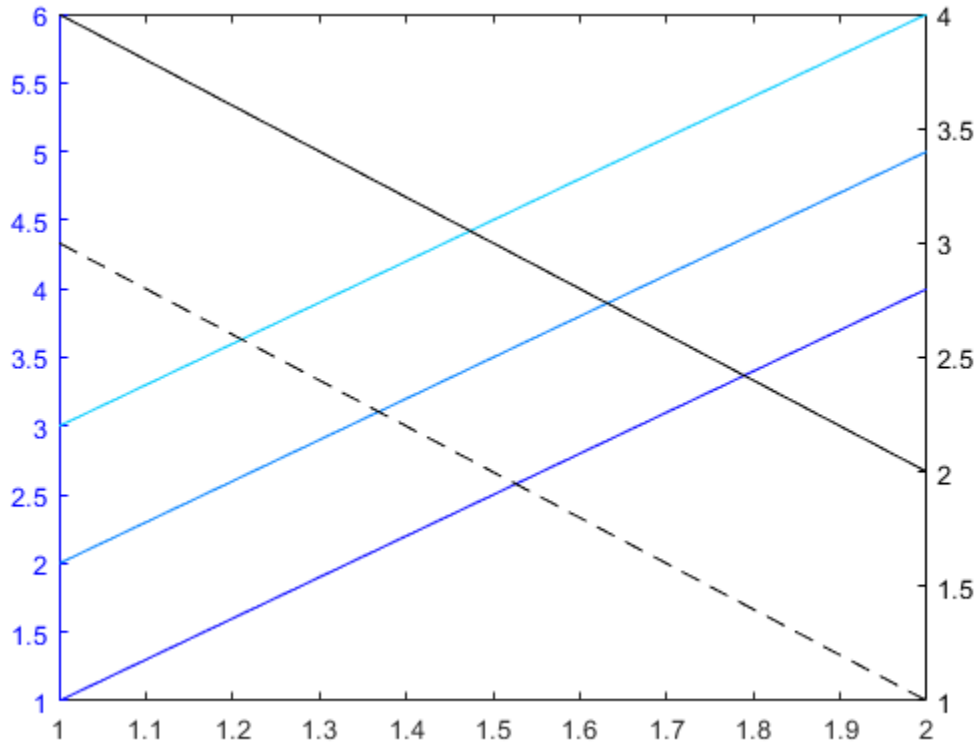
Activate the left *y*-axis and plot three lines. Set the line style order to one solid line and change the *y*-axis color to blue. Then set the color order to three shades of blue.

```
% Left side
yyaxis left
plot([1 2 3; 4 5 6])
ax = gca;
ax.LineStyleOrder = '-';
ax.YColor = 'blue';
leftcolors = [0 0 1; 0 0.50 1; 0 0.80 1];
colororder(leftcolors)
```



Activate the right *y*-axis and plot two lines. Change the *y*-axis color to black. Then set the color order to black.

```
% Right side
yyaxis right
plot([4 3; 2 1])
ax.YColor = 'black';
colororder('black')
```



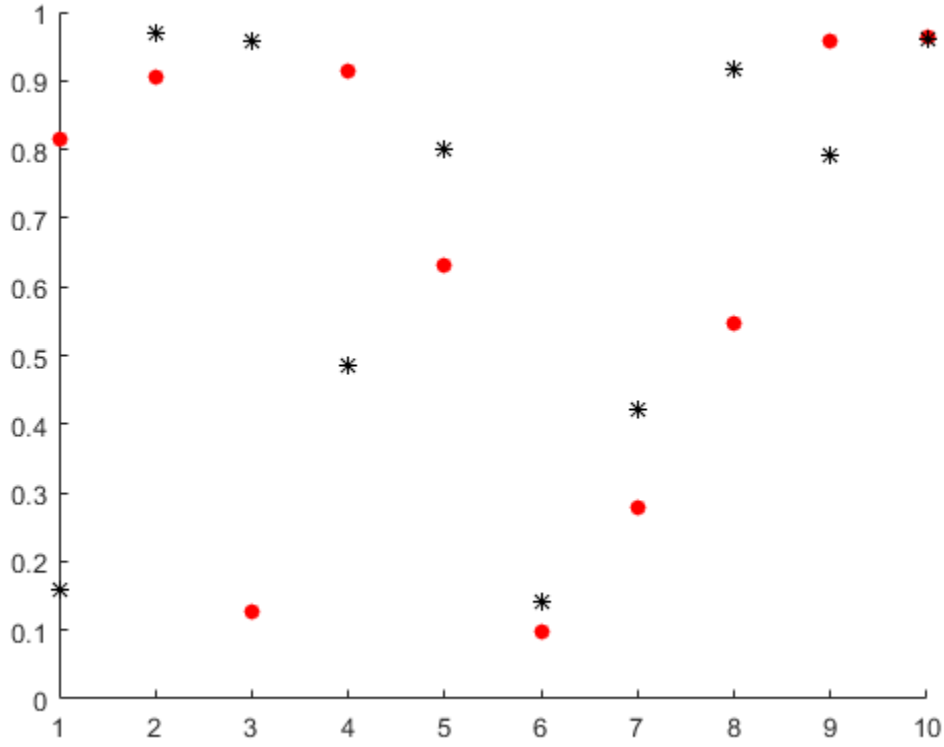
Exclude Data Series from Color Order

When you call a plotting function with a color argument, the plotting function uses that color instead of the next color in the color order.

Set the color order of the figure to red, magenta, and blue. Call the `scatter` function to plot a series of scattered points. Then plot a second series of points, and specify the markers as black asterisks.

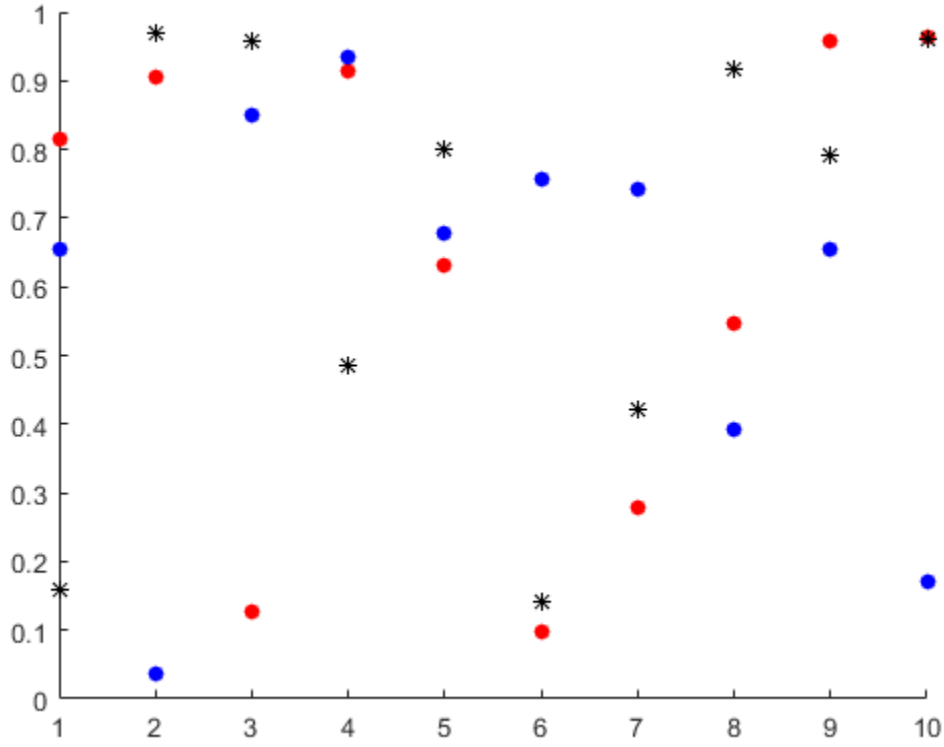
```
newcolors = {'red','magenta','blue'};
colororder(newcolors)
```

```
scatter(1:10,rand(1,10),'filled')
hold on
scatter(1:10,rand(1,10),'*k')
```



Plot a third series of points without specifying the marker color. Notice that this series uses the third color in the color order, which is blue.

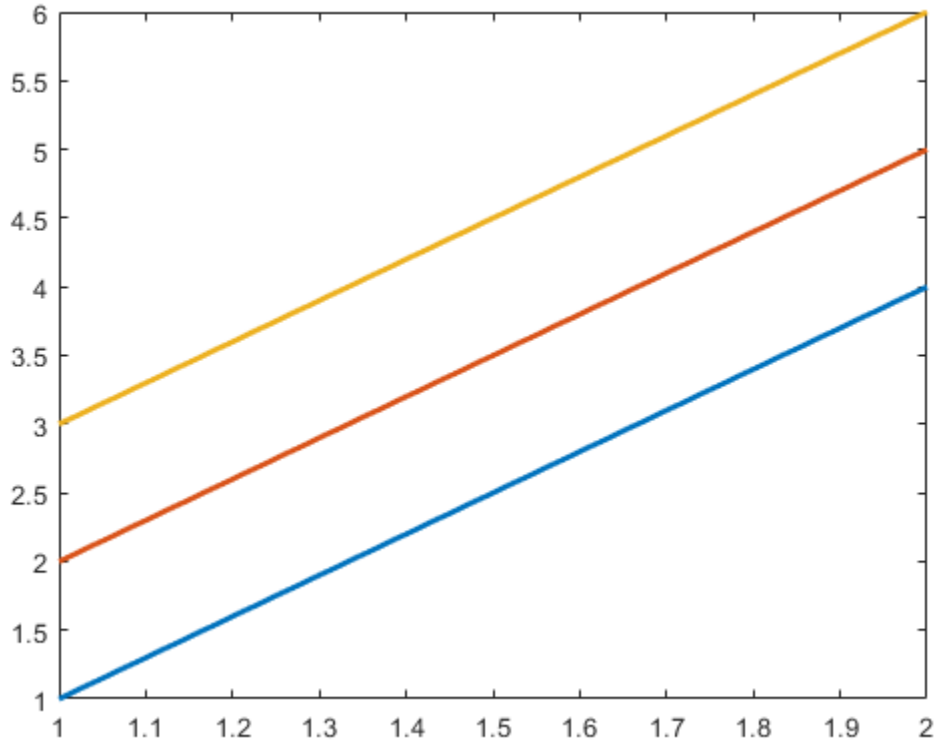
```
scatter(1:10,rand(1,10),'filled')
hold off
```



Modify Color Order for Specified Axes

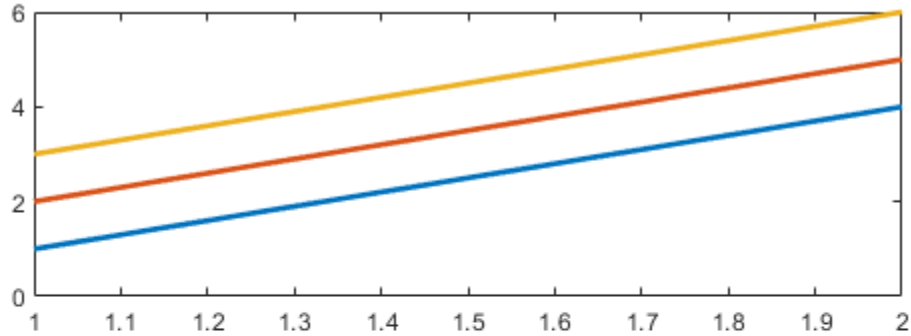
Create a tiled chart layout and plot three lines in the first tile.

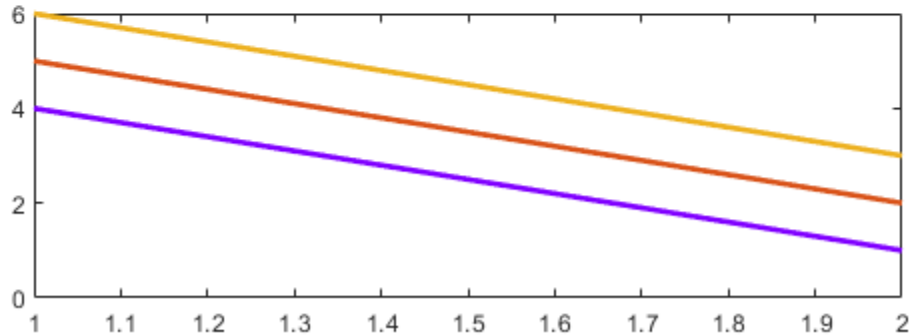
```
tiledlayout('flow')
nexttile
plot([1 2 3; 4 5 6],'LineWidth',2)
```



Call the `nexttile` function with a return argument to get the axes object for the second tile. Plot three lines in the second tile. Then get the color order matrix for the axes and return the output in `C`. Change the first color in `C` to purple, and set the axes color order to the modified `C` matrix.

```
ax = nexttile;
plot(ax,[4 5 6; 1 2 3],'LineWidth',2)
C = colororder(ax);
C(1,:) = [0.5 0 1];
colororder(ax,C)
```





Input Arguments

`newcolors` — New colors

matrix of RGB triplets | array of color names | `'default'`

New colors, specified as a matrix of RGB triplets, an array of color names, or `'default'`.

Matrix of RGB Triplets

Specify an m-by-3 matrix, where each row is an RGB triplet. An RGB triplet is a three-element vector containing the intensities of the red, green, and blue components of a

color. The intensities must be in the range `[0,1]`. For example, this matrix defines the new colors as blue, dark green, and orange:

```
newcolors = [1.0 0.0 0.0
             0.0 0.4 0.0
             1.0 0.5 0.0];
```

Array of Color Names or Hexadecimal Color Codes

Specify any combination of color names, short names, or hexadecimal color codes.

- To specify one color, set `newcolors` to a character vector or a string scalar. For example, `newcolors = 'red'` specifies red as the only color in the color order.
- To specify multiple colors, set `newcolors` to a cell array of character vectors or a string array. For example, `newcolors = {'red','green','blue'}` specifies red, green, and blue as the colors.

A hexadecimal color code starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

This table lists the short names and color names, and the equivalent hexadecimal color codes.

Color Name	Short Name	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`'#FF0000'`	
`'green'`	`'g'`	`'#00FF00'`	
`'blue'`	`'b'`	`'#0000FF'`	
`'cyan'`	`'c'`	`'#00FFFF'`	
`'magenta'`	`'m'`	`'#FF00FF'`	
`'yellow'`	`'y'`	`'#FFFF00'`	
`'black'`	`'k'`	`'#000000'`	
`'white'`	`'w'`	`'#FFFFFF'`	

Default Colors

Specify `'default'` to set the color order to the seven default colors. This option is useful for resetting the color order after you temporarily change it. Here are the RGB triplets and hexadecimal color codes for the default colors.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Data Types: `single` | `double` | `char` | `cell` | `string`

`target` — Target axes or figure
`Figure` object | `Axes` object | `PolarAxes` object | `GeographicAxes` object

Target, specified as one of these values:

- A `Figure` object. The new colors affect the contents of all the axes in the figure.
- Any type of axes object: an `Axes`, `PolarAxes`, or `GeographicAxes` object. The new colors affect the contents of the specified axes only.

More About

Color Order

The color order controls the set of colors that MATLAB uses for plotting multiple data series within an axes. Graphics objects such as `Line`, `Scatter`, and `Bar` objects are assigned colors according to their order of creation.

The colors are stored as a matrix in the `ColorOrder` property of the axes. Calling the `colororder` function replaces the matrix.

Tips

- When you set the color order for a figure, the colors persist when you call a plotting function. However, if you pass an axes object to the `colororder` function, you must first call `hold on` to make the colors persist when you call a plotting function.
- If you set the `ColorOrderIndex` or `LineStyleOrderIndex` property on the axes, the new color order does not affect existing plots. The new colors take effect only after you call `hold on` and then call a plotting function.

See Also

Functions
`bar` | `hold` | `nexttile` | `plot` | `scatter` | `tiledlayout` | `yyaxis`

Properties
Axes

Introduced in R2019b

matlab.colors Settings

MATLAB syntax highlighting color settings

You can customize the colors for syntax highlighting using the `matlab.colors` settings. Access `matlab.colors` settings using the root `SettingsGroup` object returned by the `settings` function. For example, set the temporary value for the color of comments to `[173 235 255]` (light blue).

```
s = settings;
s.matlab.colors.CommentColor.TemporaryValue = [173 235 255]
```

For more information about settings, see “Access and Modify Settings”.

Settings

matlab.colors

`SyntaxErrorColor` — Color of syntax errors

`[255 0 0]` (default) | RGB triplet

Color of syntax errors, specified as a RGB triplet with values in the range [0,255].

Example: `s.matlab.colors.SyntaxErrorColor.TemporaryValue = [173 235 255]`

`KeywordColor` — Color of keywords

`[0 0 255]` (default) | RGB triplet

Color of keywords, specified as a RGB triplet with values in the range [0,255].

Example: `s.matlab.colors.KeywordColor.TemporaryValue = [173 235 255]`

`StringColor` — Color of strings

`[160 32 240]` (default) | RGB triplet

Color of strings, specified as a RGB triplet with values in the range [0,255].

Example: `s.matlab.colors.StringColor.TemporaryValue = [173 235 255]`

`SystemCommandColor` — Color of system commands
`[179 140 0]` (default) | RGB triplet

Color of system commands, specified as a RGB triplet with values in the range [0,255].

Example: `s.matlab.colors.SystemCommandColor.TemporaryValue = [173 235 255]`

`UnterminatedStringColor` — Color of unterminated strings
`[179 0 0]` (default) | RGB triplet

Color of unterminated strings, specified as an RGB triplet with values in the range [0,255].

Example: `s.matlab.colors.UnterminatedStringColor.TemporaryValue = [173 235 255]`

`CommentColor` — Color of comments
`[34 139 34]` (default) | RGB triplet

Color of comments, specified as a RGB triplet with values in the range [0,255].

Example: `s.matlab.colors.CommentColor.TemporaryValue = [173 235 255]`

`ValidationSectionColor` — Color of validation sections
`[162 82 45]` (default) | RGB triplet

Color of validation sections, specified as a RGB triplet with values in the range [0,255].

Example: `s.matlab.colors.ValidationSectionColor.TemporaryValue = [173 235 255]`

See Also

`settings`

Topics

"Access and Modify Settings"
"Change Color Settings"
"Check Syntax as You Type"

Introduced in R2018b

ColorSpec (Color Specification)

Color specification

Description

`ColorSpec` is not a function; it refers to the three ways in which you specify color for MATLAB graphics:

- RGB triplet
- Short name
- Long name

The short names and long names are character vectors that specify one of eight predefined colors. The RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color; the intensities must be in the range [0 1]. The following table lists the predefined colors and their RGB triplet equivalents.

RGB Triplet	Short Name	Long Name
`[1 1 0]`	`y`	`yellow`
`[1 0 1]`	`m`	`magenta`
`[0 1 1]`	`c`	`cyan`
`[1 0 0]`	`r`	`red`
`[0 1 0]`	`g`	`green`
`[0 0 1]`	`b`	`blue`
`[1 1 1]`	`w`	`white`
`[0 0 0]`	`k`	`black`

Examples

This code makes the *x*-axis red.

```
ax = gca;
ax.XColor = 'red';
```

This code changes the figure background color to pink:

```
fig = gcf;
fig.Color = [1 0.4 0.6];
```

Tips

The eight predefined colors and any colors you specify as RGB triplets are not part of a figure's colormap, nor are they affected by changes to the figure's colormap. They are referred to as *fixed* colors, as opposed to *colormap* colors.

In most cases, you can specify a color using either an RGB triplet or a character vector of a color name. However, in some cases, you cannot specify the color as a color name. For example, you cannot set the `CData` property of a scatter object to a color name.

See Also

`bar` | `bar3` | `colordef` | `colormap` | `fill` | `fill3` | `uisetcolor`

colperm

Sparse column permutation based on nonzero count

Syntax

```
j = colperm(S)
```

Description

`j = colperm(S)` generates a permutation vector `j` such that the columns of `S(:,j)` are ordered according to increasing count of nonzero entries. This is sometimes useful as a preordering for LU factorization; in this case use `lu(S(:,j))`.

If `S` is symmetric, then `j = colperm(S)` generates a permutation `j` so that both the rows and columns of `S(j,j)` are ordered according to increasing count of nonzero entries. If `S` is positive definite, this is sometimes useful as a preordering for Cholesky factorization; in this case use `chol(S(j,j))`.

Examples

The `n`-by-`n` *arrowhead* matrix

```
A = [ones(1,n); ones(n-1,1) speye(n-1,n-1)]
```

has a full first row and column. Its LU factorization, `lu(A)`, is almost completely full. The statement

```
j = colperm(A)
```

returns `j = [2:n 1]`. So `A(j,j)` sends the full row and column to the bottom and the rear, and `lu(A(j,j))` has the same nonzero structure as `A` itself.

On the other hand, the Bucky ball example,

```
B = bucky
```

has exactly three nonzero elements in each row and column, so `j = colperm(B)` is the identity permutation and is no help at all for reducing fill-in with subsequent factorizations.

Algorithms

The algorithm involves a sort on the counts of nonzeros in each column.

See Also

`chol` | `colamd` | `dissect` | `lu` | `spparms` | `symamd` | `symrcm`

Introduced before R2006a

COM

Access COM components from MATLAB

Description

A Component Object Model (COM) object

Creation

To create a `COM` object, call the `actxserver` function.

Object Functions

addproperty	Add custom property to COM object
deleteproperty	Remove custom property from COM object
events	List of events COM object can trigger
invoke	Invoke method on COM object or interface, or display methods
load	Initialize COM control object from file
move	Move or resize control in parent window
propedit	Open built-in property page for COM control
release	Release COM interface
save	Serialize COM control object to file

You also can use these MATLAB functions with COM objects.

delete	Delete files or objects
get	Query graphics object properties
isprop	True if property exists
ismethod	Determine if method of object
fieldnames	Field names of structure, or public fields of Java or Microsoft COM object
inspect	Open property inspector
methods	Class method names
methodsview	View class methods
set	Set graphics object properties

See Also

Introduced before R2006a

events

List of events COM object can trigger

Syntax

```
S = events(c)
```

Description

`S = events(c)` returns information about registered and unregistered events for COM object `c`.

COM functions are available on Microsoft Windows systems only.

Examples

List Workbook Events

Open a Microsoft Excel application and list all events for a `Workbook` object.

```
myApp = actxserver('Excel.Application');
wbs = myApp.Workbooks;
wb = Add(wbs);
events(wb)
```

Input Arguments

c — COM object
function handle

COM object, specified as a function handle.

Output Arguments

S — Event information
structure array

Information about registered and unregistered events known to the COM object, returned as a structure array. The array contains the function prototype used to call the event handler routine. For each array element, the structure field is the event name and the contents of that field is the function prototype for the handler of that event.

See Also

`eventlisteners` | `isevent` | `registerevent` | `unregisterallevents` | `unregisterevent`

Introduced before R2006a

load

Initialize COM control object from file

Syntax

```
load(c,filename)
```

Description

`load(c,filename)` initializes the COM object associated with the interface represented by the MATLAB COM object `c` from the file specified by `filename`. The file must have been created previously by serializing an instance of the same control.

COM functions are available on Microsoft Windows systems only.

Input Arguments

`c` — COM object
function handle

COM object, specified as a function handle.

`filename` — File name
character vector

File name, specified as a character vector.

See Also

`actxserver` | `delete` | `release` | `set`

Introduced before R2006a

propedit

Open built-in property page for COM control

Syntax

```
propedit(c)
```

Description

`propedit(c)` requests the COM control to display its built-in property page. If a control does not have a built-in property page, then `propedit` fails.

COM functions are available on Microsoft Windows systems only.

Input Arguments

c — COM object
function handle

COM object, specified as a function handle.

See Also

Property Inspector | `get`

Introduced before R2006a

save

Serialize COM control object to file

Syntax

```
save(c,filename)
```

Description

`save(c,filename)` saves COM control object to file specified by `filename`. The COM `save` function is only supported for controls.

COM functions are available on Microsoft Windows systems only.

Input Arguments

c — COM object
function handle

COM object, specified as a function handle.

`filename` — File name
character vector

File name, specified as a character vector.

See Also

`load`

Introduced before R2006a

Combine

Convenience function for static .NET System.Delegate Combine method

Syntax

```
result = Combine(delegateA,delegateB)
```

Description

`result = Combine(delegateA,delegateB)` combines two delegates into a new delegate.

Input Arguments

`delegateA`

.NET `System.Delegate` object. The first delegate in the new delegate.

Default:

`delegateB`

.NET `System.Delegate` object. The last delegate in the new delegate.

Default:

Output Arguments

`result`

.NET `System.Delegate` object. A new delegate that delegates to the input delegate `delegateA`, then `delegateB`

Alternatives

Use the static `Combine` method of the `System.Delegate` class.

See Also

`Remove` | `RemoveAll`

Topics

"Combine and Remove .NET Delegates"

External Websites

MSDN System.Delegate.Combine Method reference page

Introduced in R2011a

comet

2-D comet plot

Syntax

```
comet(y)
comet(x,y)
comet(x,y,p)
comet(ax,...)
```

Description

`comet(y)` displays a comet graph of the vector `y`. A comet graph is an animated graph in which a circle (the comet *head*) traces the data points on the screen. The comet *body* is a trailing segment that follows the head. The *tail* is a solid line that traces the entire function.

`comet(x,y)` displays a comet graph of vector `y` versus vector `x`.

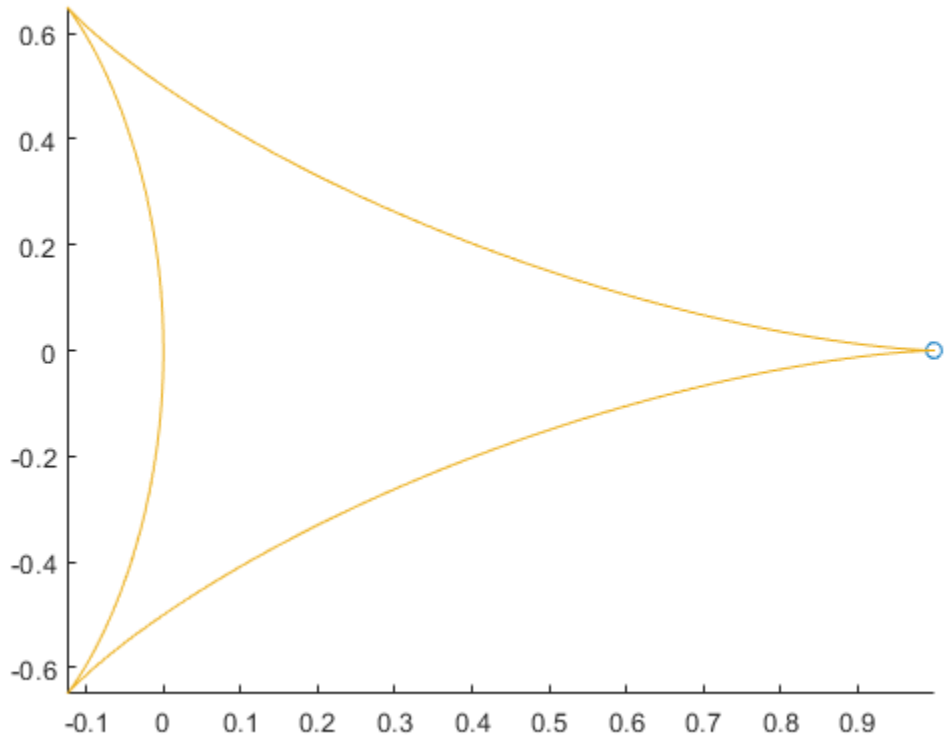
`comet(x,y,p)` specifies a comet body of length `p*length(y)`. `p` defaults to `0.1`.

`comet(ax,...)` plots into the axes specified by `ax` instead of into the current axes (`gca`). Specify Cartesian or geographic axes. If you specify geographic axes, `x` represents latitude coordinates in degrees and `y` represents longitude coordinates in degrees.

Examples

Create Comet Graph

```
t = 0:.01:2*pi;
x = cos(2*t).*(cos(t).^2);
y = sin(2*t).*(sin(t).^2);
comet(x,y);
```



Tips

- Comet graphs do not support data tips.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`animatedline` | `comet3`

Introduced before R2006a

comet3

3-D comet plot

Syntax

```
comet3(z)
comet3(x,y,z)
comet3(x,y,z,p)
comet3(ax,...)
```

Description

A comet plot is an animated graph in which a circle (the comet *head*) traces the data points on the screen. The comet *body* is a trailing segment that follows the head. The *tail* is a solid line that traces the entire function.

`comet3(z)` displays a 3-D comet graph of the vector `z`.

`comet3(x,y,z)` displays a comet graph of the curve through the points `[x(i),y(i),z(i)]`.

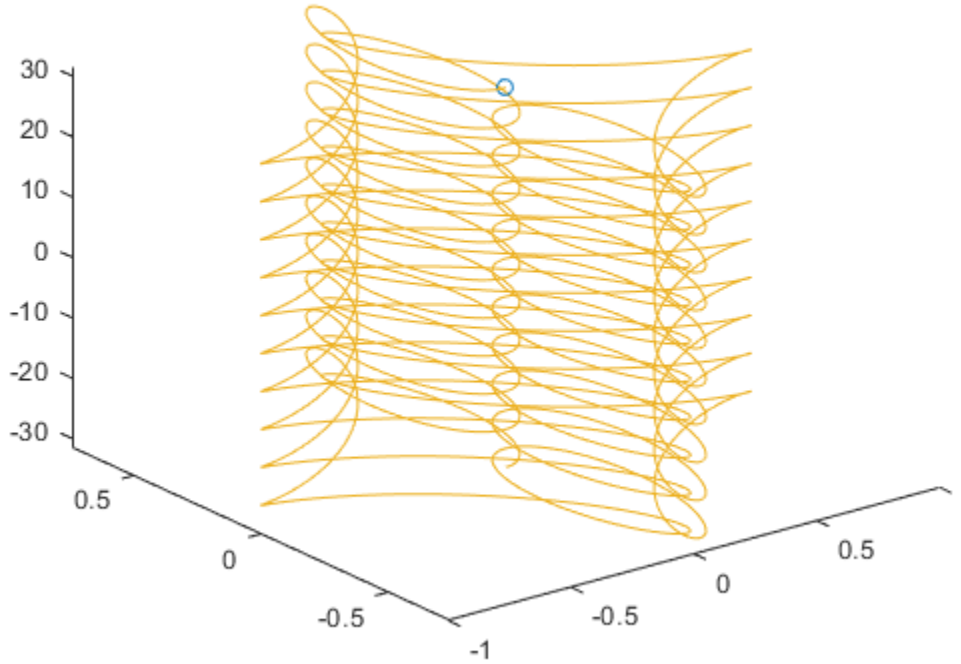
`comet3(x,y,z,p)` specifies a comet body of length `p*length(y)`. `p` must be between `0` and `1`.

`comet3(ax,...)` plots into the axes `ax` instead of into the current axes (`gca`).

Examples

Create 3-D Comet Graph

```
t = -10*pi:pi/250:10*pi;
x = (cos(2*t).^2).*sin(t);
y = (sin(2*t).^2).*cos(t);
comet3(x,y,t);
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`animatedline` | `comet`

Introduced before R2006a

Command History Window

Open Command History window

Description

The Command History window displays a log of statements that you ran in the current and previous MATLAB sessions. The Command History lists the time and date of each session in the short date format for your operating system, followed by the statements from that session. Brackets in the left margin indicate statements that were processed as a group. A colored mark precedes each statement that generated an error.

MATLAB saves statements that run in the Command Window to the history file `History.xml`. These statements include those you run using the **Evaluate Selection** item on context menus in tools such as the Editor, Command History window, and Help browser. By default, MATLAB automatically saves the command history file after each statement. The history file does not include every action taken in MATLAB. For example, changes to values in the Variables editor are not included in the Command History window. All entries remain until you delete them, or until the number of statements in the history file exceeds the number of statements to save, as specified in the Command History preferences. When the specified limit is reached, MATLAB automatically deletes the oldest entries. By default, the Command History window saves 25,000 statements.

You can select entries in the Command History window, and then perform these actions for the selected entries. Some actions are not available in MATLAB Online.

Action	How to Perform the Action
Create a script from statements.	Select an entry or entries, and then right-click and select **Create Script** or **Create Live Script** from the context menu. The Editor opens a new file that contains the statements you selected from the Command History window.

Action	How to Perform the Action
Rerun previous statements.	Do one of the following: • Press the Up Arrow key (↑) until the statement you want appears at the prompt, and then press **Enter**. • Double-click an entry or entries in the Command History window. • Select an entry in the Command History window and press **Enter**. To extend the selection to include multiple statements, press **Shift**+↑.
Copy statements to another window.	Select an entry or entries and then do one of the following: • Select **Copy** from the context menu. Paste the selection into an open file in the Editor or any application. • Drag the selection from the Command History window to an open file or another application.
Create a favorite command from statements.	Select an entry or entries, and then do one of the following: • Click the icon to the left of the selection. MATLAB creates a new favorite command in the default **Favorite Commands** category. • Right-click and select **Create Favorite**. The Favorite Command Editor opens with the selected entries added to the **Code** field. Configure the desired fields and click **Save**. For more information about favorite commands, see “Rerun Favorite Commands”.
Delete entries.	Select the entries to delete, and then right-click and select **Delete**, or press the **Delete** key. To recall a deleted entry, right-click and select **Undo Delete**, or press **Ctrl+Z**. To delete all entries, click (or in MATLAB Online), and then select **Clear Command History** from the context menu. You cannot recall entries deleted in this way.

Open the Command History Window

To open the Command History window with all history showing, in the Command Window, press the Up Arrow key (↑) or enter `commandhistory`. To open the Command History window and display a specific statement, type any part of the statement at the prompt and then press the Up Arrow key. To change how MATLAB searches for results, click (or ⋮ in MATLAB Online) and select from the available options including **Match Anywhere**, **Match Beginning**, and **Match Case**.

By default, the Command History window closes after a statement is selected or the **Esc** key is pressed. To keep the Command History window open, in the Command History window, click and then select either **Detach** or **Dock**. If the Command History window is closed while detached or docked, go to the **Home** tab, and in the **Environment** section, click **Layout**. Then, under **Show**, click **Command History** and select either **Docked** or **Popup**.

Examples

Find Previous Plot Statements

Use the Command History search feature to find and run previous `plot` statements.

In the Command Window, run these statements to create two line plots.

```
x = 0:pi/100:2*pi;
y = sin(x);
plot(x,y)

x1 = linspace(-2*pi,2*pi);
y1 = sin(x1);
y2 = cos(x1);

figure
plot(x1,y1,x1,y2)
```

Type `plot` and then press the up-arrow key. MATLAB displays the Command History with all instances of `plot` highlighted. The last run instance of `plot` is selected.

```
%-- 5/14/2018 3:38 PM --%
x = 0:pi/100:2*pi;
y = sin(x);
plot(x,y)
x1 = linspace(-2*pi,2*pi);
y1 = sin(x1);
y2 = cos(x1);
figure
plot(x1,y1,x1,y2)
```

Press the up-arrow key to select the previous run instance of `plot` and press **Enter**. MATLAB runs the statement `plot(x,y)`.

```
%-- 5/14/2018 3:38 PM --%
x = 0:pi/100:2*pi;
y = sin(x);
plot(x,y)
x1 = linspace(-2*pi,2*pi);
y1 = sin(x1);
y2 = cos(x1);
figure
plot(x1,y1,x1,y2)
```

Programmatic Use

`commandhistory` opens the MATLAB Command History window when it is closed, and selects the Command History window when it is open.

See Also

Command Window | `diary`

Topics

"Set Command History Preferences"

Introduced before R2006a

Command Window

Select the Command Window

Description

The Command Window enables you to enter individual statements at the command line and view the generated results.

Open the Command Window

The Command Window is always open. To restore the Command Window to the default location, go to the **Home** tab, and in the **Environment** section, click **Layout**. Then, select from one of the default layout options.

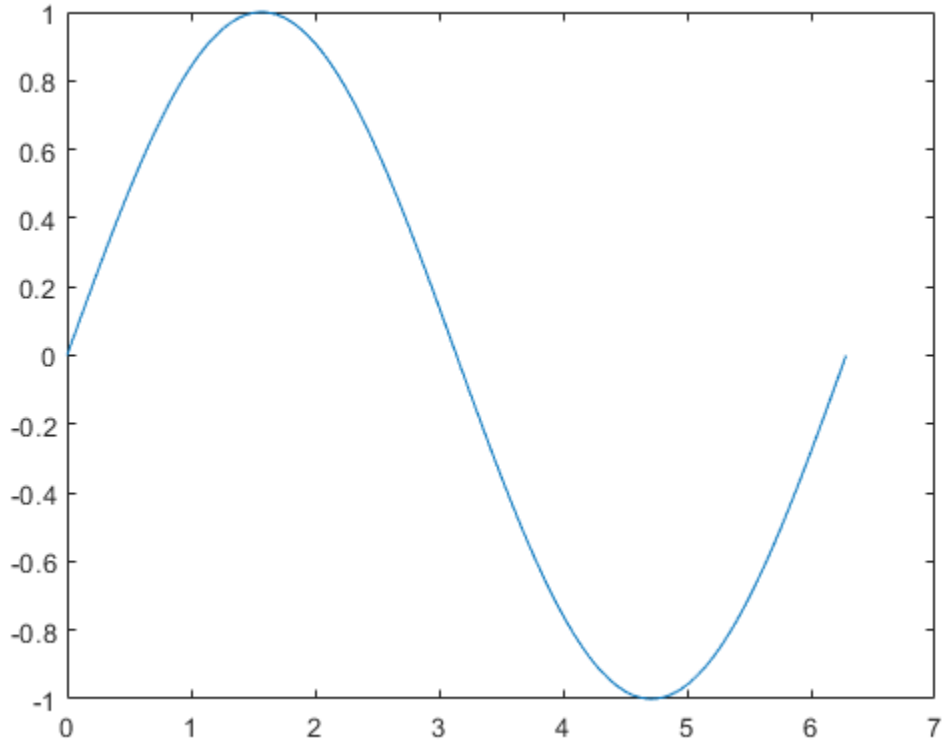
To bring focus to the Command Window from another tool such as the Editor, type `commandwindow`.

Examples

Select the Command Window After Creating a Plot

Create a line plot and then bring focus back to the Command Window.

```
x = 0:pi/100:2*pi;
y = sin(x);
plot(x,y)
```



`commandwindow`

Copyright 2018 The MathWorks, Inc

Programmatic Use

`commandwindow` selects the MATLAB Command Window. For example, type `commandwindow` after a plotting command to bring focus back to the Command Window.

`commandwindow` is not supported when running MATLAB with the `-nodesktop` option.

Tips

To determine the number of columns and rows that display in the Command Window given its current size, type `matlab.desktop.commandwindow.size` in the Command Window. MATLAB returns the number of columns and rows, respectively. For example:

```
matlab.desktop.commandwindow.size
```

```
ans =
   133    24
```

The number of columns is based on the width of the Command Window. If the **Set matrix width to eighty columns** Command Window preference is selected, the number of columns is always 80. For more information, see “Set Command Window Preferences”.

See Also

Command History Window | `input` | `inputdlg`

Topics

“Enter Statements in Command Window”
“Set Command Window Preferences”

Introduced before R2006a

compan

Companion matrix

Syntax

```
A = compan(u)
```

Description

`A = compan(u)` returns the corresponding companion matrix whose first row is `-u(2:n)/u(1)`, where `u` is a vector of polynomial coefficients. The eigenvalues of `compan(u)` are the roots of the polynomial.

Examples

Companion Matrix for Polynomial

Compute the companion matrix corresponding to the polynomial $(x-1)(x-2)(x+3) = x^3 - 7x + 6$.

```
u = [1 0 -7 6];
A = compan(u)
```

```
A = 3×3

     0     7    -6
     1     0     0
     0     1     0
```

The eigenvalues of `A` are the polynomial roots.

```
eig(A)
```

```
ans = 3×1
```

```
   -3.0000
    2.0000
    1.0000
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`eig` | `poly` | `polyval` | `roots`

Introduced before R2006a

compass

Plot arrows emanating from origin

Syntax

```
compass(U,V)
compass(Z)
compass(...,LineSpec)
compass(axes_handle,...)
h = compass(...)
```

Description

A compass graph displays the vectors with components `(U,V)` as arrows emanating from the origin. `U`, `V`, and `Z` are in Cartesian coordinates and plotted on a circular grid.

`compass(U,V)` displays a compass graph having *n* arrows, where *n* is the number of elements in `U` or `V`. The location of the base of each arrow is the origin. The location of the tip of each arrow is a point relative to the base and determined by `[U(i),V(i)]`.

`compass(Z)` displays a compass graph having *n* arrows, where *n* is the number of elements in `Z`. The location of the base of each arrow is the origin. The location of the tip of each arrow is relative to the base as determined by the real and imaginary components of `Z`. This syntax is equivalent to `compass(real(Z),imag(Z))`.

`compass(...,LineSpec)` draws a compass graph using the line type, marker symbol, and color specified by `LineSpec`.

`compass(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

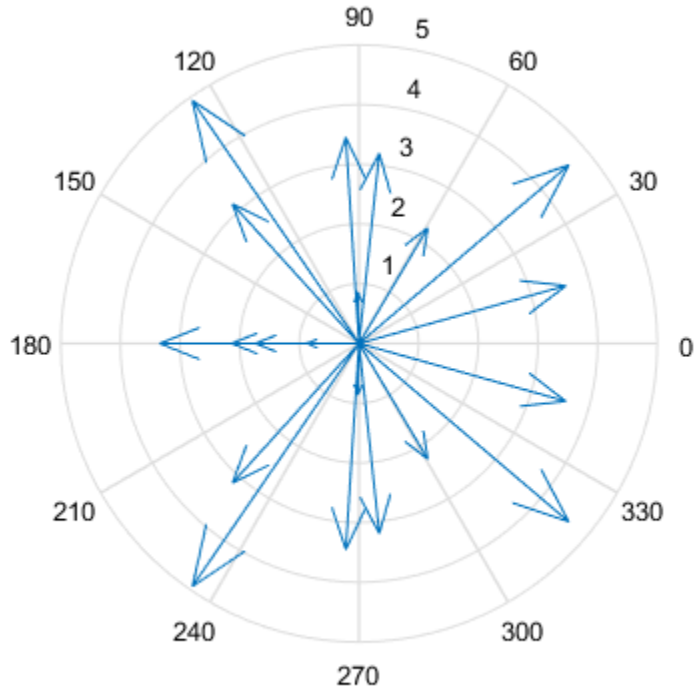
`h = compass(...)` returns handles to line objects.

Examples

Create Compass Graph

Create a compass graph of the eigenvalues of a random matrix.

```
rng(0,'twister') % initialize random number generator
M = randn(20,20);
Z = eig(M);

figure
compass(Z)
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`feather` | `polarplot` | `quiver` | `rose`

Introduced before R2006a

complex

Create complex array

Syntax

```
z = complex(a,b)
z = complex(x)
```

Description

`z = complex(a,b)` creates a complex output, `z`, from two real inputs, such that `z = a + bi`.

The `complex` function provides a useful substitute for expressions, such as `a + 1i*b` or `a + 1j*b`, when

- `a` and `b` are not `double` or `single`
- `b` is all zeros

`z = complex(x)` returns the complex equivalent of `x`, such that `isreal(z)` returns logical `0` (`false`).

- If `x` is real, then `z` is `x + 0i`.
- If `x` is complex, then `z` is identical to `x`.

Examples

Complex Scalar from Two Real Scalars

Use the `complex` function to create the complex scalar, `3 + 4i`.

```
z = complex(3,4)
```

```
z = 3.0000 + 4.0000i
```

Complex Vector from Two Real Vectors

Create a complex `uint8` vector from two real `uint8` vectors. The size of `z`, 4-by-1, is the same as the size of the input arguments.

```
a = uint8([1;2;3;4]);
b = uint8([2;2;7;7]);

z = complex(a,b)
```

```
z = 4x1 uint8 column vector

   1 +   2i
   2 +   2i
   3 +   7i
   4 +   7i
```

Complex Scalar from One Real Scalar

Create a complex scalar with zero imaginary part.

```
z = complex(12)
```

```
z = 12.0000 + 0.0000i
```

Verify that `z` is complex.

```
isreal(z)
```

```
ans = logical
   0
```

Input Arguments

a — Real component

scalar | vector | matrix | multidimensional array

Real component, specified as a scalar, vector, matrix, or multidimensional array.

The size of a must match the size of b, unless one is a scalar. If either a or b is a scalar, MATLAB expands the scalar to match the size of the other input.

a and b must be the same data type with the following exceptions:

- `single` can combine with `double`.
- scalar `double` can combine with an integer data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

b — Imaginary component

scalar | vector | matrix | multidimensional array

Imaginary component, specified as a scalar, vector, matrix, or multidimensional array.

The size of b must match the size of a, unless one is a scalar. If either a or b is a scalar, MATLAB expands the scalar to match the size of the other input.

a and b must be the same data type with the following exceptions:

- `single` can combine with `double`.
- scalar `double` can combine with an integer data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

x — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`
Complex Number Support: Yes

Output Arguments

z — Complex array
scalar | vector | matrix | multidimensional array

Complex array, returned as a scalar, vector, matrix, or multidimensional array.

The size of `z` is the same as the input arguments.

The following describes the data type of `z`, when `a` and `b` have different data types.

- If either `a` or `b` is `single`, then `z` is `single`.
- If either `a` or `b` is an integer data type, then `z` is the same integer data type.

Tips

- If `b` contains only zeros, then `z` is complex and the value of all its imaginary components is `0`. In contrast, the addition `a + 0i` returns a strictly real result.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`abs` | `angle` | `conj` | `i` | `imag` | `isreal` | `j` | `real`

Topics

“Create Complex Numbers”

Introduced before R2006a

compose

Format data into multiple strings

Syntax

```
str = compose(formatSpec,A)
str = compose(formatSpec,A1,...,AN)
str = compose(txt)
```

Description

`str = compose(formatSpec,A)` formats data values from the input array, `A`, using formatting operators specified by `formatSpec` and returns the resulting text in `str`. The `compose` function formats values from `A` in column order. If `formatSpec` is a string array, then so is the output array `str`. Otherwise, `str` is a cell array of character vectors.

`compose` also translates the escape-character sequences in `formatSpec`. Escape-character sequences represent nonprinting characters or specify actions such as newlines or tabs.

The `compose` function can return multiple pieces of formatted text as a string array or a cell array of character vectors, unlike `sprintf`. The `sprintf` function returns only a string scalar or a character vector.

- If `A` has multiple rows, then `compose` returns `str` as a string array or cell array with the same number of rows. `compose` repeats `formatSpec` in each row of `str`, with formatted values from the corresponding row of `A`.
- If the number of columns in `A` exceeds the number of operators in `formatSpec`, then `compose` repeats `formatSpec` as an additional column of `str`. The extra columns of `A` contribute formatted values to the new column in `str`.
- If the number of columns in `A` is less than the number of operators in `formatSpec`, then `compose` does not format values using those operators. Instead, `compose` puts unchanged formatting operators in `str`. However, `compose` translates all escape-character sequences except for `\\` and `%%`.

`str = compose(formatSpec,A1,...,AN)` formats data values from multiple input arrays and concatenates all the formatted values. When `compose` uses formatting operators from `formatSpec` to convert data from an input array, then those formatting operators become unavailable to the following input arrays.

For example, if `formatSpec` is `"%f %f %d %s"` and `A1` has two columns, then the operators `"%f %f"` are applied to the values in `A1` only. They cannot be applied to `A2` or any other input array. `compose` applies the remaining operators, `"%d %s"`, to `A2,...,AN`.

If the number of columns in the last input array, `AN`, exceeds the number of remaining operators, then `compose` adds an additional column to `str`, as described in the previous syntax. If the number of columns in `AN` is less than the number of remaining operators, then `compose` puts the last unchanged operators in `str`.

`str = compose(txt)` translates escape-character sequences in `txt`.

- If `txt` does not contain formatting operators, then `compose` translates all escape-character sequences. It leaves all other characters unchanged.
- If `txt` contains formatting operators, then `compose` translates all escape-character sequences except for `\\` and `%%`. It leaves all other characters, including the formatting operators, unchanged.

Examples

Format Numbers into Strings

Format `pi` to eight decimal places and return it as a string.

```
A = pi
```

```
A = 3.1416
```

Starting in R2017a, you can create strings using double quotes. Specify `formatSpec` as a string.

```
formatSpec = "%.8f"
```

```
formatSpec =
"%.8f"
```

```
str = compose(formatSpec,A)
```

```
str =
"3.14159265"
```

Create a numeric array that contains values of `pi` and `e`. Use the `%e` and `%f` operators with different precisions.

```
A = [pi exp(1)]
```

```
A = 1×2

    3.1416    2.7183
```

```
formatSpec = "The value of pi is %.2e; the value of e is %.5f.";
str = compose(formatSpec,A)
```

```
str =
"The value of pi is 3.14e+00; the value of e is 2.71828."
```

Format Columns of Values from Arrays

Format values taken from numeric arrays. Since the numeric arrays have multiple rows, `compose` returns a string array with the same number of rows.

```
X = [1 2 3 4 5]';
Y = X.^2;
```

Starting in R2017a, you can create strings using double quotes. Specify `formatSpec` as a string and return the formatted values as a string array.

```
formatSpec = "%d.^2 = %d";
str = compose(formatSpec,X,Y)
```

```
str = 5x1 string array
    "1.^2 = 1"
    "2.^2 = 4"
    "3.^2 = 9"
    "4.^2 = 16"
    "5.^2 = 25"
```

Format Values with Extra or Missing Operators

Format values when the number of columns in the data array is not equal to the number of operators. If A has more columns, then `compose` repeats `formatSpec` as an additional column of the output string array.

Starting in R2017a, you can create strings using double quotes. Specify `formatSpec` as a string.

```
formatSpec = "The time is %d:%d";
A = [8 15 9 30;
     10 20 11 50];
str = compose(formatSpec,A)
```

```
str = 2x2 string array
    "The time is 8:15"     "The time is 9:30"
    "The time is 10:20"    "The time is 11:50"
```

Format values when A has fewer columns.

```
formatSpec = "Check-in time %d:%d; Check-out time %d:%d";
A = [12 27;
     11 16];
str = compose(formatSpec,A)
```

```
str = 2x1 string array
    "Check-in time 12:27; Check-out time %d:%d"
    "Check-in time 11:16; Check-out time %d:%d"
```

Since A has only two columns, `compose` uses only the first two formatting operators in `formatSpec` to format the values. `compose` leaves the other formatting operators unchanged.

Escape Characters in String Array

Create a string array that includes escape-character sequences to specify horizontal tabs. Use the `compose` function to translate the `\t` escape characters. Starting in R2017a, you can create strings using double quotes.

```
str = ["Name\tDate of Birth\tLocation";...
       "Jones\t10/20/2015\tUK";...
       "Simpson\t09/12/2015\tUSA"];
newStr = compose(str)
```

```
newStr = 3x1 string array
    "Name->Date of Birth->Location"
    "Jones->10/20/2015->UK"
    "Simpson->09/12/2015->USA"
```

Prevent translation of `\n` using another `\` character.

```
str = "Don't escape the second\n\\n escaped-character sequence.";
newStr = compose(str)
```

```
newStr =
    "Don't escape the second
     \n escaped-character sequence."
```

Input Arguments

`formatSpec` — Formatting operators
string array | character vector | cell array of character vectors

Formatting operators, specified as a string array, character vector, or cell array of character vectors containing formatting operators. If `formatSpec` includes literal text representing escape characters, such as `\n`, then `compose` translates the escape characters.

Formatting Operator

A formatting operator starts with a percent sign, `%`, and ends with a conversion character. The conversion character is required. Optionally, you can specify flags, field width, precision, and subtype operators between `%` and the conversion character. (Spaces are invalid between operators and are shown here only for readability).

Conversion Character

This table shows conversion characters to format numeric and character data as text.

Value Type	Conversion	Details
Integer, signed	`%d` or `%i`	Base 10
Integer, unsigned	`%u`	Base 10
	`%o`	Base 8 (octal)
	`%x`	Base 16 (hexadecimal), lowercase letters `a-f`
	`%X`	Same as `%x`, uppercase letters `A-F`
Floating-point number	`%f`	Fixed-point notation (Use a precision operator to specify the number of digits after the decimal point.)
	`%e`	Exponential notation, such as `3.141593e+00` (Use a precision operator to specify the number of digits after the decimal point.)
	`%E`	Same as `%e`, but uppercase, such as `3.141593E+00` (Use a precision operator to specify the number of digits after the decimal point.)
	`%g`	The more compact of `%e` or `%f`, with no trailing zeros (Use a precision operator to specify the number of significant digits.)
	`%G`	The more compact of `%E` or `%f`, with no trailing zeros (Use a precision operator to specify the number of significant digits.)
Characters	`%c`	Single character
	`%s`	Character vector

Optional Operators

The optional identifier, flags, field width, precision, and subtype operators further define the format of the output text.

- **Flags**

`'-'`	Left-justify. **Example:** `%-5.2f`
`'+'`	Always print a sign character (+ or –) for any value. **Example:** `%+5.2f`
`' '`	Insert a space before the value. **Example:** `% 5.2f`
`'0'`	Pad to field width with zeros before the value. **Example:** `%05.2f`
`'#'`	Modify selected numeric conversions: • For `%o`, `%x`, or `%X`, print `0`, `0x`, or `0X` prefix. • For `%f`, `%e`, or `%E`, print decimal point even when precision is 0. • For `%g` or `%G`, do not remove trailing zeros or decimal point. **Example:** `%#5.0f`

- **Field Width**

  Minimum number of characters to print.

  Example: `compose('%12d',intmax)` returns `" 2147483647"`.

  The function pads to field width with spaces before the value unless otherwise specified by flags.

- **Precision**

  Number of digits to print.

For `%f`, `%e`, or `%E`	Number of digits to the right of the decimal point **Example:** `'%.4f'` prints `pi` as `'3.1416'`
For `%g` or `%G`	Number of significant digits **Example:** `'%.4g'` prints `pi` as `'3.142'`

  Example: `compose('%6.4f',pi)` returns `"3.1416"`.

  Note If you specify a precision operator for floating-point values that exceeds the precision of the input numeric data type, the results might not match the input values to the precision you specified. The result depends on your computer hardware and operating system.

- **Subtypes**

  You can use a subtype operator to print a floating-point value as its octal, decimal, or hexadecimal value. The subtype operator immediately precedes the conversion character. This table shows the conversions that can use subtypes.

Input Value Type	Subtype and Conversion Character	Output Value Type

Floating-point number	`%bx` or `%bX`
	`%bo`
	`%bu`
		Double-precision hexadecimal, octal, or decimal value
		Example: `%bx` prints `pi` as `400921fb54442d18`

	`%tx` or `%tX`
	`%to`
	`%tu`
		Single-precision hexadecimal, octal, or decimal value
		Example: `%tx` prints `pi` as `40490fdb`

Notable Behavior of Conversions with Formatting Operators

- Numeric conversions print only the real component of complex numbers.
- If you specify a conversion that does not fit the data, such as a character conversion for a numeric value, MATLAB overrides the specified conversion, and uses `%e`.

  Example: `'%s'` converts `pi` to `3.141593e+00`.
- If you apply a string conversion (`%s`) to integer values, MATLAB converts values that correspond to valid character codes to characters.

  Example: `'%s'` converts `[65 66 67]` to `"ABC"`.

Data Types: `string` | `char` | `cell`

A — Numeric, character, or string array

scalar | vector | matrix | multidimensional array

Numeric, character, or string array, specified as scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string`

`txt` — Input text
string array | character vector | cell array of character vectors

Input text, specified as a string array, character vector, or cell array of character vectors. `compose` translates any escape-character sequences in `txt`. For example, `compose` translates `\n` into a newline character.

Data Types: `string` | `char` | `cell`

Output Arguments

`str` — Formatted text
string array | cell array of character vectors

Formatted text, returned as a string array or a cell array of character vectors.

Data Types: `string` | `cell`

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitation:

The `format` input must be a non-tall string.

For more information, see “Tall Arrays for Out-of-Memory Data”.

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`fprintf` | `fscanf` | `sprintf` | `sscanf` | `string`

Topics

"Create String Arrays"
"Formatting Text"

Introduced in R2016b

computeStrip

Index number of strip containing specified coordinate

Syntax

```
stripNumber = computeStrip(t,row)
stripNumber = computeStrip(t,row,plane)
```

Description

`stripNumber = computeStrip(t,row)` returns the index of the strip containing `row`. The value of `row` must be one-based. `computeStrip` clamps out-of-range coordinate values to the bounds of the image.

`stripNumber = computeStrip(t,row,plane)` returns the index of the strip containing the row in the specified `plane`.

Examples

Determine Index of Strip

Determine the index of the strip containing a specific row from the second image in a TIFF file.

Create a `Tiff` object for the file `example.tif`, and then set the image file directory to directory number 2.

```
t = Tiff('example.tif','r');
setDirectory(t,2)
```

Get the length of the strips in the image, and then get the index of the strip containing the middle row.

```
stripLength = getTag(t,'RowsPerStrip')
```

```
stripLength = 100
```

```
numRows = getTag(t,'ImageLength');
stripNumber = computeStrip(t,numRows/2)
```

```
stripNumber = 4
```

Close the `Tiff` object.

```
close(t);
```

Input Arguments

`t` — Tiff object
`Tiff` object

`Tiff` object representing a TIFF file. Use the `Tiff` function to create the object.

`row` — Row Number
positive integer

Row number, specified as a positive integer. The value of `row` must be one-based.

Example: `10`

Data Types: `double`

`plane` — Plane Number
positive integer

Plane number, specified as a positive integer. The value of `plane` must be one-based.

Use the `plane` parameter only if the data is organized in separate planes. Data is organized in separate planes when the value of the `PlanarConfiguration` tag is `Tiff.PlanarConfiguration.Separate`.

Example: `1`

Data Types: `double`

Algorithms

References

This function corresponds to the `TIFFComputeStrip` function in the LibTIFF C API. To use this function, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff` | `computeTile`

Topics

"Importing Images"
"Exporting to Images"

Introduced in R2009b

computeTile

Index number of tile containing specified coordinates

Syntax

```
tileNumber = computeTile(t,coord)
tileNumber = computeTile(t,coord,plane)
```

Description

`tileNumber = computeTile(t,coord)` returns the index of the tile containing the pixel specified in `coord`. The `computeTile` function clamps out-of-range coordinate values to the bounds of the image.

`tileNumber = computeTile(t,coord,plane)` returns the index of the tile containing the pixel from the specified `plane`.

Examples

Get Index of Tile Containing Last Pixel

Create a `Tiff` object for a file and get the number of rows and columns in the image.

```
t = Tiff('example.tif','r');
numRows = getTag(t,'ImageLength');
numCols = getTag(t,'ImageWidth');
```

Get the index of the tile containing the last pixel of the image.

```
tileNum = computeTile(t,[numRows numCols])
```

```
tileNum = 110
```

Close the `Tiff` object.

```
close(t);
```

Input Arguments

t — Tiff object
`Tiff` object

`Tiff` object representing a TIFF file. Use the `Tiff` function to create the object.

coord — Pixel coordinates
integer array in the form `[row, col]`

Pixel coordinates, specified as an integer array in the form `[row, col]`. The integers `row` and `col` are one-based positive scalar indices.

Example: `[100,50]`

Data Types: `double`

plane — Plane Number
positive integer

Plane number, specified as a positive integer. The value of `plane` must be one-based.

Use the `plane` parameter only if the data is organized in separate planes. Data is organized in separate planes when the value of the `PlanarConfiguration` tag is `Tiff.PlanarConfiguration.Separate`.

Example: `1`

Data Types: `double`

Algorithms

References

This function corresponds to the `TIFFComputeTile` function in the LibTIFF C API. To use this function, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff` | `computeStrip`

Topics

"Importing Images"
"Exporting to Images"

Introduced in R2009b

computer

Information about computer on which MATLAB is running

Syntax

```
str = computer
archstr = computer('arch')
[str,maxsize] = computer
[str,maxsize,endian] = computer
```

Description

`str = computer` returns the computer type on which MATLAB is running.

`archstr = computer('arch')` returns the system architecture. The `mex` command and standalone applications use this value to locate MATLAB library files.

`[str,maxsize] = computer` returns the maximum number of elements allowed in an array with this version of MATLAB.

`[str,maxsize,endian] = computer` returns the endian byte order format.

Output Arguments

`str` — Computer type
`'PCWIN64'` | `'GLNXA64'` | `'MACI64'`

Computer type, returned as a character vector.

- `'PCWIN64'` - 64-bit Windows platform
- `'GLNXA64'` - 64-bit Linux platform
- `'MACI64'` - 64-bit macOS platform

`archstr` — System architecture
`'win64'` | `'glnxa64'` | `'maci64'`

System architecture, returned as a character vector.

- `'PCWIN64'` - 64-bit Windows platform
- `'GLNXA64'` - 64-bit Linux platform
- `'MACI64'` - 64-bit macOS platform

`maxsize` — Maximum allowed array elements
`2^48 - 1`

Maximum number of elements allowed in an array, returned as double.

`endian` — Endian byte order format
`'L'|'B'`

Endian byte order format, returned as a character vector.

- `'L'` - little-endian byte ordering for 64-bit Windows, Linux, and macOS platforms
- `'B'` - big-endian byte ordering

Algorithms

For 64-bit Windows platforms:

- `ispc` returns 1 (true)
- `isunix` returns 0 (false)
- `ismac` returns 0 (false)

For 64-bit Linux platforms:

- `ispc` returns 0 (false)
- `isunix` returns 1 (true)
- `ismac` returns 0 (false)

For 64-bit macOS platforms:

- `ispc` returns 0 (false)
- `isunix` returns 1 (true)
- `ismac` returns 1 (true)

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Information about the computer on which the code generator is running.
- Use only when the code generation target is S-function (Simulation) or MEX-function.

See Also

`getenv` | `ismac` | `ispc` | `isunix` | `setenv` | `winopen`

External Websites

System Requirements

Introduced before R2006a

cond

Condition number for inversion

Syntax

```
C = cond(A)
C = cond(A,p)
```

Description

`C = cond(A)` returns the 2-norm condition number for inversion on page 1-2043, equal to the ratio of the largest singular value of `A` to the smallest.

`C = cond(A,p)` returns the `p`-norm condition number, where `p` can be `1`, `2`, `Inf`, or `'fro'`.

Examples

Condition Number of Matrix

Calculate the condition number of a matrix and examine the sensitivity to the inverse calculation.

Create a 2-by-2 matrix.

```
A = [4.1 2.8;
     9.7 6.6];
```

Calculate the 2-norm condition number of `A`.

```
C = cond(A)
```

```
C = 1.6230e+03
```

Since the condition number of A is much larger than 1, the matrix is sensitive to the inverse calculation. Calculate the inverse of A, and then make a small change in the second row of A and calculate the inverse again.

```
invA = inv(A)
```

```
invA = 2×2

  -66.0000   28.0000
   97.0000  -41.0000
```

```
A2 = [4.1    2.8;
      9.671  6.608]
```

```
A2 = 2×2

    4.1000    2.8000
    9.6710    6.6080
```

```
invA2 = inv(A2)
```

```
invA2 = 2×2

  472.0000 -200.0000
 -690.7857  292.8571
```

The results indicate that making a small change in A can completely change the result of the inverse calculation.

1-Norm Condition Number

Calculate the 1-norm condition number of a matrix.

Create a 3-by-3 matrix.

```
A = [1 0 -2;
     3 4  6;
    -1 5  7];
```

Calculate the 1-norm condition number of A. The value of the 1-norm condition number for an *m*-by-*n* matrix is

$$\kappa_1(A) = \left|\left|A\right|\right|_1 \left|\left|A^{-1}\right|\right|_1,$$

where the 1-norm is the maximum absolute column sum of the matrix given by

$$\left|\left|A\right|\right|_1 = \max_{1 \le j \le n} \sum_{i=1}^{m} \left|a_{ij}\right| .$$

```
C = cond(A,1)
```

```
C = 18.0000
```

For this matrix the condition number is not too large, so the matrix is not particularly sensitive to the inverse calculation.

Input Arguments

A — Input matrix
matrix

Input matrix. A can be either square or rectangular in size.

Data Types: `single` | `double`
Complex Number Support: Yes

p — Norm type
2 (default) | `1` | `'fro'` | `Inf`

Norm type, specified as one of the values shown in this table. `cond` computes the condition number using `norm(A,p) * norm(inv(A),p)` for values of `p` other than 2. See the `norm` page for additional information about these norm types.

Value of p	Norm Type
1	1-norm condition number
2	2-norm condition number

Value of p	Norm Type
`Inf`	Infinity norm condition number
`'fro'`	Frobenius norm condition number

Example: `cond(A,1)` calculates the 1-norm condition number.

Output Arguments

`C` — Condition number
scalar

Condition number, returned as a scalar. Values of `C` near 1 indicate a well-conditioned matrix, and large values of `C` indicate an ill-conditioned matrix. Singular matrices have a condition number of `Inf`.

More About

Condition Number for Inversion

A *condition number* for a matrix and computational task measures how sensitive the answer is to changes in the input data and roundoff errors in the solution process.

The *condition number for inversion* of a matrix measures the sensitivity of the solution of a system of linear equations to errors in the data. It gives an indication of the accuracy of the results from matrix inversion and the linear equation solution. For example, the 2-norm condition number of a matrix is

$$\kappa(A) = \|A\|\|A^{-1}\| \ .$$

In this context, a large condition number indicates that a small change in the coefficient matrix `A` can lead to larger changes in the output `b` in the linear equations $Ax = b$ and $xA = b$. The extreme case is when `A` is so poorly conditioned that it is singular (an infinite condition number), in which case it has no inverse and the linear equation has no unique solution.

Tips

- `rcond` is a more efficient, but less reliable, method of estimating the condition of a matrix compared to `cond`.

Algorithms

The algorithm for `cond` has three pieces:

- If `p = 2`, then `cond` uses the singular value decomposition provided by `svd` to find the ratio of the largest and smallest singular values.
- If `p = 1`, `Inf`, or `'fro'`, then `cond` calculates the condition number using the appropriate norm of the input matrix and its inverse with `norm(A,p) * norm(inv(A),p)`.
- If the input matrix is sparse, then `cond` ignores any specified `p` value and calls `condest`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- `cond` does not support sparse matrices.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`condeig` | `condest` | `norm` | `normest` | `rank` | `rcond` | `svd`

External Websites

Cleve's Corner: What is the Condition Number of a Matrix?

Introduced before R2006a

condeig

Condition number with respect to eigenvalues

Syntax

```
c = condeig(A)
[V,D,s] = condeig(A)
```

Description

`c = condeig(A)` returns a vector of condition numbers for the eigenvalues of `A`. These condition numbers are the reciprocals of the cosines of the angles between the left and right eigenvectors.

`[V,D,s] = condeig(A)` is equivalent to

```
[V,D] = eig(A);
s = condeig(A);
```

Large condition numbers imply that `A` is near a matrix with multiple eigenvalues.

See Also

`balance` | `cond` | `eig`

Introduced before R2006a

condest

1-norm condition number estimate

Syntax

```
c = condest(A)
c = condest(A,t)
[c,v] = condest(A)
```

Description

`c = condest(A)` computes a lower bound `c` for the 1-norm condition number of a square matrix `A`.

`c = condest(A,t)` changes `t`, a positive integer parameter equal to the number of columns in an underlying iteration matrix. Increasing the number of columns usually gives a better condition estimate but increases the cost. The default is `t = 2`, which almost always gives an estimate correct to within a factor 2.

`[c,v] = condest(A)` also computes a vector `v` which is an approximate null vector if `c` is large. `v` satisfies `norm(A*v,1) = norm(A,1)*norm(v,1)/c`.

Note `condest` invokes `rand`. If repeatable results are required then use `rng` to set the random number generator to its startup settings before using `condest`.

```
rng('default')
```

Tips

This function is particularly useful for sparse matrices.

Algorithms

`condest` is based on the 1-norm condition estimator of Hager [1] and a block-oriented generalization of Hager's estimator given by Higham and Tisseur [2]. The heart of the algorithm involves an iterative search to estimate $\|A^{-1}\|_1$ without computing A^{-1}. This is posed as the convex but nondifferentiable optimization problem $\max\|A^{-1}\mathbf{x}\|_1$ subject to $\|\mathbf{x}\|_1 = 1$

References

[1] William W. Hager, “Condition Estimates,” *SIAM J. Sci. Stat. Comput. 5*, 1984, 311-316, 1984.

[2] Nicholas J. Higham and Françoise Tisseur, “A Block Algorithm for Matrix 1-Norm Estimation with an Application to 1-Norm Pseudospectra, “*SIAM J. Matrix Anal. Appl.*, Vol. 21, 1185-1201, 2000.

See Also

`cond` | `norm` | `normest`

Introduced before R2006a

coneplot

Plot velocity vectors as cones in 3-D vector field

Syntax

```
coneplot(X,Y,Z,U,V,W,Cx,Cy,Cz)
coneplot(U,V,W,Cx,Cy,Cz)
coneplot(...,s)
coneplot(...,color)
coneplot(...,'quiver')
coneplot(...,'method')
coneplot(X,Y,Z,U,V,W,'nointerp')
coneplot(axes_handle,...)
h = coneplot(...)
```

Description

`coneplot(X,Y,Z,U,V,W,Cx,Cy,Cz)` plots velocity vectors as cones pointing in the direction of the velocity vector and having a length proportional to the magnitude of the velocity vector. `X`, `Y`, `Z` define the coordinates for the vector field. `U`, `V`, `W` define the vector field. These arrays must be the same size, monotonic, and represent a Cartesian, axis-aligned grid (such as the data produced by `meshgrid`). `Cx`, `Cy`, `Cz` define the location of the cones in the vector field. The section “Specifying Starting Points for Stream Plots” in Visualization Techniques provides more information on defining starting points.

`coneplot(U,V,W,Cx,Cy,Cz)` (omitting the `X`, `Y`, and `Z` arguments) assumes `[X,Y,Z] = meshgrid(1:n,1:m,1:p)`, where `[m,n,p]= size(U)`.

`coneplot(...,s)` automatically scales the cones to fit the graph and then stretches them by the scale factor `s`. If you do not specify a value for `s`, `coneplot` uses a value of 1. Use `s = 0` to plot the cones without automatic scaling.

`coneplot(...,color)` interpolates the array `color` onto the vector field and then colors the cones according to the interpolated values. The size of the `color` array must be the same size as the `U`, `V`, `W` arrays. This option works only with cones (that is, not with the `quiver` option).

`coneplot(...,'quiver')` draws arrows instead of cones (see `quiver3` for an illustration of a quiver plot).

`coneplot(...,'method')` specifies the interpolation method to use. *`method`* can be `linear`, `cubic`, or `nearest`. `linear` is the default. (See `interp3` for a discussion of these interpolation methods.)

`coneplot(X,Y,Z,U,V,W,'nointerp')` does not interpolate the positions of the cones into the volume. The cones are drawn at positions defined by `X`, `Y`, `Z` and are oriented according to `U`, `V`, `W`. Arrays `X`, `Y`, `Z`, `U`, `V`, `W` must all be the same size.

`coneplot(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = coneplot(...)` returns the handle to the `patch` object used to draw the cones.

`coneplot` automatically scales the cones to fit the graph, while keeping them in proportion to the respective velocity vectors.

Examples

3-D Cone Plot

Plot velocity vector cones for vector volume data representing motion of air through a rectangular region of space.

Load the data. The `wind` data set contains the arrays `u`, `v`, and `w` that specify the vector components and the arrays `x`, `y`, and `z` that specify the coordinates.

```
load wind
```

Establish the range of the data to place the slice planes and to specify where you want the cone plots.

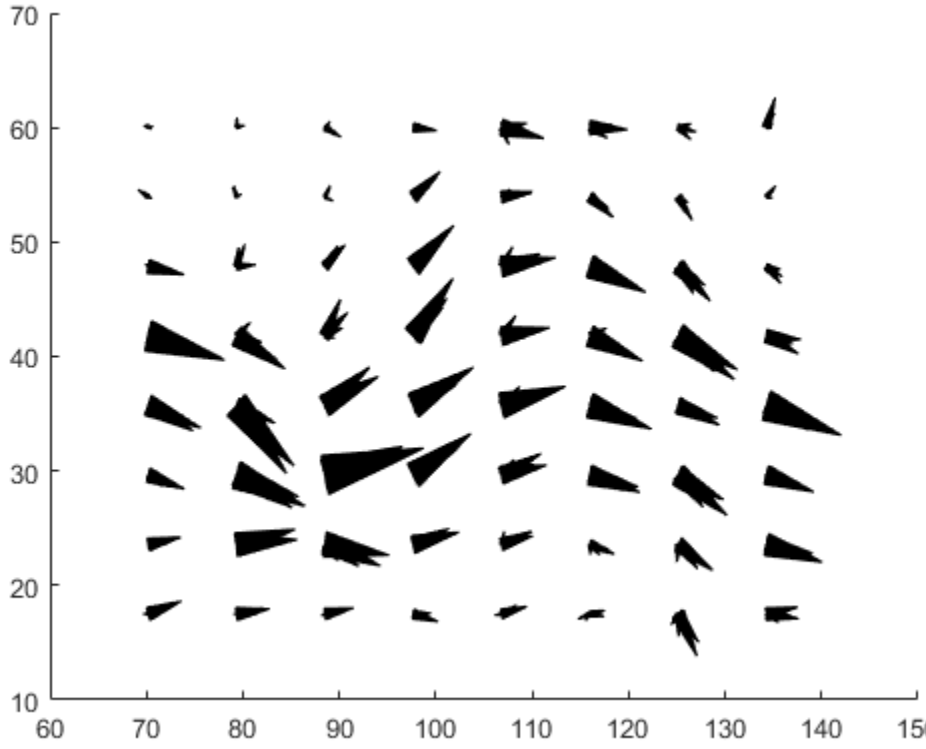
```
xmin = min(x(:));
xmax = max(x(:));
ymin = min(y(:));
ymax = max(y(:));
zmin = min(z(:));
```

Define where to plot the cones. Select the full range in x and y and select the range 3 to 15 in z.

```
xrange = linspace(xmin,xmax,8);
yrange = linspace(ymin,ymax,8);
zrange = 3:4:15;
[cx,cy,cz] = meshgrid(xrange,yrange,zrange);
```

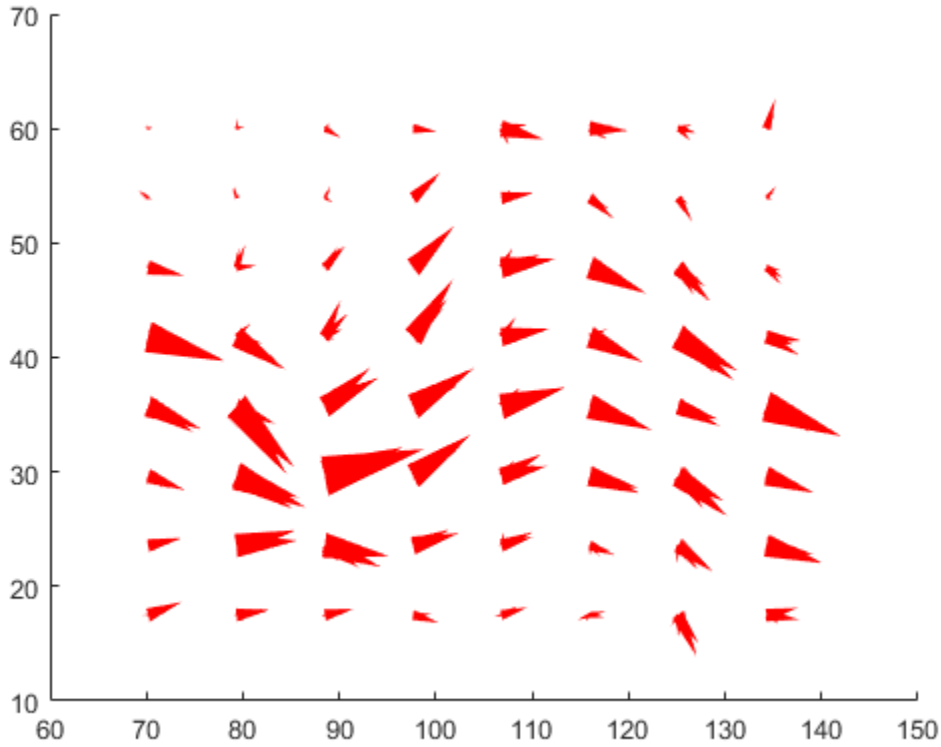
Plot the cones and set the scale factor to 5 to make the cones larger than the default size.

```
figure
hcone = coneplot(x,y,z,u,v,w,cx,cy,cz,5);
```



Set the cone colors.

```
hcone.FaceColor = 'red';
hcone.EdgeColor = 'none';
```

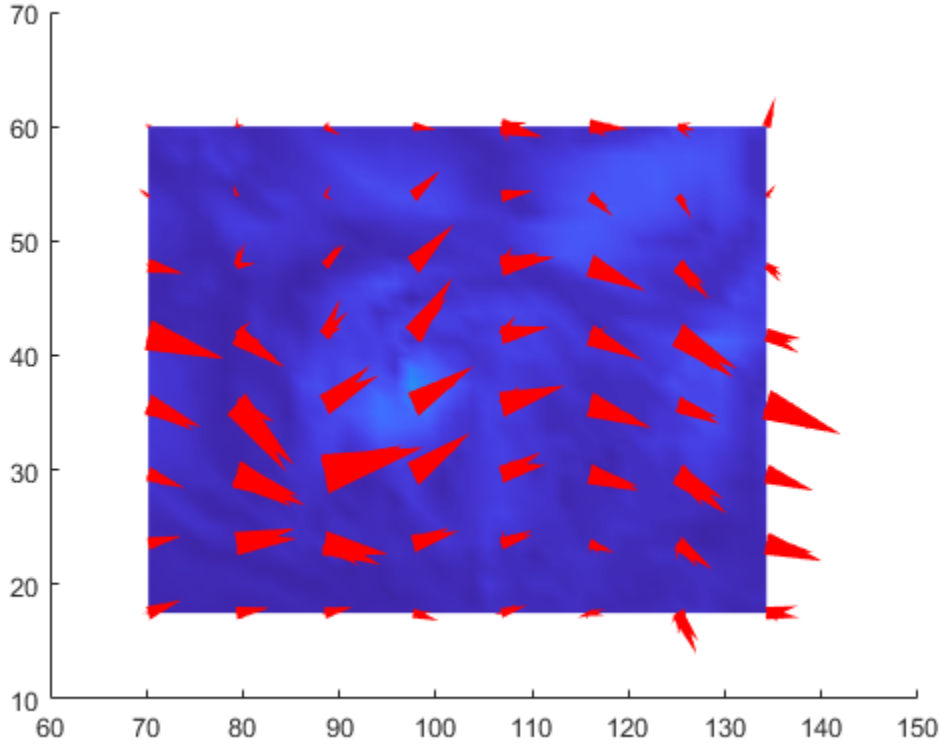


Calculate the magnitude of the vector field (which represents wind speed) to generate scalar data for the `slice` command.

```
hold on
wind_speed = sqrt(u.^2 + v.^2 + w.^2);
```

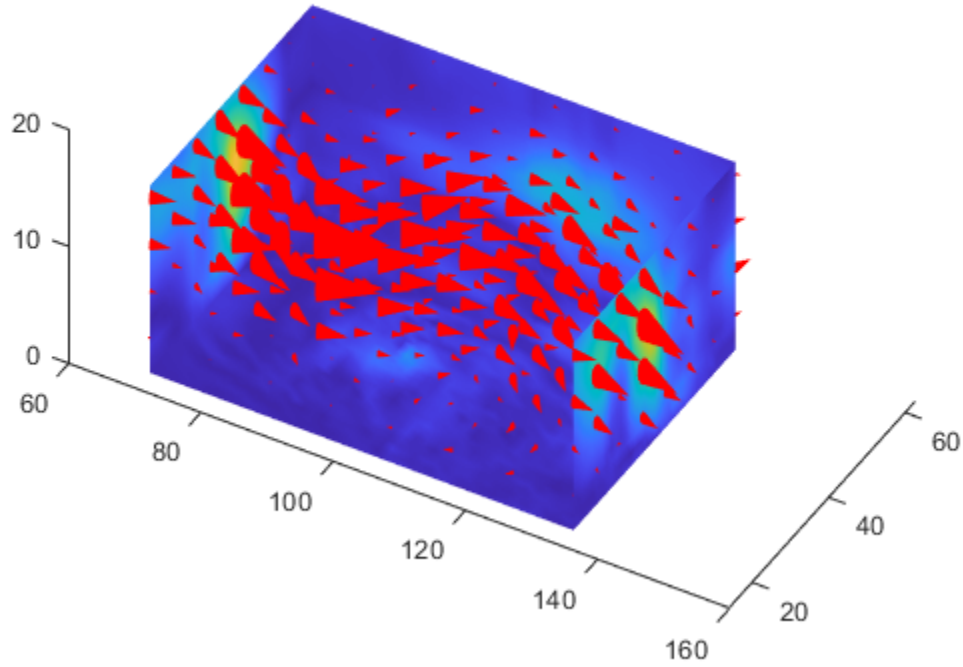
Create slice planes along the *x*-axis at `xmin` and `xmax`, along the *y*-axis at `ymax`, and along the *z*-axis at `zmin`. Specify interpolated face color so the slice coloring indicates wind speed, and do not draw edges.

```
hsurfaces = slice(x,y,z,wind_speed,[xmin,xmax],ymax,zmin);
set(hsurfaces,'FaceColor','interp','EdgeColor','none')
hold off
```



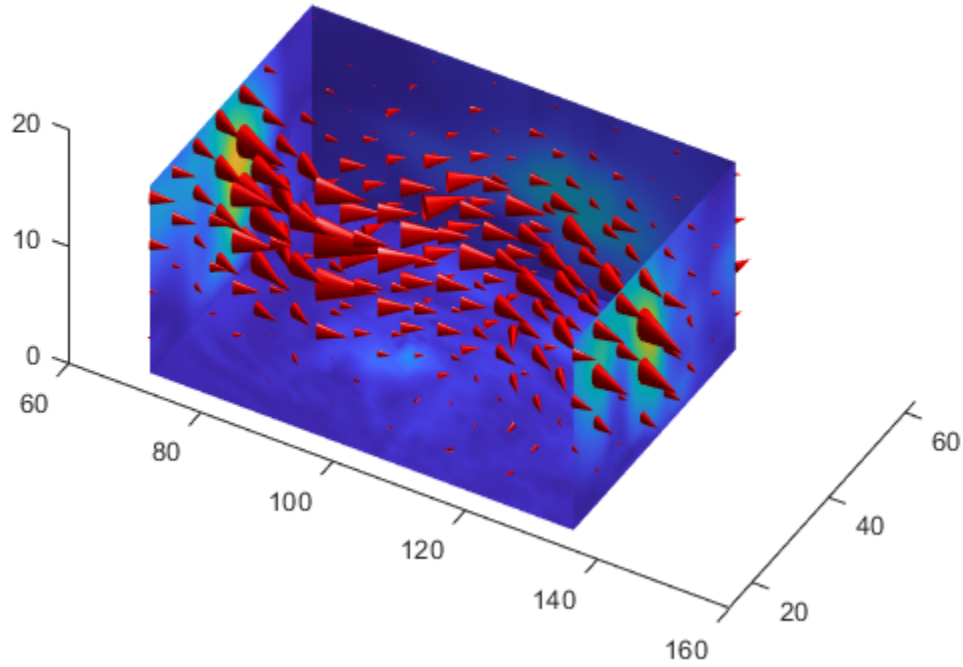
Change the axes view and set the data aspect ratio.

```
view(30,40)
daspect([2,2,1])
```



Add a light source to the right of the camera and use Gouraud lighting to give the cones and slice planes a smooth, three-dimensional appearance.

```
camlight right
lighting gouraud
set(hsurfaces,'AmbientStrength',0.6)
hcone.DiffuseStrength = 0.8;
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`isosurface` | `patch` | `reducevolume` | `smooth3` | `stream2` | `stream3` | `streamline` | `subvolume`

Topics

“Overview of Volume Visualization”

Introduced before R2006a

conj

Complex conjugate

Syntax

```
Zc = conj(Z)
```

Description

`Zc = conj(Z)` returns the complex conjugate of each element in `Z`.

Examples

Find Complex Conjugate of Complex Number

Find the complex conjugate of the complex number `Z`.

```
Z = 2+3i
```

```
Z = 2.0000 + 3.0000i
```

```
Zc = conj(Z)
```

```
Zc = 2.0000 - 3.0000i
```

Find Complex Conjugate of Complex Values in Matrix

Create a 2-by-2 matrix with complex elements.

```
Z = [0-1i 2+1i; 4+2i 0-2i]
```

```
Z = 2×2 complex
```

```
   0.0000 - 1.0000i   2.0000 + 1.0000i
   4.0000 + 2.0000i   0.0000 - 2.0000i
```

Find the complex conjugate of each complex number in matrix Z.

```
Zc = conj(Z)
```

```
Zc = 2×2 complex

   0.0000 + 1.0000i   2.0000 - 1.0000i
   4.0000 - 2.0000i   0.0000 + 2.0000i
```

Input Arguments

Z — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. `conj` operates element-wise when Z is nonscalar.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`complex` | `ctranspose` | `i` | `imag` | `j` | `real`

Introduced before R2006a

contains

Determine if pattern is in strings

Syntax

```
TF = contains(str,pattern)
TF = contains(str,pattern,'IgnoreCase',true)
```

Description

`TF = contains(str,pattern)` returns `1` (`true`) if `str` contains the specified pattern, and returns `0` (`false`) otherwise.

If `pattern` is an array containing multiple patterns, then `contains` returns `1` if it finds any element of `pattern` in `str`.

`TF = contains(str,pattern,'IgnoreCase',true)` ignores case when determining if `str` contains `pattern`.

Examples

Search for Single Pattern

Create a string array that contains names. Determine which strings contain `Paul`.

Starting in R2017a, you can create strings using double quotes.

```
str = ["Mary Ann Jones","Paul Jay Burns","John Paul Smith"]
```

```
str = 1x3 string array
    "Mary Ann Jones"    "Paul Jay Burns"    "John Paul Smith"
```

Return a logical array where the position of each element equal to `1` corresponds to the position of a string in `str` that contains `Paul`.

```
pattern = "Paul";
TF = contains(str,pattern)
```

```
TF = 1x3 logical array

   0   1   1
```

Display the strings that contain `Paul`. Index back into `str` using `TF`.

```
str(TF)
```

```
ans = 1x2 string array
    "Paul Jay Burns"    "John Paul Smith"
```

Search for Multiple Patterns

Create a string array that contains names. Determine which strings contain either `Ann` or `Paul`.

Starting in R2017a, you can create strings using double quotes.

```
str = ["Mary Ann Jones","Christopher Matthew Burns","John Paul Smith"]
```

```
str = 1x3 string array
    "Mary Ann Jones"    "Christopher Matth..."    "John Paul Smith"
```

```
pattern = ["Ann","Paul"];
TF = contains(str,pattern)
```

```
TF = 1x3 logical array

   1   0   1
```

Display the strings that contain either `Ann` or `Paul`. Index back into `str` using `TF`.

```
str(TF)
```

```
ans = 1x2 string array
    "Mary Ann Jones"    "John Paul Smith"
```

Ignore Case

Create a string array that contains names. Determine which names contain `anne`, ignoring case.

Starting in R2017a, you can create strings using double quotes.

```
str = ["Anne","Elizabeth","Marianne","Tracy"]
```

```
str = 1x4 string array
    "Anne"    "Elizabeth"    "Marianne"    "Tracy"
```

```
pattern = "anne";
TF = contains(str,pattern,'IgnoreCase',true)
```

```
TF = 1x4 logical array

   1   0   1   0
```

Display the strings that contain `anne`. Index back into `str` using `TF`.

```
str(TF)
```

```
ans = 1x2 string array
    "Anne"    "Marianne"
```

Determine if Character Vector Contains Pattern

Create a character vector that contains a list of foods. Determine whether the names of different foods are in the character vector.

```
chr = 'peppers, onions, and mushrooms';
TF = contains(chr,'onion')
```

```
TF = logical
   1
```

```
TF = contains(chr,'pineapples')
```

```
TF = logical
   0
```

Input Arguments

`str` — Input text
string array | character vector | cell array of character vectors

Input text, specified as a string array, a character vector, or a cell array of character vectors.

`pattern` — Search pattern
string array | character vector | cell array of character vectors

Search pattern, specified as a string array, a character vector, or a cell array of character vectors.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `str` and `pattern` must be a string scalar, a character vector, or a cell array containing not more than one character vector.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`endsWith` | `find` | `ismember` | `regexp` | `startsWith` | `strcmp` | `strfind`

Topics

“Create String Arrays”
“Search and Replace Text”
“Compare Text”
“Test for Empty Strings and Missing Values”

Introduced in R2016b

continue

Pass control to next iteration of `for` or `while` loop

Syntax

```
continue
```

Description

`continue` passes control to the next iteration of a `for` or `while` loop. It skips any remaining statements in the body of the loop for the current iteration. The program continues execution from the next iteration.

`continue` applies only to the body of the loop where it is called. In nested loops, `continue` skips remaining statements only in the body of the loop in which it occurs.

Examples

Selectively Display Values in Loop

Display the multiples of 7 from 1 through 50. If a number is not divisible by 7, use `continue` to skip the `disp` statement and pass control to the next iteration of the `for` loop.

```
for n = 1:50
    if mod(n,7)
        continue
    end
    disp(['Divisible by 7: ' num2str(n)])
end
```

```
Divisible by 7: 7
Divisible by 7: 14
Divisible by 7: 21
```

```
Divisible by 7: 28
Divisible by 7: 35
Divisible by 7: 42
Divisible by 7: 49
```

Skip to Next Loop Iteration

Count the number of lines of code in the file `magic.m`. Skip blank lines and comments using a `continue` statement. `continue` skips the remaining instructions in the `while` loop and begins the next iteration.

```
fid = fopen('magic.m','r');
count = 0;
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line) || strncmp(line,'%',1) || ~ischar(line)
        continue
    end
    count = count + 1;
end
count
```

```
count = 31
```

```
fclose(fid);
```

Tips

- The `continue` statement skips the rest of the instructions in a `for` or `while` loop and begins the next iteration. To exit the loop completely, use a `break` statement.
- `continue` is not defined outside a `for` or `while` loop. To exit a function, use `return`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`break` | `for` | `while`

Introduced before R2006a

contour

Contour plot of matrix

Syntax

```
contour(Z)
contour(X,Y,Z)
contour( ___ ,levels)
contour( ___ ,LineSpec)
contour( ___ ,Name,Value)
contour(ax, ___ )
M = contour( ___ )
[M,c] = contour( ___ )
```

Description

`contour(Z)` creates a contour plot containing the isolines of matrix `Z`, where `Z` contains height values on the *x*-*y* plane. MATLAB automatically selects the contour lines to display. The column and row indices of `Z` are the *x* and *y* coordinates in the plane, respectively.

`contour(X,Y,Z)` specifies the *x* and *y* coordinates for the values in `Z`.

`contour( ___ ,levels)` specifies the contour lines to display as the last argument in any of the previous syntaxes. Specify `levels` as a scalar value `n` to display the contour lines at `n` automatically chosen levels (heights). To draw the contour lines at specific heights, specify `levels` as a vector of monotonically increasing values. To draw the contours at one height (`k`), specify `levels` as a two-element row vector `[k k]`.

`contour( ___ ,LineSpec)` specifies the style and color of the contour lines.

`contour( ___ ,Name,Value)` specifies additional options for the contour plot using one or more name-value pair arguments. Specify the options after all other input arguments. For a list of properties, see Contour.

`contour(ax, ___ )` displays the contour plot in the target axes. Specify the axes as the first argument in any of the previous syntaxes.

`M = contour( ___ )` returns the contour matrix `M`, which contains the (*x*, *y*) coordinates of the vertices at each level.

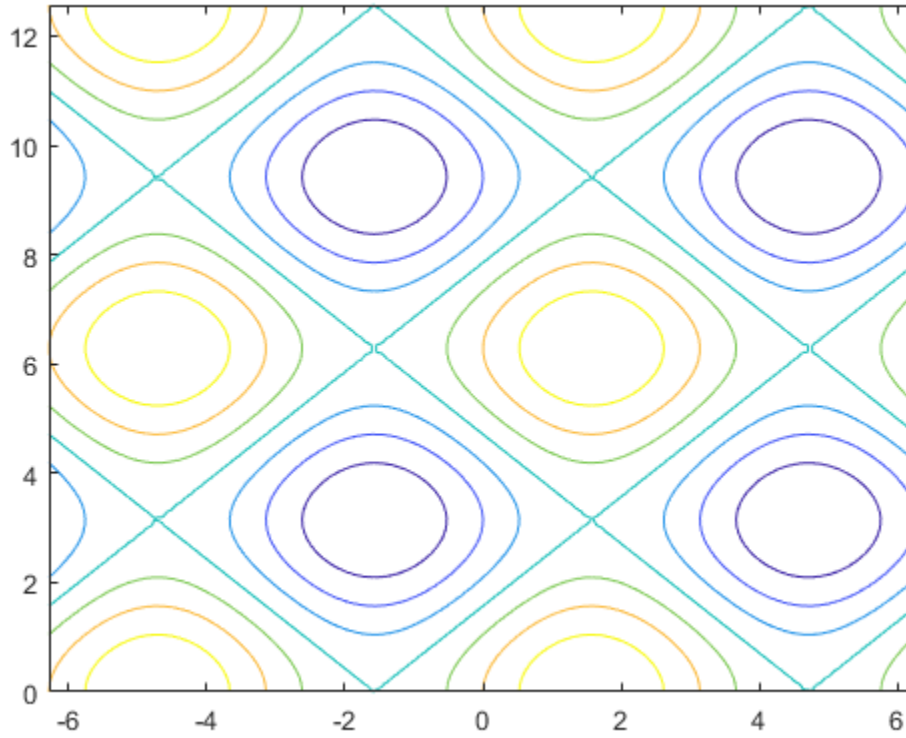
`[M,c] = contour( ___ )` returns the contour matrix and the contour object `c`. Use `c` to set properties after displaying the contour plot.

Examples

Contours of a Function

Create matrices `X` and `Y`, that define a grid in the *x*-*y* plane. Define matrix `Z` as the heights above that grid. Then plot the contours of `Z`.

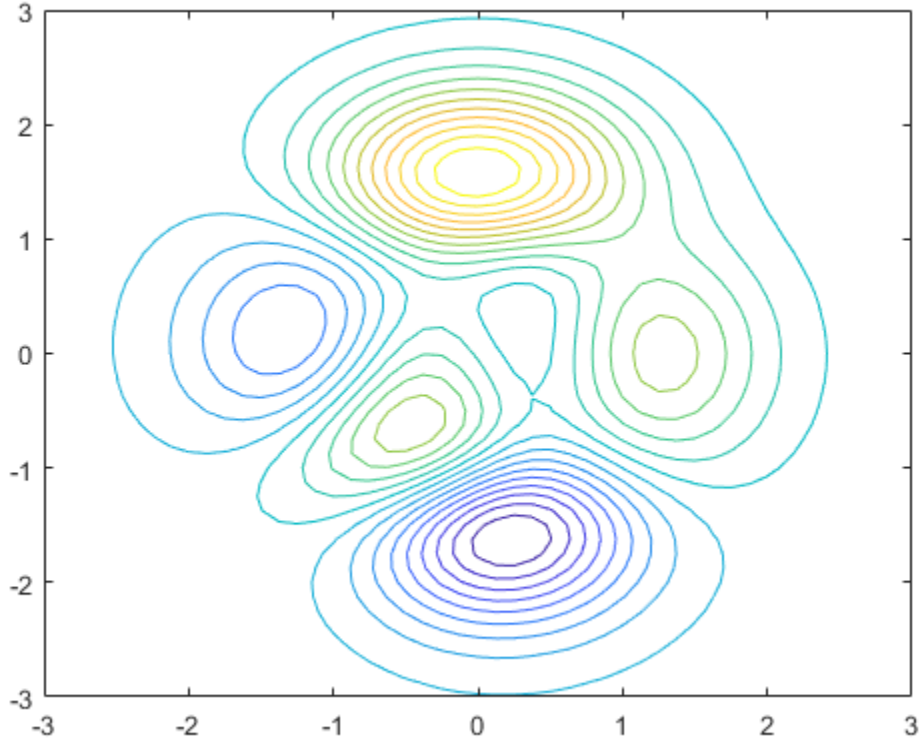
```
x = linspace(-2*pi,2*pi);
y = linspace(0,4*pi);
[X,Y] = meshgrid(x,y);
Z = sin(X)+cos(Y);
contour(X,Y,Z)
```



Contours at Twenty Levels

Define Z as a function of X and Y. In this case, call the `peaks` function to create X, Y, and Z. Then plot 20 contours of Z.

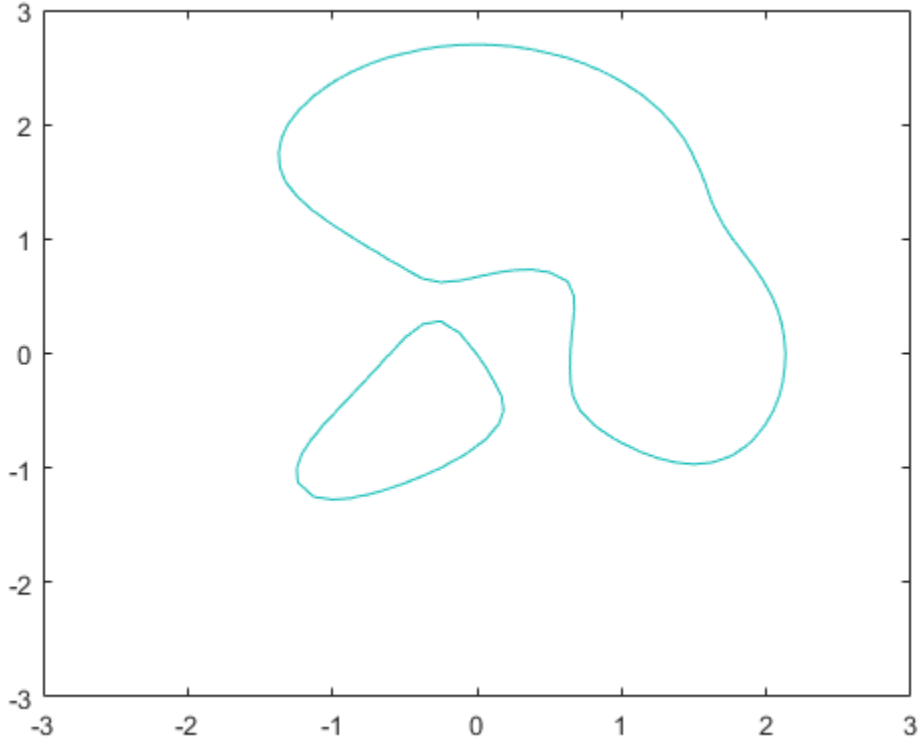
```
[X,Y,Z] = peaks;
contour(X,Y,Z,20)
```



Contours at One Level

Display the contours of the `peaks` function at `Z = 1`.

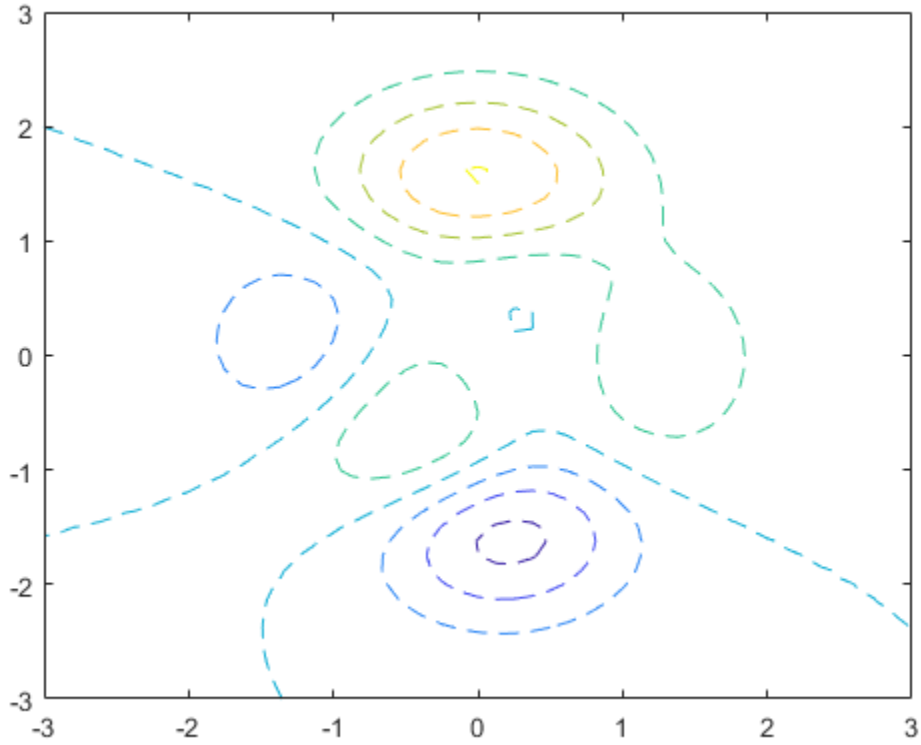
```
[X,Y,Z] = peaks;
v = [1,1];
contour(X,Y,Z,v)
```



Dashed Contour Lines

Create a contour plot of the `peaks` function, and specify the dashed line style.

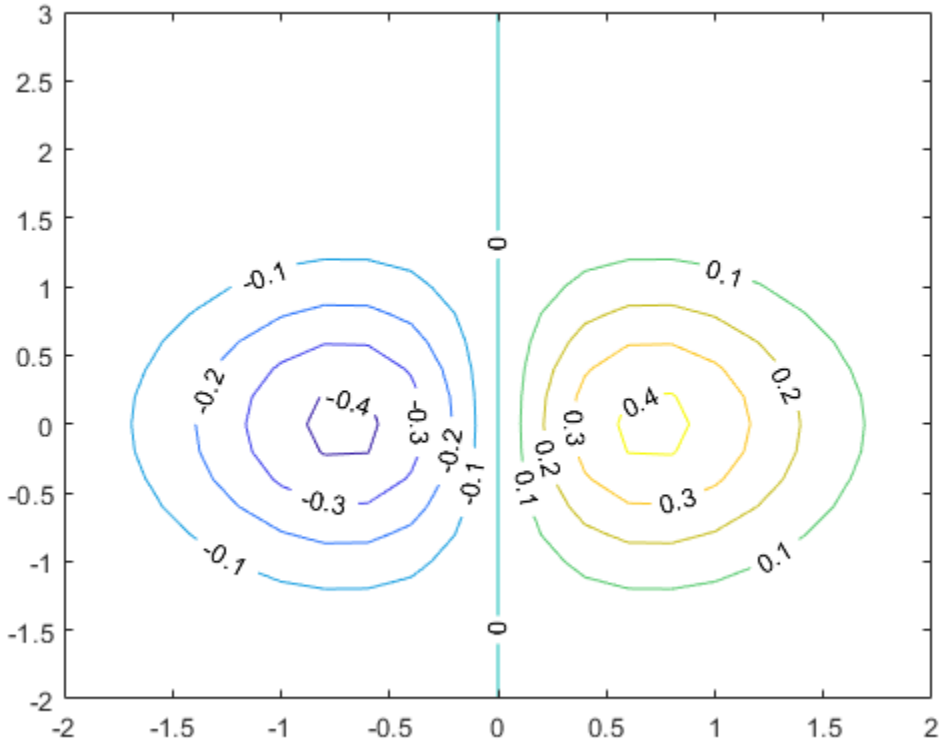
```
[X,Y,Z] = peaks;
contour(X,Y,Z,'--')
```



Contours with Labels

Define Z as a function of two variables, X and Y. Then create a contour plot of that function, and display the labels by setting the `ShowText` property to `'on'`.

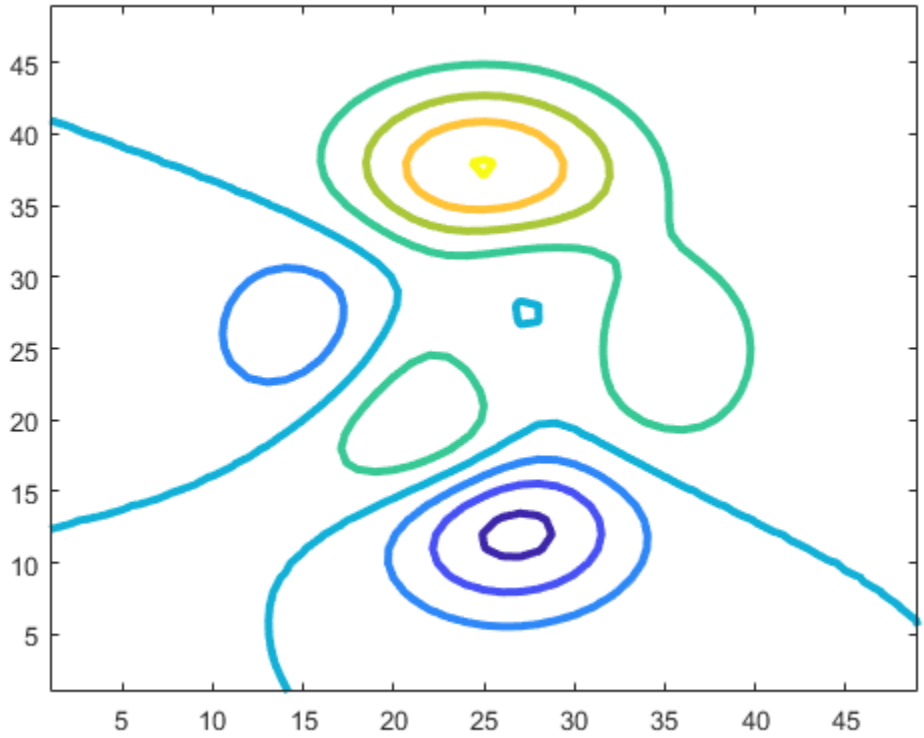
```
x = -2:0.2:2;
y = -2:0.2:3;
[X,Y] = meshgrid(x,y);
Z = X.*exp(-X.^2-Y.^2);
contour(X,Y,Z,'ShowText','on')
```



Custom Line Width

Create a contour plot of the `peaks` function. Make the contour lines thicker by setting the `LineWidth` property to 3.

```
Z = peaks;
[M,c] = contour(Z);
c.LineWidth = 3;
```

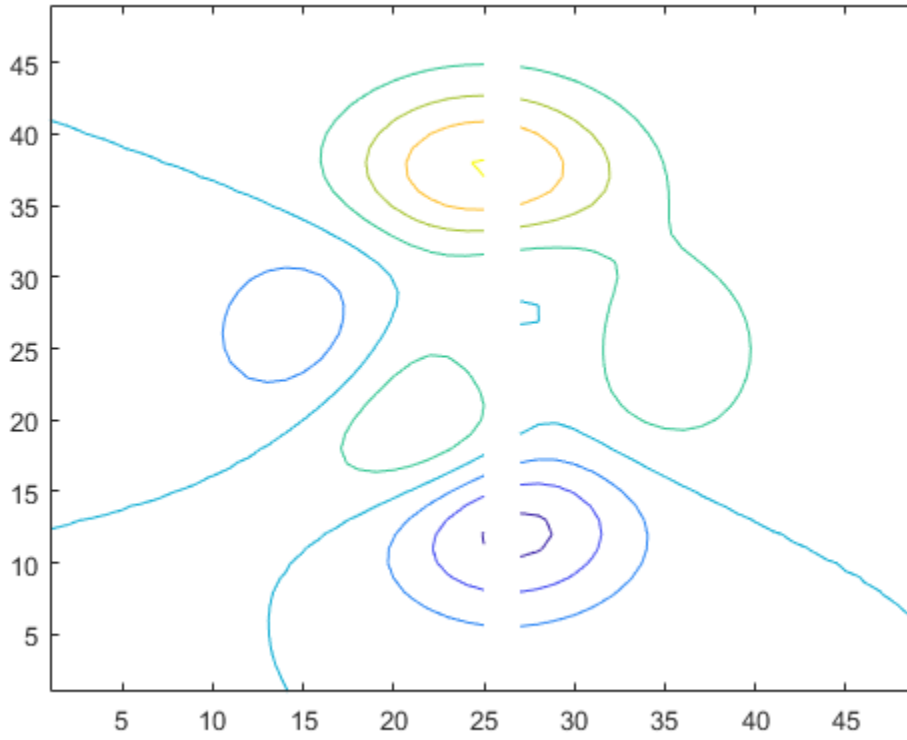


Contours Over Discontinuous Surface

Insert `NaN` values wherever there are discontinuities on a surface. The `contour` function does not draw contour lines in those regions.

Define matrix `Z` as a sampling of the `peaks` function. Replace all values in column `26` with `NaN` values. Then plot the contours of the modified `Z` matrix.

```
Z = peaks;
Z(:,26) = NaN;
contour(Z)
```



Input Arguments

X — *x*-coordinates
matrix | vector

x-coordinates, specified as a matrix the same size as Z, or as a vector with length n, where `[m,n] = size(Z)`. The default value of X is the vector `(1:n)`.

When X is a matrix, the values must be strictly increasing or decreasing along one dimension and remain constant along the other dimension. The dimension that varies

must be the opposite of the dimension that varies in `Y`. You can use the `meshgrid` function to create `X` and `Y` matrices.

When `X` is a vector, the values must be strictly increasing or decreasing.

Example: `X = 1:10`

Example: `X = [1 2 3; 1 2 3; 1 2 3]`

Example: `[X,Y] = meshgrid(1:10)`

The `XData` property of the `Contour` object stores the *x*-coordinates.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Y` — *y*-coordinates
matrix | vector

y-coordinates, specified as a matrix the same size as `Z`, or as a vector with length `m`, where `[m,n] = size(Z)`. The default value of `Y` is the vector `(1:m)`.

When `Y` is a matrix, the values must be strictly increasing or decreasing along one dimension and remain constant along the other dimension. The dimension that varies must be the opposite of the dimension that varies in `X`. You can use the `meshgrid` function to create the `X` and `Y` matrices.

When `Y` is a vector, the values must be strictly increasing or decreasing.

Example: `Y = 1:10`

Example: `Y = [1 1 1; 2 2 2; 3 3 3]`

Example: `[X,Y] = meshgrid(1:10)`

The `YData` property of the `Contour` object stores the *y*-coordinates.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Z` — *z*-coordinates
matrix

z-coordinates, specified as a matrix. This matrix must have at least two rows and two columns, and it must contain at least two different values.

Example: `Z = peaks(20)`

The `ZData` property of the `Contour` object stores the *z*-coordinates.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`levels` — Levels

scalar | vector

Contour levels, specified as a scalar whole number or a vector. Use this argument to control the number and location of the contour lines. When you do not specify the levels, the `contour` function chooses the levels automatically.

- To draw contour lines at n automatically chosen heights, specify levels as the scalar value n.
- To draw the contour lines at specific heights, specify `levels` as a vector of monotonically increasing values.
- To draw contour lines at a single height `k`, specify `levels` as a two-element row vector `[k k]`.

Example: `contour(peaks,10)` draws contour lines at 10 automatically chosen heights on the `peaks` function.

Example: `contour(peaks,[-4 0 4])` draws contour lines at 3 specific heights on the `peaks` function: `-4`, `0`, and `4`.

Example: `contour(peaks,[3 3])` draws contour lines to show where the height of the `peaks` function is 3.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`LineSpec` — Line style and color

character vector | string scalar

Line style and color, specified as a character vector or string scalar containing line style symbols, color options, or both. The line style symbols are listed in the following table, and they can appear in any order. Marker symbols such as `'o'` are ignored.

You do not need to specify both the line style and the color. For example, if you omit the line style and specify the color, then the plot shows solid lines using the specified color.

Line Style	Description	Resulting Line
`-`	Solid line (default)	
`--`	Dashed line	
`:`	Dotted line	
`-.`	Dash-dot line	

This table lists the available color options.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`

ax — Target axes
`Axes` object

Target axes, specified as an `Axes` object. If you do not specify the axes, then `contour` plots into the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `contour(Z,'ShowText','on')` displays the contour line labels.

Note The properties listed here are only a subset. For a complete list, see Contour.

`ShowText` — Contour line labels
`'off'` (default) | `'on'`

Contour line labels, specified as one of these values:

- `'off'` — Do not label the contour lines.
- `'on'` — Display the height values along the contour lines.

`LineWidth` — Line Width
`0.5` (default) | positive value

Contour line width, specified as a positive value in points. One point equals 1/72 inch.

`LabelSpacing` — Label spacing
144 (default) | scalar

Label spacing along the contour lines, specified as a scalar value in points, where one point is 1/72 inch. Use this property to control the number of contour labels along the contour lines. Smaller values produce more labels.

You must set the `ShowText` property to `'on'` for the `LabelSpacing` property to have an effect.

If you use the `clabel` function to display the labels, then the `LabelSpacing` property has no effect and the plot displays one label per line.

Output Arguments

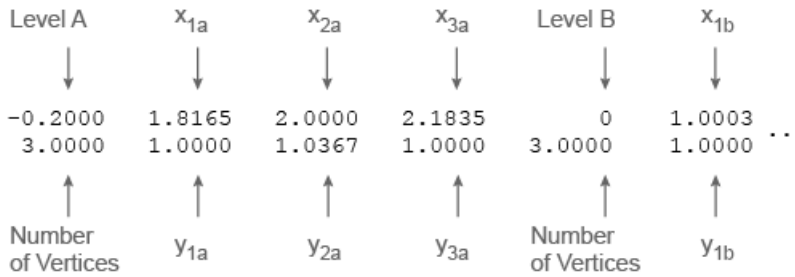
`M` — Contour matrix
matrix

Contour matrix, returned as two-row matrix. This matrix contains the contour levels (heights) and the coordinates of the vertices at each level. The data is arranged sequentially in n sets of columns for n contour lines:

- The first column in each set contains the contour level and the number of vertices at that level. The top number is the contour level, and the bottom number is the number of vertices.

- Subsequent columns in the set are the (x, y) coordinates of the vertices. Each column represents an ordered pair. The top number is the x-coordinate, and the bottom number is the y-coordinate.

For example, here are the first few columns of the contour matrix `M = contour(peaks(3))`:



The `ContourMatrix` property of the `Contour` object stores the contour matrix.

c — Contour object
`Contour` object

`Contour` object. Use this object to set properties after displaying the contour plot.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions
`clabel` | `contour3` | `contourc` | `contourf`

Properties
Contour

Topics

“Highlight Specific Contour Levels”

Introduced before R2006a

contour3

3-D contour plot

Syntax

```
contour3(Z)
contour3(X,Y,Z)
contour3( ___ ,levels)
contour3( ___ ,LineSpec)
contour3( ___ ,Name,Value)
contour3(ax, ___ )
M = contour3( ___ )
[M,c] = contour3( ___ )
```

Description

`contour3(Z)` creates a 3-D contour plot containing the isolines of matrix `Z`, where `Z` contains height values on the *x*-*y* plane. MATLAB automatically selects the contour lines to display. The column and row indices of `Z` are the *x* and *y* coordinates in the plane, respectively.

`contour3(X,Y,Z)` specifies the *x* and *y* coordinates for the values in `Z`.

`contour3( ___ ,levels)` specifies the contour lines to display as the last argument in any of the previous syntaxes. Specify `levels` as a scalar value `n` to display the contour lines at `n` automatically chosen levels (heights). To draw the contour lines at specific heights, specify `levels` as a vector of monotonically increasing values. To draw the contours at one height (`k`), specify `levels` as a two-element row vector `[k k]`.

`contour3( ___ ,LineSpec)` specifies the style and color of the contour lines.

`contour3( ___ ,Name,Value)` specifies additional options for the contour plot using one or more name-value pair arguments. Specify the options after all other input arguments. For a list of properties, see Contour.

`contour3(ax, ___ )` displays the contour plot in the target axes. Specify the axes as the first argument in any of the previous syntaxes.

`M = contour3( ___ )` returns the contour matrix `M`, which contains the (*x*, *y*) coordinates of the vertices at each level.

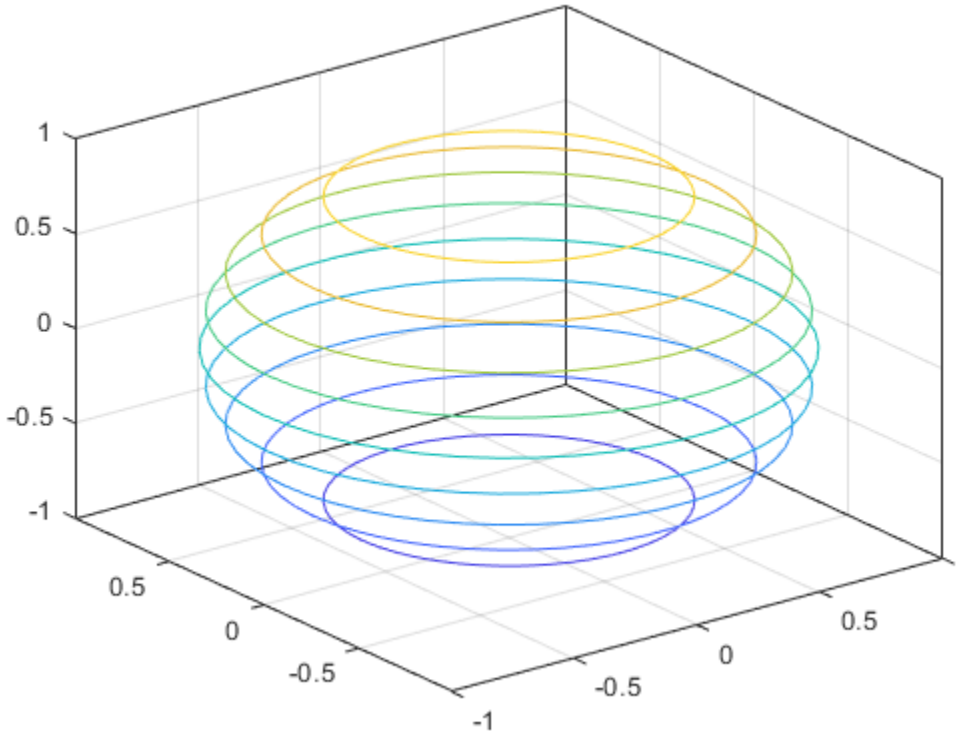
`[M,c] = contour3( ___ )` returns the contour matrix and the contour object `c`. Use `c` to set properties after displaying the contour plot.

Examples

Contours of Sphere

Define `Z` as a function of `X` and `Y`. In this case, call the `sphere` function to create `X`, `Y`, and `Z`. Then plot the contours of `Z`.

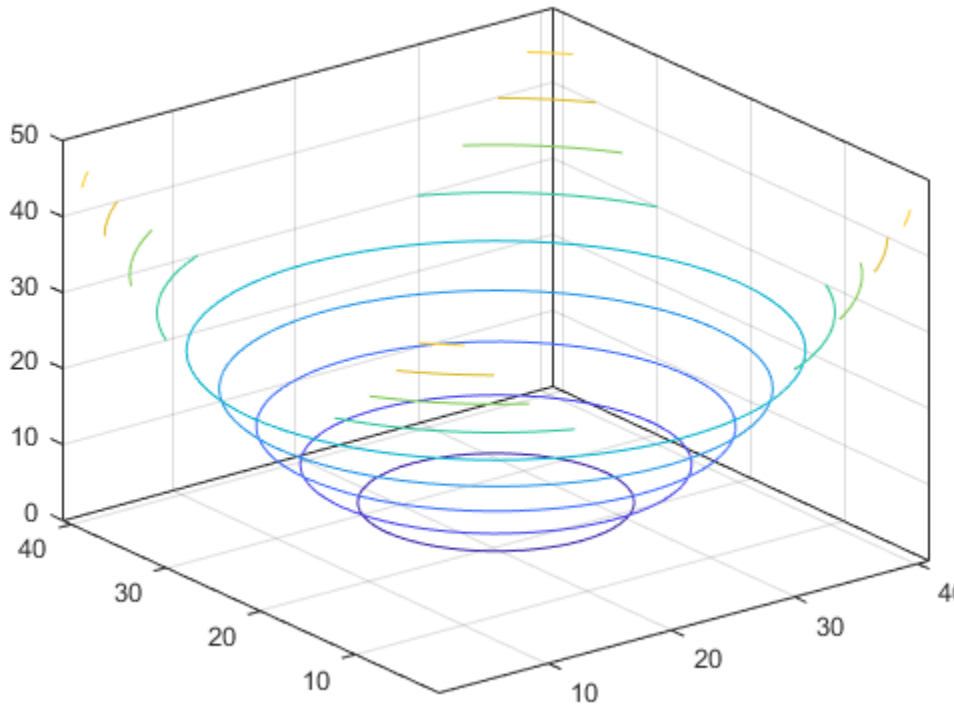
```
[X,Y,Z] = sphere(50);
contour3(X,Y,Z);
```



Contours at Fifty Levels

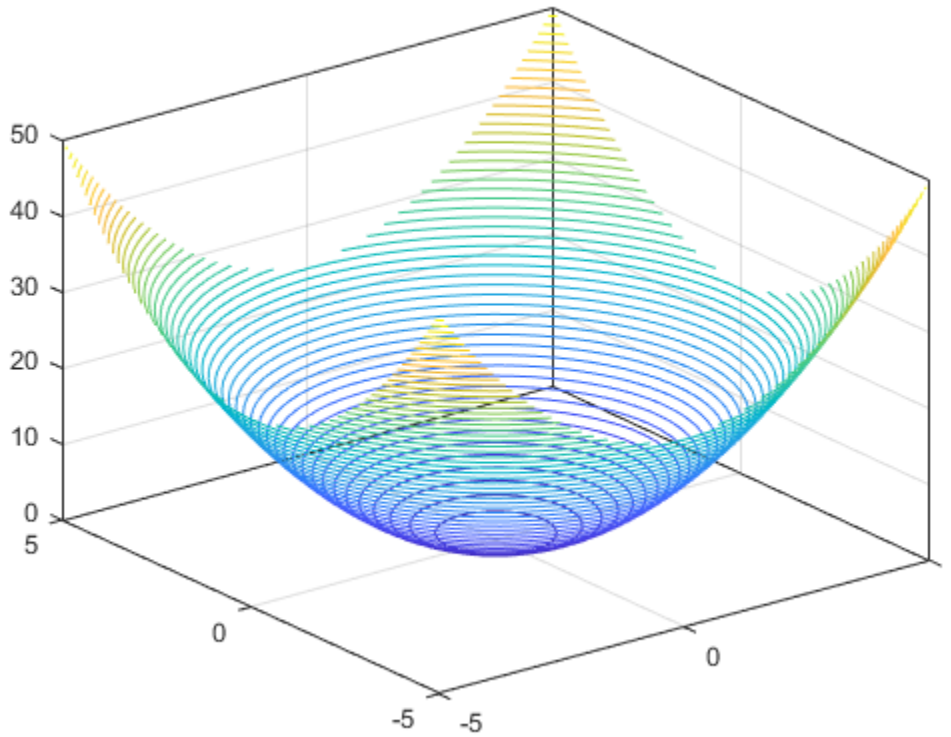
Define Z as a function of two variables, X and Y. Then plot the contours of Z. In this case, let MATLAB® choose the contours and the limits for the *x*- and *y*-axes.

```
[X,Y] = meshgrid(-5:0.25:5);
Z = X.^2 + Y.^2;
contour3(Z)
```



Now specify `50` contour levels, and display the results within the *x* and *y* limits used to calculate `Z`.

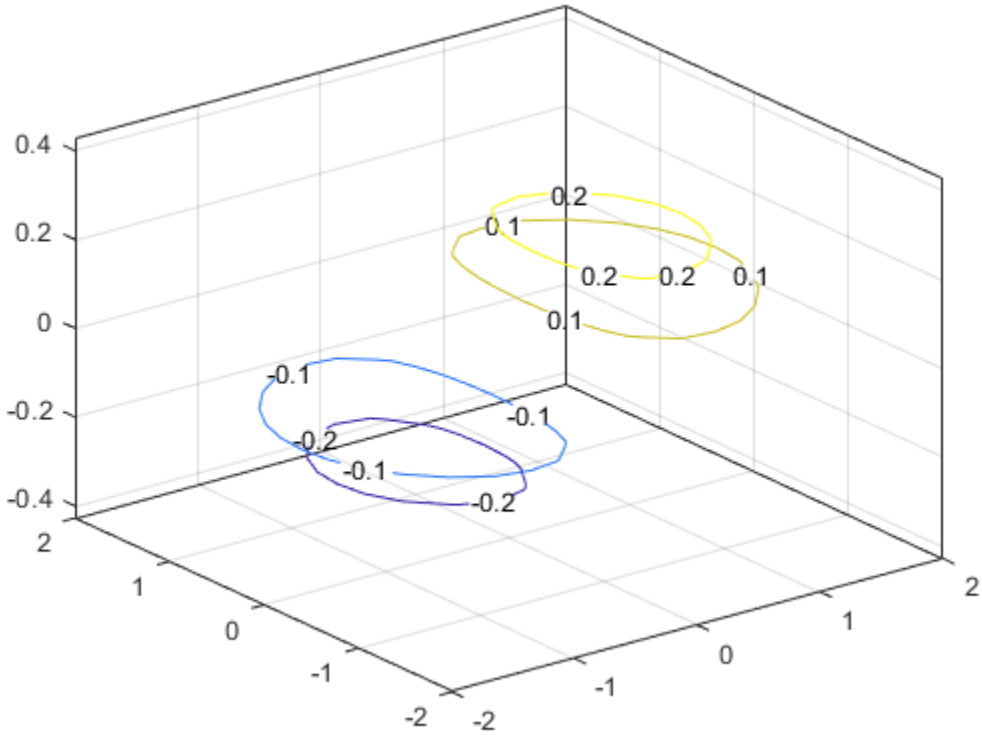
```
contour3(X,Y,Z,50)
```



Contours at Specific Levels with Labels

Define `Z` as a function of two variables, `X` and `Y`. Then plot the contours at `Z = [-.2 -.1 .1 .2]`. Show the contour labels by setting the `ShowText` property to `'on'`.

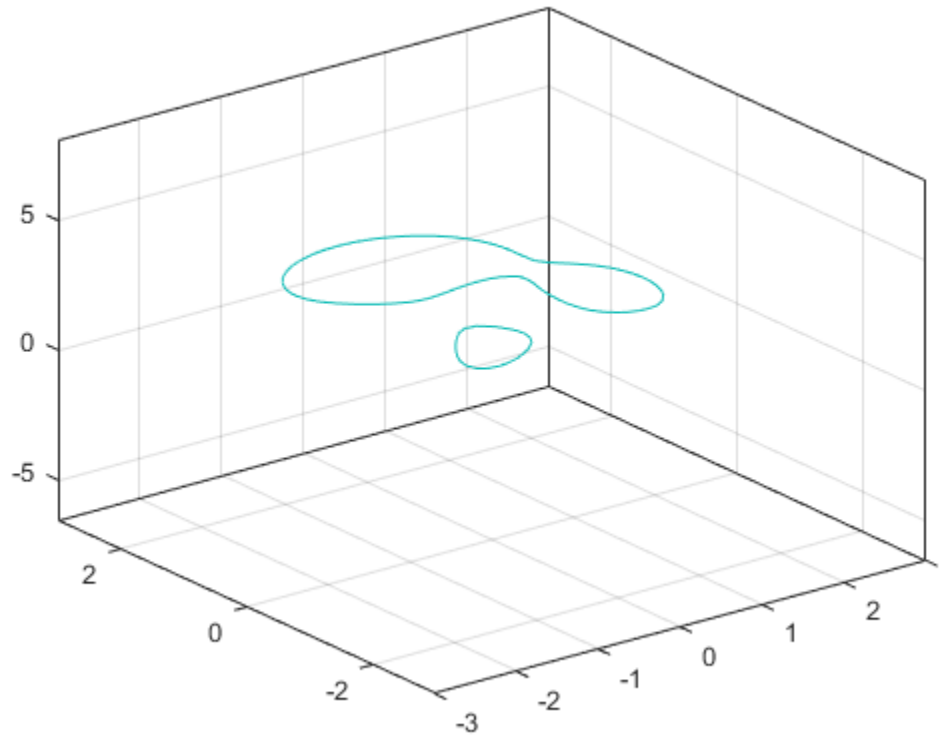
```
[X,Y] = meshgrid(-2:0.25:2);
Z = X.*exp(-X.^2-Y.^2);
contour3(X,Y,Z,[-.2 -.1 .1 .2],'ShowText','on')
```



Contours at One Level

Define Z as a function of X and Y. In this case, call the `peaks` function to create X, Y, and Z. Then display the contours at `Z = 2`.

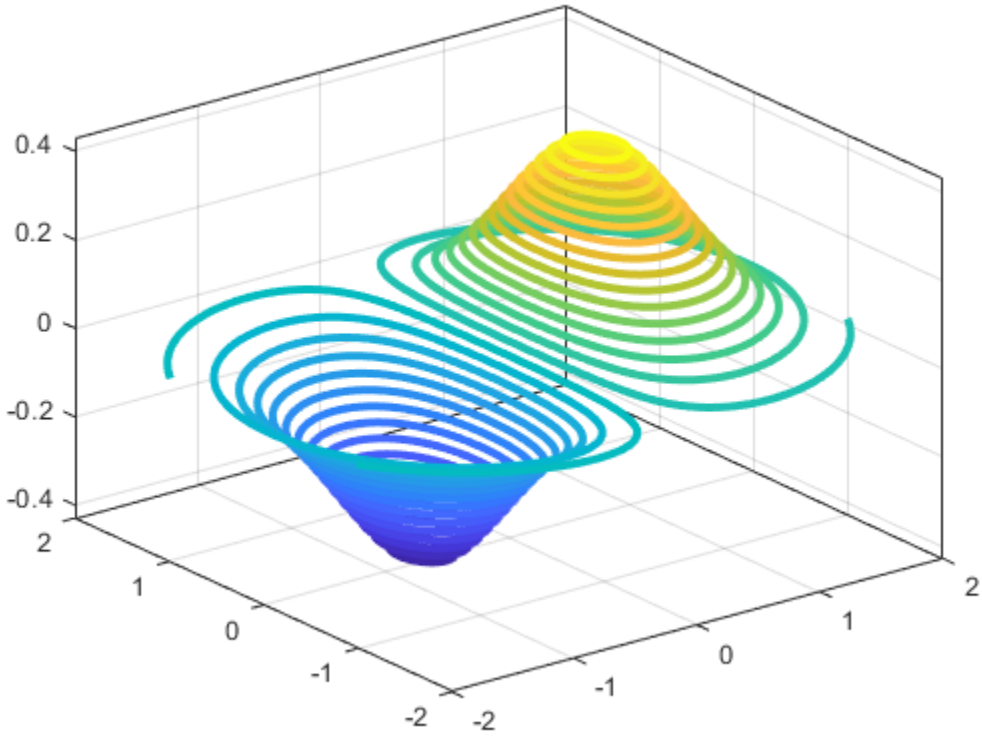
```
[X,Y,Z] = peaks;
contour3(X,Y,Z,[2 2]);
```



Custom Line Width

Define Z as a function of two variables, X and Y. Plot 30 contours of Z, and then set the line width to 3.

```
[X,Y] = meshgrid(-2:0.0125:2);
Z = X.*exp(-X.^2-Y.^2);
[M,c] = contour3(X,Y,Z,30);
c.LineWidth = 3;
```



Input Arguments

X — *x*-coordinates
matrix | vector

x-coordinates, specified as a matrix the same size as Z, or as a vector with length n, where `[m,n] = size(Z)`. The default value of X is the vector `(1:n)`.

When X is a matrix, the values must be strictly increasing or decreasing along one dimension and remain constant along the other dimension. The dimension that varies

must be the opposite of the dimension that varies in `Y`. You can use the `meshgrid` function to create `X` and `Y` matrices.

When `X` is a vector, the values must be strictly increasing or decreasing.

Example: `X = 1:10`

Example: `X = [1 2 3; 1 2 3; 1 2 3]`

Example: `[X,Y] = meshgrid(1:10)`

The `XData` property of the `Contour` object stores the *x*-coordinates.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Y` — *y*-coordinates
matrix | vector

y-coordinates, specified as a matrix the same size as `Z`, or as a vector with length `m`, where `[m,n] = size(Z)`. The default value of `Y` is the vector `(1:m)`.

When `Y` is a matrix, the values must be strictly increasing or decreasing along one dimension and remain constant along the other dimension. The dimension that varies must be the opposite of the dimension that varies in `X`. You can use the `meshgrid` function to create the `X` and `Y` matrices.

When `Y` is a vector, the values must be strictly increasing or decreasing.

Example: `Y = 1:10`

Example: `Y = [1 1 1; 2 2 2; 3 3 3]`

Example: `[X,Y] = meshgrid(1:10)`

The `YData` property of the `Contour` object stores the *y*-coordinates.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Z` — *z*-coordinates
matrix

z-coordinates, specified as a matrix. This matrix must have at least two rows and two columns, and it must contain at least two different values.

Example: `Z = peaks(20)`

The `ZData` property of the `Contour` object stores the *z*-coordinates.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`levels` — Levels

scalar | vector

Contour levels, specified as a scalar whole number or a vector. Use this argument to control the number and location of the contour lines. When you do not specify the levels, the `contour3` function chooses the levels automatically.

- To draw contour lines at n automatically chosen heights, specify `levels` as the scalar value n.
- To draw the contour lines at specific heights, specify `levels` as a vector of monotonically increasing values.
- To draw contour lines at a single height `k`, specify `levels` as a two-element row vector `[k k]`.

Example: `contour3(peaks,10)` draws contour lines at 10 automatically chosen heights on the `peaks` function.

Example: `contour3(peaks,[-4 0 4])` draws contour lines at 3 specific heights on the `peaks` function: `-4`, `0`, and `4`.

Example: `contour3(peaks,[3 3])` draws contour lines to show where the height of the `peaks` function is 3.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`LineSpec` — Line style and color

character vector | string scalar

Line style and color, specified as a character vector or string scalar containing line style symbols, color options, or both. The line style symbols are listed in the following table, and they can appear in any order. Marker symbols such as `'o'` are ignored.

You do not need to specify both the line style and the color. For example, if you omit the line style and specify the color, then the plot shows solid lines using the specified color.

Line Style	Description	Resulting Line
`-`	Solid line (default)	
`--`	Dashed line	
`:`	Dotted line	
`-.`	Dash-dot line	

This table lists the available color options.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`

ax — Target axes
`Axes` object

Target axes, specified as an `Axes` object. If you do not specify the axes, then `contour3` plots into the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `contour3(Z,'ShowText','on')` displays the contour line labels.

Note The properties listed here are only a subset. For a complete list, see Contour.

`ShowText` — Contour line labels
`'off'` (default) | `'on'`

Contour line labels, specified as one of these values:

- `'off'` — Do not label the contour lines.
- `'on'` — Display the height values along the contour lines.

`LineWidth` — Line Width
`0.5` (default) | positive value

Contour line width, specified as a positive value in points. One point equals 1/72 inch.

`LabelSpacing` — Label spacing
144 (default) | scalar

Label spacing along the contour lines, specified as a scalar value in points, where one point is 1/72 inch. Use this property to control the number of contour labels along the contour lines. Smaller values produce more labels.

You must set the `ShowText` property to `'on'` for the `LabelSpacing` property to have an effect.

If you use the `clabel` function to display the labels, then the `LabelSpacing` property has no effect and the plot displays one label per line.

Output Arguments

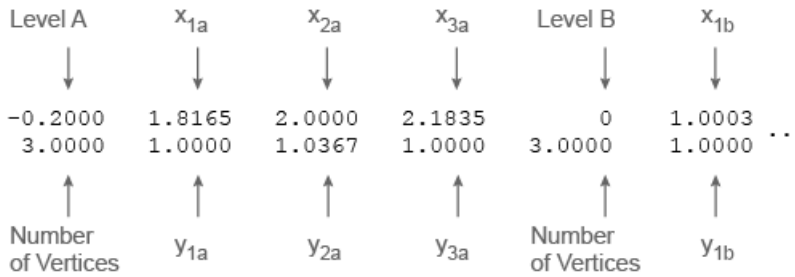
`M` — Contour matrix
matrix

Contour matrix, returned as two-row matrix. This matrix contains the contour levels (heights) and the coordinates of the vertices at each level. The data is arranged sequentially in n sets of columns for n contour lines:

- The first column in each set contains the contour level and the number of vertices at that level. The top number is the contour level, and the bottom number is the number of vertices.

- Subsequent columns in the set are the (x, y) coordinates of the vertices. Each column represents an ordered pair. The top number is the x-coordinate, and the bottom number is the y-coordinate.

For example, here are the first few columns of the contour matrix `M = contour(peaks(3))`:



The `ContourMatrix` property of the `Contour` object stores the contour matrix.

c — Contour object
`Contour` object

`Contour` object. Use this object to set properties after displaying the contour plot.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions
`clabel` | `contour` | `contourc` | `contourf`

Properties
Contour

Introduced before R2006a

contourc

Low-level contour plot computation

Syntax

```
C = contourc(Z)
C = contourc(Z,n)
C = contourc(Z,v)
C = contourc(x,y,Z)
C = contourc(x,y,Z,n)
C = contourc(x,y,Z,v)
```

Description

`contourc` calculates the contour matrix `C` used by `contour`, `contour3`, and `contourf`. The values in `Z` determine the heights of the contour lines with respect to a plane. The contour calculations use a regularly spaced grid determined by the dimensions of `Z`.

`C = contourc(Z)` computes the contour matrix from data in matrix `Z`, where `Z` must be at least a 2-by-2 matrix. The contours are isolines in the units of `Z`. The number of contour lines and the corresponding values of the contour lines are chosen automatically.

`C = contourc(Z,n)` computes contours of matrix `Z` with `n` contour levels.

`C = contourc(Z,v)` computes contours of matrix `Z` with contour lines at the values specified in vector `v`. The length of `v` determines the number of contour levels. To compute a single contour of level `k`, use `contourc(Z,[k k])`.

`C = contourc(x,y,Z)`, `C = contourc(x,y,Z,n)`, and `C = contourc(x,y,Z,v)` compute contours of `Z` using vectors `x` and `y` to determine the *x* and *y* values. `x` and `y` must be monotonically increasing.

Tips

For more information on the contour matrix, see the `ContourMatrix` property for contour objects.

Specifying irregularly spaced `x` and `y` vectors is not the same as contouring irregularly spaced data. If `x` or `y` is irregularly spaced, `contourc` calculates contours using a regularly spaced contour grid, then transforms the data to `x` or `y`.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`clabel` | `contour` | `contour3` | `contourf`

Introduced before R2006a

contourf

Filled 2-D contour plot

Syntax

```
contourf(Z)
contourf(X,Y,Z)
contourf( ___ ,levels)
contourf( ___ ,LineSpec)
contourf( ___ ,Name,Value)
contourf(ax, ___ )
M = contourf( ___ )
[M,c] = contourf( ___ )
```

Description

`contourf(Z)` creates a filled contour plot containing the isolines of matrix `Z`, where `Z` contains height values on the *x*-*y* plane. MATLAB automatically selects the contour lines to display. The column and row indices of `Z` are the *x* and *y* coordinates in the plane, respectively.

`contourf(X,Y,Z)` specifies the *x* and *y* coordinates for the values in `Z`.

`contourf( ___ ,levels)` specifies the contour lines to display as the last argument in any of the previous syntaxes. Specify `levels` as a scalar value `n` to display the contour lines at `n` automatically chosen levels (heights). To draw the contour lines at specific heights, specify `levels` as a vector of monotonically increasing values. To draw the contours at one height (`k`), specify `levels` as a two-element row vector `[k k]`.

`contourf( ___ ,LineSpec)` specifies the style and color of the contour lines.

`contourf( ___ ,Name,Value)` specifies additional options for the contour plot using one or more name-value pair arguments. Specify the options after all other input arguments. For a list of properties, see Contour.

`contourf(ax, ___ )` displays the contour plot in the target axes. Specify the axes as the first argument in any of the previous syntaxes.

`M = contourf( ___ )` returns the contour matrix `M`, which contains the (*x*, *y*) coordinates of the vertices at each level.

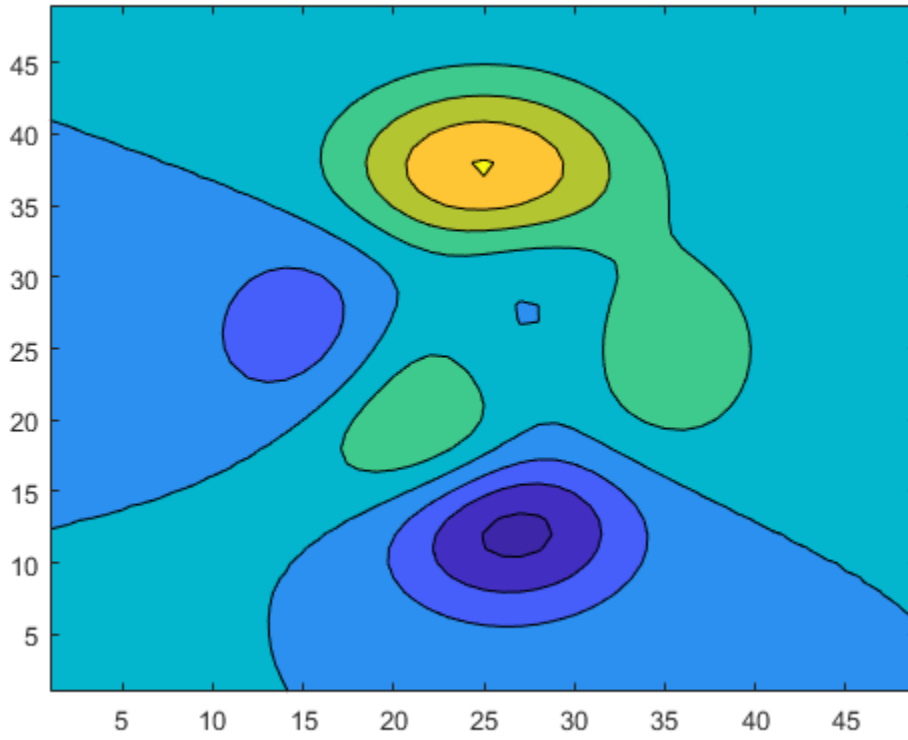
`[M,c] = contourf( ___ )` returns the contour matrix and the contour object `c`. Use `c` to set properties after displaying the contour plot.

Examples

Contours of Peaks Function

Define `Z` as a function of two variables. In this case, call the `peaks` function to create `Z`. Then display a filled contour plot of `Z`, letting MATLAB® choose the contour levels.

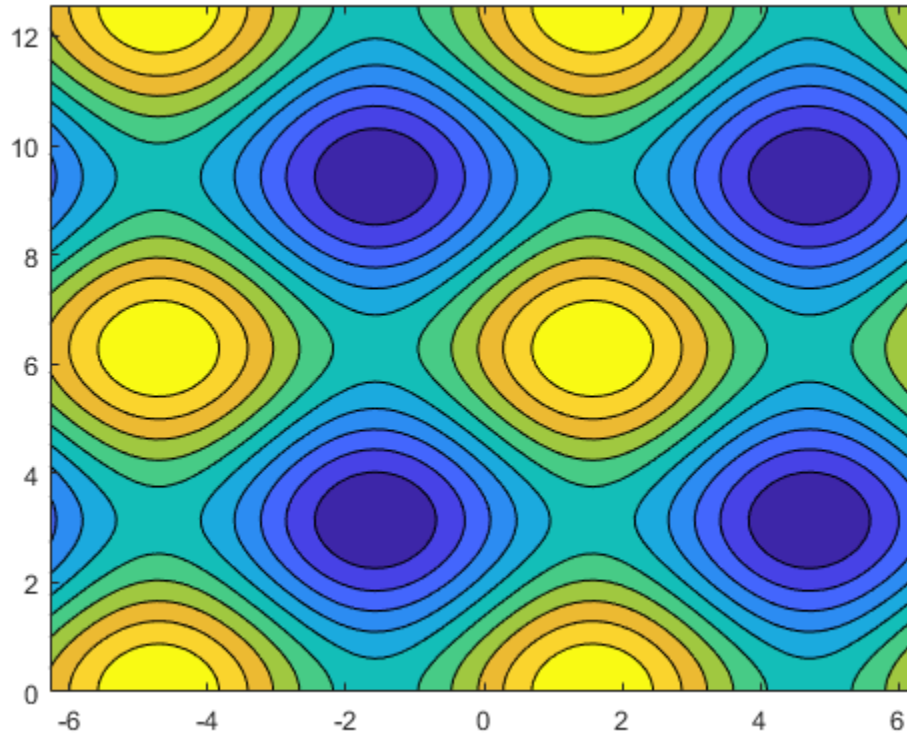
```
Z = peaks;
contourf(Z)
```



Contours at Ten Levels

Define Z as a function of two variables, X and Y. Then display contours at 10 levels of Z.

```
x = linspace(-2*pi,2*pi);
y = linspace(0,4*pi);
[X,Y] = meshgrid(x,y);
Z = sin(X) + cos(Y);
contourf(X,Y,Z,10)
```

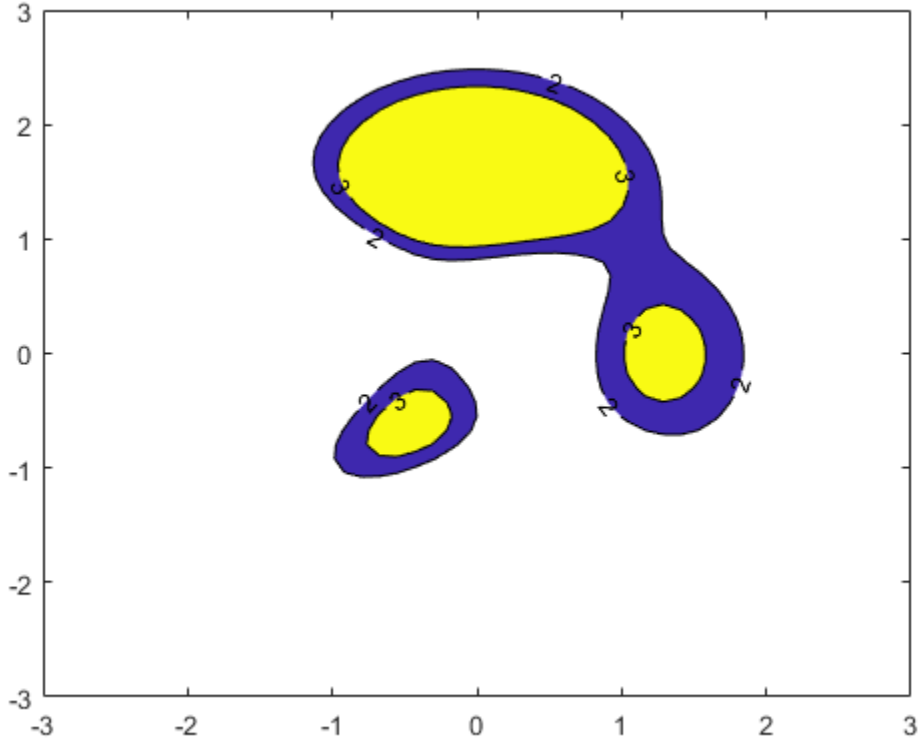


Contours at Specific Levels with Labels

Define Z as a function of X and Y. In this case, call the `peaks` function to create X, Y, and Z. Then display contours at levels 2 and 3.

The white region corresponds to the heights less than 2. The purple region corresponds to heights between 2 and 3. And the yellow region corresponds to heights that are greater than 3.

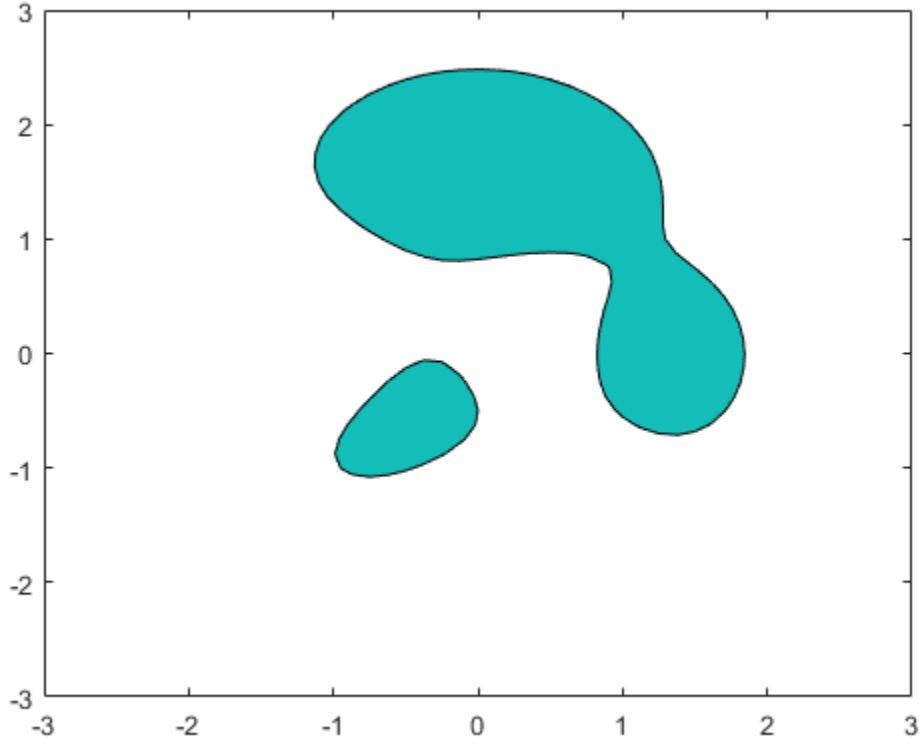
```
[X,Y,Z] = peaks(50);
contourf(X,Y,Z,[2 3],'ShowText','on')
```



Contours at One Level

Define Z as a function of X and Y. In this case, call the `peaks` function to create X, Y, and Z. Then display contours at `Z = 2`.

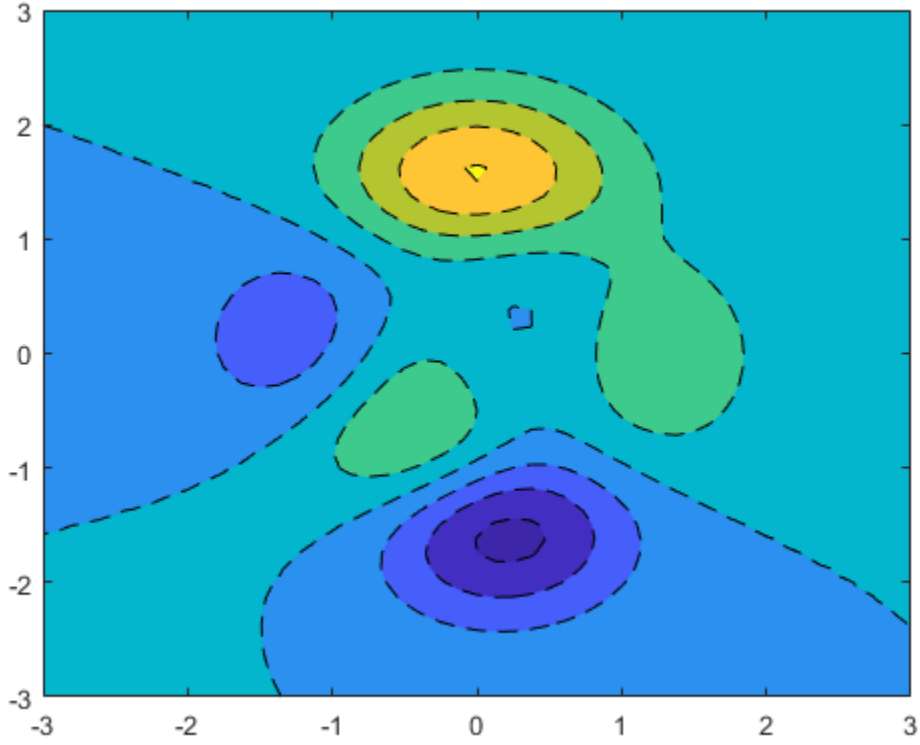
```
[X,Y,Z] = peaks;
contourf(X,Y,Z,[2 2])
```



Dashed Contour Lines

Create a contour plot, and specify the dashed line style.

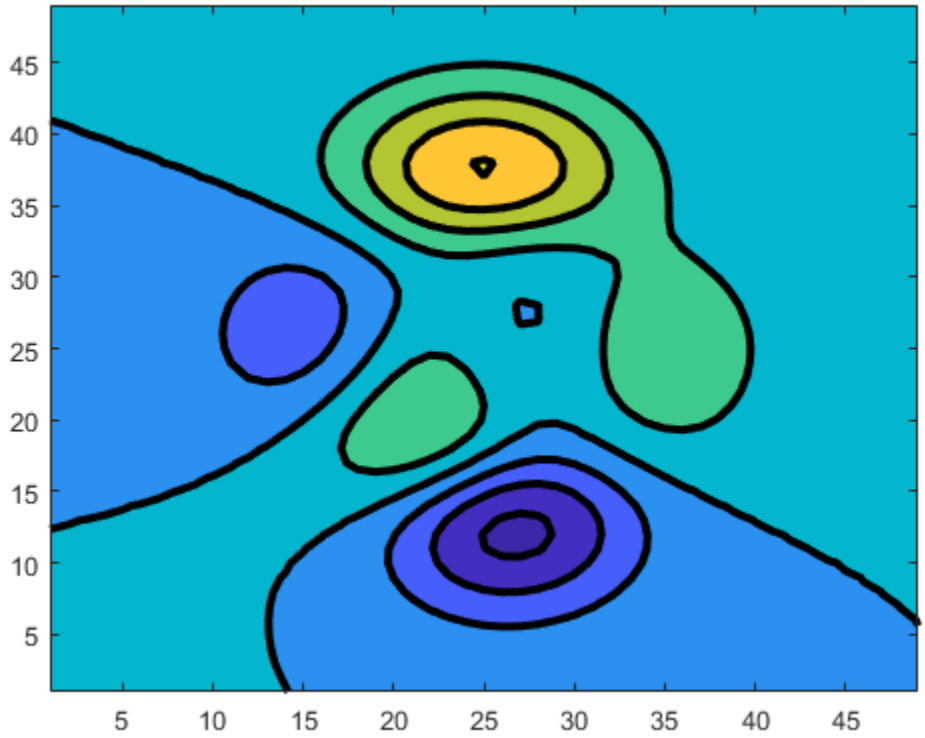
```
[X,Y,Z] = peaks;
contourf(X,Y,Z,'--')
```



Custom Line Width

Create a filled contour plot. Make the contour lines thicker by setting the `LineWidth` property to 3.

```
Z = peaks;
[M,c] = contourf(Z);
c.LineWidth = 3;
```

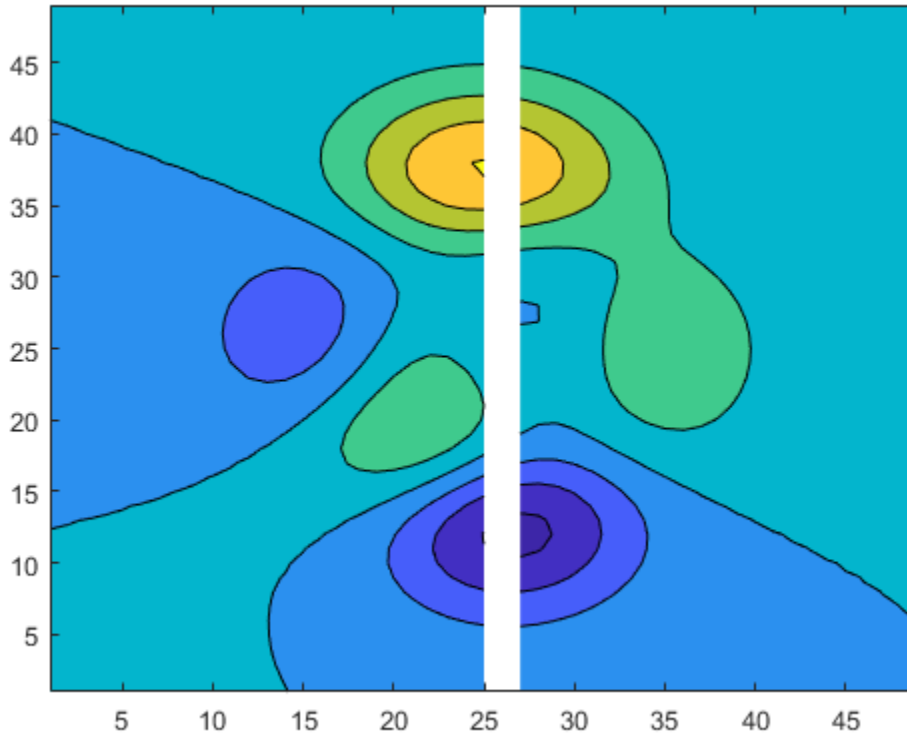


Contours Over Discontinuous Surface

Insert `NaN` values wherever there are discontinuities on a surface. The `contourf` function does not draw contour lines in those regions.

Define matrix `Z` as a sampling of the `peaks` function. Replace all values in column `26` with `NaN` values. Then plot the contours of the modified `Z` matrix.

```
Z = peaks;
Z(:,26) = NaN;
contourf(Z)
```



Input Arguments

X — *x*-coordinates
matrix | vector

x-coordinates, specified as a matrix the same size as Z, or as a vector with length n, where `[m,n] = size(Z)`. The default value of X is the vector `(1:n)`.

When X is a matrix, the values must be strictly increasing or decreasing along one dimension and remain constant along the other dimension. The dimension that varies

must be the opposite of the dimension that varies in `Y`. You can use the `meshgrid` function to create `X` and `Y` matrices.

When `X` is a vector, the values must be strictly increasing or decreasing.

Example: `X = 1:10`

Example: `X = [1 2 3; 1 2 3; 1 2 3]`

Example: `[X,Y] = meshgrid(1:10)`

The `XData` property of the `Contour` object stores the *x*-coordinates.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Y` — *y*-coordinates
matrix | vector

y-coordinates, specified as a matrix the same size as `Z`, or as a vector with length `m`, where `[m,n] = size(Z)`. The default value of `Y` is the vector `(1:m)`.

When `Y` is a matrix, the values must be strictly increasing or decreasing along one dimension and remain constant along the other dimension. The dimension that varies must be the opposite of the dimension that varies in `X`. You can use the `meshgrid` function to create the `X` and `Y` matrices.

When `Y` is a vector, the values must be strictly increasing or decreasing.

Example: `Y = 1:10`

Example: `Y = [1 1 1; 2 2 2; 3 3 3]`

Example: `[X,Y] = meshgrid(1:10)`

The `YData` property of the `Contour` object stores the *y*-coordinates.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Z` — *z*-coordinates
matrix

z-coordinates, specified as a matrix. This matrix must have at least two rows and two columns, and it must contain at least two different values.

Example: `Z = peaks(20)`

The `ZData` property of the `Contour` object stores the *z*-coordinates.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`levels` — Levels
scalar | vector

Contour levels, specified as a scalar whole number or a vector. Use this argument to control the number and location of the contour lines. When you do not specify the levels, the `contourf` function chooses the levels automatically.

- To draw contour lines at n automatically chosen heights, specify `levels` as the scalar value n.
- To draw the contour lines at specific heights, specify `levels` as a vector of monotonically increasing values.
- To draw contour lines at a single height `k`, specify `levels` as a two-element row vector `[k k]`.

The `contourf` function uses the current colormap to fill the spaces between the levels in the plot. The first color fills the space between the lowest level and the level above it. The last color corresponds to Z-values that are greater than the highest level in the plot. If Z contains values that are smaller than the lowest level displayed in the plot, the region between the lowest level and the smallest Z-value is white.

Example: `contourf(peaks,10)` draws contour lines at 10 automatically chosen heights on the `peaks` function.

Example: `contourf(peaks,[-4 0 4])` draws contour lines at 3 specific heights on the `peaks` function: `-4`, `0`, and `4`.

Example: `contourf(peaks,[3 3])` draws contour lines to show where the height of the `peaks` function is 3.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`LineSpec` — Line style and color
character vector | string scalar

Line style and color, specified as a character vector or string scalar containing line style symbols, color options, or both. The line style symbols are listed in the following table, and they can appear in any order. Marker symbols such as `'o'` are ignored.

You do not need to specify both the line style and the color. For example, if you omit the line style and specify the color, then the plot shows solid lines using the specified color.

Line Style	Description	Resulting Line
`-`	Solid line (default)	
`--`	Dashed line	
`:`	Dotted line	
`-.`	Dash-dot line	

This table lists the available color options.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`

ax — Target axes

`Axes` object

Target axes, specified as an `Axes` object. If you do not specify the axes, then `contourf` plots into the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `contourf(Z,'ShowText','on')` displays the contour line labels.

Note The properties listed here are only a subset. For a complete list, see Contour.

`ShowText` — Contour line labels
`'off'` (default) | `'on'`

Contour line labels, specified as one of these values:

- `'off'` — Do not label the contour lines.
- `'on'` — Display the height values along the contour lines.

`LineWidth` — Line Width
`0.5` (default) | positive value

Contour line width, specified as a positive value in points. One point equals 1/72 inch.

`LabelSpacing` — Label spacing
`144` (default) | scalar

Label spacing along the contour lines, specified as a scalar value in points, where one point is 1/72 inch. Use this property to control the number of contour labels along the contour lines. Smaller values produce more labels.

You must set the `ShowText` property to `'on'` for the `LabelSpacing` property to have an effect.

If you use the `clabel` function to display the labels, then the `LabelSpacing` property has no effect and the plot displays one label per line.

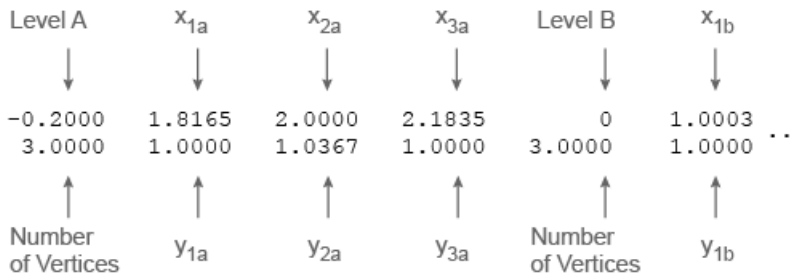
Output Arguments

`M` — Contour matrix
matrix

Contour matrix, returned as two-row matrix. This matrix contains the contour levels (heights) and the coordinates of the vertices at each level. The data is arranged sequentially in n sets of columns for n contour lines:

- The first column in each set contains the contour level and the number of vertices at that level. The top number is the contour level, and the bottom number is the number of vertices.
- Subsequent columns in the set are the (*x*, *y*) coordinates of the vertices. Each column represents an ordered pair. The top number is the *x*-coordinate, and the bottom number is the *y*-coordinate.

For example, here are the first few columns of the contour matrix `M = contour(peaks(3))`:



The `ContourMatrix` property of the `Contour` object stores the contour matrix.

`c` — Contour object
`Contour` object

`Contour` object. Use this object to set properties after displaying the contour plot.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions

`clabel` | `contour` | `contour3` | `contourc`

Properties

Contour

Topics

“Change Fill Colors for Contour Plot”

Introduced before R2006a

Contour Properties

Contour chart appearance and behavior

Description

`Contour` properties control the appearance and behavior of `Contour` objects. By changing property values, you can modify certain aspects of the contour chart.

Starting in R2014b, you can use dot notation to query and set properties.

```
[C,h] = contour(...);
w = h.LineWidth;
h.LineWidth = 2;
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Levels

`LevelList` — Contour levels
empty matrix (default) | vector of z values

Contour levels, specified as a vector of z values. By default, the `contour` function chooses values that span the range of values in the `ZData` property.

Setting this property sets the associated mode property to manual.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`LevelListMode` — Selection mode for `LevelList`
`'auto'` (default) | `'manual'`

Selection mode for the `LevelList`, specified as one of these values:

- `'auto'` — Determine the values based on the `ZData` values.

- `'manual'` — Use manually specified values. To specify the values, set the `LevelList` property. When the mode is manual, the `contour` function does not change the values as you change `ZData`.

`LevelStep` — Spacing between contour lines

`0` (default) | scalar numeric value

Spacing between contour lines, specified as a scalar numeric value. For example, specify a value of 2 to draw contour lines at increments of 2. The `contour` function determines the contour interval based on the `ZData` values.

Setting this property sets the associated mode property to manual.

Example: `3.4`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`LevelStepMode` — Selection mode for `LevelStep`

`'auto'` (default) | `'manual'`

Selection mode for the `LevelStep`, specified as one of these values:

- `'auto'` — Determine the value based on the `ZData` values.
- `'manual'` — Use a manually specified value. To specify the value, set the `LevelStep` property. When the mode is manual, the `contour` function does not change the value as you change `ZData`.

Color and Styling

`Fill` — Fill between contour lines

`'off'` (default) | `'on'`

Fill between contour lines, specified as one of these values:

- `'off'` — Do not fill the spaces between contour lines with a color. This is the default value when you create the contour chart using the `contour` or `contour3` functions.
- `'on'` — Fill the spaces between contour lines with color. This is the default value when you create the contour chart using the `contourf` function.

`LineColor` — Color of contour lines

`'flat'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of contour lines, specified as `'flat'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. To use a different color for each contour line, specify `'flat'`. The colors are determined by the contour value of the line, the colormap, and the scaling of data values into the colormap. For more information on color scaling, see `caxis`.

To use the same color for all contour lines, specify an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line Width

`0.5` (default) | positive value

Contour line width, specified as a positive value in points. One point equals 1/72 inch.

Labels

`ShowText` — Contour line labels

`'off'` (default) | `'on'`

Contour line labels, specified as one of these values:

- `'off'` — Do not label the contour lines.

- `'on'` — Display the height values along the contour lines.

`LabelSpacing` — Label spacing
144 (default) | scalar

Label spacing along the contour lines, specified as a scalar value in points, where one point is 1/72 inch. Use this property to control the number of contour labels along the contour lines. Smaller values produce more labels.

You must set the `ShowText` property to `'on'` for the `LabelSpacing` property to have an effect.

If you use the `clabel` function to display the labels, then the `LabelSpacing` property has no effect and the plot displays one label per line.

`TextStep` — Interval between labeled contour lines
0 (default) | scalar numeric value

Interval between labeled contour lines, specified as a scalar numeric value. By default, the contour plot includes a label for every contour line when the `ShowText` property is set to `'on'`.

Setting this property sets the associated mode property to manual.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`TextStepMode` — Selection mode for `TextStep`
`'auto'` (default) | `'manual'`

Selection mode for the `TextStep`, specified as one of these values:

- `'auto'` — Determine value based on the `ZData` values. If the `ShowText` property is set to `'on'`, then the `contour` function labels every contour line.
- `'manual'` — Use a manually specified value. To specify the value, set the `TextStep` property.

`TextList` — Contour lines to label
empty matrix (default) | vector of real values

Contour lines to label, specified as a vector of real values.

Setting this property sets the associated mode property to manual.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`TextListMode` — Selection mode for `TextList`

`'auto'` (default) | `'manual'`

Selection mode for the `TextList`, specified as one of these values:

- `'auto'` — Use values equal to the values of the `LevelList` property. The contour plot includes a text label for each line.
- `'manual'` — Use manually specified values. Specify the values by setting the `TextList` property.

Data

`ContourMatrix` — Contour matrix

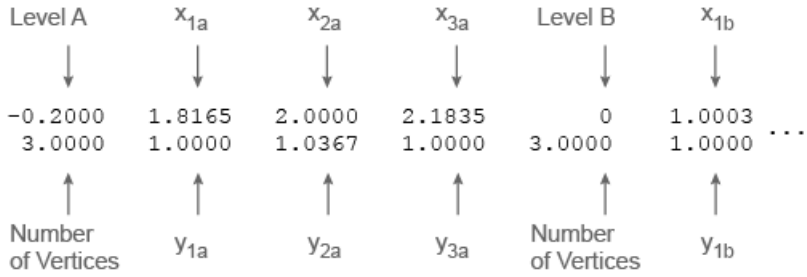
`[ ]` (default) | matrix

This property is read-only.

Contour matrix, returned as two-row matrix. This matrix contains the contour levels (heights) and the coordinates of the vertices at each level. The data is arranged sequentially in n sets of columns for n contour lines:

- The first column in each set contains the contour level and the number of vertices at that level. The top number is the contour level, and the bottom number is the number of vertices.
- Subsequent columns in the set are the (x, y) coordinates of the vertices. Each column represents an ordered pair. The top number is the x-coordinate, and the bottom number is the y-coordinate.

For example, here are the first few columns of the contour matrix `M = contour(peaks(3))`:



XData — x values
[ ] (default) | vector or matrix

x values, specified as a vector or matrix.

- If `XData` is a vector, then `length(XData)` must equal `size(ZData,2)` and `YData` must also be a vector. The `XData` values must be strictly increasing or strictly decreasing and cannot contain any duplicates.
- If `XData` is a matrix, then `size(XData)` and `size(YData)` must equal `size(ZData)`. Typically, you should set the `XData` values so that the columns are strictly increasing or strictly decreasing and the rows are uniform (or the rows are strictly increasing or strictly decreasing and the columns are uniform).

Setting this property sets the associated mode property to manual.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

XDataMode — Selection mode for XData
`'auto'` (default) | `'manual'`

Selection mode for the `XData`, specified as one of these values:

- `'auto'` — Set the `XData` using the column indices of `ZData`.
- `'manual'` — Use manually specified values. To specify the values, set the `XData` property directly, or specify the input argument `X` to the `contour`, `contourf`, or `contour3` function.

XDataSource — Variable linked to XData
`''` (default) | character vector | string

Variable linked to `XData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `XData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `XData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'x'`

`YData` — y values
`[ ]` (default) | vector or matrix

y values, specified as a vector or matrix.

- If `YData` is a vector, then `length(YData)` must equal `size(ZData,1)` and `XData` must also be a vector. The `XData` values must be strictly increasing or strictly decreasing and cannot contain any duplicates.
- If `YData` is a matrix, then `size(XData)` and `size(YData)` must equal `size(ZData)`. Typically, you should set the `YData` values so that the columns are strictly increasing or strictly decreasing and the rows are uniform (or the rows are strictly increasing or strictly decreasing and the columns are uniform).

Setting this property sets the associated mode property to manual.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`YDataMode` — Selection mode for `XData`
`'auto'` (default) | `'manual'`

Selection mode for the `YData`, specified as one of these values:

- `'auto'` — Set the `YData` using the row indices of `ZData`.
- `'manual'` — Use manually specified values. To specify the values, set the `YData` property directly, or specify the input argument `Y` to the `contour`, `contourf`, or `contour3` function.

YDataSource — Variable linked to YData

'' (default) | character vector | string

Variable linked to YData, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the YData.

By default, there is no linked variable so the value is an empty character vector, ''. If you link a variable, then MATLAB does not update the YData values immediately. To force an update of the data values, use the refreshdata function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: 'y'

ZData — Data that defines surface to contour

[ ] (default) | matrix

Data that defines the surface to contour, specified as a matrix. ZData must be at least a 2-by-2 matrix.

Setting this property sets the associated mode property to manual.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

ZDataSource — Variable linked to ZData

'' (default) | character vector | string

Variable linked to ZData, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the ZData.

By default, there is no linked variable so the value is an empty character vector, ''. If you link a variable, then MATLAB does not update the ZData values immediately. To force an update of the data values, use the refreshdata function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'z'`

Legend

`DisplayName` — Legend label

`''` (default) | character vector | string scalar

Legend label, specified as a character vector or string scalar. The legend does not display until you call the `legend` command. If you do not specify the text, then `legend` sets the label using the form `'dataN'`.

`Annotation` — Control for including or excluding object from legend

`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`DataTipTemplate` — Data tip content
`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

`UIContextMenu` — Context menu
empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state
`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

`Clipping` — Clipping of object to axes limits

`'on'` (default) | `'off'`

Clipping of the object to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the object that are outside the axes limits.
- `'off'` — Display the entire object, even if parts of it appear outside the axes limits. Parts of the object might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the object so that it is larger than the original plot.

The `Clipping` property of the axes that contains the object must be set to `'on'`. Otherwise, this property has no effect. For more information about the clipping behavior, see the `Clipping` property of the axes.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.

- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.

- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
    - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback.

MATLAB resumes executing the running callback when the interrupting callback completes.

- If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.

- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks only when visible. The `Visible` property must be set to `'on'`. The `HitTest` property determines if the `Contour` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `Contour` object passes the click to the object behind it in the current view of the figure window. The `HitTest` property of the `Contour` object has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `Contour` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `Contour` object that has one of these:
  - `HitTest` property set to `'on'`
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

---

Note The `PickableParts` property determines if the `Contour` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent
`Axes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `Group`, or `Transform` object.

`Children` — Children
empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the chart object.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object
`'contour'`

This property is read-only.

Type of graphics object, returned as `'contour'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`contour` | `contour3` | `contourf` | `meshc` | `surfc`

Topics

“Access Property Values”
“Graphics Object Properties”

Introduced before R2006a

ConstantLine Properties

Constant line appearance and behavior

Description

A constant line is a line that is graphed at a specified *x*- or *y*-value in Cartesian axes. The `xline` and `yline` functions create constant vertical and horizontal lines, respectively. You can modify the appearance and behavior of the constant line by changing `ConstantLine` property values.

```
xl = xline(4);
xl.LineWidth = 2;
```

Properties

Location

`Value` — Location of constant line

scalar

Location of the constant line on the *x*- or *y*-axis, specified as a scalar. You can specify the value as a numeric, categorical, `datetime`, or duration value.

Example

Create a constant line at x = 5.5. Then change the value to 7.

```
xl = xline(5.5);
xl.Value = 7;
```

`InterceptAxis` — Intercept axis

`'x'` | `'y'`

Intercept axis, specified as `'x'` or `'y'` for a vertical line or horizontal line, respectively. A constant line with an *x*-intercept is a vertical line, whereas a constant line with a *y*-intercept is a horizontal line.

Labels

`Label` — Line label

`''` (default) | character vector | cell array of character vectors | string array | numeric array

Line label, specified as a character vector, cell array of character vectors, string array, or numeric array. To create a multiline label, use a string array or a cell array of character vectors.

Example: `'cutoff frequency'`

Example: `{'first line','second line'}`

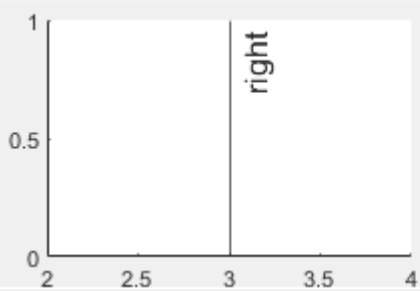
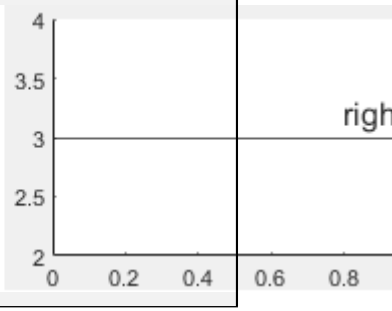
Example: `["first line" "second line"]`

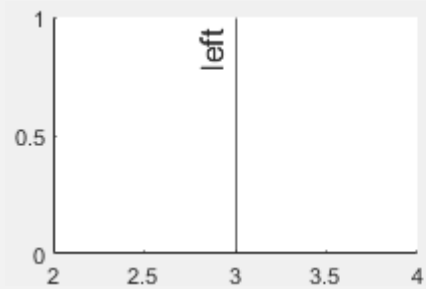
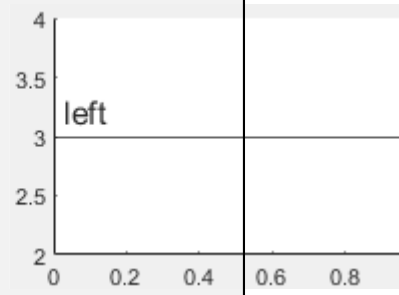
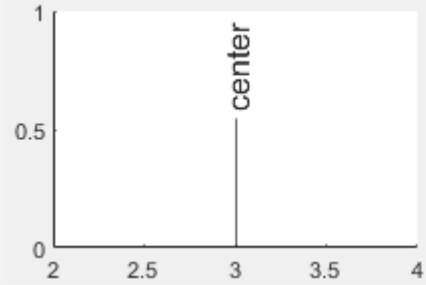
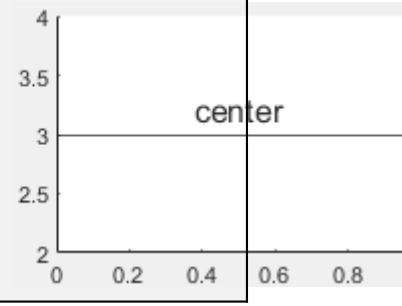
To include special characters, such as superscripts, subscripts, Greek letters, or mathematical symbols, use TeX markup. For a list of supported markup, see the `Interpreter` property.

`LabelHorizontalAlignment` — Horizontal alignment of label

`'right'` (default) | `'left'` | `'center'`

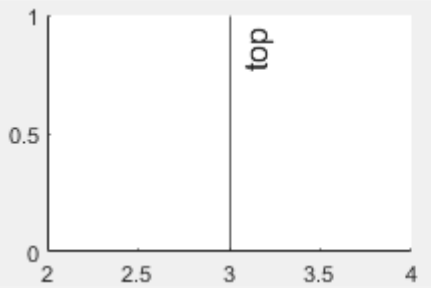
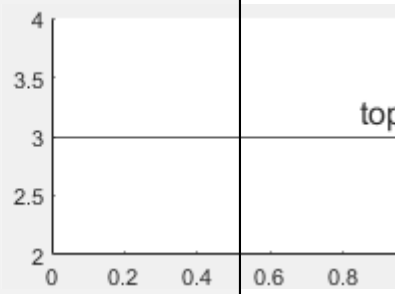
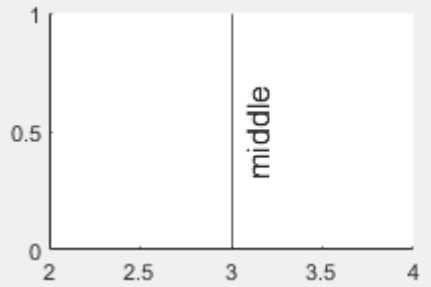
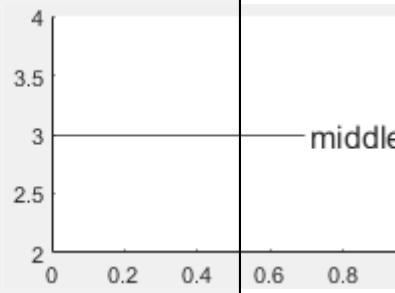
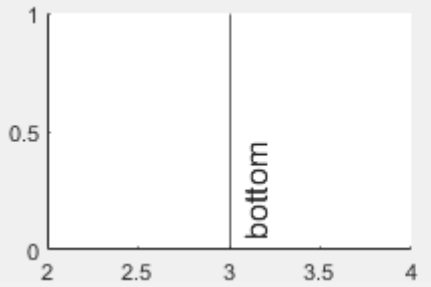
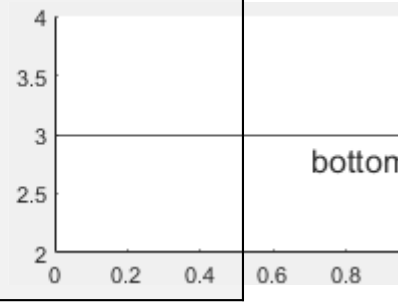
Horizontal alignment of the label with respect to the line, specified as one of the options in the table.

Option	Description
`'right'`	Right side of the line.  

Option	Description
`'left'`	Left side of the line.  
`'center'`	Center of the line. For vertical lines, the label segments the line.  

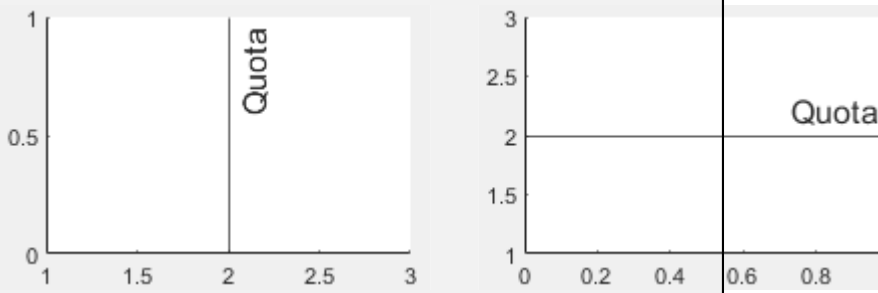
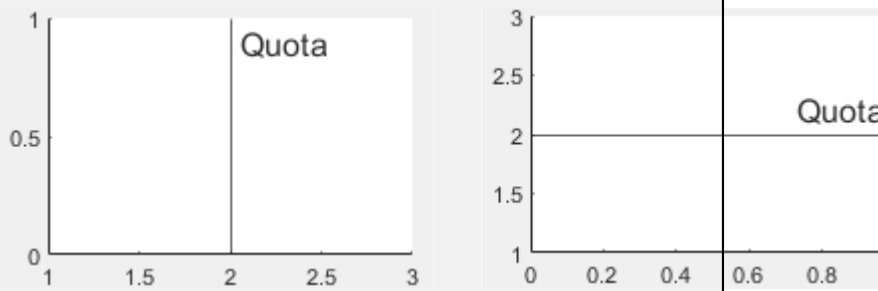
`LabelVerticalAlignment` — Vertical alignment of label
`'top'` (default) | `'middle'` | `'bottom'`

Vertical alignment of the label with respect to the line, specified as one of the options in the table.

Option	Description
`'top'`	Top of the line.  
`'middle'`	Middle of the line. For horizontal lines, the label segments the line.  
`'bottom'`	Bottom of the line.  

`LabelOrientation` — Label orientation
`'aligned'` (default) | `'horizontal'`

Label orientation, specified as `'aligned'` or `'horizontal'`. Examples are shown in the table.

Orientation	Description
`'aligned'`	Label has the same orientation as the line. 
`'horizontal'`	Label is horizontal, regardless of the line orientation. 

`Interpreter` — Interpretation of text characters
`'tex'` (default) | `'latex'` | `'none'`

Interpretation of text characters, specified as one of these values:

- `'tex'` — Interpret characters using a subset of TeX markup.
- `'latex'` — Interpret characters using LaTeX markup.
- `'none'` — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text.

Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they modify only the next character or the characters within the curly braces. When you set the `Interpreter` property to `'tex'`, the supported modifiers are as follows.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{`*`specifier`*`}`	Font name — Replace *`specifier`* with the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{`*`specifier`*`}`	Font size —Replace *`specifier`* with a numeric scalar value in point units.	`'\fontsize{15} text'`
`\color{`*`specifier`*`}`	Font color — Replace *`specifier`* with one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`

Modifier	Description	Example
`\color[rgb] {specifier}`	Custom font color — Replace *specifier* with a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	β	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ϵ	`\Delta`	Δ	`\spadesuit`	♠
`\zeta`	ζ	`\Theta`	Θ	`\leftrightarrow`	↔
`\eta`	η	`\Lambda`	Λ	`\leftarrow`	←
`\theta`	θ	`\Xi`	Ξ	`\Leftarrow`	⇐
`\vartheta`	ϑ	`\Pi`	Π	`\uparrow`	↑
`\iota`	ι	`\Sigma`	Σ	`\rightarrow`	→
`\kappa`	κ	`\Upsilon`	ϒ	`\Rightarrow`	⇒
`\lambda`	λ	`\Phi`	Φ	`\downarrow`	↓
`\mu`	µ	`\Psi`	Ψ	`\circ`	º
`\nu`	ν	`\Omega`	Ω	`\pm`	±
`\xi`	ξ	`\forall`	∀	`\geq`	≥
`\pi`	π	`\exists`	∃	`\propto`	∝

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. Use dollar symbols around the text, for example, use `'$\int_1^{20} x^2 dx$'` for inline mode or `'$$ \int_1^{20} x^2 dx$$'` for display mode.

The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup.

The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters. For multiline text, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project website at https://www.latex-project.org/.

Color and Styling

`Color` — Line color

`[0.15 0.15 0.15]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: 'g'

Example: [0.6 0.2 0.5]

Example: '#D2F9A7'

LineStyle — Line style

'-' (default) | '--' | ':' | '-.' | 'none'

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
'-'	Solid line	
'--'	Dashed line	
':'	Dotted line	
'-.'	Dash-dotted line	
'none'	No line	No line

LineWidth — Line width

0.5 (default) | positive value

Line width, specified as a positive value in points.

Alpha — Line transparency

0.7 (default) | scalar in range [0,1]

Line transparency, specified as a scalar in the range `[0,1]`. A value of `1` is opaque and `0` is completely transparent. Values between `0` and `1` are semitransparent.

Font

`FontName` — Font name
supported font name | `'FixedWidth'`

Font name, specified as a supported font name or `'FixedWidth'`. To display and print text properly, you must choose a font that your system supports. The default font depends on your operating system and locale.

To use a fixed-width font that looks good in any locale, use `'FixedWidth'`. The fixed-width font relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`FontSize` — Font Size
positive number

Font size, specified as a positive number. The unit of measurement is points. The default font size depends on your operating system and locale.

`FontWeight` — Character thickness
`'normal'` (default) | `'bold'`

Character thickness, specified as `'normal'` or `'bold'`.

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.

`FontAngle` — Character slant
`'normal'` (default) | `'italic'`

Character slant, specified as `'normal'` or `'italic'`.

Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.

Legend

`DisplayName` — Legend label

`''` (default) | character vector | string scalar

Legend label, specified as a character vector or string scalar. The legend does not display until you call the `legend` command. If you do not specify the text, then `legend` sets the label using the form `'dataN'`.

`Annotation` — Control for including or excluding object from legend

`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

UIContextMenu — Context menu

empty GraphicsPlaceholder array (default) | ContextMenu object

Context menu, specified as a ContextMenu object. Use this property to display a context menu when you right-click the object. Create the context menu using the uicontextmenu function.

Note If the PickableParts property is set to 'none' or if the HitTest property is set to 'off', then the context menu does not appear.

Selected — Selection state

'off' (default) | 'on'

Selection state, specified as one of these values:

- 'on' — Selected. If you click the object when in plot edit mode, then MATLAB sets its Selected property to 'on'. If the SelectionHighlight property also is set to 'on', then MATLAB displays selection handles around the object.
- 'off' — Not selected.

SelectionHighlight — Display of selection handles

'on' (default) | 'off'

Display of selection handles when selected, specified as one of these values:

- 'on' — Display selection handles when the Selected property is set to 'on'.
- 'off' — Never display selection handles, even when the Selected property is set to 'on'.

Callbacks

ButtonDownFcn — Mouse-click callback

'' (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks when visible. The `Visible` property must be set to `'on'` and you must click a part of the `ConstantLine` object that has a defined color. You cannot click a part that has an associated color property set to `'none'`. The `HitTest` property determines if the `ConstantLine` object responds to the click or if an ancestor does.
- `'all'` — Capture mouse clicks regardless of visibility. The `Visible` property can be set to `'on'` or `'off'` and you can click a part of the `ConstantLine` object that has no color. The `HitTest` property determines if the `ConstantLine` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `ConstantLine` object passes the click through it to the object below it in the current view of the figure window. The `HitTest` property has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `ConstantLine` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `ConstantLine` object that has one of these:
  - `HitTest` property set to `'on'`
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

Note The `PickableParts` property determines if the `ConstantLine` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent

`Axes` object (default)

Parent, specified as an `Axes` object.

`Children` — Children

empty `GraphicsPlaceholder` array

The object has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object
`'constantline'`

This property is read-only.

Type of graphics object, returned as `'constantline'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`xline` | `yline`

Topics

"Access Property Values"
"Graphics Object Properties"

Introduced in R2018b

contourslice

Draw contours in volume slice planes

Syntax

```
contourslice(X,Y,Z,V,xslice,yslice,zslice)
contourslice(V,xslice,yslice,zslice)
contourslice( ___ ,num)
contourslice( ___ ,lvls)
contourslice( ___ ,method)
contourslice(ax, ___ )
s = contourslice( ___ )
```

Description

`contourslice(X,Y,Z,V,xslice,yslice,zslice)` draws contours in slices for the volumetric data `V`, where `V` determines the contour colors. Specify `X`,`Y`, and `Z` as the coordinate data. Specify `xslice`, `yslice`, and `zslice` as the slice locations using one of these forms:

- To draw one or more slice planes that are orthogonal to a particular axis, specify the slice arguments as a scalar or vector.
- To draw a single slice along a surface, specify all the slice arguments as matrices that define a surface.

`contourslice(V,xslice,yslice,zslice)` uses the default coordinate data for `V`. The (*x*,*y*,*z*) location for each element in `V` is based on the column, row, and page index, respectively.

`contourslice( ___ ,num)` specifies the number of contour lines to draw per slice. Use this option with any of the input arguments from the previous syntaxes.

`contourslice( ___ ,lvls)` specifies the values at which to draw contour lines within each slice.

`contourslice( ___ ,method)` specifies the interpolation method, where `method` can be `'linear'`, `'cubic'`, or `'nearest'`.

`contourslice(ax, ___ )` plots into the specified axes, instead of the current axes (`gca`).

`s = contourslice( ___ )` returns the `Patch` objects created.

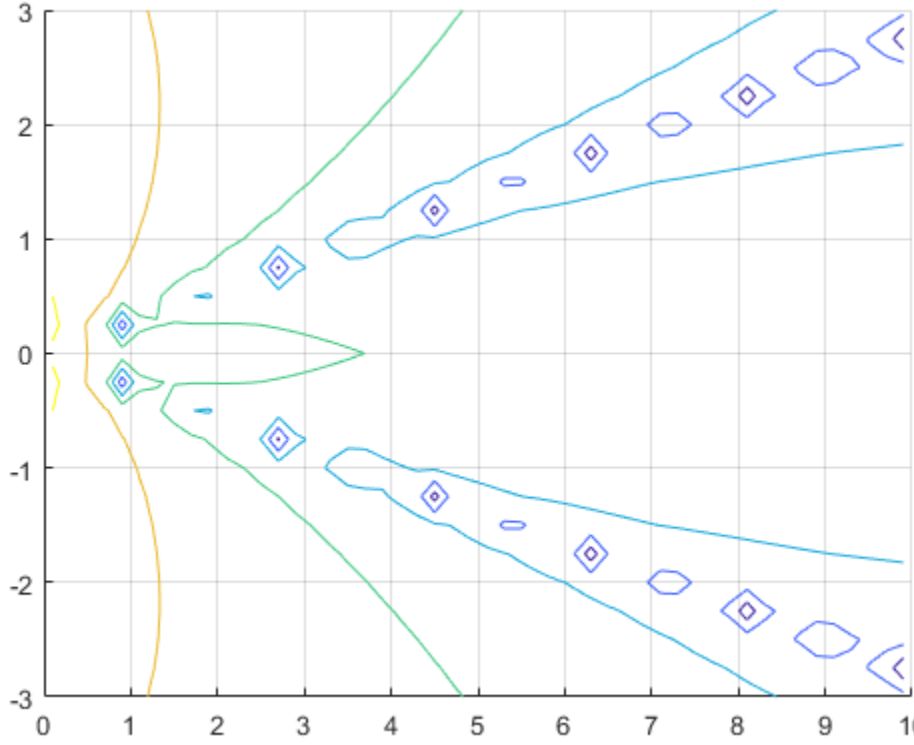
Examples

Draw Contours for Single Slice

Create a single slice plane through volumetric data and draw contours of the data values.

Create the matrices `X`, `Y`, `Z`, and `V` from the `flow` data set. Draw contours of V in the slice plane where z = 0.

```
[X,Y,Z,V] = flow;
zslice = 0;
contourslice(X,Y,Z,V,[],[],zslice)
grid on
```

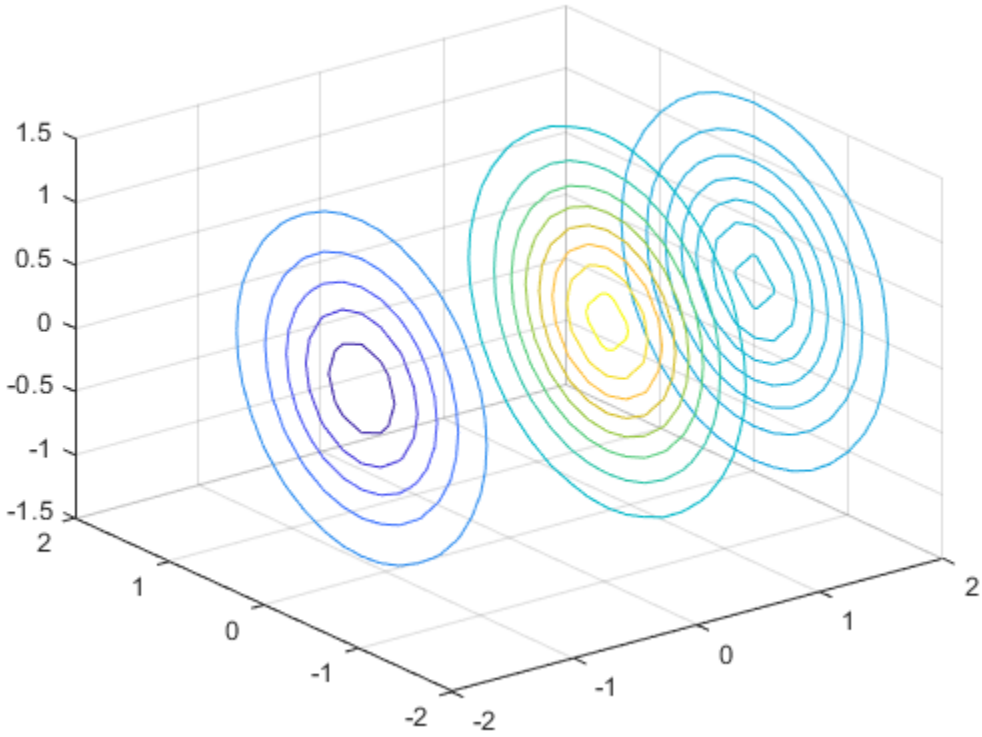


Draw Contours in Slice Planes

Create multiple slice planes through volumetric data. Draw contours of the data values in each of the slice planes. Let MATLAB choose the contour levels.

Create `V` as an array of volume data defined by $v = xe^{-x^2 - y^2 - z^2}$. Draw contours in three slice planes that are orthogonal to the x-axis at the values `-1.2`, `0.8`, and `2`. Do not create any slice planes that are orthogonal to the y-axis or z-axis by specifying empty arrays. Change the axes view to a 3-D view and add grid lines.

```
[X,Y,Z] = meshgrid(-2:.2:2);
V = X.*exp(-X.^2-Y.^2-Z.^2);

xslice = [-1.2,0.8,2];
yslice = [];
zslice = [];
contourslice(X,Y,Z,V,xslice,yslice,zslice)
view(3)
grid on
```

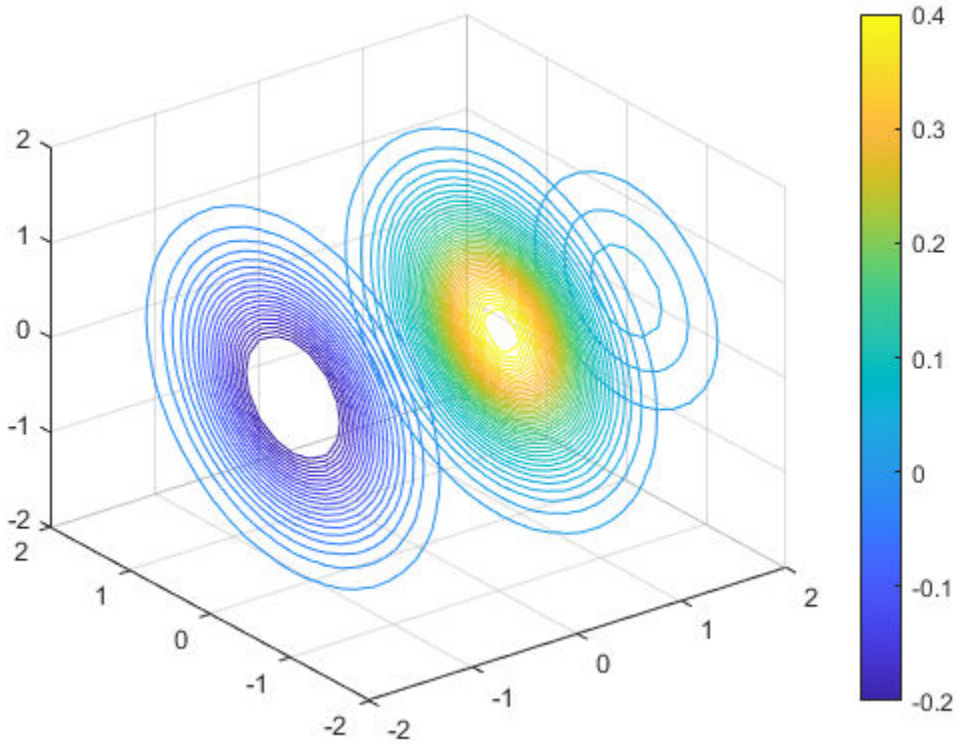


Specify Contour Levels and Add Colorbar

Create multiple slice planes through volumetric data. Draw contours of the data values in each of the slice planes. Specify the levels where you want to draw the contours. Then, add a colorbar.

Draw contours in three slice planes that are orthogonal to the *x*-axis at the values `-1.2`, `0.8`, and `2`. Draw contours for the volume data between the values `-0.2` and `0.4`, spaced every `0.01`. Add a colorbar to see how the data values map to the contour colors. Also, change the axes view to a 3-D view and add grid lines.

```
[X,Y,Z] = meshgrid(-2:.2:2);
V = X.*exp(-X.^2-Y.^2-Z.^2);
xslice = [-1.2,0.8,2];
lvls = -0.2:0.01:0.4;

contourslice(X,Y,Z,V,xslice,[],[],lvls)
colorbar
view(3)
grid on
```



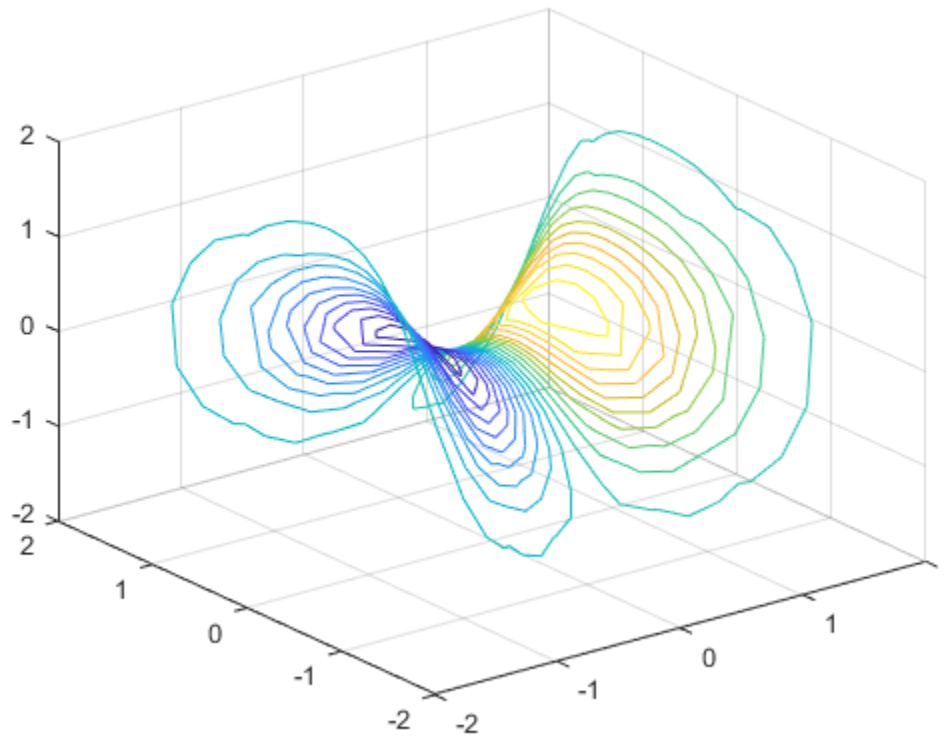
Draw Contours Along Surface Slice

Create a surface slice through volumetric data and draw contours along the surface of the volume data. Specify the number of contour levels to draw.

Create V as an array of volume data defined by $v = xe^{-x^2 - y^2 - z^2}$. Then, show a slice of the volume data along the surface defined by $z = x^2 - y^2$. Draw 20 contours along the surface. Change the axes view to a 3-D view and add grid lines.

```
[X,Y,Z] = meshgrid(-5:0.2:5);
V = X.*exp(-X.^2-Y.^2-Z.^2);
```

```
[xsurf,ysurf] = meshgrid(-2:0.2:2);
zsurf = xsurf.^2-ysurf.^2;
contourslice(X,Y,Z,V,xsurf,ysurf,zsurf,20)
view(3)
grid on
```



Input Arguments

X — *x*-axis coordinate data
vector | 3-D array

x-axis coordinate data, specified as a vector or a 3-D array the same size as `V`. If you specify a 3-D array, it must be monotonic and orthogonally spaced, as if produced by the `meshgrid` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Y` — *y*-axis coordinate data
vector | 3-D array

y-axis coordinate data, specified as a vector or a 3-D array the same size as `V`. If you specify a 3-D array, it must be monotonic and orthogonally spaced, as if produced by the `meshgrid` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Z` — *z*-axis coordinate data
vector | 3-D array

z-axis coordinate data, specified as a vector or a 3-D array the same size as `V`. If you specify a 3-D array, it must be monotonic and orthogonally spaced, as if produced by the `meshgrid` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`V` — Volume data
3-D array

Volume data, specified as a 3-D array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`xslice` — *x*-axis slice values
scalar | vector | `[]` | matrix

x-axis slice values, specified in one of these forms:

- Scalar — Draw one slice plane orthogonal to the *x*-axis at the specified location.
- Vector — Draw multiple slice planes orthogonal to the *x*-axis at the specified locations.
- `[]` — Do not draw any slice planes.

- Matrix — Draw the slice along a surface, instead of a plane. If you use this option, then `yslice` and `zslice` also must be matrices that are the same size.

Example: `[2 4]` draws slice planes orthogonal to the x-axis at the values 2 and 4.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`yslice` — *y*-axis slice values
scalar | vector | `[]` | matrix

y-axis slice values, specified in one of these forms:

- Scalar — Draw one slice plane orthogonal to the *y*-axis at the specified location.
- Vector — Draw multiple slice planes orthogonal to the *y*-axis at the specified locations.
- `[]` — Do not draw any slice planes.
- Matrix — Draw the slice along a surface, instead of a plane. If you use this option, then `xslice` and `zslice` also must be matrices that are the same size.

Example: `[2 4]` draws slice planes orthogonal to the y-axis at the values 2 and 4.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`zslice` — *z*-axis slice values
scalar | vector | `[]` | matrix

z-axis slice values, specified in one of these forms:

- Scalar — Draw one slice plane orthogonal to the *z*-axis at the specified location.
- Vector — Draw multiple slice planes orthogonal to the *z*-axis at the specified locations.
- `[]` — Do not draw any slice planes.
- Matrix — Draw the slice along a surface, instead of a plane. If you use this option, then `xslice` and `yslice` also must be matrices that are the same size.

Example: `[2 4]` draws slice planes orthogonal to the z-axis at the values 2 and 4.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`num` — Number of contour lines
scalar

Number of contour lines per slice, specified as a scalar.

Example: 4 draws four contour lines within each slice.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`lvls` — Contour line levels
vector

Contour line levels, specified as a vector. If you want to draw a single contour line per slice at a particular level, then specify the vector in the form `[lvl lvl]`.

Example: `[1 3 5]` draws contour lines at the levels 1, 3, and 5 within each slice.

Example: `[4 4]` draws a single contour line at the level 4 within each slice.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`method` — Interpolation method
`'linear'` (default) | `'cubic'` | `'nearest'`

Interpolation method, specified as one of these values:

- `'linear'` — Linear interpolation of the values at neighboring grid points in each respective dimension. This value is the default value when the slice is a surface.
- `'cubic'` — Cubic interpolation of the values at neighboring grid points in each respective dimension.
- `'nearest'` — Nearest grid point value. This value is the default value when the slices are planes.

`ax` — Target axes
current axes (default) | `Axes` object

Target axes, specified as an `Axes` object. If you do not specify the axes, then `contourslice` plots into the current axes.

Algorithms

The `contourslice` function does not clear existing plots before plotting.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`contour` | `interp3` | `isosurface` | `meshgrid` | `slice`

Introduced before R2006a

matlab.unittest.constraints Package

Summary of classes in MATLAB Constraints Interface

Description

Constraints specify business rules against which to qualify a calculated value. Use constraints in conjunction with the `matlab.unittest.qualifications` qualification methods `assertThat`, `assumeThat`, `fatalAssertThat`, or `verifyThat`. Constraints determine whether or not a calculated (actual) value satisfies the constraint. Constraints also provide diagnostics. The `matlab.unittest.constraints` package consists of the following classes.

Classes

Constraint Implementations

Fundamental Constraint-Related Interfaces

matlab.unittest.constraints.BooleanConstraint	Interface class for Boolean combinations of constraints
matlab.unittest.constraints.Constraint	Fundamental interface class for comparisons

General Purpose

matlab.unittest.constraints.Eventually	Poll for value to asynchronously satisfy constraint
matlab.unittest.constraints.HasField	Constraint specifying structure containing particular field
matlab.unittest.constraints.IsAnything	Constraint specifying any value
matlab.unittest.constraints.IsEqualTo	General constraint to compare for equality
matlab.unittest.constraints.IsFalse	Constraint specifying false value
matlab.unittest.constraints.IsSameHandleAs	Constraint specifying handle instance same as another
matlab.unittest.constraints.IsTrue	Constraint specifying true value

matlab.unittest.constraints.ReturnsTrue	Constraint specifying function handle that returns true

Errors and Warnings

matlab.unittest.constraints.IssuesNoWarnings	Constraint specifying function that issues no warnings
matlab.unittest.constraints.IssuesWarnings	Constraint specifying function that issues expected warning profile
matlab.unittest.constraints.Throws	Constraint specifying function handle that throws MException

Inequalities

matlab.unittest.constraints.IsGreaterThan	Constraint specifying value greater than another value
matlab.unittest.constraints.IsGreaterThanOrEqualTo	Constraint specifying value greater than or equal to another value
matlab.unittest.constraints.IsLessThan	Constraint specifying value less than another value
matlab.unittest.constraints.IsLessThanOrEqualTo	Constraint specifying value less than or equal to another value

Array Size

matlab.unittest.constraints.HasElementCount	Constraint specifying expected number of elements
matlab.unittest.constraints.HasLength	Constraint specifying expected length of array
matlab.unittest.constraints.HasSize	Constraint specifying expected size of array
matlab.unittest.constraints.IsEmpty	Constraint specifying empty value
matlab.unittest.constraints.IsScalar	Constraint specifying scalar value

Type

matlab.unittest.constraints.IsInstanceOf	Constraint specifying inclusion in given class hierarchy
matlab.unittest.constraints.IsOfClass	Constraint specifying class type

Strings

matlab.unittest.constraints.ContainsSubstring	Constraint specifying string containing substring

matlab.unittest.constraints.EndsWithSubstring	Constraint specifying string ending with substring
matlab.unittest.constraints.IsSubstringOf	Constraint specifying substring of another string
matlab.unittest.constraints.Matches	Constraint specifying string matches regular expression
matlab.unittest.constraints.StartsWithSubstring	Constraint specifying string starting with substring

Finiteness

matlab.unittest.constraints.HasInf	Constraint specifying array containing any infinite value
matlab.unittest.constraints.HasNaN	Constraint specifying array containing NaN value
matlab.unittest.constraints.IsFinite	Constraint specifying finite value

Numeric Attributes

matlab.unittest.constraints.IsReal	Constraint specifying real valued array
matlab.unittest.constraints.IsSparse	Constraint specifying sparse array

Set

matlab.unittest.constraints.HasUniqueElements	Constraint specifying set contains unique elements
matlab.unittest.constraints.IsSameSetAs	Constraint specifying set contains same elements as another set
matlab.unittest.constraints.IsSubsetOf	Constraint specifying actual set is subset of expected set
matlab.unittest.constraints.IsSupersetOf	Constraint specifying actual set is superset of expected set

Files and Folders

matlab.unittest.constraints.IsFile	Constraint specifying value points to file
matlab.unittest.constraints.IsFolder	Constraint specifying value points to folder

Actual Value Proxies

matlab.unittest.constraints.AnyCellOf	Test if any element of cell array meets constraint
matlab.unittest.constraints.AnyElementOf	Test if any element of array meets constraint
matlab.unittest.constraints.EveryCellOf	Test if all elements of cell array meet constraint

matlab.unittest.constraints.EveryElementOf	Test if all elements of array meet constraint

Tolerances

matlab.unittest.constraints.AbsoluteTolerance	Absolute numeric tolerance
matlab.unittest.constraints.RelativeTolerance	Relative numeric tolerance
matlab.unittest.constraints.Tolerance	Abstract interface class for tolerances

Comparators

matlab.unittest.constraints.CellComparator	Comparator for cell arrays
matlab.unittest.constraints.LogicalComparator	Comparator for two logical values
matlab.unittest.constraints.NumericComparator	Comparator for numeric data types
matlab.unittest.constraints.ObjectComparator	Comparator for MATLAB or Java objects
matlab.unittest.constraints.PublicPropertyComparator	Comparator for public properties of MATLAB objects
matlab.unittest.constraints.StringComparator	Comparator for two strings, character arrays, or cell arrays of character arrays
matlab.unittest.constraints.StructComparator	Comparator for MATLAB structure arrays
matlab.unittest.constraints.TableComparator	Comparator for MATLAB tables

See Also

`Simulink.sdi.constraints.MatchesSignal`

Introduced in R2013a

matlab.unittest.constraints.AbsoluteTolerance class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.Tolerance`

Absolute numeric tolerance

Description

This numeric `Tolerance` assesses the magnitude of the difference between actual and expected values. For the tolerance to be satisfied, `abs(expVal - actVal) <= absTol` must be true.

Construction

`AbsoluteTolerance(tolVals)` creates an absolute tolerance object that assesses the magnitude of the difference between the actual and expected values.

The data types of the inputs to the `AbsoluteTolerance` constructor determines which data types the tolerance supports. For example, `AbsoluteTolerance(10*eps)` constructs an `AbsoluteTolerance` for comparing double-precision numeric arrays, while `AbsoluteTolerance(int8(2))` constructs an `AbsoluteTolerance` for comparing numeric arrays of type `int8`. If the actual and expected values being compared contain more than one numeric data type, the tolerance only applies to the data types specified by the values passed into the constructor.

To specify different tolerance values for different data types, you can pass multiple tolerance values to the constructor. For example, `AbsoluteTolerance(10*eps, 10*eps('single'), int8(1))` constructs an `AbsoluteTolerance` object applies the following absolute tolerances:

- `10*eps` applies an absolute tolerance of `10*eps` for double-precision numeric arrays.
- `10*eps('single')` applies an absolute tolerance of `10*eps` for single-precision numeric arrays.

- `int8(1)` applies an absolute tolerance of `1` for numeric arrays of type `int8`.

You can specify more than one tolerance for a particular data type by combining tolerances with the & and | operators. To combine two tolerances, the sizes of the tolerance values for each data type must be compatible.

Input Arguments

`tolVals`

Numeric tolerances, specified as a comma-separated list of numeric arrays. Each input argument contains the tolerance specification for a particular data type. Each numeric array can be a scalar or array the same size as the actual and expected values.

Properties

`Values`

Numeric tolerances, specified by the `tolVals` input argument

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test with Absolute Tolerance

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.AbsoluteTolerance

testCase = TestCase.forInteractiveUse;
```

Assert that the difference between an actual value, `4.1`, and an expected value, `4.5`, is less than `0.5`.

```
testCase.assertThat(4.1, IsEqualTo(4.5, ...
    'Within', AbsoluteTolerance(0.5)))
```

```
Assertion passed.
```

Specify Absolute Tolerance for Different Data Types

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.AbsoluteTolerance

testCase = TestCase.forInteractiveUse;
```

Create the following actual and expected cell arrays.

```
act = {'abc', 123, single(106), int8([1, 2, 3])};
exp = {'abc', 122, single(105), int8([2, 4, 6])};
```

Test whether the arrays satisfy the `AbsoluteTolerance` constraint within a value of `2`.

```
testCase.verifyThat(act, IsEqualTo(exp, ...
    'Within', AbsoluteTolerance(2)))
```

```
Verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> Path to failure: <Value>{3}
    --> NumericComparator failed.
        --> The numeric values are not equal using "isequaln".
        --> The tolerance was ignored. The tolerance as specified does not support comparisons of single val
        --> Failure table:
                    Actual    Expected    Error    RelativeError
                    ______    ________    _____    _____________

                    106       105         1        0.00952381

        Actual single:
               106
        Expected single:
```

```
               105

Actual cell:
        'abc'    [123]    [106]    [1×3 int8]
Expected cell:
        'abc'    [122]    [105]    [1×3 int8]
```

The test fails because the tolerance is only applied to the `double` data type.

Create a tolerance object that specifies different tolerances for different data types.

```
tolObj = AbsoluteTolerance(2, single(3), int8([2, 3, 5]));
```

A tolerance of `2` is a applied to `double` valued data. A tolerance of `3` is applied to `single` valued data. A tolerance of `[2 3 5]` is applied to corresponding array elements of `int8` valued data.

Verify that the expected and actual values satisfy the `AbsoluteTolerance` constraint.

```
testCase.verifyThat(act, IsEqualTo(exp, 'Within', tolObj))
```

```
Interactive verification passed.
```

Combine Absolute and Relative Tolerances

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.AbsoluteTolerance
import matlab.unittest.constraints.RelativeTolerance

testCase = TestCase.forInteractiveUse;
```

Define an actual value approximation for `pi`.

```
act = 3.14;
```

Construct a tolerance object to test that the difference between the actual and expected values is within `0.001` and within `0.25%`.

```
tolObj = AbsoluteTolerance(0.001) & RelativeTolerance(0.0025);
```

Verify that the actual value is within the tolerance of the expected value of `pi`.

```
testCase.verifyThat(act, IsEqualTo(pi, 'Within', tolObj))
```

```
Verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> NumericComparator failed.
    --> The numeric values are not equal using "isequaln".
    --> AndTolerance failed.
        --> AbsoluteTolerance failed.
            --> The error was not within absolute tolerance.
        --> RelativeTolerance passed.
            --> The error was within relative tolerance.
        --> Failure table:
                    Actual        Expected               Error                RelativeError        AbsoluteTolerance    Re
                    ______    _______________    ___________________    ____________________    _________________    __

                    3.14      3.14159265358979    -0.00159265358979299    -0.000506957382897213    0.001                0.

    Actual double:
           3.140000000000000
    Expected double:
           3.141592653589793
```

The actual value does not satisfy the `AbsoluteTolerance` constraint.

Construct a constraint that is satisfied if the values are within `0.001` or `0.25%`, and then retest the actual value.

```
tolObj = AbsoluteTolerance(0.001) | RelativeTolerance(0.0025);
testCase.verifyThat(act, IsEqualTo(pi, 'Within', tolObj))
```

```
Verification passed.
```

Combine Absolute and Relative Tolerances to Test Small and Large Values

Combine tolerances so when you test the equality of values, an absolute (floor) tolerance dominates when the values are near zero, and a relative tolerance dominates for larger values.

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.AbsoluteTolerance
import matlab.unittest.constraints.RelativeTolerance

testCase = TestCase.forInteractiveUse;
```

Define two structures containing electromagnetic properties of a vacuum. One structure, `approxVacuumProps`, contains approximate values for the permeability and speed of light in a vacuum.

```
approxVacuumProps.Permeability = 1.2566e-06; % Approximate
approxVacuumProps.Permitivity = 8.854187817*10^-12;
approxVacuumProps.LightSpeed = 2.9979e+08; % Approximate

baselineVacuumProps.Permeability = 4*pi*10^-7;
baselineVacuumProps.Permitivity = 8.854187817*10^-12;
baselineVacuumProps.LightSpeed = 1/sqrt(...
    baselineVacuumProps.Permeability*baselineVacuumProps.Permitivity);
```

Test that the relative difference between the approximate and baseline values is within `eps*1e11`.

```
testCase.verifyThat(approxVacuumProps, IsEqualTo(baselineVacuumProps, ...
    'Within', RelativeTolerance(eps*1e11)))
```

```
Verification failed.

----------------------
Framework Diagnostic:
----------------------
IsEqualTo failed.
--> Path to failure: <Value>.Permeability
    --> NumericComparator failed.
        --> The numeric values are not equal using "isequaln".
        --> RelativeTolerance failed.
            --> The error was not within relative tolerance.
            --> Failure table:
                          Actual            Expected                  Error
                        __________    ____________________    ____________________

                        1.2566e-06    1.25663706143592e-06    -3.70614359173257e-11

        Actual double:
                 1.256600000000000e-06
        Expected double:
                 1.256637061435917e-06

Actual struct:
        Permeability: 1.256600000000000e-06
         Permitivity: 8.854187816999999e-12
          LightSpeed: 299790000
```

```
Expected struct:
        Permeability: 1.256637061435917e-06
         Permitivity: 8.854187816999999e-12
          LightSpeed: 2.997924580105029e+08
```

The test fails because the relative difference in the permeabilities is not within the tolerance. The difference between the two values is small, but the numbers are close to zero, so the difference relative to their size is not small enough to satisfy the tolerance.

Construct a tolerance object to test that the absolute difference between the approximate and baseline values is within `1e-4`.

```
testCase.verifyThat(approxVacuumProps, IsEqualTo(baselineVacuumProps, ...
    'Within', AbsoluteTolerance(1e-4)))
```

```
Verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> Path to failure: <Value>.LightSpeed
    --> NumericComparator failed.
        --> The numeric values are not equal using "isequaln".
        --> AbsoluteTolerance failed.
            --> The error was not within absolute tolerance.
            --> Failure table:
                         Actual          Expected              Error             Relati
                        _________    _________________    _________________    _________

                        299790000    299792458.010503    -2458.01050287485    -8.19904(

        Actual double:
               299790000
        Expected double:
                 2.997924580105029e+08

Actual struct:
        Permeability: 1.256600000000000e-06
         Permitivity: 8.854187816999999e-12
          LightSpeed: 299790000
Expected struct:
        Permeability: 1.256637061435917e-06
         Permitivity: 8.854187816999999e-12
          LightSpeed: 2.997924580105029e+08
```

The test fails because the absolute difference in the speed of light is not within the tolerance. The difference between the two values is small relative to their size, but too large to satisfy the tolerance.

Construct a logical disjunction of tolerance objects to test that the absolute difference between the approximate and baseline values is within `1e-4` or the relative difference is within `eps*1e11`. The test uses this tolerance so permeability values that are close to zero satisfy the absolute (floor) tolerance, and speed of light values that are large, satisfy the relative tolerance.

```
testCase.verifyThat(approxVacuumProps, IsEqualTo(baselineVacuumProps, ...
    'Within', RelativeTolerance(eps*1e11)| AbsoluteTolerance(1e-4)))
```

```
Verification passed.
```

See Also

`matlab.unittest.constraints.IsEqualTo` | `matlab.unittest.constraints.RelativeTolerance`

Introduced in R2013a

matlab.unittest.constraints.AnyCellOf class

Package: `matlab.unittest.constraints`

Test if any element of cell array meets constraint

Description

The `AnyCellOf` class creates a proxy of the actual value to the framework. The proxy enables a test writer to apply a constraint against each element of a cell array, which ensures that a passing result occurs if at least one element of the cell array satisfies the constraint.

It is intended that you use this class through `matlab.unittest` qualifications as shown in the examples. The class does not modify the provided actual value, but serves as a wrapper to perform the constraint analysis. The testing framework analyzes the constraint on an element-by-element basis.

Construction

`AnyCellOf(actVal)` creates a proxy instance that tests if any element of a provided cell array, `actVal`, meets a constraint. The test passes if at least one element individually satisfies the constraint.

Input Arguments

`actVal`

Actual value to test against constraint

Properties

ActualValue

Actual value to test against constraint. Set this property through the constructor via the `actVal` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test That Any Cell Satisfies Constraint

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.AnyCellOf

testCase = TestCase.forInteractiveUse;
```

Test that at least one cell of `actVal` is finite.

```
import matlab.unittest.constraints.IsFinite
actVal = {NaN, Inf, 5};
testCase.verifyThat(AnyCellOf(actVal), IsFinite)
```

```
Interactive verification passed.
```

Test that at least one cell of the actual value contains five elements.

```
import matlab.unittest.constraints.HasElementCount
testCase.verifyThat(AnyCellOf({42, [11 38], 1:5}), HasElementCount(5))
```

```
Interactive verification passed.
```

Test that at least one cell of the actual value matches `'tea'` regardless of case.

```
import matlab.unittest.constraints.Matches
testCase.verifyThat(AnyCellOf({'Coffee','Tea','Water'}), ...
    Matches('tea','IgnoringCase',true))
```

```
Interactive verification passed.
```

Test that at least one cell of the actual value is less than zero.

```
import matlab.unittest.constraints.IsLessThan
testCase.verifyThat(AnyCellOf({1, 5}), IsLessThan(0))
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
All cells failed. The first cell failed because:
--> IsLessThan failed.
    --> The value must be less than the maximum value.

    Actual double:
             1
    Maximum Value (Exclusive):
             0

Actual Value Cell Array:
        [1]    [5]
```

Neither actual value element is less than zero.

Test that neither cell of the actual value is empty.

```
import matlab.unittest.constraints.IsEmpty
testCase.verifyThat(AnyCellOf({inputParser.empty,''}), ~IsEmpty)
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
All cells failed. The first cell failed because:
--> Negated IsEmpty failed.
    --> The value must not be empty.
    --> The value has a size of [0  0].
```

```
    Actual inputParser:
          0x0 inputParser array with properties:

            FunctionName
            CaseSensitive
            KeepUnmatched
            PartialMatching
            StructExpand
            Parameters
            Results
            Unmatched
            UsingDefaults

Actual Value Cell Array:
        [0x0 inputParser]    ''
```

Both actual value elements are empty.

See Also

`AnyElementOf` | `EveryCellOf` | `EveryElementOf` | `matlab.unittest.qualifications`

matlab.unittest.constraints.AnyElementOf class

Package: `matlab.unittest.constraints`

Test if any element of array meets constraint

Description

The `AnyElementOf` class creates a proxy of the actual value to the framework. The proxy enables a test writer to apply a constraint against each element of an array, which ensures that a passing result occurs when at least one element of the array satisfies the constraint.

It is intended that you use this class through `matlab.unittest` qualifications as shown in the examples. The class does not modify the provided actual value, but serves as a wrapper to perform the constraint analysis. The testing framework analyzes the constraint on an element-by-element basis.

Construction

`AnyElementOf(actVal)` creates a proxy instance that tests if any element of a provided array, `actVal`, meets a constraint. The test passes if at least one element individually satisfies the constraint.

Tips

- `AnyElementOf` checks if any element in the provided array satisfies an associated constraint. However, there are some constraints, such as `HasNaN` and `HasInf`, that natively validate if any of the elements satisfy a condition. In these situations, use of `AnyElementOf` is unnecessary and impedes qualification performance.

Input Arguments

actVal

Actual value to test against constraint

Properties

ActualValue

Actual value to test against constraint. Set this property through the constructor via the `actVal` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test That Any Element Satisfies Constraint

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.AnyElementOf

testCase = TestCase.forInteractiveUse;
```

Test that at least one element of `actVal` is finite.

```
import matlab.unittest.constraints.IsFinite
actVal = [NaN, Inf, 5];
testCase.verifyThat(AnyElementOf(actVal), IsFinite)
```

```
Interactive verification passed.
```

Test that at least one element of the actual value is complex.

```
import matlab.unittest.constraints.IsReal
testCase.verifyThat(AnyElementOf([1+0i 4i]), ~IsReal)
```

```
Interactive verification passed.
```

Test that at least one element of the actual value array is less than zero.

```
import matlab.unittest.constraints.IsLessThan
testCase.verifyThat(AnyElementOf([1 5]), IsLessThan(0))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
All elements failed. The first element failed because:
--> IsLessThan failed.
    --> The value must be less than the maximum value.

    Actual Value:
             1
    Maximum Value (Exclusive):
             0

Actual Value Array:
         1     5
```

Neither actual value element is less than zero.

See Also

`AnyCellOf` | `EveryCellOf` | `EveryElementOf` | `matlab.unittest.qualifications`

matlab.unittest.constraints.BooleanConstraint class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.Constraint`

Interface class for Boolean combinations of constraints

Description

The `BooleanConstraint` interface class provides an interface for Boolean combinations of `Constraints`. Any constraint that derives from `BooleanConstraint` can be combined and negated using the `and` (&), `or` (|), and `not` (~) operators.

Classes that derive from the `BooleanConstraint` interface class must implement everything required by the standard `Constraint` interface. When a given constraint is negated, the diagnostics must be written in a different form than for a standard (non-negated) failure. Therefore, classes deriving from the `BooleanConstraint` class must implement a method to provide a `Diagnostic` object for the negated case, in addition to the non-negated case.

In exchange for meeting these requirements, all `BooleanConstraint` implementations inherit the appropriate MATLAB overloads for `and`, `or`, and `not` so that they can be combined with other `BooleanConstraint` objects or negated.

Methods

getNegativeDiagnosticFor	Produce negated diagnostic for value

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Boolean Combinations of Constraints

At the command prompt, create a test case for interactive testing and import several classes that subclass `BooleanConstraint`.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.HasElementCount
import matlab.unittest.constraints.HasLength
import matlab.unittest.constraints.HasInf
import matlab.unittest.constraints.HasNaN
import matlab.unittest.constraints.IsEmpty
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.IsGreaterThanOrEqualTo
import matlab.unittest.constraints.IsOfClass
import matlab.unittest.constraints.IsReal

testCase = TestCase.forInteractiveUse;
```

Test these passing cases.

```
testCase.verifyThat(3, IsReal & IsGreaterThanOrEqualTo(3))
testCase.verifyThat([1 2 3; 4 5 6], HasLength(3) & HasElementCount(6))
testCase.verifyThat([3 NaN 5], HasNaN | HasInf)
testCase.verifyThat(3, ~IsEqualTo(4))
testCase.verifyThat('Some char', IsOfClass(?char) & ~IsEmpty)
```

See Also

`Diagnostic` | `matlab.unittest.constraints`

Topics

"Create Custom Boolean Constraint"

getNegativeDiagnosticFor

Class: `matlab.unittest.constraints.BooleanConstraint`
Package: `matlab.unittest.constraints`

Produce negated diagnostic for value

Syntax

```
diag = getNegativeDiagnosticFor(constObj,actVal)
```

Description

`diag = getNegativeDiagnosticFor(constObj,actVal)` produces a negated diagnostic for a value. The `getNegativeDiagnosticFor` method analyzes the provided value, `actVal`, against the constraint, `constObj`, and produces a `matlab.unittest.diagnostics.Diagnostic` object, `diag`, which corresponds to the negation of the constraint, `constObj`. This method is a protected method.

The diagnostics that this method produces are expressed in the negative sense of the constraint. For example, a hypothetical `IsTasty` constraint, when negated, should express that the actual value was "tasty", when it should not have been, and it should describe the details on why it was found to be tasty.

Like the `getDiagnosticFor` method of `Constraint`, the `getNegativeDiagnosticFor` is only called upon failures, and thus can afford a more detailed analysis than the `satisfiedBy` method.

Input Arguments

`constObj`

`BooleanConstraint` instance

actVal

Value for comparison

Examples

Implement getNegativeDiagnosticFor Method

Create a custom Boolean constraint that determines if a given value has the same size as an expected value. Implement the `getNegativeDiagnosticFor` method to provide a `Diagnostic` object when the constraint is negated.

```
classdef HasSameSizeAs < matlab.unittest.constraints.BooleanConstraint

    properties(SetAccess = immutable)
        ValueWithExpectedSize
    end

    methods
        function constraint = HasSameSizeAs(value)
            constraint.ValueWithExpectedSize = value;
        end

        function bool = satisfiedBy(constraint,actual)
            bool = constraint.sizeMatchesExpected(actual);
        end

        function diag = getDiagnosticFor(constraint,actual)
            import matlab.unittest.diagnostics.StringDiagnostic

            if constraint.sizeMatchesExpected(actual)
                diag = StringDiagnostic('HasSameSizeAs passed.');
            else
                diag = StringDiagnostic(sprintf(...
                    'HasSameSizeAs failed.\nActual Size: [%s]\nExpectedSize: [%s]',...
                    int2str(size(actual)),...
                    int2str(size(constraint.ValueWithExpectedSize))));
            end
        end
    end

    methods(Access = protected)
        function diag = getNegativeDiagnosticFor(constraint,actual)
            import matlab.unittest.diagnostics.StringDiagnostic
            if constraint.sizeMatchesExpected(actual)
                diag = StringDiagnostic(sprintf(...
                    ['Negated HasSameSizeAs failed.\nSize [%s] of '...
                    'Actual Value and Expected Value were the same '...
                    'but should not have been.'],int2str(size(actual))));
```

```
            else
                diag = StringDiagnostic('Negated HasSameSizeAs passed.');
            end
        end
    end

    methods(Access = private)
        function bool = sizeMatchesExpected(constraint,actual)
            bool = isequal(size(actual),size(constraint.ValueWithExpectedSize));
        end
    end

end
```

See Also

`Diagnostic` | `getDiagnosticFor` | `satisfiedBy`

Topics

"Create Custom Boolean Constraint"

matlab.unittest.constraints.CellComparator class

Package: `matlab.unittest.constraints`

Comparator for cell arrays

Description

The `CellComparator` compares cell arrays.

Construction

`CellComparator` creates a comparator for cell arrays.

`CellComparator(compObj)` indicates a comparator, `compObj`, that defines the comparator used to compare values contained in the cell array. By default, a cell comparator supports only empty cell arrays.

`CellComparator(compObj,Name,Value)` provides a comparator with additional options specified by one or more `Name,Value` pair arguments.

`CellComparator(Name,Value)` provides a comparator for empty cell arrays with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`compObj`

Comparator object

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`Recursively`

Indicator of whether comparator operates recursively, specified as `false` or `true` (`logical` `0` or `1`). When this value is `false`, the comparator does not operate recursively on its data.

When the value is `true`, the data types the cell comparator supports are fully supported in recursion. For example:

```
comp1 = CellComparator(StringComparator)
comp2 = CellComparator(StringComparator,'Recursively', true)
```

Both `comp1` and `comp2` support cell arrays of strings and character arrays. However, only `comp2` supports cell arrays that recursively contain either cell arrays or strings as their elements.

Default: `false`

Properties

`Recursive`

Indicator of whether comparator operates recursively, specified in the name-value pair argument, `'Recursively'`

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Compare Cell Arrays

Create a test case for interactive testing.

```
import matlab.unittest.TestCase;
import matlab.unittest.constraints.CellComparator
import matlab.unittest.constraints.StringComparator
import matlab.unittest.constraints.IsEqualTo

testCase = TestCase.forInteractiveUse;
```

Use a `CellComparator` to test that two cell arrays are equal to each other.

```
actual = {'abc','def'};
expected = {'abc','def'};
testCase.verifyThat(actual, IsEqualTo(expected,...
    'Using', CellComparator(StringComparator)))
```

```
Interactive verification passed.
```

By default, the `CellComparator` supports only comparison of empty cell arrays. Therefore, it is necessary to pass it a `StringComparator`.

Change the actual value and compare it to the expected value. To satisfy the constraint, construct it to ignore case and whitespace characters.

```
actual = {'ABC','D E F'};
testCase.verifyThat(actual, IsEqualTo(expected, 'Using', ...
    CellComparator(StringComparator), 'IgnoringWhitespace', ...
    true, 'IgnoringCase',true))
```

```
Interactive verification passed.
```

Test nested cell arrays of character vectors by constructing the comparator to operate recursively.

```
actual = {'abc',{'def','ghi'}};
expected = {'abc',{'def','ghi'}};

testCase.verifyThat(actual, IsEqualTo(expected, 'Using', ...
    CellComparator(StringComparator, 'Recursively', true)))
```

```
Interactive verification passed.
```

See Also

`matlab.unittest.constraints.IsEqualTo` | `matlab.unittest.constraints.Tolerance`

Introduced in R2013a

matlab.unittest.constraints.Constraint class

Package: `matlab.unittest.constraints`

Fundamental interface class for comparisons

Description

The `Constraint` interface class is the means by which `matlab.unittest` constraints encode comparison logic and the corresponding diagnostic information. Every comparison that conditionally can produce a failure inherits from the `Constraint` interface class.

Classes deriving from the `Constraint` interface class must provide a means to determine if a given value satisfies the constraint. To do this, implement the `satisfiedBy` method, which includes the definition of the underlying comparison logic. Classes deriving from the `Constraint` class also must provide a diagnostic for any given actual value. The testing framework uses the diagnostic when it encounters a qualification failure. To do this, implement the `getDiagnosticFor` method.

In exchange for meeting these requirements, all `Constraint` implementations are easily used with all qualification types through the `verifyThat`, `assertThat`, `assumeThat`, or `fatalAssertThat` methods. The qualifications use the comparison and diagnostic knowledge contained within the constraints. Also, the constraints can be used in situations where a test failure is not wanted, but the testing framework needs to reuse the comparison logic. For example, a constraint implementation may want to use the logic defined inside of another constraint. Since the constraint can interact with the other constraint directly, it can use the logic without the potential of causing a qualification failure.

Methods

getDiagnosticFor	Produce diagnostic for compared value
satisfiedBy	Determine whether value satisfies constraint

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

See Also

`ConstraintDiagnostic` | `Diagnostic` | `assertThat` | `assumeThat` | `fatalAssertThat` | `matlab.unittest.constraints` | `verifyThat`

Topics

"Create Custom Constraint"

getDiagnosticFor

Class: `matlab.unittest.constraints.Constraint`
Package: `matlab.unittest.constraints`

Produce diagnostic for compared value

Syntax

```
diag = getDiagnosticFor(constObj,actVal)
```

Description

`diag = getDiagnosticFor(constObj,actVal)` produces a diagnostic, `diag`, for a compared value, `actVal`. When creating a custom constraint, you must implement the `getDiagnosticFor` method so that it analyzes the value, `actVal`, against the constraint, `constObj`, and instantiates and returns a `matlab.unittest.diagnostics.Diagnostic` object.

Typically, the testing framework calls this method when it encounters a qualification failure. Therefore, you can afford to undertake a more detailed analysis in the `getDiagnosticFor` method than the `satisfiedBy` method.

Input Arguments

`actVal`

Value for comparison

`constObj`

`Constraint` instance

Examples

Implement `getDiagnosticFor` Method

Create a custom constraint that determines if a given value has the same size as an expected value. Implement the `getDiagnosticFor` method to evaluate the actual value against the constraint and provide a `Diagnostic` object.

```
classdef HasSameSizeAs < matlab.unittest.constraints.Constraint

    properties(SetAccess = immutable)
        ValueWithExpectedSize
    end

    methods
        function constraint = HasSameSizeAs(value)
            constraint.ValueWithExpectedSize = value;
        end

        function bool = satisfiedBy(constraint,actual)
            bool = constraint.sizeMatchesExpected(actual);
        end

        function diag = getDiagnosticFor(constraint,actual)
            import matlab.unittest.diagnostics.StringDiagnostic
            if constraint.sizeMatchesExpected(actual)
                diag = StringDiagnostic('HasSameSizeAs passed.');
            else
                diag = StringDiagnostic(sprintf(...
                    'HasSameSizeAs failed.\nActual Size: [%s]\nExpectedSize: [%s]',...
                    int2str(size(actual)),...
                    int2str(size(constraint.ValueWithExpectedSize))));
            end
        end
    end

    methods(Access = private)
        function bool = sizeMatchesExpected(constraint,actual)
            bool = isequal(size(actual),size(constraint.ValueWithExpectedSize));
        end
    end

end
```

See Also

`ConstraintDiagnostic` | `Diagnostic` | `satisfiedBy`

Topics

“Create Custom Constraint”

satisfiedBy

Class: `matlab.unittest.constraints.Constraint`
Package: `matlab.unittest.constraints`

Determine whether value satisfies constraint

Syntax

```
TF = satisfiedBy(constObj,actVal)
```

Description

`TF = satisfiedBy(constObj,actVal)` determines whether a value, `actVal`, satisfies a constraint, `constObj`. The `satisfiedBy` method is used to determine qualification success or failure. It returns `true` or `false` (logical `0` or `1`). When creating a custom constraint, you must place the comparison logic in this method.

Since the most common usage is for the passing case, you should optimize for speed in that case. It is only in the failing case that more expensive detailed analysis is helpful.

Input Arguments

`actVal`

Value to evaluate against the constraint

`constObj`

`Constraint` instance

Examples

Implement satisfiedBy Method

Create a custom constraint that determines if a given value has the same size as an expected value. Include the comparison logic in the `satisfiedFor` method.

```
classdef HasSameSizeAs < matlab.unittest.constraints.Constraint

    properties(SetAccess = immutable)
        ValueWithExpectedSize
    end

    methods
        function constraint = HasSameSizeAs(value)
            constraint.ValueWithExpectedSize = value;
        end

        function bool = satisfiedBy(constraint,actual)
            bool = constraint.sizeMatchesExpected(actual);
        end

        function diag = getDiagnosticFor(constraint,actual)
            import matlab.unittest.diagnostics.StringDiagnostic
            if constraint.sizeMatchesExpected(actual)
                diag = StringDiagnostic('HasSameSizeAs passed.');
            else
                diag = StringDiagnostic(sprintf(...
                    'HasSameSizeAs failed.\nActual Size: [%s]\nExpectedSize: [%s]',...
                    int2str(size(actual)),...
                    int2str(size(constraint.ValueWithExpectedSize))));
            end
        end
    end

    methods(Access = private)
        function bool = sizeMatchesExpected(constraint,actual)
            bool = isequal(size(actual),size(constraint.ValueWithExpectedSize));
        end
    end

end
```

See Also

`getDiagnosticFor`

Topics

“Create Custom Constraint”

matlab.unittest.constraints.ContainsSubstring class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying string containing substring

Construction

`ContainsSubstring(substring)` creates a constraint that specifies a string scalar or character vector containing `substring`. The constraint is satisfied only if the actual value contains an expected substring.

`ContainsSubstring(substring,Name,Value)` provides a constraint with additional options specified by one or more `Name,Value` pair arguments. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

`substring`

Text that must be contained within the actual value, specified as a character vector or string scalar. `substring` can include newline characters.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`IgnoringCase`

Indicator to ignore case, specified as `false` or `true` (`logical` `0` or `1`).

Default: `false`

`IgnoringWhitespace`

Indicator to ignore whitespace, specified as `false` or `true` (`logical` `0` or `1`). If `IgnoringWhitespace` is `true`, MATLAB removes any whitespace characters from both the actual value and the expected value before determining whether the constraint is satisfied.

Note When `IgnoringWhitespace` is `true`, the input argument `substring` must contain at least one non-whitespace character.

Default: `false`

`WithCount`

Number of times `substring` must be contained within the actual value, specified as a positive integer.

Properties

`IgnoreCase`

Indicator if the constraint is insensitive to case, specified in the name-value pair argument, `'IgnoringCase'`.

`IgnoreWhitespace`

Indicator if the constraint is insensitive to whitespace, specified in the name-value pair argument, `'IgnoringWhitespace'`.

`Substring`

Character vector or string scalar that must be included in the actual value, specified in the input argument, `substring`.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test That Actual Value Contains Specified Substring

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.ContainsSubstring

testCase = TestCase.forInteractiveUse;
```

Define the actual value.

```
actVal = 'This Is One Long Message';
```

Test that `actVal` contains the text `'One'`.

```
testCase.verifyThat(actVal, ContainsSubstring('One'))
```

```
Verification passed.
```

Test that `actVal` contains the text `'long'`.

```
testCase.verifyThat(actVal, ContainsSubstring('long'))
```

```
Verification failed.
    ----------------------
    Framework Diagnostic:
    ----------------------
    ContainsSubstring failed.
    --> The value does not contain the substring.

    Actual char:
        This Is One Long Message
    Expected Substring:
        long
```

By default, the `ContainsSubstring` constraint is case sensitive.

Repeat the test ignoring case.

```
testCase.verifyThat(actVal, ContainsSubstring('long', ...
    'IgnoringCase', true))
```

```
Verification passed.
```

Test that `actVal` contains the text `'is'` twice. For the test to pass, configure the constraint to ignore case.

```
testCase.verifyThat(actVal, ContainsSubstring('is', ...
    'WithCount', 2, 'IgnoringCase', true))
```

```
Verification passed.
```

Test that `actVal` contains the text `'thisisone'`. For the test to pass, configure the constraint to ignore whitespace and case.

```
testCase.verifyThat(actVal, ContainsSubstring('thisisone', ...
    'IgnoringCase', true, 'IgnoringWhitespace', true))
```

```
Verification passed.
```

Assert that `actVal` does not contain the text `'longer'`.

```
testCase.assertThat(actVal, ~ContainsSubstring('longer', ...
    'IgnoringCase', true))
```

```
Assertion passed.
```

See Also

`EndsWithSubstring` | `IsSubstringOf` | `Matches` | `StartsWithSubstring`

matlab.unittest.constraints.EndsWithSubstring class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying string ending with substring

Construction

`EndsWithSubstring(suffix)` creates a constraint specifying a string scalar or character vector ending with a substring. The constraint is satisfied only if the actual value ends with the expected text, `suffix`.

`EndsWithSubstring(suffix,Name,Value)` provides a constraint with additional options specified by one or more `Name,Value` pair arguments. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

`suffix`

Text that occurs at the end of the actual value, specified as a string scalar or character vector. `suffix` can include newline characters.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`IgnoringCase`

Indicator to ignore case, specified as `false` or `true` (`logical` `0` or `1`).

Default: `false`

`IgnoringWhitespace`

Indicator to ignore whitespace, specified as `false` or `true` (`logical 0` or `1`). If `IgnoringWhitespace` is `true`, MATLAB removes any whitespace characters from both the actual value and the expected value before determining whether the constraint is satisfied.

Note When `IgnoringWhitespace` is `true`, the input argument `suffix` must contain at least one non-whitespace character.

Default: `false`

Properties

`IgnoreCase`

Indicator if the constraint is insensitive to case, specified in the name-value pair argument, `'IgnoringCase'`.

`IgnoreWhitespace`

Indicator if the constraint is insensitive to whitespace, specified in the name-value pair argument, `'IgnoringWhitespace'`.

`Suffix`

Text that occurs at the end of the actual value, specified in the input argument, `suffix`.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test That Actual Value Ends with Specified Text

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.EndsWithSubstring

testCase = TestCase.forInteractiveUse;
```

Define the actual value.

```
actVal = 'This Is One Long Message';
```

Test that `actVal` ends with `'Message'`.

```
testCase.verifyThat(actVal, EndsWithSubstring('Message'))
```

```
Verification passed.
```

Test that `actVal` ends with `'AgE'`.

```
testCase.verifyThat(actVal, EndsWithSubstring('AgE'))
```

```
Verification failed.

----------------------
Framework Diagnostic:
----------------------
EndsWithSubstring failed.
--> The value does not end with the supplied suffix.

Actual char:
    This Is One Long Message
Expected Suffix:
    AgE
```

By default, the `EndsWithSubstring` constraint is case sensitive.

Repeat the test ignoring case.

```
testCase.verifyThat(actVal, EndsWithSubstring('AgE', ...
    'IgnoringCase', true))
```

```
Verification passed.
```

Test that `actVal` ends with `'longmessage'`. For the test to pass, configure the constraint to ignore whitespace and case.

```
testCase.verifyThat(actVal, EndsWithSubstring('longmessage', ...
    'IgnoringCase', true, 'IgnoringWhitespace', true))
```

```
Verification passed.
```

Assert that `actVal` does not end with `'long'`.

```
testCase.assertThat(actVal, ~EndsWithSubstring('long'))
```

```
Assertion passed.
```

See Also

`ContainsSubstring` | `IsSubstringOf` | `Matches` | `StartsWithSubstring`

matlab.unittest.constraints.Eventually class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.Constraint`

Poll for value to asynchronously satisfy constraint

Construction

`outConstObj = Eventually(constObj)` creates a constraint, `outConstObj`, that polls for an actual value returned from a function handle to asynchronously satisfy the `constObj` constraint. It is not satisfied if evaluation of the function handle does not produce a value that satisfies the constraint within 20 seconds. The testing framework invokes the `drawnow` function while the `Eventually` constraint waits for specified function to satisfy the constraint.

`outConstObj = Eventually(constObj,'WithTimeoutOf',timeOutVal)` creates a constraint that polls for the constraint to be satisfied within the timer period specified in `timeOutVal`.

Input Arguments

`constObj`

`Constraint` instance

`timeOutVal`

Maximum time to attempt to produce passing behavior, specified in seconds

Default: `20` seconds

Properties

FinalReturnValue

Output produced when the test framework invokes the supplied function handle. This property is read only and is set when the test framework invokes the function handle.

Timeout

Maximum time to attempt to produce passing behavior, specified by the `timeOutVal` input argument

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Verify Test Passes Eventually

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.Eventually
import matlab.unittest.constraints.IsGreaterThan
import matlab.unittest.constraints.IsLessThan

testCase = TestCase.forInteractiveUse;
```

Verify that, within the timeout period, a call to `toc` results in a value greater than 10 (seconds). The `Eventually` constraint repeatedly calls `toc` until either the constraint is satisfied or the elapsed time exceeds the timeout period. Repeated calls to `toc` result in the elapsed time since the last call to `tic`.

```
tic
testCase.verifyThat(@toc, Eventually(IsGreaterThan(10)))
```

```
Interactive verification passed.
```

The verification may take as long as 10 seconds for `toc` to reach a passing value. If you issue the call to `tic` and wait more than 10 seconds before issuing the `verifyThat` command, the verification returns immediately since `toc` already returns a value greater than 10.

Verify that, within the timeout period, `toc` does not return a negative value.

```
testCase.verifyThat(@toc, Eventually(IsLessThan(0)))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
Eventually failed.
--> The constraint never passed with a timeout of 20 second(s).
--> IsLessThan failed.
    --> The value must be less than the maximum value.

    Actual Value:
      36.532254706346720
    Maximum Value (Exclusive):
         0

Evaluated Function:
  function_handle with value:

    @toc
```

This failure is expected since elapsed time is not going to be less than zero. However, `Eventually` polls `toc` for the duration of the timeout period.

Adjust the timeout period so `Eventually` polls for 5 seconds.

```
tic
testCase.verifyThat(@toc, Eventually(IsGreaterThan(10), ...
    'WithTimeoutOf', 5))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
Eventually failed.
--> The constraint never passed with a timeout of 5 second(s).
```

```
--> IsGreaterThan failed.
    --> The value must be greater than the minimum value.

    Actual Value:
       5.143138452046230
    Minimum Value (Exclusive):
        10

Evaluated Function:
  function_handle with value:

    @toc
```

If you didn’t wait more than 5 seconds between calls to `tic` and `verifyThat`, the test fails because the elapsed time is not greater than 10 seconds within the modified timeout period.

See Also

`drawnow` | `matlab.unittest.constraints.Constraint`

matlab.unittest.constraints.EveryCellOf class

Package: `matlab.unittest.constraints`

Test if all elements of cell array meet constraint

Description

The `EveryCellOf` class creates a proxy of the actual value to the framework. The proxy enables a test writer to apply a constraint against each element of a cell array, which ensures that a passing result occurs when every element of the cell array satisfies the constraint.

It is intended that you use this class through `matlab.unittest` qualifications as shown in the examples. The class does not modify the provided actual value, but serves as a wrapper to perform the constraint analysis. The testing framework analyzes the constraint on an element-by-element basis.

Construction

`EveryCellOf(actVal)` creates a proxy instance that tests if every element of a provided cell array, `actVal`, meets a constraint. The test passes if all elements satisfy the constraint.

Tips

- `EveryCellOf` checks if every element in the provided cell array satisfies an associated constraint. However, there are some constraints, a prominent one being `IsEqualTo`, that natively validate if all elements in cell arrays satisfy a condition. In these situations, use of `EveryCellOf` is unnecessary and impedes qualification performance.

Input Arguments

actVal

Actual value to test against constraint

Properties

ActualValue

Actual value to test against constraint. Set this property through the constructor via the `actVal` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test That Every Cell Satisfies Constraint

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.EveryCellOf

testCase = TestCase.forInteractiveUse;
```

Test that every cell of `actVal` contains the substring `'ain'`.

```
import matlab.unittest.constraints.ContainsSubstring
actVal = {'Rain','Main','Plain'};
testCase.verifyThat(EveryCellOf(actVal), ContainsSubstring('ain'))
```

```
Interactive verification passed.
```

Test that every cell of the actual value array has two elements.

```
import matlab.unittest.constraints.HasElementCount
testCase.verifyThat(EveryCellOf({{'hello','world'}, {11 38}}), HasElementCount(2))
```

```
Interactive verification passed.
```

Test that every cell of the actual value array is empty.

```
import matlab.unittest.constraints.IsEmpty
testCase.verifyThat(EveryCellOf({inputParser.empty,''}), IsEmpty)
```

```
Interactive verification passed.
```

Test that every cell of the actual value array is finite.

```
import matlab.unittest.constraints.IsFinite
testCase.verifyThat(EveryCellOf({NaN, Inf, 5}), IsFinite)
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
At least one cell failed.

Failing indices:
         1     2
The first failing cell failed because:
--> IsFinite failed.
    --> The value must be finite.

    Actual Value:
           NaN

Actual Value Cell Array:
        [NaN]    [Inf]    [5]
```

Only the third element has a finite value.

Test that every cell of the actual value array is real.

```
import matlab.unittest.constraints.IsReal
testCase.verifyThat(EveryCellOf({1 4i}), IsReal)
```

```
Interactive verification failed.

----------------------
```

```
Framework Diagnostic:
----------------------
At least one cell failed.

Failing indices:
         2
The first failing cell failed because:
--> IsReal failed.
    --> The value must be real.

    Actual Value:
          0.000000000000000 + 4.000000000000000i

Actual Value Cell Array:
        [1]    [0.000000000000000 + 4.000000000000000i]
```

The second element has an imaginary value.

See Also

AnyCellOf | AnyElementOf | EveryElementOf | matlab.unittest.qualifications

matlab.unittest.constraints.EveryElementOf class

Package: `matlab.unittest.constraints`

Test if all elements of array meet constraint

Description

The `EveryElementOf` class creates a proxy of the actual value to the framework. The proxy enables a test writer to apply a constraint against each element of an array, which ensures that a passing result occurs when every element of the array that satisfies the constraint.

It is intended that you use this class through `matlab.unittest` qualifications as shown in the examples. The class does not modify the provided actual value, but serves as a wrapper to perform the constraint analysis. The testing framework analyzes the constraint on an element-by-element basis.

Construction

`EveryElementOf(actVal)` creates a proxy instance that tests if every element of a provided array, `actVal`, meets a constraint. The test passes if all elements satisfy the constraint.

Tips

- `EveryElementOf` checks if every element in the provided array satisfies an associated constraint. However, there are some constraints, such as `IsEqualTo` and `IsGreaterThan`, `IsLessThan`, that natively validate if all elements in the array satisfy a condition. In these situations, use of `EveryElementOf` is unnecessary and impedes qualification performance.

Input Arguments

actVal

Actual value to test against constraint

Properties

ActualValue

Actual value to test against constraint. Set this property through the constructor via the `actVal` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test That Every Element Satisfies Constraint

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.EveryElementOf

testCase = TestCase.forInteractiveUse;
```

Test that every element of `actVal` is less than 55.

```
import matlab.unittest.constraints.IsLessThan
actVal = [1 1 2 3 5 8 13 21 34];
testCase.verifyThat(EveryElementOf(actVal), IsLessThan(55))
```

```
Interactive verification passed.
```

Test that every element of the actual value array is complex.

```
import matlab.unittest.constraints.IsReal
testCase.verifyThat(EveryElementOf([1+2i 4i]), ~IsReal)
```

```
Interactive verification passed.
```

Test that every element of the actual value array is less than zero.

```
import matlab.unittest.constraints.IsLessThan
testCase.verifyThat(EveryElementOf([1 -5]), IsLessThan(0))
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
At least one element failed.

Failing indices:
         1
The first failing element failed because:
--> IsLessThan failed.
    --> The value must be less than the maximum value.

    Actual Value:
             1
    Maximum Value (Exclusive):
             0

Actual Value Array:
         1    -5
```

Only the second element is less than zero.

Test that every element of the actual value array has a `NaN` value.

```
import matlab.unittest.constraints.HasNaN
testCase.verifyThat(EveryElementOf([NaN 0/0 5]), HasNaN)
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
At least one element failed.
```

```
Failing indices:
         3
The first failing element failed because:
--> HasNaN failed.
    --> The value must be NaN.

    Actual Value:
             5

Actual Value Array:
       NaN   NaN     5
```

Only the third element has a `NaN` value.

See Also

`AnyCellOf` | `AnyElementOf` | `EveryCellOf` | `matlab.unittest.qualifications`

matlab.unittest.constraints.HasElementCount class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying expected number of elements

Construction

`HasElementCount(countVal)` provides a constraint that specifies an expected number of elements. The constraint is satisfied if the actual value array has the same number of elements specified as by `countVal`.

Input Arguments

`countVal`

Number of elements a value must have to satisfy the constraint.

Properties

`Count`

Number of elements a value must have to satisfy the constraint. Set this property through the constructor via the `countVal` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test for Expected Number of Elements

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.HasElementCount

testCase = TestCase.forInteractiveUse;
```

Verify a scalar has an element count of one.

```
testCase.verifyThat(3, HasElementCount(1))
```

```
Interactive verification passed.
```

Test the element count of the vector.

```
testCase.assertThat([42 7 13], HasElementCount(3))
```

```
Interactive assertion passed.
```

Test the element count of the matrix.

```
testCase.assertThat([1 2 3; 4 5 6], HasElementCount(5))
```

```
Interactive assertion failed.

---------------------
Framework Diagnostic:
---------------------
HasElementCount failed.
--> The value did not have the correct number of elements.

    Actual Number of Elements:
             6
    Expected Number of Elements:
             5

Actual Value:
         1     2     3
         4     5     6
Assertion failed.
```

The matrix has six elements.

Test that a square identity matrix has the correct number of elements.

```
n = 7;
testCase.assumeThat(eye(n), HasElementCount(n^2))
```

```
Interactive assumption passed.
```

Verify the element count of a cell array of character vectors.

```
testCase.verifyThat({'someText', 'moreText'}, HasElementCount(2))
```

```
Interactive verification passed.
```

Test the element count of a structure.

```
s.Field1 = 1;
s.Field2 = 2;
testCase.verifyThat(s, HasElementCount(2))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
HasElementCount failed.
--> The value did not have the correct number of elements.

    Actual Number of Elements:
             1
    Expected Number of Elements:
             2

Actual Value:
        Field1: 1
        Field2: 2
```

The structure has two fields, but it only has one element.

```
testCase.verifyThat(s, HasElementCount(1))
```

```
Interactive verification passed.
```

See Also

`HasLength` | `HasSize` | `IsEmpty` | `numel`

matlab.unittest.constraints.HasField class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying structure containing particular field

Construction

`HasField(fieldname)` provides a constraint specifying structure containing particular field, `fieldname`. The constraint is satisfied if the actual value is a structure and that structure contains a field named `fieldname`.

Input Arguments

`fieldname`

Name of the field that a structure must contain to satisfy the constraint, specified as a character vector or a string scalar.

Properties

`Field`

Name of the field that a structure must contain to satisfy the constraint. Set this property through the constructor via the `fieldname` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test That Structure Has Particular Field

Create a `TestCase` for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.HasField

testCase = TestCase.forInteractiveUse;
```

Define the following structure, `S`, with two fields.

```
S = struct('Tag', 123, 'Serial', 345);
```

Verify that the structure has a `'Tag'` field.

```
testCase.verifyThat(S, HasField('Tag'))
```

```
Interactive verification passed.
```

Verify that the structure has a `'tag'` field.

```
testCase.verifyThat(S, HasField('tag'))
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
HasField failed.
--> The value did not have the expected field.

    Actual Fieldnames:
            'Tag'
            'Serial'
    Expected Fieldname:
            'tag'

Actual Value:
           Tag: 123
        Serial: 345
```

The verification fails because the field name comparison is case sensitive.

Verify that the structure has a `'Tag'` field.

```
testCase.verifyThat(S, HasField('Tag'))
```

```
Interactive verification passed.
```

Verify that the structure has both a `'Tag'` and a `'Serial'` field.

```
testCase.verifyThat(S, HasField('Tag') & HasField('Serial'))
```

```
Interactive verification passed.
```

Verify that the structure does not have a `'Name'` field.

```
testCase.verifyThat(S, ~HasField('Name'))
```

```
Interactive verification passed.
```

See Also

matlab.unittest.constraints.HasInf class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying array containing any infinite value

Construction

`HasInf` creates a constraint that is able to determine if any value of an actual value array is an infinite value. This constraint is satisfied only if the actual value array contains at least one infinite value.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test That Actual Value Array Contains Infinite Value

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.HasInf

testCase = TestCase.forInteractiveUse;
```

Test that the value `Inf` satisfies the constraint.

```
testCase.verifyThat(Inf, HasInf)
```

```
Interactive verification passed.
```

Assert that an array contains an infinite value.

```
testCase.assertThat([0 1 1 2 3 5 8 13], HasInf)
```

```
Interactive assertion failed.

---------------------
Framework Diagnostic:
---------------------
HasInf failed.
--> At least one element must be Inf or -Inf.

Actual double:
         0     1     1     2     3     5     8    13
Assertion failed.
```

The array does not contain any infinite values.

Verify that an array contains an infinite value.

```
testCase.verifyThat([-Inf 5 NaN], HasInf)
```

```
Interactive verification passed.
```

Assert that a complex number that is infinite in the imaginary part satisfies the constraint.

```
testCase.assertThat(42+Inf*1i, HasInf)
```

```
Interactive verification passed.
```

Verify that an array does not contain any infinite values.

```
testCase.verifyThat([NaN -7+NaN*1i], ~HasInf)
```

```
Interactive verification passed.
```

Negating the `HasInf` constraint does not ensure the value is finite, only that it does not contain any infinite values.

See Also

`HasNaN` | `IsFinite` | `isinf`

matlab.unittest.constraints.HasLength class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying expected length of array

Construction

`HasLength(lengthVal)` provides a constraint that specifies an expected length of an array. The constraint is satisfied if the largest dimension length of the actual value array has the same number of elements specified as by `lengthVal`.

Input Arguments

`lengthVal`

Length a value must have to satisfy the constraint.

Properties

`Count`

Length a value must have to satisfy the constraint. Set this property through the constructor via the `lengthVal` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test for Expected Array Length

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.HasLength

testCase = TestCase.forInteractiveUse;
```

Assert that a 2x5x3 array has an expected length.

```
testCase.assertThat(rand(2, 5, 3), HasLength(5))
```

```
Interactive assertion passed.
```

Verify that a cell array of character vectors has an expected length.

```
testCase.verifyThat({'SomeString', 'SomeOtherString'}, HasLength(2))
```

```
Interactive verification passed.
```

Verify that an identity matrix has an expected length.

```
testCase.verifyThat(eye(2), HasLength(4))
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
HasLength failed.
--> The array has an incorrect length.

    Actual Length:
             2
    Expected Length:
             4

Actual Array:
         1     0
         0     1
```

The matrix has a length of 2.

See Also

`HasElementCount` | `HasSize` | `IsEmpty` | `length`

matlab.unittest.constraints.HasNaN class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying array containing NaN value

Construction

`HasNaN` creates a constraint that is able to determine if any value of an actual value array is `NaN`. This constraint is satisfied only if the actual value array contains at least one `NaN` value.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test That Actual Value Array Contains NaN Value

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.HasNaN

testCase = TestCase.forInteractiveUse;
```

Test that the value `NaN` satisfies the constraint.

```
testCase.verifyThat(NaN, HasNaN)
```

```
Interactive verification passed.
```

Assert that an array contains a NaN value.

```
testCase.assertThat([0 1 1 2 3 5 8 13], HasNaN)
```

```
Interactive assertion failed.

---------------------
Framework Diagnostic:
---------------------
HasNaN failed.
--> At least one element must be NaN.

Actual double:
         0     1     1     2     3     5     8    13
Assertion failed.
```

The array does not contain a NaN value.

Verify that an array contains a NaN value.

```
testCase.verifyThat([-Inf 5 NaN], HasNaN)
```

```
Interactive verification passed.
```

Assert that a complex number satisfies the constraint.

```
testCase.assertThat(42+NaN*1i, HasNaN)
```

```
Interactive assertion passed.
```

Verify that an array does not contain any `NaN` values.

```
testCase.verifyThat([Inf -7+Inf*1i], ~HasNaN)
```

```
Interactive verification passed.
```

Negating the `HasNaN` constraint does not ensure the value is finite, only that it does not contain any `NaN` values.

See Also

`HasInf` | `IsFinite` | `isnan`

matlab.unittest.constraints.HasSize class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying expected size of array

Construction

`HasSize(sizeVal)` provides a constraint that specifies an expected size of an array. The constraint is satisfied if the actual value array size is equal to the size specified by `sizeVal`.

Input Arguments

`sizeVal`

Size a value must have to satisfy the constraint.

Properties

`Size`

Size a value must have to satisfy the constraint. Set this property through the constructor via the `sizeVal` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test for Expected Array Size

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.HasSize

testCase = TestCase.forInteractiveUse;
```

Assert that a 2x5x3 array has an expected size.

```
testCase.assertThat(rand(2, 5, 3), HasSize([2 5 3]))
```

```
Interactive assertion passed.
```

Verify that a cell array of character vectors has an expected size.

```
testCase.verifyThat({'SomeText', 'SomeOtherText'}, HasSize([1 2]))
```

```
Interactive verification passed.
```

Verify that an identity matrix has an expected size.

```
testCase.verifyThat(eye(2), HasSize([4 1]))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
HasSize failed.
--> The value had an incorrect size.

    Actual Size:
             2     2
    Expected Size:
             4     1

Actual Value:
         1     0
         0     1
```

The matrix has a size of `2x2`.

See Also

`HasElementCount` | `HasLength` | `IsEmpty` | `size`

matlab.unittest.constraints.HasUniqueElements class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying set contains unique elements

Construction

`HasUniqueElements` creates a constraint specifying that a set contains unique elements. The constraint produces a qualification failure for any actual value set that does not contain unique elements. If `numel(unique(actualSet))` is equal to `numel(actualSet)`, the constraint considers the actual value set to have unique elements.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test for Unique Elements

Create a test case for interactive testing.

```
import matlab.unittest.TestCase;
import matlab.unittest.constraints.HasUniqueElements;

testCase = TestCase.forInteractiveUse;
```

Test `'abc'` and `'Mississippi'` for unique elements.

```
testCase.verifyThat('abc', HasUniqueElements);
```

```
Interactive verification passed.
```

```
testCase.verifyThat('Mississippi', HasUniqueElements);
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
HasUniqueElements failed.
--> The value contains 3 nonunique element(s):
    --> Nonunique element found at indices [2 5 8 11]:
            i
    --> Nonunique element found at indices [9 10]:
            p
    --> Nonunique element found at indices [3 4 6 7]:
            s

Actual char:
    Mississippi
```

Assert that a set of doubles has unique elements.

```
testCase.assertThat(magic(6), HasUniqueElements);
```

```
Interactive assertion passed.
```

```
testCase.assertThat(abs(-3:3), HasUniqueElements);
```

```
Interactive assertion failed.

----------------------
Framework Diagnostic:
----------------------
HasUniqueElements failed.
--> The value contains 3 nonunique element(s):
    --> Nonunique element found at indices [3 5]:
                1
    --> Nonunique element found at indices [2 6]:
                2
    --> Nonunique element found at indices [1 7]:
                3

Actual double:
         3     2     1     0     1     2     3
Assertion failed.
```

Verify that a cell array has unique elements.

```
testCase.verifyThat({'abc','123';'abc','345'},HasUniqueElements);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
HasUniqueElements failed.
--> The value contains 1 nonunique element(s):
    --> Nonunique element found at indices [1 2]:
                'abc'

Actual cell:
        'abc'    '123'
        'abc'    '345'
```

Assert that a table has unique rows.

```
t = table([3;3;5],{'A';'C';'E'},logical([1;0;0]));
testCase.assertThat(t,HasUniqueElements);
```

```
Interactive assertion passed.
```

See Also

`matlab.unittest.constraints` | `matlab.unittest.constraints.IsSameSetAs` | `unique`

Introduced in R2016a

matlab.unittest.constraints.IsAnything class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.Constraint`

Constraint specifying any value

Construction

`IsAnything` provides a constraint specifying any value. The constraint is satisfied by any value. It is the default constraint for selectors that do not require an input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Instantiate `IsAnything` Object

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsAnything

testCase = TestCase.forInteractiveUse;
```

Verify that the following values satisfy the `IsAnything` constraint: `NaN`, an `inputParser` object, a numeric array, and a complex number.

```
testCase.verifyThat(NaN, IsAnything)
testCase.verifyThat(inputParser, IsAnything)
testCase.verifyThat(1:10, IsAnything)
testCase.verifyThat(-Inf+5j, IsAnything)
```

```
Interactive verification passed.
Interactive verification passed.
```

```
Interactive verification passed.
Interactive verification passed.
```

Test that empty cells, arrays, and character vectors satisfy the `IsAnything` constraint.

```
testCase.verifyThat({}, IsAnything)
testCase.verifyThat([], IsAnything)
testCase.verifyThat('', IsAnything)
```

```
Interactive verification passed.
Interactive verification passed.
Interactive verification passed.
```

The constraint is satisfied even though the data are empty.

See Also

`matlab.unittest.selectors`

matlab.unittest.constraints.IsEmpty class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying empty value

Construction

`IsEmpty` provides a constraint that specifies an empty value. The constraint is satisfied if the actual value array is empty.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test That Actual Value Is Empty

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEmpty

testCase = TestCase.forInteractiveUse;
```

Verify that an empty character vector satisfies the `IsEmpty` constraint.

```
testCase.verifyThat('', IsEmpty)
```

```
Interactive verification passed.
```

Assert that a vector is not empty.

```
testCase.assertThat([13 42], ~IsEmpty)
```

```
Interactive verification passed.
```

Verify that a matrix with a dimension of length zero is empty.

```
testCase.verifyThat(rand(2, 5, 0, 3), IsEmpty)
```

```
Interactive verification passed.
```

Assert that an empty object satisfies the `IsEmpty` constraint.

```
testCase.assertThat(MException.empty, IsEmpty)
```

```
Interactive assertion passed.
```

Verify that a cell array containing an empty numeric array is empty.

```
testCase.verifyThat({[]}, IsEmpty)
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
IsEmpty failed.
--> The value must be empty.
--> The value has a size of [1  1].

Actual Value:
        {[]}
```

The cell array is not empty, even though the only thing it contains is an empty array.

See Also

`HasCount` | `HasLength` | `HasSize` | `isempty`

matlab.unittest.constraints.IsFalse class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.Constraint`

Constraint specifying false value

Construction

`IsFalse` provides a constraint specifying a false value. This constraint is satisfied only by a scalar logical with a value of `false`.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test Actual Value Is False

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsFalse
```

```
testCase = TestCase.forInteractiveUse;
```

Test that `false` satisfies the `IsFalse` constraint.

```
testCase.verifyThat(false, IsFalse)
```

```
Interactive verification passed.
```

Test that the `IsFalse` constraint is not satisfied by `true`.

```
testCase.verifyThat(true, IsFalse)
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
IsFalse failed.
--> The value must evaluate to "false".

Actual Value:
         1
```

The test fails because `true` returns `logical(1)`.

Test that the `IsFalse` constraint is not satisfied by the double `0`.

```
testCase.verifyThat(0, IsFalse)
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
IsFalse failed.
--> The value must be logical. It is of type "double".

Actual Value:
         0
```

The `IsFalse` constraint is satisfied only by `logical(0)`.

Test that the `IsFalse` constraint is not satisfied by a logical array of zeros.

```
testCase.verifyThat([false false false], IsFalse)
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
IsFalse failed.
--> The value must be scalar. It has a size of [1  3].

Actual Value:
         0     0     0
```

The `IsFalse` constraint is only satisfied if the value is scalar and `logical(0)`.

See Also

`IsTrue`

matlab.unittest.constraints.IsFile class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying value points to file

Construction

`IsFile` creates a constraint specifying that a value is a string scalar or character vector that points to an existing file. The constraint is satisfied if the value is an absolute or relative path to an existing file.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test If File Exists

Create a test case for interactive testing.

```
import matlab.unittest.TestCase;
import matlab.unittest.constraints.IsFile;

testCase = TestCase.forInteractiveUse;
```

Test if `myFile.mat` is an existing file in your current working folder. This example assumes that the file does not exist and the test fails.

```
act = 'myFile.mat';
testCase.verifyThat(act,IsFile)
```

```
Interactive verification failed.

----------------------
```

```
Framework Diagnostic:
---------------------
IsFile failed.
--> Value does not point to an existing file.
--> Current folder during evaluation:
        'C:\work'

Actual char:
    myFile.mat
```

In your current working folder, create a folder `myFolder` that contains a file `myTxtFile.txt`.

```
mkdir myFolder
dlmwrite(['myFolder' filesep 'myTxtFile.txt'],rand(5))
```

Verify that `myTxtFile.txt` is an existing file in `myFolder`.

```
act = ['myFolder' filesep 'myTxtFile.txt'];
testCase.verifyThat(act,IsFile)
```

```
Interactive verification passed.
```

Test That File Does Not Exist

Create a test case for interactive testing.

```
import matlab.unittest.TestCase;
import matlab.unittest.constraints.IsFile;

testCase = TestCase.forInteractiveUse;
```

Verify that `nonexistentFile.mat` is not an existing file in your current working folder. This example assumes that the file does not exist and the test passes.

```
act = 'nonexistentFile.mat';
testCase.verifyThat(act,~IsFile)
```

```
Interactive verification passed.
```

See Also

`matlab.unittest.constraints.IsFolder`

Introduced in R2018a

matlab.unittest.constraints.IsFinite class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying finite value

Construction

`IsFinite` creates a constraint that is able to determine if all values of an actual value array are finite. This constraint is satisfied only if the actual value array does not contain any infinite or `NaN` values.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test That Actual Value Array Contains Only Finite Values

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsFinite

testCase = TestCase.forInteractiveUse;
```

Test that the value 17 satisfies the constraint.

```
testCase.verifyThat(17, IsFinite)
```

```
Interactive verification passed.
```

Assert that an array is completely finite.

```
testCase.assertThat([0 1 1 2 3 5 8 13], IsFinite)
```

```
Interactive assertion passed.
```

Verify that an array is completely finite.

```
testCase.verifyThat([-Inf 5 NaN], IsFinite)
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsFinite failed.
--> All elements must be finite-valued.
    Failing indices:
              1     3

Actual Value:
      -Inf     5   NaN
```

The array contains an infinite value.

Test if a complex number that is infinite in the imaginary part satisfies the constraint.

```
testCase.assertThat(42+Inf*1i, IsFinite)
```

```
Interactive assertion failed.

---------------------
Framework Diagnostic:
---------------------
IsFinite failed.
--> The value must be finite.

Actual Value:
     42.000000000000000 +               Infi
Assertion failed.
```

Verify that an array does not contain all finite values.

```
testCase.verifyThat([NaN -7+NaN*1i], ~IsFinite)
```

```
Interactive verification passed.
```

See Also

HasInf | HasNaN | isfinite

matlab.unittest.constraints.IsFolder class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying value points to folder

Construction

`IsFolder` creates a constraint specifying that a value is a string scalar or character vector that points to an existing folder. The constraint is satisfied if the value is an absolute or relative path to an existing folder.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test If Folder Exists

Create a test case for interactive testing.

```
import matlab.unittest.TestCase;
import matlab.unittest.constraints.IsFolder;

testCase = TestCase.forInteractiveUse;
```

Test if `myFolder` is an existing folder in your current working folder. This example assumes that the folder does not exist and the test fails.

```
act = 'myFolder';
testCase.verifyThat(act,IsFolder)
```

```
Interactive verification failed.

---------------------
```

```
Framework Diagnostic:
---------------------
IsFolder failed.
--> Value does not point to an existing folder.
--> Current folder during evaluation:
        'C:\work'

Actual char:
    myFolder
```

In your current working folder, create a folder `myFolder`.

```
mkdir myFolder
```

Verify that `myFolder` is an existing folder in your current working folder.

```
act = 'myFolder';
testCase.verifyThat(act,IsFolder)
```

```
Interactive verification passed.
```

Test That Folder Does Not Exist

Create a test case for interactive testing.

```
import matlab.unittest.TestCase;
import matlab.unittest.constraints.IsFolder;

testCase = TestCase.forInteractiveUse;
```

Verify that `nonexistentFolder` is not an existing folder in your current working folder. This example assumes that the folder does not exist and the test passes.

```
act = 'nonexistentFolder';
testCase.verifyThat(act,~IsFolder)
```

```
Interactive verification passed.
```

See Also

`matlab.unittest.constraints.IsFile`

Introduced in R2018a

matlab.unittest.constraints.IsGreaterThan class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.Constraint`

Constraint specifying value greater than another value

Construction

`IsGreaterThan(floorVal)` creates a constraint specifying that an actual value is greater than another value. The constraint is satisfied if the actual value array is greater than the specified floor value, `floorVal`. The actual value is greater than `floorVal` only if the result of the expression `actual > floorVal` is nonempty and all values are true.

Input Arguments

`floorVal`

Largest value that fails the constraint.

Properties

`FloorValue`

Largest value that fails the constraint. Set this property through the constructor via the `floorVal` input argument.

Methods

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test That Actual Value Is Greater Than Provided Floor Value

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsGreaterThan

testCase = TestCase.forInteractiveUse;
```

Test that the actual value is greater than two.

```
actVal = 3;
testCase.verifyThat(actVal, IsGreaterThan(2))
```

```
Interactive verification passed.
```

Test that the actual value is greater than three.

```
testCase.verifyThat(actVal, IsGreaterThan(3))
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
IsGreaterThan failed.
--> The value must be greater than the minimum value.

Actual double:
         3
Minimum Value (Exclusive):
         3
```

The actual value is equal to, not greater than, three.

Test that each element in the actual value array is greater than four.

```
actVal = [5 6 7];
testCase.verifyThat(actVal, IsGreaterThan(4))
```

```
Interactive verification passed.
```

Test that each element in the actual value matrix is greater than four.

```
actVal = [1 2 3; 4 5 6];
testCase.verifyThat(actVal, IsGreaterThan(4))
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
IsGreaterThan failed.
--> Each element must be greater than the minimum value.

    Failing Indices:
             1     2     3     5

Actual double:
         1     2     3
         4     5     6
Minimum Value (Exclusive):
         4
```

The matrix contains four elements with a value less than or equal to four.

Test that the actual value, 5, is greater than every element in an array.

```
testCase.verifyThat(5, IsGreaterThan([1 2 3]))
```

```
Interactive verification passed.
```

Test that elements in the actual value array are greater than the corresponding floor values.

```
testCase.verifyThat([5 -3 2], IsGreaterThan([4 -9 0]))
```

```
Interactive verification passed.
```

Repeat the test, this time negating the first actual value element.

```
testCase.verifyThat([-5 -3 2], IsGreaterThan([4 -9 0]))
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
```

```
IsGreaterThan failed.
--> Each element must be greater than each corresponding element of the minimum value a

    Failing Indices:
             1

Actual double:
        -5    -3     2
Minimum Value (Exclusive):
         4    -9     0
```

The negated element is less than four.

See Also

`IsGreaterThanOrEqualTo` | `IsLessThan` | `IsLessThanOrEqualTo` | `gt`

matlab.unittest.constraints.IsGreaterThanOrEqualTo class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.Constraint`

Constraint specifying value greater than or equal to another value

Construction

`IsGreaterThanOrEqualTo(floorVal)` creates a constraint specifying that an actual value is greater than or equal to another value. The constraint is satisfied if the actual value array is greater than or equal to the specified floor value, `floorVal`. The actual value is greater than or equal to `floorVal` only if the result of the expression `actual >= floorVal` is nonempty and all values are true.

Input Arguments

`floorVal`

Minimum value to satisfy the constraint.

Properties

`FloorValue`

Minimum value to satisfy the constraint. Set this property through the constructor via the `floorVal` input argument.

Methods

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test That Actual Value Is Greater Than or Equal to Provided Floor Value

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsGreaterThanOrEqualTo

testCase = TestCase.forInteractiveUse;
```

Test that the actual value is greater than or equal to two.

```
actVal = 3;
testCase.verifyThat(actVal, IsGreaterThanOrEqualTo(2))
```

```
Interactive verification passed.
```

Test that the actual value is greater than or equal to three.

```
testCase.verifyThat(actVal, IsGreaterThanOrEqualTo(3))
```

```
Interactive verification passed.
```

Test that each element in the actual value array is greater than or equal to four.

```
actVal = [5 6 7];
testCase.verifyThat(actVal, IsGreaterThanOrEqualTo(4))
```

```
Interactive verification passed.
```

Test that each element in the actual value matrix is greater than or equal to four.

```
actVal = [1 2 3; 4 5 6];
testCase.verifyThat(actVal, IsGreaterThanOrEqualTo(4))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsGreaterThanOrEqualTo failed.
--> Each element must be greater than or equal to the minimum value.

    Failing Indices:
             1     3     5

Actual double:
         1     2     3
         4     5     6
Minimum Value (Inclusive):
         4
```

The matrix contains three elements that are greater than or equal to four.

Test that the actual value, 5, is greater than or equal to every element in an array.

```
testCase.verifyThat(5, IsGreaterThanOrEqualTo([1 2 3 5]))
```

```
Interactive verification passed.
```

Test that elements in the actual value array are greater than or equal to the corresponding floor values.

```
testCase.verifyThat([5 -3 0], IsGreaterThanOrEqualTo([4 -9 0]))
```

```
Interactive verification passed.
```

Repeat the test, this time negating the first actual value element.

```
testCase.verifyThat([-5 -3 0], IsGreaterThanOrEqualTo([4 -9 0]))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsGreaterThanOrEqualTo failed.
--> Each element must be greater than or equal to each corresponding element of the mi

    Failing Indices:
```

```
             1

Actual double:
        -5    -3     0
Minimum Value (Inclusive):
         4    -9     0
```

The negated element is less than or equal to four.

See Also

`IsGreaterThan` | `IsLessThan` | `IsLessThanOrEqualTo` | `ge`

matlab.unittest.constraints.IsEqualTo class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

General constraint to compare for equality

Description

The `IsEqualTo` class creates a constraint that compares data for equality. The type of comparison it uses is governed by the data type of the expected value. First, the testing framework checks if the expected value is an object. This check is performed first because it is possible for the object to have overridden methods that are used in subsequent checks (e.g. `islogical`). The following list categorizes and describes the various tests.

Data Type	Equality Comparison Method
MATLAB & Java Objects	If the expected value is a MATLAB or Java object, the `IsEqualTo` constraint calls the `isequaln` method if it is defined on the expected value object, otherwise it calls `isequal`. If the check returns false and a supported tolerance is specified, the `IsEqualTo` constraint checks the actual and expected values for equivalent class, size, and sparsity before determining if the values are within the tolerance. If the constraint can determine that the actual and expected values are equal because they are of the same class and size and all properties are equal, then `IsEqualTo` does not call `isequal` or `isequaln`.
Logicals	If the expected value is a `logical`, the constraint checks the actual and expected values for equivalent sparsity. If the sparsity matches, the constraint compares the values with the `isequal` method. Otherwise, the constraint is not satisfied.

Data Type	Equality Comparison Method
Numerics	If the expected value is `numeric`, the constraint checks the actual and expected values for equivalent class, size, and sparsity. If the all these checks match, the constraint uses the `isequaln` method for comparison. If `isequaln` returns `true`, the constraint is satisfied. If the complexity does not match or `isequaln` returns `false`, and a supported tolerance is supplied, the constraint uses the tolerance in the comparison. Otherwise, the constraint is not satisfied.
Strings	If the expected value is a `string`, the constraint uses the `strcmp` function to check the actual and expected values for equality. However, if the `IgnoreCase` property is true, the strings are compared using `strcmpi`. If the `IgnoreWhitespace` is `true`, all whitespace characters are removed from the actual and expected strings before passing them to `strcmp` or `strcmpi`.
Structures	If the expected value is a `struct`, the constraint compares the field count of the actual and expected values. If not equal, the constraint is not satisfied. Otherwise, each field of the expected value `struct` must exist on the actual value `struct`. If any field names are different, the constraint is not satisfied. Then, the constraint recursively compares the fields in a depth first examination. The recursion continues until a fundamental data type is encountered (that is, logical, numeric, string, or object), and then the values are compared as described above.
Cell Arrays	If the expected value is a cell array, the constraint checks the actual and expected values for size equality. If they are not equal in size, the constraint is not satisfied. Otherwise, each element of the array is recursively compared in a manner identical to fields in a structure, described above.
Tables	If the expected value is a `table`, the actual and expected values are checked for class equality, size equality, and for equal table properties. If they are not equal in class, size, or table properties, the constraint is not satisfied. Then, the constraint compares the size and type of each column variable and recursively compares each row of the table in a depth first examination. The recursion continues until a fundamental data type is encountered (that is, logical, numeric, string, or object), and then the values are compared as described above.

Construction

`IsEqualTo(expVal)` provides a general constraint to compare for equality.

`IsEqualTo(expVal,Name,Value)` provides a constraint with additional options specified by one or more `Name,Value` pair arguments. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

`expVal`

The expected value that is compared to the actual value.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`IgnoringCase`

Indicator if the constraint is insensitive to case, specified as `false` or `true` (`logical` `0` or `1`)

Default: `false`

`IgnoringFields`

Fields to ignore during struct comparison, specified as a cell array of character vectors or a string array.

Default: (empty)

`IgnoringWhitespace`

Indicator if the constraint is insensitive to whitespace, specified as `false` or `true` (`logical` `0` or `1`)

Default: `false`

`Using`

Particular comparator to use for constraint construction, specified as a `matlab.unittest.constraints.Comparator` object

Default: (empty)

`Within`

Tolerance to use in constraint construction, specified as a `matlab.unittest.constraints.Tolerance` object

Default: (empty)

Properties

`Comparator`

Specific comparator used in construction of the constraint, specified as a `matlab.unittest.constraints.Comparator` object in the name-value pair argument, `'Using'`.

`Expected`

The expected value that is compared to the actual value specified in the `expVal` input argument.

`IgnoreCase`

Indicator if the constraint is insensitive to case, specified in the name-value pair argument, `'IgnoringCase'`. This property applies at all levels of recursion, such as nested structures.

`IgnoredFields`

Fields to ignore during struct comparison, specified in the name-value pair argument, `'IgnoringFields'`. This property applies at all levels of recursion, such as nested structures.

IgnoreWhitespace

Indicator if the constraint is insensitive to whitespace, specified in the name-value pair argument, `'IgnoringWhitespace'`. This property applies at all levels of recursion, such as nested structures.

Tolerance

Specific tolerance used in construction of the constraint, specified as a `matlab.unittest.constraints.Tolerance` object in the name-value pair argument, `'Within'`. This property applies at all levels of recursion, such as nested structures.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test Numerics for Equality

Create a `TestCase` for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.AbsoluteTolerance
import matlab.unittest.constraints.RelativeTolerance

testCase = TestCase.forInteractiveUse;
```

Verify that an actual value of 5 is equal to the expected value.

```
expVal = 5;
testCase.verifyThat(5,IsEqualTo(expVal))
```

```
Verification passed.
```

Assume that the actual value is 4.95. Verify that the difference between the actual value and expected value is less than 0.09.

```
testCase.verifyThat(4.95,IsEqualTo(expVal,'Within',AbsoluteTolerance(0.09)))
```

```
Verification passed.
```

Assume that the actual value is 4.9. Verify that the difference between the actual and expected value is less than 1%.

```
testCase.verifyThat(4.9,IsEqualTo(expVal,'Within',RelativeTolerance(0.01)))
```

```
Verification failed.
    ----------------------
    Framework Diagnostic:
    ----------------------
    IsEqualTo failed.
    --> NumericComparator failed.
        --> The numeric values are not equal using "isequaln".
        --> RelativeTolerance failed.
            --> The error was not within relative tolerance.
            --> Failure table:
                    Actual    Expected           Error              RelativeError
                    ______    ________    ____________________    ____________________
                     4.9         5        -0.0999999999999996    -0.0199999999999999

        Actual Value:
           4.900000000000000
        Expected Value:
             5
```

The two values differ by more than 1%.

Test Floating Point Calculation with Tolerance

Create a `TestCase` for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.RelativeTolerance

testCase = TestCase.forInteractiveUse;
```

Test that `0.1*3 = 0.3`.

```
act = 0.1*3;
exp = 0.3;
testCase.verifyThat(act, IsEqualTo(exp))
```

```
Verification failed.
    ----------------------
    Framework Diagnostic:
    ----------------------
    IsEqualTo failed.
    --> NumericComparator failed.
        --> The numeric values are not equal using "isequaln".
        --> Failure table:
                Actual    Expected           Error               RelativeError
                ______    ________    ____________________    ____________________
                 0.3        0.3       5.55111512312578e-17    1.85037170770859e-16

        Actual Value:
           0.300000000000000
        Expected Value:
           0.300000000000000
```

This test fails due to round off error in floating point arithmetic.

Perform the comparison of floating point numbers using a tolerance. Test that `0.1*3 = 0.3` within a relative tolerance of `2*eps`.

```
testCase.verifyThat(act, IsEqualTo(exp, ...
    'Within', RelativeTolerance(2*eps)))
```

```
Verification passed.
```

Test Strings for Equality

Create a `TestCase` for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo

testCase = TestCase.forInteractiveUse;
```

Verify that two character vectors are equal.

```
expVal = 'Hello';
testCase.verifyThat('Hello',IsEqualTo(expVal))
```

```
Verification passed.
```

Change the case of the actual value and test for equality.

```
testCase.verifyThat('hello',IsEqualTo(expVal))
```

```
Verification failed.

----------------------
Framework Diagnostic:
----------------------
IsEqualTo failed.
--> StringComparator failed.
    --> The character arrays are not equal.

    Actual char:
        hello
    Expected char:
        Hello
```

Ignore case and test again.

```
testCase.verifyThat('hello',IsEqualTo(expVal,'IgnoringCase',true))
```

```
Verification passed.
```

Ignore whitespace and test two character vectors.

```
expVal = 'a bc';
testCase.verifyThat('abc',IsEqualTo(expVal,'IgnoringWhitespace',true))
testCase.verifyThat('ab c',IsEqualTo(expVal,'IgnoringWhitespace',true))
```

```
Verification passed.
Verification passed.
```

Test Objects for Equality Using Comparator

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.RelativeTolerance
import matlab.unittest.constraints.PublicPropertyComparator

testCase = TestCase.forInteractiveUse;
```

Define actual and expected `timeseries` objects. Perturb one of the actual data points by 1%.

```
expected = timeseries(1:10);
actual = expected;
actual.Data(7) = 1.01*actual.Data(7);
```

Test that the actual and expected values are equal within a relative tolerance of 2%.

```
testCase.verifyThat(actual, IsEqualTo(expected,...
    'Within', RelativeTolerance(.02)))
```

```
Verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> ObjectComparator failed.
    --> The objects are not equal using "isequal".
    --> The tolerance was ignored. The tolerance as specified does not support comparisons of timeseries values.

    Actual timeseries:
          timeseries

          Common Properties:
                    Name: 'unnamed'
                    Time: [10x1 double]
                TimeInfo: [1x1 tsdata.timemetadata]
                    Data: [1x1x10 double]
                DataInfo: [1x1 tsdata.datametadata]

          More properties, Methods
    Expected timeseries:
          timeseries

          Common Properties:
                    Name: 'unnamed'
                    Time: [10x1 double]
                TimeInfo: [1x1 tsdata.timemetadata]
                    Data: [1x1x10 double]
                DataInfo: [1x1 tsdata.datametadata]

          More properties, Methods
```

Use the `PublicPropertyComparator` in the construction of the constraint.

```
testCase.verifyThat(actual, IsEqualTo(expected,...
    'Within', RelativeTolerance(.02),...
    'Using', PublicPropertyComparator.supportingAllValues))
```

```
Interactive verification passed.
```

The test passes because the `PublicPropertyComparator` compares each public property individually instead of comparing the object all at once. In the former test, the `ObjectComparator` compares `timeseries` objects, and therefore relies on the `isequal` method of the `timeseries` class. Due to the perturbation in the actual

`timeseries`, `isequal` returns `false`. The comparator does not apply the tolerance because the double-valued tolerance cannot apply directly to the `timeseries` object. In the latter test, the comparator applies the tolerance to each public property that contains double-valued data.

See Also

`matlab.unittest.constraints.Constraint` | `matlab.unittest.constraints.Tolerance`

Introduced in R2013a

matlab.unittest.constraints.IsInstanceOf class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying inclusion in given class hierarchy

Construction

`IsInstanceOf(class)` provides a constraint specifying inclusion in a given class hierarchy. The constraint is satisfied if the actual value instance passes the “isa” relationship with `class` .

Input Arguments

`class`

Class name that the actual value must derive from or be an instance of to satisfy the constraint, specified as a fully qualified class name represented by a character vector or a `meta.class` instance.

Properties

`Class`

Class name that the actual value must derive from or be an instance of to satisfy the constraint. Set this property through the constructor via the `class` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test That Actual Value Is Instance of Specified Class

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsInstanceOf

testCase = TestCase.forInteractiveUse;
```

Verify that the actual value, 5, is an instance of the double class.

```
testCase.verifyThat(5, IsInstanceOf('double'))
```

```
Interactive verification passed.
```

Repeat the test using an instance of `meta.class` instead of a character vector.

```
testCase.verifyThat(5, IsInstanceOf(?double))
```

```
Interactive verification passed.
```

Assert that zero is an instance of the logical class.

```
testCase.assertThat(0, IsInstanceOf('logical'))
```

```
Interactive assertion failed.

----------------------
Framework Diagnostic:
----------------------
IsInstanceOf failed.
--> The value must be an instance of the expected type.

    Actual Class:
        double
    Expected Type:
        logical

Actual Value:
         0
Assertion failed.
```

Verify that `@sin` is a function handle.

```
testCase.verifyThat(@sin, IsInstanceOf(?function_handle))
```

```
Interactive verification passed.
```

Verify that `name` is an instance of the `char` class.

```
name = 42;
testCase.verifyThat(name, IsInstanceOf('char'))
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
IsInstanceOf failed.
--> The value must be an instance of the expected type.

    Actual Class:
        double
    Expected Type:
        char

Actual Value:
        42
```

Test That Derived Class Is Instance of Specified Class

In a file in your working folder, create a class, `DerivedExample`, that inherits from the `handle` class.

```
classdef DerivedExample < handle
end
```

At the command prompt, create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsInstanceOf

testCase = TestCase.forInteractiveUse;
```

Verify that an instance of the `DerivedExample` class is an instance of a `handle`.

```
exObj = DerivedExample;
testCase.verifyThat(exObj, IsInstanceOf(?handle))
```

```
Interactive verification passed.
```

Even though `exObj` is not an instance of the `handle` class, the verification passes because it derives from the `handle` class.

See Also

`IsOfClass` | `isa`

matlab.unittest.constraints.IsLessThan class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.Constraint`

Constraint specifying value less than another value

Construction

`IsLessThan(ceilVal)` creates a constraint specifying that an actual value is less than another value. The constraint is satisfied if the actual value array is less than the specified ceiling value, `ceilVal`. The actual value is less than `ceilVal` only if the result of the expression `actual < ceilVal` is nonempty and all values are true.

Input Arguments

`ceilVal`

Smallest value that fails the constraint.

Properties

`CeilingValue`

Smallest value that fails the constraint. Set this property through the constructor via the `ceilVal` input argument.

Methods

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test That Actual Value Is Less Than Provided Ceiling Value

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsLessThan

testCase = TestCase.forInteractiveUse;
```

Test that the actual value is less than four.

```
actVal = 3;
testCase.verifyThat(actVal, IsLessThan(4))
```

```
Interactive verification passed.
```

Test that the actual value is less than three.

```
testCase.verifyThat(actVal, IsLessThan(3))
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
IsLessThan failed.
--> The value must be less than the maximum value.

Actual double:
         3
Maximum Value (Exclusive):
         3
```

The actual value is equal to, not less than, three.

Test that each element in the actual value array is less than four.

```
actVal = [1 2 3];
testCase.verifyThat(actVal, IsLessThan(4))
```

```
Interactive verification passed.
```

Test that each element in the actual value matrix is less than four.

```
actVal = [1 2 3; 4 5 6];
testCase.verifyThat(actVal, IsLessThan(4))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsLessThan failed.
--> Each element must be less than the maximum value.

    Failing Indices:
             2     4     6

Actual double:
         1     2     3
         4     5     6
Maximum Value (Exclusive):
         4
```

The matrix contains three elements that are greater than or equal to four.

Test that the actual value, 0, is less than every element in an array.

```
testCase.verifyThat(0, IsLessThan([1 2 3]))
```

```
Interactive verification passed.
```

Test that elements in the actual value array are less than the corresponding ceiling values.

```
testCase.verifyThat([4 -9 0], IsLessThan([5 -3 2]))
```

```
Interactive verification passed.
```

Repeat the test, this time negating the second actual value element.

```
testCase.verifyThat([4 9 0], IsLessThan([5 -3 2]))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
```

```
IsLessThan failed.
--> Each element must be less than each corresponding element of the maximum value arra

    Failing Indices:
             2

Actual double:
         4     9     0
Maximum Value (Exclusive):
         5    -3     2
```

The negated element is greater than `-3`.

See Also

`IsGreaterThan` | `IsGreaterThanOrEqualTo` | `IsLessThanOrEqualTo` | `lt`

matlab.unittest.constraints.IsLessThanOrEqualTo class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.Constraint`

Constraint specifying value less than or equal to another value

Construction

`IsLessThanOrEqualTo(ceilVal)` creates a constraint specifying that an actual value is less than or equal to another value. The constraint is satisfied if the actual value array is less than or equal to the specified ceiling value, `ceilVal`. The actual value is less than or equal to `ceilVal` only if the result of the expression `actual <= ceilVal` is nonempty and all values are true.

Input Arguments

`ceilVal`

Maximum value to satisfy the constraint.

Properties

`CeilingValue`

Maximum value to satisfy the constraint. Set this property through the constructor via the `ceilVal` input argument.

Methods

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test That Actual Value Is Less Than or Equal to Provided Ceiling Value

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsLessThanOrEqualTo

testCase = TestCase.forInteractiveUse;
```

Test that the actual value is less than or equal to four.

```
actVal = 3;
testCase.verifyThat(actVal, IsLessThanOrEqualTo(4))
```

```
Interactive verification passed.
```

Test that the actual value is less than or equal to three.

```
testCase.verifyThat(actVal, IsLessThanOrEqualTo(3))
```

```
Interactive verification passed.
```

Test that each element in the actual value array is less than or equal to four.

```
actVal = [1 2 3 4];
testCase.verifyThat(actVal, IsLessThanOrEqualTo(4))
```

```
Interactive verification passed.
```

Test that each element in the actual value matrix is less than or equal to four.

```
actVal = [1 2 3; 4 5 6];
testCase.verifyThat(actVal, IsLessThanOrEqualTo(4))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsLessThanOrEqualTo failed.
--> Each element must be less than or equal to the maximum value.

    Failing Indices:
             4     6

Actual double:
         1     2     3
         4     5     6
Maximum Value (Inclusive):
         4
```

The matrix contains two elements that are greater than four.

Test that the actual value, 1, is less than or equal to every element in an array.

```
testCase.verifyThat(1, IsLessThanOrEqualTo([1 2 3]))
```

```
Interactive verification passed.
```

Test that elements in the actual value array are less than the corresponding ceiling values.

```
testCase.verifyThat([4 -9 2], IsLessThanOrEqualTo([5 -3 2]))
```

```
Interactive verification passed.
```

Repeat the test, this time negating the second actual value element.

```
testCase.verifyThat([4 9 2], IsLessThanOrEqualTo([5 -3 2]))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsLessThanOrEqualTo failed.
--> Each element must be less than or equal to each corresponding element of the maximu

    Failing Indices:
```

```
             2

Actual double:
         4     9     2
Maximum Value (Inclusive):
         5    -3     2
```

The negated element is greater than `-3`.

See Also

`IsGreaterThan` | `IsGreaterThanOrEqualTo` | `IsLessThan` | `le`

matlab.unittest.constraints.IsOfClass class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying class type

Construction

`IsOfClass(class)` provides a constraint specifying the class type. The constraint is satisfied if the actual value is the same class as `class` . The constraint is not satisfied if the actual value derives from `class`.

Input Arguments

`class`

Class name that must be matched by the actual value to satisfy the constraint, specified as a fully qualified class name represented by a character vector or a `meta.class` instance.

Properties

`Class`

Class name that must be matched by the actual value to satisfy the constraint. Set this property through the constructor via the `class` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test That Actual Value Class Is Specified Class

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsOfClass

testCase = TestCase.forInteractiveUse;
```

Verify that the actual value, 5, is a double.

```
testCase.verifyThat(5, IsOfClass('double'))
```

```
Interactive verification passed.
```

Repeat the test using an instance of `meta.class` instead of a character vector.

```
testCase.verifyThat(5, IsOfClass(?double))
```

```
Interactive verification passed.
```

Assert that zero is an instance of the logical class.

```
testCase.assertThat(0, IsOfClass('logical'))
```

```
Interactive assertion failed.

----------------------
Framework Diagnostic:
----------------------
IsOfClass failed.
--> The value's class is incorrect.

    Actual Class:
        double
    Expected Class:
        logical

Actual Value:
         0
Assertion failed.
```

Verify that `@sin` is a function handle.

```
testCase.verifyThat(@sin, IsOfClass(?function_handle))
```

```
Interactive verification passed.
```

Verify that `name` is an instance of the `char` class.

```
name = 42;
testCase.verifyThat(name, IsOfClass('char'))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsOfClass failed.
--> The value's class is incorrect.

    Actual Class:
        double
    Expected Class:
        char

Actual Value:
        42
```

Test That Derived Class Is Instance of Specified Class

In a file in your working folder, create a class, `DerivedExample`, that inherits from the `handle` class.

```
classdef DerivedExample < handle
end
```

At the command prompt, create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsOfClass

testCase = TestCase.forInteractiveUse
```

Verify that an instance of the `DerivedExample` class is an instance of a `handle`.

```
exObj = DerivedExample;
testCase.verifyThat(exObj, IsOfClass(?handle))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsOfClass failed.
--> The value's class is incorrect.

    Actual Class:
        DerivedExample
    Expected Class:
        handle

Actual Value:
      DerivedExample with no properties.
```

Even though `exObj` derives from the `handle` class, it is not an instance of the `handle` class.

Verify that an instance of the `DerivedExample` class is an instance of a `DerivedExample`.

```
testCase.verifyThat(exObj, IsOfClass(?DerivedExample))
```

```
Interactive verification passed.
```

See Also

`IsInstanceOf` | `class`

matlab.unittest.constraints.IsReal class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying real valued array

Construction

`IsReal` provides a constraint specifying a real valued array. This constraint is satisfied only if the actual value contains only real values.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test That Actual Value Array Is Real Valued

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsReal

testCase = TestCase.forInteractiveUse;
```

Verify that the values `5` and `5+0i` are real.

```
testCase.verifyThat(5, IsReal)
testCase.verifyThat(5+0i, IsReal)
```

```
Interactive verification passed.
Interactive verification passed.
```

Test if the imaginary number is real.

```
testCase.verifyThat(sqrt(-1), IsReal)
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsReal failed.
--> The value must be real.

Actual Value:
      0.000000000000000 + 1.000000000000000i
```

The actual value is imaginary.

Assert that an array contains only real values.

```
testCase.assertThat([0 1 1 2 3 5 8 13], IsReal)
```

```
Interactive assertion passed.
```

Test that the array, `arr`, is real.

```
arr = [NaN -Inf];
testCase.verifyThat(arr, IsReal)
```

```
Interactive verification passed.
```

Multiply the array by a complex number and test that the values are not real.

```
testCase.verifyThat(42i*arr, ~IsReal)
```

```
Interactive verification passed.
```

See Also

`isreal`

matlab.unittest.constraints.IsSameHandleAs class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying handle instance same as another

Construction

`IsSameHandle(h)` provides a constraint specifying a handle instance or group of instances is same as another.

The constraint is satisfied only if each element of the actual value is the same instance as each corresponding element of `h`.

Input Arguments

h

`handle` object or array of `handle` objects. The actual value array passed to the qualification must be the same size as `h`.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test Handles for Equality

In a file in your working folder, create the following handle class for interactive testing.

```
classdef ExampleHandle < handle
end
```

At the command prompt, create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsSameHandleAs

testCase = TestCase.forInteractiveUse;
```

Instantiate two handles.

```
h1 = ExampleHandle;
h2 = ExampleHandle;
```

Verify that the handle, `h1`, is the same as `h1`.

```
testCase.verifyThat(h1, IsSameHandleAs(h1))
```

```
Interactive verification passed.
```

Test that `h1` is the same handle instance as `h2`.

```
testCase.verifyThat(h1, IsSameHandleAs(h2))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsSameHandleAs failed.
--> Values do not refer to the same handle.

Actual Value:
      ExampleHandle with no properties.
Expected Handle Object:
      ExampleHandle with no properties.
```

Test that two arrays of handles are the same instances.

```
expArr = [h1 h2 h1];
actArr = [h1 h2 h1];

testCase.verifyThat(expArr, IsSameHandleAs(actArr))
```

```
Interactive verification passed.
```

The arrays satisfy the constraint even though the elements within a particular array are not the same instance as each other.

Verify that the constraint is not satisfied if it expects a single handle and the actual value is an array of the same instances.

```
testCase.verifyThat([h1 h1], IsSameHandleAs(h1))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsSameHandleAs failed.
--> Sizes do not match.
        Actual Value Size           : [1  2]
        Expected Handle Object Size : [1  1]

Actual Value:
      1x2 ExampleHandle array with no properties.
Expected Handle Object:
      ExampleHandle with no properties.
```

Similarly, the constraint is not satisfied a single handle instance if it expects an array of handles.

```
testCase.verifyThat(h2, IsSameHandleAs([h2 h2]))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsSameHandleAs failed.
--> Sizes do not match.
        Actual Value Size           : [1  1]
        Expected Handle Object Size : [1  2]

Actual Value:
      ExampleHandle with no properties.
```

```
Expected Handle Object:
      1x2 ExampleHandle array with no properties.
```

See Also

IsEqualTo | eq | handle

matlab.unittest.constraints.IsSameSetAs class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying set contains same elements as another set

Construction

`IsSameSetAs(expSet)` creates a constraint specifying that a set contains same elements as another set. The constraint produces a qualification failure for any actual-value set that is not the same set as the expected-value set.

Sets can have the same elements in different orders, different numbers, or different shapes. An actual value is considered the same set as the expected set if `ismember(actual,expected)` and `ismember(expected,actual)` both return arrays that contain all `true` values and at least one of the following conditions is met:

- The actual set and the expected set are of the same class.
- The actual set is an object.
- The expected set is an object.

Input Arguments

`expSet` — Expected-value set to compare to actual-value set
depends on test values

Expected-value set to compare to actual-value set. The type of the input depends on the test values.

Properties

`ExpectedSet` — Expected-value set to compare to actual-value set
depends on test values

This property is read-only.

Expected-value set to compare to actual-value set. The data type of the property depends on the test values. Set this property through the constructor via the `expSet` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test if Actual and Expected Sets Are the Same

Create a test case for interactive testing.

```
import matlab.unittest.TestCase;
import matlab.unittest.constraints.IsSameSetAs;

testCase = TestCase.forInteractiveUse;
```

Verify that two sets are the same.

```
actSet = {'a' 'b' 'c'};
expSet = {'a' 'b' 'c'};

testCase.verifyThat(actSet,IsSameSetAs(expSet))
```

```
Interactive verification passed.
```

Repeat the test with a different expected set. The test fails because the sets do not contain the same elements.

```
expSet = {'a' 'b' 'd'};

testCase.verifyThat(actSet,IsSameSetAs(expSet))
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
```

```
----------------------
IsSameSetAs failed.
--> The actual value contains 1 element(s) not found in the expected set:
    --> Element at index 3:
            {'c'}
--> The actual value is missing 1 element(s) found in the expected set:
    --> Element at index 3:
            {'d'}

Actual Value:
  1×3 cell array

    {'a'}    {'b'}    {'c'}
Expected Set:
  1×3 cell array

    {'a'}    {'b'}    {'d'}
```

Verify that two sets are the same. Although the order of the elements and the shape of the sets are different, the sets contain the same elements.

```
actSet = [1 2 3];
expSet = [3;2;1];

testCase.verifyThat(actSet,IsSameSetAs(expSet))
```

```
Interactive verification passed.
```

Verify that two sets are the same. Although the `expSet` contains elements that are not unique and has a size that does not match `actSet`, the two sets have the same elements.

```
expSet = [1 2 3 1 2];

testCase.verifyThat(actSet,IsSameSetAs(expSet))
```

```
Interactive verification passed.
```

See Also

`ismember` | `matlab.unittest.constraints.HasUniqueElements` | `matlab.unittest.constraints.IsSubsetOf` | `matlab.unittest.constraints.IsSupersetOf`

Introduced in R2018a

matlab.unittest.constraints.IsScalar class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying scalar value

Construction

`IsScalar` provides a constraint that specifies a scalar value. The constraint is satisfied if the actual value is a scalar.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test That Actual Value Is Scalar

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsScalar

testCase = TestCase.forInteractiveUse;
```

Verify that a value of zero satisfies the `IsScalar` constraint.

```
testCase.verifyThat(0,IsScalar)
```

```
Interactive verification passed.
```

Assert that a single object is scalar.

```
testCase.assertThat(timeseries(1),IsScalar)
```

```
Interactive verification passed.
```

Verify that a vector is not scalar.

```
testCase.verifyThat([2 3],IsScalar)
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
IsScalar failed.
--> The value must be a scalar.
--> The value has a size of [1  2].

Actual Value:
         2     3
```

Assert that an empty structure does not satisfy the `IsScalar` constraint.

```
testCase.assertThat(struct([]),IsScalar)
```

```
Interactive assertion failed.

----------------------
Framework Diagnostic:
----------------------
IsScalar failed.
--> The value must be a scalar.
--> The value has a size of [0  0].

Actual Value:
    0x0 struct array with no fields.
Assertion failed.
```

See Also

`HasElementCount` | `HasLength` | `HasSize` | `IsEmpty` | `isscalar`

Introduced in R2014b

matlab.unittest.constraints.IsSparse class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying sparse array

Construction

`IsSparse` creates a constraint specifying a sparse array. This constraint is satisfied only when the actual value is sparse.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test That Actual Value Array Is Sparse

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsSparse

testCase = TestCase.forInteractiveUse;
```

Create an identity matrix, and test if it is sparse.

```
F = eye(7);
testCase.verifyThat(F, IsSparse)
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
```

```
----------------------
IsSparse failed.
--> The value must be sparse.

Actual Value:
         1     0     0     0     0     0     0
         0     1     0     0     0     0     0
         0     0     1     0     0     0     0
         0     0     0     1     0     0     0
         0     0     0     0     1     0     0
         0     0     0     0     0     1     0
         0     0     0     0     0     0     1
```

The matrix, F, is a full matrix.

Convert F to a sparse matrix and retest for sparsity.

```
S = sparse(F);
testCase.verifyThat(S, IsSparse)
```

```
Interactive verification passed.
```

See Also

issparse

matlab.unittest.constraints.IsSubsetOf class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying actual set is subset of expected set

Construction

`IsSubsetOf(expSet)` creates a constraint specifying that the actual value set is a subset of the expected value set. The constraint produces a qualification failure for any actual value set that is not a subset of the expected value set. An actual value set is considered a subset of the expected value set if `ismember(actSet,expSet)` contains all true values and the actual and expected values satisfy one of the following conditions:

- The actual and expected values are of the same class.
- The actual value is an object.
- The expected value is an object.

Input Arguments

`expSet` — Expected value set to compare to actual value set
depends on test values

Expected value set to compare to actual value set. The type of the input depends on the test values.

Properties

`Superset` — Superset of actual value set
depends on test values

Superset of the actual value set. The data type of the property depends on the test values. To satisfy the constraint, the actual value set must be a subset of `Superset`. Set this property through the constructor via the `expSet` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test if Actual Set Is Subset of Expected

Create a test case for interactive testing.

```
import matlab.unittest.TestCase;
import matlab.unittest.constraints.IsSubsetOf;

testCase = TestCase.forInteractiveUse;
```

Verify that the actual cell array is a subset of the expected set.

```
testCase.verifyThat({'c','b'}, IsSubsetOf({'a','b','c'}));
```

```
Interactive verification passed.
```

```
testCase.verifyThat({'a';'d'}, IsSubsetOf({'a','b','c'}));
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
IsSubsetOf failed.
--> The actual value contains 1 element(s) not found in the expected superset:
    --> Element at index 2 not found in the expected superset:
                'd'

Actual Value (cell):
        'a'
        'd'
Expected Superset (cell):
        'a'    'b'    'c'
```

Assert that a set of doubles is a subset of the expected set.

```
testCase.assertThat([25;209], IsSubsetOf(magic(21)));
```

```
Interactive assertion passed.
```

```
testCase.assertThat(25:33, IsSubsetOf(30:40));
```

```
Interactive assertion failed.

---------------------
Framework Diagnostic:
---------------------
IsSubsetOf failed.
--> The actual value contains 5 element(s) not found in the expected superset:
    --> Element at index 1 not found in the expected superset:
                25
    --> Element at index 2 not found in the expected superset:
                26
    --> Element at index 3 not found in the expected superset:
                27
    --> Element at index 4 not found in the expected superset:
                28
    --> Element at index 5 not found in the expected superset:
                29

Actual Value (double):
        25    26    27    28    29    30    31    32    33
Expected Superset (double):
        30    31    32    33    34    35    36    37    38    39    40
Assertion failed.
```

Verify that the rows of a table are a subset of the expected table.

```
actT = table([3,1]',{'C';'A'},logical([0;1]));
expT = table([1:2:5]',{'A';'C';'E'},logical([1;0;0]));
testCase.verifyThat(actT, IsSubsetOf(expT));
```

```
Interactive verification passed.
```

Test that if the actual and expected sets have different types, the `IsSubsetOf` constraint is not satisfied.

```
testCase.assumeThat(single(1:3), IsSubsetOf(0:5));
```

```
Interactive assumption failed.

---------------------
Framework Diagnostic:
```

```
----------------------
IsSubsetOf failed.
--> Classes do not match.

    Actual Class:
        single
    Expected Class:
        double

Actual Value (single):
         1     2     3
Expected Superset (double):
         0     1     2     3     4     5
Assumption failed.
```

See Also

ismember | matlab.unittest.constraints | matlab.unittest.constraints.IsSameSetAs | matlab.unittest.constraints.IsSupersetOf

Introduced in R2016a

matlab.unittest.constraints.IsSubstringOf class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying substring of another string

Construction

`IsSubstringOf(superstring)` creates a constraint specifying a substring of another string scalar or character vector. The constraint is satisfied only if the actual value is contained within an expected superstring, `superstring`.

`IsSubstringOf(superstring,Name,Value)` provides a constraint with additional options specified by one or more `Name,Value` pair arguments. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

`superstring`

Text that contains the actual value, specified as a string scalar or character vector. `superstring` can include newline characters.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`IgnoringCase`

Indicator to ignore case, specified as `false` or `true` (`logical` `0` or `1`).

Default: `false`

IgnoringWhitespace

Indicator to ignore whitespace, specified as `false` or `true` (`logical` `0` or `1`). If `IgnoringWhitespace` is `true`, MATLAB removes any whitespace characters from both the actual value and the expected value before determining whether the constraint is satisfied.

Note When `IgnoringWhitespace` is `true`, the input argument `superstring` must contain at least one non-whitespace character.

Default: `false`

WithCount

Number of times `superstring` must contain the actual value, specified as a positive integer.

The testing framework uses the `count` function to count occurrences of the actual value. When you specify an empty substring, the `count` function counts empty strings at the beginning and end of `superstring` and between each pair of its characters. In other words, if `superstring` has `N` characters, it also has `N+1` empty substrings.

Properties

IgnoreCase

Indicator if the constraint is insensitive to case, specified in the name-value pair argument, `'IgnoringCase'`.

IgnoreWhitespace

Indicator if the constraint is insensitive to whitespace, specified in the name-value pair argument, `'IgnoringWhitespace'`.

Superstring

Superstring that includes the actual value, specified in the input argument, `superstring`.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test That Actual Value Is Substring of Specified String

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsSubstringOf

testCase = TestCase.forInteractiveUse;
```

Define the actual value string.

```
S = string('This Is One Long String');
```

Test that the actual value `'One'` is contained in `S`.

```
testCase.verifyThat('One', IsSubstringOf(S))
```

```
Verification passed.
```

Test that the actual value `'long'` is contained in `S`.

```
testCase.verifyThat('long', IsSubstringOf(S))
```

```
Verification failed.
    ----------------------
    Framework Diagnostic:
    ----------------------
    IsSubstringOf failed.
    --> The value is not found within the superstring.

    Actual char:
        long
    Expected Superstring:
        "This Is One Long String"
```

By default, the `IsSubstringOf` constraint is case sensitive.

Repeat the test ignoring case.

```
testCase.verifyThat('long', IsSubstringOf(S, ...
    'IgnoringCase', true))
```

```
Verification passed.
```

Test that the actual value `'is'` is contained in `S` twice. For the test to pass, configure the constraint to ignore case.

```
testCase.verifyThat('is', IsSubstringOf(S, ...
    'WithCount', 2, 'IgnoringCase', true))
```

```
Verification passed.
```

Test that the actual value `'thisisone'` is contained in `S`. For the test to pass, configure the constraint to ignore whitespace and case.

```
testCase.verifyThat('thisisone', IsSubstringOf(S, ...
    'IgnoringCase', true, 'IgnoringWhitespace', true))
```

```
Verification passed.
```

Assert that the actual value `'longer'` is not contained in S.

```
testCase.assertThat('Longer', ~IsSubstringOf(S))
```

```
Assertion passed.
```

See Also

`ContainsSubstring` | `EndsWithSubstring` | `Matches` | `StartsWithSubstring`

matlab.unittest.constraints.IssuesNoWarnings class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.Constraint`

Constraint specifying function that issues no warnings

Construction

`outConstObj = IssuesNoWarnings` creates a constraint, `outConstObj`, specifying a function that issues no warnings when the testing framework invokes it. The constraint is satisfied if no warnings are issued when the testing framework invokes the function.

`outConstObj = IssuesNoWarnings('WhenNargoutIs', numOutputs)` creates a constraint that can determine if the actual value is a function handle that issues no warnings when the testing framework invokes it with a particular number of output arguments, `numOutputs`.

Input Arguments

`numOutputs`

Number of outputs the constraint requests when invoking the function handle, specified as a non-negative, real, scalar integer.

Default: `0`

Properties

`FunctionOutputs`

Output arguments produced at invocation of the supplied function handle, specified as a cell array. This property provides access to output arguments. It is read only and the testing framework sets it when it invokes the function handle. The number of outputs is determined by the `Nargout` property.

Nargout

Number of output arguments the instance uses when executing functions. Set this property through the constructor via the `numOutputs` input argument.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Instantiate IssuesNoWarnings Constraint

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IssuesNoWarnings

testCase = TestCase.forInteractiveUse;
```

Verify that a call to `true` does not result in any warning.

```
testCase.verifyThat(@true, IssuesNoWarnings)
```

```
Interactive verification passed.
```

Verify that a call to `size` with an empty array does not result in any warning. Examine the output arguments.

```
issuesNoWarningsConstraint = IssuesNoWarnings('WhenNargoutIs', 2);
testCase.verifyThat(@() size([]), issuesNoWarningsConstraint)
[actualOut1, actualOut2] = issuesNoWarningsConstraint.FunctionOutputs{:};
```

```
Interactive verification passed.
```

Verify that the constraint is not satisfied if the actual value is not a function handle.

```
testCase.verifyThat(5, IssuesNoWarnings)
```

```
Interactive verification failed.
```

```
----------------------
Framework Diagnostic:
----------------------
IssuesNoWarnings failed.
--> The value must be an instance of the expected type.

    Actual Class:
        double
    Expected Type:
        function_handle

Actual Value:
         5
```

Verify that the constraint is not satisfied if the actual value results in a warning.

```
testCase.verifyThat(@() warning('some:id', 'Message'), IssuesNoWarnings)
```

```
Warning: Message
> In @()warning('some:id','Message')
  In matlab.unittest.internal.constraints.FunctionHandleConstraint/invoke (line 36)
  In matlab.unittest.internal.constraints.WarningQualificationConstraint/invoke (line 3
  In matlab.unittest.constraints.IssuesNoWarnings/issuesNoWarnings (line 140)
  In matlab.unittest.constraints.IssuesNoWarnings/satisfiedBy (line 90)
  In matlab.unittest.internal.qualifications.QualificationDelegate/qualifyThat (line 62
  In matlab.unittest.qualifications.Verifiable/verifyThat (line 228)
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
IssuesNoWarnings failed.
--> The function issued warnings.

    Warnings Issued:
        --> 'some:id'
                Message
```

```
Evaluated Function:
        @()warning('some:id','Message')
```

See Also

matlab.unittest.constraints.Constraint | matlab.unittest.constraints.IssuesWarnings | matlab.unittest.constraints.Throws | warning

matlab.unittest.constraints.IssuesWarnings class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.Constraint`

Constraint specifying function that issues expected warning profile

Description

The `IssuesWarnings` class creates a constraint that issues an expected warning profile. The constraint is satisfied only if the actual value is a function handle that issues a specific set of warnings. You specify warnings using warning identifiers.

By default, the constraint only confirms that when the testing framework invokes the function handle, MATLAB issues the specified set of warnings. It ignores the number of times the warnings are issued, in what order they are issued, and whether or not any unspecified warnings are issued. However, you can set parameters to respect the order, the count, and the warning set. Alternatively, you can specify the exact warning profile for comparison.

Construction

`outConstObj = IssuesWarnings(warnArr)` creates a constraint, `outConstObj`, specifying a function that issues expected warnings, `warnArr`.

`outConstObj = IssuesWarnings(expVal,Name,Value)` creates a constraint with additional options specified by one or more `Name,Value` pair arguments. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

warnArr

Warning identifiers expected when the testing framework invokes the function handle, specified as a cell array of warning identifiers. If `warnArr` is empty, the constructor throws an `MException`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Exactly

Indicator if the value is a function handle that must issue a warning profile that is an exact match, specified as `false` or `true` (`logical` `0` or `1`). When this value is `false`, the instance relies on specification of other parameters and default instance behavior to determine the strictness of its comparison. When set to `true`, the instance requires the warning profile to be exactly the same as the specified warning profile.

Default: `false`

RespectingCount

Indicator whether to respect element counts, specified as `false` or `true` (`logical` `0` or `1`). When this value is `false`, the instance is insensitive to the number of occurrences of members and ignores their frequency. When set to `true`, the instance is sensitive to the total number of set members. This means that, in addition to ensuring that all of the specified warnings are issued, this instance is not satisfied if the number of times that a particular warning issues differs from the number of times that warning is specified in `warnArr`.

Default: `false`

RespectingOrder

Indicator whether to respect the order of elements, specified as `false` or `true` (`logical` `0` or `1`). When this value is `false`, the instance is insensitive to the order of the set members. When set to `true`, the instance is sensitive to the order of the set members.

This means that this instance is not satisfied if the order of the issued warnings differs from the order the warnings are specified in `warnArr`.

The order of a given set of warnings is determined by trimming the warning profiles to a profile with no repeated adjacent warnings. For example, the warning profile `{id:A, id:A, id:B, id:C, id:C, id:C, id:A, id:A, id:A}` is trimmed to `{id:A, id:B, id:C, id:A}`.

When this constraint respects order, the order of the warnings that are issued and expected must match the order of the expected warning profile. Warnings issued that are not listed in `warnArr` are ignored when determining order.

Default: `false`

`RespectingSet`

Indicator whether to respect set elements, specified as `false` or `true` (`logical` `0` or `1`). When this value is `false`, the instance ignores additional set members. When set to `true`, the instance is sensitive to additional set members. This means that, in addition to ensuring that all of the specified warnings are issued, this instance is not satisfied if any extra, unspecified warnings are issued.

Default: `false`

`WhenNargoutIs`

Number of outputs the constraint should request when invoking the function handle, specified as a non-negative, real, scalar integer.

Default: `0`

Properties

`Exact`

Indicator of whether the constraint performs exact comparisons. Set this property through the constructor via the name-value pair argument, `'Exactly'`.

`ExpectedWarnings`

Expected warning identifiers. Set this read-only property through the constructor via the `warnArr` input argument.

`FunctionOutputs`

Output arguments produced at invocation of the supplied function handle, specified as a cell array. This property provides access to output arguments. It is read only and the testing framework sets it when it invokes the function handle. The number of outputs is determined by the `Nargout` property.

`Nargout`

Number of output arguments the instance uses when it executes functions. Set this property through the constructor via the name-value pair argument, `'WhenNargoutIs'`.

`RespectCount`

Indicator if the constraint respects the element counts, specified through the constructor via the name-value pair argument, `'RespectingCount'`.

`RespectOrder`

Indicator if the constraint respects the order of elements, specified through the constructor via the name-value pair argument, `'RespectingOrder'`.

`RespectSet`

Indicator if the constraint respects set elements, specified through the constructor via the name-value pair argument, `'RespectingSet'`.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Instantiate IssuesWarnings Constraint

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IssuesWarnings

testCase = TestCase.forInteractiveUse;
```

Create a helper anonymous function for use in this example. Create several warning identifiers.

```
issueWarnings = @(idCell) cellfun(@(id) warning(id,'Message'), idCell);
firstID =   'first:id';
secondID =  'second:id';
thirdID =   'third:id';
```

Verify that the helper function issues a particular warning.

```
testCase.verifyThat(@() issueWarnings({firstID}),...
    IssuesWarnings({firstID}))
```

```
Interactive verification passed.
```

Verify the function issues a warning ignoring count, warning set, and order.

```
testCase.verifyThat(@() issueWarnings({firstID, thirdID, secondID,...
    firstID}), IssuesWarnings({secondID, firstID}))
```

```
Interactive verification passed.
```

Verify the function issues a warning while respecting the warning set.

```
testCase.verifyThat(@() issueWarnings({firstID, thirdID, secondID,...
    firstID}), IssuesWarnings({firstID, secondID, thirdID}, ...
    'RespectingSet', true))
```

```
Interactive verification passed.
```

Verify the function issues a warning while respecting the warning count.

```
testCase.verifyThat(@() issueWarnings({secondID, firstID, thirdID,...
    secondID}), IssuesWarnings({firstID, secondID, secondID}, ...
    'RespectingCount', true))
```

```
Interactive verification passed.
```

Verify the function issues a warning while respecting the warning order.

```
testCase.verifyThat(@() issueWarnings({firstID, secondID, secondID,...
    thirdID}), IssuesWarnings({firstID, secondID}, 'RespectingOrder', true))
```

```
Interactive verification passed.
```

Verify the function issues an exact match to the expected warning profile.

```
testCase.verifyThat(@() issueWarnings({firstID, secondID, secondID,...
    thirdID}), IssuesWarnings({firstID, secondID, secondID, thirdID}, ...
    'Exactly', true))
```

```
Interactive verification passed.
```

Verify that the constraint is not satisfied if the actual value is not a function handle.

```
testCase.verifyThat(5, IssuesWarnings({firstID}))
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
IssuesWarnings failed.
--> The value must be an instance of the expected type.

    Actual Class:
        double
    Expected Type:
        function_handle

Actual Value:
         5
```

Verify that the constraint is not satisfied if the function does not issue a warning.

```
testCase.verifyThat(@rand, IssuesWarnings({firstID}))
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
IssuesWarnings failed.
--> The function handle did not issue a correct warning profile.
```

```
    The expected warning profile ignores:
      Set
      Count
      Order
    --> The function handle did not issue any warnings.

    Expected Warning Profile:
        --> 'first:id'

Evaluated Function:
        @rand
```

Verify that the constraint is not satisfied if the function issues a non-specified warning identifier.

```
testCase.verifyThat(@() issueWarnings({firstID}), IssuesWarnings({secondID}))
```

```
Warning: Message
> In @(id)warning(id,'Message')
  In @(idCell)cellfun(@(id)warning(id,'Message'),idCell)
  In @()issueWarnings({firstID})
  In matlab.unittest.internal.constraints.FunctionHandleConstraint/invoke (line 36)
  In matlab.unittest.internal.constraints.WarningQualificationConstraint/invoke (line 3
  In matlab.unittest.constraints.IssuesWarnings/invoke (line 431)
  In matlab.unittest.constraints.IssuesWarnings/invokeCapturingOutput (line 510)
  In matlab.unittest.constraints.IssuesWarnings/issuesExpectedWarnings (line 519)
  In matlab.unittest.constraints.IssuesWarnings/satisfiedBy (line 239)
  In matlab.unittest.internal.qualifications.QualificationDelegate/qualifyThat (line 62
  In matlab.unittest.qualifications.Verifiable/verifyThat (line 228)
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
IssuesWarnings failed.
--> The function handle did not issue a correct warning profile.
    The expected warning profile ignores:
      Set
      Count
      Order
    --> The function handle did not issue the correct warnings.

        Missing Warnings:
            --> 'second:id'
```

```
    Actual Warning Profile:
        --> 'first:id'
    Expected Warning Profile:
        --> 'second:id'

Evaluated Function:
        @()issueWarnings({firstID})
```

Consider the following actual value and warning array.

```
actVal = @() issueWarnings({firstID, firstID, secondID, firstID});
warnArr = {firstID, secondID, firstID, firstID};
```

Test whether the warning array is exactly the same as the expected array.

```
testCase.verifyThat(actVal, IssuesWarnings(warnArr, 'Exactly', true))
```

```
Warning: Message
> In @(id)warning(id,'Message')
  In @(idCell)cellfun(@(id)warning(id,'Message'),idCell)
  In @()issueWarnings({firstID,firstID,secondID,firstID})
  In matlab.unittest.internal.constraints.FunctionHandleConstraint/invoke (line 36)
  In matlab.unittest.internal.constraints.WarningQualificationConstraint/invoke (line 3
  In matlab.unittest.constraints.IssuesWarnings/invoke (line 431)
  In matlab.unittest.constraints.IssuesWarnings/invokeCapturingOutput (line 510)
  In matlab.unittest.constraints.IssuesWarnings/issuesExpectedWarnings (line 519)
  In matlab.unittest.constraints.IssuesWarnings/satisfiedBy (line 239)
  In matlab.unittest.internal.qualifications.QualificationDelegate/qualifyThat (line 62
  In matlab.unittest.qualifications.Verifiable/verifyThat (line 228)
Warning: Message
> In @(id)warning(id,'Message')
  In @(idCell)cellfun(@(id)warning(id,'Message'),idCell)
  In @()issueWarnings({firstID,firstID,secondID,firstID})
  In matlab.unittest.internal.constraints.FunctionHandleConstraint/invoke (line 36)
  In matlab.unittest.internal.constraints.WarningQualificationConstraint/invoke (line 3
  In matlab.unittest.constraints.IssuesWarnings/invoke (line 431)
  In matlab.unittest.constraints.IssuesWarnings/invokeCapturingOutput (line 510)
  In matlab.unittest.constraints.IssuesWarnings/issuesExpectedWarnings (line 519)
  In matlab.unittest.constraints.IssuesWarnings/satisfiedBy (line 239)
  In matlab.unittest.internal.qualifications.QualificationDelegate/qualifyThat (line 62
  In matlab.unittest.qualifications.Verifiable/verifyThat (line 228)
Warning: Message
> In @(id)warning(id,'Message')
  In @(idCell)cellfun(@(id)warning(id,'Message'),idCell)
  In @()issueWarnings({firstID,firstID,secondID,firstID})
```

```
  In matlab.unittest.internal.constraints.FunctionHandleConstraint/invoke (line 36)
  In matlab.unittest.internal.constraints.WarningQualificationConstraint/invoke (line 3
  In matlab.unittest.constraints.IssuesWarnings/invoke (line 431)
  In matlab.unittest.constraints.IssuesWarnings/invokeCapturingOutput (line 510)
  In matlab.unittest.constraints.IssuesWarnings/issuesExpectedWarnings (line 519)
  In matlab.unittest.constraints.IssuesWarnings/satisfiedBy (line 239)
  In matlab.unittest.internal.qualifications.QualificationDelegate/qualifyThat (line 62
  In matlab.unittest.qualifications.Verifiable/verifyThat (line 228)
Warning: Message
> In @(id)warning(id,'Message')
  In @(idCell)cellfun(@(id)warning(id,'Message'),idCell)
  In @()issueWarnings({firstID,firstID,secondID,firstID})
  In matlab.unittest.internal.constraints.FunctionHandleConstraint/invoke (line 36)
  In matlab.unittest.internal.constraints.WarningQualificationConstraint/invoke (line 3
  In matlab.unittest.constraints.IssuesWarnings/invoke (line 431)
  In matlab.unittest.constraints.IssuesWarnings/invokeCapturingOutput (line 510)
  In matlab.unittest.constraints.IssuesWarnings/issuesExpectedWarnings (line 519)
  In matlab.unittest.constraints.IssuesWarnings/satisfiedBy (line 239)
  In matlab.unittest.internal.qualifications.QualificationDelegate/qualifyThat (line 62
  In matlab.unittest.qualifications.Verifiable/verifyThat (line 228)
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IssuesWarnings failed.
--> The function handle did not issue a correct warning profile.
    The expected warning profile must match exactly.
    --> The function handle did not issue the exact warning profile expected.

    Actual Warning Profile:
        --> 'first:id'
        --> 'first:id'
        --> 'second:id'
        --> 'first:id'
    Expected Warning Profile:
        --> 'first:id'
        --> 'second:id'
        --> 'first:id'
        --> 'first:id'

Evaluated Function:
        @()issueWarnings({firstID,firstID,secondID,firstID})
```

Test whether the warning array is the same as the expected array when respecting set, order and count.

```
testCase.verifyThat(actVal, IssuesWarnings(warnArr,...
    'RespectingSet',true,'RespectingOrder',true,'RespectingCount',true))
```

```
Interactive verification passed.
```

In this example, a constraint that specifies a warning profile that respects set, order and count is not the same as one that specifies an exact warning profile.

See Also

`matlab.unittest.constraints.IssuesNoWarnings` | `matlab.unittest.constraints.Throws` | `warning`

Topics

"Message Identifiers"

matlab.unittest.constraints.IsSupersetOf class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying actual set is superset of expected set

Construction

`IsSupersetOf(expSet)` creates a constraint specifying that the actual value set is a superset of the expected value set. The constraint produces a qualification failure for any actual value set that is not a superset of the expected value set. An actual value set is considered a superset of the expected value set if `ismember(expSet,actSet)` contains all true values and the actual and expected values satisfy one of the following conditions:

- The actual and expected values are of the same class.
- The actual value is an object.
- The expected value is an object.

Input Arguments

expSet — Expected value set to compare to actual value set
depends on test values

Expected value set to compare to actual value set. The type of the input depends on the test values.

Properties

Subset — Subset of actual value set
depends on test values

Subset of the actual value set. The data type of the property depends on the test values. To satisfy the constraint, the actual value set must be a superset of `Subset`. Set this property through the constructor via the `expSet` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test if Actual Set Is Superset of Expected

Create a test case for interactive testing.

```
import matlab.unittest.TestCase;
import matlab.unittest.constraints.IsSupersetOf;

testCase = TestCase.forInteractiveUse;
```

Verify that the actual cell array is a subset of the expected set.

```
testCase.verifyThat({'a','b','c'}, IsSupersetOf({'c';'b'}));
```

```
Interactive verification passed.
```

```
testCase.verifyThat({'a','b','c'}, IsSupersetOf({'a','d'}));
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
IsSupersetOf failed.
--> The expected subset contains 1 element(s) not found in the actual value:
    --> Element at index 2 not found in the actual value:
                'd'

Actual Value (cell):
        'a'    'b'    'c'
Expected Subset (cell):
        'a'    'd'
```

Assert that a set of doubles is a subset of the expected set.

```
testCase.assertThat(magic(21), IsSupersetOf([25;209]));
```

```
Interactive assertion passed.
```

```
testCase.assertThat(25:33, IsSupersetOf(30:40));
```

```
Interactive assertion failed.

---------------------
Framework Diagnostic:
---------------------
IsSupersetOf failed.
--> The expected subset contains elements not found in the actual value (First 5 of 7)
    --> Element at index 5 not found in the actual value:
                34
    --> Element at index 6 not found in the actual value:
                35
    --> Element at index 7 not found in the actual value:
                36
    --> Element at index 8 not found in the actual value:
                37
    --> Element at index 9 not found in the actual value:
                38

Actual Value (double):
        25    26    27    28    29    30    31    32    33
Expected Subset (double):
        30    31    32    33    34    35    36    37    38    39    40
Assertion failed.
```

Verify that the rows of a table are a subset of the expected table.

```
actT = table([1:2:5]',{'A';'C';'E'},logical([1;0;0]));
expT = table([3,1]',{'C';'A'},logical([0;1]));
testCase.verifyThat(actT, IsSupersetOf(expT));
```

```
Interactive verification passed.
```

Test that if the actual and expected sets have different types, the `IsSubsetOf` constraint is not satisfied.

```
testCase.assumeThat(single(0:5), IsSupersetOf(1:3));
```

```
Interactive assumption failed.
```

```
----------------------
Framework Diagnostic:
----------------------
IsSupersetOf failed.
--> Classes do not match.

    Actual Class:
        single
    Expected Class:
        double

Actual Value (single):
         0     1     2     3     4     5
Expected Subset (double):
         1     2     3
Assumption failed.
```

See Also

`ismember` | `matlab.unittest.constraints` | `matlab.unittest.constraints.IsSameSetAs` | `matlab.unittest.constraints.IsSubsetOf`

Introduced in R2016a

matlab.unittest.constraints.IsTrue class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.Constraint`

Constraint specifying true value

Construction

`IsTrue` provides a constraint specifying a true value. This constraint is satisfied only by a scalar logical with a value of `true`.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test Actual Value Is True

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsTrue
```

```
testCase = TestCase.forInteractiveUse;
```

Test that `true` satisfies the `IsTrue` constraint.

```
testCase.verifyThat(true, IsTrue)
```

```
Interactive verification passed.
```

Test that the `IsTrue` constraint is not satisfied by `false`.

```
testCase.verifyThat(false, IsTrue)
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
IsTrue failed.
--> The value must evaluate to "true".

Actual Value:
         0
```

The test fails because `false` returns `logical(0)`.

Test that the `IsTrue` constraint is not satisfied by the double `1`.

```
testCase.verifyThat(1, IsTrue)
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
IsTrue failed.
--> The value must be logical. It is of type "double".

Actual Value:
         1
```

The `IsTrue` constraint is satisfied only by `logical(1)`.

Test that the `IsTrue` constraint is not satisfied by a logical array of ones.

```
testCase.verifyThat([true true true], IsTrue)
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
IsTrue failed.
--> The value must be scalar. It has a size of [1  3].

Actual Value:
         1     1     1
```

The `IsTrue` constraint is satisfied only if the value is scalar and `logical(1)`.

Tips

- For faster test execution, use `verifyTrue`, `assertTrue`, `assumeTrue`, or `fatalAssertTrue` instead of `IsTrue`.
- To display custom comparisons in the form of a function handle, use `ReturnsTrue` instead of `IsTrue`.

See Also

`IsFalse` | `ReturnsTrue`

matlab.unittest.constraints.LogicalComparator class

Package: `matlab.unittest.constraints`

Comparator for two logical values

Construction

`LogicalComparator` creates a comparator for two logical values. The comparator is satisfied if the actual and expected values have the same sparsity and the logical values are equivalent.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Compare Logical Values

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.LogicalComparator
import matlab.unittest.constraints.IsEqualTo

testCase = TestCase.forInteractiveUse;
```

Test the value of `true`.

```
testCase.assertThat(true, IsEqualTo(true, ...
    'Using', LogicalComparator))
```

```
Interactive assertion passed.
```

Test an array of `true` values.

```
testCase.assertThat([true true true], IsEqualTo(true, ...
    'Using', LogicalComparator))
```

```
Interactive assertion failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> LogicalComparator failed.
    --> The logical values are not equal

Actual Logical Value:
         1     1     1
Expected Logical Value:
         1
Assertion failed.
```

The actual value must be a scalar logical to satisfy the constraint.

Compare the value of `1` to `true`.

```
testCase.verifyThat(1, IsEqualTo(true, 'Using', LogicalComparator))
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> LogicalComparator failed.
    --> Class check failed.
        --> Classes do not match.

            Actual Class:
                double
            Expected Class:
                logical

Actual double:
         1
Expected logical:
         1
```

Compare the value of `false` to `true`.

```
testCase.assertThat(false, IsEqualTo(true, 'Using', LogicalComparator))
```

```
Interactive assertion failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> LogicalComparator failed.
    --> The logical values are not equal

Actual Logical Value:
         0
Expected Logical Value:
         1
Assertion failed.
```

See Also

`matlab.unittest.constraints.IsEqualTo`

Introduced in R2013a

matlab.unittest.constraints.Matches class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying string matches regular expression

Construction

`Matches(expr)` creates a constraint that specifies that a string scalar or character vector matches a regular expression. The constraint is satisfied only if the actual value matches the given regular expression, `expr`.

`Matches(expr,Name,Value)` provides a constraint with additional options specified by one or more `Name,Value` pair arguments. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

`expr`

Regular expression that the actual value must match to satisfy the constraint, specified as a string scalar or character vector. `expr` can include newline characters.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`IgnoringCase`

Indicator to ignore case, specified as `false` or `true` (`logical` `0` or `1`).

Default: `false`

WithCount

Number of times the actual value must match `expr`, specified as a positive integer.

Properties

Expression

Regular expression that the actual value must match, specified in the input argument, `expr`.

IgnoreCase

Indicator if the constraint is insensitive to case, specified in the name-value pair argument, `'IgnoringCase'`.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test That Actual Value Matches Regular Expression

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.Matches

testCase = TestCase.forInteractiveUse;
```

Test that the actual value `'Epsilon Eridani'` matches `'eps'`.

```
testCase.verifyThat('Epsilon Eridani', Matches('^eps'));
```

```
Verification failed.
    ----------------------
    Framework Diagnostic:
```

```
    ---------------------
    Matches failed.
    --> The value does not match the regular expression.

    Actual char:
        Epsilon Eridani
    Regular Expression:
        ^eps
```

To satisfy the constraint, configure it to be case insensitive.

```
testCase.verifyThat('Epsilon Eridani', Matches('^eps', ...
    'IgnoringCase', true));
```

```
Verification passed.
```

Define the regular expression that the actual value must match.

```
expr = 'Some[Tt]?ext';
```

The `[Tt]?` contained in the regular expression indicates that either `'T'` or `'t'` matches at that location 0 or 1 times.

Test that the actual values, `'SomeText'` and `'Sometext'`, satisfy the constraint.

```
testCase.verifyThat('SomeText', Matches(expr));
testCase.verifyThat('Sometext', Matches(expr));
```

```
Verification passed.
Verification passed.
```

Test that the actual value `'SomeText Sometext Someext'` matches the regular expression three times.

```
testCase.verifyThat('SomeText Sometext Someext', Matches(expr, ...
    'WithCount', 3));
```

```
Verification passed.
```

Test that the actual value `'sometext'` does not satisfy the constraint.

```
testCase.verifyThat('sometext', ~Matches(expr));
```

```
Verification passed.
```

See Also

`ContainsSubstring` | `EndsWithSubstring` | `IsSubstringOf` | `StartsWithSubstring` | `regexp`

Topics

"Regular Expressions"

matlab.unittest.constraints.NumericComparator class

Package: `matlab.unittest.constraints`

Comparator for numeric data types

Construction

`NumericComparator` creates a comparator for numeric data types. The comparator is satisfied if inputs are of the same class with equivalent size, complexity, and sparsity, and the built-in `isequaln` function returns `true`.

`NumericComparator('Within',tolObj)` creates a comparator using a specified tolerance. In this case, `NumericComparator` first checks for equivalent class, size, and sparsity of the actual and expected values. If these checks fail, the comparator is not satisfied. If these checks pass and the `isequaln` or complexity check fails, `NumericComparator` delegates comparison to the supplied tolerance, `tolObj`.

Input Arguments

`tolObj`

`matlab.unittest.constraints.Tolerance` instance

Properties

`Tolerance`

Specific tolerance used in construction of the comparator, specified as a `Tolerance` object in the `tolObj` input argument

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Compare Numeric Values

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.NumericComparator
import matlab.unittest.constraints.IsEqualTo

testCase = TestCase.forInteractiveUse;
```

Use a numeric comparator to test that `1.618` is equal to `1.618`.

```
testCase.verifyThat(1.618, IsEqualTo(1.618,...
    'Using', NumericComparator))
```

```
Verification passed.
```

Verify that `(1+sqrt(5))/2` is equal to `1.618`.

```
testCase.verifyThat((1+sqrt(5))/2, IsEqualTo(1.618, ...
    'Using', NumericComparator))
```

```
Verification failed.
    ---------------------
    Framework Diagnostic:
    ---------------------
    IsEqualTo failed.
    --> NumericComparator failed.
        --> The numeric values are not equal using "isequaln".
        --> Failure table:
                    Actual         Expected           Error                RelativeError
                _______________    ________    ____________________    ____________________

                1.61803398874989     1.618     3.39887498947977e-05    2.10066439399244e-05

        Actual Value:
           1.618033988749895
        Expected Value:
           1.618000000000000
```

Retest using a relative tolerance of `0.25%`.

```
import matlab.unittest.constraints.RelativeTolerance

testCase.verifyThat((1+sqrt(5))/2, IsEqualTo(1.618, ...
    'Using', NumericComparator('Within', RelativeTolerance(0.0025))))
```

```
Verification passed.
```

See Also

isequaln | matlab.unittest.constraints.IsEqualTo | matlab.unittest.constraints.Tolerance

Introduced in R2013a

matlab.unittest.constraints.ObjectComparator class

Package: `matlab.unittest.constraints`

Comparator for MATLAB or Java objects

Construction

`ObjectComparator` creates a comparator for MATLAB or Java objects. The comparator is satisfied if `isequaln` returns `true`. However, if the class of the expected value defines an `isequal` method, whether visible or hidden, but not an `isequaln` method, the `ObjectComparator` uses that method for comparison instead of `isequaln`.

`ObjectComparator('Within',tolObj)` creates a comparator using a specified tolerance. `ObjectComparator` first checks that a call to `isequaln` or `isequal` returns `true`. If the check fails, the `ObjectComparator` checks for equivalent class, size, and sparsity of the actual and expected values. If these checks pass, `ObjectComparator` delegates comparison to the supplied tolerance, `tolObj`. The value of this tolerance must be of the same class as the actual and expected values.

Input Arguments

`tolObj`

`Tolerance` instance

Properties

`Tolerance`

Specific tolerance used in construction of the comparator, specified as a `matlab.unittest.constraints.Tolerance` object in the `tolObj` input argument

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Compare MATLAB Objects

In a file, `MyInt.m`, in your working folder, create a subclass of `int8`.

```
classdef MyInt < int8
    methods
        function i = MyInt(value)
            i@int8(value);
        end
    end
end
```

At the command prompt, create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.ObjectComparator
import matlab.unittest.constraints.IsEqualTo

testCase = TestCase.forInteractiveUse;
```

Use an `ObjectComparator` to test that two instances of `MyInt` are equal to each other.

```
testCase.verifyThat(MyInt(10), ...
    IsEqualTo(MyInt(10), 'Using', ObjectComparator))
```

```
Interactive verification passed.
```

Test the equality of two instances of `MyInt` that are constructed with different input values.

```
testCase.verifyThat(MyInt(11), ...
    IsEqualTo(MyInt(10), 'Using', ObjectComparator))
```

```
Interactive verification failed.

---------------------
```

```
Framework Diagnostic:
----------------------
IsEqualTo failed.
--> ObjectComparator failed.
    --> The objects are not equal using "isequal".

Actual Object:
      MyInt:

      int8 data:
       11
Expected Object:
      MyInt:

      int8 data:
       10
```

One instance of `MyInt` has a value of `11`, and the other has a value of `10`.

Repeat the test and specify that values must be equal within an absolute tolerance of `1`.

```
import matlab.unittest.constraints.AbsoluteTolerance

testCase.verifyThat(MyInt(11), IsEqualTo(MyInt(10), ...
    'Using', ObjectComparator('Within', AbsoluteTolerance(MyInt(1)))))
```

```
Interactive verification passed.
```

See Also

`isequal` | `matlab.unittest.constraints.IsEqualTo` | `matlab.unittest.constraints.Tolerance`

Introduced in R2013a

matlab.unittest.constraints.PublicPropertyComparator class

Package: `matlab.unittest.constraints`

Comparator for public properties of MATLAB objects

Description

The `PublicPropertyComparator` compares public properties of MATLAB objects.

The `PublicPropertyComparator` supports MATLAB objects or arrays of objects and recursively compares data structures contained in the public properties. The `PublicPropertyComparator` is different from the `isequal` function because it examines only the public properties of the objects.

Typically you construct a `PublicPropertyComparator` using the `supportingAllValues` static method.

Construction

`PublicPropertyComparator` creates a comparator for public properties of MATLAB objects. This comparator supports only objects with no public properties.

`PublicPropertyComparator(compObj)` indicates a comparator, `compObj`, that defines the comparator used to compare public properties. This comparator supports recursion only in the data types supported by `compObj`.

`PublicPropertyComparator(compObj,Name,Value)` provides a comparator with additional options specified by one or more `Name,Value` pair arguments.

Methods

supportingAllValues	Comparator for public properties that supports any value in recursion

Input Arguments

`compObj`

Comparator object

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`IgnoringProperties` — Properties to ignore during object comparison
cell array of character vectors | string array

Properties to ignore during object comparison, specified as a cell array of character vectors or a string array.

Example: `PublicPropertyComparator('IgnoringProperties',{'Stack'})`

`Recursively` — Setting for whether comparator operates recursively
`false` (default) | `true`

Setting for whether comparator operates recursively, specified as `false` or `true` (`logical` `0` or `1`). When this value is `false`, the comparator does not operate recursively on its data.

When the value is `true`, the data types that the public property comparator supports are fully supported in recursion.

Properties

IgnoredProperties

Properties to ignore during object comparison, specified in the name-value pair argument, `'IgnoringProperties'`.

Recursive

Indicator of whether comparator operates recursively, specified in the name-value pair argument, `'Recursively'`.

Limitations

- The `PublicPropertyComparator` does not compare public properties of objects that overload the `subsref`, `numel`, or `properties` functions.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Instantiate PublicPropertyComparator Object

In a file in your working folder, construct this `Employee` class.

```
classdef Employee
    properties (SetAccess=immutable)
        Name
    end
    properties (Access=private)
        Location
    end
    methods
        function obj = Employee(name,location)
            obj.Name = name;
```

```
            obj.Location = location;
        end
    end
end
```

At the command prompt, create two instances of the `Employee` class.

```
e1 = Employee('sam','Building A');
e2 = Employee('Sam','Building B');
```

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.PublicPropertyComparator
import matlab.unittest.constraints.StringComparator

testCase = TestCase.forInteractiveUse;
```

Construct a comparator and verify that `e1` and `e2` are equal.

```
compObj = PublicPropertyComparator;
testCase.verifyThat(e1, IsEqualTo(e2,'Using',compObj))
```

```
Error using matlab.unittest.constraints.Comparator/throwUnsupportedValue (line 313)
None of the currently available comparators support the value.

Available Comparators:
    1×0 Comparator array with no properties.

Value (char):
        Sam

Error in matlab.unittest.constraints.Comparator>getActExpCompFrom (line 402)
    throwUnsupportedValue(comparison.Comparators,expVal);

Error in matlab.unittest.constraints.Comparator>deepComparisonIsSatisfied (line 351)
[actVal,expVal,comp] = getActExpCompFrom(comparison);

Error in matlab.unittest.constraints.Comparator>deepComparisonIsSatisfied (line 355)
        if ~deepComparisonIsSatisfied(subComparisonArray(k))

Error in matlab.unittest.constraints.Comparator/satisfiedBy (line 84)
            bool = deepComparisonIsSatisfied(comparison);

Error in matlab.unittest.constraints.IsEqualTo/satisfiedBy (line 193)
            bool = constraint.Comparator.satisfiedBy(actual,constraint.Expected);

Error in matlab.unittest.internal.qualifications.QualificationDelegate/qualifyThat (line 80)
                result = constraint.satisfiedBy(actual);

Error in matlab.unittest.qualifications.Verifiable/verifyThat (line 230)
            qualifyThat(verifiable.VerificationDelegate, ...
```

The test fails because, by default, the `PublicPropertyComparator` does not support character vectors.

Construct a comparator that supports character vectors. Specify that the comparison is not case-sensitive.

```
compObj = PublicPropertyComparator(StringComparator);
testCase.verifyThat(e1, IsEqualTo(e2,'Using',compObj, 'IgnoringCase',true))
```

```
Interactive verification passed.
```

The test passes even though `e1.Location` and `e2.Location` are not the same. Since `Location` is a private property, the comparator does not compare its contents.

See Also

`matlab.unittest.constraints.IsEqualTo` | `matlab.unittest.constraints.ObjectComparator`

Introduced in R2014a

matlab.unittest.constraints.PublicPropertyComparator.supportingAllValues

Class: `matlab.unittest.constraints.PublicPropertyComparator`
Package: `matlab.unittest.constraints`

Comparator for public properties that supports any value in recursion

Syntax

```
PublicPropertyComparator.supportingAllValues
PublicPropertyComparator.supportingAllValues(Name,Value)
```

Description

`PublicPropertyComparator.supportingAllValues` creates a comparator for public properties of MATLAB objects. This comparator supports any value in recursion. `supportingAllValues` is a `Static` method of the `PublicPropertyComparator` class.

`PublicPropertyComparator.supportingAllValues(Name,Value)` creates a comparator for public properties of MATLAB objects with additional options specified by one or more `Name,Value` pair arguments.

You typically pass this comparator to another constraint, such as `IsEqualTo`. You can use the `Name,Value` pairs of the `IsEqualTo` constraint with a comparator constructed with the `PublicPropertyComparator.supportingAllValues` syntax.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`IgnoringCase` — Setting for comparator insensitivity to case
`false` (default) | `true`

Setting for whether the comparator is insensitive to case, specified as `false` or `true` (`logical` `0` or `1`). When it is `false`, the comparator is sensitive to case. The comparator uses this name-value pair when comparing character vectors and string arrays.

`IgnoringFields` — Fields to ignore during struct comparison
cell array of character vectors

Fields to ignore during `struct` comparison, specified as a cell array of character vectors.

Example:
```
PublicPropertyComparator.supportingAllValues('IgnoringFields',
{'timestamp'})
```

`IgnoringProperties` — Properties to ignore during object comparison
cell array of character vectors

Properties to ignore during object comparison, specified as a cell array of character vectors.

Example:
```
PublicPropertyComparator.supportingAllValues('IgnoringProperties',
{'Stack'})
```

`IgnoringWhitespace` — Setting for comparator insensitivity to whitespace characters
`false` (default) | `true`

Setting for whether the comparator is insensitive to whitespace characters, specified as `false` or `true` (`logical` `0` or `1`). When it is `false`, the comparator is sensitive to whitespace characters. Whitespace characters consist of space, form feed, new line, carriage return, horizontal tab, and vertical tab. The comparator uses this name-value pair only if the contents being compared consist of character vectors.

`Within` — Tolerance to use for numerical comparison
instance of `matlab.unittest.constraints.Tolerance`

Tolerance to use for numerical comparison, specified as a `matlab.unittest.constraints.Tolerance` object. The comparator uses this name-value pair only if the contents being compared consist of numeric types.

Limitations

- The `PublicPropertyComparator` does not compare public properties of objects that overload the `subsref`, `numel`, or `properties` functions.

Examples

Instantiate `PublicPropertyComparator` Object to Support All Values

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.PublicPropertyComparator

testCase = TestCase.forInteractiveUse;
```

Test a passing case.

```
m1 = MException('Msg:ID','MsgText');
m2 = MException('Msg:ID','MsgText');
testCase.verifyThat(m1, IsEqualTo(m2, 'Using', ...
    PublicPropertyComparator.supportingAllValues))
```

```
Interactive verification passed.
```

Test a failing case.

```
m1 = MException('Msg:ID','MsgText');
m2 = MException('Msg:ID','msgtext');
testCase.verifyThat(m1, IsEqualTo(m2, 'Using', ...
    PublicPropertyComparator.supportingAllValues))
```

```
Interactive verification failed.

----------------------
```

```
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> Path to failure: <Value>.message
    --> StringComparator failed.
        --> The character arrays are not equal.

        Actual char:
            MsgText
        Expected char:
            msgtext

Actual MException:
      MException with properties:

        identifier: 'Msg:ID'
           message: 'MsgText'
             cause: {}
             stack: [0×1 struct]
Expected MException:
      MException with properties:

        identifier: 'Msg:ID'
           message: 'msgtext'
             cause: {}
             stack: [0×1 struct]
```

Test a case that passes when the comparator ignores differences in case.

```
m1 = MException('Msg:ID','MsgText');
m2 = MException('Msg:ID','msgtext');
testCase.verifyThat(m1, IsEqualTo(m2,'Using', ...
    PublicPropertyComparator.supportingAllValues('IgnoringCase',true)))
```

```
Interactive verification passed.
```

Test a case that passes when the comparator ignores the `message` property of the `MException` object.

```
testCase.verifyThat(m1, IsEqualTo(m2,'Using', ...
    PublicPropertyComparator.supportingAllValues('IgnoringProperties',{'message'})))
```

```
Interactive verification passed.
```

Instantiate `PublicPropertyComparator` Object with Tolerance

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.RelativeTolerance
import matlab.unittest.constraints.PublicPropertyComparator

testCase = TestCase.forInteractiveUse;
```

Define actual and expected `timeseries` objects. Perturb one of the actual data points by 1%.

```
expected = timeseries(1:10);
actual = expected;
actual.Data(7) = 1.01*actual.Data(7);
```

Test that the actual and expected values are equal within a relative tolerance of 2%.

```
testCase.verifyThat(actual, IsEqualTo(expected,...
    'Within', RelativeTolerance(.02)))
```

```
Verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> ObjectComparator failed.
    --> The objects are not equal using "isequal".
    --> The tolerance was ignored. The tolerance as specified does not support comparisons of timeseries values.

    Actual timeseries:
          timeseries

          Common Properties:
                    Name: 'unnamed'
                    Time: [10x1 double]
                TimeInfo: [1x1 tsdata.timemetadata]
                    Data: [1x1x10 double]
                DataInfo: [1x1 tsdata.datametadata]

          More properties, Methods
    Expected timeseries:
          timeseries

          Common Properties:
                    Name: 'unnamed'
                    Time: [10x1 double]
                TimeInfo: [1x1 tsdata.timemetadata]
                    Data: [1x1x10 double]
                DataInfo: [1x1 tsdata.datametadata]
```

```
        More properties, Methods
```

Use the `PublicPropertyComparator` in the construction of the constraint.

```
testCase.verifyThat(actual, IsEqualTo(expected,...
    'Within', RelativeTolerance(.02),...
    'Using', PublicPropertyComparator.supportingAllValues))
```

```
Interactive verification passed.
```

The test passes because the `PublicPropertyComparator` compares each public property individually instead of comparing the object all at once. In the former test, the `ObjectComparator` compares `timeseries` objects, and therefore relies on the `isequal` method of the `timeseries` class. Due to the perturbation in the actual `timeseries`, `isequal` returns `false`. The comparator does not apply the tolerance because the double-valued tolerance cannot apply directly to the `timeseries` object. In the latter test, the comparator applies the tolerance to each public property that contains double-valued data.

See Also

`matlab.unittest.constraints.IsEqualTo` | `matlab.unittest.constraints.ObjectComparator`

Introduced in R2014a

matlab.unittest.constraints.RelativeTolerance class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.Tolerance`

Relative numeric tolerance

Description

This numeric `Tolerance` assesses the magnitude of the difference between actual and expected values, relative to the expected value. For the tolerance to be satisfied, `abs(expVal - actVal) <= relTol.*abs(expVal)` must be true.

Construction

`RelativeTolerance(tolVals)` creates a relative tolerance object that assesses the magnitude of the difference between actual and expected values, relative to the expected value.

The data types of the inputs to the `RelativeTolerance` constructor determine which data types the tolerance supports. For example, `RelativeTolerance(10*eps)` constructs a `RelativeTolerance` for comparing double-precision numeric arrays while `RelativeTolerance(single(2))` constructs a `RelativeTolerance` for comparing single-precision numeric arrays. If the actual and expected values being compared contain more than one numeric data type, the tolerance only applies to the data types specified by the values passed into the constructor.

To specify different tolerance values for different data types, you can pass multiple tolerance values to the constructor. For example, `RelativeTolerance(10*eps, 10*eps('single'))` constructs an `RelativeTolerance` that applies the following absolute tolerances:

- `10*eps` applies a relative tolerance of `10*eps` for double-precision numeric arrays.
- `10*eps('single')` applies a relative tolerance of `10*eps` for single-precision numeric arrays.

You can specify more than one tolerance for a particular data type by combining tolerances with the & and | operators. To combine two tolerances, the sizes of the tolerance values for each data type must be compatible.

Input Arguments

`tolVals`

Numeric tolerances, specified as a comma-separated list of arrays containing floating-point numbers. Each input argument contains the tolerance specification for a particular data type. Each numeric array can be a scalar or array the same size as the actual and expected values.

Properties

`Values`

Numeric tolerances, specified by the `tolVals` input argument

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test with Relative Tolerance

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.RelativeTolerance

testCase = TestCase.forInteractiveUse;
```

Assert that the difference between an actual value, `4.1`, and an expected value, `4.5`, is less than `10%`.

```
testCase.assertThat(4.1, IsEqualTo(4.5, ...
    'Within', RelativeTolerance(0.1)))
```

```
Assertion passed.
```

Specify Relative Tolerance for Different Data Types

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.RelativeTolerance

testCase = TestCase.forInteractiveUse;
```

Create the following actual and expected cell arrays.

```
act = {'abc', 123, single(106)};
exp = {'abc', 122, single(105)};
```

Test that the arrays satisfy the `RelativeTolerance` constraint within 2%.

```
testCase.verifyThat(act, IsEqualTo(exp, ...
    'Within', RelativeTolerance(0.02)))
```

```
Verification failed.

----------------------
Framework Diagnostic:
----------------------
IsEqualTo failed.
--> Path to failure: <Value>{3}
    --> NumericComparator failed.
        --> The numeric values are not equal using "isequaln".
        --> The tolerance was ignored. The tolerance as specified does not support comp
        --> Failure table:
                    Actual    Expected    Error    RelativeError
                    ______    ________    _____    _____________

                    106       105         1        0.00952381

        Actual single:
               106
```

```
        Expected single:
               105

Actual cell:
        'abc'    [123]    [106]
Expected cell:
        'abc'    [122]    [105]
```

The test fails because the tolerance is only applied to the `double` data type.

Create a tolerance object that specifies different tolerances for different data types.

```
tolObj = RelativeTolerance(0.02, single(0.02));
```

A tolerance of `2%` is a applied to `double` and `single` valued data.

Verify that the expected and actual values satisfy the `RelativeTolerance` constraint.

```
testCase.verifyThat(act, IsEqualTo(exp, 'Within', tolObj))
```

```
Verification passed.
```

Combine Relative and Absolute Tolerances

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.AbsoluteTolerance
import matlab.unittest.constraints.RelativeTolerance

testCase = TestCase.forInteractiveUse;
```

Define an actual value approximation for `pi`.

```
act = 3.14;
```

Construct a tolerance object to test that the difference between the actual and expected values is within `0.001` and within `0.25%`.

```
tolObj = AbsoluteTolerance(0.001) & RelativeTolerance(0.0025);
```

Verify that the actual value is within the tolerance of the expected value of `pi`.

```
testCase.verifyThat(act, IsEqualTo(pi, 'Within', tolObj))
```

```
Verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> NumericComparator failed.
    --> The numeric values are not equal using "isequaln".
    --> AndTolerance failed.
        --> AbsoluteTolerance failed.
            --> The error was not within absolute tolerance.
        --> RelativeTolerance passed.
            --> The error was within relative tolerance.
        --> Failure table:
                    Actual        Expected               Error                RelativeError        AbsoluteTolerance    Re
                    ______    _______________    ___________________    ____________________    _________________    __

                    3.14      3.14159265358979    -0.00159265358979299    -0.000506957382897213    0.001                0.

    Actual double:
           3.140000000000000
    Expected double:
           3.141592653589793
```

The actual value does not satisfy the `AbsoluteTolerance` constraint.

Construct a constraint that is satisfied if the values are within `0.001` or `0.25%`, and then retest the actual value.

```
tolObj = AbsoluteTolerance(0.001) | RelativeTolerance(0.0025);
testCase.verifyThat(act, IsEqualTo(pi, 'Within', tolObj))
```

```
Verification passed.
```

Combine Absolute and Relative Tolerances to Test Small and Large Values

Combine tolerances so when you test the equality of values, an absolute (floor) tolerance dominates when the values are near zero, and a relative tolerance dominates for larger values.

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.AbsoluteTolerance
import matlab.unittest.constraints.RelativeTolerance

testCase = TestCase.forInteractiveUse;
```

Define two structures containing electromagnetic properties of a vacuum. One structure, `approxVacuumProps`, contains approximate values for the permeability and speed of light in a vacuum.

```
approxVacuumProps.Permeability = 1.2566e-06; % Approximate
approxVacuumProps.Permitivity = 8.854187817*10^-12;
approxVacuumProps.LightSpeed = 2.9979e+08; % Approximate

baselineVacuumProps.Permeability = 4*pi*10^-7;
baselineVacuumProps.Permitivity = 8.854187817*10^-12;
baselineVacuumProps.LightSpeed = 1/sqrt(...
    baselineVacuumProps.Permeability*baselineVacuumProps.Permitivity);
```

Test that the relative difference between the approximate and baseline values is within `eps*1e11`.

```
testCase.verifyThat(approxVacuumProps, IsEqualTo(baselineVacuumProps, ...
    'Within', RelativeTolerance(eps*1e11)))
```

```
Verification failed.

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> Path to failure: <Value>.Permeability
    --> NumericComparator failed.
        --> The numeric values are not equal using "isequaln".
        --> RelativeTolerance failed.
            --> The error was not within relative tolerance.
            --> Failure table:
                          Actual            Expected                  Error
                        __________    ___________________    ____________________

                        1.2566e-06    1.25663706143592e-06    -3.70614359173257e-11

        Actual double:
                 1.256600000000000e-06
        Expected double:
                 1.256637061435917e-06

Actual struct:
        Permeability: 1.256600000000000e-06
         Permitivity: 8.854187816999999e-12
          LightSpeed: 299790000
```

```
Expected struct:
        Permeability: 1.256637061435917e-06
         Permitivity: 8.854187816999999e-12
          LightSpeed: 2.997924580105029e+08
```

The test fails because the relative difference in the permeabilities is not within the tolerance. The difference between the two values is small, but the numbers are close to zero, so the difference relative to their size is not small enough to satisfy the tolerance.

Construct a tolerance object to test that the absolute difference between the approximate and baseline values is within `1e-4`.

```
testCase.verifyThat(approxVacuumProps, IsEqualTo(baselineVacuumProps, ...
    'Within', AbsoluteTolerance(1e-4)))
```

```
Verification failed.

----------------------
Framework Diagnostic:
----------------------
IsEqualTo failed.
--> Path to failure: <Value>.LightSpeed
    --> NumericComparator failed.
        --> The numeric values are not equal using "isequaln".
        --> AbsoluteTolerance failed.
            --> The error was not within absolute tolerance.
            --> Failure table:
                         Actual          Expected              Error             Relati
                        _________    _________________    _________________    ________

                        299790000    299792458.010503    -2458.01050287485    -8.19904(

        Actual double:
               299790000
        Expected double:
                 2.997924580105029e+08

Actual struct:
        Permeability: 1.256600000000000e-06
         Permitivity: 8.854187816999999e-12
          LightSpeed: 299790000
Expected struct:
        Permeability: 1.256637061435917e-06
         Permitivity: 8.854187816999999e-12
          LightSpeed: 2.997924580105029e+08
```

The test fails because the absolute difference in the speed of light is not within the tolerance. The difference between the two values is small relative to their size, but too large to satisfy the tolerance.

Construct a logical disjunction of tolerance objects to test that the absolute difference between the approximate and baseline values is within `1e-4` or the relative difference is within `eps*1e11`. The test uses this tolerance so permeability values that are close to zero satisfy the absolute (floor) tolerance, and speed of light values that are large, satisfy the relative tolerance.

```
testCase.verifyThat(approxVacuumProps, IsEqualTo(baselineVacuumProps, ...
    'Within', RelativeTolerance(eps*1e11)| AbsoluteTolerance(1e-4)))
```

```
Verification passed.
```

See Also

`matlab.unittest.constraints.AbsoluteTolerance` | `matlab.unittest.constraints.IsEqualTo`

Introduced in R2013a

matlab.unittest.constraints.ReturnsTrue class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.Constraint`

Constraint specifying function handle that returns true

Construction

`ReturnsTrue` provides a constraint specifying that a function handle that returns true. The constraint is satisfied only by a function handle that returns a scalar logical with a value of `true`.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Test Actual Value Specified by Function Handle Returns True

These comparisons are shown for example only. There are other constraints that might better handle the particular comparisons.

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.ReturnsTrue

testCase = TestCase.forInteractiveUse;
```

Verify that the `ReturnsTrue` constraint is satisfied by the value returned by a handle to `true`.

```
testCase.verifyThat(@true, ReturnsTrue)
```

```
Interactive verification passed.
```

Verify that the `ReturnsTrue` constraint is not satisfied by a handle to `false`.

```
testCase.verifyThat(@false, ReturnsTrue)
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
ReturnsTrue failed.
--> The function handle should have evaluated to "true".
--> Returned value:
             0

Actual Function Handle:
        @false
```

Verify that a call to `isequal` returns true.

```
testCase.verifyThat(@() isequal(1,1), ReturnsTrue)
```

```
Interactive verification passed.
```

Verify that a function that returns a double-valued 1 does not satisfy the `ReturnsTrue` constraint.

```
testCase.verifyThat(@() double(true), ReturnsTrue)
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
ReturnsTrue failed.
--> The function handle should have returned a logical value. It was of type "double".
--> Returned value:
             1

Actual Function Handle:
        @()double(true)
```

Verify that the negation of a text comparison of `'a'` and `'b'` returns true.

```
testCase.verifyThat(@() ~strcmp('a','b'), ReturnsTrue)
```

```
Interactive verification passed.
```

Test if a comparison of `'a'` to the cell array `{'a','a'}` returns true.

```
testCase.verifyThat(@() strcmp('a',{'a','a'}), ReturnsTrue)
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
ReturnsTrue failed.
--> The function handle should have returned a scalar. The return value had a size of
--> Returned value:
             1     1

Actual Function Handle:
        @()strcmp('a',{'a','a'})
```

The constraint is not satisfied because the call to `strcmp` results a logical array, not a logical scalar.

Tips

- To display custom comparisons in the form of a function handle, use `ReturnsTrue` instead of `IsTrue`.

See Also

`Constraint` | `IsTrue`

matlab.unittest.constraints.StartsWithSubstring class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.BooleanConstraint`

Constraint specifying string starting with substring

Construction

`StartsWithSubstring(prefix)` creates a constraint specifying a string scalar or character vector starting with a substring. The constraint is satisfied only if the actual value starts with an expected prefix, `prefix`.

`StartsWithSubstring(prefix,Name,Value)` provides a constraint with additional options specified by one or more `Name,Value` pair arguments. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

`prefix`

Text at the start of the actual value, specified as a string scalar or character vector. `prefix` can include newline characters.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`IgnoringCase`

Indicator to ignore case, specified as `false` or `true` (`logical` `0` or `1`).

Default: `false`

`IgnoringWhitespace`

Indicator to ignore whitespace, specified as `false` or `true` (`logical 0` or `1`). If `IgnoringWhitespace` is `true`, MATLAB removes any whitespace characters from both the actual value and the expected value before determining whether the constraint is satisfied.

Note When `IgnoringWhitespace` is `true`, the input argument `prefix` must contain at least one non-whitespace character.

Default: `false`

Properties

`IgnoreCase`

Indicator if the constraint is insensitive to case, specified in the name-value pair argument, `'IgnoringCase'`.

`IgnoreWhitespace`

Indicator if the constraint is insensitive to whitespace, specified in the name-value pair argument, `'IgnoringWhitespace'`.

`Prefix`

Text at the start of the actual value, specified in the input argument, `prefix`.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test That Actual Value Starts with Specified Substring

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.StartsWithSubstring

testCase = TestCase.forInteractiveUse;
```

Define the actual value.

```
actVal = 'This Is One Long Message';
```

Test that `actVal` starts with `'This'`.

```
testCase.verifyThat(actVal, StartsWithSubstring('This'))
```

```
Verification passed.
```

Test that `actVal` starts with `'this is'`.

```
testCase.verifyThat(actVal, StartsWithSubstring('this is'))
```

```
Verification failed.

----------------------
Framework Diagnostic:
----------------------
StartsWithSubstring failed.
--> The value does not start with the supplied prefix.

Actual char:
    This Is One Long Message
Expected Prefix:
    this is
```

By default, the `StartsWithSubstring` constraint is case sensitive.

Repeat the test, this time ignoring case.

```
testCase.verifyThat(actVal, StartsWithSubstring('this is', ...
    'IgnoringCase', true))
```

```
Verification passed.
```

Test that `actVal` starts with `'thisisone'`. For the test to pass, configure the constraint to ignore whitespace and case.

```
testCase.verifyThat(actVal, StartsWithSubstring('thisisone', ...
    'IgnoringCase', true, 'IgnoringWhitespace', true))
```

```
Verification passed.
```

Assert that `actVal` does not start with `'long'`.

```
testCase.assertThat(actVal, ~StartsWithSubstring('Long'))
```

```
Assertion passed.
```

See Also

`ContainsSubstring` | `EndsWithSubstring` | `IsSubstringOf` | `Matches`

matlab.unittest.constraints.StringComparator class

Package: `matlab.unittest.constraints`

Comparator for two strings, character arrays, or cell arrays of character arrays

Construction

`StringComparator` creates a comparator for two strings, character arrays, or cell arrays of character arrays. The comparator is satisfied if the two values are equal. By default, `StringComparator` checks that the values have equal size and class, and then performs a case-sensitive comparison of each value.

`StringComparator(Name,Value)` creates a comparator with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`IgnoringCase`

Indicator if the comparator is insensitive to case, specified as `false` or `true` (`logical` `0` or `1`). When it is `false`, the comparator is sensitive to case.

Default: `false`

`IgnoringWhitespace`

Indicator if the comparator is insensitive to whitespace characters, specified as `false` or `true` (`logical` `0` or `1`). When it is `false`, the comparator is sensitive to whitespace

characters. Whitespace characters consist of space, form feed, new line, carriage return, horizontal tab, and vertical tab.

Default: `false`

Properties

`IgnoreCase`

Indicator if the comparator is insensitive to case, specified in the name-value pair argument, `'IgnoringCase'`

`IgnoreWhitespace`

Indicator if the comparator is insensitive to whitespace characters, specified in the name-value pair argument, `'IgnoringWhitespace'`

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Compare Character Vectors

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.StringComparator
import matlab.unittest.constraints.IsEqualTo

testCase = TestCase.forInteractiveUse;
```

Verify that the actual and expected character vectors are equal using a string comparator.

```
expected = 'coffee';
actual = 'coffee';
```

```
testCase.verifyThat(actual,IsEqualTo(expected, ...
    'Using', StringComparator))
```

```
Interactive verification passed.
```

Change the actual character vector and repeat the comparison.

```
expected = 'coF Fee';
testCase.verifyThat(actual,IsEqualTo(expected, ...
    'Using', StringComparator))
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
IsEqualTo failed.
--> StringComparator failed.
    --> The character arrays are not equal.

    Actual char:
        coffee
    Expected char:
        coF Fee
```

For the test to pass, construct a comparator that ignores case and whitespace characters.

```
testCase.verifyThat(actual,IsEqualTo(expected, ...
    'Using', StringComparator('IgnoringCase', true, ...
    'IgnoringWhitespace', true)))
```

```
Interactive verification passed.
```

See Also

`matlab.unittest.constraints.IsEqualTo` | `strcmp`

Introduced in R2013a

matlab.unittest.constraints.StructComparator class

Package: `matlab.unittest.constraints`

Comparator for MATLAB structure arrays

Construction

`StructComparator` creates a comparator for MATLAB structure arrays.

`StructComparator(compObj)` indicates a comparator, `compObj`, that defines the comparator used to compare values contained in the structure. By default, a `StructComparator` supports only empty structure arrays.

`StructComparator(compObj,Name,Value)` provides a comparator with additional options specified by one or more `Name,Value` pair arguments.

`StructComparator(Name,Value)` provides a comparator for empty structure arrays with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`compObj`

Comparator object

A comparator is passed into the `StructComparator` to support data types during recursion. By default, the `StructComparator` supports only empty structure arrays.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`IgnoringFields` — Fields to ignore during struct comparison
cell array of character vectors | string array

Fields to ignore during struct comparison, specified as a cell array of character vectors or a string array.

`Recursively` — Setting for whether comparator operates recursively
`false` (default) | `true`

Setting for whether the comparator operates recursively, specified as `false` or `true` (`logical` `0` or `1`). When this value is `false`, the comparator does not operate recursively on its data.

When the value is `true`, the data types the `StructComparator` supports are fully supported in recursion. For example:

```
comp1 = StructComparator(NumericComparator);
comp2 = StructComparator(NumericComparator, 'Recursively', true);
```

Both `comp1` and `comp2` support structures that contain numeric values as their fields. However, only `comp2` supports structures that recursively contain either structures or numeric values as their fields.

Properties

`IgnoredFields`

Fields to ignore during struct comparison, specified in the name-value pair argument, `'IgnoringFields'`.

`Recursive`

Indicator of whether comparator operates recursively, specified in the name-value pair argument, `'Recursively'`.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Compare Numeric Structures

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.AbsoluteTolerance
import matlab.unittest.constraints.StructComparator
import matlab.unittest.constraints.NumericComparator

testCase = TestCase.forInteractiveUse;
```

Create two equal structures.

```
s1 = struct('id',7,'score',7.3);
s2 = s1;
```

Test that the structures are equal. By default, the `StructComparator` supports only empty structures, so you need to configure the comparator with a `NumericComparator`.

```
testCase.verifyThat(s1, IsEqualTo(s2, 'Using', ...
    StructComparator(NumericComparator)))
```

```
Verification passed.
```

Change the score of `s2` and compare the structures again.

```
s2.score = 7.6;
testCase.verifyThat(s1, IsEqualTo(s2, 'Using', ...
    StructComparator(NumericComparator)))
```

```
Verification failed.
    ----------------------
    Framework Diagnostic:
    ----------------------
    IsEqualTo failed.
    --> Path to failure: <Value>.score
        --> NumericComparator failed.
            --> The numeric values are not equal using "isequaln".
            --> Failure table:
                    Actual    Expected    Error       RelativeError
```

```
                    ______    _______    _____    ____________________
                     7.3       7.6       -0.3     -0.0394736842105263

            Actual Value:
               7.300000000000000
            Expected Value:
               7.600000000000000

    Actual Value:
      struct with fields:

           id: 7
        score: 7.300000000000000
    Expected Value:
      struct with fields:

           id: 7
        score: 7.600000000000000
```

Specify an absolute tolerance for the comparison.

```
testCase.verifyThat(s1, IsEqualTo(s2, 'Using', ...
    StructComparator(NumericComparator), 'Within', ...
    AbsoluteTolerance(0.5)))
```

```
Verification passed.
```

Compare Character Structures

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.constraints.StructComparator
import matlab.unittest.constraints.StringComparator

testCase = TestCase.forInteractiveUse;
```

Create two structures. Make one of the fields a nested structure.

```
e1 = struct('name', struct('first','sam','last','smith'), ...
    'location','Building A');
e2 = e1;
```

Verify that the two structures are equal. Since the struct contains a nested structure, configure the constraint to operate recursively.

```
testCase.verifyThat(e1, IsEqualTo(e2, 'Using', ...
    StructComparator(StringComparator, 'Recursively', true)))
```

```
Verification passed.
```

Change the first name field of the `e2` structure and repeat the comparison.

```
e2.name.first = ' SAM';
testCase.verifyThat(e1, IsEqualTo(e2, 'Using', ...
    StructComparator(StringComparator, 'Recursively', true)))
```

```
Verification failed.
    ----------------------
    Framework Diagnostic:
    ----------------------
    IsEqualTo failed.
    --> Path to failure: <Value>.name.first
        --> StringComparator failed.
            --> The character arrays are not equal.

            Actual char:
                sam
            Expected char:
                 SAM

    Actual Value:
      struct with fields:

            name: [1×1 struct]
        location: 'Building A'
    Expected Value:
      struct with fields:

            name: [1×1 struct]
        location: 'Building A'
```

Configure the comparator to ignore case and whitespace characters.

```
testCase.verifyThat(e1, IsEqualTo(e2, 'Using', ...
    StructComparator(StringComparator, 'Recursively', true), ...
    'IgnoringCase', true, 'IgnoringWhitespace', true))
```

```
Verification passed.
```

See Also

matlab.unittest.constraints.IsEqualTo | matlab.unittest.constraints.Tolerance

Introduced in R2013a

matlab.unittest.constraints.TableComparator class

Package: `matlab.unittest.constraints`

Comparator for MATLAB tables

Construction

`TableComparator` creates a comparator for MATLAB tables that iterates over each column of the table. By default, a `TableComparator` supports only empty tables.

`TableComparator(compObj)` indicates a comparator, `compObj`, that defines the comparator used to compare values contained in the table columns.

`TableComparator( ___ ,Name,Value)` provides a comparator with additional options specified by one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

Input Arguments

`compObj` — Comparator object
instance of `matlab.unittest.constraints`

Comparator object, specified as an instance of `matlab.unittest.constraints` that is classified as a comparator.

Example: `CellComparator`

Example: `NumericComparator`

Example: `StringComparator`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`Recursively` — Setting for whether comparator operates recursively
`false` (default) | `true`

Setting for whether the comparator operates recursively, specified as `false` or `true` (`logical` `0` or `1`). When this value is `false`, the comparator does not operate recursively on its data.

When the value is `true`, the data types that the table comparator supports are fully supported in recursion.

Data Types: `logical`

Properties

`Recursive` — Indicator of whether comparator operates recursively
`false` (default) | `true`

Indicator of whether the comparator operates recursively, returned as `false` or `true` (`logical` `0` or `1`).

Data Types: `logical`

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test Table Equality with `TableComparator`

Create a test case for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Create two equal tables.

```
LastName = {'Williams';'Jones';'Brown'};
Age = [38;40;49];
```

```
Height = [64;67;64];
Weight = [131;133;119];
BloodPressure = [125 83; 117 75; 122 80];

T1 = table(Age,Height,Weight,BloodPressure, ...
    'RowNames',LastName);
T2 = T1;
```

Test that the tables are equal. Check the columns of the tables with a numeric comparator.

```
import matlab.unittest.constraints.TableComparator
import matlab.unittest.constraints.NumericComparator
import matlab.unittest.constraints.IsEqualTo
testCase.verifyThat(T1,IsEqualTo(T2, ...
    'Using',TableComparator(NumericComparator)))
```

```
Interactive verification passed.
```

Change the age of the last person to 50 and compare the tables again.

```
T2.Age(end) = 50;
testCase.verifyThat(T1,IsEqualTo(T2, ...
    'Using',TableComparator(NumericComparator)))
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
IsEqualTo failed.
--> Path to failure: <Value>.Age
    --> NumericComparator failed.
        --> The values are not equal using "isequaln".
        --> Failure table:
            value =

              1×5 table array

                Index    Actual    Expected    Error    RelativeError
                _____    ______    ________    _____    _____________

                3        49        50          -1       -0.02

        Actual double:
```

```
                38
                40
                49
        Expected double:
                38
                40
                50

Actual table:
    value =

      3×4 table array

                    Age    Height    Weight    BloodPressure
                    ___    ______    ______    _____________

        Williams    38     64        131       125     83
        Jones       40     67        133       117     75
        Brown       49     64        119       122     80
Expected table:
    value =

      3×4 table array

                    Age    Height    Weight    BloodPressure
                    ___    ______    ______    _____________

        Williams    38     64        131       125     83
        Jones       40     67        133       117     75
        Brown       50     64        119       122     80
```

See Also

`matlab.unittest.constraints.IsEqualTo`

Introduced in R2017a

matlab.unittest.constraints.Throws class

Package: `matlab.unittest.constraints`
Superclasses: `matlab.unittest.constraints.Constraint`

Constraint specifying function handle that throws MException

Description

The `Throws` class creates a constraint that is satisfied only if the actual value is a function handle that throws a specific exception.

If the function throws an `MException` and the `ExpectedException` property of the constraint is an error identifier, a qualification failure occurs if the actual `MException` thrown has a different identifier. Alternately, if the `ExpectedException` property is a `meta.class`, the constraint is not satisfied if the actual `MException` thrown does not derive from the `ExpectedException`.

Construction

`outConstObj = Throws(excep)` provides a constraint, `outConstObj`, specifying a function handle that throws a particular `MException`, `excep`.

`outConstObj = Throws(excep,Name,Value)` provides a constraint with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`excep`

Error identifier or `meta.class` representing the specific type of expected exception. If `excep` is a `meta.class` but does not derive from `MException`, the `Throws` constructor throws an `MException`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`CausedBy`

Expected causes, specified as a cell array of character vectors, a string array, or an array of `meta.class` instances.

The testing results in a qualification failure if any causes specified in `CausedBy` are not found within the cause tree.

Default: `{}`

`WhenNargoutIs`

Number of outputs the constraint should request when invoking the function handle, specified as a non-negative, real, scalar integer.

Default: `0`

`RespectingSet`

Indicator whether to respect the set of expected causes, specified as `false` or `true` (`logical` `0` or `1`). When this value is `false`, the instance ignores additional causes. When set to `true`, the instance is sensitive to additional causes. A `true` value means that the constraint is not satisfied if the expected exceptions contain causes that are not specified in the `'CausedBy'` name-value pair.

Default: `0`

Properties

`ExpectedException`

Expected `MException` identifier or class. Set this read-only property through the constructor via the `excep` input argument.

`Nargout`

Number of output arguments the instance uses when executing functions. Set this property through the constructor via the name-value pair argument, `'WhenNargoutIs'`.

RequiredCauses

Expected causes for the function handle throwing an `MException`. Set this property through the constructor via the name-value pair argument, `'CausedBy'`.

RespectSet

Indicator if the constraint respects set elements, specified through the constructor via the name-value pair argument, `'RespectingSet'`.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Instantiate Throws Constraint

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.Throws

testCase = TestCase.forInteractiveUse;
```

Verify that a function throws a specified error id.

```
testCase.verifyThat(@() error('SOME:error:id','Error!'), ...
    Throws('SOME:error:id'))
```

```
Interactive verification passed.
```

Verify that a function throws a specified exception class.

```
testCase.verifyThat(@() error('SOME:error:id','Error!'), ...
    Throws(?MException))
```

```
Interactive verification passed.
```

Verify that a function, when called with a specified number of outputs, throws a specified error.

```
testCase.verifyThat(@() disp('hi'), Throws('MATLAB:maxlhs', ...
    'WhenNargoutIs', 1))
```

```
Interactive verification passed.
```

Check causes by identifier.

```
me = MException('TOP:error:id','TopLevelError!');
causeBy1 = MException('causedBy:someOtherError:id','CausedByError!');
causeBy2 = MException('causedBy:yetAnotherError:id','AnotherCausedByError!');
me  = me.addCause(causeBy1);
me  = me.addCause(causeBy2);

testCase.verifyThat(@() me.throw, Throws('TOP:error:id','CausedBy',...
    {'causedBy:someOtherError:id'}))
```

```
Interactive verification passed.
```

Check that the exception does not include any additional causes than the ones specified by `'CausedBy'`.

```
testCase.verifyThat(@() me.throw, Throws('TOP:error:id','CausedBy',...
    {'causedBy:someOtherError:id'},'RespectingSet',true))
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
Throws failed.
--> The following causes were unexpectedly found in the exception tree:
    --> 'causedBy:yetAnotherError:id'

Actual Error Report:
    Error using @()me.throw
    TopLevelError!

    Caused by:
        CausedByError!
        AnotherCausedByError!

Actual Error Structure:
    ?MException 'TOP:error:id'
    --> ?MException 'causedBy:someOtherError:id'
    --> ?MException 'causedBy:yetAnotherError:id'

Evaluated Function:
        @()me.throw
```

Check causes by class.

```
me = MException('TOP:error:id','TopLevelError!');
causeBy = MException('causedBy:someOtherError:id','CausedByError!');
me = me.addCause(causeBy);
testCase.verifyThat(@() me.throw, Throws('TOP:error:id','CausedBy', ...
    ?MException))
```

```
Interactive verification passed.
```

Verify that if the actual value is not a function handle, the constraint is not satisfied.

```
testCase.fatalAssertThat(5, Throws('some:id'))
```

```
Interactive fatal assertion failed.

----------------------
Framework Diagnostic:
----------------------
Throws failed.
--> The value must be an instance of the expected type.

    Actual Class:
        double
    Expected Type:
        function_handle

Actual Value:
         5
Fatal assertion failed.
```

Verify that if the function does not throw an exception, the constraint is not satisfied.

```
testCase.assumeThat(@rand, Throws(?MException))
```

```
Interactive assumption failed.

----------------------
Framework Diagnostic:
----------------------
Throws failed.
--> The function did not throw any exception.

    Expected Exception:
        --> ?MException
```

```
Evaluated Function:
        @rand
Assumption failed.
```

Verify that if the function issues a non-specified error identifier, the constraint is not satisfied.

```
testCase.verifyThat(@() error('SOME:id','Error!'), Throws('OTHER:id'))
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
Throws failed.
--> The function threw the wrong exception.

    Actual Exception:
        'SOME:id'
    Expected Exception:
        'OTHER:id'

Actual Error Report:
    Error using @()error('SOME:id','Error!')
    Error!

Evaluated Function:
        @()error('SOME:id','Error!')
```

Verify that if the function throws an exception and the cause does not match the specified identifier, the constraint is not satisfied.

```
testCase.verifyThat(@() error('TOP:error:id','TopLevelError!'), ...
    Throws('TOP:error:id','CausedBy',{'causedBy:someOtherError:id'}))
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
Throws failed.
--> The following causes were not found in the exception tree:
    --> 'causedBy:someOtherError:id'
```

```
Actual Error Report:
    Error using @()error('TOP:error:id','TopLevelError!')
    TopLevelError!

Actual Error Structure:
    ?MException 'TOP:error:id'

Evaluated Function:
        @()error('TOP:error:id','TopLevelError!')
```

See Also

MException | error | matlab.unittest.constraints | matlab.unittest.constraints.IssuesWarnings

Topics

"Message Identifiers"

Introduced in R2013a

matlab.unittest.constraints.Tolerance class

Package: `matlab.unittest.constraints`

Abstract interface class for tolerances

Description

Tolerances define a notion of fuzzy equality for a set of data types and can be plugged in to the `IsEqualTo` constraint through the `'Within'` name-value pair argument.

Classes that derive from the `Tolerance` interface class must provide a tolerance definition. Use the `satisfiedBy` method to implement the tolerance definition. Classes that derive from the `Tolerance` class also must provide a diagnostic for two compared values. The testing framework uses the diagnostic when the compared values are outside of the allowable tolerance. Use the `getDiagnosticFor` method to implement this condition. Finally, classes that derive from the `Tolerance` class must provide a means to determine which data types the tolerance supports. Define the supported data types by implementing the `supports` method.

Methods

getDiagnosticFor	Produce diagnostic for two values specified to be within tolerance
satisfiedBy	Determine whether two values are within tolerance
supports	Determine whether tolerance supports specified data type

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

See Also

Topics

"Create Custom Tolerance"

getDiagnosticFor

Class: `matlab.unittest.constraints.Tolerance`
Package: `matlab.unittest.constraints`

Produce diagnostic for two values specified to be within tolerance

Syntax

```
diag = getDiagnosticFor(tolObj,actVal,expVal)
```

Description

`diag = getDiagnosticFor(tolObj,actVal,expVal)` produces a diagnostic, `diag`, for a value, `actVal`, evaluated against another value, `expVal`, within the tolerance defined by `tolObj`. When creating a custom tolerance, the class author must implement the `getDiagnosticFor` method so that it analyzes the two values, `actVal` and `expVal`, against the tolerance, `tolObj`, and instantiates and returns a `matlab.unittest.diagnostics.Diagnostic` object.

Typically, this diagnostic is used when the `getDiagnosticFor` method of `IsEqualTo` is invoked, and the result is incorporated into the diagnostic output of the `IsEqualTo` constraint.

Input Arguments

`actVal`

Value to determine if is within tolerance of `expVal`

`tolObj`

`Tolerance` instance

`expVal`

Expected value

See Also

`ConstraintDiagnostic` | `Diagnostic` | `satisfiedBy` | `supports`

Topics

"Create Custom Tolerance"

satisfiedBy

Class: `matlab.unittest.constraints.Tolerance`
Package: `matlab.unittest.constraints`

Determine whether two values are within tolerance

Syntax

```
TF = satisfiedBy(tolObj,actVal,expVal)
```

Description

`TF = satisfiedBy(tolObj,actVal,expVal)` determines whether two values, `actVal` and `expVal`, are within the tolerance defined by `tolObj`. The `satisfiedBy` method is used to determine whether the tolerance is met. It returns `true` or `false` (`logical` `0` or `1`). When creating a custom tolerance, a class author uses this method to contain the tolerance definition.

Input Arguments

`actVal`

Value to determine if is within tolerance of `expVal`

`tolObj`

`Tolerance` instance

`expVal`

Expected value

See Also

`getDiagnosticFor` | `supports`

Topics

"Create Custom Tolerance"

supports

Class: `matlab.unittest.constraints.Tolerance`
Package: `matlab.unittest.constraints`

Determine whether tolerance supports specified data type

Syntax

```
TF = supports(tolObj,typeVal)
```

Description

`TF = supports(tolObj,typeVal)` determines whether the tolerance supports a specific data type. It returns `true` or `false` (`logical` `0` or `1`).

The `supports` method provides the ability for a tolerance author to specify support for data types. Generally, the method operates by examining the type of `typeVal` to determine whether it is supported.

Input Arguments

`tolObj`

`Tolerance` instance

`typeVal`

Value used to determine tolerance support

See Also

`getDiagnosticFor` | `satisfiedBy`

Topics

"Create Custom Tolerance"

matlab.unittest.constraints.AndConstraint class

Package: `matlab.unittest.constraints`

Boolean conjunction of two constraints

Description

The testing framework constructs an `AndConstraint` when you denote the conjunction of two constraints with the logical AND operator (&). The framework constructs instances of the class, so there is no need to construct this class directly.

Properties

`FirstConstraint` — Left side constraint
instance of class in `matlab.unittest.constraint` package

This property is read-only.

Left side constraint in the Boolean conjunction, specified as an instance of a class in the `matlab.unittest.constraint` package.

`SecondConstraint` — Right side constraint
instance of class in `matlab.unittest.constraint` package

This property is read-only.

Right side constraint in the Boolean conjunction, specified as an instance of a class in the `matlab.unittest.constraint` package.

See Also

`matlab.unittest.constraints` | `matlab.unittest.constraints.NotConstraint` | `matlab.unittest.constraints.OrConstraint`

Introduced in R2013a

matlab.unittest.constraints.OrConstraint class

Package: `matlab.unittest.constraints`

Boolean disjunction of two constraints

Description

The testing framework constructs an `OrConstraint` when you denote the disjunction of two constraints with the logical OR operator (`|`). The framework constructs instances of the class, so there is no need to construct this class directly.

Properties

`FirstConstraint` — Left side constraint
instance of class in `matlab.unittest.constraint` package

This property is read-only.

Left side constraint in the Boolean disjunction, specified as an instance of a class in the `matlab.unittest.constraint` package.

`SecondConstraint` — Right side constraint
instance of class in `matlab.unittest.constraint` package

This property is read-only.

Right side constraint in the Boolean disjunction, specified as an instance of a class in the `matlab.unittest.constraint` package.

See Also

`matlab.unittest.constraints` | `matlab.unittest.constraints.AndConstraint` | `matlab.unittest.constraints.NotConstraint`

Introduced in R2013a

matlab.unittest.constraints.NotConstraint class

Package: `matlab.unittest.constraints`

Boolean complement of constraint

Description

The testing framework constructs a `NotConstraint` when you denote the complement of a constraint with the logical NOT operator (~). The framework constructs instances of the class, so there is no need to construct this class directly.

Properties

`Constraint` — Constraint being complemented
instance of class in `matlab.unittest.constraints` package

This property is read-only.

Constraint being complemented, specified as an instance of a class in the `matlab.unittest.constraints` package.

See Also

`matlab.unittest.constraints` | `matlab.unittest.constraints.AndConstraint` | `matlab.unittest.constraints.OrConstraint`

Introduced in R2013a

matlab.unittest.constraints.AndTolerance class

Package: `matlab.unittest.constraints`

Boolean conjunction of two tolerances

Description

The testing framework constructs an `AndTolerance` when you denote the conjunction of two tolerances with the logical AND operator (&). The framework constructs instances of the class, so there is no need to construct this class directly.

See Also

`matlab.unittest.constraints.OrTolerance`

Introduced in R2013a

matlab.unittest.constraints.OrTolerance class

Package: `matlab.unittest.constraints`

Boolean disjunction of two constraints

Description

The testing framework constructs an `OrTolerance` when you denote the disjunction of two tolerances with the logical OR operator (`|`). The framework constructs instances of the class, so there is no need to construct this class directly.

See Also

`matlab.unittest.constraints.AndTolerance`

Introduced in R2013a

contrast

Create grayscale colormap to enhance image contrast

Syntax

```
newmap = contrast(I)
newmap = contrast(I,m)
```

Description

`newmap = contrast(I)` creates a grayscale colormap that enhances the contrast of image `I`. The new colormap increases the contrast among pixels that are difficult to distinguish visually, but have slightly different brightness values.

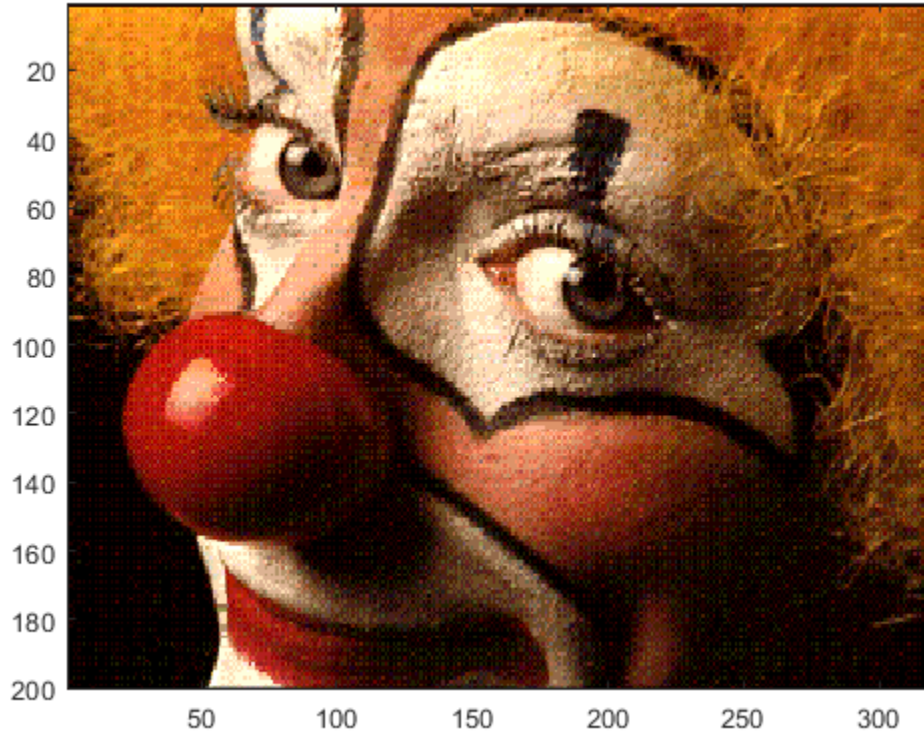
`newmap = contrast(I,m)` returns the new colormap as an m-by-3 array. Use this syntax when you want the new colormap to have a different number of rows than the original colormap.

Examples

Enhance the Contrast of an Image

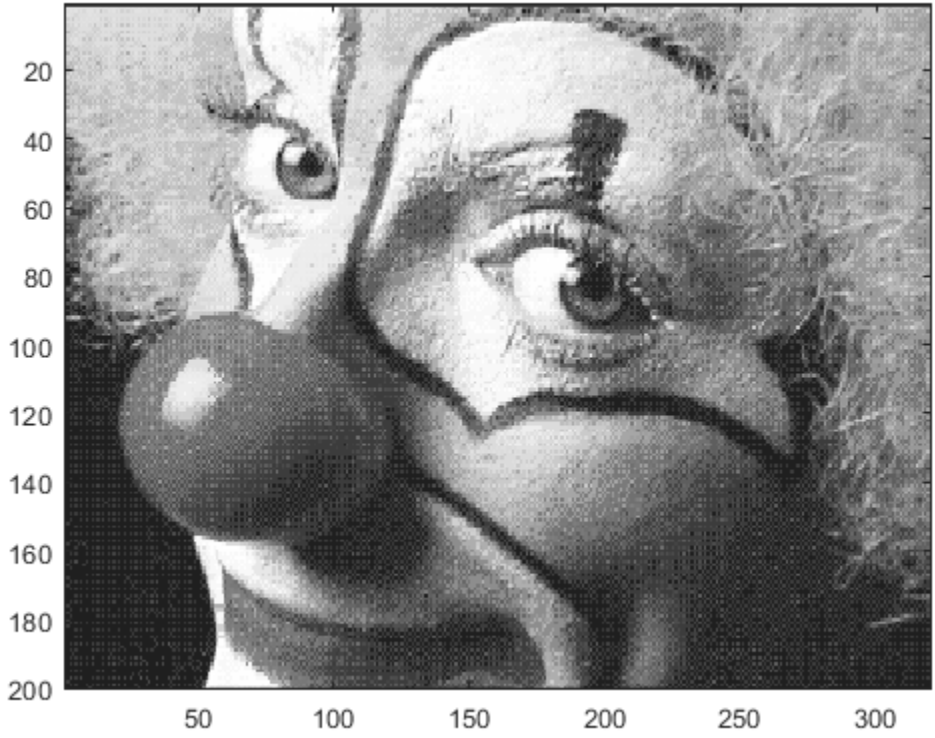
Load `clown` to get image `X` and its associated colormap, `map`. Display the image produced by `X` and `map`.

```
load clown
imagesc(X)
colormap(map)
```



Use `contrast` to return a gray colormap that enhances the contrast of image X. Then update the display with the new colormap.

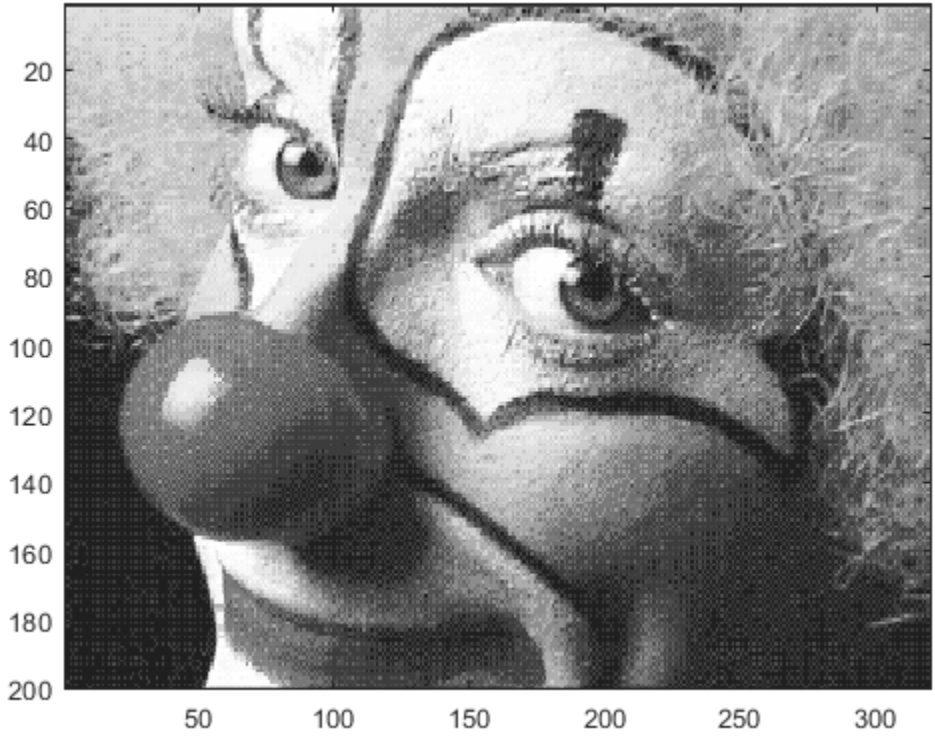
```
newmap = contrast(X);
colormap(newmap)
```



Display Image with Fewer Gray Levels

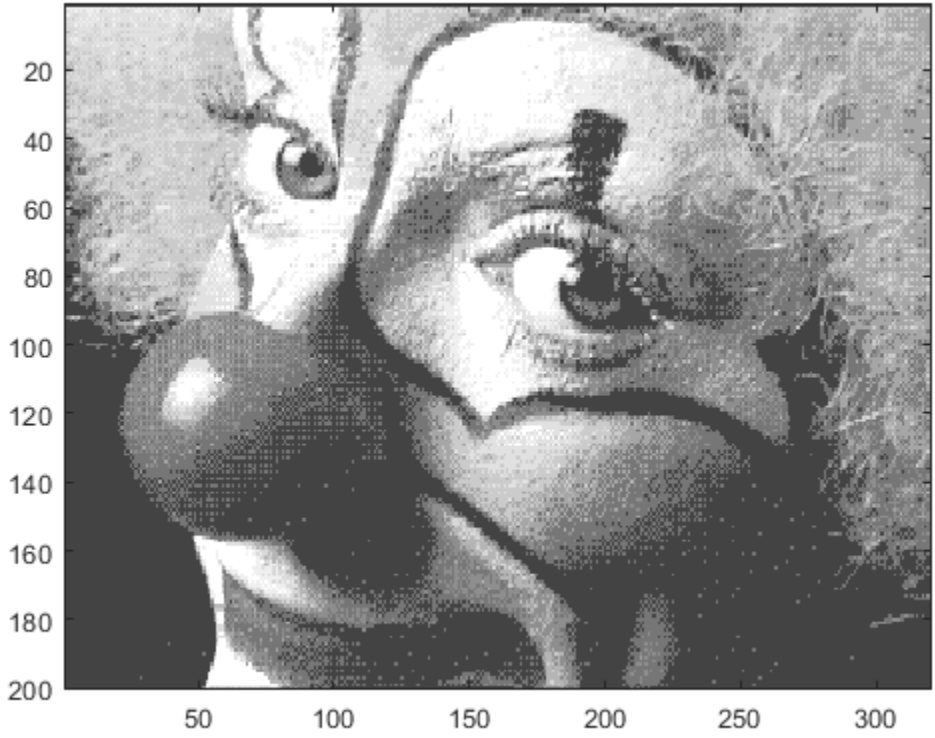
Load `clown` to get image X. Then create a contrast-enhancing colormap, and display the image using that colormap.

```
load clown
imagesc(X)
newmap1 = contrast(X);
colormap(newmap1)
```



Next, use `contrast` to create another colormap containing only 10 shades of gray. Update the display with the new colormap. Notice that the shadow areas are lighter and have lost some detail.

```
newmap2 = contrast(X,10);
colormap(newmap2)
```



Input Arguments

`I` — Image
m-by-n matrix | m-by-n-by-3 truecolor image

Image to enhance, specified as an m-by-n matrix or an m-by-n-by-3 truecolor image.

`m` — Number of rows in new colormap
numeric scalar value

Number of rows in the new colormap, specified as a numeric scalar value.

Note When m is significantly different than the number of rows in the original colormap, the new colormap might be missing some values or contain repeated values. This discrepancy might reduce the amount of tonal detail in the enhanced image.

Example: `newmap = contrast(I,50)` returns a colormap containing 50 rows.

See Also

`brighten` | `colormap` | `image`

Introduced before R2006a

conv

Convolution and polynomial multiplication

Syntax

```
w = conv(u,v)
w = conv(u,v,shape)
```

Description

`w = conv(u,v)` returns the convolution on page 1-2412 of vectors `u` and `v`. If `u` and `v` are vectors of polynomial coefficients, convolving them is equivalent to multiplying the two polynomials.

`w = conv(u,v,shape)` returns a subsection of the convolution, as specified by `shape`. For example, `conv(u,v,'same')` returns only the central part of the convolution, the same size as `u`, and `conv(u,v,'valid')` returns only the part of the convolution computed without the zero-padded edges.

Examples

Polynomial Multiplication via Convolution

Create vectors `u` and `v` containing the coefficients of the polynomials $x^2 + 1$ and $2x + 7$.

```
u = [1 0 1];
v = [2 7];
```

Use convolution to multiply the polynomials.

```
w = conv(u,v)
```

```
w = 1×4
```

```
     2     7     2     7
```

w contains the polynomial coefficients for $2x^3 + 7x^2 + 2x + 7$.

Vector Convolution

Create two vectors and convolve them.

```
u = [1 1 1];
v = [1 1 0 0 0 1 1];
w = conv(u,v)
```

```
w = 1×9

     1     2     2     1     0     1     2     2     1
```

The length of w is `length(u)+length(v)-1`, which in this example is 9.

Central Part of Convolution

Create two vectors. Find the central part of the convolution of u and v that is the same size as u.

```
u = [-1 2 3 -2 0 1 2];
v = [2 4 -1 1];
w = conv(u,v,'same')
```

```
w = 1×7

    15     5    -9     7     6     7    -1
```

w has a length of 7. The full convolution would be of length `length(u)+length(v)-1`, which in this example would be 10.

Input Arguments

u,v — Input vectors
vectors

Input vectors, specified as either row or column vectors. The vectors u and v can be different lengths or data types.

When u or v are of type `single`, then the output is of type `single`. Otherwise, `conv` converts inputs to type `double` and returns type `double`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

shape — Subsection of convolution
`'full'` (default) | `'same'` | `'valid'`

Subsection of the convolution, specified as `'full'`, `'same'`, or `'valid'`.

`'full'`	Full convolution (default).
`'same'`	Central part of the convolution of the same size as u.
`'valid'`	Only those parts of the convolution that are computed without the zero-padded edges. Using this option, `length(w)` is `max(length(u)-length(v)+1,0)`, except when `length(v)` is zero. If `length(v) = 0`, then `length(w) = length(u)`.

More About

Convolution

The convolution of two vectors, u and v, represents the area of overlap under the points as v slides across u. Algebraically, convolution is the same operation as multiplying polynomials whose coefficients are the elements of u and v.

Let `m = length(u)` and `n = length(v)` . Then w is the vector of length `m+n-1` whose kth element is

$$w(k) = \sum_j u(j)v(k - j + 1) .$$

The sum is over all the values of `j` that lead to legal subscripts for `u(j)` and `v(k-j+1)`, specifically `j = max(1,k+1-n):1:min(k,m)`. When `m = n`, this gives

```
w(1) = u(1)*v(1)
w(2) = u(1)*v(2)+u(2)*v(1)
w(3) = u(1)*v(3)+u(2)*v(2)+u(3)*v(1)
...
w(n) = u(1)*v(n)+u(2)*v(n-1)+ ... +u(n)*v(1)
...
w(2*n-1) = u(n)*v(n)
```

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- The inputs `u` and `v` must be column vectors.
- If `shape` is `'full'` (default), then only one of `u` or `v` can be a tall array.
- If `shape` is `'same'` or `'valid'`, then `v` cannot be a tall array.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For information about C/C++ code generation limitations, see “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`conv2` | `convmtx` | `convn` | `deconv` | `filter` | `xcorr`

Introduced before R2006a

conv2

2-D convolution

Syntax

```
C = conv2(A,B)
C = conv2(u,v,A)
C = conv2( ___ ,shape)
```

Description

`C = conv2(A,B)` returns the two-dimensional convolution on page 1-2422 of matrices `A` and `B`.

`C = conv2(u,v,A)` first convolves each column of `A` with the vector `u`, and then it convolves each row of the result with the vector `v`.

`C = conv2( ___ ,shape)` returns a subsection of the convolution according to `shape`. For example, `C = conv2(A,B,'same')` returns the central part of the convolution, which is the same size as `A`.

Examples

2-D Convolution

In applications such as image processing, it can be useful to compare the input of a convolution directly to the output. The `conv2` function allows you to control the size of the output.

Create a 3-by-3 random matrix `A` and a 4-by-4 random matrix `B`. Compute the full convolution of `A` and `B`, which is a 6-by-6 matrix.

```
A = rand(3);
B = rand(4);
Cfull = conv2(A,B)
```

```
Cfull = 6×6

    0.7861    1.2768    1.4581    1.0007    0.2876    0.0099
    1.0024    1.8458    3.0844    2.5151    1.5196    0.2560
    1.0561    1.9824    3.5790    3.9432    2.9708    0.7587
    1.6790    2.0772    3.0052    3.7511    2.7593    1.5129
    0.9902    1.1000    2.4492    1.6082    1.7976    1.2655
    0.1215    0.1469    1.0409    0.5540    0.6941    0.6499
```

Compute the central part of the convolution `Csame`, which is a submatrix of `Cfull` with the same size as `A`. `Csame` is equal to `Cfull(3:5,3:5)`.

```
Csame = conv2(A,B,'same')
```

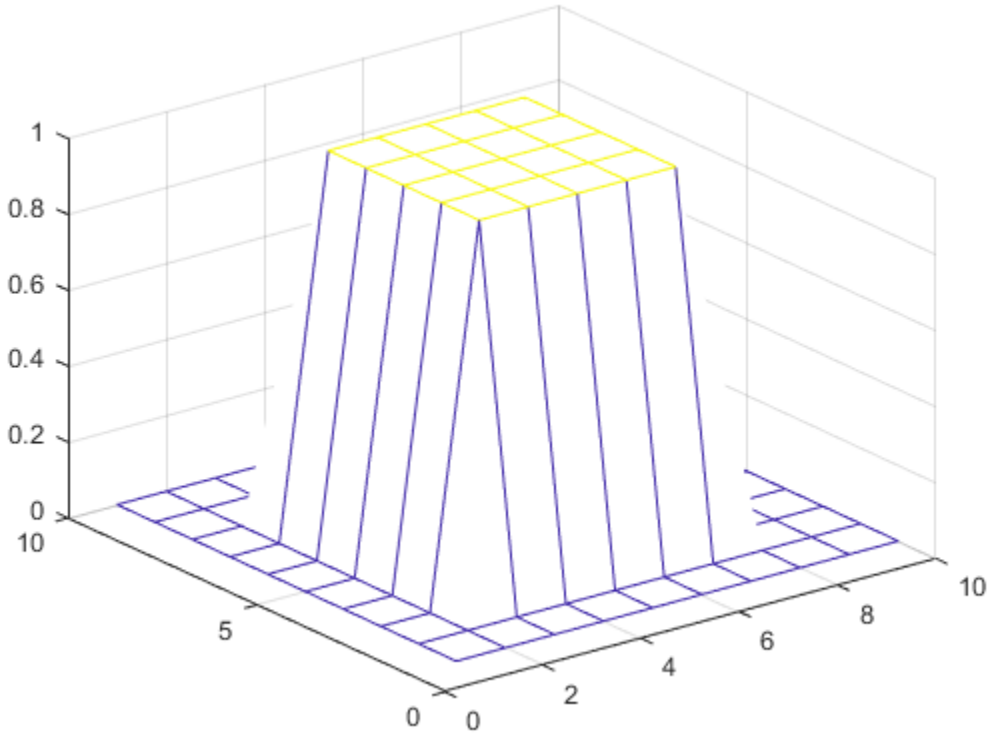
```
Csame = 3×3

    3.5790    3.9432    2.9708
    3.0052    3.7511    2.7593
    2.4492    1.6082    1.7976
```

Extract 2-D Pedestal Edges

The Sobel edge-finding operation uses a 2-D convolution to detect edges in images and other 2-D data.

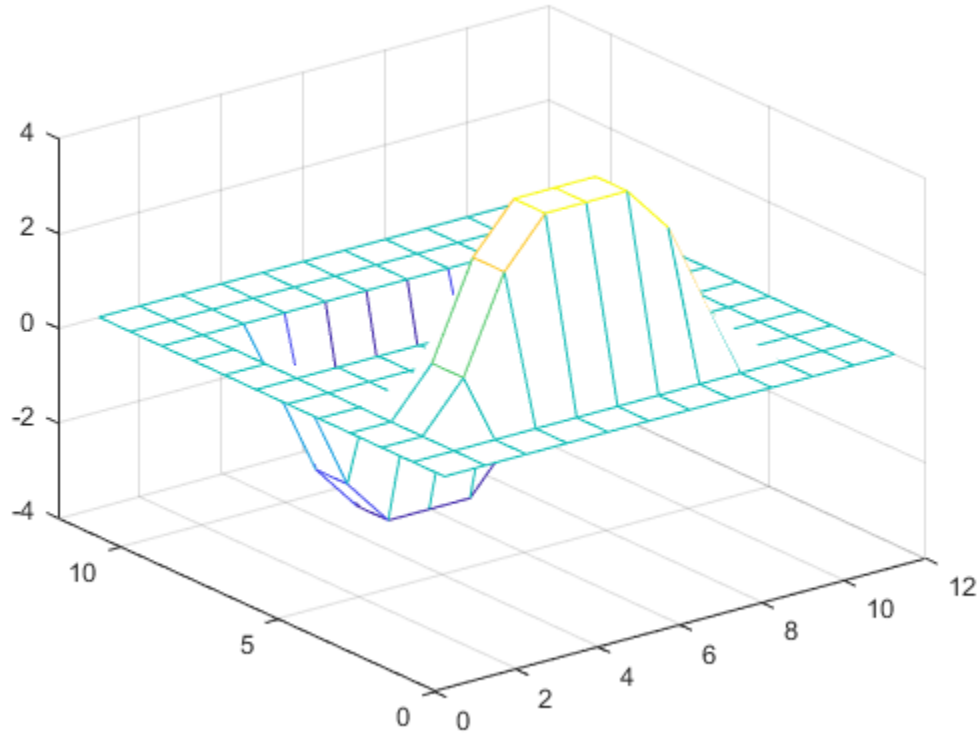
Create and plot a 2-D pedestal with interior height equal to one.

```
A = zeros(10);
A(3:7,3:7) = ones(5);
mesh(A)
```



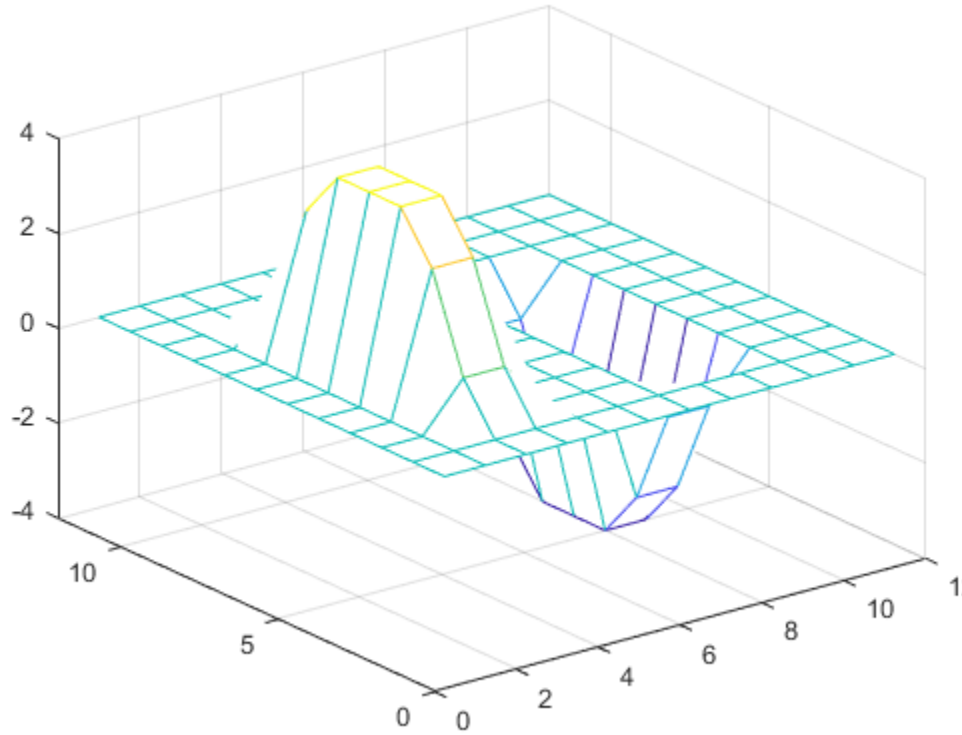
Convolve the rows of A with the vector u, and then convolve the rows of the result with the vector v. The convolution extracts the horizontal edges of the pedestal.

```
u = [1 0 -1]';
v = [1 2 1];
Ch = conv2(u,v,A);
mesh(Ch)
```



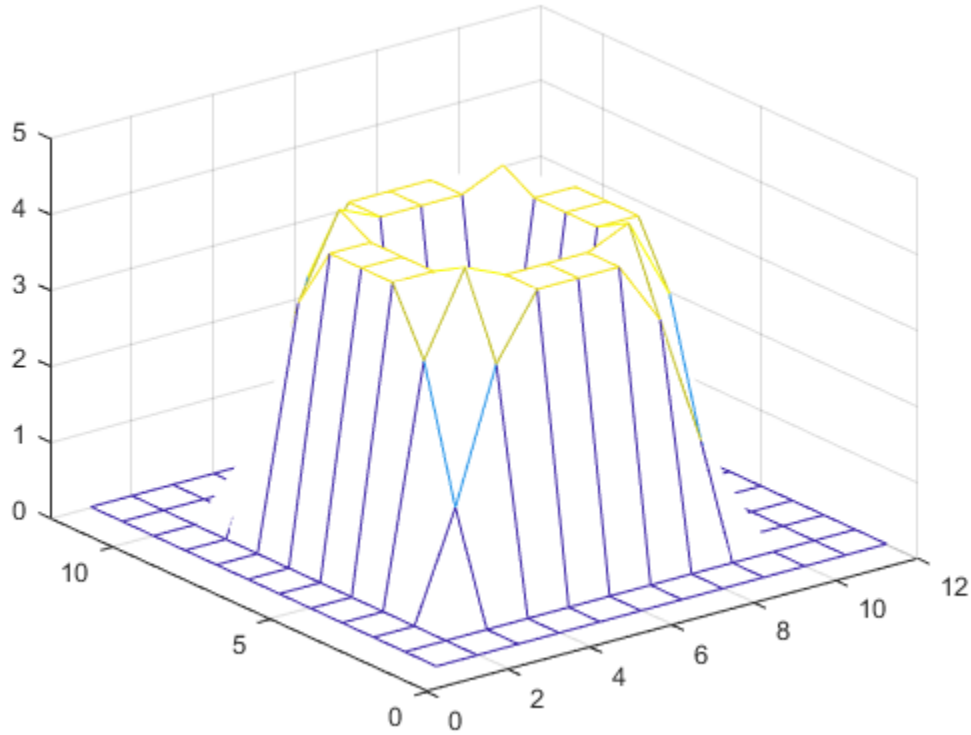
To extract the vertical edges of the pedestal, reverse the order of convolution with `u` and `v`.

```
Cv = conv2(v,u,A);
mesh(Cv)
```



Compute and plot the combined edges of the pedestal.

```
figure
mesh(sqrt(Ch.^2 + Cv.^2))
```



Input Arguments

A — Input array
vector | matrix

Input array, specified as a vector or matrix.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

B — Second input array
vector | matrix

Second input array, specified as a vector or a matrix to convolve with A. The array B does not have to be the same size as A.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical
Complex Number Support: Yes

u — Input vector
row or column vector

Input vector, specified as a row or column vector. u convolves with each column of A.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical
Complex Number Support: Yes

v — Second input vector
row or column vector

Second input vector, specified as a row or column vector. v convolves with each row of the convolution of u with the columns of A.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical
Complex Number Support: Yes

shape — Subsection of convolution
'full' (default) | 'same' | 'valid'

Subsection of the convolution, specified as one of these values:

- 'full' — Return the full 2-D convolution.
- 'same' — Return the central part of the convolution, which is the same size as A.
- 'valid' — Return only parts of the convolution that are computed without zero-padded edges.

Output Arguments

C — 2-D convolution
vector | matrix

2-D convolution, returned as a vector or matrix. When A and B are matrices, then the convolution `C = conv2(A,B)` has size `size(A)+size(B)-1`. When `[m,n] = size(A)`, `p = length(u)`, and `q = length(v)`, then the convolution `C = conv2(u,v,A)` has `m+p-1` rows and `n+q-1` columns.

When one or more input arguments to `conv2` are of type `single`, then the output is of type `single`. Otherwise, `conv2` converts inputs to type `double` and returns type `double`.

Data Types: `double` | `single`

More About

2-D Convolution

For discrete, two-dimensional variables A and B, the following equation defines the convolution of A and B:

$$C(j,k) = \sum_p \sum_q A(p,q)B(j-p+1, k-q+1)$$

p and q run over all values that lead to legal subscripts of *A(p,q)* and *B(j-p+1,k-q+1)*.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- If `shape` is `'full'` (default), then the inputs `A` and `B` must not be empty and only one them can be a tall array.
- If `shape` is `'same'` or `'valid'`, then `B` cannot be a tall array.
- `u` and `v` cannot be tall arrays.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- Input vectors `u` and `v` must not be distributed arrays.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`conv` | `convn`

Introduced before R2006a

convertCharsToStrings

Convert character arrays to string arrays, leaving other arrays unaltered

Syntax

```
B = convertCharsToStrings(A)
[B1,...,Bn] = convertCharsToStrings(A1,...,An)
```

Description

When working with your own code, you can use `convertCharsToStrings` to make your code accept character arrays. Then you do not have to make any other changes to code you had written to work with string arrays.

`B = convertCharsToStrings(A)` converts `A` to a string array if `A` is a character array or a cell array of character vectors. If `A` has any other data type, then `convertCharsToStrings` returns `A` unaltered.

`[B1,...,Bn] = convertCharsToStrings(A1,...,An)` converts any character arrays or cell arrays of character vectors in `A1,...,An` to string arrays, and then returns them as the corresponding output arguments in `B1,...,Bn`. If any of the arguments `A1,...,An` has any other data type, then `convertCharsToStrings` returns it unaltered.

Examples

Convert Character Arrays to String Arrays

Create a character vector and convert it to a string scalar.

```
chr = 'Mercury'
```

```
chr = 
'Mercury'
```

```
str = convertCharsToStrings(chr)
```

```
str =
"Mercury"
```

Convert a cell array of character vectors to a string array.

```
C = {'Venus','Earth','Mars'}
```

```
C = 1x3 cell array
    {'Venus'}    {'Earth'}    {'Mars'}
```

```
str = convertCharsToStrings(C)
```

```
str = 1x3 string array
    "Venus"    "Earth"    "Mars"
```

Process and Return Input Arrays

Process an arbitrary number of input arrays of different types, converting only the character arrays to string arrays.

Create a set of numeric, character, and string arrays.

```
A = [1 2 3]
```

```
A = 1×3

     1     2     3
```

```
str = ["Mercury","Gemini","Apollo"]
```

```
str = 1x3 string array
    "Mercury"    "Gemini"    "Apollo"
```

```
B = [2 5; 7 6]
```

```
B = 2×2
```

```
     2     5
     7     6
```

```
C = {'volts','amps'}
```

```
C = 1x2 cell array
    {'volts'}    {'amps'}
```

Convert the character array and leave the other arrays unaltered.

```
[newA,newStr,newB,newC] = convertCharsToStrings(A,str,B,C)
```

```
newA = 1×3

     1     2     3


newStr = 1x3 string array
    "Mercury"    "Gemini"    "Apollo"


newB = 2×2

     2     5
     7     6


newC = 1x2 string array
    "volts"    "amps"
```

Input Arguments

A — Input array
array of any size and data type

Input array, specified as an array of any size or data type.

Output Arguments

B — Output array
array

Output array. The data type of the output array depends on the data type of the input array, A.

- If A is a character vector, then B is a string scalar.
- If A is a cell array of character vectors, then B is a string array that has the same size.
- If A is a character array with multiple rows, then the columns of A are concatenated and B is returned as a string scalar. For example, the 3-by-2 character array `['Xx';'Yy';'Zz']` is converted to `"XYZxyz"`.
- If A has any other data type, then B is identical to A.

Tips

- To enable code that works with strings to accept character arrays as inputs, add a call to `convertCharsToStrings` at the beginning of your code.

  For example, if you have defined a function `myFunc` that accepts three input arguments, process all three inputs using `convertCharsToStrings`. Leave the rest of your code unchanged.

  ```
  function y = myFunc(a,b,c)
      [a,b,c] = convertCharsToStrings(a,b,c);
      <line 1 of original code>
      <line 2 of original code>
      ...
  ```

  In this example, the output arguments `[a,b,c]` overwrite the input arguments in place. If any input argument is not a character array or a cell array of character vectors, then it is unaltered.

  If `myFunc` accepts a variable number of input arguments, then process all the arguments specified by `varargin`.

  ```
  function y = myFunc(varargin)
      [varargin{:}] = convertCharsToStrings(varargin{:});
      ...
  ```

- The `convertCharsToStrings` function is more efficient when converting one input argument. If performance is a concern, then call `convertCharsToStrings` on one input argument at a time, rather than calling it once on multiple inputs.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- A cell array input must be scalar at compile time.

See Also

`convertContainedStringsToChars` | `convertStringsToChars` | `isStringScalar` | `iscellstr` | `ischar` | `ismissing` | `isstring` | `string` | `varargin`

Topics

"Update Your Code to Accept Strings"

Introduced in R2017b

convertContainedStringsToChars

Convert string arrays at any level of cell array or structure

Syntax

```
B = convertContainedStringsToChars(A)
[B1,...,Bn] = convertContainedStringsToChars(A1,...,An)
```

Description

To make your code accept cell arrays and structures that contain strings as input arguments, add a call to `convertContainedStringsToChars` to the beginning of your code. Then you do not have to make any other changes to code that you had written to work with cell arrays or structures containing character arrays.

`B = convertContainedStringsToChars(A)` converts string arrays at any level in `A`.

- If `A` is a string array, then `B` is a character vector or cell array of character vectors.
- If `A` is a cell array or a structure, then string arrays in any cell or field of `A` become character vectors or cell arrays of character vectors in `B`. All other cells or fields of `A` are unaltered in `B`.
- Otherwise, the function returns `A` unaltered.

`[B1,...,Bn] = convertContainedStringsToChars(A1,...,An)` converts the input arguments `A1,...,An`. For every input argument, there must be a corresponding output argument in `B1,...,Bn`.

Examples

Convert Strings in Cell Array

Create a cell array containing launch dates, spacecraft names, and planets visited. Some cells contain string arrays, and others do not.

```
C = {2004,"Messenger","Mercury"; ...
     1977,"Voyager 1",["Jupiter","Saturn"]; ...
     2006,"New Horizons","Pluto"}
```

```
C=3×3 cell
    {[2004]}    {["Messenger"   ]}    {["Mercury"]}
    {[1977]}    {["Voyager 1"   ]}    {1x2 string }
    {[2006]}    {["New Horizons"]}    {["Pluto"  ]}
```

Convert the strings in `C` to character vectors.

```
C = convertContainedStringsToChars(C)
```

```
C=3×3 cell
    {[2004]}    {'Messenger'   }    {'Mercury'}
    {[1977]}    {'Voyager 1'   }    {1x2 cell }
    {[2006]}    {'New Horizons'}    {'Pluto'  }
```

Note that the function converts the string array `["Jupiter","Saturn"]` to a cell array of character vectors, contained in cell `C(2,3)`. To access the contents of the cell, use curly brace indexing.

```
C{2,3}
```

```
ans = 1x2 cell array
    {'Jupiter'}    {'Saturn'}
```

Convert Strings in Structure

Create a structure containing arrays of data, a title, and labels for a plot. Some of the fields contain strings, and others do not.

```
S.x = linspace(0,2*pi);
S.y = sin(S.x);
S.title = "y = sin(x)";
S.axislabels = ["x (radians)","y"]
```

```
S = struct with fields:
             x: [1x100 double]
             y: [1x100 double]
```

```
         title: "y = sin(x)"
    axislabels: ["x (radians)"    "y"]
```

Convert the strings in S.

```
S = convertContainedStringsToChars(S)
```

```
S = struct with fields:
              x: [1x100 double]
              y: [1x100 double]
         title: 'y = sin(x)'
    axislabels: {'x (radians)'  'y'}
```

Process and Return Multiple Inputs

Process an arbitrary number of input arrays of different types.

Create an array of doubles, a structure, and a cell array. The structure and cell array contain strings, specified using double quotes.

```
x = linspace(0,2*pi,8);
S.y = sin(x);
S.title = "y = sin(x)";
C = {{sin(x),"y = sin(x)"},{cos(x),"y = cos(x)"}};
```

Convert the strings and return all other data unaltered.

```
[x,S,C] = convertContainedStringsToChars(x,S,C)
```

```
x = 1×8

         0    0.8976    1.7952    2.6928    3.5904    4.4880    5.3856    6.2832


S = struct with fields:
        y: [0 0.7818 0.9749 0.4339 -0.4339 -0.9749 -0.7818 -2.4493e-16]
    title: 'y = sin(x)'
```

```
C=1×2 cell
    {1x2 cell}    {1x2 cell}
```

Display the contents of the cell array contained in the first cell, `C(1)`. The function converts strings that are in nested cells or nested structures to character vectors. Note that character vectors display with single quotes.

```
C{1}
```

```
ans=1×2 cell
    {1x8 double}    {'y = sin(x)'}
```

Input Arguments

A — Input array
array

Input array, specified as an array of any size or data type.

If `A` is a cell array or a structure, then it can be nested. A cell can contain another cell array, and a structure can have a field that is another structure. `convertContainedStringsToChars` converts every element or field value that is a string into a corresponding character vector or cell array of character vectors. The function converts each element or field value using the rules defined by the `convertStringsToChars` function, traversing every level and combination of nested containers.

See Also

`convertCharsToStrings` | `convertStringsToChars` | `isStringScalar` | `ischar` | `isstring` | `string` | `validateattributes`

Topics

"Update Your Code to Accept Strings"

Introduced in R2018b

convertStringsToChars

Convert string arrays to character arrays, leaving other arrays unaltered

Syntax

```
B = convertStringsToChars(A)
[B1,...,Bn] = convertStringsToChars(A1,...,An)
```

Description

When working with your own code, you can use `convertStringsToChars` to make your code accept string inputs. Then you do not have to make any other changes to code that you had written to work with character arrays.

`B = convertStringsToChars(A)` converts `A` to a character vector or a cell array of character vectors if `A` is a string array. Otherwise, `convertStringsToChars` returns `A` unaltered.

`[B1,...,Bn] = convertStringsToChars(A1,...,An)` converts any string arrays in `A1,...,An` to character vectors or cell arrays of character vectors, and then returns them as the corresponding output arguments in `B1,...,Bn`. If any of the arguments `A1,...,An` has any other data type, then `convertStringsToChars` returns it unaltered.

Examples

Convert String Arrays to Character Arrays

Create a string scalar and convert it to a character vector.

```
str = "Mercury"
```

```
str =
"Mercury"
```

```
chr = convertStringsToChars(str)
```

```
chr =
'Mercury'
```

Convert a string array to a cell array of character vectors.

```
str = ["Venus","Earth","Mars"]
```

```
str = 1x3 string array
    "Venus"    "Earth"    "Mars"
```

```
C = convertStringsToChars(str)
```

```
C = 1x3 cell array
    {'Venus'}    {'Earth'}    {'Mars'}
```

Process and Return Input Arrays

Process an arbitrary number of input arrays of different types, converting only the string arrays to character arrays.

Create a set of numeric, character, and string arrays.

```
A = [1 2 3]
```

```
A = 1×3

     1     2     3
```

```
str = ["Mercury","Gemini","Apollo"]
```

```
str = 1x3 string array
    "Mercury"    "Gemini"    "Apollo"
```

```
B = [2 5; 7 6]
```

```
B = 2×2
```

```
     2     5
     7     6


C = {'volts','amps'}

C = 1x2 cell array
    {'volts'}    {'amps'}
```

Convert the string array and return the other arrays unaltered.

```
[newA,newStr,newB,newC] = convertStringsToChars(A,str,B,C)

newA = 1×3

     1     2     3


newStr = 1x3 cell array
    {'Mercury'}    {'Gemini'}    {'Apollo'}


newB = 2×2

     2     5
     7     6


newC = 1x2 cell array
    {'volts'}    {'amps'}
```

Input Arguments

A — Input array
array of any size and data type

Input array, specified as an array of any size or data type.

Output Arguments

B — Output array
array

Output array. The data type of the output array depends on the data type of the input array, A.

- If A is a string scalar, then B is a character vector.
- If A is a string array of any other size, then B is a cell array of character vectors that has the same size.
- If A has any other data type, then B is identical to A.

If A is a string array, then convertStringsToChars converts any element that is:

- An empty string (displayed as "") to a 0-by-0 character array (displayed as '')
- A missing string (displayed as <missing>) to a 0-by-0 character array

If A is an empty string array, then B is an empty cell array. An empty array has at least one dimension whose size is 0.

Tips

- To enable your existing code to accept string arrays as input, add a call to convertStringsToChars at the beginning of your code.

  For example, if you have defined a function myFunc that accepts three input arguments, process all three inputs using convertStringsToChars. Leave the rest of your code unchanged.

  ```
  function y = myFunc(a,b,c)
      [a,b,c] = convertStringsToChars(a,b,c);
      <line 1 of original code>
      <line 2 of original code>
      ...
  ```

  In this example, the output arguments [a,b,c] overwrite the input arguments in place. If any input argument is not a string array, then it is unaltered.

If `myFunc` accepts a variable number of input arguments, then process all the arguments specified by `varargin`.

```
function y = myFunc(varargin)
    [varargin{:}] = convertStringsToChars(varargin{:});
    ...
```

- The `convertStringsToChars` function is more efficient when converting one input argument. If performance is a concern, then call `convertStringsToChars` on one input argument at a time, rather than calling it once on multiple inputs.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`convertCharsToStrings` | `convertContainedStringsToChars` | `isStringScalar` | `iscellstr` | `ischar` | `ismissing` | `isstring` | `string` | `varargin`

Topics

"Update Your Code to Accept Strings"

Introduced in R2017b

convertvars

Convert table or timetable variables to specified data type

Syntax

```
T2 = convertvars(T1,vars,dataType)
```

Description

`T2 = convertvars(T1,vars,dataType)` converts the specified variables to the specified data type. The input argument `T1` can be a table or timetable.

While you can specify `dataType` as the name of a data type, you also can specify it as a function handle. In that case, it is a handle to a function that converts or otherwise modifies the variables specified by `vars`. Similarly, `vars` can contain variable names or positions of variables in `T1`, or it can be a handle to a function that identifies variables.

Examples

Convert Table Variables

Read a table from a spreadsheet containing data on electric power outages. The table has text variables showing the region and cause for each power outage, datetime variables showing the outage and restoration times, and numeric variables showing the power loss and number of customers affected. Display the first five rows.

```
T1 = readtable('outages.csv');
head(T1,5)
```

```
ans=5×6 table
       Region           OutageTime        Loss     Customers     RestorationTime
    _____________    ________________    ______    __________    ________________    __

    {'SouthWest'}    2002-02-01 12:18    458.98    1.8202e+06    2002-02-07 16:50    {
```

```
    {'SouthEast'}    2003-01-23 00:49    530.14    2.1204e+05                 NaT    {
    {'SouthEast'}    2003-02-07 21:15     289.4    1.4294e+05    2003-02-17 08:14    {
    {'West'     }    2004-04-06 05:44    434.81    3.4037e+05    2004-04-06 06:10    {
    {'MidWest'  }    2002-03-16 06:18    186.44    2.1275e+05    2002-03-18 23:23    {
```

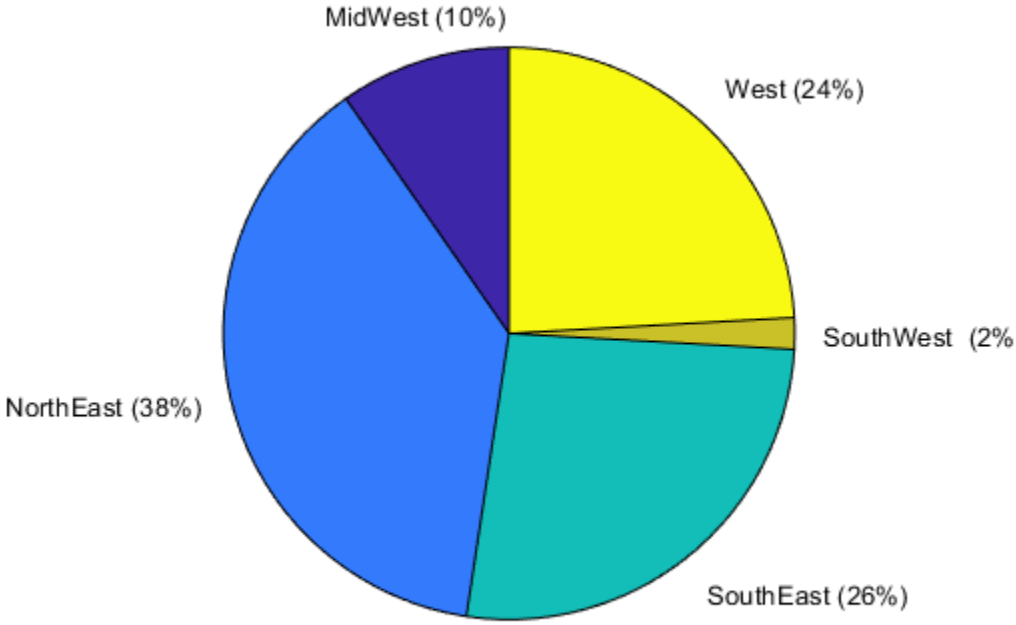
Convert the variables `Region` and `Cause` to categorical variables. Note that categorical values are not displayed with quotation marks.

```
T2 = convertvars(T1,{'Region','Cause'},'categorical');
head(T2,5)
```

```
ans=5×6 table
     Region         OutageTime        Loss     Customers     RestorationTime          C
    _________    _________________    ______    __________    _________________    ______

    SouthWest    2002-02-01 12:18    458.98    1.8202e+06    2002-02-07 16:50    winter
    SouthEast    2003-01-23 00:49    530.14    2.1204e+05                 NaT    winter
    SouthEast    2003-02-07 21:15     289.4    1.4294e+05    2003-02-17 08:14    winter
    West         2004-04-06 05:44    434.81    3.4037e+05    2004-04-06 06:10    equipm
    MidWest      2002-03-16 06:18    186.44    2.1275e+05    2002-03-18 23:23    severe
```

It can be convenient to convert variables to data types that offer different functionality. For example, now that `T2.Region` is a categorical variable, you can use the `pie` function to make a pie chart of power outages by region. But you cannot use `T1.Region` as the input argument to `pie`, because that variable contains text, not categorical data.

```
pie(T2.Region)
```



Detect Variable Types Without Specifying Names

Detect which table variables are datetime arrays. Then use the `datetime` function as an argument to the `convertvars` function to specify a time zone and display format.

Read power outage data into a table and display the first three rows.

```
T1 = readtable('outages.csv');
head(T1,3)
```

```
ans=3×6 table
       Region           OutageTime        Loss     Customers     RestorationTime
```

```
    _____________    _________________    ______    __________    _________________    __
    {'SouthWest'}    2002-02-01 12:18    458.98    1.8202e+06    2002-02-07 16:50    {
    {'SouthEast'}    2003-01-23 00:49    530.14    2.1204e+05                  NaT    {
    {'SouthEast'}    2003-02-07 21:15     289.4    1.4294e+05    2003-02-17 08:14    {
```

The datetime arrays in `T1` do not have their time zones set. Without specifying the names or locations of table variables, you can detect which variables are datetime arrays using a *function handle* to the `isdatetime` function. (A function handle is a variable that stores an association to a function. You can use a function handle to pass a function to another function. For example, specify `@isdatetime` to pass the handle to `convertvars`.) Then you can convert all datetime variables so that they have a time zone and a different display format. This technique is useful when converting many table variables that all have the same data type.

Call the `convertvars` function. To modify the time zone and format in place, specify an *anonymous function* that calls the `datetime` function with the `'TimeZone'` and `'Format'` name-value pair arguments. (An anonymous function is not stored in a program file. It can be useful for a function that requires only a brief definition. In this case, it also allows a call to `datetime` with multiple inputs, while passing `convertvars` a function that accepts only one input, as `convertvars` requires.) Display the first three rows, showing the change in format.

```
modifyTimeZoneAndFormat = @(x)(datetime(x,'TimeZone','UTC','Format','MMM dd, yyyy, HH:m
T2 = convertvars(T1,@isdatetime,modifyTimeZoneAndFormat);
head(T2,3)
```

```
ans=3×6 table
       Region              OutageTime            Loss     Customers         Restoratio
    _____________    ________________________    ______    __________    ______________

    {'SouthWest'}    Feb 01, 2002, 12:18 UTC    458.98    1.8202e+06    Feb 07, 2002, 1
    {'SouthEast'}    Jan 23, 2003, 00:49 UTC    530.14    2.1204e+05
    {'SouthEast'}    Feb 07, 2003, 21:15 UTC     289.4    1.4294e+05    Feb 17, 2003, 0
```

Input Arguments

T1 — Input table
table | timetable

Input table, specified as a table or timetable.

If `T1` is a timetable, then you cannot use `convertvars` to convert its row times, because the row times are not contained in a timetable variable. The row times are timetable metadata.

`vars` — Variables in input table or timetable

character vector | cell array of character vectors | string array | numeric array | logical array | function handle

Variables in the input table or timetable, specified as a character vector, cell array of character vectors, string array, numeric array, logical array, or function handle.

If `vars` is a function handle, then the function must accept one input argument, identify its data type, and return a logical scalar. For example, use the `isnumeric` function to detect which variables are numeric.

Example: `T2 = convertvars(T1,'Region','categorical')` converts the type of the variable `Region`.

Example: `T2 = convertvars(T1,[1,3:6],'string')` converts variables specified by position to string arrays.

Example: `T2 = convertvars(T1,@isnumeric,'int32')` converts all numeric variables to 32-bit integers.

`dataType` — Data type of converted variables

character vector | string scalar | function handle

Data type of the converted variables, specified as a character vector, string scalar, or function handle.

If `dataType` is a function handle, then the function must accept one input argument and convert it to another data type. For example, the `string` function converts an input argument to a string array.

The table shows the names of many common data types.

`'single'`	Single-precision number
`'double'`	Double-precision number
`'int8'`	Signed 8-bit integer

`'int16'`	Signed 16-bit integer
`'int32'`	Signed 32-bit integer
`'int64'`	Signed 64-bit integer
`'uint8'`	Unsigned 8-bit integer
`'uint16'`	Unsigned 16-bit integer
`'uint32'`	Unsigned 32-bit integer
`'uint64'`	Unsigned 64-bit integer
`'logical'`	Logical `1` (`true`) or `0` (`false`)
`'string'`	String array
`'cell'`	Cell array
`'cellstr'`	Cell array of character vectors
`'categorical'`	Categorical array
`'datetime'`	Datetime array
`'duration'`	Duration array
`'calendarDuration'`	Calendar duration array

If you specify `'char'` as a data type, then `convertvars` converts variables to character arrays. Best practice is to avoid creating table or timetable variables that are character arrays. Instead, consider converting variables to string arrays, categorical arrays, or cell arrays of character vectors.

Example: `T2 = convertvars(T1,'OutageTime','datetime')` converts the type of the variable `OutageTime`.

Example: `T2 = convertvars(T1,'Region',@categorical)` converts a variable using a function handle to the `categorical` function.

See Also

`addvars` | `mergevars` | `movevars` | `removevars` | `splitvars`

Topics

"Add, Delete, and Rearrange Table Variables"
"Modify Units, Descriptions, and Table Variable Names"
"Clean Messy and Missing Data in Tables"

“Create Function Handle”
“Pass Function to Another Function”

Introduced in R2018b

convhull

Convex hull

Syntax

```
k = convhull(P)
k = convhull(x,y)
k = convhull(x,y,z)
k = convhull( ___ ,'Simplify',tf)
[k,av] = convhull( ___ )
```

Description

`k = convhull(P)` computes the 2-D or 3-D convex hull of the points in matrix `P`.

`k = convhull(x,y)` computes the 2-D convex hull of the points in column vectors `x` and `y`.

`k = convhull(x,y,z)` computes the 3-D convex hull of the points in column vectors `x`, `y`, and `z`.

`k = convhull( ___ ,'Simplify',tf)` specifies whether to remove vertices that do not contribute to the area or volume of the convex hull. `tf` is `false` by default.

`[k,av] = convhull( ___ )` also computes the area or volume of the convex hull.

Examples

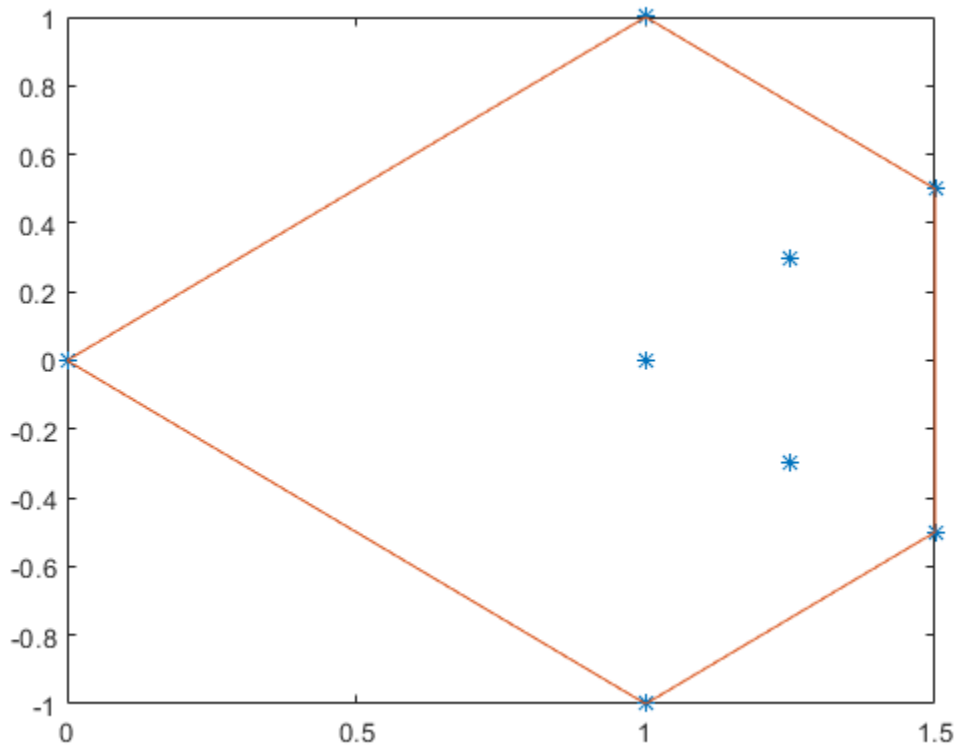
2-D Convex Hull

Create a matrix of 2-D points. Compute the convex hull and its area.

```
P = [0 0; 1 1; 1.5 0.5; 1.5 -0.5; 1.25 0.3; 1 0; 1.25 -0.3; 1 -1];
[k,av] = convhull(P);
```

Plot the original points and the convex hull.

```
plot(P(:,1),P(:,2),'*')
hold on
plot(P(k,1),P(k,2))
```



Display the area.

```
av
```

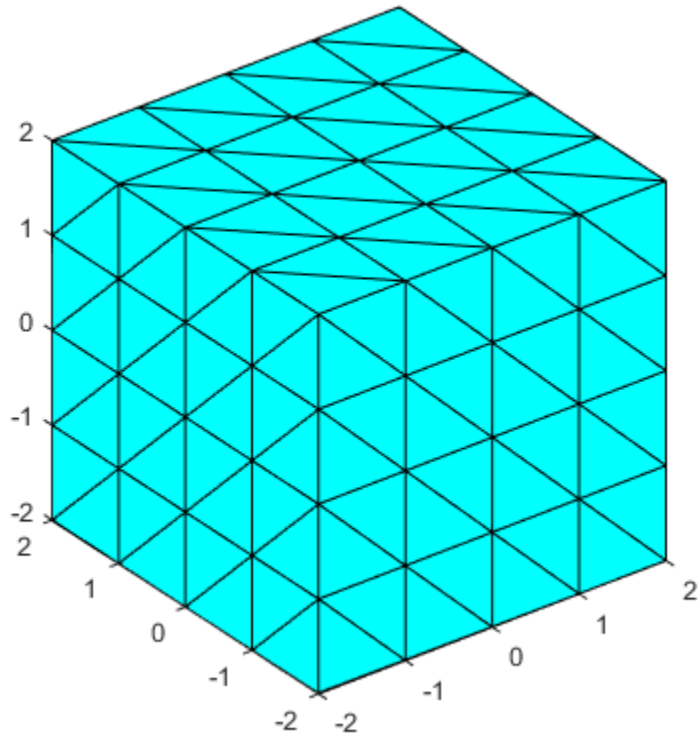
```
av = 1.7500
```

Simplify 3-D Convex Hull

Simplify a 3-D convex hull by removing points that do not affect its volume.

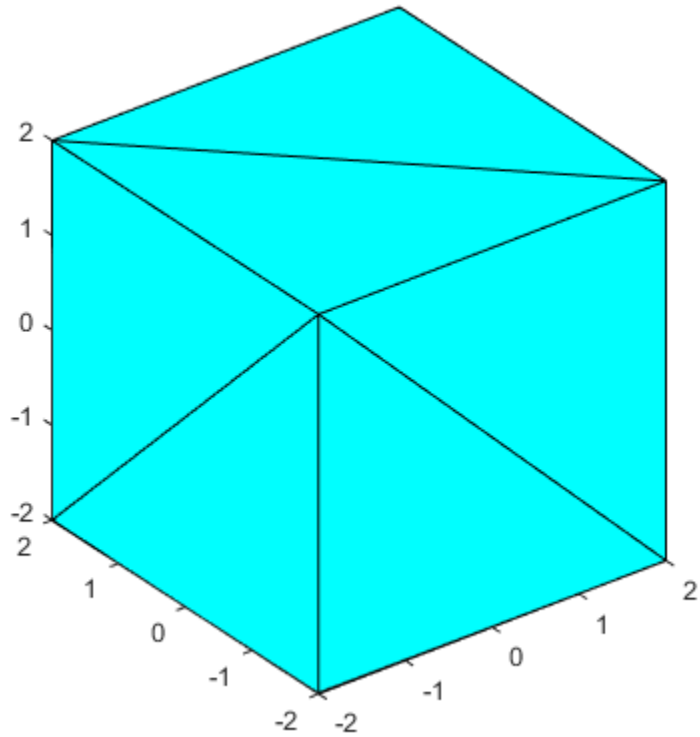
Create a set of 3-D points. Compute the convex hull and its volume. Plot the convex hull.

```
[x,y,z] = meshgrid(-2:1:2,-2:1:2,-2:1:2);
x = x(:);
y = y(:);
z = z(:);

[k1,av1] = convhull(x,y,z);

trisurf(k1,x,y,z,'FaceColor','cyan')
axis equal
```



Compute and plot a simplified version of the convex hull that contains fewer points, but preserves the volume.

```
[k2,av2] = convhull(x,y,z,'Simplify',true);

trisurf(k2,x,y,z,'FaceColor','cyan')
axis equal
```



Display the volumes of both convex hulls. The volumes are the same, but the simplied convex hull uses fewer points.

```
av1
```

```
av1 = 64.0000
```

```
av2
```

```
av2 = 64
```

Input Arguments

P — Points
matrix

Points, specified as a matrix whose columns are the *x*-coordinates, *y*-coordinates, and (in three dimensions) *z*-coordinates.

x — *x*-coordinates
column vector

x-coordinates, specified as a column vector.

y — *y*-coordinates
column vector

y-coordinates, specified as a column vector.

z — *z*-coordinates
column vector

z-coordinates, specified as a column vector.

`tf` — Simplify indicator
`false` or `0` (default) | `true` or `1`

Simplify indicator, specified as a numeric or logical `0` (`false`) or `1` (`true`).

Output Arguments

k — Indices
vector | matrix

Indices, returned as a vector or matrix.

- For 2-D points, `k` is a column vector containing the row indices of the input points that make up the convex hull, arranged counterclockwise.
- For 3-D points, `k` is a 3-column matrix representing a triangulation that makes up the convex hull. Each row represents a facet of the triangulation. The values represent the row indices of the input points.

av — Area or volume
scalar

Area or volume of the convex hull, returned as a scalar.

Tips

- To plot the output of `convhull` in two dimensions, use the `plot` function. To plot the output of `convhull` in three dimensions, use `trisurf` or `trimesh`.

See Also

`convhulln`

Introduced before R2006a

convhulln

N-D convex hull

Syntax

```
k = convhulln(P)
k = convhulln(P,opts)
[k,vol] = convhulln( ___ )
```

Description

`k = convhulln(P)` computes the N-D convex hull of the points in a matrix `P`.

`k = convhulln(P,opts)` also specifies the Qhull options used to compute the convex hull.

`[k,vol] = convhulln( ___ )` also returns the volume `vol` of the convex hull for either of the previous syntaxes.

Examples

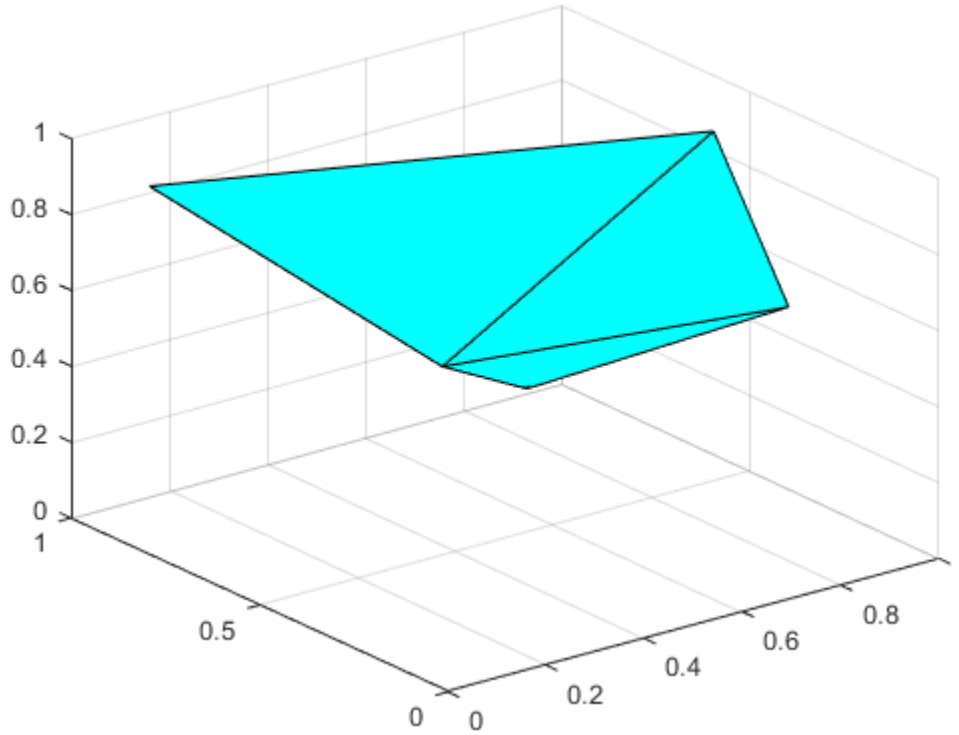
3-D Convex Hull

Create a matrix of 3-D points and compute the convex hull and its volume.

```
P = rand(10,3);
[k,vol] = convhulln(P);
```

Plot the convex hull.

```
trisurf(k,P(:,1),P(:,2),P(:,3),'FaceColor','cyan')
```



Display the volume of the convex hull.

```
vol
```

```
vol = 0.1857
```

Input Arguments

P — Points
matrix

Points, specified as a matrix whose columns contain the coordinates for the corresponding dimension. For example, to define a set of 2-D points, place the *x*-coordinates in the first column of `P` and the corresponding *y*-coordinates in the second column.

`opts` — Qhull options
cell array of character vectors

Qhull options, specified as a cell array of character vectors indicating which Qhull algorithms to use. For a list of options, see http://www.qhull.org/html/qh-optq.htm.

By default, `opts` is set to `{'Qt'}` for 2-, 3-, and 4-dimensional input. For 5-dimensional input and higher, `opts` is set to `{'Qt','Qx'}`.

Output Arguments

`k` — Indices
vector | matrix

Indices, returned as a vector or matrix.

- For 2-D points, `k` is a column vector containing the row indices of the input points that make up the convex hull, arranged counterclockwise.
- For 3-D points, `k` is a three-column matrix where each row represents a facet of a triangulation that makes up the convex hull. The values represent the row indices of the input points.
- For N-D points, `k` is an N-column matrix. The values represent the row indices of the input points.

`vol` — Volume
scalar

Volume of the convex hull (area for two dimensions), returned as a scalar.

Tips

- Use the `plot` function to plot the output of `convhulln` in two dimensions. Use `trisurf` or `trimesh` to plot the output of `convhulln` in three dimensions.

- For better algorithm efficiency with 2-D and 3-D input, consider using the `convhull` function or creating a `triangulation` or `delaunayTriangulation` object and using the `convexHull` object function.

Algorithms

`convhulln` is based on Qhull [1]. For information, see http://www.qhull.org/. For copyright information, see http://www.qhull.org/COPYING.txt.

References

[1] Barber, C.B., D.P. Dobkin, and H.T. Huhdanpaa. "The Quickhull Algorithm for Convex Hulls." *ACM Transactions on Mathematical Software*, Vol. 22, No. 4, Dec. 1996, p 469–483.

See Also

`convexHull` | `convhull` | `delaunayTriangulation` | `triangulation`

Introduced before R2006a

convn

N-D convolution

Syntax

```
C = convn(A,B)
C = convn(A,B,shape)
```

Description

`C = convn(A,B)` returns the N-dimensional convolution on page 1-2460 of arrays `A` and `B`.

`C = convn(A,B,shape)` returns a subsection of the convolution according to `shape`. For example, `C = convn(A,B,'same')` returns the central part of the convolution, which is the same size as `A`.

Examples

3-D Convolution

You can control the size of the output of the `convn` function. For example, the `'same'` option trims the outer part of the convolution and returns only the central part, which is the same size as the input.

Convolve a random 2-by-3-by-2 array `A` with a 2-by-2-by-2 kernel `B`. The result is a 3-by-4-by-3 array, which is `size(A) + size(B) - 1`.

```
A = rand(2,3,2);
B = 0.25*ones(2,2,2);
C = convn(A,B)
```

```
C =
C(:,:,1) =
```

```
    0.2037    0.2354    0.1898    0.1581
    0.4301    0.6902    0.4426    0.1825
    0.2264    0.4548    0.2527    0.0244


C(:,:,2) =

    0.2733    0.5444    0.4686    0.1975
    0.6365    1.3772    1.2052    0.4645
    0.3632    0.8327    0.7366    0.2670


C(:,:,3) =

    0.0696    0.3090    0.2788    0.0394
    0.2063    0.6869    0.7627    0.2821
    0.1367    0.3779    0.4839    0.2426
```

```
sizeC = size(A) + size(B) - 1
```

```
sizeC = 1×3

     3     4     3
```

Return the central part of the convolution, which is the same size as A.

```
C = convn(A,B,'same')
```

```
C =
C(:,:,1) =

    1.3772    1.2052    0.4645
    0.8327    0.7366    0.2670


C(:,:,2) =

    0.6869    0.7627    0.2821
    0.3779    0.4839    0.2426
```

Input Arguments

A — Input array

vector | matrix | multidimensional array

Input array, specified as vector, a matrix, or a multidimensional array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

B — Second input array

vector | matrix | multidimensional array

Second input array, specified as a vector, a matrix, or a multidimensional array to convolve with A. The array B does not have to be the same size as A.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

shape — Subsection of convolution

`'full'` (default) | `'same'` | `'valid'`

Subsection of the convolution, specified as one of these values:

- `'full'` — Return the full N-D convolution.
- `'same'` — Return the central part of the convolution, which is the same size as A.
- `'valid'` — Return only parts of the convolution that are computed without zero-padded edges.

Output Arguments

C — N-D convolution

vector | matrix | multidimensional array

N-D convolution, returned as a vector, a matrix, or a multidimensional array. If A and B have the same number of dimensions, the full convolution `C = convn(A,B)` has size `size(A)+size(B)-1`.

When one or both of `A` and `B` are of type `single`, then the output is of type `single`. Otherwise, `convn` converts inputs to type `double` and returns type `double`.

Data Types: `double` | `single`

More About

N-D Convolution

For discrete, N-dimensional variables A and B, this equation defines the convolution of A and B:

$$C(j_1, j_2, \ldots, j_N) = \sum_{k_1}\sum_{k_2}\ldots\sum_{k_N} A(k_1, k_2, \ldots, k_N)B(j_1 - k_1, j_2 - k_2, \ldots, j_N - k_N)$$

Each k_i runs over all values that lead to legal subscripts of A and B.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- If `shape` is `'full'` (default), then the inputs `A` and `B` must not be empty and only one them can be a tall array.
- If `shape` is `'same'` or `'valid'`, then `B` cannot be a tall array.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

conv | conv2

Introduced before R2006a

copper

Copper colormap array

Syntax

```
c = copper
c = copper(m)
```

Description

`c = copper` returns the copper colormap as a three-column array with the same number of rows as the colormap for the current figure. If no figure exists, then the number of rows is equal to the default length of 256. Each row in the array contains the red, green, and blue intensities for a specific color. The intensities are in the range [0,1], and the color scheme looks like this image.

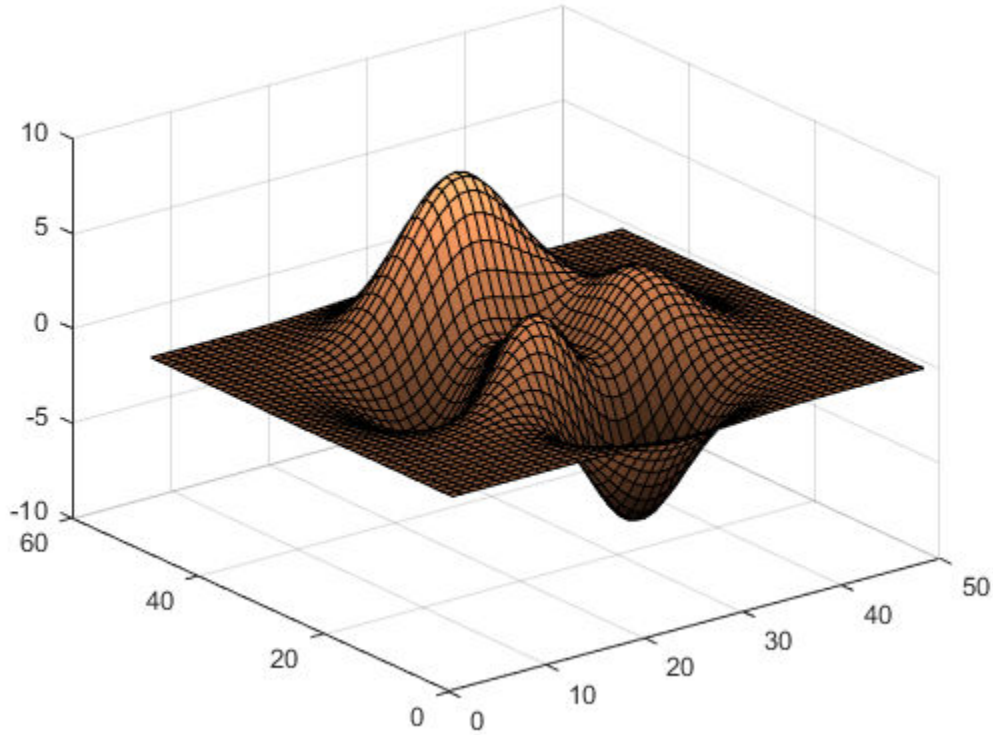
`c = copper(m)` returns the colormap with `m` colors.

Examples

Reverse the Colormap

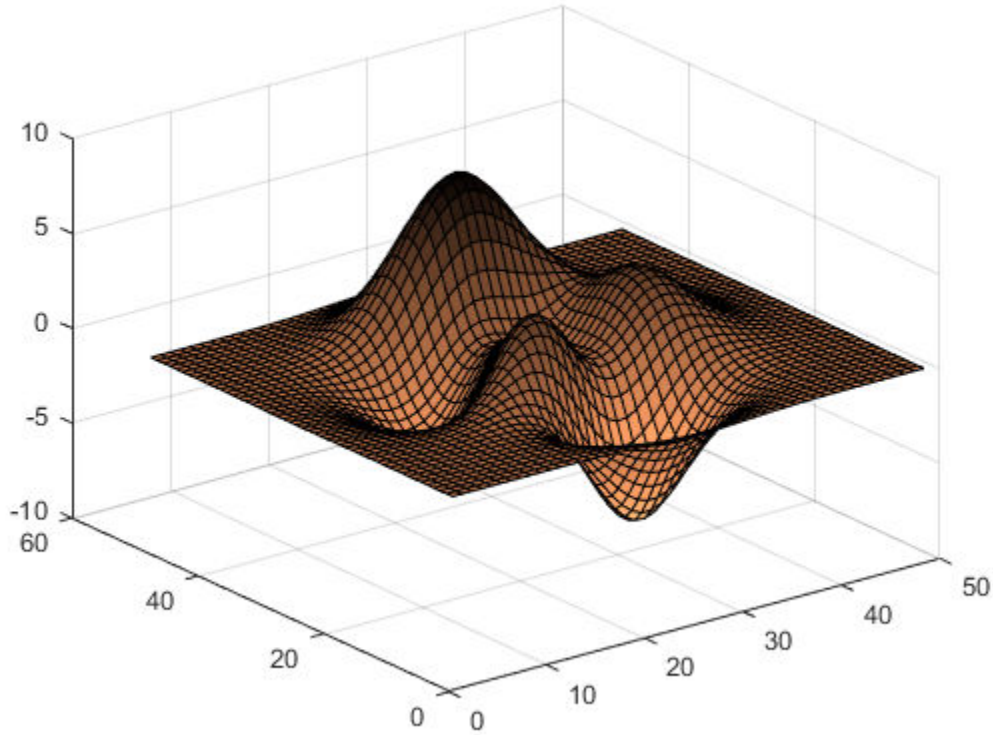
Plot a surface and assign the copper colormap.

```
surf(peaks);
colormap('copper');
```



Get the copper colormap array and reverse the order. Then apply the modified colormap to the surface.

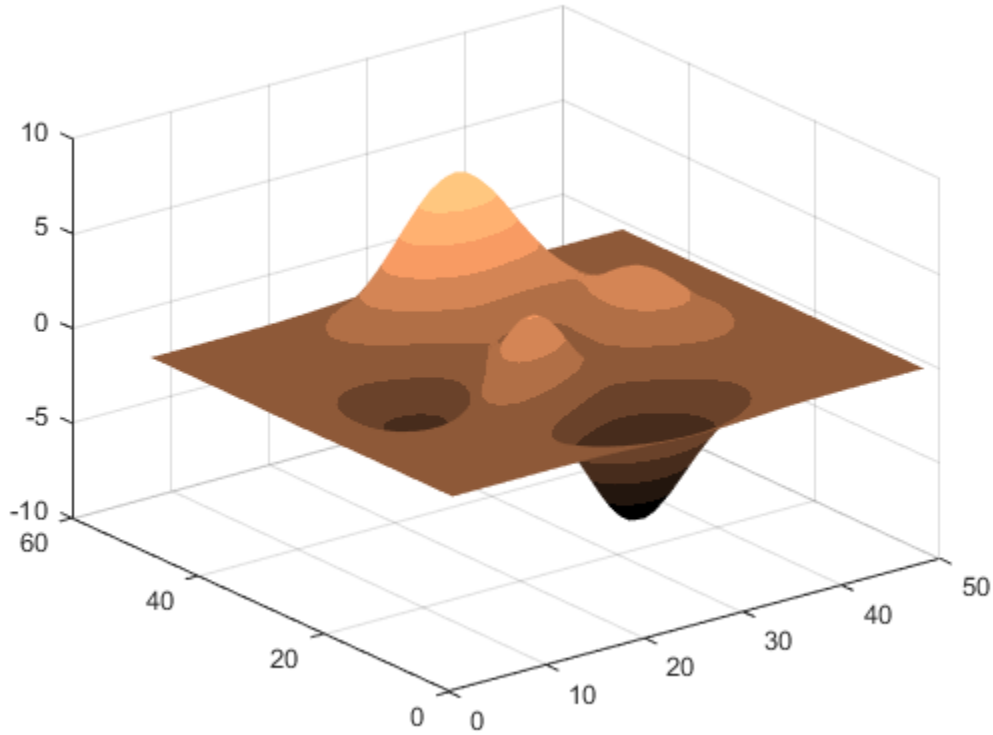
```
c = copper;
c = flipud(c);
colormap(c);
```



Downsample the Copper Colormap

Get a downsampled version of the copper colormap containing only ten colors. Then display the contours of the peaks function by applying the colormap and interpolated shading.

```
c = copper(10);
surf(peaks);
colormap(c);
shading interp;
```



Input Arguments

`m` — Number of colors

256 (default) | scalar integer value

Number of colors, specified as a scalar integer value. The default value of `m` is equal to the length of the colormap for the current figure. If no figure exists, the default value is 256.

Data Types: `single` | `double`

Compatibility Considerations

Colormaps Have 256 Colors by Default

Behavior changed in R2019b

Starting in R2019b, colormaps have 256 colors by default.

In R2019a and previous releases, the default size is 64. If you have code that depends on a colormap having 64 colors, specify the number of colors when you set the colormap for the figure, axes, or chart. For example, `colormap(copper(64))` sets the figure's colormap to the 64-color `copper` colormap.

Alternatively, you can change the default colormap for all figures within your MATLAB session:

```
set(groot,'defaultFigureColormap',copper(64))
```

See Also

`colormap`

Topics

"Change Color Scheme Using a Colormap"

Introduced before R2006a

matlab.mixin.CustomDisplay class

Package: `matlab.mixin`

Interface for customizing object display

Description

This class provides an interface for customizing the way MATLAB displays objects. Derive your class from `matlab.mixin.CustomDisplay` to add the custom display functionality to your class.

For customizing object display, `matlab.mixin.CustomDisplay` defines a number of protected methods that you can override in your subclass. By overriding specific methods, you can customize specific aspects of the object display.

`matlab.mixin.CustomDisplay` also implements three public, sealed methods. `disp` and `display` provide a simple object display. The `details` method provides a standard formal display of object information.

Note You cannot use `matlab.mixin.CustomDisplay` to derive a custom display for enumeration classes. For an alternative approach, see “Overloading the disp Function”

Methods

convertDimensionsToString	Return array dimensions as text
displayEmptyObject	Display for empty object arrays
displayNonScalarObject	Display format for non-scalar objects
displayPropertyGroups	Display titles and property groups as defined
displayScalarHandleToDeletedObject	Display format for deleted scalar handles
displayScalarObject	Display format for scalar objects
getClassNameForHeader	Return class name for display
getDeletedHandleText	Returns text for handle to deleted object display
getDetailedFooter	Returns default detailed footer for object display
getDetailedHeader	Returns default detailed header for object display
getFooter	Build and return display footer text
getHandleText	Return text `'handle'` with link to documentation
getHeader	Build and return display header text
getPropertyGroups	Construct array of property groups
getSimpleHeader	Return simple header for object display

Attributes

`Abstract`	This class defines an interface that subclasses inherit. You cannot instantiate this class.
`HandleCompatible`	You can use this class to derive both handle and value classes.

See Also

`matlab.mixin.util.PropertyGroup`

Topics

“Custom Display Interface”

Introduced in R2013b

matlab.mixin.CustomDisplay.convertDimensionsToString

Class: `matlab.mixin.CustomDisplay`
Package: `matlab.mixin`

Return array dimensions as text

Syntax

```
dimstr = matlab.mixin.CustomDisplay.convertDimensionsToString(obj)
```

Description

`dimstr = matlab.mixin.CustomDisplay.convertDimensionsToString(obj)` converts a size vector into a properly formatted text of dimensions for the nonscalar header.

Input Arguments

`obj`

MATLAB object

Output Arguments

`dimstr`

Text representing the object's dimensions as determined by calling `size`.

Attributes

`Static`	`true`
`Access`	`Protected`
`Sealed`	`true`
`Hidden`	`true`

To learn about attributes of methods, see Method Attributes.

See Also

`getHeader`

Topics

"Custom Display Interface"

displayEmptyObject

Class: `matlab.mixin.CustomDisplay`
Package: `matlab.mixin`

Display for empty object arrays

Syntax

```
displayEmptyObject(obj)
```

Description

`displayEmptyObject(obj)` is called by `disp` when the object, `obj`, is empty. An object array is empty if one or more of its dimensions are zero. An empty object array is never scalar.

The default display of an empty object consists of a header and a list of property names. The header consists of the object's dimensions and the properties are shown in the order defined in the class definition. `displayEmptyObject` shows only those properties with public `GetAccess` and `Hidden` set to `false`.

Override this method to customize the appearance of an empty object array.

Input Arguments

`obj`

Object of a class derived from `matlab.mixin.CustomDisplay`

Default:

Attributes

`Access`	`protected`

To learn about attributes of methods, see Method Attributes.

See Also

`matlab.mixin.CustomDisplay.displayNonScalarObject` | `matlab.mixin.CustomDisplay.displayScalarHandleToDeletedObject` | `matlab.mixin.CustomDisplay.displayScalarObject`

Topics

"Custom Display Interface"

displayNonScalarObject

Class: `matlab.mixin.CustomDisplay`
Package: `matlab.mixin`

Display format for non-scalar objects

Syntax

```
displayNonScalarObject(obj)
```

Description

`displayNonScalarObject(obj)` is called by the `disp` method when the object, `obj`, is nonscalar (`prod(size(obj)) > 1`)

The default display of a nonscalar object array consists of a header and a list of property names. The header consists of the object's dimensions and the properties are shown in the order defined in the class definition. `displayNonScalarObject` shows only those properties with public `GetAccess` and `Hidden` set to `false`.

Override this method to customize the display a nonscalar object array.

Input Arguments

`obj`

Object array of a class derived from `matlab.mixin.CustomDisplay`

Default:

Attributes

`Access`	`protected`

To learn about attributes of methods, see Method Attributes.

See Also

`matlab.mixin.CustomDisplay.displayEmptyObject` | `matlab.mixin.CustomDisplay.displayScalarHandleToDeletedObject` | `matlab.mixin.CustomDisplay.displayScalarObject`

Topics

"Custom Display Interface"

matlab.mixin.CustomDisplay.displayPropertyGroups

Class: `matlab.mixin.CustomDisplay`
Package: `matlab.mixin`

Display titles and property groups as defined

Syntax

```
matlab.mixin.CustomDisplay.displayPropertyGroups(obj,
propertyGroupArray)
```

Description

`matlab.mixin.CustomDisplay.displayPropertyGroups(obj, propertyGroupArray)` displays titles and custom property lists as defined by the property groups.

Input Arguments

`obj`

MATLAB object

Default:

`propertyGroupArray`

Array of `matlab.mixin.util.PropertyGroup` objects.

Default:

Attributes

`Static`	`true`
`Access`	`Protected`
`Sealed`	`true`
`Hidden`	`true`

To learn about attributes of methods, see Method Attributes.

See Also

`PropertyGroup`

Topics

"Custom Display Interface"

displayScalarHandleToDeletedObject

Class: `matlab.mixin.CustomDisplay`
Package: `matlab.mixin`

Display format for deleted scalar handles

Syntax

```
displayScalarHandleToDeletedObject(obj)
```

Description

`displayScalarHandleToDeletedObject(obj)` is called by the `disp` method when `obj` is:

- An instance of a handle class
- Scalar
- A handle to a deleted object

That is, the following expression is `true`.

```
isa(obj,'handle') && isscalar(obj) && ~isvalid(obj)
```

Override this method to customize the appearance of your object's display when it is deleted.

Input Arguments

`obj`

Object of a class derived from `matlab.mixin.CustomDisplay`

Default:

Attributes

`Access`	`protected`

To learn about attributes of methods, see Method Attributes.

See Also

`matlab.mixin.CustomDisplay.displayEmptyObject` | `matlab.mixin.CustomDisplay.displayNonScalarObject` | `matlab.mixin.CustomDisplay.displayScalarObject`

Topics

"Custom Display Interface"

displayScalarObject

Class: `matlab.mixin.CustomDisplay`
Package: `matlab.mixin`

Display format for scalar objects

Syntax

```
displayScalarObject(obj)
```

Description

`displayScalarObject(obj)` is called by the `disp` method when the object, `obj`, is scalar (`prod(size(obj)) == 1`).

The default display of a scalar object consists of a header and a list of properties and their values. Properties are shown in the order they are defined in the class definition. `displayScalarObject` shows only those properties with public `GetAccess` and `Hidden` set to `false`.

Override this method to customize the display of a scalar object.

Input Arguments

`obj`

Object of a class derived from `matlab.mixin.CustomDisplay`

Default:

Attributes

`Access`	`protected`

To learn about attributes of methods, see Method Attributes.

See Also

`matlab.mixin.CustomDisplay.displayEmptyObject` | `matlab.mixin.CustomDisplay.displayNonScalarObject` | `matlab.mixin.CustomDisplay.displayScalarHandleToDeletedObject`

Topics

"Custom Display Interface"

matlab.mixin.CustomDisplay.getClassNameForHeader

Class: `matlab.mixin.CustomDisplay`
Package: `matlab.mixin`

Return class name for display

Syntax

```
name = matlab.mixin.CustomDisplay.getClassNameForHeader(obj)
```

Description

`name = matlab.mixin.CustomDisplay.getClassNameForHeader(obj)` returns the class name of `obj`. If the display supports hypertext links, the text is linked to the help for the class of `obj`.

Use this method when building a custom display that includes the class name, but differs from the default header.

Input Arguments

`obj`

MATLAB object

Default:

Output Arguments

name

The simple class name, linked to the help for the class if the display supports hypertext links

Attributes

Static	true
Access	Protected
Sealed	true
Hidden	true

To learn about attributes of methods, see Method Attributes.

See Also

getHeader

Topics

"Custom Display Interface"

matlab.mixin.CustomDisplay.getDeletedHandleText

Class: `matlab.mixin.CustomDisplay`
Package: `matlab.mixin`

Returns text for handle to deleted object display

Syntax

```
handleText = matlab.mixin.CustomDisplay.getDeletedHandleText
```

Description

`handleText = matlab.mixin.CustomDisplay.getDeletedHandleText` returns the text:

```
'handle to deleted'
```

The text is linked to the documentation on deleted handle objects.

Output Arguments

`handleText`

Text `'handle to deleted'`, linked if the display supports hypertext links

Attributes

`Static`	`true`
`Access`	`Protected`
`Sealed`	`true`

`Hidden`	`true`

To learn about attributes of methods, see Method Attributes.

See Also

`getHeader`

Topics

"Custom Display Interface"

matlab.mixin.CustomDisplay.getDetailedFooter

Class: `matlab.mixin.CustomDisplay`
Package: `matlab.mixin`

Returns default detailed footer for object display

Syntax

```
headerText = matlab.mixin.CustomDisplay.getDetailedFooter(obj)
```

Description

`headerText = matlab.mixin.CustomDisplay.getDetailedFooter(obj)` returns the text containing:

```
Methods, Events, Superclass
```

Each link executes the respective command on `obj`.

Input Arguments

`obj`

MATLAB object

Output Arguments

`headerText`

Text containing the linked phrase `'Methods, Events, Superclasses'` or an empty character vector if the display does not support hypertext links

Attributes

`Static`	`true`
`Access`	`Protected`
`Sealed`	`true`
`Hidden`	`true`

To learn about attributes of methods, see Method Attributes.

See Also

`events` | `getFooter` | `methods` | `superclasses`

Topics

"Custom Display Interface"

matlab.mixin.CustomDisplay.getDetailedHeader

Class: `matlab.mixin.CustomDisplay`
Package: `matlab.mixin`

Returns default detailed header for object display

Syntax

```
header = matlab.mixin.CustomDisplay.getDetailedHeader(obj)
```

Description

`header = matlab.mixin.CustomDisplay.getDetailedHeader(obj)` returns text containing:

- Linked class name of `obj`
- Link to handle documentation if `obj` is a handle class
- The text `'with properties:'`

Input Arguments

`obj`

MATLAB object

Output Arguments

`header`

Text containing the full detailed header, with properly inserted links if the display supports hypertext linking

Attributes

`Static`	`true`
`Access`	`Protected`
`Sealed`	`true`
`Hidden`	`true`

To learn about attributes of methods, see Method Attributes.

See Also

`handle`

Topics

"Custom Display Interface"

getFooter

Class: `matlab.mixin.CustomDisplay`
Package: `matlab.mixin`

Build and return display footer text

Syntax

```
s = getFooter(obj)
```

Description

`s = getFooter(obj)` returns the text used as the footer when displaying `obj`. This method is called once for the entire object array.

Override this method to create a custom footer. The overriding implementation must support all states of the object, including scalar, nonscalar, empty, and deleted (if `obj` is an instance of a handle class).

Input Arguments

`obj`

Object array of a class derived from `matlab.mixin.CustomDisplay`

Output Arguments

`s`

Footer text, returned as a `char` vector.

The default implementation returns an empty `char` vector

Attributes

`Access`	`protected`

To learn about attributes of methods, see Method Attributes.

See Also

`matlab.mixin.CustomDisplay.getHeader` | `matlab.mixin.CustomDisplay.getPropertyGroups`

Topics

"Custom Display Interface"

matlab.mixin.CustomDisplay.getHandleText

Class: `matlab.mixin.CustomDisplay`
Package: `matlab.mixin`

Return text `'handle'` with link to documentation

Syntax

```
handleText = matlab.mixin.CustomDisplay.getHandleText
```

Description

`handleText = matlab.mixin.CustomDisplay.getHandleText` returns the text `'handle'`. If the display supports hypertext linking, the text is linked to documentation describing handle classes.

Output Arguments

`handleText`

Text `handle`, linked to the handle documentation.

Attributes

`Static`	`true`
`Access`	`Protected`
`Sealed`	`true`
`Hidden`	`true`

To learn about attributes of methods, see Method Attributes.

See Also

`getHeader`

Topics

"Custom Display Interface"

getHeader

Class: `matlab.mixin.CustomDisplay`
Package: `matlab.mixin`

Build and return display header text

Syntax

```
s = getHeader(obj)
```

Description

`s = getHeader(obj)` returns the text used as the header when displaying `obj`. This method is called once for the entire object array.

Override this method to create a custom header. The overriding implementation must support all states of the object, including scalar, nonscalar, empty, and deleted (if `obj` is an instance of a handle class).

Input Arguments

`obj`

Object array of a class derived from `matlab.mixin.CustomDisplay`

Output Arguments

`s`

Header string, returned as a `char` array

The default implementation returns the following:

- If `obj` is scalar, returns *`classname`*, which is the simple name of the class (the nonpackage-qualified name).
- If `obj` is nonscalar, returns *`classname`* and dimensions.
- If `obj` is empty, returns an empty `char`.
- If `obj` is a deleted handle, returns the string `deleted` *`classname`* `handle`

`classname` is linked to MATLAB documentation for the class. Selecting the link displays the `helpPopup` window.

If you override this method, you might need to terminate `s` with a newline (`\n`) character.

Examples

Append Text to Default Header

Append the text, `'with Customized Display'`, to the header text.

Write a `getHeader` method.

```
methods (Access = protected)
   function header = getHeader(obj)
      if ~isscalar(obj)
         header = getHeader@matlab.mixin.CustomDisplay(obj);
      else
         headerStr = matlab.mixin.CustomDisplay.getClassNameForHeader(obj);
         headerStr = [headerStr,' with Customized Display'];
         header = sprintf('%s\n',headerStr);
      end
   end
end
```

Add `getHeader` method to class definition.

Attributes

`Access`	`protected`

To learn about attributes of methods, see Method Attributes.

See Also

`matlab.mixin.CustomDisplay.getFooter` | `matlab.mixin.CustomDisplay.getPropertyGroups`

Topics

"Custom Display Interface"

getPropertyGroups

Class: `matlab.mixin.CustomDisplay`
Package: `matlab.mixin`

Construct array of property groups

Syntax

```
groups = getPropertyGroups(obj)
```

Description

`groups = getPropertyGroups(obj)` returns an array of `matlab.mixin.util.PropertyGroup` objects. MATLAB displays property groups separated by blank spaces.

Each default display state handler method calls this method once. The default implementation returns the properties in one group. These properties must have public `GetAccess` and not be defined as `Hidden`. If the object is scalar, MATLAB includes dynamic properties.

Override this method to construct one or more customized groups of properties to display.

Each group object array has the following fields:

- `Title` — Text used as the header for the property group or an empty string if no title is used.
- `PropertyList` — The property list can be either:
  - A 1-by-1 `struct` of property name-property value pairs
  - A cell array of property names.

Use the `struct` of name-value pairs if the object is scalar and you want to assign custom property values. Otherwise, use a cell array of property names. If the object is scalar MATLAB adds the property values retrieved from the object.

Input Arguments

obj

Object array of a class derived from `matlab.mixin.CustomDisplay`

Default:

Output Arguments

groups

1xN array of `matlab.mixin.util.PropertyGroup` objects, where N is the number of groups

Examples

Custom Property Group

Customize the values returned by some properties.

Write a `getPropertyGroups` method.

```
methods (Access = protected)
   function propgrp = getPropertyGroups(obj)
      if ~isscalar(obj)
         propgrp = getPropertyGroups@matlab.mixin.CustomDisplay(obj);
      else
         pd(1:length(obj.Password)) = '*';
         propList = struct('Department',obj.Department,...
            'JobTitle',obj.JobTitle,...
            'Name',obj.Name,...
            'Salary','Not available',...
            'Password',pd);
         propgrp = matlab.mixin.util.PropertyGroup(propList);
      end
   end
end
```

Add function to class definition.

Attributes

`Access`	`protected`

To learn about attributes of methods, see Method Attributes.

See Also

`matlab.mixin.CustomDisplay.getFooter` | `matlab.mixin.CustomDisplay.getHeader` | `matlab.mixin.util.PropertyGroup`

Topics

"Custom Display Interface"
"Customize Property Display"

matlab.mixin.CustomDisplay.getSimpleHeader

Class: `matlab.mixin.CustomDisplay`
Package: `matlab.mixin`

Return simple header for object display

Syntax

```
header = matlab.mixin.CustomDisplay.getSimpleHeader(obj)
```

Description

`header = matlab.mixin.CustomDisplay.getSimpleHeader(obj)` returns the default simple header for `obj`.

Input Arguments

`obj`

MATLAB object.

Default:

Output Arguments

`header`

Text containing the linked class name and the phrase `'with properties'`

Attributes

`Static`	`true`
`Access`	`Protected`
`Sealed`	`true`
`Hidden`	`true`

To learn about attributes of methods, see Method Attributes.

See Also

`matlab.mixin.CustomDisplay.getHeader`

Topics

"Custom Display Interface"

matlab.mixin.Copyable class

Package: `matlab.mixin`

Superclass providing copy functionality for handle objects

Description

The `matlab.mixin.Copyable` class is an abstract handle class that provides a `copy` method for copying handle objects. The `copy` method makes a shallow copy of the object (that is, it shallow-copies all nondependent properties from the source object to the destination object). MATLAB does not call `copy` recursively on any handles contained in property values.

Subclass `matlab.mixin.Copyable` to define handle classes that inherit a `copy` method. The copy method copies data without calling the class constructor or property set functions. It therefore produces no side effects.

Subclasses can customize copy behavior by deriving from `matlab.mixin.Copyable` and overriding the `copyElement` method. For more information, see the example "Customizing Subclass Copy Behavior" on page 1-2503.

The `matlab.mixin.Copyable` class is a `handle` class.

Class Attributes

`Abstract`	`true`
`ConstructOnLoad`	`true`
`HandleCompatible`	`true`

For information on class attributes, see "Class Attributes".

Methods

Public Methods

copy Copy array of handle objects

Protected Methods

copyElement Copy scalar MATLAB object

Examples

Add Copy Method

Add a copy method to your handle class by subclassing `matlab.mixin.Copyable`.

```
classdef MyClass < matlab.mixin.Copyable
   properties
      Prop
   end
end
```

Create an object.

```
a = MyClass;
```

Create a copy of the object.

```
b = copy(a);
```

For more information, see “Implement Copy for Handle Classes”.

Customizing Subclass Copy Behavior

The `copy` method provides the public, non-overrideable interface to copy behavior. This method takes an array of objects as input and returns an array of the same dimensions.

`copyElement` is a protected method that the `copy` method uses to perform the copy operation on each object in the input array. You can override `copyElement` in your subclass to customize the behavior of the inherited `copy` method.

Use the property `NonCopyable` attribute to control if the copy operation copies specific property values.

Implement Selective Deep Copy

This example overrides the `copyElement` method in a subclass of `matlab.mixin.Copyable` to implement a deep copy of a specific class of `handle` objects.

Consider the following classes:

- `ContainsHandles` — subclass of `matlab.mixin.Copyable` that contains `handle` objects in two properties
- `DeepCp` — subclass of `matlab.mixin.Copyable`
- `ShallowCp` — subclass of `handle`

Here are the simplified class definitions.

```
classdef ContainsHandles < matlab.mixin.Copyable
   properties
      Prop1
      Prop2
      DeepObj     % Contains a DeepCp object
      ShallowObj  % Contains a ShallowCp object
   end
   methods
      function obj = ContainsHandles(val1,val2,deepobj,shallowobj)
         if nargin > 0
            obj.Prop1 = val1;
            obj.Prop2 = val2;
            obj.DeepObj = deepobj;
            obj.ShallowObj = shallowobj;
         end
      end
   end
   methods(Access = protected)
      % Override copyElement method:
      function cpObj = copyElement(obj)
         % Make a shallow copy of all four properties
         cpObj = copyElement@matlab.mixin.Copyable(obj);
         % Make a deep copy of the DeepCp object
         cpObj.DeepObj = copy(obj.DeepObj);
      end
```

```
   end
end
```

The `DeepCp` class derives from `matlab.mixin.Copyable`.

```
classdef DeepCp < matlab.mixin.Copyable
   properties
      DpProp
   end
   methods
      function obj = DeepCp(val)
         ...
      end
   end
end
```

The `handle` class `ShallowCp` does not derive from `matlab.mixin.Copyable` and, therefore, has no `copy` method.

```
classdef ShallowCp < handle
   properties
      ShProp
   end
   methods
      function obj = ShallowCp(val)
         ...
      end
   end
end
```

Create a `ContainsHandles` object, which contains the two handle objects in its `DpProp` and `ShProp` properties.

```
sc = ShallowCp(7);
dc = DeepCp(7);
a = ContainsHandles(4,5,dc,sc);
a.DeepObj
```

```
ans =

  DeepCp with properties:

    DpProp: 7
```

```
a.ShallowObj.ShProp
```

```
ans =

  ShallowCp with properties:

    ShProp: 7
```

Make a copy of the `ContainsHandles` object.

```
b = copy(a);
```

The returned copy `b` contains a shallow copy of object `sc`, and a deep copy of object `dc`. That is, the `dc` object passed to `ContainsHandles` constructor is now a new, independent object as a result of the copy operation. You can now change the `dc` object without affecting the copy. This is not the case for the shallow-copied object, `sc`.

Change the property values of the handle objects.

```
sc.ShProp = 5;
dc.DpProp = 5;
```

Note that the object that is deep-copied is not affected.

```
b.DeepObj
```

```
ans =

  DeepCp with properties:

    DpProp: 7
```

The shallow-copied object still references the same data.

```
b.ShallowObj
```

```
ans =

  ShallowCp with properties:

    ShProp: 5
```

Override Copy Behavior in Hierarchies

The `copyElement` method in a superclass cannot access the private data in a subclass.

If you override `copyElement` in a subclass of `matlab.mixin.Copyable`, and then use this subclass as a superclass, you need to override `copyElement` in all subclasses that

contain private properties. The override of `copyElement` in subclasses should call the `copyElement` in the respective superclass, as in the previous example.

The following simplified code demonstrates this approach.

```
classdef SuperClass < matlab.mixin.Copyable
   properties(Access = private)
      super_prop
   end
   methods
        ...

      function cpObj = copyElement(obj)
            ...
         cpObj = copyElement@matlab.mixin.Copyable(obj);
            ...
      end
   end
end



classdef SubClass1 < SuperClass
   properties(Access=private)
      sub_prop1
   end
   methods
      function cpObj = copyElement(obj)
         % Copy super_prop
         cpObj = copyElement@SuperClass(obj);
         % Copy sub_prop1 in subclass
         % Assignment can introduce side effects
         cpObj.sub_prop1 = obj.sub_prop1;
      end
   end
end
```

The override of `copyElement` in `SubClass1` copies the private subclass property because the superclass cannot access private data in the subclass.

Note The assignment of `sub_prop1` in the override of `copyElement` in `SubClass1` calls the property set method, if one exists, possibly introducing side effects to the copy operation.

Copy Behaviors for Specific Inputs

Consider a call to the `matlab.mixin.Copyable` `copy` method of this form:

```
B = copy(A);
```

This call to `copy` produces the results described for each of the following conditions:

- `A` has dynamic properties — `copy` does not copy dynamic properties. You can implement dynamic-property copying in the subclass if needed.
- `A` has no non-dependent properties — `copy` creates a new object with no property values without calling the class constructor to avoid introducing side effects.
- `A` contains deleted handles — `copy` creates deleted handles of the same class in the output array.
- `A` has attached listeners — `copy` does not copy listeners.
- `A` contains objects of enumeration classes — Enumeration classes cannot subclass `matlab.mixin.Copyable`.
- A `delete` method calls `copy` — `copy` creates a legitimate copy, obeying all the behaviors that apply in any other usage.

Note You cannot derive an enumeration class from `matlab.mixin.Copyable` because the instances you can create is limited to the ones defined inside the enumeration block. See “Define Enumeration Classes” for more information about enumeration classes.

More About

Making a Deep Copy

Copy each property value and assign it to the new (copied) property. Recursively copy property values that reference handle objects to copy all of the underlying data.

Making a Shallow Copy

Copy each property value and assign it to the new (copied) property. If a property value is a handle, copy the handle but not the underlying data.

See Also

Topics

"Implement Copy for Handle Classes"

Introduced in R2011a

copy

Class: `matlab.mixin.Copyable`
Package: `matlab.mixin`

Copy array of handle objects

Syntax

```
B = copy(A)
```

Description

`B = copy(A)` copies each element in the array of handles `A` to the new array of handles `B`.

The `copy` method performs a copy according to the following rules:

- The `copy` method does not copy `Dependent` properties
- MATLAB does not call the `copy` method recursively on any handles contained in property values
- MATLAB does not call the class constructor or property set methods during the copy operation.
- `B` has the same number of elements and same size as `A`.
- `B` is the same class as `A`.
- If `A` is empty, `B` is also empty.
- If `A` is heterogeneous, `B` is also heterogeneous.
- If `A` contains deleted handle objects, copy creates deleted handles of the same class in `B`.
- Dynamic properties and listeners associated with objects in `A` are not copied to objects in `B`.
- You can call `copy` inside your class `delete` method.

Input Arguments

A

Handle object array

Default:

Output Arguments

B

Handle object array containing copies of the objects in A.

Attributes

Sealed	true

To learn about attributes of methods, see Method Attributes.

See Also

handle | matlab.mixin.Copyable

Topics

"Implement Copy for Handle Classes"

Introduced in R2011a

copyElement

Class: `matlab.mixin.Copyable`
Package: `matlab.mixin`

Copy scalar MATLAB object

Syntax

```
b = copyElement(h)
```

Description

`b = copyElement(h)` makes a copy of the scalar handle `h` and returns a scalar handle of the same class.

The sealed `matlab.mixin.Copyable` `copy` method calls the protected `copyElement` method to copy each object in the array. Override `copyElement` in your subclass to control copy behavior.

Input Arguments

h — Handle to copy
scalar handle

Handle to copy, specified as a scalar handle.

Output Arguments

b — Copy of input handle
scalar handle

Copy of input handle, returned as a scalar handle.

Attributes

`Access`	`protected`

To learn about attributes of methods, see Method Attributes.

See Also

`copy`

Topics

"Implement Copy for Handle Classes"

Introduced in R2011a

copyfile

Copy file or folder

Syntax

```
copyfile source
copyfile source destination
copyfile source destination f
```

```
status = copyfile( ___ )
[status,msg] = copyfile( ___ )
[status,msg,msgID] = copyfile( ___ )
```

Description

`copyfile source` copies the file or folder `source` to the current folder. After a successful `copyfile` operation, the timestamp for the new file is the same as the timestamp for `source`.

`copyfile source destination` copies `source` to the file or folder `destination`.

- If `source` is a file, then `destination` can be a file or a folder.
- If `source` is a folder, then `destination` must be a folder.
- If `source` is a folder or specifies multiple files and `destination` does not exist, then `copyfile` attempts to create `destination`.

`copyfile source destination f` copies `source` to `destination`, even when `destination` is not writable. The state of the read/write attribute for `destination` does not change.

`status = copyfile( ___ )` copies the specified file or folder and returns a status of `1` if the operation is successful. Otherwise, `copyfile` returns `0`. You can use this syntax with any of the input argument combinations in the previous syntaxes.

`[status,msg] = copyfile( ___ )` also returns the message text for any warning or error that occurs.

`[status,msg,msgID] = copyfile( ___ )` additionally returns the message ID for any warning or error that occurs.

Examples

Copy File to Another Folder

Copy `myfile1.m` from the current folder to the subfolder `myFolder`.

```
mkdir myFolder
copyfile myfile1.m myFolder
```

Create Copy of File in Current Folder

Create a copy of `myfile1.m` in the current folder, assigning it the name `myfile2.m`.

```
copyfile myfile1.m myfile2.m
```

Copy Files and Folder to New Folder

Copy files and subfolders with names beginning with `my` from the current folder to the folder `newFolder`, where `newFolder` does not already exist.

```
copyfile my* newFolder
```

Copy File to Read-Only Folder

Copy the file `myfile1.m` from the current folder to the read-only folder `restricted`.

Create the read-only folder `restricted`.

```
mkdir restricted
fileattrib restricted -w
```

Copy and rename the file `myfile1.m`. A status of 0 shows the copy was unsuccessful.

```
status = copyfile('myfile1.m', 'restricted');
status
```

```
status = logical
   0
```

Copy the file `myfile1.m` using the 'f' option to override the read-only status of the destination folder. A status of 1 and an empty `message` and `messageId` confirm the copy was successful.

```
[status,message,messageId] = copyfile('myfile1.m', 'restricted', 'f');
status
```

```
status = logical
   1
```

```
message
```

```
message =

  0x0 empty char array
```

```
messageId
```

```
messageId =

  0x0 empty char array
```

Input Arguments

`source` — File or folder to copy

character vector | string scalar

File or folder to copy, specified as a character vector or string scalar. `source` can be an absolute or relative path. To copy multiple files or folders, use wildcards (*).

Note If `source` is a string, enclose all the inputs in parentheses. For example, `movefile("myfile.m","newfolder")`.

`destination` — File or folder destination

character vector | string scalar

File or folder destination, specified as a character vector or string scalar. `destination` can be an absolute or relative path, but cannot include wildcards.

Note If `destination` is a string, enclose all the inputs in parentheses. For example, `movefile("myfile.m","newfolder")`.

Output Arguments

`status` — Copy status

0 | 1

Copy status, indicating if the attempt to move the file or folder is successful, returned as `0` or `1`. If the attempt is successful, the value of `status` is 1. Otherwise, the value is 0.

Data Types: `logical`

`msg` — Error message

character vector

Error message, returned as a character vector. If an error or warning occurs, `msg` contains the message text of the error or warning. Otherwise, `msg` is empty, `''`.

`msgID` — Error message identifier

character vector

Error message identifier, returned as a character vector. If an error or warning occurs, `msgID` contains the message identifier of the error or warning. Otherwise, `msgID` is empty, `''`.

See Also

`cd` | `delete` | `dir` | `mkdir` | `movefile` | `rmdir`

Topics

"Manage Files and Folders"
"Specify File Names"

Introduced before R2006a

copyobj

Copy graphics objects and their descendants

Syntax

```
new_handle = copyobj(h,p)
copyobj( ___ ,'legacy')
```

Description

`copyobj` creates copies of graphics objects and assigns the objects to the new parent.

The new parent must be appropriate for the copied object (for example, you can copy an axes only to figure or uipanel). `copyobj` copies children as well.

`new_handle = copyobj(h,p)` copies one or more graphics objects identified by `h` and returns the handle of the new object or an array of new objects. The new graphics objects are children of the graphics objects specified by `p`.

`copyobj( ___ ,'legacy')` copies object callback properties and object application data. This behavior is consistent with versions of `copyobj` before MATLAB release R2014b.

What is Not Copied

`copyobj` does not copy properties or objects that depend on their original context to operate properly. Objects with default context menus (such as legends and colorbars) create new context menus for the new object. Figures create new toolbars and menus for the new figure.

`copyobj` does *not* copy:

- Callback properties (except when using the `legacy` option).
- Application data associated with the object (except when using the `legacy` option).
- Context menu of legends, colorbars, or other objects that define default context menus.

- Default figure toolbar and menus.
- Axes objects used with the `yyaxis` function.
- The `Interactions` property of an axes object.
- The `DataTipTemplate` property for objects that have this property, such as `Line`, `Scatter`, and `Surface` objects.
- You cannot copy the same object more than once to the same parent in a single call to `copyobj`.

MATLAB changes the `Parent` property to the new parent and assigns the new objects a new handle.

Examples

Copy a surface to a new axes that is in a different figure.

```
h = surf(peaks);
colormap hsv
```

Create the destination figure and axes:

```
fig = figure;
ax = axes;
```

Copy the surface to the new axes and set properties that are not surface properties:

```
new_handle = copyobj(h,ax);
colormap(fig,hsv)
view(ax,3)
grid(ax,'on')
```

Note that while the surface is copied, the `colormap`, `view`, and `grid` are not copied.

Tips

`h` and `p` can be scalars or vectors. When both are vectors, they must be the same length, and the output argument, `new_handle`, is a vector of the same length. In this case, `new_handle(i)` is a copy of `h(i)` with its `Parent` property set to `p(i)`.

When `h` is a scalar and `p` is a vector, `h` is copied once to each of the parents in `p`. Each `new_handle(i)` is a copy of `h` with its `Parent` property set to `p(i)`, and `length(new_handle)` equals `length(p)`.

When `h` is a vector and `p` is a scalar, each `new_handle(i)` is a copy of `h(i)` with its `Parent` property set to `p`. The length of `new_handle` equals `length(h)`.

Note You must copy the associated axes when copying a legend or a colorbar.

When programming a UI, do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the recursion limit.

See Also

`findobj` | `gca` | `gcf` | `gco` | `get` | `set`

Introduced before R2006a

corrcoef

Correlation coefficients

Syntax

```
R = corrcoef(A)
R = corrcoef(A,B)
```

```
[R,P] = corrcoef( ___ )
[R,P,RL,RU] = corrcoef( ___ )
```

```
___ = corrcoef( ___ ,Name,Value)
```

Description

`R = corrcoef(A)` returns the matrix of correlation coefficients on page 1-2530 for `A`, where the columns of `A` represent random variables and the rows represent observations.

`R = corrcoef(A,B)` returns coefficients between two random variables `A` and `B`.

`[R,P] = corrcoef( ___ )` returns the matrix of correlation coefficients and the matrix of p-values for testing the hypothesis that there is no relationship between the observed phenomena (null hypothesis). Use this syntax with any of the arguments from the previous syntaxes. If an off-diagonal element of `P` is smaller than the significance level (default is `0.05`), then the corresponding correlation in `R` is considered significant. This syntax is invalid if `R` contains complex elements.

`[R,P,RL,RU] = corrcoef( ___ )` includes matrices containing lower and upper bounds for a 95% confidence interval for each coefficient. This syntax is invalid if `R` contains complex elements.

`___ = corrcoef( ___ ,Name,Value)` returns any of the output arguments from the previous syntaxes with additional options specified by one or more `Name,Value` pair arguments. For example, `corrcoef(A,'Alpha',0.1)` specifies a 90% confidence interval, and `corrcoef(A,'Rows','complete')` omits all rows of `A` containing one or more `NaN` values.

Examples

Random Columns of Matrix

Compute the correlation coefficients for a matrix with two normally distributed, random columns and one column that is defined in terms of another. Since the third column of A is a multiple of the second, these two variables are directly correlated, thus the correlation coefficient in the (2,3) and (3,2) entries of R is 1.

```
x = randn(6,1);
y = randn(6,1);
A = [x y 2*y+3];
R = corrcoef(A)
```

```
R = 3×3

    1.0000   -0.6237   -0.6237
   -0.6237    1.0000    1.0000
   -0.6237    1.0000    1.0000
```

Two Random Variables

Compute the correlation coefficient matrix between two normally distributed, random vectors of 10 observations each.

```
A = randn(10,1);
B = randn(10,1);
R = corrcoef(A,B)
```

```
R = 2×2

    1.0000    0.4518
    0.4518    1.0000
```

P-Values of Matrix

Compute the correlation coefficients and p-values of a normally distributed, random matrix, with an added fourth column equal to the sum of the other three columns. Since the last column of A is a linear combination of the others, a correlation is introduced between the fourth variable and each of the other three variables. Therefore, the fourth row and fourth column of P contain very small p-values, identifying them as significant correlations.

```
A = randn(50,3);
A(:,4) = sum(A,2);
[R,P] = corrcoef(A)
```

```
R = 4×4

    1.0000    0.1135    0.0879    0.7314
    0.1135    1.0000   -0.1451    0.5082
    0.0879   -0.1451    1.0000    0.5199
    0.7314    0.5082    0.5199    1.0000


P = 4×4

    1.0000    0.4325    0.5438    0.0000
    0.4325    1.0000    0.3146    0.0002
    0.5438    0.3146    1.0000    0.0001
    0.0000    0.0002    0.0001    1.0000
```

Correlation Bounds

Create a normally distributed, random matrix, with an added fourth column equal to the sum of the other three columns, and compute the correlation coefficients, p-values, and lower and upper bounds on the coefficients.

```
A = randn(50,3);
A(:,4) = sum(A,2);
[R,P,RL,RU] = corrcoef(A)
```

```
R = 4×4

    1.0000    0.1135    0.0879    0.7314
```

```
    0.1135    1.0000   -0.1451    0.5082
    0.0879   -0.1451    1.0000    0.5199
    0.7314    0.5082    0.5199    1.0000


P = 4×4

    1.0000    0.4325    0.5438    0.0000
    0.4325    1.0000    0.3146    0.0002
    0.5438    0.3146    1.0000    0.0001
    0.0000    0.0002    0.0001    1.0000


RL = 4×4

    1.0000   -0.1702   -0.1952    0.5688
   -0.1702    1.0000   -0.4070    0.2677
   -0.1952   -0.4070    1.0000    0.2825
    0.5688    0.2677    0.2825    1.0000


RU = 4×4

    1.0000    0.3799    0.3575    0.8389
    0.3799    1.0000    0.1388    0.6890
    0.3575    0.1388    1.0000    0.6974
    0.8389    0.6890    0.6974    1.0000
```

The matrices RL and RU give lower and upper bounds, respectively, on each correlation coefficient according to a 95% confidence interval by default. You can change the confidence level by specifying the value of `Alpha`, which defines the percent confidence, `100*(1-Alpha)`%. For example, use an `Alpha` value equal to 0.01 to compute a 99% confidence interval, which is reflected in the bounds RL and RU. The intervals defined by the coefficient bounds in RL and RU are bigger for 99% confidence compared to 95%, since higher confidence requires a more inclusive range of potential correlation values.

```
[R,P,RL,RU] = corrcoef(A,'Alpha',0.01)
```

```
R = 4×4

    1.0000    0.1135    0.0879    0.7314
    0.1135    1.0000   -0.1451    0.5082
    0.0879   -0.1451    1.0000    0.5199
```

```
    0.7314    0.5082    0.5199    1.0000


P = 4×4

    1.0000    0.4325    0.5438    0.0000
    0.4325    1.0000    0.3146    0.0002
    0.5438    0.3146    1.0000    0.0001
    0.0000    0.0002    0.0001    1.0000


RL = 4×4

    1.0000   -0.2559   -0.2799    0.5049
   -0.2559    1.0000   -0.4792    0.1825
   -0.2799   -0.4792    1.0000    0.1979
    0.5049    0.1825    0.1979    1.0000


RU = 4×4

    1.0000    0.4540    0.4332    0.8636
    0.4540    1.0000    0.2256    0.7334
    0.4332    0.2256    1.0000    0.7407
    0.8636    0.7334    0.7407    1.0000
```

NaN Values

Create a normally distributed matrix involving NaN values, and compute the correlation coefficient matrix, excluding any rows that contain NaN.

```
A = randn(5,3);
A(1,3) = NaN;
A(3,2) = NaN;
A
```

```
A = 5×3

    0.5377   -1.3077       NaN
    1.8339   -0.4336    3.0349
   -2.2588       NaN    0.7254
    0.8622    3.5784   -0.0631
```

```
    0.3188    2.7694    0.7147
```

```
R = corrcoef(A,'Rows','complete')
```

```
R = 3×3

    1.0000   -0.8506    0.8222
   -0.8506    1.0000   -0.9987
    0.8222   -0.9987    1.0000
```

Use `'all'` to include all `NaN` values in the calculation.

```
R = corrcoef(A,'Rows','all')
```

```
R = 3×3

     1   NaN   NaN
   NaN   NaN   NaN
   NaN   NaN   NaN
```

Use `'pairwise'` to compute each two-column correlation coefficient on a pairwise basis. If one of the two columns contains a `NaN`, that row is omitted.

```
R = corrcoef(A,'Rows','pairwise')
```

```
R = 3×3

    1.0000   -0.3388    0.4649
   -0.3388    1.0000   -0.9987
    0.4649   -0.9987    1.0000
```

Input Arguments

A — Input array
matrix

Input array, specified as a matrix.

- If A is a scalar, `corrcoef(A)` returns `NaN`.
- If A is a vector, `corrcoef(A)` returns `1`.

Data Types: `single` | `double`
Complex Number Support: Yes

B — Additional input array
vector | matrix | multidimensional array

Additional input array, specified as a vector, matrix, or multidimensional array.

- A and B must be the same size.
- If A and B are scalars, then `corrcoef(A,B)` returns `1`. If A and B are equal, however, `corrcoef(A,B)` returns `NaN`.
- If A and B are matrices or multidimensional arrays, then `corrcoef(A,B)` converts each input into its vector representation and is equivalent to `corrcoef(A(:),B(:))` or `corrcoef([A(:) B(:)])`.
- If A and B are 0-by-0 empty arrays, `corrcoef(A,B)` returns a 2-by-2 matrix of `NaN` values.

Data Types: `single` | `double`
Complex Number Support: Yes

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `R = corrcoef(A,'Alpha',0.03)`

`Alpha` — Significance level
0.05 (default) | number between 0 and 1

Significance level, specified as a number between 0 and 1. The value of the `'Alpha'` parameter defines the percent confidence level, 100*(1-`Alpha`)%, for the correlation coefficients, which determines the bounds in RL and RU.

Data Types: `single` | `double`

Rows — Use of NaN option
`'all'` (default) | `'complete'` | `'pairwise'`

Use of `NaN` option, specified as one of these values:

- `'all'` — Include all `NaN` values in the input before computing the correlation coefficients.
- `'complete'` — Omit any rows of the input containing `NaN` values before computing the correlation coefficients. This option always returns a positive semi-definite matrix.
- `'pairwise'` — Omit any rows containing `NaN` only on a pairwise basis for each two-column correlation coefficient calculation. This option can return a matrix that is not positive semi-definite.

Data Types: `char`

Output Arguments

R — Correlation coefficients
matrix

Correlation coefficients, returned as a matrix.

- For one matrix input, `R` has size `[size(A,2) size(A,2)]` based on the number of random variables (columns) represented by `A`. The diagonal entries are set to one by convention, while the off-diagonal entries are correlation coefficients of variable pairs. The values of the coefficients can range from -1 to 1, with -1 representing a direct, negative correlation, 0 representing no correlation, and 1 representing a direct, positive correlation. `R` is symmetric.
- For two input arguments, `R` is a 2-by-2 matrix with ones along the diagonal and the correlation coefficients along the off-diagonal.
- If any random variable is constant, its correlation with all other variables is undefined, and the respective row and column value is `NaN`.

P — P-values
matrix

P-values, returned as a matrix. `P` is symmetric and is the same size as `R`. The diagonal entries are all ones and the off-diagonal entries are the p-values for each variable pair. P-

values range from 0 to 1, where values close to 0 correspond to a significant correlation in R and a low probability of observing the null hypothesis.

RL — Lower bound for correlation coefficient
matrix

Lower bound for correlation coefficient, returned as a matrix. RL is symmetric and is the same size as R. The diagonal entries are all ones and the off-diagonal entries are the 95% confidence interval lower bound for the corresponding coefficient in R. The syntax returning RL is invalid if R contains complex values.

RU — Upper bound for correlation coefficient
matrix

Upper bound for correlation coefficient, returned as a matrix. RU is symmetric and is the same size as R. The diagonal entries are all ones and the off-diagonal entries are the 95% confidence interval upper bound for the corresponding coefficient in R. The syntax returning RL is invalid if R contains complex values.

More About

Correlation Coefficient

The correlation coefficient of two random variables is a measure of their linear dependence. If each variable has N scalar observations, then the Pearson correlation coefficient is defined as

$$\rho(A,B) = \frac{1}{N-1}\sum_{i=1}^{N}\left(\overline{\frac{A_i-\mu_A}{\sigma_A}}\right)\left(\frac{B_i-\mu_B}{\sigma_B}\right),$$

where μ_A and σ_A are the mean and standard deviation of A, respectively, and μ_B and σ_B are the mean and standard deviation of B. Alternatively, you can define the correlation coefficient in terms of the covariance of A and B:

$$\rho(A,B) = \frac{\mathrm{cov}(A,B)}{\sigma_A\sigma_B}.$$

The correlation coefficient *matrix* of two random variables is the matrix of correlation coefficients for each pairwise variable combination,

$$R = \begin{pmatrix} \rho(A,A) & \rho(A,B) \\ \rho(B,A) & \rho(B,B) \end{pmatrix}.$$

Since A and B are always directly correlated to themselves, the diagonal entries are just 1, that is,

$$R = \begin{pmatrix} 1 & \rho(A,B) \\ \rho(B,A) & 1 \end{pmatrix}.$$

References

[1] Fisher, R.A. *Statistical Methods for Research Workers*, 13th Ed., Hafner, 1958.

[2] Kendall, M.G. *The Advanced Theory of Statistics*, 4th Ed., Macmillan, 1979.

[3] Press, W.H., Teukolsky, S.A., Vetterling, W.T., and Flannery, B.P. *Numerical Recipes in C*, 2nd Ed., Cambridge University Press, 1992.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- `A` and `B` must be tall arrays of the same size, even if both are vectors.
- Inputs `A` and `B` cannot be scalars for `corrcoef(A,B)`.
- The second input `B` must be 2-D.
- The `'pairwise'` option is not supported.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Row-vector input is only supported when the first two inputs are vectors and nonscalar.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cov` | `mean` | `plotmatrix` | `std`

Introduced before R2006a

matlab.lang.correction.AppendArgumentsCorrection class

Package: `matlab.lang.correction`

Correct error by appending missing input arguments

Description

Use `AppendArgumentsCorrection` objects in functions that throw an `MException` object. `MException` uses `AppendArgumentsCorrection` instances to display suggestions for missing input arguments.

Creation

Description

`aac = matlab.lang.correction.AppendArgumentsCorrection(arguments)` creates a correction that suggests appending input `arguments` to the function call that threw the `MException` object.

Input Arguments

`arguments` — Suggested arguments
string vector | character vector | cell array of character vectors

Suggested arguments, specified as a string or character vector, or a cell array of character vectors.

Examples

Suggest Fix When Function Called Without Arguments

The function `hello` requires one input argument. Add a suggested input argument `"world"` to the error message.

```
function hello(audience)
if nargin < 1
    aac = matlab.lang.correction.AppendArgumentsCorrection('"world"');
    error(aac, 'MATLAB:notEnoughInputs', 'Not enough input arguments.')
end
fprintf("Hello, %s!\n", audience)
end
```

Call the function without an argument.

```
hello
```

```
Error using hello (line 4)
Not enough input arguments.

Did you mean:
>> hello("world")
```

See Also

Functions
`addCorrection` | `error`

Properties
Correction

Introduced in R2019a

matlab.lang.correction.ConvertToFunctionNotationCorrection class

Package: `matlab.lang.correction`

Correct error by converting to function notation

Description

Use `ConvertToFunctionNotationCorrection` objects in classes whose methods should not be called using dot notation. An `MException` object thrown by a method can use a `ConvertToFunctionNotationCorrection` instance to suggest converting dot notation to function notation syntax for calling the method.

Creation

Description

`cfnc = matlab.lang.correction.ConvertToFunctionNotationCorrection(method)` creates a correction that suggests converting dot notation to function notation syntax for calling the `method` that threw the `MException` object.

Input Arguments

`method` — Name of method
string scalar | character vector

Name of the method incorrectly called using dot notation, specified as a string scalar or character vector. `method` must be a valid MATLAB identifier. A valid MATLAB identifier is a string scalar or character vector of alphanumerics (A–Z, a–z, 0–9) and underscores, where the first character is a letter and the length of the text is less than or equal to `namelengthmax`.

Examples

Suggest Fix When Method Called with Dot Notation

Create a class whose methods should not be called using dot notation. Suggest the function notation syntax whenever a method of the class is invoked using dot notation.

In your current folder, create a class `myClass` by subclassing the `handle` superclass. Within a `methods` block, overload `subsref` to restrict invoking methods on `myClass` objects to the function notation syntax. To add a suggested syntax to the error message when a method is called using dot notation, use a `ConvertToFunctionNotationCorrection` instance within the `subsref` method.

```
classdef myClass < handle
    properties
        myProperty
    end
    methods (Hidden)
        function ref = subsref(obj, idx)
            firstSubs = idx(1).subs;
            if idx(1).type ~= "." || any(string(firstSubs) == properties(obj))
                % Parentheses indexing, brace indexing, or property indexing
                try
                    ref = builtin('subsref', obj, idx);
                    return
                catch me
                end
            elseif any(string(firstSubs) == methods(obj))
                % Valid method called using dot notation
                me = MException('myClass:useFunctionForm', ...
                    'Use function syntax to call the ''%s'' method.', ...
                    firstSubs);
                cfnc = matlab.lang.correction.ConvertToFunctionNotationCorrection(firstSubs);
                me = me.addCorrection(cfnc);
            else
                % Invalid method, property, or field called using dot notation
                me = MException('MATLAB:noSuchMethodOrField', ...
                    'Unrecognized method, property, or field ''%s'' for class ''%s''.', ...
                    firstSubs, class(obj));
            end
            throwAsCaller(me)
        end
    end
end
```

Create an instance of `myClass` and call the `isvalid` method using dot notation. `isvalid` is one of the methods that `myClass` inherits from its superclass.

```
myObject = myClass;
myObject.isvalid
```

```
Use function syntax to call the 'isvalid' method.

Did you mean:
>> isvalid(myObject)
```

See Also

Functions
`addCorrection` | `error` | `subsref` | `throwAsCaller`

Properties
Correction

Topics

"Method Invocation"

Introduced in R2019b

matlab.lang.correction.ReplaceIdentifierCorrection class

Package: `matlab.lang.correction`

Correct error by replacing identifier in function call

Description

Use `ReplaceIdentifierCorrection` objects in functions that throw an `MException` object. `MException` uses `ReplaceIdentifierCorrection` instances to display suggestions for incorrect identifiers in function calls.

Creation

Description

`ric = matlab.lang.correction.ReplaceIdentifierCorrection(identifier, replacement)` creates a correction that suggests replacing `identifier` with `replacement` in the function call that threw the `MException` object.

Input Arguments

`identifier` — Incorrect identifier
string scalar | character vector

Incorrect identifier in the function call, specified as a string scalar or character vector. `identifier` must be a valid MATLAB identifier. A valid MATLAB identifier is a string scalar or character vector of alphanumerics (A–Z, a–z, 0–9) and underscores, where the first character is a letter and the length of the text is less than or equal to `namelengthmax`.

`replacement` — Suggested identifier
string scalar | character vector

Suggested identifier for use in the function call, specified as a string scalar or character vector. `replacement` must be a valid MATLAB identifier.

Examples

Suggest Fix When Function Called with Incorrect Value

The functions `walk` and `sprint` require their input arguments to fall within different ranges, and their ranges do not overlap. To add a suggested replacement to the error message when one of these functions is called with an incorrect value, include a `ReplaceIdentifierCorrection` instance in each function.

Create the function `walk` by saving the following code in `walk.m`.

```
function walk(speed)
if speed > 6.5
    ric = matlab.lang.correction.ReplaceIdentifierCorrection('walk', 'sprint');
    error(ric, 'walk:maxSpeed', 'Cannot walk faster than 6.5 km/h.')
elseif speed <= 0
    error('walk:minSpeed', 'Speed must be greater than zero.')
end
fprintf('You are walking at a speed of %2.2f km/h.\n', speed)
end
```

Create the function `sprint` by saving this code in `sprint.m`.

```
function sprint(speed)
if speed <= 6.5
    ric = matlab.lang.correction.ReplaceIdentifierCorrection('sprint', 'walk');
    error(ric, 'sprint:minSpeed', 'Cannot sprint slower than 6.5 km/h.')
elseif speed > 20
    error('sprint:maxSpeed', 'Cannot sprint faster than 20 km/h.')
end
fprintf('You are sprinting at a speed of %2.2f km/h.\n', speed)
end
```

Call `walk` with an input greater than `6.5`.

```
walk(10)
```

```
Error using walk (line 4)
Cannot walk faster than 6.5 km/h.
```

```
Did you mean:
>> sprint(10)
```

See Also

Functions
`addCorrection` | `error`

Properties
Correction

Introduced in R2019b

cos

Cosine of argument in radians

Syntax

```
Y = cos(X)
```

Description

`Y = cos(X)` returns the cosine for each element of `X`. The `cos` function operates element-wise on arrays. The function accepts both real and complex inputs.

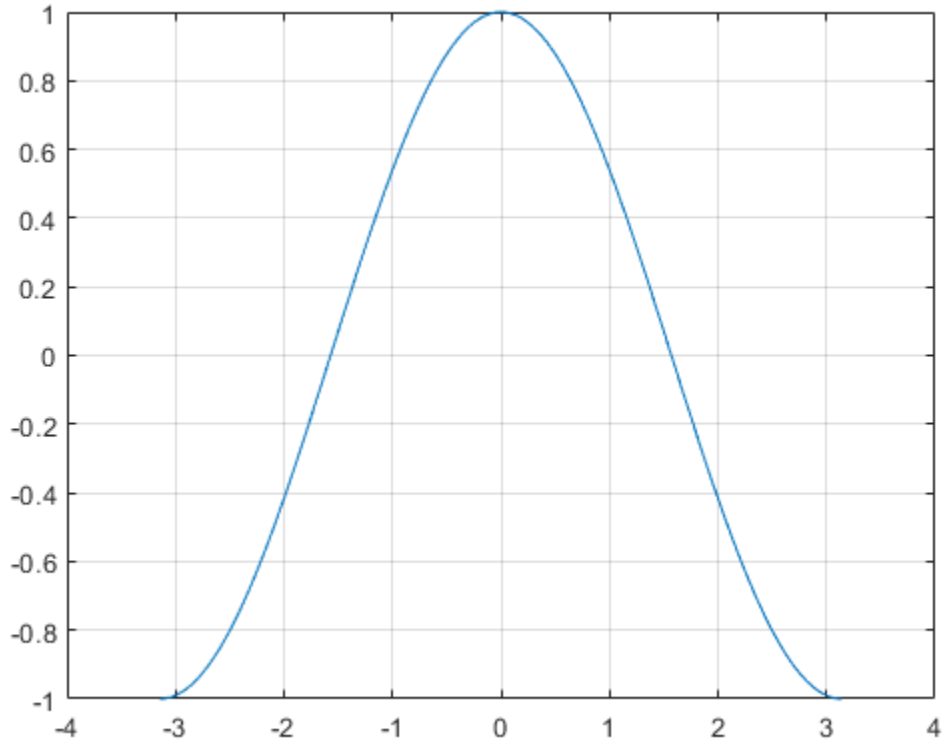
- For real values of `X`, `cos(X)` returns real values in the interval [-1, 1].
- For complex values of `X`, `cos(X)` returns complex values.

Examples

Plot Cosine Function

Plot the cosine function over the domain $-\pi \le x \le \pi$.

```
x = -pi:0.01:pi;
plot(x,cos(x))
grid on
```



Cosine of Vector of Complex Angles

Calculate the cosine of the complex angles in vector x.

```
x = [-i pi+i*pi/2 -1+i*4];
y = cos(x)
```

```
y = 1×3 complex

   1.5431 + 0.0000i  -2.5092 - 0.0000i  14.7547 +22.9637i
```

Input Arguments

X — Input angle in radians

scalar | vector | matrix | multidimensional array

Input angle in radians, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Cosine of input angle

scalar | vector | matrix | multidimensional array

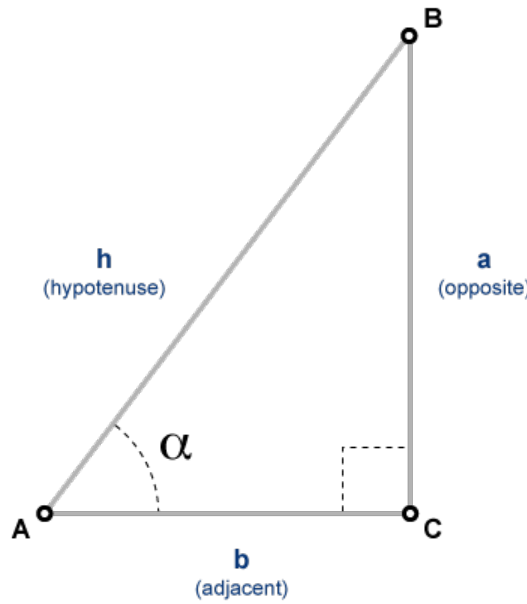
Cosine of input angle, returned as a real-valued or complex-valued scalar, vector, matrix or multidimensional array.

More About

Cosine Function

The cosine of an angle, α, defined with reference to a right angled triangle is

$$\cos(\alpha) = \frac{\text{adjacent side}}{\text{hypotenuse}} = \frac{b}{h} \ .$$



The cosine of a complex argument, α, is

$$\cos(\alpha) = \frac{e^{i\alpha} + e^{-i\alpha}}{2} .$$

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`acos` | `acosd` | `cosd` | `cosh` | `cospi`

Introduced before R2006a

cosd

Cosine of argument in degrees

Syntax

```
Y = cosd(X)
```

Description

`Y = cosd(X)` returns the cosine of the elements of `X`, which are expressed in degrees.

Examples

Cosine of 90 degrees compared to cosine of π/2 radians

```
cosd(90)
```

```
ans = 0
```

```
cos(pi/2)
```

```
ans = 6.1232e-17
```

Cosine of complex angles specified in degrees

Create an array of three complex angles and compute the cosine.

```
z = [180+i 45+2i 10+3i];
y = cosd(z)
```

```
y = 1×3 complex
```

```
-1.0002 + 0.0000i   0.7075 - 0.0247i   0.9862 - 0.0091i
```

Input Arguments

X — Angle in degrees
scalar value | vector | matrix | N-D array

Angle in degrees, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `cosd` operation is element-wise when X is nonscalar.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Cosine of angle
scalar value | vector | matrix | N-D array

Cosine of angle, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as X.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`acos` | `acosd` | `cos`

Introduced before R2006a

cospi

Compute cos(X*pi) accurately

Syntax

```
Y = cospi(X)
```

Description

`Y = cospi(X)` computes `cos(X*pi)` without explicitly computing `X*pi`. This calculation is more accurate than `cos(X*pi)` because the floating-point value of `pi` is an approximation of π. In particular:

- For odd integers, `cospi(n/2)` is exactly zero.
- For integers, `cospi(n)` is +1 or -1.

Examples

Calculate Cosine of Multiples of π

Compare the accuracy of `cospi(X)` vs. `cos(X*pi)`.

Create a vector of values.

```
X = [0 1/2 1 3/2 2];
```

Calculate the cosine of `X*pi` using the normal `cos` function.

```
Y = cos(X*pi)
```

```
Y = 1×5

    1.0000    0.0000   -1.0000   -0.0000    1.0000
```

The results contain small numerical errors due to the fact that `pi` is a floating-point approximation of the true value of π. For instance, `Y(2)` is not *exactly* zero even though $\cos\left(\frac{\pi}{2}\right) = 0$.

```
Y(2)
```

```
ans = 6.1232e-17
```

Use `cospi` to calculate the same values. In this case, the results are exact.

```
Z = cospi(X)
```

```
Z = 1×5

     1     0    -1     0     1
```

```
Z(2)
```

```
ans = 0
```

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cos` | `cosd` | `sinpi`

Introduced in R2018b

cosh

Hyperbolic cosine

Syntax

```
Y = cosh(X)
```

Description

`Y = cosh(X)` returns the hyperbolic cosine of the elements of `X`. The `cosh` function operates element-wise on arrays. The function accepts both real and complex inputs. All angles are in radians.

Examples

Hyperbolic Cosine of Vector

Create a vector and calculate the hyperbolic cosine of each value.

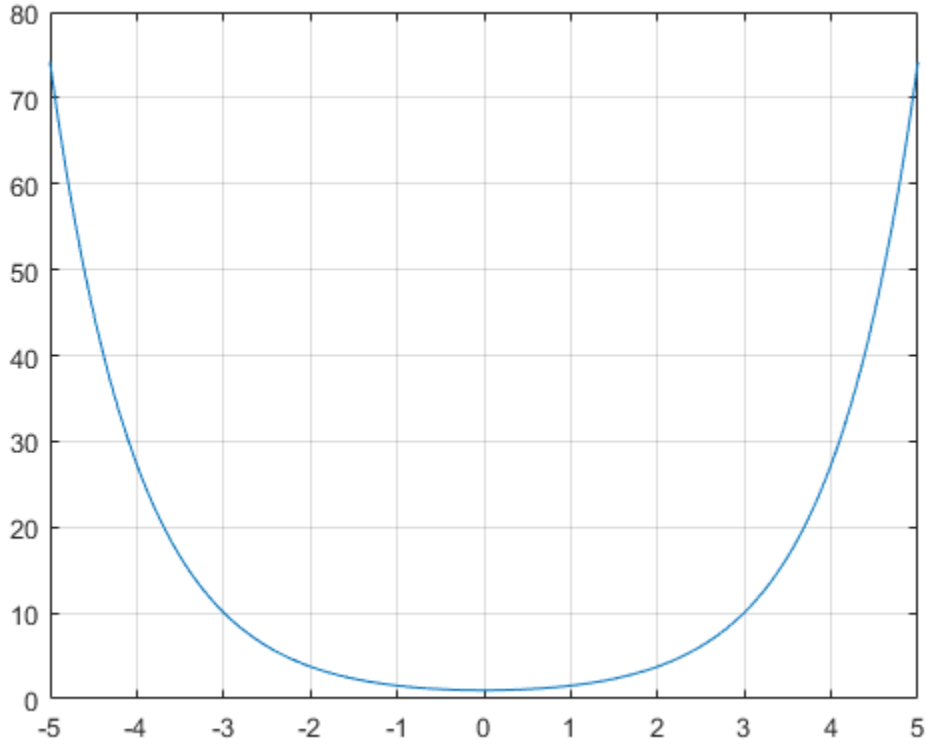
```
X = [0 pi 2*pi 3*pi];
Y = cosh(X)
```

```
Y = 1×4
10^3 ×

    0.0010    0.0116    0.2677    6.1958
```

Graph of Hyperbolic Cosine

Plot the hyperbolic cosine function over the domain $-5 \le x \le 5$.

```
x = -5:0.01:5;
y = cosh(x);
plot(x,y)
grid on
```

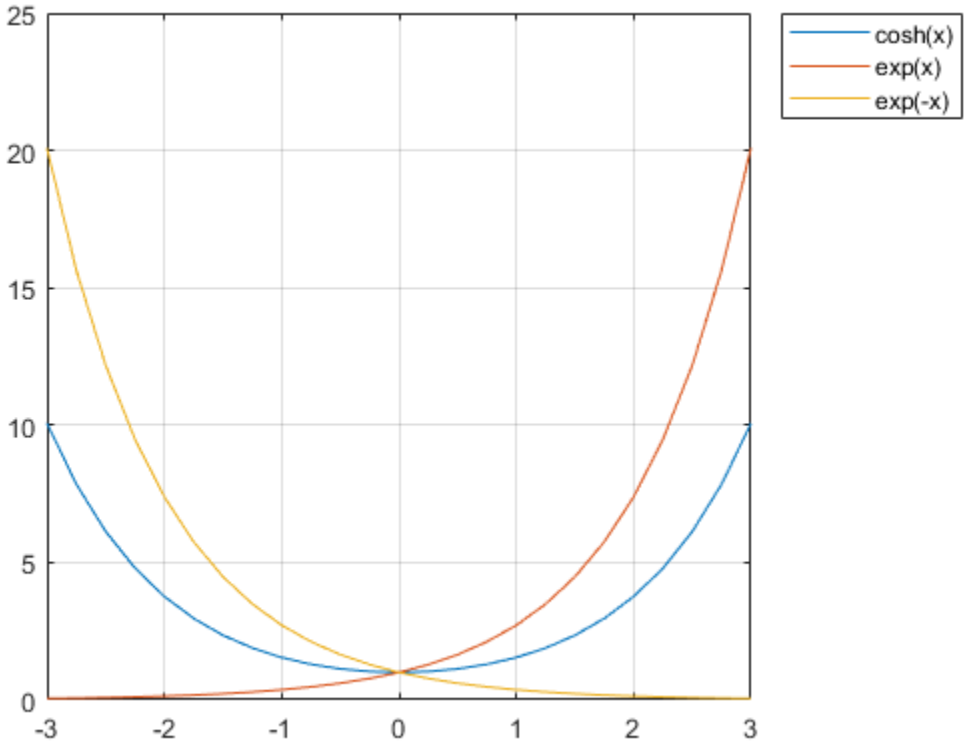


Plot Hyperbolic Cosine and Exponential Functions

The hyperbolic cosine satisfies the identity $\cosh(x) = \frac{e^x + e^{-x}}{2}$. In other words, $\cosh(x)$ is the *average* of e^x and e^{-x}. Verify this by plotting the functions.

Create a vector of values between -3 and 3 with a step of 0.25. Calculate and plot the values of `cosh(x)`, `exp(x)`, and `exp(-x)`. As expected, the curve for `cosh(x)` lies between the two exponential curves.

```
x = -3:0.25:3;
y1 = cosh(x);
y2 = exp(x);
y3 = exp(-x);
plot(x,y1,x,y2,x,y3)
grid on
legend('cosh(x)','exp(x)','exp(-x)','Location','bestoutside')
```



Input Arguments

X — Input angles in radians
scalar | vector | matrix | multidimensional array

Input angles in radians, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Hyperbolic Cosine

The hyperbolic cosine of an angle x can be expressed in terms of exponential functions as

$$\cosh(x) = \frac{e^x + e^{-x}}{2}.$$

In terms of the traditional cosine function with a complex argument, the identity is

$$\cosh(x) = \cos(ix) \ .$$

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`acosh` | `cos` | `rad2deg` | `sinh` | `tanh`

Introduced before R2006a

cot

Cotangent of angle in radians

Syntax

```
Y = cot(X)
```

Description

`Y = cot(X)` returns the cotangent of elements of `X`. The `cot` function operates element-wise on arrays. The function accepts both real and complex inputs.

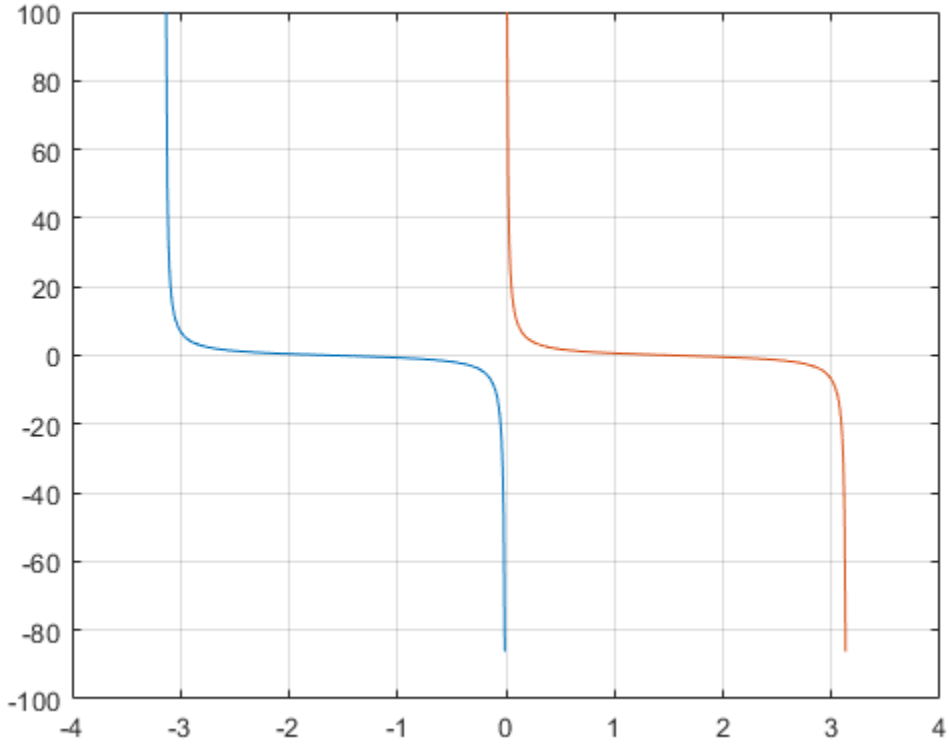
- For real values of `X`, `cot(X)` returns real values in the interval [-∞, ∞].
- For complex values of `X`, `cot(X)` returns complex values.

Examples

Plot Cotangent Function

Plot the cotangent function over the domain $-\pi < x < 0$ and $0 < x < \pi$.

```
x1 = -pi+0.01:0.01:-0.01;
x2 = 0.01:0.01:pi-0.01;
plot(x1,cot(x1),x2,cot(x2)), grid on
```



Cotangent of Vector of Complex Angles

Calculate the cotangent of the complex angles in vector x.

```
x = [-i pi+i*pi/2 -1+i*4];
y = cot(x)
```

```
y = 1×3 complex

   0.0000 + 1.3130i  -0.0000 - 1.0903i  -0.0006 - 0.9997i
```

Input Arguments

X — Input angle in radians
scalar | vector | matrix | multidimensional array

Input angle in radians, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Cotangent of input angle
scalar | vector | matrix | multidimensional array

Cotangent of input angle, returned as a real-valued or complex-valued scalar, vector, matrix or multidimensional array.

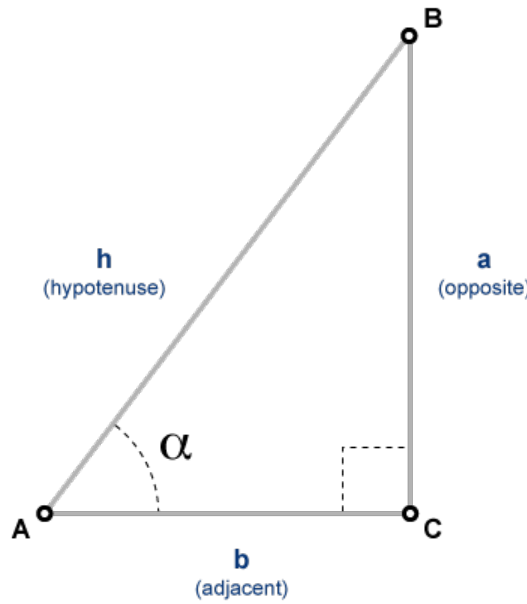
More About

Cotangent Function

The cotangent of an angle, α, defined with reference to a right angled triangle is

$$\cot(\alpha) = \frac{1}{\tan(\alpha)} = \frac{\text{adjacent side}}{\text{opposite side}} = \frac{b}{a} \ .$$

.



The cotangent of a complex argument α is

$$\cot(\alpha) = \frac{i\left(e^{i\alpha} + e^{-i\alpha}\right)}{\left(e^{i\alpha} - e^{-i\alpha}\right)} .$$

.

Tips

- In floating-point arithmetic, `cot` is a bounded function. That is, `cot` does not return values of `Inf` or `-Inf` at points of divergence that are multiples of `pi`, but a large magnitude number instead. This stems from the inaccuracy of the floating-point representation of π.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`acot` | `acotd` | `acoth` | `cotd` | `coth`

Introduced before R2006a

cotd

Cotangent of argument in degrees

Syntax

```
Y = cotd(X)
```

Description

`Y = cotd(X)` returns the cotangent of the elements of `X`, which are expressed in degrees.

Examples

Cotangent of angles approaching 90 and 180 degrees

Create a vector of input angles consisting of 90° and the next smaller and larger double precision numbers. Then compute the cotangent.

```
x1 = [90-eps(90) 90 90+eps(90)];
y1 = cotd(x1)
```

```
y1 = 1×3
10^-15 ×

    0.2480         0   -0.2480
```

`cotd` returns zero when the input angle is exactly 90°. Evaluation at the next smaller double-precision angle returns a slightly positive result. Likewise, the cotangent is slightly negative when the input angle is the next double-precision number larger than 90.

The behavior is similar for input angles near 180°.

```
x2 = [180-eps(180) 180 180+eps(180)];
y2 = cotd(x2)
```

```
y2 = 1×3
10^15 ×

   -2.0159      -Inf    2.0159
```

Cotangent of complex angle, specified in degrees

```
x = 35+5i;
y = cotd(x)
```

```
y = 1.3958 - 0.2606i
```

Input Arguments

X — Angle in degrees
scalar value | vector | matrix | N-D array

Angle in degrees, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `cotd` operation is element-wise when X is nonscalar.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Cotangent of angle
scalar value | vector | matrix | N-D array

Cotangent of angle, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as X.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- In some cases, generated code returns `-Inf` when MATLAB returns `Inf`.
- In some cases, generated code returns `Inf` when MATLAB returns `-Inf`.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`acot` | `acotd` | `cot`

Introduced before R2006a

coth

Hyperbolic cotangent

Syntax

```
Y = coth(X)
```

Description

`Y = coth(X)` returns the hyperbolic tangent of the elements of `X`. The `coth` function operates element-wise on arrays. The function accepts both real and complex inputs. All angles are in radians.

Examples

Hyperbolic Cotangent of Vector

Create a vector and calculate the hyperbolic cotangent of each value.

```
X = [0 pi 2*pi 3*pi];
Y = coth(X)
```

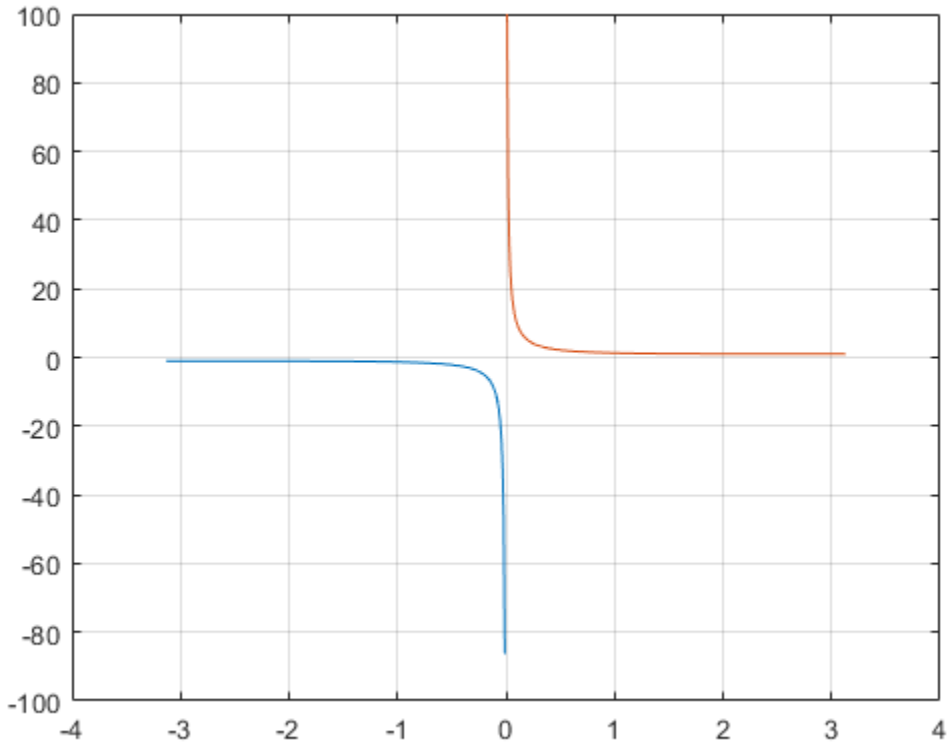
```
Y = 1×4

       Inf    1.0037    1.0000    1.0000
```

Graph of Hyperbolic Cotangent

Plot the hyperbolic cotangent over the domain $-\pi < x < 0$ and $0 < x < \pi$.

```
x1 = -pi+0.01:0.01:-0.01;
x2 = 0.01:0.01:pi-0.01;
```

```
y1 = coth(x1);
y2 = coth(x2);
plot(x1,y1,x2,y2)
grid on
```



Input Arguments

X — Input angles in radians
scalar | vector | matrix | multidimensional array

Input angles in radians, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Hyperbolic Cotangent

The hyperbolic cotangent of x is equal to the inverse of the hyperbolic tangent

$$\coth(x) = \frac{1}{\tanh(x)} = \frac{e^{2x}+1}{e^{2x}-1}.$$

In terms of the traditional cotangent function with a complex argument, the identity is

$$\coth(x) = i\cot(ix) \ .$$

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`acoth` | `cosh` | `cot` | `sinh` | `tanh`

Introduced before R2006a

count

Count occurrences of pattern in strings

Syntax

```
A = count(str,pattern)
A = count(str,pattern,'IgnoreCase',true)
```

Description

`A = count(str,pattern)` returns the number of occurrences of `pattern` in `str`.

If `pattern` is an array containing multiple patterns, then `count` returns the sum of the occurrences of all elements of `pattern` in `str`.

`A = count(str,pattern,'IgnoreCase',true)` ignores case when counting the number of occurrences of `pattern`.

Examples

Count Occurrences of Pattern

Count the number of occurrences of the sequence of characters, `red`, in string arrays.

Starting in R2017a, you can create a string using double quotes.

```
str = "paired with red shoes"
```

```
str =
"paired with red shoes"
```

To count the occurrences of `red`, use the `count` function. In this example, the result is 2 because `red` is also part of the word `paired`.

```
A = count(str,"red")
```

```
A = 2
```

Create a 2-by-1 string array.

```
str = ["red green red red blue blue green";
       "green red blue green green blue"]
```

```
str = 2x1 string array
    "red green red red blue blue green"
    "green red blue green green blue"
```

Count the occurrences of `red` in each element of `str`. If `str` is a string array or cell array of character vectors, then `A` is a numeric array that has the same size.

```
A = count(str,"red")
```

```
A = 2×1

     3
     1
```

Occurrences of Multiple Patterns

Count the total number of occurrences of `red` and `blue` in a string array.

Starting in R2017a, you can create strings using double quotes.

```
str = ["red green blue";
       "green red blue green blue"]
```

```
str = 2x1 string array
    "red green blue"
    "green red blue green blue"
```

`count` returns 2 for the first string because `red` and `blue` each occur once. `count` returns 3 for the second string because `red` occurs once and `blue` occurs twice.

```
A = count(str,["red","blue"])
```

```
A = 2×1
```

```
     2
     3
```

Ignore Case When Counting Patterns

Count the number of occurrences of the letter E in a string array that contains names, ignoring case.

Starting in R2017a, you can create strings using double quotes.

```
str = ["Edgar Allan Poe";"Louisa May Alcott"]
```

```
str = 2x1 string array
    "Edgar Allan Poe"
    "Louisa May Alcott"
```

```
A = count(str,'E','IgnoreCase',true)
```

```
A = 2×1

     2
     0
```

Count Pattern in Character Vector

Count the number of times `al` occurs in the word `alphabetical`.

```
chr = 'alphabetical'
```

```
chr =
'alphabetical'
```

```
A = count(chr,'al')
```

```
A = 2
```

Input Arguments

`str` — Input text
string array | character vector | cell array of character vectors

Input text, specified as a string array, a character vector, or a cell array of character vectors.

`pattern` — Search pattern
string array | character vector | cell array of character vectors

Search pattern, specified as a string array, a character vector, or a cell array of character vectors.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `str` and `pattern` must be a string scalar, a character vector, or a cell array containing not more than one character vector.

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`contains` | `endsWith` | `find` | `numel` | `startsWith` | `strlength`

Topics

"Create String Arrays"
"Search and Replace Text"
"Test for Empty Strings and Missing Values"

Introduced in R2016b

countcats

Count occurrences of categorical array elements by category

Syntax

```
B = countcats(A)
B = countcats(A,dim)
```

Description

`B = countcats(A)` returns the number of elements in each category of the categorical array, `A`.

- If `A` is a vector, then `countcats` returns the number of elements in each category.
- If `A` is a matrix, then `countcats` treats the columns of `A` as vectors and returns the category counts for each column of `A`.
- If `A` is a multidimensional array, then `countcats` acts along the first array dimension whose size does not equal `1`.

`B = countcats(A,dim)` returns the category counts along dimension `dim`.

For example, you can return the category counts of each row in a categorical array using `countcats(A,2)`.

Examples

Category Counts of Categorical Vector

Create a 1-by-5 categorical vector.

```
A = categorical({'plane' 'car' 'train' 'car' 'plane'})
```

```
A = 1x5 categorical array
     plane      car      train      car      plane
```

A has three categories, car, plane, and train.

Find the number of elements in each category of A.

```
B = countcats(A)
```

```
B = 1×3

     2     2     1
```

The first element in B corresponds to the first category of A, which is car. The second element in B corresponds to the second category of A, which is plane. The third element of B corresponds to the third category of A, which is train.

Since A is a row vector, countcats returns a row vector.

Category Counts of Each Column in Array

Create a 3-by-2 categorical array, A, from a numeric array.

```
valueset = 1:3;
catnames = {'red' 'green' 'blue'};

A = categorical([1 3; 2 1; 3 1],valueset,catnames)
```

```
A = 3x2 categorical array
     red        blue
     green      red
     blue       red
```

A has three categories, red, green, and blue.

Find the category counts of each column in A.

```
B = countcats(A)
```

```
B = 3×2

     1     2
     1     0
     1     1
```

The first row of B corresponds to the first category of A. The value, red, occurs once in the first column of A and twice in the second column.

The second row of B corresponds to the second category of A. The value, green, occurs once in the first column of A, and it does not occur in the second column.

The third row of B corresponds to the third category of A. The value, blue, occurs once in the first column of A and once in the second column.

Category Counts of Each Row in Array

Create a 3-by-2 categorical array, A, from a numeric array.

```
valueset = 1:3;
catnames = {'red' 'green' 'blue'};

A = categorical([1 3; 2 1; 3 1],valueset,catnames)
```

```
A = 3x2 categorical array
     red        blue
     green      red
     blue       red
```

A has three categories, red, green, and blue.

Find the category counts of A along the second dimension.

```
B = countcats(A,2)
```

```
B = 3×3

     1     0     1
     1     1     0
```

```
     1     0     1
```

The first column of B corresponds to the first category of A. The value, red, occurs once in the first row of A, once in the second row, and once in the third row.

The second column of B corresponds to the second category of A. The value, green, occurs in only one element. It occurs in the second row of A.

The third column of B corresponds to the third category of A. The value, blue, occurs once in the first row of A and once in the third row.

Category Counts of Array Containing Undefined Elements

Create a 6-by-1 categorical array, A, from a numeric array.

```
valueset = 1:3;
catnames = {'red' 'green' 'blue'};

A = categorical([1;3;2;1;3;1],valueset,catnames)
```

```
A = 6x1 categorical array
     red
     blue
     green
     red
     blue
     red
```

Remove the blue category.

```
A = removecats(A,'blue')
```

```
A = 6x1 categorical array
     red
     <undefined>
     green
     red
     <undefined>
     red
```

A has two categories, red and green. Elements of A that were from the blue category are now undefined.

Find the number of elements in each category of A.

```
B = countcats(A)
```

```
B = 2×1

     3
     1
```

The first element in B corresponds to the first category of A. The value, red, occurs three times in A.

The second element in B corresponds to the second category of A. The value, green, occurs once in A.

countcats does not return any information on undefined elements.

Use the summary function to view the number of undefined elements in addition to the number of elements in each category of A.

```
summary(A)
```

```
     red              3
     green            1
     <undefined>      2
```

Input Arguments

A — Categorical array
vector | matrix | multidimensional array

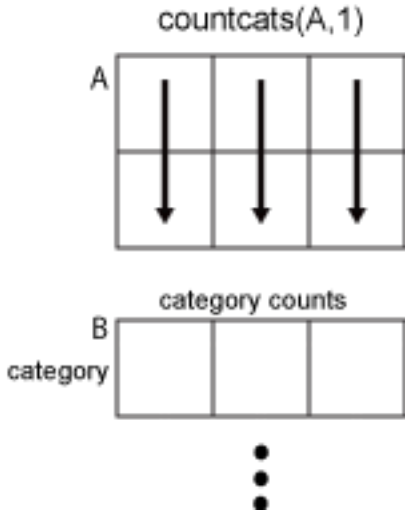
Categorical array, specified as a vector, matrix, or multidimensional array.

dim — Dimension to operate along
positive integer scalar

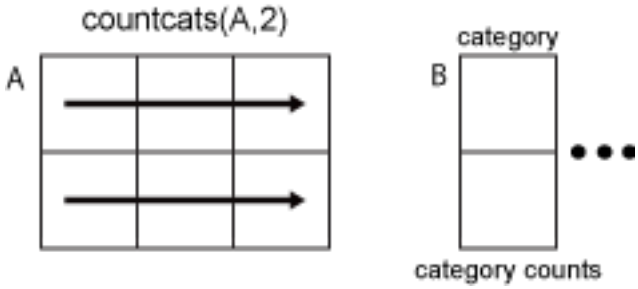
Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a two-dimensional categorical array, `A`.

If `dim = 1`, then `countcats(A,1)` returns the category counts for each column of `A`.



If `dim = 2`, then `countcats(A,2)` returns the category counts of each row of `A`.



If `dim` is greater than `ndims(A)`, then `countcats(A)` returns an array the same size as `A` for each category. `countcats` returns `1` for elements in the corresponding category and `0` otherwise.

Tips

- To find the number of undefined elements in a categorical array, `A`, you must use `summary` or `isundefined`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`categories` | `iscategory` | `ismember` | `isundefined` | `summary`

Introduced in R2013b

cov

Covariance

Syntax

```
C = cov(A)
C = cov(A,B)
C = cov(___,w)
C = cov(___,nanflag)
```

Description

`C = cov(A)` returns the covariance on page 1-2587.

- If `A` is a vector of observations, `C` is the scalar-valued variance on page 1-2588.
- If `A` is a matrix whose columns represent random variables and whose rows represent observations, `C` is the covariance matrix with the corresponding column variances along the diagonal.
- `C` is normalized by the number of observations`-1`. If there is only one observation, it is normalized by 1.
- If `A` is a scalar, `cov(A)` returns `0`. If `A` is an empty array, `cov(A)`returns `NaN`.

`C = cov(A,B)` returns the covariance between two random variables `A` and `B`.

- If `A` and `B` are vectors of observations with equal length, `cov(A,B)` is the 2-by-2 covariance matrix.
- If `A` and `B` are matrices of observations, `cov(A,B)` treats `A` and `B` as vectors and is equivalent to `cov(A(:),B(:))`. `A` and `B` must have equal size.
- If `A` and `B` are scalars, `cov(A,B)` returns a 2-by-2 block of zeros. If `A` and `B` are empty arrays, `cov(A,B)` returns a 2-by-2 block of `NaN`.

`C = cov(___,w)` specifies the normalization weight for any of the previous syntaxes. When `w = 0` (default), `C` is normalized by the number of observations`-1`. When `w = 1`, it is normalized by the number of observations.

`C = cov( ___ ,nanflag)` specifies a condition for omitting `NaN` values from the calculation for any of the previous syntaxes. For example, `cov(A,'omitrows')` will omit any rows of `A` with one or more `NaN` elements.

Examples

Covariance of Matrix

Create a 3-by-4 matrix and compute its covariance.

```
A = [5 0 3 7; 1 -5 7 3; 4 9 8 10];
C = cov(A)
```

```
C = 4×4

    4.3333    8.8333   -3.0000    5.6667
    8.8333   50.3333    6.5000   24.1667
   -3.0000    6.5000    7.0000    1.0000
    5.6667   24.1667    1.0000   12.3333
```

Since the number of columns of `A` is 4, the result is a 4-by-4 matrix.

Covariance of Two Vectors

Create two vectors and compute their 2-by-2 covariance matrix.

```
A = [3 6 4];
B = [7 12 -9];
cov(A,B)
```

```
ans = 2×2

    2.3333    6.8333
    6.8333  120.3333
```

Covariance of Two Matrices

Create two matrices of the same size and compute their 2-by-2 covariance.

```
A = [2 0 -9; 3 4 1];
B = [5 2 6; -4 4 9];
cov(A,B)
```

```
ans = 2×2

   22.1667   -6.9333
   -6.9333   19.4667
```

Specify Normalization Weight

Create a matrix and compute the covariance normalized by the number of rows.

```
A = [1 3 -7; 3 9 2; -5 4 6];
C = cov(A,1)
```

```
C = 3×3

   11.5556    5.1111  -10.2222
    5.1111    6.8889    5.2222
  -10.2222    5.2222   29.5556
```

Covariance Excluding NaN

Create a matrix and compute its covariance, excluding any rows containing NaN values.

```
A = [1.77 -0.005 3.98; NaN -2.95 NaN; 2.54 0.19 1.01]
```

```
A = 3×3

    1.7700   -0.0050    3.9800
       NaN   -2.9500       NaN
    2.5400    0.1900    1.0100
```

```
C = cov(A,'omitrows')
```

```
C = 3×3

    0.2964    0.0751   -1.1435
    0.0751    0.0190   -0.2896
   -1.1435   -0.2896    4.4104
```

Input Arguments

A — Input array
vector | matrix

Input array, specified as a vector or matrix.

Data Types: `single` | `double`

B — Additional input array
vector | matrix

Additional input matrix, specified as a vector or matrix. `B` must be the same size as `A`.

Data Types: `single` | `double`

w — Normalization weight
0 (default) | 1

Normalization weight, specified as one of these values:

- `0` — The output is normalized by the number of observations `-1`. If there is only one observation, it is normalized by 1.
- `1` — The output is normalized by the number of observations.

Data Types: `single` | `double`

nanflag — NaN condition
`'includenan'` (default) | `'omitrows'` | `'partialrows'`

`NaN` condition, specified as one of these values:

- `'includenan'` — include all `NaN` values in the input prior to computing the covariance.
- `'omitrows'` — omit any row of input containing one or more `NaN` values prior to computing the covariance.
- `'partialrows'` — omit rows containing `NaN` only on a pairwise basis for each two-column covariance calculation.

Data Types: `char`

Output Arguments

`C` — Covariance
scalar | matrix

Covariance, specified as a scalar or matrix.

- For single matrix input, `C` has size `[size(A,2) size(A,2)]` based on the number of random variables (columns) represented by `A`. The variances of the columns are along the diagonal. If `A` is a row or column vector, `C` is the scalar-valued variance.
- For two-vector or two-matrix input, `C` is the 2-by-2 covariance matrix between the two random variables. The variances are along the diagonal of `C`.

More About

Covariance

For two random variable vectors A and B, the covariance is defined as

$$\mathrm{cov}(A,B) = \frac{1}{N-1}\sum_{i=1}^{N}(A_i - \mu_A)^*(B_i - \mu_B)$$

where μ_A is the mean of A, μ_B is the mean of B, and * denotes the complex conjugate.

The *covariance matrix* of two random variables is the matrix of pairwise covariance calculations between each variable,

$$C = \begin{pmatrix} \mathrm{cov}(A,A) & \mathrm{cov}(A,B) \\ \mathrm{cov}(B,A) & \mathrm{cov}(B,B) \end{pmatrix}.$$

For a matrix A whose columns are each a random variable made up of observations, the covariance matrix is the pairwise covariance calculation between each column combination. In other words, $C(i, j) = \mathrm{cov}(A(:, i), A(:, j))$.

Variance

For a random variable vector A made up of N scalar observations, the variance is defined as

$$V = \frac{1}{N-1} \sum_{i=1}^{N} |A_i - \mu|^2$$

where μ is the mean of A,

$$\mu = \frac{1}{N} \sum_{i=1}^{N} A_i .$$

Some definitions of variance use a normalization factor of N instead of $N-1$, which can be specified by setting w to 1. In either case, the mean is assumed to have the usual normalization factor N.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

- A and B must be tall arrays of the same size, even if both are vectors.
- The `'partialrows'` option is not supported.

For more information, see “Tall Arrays for Out-of-Memory Data”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If the input is variable-size and is `[]` at run time, returns `[]` not `NaN`.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`corrcoef` | `mean` | `median` | `std` | `var` | `xcorr` | `xcov`

Introduced before R2006a

cplxpair

Sort complex numbers into complex conjugate pairs

Syntax

```
B = cplxpair(A)
B = cplxpair(A,tol)
B = cplxpair(A,[],dim)
B = cplxpair(A,tol,dim)
```

Description

`B = cplxpair(A)` sorts the elements along different dimensions of a complex array, grouping together complex conjugate pairs.

The conjugate pairs are ordered by increasing real part. Within a pair, the element with negative imaginary part comes first. The purely real values are returned following all the complex pairs. The complex conjugate pairs are forced to be exact complex conjugates. A default tolerance of `100*eps` relative to `abs(A(i))` determines which numbers are real and which elements are paired complex conjugates.

If `A` is a vector, `cplxpair(A)` returns `A` with complex conjugate pairs grouped together.

If `A` is a matrix, `cplxpair(A)` returns `A` with its columns sorted and complex conjugates paired.

If `A` is a multidimensional array, `cplxpair(A)` treats the values along the first non-singleton dimension as vectors, returning an array of sorted elements.

`B = cplxpair(A,tol)` overrides the default tolerance.

`B = cplxpair(A,[],dim)` sorts `A` along the dimension specified by scalar dim.

`B = cplxpair(A,tol,dim)` sorts `A` along the specified dimension and overrides the default tolerance.

Diagnostics

If there are an odd number of complex numbers, or if the complex numbers cannot be grouped into complex conjugate pairs within the tolerance, `cplxpair` generates the error message

```
Complex numbers can't be paired.
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Introduced before R2006a

cputime

Elapsed CPU time

Syntax

```
cputime
```

Description

`cputime` returns the total CPU time (in seconds) used by your MATLAB application from the time it was started. This number can overflow the internal representation and wrap around.

Examples

The following code returns the CPU time used to run `surf(peaks(40))`.

```
t = cputime;
surf(peaks(40));
e = cputime-t

e =
     0.4667
```

Tips

To measure performance, it is recommended that you use the `timeit` or `tic` and `toc` functions. For more information, see Using tic and toc Versus the cputime Function.

See Also

`clock` | `etime` | `tic` | `timeit` | `toc`

Introduced before R2006a

RandStream.create

Create random number streams

Class

`RandStream`

Syntax

```
[s1,s2,...] = RandStream.create('gentype','NumStreams',n)
s = RandStream.create('gentype')
[ ... ] = RandStream.create('gentype', Name, Value,...)
```

Description

`[s1,s2,...] = RandStream.create('gentype','NumStreams',n)` creates `n` random number streams that use the uniform pseudorandom number generator algorithm specified by `gentype`. The streams are independent in a pseudorandom sense. The streams are not necessarily independent from streams created at other times. `RandStream.list` returns all possible values for `gentype` or see "Choosing a Random Number Generator" for details on generator algorithms.

Note Multiple streams are not supported by all generator types. Use either the multiplicative lagged Fibonacci generator (`mlfg6331_64`) or the combined multiple recursive generator (`mrg32k3a`) to create multiple streams.

`s = RandStream.create('gentype')` creates a single random stream. The `RandStream` constructor is a more concise alternative when you need to create a single stream.

`[ ... ] = RandStream.create('gentype', Name, Value,...)` allows you to specify optional Name, Value pairs to control creation of the stream. The parameters are:

`NumStreams`	Total number of streams of this type that will be created across sessions or labs. Default is 1.
`StreamIndices`	Stream indices that should be created in this call. Default is `1:N`, where `N` is the value given with the `'NumStreams'` parameter.
`Seed`	Nonnegative scalar integer with which to initialize all streams. Default is 0. Seeds must be an integer between 0 and $2^{32} - 1$ or `'shuffle'` to create a seed based on the current time.
`NormalTransform`	Transformation algorithm used by `randn(S, ...)` to generate normal pseudorandom values. Options are `'Ziggurat'`, `'Polar'`, or `'Inversion'`.
`CellOutput`	Logical flag indicating whether or not to return the stream objects as elements of a cell array. Default is false.

Typically, you call `RandStream.create` once to create multiple independent streams in a single pass. Alternatively, you can create each stream from separate calls to `RandStream.create`, but you must specify the appropriate values for `gentype`, `'NumStreams'`, `'Seed'`, and `'StreamIndices'` to ensure their independence:

- Specify the same set of values for `gentype`, `'NumStreams'`, and `'Seed'` in each case.
- Specify a different value for `'StreamIndices'` that is between `1` and the `'NumStreams'` value in each case.

Examples

Create three independent streams.

```
[s1,s2,s3] = RandStream.create('mrg32k3a','NumStreams',3);
r1 = rand(s1,100000,1);
r2 = rand(s2,100000,1);
```

```
r3 = rand(s3,100000,1);
corrcoef([r1,r2,r3])
```

Create one stream from a set of three independent streams and designate it as the global stream.

```
s2 = RandStream.create('mrg32k3a','NumStreams',3,'StreamIndices',2);
RandStream.setGlobalStream(s2);
```

See Also

`RandStream` | `RandStream.list`

createClassFromWsdl

Create MATLAB class based on WSDL document

Note `createClassFromWsdl` will be removed in a future release. Use `matlab.wsdl.createWSDLClient` instead.

Syntax

```
createClassFromWsdl(source)
```

Description

`createClassFromWsdl(source)` creates a MATLAB class based on the service name defined in `source`.

`createClassFromWsdl` creates a class folder, `@servicename`, in the current folder. The class folder contains:

- A method file for each Web service operation.
- A display method, `display.m`.
- A constructor, `servicename.m`.

Examples

Display Standardized Test Scores

This example illustrates how to use the function. It does not use an actual WSDL document; therefore, you cannot run it. Retrieve information from a database that provides standardized test scores. Assume the WSDL document is located at `http://examplestandardtests.com/scoreswebservice?WSDL`.

Create the MATLAB class, `@TestScoreWebService`, in the current folder.

```
createClassFromWsdl('http://examplestandardtests.com/scoreswebservice?WSDL')
```

```
Retrieving document at 'http://examplestandardtests.com/scoreswebservice?WSDL'
```

Create the service.

```
svc = TestScoreWebService
```

```
endpoint: 'http://examplestandardtests.com/scoreswebservice'
    wsdl: 'http://examplestandardtests.com/scoreswebservice?WSDL'
```

Display the class methods.

```
dir @TestScoreWebService
```

```
display.m
StudentNames.m
Tests.m
TestScoreWebService.m
```

Display the calling syntax for the `StudentNames` function.

```
help StudentNames
```

```
 StudentNames(obj)

    Get names of students who took tests

      Output:
        Names = (string)
```

Get the names. MATLAB creates a structure with the names of the test takers.

```
students = StudentNames(svc)
```

```
students =

    StudentInfo: [125x1 struct]
```

View the data for the first student.

```
students.StudentInfo(1)
```

```
StudentNameLast: 'Benjamin'
StudentNameFirst: 'Ali'
```

Input Arguments

`source` — Web Services Description Language (WSDL) document
string

Web Services Description Language (WSDL) document, specified as a string. The name must include the location of the document, using one of the following:

- URL
- Full path
- Relative path

Example: `'http://examplestandardtests.com/scoreswebservice?WSDL'`

See Also

`createSoapMessage` | `matlab.wsdl.createWSDLClient` | `xmlread`

Topics

"Specify Proxy Server Settings for Connecting to the Internet"

Introduced before R2006a

matlab.project.createProject

Package: `matlab.project`

Create blank project

Syntax

```
proj = matlab.project.createProject
proj = matlab.project.createProject(path)
proj = matlab.project.createProject(name)
```

Description

`proj = matlab.project.createProject` creates and opens a blank project and returns the project object. Use the project object to manipulate the new project at the command line. The new project is created in the default project folder.

`proj = matlab.project.createProject(path)` creates the project in the specified folder.

`proj = matlab.project.createProject(name)` creates the project in the default folder, with the specified name.

Examples

Create Blank Project in the Default Folder

Create a blank project in the default project folder and specify the project name. After the project is created, you can set up the project by adding files to the project, configuring the project path, and specifying startup and shutdown files.

```
proj = matlab.project.createProject;
proj.Name = "My New Project";
```

Create a Blank Project in a Specified Folder

Create a blank project in the specified folder and specify the project name.

```
proj = matlab.project.createProject("C:\work\myprojectname");
```

Create a Named Blank Project in the Default Folder

Create a blank project in the default project folder with the specified name.

```
proj = matlab.project.createProject("myprojectname");
```

Input Arguments

`path` — New project path
character vector | string scalar

New project path, specified as a character vector or string scalar. If you do not specify the path, `matlab.project.createProject` creates the project in the default location. You can change the default location in the project preferences.

`name` — New project name
character vector | string scalar

New project name, specified as a character vector or string scalar.

Output Arguments

`proj` — Project
`matlab.project.Project` object

Project, returned as a `matlab.project.Project` object. Use the `matlab.project.Project` object to programmatically manipulate the currently open project.

Tips

- To change the default folder for new projects, on the **Home** tab, in the **Environment** section, click **Preferences**. Go to **MATLAB > Project** and in the **New Projects** section, set the **Default folder**.

See Also

`addFile` | `addFolderIncludingChildFiles` | `addPath` | `addReference` | `addShortcut` | `currentProject` | `openProject`

Topics

"Create Projects"

Introduced in R2019a

createSoapMessage

Create SOAP (Simple Object Access Protocol) message to send to server

Note `createSoapMessage` will be removed in a future release. Use `matlab.wsdl.createWSDLClient` instead.

Syntax

```
message = createSoapMessage(namespace,method,values,names,types)
message = createSoapMessage(namespace,method,values,names,types,
style)
```

Description

`message = createSoapMessage(namespace,method,values,names,types)` creates a SOAP `message`.

`message = createSoapMessage(namespace,method,values,names,types, style)` creates `message` with specified `style`.

Examples

Retrieve Book Information from Library Database

This example assumes the library is on a local intranet and does not use an actual endpoint; therefore, you cannot run it.

Retrieve the name of the author of a book titled “In the Fall.” The relative path of the library service is `urn:LibraryCatalog`. To get the author's name, use the `getAuthor` function, which takes the book name as the input value. The `getAuthor` parameter is `nameToLookUp`. The XML data type for title is `{http://www.w3.org/2001/XMLSchema}string`. The SOAP message style is `rpc` by default.

Create the SOAP message.

```
message = createSoapMessage(...
  'urn:LibraryCatalog',...
  'getAuthor',...
  {'In the Fall'},...
  {'nameToLookUp'},...
  {'{http://www.w3.org/2001/XMLSchema}string'})
```

```
message =

[#document: null]
```

This response does not necessarily indicate that the message is valid, although certain input problems produce error messages.

Send the message to the server for processing, and get the author's name back. The server endpoint is `http://test/soap/services/LibraryCatalog`. The server method is `urn:LibraryCatalog#getAuthor`.

```
response = callSoapService(...
  'http://test/soap/services/LibraryCatalog',...
  'urn:LibraryCatalog#getAuthor',...
  message)
```

```
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body>
<getAuthorResponse xmlns="urn:LibraryCatalog">
<ns1:getAuthorReturn xmlns:ns1="http://latestversion.soap.test">
Kate Alvin
</ns1:getAuthorReturn>
</getAuthorResponse>
</soapenv:Body>
</soapenv:Envelope>
```

MATLAB returns the message in a single line, displayed here on separate lines for legibility.

Extract the author's name.

```
author = parseSoapResponse(response)
```

```
author = Kate Alvin
```

MATLAB automatically converted the XML string data type to `char`.

Input Arguments

`namespace` — Location of Web service
string

Location of Web service, specified as a string in the form of a valid Uniform Resource Identifier (URI).

Example: `'urn:LibraryCatalog'`

`method` — Name of Web service operation
string

Name of Web service operation, specified as a string.

Example: `'getAuthor'`

`values` — Input arguments for `method`
cell array

Input arguments for `method`, specified as a cell array.

Example: `{'In the Fall'}`

`names` — Parameter for `method`
cell array

Parameter for `method`, specified as a cell array.

Example: `{'nameToLookUp'}`

`types` — XML data types for `values`
cell array

XML data types for `values`, specified as a cell array.

Example: `{'{http://www.w3.org/2001/XMLSchema}string'}`

`style` — Style for structuring SOAP message
`'rpc'` (default) | `'document'`

Style for structuring the SOAP message, specified as one of these values. Use a style supported by the service specified in `namespace`.

`'rpc'`	Remote Procedure Call (RPC) encoding
`'document'`	Document-style encoding

Output Arguments

`message` — Java document object model (DOM)
string

Java document object model (DOM), returned as a string. Use `message` as input to `callSoapService` function.

See Also

`callSoapService` | `matlab.wsdl.createWSDLClient` | `parseSoapResponse` | `urlread` | `xmlread`

Introduced before R2006a

cross

Cross product

Syntax

```
C = cross(A,B)
C = cross(A,B,dim)
```

Description

`C = cross(A,B)` returns the cross product on page 1-2612 of `A` and `B`.

- If `A` and `B` are vectors, then they must have a length of 3.
- If `A` and `B` are matrices or multidimensional arrays, then they must have the same size. In this case, the `cross` function treats `A` and `B` as collections of three-element vectors. The function calculates the cross product of corresponding vectors along the first array dimension whose size equals 3.

`C = cross(A,B,dim)` evaluates the cross product of arrays `A` and `B` along dimension, `dim`. `A` and `B` must have the same size, and both `size(A,dim)` and `size(B,dim)` must be 3. The `dim` input is a positive integer scalar.

Examples

Cross Product of Vectors

Create two 3-D vectors.

```
A = [4 -2 1];
B = [1 -1 3];
```

Find the cross product of `A` and `B`. The result, `C`, is a vector that is perpendicular to both `A` and `B`.

```
C = cross(A,B)
```

```
C = 1×3

    -5   -11    -2
```

Use dot products to verify that C is perpendicular to A and B.

```
dot(C,A)==0 & dot(C,B)==0
```

```
ans = logical
   1
```

The result is logical 1 (`true`).

Cross Product of Matrices

Create two matrices containing random integers.

```
A = randi(15,3,5)
```

```
A = 3×5

    13    14     5    15    15
    14    10     9     3     8
     2     2    15    15    13
```

```
B = randi(25,3,5)
```

```
B = 3×5

     4    20     1    17    10
    11    24    22    19    17
    23    17    24    19     5
```

Find the cross product of A and B.

```
C = cross(A,B)
```

```
C = 3×5

   300   122  -114  -228  -181
  -291  -198  -105   -30    55
    87   136   101   234   175
```

The result, `C`, contains five independent cross products between the columns of `A` and `B`. For example, `C(:,1)` is equal to the cross product of `A(:,1)` with `B(:,1)`.

Cross Product of Multidimensional Arrays

Create two 3-by-3-by-3 multidimensional arrays of random integers.

```
A = randi(10,3,3,3);
B = randi(25,3,3,3);
```

Find the cross product of `A` and `B`, treating the rows as vectors.

```
C = cross(A,B,2)
```

```
C =
C(:,:,1) =

   -34    12    62
    15    72  -109
   -49     8     9


C(:,:,2) =

   198  -164  -170
    45     0   -18
   -55   190  -116


C(:,:,3) =

  -109   -45   131
     1   -74    82
    -6   101  -121
```

The result is a collection of row vectors. For example, `C(1,:,1)` is equal to the cross product of `A(1,:,1)` with `B(1,:,1)`.

Find the cross product of `A` and `B` along the third dimension (`dim = 3`).

```
D = cross(A,B,3)
```

```
D =
D(:,:,1) =

   -14   179  -106
   -56    -4   -75
     2   -37    10


D(:,:,2) =

   -37  -162   -37
    50  -124   -78
     1    63   118


D(:,:,3) =

    62  -170    56
    46    72   105
    -2   -53  -160
```

The result is a collection of vectors oriented in the third dimension. For example, `D(1,1,:)` is equal to the cross product of `A(1,1,:)` with `B(1,1,:)`.

Cross Product of Complex Vectors

Create two complex three-element vectors and find their cross product.

```
x = rand(3,1) + 1i*rand(3,1);
y = rand(3,1) + 1i*rand(3,1);
z = cross(x,y)
```

```
z = 3×1 complex
```

```
   0.1995 - 1.4113i
   0.0477 + 1.5156i
   0.6595 + 0.4222i
```

Use dot products to verify that $x \cdot z = 0$ and $y \cdot z = 0$, within machine precision. In order to preserve the dot product, `cross` calculates a complex-conjugate cross product for complex vectors: $(x \times y)^*$.

```
norm(dot(x,z))
```

```
ans = 3.5544e-16
```

```
norm(dot(y,z))
```

```
ans = 2.7756e-16
```

Input Arguments

`A,B` — Input arrays
numeric arrays

Input arrays, specified as numeric arrays.

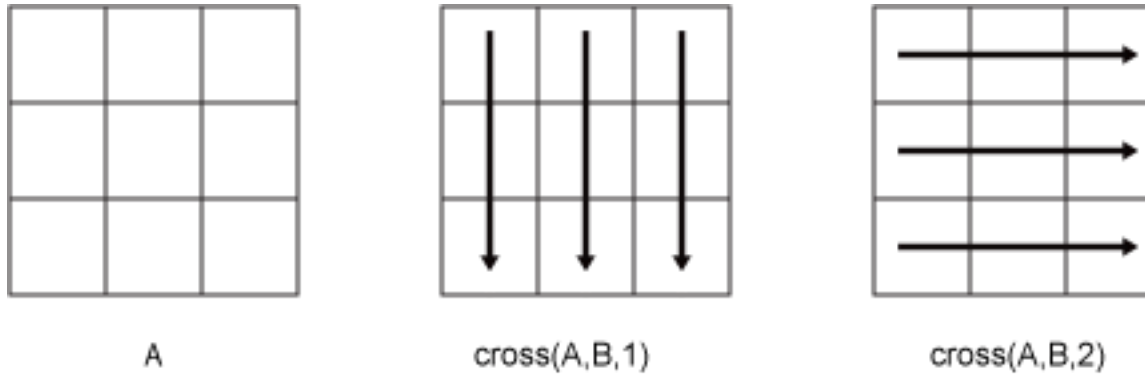
Data Types: `single` | `double`
Complex Number Support: Yes

`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. The size of dimension `dim` must be 3. If no value is specified, the default is the first array dimension whose size equals 3.

Consider two 2-D input arrays, `A` and `B`:

- `cross(A,B,1)` treats the columns of `A` and `B` as vectors and returns the cross products of corresponding columns.
- `cross(A,B,2)` treats the rows of `A` and `B` as vectors and returns the cross products of corresponding rows.



`cross` returns an error if `dim` is greater than `ndims(A)`.

More About

Cross Product

The cross product between two 3-D vectors produces a new vector that is perpendicular to both.

Consider the two vectors

$$A = a_1\hat{i} + a_2\hat{j} + a_3\hat{k} \ ,$$
$$B = b_1\hat{i} + b_2\hat{j} + b_3\hat{k} \ .$$

In terms of a matrix determinant involving the basis vectors $\hat{i}$, $\hat{j}$, and $\hat{k}$, the cross product of A and B is

$$C = A \times B = \begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix}$$

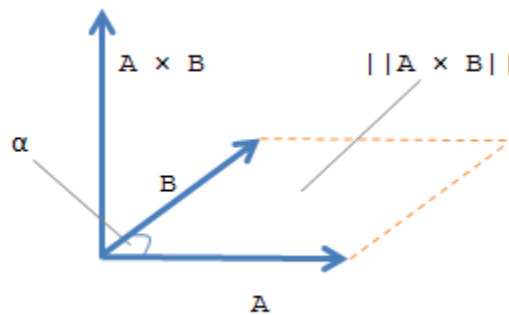
$$= (a_2b_3 - a_3b_2)\hat{i} + (a_3b_1 - a_1b_3)\hat{j} + (a_1b_2 - a_2b_1)\hat{k} \ .$$

For vectors with complex components, the complex conjugate cross-product is calculated $C = (A \times B)^*$. This conjugation preserves the dot products $A \cdot C = B \cdot C = 0$ for complex vectors.

Geometrically, $A \times B$ is perpendicular to both A and B. The magnitude of the cross product, $\|A \times B\|$, is equal to the area of the parallelogram formed using A and B as sides. This area is related to the magnitudes of A and B as well as the angle between the vectors by

$$\|A \times B\| = \|A\| \, \|B\| \sin\alpha \quad .$$

Thus, if A and B are parallel, then the cross product is zero.



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If supplied, `dim` must be a constant.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

- Code generation does not support sparse matrix inputs for this function.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- If supplied, `dim` must be a constant.
- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`dot` | `kron` | `norm`

Introduced before R2006a

csc

Cosecant of input angle in radians

Syntax

```
Y = csc(X)
```

Description

`Y = csc(X)` returns the cosecant of the elements of `X`. The `csc` function operates element-wise on arrays. The function accepts both real and complex inputs.

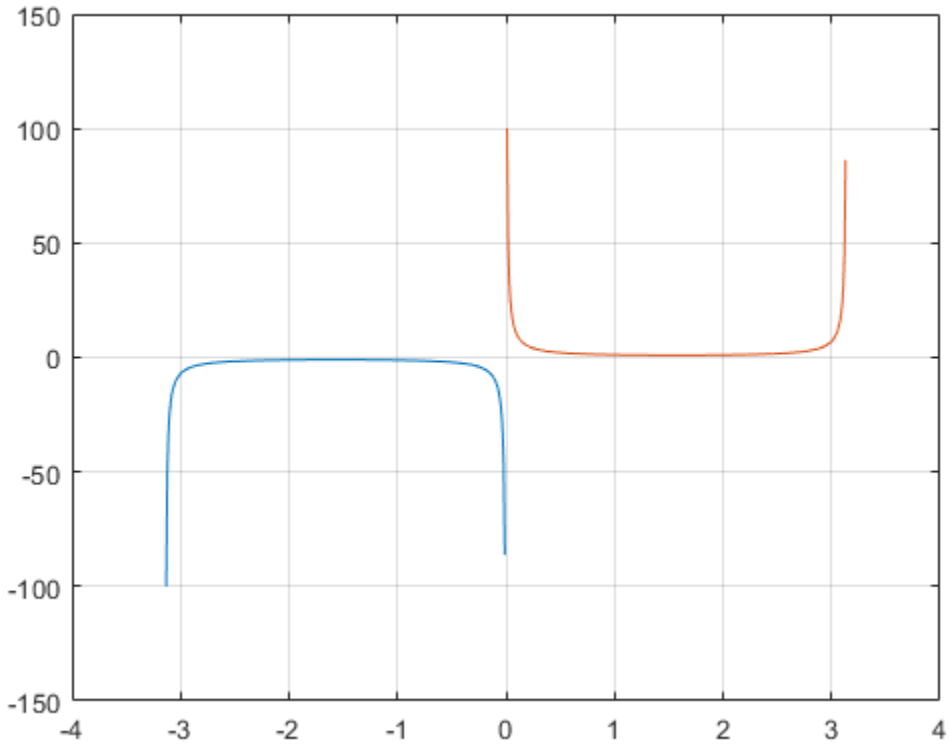
- For real values of `X`, `csc(X)` returns real values in the interval [-∞, -1] and [1, ∞].
- For complex values of `X`, `csc(X)` returns complex values.

Examples

Plot Cosecant Function

Plot the cosecant function over the domain $-\pi < x < 0$ and $0 < x < \pi$ as shown.

```
x1 = -pi+0.01:0.01:-0.01;
x2 = 0.01:0.01:pi-0.01;
plot(x1,csc(x1),x2,csc(x2)), grid on
```



Cosecant of Vector of Complex Angles

Calculate the cosecant of the complex angles in vector x.

```
x = [-i pi+i*pi/2 -1+i*4];
y = csc(x)
```

```
y = 1×3 complex

   0.0000 + 0.8509i   0.0000 + 0.4345i  -0.0308 - 0.0198i
```

Input Arguments

X — Input angle in radians

scalar | vector | matrix | multidimensional array

Input angle in radians, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Cosecant of input angle

scalar | vector | matrix | multidimensional array

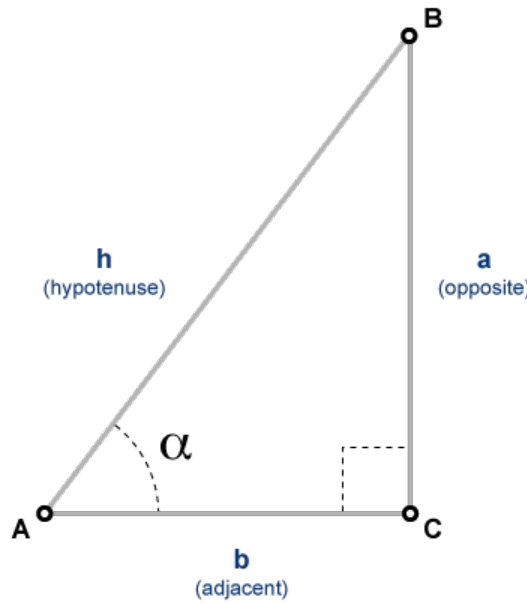
Cosecant of input angle, returned as a real-valued or complex-valued scalar, vector, matrix or multidimensional array.

More About

Cosecant Function

The cosecant of an angle, α, defined with reference to a right angled triangle is

$$\csc(\alpha) = \frac{1}{\sin(\alpha)} = \frac{\text{hypotenuse}}{\text{opposite side}} = \frac{h}{a} \ .$$



The cosecant of a complex argument, α, is

$$\csc(\alpha) = \frac{2i}{e^{i\alpha} - e^{-i\alpha}} .$$

Tips

- In floating-point arithmetic, `csc` is a bounded function. That is, `csc` does not return values of `Inf` or `-Inf` at points of divergence that are multiples of `pi`, but a large magnitude number instead. This stems from the inaccuracy of the floating-point representation of π.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`acsc` | `acscd` | `acsch` | `cscd` | `csch`

Introduced before R2006a

cscd

Cosecant of argument in degrees

Syntax

```
Y = cscd(X)
```

Description

`Y = cscd(X)` returns the cosecant of the elements of `X`, which are expressed in degrees.

Examples

Cosecant of 180 degrees compared to cosecant of π radians

`cscd(180)` is infinite, whereas `csc(pi)` is large but finite.

```
cscd(180)
```

```
ans = Inf
```

```
csc(pi)
```

```
ans = 8.1656e+15
```

Cosecant of vector of complex angles, specified in degrees

```
z = [35+i 15+2i 10+3i];
y = cscd(z)
```

```
y = 1×3 complex
```

```
1.7421 - 0.0434i   3.7970 - 0.4944i   5.2857 - 1.5681i
```

Input Arguments

X — Angle in degrees
scalar value | vector | matrix | N-D array

Angle in degrees, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `cscd` operation is element-wise when X is nonscalar.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Cosecant of angle
scalar value | vector | matrix | N-D array

Cosecant of angle, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as X.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- In some cases, generated code returns `-Inf` when MATLAB returns `Inf`.
- In some cases, generated code returns `Inf` when MATLAB returns `-Inf`.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`acsc` | `acscd` | `csc`

Introduced before R2006a

csch

Hyperbolic cosecant

Syntax

```
Y = csch(X)
```

Description

`Y = csch(X)` returns the hyperbolic cosecant of the elements of X. The `csch` function operates element-wise on arrays. The function accepts both real and complex inputs. All angles are in radians.

Examples

Hyperbolic Cosecant of Vector

Create a vector and calculate the hyperbolic cosecant of each value.

```
X = [0 pi 2*pi 3*pi];
Y = csch(X)
```

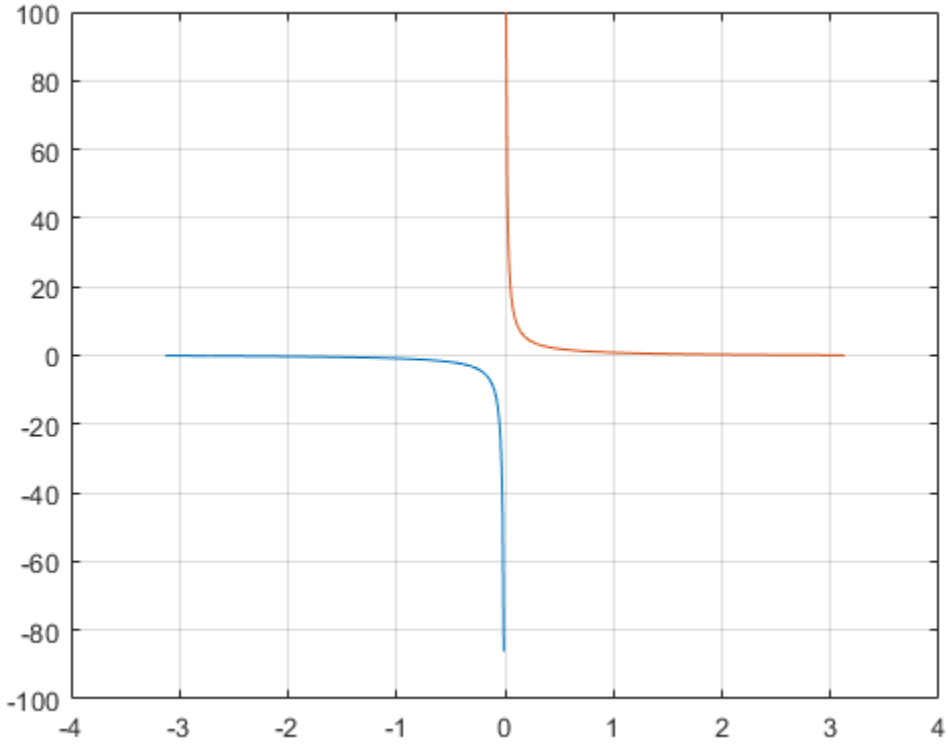
```
Y = 1×4

       Inf    0.0866    0.0037    0.0002
```

Graph of Hyperbolic Cosecant

Plot the hyperbolic cosecant over the domain $-\pi < x < 0$ and $0 < x < \pi$.

```
x1 = -pi+0.01:0.01:-0.01;
x2 = 0.01:0.01:pi-0.01;
```

```
y1 = csch(x1);
y2 = csch(x2);
plot(x1,y1,x2,y2)
grid on
```



Input Arguments

X — Input angles in radians
scalar | vector | matrix | multidimensional array

Input angles in radians, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Hyperbolic Cosecant

The hyperbolic cosecant of x is equal to the inverse of the hyperbolic sine

$$\operatorname{csch}(x) = \frac{1}{\sinh(x)} = \frac{2}{e^x - e^{-x}}.$$

In terms of the traditional cosecant function with a complex argument, the identity is

$$\operatorname{csch}(x) = i\csc(ix) .$$

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`acsch` | `cosh` | `csc` | `sinh`

Introduced before R2006a

csvread

(Not recommended) Read comma-separated value (CSV) file

Note `csvread` is not recommended. Use `readmatrix` instead. For more information, see Compatibility Considerations.

Syntax

```
M = csvread(filename)
M = csvread(filename,R1,C1)
M = csvread(filename,R1,C1,[R1 C1 R2 C2])
```

Description

`M = csvread(filename)` reads a comma-separated value (CSV) formatted file into array `M`. The file must contain only numeric values.

`M = csvread(filename,R1,C1)` reads data from the file starting at row offset `R1` and column offset `C1`. For example, the offsets `R1=0`, `C1=0` specify the first value in the file.

`M = csvread(filename,R1,C1,[R1 C1 R2 C2])` reads only the range bounded by row offsets `R1` and `R2` and column offsets `C1` and `C2`. Another way to define the range is to use spreadsheet notation, such as `'A1..B7'` instead of `[0 0 6 1]`.

Examples

Read Entire CSV File

Create a file named `csvlist.dat` that contains comma-separated values.

```
02, 04, 06, 08
03, 06, 09, 12
05, 10, 15, 20
```

```
07, 14, 21, 28
```

Read the numeric values in the file.

```
filename = 'csvlist.dat';
M = csvread(filename)
```

```
M =

     2     4     6     8
     3     6     9    12
     5    10    15    20
     7    14    21    28
```

Read CSV File Starting at Specific Row and Column Offset

Read the matrix starting two rows below the first row from the file described in the previous example.

```
M = csvread('csvlist.dat',2,0)
```

```
M =

     5    10    15    20
     7    14    21    28
```

Read Specific Range from CSV File

Read the matrix bounded by row offsets 1 and 2 and column offsets 0 and 2 from the file described in the first example.

```
M = csvread('csvlist.dat',1,0,[1,0,2,2])
```

```
M =
```

```
     3     6     9
     5    10    15
```

Input Arguments

filename — File name
character vector | string

File name, specified as a character vector or string.

Example: `'myFile.dat'` or `"myFile.dat"`

Data Types: `char` | `string`

R1 — Starting row offset
0 (default) | nonnegative integer

Starting row offset, specified as a nonnegative integer. The first row has an offset of 0.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

C1 — Starting column offset
0 (default) | nonnegative integer

Starting column offset, specified as a nonnegative integer. The first column has an offset of 0.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

R2 — Ending row offset
nonnegative integer

Ending row offset, specified as a nonnegative integer. The first row has an offset of 0.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

C2 — Ending column offset
nonnegative integer

Ending column offset, specified as a nonnegative integer. The first column has an offset of 0.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Tips

- Skip header rows or columns by specifying row and column offsets. All values in the file other than headers must be numeric.

Algorithms

`csvread` fills empty delimited fields with zero. When the `csvread` function reads data files with lines that end with a nonspace delimiter, such as a semicolon, it returns a matrix, `M`, that has an additional last column of zeros.

`csvread` imports any complex number as a whole into a complex numeric field, converting the real and imaginary parts to the specified numeric type. The table shows valid forms for a complex number.

Form	Example
±<real>±<imag>i|j	5.7-3.1i
±<imag>i|j	-7j

Embedded white space in a complex number is invalid and is regarded as a field delimiter.

Compatibility Considerations

`csvread` is not recommended

Not recommended starting in R2019a

`csvread` is not recommended. Use `readmatrix` instead. There are no plans to remove `csvread`.

Starting in R2019a, use the `readmatrix` function to read a matrix from a csv file. The `readmatrix` function has these advantages over the `csvread` function:

- Better cross-platform support and performance
- Automatic detection of data format and types
- Ability to use import options to control the data import process, including the handling of errors and missing data

This table shows typical usages of `csvread` and how to update your code to use `readmatrix` instead.

Not Recommended	Recommended
`M = csvread(filename)`	`M = readmatrix(filename)`

See Also

`csvwrite` | `dlmread` | `readmatrix` | `readtable` | `textscan` | `uiimport`

Topics

"Import Text Files"

Introduced before R2006a

ctranspose, '

Complex conjugate transpose

Syntax

```
B = A'
B = ctranspose(A)
```

Description

`B = A'` computes the complex conjugate transpose on page 1-2635 of `A`.

`B = ctranspose(A)` is an alternate way to execute `A'`, but is rarely used. It enables operator overloading for classes.

Examples

Conjugate Transpose of Real Matrix

Create a 4-by-2 matrix.

```
A = [2 1; 9 7; 2 8; 3 5]
```

```
A = 4×2

     2     1
     9     7
     2     8
     3     5
```

Find the conjugate transpose of `A`.

```
B = A'
```

```
B = 2×4

     2     9     2     3
     1     7     8     5
```

The result is a 2-by-4 matrix. B has the same elements as A, but the row and column index for each element are interchanged. When no complex elements are present, A' produces the same result as A.'.

Conjugate Transpose of Complex Matrix

Create a 2-by-2 matrix with complex elements.

```
A = [0-1i 2+1i;4+2i 0-2i]
```

```
A = 2×2 complex

   0.0000 - 1.0000i   2.0000 + 1.0000i
   4.0000 + 2.0000i   0.0000 - 2.0000i
```

Find the conjugate transpose of A.

```
B = A'
```

```
B = 2×2 complex

   0.0000 + 1.0000i   4.0000 - 2.0000i
   2.0000 - 1.0000i   0.0000 + 2.0000i
```

The result, B, contains the elements of A with the row and column indices interchanged. The sign of the imaginary part of each number is also switched.

Input Arguments

A — Input array
vector | matrix

Input array, specified as a vector or matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `cell` | `categorical` | `datetime` | `duration` | `calendarDuration`
Complex Number Support: Yes

More About

Complex Conjugate Transpose

The complex conjugate transpose of a matrix interchanges the row and column index for each element, reflecting the elements across the main diagonal. The operation also negates the imaginary part of any complex numbers.

For example, if `B = A'` and `A(1,2)` is `1+1i`, then the element `B(2,1)` is `1-1i`.

Tips

- The nonconjugate transpose operator, `A.'`, performs a transpose without conjugation. That is, it does not change the sign of the imaginary parts of the elements.
- For logical or non-numeric inputs, `ctranspose` and `transpose` produce the same result.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`conj` | `permute` | `transpose`

Topics

“Array vs. Matrix Operations”
“Operator Precedence”
“MATLAB Operators and Special Characters”

Introduced before R2006a

csvwrite

(Not recommended) Write comma-separated value file

Note `csvwrite` is not recommended. Use `writematrix` instead. For more information, see Compatibility Considerations.

Syntax

```
csvwrite(filename,M)
csvwrite(filename,M,row,col)
```

Description

`csvwrite(filename,M)` writes matrix `M` to file `filename` as comma-separated values.

`csvwrite(filename,M,row,col)` writes matrix `M` to file `filename` starting at the specified row and column offset. The row and column arguments are zero based, so that `row=0` and `col=0` specify the first value in the file.

Examples

Write Matrix to Comma-Separated Value File

Create an array of sample data `M`.

```
M = magic(3)
```

```
M = 3×3

     8     1     6
     3     5     7
     4     9     2
```

Write matrix M to the file 'myFile.txt'.

```
csvwrite('myFile.txt',M)
```

View the data in the file.

```
type('myFile.txt')
```

```
8,1,6
3,5,7
4,9,2
```

Write Matrix Starting at Offset

Write a matrix to a file starting at a defined offset position.

Create an array of sample data M.

```
M = magic(3)
```

```
M = 3×3

     8     1     6
     3     5     7
     4     9     2
```

Define the starting offsets to skip one row and two columns.

```
row = 1 ;
col = 2 ;
```

Write matrix M to the file 'myFile.txt', starting at the offset position.

```
csvwrite('myFile.txt',M,row,col)
```

View the data in the file.

```
type('myFile.txt')
```

```
,,,,
,,8,1,6
,,3,5,7
,,4,9,2
```

Input Arguments

`filename` — File name

character vector | string

File name, specified as a character vector or string.

Example: `'myFile.dat'`

Data Types: `char` | `string`

`M` — Numeric data to write

matrix

Numeric data to write, specified as a matrix of numeric values.

Example: `[1,2,3;4,5,6]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

`row` — Row offset

0 (default) | scalar

Row offset, specified as a scalar. The row offset indicates the number of rows to skip before writing the numeric data. `row` is zero-based, so that `row = 0` instructs MATLAB to begin writing in the first row of the destination file. Skipped rows are populated with commas.

`col` — Column offset

0 (default) | scalar

Column offset, specified as a scalar. The column offset indicates the number of columns to skip before writing the numeric data. `col` is zero-based, so that `col = 0` instructs MATLAB to begin writing in the first column of the destination file. Skipped columns are separated by commas.

Limitations

- `csvwrite` writes a maximum of five significant digits. If you need greater precision, use `dlmwrite` with a precision argument.
- `csvwrite` does not accept cell arrays for the input matrix `M`. To export a cell array that contains only numeric data, use `cell2mat` to convert the cell array to a numeric matrix before calling `csvwrite`.

Algorithms

- `csvwrite` terminates each line with a line feed character (`'\n'` or `char(10)`) and no carriage return.

Compatibility Considerations

`csvwrite` is not recommended

Not recommended starting in R2019a

`csvwrite` is not recommended. Use `writematrix` instead. There are no plans to remove `csvwrite`.

Starting in R2019a, use the `writematrix` function to write a matrix to a comma separated text file. The `writematrix` function has better cross-platform support and performance over the `csvwrite` function.

This table shows typical usages of `csvwrite` and how to update your code to use `writematrix` instead.

Not Recommended	Recommended
`csvwrite('mydata.txt',M)`	`writematrix(M,'mydata.txt')`

See Also

`csvread` | `dlmwrite` | `uiimport` | `writematrix` | `writetable` | `xlswrite`

Topics

"Write Data to Text Files"

Introduced before R2006a

cummax

Cumulative maximum

Syntax

```
M = cummax(A)
M = cummax(A,dim)
M = cummax( ___ ,direction)
M = cummax( ___ ,nanflag)
```

Description

`M = cummax(A)` returns the cumulative maximum elements of `A`. By default, `cummax(A)` operates along the first array dimension whose size does not equal 1.

- If `A` is a vector, then `cummax(A)` returns a vector of the same size containing the cumulative maxima of `A`.
- If `A` is a matrix, then `cummax(A)` returns a matrix of the same size containing the cumulative maxima in each column of `A`.
- If `A` is a multidimensional array, then `cummax(A)` returns an array of the same size containing the cumulative maxima along the first array dimension of `A` whose size does not equal 1.

`M = cummax(A,dim)` returns the cumulative maxima along the dimension `dim`. For example, if `A` is a matrix, then `cummax(A,2)` returns the cumulative maxima along the rows of `A`.

`M = cummax( ___ ,direction)` optionally specifies the direction using any of the previous syntaxes. You must specify `A` and, optionally, can specify `dim`. For instance, `cummax(A,2,'reverse')` returns the cumulative maxima of `A` by working from end to beginning of the second dimension of `A`.

`M = cummax( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `cummax(A,'includenan')` includes all `NaN` values in the calculation while `cummax(A,'omitnan')` ignores them.

Examples

Cumulative Maximum Values in Vector

Find the cumulative maxima of a 1-by-10 vector of random integers.

```
v = randi(10,1,10)
```

```
v = 1×10

     9    10     2    10     7     1     3     6    10    10
```

```
M = cummax(v)
```

```
M = 1×10

     9    10    10    10    10    10    10    10    10    10
```

Cumulative Maximum Values in Matrix Columns

Find the cumulative maxima of the columns of a 3-by-3 matrix.

```
A = [3 5 2; 1 6 3; 7 8 1]
```

```
A = 3×3

     3     5     2
     1     6     3
     7     8     1
```

```
M = cummax(A)
```

```
M = 3×3

     3     5     2
     3     6     3
     7     8     3
```

Cumulative Maximum Values in Matrix Rows

Find the cumulative maxima of the rows of a 3-by-3 matrix.

```
A = [3 5 2; 1 6 3; 7 8 1]
```

```
A = 3×3

     3     5     2
     1     6     3
     7     8     1
```

```
M = cummax(A,2)
```

```
M = 3×3

     3     5     5
     1     6     6
     7     8     8
```

Cumulative Maximum Array Values in Reverse Direction

Calculate the cumulative maxima in the third dimension of a 2-by-2-by-3 array. Specify `direction` as `'reverse'` to work from the end of the third dimension to the beginning.

```
A = cat(3,[1 2; 3 4],[9 10; 11 12],[5 6; 7 8])
```

```
A =
A(:,:,1) =

     1     2
     3     4


A(:,:,2) =

     9    10
    11    12
```

```
A(:,:,3) =

     5     6
     7     8


M = cummax(A,3,'reverse')

M =
M(:,:,1) =

     9    10
    11    12


M(:,:,2) =

     9    10
    11    12


M(:,:,3) =

     5     6
     7     8
```

Vector with NaN Values

Create a vector containing NaN values and compute the cumulative maxima. By default, cummax ignores NaN values.

```
A = [3 5 NaN 9 0 NaN];
M = cummax(A)

M = 1×6

     3     5     5     9     9     9
```

If you include NaN values in the calculation, then the cumulative maximum becomes NaN as soon as the first NaN value in A is encountered.

```
M = cummax(A,'includenan')
```

```
M = 1×6

     3     5   NaN   NaN   NaN   NaN
```

Input Arguments

A — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array. For complex elements, cummax compares the magnitude of the elements. If magnitudes are equal, cummax also compares the phase angles.

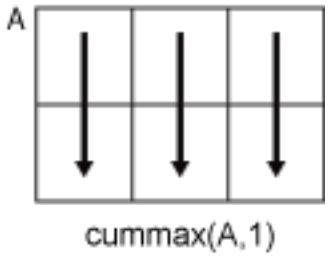
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | duration
Complex Number Support: Yes

dim — Dimension to operate along
positive integer scalar

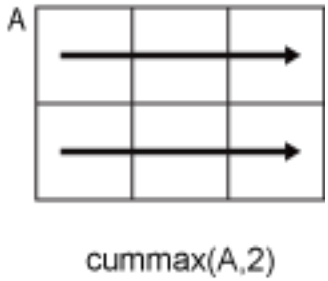
Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a two-dimensional input array, A:

- cummax(A,1) works on successive elements in the columns of A and returns an array of the same size as A with the cumulative maxima in each column.



- `cummax(A,2)` works on successive elements in the rows of `A` and returns an array of the same size as `A` with the cumulative maxima in each row.



`cummax` returns `A` if `dim` is greater than `ndims(A)`.

`direction` — Direction of cumulation
`'forward'` (default) | `'reverse'`

Direction of cumulation, specified as `'forward'` (default) or `'reverse'`.

- `'forward'` works from `1` to `end` of the active dimension.
- `'reverse'` works from `end` to `1` of the active dimension.

Data Types: `char`

`nanflag` — `NaN` condition
`'omitnan'` (default) | `'includenan'`

`NaN` condition, specified as one of the following values:

- `'omitnan'` — Ignore all `NaN` values in the input. If the input has consecutive leading `NaN` values, then `cummax` returns `NaN` in the corresponding elements of the output. For example, `cummax([NaN 7 13 6],'omitnan')` returns the row vector `[NaN 7 13 13]`.
- `'includenan'` — Include `NaN` values from the input when computing the cumulative maxima, resulting in `NaN` values in the output.

Data Types: `char`

Output Arguments

M — Cumulative maxima
vector | matrix | multidimensional array

Cumulative maxima, returned as a vector, matrix, or multidimensional array. The size and data type of `M` are the same as those of `A`.

Tips

- The `'reverse'` option in many cumulative functions allows quick directional calculations without requiring a flip or reflection of the input array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

The `'reverse'` direction is not supported.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `nanflag` argument is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cummin` | `cumprod` | `cumsum` | `max` | `min` | `movmax`

Introduced in R2014b

cummin

Cumulative minimum

Syntax

```
M = cummin(A)
M = cummin(A,dim)
M = cummin( ___ ,direction)
M = cummin( ___ ,nanflag)
```

Description

`M = cummin(A)` returns the cumulative minimum elements of `A`. By default, `cummin(A)` operates along the first array dimension whose size does not equal 1.

- If `A` is a vector, then `cummin(A)` returns a vector of the same size containing the cumulative minima of `A`.
- If `A` is a matrix, then `cummin(A)` returns a matrix of the same size containing the cumulative minima in each column of `A`.
- If `A` is a multidimensional array, then `cummin(A)` returns an array of the same size containing the cumulative minima along the first array dimension of `A` whose size does not equal 1.

`M = cummin(A,dim)` returns the cumulative minima along the dimension `dim`. For example, if `A` is a matrix, then `cummin(A,2)` returns the cumulative minima along the rows of `A`.

`M = cummin( ___ ,direction)` optionally specifies the direction using any of the previous syntaxes. You must specify `A` and, optionally, can specify `dim`. For instance, `cummin(A,2,'reverse')` returns the cumulative minima of `A` by working from end to beginning of the second dimension of `A`.

`M = cummin( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `cummin(A,'includenan')` includes all `NaN` values in the calculation while `cummin(A,'omitnan')` ignores them.

Examples

Cumulative Minimum Values in Vector

Find the cumulative minima of a 1-by-10 vector of random integers.

```
v = randi([0,10],1,10)
```

```
v = 1×10

     8     9     1    10     6     1     3     6    10    10
```

```
M = cummin(v)
```

```
M = 1×10

     8     8     1     1     1     1     1     1     1     1
```

Cumulative Minimum Values in Matrix Columns

Find the cumulative minima of the columns of a 3-by-3 matrix.

```
A = [3 5 2; 1 6 3; 7 8 1]
```

```
A = 3×3

     3     5     2
     1     6     3
     7     8     1
```

```
M = cummin(A)
```

```
M = 3×3

     3     5     2
     1     5     2
     1     5     1
```

Cumulative Minimum Values in Matrix Rows

Find the cumulative minima of the rows of a 3-by-3 matrix.

```
A = [3 5 2; 1 6 3; 7 8 1]
```

```
A = 3×3

     3     5     2
     1     6     3
     7     8     1
```

```
M = cummin(A,2)
```

```
M = 3×3

     3     3     2
     1     1     1
     7     7     1
```

Cumulative Minimum Array Values in Reverse Direction

Calculate the cumulative minima in the third dimension of a 2-by-2-by-3 array. Specify `direction` as `'reverse'` to work from the end of the third dimension to the beginning.

```
A = cat(3,[1 2; 3 4],[9 10; 11 12],[5 6; 7 8])
```

```
A =
A(:,:,1) =

     1     2
     3     4


A(:,:,2) =

     9    10
    11    12
```

```
A(:,:,3) =

     5     6
     7     8


M = cummin(A,3,'reverse')

M =
M(:,:,1) =

     1     2
     3     4


M(:,:,2) =

     5     6
     7     8


M(:,:,3) =

     5     6
     7     8
```

Vector with NaN Values

Create a vector containing NaN values and compute the cumulative minima. By default, cummin ignores NaN values.

```
A = [3 5 NaN 9 0 NaN];
M = cummin(A)

M = 1×6

     3     3     3     3     0     0
```

If you include NaN values in the calculation, then the cumulative minimum becomes NaN as soon as the first NaN value in A is encountered.

```
M = cummin(A,'includenan')
```

```
M = 1×6

     3     3   NaN   NaN   NaN   NaN
```

Input Arguments

A — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array. For complex elements, cummin compares the magnitude of the elements. If magnitudes are equal, cummin also compares the phase angles.

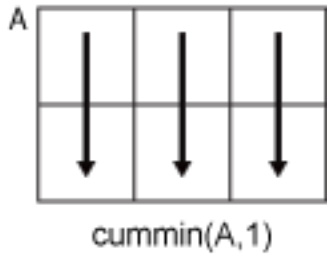
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | duration
Complex Number Support: Yes

dim — Dimension to operate along
positive integer scalar

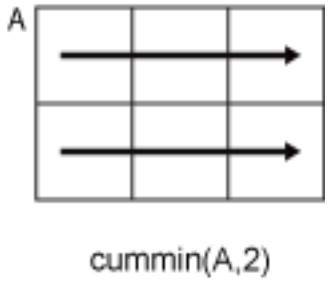
Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a two-dimensional input array, A:

- cummin(A,1) works on successive elements in the columns of A and returns an array of the same size as A with the cumulative minima in each column.



- `cummin(A,2)` works on successive elements in the rows of `A` and returns an array of the same size as `A` with the cumulative minima in each row.



`cummin` returns `A` if `dim` is greater than `ndims(A)`.

`direction` — Direction of cumulation
`'forward'` (default) | `'reverse'`

Direction of cumulation, specified as `'forward'` (default) or `'reverse'`.

- `'forward'` works from `1` to `end` of the active dimension.
- `'reverse'` works from `end` to `1` of the active dimension.

Data Types: `char`

`nanflag` — `NaN` condition
`'omitnan'` (default) | `'includenan'`

`NaN` condition, specified as one of the following values:

- `'omitnan'` — Ignore all `NaN` values in the input. If the input has consecutive leading `NaN` values, then `cummin` returns `NaN` in the corresponding elements of the output. For example, `cummin([NaN 7 13 6],'omitnan')` returns the row vector `[NaN 7 7 6]`.
- `'includenan'` — Include `NaN` values from the input when computing the cumulative minima, resulting in `NaN` values in the output.

Data Types: `char`

Output Arguments

M — Cumulative minima
vector | matrix | multidimensional array

Cumulative minima, returned as a vector, matrix, or multidimensional array. The size and data type of `M` are the same as those of `A`.

Tips

- The `'reverse'` option in many cumulative functions allows quick directional calculations without requiring a flip or reflection of the input array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

The `'reverse'` direction is not supported.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `nanflag` argument is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cummax` | `cumprod` | `cumsum` | `max` | `min` | `movmin`

Introduced in R2014b

cumprod

Cumulative product

Syntax

```
B = cumprod(A)
B = cumprod(A,dim)
B = cumprod( ___ ,direction)
B = cumprod( ___ ,nanflag)
```

Description

`B = cumprod(A)` returns the cumulative product of `A` starting at the beginning of the first array dimension in `A` whose size does not equal 1.

- If `A` is a vector, then `cumprod(A)` returns a vector containing the cumulative product of the elements of `A`.
- If `A` is a matrix, then `cumprod(A)` returns a matrix containing the cumulative products for each column of `A`.
- If `A` is a multidimensional array, then `cumprod(A)` acts along the first nonsingleton dimension on page 1-2664.

`B = cumprod(A,dim)` returns the cumulative product along dimension `dim`. For example, if `A` is a matrix, then `cumprod(A,2)` returns the cumulative product of each row.

`B = cumprod( ___ ,direction)` optionally specifies the direction using any of the previous syntaxes. You must specify `A`, and optionally can specify `dim`. For instance, `cumprod(A,2,'reverse')` returns the cumulative product within the rows of `A` by working from end to beginning of the second dimension.

`B = cumprod( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `cumprod(A,'includenan')` includes `NaN` values in the calculation while `cumprod(A,'omitnan')` ignores them.

Examples

Cumulative Product of Vector

Find the cumulative product of the integers from `1` to `5`. The element `B(2)` is the product of `A(1)` and `A(2)`, while `B(5)` is the product of elements `A(1)` through `A(5)`.

```
A = 1:5;
B = cumprod(A)
```

```
B = 1×5

     1     2     6    24   120
```

Cumulative Product of Each Column in Matrix

Define a 3-by-3 matrix whose elements correspond to their linear indices.

```
A = [1 4 7; 2 5 8; 3 6 9]
```

```
A = 3×3

     1     4     7
     2     5     8
     3     6     9
```

Find the cumulative product of the columns of `A`. The element `B(5)` is the product of `A(4)` and `A(5)`, while `B(9)` is the product of `A(7)` , `A(8)`, and `A(9)`.

```
B = cumprod(A)
```

```
B = 3×3

     1     4     7
     2    20    56
     6   120   504
```

Cumulative Product of Each Row in Matrix

Define a 2-by-3 matrix whose elements correspond to their linear indices.

```
A = [1 3 5; 2 4 6]
```

```
A = 2×3

     1     3     5
     2     4     6
```

Find the cumulative product of the rows of `A`. The element `B(3)` is the product of `A(1)` and `A(3)`, while `B(5)` is the product of `A(1)`, `A(3)`, and `A(5)`.

```
B = cumprod(A,2)
```

```
B = 2×3

     1     3    15
     2     8    48
```

Logical Input with Double Output

Create an array of logical values.

```
A = [true false true; true true false]
```

```
A = 2x3 logical array

   1   0   1
   1   1   0
```

Find the cumulative product of the rows of `A`.

```
B = cumprod(A,2)
```

```
B = 2×3
```

```
     1     0     0
     1     1     0
```

The output has type `double`.

```
class(B)
```

```
ans =
'double'
```

Reverse Cumulative Product

Create a 3-by-3 matrix of random integers between 1 and 10.

```
rng default;
A = randi([1,10],3)
```

```
A = 3×3

     9    10     3
    10     7     6
     2     1    10
```

Calculate the cumulative product along the columns. Specify the `'reverse'` option to work from bottom to top in each column. The result is the same size as `A`.

```
B = cumprod(A,'reverse')
```

```
B = 3×3

   180    70   180
    20     7    60
     2     1    10
```

Vector with NaN Values

Create a vector containing NaN values and compute the cumulative products. By default, cumprod includes NaN values. When you include NaN values in the calculation, the cumulative product becomes NaN as soon as the first NaN value in A is encountered.

```
A = [1 3 NaN 2 4 NaN];
B = cumprod(A)
```

```
B = 1×6

     1     3   NaN   NaN   NaN   NaN
```

Ignore NaN values in the cumulative product calculation using the 'omitnan' option.

```
B = cumprod(A,'omitnan')
```

```
B = 1×6

     1     3     3     6    24    24
```

Input Arguments

A — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

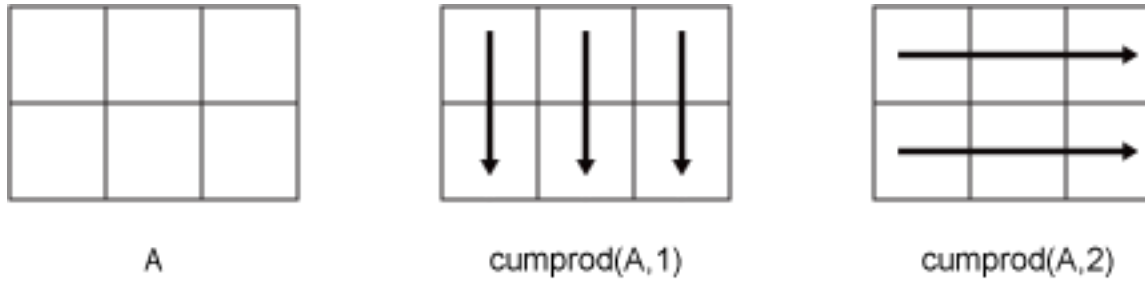
Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical
Complex Number Support: Yes

dim — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a two-dimensional input array, A.

- `cumprod(A,1)` works on successive elements in the columns of `A` and returns the cumulative products of each column.
- `cumprod(A,2)` works on successive elements in the rows of `A` and returns the cumulative products of each row.



`cumprod` returns `A` if `dim` is greater than `ndims(A)`.

`direction` — Direction of cumulation
`'forward'` (default) | `'reverse'`

Direction of cumulation, specified as `'forward'` (default) or `'reverse'`.

- `'forward'` works from `1` to `end` of the active dimension.
- `'reverse'` works from `end` to `1` of the active dimension.

`nanflag` — `NaN` condition
`'includenan'` (default) | `'omitnan'`

`NaN` condition, specified as one of these values:

- `'includenan'` — Include `NaN` values from the input when computing each product, resulting in `NaN` values in the output.
- `'omitnan'` — Ignore `NaN` values in the input. The product of elements containing `NaN` values is the product of all non-`NaN` elements. If all elements are `NaN`, then `cumprod` returns 1.

Output Arguments

`B` — Cumulative product array
vector | matrix | multidimensional array

Cumulative product array, returned as a vector, matrix, or multidimensional array of the same size as the input array A.

The class of B is the same as the class of A except if A is `logical`, in which case B is `double`.

More About

First Nonsingleton Dimension

The first nonsingleton dimension is the first dimension of an array whose size is not equal to 1.

For example:

- If X is a 1-by-n row vector, then the second dimension is the first nonsingleton dimension of X.
- If X is a 1-by-0-by-n empty array, then the second dimension is the first nonsingleton dimension of X.
- If X is a 1-by-1-by-3 array, then the third dimension is the first nonsingleton dimension of X.

Tips

- Many cumulative functions in MATLAB support the `'reverse'` option. This option allows quick directional calculations without needing a flip or reflection of the input array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

The `'reverse'` direction is not supported.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Logical inputs are not supported. Cast input to `double` first.
- Code generation does not support sparse matrix inputs for this function.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Logical inputs are not supported. Cast input to `double` first.
- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `nanflag` argument is not supported.
- The order of the products within the `comprod` operation is not defined. Therefore, the `cumprod` operation on a `gpuArray` might not return exactly the same answer as the `cumprod` operation on the corresponding MATLAB numeric array. The differences might be significant when `A` is a signed integer type.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- The order of the products within the `comprod` operation is not defined. Therefore, the `cumprod` operation on a distributed array might not return exactly the same answer as the `cumprod` operation on the corresponding MATLAB numeric array. The differences might be significant when `A` is a signed integer type.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cummax` | `cummin` | `cumsum` | `movprod` | `prod` | `sum`

Introduced before R2006a

cumsum

Cumulative sum

Syntax

```
B = cumsum(A)
B = cumsum(A,dim)
B = cumsum( ___ ,direction)
B = cumsum( ___ ,nanflag)
```

Description

`B = cumsum(A)` returns the cumulative sum of `A` starting at the beginning of the first array dimension in `A` whose size does not equal 1.

- If `A` is a vector, then `cumsum(A)` returns a vector containing the cumulative sum of the elements of `A`.
- If `A` is a matrix, then `cumsum(A)` returns a matrix containing the cumulative sums for each column of `A`.
- If `A` is a multidimensional array, then `cumsum(A)` acts along the first nonsingleton dimension on page 1-2673.

`B = cumsum(A,dim)` returns the cumulative sum of the elements along dimension `dim`. For example, if `A` is a matrix, then `cumsum(A,2)` returns the cumulative sum of each row.

`B = cumsum( ___ ,direction)` optionally specifies the direction using any of the previous syntaxes. You must specify `A`, and optionally can specify `dim`. For instance, `cumsum(A,2,'reverse')` returns the cumulative sum within the rows of `A` by working from end to beginning of the second dimension.

`B = cumsum( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `cumsum(A,'includenan')` includes all `NaN` values in the calculation while `cumsum(A,'omitnan')` ignores them.

Examples

Cumulative Sum of Vector

Find the cumulative sum of the integers from 1 to 5. The element B(2) is the sum of A(1) and A(2), while B(5) is the sum of elements A(1) through A(5).

```
A = 1:5;
B = cumsum(A)
```

```
B = 1×5

     1     3     6    10    15
```

Cumulative Sum of Each Column in Matrix

Define a 3-by-3 matrix whose elements correspond to their linear indices.

```
A = [1 4 7; 2 5 8; 3 6 9]
```

```
A = 3×3

     1     4     7
     2     5     8
     3     6     9
```

Find the cumulative sum of the columns of A. The element B(5) is the sum of A(4) and A(5), while B(9) is the sum of A(7), A(8), and A(9).

```
B = cumsum(A)
```

```
B = 3×3

     1     4     7
     3     9    15
     6    15    24
```

Cumulative Sum of Each Row in Matrix

Define a 2-by-3 matrix whose elements correspond to their linear indices.

```
A = [1 3 5; 2 4 6]
```

```
A = 2×3

     1     3     5
     2     4     6
```

Find the cumulative sum of the rows of `A`. The element `B(3)` is the sum of `A(1)` and `A(3)`, while `B(5)` is the sum of `A(1)`, `A(3)`, and `A(5)`.

```
B = cumsum(A,2)
```

```
B = 2×3

     1     4     9
     2     6    12
```

Cumulative Sum of Logical Input

Create an array of logical values.

```
A = [true false true; true true false]
```

```
A = 2x3 logical array

   1   0   1
   1   1   0
```

Find the cumulative sum of the rows of `A`.

```
B = cumsum(A,2)
```

```
B = 2×3
```

```
     1     1     2
     1     2     2
```

The output has type `double`.

```
class(B)
```

```
ans =
'double'
```

Reverse Cumulative Sum

Create a 3-by-3 matrix of random integers between 1 and 10.

```
rng default;
A = randi([1,10],3)
```

```
A = 3×3

     9    10     3
    10     7     6
     2     1    10
```

Calculate the cumulative sum along the rows. Specify the `'reverse'` option to work from right to left in each row. The result is the same size as `A`.

```
B = cumsum(A,2,'reverse')
```

```
B = 3×3

    22    13     3
    23    13     6
    13    11    10
```

Vector with NaN Values

Create a vector containing NaN values and compute the cumulative sums. By default, cumsum includes NaN values. When you include NaN values in the calculation, the cumulative sum becomes NaN as soon as the first NaN value in A is encountered.

```
A = [3 5 NaN 9 0 NaN];
B = cumsum(A)
```

```
B = 1×6

     3     8   NaN   NaN   NaN   NaN
```

You can ignore NaN values in the cumulative sum calculation using the 'omitnan' option.

```
B = cumsum(A,'omitnan')
```

```
B = 1×6

     3     8     8    17    17    17
```

Input Arguments

A — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

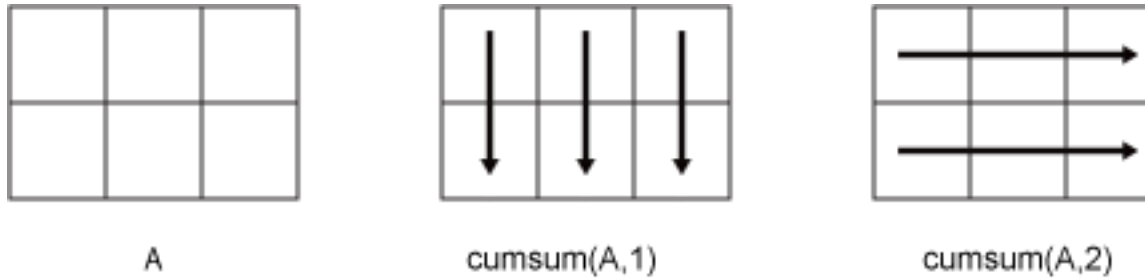
Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | duration
Complex Number Support: Yes

dim — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a two-dimensional input array, A:

- `cumsum(A,1)` works on successive elements in the columns of `A` and returns the cumulative sums of each column.
- `cumsum(A,2)` works on successive elements in the rows of `A` and returns the cumulative sums of each row.



`cumsum` returns `A` if `dim` is greater than `ndims(A)`.

`direction` — Direction of cumulation
`'forward'` (default) | `'reverse'`

Direction of cumulation, specified as `'forward'` (default) or `'reverse'`.

- `'forward'` works from `1` to `end` of the active dimension.
- `'reverse'` works from `end` to `1` of the active dimension.

Data Types: `char`

`nanflag` — `NaN` condition
`'includenan'` (default) | `'omitnan'`

`NaN` condition, specified as one of the following values:

- `'includenan'` — Include `NaN` values from the input when computing the cumulative sums, resulting in `NaN` values in the output.
- `'omitnan'` — Ignore all `NaN` values in the input. The sum of elements containing `NaN` values is the sum of all non-`NaN` elements. If all elements are `NaN`, then `cumsum` returns 0.

Data Types: `char`

Output Arguments

B — Cumulative sum array
vector | matrix | multidimensional array

Cumulative sum array, returned as a vector, matrix, or multidimensional array of the same size as the input array A.

The class of B is the same as the class of A except if A is `logical`, in which case B is `double`.

More About

First Nonsingleton Dimension

The first nonsingleton dimension is the first dimension of an array whose size is not equal to 1.

For example:

- If X is a 1-by-n row vector, then the second dimension is the first nonsingleton dimension of X.
- If X is a 1-by-0-by-n empty array, then the second dimension is the first nonsingleton dimension of X.
- If X is a 1-by-1-by-3 array, then the third dimension is the first nonsingleton dimension of X.

Tips

- Many cumulative functions in MATLAB support the `'reverse'` option. This option allows quick directional calculations without needing a flip or reflection of the input array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

The `'reverse'` direction is not supported.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Logical inputs are not supported. Cast input to `double` first.
- Code generation does not support sparse matrix inputs for this function.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Logical inputs are not supported. Cast input to `double` first.
- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `nanflag` argument is not supported.

- The order of the additions within the `cumsum` operation is not defined. Therefore, the `cumsum` operation on a `gpuArray` might not return exactly the same answer as the `cumsum` operation on the corresponding MATLAB numeric array. The differences might be significant when `A` is a signed integer type.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- The order of the additions within the `cumsum` operation is not defined. Therefore, the `cumsum` operation on a distributed array might not return exactly the same answer as the `cumsum` operation on the corresponding MATLAB numeric array. The differences might be significant when `A` is a signed integer type.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cummax` | `cummin` | `cumprod` | `diff` | `movsum` | `prod` | `sum`

Introduced before R2006a

cumtrapz

Cumulative trapezoidal numerical integration

Syntax

```
Q = cumtrapz(Y)
Q = cumtrapz(X,Y)
Q = cumtrapz( ___ ,dim)
```

Description

`Q = cumtrapz(Y)` computes the approximate cumulative integral of `Y` via the trapezoidal method on page 1-15908 with unit spacing. The size of `Y` determines the dimension to integrate along:

- If `Y` is a vector, then `cumtrapz(Y)` is the cumulative integral of `Y`.
- If `Y` is a matrix, then `cumtrapz(Y)` is the cumulative integral over each column.
- If `Y` is a multidimensional array, then `cumtrapz(Y)` integrates over the first dimension whose size does not equal 1.

`Q = cumtrapz(X,Y)` integrates `Y` with respect to the coordinates or scalar spacing specified by `X`.

- If `X` is a vector of coordinates, then `length(X)` must be equal to the size of the first dimension of `Y` whose size does not equal 1.
- If `X` is a scalar spacing, then `cumtrapz(X,Y)` is equivalent to `X*cumtrapz(Y)`.

`Q = cumtrapz( ___ ,dim)` integrates along the dimension `dim` using any of the previous syntaxes. You must specify `Y`, and optionally can specify `X`. If you specify `X`, then it can be a scalar or a vector with length equal to `size(Y,dim)`. For example, if `Y` is a matrix, then `cumtrapz(X,Y,2)` cumulatively integrates each row of `Y`.

Examples

Cumulative Integral of Vector with Unit Spacing

Calculate the cumulative integral of a vector where the spacing between data points is 1.

Create a numeric vector of data.

```
Y = [1 4 9 16 25];
```

`Y` contains function values for $f(x) = x^2$ in the domain `[1 5]`.

Use cumtrapz to integrate the data with unit spacing.

```
Q = cumtrapz(Y)
```

```
Q = 1×5

         0    2.5000    9.0000   21.5000   42.0000
```

This approximate integration yields a final value of 42. In this case, the exact answer is a little less, $41\frac{1}{3}$. The `cumtrapz` function overestimates the value of the integral because *f(x)* is concave up.

Cumulatively Integrate Vector of Data with Nonunit Spacing

Calculate the cumulative integral of a vector where the spacing between data points is uniform, but not equal to 1.

Create a domain vector.

```
X = 0:pi/5:pi;
```

Calculate the sine of `X`.

```
Y = sin(X');
```

Cumulatively integrate `Y` using `cumtrapz`. When the spacing between points is constant, but not equal to 1, an alternative to creating a vector for `X` is to specify the scalar spacing value. In that case, `cumtrapz(pi/5,Y)` is the same as `pi/5*cumtrapz(Y)`.

```
Q = cumtrapz(X,Y)
```

```
Q = 6×1

         0
    0.1847
    0.6681
    1.2657
    1.7491
    1.9338
```

Cumulatively Integrate Matrix with Nonuniform Spacing

Cumulatively integrate the rows of a matrix where the data has a nonuniform spacing.

Create a vector of *x*-coordinates and a matrix of observations that take place at the irregular intervals. The rows of `Y` represent velocity data, taken at the times contained in `X`, for three different trials.

```
X = [1 2.5 7 10];
Y = [5.2   7.7   9.6   13.2;
     4.8   7.0  10.5   14.5;
     4.9   6.5  10.2   13.8];
```

Use `cumtrapz` to integrate each row independently and find the cumulative distance traveled in each trial. Since the data is not evaluated at constant intervals, specify `X` to indicate the spacing between the data points. Specify `dim = 2` since the data is in the rows of `Y`.

```
Q1 = cumtrapz(X,Y,2)

Q1 = 3×4

         0    9.6750   48.6000   82.8000
         0    8.8500   48.2250   85.7250
         0    8.5500   46.1250   82.1250
```

The result is a matrix of the same size as `Y` with the cumulative integral of each row.

Multiple Cumulative Integrations

Perform nested integrations in the x and y directions. Plot the results to visualize the cumulative integral value in both directions.

Create a grid of values for the domain.

```
x = -2:0.1:2;
y = -2:0.2:2;
[X,Y] = meshgrid(x,y);
```

Calculate the function $f(x, y) = 10x^2 + 20y^2$ on the grid.

```
F = 10*X.^2 + 20*Y.^2;
```

`cumtrapz` integrates numeric data rather than functional expressions, so in general the underlying function does not need to be known to use `cumtrapz` on a matrix of data. In cases where the functional expression is known, you can instead use `integral`, `integral2`, or `integral3`.

Use `cumtrapz` to approximate the double integral

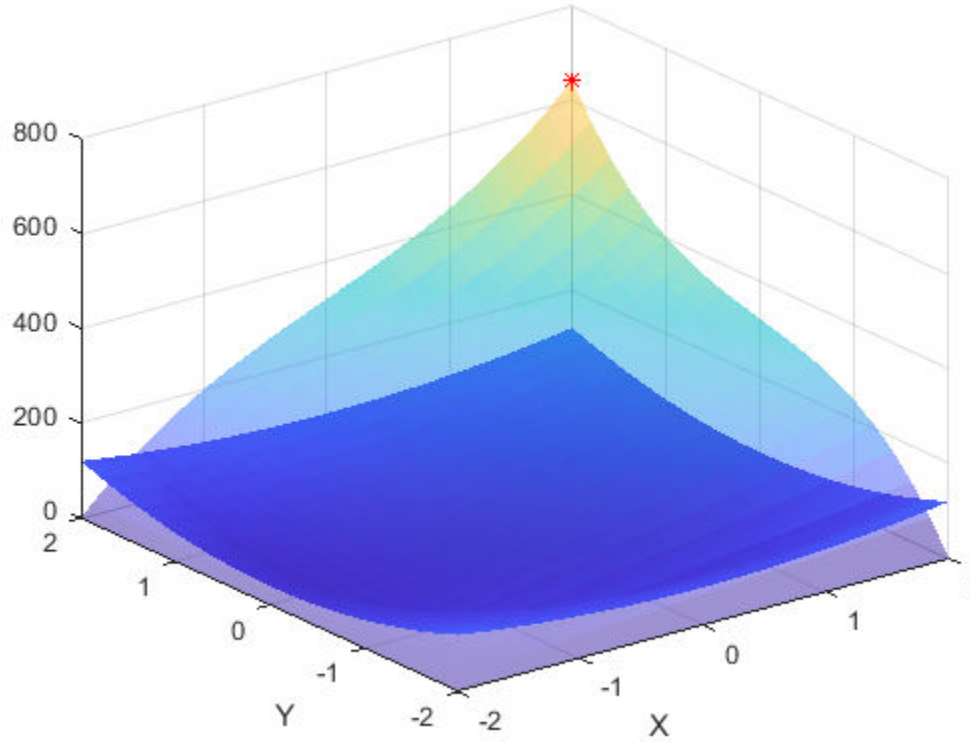
$$I(a, b) = \int_{-2}^{b}\int_{-2}^{a}\left(10x^2 + 20y^2\right)dx\,dy\,.$$

To perform this double integration, use nested function calls to `cumtrapz`. The inner call first integrates the rows of data, then the outer call integrates the columns.

```
I = cumtrapz(y,cumtrapz(x,F,2));
```

Plot the surface representing the original function as well as the surface representing the cumulative integration. Each point on the surface of the cumulative integration gives an intermediate value of the double integral. The last value in `I` gives the overall approximation of the double integral, `I(end) = 642.4`. Mark this point in the plot with a red star.

```
surf(X,Y,F,'EdgeColor','none')
xlabel('X')
ylabel('Y')
hold on
surf(X,Y,I,'FaceAlpha',0.5,'EdgeColor','none')
plot3(X(end),Y(end),I(end),'r*')
hold off
```



Input Arguments

Y — Numeric data
vector | matrix | multidimensional array

Numeric data, specified as a vector, matrix, or multidimensional array. By default, `cumtrapz` integrates along the first dimension of `Y` whose size does not equal 1.

Data Types: `single` | `double`
Complex Number Support: Yes

X — Point spacing
1 (default) | uniform scalar spacing | vector of coordinates

Point spacing, specified as 1 (default), a uniform scalar spacing, or a vector of coordinates.

- If X is a scalar, then it specifies a uniform spacing between the data points and `cumtrapz(X,Y)` is equivalent to `X*cumtrapz(Y)`.
- If X is a vector, then it specifies *x*-coordinates for the data points and `length(X)` must be the same as the size of the integration dimension in Y.

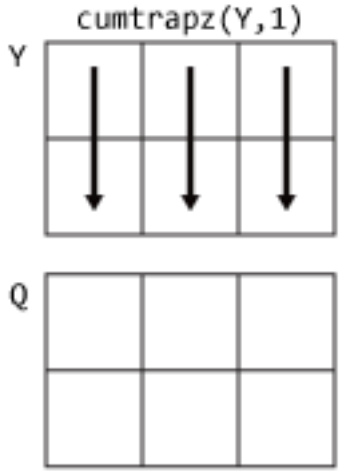
Data Types: `single` | `double`

`dim` — Dimension to operate along
positive integer scalar

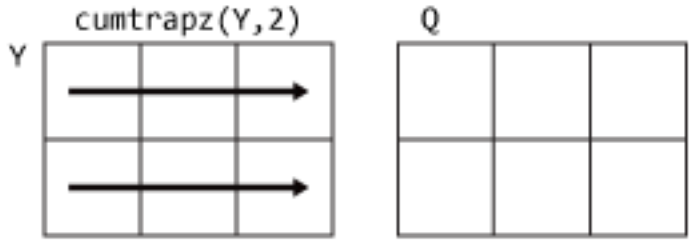
Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a two-dimensional input array, Y:

- `cumtrapz(Y,1)` works on successive elements in the columns of Y.



- `cumtrapz(Y,2)` works on successive elements in the rows of Y.



If `dim` is greater than `ndims(Y)`, then `cumtrapz` returns an array of zeros of the same size as `Y`.

Tips

- Use `trapz` and `cumtrapz` to perform numerical integrations on discrete data sets. Use `integral`, `integral2`, or `integral3` instead if a functional expression for the data is available.
- `trapz` reduces the size of the dimension it operates on to 1, and returns only the final integration value. `cumtrapz` also returns the intermediate integration values, preserving the size of the dimension it operates on.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`cumprod` | `cumsum` | `trapz`

Topics

"Integration of Numeric Data"

Introduced before R2006a

curl

Compute curl and angular velocity of vector field

Syntax

```
[curlx,curly,curlz,cav] = curl(X,Y,Z,U,V,W)
[curlx,curly,curlz,cav] = curl(U,V,W)
[curlz,cav]= curl(X,Y,U,V)
[curlz,cav]= curl(U,V)
[curlx,curly,curlz] = curl(...)
[curlx,curly] = curl(...)
cav = curl(...)
```

Description

`[curlx,curly,curlz,cav] = curl(X,Y,Z,U,V,W)` computes the curl (`curlx`, `curly`, `curlz`) and angular velocity (`cav`) perpendicular to the flow (in radians per time unit) of a 3-D vector field `U`, `V`, `W`.

The arrays `X`, `Y`, and `Z`, which define the coordinates for `U`, `V`, and `W`, must be monotonic, but do not need to be uniformly spaced. `X`, `Y`, and `Z` must have the same number of elements, as if produced by `meshgrid`.

`[curlx,curly,curlz,cav] = curl(U,V,W)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`[curlz,cav]= curl(X,Y,U,V)` computes the curl `z`-component and the angular velocity perpendicular to `z` (in radians per time unit) of a 2-D vector field `U`, and `V`.

The arrays `X` and `Y`, which define the coordinates for `U` and `V`, must be monotonic, but do not need to be uniformly spaced. `X` and `Y` must have the same number of elements, as if produced by `meshgrid`.

`[curlz,cav]= curl(U,V)` assumes `X` and `Y` are determined by the expression

```
[X Y] = meshgrid(1:n,1:m)
```

where `[m,n] = size(U)`.

`[curlx,curly,curlz] = curl(...)`, `[curlx,curly] = curl(...)` returns only the curl.

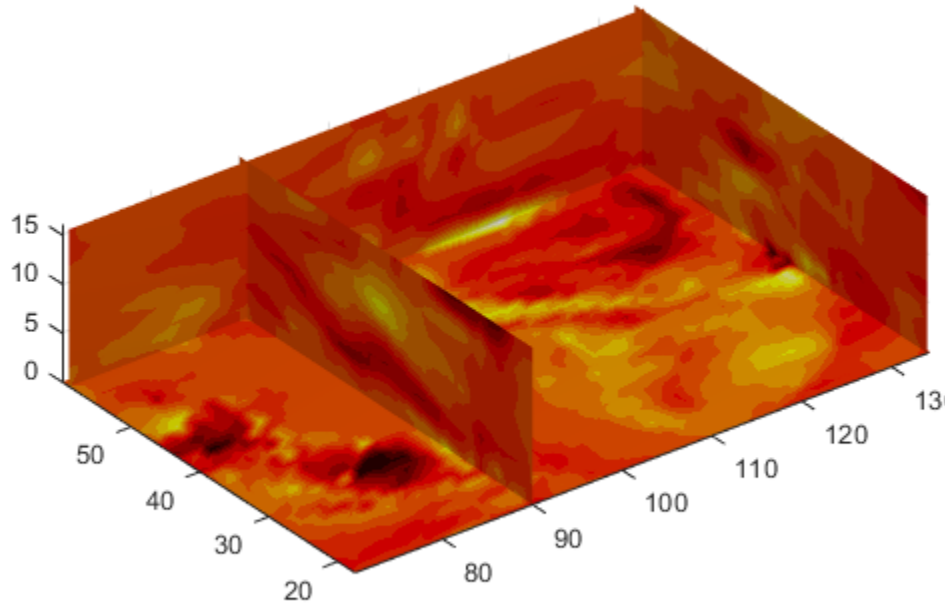
`cav = curl(...)` returns only the curl angular velocity.

Examples

Display Curl Angular Velocity Using Colored Slice Planes

Use colored slice planes to display the curl angular velocity at specified locations in the vector field.

```
load wind
cav = curl(x,y,z,u,v,w);
h = slice(x,y,z,cav,[90 134],59,0);
shading interp
daspect([1 1 1]);
axis tight
colormap(hot(16));
camlight
set([h(1),h(2)],'ambientstrength',.6);
```

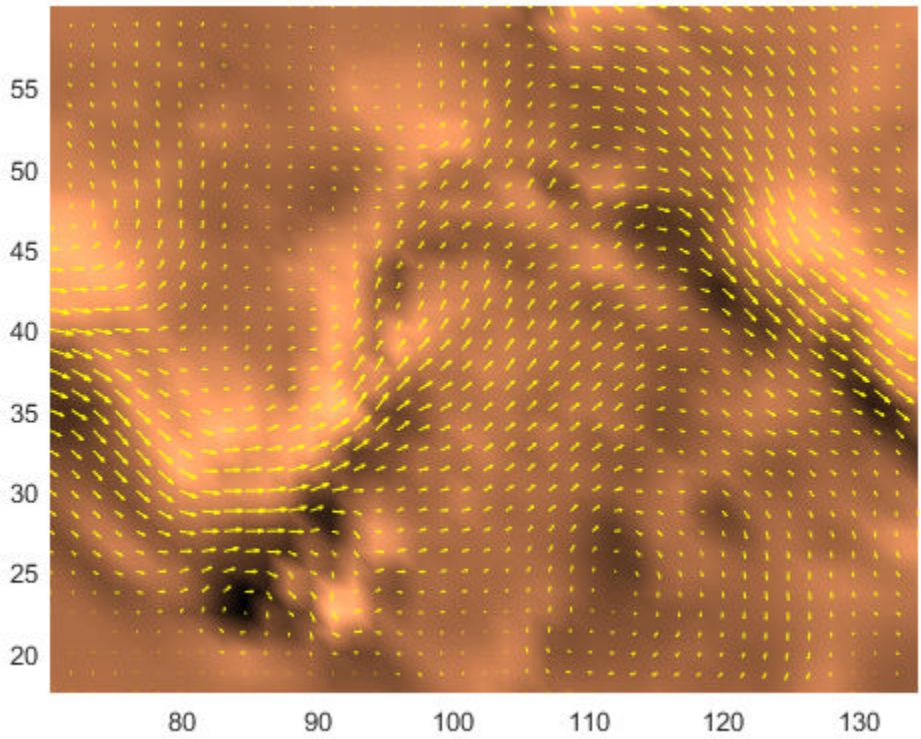


Display Curl Angular Velocity in One Plane

Display the curl angular velocity in one plane of the volume. Then plot the velocity vectors in the same plane.

```
load wind
k = 4;
x = x(:,:,k);
y = y(:,:,k);
u = u(:,:,k);
v = v(:,:,k);
```

```
cav = curl(x,y,u,v);
pcolor(x,y,cav);
shading interp
hold on
quiver(x,y,u,v,'y');
hold off
colormap('copper');
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`divergence` | `streamribbon`

Topics

“Displaying Curl with Stream Ribbons”

Introduced before R2006a

currentDirectory

Return index of current IFD

Syntax

```
dirNum = currentDirectory(t)
```

Description

`dirNum = currentDirectory(t)` returns the index of the current image file directory (IFD). Index values are one-based.

Examples

Determine Current IFD

Create a `Tiff` object for a TIFF file and determine which IFD is the current IFD.

```
t = Tiff('example.tif','r');
dnum = currentDirectory(t)
```

```
dnum = 1
```

Close the `Tiff` object.

```
close(t);
```

Input Arguments

t — Tiff object
`Tiff` object

`Tiff` object representing a TIFF file. Use the `Tiff` function to create the object.

Algorithms

References

This function corresponds to the `TIFFCurrentDirectory` function in the LibTIFF C API. To use this function, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff` | `setDirectory`

Topics

"Importing Images"
"Exporting to Images"

Introduced in R2009b

currentProject

Package: `matlab.project`

Get current project

Syntax

```
proj = currentProject
```

Description

`proj = currentProject` gets the currently open project, and returns a project object `proj` that you can use to manipulate the project programmatically. When the currently open project is running a shortcut, startup file, or shutdown file for a referenced project, `currentProject` returns the object for the referenced project instead. If no project is open, MATLAB displays an error.

For more information about working with projects programmatically, see “Create and Edit Projects Programmatically”

Examples

Get Example Project

Open the Times Table App example project and get a project object you can use to manipulate the project at the command line.

```
matlab.project.example.timesTable
proj = currentProject
```

```
proj 

  Project with properties:

                       Name: "Times Table App"
    SourceControlIntegration: "Git"
```

```
   RepositoryLocation: "C:\myProjects\examples\repositories\TimesTableApp"
SourceControlMessages: [1×3 string]
             ReadOnly: 0
             TopLevel: 1
         Dependencies: [1×1 digraph]
           Categories: [1×1 matlab.project.Category]
                Files: [1×14 matlab.project.ProjectFile]
            Shortcuts: [1×4 matlab.project.Shortcut]
          ProjectPath: [1×3 matlab.project.PathFolder]
    ProjectReferences: [1×0 matlab.project.ProjectReference]
         StartupFiles: [1×0 string]
        ShutdownFiles: [1×0 string]
          Description: "This example project contains the source code and tests for a ...
           RootFolder: "C:\myProjects\examples\TimesTableApp"
 ProjectStartupFolder: "C:\myProjects\examples\TimesTableApp"
```

Output Arguments

`proj` — Project
`matlab.project.Project` object

Project, returned as a `matlab.project.Project` object. Use the `matlab.project.Project` to programmatically manipulate the currently open project.

See Also

`openProject`

Topics

"Create and Edit Projects Programmatically"

Introduced in R2019a

customverctrl

(Has been removed) Allow custom source control system (UNIX platforms)

Note `customverctrl` has been removed.

Syntax

```
customverctrl
```

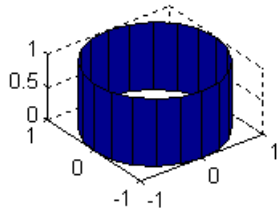
Description

`customverctrl` function is for customers who want to integrate a source control system that is not supported for use with MATLAB software. When using this function, conform to the structure of one of the supported version control systems, for example, RCS. For examples, see the files `clearcase.m`, `cvs.m`, `pvcs.m`, and `rcs.m` in *`matlabroot`* `\toolbox\matlab\verctrl`.

Introduced before R2006a

cylinder

Generate cylinder



Syntax

```
[X,Y,Z] = cylinder
[X,Y,Z] = cylinder(r)
[X,Y,Z] = cylinder(r,n)
cylinder(axes_handle,...)
cylinder(...)
```

Description

`cylinder` generates *x*-, *y*-, and *z*-coordinates of a unit cylinder. You can draw the cylindrical object using `surf` or `mesh`, or draw it immediately by not providing output arguments.

`[X,Y,Z] = cylinder` returns the *x*-, *y*-, and *z*-coordinates of a cylinder with a radius equal to `1`. The cylinder has 20 equally spaced points around its circumference.

`[X,Y,Z] = cylinder(r)` returns the *x*-, *y*-, and *z*-coordinates of a cylinder using `r` to define a profile curve. `cylinder` treats each element in `r` as a radius at equally spaced heights along the unit height of the cylinder. The cylinder has 20 equally spaced points around its circumference.

`[X,Y,Z] = cylinder(r,n)` returns the *x*-, *y*-, and *z*-coordinates of a cylinder based on the profile curve defined by vector `r`. The cylinder has `n` equally spaced points around its circumference.

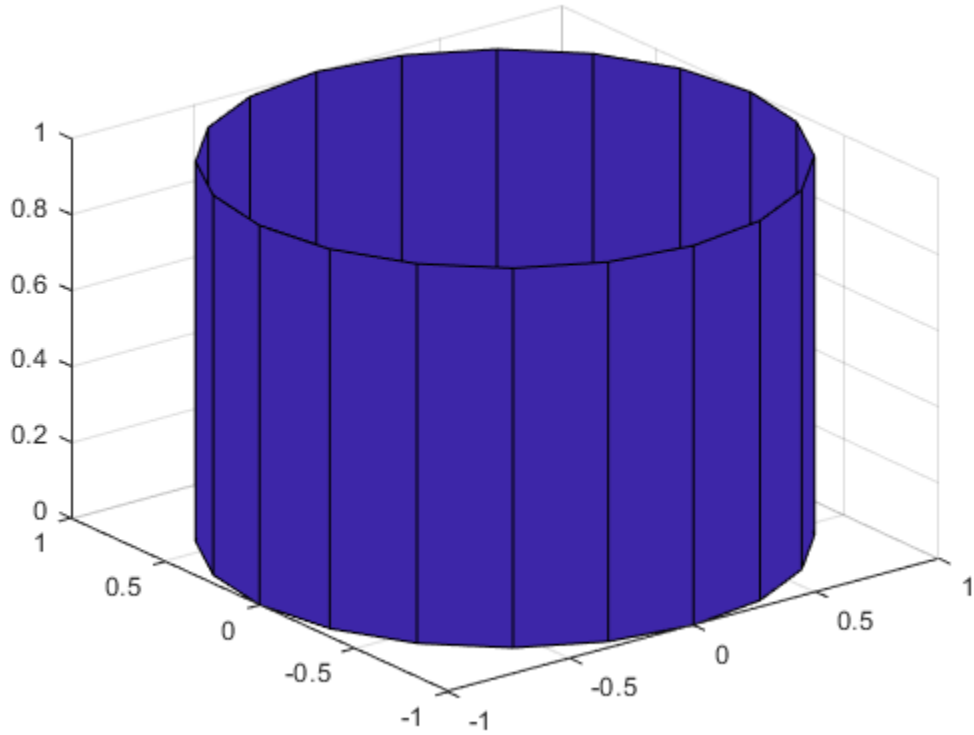
`cylinder(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`cylinder(...)`, with no output arguments, plots the cylinder using `surf`.

Examples

Display Unit Cylinder

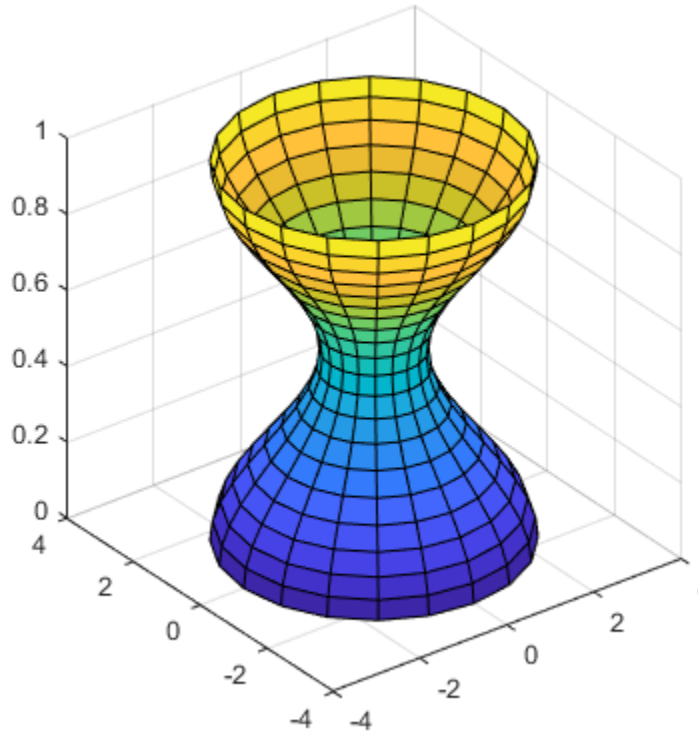
```
figure
cylinder
```



Generate Coordinates of Cylinder and Display Surface

Generate a cylinder defined by the profile function `2+sin(t)`.

```
t = 0:pi/10:2*pi;
figure
[X,Y,Z] = cylinder(2+cos(t));
surf(X,Y,Z)
axis square
```



Tips

`cylinder` treats its first argument as a profile curve. The resulting surface graphics object is generated by rotating the curve about the *x*-axis, and then aligning it with the *z*-axis.

See Also

`sphere` | `surf`

Introduced before R2006a

daqread

Read Data Acquisition Toolbox (.daq) file

Syntax

```
data = daqread('filename')
[data,time] = daqread(...)
[data,time,abstime] = daqread(...)
[data,time,abstime,events] = daqread(...)
[data,time,abstime,events,daqinfo] = daqread(...)
data = daqread(...,'Param1', Val1,...)
daqinfo = daqread('filename','info')
```

Description

`data = daqread('filename')` reads all the data from the Data Acquisition Toolbox™ (`.daq`) file specified by `filename`. `daqread` returns `data`, an *m*-by-*n* data matrix, where *m* is the number of samples and *n* is the number of channels. If `data` includes data from multiple triggers, the data from each trigger is separated by a `NaN`. If you set the `OutputFormat` property to `tscollection`, `daqread` returns a time series collection object. See below for more information.

`[data,time] = daqread(...)` returns time/value pairs. `time` is an *m*-by-1 vector, the same length as `data`, that contains the relative time for each sample. Relative time is measured with respect to the first trigger that occurs.

`[data,time,abstime] = daqread(...)` returns the absolute time of the first trigger. `abstime` is returned as a `clock` vector.

`[data,time,abstime,events] = daqread(...)` returns a log of events. `events` is a structure containing event information. If you specify either the`Samples`, `Time`, or `Triggers` parameters (see below), the events structure contains only the specified events.

`[data,time,abstime,events,daqinfo] = daqread(...)` returns a structure, `daqinfo`, that contains two fields: `ObjInfo` and `HwInfo`. `ObjInfo` is a structure

containing property name/property value pairs and `HwInfo` is a structure containing hardware information. The entire event log is returned to `daqinfo.ObjInfo.EventLog`.

`data = daqread(...,'Param1', Val1,...)` specifies the amount of data returned and the format of the data, using the following parameters.

Parameter	Description
`Samples`	Specify the sample range.
`Time`	Specify the relative time range.
`Triggers`	Specify the trigger range.
`Channels`	Specify the channel range. Channel names can be specified as a cell array.
`DataFormat`	Specify the data format as `doubles` (default) or `native`.
`TimeFormat`	Specify the time format as `vector` (default) or `matrix`.
`OutputFormat`	Specify the output format as `matrix` (the default) or `tscollection`. When you specify `tscollection`, `daqread` only returns `data`.

The `Samples`, `Time`, and `Triggers` properties are mutually exclusive; that is, either `Samples`, `Triggers` or `Time` can be defined at once.

`daqinfo = daqread('filename','info')` returns metadata from the file in the `daqinfo` structure, without incurring the overhead of reading the data from the file as well. The `daqinfo` structure contains two fields:

`daqinfo.ObjInfo`

a structure containing parameter/value pairs for the data acquisition object used to create the file, `filename`. Note: The `UserData` property value is not restored.

`daqinfo.HwInfo`

a structure containing hardware information. The entire event log is returned to `daqinfo.ObjInfo.EventLog`.

Examples

Use Data Acquisition Toolbox to acquire data. The analog input object, `ai`, acquires one second of data for four channels, and saves the data to the output file `data.daq`.

```
ai = analoginput('nidaq','Dev1');
chans = addchannel(ai,0:3);
set(ai,'SampleRate',1000)
ActualRate = get(ai,'SampleRate');
set(ai,'SamplesPerTrigger, ActualRate)
set(ai,'LoggingMode','Disk&Memory')
set(ai,'LogFileName','data.daq')
start(ai)
```

After the data has been collected and saved to a disk file, you can retrieve the data and other acquisition-related information using `daqread`. To read all the sample-time pairs from `data.daq`:

```
[data,time] = daqread('data.daq');
```

To read samples 500 to 1000 for all channels from `data.daq`:

```
data = daqread('data.daq','Samples',[500 1000]);
```

To read only samples 1000 to 2000 of channel indices 2, 4 and 7 in native format from the file, `data.daq`:

```
data = daqread('data.daq', 'Samples', [1000 2000],...
               'Channels', [2 4 7], 'DataFormat', 'native');
```

To read only the data which represents the first and second triggers on all channels from the file, `data.daq`:

```
[data,time] = daqread('data.daq', 'Triggers', [1 2]);
```

To obtain the channel property information from `data.daq`:

```
daqinfo = daqread('data.daq','info');
chaninfo = daqinfo.ObjInfo.Channel;
```

To obtain a list of event types and event data contained by `data.daq`:

```
daqinfo = daqread('data.daq','info');
events = daqinfo.ObjInfo.EventLog;
event_type = {events.Type};
event_data = {events.Data};
```

To read all the data from the file `data.daq` and return it as a time series collection object:

```
data = daqread('data.daq','OutputFormat','tscollection');
```

Tips

More About .daq Files

- The format used by `daqread` to return data, relative time, absolute time, and event information is identical to the format used by the `getdata` function that is part of Data Acquisition Toolbox. For more information, see the Data Acquisition Toolbox documentation.
- If data from multiple triggers is read, then the size of the resulting data array is increased by the number of triggers issued because each trigger is separated by a `NaN`.
- `ObjInfo.EventLog` always contains the entire event log regardless of the value specified by `Samples`, `Time`, or `Triggers`.
- The `UserData` property value is not restored when you return device object (`ObjInfo`) information.
- When reading a `.daq` file, the `daqread` function does not return property values that were specified as a cell array.
- Data Acquisition Toolbox (`.daq`) files are created by specifying a value for the `LogFileName` property (or accepting the default value), and configuring the `LoggingMode` property to `Disk` or `Disk&Memory`.

More About Time Series Collection Object Returned

When `OutputFormat` is set to `tscollection`, `daqread` returns a time series collection object. This times series collection object contains an absolute time series object for each channel in the file. The following describes how `daqread` sets some of the properties of the times series collection object and the time series objects.

- The `time` property of the time series collection object is set to the value of the `InitialTriggerTime` property specified in the file.
- The `name` property of each time series object is set to the value of the `Name` property of a channel in the file. If this name cannot be used as a time series object name, `daqread` sets the name to `'Channel'` with the `HwChannel` property of the channel appended.

- The value of the `Units` property of the time series object depends on the value of the `DataFormat` parameter. If the `DataFormat` parameter is set to `'double'`, `daqread` sets the `DataInfo` property of each time series object in the collection to the value of the `Units` property of the corresponding channel in the file. If the `DataFormat` parameter is set to `'native'`, `daqread` sets the `Units` property to `'native'`. See the Data Acquisition Toolbox documentation for more information on these properties.
- Each time series object will have `tsdata.event` objects attached corresponding to the log of events associated with the channel.

If `daqread` returns data from multiple triggers, the data from each trigger is separated by a `NaN` in the time series data. This increases the length of data and time vectors in the time series object by the number of triggers.

See Also

`timeseries` | `tscollection`

daspect

Control data unit length along each axis

Syntax

```
daspect(ratio)
d = daspect

daspect auto
daspect manual
m = daspect('mode')

___ = daspect(ax,___)
```

Description

`daspect(ratio)` sets the data aspect ratio for the current axes. The data aspect ratio is the relative length of the data units along the *x*-axis, *y*-axis, and *z*-axis. Specify `ratio` as a three-element vector of positive values that represent the relative lengths of data units along each axis. For example, `[1 2 3]` indicates that the length from 0 to 1 along the *x*-axis is equal to the length from 0 to 2 along the *y*-axis and 0 to 3 along the *z*-axis. For equal data unit lengths in all directions, use `[1 1 1]`.

`d = daspect` returns the data aspect ratio for the current axes.

`daspect auto` sets an automatic mode, enabling the axes to choose the data aspect ratio. The mode must be automatic to enable the “stretch-to-fill on page 1-2713” feature of the axes.

`daspect manual` sets a manual mode and uses the ratio stored in the `DataAspectRatio` property for the `Axes` object. When the mode is manual, it disables the stretch-to-fill behavior of the axes. Specifying a value for the data aspect ratio sets the mode to manual.

`m = daspect('mode')` returns the current mode, which is either `'auto'` or `'manual'`. By default, the mode is automatic unless you specify the data aspect ratio or set the mode to manual.

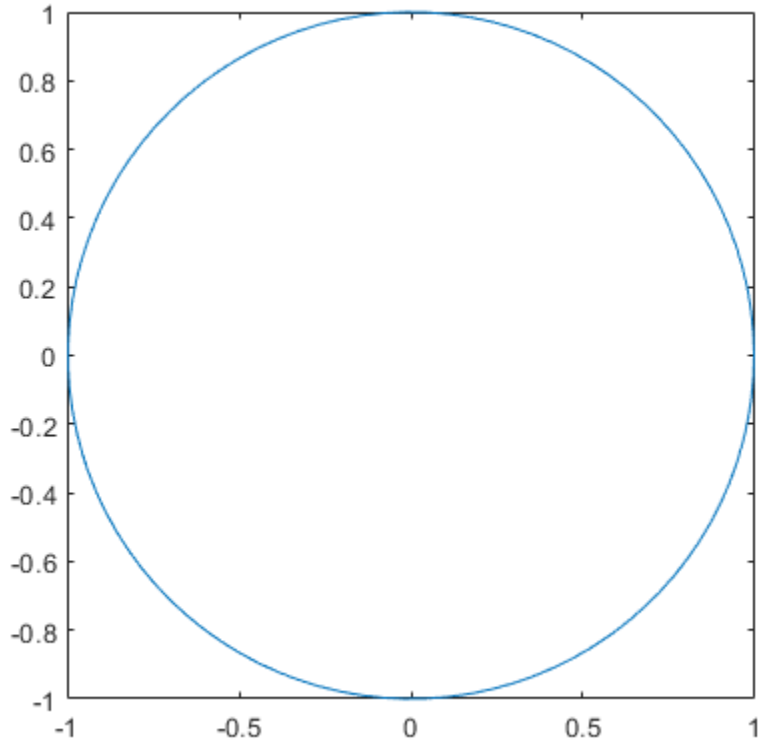
`___ = daspect(ax, ___)` uses the axes specified by `ax` instead of the current axes. Specify the axes as the first input argument for any of the previous syntaxes. Use single quotes around the `'auto'` and `'manual'` inputs.

Examples

Use Equal Data Unit Lengths Along Each Axis

Plot a set of data. Use data units of equal lengths in all directions by setting the data aspect ratio to `[1 1 1]`.

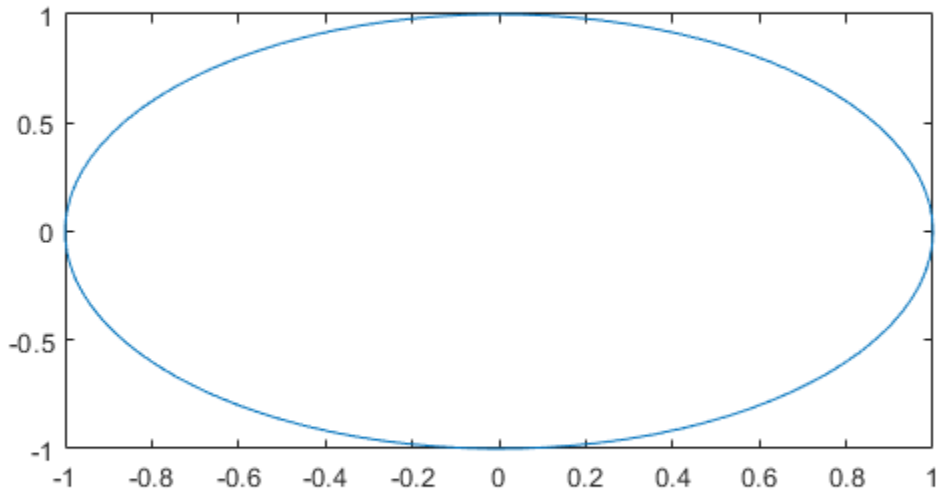
```
t = linspace(0,2*pi);
x = cos(t);
y = sin(t);
plot(x,y)
daspect([1 1 1])
```



Use Different Data Unit Lengths Along Each Axis

Plot a set of data. Set the data aspect ratio so that the length of one data unit in the *x*-axis direction equals the length of two data units in the *y*-axis direction and *z*-axis direction (not shown).

```
t = linspace(0,2*pi);
x = cos(t);
y = sin(t);
plot(x,y)
daspect([1 2 2])
```



When you set the data aspect ratio, the associated mode changes to manual. Query the current data aspect ratio mode.

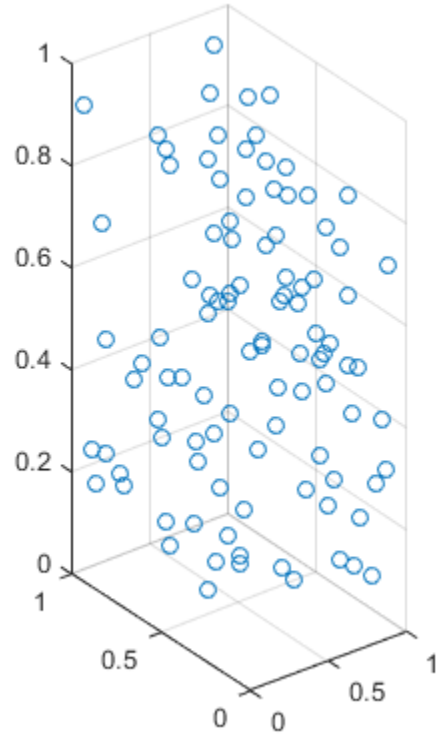
```
d = daspect('mode')
```

```
d =
'manual'
```

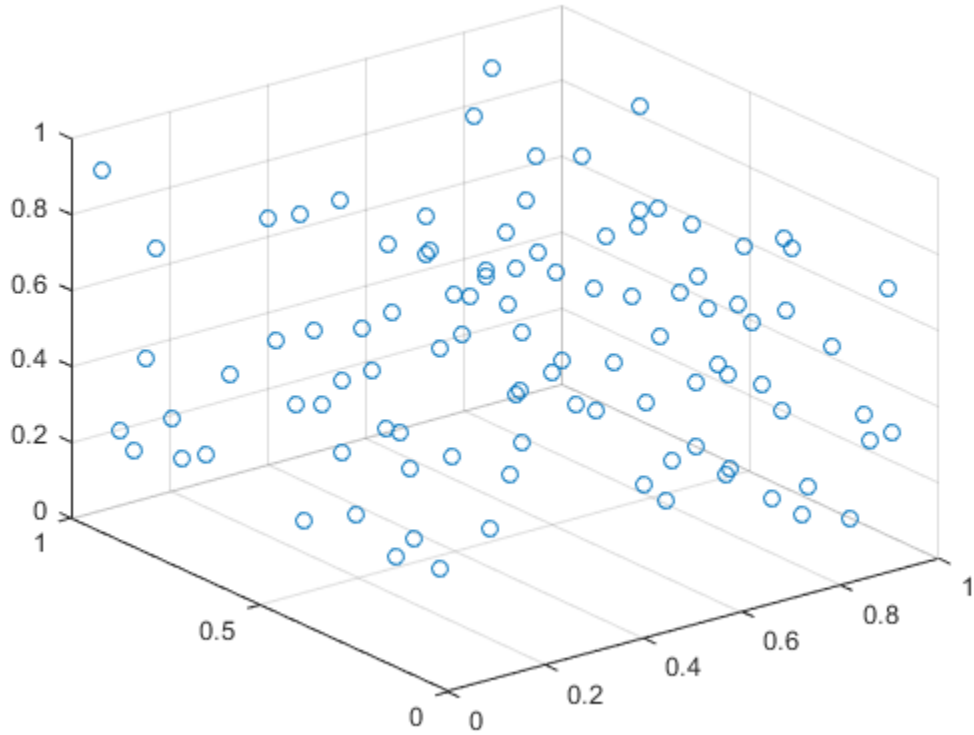
Revert Back to Default Data Aspect Ratio

Create a 3-D scatter chart of random data and set the data aspect ratio. Then revert back to the default data aspect ratio.

```
X = rand(100,1);
Y = rand(100,1);
Z = rand(100,1);
scatter3(X,Y,Z)
daspect([3 2 1])
```



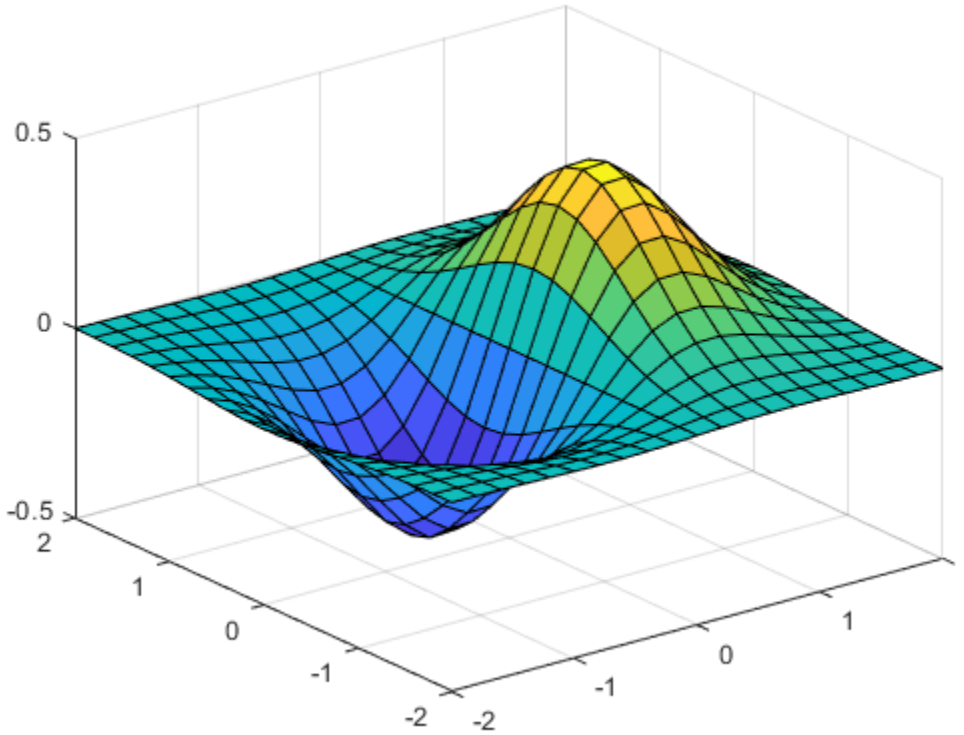

```
daspect auto
```



Query Data Aspect Ratio

Create a surface plot and query the data aspect ratio.

```
[x,y] = meshgrid(-2:.2:2);
z = x.*exp(-x.^2 - y.^2);
surf(x,y,z)
```




```
d = daspect
```

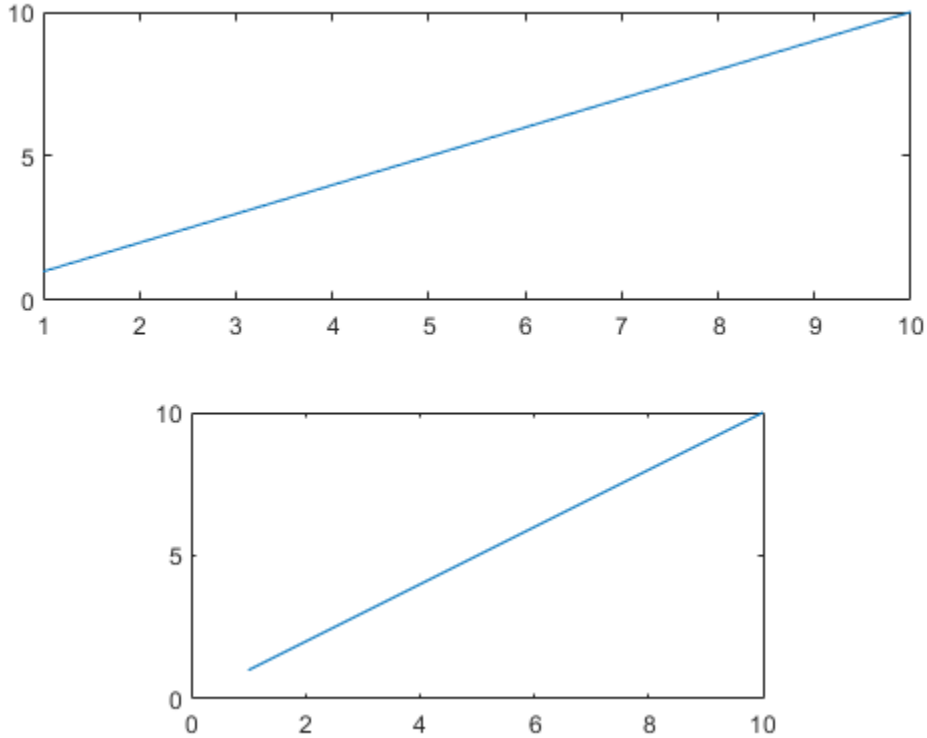
```
d = 1×3

     4     4     1
```

The ratio indicates that the length of four units in the *x*-axis direction equals four units in the *y*-axis direction and one unit in the *z*-axis direction. For example, if the ratio is `[4 4 1]`, then the length from 0 to 1 on the *x*-axis equals the length from 0 to 1 on the *y*-axis and 0 to 0.25 on the *z*-axis.

Set Data Aspect Ratio for Specific Axes Object

Create a figure with two subplots and return the axes objects from the `subplot` function. Set the data aspect ratio for the lower subplot by specifying `ax2` as the first input argument to the `daspect` function.

```
ax1 = subplot(2,1,1);
plot(1:10)

ax2 = subplot(2,1,2);
plot(1:10)
daspect(ax2,[1 2 1])
```



Input Arguments

`ratio` — Data aspect ratio
three-element vector

Data aspect ratio, specified as a three-element vector of positive values. For example, `daspect([1 2 3])` specifies that the length of one unit along the *x*-axis equals the length of two units along the *y*-axis and three units along the *z*-axis.

Specifying the data aspect ratio sets the `DataAspectRatio` property for the `Axes` object. The `DataAspectRatio` property interacts with the `PlotBoxAspectRatio`, `XLim`, `YLim`, and `ZLim` properties to control the length and scale of the *x*-axis, *y*-axis, and *z*-axis.

Specifying the ratio sets the `DataAspectRatioMode` property to `'manual'` and disables the stretch-to-fill on page 1-2713 behavior of the axes.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ax` — Target axes
`Axes` object

Target axes, specified as an `Axes` object. If you do not specify the axes, then `daspect` uses the current axes (`gca`).

Output Arguments

`d` — Current data aspect ratio values
three-element vector

Current data aspect ratio, returned as a three-element vector of values representing the ratio of the *x*-axis, *y*-axis, and *z* data unit lengths.

Querying the data aspect ratio returns the `DataAspectRatio` property value for the corresponding `Axes` object.

`m` — Current data aspect ratio mode
`'auto'` | `'manual'`

Current data aspect ratio mode, returned as either `'auto'` or `'manual'`. When the mode is automatic, MATLAB determines the appropriate data aspect ratio value. If you specify a value, then the mode changes to manual.

Querying the data aspect ratio mode returns the `DataAspectRatioMode` property value for the corresponding `Axes` object.

More About

Stretch-to-Fill

When the “stretch-to-fill” behavior is enabled, MATLAB stretches the axes to fill the available space. The axes might not exactly match the data aspect ratio, plot box aspect ratio, and camera-view angle values stored in its `DataAspectRatio`, `PlotBoxAspectRatio`, and `CameraViewAngle` properties.

If you specify the data aspect ratio, plot box aspect ratio, or camera-view angle, then the “stretch-to-fill” behavior is disabled. When the behavior is disabled, MATLAB makes the axes as large as possible within the available space and strictly adheres to the property values. There is no distortion. For more information, see “Control Axes Layout”.

Algorithms

The `daspect` function sets and queries several axes properties related to the data aspect ratio.

- `DataAspectRatio` — Property that stores the data aspect ratio value.
- `DataAspectRatioMode` — Property that stores the data aspect ratio mode. When you set the ratio, this property changes to `'manual'`.

See Also

`axis` | `pbaspect` | `xlim` | `ylim` | `zlim`

Topics

“Control Ratio of Axis Lengths and Data Unit Lengths”

"Control Axes Layout"

Introduced before R2006a

datacursormode

Enable, disable, and manage interactive data cursor mode

Syntax

```
datacursormode on
datacursormode off
datacursormode
datacursormode toggle
datacursormode(figure_handle)
dcm_obj = datacursormode(figure_handle)
```

Description

`datacursormode on` enables data cursor mode on the current figure.

`datacursormode off` disables data cursor mode on the current figure. Starting in R2018b, some data cursor interactions are enabled by default, regardless of the mode. If you want to disable these default interactions, then use the `disableDefaultInteractivity` function.

`datacursormode` or `datacursormode toggle` toggles data cursor mode in the current figure.

`datacursormode(figure_handle)` enables or disables data cursor mode on the specified figure.

`dcm_obj = datacursormode(figure_handle)` returns the data cursor mode object for the figure. The object enables you to customize the data cursor. For more information on data cursor mode objects, see “Output Arguments” on page 1-2716. You cannot change the state of data cursor mode in a call to `datacursormode` that returns a mode object.

A data cursor is a small black square with a white border that you interactively position on a graph in data cursor mode. When you click a graphic object such as a line on a graph, a data tip appears. Data tips are small text boxes or windows that float within an axes that display data values at data cursor locations. The default style is a text box. Data

tips list *x*-, *y*- and (where appropriate) *z*-values for one data point at a time. See “Examples” on page 1-2719 for an illustration of these two styles.

Input Arguments

`figure_handle`

Optional handle of figure window

Default: The current figure

`state`

`''`, `'toggle'`, `'on'`, or `'off'`

Default: `'toggle'`

Output Arguments

`dcm_obj`

Use the object returned by `datacursormode` to control aspects of data cursor behavior. You can use the `set` and `get` commands to set and query object property values. You can customize how data cursor mode presents information by coding callback functions for these objects.

Parameter Name/Value Pairs for Data Cursor Mode Objects

The following parameters apply to objects returned by calls to `datacursormode`, not to the function itself.

`DisplayStyle`

`datatip | window`

Determines how the data cursor displays.

- `datatip` displays data cursor information in a small yellow text box attached to a black square marker at a data point you interactively select.

- `window` displays data cursor information for the data point you interactively select in a floating window within the figure.

Default: `datatip`

`Enable`

`on | off`

Specifies whether data cursor mode is currently enabled for the figure.

Default: `off`

`Figure`

handle

Handle of the figure associated with the data cursor mode object.

`Interpreter`

`none | tex | latex`

Specifies the interpretation of text characters. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text. For more information, see `Interpreter`.

Default: `tex`

`SnapToDataVertex`

`on | off`

Specifies whether the data cursor snaps to the nearest data value or is located at the actual pointer position.

Default: `on`

`UpdateFcn`

function handle

Reference to a function that formats the text appearing in the data cursor. You can supply your own function to customize data tip display. Your function must include at least two

arguments. The first argument is unused, and can be a variable name or tilde (~). The second argument passes the data cursor event object to your update function. The event object encapsulates the state of the data cursor. The following function definition illustrates the update function:

```
function output_txt = myfunction(~,event_obj)
% ~            Currently not used (empty)
% event_obj    Object containing event data structure
% output_txt   Data cursor text
```

`event_obj` is an object that has the following properties.

`Target`	Handle of the object the data cursor is referencing (the object which you click, for example, a line or a bar from a series)
`Position`	An array specifying the *x*, *y* (and *z* for 3-D graphs) coordinates of the cursor

You can query these properties within your function. For example,

```
pos = get(event_obj,'Position');
```

returns the coordinates of the cursor. Another way of accessing that data is to obtain the struct and query its `Position` field:

```
eventdata = get(event_obj);
pos = eventdata.Position;
```

You can also obtain the position directly from the object:

```
pos = event_obj.Position;
```

You can redefine the data cursor `Updatefcn` at run time. For example:

```
set(dcm_obj,'UpdateFcn',@myupdatefcn)
```

applies the function `myupdatefcn` to the current data tip or tips. When you set an update function in this way, the function must be on the MATLAB path. If instead you select the data cursor mode context menu item **Select text update function**, you can interactively select a function that is not on the path.

Do not redefine figure window callbacks, such as `ButtonDownFcn`, `KeyPressFcn`, or `CloseRequestFcn` *while in data cursor mode*. If you attempt to change any *figure* callbacks when you are in an interactive mode, you receive a warning and the attempt fails. MATLAB interactive modes are:

- `brush`
- `datacursormode`
- `pan`
- `rotate3d`
- `zoom`

This restriction does not apply to changing the figure `WindowButtonMotionFcn` callback or `uicontrol` callbacks.

Querying Data Cursor Mode

Use the `getCursorInfo` function to query the data cursor mode object (`dcm_obj` in the update function syntax) to obtain information about the data cursor. For example,

```
info_struct = getCursorInfo(dcm_obj);
```

returns a vector of structures, one for each data cursor on the graph. Each structure has the following fields.

`Target`	The handle of the graphics object containing the data point
`Position`	An array specifying the *x*, *y*, (and *z*) coordinates of the cursor

Line and lineseries objects have an additional field.

`DataIndex`	A scalar index into the data arrays that correspond to the nearest data point. The value is the same for each array.

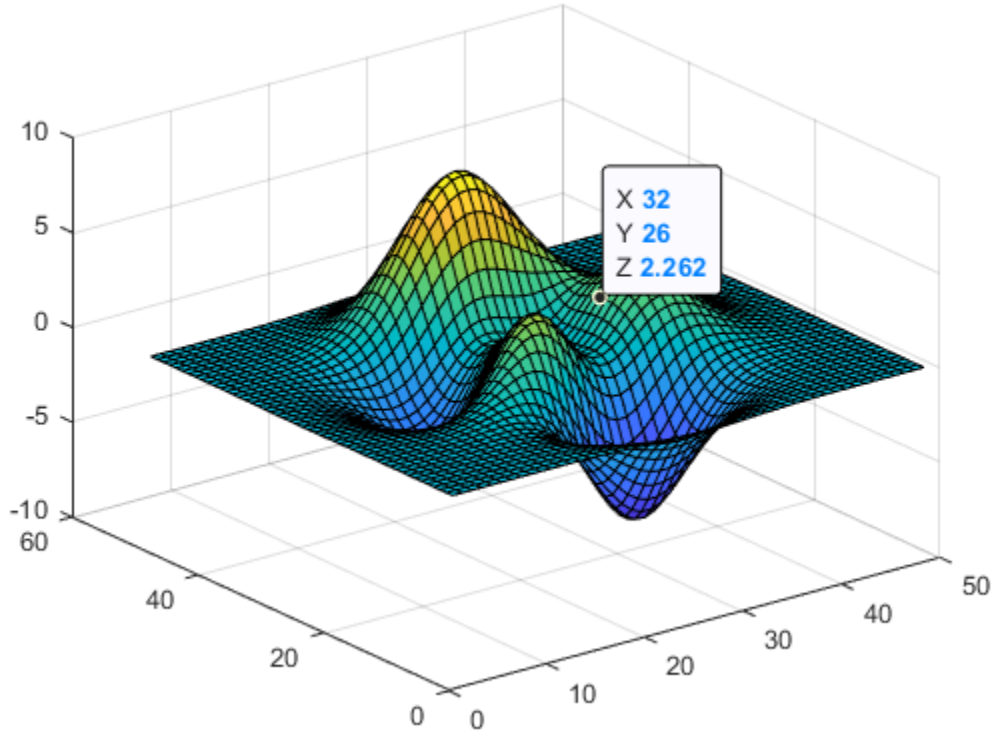
See “Output Arguments” on page 1-2716 for more details on data cursor mode objects.

Examples

This example creates a plot and enables data cursor mode from the command line.

```
surf(peaks)
datacursormode on
% Click mouse on surface to display data cursor
```

Selecting a point on the surface opens a data tip displaying its *x*-, *y*-, and z-coordinates.



You change the data tip display style to be a window instead of a text box using the **Tools > Options > Display cursor in window** , or use the context menu **Display Style > Window inside figure** to view the data tip in a floating window that you can move around inside the axes.

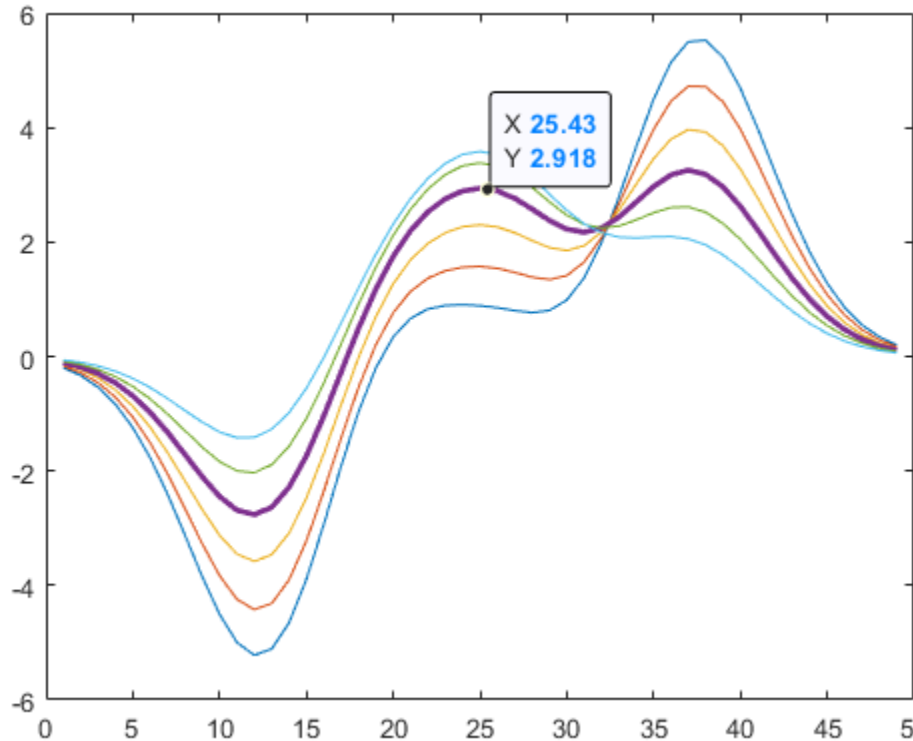
You can position multiple text box data tips on the same graph, the window style of data tip displays only one value at a time.

This example enables data cursor mode on the current figure and sets data cursor mode options. The following statements

- Create a graph
- Toggle data cursor mode to on

- Obtain the data cursor mode object, specify data tip options, and get the handle of the line the data tip occupies:

```
fig = figure;
z = peaks;
plot(z(:,30:35))
dcm_obj = datacursormode(fig);
set(dcm_obj,'DisplayStyle','datatip',...
    'SnapToDataVertex','off','Enable','on')

disp('Click line to display a data tip, then press Return.')
% Wait while the user does this.
pause

c_info = getCursorInfo(dcm_obj);
% Make selected line wider
set(c_info.Target,'LineWidth',2)
```



This example shows you how to customize the text that the data cursor displays. For example, you can replace the text displayed in the data tip and data window (`x:` and `y:`) with `Time:` and `Amplitude:` by creating a simple update function.

Save the following functions in your current directory or any writable directory on the MATLAB path before running them. As they are functions, you cannot highlight them and then evaluate the selection to make them work.

Save this code as `doc_datacursormode.m`:

```
function doc_datacursormode
% Plots graph and sets up a custom data tip update function
fig = figure;
a = -16; t = 0:60;
plot(t,sin(a*t))
```

```
dcm_obj = datacursormode(fig);
set(dcm_obj,'UpdateFcn',@myupdatefcn)
```

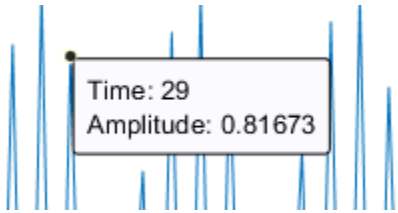
Save the following code as `myupdatefcn.m` on the MATLAB path:

```
function txt = myupdatefcn(empt,event_obj)
% Customizes text of data tips

pos = get(event_obj,'Position');
txt = {['Time: ',num2str(pos(1))],...
          ['Amplitude: ',num2str(pos(2))]};
```

To set up and use the update function, type:

```
doc_datacursormode
```

When you place a data tip using this update function, it looks like the one in the following figure.



Tips

- You can create data tips on most types of graphs and 3-D plots. However, some types of plots do not support data cursor mode, such as pie charts created with `pie`.
- In general, data tips show the coordinates of the selected point. However, for some types of charts, data tips display specialized information. For example, histograms created with `histogram` display data tips that itemize the observation counts and bin edges.
- When the `DisplayStyle` property of the data cursor mode object is `'datatip'`, you can create multiple data tips by holding **Shift** as you select data points. When the `DisplayStyle` is `'window'`, you can create only one data tip at a time.
- Data tips on a plot persist if you turn off data cursor mode when the `DisplayStyle` property is `'datatip'`. However, the data cursor disappears and the data tip window closes when the `DisplayStyle` is `'window'`.

Alternatives

Use the Data Cursor button in the toolbar to label *x*, *y*, and *z* values on graphs and surfaces. You can control how data tips display by right-clicking and selecting items from the context menu.

See Also

`brush` | `datatip` | `pan` | `rotate3d` | `zoom`

Topics

"Interactively Explore Plotted Data"

Introduced before R2006a

datastore

Create datastore for large collections of data

Syntax

```
ds = datastore(location)
ds = datastore(location,Name,Value)
```

Description

`ds = datastore(location)` creates a datastore from the collection of data specified by `location`. A datastore is a repository for collections of data that are too large to fit in memory. After creating `ds`, you can read and process the data.

`ds = datastore(location,Name,Value)` specifies additional parameters for `ds` using one or more name-value pair arguments. For example, you can create a datastore for image files by specifying `'Type','image'`.

Examples

Create Datastore for Text Data

Create a datastore associated with the sample file `airlinesmall.csv`. This file contains airline data from the years 1987 through 2008.

To manage the import of missing data in numeric columns, use the `'TreatAsMissing'` name-value pair argument. In this example, specifying the value `'NA'` for `'TreatAsMissing'`, replaces every instance of `'NA'` with a `NaN` in the imported data. Where, `NaN` is the value specified in the `'MissingValue'` property of the datastore.

```
ds = datastore('airlinesmall.csv', ...
                   'TreatAsMissing','NA')
```

```
ds 
  TabularTextDatastore with properties:
```

```
                      Files: {
                             'B:\matlab\toolbox\matlab\demos\airlinesmall.csv'
                             }
                FileEncoding: 'UTF-8'
    AlternateFileSystemRoots: {}
       PreserveVariableNames: false
           ReadVariableNames: true
               VariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
              DatetimeLocale: en_US

  Text Format Properties:
              NumHeaderLines: 0
                   Delimiter: ','
                RowDelimiter: '\r\n'
              TreatAsMissing: 'NA'
                MissingValue: NaN

  Advanced Text Format Properties:
            TextscanFormats: {'%f', '%f', '%f' ... and 26 more}
                   TextType: 'char'
         ExponentCharacters: 'eEdD'
                CommentStyle: ''
                  Whitespace: ' \b\t'
    MultipleDelimitersAsOne: false

  Properties that control the table returned by preview, read, readall:
      SelectedVariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
            SelectedFormats: {'%f', '%f', '%f' ... and 26 more}
                   ReadSize: 20000 rows
```

`datastore` creates a `TabularTextDatastore`.

Create Datastore for Image Data

Create a datastore containing all `.tif` files in the MATLAB® path and its subfolders.

```
ds = datastore(fullfile(matlabroot, 'toolbox', 'matlab'),...
'IncludeSubfolders', true,'FileExtensions', '.tif','Type', 'image')
```

```
ds =
```

```
  ImageDatastore with properties:

       Files: {
              ' ...\matlab\toolbox\matlab\demos\example.tif';
              ' ...\matlab\toolbox\matlab\imagesci\corn.tif'
              }
    ReadSize: 1
      Labels: {}
     ReadFcn: @readDatastoreImage
```

Input Arguments

`location` — Files or folders to include in the datastore
path | `DsFileSet` object

Files or folders included in the datastore, specified as a path or a `DsFileSet` object.

- path — Specify the path as a character vector, cell array of character vectors, string scalar, or a string array, containing the location of files or folders that are local or remote.
  - Local files or folders — Specify `location` as a local path to files or folders. If the files are not in the current folder, then local path must specify full or relative paths. Files within subfolders of the specified folder are not automatically included in the datastore. You can use the wildcard character (*) when specifying the local path. This character specifies that the datastore include all matching files or all files in the matching folders.
  - Remote files or folders — Specify `location` to be the full paths of the files or folders as a uniform resource locator (URL) of the form `hdfs:///path_to_file`. For more information, see “Work with Remote Data”.
- `DsFileSet` object — You also can specify `location` as a `DsFileSet` object. For more information, see `matlab.io.datastore.DsFileSet`.

When `location` represents a folder, the datastore includes only supported file formats and ignores any other format. To specify a custom list of file extensions to include in your datastore, see the `FileExtensions` property.

For `KeyValueDatastore`, the files must be MAT-files or Sequence files generated by the `mapreduce` function. MAT-files must be in a local file system or in a network file system.

Sequence files can be in a local, network, or HDFS™ file system. For `DatabaseDatastore`, the location argument need not be files. For more information, see `DatabaseDatastore`.

Example: `'file1.csv'`

Example: `'../dir/data/file1.jpg'`

Example: `{'C:\dir\data\file1.xls','C:\dir\data\file2.xlsx'}`

Example: `'C:\dir\data\*.mat'`

Example: `'hdfs:///data/file1.txt'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'FileExtensions',{'.jpg','.tif'}` includes all extensions with a `.jpg` or `.tif` extension for an `ImageDatastore` object.

Type — Type of datastore
`'tabulartext'` | `'image'` | `'spreadsheet'` | `'keyvalue'` | `'file'` | `'tall'` | ...

Type of datastore, specified as the comma-separated pair consisting of `'Type'` and one of the following:

Value of 'Type'	Description
`'tabulartext'`	Text files containing tabular data. The encoding of the data must be ASCII or UTF-8.
`'image'`	Image files in a format such as JPEG or PNG. Acceptable files include `imformats` formats.
`'spreadsheet'`	Spreadsheet files containing one or more sheets.

Value of 'Type'	Description
'keyvalue'	Key-value pair data contained in MAT-files or Sequence files with data generated by mapreduce.
'file'	Custom format files, which require a specified read function to read the data. For more information, see FileDatastore.
'tall'	MAT-files or Sequence files produced by the write function of the tall data type. For more information see, TallDatastore.
'parquet'	Parquet files containing column-oriented data. For more information see, ParquetDatastore.
'database'	Data stored in database. Requires Database Toolbox™. Requires specification of additional input argument when using the type parameter. For more information, see DatabaseDatastore.

- If there are multiple types that support the format of the files, then use the 'Type' argument to specify a datastore type.
- If you do not specify a value for 'Type', then datastore automatically determines the appropriate type of datastore to create based on the extensions of the files.

Data Types: char | string

IncludeSubfolders — Include subfolders within folder
true or false | 0 or 1

Include subfolders within a folder, specified as the comma-separated pair consisting of 'IncludeSubfolders' and true (1) or false (0). Specify true to include all files and subfolders within each folder or false to include only the files within each folder.

When you do not specify 'IncludeSubfolders', then the default value is false.

The 'IncludeSubfolders' name-value pair is only valid when creating these objects:

- TabularTextDatastore

- `ImageDatastore`
- `SpreadsheetDatastore`
- `FileDatastore`
- `KeyValueDatastore`
- `ParquetDatastore`

Example: `'IncludeSubfolders',true`

Data Types: `logical` | `double`

`FileExtensions` — Extensions of files
character vector | cell array of character vectors | string scalar | string array

Extensions of files, specified as the comma-separated pair consisting of `'FileExtensions'` and a character vector, cell array of character vectors, string scalar, or string array. When specifying `'FileExtensions'`, also specify `'Type'`. You can use the empty quotes `''` to represent files without extensions.

If `'FileExtensions'` is not specified, then `datastore` automatically includes all supported file extensions depending on the datastore type. If you want to include unsupported extensions, then specify each extension you want to include individually.

- For `TabularTextDatastore` objects, supported extensions include `.txt`, `.csv`, `.dat`, `.dlm`, `.asc`, `.text`, and no extension.
- For `ImageDatastore` objects, supported extensions include all `imformats` extensions.
- For `SpreadsheetDatastore` objects, supported extensions include `.xls`, `.xlsx`, `.xlsm`, `.xltx`, and `.xltm`.
- For `TallDatastore` objects, supported extensions include `.mat` and `.seq`.
- For `ParquetDatastore` objects, supported extensions include `.parquet` and `.parq`.

The `'FileExtensions'` name-value pair is only valid when creating these objects:

- `TabularTextDatastore`
- `ImageDatastore`
- `SpreadsheetDatastore`
- `FileDatastore`
- `KeyValueDatastore`

- `ParquetDatastore`

Example: `'FileExtensions','.jpg'`

Example: `'FileExtensions',{'.txt','.text'}`

Data Types: `char` | `cell` | `string`

`AlternateFileSystemRoots` — Alternate file system root paths
string vector | cell array

Alternate file system root paths, specified as the comma-separated pair consisting of `'AlternateFileSystemRoots'` and a string vector or a cell array. Use `'AlternateFileSystemRoots'` when you create a datastore on a local machine, but need to access and process the data on another machine (possibly of a different operating system). Also, when processing data using the Parallel Computing Toolbox and the MATLAB Parallel Server™, and the data is stored on your local machines with a copy of the data available on different platform cloud or cluster machines, you must use `'AlternateFileSystemRoots'` to associate the root paths.

- To associate a set of root paths that are equivalent to one another, specify `'AlternateFileSystemRoots'` as a string vector. For example,

  ```
  ["Z:\datasets","/mynetwork/datasets"]
  ```

- To associate multiple sets of root paths that are equivalent for the datastore, specify `'AlternateFileSystemRoots'` as a cell array containing multiple rows where each row represents a set of equivalent root paths. Specify each row in the cell array as either a string vector or a cell array of character vectors. For example:

  - Specify `'AlternateFileSystemRoots'` as a cell array of string vectors.

    ```
    {["Z:\datasets", "/mynetwork/datasets"];...
     ["Y:\datasets", "/mynetwork2/datasets","S:\datasets"]}
    ```

  - Alternatively, specify `'AlternateFileSystemRoots'` as a cell array of cell array of character vectors.

    ```
    {{'Z:\datasets','/mynetwork/datasets'};...
     {'Y:\datasets', '/mynetwork2/datasets','S:\datasets'}}
    ```

The value of `'AlternateFileSystemRoots'` must satisfy these conditions:

- Contains one or more rows, where each row specifies a set of equivalent root paths.

- Each row specifies multiple root paths and each root path must contain at least two characters.
- Root paths are unique and are not subfolders of one another.
- Contains at least one root path entry that points to the location of the files.

For more information, see “Set Up Datastore for Processing on Different Machines or Clusters”.

Example: `["Z:\datasets","/mynetwork/datasets"]`

Data Types: `string` | `cell`

`TextType` — Output data type of text variables
`'char'` (default) | `'string'`

Output data type of text variables, specified as the comma-separated pair consisting of `'TextType'` and either `'char'` or `'string'`. If the output table from the `read`, `readall`, or `preview` functions contains text variables, then `'TextType'` specifies the data type of those variables for `TabularTextDatastore` and `SpreadsheetDatastore` objects only. If `'TextType'` is `'char'`, then the output is a cell array of character vectors. If `'TextType'` is `'string'`, then the output has type `string`.

Data Types: `char` | `string`

`DatetimeType` — Type for imported date and time data
`'datetime'` (default) | `'text'`

Type for imported date and time data, specified as the comma-separated pair consisting of `'DatetimeType'` and one of these values: `'datetime'` or `'text'`. The `'DatetimeType'` argument only applies when creating a `TabularTextDatastore` object.

Value	Type for Imported Date and Time Data
`'datetime'`	MATLAB `datetime` data type For more information, see `datetime`.

Value	Type for Imported Date and Time Data
`'text'`	If `'DatetimeType'` is specified as `'text'`, then the type for imported date and time data depends on the value specified in the `'TextType'` parameter: • If `'TextType'` is `'char'`, then the `datastore` returns dates as a cell array of character vectors. • If `'TextType'` is `'string'`, then the `datastore` returns dates as an array of strings.

Example: `'DatetimeType','datetime'`

Data Types: `char` | `string`

`DurationType` — Output data type of duration data
`'duration'` (default) | `'text'`

Output data type of duration data from text files, specified as the comma-separated pair consisting of `'DurationType'` and either `'duration'` or `'text'`.

Value	Type for Imported Duration Data
`'duration'`	MATLAB `duration` data type For more information, see `duration`.
`'text'`	If `'DurationType'` is specified as `'text'`, then the type for imported duration data depends on the value specified in the `'TextType'` parameter: • If `'TextType'` is `'char'`, then the importing function returns duration data as a cell array of character vectors. • If `'TextType'` is `'string'`, then the importing function returns duration data as an array of strings.

Data Types: `char` | `string`

`PreserveVariableNames` — Flag to preserve variable names
`false` (default) | `true`

Flag to preserve variable names, specified as the comma-separated pair consisting of `PreserveVariableNames` and either `true`, or `false`.

- `true` — Preserve variable names that are not valid MATLAB identifiers such as variable names that include spaces and non-ASCII characters.
- `false` — Convert invalid variable names (as determined by the `isvarname` function) to valid MATLAB identifiers.

Starting in R2019b, variable names and row names can include any characters, including spaces and non-ASCII characters. Also, they can start with any characters, not just letters. Variable and row names do not have to be valid MATLAB identifiers (as determined by the `isvarname` function). To preserve these variable names and row names, set `PreserveVariableNames` to `true`.

In addition to these name-value pairs, you also can specify any of the properties of the following objects as name-value pairs, except for the `Files` property:

- `TabularTextDatastore`
- `ImageDatastore`
- `SpreadsheetDatastore`
- `KeyValueDatastore`
- `TallDatastore`
- `DatabaseDatastore`

Output Arguments

ds — Datastore for collection of data
`TabularTextDatastore` | `ImageDatastore` | `SpreadsheetDatastore` | `KeyValueDatastore` | `FileDatastore` | `TallDatastore` | ...

Datastore for a collection of data, returned as one of these objects: `TabularTextDatastore`, `ImageDatastore`, `SpreadsheetDatastore`, `KeyValueDatastore`, `FileDatastore`, `TallDatastore`, `ParquetDatastore`, or `DatabaseDatastore`. The type of the datastore depends on the type of files or the `location` argument. For more information, click the datastore name in the following table:

Type	Output
Text files	`TabularTextDatastore`

Type	Output
Image files	`ImageDatastore`
Spreadsheet files	`SpreadsheetDatastore`
MAT-files or Sequence files produced by `mapreduce`	`KeyValueDatastore`
Custom format files	`FileDatastore`
MAT-files or Sequence files produced by the `write` function of the `tall` data type.	`TallDatastore`
Parquet Files	`ParquetDatastore`
Database	`DatabaseDatastore`

For each of these datastore types, the `Files` property is a cell array of character vectors. Each character vector is an absolute path to a file resolved by the `location` argument.

See Also

`CombinedDatastore` | `TransformedDatastore` | `fileDatastore` | `imageDatastore` | `imformats` | `javaaddpath` | `mapreduce` | `parquetDatastore` | `spreadsheetDatastore` | `tabularTextDatastore` | `tall`

Topics

"Getting Started with Datastore"
"Work with Remote Data"
"Set Up Datastore for Processing on Different Machines or Clusters"

Introduced in R2014b

combine

Package: `matlab.io`

Combine data from multiple datastores

Syntax

```
dsnew = combine(ds1,ds2,...,dsN)
```

Description

`dsnew = combine(ds1,ds2,...,dsN)` combines two or more datastores by horizontally concatenating the data returned by the `read` function on the input datastores.

Examples

Combine Datastores

Create a datastore that maintains parity between the pair of images of the underlying datastores. For instance, create two separate image datastores, and then create a combined datastore representing the two underlying datastores.

Create a datastore `imds1` representing a collection of three images.

```
imds1 = imageDatastore({'street1.jpg','street2.jpg','peppers.png'});
```

Create a second datastore `imds2` by transforming the images of `imds1` to grayscale images.

```
imds2 = transform(imds1,@(x) rgb2gray(x));
```

Create a combined datastore from `imds1` and `imds2`.

```
imdsCombined = combine(imds1,imds2);
```

Read the first pair of images from the combined datastore. Each read operation on this combined datastore returns a pair of images represented by a 1-by-2 cell array.

```
dataOut = read(imdsCombined)
```

```
dataOut=1×2 cell
    {480x640x3 uint8}    {480x640 uint8}
```

Display the output of the first read from the combined datastore.

```
imshow(imtile(dataOut));
```

Input Arguments

`ds1,ds2,...,dsN` — Datastores to combine
datastore objects

Datastores to combine, specified as two or more comma separated datastore objects. For a complete list of built-in datastores, see “Select Datastore for File Format or Application”. You can also specify custom datastores.

Output Arguments

dsnew — New datastore with combined data
`CombinedDatastore` object

New datastore with combined data, returned as a `CombinedDatastore` object.

The `dsnew` object is a new datastore instance that is the horizontally concatenated result of a read operation on each of the underlying datastores.

See Also

`CombinedDatastore` | `hasdata` | `preview` | `read` | `readall` | `reset` | `transform`

Topics

"Select Datastore for File Format or Application"

Introduced in R2019a

CombinedDatastore

Datastore to combine data read from multiple underlying datastores

Description

Use a `CombinedDatastore` object to combine the data read from multiple other datastores.

A `CombinedDatastore` maintains parity between the underlying datastores, so that data is read from corresponding parts of the underlying datastores. Resetting the `CombinedDatastore` resets all of the underlying datastores.

Creation

You can create a `CombinedDatastore` object using the `combine` function. For example, `dsnew = combine(ds1,ds2)` creates a datastore that combines the read data from datastores `ds1` and `ds2`.

Properties

`UnderlyingDatastores` — Underlying datastores
cell array

Underlying datastores, specified as a cell array of datastore objects.

Data Types: `cell`

Object Functions

combine	Combine data from multiple datastores
hasdata	Determine if data is available to read
preview	Subset of data in datastore
read	Read data in datastore
readall	Read all data in datastore

reset	Reset datastore to initial state
transform	Transform datastore

Examples

Combine Datastores

Create a datastore that maintains parity between the pair of images of the underlying datastores. For instance, create two separate image datastores, and then create a combined datastore representing the two underlying datastores.

Create a datastore `imds1` representing a collection of three images.

```
imds1 = imageDatastore({'street1.jpg','street2.jpg','peppers.png'});
```

Create a second datastore `imds2` by transforming the images of `imds1` to grayscale images.

```
imds2 = transform(imds1,@(x) rgb2gray(x));
```

Create a combined datastore from `imds1` and `imds2`.

```
imdsCombined = combine(imds1,imds2);
```

Read the first pair of images from the combined datastore. Each read operation on this combined datastore returns a pair of images represented by a 1-by-2 cell array.

```
dataOut = read(imdsCombined)
```

```
dataOut=1×2 cell
    {480x640x3 uint8}    {480x640 uint8}
```

Display the output of the first read from the combined datastore.

```
imshow(imtile(dataOut));
```

See Also

`TransformedDatastore` | `datastore`

Topics

"Getting Started with Datastore"

Introduced in R2019a

counteachlabel

Package: `matlab.io.datastore`

Count files in ImageDatastore labels

Syntax

```
T = countEachLabel(imds)
```

Description

`T = countEachLabel(imds)` returns a summary table of the labels in `imds` and the number of files associated with each.

Examples

Label Count

Create an `ImageDatastore` object and label each image according to the folder it is in.

```
imds = imageDatastore(fullfile(matlabroot, 'toolbox', 'matlab', {'demos','imagesci'}),
'LabelSource', 'foldernames', 'FileExtensions', {'.jpg', '.png', '.tif'})
```

```
imds = 

  ImageDatastore with properties:

      Files: {
             ' ...\matlab\toolbox\matlab\demos\cloudCombined.jpg';
             ' ...\matlab\toolbox\matlab\demos\example.tif';
             ' ...\matlab\toolbox\matlab\demos\landOcean.jpg'
              ... and 5 more
             }
     Labels: [demos; demos; demos ... and 5 more categorical]
```

```
    ReadFcn: @readDatastoreImage
```

List the file count for each label.

```
T = countEachLabel(imds)
```

```
T = 

     Label      Count
    ________    _____

    demos       6
    imagesci    2
```

Input Arguments

imds — Input datastore
ImageDatastore object

Input datstore, specified as an ImageDatastore object. To create an ImageDatstore from your image data, use the imageDatastore function.

Output Arguments

T — Table of label counts
two-column table

Table of label counts, returned as a two-column table containing the name of each label in imds and the number of files associated with each label.

Data Types: table

See Also

ImageDatastore | datastore | splitEachLabel

Introduced in R2016a

fileDatastore

Datastore with custom file reader

Description

Use a `FileDatastore` object to manage large collections of custom format files where the collection does not necessarily fit in memory or when a large custom format file does not fit in memory. You can create a `FileDatastore` object using the `fileDatastore` function, specify its properties, and then import and process the data using object functions.

Creation

Syntax

```
fds = fileDatastore(location,'ReadFcn',@fcn)
fds = fileDatastore(location,'ReadFcn',@fcn,Name,Value)
```

Description

`fds = fileDatastore(location,'ReadFcn',@fcn)` creates a datastore from the collection of files specified by `location` and uses the function `fcn` to read the data from the files.

`fds = fileDatastore(location,'ReadFcn',@fcn,Name,Value)` specifies additional parameters and properties for `fds` using one or more name-value pair arguments. For example, you can specify which files to include in the datastore depending on their extensions with `fileDatastore(location,'ReadFcn',@customreader,'FileExtensions',{'.exts','.extx'})`.

Input Arguments

`location` — Files or folders to include in datastore

path | `DsFileSet` object

Files or folders included in the datastore, specified as a path or a `DsFileSet` object.

- path — Specify the path as a character vector, cell array of character vectors, string scalar, or a string array, containing the location of files or folders that are local or remote.
  - Local files or folders — Specify `location` as a local path to files or folders. If the files are not in the current folder, then local path must specify full or relative paths. Files within subfolders of the specified folder are not automatically included in the datastore. You can use the wildcard character (*) when specifying the local path. This character specifies that the datastore include all matching files or all files in the matching folders.
  - Remote files or folders — Specify `location` to be the full paths of the files or folders as a uniform resource locator (URL) of the form `hdfs:///path_to_file`. For more information, see “Work with Remote Data”.
- `DsFileSet` object — You also can specify `location` as a `DsFileSet` object. For more information, see `matlab.io.datastore.DsFileSet`.

When `location` represents a folder, the datastore includes only supported file formats and ignores any other format. To specify a custom list of file extensions to include in your datastore, see the `FileExtensions` property.

Example: `'file1.ext'`

Example: `'../dir/data/file1.ext'`

Example: `{'C:\dir\data\file1.exts','C:\dir\data\file2.extx'}`

Example: `'C:\dir\data\*.ext'`

`@fcn` — Function that reads file data

function handle

Function that reads the file data, specified as a function handle.

The signature of the function represented by the function handle `@fcn` depends on the value of the specified `ReadMode`. The function that reads the file data must confirm to one of these signatures.

ReadMode	ReadFcn signature

`'file'` (default)	The function must have this signature:
	`function data = MyReadFcn(filename)`
	`...`
	`end`
	`filename` — Name of file to read.
	`data` — Corresponding file data.

`'partialfile'`	The function must have this signature:
	`function [data,userdata,done] = MyReadFcn(filename,userdata)`
	`...`
	`end`
	`userdata` — Set and read fields of `userdata` to persist data between multiple `FileDatastore` read calls.
	`done` — Set this `logical` argument to either `true` or `false`.
	• `false` — Continue to read the current file.
	• `true` — Terminate current file read and read the next file.
	`data` — Portion of file data.

ReadMode	ReadFcn signature

'byte'	The function must have this signature:
	`function data = MyReadFcn(filename,offset,size)`
	`...`
	`end`
	
	`offset` — Specify the byte offset from the first byte in the file.
	
	`size` — Specify the number of bytes to read during the current read operation.
	
	`data` — Portion of file data of the size specified in `BlockSize`.
	
	The `FileDatastore` increments both the `offset` and `size` inputs based on the value specified in `BlockSize`.

The value specified in `@fcn`, sets the value of the ReadFcn property.

Example: `@customreader`

Data Types: `function_handle`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `fds = fileDatastore('C:\dir\data','FileExtensions', {'.exts','.extx'})`

`IncludeSubfolders` — Subfolder inclusion flag
`true` or `false` | 0 or 1

Subfolder inclusion flag, specified as the comma-separated pair consisting of `'IncludeSubfolders'` and `true`, `false`, 0, or 1. Specify `true` to include all files and subfolders within each folder or `false` to include only the files within each folder.

If you do not specify `'IncludeSubfolders'`, then the default value is `false`.

Example: `'IncludeSubfolders',true`

Data Types: `logical` | `double`

`FileExtensions` — Custom format file extensions
character vector | cell array of character vectors | string scalar | string array

Custom format file extensions, specified as the comma-separated pair consisting of `'FileExtensions'` and a character vector, cell array of character vectors, string scalar, or string array.

When you specify a file extension, the `fileDatastore` function creates a datastore object only for files with the specified extension. You can also create a datastore for files without any extensions by specifying `'FileExtensions'` as an empty character vector, `''`. If you do not specify `'FileExtensions'`, then `fileDatastore` automatically includes all files within a folder.

Example: `'FileExtensions',''`

Example: `'FileExtensions','.ext'`

Example: `'FileExtensions',{'.exts','.extx'}`

Data Types: `char` | `cell` | `string`

`PreviewFcn` — Function to preview input data
`@ReadFcn` (default) | function handle

Function to preview the input data, specified as a function handle.

If you do not specify a preview function, `FileDatastore` uses the value specified in `@ReadFcn` as the default preview function. Alternatively, you can specify your own custom preview function for your data.

- `@ReadFcn` (default) — Use `ReadFcn` to sample `FileDatastore` data. This option can lead to slower performance for `tall` construction.
- `Function handle` — Use your custom preview function for `FileDatastore` and `tall` construction to sample the input data. Use `PreviewFcn` to provide a function that reads only the minimum needed part of input data for preview and tall construction.

The function specified by `PreviewFcn` must return values with the same data types that the `ReadFcn` returns.

Data Types: `function_handle`

ReadMode — Portion of the file to read
`'file'` (default) | `'partialfile'` | `'bytes'`

Portion of the file to read, specified as `'file'`, `'partialfile'`, or `'bytes'`.

`'file'` (default)	Use read mode `'file'` when your custom function, specified in `ReadFcn`, reads the complete file in one read operation. Based on your custom read function, the file datastore reads the complete file with each call to `read`. The unit of parallelization to is a complete file.
`'partialfile'`	Use read mode `'partialfile'` when your custom file read function, specified in `ReadFcn`, reads only a portion of the file with each read operation. Based on your custom read function, the file datastore reads only a portion of the file with every call to the `read` function. In the `'partialfile'` read mode, the unit of parallelization is a complete file. Multiple `read` operations, in serial, are necessary to read a complete file.
`'bytes'`	Use read mode `'bytes'` when your custom function, specified in `ReadFcn`, reads a `BlockSize` sized portion of the file with each read operation. `FileDatastore` sets the unit of parallelization to a block of the file containing the number of bytes specified by BlockSize. Based on your custom read function, the file datastore reads `BlockSize` sized portions of a file with every call to the read function. Multiple `read` operations in parallel are necessary to read a complete file.

Data Types: `char` | `string`

`BlockSize` — Number of bytes to read
`positive integer`

Number of bytes to read with every `read` operation, specified as a positive integer.

To ensure that you can distribute multiple blocks of a file across multiple parallel MATLAB workers, specify `BlockSize` as a positive integer greater than `131072` bytes (`128` kilobytes).

To specify or to change the value of `BlockSize`, you must first set `ReadMode` to `'bytes'`. `FileDatastore` sets the default value of `BlockSize` based on the value specified in `ReadMode`.

- If `ReadMode` is `'file'` or `'partialfile'`, then `FileDatastore` sets the default `BlockSize` to `inf`.
- If `ReadMode` is `'bytes'`, then `FileDatastore` sets the default `BlockSize` to `128` megabytes.

`AlternateFileSystemRoots` — Alternate file system root paths
string vector | cell array

Alternate file system root paths, specified as the comma-separated pair consisting of `'AlternateFileSystemRoots'` and a string vector or a cell array. Use `'AlternateFileSystemRoots'` when you create a datastore on a local machine, but need to access and process the data on another machine (possibly of a different operating system). Also, when processing data using the Parallel Computing Toolbox and the MATLAB Parallel Server, and the data is stored on your local machines with a copy of the data available on different platform cloud or cluster machines, you must use `'AlternateFileSystemRoots'` to associate the root paths.

- To associate a set of root paths that are equivalent to one another, specify `'AlternateFileSystemRoots'` as a string vector. For example,

  ```
  ["Z:\datasets","/mynetwork/datasets"]
  ```

- To associate multiple sets of root paths that are equivalent for the datastore, specify `'AlternateFileSystemRoots'` as a cell array containing multiple rows where each row represents a set of equivalent root paths. Specify each row in the cell array as either a string vector or a cell array of character vectors. For example:

  - Specify `'AlternateFileSystemRoots'` as a cell array of string vectors.

```
{["Z:\datasets", "/mynetwork/datasets"];...
 ["Y:\datasets", "/mynetwork2/datasets","S:\datasets"]}
```

- Alternatively, specify `'AlternateFileSystemRoots'` as a cell array of cell array of character vectors.

```
{{'Z:\datasets','/mynetwork/datasets'};...
 {'Y:\datasets', '/mynetwork2/datasets','S:\datasets'}}
```

The value of `'AlternateFileSystemRoots'` must satisfy these conditions:

- Contains one or more rows, where each row specifies a set of equivalent root paths.
- Each row specifies multiple root paths and each root path must contain at least two characters.
- Root paths are unique and are not subfolders of one another.
- Contains at least one root path entry that points to the location of the files.

For more information, see “Set Up Datastore for Processing on Different Machines or Clusters”.

Example: `["Z:\datasets","/mynetwork/datasets"]`

Data Types: `string` | `cell`

Properties

`FileDatastore` properties describe the files associated with a `FileDatastore` object. Except for the `Files` property, you can specify the value of `FileDatastore` properties using name-value pair arguments. To view or modify a property after creating the object, use the dot notation.

`Files` — Files included in datastore

character vector | cell array of character vectors | string scalar | string array

Files included in the datastore, resolved as a character vector, cell array of character vectors, string scalar, or string array, where each character vector or string is a full path to a file. The `location` argument in the `fileDatastore` and `datastore` functions defines `Files` when the datastore is created.

Example: `{'C:\dir\data\file1.ext';'C:\dir\data\file2.ext'}`

Example: `'hdfs:///data/*.mat'`

Data Types: `char` | `cell` | `string`

`ReadFcn` — Function that reads file data
function handle

Function that reads the file data, specified as a function handle.

The value specified by @fcn, sets the value of the `ReadFcn` property.

Example: `@MyCustomFileReader`

Data Types: `function_handle`

`UniformRead` — Vertically concatenateable flag
`false` (default) | `true`

This property is read-only.

Vertically concatenateable flag, specified as a logical `true` or `false`. Specify the value of this property when you first create the `FileDatastore` object.

`true`	Multiple reads of the `FileDatastore` object return uniform data that is vertically concatenateable. When the `UniformRead` property value is `true`: • The `ReadFcn` function must return data that is vertically concatenateable ; otherwise, the `readall` method returns an error. • The underlying data type of the output of the `tall` function is the same as the data type of the output from `ReadFcn`.
`false` (default)	Multiple reads of the `FileDatastore` object do not return uniform data that is vertically concatenateable. When the `UniformRead` property value is `false`: • `readall` returns a cell array. • `tall` returns a tall cell array.

Example: `fds = fileDatastore(location,'ReadFcn',@load,'UniformRead',true)`

Data Types: `logical` | `double`

Object Functions

hasdata	Determine if data is available to read
numpartitions	Number of datastore partitions
partition	Partition a datastore
preview	Subset of data in datastore
read	Read data in datastore
readall	Read all data in datastore
reset	Reset datastore to initial state
transform	Transform datastore
combine	Combine data from multiple datastores

Examples

Create FileDatastore Object

Create a datastore for the files in the MATLAB® `demos` folder that have a `.mat` extension.

```
fds = fileDatastore(fullfile(matlabroot,'toolbox','matlab','demos'),'ReadFcn',@load,'F
```

Read Datastore of MAT-Files

Create a datastore containing all the `.mat` files within the MATLAB® `demos` folder, specifying the `load` function to read the file data.

```
fds = fileDatastore(fullfile(matlabroot,'toolbox','matlab','demos'),'ReadFcn',@load,'F
```

Read the first file in the datastore, and then read the second file.

```
data1 = read(fds);
data2 = read(fds);
```

Read all files in the datastore simultaneously.

```
readall(fds);
```

Initialize a cell array to hold the data and counter `i`.

```
dataarray = cell(numel(fds.Files), 1);
i = 1;
```

Reset the datastore to the first file and read the files one at a time until there is no data left. Assign the data to the array `dataarray`.

```
reset(fds);
while hasdata(fds)
    dataarray{i} = read(fds);
    i = i+1;
end
```

Read One Array at a Time From Large MAT-File

You can create a datastore to read from a large MAT-file that does not necessarily fit in memory. Assuming that each array in the large MAT-file fits in the available memory, create a datastore to read and process the data in three steps:

1. Write a custom reading function that reads one array at a time from a MAT-file.
2. Set up the parameters of the datastore function to perform partial reads.
3. Read one array at a time from the MAT-file.

Write a custom function that reads one array at time from MAT-file. The function must have a signature as described in the `@ReadFcn` argument of `FileDatastore`. Save this file in your working folder or in a folder that is on the MATLAB path. For this example, a custom function `load_variable` is included here.

```
type load_variable.m
```

```
function [data,variables,done] = load_variable(filename,variables)

    % If variable list is empty,
    % create list of variables from the file
    if isempty(variables)
        variables = who('-file', filename);
    end

    % Load a variable from the list of variables
    data = load(filename, variables{1});
```

```
    % Remove the newly-read variable from the list
    variables(1) = [];

    % Move on to the next file if this file is done reading.
    done = isempty(variables);

end
```

Create and setup a `FileDatastore` containing `accidents.mat`. Specify the datastore parameters to use `'partialfile'` as the read mode and `load_variable` as the custom reading function.

```
fds = fileDatastore('accidents.mat','ReadMode','partialfile','ReadFcn',@load_variable);
```

Read the first three variables from the file using the datastore. The file `accidents.mat` contains nine variables and every call to `read` returns one variable. Therefore, to get the first three variables, call the read function three times.

```
data = read(fds)
```

```
data = struct with fields:
    datasources: {3x1 cell}
```

```
data = read(fds)
```

```
data = struct with fields:
    hwycols: 17
```

```
data = read(fds)
```

```
data = struct with fields:
    hwydata: [51x17 double]
```

Note that the sample file `accidents.mat` is small and fits in memory, but you can expect similar results for large MAT-files that do not fit in memory.

Alternatives

You also can create a `FileDatastore` object using the `datastore` function. For example, `ds = datastore(location,'Type','file','ReadFcn',@fcn)` creates a datastore from a collection of files specified by `location`.

See Also

`datastore` | `mapreduce` | `tall`

Topics

"Getting Started with Datastore"
"Work with Remote Data"
"Develop Custom Datastore"
"Set Up Datastore for Processing on Different Machines or Clusters"

Introduced in R2016a

hasdata

Package: `matlab.io`

Determine if data is available to read

Syntax

```
tf = hasdata(ds)
```

Description

`tf = hasdata(ds)` returns logical `1` (`true`) if there is data available to read from the datastore specified by `ds`. Otherwise, it returns logical `0` (`false`).

Examples

Determine if Data is Available to Read

Create a datastore from the sample file, `mapredout.mat`, which is the output file of the `mapreduce` function.

```
ds = datastore('mapredout.mat');
```

While there is data available in the datastore, read the data.

```
while hasdata(ds)
    T = read(ds);
end
```

Input Arguments

ds — Input datastore
datastore

Input datastore. You can use these datastores as input to the `hasdata` method.

- MATLAB datastores — Datastores created using MATLAB `datastore` functions. For example, create a datastore for a collection of images using `ImageDatastore`. For a complete list of datastores, see “Select Datastore for File Format or Application”.
- Combined or transformed datastores — Datastores created using the `combine` and `transform` functions.
- Custom datastores — Datastores created using the custom datastore framework. See “Develop Custom Datastore”.

See Also

`datastore`

Introduced in R2014b

imageDatastore

Datastore for image data

Description

Use an `ImageDatastore` object to manage a collection of image files, where each individual image fits in memory, but the entire collection of images does not necessarily fit. You can create an `ImageDatastore` object using the `imageDatastore` function, specify its properties, and then import and process the data using object functions.

Creation

Syntax

```
imds = imageDatastore(location)
imds = imageDatastore(location,Name,Value)
```

Description

`imds = imageDatastore(location)` creates a datastore `imds` from the collection of image data specified by `location`.

`imds = imageDatastore(location,Name,Value)` specifies additional parameters and properties for `imds` using one or more name-value pair arguments.

Input Arguments

`location` — Files or folders to include in datastore
path | `DsFileSet` object

Files or folders included in the datastore, specified as a path or a `DsFileSet` object.

- path — Specify the path as a character vector, cell array of character vectors, string scalar, or a string array, containing the location of files or folders that are local or remote.
  - Local files or folders — Specify `location` as a local path to files or folders. If the files are not in the current folder, then local path must specify full or relative paths. Files within subfolders of the specified folder are not automatically included in the datastore. You can use the wildcard character (*) when specifying the local path. This character specifies that the datastore include all matching files or all files in the matching folders.
  - Remote files or folders — Specify `location` to be the full paths of the files or folders as a uniform resource locator (URL) of the form `hdfs:///path_to_file`. For more information, see “Work with Remote Data”.
- `DsFileSet` object — You also can specify `location` as a `DsFileSet` object. For more information, see `matlab.io.datastore.DsFileSet`.

When `location` represents a folder, the datastore includes only supported file formats and ignores any other format. To specify a custom list of file extensions to include in your datastore, see the `FileExtensions` property.

The `imageDatastore` function supports files that have an `imformats` format.

Example: `'file1.jpg'`

Example: `'../dir/data/file1.png'`

Example: `{'C:\dir\data\file1.tif','C:\dir\data\file2.tif'}`

Example: `'C:\dir\data\*.jpg'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `imds = imageDatastore('C:\dir\imagedata','FileExtensions', {'.jpg','.tif'})`

`IncludeSubfolders` — Subfolder inclusion flag
`false` (default) | `true`

Subfolder inclusion flag, specified as the comma-separated pair consisting of `'IncludeSubfolders'` and `true` or `false`. Specify `true` to include all files and subfolders within each folder or `false` to include only the files within each folder.

If you do not specify `'IncludeSubfolders'`, then the default value is `false`.

Example: `'IncludeSubfolders',true`

Data Types: `logical` | `double`

`FileExtensions` — Image file extensions
character vector | cell array of character vectors | string scalar | string array

Image file extensions, specified as the comma-separated pair consisting of `'FileExtensions'` and a character vector, cell array of character vectors, string scalar, or string array. The specified extensions do not require an `imformats` format, and you can use the empty quotes `''` to represent files without extensions. If you do not specify `'FileExtensions'`, then `imageDatastore` automatically includes all images with `imformats` extensions in the specified path. If you want to include extensions that `imformats` does not recognize, then specify all extensions.

Example: `'FileExtensions','.jpg'`

Example: `'FileExtensions',{'.jpg','.png'}`

Data Types: `char` | `cell` | `string`

`AlternateFileSystemRoots` — Alternate file system root paths
string vector | cell array

Alternate file system root paths, specified as the comma-separated pair consisting of `'AlternateFileSystemRoots'` and a string vector or a cell array. Use `'AlternateFileSystemRoots'` when you create a datastore on a local machine, but need to access and process the data on another machine (possibly of a different operating system). Also, when processing data using the Parallel Computing Toolbox and the MATLAB Parallel Server, and the data is stored on your local machines with a copy of the data available on different platform cloud or cluster machines, you must use `'AlternateFileSystemRoots'` to associate the root paths.

- To associate a set of root paths that are equivalent to one another, specify `'AlternateFileSystemRoots'` as a string vector. For example,

  ```
  ["Z:\datasets","/mynetwork/datasets"]
  ```

- To associate multiple sets of root paths that are equivalent for the datastore, specify `'AlternateFileSystemRoots'` as a cell array containing multiple rows where each row represents a set of equivalent root paths. Specify each row in the cell array as either a string vector or a cell array of character vectors. For example:
  - Specify `'AlternateFileSystemRoots'` as a cell array of string vectors.

    ```
    {["Z:\datasets", "/mynetwork/datasets"];...
     ["Y:\datasets", "/mynetwork2/datasets","S:\datasets"]}
    ```
  - Alternatively, specify `'AlternateFileSystemRoots'` as a cell array of cell array of character vectors.

    ```
    {{'Z:\datasets','/mynetwork/datasets'};...
     {'Y:\datasets', '/mynetwork2/datasets','S:\datasets'}}
    ```

The value of `'AlternateFileSystemRoots'` must satisfy these conditions:

- Contains one or more rows, where each row specifies a set of equivalent root paths.
- Each row specifies multiple root paths and each root path must contain at least two characters.
- Root paths are unique and are not subfolders of one another.
- Contains at least one root path entry that points to the location of the files.

For more information, see “Set Up Datastore for Processing on Different Machines or Clusters”.

Example: `["Z:\datasets","/mynetwork/datasets"]`

Data Types: `string` | `cell`

`LabelSource` — Source providing label data
`'none'` (default) | `'foldernames'`

Source providing label data, specified as the comma-separated pair consisting of `'LabelSource'` and `'none'` or `'foldernames'`. If `'none'` is specified, then the `Labels` property is empty. If `'foldernames'` is specified, then labels are assigned according to the folder names and stored in the `Labels` property. You can later modify the labels by accessing the `Labels` property directly.

Data Types: `char` | `string`

In addition to these name-value pairs, you also can specify any of the properties on this page as name-value pairs, except for the `Files` property.

Properties

`ImageDatastore` properties describe the data and specify how to read the data from the datastore. You can specify the value of `ImageDatastore` properties using name-value pair arguments when you create the datastore object. To view or modify a property after creating the object, use the dot notation.

For example, you can create an `ImageDatastore` object and specify the `'ReadFcn'` parameter:

```
imds = imageDatastore('peppers.png','ReadFcn',@customreader);
```

Alternatively, you can assign `'ReadFcn'` to `@customreader` after you create the `ImageDatastore`:

```
imds = imageDatastore('peppers.png');
imds.ReadFcn = @customreader;
```

`Files` — Files included in datastore

character vector | cell array of character vectors | string scalar | string array

Files included in the datastore, resolved as a character vector, cell array of character vectors, string scalar, or string array. Each character vector or string is a full path to a file. The `location` argument in the `imageDatastore` and `datastore` functions defines `Files` when the datastore is created.

Example: `{'C:\dir\data\file1.jpg';'C:\dir\data\file2.jpg'}`

Data Types: `char` | `cell` | `string`

`ReadSize` — Number of image files to read

1 (default) | positive integer scalar

Number of image files to read in a call to the `read` function, specified as a positive integer scalar. Each call to the `read` function reads at most `ReadSize` images.

`Labels` — File labels

categorical, logical, or numeric vector | cell array | string array

File labels for the files in the datastore, specified as a vector, a cell array, or a string array. The order of the labels in the array corresponds to the order of the associated files in the datastore. If you specify `'LabelSource','foldernames'` when creating the `ImageDatastore` object, then the label name for a file is the name of the folder

containing it. If you do not specify `'LabelSource','foldernames'`, then `Labels` is an empty cell array or string array. If you change the `Files` property after the datastore is created, then the `Labels` property is not automatically updated to incorporate the added files.

Data Types: `categorical` | `cell` | `logical` | `double` | `single` | `string`

`ReadFcn` — Function that reads image data
`@readDatastoreImage` (default) | function handle

Function that reads image data, specified as a function handle. The function must take an image file name as input, and then it outputs the corresponding image data. For example, if `customreader` is the specified function to read the image data, then it must have a signature similar to this:

```
function data = customreader(filename)
...
end
```

If more than one output argument exists, then `imageDatastore` uses only the first argument and ignores the rest.

Note Using `ReadFcn` to transform or pre-process 2-D images is not recommended. For file formats recognized by `imformats`, specifying `ReadFcn` slows down the performance of `imageDatastore`. For more efficient ways to transform and pre-process images, see "Preprocess Images for Deep Learning" (Deep Learning Toolbox).

Example: `@customreader`

Data Types: `function_handle`

Object Functions

countEachLabel	Count files in ImageDatastore labels
hasdata	Determine if data is available to read
numpartitions	Number of datastore partitions
partition	Partition a datastore
preview	Subset of data in datastore
read	Read data in datastore
readall	Read all data in datastore

readimage	Read specified image from datastore
reset	Reset datastore to initial state
shuffle	Shuffle files in ImageDatastore
splitEachLabel	Split ImageDatastore labels by proportions
subset	Create subset of datastore or file-set
transform	Transform datastore
combine	Combine data from multiple datastores

Examples

Create ImageDatastore Object

Create an `ImageDatastore` object associated with all `.tif` files in the MATLAB® path and its subfolders. Use the folder names as label names.

```
imds = imageDatastore(fullfile(matlabroot,'toolbox','matlab'),...
'IncludeSubfolders',true,'FileExtensions','.tif','LabelSource','foldernames')
```

```
imds = 

  ImageDatastore with properties:

       Files: {
              ' ...\matlab\toolbox\matlab\demos\example.tif';
              ' ...\matlab\toolbox\matlab\imagesci\corn.tif'
              }
      Labels: [demos; imagesci]
    ReadSize: 1
     ReadFcn: @readDatastoreImage
```

Specify Images to Read

Create an `ImageDatastore` object containing four images, and preview the first image.

```
imds = imageDatastore({'street1.jpg','street2.jpg','peppers.png','corn.tif'})
```

```
imds = 
```

```
  ImageDatastore with properties:

       Files: {
              ' ...\matlab\toolbox\matlab\demos\street1.jpg';
              ' ...\matlab\toolbox\matlab\demos\street2.jpg';
              ' ...\matlab\toolbox\matlab\imagesci\peppers.png'
               ... and 1 more
              }
    ReadSize: 1
      Labels: {}
     ReadFcn: @readDatastoreImage
```

```
imshow(preview(imds));
```

Read only the second and third images, one at a time.

```
for i = 2:3
    img = readimage(imds,i);
end
```

Read all four images and view the third image.

```
imgs = readall(imds);
imshow(imgs{3})
```

Limitations

- Using ReadFcn to transform or pre-process 2-D images is not recommended. For file formats recognized by `imformats`, specifying `ReadFcn` slows down the performance of `imageDatastore`. For more efficient ways to transform and pre-process images, see “Preprocess Images for Deep Learning” (Deep Learning Toolbox).

Alternatives

You also can create an `ImageDatastore` object using the `datastore` function. For example, `ds = datastore(location,'Type','image')` creates a datastore from a collection of files specified by `location`.

See Also

`datastore` | `imformats` | `mapreduce` | `tall`

Topics

"Read and Analyze Image Files"
"Getting Started with Datastore"
"Work with Remote Data"
"Set Up Datastore for Processing on Different Machines or Clusters"

Introduced in R2015b

KeyValueDatastore

Datastore for key-value pair data for use with `mapreduce`

Description

`KeyValueDatastore` objects are associated with files containing key-value pair data that are outputs of or inputs to `mapreduce`. Use the `KeyValueDatastore` properties to specify how you want to access the data. Use dot notation to view or modify a particular property of a `KeyValueDatastore` object:

```
ds = datastore('mapredout.mat');
ds.ReadSize = 20;
```

You also can specify the value of `KeyValueDatastore` properties using name-value pair arguments when you create a datastore using the `datastore` function:

```
ds = datastore('mapredout.mat','ReadSize',20);
```

Creation

Create `KeyValueDatastore` objects using the `datastore` function.

Properties

`Files` — Files included in datastore
cell array of character vectors | string array

Files included in the datastore, specified as an `n`-by-1 cell array of character vectors or string array, where each character vector or string is a full path to a file. These are the files defined by the `location` argument to the `datastore` function. The `location` argument contains full paths to files on a local file system, a network file system, or a supported remote location such as Amazon S3™, Windows Azure® Blob Storage, and HDFS. For more information, see “Work with Remote Data”.

The files must be either MAT-files or Sequence files generated by the `mapreduce` function.

Example: `{'C:\dir\data\file1.mat';'C:\dir\data\file2.mat'}`

Example: `{'s3://bucketname/path_to_files/your_file01.mat';'s3://bucketname/path_to_files/your_file02.mat'}`

Data Types: `cell` | `string`

`FileType` — File type
`'mat'` (default) | `'seq'`

File type, specified as either `'mat'` for MAT-files or `'seq'` for sequence files. By default, the output of `mapreduce` running against Hadoop® is a datastore containing sequence files. By default, the output of all other `mapreduce` operations is a datastore containing MAT-files.

Data Types: `cell` | `string`

`ReadSize` — Maximum number of key-value pairs to read
1 (default) | positive integer

Maximum number of key-value pairs to read in a call to the `read` or `preview` functions, specified as a positive integer.

`AlternateFileSystemRoots` — Alternate file system root paths
string vector | cell array

Alternate file system root paths, specified as the comma-separated pair consisting of `'AlternateFileSystemRoots'` and a string vector or a cell array. Use `'AlternateFileSystemRoots'` when you create a datastore on a local machine, but need to access and process the data on another machine (possibly of a different operating system). Also, when processing data using the Parallel Computing Toolbox and the MATLAB Parallel Server, and the data is stored on your local machines with a copy of the data available on different platform cloud or cluster machines, you must use `'AlternateFileSystemRoots'` to associate the root paths.

- To associate a set of root paths that are equivalent to one another, specify `'AlternateFileSystemRoots'` as a string vector. For example,

  ```
  ["Z:\datasets","/mynetwork/datasets"]
  ```

- To associate multiple sets of root paths that are equivalent for the datastore, specify `'AlternateFileSystemRoots'` as a cell array containing multiple rows where each

row represents a set of equivalent root paths. Specify each row in the cell array as either a string vector or a cell array of character vectors. For example:

- Specify `'AlternateFileSystemRoots'` as a cell array of string vectors.

  ```
  {["Z:\datasets", "/mynetwork/datasets"];...
   ["Y:\datasets", "/mynetwork2/datasets","S:\datasets"]}
  ```

- Alternatively, specify `'AlternateFileSystemRoots'` as a cell array of cell array of character vectors.

  ```
  {{'Z:\datasets','/mynetwork/datasets'};...
   {'Y:\datasets', '/mynetwork2/datasets','S:\datasets'}}
  ```

The value of `'AlternateFileSystemRoots'` must satisfy these conditions:

- Contains one or more rows, where each row specifies a set of equivalent root paths.
- Each row specifies multiple root paths and each root path must contain at least two characters.
- Root paths are unique and are not subfolders of one another.
- Contains at least one root path entry that points to the location of the files.

For more information, see “Set Up Datastore for Processing on Different Machines or Clusters”.

Example: `["Z:\datasets","/mynetwork/datasets"]`

Data Types: `string` | `cell`

Object Functions

hasdata	Determine if data is available to read
numpartitions	Number of datastore partitions
partition	Partition a datastore
preview	Subset of data in datastore
read	Read data in datastore
readall	Read all data in datastore
reset	Reset datastore to initial state
transform	Transform datastore
combine	Combine data from multiple datastores

Examples

Set Number of Key-Value Pairs to Read

Create a datastore from the sample file, `mapredout.mat`, which is an output file of the `mapreduce` function.

```
ds = datastore('mapredout.mat')
```

```
ds = 

  KeyValueDatastore with properties:

       Files: {
              ' ...\matlab\toolbox\matlab\demos\mapredout.mat'
              }
    ReadSize: 1 key-value pairs
    FileType: 'mat'
```

Set the `ReadSize` property to `8` so that each call to read reads at most `8` key-value pairs.

```
ds.ReadSize = 8
```

```
ds = 

  KeyValueDatastore with properties:

       Files: {
              ' ...\matlab\toolbox\matlab\demos\mapredout.mat'
              }
    ReadSize: 8 key-value pairs
    FileType: 'mat'
```

Read 8 key-value pairs at a time using the `read` function in a `while` loop. The loop executes until there is no more data available to read and `hasdata(ds)` returns `false`.

```
while hasdata(ds)
    T = read(ds);
end
```

Show the last set of key-value pairs read.

```
T
```

```
T =

    Key     Value
    ____    ______

    'OO'    [3090]
    'TZ'    [ 216]
    'XE'    [2357]
    '9E'    [ 521]
    'YV'    [ 849]
```

Limitations

- `KeyValueDatastore` does not support sequence files written in R2013b. Rewrite the sequence files using a version of MATLAB between R2014a and R2018a.

See Also

`datastore` | `mapreduce`

Topics

"Read and Analyze MAT-File with Key-Value Data"
"Read and Analyze Hadoop Sequence File"
"Process Large Set of Images Using MapReduce Framework and Hadoop" (Image Processing Toolbox)
"Set Up Datastore for Processing on Different Machines or Clusters"

Introduced in R2014b

numpartitions

Package: `matlab.io.datastore`

Number of datastore partitions

Syntax

```
n = numpartitions(ds)
n = numpartitions(ds,pool)
```

Description

`n = numpartitions(ds)` returns the default number of partitions for datastore `ds`.

`n = numpartitions(ds,pool)` returns a number of partitions to parallelize datastore access over the parallel pool specified by `pool`. To parallelize datastore access, you must have Parallel Computing Toolbox installed.

Examples

Number of Partitions

Create a datastore from the sample file, `mapredout.mat`, which is the output file of the `mapreduce` function.

```
ds = datastore('mapredout.mat');
```

Get the default number of partitions.

```
n = numpartitions(ds)
```

```
n = 1
```

By default, there is only one partition in `ds` because it contains only one small file.

Partition the datastore and return the datastore corresponding to the first part.

```
subds = partition(ds,n,1);
```

Read the data in `subds`.

```
while hasdata(subds)
    data = read(subds);
end
```

Number of Partitions for Parallel Datastore Access

Get a number of partitions to parallelize datastore access over the current parallel pool. You must have Parallel Computing Toolbox installed.

Create a datastore from the sample file, `mapredout.mat`, which is the output file of the `mapreduce` function.

```
ds = datastore('mapredout.mat');
```

Get a number of partitions to parallelize datastore access over the current parallel pool.

```
n = numpartitions(ds, gcp);
```

Partition the datastore and read the data in each part.

```
parfor ii=1:n
    subds = partition(ds,n,ii);
    while hasdata(subds)
        data = read(subds);
    end
end
```

Input Arguments

`ds` — Input datastore
datastore

Input datastore. You can use the `datastore` function to create a datastore object from your data.

`pool` — Parallel pool
parallel pool object

Parallel pool object.

Example: `gcp`

See Also

`datastore` | `partition`

Topics

"Partition a Datastore in Parallel" (Parallel Computing Toolbox)

Introduced in R2015a

parquetDatastore

Datastore for collection of Parquet files

Description

Use a `ParquetDatastore` object to manage a collection of Parquet files, where each individual Parquet file fits in memory, but the entire collection of files does not necessarily fit. You can create a `ParquetDatastore` object using the `parquetDatastore` function, specify its properties, and then import and process the data using object functions.

Creation

Syntax

```
pds = parquetDatastore(location)
pds = parquetDatastore(location,Name,Value)
```

Description

`pds = parquetDatastore(location)` creates a datastore `pds` from the collection of Parquet files specified by `location`.

`pds = parquetDatastore(location,Name,Value)` specifies additional parameters and properties for `pds` using one or more name-value pair arguments.

Input Arguments

`location` — Files or folders to include in datastore
path | `DsFileSet` object

Files or folders included in the datastore, specified as a path or a `DsFileSet` object.

- path — Specify the path as a character vector, cell array of character vectors, string scalar, or a string array, containing the location of files or folders that are local or remote.
  - Local files or folders — Specify `location` as a local path to files or folders. If the files are not in the current folder, then local path must specify full or relative paths. Files within subfolders of the specified folder are not automatically included in the datastore. You can use the wildcard character (*) when specifying the local path. This character specifies that the datastore include all matching files or all files in the matching folders.
  - Remote files or folders — Specify `location` to be the full paths of the files or folders as a uniform resource locator (URL) of the form `hdfs:///path_to_file`. For more information, see “Work with Remote Data”.
- `DsFileSet` object — You also can specify `location` as a `DsFileSet` object. For more information, see `matlab.io.datastore.DsFileSet`.

When `location` represents a folder, the datastore includes only supported file formats and ignores any other format. To specify a custom list of file extensions to include in your datastore, see the `FileExtensions` property.

The `parquetDatastore` function supports the `.parquet` file format.

Example: `'myfile.parquet'`

Example: `'../dir/data/myfile.parquet'`

Example: `{'C:\dir\data\myfile01.parquet','C:\dir\data\myfile02.parquet'}`

Example: `'s3://bucketname/path_to_files/*.parquet'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'IncludeSubfolders',true`

`FileExtensions` — Extensions to include in datastore
character vector | cell array of character vectors | string scalar | string array

Extensions to include in datastore, specified as the comma-separated pair consisting of `'FileExtensions'` and a character vector, cell array of character vectors, string scalar, or string array.

- If you do not specify `'FileExtensions'`, then `parquetDatastore` automatically includes all files with `.parquet` and `.parq` extensions in the specified path.
- If you want to include parquet files with non-standard file extensions in the `parquetDatastore`, then specify those extensions explicitly.
- If you want to create a `parquetDatastore` for files without any extensions, then specify `'FileExtensions'` as an empty character vector, `''`.

Example: `'FileExtensions',{'.parquet','.parq'}`

Example: `'FileExtensions','.myformat'`

Example: `'FileExtensions',''`

Data Types: `char` | `cell` | `string`

`IncludeSubfolders` — Subfolder inclusion flag
`false` (default) | `true`

Subfolder inclusion flag, specified as the comma-separated pair consisting of `'IncludeSubfolders'` and `true` or `false`. Specify `true` to include all files and subfolders within each folder or `false` to include only the files within each folder.

If you do not specify `'IncludeSubfolders'`, then the default value is `false`.

Example: `'IncludeSubfolders',true`

Data Types: `logical` | `double`

`OutputType` — Output datatype
`'auto'` (default) | `'table'` | `'timetable'`

Output datatype, specified as the comma-separated pair consisting of `'OutputType'` and `'auto'`, `'table'`, or `'timetable'`.

- `'auto'` — Return a table or a timetable. The `parquetDatastore` detects if the output should be a table or a timetable based on other name-value pairs that you specify. When you specify the `RowTimes` name-value pair, the `parquetDatastore` infers that the output is a timetable.
- `'table'` — Return a table. For more information on the table datatype, see `table`.

- `'timetable'` — Return a timetable. For more information on the timetables, see `timetable`.

The value of `OutputType` selects the data type returned from the `preview`, `read`, and `readall` functions.

Example: `'OutputType','timetable'`

Data Types: `char` | `string`

`PreserveVariableNames` — Flag to preserve variable names
`false` (default) | `true`

Flag to preserve variable names, specified as the comma-separated pair consisting of `PreserveVariableNames` and either `true`, or `false`.

- `true` — Preserve variable names that are not valid MATLAB identifiers such as variable names that include spaces and non-ASCII characters.
- `false` — Convert invalid variable names (as determined by the `isvarname` function) to valid MATLAB identifiers.

Starting in R2019b, variable names and row names can include any characters, including spaces and non-ASCII characters. Also, they can start with any characters, not just letters. Variable and row names do not have to be valid MATLAB identifiers (as determined by the `isvarname` function). To preserve these variable names and row names, set `PreserveVariableNames` to `true`.

`AlternateFileSystemRoots` — Alternate file system root paths
string vector | cell array

Alternate file system root paths, specified as the comma-separated pair consisting of `'AlternateFileSystemRoots'` and a string vector or a cell array. Use `'AlternateFileSystemRoots'` when you create a datastore on a local machine, but need to access and process the data on another machine (possibly of a different operating system). Also, when processing data using the Parallel Computing Toolbox and the MATLAB Parallel Server, and the data is stored on your local machines with a copy of the data available on different platform cloud or cluster machines, you must use `'AlternateFileSystemRoots'` to associate the root paths.

- To associate a set of root paths that are equivalent to one another, specify `'AlternateFileSystemRoots'` as a string vector. For example,

  ```
  ["Z:\datasets","/mynetwork/datasets"]
  ```

- To associate multiple sets of root paths that are equivalent for the datastore, specify `'AlternateFileSystemRoots'` as a cell array containing multiple rows where each row represents a set of equivalent root paths. Specify each row in the cell array as either a string vector or a cell array of character vectors. For example:
  - Specify `'AlternateFileSystemRoots'` as a cell array of string vectors.

    ```
    {["Z:\datasets", "/mynetwork/datasets"];...
     ["Y:\datasets", "/mynetwork2/datasets","S:\datasets"]}
    ```
  - Alternatively, specify `'AlternateFileSystemRoots'` as a cell array of cell array of character vectors.

    ```
    {{'Z:\datasets','/mynetwork/datasets'};...
     {'Y:\datasets', '/mynetwork2/datasets','S:\datasets'}}
    ```

The value of `'AlternateFileSystemRoots'` must satisfy these conditions:

- Contains one or more rows, where each row specifies a set of equivalent root paths.
- Each row specifies multiple root paths and each root path must contain at least two characters.
- Root paths are unique and are not subfolders of one another.
- Contains at least one root path entry that points to the location of the files.

For more information, see “Set Up Datastore for Processing on Different Machines or Clusters”.

Example: `["Z:\datasets","/mynetwork/datasets"]`

Data Types: `string` | `cell`

Properties

`ParquetDatastore` properties describe the format of the files in a datastore object, and control how the data is read from the datastore. With the exception of the `Files` property, you can specify the value of `ParquetDatastore` properties using name-value pair arguments when you create the datastore object. To view or modify a property after creating the object, use the dot notation.

`Files` — Files included in datastore

cell array of character vectors | string array

Files included in the datastore, resolved as a cell array of character vectors or a string array, where each character vector or string is a full path to a file. The `location` argument defines these files.

The first file specified in the cell array determines the variable names and format information for all files in the datastore.

Example: `{'C:\dir\data\file1.ext';'C:\dir\data\file2.ext'}`

Data Types: `cell` | `string`

`ReadSize` — Amount of data to read
`'rowgroup'` (default) | `'file'` | positive integer

Amount of data to read in a call to the `read` function, specified as `'rowgroup'`, `'file'`, or a positive integer.

- `'rowgroup'` — Each call to `read` reads the number of rows specified in the row groups of the Parquet file. To get the number of rows in row groups, see the `RowGroupHeights` property of the `ParquetInfo` object.
- `'file'` — Each call to `read` reads all of the data in one file.
- positive integer — Each call to `read` reads a maximum of `ReadSize` rows.

When you change `ReadSize` from a positive integer to `'file'` or `'rowgroup'`, or vice versa, MATLAB resets the datastore to an unread state where no data has been read from it.

Data Types: `double` | `char` | `string`

`VariableNames` — Names of variables
character vector | cell array of character vectors | string scalar | string array

Names of variables in the datastore, specified as a character vector, cell array of character vectors, string scalar, or string array. Specify the variable names in the order in which they appear in the files. If you do not specify the variable names, the datastore detects them from the first nonheader line in the first file. You can specify `VariableNames` with a character vector or string scalar, however the datastore converts and stores the property value to a cell array of character vectors. When modifying the `VariableNames` property, the number of new variable names must match the number of original variable names.

To support invalid MATLAB identifiers as variable names, such as varible names containing spaces and non-ASCII characters, set the `PreserveVariableNames` parameter to `true`.

If `ReadVariableNames` is `false`, then `VariableNames` defaults to `{'Var1','Var2', ...}`.

Example: `{'Time','Date','Quantity'}`

Data Types: `char` | `cell` | `string`

`SelectedVariableNames` — Variables to read
cell array of character vectors | string array

Variables to read from the file, specified as a cell array of character vectors or a string array, where each character vector or string contains the name of one variable. You can specify the variable names in any order.

To support invalid MATLAB identifiers as variable names, such as varible names containing spaces and non-ASCII characters, set the `PreserveVariableNames` parameter to `true`.

Example: `{'Var3','Var7','Var4'}`

Data Types: `cell` | `string`

`RowTimes` — Name of row times variable
character vector | string scalar

Name of the row times variable in the Parquet data, specified as the comma-separated pair consisting of `'RowTimes'` and a character vector or string array containing the variable name.

`RowTimes` is a timetable related parameter. Each row of a timetable is associated with a time, which is captured in a time vector for the timetable. The variable specified in `RowTimes`, must contain a `datetime` or a `duration` vector.

By default, `parquetDatastore` uses the first `datetime` or `duration` variable as row times for the timetable.

Object Functions

hasdata	Determine if data is available to read

numpartitions	Number of datastore partitions
partition	Partition a datastore
preview	Subset of data in datastore
read	Read data in datastore
readall	Read all data in datastore
reset	Reset datastore to initial state
transform	Transform datastore
combine	Combine data from multiple datastores

Examples

Create ParquetDatastore Object

Create a `ParquetDatastore` object containing the file `outages.parquet`.

```
pds = parquetDatastore('outages.parquet')
```

```
pds 
  ParquetDatastore with properties:

                       Files: {
                              'B:\matlab\toolbox\matlab\demos\outages.parquet'
                              }
       PreserveVariableNames: false
               VariableNames: {1x6 cell}
       SelectedVariableNames: {1x6 cell}
                    ReadSize: 'rowgroup'
                  OutputType: 'table'
                    RowTimes: []
    AlternateFileSystemRoots: {}
```

Specify Read Size for ParquetDatastore

Create a datastore for a sample Parquet file, and then read data from the file with different `ReadSize` values.

Create a datastore for `airlinesmall.parquet`, set `ReadSize` to `10` rows, and then read from the datastore. The value of `ReadSize` determines how many rows of data are read from the datastore with each call to the `read` function.

```
pds = parquetDatastore('outages.parquet','ReadSize',10);
read(pds)
```

```
ans=10×6 table
      Region            OutageTime          Loss     Customers       RestorationTime
    ___________    ____________________    ______    __________    ___________________

    "SouthWest"    01-Feb-2002 12:18:00    458.98    1.8202e+06    07-Feb-2002 16:50:00
    "SouthEast"    23-Jan-2003 00:49:00    530.14    2.1204e+05                   NaT
    "SouthEast"    07-Feb-2003 21:15:00     289.4    1.4294e+05    17-Feb-2003 08:14:00
    "West"         06-Apr-2004 05:44:00    434.81    3.4037e+05    06-Apr-2004 06:10:00
    "MidWest"      16-Mar-2002 06:18:00    186.44    2.1275e+05    18-Mar-2002 23:23:00
    "West"         18-Jun-2003 02:49:00         0             0    18-Jun-2003 10:54:00
    "West"         20-Jun-2004 14:39:00    231.29           NaN    20-Jun-2004 19:16:00
    "West"         06-Jun-2002 19:28:00    311.86           NaN    07-Jun-2002 00:51:00
    "NorthEast"    16-Jul-2003 16:23:00    239.93         49434    17-Jul-2003 01:12:00
    "MidWest"      27-Sep-2004 11:09:00    286.72         66104    27-Sep-2004 16:37:00
```

Set the `ReadSize` property value to `'file'` and read from the datastore. Every call to the `read` function reads all the data from the datastore.

```
pds.ReadSize ='file';
data = read(pds)
```

```
data=1468×6 table
      Region            OutageTime          Loss     Customers       RestorationTime
    ___________    ____________________    ______    __________    ___________________

    "SouthWest"    01-Feb-2002 12:18:00    458.98    1.8202e+06    07-Feb-2002 16:50:00
    "SouthEast"    23-Jan-2003 00:49:00    530.14    2.1204e+05                   NaT
    "SouthEast"    07-Feb-2003 21:15:00     289.4    1.4294e+05    17-Feb-2003 08:14:00
    "West"         06-Apr-2004 05:44:00    434.81    3.4037e+05    06-Apr-2004 06:10:00
    "MidWest"      16-Mar-2002 06:18:00    186.44    2.1275e+05    18-Mar-2002 23:23:00
    "West"         18-Jun-2003 02:49:00         0             0    18-Jun-2003 10:54:00
    "West"         20-Jun-2004 14:39:00    231.29           NaN    20-Jun-2004 19:16:00
    "West"         06-Jun-2002 19:28:00    311.86           NaN    07-Jun-2002 00:51:00
    "NorthEast"    16-Jul-2003 16:23:00    239.93         49434    17-Jul-2003 01:12:00
    "MidWest"      27-Sep-2004 11:09:00    286.72         66104    27-Sep-2004 16:37:00
    "SouthEast"    05-Sep-2004 17:48:00    73.387         36073    05-Sep-2004 20:46:00
    "West"         21-May-2004 21:45:00    159.99           NaN    22-May-2004 04:23:00
```

```
    "SouthEast"    01-Sep-2002 18:22:00    95.917         36759    01-Sep-2002 19:12:00
    "SouthEast"    27-Sep-2003 07:32:00       NaN    3.5517e+05    04-Oct-2003 07:02:00
    "West"         12-Nov-2003 06:12:00    254.09    9.2429e+05    17-Nov-2003 02:04:00
    "NorthEast"    18-Sep-2004 05:54:00         0             0                     NaT
      ⋮
```

You also can set the value of `ReadSize` property to `'rowgroup'`. For more information, see the `ReadSize` property of the `ParquetDatastore` object reference page.

Limitations

If you use `parquetread` or `datastore` to read the files, then the result might not have the same format or contents as the original table. For more information, see “Apache Parquet Data Type Mappings”.

Alternatives

You also can create a `ParquetDatastore` object using the `datastore` function. For example, `ds = datastore(location,'Type','parquet')` creates a datastore from a collection of files specified by `location`.

See Also

`datastore` | `mapreduce` | `parquetinfo` | `parquetread` | `tall`

Introduced in R2019a

partition

Package: `matlab.io.datastore`

Partition a datastore

Syntax

```
subds = partition(ds,n,index)
```

```
subds = partition(ds,'Files',index)
subds = partition(ds,'Files',filename)
```

Description

`subds = partition(ds,n,index)` partitions datastore `ds` into the number of parts specified by `n` and returns the partition corresponding to the index `index`.

`subds = partition(ds,'Files',index)` partitions the datastore by files and returns the partition corresponding to the file of index `index` in the `Files` property.

`subds = partition(ds,'Files',filename)` partitions the datastore by files and returns the partition corresponding to the file specified by `filename`.

Examples

Partition Datastore into Specific Number of Parts

Create a datastore for a large collection of files. For this example, use ten copies of the sample file `airlinesmall.csv`. To handle missing fields in the tabular data, specify the name-value pairs `TreatAsMissing` and `MissingValue`.

```
files = repmat({'airlinesmall.csv'},1,10);
ds = tabularTextDatastore(files,...
                  'TreatAsMissing','NA','MissingValue',0);
```

Partition the datastore into three parts and return the first partition. The `partition` function returns approximately the first third of the data from the datastore `ds`.

```
subds = partition(ds,3,1)
```

```
subds 
  TabularTextDatastore with properties:

                      Files: {
                             ' ...\matlab\toolbox\matlab\demos\airlinesmall.csv';
                             ' ...\matlab\toolbox\matlab\demos\airlinesmall.csv';
                             ' ...\matlab\toolbox\matlab\demos\airlinesmall.csv'
                              ... and 1 more
                             }
              FileEncoding: 'UTF-8'
   AlternateFileSystemRoots: {}
           ReadVariableNames: true
               VariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}

  Text Format Properties:
            NumHeaderLines: 0
                 Delimiter: ','
              RowDelimiter: '\r\n'
            TreatAsMissing: 'NA'
              MissingValue: 0

  Advanced Text Format Properties:
           TextscanFormats: {'%f', '%f', '%f' ... and 26 more}
                  TextType: 'char'
        ExponentCharacters: 'eEdD'
              CommentStyle: ''
                Whitespace: ' \b\t'
    MultipleDelimitersAsOne: false

  Properties that control the table returned by preview, read, readall:
      SelectedVariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
            SelectedFormats: {'%f', '%f', '%f' ... and 26 more}
                   ReadSize: 20000 rows
```

The `Files` property of the datastore contains a list of files included in the datastore. Check the number of files in the `Files` property of the datastore `ds` and the partitioned datastore `subds`. The datastore `ds` contains ten files and the partition `subds` contains the first four files.

```
length(ds.Files)
```

```
ans = 10
```

```
length(subds.Files)
```

```
ans = 4
```

Partition Datastore into Default Number of Parts

Create a datastore from the sample file, `mapredout.mat`, which is the output file of the `mapreduce` function.

```
ds = datastore('mapredout.mat');
```

Get the default number of partitions for `ds`.

```
n = numpartitions(ds);
```

Partition the datastore into the default number of partitions and return the datastore corresponding to the first partition.

```
subds = partition(ds,n,1);
```

Read the data in `subds`.

```
while hasdata(subds)
    data = read(subds);
end
```

Partition Datastore by Files

Create a datastore that contains three image files.

```
ds = imageDatastore({'street1.jpg','peppers.png','corn.tif'})
```

```
ds = 

  ImageDatastore with properties:
```

```
       Files: {
              ' ...\matlab\toolbox\matlab\demos\street1.jpg';
              ' ...\matlab\toolbox\matlab\imagesci\peppers.png';
              ' ...\matlab\toolbox\matlab\imagesci\corn.tif'
              }
    ReadSize: 1
      Labels: {}
     ReadFcn: @readDatastoreImage
```

Partition the datastore by files and return the part corresponding to the second file.

```
subds = partition(ds,'Files',2)
```

```
subds = 

  ImageDatastore with properties:

       Files: {
              ' ...\matlab\toolbox\matlab\imagesci\peppers.png'
              }
    ReadSize: 1
      Labels: {}
     ReadFcn: @readDatastoreImage
```

`subds` contains one file.

Partition Data in Parallel

Create a datastore from the sample file, `mapredout.mat`, which is the output file of the `mapreduce` function.

```
ds = datastore('mapredout.mat');
```

Partition the datastore into three parts on three workers in a parallel pool.

```
numWorkers = 3;
p = parpool('local',numWorkers);
n = numpartitions(ds,p);
```

```
parfor ii=1:n
    subds = partition(ds,n,ii);
    while hasdata(subds)
        data = read(subds);
    end
end
```

Input Arguments

ds — Input datastore
datastore

Input datastore. You can use the `datastore` function to create a datastore object from your data.

n — Number of partitions
positive integer

Number of partitions, specified as a positive integer.

Example: `3`

Data Types: `double`

index — Index
positive integer

Index, specified as a positive integer.

Example: `1`

Data Types: `double`

filename — file name
character vector | string scalar

File name, specified as a character vector or string scalar.

The value of `filename` must match exactly the file name contained in the `Files` property of the datastore. To ensure that the file names match exactly, specify `filename` using `ds.Files{N}` where `N` is the index of the file in the `Files` property. For example, `ds.Files{3}` specifies the third file in the datastore `ds`.

Example: `ds.Files{3}`

Example: `'file1.csv'`

Example: `'../dir/data/file1.csv'`

Example: `'hdfs://myserver:7867/data/file1.txt'`

Data Types: `char`

Output Arguments

subds — Output datastore
datastore

Output datastore. The output datastore is of the same type as the input datastore `ds`.

See Also

`datastore` | `numpartitions`

Topics

"Partition a Datastore in Parallel" (Parallel Computing Toolbox)

Introduced in R2015a

preview

Package: `matlab.io`

Subset of data in datastore

Syntax

```
data = preview(ds)
```

Description

`data = preview(ds)` returns a subset of data from datastore `ds` without changing its current position.

Examples

Preview Data in TabularTextDatastore

Create a datastore from the sample file, `airlinesmall.csv`, which contains tabular data.

```
ds = tabularTextDatastore('airlinesmall.csv','TreatAsMissing','NA');
```

Modify the `SelectedVariableNames` property to specify the variables of interest.

```
ds.SelectedVariableNames = {'DepTime','ArrTime','ActualElapsedTime'};
```

Preview the data for the selected variables.

```
data = preview(ds)
```

```
data=8×3 table
    DepTime    ArrTime    ActualElapsedTime
    _______    _______    _________________
```

```
       642        735              53
      1021       1124              63
      2055       2218              83
      1332       1431              59
       629        746              77
      1446       1547              61
       928       1052              84
       859       1134             155
```

Preview Data in KeyValueDatastore

Create a datastore from the sample file, `mapredout.mat`, which is the output file of the `mapreduce` function.

```
ds = datastore('mapredout.mat');
```

Preview the data in the datastore.

```
data = preview(ds)
```

```
data=1×2 table
     Key        Value
    ______    _________

    {'AA'}    {[14930]}
```

Preview Data in CombinedDatastore

Create a datastore that maintains parity between the pair of images of the underlying datastores. For instance, create two separate image datastores, and then create a combined datastore representing the two underlying datastores.

Create an image datastore `imds1` representing a collection of three images.

```
imds1 = imageDatastore({'street1.jpg','street2.jpg','peppers.png'});
```

Create a second datastore `imds2` by transforming the images of `imds1` to grayscale and then reflecting the images horizontally.

```
imds2 = transform(imds1,@(x) fliplr(rgb2gray(x)));
```

Create a combined datastore from `imds1` and `imds2`.

```
imdsCombined = combine(imds1,imds2);
```

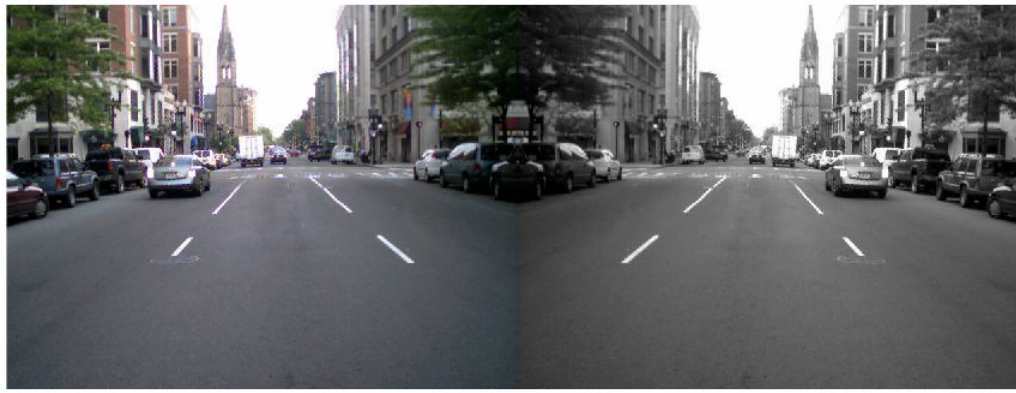
Preview the data in the combined datastore. The output is a 1-by-2 cell array. The two columns represent the first subset of data from the two underlying datastores `imds1` and `imds2`, respectively.

```
dataOut = preview(imdsCombined)
```

```
dataOut=1×2 cell
    {480x640x3 uint8}    {480x640 uint8}
```

Display the previewed data as a pair of tiled images.

```
tile = imtile(dataOut);
imshow(tile)
```



Input Arguments

ds — Input datastore
datastore

Input datastore. You can use these datastores as input to the `preview` method.

- MATLAB datastores — Datastores created using MATLAB `datastore` functions. For example, create a datastore for a collection of images using `ImageDatastore`. For a complete list of datastores, see “Select Datastore for File Format or Application”.
- Combined and transformed datastores — Datastores created using the `combine` and `transform` functions.
- Custom datastores — Datastores created using the custom datastore framework. See “Develop Custom Datastore”.

Output Arguments

data — Subset of data
table | array

Subset of data, returned as a table or an array depending on the type of `ds`.

Type of Datastore	Data type of data	Description
`TabularTextDatastore` and `SpreadsheetDatastore`	Table	Table with variables specified by the `SelectedVariableNames` property. The table contains at most eight rows.

Type of Datastore	Data type of `data`	Description
`ImageDatastore`	Integer array	Array of integers corresponding to the first image. The dimensions of the integer array depend on the type of image: • For grayscale images, `data` is m-by-n. • For truecolor images, `data` is m-by-n-by-3. • For CMYK Tiff images, `data` is m-by-n-by-4. The `preview` function supports all image types supported by the `imread` function. For more information on the supported image types, see `imread`.
`KeyValueDatastore`	Table	Table with the variables `Key` and `Value`.
`FileDatastore`	Table	Table containing the output returned by the read function, specified by the `'ReadFcn'` parameter in the `fileDatastore` function.
`TransformedDatastore`	Varies	The output is the same as the output returned by the underlying datastore specified by the `UnderlyingDatastore` property. For example, if the underlying datastore is an image datastore with a `ReadSize` property value of 1, then `data` is returned as a integer array.
`CombinedDatastore`	Cell array	Each element of the cell array contains the output returned by the corresponding underlying datastore specified by the `UnderlyingDatastores` property.

See Also

`datastore` | `hasdata`

Introduced in R2014b

read

Package: `matlab.io`

Read data in datastore

Syntax

```
data = read(ds)
[data,info] = read(ds)
```

Description

`data = read(ds)` returns data from a datastore. Subsequent calls to the `read` function continue reading from the endpoint of the previous call.

`[data,info] = read(ds)` also returns information about the extracted data in `info`, including metadata.

Examples

Read Data in TabularTextDatastore

Create a datastore from the sample file, `airlinesmall.csv`, which contains tabular data.

```
ds = tabularTextDatastore('airlinesmall.csv','TreatAsMissing','NA','MissingValue',0);
```

Modify the `SelectedVariableNames` property to specify the variables of interest.

```
ds.SelectedVariableNames = {'DepTime','ArrTime','ActualElapsedTime'};
```

While there is data available to be read from the datastore, read one block of data at a time and analyze the data. In this example, sum the actual elapsed time.

```
sumElapsedTime = 0;
while hasdata(ds)
    T = read(ds);
    sumElapsedTime = sumElapsedTime + sum(T.ActualElapsedTime);
end
```

View the sum of the actual elapsed time.

```
sumElapsedTime
```

```
sumElapsedTime = 14531797
```

Read Data in KeyValueDatastore

Create a datastore from the sample file, `mapredout.mat`, which is the output file of the `mapreduce` function.

```
ds = datastore('mapredout.mat');
```

Read a subset of data in the datastore.

```
T = read(ds)
```

```
T=1×2 table
     Key        Value
    ______    _________

    {'AA'}    {[14930]}
```

Change the number of key-value pairs to read at a time, by changing the `ReadSize` property of the datastore.

```
ds.ReadSize = 5;
```

Read the next five key-value pairs in the datastore.

```
T = read(ds)
```

```
T=5×2 table
     Key        Value
    ______    _________
```

```
    {'AS'}    {[ 2910]}
    {'CO'}    {[ 8138]}
    {'DL'}    {[16578]}
    {'EA'}    {[  920]}
    {'HP'}    {[ 3660]}
```

Read Data in CombinedDatastore

Create a datastore that maintains parity between the pair of images of the underlying datastores. For instance, create two separate image datastores, and then create a combined datastore representing the two underlying datastores.

Create an image datastore `imds1` representing a collection of three images.

```
imds1 = imageDatastore({'street1.jpg','street2.jpg','peppers.png'});
```

Create a second datastore `imds2` containing a mask of the bright regions of the three images. To create this datastore, first transform the images of `imds1` to grayscale. Then convert each image to a binary mask by mapping bright pixels (with a value greater than 250) to white and dark pixels to black.

```
imds2 = transform(imds1,@(x) rgb2gray(x)>250);
```

Create a combined datastore from `imds1` and `imds2`.

```
imdsCombined = combine(imds1,imds2);
```

Read the first subset of data from the combined datastore. The output is a 1-by-2 cell array. The two columns represent the first subset of data read from the two underlying datastores `imds1` and `imds2`, respectively.

```
dataOut = read(imdsCombined)
```

```
dataOut=1×2 cell
    {480x640x3 uint8}    {480x640 logical}
```

Display the read data from the combined datastore as a pair of tiled images.

```
tile = imtile(dataOut);
imshow(tile)
```

Input Arguments

ds — Input datastore
datastore

Input datastore. You can use these datastores as input to the `read` method.

- MATLAB datastores — Datastores created using MATLAB `datastore` functions. For example, create a datastore for a collection of images using `ImageDatastore`. For a complete list of datastores, see “Select Datastore for File Format or Application”.
- Combined and transformed datastores — Datastores created using the `combine` and `transform` functions.
- Custom datastores — Datastores created using the custom datastore framework. See “Develop Custom Datastore”.

Output Arguments

data — Output data
table

Output data, returned as a table or an array depending on the type of `ds`.

Type of Datastore	Data type of data	Description
`TabularTextDatastore` and `SpreadsheetDatastore`	Table	The `SelectedVariableNames` property determines the table variables.
`ImageDatastore`	Integer array	The dimensions of the integer array depend on the type of image: • For grayscale images, `data` is m-by-n. • For truecolor images, `data` is m-by-n-by-3. • For CMYK Tiff images, `data` is m-by-n-by-4. If the `ReadSize` property is greater than 1, then `data` is a cell array of image data corresponding to each image. The `read` function supports all image types supported by the `imread` function. For more information on the supported image types, see `imread`.
`KeyValueDatastore`	Table	The table variable names are `Key` and `Value`.
`FileDatastore`	Varies	The output is the same as the output returned by the custom read function, specified by the `'ReadFcn'` value.
`TransformedDatastore`	Varies	The output is the same as the output of the transformation function `@fcn` specified in the `transform` method used to create the `TransformedDatastore`.
`CombinedDatastore`	Varies	Contains the horizontal concatenation of the output of read from the corresponding underlying datastores.

`info` — Information about read data

structure array | cell array

Information about read data, returned as a structure array or a cell array of structure arrays.

- For MATLAB datastores and `TransformedDatastore`, `info` is a structure array that has fields with information about the datastore.
- For `CombinedDatastore`, `info` is a cell array of structure arrays. Each element of the cell array contains a structure with the relevant fields of the corresponding underlying datastore.

Information in the structure array depends on the type of the input datastore. The structure array can contain the following fields.

Field Name	Datastore Types	Description
`Filename`	All	`Filename` is a fully resolved path containing the path string, name of the file, and file extension. For `ImageDatastore` objects whose `ReadSize` property is greater than 1, `Filename` is a cell array of file names corresponding to each image.

Field Name	Datastore Types	Description
`FileSize`	All	Total file size, in bytes. For `ImageDatastore` objects whose `ReadSize` property is greater than 1, `FileSize` is a vector of file sizes corresponding to each image. For MAT-files, the value of `FileSize` depends on the type of the datastore. • `KeyValueDatastore` and `TallDatastore` — The `FileSize` field contains the total number of key-value pairs in the file. • `FileDatastore` — The `FileSize` field contains the total file size in bytes.
`FileType`	`KeyValueDatastore` only	The type of file from which data is read, either `'mat'` for MAT-files or `'seq'` for sequence files.
`Label`	`ImageDatastore` only	Image label name. If the `ReadSize` property is greater than 1, then `Label` is a vector of label names corresponding to each image. If the `Labels` property is empty, then `Label` is an empty cell array.
`NumCharactersRead`	`TabularTextDatastore` only	Number of characters read.
`NumDataRows`	`SpreadsheetDatastore` only	Vector containing number of rows read from each sheet.

Field Name	Datastore Types	Description
`Offset`	`KeyValueDatastore` and `TabularTextDatastore` only	Starting position of the read operation, in bytes. For MAT-files, `Offset` is the index of the first key and value read.
`SheetNames`	`SpreadsheetDatastore` only	Names of sheets read.
`SheetNumbers`	`SpreadsheetDatastore` only	Numbering associated with sheets read.

See Also

`datastore` | `hasdata` | `imread` | `readall`

Introduced in R2014b

readall

Package: `matlab.io`

Read all data in datastore

Syntax

```
data = readall(ds)
```

Description

`data = readall(ds)` returns all the data in the datastore specified by `ds`.

If all the data in the datastore does not fit in memory, then `readall` returns an error.

Examples

Read All Data in TabularTextDatastore

Create a datastore from the sample file `airlinesmall.csv`, which contains tabular data.

```
ds = tabularTextDatastore('airlinesmall.csv','TreatAsMissing','NA');
```

Modify the `SelectedVariableNames` property to specify the variables of interest.

```
ds.SelectedVariableNames = {'DepTime','ArrTime','ActualElapsedTime'};
```

Read all the data in the datastore.

```
T = readall(ds);
```

`readall` returns all the data in a table.

View information about the table.

```
T.Properties
```

```
ans = 
  TableProperties with properties:

             Description: ''
                UserData: []
          DimensionNames: {'Row'  'Variables'}
           VariableNames: {'DepTime'  'ArrTime'  'ActualElapsedTime'}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                RowNames: {}
        CustomProperties: No custom properties are set.
      Use addprop and rmprop to modify CustomProperties.
```

View a summary of the output table.

```
summary(T)
```

```
Variables:

    DepTime: 123523x1 double

        Values:

            Min             1
            Median          1335
            Max             2505
            NumMissing      2351

    ArrTime: 123523x1 double

        Values:

            Min             1
            Median          1522
            Max             2608
            NumMissing      2656

    ActualElapsedTime: 123523x1 double

        Values:
```

```
            Min                 11
            Median              102
            Max                 1650
            NumMissing          2657
```

Read All Data in KeyValueDatastore

Create a datastore from the sample file, `mapredout.mat`, which is the output file of the `mapreduce` function.

```
ds = datastore('mapredout.mat');
```

Read all the data in the datastore.

```
T = readall(ds);
```

View a summary of the output table.

```
summary(T)
```

```
Variables:

    Key: 29x1 cell array of character vectors

    Value: 29x1 cell
```

Read All Data in CombinedDatastore

Create a datastore that maintains parity between the pair of images of the underlying datastores. For instance, create two separate image datastores, and then create a combined datastore representing the two underlying datastores.

Create an image datastore `imds1` representing a collection of three images.

```
imds1 = imageDatastore({'street1.jpg','street2.jpg','peppers.png'});
```

Create a second datastore `imds2` by transforming the images of `imds1` to grayscale and then downsizing the images.

```
imds2 = transform(imds1,@(x) imresize(rgb2gray(x),0.5));
```

Create a combined datastore from `imds1` and `imds2`.

```
imdsCombined = combine(imds1,imds2);
```

Read all of the data from the combined datastore. The output is a 3-by-2 cell array. The two columns represent all of the read data from the two underlying datastores `imds1` and `imds2`, respectively.

```
dataOut = readall(imdsCombined)
```

```
dataOut=3×2 cell
    {480x640x3 uint8}    {240x320 uint8}
    {480x640x3 uint8}    {240x320 uint8}
    {384x512x3 uint8}    {192x256 uint8}
```

Input Arguments

`ds` — Input datastore
datastore

Input datastore. You can use these datastores as input to the `readall` method.

- MATLAB datastores — Datastores created using MATLAB `datastore` functions. For example, create a datastore for a collection of images using `ImageDatastore`. For a complete list of datastores, see “Select Datastore for File Format or Application”.
- Combined and transformed datastores — Datastores created using the `combine` and `transform` functions.
- Custom datastores — Datastores created using the custom datastore framework. See “Develop Custom Datastore”.

Output Arguments

`data` — All data in the datastore
table | cell array

All data in the datastore, returned as a table or a cell array depending on the type of `ds`.

Type of Datastore	Data type of data	Description
`TabularTextDatastore` and `SpreadsheetDatastore`	Table	The `SelectedVariableNames` property determines the table variables.
`ImageDatastore`	Cell array	Each element in the cell array contains the image data for one image. The `readall` function supports all image types supported by the `imread` function. For more information on the supported image types, see `imread`.
`KeyValueDatastore`	Table	The table variable names are `Key` and `Value`.
`FileDatastore`	Cell array	Each element in the cell array contains the data read from one file using the custom read function specified by the `ReadFcn` property.
`TransformedDatastore`	Varies	The output is the same as the output returned by the underlying datastore specified by the `UnderlyingDatastore` property. For example, if the underlying datastore is an image datastore, then `data` is returned as a cell array where each element in the cell array contains the image data for one image.

Type of Datastore	Data type of data	Description
`CombinedDatastore`	Cell array	Each column of the cell array contains the result from calling `readall` on the corresponding underlying datastore specified by the `UnderlyingDatastores` property. If the number of subsets of data in the underlying datastores differs, then `readall` only returns data while all underlying datastores have data. For example, suppose a combined datastore has two underlying datastores, one with *m* subsets of data and one with *n* subsets of data, where *m* > *n*. The output is a cell array with two columns and *n* rows.

See Also

`datastore` | `hasdata` | `read`

Introduced in R2014b

readimage

Package: `matlab.io.datastore`

Read specified image from datastore

Syntax

```
img = readimage(imds,I)
[img,fileinfo] = readimage(imds,I)
```

Description

`img = readimage(imds,I)` reads the `I`th image file from the datastore `imds` and returns the image data `img`. The size and data type of the `img` array depends on the image formats of the files in the datastore. The image formats supported by `readimage` function are those formats supported by `imread`. For more information on the supported formats, see `imread`.

`[img,fileinfo] = readimage(imds,I)` also returns a struct `fileinfo` that contains two file information fields:

- `Filename` — Name of the file from which the image is read.
- `FileSize` — Size of the file in bytes.

Examples

Read Specified Image

Create an `ImageDatastore` object containing two images, and then read and view the second image.

```
imds = imageDatastore({'street1.jpg','street2.jpg'});
img = readimage(imds,2);
imshow(img)
```

File Information

Create a datastore object containing two images and read the second image.

```
imds = imageDatastore({'street1.jpg','street2.jpg'});
[img,info] = readimage(imds,2);
```

The `info` structure contains the following information for the second image: `Filename`, `FileSize`, and `Label`. To display the filename and path, type `info.Filename` in the command window. To display the file size, type `info.FileSize`.

Input Arguments

`imds` — Input datastore
`ImageDatastore` object

Input datastore. You can use the `imageDatastore` function or the `datastore` function to create a datastore object from your data.

`I` — Image file index
positive integer

Image file index, specified as a positive integer.

Data Types: `double`

Output Arguments

`img` — Image data
multidimensional array of integers

Image data, returned as a multidimensional array of integers. The size of `img` depends on the image format:

- m-by-n array for grayscale images
- m-by-n-by-3 array for truecolor images
- m-by-n-by-4 array for CMYK Tiff images

The `readimage` function supports the image formats that `imread` supports. For more information on the supported formats, see `imread`.

Data Types: `double`

`fileinfo` — File information
structure array

File information, returned as a structure array containing two fields:

- `Filename` — Name of the file from which the image is read.

- `FileSize` — Size of the file in bytes.

Data Types: `struct`

See Also

`ImageDatastore` | `datastore` | `imread` | `read` | `readall`

Introduced in R2015b

reset

Package: `matlab.io`

Reset datastore to initial state

Syntax

```
reset(ds)
```

Description

`reset(ds)` resets the datastore specified by `ds` to the state where no data has been read from it. Resetting allows re-reading from the same datastore.

Examples

Reset Datastore to Initial State

Create a datastore from the sample file, `mapredout.mat`, which is the output file of the `mapreduce` function.

```
ds = datastore('mapredout.mat');
```

Read the first key-value pair.

```
T = read(ds);
```

Reset the datastore to the state where no data has been read from it.

```
reset(ds)
```

Input Arguments

ds — Input datastore
datastore

Input datastore. You can use these datastores as input to the `reset` method.

- MATLAB datastores — Datastores created using MATLAB `datastore` functions. For example, create a datastore for a collection of images using `ImageDatastore`. For a complete list of datastores, see “Select Datastore for File Format or Application”.
- Combined or transformed datastores — Datastores created using the `combine` and `transform` functions.
- Custom datastores — Datastores created using the custom datastore framework. See “Develop Custom Datastore”.

See Also

`datastore`

Introduced in R2014b

sheetnames

Package: `matlab.io.datastore`

Query sheet names from datastore

Syntax

```
names = sheetnames(ssds,filename)
names = sheetnames(ssds,I)
```

Description

`names = sheetnames(ssds,filename)` returns the sheet names from a specified file in the datastore `ssds`.

`names = sheetnames(ssds,I)` returns the sheet names from a specified file index.

Examples

Query Sheet Names

Create a datastore containing the file `airlinesmall_subset.xlsx`.

```
ssds = spreadsheetDatastore('airlinesmall_subset.xlsx')
```

```
ssds = 

  SpreadsheetDatastore with properties:

                      Files: {
                             ' ...\matlab\toolbox\matlab\demos\airlinesmall_subset.xlsx
                             }
   AlternateFileSystemRoots: {}
                     Sheets: ''
```

```
                     Range: ''
           ReadFailureRule: 'error'
               MaxFailures: Inf

  Sheet Format Properties:
            NumHeaderLines: 0
         ReadVariableNames: true
             VariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
             VariableTypes: {'double', 'double', 'double' ... and 26 more}

  Properties that control the table returned by preview, read, readall:
      SelectedVariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
      SelectedVariableTypes: {'double', 'double', 'double' ... and 26 more}
                   ReadSize: 'file'
```

Query the sheet names of the first (and only) file in the datastore.

```
sheetnames(ssds,1)
```

```
ans =

  13×1 string array

    "1996"
    "1997"
    "1998"
    "1999"
    "2000"
    "2001"
    "2002"
    "2003"
    "2004"
    "2005"
    "2006"
    "2007"
```

```
"2008"
```

Input Arguments

ssds — Input datastore
SpreadsheetDatastore object

Input datastore, specified as a SpreadsheetDatastore object. Use the spreadsheetDatastore function or the datastore function to create a datastore object from your data.

filename — File in datastore
character vector | string scalar

File in datastore, specified as a character vector or string scalar containing the name of the file to query.

The value of filename must match exactly the file name contained in the Files property of the datastore. To ensure that the file names match exactly, specify filename using ssds.Files{N} where N is the index of the file in the Files property.

Example: names = sheetnames(ssds,ssds.Files{1}) returns sheet names contained in the first file specified in the Files property of the datastore ssds.

Data Types: char | string

I — File index
positive integer

File index, specified as a positive integer. I cannot exceed the number of files in ssds.

Data Types: double

Output Arguments

names — Sheet names
string array

Sheet names, returned as a string array containing the sheet names in the specified file.

Data Types: `string`

See Also

`spreadsheetDatastore`

Introduced in R2016a

shuffle

Package: `matlab.io.datastore`

Shuffle files in ImageDatastore

Syntax

```
imdsrand = shuffle(imds)
```

Description

`imdsrand = shuffle(imds)` returns an `ImageDatastore` object containing a random ordering of the files from `imds`.

Examples

Shuffle Files

Create an `ImageDatastore` object `imds`. Shuffle the files to create a new datastore containing the same files in random order.

```
imds = imageDatastore(fullfile(matlabroot, 'toolbox', 'matlab', {'demos','imagesci'}),
```

```
imds = 

  ImageDatastore with properties:

      Files: {
             ' ...\matlab\toolbox\matlab\demos\cloudCombined.jpg';
             ' ...\matlab\toolbox\matlab\demos\example.tif';
             ' ...\matlab\toolbox\matlab\demos\landOcean.jpg'
              ... and 5 more
             }
     Labels: [demos; demos; demos ... and 5 more categorical]
```

```
    ReadFcn: @readDatastoreImage
```

```
imdsrand = shuffle(imds)
```

```
imdsrand = 

  ImageDatastore with properties:

      Files: {
             ' ...\matlab\toolbox\matlab\demos\street2.jpg';
             ' ...\matlab\toolbox\matlab\demos\landOcean.jpg';
             ' ...\matlab\toolbox\matlab\imagesci\corn.tif'
              ... and 5 more
             }
     Labels: [demos; demos; imagesci ... and 5 more categorical]
    ReadFcn: @readDatastoreImage
```

Input Arguments

`imds` — Input datastore
`ImageDatastore` object

Input datstore, specified as an `ImageDatastore` object. To create an `ImageDatstore` from your image data, use the `imageDatastore` function.

Output Arguments

`imdsrand` — Output datastore
`ImageDatastore` object

Output datastore, returned as an `ImageDatastore` object containing randomly ordered files from `imds`.

See Also

`ImageDatastore` | `datastore`

Introduced in R2016a

splitEachLabel

Package: `matlab.io.datastore`

Split ImageDatastore labels by proportions

Syntax

```
[imds1,imds2] = splitEachLabel(imds,p)
[imds1,...,imdsM] = splitEachLabel(imds,p1,...,pN)
___ = splitEachLabel(___,'randomized')
___ = splitEachLabel(___,Name,Value)
```

Description

`[imds1,imds2] = splitEachLabel(imds,p)` splits the image files in `imds` into two new datastores, `imds1` and `imds2`. The new datastore `imds1` contains the first `p` files from each label and `imds2` contains the remaining files from each label. `p` can be either a number between 0 and 1 indicating the percentage of the files from each label to assign to `imds1`, or an integer indicating the absolute number of files from each label to assign to `imds1`.

`[imds1,...,imdsM] = splitEachLabel(imds,p1,...,pN)` splits the datastore into `N+1` new datastores. The first new datastore `imds1` contains the first `p1` files from each label, the next new datastore `imds2` contains the next `p2` files, and so on. If `p1,...,pN` represent numbers of files, then their sum must be no more than the number of files in the smallest label in the original datastore `imds`.

`___ = splitEachLabel(___,'randomized')` randomly assigns the specified proportion of files from each label to the new datastores.

`___ = splitEachLabel(___,Name,Value)` specifies the properties of the new datastores using one or more name-value pair arguments. For example, you can specify which labels to split with `'Include','labelname'`.

Examples

Split Labels by Percentage

Create an `ImageDatastore` object and label each image according to the name of the folder it is in. The resulting label names are `demos` and `imagesci`.

```
imds = imageDatastore(fullfile(matlabroot, 'toolbox', 'matlab', {'demos','imagesci'}),
'LabelSource', 'foldernames', 'FileExtensions', {'.jpg', '.png', '.tif'});
```

```
imds.Labels
```

```
ans =

     demos
     demos
     demos
     demos
     demos
     demos
     imagesci
     imagesci
```

Create two new datastores from the files in `imds`. The first datastore `imds60` contains the first 60% of files with the `demos` label and the first 60% of files with the `imagesci` label. The second datastore `imds40` contains the remaining 40% of files from each label. If the percentage applied to a label does not result in a whole number of files, `splitEachLabel` rounds down to the nearest whole number.

```
[imds60,imds40] = splitEachLabel(imds,0.6)
```

```
imds60 =

  ImageDatastore with properties:

      Files: {
             ' ...\matlab\toolbox\matlab\demos\cloudCombined.jpg';
             ' ...\matlab\toolbox\matlab\demos\example.tif';
             ' ...\matlab\toolbox\matlab\demos\landOcean.jpg'
              ... and 2 more
```

```
             }
     Labels: [demos; demos; demos ... and 2 more categorical]
    ReadFcn: @readDatastoreImage


imds40 =

  ImageDatastore with properties:

      Files: {
             ' ...\matlab\toolbox\matlab\demos\street1.jpg';
             ' ...\matlab\toolbox\matlab\demos\street2.jpg';
             ' ...\matlab\toolbox\matlab\imagesci\peppers.png'
             }
     Labels: [demos; demos; imagesci]
    ReadFcn: @readDatastoreImage
```

Split Labels by Number of Files

Create an `ImageDatastore` object and label each image according to the name of the folder it is in. The resulting label names are `demos` and `imagesci`.

```
imds = imageDatastore(fullfile(matlabroot, 'toolbox', 'matlab', {'demos','imagesci'}),
'LabelSource', 'foldernames', 'FileExtensions', {'.jpg', '.png', '.tif'});
```

```
imds.Labels
```

```
ans =

     demos
     demos
     demos
     demos
     demos
     demos
     imagesci
     imagesci
```

Create two new datastores from the files in `imds`. The first datastore `imds1` contains the first file with the `demos` label and the first file with the `imagesci` label. The second datastore `imds2` contains the remaining files from each label.

```
[imds1,imds2] = splitEachLabel(imds,1)
```

```
imds1 = 

  ImageDatastore with properties:

      Files: {
             ' ...\matlab\toolbox\matlab\demos\cloudCombined.jpg';
             ' ...\matlab\toolbox\matlab\imagesci\corn.tif'
             }
     Labels: [demos; imagesci]
    ReadFcn: @readDatastoreImage


imds2 = 

  ImageDatastore with properties:

      Files: {
             ' ...\matlab\toolbox\matlab\demos\example.tif';
             ' ...\matlab\toolbox\matlab\demos\landOcean.jpg';
             ' ...\matlab\toolbox\matlab\demos\ngc6543a.jpg'
              ... and 3 more
             }
     Labels: [demos; demos; demos ... and 3 more categorical]
    ReadFcn: @readDatastoreImage
```

Split Labels Several Ways by Percentage

Create an `ImageDatastore` object and label each image according to the name of the folder it is in. The resulting label names are `demos` and `imagesci`.

```
imds = imageDatastore(fullfile(matlabroot, 'toolbox', 'matlab', {'demos','imagesci'}),
'LabelSource', 'foldernames', 'FileExtensions', {'.jpg', '.png', '.tif'});

imds.Labels
```

```
ans =

     demos
     demos
     demos
     demos
     demos
     demos
     imagesci
     imagesci
```

Create three new datastores from the files in `imds`. The first datastore `imds60` contains the first 60% of files with the `demos` label and the first 60% of files with the `imagesci` label. The second datastore `imds10` contains the next 10% of files from each label. The third datastore `imds30` contains the remaining 30% of files from each label. If the percentage applied to a label does not result in a whole number of files, `splitEachLabel` rounds down to the nearest whole number.

```
[imds60, imds10, imds30] = splitEachLabel(imds,0.6,0.1)
```

```
imds60 =

  ImageDatastore with properties:

      Files: {
             ' ...\matlab\toolbox\matlab\demos\cloudCombined.jpg';
             ' ...\matlab\toolbox\matlab\demos\example.tif';
             ' ...\matlab\toolbox\matlab\demos\landOcean.jpg'
              ... and 2 more
             }
     Labels: [demos; demos; demos ... and 2 more categorical]
    ReadFcn: @readDatastoreImage


imds10 =

  ImageDatastore with properties:

      Files: {
             ' ...\matlab\toolbox\matlab\demos\street1.jpg'
             }
```

```
     Labels: demos
    ReadFcn: @readDatastoreImage


imds30 =

  ImageDatastore with properties:

      Files: {
             ' ...\matlab\toolbox\matlab\demos\street2.jpg';
             ' ...\matlab\toolbox\matlab\imagesci\peppers.png'
             }
     Labels: [demos; imagesci]
    ReadFcn: @readDatastoreImage
```

Split Labels Several Ways by Number of Files

Create an `ImageDatastore` object and label each image according to the name of the folder it is in. The resulting label names are `demos` and `imagesci`.

```
imds = imageDatastore(fullfile(matlabroot, 'toolbox', 'matlab', {'demos','imagesci'}),
'LabelSource', 'foldernames', 'FileExtensions', {'.jpg', '.png', '.tif'});
```

```
imds.Labels
```

```
ans =

     demos
     demos
     demos
     demos
     demos
     demos
     imagesci
     imagesci
```

Create three new datastores from the files in `imds`. The first datastore `imds1` contains the first file with the `demos` label and the first file with the `imagesci` label. The second datastore `imds2` contains the next file from each label. The third datastore `imds3` contains the remaining files from each label.

```
[imds1, imds2, imds3] = splitEachLabel(imds,1,1)
```

```
imds1 = 

  ImageDatastore with properties:

      Files: {
             ' ...\matlab\toolbox\matlab\demos\cloudCombined.jpg';
             ' ...\matlab\toolbox\matlab\imagesci\corn.tif'
             }
     Labels: [demos; imagesci]
    ReadFcn: @readDatastoreImage


imds2 = 

  ImageDatastore with properties:

      Files: {
             ' ...\matlab\toolbox\matlab\demos\example.tif';
             ' ...\matlab\toolbox\matlab\imagesci\peppers.png'
             }
     Labels: [demos; imagesci]
    ReadFcn: @readDatastoreImage


imds3 = 

  ImageDatastore with properties:

      Files: {
             ' ...\matlab\toolbox\matlab\demos\landOcean.jpg';
             ' ...\matlab\toolbox\matlab\demos\ngc6543a.jpg';
             ' ...\matlab\toolbox\matlab\demos\street1.jpg'
              ... and 1 more
             }
     Labels: [demos; demos; demos ... and 1 more categorical]
    ReadFcn: @readDatastoreImage
```

Randomly Split Labels

Create an `ImageDatastore` object and label each image according to the name of the folder it is in. The resulting label names are `demos` and `imagesci`.

```
imds = imageDatastore(fullfile(matlabroot, 'toolbox', 'matlab', {'demos','imagesci'}),
'LabelSource', 'foldernames', 'FileExtensions', {'.jpg', '.png', '.tif'});
```

```
imds.Labels
```

```
ans =

     demos
     demos
     demos
     demos
     demos
     demos
     imagesci
     imagesci
```

Create two new datastores from the files in `imds` by randomly drawing from each label. The first datastore `imds1` contains one random file with the `demos` label and one random file with the `imagesci` label. The second datastore `imds2` contains the remaining files from each label.

```
[imds1, imds2] = splitEachLabel(imds,1,'randomized')
```

```
imds1 =

  ImageDatastore with properties:

      Files: {
             ' ...\matlab\toolbox\matlab\demos\street2.jpg';
             ' ...\matlab\toolbox\matlab\imagesci\corn.tif'
             }
     Labels: [demos; imagesci]
    ReadFcn: @readDatastoreImage


imds2 =
```

```
  ImageDatastore with properties:

      Files: {
             ' ...\matlab\toolbox\matlab\demos\cloudCombined.jpg';
             ' ...\matlab\toolbox\matlab\demos\example.tif';
             ' ...\matlab\toolbox\matlab\demos\landOcean.jpg'
              ... and 3 more
             }
     Labels: [demos; demos; demos ... and 3 more categorical]
    ReadFcn: @readDatastoreImage
```

Include and Exclude Specified Labels

Create an `ImageDatastore` object and label each image according to the name of the folder it is in. The resulting label names are `demos` and `imagesci`.

```
imds = imageDatastore(fullfile(matlabroot, 'toolbox', 'matlab', {'demos','imagesci'}),
'LabelSource', 'foldernames', 'FileExtensions', {'.jpg', '.png', '.tif'});
```

```
imds.Labels
```

```
ans =

     demos
     demos
     demos
     demos
     demos
     demos
     imagesci
     imagesci
```

Create two new datastores from the files in `imds`, including only the files with the `demos` label. The first datastore `imds60` contains the first 60% of files with the `demos` label and the second datastore `imds40` contains the remaining 40% of files with the `demos` label.

```
[imds60, imds40] = splitEachLabel(imds,0.6,'Include','demos')
```

```
imds60 =
```

```
  ImageDatastore with properties:

      Files: {
             ' ...\matlab\toolbox\matlab\demos\cloudCombined.jpg';
             ' ...\matlab\toolbox\matlab\demos\example.tif';
             ' ...\matlab\toolbox\matlab\demos\landOcean.jpg'
              ... and 1 more
             }
     Labels: [demos; demos; demos ... and 1 more categorical]
    ReadFcn: @readDatastoreImage


imds40 =

  ImageDatastore with properties:

      Files: {
             ' ...\matlab\toolbox\matlab\demos\street1.jpg';
             ' ...\matlab\toolbox\matlab\demos\street2.jpg'
             }
     Labels: [demos; demos]
    ReadFcn: @readDatastoreImage
```

Equivalently, you can split only the `demos` label by excluding the `imagesci` label.

```
[imds60, imds40] = splitEachLabel(imds,0.6,'Exclude','imagesci')
```

```
imds60 =

  ImageDatastore with properties:

      Files: {
             ' ...\matlab\toolbox\matlab\demos\cloudCombined.jpg';
             ' ...\matlab\toolbox\matlab\demos\example.tif';
             ' ...\matlab\toolbox\matlab\demos\landOcean.jpg'
              ... and 1 more
             }
     Labels: [demos; demos; demos ... and 1 more categorical]
    ReadFcn: @readDatastoreImage


imds40 =
```

```
  ImageDatastore with properties:

      Files: {
             ' ...\matlab\toolbox\matlab\demos\street1.jpg';
             ' ...\matlab\toolbox\matlab\demos\street2.jpg'
             }
     Labels: [demos; demos]
    ReadFcn: @readDatastoreImage
```

Input Arguments

`imds` — Input datastore
`ImageDatastore` object

Input datastore, specified as an `ImageDatastore` object. To create an `ImageDatstore` from your image data, use the `imageDatastore` function.

`p` — Proportion of files to split
scalar in interval (0,1) | positive integer scalar

Proportion of files to split, specified as a scalar in the interval (0,1) or a positive integer scalar.

- If `p` is in the interval (0,1), then it represents the percentage of the files from each label to assign to `imds1`. If `p` does not result in a whole number of files, then `splitEachLabel` rounds down to the nearest whole number.
- If `p` is an integer, then it represents the absolute number of files from each label to assign to `imds1`. There must be at least `p` files associated with each label.

Data Types: `double`

`p1,...,pN` — List of proportions
scalars in interval (0,1) | positive integer scalars

List of proportions, specified as scalars in the interval (0,1) or positive integer scalars. If the proportions are in the interval (0,1), then they represent the percentage of the files from each label to assign to the output datastores. If the proportions are integers, then they indicate the absolute number of files from each label to assign to the output datastores. When the proportions represent percentages, their sum must be no more than

1. When the proportions represent numbers of files, there must be enough files associated with each label to satisfy each proportion.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[imds1 imds2] = splitEachLabel(imds,0.5,'Exclude','demos')`

`Include` — Labels to include
categorical, logical, or numeric vector | cell array of character vectors | string array

Labels to include, specified as the comma-separated pair consisting of `'Include'` and a vector, cell array, or string array of label names with the same type as the `Labels` property. Each name must match one of the labels in the `Labels` property of the datastore.

Data Types: `char` | `cell` | `string`

`Exclude` — Labels to exclude
categorical, logical, or numeric vector | cell array of character vectors | string array

Labels to exclude, specified as the comma-separated pair consisting of `'Exclude'` and a vector, cell array, or string array of label names with the same type as the `Labels` property. Each name defines a label associated with the datastore and must match the names in `Labels`. This option cannot be used with the `'Include'` option.

Data Types: `char` | `cell` | `string`

Output Arguments

`imds1,imds2` — Output datastores
`ImageDatastore` objects

Output datastores, returned as `ImageDatastore` objects. `imds1` contains the specified proportion of files from each label in `imds`, and `imds2` contains the remaining files.

`imds1,...,imdsM` — List of output datastores
`ImageDatastore` objects

List of output datastores, returned as `ImageDatastore` objects. The number of elements in the list is one more than the number of listed proportions. Each of the new datastores contains the proportion of each label in `imds` defined by `p1,...,pN`. Any files left over are assigned to the Mth datastore.

See Also

`ImageDatastore` | `countEachLabel` | `datastore`

Introduced in R2016a

spreadsheetDatastore

Datastore for spreadsheet files

Description

Use a `SpreadsheetDatastore` object to manage large collections of spreadsheet files where the collection does not necessarily fit in memory. You can create a `SpreadsheetDatastore` object using the `spreadsheetDatastore` function, specify its properties, and then import the data using object functions.

Creation

Syntax

```
ssds = spreadsheetDatastore(location)
ssds = spreadsheetDatastore(location,Name,Value)
```

Description

`ssds = spreadsheetDatastore(location)` creates a spreadsheet datastore from the collection of data specified by `location`.

`ssds = spreadsheetDatastore(location,Name,Value)` specifies additional parameters and properties for `ssds` using one or more name-value pair arguments. For example, `spreadsheetDatastore(location,'FileExtensions', {'.xlsx','.xls'})` specifies which files to include in the datastore depending on the file extensions.

Input Arguments

`location` — Files or folders to include in datastore
path | `DsFileSet` object

Files or folders included in the datastore, specified as a path or a `DsFileSet` object.

- path — Specify the path as a character vector, cell array of character vectors, string scalar, or a string array, containing the location of files or folders that are local or remote.
  - Local files or folders — Specify `location` as a local path to files or folders. If the files are not in the current folder, then local path must specify full or relative paths. Files within subfolders of the specified folder are not automatically included in the datastore. You can use the wildcard character (*) when specifying the local path. This character specifies that the datastore include all matching files or all files in the matching folders.
  - Remote files or folders — Specify `location` to be the full paths of the files or folders as a uniform resource locator (URL) of the form `hdfs:///path_to_file`. For more information, see “Work with Remote Data”.
- `DsFileSet` object — You also can specify `location` as a `DsFileSet` object. For more information, see `matlab.io.datastore.DsFileSet`.

When `location` represents a folder, the datastore includes only supported file formats and ignores any other format. To specify a custom list of file extensions to include in your datastore, see the `FileExtensions` property.

The `spreadsheetDatastore` function supports these extensions: `.xls`, `.xlsx`, `.xlsm`, `.xltx`, and `.xltm`.

Example: `'file1.xlsx'`

Example: `'../dir/data/file1.xlsx'`

Example: `{'C:\dir\data\file1.xlsx','C:\dir\data\file2.xlsx'}`

Example: `'s3://bucketname/path_to_files/*.xls'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `ssds = spreadsheetDatastore('C:\dir\spreadsheetdata','FileExtensions',{'.xls','.xlsm'})`

`IncludeSubfolders` — Subfolder inclusion flag
`true` or `false` | 0 or 1

Subfolder inclusion flag, specified as the comma-separated pair consisting of `'IncludeSubfolders'` and `true`, `false`, 0, or 1. Specify `true` to include all files and subfolders within each folder or `false` to include only the files within each folder.

If you do not specify `'IncludeSubfolders'`, then the default value is `false`.

Example: `'IncludeSubfolders',true`

Data Types: `logical` | `double`

`FileExtensions` — Spreadsheet file extensions
character vector | cell array of character vectors | string scalar | string array

Spreadsheet file extensions, specified as the comma-separated pair consisting of `'FileExtensions'` and a character vector, cell array of character vectors, string scalar, or string array.

- If you do not specify `'FileExtensions'`, then `spreadsheetDatastore` automatically includes all files with `.xls`, `.xlsx`, `.xlsm`, `.xltx`, and `.xltm` extensions in the specified path.
- If you want to include spreadsheets with non-standard file extensions in the `SpreadsheetDatastore`, then specify those extensions explicitly.
- If you want to create a `SpreadsheetDatastore` for files without any extensions, then specify `'FileExtensions'` as an empty character vector, `''`.

Example: `'FileExtensions',''`

Example: `'FileExtensions','.xls'`

Example: `'FileExtensions',{'.xlsx','.xlsm'}`

Data Types: `char` | `cell` | `string`

`AlternateFileSystemRoots` — Alternate file system root paths
string vector | cell array

Alternate file system root paths, specified as the comma-separated pair consisting of `'AlternateFileSystemRoots'` and a string vector or a cell array. Use `'AlternateFileSystemRoots'` when you create a datastore on a local machine, but need to access and process the data on another machine (possibly of a different operating

system). Also, when processing data using the Parallel Computing Toolbox and the MATLAB Parallel Server, and the data is stored on your local machines with a copy of the data available on different platform cloud or cluster machines, you must use `'AlternateFileSystemRoots'` to associate the root paths.

- To associate a set of root paths that are equivalent to one another, specify `'AlternateFileSystemRoots'` as a string vector. For example,

  ```
  ["Z:\datasets","/mynetwork/datasets"]
  ```

- To associate multiple sets of root paths that are equivalent for the datastore, specify `'AlternateFileSystemRoots'` as a cell array containing multiple rows where each row represents a set of equivalent root paths. Specify each row in the cell array as either a string vector or a cell array of character vectors. For example:

  - Specify `'AlternateFileSystemRoots'` as a cell array of string vectors.

    ```
    {["Z:\datasets", "/mynetwork/datasets"];...
     ["Y:\datasets", "/mynetwork2/datasets","S:\datasets"]}
    ```

  - Alternatively, specify `'AlternateFileSystemRoots'` as a cell array of cell array of character vectors.

    ```
    {{'Z:\datasets','/mynetwork/datasets'};...
     {'Y:\datasets', '/mynetwork2/datasets','S:\datasets'}}
    ```

The value of `'AlternateFileSystemRoots'` must satisfy these conditions:

- Contains one or more rows, where each row specifies a set of equivalent root paths.
- Each row specifies multiple root paths and each root path must contain at least two characters.
- Root paths are unique and are not subfolders of one another.
- Contains at least one root path entry that points to the location of the files.

For more information, see “Set Up Datastore for Processing on Different Machines or Clusters”.

Example: `["Z:\datasets","/mynetwork/datasets"]`

Data Types: `string` | `cell`

TextType — Output data type of text variables
`'char'` (default) | `'string'`

Output data type of text variables, specified as the comma-separated pair consisting of `'TextType'` and either `'char'` or `'string'`.

- If the output table from the `read`, `readall`, or `preview` functions contains text variables, then `'TextType'` specifies the data type of those variables.
- If `'TextType'` is `'char'`, then the output is a cell array of character vectors.
- If `'TextType'` is `'string'`, then the output has type `string`.

Data Types: `char` | `string`

In addition to these name-value pairs, you also can specify any of the properties on this page as name-value pairs. Except for the `Files` property.

Properties

`SpreadsheetDatastore` properties describe the format of the files in a datastore object, and control how the data is read from the datastore. You can specify the value of `SpreadsheetDatastore` properties using name-value pair arguments when you create the datastore object. Except for the `Files` property. To view or modify a property after creating the object, use the dot notation.

File Properties

`Files` — Files included in datastore

cell array of character vectors | string array

Files included in the datastore, resolved as a cell array of character vectors or a string array, where each character vector or string is a full path to a file. The `location` argument in the `spreadsheetDatastore` and `datastore` functions define these files.

The first file specified in the cell array determines the variable names and format information for all files in the datastore.

Example: `{'C:\dir\data\file1.xls';'C:\dir\data\file2.xls'}`

Data Types: `cell` | `string`

`NumHeaderLines` — Number of lines to skip

`0` (default) | positive integer

Number of lines to skip at the beginning of each sheet when reading, specified as a positive integer. When you also specify the `Range` property, `NumHeaderLines` is the number of lines to skip at the beginning of the specified block of data.

Data Types: `double`

`Sheets` — Sheets in files
`''` (default) | sheet names | sheet indices

Sheets in files, specified as a character vector, cell array of character vectors, string scalar, or string array containing sheet names, or as a numeric vector of sheet indices. The empty character vector `''` indicates that all sheets in the files are included.

Example: `{'sheet1','sheet7'}`

Example: `[3 5 7]`

Data Types: `char` | `cell` | `string` | `double`

`Range` — Row and column bounds
`''` (default) | character vector | string scalar

Row and column bounds, specified as a character vector or string scalar that defines a rectangular block of data in the sheets. The empty character vector `''` indicates that the bounds are the beginning of the file and the end of the data.

Example: `'B1:T7'`

Example: `'A:C'`

Data Types: `char` | `string`

`ReadVariableNames` — Indicator for reading variable names
`true` (default) | `false` | `1` | `0`

Indicator for reading the first row of the first file in the datastore as variable names, specified as either `true` (`1`) or `false` (`0`).

- If `true`, then the first nonheader row of the first file determines the variable names for the data.
- If `false`, then the first nonheader row of the first file contains the first row of data. Default variable names are assigned as `Var1`, `Var2`, and so on.

Data Types: `logical` | `double`

`PreserveVariableNames` — Flag to preserve variable names
`false` (default) | `true`

Flag to preserve variable names, specified as the comma-separated pair consisting of `PreserveVariableNames` and either `true`, or `false`.

- `true` — Preserve variable names that are not valid MATLAB identifiers such as variable names that include spaces and non-ASCII characters.
- `false` — Convert invalid variable names (as determined by the `isvarname` function) to valid MATLAB identifiers.

Starting in R2019b, variable names and row names can include any characters, including spaces and non-ASCII characters. Also, they can start with any characters, not just letters. Variable and row names do not have to be valid MATLAB identifiers (as determined by the `isvarname` function). To preserve these variable names and row names, set `PreserveVariableNames` to `true`.

`VariableNames` — Names of variables
character vector | cell array of character vectors | string scalar | string array

Names of variables in the datastore, specified as a character vector, cell array of character vectors, string scalar, or string array. Specify the variable names in the order in which they appear in the files. If you do not specify the variable names, the datastore detects them from the first nonheader line in the first file. You can specify `VariableNames` with a character vector or string scalar, however the datastore converts and stores the property value to a cell array of character vectors. When modifying the `VariableNames` property, the number of new variable names must match the number of original variable names.

To support invalid MATLAB identifiers as variable names, such as varible names containing spaces and non-ASCII characters, set the `PreserveVariableNames` parameter to `true`.

If `ReadVariableNames` is `false`, then `VariableNames` defaults to `{'Var1','Var2', ...}`.

Example: `{'Time','Date','Quantity'}`

Data Types: `char` | `cell` | `string`

`VariableTypes` — Variable types
`'double'` | `'char'` | `'string'` | `'categorical'` | `'datetime'`

Variable types, specified as `'double'`, `'char'`, `'string'`, `'categorical'`, or `'datetime'`, which indicates the type of each variable when reading the data.

The list of variable types corresponds with the variables in `VariableNames`. Types `double`, `char`, and `datetime` can be automatically detected from the data. You can specify `VariableTypes` as a character vector or string scalar, however `spreadsheetDatastore` automatically converts it to a cell array of character vectors or a string array.

Example: `{'char','categorical'}`

Data Types: `char` | `cell` | `string`

Properties for preview, read, readall Table

`SelectedVariableNames` — Variables to read

character vector | cell array of character vectors | string scalar | string array

Variables to read from the file, specified as a character vector, cell array of character vectors, string scalar, or string array. Each character vector or string contains the name of one variable. You can specify the variable names in any order. You can specify `SelectedVariableNames` with a character vector or string scalar, however `spreadsheetDatastore` automatically converts it to a cell array of character vectors or a string array.

To support invalid MATLAB identifiers as variable names, such as varible names containing spaces and non-ASCII characters, set the `PreserveVariableNames` parameter to `true`.

Example: `{'Var3','Var7','Var4'}`

Data Types: `char` | `cell` | `string`

`SelectedVariableTypes` — Selected variable types

`'double'` | `'char'` | `'string'` | `'categorical'` | `'datetime'`

Selected variable types, specified as `'double'`, `'char'`, `'string'`, `'categorical'`, or `'datetime'`, which indicates the type of each selected variable when reading the data. The list of variable types corresponds with the variables in `SelectedVariableNames`. Types `double`, `char`, and `datetime` can be automatically detected from the data. You can specify `SelectedVariableTypes` as a character vector or string scalar, however it is automatically converted to a cell array of character vectors or string array.

Example: `{'double','datetime'}`

Data Types: `char` | `cell` | `string`

ReadSize — Amount of data to read
`'file'` (default) | `'sheet'` | positive integer scalar

Amount of data to read in a call to the `read` function, specified as `'file'` or `'sheet'`, or as a positive integer scalar.

- If `ReadSize` is `'file'`, then each call to `read` reads all the data one file at a time.
- If `ReadSize` is `'sheet'`, then each call to `read` reads all the data one sheet at a time.
- If `ReadSize` is a positive integer, then each call to `read` reads the rows specified by `ReadSize`, or fewer if it reaches the end of the data.

When you change `ReadSize` from an integer scalar to `'file'` or `'sheet'`, or conversely, the datastore resets using the `reset` function.

Data Types: `char` | `string` | `double`

Object Functions

hasdata	Determine if data is available to read
numpartitions	Number of datastore partitions
partition	Partition a datastore
preview	Subset of data in datastore
read	Read data in datastore
readall	Read all data in datastore
reset	Reset datastore to initial state
sheetnames	Query sheet names from datastore
transform	Transform datastore
combine	Combine data from multiple datastores

Examples

Create SpreadsheetDatastore Object

```
ssds = datastore('airlinesmall_subset.xlsx')
```

```
ssds =
```

```
  SpreadsheetDatastore with properties:

                     Files: {
                            ' ...\matlab\toolbox\matlab\demos\airlinesmall_subset.xlsx
                            }
                    Sheets: ''
                     Range: ''

  Sheet Format Properties:
            NumHeaderLines: 0
         ReadVariableNames: true
             VariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
             VariableTypes: {'double', 'double', 'double' ... and 26 more}

  Properties that control the table returned by preview, read, readall:
     SelectedVariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
     SelectedVariableTypes: {'double', 'double', 'double' ... and 26 more}
                  ReadSize: 'file'
```

Read Spreadsheet File

Create a `SpreadsheetDatastore` object containing the file `airlinesmall_subset.xlsx`.

```
ssds = spreadsheetDatastore('airlinesmall_subset.xlsx')
```

```
ssds =

  SpreadsheetDatastore with properties:

                     Files: {
                            ' ...\matlab\toolbox\matlab\demos\airlinesmall_subset.xlsx
                            }
                    Sheets: ''
                     Range: ''

  Sheet Format Properties:
            NumHeaderLines: 0
         ReadVariableNames: true
             VariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
```

```
              VariableTypes: {'double', 'double', 'double' ... and 26 more}

  Properties that control the table returned by preview, read, readall:
      SelectedVariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
      SelectedVariableTypes: {'double', 'double', 'double' ... and 26 more}
                   ReadSize: 'file'
```

Display the sheet names for the file. The file contains one sheet per year.

```
sheetnames(ssds,1)
```

```
ans =

  Columns 1 through 7

    '1996'    '1997'    '1998'    '1999'    '2000'    '2001'    '2002'

  Columns 8 through 13

    '2003'    '2004'    '2005'    '2006'    '2007'    '2008'
```

Specify the variable `FlightNum` in the second sheet as the data of interest, and preview the first eight rows.

```
ssds.Sheets = 2;
ssds.SelectedVariableNames = 'FlightNum';
preview(ssds)
```

```
ans =

    FlightNum
    _________

    1014
    1201
     702
    1184
    1310
    1759
    1242
    1558
```

Read only the first three rows of variables `DepTime` and `ArrTime` in the first sheet.

```
ssds.ReadSize = 3;
ssds.Sheets = 1;
ssds.SelectedVariableNames = {'DepTime','ArrTime'};
read(ssds)
```

```
ans =

    DepTime    ArrTime
    _______    _______

    2117       2305
    1252       1511
    1441       1708
```

Read all of sheets four, five, and six.

```
ssds.Sheets = 4:6;
readall(ssds);
```

Alternatives

You also can create a `SpreadsheetDatastore` object using the `datastore` function. For example, `ds = datastore(location,'Type','spreadsheet')` creates a datastore from a collection of files specified by `location`.

See Also

`datastore` | `mapreduce` | `readtable` | `tall`

Topics

"Getting Started with Datastore"
"Work with Remote Data"
"Set Up Datastore for Processing on Different Machines or Clusters"

Introduced in R2016a

subset

Package: `matlab.io.datastore`

Create subset of datastore or file-set

Syntax

```
subds = subset(ds,indices)
```

Description

`subds = subset(ds,indices)` returns a subset containing files corresponding to `indices`. The subset `subds` is of the same type as the input.

- if the input `ds` is a datastore, then the output `outds` is a datastore of the same type.
- if the input `ds` is a `DsFileSet` object, then the output `subds` is also a `DsFileSet` object.

Examples

Create Subset of ImageDatastore

Make an image datastore object and then create a subset of that image datastore.

Create an image datastore `imds` for all the image files in a sample folder. Then, display the `Files` property of `imds`.

```
folders = fullfile(matlabroot,'toolbox','matlab',{'demos','imagesci'});
exts = {'.jpg','.png','.tif'};
imds = imageDatastore(folders,'LabelSource','foldernames','FileExtensions',exts);
imds.Files
```

```
ans =
```

```
  8×1 cell array

    {'...\matlab\toolbox\matlab\demos\cloudCombined.jpg'}
    {'...\matlab\toolbox\matlab\demos\example.tif'      }
    {'...\matlab\toolbox\matlab\demos\landOcean.jpg'    }
    {'...\matlab\toolbox\matlab\demos\ngc6543a.jpg'     }
    {'...\matlab\toolbox\matlab\demos\street1.jpg'      }
    {'...\matlab\toolbox\matlab\demos\street2.jpg'      }
    {'...\matlab\toolbox\matlab\imagesci\corn.tif'      }
    {'...\matlab\toolbox\matlab\imagesci\peppers.png'   }
```

Create a subset datastore `subimds` that contains the first four files of `imds` and examine the `Files` property of `subimds`.

```
indices = 1:4;
subimds = subset(imds,indices);
subimds.Files
```

```
ans =

  4×1 cell array

    {'...\matlab\toolbox\matlab\demos\cloudCombined.jpg'}
    {'...\matlab\toolbox\matlab\demos\example.tif'      }
    {'...\matlab\toolbox\matlab\demos\landOcean.jpg'    }
    {'...\matlab\toolbox\matlab\demos\ngc6543a.jpg'     }
```

Create Subset Datastore with Randomly Selected Files

Make an image datastore, and then create subset datastore containing only a specified percentage of files, randomly selected from the original datastore.

Create `imageDatastore` for all the image files in a sample folder and display the `Files` property. This datastore contains 8 files.

```
folders = fullfile(matlabroot,'toolbox','matlab',{'demos','imagesci'});
exts = {'.jpg','.png','.tif'};
imds = imageDatastore(folders,'LabelSource','foldernames','FileExtensions',exts);
imds.Files
```

```
ans =

  8×1 cell array
```

```
    {'...\matlab\toolbox\matlab\demos\cloudCombined.jpg'}
    {'...\matlab\toolbox\matlab\demos\example.tif'      }
    {'...\matlab\toolbox\matlab\demos\landOcean.jpg'    }
    {'...\matlab\toolbox\matlab\demos\ngc6543a.jpg'     }
    {'...\matlab\toolbox\matlab\demos\street1.jpg'      }
    {'...\matlab\toolbox\matlab\demos\street2.jpg'      }
    {'...\matlab\toolbox\matlab\imagesci\corn.tif'      }
    {'...\matlab\toolbox\matlab\imagesci\peppers.png'   }
```

Create a set of indices that represents randomly selected subset containing `60%` of the files.

```
nFiles = length(imds.Files);
RandIndices = randperm(nFiles);
nSixtyPercent = round(0.6*nFiles);
indices = RandIndices(1:nSixtyPercent)
```

```
indices =

     8     6     4     5     1
```

Create a subset datastore `submids` using `indices` and examine its `Files` property.

```
subimds = subset(imds,indices);
subimds.Files
```

```
ans =

  5×1 cell array

    {'...\matlab\toolbox\matlab\imagesci\peppers.png'   }
    {'...\matlab\toolbox\matlab\demos\street2.jpg'      }
    {'...\matlab\toolbox\matlab\demos\ngc6543a.jpg'     }
    {'...\matlab\toolbox\matlab\demos\street1.jpg'      }
    {'...\matlab\toolbox\matlab\demos\cloudCombined.jpg'}
```

Input Arguments

ds — Input datastore or file-set

`ImageDatastore` object | `DsFileSet` object

Input datastore or file-set, specified as `ImageDatastore` object or a `DsFileSet` object.

- To create an `ImageDatastore` object, use `imageDatastore`.
- To create a `DsFileSet` object, use `matlab.io.datastore.DsFileSet`.

`indices` — Indices of files to include in subset
vector of indices | logical vector

Indices of files to include in subset, specified as a vector of indices or a logical vector.

- The vector of indices must contain the indices of files to include in the subset `subds`.
- The logical vector must be of the same length as the number of files in the input `ds`. The `subset` method creates a subset `subds` containing files corresponding to the elements in the logical vector that have a value of `true`.

Elements of `indices` must be unique.

Data Types: `double` | `logical`

See Also

`DsFileSet` | `ImageDatastore` | `splitEachLabel`

Introduced in R2019a

tabularTextDatastore

Datastore for tabular text files

Description

Use a `TabularTextDatastore` object to manage large collections of text files containing column-oriented or tabular data where the collection does not necessarily fit in memory. Tabular data is data that is arranged in a rectangular fashion with each row having the same number of entries. You can create a `TabularTextDatastore` object using the `tabularTextDatastore` function, specify its properties, and then import and process the data using object functions.

Creation

Syntax

```
ttds = tabularTextDatastore(location)
ttds = tabularTextDatastore(location,Name,Value)
```

Description

`ttds = tabularTextDatastore(location)` creates a datastore from the collection of data specified by `location`.

`ttds = tabularTextDatastore(location,Name,Value)` specifies additional parameters and properties for `ttds` using one or more name-value pair arguments. For example, `tabularTextDatastore(location,'FileExtensions', {'.txt','.csv'})` creates a datastore from only the files in `location` with extensions `.txt` and `.csv`.

Input Arguments

`location` — Files or folders included in datastore

path | `DsFileSet` object

Files or folders included in the datastore, specified as a path or a `DsFileSet` object.

- path — Specify the path as a character vector, cell array of character vectors, string scalar, or a string array, containing the location of files or folders that are local or remote.
  - Local files or folders — Specify `location` as a local path to files or folders. If the files are not in the current folder, then local path must specify full or relative paths. Files within subfolders of the specified folder are not automatically included in the datastore. You can use the wildcard character (*) when specifying the local path. This character specifies that the datastore include all matching files or all files in the matching folders.
  - Remote files or folders — Specify `location` to be the full paths of the files or folders as a uniform resource locator (URL) of the form `hdfs:///path_to_file`. For more information, see “Work with Remote Data”.
- `DsFileSet` object — You also can specify `location` as a `DsFileSet` object. For more information, see `matlab.io.datastore.DsFileSet`.

When `location` represents a folder, the datastore includes only supported file formats and ignores any other format. To specify a custom list of file extensions to include in your datastore, see the `FileExtensions` property.

The `tabularTextDatastore` function supports these extensions: `.txt`, `.csv`, `.dat`, `.dlm`, `.asc`, `.text`, or no extension.

Example: `'file1.csv'`

Example: `'../dir/data/file1'`

Example: `{'C:\dir\data\file1.csv','C:\dir\data\file2.dat'}`

Example: `'C:\dir\data\*.text'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `ttds = tabularTextDatastore('C:\dir\textdata','FileExtensions',{'.csv','.txt'})`

`IncludeSubfolders` — Subfolder inclusion flag
`true` or `false` | 0 or 1

Subfolder inclusion flag, specified as the comma-separated pair consisting of `'IncludeSubfolders'` and `true`, `false`, 0, or 1. Specify `true` to include all files and subfolders within each folder or `false` to include only the files within each folder.

When you do not specify `'IncludeSubfolders'`, then the default value is `false`.

Example: `'IncludeSubfolders',true`

Data Types: `logical` | `double`

`FileExtensions` — Text file extensions
character vector | cell array of character vectors | string scalar | string array

Text file extensions, specified as the comma-separated pair consisting of `'FileExtensions'` and a character vector, cell array of character vectors, string scalar, or string array. The specified extensions do not require a supported format. If you want to include unsupported extensions, then specify all extensions. Use empty quotes `''` to represent files without extensions.

Example: `'FileExtensions','.txt'`

Example: `'FileExtensions',{'.text','.csv'}`

Data Types: `char` | `cell` | `string`

`AlternateFileSystemRoots` — Alternate file system root paths
string vector | cell array

Alternate file system root paths, specified as the comma-separated pair consisting of `'AlternateFileSystemRoots'` and a string vector or a cell array. Use `'AlternateFileSystemRoots'` when you create a datastore on a local machine, but need to access and process the data on another machine (possibly of a different operating system). Also, when processing data using the Parallel Computing Toolbox and the MATLAB Parallel Server, and the data is stored on your local machines with a copy of the data available on different platform cloud or cluster machines, you must use `'AlternateFileSystemRoots'` to associate the root paths.

- To associate a set of root paths that are equivalent to one another, specify `'AlternateFileSystemRoots'` as a string vector. For example,

  ```
  ["Z:\datasets","/mynetwork/datasets"]
  ```

- To associate multiple sets of root paths that are equivalent for the datastore, specify `'AlternateFileSystemRoots'` as a cell array containing multiple rows where each row represents a set of equivalent root paths. Specify each row in the cell array as either a string vector or a cell array of character vectors. For example:

  - Specify `'AlternateFileSystemRoots'` as a cell array of string vectors.

    ```
    {["Z:\datasets", "/mynetwork/datasets"];...
     ["Y:\datasets", "/mynetwork2/datasets","S:\datasets"]}
    ```

  - Alternatively, specify `'AlternateFileSystemRoots'` as a cell array of cell array of character vectors.

    ```
    {{'Z:\datasets','/mynetwork/datasets'};...
     {'Y:\datasets', '/mynetwork2/datasets','S:\datasets'}}
    ```

The value of `'AlternateFileSystemRoots'` must satisfy these conditions:

- Contains one or more rows, where each row specifies a set of equivalent root paths.
- Each row specifies multiple root paths and each root path must contain at least two characters.
- Root paths are unique and are not subfolders of one another.
- Contains at least one root path entry that points to the location of the files.

For more information, see “Set Up Datastore for Processing on Different Machines or Clusters”.

Example: `["Z:\datasets","/mynetwork/datasets"]`

Data Types: `string` | `cell`

`TextType` — Output data type of text variables
`'char'` (default) | `'string'`

Output data type of text variables, specified as the comma-separated pair consisting of `'TextType'` and either `'char'` or `'string'`. If the output table from the `read`, `readall`, or `preview` functions contains text variables, then `'TextType'` specifies the data type of those variables for `TabularTextDatastore`. If `'TextType'` is `'char'`,

then the output is a cell array of character vectors. If `'TextType'` is `'string'`, then the output has type `string`.

Data Types: `char` | `string`

`DatetimeType` — Type for imported date and time data
`'datetime'` (default) | `'text'`

Type for imported date and time data, specified as the comma-separated pair consisting of `'DatetimeType'` and one of these values: `'datetime'` or `'text'`.

Value	Type for Imported Date and Time Data
`'datetime'`	MATLAB `datetime` data type For more information, see `datetime`.
`'text'`	If `'DatetimeType'` is specified as `'text'`, then the type for imported date and time data depends on the value specified in the `'TextType'` property: • If `'TextType'` is `'char'`, then the `tabularTextdatastore` imports dates as a cell array of character vectors. • If `'TextType'` is `'string'`, then the `tabularTextdatastore` imports dates as an array of strings.

If the specified `TextscanFormats` property contains a `%D`, then the `tabularTextdatastore` ignores the value specified in `DatetimeType`.

Example: `'DatetimeType','datetime'`

Data Types: `char` | `string`

`DurationType` — Output data type of duration data
`'duration'` (default) | `'text'`

Output data type of duration data, specified as the comma-separated pair consisting of `'DurationType'` and either `'duration'` or `'text'`.

Value	Type for Imported Duration Data
`'duration'`	MATLAB `duration` data type For more information, see `duration`.
`'text'`	If `'DurationType'` is specified as `'text'`, then the type for imported duration data depends on the value specified in the `'TextType'` parameter: • If `'TextType'` is `'char'`, then the importing function returns duration data as a cell array of character vectors. • If `'TextType'` is `'string'`, then the importing function returns duration data as an array of strings.

Data Types: `char` | `string`

`DatetimeLocale` — Locale to interpret dates
`'en_US'` (default) | character vector | string scalar

Locale to interpret dates, specified as a character vector or string scalar. The `DatetimeLocale` value determines how the importing function interprets text that represents dates and times.

When specifying the `DatetimeLocale`, use the form *xx_YY*, where *xx* is a lowercase ISO 639-1 two-letter code that specifies a language, and *YY* is an uppercase ISO 3166-1 alpha-2 code that specifies a country.

This table lists some common values for the locale.

Locale	Language	Country
`'de_DE'`	German	Germany
`'en_GB'`	English	United Kingdom
`'en_US'`	English	United States
`'es_ES'`	Spanish	Spain
`'fr_FR'`	French	France
`'it_IT'`	Italian	Italy
`'ja_JP'`	Japanese	Japan

Locale	Language	Country
'ko_KR'	Korean	Korea
'nl_NL'	Dutch	Netherlands
'zh_CN'	Chinese (simplified)	China

Note The `Locale` value determines how input values are interpreted. The display format and language is specified by the **Locale** option in the **Datetime format** section of the Preferences panel. To change the default datetime locale, see "Set Command Window Preferences".

Data Types: `char` | `string`

In addition to these name-value pairs, you also can specify the properties on this page as name-value pairs, with the exception of the `Files` property.

Properties

`TabularTextDatastore` properties describe the files associated with a `TabularTextDatastore` object. Specifically, the properties describe the format of the data in the files and control how the data should be read from the datastore. When you create a `TabularTextDatastore` object, the datastore function uses the first file in the `Files` property to determine the values of the properties. With the exception of the `Files` property, you can specify the value of `TabularTextDatastore` properties using name-value pair arguments when you create the datastore object. To view or modify a property after creating the object, use the dot notation:

```
ds = datastore('airlinesmall.csv');
ds.TreatAsMissing = 'NA';
ds.MissingValue = 0;
```

File Properties

`Files` — Files included in datastore

cell array of character vectors | string array

Files included in the datastore, resolved as a cell array of character vectors or a string array, where each character vector or string is a full path to a file. The `location` argument in the `tabularTextDatastore` and `datastore` functions define these files.

The first file specified by the `Files` property determines the variable names and format information for all files in the datastore.

When you change the value of this property, the datastore function reevaluates the values of the `TabularTextDatastore` properties.

Example: `{'C:\dir\data\mydata1.csv';'C:\dir\data\mydata2.csv'}`

Data Types: `cell` | `string`

`FileEncoding` — File encoding

`'UTF-8'` (default) | `'US-ASCII''Macintosh'` | ...

File encoding, specified as a character vector or a string scalar like one of these values.

`'IBM866'`	`'ISO-8859-1'`	`'windows-847'`
`'KOI8-R'`	`'ISO-8859-2'`	`'windows-1250'`
`'KOI8-U'`	`'ISO-8859-3'`	`'windows-1251'`
`'Macintosh'`	`'ISO-8859-4'`	`'windows-1252'`
`'US-ASCII'`	`'ISO-8859-5'`	`'windows-1253'`
`'UTF-8'`	`'ISO-8859-6'`	`'windows-1254'`
	`'ISO-8859-7'`	`'windows-1255'`
	`'ISO-8859-8'`	`'windows-1256'`
	`'ISO-8859-9'`	`'windows-1257'`
	`'ISO-8859-11'`	`'windows-1258'`
	`'ISO-8859-13'`	
	`'ISO-8859-15'`	

If each file in the datastore fits into memory, then `FileEncoding` also can be one of these values.

`'Big5'`	`'EUC-KR'`	`'GB18030'`	`'Shift_JIS'`
`'Big5-HKSCS'`	`'EUC-JP'`	`'GB2312'`	`'windows-949'`
`'CP949'`	`'EUC-TW'`	`'GBK'`	

When you change the value of this property, the datastore function reevaluates the values of the `TabularTextDatastore` properties.

Data Types: `char` | `string`

`ReadVariableNames` — Read variable names
`true` | `false`

Read variable names, specified as a logical `true` or `false`.

- If unspecified, the `tabularTextDatastore` function detects the presence of variable names automatically.
- If `true`, then the first nonheader row of the first file determines the variable names for the data.
- If `false`, then the first nonheader row of the first file contains the first row of data. The data is assigned default variable names, `Var1`, `Var2`, and so on.

When you change the value of this property, the datastore function reevaluates the values of the `TabularTextDatastore` properties.

Data Types: `logical`

`PreserveVariableNames` — Flag to preserve variable names
`false` (default) | `true`

Flag to preserve variable names, specified as the comma-separated pair consisting of `PreserveVariableNames` and either `true`, or `false`.

- `true` — Preserve variable names that are not valid MATLAB identifiers such as variable names that include spaces and non-ASCII characters.
- `false` — Convert invalid variable names (as determined by the `isvarname` function) to valid MATLAB identifiers.

Starting in R2019b, variable names and row names can include any characters, including spaces and non-ASCII characters. Also, they can start with any characters, not just letters. Variable and row names do not have to be valid MATLAB identifiers (as determined by the `isvarname` function). To preserve these variable names and row names, set `PreserveVariableNames` to `true`.

`VariableNames` — Names of variables
cell array of character vectors | string array

Names of variables in the datastore, specified as a cell array of character vectors or a string array. Specify the variable names in the order in which they appear in the files. If you do not specify the variable names, they are detected from the first nonheader line in

the first file of the datastore. When modifying the `VariableNames` property, the number of new variable names must match the number of original variable names.

To support invalid MATLAB identifiers as variable names, such as varible names containing spaces and non-ASCII characters, set the `PreserveVariableNames` parameter to `true`.

If `ReadVariableNames` is `false`, then `VariableNames` defaults to `{'Var1','Var2', ...}`.

Example: `{'Time','Name','Quantity'}`

Data Types: `cell` | `string`

Text Format Properties

`NumHeaderLines` — Number of lines to skip
non-negative integer

Number of lines to skip at the beginning of the file, specified as a nonnegative integer. If unspecified, the `tabularTextDatastore` function detects the number of lines to skip automatically.

The `tabularTextDatastore` function ignores the specified number of header lines before reading the variable names or data.

When you change the value of this property, the datastore function reevaluates the values of the `TabularTextDatastore` properties.

Data Types: `double`

`Delimiter` — Field delimiter characters
character vector | cell array of character vectors | string scalar | string array

Field delimiter characters, specified as a character vector, cell array of character vectors, string scalar, or string array. Specify multiple delimiters in a cell array of character vectors or a string array. If unspecified, the `tabularTextDatastore` function detects the delimiter automatically.

Example: `'|'`

Example: `{';','*'}`

Repeated delimiter characters in a file are interpreted as separate delimiters with empty fields between them. If unspecified, the read function detects the delimiter automatically by default.

When you specify one of the following escape sequences as a delimiter, it is converted to the corresponding control character.

`\b`	Backspace
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\\`	Backslash (`\`)

When you change the value of this property, the datastore function reevaluates the values of the `TabularTextDatastore` properties.

Data Types: `char` | `cell` | `string`

`RowDelimiter` — Row delimiter character
`\r\n` (default) | character vector | string scalar

Row delimiter character, specified as a character vector or string scalar that must be either a single character or one of `'\r'`, `'\n'`, or `'\r\n'`.

When you change the value of this property, the datastore function reevaluates the values of the `TabularTextDatastore` properties.

Example: `':'`

Data Types: `char` | `string`

`TreatAsMissing` — Numeric values to treat as missing
`''` (default) | character vector | cell array of character vectors | string scalar | string array

Numeric values to treat as missing values, specified as a single character vector, cell array of character vectors, string scalar, or string array. Values specified as `TreatAsMissing` are substituted with the value defined in the `MissingValue` property. For instance, if `MissingValue` is defined to be a `NaN`, and the `TreatAsMissing` is specified as `'NA'`. Then, in the imported data, all occurrences of `'NA'` are replaced by `NaN`.

This option only applies to numeric fields. Also, this property is equivalent to the `TreatAsEmpty` name-value pair argument for the `textscan` function.

When you change the value of this property, the datastore function reevaluates the values of the `TabularTextDatastore` properties.

Example: `'NA'`

Example: `'-99'`

Example: `{'-',''}`

Data Types: `char` | `cell` | `string`

`MissingValue` — Value for missing numeric fields
`NaN` (default) | scalar

Value for missing numeric fields in delimited text files, specified as a scalar. This property is equivalent to the `EmptyValue` name-value pair argument for the `textscan` function.

Data Types: `double`

Advanced Text Format Properties

`TextscanFormats` — Data field format
cell array of character vectors | string array

Data field format, specified as a cell array of character vectors or a string array, where each character vector or string contains one conversion specifier.

When you specify or modify the `TextscanFormats` property, you can use the same conversion specifiers that the `textscan` function accepts for the `formatSpec` argument. Valid values for `TextscanFormats` include conversion specifiers that skip fields using an asterisk (*) character and ones that skip literal text. The number of conversion specifiers must match the number of variables in the `VariableNames` property.

- If the value of `TextscanFormats` includes conversion specifiers that skip fields using asterisk characters (*), then the value of the `SelectedVariableNames` property automatically updates. MATLAB uses the `%*q` conversion specifier to skip fields omitted by the `SelectedVariableNames` property and treats the field contents as literal character vectors. For fixed-width files, indicate a skipped field using the appropriate conversion specifier along with the field width. For example, `%*52c` skips a field that contains 52 characters.

- If you do not specify a value for `TextscanFormats`, then `datastore` determines the format of the data fields by scanning text from the first nonheader line in the first file of the datastore.

Example: `{'%s','%s','%f'}`

Data Types: `cell` | `string`

`ExponentCharacters` — Exponent characters
`'eEdD'` (default) | character vector | string scalar

Exponent characters, specified as a character vector or string scalar. The default exponent characters are `e`, `E`, `d`, and `D`.

Data Types: `char` | `string`

`CommentStyle` — Style of comments
`''` (default) | character vector | cell array of character vectors | string scalar | string array

Style of comments in the file, specified as a character vector, cell array of character vectors, string scalar, or string array.

For example, specify `'%'` to ignore characters following the text on the same line. Specify `{'/*','*/'}` to ignore characters between the text.

When reading from a `TabularTextDatastore`, the `read` function checks for comments only at the start of each field, not within a field.

When you change the value of this property, the datastore function reevaluates the values of the `TabularTextDatastore` properties.

Example: `'CommentStyle',{'/*', '*/'}`

Data Types: `char` | `cell` | `string`

`Whitespace` — White-space characters
`' \b\t'` (default) | character vector | string scalar

White-space characters, specified as a character vector or a string scalar of one or more characters.

When you specify one of the following escape sequences as any white-space character, the `datastore` function converts that sequence to the corresponding control character.

`\b`	Backspace
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\\`	Backslash (\)

When you change the value of this property, the datastore function reevaluates the values of the `TabularTextDatastore` properties.

Example: `' \b\t'`

Data Types: `char` | `string`

`MultipleDelimitersAsOne` — Multiple delimiter handling
`0 (false)` (default) | `1 (true)`

Multiple delimiter handling, specified as either `true` or `false`. If `true`, then `datastore` treats consecutive delimiters as a single delimiter. Repeated delimiters separated by white-space are also treated as a single delimiter.

When you change the value of this property, the datastore function reevaluates the values of the `TabularTextDatastore` properties.

Properties That Control Table Returned by preview, read, readall

`SelectedVariableNames` — Variables to read
cell array of character vectors | string array

Variables to read from the file, specified as a cell array of character vectors or a string array, where each character vector or string contains the name of one variable. You can specify the variable names in any order.

To support invalid MATLAB identifiers as variable names, such as varible names containing spaces and non-ASCII characters, set the `PreserveVariableNames` parameter to `true`.

Example: `{'Var3','Var7','Var4'}`

Data Types: `cell` | `string`

`SelectedFormats` — Formats of selected variables
cell array of character vectors | string array

Formats of the selected variables to read, specified as a cell array of character vectors or a string array, where each character vector or string contains one conversion specifier. The variables to read are indicated by the `SelectedVariableNames` property. The number of character vectors or strings in `SelectedFormats` must match the number of variables to read.

You can use the same conversion specifiers that the `textscan` function accepts, including specifiers that skip literal text. However, you cannot use a conversion specifier that skips a field. That is, the conversion specifier cannot include an asterisk character (*).

Example: `{'%d','%d'}`

Data Types: `cell` | `string`

`ReadSize` — Amount of data to read
20000 (default) | positive scalar | `'file'`

Amount of data to read in a call to the `read` function, specified as a positive scalar or `'file'`.

- If `ReadSize` is a positive integer, then each call to `read` reads at most `ReadSize` rows.
- If `ReadSize` is `'file'`, then each call to `read` reads all of the data in one file.

When you change `ReadSize` from a numeric scalar to `'file'` or vice versa, MATLAB resets the datastore to the state where no data has been read from it.

Data Types: `double` | `char` | `string`

`TextType` — Output data type of text variables
`'char'` (default) | `'string'`

Output data type of text variables, specified as `'char'` or `'string'`. `TextType` specifies the data type of text variables formatted with `%s`, `%q`, or `[...]`.

- If `TextType` is `'char'`, then the output is a cell array of character vectors.
- If `TextType` is `'string'`, then the output has type `string`.

Data Types: `char` | `string`

Object Functions

hasdata	Determine if data is available to read
numpartitions	Number of datastore partitions
partition	Partition a datastore
preview	Subset of data in datastore
read	Read data in datastore
readall	Read all data in datastore
reset	Reset datastore to initial state
transform	Transform datastore
combine	Combine data from multiple datastores

Examples

Create TabularTextDatastore Object

Create a `TabularTextDatastore` object containing the text file `airlinesmall.csv`.

```
ttds = tabularTextDatastore('airlinesmall.csv')
```

```
ttds = 

  TabularTextDatastore with properties:

                      Files: {
                             ' ...\matlab\toolbox\matlab\demos\airlinesmall.csv'
                             }
               FileEncoding: 'UTF-8'
          ReadVariableNames: true
              VariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}

  Text Format Properties:
            NumHeaderLines: 0
                 Delimiter: ','
              RowDelimiter: '\r\n'
            TreatAsMissing: ''
              MissingValue: NaN

  Advanced Text Format Properties:
           TextscanFormats: {'%f', '%f', '%f' ... and 26 more}
```

```
        ExponentCharacters: 'eEdD'
              CommentStyle: ''
                Whitespace: ' \b\t'
    MultipleDelimitersAsOne: false

  Properties that control the table returned by preview, read, readall:
      SelectedVariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
            SelectedFormats: {'%f', '%f', '%f' ... and 26 more}
                   ReadSize: 20000 rows
```

Select Variables to Read

Create a datastore from the sample file `airlinesmall.csv`, which contains tabular data.

```
ds = tabularTextDatastore('airlinesmall.csv','TreatAsMissing','NA');
```

View the variables in the datastore.

```
ds.VariableNames
```

```
ans = 1x29 cell array
  Columns 1 through 5

    {'Year'}    {'Month'}    {'DayofMonth'}    {'DayOfWeek'}    {'DepTime'}

  Columns 6 through 9

    {'CRSDepTime'}    {'ArrTime'}    {'CRSArrTime'}    {'UniqueCarrier'}

  Columns 10 through 13

    {'FlightNum'}    {'TailNum'}    {'ActualElapsedTime'}    {'CRSElapsedTime'}

  Columns 14 through 18

    {'AirTime'}    {'ArrDelay'}    {'DepDelay'}    {'Origin'}    {'Dest'}

  Columns 19 through 22

    {'Distance'}    {'TaxiIn'}    {'TaxiOut'}    {'Cancelled'}
```

```
  Columns 23 through 25

    {'CancellationCode'}    {'Diverted'}    {'CarrierDelay'}

  Columns 26 through 28

    {'WeatherDelay'}    {'NASDelay'}    {'SecurityDelay'}

  Column 29

    {'LateAircraftDelay'}
```

Modify the `SelectedVariableNames` property to specify the variables of interest.

```
ds.SelectedVariableNames = {'Year','Month','Cancelled'};
```

Alternatively, you can specify the variables of interest when you create the datastore.

```
ds = tabularTextDatastore('airlinesmall.csv','TreatAsMissing','NA','SelectedVariableNar
```

Specify Format of Data to Read

Create a datastore from the sample file `airlinesmall.csv`, which contains tabular data.

```
ds = tabularTextDatastore('airlinesmall.csv','TreatAsMissing','NA');
```

Specify the variables of interest.

```
ds.SelectedVariableNames = {'Year','Month','UniqueCarrier'};
```

View the `SelectedFormats` property.

```
ds.SelectedFormats
```

```
ans = 1x3 cell array
    {'%f'}    {'%f'}    {'%q'}
```

The `SelectedFormats` property indicates that the `Year` and `Month` variables will be interpreted as columns of floating-point values, and the `UniqueCarrier` variable will be interpreted as a column of text.

Specify that the first two variables should be read as signed integers, and the third variable should be read as a categorical value by modifying the `SelectedFormats` property.

```
ds.SelectedFormats = {'%d','%d','%C'};
```

Preview the data.

```
T = preview(ds)
```

```
T=8×3 table
    Year    Month    UniqueCarrier
    ____    _____    _____________

    1987     10           PS
    1987     10           PS
    1987     10           PS
    1987     10           PS
    1987     10           PS
    1987     10           PS
    1987     10           PS
    1987     10           PS
```

Limitations

- Datetime data containing day, month, or time zone names in a language foreign to the `en_US` locale are not supported. For unrecognized datetime formats, specify the format using the `TextscanFormats` parameter.

Alternatives

You also can create a `TabularTextDatastore` object using the `datastore` function. For example, `ds = datastore(location,'Type','tabulartext')` creates a datastore from a collection of files specified by `location`.

See Also

`datastore` | `mapreduce` | `readtable` | `reset` | `tall` | `textscan`

Topics

"Read and Analyze Large Tabular Text File"
"Getting Started with Datastore"
"Work with Remote Data"
"Set Up Datastore for Processing on Different Machines or Clusters"

Introduced in R2014b

TallDatastore

Datastore for checkpointing `tall` arrays

Description

`TallDatastore` objects are for recreating `tall` arrays from binary files written to disk by the `write` function. You can use the object to recreate the original `tall` array, or you can access and manage the data by specifying `TallDataStore` properties and using the object functions.

Creation

Create `TallDatastore` objects using the `datastore` function. For example, `tds = datastore(location,'Type','tall')` creates a datastore from a collection of files specified by `location`.

Properties

`Files` — Files included in datastore

character vector | cell array of character vectors | string scalar | string array

Files included in the datastore, resolved as a character vector, cell array of character vectors, string scalar, or string array, where each character vector or string is a full path to a file.

The `location` argument of the `datastore` function defines the `Files` property when the datastore is created. The `location` argument contains full paths to files on a local file system, a network file system, or a supported remote location such as Amazon S3, Windows Azure Blob Storage, and HDFS. For more information, see “Work with Remote Data”.

The files must be either MAT-files or Sequence files generated by the `write` function.

Example: `{'C:\dir\data\file1.ext';'C:\dir\data\file2.ext'}`

Example: `{'s3://bucketname/path_to_files/your_file01.ext';'s3://bucketname/path_to_files/your_file02.ext'}`

Data Types: `char` | `cell` | `string`

`FileType` — File type
`'mat'` | `'seq'`

File type, specified as either `'mat'` for MAT-files or `'seq'` for sequence files. By default, the type of file in the provided location determines the `FileType`.

Data Types: `char` | `string`

`ReadSize` — Maximum number of data rows to read
positive integer

Maximum number of data rows to read in a call to the `read` or `preview` functions, specified as a positive integer. When the `datastore` function creates a `TallDatastore`, it determines and assigns the best possible value for `ReadSize`.

`AlternateFileSystemRoots` — Alternate file system root paths
string vector | cell array

Alternate file system root paths, specified as the comma-separated pair consisting of `'AlternateFileSystemRoots'` and a string vector or a cell array. Use `'AlternateFileSystemRoots'` when you create a datastore on a local machine, but need to access and process the data on another machine (possibly of a different operating system). Also, when processing data using the Parallel Computing Toolbox and the MATLAB Parallel Server, and the data is stored on your local machines with a copy of the data available on different platform cloud or cluster machines, you must use `'AlternateFileSystemRoots'` to associate the root paths.

- To associate a set of root paths that are equivalent to one another, specify `'AlternateFileSystemRoots'` as a string vector. For example,

  ```
  ["Z:\datasets","/mynetwork/datasets"]
  ```

- To associate multiple sets of root paths that are equivalent for the datastore, specify `'AlternateFileSystemRoots'` as a cell array containing multiple rows where each row represents a set of equivalent root paths. Specify each row in the cell array as either a string vector or a cell array of character vectors. For example:

  - Specify `'AlternateFileSystemRoots'` as a cell array of string vectors.

```
{["Z:\datasets", "/mynetwork/datasets"];...
 ["Y:\datasets", "/mynetwork2/datasets","S:\datasets"]}
```

- Alternatively, specify `'AlternateFileSystemRoots'` as a cell array of cell array of character vectors.

```
{{'Z:\datasets','/mynetwork/datasets'};...
 {'Y:\datasets', '/mynetwork2/datasets','S:\datasets'}}
```

The value of `'AlternateFileSystemRoots'` must satisfy these conditions:

- Contains one or more rows, where each row specifies a set of equivalent root paths.
- Each row specifies multiple root paths and each root path must contain at least two characters.
- Root paths are unique and are not subfolders of one another.
- Contains at least one root path entry that points to the location of the files.

For more information, see “Set Up Datastore for Processing on Different Machines or Clusters”.

Example: `["Z:\datasets","/mynetwork/datasets"]`

Data Types: `string` | `cell`

Object Functions

hasdata	Determine if data is available to read
numpartitions	Number of datastore partitions
partition	Partition a datastore
preview	Subset of data in datastore
read	Read data in datastore
readall	Read all data in datastore
reset	Reset datastore to initial state
transform	Transform datastore
combine	Combine data from multiple datastores

Examples

Recreate `tall` Arrays from Files Saved Using `write` Function

Use `TallDatastore` objects to reconstruct tall arrays directly from files on disk rather than re-executing all of the commands that produced the tall array. Create a tall array and save it to disk using `write` function. Retrieve the `tall` array using `datastore` and then convert it back to `tall`.

Create a simple tall double.

```
t = tall(rand(500,1))
```

```
t =

  500×1 tall double column vector

    0.8147
    0.9058
    0.1270
    0.9134
    0.6324
    0.0975
    0.2785
    0.5469
    :
    :
```

Save the results to a new folder named `ExampleData` on the `C:\` disk. (You might want to specify a different write location, especially if you are not using a Windows® computer.)

```
location = 'C:\ExampleData';
write(location, t);
```

```
Writing tall data to folder C:\ExampleData
Evaluating tall expression using the Parallel Pool 'local':
Evaluation completed in 0 sec
```

To recover the `tall` array that was written to disk, first create a new datastore that references the same directory. Then convert the datastore into a `tall` array.

```
tds = datastore(location);
t1 = tall(tds)
```

```
t1 =

  M×1 tall double column vector

    0.8147
    0.9058
    0.1270
    0.9134
    0.6324
    0.0975
    0.2785
    0.5469
    :
    :
```

See Also

datastore | write

Topics

"Set Up Datastore for Processing on Different Machines or Clusters"

Introduced in R2016b

transform

Package: `matlab.io`

Transform datastore

Syntax

```
dsnew = transform(ds,@fcn)
dsnew = transform(ds,@fcn,'IncludeInfo',IncludeInfo)
```

Description

`dsnew = transform(ds,@fcn)` transforms an input datastore `ds` using the transformation function `fcn` and returns the transformed datastore `dsnew`.

`dsnew = transform(ds,@fcn,'IncludeInfo',IncludeInfo)` uses an alternative definition of the transform function `fcn`. The alternative definition enables you to use the additional information returned by the `read` function of the datastore.

Examples

Apply Transformation to Collection of Images

Create a datastore for a collection of images and apply the same transformation to all the images in the datastore. For instance, resize all the images in a collection to a specified target size.

Create an `ImageDatastore` with two images.

```
imds = imageDatastore({'street1.jpg','peppers.png'})
```

```
imds 
  ImageDatastore with properties:
```

```
                       Files: {
                              'B:\matlab\toolbox\matlab\demos\street1.jpg';
                              'B:\matlab\toolbox\matlab\imagesci\peppers.png'
                              }
    AlternateFileSystemRoots: {}
                    ReadSize: 1
                      Labels: {}
                     ReadFcn: @readDatastoreImage
```

Read all the images. Notice that the datastore contains images of different sizes.

```
img1 = read(imds); % reads the first image
img2 = read(imds); % reads the next image
whos img1 img2
```

```
  Name        Size                Bytes  Class    Attributes

  img1      480x640x3            921600  uint8
  img2      384x512x3            589824  uint8
```

Transform all the images in the datastore to a specified target size.

```
targetSize = [224,224];
imdsReSz = transform(imds,@(x) imresize(x,targetSize));
```

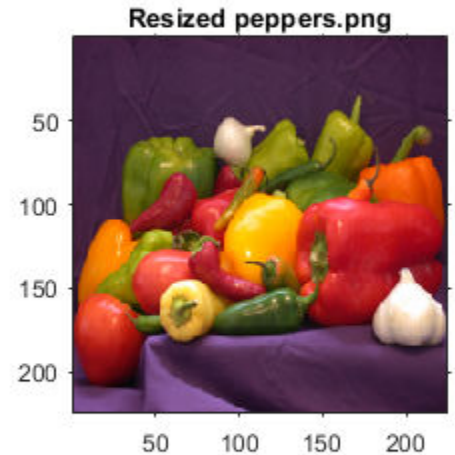
Read the images and display their sizes.

```
imgReSz1 = read(imdsReSz);
imgReSz2 = read(imdsReSz);
whos imgReSz1 imgReSz2
```

```
  Name            Size                Bytes  Class    Attributes

  imgReSz1      224x224x3            150528  uint8
  imgReSz2      224x224x3            150528  uint8
```

Display the resized images.

```
subplot(121); imshow(imgReSz1); axis on; title('Resized Street1.jpg');
subplot(122); imshow(imgReSz2); axis on; title('Resized peppers.png');
```



Input Arguments

ds — Input datastore
datastore

Input datastore. For a complete list of built-in datastores, see “Select Datastore for File Format or Application”. You can also specify a custom datastore.

`@fcn` — Function that transforms data
function handle

Function that transforms the data, specified as a function handle. The function takes data as an input and returns the transformed data, based on the transformations defined in `fcn`.

The transform function must have this signature and `dataIn` must be of the same form as the data that is returned by using the `read` function.

```
function dataOut = transformFcn(dataIn)
..
end
```

Alternatively, you can define your transform function `fcn` to use additional information about the data returned by the `read` function. To use this alternative definition, you must specify the value of `IncludeInfo` to be `true`. In this case, the transformation function must have this signature.

```
function [dataOut,infoOut] = transformFcn(dataIn,infoIn)
..
end
```

Example: `@transformFcn`

Data Types: `function_handle`

`IncludeInfo` — Include information from `read` function
`true` | `false`

Include information from `read` function, specified as either `true` or `false`. The `read` function returns information about the extracted data in an `info` struct. For more information, see the `read` function page.

When you set the value of `IncludeInfo` to `true`, you must use the alternative signature for your transform function `fcn`.

Output Arguments

`dsnew` — New datastore with transformed data
`TransformedDatastore` object

New datastore with transformed data, returned as a `TransformedDatastore` object.

See Also

`TransformedDatastore` | `combine` | `datastore` | `hasdata` | `preview` | `read` | `readall` | `reset`

Topics

"Select Datastore for File Format or Application"

Introduced in R2019a

TransformedDatastore

Datastore to transform underlying datastore

Description

Use a `TransformedDatastore` object to transform, or process, data read from an underlying datastore.

Creation

You can create a `TransformedDatastore` object using the `transform` function. For example, `dsnew = transform(ds,@fcn)` creates a datastore that transforms datastore `ds` using the transformation function `fcn`.

Properties

`UnderlyingDatastore` — Underlying datastore
datastore object

Underlying datastore, specified as a datastore object.

`TransformSet` — Set of transform functions
cell array

Set of transformation functions, specified as a cell array of function handles.

Data Types: `cell`

`IncludeInfo` — Include information from `read` function
logical vector

Include information from `read` function, specified as a logical vector. For each value of `IncludeInfo` that is `true`, the transformed datastore uses the alternative signature of the corresponding transform function in `transformSet`.

The `read` function returns information about the extracted data in an `info` struct. For more information, see the `read` function page.

Data Types: `logical`

Object Functions

combine	Combine data from multiple datastores
hasdata	Determine if data is available to read
preview	Subset of data in datastore
read	Read data in datastore
readall	Read all data in datastore
reset	Reset datastore to initial state
transform	Transform datastore

Examples

Apply Transformation to Collection of Images

Create a datastore for a collection of images and apply the same transformation to all the images in the datastore. For instance, resize all the images in a collection to a specified target size.

Create an `ImageDatastore` with two images.

```
imds = imageDatastore({'street1.jpg','peppers.png'})
```

```
imds 
  ImageDatastore with properties:

                       Files: {
                              'B:\matlab\toolbox\matlab\demos\street1.jpg';
                              'B:\matlab\toolbox\matlab\imagesci\peppers.png'
                              }
    AlternateFileSystemRoots: {}
                    ReadSize: 1
                      Labels: {}
                     ReadFcn: @readDatastoreImage
```

Read all the images. Notice that the datastore contains images of different sizes.

```
img1 = read(imds); % reads the first image
img2 = read(imds); % reads the next image
whos img1 img2
```

```
  Name        Size                Bytes  Class    Attributes

  img1      480x640x3            921600  uint8
  img2      384x512x3            589824  uint8
```

Transform all the images in the datastore to a specified target size.

```
targetSize = [224,224];
imdsReSz = transform(imds,@(x) imresize(x,targetSize));
```

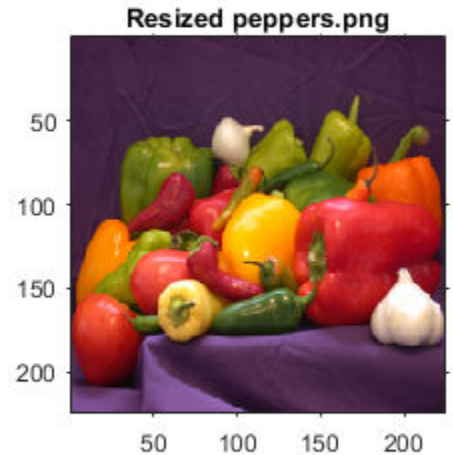
Read the images and display their sizes.

```
imgReSz1 = read(imdsReSz);
imgReSz2 = read(imdsReSz);
whos imgReSz1 imgReSz2
```

```
  Name            Size                Bytes  Class    Attributes

  imgReSz1      224x224x3            150528  uint8
  imgReSz2      224x224x3            150528  uint8
```

Display the resized images.

```
subplot(121); imshow(imgReSz1); axis on; title('Resized Street1.jpg');
subplot(122); imshow(imgReSz2); axis on; title('Resized peppers.png');
```



See Also

`CombinedDatastore` | `datastore`

Topics

"Getting Started with Datastore"

Introduced in R2019a

matlab.io.Datastore class

Package: `matlab.io`

Base datastore class

Description

`matlab.io.Datastore` is an abstract class for creating a custom datastore. A datastore helps access large collections of data iteratively, especially when data is too large to fit in memory. The `Datastore` abstract class declares and captures the interface expected for all custom datastores in MATLAB. Derive your class using this syntax:

```
classdef MyDatastore < matlab.io.Datastore
    ...
end
```

To implement your custom datastore:

- Inherit from the class `matlab.io.Datastore`
- Define the four required methods: `hasdata`, `read`, `reset`, and `progress`

For more details and steps to create your custom datastore, see “Develop Custom Datastore”.

Methods

`read`	Read data from the datastore.
	
	`[data,info] = read(ds)`
	
	The `data` output can be any data type and must be vertically concatenateable. Best practice is to return the `info` output as a structure.
	
	The data type of the output `data` dictates the data type of the output of the `tall` function.
	
	`Access: Public`, `Abstract: true`

`hasdata`	Determine if data is available to read. The output is of type logical.
	
	`tf = hasdata(ds)`
	
	`Access: Public`, `Abstract: true`

`reset`	Reset the datastore to an initial state before any data is read.
	
	`reset(ds)`
	
	`Access: Public`, `Abstract: true`

`progress`	Determine how much data is already read.
	
	The output is a scalar double between `0` and `1`. A return value of `0.55` means that you have read `55%` of the data.
	
	`p = progress(ds)`
	
	`Access: Public`, `Abstract: true`, `Hidden:true`

`preview`	Return a subset of the data.
	
	`data = preview(ds)`
	
	The default implementation returns the first eight rows of data. The output has the same data type as the output of `read`.
	
	The default implementation of the `preview` method is not optimized for tall array construction. For improved tall array performance, optimize your implementation based on your data.
	
	`Access: Public`

`readall`	Read all data in the datastore.
	
	`data = readall(ds)`
	
	The output has the same data type as the output of `read`. If the data does not fit in memory, `readall` returns an error.
	
	The default implementation of the `readall` method is not optimized for tall array construction. For improved tall array performance, optimize your implementation based on your data.
	
	`Access: Public`

Properties

To add handle properties to your custom datastore, you must implement the `copyElement` method. For example, if you use the `DsFileSet` object as a property in your custom datastore, then implement the `copyElement` method. Implementing the `copyElement` method enables you to create a deep copy of the datastore object. For more information, see “Customize Copy Operation”. For an example implementation of the `copyElement` method, see “Develop Custom Datastore”.

Attributes

Sealed	false

For information on class attributes, see “Class Attributes”.

Examples

Build Datastore to Read Binary Files

Build a datastore to bring your custom or proprietary data into MATLAB® for serial processing.

Create a `.m` class definition file that contains the code implementing your custom datastore. You must save this file in your working folder or in a folder that is on the MATLAB® path. The name of the `.m` file must be the same as the name of your object constructor function. For example, if you want your constructor function to have the name MyDatastore, then the name of the `.m` file must be `MyDatastore.m`. The `.m` class definition file must contain the following steps:

- Step 1: Inherit from the datastore classes.
- Step 2: Define the constructor and the required methods.
- Step 3: Define your custom file reading function.

In addition to these steps, define any other properties or methods that you need to process and analyze your data.

```
%% STEP 1: INHERIT FROM DATASTORE CLASSES
classdef MyDatastore < matlab.io.Datastore

    properties(Access = private)
        CurrentFileIndex double
        FileSet matlab.io.datastore.DsFileSet
    end

    % Property to support saving, loading, and processing of
    % datastore on different file system machines or clusters.
    % In addition, define the methods get.AlternateFileSystemRoots()
    % and set.AlternateFileSystemRoots() in the methods section.
```

```
    properties(Dependent)
        AlternateFileSystemRoots
    end


%% STEP 2: DEFINE THE CONSTRUCTOR AND THE REQUIRED METHODS
    methods
        % Define your datastore constructor
        function myds = MyDatastore(location,altRoots)
            myds.FileSet = matlab.io.datastore.DsFileSet(location,...
                'FileExtensions','.bin', ...
                'FileSplitSize',8*1024);
            myds.CurrentFileIndex = 1;

            if nargin == 2
                 myds.AlternateFileSystemRoots = altRoots;
            end

            reset(myds);
        end

        % Define the hasdata method
        function tf = hasdata(myds)
            % Return true if more data is available
            tf = hasfile(myds.FileSet);
        end

        % Define the read method
        function [data,info] = read(myds)
            % Read data and information about the extracted data
            % See also: MyFileReader()
            if ~hasdata(myds)
                error(sprintf(['No more data to read.\nUse the reset ',...
                      'method to reset the datastore to the start of ' ,...
                      'the data. \nBefore calling the read method, ',...
                      'check if data is available to read ',...
                      'by using the hasdata method.']))
            end

            fileInfoTbl = nextfile(myds.FileSet);
            data = MyFileReader(fileInfoTbl);
            info.Size = size(data);
            info.FileName = fileInfoTbl.FileName;
            info.Offset = fileInfoTbl.Offset;
```

```
            % Update CurrentFileIndex for tracking progress
            if fileInfoTbl.Offset + fileInfoTbl.SplitSize >= ...
                    fileInfoTbl.FileSize
                myds.CurrentFileIndex = myds.CurrentFileIndex + 1 ;
            end
        end

        % Define the reset method
        function reset(myds)
            % Reset to the start of the data
            reset(myds.FileSet);
            myds.CurrentFileIndex = 1;
        end

        % Getter for AlternateFileSystemRoots property
        function altRoots = get.AlternateFileSystemRoots(myds)
            altRoots = myds.FileSet.AlternateFileSystemRoots;
        end

        % Setter for AlternateFileSystemRoots property
        function set.AlternateFileSystemRoots(myds,altRoots)
            try
              % The DsFileSet object manages the AlternateFileSystemRoots
              % for your datastore
              myds.FileSet.AlternateFileSystemRoots = altRoots;

              % Reset the datastore
              reset(myds);
            catch ME
              throw(ME);
            end
        end
    end

    methods (Hidden = true)
        % Define the progress method
        function frac = progress(myds)
            % Determine percentage of data read from datastore
            if hasdata(myds)
               frac = (myds.CurrentFileIndex-1)/...
                              myds.FileSet.NumFiles;
            else
               frac = 1;
```

```
            end
        end
    end

    methods(Access = protected)
        % If you use the  FileSet property in the datastore,
        % then you must define the copyElement method. The
        % copyElement method allows methods such as readall
        % and preview to remain stateless
        function dscopy = copyElement(ds)
            dscopy = copyElement@matlab.mixin.Copyable(ds);
            dscopy.FileSet = copy(ds.FileSet);
        end

    end
end

%% STEP 3: IMPLEMENT YOUR CUSTOM FILE READING FUNCTION
function data = MyFileReader(fileInfoTbl)
% create a reader object using FileName
reader = matlab.io.datastore.DsFileReader(fileInfoTbl.FileName);

% seek to the offset
seek(reader,fileInfoTbl.Offset,'Origin','start-of-file');

% read fileInfoTbl.SplitSize amount of data
data = read(reader,fileInfoTbl.SplitSize);

end
```

Your custom datastore is now ready. Use `MyDatastore` to create a datastore object for reading your binary data files.

Create Datastore Object Using Custom Datastore And Read Data

Use custom datastore to preview and read your proprietary data into MATLAB for serial processing.

This example uses a simple data set to illustrate a workflow using your custom datastore. The data set is a collection of 15 binary (`.bin`) files where each file contains a column (`1` variable) and `10000` rows (records) of unsigned integers.

```
dir('*.bin')
```

```
binary_data01.bin  binary_data05.bin  binary_data09.bin  binary_data13.bin
binary_data02.bin  binary_data06.bin  binary_data10.bin  binary_data14.bin
binary_data03.bin  binary_data07.bin  binary_data11.bin  binary_data15.bin
binary_data04.bin  binary_data08.bin  binary_data12.bin
```

Create a datastore object using the `MyDatastore` function. For implementation details of `MyDatastore`, see the example **Build Datastore to Read Binary Files**.

```
folder = fullfile('*.bin');
ds = MyDatastore(folder);
```

Preview the data from the datastore.

```
preview(ds)
```

```
ans = 8x1 uint8 column vector

   113
   180
   251
    91
    29
    66
   254
   214
```

Read the data in a `while` loop and use the `hasdata` method to check if more data is available to read.

```
while hasdata(ds)
    data = read(ds);
    % do something
end
```

Reset the datastore to its initial state and read the data from the start of the datastore.

```
reset(ds);
data = read(ds);
```

Alternatively, if your data collection fits in memory, then read all the data in the datastore. Since the folder contains `15` files with `10000` records in each file, the size of the output should be `150000` records.

```
dataAll = readall(ds);
whos dataAll
```

```
  Name              Size             Bytes  Class    Attributes

  dataAll      150000x1             150000  uint8
```

Save and Load Datastore on Different Platforms

Create custom datastore object, save it on a Windows machine, and then load and process it on a Linux machine.

Before creating and saving your custom datastore, identify the root path of your data on the different platforms. The root paths differ based on the machine or file system. For example, if you access the data using these root paths:

- `"Z:\DataSet"` on your local Windows machine
- `"/nfs-bldg001/DataSet"` on your Linux cluster

Then, associate these root paths using the `AlternateFileSystemRoots` property. For implementation details of `MyDatastore`, see the example `Build Datastore to Read Binary Files`.

```
altRoots = ["Z:\DataSet","/nfs-bldg001/DataSet"];
ds = MyDatastore('Z:\DataSet\*.bin',altRoots);
```

Examine the files in the datastore.

```
fileTbl = resolve(ds.Fileset);
fileTbl.FileName
```

```
ans =

  12×1 cell array

    {'Z:\DataSet\binary_data01.bin'}
    {'Z:\DataSet\binary_data02.bin'}
    {'Z:\DataSet\binary_data03.bin'}
      .
      .
      .
```

Save the datastore.

```
save ds_saved_on_Windows.mat ds
```

Load the datastore on a Linux platform and examine the files in the datastore. Since the root path `'Z:\DataSet'` is not accessible on the Linux cluster at load time, the datastore function automatically updates the root paths based on the values specified in the `AlternateFileSystemRoots` property.

```
load ds_saved_on_Windows.mat
fileTbl = resolve(ds.Fileset);
fileTbl.FileName
```

```
ans =

  12×1 cell array

    {'/nfs-bldg001/DataSet/binary_data01.bin'}
    {'/nfs-bldg001/DataSet/binary_data02.bin'}
    {'/nfs-bldg001/DataSet/binary_data03.bin'}
      .
      .
      .
```

You can now process and analyze this datastore on your Linux machine.

See Also

`datastore` | `mapreduce` | `matlab.io.datastore.DsFileReader` | `matlab.io.datastore.DsFileSet` | `matlab.io.datastore.HadoopLocationBased` | `matlab.io.datastore.Partitionable`

Topics

"Develop Custom Datastore"
"Customize Copy Operation"

Introduced in R2017b

progress

Class: `matlab.io.Datastore`
Package: `matlab.io`

Determine how much data has been read

Syntax

```
p = progress(ds)
```

Description

`p = progress(ds)` returns the percentage of the data that you have read from the `matlab.io.Datastore` object specified by `ds`. The `p` output is a number between `0` and `1`. A return value of `0.55` means you have read 55% of the data.

Input Arguments

ds — Input datastore
`matlab.io.Datastore` object

Input datastore, specified as a `matlab.io.Datastore` object. To create a `Datastore` object, see `matlab.io.Datastore`.

Output Arguments

p — Percentage of data
scalar double

Percentage of data that you have read from the datastore, returned as a scalar double.

Data Types: `double`

Attributes

`Abstract`	`true`
`Access`	`public`
`Hidden`	`false`
`Sealed`	`false`
`Static`	`false`

To learn about attributes of methods, see Method Attributes.

Tips

- You must implement the `progress` method by deriving a subclass from the `matlab.io.Datastore` class. For more information, see “Develop Custom Datastore”.

See Also

`hasdata` | `matlab.io.Datastore` | `preview` | `progress` | `read` | `readall` | `reset`

Topics

“Develop Custom Datastore”

Introduced in R2017b

datatip

Create data tip

Description

Data tips are small text boxes that display information about individual data points. By default, data tips show the coordinates of a data point.

Creation

Interactively create data tips by clicking on data points in a chart, for example line and scatter plots. You can also create data tips using the `datatip` function.

Syntax

```
datatip(target)
datatip(target,x,y)
datatip(target,x,y,z)
datatip(target,'DataIndex',n)
datatip( ___ ,Name,Value)
dt = datatip( ___ )
```

Description

`datatip(target)` creates a data tip on the first plotted data point of the specified chart, such as a line or scatter plot.

`datatip(target,x,y)` creates a data tip on the 2-D plotted data point specified by `x` and `y`. If you specify approximate coordinates, then `datatip` creates a data tip at the nearest data point.

`datatip(target,x,y,z)` creates a data tip on the 3-D plotted data point specified by `x`, `y`, and `z`. If you specify approximate coordinates, then `datatip` creates a data tip at the nearest data point.

`datatip(target,'DataIndex',n)` creates a data tip on the `n`th plotted data point by setting the `DataIndex` property.

`datatip( ___ ,Name,Value)` specifies properties on page 1-2905 using one or more name-value pair arguments. Specify name-value pairs after all other arguments.

`dt = datatip( ___ )` returns a `DataTip` object. You can specify any of the input argument combinations in the previous syntaxes. This syntax is useful for controlling the properties of the data tip.

Input Arguments

`target` — Target
object with a `DataTipTemplate` property

Target, specified as an object with a `DataTipTemplate` property. Examples of objects with a `DataTipTemplate` property include `Line`, `Histogram`, `Surface`, and `FunctionLine` objects.

`x` — First coordinate
scalar

First coordinate of the data tip, specified as a scalar. The interpretation of the first coordinate depends on the type of axes:

- For Cartesian axes, the first coordinate is the *x*-axis position using the same units as your data.
- For polar axes, the first coordinate is the polar angle θ in radians.
- For geographic axes, the first coordinate is the latitude in degrees.

By default, if you specify the approximate coordinates of a data point, then the `datatip` function creates a data tip at the nearest data point. The nearest data point depends on the type of chart. For example, on a line chart the nearest point has the smallest Euclidean distance from the approximate coordinates.

To create data tips between data points, set the `SnapToDataVertex` property to `'off'`.

`y` — Second coordinate
scalar

Second coordinate of the data tip, specified as a scalar. The interpretation of the second coordinate depends on the type of axes:

- For Cartesian axes, the second coordinate is the *y*-axis position using the same units as your data.
- For polar axes, the second coordinate is the radius *r* in the same units as your data.
- For geographic axes, the second coordinate is the longitude in degrees.

By default, if you specify the approximate coordinates of a data point, then the `datatip` function creates a data tip at the nearest data point. The nearest data point depends on the type of chart. For example, on a line chart the nearest point has the smallest Euclidean distance from the approximate coordinates.

To create data tips between data points, set the `SnapToDataVertex` property to `'off'`.

z — Third coordinate
scalar

Third coordinate of the data tip, specified as a scalar. For Cartesian axes, the third coordinate is the *z*-axis position using the same units as your data.

By default, if you specify the approximate coordinates of a data point, then the `datatip` function creates a data tip at the nearest data point. The nearest data point depends on the type of chart. For example, on a line chart the nearest point has the smallest Euclidean distance from the approximate coordinates.

To create data tips between data points, set the `SnapToDataVertex` property to `'off'`.

Properties

Data Tip

`DataIndex` — Index of plotted data
positive integer

Index of the plotted data, specified as a positive integer.

For example, the plotted data of a 2-D object on a set of Cartesian axes is contained in its `XData` and `YData` properties. When you specify the `DataIndex` of a data tip, MATLAB positions the data tip on the chart by indexing into `XData` and `YData`.

When you create a data tip using this property, do not specify the `x`, `y`, or `z` arguments.

Example: `datatip(chart,'DataIndex',3)` creates a data tip at the third plotted data point.

`SnapToDataVertex` — Display at closest data point
`'on'` (default) | `'off'`

Display at closest data point, specified as one of these values:

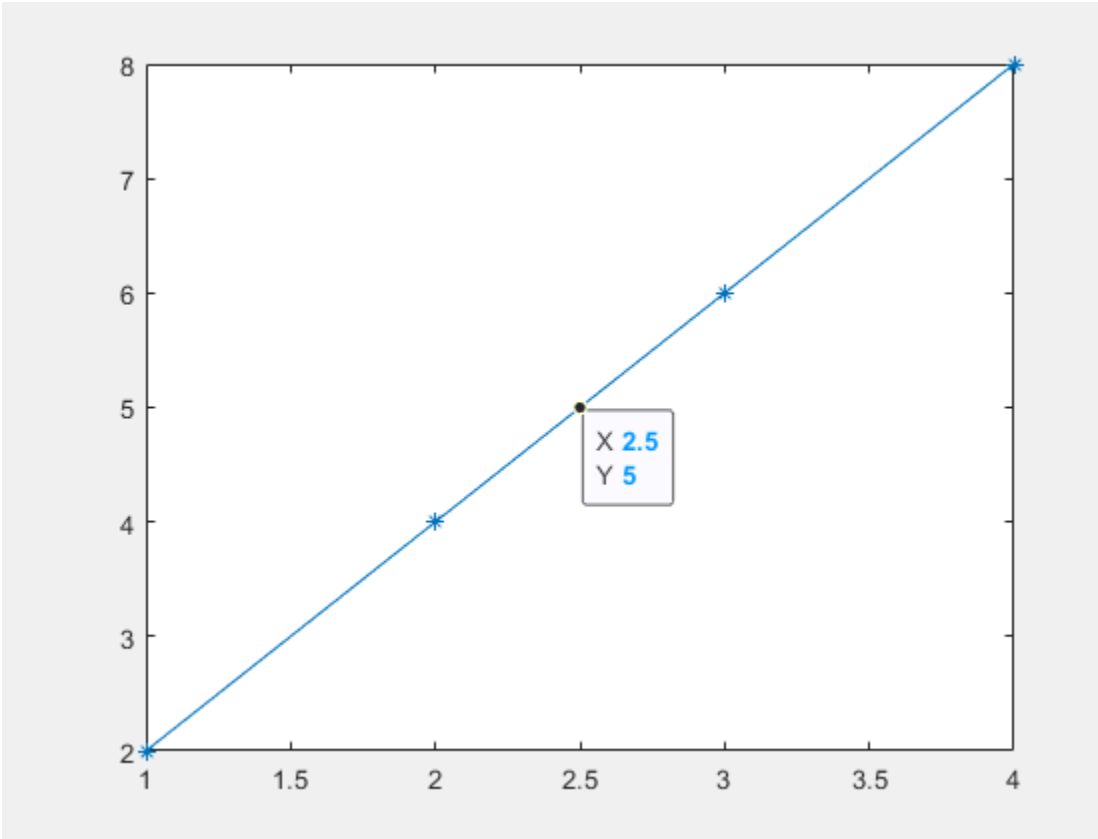
- `'on'` – Display the data tip at the closest data point.
- `'off'` – Display the data tip at the nearest specified location on the chart, even if it is between data points.

`InterpolationFactor` — Offset from `DataIndex`
0 (default) | scalar

Offset from `DataIndex`, specified as a scalar. Use this property to create data tips between adjacent data points on a `Line` object by setting `SnapToDataVertex` to `'off'` and `InterpolationFactor` to a scalar between 0 and 1.

For example, create a data tip halfway between adjacent data points on a line by specifying the `InterpolationFactor` as 0.5.

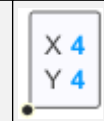
```
x = 1:4;
y = 2.*x;
p = plot(x,y,'-*');
dt = datatip(p,'DataIndex',2, ...
               'Location','southeast', ...
               'SnapToDataVertex','off', ...
               'InterpolationFactor',0.5);
```

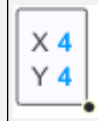
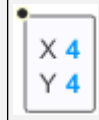
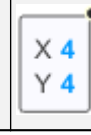


`InterpolationFactor` is not supported for data tips on all types of objects.

`Location` — Location
`'northeast'` | `'northwest'` | `'southeast'` | `'southwest'`

Location with respect to the data point, specified as one of these values:

Value	Description	Appearance
`'northeast'`	Top-right corner of data point	

Value	Description	Appearance
`'northwest'`	Top-left corner of data point	
`'southeast'`	Bottom-right corner of data point	
`'southwest'`	Bottom-left corner of data point	

Example: `dt.Location = 'southwest';`

`LocationMode` — Selection mode for `Location`
`'auto'` (default) | `'manual'`

Selection mode for the `Location` property value, specified as one of these values:

- `'auto'` – The `datatip` function automatically selects the location.
- `'manual'` – If you set the `Location` property, the `LocationMode` property automatically changes to `'manual'`.

`Content` — Data tip content
cell array

This property is read-only.

Data tip content, specified as a cell array. By default, data tip content is a 2-by-1 cell array for 2-D data tips or a 3-by-1 cell array for 3-D data tips.

To add additional rows to the data tip content, use `dataTipTextRow`.

Font

`FontName` — Font name
supported font name | `'FixedWidth'`

Font name, specified as a supported font name or `'FixedWidth'`. To display and print text properly, you must choose a font that your system supports. The default font depends on your operating system and locale.

To use a fixed-width font that looks good in any locale, use `'FixedWidth'`. The fixed-width font relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`FontNameMode` — Selection mode for `FontName`
`'auto'` (default) | `'manual'`

Selection mode for the `FontName` property, specified as one of these values:

- `'auto'` – Use the same value as the `FontName` property for the `DataTipTemplate` object associated with the chart that contains the data tip.
- `'manual'` – If you set the `FontName` property, the `FontNameMode` property automatically changes to `'manual'`.

`FontSize` — Font size
10 (default) | scalar value greater than zero

Font size, specified as a scalar value greater than zero in point units. One point equals `1/72` inch.

`FontSizeMode` — Selection mode for font size
`'auto'` (default) | `'manual'`

Selection mode for the font size, specified as one of these values:

- `'auto'` – Use the same value as the `FontSize` property for the `DataTipTemplate` object associated with the chart that contains the data tip.
- `'manual'` – If you set the `FontSize` property, the `FontSizeMode` property automatically changes to `'manual'`.

`FontAngle` — Character slant
`'normal'` (default) | `'italic'`

Character slant, specified as `'normal'` or `'italic'`.

Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.

FontAngleMode — Selection mode for FontAngle

'auto' (default) | 'manual'

Selection mode for the FontAngle property, specified as one of these values:

- 'auto' – Use the same value as the FontAngle property for the DataTipTemplate object associated with the chart that contains the data tip.
- 'manual' – If you set the FontAngle property, the FontAngleMode property automatically changes to 'manual'.

Interpreter — Interpretation of text characters

'tex' (default) | 'latex' | 'none'

Interpretation of text characters, specified as one of these values:

- 'tex' — Interpret characters using a subset of TeX markup.
- 'latex' — Interpret characters using LaTeX markup.
- 'none' — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text.

Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they modify only the next character or the characters within the curly braces. When you set the Interpreter property to 'tex', the supported modifiers are as follows.

Modifier	Description	Example
^{ }	Superscript	'text^{superscript}'
_{ }	Subscript	'text_{subscript}'
\bf	Bold font	'\bf text'
\it	Italic font	'\it text'
\sl	Oblique font (usually the same as italic font)	'\sl text'
\rm	Normal font	'\rm text'

Modifier	Description	Example
`\fontname{`*`specifier`*`}`	Font name — Replace *`specifier`* with the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{`*`specifier`*`}`	Font size —Replace *`specifier`* with a numeric scalar value in point units.	`'\fontsize{15} text'`
`\color{`*`specifier`*`}`	Font color — Replace *`specifier`* with one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Replace *`specifier`* with a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	β	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ϵ	`\Delta`	Δ	`\spadesuit`	♠
`\zeta`	ζ	`\Theta`	Θ	`\leftrightarrow`	↔

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	µ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. Use dollar symbols around the text, for example, use `'$\int_1^{20} x^2 dx$'` for inline mode or `'$$ \int_1^{20} x^2 dx$$'` for display mode.

The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup.

The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters. For multiline text, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project website at https://www.latex-project.org/.

`InterpreterMode` — Selection mode for `Interpreter`

`'auto'` (default) | `'manual'`

Selection mode for the `Interpreter` property, specified as one of these values:

- `'auto'` — Use the same value as the `Interpreter` property for the `DataTipTemplate` object associated with the chart that contains the data tip.
- `'manual'` — If you set the `Interpreter` property, the `InterpreterMode` property automatically changes to `'manual'`.

Cartesian Coordinate Data

X — Position along x-axis

scalar

Position along the *x*-axis, specified as a scalar using the same units as your data.

Y — Position along y-axis

scalar

Position along the *y*-axis, specified as a scalar using the same units as your data.

Z — Position along z-axis

scalar

Position along the *z*-axis, specified as a scalar using the same units as your data.

Polar Coordinate Data

R — Radius
scalar

Radius, specified as a scalar using the same units as your data.

This property applies only to data tips on polar plots.

`Theta` — Angle
scalar

Angle, specified as a scalar in degrees.

This property applies only to data tips on polar plots.

Geographic Coordinate Data

`Latitude` — Latitude
scalar

Latitude, specified as a scalar in degrees.

This property applies only to data tips on geographic plots.

`Longitude` — Longitude
scalar

Longitude, specified as a scalar in degrees.

This property applies only to data tips on geographic plots.

Interactivity

`Visible` — State of visibility
`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.

- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`UIContextMenu` — Context menu
`ContextMenu` object (default)

Context menu, specified as a `ContextMenu` object that provides data tip options. You cannot set this property.

`Selected` — Selection state
`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles
`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

Callbacks

`ButtonDownFcn` — Mouse-click callback
`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see "Callback Definition".

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks when visible. The `Visible` property must be set to `'on'` and you must click a part of the `DataTip` object that has a defined color. You cannot click a part that has an associated color property set to `'none'`. The `HitTest` property determines if the `DataTip` object responds to the click or if an ancestor does.
- `'all'` — Capture mouse clicks regardless of visibility. The `Visible` property can be set to `'on'` or `'off'` and you can click a part of the `DataTip` object that has no color. The `HitTest` property determines if the `DataTip` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `DataTip` object passes the click through it to the object below it in the current view of the figure window. The `HitTest` property has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `DataTip` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `DataTip` object that has one of these:
  - `HitTest` property set to `'on'`
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

Note The `PickableParts` property determines if the `DataTip` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent

object with a `DataTipTemplate` property

Parent, specified as a object with a `DataTipTemplate` property. Examples of objects with a data tip template include `Line`, `Histogram`, `Surface`, and `FunctionLine` objects. Move a data tip to another object by setting this property.

`Children` — Children

empty `GraphicsPlaceholder` array

The object has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.

- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'datatip'`

This property is read-only.

Type of graphics object, returned as `'datatip'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, when searching for the type using `findobj`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[ ]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

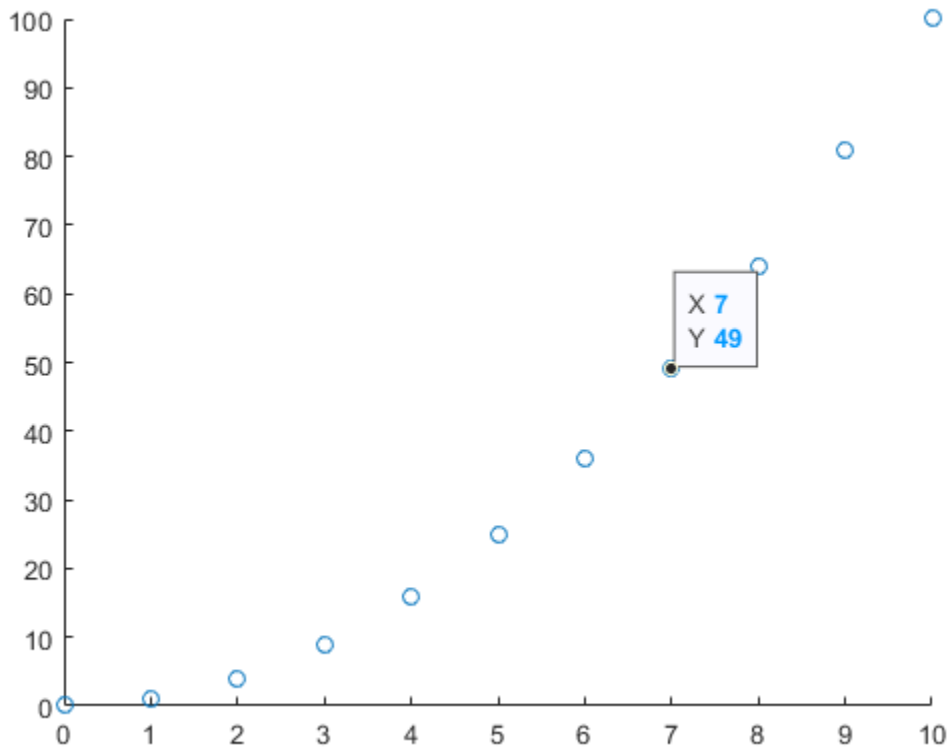
If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

Examples

Create Data Tip Using Coordinates

Plot data on a scatter chart and return the `Scatter` object. Then, create a data tip at a particular data point by specifying the *x*- and *y*-coordinates.

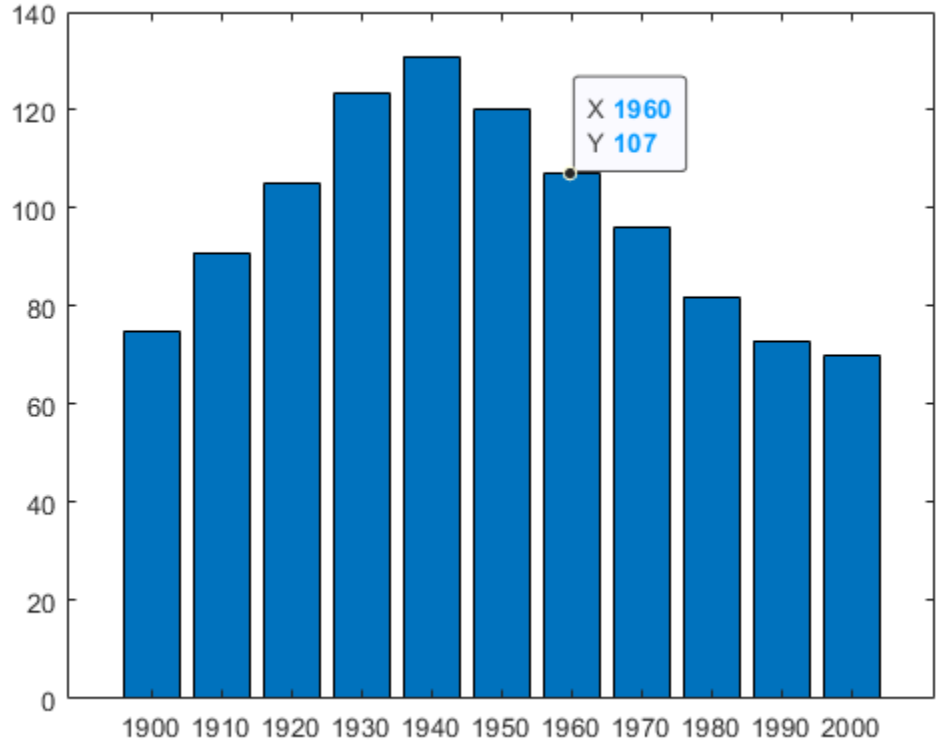
```
x = linspace(0,10,11);
y = x.^2;
sc = scatter(x,y);

dt = datatip(sc,7,49);
```



Create Data Tip Using Index

Plot data on a bar chart and return the `Bar` object. Then, create a data tip on the seventh bar by specifying the `DataIndex` name-value pair argument.

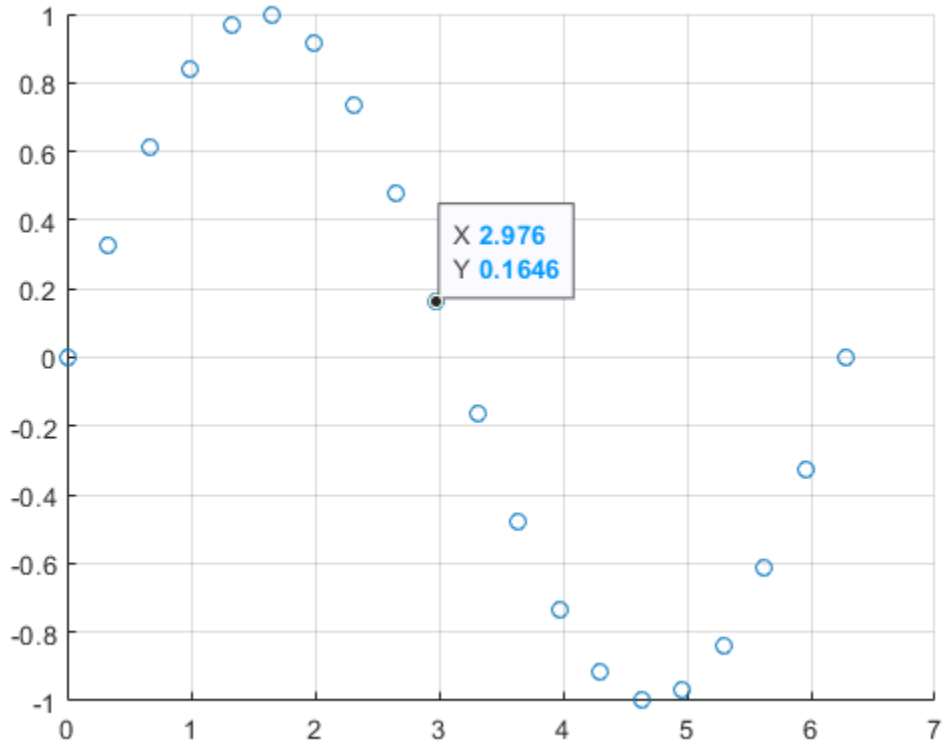
```
x = 1900:10:2000;
y = [75 91 105 123.5 131 120 107 96 82 73 70];
b = bar(x,y);

dt = datatip(b,'DataIndex',7);
```



Create Data Tip Using Approximate Coordinates

Plot data on a scatter chart and return the `Scatter` object. Create a data tip at the data point near (3,0) by specifying $x = 3$ and $y = 0$ as approximate coordinates. When you create a data tip using approximate coordinates, the data tip appears at the closest data point.

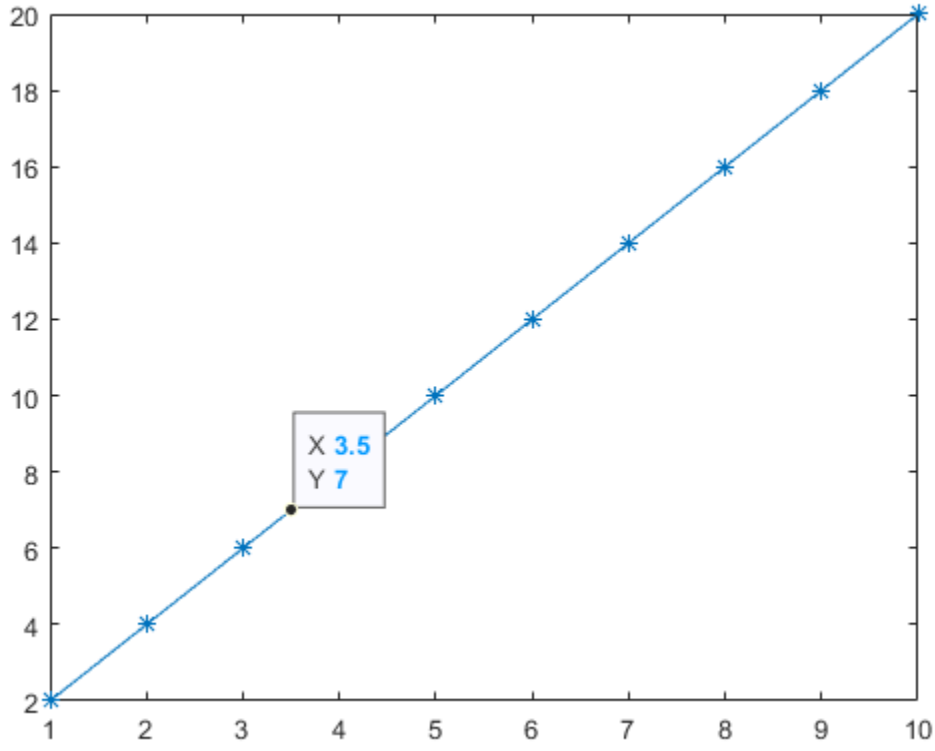
```
x = linspace(0,2*pi,20);
y = sin(x);
sc = scatter(x,y);
grid on;

dt = datatip(sc,3,0);
```



Create Data Tip Between Data Points

Plot data and return the `Line` object. Show a marker at each plotted data point. Then, create a data tip between two data points by specifying the *x*- and *y*-coordinates for the data tip and specifying the `SnapToDataVertex` name-value pair argument as `'off'`.

```
x = linspace(1,10,10);
y = 2.*x;
p = plot(x,y,'-*');

dt = datatip(p,3.5,7,'SnapToDataVertex','off');
```



Tips

- Interactively create data tips by clicking on data points in a chart. To assign interactively created data tips to a variable, use the `findobj` function.

  ```
  dt = findobj(chart,'Type','datatip');
  ```

- To delete data tips, use the `delete` function.

  ```
  delete(dt);
  ```

See Also

DataTipTemplate Properties | `dataTipTextRow`

Topics

"Create Custom Data Tips"
"Interactively Explore Plotted Data"

Introduced in R2019b

dataTipInteraction

Data tip interaction

Description

A data tip interaction allows you to display data tips within a chart without having to select any buttons in the axes toolbar. To enable data tips, set the `Interactions` property of the axes to a `dataTipInteraction` object. When this interaction is enabled, the data tips appear when you hover, click, or tap within the chart.

To enable multiple interactions, set the `Interactions` property to an array of objects.

Creation

Syntax

```
d = dataTipInteraction
d = dataTipInteraction('SnapToDataVertex',snap)
```

Description

`d = dataTipInteraction` creates a data tip interaction object. With this interaction enabled, clicking or tapping displays a data tip at the nearest data point.

`d = dataTipInteraction('SnapToDataVertex',snap)` specifies whether data tips snap to the nearest data point. When `snap` is `'off'`, clicking or tapping displays a data tip at the exact cursor location instead.

Properties

`SnapToDataVertex` — Display at closest data point
`'on'` (default) | `'off'`

Display at closest data point, specified as one of these values:

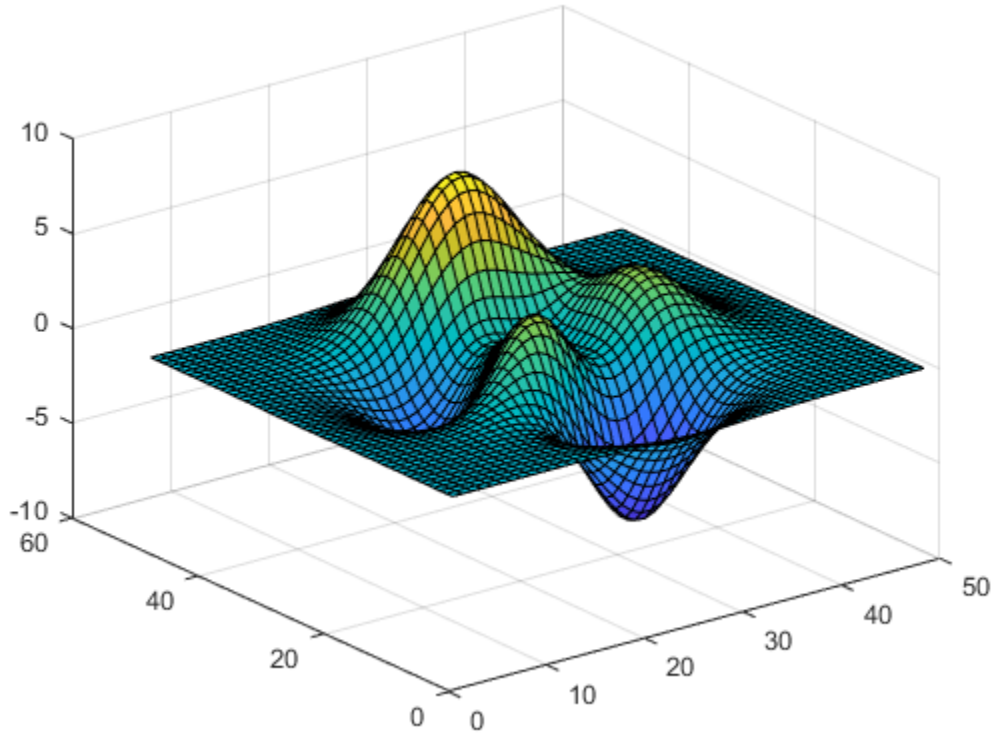
- `'on'` — Display data tip at closest data point.
- `'off'` — Display data tip at cursor location.

Examples

Axes with Data Tip and Pan Interactions

Create a surface plot. Get the current axes and replace the default interactions with the data tip and pan interactions. Then hover over the surface to display data tips. Click and drag or tap and drag to pan.

```
surf(peaks)
ax = gca;
ax.Interactions = [dataTipInteraction panInteraction];
```



Limitations

The `dataTipInteraction` function is not supported in the Live Editor.

Tips

- To customize the content of the data tips, set the `DataTipTemplate` property of the chart object. For more information, see “Create Custom Data Tips”.

- In most cases, the axes have a default set of interactions which depend on the type of chart you are displaying. You can replace the default set with a new set of interactions, but you cannot access or modify any of the interactions in the default set.

See Also

`disableDefaultInteractivity` | `enableDefaultInteractivity` | `panInteraction` | `regionZoomInteraction` | `rotateInteraction` | `rulerPanInteraction` | `zoomInteraction`

Topics

"Control Chart Interactivity"
"Create Custom Data Tips"

Introduced in R2019a

date

Current date as character vector

Syntax

```
c = date
```

Description

`c = date` returns the current date as a character vector in the format `dd-MMM-yyyy`.

This format represents the day of the month (`dd`) as a number, the month name (`MMM`) as its three-letter abbreviation, and the year (`yyyy`) as a number.

Examples

Return Current Date

To return the current date as a character vector, use the `date` function.

```
c = date
```

```
c =
'27-Aug-2019'
```

Limitations

- `date` always returns the English abbreviation for the month name. The function does not take your system locale into account when determining the month name.
- MATLAB Online determines the current date from Coordinated Universal Time (UTC) rather than from local time.

Tips

- As an alternative, use the `datetime` function to represent the current date.

  ```
  d = datetime('today')
  ```

  The `datetime` function takes your locale into account. Also, you can specify the format that a `datetime` array uses to represent its values.

See Also

`datestr` | `datetime` | `now`

Topics

"Represent Dates and Times in MATLAB"
"Convert Between Datetime Arrays, Numbers, and Text"

Introduced before R2006a

datenum

Convert date and time to serial date number

Syntax

```
DateNumber = datenum(t)
```

```
DateNumber = datenum(DateString)
DateNumber = datenum(DateString,formatIn)
DateNumber = datenum(DateString,PivotYear)
DateNumber = datenum(DateString,formatIn,PivotYear)
```

```
DateNumber = datenum(DateVector)
DateNumber = datenum(Y,M,D)
DateNumber = datenum(Y,M,D,H,MN,S)
```

Description

`DateNumber = datenum(t)` converts the datetime or duration values in the input array `t` to serial date numbers.

A serial date number represents the whole and fractional number of days from a fixed, preset date (January 0, 0000) in the proleptic ISO calendar.

Note While serial date numbers can represent dates and times, it is recommended that you use `datetime` values to represent points in time, and `duration` or `calendarDuration` values to represent elapsed times.

`DateNumber = datenum(DateString)` converts text representing dates and times to serial date numbers. If the format used in the text is known, specify the format as `formatIn`. Syntaxes without `formatIn` are significantly slower than syntaxes that include it.

`DateNumber = datenum(DateString,formatIn)` uses `formatIn` to interpret the dates and times represented by `DateString`.

`DateNumber = datenum(DateString,PivotYear)` uses `PivotYear` to interpret text that specifies the year as two characters. If the format used in the text is known, specify the format as `formatIn`. Syntaxes without `formatIn` are significantly slower than syntaxes that include it.

`DateNumber = datenum(DateString,formatIn,PivotYear)` uses `formatIn` to interpret the dates and times represented by `DateString`, and `PivotYear` to interpret text that specifies the year as two characters. You can specify `formatIn` and `PivotYear` in either order.

`DateNumber = datenum(DateVector)` converts date vectors to serial date numbers, and returns a column vector of `m` date numbers, where `m` is the total number of date vectors in `DateVector`.

`DateNumber = datenum(Y,M,D)` returns the serial date numbers for corresponding elements of the `Y`, `M`, and `D` (year, month, day) arrays. The arrays must be of the same size (or any can be a scalar). You also can specify the input arguments as a date vector, `[Y,M,D]`.

`DateNumber = datenum(Y,M,D,H,MN,S)` additionally returns the serial date numbers for corresponding elements of the `H`, `MN`, and `S` (hour, minute, and second) arrays. The arrays must be of the same size (or any can be a scalar). You also can specify the input arguments as a date vector, `[Y,M,D,H,MN,S]`.

Examples

Convert datetime Array to Date Numbers

```
format long
```

```
t = [datetime('now');datetime('tomorrow')]
```

```
t = 2x1 datetime array
   26-Aug-2019 17:37:05
   27-Aug-2019 00:00:00
```

```
DateNumber = datenum(t)
```

```
DateNumber = 2×1
10^5 ×
```

```
   7.376637340918519
   7.376640000000000
```

Convert Text Representing Date to Date Number

```
DateString = '19-May-2001';
formatIn = 'dd-mmm-yyyy';
datenum(DateString,formatIn)
```

```
ans = 730990
```

`datenum` returns a date number for text representing a date with the format `'dd-mmm-yyyy'`.

Convert Text with Multiple Dates to Date Numbers

Pass several dates as character vectors in a cell array. All input dates must use the same format.

```
DateString = {'09/16/2007';'05/14/1996';'11/29/2010'};
formatIn = 'mm/dd/yyyy';
datenum(DateString,formatIn)
```

```
ans = 3×1

      733301
      729159
      734471
```

Convert Text Representing Date to Date Number Using Pivot Year

Convert text representing a date to a serial date number using the default pivot year.

```
n = datenum('12-jun-17','dd-mmm-yy')
```

```
n = 736858
```

The text that represents this date number is `'12-Jun-2017'`.

Convert the same text to a serial date number using 1400 as the pivot year.

```
n = datenum('12-jun-17','dd-mmm-yy',1400)
```

```
n = 517712
```

The text that represents this date number is `'12-Jun-1417'`.

Convert Date Vector to Date Number

```
datenum([2009,4,2,11,7,18])
```

```
ans = 7.3387e+05
```

Convert Year, Month, and Day to Date Number

Convert a date specified by year, month and day values to a serial date number.

```
n = datenum(2001,12,19)
```

```
n = 731204
```

Input Arguments

`t` — Dates and times
array of datetime or duration values

Dates and times, specified as an array of datetime or duration values. The `datenum` function does not account for time zone information in `t` and does not adjust datetime values that occur during Daylight Saving Time. That is, `datenum` treats the `TimeZone` property of `datetime` arrays as empty and converts the remaining date and time information to a serial date number.

Data Types: `datetime` | `duration`

`DateVector` — Date vectors
matrix

Date vectors, specified as an `m`-by-6 or `m`-by-3 matrix containing `m` full or partial date vectors, respectively. A full date vector has six elements, specifying year, month, day, hour, minute, and second, in that order. A partial date vector has three elements, specifying year, month, and day, in that order. Each element of `DateVector` must be a positive or negative integer value except for the seconds element, which can be fractional. If an element falls outside the conventional range, `datenum` adjusts both that date vector element and the previous element. For example, if the minutes element is `70`, then `datenum` adjusts the hours element by `1` and sets the minutes element to `10`. If the minutes element is `-15`, then `datevec` decreases the hours element by `1` and sets the minutes element to `45`. Month values are an exception. The `datenum` function sets month values less than `1` to `1`.

Example: `[2003,10,24,12,45,07]`

Data Types: `double`

`DateString` — Text representing dates and times
character array | cell vector of character vectors | string vector

Text representing dates and times, specified as a character array where each row contains text representing one point in time, as a cell vector of character vectors, or as a string vector. All rows of a character array, or all elements of a cell vector or string vector, must have the same format.

Example: `'24-Oct-2003 12:45:07'`

Example: `['19-Sep-2013';'20-Sep-2013';'21-Sep-2013']`

Example: `{'15-Oct-2010' '20-Nov-2012'}`

If the format used in the text is known, you should also specify `formatIn`. If you do not specify `formatIn`, then `DateString` must be in one of the following formats.

Format of Text Representing Dates and Times	Example
`'dd-mmm-yyyy HH:MM:SS'`	`01-Mar-2000 15:45:17`
`'dd-mmm-yyyy'`	`01-Mar-2000`

Format of Text Representing Dates and Times	Example
'mm/dd/yyyy'	03/01/2000
'mm/dd/yy'	03/01/00
'mm/dd'	03/01
'mmm.dd,yyyy HH:MM:SS'	Mar.01,2000 15:45:17
'mmm.dd,yyyy'	Mar.01,2000
'yyyy-mm-dd HH:MM:SS'	2000-03-01 15:45:17
'yyyy-mm-dd'	2000-03-01
'yyyy/mm/dd'	2000/03/01
'HH:MM:SS'	15:45:17
'HH:MM:SS PM'	3:45:17 PM
'HH:MM'	15:45
'HH:MM PM'	3:45 PM

Note The symbolic identifiers describing date and time formats are different from those that describe the display formats of `datetime` arrays.

Certain formats might not contain enough information to convert text representations of dates and times. In those cases, hours, minutes, and seconds default to 0, days default to 1, months default to January, and years default to the current year. `datevec` and `datenum` consider two-character years (e.g., `'79'`) to fall within the 100-year range centered around the current year.

When you do not specify `formatIn`, note the following:

- For the formats that specify the month as two digits (`mm`), the month value must not be greater than 12.
- However, for the format `'mm/dd/yy'`, if the first entry in the text is greater than 12 and the second entry is less than or equal to 12, then `datenum` considers the text to be in `'yy/mm/dd'` format.

`formatIn` — Format of input text representing dates and times
character vector | string scalar

Format of the input text representing dates and times, specified as a character vector or string scalar of symbolic identifiers.

Example: `'dddd, mmm dd, yyyy'`

The following table shows symbolic identifiers that you can use to construct the `formatIn` character vector. You can include characters such as a hyphen, space, or colon to separate the fields.

Note The symbolic identifiers describing date and time formats are different from the identifiers that describe the display formats of `datetime` arrays.

Symbolic Identifier	Description	Example
`yyyy`	Year in full	`1990`, `2002`
`yy`	Year in two digits	`90`, `02`
`QQ`	Quarter year using letter `Q` and one digit	`Q1`
`mmmm`	Month using full name	`March`, `December`
`mmm`	Month using first three letters	`Mar`, `Dec`
`mm`	Month in two digits	`03`, `12`
`m`	Month using capitalized first letter	`M`, `D`
`dddd`	Day using full name	`Monday`, `Tuesday`
`ddd`	Day using first three letters	`Mon`, `Tue`
`dd`	Day in two digits	`05`, `20`
`d`	Day using capitalized first letter	`M`, `T`
`HH`	Hour in two digits (no leading zeros when symbolic identifier `AM` or `PM` is used)	`05`, `5 AM`
`MM`	Minute in two digits	`12`, `02`
`SS`	Second in two digits	`07`, `59`
`FFF`	Millisecond in three digits	`057`

Symbolic Identifier	Description	Example
`AM or PM`	`AM` or `PM` inserted in text representing time	`3:45:02 PM`

The `formatIn` value must follow these guidelines:

- You cannot specify any field more than once. For example, you cannot use `'yy-mmm-dd-m'` because it has two month identifiers. The one exception to this is that you can combine one instance of `dd` with one instance of any of the other day identifiers. For example, `'dddd mmm dd yyyy'` is a valid input.
- When you use `AM` or `PM`, the `HH` field is also required.
- You only can use `QQ` alone or with a year specifier.

`PivotYear` — Start year of 100-year date range
present minus 50 years (default) | integer

Start year of the 100-year date range in which a two-character year resides, specified as an integer. Use a pivot year to interpret dates that specify the year as two characters.

If `formatIn` contains the time of day, the pivot year is computed from the current time of the current day, month, and year. Otherwise it is computed from midnight of the current day, month, and year.

Example: `2000`

Note If the input date format specifies a four-character year, then the last two characters are truncated, and the *first* two characters specify the year. For example, if the date and pivot year are specified as `('25122015','ddmmyyyy',2000)`, then the resulting date is `25-12-2020`, not `25-12-2015`.

Data Types: `double`

`Y,M,D` — Year, month, and day arrays
numeric arrays

Year, month, and day arrays specified as numeric arrays. These arrays must be the same size, or any one can be a scalar. The values in `Y,M,D` must be integer values.

If `Y,M,D` are all scalars or all column vectors, you can specify the input arguments as a date vector, `[Y,M,D]`.

Example: `2003,10,24`

Data Types: `double`

`Y,M,D,H,MN,S` — Year, month, day, hour, minute, and second arrays
numeric arrays

Year, month, day, hour, minute, and second arrays specified as numeric arrays. These arrays must be the same size, or any one can be a scalar. `datenum` does not accept milliseconds as a separate input, but as a fractional part of the seconds input, `S`. The values in `Y,M,D,H,MN` must be integer values.

If `Y,M,D,H,MN,S` are all scalars or all column vectors, you can specify the input arguments as a date vector, `[Y,M,D,H,MN,S]`.

Example: `2003,10,24,12,45,07.451`

Data Types: `double`

Output Arguments

`DateNumber` — Serial date numbers
scalar | vector

Serial date numbers, returned as a column vector of length `m`, where `m` is the total number of input date vectors or character vectors representing dates and times.

Tips

- To create arbitrarily shaped output, use the `datenum(Y,M,D)` and `datenum(Y,M,D,H,MN,S)` syntaxes. The `datenum(DateVector)` syntax creates only a column vector of date numbers.

  ```
  datenum(2013,[1 3; 2 4],ones(2,2))
  ```

  ```
  ans =

        735235       735294
        735266       735325
  ```

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`datestr` | `datetick` | `datetime` | `datevec`

Topics

“Represent Dates and Times in MATLAB”
“Carryover in Date Vectors and Strings”

Introduced before R2006a

DatePicker Properties

Control appearance and behavior of date picker

Description

Date pickers allow users to select dates from an interactive calendar. The `uidatepicker` function creates a date picker and sets any required properties before displaying it. By changing property values of a date picker, you can modify certain aspects of its appearance and behavior. Use dot notation to refer to a specific object and property:

```
uf = uifigure;
d = uidatepicker(uf);
d.DisplayFormat = 'M/d/yyyy';
```

Properties

Date Picker

`Value` — Selected date
`NaT` (default) | `datetime` object

Selected date, specified as a `datetime` object within the range of the `Limits` property. To make the selected date unspecified, set this property to `NaT`.

If the specified `datetime` object contains time information, only the date information is preserved in the `Value` property.

Example: `d = uidatepicker('Value',datetime('today'))`

Data Types: `datetime`

`Limits` — Selection limits
1-by-2 `datetime` array

Selection limits, specified as a 1-by-2 `datetime` array. The second value in this array must be later than the first value. The default value is `[datetime(0000,1,1) datetime(9999,12,31)]`. This default value starts at the earliest possible date and ends at the latest possible date that `DatePicker` supports.

In the running app, the date picker allows the user to select dates on the closed interval defined by this property. If there are disabled dates or disabled days within the interval, then those dates and days are excluded.

Example: `d = uidatepicker('Limits',[datetime('today') datetime(2050,1,1)])`

Data Types: `datetime`

`DisplayFormat` — Display format

character vector | string scalar

Display format for the date picker text field, specified as a character vector or string scalar. The default format depends on the locale of the system running the app.

The format you specify must use valid letter identifiers that correspond to the Unicode Locale Data Markup Language (LDML) standard for dates and times. To separate fields, you can include nonletter characters such as a hyphen, space, colon, or any non-ASCII characters.

Example: `d = uidatepicker('DisplayFormat','dd/MM/yy')`

Examples of Common Formats

This table lists common display formats. The examples show formatted output for the date, Wednesday, April 9, 2014.

Value of Format	Example
'yyyy-MM-dd'	2014-04-09
'dd/MM/yyyy'	09/04/2014
'dd.MM.yyyy'	09.04.2014
'yyyy 年 MM 月 dd 日'	2014 年 04 月 09 日
'MMMM d, yyyy'	April 9, 2014

All Date and Time Formats

Use these letter identifiers to create a display format. The third column of this table shows output for the date, Wednesday, April 9, 2014.

Letter Identifier	Description	Display
G	Era	CE
y	Year, with no leading zeros.	2014
yy	Year, using last two digits.	14
yyy, yyyy ...	Year, using at least as many digits as there are instances of 'y'	For the year 2014, 'yyy' displays 2014, while 'yyyyy' displays 02014.
u, uu, ...	ISO year, a single number designating the year.	2014
Q	Quarter, using one digit	2
QQ	Quarter, using two digits	02
QQQ	Quarter, abbreviated	Q2
QQQQ	Quarter, full name	2nd quarter
M	Month, numerical, using one or two digits	4
MM	Month, numerical, using two digits	04
MMM	Month, abbreviated name	Apr
MMMM	Month, full name	April
MMMMM	Month, capitalized first letter	A
W	Week of the month, using one digit	2
d	Day of the month, using one or two digits	9
dd	Day of the month, using two digits	09
D	Day of the year, using one, two, or three digits	99
DD	Day of the year, using two digits	99
DDD	Day of the year using three digits	099
e	Day of the week, numerical, using one or two digits	4, where Sunday is the first day of the week

Letter Identifier	Description	Display
ee	Day of the week, numerical, using two digits	04
eee	Day, abbreviated name	Wed
eeee	Day, full name	Wednesday
eeeee	Day, capitalized first letter	W

Note

- The edit field in the running app accepts delimited numeric values, even when the `DisplayFormat` includes words. For instance, if the month format is specified as `'MMMM'`, the app accepts a numeric month such as `04`, but will display a month name such as `'April'`.
- If the user specifies a day-of-year number in the running app, and the format contains identifiers for both the day of year (`D`) and Gregorian year (`y`), then `datetime` might not read the day-of-year number correctly. Use ISO year (`u`) in place of `y`.
- Use one or more `u` characters instead of `y` characters to represent the year when working with year numbers near zero.

`DisabledDates` — Disabled dates

empty `datetime` array (default) | m-by-1 `datetime` array

Disabled dates, specified as an m-by-1 `datetime` array. This property specifies dates that are not available for selection in the running app.

Example: `d = uidatepicker('DisabledDates',datetime(2018,1,1))` disables January 1, 2018.

The `datetime` array cannot not contain any `NaT` values, and the dates must be sorted in ascending order.

To reenable all previously disabled dates, call `NaT(0)` to create an empty `datetime` array:

```
d.DisabledDates = NaT(0);
```

Data Types: `datetime`

DisabledDaysOfWeek — Disabled days of week

[] (default) | vector of integers in the range [1, 7] | cell array of character vectors | string vector

Disabled days of the week, specified as one of the following:

- Empty array `[]`, which enables all days of the week.
- Vector of whole numbers in the range [1, 7]. The numbers correspond to the days of the week. For example, `[1 3]` disables Sundays and Tuesdays.
- 1-D cell array of character vectors, where the array elements contain localized day names. Partial day names must be unambiguous. For example, `{'F','Sa'}` disables Fridays and Saturdays.
- String vector containing full or partial localized day names.

When you specify day names using a cell array or string vector, the code works only in the locale that you write the code. To make the code work in any locale, specify this property as a vector of numbers.

Data Types: `double` | `cell` | `string`

Font and Color

FontName — Font name

system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

FontSize — Font size

positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

FontWeight — Font weight

`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

`FontAngle` — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

`FontColor` — Font color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`BackgroundColor` — Background color

`[.96 .96 .96]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Interactivity

`Visible` — Component visibility

`'on'` (default) | `'off'`

Component visibility, specified as `'on'` or `'off'`. When the `Visible` property is set to `'off'`, the component is not visible in the UI, but you can query and set its properties.

`Editable` — Allow edit field changes

`'on'` (default) | `'off'`

Allow edit field changes, specified as `'on'` or `'off'`. Set this property to `'on'` to allow the user to change the date in the edit field at run time. The `Enable` property must also be set to `'on'` to allow changes in the edit field.

`Enable` — Enable interactions

`'on'` (default) | `'off'`

Enable interactions, specified as `'on'` or `'off'`. Set this property to `'off'` to make the component appear dim, indicating that the user cannot interact with it.

`Tooltip` — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size

`[100 100 150 22]` (default) | `[left bottom width height]`

Location and size of the collapsed date picker relative to the parent container, specified as a vector of the form `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the date picker
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the date picker
`width`	Distance between the right and left outer edges of the date picker

Element	Description
`height`	Distance between the top and bottom outer edges of the date picker

All measurements are in pixel units.

`InnerPosition` — Location and size
`[100 100 150 22]` (default) | `[left bottom width height]`

Location and size of the collapsed date picker relative to the parent container, specified as a vector of the form `[left bottom width height]`. This property value is identical to the `Position` property.

`OuterPosition` — Location and size
`[100 100 150 22]` (default) | `[left bottom width height]`

Location and size of the collapsed date picker relative to the parent container, specified as a vector of the form `[left bottom width height]`. This property value is identical to the `Position` property.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a date picker in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
d = uidatepicker(g);
d.Layout.Row = 3;
d.Layout.Column = 2;
```

To make the date picker span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this date picker spans columns `2` through `3`:

```
d.Layout.Column = [2 3];
```

Callbacks

`ValueChangedFcn` — Value changed function

`''` (default) | function handle | cell array | character vector

Value changed function, specified as one of the following:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

The `ValueChangedFcn` callback executes when the user changes the date by typing in the text field or by expanding the date picker and selecting a date.

This callback function can access specific information about the user's interaction with the date picker. MATLAB passes this information in a `ValueChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can get the object properties using dot notation. For example, `event.PreviousValue` gets the previously selected date. The `ValueChangedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangedData` object.

Property	Value
`Value`	New selected date
`PreviousValue`	Previously selected date
`Source`	Component that executes the callback
`EventName`	`'ValueChanged'`

The `ValueChangedFcn` callback does not execute when the user re-selects or re-types the currently selected date. The callback also does not execute when the `Value` property changes programmatically.

For more information about creating callbacks in App Designer, see "Write Callbacks in App Designer".

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object.

If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object
`'uidatepicker'`

This property is read-only.

Type of graphics object, returned as `'uidatepicker'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also
`NaT` | `appdesigner` | `datetime` | `uidatepicker`

Introduced in R2018a

dateshift

Shift date or generate sequence of dates and time

Syntax

```
t2 = dateshift(t,'start',unit)
t2 = dateshift(t,'end',unit)
t2 = dateshift(t,'dayofweek',dow)

t2 = dateshift( ___ ,rule)
```

Description

`t2 = dateshift(t,'start',unit)` shifts each value in the `datetime` array `t` back to the beginning of the unit of time specified by `unit`. The output `t2` is the same size as `t`.

`t2 = dateshift(t,'end',unit)` shifts the values ahead to the end of the unit of time specified by `unit`. The end of a day, hour, minute, or second is also the beginning of the next one. For example, the end of a day occurs at midnight at the beginning of the next day. The end of a year, quarter, month, or week is midnight at the beginning of the last day of that time period.

`t2 = dateshift(t,'dayofweek',dow)` returns the next occurrence of the specified day of the week on or after each datetime in array `t`. If the date in `t` falls on the specified day of the week, then `dateshift` returns the same date.

`t2 = dateshift( ___ ,rule)` shifts each value in array `t` according to the pattern specified by `rule`. You can use this syntax with any of the arguments in the previous syntaxes.

Examples

Shift Current Date to End of Current Month

Define the current date.

```
t = datetime('today')
```

```
t = datetime
   26-Aug-2019
```

Shift the date to the end of the same month.

```
t2 = dateshift(t,'end','month')
```

```
t2 = datetime
   31-Aug-2019
```

Shift Current Date to Next Month

Define the current date.

```
t = datetime('today')
```

```
t = datetime
   27-Aug-2019
```

Shift the date to the start of the next month.

```
t2 = dateshift(t,'start','month','next')
```

```
t2 = datetime
   01-Sep-2019
```

Shift the date to the end of the next month.

```
t2 = dateshift(t,'end','month','next')
```

```
t2 = datetime
   30-Sep-2019
```

Shift Dates to Specific Day of Week

Shift an array of dates forward to the next Friday.

```
t = datetime([2014,08,03;2014,04,15])
```

```
t = 2x1 datetime array
   03-Aug-2014
   15-Apr-2014
```

```
t2 = dateshift(t,'dayofweek','Friday')
```

```
t2 = 2x1 datetime array
   08-Aug-2014
   18-Apr-2014
```

Shift the array of dates backward to the previous Monday.

```
t2 = dateshift(t,'dayofweek','Monday','previous')
```

```
t2 = 2x1 datetime array
   28-Jul-2014
   14-Apr-2014
```

Determine Future Date

Find the date that falls at the end of the fifth week from today.

```
t = datetime('today')
```

```
t = datetime
   26-Aug-2019
```

```
t2 = dateshift(t,'end','week',5)
```

```
t2 = datetime
   05-Oct-2019
```

Create Sequence of Dates Falling on Specific Day of Week

Generate a sequence of dates consisting of the next three occurrences of Friday.

```
t = datetime('today')
```

```
t = datetime
   27-Aug-2019
```

```
t2 = dateshift(t,'dayofweek','Friday',1:3)
```

```
t2 = 1x3 datetime array
   30-Aug-2019   06-Sep-2019   13-Sep-2019
```

Input Arguments

`t` — Input date and time
`datetime` array

Input date and time, specified as a `datetime` array.

`unit` — Unit of time
`'year'` | `'quarter'` | `'month'` | `'week'` | `'day'` | `'hour'` | `'minute'` | `'second'`

Unit of time, specified as one of the following values:

- `'year'`
- `'quarter'`
- `'month'`
- `'week'`
- `'day'`

- `'hour'`
- `'minute'`
- `'second'`

`dow` — Day of Week
scalar integer | character vector | string scalar

Day of the week, specified as a scalar integer indicating the day of week number, or a character vector or string scalar containing a localized day name.

Example: `'Sunday'`

Example: 1

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`rule` — Rule for shifting datetime values
character vector | string scalar | scalar integer | array of integer values

Rule for shifting datetime values, specified as a character vector, string scalar, scalar integer, or an array of integer values. If `rule` is a character vector or string, it must be one of the following.

Value of `rule`	Description
`'next'`	Shift datetime to next unit of time or specified day. (Default rule for day of week.)
`'previous'`	Shift datetime to previous unit of time or specified day.
`'nearest'`	Shift datetime to nearest occurrence of unit of time or specified day.
`'current'`	Shift datetime within the current unit of time, or to the specified day in the current week. (Default rule for unit of time.)

If `rule` is an integer or an array of integers, then:

- When used with the input argument, `unit`, `0` corresponds to the start or end of the current unit for each datetime, `1` corresponds to the next unit, `-1` corresponds to the previous unit, and so on.
- When used with the input argument, `dow`, `0` corresponds to the specified day in the current week for each datetime, `1` corresponds to the next occurrence of the specified day, `-1` corresponds to the previous occurrence, and so on.
- `t` and `rule` must be the same size, or one must be a scalar.

If you specify `'dayofweek'` and `t` contains datetime values that fall on that day of the week, then `dateshift` treats those datetime values as the next or previous occurrences of the specified day of the week. For example, `dateshift(datetime(2015,12,24),'dayofweek','thu',rule)` returns a datetime value with a date of December 24, 2015 if `rule` is `'next'`, `'previous'`, `1`, or `-1`, because December 24, 2015 is a Thursday.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`between` | `colon`

Introduced in R2014b

datestr

Convert date and time to string format

Syntax

```
DateString = datestr(t)
DateString = datestr(DateVector)
DateString = datestr(DateNumber)

DateString = datestr( ___ ,formatOut)

DateString = datestr(DateStringIn)
DateString = datestr(DateStringIn,formatOut,PivotYear)

DateString = datestr( ___ ,'local')
```

Description

`DateString = datestr(t)` converts the datetime values in the input array `t` to text representing dates and times.

The `datestr` function returns a character array with `m` rows, where `m` is the total number of datetime values in `t`. By default, `datestr` returns text in the format, day-month-year hour:minute:second. If hour:minute:second is `00:00:00`, then the text returned has the format, day-month-year.

Note While `datestr` returns text representing dates and times, it is recommended that you use `datetime` values to represent points in time, and then convert them to text using the `char`, `cellstr`, or `string` functions.

`DateString = datestr(DateVector)` converts date vectors to text representing dates and times. The `datestr` function returns a character array with `m` rows, where `m` is the total number of date vectors in `DateVector`.

`DateString = datestr(DateNumber)` converts serial date numbers to text representing dates and times. The `datestr` function returns a character array with `m` rows, where `m` is the total number of date numbers in `DateNumber`.

`DateString = datestr( ___ ,formatOut)` specifies the format of the output text using `formatOut`. You can use `formatOut` with any of the input arguments in the previous syntaxes.

`DateString = datestr(DateStringIn)` converts `DateStringIn` to text in the format, day-month-year hour:minute:second. All dates and times represented in `DateStringIn` must have the same format.

`DateString = datestr(DateStringIn,formatOut,PivotYear)` converts `DateStringIn` to `DateString`, in the format specified by `formatOut`, and using optional `PivotYear` to interpret dates that specify the year as two characters.

`DateString = datestr( ___ ,'local')` returns the date in the language of the current locale. This language is the language you select by means of your computer's operating system. If you leave `'local'` out of the argument list, `datestr` returns text in the default language, which is US English. Use `'local'` with any of the previous syntaxes. The `'local'` argument must be last in the argument sequence.

Examples

Convert datetime Array to Date Strings

```
t = [datetime('now');datetime('tomorrow')]
```

```
t = 2x1 datetime array
   26-Aug-2019 17:38:27
   27-Aug-2019 00:00:00
```

```
DateString = datestr(t)
```

```
DateString = 2x20 char array
    '26-Aug-2019 17:38:27'
    '27-Aug-2019 00:00:00'
```

`datestr` returns text representing the dates and times in the format, day-month-year hour:minute:second.

Convert Date Vector to Text

```
DateVector = [2009,4,2,11,7,18];

datestr(DateVector)
```

```
ans =
'02-Apr-2009 11:07:18'
```

`datestr` returns text representing the date and time in the default format.

Convert Date and Time to Specific Format

Format the current date in the `mm/dd/yy` format.

You can specify this format using symbolic identifiers.

```
formatOut = 'mm/dd/yy';
datestr(now,formatOut)
```

```
ans =
'08/26/19'
```

Alternatively, you can specify this format using a numeric identifier.

```
formatOut = 2;
datestr(now,formatOut)
```

```
ans =
'08/26/19'
```

You can reformat the date and time, and also show milliseconds.

```
dt = datestr(now,'mmmm dd, yyyy HH:MM:SS.FFF AM')
```

```
dt =
'August 26, 2019  5:37:22.399 PM'
```

Convert 12-Hour Time String to 24-Hour Equivalent

Convert the 12-hour time 05:32 p.m. to its 24-hour equivalent.

```
datestr('05:32 PM','HH:MM')
```

```
ans =
'17:32'
```

Convert the 24-hour time 05:32 to its 12-hour equivalent.

```
datestr('05:32','HH:MM PM')
```

```
ans =
' 5:32 AM'
```

The use of `AM` or `PM` in the `formatOut` output text does not influence which characters actually become part of the text; they only determine whether or not to include the characters. MATLAB® selects `AM` or `PM` based on the time entered.

Convert Date String from Custom Format

Call `datenum` inside of `datestr` to specify the format of the input text representing a date.

```
formatOut = 'dd mmm yyyy';
datestr(datenum('16-04-55','dd-mm-yy',1900),formatOut)
```

```
ans =
'16 Apr 1955'
```

Convert Multiple Date Strings

Convert multiple character vectors representing dates by passing them in a cell array.

All input dates must use the same format. For example, the following command passes three dates that all use the `mm/dd/yyyy` format.

```
datestr(datenum({'09/16/2007';'05/14/1996';'11/29/2010'}, ...
    'mm/dd/yyyy'))
```

```
ans = 3x11 char array
    '16-Sep-2007'
    '14-May-1996'
    '29-Nov-2010'
```

`datestr` returns a character array of converted dates in the format, day-month-year.

Convert Date String with Values Outside Normal Range

Call `datenum` inside of `datestr` to return the expected value, because the date below uses a value outside its normal range (month=13).

```
datestr(datenum('13/24/88','mm/dd/yy'))
```

```
ans =
'24-Jan-1989'
```

Use Pivot Year

Change the pivot year to change the year range.

Use a pivot year of 1900.

```
DateStringIn = '4/16/55';
formatOut = 1;
PivotYear = 1900;
datestr(DateStringIn,formatOut,PivotYear)
```

```
ans =
'16-Apr-1955'
```

For the same date, use a pivot year of 2000.

```
PivotYear = 2000;
datestr(DateStringIn,formatOut,PivotYear)
```

```
ans =
'16-Apr-2055'
```

Return Date String in Local Language

Convert a date number to text in the language of the current locale.

Use the `'local'` argument in a French locale.

```
DateNumber = 725935;
formatOut = 'mmmm-dd-yyyy';
str = datestr(DateNumber,formatOut,'local')
```

```
str =
Juillet-17-1987
```

You can make the same call without specifying `'local'`.

```
str = datestr(DateNumber,formatOut)
```

```
str =
July-17-1987
```

In this case, the output defaults to the English language.

Input Arguments

`t` — Dates and times
`datetime` array

Dates and times, specified as a `datetime` array.

Data Types: `datetime`

`DateVector` — Date vectors
matrix

Date vectors, specified as an `m`-by-6 matrix, where `m` is the number of full (six-element) date vectors. Each element of `DateVector` must be a positive or negative integer value except for the seconds element, which can be fractional. If an element falls outside the

conventional range, `datestr` adjusts both that date vector element and the previous element. For example, if the minutes element is `70`, then `datestr` adjusts the hours element by `1` and sets the minutes element to `10`. If the minutes element is `-15`, then `datestr` decreases the hours element by `1` and sets the minutes element to `45`. Month values are an exception. `datestr` sets month values less than `1` to `1`.

Example: `[2003,10,24,12,45,07]`

Data Types: `double`

`DateNumber` — Serial date numbers
array of positive double-precision numbers

Serial date numbers, specified as an array of positive double-precision numbers.

Example: `731878`

Data Types: `double`

`formatOut` — Format of the output representing dates and times
`-1` (default) | character vector | string scalar | integer

Format of the output representing dates and times, specified as a character vector or string scalar of symbolic identifiers or an integer that corresponds to a predefined format. If you do not specify `formatOut`, then `datestr` returns text in the default format `dd-mmm-yyyy HH:MM:SS` (day-month-year hour:minute:second). By default, if `HH:MM:SS = 00:00:00` then the text returned has the format `dd-mmm-yyyy`.

The following table shows symbolic identifiers that you can use to construct the `formatOut` character vector. You can include characters such as a hyphen, space, or colon to separate the fields.

Note The symbolic identifiers describing date and time formats are different from the identifiers that describe the display formats of `datetime` arrays.

Symbolic Identifier	Description	Example
`yyyy`	Year in full	`1990`, `2002`
`yy`	Year in two digits	`90`, `02`

Symbolic Identifier	Description	Example
QQ	Quarter year using letter Q and one digit	Q1
mmmm	Month using full name	March, December
mmm	Month using first three letters	Mar, Dec
mm	Month in two digits	03, 12
m	Month using capitalized first letter	M, D
dddd	Day using full name	Monday, Tuesday
ddd	Day using first three letters	Mon, Tue
dd	Day in two digits	05, 20
d	Day using capitalized first letter	M, T
HH	Hour in two digits (no leading zeros when symbolic identifier AM or PM is used)	05, 5 AM
MM	Minute in two digits	12, 02
SS	Second in two digits	07, 59
FFF	Millisecond in three digits	057
AM or PM	AM or PM inserted in text representing time	3:45:02 PM

The `formatOut` character vector must follow these guidelines:

- You cannot specify any field more than once. For example, you cannot use `'yy-mmm-dd-m'` because it has two month identifiers. The one exception to this is that you can combine one instance of `dd` with one instance of any of the other day identifiers. For example, `'dddd mmm dd yyyy'` is a valid input.
- When you use `AM` or `PM`, the `HH` field is also required.
- You only can use `QQ` alone or with a year specifier.

This table lists predefined date formats that you can use with `datestr`.

Numeric Identifier	Date and Time Format	Example
-1 (default)	'dd-mmm-yyyy HH:MM:SS' or 'dd-mmm-yyyy' if 'HH:MM:SS'= 00:00:00	01-Mar-2000 15:45:17 or 01-Mar-2000
0	'dd-mmm-yyyy HH:MM:SS'	01-Mar-2000 15:45:17
1	'dd-mmm-yyyy'	01-Mar-2000
2	'mm/dd/yy'	03/01/00
3	'mmm'	Mar
4	'm'	M
5	'mm'	03
6	'mm/dd'	03/01
7	'dd'	01
8	'ddd'	Wed
9	'd'	W
10	'yyyy'	2000
11	'yy'	00
12	'mmmyy'	Mar00
13	'HH:MM:SS'	15:45:17
14	'HH:MM:SS PM'	3:45:17 PM
15	'HH:MM'	15:45
16	'HH:MM PM'	3:45 PM
17	'QQ-YY'	Q1-01
18	'QQ'	Q1
19	'dd/mm'	01/03
20	'dd/mm/yy'	01/03/00
21	'mmm.dd,yyyy HH:MM:SS'	Mar.01,2000 15:45:17
22	'mmm.dd,yyyy'	Mar.01,2000
23	'mm/dd/yyyy'	03/01/2000
24	'dd/mm/yyyy'	01/03/2000

Numeric Identifier	Date and Time Format	Example
25	'yy/mm/dd'	00/03/01
26	'yyyy/mm/dd'	2000/03/01
27	'QQ-YYYY'	Q1-2001
28	'mmmyyyy'	Mar2000
29	'yyyy-mm-dd' (ISO 8601)	2000-03-01
30	'yyyymmddTHHMMSS' (ISO 8601)	20000301T154517
31	'yyyy-mm-dd HH:MM:SS'	2000-03-01 15:45:17

`DateStringIn` — Text representing dates and times to convert
character vector | cell array | string array

Text representing dates and times to convert, specified as a single character vector, a cell array of character vectors, or a string array, where each row corresponds to one date and time.

`datestr` considers two-character years (for example, `'79'`) to fall within the 100-year range centered around the current year.

All text representing dates and times must have the same date format, and they must be in one of the following date formats.

Format of Text Representing Dates and Times	Example
'dd-mmm-yyyy HH:MM:SS'	01-Mar-2000 15:45:17
'dd-mmm-yyyy'	01-Mar-2000
'mm/dd/yyyy'	03/01/2000
'mm/dd/yy'	03/01/00
'mm/dd'	03/01
'mmm.dd,yyyy HH:MM:SS'	Mar.01,2000 15:45:17
'mmm.dd,yyyy'	Mar.01,2000
'yyyy-mm-dd HH:MM:SS'	2000-03-01 15:45:17

Format of Text Representing Dates and Times	Example
'yyyy-mm-dd'	2000-03-01
'yyyy/mm/dd'	2000/03/01
'HH:MM:SS'	15:45:17
'HH:MM:SS PM'	3:45:17 PM
'HH:MM'	15:45
'HH:MM PM'	3:45 PM

Note When converting from one date and time format to another, first pass the text to the `datenum` function, so that you can specify the format of the inputs. This approach ensures that the format of the input dates and times is correctly interpreted. For example, see "Convert Date String from Custom Format" on page 1-2969.

`PivotYear` — Start year of 100-year date range

present minus 50 years (default) | integer

Start year of the 100-year date range in which a two-character year resides, specified as an integer. Use a pivot year to interpret dates that specify the year as two characters.

If `formatIn` contains the time of day, the pivot year is computed from the current time of the current day, month, and year. Otherwise it is computed from midnight of the current day, month, and year.

Example: `2000`

Note If the input date format specifies a four-character year, then the last two characters are truncated, and the *first* two characters specify the year. For example, if the date and pivot year are specified as `('25122015','ddmmyyyy',2000)`, then the resulting date is `25-12-2020`, not `25-12-2015`.

Data Types: `double`

Output Arguments

`DateString` — Text representing dates and times
character vector | two-dimensional character array

Text representing dates and times, returned as a character array with `m` rows, where `m` is the total number of input dates and times. The default output format is `dd-mmm-yyyy HH:MM:SS` (day-month-year hour:minute:second) unless the hours, minutes, and seconds are all 0 in which case `HH:MM:SS` is suppressed.

Tips

- To convert text not in a predefined MATLAB date format, first convert the text to a date number, using either the `datenum` or `datevec` functions.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- The first argument must be a column vector or an array returned by `datevec`.
- If the first argument is a `datevec` array, then the rows must be within the year range `1500:2499`.

For more information, see “Tall Arrays”.

See Also

`cellstr` | `char` | `datenum` | `datetick` | `datetime` | `datevec` | `string`

Topics

“Represent Dates and Times in MATLAB”

"Converting Date Vector Returns Unexpected Output"

Introduced before R2006a

datetick

Date formatted tick labels

Syntax

```
datetick(tickaxis)
datetick(tickaxis,dateFormat)

datetick( ___ ,'keeplimits')

datetick( ___ ,'keepticks')

datetick(axes_handle, ___ )
```

Description

For 2-D line plots, it is more convenient to plot datetime values using the `plot` function. You can then format the tick labels using the `xtickformat` and `ytickformat` functions. `datetick` is useful when plotting numeric values that are serial date numbers.

`datetick(tickaxis)` labels the tick lines of the axis specified by `tickaxis` using dates, replacing the default numeric labels. `datetick` selects a label format based on the minimum and maximum limits of the specified axis. The axis data values should be serial date numbers, as returned by the `datenum` function.

`datetick(tickaxis,dateFormat)` formats the labels according to `dateFormat`.

`datetick( ___ ,'keeplimits')` changes the tick labels to date-based labels while preserving the axis limits. Append `'keeplimits'` to any of the previous syntaxes.

`datetick( ___ ,'keepticks')` changes the tick labels to date-based labels while preserving their locations. Append `'keepticks'` to any of the previous syntaxes.

`datetick(axes_handle, ___ )` labels the tick lines of an axis on the axes specified by `axes_handle`. The `axes_handle` argument can precede any of the input argument combinations in the previous syntaxes.

Examples

Label x-Axis Ticks with 2-digit Years

Graph population data for the 20th Century taken from the 1990 US census and label *x*-axis ticks with 2-digit years.

Create time data by decade.

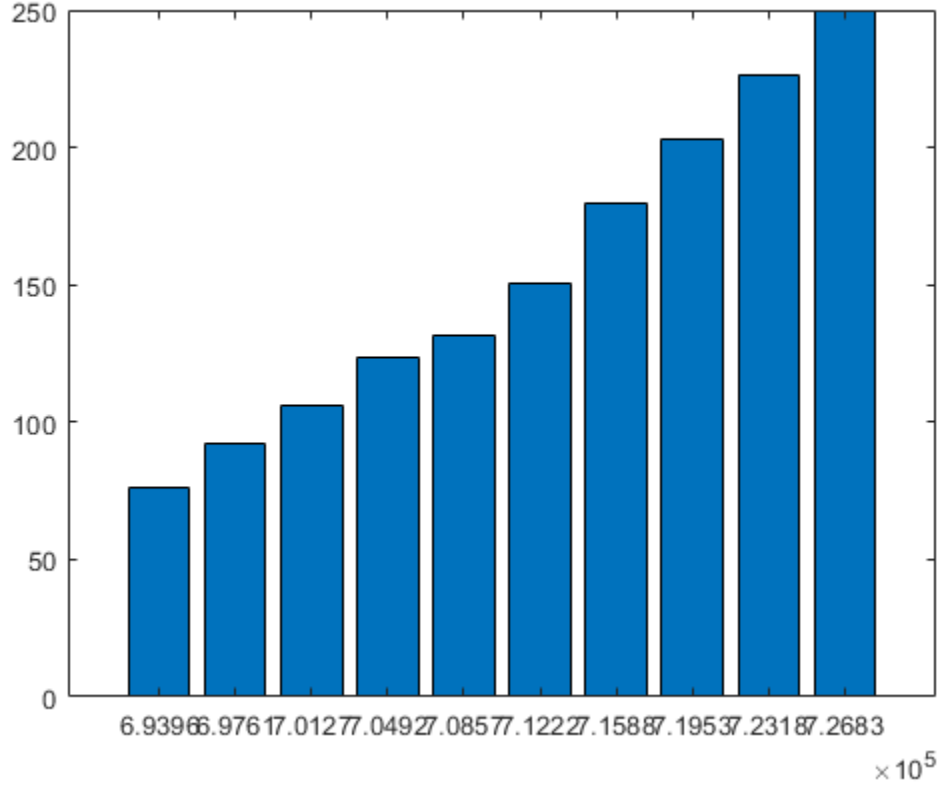
```
t = (1900:10:1990)';
```

Enter total population counts for the USA.

```
p = [75.995 91.972 105.711 123.203 131.669 ...
    150.697 179.323 203.212 226.505 249.633]';
```

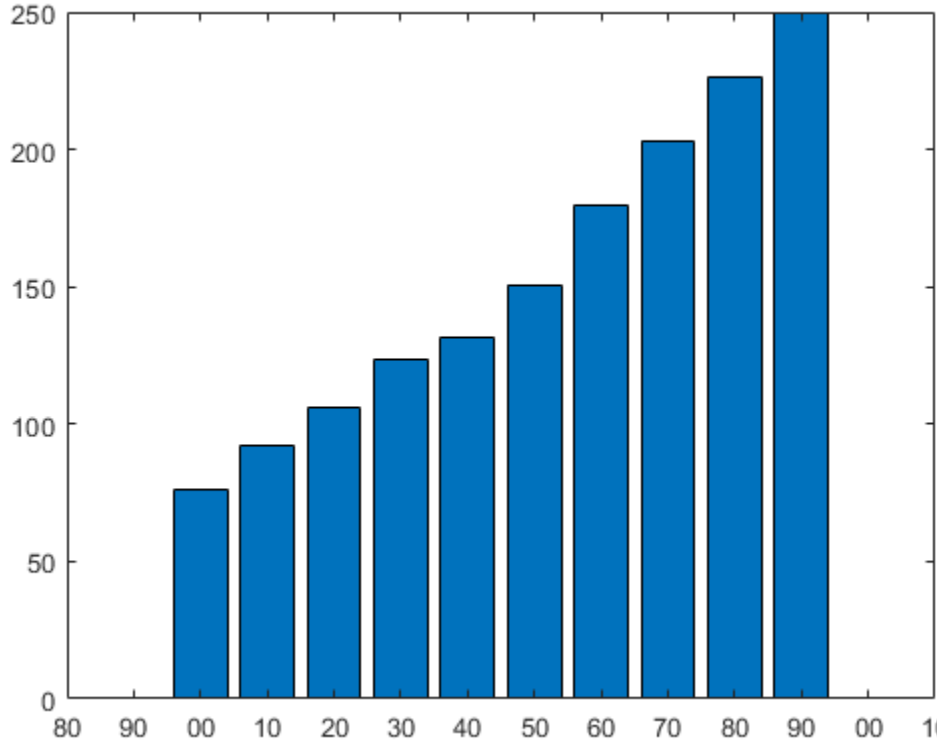
Convert years to serial date numbers using the `datenum` function, and then create a bar graph of the data.

```
figure
bar(datenum(t,1,1),p)
```



Replace *x*-axis ticks with 2-digit years. The numeric identifier `11` corresponds to the predefined MATLAB® date format `'yy'`.

```
dateFormat = 11;
datetick('x',dateFormat)
```



Label x-Axis Ticks with Hours of the Day

Plot traffic count data against date ticks for hours of the day showing AM and PM.

Get traffic count data.

```
load count.dat
```

Create arrays for an arbitrary date, for example, April 18, 1995.

```
n = length(count);
year = repmat(1995,1,n);
```

```
month = repmat(4,1,n);
day = repmat(18,1,n);
```

Create arrays for each of 24 hours.

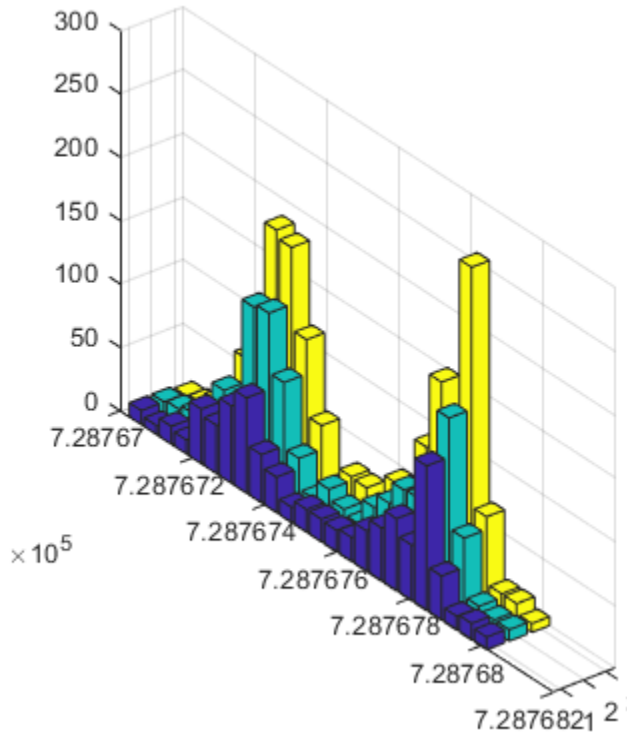
```
hour = 1:n;
minutes = zeros(1,n);
```

Get the serial date numbers for the date arrays.

```
sdate = datenum(year,month,day,hour,minutes,minutes);
```

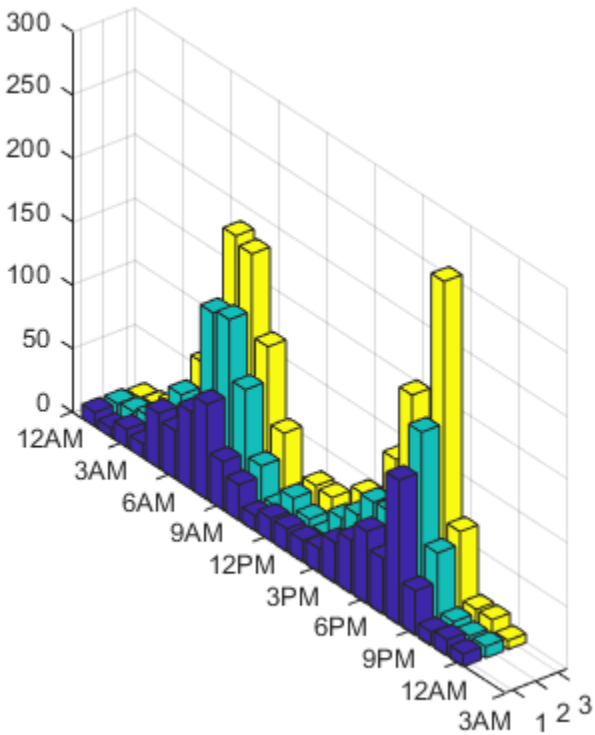
Plot a 3-D bar graph of the traffic data against the serial date numbers.

```
bar3(sdate,count)
```



Label the tick lines of the graph's *y*-axis with the hours of the day.

```
datetick('y','HHPM')
```



Label x-Axis and Preserve Axis Limits

Select a starting date.

```
startDate = datenum('02-01-1962');
```

Select an ending date.

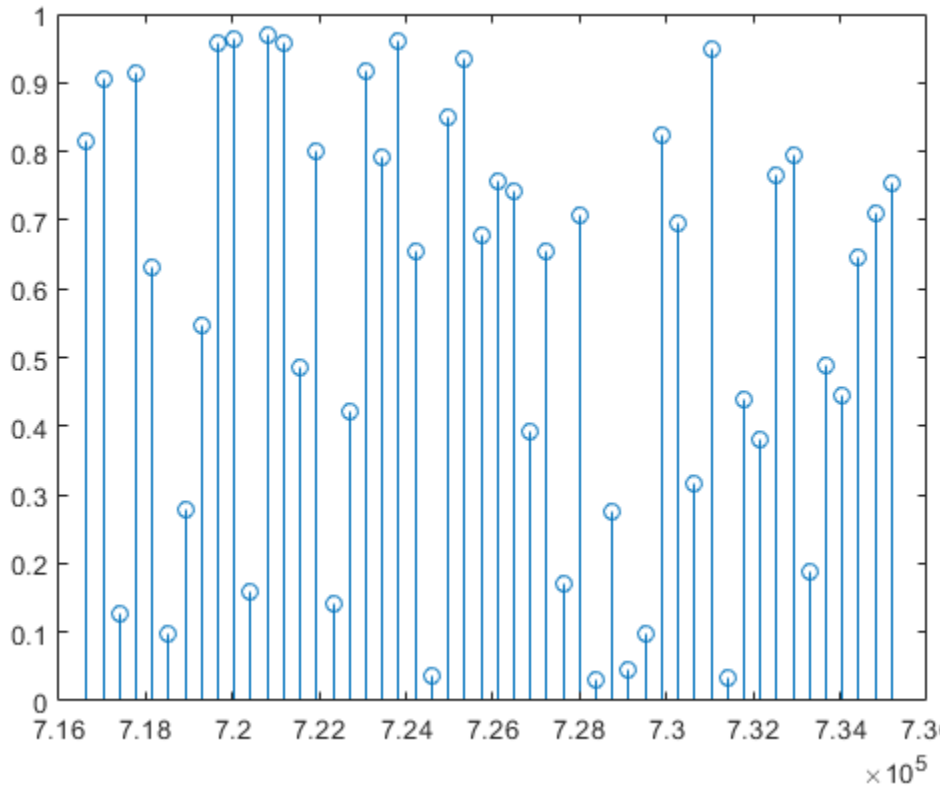
```
endDate = datenum('11-15-2012');
```

Create a variable, `xdata`, that corresponds to the number of years between the start and end dates.

```
xData = linspace(startDate,endDate,50);
```

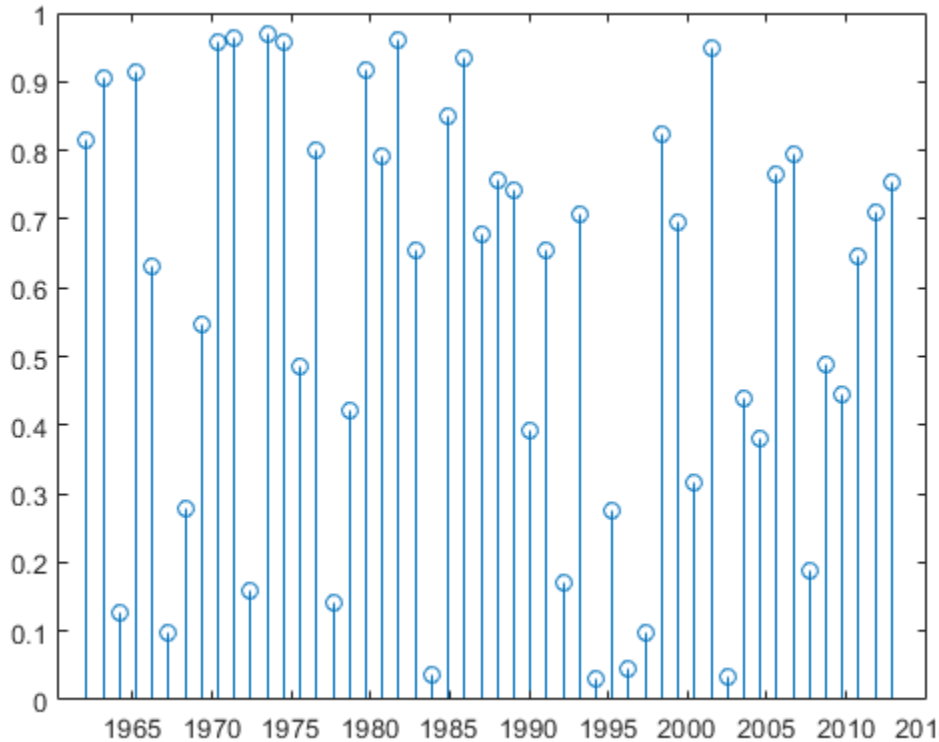
Plot random data.

```
figure
stem(xData,rand(1,50))
```



Label the *x*-axis with 4-digit years, preserving the *x*-axis limits by using the `'keeplimits'` option.

```
datetick('x','yyyy','keeplimits')
```



Add Month Labels to Plot and Preserve Number of Ticks

Select a starting date.

```
startDate = datenum('01-01-2009');
```

Select an ending date.

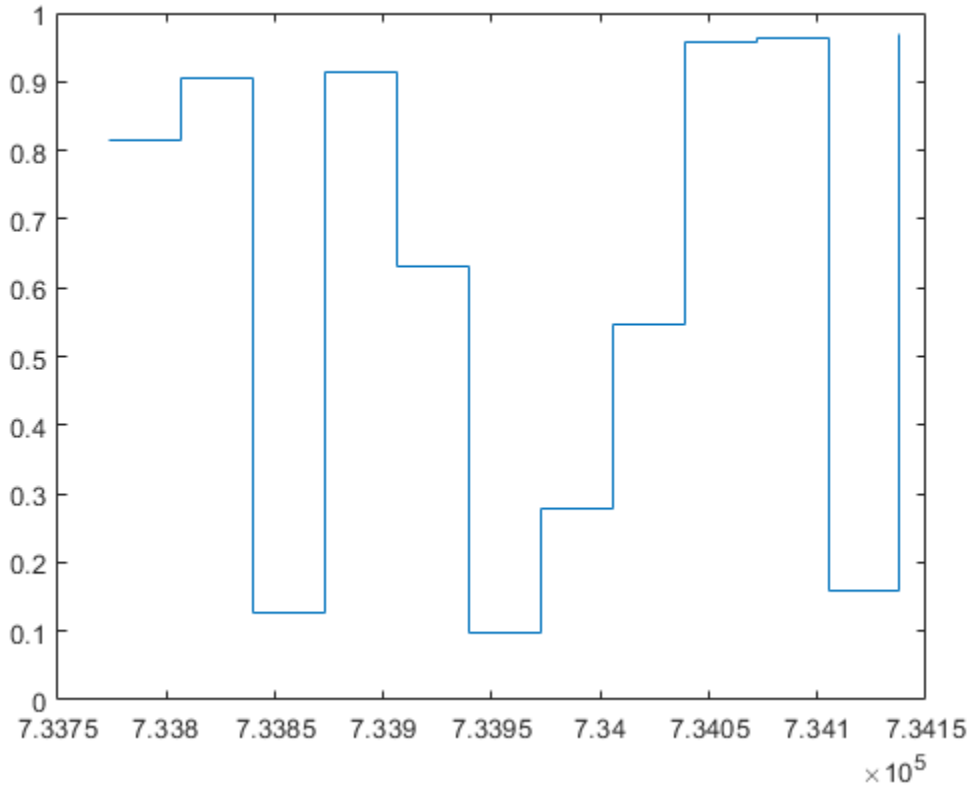
```
endDate = datenum('12-31-2009');
```

Create a variable, `xdata`, that corresponds to the number of months between the start and end dates.

```
xData = linspace(startDate,endDate,12);
```

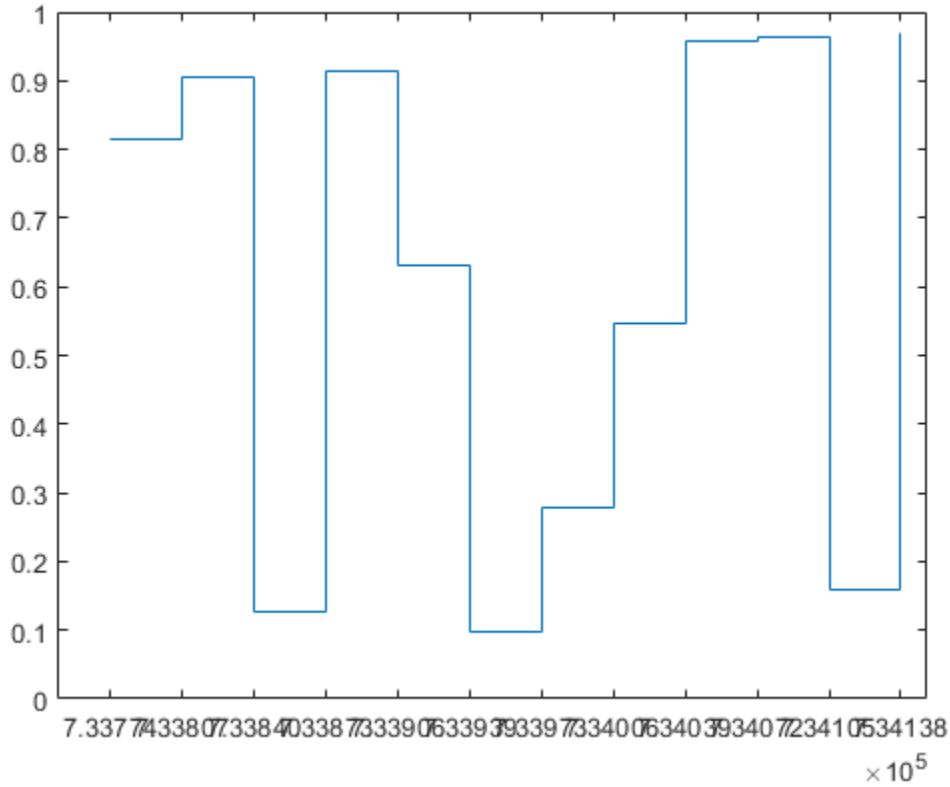
Plot random data.

```
figure
stairs(xData,rand(1,12))
```



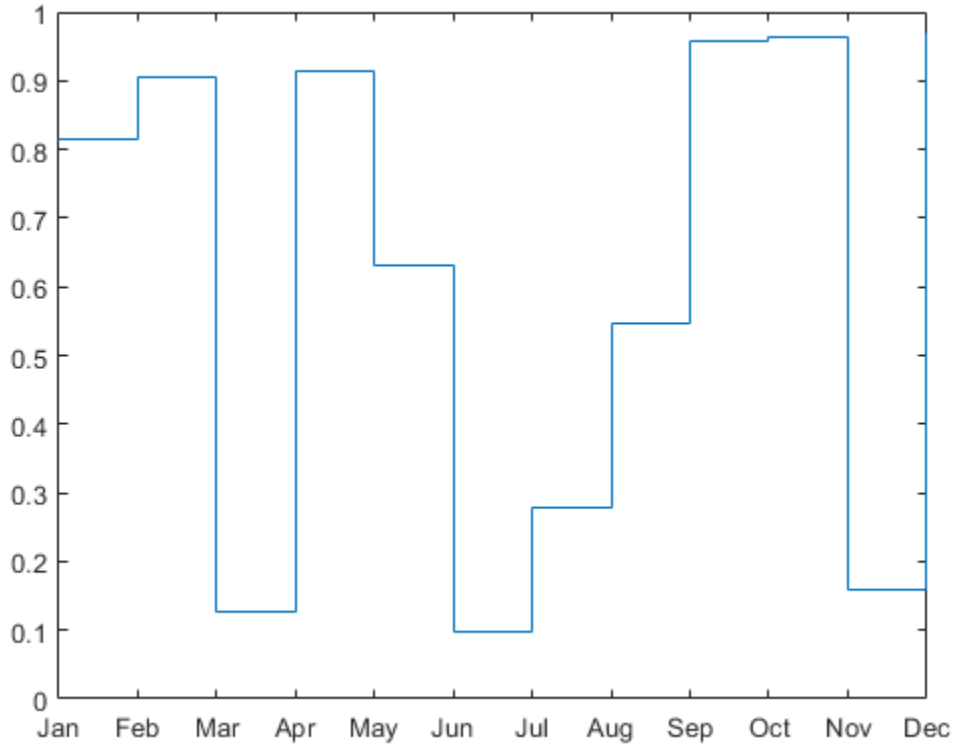
Set the number of `XTicks` to the number of points in `xData`.

```
ax = gca;
ax.XTick = xData;
```



Label the *x*-axis with month names, preserving the total number of ticks by using the `'keepticks'` option.

```
datetick('x','mmm','keepticks')
```



Input Arguments

`tickaxis` — Axis to label
`'x'` (default) | `'y'` | `'z'`

Axis to label with dates, specified as `'x'`, `'y'`, or `'z'`.

`dateFormat` — Format of tick line labels
character vector | integer

Format of the tick line labels, specified as a character vector of symbolic identifiers or an integer that corresponds to a predefined format.

The following table shows symbolic identifiers that you can use to construct the format. You can include characters such as a hyphen, space, or colon to separate the fields. For example, to display the day of the month followed by the three-letter abbreviation of the day of the week in parentheses, use `dateFormat = 'dd (ddd)'`.

Note The letter identifiers that `datetick` accepts are different from the identifiers used by the `datetime` function.

Symbolic Identifier	Description	Example
`yyyy`	Year in full	`1990`, `2002`
`yy`	Year in two digits	`90`, `02`
`QQ`	Quarter year using letter `Q` and one digit	`Q1`
`mmmm`	Month using full name	`March`, `December`
`mmm`	Month using first three letters	`Mar`, `Dec`
`mm`	Month in two digits	`03`, `12`
`m`	Month using capitalized first letter	`M`, `D`
`dddd`	Day using full name	`Monday`, `Tuesday`
`ddd`	Day using first three letters	`Mon`, `Tue`
`dd`	Day in two digits	`05`, `20`
`d`	Day using capitalized first letter	`M`, `T`
`HH`	Hour in two digits (no leading zeros when symbolic identifier `AM` or `PM` is used)	`05`, `5 AM`
`MM`	Minute in two digits	`12`, `02`
`SS`	Second in two digits	`07`, `59`
`FFF`	Millisecond in three digits	`057`
`AM or PM`	`AM` or `PM` inserted in text representing time	`3:45:02 PM`

The following table lists predefined MATLAB date formats.

Numeric Identifier	Date and Time Format	Example
-1 (default)	'dd-mmm-yyyy HH:MM:SS' or 'dd-mmm-yyyy' if 'HH:MM:SS'= 00:00:00	01-Mar-2000 15:45:17 or 01-Mar-2000
0	'dd-mmm-yyyy HH:MM:SS'	01-Mar-2000 15:45:17
1	'dd-mmm-yyyy'	01-Mar-2000
2	'mm/dd/yy'	03/01/00
3	'mmm'	Mar
4	'm'	M
5	'mm'	03
6	'mm/dd'	03/01
7	'dd'	01
8	'ddd'	Wed
9	'd'	W
10	'yyyy'	2000
11	'yy'	00
12	'mmmyy'	Mar00
13	'HH:MM:SS'	15:45:17
14	'HH:MM:SS PM'	3:45:17 PM
15	'HH:MM'	15:45
16	'HH:MM PM'	3:45 PM
17	'QQ-YY'	Q1-01
18	'QQ'	Q1
19	'dd/mm'	01/03
20	'dd/mm/yy'	01/03/00
21	'mmm.dd,yyyy HH:MM:SS'	Mar.01,2000 15:45:17
22	'mmm.dd,yyyy'	Mar.01,2000
23	'mm/dd/yyyy'	03/01/2000

Numeric Identifier	Date and Time Format	Example
24	'dd/mm/yyyy'	01/03/2000
25	'yy/mm/dd'	00/03/01
26	'yyyy/mm/dd'	2000/03/01
27	'QQ-YYYY'	Q1-2001
28	'mmmyyyy'	Mar2000
29	'yyyy-mm-dd' (ISO 8601)	2000-03-01
30	'yyyymmddTHHMMSS' (ISO 8601)	20000301T154517
31	'yyyy-mm-dd HH:MM:SS'	2000-03-01 15:45:17

Tips

- To change the tick spacing and locations, set the appropriate axes property (that is, `XTick`, `YTick`, or `ZTick`) before calling `datetick`.
- Calling `datetick` sets the `TickMode` of the specified axis to `'manual'`. This means that after zooming, panning or otherwise changing axis limits, you should call `datetick` again to update the ticks and labels.
- The best way to work with dates and times in MATLAB is to use `datetime` values, which offer more features than serial date numbers. Plot `datetime` values using the `plot` function. Use the `DatetimeTickFormat` name-value pair argument to modify the format of the axis tick labels.

Algorithms

`datetick` calls the `datestr` function to convert date numbers to text.

See Also

`datenum` | `datestr` | `datetime` | `plot`

Topics

"Plot Dates and Durations"

Introduced before R2006a

datetime

Arrays that represent points in time

Description

`datetime` arrays represent points in time using the proleptic ISO calendar. `datetime` values have flexible display formats up to nanosecond precision and can account for time zones, daylight saving time, and leap seconds.

Creation

Syntax

```
t = datetime
t = datetime(relativeDay)

t = datetime(DateStrings)
t = datetime(DateStrings,'InputFormat',infmt)

t = datetime(DateVectors)
t = datetime(Y,M,D)
t = datetime(Y,M,D,H,MI,S)
t = datetime(Y,M,D,H,MI,S,MS)

t = datetime(X,'ConvertFrom',dateType)

t = datetime( ___ ,Name,Value)
```

Description

`t = datetime` returns a scalar `datetime` array corresponding to the current date and time.

`t = datetime(relativeDay)` uses the date specified by `relativeDay`. The `relativeDay` input can be `'today'`, `'tomorrow'`, `'yesterday'`, or `'now'`.

`t = datetime(DateStrings)` creates an array of datetime values from the text in `DateStrings` representing points in time.

`t = datetime(DateStrings,'InputFormat',infmt)` interprets `DateStrings` using the format specified by `infmt`. All values in `DateStrings` must have the same format.

To avoid ambiguities between similar formats, specify `'InputFormat'` and its corresponding value, `infmt`.

`t = datetime(DateVectors)` creates a column vector of datetime values from the date vectors in `DateVectors`.

`t = datetime(Y,M,D)` creates an array of datetime values for corresponding elements of the `Y`, `M`, and `D` (year, month, day) arrays. The arrays must be of the same size (or any can be a scalar). You also can specify the input arguments as a date vector, `[Y M D]`.

`t = datetime(Y,M,D,H,MI,S)` also creates `H`, `MI`, and `S` (hour, minute, and second) arrays. All arrays must be of the same size (or any can be a scalar). You also can specify the input arguments as a date vector, `[Y M D H MI S]`.

`t = datetime(Y,M,D,H,MI,S,MS)` adds an `MS` (millisecond) array. All arrays must be of the same size (or any can be a scalar).

`t = datetime(X,'ConvertFrom',dateType)` converts the numeric values in `X` to a `datetime` array `t`. The `dateType` argument specifies the type of values in `X`.

If `X` contains POSIX® times or Julian dates that represent local times, then specify the appropriate time zone for `t` using the `'TimeZone'` name-value pair argument. If you do not specify a time zone, then the POSIX times or Julian dates in `X` are treated as UTC times, not local times.

`t = datetime( ___ ,Name,Value)` specifies additional options using one or more name-value pair arguments, in addition to any of the input arguments in the previous syntaxes. For example, you can specify the display format of `t` using the `'Format'` name-value pair argument.

For best performance when creating datetime values from text, specify either `'Format'` or `'InputFormat'` and its corresponding value, `infmt`.

Input Arguments

`relativeDay` — Day relative to current date
`'yesterday'` | `'today'` | `'tomorrow'` | `'now'`

Day relative to the current date, specified as one of the following values.

Value of `relativeDay`	Description
`'yesterday'`	Date of the previous day, at midnight
`'today'`	Current date, at midnight
`'tomorrow'`	Date of the following day, at midnight
`'now'`	Current date and time

`DateStrings` — Text representing dates and times
character array | cell array of character vectors | string array

Text representing dates and times, specified as a character array, a cell array of character vectors, or a string array. The `datetime` function first attempts to match the format of `DateStrings` to common formats. If you know the format, specify `'InputFormat'` and its corresponding `infmt` value, or the `'Format'` name-value pair argument.

Example: `'24-Oct-2014 12:45:07'`

Example: `{'15-Oct-2013','20-Nov-2014'}`

Example: `["11-Nov-2016","12-Dec-2016"]`

Data Types: `char` | `cell` | `string`

`infmt` — Format of input text
character vector | string scalar

Format of the input text representing dates and times, specified as a character vector or string scalar that contains letter identifiers:

- If `infmt` does not include a date specifier, then `datetime` assumes that the values in `DateStrings` occur during the current day.
- If `infmt` does not include a time specifier, then `datetime` assumes that the values in `DateStrings` occur at midnight.

This table shows several common input formats and includes examples of the formatted input for the date, Saturday, April 19, 2014 at 9:41:06.12345 PM in New York City.

Value of infmt	Example
'yyyy-MM-dd'	2014-04-19
'dd/MM/yyyy'	19/04/2014
'dd.MM.yyyy'	19.04.2014
'yyyy 年 MM 月 dd 日'	2014 年 04 月 19 日 (Characters for Japanese locale, ja_JP)
'MMMM d, yyyy'	April 19, 2014
'eeee, MMMM d, yyyy h:mm a'	Saturday, April 19, 2014 9:41 PM
'MMMM d, yyyy HH:mm:ss Z'	April 19, 2014 21:41:06 -0400
'yyyy-MM-dd''T''HH:mmXXX'	2014-04-19T21:41-04:00
'yyyy-MM-dd HH:mm:ss.SSS'	2014-04-19 21:41:06.123

For input text that represents fractional seconds, you can specify `infmt` with up to nine `S` characters to indicate fractional second digits. For example, `'yyyy-MM-dd HH:mm:ss.SSS'` is a format for text that represents dates and times to millisecond precision.

For a complete list of valid letter identifiers, see the `Format` property for datetime arrays.

Note The letter identifiers that `datetime` accepts are different from the identifiers used by the `datestr`, `datenum`, and `datevec` functions.

Data Types: `char` | `string`

`DateVectors` — Date vectors
matrix

Date vectors, specified as an `m`-by-6 or `m`-by-3 matrix containing `m` full or partial date vectors, respectively. A full date vector has six elements, specifying year, month, day, hour, minute, and second, in that order. A partial date vector has three elements, specifying year, month, and day, in that order. Each element of `DateVector` should be a positive or negative integer value except for the seconds element, which can be fractional. If an element falls outside the conventional range, `datetime` adjusts both that date vector element and the previous element. For example, if the minutes element is `70`, then `datetime` adjusts the hours element by `1` and sets the minutes element to `10`. If the

minutes element is -15, then datetime decreases the hours element by 1 and sets the minutes element to 45.

Example: [2014 10 24 12 45 07]

Example: [2014 10 24]

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Y,M,D — Year, month, and day arrays
numeric arrays

Year, month, and day arrays specified as numeric arrays. These arrays must be the same size, or any one can be a scalar. Y,M,D should be integer values.

- If Y,M,D are all scalars or all column vectors, then you can specify the input arguments as a date vector, [Y M D].
- If an element of the Y, M, or D inputs falls outside the conventional range, then datetime adjusts both that element and the same element of the previous input. For details, see the description for the DateVectors input argument.

Example: 2003,10,24

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Y,M,D,H,MI,S — Year, month, day, hour, minute, and second arrays
numeric arrays

Year, month, day, hour, minute, and second arrays specified as numeric arrays. These arrays must be the same size, or any one can be a scalar. Specify fractional seconds as part of the seconds input, S. The Y,M,D,H,MI arrays must contain integer values.

- If Y,M,D,H,MI,S are all scalars or all column vectors, then you can specify the input arguments as a date vector[Y M D H MI S].
- If an element of the Y, M, D, H, MI, or S inputs falls outside the conventional range, then datetime adjusts both that element and the same element of the previous input. For details, see the description for the DateVectors input argument.

Example: 2003,10,24,12,45,07.451

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

`Y,M,D,H,MI,S,MS` — Year, month, day, hour, minute, second, and millisecond arrays
numeric arrays

Year, month, day, hour, minute, second, and millisecond arrays, specified as numeric arrays. These arrays must be the same size, or any one can be a scalar. The `Y,M,D,H,MI,S` arrays must contain integer values. `MS` can contain fractional milliseconds.

If an element of the `Y`, `M`, `D`, `H`, `MI`, `S`, or `MS` inputs falls outside the conventional range, then `datetime` adjusts both that element and the same element of the previous input. For details, see the description for the `DateVectors` input argument.

Example: `2003,10,24,12,45,07,10.52`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`X` — Numeric values
array

Numeric values, specified as an array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`dateType` — Type of values in `X`
`'datenum'` | `'excel'` | `'excel1904'` | `'juliandate'` | ...

Type of values in `X`, specified as one of these values.

Value of `dateType`	Type of Values in `X`
`'datenum'`	Number of days since 0-Jan-0000 (proleptic ISO calendar).

Value of dateType	Type of Values in X
`'excel'`	Number of days since 0-Jan-1900. Excel date numbers are rounded to the nearest microsecond. **Note:** Excel incorrectly assumes that the year 1900 is a leap year. Therefore, when computing Excel date numbers there is a discontinuity of one extra day between February 28, 1900 and March 1, 1900.
`'excel1904'`	Number of days since 0-Jan-1904. Excel date numbers are rounded to the nearest microsecond. **Note:** Excel incorrectly assumes that the year 1900 is a leap year. Therefore, when computing Excel date numbers there is a discontinuity of one extra day between February 28, 1900 and March 1, 1900.

Value of dateType	Type of Values in X

`'juliandate'`	Number of days since noon UTC 24-Nov-4714 BCE (proleptic Gregorian calendar).
	
	If you convert `X` to a `datetime` array without specifying a time zone, then the `datetime` values represent UTC times, not local times. To represent local times, specify a time zone using the `'TimeZone'` name-value pair argument.
	
	Example: Convert `X` using the time zone for New York.
	
	`T = datetime(X,'ConvertFrom','juliandate',...`
	`'TimeZone','America/New_York')`
	
	Then, you can convert `T` to an unzoned `datetime` array representing local times by assigning an empty character vector to `TimeZone`.
	
	`T.TimeZone = ''`

Value of dateType	Type of Values in X

`'modifiedjuliandate'`	Number of days since midnight UTC 17-Nov-1858.
	
	If you convert `X` to a `datetime` array without specifying a time zone, then the `datetime` values represent UTC times, not local times. To represent local times, specify a time zone using the `'TimeZone'` name-value pair argument.
	
	Example: Convert `X` using the time zone for New York.
	
	`T = datetime(X,'ConvertFrom','modifiedjuliandate`
	`'TimeZone','America/New_York')`
	
	Then, you can convert `T` to an unzoned `datetime` array representing local times by assigning an empty character vector to `TimeZone`.
	
	`T.TimeZone = ''`

Value of dateType	Type of Values in X

'posixtime'	Number of seconds since 1-Jan-1970 00:00:00 UTC, not counting leap seconds.
	
	If you convert X to a datetime array without specifying a time zone, then the datetime values represent UTC times, not local times. To represent local times, specify a time zone using the 'TimeZone' name-value pair argument.
	
	Example: Convert X using the time zone for New York.
	
	`T = datetime(X,'ConvertFrom','posixtime',...`
	`'TimeZone','America/New_York')`
	
	Then, you can convert T to an unzoned datetime array representing local times by assigning an empty character vector to TimeZone.
	
	`T.TimeZone = ''`

'yyyymmdd'	Dates as YYYYMMDD numeric values. For example, 20140402 represents April 2, 2014.
'ntp'	Number of "clock ticks" since 1-Jan-1900 00:00:00 UTC, where each clock tick is 2^-32 seconds.
'.net'	Number of "clock ticks" since 1-Jan-0001 00:00:00 UTC, where each clock tick is 100 ns.
'ntfs'	Number of "clock ticks" since 1-Jan-1601 00:00:00 UTC, where each clock tick is 100 ns.

Value of dateType	Type of Values in X

`'epochtime','Epoch',`*`epochValue`*	Number of seconds since an epoch.
	
	You must also specify *`epochValue`*, which is a scalar `datetime`, or a character vector or string scalar representing the epoch time.
	
	Example: Return the number of days since January 1, 2000.
	
	`T = datetime(X,'ConvertFrom',...`
	`'epochtime','Epoch','2000-01-01')`

`'epochtime','Epoch',`*`epochValue`*`,'TicksPerSecond',`*`n`*	Number of "clock ticks" since an epoch.
	
	In addition to specifying *`epochValue`*, you also can specify *`n`*, which is a scalar integer that specifies the number of "clock ticks" per second.
	
	Example: Return the number of days since January 1, 2000, where `X` represents the time in milliseconds since or before that date.
	
	`T = datetime(X,'ConvertFrom',...`
	`'epochtime','Epoch','2000-01-01','TicksPerSecond`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Format','eeee MMMM d, y','TimeZone','local'` applies a display format to datetime values and specifies the local time zone.

`Format` — Display format
`'default'` | `'defaultdate'` | `'preserveinput'` | character vector | string scalar

Display format of the values in the output array, specified as the comma-separated pair consisting of `'Format'` and one of the following values.

Value of `Format`	Description
`'default'`	Use the default display format.
`'defaultdate'`	Use the default display format for datetime values created without time components.
`'preserveinput'`	Use the format specified by the input format, `infmt`. If you do not specify `infmt`, then `datetime` determines the format automatically.
character vector or string scalar	Use a format you specify with a character vector or string scalar.

If you specify your own format, then you must use valid letter identifiers. For a complete list of valid letter identifiers, see the `Format` property.

The factory default format depends on your system locale. To change the default display format, see "Default datetime Format".

If you specify a `DateStrings` input but do not specify the `'InputFormat'` parameter, then `datetime` tries to use the `Format` value to interpret `DateStrings`.

Example: `'Format','eeee, MMMM d, yyyy HH:mm:ss'` displays a date and time such as `Wednesday, April 9, 2014 21:41:06`.

Data Types: `char` | `string`

`Locale` — Locale of `DateStrings`
character vector | string scalar

Locale of `DateStrings` input argument values, specified as the comma-separated pair consisting of `'Locale'` and a character vector or string scalar. The `Locale` value determines how `datetime` interprets `DateStrings`. However, it does not determine how the output `datetime` values display.

The `Locale` value can be:

- `'system'`, to specify your system locale.

- A character vector or string scalar in the form *xx_YY*, where *xx* is a lowercase ISO 639-1 two-letter code that specifies a language, and *YY* is an uppercase ISO 3166-1 alpha-2 code that specifies a country.

This table lists some common values for the locale.

Locale	Language	Country
`'de_DE'`	German	Germany
`'en_GB'`	English	United Kingdom
`'en_US'`	English	United States
`'es_ES'`	Spanish	Spain
`'fr_FR'`	French	France
`'it_IT'`	Italian	Italy
`'ja_JP'`	Japanese	Japan
`'ko_KR'`	Korean	Korea
`'nl_NL'`	Dutch	Netherlands
`'zh_CN'`	Chinese (simplified)	China

You can use the `'Locale'` name-value pair only when you use the `DateStrings` input argument.

Example: `'Locale','de_DE'`

Note The `Locale` value determines how input text values are interpreted. The output datetime values always display in the language specified by the **Locale** option in the **Datetime format** section of the Preferences panel.

- To change the default datetime locale for display, see “Set Command Window Preferences”.
- To convert datetime values to text using a different locale, use the `char`, `cellstr`, or `string` functions.

Data Types: `char` | `string`

`PivotYear` — Start year of 100-year date range
`year(datetime('now'))-50` (default) | integer

Start year of the 100-year date range in which a two-character year resides, specified as the comma-separated pair consisting of `'PivotYear'` and an integer. Use a pivot year to interpret dates that specify the year as two characters. That is, the pivot year has an effect only when the `infmt` argument includes `y` or `yy`.

You can use the `'PivotYear'` name-value pair only when you use the `DateStrings` input argument.

Example: `'PivotYear',1900`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`TimeZone` — Time zone
`''` (default) | character vector | string scalar

Time zone, specified as the comma-separated pair consisting of `'TimeZone'` and a character vector or string scalar that specifies the `TimeZone` property.

Data Types: `char` | `string`

Properties

`Format` — Display format
`'default'` | `'defaultdate'` | character vector | string scalar

Display format, specified as `'default'`, `'defaultdate'`, a character vector, or a string scalar.

Value of `Format`	Description
`'default'`	Use the default display format.
`'defaultdate'`	Use the default display format for datetime values created without time components.
character vector or string scalar	Use a format you specify with a character vector or string scalar.

If you specify a format of your own, then it must use valid letter identifiers that correspond to the Unicode Locale Data Markup Language (LDML) standard for dates and times.

Example: `'eeee, MMMM d, yyyy HH:mm:ss'` displays a date and time such as `Wednesday, April 9, 2014 21:41:06`.

To separate fields, you can include nonletter characters such as a hyphen, space, colon, or any non-ASCII characters. To include the letters `A-Z` and `a-z` as literal characters in the format, enclose them in single quotes.

Example: `'uuuu-MM-dd''T''HH:mm:ss'` displays a date and time, such as `2014-04-09T21:41:06`.

Note The letter identifiers that `datetime` accepts are different from the identifiers used by the `datestr`, `datenum`, and `datevec` functions.

The factory default format depends on your system locale. To change the default display format, see “Default datetime Format”.

Examples of Common Formats

This table lists common display formats. The examples show formatted output for the date, Wednesday, April 9, 2014 at 9:41:06.12345 PM, in New York City.

Value of Format	Example
`'yyyy-MM-dd'`	`2014-04-09`
`'dd/MM/yyyy'`	`09/04/2014`
`'dd.MM.yyyy'`	`09.04.2014`
`'yyyy 年 MM 月 dd 日'`	`2014 年 04 月 09 日` (Characters for Japanese locale, `ja_JP`)
`'MMMM d, yyyy'`	`April 9, 2014`
`'eeee, MMMM d, yyyy h:mm a'`	`Wednesday, April 9, 2014 9:41 PM`
`'MMMM d, yyyy HH:mm:ss Z'`	`April 9, 2014 21:41:06 -0400`
`'yyyy-MM-dd''T''HH:mmXXX'`	`2014-04-09T21:41-04:00`
`'yyyy-MM-dd HH:mm:ss.SSS'`	`2014-04-09 21:41:06.123`

All Date and Time Formats

Use these identifiers to specify the display formats of date and time fields. The display formats show output for the date, Wednesday, April 9, 2014 at 9:41:06.12345 PM, in New York City.

Letter Identifier	Description	Display
`G`	Era	`CE`
`y`	Year, with no leading zeros. See the Note that follows this table.	`2014`
`yy`	Year, using last two digits. See the Note that follows this table.	`14`
`yyy`, `yyyy` ...	Year, using at least the number of digits specified by the number of instances of `'y'`	For the year 2014, `'yyy'` displays `2014`, while `'yyyyy'` displays `02014`.
`u`, `uu`, ...	ISO year. A single number designating the year. An ISO year value assigns positive values to CE years and negative values to BCE years, with 1 BCE being year 0.	`2014`
`Q`	Quarter, using one digit	`2`
`QQ`	Quarter, using two digits	`02`
`QQQ`	Quarter, abbreviated	`Q2`
`QQQQ`	Quarter, full name	`2nd quarter`
`M`	Month, numerical using one or two digits	`4`
`MM`	Month, numerical using two digits	`04`
`MMM`	Month, abbreviated name	`Apr`
`MMMM`	Month, full name	`April`
`MMMMM`	Month, capitalized first letter	`A`
`W`	Week of the month, using one digit	`2`
`d`	Day of the month, using one or two digits	`9`

Letter Identifier	Description	Display
dd	Day of the month using two digits	09
D	Day of the year, using one, two, or three digits	99
DD	Day of the year using two digits	99
DDD	Day of the year using three digits	099
e	Day of the week (numerical, using one or two digits)	4, where Sunday is the first day of the week.
ee	Day of the week (numerical, using two digits)	04
eee	Day, abbreviated name	Wed
eeee	Day, full name	Wednesday
eeeee	Day, capitalized first letter	W
a	Day period (AM or PM)	PM
h	Hour, 12-hour clock notation using one or two digits	9
hh	Hour, 12-hour clock notation using two digits	09
H	Hour, 24-hour clock notation using one or two digits	21
HH	Hour, 24-hour clock notation using two digits	21
m	Minute, using one or two digits	41
mm	Minute, using two digits	41
s	Second, using one or two digits	6
ss	Second, using two digits	06
S, SS, ..., SSSSSSSSS	Fractional second, using the number of digits specified by the number of instances of 'S' (up to 9 digits).	'SSS' truncates 6.12345 seconds to 6.123.

Note

- If you read a two-digit year number and specify the format as `y` or `yy`, then the pivot year determines the century to which the year belongs.
- If you read a day-of-year number and specify a format that contains identifiers for both the day of year (`D`) and Gregorian year (`y`), then `datetime` might not read the day-of-year number correctly. Use ISO year (`u`) in place of `y`.
- Use one or more `u` characters instead of `y` characters to represent the year when working with year numbers near zero.
- Datetime values later than 144683 years CE or before 140743 BCE display only the year numbers, regardless of the specified `Format` value.

Time Zone Offset Formats

Use these identifiers to specify the display format of the time zone offset. A time zone offset is the amount of time that a specific datetime is offset from UTC. A time zone has rules for determining the time zone offset, and the offset can differ at different times of the year. Include a time zone offset identifier in the display format for a datetime array when you want to ensure that the time components are displayed unambiguously.

Letter Identifier	Description	Display
`z`	Abbreviated name of the time zone offset. If this value is not available, then the time zone offset uses the short UTC format, such as `UTC-4`.	`EDT`
`Z`	ISO 8601 basic format with hours, minutes, and optional seconds fields.	`-0400`
`ZZZZ`	Long UTC format.	`UTC-04:00`
`ZZZZZ`	ISO 8601 extended format with hours, minutes, and optional seconds fields. A time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	`-04:00`

Letter Identifier	Description	Display
x or X	ISO 8601 basic format with hours field and optional minutes field. If you specify X, then a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-04
xx or XX	ISO 8601 basic format with hours and minutes fields. If you specify XX, then a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-0400
xxx or XXX	ISO 8601 extended format with hours and minutes fields. If you specify XXX, then a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-04:00
xxxx or XXXX	ISO 8601 basic format with hours, minutes, and optional seconds fields. If you specify XXXX, then a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-0400
xxxxx or XXXXX	ISO 8601 extended format with hours, minutes, and optional seconds fields. If you specify XXXXX, then a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-04:00

`TimeZone` — Time zone

`''` (default) | character vector | string scalar

Time zone, specified as a character vector or string scalar. The value of `TimeZone` specifies the time zone that the `datetime` function uses to interpret the input data. `TimeZone` also specifies the time zone of the output array. If the input data are character vectors or strings that include a time zone, then the `datetime` function converts all values to the specified time zone.

The value of `TimeZone` can be:

- `''`, to create an “unzoned” `datetime` array that does not belong to a specific time zone.
- The name of a time zone region from the IANA Time Zone Database; for example, `'America/Los_Angeles'`. The name of a time zone region accounts for the current and historical rules for standard and daylight offsets from UTC that are observed in a geographic region.
- An ISO 8601 character vector of the form `+HH:mm` or `-HH:mm`; for example, `'+01:00'`, to specify a time zone that is a fixed offset from UTC.
- `'UTC'`, to create a `datetime` array in Universal Coordinated Time.
- `'UTCLeapSeconds'`, to create a `datetime` array in Universal Coordinated Time that accounts for leap seconds.
- `'local'`, to create a `datetime` array in the system time zone. When you query the `TimeZone` property, the IANA value is returned.

This table lists some common names of time zone regions from the IANA Time Zone Database.

Value of `TimeZone`	UTC Offset	UTC DST Offset
`'Africa/Johannesburg'`	+02:00	+02:00
`'America/Chicago'`	−06:00	−05:00
`'America/Denver'`	−07:00	−06:00
`'America/Los_Angeles'`	−08:00	−07:00
`'America/New_York'`	−05:00	−04:00
`'America/Sao_Paulo'`	−03:00	−02:00
`'Asia/Hong_Kong'`	+08:00	+08:00
`'Asia/Kolkata'`	+05:30	+05:30
`'Asia/Tokyo'`	+09:00	+09:00
`'Australia/Sydney'`	+10:00	+11:00
`'Europe/London'`	+00:00	+01:00
`'Europe/Zurich'`	+01:00	+02:00

You also can use the `timezones` function to display a list of IANA time zone names that the `datetime` function accepts.

Data Types: `char`

`Year` — Year number
numeric array

Year number of each value in the `datetime` array, specified as a numeric array that is the same size and shape as the `datetime` array. Each year number is an integer value based on the ISO calendar. Years in the current era are positive and years in the previous era are zero or negative. For example, the year number of 1 BCE is 0.

If you set the `Year` property to a nonleap year for a `datetime` value that occurs on a leap day (February 29), then the `Day` and `Month` properties change to March 1.

`Month` — Month number
numeric array

Month number of each value in the `datetime` array, specified as a numeric array that is the same size and shape as the `datetime` array. Each month number is an integer value from 1 to 12. If you set a value outside that range, then the `Year` property adjusts accordingly, and the `Month` property stays within the range 1 to 12. For example, month 0 corresponds to month 12 of the previous year. For historical dates, the month number is based on the proleptic Gregorian calendar.

`Day` — Day-of-month number
numeric array

Day-of-month number of each value in the `datetime` array, specified as a numeric array that is the same size and shape as the `datetime` array. Each day-of-month number is an integer value from 1 to 28, 29, 30, or 31, depending on the month and year. If you set a value outside that range, then the `Month` and `Year` properties adjust accordingly, and the `Day` property stays within the appropriate range. For example, day 0 corresponds to the last day of the previous month. For historical dates, the day number is based on the proleptic Gregorian calendar.

`Hour` — Hour number
numeric array

Hour number of each value in the `datetime` array, specified as a numeric array that is the same size and shape as the `datetime` array. Each hour number is an integer value from 0 to 23. If you set a value outside that range, then the `Day`, `Month`, and `Year` properties adjust accordingly, and the `Hour` property stays within the appropriate range. For example, hour -1 corresponds to hour 23 of the previous day.

These conditions apply to datetime arrays with a specific time zone that follows daylight saving time:

- When you specify a value for the `Hour` property that would create a nonexistent datetime in the hour gap when daylight saving time begins, the value of the `Hour` property adjusts to the next hour.
- When you specify a value for the `Hour` property that would create an ambiguous datetime in the hour overlap when daylight saving time ends, the datetime adjusts to the second of the two times (in standard time) with that hour.

`Minute` — Minute number
numeric array

Minute number of each value in the `datetime` array, specified as a numeric array that is the same size and shape as the `datetime` array. Each minute number is an integer value from 0 to 59. If you specify a value outside that range, then the `Hour`, `Day`, `Month`, and `Year` properties adjust accordingly, and the `Minute` property stays within the appropriate range. For example, minute -1 corresponds to minute 59 of the previous hour.

`Second` — Second number
numeric array

Second number of each value in the `datetime` array, specified as a numeric array that is the same size and shape as the `datetime` array. Each second value is a floating-point value ordinarily ranging from 0 to less than 60. If you set a value outside that range, then the `Minute`, `Hour`, `Day`, `Month`, and `Year` properties adjust accordingly, and the `Second` property stays within the appropriate range. For example, second -1 corresponds to second 59 of the previous minute.

A `datetime` array with a `TimeZone` value of `'UTCLeapSeconds'` has seconds ranging from 0 to less than 61. The values from 60 to 61 represent datetimes that occur during a leap second.

`SystemTimeZone` — System time zone setting
character vector | string scalar

This property is read-only.

System time zone setting, specified as a character vector or string scalar. The system where MATLAB is running determines this time zone setting.

Example: `America/New_York`

Examples

Current Date and Time in Specific Time Zone

Specify the current date and time in the local system time zone.

```
t = datetime('now','TimeZone','local','Format','d-MMM-y HH:mm:ss Z')
```

```
t = datetime
   26-Aug-2019 17:39:37 -0400
```

Specify the current date and time in the time zone represented by Seoul, Korea

```
t = datetime('now','TimeZone','Asia/Seoul','Format','d-MMM-y HH:mm:ss Z')
```

```
t = datetime
   27-Aug-2019 06:39:37 +0900
```

Date and Time from Character Vectors

Create a `datetime` array from a cell array of character vectors.

```
DateStrings = {'2014-05-26';'2014-08-03'};
t = datetime(DateStrings,'InputFormat','yyyy-MM-dd')
```

```
t = 2x1 datetime array
   26-May-2014
   03-Aug-2014
```

The datetime values in `t` display using the default format, and not the format of the input dates.

Date and Time from String Array

Starting in R2016b, you can create string arrays with the `string` function and convert them to datetime values.

```
str = string({'2016-03-24','2016-04-19'})
```

```
str = 1x2 string array
    "2016-03-24"    "2016-04-19"
```

Convert the strings, specifying the input format as `yyyy-MM-dd`. The format must be specified as a character vector, even though `str` is a string array.

```
t = datetime(str,'InputFormat','yyyy-MM-dd')
```

```
t = 1x2 datetime array
   24-Mar-2016   19-Apr-2016
```

Time from Text Representing Fractional Seconds

Create a `datetime` value from text that represents a date and time to millisecond precision. To convert text in a format that the `datetime` function cannot parse without more information, specify the `'InputFormat'` name-value pair argument.

```
d = '2018-06-25 11:23:37.712';
t = datetime(d,'InputFormat','yyyy-MM-dd HH:mm:ss.SSS')
```

```
t = datetime
   25-Jun-2018 11:23:37
```

The conversion does keep the fractional seconds. However, by default `datetime` arrays do not *display* fractional seconds. To display them, specify either the `'Format'` name-value pair or the `Format` property.

```
t.Format = 'MMM dd, yyyy HH:mm:ss.SSS'
```

```
t = datetime
   Jun 25, 2018 11:23:37.712
```

The `'InputFormat'` argument applies only to conversions from input text. The `Format` property specifies the display for any `datetime` array.

Date and Time from Text with Literal Characters

Convert dates in ISO 8601 format to datetime values.

Create a cell array of character vectors containing dates in ISO 8601 format. In this format, the letter T is used as a delimiter that separates a date and a time. Each character vector includes a time zone offset. The letter Z indicates no offset from UTC.

```
DateStrings = {'2014-05-26T13:30-05:00';'2014-08-26T13:30-04:00';'2014-09-26T13:30Z'}
```

```
DateStrings = 3x1 cell array
    {'2014-05-26T13:30-05:00'}
    {'2014-08-26T13:30-04:00'}
    {'2014-09-26T13:30Z'     }
```

Convert the character vectors to datetime values. When specifying the input format, enclose the letter T in single quotes to indicate that it is a literal character. Specify the time zone of the output datetime array using the `TimeZone` name-value pair argument.

```
t = datetime(DateStrings,'InputFormat','uuuu-MM-dd''T''HH:mmXXX','TimeZone','UTC')
```

```
t = 3x1 datetime array
   26-May-2014 18:30:00
   26-Aug-2014 17:30:00
   26-Sep-2014 13:30:00
```

The datetime values in `t` display in the default format.

Date and Time from Text in Foreign Language

Create a cell array of character vectors containing dates in French.

```
C = {'8 avril 2013','9 mai 2013';'10 juin 2014','11 juillet 2014'}
```

```
C = 2x2 cell array
    {'8 avril 2013'}    {'9 mai 2013'     }
    {'10 juin 2014'}    {'11 juillet 2014'}
```

Convert the character vectors in `C` to datetime values. If your computer is set to a locale that uses English, you must specify the `'Locale'` name-value pair argument to indicate that the strings are in French.

```
t = datetime(C,'InputFormat','d MMMM yyyy','Locale','fr_FR')
```

```
t = 2x2 datetime array
   08-Apr-2013   09-May-2013
   10-Jun-2014   11-Jul-2014
```

The datetime values in `t` display in the default format, and in the language MATLAB uses depending on your system locale.

Date and Time from Vectors

Create a `datetime` array from individual arrays of year, month, and day values.

Create sample numeric arrays of year values Y and day values D. In this case, the month value M is a scalar.

```
Y = [2014;2013;2012];
M = 01;
D = [31;30;31];
```

Create the `datetime` array.

```
t = datetime(Y,M,D)
```

```
t = 3x1 datetime array
   31-Jan-2014
   30-Jan-2013
   31-Jan-2012
```

Specify a custom display format for the output, using the `Format` name-value pair argument.

```
t = datetime(Y,M,D,'Format','eeee, MMMM d, y')
```

```
t = 3x1 datetime array
   Friday, January 31, 2014
   Wednesday, January 30, 2013
   Tuesday, January 31, 2012
```

Convert Excel Date Number to Datetime

Create a sample array of Excel® date numbers that represent a number of days since January 0, 1900.

```
X = [39558, 39600; 39700, 39800]
```

```
X = 2×2

       39558       39600
       39700       39800
```

Convert the values in X to datetime values.

```
t = datetime(X,'ConvertFrom','excel')
```

```
t = 2x2 datetime array
   20-Apr-2008 00:00:00   01-Jun-2008 00:00:00
   09-Sep-2008 00:00:00   18-Dec-2008 00:00:00
```

Tips

- For a list of `datetime` functions, see “Dates and Time”.
- For a list of core MATLAB functions that accept `datetime` arrays as input arguments, see “Core Functions Supporting Date and Time Arrays”.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- Always specify the input datetime format when creating a tall datetime array for a string array or character vectors in a cell array.
- If you specify `'Format'` as `'preserveinput'`, then MATLAB might need to evaluate the tall array to determine the format.

For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- When you create a `datetime` array from the text in `DateStrings`, specify the input format `infmt`.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`NaT` | `calendarDuration` | `cellstr` | `char` | `duration` | `string` | `timezones`

Topics

“Represent Dates and Times in MATLAB”
“Set Date and Time Display Format”
“Specify Time Zones”
“Generate Sequence of Dates and Time”

Introduced in R2014b

DatetimeRuler Properties

Control axis with datetime values

Description

`DatetimeRuler` properties control the appearance and behavior of an *x*-axis, *y*-axis, or *z*-axis that shows datetime values. Each individual axis has its own ruler object. By changing property values of the ruler, you can modify certain aspects of a specific axis.

Use dot notation to refer to a particular ruler and property. Access the ruler objects through the `XAxis`, `YAxis`, and `ZAxis` properties of the `Axes` object.

```
ax = gca;
co = ax.XAxis.Color;
ax.XAxis.Color = 'blue';
```

Properties

Appearance

`Color` — Color of axis line and labels

RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of the axis line and labels, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Note The `Color` property for the ruler and the associated `XColor`, `YColor`, or `ZColor` property for the parent axes always have the same value. Setting one also sets the other.

`LineWidth` — Width of axis line and tick marks

`0.5` (default) | positive value

Width of axis line and tick marks, specified as a positive value in point units. One point equals 1/72 inch.

Example: `ax.XAxis.LineWidth = 2;`

Note Setting the `LineWidth` property for the parent axes sets the `LineWidth` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

`Label` — Axis label

text object (default)

Axis label, which is a text object. To display text or change existing text, set the `String` property for the text object. Use other properties to change the text appearance, such as the font style or color.

```
ax = gca;
ax.XAxis.Label.String = 'X Axis';
ax.XAxis.Label.FontSize = 12;
```

For a full list of options, see Text.

Alternatively, add or modify the axis labels using the `xlabel`, `ylabel`, and `zlabel` functions.

Note The text object is not a child of the ruler object, so it cannot be returned by `findobj` and it does not use the default text property values.

`Visible` — Axis visibility

`'on'` (default) | `'off'`

Axis visibility, specified as one of these values:

- `'on'` — Display the axis.
- `'off'` — Hide the axis without deleting it. You still can access properties of an invisible axis using the ruler object.

Example: `ax.XAxis.Visible = 'off';`

Scale and Direction

Limits — Minimum and maximum axis limits

two-element vector of the form [min max]

Minimum and maximum axis limits, specified as a two-element vector of the form [min max], where min and max are datetime values. For example:

```
t = datetime(2014,6,28) + caldays(1:10);
y = rand(1,10);
plot(t,y);
ax = gca;
ax.XAxis.Limits = [t(2) t(8)];
```

Alternatively, set the limits using the xlim, ylim, and zlim functions.

If you assign a value to this property, then MATLAB sets the associated mode to 'manual'.

Note The Limits property for the ruler and the associated XLim, YLim, or ZLim property for the parent axes always have the same value. Setting one also sets the other.

LimitsMode — Selection mode for Limits property

'auto' (default) | 'manual'

Selection mode for the Limits property, specified as one of these values:

- 'auto' — Automatically select the axis limits based on the data plotted.
- 'manual' — Use axis limit values that you specify. To specify the axis limits, set the Limits property.

Note The LimitsMode property for the ruler and the associated XLimMode, YLimMode, or ZLimMode property for the parent axes always have the same value. Setting one also sets the other.

Scale — Scale of values along axis

'linear' (default)

Scale of values along axis, returned as 'linear'. DatetimeRuler objects do not support log scales.

`Direction` — Direction of increasing values

`'normal'` (default) | `'reverse'`

Direction of increasing values, specified as one of these values:

- `'normal'` — Values increase from left to right or bottom to top.
- `'reverse'` — Values increase from right to left or top to bottom.

Note The `Direction` property for the ruler and the associated `XDir`, `YDir`, or `ZDir` property for the parent axes always have the same value. Setting one also sets the other.

Example: `ax.XAxis.Direction = 'reverse';`

Tick Values and Labels

`TickValues` — Tick mark locations along the axis

`[ ]` (default) | vector of datetime values

Tick mark locations along the axis, specified as a vector of datetime values. For example:

```
t = datetime(2014,6,28) + caldays(1:10);
y = rand(1,10);
plot(t,y);
ax = gca;
ax.XAxis.TickValues = [t(1) t(3) t(5) t(10)];
```

If you assign a value to this property, then MATLAB sets the `TickValuesMode` property to `'manual'`.

Alternatively, use the `xticks`, `yticks`, and `zticks` functions.

Note The `TickValues` property for the ruler and the associated `XTick`, `YTick`, or `ZTick` property for the parent axes always have the same value. Setting one also sets the other.

`TickValuesMode` — Selection mode for `TickValues` property

`'auto'` (default) | `'manual'`

Selection mode for the `TickValues` property, specified as one of these values:

- `'auto'` — Automatically select the tick values based on the data plotted.
- `'manual'` — Use tick values that you specify. To specify the values, set the `TickValues` property.

Note The `TickValuesMode` property for the ruler and the associated `XTickMode`, `YTickMode`, or `ZTickMode` property for the parent axes always have the same value. Setting one also sets the other.

`TickLabels` — Tick mark labels

`''` (default) | cell array of character vectors | string array | categorical array

Tick mark labels, specified as a cell array of character vectors, string array, or categorical array. If you do not specify enough labels for all of the tick values, then the labels repeat. The labels support TeX and LaTeX markup. See the `TickLabelInterpreter` property for more information.

Example: `ax.XAxis.TickLabels = {'January','February','March','April','May'}';`

Example: `ax.YAxis.TickLabels = {'\pi','2\pi','3\pi'}'`

If you set this property, then MATLAB sets the `TickLabelsMode` property to `'manual'`.

Alternatively, specify the tick labels using the `xticklabels`, `yticklabels`, and `zticklabels` functions.

Note

- If you specify this property as a categorical array, MATLAB uses the values in the array, not the categories.
- The `TickLabels` property for the ruler and the associated `XTickLabel`, `YTickLabel`, or `ZTickLabel` property for the parent axes always have the same value. Setting one also sets the other.

Data Types: `char` | `string` | `categorical`

`TickLabelsMode` — Selection mode for `TickLabels` property

`'auto'` (default) | `'manual'`

Selection mode for the `TickLabels` property, specified as one of these values:

- `'auto'` — Automatically select the tick labels.
- `'manual'` — Use tick labels that you specify. To specify the labels, set the `TickLabels` property.

Note The `TickLabelsMode` property for the ruler and the associated `XTickLabelMode`, `YTickLabelMode`, or `ZTickLabelMode` property for the parent axes always have the same value. Setting one also sets the other.

`TickLabelInterpreter` — Interpretation of tick label characters

`'tex'` (default) | `'latex'` | `'none'`

Interpretation of tick label characters, specified as one of these values:

- `'tex'` — Interpret characters using a subset of TeX markup.
- `'latex'` — Interpret characters using LaTeX markup.
- `'none'` — Display literal characters.

Note Setting the `TickLabelInterpreter` property for the parent axes sets the `TickLabelInterpreter` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.XAxis.TickLabelInterpreter = 'latex';`

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the text type and color, and include special characters in the text.

This table lists the supported modifiers with the `TickLabelInterpreter` property set to `'tex'`. Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they only modify the next character or the text within the curly braces `{}`.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (rarely available)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{specifier}`	Set `specifier` as the name of a font family to change the font style. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{specifier}`	Set `specifier` as a scalar numeric value to change the font size.	`'\fontsize{15} text'`
`\color{specifier}`	Set `specifer` as one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Set `specifier` as a three-element RGB triplet to change the font color.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\beta	β	\psi	ψ	\clubsuit	♣
\gamma	γ	\omega	ω	\diamondsuit	♦
\delta	δ	\Gamma	Γ	\heartsuit	♥
\epsilon	ϵ	\Delta	Δ	\spadesuit	♠
\zeta	ζ	\Theta	Θ	\leftrightarrow	↔
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	µ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	…
\perp	⊥	\surd	√	\prime	′
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `TickLabelInterpreter` property to `'latex'`. Use dollar symbols around the labels, for example, use `'$\int_1^{20} x^2 dx$'` for inline mode or `'$$\int_1^{20} x^2 dx$$'` for display mode.

The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup within the text. The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters. For multiline text, the maximum size of the text reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project website at https://www.latex-project.org/.

`TickLabelFormat` — Tick label format

character vector | string

Tick label format, specified as a character vector or string. The default format is based on the data.

Example: `ax.XAxis.TickLabelFormat = 'yyyy-MM-dd';` displays a date and time such as `2014-04-19`.

Example: `ax.XAxis.TickLabelFormat = 'eeee, MMMM d, yyyy HH:mm:ss';` displays a date and time such as `Saturday, April 19, 2014 21:41:06`.

Example: `ax.XAxis.TickLabelFormat = 'MMMM d, yyyy HH:mm:ss Z';` displays a date and time such as `April 19, 2014 21:41:06 -0400`.

The following tables show the letter identifiers that you can use to construct the format. To separate the fields, use nonletter characters such as a hyphen, space, colon, or any non-ASCII character. The identifiers correspond to the Unicode Locale Data Markup Language (LDML) standard for dates.

Date and Time Formats

Use these identifiers to specify the display formats of the date and time fields.

Letter Identifier	Description	Display
`G`	Era	`CE`
`y`	Year, with no leading zeros. See the Note that follows this table.	`2014`
`yy`	Year, using last two digits. See the Note that follows this table.	`14`
`yyy`, `yyyy` ...	Year, using at least the number of digits specified by the number of instances of `'y'`	For the year 2014, `'yyy'` displays `2014`, while `'yyyyy'` displays `02014`.
`u`, `uu`, ...	ISO year. A single number designating the year. An ISO year value assigns positive values to CE years and negative values to BCE years, with 1 BCE being year 0.	`2014`
`Q`	Quarter, using one digit	`2`
`QQ`	Quarter, using two digits	`02`
`QQQ`	Quarter, abbreviated	`Q2`
`QQQQ`	Quarter, full name	`2nd quarter`
`M`	Month, numerical using one or two digits	`4`
`MM`	Month, numerical using two digits	`04`
`MMM`	Month, abbreviated name	`Apr`
`MMMM`	Month, full name	`April`
`MMMMM`	Month, capitalized first letter	`A`
`W`	Week of the month	`1`

Letter Identifier	Description	Display
d	Day of the month, using one or two digits	5
dd	Day of the month using two digits	05
D	Day of the year, using one, two or three digits	95
DD	Day of the year using two digits	95
DDD	Day of the year using three digits	095
e	Day of the week, numerical using one or two digits.	7, where Sunday is the first day of the week.
ee	Day of the week, numerical using two digits	07
eee	Day, abbreviated name	Sat
eeee	Day, full name	Saturday
eeeee	Day, capitalized first letter	S
a	Day period (AM or PM)	PM
h	Hour, 12-hour clock notation using one or two digits	9
hh	Hour, 12-hour clock notation using two digits	09
H	Hour, 24-hour clock notation using one or two digits	21
HH	Hour, 24-hour clock notation using two digits	21
m	Minute, using one or two digits	41
mm	Minute, using two digits	41
s	Second, using one or two digits	6
ss	Second, using two digits	06

Letter Identifier	Description	Display
S, SS, ..., SSSSSSSSS	Fractional second, using the number of digits specified by the number of instances of `'S'` (up to 9 digits).	`'SSS'` truncates 6.12345 seconds to 123.

Some tips and considerations:

- If you read a two-digit year number and specify the format as `y` or `yy`, then the pivot year determines the century to which the year belongs.
- Use one or more `u` characters instead of `y` characters to represent the year when working with year numbers near zero.
- Datetime values later than 144683 years CE or before 140743 BCE display only the year numbers, regardless of the specified format value.

Time Zone Offset Formats

Use these identifiers to specify the display format of the time zone offset. A time zone offset is the amount of time that a specific date and time is offset from UTC. This is different from a time zone, which comprises rules that determine the offsets for specific times of the year. Include a time zone offset identifier when you want to ensure that the time components are displayed unambiguously.

Letter Identifier	Description	Display
`z`	Abbreviated name of the time zone offset. If this value is not available, then the time zone offset uses the short UTC format, such as `UTC-4`.	`EDT`
`Z`	ISO 8601 basic format with hours, minutes, and optional seconds fields.	`-0400`
`ZZZZ`	Long UTC format.	`UTC-04:00`

Letter Identifier	Description	Display
ZZZZZ	ISO 8601 extended format with hours, minutes, and optional seconds fields. A time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-04:00
x or X	ISO 8601 basic format with hours field and optional minutes field. If you specify X, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-04
xx or XX	ISO 8601 basic format with hours and minutes fields. If you specify XX, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-0400
xxx or XXX	ISO 8601 extended format with hours and minutes fields. If you specify XXX, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-04:00
xxxx or XXXX	ISO 8601 basic format with hours, minutes, and optional seconds fields. If you specify XXXX, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-0400
xxxxx or XXXXX	ISO 8601 extended format with hours, minutes, and optional seconds fields. If you specify XXXXX, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-04:00

`TickLabelFormatMode` — Selection mode for `TickLabelFormat` property
`'auto'` (default) | `'manual'`

Selection mode for the `TickLabelFormat` property, specified as one of these values:

- `'auto'` — Automatically select the tick label format.
- `'manual'` — Use a tick label format that you specify. To specify the format, set the `TickLabelFormat` property.

`TickLabelRotation` — Rotation of tick labels
scalar value in degrees

Rotation of tick labels, specified as a scalar value in degrees. Positive values give counterclockwise rotation. Negative values give clockwise rotation.

Alternatively, you can rotate the tick labels using the `xtickangle`, `ytickangle`, and `ztickangle` functions.

Note The `TickLabelRotation` property for the ruler and the associated `XTickLabelRotation`, `YTickLabelRotation`, or `ZTickLabelRotation` property for the parent axes always have the same value. Setting one also sets the other.

Example: `ax.XAxis.TickLabelRotation = 45;`

Example: `ax.YAxis.TickLabelRotation = -45;`

`TickLabelRotationMode` — Selection mode for `TickLabelRotation` property
`'auto'` (default) | `'manual'`

Selection mode for the `TickLabelRotation` property, specified as one of these values:

- `'auto'` — Automatically select the tick label rotation.
- `'manual'` — Use a tick label rotation that you specify. To specify the rotation, set the `TickLabelRotation` property.

`TickDirection` — Tick mark direction
`'in'` (default) | `'out'` | `'both'`

Tick mark direction, specified as one of these values:

- `'in'` — Direct the tick marks inward from the axis lines. This is the default for 2-D views.
- `'out'` — Direct the tick marks outward from the axis lines. This is the default for 3-D views.

- `'both'` — Center the tick marks over the axis lines.

If you assign a value to this property, then MATLAB sets the `TickDirectionMode` property to `'manual'`.

Note Setting the `TickDir` property for the parent axes sets the `TickDirection` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.YAxis.TickDirection = 'out';`

`TickDirectionMode` — Selection mode for `TickDirection` property
`'auto'` (default) | `'manual'`

Selection mode for the `TickDirection` property, specified as one of these values:

- `'auto'` — Automatically select the tick direction.
- `'manual'` — Use a tick direction that you specify. To specify the tick direction, set the `TickDirection` property.

`TickLength` — Tick mark length
two-element vector

Tick mark length, specified as a two-element vector of the form `[2Dlength 3Dlength]`. The first element is the tick mark length in 2-D views. The second element is the tick mark length in 3-D views. Specify the values in units normalized relative to the longest axes dimension.

Note Setting the `TickLength` property for the parent axes sets the `TickLength` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.YAxis.TickLength = [0.02 0.035];`

`MinorTick` — Minor tick mark display
`'off'` (default) | `'on'`

Minor tick mark display, specified as one of these values:

- `'off'` — Do not display minor tick marks. This is the default value for an axis with a linear scale.
- `'on'` — Display minor tick marks between the major tick marks on the axis. This is the default value for an axis with a log scale. The space between the major tick marks determines the number of minor tick marks. If the `MinorTickValues` property is set to empty `[]`, then no minor tick marks appear. Specify the tick mark locations by setting the `MinorTickValues` property.

Note The `MinorTick` property for the ruler and the associated `XMinorTick`, `YMinorTick`, or `ZMinorTick` property for the parent axes always have the same value. Setting one also sets the other.

Example: `ax.XAxis.MinorTick = 'on';`

`MinorTickValues` — Minor tick mark locations

`[]` (default) | vector of increasing values

Minor tick mark locations, specified as a vector of increasing datetime values.

If you assign values to this property, then MATLAB sets the `MinorTickValuesMode` property to `'manual'`.

`MinorTickValuesMode` — Selection mode for `MinorTickValues` property

`'auto'` (default) | `'manual'`

Selection mode for the `MinorTickValues` property, specified as one of these values:

- `'auto'` — Use automatically calculated minor tick values.
- `'manual'` — Use minor tick values that you specify. To specify the values, set the `MinorTickValues` property.

Font

`FontName` — Font name

system supported font name | `'FixedWidth'`

Font name, specified as a system supported font name or `'FixedWidth'`. The default font depends on the specific system and locale. To use a fixed-width font that renders

well, specify `'FixedWidth'`. The actual fixed-width font used depends on the `FixedWidthFontName` property of the root object.

Note Setting the `FontName` property for the parent axes sets the `FontName` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.XAxis.FontName = 'Cambria';`

`FontSize` — Font size
scalar numeric value

Font size, specified as a scalar numeric value. The default font size depends on the specific operating system and locale.

Note Setting the `FontSize` property for the parent axes sets the `FontSize` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.XAxis.FontSize = 12;`

`FontWeight` — Thickness of text characters
`'normal'` (default) | `'bold'`

Thickness of the text characters, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font.
- `'bold'` — Thicker character outlines than normal.

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold font weight. Therefore, specifying a bold font weight still can result in the normal font weight.

Note Setting the `FontWeight` property for the parent axes sets the `FontWeight` property for the ruler to the same value. However, setting the ruler property does not set

the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.XAxis.FontWeight = 'bold';`

`FontAngle` — Text character slant
`'normal'` (default) | `'italic'`

Character slant, specified as `'normal'` or `'italic'`. Not all fonts have both font styles. Therefore, the italic font can look the same as the normal font.

Note Setting the `FontAngle` property for the parent axes sets the `FontAngle` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.XAxis.FontAngle = 'italic';`

`FontSmoothing` — Text antialiasing
`'on'` (default) | `'off'`

Text smoothing, specified as one of these values:

- `'on'` — Enable text antialiasing to reduce the jagged appearance of text characters and make the text easier to read. In certain cases, smoothed text blends against the background color and can make the text appear blurry.
- `'off'` — Disable text antialiasing. Use this setting if the text seems blurry.

Note Setting the `FontSmoothing` property for the parent axes sets the `FontSmoothing` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.XAxis.FontSmoothing = 'off';`

Parent/Child

`Parent` — Ruler parent

`Axes` object

Ruler parent, specified as an `Axes` object.

Note Ruler objects are not listed in the `Children` property of the parent `Axes` object.

`Children` — Ruler children

empty `GraphicsPlaceholder` array

The ruler has no children. You cannot set this property.

See Also

Axes

Introduced in R2016b

dataTipTextRow

Add row to data tips

Description

Use the `dataTipTextRow` function to create a new data tip row with a particular label, value source, and value format. After you create the new data tip row, you must attach it to the `DataTipTemplate` property of the plotted object.

Creation

Syntax

```
r = dataTipTextRow(label,value)
r = dataTipTextRow(label,value,format)
```

Description

`r = dataTipTextRow(label,value)` creates a new data tip row that uses the specified label and value source.

`r = dataTipTextRow(label,value,format)` additionally specifies the format for the displayed values.

Properties

`Label` — Label
character vector | string scalar

Label, specified as a character vector or string scalar.

Example: `'My Label'`

`Value` — Value source

character vector | string scalar | vector | function handle

Value source, specified as a character vector or string scalar containing the name of a data property (such as `'XData'`), the name of a workspace or table variable, a vector, or a function handle. The number of source values must match the number of data points.

Example: `'XData'` specifies the value source as the `XData` property of the plotted object.

Example: `[1 2 3 4 5]` specifies the value source as a vector.

Example: `@(x)sin(x)` specifies a function handle where `x` is the `XData` values of the object.

Example: `@(x,y)sin(y)` specifies a function handle where `y` is the `YData` values of the object.

`Format` — Value format

`'auto'` (default) | character vector | string scalar

Value format, specified as `'auto'` or a character vector or string scalar with a numeric, datetime, or duration format. The type of format that you specify must match the type of values in the `Value` property.

Example: `r = dataTipTextRow(label,value,'usd')` specifies a predefined numeric format for U.S. dollars.

Example: `r = dataTipTextRow(label,value,'%#4.4g')` specifies a numeric format.

Example: `r = dataTipTextRow(label,value,'yyyy-MM-dd')` specifies a datetime format.

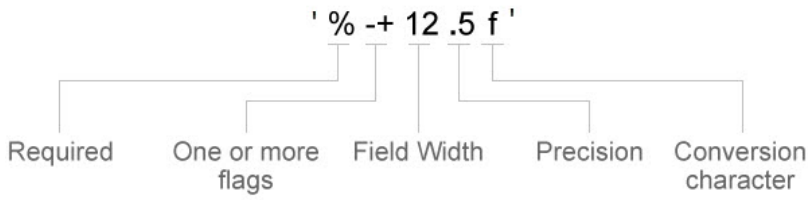
Example: `r = dataTipTextRow(label,value,'hh:mm:ss')` specifies a duration format.

Numeric Formats

You can specify one of the predefined formats listed in this table. Alternatively, you can specify a custom numeric format.

Predefined Format	Description
`'usd'`	U.S. dollars. This option is equivalent using `'$%,.2f'`. If the labels use scientific notation, this option sets the exponent value to 0.
`'eur'`	Euro. This option is equivalent to using `'\x20AC%,.2f'` with an exponent value of 0.
`'gbp'`	British pound. This option is equivalent to using `'\x00A3%,.2f'` with an exponent value of 0.
`'jpy'`	Japanese yen. This option is equivalent to using `'\x00A5%,d'` with an exponent value of 0.
`'degrees'`	Display degree symbol after values. This option is equivalent to using `'%g\x00B0'` with the default exponent value.
`'percentage'`	Display percent sign after values. This option is equivalent to using `'%g%%'` with the default exponent value.
`'auto'`	Display value with four significant figures.

If none of the formats mentioned in the table gives the format you want, then create a custom character vector or string with identifiers.



Identifiers are optional, except the percent sign and conversion character. Construct the format in this order:

- One or more flags — Options such as adding a plus sign before positive values. For a full list of options, see the table of optional flags.

- Field width — Minimum number of characters to print in the tick label. Specify the field width as an integer value. If the number of significant digits in the tick value is smaller than the field width, then the label is padded with spaces.
- Precision — Number of digits to the right of the decimal point or the number of significant digits, depending on the conversion character. Specify the precision as an integer value.
- Conversion character — Value type. For a full list of options, see the table of conversion characters. If you specify a conversion that does not fit the data, MATLAB overrides the specified conversion and uses `%e`.

Also, you can specify literal text at the beginning or end of the format. To print a single quotation mark, use `''`. To print a percent character, use `%%`.

- `'$%.2f'` — Display a dollar sign before each value and use fixed-point notation with two decimal values.
- `'%.3f Million'` — Display `Million` after each value and use fixed-point notation with three decimal values.

Optional Flag	Description	Example of Numeric Format
`+`	Print the sign character (`+`) for positive values, such as `'+100'`.	`'%+4.4g'`
`0`	Pad the field width with leading zeros instead of spaces, such as `'0100'`.	`'%04.4g'`
`-`	Left-justify, which pads the end of the value with spaces instead of the beginning. For example, if the field width is `4`, then this flag formats the label as `'100 '` instead of `' 100'`.	`'%-4.4g'`

Optional Flag	Description	Example of Numeric Format
`#`	For the `%f`, `%e`, and `%g` conversion characters, print the decimal point even when the precision is 0, such as `'100.'`. For `%g`, do not remove trailing zeros.	`'%#4.4g'`

Conversion Character	Description	Example
`d` or `i`	Signed integer with base 10. The precision value indicates the number of significant digits.	`'%.4d'` displays π as `0003`.
`f`	Fixed-point notation. The precision value indicates the number of decimal places.	`'%.4f'` displays π as `3.1416`.
`e`	Exponential notation. The precision value indicates the number of decimal places.	`'%.4e'` displays π as 3.1416×10^0.
`g`	The more compact version of `e` or `f`, with no trailing zeros. The precision value indicates the maximum number of decimal places.	`'%.4g'` displays π as `3.1416`.

Datetime Formats

This table shows several common formats and includes examples of the formatted input for the date, Saturday, April 19, 2014 at 9:41:06.12345 PM in New York City.

Format	Example
`'yyyy-MM-dd'`	`2014-04-19`
`'dd/MM/yyyy'`	`19/04/2014`
`'dd.MM.yyyy'`	`19.04.2014`

Format	Example
'yyyy 年 MM 月 dd 日'	2014 年 04 月 19 日 (Characters for Japanese locale, ja_JP)
'MMMM d, yyyy'	April 19, 2014
'eeee, MMMM d, yyyy h:mm a'	Saturday, April 19, 2014 9:41 PM
'MMMM d, yyyy HH:mm:ss Z'	April 19, 2014 21:41:06 -0400
'yyyy-MM-dd''T''HH:mmXXX'	2014-04-19T21:41-04:00
'yyyy-MM-dd HH:mm:ss.SSS'	2014-04-19 21:41:06.123

The following tables show the letter identifiers that you can use to construct datetime formats. To separate the fields, use nonletter characters such as a hyphen, space, colon, or any non-ASCII character. The identifiers correspond to the Unicode Locale Data Markup Language (LDML) standard for dates.

Use the identifiers in this table to specify the display formats of the date and time fields.

Letter Identifier	Description	Display
G	Era	CE
y	Year, with no leading zeros. See the Note that follows this table.	2014
yy	Year, using last two digits. See the Note that follows this table.	14
yyy, yyyy ...	Year, using at least the number of digits specified by the number of instances of 'y'	For the year 2014, 'yyy' displays 2014, while 'yyyyy' displays 02014.
u, uu, ...	ISO year. A single number designating the year. An ISO year value assigns positive values to CE years and negative values to BCE years, with 1 BCE being year 0.	2014
Q	Quarter, using one digit	2
QQ	Quarter, using two digits	02
QQQ	Quarter, abbreviated	Q2

Letter Identifier	Description	Display
QQQQ	Quarter, full name	2nd quarter
M	Month, numerical using one or two digits	4
MM	Month, numerical using two digits	04
MMM	Month, abbreviated name	Apr
MMMM	Month, full name	April
MMMMM	Month, capitalized first letter	A
W	Week of the month	1
d	Day of the month, using one or two digits	5
dd	Day of the month using two digits	05
D	Day of the year, using one, two or three digits	95
DD	Day of the year using two digits	95
DDD	Day of the year using three digits	095
e	Day of the week, numerical using one or two digits.	7, where Sunday is the first day of the week.
ee	Day of the week, numerical using two digits	07
eee	Day, abbreviated name	Sat
eeee	Day, full name	Saturday
eeeee	Day, capitalized first letter	S
a	Day period (AM or PM)	PM
h	Hour, 12-hour clock notation using one or two digits	9
hh	Hour, 12-hour clock notation using two digits	09
H	Hour, 24-hour clock notation using one or two digits	21

Letter Identifier	Description	Display
HH	Hour, 24-hour clock notation using two digits	21
m	Minute, using one or two digits	41
mm	Minute, using two digits	41
s	Second, using one or two digits	6
ss	Second, using two digits	06
S, SS, ..., SSSSSSSSS	Fractional second, using the number of digits specified by the number of instances of 'S' (up to 9 digits).	'SSS' truncates 6.12345 seconds to 123.

Some tips and considerations:

- If you read a two-digit year number and specify the format as y or yy, then the pivot year determines the century to which the year belongs.
- Use one or more u characters instead of y characters to represent the year when working with year numbers near zero.
- Datetime values later than 144683 years CE or before 140743 BCE display only the year numbers, regardless of the specified format value.

Use the identifiers in this table to specify the display format of the time zone offset. A time zone offset is the amount of time that a specific date and time is offset from UTC. This is different from a time zone, which comprises rules that determine the offsets for specific times of the year. Include a time zone offset identifier when you want to ensure that the time components are displayed unambiguously.

Letter Identifier	Description	Display
z	Abbreviated name of the time zone offset. If this value is not available, then the time zone offset uses the short UTC format, such as UTC-4.	EDT

Letter Identifier	Description	Display
Z	ISO 8601 basic format with hours, minutes, and optional seconds fields.	-0400
ZZZZ	Long UTC format.	UTC-04:00
ZZZZZ	ISO 8601 extended format with hours, minutes, and optional seconds fields. A time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-04:00
x or X	ISO 8601 basic format with hours field and optional minutes field. If you specify X, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-04
xx or XX	ISO 8601 basic format with hours and minutes fields. If you specify XX, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-0400
xxx or XXX	ISO 8601 extended format with hours and minutes fields. If you specify XXX, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-04:00
xxxx or XXXX	ISO 8601 basic format with hours, minutes, and optional seconds fields. If you specify XXXX, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-0400

Letter Identifier	Description	Display
xxxxx or XXXXX	ISO 8601 extended format with hours, minutes, and optional seconds fields. If you specify XXXXX, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	`-04:00`

Duration Formats

Duration formats can be either a single number with time units (such as `'0.5 yrs'`) or a digital timer (such as `'hh:mm:ss'` for hours, minutes, and seconds).

For numbers with time units, specify one of the following:

- `'y'` — Fixed-length years, where one year equals 365.2425 days
- `'d'` — Fixed-length days, where one day equals 24 hours
- `'h'` — Hours
- `'m'` — Minutes
- `'s'` — Seconds

For digital timer formats, specify one of the following:

- `'dd:hh:mm:ss'`
- `'hh:mm:ss'`
- `'mm:ss'`
- `'hh:mm'`
- Any of the first three formats, with up to nine `S` characters to indicate fractional second digits, such as `'hh:mm:ss.SSSS'`

Examples

Add Row to Data Tips

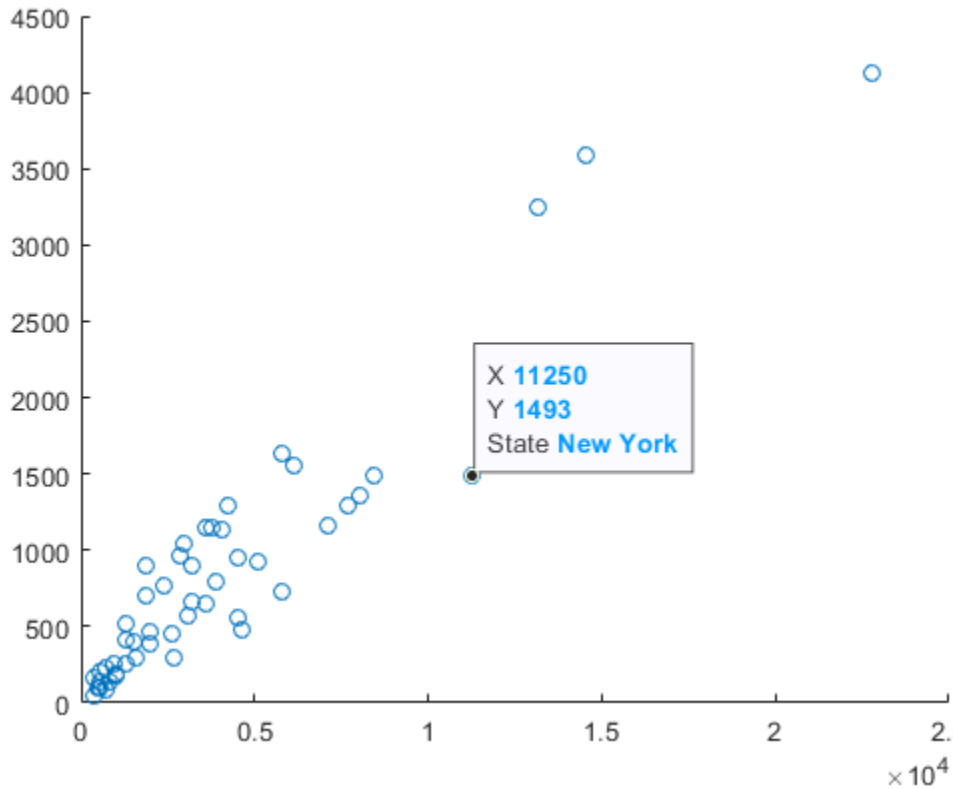
Load sample accident data and create a scatter plot. Then add a third row to the default data tips. For the label, use `State`. For the value source, use the `statelabel` workspace

variable. After you create the new row, add it to the `DataTipTemplate` property for the `Scatter` object.

```
load('accidents.mat','hwydata','statelabel')
s = scatter(hwydata(:,5),hwydata(:,4));
row = dataTipTextRow('State',statelabel);
s.DataTipTemplate.DataTipRows(end+1) = row;
```

Create a data tip interactively or by using the `datatip` function.

```
dt = datatip(s,11250,1493);
```



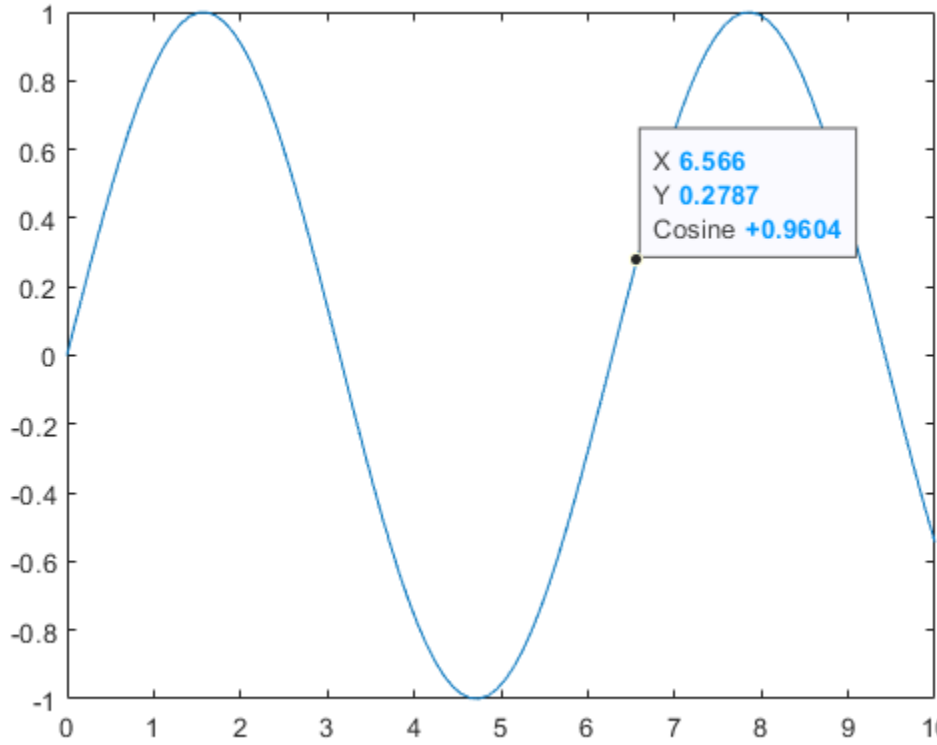
Change Format for Displayed Values

Create a line plot of sine values. Add a third row to the default data tips to show the cosine values. Specify the numeric format for the displayed cosine values. In this case, use `'%+4.4g'` to include the sign character (+) for positive values. After you create the new row, add it to the `DataTipTemplate` property for the `Line` object.

```
x = linspace(0,10);
y1 = sin(x);
y2 = cos(x);
p = plot(x,y1);
row = dataTipTextRow('Cosine',y2,'%+4.4g');
p.DataTipTemplate.DataTipRows(end+1) = row;
```

Create a data tip interactively or by using the `datatip` function.

```
dt = datatip(p,6.566,0.2787);
```



See Also

DataTipTemplate Properties | `datatip`

Topics

"Create Custom Data Tips"

Introduced in R2019a

DataTipTemplate Properties

Data tip content and appearance

Description

`DataTipTemplate` properties control the content and appearance of data tips. By changing property values, you can modify certain aspects of data tips.

```
p = plot(1:10);
dtt = p.DataTipTemplate
```

Properties

Content

`DataTipRows` — Row content
array of `DataTipTextRow` objects

Row content, specified as an array of `DataTipTextRow` objects. The default data tips have two or three rows, depending on the plotted data.

You can modify the existing rows by setting properties of the `DataTipTextRow` objects. Access a particular row by indexing into the array of objects.

Example: `dtt.DataTipRows(1).Label = 'My label'` changes the label for the first row.

Example: `dtt.DataTipRows(1).Value = 'YData'` changes the value source for the first row to use the `YData` property of the plotted object.

Example: `dtt.DataTipRows(1).Format = 'usd'` changes the value format for the first row.

Alternatively, you can add new rows to the data tip using the `dataTipTextRow` function.

Example: `dtt.DataTipRows(end+1) = dataTipTextRow(label,value)` adds a new row to the data tip with the specified label and value source.

Font

`FontName` — Font name

supported font name | `'FixedWidth'`

Font name, specified as a supported font name or `'FixedWidth'`. To display and print text properly, you must choose a font that your system supports. The default font depends on your operating system and locale.

To use a fixed-width font that looks good in any locale, use `'FixedWidth'`. The fixed-width font relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`FontSize` — Font size

10 (default) | scalar value greater than zero

Font size, specified as a scalar value greater than zero in point units. One point equals `1/72` inch.

`FontAngle` — Character slant

`'normal'` (default) | `'italic'`

Character slant, specified as `'normal'` or `'italic'`.

Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.

`Interpreter` — Interpretation of text characters

`'tex'` (default) | `'latex'` | `'none'`

Interpretation of text characters, specified as one of these values:

- `'tex'` — Interpret characters using a subset of TeX markup.
- `'latex'` — Interpret characters using LaTeX markup.
- `'none'` — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text.

Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they modify only the next character or the characters within the curly braces. When you set the `Interpreter` property to `'tex'`, the supported modifiers are as follows.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{`*`specifier`*`}`	Font name — Replace *`specifier`* with the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{`*`specifier`*`}`	Font size —Replace *`specifier`* with a numeric scalar value in point units.	`'\fontsize{15} text'`
`\color{`*`specifier`*`}`	Font color — Replace *`specifier`* with one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Replace *`specifier`* with a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\alpha	α	\upsilon	υ	\sim	∼
\angle	∠	\phi	ϕ	\leq	≤
\ast	*	\chi	χ	\infty	∞
\beta	β	\psi	ψ	\clubsuit	♣
\gamma	γ	\omega	ω	\diamondsuit	♦
\delta	δ	\Gamma	Γ	\heartsuit	♥
\epsilon	ϵ	\Delta	Δ	\spadesuit	♠
\zeta	ζ	\Theta	Θ	\leftrightarrow	↔
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	µ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. Use dollar symbols around the text, for example, use `'$\int_1^{20} x^2 dx$'` for inline mode or `'$$ \int_1^{20} x^2 dx$$'` for display mode.

The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup.

The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters. For multiline text, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project website at https://www.latex-project.org/.

Tips

Control the behavior of data tips by modifying properties of the `datatip` object.

See Also

`dataTipTextRow`

Introduced in R2019a

datevec

Convert date and time to vector of components

Syntax

```
DateVector = datevec(t)
DateVector = datevec(DateNumber)

DateVector = datevec(DateString)
DateVector = datevec(DateString,formatIn)
DateVector = datevec(DateString,PivotYear)
DateVector = datevec(DateString,formatIn,PivotYear)

[Y,M,D,H,MN,S] = datevec( ___ )
```

Description

`DateVector = datevec(t)` converts the datetime or duration value `t` to a *date vector*—that is, a numeric vector whose six elements represent the year, month, day, hour, minute, and second components of `t`.

If `t` is a datetime or duration array having `m` elements, then `datevec` returns an `m`-by-6 matrix where each row corresponds to a value in `t`.

- If `t` is a datetime array, then its values represent points in time. Each row of `DateVector` represents a specific date and time.
- If `t` is a duration array, then its values represent elapsed time. Each row of `DateVector` represents a length of time measured in fixed-length time units. For this reason, the values in the second column of `DateVector` (representing number of months) are always zeros. For more information, see the “Limitations” on page 1-3070.

Note While you can represent dates and times as date vectors, it is recommended that you use `datetime` values to represent points in time, and `duration` or `calendarDuration` values to represent elapsed times.

`DateVector = datevec(DateNumber)` converts one or more date numbers to date vectors. The `datevec` function returns an `m`-by-6 matrix containing `m` date vectors, where `m` is the total number of date numbers in `DateNumber`.

`DateVector = datevec(DateString)` converts text representing dates and times to date vectors. If the format used in the text is known, specify the format as `formatIn`. Syntaxes without `formatIn` are significantly slower than syntaxes that include it.

`DateVector = datevec(DateString,formatIn)` uses `formatIn` to interpret the dates and times represented by `DateString`.

`DateVector = datevec(DateString,PivotYear)` uses `PivotYear` to interpret text that specifies the year as two characters. If the format used in the text is known, specify the format as `formatIn`. Syntaxes without `formatIn` are significantly slower than syntaxes that include it.

`DateVector = datevec(DateString,formatIn,PivotYear)` uses `formatIn` to interpret the dates and times represented by `DateString`, and `PivotYear` to interpret text that specifies the year as two characters. You can specify `formatIn` and `PivotYear` in either order.

`[Y,M,D,H,MN,S] = datevec( ___ )` returns the components of the date vector as individual variables `Y`, `M`, `D`, `H`, `MN`, and `S` (year, month, day, hour, minutes, and seconds). The `datevec` function returns milliseconds as a fractional part of the seconds (`S`) output.

Examples

Convert datetime Array to Date Vectors

```
format short g
```

```
t = [datetime('now');datetime('tomorrow')]
```

```
t = 2x1 datetime array
   26-Aug-2019 17:40:17
   27-Aug-2019 00:00:00
```

```
DateVector = datevec(t)
```

```
DateVector = 2×6
```

```
        2019           8          26          17          40      17.284
        2019           8          27           0           0           0
```

Convert Date Number to Date Vector

```
format short g

n = 733779.651;
datevec(n)
```

```
ans = 1×6

        2009           1           6          15          37        26.4
```

Convert Text Representing Date to Date Vector

```
DateString = '28.03.2005';
formatIn = 'dd.mm.yyyy';
datevec(DateString,formatIn)
```

```
ans = 1×6

       2005           3          28           0           0           0
```

`datevec` returns a date vector for text representing a date with the format `'dd.mm.yyyy'`.

Convert Multiple Date Strings to Date Vectors

Pass multiple dates as character vectors in a cell array. All input dates must use the same format.

```
DateString = {'09/16/2007';'05/14/1996';'11/29/2010'};
formatIn = 'mm/dd/yyyy';
datevec(DateString,formatIn)
```

```
ans = 3×6

        2007           9          16           0           0           0
        1996           5          14           0           0           0
        2010          11          29           0           0           0
```

Convert Date with Milliseconds to Date Vector

```
datevec('11:21:02.647','HH:MM:SS.FFF')
```

```
ans = 1×6
10^3 ×

    2.0190    0.0010    0.0010    0.0110    0.0210    0.0026
```

In the output date vector, milliseconds are a fractional part of the seconds field. The text `'11:21:02.647'` does not contain enough information to convert to a full date vector. The days default to 1, months default to January, and years default to the current year.

Convert Date String to Date Vector Using Pivot Year

Convert text representing a date to a date vector using the default pivot year.

```
DateString = '12-jun-17';
formatIn = 'dd-mmm-yy';
DateVector = datevec(DateString,formatIn)
```

```
DateVector = 1×6

        2017           6          12           0           0           0
```

Convert the same date to a date vector using 1800 as the pivot year.

```
DateVector = datevec(DateString,formatIn,1800)
```

```
DateVector = 1×6

        1817           6          12           0           0           0
```

Assign Elements of Returned Date Vector

Convert text representing a date to a date vector and return the components of the date vector.

```
[y, m, d, h, mn, s] = datevec('01.02.12','dd.mm.yy')
```

```
y = 2012

m = 2

d = 1

h = 0

mn = 0

s = 0
```

Input Arguments

`t` — Dates and times
array of datetime or duration values

Dates and times, specified as an array of datetime or duration values.

Data Types: `datetime` | `duration` | `calendarDuration`

`DateNumber` — Serial date number
array of positive double-precision numbers

Serial date number, specified as an array of positive double-precision numbers.

Example: `731878`

Data Types: `double`

`DateString` — Text representing dates and times
character array | cell vector of character vectors | string vector

Text representing dates and times, specified as a character array where each row contains text representing one point in time, as a cell vector of character vectors, or as a string vector. All rows of a character array, or all elements of a cell vector or string vector, must have the same format.

Example: `'24-Oct-2003 12:45:07'`

Example: `['19-Sep-2013';'20-Sep-2013';'21-Sep-2013']`

Example: `{'15-Oct-2010' '20-Nov-2012'}`

If the format used in the text is known, you should also specify `formatIn`. If you do not specify `formatIn`, then `DateString` must be in one of the following formats.

Format of Text Representing Dates and Times	Example
`'dd-mmm-yyyy HH:MM:SS'`	`01-Mar-2000 15:45:17`
`'dd-mmm-yyyy'`	`01-Mar-2000`
`'mm/dd/yyyy'`	`03/01/2000`
`'mm/dd/yy'`	`03/01/00`
`'mm/dd'`	`03/01`
`'mmm.dd,yyyy HH:MM:SS'`	`Mar.01,2000 15:45:17`
`'mmm.dd,yyyy'`	`Mar.01,2000`
`'yyyy-mm-dd HH:MM:SS'`	`2000-03-01 15:45:17`
`'yyyy-mm-dd'`	`2000-03-01`
`'yyyy/mm/dd'`	`2000/03/01`
`'HH:MM:SS'`	`15:45:17`
`'HH:MM:SS PM'`	`3:45:17 PM`
`'HH:MM'`	`15:45`
`'HH:MM PM'`	`3:45 PM`

Note The symbolic identifiers describing date and time formats are different from those that describe the display formats of `datetime` arrays.

Certain formats might not contain enough information to convert text representations of dates and times. In those cases, hours, minutes, and seconds default to 0, days default to 1, months default to January, and years default to the current year. `datevec` and `datenum` consider two-character years (e.g., `'79'`) to fall within the 100-year range centered around the current year.

When you do not specify `formatIn`, note the following:

- For the formats that specify the month as two digits (`mm`), the month value must not be greater than 12.
- However, for the format `'mm/dd/yy'`, if the first entry in the text is greater than 12 and the second entry is less than or equal to 12, then `datevec` considers the text to be in `'yy/mm/dd'` format.

`formatIn` — Format of the input text representing dates and times
character vector | string scalar

Format of the input text representing dates and times, specified as a character vector or string scalar of symbolic identifiers.

Example: `'dddd, mmm dd, yyyy'`

The following table shows symbolic identifiers that you can use to construct the `formatIn` character vector. You can include characters such as a hyphen, space, or colon to separate the fields.

Note The symbolic identifiers describing date and time formats are different from the identifiers that describe the display formats of `datetime` arrays.

Symbolic Identifier	Description	Example
`yyyy`	Year in full	`1990`, `2002`
`yy`	Year in two digits	`90`, `02`

Symbolic Identifier	Description	Example
QQ	Quarter year using letter Q and one digit	Q1
mmmm	Month using full name	March, December
mmm	Month using first three letters	Mar, Dec
mm	Month in two digits	03, 12
m	Month using capitalized first letter	M, D
dddd	Day using full name	Monday, Tuesday
ddd	Day using first three letters	Mon, Tue
dd	Day in two digits	05, 20
d	Day using capitalized first letter	M, T
HH	Hour in two digits (no leading zeros when symbolic identifier AM or PM is used)	05, 5 AM
MM	Minute in two digits	12, 02
SS	Second in two digits	07, 59
FFF	Millisecond in three digits	057
AM or PM	AM or PM inserted in text representing time	3:45:02 PM

The `formatIn` value must follow these guidelines:

- You cannot specify any field more than once. For example, you cannot use `'yy-mmm-dd-m'` because it has two month identifiers. The one exception to this is that you can combine one instance of `dd` with one instance of any of the other day identifiers. For example, `'dddd mmm dd yyyy'` is a valid input.
- When you use `AM` or `PM`, the `HH` field is also required.
- `datevec` does not accept formats that include `'QQ'`

`PivotYear` — Start year of 100-year date range
present minus 50 years (default) | integer

Start year of the 100-year date range in which a two-character year resides, specified as an integer. Use a pivot year to interpret dates that specify the year as two characters.

If `formatIn` contains the time of day, the pivot year is computed from the current time of the current day, month, and year. Otherwise it is computed from midnight of the current day, month, and year.

Example: `2000`

Note If the input date format specifies a four-character year, then the last two characters are truncated, and the *first* two characters specify the year. For example, if the date and pivot year are specified as `('25122015','ddmmyyyy',2000)`, then the resulting date is `25-12-2020`, not `25-12-2015`.

Data Types: `double`

Output Arguments

`DateVector` — Date vectors
vector | matrix

Date vectors, returned as an `m`-by-6 matrix, where each row corresponds to one date vector, and `m` is the total number of input date numbers or character vectors representing dates and times.

`[Y,M,D,H,MN,S]` — Components of the date vector
numeric scalars | numeric vectors

Components of the date vector (year, month, day, hour, minute, and second), returned as numeric scalars or numeric vectors. Milliseconds are a fractional part of the seconds output. When converting a `datetime` array `t`, these components are equal to the values of the `Year`, `Month`, `Day`, `Hour`, `Minute`, and `Second` properties. For example, `Y = t.Year`.

Limitations

- If the first input argument is a datetime array, a numeric array, or text representing dates and times, then `datevec` sets any month values less than 1 to 1 in the output.

Day values, `D`, less than 1 are set to the last day of the previous month minus |`D`|. However, if 0 ≤ `DateNumber` < 1, then `datevec(DateNumber)` returns a date vector of the form `[0 0 0 H MN S]`, where `H`, `MN`, and `S` are hours, minutes, and seconds, respectively.

- If the first input argument is a duration array, then the output of `datevec` does not represent points in time, but rather lengths of time measured in fixed-length time units. Each row represents a duration as the number of years, months, days, hours, minutes, and seconds whose sum is the total length of time represented by the input duration.

  However, a month cannot be a fixed-length time unit because different months have different numbers of days. Therefore the number of months is always zero. Instead, `datevec` represents a duration as the number of years (each 365.2425 days long), the number of days (each 86,400 seconds long), and the number of hours, minutes, and seconds whose sum is that duration. As another consequence, the number of days can be greater than 31.

Tips

- The vectorized calling syntax can offer significant performance improvement for large arrays.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- First argument must be a column vector, character vector, or character array.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- In generated code, this function supports duration arrays only. For more information, see “Code Generation for Duration Arrays” (MATLAB Coder) and “Duration Array Limitations for Code Generation” (MATLAB Coder).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- Only these syntaxes are supported:

  ```
  DateVector = datevec(t)
  ```

  ```
  [Y,M,D,H,MN,S] = datevec(t)
  ```

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`datenum` | `datestr` | `datetime`

Topics

“Represent Dates and Times in MATLAB”
“Carryover in Date Vectors and Strings”

Introduced before R2006a

day

Day number or name

Syntax

```
d = day(t)
d = day(t,dayType)
```

Description

`d = day(t)` returns the day-of-month numbers for the datetime values in `t`. The `d` output contains integer values from 1 to 31, depending on the month and year.

`d = day(t,dayType)` returns the type of day number or name specified by `dayType`.

The `day` function returns the day numbers or names of datetime values. To assign day values to datetime array `t`, use `t.Day` and modify the `Day` property.

Examples

Extract Day Number from Dates

Extract the day of month numbers from an array of dates.

```
t = [datetime('yesterday');datetime('today');datetime('tomorrow')]
```

```
t = 3x1 datetime array
   25-Aug-2019
   26-Aug-2019
   27-Aug-2019
```

```
d = day(t)
```

```
d = 3×1
```

```
    25
    26
    27
```

Determine Day of Week

Determine the day of the week for an arbitrary date, by specifying `'name'` as the second input to the `day` function.

```
t = datetime(2014,05,16)
```

```
t = datetime
   16-May-2014
```

```
d = day(t,'name')
```

```
d = 1x1 cell array
    {'Friday'}
```

Alternatively, specify `'dayofweek'` to return the day of the week as a number.

```
d = day(t,'dayofweek')
```

```
d = 6
```

Input Arguments

`t` — Input date and time
`datetime` array

Input date and time, specified as a `datetime` array.

dayType — Type of day values
`'dayofmonth'` (default) | `'dayofweek'` | `'dayofyear'` | `'name'` | `'shortname'`

Type of day values, specified as a value in the table.

Value of dayType	Description
`'dayofmonth'`	Day-of-month number, from 1 to 28, 29, 30, or 31. The range depends on the month.
`'dayofweek'`	Day-of-week number, from 1 to 7, where day 1 of the week is Sunday.
`'dayofyear'`	Day-of-year number, from 1 to 365 or 366, depending on the year.
`'name'`	Full day names, for example, `Sunday`. For `NaT` datetime values, the day name is the empty character vector, `''`.
`'shortname'`	Abbreviated day names, for example, `Sun`. For `NaT` datetime values, the day name is the empty character vector, `''`.

Note `day` returns day names in the language specified by the **Locale** option in the **Datetime format** section of the Preferences panel. To change the default datetime locale, see “Set Command Window Preferences”.

Output Arguments

d — Day number or name
`double` array | cell array of character vectors

Day number or name, returned as a numeric array of integers of type `double`, or a cell array of character vectors. `d` is the same size as `t`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`datetime` | `isweekend` | `month` | `quarter` | `week` | `year` | `ymd`

Introduced in R2014b

days

Duration in days

Syntax

```
D = days(X)
```

Description

`D = days(X)` returns an array of days equivalent to the values in `X`.

- If `X` is a numeric array, then `D` is a `duration` array in units of fixed-length days. A fixed-length day is equal to 24 hours.
- If `X` is a `duration` array, then `D` is a `double` array with each element equal to the number of fixed-length (24-hour) days in the corresponding element of `X`.

The `days` function converts between `duration` and `double` values. To display a duration in units of days, set its `Format` property to `'d'`.

Examples

Create Duration Array of Fixed-Length Days

```
X = magic(2);
D = days(X)
```

```
D = 2x2 duration array
    1 day   3 days
   4 days   2 days
```

Add each number of fixed-length days to the current date and time.

```
t = datetime('now') + D
```

```
t = 2x2 datetime array
   28-Aug-2019 02:32:00   30-Aug-2019 02:32:00
   31-Aug-2019 02:32:00   29-Aug-2019 02:32:00
```

Convert Durations to Numeric Array of Days

Create a `duration` array.

```
X = hours(23:20:95) + minutes(45)
```

```
X = 1x4 duration array
   23.75 hr   43.75 hr   63.75 hr   83.75 hr
```

Convert each duration in `X` to a number of days.

```
D = days(X)
```

```
D = 1×4

    0.9896    1.8229    2.6563    3.4896
```

Input Arguments

X — Input array
numeric array | duration array | logical array

Input array, specified as a numeric array, duration array, or logical array.

Tips

- `days` creates fixed-length (24 hour) days. To create days that account for Daylight Saving Time shifts when used in calendar calculations, use the `caldays` function.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This function fully supports code generation. For more information, see “Code Generation for Duration Arrays” (MATLAB Coder).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`caldays` | `duration`

Introduced in R2014b

dbclear

Remove breakpoints

Syntax

```
dbclear all
```

```
dbclear in file
dbclear in file at location
```

```
dbclear if condition
```

Description

`dbclear all` removes all breakpoints in all MATLAB code files, and all breakpoints set for errors, caught errors, caught error identifiers, warnings, warning identifiers, and `naninf`.

`dbclear in file` removes all breakpoints in the specified file. The `in` keyword is optional.

`dbclear in file at location` removes the breakpoint set at the specified location in the specified file. The `at` and `in` keywords are optional.

`dbclear if condition` removes all breakpoints set using the specified `condition`, such as `dbstop if error` or `dbstop if naninf`.

Examples

Clear Breakpoints in File

Set and then clear breakpoints in a program file.

Create a file, `buggy.m`, that contains these statements.

```
function z = buggy(x)
n = length(x);
z = (1:n)./x;
```

Add breakpoints at line 2 and line 3. List all breakpoints using `dbstatus`.

```
dbstop in buggy at 2
dbstop in buggy at 3
dbstatus
```

```
Breakpoints for buggy are on lines  2, 3.
```

Remove all the breakpoints in `buggy.m`. Call `dbstatus` to confirm that all breakpoints are cleared.

```
dbclear in buggy
dbstatus
```

Clear Breakpoints in File at Location

Set and then clear breakpoints in a program file at a certain location.

Create a file, `buggy.m`, that contains these statements.

```
function z = buggy(x)
n = length(x);
z = (1:n)./x;
```

Add breakpoints at line 2 and line 3. List all breakpoints using `dbstatus`.

```
dbstop in buggy at 2
dbstop in buggy at 3
dbstatus
```

```
Breakpoints for buggy are on lines  2, 3.
```

Remove the breakpoint at line 3 and call `dbstatus`.

```
dbclear in buggy at 3
dbstatus
```

```
Breakpoint for buggy is on line 2.
```

Clear Error Breakpoints

Set and clear an error breakpoint.

Create a file, `buggy.m`, that requires an input vector.

```
function z = buggy(x)
n = length(x);
z = (1:n)./x;
```

Set an error breakpoint, and call `buggy` with a matrix input instead of a vector.

```
dbstop if error
buggy(magic(3))
```

A run-time error occurs, and MATLAB goes into debug mode, pausing at line 3 in `buggy.m`.

```
Error using  ./
Matrix dimensions must agree.

Error in buggy at 3
z = (1:n)./x;
3   z = (1:n)./x;
```

Call `dbquit` to exit debug mode.

Clear the breakpoint, and call `buggy` again with a matrix input instead of a vector.

```
dbclear if error
buggy(magic(3))
```

A run-time error occurs, and MATLAB pauses execution immediately, without going into debug mode.

```
Error using  ./
Matrix dimensions must agree.
```

```
Error in buggy (line 3)
z = (1:n)./x;
```

Input Arguments

`file` — File name

character vector | string scalar

File name, specified as a character vector or string scalar. The file name can include a partial path, but must be in a folder on the search path, or in the current folder.

Example: `myfile.m`

If the file name is followed by the `-completenames` option, then the file does not need to be on the search path, as long as the file name is a fully qualified name.

Example: `c:\Program Files\MATLAB\myfile.m -completenames`

In addition, `file` can include a filemarker (`>`) to specify the path to a particular local function or to a nested function within the file.

Example: `myfile>myfunction`

Data Types: `char` | `string`

`location` — Location in `file`

line number | line number and anonymous function number | local function name

Location in `file` of breakpoint to clear, specified as follows:

- Line number in `file` specified as a character vector or string scalar. The default is `1`.
- Line number in `file`, at the anonymous function number, specified as a character vector or string scalar. For example, `1@2` specifies the second anonymous function on line number 1. If no anonymous function number is specified, then the default is `1`.
- Name of a local function in `file` specified as a character vector or string scalar.

Data Types: `char` | `string`

`condition` — Type of error breakpoint

`error` | `caught error` | `warning` | `naninf` | `...`

Type of error breakpoint, specified as follows:

- `error` — Run-time error that occurs outside a `try/catch` block. If you want to clear a breakpoint set for a specific error, then specify the message id. For example:
  - `dbclear if error` clears all breakpoints set with the `dbstop if error` command, including breakpoints with a specified message id.
  - `dbclear if error MATLAB:ls:InputsMustBeStrings` clears the error with a message ID of `MATLAB:ls:InputsMustBeStrings`.
- `caught error` — Run-time error that occurs within the `try` portion of a `try/catch` block. If you want to clear a breakpoint set for a specific error, then specify the message id.
- `warning` — Run-time warning. If you want to clear a breakpoint set for a specific warning, then specify the message id.

  This condition has no effect when you disable warnings with the `warning off all` command or when you disable warnings for the specified message id. For more information about disabling warnings, see `warning`.
- `naninf` — Not-a-number error or infinite value error. These errors occur when code returns an infinite value (`Inf`) or a value that is not a number (`NaN`) as a result of an operator, function call, or scalar assignment.

See Also

`dbquit` | `dbstack` | `dbstatus` | `dbstop` | `filemarker`

Topics

"Debug a MATLAB Program"

Introduced before R2006a

dbcont

Resume execution

Syntax

`dbcont`

Description

`dbcont` resumes execution of a MATLAB code file after pausing. Execution continues until another breakpoint is encountered, a pause condition is met, an error occurs, or execution completes successfully.

Note If you want to edit a file as a result of debugging, then it is a good practice to first quit debugging. Otherwise, you can get unexpected results.

Examples

Resume Execution While Debugging

Resume the execution of a program while debugging using the `dbcont` command.

Create a file, `buggy.m`, that contains these statements.

```
function z = buggy(x)
n = length(x);
z = (1:n)./x;
```

Add a breakpoint at line 2 and run the code. MATLAB pauses execution at the breakpoint.

```
dbstop in buggy at 2
buggy(5)
```

Run the `dbcont` command to continue execution to the end of the program and end debugging.

```
dbcont
```

See Also

`dbclear` | `dbquit` | `dbstatus` | `dbstep` | `dbstop`

Topics

"Debug a MATLAB Program"

Introduced before R2006a

dbdown

Reverse dbup workspace shift

Syntax

dbdown

Description

dbdown changes the current workspace and function context to the workspace and function context of the called MATLAB function or script in debug mode. You must issue the dbup command at least once before you issue this command. dbdown is the opposite of dbup.

Multiple dbdown commands change the workspace and function context to each successively executed MATLAB function or script on the stack until the current workspace and function context is the line at which MATLAB is paused. You do not have to move back to the paused line to continue execution or to step to the next line.

Examples

View Current and Calling Function Workspaces

Use the dbup and dbdown commands to view the current and calling function workspace of a function.

Create a file, myfile.m, that contains these statements.

```
function n = myfile(x)
n = myfunction(x-1);

function z = myfunction(y)
z = 2 / y ;
```

Set a breakpoint at `myfunction` and run `myfile` with an input of `1`. MATLAB pauses in the function `myfunction`, at the line `z = 2 / y`. Call `whos` to view the variables in the current workspace.

```
dbstop in myfile>myfunction
myfile(1);
whos
```

```
Name      Size            Bytes  Class     Attributes

  y         1x1                 8  double
```

The workspace contains the variable `y`, which is in the workspace context for `myfunction`.

Run the `dbup` command to switch to the workspace of the calling function, `myfile`. Call `whos` to view the variables in the new workspace.

```
dbup
whos
```

```
In workspace belonging to myfile (line 2)

  Name      Size            Bytes  Class     Attributes

  x         1x1                 8  double
```

The workspace contains the variable `x`, which is in the workspace context for `myfile`.

Run the `dbdown` command, and then call `whos`.

```
dbdown
whos
```

```
In workspace belonging to myfile>myfunction (line 5)

  Name      Size            Bytes  Class     Attributes

  y         1x1                 8  double
```

The workspace once again contains the variable `y`, which is in the workspace context for `myfunction`.

See Also

`dbstack` | `dbup` | `whos`

Topics

"Examine Values While Debugging"

Introduced before R2006a

dblquad

(Not recommended) Numerically evaluate double integral over rectangle

Note `dblquad` is not recommended. Use `integral2` instead.

Syntax

```
q = dblquad(fun,xmin,xmax,ymin,ymax)
q = dblquad(fun,xmin,xmax,ymin,ymax,tol)
q = dblquad(fun,xmin,xmax,ymin,ymax,tol,method)
```

Description

`q = dblquad(fun,xmin,xmax,ymin,ymax)` calls the `quad` function to evaluate the double integral `fun(x,y)` over the rectangle `xmin <= x <= xmax`, `ymin <= y <= ymax`. The input argument, `fun`, is a function handle that accepts a vector `x`, a scalar `y`, and returns a vector of integrand values.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

`q = dblquad(fun,xmin,xmax,ymin,ymax,tol)` uses a tolerance `tol` instead of the default, which is `1.0e-6`.

`q = dblquad(fun,xmin,xmax,ymin,ymax,tol,method)` uses the quadrature function specified as `method`, instead of the default `quad`. Valid values for `method` are `@quadl` or the function handle of a user-defined quadrature method that has the same calling sequence as `quad` and `quadl`.

Examples

Pass function handle `@integrnd` to `dblquad`:

```
Q = dblquad(@integrnd,pi,2*pi,0,pi);
```

where the function `integrnd.m` is:

```
function z = integrnd(x, y)
z = y*sin(x)+x*cos(y);
```

Pass anonymous function handle `F` to `dblquad`:

```
F = @(x,y)y*sin(x)+x*cos(y);
Q = dblquad(F,pi,2*pi,0,pi);
```

The `integrnd` function integrates `y*sin(x)+x*cos(y)` over the square `pi <= x <= 2*pi`, `0 <= y <= pi`. Note that the integrand can be evaluated with a vector `x` and a scalar `y`.

Nonsquare regions can be handled by setting the integrand to zero outside of the region. For example, the volume of a hemisphere is:

```
dblquad(@(x,y)sqrt(max(1-(x.^2+y.^2),0)), -1, 1, -1, 1)
```

or

```
dblquad(@(x,y)sqrt(1-(x.^2+y.^2)).*(x.^2+y.^2<=1), -1, 1, -1, 1)
```

See Also

`integral` | `integral2` | `integral3` | `quad` | `quad2d` | `quadgk` | `quadl` | `triplequad`

Topics

"Create Function Handle"
"Anonymous Functions"

Introduced before R2006a

dbmex

Enable MEX-file debugging on UNIX platforms

Syntax

```
dbmex state
```

Description

`dbmex state` enables or disables MEX-file debugging or returns to the debugger prompt on UNIX platforms.

To use this option, first start MATLAB from a debugger by typing `matlab -D`*`debugger`*, where *`debugger`* is the name of the debugger program. If you have already loaded the MEX file, remove it from memory using the `clear` function. Then call `dbmex on` before calling your MEX file.

Examples

Debug `yprime.c` with gdb

For instructions on debugging the `yprime.c` MEX file with the GNU® Debugger gdb, see “Debug on Linux Platforms”.

Input Arguments

`state` — Debug setting
`'on'` | `'off'` | `'stop'`

Debug control indicator specified as `'on'`, `'off'`, or `'stop'`. Use `'on'` or `'off'` to enable or disable MEX-file debugging. Use `'stop'` to return to the debugger prompt.

See Also

`clear`

Topics

"Debugging and Analysis"
"Debug on Mac Platforms"
"Debug on Linux Platforms"

Introduced before R2006a

dbquit

Quit debug mode

Syntax

```
dbquit
dbquit all
```

Description

`dbquit` terminates debug mode. The Command Window then displays the standard prompt (>>). The file being executed is *not* completed and no result is returned. All breakpoints remain in effect.

If MATLAB is in debug mode for more than one function, `dbquit` only terminates debugging for the active function. For example, if you debug `file1` and also debug `file2`, then running `dbquit` terminates debugging for `file2`, while `file1` remains in debug mode until you run `dbquit` again. However, if you debug `file3` and step into `file4`, then running `dbquit` terminates debugging for both file.

`dbquit all` ends debugging for all files simultaneously.

Examples

Quit Debugging Multiple Files

Create a file, `buggy.m`, that contains these statements.

```
function z = buggy(x)
n = length(x);
z = (1:n)./x;
```

Create a second file, `buggy2.m`, that contains these statements.

```
function z2 = buggy2(y)
m = length(y);
z2 = (1:m).*y;
```

Set breakpoints in `buggy` and `buggy2` and run both files. MATLAB pauses at the first line in `buggy` and `buggy2`.

```
dbstop in buggy
dbstop in buggy2
buggy(5)
buggy2(5)
```

Call the `dbstack` command to check the debugging status.

```
dbstack
```

```
In buggy2 (line 2)
In buggy (line 2)
```

Quit debugging. MATLAB ends debugging for `buggy2`, while `buggy` remains in debug mode.

```
dbquit
dbstack
```

```
In buggy (line 2)
```

Run `dbquit` again to exit debug mode for `buggy`.

Alternatively, `dbquit all` ends debugging for both files simultaneously.

See Also

`dbclear` | `dbcont` | `dbstack` | `dbstatus` | `dbstep` | `dbstop`

Topics

"Debug a MATLAB Program"

Introduced before R2006a

dbstack

Function call stack

Syntax

```
dbstack
dbstack(n)
dbstack( ___ , '-completenames')
ST = dbstack( ___ )
[ST,I] = dbstack( ___ )
```

Description

`dbstack` displays the line numbers and file names of the function calls that led to the current pause condition, listed in the order in which they execute. The display starts with the currently executing functions and continues until it reaches the topmost function. Each line number is a hyperlink to that line in the Editor. The notation `functionname>localfunctionname` describes the location of a local function.

`dbstack(n)` omits the first `n` stack frames from the display. This syntax can be useful, for example, when issuing a `dbstack` from within an error handler.

`dbstack( ___ , '-completenames')` outputs the fully qualified name of each function in the stack.

You can specify `'-completenames'` with any of the input arguments in the previous syntaxes.

`ST = dbstack( ___ )` returns the stack trace information in an `m`-by-`1` structure, `ST`.

`[ST,I] = dbstack( ___ )` also returns `I`, the current workspace index.

Examples

View Stack Trace Information While Debugging

While debugging a MATLAB code file, issue the `dbstack` command to view the stack trace information.

Create a file, `myfile.m`, that contains these statements.

```
function n = myfile(x)
n = myfunction(x-1);
end

function z = myfunction(y)
z = 2 / y;
end
```

Set a breakpoint at `myfunction` and run `myfile` with an input of `1`. While executing `myfunction`, MATLAB pauses at the line `z = 2/y`.

```
dbstop in myfile>myfunction
myfile(1);
```

Run the `dbstack` command. MATLAB displays the line numbers and file names of the function calls that led to the current breakpoint.

```
dbstack
```

```
In myfile>myfunction (line 5)
In myfile (line 2)
```

Store Complete Names for Each Function in Stack

Store the complete file name, function name, and line number for each function in the stack while debugging a file.

Create a file, `myfile.m`, that contains these statements.

```
function n = myfile(x)
n = myfunction(x-1);
end

function z = myfunction(y)
z = 2 / y;
end
```

Set a breakpoint at `myfunction` and run `myfile` with an input of `1`. While executing `myfunction`, MATLAB pauses at the line `z = 2/y`.

```
dbstop in myfile>myfunction
myfile(1);
```

Run the `dbstack` command, omitting the first frame and requesting complete names. MATLAB returns the stack trace information in the specified structure `ST`.

```
[ST, I] = dbstack('-completenames', 1)
```

```
ST =

    file: 'C:\myProject\myfile.m'
    name: 'myfile'
    line: 2


I =

     1
```

Input Arguments

n — Number of frames to omit
nonnegative integer

Number of frames to omit, specified as a nonnegative integer.

Output Arguments

ST — Stack trace information
structure array

Stack trace information, returned as an `m`-by-`1` structure, where `m` is the number of functions in the call stack. The structure has these fields.

`file`	File in which the function appears. This field is empty if there is no file.
`name`	Function name within the file.

`line`	Line number of function call.

Note If you step past the end of a file, `dbstack` returns a negative line number value to identify that special case. For example, if the last line to be executed is line `15`, then the `dbstack` line number is `15` before you execute that line and `-15` after.

`I` — Current workspace index
positive integer

Current workspace index, returned as a positive integer. The index represents the number of workspaces between your current workspace and the workspace in which MATLAB is currently paused or executing.

See Also

`dbdown` | `dbquit` | `dbstep` | `dbstop` | `dbup` | `evalin` | `mfilename` | `whos`

Topics

"Examine Values While Debugging"

Introduced before R2006a

dbstatus

List all breakpoints

Syntax

```
dbstatus
dbstatus file
dbstatus -completenames
dbstatus file -completenames
b = dbstatus( ___ )
```

Description

`dbstatus` lists all the breakpoints in effect, including errors, caught errors, warnings, and `naninfs`. For nonerror breakpoints, MATLAB displays the line number for which the breakpoint is set. Each line number is a hyperlink that you can click to go directly to that line in the Editor.

`dbstatus file` lists all breakpoints in effect for the specified `file`.

`dbstatus -completenames` displays, for each breakpoint, the fully qualified name of the function or file containing the breakpoint.

`dbstatus file -completenames` displays, for each breakpoint in the specified file, the fully qualified name for the function or file containing the breakpoint.

`b = dbstatus( ___ )` returns breakpoint information in an `m`-by-`1` structure. To save the current breakpoints to restore them later using `dbstop(b)`, use this syntax. You also can specify the file name and `'completenames'`.

Examples

List All Breakpoints

Create a file, `myfile.m`, that contains these statements.

```
function n = myfile(x)
n = myfunction(x-1);

function z = myfunction(y)
z = 2/y ;
```

Set an error breakpoint, and a standard breakpoint at the first line in `myfile`.

```
dbstop if error
dbstop in myfile
```

Run the `dbstatus` command. MATLAB displays the active breakpoints: the standard breakpoint in`myfile`, and the error breakpoint.

```
dbstatus
```

```
Breakpoint for myfile is on line 2.
Stop if error.
```

List All Breakpoints in File

List all breakpoints in a specified file, showing complete names.

Create a file, `myfile.m`, that contains these statements.

```
function n = myfile(x)
n = myfunction(x-1);

function z = myfunction(y)
z = 2 / y ;
```

Set an error breakpoint and a standard breakpoint at the first line in `myfile`.

```
dbstop if error
dbstop in myfile
```

Run the `dbstatus` command, specifying the file `myfile` and requesting complete names. MATLAB displays the active breakpoints: the standard breakpoint in`myfile`, and the error breakpoint.

```
dbstatus myfile -completenames
```

```
Breakpoint for C:\myProject\myfile.m>myfile is on line 2.
```

Notice that the error breakpoint is not listed. Only breakpoints specific to the specified file are included in the list.

Restore Saved Breakpoints

Set, save, clear, and then restore saved breakpoints.

Create a file, `buggy.m`, which contains these statements.

```
function z = buggy(x)
n = length(x);
z = (1:n)./x;
```

Set an error breakpoint and a standard breakpoint at the second line in `buggy`.

```
dbstop at 2 in buggy
dbstop if error
```

Run `dbstatus`. MATLAB describes the breakpoints you set.

```
dbstatus
```

```
Breakpoint for buggy is on line 2.
Stop if error.
```

Assign a structure representing the breakpoints to the variable `b`, and then save `b` to the MAT-file `buggybrkpnts`. Use `b=dbstatus('-completenames')` to save absolute paths and the breakpoint function nesting sequence.

```
b = dbstatus('-completenames');
save buggybrkpnts b
```

Clear all breakpoints.

```
dbclear all
```

Restore the breakpoints by loading the MAT-file and calling `dbstop` with the saved structure, `b`.

```
load buggybrkpnts
dbstop(b)
```

Input Arguments

file — File name

character vector | string scalar

File name, specified as a character vector or string scalar. The file name can include a partial path, but must be in a folder on the search path, or in the current folder.

Example: `myfile.m`

When specifying methods, private functions, or private methods, use the `/` character.

Example: `myclass/myfunction`

Example: `private/myfunction`

Example: `myclass/private/myfunction`

In addition, `file` can include a filemarker (`>`) to specify the path to a particular local function or to a nested function within the file.

Example: `myfile>myfunction`

Data Types: `char` | `string`

Output Arguments

b — List of breakpoints

structure array

List of breakpoints currently in effect, returned as a `m`-by-`1` structure, where `m` is the number of breakpoints. This table shows the fields in the structure.

`name`	Function name.
`file`	Full path for file containing breakpoints.
`line`	Vector of breakpoint line numbers.

`anonymous`	Vector of integers representing the anonymous functions in the line numbers represented by the `line` field. For example, `2` means the second anonymous function in that line. A value of `0` means that the breakpoint is at the start of the line, not in an anonymous function.
`expression`	Cell vector of character vectors, containing the breakpoint conditional expressions corresponding to line numbers in the `line` field.
`cond`	Character vector containing the condition (`'error'`, `'caught error'`, `'warning'`, or `'naninf'`).
`identifier`	If `cond` is `'error'`, `'caught error'`, or `'warning'`, a cell vector of character vectors containing MATLAB message identifiers for which the particular `cond` state is set.

See Also

`dbclear` | `dbcont` | `dbquit` | `dbstop` | `error` | `warning`

Introduced before R2006a

dbstep

Execute next executable line from current breakpoint

Syntax

```
dbstep
dbstep in
dbstep out
dbstep nlines
```

Description

`dbstep` executes the next executable line of the current file during debugging, skipping any breakpoints set in functions called by the current line.

`dbstep in` steps to the next executable line. If that line contains a call to another MATLAB code file function, then execution steps to the first executable line of the called function. If no call exists to a MATLAB code file on that line, `dbstep in` is the same as `dbstep`.

`dbstep out` runs the rest of the current function and pauses just after leaving the function. MATLAB pauses execution at any breakpoint it encounters.

`dbstep nlines` executes the specified number of executable lines. MATLAB pauses execution at any breakpoint it encounters.

Examples

Step Over Called Functions

Use `dbstep` to step over a called local function.

Create a file, `myfile.m`, that contains these statements.

```
function n = myfile(x)
n = myfunction(x-1);

function z = myfunction(y)
z = 2/y;
```

Set a breakpoint at the first line in `myfile` and run `myfile` with an input of `1`. MATLAB pauses in the function `myfile`, at the line `n = myfunction(x-1)`.

```
dbstop in myfile
myfile(2);
```

```
2   n = myfunction(x-1);
```

Step to the next execution line. MATLAB reaches the end of the function `myfile`.

```
K>> dbstep
```

```
End of function myfile.
```

Step once more to complete the execution of `myfile` and end debugging.

Step Through Called Functions

Use `dbstep` to step through a called local function.

Create a file, `myfile.m`, that contains these statements.

```
function n = myfile(x)
n = myfunction(x-1);

function z = myfunction(y)
z = 2/y;
```

Set a breakpoint at the first line in `myfile` and run `myfile` with an input of `2`. MATLAB pauses in the function `myfile`, at the line `n = myfunction(x-1)`.

```
dbstop in myfile
myfile(2);
```

```
2   n = myfunction(x-1);
```

Step into `myfunction`. MATLAB enters `myfunction` and pauses at the first line in the function.

```
K>> dbstep in
```

```
5   z = 2/y;
```

Step through the next four lines of code, completing the execution of `myfile` and ending debugging.

```
K>> dbstep 4
```

```
ans =

   2
```

Step Into and Out of Called Functions

Use `dbstep` to step into and out of a called local function.

Create a file, `myfile.m`, that contains these statements.

```
function n = myfile(x)
n = myfunction(x-1);

function z = myfunction(y)
z = 2/y;
```

Set a breakpoint at the first line in `myfile` and run `myfile` with an input of `2`. MATLAB pauses in the function `myfile`, at the line `n = myfunction(x-1)`.

```
dbstop in myfile
myfile(2);
```

Step into `myfunction`. MATLAB enters `myfunction` and pauses at the first line in the function.

```
K>> dbstep in
```

```
5   z = 2/y;
```

Step out of `myfunction`. MATLAB finishes executing `myfunction` and returns to the calling function `myfile`.

```
K>> dbstep out
```

```
2   n = myfunction(x-1);
```

Step out one more time to complete the execution of `myfile` and end debugging.

Input Arguments

`nlines` — Number of executable lines
positive integer

Number of executable lines to execute, specified as a positive integer.

See Also

`dbclear` | `dbcont` | `dbquit` | `dbstatus` | `dbstop`

Topics

"Debug a MATLAB Program"

Introduced before R2006a

dbstop

Set breakpoints for debugging

Syntax

```
dbstop in file
dbstop in file at location
dbstop in file if expression
dbstop in file at location if expression

dbstop if condition

dbstop(b)
```

Description

`dbstop in file` sets a breakpoint at the first executable line in `file`. When you run `file`, MATLAB enters debug mode, pauses execution at the breakpoint, and displays the line where it is paused.

`dbstop in file at location` sets a breakpoint at the specified location. MATLAB execution pauses immediately before that location, unless the location is an anonymous function. If the location is an anonymous function, then execution pauses just after the breakpoint.

`dbstop in file if expression` sets a conditional breakpoint at the first executable line of the file. Execution pauses only if `expression` evaluates to true `(1)`.

`dbstop in file at location if expression` sets a conditional breakpoint at the specified location. Execution pauses at or just before that location only if the `expression` evaluates to true.

`dbstop if condition` pauses execution at the line that meets the specified `condition`, such as `error` or `naninf`. Unlike other breakpoints, you do not set this breakpoint at a specific line in a specific file. MATLAB pauses at any line in any file when the specified `condition` occurs.

`dbstop(b)` restores breakpoints you previously saved to `b`. The files containing the saved breakpoints must be on the search path or in the current folder. MATLAB assigns breakpoints by line number, so the lines in the file must be the same as when you saved the breakpoints.

Examples

Pause at First Executable Line

Set a breakpoint and pause execution at the first executable line of a program.

Create a file, `buggy.m`, that contains these statements.

```
function z = buggy(x)
n = length(x);
z = (1:n)./x;
```

Issue the `dbstop` command and run `buggy`.

```
dbstop in buggy
buggy(1:5)
```

MATLAB displays the line where it pauses and enters debug mode.

```
2   n = length(x);
K>>
```

Type `dbquit` to exit debug mode.

Pause at Function in File

Set a breakpoint in a program at the first executable line of a local function.

Create a file, `myfile.m`, that contains these statements

```
function n = myfile(x)
n = myfunction(x);

function y = myfunction(x)
y = x + 1;
```

Set a breakpoint at `myfunction`.

```
dbstop in myfile>myfunction
```

Pause in File After *n* Loop Iterations

Set a breakpoint in a program that causes MATLAB to pause after some iterations of a loop.

Create a file, `myprogram.m`, that contains these statements

```
x = ones(1,10);

for n = 1:10
x(n) = x(n) + 1;
end
```

Set a breakpoint to pause when `n >= 4`, and run the code.

```
dbstop in myprogram at 4 if n>=4
myprogram
```

MATLAB pauses at line 4 after 3 iterations of the loop, when `n = 4`.

```
4   x(n) = x(n) + 1;
K>>
```

Type `dbquit` to exit debug mode.

Pause If Error

Set a breakpoint and pause execution if a run-time error occurs.

Create a file, `mybuggyprogram.m`, that contains these statements.

```
x = ones(1,10);

for n = 1:10
x(n) = x(n+1) + 1;
end
```

Set an error breakpoint, and call `mybuggyprogram`.

```
dbstop if error
mybuggyprogram
```

A run-time error occurs, and MATLAB goes into debug mode, pausing at line 4 in `mybuggyprogram.m`.

```
Index exceeds matrix dimensions.
Error in mybuggyprogram (line 4)
x(n) = x(n+1) + 1;
4   x(n) = x(n+1) + 1;
```

Type `dbquit` to exit debug mode.

Run `MException.last` to obtain the error message identifier generated by the program.

```
MException.last
```

```
ans =

  MException with properties:

    identifier: 'MATLAB:badsubscript'
       message: 'Index exceeds matrix dimensions.'
         cause: {}
         stack: [1×1 struct]
```

Clear the error breakpoint and set a new error breakpoint specifying the identifier of the error message to catch. Call `mybuggyprogram`.

```
dbclear if error
dbstop if error MATLAB:badsubscript
mybuggyprogram
```

The same run-time error occurs, and MATLAB goes into debug mode, pausing at line 4 in `mybuggyprogram.m`.

```
Index exceeds matrix dimensions.
Error in mybuggyprogram (line 4)
x(n) = x(n+1) + 1;
4   x(n) = x(n+1) + 1;
```

Type `dbquit` to exit debug mode.

Pause If NanInf

Set a breakpoint and pause execution if the code returns a `NaN` value.

Create a file, `buggy.m`, that requires an input vector.

```
function z = buggy(x)
n = length(x);
z = (1:n)./x;
```

Set a warning breakpoint, and call `buggy` with an input vector containing a `0` as one of its elements.

```
dbstop if naninf
buggy(0:2)
```

A division by zero error occurs, and MATLAB goes into debug mode, pausing at line 3 in `buggy.m`.

```
NaN/Inf breakpoint hit for buggy on line 3.
```

Type `dbquit` to exit debug mode.

Restore Saved Breakpoints

Set, save, clear, and then restore saved breakpoints.

Create a file, `buggy.m`, which contains these statements.

```
function z = buggy(x)
n = length(x);
z = (1:n)./x;
```

Set an error breakpoint and a standard breakpoint at the second line in `buggy`.

```
dbstop at 2 in buggy
dbstop if error
```

Run `dbstatus`. MATLAB describes the breakpoints you set.

```
dbstatus
```

```
Breakpoint for buggy is on line 2.
Stop if error.
```

Assign a structure representing the breakpoints to the variable `b`, and then save `b` to the MAT-file `buggybrkpnts`. Use `b=dbstatus('-completenames')` to save absolute paths and the breakpoint function nesting sequence.

```
b = dbstatus('-completenames');
save buggybrkpnts b
```

Clear all breakpoints.

```
dbclear all
```

Restore the breakpoints by loading the MAT-file and calling `dbstop` with the saved structure, `b`.

```
load buggybrkpnts
dbstop(b)
```

Input Arguments

`file` — File name

character vector | string scalar

File name, specified as a character vector or string scalar. The file name can include a partial path, but must be in a folder on the search path or in the current folder.

Example: `myfile.m`

If the file name includes the `-completenames` option, then the file does not need to be on the search path, as long as the file name is a “Fully Qualified Name” on page 1-3116.

Example: `c:\Program Files\MATLAB\myfile.m -completenames`

In addition, `file` can include a filemarker (>) to specify the path to a particular local function or to a nested function within the file.

Example: `myfile>myfunction`

If `file` is not a MATLAB code file (for instance, it is a built-in or MDL-file), then MATLAB issues a warning. MATLAB cannot pause *in* the file, so it pauses before executing the file.

Data Types: `char` | `string`

`location` — Breakpoint location

line number | line number and anonymous function number | local function name

Breakpoint location to set in `file`, specified as one of these options:

- Line number in `file` specified as a character vector or string scalar. The default is `1`.
- Line number in `file`, located at the anonymous function number and specified as a character vector or string scalar. For example, `1@2` specifies line number 1 at the second anonymous function. The default anonymous function number is `1`.
- Name of a local function in `file`, specified as a character vector or string scalar.

Note When setting a breakpoint, you cannot specify `location` if `file` includes a filemarker. For example, the command `dbstop in myfile>myfilefunction at 5` is invalid.

Data Types: `char` | `string`

`expression` — Logical expression
character vector | string scalar

Logical expression that evaluates to a scalar logical value of `1` or `0`, specified as a character vector or string scalar.

Example: `n >= 4`

Data Types: `char` | `string`

`condition` — Pause condition
`error` | `caught error` | `warning` | `naninf` | `...`

Pause condition, specified as one of these options:

- `error` — Run-time error that occurs outside a `try/catch` block. You cannot resume execution after an uncaught run-time error.

  If you want execution to pause only if a specific error occurs, specify the message id. For example:

  - `dbstop if error` pauses execution at the first run-time error that occurs outside a `try/catch` block.
  - `dbstop if error MATLAB:ls:InputsMustBeStrings` pauses execution at the first run-time error outside a `try/catch` block that has a message ID of `MATLAB:ls:InputsMustBeStrings`.

- `caught error` — Run-time error that occurs within the `try` portion of a `try/catch` block. If you want execution to pause only if a specific error occurs, specify the message id.
- `warning` — Run-time warning occurs. If you want execution to pause only if a specific warning occurs, specify the message id.

  This condition has no effect if you disable warnings with the `warning off all` command or if you disable warnings for the specified `id`. For more information about disabling warnings, see `warning`.
- `naninf` — The code returns an infinite value (`Inf`) or a value that is not a number (`NaN`) as a result of an operator, function call, or scalar assignment.

b — List of breakpoints
structure array

List of breakpoints previously saved to a structure array using `b=dbstatus`.

More About

Fully Qualified Name

A fully qualified name is an exact file name that is uniquely specified such that it cannot be mistaken for any other file on your system.

- Windows platforms — A file name that begins with two back slashes (`\\`) or with a drive letter followed by a colon (`:`).
- UNIX platforms — A file name that begins with a slash (`/`) or a tilde (~).

Tips

- Before you begin debugging, make sure that your program is saved and that the program and any files it calls exist on your search path or in the current folder.
- To resume execution after a breakpoint pauses execution, use `dbcont` or `dbstep`. To exit debug mode, use `dbquit`. To remove all the breakpoints in the file, use `dbclear in` *`filename`*. To remove all breakpoints in all files, use `dbclear all`. For more information, see `dbclear`.

- MATLAB can become unresponsive when it pauses at a breakpoint while displaying a modal dialog box or figure created by your program. To exit debug mode and return to the MATLAB prompt (>>), use **Ctrl+C**.

See Also

`dbclear` | `dbcont` | `dbquit` | `dbstack` | `dbstatus` | `dbstep` | `dbtype` | `keyboard`

Topics

"Debug a MATLAB Program"

Introduced before R2006a

dbtype

Display file with line numbers

Syntax

```
dbtype filename
dbtype filename start:end
```

Description

`dbtype filename` displays the contents of `filename` with line numbers preceding each line. This is helpful when setting breakpoints in a program file with `dbstop`. You cannot use `dbtype` to display the source code of built-in functions.

`dbtype filename start:end` displays the portion of the file starting at line number `start` and ending at line number `end`. To display a single line, you can specify only the `start` value.

Examples

View File with Line Numbers

Display the example file `buggy.m` with line numbers preceding each line.

```
dbtype buggy
```

```
1     function z = buggy(x)
2     n = length(x);
3     z = (1:n)./x;
```

View Function Arguments

View the input and output arguments for an example function, `buggy`, by displaying the first line of the file. Since `start` and `end` are the same in this case, you can omit the `end` value.

```
dbtype buggy 1
```

```
1     function z = buggy(x)
```

Input Arguments

`filename` — File name
character vector | string scalar

File name, specified as a character vector or string scalar. The file name can be a partial path, but it must be in a folder on the search path or in the current folder. Otherwise, specify the full path name.

Data Types: `char` | `string`

`start` — Starting line number
positive integer

Starting line number, specified as a positive integer.

`end` — Ending line number
positive integer

Ending line number, specified as a positive integer.

See Also

`dbclear` | `dbstatus` | `dbstop`

Introduced before R2006a

dbup

Shift current workspace to workspace of caller in debug mode

Syntax

dbup

Description

dbup changes the current workspace and function context to the workspace and function context of the calling function or script in debug mode. Then you can examine the calling MATLAB function or script to determine what caused the arguments to be passed to the called function.

Each dbup command changes the workspace and function context to an earlier calling function or script on the stack until the base workspace and function context is reached. You do not need to return to the line at which MATLAB is paused to continue execution or to step to the next line.

Examples

View Current and Calling Function Workspaces

Create a file, myfile.m, that contains these statements.

```
function n = myfile(x)
n = myfunction(x-1);

function z = myfunction(y)
z = 2 / y ;
```

Set a breakpoint at myfunction and run myfile with an input of 1. MATLAB pauses in the function myfunction, at the line z = 2 / y. Call whos to view the variables in the current workspace.

```
dbstop in myfile>myfunction
myfile(1);
whos
```

```
Name      Size            Bytes  Class     Attributes

  y         1x1                 8  double
```

The workspace contains the variable `y`, which is in the workspace context for `myfunction`.

Run the `dbup` command to switch to the workspace of the calling function, `myfile`. Then, call `whos`.

```
dbup
whos
```

```
In workspace belonging to myfile (line 2)

  Name      Size            Bytes  Class     Attributes

  x         1x1                 8  double
```

The workspace contains the variable `x`, which is in the workspace context for `myfile`.

Tips

- If you receive an error message such as the following, it means that the parent workspace is under construction so that the value of `x` is unavailable:

  ```
  ??? Reference to a called function result under construction x
  ```

See Also

`dbdown` | `dbstack` | `whos`

Topics

"Examine Values While Debugging"

Introduced before R2006a

dde23

Solve delay differential equations (DDEs) with constant delays

Syntax

```
sol = dde23(ddefun,lags,history,tspan)
sol = dde23(ddefun,lags,history,tspan,options)
```

Arguments

`ddefun`	Function handle that evaluates the right side of the differential equations $y'(t) = f(t, y(t), y(t - \tau_1), ..., y(t - \tau_k))$. The function must have the form `dydt = ddefun(t,y,Z)` where `t` corresponds to the current t, `y` is a column vector that approximates $y(t)$, and `Z(:,j)` approximates $y(t - \tau_j)$ for delay τ_j = `lags(j)`. The output is a column vector corresponding to $f(t, y(t), y(t - \tau_1), ..., y(t - \tau_k))$.
`lags`	Vector of constant, positive delays $\tau_1, ..., \tau_k$.
`history`	Specify `history` in one of three ways: • A function of t such that `y = history(t)` returns the solution $y(t)$ for $t \le t_0$ as a column vector • A constant column vector, if $y(t)$ is constant • The solution `sol` from a previous integration, if this call continues that integration
`tspan`	Interval of integration from `t0=tspan(1)` to `tf=tspan(end)` with `t0 < tf`.
`options`	Optional integration argument. A structure you create using the `ddeset` function. See `ddeset` for details.

Description

`sol = dde23(ddefun,lags,history,tspan)` integrates the system of DDEs

$y'(t) = f(t, y(t), y(t - \tau_1), ..., y(t - \tau_k))$

on the interval $[t_0,t_f]$, where τ_1, ..., τ_k are constant, positive delays and t_0,t_f. The input argument, `ddefun`, is a function handle.

"Parameterizing Functions" explains how to provide additional parameters to the function `ddefun`, if necessary.

`dde23` returns the solution as a structure `sol`. Use the auxiliary function `deval` and the output `sol` to evaluate the solution at specific points `tint` in the interval `tspan = [t0,tf]`.

```
yint = deval(sol,tint)
```

The structure `sol` returned by `dde23` has the following fields.

`sol.x`	Mesh selected by `dde23`
`sol.y`	Approximation to $y(x)$ at the mesh points in `sol.x`.
`sol.yp`	Approximation to $y'(x)$ at the mesh points in `sol.x`
`sol.solver`	Solver name, `'dde23'`

`sol = dde23(ddefun,lags,history,tspan,options)` solves as above with default integration properties replaced by values in `options`, an argument created with `ddeset`. See `ddeset` and "Solving Delay Differential Equations" for more information.

Commonly used options are scalar relative error tolerance `'RelTol'` (`1e-3` by default) and vector of absolute error tolerances `'AbsTol'` (all components are `1e-6` by default).

Use the `'Jumps'` option to solve problems with discontinuities in the history or solution. Set this option to a vector that contains the locations of discontinuities in the solution prior to `t0` (the history) or in coefficients of the equations at known values of t after `t0`.

Use the `'Events'` option to specify a function that `dde23` calls to find where functions $g(t, y(t), y(t - \tau_1), ..., y(t - \tau_k))$ vanish. This function must be of the form

```
[value,isterminal,direction] = events(t,y,Z)
```

and contain an event function for each event to be tested. For the `k`th event function in `events`:

- `value(k)` is the value of the `k`th event function.
- `isterminal(k) = 1` if you want the integration to terminate at a zero of this event function and `0` otherwise.
- `direction(k) = 0` if you want `dde23` to compute all zeros of this event function, `+1` if only zeros where the event function increases, and `-1` if only zeros where the event function decreases.

If you specify the `'Events'` option and events are detected, the output structure `sol` also includes fields:

`sol.xe`	Row vector of locations of all events, i.e., times when an event function vanished
`sol.ye`	Matrix whose columns are the solution values corresponding to times in `sol.xe`
`sol.ie`	Vector containing indices that specify which event occurred at the corresponding time in `sol.xe`

Examples

This example solves a DDE on the interval [0, 5] with lags 1 and 0.2. The function `ddex1de` computes the delay differential equations, and `ddex1hist` computes the history for `t <= 0`.

Note The file, `ddex1.m`, contains the complete code for this example. To see the code in an editor, type `edit ddex1` at the command line. To run it, type `ddex1` at the command line.

```
sol = dde23(@ddex1de,[1, 0.2],@ddex1hist,[0, 5]);
```

This code evaluates the solution at 100 equally spaced points in the interval `[0,5]`, then plots the result.

```
tint = linspace(0,5);
yint = deval(sol,tint);
plot(tint,yint);
```

`ddex1` shows how you can code this problem using local functions. For more examples see `ddex2`.

Algorithms

`dde23` tracks discontinuities and integrates with the explicit Runge-Kutta (2,3) pair and interpolant of `ode23`. It uses iteration to take steps longer than the lags.

References

[1] Shampine, L.F. and S. Thompson, “Solving DDEs in MATLAB,” *Applied Numerical Mathematics*, Vol. 37, 2001, pp. 441-458.

[2] Kierzenka, J., L.F. Shampine, and S. Thompson, “Solving Delay Differential Equations with dde23”

See Also

`ddeget` | `ddensd` | `ddesd` | `ddeset` | `deval`

Topics

“Create Function Handle”

Introduced before R2006a

ddeget

Extract properties from delay differential equations options structure

Syntax

```
val = ddeget(options,'name')
val = ddeget(options,'name',default)
```

Description

`val = ddeget(options,'name')` extracts the value of the named property from the structure `options`, returning an empty matrix if the property value is not specified in `options`. It is sufficient to type only the leading characters that uniquely identify the property. Case is ignored for property names. `[]` is a valid `options` argument.

`val = ddeget(options,'name',default)` extracts the named property as above, but returns `val = default` if the named property is not specified in `options`. For example,

```
val = ddeget(opts,'RelTol',1e-4);
```

returns `val = 1e-4` if the `RelTol` is not specified in `opts`.

See Also

`dde23` | `ddensd` | `ddesd` | `ddeset`

Introduced before R2006a

ddensd

Solve delay differential equations (DDEs) of neutral type

Syntax

```
sol = ddensd(ddefun,dely,delyp,history,tspan)
sol = ddensd(ddefun,dely,delyp,history,tspan,options)
```

Description

`sol = ddensd(ddefun,dely,delyp,history,tspan)` integrates a system of delay differential equations of neutral type, that has the form

$$y'(t) = f(t, y(t), y(dy_1),..., y(dy_p), y'(dyp_1),..., y'(dyp_q)) \tag{1-1}$$

where

- t is the independent variable representing time.
- dy_i is any of p solution delays.
- dyp_j is any of q derivative delays.

`sol = ddensd(ddefun,dely,delyp,history,tspan,options)` replaces default integration parameters with those specified in `options`, a structure created with the `ddeset` function.

Examples

Neutral DDE with Two Delays

Solve the following neutral DDE, presented by Paul, for $0 \le t \le \pi$.

$$y'(t) = 1 + y(t) - 2y(t/2)^2 - y'(t - \pi)$$

The solution history is $y(t) = \cos(t)$ for $t \leq 0$.

Create a new program file in the editor. This file will contain a main function and four local functions.

Define the first-order DDE as a local function named `ddefun`.

```
function yp = ddefun(t,y,ydel,ypdel)
    yp = 1 + y - 2*ydel^2 - ypdel;
end
```

Define the solution delay as a local function named `dely`.

```
function dy = dely(t,y)
    dy = t/2;
end
```

Define the derivative delay as a local function named `delyp`.

```
function dyp = delyp(t,y)
    dyp = t-pi;
end
```

Define the solution history as a local function named `history`.

```
function y = history(t)
    y = cos(t);
end
```

Define the interval of integration and solve the DDE using `ddensd`. Add this code to the main function.

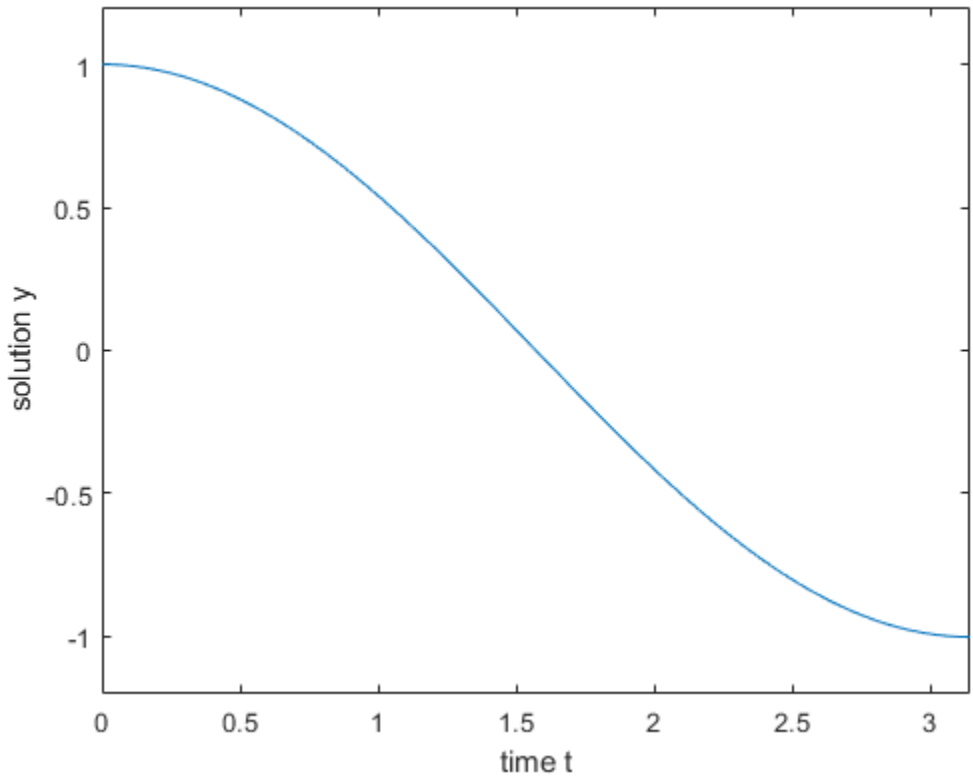
```
tspan = [0 pi];
sol = ddensd(@ddefun,@dely,@delyp,@history,tspan);
```

Evaluate the solution at 100 equally spaced points between 0 and π. Add this code to the main function.

```
tn = linspace(0,pi);
yn = deval(sol,tn);
```

Plot the results. Add this code to the main function.

```
plot(tn,yn);
xlim([0 pi]);
ylim([-1.2 1.2]);
xlabel('time t');
ylabel('solution y');
```



Run your entire program to calculate the solution and display the plot. The file `ddex4.m` contains the complete code for this example. To see the code in an editor, type `edit ddex4` at the command line.

Input Arguments

`ddefun` — Derivative function
function handle

Derivative function, specified as a function handle whose syntax is `yp = ddefun(t,y,ydel,ypdel)`. The arguments for `ddefun` are described in the table below.

ddefun Argument	Description
`t`	A scalar value representing the current value of time, *t*.
`y`	A vector that represents $y(t)$ in "Equation 1-1". The size of this vector is `n`-by-`1`, where `n` is the number of equations in the system you want to solve.
`ydel`	A matrix whose columns, `ydel(:,i)`, represent $y(dy_i)$. The size of this matrix is `n`-by-`p`, where `n` is the number of equations in the system you want to solve, and `p` is the number of $y(dy)$ terms in "Equation 1-1".
`ypdel`	A matrix whose columns, `ypdel(:,j)` represent $y'(dyp_j)$. The size of this matrix is `n`-by-`q`, where `n` is the number of equations in the system you want to solve, and `q` is the number of $y'(dyp)$ terms in "Equation 1-1".
`yp`	The result returned by `ddefun`. It is an `n`-by-`1` vector whose elements represent the right side of "Equation 1-1".

`dely` — Solution delays
function handle | vector

Solution delays, specified as a function handle, which returns $dy_1,\ldots, dy_p$ in "Equation 1-1". Alternatively, you can pass constant delays in the form of a vector.

If you specify `dely` as a function handle, the syntax must be `dy = dely(t,y)`. The arguments for this function are described in the table below.

dely Argument	Description
`t`	A scalar value representing the current value of time, *t*.
`y`	A vector that represents *y*(*t*) in “Equation 1-1”. The size of this vector is `n`-by-`1`, where `n` is the number of equations in the system you want to solve.
`dy`	A vector returned by the `dely` function whose values are the solution delays, dy_i , in “Equation 1-1”. The size of this vector is `p`-by-`1`, where `p` is the number of solution delays in the equation. Each element must be less than or equal to *t*.

If you want to specify constant solution delays having the form $dy_i = t - \tau_i$, then `dely` must be a vector, where `dely(i)` = τ_i. Each value in this vector must be greater than or equal to zero.

If *dy* is not present in the problem, set `dely` to `[]`.

Data Types: `function_handle` | `single` | `double`

`delyp` — Derivative delays
function handle | vector

Derivative delays, specified as a function handle, which returns $dyp_1,..., dyp_q$ in “Equation 1-1”. Alternatively, you can pass constant delays in the form of a vector.

If `delyp` is a function handle, its syntax must be `dyp = delyp(t,y)`. The arguments for this function are described in the table below.

delyp Argument	Description
`t`	A scalar value representing the current value of time, *t*.
`y`	A vector that represents *y*(*t*) in “Equation 1-1”. The size of this vector is `n`-by-`1`, where `n` is the number of equations in the system you want to solve.

delyp Argument	Description
`dyp`	A vector returned by the `delyp` function whose values are the derivative delays, dyp_j, in "Equation 1-1". The size of this vector must be `q`-by-`1`, where `q` is the number of solution delays, dyp_j, in the equation. Each element of `dyp` must be less than t. There is one exception to this restriction: if you are solving an initial value DDE, the value of `dyp` can equal t at $t = t_0$. For more information, see "Initial Value Neutral Delay Differential Equations" on page 1-3134.

If you want specify constant derivative delays having the form $dyp_j = t - \tau_j$, then `delyp` must be a vector, where `delyp(j)` = τ_j. Each value in this vector must be greater than zero. An exception to this restriction occurs when you solve initial value problems for DDEs of neutral type. In such cases, a value in `delyp` can equal zero at $t = t_0$. See "Initial Value Neutral Delay Differential Equations" on page 1-3134 for more information.

If *dyp* is not present in the problem, set `delyp` to `[]`.

Data Types: `function_handle` | `single` | `double`

`history` — Solution history

function handle | column vector | structure (`sol`, from previous integration) | 1-by-2 cell array

Solution history, specified as a function handle, column vector, `sol` structure (from a previous integration), or a cell array. This is the solution at $t \le t_0$.

- If the history varies with time, specify the solution history as a function handle whose syntax is `y = history(t)`. This function returns an `n`-by-`1` vector that approximates the solution, $y(t)$, for $t <= t_0$. The length of this vector, `n`, is the number of equations in the system you want to solve.
- If $y(t)$ is constant, you can specify `history` as an `n`-by-`1` vector of the constant values.
- If you are calling `ddensd` to continue a previous integration to t_0, you can specify history as the output, `sol`, from the previous integration.
- If you are solving an initial value DDE, specify history as a cell array, `{y0, yp0}`. The first element, `y0`, is a column vector of initial values, $y(t_0)$. The second element, yp_0, is a column vector whose elements are the initial derivatives, $y'(t_0)$. These vectors must be consistent, meaning that they satisfy "Equation 1-1" at t_0. See "Initial Value Neutral Delay Differential Equations" on page 1-3134 for more information.

Data Types: `function_handle` | `single` | `double` | `struct` | `cell`

`tspan` — Interval of integration
1-by-2 vector

Interval of integration, specified as the vector `[t0 tf]`. The first element, `t0`, is the initial value of *t*. The second element, `tf`, is the final value of *t*. The value of `t0` must be less than `tf`.

Data Types: `single` | `double`

`options` — Optional integration parameters
structure returned by `ddeset`

Optional integration parameters, specified as a structure created and returned by the `ddeset` function. Some commonly used properties are: `'RelTol'`, `'AbsTol'`, and `'Events'`. See the `ddeset` reference page for more information about specifying `options`.

Output Arguments

`sol` — Solution
structure

Solution, returned as a structure containing the following fields.

`sol.x`	Mesh selected by `ddensd`.
`sol.y`	An approximation to $y(t)$ at the mesh points.
`sol.yp`	An approximation to $y'(t)$ at the mesh points.
`sol.solver`	A character vector identifying the solver, `'ddensd'`.

You can pass `sol` to the `deval` function to evaluate the solution at specific points. For example, `y = deval(sol, 0.5*(sol.x(1) + sol.x(end)))` evaluates the solution at the midpoint of the interval of integration.

More About

Initial Value Neutral Delay Differential Equations

An initial value DDE has $dy_i \geq t_0$ and $dyp_j \geq t_0$, for all i and j. At $t = t_0$, all delayed terms reduce to $y(dy_i) = y(t_0)$ and $y'(dyp_j) = y'(t_0)$:

$$y'(t_0) = f(t_0, y(t_0), y(t_0),..., y(t_0), y'(t_0),..., y'(t_0)) \tag{1-2}$$

For $t > t_0$, all derivative delays must satisfy $dyp <$ t.

When you solve initial value neutral DDEs, you must supply $y'(t_0)$ to `ddensd`. To do this, specify `history` as a cell array `{Y0,YP0}`. Here, `Y0` is the column vector of initial values, $y(t_0)$, and `YP0` is a column vector of initial derivatives, $y'(t_0)$. These vectors must be consistent, meaning that they satisfy "Equation 1-2" at t_0.

Algorithms

For information about the algorithm used in this solver, see Shampine [2].

References

[1] Paul, C.A.H. "A Test Set of Functional Differential Equations." *Numerical Analysis Reports*. No. 243. Manchester, UK: Math Department, University of Manchester, 1994.

[2] Shampine, L.F. "Dissipative Approximations to Neutral DDEs." *Applied Mathematics & Computation*. Vol. 203, Number 2, 2008, pp. 641–648.

See Also

`dde23` | `ddesd` | `ddeset` | `deval`

Topics

"Create Function Handle"

Introduced in R2012b

ddesd

Solve delay differential equations (DDEs) with general delays

Syntax

```
sol = ddesd(ddefun,delays,history,tspan)
sol = ddesd(ddefun,delays,history,tspan,options)
```

Arguments

`ddefun`	Function handle that evaluates the right side of the differential equations $y'(t) = f(t,y(t),y(d(1),\ldots,y(d(k)))$. The function must have the form `dydt = ddefun(t,y,Z)` where `t` corresponds to the current t, `y` is a column vector that approximates $y(t)$, and `Z(:,j)` approximates $y(d(j))$ for delay $d(j)$ given as component j of `delays(t,y)`. The output is a column vector corresponding to $f(t,y(t),y(d(1),\ldots,y(d(k)))$.
`delays`	Function handle that returns a column vector of delays $d(j)$. The delays can depend on both t and $y(t)$. `ddesd` imposes the requirement that $d(j) \le t$ by using `min`$(d(j),t)$. If all the delay functions have the form $d(j) = t - \tau_j$, you can set the argument `delays` to a constant vector `delays`(j) = τ_j. With delay functions of this form, `ddesd` is used exactly like `dde23`.

`history`	Specify `history` in one of three ways: • A function of t such that `y = history(t)` returns the solution $y(t)$ for $t \le t_0$ as a column vector • A constant column vector, if $y(t)$ is constant • The solution `sol` from a previous integration, if this call continues that integration
`tspan`	Interval of integration from `t0=tspan(1)` to `tf=tspan(end)` with `t0 < tf`.
`options`	Optional integration argument. A structure you create using the `ddeset` function. See `ddeset` for details.

Description

`sol = ddesd(ddefun,delays,history,tspan)` integrates the system of DDEs

$$y'(t) = f(t, y(t), y(d(1)), \ldots, y(d(k)))$$

on the interval $[t_0,t_f]$, where delays $d(j)$ can depend on both t and $y(t)$, and $t_0 < t_f$. Inputs `ddefun` and `delays` are function handles. See “Create Function Handle” for more information.

“Parameterizing Functions” explains how to provide additional parameters to the functions `ddefun`, `delays`, and `history`, if necessary.

`ddesd` returns the solution as a structure `sol`. Use the auxiliary function `deval` and the output `sol` to evaluate the solution at specific points `tint` in the interval `tspan = [t0,tf]`.

```
yint = deval(sol,tint)
```

The structure `sol` returned by `ddesd` has the following fields.

`sol.x`	Mesh selected by `ddesd`
`sol.y`	Approximation to $y(x)$ at the mesh points in `sol.x`.
`sol.yp`	Approximation to $y'(x)$ at the mesh points in `sol.x`
`sol.solver`	Solver name, `'ddesd'`

`sol = ddesd(ddefun,delays,history,tspan,options)` solves as above with default integration properties replaced by values in `options`, an argument created with `ddeset`. See `ddeset` and “Solving Delay Differential Equations” for more information.

Commonly used options are scalar relative error tolerance `'RelTol'` (`1e-3` by default) and vector of absolute error tolerances `'AbsTol'` (all components are `1e-6` by default).

Use the `'Events'` option to specify a function that `ddesd` calls to find where functions $g(t,y(t),y(d(1)),...,y(d(k)))$ vanish. This function must be of the form

```
[value,isterminal,direction] = events(t,y,Z)
```

and contain an event function for each event to be tested. For the `k`th event function in `events`:

- `value(k)` is the value of the `k`th event function.
- `isterminal(k) = 1` if you want the integration to terminate at a zero of this event function and `0` otherwise.
- `direction(k) = 0` if you want `ddesd` to compute all zeros of this event function, `+1` if only zeros where the event function increases, and `-1` if only zeros where the event function decreases.

If you specify the `'Events'` option and events are detected, the output structure `sol` also includes fields:

`sol.xe`	Row vector of locations of all events, i.e., times when an event function vanished
`sol.ye`	Matrix whose columns are the solution values corresponding to times in `sol.xe`
`sol.ie`	Vector containing indices that specify which event occurred at the corresponding time in `sol.xe`

Examples

The equation

```
sol = ddesd(@ddex1de,@ddex1delays,@ddex1hist,[0,5]);
```

solves a DDE on the interval `[0,5]` with delays specified by the function `ddex1delays` and differential equations computed by `ddex1de`. The history is evaluated for $t \le 0$ by the function `ddex1hist`. The solution is evaluated at 100 equally spaced points in `[0,5]`:

```
tint = linspace(0,5);
yint = deval(sol,tint);
```

and plotted with

```
plot(tint,yint);
```

This problem involves constant delays. The `delay` function has the form

```
function d = ddex1delays(t,y)
%DDEX1DELAYS  Delays for using with DDEX1DE.
d = [ t - 1
      t - 0.2];
```

The problem can also be solved with the syntax corresponding to constant delays

```
delays = [1, 0.2];
sol = ddesd(@ddex1de,delays,@ddex1hist,[0, 5]);
```

or using `dde23`:

```
sol = dde23(@ddex1de,delays,@ddex1hist,[0, 5]);
```

For more examples of solving delay differential equations see `ddex2` and `ddex3`.

References

[1] Shampine, L.F., “Solving ODEs and DDEs with Residual Control,” *Applied Numerical Mathematics*, Vol. 52, 2005, pp. 113-127.

See Also

`dde23` | `ddeget` | `ddensd` | `ddeset` | `deval`

Topics

“Create Function Handle”

ddeset

Create or alter delay differential equations options structure

Syntax

```
options = ddeset('name1',value1,'name2',value2,...)
options = ddeset(oldopts,'name1',value1,...)
options = ddeset(oldopts,newopts)
ddeset
```

Description

`options = ddeset('name1',value1,'name2',value2,...)` creates an integrator options structure `options` in which the named properties have the specified values. Any unspecified properties have default values. It is sufficient to type only the leading characters that uniquely identify the property. `ddeset` ignores case for property names.

`options = ddeset(oldopts,'name1',value1,...)` alters an existing options structure `oldopts`. This overwrites any values in `oldopts` that are specified using name/value pairs and returns the modified structure as the output argument.

`options = ddeset(oldopts,newopts)` combines an existing options structure `oldopts` with a new options structure `newopts`. Any values set in `newopts` overwrite the corresponding values in `oldopts`.

`ddeset` with no input arguments displays all property names and their possible values, indicating defaults with braces `{}`.

You can use the function `ddeget` to query the `options` structure for the value of a specific property.

DDE Properties

The following sections describe the properties that you can set using `ddeset`. There are several categories of properties:

- Error control on page 1-3141
- Solver output on page 1-3142
- Step size on page 1-3145
- Event location on page 1-3147
- Discontinuities on page 1-3149

Error Control Properties

At each step, the DDE solvers estimate an error `e`. The `dde23` function estimates the local truncation error, and the other solvers estimate the residual. In either case, this error must be less than or equal to the acceptable error, which is a function of the specified relative tolerance, `RelTol`, and the specified absolute tolerance, `AbsTol`.

```
|e(i)|*max(RelTol*abs(y(i)),AbsTol(i))
```

For routine problems, the solvers deliver accuracy roughly equivalent to the accuracy you request. They deliver less accuracy for problems integrated over “long” intervals and problems that are moderately unstable. Difficult problems may require tighter tolerances than the default values. For relative accuracy, adjust `RelTol`. For the absolute error tolerance, the scaling of the solution components is important: if `|y|` is somewhat smaller than `AbsTol`, the solver is not constrained to obtain any correct digits in `y`. You might have to solve a problem more than once to discover the scale of solution components.

Roughly speaking, this means that you want `RelTol` correct digits in all solution components except those smaller than thresholds `AbsTol(i)`. Even if you are not interested in a component `y(i)` when it is small, you may have to specify `AbsTol(i)` small enough to get some correct digits in `y(i)` so that you can accurately compute more interesting components.

The following table describes the error control properties.

DDE Error Control Properties

Property	Value	Description
`RelTol`	Positive scalar {`1e-3`}	A relative error tolerance that applies to all components of the solution vector `y`. It is a measure of the error relative to the size of each solution component. Roughly, it controls the number of correct digits in all solution components except those smaller than thresholds `AbsTol(i)`. The default, `1e-3`, corresponds to 0.1% accuracy. The estimated error in each integration step satisfies `|e(i)| max(RelTol*abs(y(i)),AbsTol(i))`.
`AbsTol`	Positive scalar or vector {`1e-6`}	Absolute error tolerances that apply to the individual components of the solution vector. `AbsTol(i)` is a threshold below which the value of the `i`th solution component is unimportant. The absolute error tolerances determine the accuracy when the solution approaches zero. Even if you are not interested in a component `y(i)` when it is small, you may have to specify `AbsTol(i)` small enough to get some correct digits in `y(i)` so that you can accurately compute more interesting components. If `AbsTol` is a vector, the length of `AbsTol` must be the same as the length of the solution vector `y`. If `AbsTol` is a scalar, the value applies to all components of `y`.
`NormControl`	`on` | {`off`}	Control error relative to norm of solution. Set this property `on` to request that the solvers control the error in each integration step with `norm(e)<= max(RelTol*norm(y),AbsTol)`. By default, the solvers use a more stringent component-wise error control.

Solver Output Properties

You can use the solver output properties to control the output that the solvers generate.

DDE Solver Output Properties

Property	Value	Description

`OutputFcn`	Function handle
	{`@odeplot`}
		The output function is a function that the solver calls after every successful integration step. To specify an output function, set `'OutputFcn'` to a function handle. For example,
		
		`options = ddeset('OutputFcn',...`
		`@myfun)`
		
		sets '`OutputFcn`' to @myfun, a handle to the function `myfun`. See “Create Function Handle” for more information.
		
		The output function must be of the form
		
		`status = myfun(t,y,flag)`
		
		“Parameterizing Functions” explains how to provide additional parameters to `myfun`, if necessary.
		
		The solver calls the specified output function with the following flags. Note that the syntax of the call differs with the flag. The function must respond appropriately:
		
		• `init` — The solver calls `myfun([tspan(1) tspan(end)],y0,'init')` before beginning the integration to allow the output function to initialize. `tspan` is the input argument to the solvers. `y0` is the initial value of the solution, either from `history(t0)` or specified in the `initialY` option.
		• {none} — The solver calls `status = myfun(t,y)` after each integration step on which output is requested. `t` contains points where output was generated during the step, and `y` is the numerical solution at the points in `t`. If `t` is a vector, the `i`th column of `y` corresponds to the `i`th element of `t`.
		
		`myfun` must return a `status` output value of `0` or `1`. If `status = 1`, the solver halts integration. You can use

Property	Value	Description
		this mechanism, for instance, to implement a **Stop** button. • done — The solver calls `myfun([],[],'done')` when integration is complete to allow the output function to perform any cleanup chores. You can use these general purpose output functions or you can edit them to create your own. Type `help functionname` at the command line for more information. • `odeplot` – time series plotting (default when you call the solver with no output argument and you have not specified an output function) • `odephas2` – two-dimensional phase plane plotting • `odephas3` – three-dimensional phase plane plotting • `odeprint` – print solution as the solver computes it

`OutputSel`	Vector of indices	Vector of indices specifying which components of the solution vector the solvers pass to the output function. For example, if you want to use the `odeplot` output function, but you want to plot only the first and third components of the solution, you can do this using
		
		`options = ddeset...`
		`('OutputFcn',@odeplot,...`
		`'OutputSel',[1 3]);`
		
		By default, the solver passes all components of the solution to the output function.

`Stats`	`on` | `{off}`	Specifies whether the solver should display statistics about its computations. By default, `Stats` is `off`. If it is `on`, after solving the problem the solver displays: • The number of successful steps • The number of failed attempts • The number of times the DDE function was called

Step Size Properties

The step size properties let you specify the size of the first step the solver tries, potentially helping it to better recognize the scale of the problem. In addition, you can specify bounds on the sizes of subsequent time steps.

The following table describes the step size properties.

DDE Step Size Properties

Property	Value	Description
`InitialStep`	Positive scalar	Suggested initial step size. `InitialStep` sets an upper bound on the magnitude of the first step size the solver tries. If you do not set `InitialStep`, the solver bases the initial step size on the slope of the solution at the initial time `tspan(1)`. The initial step size is limited by the shortest delay. If the slope of all solution components is zero, the procedure might try a step size that is much too large. If you know this is happening or you want to be sure that the solver resolves important behavior at the start of the integration, help the code start by providing a suitable `InitialStep`.
`MaxStep`	Positive scalar `{0.1* abs(t0-tf)}`	Upper bound on solver step size. If the differential equation has periodic coefficients or solutions, it may be a good idea to set `MaxStep` to some fraction (such as 1/4) of the period. This guarantees that the solver does not enlarge the time step too much and step over a period of interest. Do *not* reduce `MaxStep`: • When the solution does not appear to be accurate enough. Instead, reduce the relative error tolerance `RelTol`, and use the solution you just computed to determine appropriate values for the absolute error tolerance vector `AbsTol`. (See “Error Control Properties” on page 1-3141 for a description of the error tolerance properties.) • To make sure that the solver doesn't step over some behavior that occurs only once during the simulation interval. If you know the time at which the change occurs, break the simulation interval into two pieces and call the solver twice. If you do not know the time at which the change occurs, try reducing the error tolerances `RelTol` and `AbsTol`. Use `MaxStep` as a last resort.

Event Location Property

In some DDE problems, the times of specific events are important. While solving a problem, the solvers can detect such events by locating transitions to, from, or through zeros of user-defined functions.

The following table describes the `Events` property.

DDE Events Property

Property	Value	Description

`Events`	Function handle	A function handle that includes one or more event functions. For `dde23` and `ddesd`, this function has the following syntax:
		`[value,isterminal,direction] = events(t,y,YDEL)`
		For `ddensd`, the syntax is:
		`[value,isterminal,direction] = events(t,y,YDEL,YPDEL)`
		The output arguments, `value`, `isterminal`, and `direction`, are vectors for which the `i`th element corresponds to the `i`th event function:
		• `value(i)` is the value of the `i`th event function.
		• `isterminal(i) = 1` if you want the integration to terminate at a zero of this event function, and `0` otherwise.
		• `direction(i) = 0` if you want the solver to locate all zeros (the default), `+1` if only zeros where the event function is increasing, and `-1` if only zeros where the event function is decreasing.
		If you specify an events function and events are detected, the solver returns three additional fields in the solution structure `sol`:
		• `sol.xe` is a row vector of times at which events occur.
		• `sol.ye` is a matrix whose columns are the solution values corresponding to times in `sol.xe`.
		• `sol.ie` is a vector containing indices that specify which event occurred at the corresponding time in `sol.xe`.
		For examples that use an event function while solving ordinary differential equation problems, see “ODE Event Location”.

Discontinuity Properties

The solver functions can solve problems with discontinuities in the history or in the coefficients of the equations. The following properties enable you to provide these solvers with a different initial value, and, for `dde23`, locations of known discontinuities. For more information, see “Discontinuities in DDEs” .

The following table describes the discontinuity properties.

DDE Discontinuity Properties

Property	Value	Description
`Jumps`	Vector	Location of discontinuities. Points *t* where the history or solution may have a jump discontinuity in a low-order derivative. This applies only to the `dde23` solver.
`InitialY`	Vector	Initial value of solution. By default the initial value of the solution is the value returned by `history` at the initial point. Supply a different initial value as the value of the `InitialY` property.

Examples

To create an options structure that changes the relative error tolerance of the solver from the default value of `1e-3` to `1e-4`, enter

```
options = ddeset('RelTol',1e-4);
```

To recover the value of `'RelTol'` from `options`, enter

```
ddeget(options,'RelTol')

ans =

  1.0000e-004
```

See Also

`dde23` | `ddeget` | `ddensd` | `ddesd`

Topics

"Create Function Handle"

Introduced before R2006a

deal

Distribute inputs to outputs

Note Beginning with MATLAB® Version 7.0 software, you can access the contents of cell arrays and structure fields without using the `deal` function. See Example 3, below.

Syntax

```
[Y1, Y2, Y3, ...] = deal(X)
[Y1, Y2, Y3, ...] = deal(X1, X2, X3, ...)
[S.field] = deal(X)
[X{:}] = deal(A.field)
[Y1, Y2, Y3, ...] = deal(X{:})
[Y1, Y2, Y3, ...] = deal(S.field)
```

Description

`[Y1, Y2, Y3, ...] = deal(X)` copies the single input to all the requested outputs. It is the same as `Y1 = X`, `Y2 = X`, `Y3 = X`, `...`

`[Y1, Y2, Y3, ...] = deal(X1, X2, X3, ...)` is the same as `Y1 = X1`; `Y2 = X2`; `Y3 = X3`; `...`

Examples

Example 1 — Assign Data From a Cell Array

Use `deal` to copy the contents of a 4-element cell array into four separate output variables.

```
C = {rand(3) ones(3,1) eye(3) zeros(3,1)};
[a,b,c,d] = deal(C{:})
```

```
a =
    0.9501   0.4860   0.4565
    0.2311   0.8913   0.0185
    0.6068   0.7621   0.8214

b =
    1
    1
    1

c =
    1   0   0
    0   1   0
    0   0   1

d =
    0
    0
    0
```

Example 2 — Assign Data From Structure Fields

Use `deal` to obtain the contents of all the name fields in a structure array:

```
A.name = 'Pat'; A.number = 176554;
A(2).name = 'Tony'; A(2).number = 901325;
[name1,name2] = deal(A(:).name)

name1 =
   Pat

name2 =
   Tony
```

Example 3 — Doing the Same Without deal

Beginning with MATLAB Version 7.0 software, you can, in most cases, access the contents of cell arrays and structure fields without using the `deal` function. The two commands shown below perform the same operation as those used in the previous two examples, except that these commands do not require `deal`.

```
[a,b,c,d] = C{:}
[name1,name2] = A(:).name
```

Tips

`deal` is most useful when used with cell arrays and structures via comma-separated list expansion. Here are some useful constructions:

`[S.field] = deal(X)` sets all the fields with the name `field` in the structure array `S` to the value `X`. If `S` doesn't exist, use `[S(1:m).field] = deal(X)`.

`[X{:}] = deal(A.field)` copies the values of the field with name `field` to the cell array `X`. If `X` doesn't exist, use `[X{1:m}] = deal(A.field)`.

`[Y1, Y2, Y3, ...] = deal(X{:})` copies the contents of the cell array `X` to the separate variables `Y1, Y2, Y3, ...`

`[Y1, Y2, Y3, ...] = deal(S.field)` copies the contents of the fields with the name `field` to separate variables `Y1, Y2, Y3, ...`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`cell` | `cell2struct` | `celldisp` | `fieldnames` | `iscell` | `isfield` | `isstruct` | `orderfields` | `rmfield` | `struct` | `struct2cell`

Introduced before R2006a

deblank

Remove trailing whitespace from ends of strings

Syntax

```
newStr = deblank(str)
```

Description

`newStr = deblank(str)` removes trailing whitespace and null characters from `str` and returns the result as `newStr`. However, `deblank` does not remove significant whitespace characters. For example, `deblank` removes trailing space and tab characters, but does not remove the nonbreaking space character, `char(160)`.

Examples

Remove Trailing Blanks from Character Vector

Create a character vector that contains space, tab, and null characters.

```
chr = sprintf(' \t MathWorks \t');
chr = [chr char(0)];
```

Display `chr` between | characters.

```
['|' chr '|']
```

```
ans =
'|      MathWorks      |'
```

Remove the trailing blanks and display `newChr` between | characters.

```
newChr = deblank(chr);
['|' newChr '|']
```

```
ans =
'|      MathWorks|'
```

Convert Character Array and Remove Trailing Spaces

Create a character array that contains multiple pieces of text. When a character array contains multiple rows, it is often necessary to pad the array with space characters.

```
chr = ['Mercury';
       'Apollo ';
       'ISS    ']
```

```
chr = 3x7 char array
    'Mercury'
    'Apollo '
    'ISS    '
```

Convert `chr` to a string array. The elements of `str` include the trailing space characters that padded `chr`.

```
str = string(chr)
```

```
str = 3x1 string array
    "Mercury"
    "Apollo "
    "ISS    "
```

To remove the trailing spaces, use the `deblank` function.

```
newStr = deblank(str)
```

```
newStr = 3x1 string array
    "Mercury"
    "Apollo"
    "ISS"
```

Remove Trailing Blanks from Cell Array

Remove trailing blanks from all the character vectors in a cell array and display them.

```
A = {'MATLAB    ','SIMULINK    ';
     'Toolboxes    ','MathWorks    '}
```

```
A = 2x2 cell array
    {'MATLAB    '   }    {'SIMULINK    ' }
    {'Toolboxes    '}    {'MathWorks    '}
```

```
B = deblank(A)
```

```
B = 2x2 cell array
    {'MATLAB'   }    {'SIMULINK' }
    {'Toolboxes'}    {'MathWorks'}
```

Keep Trailing Nonbreaking Space Character

Create a character vector that includes the nonbreaking space character, `char(160)`, as a trailing blank character.

```
chr = '     MathWorks';
chr = [chr char(160) '     '];
```

Display `chr` between | symbols to show the leading and trailing whitespace.

```
['|' chr '|']
```

```
ans =
'|     MathWorks      |'
```

Remove the trailing whitespace characters.

```
newChr = deblank(chr);
```

Display `newChr` between | symbols. `deblank` removes the trailing space characters, but leaves the nonbreaking space at the end of `newChr`.

```
['|' newChr '|']
```

```
ans =
'|     MathWorks |'
```

Input Arguments

`str` — Input text
string array | character array | cell array of character arrays

Input text, specified as a string array, a character array, or as a cell array of character arrays.

Algorithms

`deblank` does not remove significant whitespace characters.

This table shows the most common characters that are significant whitespace characters and their descriptions. For more information, see Whitespace character.

Significant Whitespace Character	Description
`char(133)`	Next line
`char(160)`	Nonbreaking space
`char(8199)`	Figure space
`char(8239)`	Narrow no-break space

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Input text must be specified as a character array. Cell arrays are not supported.
- Input values must be in the range 0–127.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- `str` must be a string array or a cell array of character vectors.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cellstr` | `isspace` | `strjust` | `strtrim`

Introduced before R2006a

dec2base

Convert decimal number to character array representing base-N number

Syntax

```
baseStr = dec2base(D,n)
baseStr = dec2base(D,n,minDigits)
```

Description

`baseStr = dec2base(D,n)` returns a base-`n` representation of the number `D`. The output argument `baseStr` is a character array that represents digits using numeric characters and, when `n` is greater than 10, letters. For example, if `n` is 12, then `dec2base` represents the numbers 9, 10, and 11 using the characters `'9'`, `'A'`, and `'B'`, and represents the number 12 as the character sequence `'10'`.

If `D` is a numeric vector, matrix, or multidimensional array, then `baseStr` is a two-dimensional character array. Each row of `baseStr` represents an element of `D`.

`baseStr = dec2base(D,n,minDigits)` returns a base-`n` representation of `D` with no fewer than `minDigits` digits.

Examples

Convert Decimal Number

Convert a decimal number to a character vector that represents its value in base 12. In this base system, the characters `'A'` and `'B'` represent the numbers denoted as 10 and 11 in base 10.

```
D = 23;
baseStr = dec2base(D,12)
```

```
baseStr =
'1B'
```

Specify Number of Digits

Specify the number of base-12 digits that `dec2base` returns. If you specify more digits than are required, then `dec2base` pads the output with leading zeros.

```
D = 23;
baseStr = dec2base(D,12,6)
```

```
baseStr =
'00001B'
```

If you specify fewer digits, then `dec2base` still returns as many digits as required to represent the input number.

```
baseStr = dec2base(D,12,1)
```

```
baseStr =
'1B'
```

Convert Numeric Array to Octal Values

Create a numeric array.

```
D = [1023 122 14];
```

To represent the elements of `D` as octal, or base-8, values, use the `dec2base` function. Each row of `baseStr` corresponds to an element of `D`.

```
baseStr = dec2base(D,8)
```

```
baseStr = 3x4 char array
    '1777'
    '0172'
    '0016'
```

The `dec2base` function returns a character array padded with leading zeros. Starting in R2016b, the `compose` function is recommended for converting numeric arrays to octal

representations. It returns a string array whose elements do not have leading zeros. To represent the elements of D as octal values, use the %o formatting operator.

```
hexStr = compose("%o",D)
```

```
hexStr = 1x3 string array
    "1777"    "172"    "16"
```

Input Arguments

D — Input array
array of nonnegative numbers

Input array, specified as an array of nonnegative numbers. Each element of D must have a value between zero and the value returned by flintmax.

- If D is an array of floating-point numbers, and any element of D has a fractional part, then dec2base produces an error. For example, dec2base(10,8) converts 10 to '12', but dec2base(10.5,8) produces an error.
- If D is a character or logical array, then dec2base treats the elements of D as integers. However, dec2base treats characters as their Unicode values, so specifying D as a character array is not recommended.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char

n — Base of output representation
integer between 2 and 36

Base of output representation, specified as an integer between 2 and 36.

minDigits — Minimum number of digits in output
integer

Minimum number of digits in output, specified as an integer.

- If D can be represented with fewer than minDigits digits, then dec2base pads the output with leading zeros.

- If `D` is so large that it must be represented with more than `minDigits` digits, then `dec2base` returns the output with as many digits as required.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`base2dec` | `compose` | `dec2bin` | `dec2hex` | `flintmax`

Topics

"Hexadecimal and Binary Values"

Introduced before R2006a

dec2bin

Convert decimal number to character array representing binary number

Syntax

```
binStr = dec2bin(D)
binStr = dec2bin(D,minDigits)
```

Description

`binStr = dec2bin(D)` returns the binary, or base-2, representation of the number `D`. The output argument `binStr` is a character vector that represents binary digits using the characters `'0'` and `'1'`.

If `D` is a numeric vector, matrix, or multidimensional array, then `binStr` is a two-dimensional character array. Each row of `binStr` represents an element of `D`.

`binStr = dec2bin(D,minDigits)` returns a binary representation with no fewer than `minDigits` digits.

Examples

Convert Decimal Number

Convert a decimal number to a character vector that represents its binary value.

```
D = 23;
binStr = dec2bin(D)
```

```
binStr =
'10111'
```

Specify Number of Digits

Specify the number of binary digits that `dec2bin` returns. If you specify more digits are required, then `dec2bin` pads the output with leading zeros.

```
D = 23;
binStr = dec2bin(D,8)
```

```
binStr =
'00010111'
```

If you specify fewer digits, then `dec2bin` still returns as many binary digits as required to represent the input number.

```
binStr = dec2bin(D,1)
```

```
binStr =
'10111'
```

Convert Numeric Array

Create a numeric array.

```
D = [1023 122 14];
```

To represent the elements of `D` as binary values, use the `dec2bin` function. Each row of `binStr` corresponds to an element of `D`.

```
binStr = dec2bin(D)
```

```
binStr = 3x10 char array
    '1111111111'
    '0001111010'
    '0000001110'
```

Since all rows of a character array must have the same number of characters, `dec2bin` pads some rows of `binStr` with leading zeros. For example, the number `14` can be represented by the binary digits `'1110'`. But to match the length of the first row of `binStr`, the `dec2bin` function pads the third row to `'0000001110'`.

Input Arguments

`D` — Input array

array of nonnegative numbers

Input array, specified as an array of nonnegative numbers.

- If `D` is an array of floating-point numbers, and any element of `D` has a fractional part, then `dec2bin` truncates it before conversion. For example, `dec2bin` converts both `12` and `12.5` to `'1100'`. The truncation is always to the nearest integer less than or equal to that element.
- If `D` is a character or logical array, then `dec2bin` treats the elements of `D` as integers. However, `dec2bin` treats characters as their Unicode values, so specifying `D` as a character array is not recommended.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`

`minDigits` — Minimum number of digits in output

integer

Minimum number of digits in the output, specified as an integer.

- If `D` can be represented with fewer than `minDigits` binary digits, then `dec2bin` pads the output with leading zeros.
- If `D` is so large that it must be represented with more than `minDigits` digits, then `dec2bin` returns the output with as many digits as required.

Tips

- If the input argument `D` has a value greater than the value returned by `flintmax`, then `dec2bin` might not return an exact representation.
- The output of `dec2bin` is the same whether your computer stores values in memory using big-endian or little-endian format. For more information on these formats, see Endianness.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If input `D` is `double`, then it must be less than `2^52`.
- If input `D` is `single`, then it must be less than `2^23`.
- Unless you specify input `minDigits` to be constant and `minDigits` is large enough that the output has a fixed number of columns regardless of the input values, this function requires variable-sizing support. Without variable-sizing support, `minDigits` must be at least `52` for `double`, `23` for `single`, `16` for `char`, `32` for `int32`, `16` for `int16`, and so on.

See Also

`bin2dec` | `dec2base` | `dec2hex` | `flintmax`

Topics

"Hexadecimal and Binary Values"

Introduced before R2006a

dec2hex

Convert decimal number to character array representing hexadecimal number

Syntax

```
hexStr = dec2hex(D)
hexStr = dec2hex(D,minDigits)
```

Description

`hexStr = dec2hex(D)` returns the hexadecimal, or base-16, representation of the number `D`. The output argument `hexStr` is a character vector that represents hexadecimal digits using the characters `0`-`9` and `A`-`F`.

If `D` is a numeric vector, matrix, or multidimensional array, then `hexStr` is a two-dimensional character array. Each row of `hexStr` represents an element of `D`.

`hexStr = dec2hex(D,minDigits)` returns a hexadecimal representation with no fewer than `minDigits` digits.

Examples

Convert Decimal Number

Convert a decimal number to a character vector that represents its hexadecimal value.

```
D = 1023;
hexStr = dec2hex(D)
```

```
hexStr =
'3FF'
```

Specify Number of Digits

Specify the number of hexadecimal digits that `dec2hex` returns. If you specify more digits than are required, then `dec2hex` pads the output with leading zeros.

```
D = 1023;
hexStr = dec2hex(D,6)
```

```
hexStr =
'0003FF'
```

If you specify fewer digits, then `dec2hex` still returns as many hexadecimal digits as required to represent the input number.

```
hexStr = dec2hex(D,1)
```

```
hexStr =
'3FF'
```

Convert Numeric Array

Create a numeric array.

```
D = [1023 122 14];
```

To represent the elements of `D` as hexadecimal values, use the `dec2hex` function. Each row of `hexStr` corresponds to an element of `D`.

```
hexStr = dec2hex(D)
```

```
hexStr = 3x3 char array
    '3FF'
    '07A'
    '00E'
```

The `dec2hex` function returns a character array padded with leading zeros. Starting in R2016b, the `compose` function is recommended for converting numeric arrays to hexadecimal representations. It returns a string array whose elements do not have leading zeros. To represent the elements of `D` as hexadecimal values, use either the `%X` or `%x` formatting operator.

```
hexStr = compose("%X",D)
```

```
hexStr = 1x3 string array
    "3FF"    "7A"    "E"
```

Input Arguments

D — Input array
array of nonnegative numbers

Input array, specified as an array of nonnegative numbers.

- If D is an array of floating-point numbers, and any element of D has a fractional part, then dec2hex produces an error. For example, dec2hex converts 10 to 'A', but does not convert 10.5.
- If D is a character or logical array, then dec2hex treats the elements of D as integers. However, dec2hex treats characters as their Unicode values, so specifying D as a character array is not recommended.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char

minDigits — Minimum number of digits in output
integer

Minimum number of digits in the output, specified as an integer.

- If D can be represented with fewer than minDigits hexadecimal digits, then dec2hex pads the output with leading zeros.
- If D is so large that it must be represented with more than minDigits digits, then dec2hex returns the output with as many digits as required.

Tips

- If the input argument D has a value greater than the value returned by flintmax, then dec2hex might not return an exact representation.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If input `D` is `double`, then `D` must be less than `2^52`.
- If input `D` is `single`, then `D` must be less than `2^23`.
- Unless you specify input `minDigits` to be constant, and `minDigits` is large enough that the output has a fixed number of columns regardless of the input values, this function requires variable-sizing support. Without variable-sizing support, `minDigits` must be at least `13` for `double`, `6` for `single`, `4` for `char`, `8` for `int32`, `4` for `int16`, and so on.

See Also

`compose` | `dec2base` | `dec2bin` | `flintmax` | `format` | `hex2dec` | `hex2num`

Topics

"Hexadecimal and Binary Values"

Introduced before R2006a

decic

Compute consistent initial conditions for `ode15i`

Syntax

```
[y0_new,yp0_new] = decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0)
[y0_new,yp0_new] = decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0,
options)
[y0_new,yp0_new,resnrm] = decic( ___ )
```

Description

`[y0_new,yp0_new] = decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0)` uses `y0` and `yp0` as guesses for the initial conditions of the fully implicit function `odefun`, holds the components specified by `fixed_y0` and `fixed_yp0` as fixed, then computes values for the nonfixed components. The result is a complete set of consistent initial conditions. The new values `yo_new` and `yp0_new` satisfy `odefun(t0,y0_new,yp0_new) = 0` and are suitable to be used as initial conditions with `ode15i`.

`[y0_new,yp0_new] = decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0, options)` also uses the options structure `options` to specify values for `AbsTol` and `RelTol`. Create the options structure using `odeset`.

`[y0_new,yp0_new,resnrm] = decic( ___ )` returns the norm of `odefun(t0,y0_new,yp0_new)` as `resnrm`. If the norm seems unduly large, then use `options` to decrease the relative error tolerance `RelTol`, which has a default value of `1e-3`.

Examples

Compute Consistent Initial Conditions for Implicit Equations

Consider the implicit system of equations

$$0 = 2y_1' - y_2$$
$$0 = y_1 + y_2$$

These equations are straightforward enough that it is simple to read off consistent initial conditions for the variables. For example, if you fix $y_1 = 1$, then $y_2 = -1$ according to the second equation and $y_1' = -1/2$ according to the first equation. Since these values of y_1, y_1', and y_2 satisfy the equations, they are consistent.

Confirm these values by using `decic` to compute consistent initial conditions for the equations, fixing the value $y_1 = 1$. Use guesses of `y0 = [1 0]` and `yp0 = [0 0]`, which do not satisfy the equations and are thus inconsistent.

```
odefun = @(t,y,yp) [2*yp(1)-y(2); y(1)+y(2)];
t0 = 0;
y0 = [1 0];
yp0 = [0 0];
[y0,yp0] = decic(odefun,t0,y0,[1 0],yp0,[])
```

```
y0 = 2×1

     1
    -1


yp0 = 2×1

   -0.5000
         0
```

Solve Weissinger Implicit ODE

Calculate consistent initial conditions and solve an implicit ODE with `ode15i`.

Weissinger's equation is

$$ty^2(y')^3 - y^3(y')^2 + t\left(t^2 + 1\right)y' - t^2y = 0.$$

Since the equation is in the generic form $f(t, y, y') = 0$, you can use the `ode15i` function to solve the implicit differential equation.

Code Equation

To code the equation in a form suitable for `ode15i`, you need to write a function with inputs for t, y, and y' that returns the residual value of the equation. The function `@weissinger` encodes this equation. View the function file.

```
type weissinger
```

```
function res = weissinger(t,y,yp)
%WEISSINGER  Evaluate the residual of the Weissinger implicit ODE
%
%   See also ODE15I.

%   Jacek Kierzenka and Lawrence F. Shampine
%   Copyright 1984-2014 The MathWorks, Inc.

res = t*y^2 * yp^3 - y^3 * yp^2 + t*(t^2 + 1)*yp - t^2 * y;
```

Calculate Consistent Initial Conditions

The `ode15i` solver requires *consistent* initial conditions, that is, the initial conditions supplied to the solver must satisfy

$$f(t_0, y, y') = 0.$$

Since it is possible to supply inconsistent initial conditions, and `ode15i` does not check for consistency, it is recommended that you use the helper function `decic` to compute such conditions. `decic` holds some specified variables fixed and computes consistent initial values for the unfixed variables.

In this case, fix the initial value $y(t_0) = \sqrt{\frac{3}{2}}$ and let `decic` compute a consistent initial value for the derivative $y'(t_0)$, starting from an initial guess of $y'(t_0) = 0$.

```
t0 = 1;
y0 = sqrt(3/2);
yp0 = 0;
[y0,yp0] = decic(@weissinger,t0,y0,1,yp0,0)
```

```
y0 = 1.2247

yp0 = 0.8165
```

Solve Equation

Use the consistent initial conditions returned by `decic` with `ode15i` to solve the ODE over the time interval [1 10].

```
[t,y] = ode15i(@weissinger,[1 10],y0,yp0);
```

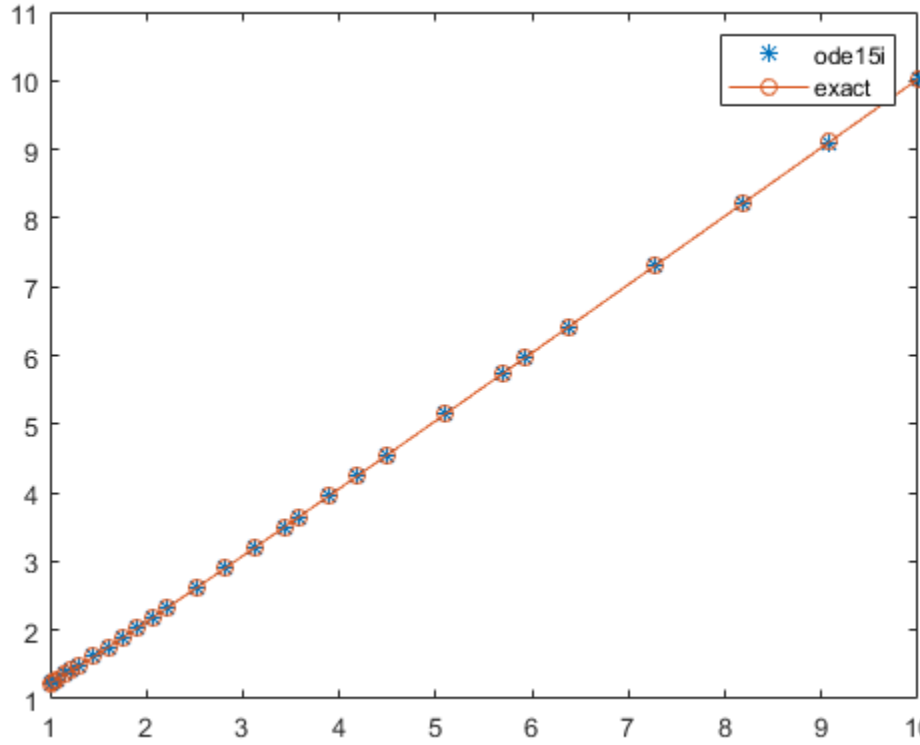
Plot Results

The exact solution of this ODE is

$$y(t) = \sqrt{t^2 + \frac{1}{2}}.$$

Plot the numerical solution `y` computed by `ode15i` against the analytical solution `ytrue`.

```
ytrue = sqrt(t.^2 + 0.5);
plot(t,y,'*',t,ytrue,'-o')
legend('ode15i', 'exact')
```



Input Arguments

odefun — Functions to solve
function handle

Functions to solve, specified as a function handle that defines the functions to be integrated. `odefun` represents the system of implicit differential equations that you want to solve using `ode15i`.

The function `f = odefun(t,y,yp)`, for a scalar `t` and column vectors `y` and `yp`, must return a column vector `f` of data type `single` or `double` that corresponds to $f(t, y, y')$.

odefun must accept all three input arguments, t, y, and yp even if one of the arguments is not used in the function.

For example, to solve $y' - y = 0$, use this function.

```
function f = odefun(t,y,yp)
f = yp - y;
```

For a system of equations, the output of odefun is a vector. Each equation becomes an element in the solution vector. For example, to solve

$$y'_1 - y_2 = 0$$
$$y'_2 + 1 = 0\,,$$

use this function.

```
function dy = odefun(t,y,yp)
dy = zeros(2,1);
dy(1) = yp(1)-y(2);
dy(2) = yp(2)+1;
```

For information on how to provide additional parameters to the function odefun, see "Parameterizing Functions".

Example: @myFcn

Data Types: function_handle

t0 — Initial time
scalar

Initial time, specified as a scalar. decic uses the initial time to compute consistent initial conditions that satisfy odefun(t0,y0_new,yp0_new) = 0.

Data Types: single | double

y0 — Initial guesses for y-components
vector

Initial guesses for y-components, specified as a vector. Each element in y0 specifies an initial condition for one dependent variable y_n in the system of equations defined by odefun.

Data Types: single | double

`fixed_y0` — y-components to hold fixed

vector of 1s and 0s | `[]`

`y`-components to hold fixed, specified as a vector of 1s and 0s, or as `[]`.

- Set `fixed_y0(i) = 1` if no change is permitted in the guess for `y0(i)`.
- Set `fixed_y0 = []` if any entry can be changed.

You cannot fix more than `length(yp0)` components. Depending on the specific problem, it is not always possible to fix certain components of `y0` or `yp0`. It is a best practice not to fix more components than is necessary.

`yp0` — Initial guesses for `y'`-components

vector

Initial guesses for `y'`-components, specified as a vector. Each element in `yp0` specifies an initial condition for one differentiated dependent variable y'_n in the system of equations defined by `odefun`.

Data Types: `single` | `double`

`fixed_yp0` — `y'`-components to hold fixed

vector of 1s and 0s | `[]`

`y'`-components to hold fixed, specified as a vector of 1s and 0s, or as `[]`.

- Set `fixed_yp0(i) = 1` if no change is permitted in the guess for `yp0(i)`.
- Set `fixed_yp0 = []` if any entry can be changed.

You cannot fix more than `length(yp0)` components. Depending on the specific problem, it is not always possible to fix certain components of `y0` or `yp0`. It is a best practice not to fix more components than is necessary.

`options` — Options structure

structure array

Options structure, specified as a structure array. Use the `odeset` function to create or modify the options structure. The relevant options for use with the `decic` function are `RelTol` and `AbsTol`, which control the error thresholds used to compute the initial conditions.

Example: `options = odeset('RelTol',1e-5)`

Data Types: `struct`

Output Arguments

`y0_new` — Consistent initial conditions for `y0`
vector

Consistent initial conditions for `y0`, returned as a vector. If the value of `resnrm` is small, then `yo_new` and `yp0_new` satisfy `odefun(t0,y0_new,yp0_new) = 0` and are suitable to be used as initial conditions with `ode15i`.

`yp0_new` — Consistent initial conditions for `yp0`
vector

Consistent initial conditions for `yp0`, returned as a vector. If the value of `resnrm` is small, then `yo_new` and `yp0_new` satisfy `odefun(t0,y0_new,yp0_new) = 0` and are suitable to be used as initial conditions with `ode15i`.

`resnrm` — Norm of residual
vector

Norm of residual, returned as a vector. `resnrm` is the norm of `odefun(t0,y0_new,yp0_new)`.

- A small value of `resnrm` indicates that `decic` successfully computed consistent initial conditions that satisfy `odefun(t0,y0_new,yp0_new) = 0`.
- If the value of `resnrm` is large, try adjusting the error thresholds `RelTol` and `AbsTol` using the `options` input.

Tips

- The `ihb1dae` and `iburgersode` example files use `decic` to compute consistent initial conditions before solving with `ode15i`. Type `edit ihb1dae` or `edit iburgersode` to view the code.
- You can additionally use `decic` to compute consistent initial conditions for DAEs solved by `ode15s` or `ode23t`. To do this, follow these steps.

  1. Rewrite the system of equations in fully implicit form `f(t,y,y') = 0`.

2. Call `decic` to compute consistent initial conditions for the equations.
3. Specify `y0_new` as the initial condition in the call to the solver, and specify `yp_new` as the value of the `InitialSlope` option of `odeset`.

See Also

`ode15i` | `odeget` | `odeset`

Introduced before R2006a

decomposition

Matrix decomposition for solving linear systems

Description

`decomposition` creates reusable matrix decompositions (LU, LDL, Cholesky, QR, and more) that enable you to solve linear systems ($Ax = b$ or $xA = b$) more efficiently. For example, after computing `dA = decomposition(A)` the call `dA\b` returns the same vector as `A\b`, but is typically much faster. `decomposition` objects are well-suited to solving problems that require repeated solutions, since the decomposition of the coefficient matrix does not need to be performed multiple times.

You can use a `decomposition` object `dA` with many of the same operators you might use on the original coefficient matrix `A`:

- Complex conjugate transpose `dA'`
- Negation `-dA`
- Multiply or divide by a scalar using `c*dA` or `dA/c`.
- Solve a linear system $Ax = b$ using `x = dA\b`.
- Solve a linear system $xA = b$ using `x = b/dA`.

Creation

Syntax

```
dA = decomposition(A)
dA = decomposition(A,type)
dA = decomposition(A,type,triangularFlag)
dA = decomposition( ___ ,Name,Value)
```

Description

`dA = decomposition(A)` returns a decomposition of matrix `A` that you can use to solve linear systems more efficiently. The decomposition type is automatically chosen based on the properties of the input matrix.

`dA = decomposition(A,type)` specifies the type of decomposition to perform. `type` can be `'qr'`, `'cod'`, `'lu'`, `'ldl'`, `'chol'`, `'triangular'`, `'permutedTriangular'`, `'banded'`, `'hessenberg'`, or `'diagonal'`.

`dA = decomposition(A,type,triangularFlag)` specifies that only the upper or lower triangular portion of `A` is to be used in the decomposition. `triangularFlag` can be `'upper'` or `'lower'`. With this syntax, the decomposition type must be `'ldl'`, `'chol'`, or `'triangular'`.

`dA = decomposition( ___ ,Name,Value)` specifies additional options using one or more `Name,Value` pair arguments using any of the previous syntaxes. For example, `dA = decomposition(A,'CheckCondition',false)` specifies not to throw a warning based on the condition of `A` while solving `dA\b`.

Input Arguments

`A` — Coefficient matrix

matrix

Coefficient matrix. The coefficient matrix appears in the system of linear equations on the left as $Ax = b$ or on the right as $xA = b$.

Data Types: `single` | `double`
Complex Number Support: Yes

`type` — Decomposition type

`'auto'` (default) | `'qr'` | `'cod'` | `'lu'` | `'ldl'` | `'chol'` | `'triangular'` | `'permutedTriangular'` | `'banded'` | `'hessenberg'` | `'diagonal'`

Decomposition type, specified as one of the options in these tables.

These options work for **any** coefficient matrix.

Value	Matrix Decomposition of `A`	Notes
`'auto'` (default)	N/A	Automatic selection of the matrix decomposition type based on the properties of the coefficient matrix. For information about how the decomposition is chosen, see the “Algorithms” on page 1-9862 section of `mldivide`.
`'qr'`	$AP = QR$ `Q` is unitary, `R` is upper triangular, and `P` is a permutation matrix.	QR decompositions give a least-squares solution. If `type` is `'qr'`, then you cannot solve `A'\B` or `B/A`. Instead, use `'cod'` for problems with those forms.
`'cod'`	$A = QRZ^*$ `R` is upper triangular, and both `Q` and `Z` have orthonormal columns.	Complete orthogonal decompositions give a minimum norm least-squares solution.

For **square** coefficient matrices, you also can use these options.

Value	Matrix Decomposition of `A`	Notes
`'lu'`	**Dense matrices:** $PA = LU$ `L` is lower triangular, `U` is upper triangular, and `P` is a permutation matrix. **Sparse matrices:** $P(R\backslash A)Q = LU$ `P` and `Q` are permutation matrices and `R` is a diagonal scaling matrix.	
`'ldl'`	**Dense matrices:** $P^*AP = LDL^*$ `L` is a lower triangular matrix with 1s on the diagonal, `D` is a diagonal matrix, and `P` is a permutation matrix. **Sparse matrices:** $P^*SASP = LDL^*$ `S` is a scaling matrix.	`A` must be symmetric.

Value	Matrix Decomposition of A	Notes
'chol'	**Dense matrices:** $A = LL^*$ L is lower triangular. **Sparse matrices:** $A = PLL^*P^*$ P is a permutation matrix.	A must be symmetric positive definite.
'triangular'	$A = T$ T is triangular.	A must be triangular.
'permutedTriangular'	$A = PTQ^*$ T is triangular, and both P and Q are permutation matrices.	A must be a permutation of a triangular matrix.
'banded'	$A = P^*LU$ P is a permutation matrix and both L and U are banded.	Most effective for matrices with a low bandwidth. See bandwidth for more information.
'hessenberg'	$A = P^*LU$ P is a permutation matrix, L is banded, and U is upper triangular.	A must have zeros below the first subdiagonal.
'diagonal'	$A = D$ D is diagonal.	A must be diagonal.

triangularFlag — Flag to use only upper or lower triangular portion of coefficient matrix
'upper' | 'lower'

Flag to use only upper or lower triangular portion of coefficient matrix, specified as either `'upper'` or `'lower'`. This option supports the `'triangular'`, `'chol'`, and `'ldl'` decomposition types.

- `'triangular'` — If both an upper and lower triangular matrix are stored in the same matrix, then use `triangularFlag` to specify only one of the triangular matrices.
- `'chol'` and `'ldl'` — Use `triangularFlag` to avoid symmetrizing a nearly symmetric coefficient matrix.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `dA = decomposition(A,'qr','CheckCondition',false)` performs a QR decomposition of `A` and turns off warnings about the condition of the coefficient matrix when it is used to solve a linear system.

General Parameters

`CheckCondition` — Toggle to check condition of coefficient matrix
`true` (default) | `false`

Toggle to check condition of coefficient matrix, specified as the comma-separated pair consisting of `'CheckCondition'` and either logical `1` (`true`) or logical `0` (`false`). If `CheckCondition` is `true` and the coefficient matrix is badly conditioned or of low rank, then solving linear systems using `mldivide (\)` or `mrdivide (/)` produces warnings.

Data Types: `logical`

`RankTolerance` — Rank tolerance
nonnegative scalar

Rank tolerance, specified as a nonnegative scalar. Specifying the tolerance can help prevent the solution from being susceptible to random noise in the coefficient matrix.

`decomposition` computes the rank of `A` as the number of diagonal elements in the `R` matrix of the QR decomposition `[Q,R,p] = qr(A,0)` with absolute value larger than `tol`. If the rank of `A` is `k`, then a low-rank approximation of `A` is formed by multiplying the first `k` columns of `Q` by the first `k` rows of `R`. Changing the tolerance affects this low-rank approximation of `A`.

Note This option only applies when `'Type'` is `'qr'` or `'cod'`, or when `'Type'` is `'auto'` and `A` is rectangular. Otherwise, this option is ignored.

Sparse Matrix Parameters

`BandDensity` — Band density threshold
`0.5` (default) | scalar

Band density threshold, specified as a scalar value in the range `[0 1]`. The value of `'BandDensity'` determines how dense a sparse, banded coefficient matrix must be for the banded solver to be used by `mldivide (\)` or `mrdivide (/)` when solving a system of equations. If the band density of the coefficient matrix is larger than the specified band density, then the banded solver is used.

The band density is defined as: (# nonzeros in the band) / (# elements in the band). A value of `1.0` indicates to never use the banded solver.

`LDLPivotTolerance` — Pivot tolerance for LDL factorization
`0.01` (default) | scalar

Pivot tolerance for LDL factorization, specified as a scalar value in the interval `[0 0.5]`. Using smaller values of the pivot tolerance can give faster factorization times and fewer entries, but also can result in a less stable factorization.

This pivot tolerance is the same that `ldl` uses for real sparse matrices.

`LUPivotTolerance` — Pivot tolerance for LU factorization
`[0.1 0.001]` (default) | scalar | vector

Pivot tolerance for LU factorization, specified as a scalar or vector. Specify a scalar value to change the first element in the tolerance vector, or specify a two-element vector to change both values. Smaller pivot tolerances tend to lead to sparser LU factors, but the solution can become inaccurate. Larger values can lead to a more accurate solution, but not always, and usually increase the total work and memory usage.

This pivot tolerance is the same that `lu` uses for sparse matrices.

Properties

`MatrixSize` — Size of coefficient matrix
vector

This property is read-only.

Size of coefficient matrix, returned as a two-element row vector.

Data Types: `double`

`Type` — Decomposition type
`'qr'` | `'cod'` | `'lu'` | `'ldl'` | `'chol'` | `'triangular'` | `'permutedTriangular'` | `'banded'` | `'hessenberg'` | `'diagonal'`

This property is read-only.

Decomposition type, returned as `'qr'`, `'cod'`, `'lu'`, `'ldl'`, `'chol'`, `'triangular'`, `'permutedTriangular'`, `'banded'`, `'hessenberg'`, or `'diagonal'`.

Data Types: `char`

`CheckCondition` — Toggle to check condition of coefficient matrix
`true` (default) | `false`

Toggle to check condition of coefficient matrix, specified as either logical `1` (`true`) or logical `0` (`false`). If `CheckCondition` is `true` and the coefficient matrix is badly conditioned or of low rank, then solving linear systems using `mldivide (\)` or `mrdivide (/)` produces warnings.

Data Types: `logical`

`Datatype` — Data type of coefficient matrix
`'double'` | `'single'`

This property is read-only.

Data type of coefficient matrix, returned as either `'double'` or `'single'`.

Data Types: `char`

`IsConjugateTransposed` — Indicator that coefficient matrix is complex conjugate transposed
`false` (default) | `true`

This property is read-only.

Indicator that coefficient matrix is complex conjugate transposed, returned as either logical `1` (`true`) or logical `0` (`false`). This indicator is `false` by default for any `decomposition` object constructed from the coefficient matrix. However, the value is `true` if you use the `ctranspose` operator on a `decomposition` object in an expression, such as `dA'\b`. In that case, `dA'` is the same `decomposition` object as `dA`, but with a value of `true` for `IsConjugateTransposed`.

Data Types: `logical`

`IsReal` — Indicator that coefficient matrix is real
`true` | `false`

This property is read-only.

Indicator that coefficient matrix is real, returned as either logical `1` (`true`) or logical `0` (`false`). A value of `false` indicates that the coefficient matrix contains complex numbers.

Data Types: `logical`

`IsSparse` — Indicator that coefficient matrix is sparse
`true` | `false`

This property is read-only.

Indicator that coefficient matrix is sparse, returned as either logical `1` (`true`) or logical `0` (`false`).

Data Types: `logical`

`ScaleFactor` — Multiplicative scale factor for coefficient matrix
`1` (default) | scalar

This property is read-only.

Multiplicative scale factor for coefficient matrix, returned as a scalar. The default value of `1` indicates that the coefficient matrix is not scaled. However, when you multiply or divide the `decomposition` object by a scalar, the value of `ScaleFactor` changes. For example, `3*dA` is a `decomposition` object equivalent to `dA`, but with a value of `3` for `ScaleFactor`.

Data Types: `double`

Complex Number Support: Yes

Object Functions

The primary functions and operators that you can use with `decomposition` objects are related to solving linear systems of equations.

ctranspose	Complex conjugate transpose
mldivide	Solve systems of linear equations Ax = B for x
mrdivide	Solve systems of linear equations xA = B for x
isIllConditioned	Determine whether matrix is ill conditioned

You also can check the condition number or rank of the underlying matrix of `decomposition` objects. Since different algorithms are employed, the results of using these functions on the `decomposition` object can differ compared to using the same functions directly on the coefficient matrix.

`rank`	• Only the 1-input form `rank(dA)` is supported. • The decomposition type must be `'qr'` or `'cod'`. • The value of the rank depends on the choice of `RankTolerance`, if specified.
`rcond`	• Runs the same condition check that backslash \ uses to determine whether to issue a warning. • Supports all decomposition types except for `'qr'` and `'cod'`.

Examples

Solve Linear System with Several Right-Hand Sides

Show how using `decomposition` objects can improve the efficiency of solving Ax = *b* with many right-hand sides.

The *inverse iteration* is an iterative eigenvalue algorithm that solves linear systems with many right-hand sides. It is a method to iteratively compute an eigenvalue of a matrix

starting from a guess of the corresponding eigenvector. Each iteration computes `x = A \x`, and then scales `x` by its norm.

Create a sparse matrix `A` and random starting vectors `x1` and `x2`.

```
n = 1e3;
rng default % for reproducibility
A = sprandn(n,n,0.2) + speye(n);
x1 = randn(n,1);
x2 = x1;
```

Apply 100 iterations of the inverse iteration algorithm using backslash to calculate an eigenvalue of `A`.

```
tic
for ii=1:100
    x1 = A \ x1;
    x1 = x1 / norm(x1);
end
toc
```

```
Elapsed time is 26.783357 seconds.
```

```
lambda = x1'*A*x1
```

```
lambda = -0.6707
```

Now use a `decomposition` object to solve the same problem.

```
tic
dA = decomposition(A);
for ii=1:100
    x2 = dA \ x2;
    x2 = x2 / norm(x2);
end
toc
```

```
Elapsed time is 1.062422 seconds.
```

```
lambda = x2'*A*x2
```

```
lambda = -0.6707
```

The performance of the algorithm improves dramatically because the matrix `A` does not need to be factorized during each iteration. Also, even though the backslash algorithm can be improved by performing an LU decomposition of `A` before the `for`-loop, the

`decomposition` object gives access to all of the same performance gains without requiring that you write complex code.

Select Decomposition Type

Choose a decomposition type to override the automatic default selection based on the input matrix.

Create a coefficient matrix and decompose the matrix using the default selection of decomposition type.

```
A = ones(3);
dA = decomposition(A)
```

```
dA = 
  decomposition with properties:

    MatrixSize: [3 3]
          Type: 'ldl'

  Show all properties
```

Solve the linear system using a vector of ones for the right-hand side.

```
b = ones(3,1);
x = dA\b
```

```
Warning: Matrix is singular to working precision.
```

```
x = 3×1

   NaN
   NaN
   NaN
```

Specify the decomposition type to use the `'qr'` method instead of the default `'ldl'` method. This forces backslash (\) to find a least-squares solution to the problem instead of returning a vector of `NaN`s.

```
dA_qr = decomposition(A,'qr')
```

```
dA_qr =
  decomposition with properties:

    MatrixSize: [3 3]
          Type: 'qr'

  Show all properties
```

```
x = dA_qr\b
```

```
Warning: Rank deficient, rank = 1, tol =  1.153778e-15.
```

```
x = 3×1

     1
     0
     0
```

Use Triangular Portion of Matrix

Specify `'upper'` to use only the upper triangular portion of an input matrix in the decomposition.

Create a coefficient matrix. Construct a triangular decomposition for the matrix using only the upper triangular portion. This option can be useful in cases where both an upper triangular and lower triangular matrix are stored in the same matrix.

```
A = randi([0 5],10)
```

```
A = 10×10

     4     0     3     4     2     1     4     5     2     0
     5     5     0     0     2     4     1     1     4     0
     0     5     5     1     4     3     3     4     3     3
     5     2     5     0     4     0     4     1     3     4
     3     4     4     0     1     0     5     5     5     5
     0     0     4     4     2     2     5     2     1     0
     1     2     4     4     2     5     3     1     4     3
     3     5     2     1     3     2     0     1     4     2
     5     4     3     5     4     3     0     3     2     0
```

```
     5     5     1     0     4     1     1     2     3     2
```

```
dA = decomposition(A,'triangular','upper')
```

```
dA = 
  decomposition with properties:

    MatrixSize: [10 10]
          Type: 'triangular'

  Show all properties
```

Turn Off Matrix Condition Warnings

Use the `'CheckCondition'` name-value pair to turn off warnings based on the condition of the coefficient matrix when solving a linear system using `decomposition`.

Create a coefficient matrix that is ill conditioned. In this matrix, averaging together the first two columns produces the third column.

```
A = [1 2 1.5; 3 4 3.5; 5 6 5.5]
```

```
A = 3×3

    1.0000    2.0000    1.5000
    3.0000    4.0000    3.5000
    5.0000    6.0000    5.5000
```

Solve a linear system Ax = *b* using a vector of 1s for the right-hand side. `mldivide` produces a warning about the conditioning of the coefficient matrix.

```
b = ones(3,1);
x = A\b
```

```
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND

x = 3×1

   -0.6250
    1.3750
```

```
   -0.7500
```

Now create a `decomposition` object for the matrix and solve the same linear system. Specify `'CheckCondition'` as `false` so that `mldivide` does not check the condition of the coefficient matrix. Even though the same solution is returned, `mldivide` does not display the warning message.

```
dA = decomposition(A,'CheckCondition',false);
x = dA\b
```

```
x = 3×1

   -0.6250
    1.3750
   -0.7500
```

Use the `isIllConditioned` function to check whether the `decomposition` object is based on an ill-conditioned matrix.

```
tf = isIllConditioned(dA)
```

```
tf = logical
   1
```

Extended Capabilities

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- If `A` is sparse and type is `'lu'`, then `CheckCondition` must be false.
- If `A` is sparse, then types `'chol'` and `'qr'` are not supported.
- If `A` is sparse rectangular, then `decomposition` is not supported.

- `rank` only supports decomposition type `'qr'`.
- `rcond` does not support decomposition of sparse matrices and always returns `[]`.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`lsqminnorm` | `mldivide` | `mrdivide`

Topics

“Systems of Linear Equations”

Introduced in R2017b

isIllConditioned

Determine whether matrix is ill conditioned

Syntax

```
tf = isIllConditioned(dA)
```

Description

`tf = isIllConditioned(dA)` returns logical `1` (`true`) if the original coefficient matrix `A` used to create decomposition `dA` is ill conditioned; otherwise, it returns logical `0` (`false`).

The test used depends on the type of decomposition:

- `'qr'` and `'cod'` decompositions — The coefficient matrix is ill conditioned if `rank(dA) < min(size(A))`.
- All other decompositions — The coefficient matrix is ill conditioned if `rcond(dA) < eps`.

If `isIllConditioned` returns logical `1` (`true`), then solving a linear system with either `dA\b` or `b/dA` displays a warning. Use the `CheckCondition` property of the decomposition object `dA` to turn off these warnings.

Examples

Check Condition of Coefficient Matrix

Create a matrix decomposition object for a 25-by-25 Hilbert coefficient matrix and then check to see whether the underlying coefficient matrix is ill conditioned.

```
A = hilb(25);
dA = decomposition(A)
```

```
dA = 
  decomposition with properties:

    MatrixSize: [25 25]
          Type: 'ldl'

  Show all properties
```

```
tf = isIllConditioned(dA)
```

```
tf = logical
   1
```

Check the reciprocal condition number of the coefficient matrix. In this case `isIllConditioned` determines that the coefficient matrix `A` is ill conditioned because `rcond(dA)` is smaller than `eps`.

```
rcond(dA)
```

```
ans = 2.3569e-20
```

Input Arguments

dA — Input decomposition
`decomposition` object

Input decomposition, specified as a `decomposition` object.

Example: `dA = decomposition(A,'qr')`

Tips

- `isIllConditioned` uses rank and condition number estimates of the decomposition object. These estimates can differ compared to calling `rank(A)` or `rcond(A)` on the coefficient matrix directly.

See Also

`decomposition` | `rank` | `rcond`

Introduced in R2017b

deconv

Deconvolution and polynomial division

Syntax

```
[q,r] = deconv(u,v)
```

Description

`[q,r] = deconv(u,v)` deconvolves a vector `v` out of a vector `u` using long division, and returns the quotient `q` and remainder `r` such that `u = conv(v,q)+r`. If `u` and `v` are vectors of polynomial coefficients, then deconvolving them is equivalent to dividing the polynomial represented by `u` by the polynomial represented by `v`.

Examples

Polynomial Division

Create two vectors `u` and `v` containing the coefficients of the polynomials $2x^3 + 7x^2 + 4x + 9$ and $x^2 + 1$, respectively. Divide the first polynomial by the second by deconvolving `v` out of `u`, which results in quotient coefficients corresponding to the polynomial $2x + 7$ and remainder coefficients corresponding to $2x + 2$.

```
u = [2 7 4 9];
v = [1 0 1];
[q,r] = deconv(u,v)
```

```
q = 1×2

     2     7


r = 1×4
```

```
     0     0     2     2
```

Input Arguments

u,v — Input vectors
row or column vectors

Input vectors, specified as either row or column vectors. u and v can be different lengths or data types. If one or both of u and v are of type single, then the output is also of type single. Otherwise, deconv returns type double.

Data Types: double | single
Complex Number Support: Yes

Output Arguments

q — Quotient
row or column vector

Quotient, returned as a row or column vector such that u = conv(v,q)+r.

Data Types: double | single

r — Remainder
row or column vector

Remainder, returned as a row or column vector such that u = conv(v,q)+r.

Data Types: double | single

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

See Also

`conv` | `residue`

Introduced before R2006a

deg2rad

Convert angle from degrees to radians

Syntax

```
R = deg2rad(D)
```

Description

`R = deg2rad(D)` converts angle units from degrees to radians for each element of `D`.

Examples

Right Angle in Radians

Convert a 90 degree angle into radians.

```
R = deg2rad(90)
```

```
R = 1.5708
```

Surface Distance

Specify the spherical distance between Munich and Bangalore in degrees and the mean radius of Earth in kilometers. Compute the distance (measured along the Earth's surface) between Munich and Bangalore, in kilometers.

```
D = 64.7;
radEarth = 6371;
R = deg2rad(D);
dist = radEarth*R
```

```
dist = 7.1943e+03
```

Input Arguments

D — Angle in degrees
scalar | vector | matrix | multidimensional array

Angle in degrees, specified as a scalar, vector, matrix, or multidimensional array. If D contains complex elements, deg2rad converts the real and imaginary parts separately.

Data Types: single | double
Complex Number Support: Yes

Output Arguments

R — Angle in radians
scalar | vector | matrix | multidimensional array

Angle in radians, returned as a scalar, vector, matrix, or multidimensional array. R is the same size as D.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`rad2deg`

Introduced in R2015b

del2

Discrete Laplacian

Syntax

```
L = del2(U)
L = del2(U,h)
L = del2(U,hx,hy,...,hN)
```

Description

`L = del2(U)` returns a discrete approximation of Laplace's differential operator on page 1-3213 applied to `U` using the default spacing, `h = 1`, between all points.

`L = del2(U,h)` specifies a uniform, scalar spacing, `h`, between points in all dimensions of `U`.

`L = del2(U,hx,hy,...,hN)` specifies the spacing `hx,hy,...,hN` between points in each dimension of `U`. Specify each spacing input as a scalar or a vector of coordinates. The number of spacing inputs must equal the number of dimensions in `U`.

- The first spacing value `hx` specifies the *x*-spacing (as a scalar) or *x*-coordinates (as a vector) of the points. If it is a vector, its length must be equal to `size(U,2)`.
- The second spacing value `hy` specifies the *y*-spacing (as a scalar) or *y*-coordinates (as a vector) of the points. If it is a vector, its length must be equal to `size(U,1)`.
- All other spacing values specify the spacing (as scalars) or coordinates (as vectors) of the points in the corresponding dimension in `U`. If, for `n > 2`, the `n`th spacing input is a vector, then its length must be equal to `size(U,n)`.

Examples

Second Derivative of Vector

Calculate the acceleration of an object from a vector of position data.

Create a vector of position data.

```
p = [1 3 6 10 16 18 29];
```

To find the acceleration of the object, use `del2` to calculate the second numerical derivative of `p`. Use the default spacing `h = 1` between data points.

```
L = 4*del2(p)
```

```
L = 1×7

     1     1     1     2    -4     9    22
```

Each value of `L` is an approximation of the instantaneous acceleration at that point.

Second Derivative of Cosine Vector

Calculate the discrete 1-D Laplacian of a cosine vector.

Define the domain of the function.

```
x = linspace(-2*pi,2*pi);
```

This produces 100 evenly spaced points in the range $-2\pi \le x \le 2\pi$.

Create a vector of cosine values in this domain.

```
U = cos(x);
```

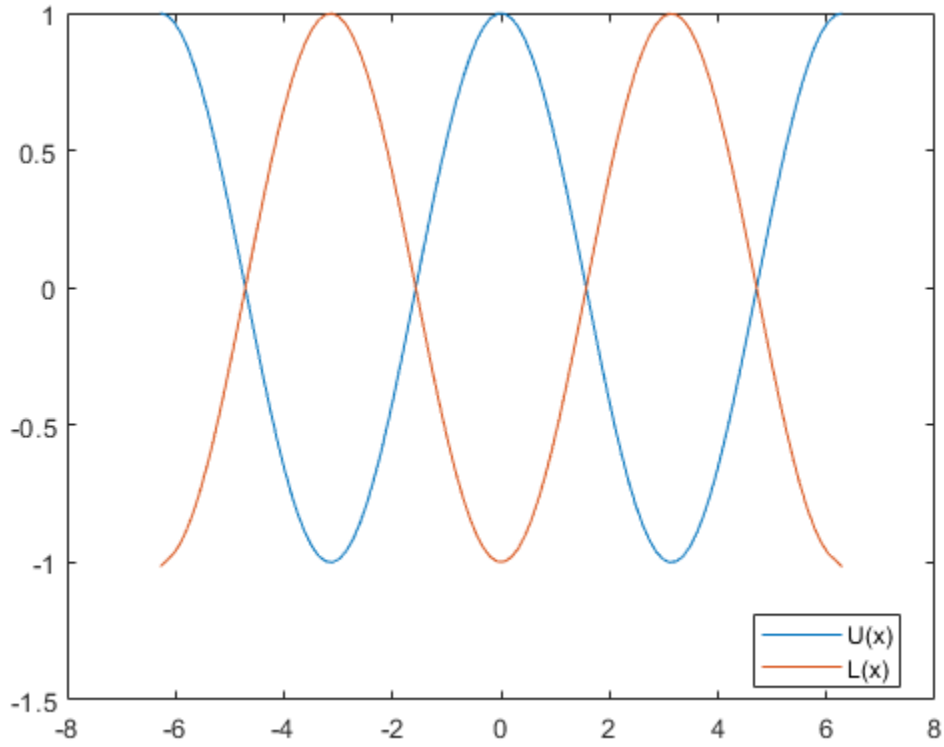
Calculate the Laplacian of `U` using `del2`. Use the domain vector `x` to define the 1-D coordinate of each point in `U`.

```
L = 4*del2(U,x);
```

Analytically, the Laplacian of this function is equal to $\Delta U = -\cos(x)$.

Plot the results.

```
plot(x,U,x,L)
legend('U(x)','L(x)','Location','Best')
```



The graph of U and L agrees with the analytic result for the Laplacian.

Laplacian of Multivariate Function

Calculate and plot the discrete Laplacian of a multivariate function.

Define the x and y domain of the function.

```
[x,y] = meshgrid(-5:0.25:5,-5:0.25:5);
```

Define the function $U(x, y) = \frac{1}{3}\left(x^4 + y^4\right)$ over this domain.

```
U = 1/3.*(x.^4+y.^4);
```

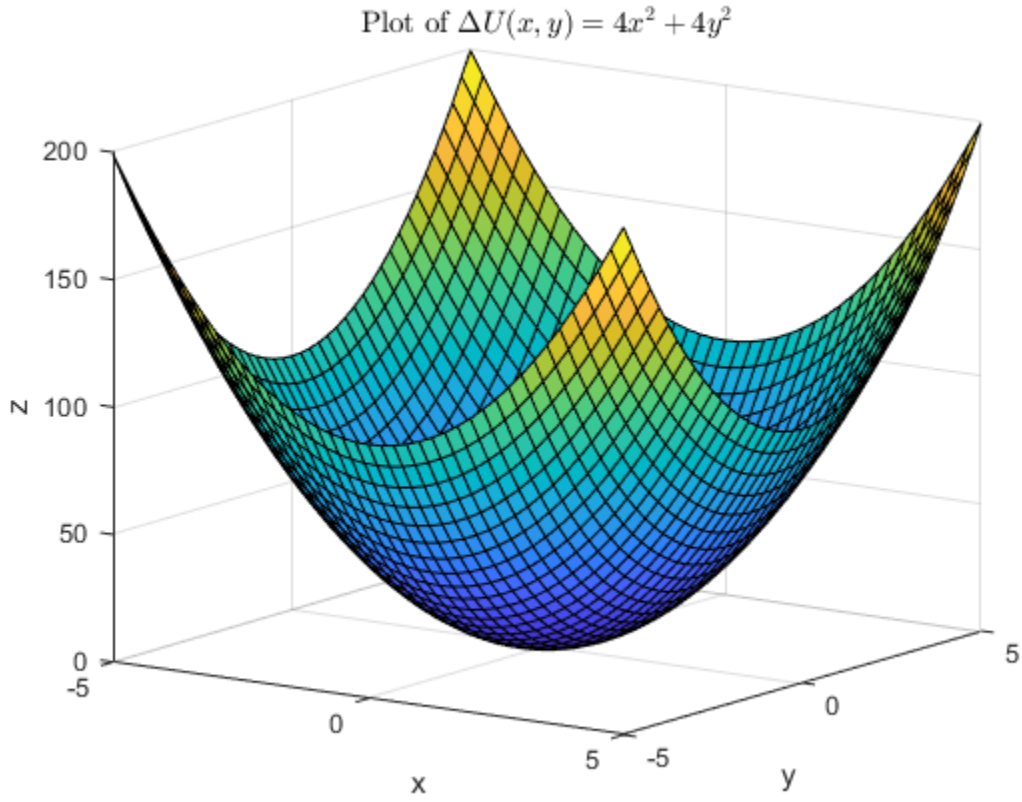
Calculate the Laplacian of this function using `del2`. The spacing between the points in `U` is equal in all directions, so you can specify a single spacing input, `h`.

```
h = 0.25;
L = 4*del2(U,h);
```

Analytically, the Laplacian of this function is equal to $\Delta U(x, y) = 4x^2 + 4y^2$.

Plot the discrete Laplacian, `L`.

```
figure
surf(x,y,L)
grid on
title('Plot of $\Delta U(x,y) = 4x^2+4y^2$','Interpreter','latex')
xlabel('x')
ylabel('y')
zlabel('z')
view(35,14)
```



The graph of L agrees with the analytic result for the Laplacian.

Laplacian of Natural Logarithm Function

Calculate the discrete Laplacian of a natural logarithm function.

Define the x and y domain of the function on a grid of real numbers.

```
[x,y] = meshgrid(-5:5,-5:0.5:5);
```

Define the function $U(x, y) = \frac{1}{2}\log\left(x^2 y\right)$ over this domain.

```
U = 0.5*log(x.^2.*y);
```

The logarithm is complex-valued when the argument `y` is negative.

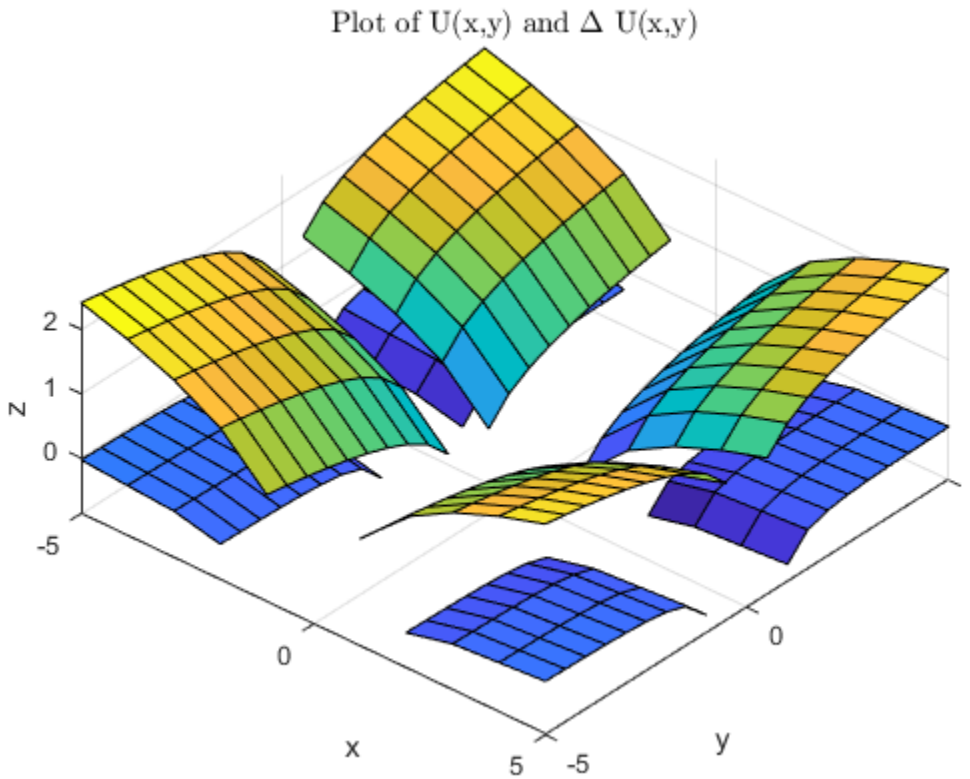
Use `del2` to calculate the discrete Laplacian of this function. Specify the spacing between grid points in each direction.

```
hx = 1;
hy = 0.5;
L = 4*del2(U,hx,hy);
```

Analytically, the Laplacian is equal to $\Delta U(x, y) = -\left(1/x^2 + 1/2y^2\right)$. This function is not defined on the lines $x = 0$ or $y = 0$.

Plot the real parts of `U` and `L` on the same graph.

```
figure
surf(x,y,real(L))
hold on
surf(x,y,real(U))
grid on
title('Plot of U(x,y) and $\Delta$ U(x,y)','Interpreter','latex')
xlabel('x')
ylabel('y')
zlabel('z')
view(41,58)
```



The top surface is U and the bottom surface is L.

Input Arguments

U — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

h — Spacing in all dimensions
1 (default) | scalar

Spacing in all dimensions, specified as 1 (default), or a scalar.

Data Types: `single` | `double`
Complex Number Support: Yes

hx,hy,...,hN — Spacing in each dimension (as separate arguments)
scalars | vectors

Spacing in each dimension, specified as separate arguments of scalars (for uniform spacing) or vectors (for nonuniform spacing). The number of spacing inputs must be equal to the number of dimensions in U. Each spacing input defines the spacing between points in one dimension of U:

- The first spacing value hx specifies the *x*-spacing (as a scalar) or *x*-coordinates (as a vector) of the points. If it is a vector, its length must be equal to `size(U,2)`.
- The second spacing value hy specifies the *y*-spacing (as a scalar) or *y*-coordinates (as a vector) of the points. If it is a vector, its length must be equal to `size(U,1)`.
- All other spacing values specify the spacing (as scalars) or coordinates (as vectors) of the points in the corresponding dimension in U. If, for `n > 2`, the nth spacing input is a vector, then its length must be equal to `size(U,n)`.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

L — Discrete Laplacian approximation
vector | matrix | multidimensional array

Discrete Laplacian approximation, returned as a vector, matrix, or multidimensional array. L is the same size as the input, U.

More About

Laplace’s differential operator

If a matrix `U` is a function *U(x,y)* that is evaluated at the points of a square grid, then `4*del2(U)` is a finite difference approximation of Laplace's differential operator applied to *U*,

$$L = \frac{\Delta U}{4} = \frac{1}{4}\left(\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2}\right).$$

For functions of more variables, *U(x,y,z,...)*, the discrete Laplacian `del2(U)` calculates second-derivatives in each dimension,

$$L = \frac{\Delta U}{2N} = \frac{1}{2N}\left(\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} + \frac{\partial^2 U}{\partial z^2} + \ldots\right),$$

where N is the number of dimensions in U and $N \geq 2$.

Algorithms

If the input `U` is a matrix, the interior points of `L` are found by taking the difference between a point in `U` and the average of its four neighbors:

$$L_{ij} = \left[\frac{\left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}\right)}{4} - u_{i,j}\right].$$

Then, `del2` calculates the values on the edges of `L` by linearly extrapolating the second differences from the interior. This formula is extended for multidimensional `U`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`diff` | `gradient`

Introduced before R2006a

matlab.project.deleteProject

Stop folder management and delete project definition files

Syntax

```
matlab.project.deleteProject(path)
```

Description

`matlab.project.deleteProject(path)` stops managing the specified folder with a project and deletes all files generated when you created or made changes to the project. Use this command to delete the project `(*.prj)` file and project definition files stored in the `resources/project` folder. The project no longer manages the folder. All other files remain intact.

Examples

Delete the project files that manage the specified folder

Stop managing the specified folder with a project and delete all related project definitions files. See “Project Definitions Files”.

```
matlab.project.deleteProject("C:\work\myprojectname");
```

Input Arguments

path — Project path
character vector | string scalar

Project path, specified as a character vector or string scalar.

See Also

`createProject` | `currentProject` | `openProject`

Introduced in R2019b

DelaunayTri class

Superclasses: `TriRep`

(Not recommended) Delaunay triangulation in 2-D and 3-D

Note `DelaunayTri` is not recommended. Use `delaunayTriangulation` instead.

Description

`DelaunayTri` creates a Delaunay triangulation object from a set of points. You can incrementally modify the triangulation by adding or removing points. In 2-D triangulations you can impose edge constraints. You can perform topological and geometric queries, and compute the Voronoi diagram and convex hull.

Construction

DelaunayTri	(Not recommended) Construct Delaunay triangulation

Methods

convexHull	(Not recommended) Convex hull
inOutStatus	(Not recommended) Status of triangles in 2-D constrained Delaunay triangulation
nearestNeighbor	(Not recommended) Point closest to specified location
pointLocation	(Not recommended) Simplex containing specified location
voronoiDiagram	(Not recommended) Voronoi diagram

Inherited methods

baryToCart	(Not recommended) Convert point coordinates from barycentric to Cartesian
cartToBary	(Not recommended) Convert point coordinates from Cartesian to barycentric
circumcenters	(Not recommended) Circumcenters of specified simplices
edgeAttachments	(Not recommended) Simplices attached to specified edges
edges	(Not recommended) Triangulation edges
faceNormals	(Not recommended) Unit normals to specified triangles
featureEdges	(Not recommended) Sharp edges of surface triangulation
freeBoundary	(Not recommended) Facets referenced by only one simplex
incenters	(Not recommended) Incenters of specified simplices
isEdge	(Not recommended) Test if vertices are joined by edge
neighbors	(Not recommended) Simplex neighbor information
size	(Not recommended) Size of triangulation matrix
vertexAttachments	(Not recommended) Return simplices attached to specified vertices

Properties

`Constraints`	`Constraints` is a `numc`-by-2 matrix that defines the constrained edge data in the triangulation, where `numc` is the number of constrained edges. Each constrained edge is defined in terms of its endpoint indices into `X`. The constraints can be specified when the triangulation is constructed or can be imposed afterwards by directly editing the constraints data. This feature is only supported for 2-D triangulations.

`X`	The dimension of `X` is `mpts`-by-`ndim`, where `mpts` is the number of points and `ndim` is the dimension of the space where the points reside. If column vectors of `x`,`y` or `x`,`y`,`z` coordinates are used to construct the triangulation, the data is consolidated into a single matrix `X`.
`Triangulation`	Triangulation is a matrix representing the set of simplices (triangles or tetrahedra etc.) that make up the triangulation. The matrix is of size `mtri`-by-`nv`, where `mtri` is the number of simplices and `nv` is the number of vertices per simplex. The triangulation is represented by standard simplex-vertex format; each row specifies a simplex defined by indices into `X`, where `X` is the array of point coordinates.

Instance Hierarchy

`DelaunayTri` is a subclass of `TriRep`.

Copy Semantics

Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

More About

2-D Delaunay Triangulation

The 2-D Delaunay triangulation of a set of points is the triangulation in which no point of the set is contained in the circumcircle for any triangle in the triangulation. The definition extends naturally to higher dimensions.

See Also

`delaunayTriangulation` | `scatteredInterpolant` | `triangulation`

DelaunayTri

Class: `DelaunayTri`

(Not recommended) Construct Delaunay triangulation

Note `DelaunayTri` is not recommended. Use `delaunayTriangulation` instead.

Syntax

```
DT = DelaunayTri()
DT = DelaunayTri(X)
DT = DelaunayTri(x,y)
DT = DelaunayTri(x,y,z)
DT = DelaunayTri(..., C)
```

Description

`DT = DelaunayTri()` creates an empty Delaunay triangulation.

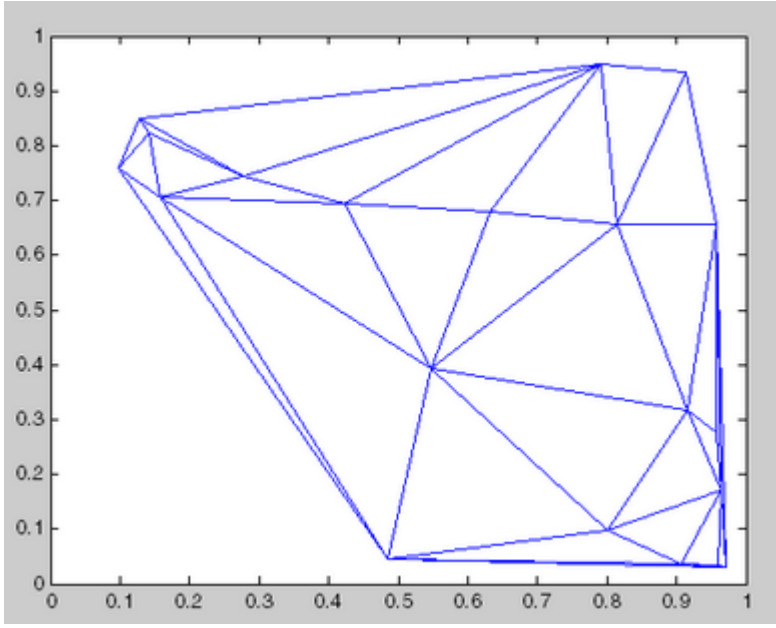
`DT = DelaunayTri(X)`, `DT = DelaunayTri(x,y)` and `DT = DelaunayTri(x,y,z)` create a Delaunay triangulation from a set of points. The points can be specified as an `mpts`-by-`ndim` matrix `X`, where `mpts` is the number of points and `ndim` is the dimension of the space where the points reside, where `ndim` is 2 or 3. Alternatively, the points can be specified as column vectors `(x,y)` or `(x,y,z)` for 2-D and 3-D input.

`DT = DelaunayTri(..., C)` creates a constrained Delaunay triangulation. The edge constraints `C` are defined by an `numc`-by-2 matrix, `numc` being the number of constrained edges. Each row of `C` defines a constrained edge in terms of its endpoint indices into the point set `X`. This feature is only supported for 2-D triangulations.

Examples

Compute the Delaunay triangulation of twenty random points located within a unit square.

```
x = rand(20,1);
y = rand(20,1);
dt = DelaunayTri(x,y)
triplot(dt);
```



For more examples, type `help demoDelaunayTri` at the MATLAB command-line prompt.

More About

2-D Delaunay Triangulation

The 2-D Delaunay triangulation of a set of points is the triangulation in which no point of the set is contained in the circumcircle for any triangle in the triangulation. The definition extends naturally to higher dimensions.

See Also

`delaunayTriangulation` | `scatteredInterpolant` | `triangulation`

delaunay

Delaunay triangulation

Syntax

```
DT = delaunay(P)
DT = delaunay(x,y)
DT = delaunay(x,y,z)
```

Description

`DT = delaunay(P)` creates a 2-D or 3-D Delaunay triangulation from the points in a matrix `P`. The output `DT` is a three-column (for two dimensions) or four-column (for three dimensions) matrix where each row contains the row indices of the input points that make up a triangle or tetrahedron in the triangulation.

`DT = delaunay(x,y)` creates a 2-D Delaunay triangulation from the points in vectors `x` and `y`.

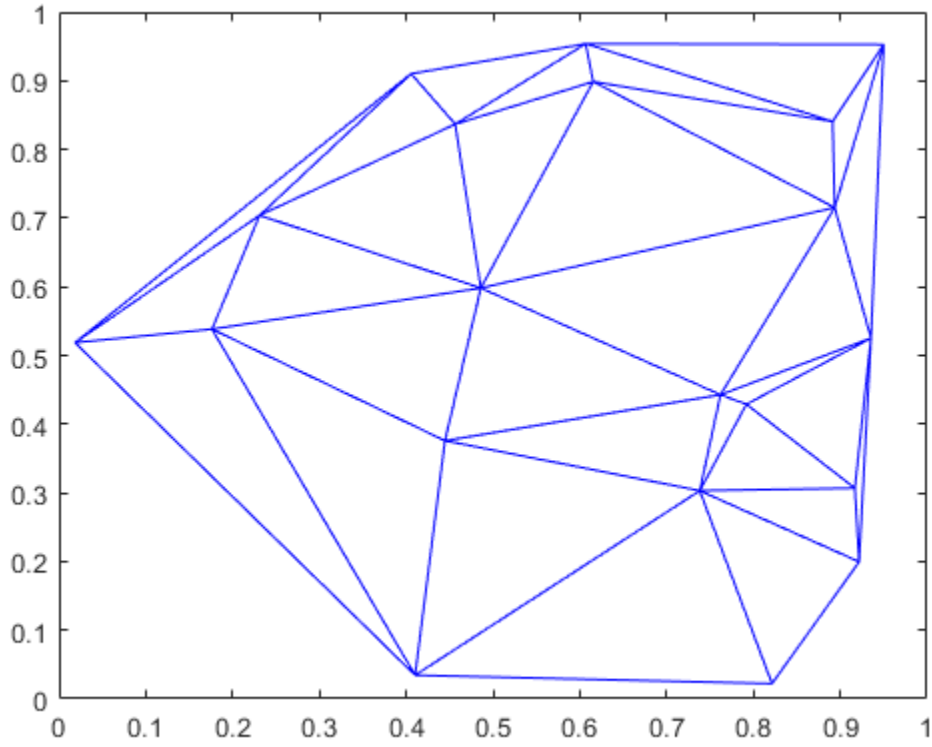
`DT = delaunay(x,y,z)` creates a 3-D Delaunay triangulation from the points in vectors `x`, `y`, and `z`.

Examples

2-D Delaunay Triangulation

Create a set of 2-D points. Compute and plot the Delaunay triangulation.

```
x = gallery('uniformdata',[20,1],0);
y = gallery('uniformdata',[20,1],1);
DT = delaunay(x,y);
triplot(DT,x,y);
```



Input Arguments

P — Points
matrix

Points, specified as a matrix whose columns are the *x*-coordinates, *y*-coordinates, and (possibly) *z*-coordinates of the triangulation points. Each row of the output DT contains the row numbers of P that make up a single triangle or tetrahedron in the triangulation.

x — *x*-coordinates
column vector

x-coordinates, specified as a column vector.

y — *y*-coordinates
column vector

y-coordinates, specified as a column vector.

z — *z*-coordinates
column vector

z-coordinates, specified as a column vector.

Tips

- Use the `triplot`, `trisurf`, `trimesh`, or `tetramesh` functions to plot the output of `delaunay`.
- For more functionality with Delaunay triangulations, consider creating a `delaunayTriangulation` object and using object functions to compute geometric quantities and query properties of the triangulation.

See Also

`delaunayTriangulation` | `plot` | `tetramesh` | `trimesh` | `triplot` | `trisurf`

Introduced before R2006a

delaunayn

N-D Delaunay triangulation

Syntax

```
T = delaunayn(X)
T = delaunayn(X,options)
```

Description

`T = delaunayn(X)` computes a set of simplices such that no data points of `X` are contained in any circumspheres of the simplices. The set of simplices forms the Delaunay triangulation. `X` is an `m`-by-`n` array representing m points in n-dimensional space. `T` is a `numt`-by-(`n+1`) array where each row contains the indices into `X` of the vertices of the corresponding simplex.

`T = delaunayn(X,options)` specifies a cell array of options. The default options are:

- `{'Qt','Qbb','Qc'}` for 2- and 3-dimensional input
- `{'Qt','Qbb','Qc','Qx'}` for 4 and higher-dimensional input

If `options` is `[]`, the default options used. If `options` is `{''}`, no options are used, not even the default.

Visualization

Plotting the output of `delaunayn` depends of the value of `n`:

- For `n = 2`, use `triplot`, `trisurf`, or `trimesh` as you would for `delaunay`.
- For `n = 3`, use `tetramesh`.

  For more control over the color of the facets, use `patch` to plot the output.
- You cannot plot `delaunayn` output for `n > 3`.

Examples

3-D Delaunay Triangulation

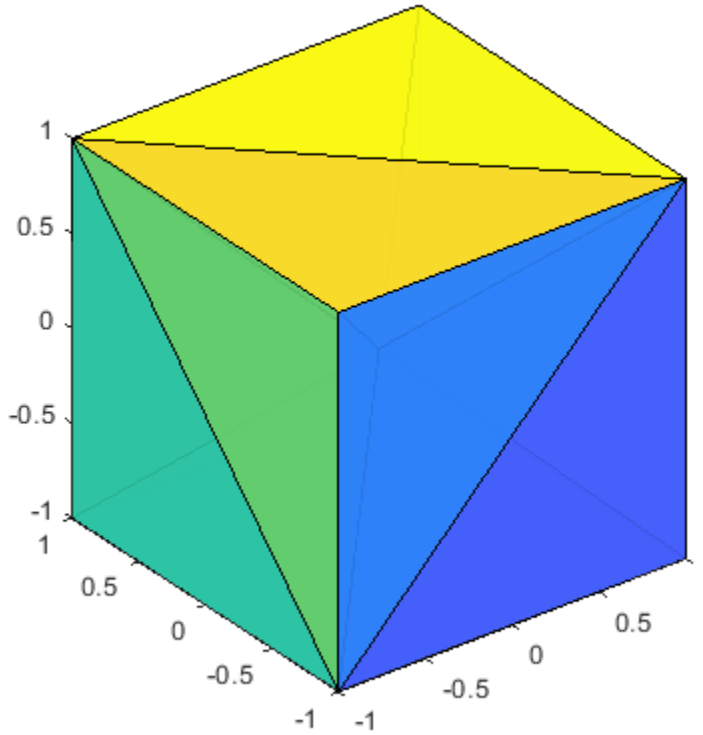
This example generates an n-dimensional Delaunay triangulation, where `n = 3`.

```
d = [-1 1];
[x,y,z] = meshgrid(d,d,d);  % A cube
x = [x(:);0];
y = [y(:);0];
z = [z(:);0];
% [x,y,z] are corners of a cube plus the center.
X = [x(:) y(:) z(:)];
Tes = delaunayn(X)
```

```
Tes = 12×4

     4     3     9     1
     4     9     2     1
     7     9     3     1
     7     5     9     1
     7     9     4     3
     7     8     4     9
     6     2     9     1
     6     9     5     1
     6     4     9     2
     6     4     8     9
       ⋮
```

You can use `tetramesh` to visualize the tetrahedrons that form the corresponding simplex. `camorbit` rotates the camera position to provide a meaningful view of the figure.

```
tetramesh(Tes,X);
camorbit(20,0)
```



See Also

`camorbit` | `convhulln` | `delaunayTriangulation` | `tetramesh` | `voronoin`

Introduced before R2006a

delaunayTriangulation

Delaunay triangulation in 2-D and 3-D

Description

Use the `delaunayTriangulation` object to create a 2-D or 3-D Delaunay triangulation on page 1-3235 from a set of points. For 2-D data, you can also specify edge constraints.

You can perform a variety of topological and geometric queries on a `delaunayTriangulation`, including any `triangulation` query. For example, locate a facet that contains a specific point, find the vertices of the convex hull, or compute the Voronoi Diagram.

Creation

To create a `delaunayTriangulation` object, use the `delaunayTriangulation` function with input arguments that define the triangulation's points and constrained edges.

Syntax

```
DT = delaunayTriangulation(P)
DT = delaunayTriangulation(P,C)
DT = delaunayTriangulation(x,y)
DT = delaunayTriangulation(x,y,C)
DT = delaunayTriangulation(x,y,z)
DT = delaunayTriangulation()
```

Description

`DT = delaunayTriangulation(P)` creates a Delaunay triangulation from the points in `P`. The matrix `P` has 2 or 3 columns, depending on whether your points are in 2-D or 3-D space.

`DT = delaunayTriangulation(P,C)` specifies the edge constraints in the matrix `C` for the 2-D points in `P`. Each row of `C` defines the start and end vertex IDs of a constrained edge. Vertex IDs are the row numbers of the corresponding vertices in the property `DT.Points`.

`DT = delaunayTriangulation(x,y)` creates a 2-D Delaunay triangulation from the point coordinates in the column vectors `x` and `y`.

`DT = delaunayTriangulation(x,y,C)` specifies the edge constraints in a matrix `C`.

`DT = delaunayTriangulation(x,y,z)` creates a 3-D Delaunay triangulation from the point coordinates in the column vectors `x`, `y`, and `z`.

`DT = delaunayTriangulation()` creates an empty Delaunay triangulation.

Input Arguments

`P` — Points
matrix

Points, specified as a matrix whose columns are the *x*-coordinates, *y*-coordinates, and (possibly) `z`-coordinates of the triangulation points. The row numbers of `P` are the vertex IDs in the triangulation.

`x` — *x*-coordinates
column vector

x-coordinates of triangulation points, specified as a column vector.

`y` — *y*-coordinates
column vector

y-coordinates of triangulation points, specified as a column vector.

`z` — *z*-coordinates
column vector

z-coordinates of triangulation points, specified as a column vector.

`C` — Vertex IDs of constrained edges
2-column matrix

Vertex IDs of constrained edges, specified as a 2-column matrix. Each row of `C` corresponds to a constrained edge and contains two IDs:

- `C(j,1)` is the ID of the vertex at the start of an edge.
- `C(j,2)` is the ID of the vertex at end of the edge.

You can specify edge constraints for 2-D triangulations only.

Properties

`Points` — Triangulation points

matrix

Points in the triangulation, represented as a matrix with the following characteristics:

- Each row in `DT.Points` contains the coordinates of a vertex.
- Each row number of `DT.Points` is a vertex ID.

`ConnectivityList` — Triangulation connectivity list

matrix

Triangulation connectivity list, represented as a matrix with the following characteristics:

- Each element in `DT.ConnectivityList` is a vertex ID.
- Each row represents a triangle or tetrahedron in the triangulation.
- Each row number of `DT.ConnectivityList` is a triangle or tetrahedron ID.

`Constraints` — Constrained edges

2-column matrix of vertex IDs

Constrained edges, represented as a 2-column matrix of vertex IDs. Each row of `DT.Constraints` corresponds to a constrained edge and contains two IDs:

- `DT.Constraints(j,1)` is the ID of the vertex at the start of an edge.
- `DT.Constraints(j,2)` is the ID of the vertex at end of the edge.

`DT.Constraints` is an empty matrix when the triangulation has no constrained edges.

Object Functions

convexHull	Convex hull of Delaunay triangulation
isInterior	Query interior points of Delaunay triangulation
voronoiDiagram	Voronoi diagram of Delaunay triangulation
barycentricToCartesian	Convert coordinates from barycentric to Cartesian
cartesianToBarycentric	Convert coordinates from Cartesian to barycentric
circumcenter	Circumcenter of triangle or tetrahedron
edgeAttachments	Triangles or tetrahedra attached to specified edge
edges	Triangulation edges
faceNormal	Triangulation unit normal vectors
featureEdges	Handle sharp edges of triangulation
freeBoundary	Free boundary facets
incenter	Incenter of triangulation elements
isConnected	Test if two vertices are connected by an edge
nearestNeighbor	Closest vertex
neighbors	Triangle or tetrahedron neighbors
pointLocation	Triangle or tetrahedron enclosing point
size	Size of triangulation connectivity list
vertexAttachments	Triangles or tetrahedra attached to vertex
vertexNormal	Triangulation vertex normal

Examples

2-D Delaunay Triangulation

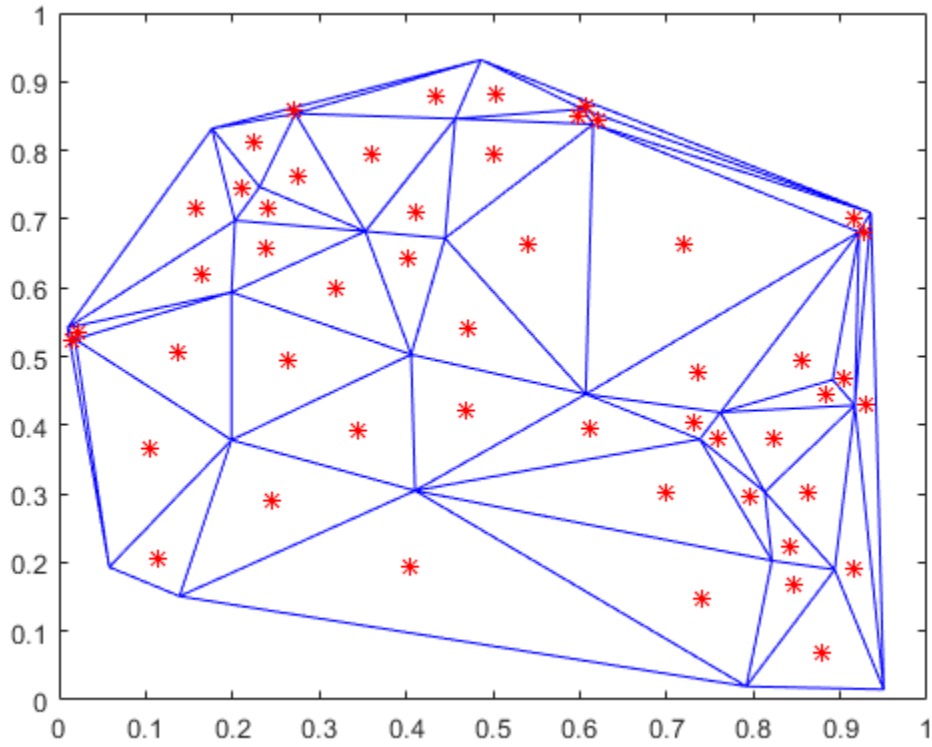
Create a 2-D `delaunayTriangulation` object for 30 random points.

```
P = gallery('uniformdata',[30 2],0);
DT = delaunayTriangulation(P)
```

```
DT = 
  delaunayTriangulation with properties:

              Points: [30x2 double]
    ConnectivityList: [50x3 double]
         Constraints: []
```

Compute the center points of each triangle, and plot the triangulation with the center points.

```
IC = incenter(DT);
triplot(DT)
hold on
plot(IC(:,1),IC(:,2),'*r')
```



3-D Delaunay Triangulation

Create a 3-D `delaunayTriangulation` object for 30 random points.

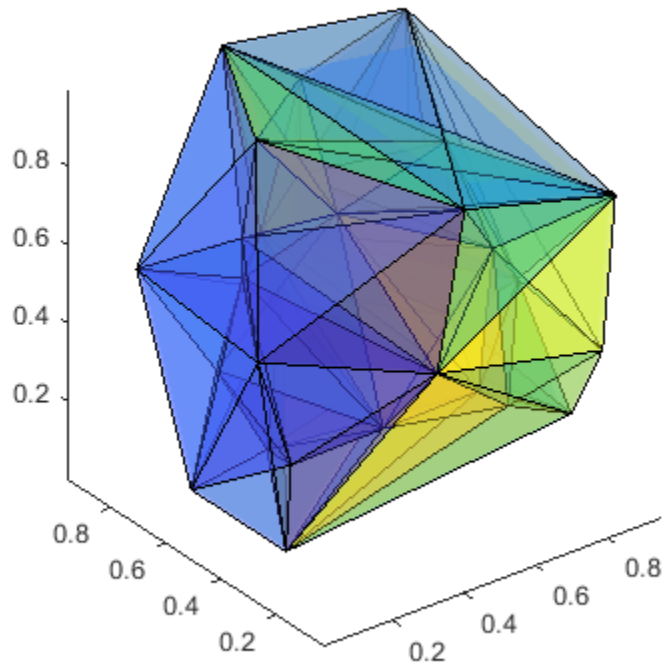
```
x = gallery('uniformdata',[30 1],0);
y = gallery('uniformdata',[30 1],1);
```

```
z = gallery('uniformdata',[30 1],2);
DT = delaunayTriangulation(x,y,z)
```

```
DT 
  delaunayTriangulation with properties:

              Points: [30x3 double]
    ConnectivityList: [111x4 double]
         Constraints: []
```

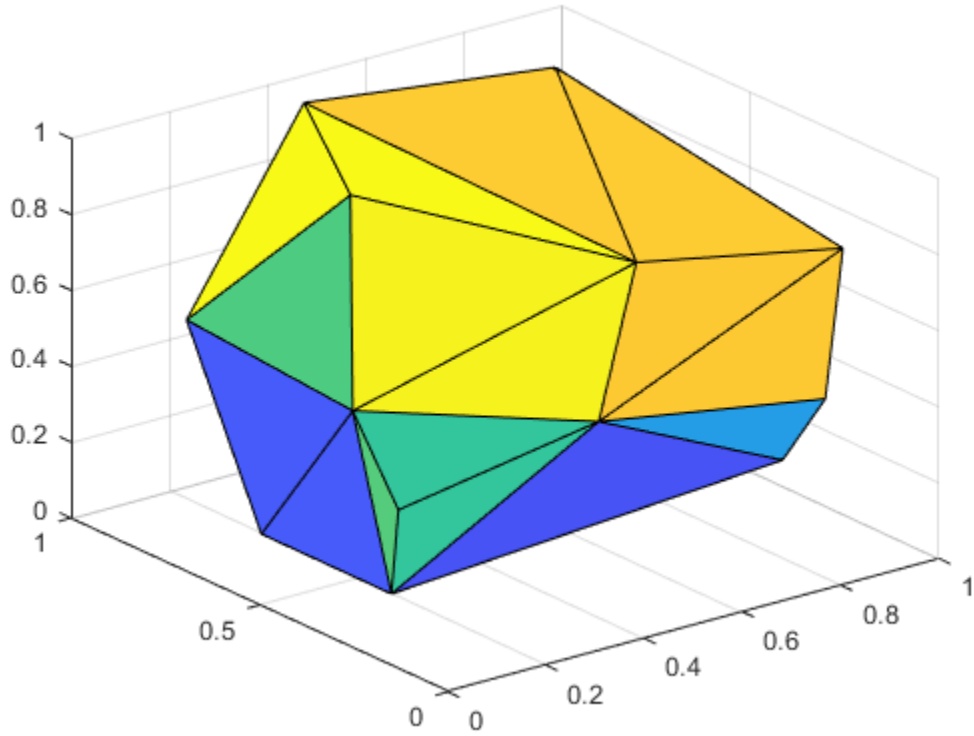
Plot the triangulation.

```
tetramesh(DT,'FaceAlpha',0.3);
```



Compute and plot the convex hull of the triangulation.

```
[K,v] = convexHull(DT);
trisurf(K,DT.Points(:,1),DT.Points(:,2),DT.Points(:,3))
```



More About

Delaunay Triangulation

In a 2-D Delaunay triangulation, the circumcircle associated with each triangle does not contain any points in its interior. Similarly, a 3-D Delaunay triangulation does not have

any points in the interior of the circumsphere associated with each tetrahedron. This definition extends to N-D, although `delaunayTriangulation` supports only 2-D and 3-D.

See Also

`delaunay` | `delaunayn` | `triangulation`

Topics

"Working with Delaunay Triangulations"

Introduced in R2013a

convexHull

Convex hull of Delaunay triangulation

Syntax

```
C = convexHull(DT)
[C,v] = convexHull(DT)
```

Description

`C = convexHull(DT)` returns the vertices of the convex hull of a Delaunay triangulation.

`[C,v] = convexHull(DT)` also returns the area or volume bounded by the convex hull.

Examples

2-D Delaunay Triangulation

Compute and plot the convex hull of a 2-D Delaunay triangulation.

Create a Delaunay triangulation from a set of 2-D points.

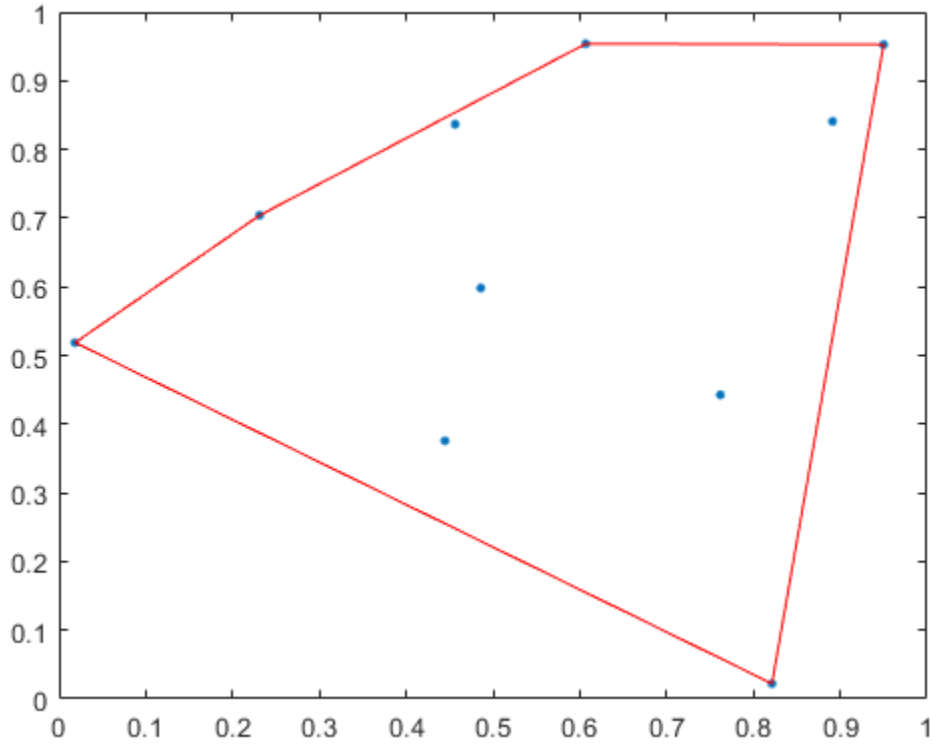
```
x = gallery('uniformdata',[10,1],0);
y = gallery('uniformdata',[10,1],1);
DT = delaunayTriangulation(x,y);
```

Compute the convex hull.

```
C = convexHull(DT);
```

Plot the triangulation and highlight the convex hull in red.

```
plot(DT.Points(:,1),DT.Points(:,2),'.','MarkerSize',10)
hold on
plot(DT.Points(C,1),DT.Points(C,2),'r')
```



3-D Delaunay Triangulation

Compute and plot the convex hull of a 3-D Delaunay Triangulation.

Create a Delaunay triangulation from a 3-D set of points.

```
P = gallery('uniformdata',[25,3],1);
DT = delaunayTriangulation(P);
```

Compute the convex hull and the volume bounded by the convex hull.

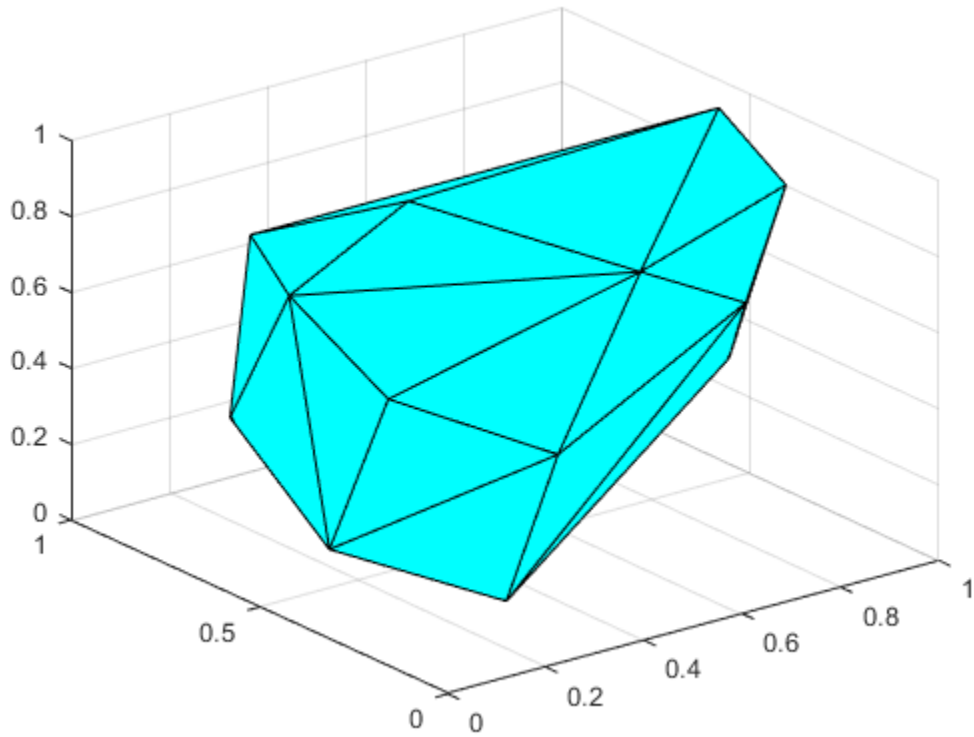
```
[C,v] = convexHull(DT);
```

Display the volume and plot the convex hull.

```
v
```

```
v = 0.3561
```

```
trisurf(C,DT.Points(:,1),DT.Points(:,2),DT.Points(:,3), ...
       'FaceColor','cyan')
```



Input Arguments

DT — Delaunay triangulation
scalar

Delaunay triangulation, specified as a scalar `delaunayTriangulation` object.

Data Types: `delaunayTriangulation`

Output Arguments

C — Convex hull vertices
column vector | matrix

Convex hull vertices, returned as a column vector or matrix of vertex IDs.

- When `DT` is a 2-D triangulation, `C` is a column vector containing the sequence of vertex IDs around the convex hull. The vertex IDs are the row numbers of the vertices in the `Points` property.
- When `DT` is 3-D triangulation, `C` is a 3-column matrix containing the connectivity list of triangle vertices in the convex hull.

Data Types: `double`

v — Area or volume
scalar

Area or volume of the convex hull, returned as a scalar.

Data Types: `double`

See Also

`delaunayTriangulation`

Introduced in R2013a

isInterior

Query interior points of Delaunay triangulation

Syntax

```
TF = isInterior(DT)
```

Description

`TF = isInterior(DT)` returns a column vector of logical values that indicate whether the triangles in a 2-D constrained Delaunay triangulation are inside a bounded geometric domain. An element of `TF` is 1 (`true`) when the corresponding triangle in `DT` is inside the domain and 0 (`false`) otherwise.

Examples

Triangles in Specified Boundary

Compute and plot the triangles of a 2-D constrained Delaunay triangulation within a specified boundary.

Create a geometric domain whose shape is a square frame.

```
outerprofile = [-5 -5; -3 -5; -1 -5;  1 -5;
                 3 -5;  5 -5;  5 -3;  5 -1;
                 5  1;  5  3;  5  5;  3  5;
                 1  5; -1  5; -3  5; -5  5;
                -5  3; -5  1; -5 -1; -5 -3];
innerprofile = outerprofile.*0.5;
P = [outerprofile; innerprofile];
```

Define the edge constraints.

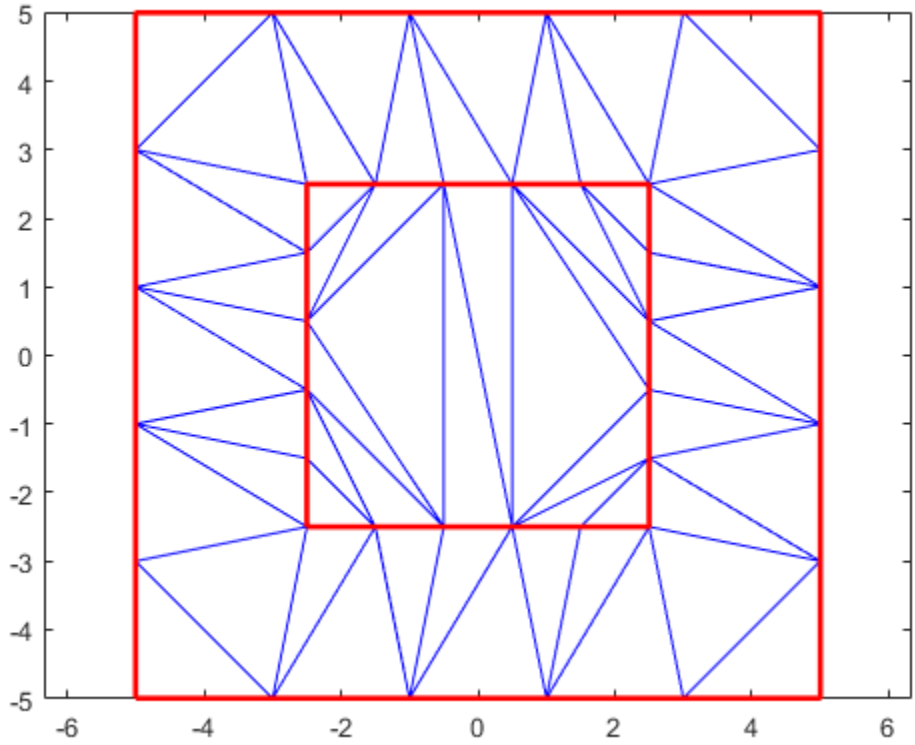
```
outercons = [(1:19)' (2:20)'; 20 1;];
innercons = [(21:39)' (22:40)'; 40 21];
C = [outercons; innercons];
```

Create the constrained Delaunay triangulation.

```
DT = delaunayTriangulation(P,C);
```

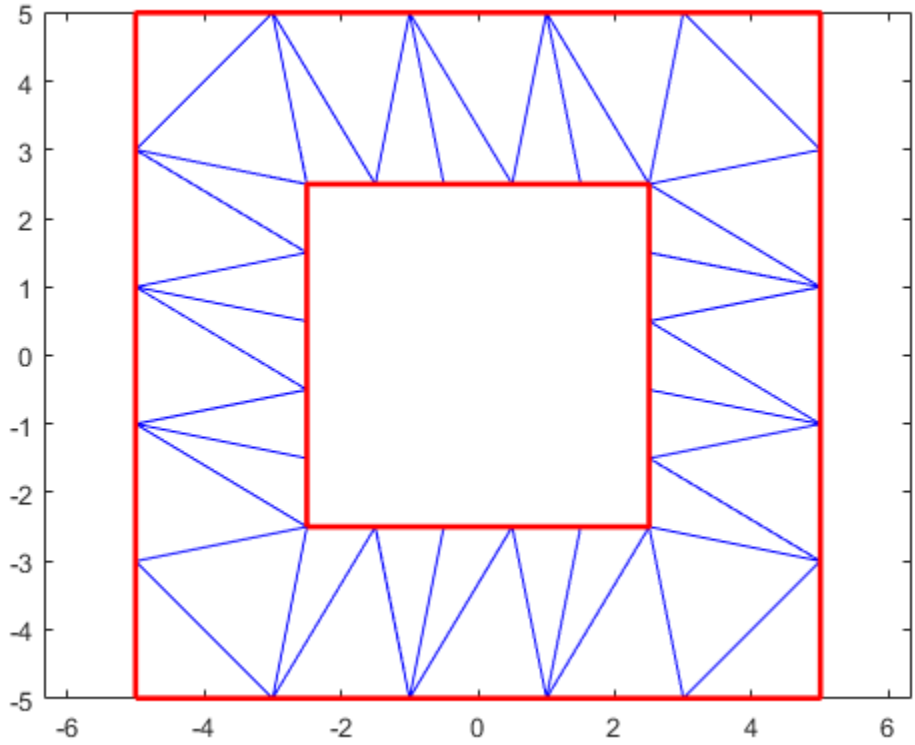
Plot the triangulation, highlighting the inner and outer squares in red.

```
triplot(DT)
hold on
plot(DT.Points(innercons',1),DT.Points(innercons',2), ...
     '-r','LineWidth',2)
plot(DT.Points(outercons',1),DT.Points(outercons',2), ...
     '-r','LineWidth',2)
axis equal
```



Plot only the triangles between the inner and outer squares, highlighting the inner and outer squares in red.

```
figure
TF = isInterior(DT);
triplot(DT.ConnectivityList(TF,:),DT.Points(:,1),DT.Points(:,2))
hold on
plot(DT.Points(outercons',1),DT.Points(outercons',2), ...
     '-r','LineWidth',2)
plot(DT.Points(innercons',1),DT.Points(innercons',2), ...
     '-r','LineWidth',2)
axis equal
```



Input Arguments

DT — Constrained Delaunay triangulation
scalar

Constrained Delaunay triangulation, specified as a scalar 2-D `delaunayTriangulation` object with a set of constrained edges that define a bounded geometric domain. A bounded domain is a region enclosed by multiple constrained edges that do not intersect or overlap.

Data Types: `delaunayTriangulation`

Tips

- `isInterior` can produce incorrect or inconsistent results when boundary constraints intersect or overlap. To avoid this behavior, use constraints that form one or multiple closed boundaries that do not intersect or overlap. When boundary constraints are nested without intersections or overlaps, the inside or outside status alternates across the boundaries.

See Also

`delaunayTriangulation`

Introduced in R2013a

voronoiDiagram

Voronoi diagram of Delaunay triangulation

Syntax

```
[V,r] = voronoiDiagram(DT)
```

Description

`[V,r] = voronoiDiagram(DT)` returns the Voronoi vertices `V` and the Voronoi regions `r` of the points in a Delaunay triangulation. Each region in `r` represents the points surrounding a triangulation vertex that are closer to that vertex than any other vertex in the triangulation. The collection of Voronoi regions make up a Voronoi diagram.

Examples

2-D Delaunay Triangulation

Compute the Voronoi vertices and regions of a 2-D Delaunay triangulation.

Create a Delaunay triangulation from a set of 2-D points.

```
P = [ 0.5    0
      0      0.5
     -0.5   -0.5
     -0.2   -0.1
     -0.1    0.1
      0.1   -0.1
      0.1    0.1 ];
DT = delaunayTriangulation(P);
```

Compute the Voronoi vertices and regions.

```
[V,r] = voronoiDiagram(DT);
```

Display the connectivity of the Voronoi region associated with the 3rd point in the triangulation.

```
r{3}
```

```
ans = 1×4

     1    10     7     4
```

Display the coordinates of the Voronoi vertices bounding the 3rd region. The `Inf` values indicate that the region contains points on the convex hull.

```
V(r{3},:)
```

```
ans = 4×2

       Inf       Inf
    0.7000   -1.6500
   -0.0500   -0.5250
   -1.7500    0.7500
```

Input Arguments

`DT` — Delaunay triangulation
scalar

Delaunay triangulation, specified as a scalar `delaunayTriangulation` object.

Data Types: `delaunayTriangulation`

Output Arguments

`V` — Voronoi vertices
matrix

Voronoi vertices, returned as a 2-column matrix (2-D) or a 3-column matrix (3-D). Each row of `V` contains the coordinates of a Voronoi vertex.

The Voronoi regions associated with points that lie on the convex hull of the triangulation vertices are unbounded. Bounding edges of these regions radiate to infinity. The first vertex in `V` represents the vertex at infinity and is designated with `Inf`.

Data Types: `double`

`r` — Voronoi regions
cell array

Voronoi regions, returned as a cell array whose elements contain the connectivity of the Voronoi vertices in `V`. The points in each row of `r` form the bounding region associated with the corresponding row in the `Points` property.

Data Types: `double`

See Also

`delaunayTriangulation`

Introduced in R2013a

delete

Delete files or objects

Syntax

```
delete filename
delete filename1 ... filenameN
delete(obj)
```

Description

`delete filename` deletes `filename` from disk, without requesting verification. To change whether the specified file is permanently deleted or sent to the recycle bin, change the **Deleting files** preference. To do so, go to the **Home** tab and in the **Environment** section, click **Preferences**. Select **MATLAB > General**. Then, choose from one of the two options in the **Deleting files** section. By default, the **Delete permanently** option is selected.

Note On Mac platforms, the **Deleting files** preference is not applied to files deleted from network drives. All files are deleted permanently.

`delete filename1 ... filenameN` deletes the specified files from disk.

`delete(obj)` deletes the specified object. If `obj` is an array, then `delete` deletes all objects in the array. `obj` remains in the workspace, but is no longer valid.

Examples

Delete Files in Folder

Delete all files in the current folder with a `.mat` extension.

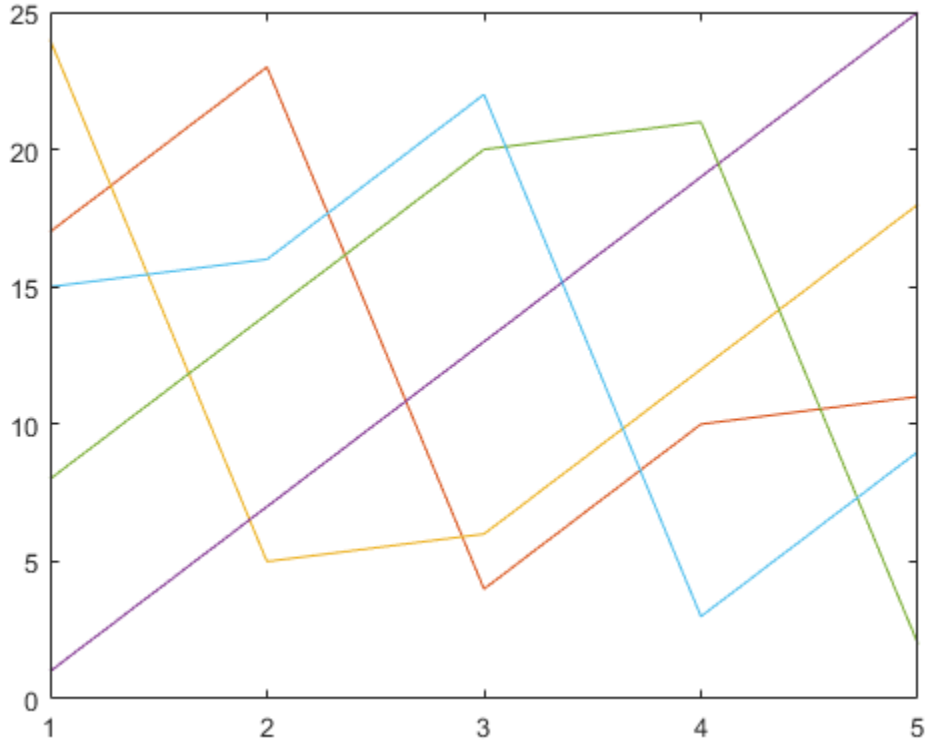
```
delete *.mat
```

Delete Graphics Objects

Delete a graphics object and a graphics object array.

Create a bar chart and plot five lines. Then delete the bar chart.

```
b = bar(1:5);
hold on
P = plot(magic(5));
delete(b)
```



The `Bar` object variable `b` remains in the workspace, but no longer refers to an object.

```
display(b)
```

```
b =
  handle to deleted Bar
```

Delete all the `Line` objects created by `plot`.

```
delete(P)
```

Input Arguments

`filename` — File name
character vector | string scalar

File name to delete, specified as a character vector or string scalar. `filename` can be an absolute or relative path and can include wildcards (*). To delete files from a remote location, `fileName` must contain a full path specified as a uniform resource locator (URL). For more information, see “Work with Remote Data”.

`obj` — Object
single object | array of objects

Object to delete, specified as a single object or an array of objects.

See Also

`clear` | `delete` | `dir` | `recycle` | `rmdir`

Topics

“Specify File Names”
“Work with Remote Data”

Introduced before R2006a

delete

Delete file on FTP server

Syntax

```
delete(ftpobj,filename)
```

Description

`delete(ftpobj,filename)` deletes the specified file from the current folder on the FTP server associated with `ftpobj`.

Examples

Delete File

Connect to an FTP server and delete a file. This example shows a hypothetical FTP session on `ftp.example.com`, a machine that does not exist. If you have an account on an FTP server that grants you permission to upload and delete files on that server, then you can use the `delete` function as shown in this example.

First, connect to the server.

```
ftpobj = ftp('ftp.example.com')
```

```
ftpobj = 

  FTP Object
     host: ftp.example.com
     user: anonymous
      dir: /
     mode: binary
```

Display the contents of the current folder on the FTP server.

```
dir(ftpobj)
```

```
myscript.m                  README.txt                pub
```

Delete the file named `README.txt`.

```
delete(ftpobj,'README.txt')
```

Display the updated contents of the current folder. The `README.txt` file is no longer available for download.

```
dir(ftpobj)
```

```
myscript.m                  pub
```

Input Arguments

`ftpobj` — Connection to FTP server
FTP object

Connection to an FTP server, specified as an FTP object.

`filename` — Name of file to delete
character vector | string scalar

Name of the file to delete, specified as a character vector or string scalar.

See Also

`ftp` | `rmdir`

Introduced before R2006a

deleteproperty

Remove custom property from COM object

Syntax

```
deleteproperty(c,name)
```

Description

`deleteproperty(c,name)` deletes the `name` property from custom properties belonging to the object or interface `c`.

You can delete properties only created with the `addproperty` function.

Input Arguments

`c` — COM object
function handle

COM object, specified as a function handle.

`name` — Property name
character vector

Property name, specified as a character vector.

Example: `'Position'`

Limitations

- COM functions are available on Microsoft Windows systems only.

See Also

Property Inspector | `addproperty` | `get` | `set`

Introduced before R2006a

delevent

Remove event from `timeseries`

Syntax

```
tsout = delevent(tsin,eventname)
tsout = delevent(tsin,eventname,n)
```

Description

`tsout = delevent(tsin,eventname)` removes a `tsdata.event` object from the `tsin.Events` property of a `timeseries` object `tsin`. The `eventname` argument can be a character vector containing the name of a single event or a cell array containing a collection of event names.

`tsout = delevent(tsin,eventname,n)` removes the `n`th `tsdata.event` object from the `tsin.Events` property for the event named `eventname`.

Examples

Remove Event

Create a `timeseries` and an event, and add the event to the `timeseries`.

```
tsin = timeseries((1:5)');
tsevent = tsdata.event('MyEvent',1);
tsin = addevent(tsin,tsevent);
tsin.Events
```

```
    EventData: []
         Name: 'MyEvent'
         Time: 1
        Units: 'seconds'
    StartDate: ''
```

Remove the event named `MyEvent`.

```
tsout = delevent(tsin,'MyEvent');
tsout.Events
```

```
ans =

    1x0 empty handle
```

Input Arguments

`tsin` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

`eventname` — Event name
character vector | cell array

Event name, specified as a character vector containing the name of a `tsdata.event` object or a cell array containing multiple event names.

Data Types: `char` | `cell`

`n` — Event number
scalar integer

Event number, specified as a scalar integer corresponding to the nth `tsdata.event` of a `timeseries` object.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

See Also

`addevent` | `timeseries` | `tsdata.event`

Introduced before R2006a

delsamplefromcollection

Delete sample from `tscollection`

Syntax

```
tscout = delsamplefromcollection(tscin,'Index',ind)
tscout = delsamplefromcollection(tscin,'Value',timevals)
```

Description

`tscout = delsamplefromcollection(tscin,'Index',ind)` deletes samples from a `tscollection` object. `ind` specifies the indices of the `tscin` time vector corresponding to the samples to delete.

`tscout = delsamplefromcollection(tscin,'Value',timevals)` deletes the samples corresponding to the time values in `timeval`.

Examples

Delete Sample

Create a `tscollection` object from two `timeseries` objects. Then, remove the fifth sample.

```
ts1 = timeseries([1.1 2.9 3.7 4.0 3.0]',1:5,...
                 'Name','Acceleration');
ts2 = timeseries([3.2 4.2 6.2 8.5 1.1]',1:5,...
                 'Name','Speed');
tscin = tscollection({ts1;ts2});
tscout = delsamplefromcollection(tscin,'Index',5)
```

```
Time Series Collection Object: unnamed

Time vector characteristics
```

```
      Start time            1 seconds
      End time              4 seconds

Member Time Series Objects:

      Acceleration
      Speed
```

Input Arguments

`tscin` — Input `tscollection`
scalar

Input `tscollection`, specified as a scalar.

`ind` — Time vector indices
scalar | vector

Time vector indices, specified as a scalar or vector.

`timevals` — Sample times
scalar | vector

Sample times, specified as a numeric scalar or vector, or a cell array of date character vectors. Valid date character vectors and strings can have the following forms:

Format	Example
dd-mmm-yyyy HH:MM:SS	01-Mar-2000 15:45:17
dd-mmm-yyyy	01-Mar-2000
mm/dd/yy	03/01/00
mm/dd	03/01
HH:MM:SS	15:45:17
HH:MM:SS PM	3:45:17 PM
HH:MM	15:45
HH:MM PM	3:45 PM

Format	Example
mmm.dd,yyyy HH:MM:SS	Mar.01,2000 15:45:17
mmm.dd,yyyy	Mar.01,2000
mm/dd/yyyy	03/01/2000

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | cell

See Also

addsampletocollection | timeseries | tscollection

Introduced before R2006a

demo

Access product examples in Help browser

Syntax

```
demo
demo type
demo type name
```

Description

`demo` displays a list of featured MATLAB and Simulink examples in the Help browser.

`demo type` lists the examples for the specified product. Valid values for `type` are `matlab` or `simulink`.

`demo type name` lists the examples for products other than MATLAB or Simulink. Valid values for `type` include `matlab`, `simulink`, `toolbox`, or `blockset`.

Examples

Open MATLAB Examples

```
demo matlab
```

Open Statistics and Machine Learning Toolbox Examples

```
demo toolbox statistics
```

Open Communications System Toolbox Examples

```
demo toolbox 'communications system'
```

Open Simulink Control Design Examples

```
demo simulink 'simulink control design'
```

Input Arguments

type — Product name or type
`matlab` (default) | `simulink` | `toolbox` | `blockset`

Product name or type, specified as `'matlab'`, `'simulink'`, `'toolbox'`, or `'blockset'`. For products other than MATLAB or Simulink, you must also specify a `name` input that corresponds to the product name.

Example: `matlab`

Example: `toolbox statistics`

name — Product name other than MATLAB or Simulink
character vector

Product name other than MATLAB or Simulink, specified as a character vector. If `name` requires multiple words, enclose it in single quotes.

Example: `statistics`

Example: `'communications system'`

Tips

- To access third-party and custom examples without using the `demo` command, open the Help browser and navigate to the documentation home page. Then, at the bottom of the page, click **Supplemental Software**.

See Also

`doc` | `echodemo` | `grabcode` | `help`

Introduced before R2006a

DensityPlot Properties

Control appearance and behavior of density plot

Description

`DensityPlot` properties control the appearance and behavior of a `DensityPlot` object. By changing property values, you can modify certain aspects of a density plot.

Properties

Density

`Radius` — Radius of influence on density calculation
numeric scalar

Radius of influence on density calculation, specified as a numeric scalar. When used in a `GeographicAxes`, the value is measured in meters.

`RadiusMode` — Radius calculation
`'auto'` (default) | `'manual'`

Radius calculation, specified as `'auto'` or `'manual'`.

Radius Mode

Value	Description
`'auto'`	`DensityPlot` determines the size of the radius value.
`'manual'`	You specify size of radius. When used in a `GeographicAxes`, this value is measured in meters.

`WeightData` — Weights for data
`[]` (default) | numeric scalar | numeric vector

Weights assigned to data, specified as an empty array, a numeric scalar, or a numeric vector. If you specify a numeric vector, the vector must be the same length as the

`LatitudeData` and `LongitudeData` vectors. The `WeightData` vector is typically additional data you have related to the location data in `LatitudeData` and `LongitudeData`.

`WeightDataSource` — Workspace variable linked to `WeightData`
`''` (default) | character vector | string

Workspace variable linked to `WeightData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `WeightData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `WeightData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning. `geodensityplot` will not render the graph until you have changed all data source properties to appropriate values.

Color and Transparency

`FaceAlpha` — Face transparency
`'interp'` (default) | scalar in range `[0,1]`

Face transparency, specified as one of these values:

- Scalar in range `[0,1]` — Use uniform transparency across all the faces. A value of `1` is fully opaque and `0` is completely transparent. Values between `0` and `1` are semitransparent.
- `'interp'` — Use interpolated transparency for each face. The transparency varies across each face by interpolating the values at the vertices.

`FaceColor` — Face color
`'interp'` | `ColorSpec` value | RGB triplet | ...

Face color, specified as one of the values in this table.

Value	Description
`'interp'`	Use interpolated coloring based on density values. Colors area chosen from the parent axes colormap.
RGB triplet	Three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`.
`ColorSpec`	MATLAB`ColorSpec (Color Specification)`. You can specify some common colors by name.

`FaceColorMode` — Control how `FaceColor` is set

`'auto'` (default) | `'manual'`

Control how the `FaceColor` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the `FaceColor` property by selecting a color from the `ColorOrder` property of the axes.
- `'manual'` — You control the value of the `FaceColor` property manually, either by specifying a color when you call a charting function that creates the object, or by setting the `FaceColor` property on the object after creating it.

If you change the value of the `FaceColor` property manually, MATLAB changes the value of the `FaceColorMode` property to `'manual'`.

Geographic Coordinate Data

`LatitudeData` — Latitude coordinates of data points

real, finite, numeric vector | `[]`

Latitude coordinates of data points, specified as a real, finite, numeric vector in degrees, in the range [-90,90], or as an empty (`[]`) array. `LatitudeData` must be the same size as `LongitudeData` and can contain `NaN`s.

Data Types: `single` | `double`

`LatitudeDataSource` — Variable linked to `LatitudeData`

`''` (default) | character vector | string scalar

Variable linked to `latitudeData`, specified as a character vector or string scalar containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `LatitudeData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `LatitudeData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning. `geodensityplot` will not render the graph until you have changed all data source properties to appropriate values.

`LongitudeData` — Longitude coordinates of data points

real, finite numeric vector | `[]`

Longitude coordinates of data points, specified as a real, finite, numeric vector of values in degrees, in the range `(-Inf,Inf)`, or as an empty (`[]`) array. `LongitudeData` must be the same size as `LatitudeData` and can contain `NaN`s.

Longitudes must span less than (or equal to) 360 degrees.

Data Types: `single` | `double`

`LongitudeDataSource` — Variable linked to `LongitudeData`

`''` (default) | character vector | string

Variable linked to `LongitudeData`, specified as a character vector, or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `LongitudeData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `LongitudeData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning. `geodensityplot` will not render the graph until you have changed all data source properties to appropriate values.

Legend

`DisplayName` — Legend label

`''` (default) | character vector | string scalar

Legend label, specified as a character vector or string scalar. If you do not specify the text, the legend uses a label of the form `'dataN'`. The legend does not display until you call the `legend` command.

`Annotation` — Control for including or excluding object from legend

`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

UIContextMenu — Context menu

empty GraphicsPlaceholder array (default) | ContextMenu object

Context menu, specified as a ContextMenu object. Use this property to display a context menu when you right-click the object. Create the context menu using the uicontextmenu function.

Note If the PickableParts property is set to 'none' or if the HitTest property is set to 'off', then the context menu does not appear.

Selected — Selection state

'off' (default) | 'on'

Selection state, specified as one of these values:

- 'on' — Selected. If you click the object when in plot edit mode, then MATLAB sets its Selected property to 'on'. If the SelectionHighlight property also is set to 'on', then MATLAB displays selection handles around the object.
- 'off' — Not selected.

SelectionHighlight — Display of selection handles

'on' (default) | 'off'

Display of selection handles when selected, specified as one of these values:

- 'on' — Display selection handles when the Selected property is set to 'on'.
- 'off' — Never display selection handles, even when the Selected property is set to 'on'.

Callbacks

ButtonDownFcn — Mouse-click callback

'' (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see "Callback Definition".

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks only when visible. The `Visible` property must be set to `'on'`. The `HitTest` property determines if the `DensityPlot` object responds to the click or if an ancestor does.
- `'all'` — Capture mouse clicks regardless of visibility. The `Visible` property can be set to `'on'` or `'off'`. The `HitTest` property determines if the `DensityPlot` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `DensityPlot` object passes the click to the object below it in the current view of the figure window, which is typically the axes or the figure. The `HitTest` property has no effect.

If you want an object to be clickable when it is underneath other objects that you do not want to be clickable, then set the `PickableParts` property of the other objects to `'none'` so that the click passes through them.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `DensityPlot` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `DensityPlot` object that has one of these:
  - `HitTest` property set to `'on'`

- `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

Note The `PickableParts` property determines if the `DensityPlot` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent

`GeographicAxes` object

Parent, specified as a `GeographicAxes` object.

`Children` — Children

empty `GraphicsPlaceholder` array

The object has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.

- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'densityplot'` (default)

This property is read-only.

Type of graphics object, returned as `'densityplot'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

This property is read-only.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`geodensityplot`

Introduced in R2018b

det

Matrix determinant

Syntax

```
d = det(A)
```

Description

`d = det(A)` returns the determinant of square matrix `A`.

Examples

Calculate Determinant of Matrix

Create a 3-by-3 square matrix, `A`.

```
A = [1 -2 4; -5 2 0; 1 0 3]
```

```
A = 3×3

     1    -2     4
    -5     2     0
     1     0     3
```

Calculate the determinant of `A`.

```
d = det(A)
```

```
d = -32
```

Determine if Matrix Is Singular

Examine why the determinant is not an accurate measure of singularity.

Create a 10-by-10 matrix by multiplying an identity matrix, `eye(10)`, by a small number.

```
A = eye(10)*0.0001;
```

The matrix `A` has very small entries along the main diagonal. However, `A` is *not* singular, because it is a multiple of the identity matrix.

Calculate the determinant of `A`.

```
d = det(A)
```

```
d = 1.0000e-40
```

The determinant is extremely small. A tolerance test of the form `abs(det(A)) < tol` is likely to flag this matrix as singular. Although the determinant of the matrix is close to zero, `A` is actually not ill conditioned. Therefore, `A` is not close to being singular. The determinant of a matrix can be arbitrarily close to zero without conveying information about singularity.

To investigate if `A` is singular, use either the `cond` or `rcond` functions.

Calculate the condition number of `A`.

```
c = cond(A)
```

```
c = 1
```

The result confirms that `A` is not ill conditioned.

Compute Determinant of Inverse of Ill-Conditioned Matrix

Examine how to calculate the determinant of the matrix inverse `A^(-1)`, for an ill-conditioned matrix `A`, without explicitly calculating `A^(-1)`.

Create a 10-by-10 Hilbert matrix, `A`.

```
A = hilb(10);
```

Find the condition number of A.

```
c = cond(A)
```

```
c = 1.6025e+13
```

The large condition number suggests that A is close to being singular, so calculating inv(A) might produce inaccurate results. Therefore, the inverse determinant calculation det(inv(A)) is also inaccurate.

Calculate the determinant of the inverse of A by exploiting the fact that

$$det\left(A^{-1}\right) = \frac{1}{det(A)}$$

```
d1 = 1/det(A)
```

```
d1 = 4.6202e+52
```

This method avoids computing the inverse of the matrix, A.

Calculate the determinant of the exact inverse of the Hilbert matrix, A, using invhilb. Compare the result to d1 to find the relative error in d1.

```
d = det(invhilb(10));
relError = abs(d1-d)/abs(d)
```

```
relError = 1.0443e-04
```

The relative error in d1 is reasonably small. Avoiding the explicit computation of the inverse of A minimizes it.

For comparison, also calculate the determinant of the inverse of A by explicitly calculating the inverse. Compare the result to d to see the relative error.

```
d2 = det(inv(A));
relError2 = abs(d2-d)/abs(d)
```

```
relError2 = 2.2039e-05
```

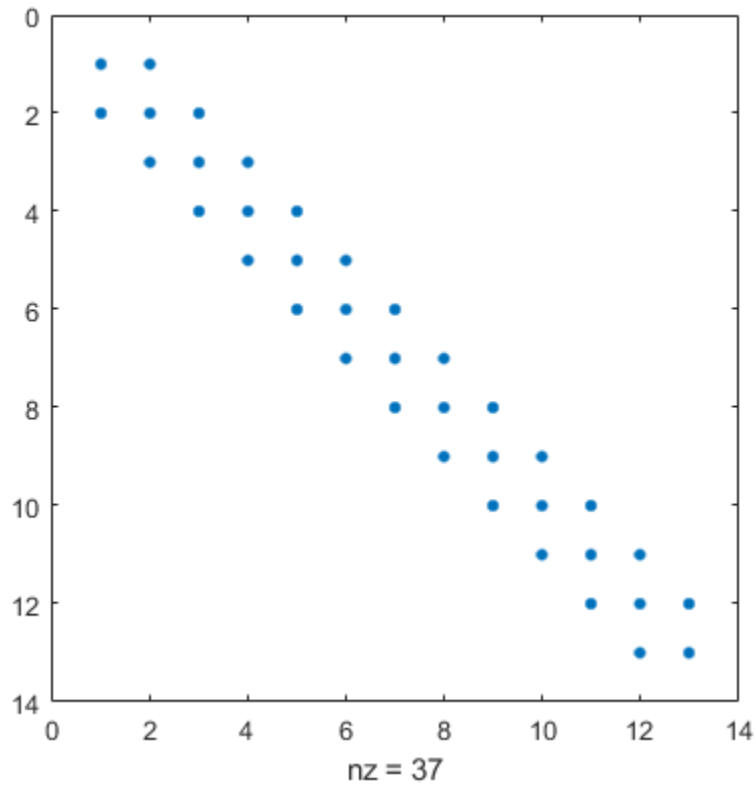
The relative error in the calculation of d2 is many orders of magnitude larger than that of d1.

Find Determinant of Singular Matrix

Examine a matrix that is exactly singular, but which has a large nonzero determinant. In theory, the determinant of any singular matrix is zero, but because of the nature of floating-point computation, this ideal is not always achievable.

Create a 13-by-13 diagonally dominant singular matrix A and view the pattern of nonzero elements.

```
A = diag([24 46 64 78 88 94 96 94 88 78 64 46 24]);
S = diag([-13 -24 -33 -40 -45 -48 -49 -48 -45 -40 -33 -24],1);
A = A + S + rot90(S,2);
spy(A)
```



A is singular because the rows are linearly dependent. For instance, `sum(A)` produces a vector of zeros.

Calculate the determinant of A.

```
d = det(A)
```

```
d = 1.0597e+05
```

The determinant of A is quite large despite the fact that A is singular. In fact, the determinant of A should be exactly zero! The inaccuracy of d is due to an aggregation of round-off errors in the MATLAB® implementation of the LU decomposition, which `det` uses to calculate the determinant. This result demonstrates a few important aspects of calculating numeric determinants. See the "Limitations" on page 1-3282 section for more details.

Input Arguments

A — Input matrix
square numeric matrix

Input matrix, specified as a square numeric matrix.

Data Types: `single` | `double`
Complex Number Support: Yes

Limitations

Avoid using `det` to examine if a matrix is singular because of the following limitations. Use `cond` or `rcond` instead.

Limitation	Result
The magnitude of the determinant is typically unrelated to the condition number of a matrix.	The determinant of a matrix can be arbitrarily large or small without changing the condition number.

Limitation	Result
`det` uses the LU decomposition to calculate the determinant, which is susceptible to floating-point round-off errors.	The determinant calculation is sometimes numerically unstable. For example, `det` can produce a large-magnitude determinant for a singular matrix, even though it should have a magnitude of 0.

Algorithms

`det` computes the determinant from the triangular factors obtained by Gaussian elimination with the `lu` function.

```
[L,U] = lu(X)
s =  det(L)      % This is always +1 or -1
det(X) = s*prod(diag(U))
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`cond` | `condest` | `inv` | `lu` | `mldivide` | `rcond` | `rref`

Introduced before R2006a

details

Display array details

Syntax

```
details(A)
```

Description

`details(A)` displays detailed information about the array, `A`.

When `A` is a MATLAB object array, `details` displays more information than the default display. This information includes:

- Fully qualified class name, including package names
- Link to class documentation
- Link to `handle` class documentation for classes that subclass `handle`
- List of all properties that have public get access
- List of property values if the array is scalar
- Link to list of public methods
- Link to list of events
- Link to list of all nonhidden superclasses

Examples

Display Object Details

Display object details for a class that overloads its own object display. The `details` function never calls overloaded display methods. Therefore, you can use this function to obtain information about the object array in all cases.

Suppose `PolyNom` is a class that provides a specialized default display for polynomials. Use `details` to display information about the object.

Create an object using the polynomial coefficients:

```
pn = PolyNom([1,2,3,0,4])
```

The overloaded `disp` method displays the code for evaluating the polynomial:

```
pn =

x^4 + 2*x^3 + 3*x^2 + 4
```

Calling the `details` function provides information about the object:

```
details(pn)
```

```
 PolyNom with properties:

    coef: [1 2 3 0 4]

  Methods
```

Input Arguments

A — Input array
scalar or nonscalar array of any type

Input array, specified as a scalar or nonscalar array of any type. The `details` function displays detailed information about this array.

See Also

`classdef` | `disp` | `display`

Introduced in R2013b

detectImportOptions

Create import options based on file content

Syntax

```
opts = detectImportOptions(filename)
opts = detectImportOptions(filename,Name,Value)
```

Description

`opts = detectImportOptions(filename)` locates a table in a file and returns the import options for importing the table. You can modify the options object and use it with `readtable` to control how MATLAB imports tabular data. The type of the options returned depends on the file extension. For example, the function returns a `SpreadsheetImportOptions` object if `filename` is a spreadsheet file. However, the function returns a `DelimitedTextImportOptions` or `FixedWidthImportOptions` object if `filename` is a text file.

`opts = detectImportOptions(filename,Name,Value)` locates a table in a file with the help of additional parameters specified by one or more `Name,Value` pair arguments.

Examples

Detect and Use Import Options for Spreadsheet Files

Detect import options for a spreadsheet file, specify the variables to import, and then read the data.

Create an import options object from a file.

```
opts = detectImportOptions('patients.xls')
```

```
opts 
  SpreadsheetImportOptions with properties:
```

```
  Sheet Properties:
                      Sheet: ''

  Replacement Properties:
                 MissingRule: 'fill'
             ImportErrorRule: 'fill'

  Variable Import Properties: Set types by name using setvartype
               VariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
               VariableTypes: {'char', 'char', 'double' ... and 7 more}
       SelectedVariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
             VariableOptions: Show all 10 VariableOptions
   Access VariableOptions sub-properties using setvaropts/getvaropts
       PreserveVariableNames: false

  Range Properties:
                   DataRange: 'A2' (Start Cell)
          VariableNamesRange: 'A1'
               RowNamesRange: ''
          VariableUnitsRange: ''
   VariableDescriptionsRange: ''
   To display a preview of the table, use preview
```

Modify the options object to specify which variables to import.

```
opts.SelectedVariableNames = {'Systolic','Diastolic'};
```

Use `readtable` along with the options object to import the specified variables.

```
T = readtable('patients.xls',opts);
summary(T)
```

```
Variables:

    Systolic: 100x1 double

        Values:

            Min          109
            Median       122
            Max          138

    Diastolic: 100x1 double
```

```
        Values:

            Min            68
            Median       81.5
            Max            99
```

Detect and Use Import Options for Text Files

Create import options, tailor the data types for multiple variables, and then read the data.

Create an import options object from a text file.

```
opts = detectImportOptions('airlinesmall.csv')
```

```
opts = 
  DelimitedTextImportOptions with properties:

   Format Properties:
                   Delimiter: {','}
                  Whitespace: '\b\t '
                  LineEnding: {'\n'  '\r'  '\r\n'}
                CommentStyle: {}
    ConsecutiveDelimitersRule: 'split'
        LeadingDelimitersRule: 'keep'
                EmptyLineRule: 'skip'
                     Encoding: 'windows-1252'

   Replacement Properties:
                 MissingRule: 'fill'
             ImportErrorRule: 'fill'
            ExtraColumnsRule: 'addvars'

   Variable Import Properties: Set types by name using setvartype
                VariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
                VariableTypes: {'double', 'double', 'double' ... and 26 more}
        SelectedVariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
              VariableOptions: Show all 29 VariableOptions
    Access VariableOptions sub-properties using setvaropts/getvaropts
        PreserveVariableNames: false

   Location Properties:
```

```
                    DataLines: [2 Inf]
            VariableNamesLine: 1
               RowNamesColumn: 0
            VariableUnitsLine: 0
     VariableDescriptionsLine: 0
    To display a preview of the table, use preview
```

Examine the `Type` property of variables `TaxiIn` and `TaxiOut`.

```
getvaropts(opts,{'TaxiIn','TaxiOut'})
```

```
ans = 
  1x2 TextVariableImportOptions array with properties:

    Name
    Type
    FillValue
    TreatAsMissing
    QuoteRule
    Prefixes
    Suffixes
    EmptyFieldRule
    WhitespaceRule
```

Change the type of the variables `TaxiIn` and `TaxiOut` to `double`.

```
 opts = setvartype(opts,{'TaxiIn','TaxiOut'},'double');
```

Specify the subset of variables to import and examine.

```
opts.SelectedVariableNames = {'TaxiIn','TaxiOut'};
```

Use the `readtable` function along with the options object to import the selected variables. Display a summary of the table.

```
T = readtable('airlinesmall.csv',opts);
summary(T)
```

```
Variables:

    TaxiIn: 123523x1 double

        Values:
```

```
            Min             0
            Median          5
            Max             1451
            NumMissing      37383

    TaxiOut: 123523x1 double

        Values:

            Min             0
            Median          13
            Max             755
            NumMissing      37364
```

Designate Data Type for Imported Text Data

Import text data as a string data type by specifying import options.

Create an options object for the file.

```
opts = detectImportOptions('outages.csv');
```

Specify which variables to import using `readtable`, and then show a summary. The data type of the selected variables is `char`.

```
opts.SelectedVariableNames = {'Region','Cause'};
T = readtable('outages.csv',opts);
summary(T)
```

```
Variables:

    Region: 1468x1 cell array of character vectors

    Cause: 1468x1 cell array of character vectors
```

Import text data as a `string` data type, and then create import options by specifying the `TextType` name-value pair.

```
opts = detectImportOptions('outages.csv','TextType','string');
```

Specify which variables to import using `readtable`, and then show a summary. The data type of the selected variables is now `string`.

```
opts.SelectedVariableNames = {'Region','Cause'};
T = readtable('outages.csv',opts);
summary(T)
```

```
Variables:

    Region: 1468x1 string

    Cause: 1468x1 string
```

Input Arguments

`filename` — Name of file to read
character vector | string scalar

Name of the file to read, specified as a character vector or string scalar.

Depending on the location of your file, `filename` can take on one of these forms.

Location	Form
Current folder or folder on the MATLAB path	Specify the name of the file in `filename`. **Example:** `'myFile.txt'`
File in a folder	If the file is not in the current folder or in a folder on the MATLAB path, then specify the full or relative path name in `filename`. **Example:** `'C:\myFolder\myFile.xlsx'` **Example:** `'\imgDir\myFile.txt'`

<table>
<tr><th>Location</th><th>Form</th></tr>
<tr><td>Remote Location</td><td>If the file is stored at a remote location, then <code>filename</code> must contain the full path of the file specified as a uniform resource locator (URL) of the form:

<code>scheme_name://path_to_file/my_file.ext</code>

Based on your remote location, <code>scheme_name</code> can be one of the values in this table.
<table>
<tr><th>Remote Location</th><th><code>scheme_name</code></th></tr>
<tr><td>Amazon S3</td><td><code>s3</code></td></tr>
<tr><td>Windows Azure Blob Storage</td><td><code>wasb</code>, <code>wasbs</code></td></tr>
<tr><td>HDFS</td><td><code>hdfs</code></td></tr>
</table>
For more information, see “Work with Remote Data”.

Example: <code>'s3://bucketname/path_to_file/my_file.csv'</code></td></tr>
</table>

If `filename` includes the file extension, then `detectImportOptions` determines the file format from the extension. Otherwise, you must specify the `'FileType'` name-value pair to indicate the type of file.

The `detectImportOptions` function supports these file extensions: `.txt`, `.dat`, `.csv`, `.xls`, `.xlsb`, `.xlsm`, `.xlsx`, `.xltm`, `.xltx`, and `.ods`.

Note File extensions `.xlsb` and `.ods` are only supported on platforms with Excel for Windows.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'FileType','spreadsheet'`

Parameters

`FileType` — Type of file
`'spreadsheet'` | `'text'` | `'delimitedtext'` | `'fixedwidth'`

Type of file, specified as the comma-separated pair consisting of `'FileType'` and one of these values.

Value	Import Options for File
`'spreadsheet'`	Return a `SpreadsheetImportOptions` object.
`'text'`	Return a `DelimitedTextImportOptions` or `FixedWidthImportOptions` object depending on the layout of the text file.
`'delimitedtext'`	Return a `DelimitedTextImportOptions` object. Use this option for text files that have columns separated by delimiters.
`'fixedwidth'`	Return a `FixedWidthImportOptions` object. Use this option for text files that have columns of fixed widths.

Use the `'FileType'` name-value pair argument when `filename` does not include the file extension, or when the extension is not one of these:

- `.txt`, `.dat`, or `.csv` for text files
- `.xls`, `.xlsb`, `.xlsm`, `.xlsx`, `.xltm`, `.xltx`, or `.ods` for spreadsheet files

File extensions `.xlsb` and `.ods` are only supported on platforms with Excel for Windows.

Example: `'FileType','text'`

Data Types: `char` | `string`

`TextType` — Type for imported text data
`'char'` (default) | `'string'`

Type for imported text data, specified as the comma-separated pair consisting of `'TextType'` and either `'char'` or `'string'`.

- `'char'` — Import text data into MATLAB as character vectors.

- `'string'` — Import text data into MATLAB as string arrays.

Example: `'TextType','char'`

`DatetimeType` — Type for imported date and time data
`'datetime'` (default) | `'text'` | `'exceldatenum'` (spreadsheet files only)

Type for imported date and time data, specified as the comma-separated pair consisting of `'DatetimeType'` and one of these values: `'datetime'`, `'text'`, or `'exceldatenum'`. The value `'exceldatenum'` is applicable only for spreadsheet files, and is not valid for text files.

Value	Type for Imported Date and Time Data
`'datetime'`	MATLAB `datetime` data type For more information, see `datetime`.
`'text'`	If `'DatetimeType'` is specified as `'text'`, then the type for imported date and time data depends on the value specified in the `'TextType'` parameter: • If `'TextType'` is set to `'char'`, then the importing function returns dates as a cell array of character vectors. • If `'TextType'` is set to `'string'`, then the importing function returns dates as an array of strings.
`'exceldatenum'`	Excel serial date numbers A serial date number is a single number equal to the number of days from a given reference date. Excel serial date numbers use a different reference date than MATLAB serial date numbers. For more information on Excel dates, see `https://support.microsoft.com/en-us/kb/214330`.

Data Types: `char` | `string`

`ExpectedNumVariables` — Expected number of variables
positive integer

Expected number of variables, specified as the comma-separated pair consisting of `'ExpectedNumVariables'` and a positive integer. If unspecified, the importing function automatically detects the number of variables.

Data Types: `single` | `double`

Range — Portion of data to read
character vector | string scalar | numeric vector

Portion of the data to read from text or spreadsheet files, specified as the comma separated pair consisting of `'Range'` and a character vector, string scalar, or numeric vector in one of these forms.

Ways to specify Range	Description
Starting Cell `'Cell'` or `[row col]`	Specify the starting cell for the data as a character vector or string scalar or a two element numeric vector. • Character vector or string scalar containing a column letter and row number using Excel `A1` notation. For example, `A5` is the identifier for the cell at the intersection of column `A` and row `5`. • Two element numeric vector of the form `[row col]` indicating the starting row and column. Using the starting cell, the importing function automatically detects the extent of the data by beginning the import at the start cell and ending at the last empty row or footer range. **Example:** `'A5'` or `[5 1]`

Ways to specify Range	Description
Rectangular Range `'Corner1:Corner2'` or `[r1 c1 r2 c2]`	Specify the exact range to read using the rectangular range in one of these forms. • `'Corner1:Corner2'` — Specify the range using `Corner1` and `Corner2` which are the two opposing corners that define the region to read in Excel `A1` notation. For example, `'C2:N15'`. • `[r1 c1 r2 c2]` — Specify the range using a four element numeric vector containing start-row, start-column, end-row, and end-column. For example, `[2 3 15 13]`. The importing function only reads the data contained in the specified range. Any empty fields within the specified range are imported as missing cells.
Row Range or Column Range `'Row1:Row2'` or `'Column1:Column2'`	Specify the range by identifying the beginning and ending rows using Excel row numbers. Using the specified row range, the importing function automatically detects the column extent by reading from the first nonempty column to the end of the data, and creates one variable per column. **Example:** `'5:500'` Alternatively, specify the range by identifying the beginning and ending columns using Excel column letters or numbers. Using the specified column range, the import function automatically detects the row extent by reading from the first nonempty row to the end of the data or the footer range. The number of columns in the specified range must match the number specified in the `ExpectedNumVariables` property. **Example:** `'A:K'`

Ways to specify Range	Description
Starting Row Number `n`	Specify the first row containing the data using the positive scalar row index. Using the specified row index, the importing function automatically detects the extent of the data by reading from the specified first row to the end of the data or the footer range. **Example:**5
Excel's Named Range `'NamedRange'`	In Excel, you can create names to identify ranges in the spreadsheet. For instance, you can select a rectangular portion of the spreadsheet and call it `'myTable'`. If such named ranges exist in a spreadsheet, then the importing function can read that range using its name. **Example:** `'Range','myTable'`
Unspecified or Empty `''`	If unspecified, the importing function automatically detects the used range. **Example:** `'Range',''` **Note:** *Used Range* refers to the rectangular portion of the spreadsheet that actually contains data. The importing function automatically detects the used range by trimming any leading and trailing rows and columns that do not contain data. Text that is only white space is considered data and is captured within the used range.

Data Types: `char` | `string` | `double`

`NumHeaderLines` — Number of header lines
positive integer

Number of header lines in the file, specified as the comma-separated pair consisting of `'NumHeaderLines'` and a positive integer. If unspecified, the importing function automatically detects the number of header lines in the file.

Example: `'NumHeaderLines',7`

Data Types: `single` | `double`

`ReadVariableNames` — Read first row as variable names
`true` | `false`

Indicator for reading the first row as variable names, specified as the comma-separated pair consisting of `'ReadVariableNames'` and either `true` or `false`. If unspecified, `readtable` automatically detects the presence of variable names.

Indicator	Description
`true`	Use when the first row of the region to read contains the variable names for the table. `readtable` creates a variable, with the detected variable name, for each column in `T`.
`false`	Use when the first row of the region to read contains data in the table. `readtable` creates default variable names of the form `'Var1',...,'VarN'`, where `N` is the number of variables.
unspecified	When left unspecified, the importing function automatically detects `true` or `false` and proceeds accordingly.

Data Types: `logical`

`PreserveVariableNames` — Flag to preserve variable names
`false` (default) | `true`

Flag to preserve variable names, specified as the comma-separated pair consisting of `PreserveVariableNames` and either `true`, or `false`.

- `true` — Preserve variable names that are not valid MATLAB identifiers such as variable names that include spaces and non-ASCII characters.
- `false` — Convert invalid variable names (as determined by the `isvarname` function) to valid MATLAB identifiers.

Starting in R2019b, variable names and row names can include any characters, including spaces and non-ASCII characters. Also, they can start with any characters, not just letters. Variable and row names do not have to be valid MATLAB identifiers (as determined by the `isvarname` function). To preserve these variable names and row names, set `PreserveVariableNames` to `true`.

`ReadRowNames` — Indicator for reading the first column as row names
`false` (default) | `true`

Indicator for reading first column as row names, specified as the comma-separated pair consisting of `'ReadRowNames'` and either `false` or `true`.

Indicator	Description
`false`	Use when the first column of the region to read contains data, and not the row names for the table.
`true`	Use when the first column of the region to read contains the row names for the table.
unspecified	When left unspecified, the importing function assumes `false`.

Data Types: `logical`

`MissingRule` — Procedure to manage missing data
`'fill'` | `'error'` | `'omitrow'` | `'omitvar'`

Procedure to manage missing data, specified as one of the values in this table.

Missing Rule	Behavior
`'fill'`	Replace missing data with the contents of the `FillValue` property. The `FillValue` property is specified in the `VariableImportOptions` object of the variable being imported. For more information on accessing the `FillValue` property, see `getvaropts`.
`'error'`	Stop importing and display an error message showing the missing record and field.
`'omitrow'`	Omit rows that contain missing data.
`'omitvar'`	Omit variables that contain missing data.

Example: `opts.MissingRule = 'omitrow';`uint32

Data Types: `char` | `string`

`ImportErrorRule` — Procedure to handle import errors
`'fill'` | `'error'` | `'omitrow'` | `'omitvar'`

Procedure to handle import errors, specified as one of the values in this table.

Import Error Rule	Behavior
`'fill'`	Replace the data where the error occurred with the contents of the `FillValue` property. The `FillValue` property is specified in the `VariableImportOptions` object of the variable being imported. For more information on accessing the `FillValue` property, see `getvaropts`.
`'error'`	Stop importing and display an error message showing the error-causing record and field.
`'omitrow'`	Omit rows where errors occur.
`'omitvar'`	Omit variables where errors occur.

Example: `opts.ImportErrorRule = 'omitvar';`

Data Types: `char` | `string`

Parameters for Spreadsheet Files Only

Sheet — Sheet to read from

`''` empty character array (default) | character vector | string scalar | positive scalar integer

Sheet to read from, specified as an empty character array, a character vector or string scalar containing the sheet name, or a positive scalar integer denoting the sheet index. Based on the value specified for the `Sheet` property, the import function behaves as described in the table.

Specification	Behavior
`''` (default)	Import data from the first sheet.
Name	Import data from the matching sheet name, regardless of order of sheets in the spreadsheet file.
Integer	Import data from sheet in the position denoted by the integer, regardless of the sheet names in the spreadsheet file.

Data Types: `char` | `string` | `single` | `double`

`DataRange` — Location of data

character vector | string scalar | positive scalar integer | array of positive scalar integers | cell array of character vector | string array

Location of data to be imported, specified as a character vector, string scalar, cell array of character vectors, string array, positive scalar integer or an `N`-by-`2` array of positive scalar integers. Specify `DataRange` using one of these forms.

Specified by	Behavior
`'Cell'` or *`n`* Starting Cell or Starting Row	Specify the starting cell for the data, using Excel `A1` notation. For example, `A5` is the identifier for the cell at the intersection of column `A` and row `5`. Using the starting cell, the importing function automatically detects the extent of the data, by beginning the import at the start cell and ending at the last empty row or footer range. Alternatively, specify the first row containing the data using the positive scalar row index. Using the specified row index, the importing function automatically detects the extent of the data by reading from the specified first row to the end of the data or the footer range. **Example:** `'A5'` or `5`
`'Corner1:Corner2'` Rectangular Range	Specify the exact range to read using the rectangular range form, where *`Corner1`* and *`Corner2`* are two opposing corners that define the region to read. The importing function only reads the data contained in the specified range. Any empty fields within the specified range are imported as missing cells. The number of columns must match the number specified in the `NumVariables` property. **Example:** `'A5:K50'`

Specified by	Behavior
'*Row1:Row2*' or '*Column1:Column2*' Row Range or Column Range	Specify the range by identifying the beginning and ending rows using Excel row numbers. Using the specified row range, the importing function automatically detects the column extent by reading from the first nonempty column to the end of the data, and creates one variable per column. **Example:** `'5:500'` Alternatively, specify the range by identifying the beginning and ending columns using Excel column letters or numbers. Using the specified column range, the import function automatically detects the row extent by reading from the first nonempty row to the end of the data or the footer range. The number of columns in the specified range must match the number specified in the `NumVariables` property. **Example:** `'A:K'`
[n1 n2; n3 n4;...] Multiple Row Ranges	Specify *multiple row ranges* to read with an `N`-by-`2` array containing `N` different row ranges. A valid array of multiple row ranges must: • Specify line ranges in an increasing order, that is the first row range specified in the array appears in the file before the other row ranges. • Contain only non-overlapping row ranges. Use of `Inf` is only supported to indicate the last range in the numeric array specifying multiple row ranges. For example, `[1 3; 5 6; 8 Inf]`. **Example:** `[1 3; 5 6; 8 Inf]`

Specified by	Behavior
`''` Unspecified or Empty	Do not fetch any data. **Example:** `''`

Data Types: `char` | `string` | `cell` | `single` | `double`

`RowNamesRange` — Location of row names
character vector | string scalar | positive scalar integer | `''` empty character array

Location of row names, specified as a character vector, string scalar, positive scalar integer, or an empty character array. Specify `RowNamesRange` as one of the values in this table.

Specified by	Behavior
`'Cell'`	Specify the starting cell for the data, using Excel `A1` notation. For example, `A5` is the identifier for the cell at the intersection of column `A` and row `5`. The importing function identifies a name for each variable in the data. **Example:** `'A5'`
`'Corner1:Corner2'` Rectangular Range	Specify the exact range to read using the rectangular range form, where *`Corner1`* and *`Corner2`* are two opposing corners that define the region to read. The number of rows contained in `RowNamesRange` must match the number of data rows, and the range indicated by `RowNamesRange` must span only one column. **Example:** `'A5:A50'`
`'Row1:Row2'` Row Range	Specify range by identifying the beginning and ending rows using Excel row numbers. Row names must be in a single column. **Example:** `'5:50'`

Specified by	Behavior
n Number Index	Specify the column containing the row names using a positive scalar column index. **Example:** 5
`''` Unspecified or Empty	Indicate that there are no row names. **Example:** `''`

Data Types: `char` | `single` | `double`

`VariableNamesRange` — Location of variable names
character vector | string scalar | positive scalar integer | `''` empty character array

Location of variable names, specified as a character vector, string scalar, positive scalar integer, or an empty character array. Specify `VariableNamesRange` as one of the values in this table.

Specified by	Behavior
'*Cell*'	Specify the starting cell for the data, using Excel `A1` notation. For example, `A5` is the identifier for the cell at the intersection of column `A` and row `5`. The importing function reads a name for each variable in the data. **Example:** `'A5'`
'*Corner1:Corner2*' Rectangular Range	Specify the exact range to read using the rectangular range form, where *Corner1* and *Corner2* are two opposing corners that define the region to read. The number of columns must match the number specified in the `NumVariables` property, and the range must span only one row. **Example:** `'A5:K5'`

Specified by	Behavior
'*Row1:Row2*' Row Range	Specify range by identifying the beginning and ending rows using Excel row numbers. Must be a single row. **Example:** `'5:5'`
n Number Index	Specify the row containing the variable names using a positive scalar row index. **Example:** 5
`''` Unspecified or Empty	Indicate that there are no variable names. **Example:** `''`

Data Types: `char` | `single` | `double`

`VariableUnitsRange` — Location of variable units
character vector | string scalar | positive scalar integer | `''` empty character array

Location of variable units, specified as a character vector, string scalar, positive scalar integer, or an empty character array. Specify `VariableUnitsRange` as one of the values in this table.

Specified by	Behavior
'*Cell*'	Specify the starting cell for the data, using Excel `A1` notation. For example, `A5` is the identifier for the cell at the intersection of column `A` and row `5`. The importing function reads a unit for each variable in the data. **Example:** `'A5'`

Specified by	Behavior
`'Corner1:Corner2'` Rectangular Range	Specify the exact range to read using the rectangular range form, where *`Corner1`* and *`Corner2`* are two opposing corners that define the region to read. The number of columns must match the number specified in the `NumVariables` property, and the range must span only one row. **Example:** `'A5:K5'`
`'Row1:Row2'` Row Range	Specify range by identifying the beginning and ending rows using Excel row numbers. Must be a single row. **Example:** `'5:5'`
`n` Number Index	Specify the row containing the data units using a positive scalar row index. **Example:** 5
`''` Unspecified or Empty	Indicate that there are no variable units. **Example:** `''`

Data Types: `char` | `string` | `single` | `double`

`VariableDescriptionsRange` — Location of variable descriptions
character vector | string scalar | `''` empty character array

Location of variable descriptions, specified as a character vector, string scalar, positive scalar integer, or an empty character array. Specify `VariableDescriptionRange` as one of the values in this table.

Specified by	Behavior
`'Cell'`	Specify the starting cell for the data, using Excel `A1` notation. For example, `A5` is the identifier for the cell at the intersection of column `A` and row `5`. The importing function reads a description for each variable in the data. **Example:** `'A5'`
`'Corner1:Corner2'` Rectangular Range	Specify the exact range to read using the rectangular range form, where *`Corner1`* and *`Corner2`* are two opposing corners that define the region to read. The number of columns must match the number specified in the `NumVariables` property, and the range must span only one row. **Example:** `'A5:K5'`
`'Row1:Row2'` Row Range	Specify range by identifying the beginning and ending rows using Excel row numbers. Must be a single row. **Example:** `'5:5'`
`n` Number Index	Specify the row containing the descriptions using a positive scalar row index. **Example:** `5`
`''` Unspecified or Empty	Indicate that there are no variable descriptions. **Example:** `''`

Data Types: `char` | `string` | `single` | `double`

Parameters for Text Files Only

`Delimiter` — Field delimiter characters

character vector | string scalar | cell array of character vectors | string array

Field delimiter characters in a delimited text file, specified as a character vector, string scalar, cell array of character vectors, or string array.

Example: `'Delimiter','|'`

Example: `'Delimiter',{';','*'}`

Data Types: `char` | `string` | `cell`

`LeadingDelimitersRule` — Procedure to manage leading delimiters
`'keep'` | `'ignore'` | `'error'`

Procedure to manage leading delimiters in a delimited text file, specified as one of the values in this table.

Leading Delimiters Rule	Behavior
`'keep'`	Keep the delimiter.
`'ignore'`	Ignore the delimiter.
`'error'`	Return an error and abort the import operation.

`ConsecutiveDelimitersRule` — Procedure to handle consecutive delimiters
`'split'` | `'join'` | `'error'`

Procedure to handle consecutive delimiters in a delimited text file, specified as one of the values in this table.

Consecutive Delimiters Rule	Behavior
`'split'`	Split the consecutive delimiters into multiple fields.
`'join'`	Join the delimiters into one delimiter.
`'error'`	Return an error and abort the import operation.

Data Types: `char` | `string`

`VariableWidths` — Field widths of variables
vector of positive integer values

Field widths of variables in a fixed-width text file, specified as a vector of positive integer values. Each positive integer in the vector corresponds to the number of characters in a field that makes up the variable. The `VariableWidths` property contains an entry corresponding to each variable specified in the `VariableNames` property.

`Whitespace` — Characters to treat as white space
character vector | string scalar

Characters to treat as white space, specified as a character vector or string scalar containing one or more characters.

Example: `'Whitespace',' _'`

Example: `'Whitespace','?!.,'`

`LineEnding` — End-of-line characters
`{'\n','\r','\r\n'}` (default) | character vector | string scalar | cell array of character vectors | string array

End-of-line characters, specified as a character vector, string scalar, cell array of character vectors, or string array.

Example: `'LineEnding','\n'`

Example: `'LineEnding','\r\n'`

Example: `'LineEnding',{'\b',':'}`

Data Types: `char` | `string` | `cell`

`Encoding` — Character encoding scheme
`'system'` | `'UTF-8'` | `'ISO-8859-1'` | `'windows-1251'` | `'windows-1252'` | ...

Character encoding scheme associated with the file, specified as the comma-separated pair consisting of `'Encoding'` and `'system'` or a standard character encoding scheme name, such as one of the values in this table.

`'Big5'`	`'ISO-8859-1'`	`'windows-847'`
`'Big5-HKSCS'`	`'ISO-8859-2'`	`'windows-949'`
`'CP949'`	`'ISO-8859-3'`	`'windows-1250'`
`'EUC-KR'`	`'ISO-8859-4'`	`'windows-1251'`
`'EUC-JP'`	`'ISO-8859-5'`	`'windows-1252'`
`'EUC-TW'`	`'ISO-8859-6'`	`'windows-1253'`
`'GB18030'`	`'ISO-8859-7'`	`'windows-1254'`
`'GB2312'`	`'ISO-8859-8'`	`'windows-1255'`

'GBK'	'ISO-8859-9'	'windows-1256'
'IBM866'	'ISO-8859-11'	'windows-1257'
'KOI8-R'	'ISO-8859-13'	'windows-1258'
'KOI8-U'	'ISO-8859-15'	'US-ASCII'
	'Macintosh'	'UTF-8'
	'Shift_JIS'	

Example: `'Encoding','system'` uses the system default encoding.

Data Types: `char` | `string`

`CommentStyle` — Style of comments
character vector | string scalar | cell array of character vectors | string array

Style of comments, specified as a character vector, string scalar, cell array of character vectors, or string array.

For example, to ignore the text following a percent sign on the same line, specify `CommentStyle` as `'%'`.

Example: `'CommentStyle',{'/*'}`

Data Types: `char` | `string` | `cell`

`DurationType` — Output data type of duration data
`'duration'` (default) | `'text'`

Output data type of duration data from text files, specified as the comma-separated pair consisting of `'DurationType'` and either `'duration'` or `'text'`.

Value	Type for Imported Duration Data
`'duration'`	MATLAB `duration` data type For more information, see `duration`.

Value	Type for Imported Duration Data
`'text'`	If `'DurationType'` is specified as `'text'`, then the type for imported duration data depends on the value specified in the `'TextType'` parameter: • If `'TextType'` is set to `'char'`, then the importing function returns duration data as a cell array of character vectors. • If `'TextType'` is set to `'string'`, then the importing function returns duration data as an array of strings.

Data Types: `char` | `string`

`ExtraColumnsRule` — Procedure to handle extra columns
`'addvars'` | `'ignore'` | `'wrap'` | `'error'`

Procedure to handle extra columns in the data, specified as one of the values in this table.

Extra Columns Rule	Behavior
`'addvars'`	To import extra columns, create new variables. If there are `N` extra columns, then import new variables as `'ExtraVar1', 'ExtraVar2',..., 'ExtraVarN'`. NOTE: The extra columns are imported as text with data type`char`.
`'ignore'`	Ignore the extra columns of data.
`'wrap'`	Wrap the extra columns of data to new records. This action does not change the number of variables.
`'error'`	Display an error message and abort the import operation.

Data Types: `char` | `string`

`TreatAsMissing` — Text to interpret as missing data
character vector | string scalar | cell array of character vectors | string array

Text to interpret as missing data, specified as a character vector, string scalar, cell array of character vectors, or string array.

When the importing function finds missing instances, it uses the specification in the `MissingRule` property to determine the appropriate action.

Example: `'TreatAsMissing',{'NA','TBD'}` instructs the importing function to treat any occurrence of `NA` or `TBD` as a missing fields.

Data Types: `char` | `string` | `cell`

`DateLocale` — Locale for reading dates
character vector | string scalar

Locale for reading dates, specified as the comma-separated pair consisting of `'DateLocale'` and a character vector or a string scalar of the form *xx_YY*, where:

- *YY* is an uppercase ISO 3166-1 alpha-2 code indicating a country.
- *xx* is a lowercase ISO 639-1 two-letter code indicating a language.

For a list of common values for the locale, see the `Locale` name-value pair argument for the `datetime` function.

When using the `%D` format specifier to read text as `datetime` values, use `DateLocale` to specify the locale in which the importing function should interpret month and day-of-week names and abbreviations.

If you specify the `DateLocale` argument in addition to `opts` the import options, then the importing function uses the specified value for the `DateLocale` argument, overriding the locale defined in the import options.

Example: `'DateLocale','ja_JP'`

`ThousandsSeparator` — Characters that indicate the thousands grouping
character vector | string scalar

Characters that indicate the thousands grouping in numeric variables, specified as a character vector or string scalar. The thousands grouping characters act as visual separators, grouping the number at every three place values. The importing function uses the characters in the `ThousandsSeparator` property to interpret the numbers being imported.

Data Types: `char` | `string`

`DecimalSeparator` — Characters indicating decimal separator
character vector | string scalar

Characters indicating the decimal separator in numeric variables, specified as a character vector or string scalar. The importing function uses the `DecimalSeparator` property to distinguish the integer part of a number from the decimal part.

When converting to integer data types, numbers with a decimal part are rounded to the nearest integer.

Data Types: `char` | `string`

`TrimNonNumeric` — Remove nonnumeric characters
`false` (default) | `true`

Remove nonnumeric characters from a numeric variable, specified as a logical `true` or `false`.

Data Types: `logical`

Output Arguments

`opts` — Import options for file
`SpreadsheetImportOptions` | `DelimitedtextImportOptions` | `FixedWidthImportOptions`

Import options for the specified file, returned as a `SpreadsheetImportOptions` object or a `DelimitedTextImportOptions` object. The type of options object depends on the type of file specified. For text files (`.txt`, `.dat`, or `.csv`), the `detectImportOptions` function returns a `DelimitedTextImportOptions` or `FixedWidthImportOptions` object. For spreadsheet files (`.xls`, `.xlsb`, `.xlsm`, `.xlsx`, `.xltm`, `.xltx`, or `.ods`), the `detectImportOptions` function returns a `SpreadsheetImportOptions` object.

Tips

- **Updating Property Values After Creating the Import Options Object:** Use of dot notation to update the properties of the import options object created by `detecImportOptions` is not recommended. Setting properties using dot notation does not lead to a re-detection of all the import options for the file. Therefore, to update and re-detect all the properties, you must specify the new values by using name-value pairs. For example, update the value for the `ConsecutiveDelimitersRule` property and re-detect the import options as follows.

  ```
  opts = detectImportOptions(__,'ConsecutiveDelimitersRule','join')
  ```

See Also

`DelimitedTextImportOptions` | `FixedWidthImportOptions` | `SpreadsheetImportOptions` | `readtable`

Topics

"Define Import Options for Tables"

Introduced in R2016b

detrend

Remove polynomial trend

Syntax

```
y = detrend(x)
y = detrend(x,n)
y = detrend(x,n,bp)
y = detrend( ___ ,Name,Value)
```

Description

`y = detrend(x)` removes the best straight-fit line from the data in `x`.

- If `x` is a vector, then `detrend` subtracts the trend from the elements of `x`.
- If `x` is a matrix, then `detrend` operates on each column separately, subtracting each trend from the corresponding column.

`y = detrend(x,n)` removes the `n`th-degree polynomial trend. For example, when `n = 0`, `detrend` removes the mean value from `x`. When `n = 1`, `detrend` removes the linear trend, which is equivalent to the previous syntax. When `n = 2`, `detrend` removes the quadratic trend.

`y = detrend(x,n,bp)` removes a continuous, piecewise trend with segments defined by the break points `bp`.

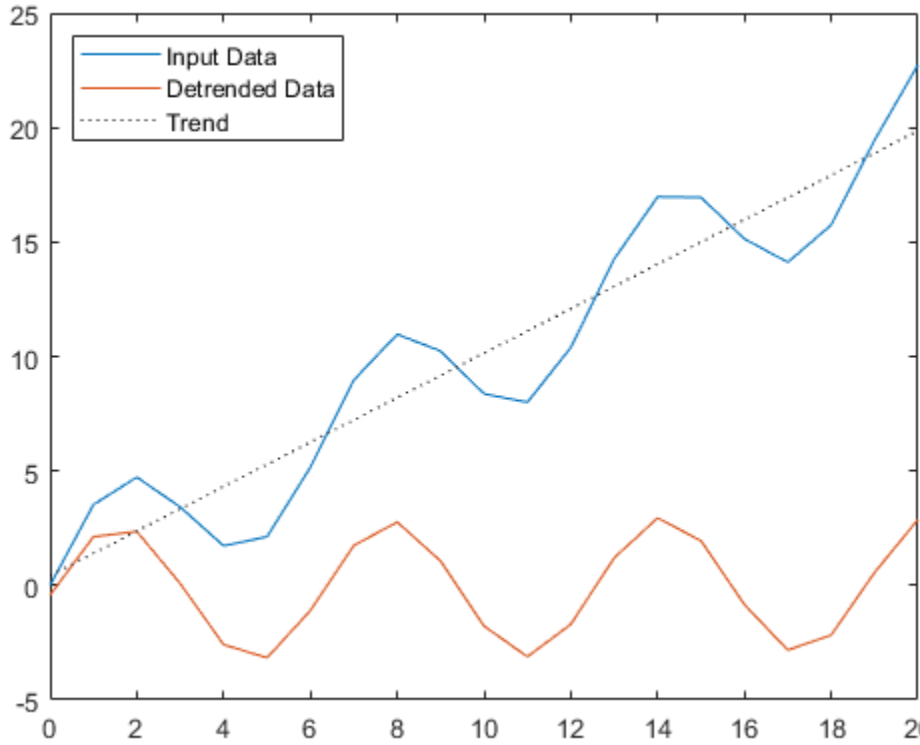
`y = detrend( ___ ,Name,Value)` specifies additional parameters for any of the previous syntaxes using one or more name-value pairs. For example, `detrend(x,1,bp,'Continuous','false')` specifies that the fitted trend can have discontinuities.

Examples

Continuous Linear Trend

Create a vector of data, and remove the continuous linear trend. Plot the original data, the detrended data, and the linear trend.

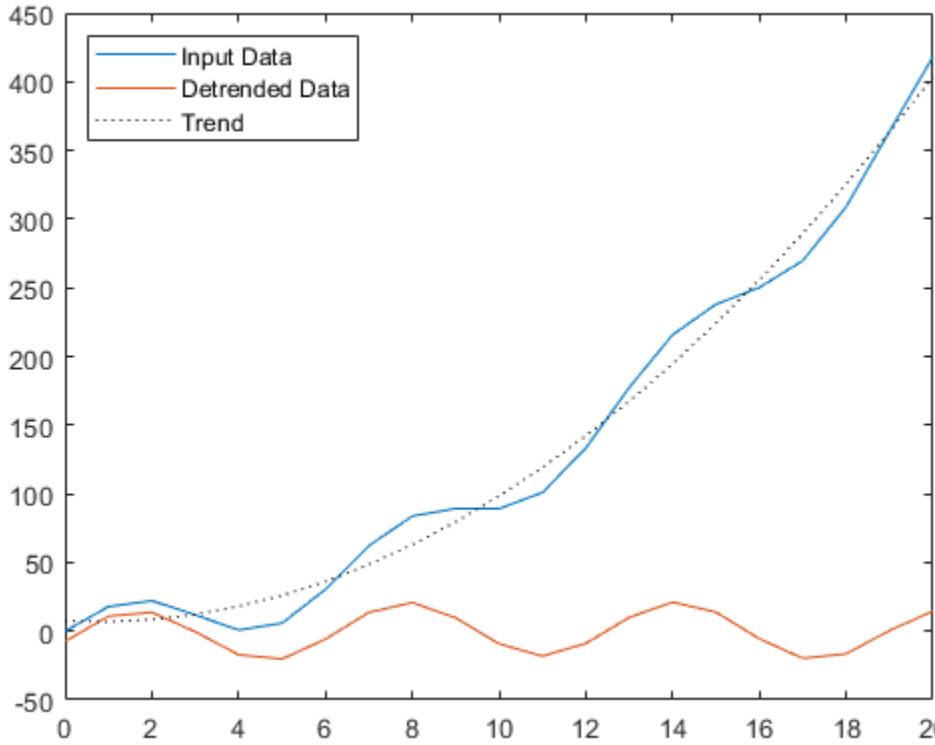
```
t = 0:20;
x = 3*sin(t) + t;
y = detrend(x);
plot(t,x,t,y,t,x-y,':k')
legend('Input Data','Detrended Data','Trend','Location','northwest')
```



Continuous Quadratic Trend

Create a vector of data, and remove the continuous quadratic trend. Plot the original data, the detrended data, and the trend.

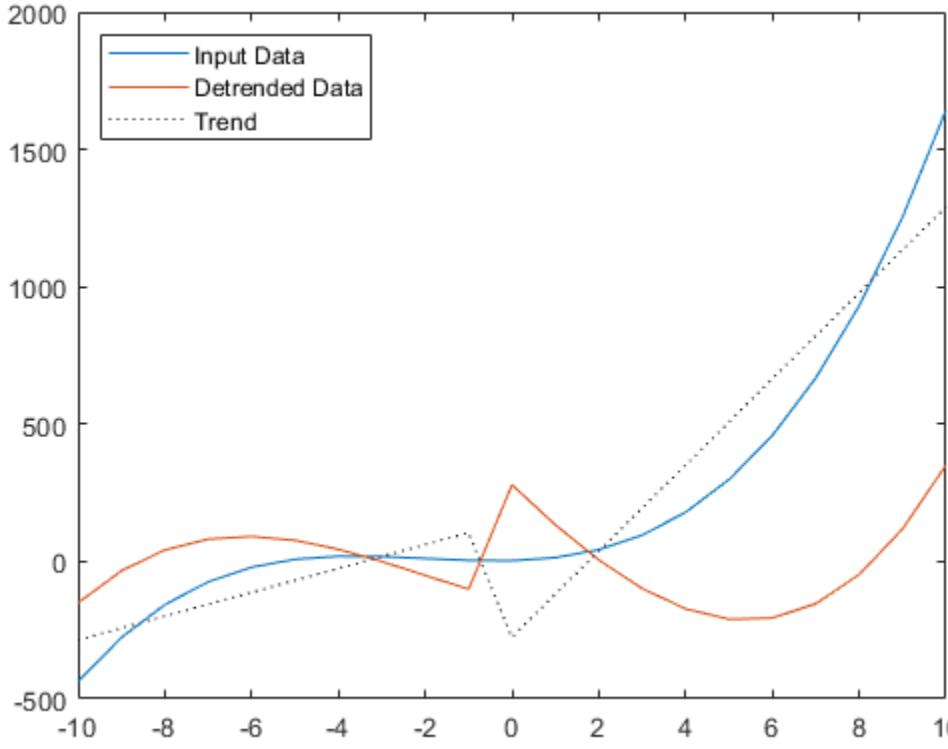
```
t = 0:20;
x = 20*sin(t) + t.^2;
y = detrend(x,2);
plot(t,x,t,y,t,x-y,':k')
legend('Input Data','Detrended Data','Trend','Location','northwest')
```



Discontinuous Linear Trend

Create a vector of data, and remove the piecewise linear trend using a break point at 0. Specify that the resulting output can be discontinuous. Plot the original data, the detrended data, and the trend.

```
t = -10:10;
x = t.^3 + 6*t.^2 + 4*t + 3;
bp = 0;
y = detrend(x,1,bp,'SamplePoints',t,'Continuous',false);
plot(t,x,t,y,t,x-y,':k')
legend('Input Data','Detrended Data','Trend','Location','northwest')
```



Input Arguments

x — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array. When `x` is a multidimensional array, `detrend` operates column-wise across all dimensions.

Data Types: `double` | `single`
Complex Number Support: Yes

n — Polynomial degree
non-negative integer scalar | `'constant'` | `'linear'`

Polynomial degree, specified as a non-negative integer scalar, or as `'constant'` (equivalent to `0`) or `'linear'` (equivalent to `1`).

bp — Break points
vector

Break points to define piecewise segments of the data, specified as a vector containing one of the following:

- Sample point values indicating the location of the break points. Sample point values are contained either in the default sample points vector `[1 2 3 ...]` or in the vector specified by the `'SamplePoints'` parameter.
- Logical values where logical 1 (`true`) indicates a break point in the corresponding element of the input data. If `bp` contains logical values, it must be the same length as the sample points.

Break points are useful when you want to compute separate trends for different segments of the data.

Data Types: `double` | `single` | `datetime` | `duration` | `logical`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `y = detrend(x,'SamplePoints',1:10:1000)`

`Continuous` — Continuity constraint
`true` (default) | `false`

Continuity constraint, specified as the comma-separated pair consisting of `'Continuous'` and one of the following:

- `true` — The fitted trend must be continuous everywhere.
- `false` — The fitted trend can contain discontinuities.

`SamplePoints` — Sample points
`[1 2 3 ...]` (default) | vector

Sample points, specified as the comma-separated pair consisting of `'SamplePoints'` and a vector. The sample points represent the locations of the input data on the *x*-axis, and they must be unique and sorted.

Data Types: `double` | `single` | `datetime` | `duration`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If the input argument `bp` is supplied but not logical, the argument must be sorted in ascending order.
- If the input argument `bp` is supplied but variable-sizing disabled, the argument must contain integers in the interval `[1,m-2]`. In this case, `m` is the number of elements in a column of the input argument `x` or the number of elements in `x` when `x` is a row vector (`m = length(x)`).
- If `'SamplePoints'` parameter values are supplied, variable-sizing is required.
- If polynomial degree `n` is neither a constant nor a logical, variable-sizing is required.

- Code generation uses a different method than MATLAB to detect nonunique or ill-conditioned problems to issue warnings. The code generation warnings do not always match the MATLAB warnings.
- The function `detrend` does not use `rand` in code generation.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

Remove Trends | `polyfit`

Topics

“Detrending Data”

Introduced before R2006a

Remove Trends

Remove polynomial trend from data in the Live Editor

Description

The **Remove Trends** task lets you interactively remove a polynomial trend in data. The task automatically generates MATLAB code for your live script.

Using this task, you can:

- Choose the degree of the polynomial trend to remove from data in a workspace variable.
- Dynamically arrange breakpoints to define piecewise segments of the data.
- Specify continuity constraints.
- Automatically visualize the computed trend and data with the trend removed.

Open the Task

To add the **Remove Trends** task to a live script in the MATLAB Editor:

- On the **Live Editor** tab, select **Task > Remove Trends**.
- In a code block in the script, type a relevant keyword, such as `remove` or `detrend`. Select **Remove Trends** from the suggested command completions.

See Also

Functions
`detrend`

Live Editor Tasks
Clean Missing Data | Clean Outlier Data | Find Change Points | Find Local Extrema | Smooth Data

Topics

"Add Interactive Tasks to a Live Script"
Clean Messy Data and Locate Extrema Using Live Editor Tasks
"Detrending Data"

Introduced in R2019b

deval

Evaluate differential equation solution structure

Syntax

```
y = deval(sol,x)
y = deval(x,sol)
y = deval( ___ ,idx)
[y,yp] = deval( ___ )
```

Description

`y = deval(sol,x)` and `y = deval(x,sol)` evaluate the solution `sol` of a differential equation problem at the points contained in `x`.

`y = deval( ___ ,idx)` returns only the solution components with indices listed in the vector `idx`. You can use either of the previously listed input argument combinations.

`[y,yp] = deval( ___ )` also returns `yp`, which is the first derivative of the numeric solution produced by the solver.

Examples

Evaluate DDE with Constant Delays

This example solves the DDE equation `y' = ddex1de(t,y,Z)` using `dde23`, then plots the solution.

Solve the system using `dde23`.

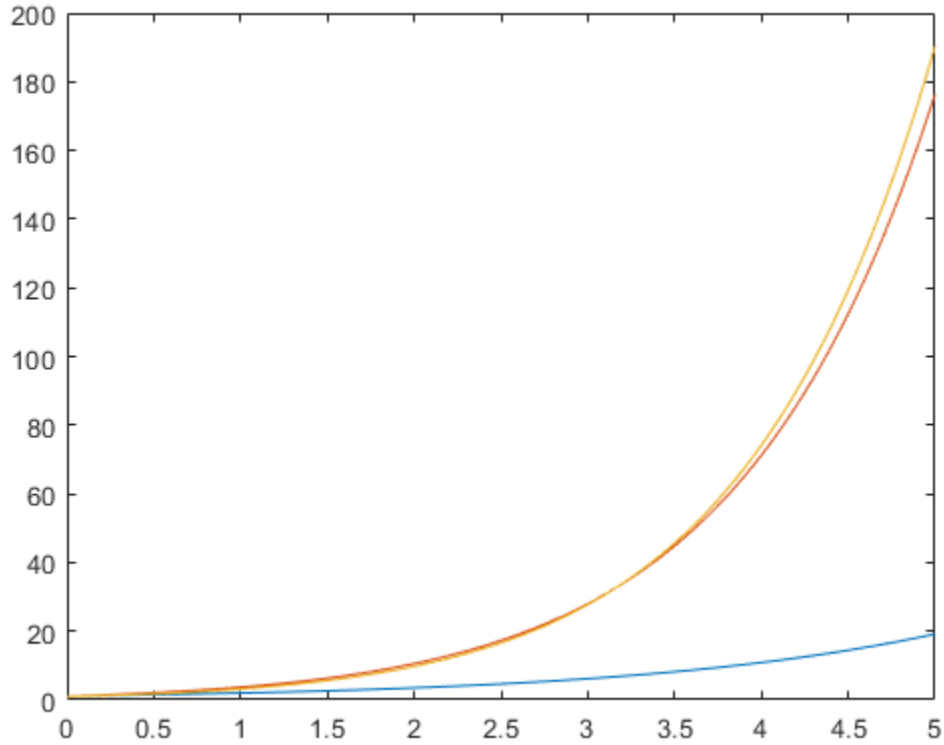
```
sol = dde23(@ddex1de, [1 0.2], @ddex1hist, [0 5]);
```

Evaluate the solution at 100 points in the interval `[0 5]`.

```
x = linspace(0,5);
y = deval(sol,x);
```

Plot the solution.

```
plot(x,y)
```



Evaluate van der Pol Equation

This example solves the system `y' = vdp1(t,y)` using `ode45`, then plots the first component of the solution.

Solve the system using `ode45`.

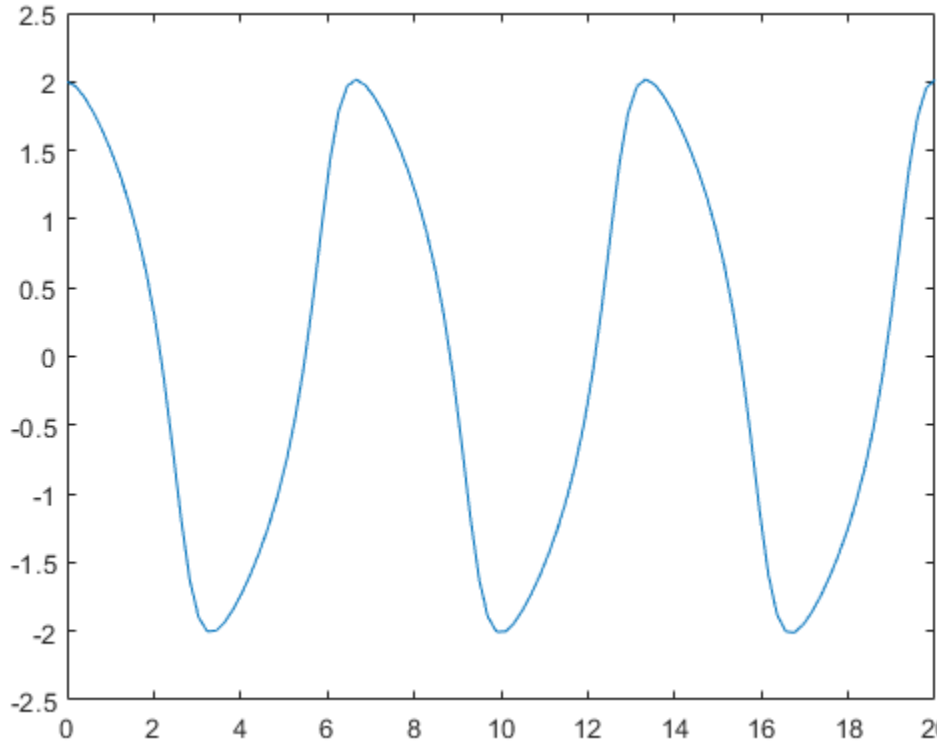
```
sol = ode45(@vdp1, [0 20], [2 0]);
```

Evaluate the first component of the solution at 100 points in the interval `[0 20]`.

```
x = linspace(0,20,100);
y = deval(sol,x,1);
```

Plot the solution.

```
plot(x,y)
```



Slope of Interpolating Function

Solve the simple ODE `y' = t^2` with initial condition `y0 = 0` in the interval $[0, 3]$ using `ode23`.

```
sol = ode23(@(t,y) t^2, [0 3], 0);
```

Evaluate the solution at seven points. The solution structure `sol` contains an interpolating function that `deval` uses to produce a continuous solution at these points. Specify a second output argument with `deval` to also return the derivative of the interpolating function at the specified points.

```
x = linspace(0,3,7);
[y,yp] = deval(sol,x)
```

```
y = 1×7

         0    0.0417    0.3333    1.1250    2.6667    5.2083    9.0000


yp = 1×7

         0    0.2500    1.0000    2.2500    4.0000    6.2500    9.0000
```

Input Arguments

`sol` — Solution structure
structure

Solution structure, specified as a structure returned by one of these differential equation solvers.

- Initial value problem solver — `ode45`, `ode23`, `ode113`, `ode15s`, `ode23s`, `ode23t`, `ode23tb`, `ode15i`
- Delay differential equations solver — `dde23`, `ddesd`, or `ddensd`
- Boundary value problem solver — `bvp4c` or `bvp5c`

Example: `sol = ode45(@myode,tspan,y0)`

Data Types: `struct`

x — Evaluation points
vector

Evaluation points, specified as a vector. `x` specifies the points at which you want the value of the solution. The elements of `x` must be contained in the original integration interval, `[sol.x(1) sol.x(end)]`. For each index `i`, the solution `y(:,i)` corresponds to `x(i)`.

Example: `0:0.1:1`

Example: `[2 3 5 8]`

Data Types: `single` | `double`

idx — Solution components to return
vector

Solution components to return, specified as a vector. Use this input when you are only interested in certain components of the solution.

Example: `y = deval(sol,x,[2 3])` returns only the second and third solution components.

Data Types: `single` | `double`

Output Arguments

y — Interpolated solution
vector | matrix

Interpolated solution, returned as a vector or matrix. The number of rows in `y` is equal to the number of solution components being returned.

For multipoint boundary value problems, the solution obtained by `bvp4c` or `bvp5c` might be discontinuous at the interfaces. For an interface point `xc`, the `deval` function returns the average of the limits from the left and right of `xc`. To get the limit values, set the value of `x` to be slightly larger or smaller than `xc`.

yp — Derivative of continuous solution produced by sol
vector | matrix

Derivative of continuous solution produced by `sol`, returned as a vector or matrix. `yp` is the same size as `y` and indicates the slope of the interpolating function used by `sol` at each solution point in `y`.

See Also

bvp4c | bvp5c | dde23 | ddensd | ddesd | ode113 | ode15i | ode15s | ode23 | ode23s | ode23t | ode23tb | ode45

Introduced before R2006a

diag

Create diagonal matrix or get diagonal elements of matrix

Syntax

```
D = diag(v)
D = diag(v,k)

x = diag(A)
x = diag(A,k)
```

Description

`D = diag(v)` returns a square diagonal matrix with the elements of vector `v` on the main diagonal.

`D = diag(v,k)` places the elements of vector `v` on the `k`th diagonal. `k=0` represents the main diagonal, `k>0` is above the main diagonal, and `k<0` is below the main diagonal.

`x = diag(A)` returns a column vector of the main diagonal elements of `A`.

`x = diag(A,k)` returns a column vector of the elements on the `k`th diagonal of `A`.

Examples

Create Diagonal Matrices

Create a 1-by-5 vector.

```
v = [2 1 -1 -2 -5];
```

Use `diag` to create a matrix with the elements of `v` on the main diagonal.

```
D = diag(v)
```

```
D = 5×5

     2     0     0     0     0
     0     1     0     0     0
     0     0    -1     0     0
     0     0     0    -2     0
     0     0     0     0    -5
```

Create a matrix with the elements of `v` on the first super diagonal (`k=1`).

```
D1 = diag(v,1)
```

```
D1 = 6×6

     0     2     0     0     0     0
     0     0     1     0     0     0
     0     0     0    -1     0     0
     0     0     0     0    -2     0
     0     0     0     0     0    -5
     0     0     0     0     0     0
```

The result is a 6-by-6 matrix. When you specify a vector of length `n` as an input, `diag` returns a square matrix of size `n+abs(k)`.

Get Diagonal Elements

Get the elements on the main diagonal of a random 6-by-6 matrix.

```
A = randi(10,6)
```

```
A = 6×6

     9     3    10     8     7     8
    10     6     5    10     8     1
     2    10     9     7     8     3
    10    10     2     1     4     1
     7     2     5     9     7     1
     1    10    10    10     2     9
```

```
x = diag(A)
```

```
x = 6×1

     9
     6
     9
     1
     7
     9
```

Get the elements on the first subdiagonal (`k=-1`) of `A`. The result has one fewer element than the main diagonal.

```
x1 = diag(A,-1)
```

```
x1 = 5×1

    10
    10
     2
     9
     2
```

Calling `diag` twice returns a diagonal matrix composed of the diagonal elements of the original matrix.

```
A1 = diag(diag(A))
```

```
A1 = 6×6

     9     0     0     0     0     0
     0     6     0     0     0     0
     0     0     9     0     0     0
     0     0     0     1     0     0
     0     0     0     0     7     0
     0     0     0     0     0     9
```

Input Arguments

v — Diagonal elements
vector

Diagonal elements, specified as a vector. If `v` is a vector with `N` elements, then `diag(v,k)` is a square matrix of order `N+abs(k)`.

`diag([])` returns an empty matrix, `[]`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

`A` — Input matrix
matrix

Input matrix. `diag` returns an error if `ndims(A) > 2`.

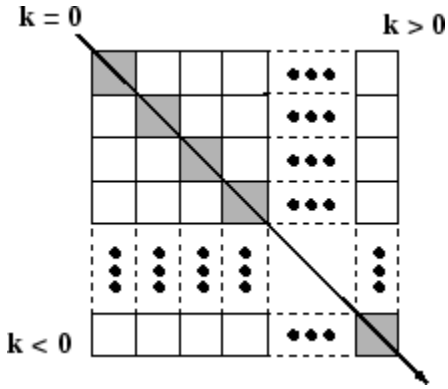
`diag([])` returns an empty matrix, `[]`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

`k` — Diagonal number
integer

Diagonal number, specified as an integer. `k=0` represents the main diagonal, `k>0` is above the main diagonal, and `k<0` is below the main diagonal.

For an m-by-n matrix, `k` is in the range $(-m+1) \le k \le (n-1)$.



Tips

- The `trace` of a matrix is equal to `sum(diag(A))`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If you supply `k`, then it must be a real and scalar integer value.
- For variable-size inputs that are variable-length vectors (1-by-: or :-by-1), `diag`:
  - Treats the input as a vector
  - Returns a matrix with the input vector along the specified diagonal
- For variable-size inputs that are not variable-length vectors, `diag`:
  - Treats the input as a matrix
  - Does not support inputs that are vectors at run time
  - Returns a variable-length vector

  If the input is variable-size (:m-by-:n) and has shape 0-by-0 at run time, then the output is 0-by-1, not 0-by-0. However, if the input is a constant size 0-by-0, then the output is `[]`.
- For variable-size inputs that are not variable-length vectors (1-by-: or :-by-1), `diag` treats the input as a matrix from which to extract a diagonal vector. This behavior occurs even if the input array is a vector at run time. To force `diag` to build a matrix from variable-size inputs that are not 1-by-: or :-by-1, use:
  - `diag(x(:))` instead of `diag(x)`
  - `diag(x(:),k)` instead of `diag(x,k)`
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- If you supply `k`, then it must be a real and scalar integer value.
- For variable-size inputs that are variable-length vectors (1-by-: or :-by-1), `diag`:
  - Treats the input as a vector
  - Returns a matrix with the input vector along the specified diagonal
- For variable-size inputs that are not variable-length vectors, `diag`:
  - Treats the input as a matrix
  - Does not support inputs that are vectors at run time
  - Returns a variable-length vector

  If the input is variable-size (:m-by-:n) and has shape 0-by-0 at run time, then the output is 0-by-1, not 0-by-0. However, if the input is a constant size 0-by-0, then the output is `[]`.
- For variable-size inputs that are not variable-length vectors (1-by-: or :-by-1), `diag` treats the input as a matrix from which to extract a diagonal vector. This behavior occurs even if the input array is a vector at run time. To force `diag` to build a matrix from variable-size inputs that are not 1-by-: or :-by-1, use:
  - `diag(x(:))` instead of `diag(x)`
  - `diag(x(:),k)` instead of `diag(x,k)`

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`blkdiag` | `isdiag` | `istril` | `istriu` | `spdiags` | `tril` | `triu`

Introduced before R2006a

matlab.unittest.diagnostics Package

Summary of classes in MATLAB Diagnostics Interface

Description

Use diagnostics to communicate relevant information in the event of a failure. To add a diagnostic message to a test case, use the `diagnostic` argument in any of the `matlab.unittest.qualifications` methods. The framework also displays diagnostic messages related to the nature of the qualification failure. The `matlab.unittest.diagnostics` package consists of the following classes.

Classes

matlab.unittest.diagnostics.ConstraintDiagnostic	Diagnostic with fields common to most constraints
matlab.unittest.diagnostics.Diagnostic	Fundamental interface class for matlab.unittest diagnostics
matlab.unittest.diagnostics.DiagnosticResult	Result of a diagnosed Diagnostic instance
matlab.unittest.diagnostics.DisplayDiagnostic	Diagnostic using a value's displayed output
matlab.unittest.diagnostics.FigureDiagnostic	Diagnostic to save specified figure
matlab.unittest.diagnostics.FileArtifact	Artifact associated with a file on disk
matlab.unittest.diagnostics.FrameworkDiagnostic	Diagnostic provided by testing framework
matlab.unittest.diagnostics.FunctionHandleDiagnostic	Diagnostic using a function's displayed output
matlab.unittest.diagnostics.LoggedDiagnosticEventData	Event data for DiagnosticLogged event listeners
matlab.unittest.diagnostics.ScreenshotDiagnostic	Diagnostic to capture screen as image file
matlab.unittest.diagnostics.StringDiagnostic	Diagnostic using string

See Also

Introduced in R2013a

matlab.unittest.diagnostics.ConstraintDiagnostic class

Package: `matlab.unittest.diagnostics`
Superclasses: `matlab.unittest.diagnostics.Diagnostic`

Diagnostic with fields common to most constraints

Description

The `ConstraintDiagnostic` class provides various textual fields that are common to most constraints. These fields may be turned on or off depending on their applicability.

The `ConstraintDiagnostic` class is a helper class for displaying diagnostics when using constraints. The `ConstraintDiagnostic` class provides custom constraint authors a way to add a common look and feel to diagnostics produced by the `getDiagnosticFor` method of constraints.

Constraint diagnostics are displayed in the following order: Description, Conditions, Actual Value, and Expected Value.

Properties

`ActVal`

The actual value passed to the constraint for testing.

`ActValHeader`

Header information for the actual value property, `ActVal`, specified as a character vector. The default header is `'Actual Value:'`.

`Conditions`

Formatted list of conditions, specified as a single character vector. Each condition starts on a new line and begins with an arrow (`-->`) delimiter. Conditions are added to the list using the `addCondition` and `addConditionsFrom` methods.

`ConditionsCount`

Number of conditions in the condition list. This is a read-only property generated from the conditions list. The conditions list is defined in the `Conditions` property.

`Description`

General diagnostic information, specified as a character vector.

`DisplayActVal`

Indicator whether to display the actual value property, `ActVal`, specified as a boolean. By default, the actual value is not displayed and the value of this property is `false`.

`DisplayConditions`

Indicator of whether to display the `Conditions` property, specified as a boolean. By default, the conditions are not displayed and the value of this property is `false`. Even if `DisplayConditions` is set to `true`, if there are no conditions on the conditions list, neither the conditions header or the conditions list are displayed.

`DisplayDescription`

Indicator of whether to display the `Description` property, specified as a boolean. By default, the description is not displayed and the value of this property is `false`.

`DisplayExpVal`

Indicator whether to display the expected value property, `ExpVal`, specified as a boolean. By default, the expected value is not displayed and the value of this property is `false`.

`ExpVal`

If applicable, the expected value. This property can be turned off if the associated constraint does not contain an expected value.

`ExpValHeader`

Header information for the expected value property, `ExpVal`, specified as a character vector. The default header is `'Expected Value:'`.

Inherited Properties

`Artifacts`

The artifacts produced during the last diagnostic evaluation, returned as an array of `FileArtifact` instances.

`DiagnosticText`

The `DiagnosticText` property provides the means by which the actual diagnostic information is communicated to consumers of diagnostics, such as testing frameworks. The property is a character vector that is defined during evaluation of the `diagnose` method.

Methods

addCondition	Add condition to condition list
addConditionsFrom	Add condition from another `ConstraintDiagnostic` to condition list
getDisplayableString	Convert object to string for display
getPreDescriptionString	Returns text to be displayed prior to description
getPostDescriptionString	Returns text to be displayed following description
getPostConditionString	Returns text to be displayed following conditions list
getPostActValString	Returns text to be displayed following actual value
getPostExpValString	Returns text to be displayed following expected value

Inherited Methods

diagnose	Execute diagnostic action
join	Join multiple diagnostics into a single array

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

See Also

`Diagnostic` | `matlab.unittest.constraints` | `matlab.unittest.diagnostics`

addCondition

Class: `matlab.unittest.diagnostics.ConstraintDiagnostic`
Package: `matlab.unittest.diagnostics`

Add condition to condition list

Syntax

```
addCondition(diag, cond)
```

Description

`addCondition(diag, cond)` adds the condition, `cond`, to the condition list. Add conditions to the condition list one at a time. When the condition list is displayed, each condition is preceded by an arrow (`-->`) delimiter and indented.

Input Arguments

cond

Condition, specified as a character vector or string scalar containing information specific to the cause of the constraint failure or another `Diagnostic` instance, which acts as a "subdiagnostic". To set multiple conditions, specify `cond` as a string array or array of `Diagnostic` instances.

diag

`matlab.unittest.diagnostics.Diagnostic` instance.

See Also

`addConditionsFrom`

addConditionsFrom

Class: `matlab.unittest.diagnostics.ConstraintDiagnostic`
Package: `matlab.unittest.diagnostics`

Add condition from another `ConstraintDiagnostic` to condition list

Syntax

```
addConditionsFrom(constDiag, otherConstDiag)
```

Description

`addConditionsFrom(constDiag, otherConstDiag)` adds the conditions from the `ConstraintDiagnostic` instance, `constDiag`, to the condition list in the `Diagnostic` instance, `diag`. This is useful when a constraints composes another constraint, and needs to use the conditions produced in the diagnostics of the composed constraint.

Input Arguments

`constDiag`

Diagnostic to add conditions to, specified as a `matlab.unittest.diagnostics.ConstraintDiagnostic` instance

`otherConstDiag`

Diagnostic to add conditions from, specified as a `matlab.unittest.diagnostics.ConstraintDiagnostic` instance

Examples

Add Conditions from a Constraint

```
% This demonstrates a constraint that composes another constraint
% and uses the addConditionsFrom method to utilize the conditions
% from the composed ConstraintDiagnostic.
classdef IsDouble < matlab.unittest.constraints.Constraint

    properties(Constant, GetAccess=private)
        DoubConst = matlab.unittest.constraints.IsInstanceOf(?double);
    end

    methods
        function tf = satisfiedBy(constraint, actual)
            tf = constraint.DoubConst.satisfiedBy(actual);
        end
        function diag = getDiagnosticFor(constraint, actual)
            diag = ConstraintDiagnostic;

            % Now add conditions from the IsInstanceOf
            % Diagnostic
            otherDiag = constraint.DoubConst.getDiagnosticFor(actual);
            diag.addConditionsFrom(otherDiag)

            % ...
        end
    end
end
```

See Also

addCondition

matlab.unittest.diagnostics.ConstraintDiagnostic.getDisplayableString

Class: `matlab.unittest.diagnostics.ConstraintDiagnostic`
Package: `matlab.unittest.diagnostics`

Convert object to string for display

Syntax

```
str =
matlab.unittest.diagnostics.ConstraintDiagnostic.getDisplayableStrin
g(value)
```

Description

`str = matlab.unittest.diagnostics.ConstraintDiagnostic.getDisplayableString(value)` converts the object, `obj` to a character vector, `str` for display in a diagnostic result. This conversion determines if hotlinks should be included in the character vector and truncates large numeric or cell arrays.

Input Arguments

`value`

Object of arbitrary class

See Also

getPreDescriptionString

Class: `matlab.unittest.diagnostics.ConstraintDiagnostic`
Package: `matlab.unittest.diagnostics`

Returns text to be displayed prior to description

Syntax

```
str = getPreDescriptionString(constDiag)
```

Description

`str = getPreDescriptionString(constDiag)` returns text to be displayed prior to the description. This method can be overridden to inject character vectors prior to displaying the `Description` property of the `ConstraintDiagnostic`. The location of this text is tied to the `Description` property. Its placement relative to other fields is not guaranteed.

Input Arguments

`constDiag`

`matlab.unittest.diagnostics.ConstraintDiagnostic` instance.

See Also

`ConstraintDiagnostic` | `getPostActValString` | `getPostConditionString` | `getPostDescriptionString` | `getPostExpValString`

getPostDescriptionString

Class: `matlab.unittest.diagnostics.ConstraintDiagnostic`
Package: `matlab.unittest.diagnostics`

Returns text to be displayed following description

Syntax

```
str = getPostDescriptionString(constDiag)
```

Description

`str = getPostDescriptionString(constDiag)` returns text to be displayed following the description. This method can be overridden to inject character vectors subsequent to displaying the `Description` property of the `ConstraintDiagnostic`. The location of this text is tied to the `Description` property. Its placement relative to other fields is not guaranteed.

Input Arguments

`constDiag`

`matlab.unittest.diagnostics.ConstraintDiagnostic` instance.

See Also

`ConstraintDiagnostic` | `getPostActValString` | `getPostConditionString` | `getPostExpValString` | `getPreDescriptionString`

getPostConditionString

Class: `matlab.unittest.diagnostics.ConstraintDiagnostic`
Package: `matlab.unittest.diagnostics`

Returns text to be displayed following conditions list

Syntax

```
str = getPostConditionsString(constDiag)
```

Description

`str = getPostConditionsString(constDiag)` returns text to be displayed following the conditions list. This method can be overridden to inject character vectors subsequent to displaying the `Conditions` property of the `ConstraintDiagnostic`. The location of this text is tied to the `Conditions` property. Its placement relative to other fields is not guaranteed.

Input Arguments

`constDiag`

`matlab.unittest.diagnostics.ConstraintDiagnostic` instance.

See Also

`ConstraintDiagnostic` | `getPostActValString` | `getPostDescriptionString` | `getPostExpValString` | `getPreDescriptionString`

getPostActValString

Class: `matlab.unittest.diagnostics.ConstraintDiagnostic`
Package: `matlab.unittest.diagnostics`

Returns text to be displayed following actual value

Syntax

```
str = getPostActualValString(constDiag)
```

Description

`str = getPostActualValString(constDiag)` returns text to be displayed following the actual value. This method can be overridden to inject character vectors subsequent to displaying the `ActVal` property of the `ConstraintDiagnostic`. The location of this text is tied to the `ActVal` property. Its placement relative to other fields is not guaranteed.

Input Arguments

`constDiag`

`matlab.unittest.diagnostics.ConstraintDiagnostic` instance.

See Also

`ConstraintDiagnostic` | `getPostConditionString` | `getPostDescriptionString` | `getPostExpValString` | `getPreDescriptionString`

getPostExpValString

Class: `matlab.unittest.diagnostics.ConstraintDiagnostic`
Package: `matlab.unittest.diagnostics`

Returns text to be displayed following expected value

Syntax

```
str = getPostExpValString(constDiag)
```

Description

`str = getPostExpValString(constDiag)` returns text to be displayed following the expected value. This method can be overridden to inject character vectors subsequent to displaying the `ExpVal` property of the `ConstraintDiagnostic`. The location of this text is tied to the `ExpVal` property. Its placement relative to other fields is not guaranteed.

Input Arguments

`constDiag`

`matlab.unittest.diagnostics.ConstraintDiagnostic` instance.

See Also

`ConstraintDiagnostic` | `getPostActValString` | `getPostConditionString` | `getPostDescriptionString` | `getPreDescriptionString`

matlab.unittest.diagnostics.Diagnostic class

Package: `matlab.unittest.diagnostics`

Fundamental interface class for `matlab.unittest` diagnostics

Description

The `Diagnostic` interface class is the means by which the `matlab.unittest` framework and its clients package diagnostic information. All diagnostics are derived from `Diagnostic`, whether they are user-supplied test diagnostics for an individual comparison or diagnostics associated with the `Constraint` used in the comparison.

Classes which derive from `Diagnostic` encode the diagnostic actions to be performed. They produce a diagnostic result that is displayed appropriately by the test running framework. In exchange for meeting this requirement, any `Diagnostic` implementation can be used directly with `matlab.unittest` qualifications. These qualifications execute the diagnostic action and store the result for the test running framework to use.

As a convenience, the framework creates appropriate diagnostic instances for arrays of character vectors, strings, and function handles when they are user supplied test diagnostics. To retain good performance, these values are only converted into `Diagnostic` instances when a qualification failure occurs or when the test running framework is explicitly observing passing qualifications. The default test runner does not explicitly observe passing qualifications.

Properties

`Artifacts`

The artifacts produced during the last diagnostic evaluation, returned as an array of artifacts.

`DiagnosticText`

The `DiagnosticText` property provides the means by which the actual diagnostic information is communicated to consumers of diagnostics, such as testing frameworks.

The property is a character vector that is defined during evaluation of the `diagnose` method.

Methods

diagnose	Execute diagnostic action
join	Join multiple diagnostics into a single array

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Use Diagnostic Result

```
import matlab.unittest.constraints.IsEqualTo

% Create a TestCase for interactive use
testCase = matlab.unittest.TestCase.forInteractiveUse;

% Create StringDiagnostic upon failure
testCase.verifyThat(1, IsEqualTo(2), 'User supplied Diagnostic')

% Create FunctionHandleDiagnostic upon failure
testCase.verifyThat(1, IsEqualTo(2), @() system('ps'))

% Usage of user defined Diagnostic upon failure (see definition below)
testCase.verifyThat(1, IsEqualTo(2), ProcessStatusDiagnostic...
    ('Could not close my third party application!'))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Diagnostic definition
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
classdef ProcessStatusDiagnostic < matlab.unittest.diagnostics.Diagnostic
    % ProcessStatusDiagnostic - an example diagnostic
    %
```

```
    %   Simple example to demonstrate how to create a custom
    %   diagnostic.

    properties

        % HeaderText - user-supplied header to display
        HeaderText = '(No header supplied)';
    end

    methods
        function diag = ProcessStatusDiagnostic(header)
            % Constructor - construct a ProcessStatusDiagnostic
            %
            %   The ProcessStatusDiagnostic constructor takes an
            %   optional header to be displayed along with process
            %   information.
            if (nargin >0)
                diag.HeaderText = header;
            end
        end

        function diagnose(diag)

            [status, processInfo] = system('ps');
            if (status ~= 0)
                processInfo = sprintf(...
                    ['!!! Could not obtain status diagnostic information!!!'...
                    ' [exit status code: %d]\n%s'], status, processInfo);
            end
            diag.DiagnosticText = sprintf('%s\n%s', diag.HeaderText,...
                processInfo);
        end
    end

end % classdef
```

See Also

FunctionHandleDiagnostic | StringDiagnostic | matlab.unittest.constraints.Constraint | matlab.unittest.diagnostics | matlab.unittest.plugins.DiagnosticsValidationPlugin

Topics

"Table of Verifications, Assertions, and Other Qualifications"

diagnose

Class: `matlab.unittest.diagnostics.Diagnostic`
Package: `matlab.unittest.diagnostics`

Execute diagnostic action

Syntax

```
diagnose(diag)
```

Description

`diagnose(diag)` executes diagnostic action for the `matlab.unittest.diagnostics.Diagnostic` instance, `diag`. The `diagnose` method is the means by which individual `Diagnostic` implementations can perform their respective diagnostic evaluations. Each concrete implementation is responsible for populating the `DiagnosticText` property of the `Diagnostic` object. Typically, text printed to the Command Window during diagnostic evaluation is not considered part of the diagnostic result and is ignored by the testing framework.

See Also

`Diagnostic`

join

Class: `matlab.unittest.diagnostics.Diagnostic`
Package: `matlab.unittest.diagnostics`

Join multiple diagnostics into a single array

Syntax

```
diagArray = join(diag1,...,diagN)
```

Description

`diagArray = join(diag1,...,diagN)` joins multiple diagnostics, specified by `diag1` through `diagN`, into a single array, `diagArray`.

Input Arguments

diag

Diagnostic content, specified as an instance of a `Diagnostic` object, a string array, a character array, a function handle, or an arbitrary type.

Output Arguments

diagArray

Array of joined diagnostic content.

- If `diagN` is an object that derives from `Diagnostic`, it is included in the array unmodified.
- If `diagN` is a `char` or a `string`, it is formed into a `StringDiagnostic` and included in the array.

- If `diagN` is a `function_handle`, it is formed into a `FunctionHandleDiagnostic` and included in the array.
- If `diagN` is any other type, it is formed into a `DisplayDiagnostic` and included in the array.

Examples

Join Diagnostic Content

```
% The following example creates a diagnostic array of length 4,
% demonstrating standard Diagnostic conversions. Note:
% MyCustomDiagnostic is for example purposes and is not executable
% code.

import matlab.unittest.diagnostics.Diagnostic
import matlab.unittest.constraints.IsTrue

arbitraryValue = 5;
testCase.verifyThat(false, IsTrue, ...
    Diagnostic.join(...
        'should have been true', ...
        @() system('ps'), ...
        arbitraryValue, ...
        MyCustomDiagnostic))
```

Alternatives

You can use array concatenation join diagnostics into an array if at least one of the values is a diagnostic. The `join` method prevents the need to have any `Diagnostics` in the array. Considering the following example.

```
arbitraryValue = 5;
testCase.verifyThat(false, IsTrue, ...
    ['should have been true', ...
    @() system('ps'), ...
    arbitraryValue, ...
    MyCustomDiagnostic]);
```

Since `MyCustomDiagnostic` is a `Diagnostic`, the other values are correctly converted to diagnostics as well.

See Also

`matlab.mixin.Heterogeneous`

matlab.unittest.diagnostics.DiagnosticResult class

Package: `matlab.unittest.diagnostics`

Result of a diagnosed `Diagnostic` instance

Description

A `DiagnosticResult` object captures the result of a diagnosed `Diagnostic` instance. It is safer to work with `DiagnosticResult` instances than to work directly with the `Diagnostic` instances.

Only the test framework constructs this class directly. The test framework assigns property values to the `DiagnosticResult` instance from the property values of objects of the `QualificationEventData`, `LoggedDiagnosticEventData`, `QualificationDiagnosticRecord`, and `LoggedDiagnosticRecord` classes.

Properties

`Artifacts` — Artifacts produced during last diagnostic evaluation
array of artifacts

The artifacts produced during the last diagnostic evaluation of a `Diagnostic`, returned as an array of artifacts. After the test framework diagnoses the `Diagnostic` object, it copies the value of the `Artifacts` property to the `DiagnosticResult` instance.

`DiagnosticText` — Result of diagnostic evaluation
character vector

Result of a diagnostic evaluation of a `Diagnostic` object, returned as a character vector. After the test framework diagnoses the `Diagnostic` object, it copies the value of the `DiagnosticText` property to the `DiagnosticResult` instance.

Data Types: `char`

See Also

`matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.diagnostics.LoggedDiagnosticEventData` | `matlab.unittest.plugins.diagnosticrecord.LoggedDiagnosticRecord` | `matlab.unittest.plugins.diagnosticrecord.QualificationDiagnosticRecord` | `matlab.unittest.qualifications.QualificationEventData`

Introduced in R2017a

matlab.unittest.diagnostics.DisplayDiagnostic class

Package: `matlab.unittest.diagnostics`
Superclasses: `matlab.unittest.diagnostics.Diagnostic`

Diagnostic using a value's displayed output

Description

The `DisplayDiagnostic` class provides a diagnostic result that uses a value’s displayed output. This output is the same text displayed using the `display` function. When the diagnostic information is accessible through a variable in the current workspace, the `DisplayDiagnostic` class is a means to provide quick diagnostic information.

Construction

`DisplayDiagnostic(diagValue)` creates a new `DisplayDiagnostic` instance.

Input Arguments

`diagValue`

The value that the `Diagnostic` uses to generate diagnostic information.

The resulting diagnostic information is equivalent to displaying this value at the MATLAB command prompt. The result is packaged for consumption by the testing framework, which may or may not display the information at the command prompt.

Properties

`Value`

The value that the `Diagnostic` uses to generate diagnostic information, specified in the `diagValue` input argument. This property is read-only.

Inherited Properties

DiagnosticText

The `DiagnosticText` property provides the means by which the actual diagnostic information is communicated to consumers of diagnostics, such as testing frameworks. The property is a character vector that is defined during evaluation of the `diagnose` method.

Methods

Inherited Methods

diagnose	Execute diagnostic action
join	Join multiple diagnostics into a single array

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Create DisplayDiagnostic Object

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.diagnostics.DisplayDiagnostic

testCase = TestCase.forInteractiveUse;
```

Use a `DisplayDiagnostic` to display diagnostic information upon test failure.

```
testCase.verifyThat(1, IsEqualTo(2), DisplayDiagnostic(inputParser))
```

```
Verification failed.

----------------
Test Diagnostic:
----------------
      inputParser with properties:

           FunctionName: ''
          CaseSensitive: 0
          KeepUnmatched: 0
        PartialMatching: 1
           StructExpand: 1
             Parameters: {1x0 cell}
                Results: [1x1 struct]
               Unmatched: [1x1 struct]
          UsingDefaults: {1x0 cell}

---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> NumericComparator failed.
    --> The numeric values are not equal using "isequaln".
    --> Failure table:
                Actual    Expected    Error    RelativeError
                ______    ________    _____    _____________

                1         2           -1       -0.5

Actual double:
         1
Expected double:
         2
```

In the test diagnostic section of the output, the output from `inputParser` object is the same as MATLAB displays at the command prompt.

See Also

`FunctionHandleDiagnostic` | `StringDiagnostic` | `matlab.unittest.diagnostics`

Introduced in R2013a

matlab.unittest.diagnostics.FigureDiagnostic class

Package: `matlab.unittest.diagnostics`

Diagnostic to save specified figure

Description

Use the `FigureDiagnostic` class to create a diagnostic that saves a figure to a file. The file persists after MATLAB completes the test run, and so it is available for post-test inspection.

Construction

`FigureDiagnostic(fig)` creates a diagnostic to save a specified figure. When the testing framework diagnoses the `FigureDiagnostic` instance, it saves `fig` to a FIG file and to a PNG file. Each file has a unique name consisting of a prefix (`'Figure_'`, by default), an automatically generated identifier, and the file extension. An example file name is `Figure_cf95fe7f-5a7c-4310-9c19-16c0c17a969f.png`. To view the location of the file, access the `FileArtifact` object through the `TestResult` instance.

`FigureDiagnostic(fig,Name,Value)` creates a diagnostic with additional options specified by one or more `Name,Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. For example, `FigureDiagnostic(fig,'Prefix','LoggedFigure_','Formats','png')` saves `fig` as a PNG file only, and uses the prefix `'LoggedFigure_'` instead of `'Figure_'`.

Input Arguments

`fig` — Figure to save
`Figure` object

Figure to save when the testing framework diagnoses the `FigureDiagnostic` instance, specified as a `Figure` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `FigureDiagnostic(testFig,'Formats','fig')`

`Formats` — File format of figure
`["fig" "png"]` (default) | `"fig"` | `"png"`

File format of figure, specified as `["fig" "png"]`, `"fig"`, or `"png"`. You can specify these values as character vectors, as `{'fig','png'}`, `'fig'`, or `'png'`. Within the object, MATLAB stores them as strings.

`Prefix` — File name prefix
`'Figure_'` (default) | text

File name prefix, specified as text. If you do not specify a prefix, the default prefix is `'Figure_'`. Specify the value as a character vector or string scalar. Within the object, MATLAB stores it as character vectors.

Example: `'LoggedFigure_'`

Example: `"TestFig-"`

Properties

`Figure` — Figure to save
`Figure` object

Figure to save when the testing framework diagnoses the `FigureDiagnostic` instance, returned as a `Figure` object. The `Figure` property is read-only, and its value is set during construction.

`Formats` — File format for the saved figure
`["fig" "png"]` (default) | `"fig"` | `"png"`

File format for the saved figure, returned as `["fig" "png"]`, `"fig"`, or `"png"`. The `Formats` property is read-only, and its value is set during construction.

`Prefix` — File name prefix
`'Figure_'` (default) | character vector

File name prefix, returned as a character vector. The default prefix is `'Figure_'`. The `Prefix` property is read-only, and its value is set during construction.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Create `FigureDiagnostic` Object

Create a `TestCase` for interactive use.

```
import matlab.unittest.TestCase
testCase = TestCase.forInteractiveUse;
```

Create a figure.

```
fig = figure;
ax = axes(fig);
surf(ax,peaks)
```

Use a `FigureDiagnostic` to save the figure as a test diagnostic. Verify that the figure has no children. This qualification fails and MATLAB displays the test diagnostic.

```
import matlab.unittest.diagnostics.FigureDiagnostic
testCase.verifyEmpty(fig.Children,FigureDiagnostic(fig))
```

```
Interactive verification failed.

----------------
Test Diagnostic:
----------------
Figure saved to:
--> C:\work\Temp\Figure_0b3da19f-5248-442b-aebf-3fb6d707fd1b.fig
--> C:\work\Temp\Figure_0b3da19f-5248-442b-aebf-3fb6d707fd1b.png
```

```
---------------------
Framework Diagnostic:
---------------------
verifyEmpty failed.
--> The value must be empty.
--> The value has a size of [1  1].

Actual matlab.graphics.axis.Axes:
      Axes with properties:

                 XLim: [0 50]
                 YLim: [0 60]
               XScale: 'linear'
               YScale: 'linear'
        GridLineStyle: '-'
             Position: [0.130000000000000 0.110000000000000 0.775000000000000 0.815000
                Units: 'normalized'

      Use get to show all properties
```

Specify Format and Prefix for `FigureDiagnostic`

Create a `TestCase` for interactive use.

```
import matlab.unittest.TestCase
testCase = TestCase.forInteractiveUse;
```

Create a figure.

```
fig = figure;
membrane(6)
```

Use a `FigureDiagnostic` to log the figure as a test diagnostic. Save the file only as a PNG and use a custom prefix for the file name.

```
import matlab.unittest.diagnostics.FigureDiagnostic
testCase.log(FigureDiagnostic(fig,'Formats',{'png'},'Prefix','LoggedFigure_'))
```

```
Interactive diagnostic logged.
Figure saved to:
--> C:\work\Temp\LoggedFigure_0a02faa1-3e14-4783-9954-b56caa6b326d.png
```

Create Test Report That Includes Artifacts

In a file in your current working folder, create the `FigurePropTest` test class. If the `failingTest` test method fails (it always does in this example), it uses a `FigureDiagnostic` to save the figure so you can examine it later.

```
classdef FigurePropTest < matlab.unittest.TestCase
    properties
        TestFigure
    end
    methods(TestMethodSetup)
        function createFigure(testCase)
            testCase.TestFigure = figure;
        end
    end
    methods(TestMethodTeardown)
        function closeFigure(testCase)
            close(testCase.TestFigure)
        end
    end
    methods(Test)
        function defaultCurrentPoint(testCase)
            cp = testCase.TestFigure.CurrentPoint;
            testCase.verifyEqual(cp,[0 0], ...
                'Default current point is incorrect')
        end
        function defaultCurrentObject(testCase)
            import matlab.unittest.constraints.IsEmpty
            co = testCase.TestFigure.CurrentObject;
            testCase.verifyThat(co,IsEmpty, ...
                'Default current object should be empty')
        end
        function failingTest(testCase)
            import matlab.unittest.diagnostics.FigureDiagnostic
            fig = testCase.TestFigure;
            ax = axes(fig);
            surf(ax,peaks)
            testCase.verifyEmpty(testCase.TestFigure.Children, ...
                FigureDiagnostic(testCase.TestFigure))
        end
    end
end
```

At the command prompt, create a test suite.

```
suite = testsuite('FigurePropTest');
```

Create a silent test runner that records diagnostics and generates a PDF report.

```
import matlab.unittest.plugins.DiagnosticsRecordingPlugin
import matlab.unittest.plugins.TestReportPlugin
runner = matlab.unittest.TestRunner.withNoPlugins;
runner.addPlugin(DiagnosticsRecordingPlugin);
runner.addPlugin(TestReportPlugin.producingPDF('MyTestReport.pdf'));
```

Change the default artifact root to your current working folder.

```
runner.ArtifactsRootFolder = pwd;
```

Run the tests. The third test fails.

```
results = runner.run(suite)
```

```
Generating report. Please wait.
    Preparing content for the report.
    Adding content to the report.
    Writing report to file.
Report has been saved to: C:\work\MyTestReport.pdf

results =

  1×3 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
   2 Passed, 1 Failed, 0 Incomplete.
   1.2295 seconds testing time.
```

Display the test diagnostic results for the third test. The test framework saved two artifacts related to the third test. By default, a `FigureDiagnostic` object saves a figure as both a PNG file and a FIG file.

```
results(3).Details.DiagnosticRecord.TestDiagnosticResults
```

```
ans =

  DiagnosticResult with properties:

        Artifacts: [1×2 matlab.unittest.diagnostics.FileArtifact]
    DiagnosticText: 'Figure saved to:↵--> C:\work\715b5416-5c52-4a53-bbec-837a5db57392\
```

Display the stored location of the first artifact.

```
results(3).Details.DiagnosticRecord.TestDiagnosticResults.Artifacts(1)
```

```
ans =

  FileArtifact with properties:

       Name: "Figure_284d9501-2121-45a1-bf5c-18904ce11e8f.fig"
    Location: "C:\work\715b5416-5c52-4a53-bbec-837a5db57392"
    FullPath: "C:\work\715b5416-5c52-4a53-bbec-837a5db57392\Figure_284d9501-2121-45a1-b
```

To inspect the image related to the failed test, open the file at the location shown in the `FullPath` field. Additionally, since you generated a PDF test report, the image is captured in `MyTestReport.pdf`. The test report also contains the path to the artifacts.

Tips

- The location of the saved figure is a folder with a name unique to a test run within the folder contained in the `ArtifactsRootFolder`. If you are running a test without a `TestRunner`, for example with `matlab.unittest.TestCase.forInteractiveUse`, the root folder is the value returned by `tempdir()`.
- To determine the path of the saved figure, access the `FileArtifact` object for a particular test result. For example, assume that `res` is a `TestResult` array. Determine the location of the saved figure for the first element of the array as follows.

  ```
  res(1).Details.DiagnosticRecord.TestDiagnosticResults.Artifacts
  ```

  ```
  ans =

    FileArtifact with properties:

          Name: "Figure_3984704d-b884-44c2-b3ee-7ed10d36e967.png"
  ```

```
    Location: "C:\mywork\Temp\a1f80242-8f8a-4678-9124-415980432d08"
    FullPath: "C:\mywork\Temp\a1f80242-8f8a-4678-9124-415980432d08\Figure_3984704d-b
```

See Also

matlab.unittest.TestRunner | matlab.unittest.diagnostics.FileArtifact

Introduced in R2017a

matlab.unittest.diagnostics.FileArtifact class

Package: `matlab.unittest.diagnostics`

Artifact associated with a file on disk

Description

Use the `FileArtifact` class to reference files on disk and to copy the files to a new location conveniently.

Construction

`artifact = FileArtifact(filename)` creates an artifact associated with a particular file on disk.

Input Arguments

`filename` — Name of file
character vector | string scalar

Name of file on disk, specified as a character vector or string scalar.

Example: `"myFile.txt"`

Properties

`Name` — Artifact name
string scalar

Artifact name, returned as a string scalar. For example, if the full path of a file artifact is `"C:\Hello\World.txt"`, the value of `Name` would be `"World.txt"`.

`Location` — Parent folder
string scalar

The parent folder where the artifact is located, returned as a string scalar.

Example: If the full path of a file artifact is `"C:\Hello\World.txt"`, the value of `Location` is `"C:\Hello"`.

`FullPath` — Full path to artifact
string scalar

The full path to the artifact, returned as a string scalar.

Example: `"C:\Hello\World.txt"`

Methods

copyTo	Copy artifacts to new location

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Create `FileArtifact` Instance

Create a file and write data to it.

```
filename = "MyFile.txt";
fileID = fopen(filename,'w');
fwrite(fileID,rand(5));
fclose(fileID);
```

Create a `FileArtifact` instance. In this example, the current working folder is `"C:\work"`.

```
import matlab.unittest.diagnostics.FileArtifact;
artifact = FileArtifact(filename)
```

```
artifact =
```

```
  FileArtifact with properties:

        Name: "MyFile.txt"
    Location: "C:\work\Desktop"
    FullPath: "C:\work\Desktop\MyFile.txt"
```

See Also

matlab.unittest.diagnostics.DiagnosticResult | matlab.unittest.diagnostics.FigureDiagnostic | matlab.unittest.diagnostics.ScreenshotDiagnostic

Introduced in R2017a

copyTo

Class: `matlab.unittest.diagnostics.FileArtifact`
Package: `matlab.unittest.diagnostics`

Copy artifacts to new location

Syntax

```
newArtifacts = copyTo(artifact,destination)
```

Description

`newArtifacts = copyTo(artifact,destination)` copies artifacts to a new location and returns an array of updated `matlab.unittest.diagnostics.Artifact` instances. The `copyTo` method sets the `Location` property on each instance of `newArtifacts` to `destination`.

Input Arguments

`artifact` — Artifacts to copy to new location
array of `matlab.unittest.diagnostics.FileArtifact` instances

Artifacts to copy to a new location, specified as an array of `matlab.unittest.diagnostics.FileArtifact` instances.

`destination` — New location for artifacts
character vector | string scalar

New location for the artifacts, specified as a character vector or string scalar.

Example: `tempdir()`

Example: `"mySubFolder`

Example: `'C:\work'`

Examples

Copy Artifacts to New Location

Create an array of file artifacts. This example assumes that files called `someFile.m` and `anotherFile.m` exist in your current working folder.

```
import matlab.unittest.diagnostics.FileArtifact;
fileArtifact1 = FileArtifact('someFile.m');
fileArtifact2 = FileArtifact('anotherFile.m');
artifacts = [fileArtifact1 fileArtifact2];
```

Copy the artifacts to a temporary folder.

```
destination = tempdir();
newArtifacts = artifacts.copyTo(destination);
```

See Also

Introduced in R2017a

matlab.unittest.diagnostics.FrameworkDiagnostic class

Package: `matlab.unittest.diagnostics`
Superclasses: `matlab.unittest.diagnostics.Diagnostic`

Diagnostic provided by testing framework

Description

The `FrameworkDiagnostic` class provides a diagnostic result from certain testing framework comparators, constraints, and tolerances. Only the test framework constructs this class directly.

See Also

`matlab.unittest.diagnostics.Diagnostic`

Introduced in R2018b

matlab.unittest.diagnostics.FunctionHandleDiagnostic class

Package: `matlab.unittest.diagnostics`
Superclasses: `matlab.unittest.diagnostics.Diagnostic`

Diagnostic using a function's displayed output

Description

The `FunctionHandleDiagnostic` class provides a diagnostic result using a function’s displayed output. This output is the same as the text displayed at the command prompt when MATLAB executes the function handle. When the diagnostic information is accessible through information displayed as output of the function handle, the `FunctionHandleDiagnostic` is a means to provide quick diagnostic information.

When using `matlab.unittest` qualifications, a function handle can be supplied directly as a test diagnostic. In this case, the testing framework automatically creates a `FunctionHandleDiagnostic` object.

Construction

`FunctionHandleDiagnostic(fcnHandle)` creates a new `FunctionHandleDiagnostic` instance.

Input Arguments

`fcnHandle`

The function handle that the `Diagnostic` uses to generate diagnostic information.

The resulting diagnostic information is equivalent to output displayed at the MATLAB command prompt. The result is packaged for consumption by the testing framework, which may or may not display the information at the command prompt.

Properties

Fcn

The function handle that the `Diagnostic` uses to generate diagnostic information, specified in the `fcnHandle` input argument. This property is read-only.

Inherited Properties

DiagnosticText

The `DiagnosticText` property provides the means by which the actual diagnostic information is communicated to consumers of diagnostics, such as testing frameworks. The property is a character vector that is defined during evaluation of the `diagnose` method.

Methods

Inherited Methods

diagnose	Execute diagnostic action
join	Join multiple diagnostics into a single array

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Create FunctionHandleDiagnostic Object

Create a diagnostic result that displays the output of the `dir` function when a test fails.

Create a folder in your current working folder.

```
mkdir('subfolderInCurrentFolder')
```

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.diagnostics.FunctionHandleDiagnostic

testCase = TestCase.forInteractiveUse;
```

Use a `FunctionHandleDiagnostic` to display diagnostic information upon test failure.

```
testCase.verifyThat(1, IsEqualTo(2), FunctionHandleDiagnostic(@dir))
```

```
Verification failed.

----------------
Test Diagnostic:
----------------

.                       ..                      subfolderInCurrentFolder



---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> NumericComparator failed.
    --> The numeric values are not equal using "isequaln".
    --> Failure table:
                Actual    Expected    Error    RelativeError
                ______    ________    _____    _____________

                1         2           -1       -0.5

Actual double:
         1
Expected double:
         2
```

Upon test failure, the diagnostic displays the contents of the current working folder. In this example output, the folder only contains the subfolder `subfolderInCurrentFolder`.

Alternatively, the test framework can create a `FunctionHandleDiagnostic` object for you from a function handle input to the `verifyThat` qualification.

```
testCase.verifyThat(1, IsEqualTo(2), @dir)
```

```
Verification failed.

----------------
Test Diagnostic:
```

```
----------------

.                        ..                       subfolderInCurrentFolder



---------------------
Framework Diagnostic:
---------------------
IsEqualTo failed.
--> NumericComparator failed.
    --> The numeric values are not equal using "isequaln".
    --> Failure table:
                Actual    Expected    Error    RelativeError
                ______    ________    _____    _____________

                1         2           -1       -0.5

Actual double:
         1
Expected double:
         2
```

The testing framework only creates the `FunctionHandleDiagnostic` object as needed, typically only in the event of a test failure.

See Also

`StringDiagnostic` | `matlab.unittest.diagnostics` | `matlab.unittest.plugins.DiagnosticsValidationPlugin`

Introduced in R2013a

matlab.unittest.diagnostics.LoggedDiagnosticEventData class

Package: `matlab.unittest.diagnostics`

Event data for DiagnosticLogged event listeners

Description

The `LoggedDiagnosticEventData` class holds event data for `DiagnosticLogged` event listeners. Invoking the `log` method within your tests triggers the `DiagnosticLogged` event listeners. Only the test framework constructs this class directly.

Properties

`Verbosity`

Verbosity level of the logged message, represented as a `matlab.unittest.Verbosity` enumeration.

`Timestamp`

Date and time of the call to the `log` method, represented as a `datetime` value.

`Diagnostic`

Diagnostic specified in the call to the `log` method, represented as a character vector, string, function handle, or instance of `matlab.unittest.diagnostics.Diagnostic`.

`Stack`

Function call stack leading up to the call to the `log` method, represented as a structure array.

`DiagnosticResults`

Results of diagnostics specified in a `matlab.unittest.TestCase.log` or `matlab.unittest.fixtures.Fixture.log` method call, represented as an array of `DiagnosticResult` instances.

See Also

`log` | `log` | `matlab.unittest.TestCase` | `matlab.unittest.fixtures.Fixture`

matlab.unittest.diagnostics.ScreenshotDiagnostic class

Package: `matlab.unittest.diagnostics`

Diagnostic to capture screen as image file

Description

Use the `ScreenshotDiagnostic` class to create a diagnostic that captures available screens as image files. The files persist after MATLAB completes the test run, and so they are available for post-test inspection.

Construction

`ScreenshotDiagnostic()` creates a diagnostic that captures available screens as image files. When the testing framework diagnoses the `ScreenshotDiagnostic` instance, it saves the screenshots to PNG files. The files each have a unique name consisting of a prefix (`Screenshot_'`, by default), an automatically generated identifier, and the file extension. An example file name is `Screenshot_cf95fe7f-5a7c-4310-9c49-16c0c18a969f.png`. To view the location of the files, access the `FileArtifact` object through the `TestResult` instance.

`ScreenshotDiagnostic('Prefix',prefix)` creates a diagnostic that saves screenshots to files with names that begin with a specified prefix.

Input Arguments

`prefix` — File name prefix
`'Screenshot_'` (default) | text

File name prefix, specified as text. If you do not specify a prefix, the default prefix is `'Screenshot_'`. Specify the value as a character vector or string scalar. Within the object, MATLAB stores them as character vectors.

Example: `'LoggedScreenshot_'`

Example: `"TestScreenshot-"`

Properties

`Prefix` — File name prefix
`'Screenshot_'` (default) | character vector

File name prefix, returned as a character vector. The default prefix is `'Screenshot_'`. The `Prefix` property is read-only, and its value is set during construction.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Create `ScreenshotDiagnostic` Object

Create a `TestCase` for interactive use.

```
import matlab.unittest.TestCase
testCase = TestCase.forInteractiveUse;
```

Use a `ScreenshotDiagnostic` to save an image of the desktop as a test diagnostic.

```
import matlab.unittest.diagnostics.ScreenshotDiagnostic
testCase.verifyTrue(false,ScreenshotDiagnostic)
```

```
Interactive verification failed.

----------------
Test Diagnostic:
----------------
Screenshot captured to:
--> C:\work\Temp\Screenshot_e99834ed-75e9-4ee1-9596-2f95e64b0ff1.png

---------------------
Framework Diagnostic:
```

```
----------------------
verifyTrue failed.
--> The value must evaluate to "true".

Actual logical:
       0
```

Specify Prefix for `ScreenshotDiagnostic` Object

Create a `TestCase` for interactive use.

```
import matlab.unittest.TestCase
testCase = TestCase.forInteractiveUse;
```

Use a `ScreenshotDiagnostic` to log an image of the desktop as a test diagnostic with a custom prefix

```
import matlab.unittest.diagnostics.ScreenshotDiagnostic
testCase.log(ScreenshotDiagnostic('Prefix','LoggedScreenshot_'))
```

```
Interactive diagnostic logged.
Screenshot captured to:
--> C:\work\Temp\LoggedScreenshot_35bb1ecb-441a-4f79-9f59-8b18ac3f9d12.png
```

Create Test with Screenshot Artifacts

In a file in your current working folder, create the `ScreenShotExampleTest` test class. When the `failingTest` test method fails (it always does in this example), it uses a `ScreenshotDiagnostic` to capture an image of the screen so you can examine it later. The `logScreenshotTest` test always captures an image of the screen and saves it in a file prefixed with `LoggedScreenshot_`.

```
classdef ScreenShotExampleTest < matlab.unittest.TestCase
    methods (Test)
        function passingTest(testCase)
            testCase.verifyEqual(7,4+3);
        end
        function failingTest(testCase)
            import matlab.unittest.diagnostics.ScreenshotDiagnostic
            testCase.verifyFalse(true,ScreenshotDiagnostic);
        end
```

```
        function logScreenshotTest(testCase)
            import matlab.unittest.diagnostics.ScreenshotDiagnostic
            testCase.verifySubstring("Some Long Message","sage");
            testCase.log(1,ScreenshotDiagnostic('Prefix','LoggedScreenshot_'));
        end
    end
end
```

Run the tests.

```
res = runtests('ScreenShotExampleTest');
```

```
Running ScreenShotExampleTest
.
================================================================================
Verification failed in ScreenShotExampleTest/failingTest.

    ----------------
    Test Diagnostic:
    ----------------
    Screenshot captured to:
    --> C:\Temp\bf0e26d1-8d47-4462-b1f2-673c7d35c236\Screenshot_b24f2219-61a4-4cf8-b4da

    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyFalse failed.
    --> The value must evaluate to "false".

    Actual logical:
           1

    ------------------
    Stack Information:
    ------------------
    In C:\work\ScreenShotExampleTest.m (ScreenShotExampleTest.failingTest) at 8
================================================================================
.   [Terse] Diagnostic logged (2016-12-22T11:21:54):
Screenshot captured to:
--> C:\Temp\bf0e26d1-8d47-4462-b1f2-673c7d35c236\LoggedScreenshot_26def240-9a9d-4147-9c

.
Done ScreenShotExampleTest
__________
```

```
Failure Summary:

     Name                                Failed  Incomplete  Reason(s)
    ================================================================================
     ScreenShotExampleTest/failingTest    X                  Failed by verification.
```

The framework diagnostics in the Command Window indicate the captured screenshots. You can also access these locations programmatically through the diagnostic records of the test results.

Since the second test fails, the screenshot is saved as part of the test diagnostics. Access the path to the screenshot through the `Artifacts` object on the `TestDiagnosticResults`.

```
res(2).Details.DiagnosticRecord.TestDiagnosticResults.Artifacts.FullPath
```

```
ans =

    "C:\Temp\bf0e26d1-8d47-4462-b1f2-673c7d35c236\Screenshot_b24f2219-61a4-4cf8-b4da-fe
```

The third test passes, and the screenshot is saved as part of a logged diagnostic (through the `log` method on the `TestCase`). Access the path to the screenshot through the `Artifacts` object on the `LoggedDiagnosticResults`.

```
res(3).Details.DiagnosticRecord.LoggedDiagnosticResults.Artifacts.FullPath
```

```
ans =

    "C:\Temp\bf0e26d1-8d47-4462-b1f2-673c7d35c236\LoggedScreenshot_26def240-9a9d-4147-9
```

Tips

- The location of the saved screenshots is a folder with a name unique to a test run within the folder contained in the `ArtifactsRootFolder`. If you are running a test interactively, the location of the root folder is the value returned by `tempdir()`.
- To determine the path of the saved screenshots, access the `FileArtifact` object for a particular test result. For example, assume that you have a `TestResult` array, `res`. Determine the location of the saved figure for the first element of the array as follows.

  ```
  res(1).Details.DiagnosticRecord.TestDiagnosticResults.Artifacts
  ```

  ```
  ans =
  ```

```
FileArtifact with properties:

      Name: "Screenshot_f51601ef-86bc-499c-bcec-203969f72a85.png"
  Location: "C:\work\Temp\1f4d3b64-3201-4bde-92ed-ad6859e97051"
  FullPath: "C:\work\Temp\1f4d3b64-3201-4bde-92ed-ad6859e97051\Screenshot_f51601ef
```

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.diagnostics.FileArtifact`

Introduced in R2017a

matlab.unittest.diagnostics.StringDiagnostic class

Package: `matlab.unittest.diagnostics`
Superclasses: `matlab.unittest.diagnostics.Diagnostic`

Diagnostic using string

Description

The `StringDiagnostic` class provides a diagnostic result that uses a string. When the diagnostic information is known at the time of construction, the `StringDiagnostic` is a means to provide quick diagnostic information.

When using `matlab.unittest` qualifications, a string can be supplied directly as a test diagnostic. In this case, the testing framework automatically creates a `StringDiagnostic` object.

Construction

`StringDiagnostic(diagString)` creates a new `StringDiagnostic` instance.

Input Arguments

`diagString`

The string that the `Diagnostic` uses to generate diagnostic information.

Methods

Inherited Methods

diagnose	Execute diagnostic action
join	Join multiple diagnostics into a single array

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Create `StringDiagnostic` Object

Create a test case for interactive testing.

```
import matlab.unittest.TestCase
import matlab.unittest.constraints.IsEqualTo
import matlab.unittest.diagnostics.StringDiagnostic

testCase = TestCase.forInteractiveUse;
```

Use a `StringDiagnostic` to display diagnostic information upon test failure.

```
testCase.verifyThat(1, IsEqualTo(2), ...
    StringDiagnostic('actual was supposed to be equal to expected') )
```

```
Verification failed.
    ----------------
    Test Diagnostic:
    ----------------
    actual was supposed to be equal to expected
    ---------------------
    Framework Diagnostic:
    ---------------------
    IsEqualTo failed.
    --> NumericComparator failed.
```

```
        --> The numeric values are not equal using "isequaln".
        --> Failure table:
                Actual    Expected    Error    RelativeError
                ______    ________    _____    _____________
                  1          2         -1          -0.5

        Actual Value:
             1
        Expected Value:
             2
```

Alternatively, the test framework can create a `StringDiagnostic` object for you from a string input to the `verifyThat` qualification.

```
testCase.verifyThat(1, IsEqualTo(2), ...
    'actual was supposed to be equal to expected' )
```

```
Verification failed.
    ----------------
    Test Diagnostic:
    ----------------
    actual was supposed to be equal to expected
    ---------------------
    Framework Diagnostic:
    ---------------------
    IsEqualTo failed.
    --> NumericComparator failed.
        --> The numeric values are not equal using "isequaln".
        --> Failure table:
                Actual    Expected    Error    RelativeError
                ______    ________    _____    _____________
                  1          2         -1          -0.5

        Actual Value:
             1
        Expected Value:
             2
```

The testing framework only creates the `StringDiagnostic` object as needed, typically only in the event of a test failure.

See Also

`FunctionHandleDiagnostic` | `matlab.unittest.diagnostics`

Introduced in R2013a

dialog

Create empty modal dialog box

Syntax

```
d = dialog
d = dialog(Name,Value)
```

Description

`d = dialog` creates an empty dialog box and returns `d`, a `Figure` object. Use the `uicontrol` function to add user interface controls to a dialog.

`d = dialog(Name,Value)` specifies one or more `Figure` property names and corresponding values. Use this syntax to override the default properties.

Examples

Dialog Containing Text and a Button

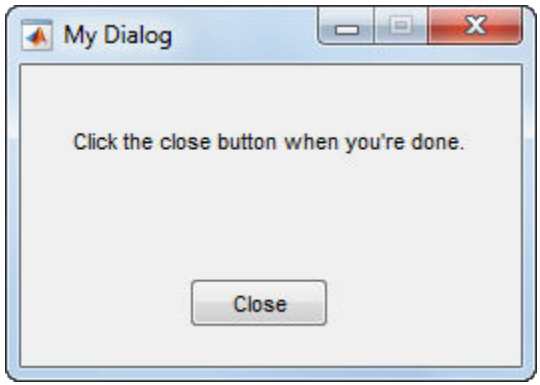
Use the `uicontrol` function to add user interface controls to a dialog box. For instance, create a program file called `mydialog.m` that displays a dialog containing text and a button.

```
function mydialog
    d = dialog('Position',[300 300 250 150],'Name','My Dialog');

    txt = uicontrol('Parent',d,...
               'Style','text',...
               'Position',[20 80 210 40],...
               'String','Click the close button when you''re done.');

    btn = uicontrol('Parent',d,...
               'Position',[85 20 70 25],...
```

```
                'String','Close',...
                'Callback','delete(gcf)');
end
```

Next, run the `mydialog` function from the Command Window.

```
mydialog
```



Dialog That Returns Output

Use the `uiwait` function to return output based on user selections in the dialog box. For instance, create a program file called `choosedialog.m` to perform these tasks:

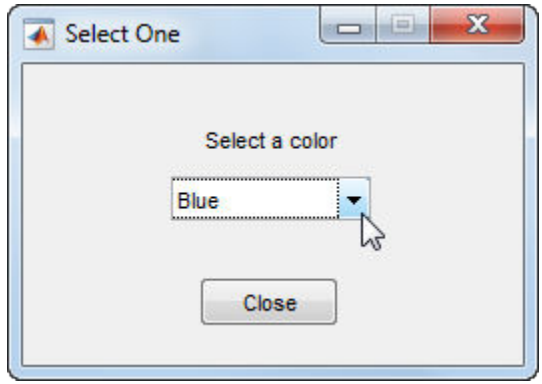
- Call the `dialog` function to create dialog with a specific size, location, and the title, “Select One”.
- Call the `uicontrol` function three times to add text, a pop-up menu, and a button, respectively.
- Define the function, `popup_callback`, to serve as the callback function for the button.
- Call the `uiwait` function to wait for the user to close the dialog before returning the output to the command line.

```
function choice = choosedialog

    d = dialog('Position',[300 300 250 150],'Name','Select One');
    txt = uicontrol('Parent',d,...
           'Style','text',...
```

```
            'Position',[20 80 210 40],...
            'String','Select a color');

    popup = uicontrol('Parent',d,...
            'Style','popup',...
            'Position',[75 70 100 25],...
            'String',{'Red';'Green';'Blue'},...
            'Callback',@popup_callback);

    btn = uicontrol('Parent',d,...
            'Position',[89 20 70 25],...
            'String','Close',...
            'Callback','delete(gcf)');

    choice = 'Red';

    % Wait for d to close before running to completion
    uiwait(d);

        function popup_callback(popup,event)
           idx = popup.Value;
           popup_items = popup.String;
           % This code uses dot notation to get properties.
           % Dot notation runs in R2014b and later.
           % For R2014a and earlier:
           % idx = get(popup,'Value');
           % popup_items = get(popup,'String');
           choice = char(popup_items(idx,:));
        end
end
```

Run the `choosedialog` function from the Command Window. Then, select a color in the dialog.

```
color = choosedialog
```



`choosedialog` returns the last selected color when you close the dialog.

```
color =

Blue
```

Note The `uiwait` function blocks the MATLAB thread. Although `uiwait` works well in a simple modal dialog, it is not recommended for use in more sophisticated applications.

Input Arguments

Name-Value Pair Arguments

Example: `'WindowStyle','normal'` sets the `WindowStyle` property to `'normal'`.

The properties listed here are only a subset. For a complete list, see Figure.

`Position` — Location and size of drawable area
`[left bottom width height]`

Location and size of the drawable area, specified as a vector of the form `[left bottom width height]`. This area excludes the figure borders, title bar, menu bar, and tool bars.

This table describes each element in the `Position` vector.

Element	Description
`left`	Distance from the left edge of the primary display to the inner left edge of the window. This value can be negative on systems that have more than one monitor. If the figure is docked, then this value is relative to the Figure panel within the MATLAB desktop.
`bottom`	Distance from the bottom edge of the primary display to the inner bottom edge of the window. This value can be negative on systems that have more than one monitor. If the figure is docked, then this value is relative to the Figure panel within the MATLAB desktop.
`width`	Distance between the right and left inner edges of the dialog.
`height`	Distance between the top and bottom inner edges of the window.

All measurements are in units specified by the `Units` property.

You cannot specify the figure `Position` property when the figure is docked.

In MATLAB Online, the `bottom` and `left` elements of the `Position` vector are ignored.

To place the full window, including the borders, title bar, menu bar, tool bars, use the `OuterPosition` property.

Note The Windows operating system enforces a minimum window width and a maximum window size. If you specify a figure size outside of those limits, the displayed figure will conform to the limits instead of the size you specified.

`ButtonDownFcn` — Button down callback

`''` (default) | function handle | cell array | character vector

Button down callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes whenever the user clicks a blank area of the figure.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

Use the `SelectionType` property to determine whether the user pressed modifier keys.

`WindowStyle` — Window style
`'normal'` (default) | `'modal'` | `'docked'`

Window style, specified as one of the following:

- `'normal'` — The figure window is independent of other windows, and the other windows are accessible while the figure is displaying.
- `'modal'` — The figure displays on top of all existing figure windows, making them inaccessible as long as the top figure exists and remains modal. However, any new figures created after a modal figure will display.

  When multiple modal windows exist, the most recently created window keeps focus and stays above all other windows until it becomes invisible, or is returned to a normal window style, or is deleted. At that time, focus reverts to the window that last had focus.
- `'docked'` — The figure displays in the desktop or a document window. When the `WindowStyle` property is set to `'docked'`, you cannot set the `DockControls` property to `'off'`. The `'docked'` option is not supported in MATLAB Online.

Note These are some important characteristics of the `WindowStyle` property and some recommended best practices:

- When you create UI windows, always specify the `WindowStyle` property. If you also want to set the `Resize`, `Position`, or `OuterPosition` properties of the figure, then set the `WindowStyle` property first.
- You can change the `WindowStyle` property of a figure at any time, including when the figure is visible and contains children. However on some systems, setting this property might cause the figure to flash or disappear and reappear, depending on the system's implementation of normal and modal windows. For best visual results, set the `WindowStyle` property at creation time or when the figure is invisible.

- Calling `reset` on a figure does not change the value of the `WindowStyle` property.

Modal Window Style Behavior

When `WindowStyle` is set to `'modal'`, the figure window traps all keyboard and mouse actions over all MATLAB windows as long as the windows are visible. Windows belonging to applications other than MATLAB are unaffected.

Typing **Ctrl+C** when a modal figure has focus causes that figure to revert to a `'normal'` `WindowStyle` property setting. This allows the user to type at the command line.

Figures with the `WindowStyle` property set to `'modal'` and the `Visible` property set to `'off'` do not behave modally until MATLAB makes them visible. Therefore, you can hide a modal window for later reuse, instead of destroying it.

Modal figures do not display menu children, built-in menus, or toolbars. But, it is not an error to create menus in a modal figure or to change the `WindowStyle` property setting to `'modal'` on a figure with menu children. The `Menu` objects exist and the figure retains them. If you reset the figure's `WindowStyle` property to `'normal'`, the menus display.

Output Arguments

d — Dialog window
`Figure` object

Dialog window, returned as a `Figure` object with these properties values set.

Property	Value
`ButtonDownFcn`	`'if isempty(allchild(gcbf)), close(gcbf), end'`
`Colormap`	`[]`
`DockControls`	`'off'`
`HandleVisibility`	`'callback'`
`IntegerHandle`	`'off'`
`InvertHardcopy`	`'off'`

Property	Value
MenuBar	'none'
Number	[]
NumberTitle	'off'
PaperPositionMode	'auto'
Resize	'off'
WindowStyle	'modal'

See Also

Functions
errordlg | msgbox | questdlg | waitfor | warndlg

Properties
Figure

Introduced before R2006a

diary

Log Command Window text to file

Syntax

```
diary
diary filename
diary off
diary on
```

Description

`diary` toggles logging on and off. When logging is on, MATLAB captures entered commands, keyboard input, and text output from the Command Window. It saves the resulting log to the current folder as an ASCII text file named `diary`.

To see whether logging is on, type `get(0,'Diary')`. MATLAB returns either `'on'` or `'off'`.

`diary filename` saves the resulting log to `filename`. If the file exists, MATLAB appends the text to the end of the file. To see the name of the current `diary` file, type `get(0,'DiaryFile')`.

`diary off` disables logging.

`diary on` enables logging using the current `diary` file name.

Examples

Log Statements and Output

Create a diary file and record several statements and their output.

Enable logging and save the resulting log to `myDiaryFile`.

```
diary myDiaryFile
```

Perform a calculation, and create and display a matrix of ones in the Command Window.

```
a = 1;
b = sin(a);

x = ones(4)
```

```
x =

     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1
```

Disable logging and display the log file in the Command Window.

```
diary off
type myDiaryFile
```

```
a = 1;
b = sin(a);
x = ones(4)

x =

     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1

diary off
```

Input Arguments

`filename` — Log file name
character vector | string

Log file name, specified as a character vector or string. `filename` can include a full path or a path relative to the current folder. Otherwise, MATLAB saves `filename` in the current folder.

Limitations

Because the output of `diary` is plain text, the log file does not exactly mirror what you see on screen:

- The diary file does not include graphics (figure windows).
- The diary file does not preserve syntax highlighting and font preferences.
- The diary file shows hidden components in the Command Window, such as hyperlink information generated with `matlab:`, in plain text. For example, if you enter this statement in the Command Window

  ```
  str = sprintf('%s%s', ...
     '<a href="matlab:magic(4)">', ...
     'Generate magic square</a>');
  disp(str)
  ```

  MATLAB displays

  Generate magic square

  However, the diary file, when viewed in a text editor, shows

  ```
  str = sprintf('%s%s', ...
     '<a href="matlab:magic(4)">', ...
     'Generate magic square</a>');
  disp(str)
  <a href="matlab:magic(4)">Generate magic square</a>
  ```

Tips

- To view the contents of the diary file with syntax highlighting, use the `type` function.

See Also

`evalc` | `type`

Topics

"Write to a Diary File"
"Set Command History Preferences"

Introduced before R2006a

diff

Differences and approximate derivatives

Syntax

```
Y = diff(X)
Y = diff(X,n)
Y = diff(X,n,dim)
```

Description

`Y = diff(X)` calculates differences between adjacent elements of `X` along the first array dimension whose size does not equal 1:

- If `X` is a vector of length `m`, then `Y = diff(X)` returns a vector of length `m-1`. The elements of `Y` are the differences between adjacent elements of `X`.

  ```
  Y = [X(2)-X(1) X(3)-X(2) ... X(m)-X(m-1)]
  ```

- If `X` is a nonempty, nonvector p-by-m matrix, then `Y = diff(X)` returns a matrix of size (p-1)-by-m, whose elements are the differences between the rows of `X`.

  ```
  Y = [X(2,:)-X(1,:); X(3,:)-X(2,:); ... X(p,:)-X(p-1,:)]
  ```

- If `X` is a 0-by-0 empty matrix, then `Y = diff(X)` returns a 0-by-0 empty matrix.

`Y = diff(X,n)` calculates the nth difference by applying the `diff(X)` operator recursively `n` times. In practice, this means `diff(X,2)` is the same as `diff(diff(X))`.

`Y = diff(X,n,dim)` is the nth difference calculated along the dimension specified by `dim`. The `dim` input is a positive integer scalar.

Examples

Differences Between Vector Elements

Create a vector, then compute the differences between the elements.

```
X = [1 1 2 3 5 8 13 21];
Y = diff(X)
```

```
Y = 1×7

     0     1     1     2     3     5     8
```

Note that Y has one fewer element than X.

Differences Between Matrix Rows

Create a 3-by-3 matrix, then compute the first difference between the rows.

```
X = [1 1 1; 5 5 5; 25 25 25];
Y = diff(X)
```

```
Y = 2×3

     4     4     4
    20    20    20
```

Y is a 2-by-3 matrix.

Multiple Differences

Create a vector and compute the second-order difference between the elements.

```
X = [0 5 15 30 50 75 105];
Y = diff(X,2)
```

```
Y = 1×5

     5     5     5     5     5
```

Differences Between Matrix Columns

Create a 3-by-3 matrix, then compute the first-order difference between the columns.

```
X = [1 3 5;7 11 13;17 19 23];
Y = diff(X,1,2)
```

```
Y = 3×2

     2     2
     4     2
     2     4
```

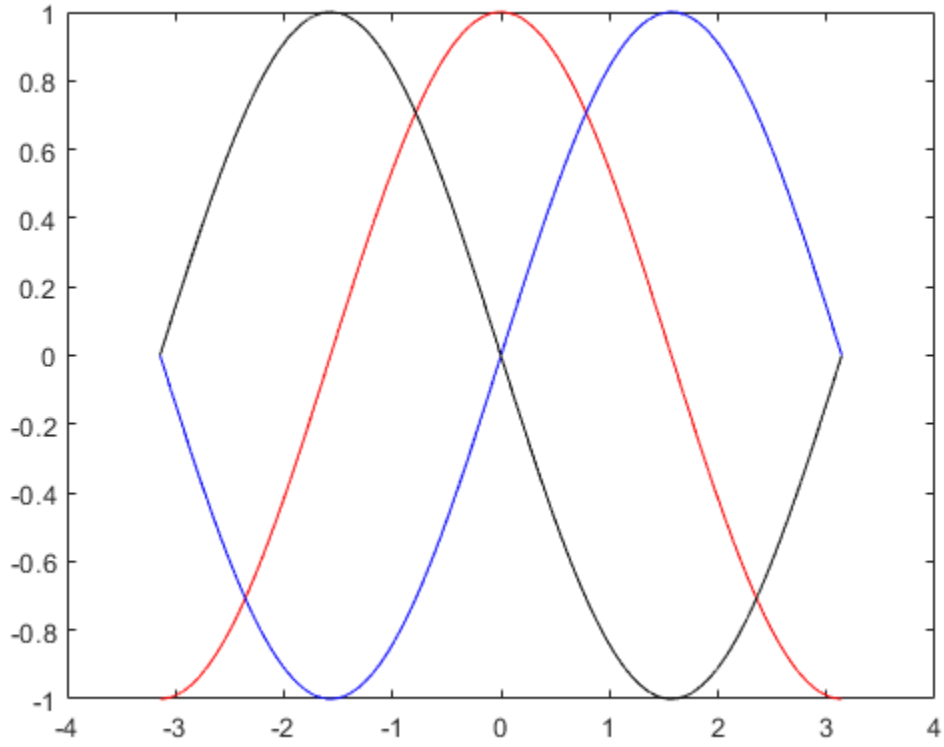
`Y` is a 3-by-2 matrix.

Approximate Derivatives with diff

Use the `diff` function to approximate partial derivatives with the syntax `Y = diff(f)/h`, where `f` is a vector of function values evaluated over some domain, `X`, and `h` is an appropriate step size.

For example, the first derivative of `sin(x)` with respect to `x` is `cos(x)`, and the second derivative with respect to `x` is `-sin(x)`. You can use `diff` to approximate these derivatives.

```
h = 0.001;        % step size
X = -pi:h:pi;     % domain
f = sin(X);       % range
Y = diff(f)/h;    % first derivative
Z = diff(Y)/h;    % second derivative
plot(X(:,1:length(Y)),Y,'r',X,f,'b', X(:,1:length(Z)),Z,'k')
```



In this plot the blue line corresponds to the original function, `sin`. The red line corresponds to the calculated first derivative, `cos`, and the black line corresponds to the calculated second derivative, `-sin`.

Differences Between Datetime Values

Create a sequence of equally-spaced datetime values, and find the time differences between them.

```
t1 = datetime('now');
t2 = t1 + minutes(5);
t = t1:minutes(1.5):t2
```

```
t = 1x4 datetime array
Columns 1 through 3

   27-Aug-2019 00:16:44   27-Aug-2019 00:18:14   27-Aug-2019 00:19:44

Column 4

   27-Aug-2019 00:21:14
```

```
dt = diff(t)
```

```
dt = 1x3 duration array
   00:01:30   00:01:30   00:01:30
```

`diff` returns a `duration` array.

Input Arguments

X — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array. `X` can be a numeric array, logical array, datetime array, or duration array.
Complex Number Support: Yes

n — Difference order
positive integer scalar | `[]`

Difference order, specified as a positive integer scalar or `[]`. The default value of `n` is 1.

It is possible to specify `n` sufficiently large so that `dim` reduces to a single (`size(X,dim) = 1`) dimension. When this happens, `diff` continues calculating along the next array dimension whose size does not equal 1. This process continues until a 0-by-0 empty matrix is returned.

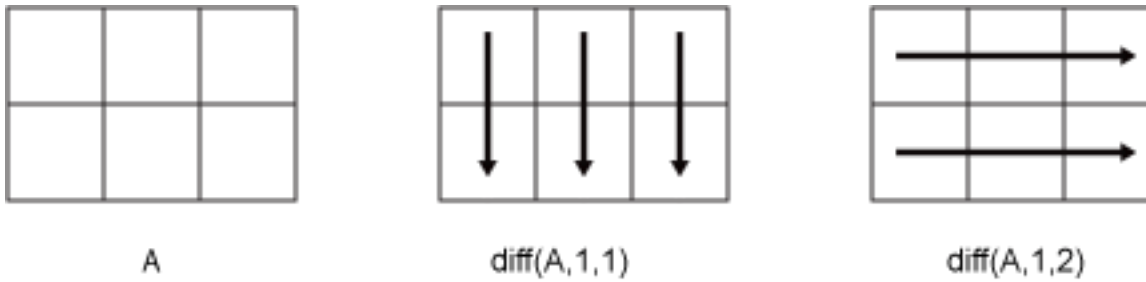
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a two-dimensional p-by-m input array, `A`:

- `diff(A,1,1)` works on successive elements in the columns of `A` and returns a (p-1)-by-m difference matrix.
- `diff(A,1,2)` works on successive elements in the rows of `A` and returns a p-by-(m-1) difference matrix.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`Y` — Difference array
scalar | vector | matrix | multidimensional array

Difference array, returned as a scalar, vector, matrix, or multidimensional array. If `X` is a nonempty array, then the dimension of `X` acted on by `diff` is reduced in size by `n` in the output.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

You must use the three-input syntax `Y = diff(X,N,dim)`.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If supplied, the arguments representing the number of times to apply `diff` and the dimension along which to calculate the difference must be constants.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).
- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cumsum` | `gradient` | `prod` | `sum`

Introduced before R2006a

diffuse

Calculate diffuse reflectance

Syntax

```
R = diffuse(Nx,Ny,Nz,S)
```

Description

`R = diffuse(Nx,Ny,Nz,S)` returns the reflectance of a surface with normal vector components `[Nx,Ny,Nz]`. `S` specifies the direction to the light source. You can specify these directions as three vectors `[x,y,z]` or two vectors `[Theta Phi` (in spherical coordinates).

Lambert's Law: `R = cos(PSI)` where `PSI` is the angle between the surface normal and light source.

See Also

`specular` | `surfl` | `surfnorm`

Topics

"Lighting Overview"

condensation

Graph condensation

Syntax

```
C = condensation(G)
```

Description

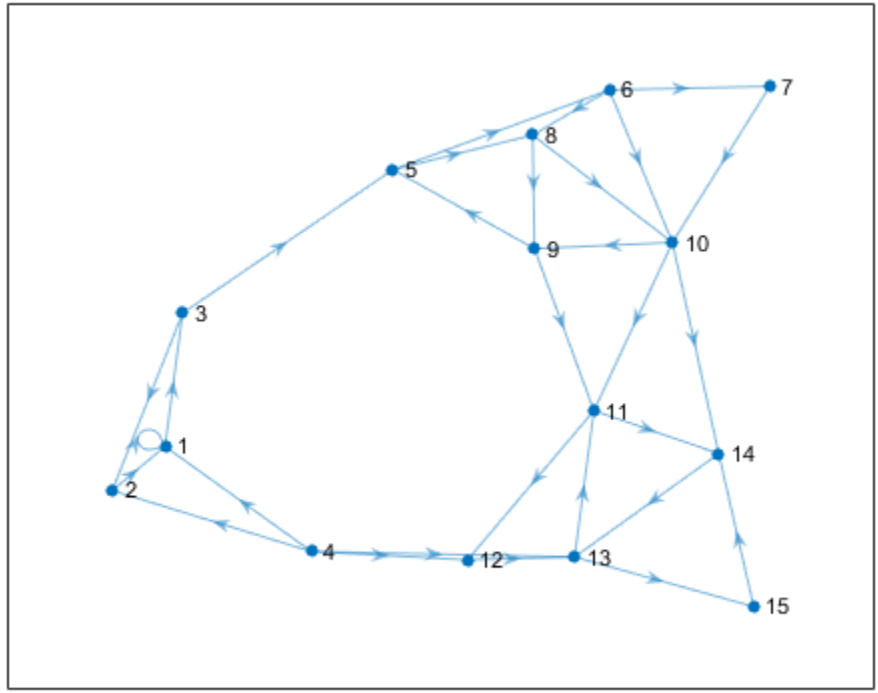
`C = condensation(G)` returns a directed graph `C` whose nodes represent the strongly connected components in `G`. This reduction provides a simplified view of the connectivity between components.

Examples

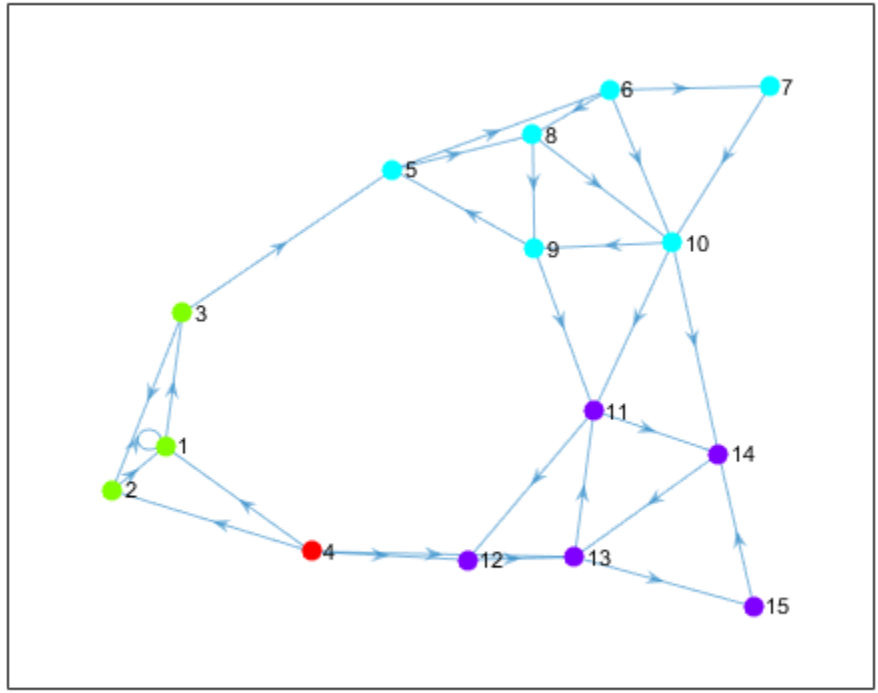
Condense Strongly Connected Components into Single Nodes

Create and plot a graph that contains several strongly connected components. Highlight the strongly connected components.

```
s = [1 1 2 3 3 4 4 4 4 5 5 6 6 6 7 8 8 9 9 10 10 10 11 11 12 13 13 14 15];
t = [1 3 1 2 5 1 2 12 13 6 8 7 8 10 10 9 10 5 11 9 11 14 12 14 13 11 15 13 14];
G = digraph(s,t);
p = plot(G);
```

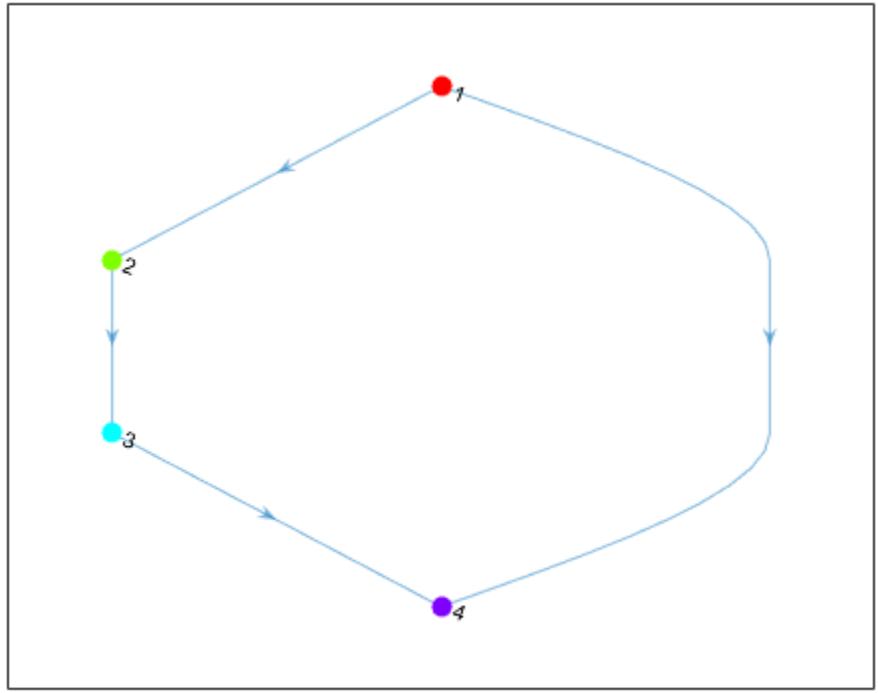



```
bins = conncomp(G);
p.MarkerSize = 7;
p.NodeCData = bins;
colormap(hsv(4))
```



Use `condensation` to represent each component as a single node. Color the nodes based on the components they represent.

```
C = condensation(G);
p2 = plot(C);
p2.MarkerSize = 7;
p2.NodeCData = 1:4;
colormap(hsv(4))
```



Input Arguments

G — Input graph
`digraph` object

Input graph, specified as a `digraph` object. Use `digraph` to create a directed graph object.

Example: `G = digraph([1 2],[2 3])`

Output Arguments

C — Condensation graph
digraph object

Condensation graph, returned as a digraph object. C is a directed acyclic graph (DAG), and is topologically sorted. The node numbers in C correspond to the bin numbers returned by conncomp.

condensation determines the nodes and edges in C by the components and connectivity in G:

- C contains a node for each strongly connected component in G.
- C contains an edge between node I and node J if there is an edge from any node in component I to any node in component J of G.

See Also

bctree | conncomp

Topics

"Directed and Undirected Graphs"
"Visualize Breadth-First and Depth-First Search"

Introduced in R2016b

flipedge

Reverse edge directions

Syntax

```
H = flipedge(G)
H = flipedge(G,s,t)
H = flipedge(G,idx)
```

Description

`H = flipedge(G)` returns a directed graph that has the same edges as `G`, but with reversed directions. `H` contains the same node and edge properties as `G`.

`H = flipedge(G,s,t)` reverses a subset of edges using the node pairs `s` and `t`.

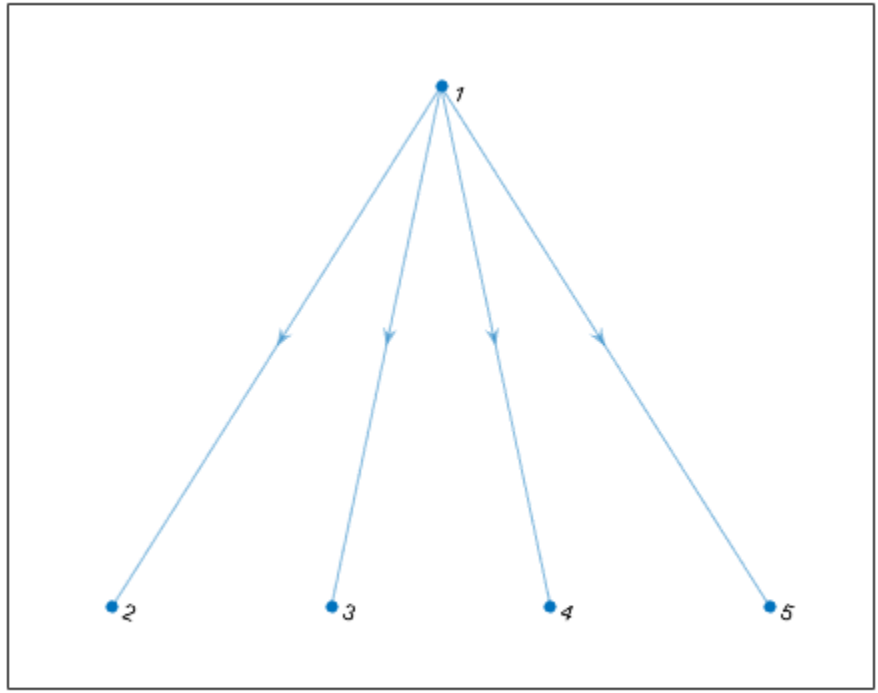
`H = flipedge(G,idx)` reverses a subset of edges using the edge indices `idx`.

Examples

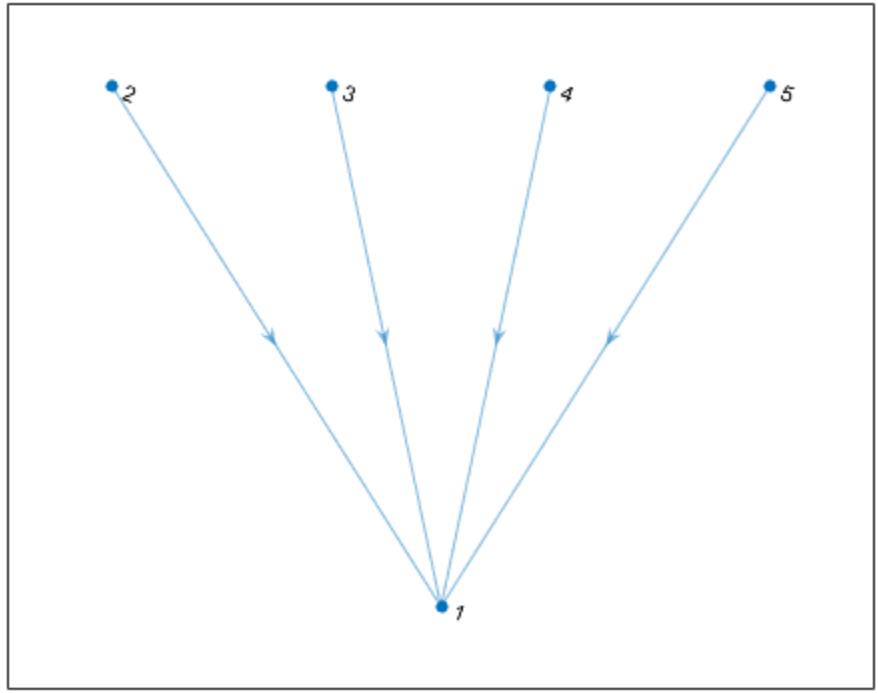
Reverse All Edges in Graph

Create and plot a directed graph. Then reverse the direction of all of the edges in the graph.

```
G = digraph([1 1 1 1],[2 3 4 5]);
plot(G)
```



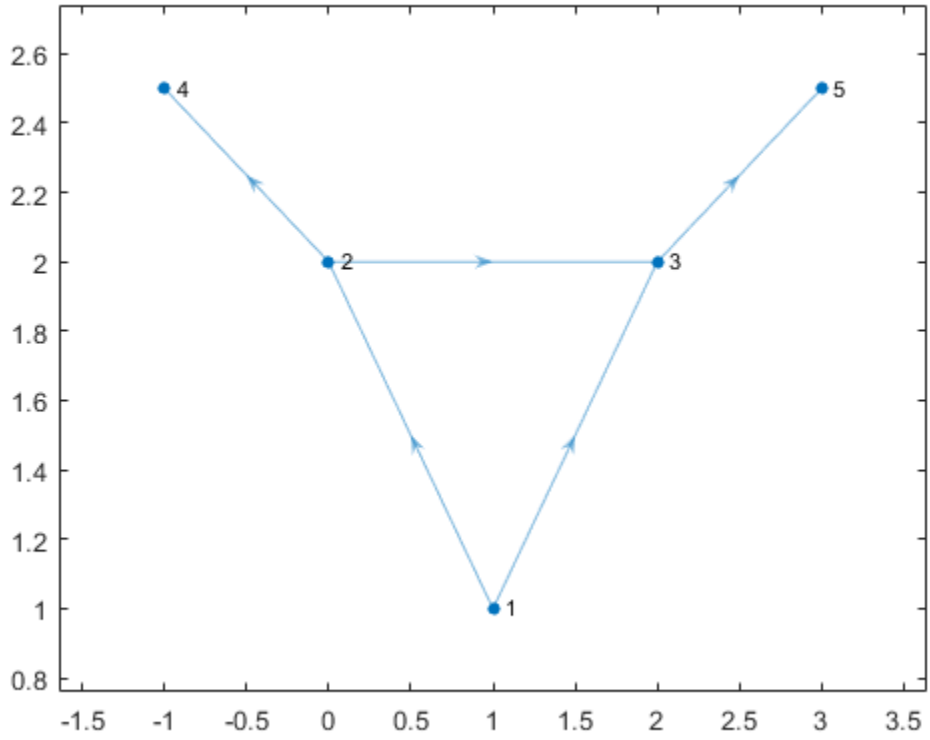

```
H = flipedge(G);
plot(H)
```



Reverse Subset of Graph Edges

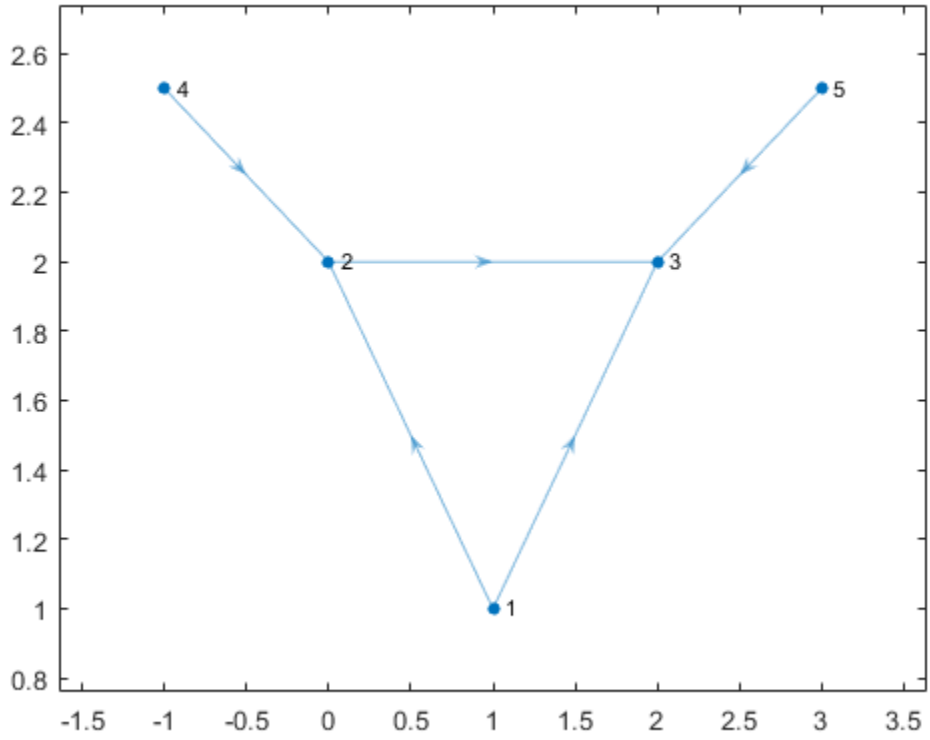
Create and plot a directed graph. Specify custom *xy* node coordinates for the plot.

```
G = digraph([1 1 2 2 3],[2 3 3 4 5]);
x = [1 0 2 -1 3];
y = [1 2 2 2.5 2.5];
plot(G,'XData',x,'YData',y)
```



Reverse the direction of the edges (2,4) and (3,5), and then replot the graph.

```
H = flipedge(G,[2 3],[4 5]);
plot(H,'XData',x,'YData',y)
```



Input Arguments

G — Input graph
`digraph` object

Input graph, specified as a `digraph` object. Use `digraph` to create a directed graph object.

Example: `G = digraph([1 2],[2 3])`

s,t — Node pairs (as separate arguments)
node indices | node names

Node pairs, specified as separate arguments of node indices or node names. Similarly located elements in `s` and `t` specify the source and target nodes for edges in the graph.

This table shows the different ways to refer to one or more nodes either by their numeric node indices or by their node names.

Form	Single Node	Multiple Nodes
Node index	Scalar **Example:** `1`	Vector **Example:** `[1 2 3]`
Node name	Character vector **Example:** `'A'`	Cell array of character vectors **Example:** `{'A' 'B' 'C'}`
	String scalar **Example:** `"A"`	String array **Example:** `["A" "B" "C"]`

Example: `H = flipedge(G,[1 4],[2 3])` reverses the direction of the edges (1,2) and (4,3).

Example: `H = flipedge(G,{'a' 'd'},{'b' 'c'})` reverses the direction of the edges (a,b) and (d,c).

`idx` — Edge indices
scalar | vector

Edge indices, specified as a scalar or vector of positive integers. Each edge index corresponds to a row in the `G.Edges` table of the graph `G.Edges(idx,:)`.

Example: `H = flipedge(G,3)` flips the direction of edge `G.Edges(3,:)`.

Output Arguments

`H` — Output graph
`digraph` object

Output graph, returned as a `digraph` object. Compared to `G`, `H` has the same nodes but some or all of the edges are in reversed direction. `H` also has the same node and edge properties as `G`.

See Also

`addedge` | `digraph` | `findedge` | `rmedge`

Topics

"Directed and Undirected Graphs"
"Modify Nodes and Edges of Existing Graph"
"Add Graph Node Names, Edge Weights, and Other Attributes"

Introduced in R2016b

indegree

In-degree of nodes

Syntax

```
D = indegree(G)
D = indegree(G,nodeIDs)
```

Description

`D = indegree(G)` returns a column vector containing the in-degree of each node in `G`.

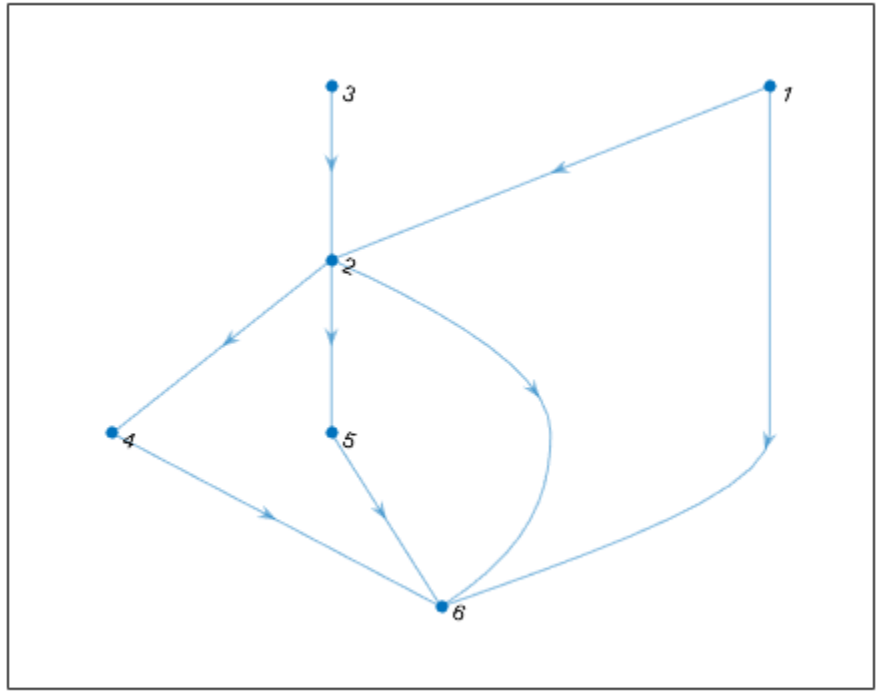
`D = indegree(G,nodeIDs)` returns the in-degree of the nodes specified by `nodeIDs`.

Examples

In-degree of All Graph Nodes

Create and plot a directed graph, and then compute the in-degree of every node in the graph. The in-degree of a node is equal to the number of edges with that node as the target.

```
s = [1 3 2 2 4 5 1 2];
t = [2 2 4 5 6 6 6 6];
G = digraph(s,t);
plot(G)
```




```
indeg = indegree(G)
```

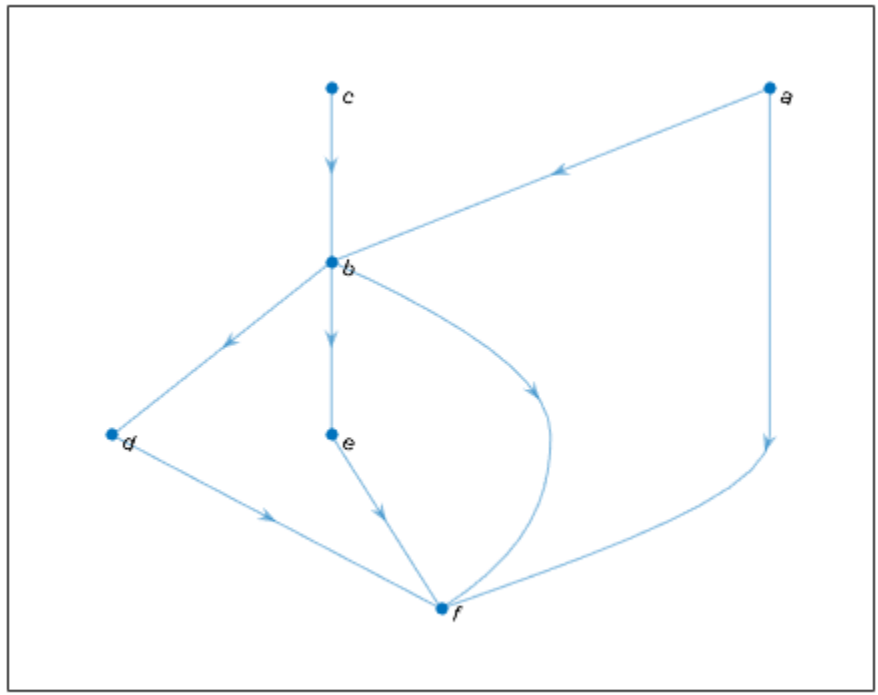
```
indeg = 6×1

     0
     2
     0
     1
     1
     4
```

`indeg(j)` indicates the in-degree of node `j`.

In-degree of Subset of Graph Nodes

Create and plot a directed graph with named nodes. Then compute the number of edges that have the `'a'`, `'b'`, and `'f'` nodes as their target.

```
s = {'a' 'c' 'b' 'b' 'd' 'e' 'a' 'b'};
t = {'b' 'b' 'd' 'e' 'f' 'f' 'f' 'f'};
G = digraph(s,t);
plot(G)
```




```
nodeID = {'a' 'b' 'f'}';
indeg = indegree(G,nodeID)
```

```
indeg = 3×1

     0
     2
     4
```

`indeg(j)` indicates the in-degree of node `nodeID(j)`.

Input Arguments

`G` — Input graph
`digraph` object

Input graph, specified as a `digraph` object. Use `digraph` to create a directed graph object.

Example: `G = digraph([1 2],[2 3])`

`nodeIDs` — Node identifiers
node indices | node names

Node identifiers, specified as one or more node indices or node names.

This table shows the different ways to refer to one or more nodes either by their numeric node indices or by their node names.

Form	Single Node	Multiple Nodes
Node index	Scalar **Example:** `1`	Vector **Example:** `[1 2 3]`
Node name	Character vector **Example:** `'A'`	Cell array of character vectors **Example:** `{'A' 'B' 'C'}`
	String scalar **Example:** `"A"`	String array **Example:** `["A" "B" "C"]`

Example: `indegree(G,1)`

Example: `indegree(G,["A" "B" "C"])`

Output Arguments

`D` — In-degree of nodes
array

In-degree of nodes, returned as a numeric array. `D` is a column vector unless you specify `nodeIDs`, in which case `D` has the same size as `nodeIDs`.

See Also

`digraph` | `outdegree` | `predecessors`

Introduced in R2015b

inedges

Incoming edges to node

Syntax

```
eid = inedges(G,nodeID)
[eid,nid] = inedges(G,nodeID)
```

Description

`eid = inedges(G,nodeID)` returns the indices of all incoming edges to node `nodeID` in directed graph `G`.

`[eid,nid] = inedges(G,nodeID)` additionally returns the node IDs of the predecessor nodes connected to `nodeID` by the edges in `eid`.

Examples

Incoming Edges of Selected Node

Create a multigraph with three nodes and four edges. Find the incoming edges of node 3.

```
G = digraph([1 1 1 2],[2 2 3 3]);
G.Edges
```

```
ans=4×1 table
    EndNodes
    ________

     1    2
     1    2
     1    3
     2    3
```

```
eid = inedges(G,3)
```

```
eid = 2×1

     3
     4
```

```
G.Edges(eid,:)
```

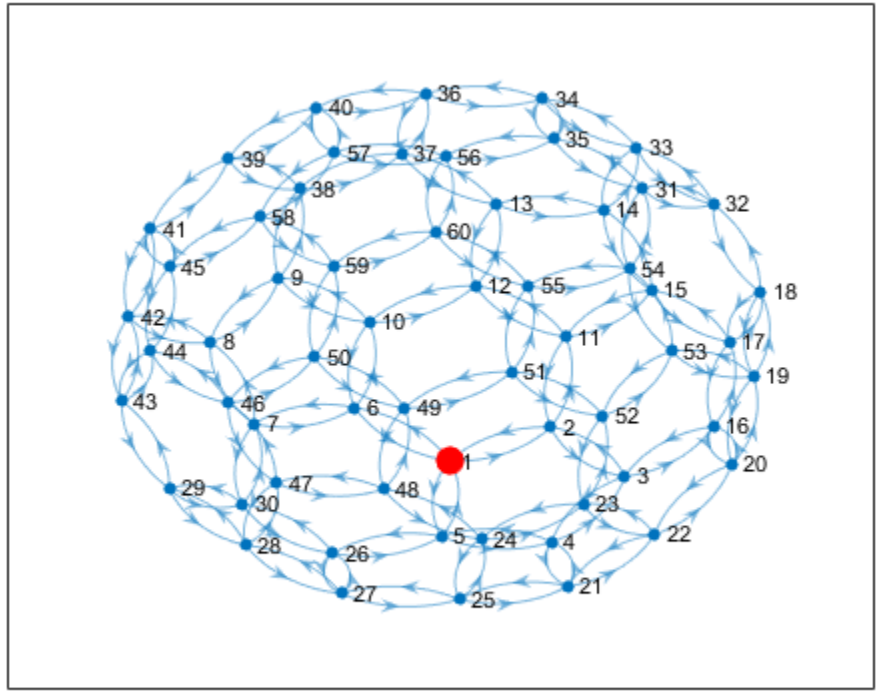
```
ans=2×1 table
    EndNodes
    ________

     1    3
     2    3
```

Find Incoming Edges and Node Predecessors

Plot a graph and highlight the incoming edges and predecessors of a selected node.

Create and plot a directed graph using the `bucky` adjacency matrix. Highlight node 1 for reference.

```
G = digraph(bucky);
p = plot(G);
highlight(p,1,'NodeColor','r','MarkerSize',10)
```



Determine the incoming edges and predecessors of node 1. Highlight these nodes and edges.

```
[eid,nid] = inedges(G,1)
```

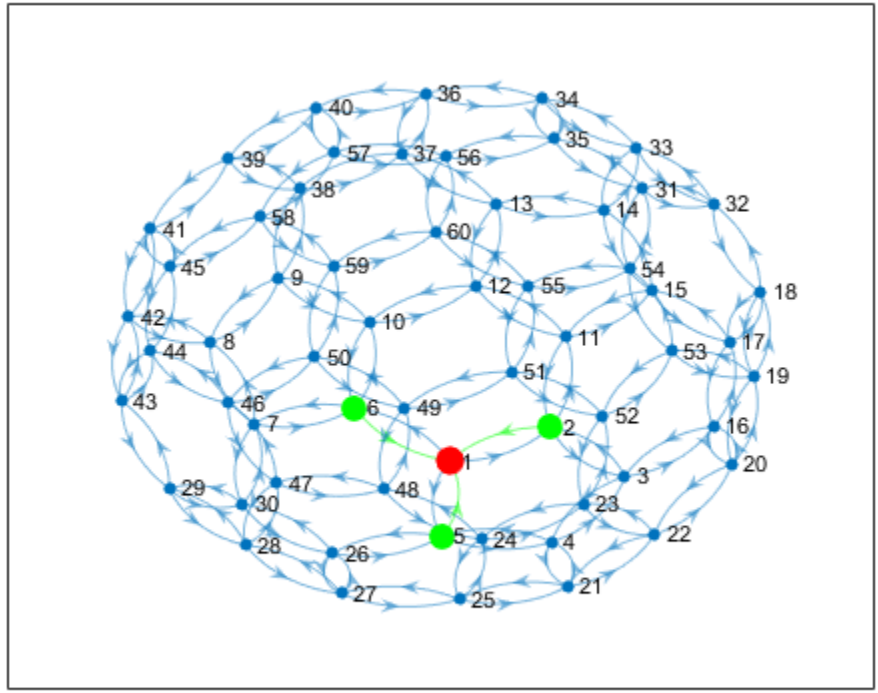
```
eid = 3×1

     4
    13
    16


nid = 3×1
```

```
     2
     5
     6
```

```
X = G.Edges(eid,:)
```

```
X=3×2 table
    EndNodes    Weight
    ________    ______

     2    1       1
     5    1       1
     6    1       1
```

```
highlight(p,nid,'NodeColor','g','MarkerSize',9)
highlight(p,'Edges',eid,'EdgeColor','g')
```



Input Arguments

G — Input graph
`digraph` object

Input graph, specified as a `digraph` object. Use `digraph` to create a directed graph object.

Example: `G = digraph([1 2],[2 3])`

nodeID — Node identifier
node index | node name

Node identifier, specified as one of the values in this table.

Value	Example
Scalar node index	1
Character vector node name	'A'
String scalar node name	"A"

Example: `inedges(G,1)`

Example: `inedges(G,'A')`

Output Arguments

`eid` — Edge indices
column vector

Edge indices, returned as a column vector. You can use the edge indices to index into the edges table of the graph with `G.Edges(eid,:)`.

`nid` — Node IDs of predecessors
node indices | node names

Node IDs of predecessors, returned as node indices if `nodeID` is numeric, or as node names if `nodeID` is a node name. Use `findnode(G,nid)` to convert node names into node indices. You can use node indices to index into the nodes table of the graph with `G.Nodes(nid,:)`.

The node IDs in `nid` are the same as those returned by the `predecessors` function. However, if there are multiple incoming edges from the same node, this node is listed more than once in `nid`.

Tips

- By convention, for undirected graphs, all edges incident to a node are considered to be outgoing edges. Use `outedges` with undirected graphs.
- For graphs with multiple edges, `inedges` and `predecessors` can return arrays of different lengths, since there can be multiple incoming edges from some of the predecessors.

See Also

`digraph` | `outedges` | `predecessors`

Introduced in R2018a

isdag

Determine if graph is acyclic

Syntax

```
tf = isdag(G)
```

Description

`tf = isdag(G)` returns logical `1` (`true`) if `G` is a directed acyclic graph on page 1-3444; otherwise, it returns logical `0` (`false`).

Examples

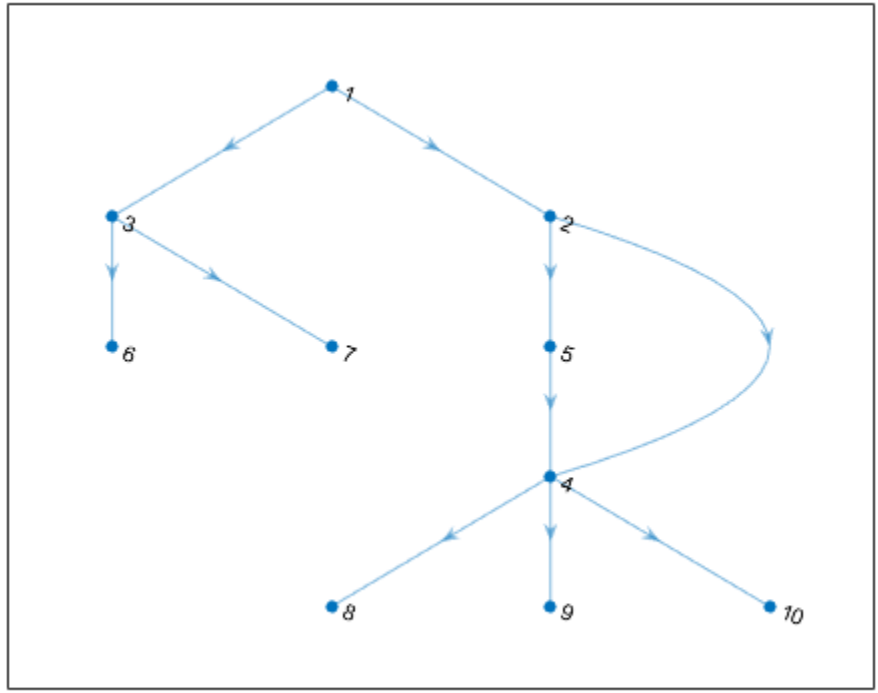
Determine if Directed Graph is Acyclic

Create and plot a directed graph, and then test the graph to determine if it is acyclic.

```
s = [1 1 2 2 3 3 4 4 4 5];
t = [2 3 4 5 6 7 8 9 10 4];
G = digraph(s,t)
```

```
G = 
  digraph with properties:

    Edges: [10x1 table]
    Nodes: [10x0 table]
```

```
plot(G)
```




```
tf = isdag(G)
```

```
tf = logical
   1
```

Input Arguments

G — Input graph
`digraph` object

Input graph, specified as a `digraph` object. Use `digraph` to create a directed graph object.

Example: `G = digraph([1 2],[2 3])`

More About

Directed Acyclic Graph (DAG)

A directed graph is acyclic if it contains no cycles. That is, starting at any node in the graph, no sequence of edges exists that can be followed to loop back to that starting node. As a result, directed acyclic graphs do not contain any self-loops.

See Also

`digraph` | `reordernodes` | `toposort`

Introduced in R2015b

outdegree

Out-degree of nodes

Syntax

```
D = outdegree(G)
D = outdegree(G,nodeIDs)
```

Description

`D = outdegree(G)` returns a column vector containing the out-degree of each node in `G`.

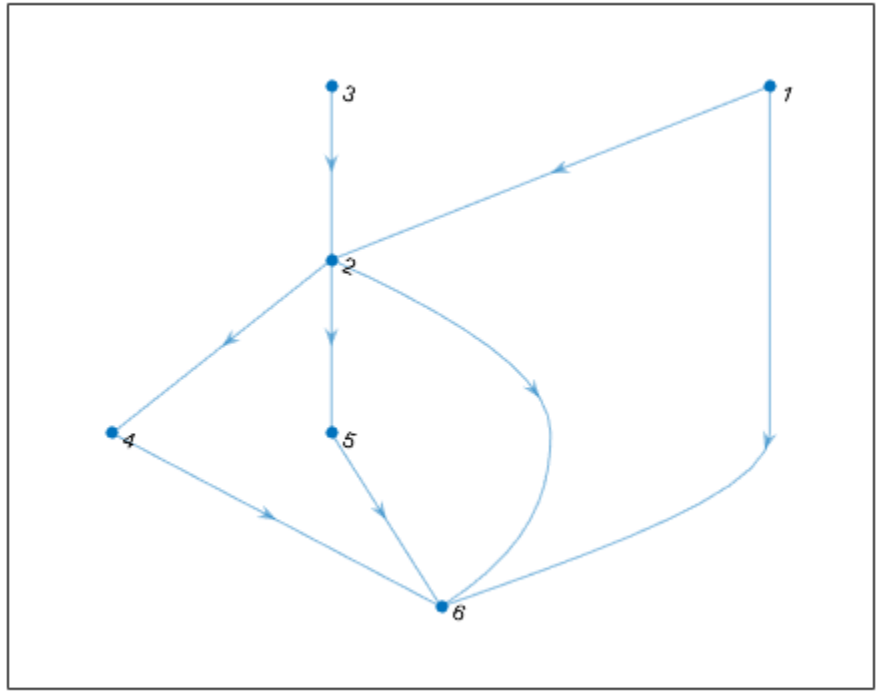
`D = outdegree(G,nodeIDs)` returns the out-degree of the nodes specified by `nodeIDs`.

Examples

Out-degree of All Graph Nodes

Create and plot a directed graph, and then compute the out-degree of every node in the graph. The out-degree of a node is equal to the number of edges with that node as the source.

```
s = [1 3 2 2 4 5 1 2];
t = [2 2 4 5 6 6 6 6];
G = digraph(s,t);
plot(G)
```




```
outdeg = outdegree(G)
```

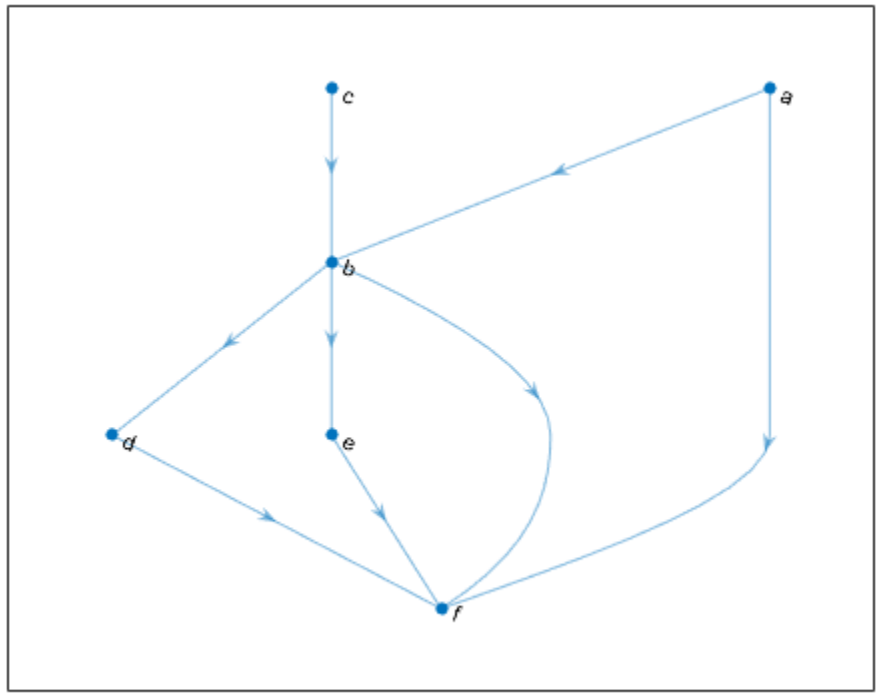
```
outdeg = 6×1

     2
     3
     1
     1
     1
     0
```

`outdeg(j)` indicates the out-degree of node `j`.

Out-degree of Subset of Graph Nodes

Create and plot a directed graph with named nodes. Then compute the number of edges that have the `'a'`, `'b'`, and `'f'` nodes as their source.

```
s = {'a' 'c' 'b' 'b' 'd' 'e' 'a' 'b'};
t = {'b' 'b' 'd' 'e' 'f' 'f' 'f' 'f'};
G = digraph(s,t);
plot(G)
```




```
nodeID = {'a' 'b' 'f'}';
outdeg = outdegree(G,nodeID)
```

```
outdeg = 3×1

     2
     3
     0
```

`outdeg(j)` indicates the out-degree of node `nodeID(j)`.

Input Arguments

`G` — Input graph
`digraph` object

Input graph, specified as a `digraph` object. Use `digraph` to create a directed graph object.

Example: `G = digraph([1 2],[2 3])`

`nodeIDs` — Node identifiers
node indices | node names

Node identifiers, specified as one or more node indices or node names.

This table shows the different ways to refer to one or more nodes either by their numeric node indices or by their node names.

Form	Single Node	Multiple Nodes
Node index	Scalar **Example:** `1`	Vector **Example:** `[1 2 3]`
Node name	Character vector **Example:** `'A'`	Cell array of character vectors **Example:** `{'A' 'B' 'C'}`
	String scalar **Example:** `"A"`	String array **Example:** `["A" "B" "C"]`

Example: `outdegree(G,1)`

Example: `outdegree(G,["A" "B" "C"])`

Output Arguments

`D` — Out-degree of nodes
array

Out-degree of nodes, returned as a numeric array. `D` is a column vector unless you specify `nodeIDs`, in which case `D` has the same size as `nodeIDs`.

See Also

`digraph` | `indegree` | `successors`

Introduced in R2015b

predecessors

Node predecessors

Syntax

```
preIDs = predecessors(G,nodeID)
```

Description

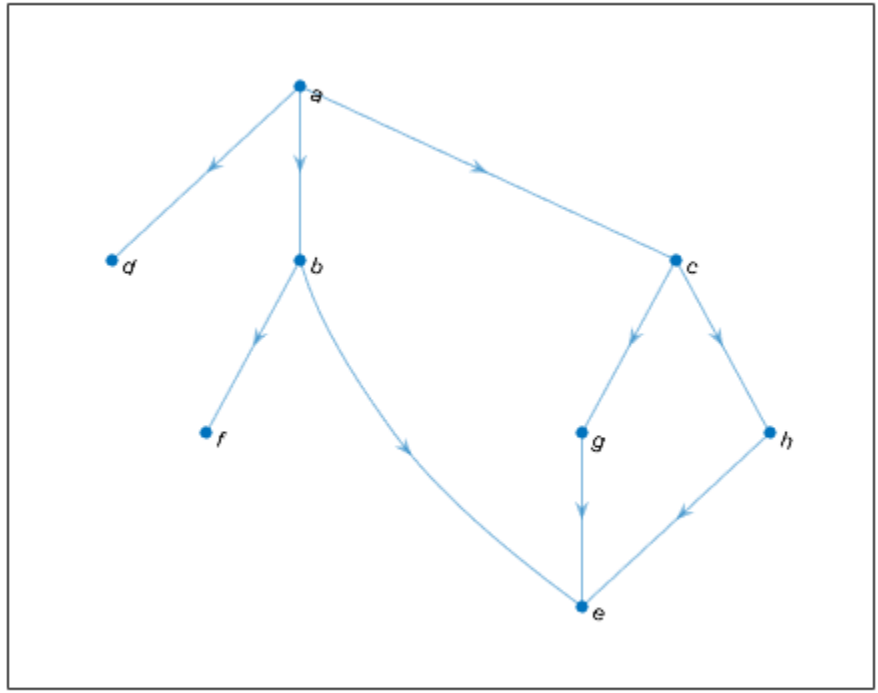
`preIDs = predecessors(G,nodeID)` returns the predecessor nodes of the node in directed graph `G` specified by `nodeID`. The predecessor nodes form directed edges with `preIDs` as the source nodes, and `nodeID` as the target node.

Examples

Node Predecessors

Create and plot a directed graph and determine the predecessor nodes of node `'e'`.

```
s = [1 1 1 2 2 3 3 7 8];
t = [2 3 4 5 6 7 8 5 5];
names = {'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h'};
G = digraph(s,t,[],names);
plot(G)
```




```
preIDs = predecessors(G,'e')
```

```
preIDs = 3x1 cell array
    {'b'}
    {'g'}
    {'h'}
```

Input Arguments

G — Input graph
`digraph` object

Input graph, specified as a `digraph` object. Use `digraph` to create a directed graph object.

Example: `G = digraph([1 2],[2 3])`

`nodeID` — Node identifier
node index | node name

Node identifier, specified as one of the values in this table.

Value	Example
Scalar node index	`1`
Character vector node name	`'A'`
String scalar node name	`"A"`

Example: `preIDs = predecessors(G,3)` finds the predecessor nodes of node 3.

Output Arguments

`preIDs` — Predecessor node IDs
node indices | node names

Predecessor node IDs, returned as a node indices or node names.

- If `nodeID` is a numeric node index, then `preIDs` contains numeric node indices.
- If `nodeID` is a node name, then `preIDs` contains node names.

Tips

- For multigraphs with multiple edges between the same two nodes, the output of `inedges` can be longer than that of `predecessors`, since there can be multiple incoming edges from some of the predecessors.

See Also

`digraph` | `indegree` | `inedges` | `neighbors` | `successors`

Introduced in R2015b

successors

Node successors

Syntax

```
sucIDs = successors(G,nodeID)
```

Description

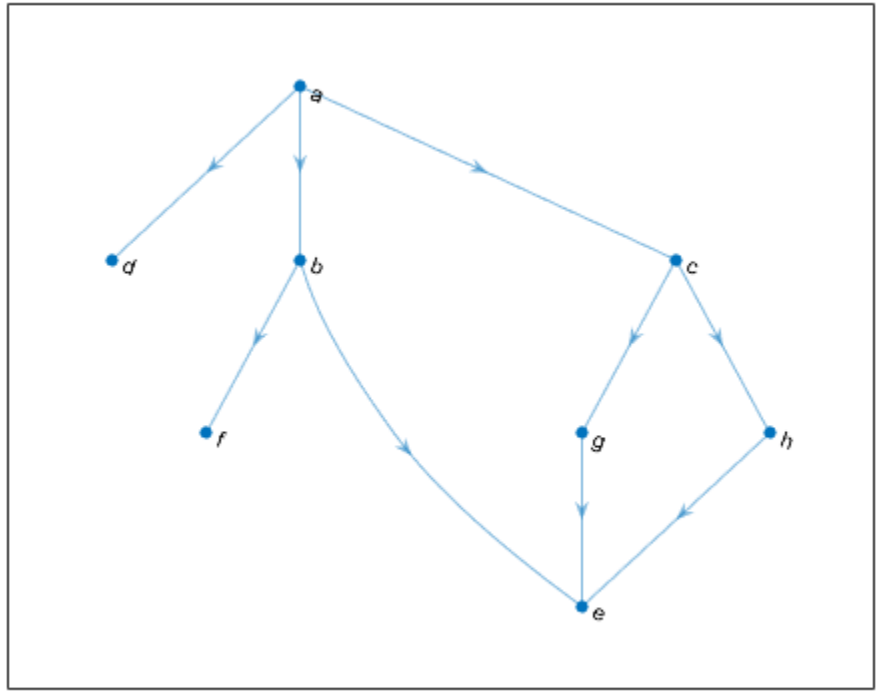
`sucIDs = successors(G,nodeID)` returns the successor nodes of the node in directed graph `G` specified by `nodeID`. The successor nodes form directed edges with `nodeID` as the source node, and `sucIDs` as the target nodes.

Examples

Node Successors

Create and plot a directed graph and determine the successor nodes of node `'a'`.

```
s = [1 1 1 2 2 3 3 7 8];
t = [2 3 4 5 6 7 8 5 5];
names = {'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h'};
G = digraph(s,t,[],names);
plot(G)
```




```
sucIDs = successors(G,'a')
```

```
sucIDs = 3x1 cell array
    {'b'}
    {'c'}
    {'d'}
```

Input Arguments

G — Input graph
`digraph` object

Input graph, specified as a `digraph` object. Use `digraph` to create a directed graph object.

Example: `G = digraph([1 2],[2 3])`

`nodeID` — Node identifier
node index | node name

Node identifier, specified as one of the values in this table.

Value	Example
Scalar node index	`1`
Character vector node name	`'A'`
String scalar node name	`"A"`

Example: `sucIDs = successors(G,3)` finds the successor nodes of node 3.

Output Arguments

`sucIDs` — Successor node IDs
node indices | node names

Successor node IDs, returned as a node indices or node names.

- If `nodeID` is a numeric node index, then `sucIDs` contains numeric node indices.
- If `nodeID` is a node name, then `sucIDs` contains node names.

Tips

- For multigraphs with multiple edges between the same two nodes, the output of `outedges` can be longer than that of `successors`, since there can be multiple outgoing edges to some of the successors.

See Also

`digraph` | `neighbors` | `outdegree` | `outedges` | `predecessors`

Introduced in R2015b

toposort

Topological order of directed acyclic graph

Syntax

```
n = toposort(G)
n = toposort(G,'Order',algorithm)
[n,H] = toposort( ___ )
```

Description

`n = toposort(G)` returns the topological order on page 1-3463 of the nodes in `G` such that `i < j` for every edge `(n(i),n(j))` in `G`. The directed graph `G` cannot have any cycles.

`n = toposort(G,'Order',algorithm)` specifies the ordering algorithm. For example, `toposort(G,'Order','stable')` uses a stable ordering algorithm based on the lexicographical order of the nodes.

`[n,H] = toposort( ___ )` additionally returns directed graph `H` whose nodes are in the given topological order. You can use any of the input argument combinations in previous syntaxes.

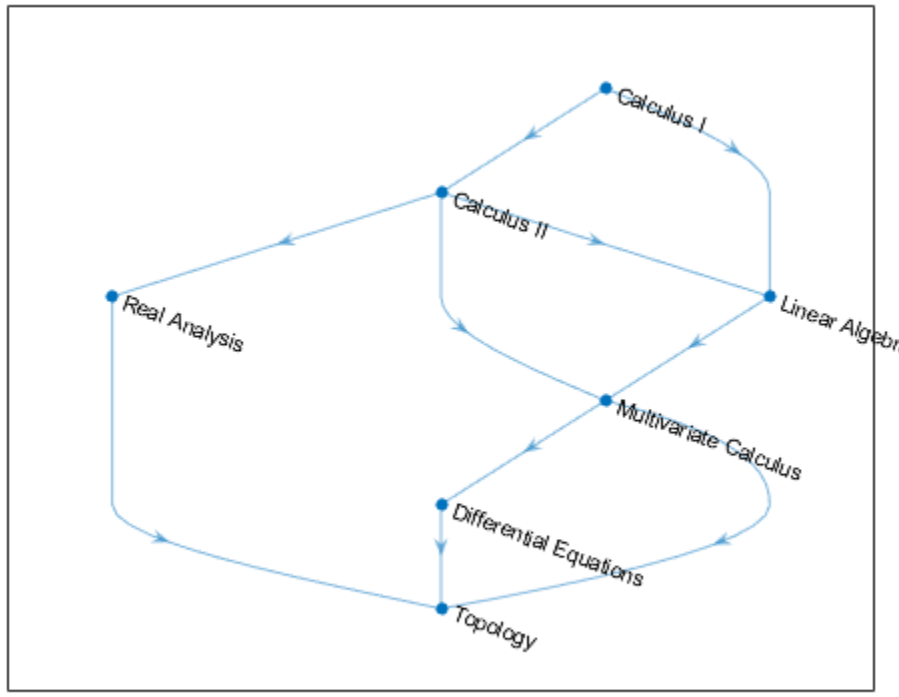
Examples

Topological Sort of Nodes

Create and plot a graph that represents a progression of university-level Mathematics courses. An edge between two courses signifies a course requirement.

```
A = [0 1 1 0 0 0 0
     0 0 0 1 0 0 0
     0 1 0 1 0 0 1
     0 0 0 0 1 1 0
```

```
     0 0 0 0 0 0 0
     0 0 0 0 1 0 0
     0 0 0 0 1 0 0];
names = {'Calculus I','Linear Algebra','Calculus II', ...
    'Multivariate Calculus','Topology', ...
    'Differential Equations','Real Analysis'};
G = digraph(A,names);
plot(G)
```



Find the topological sorting of the courses to determine the proper order in which they should be completed.

```
N = toposort(G)
```

```
N = 1×7

     1     3     7     2     4     6     5
```

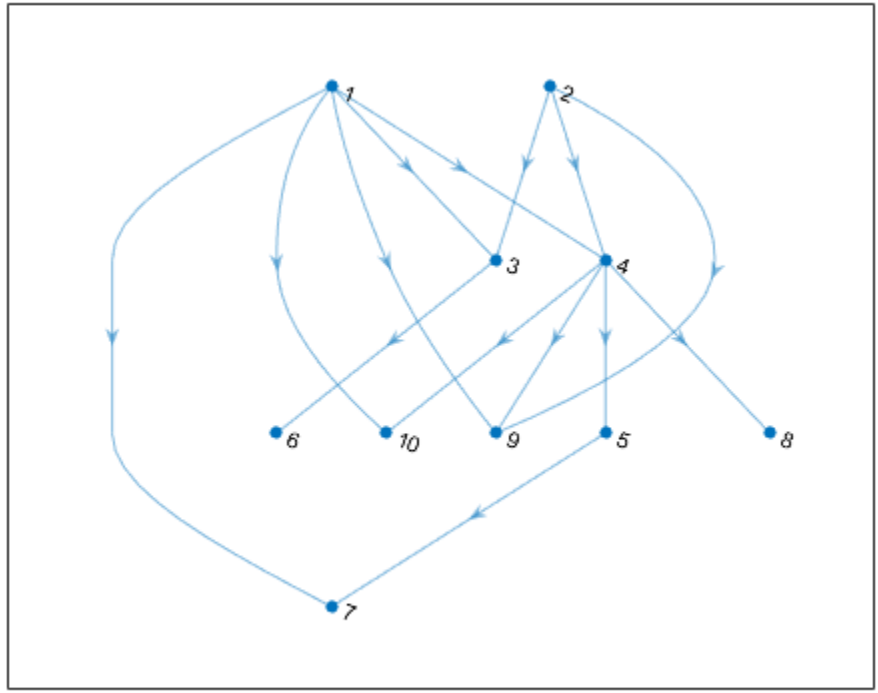
```
G.Nodes.Name(N,:)
```

```
ans = 7x1 cell array
    {'Calculus I'            }
    {'Calculus II'           }
    {'Real Analysis'         }
    {'Linear Algebra'        }
    {'Multivariate Calculus' }
    {'Differential Equations'}
    {'Topology'              }
```

Stable Topological Sort

Create a directed graph using a logical adjacency matrix, and then plot the graph.

```
rng default;
A = tril(sprand(10, 10, 0.3), -1)~=0;
G = digraph(A);
[~,G] = toposort(G);
plot(G)
```



Find the topological sorting of the graph nodes. Even though `G` is already in topological order, `(1 2 3 4 5 6 7 8 9 10)`, `toposort` reorders the nodes.

```
toposort(G)
```

```
ans = 1×10

     2     1     4    10     9     8     5     7     3     6
```

Specify `Order` as `'stable'` to use the stable ordering algorithm, so that the sort orders the nodes with smaller indices first. Stable sort does not rearrange `G` if it is already in topological order.

```
toposort(G,'Order','stable')
```

```
ans = 1×10

     1     2     3     4     5     6     7     8     9    10
```

Input Arguments

G — Input graph
digraph object

Input graph, specified as a digraph object. G must be a directed acyclic graph. Use isdag to confirm that G does not contain cycles.

Use digraph to create a directed graph.

Example: G = digraph([1 2],[2 3])

algorithm — Ordering algorithm
'fast' (default) | 'stable'

Ordering algorithm, specified as 'fast' or 'stable':

'fast' (default)	Based on a depth-first search. A node is added to the beginning of the list after considering all of its descendents. If G is already in topological order, this method might still reorder the nodes.
'stable'	Based on lexicographical order. n(1) is the node with smallest index, n(2) the next smallest given n(1), and so on. If G is in topological order then H is unchanged and n is 1:numnodes(G).

Example: [n,H] = toposort(G,'Order','stable')

Output Arguments

n — Node indices
row vector

Node indices, returned as a row vector.

H — Topologically sorted graph
`digraph` object

Topologically sorted graph, returned as a `digraph` object. `H` is the same graph as `G`, but has the nodes reordered according to `n`.

More About

Topological Order

The topological ordering of a directed graph is an ordering of the nodes in the graph such that each node appears before its successors (descendents).

Consider a directed graph whose nodes represent tasks and whose edges represent dependencies that certain tasks must be completed before others. For such a graph, the topological sorting of the graph nodes produces a valid sequence in which the tasks could be performed.

See Also

`digraph` | `isdag` | `reordernodes`

Introduced in R2015b

transclosure

Transitive closure

Syntax

```
H = transclosure(G)
```

Description

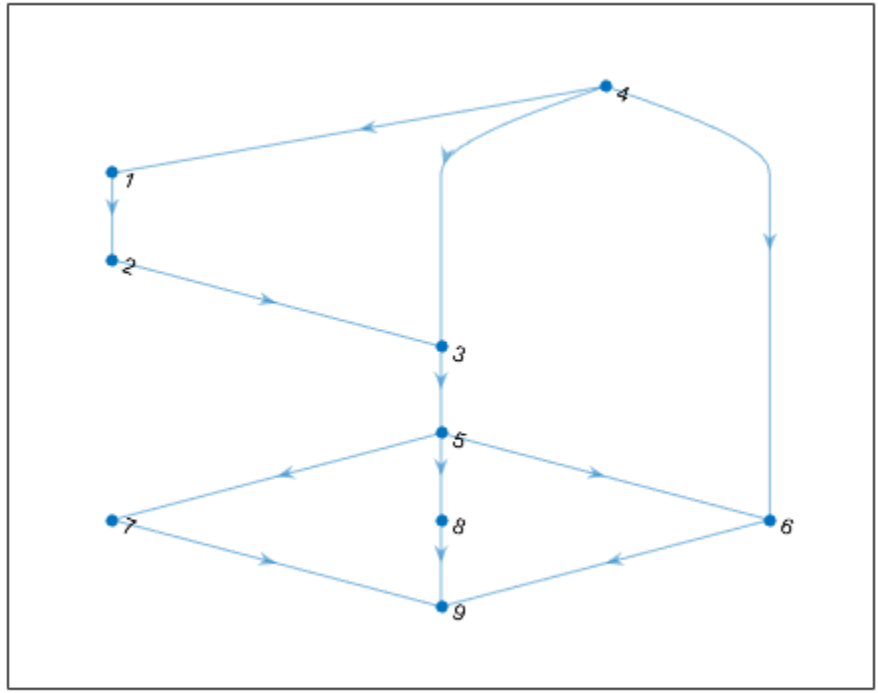
`H = transclosure(G)` returns the transitive closure on page 1-3467 of graph `G` as a new graph, `H`. The nodes in `H` are the same as those in `G`, but `H` has additional edges. If there is a path from node `i` to node `j` in `G`, then there is an edge between node `i` and node `j` in `H`. For multigraphs with multiple edges between the same two nodes, the output graph replaces these with a single edge.

Examples

Transitive Closure of Graph

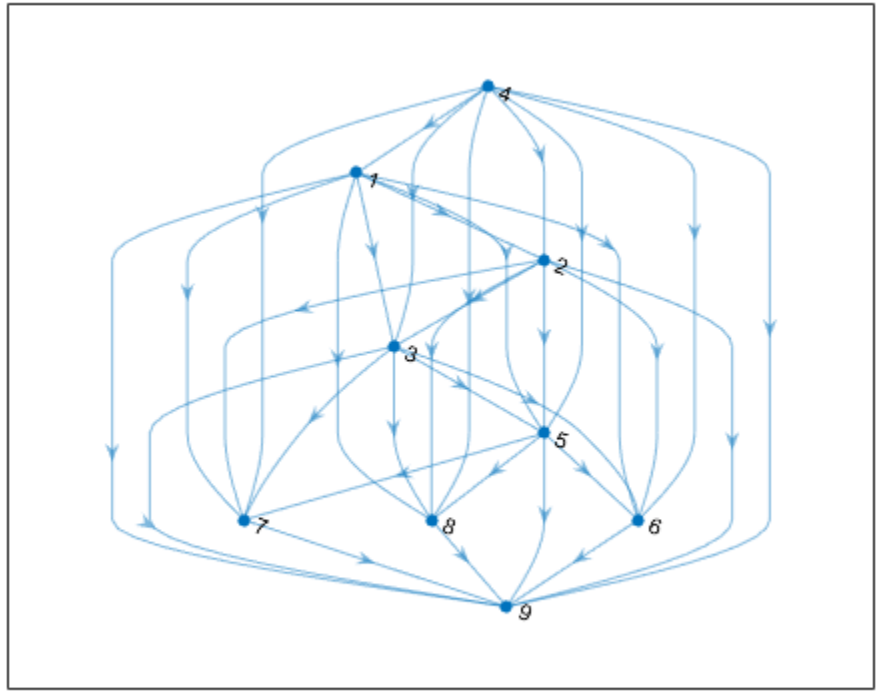
Create and plot a directed graph.

```
G = digraph([1 2 3 4 4 4 5 5 5 6 7 8],[2 3 5 1 3 6 6 7 8 9 9 9]);
plot(G)
```



Find the transitive closure of graph G and plot the resulting graph. H contains the same nodes as G, but has additional edges.

```
H = transclosure(G);
plot(H)
```



The transitive closure information in H can be used to answer reachability questions about the original graph, G.

Determine the nodes in G that can be reached from node 1. These nodes are the successors of node 1 in the transitive closure graph, H.

```
N = successors(H,1)
```

```
N = 7×1

     2
     3
     5
     6
```

```
7
8
9
```

Input Arguments

G — Input graph
`digraph` object

Input graph, specified as a `digraph` object. Use `digraph` to create a directed graph object.

Example: `G = digraph([1 2],[2 3])`

Output Arguments

H — Transitive closure of G
`digraph` object

Transitive closure of `G`, returned as a `digraph` object. The table `G.Nodes` is copied to `H`, but any properties in `G.Edges` are dropped.

Use `successors(H,n)` to determine the nodes in `G` that are reachable from node `n`.

More About

Transitive Closure

The transitive closure of a graph describes the paths between the nodes. If there is a path from node `i` to node `j` in a graph, then an edge exists between node `i` and node `j` in the transitive closure of that graph. Thus, for a given node in the graph, the transitive closure turns any reachable node into a direct successor (descendent) of that node.

See Also

`conncomp` | `digraph` | `predecessors` | `successors` | `transreduction`

Introduced in R2015b

transreduction

Transitive reduction

Syntax

```
H = transreduction(G)
```

Description

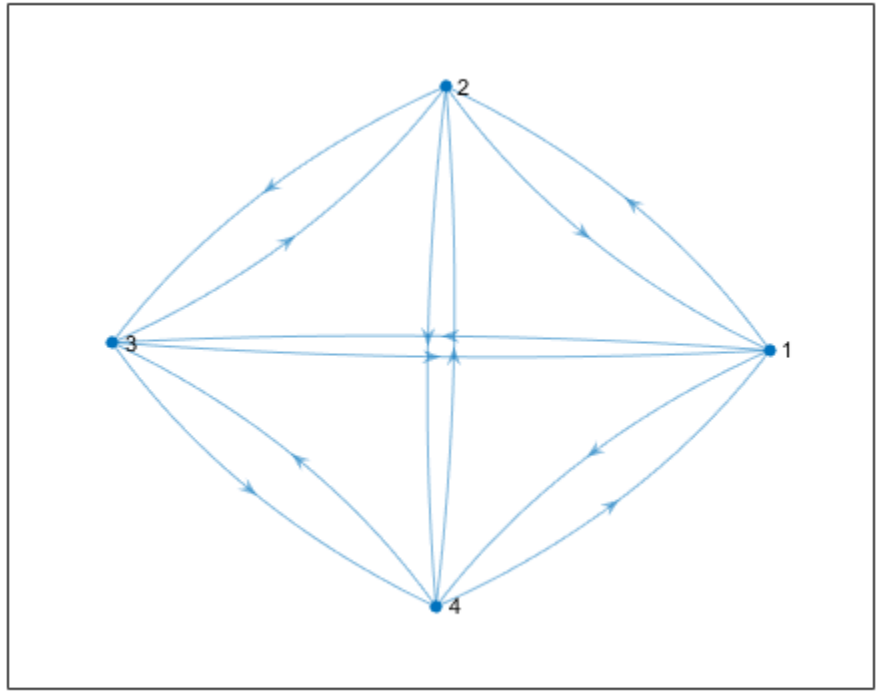
`H = transreduction(G)` returns the transitive reduction on page 1-3476 of graph `G` as a new graph, `H`. The nodes in `H` are the same as those in `G`, but `H` has different edges. `H` contains the fewest number of edges such that if there is a path from node `i` to node `j` in `G`, then there is also a path from node `i` to node `j` in `H`.

Examples

Transitive Reduction of Complete Graph

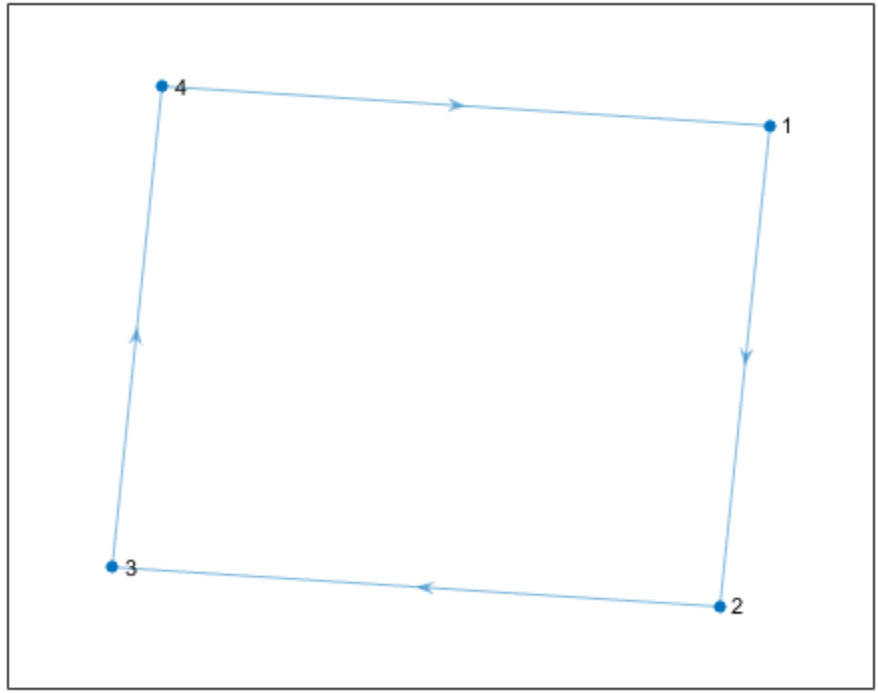
Create and plot a complete graph of order four.

```
G = digraph([1 1 1 2 2 2 3 3 3 4 4 4],[2 3 4 1 3 4 1 2 4 1 2 3]);
plot(G)
```



Find the transitive reduction and plot the resulting graph. Since the reachability in a complete graph is extensive, there are theoretically several possible transitive reductions, as any cycle through the four nodes is a candidate.

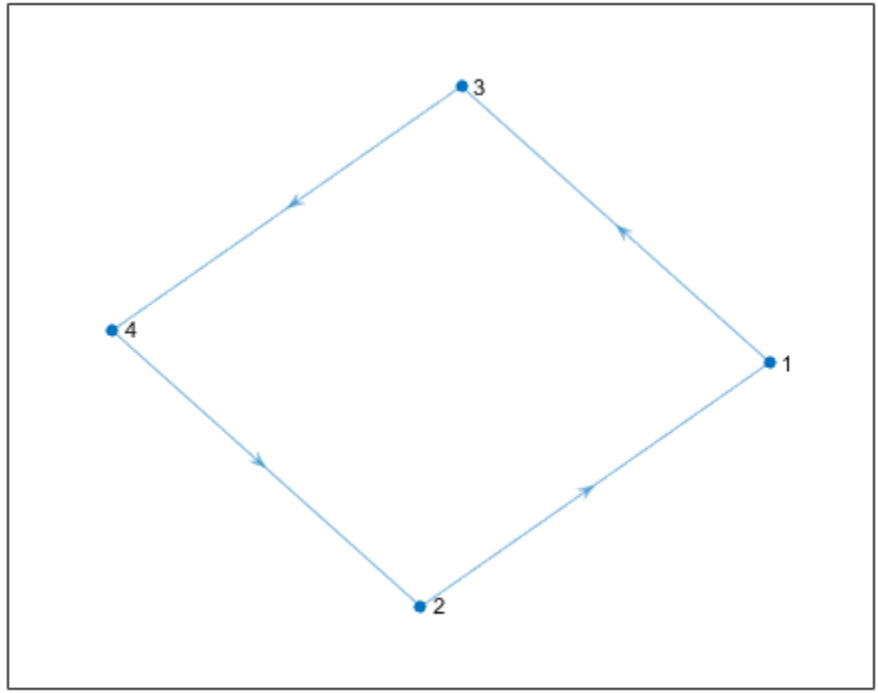
```
H = transreduction(G);
plot(H)
```



Two graphs with the same reachability also have the same transitive reduction. Therefore, any cycle of four nodes produces the same transitive reduction as H.

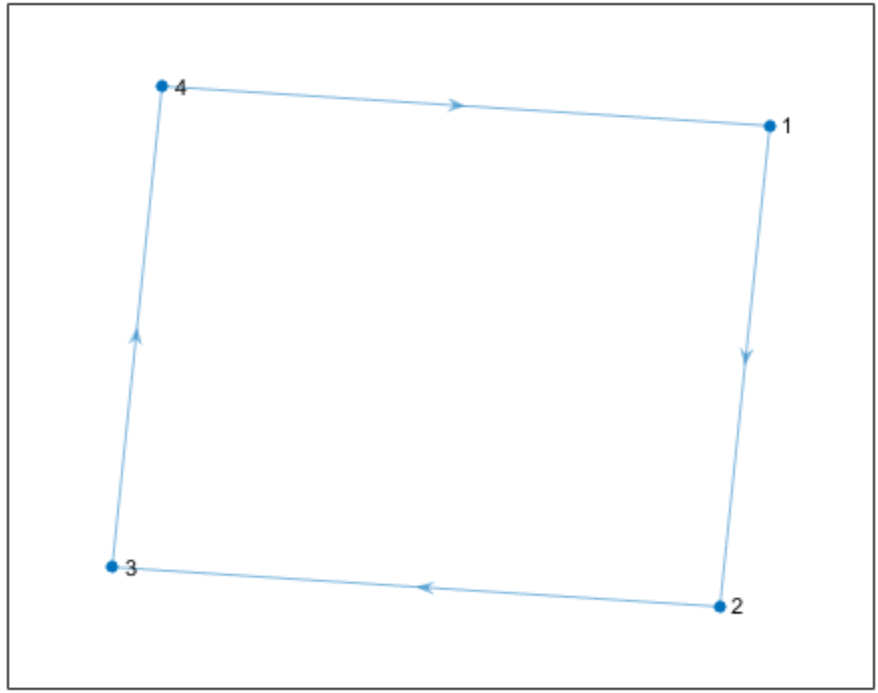
Create a directed graph that contains a different four node cycle: (1,3,4,2,1).

```
G1 = digraph([1 3 4 2],[3 4 2 1]);
plot(G1)
```



Find the transitive reduction of `G1`. The cycle in `G1` is reordered so that the transitive reductions `H` and `H1` have the same cycle, (1,2,3,4,1).

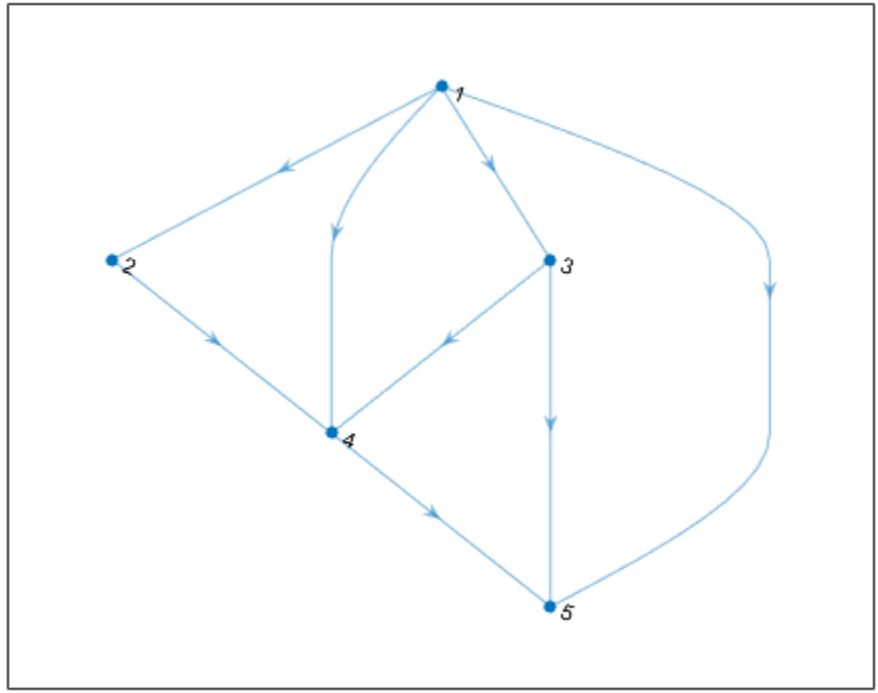
```
H1 = transreduction(G1);
plot(H1)
```



Unique Transitive Reduction

Create and plot a directed acyclic graph.

```
s = [1 1 1 1 2 3 3 4];
t = [2 3 4 5 4 4 5 5];
G = digraph(s,t);
plot(G)
```



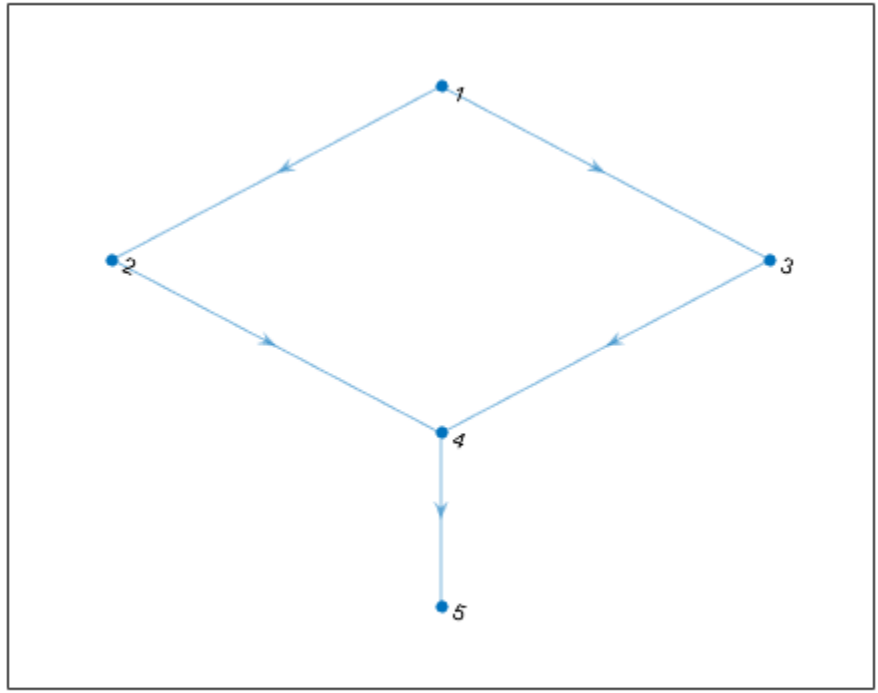
Confirm that G does not contain any cycles.

```
tf = isdag(G)
```

```
tf = logical
   1
```

Find the transitive reduction of the graph. Since the graph does not contain cycles, the transitive reduction is unique and is a subgraph of G.

```
H = transreduction(G);
plot(H)
```



Input Arguments

G — Input graph
`digraph` object

Input graph, specified as a `digraph` object. Use `digraph` to create a directed graph object.

Example: `G = digraph([1 2],[2 3])`

Output Arguments

H — Transitive reduction of G
digraph object

Transitive reduction of G, returned as a digraph object. The table G.Nodes is copied to H, but any properties in G.Edges are dropped. H might contain new edges not present in G.

H contains the fewest number of edges that still preserve the reachability of graph G. In other words, transclosure(H) is the same as transclosure(G).

If isdag(G) is true, then H is unique and is a subgraph of G.

More About

Transitive Reduction

The transitive reduction of graph G is the graph with the fewest edges that still shares the same reachability as G. Therefore, of all the graphs that have the same transitive closure as G, the transitive reduction is the one with the fewest edges. If two directed graphs have the same transitive closure, they also have the same transitive reduction.

See Also

conncomp | digraph | transclosure

Introduced in R2015b

digraph

Graph with directed edges

Description

`digraph` objects represent directed graphs, which have directional edges connecting the nodes. After you create a `digraph` object, you can learn more about the graph by using the object functions to perform queries against the object. For example, you can add or remove nodes or edges, determine the shortest path between two nodes, or locate a specific node or edge.

```
G = digraph([1 1], [2 3])
e = G.Edges
G = addedge(G,2,3)
G = addnode(G,4)
plot(G)
```

Creation

Syntax

```
G = digraph
```

```
G = digraph(A)
G = digraph(A,nodenames)
G = digraph(A,NodeTable)
G = digraph(A,___,'omitselfloops')
```

```
G = digraph(s,t)
G = digraph(s,t,weights)
G = digraph(s,t,weights,nodenames)
G = digraph(s,t,weights,NodeTable)
G = digraph(s,t,weights,num)
G = digraph(s,t,___,'omitselfloops')
G = digraph(s,t,EdgeTable,___)
```

```
G = digraph(EdgeTable)
G = digraph(EdgeTable,NodeTable)
G = digraph(EdgeTable, ___ ,'omitselfloops')
```

Description

`G = digraph` creates an empty directed graph object, `G`, which has no nodes or edges.

`G = digraph(A)` creates a weighted directed graph using a square adjacency matrix, `A`. The location of each nonzero entry in `A` specifies an edge for the graph, and the weight of the edge is equal to the value of the entry. For example, if `A(2,1) = 10`, then `G` contains an edge from node 2 to node 1 with a weight of 10.

`G = digraph(A,nodenames)` additionally specifies node names. The number of elements in `nodenames` must be equal to `size(A,1)`.

`G = digraph(A,NodeTable)` specifies node names (and possibly other node attributes) using a table, `NodeTable`. The table must have the same number of rows as `A`. Specify node names using the table variable `Name`.

`G = digraph(A, ___ ,'omitselfloops')` ignores the diagonal elements of `A` and returns a graph without any self-loops. You can use any of the input argument combinations in previous syntaxes.

`G = digraph(s,t)` specifies directed graph edges `(s,t)` in pairs to represent the source and target nodes. `s` and `t` can specify node indices or node names.

`G = digraph(s,t,weights)` also specifies edge weights with the array `weights`.

`G = digraph(s,t,weights,nodenames)` additionally specifies node names using the cell array of character vectors or string array, `nodenames`. `s` and `t` cannot contain node names that are not in `nodenames`.

`G = digraph(s,t,weights,NodeTable)` specifies node names (and possibly other node attributes) using a table, `NodeTable`. Specify node names using the `Name` table variable. `s` and `t` cannot contain node names that are not in `NodeTable`.

`G = digraph(s,t,weights,num)` specifies the number of nodes in the graph with the numeric scalar `num`.

`G = digraph(s,t, ___ ,'omitselfloops')` does not add any self-loops to the graph. That is, any `k` that satisfies `s(k) == t(k)` is ignored. You can use any of the input argument combinations in previous syntaxes.

`G = digraph(s,t,EdgeTable, ___ )` uses a table to specify edge attributes instead of specifying `weights`. The `EdgeTable` input must be a table with a row for each corresponding pair of elements in `s` and `t`. Specify edge weights using the table variable `Weight`.

`G = digraph(EdgeTable)` uses the table `EdgeTable` to define the graph. With this syntax, the first variable in `EdgeTable` must be named `EndNodes`, and it must be a two-column array defining the edge list of the graph.

`G = digraph(EdgeTable,NodeTable)` additionally specifies the names (and possibly other attributes) of the graph nodes using a table, `NodeTable`.

`G = digraph(EdgeTable, ___ ,'omitselfloops')` does not add self-loops to the graph. That is, any `k` that satisfies `EdgeTable.EndNodes(k,1) == EdgeTable.EndNodes(k,2)` is ignored. You must specify `EdgeTable` and optionally can specify `NodeTable`.

Input Arguments

`A` — Adjacency matrix

matrix

Adjacency matrix, specified as a full or sparse, numeric matrix. The entries in `A` specify the network of connections (edges) between the nodes of the graph. The location of each nonzero entry in `A` specifies an edge between two nodes. The value of that entry provides the edge weight. A logical adjacency matrix results in an unweighted graph.

Nonzero entries on the main diagonal of `A` specify *self-loops*, or nodes that are connected to themselves with an edge. Use the `'omitselfloops'` input option to ignore diagonal entries.

Example: `A = [0 1 0; 0 0 0; 5 0 0]` describes a graph with three nodes and two edges. The edge from node 1 to node 2 has a weight of `1`, and the edge from node 3 to node 1 has a weight of `5`.

Data Types: `single` | `double` | `logical`

nodenames — Node names
cell array of character vectors | string array

Node names, specified as a cell array of character vectors or string array. `nodenames` must have length equal to `numnodes(G)` so that it contains a nonempty, unique name for each node in the graph.

Example: `G = digraph(A,{'n1','n2','n3'})` specifies three node names for a 3-by-3 adjacency matrix, `A`.

Data Types: `cell` | `string`

s,t — Source and target node pairs (as separate arguments)
node indices | node names

Source and target node pairs, specified as node indices or node names. `digraph` creates directed edges between the corresponding nodes in `s` and `t`, which must both be numeric, or both be character vectors, cell arrays of character vectors, categorical arrays, or string arrays. In all cases, `s` and `t` must have the same number of elements.

- If `s` and `t` are numeric, then they correspond to indices of graph nodes. Numeric node indices must be positive integers greater than or equal to 1.
- If `s` and `t` are character vectors, cell arrays of character vectors, or string arrays, then they specify names for the nodes. The `Nodes` property of the graph is a table containing a `Name` variable with the node names, `G.Nodes.Name`.
- If `s` and `t` are categorical arrays, then the categories in `s` and `t` are used as the node names in the graph. This can include categories that are not elements in `s` or `t`.
- If `s` and `t` specify multiple edges between the same two nodes, then the result is a multigraph.

This table shows the different ways to refer to one or more nodes either by their numeric node indices or by their node names.

Form	Single Node	Multiple Nodes
Node index	Scalar **Example:** 1	Vector **Example:** `[1 2 3]`

Form	Single Node	Multiple Nodes
Node name	Character vector **Example:** `'A'`	Cell array of character vectors **Example:** `{'A' 'B' 'C'}`
	String scalar **Example:** `"A"`	String array **Example:** `["A" "B" "C"]`
	Categorical array **Example:** `categorical("A")`	Categorical array **Example:** `categorical(["A" "B" "C"])`

Example: `G = digraph([1 2 3],[2 4 5])` creates a graph with five nodes and three edges.

Example: `G = digraph({'Boston' 'New York' 'Washington D.C.'},{'New York' 'New Jersey' 'Pittsburgh'})` creates a graph with five named nodes and three edges.

`weights` — Edge weights
scalar | vector | matrix | multidimensional array | `[]`

Edge weights, specified as a scalar, vector, matrix, or multidimensional array. `weights` must be a scalar or an array with the same number of elements as `s` and `t`.

`digraph` stores the edge weights as a `Weight` variable in the `G.Edges` property table. To add or change weights after creating a graph, you can modify the table variable directly, for example, `G.Edges.Weight = [25 50 75]'`.

If you specify `weights` as an empty array `[]`, then it is ignored.

Example: `G = digraph([1 2],[2 3],[100 200])` creates a graph with three nodes and two edges. The edges have weights of `100` and `200`.

Data Types: `single` | `double`

`num` — Number of graph nodes
positive scalar integer

Number of graph nodes, specified as a positive scalar integer. `num` must be greater than or equal to the largest elements in `s` and `t`.

Example: `G = digraph([1 2],[2 3],[],5)` creates a graph with three connected nodes and two isolated nodes.

`EdgeTable` — Table of edge information
table

Table of edge information. If you do not specify `s` and `t`, then the first variable in `EdgeTable` is required to be a two-column matrix, cell array of character vectors, or string array called `EndNodes` that defines the graph edges. For edge weights, use the variable `Weight`, since this table variable name is used by some graph functions. If there is a variable `Weight`, then it must be a numeric column vector. See `table` for more information on constructing a table.

After creating a graph, query the edge information table using `G.Edges`.

Example: `EdgeTable = table([1 2; 2 3; 3 5; 4 5],'VariableNames', {'EndNodes'})`

Data Types: `table`

`NodeTable` — Table of node information
table

Table of node information. `NodeTable` can contain any number of variables to describe attributes of the graph nodes. For node names, use the variable `Name`, since this variable name is used by some graph functions. If there is a variable `Name`, then it must be a cell array of character vectors or string array specifying a unique name in each row. See `table` for more information on constructing a table.

After the graph is created, query the node information table using `G.Nodes`.

Example: `NodeTable = table({'a'; 'b'; 'c'; 'd'},'VariableNames', {'Name'})`

Data Types: `table`

Output Arguments

`G` — Directed graph
`digraph` object

Directed graph, returned as a `digraph` object.

Properties

Edges — Edges of graph
table

Edges of graph, returned as a table. By default this is an `M`-by-`1` table, where `M` is the number of edges in the graph.

- To add new edge properties to the graph, create a new variable in the `Edges` table.
- To add or remove edges from the graph, use the `addedge` or `rmedge` object functions.

Example: `G.Edges` returns a table listing the edges in the graph

Example: `G.Edges.Weight` returns a numeric vector of the edge weights.

Example: `G.Edges.Weight = [10 20 30 55]'` specifies new edge weights for the graph.

Example: `G.Edges.NormWeight = G.Edges.Weight/sum(G.Edges.Weight)` adds a new edge property to the table containing the normalized weights of the edges.

Data Types: `table`

Nodes — Nodes of graph
table

Nodes of graph, returned as a table. By default this is an empty `N`-by-`0` table, where `N` is the number of nodes in the graph.

- To add new node properties to the graph, create a new variable in the `Nodes` table.
- To add or remove nodes from the graph, use the `addnode` or `rmnode` object functions.

Example: `G.Nodes` returns a table listing the node properties of the graph. This table is empty by default.

Example: `G.Nodes.Names = {'Montana', 'New York', 'Washington', 'California'}'` adds node names to the graph by adding the variable `Names` to the `Nodes` table.

Example: `G.Nodes.WiFi = logical([1 0 0 1 1]')` adds the variable `WiFi` to the `Nodes` table. This property specifies that certain airports have wireless internet coverage.

Data Types: `table`

Object Functions

Access and Modify Nodes and Edges

addedge	Add new edge to graph
rmedge	Remove edge from graph
flipedge	Reverse edge directions
addnode	Add new node to graph
rmnode	Remove node from graph
findedge	Locate edge in graph
findnode	Locate node in graph
numedges	Number of edges in graph
numnodes	Number of nodes in graph
edgecount	Number of edges between two nodes
reordernodes	Reorder graph nodes
subgraph	Extract subgraph

Search and Structure

bfsearch	Breadth-first graph search
dfsearch	Depth-first graph search
centrality	Measure node importance
toposort	Topological order of directed acyclic graph
transclosure	Transitive closure
transreduction	Transitive reduction
isdag	Determine if graph is acyclic
conncomp	Connected graph components
condensation	Graph condensation
maxflow	Maximum flow in graph
isisomorphic	Determine whether two graphs are isomorphic
isomorphism	Compute isomorphism between two graphs
ismultigraph	Determine whether graph has multiple edges
simplify	Reduce multigraph to simple graph

Shortest Path

shortestpath	Shortest path between two single nodes

shortestpathtree	Shortest path tree from node
distances	Shortest path distances of all node pairs

Matrix Representation

adjacency	Graph adjacency matrix
incidence	Graph incidence matrix

Node Information

indegree	In-degree of nodes
outdegree	Out-degree of nodes
predecessors	Node predecessors
successors	Node successors
nearest	Nearest neighbors within radius
inedges	Incoming edges to node
outedges	Outgoing edges from node

Visualization

plot	Plot graph nodes and edges

Examples

Create and Modify Digraph Object

Create a `digraph` object with three nodes and three edges. One edge is from node 1 to node 2, another is from node 1 to node 3, and the third is from node 2 to node 1.

```
G = digraph([1 1 2],[2 3 1])
```

```
G = 
  digraph with properties:

    Edges: [3x1 table]
    Nodes: [3x0 table]
```

View the edge table of the graph. For directed graphs, the first column indicates the source nodes of each edge, and the second column indicates the target nodes.

```
G.Edges
```

```
ans=3×1 table
    EndNodes
    ________

     1    2
     1    3
     2    1
```

Add node names to the graph, then view the new node and edge tables. The source and target nodes of each edge are now expressed using their node names.

```
G.Nodes.Name = {'A' 'B' 'C'}';
G.Nodes
```

```
ans=3×1 table
    Name
    _____

    {'A'}
    {'B'}
    {'C'}
```

```
G.Edges
```

```
ans=3×1 table
       EndNodes
    _______________

    {'A'}    {'B'}
    {'A'}    {'C'}
    {'B'}    {'A'}
```

You can add or modify extra variables in the `Nodes` and `Edges` tables to describe attributes of the graph nodes or edges. However, you cannot directly change the number of nodes or edges in the graph by modifying these tables. Instead, use the `addedge`, `rmedge`, `addnode`, or `rmnode` functions to modify the number of nodes or edges in a graph.

For example, add an edge to the graph between nodes 2 and 3 and view the new edge list.

```
G = addedge(G,2,3)
```

```
G = 
  digraph with properties:

    Edges: [4x1 table]
    Nodes: [3x1 table]
```

```
G.Edges
```

```
ans=4×1 table
       EndNodes
    _____________

    {'A'}    {'B'}
    {'A'}    {'C'}
    {'B'}    {'A'}
    {'B'}    {'C'}
```

Adjacency Matrix Graph Construction

Create a symmetric adjacency matrix, A, that creates a complete directed graph of order 4. Use a logical adjacency matrix to create a graph without weights.

```
A = ones(4) - diag([1 1 1 1])
```

```
A = 4×4

     0     1     1     1
     1     0     1     1
     1     1     0     1
     1     1     1     0
```

```
G = digraph(A~=0)
```

```
G = 
  digraph with properties:

    Edges: [12x1 table]
```

```
    Nodes: [4x0 table]
```

View the edge list of the graph.

```
G.Edges
```

```
ans=12×1 table
    EndNodes
    ________

     1    2
     1    3
     1    4
     2    1
     2    3
     2    4
     3    1
     3    2
     3    4
     4    1
     4    2
     4    3
```

Adjacency Matrix Construction with Node Names

Create an adjacency matrix.

```
A = magic(4);
A(A>10) = 0
```

```
A = 4×4

     0     2     3     0
     5     0    10     8
     9     7     6     0
     4     0     0     1
```

Create a graph with named nodes using the adjacency matrix. Specify `'omitselfloops'` to ignore the entries on the diagonal of `A`.

```
names = {'alpha' 'beta' 'gamma' 'delta'};
G = digraph(A,names,'omitselfloops')
```

```
G = 
  digraph with properties:

    Edges: [8x2 table]
    Nodes: [4x1 table]
```

View the edge and node information.

```
G.Edges
```

```
ans=8×2 table
           EndNodes            Weight
    ______________________     ______

    {'alpha'}    {'beta' }        2
    {'alpha'}    {'gamma'}        3
    {'beta' }    {'alpha'}        5
    {'beta' }    {'gamma'}       10
    {'beta' }    {'delta'}        8
    {'gamma'}    {'alpha'}        9
    {'gamma'}    {'beta' }        7
    {'delta'}    {'alpha'}        4
```

```
G.Nodes
```

```
ans=4×1 table
      Name
    _________

    {'alpha'}
    {'beta' }
    {'gamma'}
    {'delta'}
```

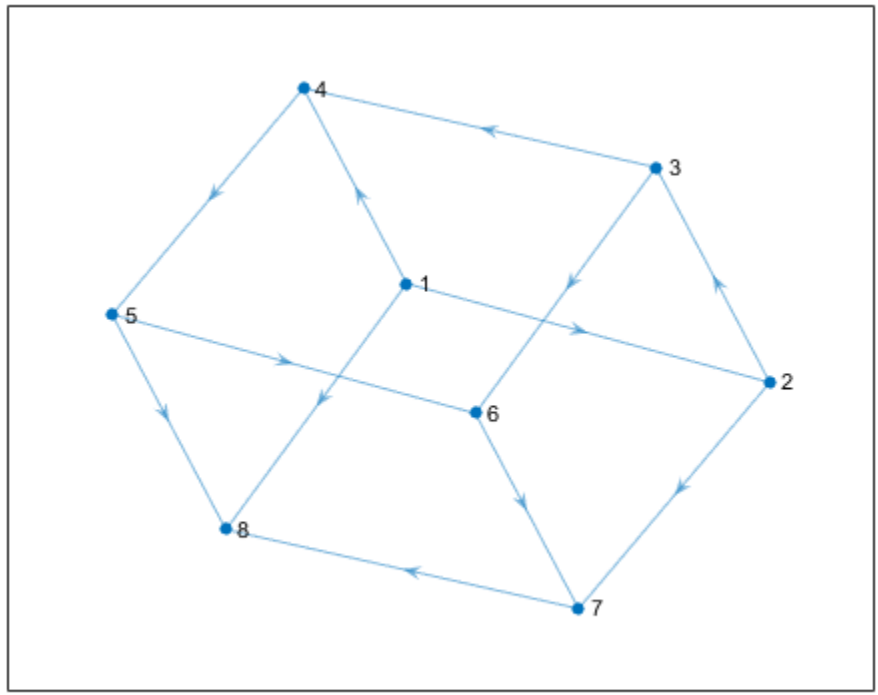
Edge List Graph Construction

Create and plot a cube graph using a list of the end nodes of each edge.

```
s = [1 1 1 2 2 3 3 4 5 5 6 7];
t = [2 4 8 3 7 4 6 5 6 8 7 8];
G = digraph(s,t)
```

```
G 
  digraph with properties:

    Edges: [12x1 table]
    Nodes: [8x0 table]
```

```
plot(G,'Layout','force')
```



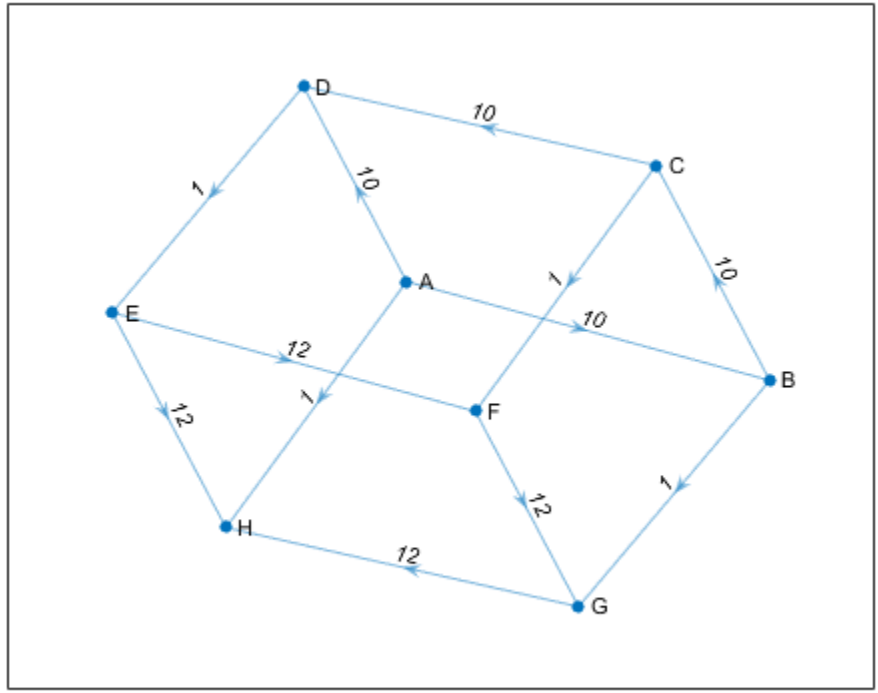
Edge List Graph Construction with Node Names and Edge Weights

Create and plot a cube graph using a list of the end nodes of each edge. Specify node names and edge weights as separate inputs.

```
s = [1 1 1 2 2 3 3 4 5 5 6 7];
t = [2 4 8 3 7 4 6 5 6 8 7 8];
weights = [10 10 1 10 1 10 1 1 12 12 12 12];
names = {'A' 'B' 'C' 'D' 'E' 'F' 'G' 'H'};
G = digraph(s,t,weights,names)
```

```
G = 
  digraph with properties:

    Edges: [12x2 table]
    Nodes: [8x1 table]
```

```
plot(G,'Layout','force','EdgeLabel',G.Edges.Weight)
```



Edge List Construction with Extra Nodes

Create a weighted graph using a list of the end nodes of each edge. Specify that the graph should contain a total of 10 nodes.

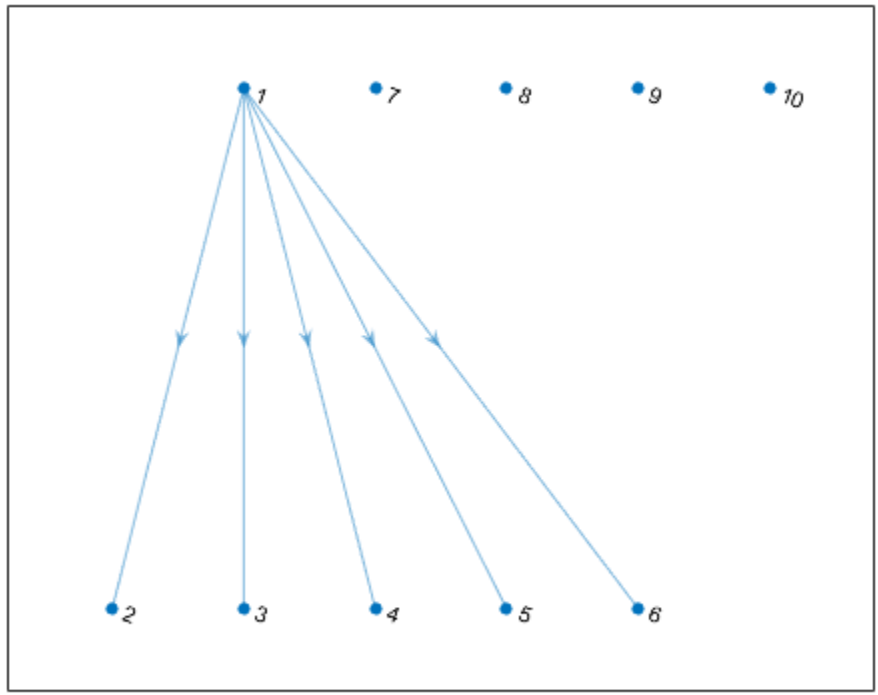
```
s = [1 1 1 1 1];
t = [2 3 4 5 6];
weights = [5 5 5 6 9];
G = digraph(s,t,weights,10)
```

```
G = 
  digraph with properties:
```

```
    Edges: [5x2 table]
    Nodes: [10x0 table]
```

Plot the graph. The extra nodes are disconnected from the primary connected component.

```
plot(G)
```



Add Nodes and Edges to Empty Graph

Create an empty digraph object, G.

```
G = digraph;
```

Add three nodes and three edges to the graph. The corresponding entries in `s` and `t` define the source and target nodes of the edges. `addedge` automatically adds the appropriate nodes to the graph if they are not already present.

```
s = [1 2 1];
t = [2 3 3];
G = addedge(G,s,t)
```

```
G = 
  digraph with properties:

    Edges: [3x1 table]
    Nodes: [3x0 table]
```

View the edge list. Each row describes an edge in the graph.

```
G.Edges
```

```
ans=3×1 table
    EndNodes
    ________

     1    2
     1    3
     2    3
```

For the best performance, construct graphs all at once using a single call to `digraph`. Adding nodes or edges in a loop can be slow for large graphs.

Graph Construction with Tables

Create an edge table that contains the variables `EndNodes`, `Weight`, and `Code`. Then create a node table that contains the variables `Name` and `Country`. The variables in each table specify properties of the graph nodes and edges.

```
s = [1 1 1 2 2 3];
t = [2 3 4 3 4 4];
weights = [6 6.5 7 11.5 12 17]';
```

```
code = {'1/44' '1/49' '1/33' '44/49' '44/33' '49/33'}';
EdgeTable = table([s' t'],weights,code, ...
    'VariableNames',{'EndNodes' 'Weight' 'Code'})
```

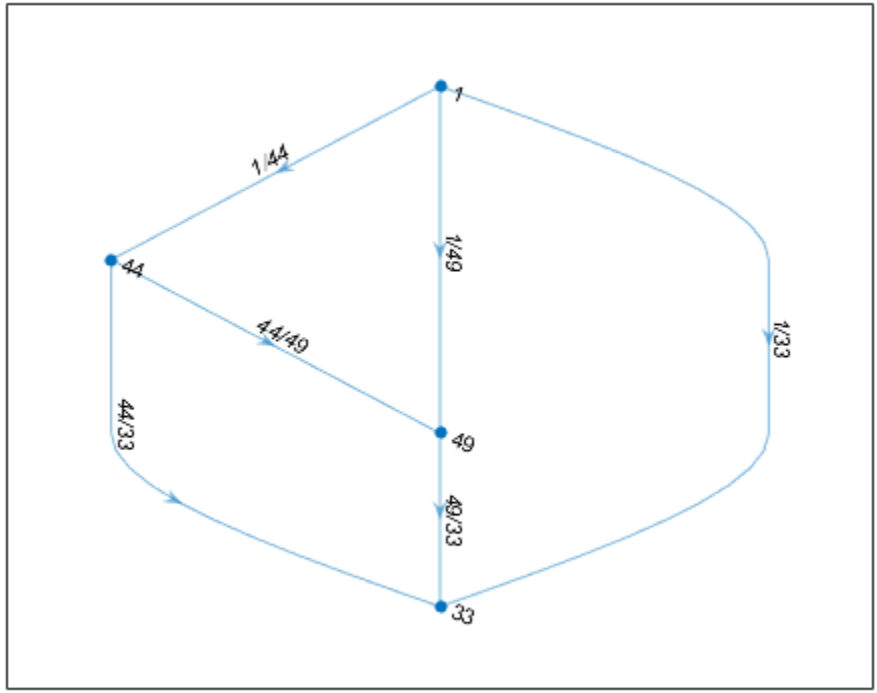
```
EdgeTable=6×3 table
    EndNodes    Weight      Code
    ________    ______    _________

     1    2         6     {'1/44' }
     1    3       6.5     {'1/49' }
     1    4         7     {'1/33' }
     2    3      11.5     {'44/49'}
     2    4        12     {'44/33'}
     3    4        17     {'49/33'}
```

```
names = {'USA' 'GBR' 'DEU' 'FRA'}';
country_code = {'1' '44' '49' '33'}';
NodeTable = table(names,country_code,'VariableNames',{'Name' 'Country'})
```

```
NodeTable=4×2 table
     Name      Country
    _______    _______

    {'USA'}    {'1' }
    {'GBR'}    {'44'}
    {'DEU'}    {'49'}
    {'FRA'}    {'33'}
```

Create a graph using the node and edge tables. Plot the graph using the country codes as node and edge labels.

```
G = digraph(EdgeTable,NodeTable);
plot(G,'NodeLabel',G.Nodes.Country,'EdgeLabel',G.Edges.Code)
```



Compatibility Considerations

Change in handling of duplicate edges

Behavior changed in R2018a

`graph`, `digraph`, and `addedge` no longer produce errors when they encounter duplicate edges. Instead, the duplicate edges are added to the graph and the result is a multigraph. The `ismultigraph` function is useful to detect this situation, and `simplify` provides an easy way to remove the extra edges.

See Also

`graph` | `subgraph`

Topics

"Build Watts-Strogatz Small World Graph Model"
"Use PageRank Algorithm to Rank Websites"
"Directed and Undirected Graphs"
"Modify Nodes and Edges of Existing Graph"
"Add Graph Node Names, Edge Weights, and Other Attributes"
"Graph Plotting and Customization"

Introduced in R2015b

dir

List folder contents

Syntax

```
dir
dir name
listing = dir(name)
```

Description

`dir` lists files and folders in the current folder.

`dir name` lists files and folders that match `name`. When `name` is a folder, `dir` lists the contents of the folder. Specify `name` using absolute or relative path names. The `name` argument can include the * wildcard in the file name, and both the * and the ** wildcard in the path name. Characters next to a ** wildcard must be file separators.

`listing = dir(name)` returns attributes about `name`.

Examples

View Contents of Folder

List the contents of a folder.

Create a folder, `myfolder`, that contains the files `myfile1.m`, `myfile2.m`, and `myfile3.m`.

```
mkdir myfolder
movefile myfile1.m myfolder
movefile myfile2.m myfolder
movefile myfile3.m myfolder
```

List the files in `myfolder`.

```
dir myfolder
```

```
.           ..          myfile1.m  myfile2.m  myfile3.m
```

Find Files Matching Specified Name

List all files with a `.m` extension that contain the term `my`.

Create a folder, `myfolder`, that contains the files `myfile1.m`, `myfile2.m`, and `myfile3.txt`.

```
mkdir myfolder
movefile myfile1.m myfolder
movefile myfile2.m myfolder
movefile myfile3.txt myfolder
```

List the matching files in `myfolder`.

```
cd myfolder
dir *my*.m
```

```
myfile1.m  myfile2.m
```

Find Files in Subfolders

List all files in the current folder and all of the subfolders of the current folder.

Create a folder, `myfolder1`, that contains these files and folders:

```
myfile1.m
myfolder2
     myfile2.m
     myfolder3.m
         myfile3.m
```

```
mkdir myfolder1
mkdir myfolder1/myfolder2
mkdir myfolder1/myfolder2/myfolder3

movefile myfile1.m myfolder1
```

```
movefile myfile2.m myfolder1/myfolder2
movefile myfile3.m myfolder1/myfolder2/myfolder3
```

List all files with a `.m` extension in `myfolder1` and all of the subfolders of `myfolder1`.

```
cd myfolder1
dir **/*.m
```

```
Files Found in Current Folder:

myfile1.m

Files Found in: myfolder2

myfile2.m

Files Found in: myfolder2\myfolder3

myfile3.m
```

Find Information in the Return Structure

Return the folder listing of `myfolder` to the variable `MyFolderInfo`.

Create a folder, `myfolder`, that contains the files `myfile1.m`, `myfile2.m`, and `myfile3.m`.

```
mkdir myfolder
movefile myfile1.m myfolder
movefile myfile2.m myfolder
movefile myfile3.m myfolder
```

Get a list of the files in `myfolder`. MATLAB returns the information in a structure array.

```
MyFolderInfo = dir('myfolder')
```

```
MyFolderInfo=5×6 struct
    name
    folder
    date
    bytes
    isdir
```

```
    datenum
```

Index into the structure to access a particular item.

```
MyFolderInfo(3).name
```

```
ans =
'myfile1.m'
```

Find Date File Last Modified

Get the serial date number for the date and time a file was last modified.

Use the `datenum` field of the structure returned by the `dir` command. Do not use the `datenum` function to convert the `date` field of the structure to a number. The results of the `datenum` function vary depending on the locale. Instead, use the `datenum` field.

```
MyFileInfo = dir('myfile1.m');
FileDate = MyFileInfo.datenum
```

```
FileDate = 7.3647e+05
```

Input Arguments

`name` — File or folder name
character vector | string scalar

File or folder name, specified as a character vector or string scalar. If `name` is a string, enclose it in parentheses. For example, `dir("FolderName")`.

To list files and folders at a remote location, `name` must contain a full path specified as a uniform resource locator (URL). For more information, see “Work with Remote Data”.

To search for multiple files, use wildcards in the file name. For example, `dir *.txt` lists all files with a `txt` extension in the current folder. To search through folders and subfolders on the path recursively, use wildcards in the path name. For example, `dir */*.txt` lists all files with a `txt` extension exactly one folder under the current folder, and

`dir **/*.txt` lists all files with a `txt` extension zero or more folders under the current folder. Characters next to a ** wildcard must be file separators.

Note MATLAB always treats the * character as a wildcard, even on file systems that support * in file names.

The MATLAB `dir` function is consistent with the Microsoft Windows operating system `dir` command in that both support short file names generated by DOS.

Output Arguments

`listing` — File attributes
structure array

File attributes, returned as a `n`-by-1 structure array, where `n` is the number of files and folders returned by the `dir` command.

This table shows the fields in the structure.

Field Name	Description	Class
`name`	File or folder name	`char`
`folder`	Location of file or folder	`char`
`date`	Modification date timestamp	`char`
`bytes`	Size of the file in bytes	`double`
`isdir`	`1` if name is a folder; `0` if name is a file	`logical`
`datenum`	Modification date as serial date number.	`double`

Tips

- To exclude invalid entries returned by the `dir` command, use the `cellfun` function.

```
MyFolderInfo = dir;
MyFolderInfo = MyFolderInfo(~cellfun('isempty', {MyFolderInfo.date}));
```

Invalid entries occur when you run `dir` with an output argument and the results include a nonexistent file or a file that `dir` cannot query for some other reason. In this case, `dir` returns the following default values.

```
date: ''
bytes: []
isdir: 0
datenum: []
```

Invalid entries most commonly occur on UNIX platforms when `dir` queries a symbolic link pointing to a nonexistent target. A nonexistent target is a target that is moved, removed, or renamed.

- To obtain a list of available drives on Microsoft Windows platforms, use the DOS `net use` command at the command line.

  ```
  dos('net use')
  ```

  Or type

  ```
  [s,r] = dos('net use')
  ```

  MATLAB returns the results to the character array `r`.

See Also

`cd` | `fileattrib` | `isfolder` | `ls` | `mkdir` | `rmdir` | `what`

Topics

"Specify File Names"
"Work with Remote Data"

Introduced before R2006a

dir

List folder contents on FTP server

Syntax

```
dir(ftpobj)
dir(ftpobj,folder)
```

```
listing = dir( ___ )
```

Description

`dir(ftpobj)` lists the contents of the current folder on the FTP server associated with `ftpobj`. The contents of the current folder can be files and other folders.

`dir(ftpobj,folder)` lists the contents of the specified folder.

`listing = dir( ___ )` returns a structure array that contains the name, modification date, and size of each item. You can use this syntax with the input arguments of either of the previous syntaxes.

Examples

List Contents of Folder

List the contents of a folder and a subfolder on an FTP server.

First, connect to the National Centers for Environmental Information (NCEI) FTP server.

```
ftpobj = ftp('ftp.ngdc.noaa.gov')
```

```
ftpobj = 

  FTP Object
```

```
     host: ftp.ngdc.noaa.gov
     user: anonymous
      dir: /
     mode: binary
```

List the contents of the current folder on the server. At the start of a session, the current folder is the highest level folder to which you have access.

```
dir(ftpobj)
```

```
DMSP                          dmsp4alan                     ionosonde
INDEX.txt                     ftp.html                      mgg
README.txt                    geomag                        pub
STP                           google12c4c939d7b90761.html   tmp
Snow_Ice                      hazards                       wdc
Solid_Earth                   index.html
coastwatch                    international
```

Specify a subfolder and list its contents.

```
dir(ftpobj,'pub')
```

```
WebCD     coast     glac_lib  krm       outgoing  results   rgon
```

FTP service courtesy of the National Centers for Environmental Information (NCEI). See the NCEI Privacy Policy, Disclaimer, and Copyright for NCEI terms of service.

List Details of Contents

List details of the contents on an FTP server. The `dir` function can return a structure array that contains the name, modification date, and size of each item in the specified folder.

First, connect to the National Centers for Environmental Information (NCEI) FTP server.

```
ftpobj = ftp('ftp.ngdc.noaa.gov')
```

```
ftpobj =
```

```
  FTP Object
     host: ftp.ngdc.noaa.gov
     user: anonymous
      dir: /
     mode: binary
```

Return details about the items in the top-level folder on the FTP server. Some items are files and the others are folders.

```
listing = dir(ftpobj)
```

```
listing = 19×1 struct array with fields:
    name
    bytes
    isdir
    date
    datenum
```

Display details about the first item in the current folder, which is a folder named `DSMP`.

```
listing(1)
```

```
ans = struct with fields:
       name: 'DMSP'
      bytes: 32
      isdir: 1
       date: '10-Sep-2012 00:00:00'
    datenum: 735122
```

FTP service courtesy of the National Centers for Environmental Information (NCEI). See the NCEI Privacy Policy, Disclaimer, and Copyright for NCEI terms of service.

Input Arguments

`ftpobj` — Connection to FTP server
FTP object

Connection to an FTP server, specified as an FTP object.

`folder` — Name of target folder
character vector | string scalar

Name of the target folder on the FTP server, specified as a character vector or string scalar. To specify the folder above the current one, use `'..'`.

Output Arguments

`listing` — Content attributes
structure array

Content attributes, returned as an `m`-by-1 structure array, where `m` is the number of items in the folder.

This table shows the fields in the structure.

Field Name	Description	Data Type
`name`	File or folder name	`char`
`bytes`	Size of the item in bytes	`double`
`isdir`	`1` if `name` is a folder; `0` if `name` is a file	`logical`
`date`	Modification date timestamp	`char`
`datenum`	Modification date as serial date number (for more information, see `datenum`)	`double`

Tips

- The `dir` function might return a structure array in which the last four fields are empty or missing. When `dir` returns a structure with missing information, it might mean the FTP object is not configured for the operating system that is running on the FTP server. By default, an FTP object is configured to connect to a server running a UNIX operating system.

  To configure an FTP object for a connection to a server running Windows, call the `ftp` function and specify the `'System','WINDOWS'` name-value pair. Then call `dir` using the new FTP object.

See Also

`cd` | `ftp` | `mkdir` | `rmdir`

Introduced before R2006a

disableDefaultInteractivity

Disable built-in axes interactions

Syntax

```
disableDefaultInteractivity(ax)
```

Description

`disableDefaultInteractivity(ax)` disables the built-in interactions on page 1-3510 for the specified axes. You can use this function to disable the default axes interactions or a custom set of axes interactions. After disabling the interactions, you can still use the buttons in the axes toolbar to interact with the plot.

Examples

Disable and Reenable Default Axes Interactions

Create a plot. Some interactions are enabled by default, such as scrolling to zoom. Disable these interactions.

```
plot(magic(10))
ax = gca;
disableDefaultInteractivity(ax)
```

Then, reenable the interactions.

```
enableDefaultInteractivity(ax)
```

Disable Custom Axes Interactions

Create a plot, and replace the default set of axes interactions with just the rotate and zoom interactions.

```
plot(magic(10))
ax = gca;
ax.Interactions = [rotateInteraction zoomInteraction];
```

Then, disable the interactions you created.

```
disableDefaultInteractivity(ax)
```

Input Arguments

ax — Axes
`Axes` object | `PolarAxes` object | `GeographicAxes` object

Axes, specified as an `Axes`, `PolarAxes`, or `GeographicAxes` object.

More About

Built-In Axes Interactions

Built-in axes interactions allow you to explore charts using gestures, for example dragging to pan or scrolling to zoom. These interactions are built into the parent axes and are available without having to select any of the buttons in the axes toolbar. Most types of axes include a default set of built-in interactions. You can customize the default set of interactions on axes with an `Interactions` property.

If you do not want any of the interactions enabled, use the `disableDefaultInteractivity` function to disable them. To enable them if they are not already enabled, use the `enableDefaultInteractivity` function.

See Also

`enableDefaultInteractivity`

Topics

"Control Chart Interactivity"

Introduced in R2018b

discretize

Group data into bins or categories

Syntax

```
Y = discretize(X,edges)
[Y,E] = discretize(X,N)
[Y,E] = discretize(X,dur)
[ ___ ] = discretize( ___ ,values)

[ ___ ] = discretize( ___ ,'categorical')
[ ___ ] = discretize( ___ ,'categorical',displayFormat)
[ ___ ] = discretize( ___ ,'categorical',categoryNames)

[ ___ ] = discretize( ___ ,'IncludedEdge',side)
```

Description

`Y = discretize(X,edges)` returns the indices of the bins that contain the elements of `X`. The `j`th bin contains element `X(i)` if `edges(j) <= X(i) < edges(j+1)` for `1 <= j < N`, where `N` is the number of bins and `length(edges) = N+1`. The last bin contains both edges such that `edges(N) <= X(i) <= edges(N+1)`.

`[Y,E] = discretize(X,N)` divides the data in `X` into `N` bins of uniform width, and also returns the bin edges `E`.

`[Y,E] = discretize(X,dur)`, where `X` is a datetime or duration array, divides `X` into uniform bins of `dur` length of time. `dur` can be a scalar `duration` or `calendarDuration`, or a unit of time. For example, `[Y,E] = discretize(X,'hour')` divides `X` into bins with a uniform duration of 1 hour.

`[ ___ ] = discretize( ___ ,values)` returns the corresponding element in `values` rather than the bin number, using any of the previous input or output argument combinations. For example, if `X(1)` is in bin 5, then `Y(1)` is `values(5)` rather than `5`. `values` must be a vector with length equal to the number of bins.

`[ ___ ] = discretize( ___ ,'categorical')` creates a categorical array where each bin is a category. In most cases, the default category names are of the form "`[A,B)`" (or "`[A,B]`" for the last bin), where `A` and `B` are consecutive bin edges. If you specify `dur` as a character vector, then the default category names might have special formats. See `Y` for a listing of the display formats.

`[ ___ ] = discretize( ___ ,'categorical',displayFormat)`, for datetime or duration array inputs, uses the specified datetime or duration display format in the category names of the output.

`[ ___ ] = discretize( ___ ,'categorical',categoryNames)` also names the categories in `Y` using the cell array of character vectors, `categoryNames`. The length of `categoryNames` must be equal to the number of bins.

`[ ___ ] = discretize( ___ ,'IncludedEdge',side)`, where `side` is `'left'` or `'right'`, specifies whether each bin includes its right or left bin edge. For example, if `side` is `'right'`, then each bin includes the right bin edge, except for the *first* bin which includes both edges. In this case, the `j`th bin contains an element `X(i)` if `edges(j) < X(i) <= edges(j+1)`, where `1 < j <= N` and `N` is the number of bins. The first bin includes the left edge such that it contains `edges(1) <= X(i) <= edges(2)`. The default for `side` is `'left'`.

Examples

Group Data into Bins

Use `discretize` to group numeric values into discrete bins. `edges` defines five bin edges, so there are four bins.

```
data = [1 1 2 3 6 5 8 10 4 4]
```

```
data = 1×10

     1     1     2     3     6     5     8    10     4     4
```

```
edges = 2:2:10
```

```
edges = 1×5
```

```
     2     4     6     8    10
```

```
Y = discretize(data,edges)
```

```
Y = 1×10

   NaN   NaN     1     1     3     2     4     4     2     2
```

Y indicates which bin each element of data belongs to. Since the value 1 falls outside the range of the bins, Y contains NaN values for those elements.

Group Data into Specified Number of Bins

Group random data into three bins. Specify a second output to return the bin edges calculated by discretize.

```
X = randn(10,1);
[Y,E] = discretize(X,3)
```

```
Y = 10×1

     2
     2
     1
     2
     2
     1
     1
     2
     3
     2


E = 1×4

    -3     0     3     6
```

Group Datetime Data by Month

Create a 10-by-1 datetime vector with random dates in the year 2016. Then, group the datetime values by month and return the result as a categorical array.

```
X = datetime(2016,1,randi(365,10,1))
```

```
X = 10x1 datetime array
   24-Oct-2016
   26-Nov-2016
   16-Feb-2016
   29-Nov-2016
   18-Aug-2016
   05-Feb-2016
   11-Apr-2016
   18-Jul-2016
   15-Dec-2016
   18-Dec-2016
```

```
Y = discretize(X,'month','categorical')
```

```
Y = 10x1 categorical array
     Oct-2016
     Nov-2016
     Feb-2016
     Nov-2016
     Aug-2016
     Feb-2016
     Apr-2016
     Jul-2016
     Dec-2016
     Dec-2016
```

Change Display Format of Duration Values

Group duration values by hour and return the result in a variety of display formats.

Group some random duration values by hour and return the results as a categorical array.

```
X = hours(abs(randn(1,10)))'
```

```
X = 10x1 duration array
   0.53767 hr
    1.8339 hr
    2.2588 hr
   0.86217 hr
   0.31877 hr
    1.3077 hr
   0.43359 hr
   0.34262 hr
    3.5784 hr
    2.7694 hr
```

```
Y = discretize(X,'hour','categorical')
```

```
Y = 10x1 categorical array
     [0 hr, 1 hr)
     [1 hr, 2 hr)
     [2 hr, 3 hr)
     [0 hr, 1 hr)
     [0 hr, 1 hr)
     [1 hr, 2 hr)
     [0 hr, 1 hr)
     [0 hr, 1 hr)
     [3 hr, 4 hr]
     [2 hr, 3 hr)
```

Change the display of the results to be a number of minutes.

```
Y = discretize(X,'hour','categorical','m')
```

```
Y = 10x1 categorical array
     [0 min, 60 min)
     [60 min, 120 min)
     [120 min, 180 min)
     [0 min, 60 min)
     [0 min, 60 min)
     [60 min, 120 min)
     [0 min, 60 min)
     [0 min, 60 min)
     [180 min, 240 min]
     [120 min, 180 min)
```

Change the format again to display as a number of hours, minutes and seconds.

```
Y = discretize(X,'hour','categorical','hh:mm:ss')
```

```
Y = 10x1 categorical array
     [00:00:00, 01:00:00)
     [01:00:00, 02:00:00)
     [02:00:00, 03:00:00)
     [00:00:00, 01:00:00)
     [00:00:00, 01:00:00)
     [01:00:00, 02:00:00)
     [00:00:00, 01:00:00)
     [00:00:00, 01:00:00)
     [03:00:00, 04:00:00]
     [02:00:00, 03:00:00)
```

Assign Bin Values

Use the right edge of each bin as the `values` input. The values of the elements in each bin are always less than the bin value.

```
X = randi(100,1,10);
edges = 0:25:100;
values = edges(2:end);
Y = discretize(X,edges,values)
```

```
Y = 1×10

   100   100    25   100    75    25    50    75   100   100
```

Include Right Edge of Each Bin

Use the `'IncludedEdge'` input to specify that each bin includes its right bin edge. The first bin includes both edges. Compare the result to the default inclusion of left bin edges.

```
X = 1:2:11;
edges = [1 3 4 7 10 11];
Y = discretize(X,edges,'IncludedEdge','right')
```

```
Y = 1×6
```

```
     1     1     3     3     4     5
```

```
Z = discretize(X,edges)
```

```
Z = 1×6

     1     2     3     4     4     5
```

Group Data into Categorical Array

Group numeric data into a categorical array. Use the result to confirm the amount of data that falls within 1 standard deviation of the mean value.

Group normally distributed data into bins according to the distance from the mean, measured in standard deviations.

```
X = randn(1000,1);
edges = std(X)*(-3:3);
Y = discretize(X,edges, 'categorical', ...
    {'-3sigma', '-2sigma', '-sigma', 'sigma', '2sigma', '3sigma'});
```

Y contains undefined categorical values for the elements in X that are farther than 3 standard deviations from the mean.

Preview the values in Y.

```
Y(1:15)
```

```
ans = 15x1 categorical array
     sigma
     2sigma
     -3sigma
     sigma
     sigma
     -2sigma
     -sigma
     sigma
     <undefined>
     3sigma
     -2sigma
```

```
<undefined>
sigma
-sigma
sigma
```

Confirm that approximately 68% of the data falls within one standard deviation of the mean.

```
nnz(Y=='-sigma' | Y=='sigma')/numel(Y)
```

```
ans = 0.6910
```

Input Arguments

X — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array. X contains the data that you want to distribute into bins.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `datetime` | `duration`

edges — Bin edges
numeric vector

Bin edges, specified as a monotonically increasing numeric vector. Consecutive elements in `edges` form discrete bins, which `discretize` uses to partition the data in X. By default, each bin includes the left bin edge, except for the last bin, which includes both bin edges.

`edges` must have at least two elements, since `edges(1)` is the left edge of the first bin and `edges(end)` is the right edge of the last bin.

Example: `Y = discretize([1 3 5],[0 2 4 6])` distributes the values 1, 3, and 5 into three bins, which have edges `[0,2)`, `[2,4)`, and `[4,6]`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `datetime` | `duration`

N — Number of bins
scalar integer

Number of bins, specified as a scalar integer.

`discretize` divides the data into `N` bins of uniform width, choosing the bin edges to be "nice" numbers that overlap the range of the data. The largest and smallest elements in `X` do not typically fall right on the bin edges. If the data is unevenly distributed, then some of the intermediate bins can be empty. However, the first and last bin always include at least one piece of data.

Example: `[Y,E] = discretize(X,5)` distributes the data in `X` into 5 bins with a uniform width.

dur — Uniform bin duration
scalar `duration` | scalar `calendarDuration` | `'second'` | `'minute'` | `'hour'` | `'day'` | `'week'` | `'month'` | `'quarter'` | `'year'` | `'decade'` | `'century'`

Uniform bin duration, specified as a scalar `duration` or `calendarDuration`, or as one of the values in the table.

If you specify `dur`, then `discretize` can use a maximum of 65,536 bins (or 2^{16}). If the specified bin duration requires more bins, then `discretize` uses a larger bin width corresponding to the maximum number of bins.

Value	Works with...	Description
`'second'`	Datetime or duration values	Each bin is 1 second.
`'minute'`	Datetime or duration values	Each bin is 1 minute.
`'hour'`	Datetime or duration values	Each bin is 1 hour.
`'day'`	Datetime or duration values	• For datetime inputs, each bin is 1 calendar day. This value accounts for Daylight Saving Time shifts. • For duration inputs, each bin is 1 fixed-length day (24 hours).
`'week'`	Datetime values	Each bin is 1 calendar week.

Value	Works with...	Description
`'month'`	Datetime values	Each bin is 1 calendar month.
`'quarter'`	Datetime values	Each bin is 1 calendar quarter.
`'year'`	Datetime or duration values	• For datetime inputs, each bin is 1 calendar year. This value accounts for leap days. • For duration inputs, each bin is 1 fixed-length year (365.2425 days).
`'decade'`	Datetime values	Each bin is 1 decade (10 calendar years).
`'century'`	Datetime values	Each bin is 1 century (100 calendar years).

Example: `[Y,E] = discretize(X,'hour')` divides `X` into bins with a uniform duration of 1 hour.

Data Types: `char` | `duration` | `calendarDuration`

`values` — Bin values
vector

Bin values, specified as a vector of any data type. `values` must have the same length as the number of bins, `length(edges)-1`. The elements in `values` replace the normal bin index in the output. That is, if `X(1)` falls into bin `2`, then `discretize` returns `Y(1)` as `values(2)` rather than `2`.

If `values` is a cell array, then all the input data must belong to a bin.

Example: `Y = discretize(randi(5,10,1),[1 1.5 3 5],diff([1 1.5 3 5]))` returns the widths of the bins, rather than indices ranging from 1 to 3.

`displayFormat` — Datetime and duration display format
character vector

Datetime and duration display format, specified as a character vector. The `displayFormat` value does not change the values in `Y`, only their display. You can specify

`displayFormat` using any valid display format for datetime and duration arrays. For more information about the available options, see “Set Date and Time Display Format”.

Example: `discretize(X,'day','categorical','h')` specifies a display format for a duration array.

Example: `discretize(X,'day','categorical','yyyy-MM-dd')` specifies a display format for a datetime array.

Data Types: `char`

`categoryNames` — Categorical array category names
cell array of character vectors

Categorical array category names, specified as a cell array of character vectors. `categoryNames` must have length equal to the number of bins.

Example: `Y = discretize(randi(5,10,1),[1 1.5 3 5],'categorical',{'A' 'B' 'C'})` distributes the data into three categories, `A`, `B`, and `C`.

Data Types: `cell`

`side` — Edges to include in each bin
`'left'` (default) | `'right'`

Edges to include in each bin, specified as one of these values:

- `'left'` — All bins include the left bin edge, except for the last bin, which includes both edges. This is the default.
- `'right'` — All bins include the right bin edge, except for the first bin, which includes both edges.

Example: `Y = discretize(randi(11,10,1),1:2:11,'IncludedEdge','right')` includes the right bin edge in each bin.

Output Arguments

`Y` — Bins
vector | matrix | multidimensional array | ordinal categorical array

Bins, returned as a numeric vector, matrix, multidimensional array, or ordinal categorical array. `Y` is the same size as `X`, and each element describes the bin placement for the

corresponding element in X. If values is specified, then the data type of Y is the same as values. Out-of-range elements are expressed differently depending on the data type of the output:

- For numeric outputs, Y contains NaN values for out-of-range elements in X (where X(i) < edges(1) or X(i) > edges(end)), or where X contains a NaN.
- If Y is a categorical array, then it contains undefined elements for out-of-range or NaN inputs.
- If values is a vector of an integer data type, then Y contains 0 for out-of-range or NaN inputs.

The default category name formats in Y for the syntax discretize(X,dur,'categorical') are:

Value of dur	Default Category Name Format	Format Example
'second'	global default format	28-Jan-2016 10:32:06
'minute'		
'hour'		
'day'	global default date format	28-Jan-2016
'week'	[global_default_date_format, global_default_date_format)	[24-Jan-2016, 30-Jan-2016)
'month'	'MMM-uuuu'	Jun-2016
'quarter'	'QQQ uuuu'	Q4 2015
'year'	'uuuu'	2016
'decade'	'[uuuu, uuuu)'	[2010, 2020)
'century'		

E — Bin edges
vector

Bin edges, returned as a vector. Specify this output to see the bin edges that discretize calculates in cases where you do not explicitly pass in the bin edges.

Tips

- The behavior of `discretize` is similar to that of the `histcounts` function. Use `histcounts` to find the number of elements in each bin. On the other hand, use `discretize` to find which bin each element belongs to (without counting).

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

- The `categorical` option is not supported.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `categorical` option is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`categorical` | `histcounts` | `histogram`

Topics

"Replace Discouraged Instances of hist and histc"

Introduced in R2015a

disp

Display value of variable

Syntax

```
disp(X)
```

Description

`disp(X)` displays the value of variable `X` without printing the variable name. Another way to display a variable is to type its name, which displays a leading "`X =`" before the value.

If a variable contains an empty array, `disp` returns without displaying anything.

Examples

Display Variable Values

Create a variable with numbers and another variable with text.

```
A = [15 150];
S = 'Hello World.';
```

Display the value of each variable.

```
disp(A)
```

```
    15   150
```

```
disp(S)
```

```
Hello World.
```

Display Matrix with Column Labels

Display a matrix and label the columns as `Corn`, `Oats`, and `Hay`.

```
X = rand(5,3);
disp('     Corn      Oats      Hay')
```

```
     Corn      Oats      Hay
```

```
disp(X)
```

```
    0.8147    0.0975    0.1576
    0.9058    0.2785    0.9706
    0.1270    0.5469    0.9572
    0.9134    0.9575    0.4854
    0.6324    0.9649    0.8003
```

Display Hyperlink in Command Window

Display a link to a Web page by including HTML hyperlink code as input to `disp`. For example, display a link to the MathWorks Web site.

```
X = '<a href = "https://www.mathworks.com">MathWorks Web Site</a>';
disp(X)
```

```
MathWorks Web Site
```

Display Multiple Variables on Same Line

Here are three ways to display multiple variable values on the same line in the Command Window.

Concatenate multiple character vectors together using the `[]` operator. Convert any numeric values to characters using the `num2str` function. Use `disp` to display the result.

```
name = 'Alice';
age = 12;
X = [name,' will be ',num2str(age),' this year.'];
disp(X)
```

```
Alice will be 12 this year.
```

Use `sprintf` to create text, and then display it with `disp`.

```
name = 'Alice';
age = 12;
X = sprintf('%s will be %d this year.',name,age);
disp(X)
```

```
Alice will be 12 this year.
```

Use `fprintf` to directly display the text without creating a variable. However, to terminate the display properly, you must end the text with the newline (`\n`) metacharacter.

```
name = 'Alice';
age = 12;
fprintf('%s will be %d this year.\n',name,age);
```

```
Alice will be 12 this year.
```

Input Arguments

X — Input array
array

Input array.

To display more than one array, you can use concatenation or the `sprintf` or `fprintf` functions as shown in the example, “Display Multiple Variables on Same Line” on page 1-3527.

See Also

`format` | `fprintf` | `int2str` | `num2str` | `sprintf`

Introduced before R2006a

display

Show information about variable or expression result

Syntax

```
display(X)
```

Description

`display(X)` is called by MATLAB when a statement or expression is not terminated by a semicolon. Omit the terminating semicolon from a statement or expression when you want to see an intermediate result.

MATLAB calls the `display` function to show information about an intermediate result, such as the values, size, type, and variable name.

To show the value of a variable or to show program output in the command window, use the `disp` function.

To customize the display of user-defined objects, use the techniques described in the “Customize Object Display for Classes” topic.

Examples

Unterminated Variable Assignment

MATLAB calls `display` when you make an assignment to a variable without terminating the statement with a semicolon. In this example, `display` shows the variable name and the value.

```
a = 7
```

```
a =

     7
```

Terminated Variable Assignment

MATLAB does not call `display` when you make an assignment to a variable and the statement is terminated with a semicolon.

```
a = 7;
```

Unterminated Expression

When you execute an expression without a semicolon, MATLAB assigns the result to a variable called `ans`, which the `display` function shows in the command window.

```
format long
sqrt(2)
```

```
ans =

   1.414213562373095
```

Input Arguments

X — Result of executing a statement or expression
variable | expression

Result of executing a statement or expression, passed to the `display` function by MATLAB.

More About

Assignment to ans

Executing an expression without terminating the expression with a semicolon causes the result to be displayed in the command window. MATLAB assigns the result of an

expression to a variable named `ans` when the result is not assigned to a variable explicitly.

```
4 * 5 - 13
```

```
ans =

    7
```

To display the result in the command window without displaying `ans`, use the `disp` function.

```
disp(4 * 5 - 13)
```

```
7
```

If an expression is terminated by a semicolon, MATLAB does not display a value, but still assigns the result to the `ans` variable.

```
4 * 5 - 13;
ans
```

```
ans =

     7
```

Display Results in Command Window

Omitting the terminating semicolon is useful when you want to see intermediate results from statements in a program. For example, compare these two statements by omitting the semicolon. The `display` function shows the results in the command window.

```
result1 = 4 * 5 - 13
```

```
result1 =

     7
```

```
result2 = 4 * (5 - 13)
```

```
result2 =

   -32
```

Information Shown by the `display` Function

The `display` function provides information about the kind of values that are the result of executing a statement or expression. This information is useful for understanding how a program or script works.

For example, this statement assigns a `uint8` vector of values `1 2 3 4` to the variable named `a`. The `display` function shows the variable name, the size and type, and the values.

```
a = uint8([1 2 3 4])
```

```
a =

  1×4 uint8 row vector

   1   2   3   4
```

For empty values (numeric types, `char`, `struct`, and `cell`) the `display` function displays:

- `[]` — for numeric types
- `"0x0 struct array with no fields."` — for empty `structs`.
- `"0x0 empty cell array"` — for empty cell arrays.
- `"0x0 empty char array"` — for empty `char` arrays
- `"0x0 empty string array"` — for empty `string` arrays

To show the actual values that are the intended output from a program, like text and numbers, call the `disp` function. The `disp` function does not display the variable name or `ans`. Also, `disp` displays nothing for built-in types (numeric types, `char`, `struct`, and `cell`) when the value is empty.

See Also

`ans` | `disp` | `fprintf` | `matlab.mixin.CustomDisplay` | `sprintf`

Topics

"Customize Object Display for Classes"

Introduced before R2006a

dissect

Nested dissection permutation

Syntax

```
p = dissect(A)
p = dissect(A,Name,Value)
```

Description

`p = dissect(A)` returns a permutation vector computed using nested dissection of the sparsity structure of `A`.

`p = dissect(A,Name,Value)` specifies additional options using one or more name-value pair arguments. For example, `dissect(A,'NumIterations',15)` uses 15 refinement iterations in the nested dissection algorithm instead of 10.

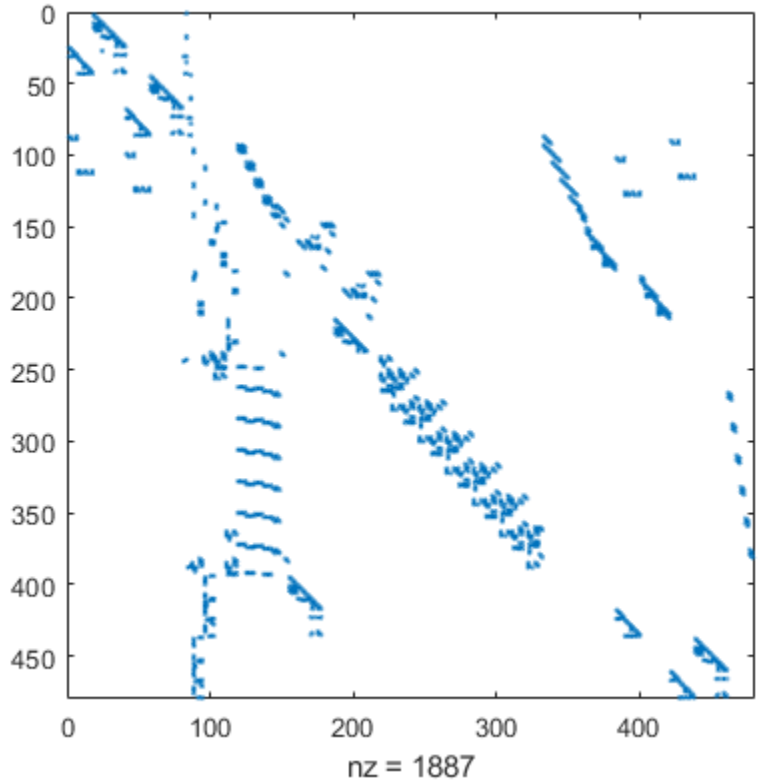
Examples

Reorder Sparse Matrix Columns

Reorder a sparse matrix with several methods and compare the fill-in incurred by the LU decomposition of the reordered matrices.

Load the `west0479` matrix, which is a real-valued 479-by-479 sparse matrix with both real and complex pairs of conjugate eigenvalues. View the sparsity structure.

```
load west0479.mat
A = west0479;
spy(A)
```



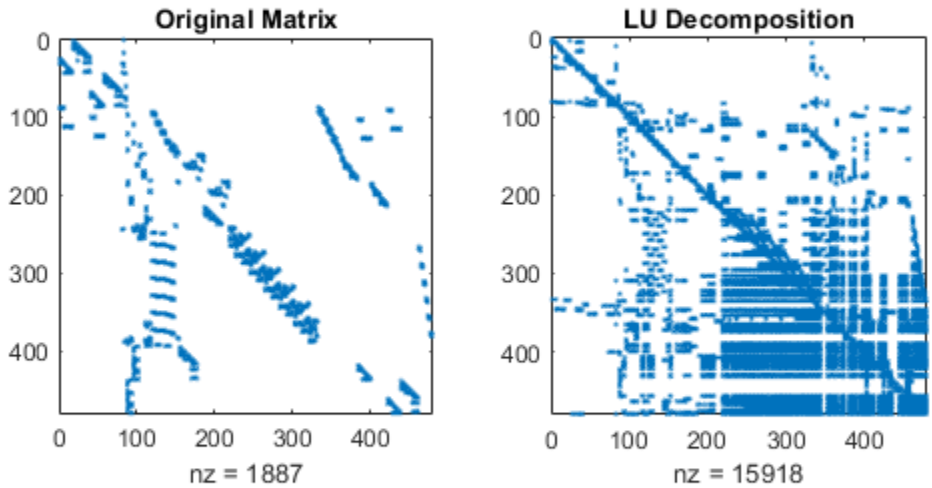
Calculate several different permutations of the matrix columns, including the nested dissection ordering.

```
p1 = dissect(A);
p2 = amd(A);
p3 = symrcm(A);
```

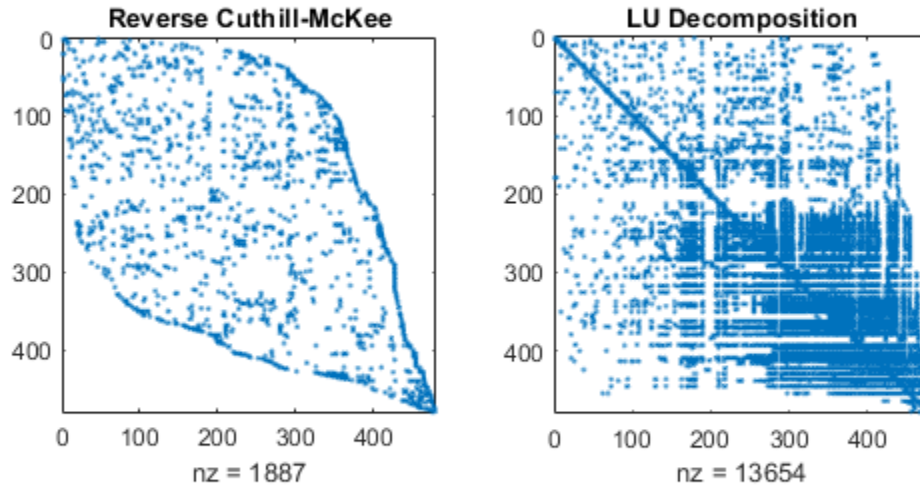
Compare the sparsity structures of the LU decomposition of `A` using the different ordering methods. The `dissect` function produces the reordering that incurs the least amount of fill-in.

```
subplot(1,2,1)
spy(A)
title('Original Matrix')
```

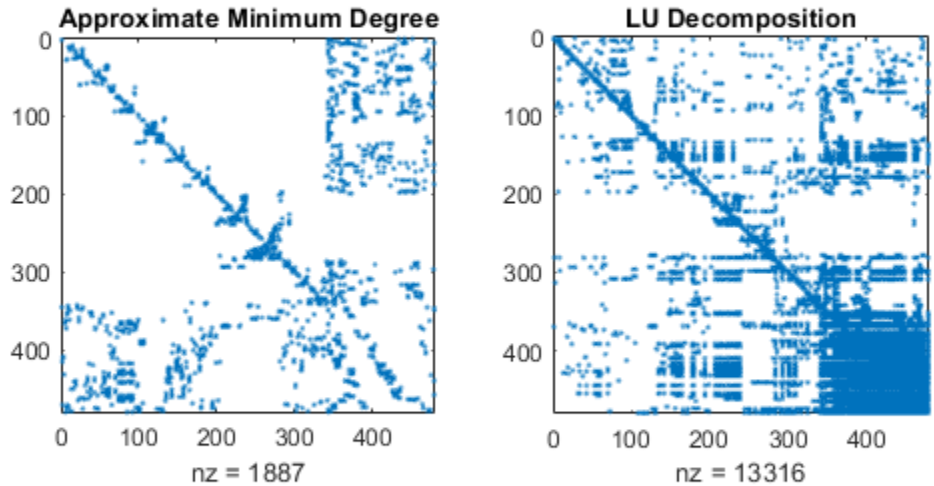
```
subplot(1,2,2)
spy(lu(A))
title('LU Decomposition')
```



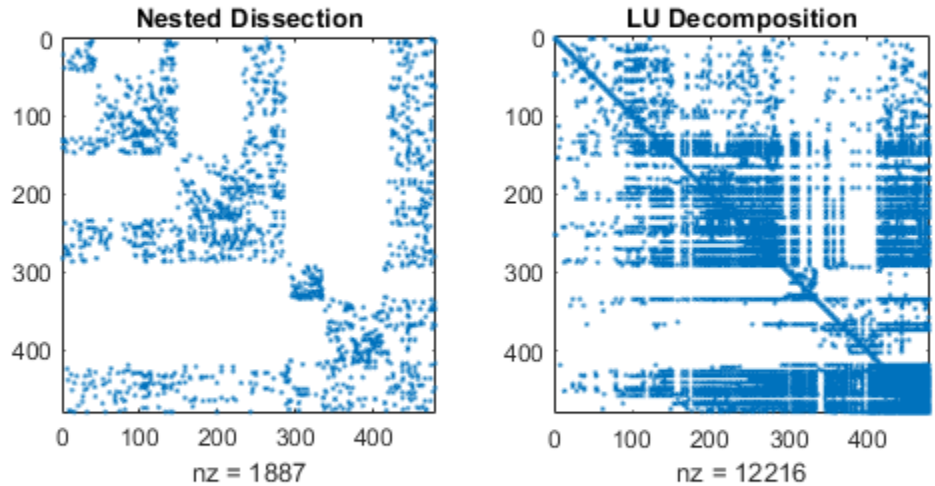

```
figure
subplot(1,2,1)
spy(A(p3,p3))
title('Reverse Cuthill-McKee')
subplot(1,2,2)
spy(lu(A(p3,p3)))
title('LU Decomposition')
```




```
figure
subplot(1,2,1)
spy(A(p2,p2))
title('Approximate Minimum Degree')
subplot(1,2,2)
spy(lu(A(p2,p2)))
title('LU Decomposition')
```




```
figure
subplot(1,2,1)
spy(A(p1,p1))
title('Nested Dissection')
subplot(1,2,2)
spy(lu(A(p1,p1)))
title('LU Decomposition')
```

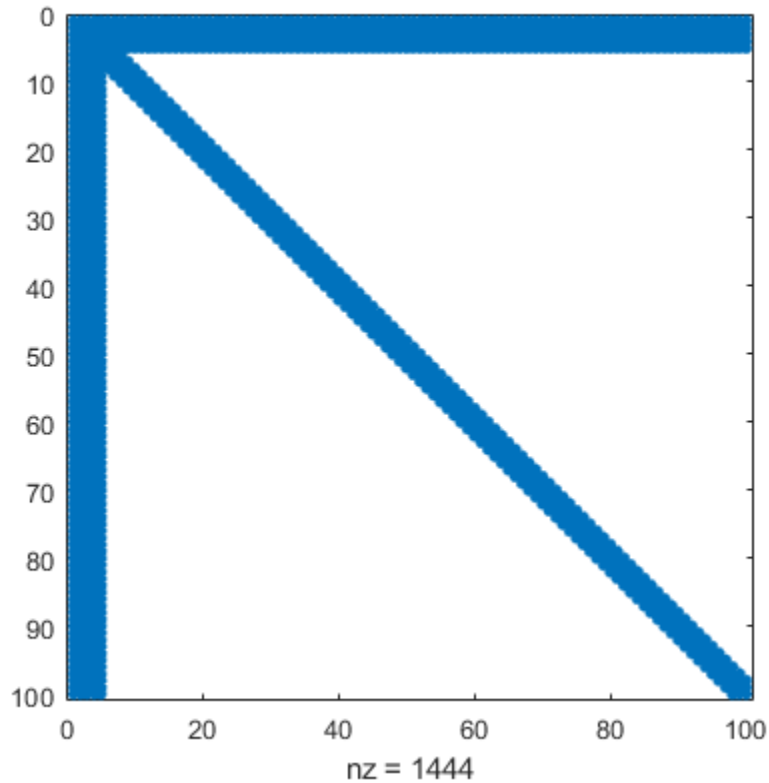


Filter Out Dense Columns

An arrowhead matrix is a sparse matrix that has a few dense columns. Use the `'MaxDegreeThreshold'` name-value pair to filter the dense columns to the end of the reordered matrix.

Create an arrowhead sparse matrix and view the sparsity pattern.

```
A = speye(100) + diag(ones(1,99),1) + diag(ones(1,98),2);
A(1:5,:) = ones(5,100);
A = A + A';
spy(A)
```

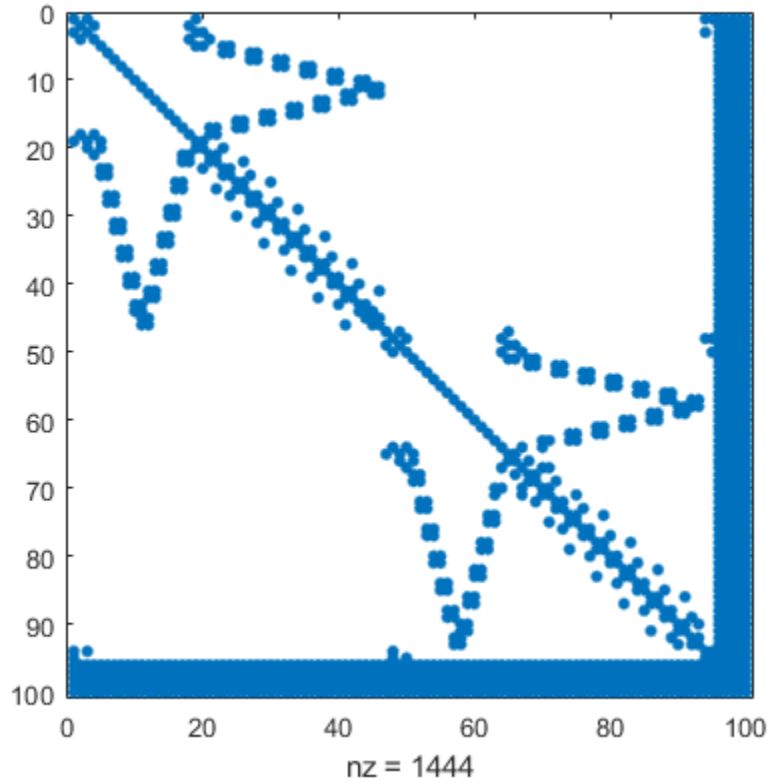


Calculate the nested dissection ordering, and filter out the columns that have more than 10 nonzero elements.

```
p = dissect(A,'MaxDegreeThreshold',10);
```

View the sparsity pattern of the reordered matrix. `dissect` places the dense columns at the end of the reordered matrix.

```
spy(A(p,p))
```



Input Arguments

A — Input matrix
square matrix

Input matrix, specified as a square matrix. A can be either full or sparse. If A is nonsymmetric, then `dissect` symmetrizes it.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `p = dissect(A,'NumIterations',15,'NumSeparators',2)` uses 15 refinement iterations and 2 separators in the nested dissection algorithm.

`VertexWeights` — Vertex weights
vector

Vertex weights, specified as the comma-separated pair consisting of `'VertexWeights'` and a vector. The vector of weights must have length equal to `size(A,1)` so that a weight is specified for each vertex. Use this option to specify how the graph vertices (matrix columns) are weighted, which affects how the algorithm computes the balance between partitions.

By default, the nested dissection algorithm weights all vertices equally.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`NumSeparators` — Number of separators
`1` (default) | positive integer

Number of separators, specified as the comma-separated pair consisting of `'NumSeparators'` and a positive integer. Use this option to specify how many partitions the graph is split into during each partitioning step. Increasing the number of separators in the nested dissection algorithm can result in a higher quality permutation at the cost of additional execution time.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`NumIterations` — Number of refinement iterations
`10` (default) | positive integer

Number of refinement iterations, specified as the comma-separated pair consisting of `'NumIterations'` and a positive integer. More refinement iterations can result in a higher quality permutation at the cost of increased execution time.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`MaxImbalance` — Threshold for partition imbalance
`1.2` (default) | scalar

Threshold for partition imbalance, specified as the comma-separated pair consisting of `'MaxImbalance'` and a scalar value that is an integer multiple of `0.001` greater than or equal to `1.001` and less than or equal to `1.999`. Larger threshold values might reduce execution time by allowing the algorithm to accept a worse permutation.

Data Types: `single` | `double`

`MaxDegreeThreshold` — Threshold for vertex degree
`0` (default) | nonnegative integer

Threshold for vertex degree, specified as the comma-separated pair consisting of `'MaxDegreeThreshold'` and a nonnegative integer. The nested dissection algorithm ignores vertices with degree larger than `threshold*(avg degree)/10` during ordering. `dissect` places vertices ignored in this way at the end of the permutation. This effectively places any vertices with degree greater than the threshold in the first, top-level separator. Filtering out highly connected vertices can sometimes improve the speed and accuracy of the ordering.

The default value of `0` means that all vertices are ordered.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`p` — Permutation vector
vector

Permutation vector, returned as a vector. Use the permutation vector to reorder the columns of `A` using the indexing expression `A(p,p)`. For example, the Cholesky factorization `chol(A(p,p))` tends to be sparser than that of `A`.

Algorithms

The nested dissection ordering algorithm described in [1] is a multilevel graph partitioning algorithm that is used to produce fill-reducing orderings of sparse matrices. The input matrix is treated as the adjacency matrix of a graph. The algorithm coarsens the graph by collapsing vertices and edges, reorders the smaller graph, and then uses refinement steps to uncoarsen the small graph and produce a reordering of the original graph.

The name-value pairs for `dissect` enable you to control various stages of the algorithm:

- **Coarsening**

  In this phase, the algorithm creates successively smaller graphs from the original graph by collapsing together adjacent pairs of vertices. `'MaxDegreeThreshold'` enables you to filter out highly connected graph vertices (which are dense columns in the matrix) by ordering them last.

- **Partitioning**

  After the graph is coarsened, the algorithm completely reorders the smaller graph. At each partitioning step, the algorithm attempts to partition the graph into equal parts: `'NumSeparators'` specifies how many parts to partition the graph into, `'VertexWeights'` optionally assigns weights to the vertices, and `'MaxImbalance'` specifies the threshold for the difference in weight between the different partitions.

- **Refinement**

  After the smallest graph is reordered, the algorithm makes projections to enlarge the graph back to the original size by expanding the vertices that were previously combined. After each projection step, a refinement step is performed that moves vertices between partitions to improve the quality of the solution. `'NumIterations'` controls how many refinement steps are used during this uncoarsening phase.

References

[1] Karypis, George and Vipin Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs." *SIAM Journal on Scientific Computing*. Vol. 20, Number 1, 1999, pp. 359–392.

See Also

`amd` | `colamd` | `colperm` | `symamd` | `symrcm`

Topics

"Permutations and Reordering"

Introduced in R2017b

dither

Convert image, increasing apparent color resolution by dithering

Syntax

```
X = dither(RGB,map)
X = dither(RGB,map,Qm,Qe)
BW = dither(I)
```

Description

`X = dither(RGB,map)` creates an indexed image approximation of the RGB image by dithering the colors in the colormap `map`.

`X = dither(RGB,map,Qm,Qe)` also specifies the number `Qm` of quantization bits to use along each color axis for the inverse color map, and the number `Qe` of quantization bits to use for the color space error calculations.

`BW = dither(I)` converts the grayscale image `I` to the binary (black and white) image `BW` by dithering.

Examples

Convert Grayscale Image to Binary Image Using Dithering

Read the grayscale image from the `corn.tif` file into the MATLAB® workspace. The grayscale version of the image is the third image in the file.

```
corn_gray = imread('corn.tif',3);
```

Display the grayscale image using `imshow`.

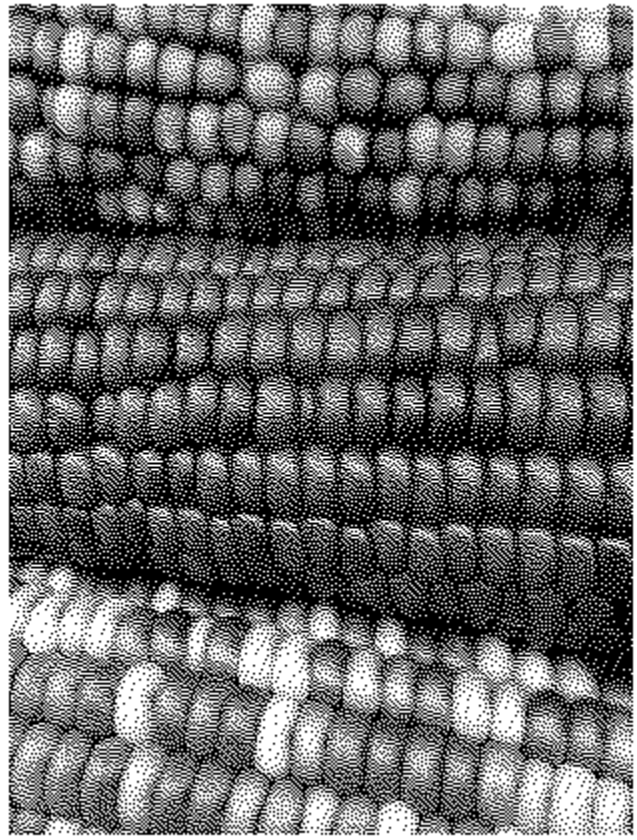
```
imshow(corn_gray)
```

Convert the image to binary by using the `dither` function.

```
corn_bw = dither(corn_gray);
```

Display the binary image. Although pixels in the binary image only have the value `0` or `1`, the image appears to have shades of gray because of dithering.

```
imshow(corn_bw)
```



Input Arguments

RGB — RGB image
m-by-*n*-by-3 array of nonnegative numbers

RGB image, specified as an *m*-by-*n*-by-3 array of nonnegative numbers. If you specify an image of data type `double`, then values must be in the range [0, 1].

Data Types: `single` | `double` | `uint8` | `uint16`

`map` — Input colormap
c-by-3 matrix

Input colormap, specified as a *c*-by-3 matrix with values in the range [0, 1]. Each row of `map` is a three-element RGB triplet that specifies the red, green, and blue components of a single color of the colormap. The colormap has a maximum of 65,536 colors.

Data Types: `double`

`Qm` — Quantization bits along each color axis
5 (default) | positive integer

Quantization bits along each color axis for the inverse color map, specified as a positive integer. If `Qe` is less than `Qm`, then dithering cannot be performed and the `dither` function returns an undithered indexed image in `X`.

Data Types: `double`

`Qe` — Quantization bits for color space error calculations
8 (default) | positive integer

Quantization bits for the color space error calculations, specified as a positive integer. If `Qe` is less than `Qm`, then dithering cannot be performed and the `dither` function returns an undithered indexed image in `X`.

Data Types: `double`

`I` — Grayscale image
m-by-*n* matrix of nonnegative numbers

Grayscale image, specified as an *m*-by-*n* matrix of nonnegative numbers. If you specify an image of data type `double`, then values must be in the range [0, 1].

Data Types: `single` | `double` | `int16` | `uint8` | `uint16`

Output Arguments

X — Indexed image
m-by-*n* matrix of nonnegative integers

Indexed image, returned as an *m*-by-*n* matrix of nonnegative integers. If the length of `map` is less than or equal to 256, then the output image is of class `uint8`. Otherwise, the output image is of class `uint16`. The value `0` in the output array `X` corresponds to the first color in the colormap.

Note The values in image `X` are indexes into the colormap `map` and should not be used in mathematical processing, such as filtering operations.

Data Types: `uint8` | `uint16`

BW — Binary image
m-by-*n* logical matrix

Binary image, returned as an *m*-by-*n* logical matrix.

Data Types: `logical`

Algorithms

`dither` increases the apparent color resolution of an image by applying Floyd-Steinberg's error diffusion dither algorithm [1].

References

[1] Floyd, R. W., and L. Steinberg, "An Adaptive Algorithm for Spatial Gray Scale," *International Symposium Digest of Technical Papers*, Society for Information Displays, 1975, p. 36.

[2] Lim, Jae S., *Two-Dimensional Signal and Image Processing*, Englewood Cliffs, NJ, Prentice Hall, 1990, pp. 469–476.

See Also

`imapprox` | `rgb2ind`

Topics

"Image Types"
"Reduce the Number of Colors in an Image" (Image Processing Toolbox)

Introduced before R2006a

divergence

Compute divergence of vector field

Syntax

```
div = divergence(X,Y,Z,U,V,W)
div = divergence(U,V,W)
div = divergence(X,Y,U,V)
div = divergence(U,V)
```

Description

`div = divergence(X,Y,Z,U,V,W)` computes the divergence of a 3-D vector field having vector components `U`, `V`, `W`.

The arrays `X`, `Y`, and `Z`, which define the coordinates for the vector components `U`, `V`, and `W`, must be monotonic, but do not need to be uniformly spaced. `X`, `Y`, and `Z` must have the same number of elements.

`div = divergence(U,V,W)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`div = divergence(X,Y,U,V)` computes the divergence of a 2-D vector field `U`, `V`.

The arrays `X` and `Y`, which define the coordinates for `U` and `V`, must be monotonic, but do not need to be uniformly spaced. `X` and `Y` must have the same number of elements, as if produced by `meshgrid`.

`div = divergence(U,V)` assumes `X` and `Y` are determined by the expression

```
[X Y] = meshgrid(1:n,1:m)
```

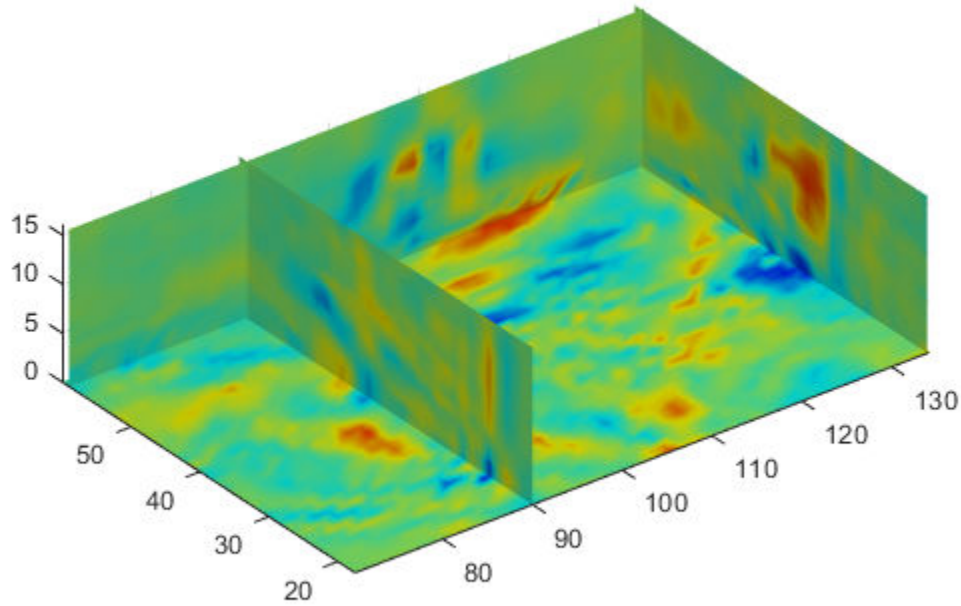
where `[m,n] = size(U)`.

Examples

Divergence of Vector Volume Data as Slice Planes

Display the divergence of vector volume data as slice planes. Use color to indicate divergence.

```
load wind
div = divergence(x,y,z,u,v,w);
h = slice(x,y,z,div,[90 134],59,0);
colormap('jet');
shading interp
daspect([1 1 1]);
axis tight
camlight
set([h(1),h(2)],'ambientstrength',.6);
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`curl` | `isosurface` | `streamtube`

Topics

“Overview of Volume Visualization”
“Displaying Divergence with Stream Tubes”

Introduced before R2006a

dlmread

(Not recommended) Read ASCII-delimited file of numeric data into matrix

Note `dlmread` is not recommended. Use `readmatrix` instead. For more information, see Compatibility Considerations.

Syntax

```
M = dlmread(filename)
M = dlmread(filename,delimiter)
M = dlmread(filename,delimiter,R1,C1)
M = dlmread(filename,delimiter,[R1 C1 R2 C2])
```

Description

`M = dlmread(filename)` reads an ASCII-delimited numeric data file into matrix `M`. The `dlmread` function detects the delimiter from the file and treats repeated white spaces as a single delimiter.

`M = dlmread(filename,delimiter)` reads data from the file using the specified delimiter and treats repeated delimiter characters as separate delimiters.

`M = dlmread(filename,delimiter,R1,C1)` starts reading at row offset `R1` and column offset `C1`. For example, the offsets `R1=0`, `C1=0` specify the first value in the file.

To specify row and column offsets without specifying a delimiter, use an empty character as a placeholder, for example, `M = dlmread(filename,'',2,1)`.

`M = dlmread(filename,delimiter,[R1 C1 R2 C2])` reads only the range bounded by row offsets `R1` and `R2` and column offsets `C1` and `C2`. Another way to define the range is to use spreadsheet notation, such as `'A1..B7'` instead of `[0 0 6 1]`.

Examples

Read Entire Delimited File

Read the sample file, `count.dat`.

```
M = dlmread('count.dat')
```

```
M = 24×3

    11    11     9
     7    13    11
    14    17    20
    11    13     9
    43    51    69
    38    46    76
    61   132   186
    75   135   180
    38    88   115
    28    36    55
      ⋮
```

`dlmread` detects the delimiter from the file and returns a matrix.

Read File Containing Empty Delimited Fields

Write two matrices to a file, and then read the entire file using `dlmread`.

Export a matrix to a file named `myfile.txt`. Then, append an additional matrix to the file that is offset one row below the first.

```
X = magic(3);
dlmwrite('myfile.txt',[X*5 X/5],' ')
dlmwrite('myfile.txt',X,'-append', ...
   'roffset',1,'delimiter',' ')
```

View the file contents.

```
type myfile.txt
```

```
40 5 30 1.6 0.2 1.2
15 25 35 0.6 1 1.4
20 45 10 0.8 1.8 0.4
```

```
8 1 6
3 5 7
4 9 2
```

Read the entire file using `dlmread`.

```
M = dlmread('myfile.txt')
```

```
M = 6×6

   40.0000    5.0000   30.0000    1.6000    0.2000    1.2000
   15.0000   25.0000   35.0000    0.6000    1.0000    1.4000
   20.0000   45.0000   10.0000    0.8000    1.8000    0.4000
    8.0000    1.0000    6.0000         0         0         0
    3.0000    5.0000    7.0000         0         0         0
    4.0000    9.0000    2.0000         0         0         0
```

When `dlmread` imports a file containing nonrectangular data, it fills empty fields with zeros.

Read Delimited File Starting At Specific Row and Column Offset

Create a file named `dlmlist.txt` that contains column headers and space-delimited values.

```
test max min direction
10 27.7 12.4 12
11 26.9 13.5 18
12 27.4 16.9 31
13 25.1 12.7 29
```

Read the numeric values in the file. Specify a space delimiter, a row offset of 1, and a column offset of 0.

```
filename = 'dlmlist.txt';
M = dlmread(filename,' ',1,0)
```

```
M =

   10.0000   27.7000   12.4000   12.0000
```

```
   11.0000   26.9000   13.5000   18.0000
   12.0000   27.4000   16.9000   31.0000
   13.0000   25.1000   12.7000   29.0000
```

Read Specific Range from Delimited File

Create a file named `dlmlist.txt` that contains column headers and space-delimited values.

```
test max min direction
10 27.7 12.4 12
11 26.9 13.5 18
12 27.4 16.9 31
13 25.1 12.7 29
```

Read only the last two rows of numeric data from the file.

```
M = dlmread('dlmlist.txt',' ',[3 0 4 3])
```

```
M =

   12.0000   27.4000   16.9000   31.0000
   13.0000   25.1000   12.7000   29.0000
```

Input Arguments

`filename` — File name
character vector | string

File name, specified as a character vector or string.

Example: `'myFile.dat'` or `"myFile.dat"`

Data Types: `char` | `string`

`delimiter` — Field delimiter character
character vector | string

Field delimiter character, specified as a character vector or string. Use `'\t'` to specify a tab delimiter.

Example: `','` or `","`

Example: `'  '` or `"  "`

Data Types: `char` | `string`

R1 — Starting row offset
0 (default) | nonnegative integer

Starting row offset, specified as a nonnegative integer. The first row has an offset of 0.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

C1 — Starting column offset
0 (default) | nonnegative integer

Starting column offset, specified as a nonnegative integer. The first column has an offset of 0.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

R2 — Ending row offset
nonnegative integer

Ending row offset, specified as a nonnegative integer. The first row has an offset of 0.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

C2 — Ending column offset
nonnegative integer

Ending column offset, specified as a nonnegative integer. The first column has an offset of 0.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Tips

- Skip header rows or columns by specifying row and column offsets. All values in the file other than headers must be numeric.

Algorithms

`dlmread` fills empty delimited fields with zero. When the `dlmread` function reads data files with lines that end with a nonspace delimiter, such as a semicolon, it returns a matrix, `M`, that has an additional last column of zeros.

`dlmread` imports any complex number as a whole into a complex numeric field. This table shows valid forms for a complex number.

Form	Example
±<real>±<imag>i|j	5.7-3.1i
±<imag>i|j	-7j

Embedded white space in a complex number is invalid and `dlmread` regards it as a field delimiter.

Compatibility Considerations

`dlmread` is not recommended

Not recommended starting in R2019a

`dlmread` is not recommended. Use `readmatrix` instead. There are no plans to remove `dlmread`.

Starting in R2019a, use the `readmatrix` function to read a matrix from a delimited text file. The `readmatrix` function has these advantages over the `dlmread` function:

- Better cross-platform support and performance
- Automatic detection of data format and types
- Ability to use import options to control the data import process, including the handling of errors and missing data

This table shows typical usages of `dlmread` and how to update your code to use `readmatrix` instead.

Not Recommended	Recommended
`M = dlmread(filename)`	`M = readmatrix(filename)`

See Also

`dlmwrite` | `readmatrix` | `readtable` | `textscan` | `uiimport`

Topics

"Import Text Files"

Introduced before R2006a

dlmwrite

(Not recommended) Write matrix to ASCII-delimited file

Note `dlmwrite` is not recommended. Use `writematrix` instead. For more information, see Compatibility Considerations.

Syntax

```
dlmwrite(filename,M)
dlmwrite(filename,M,'-append')

dlmwrite( ___ ,Name,Value)

dlmwrite(filename,M,delimiter)
dlmwrite(filename,M,delimiter,row,col)
```

Description

`dlmwrite(filename,M)` writes numeric data in array `M` to an ASCII format file, `filename`, using the default delimiter (`,`) to separate array elements. If the file, `filename`, already exists, `dlmwrite` overwrites the file.

`dlmwrite(filename,M,'-append')` appends the data to the end of the existing file, `filename`.

`dlmwrite( ___ ,Name,Value)` additionally specifies delimiter, newline character, offset, and precision options using one or more name-value pair arguments.

`dlmwrite(filename,M,delimiter)` writes array `M` to the file, `filename`, using the specified delimiter, `delimiter`, to separate array elements.

`dlmwrite(filename,M,delimiter,row,col)` writes the array starting at the specified row and column `row` and `col`, in the destination file. Empty elements separated by `delimiter` fill the leading rows and columns.

Examples

Write Comma-Separated Data

Create an array of sample data, M.

```
M = magic(3);
```

Write matrix M to a file myFile.txt.

```
dlmwrite('myFile.txt',M)
```

View the data in the file.

```
type('myFile.txt')
```

```
8,1,6
3,5,7
4,9,2
```

Write Tab-Delimited Data and Specify Precision

Create an array of sample data, M.

```
M = magic(3)*pi
```

```
M = 3×3

   25.1327    3.1416   18.8496
    9.4248   15.7080   21.9911
   12.5664   28.2743    6.2832
```

Write matrix M to a file, 'myFile.txt', delimited by the tab character and using a precision of 3 significant digits.

```
dlmwrite('myFile.txt',M,'delimiter','\t','precision',3)
```

View the data in the file.

```
type('myFile.txt')
```

```
25.1    3.14    18.8
9.42    15.7    22
12.6    28.3    6.28
```

Write and Append Data to File

Create two arrays of sample numeric data.

```
M = magic(5);
N = magic(3);
```

Export matrix `M` to a file and use whitespace as the delimiter.

```
dlmwrite('myFile.txt',M,'delimiter',' ');
```

Append matrix `N` to the file, offset from the existing data by one row. Then, view the file.

```
dlmwrite('myFile.txt',N,'-append',...
'delimiter',' ','roffset',1)
type('myFile.txt')
```

```
17 24 1 8 15
23 5 7 14 16
4 6 13 20 22
10 12 19 21 3
11 18 25 2 9

8 1 6
3 5 7
4 9 2
```

Read the data in `'myFile.txt'` using `dlmread`.

```
dlmread('myFile.txt')
```

```
ans = 8×5

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
     8     1     6     0     0
```

```
     3     5     7     0     0
     4     9     2     0     0
```

When `dlmread` reads the two matrices from the file, it pads the smaller matrix with zeros.

Write Data and Specify Precision

Create an array of sample numeric data.

```
M = magic(3);
```

Export matrix `M` to a file using a precision of 6 decimal places.

```
dlmwrite('myFile.txt',M,'precision','%.6f');
```

View the data in the file.

```
type('myFile.txt')
```

```
8.000000,1.000000,6.000000
3.000000,5.000000,7.000000
4.000000,9.000000,2.000000
```

Input Arguments

`filename` — Name of file to write
character vector | string

Name of file to write, specified as a character vector or string.

Example: `'myFile.txt'` or `"myFile.txt"`

Data Types: `char` | `string`

`M` — Numeric data to write
matrix | cell array of numeric values

Numeric data to write, specified as a matrix or a cell array of numeric values with one value per cell.

Example: `[1,2,3;4,5,6]`

Example: `{1,2,3;4,5,6}`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `cell`
Complex Number Support: Yes

`delimiter` — Delimiter to separate array elements
`','` (default) | character vector | string

Delimiter to separate array elements, specified as a character vector or string, containing a single character or characters of a control sequence. Use `'\t'` to produce tab-delimited files.

Example: `';'` or `";"`

Example: `'\t'` or `"\t"`

Data Types: `char` | `string`

`row` — Row offset
0 (default) | scalar

Row offset, specified as a scalar. The row offset indicates the number of rows to skip before writing the numeric data. `row` is zero-based, so that `row = 0` instructs MATLAB to begin writing in the first row of the destination file. Skipped rows are populated with the specified delimiter.

`col` — Column offset
0 (default) | scalar

Column offset, specified as a scalar. The column offset indicates the number of columns to skip before writing the numeric data. `col` is zero-based, so that `col = 0` instructs MATLAB to begin writing in the first column of the destination file. Skipped columns are separated with the specified delimiter.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `dlmwrite('myFile.txt',M,'precision',4,'delimiter',' ')` writes the numeric values in array `M` with four significant digits and delimited using the whitespace character.

`delimiter` — Delimiter to separate array elements
`','` (default) | character vector | string

Delimiter to separate array elements, specified as the comma-separated pair consisting of `'delimiter'` and a character vector or string, containing a single character or characters of a control sequence. Use `'\t'` to produce tab-delimited files.

Example: `'delimiter',';'` or `'delimiter',";"`

Example: `'delimiter','\t'` or `'delimiter',"\t"`

Data Types: `char` | `string`

`roffset` — Row offset
0 (default) | scalar

Row offset, specified as the comma-separated pair consisting of `'roffset'` and a scalar. The row offset indicates the number of rows to skip before writing the numeric data. These rows are populated with the specified delimiter. When appending to an existing file, the new data is offset from the end of the existing data.

The row offset is zero-based, so that `'roffset',0` instructs MATLAB to begin writing in the first row of the destination file, which is the default. However, when appending to a file, `'roffset',0` instructs MATLAB to begin writing in the first row immediately following existing data.

Example: `'roffset',2`

`coffset` — Column offset
0 (default) | scalar

Column offset from the left side of the destination file, specified as the comma-separated pair consisting of `'coffset'` and a scalar. The column offset indicates the number of columns to skip before writing the numeric data. These columns are separated with the specified delimiter.

The column offset is zero-based, so that `'coffset',0` instructs MATLAB to begin writing in the first column of the destination file, which is the default.

Example: `'coffset',1`

`precision` — Numeric precision
5 (default) | scalar | C-style format specifier | character vector

Numeric precision to use in writing data to the file, specified as the comma-separated pair consisting of `'precision'` and a scalar or a C-style format specifier that begins with `%`, such as `'%10.5f'`. If the value of `precision` is a scalar, then it indicates the number of significant digits.

Example: `'precision',3`

Example: `'precision','%10.5f'`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char`

`newline` — Line terminator
`'pc'` | `'unix'`

Line terminator, specified as the comma-separated pair consisting of `'newline'` and either `'pc'` to use a carriage return/line feed (`CR/LF`), or `'unix'` to use a line feed (`LF`).

Example: `'newline','pc'`

Tips

- `dlmwrite` writes a file that spreadsheet programs can read. Alternatively, if your system has Excel for Windows installed, you can create a spreadsheet using `xlswrite`.

Compatibility Considerations

`dlmwrite` is not recommended

Not recommended starting in R2019a

`dlmwrite` is not recommended. Use `writematrix` instead. There are no plans to remove `dlmwrite`.

Starting in R2019a, use the `writematrix` function to write a matrix to a delimited text file. The `writematrix` function has better cross-platform support and performance over the `dlmwrite` function.

This table shows typical usages of `dlmwrite` and how to update your code to use `writematrix` instead.

Not Recommended	Recommended
`dlmwrite('mydata.txt',M)`	`writematrix(M,'mydata.txt')`

See Also

`dlmread` | `writematrix` | `writetable` | `xlswrite`

Introduced before R2006a

dmperm

Dulmage-Mendelsohn decomposition

Syntax

```
p = dmperm(A)
[p,q,r,s,cc,rr] = dmperm(A)
```

Description

`p = dmperm(A)` finds a vector `p` such that `p(j) = i` if column `j` is matched to row `i`, or zero if column `j` is unmatched. If `A` is a square matrix with full structural rank, `p` is a maximum matching row permutation and `A(p,:)` has a zero-free diagonal. The structural rank of `A` is `sprank(A) = sum(p>0)`.

`[p,q,r,s,cc,rr] = dmperm(A)` where `A` need not be square or full structural rank, finds the Dulmage-Mendelsohn decomposition of `A`. `p` and `q` are row and column permutation vectors, respectively, such that `A(p,q)` has a block upper triangular form. `r` and `s` are index vectors indicating the block boundaries for the fine decomposition. `cc` and `rr` are vectors of length five indicating the block boundaries of the coarse decomposition.

`C = A(p,q)` is split into a 4-by-4 set of coarse blocks:

```
A11 A12 A13 A14
0    0  A23 A24
0    0   0  A34
0    0   0  A44
```

where `A12`, `A23`, and `A34` are square with zero-free diagonals. The columns of `A11` are the unmatched columns, and the rows of `A44` are the unmatched rows. Any of these blocks can be empty. In the coarse decomposition, the `(i,j)th` block is `C(rr(i):rr(i+1)-1,cc(j):cc(j+1)-1)`. If `A` is square and structurally nonsingular, then `A23 = C`. That is, all of the other coarse blocks are `0`-by-`0`.

For a linear system:

- `[A11 A12]` is the underdetermined part of the system—it is always rectangular and with more columns than rows, or is `0`-by-`0`.
- `A23` is the well-determined part of the system—it is always square. The `A23` submatrix is further subdivided into block upper triangular form via the fine decomposition (the strongly connected components of `A23`).
- `[A34; A44]` is the overdetermined part of the system—it is always rectangular with more rows than columns, or is `0`-by-`0`.

The structural rank of `A` is `sprank(A) = rr(4)-1`, which is an upper bound on the numerical rank of `A`. `sprank(A) = rank(full(sprand(A)))` with probability 1 in exact arithmetic.

`C(r(i):r(i+1)-1,s(j):s(j+1)-1)` is the `(i,j)`th block of the fine decomposition. The `(1,1)` block is the rectangular block `[A11 A12]`, unless this block is `0`-by-`0`. The `(b,b)` block is the rectangular block `[A34 ; A44]`, unless this block is `0`-by-`0`, where `b = length(r)-1`. All other blocks of the form `C(r(i):r(i+1)-1,s(i):s(i+1)-1)` are diagonal blocks of `A23`, and are square with a zero-free diagonal.

Tips

- If `A` is a reducible matrix, the linear system $Ax = b$ can be solved by permuting `A` to a block upper triangular form, with irreducible diagonal blocks, and then performing block back-substitution. Only the diagonal blocks of the permuted matrix need to be factored, saving fill and arithmetic in the blocks above the diagonal.
- In graph theoretic terms, `dmperm` finds a maximum-size matching in the bipartite graph of `A`, and the diagonal blocks of `A(p,q)` correspond to the strong Hall components of that graph. The output of `dmperm` can also be used to find the connected or strongly connected components of an undirected or directed graph. For more information see Pothen and Fan [1].

References

[1] Pothen, Alex and Chin-Ju Fan “Computing the Block Triangular Form of a Sparse Matrix” *ACM Transactions on Mathematical Software* Vol 16, No. 4 Dec. 1990, pp. 303-324.

See Also

`matchpairs` | `sprank`

Introduced before R2006a

doc

Reference page in Help browser

Syntax

```
doc
doc name
```

Description

`doc` opens the Help browser. If the Help browser is already open, but not visible, then `doc` brings it to the foreground and opens a new tab.

`doc name` displays documentation for the functionality specified by `name`, such as a function, class, or block.

- If `name` corresponds to a MathWorks reference page, then `doc` displays the page in the Help browser. The `doc` command does not display third-party or custom HTML documentation.
- If `name` does not correspond to a reference page, then `doc` searches for help text in a file named *`name`*`.m` or *`name`*`.mlx`. If help text is available, `doc` displays it in the Help browser.
- If `name` does not correspond to a reference page and there is no associated help text, then `doc` searches the documentation for `name` and displays the search results in the Help browser.

Examples

Display Function Reference Page

Display the reference page for the `abs` function.

```
doc abs
```

Several products include different versions of `abs`. If your Help preferences support displaying documentation for those products, then the Help browser displays the MATLAB® `abs` reference page and a message with links to other versions of `abs`. This message appears at the top of the page.

Class and Method Reference Pages

Display the reference page for the `handle` class.

```
doc handle
```

Display the reference page for the `findobj` method in the `handle` class.

```
doc handle.findobj
```

Display the reference page for the `Map` class in the `containers` package.

```
doc containers.Map
```

Custom Class Pages

Display formatted help text for a custom class.

The class file `sads.m` is an example file that shows how to create a class. Add the example folder to the path and request the documentation for `sads`.

```
addpath(...
   fullfile(matlabroot,'help','techdoc','matlab_env',...
   'examples'))
doc sads
```

Display the help for the `steer` method of the `sads` class. Because the help text follows MATLAB conventions, MATLAB formats the display in the browser.

```
doc sads.steer
```

Input Arguments

name — Name of functionality
character vector

Name of functionality, such as function, class, or block, specified as a character vector. Alternatively, an operator symbol.

Some classes and other packaged items require that you specify the package name. Events, properties, and some methods require that you specify the class name. Separate the components of the name with periods, such as:

```
doc className.name
doc packageName.name
doc packageName.className.name
```

Methods for some classes are not accessible using the `doc` command; instead, use links on the class reference page.

Tips

- To access third-party or custom documentation, open the Help browser and navigate to the documentation home page. Then, at the bottom of the page, click **Supplemental Software**.

See Also

`help` | `web`

Topics

"Ways to Get Function Help"
"Add Help for Your Program"
"Display Custom Documentation"

Introduced before R2006a

docsearch

Help browser search

Syntax

```
docsearch
docsearch expression
```

Description

`docsearch` opens the Help browser and displays the documentation home page. If the Help browser is already open, but not visible, then `docsearch` brings it to the foreground.

`docsearch expression` searches the documentation for pages with words that match the specified expression and highlights them. To clear highlighting, press the **Esc** key. To search third-party or custom documentation, you must first run the `builddocsearchdb` command to build a search database for the additional help files.

Examples

Search for Single Words

Find all documentation pages that contain the word *plot*.

```
docsearch plot
```

Search for Multiple Words

Find documentation pages containing the words *plot* and *tools*.

```
docsearch plot tools
```

Expand the search to include variations of the word *plot*, such as *plotting* or *plots*, using a wildcard character (*).

```
docsearch plot* tools
```

Find pages containing either word, but not necessarily both words, using the `OR` operator.

```
docsearch plot OR tools
```

Narrow the search to pages that include an exact phrase by enclosing the phrase in quotation marks.

```
docsearch('"plot tools"')
```

Input Arguments

`expression` — Expression that defines search terms
character vector

Expression that defines search terms, specified as a character vector. Expressions can include:

- Quotation marks to specify exact phrases, such as `"plot tools"`
- Boolean operator keywords in uppercase (listed here in order of precedence): `NOT`, `OR`, `AND`
- Asterisk (*) wildcard characters, except at the beginning of a word or in an exact phrase (Searches require that at least two characters in the expression are *not* wildcard characters)

Tips

- To access third-party or custom documentation without the `docsearch` command, open the Help browser and navigate to the documentation home page. Then, at the bottom of the page, click **Supplemental Software**.

See Also

`builddocsearchdb` | `doc`

Topics

"Search Syntax and Tips"

Introduced before R2006a

dos

Execute DOS command and return output

Note For platform-independent code, use the `system` command.

Syntax

```
[status,cmdout] = dos(command)
```

Description

`[status,cmdout] = dos(command)` calls the operating system to execute the specified command and returns the output of the DOS command to `cmdout`.

For more examples, tips, and information about limitations, see `system`.

Examples

Display Operating System Command Status and Output

Display the current folder using the `cd` command. A `status` of zero indicates that the command completed successfully. MATLAB returns a character vector containing the current folder in `cmdout`.

```
command = 'cd';
[status,cmdout] = dos(command)
```

Input Arguments

command — MS-DOS® command
string | character vector

MS-DOS command, specified as a string or a character vector. The command can be a Windows UI program that opens a user interface, or a DOS console command that you typically run in a DOS command window. The command executes in a DOS shell, which might not be the shell from which you started MATLAB.

Example: `'dir'`

Output Arguments

`status` — Command exit status
0 | nonzero integer

Command exit status, returned as either `0` or a nonzero integer. When the command is successful, `status` is `0`. Otherwise, `status` is a nonzero integer.

- If `command` includes the ampersand character (&), then `status` is the exit status when `command` starts
- If `command` does not include the ampersand character (&), then `status` is the exit status upon `command` completion.

`cmdout` — Output of operating system command
character vector

Output of the operating system command, returned as a character vector. The system shell might not properly represent non-Unicode characters.

See Also

`computer` | `perl` | `system` | `winopen`

Topics

`!` (exclamation point)

Introduced before R2006a

dot

Dot product

Syntax

```
C = dot(A,B)
C = dot(A,B,dim)
```

Description

`C = dot(A,B)` returns the scalar dot product on page 1-3586 of `A` and `B`.

- If `A` and `B` are vectors, then they must have the same length.
- If `A` and `B` are matrices or multidimensional arrays, then they must have the same size. In this case, the `dot` function treats `A` and `B` as collections of vectors. The function calculates the dot product of corresponding vectors along the first array dimension whose size does not equal 1.

`C = dot(A,B,dim)` evaluates the dot product of `A` and `B` along dimension, `dim`. The `dim` input is a positive integer scalar.

Examples

Dot Product of Real Vectors

Create two simple, three-element vectors.

```
A = [4 -1 2];
B = [2 -2 -1];
```

Calculate the dot product of `A` and `B`.

```
C = dot(A,B)
```

```
C = 8
```

The result is 8 since

```
C = A(1)*B(1) + A(2)*B(2) + A(3)*B(3)
```

Dot Product of Complex Vectors

Create two complex vectors.

```
A = [1+i 1-i -1+i -1-i];
B = [3-4i 6-2i 1+2i 4+3i];
```

Calculate the dot product of A and B.

```
C = dot(A,B)
```

```
C = 1.0000 - 5.0000i
```

The result is a complex scalar since A and B are complex. In general, the dot product of two complex vectors is also complex. An exception is when you take the dot product of a complex vector with itself.

Find the inner product of A with itself.

```
D = dot(A,A)
```

```
D = 8
```

The result is a real scalar. The inner product of a vector with itself is related to the Euclidean length of the vector, `norm(A)`.

Dot Product of Matrices

Create two matrices.

```
A = [1 2 3;4 5 6;7 8 9];
B = [9 8 7;6 5 4;3 2 1];
```

Find the dot product of A and B.

```
C = dot(A,B)
```

```
C = 1×3

    54    57    54
```

The result, `C`, contains three separate dot products. `dot` treats the columns of `A` and `B` as vectors and calculates the dot product of corresponding columns. So, for example, `C(1) = 54` is the dot product of `A(:,1)` with `B(:,1)`.

Find the dot product of `A` and `B`, treating the *rows* as vectors.

```
D = dot(A,B,2)
```

```
D = 3×1

    46
    73
    46
```

In this case, `D(1) = 46` is the dot product of `A(1,:)` with `B(1,:)`.

Dot Product of Multidimensional Arrays

Create two multidimensional arrays.

```
A = cat(3,[1 1;1 1],[2 3;4 5],[6 7;8 9])
```

```
A =
A(:,:,1) =

     1     1
     1     1


A(:,:,2) =

     2     3
     4     5


A(:,:,3) =
```

```
     6     7
     8     9
```

```
B = cat(3,[2 2;2 2],[10 11;12 13],[14 15; 16 17])
```

```
B =
B(:,:,1) =

     2     2
     2     2


B(:,:,2) =

    10    11
    12    13


B(:,:,3) =

    14    15
    16    17
```

Calculate the dot product of `A` and `B` along the third dimension (`dim = 3`).

```
C = dot(A,B,3)
```

```
C = 2×2

   106   140
   178   220
```

The result, `C`, contains four separate dot products. The first dot product, `C(1,1) = 106`, is equal to the dot product of `A(1,1,:)` with `B(1,1,:)`.

Input Arguments

`A,B` — Input arrays
numeric arrays

Input arrays, specified as numeric arrays.

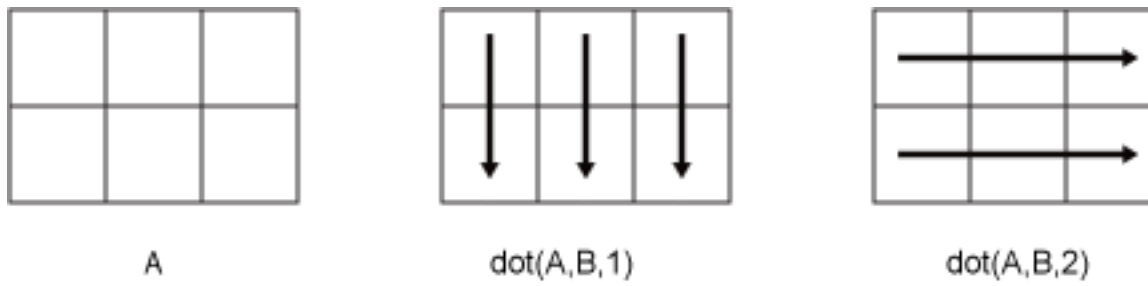
Data Types: `single` | `double`
Complex Number Support: Yes

`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, the default is the first array dimension whose size does not equal 1.

Consider two 2-D input arrays, `A` and `B`:

- `dot(A,B,1)` treats the columns of `A` and `B` as vectors and returns the dot products of corresponding columns.
- `dot(A,B,2)` treats the rows of `A` and `B` as vectors and returns the dot products of corresponding rows.



`dot` returns `conj(A).*B` if `dim` is greater than `ndims(A)`.

More About

Scalar Dot Product

The scalar dot product of two real vectors of length n is equal to

$$u \cdot v = \sum_{i=1}^{n} u_i v_i = u_1 v_1 + u_2 v_2 + \ldots + u_n v_n \ .$$

This relation is commutative for real vectors, such that `dot(u,v)` equals `dot(v,u)`. If the dot product is equal to zero, then *u* and *v* are perpendicular.

For complex vectors, the dot product involves a complex conjugate. This ensures that the inner product of any vector with itself is real and positive definite.

$$u \cdot v = \sum_{i=1}^{n} \bar{u}_i v_i \ .$$

Unlike the relation for real vectors, the complex relation is not commutative, so `dot(u,v)` equals `conj(dot(v,u))`.

Algorithms

- When inputs `A` and `B` are real or complex vectors, the `dot` function treats them as column vectors and `dot(A,B)` is the same as `sum(conj(A).*B)`.
- When the inputs are matrices or multidimensional arrays, the `dim` argument determines which dimension the `sum` function operates on. In this case, `dot(A,B)` is the same as `sum(conj(A).*B,dim)`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitation:

For the syntax `dot(A,B)`, the arrays `A` and `B` must have the same size, even if they are vectors.

For more information, see “Tall Arrays for Out-of-Memory Data”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`conj` | `cross` | `norm` | `sum`

Introduced before R2006a

double

Double-precision arrays

Description

`double` is the default numeric data type (class) in MATLAB, providing sufficient precision for most computational tasks. Numeric variables are automatically stored as 64-bit (8-byte) double-precision floating-point values. For example:

```
x = 10;
whos x
```

```
  Name      Size            Bytes  Class     Attributes

  x         1x1                 8  double
```

MATLAB constructs the `double` data type according to IEEE Standard 754 for double precision. The range for a negative number of type `double` is between -1.79769 x 10^{308} and -2.22507 x 10^{-308}, and the range for positive numbers is between 2.22507 x 10^{-308} and 1.79769 x 10^{308}.

For more information on double- and single-precision floating-point values, see “Floating-Point Numbers”.

Creation

You create a double-precision array automatically when you assign a numeric scalar or array to a variable, such as `A = [1 2 3; 4 5 6]`. The variable `A` has type `double`. For more information on creating and combining arrays, see “Creating, Concatenating, and Expanding Matrices”. In addition, operations on double-precision variables and functions with double-precision input typically return double-precision values, such as `+` or `sin`.

If you have an array of a different data type, such as `single` or `int8`, then you can convert that array to double precision using the `double` function, which then stores the array with more precision for further computations.

Syntax

```
Y = double(X)
```

Description

`Y = double(X)` converts the values in `X` to double precision.

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string`

Examples

Create Double-Precision Variable

By default, numbers in MATLAB are of the data type `double`. You can use the `class` function to verify a variable's type.

```
x = 100;
xtype = class(x)
```

```
xtype =
'double'
```

Use the `double` function to convert variables that are not double precision to type `double`.

```
y = true
```

```
y = logical
   1
```

```
ydouble = double(y);
ynewtype = class(ydouble)
```

```
ynewtype =
'double'
```

Double-Precision Versus Single-Precision

Compare the ranges of numeric values in double-precision to the ranges for single-precision.

Use the `realmin` and `realmax` functions to display the minimum and maximum positives values that can be represented in double precision.

```
doublemin = realmin('double')
```

```
doublemin = 2.2251e-308
```

```
doublemax = realmax('double')
```

```
doublemax = 1.7977e+308
```

Now display the minimum and maximum positive values that can be represented in single precision. The range of values is smaller compared to double-precision, but requires less memory.

```
singlemin = realmin('single')
```

```
singlemin = single
    1.1755e-38
```

```
singlemax = realmax('single')
```

```
singlemax = single
    3.4028e+38
```

The `eps` function returns a measure of how close numbers can be in double-precision versus single-precision. Display the distance from the number 1.0 to the next larger double-precision number.

```
doubleeps = eps('double')
```

```
doubleeps = 2.2204e-16
```

Now display the distance from 1.0 to the next larger single-precision number. Double-precision values are closer to each other, since you can represent more of them.

```
singleeps = eps('single')
```

```
singleeps = single
    1.1921e-07
```

Tips

- When you are creating a class, overload `double` when it makes sense to convert an object of that class to a double-precision value.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For string inputs with misplaced commas (commas that are not used as thousands separators), generated code results can differ from MATLAB results.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cast` | `eps` | `realmax` | `realmin` | `single` | `str2double` | `typecast`

Topics

“Floating-Point Numbers”
“Identifying Numeric Classes”

Introduced before R2006a

dragrect

Drag rectangles with mouse

Syntax

```
[finalrect] = dragrect(initialrect)
[finalrect] = dragrect(initialrect,stepsize)
```

Description

`[finalrect] = dragrect(initialrect)` tracks one or more rectangles anywhere on the screen. The n-by-4 matrix `initialrect` defines the rectangles. Each row of `initialrect` must contain the initial rectangle position as `[left bottom width height]` values. `dragrect` returns the final position of the rectangles in `finalrect`.

`[finalrect] = dragrect(initialrect,stepsize)` moves the rectangles in increments of `stepsize`. The lower left corner of the first rectangle is constrained to a grid of size equal to `stepsize` starting at the lower left corner of the figure, and all other rectangles maintain their original offset from the first rectangle.

`[finalrect] = dragrect(...)` returns the final positions of the rectangles when the mouse button is released. The default step size is `1`.

Examples

Drag a rectangle with dimensions of 50-by-100 pixels on page 1-3595.

```
waitforbuttonpress
point1 = get(gcf,'CurrentPoint') % button down detected
rect = [point1(1,1) point1(1,2) 50 100]
[r2] = dragrect(rect)
```

More About

Pixels

Distances in pixels are independent of your system resolution on Windows and Macintosh systems:

- On Windows systems, a pixel is 1/96th of an inch.
- On Macintosh systems, a pixel is 1/72nd of an inch.

On Linux systems, the size of a pixel is determined by your system resolution.

Tips

`dragrect` returns immediately if a mouse button is not currently pressed. Use `dragrect` in a `ButtonDownFcn`, or from the command line in conjunction with `waitforbuttonpress`, to ensure that the mouse button is down when `dragrect` is called. `dragrect` returns when you release the mouse button.

If the drag ends over a figure window, the positions of the rectangles are returned in that figure's coordinate system. If the drag ends over a part of the screen not contained within a figure window, the rectangles are returned in the coordinate system of the figure over which the drag began.

Note You cannot use normalized figure units with `dragrect`.

See Also

`rbbox` | `waitforbuttonpress`

Introduced before R2006a

drawnow

Update figures and process callbacks

Syntax

```
drawnow
drawnow limitrate
drawnow nocallbacks
drawnow limitrate nocallbacks

drawnow update
drawnow expose
```

Description

`drawnow` updates figures and processes any pending callbacks. Use this command if you modify graphics objects and want to see the updates on the screen immediately.

`drawnow limitrate` limits the number of updates to 20 frames per second. If it has been fewer than 50 milliseconds since the last update, or if the graphics renderer is busy with the previous change, then `drawnow` discards the new updates. Use this command if you are updating graphics objects in a loop and do not need to see every update on the screen. Skipping updates can create faster animations. Pending callbacks are processed, so you can interact with figures during animations.

`drawnow nocallbacks` defers callbacks, such as the `ButtonDownFcn` callback, until the next full `drawnow` command. Use this option if you want to prevent callbacks from interrupting your code. Deferring callbacks temporarily disables figure interactions, such as mouse clicks or resizing the figure. Deferring callbacks does not affect animation speed.

`drawnow limitrate nocallbacks` limits the number of updates to 20 frames per second and skips updates if the renderer is busy. This syntax also prevents callbacks from interrupting your code, which temporarily disables figure interactions.

`drawnow update` skips updates if the renderer is busy and defers callbacks. This syntax is not recommended. Use the `limitrate` option instead.

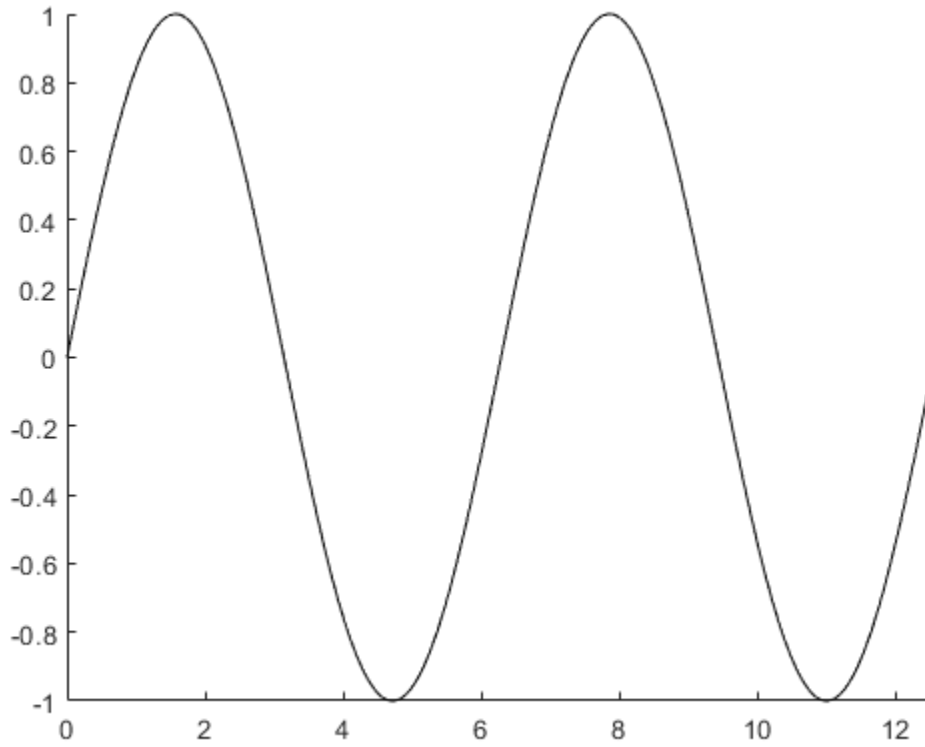
`drawnow expose` updates figures, but defers callbacks. This syntax is not recommended. Use the `nocallbacks` option instead.

Examples

Create Animation of Streaming Data

Create an animation of a line growing as it accumulates 2,000 data points. Use `drawnow` to display the changes on the screen after each iteration through the loop.

```
h = animatedline;
axis([0 4*pi -1 1])
x = linspace(0,4*pi,2000);

for k = 1:length(x)
    y = sin(x(k));
    addpoints(h,x(k),y);
    drawnow
end
```

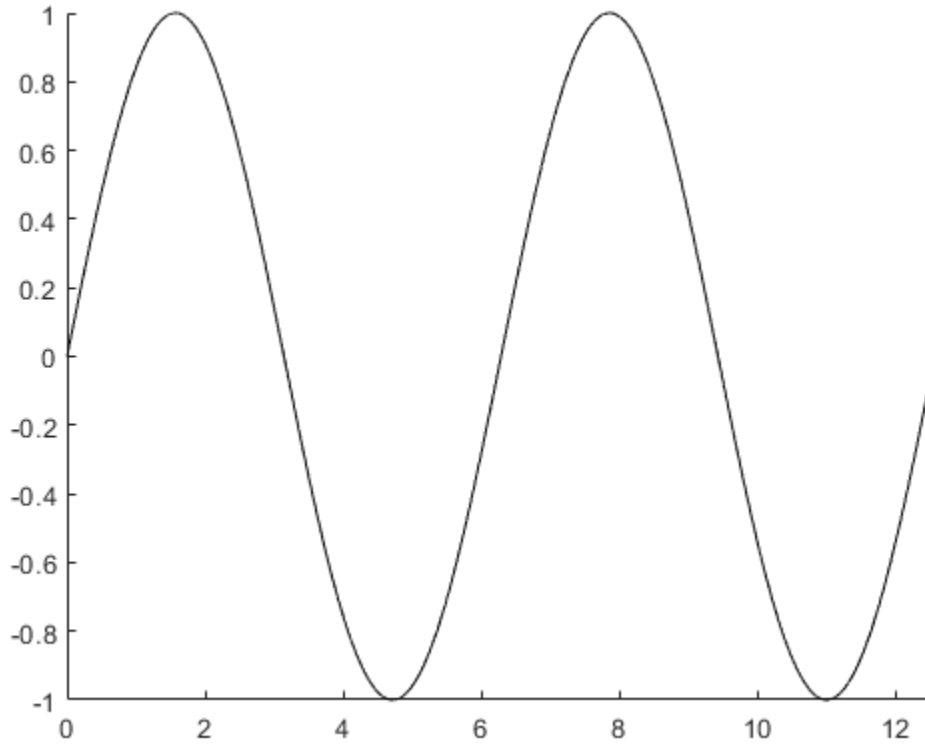


Skip Updates for Faster Animation

Create an animation of a line growing as it accumulates 10,000 points. Since there are 10,000 points, drawing each update on the screen is slow. Create a faster, smooth animation by limiting the number of updates using `drawnow limitrate`. Then, display the final updates on the screen by calling `drawnow` after the loop ends.

```
h = animatedline;
axis([0 4*pi -1 1])
x = linspace(0,4*pi,10000);

for k = 1:length(x)
```

```
    y = sin(x(k));
    addpoints(h,x(k),y);
    drawnow limitrate
end
drawnow
```

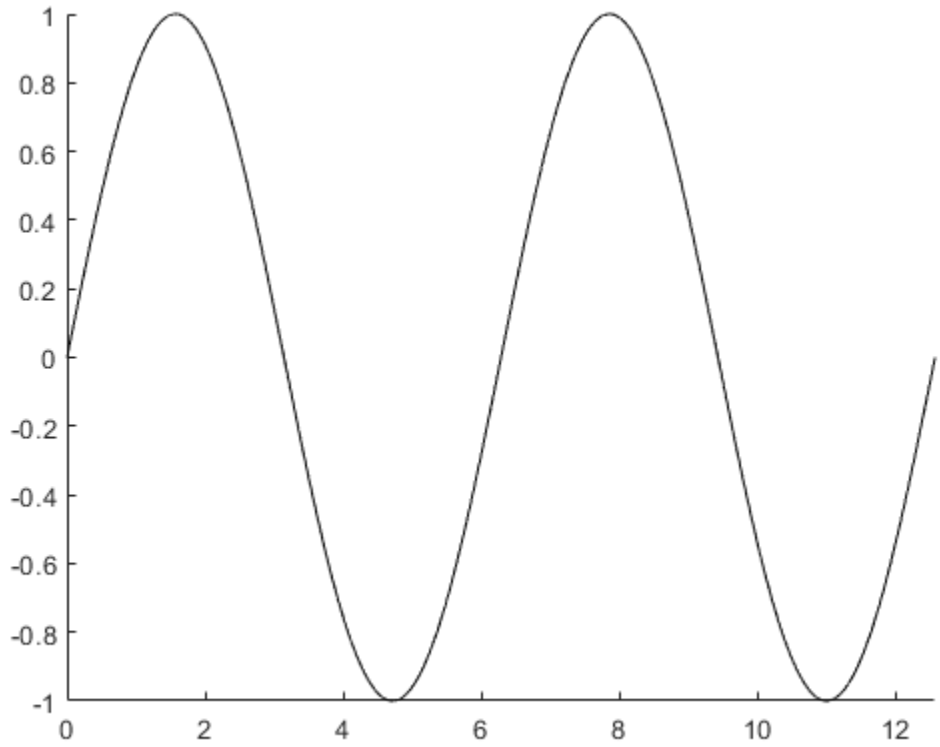


Precompute Data, Then Create Animation

Compute all the data before the animation loop.

```
h = animatedline;
axis([0 4*pi -1 1])
```

```
x = linspace(0,4*pi,10000);
y = sin(x);

for k = 1:length(x)
    addpoints(h,x(k),y(k));
    drawnow limitrate
end
drawnow
```



If you have long computations, precomputing the data can improve performance. Precomputing minimizes the computation time by letting the computation run without

interruptions. Additionally, it helps ensure a smooth animation by focusing on only graphics code in the animation loop.

More About

Actions Equivalent to `drawnow`

These actions are equivalent to calling a full `drawnow` command:

- Returning to the MATLAB prompt.
- Using the `getframe`, `pause`, and `waitfor` functions.
- Using functions that wait for user input, such as `waitforbuttonpress`, `input`, `keyboard`, or `ginput`.

Tips

- The `nocallbacks` option always adds interrupting callbacks to the queue. If you want to discard interrupting callbacks, then use the `Interruptible` and `BusyAction` properties instead.

See Also

`pause` | `refreshdata` | `waitfor`

Topics

"Callbacks — Programmed Response to User Action"

Introduced before R2006a

dsearchn

Nearest point search

Syntax

```
k = dsearchn(P,PQ)
k = dsearchn(P,T,PQ)
k = dsearchn(P,T,PQ,outind)
[k,dist] = dsearchn( ___ )
```

Description

`k = dsearchn(P,PQ)` returns the indices of the closest points in `P` to the query points in `PQ` measured in Euclidean distance.

`k = dsearchn(P,T,PQ)` returns the indices of the closest points in `P` by using the Delaunay triangulation `T`, where `T = delaunayn(P)`. Providing `T` can improve search performance when `PQ` contains a large number of points.

`k = dsearchn(P,T,PQ,outind)` returns the indices of the closest points in `P`, but assigns an index value of `outind` for query points that are outside of the convex hull of `P`. For example, `desearchn(P,T,PQ,Inf)` returns an index value of `Inf` for query points outside of the convex hull.

`[k,dist] = dsearchn( ___ )` also returns the distance from each point in `P` to the corresponding query point in `PQ`.

Examples

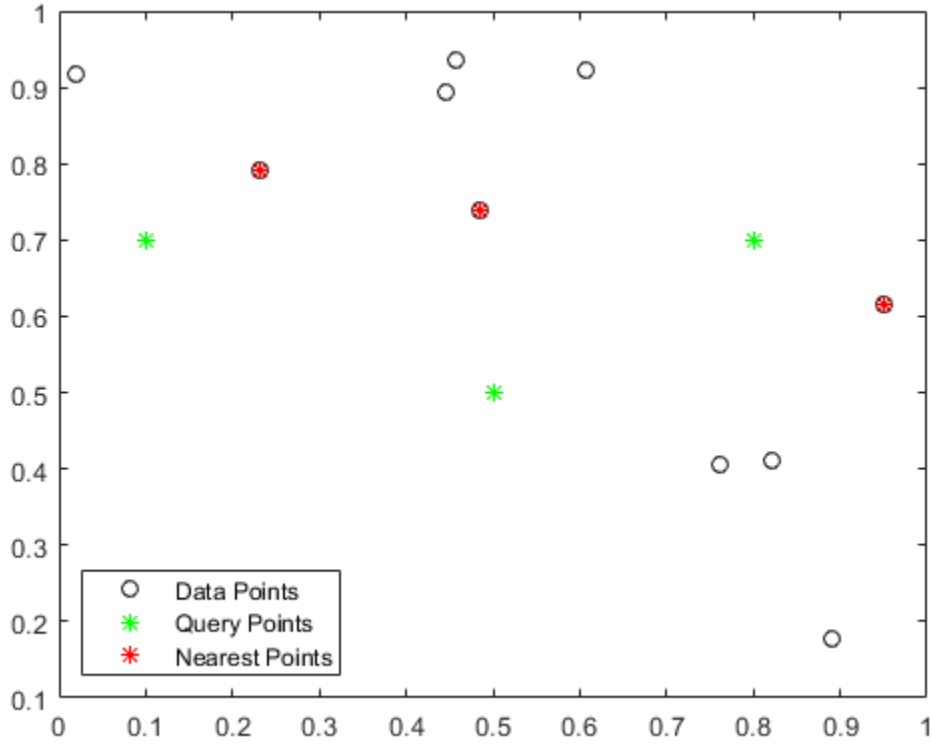
Nearest 2-D Points

Create a matrix `P` of 2-D data points and a matrix `PQ` of 2-D query points. Find the nearest data point to each query point, and compute the corresponding distances.

```
P = gallery('uniformdata',[10 2],0);
PQ = [0.5 0.5; 0.1 0.7; 0.8 0.7];
[k,dist] = dsearchn(P,PQ);
```

Plot the data points and query points, and highlight the data point nearest to each query point.

```
plot(P(:,1),P(:,2),'ko')
hold on
plot(PQ(:,1),PQ(:,2),'*g')
hold on
plot(P(k,1),P(k,2),'*r')
legend('Data Points','Query Points','Nearest Points','Location','sw')
```



Display the distances.

```
dist
```

```
dist = 3×1

    0.2386
    0.1602
    0.1723
```

Input Arguments

P — Points
matrix

Points, specified as an *m*-by-*n* matrix containing *m* points of dimension *n*. For example, `P = [0 0 0; 1 1 1]` represents the 3-D coordinates for the points (0,0,0) and (1,1,1).

PQ — Query points
matrix

Query points, specified as an *r*-by-*n* matrix containing *r* points of dimension *n*. For example, the 2-by-3 matrix `PQ = [-1 -1 -1; 2 2 2]` represents the 3-D coordinates for the two query points (-1,-1,-1) and (2,2,2).

The number of columns in `PQ` must match the number of columns in `P`.

T — Delaunay triangulation
matrix

Delaunay triangulation, specified as a matrix returned by the `delaunayn` function.

`outind` — Outside index value
scalar

Outside index value, specified as a scalar index value for query points outside of the convex hull.

If `outval` is specified as `[]`, then the output `k` is equivalent to the syntax `k = dsearchn(P,T,PQ)`.

Output Arguments

k — Indices
column vector

Indices, returned as a column vector containing the indices of the data points closest to the query points. The length of k is equal to the number of query points.

dist — Distance
column vector

Distance, returned as a column vector containing the Euclidean distances between each query point and the closest input point. The length of `dist` is equal to the number of query points.

Algorithms

`dsearchn` is based on Qhull [1]. For information about Qhull, see `http://www.qhull.org/`. For copyright information, see `http://www.qhull.org/COPYING.txt`.

References

[1] Barber, C.B., D.P. Dobkin, and H.T. Huhdanpaa. "The Quickhull Algorithm for Convex Hulls." *ACM Transactions on Mathematical Software*, Vol. 22, No. 4, Dec. 1996, p 469–483.

See Also

`delaunayTriangulation` | `delaunayn`

Introduced before R2006a

matlab.io.datastore.DsFileReader class

Package: `matlab.io.datastore`

File-reader object for files in a datastore

Description

The `DsFileReader` object enables low-level file reading access for files in your datastore.

Construction

`fr = matlab.io.datastore.DsFileReader(filename)` returns a `DsFileReader` object for read access to the file specified by `filename`.

`fr = matlab.io.datastore.DsFileReader(filename,'TextEncoding', encoding)` specifies the character encoding scheme associated with the file. Additionally, specifying `encoding` sets the `TextEncoding` property of the `DsFileReader` object.

Input Arguments

`filename` — File name
character vector | string scalar

File name, including the file extension, specified as a character vector or a string scalar. If the file is not in the current folder, `filename` must include a full or a relative path.

The `Name` property of the `DsFileReader` object stores the file name.

Example: `'myFile.txt'`

Data Types: `char` | `string`

`encoding` — Character encoding scheme
`'UTF-8'` (default) | `'ISO-8859-1'` | `'windows-1251'` | `'windows-1252'` | ...

Character encoding scheme associated with the file, specified as the name of a standard, character-encoding scheme listed in this table.

'Big5'	'ISO-8859-1'	'windows-847'
'Big5-HKSCS'	'ISO-8859-2'	'windows-949'
'CP949'	'ISO-8859-3'	'windows-1250'
'EUC-KR'	'ISO-8859-4'	'windows-1251'
'EUC-JP'	'ISO-8859-5'	'windows-1252'
'EUC-TW'	'ISO-8859-6'	'windows-1253'
'GB18030'	'ISO-8859-7'	'windows-1254'
'GB2312'	'ISO-8859-8'	'windows-1255'
'GBK'	'ISO-8859-9'	'windows-1256'
'IBM866'	'ISO-8859-11'	'windows-1257'
'KOI8-R'	'ISO-8859-13'	'windows-1258'
'KOI8-U'	'ISO-8859-15'	'US-ASCII'
	'Macintosh'	'UTF-8'
	'Shift_JIS'	

The `TextEncoding` property of the `DsFileReader` object stores the value specified in `encoding`.

Example: `'Shift_JIS'`

Data Types: `char` | `string`

Properties

`Name` — File name

character vector | string scalar

File name, specified as a character vector or string scalar.

Example: `fr.Name` returns the file name.

Data Types: `char` | `string`

`Size` — File size

numeric scalar

File size in bytes, returned as a numeric scalar integer.

Example: `fr.Size`

Data Types: `double`

`TextEncoding` — Character encoding scheme
`'UTF-8'` (default) | `'ISO-8859-1'` | `'windows-1251'` | `'windows-1252'` | ...

Character encoding scheme associated with the file, specified as the name of a standard, character-encoding scheme. To set the value for the `TextEncoding` property, see the description of the `encoding` input argument.

Example: `'TextEncoding','Shift_JIS'`

`Position` — Position pointer location
integer

Position pointer location in the file, specified as an integer. The position pointer is a zero-based integer that tracks the number of bytes from the beginning of the file.

If a file has `n` bytes of data, then those `n` bytes are in positions `0` through `n-1`.

You can set the `Position` property using the `seek` method. Calls to the `read` method begin reading the file from the location indicated by the `Position` property. When reading a file iteratively, the `read` method automatically updates the position pointer. Subsequent calls to the `read` method begin reading from the end position of the previous read operation.

Data Types: `double`

Methods

hasdata	Determine if data is available to read
read	Read bytes from file
seek	Seek to a position in the file

Examples

Read File Portion Specified by Starting Position and Size

Create a file-reader object for a file, seek to the desired starting position, and read a portion of the file.

Create a `DsFileReader` object for `airlinesmall.csv`.

```
fr = matlab.io.datastore.DsFileReader('airlinesmall.csv');
```

The `airlinesmall.csv` file has variable names at the beginning of the file. The variable names line ends at the position marked by `299` bytes. To get past the variable names line, use the `seek` method to move the read pointer to the starting position.

```
seek(fr,299,'RespectTextEncoding',true);
```

Read the first `1000` characters.

```
if hasdata(fr)
   d = read(fr,1000,'SizeMethod','OutputSize','OutputType','char');
end
```

See Also

`matlab.io.Datastore` | `matlab.io.datastore.DsFileSet` | `matlab.io.datastore.HadoopLocationBased` | `matlab.io.datastore.Partitionable`

Introduced in R2017b

hasdata

Class: `matlab.io.datastore.DsFileReader`
Package: `matlab.io.datastore`

Determine if data is available to read

Syntax

```
tf = hasdata(fr)
```

Description

`tf = hasdata(fr)` returns logical `1` (`true`) if there is data available to read from the file-reader object specified by `fr`. Otherwise, `hasdata` returns logical `0` (`false`).

Input Arguments

`fr` — File-reader object
`matlab.io.datastore.DsFileReader` object

File-reader object, specified as a `matlab.io.datastore.DsFileReader` object. To create a `DsFileReader` object, see `matlab.io.datastore.DsFileReader`.

Examples

Determine if File-Reader Object Has Data to Read

Create a file-reader object for a file, check if the file has data to read, and then read the data.

Create a `DsFileReader` object for `airlinesmall.csv`.

```
fr = matlab.io.datastore.DsFileReader('airlinesmall.csv');
```

Check if the file has data to read using the `hasdata` method. Then, read the first `1000` characters.

```
if hasdata(fr)
   d = read(fr,1000,'SizeMethod','OutputSize','OutputType','char');
end
```

See Also

`matlab.io.datastore.DsFileReader` | `read` | `seek`

Introduced in R2017b

read

Class: `matlab.io.datastore.DsFileReader`
Package: `matlab.io.datastore`

Read bytes from file

Syntax

```
A = read(fr,size)
A = read(fr,size,Name,Value)
[A,count] = read( ___ )
```

Description

`A = read(fr,size)` returns data, from the file represented by the file-reader object `fr`. The number of bytes specified in `size` determines the amount of data that is read.

`A = read(fr,size,Name,Value)` specifies additional parameters using one or more name-value pair arguments. For example, you can specify the output type from the read operation to be `char` by specifying `'OutputType','char'`.

`[A,count] = read( ___ )` returns a count of the number of bytes of data that were actually read by the `read` method.

Input Arguments

`fr` — File-reader object
`matlab.io.datastore.DsFileReader` object

File-reader object, specified as a `matlab.io.datastore.DsFileReader` object. To create a `DsFileReader` object, see `matlab.io.datastore.DsFileReader`.

`size` — Size of data to read
integer

Size of data to read, specified as an integer that represents the number of bytes to read.

Example: `read(fr,20)`

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'OutputType','uint8'`

`OutputType` — Output data type
'uint8' (default) | 'int8' | 'int16' | 'int32' | 'int64' | 'uint16' | 'uint32' | 'uint64' | 'single' | 'double' | 'char'

Output data type, specified as the comma-separated pair consisting of `'OutputType'` and a character vector or string scalar containing one of these values: `'uint8'`, `'int8'`, `'int16'`, `'int32'`, `'int64'`, `'uint16'`, `'uint32'`, `'uint64'`, `'single'`, `'double'`, or `'char'`.

Example: `'OutputType','uint8'`

Data Types: `char` | `string`

`SizeMethod` — Interpret size input
`'NumBytes'` (default) | `'OutputSize'`

Interpret size input, specified as the comma-separated pair consisting of `'SizeMethod'` and one of these values:

- `'NumBytes'` - Interpret the `size` input argument as the number of bytes to read from the file.
- `'OutputSize'` - Interpret the `size` input argument as the size of the output `A` from the read method.

Example: `'SizeMethod','OutputSize'`

Data Types: `char` | `string`

Output Arguments

A — Output data
array

Output data, returned as an array.

count — Number of bytes read
integer

Number of bytes read, returned as a numeric scalar integer.

- If the `'SizeMethod'` property is unspecified or set to `'NumBytes'`, then `count` is the number of bytes read.
- If the `'SizeMethod'` property is set to `'OutputSize'`, then `count` is equal to `size(A)`.

Data Types: `double`

Examples

Read Portion of File Specified by Starting Position and Size

Create a file-reader object for a file, seek to the desired starting position, and read a portion of the file.

Create a `DsFileReader` object for `airlinesmall.csv`.

```
fr = matlab.io.datastore.DsFileReader('airlinesmall.csv');
```

The `airlinesmall.csv` file has variable names at the beginning of the file. The variable names line ends at the position marked by `299` bytes. To get past the variable names line, use the `seek` method to move the read pointer to the starting position.

```
seek(fr,299,'RespectTextEncoding',true);
```

Check if the file has data to read using the `hasdata` method. The read method reads `1000` bytes from the file and interprets them as characters.

```
if hasdata(fr)
   [d,count] = read(fr,1000,'OutputType','char');
end
```

Read enough bytes from the file to fill `1000` characters by setting the `SizeMethod` parameter to `OutputSize`.

```
if hasdata(fr)
    [d,count] = read(fr,1000,'SizeMethod','OutputSize',...
                                        'OutputType','char');
end
```

See Also

`hasdata` | `matlab.io.datastore.DsFileReader` | `seek`

Introduced in R2017b

seek

Class: `matlab.io.datastore.DsFileReader`
Package: `matlab.io.datastore`

Seek to a position in the file

Syntax

```
numBytes = seek(fr,n)
numBytes = seek(fr,n,Name,Value)
```

Description

`numBytes = seek(fr,n)` moves the file position indicator by `n` bytes past current position in the file specified by the `fr` object. `seek` returns the actual number of bytes by which the position indicator was moved.

`numBytes = seek(fr,n,Name,Value)` specifies additional parameters using one or more name-value pair arguments. For example, you can specify the starting position of the `seek` operation by specifying `'Origin','start-of-file'`.

Input Arguments

`fr` — File-reader object
`matlab.io.datastore.DsFileReader` object

File-reader object, specified as a `matlab.io.datastore.DsFileReader` object. To create a `DsFileReader` object, see `matlab.io.datastore.DsFileReader`.

`n` — Number of bytes
integer

Number of bytes, specified as an integer. The `seek` method moves the file position indicator `n` bytes from current position in the specified file. If `n` is negative, `seek` will move the position indicator backwards in the file.

Example: `seek(fr,20)`

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'RespectTextEncoding',true`

`RespectTextEncoding` — Respect character boundaries
false (default) | true

Respect character boundaries, specified as the comma-separated pair consisting of `'RespectTextEncoding'` followed by `true` or `false`.

- `true` - Respect character boundaries of multibyte characracters.
- `false` - Do not respect character boundaries of multibyte characters.

Example: `'RespectTextEncoding',true`

Data Types: `logical`

`Origin` — Starting position
`'currentposition'` (default) | `'start-of-file'` | `'end-of-file'`

Starting position, specified as the comma-separated pair consisting of `'Origin'` followed by one of these values.

- `'currentposition'` - Start the `seek` operation from the current position indicator in the file.
- `'start-of-file'` - Starts the `seek` operation from position zero.
- `'end-of-file'` - Starts the `seek` operation from the end of the file.

Example: `'Origin','start-of-file'`

Data Types: `char` | `string`

Examples

Seek to Position in File and Read

Create a file-reader object for a file, seek to the desired starting position, and read a portion of the file.

Create a `DsFileReader` object for `airlinesmall.csv`.

```
fr = matlab.io.datastore.DsFileReader('airlinesmall.csv');
```

The `airlinesmall.csv` file has variable names at the beginning of the file. The variable names line ends at the position marked by `299` bytes. To get past the variable names line, use the `seek` method to move the read pointer to the starting position.

```
seek(fr,299,'RespectTextEncoding',true);
```

Read the first `1000` characters.

```
if hasdata(fr)
   d = read(fr,1000,'SizeMethod','OutputSize','OutputType','char');
end
```

See Also

`hasdata` | `matlab.io.datastore.DsFileReader` | `read`

Introduced in R2017b

matlab.io.datastore.DsFileSet class

Package: `matlab.io.datastore`

File-set object for collection of files in datastore

Description

The `DsFileSet` object helps you manage the iterative processing of large collections of files. Use the `DsFileSet` object together with the `DsFileReader` object to manage and read files from your datastore.

Construction

`fs = matlab.io.datastore.DsFileSet(location)` returns a `DsFileSet` object for a collection of files based on the specified `location`.

`fs = matlab.io.datastore.DsFileSet(location,Name,Value)` specifies additional parameters for the `DsFileSet` object using one or more name-value pair arguments. `Name` also can be a property name, and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

`location` — Files or folders to include
character vector | cell array of character vectors | string | struct

Files or folders to include in the file-set object, specified as a character vector, cell array of character vectors, string, or a struct. If the files are not in the current folder, then `location` must be full or relative paths. Files within subfolders of the specified folder are not automatically included in the file-set object.

Typically for a Hadoop workflow, when you specify `location` as a struct, it must contain the fields `FileName`, `Offset`, and `Size`. This requirement enables you to use the `location` argument directly with the `initializeDatastore` method of the

`matlab.io.datastore.HadoopLocationBased` class. For an example, see "Add Support for Hadoop".

You can use the wildcard character (*) when specifying `location`. Specifying this character includes all matching files or all files in the matching folders in the file-set object.

If the files are not available locally, then the full path of the files or folders must be a uniform resource locator (URL), such as `hdfs://hostname:portnumber/path_to_file`.

Data Types: `char` | `cell` | `string` | `struct`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'FileExtensions',{'.jpg','.tif'}` includes all files with a `.jpg` or `.tif` extension in the `FileSet` object.

`FileExtensions` — File extensions
character vector | cell array of character vectors | string

File extensions, specified as the comma-separated pair consisting of `'FileExtensions'` and a character vector, cell array of character vectors, or string. You can use the empty quotes `''` to represent files without extensions.

If `'FileExtensions'` is not specified, then `DsFileSet` automatically includes all file extensions.

Example: `'FileExtensions','.jpg'`

Example: `'FileExtensions',{'.txt','.csv'}`

Data Types: `char` | `cell` | `string`

`IncludeSubfolders` — Subfolder inclusion flag
`false` (default) | `true`

Subfolder inclusion flag, specified as the comma-separated pair consisting of `'IncludeSubfolders'` and `true` or `false`. Specify `true` to include all files and subfolders within each folder or `false` to include only the files within each folder.

Example: `'IncludeSubfolders',true`

Data Types: `logical` | `double`

`AlternateFileSystemRoots` — Alternate file system root paths
string vector | cell array

Alternate file system root paths, specified as the comma-separated pair consisting of `'AlternateFileSystemRoots'` and a string vector or a cell array. Use `'AlternateFileSystemRoots'` when you create a datastore on a local machine, but need to access and process the data on another machine (possibly of a different operating system). Also, when processing data using the Parallel Computing Toolbox and the MATLAB Parallel Server, and the data is stored on your local machines with a copy of the data available on different platform cloud or cluster machines, you must use `'AlternateFileSystemRoots'` to associate the root paths.

- To associate a set of root paths that are equivalent to one another, specify `'AlternateFileSystemRoots'` as a string vector. For example,

  ```
  ["Z:\datasets","/mynetwork/datasets"]
  ```

- To associate multiple sets of root paths that are equivalent for the datastore, specify `'AlternateFileSystemRoots'` as a cell array containing multiple rows where each row represents a set of equivalent root paths. Specify each row in the cell array as either a string vector or a cell array of character vectors. For example:

  - Specify `'AlternateFileSystemRoots'` as a cell array of string vectors.

    ```
    {["Z:\datasets", "/mynetwork/datasets"];...
     ["Y:\datasets", "/mynetwork2/datasets","S:\datasets"]}
    ```

  - Alternatively, specify `'AlternateFileSystemRoots'` as a cell array of cell array of character vectors.

    ```
    {{'Z:\datasets','/mynetwork/datasets'};...
     {'Y:\datasets', '/mynetwork2/datasets','S:\datasets'}}
    ```

The value of `'AlternateFileSystemRoots'` must satisfy these conditions:

- Contains one or more rows, where each row specifies a set of equivalent root paths.
- Each row specifies multiple root paths and each root path must contain at least two characters.
- Root paths are unique and are not subfolders of one another.

- Contains at least one root path entry that points to the location of the files.

For more information, see “Set Up Datastore for Processing on Different Machines or Clusters”.

Example: `["Z:\datasets","/mynetwork/datasets"]`

Data Types: `string` | `cell`

Properties

`NumFiles` — Number of files

numeric scalar

This property is read-only.

Number of files in the file-set object, specified as a numeric scalar.

Example: `fs.NumFiles`

Data Types: `double`

`FileSplitSize` — Split Size

`'file'` (default) | numeric scalar

This property is read-only.

Split size, specified as `'file'` or a numeric scalar.

The value assigned to `FileSplitSize` dictates the output from the `nextfile` method.

- If `FileSplitSize` is `'file'`, then the `nextfile` method returns a table with `FileName`, `FileSize`, `Offset`, and `SplitSize`. The value of `SplitSize` is set equal to the `FileSize`.
- If `FileSplitSize` is a numeric scalar `n`, then the `nextfile` method returns `FileName`, `FileSize`, `Offset`, and `SplitSize`. The value of `SplitSize` is set equal to the `FileSplitSize`. This information is used to read `n` bytes of the file. Subsequent calls to the `nextfile` method return information to help read the next `n` bytes of the same file until the end of the file.

Example: `'FileSplitSize',20`

Data Types: `double` | `char`

Methods

hasfile	Determine if more files are available in file-set object
maxpartitions	Maximum number of partitions
nextfile	Information on next file or file chunk
partition	Partition file-set object
subset	Create subset of datastore or file-set
reset	Reset the file-set object
resolve	Information on all files in file-set object

Examples

Get File Information for Collection of Files

Create a file-set object, get file information one file at time, or get information for all the files in the file-set object.

Create a file-set object for all the `.mat` files from the `demos` folder.

```
folder = fullfile(matlabroot,'toolbox','matlab','demos');
fs = matlab.io.datastore.DsFileSet(folder,...
                 'IncludeSubfolders',true,...
                 'FileExtensions','.mat');
```

Obtain information for the first and second file from the file-set object.

```
fTable1 = nextfile(fs) ; % first file
fTable2 = nextfile(fs) ; % second file
```

Obtain information on all the files by getting information for one file at a time and collect the information into a table.

```
ft = cell(fs.NumFiles,1); % using cell for efficiency
i = 1;
reset(fs); % reset to the beginning of the fileset
while hasfile(fs)
    ft{i} = nextfile(fs);
    i = i + 1;
end
allFiles = vertcat(ft{:});
```

Alternatively, obtain information on all files at the same time.

```
allfiles = resolve(fs);
```

Tips

- If you use the `DsFileSet` object as a property in your custom datastore, then implement the `copyElement` method. Implementing the `copyElement` method enables you to create a deep copy of the datastore object. For more information, see "Customize Copy Operation". For an example implementation of the `copyElement` method, see "Develop Custom Datastore".

See Also

`matlab.io.Datastore` | `matlab.io.datastore.DsFileReader` | `matlab.io.datastore.HadoopLocationBased` | `matlab.io.datastore.Partitionable`

Introduced in R2017b

hasfile

Class: `matlab.io.datastore.DsFileSet`
Package: `matlab.io.datastore`

Determine if more files are available in file-set object

Syntax

```
tf = hasfile(fs)
```

Description

`tf = hasfile(fs)` returns logical `1` (`true`) if any more files are available to process in the `DsFileSet` object specified by `fs`. Otherwise, it returns logical `0` (`false`).

Input Arguments

`fs` — Input file set
`matlab.io.datastore.DsFileSet` object

Input file set, specified as a `matlab.io.datastore.DsFileSet` object. To create a `DsFileSet` object, see `matlab.io.datastore.DsFileSet`.

Example: `hasfile(fs)`

Examples

Check if File-set Object has More Files

Create a file-set object, check if the file-set object has any files, and then read the data iteratively.

Create a file-set object for all the `.mat` files in the `demos` folder.

```
folder = fullfile(matlabroot,'toolbox','matlab','demos');
fs = matlab.io.datastore.DsFileSet(folder,...
                 'IncludeSubfolders',true,...
                 'FileExtensions','.mat');
```

Use the `hasfile` method to check if the file-set object has more files to read. Get the file information, one file at a time, while files remain in the file-set object.

```
while hasfile(fs)
   file = nextfile(fs);
end
```

See Also

`matlab.io.datastore.DsFileSet` | `maxpartitions` | `nextfile` | `partition` | `reset` | `resolve`

Topics

"Develop Custom Datastore"

Introduced in R2017b

maxpartitions

Class: `matlab.io.datastore.DsFileSet`
Package: `matlab.io.datastore`

Maximum number of partitions

Syntax

```
N = maxpartitions(fs)
```

Description

`N = maxpartitions(fs)` returns the maximum number of partitions for a given `DsFileSet` object `fs`.

Input Arguments

`fs` — Input file-set object
`matlab.io.datastore.DsFileSet` object

Input file-set object, specified as a `matlab.io.datastore.DsFileSet` object. To create a `DsFileSet` object, see `matlab.io.datastore.DsFileSet`.

Output Arguments

`N` — Number of partitions
positive integer

Number of partitions, returned as an integer. The value of `N` depends on the `FileSplitSize` property of the `DsFileSet` object.

- If `FileSplitSize` contains `'file'`, then `maxpartitions` sets `N` equal to the total number of files in `fs`.

- If `FileSplitSize` contains a numeric value, then `maxpartitions` determines `N` as the `sum` of the `ceil` of the file sizes of each file divided by the `FileSplitSize`.

Examples

Get Maximum Number of Partitions for the File-Set Object

Create a file-set object and get the number of partitions for different values of the `FileSplitSize` property.

Create a file-set object for all `.mat` files in a folder and check the number of files.

```
folder = fullfile(matlabroot,'toolbox','matlab','demos');
fs = matlab.io.datastore.DsFileSet(folder,'FileExtensions','.mat');
fs.NumFiles
```

```
ans =

   40
```

The `FileSplitSize` property of the `DSFileSet` object is set to `'file'`. Therefore, `maxpartitions` returns a value equal to the number of files.

```
maxpartitions(fs)
```

```
ans =

   40
```

Set the `FileSplitSize` property to a numeric size (`2000` bytes) and compute `maxpartitions`.

```
fs = matlab.io.datastore.DsFileSet(folder,...
                                   'FileExtensions','.mat',...
                                   'FileSplitSize',2000);
n = maxpartitions(fs)
```

```
n =

       1752
```

See Also

`hasfile` | `matlab.io.datastore.DsFileSet` | `nextfile` | `partition` | `reset` | `resolve`

Introduced in R2017b

nextfile

Class: `matlab.io.datastore.DsFileSet`
Package: `matlab.io.datastore`

Information on next file or file chunk

Syntax

```
fileInfo = nextfile(fs)
```

Description

`fileInfo = nextfile(fs)` returns the information on the next file available in the file set object `fs`. Subsequent calls to the `nextfile` function continue reading from the endpoint of the previous call.

Input Arguments

`fs` — Input file-set object
`matlab.io.datastore.DsFileSet` object

Input file-set object, specified as a `matlab.io.datastore.DsFileSet` object. To create a `DsFileSet` object, see `matlab.io.datastore.DsFileSet`.

Output Arguments

`fileInfo` — File information
table

File information, returned as a table. The table contains columns `FileName`, `FileSize`, `Offset`, and `SplitSize`.

Examples

Get File Information One File at a Time

Create a file-set object and get the file information iteratively, one file at a time.

Create a file-set object for all `.mat` files in the `demos` folder.

```
folder = fullfile(matlabroot,'toolbox','matlab','demos');
fs = matlab.io.datastore.DsFileSet(folder,...
                 'IncludeSubfolders',true,...
                 'FileExtensions','.mat');
```

Get the file information from the file-set object `fs`, one file at a time.

```
while hasfile(fs)
   file = nextfile(fs);
end
```

Tips

- The `nextfile` method returns an error if no more files exist in the file-set object `fs`. To avoid this error, use `nextfile` along with `hasfile` and `reset`. The `hasfile` method checks if files still remain in the file-set object, while the `reset` method resets the internal file pointer to the beginning of the file-set object.

See Also

`hasfile` | `matlab.io.datastore.DsFileSet` | `maxpartitions` | `partition` | `reset` | `resolve`

Introduced in R2017b

partition

Class: `matlab.io.datastore.DsFileSet`
Package: `matlab.io.datastore`

Partition file-set object

Syntax

```
subfs = partition(fs,n,index)
```

Description

`subfs = partition(fs,n,index)` partitions file set `fs` into the number of parts specified by `n` and returns the partition corresponding to the index `index`.

Input Arguments

`fs` — Input file-set object
`matlab.io.datastore.DsFileSet` object

Input file-set object, specified as a `matlab.io.datastore.DsFileSet` object. To create a `DsFileSet` object, see `matlab.io.datastore.DsFileSet`.

`n` — Number of partitions
positive integer

Number of partitions, specified as a positive integer.

Example: 3

Data Types: `double`

`index` — Index
positive integer

Index, specified as a positive integer.

Example: 1

Data Types: double

Output Arguments

subfs — Output file-set object
matlab.io.datastore.DsFileSet object

Output file-set object, specified as a matlab.io.datastore.DsFileSet object. The output file-set object subfs is a subset of the file-set object fs.

Examples

Partition File-Set Object and Get One Partition

Create a file-set object, partition the file-set object, and get the files contained in a specified partition.

Create a file-set object for all .mat files in a folder.

```
folder = fullfile(matlabroot,'toolbox','matlab','demos');
fs = matlab.io.datastore.DsFileSet(folder,'FileExtensions','.mat');
```

Partition the 40 files into five partitions and obtain the first partition. Then, get the file information of all files in the first partition.

```
subfsA = partition(fs,5,1) % first 8 files
subfsA_fileInfo = resolve(subfsA)
```

Obtain the second partition and files information for files in this partition.

```
subfsB = partition(fs,5,2) % next 8 files
subfsB_fileInfo = resolve(subfsB)
```

Tips

- Use the NumFiles property of the DsFileSet object to find the number of files in fs and use that number to estimate a reasonable number of partitions for the datastore.

See Also

`hasfile` | `matlab.io.datastore.DsFileSet` | `maxpartitions` | `nextfile` | `reset` | `resolve`

Introduced in R2017b

reset

Class: `matlab.io.datastore.DsFileSet`
Package: `matlab.io.datastore`

Reset the file-set object

Syntax

```
reset(fs)
```

Description

`reset(fs)` resets the file-set object `fs` to the state where no file has been read from it. Resetting enables rereading from the same file-set object.

Input Arguments

`fs` — Input file-set object
`matlab.io.datastore.DsFileSet` object

Input file-set object, specified as a `matlab.io.datastore.DsFileSet` object. To create a `DsFileSet` object, see `matlab.io.datastore.DsFileSet`.

Examples

Reset File-Set Object

Create a file-set object, get file information from the file-set object, and then reset the file-set object to its original state.

Create a file-set object for all the `.mat` files in the `demos` folder.

```
folder = fullfile(matlabroot,'toolbox','matlab','demos');
fs = matlab.io.datastore.DsFileSet(folder,...
```

```
                 'IncludeSubfolders',true,...
                 'FileExtensions','.mat');
```

While there are files in the file-set object, get the file information one file at a time.

```
while hasfile(fs)
   file = nextfile(fs);
end
```

The `nextfile` operation changes the position pointer in the file-set object. To get the information on the first file, you must first `reset` the file-set object.

```
reset(fs);
fTable1 = nextfile(fs); % first file information
```

See Also

`hasfile` | `matlab.io.datastore.DsFileSet` | `maxpartitions` | `nextfile` | `partition` | `resolve`

Introduced in R2017b

resolve

Class: `matlab.io.datastore.DsFileSet`
Package: `matlab.io.datastore`

Information on all files in file-set object

Syntax

```
allFilesInfo = resolve(fs)
```

Description

`allFilesInfo = resolve(fs)` returns file information on all the files from the file-set object `fs`.

Input Arguments

`fs` — Input file-set object
`matlab.io.datastore.DsFileSet` object

Input file-set object, specified as a `matlab.io.datastore.DsFileSet` object. To create a `DsFileSet` object, see `matlab.io.datastore.DsFileSet`.

Output Arguments

`allFilesInfo` — File information for all files
table

File information for all files in the file set, returned as a table. The table contains the columns `FileName` and `FileSize`.

Examples

Get File Information on All Files

Create a file-set object and get information for all the files in the file-set object at the same time.

Create a file-set object for all `.mat` files in a folder.

```
folder = fullfile(matlabroot,'toolbox','matlab','demos');
fs = matlab.io.datastore.DsFileSet(folder,'FileExtensions','.mat');
```

Obtain a table with information on all the files using `resolve`. The table contains columns for `FileName` and `FileSize`.

```
allfiles = resolve(fs);
```

See Also

`hasfile` | `matlab.io.datastore.DsFileSet` | `maxpartitions` | `nextfile` | `partition` | `reset`

Introduced in R2017b

duration

Lengths of time in fixed-length units

Description

The values in a *duration array* represent elapsed times in units of fixed length, such as hours, minutes, and seconds. You also can create elapsed times in terms of fixed-length (24-hour) days and fixed-length (365.2425-day) years.

Work with duration arrays as you would work with numeric arrays. You can add, subtract, sort, compare, concatenate, and plot duration arrays. Use duration arrays to simplify calculations on datetime arrays that involve time units such as hours and minutes.

Creation

You can create duration arrays that have specified time units using the `years`, `days`, `hours`, `minutes`, `seconds`, and `milliseconds` functions. For example, to create an array that has elapsed times of 1, 2, and 3 hours, use the `hours` function.

```
D = hours(1:3)
```

```
D =

  1×3 duration array

   1 hr   2 hr   3 hr
```

You also can create a duration array using the `duration` function, described below.

Syntax

```
D = duration(H,MI,S)
D = duration(H,MI,S,MS)
D = duration(X)
```

```
D = duration(TimeStrings)
```

```
D = duration(TimeStrings,'InputFormat',infmt)

D = duration( ___ ,'Format',displayFormat)
```

Description

`D = duration(H,MI,S)` creates a duration array from numeric arrays containing the number of hours, minutes, and seconds specified by `H`, `MI`, and `S`.

`D = duration(H,MI,S,MS)` adds milliseconds to the duration array, specified by `MS`.

`D = duration(X)` creates a column vector of durations from a numeric matrix.

`D = duration(TimeStrings)` converts text that represents elapsed times into a duration array. `TimeStrings` must represent times using either the `'hh:mm:ss'` or the `'dd:hh:mm:ss'` format.

`D = duration(TimeStrings,'InputFormat',infmt)` converts text using the format specified by `infmt`.

`D = duration( ___ ,'Format',displayFormat)` additionally specifies a display format for `D`. This property changes the display of `D`, but not its values. You can use this syntax with any of the arguments from the previous syntaxes.

Input Arguments

`H,MI,S` — Hour, minute, and second arrays

numeric arrays

Hour, minute, and second arrays, specified as numeric arrays. Any of these arrays can be a scalar. All arrays that are not scalars must be the same size.

Example: `duration(12,45,7)` returns a duration of 12 hours, 45 minutes, and 7 seconds.

`MS` — Millisecond array

numeric array

Millisecond array, specified as a numeric array. `MS` either must be a scalar or the same size as the `H`, `MI`, and `S` input arguments.

Example: `duration(12,45,30,35)` returns a duration of 12 hours, 45 minutes, 30 seconds, and 35 milliseconds.

X — Input matrix
numeric matrix

Input matrix, specified as a numeric matrix. X must have three columns, containing the numbers of hours, minutes, and seconds, respectively.

Example: `duration([12 30 16])` returns a duration of 12 hours, 30 minutes, and 16 seconds.

`TimeStrings` — Text representing elapsed times
character vector | cell array of character vectors | string array

Text representing elapsed times, specified as a character vector, a cell array of character vectors, or a string array. The `duration` function attempts to match the format of `TimeStrings` to either the `'hh:mm:ss'` or `'dd:hh:mm:ss'` formats, where `dd`, `hh`, `mm`, and `ss` represent days, hours, minutes, and seconds. The last field can include digits to the right of the decimal mark representing fractional seconds.

If you know the format, specify `'InputFormat'` and its corresponding `infmt` value.

Example: `duration('12:30:16')` returns a duration of 12 hours, 30 minutes, and 16 seconds.

Example: `duration('00:05:23.86')` returns a duration of 5 minutes and 23.86 seconds.

Example: `duration({'01:34:21';'23:16:54'})` returns a column vector containing two durations.

`infmt` — Format of input text
character vector | string scalar

Format of the input text, specified as a character vector or string scalar.

Specify `infmt` as any of the following formats, where `dd`, `hh`, `mm`, and `ss` represent days, hours, minutes, and seconds:

- `'dd:hh:mm:ss'`
- `'hh:mm:ss'`
- `'mm:ss'`

- `'hh:mm'`
- Any of the first three formats, with up to nine `S` characters to indicate fractional second digits, such as `'hh:mm:ss.SSSS'`

Properties

`Format` — Display format
`'hh:mm:ss'` (default) | character vector | string scalar

Display format, specified as a character vector or string scalar. The format can specify either a single number with time units (such as `'y'` for number of years) or a digital timer (such as `'hh:mm:ss'` for numbers of hours, minutes, and seconds).

For numbers with time units, specify one of the following:

- `'y'` — Fixed-length years, where one year equals 365.2425 days
- `'d'` — Fixed-length days, where one day equals 24 hours
- `'h'` — Hours
- `'m'` — Minutes
- `'s'` — Seconds

For digital timer formats, specify one of the following:

- `'dd:hh:mm:ss'`
- `'hh:mm:ss'`
- `'mm:ss'`
- `'hh:mm'`
- Any of the first three formats, with up to nine `S` characters to indicate fractional second digits, such as `'hh:mm:ss.SSSS'`

Example: `D.Format = 'm'` displays each value in `D` as a number of minutes.

Examples

Use Durations with Datetime Values

Create a datetime value.

```
D = datetime('today')
```

```
D = datetime
   26-Aug-2019
```

Create a datetime array in which each value has the same date but different time components. One convenient way to create such an array is to add a duration array to D.

First, create an array of hours using the `hours` function. Each element is two hours longer than the previous element.

```
H = hours(0:2:6)
```

```
H = 1x4 duration array
   0 hr   2 hr   4 hr   6 hr
```

Then, add D and H.

```
T = D + H
```

```
T = 1x4 datetime array
Columns 1 through 3

   26-Aug-2019 00:00:00   26-Aug-2019 02:00:00   26-Aug-2019 04:00:00

Column 4

   26-Aug-2019 06:00:00
```

Use duration arrays for arithmetic operations with datetime arrays and fixed lengths of time.

Create Duration Array

Create a duration array, specifying hours, minutes, and seconds as input arguments. Since the second argument is an array, output D is an array that has the same size.

```
D = duration(1,30:33,0)
```

```
D = 1x4 duration array
   01:30:00   01:31:00   01:32:00   01:33:00
```

Convert Matrix

Create a numeric matrix with three columns. The columns represent hours, minutes, and seconds respectively.

```
X = [12 17 54;9 32 3]
```

```
X = 2×3

    12    17    54
     9    32     3
```

Convert the matrix to a duration array.

```
D = duration(X)
```

```
D = 2x1 duration array
   12:17:54
   09:32:03
```

Convert Text to Duration Arrays

Convert a character vector representing a time as hours, minutes, and seconds.

```
T = '6:34:12';
D = duration(T)
```

```
D = duration
   06:34:12
```

Convert a cell array of character vectors.

```
T = {'12:54:37','8:03:12'};
D = duration(T)
```

```
D = 1x2 duration array
   12:54:37   08:03:12
```

Convert text that also has a day component. For display, the default format for duration arrays converts the number of days to hours.

```
T = '1:00:54:21';
D = duration(T)
```

```
D = duration
   24:54:21
```

Specify Input Format of Text

Specify the format of text representing elapsed times, and then convert them to duration arrays.

Convert a character vector. The input format represents minutes and seconds. The output argument is a duration value, whose format represents hours, minutes, and seconds.

```
T = '78:34';
infmt = 'mm:ss';
D = duration(T,'InputFormat',infmt)
```

```
D = duration
   01:18:34
```

Create a cell array of character vectors whose format represents minutes, seconds, and fractions of a second to three decimal places.

```
infmt = 'mm:ss.SSS';
T = {'1:34.862' '67:07.218'}
```

```
T = 1x2 cell array
    {'1:34.862'}    {'67:07.218'}
```

Convert T to a duration array. Specify that the format of the duration array represents hours, minutes, seconds, and fractions of a second.

```
outfmt = 'hh:mm:ss.SSS';
D = duration(T,'InputFormat',infmt,'Format',outfmt)
```

```
D = 1x2 duration array
   00:01:34.862   01:07:07.218
```

Specify Output Format of Duration Array

Create a duration array from a matrix. The three columns specify hours, minutes, and seconds, respectively. Display the values in digital timer format showing minutes and seconds.

```
X = [2 3 16;1 5 59;1 45 0]
```

```
X = 3×3

     2     3    16
     1     5    59
     1    45     0
```

```
D = duration(X,'Format','mm:ss')
```

```
D = 3x1 duration array
   123:16
    65:59
   105:00
```

Tips

- For more information on functions that accept or return duration arrays, see “Dates and Time”.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Starting in R2019b, you can use duration arrays in MATLAB code intended for code generation. For more information, see “Code Generation for Duration Arrays” (MATLAB Coder) and “Duration Array Limitations for Code Generation” (MATLAB Coder).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`calendarDuration` | `cellstr` | `char` | `datetime` | `isduration` | `string`

Topics

“Generate Sequence of Dates and Time”
“Compare Dates and Time”
“Date and Time Arithmetic”
“Plot Dates and Durations”

Introduced in R2014b

DurationRuler Properties

Control axis with duration values

Description

`DurationRuler` properties control the appearance and behavior of an *x*-axis, *y*-axis, or *z*-axis that shows duration values. Each individual axis has its own ruler object. By changing property values of the ruler, you can modify certain aspects of a specific axis.

Use dot notation to refer to a particular ruler and property. Access the ruler objects through the `XAxis`, `YAxis`, and `ZAxis` properties of the `Axes` object.

```
ax = gca;
co = ax.XAxis.Color;
ax.XAxis.Color = 'blue';
```

Properties

Appearance

`Color` — Color of axis line and labels

RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of the axis line and labels, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Note The `Color` property for the ruler and the associated `XColor`, `YColor`, or `ZColor` property for the parent axes always have the same value. Setting one also sets the other.

`LineWidth` — Width of axis line and tick marks

`0.5` (default) | positive value

Width of axis line and tick marks, specified as a positive value in point units. One point equals 1/72 inch.

Example: `ax.XAxis.LineWidth = 2;`

Note Setting the `LineWidth` property for the parent axes sets the `LineWidth` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

`Label` — Axis label

text object (default)

Axis label, which is a text object. To display text or change existing text, set the `String` property for the text object. Use other properties to change the text appearance, such as the font style or color.

```
ax = gca;
ax.XAxis.Label.String = 'X Axis';
ax.XAxis.Label.FontSize = 12;
```

For a full list of options, see Text.

Alternatively, add or modify the axis labels using the `xlabel`, `ylabel`, and `zlabel` functions.

Note The text object is not a child of the ruler object, so it cannot be returned by `findobj` and it does not use the default text property values.

`Visible` — Axis visibility

`'on'` (default) | `'off'`

Axis visibility, specified as one of these values:

- `'on'` — Display the axis.
- `'off'` — Hide the axis without deleting it. You still can access properties of an invisible axis using the ruler object.

Example: `ax.XAxis.Visible = 'off';`

Scale and Direction

Limits — Minimum and maximum axis limits

two-element vector of the form `[min max]`

Minimum and maximum axis limits, specified as a two-element vector of the form `[min max]`, where `min` and `max` are duration values. For example:

```
d = duration(1,40:45,0);
y = rand(6,1);
plot(d,y)
ax = gca;
ax.XAxis.Limits = [d(2) d(5)];
```

You can specify both limits, or specify one limit and let the ruler automatically calculate the other using an `Inf` or `-Inf` duration value, such as `days(Inf)`.

Alternatively, set the limits using the `xlim`, `ylim`, and `zlim` functions.

If you assign a value to this property, then MATLAB sets the associated mode to `'manual'`.

Note The `Limits` property for the ruler and the associated `XLim`, `YLim`, or `ZLim` property for the parent axes always have the same value. Setting one also sets the other.

LimitsMode — Selection mode for Limits property

`'auto'` (default) | `'manual'`

Selection mode for the `Limits` property, specified as one of these values:

- `'auto'` — Automatically select the axis limits based on the data plotted.
- `'manual'` — Use axis limit values that you specify. To specify the axis limits, set the `Limits` property.

Note The `LimitsMode` property for the ruler and the associated `XLimMode`, `YLimMode`, or `ZLimMode` property for the parent axes always have the same value. Setting one also sets the other.

Scale — Scale of values along axis

`'linear'` (default)

Scale of values along axis, returned as `'linear'`. `DurationRuler` objects do not support log scales.

`Direction` — Direction of increasing values

`'normal'` (default) | `'reverse'`

Direction of increasing values, specified as one of these values:

- `'normal'` — Values increase from left to right or bottom to top.
- `'reverse'` — Values increase from right to left or top to bottom.

Note The `Direction` property for the ruler and the associated `XDir`, `YDir`, or `ZDir` property for the parent axes always have the same value. Setting one also sets the other.

Example: `ax.XAxis.Direction = 'reverse';`

Tick Values and Labels

`TickValues` — Tick mark locations along the axis

`[]` (default) | vector of duration values

Tick mark locations along the axis, specified as a vector of duration values. For example:

```
d = duration(1,40:45,0);
y = rand(6,1);
plot(d,y)
ax = gca;
ax.XAxis.TickValues = [d(1) d(3) d(5)];
```

If you assign a value to this property, then MATLAB sets the `TickValuesMode` property to `'manual'`.

Alternatively, use the `xticks`, `yticks`, and `zticks` functions.

Note The `TickValues` property for the ruler and the associated `XTick`, `YTick`, or `ZTick` property for the parent axes always have the same value. Setting one also sets the other.

`TickValuesMode` — Selection mode for `TickValues` property

`'auto'` (default) | `'manual'`

Selection mode for the `TickValues` property, specified as one of these values:

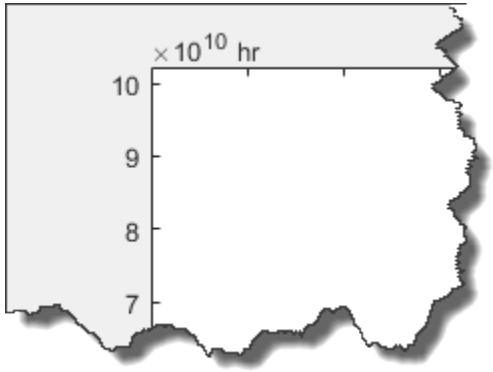
- `'auto'` — Automatically select the tick values based on the data plotted.
- `'manual'` — Use tick values that you specify. To specify the values, set the `TickValues` property.

Note The `TickValuesMode` property for the ruler and the associated `XTickMode`, `YTickMode`, or `ZTickMode` property for the parent axes always have the same value. Setting one also sets the other.

`Exponent` — Exponential notation common to all tick values

integer value

Exponential notation common to all tick values, specified as an integer value. The `Exponent` property applies only to tick labels that show a single unit of time, such as seconds, minutes, or hours. It does not apply to tick labels that show mixed units of time. When applicable, the ruler displays an exponent label showing the base and exponent value.



The base value is always 10. You can change the exponent value by setting the `Exponent` property. If the exponent value is `0`, then the exponent label is not displayed.

For example, this code changes the exponent value to 8.

```
x = hours((1:10)*1e10);
plot(x,1:10)
```

```
ax = gca;
ax.XAxis.Exponent = 8;
```

If you assign a value to this property, then MATLAB sets the `ExponentMode` property to `'manual'`.

`ExponentMode` — Selection mode for `Exponent` property
`'auto'` (default) | `'manual'`

Selection mode for the `Exponent` property, specified as one of these values:

- `'auto'` — Automatically select the exponent value based on the axis limits.
- `'manual'` — Use an exponent value that you specify. To specify the value, set the `Exponent` property.

`TickLabels` — Tick mark labels
`''` (default) | cell array of character vectors | string array | categorical array

Tick mark labels, specified as a cell array of character vectors, string array, or categorical array. If you do not specify enough labels for all of the tick values, then the labels repeat. The labels support TeX and LaTeX markup. See the `TickLabelInterpreter` property for more information.

Example: `ax.XAxis.TickLabels = {'January','February','March','April','May'}';`

Example: `ax.YAxis.TickLabels = {'\pi','2\pi','3\pi'}'`

If you set this property, then MATLAB sets the `TickLabelsMode` property to `'manual'`.

Alternatively, specify the tick labels using the `xticklabels`, `yticklabels`, and `zticklabels` functions.

Note

- If you specify this property as a categorical array, MATLAB uses the values in the array, not the categories.
- The `TickLabels` property for the ruler and the associated `XTickLabel`, `YTickLabel`, or `ZTickLabel` property for the parent axes always have the same value. Setting one also sets the other.

Data Types: `char` | `string` | `categorical`

`TickLabelsMode` — Selection mode for `TickLabels` property
`'auto'` (default) | `'manual'`

Selection mode for the `TickLabels` property, specified as one of these values:

- `'auto'` — Automatically select the tick labels.
- `'manual'` — Use tick labels that you specify. To specify the labels, set the `TickLabels` property.

Note The `TickLabelsMode` property for the ruler and the associated `XTickLabelMode`, `YTickLabelMode`, or `ZTickLabelMode` property for the parent axes always have the same value. Setting one also sets the other.

`TickLabelInterpreter` — Interpretation of tick label characters
`'tex'` (default) | `'latex'` | `'none'`

Interpretation of tick label characters, specified as one of these values:

- `'tex'` — Interpret characters using a subset of TeX markup.
- `'latex'` — Interpret characters using LaTeX markup.
- `'none'` — Display literal characters.

Note Setting the `TickLabelInterpreter` property for the parent axes sets the `TickLabelInterpreter` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.XAxis.TickLabelInterpreter = 'latex';`

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the text type and color, and include special characters in the text.

This table lists the supported modifiers with the `TickLabelInterpreter` property set to `'tex'`. Modifiers remain in effect until the end of the text. Superscripts and subscripts

are an exception because they only modify the next character or the text within the curly braces `{}`.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (rarely available)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{specifier}`	Set `specifier` as the name of a font family to change the font style. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{specifier}`	Set `specifier` as a scalar numeric value to change the font size.	`'\fontsize{15} text'`
`\color{specifier}`	Set `specifer` as one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Set `specifier` as a three-element RGB triplet to change the font color.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\angle	∠	\phi	ϕ	\leq	≤
\ast	*	\chi	χ	\infty	∞
\beta	β	\psi	ψ	\clubsuit	♣
\gamma	γ	\omega	ω	\diamondsuit	♦
\delta	δ	\Gamma	Γ	\heartsuit	♥
\epsilon	ϵ	\Delta	Δ	\spadesuit	♠
\zeta	ζ	\Theta	Θ	\leftrightarrow	↔
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	µ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\cap`	∩	`\in`	∈	`\supseteq`	⊇
`\supset`	⊃	`\lceil`	⌈	`\subset`	⊂
`\int`	∫	`\cdot`	·	`\o`	o
`\rfloor`	⌋	`\neg`	¬	`\nabla`	∇
`\lfloor`	⌊	`\times`	x	`\ldots`	...
`\perp`	⊥	`\surd`	√	`\prime`	´
`\wedge`	∧	`\varpi`	ϖ	`\0`	∅
`\rceil`	⌉	`\rangle`	〉	`\mid`	|
`\vee`	∨	`\langle`	〈	`\copyright`	©

LaTeX Markup

To use LaTeX markup, set the `TickLabelInterpreter` property to `'latex'`. Use dollar symbols around the labels, for example, use `'$\int_1^{20} x^2 dx$'` for inline mode or `'$$\int_1^{20} x^2 dx$$'` for display mode.

The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup within the text. The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters. For multiline text, the maximum size of the text reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project website at https://www.latex-project.org/.

`TickLabelFormat` — Tick label format

character vector | string scalar

Tick label format, specified as a character vector or string scalar.

To display a duration as a single number that includes a fractional part (for example, 1.234 hours), specify one of the following formats.

Format	Description
`'y'`	Number of exact, fixed-length years. A fixed-length year is equal to 365.2425 days.
`'d'`	Number of exact, fixed-length days. A fixed-length day is equal to 24 hours.
`'h'`	Number of hours
`'m'`	Number of minutes
`'s'`	Number of seconds

To display a duration in the form of a digital timer, specify one of these formats:

- `'dd:hh:mm:ss'`
- `'hh:mm:ss'`
- `'mm:ss'`
- `'hh:mm'`

In addition, you can display up to nine fractional second digits by appending up to nine `S` characters. For example, `'hh:mm:ss.SSS'` displays the milliseconds of a duration value to three digits.

`TickLabelRotation` — Rotation of tick labels
scalar value in degrees

Rotation of tick labels, specified as a scalar value in degrees. Positive values give counterclockwise rotation. Negative values give clockwise rotation.

Alternatively, you can rotate the tick labels using the `xtickangle`, `ytickangle`, and `ztickangle` functions.

Note The `TickLabelRotation` property for the ruler and the associated `XTickLabelRotation`, `YTickLabelRotation`, or `ZTickLabelRotation` property for the parent axes always have the same value. Setting one also sets the other.

Example: `ax.XAxis.TickLabelRotation = 45;`

Example: `ax.YAxis.TickLabelRotation = -45;`

`TickLabelRotationMode` — Selection mode for `TickLabelRotation` property

`'auto'` (default) | `'manual'`

Selection mode for the `TickLabelRotation` property, specified as one of these values:

- `'auto'` — Automatically select the tick label rotation.
- `'manual'` — Use a tick label rotation that you specify. To specify the rotation, set the `TickLabelRotation` property.

`TickDirection` — Tick mark direction

`'in'` (default) | `'out'` | `'both'`

Tick mark direction, specified as one of these values:

- `'in'` — Direct the tick marks inward from the axis lines. This is the default for 2-D views.
- `'out'` — Direct the tick marks outward from the axis lines. This is the default for 3-D views.
- `'both'` — Center the tick marks over the axis lines.

If you assign a value to this property, then MATLAB sets the `TickDirectionMode` property to `'manual'`.

Note Setting the `TickDir` property for the parent axes sets the `TickDirection` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.YAxis.TickDirection = 'out';`

`TickDirectionMode` — Selection mode for `TickDirection` property

`'auto'` (default) | `'manual'`

Selection mode for the `TickDirection` property, specified as one of these values:

- `'auto'` — Automatically select the tick direction.
- `'manual'` — Use a tick direction that you specify. To specify the tick direction, set the `TickDirection` property.

TickLength — Tick mark length

two-element vector

Tick mark length, specified as a two-element vector of the form `[2Dlength 3Dlength]`. The first element is the tick mark length in 2-D views. The second element is the tick mark length in 3-D views. Specify the values in units normalized relative to the longest axes dimension.

Note Setting the `TickLength` property for the parent axes sets the `TickLength` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.YAxis.TickLength = [0.02 0.035];`

MinorTick — Minor tick mark display

`'off'` (default) | `'on'`

Minor tick mark display, specified as one of these values:

- `'off'` — Do not display minor tick marks. This is the default value for an axis with a linear scale.
- `'on'` — Display minor tick marks between the major tick marks on the axis. This is the default value for an axis with a log scale. The space between the major tick marks determines the number of minor tick marks. If the `MinorTickValues` property is set to empty `[]`, then no minor tick marks appear. Specify the tick mark locations by setting the `MinorTickValues` property.

Note The `MinorTick` property for the ruler and the associated `XMinorTick`, `YMinorTick`, or `ZMinorTick` property for the parent axes always have the same value. Setting one also sets the other.

Example: `ax.XAxis.MinorTick = 'on';`

MinorTickValues — Minor tick mark locations

`[]` (default) | vector of increasing values

Minor tick mark locations, specified as a vector of increasing duration values.

If you assign values to this property, then MATLAB sets the `MinorTickValuesMode` property to `'manual'`.

`MinorTickValuesMode` — Selection mode for `MinorTickValues` property

`'auto'` (default) | `'manual'`

Selection mode for the `MinorTickValues` property, specified as one of these values:

- `'auto'` — Use automatically calculated minor tick values.
- `'manual'` — Use minor tick values that you specify. To specify the values, set the `MinorTickValues` property.

Font

`FontName` — Font name

system supported font name | `'FixedWidth'`

Font name, specified as a system supported font name or `'FixedWidth'`. The default font depends on the specific system and locale. To use a fixed-width font that renders well, specify `'FixedWidth'`. The actual fixed-width font used depends on the `FixedWidthFontName` property of the root object.

Note Setting the `FontName` property for the parent axes sets the `FontName` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.XAxis.FontName = 'Cambria';`

`FontSize` — Font size

scalar numeric value

Font size, specified as a scalar numeric value. The default font size depends on the specific operating system and locale.

Note Setting the `FontSize` property for the parent axes sets the `FontSize` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.XAxis.FontSize = 12;`

`FontWeight` — Thickness of text characters
`'normal'` (default) | `'bold'`

Thickness of the text characters, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font.
- `'bold'` — Thicker character outlines than normal.

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold font weight. Therefore, specifying a bold font weight still can result in the normal font weight.

Note Setting the `FontWeight` property for the parent axes sets the `FontWeight` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.XAxis.FontWeight = 'bold';`

`FontAngle` — Text character slant
`'normal'` (default) | `'italic'`

Character slant, specified as `'normal'` or `'italic'`. Not all fonts have both font styles. Therefore, the italic font can look the same as the normal font.

Note Setting the `FontAngle` property for the parent axes sets the `FontAngle` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.XAxis.FontAngle = 'italic';`

`FontSmoothing` — Text antialiasing
`'on'` (default) | `'off'`

Text smoothing, specified as one of these values:

- `'on'` — Enable text antialiasing to reduce the jagged appearance of text characters and make the text easier to read. In certain cases, smoothed text blends against the background color and can make the text appear blurry.
- `'off'` — Disable text antialiasing. Use this setting if the text seems blurry.

Note Setting the `FontSmoothing` property for the parent axes sets the `FontSmoothing` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.XAxis.FontSmoothing = 'off';`

Parent/Child

`Parent` — Ruler parent

`Axes` object

Ruler parent, specified as an `Axes` object.

Note Ruler objects are not listed in the `Children` property of the parent `Axes` object.

`Children` — Ruler children

empty `GraphicsPlaceholder` array

The ruler has no children. You cannot set this property.

See Also

Axes

Introduced in R2016b

dynamicprops class

Superclass for classes that support dynamic properties

Description

`dynamicprops` is an abstract class derived from the `handle` class. Subclass `dynamicprops` to define classes that support dynamic properties.

Dynamic properties are associated with a specific object of the class, but are not part of the class definition. Use dynamic properties to attach temporary data to objects. For information on using dynamic properties, see “Dynamic Properties — Adding Properties to an Instance”.

Note The `isequal` function always returns `false` when comparing objects that have dynamic properties, even if the properties have the same name and value. If you need to compare objects that contain dynamic properties, overload `isequal` as a method of your class.

The `dynamicprops` class is a `handle` class.

Class Attributes

`Abstract`	`true`
`ConstructOnLoad`	`true`
`HandleCompatible`	`true`

For information on class attributes, see “Class Attributes”.

Methods

Public Methods

addprop Add dynamic property

Events

Event Name	Trigger	Event Data	Event Attributes
`PropertyAdded`	When property is added	`event.DynamicPropertyEvent`	`NotifyAccess = private`, `ListenAccess = public`
`PropertyRemoved`	When property is removed	`event.DynamicPropertyEvent`	`NotifyAccess = private`, `ListenAccess = public`

Examples

Subclass dynamicprops

Derive a class from the `dynamicprops` class:

```
classdef InstanceProperty < dynamicprops
end
```

Add a property to an object of the class using the `addprop` method and assign a value to the property.

```
obj = InstanceProperty;
obj.addprop('NewProperty');
obj.NewProperty = 1:10
```

```
obj = 

  InstanceProperty with properties:
```

```
NewProperty: [1 2 3 4 5 6 7 8 9 10]
```

See Also

`handle`

Topics

"Dynamic Properties — Adding Properties to an Instance"
"Set and Get Methods for Dynamic Properties"
"Dynamic Property Events"
"Dynamic Properties and ConstructOnLoad"

Introduced in R2008a

echo

Display statements during function execution

Syntax

```
echo on
echo off
echo
echo fcnname on
echo fcnname off
echo fcnname
echo on all
echo off all
```

Description

The `echo` command controls the display (or echoing) of statements in a function during their execution. Normally, statements in a function file are not displayed on the screen during execution. Command echoing is useful for debugging or for demonstrations, allowing the commands to be viewed as they execute.

The `echo` command behaves in a slightly different manner for script files and function files. For script files, the use of `echo` is simple; echoing can be either `on` or `off`, in which case any script used is affected.

`echo on` turns on the echoing of commands in all script files.

`echo off` turns off the echoing of commands in all script files.

`echo` toggles the echo state.

When you turn on echoing for a script or function file, each line in the file is displayed as it is executed. Since this results in inefficient execution, use `echo` only for debugging.

`echo fcnname on` turns on echoing of the named function file.

`echo fcnname off` turns off echoing of the named function file.

`echo` *`fcnname`* toggles the echo state of the named function file.

`echo on all` sets echoing on for all function files.

`echo off all` sets echoing off for all function files.

Tips

- To avoid confusing syntax, do not use `on` or `off` as a function name.

See Also

`function`

Introduced before R2006a

echodemo

Run example script step-by-step in Command Window

Syntax

```
echodemo filename
echodemo(filename,index)
```

Description

`echodemo filename` runs the script specified by `filename` step-by-step in the Command Window. If file contains sections, `echodemo` treats each section as a step and pauses after each one. For more information about sections, see “Code Sections”.

At each step, you can click links in the Command Window to proceed or stop. If the Command Window is not large enough to show the links, scroll up to see them.

Caution If variables in your base workspace have the same name as variables that the example file creates, the example could overwrite your data. Preserve your data by saving it to a MAT-file before running the example.

`echodemo(filename,index)` starts with the section number specified by `index`. If the example relies on results of previous steps, using this syntax can produce errors or unexpected results.

Examples

Run Script Step-by-Step in Command Window

Run an example script that demonstrates how to work with multidimensional arrays step-by-step.

```
echodemo MultidimensionalArraysDemo
```

```
Click Next to continue or Stop to end     MultidimensionalArraysDemo.m (1/10)
--------------------------------------------------------------------------
 MANIPULATING MULTIDIMENSIONAL ARRAYS

 This example shows how to work with arrays having more than two dimensions.
 Multidimensional arrays can be numeric, character, cell, or structure arrays.

 Multidimensional arrays can be used to represent multivariate data. MATLAB®
 provides a number of functions that directly support multidimensional arrays.
--------------------------------------------------------------------------
```

Start Script from Specified Section

Run an example script that demonstrates how to work with multidimensional arrays step-by-step, starting from the third section.

```
filename = 'MultidimensionalArraysDemo';
index = 3;
echodemo(filename,index)
```

```
Click Next to continue or Stop to end     MultidimensionalArraysDemo.m (3/10)
--------------------------------------------------------------------------
 The |cat| function is a useful tool for building multidimensional arrays.
 |B = cat(DIM,A1,A2,...)| builds a multidimensional array by concatenating |A1,
 A2 ...| along the dimension |DIM|.
--------------------------------------------------------------------------

B = cat( 3, [2 8; 0 5], [1 3; 7 9], [2 3; 4 6])


B(:,:,1) =

     2     8
     0     5


B(:,:,2) =

     1     3
     7     9
```

```
B(:,:,3) =

     2     3
     4     6
```

Input Arguments

`filename` — Script file name
character vector | string scalar

Script file name, specified as a character vector or string scalar.

`index` — Section index
scalar integer

Section index, specified as a scalar integer.

The link text in the Command Window shows the current section number, `n`, and the total number of sections, `m`, as `n/m`.

Limitations

- `echodemo` does not support functions or live code files (`*.mlx`). `echodemo` can run any script (`*.m`) that you can execute, but only scripts with sections pause between steps.

See Also

`demo` | `doc` | `publish`

Topics

"Code Sections"

Introduced before R2006a

edgeAttachments

Class: `TriRep`

(Not recommended) Simplices attached to specified edges

Note `edgeAttachments(TriRep)` is not recommended. Use `edgeAttachments(triangulation)` instead.

`TriRep` is not recommended. Use `triangulation` instead.

Syntax

```
SI = edgeAttachments(TR, V1, V2)
SI = edgeAttachments(TR, EDGE)
```

Description

`SI = edgeAttachments(TR, V1, V2)` returns the simplices `SI` attached to the edges specified by `(V1, V2)`. `(V1, V2)` represents the start and end vertices of the edges to be queried.

`SI = edgeAttachments(TR, EDGE)` specifies edges in matrix format.

Input Arguments

`TR`	Triangulation representation.
`V1,V2`	Column vectors of vertex indices into the array of points representing the vertex coordinates.
`EDGE`	Matrix specifying edge start and end points. `EDGE` is of size `m`-by-2, `m` being the number of edges to query.

Output Arguments

`SI`	Vector cell array of indices into the triangulation matrix. `SI` is a cell array because the number of simplices associated with each edge can vary.

Examples

Example 1

Load a 3-D triangulation to compute the tetrahedra attached to an edge.

```
load tetmesh
trep = TriRep(tet, X);
v1 = [15 21]';
v2 = [936 716]';
t1 = edgeAttachments(trep, v1, v2);
```

You can also specify the input as edges.

```
e = [v1 v2];
t2 = edgeAttachments(trep, e);
isequal(t1,t2);
```

Example 2

Create a triangulation with `DelaunayTri`.

```
x = [0 1 1 0 0.5]';
y = [0 0 1 1 0.5]';
dt = DelaunayTri(x,y);
```

Query the triangles attached to edge (1,5).

```
t = edgeAttachments(dt, 1,5);
t{:};
```

More About

Simplex

A simplex is a triangle/tetrahedron or higher dimensional equivalent.

See Also

`delaunayTriangulation` | `edges` | `triangulation`

edges

Class: `TriRep`

(Not recommended) Triangulation edges

Note `edges(TriRep)` is not recommended. Use `edges(triangulation)` instead.

`TriRep` is not recommended. Use `triangulation` instead.

Syntax

```
E = edges(TR)
```

Description

`E = edges(TR)` returns the edges in the triangulation in an `n`-by-2 matrix. `n` is the number of edges. The vertices of the edges index into `TR.X`, the array of points representing the vertex coordinates.

Input Arguments

`TR`	Triangulation representation.

Output Arguments

`E`	Edge matrix.

Examples

Example 1

Load a 2-D triangulation.

```
load trimesh2d
trep = TriRep(tri, x,y);
```

Return all edges.

```
e = edges(trep);
```

Example 2

Query a 2-D `DelaunayTri`-generated triangulation.

```
X = rand(10,2);
dt = DelaunayTri(X);
e = edges(dt);
```

See Also

`delaunayTriangulation` | `edgeAttachments` | `triangulation`

edit

Edit or create file

Syntax

```
edit
edit file
edit file1 ... fileN
```

Description

`edit` opens a new file called `Untitled.m` in the Editor. MATLAB does not automatically save `Untitled.m`.

`edit file` opens the specified file in the Editor. If `file` does not already exist, MATLAB asks if you want to create it. `file` can include a partial path, complete path, relative path, or no path. If `file` includes a partial path or no path, `edit` will look for the file on the search path. You must have write permission to the path to create `file`. Otherwise, MATLAB ignores the argument.

You must specify the extension to open `.mat` and `.mdl` files. MATLAB cannot directly edit binary files, such as `.p` and `.mex` files.

`edit file1 ... fileN` opens each file, `file1 ... fileN`, in the Editor.

Examples

Open New File

Create and open a new file titled `Untitled.m` in the MATLAB® Editor (or default editor). `Untitled.m` does not appear in your Current Folder.

```
edit
```

Create New Files

Create two new files in a new folder.

Create a file in a new directory using a character vector. A dialog box appears, asking if you want to create `new_script.m`. Click **Yes** to create and open `tests/new_script.m`.

```
mkdir tests
edit tests/new_script.m
```

Create a second file using a string. Click **Yes** to create and open `tests/new_script2.m`.

```
S = "tests/new_script2.m";
edit(S)
```

Open Files

Create and open the files `file1`, `file2`, `file3`, and `file4` in sequence.

```
edit file1 file2 file3 file4
```

Input Arguments

`file` — Name of file
character vector | string

Name of file, specified as a character vector or string. If `file` specifies a path that contains a nonexistent folder, MATLAB throws an error. Specify multiple files on the same line by separating file names with a space.

If you do not specify the extension and MATLAB is unable to find `file`, then `edit` creates a file with the specified name and a `.m` extension.

If `file` is overloaded (that is, appears in multiple folders on the search path), then include a partial path to edit the correct page, such as:

```
edit folderName/file
```

If the file is part of a class or package, then either specify the path and extension or separate the components of the name with periods, such as:

```
edit className.name
edit packageName.name
edit packageName.className.name
edit packageName.name
```

Data Types: `char` | `string`

See Also

`open` | `type`

Topics

"Editor/Debugger Preferences"

Introduced before R2006a

matlab.editor Settings

Editor settings, including language, backup, and display settings

You can customize the visual appearance and behavior of the Editor using `matlab.editor` settings. Access `matlab.editor` settings using the root `SettingsGroup` object returned by the `settings` function. For example, set the temporary value for the maximum column width for comments to `80`.

```
s = settings;
s.matlab.editor.language.matlab.comments.MaxWidth.TemporaryValue = 80
```

For more information about settings, see “Access and Modify Settings”.

Settings

matlab.editor

`UseMATLABEditor` — Use the MATLAB Editor

`1` (default) | `0`

Use the MATLAB Editor, specified as `1` or `0`.

Set to `1` if you want the MATLAB desktop to use the MATLAB Editor when you edit a file.

This setting only applies to plain text files such as files with a `.m` or `.txt` extension.

Example: `s.matlab.editor.UseMATLABEditor.TemporaryValue = 0`

`OtherEditor` — Other editor name

character vector

Other editor name, specified as a character vector.

If `UseMATLABEditor` is set to `0`, specify the full path for the editor application you want to use, such as Emacs or vi.

Example: `s.matlab.editor.OtherEditor.TemporaryValue = 'c:\Applications\Emacs.exe'`

`RecentFileListSize` — Number of recent files to display

8 (default) | positive integer

Number of recent files to display in the **Open** menu, specified as a positive integer.

Example: `s.matlab.editor.RecentFileListSize.TemporaryValue = 5`

`ReopenFilesOnRestart` — Reopen files on restart
`1` (default) | `0`

Reopen files on restart, specified as `1` or `0`.

Set to `1` if you want the Editor and the files it contained during your last MATLAB session to reopen when you restart MATLAB.

Example: `s.matlab.editor.AddLineTerminationOnsave.TemporaryValue = 0`

`OpenFileAtBreakpoint` — Open a file at a breakpoint
`1` (default) | `0`

Open a file at a breakpoint, specified as `1` or `0`.

Set to `1` to open a running program file when MATLAB encounters a breakpoint in that file.

Example: `s.matlab.editor.OpenFileAtBreakpoint.TemporaryValue = 0`

`SaveFilesOnClickAway` — Save files on click away
`1` (default) | `0`

Save files on click away, specified as `1` or `0`.

Set to `1` if you want the Editor to automatically save changes to a file in the Editor when you click away from the Editor. For the changes to be automatically saved upon clicking away from the Editor, you must have already saved the file at least once.

Example: `s.matlab.editor.SaveFilesOnClickAway.TemporaryValue = 0`

`ReloadFilesOnChange` — Reload files when they change
`1` (default) | `0`

Reload files when they change, specified as `1` or `0`.

Set to `1` if you want the Editor to automatically reload the new version of a file that you opened and edited outside of MATLAB. MATLABonly reloads the file if it contains no unsaved changes.

Example: `s.matlab.editor.ReloadFilesOnChange.TemporaryValue = 0`

`AddLineTerminationOnSave` — Add a new empty line on save
`1` (default) | `0`

Add a new empty line on save, specified as `1` or `0`.

Set to `1` to have MATLAB add a new empty line (sometimes referred to as a `<CR>`) to the end of a file automatically if the last line in the file is not empty.

This setting only applies to plain text files such as files with a `.m` or `.txt` extension.

Example: `s.matlab.editor.AddLineTerminationOnSave.TemporaryValue = 0`

`OnlyStepInToUserFunctions` — Step only into user-defined functions
`1` (default) | `0`

Step only into user-defined functions and scripts when debugging in the Live Editor, specified as `1` or `0`.

Set to `1` to only step into user-defined functions and scripts when debugging in the Live Editor. Set to `0` to step into MathWorks functions as well.

Example: `s.matlab.editor.OnlyStepInToUserFunctions.TemporaryValue = 0`

`AllowFigureAnimation` — Enable animations in Live Editor
`1` (default) | `0`

Enable animations in the Live Editor, specified as `1` or `0`.

Set to `1` to enable for-loop animations in the Live Editor and show changes in plotted data over time.

Example: `s.matlab.editor.AllowFigureAnimation.TemporaryValue = 0`

matlab.editor.displaysettings

`HighlightCurrentLine` — Highlight the current line
`1` (default) | `0`

Highlight the current line, specified as `1` or `0`.

Set to `1` to highlight the current line. The current line is the line with the cursor (also called the caret). This setting does not apply in the Live Editor.

Example:
`s.matlab.editor.displaysettings.HighlightCurrentLine.TemporaryValue = 0`

`HighlightCurrentLineColor` — Current line color
`[222 238 216]` (default) | RGB triplet

Current line color, specified as an RGB triplet with values in the range [0,255]. The current line is the line with the cursor (also called the caret).

`HighlightCurrentLine` must be set to `1` to see the change in the current line color.

Example:
`s.matlab.editor.displaysettings.HighlightCurrentLineColor.TemporaryValue = [222 230 200]`

`ShowLineNumbers` — Show line numbers
`1` | `0`

Show line numbers, specified as `1` or `0`.

Set to `1` to display line numbers along the left edge of the Editor window.

Example:
`s.matlab.editor.displaysettings.ShowLineNumbers.TemporaryValue = 0`

`DataTipsInEditMode` — Enable data tips when editing
`1` (default) | `0`

Enable data tips when editing, specified as `1` or `0`.

Set to `1` to display data tips when you are editing a MATLAB code file. (Data tips are always enabled in debug mode.)

This setting does not apply in the Live Editor.

Example:
`s.matlab.editor.displaysettings.DataTipsInEditMode.TemporaryValue = 0`

`ShowOpenAsLiveScriptBanner` — Show Open as a Live Script banner
`1` (default) | `0`

Show Open as a Live Script banner, specified as `1` or `0`.

Set to `1` to display the Open as a Live Script banner at the top of documents in the Editor.

Example:
`s.matlab.editor.displaysettings.ShowOpenAsLiveScriptBanner.TemporaryValue = 0`

`ShowWelcomeToLiveEditorBanner` — Show Welcome to the Live Editor banner
`1` (default) | `0`

Show Welcome to the Live Editor banner, specified as `1` or `0`.

Set to `1` to display the Welcome to the Live Editor banner at the top of documents in the Live Editor.

Example:
`s.matlab.editor.displaysettings.ShowWelcomeToLiveEditorBanner.TemporaryValue = 0`

matlab.editor.displaysettings.linelimit

`ShowLine` — Display the line limit
`1` (default) | `0`

Display the line limit, specified as `1` or `0`.

Set to `1` to display a vertical line at a specified column in the Editor to denote the right-hand text limit. `LineColumn` specifies the location of the line, and `LineWidth` specifies the width of the line.

Example:
`s.matlab.editor.displaysettings.linelimit.ShowLine.TemporaryValue = 0`

`LineColumn` — Location of the line limit
`75` (default) | positive integer

Location of the line limit, specified as a positive integer. Specify the location as a column number.

Example:
`s.matlab.editor.displaysettings.linelimit.LineColumn.TemporaryValue = 80`

`LineWidth` — Width of line limit
`1` (default) | positive integer

Width of the line limit, specified as a positive integer.

```
s.matlab.editor.displaysettings.linelimit.LineWidth.TemporaryValue =
2
```

`LineColor` — Color of line limit

`[204 204 204]` (default) | RGB triplet

Color of the line limit, specified as a RGB triplet with values in the range [0,255].

```
s.matlab.editor.displaysettings.linelimit.LineColor.TemporaryValue =
[173 235 255]
```

matlab.editor.tab

`TabSize` — Tab size

4 (default) | positive integer

Tab size (the number of spaces inserted when you press the **Tab** key), specified as a positive integer.

When you change the `TabSize`, it changes the tab size for existing lines in that file, unless you also set `InsertSpaces` to `1`.

Example: `s.matlab.editor.tab.TabSize.TemporaryValue = 5`

`IndentSize` — Indent size for smart indenting

4 (default) | positive integer

Indent size for smart indenting, specified as a positive integer. For more information about smart indenting, see “Improve Code Readability”.

Example: `s.matlab.editor.tab.IndentSize.TemporaryValue = 5`

`InsertSpaces` — Insert spaces

`1` (default) | `0`

Insert spaces when you press the **Tab** key, specified as a `1` or `0`.

Set to `1` to insert a series of spaces when you press the **Tab** key. Otherwise, a tab acts as one space whose length is equal to the `TabSize`.

Example: `s.matlab.editor.tab.InsertSpaces.TemporaryValue = 0`

EmacsStyle — Use Emacs indenting style
1 (default) | 0

Use Emacs indenting style, specified as 1 or 0.

Set to 1 to use an indenting style similar to the style that the Emacs editor uses.

Example: s.matlab.editor.tab.IndentSize.TemporaryValue = 5

matlab.editor.language.matlab

EnableSyntaxHighlighting — Enable syntax highlighting
1 (default) | 0

Enable syntax highlighting, specified as 1 or 0.

Set to 1 to have the Editor use different colors for different language constructs. Then, adjust the colors you want to use for each language element. For more information, see "Syntax Highlighting". Disabling syntax highlighting is not supported in the Live Editor.

Example:
s.matlab.editor.language.matlab.EnableSyntaxHighlighting.TemporaryValue = 0

EnableVariableAndFunctionRenaming — Enable variable and function renaming
1 (default) | 0

Enable variable and function renaming, specified as 1 or 0.

Set to 1 to have MATLAB prompt you to rename all instances of a function or variable in a file when you rename a single instance.

Example:
s.matlab.editor.language.matlab.EnableVariableAndFunctionRenaming.TemporaryValue = 0

SmartIndentWhileTyping — Apply smart indenting while typing
SmartIndent (default) | NoIndent

Apply smart indenting while typing, specified as SmartIndent or NoIndent.

Specify SmartIndent to automatically:

- Indent the body of loops within the start and end of the loop statement.

- Align subsequent lines with lines you indent using tabs or spaces.
- Indent functions as specified with the `FunctionIndentingFormat` setting.

Example:
`s.matlab.editor.language.matlab.SmartIndentWhileTyping.TemporaryValue = 'NoIndent'`

`FunctionIndentingFormat` — Function indenting format
`MixedFunctionIndent` (default) | `AllFunctionIndent` | `ClassicFunctionIndent`

Function indenting format, specified as one of these values:

- **`ClassicFunctionIndent`** — Align function code with the function declaration.
- **`MixedFunctionIndent`** — Indent function code within a nested function.
- **`AllFunctionIndent`** — Indent function code for both main and nested functions.

Example:
`s.matlab.editor.language.matlab.FunctionIndentingFormat.TemporaryValue = 'ClassicFunctionIndent'`

matlab.editor.language.matlab.comments

`MaxWidth` — Maximum number of characters
`75` (default) | positive integer

Maximum number of characters in a comment line, specified as a positive integer.

Specify `FromCommentStart` to select where you want counting to begin.

Example:
`s.matlab.editor.language.matlab.comments.MaxWidth.TemporaryValue = 80`

`FromCommentStart` — Count maximum width from comment start
`1` (default) | `0`

Count maximum width from comment start, specified as `1` or `0`.

Set to `1` to count width from the beginning of a comment. This is useful when comments are indented, and you want each block of comments to have the same indent and width.

Set to `0` to count width from the beginning of a line. This is useful when the absolute width of the comments is important. For example, set the maximum width to 75 columns

and set `FromCommentStart` to `0` to make your code fit on a printed page when you use the default font for the Editor.

Example:
`s.matlab.editor.language.matlab.comments.FromCommentStart.TemporaryValue = 0`

`WrapAutomatically` — Wrap comments automatically
`1` (default) | `0`

Wrap comments automatically, specified as `1` or `0`.

Set to `1` to automatically wrap comments after `MaxWidth` columns when you type comments in an Editor document.

Example:
`s.matlab.editor.language.matlab.comments.WrapAutomatically.TemporaryValue = 0`

matlab.editor.codefolding

`EnableCodeFolding` — Enable code folding
`1` (default) | `0`

Enable code folding, specified as `1` or `0`.

Set to `1` to enable code folding.

Example: `s.matlab.editor.codefolding.EnableCodeFolding.TemporaryValue = 0`

`FoldBlockComments` — Fold block comments
`1` (default) | `0`

Fold block comments, specified as `1` or `0`.

Set to `1` to enable code folding for block comments. `EnableCodeFolding` must be set to `1`. Otherwise, code folding for block comments is disabled.

Example: `s.matlab.editor.codefolding.FoldBlockComments.TemporaryValue = 0`

`InitiallyFoldBlockComments` — Initially fold block comments
`0` (default) | `1`

Initially fold block comments, specified as `1` or `0`.

Set to `1` to fold block comments the first time that you open a MATLAB file.

Example: `s.matlab.editor.codefolding.InitiallyFoldBlockComments.TemporaryValue = 1`

`FoldClassCode` — Fold class code
`1` (default) | `0`

Fold class code, specified as `1` or `0`.

Set to `1` to enable code folding for class code. `EnableCodeFolding` must be set to `1`. Otherwise, code folding for class code is disabled.

Example: `s.matlab.editor.codefolding.FoldClassCode.TemporaryValue = 0`

`InitiallyFoldClassCode` — Initially fold class code
`0` (default) | `1`

Initially fold class code, specified as `1` or `0`.

Set to `1` to fold class code the first time that you open a MATLAB file.

Example: `s.matlab.editor.codefolding.InitiallyFoldClassCode.TemporaryValue = 1`

`FoldClassEnumerationBlocks` — Fold class enumeration blocks
`1` (default) | `0`

Fold class enumeration blocks, specified as `1` or `0`.

Set to `1` to enable code folding for class enumeration blocks. `EnableCodeFolding` must be set to `1`. Otherwise, code folding for class enumeration blocks is disabled.

Example: `s.matlab.editor.codefolding.FoldClassEnumerationBlocks.TemporaryValue = 0`

`InitiallyFoldClassEnumerationBlocks` — Initially fold class enumeration blocks
`0` (default) | `1`

Initially fold class enumeration blocks, specified as `1` or `0`.

Set to `1` to fold class enumeration blocks the first time that you open a MATLAB file.

Example:
```
s.matlab.editor.codefolding.InitiallyFoldClassEnumerationBlocks.Temporary
Value = 1
```

`FoldClassEventBlocks` — Fold class event blocks
`1` (default) | `0`

Fold class event blocks, specified as `1` or `0`.

Set to `1` to enable code folding for class event blocks. `EnableCodeFolding` must be set to `1`. Otherwise, code folding for class event blocks is disabled.

Example:
```
s.matlab.editor.codefolding.FoldClassEventBlocks.TemporaryValue = 0
```

`InitiallyFoldClassEventBlocks` — Initially fold class event blocks
`0` (default) | `1`

Initially fold class event blocks, specified as `1` or `0`.

Set to `1` to fold class event blocks the first time that you open a MATLAB file.

Example:
```
s.matlab.editor.codefolding.InitiallyFoldClassEventBlocks.TemporaryV
alue = 1
```

`FoldClassMethodBlocks` — Fold class method blocks
`1` (default) | `0`

Fold class method blocks, specified as `1` or `0`.

Set to `1` to enable code folding for class method blocks. `EnableCodeFolding` must be set to `1`. Otherwise, code folding for class method blocks is disabled.

Example:
```
s.matlab.editor.codefolding.FoldClassMethodBlocks.TemporaryValue = 0
```

`InitiallyFoldClassMethodBlocks` — Initially fold class method blocks
`0` (default) | `1`

Initially fold class method blocks, specified as `1` or `0`.

Set to `1` to fold class method blocks the first time that you open a MATLAB file.

Example:
```
s.matlab.editor.codefolding.InitiallyFoldClassMethodBlocks.TemporaryValue = 1
```

`FoldClassPropertiesBlocks` — Fold class properties blocks
`1` (default) | `0`

Fold class properties blocks, specified as `1` or `0`.

Set to `1` to enable code folding for class properties blocks. `EnableCodeFolding` must be set to `1`. Otherwise, code folding for class properties blocks is disabled.

Example:
```
s.matlab.editor.codefolding.FoldClassPropertiesBlocks.TemporaryValue = 0
```

`InitiallyFoldClassPropertiesBlocks` — Initially fold class properties blocks
`0` (default) | `1`

Initially fold class properties blocks, specified as `1` or `0`.

Set to `1` to fold class properties blocks the first time that you open a MATLAB file.

Example:
```
s.matlab.editor.codefolding.InitiallyFoldClassPropertiesBlocks.TemporaryValue = 1
```

`FoldForAndParforBlocks` — Fold `for` and `parfor` blocks
`1` (default) | `0`

Fold `for` and `parfor` blocks, specified as `1` or `0`.

Set to `1` to enable code folding for `for` and `parfor` blocks. `EnableCodeFolding` must be set to `1`. Otherwise, code folding for `for` and `parfor` blocks is disabled.

Example:
```
s.matlab.editor.codefolding.FoldForAndParforBlocks.TemporaryValue = 0
```

`InitiallyFoldForAndParforBlocks` — Initially fold `for` and `parfor` blocks
`0` (default) | `1`

Initially fold `for` and `parfor` blocks, specified as `1` or `0`.

Set to `1` to fold `for` and `parfor` blocks the first time that you open a MATLAB file.

Example: `s.matlab.editor.codefolding.InitiallyFoldForAndParforBlocks.TemporaryValue = 1`

`FoldFunctionCode` — Fold function code
`1` (default) | `0`

Fold function code, specified as `1` or `0`.

Set to `1` to enable code folding for function code. `EnableCodeFolding` must be set to `1`. Otherwise, code folding for function code is disabled.

Example: `s.matlab.editor.codefolding.FoldFunctionCode.TemporaryValue = 0`

`InitiallyFoldFunctionCode` — Initially fold function code
`0` (default) | `1`

Initially fold function code, specified as `1` or `0`.

Set to `1` to fold function code the first time that you open a MATLAB file.

Example: `s.matlab.editor.codefolding.InitiallyFoldFunctionCode.TemporaryValue = 1`

`FoldIfElseBlocks` — Fold `if`/`else` blocks
`1` (default) | `0`

Fold `if`/`else` blocks, specified as `1` or `0`.

Set to `1` to enable code folding for `if`/`else` blocks. `EnableCodeFolding` must be set to `1`. Otherwise, code folding for `if`/`else` blocks is disabled.

Example: `s.matlab.editor.codefolding.FoldIfElseBlocks.TemporaryValue = 0`

`InitiallyFoldIfElseBlocks` — Initially fold `if`/`else` blocks
`0` (default) | `1`

Initially fold `if`/`else` blocks, specified as `1` or `0`.

Set to `1` to fold `if`/`else` blocks the first time that you open a MATLAB file.

Example: `s.matlab.editor.codefolding.InitiallyFoldIfElseBlocks.TemporaryValue = 1`

`FoldSpmdBlocks` — Fold `spmd` blocks
`1` (default) | `0`

Fold `spmd` blocks, specified as `1` or `0`.

Set to `1` to enable code folding for `spmd` (single program, multiple data) blocks. `EnableCodeFolding` must be set to `1`. Otherwise, code folding for `spmd` blocks is disabled.

Example: `s.matlab.editor.codefolding.FoldSpmdBlocks.TemporaryValue = 0`

`InitiallyFoldSpmdBlocks` — Initially fold `spmd` blocks
`0` (default) | `1`

Initially fold `spmd` blocks, specified as `1` or `0`.

Set to `1` to fold `spmd` (single program, multiple data) blocks the first time that you open a MATLAB file.

Example: `s.matlab.editor.codefolding.InitiallyFoldSpmdBlocks.TemporaryValue = 1`

`FoldSwitchCaseBlocks` — Fold `switch`/`case` blocks
`1` (default) | `0`

Fold `switch`/`case` blocks, specified as `1` or `0`.

Set to `1` to enable code folding for `switch`/`case` blocks. `EnableCodeFolding` must be set to `1`. Otherwise, code folding for `switch`/`case` blocks is disabled.

Example: `s.matlab.editor.codefolding.FoldSwitchCaseBlocks.TemporaryValue = 0`

`InitiallyFoldSwitchCaseBlocks` — Initially fold `switch`/`case` blocks
`0` (default) | `1`

Initially fold `switch`/`case` blocks, specified as `1` or `0`.

Set to `1` to fold `switch`/`case` blocks the first time that you open a MATLAB file.

Example: `s.matlab.editor.codefolding.InitiallyFoldSwitchCaseBlocks.TemporaryValue = 1`

`FoldTryCatchBlocks` — Fold `try`/`catch` blocks
`1` (default) | `0`

Fold `try`/`catch` blocks, specified as `1` or `0`.

Set to `1` to enable code folding for `try`/`catch` blocks. `EnableCodeFolding` must be set to `1`. Otherwise, code folding for `try`/`catch` blocks is disabled.

Example: `s.matlab.editor.codefolding.FoldTryCatchBlocks.TemporaryValue = 0`

`InitiallyFoldTryCatchBlocks` — Initially fold `try`/`catch` blocks
`0` (default) | `1`

Initially fold `try`/`catch` blocks, specified as `1` or `0`.

Set to `1` to fold `try`/`catch` blocks the first time that you open a MATLAB file.

Example: `s.matlab.editor.codefolding.InitiallyFoldTryCatchBlocks.TemporaryValue = 1`

`FoldWhileBlocks` — Fold `while` blocks
`1` (default) | `0`

Fold `while` blocks, specified as `1` or `0`.

Set to `1` to enable code folding for `while` blocks. `EnableCodeFolding` must be set to `1`. Otherwise, code folding for `while` blocks is disabled.

Example: `s.matlab.editor.codefolding.FoldWhileBlocks.TemporaryValue = 0`

`InitiallyFoldWhileBlocks` — Initially fold `while` blocks
`0` (default) | `1`

Initially fold `while` blocks, specified as `1` or `0`.

Set to `1` to fold `while` blocks the first time that you open a MATLAB file.

Example: `s.matlab.editor.codefolding.InitiallyFoldWhileBlocks.TemporaryValue = 1`

matlab.editor.backup

`EnableFileBackup` — Enable file backups

1 (default) | `0`

Enable file backups, specified as `1` or `0`.

Set to `1` to automatically save copies of the files you are currently editing.

Example: `s.matlab.editor.backup.EnableFileBackup.TemporaryValue = 0`

`Frequency` — How often to create backups of files

5 (default) | positive integer

How often to create backups of files you are editing, specified in minutes as a positive integer.

Example: `s.matlab.editor.backup.Frequency.TemporaryValue = 6`

`SaveUntiltled` — Create backups of untitled files

1 (default) | `0`

Create backups of untitled files, specified as `1` or `0`.

Set to `1` to save a copy of a new, untitled, file to `Untitled.asv`. When there is more than one untitled file, each additional file is saved to `Untitled`*n*`.asv` (where *n* is an integer value).

Example: `s.matlab.editor.backup.SaveUntitled.TemporaryValue = 0`

`DeleteOnClose` — Delete backup files on close

1 (default) | `0`

Delete backup files on close, specified as `1` or `0`.

Set to `1` to direct MATLAB to delete the backup file when you close the source file in the Editor.

Example: `s.matlab.editor.backup.DeleteOnClose.TemporaryValue = 0`

`FileExtensionToReplace` — File extension of backup file to replace
`'asv'` (default) | character vector

File extension of backup file to replace the original file name extension with, specified as a character vector.

Example: `s.matlab.editor.backup.FileExtensionToReplace.TemporaryValue = 'asv'`

`FileExtensionToAppend` — File extension of backup file to append
`'~'` (default) | character vector

File extension of backup file to append to the original file name, specified as a character vector.

Example: `s.matlab.editor.backup.FileExtensionToAppend.TemporaryValue = '~'`

`SaveIntoSourceFolder` — Save backup files to the source folder
`1` (default) | `0`

Save backup files to the source folder, specified as `1` or `0`.

Set to `1` to store backup files in the same folder as the files being edited. Otherwise, MATLAB saves the backup files in the folder specified by `SaveToFolder`.

Example: `s.matlab.editor.backup.SaveIntoSourceFolder.TemporaryValue = 0`

`SaveToFolder` — Folder name in which to store all backup files
character vector

Folder name in which to store all backup files, specified as a character vector.

Specify the full path to that folder and be sure that you have write permissions for it.

`SaveIntoSourceFolder` must be set to `0`. Otherwise, MATLAB stores backup files in the same folder as the files being edited.

Example: `s.matlab.editor.backup.SaveToFolder.TemporaryValue = 'C:\backups'`

matlab.editor.autoformat

`EnableAutoFormatting` — Enable autoformatting

`1` (default) | `0`

Enable autoformatting, specified as `1` or `0`.

Set to `1` to enable autoformatting in the Live Editor. Once enabled, specific types of formatting can then be enabled or disabled individually.

Example:
```
s.matlab.editor.autoformat.EnableAutoFormatting.TemporaryValue = 0
```

`AsteriskForItalicAndBold` — Enable italic and bold formatting using asterisks

`1` (default) | `0`

Enable italic and bold formatting using asterisks, specified as `1` or `0`.

Set to `1` to enable bold and italic formatting using asterisks (`*`).

Example:
```
s.matlab.editor.autoformat.AsteriskForItalicAndBold.TemporaryValue =
0
```

`UnderscoreForItalicAndBold` — Enable italic and bold formatting using underscores

`1` (default) | `0`

Enable italic and bold formatting using underscores, specified as `1` or `0`.

Set to `1` to enable bold and italic formatting using underscores (`_`).

Example:
```
s.matlab.editor.autoformat.UnderscoreForItalicAndBold.TemporaryValue
= 0
```

`BacktickAndBarForMonospace` — Enable monospace formatting using backticks or bars

`1` (default) | `0`

Enable monospace formatting using backticks or bars, specified as `1` or `0`.

Set to `1` to enable monospace formatting using backticks (`` ` ``) or bars (`|`).

Example: `s.matlab.editor.autoformat.BacktickAndBarForMonospace.TemporaryValue = 0`

`SectionBreak` — Enable autoformatting for section breaks
`1` (default) | `0`

Enable autoformatting for section breaks, specified as `1` or `0`.

Set to `1` to enable inserting section breaks by entering `%%`, `***`, or `---` and then **Enter**.

Example: `s.matlab.editor.autoformat.SectionBreak.TemporaryValue = 0`

`SectionBreakAndHeading` — Enable autoformatting for section breaks with headings
`1` (default) | `0`

Enable autoformatting for section breaks with headings, specified as `1` or `0`.

Set to `1` to enable inserting section breaks with headings by entering `%%`*`text`* and then **Enter**.

Example: `s.matlab.editor.autoformat.SectionBreakAndHeading.TemporaryValue = 0`

`DollarSignForLatex` — Enable autoformatting for LaTeX expressions
`1` (default) | `0`

Enable autoformatting for LaTeX expressions, specified as `1` or `0`.

Set to `1` to enable converting LaTeX expressions into equations using the format `$`*`LaTeX`*`$`.

Example: `s.matlab.editor.autoformat.DollarSignForLatex.TemporaryValue = 0`

`URLToHyperlink` — Enable autoformatting for URLs
`1` (default) | `0`

Enable autoformatting for URLs, specified as `1` or `0`.

Set to `1` to enable converting URLs automatically to hyperlinks.

Example: `s.matlab.editor.autoformat.URLToHyperlink.TemporaryValue = 0`

`AngleBracketsForHyperlink` — Enable autoformatting for URLs using angle brackets
`1` (default) | `0`

Enable autoformatting for URLs using angle brackets, specified as `1` or `0`.

Set to `1` to enable converting URLs to hyperlinks using the format *<URL>*.

Example: `s.matlab.editor.autoformat.AngleBracketsForHyperlink.TemporaryValue = 0`

`LabelURLToHyperlink` — Enable autoformatting for labeled hyperlinks
`1` (default) | `0`

Enable autoformatting for labeled hyperlinks, specified as `1` or `0`.

Set to `1` to enable converting URLs to labeled hyperlinks using the format `[`*`Label`*`]``(`*`URL`*`)`.

Example: `s.matlab.editor.autoformat.LabelURLToHyperlink.TemporaryValue = 0`

`TrademarkCopyrightSymbol` — Enable autoformatting for trademark and copyright symbols
`1` (default) | `0`

Enable autoformatting for trademark and copyright symbols, specified as `1` or `0`.

Set to `1` to enable inserting trademark and copyright symbols (™, ℠, ®, and ©) using the format `(TM)`, `(SM)`, `(R)`, and `(C)`.

Example: `s.matlab.editor.autoformat.TrademarkCopyrightSymbol.TemporaryValue = 0`

`HashtagForTitle` — Enable inserting titles using hashtags
`1` (default) | `0`

Enable inserting titles using hashtags, specified as `1` or `0`.

Set to `1` to enable inserting titles using the format `#`*`text`*.

Example: `s.matlab.editor.autoformat.HashtagForTitle.TemporaryValue = 0`

`HashtagsForHeading` — Enable inserting headings using hashtags
`1` (default) | `0`

Enable inserting headings using hashtags, specified as `1` or `0`.

Set to `1` to enable inserting headings using the format `##text` for heading 1, `###text` for heading 2, or `####text` for heading 3.

Example: `s.matlab.editor.autoformat.HashtagsForHeading.TemporaryValue = 0`

`AutomaticBulletedList` — Enable creating bulleted lists automatically
`1` (default) | `0`

Enable creating bulleted lists automatically, specified as `1` or `0`.

Set to `1` to enable creating bulleted lists by entering `*`, `+`, or `-` followed by a space.

Example: `s.matlab.editor.displaysettings.AutomaticBulletedList.TemporaryValue = 0`

`AutomaticNumberedList` — Enable creating numbered lists automatically
`1` (default) | `0`

Enable creating numbered lists automatically, specified as `1` or `0`.

Set to `1` to enable creating numbered lists by entering `1.`, `2.`, and so on, followed by a space.

Example: `s.matlab.editor.displaysettings.AutomaticNumberedList.TemporaryValue = 0`

matlab.editor.autocoding

`EnableAutoCoding` — Enable autocoding
`1` (default) | `0`

Enable autocoding, specified as `1` or `0`.

Set to `1` to enable autocoding in the Live Editor and in App Designer. Once enabled, options can be enabled or disabled individually.

Example: `s.matlab.editor.autocoding.EnableAutoCoding.TemporaryValue = 0`

`Parentheses` — Autocomplete parentheses
`1` (default) | `0`

Autocomplete parentheses, specified as `1` or `0`.

Set to `1` to enable autocompletion of parentheses.

Example: `s.matlab.editor.autocoding.Parentheses.TemporaryValue = 0`

`CurlyBraces` — Autocomplete curly braces
`1` (default) | `0`

Autocomplete curly braces, specified as `1` or `0`.

Set to `1` to enable autocompletion of curly braces.

Example: `s.matlab.editor.autocoding.CurlyBraces.TemporaryValue = 0`

`SquareBrackets` — Autocomplete square brackets
`1` (default) | `0`

Autocomplete square brackets, specified as `1` or `0`.

Set to `1` to enable autocompletion of square brackets.

Example: `s.matlab.editor.autocoding.SquareBrackets.TemporaryValue = 0`

`SingleQuotes` — Autocomplete single quotes
`1` (default) | `0`

Autocomplete single quotes, specified as `1` or `0`.

Set to `1` to enable autocompletion of single quotes.

Example: `s.matlab.editor.autocoding.SingleQuotes.TemporaryValue = 0`

`DoubleQuotes` — Autocomplete double quotes
`1` (default) | `0`

Autocomplete double quotes, specified as `1` or `0`.

Set to `1` to enable autocompletion of double quotes.

Example: `s.matlab.editor.autocoding.DoubleQuotes.TemporaryValue = 0`

`Comments` — Autocomplete comments
`1` (default) | `0`

Autocomplete comments, specified as `1` or `0`.

Set to `1` to enable autocompletion of comments when split across two lines.

Example: `s.matlab.editor.autocoding.Comments.TemporaryValue = 0`

`CharacterVectors` — Autocomplete character vectors
`1` (default) | `0`

Autocomplete character vectors, specified as `1` or `0`.

Set to `1` to enable autocompletion of character vectors when split across two lines.

Example: `s.matlab.editor.autocoding.CharacterVectors.TemporaryValue = 0`

`Strings` — Autocomplete strings
`1` (default) | `0`

Autocomplete strings, specified as `1` or `0`. Set to `1` to enable autocompletion of strings when split across two lines.

Example: `s.matlab.editor.autocoding.Strings.TemporaryValue = 0`

`SplitParentheses` — Autocomplete parentheses when splitting
`1` (default) | `0`

Autocomplete parentheses when splitting, specified as `1` or `0`. Set to `1` to enable autocompletion of parentheses when split across two lines.

Example: `s.matlab.editor.autocoding.Strings.TemporaryValue = 0`

`ControlFlows` — Autocomplete control flow statements
`1` (default) | `0`

Autocomplete control flow statements, specified as `1` or `0`. Set to `1` to enable autocompletion of control flow statements such as `if` and `for`.

Example: `s.matlab.editor.autocoding.ControlFlows.TemporaryValue = 0`

`Definitions` — Autocomplete function and class definitions

`1` (default) | `0`

Autocomplete function and class definitions, specified as `1` or `0`. Set to `1` to enable autocompletion of function and class definitions.

Example: `s.matlab.editor.autocoding.Definitions.TemporaryValue = 0`

matlab.editor.export

`FigureFormat` — Figure format

`'png'` (default) | `'jpeg'`

Figure format when exporting from the Live Editor, specified as `'png'` or `'jpeg'`. The value of `FigureFormat` is used when exporting to all document types except LaTeX. For LaTeX documents, the value of `matlab.editor.export.latex.FigureFormat` is used instead. To export with the specified figure format, run the live script before exporting.

Example: `s.matlab.editor.export.FigureFormat.TemporaryValue = 'jpeg'`

To change the format of exported figures for an individual export document type, specify a value for that document type's `FigureFormat` setting. For example, the following code specifies the JPEG figure format when exporting to PDF documents in the Live Editor, and the BMP figure format when exporting to all other document types:

```
s.matlab.editor.export.FigureFormat.TemporaryValue = 'png'
s.matlab.editor.export.pdf.FigureFormat.TemporaryValue = 'jpeg'
```

To return to using the active value of the `matlab.editor.export.FigureFormat` setting for all document types, clear the temporary and personal values for the individual export document type setting using the `clearTemporaryValue` and `clearPersonalValue` functions. For example:

```
clearTemporaryValue(s.matlab.editor.export.pdf.FigureFormat)
clearPersonalValue(s.matlab.editor.export.pdf.FigureFormat)
```

`FigureResolution` — Figure resolution

`600` (default) | positive integer

Figure resolution when exporting from the Live Editor, specified in DPI as a positive integer between 36 and 2880. To export with the specified figure resolution, run the live script before exporting.

Example: `s.matlab.editor.export.FigureResolution.TemporaryValue = 800`

To change the resolution of exported figures for an individual export type, specify a value for that document type's `FigureResolution` setting. For example, the following code specifies a figure resolution of 800 DPI when exporting to PDF documents in the Live Editor and a figure resolution of 1000 DPI when exporting to all other document types:

```
s.matlab.editor.export.FigureResolution.TemporaryValue = 1000
s.matlab.editor.export.pdf.FigureResolution.TemporaryValue = 800
```

To return to using the active value of the `matlab.editor.export.FigureResolution` setting for a document type, clear the temporary and personal values for that document type setting using the `clearTemporaryValue` and `clearPersonalValue` functions. For example:

```
clearTemporaryValue(s.matlab.editor.export.pdf.FigureResolution)
clearPersonalValue(s.matlab.editor.export.pdf.FigureResolution)
```

matlab.editor.export.pagesetup

PaperSize — Paper size

`'Letter'` | `'Legal'` | `'Tabloid'` | `'A2'` | `'A3'` | `'A4'` | `'A5'`

Paper size of documents when exporting from the Live Editor, specified as one of the standard paper type values in this table.

Value	Paper Size (Width x Height)
`'Letter'`	8.5-by-11 in (default in US)
`'Legal'`	8.5-by-14 in
`'Tabloid'`	11-by-17 in
`'A2'`	42-by-59.4 cm
`'A3'`	29.7-by-42 cm
`'A4'`	21-by-29.7 cm (default in Europe and Asia)
`'A5'`	14.8-by-21 cm

Example: `s.matlab.editor.export.pagesetup.PaperSize.TemporaryValue = 'Legal'`

To change the paper size for an individual export document type, specify a value for that document type's `PaperSize` setting. For example, the following code specifies the Legal paper size when exporting to PDF documents in the Live Editor, and the A4 paper size when exporting to all other document types:

```
s.matlab.editor.export.pagesetup.PaperSize.TemporaryValue = 'A4'
s.matlab.editor.export.pagesetup.pdf.PaperSize.TemporaryValue = 'Legal'
```

To return to using the active value of the `matlab.editor.export.pagesetup.PaperSize` setting for a document type, clear the temporary and personal values for that document type setting using the `clearTemporaryValue` and `clearPersonalValue` functions. For example:

```
clearTemporaryValue(s.matlab.editor.export.pagesetup.pdf.PaperSize)
clearPersonalValue(s.matlab.editor.export.pagesetup.pdf.PaperSize)
```

`Orientation` — Page orientation

`'Portrait'` (default) | `'Landscape'`

Page orientation of documents when exporting from the Live Editor, specified as `'Portrait'` or `'Landscape'`.

Example: `s.matlab.editor.export.pagesetup.Orientation.TemporaryValue = 'Landscape'`

To change the page orientation for an individual export document type, specify a value for that document type's `Orientation` setting. For example, the following code specifies a portrait page orientation when exporting to PDF documents in the Live Editor, and a landscape page orientation when exporting to all other document types:

```
s.matlab.editor.export.pagesetup.Orientation.TemporaryValue = 'Landscape'
s.matlab.editor.export.pagesetup.pdf.Orientation.TemporaryValue = 'Portrait'
```

To return to using the active value of the `matlab.editor.export.pagesetup.Orientation` setting for a document type, clear the temporary and personal values for that document type setting using the `clearTemporaryValue` and `clearPersonalValue` functions. For example:

```
clearTemporaryValue(s.matlab.editor.export.pagesetup.pdf.Orientation)
clearPersonalValue(s.matlab.editor.export.pagesetup.pdf.Orientation)
```

`Margins` — Document margins

`[72 72 72 72]` (default) | 1-by-4 vector

Document margins when exporting from the Live Editor, specified as a 1-by-4 vector of margin values with the format [*left top right bottom*]. Each margin value is specified in points as a positive integer between 0 and 255. A value of 72 points is equivalent to 1 inch or 2.54 cm. The value of `Margins` is used when exporting to all document types except LaTeX. For LaTeX documents, the value of `matlab.editor.export.pagesetup.latex.Margins` is used instead.

Example: `s.matlab.editor.export.pagesetup.Margins.TemporaryValue = [25 72 72 25]`

To change the document margins for an individual export document type, specify a value for that document type's `Margins` setting. For example, the following code specifies 25-point margins when exporting to PDF documents in the Live Editor, and 90-point margins when exporting to all other document types:

```
s.matlab.editor.export.pagesetup.Margins.TemporaryValue = [90 90 90 90]
s.matlab.editor.export.pagesetup.pdf.Margins.TemporaryValue = [25 25 25 25]
```

To return to using the active value of the `matlab.editor.export.pagesetup.Margins` setting for a document type, clear the temporary and personal values for that document type setting using the `clearTemporaryValue` and `clearPersonalValue` functions. For example:

```
clearTemporaryValue(s.matlab.editor.export.pagesetup.pdf.Margins)
clearPersonalValue(s.matlab.editor.export.pagesetup.pdf.Margins)
```

matlab.editor.export.pdf

`FigureFormat` — Figure format

`<no value>` (default) | `'png'` | `'jpeg'` | `'bmp'` | `'svg'` | `'tiff'`

Figure format when exporting to PDF documents from the Live Editor, specified as `'png'`, `'jpeg'`, `'bmp'`, `'svg'`, or `'tiff'`.

By default, the value of this setting is unset and the Live Editor uses the active value for the `matlab.editor.export.FigureFormat` setting when exporting to PDF documents. Specify a value for this setting to change the figure format when exporting to PDF documents only. To export with the specified figure format, run the live script before exporting.

Example: `s.matlab.editor.export.pdf.FigureFormat.TemporaryValue = 'jpeg'`

To return to using the active value of the `matlab.editor.export.FigureFormat` setting for PDF documents, clear the temporary and personal values for this setting using the `clearTemporaryValue` and `clearPersonalValue` functions. For example:

```
clearTemporaryValue(s.matlab.editor.export.pdf.FigureFormat)
clearPersonalValue(s.matlab.editor.export.pdf.FigureFormat)
```

`FigureResolution` — Figure resolution

`<no value>` (default) | positive integer

Figure resolution when exporting to PDF documents from the Live Editor, specified in DPI as a positive integer between 36 and 2880.

By default, the value of this setting is unset and the Live Editor uses the active value for the `matlab.editor.export.FigureResolution` setting when exporting to PDF documents. Specify a value for this setting to change the resolution of figures when exporting to PDF documents only. To export with the specified figure resolution, run the live script before exporting.

Example: `s.matlab.editor.export.pdf.FigureResolution.TemporaryValue = 800`

To return to using the active value of the `matlab.editor.export.FigureResolution` setting for PDF documents, clear the temporary and personal values for this setting using the `clearTemporaryValue` and `clearPersonalValue` functions. For example:

```
clearTemporaryValue(s.matlab.editor.export.pdf.FigureResolution)
clearPersonalValue(s.matlab.editor.export.pdf.FigureResolution)
```

matlab.editor.export.pagesetup.pdf

PaperSize — Paper size

`<no value>` (default) | `'Letter'` | `'Legal'` | `'Tabloid'` | `'A2'` | `'A3'` | `'A4'` | `'A5'`

Paper size of documents when exporting to PDF from the Live Editor, specified as one of the standard paper type values in this table.

Value	Paper Size (Width x Height)
`'Letter'`	8.5-by-11 in
`'Legal'`	8.5-by-14 in
`'Tabloid'`	11-by-17 in
`'A2'`	42-by-59.4 cm
`'A3'`	29.7-by-42 cm
`'A4'`	21-by-29.7 cm
`'A5'`	14.8-by-21 cm

By default, the value of this setting is unset and the Live Editor uses the active value for the `matlab.editor.export.pagesetup.PaperSize` setting when exporting to PDF documents. Specify a value for this setting to change the paper size of documents when exporting to PDF documents only.

Example: `s.matlab.editor.export.pagesetup.pdf.PaperSize.TemporaryValue = 'Legal'`

To return to using the active value of the `matlab.editor.export.pagesetup.PaperSize` setting for PDF documents, clear the temporary and personal values for this setting using the `clearTemporaryValue` and `clearPersonalValue` functions. For example:

```
clearTemporaryValue(s.matlab.editor.export.pagesetup.pdf.PaperSize)
clearPersonalValue(s.matlab.editor.export.pagesetup.pdf.PaperSize)
```

`Orientation` — Page orientation

`<no value>` (default) | `'Portrait'` | `'Landscape'`

Page orientation of documents when exporting to PDF from the Live Editor, specified as `'Portrait'` or `'Landscape'`.

By default, the value of this setting is unset and the Live Editor uses the active value for the `matlab.editor.export.pagesetup.Orientation` setting when exporting to PDF documents. Specify a value for this setting to change the page orientation of documents when exporting to PDF documents only.

Example: `s.matlab.editor.export.pagesetup.pdf.Orientation.TemporaryValue = 'Landscape'`

To return to using the active value of the `matlab.editor.export.pagesetup.Orientation` setting for PDF documents, clear the temporary and personal values for this setting using the `clearTemporaryValue` and `clearPersonalValue` functions. For example:

```
clearTemporaryValue(s.matlab.editor.export.pagesetup.pdf.Orientation)
clearPersonalValue(s.matlab.editor.export.pagesetup.pdf.Orientation)
```

`Margins` — Document margins

`<no value>` (default) | 1-by-4 vector

Document margins when exporting to PDF from the Live Editor, specified as a 1-by-4 vector of margin values with the format [*`left top right bottom`*]. Each margin value is specified in points as a positive integer between 0 and 255. A value of 72 points is equivalent to 1 inch or 2.54 cm.

By default, the value of this setting is unset and the Live Editor uses the active value for the `matlab.editor.export.pagesetup.Margins` setting when exporting to PDF

documents. Specify a value for this setting to change the margins of documents when exporting to PDF documents only.

Example: `s.matlab.editor.export.pagesetup.pdf.Margins.TemporaryValue = [20 28 28 20]`

To return to using the active value of the `matlab.editor.export.pagesetup.Margins` setting for PDF documents, clear the temporary and personal values for this setting using the `clearTemporaryValue` and `clearPersonalValue` functions. For example:

```
clearTemporaryValue(s.matlab.editor.export.pagesetup.pdf.Margins)
clearPersonalValue(s.matlab.editor.export.pagesetup.pdf.Margins)
```

matlab.editor.export.latex

`FigureFormat` — Figure format

`'eps'` | `'png'` | `'jpeg'` | `'pdf'`

Figure format when exporting to LaTeX documents from the Live Editor, specified as `'eps'`, `'png'`, `'jpeg'`, or `'pdf'`. By default, the value of this setting is `'eps'` or `'png'`, depending on whether the live script was run before exporting or not.

Specify a value for this setting to change the figure format when exporting to LaTeX documents only. The Live Editor always uses the figure format value of this setting when exporting to LaTeX instead of the `matlab.editor.export.FigureFormat` setting. To export with the specified figure format, run the live script before exporting.

Example: `s.matlab.editor.export.latex.FigureFormat.TemporaryValue = 'png'`

`FigureResolution` — Figure resolution

`<no value>` (default) | positive integer

Figure resolution when exporting to LaTeX documents from the Live Editor, specified in DPI as a positive integer between 36 and 2880.

By default, the value of this setting is unset and the Live Editor uses the active value for the `matlab.editor.export.FigureResolution` setting when exporting to LaTeX documents. Specify a value for this setting to change the resolution of figures when exporting to LaTeX documents only. To export with the specified figure resolution, run the live script before exporting.

Example: `s.matlab.editor.export.latex.FigureResolution.TemporaryValue = 800`

To return to using the active value of the `matlab.editor.export.FigureResolution` setting for LaTeX documents, clear the temporary and personal values for this setting using the `clearTemporaryValue` and `clearPersonalValue` functions. For example:

```
clearTemporaryValue(s.matlab.editor.export.latex.FigureResolution)
clearPersonalValue(s.matlab.editor.export.latex.FigureResolution)
```

matlab.editor.export.pagesetup.latex

PaperSize — Paper size

`<no value>` (default) | `'Letter'` | `'Legal'` | `'Tabloid'` | `'A2'` | `'A3'` | `'A4'` | `'A5'`

Paper size of documents when exporting to LaTeX from the Live Editor, specified as one of the standard paper type values in this table.

Value	Paper Size (Width x Height)
`'Letter'`	8.5-by-11 in
`'Legal'`	8.5-by-14 in
`'Tabloid'`	11-by-17 in
`'A2'`	42-by-59.4 cm
`'A3'`	29.7-by-42 cm
`'A4'`	21-by-29.7 cm
`'A5'`	14.8-by-21 cm

By default, the value of this setting is unset and the Live Editor uses the active value for the `matlab.editor.export.pagesetup.PaperSize` setting when exporting to LaTeX documents. Specify a value for this setting to change the paper size of documents when exporting to LaTeX documents only.

Example: `s.matlab.editor.export.pagesetup.latex.PaperSize.TemporaryValue = 'Legal'`

To return to using the active value of the `matlab.editor.export.pagesetup.PaperSize` setting for LaTeX documents, clear the temporary and personal values for this setting using the `clearTemporaryValue` and `clearPersonalValue` functions. For example:

```
clearTemporaryValue(s.matlab.editor.export.pagesetup.latex.PaperSize)
clearPersonalValue(s.matlab.editor.export.pagesetup.latex.PaperSize)
```

`Orientation` — Page orientation

`<no value>` (default) | `'Portrait'` | `'Landscape'`

Page orientation of documents when exporting to LaTeX from the Live Editor, specified as `'Portrait'` or `'Landscape'`.

By default, the value of this setting is unset and the Live Editor uses the active value for the `matlab.editor.export.pagesetup.Orientation` setting when exporting to LaTeX documents. Specify a value for this setting to change the page orientation of documents when exporting to LaTeX documents only.

Example: `s.matlab.editor.export.pagesetup.latex.Orientation.TemporaryValue = 'Landscape'`

To return to using the active value of the `matlab.editor.export.pagesetup.Orientation` setting for LaTeX documents, clear the temporary and personal values for this setting using the `clearTemporaryValue` and `clearPersonalValue` functions. For example:

```
clearTemporaryValue(s.matlab.editor.export.pagesetup.latex.Orientation)
clearPersonalValue(s.matlab.editor.export.pagesetup.latex.Orientation)
```

`Margins` — Document margins

`[134 125 134 120]` (default) | 1-by-4 vector

Document margins when exporting to LaTeX from the Live Editor, specified as a 1-by-4 vector of margin values with the format [*`left top right bottom`*]. Each margin value is specified in points as a positive integer between 0 and 255. A value of 72 points is equivalent to 1 inch or 2.54 cm.

Specify a value for this setting to change the margins of documents when exporting to LaTeX documents only. The Live Editor always uses the margin values of this setting when exporting to LaTeX instead of the `matlab.editor.export.pagesetup.Margins` setting.

Example: `s.matlab.editor.export.pagesetup.latex.Margins.TemporaryValue = [25 72 72 25]`

matlab.editor.export.pagesetup.docx

PaperSize — Paper size

<no value> (default) | 'Letter' | 'Legal' | 'Tabloid' | 'A2' | 'A3' | 'A4' | 'A5'

Paper size of documents when exporting to Microsoft Word from the Live Editor, specified as one of the standard paper type values in this table.

Value	Paper Size (Width x Height)
'Letter'	8.5-by-11 in
'Legal'	8.5-by-14 in
'Tabloid'	11-by-17 in
'A2'	42-by-59.4 cm
'A3'	29.7-by-42 cm
'A4'	21-by-29.7 cm
'A5'	14.8-by-21 cm

By default, the value of this setting is unset and the Live Editor uses the active value for the `matlab.editor.export.pagesetup.PaperSize` setting when exporting to Microsoft Word documents. Specify a value for this setting to change the paper size of documents when exporting to Microsoft Word documents only.

Example:
`s.matlab.editor.export.pagesetup.docx.PaperSize.TemporaryValue = 'Legal'`

To return to using the active value of the `matlab.editor.export.pagesetup.PaperSize` setting for Microsoft Word documents, clear the temporary and personal values for this setting using the `clearTemporaryValue` and `clearPersonalValue` functions. For example:

```
clearTemporaryValue(s.matlab.editor.export.pagesetup.docx.PaperSize)
clearPersonalValue(s.matlab.editor.export.pagesetup.docx.PaperSize)
```

Orientation — Page orientation

<no value> (default) | 'Portrait' | 'Landscape'

Page orientation of documents when exporting to Microsoft Word from the Live Editor, specified as `'Portrait'` or `'Landscape'`.

By default, the value of this setting is unset and the Live Editor uses the active value for the `matlab.editor.export.pagesetup.Orientation` setting when exporting to Microsoft Word documents. Specify a value for this setting to change the page orientation of documents when exporting to Microsoft Word documents only.

Example: `s.matlab.editor.export.pagesetup.docx.Orientation.TemporaryValue = 'Landscape'`

To return to using the active value of the `matlab.editor.export.pagesetup.Orientation` setting for Microsoft Word documents, clear the temporary and personal values for this setting using the `clearTemporaryValue` and `clearPersonalValue` functions. For example:

```
clearTemporaryValue(s.matlab.editor.export.pagesetup.docx.Orientation)
clearPersonalValue(s.matlab.editor.export.pagesetup.docx.Orientation)
```

`Margins` — Document margins

`<no value>` (default) | 1-by-4 vector

Document margins when exporting to Microsoft Word documents from the Live Editor, specified as a 1-by-4 vector of margin values with the format [*`left top right bottom`*]. Each margin value is specified in points as a positive integer between 0 and 255. A value of 72 points is equivalent to 1 inch or 2.54 cm.

By default, the value of this setting is unset and the Live Editor uses the active value for the `matlab.editor.export.pagesetup.Margins` setting when exporting to Microsoft Word documents. Specify a value for this setting to change the margins of documents when exporting to Microsoft Word documents only.

Example: `s.matlab.editor.export.pagesetup.docx.Margins.TemporaryValue = [25 72 72 25]`

To return to using the active value of the `matlab.editor.export.pagesetup.Margins` setting for Microsoft Word documents, clear the temporary and personal values for this setting using the `clearTemporaryValue` and `clearPersonalValue` functions. For example:

```
clearTemporaryValue(s.matlab.editor.export.pagesetup.docx.Margins)
clearPersonalValue(s.matlab.editor.export.pagesetup.docx.Margins)
```

See Also

`settings`

Topics

"Access and Modify Settings"
"Editor/Debugger Preferences"

Introduced in R2018a

eig

Eigenvalues and eigenvectors

Syntax

```
e = eig(A)
[V,D] = eig(A)
[V,D,W] = eig(A)

e = eig(A,B)
[V,D] = eig(A,B)
[V,D,W] = eig(A,B)

[ ___ ] = eig(A,balanceOption)
[ ___ ] = eig(A,B,algorithm)

[ ___ ] = eig( ___ ,eigvalOption)
```

Description

`e = eig(A)` returns a column vector containing the eigenvalues of square matrix `A`.

`[V,D] = eig(A)` returns diagonal matrix `D` of eigenvalues and matrix `V` whose columns are the corresponding right eigenvectors, so that `A*V = V*D`.

`[V,D,W] = eig(A)` also returns full matrix `W` whose columns are the corresponding left eigenvectors, so that `W'*A = D*W'`.

The eigenvalue problem is to determine the solution to the equation $Av = \lambda v$, where A is an `n`-by-`n` matrix, v is a column vector of length `n`, and λ is a scalar. The values of λ that satisfy the equation are the eigenvalues. The corresponding values of v that satisfy the equation are the right eigenvectors. The left eigenvectors, w, satisfy the equation $w'A = \lambda w'$.

`e = eig(A,B)` returns a column vector containing the generalized eigenvalues of square matrices `A` and `B`.

`[V,D] = eig(A,B)` returns diagonal matrix `D` of generalized eigenvalues and full matrix `V` whose columns are the corresponding right eigenvectors, so that `A*V = B*V*D`.

`[V,D,W] = eig(A,B)` also returns full matrix `W` whose columns are the corresponding left eigenvectors, so that `W'*A = D*W'*B`.

The generalized eigenvalue problem is to determine the solution to the equation $Av = \lambda Bv$, where A and B are `n`-by-`n` matrices, v is a column vector of length `n`, and λ is a scalar. The values of λ that satisfy the equation are the generalized eigenvalues. The corresponding values of v are the generalized right eigenvectors. The left eigenvectors, w, satisfy the equation $w'A = \lambda w'B$.

`[ ___ ] = eig(A,balanceOption)`, where `balanceOption` is `'nobalance'`, disables the preliminary balancing step in the algorithm. The default for `balanceOption` is `'balance'`, which enables balancing. The `eig` function can return any of the output arguments in previous syntaxes.

`[ ___ ] = eig(A,B,algorithm)`, where `algorithm` is `'chol'`, uses the Cholesky factorization of `B` to compute the generalized eigenvalues. The default for `algorithm` depends on the properties of `A` and `B`, but is generally `'qz'`, which uses the QZ algorithm.

If `A` is Hermitian and `B` is Hermitian positive definite, then the default for `algorithm` is `'chol'`.

`[ ___ ] = eig( ___ ,eigvalOption)` returns the eigenvalues in the form specified by `eigvalOption` using any of the input or output arguments in previous syntaxes. Specify `eigvalOption` as `'vector'` to return the eigenvalues in a column vector or as `'matrix'` to return the eigenvalues in a diagonal matrix.

Examples

Eigenvalues of Matrix

Use `gallery` to create a symmetric positive definite matrix.

```
A = gallery('lehmer',4)
```

```
A = 4×4

    1.0000    0.5000    0.3333    0.2500
```

```
    0.5000    1.0000    0.6667    0.5000
    0.3333    0.6667    1.0000    0.7500
    0.2500    0.5000    0.7500    1.0000
```

Calculate the eigenvalues of A. The result is a column vector.

```
e = eig(A)
```

```
e = 4×1

    0.2078
    0.4078
    0.8482
    2.5362
```

Alternatively, use `eigvalOption` to return the eigenvalues in a diagonal matrix.

```
D = eig(A,'matrix')
```

```
D = 4×4

    0.2078         0         0         0
         0    0.4078         0         0
         0         0    0.8482         0
         0         0         0    2.5362
```

Eigenvalues and Eigenvectors of Matrix

Use `gallery` to create a circulant matrix.

```
A = gallery('circul',3)
```

```
A = 3×3

     1     2     3
     3     1     2
     2     3     1
```

Calculate the eigenvalues and right eigenvectors of A.

```
[V,D] = eig(A)
```

```
V = 3×3 complex

  -0.5774 + 0.0000i   0.2887 - 0.5000i   0.2887 + 0.5000i
  -0.5774 + 0.0000i  -0.5774 + 0.0000i  -0.5774 + 0.0000i
  -0.5774 + 0.0000i   0.2887 + 0.5000i   0.2887 - 0.5000i


D = 3×3 complex

   6.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i  -1.5000 + 0.8660i   0.0000 + 0.0000i
   0.0000 + 0.0000i   0.0000 + 0.0000i  -1.5000 - 0.8660i
```

Verify that the results satisfy `A*V = V*D`.

```
A*V - V*D
```

```
ans = 3×3 complex
10^-14 ×

  -0.2665 + 0.0000i  -0.0333 + 0.1110i  -0.0333 - 0.1110i
   0.0888 + 0.0000i   0.0000 + 0.1221i   0.0000 - 0.1221i
  -0.0444 + 0.0000i  -0.0111 + 0.1221i  -0.0111 - 0.1221i
```

Ideally, the eigenvalue decomposition satisfies the relationship. Since `eig` performs the decomposition using floating-point computations, then `A*V` can, at best, approach `V*D`. In other words, `A*V - V*D` is close to, but not exactly, `0`.

Sorted Eigenvalues and Eigenvectors

By default `eig` does not always return the eigenvalues and eigenvectors in sorted order. Use the `sort` function to put the eigenvalues in ascending order and reorder the corresponding eigenvectors.

Calculate the eigenvalues and eigenvectors of a 5-by-5 magic square matrix.

```
A = magic(5)
```

```
A = 5×5

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

```
[V,D] = eig(A)
```

```
V = 5×5

   -0.4472    0.0976   -0.6330    0.6780   -0.2619
   -0.4472    0.3525    0.5895    0.3223   -0.1732
   -0.4472    0.5501   -0.3915   -0.5501    0.3915
   -0.4472   -0.3223    0.1732   -0.3525   -0.5895
   -0.4472   -0.6780    0.2619   -0.0976    0.6330


D = 5×5

   65.0000         0         0         0         0
         0  -21.2768         0         0         0
         0         0  -13.1263         0         0
         0         0         0   21.2768         0
         0         0         0         0   13.1263
```

The eigenvalues of `A` are on the diagonal of `D`. However, the eigenvalues are unsorted.

Extract the eigenvalues from the diagonal of `D` using `diag(D)`, then sort the resulting vector in ascending order. The second output from `sort` returns a permutation vector of indices.

```
[d,ind] = sort(diag(D))
```

```
d = 5×1

  -21.2768
  -13.1263
   13.1263
   21.2768
   65.0000
```

```
ind = 5×1

     2
     3
     5
     4
     1
```

Use `ind` to reorder the diagonal elements of `D`. Since the eigenvalues in `D` correspond to the eigenvectors in the columns of `V`, you must also reorder the columns of `V` using the same indices.

```
Ds = D(ind,ind)
```

```
Ds = 5×5

  -21.2768         0         0         0         0
         0  -13.1263         0         0         0
         0         0   13.1263         0         0
         0         0         0   21.2768         0
         0         0         0         0   65.0000
```

```
Vs = V(:,ind)
```

```
Vs = 5×5

    0.0976   -0.6330   -0.2619    0.6780   -0.4472
    0.3525    0.5895   -0.1732    0.3223   -0.4472
    0.5501   -0.3915    0.3915   -0.5501   -0.4472
   -0.3223    0.1732   -0.5895   -0.3525   -0.4472
   -0.6780    0.2619    0.6330   -0.0976   -0.4472
```

Both `(V,D)` and `(Vs,Ds)` produce the eigenvalue decomposition of `A`. The results of `A*V-V*D` and `A*Vs-Vs*Ds` agree, up to round-off error.

```
e1 = norm(A*V-V*D);
e2 = norm(A*Vs-Vs*Ds);
e = abs(e1 - e2)
```

```
e = 1.8933e-29
```

Left Eigenvectors

Create a 3-by-3 matrix.

```
A = [1 7 3; 2 9 12; 5 22 7];
```

Calculate the right eigenvectors, `V`, the eigenvalues, `D`, and the left eigenvectors, `W`.

```
[V,D,W] = eig(A)
```

```
V = 3×3

   -0.2610   -0.9734    0.1891
   -0.5870    0.2281   -0.5816
   -0.7663   -0.0198    0.7912


D = 3×3

   25.5548         0         0
         0   -0.5789         0
         0         0   -7.9759


W = 3×3

   -0.1791   -0.9587   -0.1881
   -0.8127    0.0649   -0.7477
   -0.5545    0.2768    0.6368
```

Verify that the results satisfy `W'*A = D*W'`.

```
W'*A - D*W'
```

```
ans = 3×3
10^-13 ×

    0.1155   -0.0711   -0.0711
   -0.0033   -0.0215   -0.0408
    0.0022    0.0266    0.0178
```

Ideally, the eigenvalue decomposition satisfies the relationship. Since `eig` performs the decomposition using floating-point computations, then `W'*A` can, at best, approach `D*W'`. In other words, `W'*A - D*W'` is close to, but not exactly, `0`.

Eigenvalues of Nondiagonalizable (Defective) Matrix

Create a 3-by-3 matrix.

```
A = [3 1 0; 0 3 1; 0 0 3];
```

Calculate the eigenvalues and right eigenvectors of A.

```
[V,D] = eig(A)
```

```
V = 3×3

    1.0000   -1.0000    1.0000
         0    0.0000   -0.0000
         0         0    0.0000


D = 3×3

     3     0     0
     0     3     0
     0     0     3
```

A has repeated eigenvalues and the eigenvectors are not independent. This means that A is not diagonalizable and is, therefore, defective.

Verify that V and D satisfy the equation, A*V = V*D, even though A is defective.

```
A*V - V*D
```

```
ans = 3×3
10^-15 ×

         0    0.8882   -0.8882
         0         0    0.0000
         0         0         0
```

Ideally, the eigenvalue decomposition satisfies the relationship. Since eig performs the decomposition using floating-point computations, then A*V can, at best, approach V*D. In other words, A*V - V*D is close to, but not exactly, 0.

Generalized Eigenvalues

Create two matrices, `A` and `B`, then solve the generalized eigenvalue problem for the eigenvalues and right eigenvectors of the pair `(A,B)`.

```
A = [1/sqrt(2) 0; 0 1];
B = [0 1; -1/sqrt(2) 0];
[V,D]=eig(A,B)
```

```
V = 2×2 complex

   1.0000 + 0.0000i   1.0000 + 0.0000i
   0.0000 - 0.7071i   0.0000 + 0.7071i


D = 2×2 complex

   0.0000 + 1.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i   0.0000 - 1.0000i
```

Verify that the results satisfy `A*V = B*V*D`.

```
A*V - B*V*D
```

```
ans = 2×2

     0     0
     0     0
```

The residual error `A*V - B*V*D` is exactly zero.

Generalized Eigenvalues Using QZ Algorithm for Badly Conditioned Matrices

Create a badly conditioned symmetric matrix containing values close to machine precision.

```
format long e
A = diag([10^-16, 10^-15])
```

```
A = 2×2

     1.000000000000000e-16                           0
                         0     1.000000000000000e-15
```

Calculate the generalized eigenvalues and a set of right eigenvectors using the default algorithm. In this case, the default algorithm is `'chol'`.

```
[V1,D1] = eig(A,A)
```

```
V1 = 2×2

     1.000000000000000e+08                           0
                         0     3.162277660168380e+07


D1 = 2×2

     9.999999999999999e-01                           0
                         0     1.000000000000000e+00
```

Now, calculate the generalized eigenvalues and a set of right eigenvectors using the `'qz'` algorithm.

```
[V2,D2] = eig(A,A,'qz')
```

```
V2 = 2×2

     1     0
     0     1


D2 = 2×2

     1     0
     0     1
```

Check how well the `'chol'` result satisfies `A*V1 = A*V1*D1`.

```
format short
A*V1 - A*V1*D1
```

```
ans = 2×2
10^-23 ×
```

```
    0.1654         0
         0   -0.6617
```

Now, check how well the `'qz'` result satisfies `A*V2 = A*V2*D2`.

```
A*V2 - A*V2*D2
```

```
ans = 2×2

     0     0
     0     0
```

When both matrices are symmetric, `eig` uses the `'chol'` algorithm by default. In this case, the QZ algorithm returns more accurate results.

Generalized Eigenvalues Where One Matrix is Singular

Create a 2-by-2 identity matrix, `A`, and a singular matrix, `B`.

```
A = eye(2);
B = [3 6; 4 8];
```

If you attempt to calculate the generalized eigenvalues of the matrix $B^{-1}A$ with the command `[V,D] = eig(B\A)`, then MATLAB® returns an error because `B\A` produces `Inf` values.

Instead, calculate the generalized eigenvalues and right eigenvectors by passing both matrices to the `eig` function.

```
[V,D] = eig(A,B)
```

```
V = 2×2

   -0.7500   -1.0000
   -1.0000    0.5000


D = 2×2
```

```
    0.0909         0
         0       Inf
```

It is better to pass both matrices separately, and let `eig` choose the best algorithm to solve the problem. In this case, `eig(A,B)` returns a set of eigenvectors and at least one real eigenvalue, even though `B` is not invertible.

Verify $Av = \lambda Bv$ for the first eigenvalue and the first eigenvector.

```
eigval = D(1,1);
eigvec = V(:,1);
A*eigvec - eigval*B*eigvec
```

```
ans = 2×1
10^-15 ×

    0.1110
    0.2220
```

Ideally, the eigenvalue decomposition satisfies the relationship. Since the decomposition is performed using floating-point computations, then `A*eigvec` can, at best, approach `eigval*B*eigvec`, as it does in this case.

Input Arguments

`A` — Input matrix
square matrix

Input matrix, specified as a real or complex square matrix.

Data Types: `double` | `single`
Complex Number Support: Yes

`B` — Generalized eigenvalue problem input matrix
square matrix

Generalized eigenvalue problem input matrix, specified as a square matrix of real or complex values. `B` must be the same size as `A`.

Data Types: `double` | `single`

Complex Number Support: Yes

balanceOption — Balance option
'balance' (default) | 'nobalance'

Balance option, specified as: 'balance', which enables a preliminary balancing step, or 'nobalance' which disables it. In most cases, the balancing step improves the conditioning of A to produce more accurate results. However, there are cases in which balancing produces incorrect results. Specify 'nobalance' when A contains values whose scale differs dramatically. For example, if A contains nonzero integers, as well as very small (near zero) values, then the balancing step might scale the small values to make them as significant as the integers and produce inaccurate results.

'balance' is the default behavior. For more information about balancing, see balance.

algorithm — Generalized eigenvalue algorithm
'chol' | 'qz'

Generalized eigenvalue algorithm, specified as 'chol' or 'qz', which selects the algorithm to use for calculating the generalized eigenvalues of a pair.

algorithm	Description
'chol'	Computes the generalized eigenvalues of A and B using the Cholesky factorization of B.
'qz'	Uses the QZ algorithm, also known as the generalized Schur decomposition. This algorithm ignores the symmetry of A and B.

In general, the two algorithms return the same result. The QZ algorithm can be more stable for certain problems, such as those involving badly conditioned matrices.

When you omit the algorithm argument, the eig function selects an algorithm based on the properties of A and B. It uses the 'chol' algorithm for symmetric (Hermitian) A and symmetric (Hermitian) positive definite B. Otherwise, it uses the 'qz' algorithm.

Regardless of the algorithm you specify, the eig function always uses the QZ algorithm when A or B are not symmetric.

eigvalOption — Eigenvalue option
'vector' | 'matrix'

Eigenvalue option, specified as `'vector'` or `'matrix'`. This option allows you to specify whether the eigenvalues are returned in a column vector or a diagonal matrix. The default behavior varies according to the number of outputs specified:

- If you specify one output, such as `e = eig(A)`, then the eigenvalues are returned as a column vector by default.
- If you specify two or three outputs, such as `[V,D] = eig(A)`, then the eigenvalues are returned as a diagonal matrix, `D`, by default.

Example: `D = eig(A,'matrix')` returns a diagonal matrix of eigenvalues with the one output syntax.

Output Arguments

e — Eigenvalues (returned as vector)
column vector

Eigenvalues, returned as a column vector containing the eigenvalues (or generalized eigenvalues of a pair) with multiplicity.

When `A` is real and symmetric or complex Hermitian, the values of `e` that satisfy $Av = \lambda v$ are real.

V — Right eigenvectors
square matrix

Right eigenvectors, returned as a square matrix whose columns are the right eigenvectors of `A` or generalized right eigenvectors of the pair, `(A,B)`. The form and normalization of `V` depends on the combination of input arguments:

- `[V,D] = eig(A)` returns matrix `V`, whose columns are the right eigenvectors of `A` such that `A*V = V*D`. The eigenvectors in `V` are normalized so that the 2-norm of each is 1.

  If `A` is real symmetric, then the right eigenvectors, `V`, are orthonormal.
- `[V,D] = eig(A,'nobalance')` also returns matrix `V`. However, the 2-norm of each eigenvector is not necessarily 1.
- `[V,D] = eig(A,B)` and `[V,D] = eig(A,B,algorithm)` returns `V` as a matrix whose columns are the generalized right eigenvectors that satisfy `A*V = B*V*D`. The

2-norm of each eigenvector is not necessarily 1. In this case, `D` contains the generalized eigenvalues of the pair, `(A,B)`, along the main diagonal.

When `eig` uses the `'chol'` algorithm with symmetric (Hermitian) `A` and symmetric (Hermitian) positive definite `B`, it normalizes the eigenvectors in `V` so that the `B`-norm of each is 1.

Different machines and releases of MATLAB can produce different eigenvectors that are still numerically accurate:

- For real eigenvectors, the sign of the eigenvectors can change.
- For complex eigenvectors, the eigenvectors can be multiplied by any complex number of magnitude 1.
- For a multiple eigenvalue, its eigenvectors can be recombined through linear combinations. For example, if $Ax = \lambda x$ and $Ay = \lambda y$, then $A(x+y) = \lambda(x+y)$, so $x+y$ also is an eigenvector of A.

`D` — Eigenvalues (returned as matrix)
diagonal matrix

Eigenvalues, returned as a diagonal matrix with the eigenvalues of `A` on the main diagonal or the eigenvalues of the pair, `(A,B)`, with multiplicity, on the main diagonal.

When `A` is real and symmetric or complex Hermitian, the values of `D` that satisfy $Av = \lambda v$ are real.

`W` — Left eigenvectors
square matrix

Left eigenvectors, returned as a square matrix whose columns are the left eigenvectors of `A` or generalized left eigenvectors of the pair, `(A,B)`. The form and normalization of `W` depends on the combination of input arguments:

- `[V,D,W] = eig(A)` returns matrix `W`, whose columns are the left eigenvectors of `A` such that `W'*A = D*W'`. The eigenvectors in `W` are normalized so that the 2-norm of each is 1. If `A` is symmetric, then `W` is the same as `V`.
- `[V,D,W] = eig(A,'nobalance')` also returns matrix `W`. However, the 2-norm of each eigenvector is not necessarily 1.
- `[V,D,W] = eig(A,B)` and `[V,D,W] = eig(A,B,algorithm)` returns `W` as a matrix whose columns are the generalized left eigenvectors that satisfy `W'*A = D*W'*B`. The

2-norm of each eigenvector is not necessarily 1. In this case, `D` contains the generalized eigenvalues of the pair, `(A,B)`, along the main diagonal.

If `A` and `B` are symmetric, then `W` is the same as `V`.

Different machines and releases of MATLAB can produce different eigenvectors that are still numerically accurate:

- For real eigenvectors, the sign of the eigenvectors can change.
- For complex eigenvectors, the eigenvectors can be multiplied by any complex number of magnitude 1.
- For a multiple eigenvalue, its eigenvectors can be recombined through linear combinations. For example, if $Ax = \lambda x$ and $Ay = \lambda y$, then $A(x+y) = \lambda(x+y)$, so $x+y$ also is an eigenvector of A.

Tips

- The `eig` function can calculate the eigenvalues of sparse matrices that are real and symmetric. To calculate the eigenvectors of a sparse matrix, or to calculate the eigenvalues of a sparse matrix that is not real and symmetric, use the `eigs` function.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `V` might represent a different basis of eigenvectors. The eigenvalues in `D` might not be in the same order as in MATLAB.
  - For the generalized eigenvalue problem, `[V,D] = eig(A,B)`, code generation always uses the `'qz'` option.
  - For the standard eigenvalue problem, `[V,D] = eig(A)`, when `A` is Hermitian, code generation uses `schur` to calculate `V` and `D`. Otherwise, the results of `[V,D] = eig(A)` are similar to the results obtained by using `[V,D] =`

`eig(A,eye(size(A)),'qz')` in MATLAB, except that the columns of `V` are normalized.

- The `'balance'`and `'nobalance'` options are not supported for the standard eigenvalue problem.
- The `'chol'` option is not supported for the symmetric generalized eigenvalue problem.
- The option to calculate left eigenvectors is not supported.
- Outputs are complex.
- When the input matrix contains a nonfinite value, the generated code does not issue an error. Instead, the output contains `NaN` values.
- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Only these one input argument syntaxes are supported:

  ```
  e = eig(A)
  ```

  ```
  [V,D] = eig(A)
  ```

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- For the generalized case, `eig(A,B)`, `A` and `B` must be real symmetric or complex Hermitian. Additionally, `B` must be positive definite.
- These syntaxes are not supported for full distributed arrays:

```
[__] = eig(A,'balance')
```

```
[__] = eig(A,B,'qz')
```

```
[V,D,W] = eig(A,B)
```

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`balance` | `condeig` | `eigs` | `hess` | `qz` | `schur`

Introduced before R2006a

eigs

Subset of eigenvalues and eigenvectors

Syntax

```
d = eigs(A)
d = eigs(A,k)
d = eigs(A,k,sigma)
d = eigs(A,k,sigma,Name,Value)
d = eigs(A,k,sigma,opts)

d = eigs(A,B, ___ )

d = eigs(Afun,n, ___ )

[V,D] = eigs( ___ )
[V,D,flag] = eigs( ___ )
```

Description

`d = eigs(A)` returns a vector of the six largest magnitude eigenvalues of matrix `A`. This is most useful when computing all of the eigenvalues with `eig` is computationally expensive, such as with large sparse matrices.

`d = eigs(A,k)` returns the `k` largest magnitude eigenvalues.

`d = eigs(A,k,sigma)` returns `k` eigenvalues based on the value of `sigma`. For example, `eigs(A,k,'smallestabs')` returns the `k` smallest magnitude eigenvalues.

`d = eigs(A,k,sigma,Name,Value)` specifies additional options with one or more name-value pair arguments. For example, `eigs(A,k,sigma,'Tolerance',1e-3)` adjusts the convergence tolerance for the algorithm.

`d = eigs(A,k,sigma,opts)` specifies options using a structure.

`d = eigs(A,B, ___ )` solves the generalized eigenvalue problem `A*V = B*V*D`. You can optionally specify `k`, `sigma`, `opts`, or name-value pairs as additional input arguments.

`d = eigs(Afun,n, ___ )` specifies a function handle `Afun` instead of a matrix. The second input `n` gives the size of matrix `A` used in `Afun`. You can optionally specify `B`, `k`, `sigma`, `opts`, or name-value pairs as additional input arguments.

`[V,D] = eigs( ___ )` returns diagonal matrix `D` containing the eigenvalues on the main diagonal, and matrix `V` whose columns are the corresponding eigenvectors. You can use any of the input argument combinations in previous syntaxes.

`[V,D,flag] = eigs( ___ )` also returns a convergence flag. If `flag` is `0`, then all the eigenvalues converged.

Examples

Largest Eigenvalues of Sparse Matrix

The matrix `A = delsq(numgrid('C',15))` is a symmetric positive definite matrix with eigenvalues reasonably well-distributed in the interval (0 8). Compute the six largest magnitude eigenvalues.

```
A = delsq(numgrid('C',15));
d = eigs(A)
```

```
d = 6×1

    7.8666
    7.7324
    7.6531
    7.5213
    7.4480
    7.3517
```

Specify a second input to compute a specific number of the largest eigenvalues.

```
d = eigs(A,3)
```

```
d = 3×1

    7.8666
    7.7324
    7.6531
```

Smallest Eigenvalues of Sparse Matrix

The matrix `A = delsq(numgrid('C',15))` is a symmetric positive definite matrix with eigenvalues reasonably well-distributed in the interval (0 8). Compute the five smallest eigenvalues.

```
A = delsq(numgrid('C',15));
d = eigs(A,5,'smallestabs')
```

```
d = 5×1

    0.1334
    0.2676
    0.3469
    0.4787
    0.5520
```

Eigenvalues Using Function Handle

Create a 1500-by-1500 random sparse matrix with a 25% approximate density of nonzero elements.

```
n = 1500;
A = sprand(n,n,0.25);
```

Find the LU factorization of the matrix, returning a permutation vector `p` that satisfies `A(p,:) = L*U`.

```
[L,U,p] = lu(A,'vector');
```

Create a function handle `Afun` that accepts a vector input `x` and uses the results of the LU decomposition to, in effect, return `A\x`.

```
Afun = @(x) U\(L\(x(p)));
```

Calculate the six smallest magnitude eigenvalues using `eigs` with the function handle `Afun`. The second input is the size of `A`.

```
d = eigs(Afun,1500,6,'smallestabs')
```

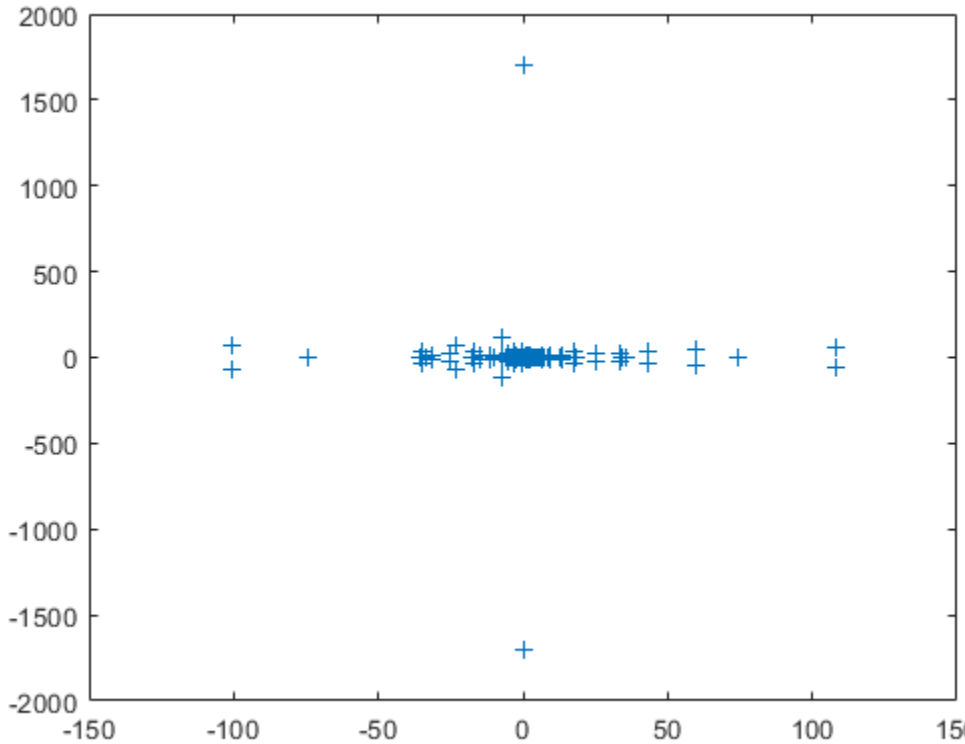
```
d = 6×1 complex

   0.1423 + 0.0000i
   0.4859 + 0.0000i
  -0.3323 - 0.3881i
  -0.3323 + 0.3881i
   0.1019 - 0.5381i
   0.1019 + 0.5381i
```

Types of Eigenvalues

`west0479` is a real-valued 479-by-479 sparse matrix with both real and complex pairs of conjugate eigenvalues.

Load the `west0479` matrix, then compute and plot all of the eigenvalues using `eig`. Since the eigenvalues are complex, `plot` automatically uses the real parts as the x-coordinates and the imaginary parts as the y-coordinates.

```
load west0479
A = west0479;
d = eig(full(A));
plot(d,'+')
```

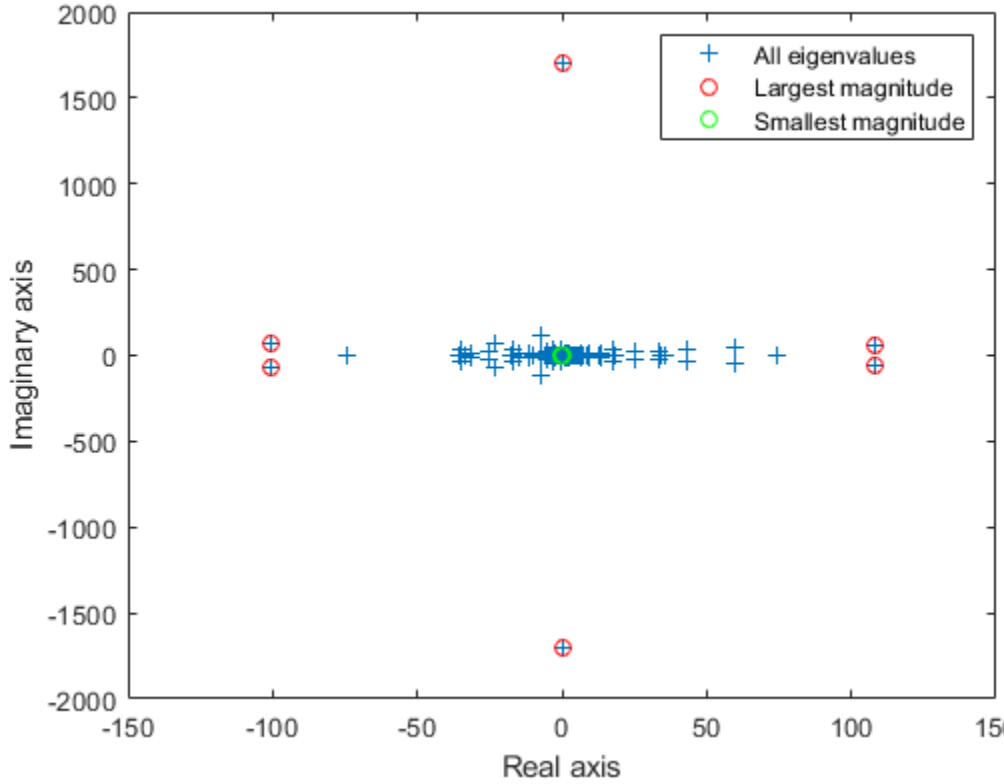


The eigenvalues are clustered along the real line (x-axis), particularly near the origin.

`eigs` has several options for `sigma` that can pick out the largest or smallest eigenvalues of varying types. Compute and plot some eigenvalues for each of the available options for `sigma`.

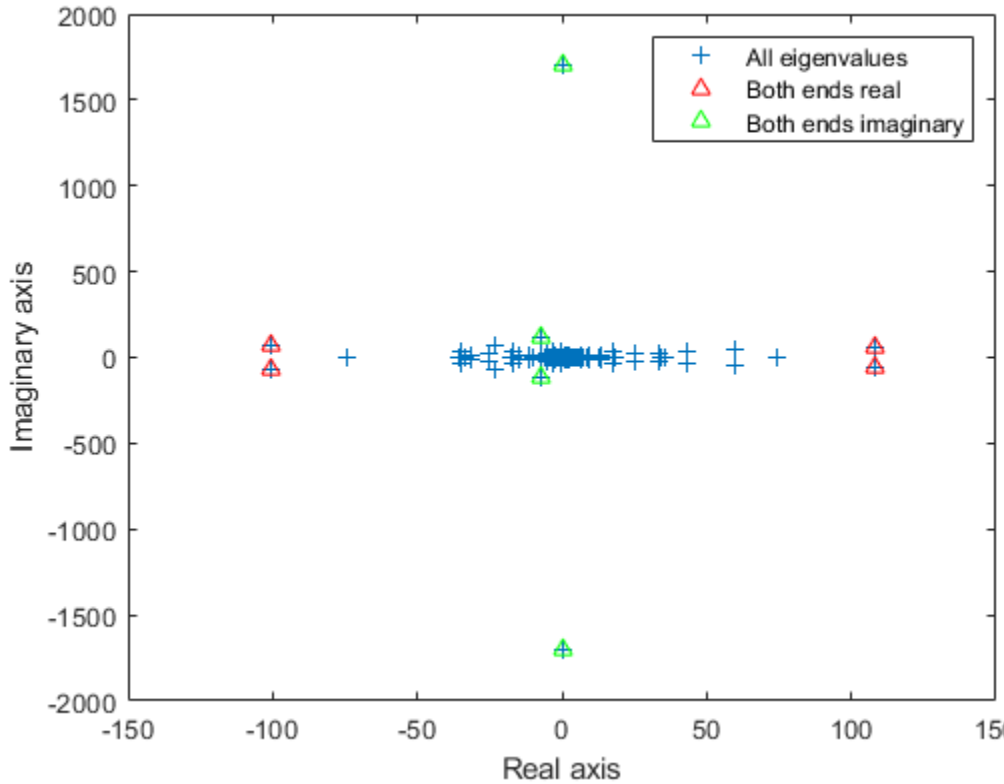
```
figure
plot(d, '+')
hold on
la = eigs(A,6,'largestabs');
plot(la,'ro')
sa = eigs(A,6,'smallestabs');
plot(sa,'go')
hold off
```

```
legend('All eigenvalues','Largest magnitude','Smallest magnitude')
xlabel('Real axis')
ylabel('Imaginary axis')
```



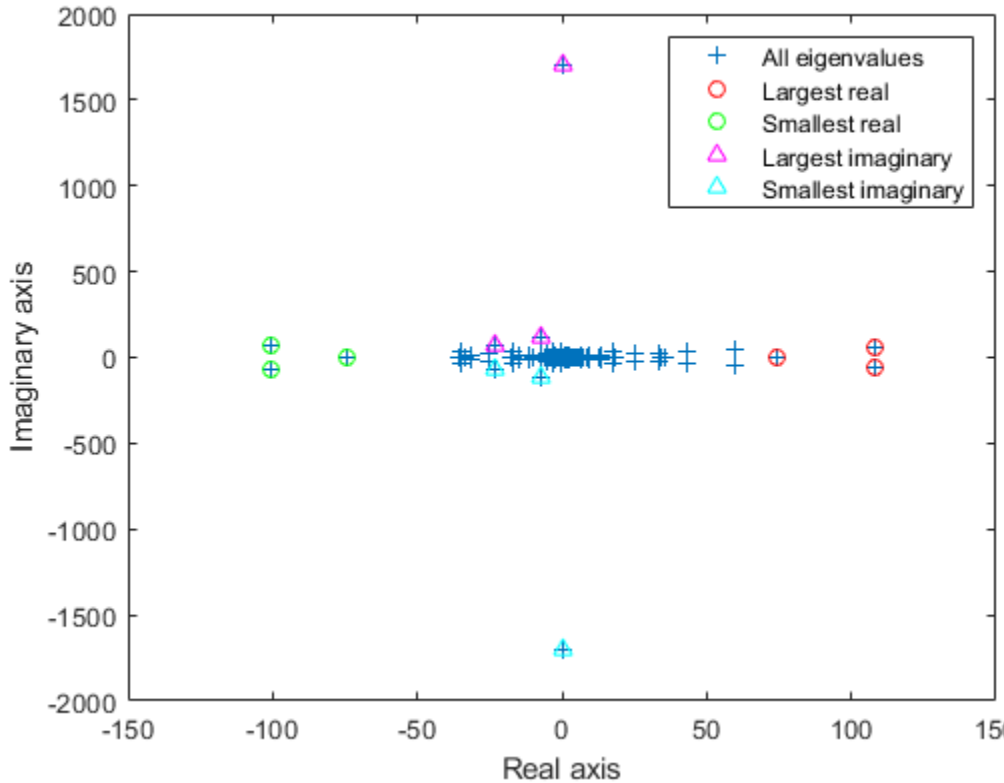

```
figure
plot(d, '+')
hold on
ber = eigs(A,4,'bothendsreal');
plot(ber,'r^')
bei = eigs(A,4,'bothendsimag');
plot(bei,'g^')
hold off
legend('All eigenvalues','Both ends real','Both ends imaginary')
```

```
xlabel('Real axis')
ylabel('Imaginary axis')
```




```
figure
plot(d, '+')
hold on
lr = eigs(A,3,'largestreal');
plot(lr,'ro')
sr = eigs(A,3,'smallestreal');
plot(sr,'go')
li = eigs(A,3,'largestimag','SubspaceDimension',45);
plot(li,'m^')
si = eigs(A,3,'smallestimag','SubspaceDimension',45);
plot(si,'c^')
```

```
hold off
legend('All eigenvalues','Largest real','Smallest real','Largest imaginary','Smallest i
xlabel('Real axis')
ylabel('Imaginary axis')
```



Difference Between `'smallestabs'` and `'smallestreal'` Eigenvalues

Create a symmetric positive definite sparse matrix.

```
A = delsq(numgrid('C', 150));
```

Compute the six smallest real eigenvalues using `'smallestreal'`, which employs a Krylov method using `A`.

```
tic
d = eigs(A, 6, 'smallestreal')
```

```
d = 6×1

    0.0013
    0.0025
    0.0033
    0.0045
    0.0052
    0.0063
```

```
toc
```

```
Elapsed time is 3.420533 seconds.
```

Compute the same eigenvalues using `'smallestabs'`, which employs a Krylov method using the inverse of `A`.

```
tic
dsm = eigs(A, 6, 'smallestabs')
```

```
dsm = 6×1

    0.0013
    0.0025
    0.0033
    0.0045
    0.0052
    0.0063
```

```
toc
```

```
Elapsed time is 0.399708 seconds.
```

The eigenvalues are clustered near zero. The `'smallestreal'` computation struggles to converge using `A` since the gap between the eigenvalues is so small. Conversely, the `'smallestabs'` option uses the inverse of `A`, and therefore the inverse of the eigenvalues of `A`, which have a much larger gap and are therefore easier to compute. This improved performance comes at the cost of factorizing `A`, which is not necessary with `'smallestreal'`.

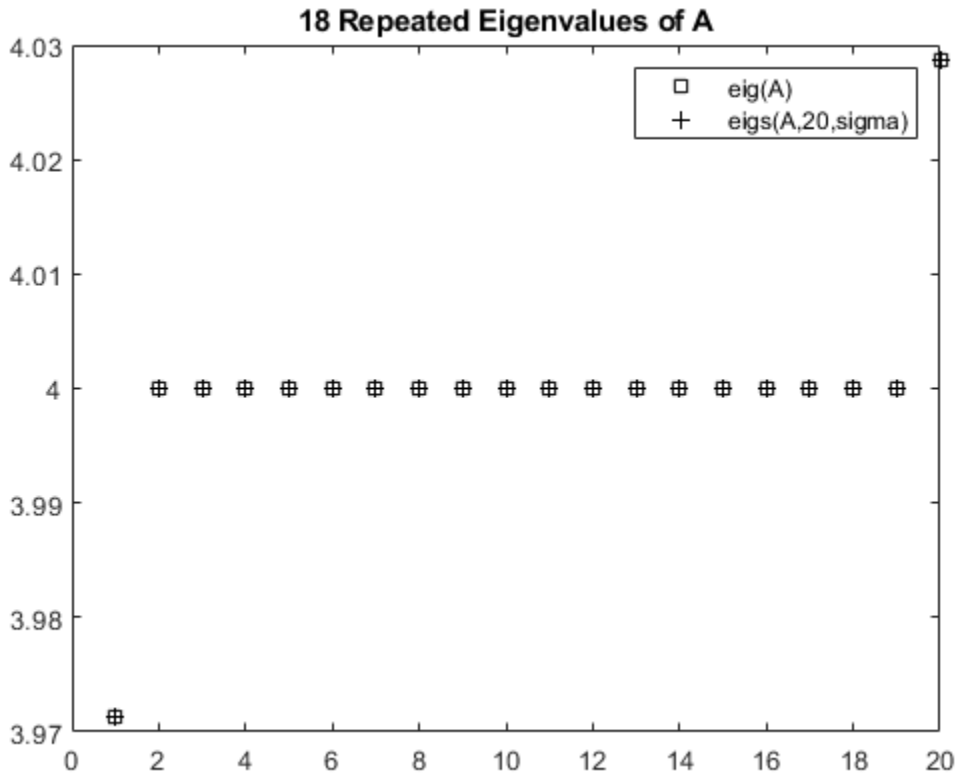
Sigma Value Near Eigenvalue

Compute eigenvalues near a numeric `sigma` value that is nearly equal to an eigenvalue.

The matrix `A = delsq(numgrid('C',30))` is a symmetric positive definite matrix of size 632 with eigenvalues reasonably well-distributed in the interval (0 8), but with 18 eigenvalues repeated at 4.0. To calculate some eigenvalues near 4.0, it is reasonable to try the function call `eigs(A,20,4.0)`. However, this call computes the largest eigenvalues of the inverse of `A - 4.0*I`, where `I` is an identity matrix. Because 4.0 is an eigenvalue of `A`, this matrix is singular and therefore does not have an inverse. `eigs` fails and produces an error message. The numeric value of `sigma` cannot be exactly equal to an eigenvalue. Instead, you must use a value of `sigma` that is near but not equal to 4.0 to find those eigenvalues.

Compute all of the eigenvalues using `eig`, and the 20 eigenvalues closest to 4 - 1e-6 using `eigs` to compare results. Plot the eigenvalues calculated with each method.

```
A = delsq(numgrid('C',30));
sigma = 4 - 1e-6;
d = eig(A);
D = sort(eigs(A,20,sigma));

plot(d(307:326),'ks')
hold on
plot(D,'k+')
hold off
legend('eig(A)','eigs(A,20,sigma)')
title('18 Repeated Eigenvalues of A')
```



Eigenvalues of Permuted Cholesky Factor

Create sparse random matrices A and B that both have low densities of nonzero elements.

```
B = sprandn(1e3,1e3,0.001) + speye(1e3);
B = B'*B;
A = sprandn(1e3,1e3,0.005);
A = A+A';
```

Find the Cholesky decomposition of matrix B, using three outputs to return the permutation vector s and test value p.

```
[R,p,s] = chol(B,'vector');
p
```

```
p = 0
```

Since p is zero, B is a symmetric positive definite matrix that satisfies B(s,s) = R'*R.

Calculate the six largest magnitude eigenvalues and eigenvectors of the generalized eigenvalue problem involving A and R. Since R is the Cholesky factor of B, specify 'IsCholesky' as true. Furthermore, since B(s,s) = R'*R and thus R = chol(B(s,s)), use the permutation vector s as the value of 'CholeskyPermutation'.

```
[V,D,flag] = eigs(A,R,6,'largestabs','IsCholesky',true,'CholeskyPermutation',s);
flag
```

```
flag = 0
```

Since flag is zero, all of the eigenvalues converged.

Input Arguments

A — Input matrix
matrix

Input matrix, specified as a square matrix. A is typically, but not always, a large and sparse matrix.

If A is symmetric, then eigs uses a specialized algorithm for that case. If A is *nearly* symmetric, then consider using A = (A+A')/2 to make A symmetric before calling eigs. This ensures that eigs calculates real eigenvalues instead of complex ones.

Data Types: double
Complex Number Support: Yes

B — Input matrix
matrix

Input matrix, specified as a square matrix of the same size as A. When B is specified, eigs solves the generalized eigenvalue problem A*V = B*V*D.

If B is symmetric positive definite, then `eigs` uses a specialized algorithm for that case. If B is *nearly* symmetric positive definite, then consider using `B = (B+B')/2` to make B symmetric before calling `eigs`.

When A is scalar, you can specify B as an empty matrix `eigs(A,[],k)` to solve the standard eigenvalue problem and disambiguate between B and k.

Data Types: `double`
Complex Number Support: Yes

k — Number of eigenvalues to compute
scalar

Number of eigenvalues to compute, specified as a positive scalar integer.

Example: `eigs(A,2)` returns the two largest eigenvalues of A.

`sigma` — Type of eigenvalues
`'largestabs'` (default) | `'smallestabs'` | `'largestreal'` | `'smallestreal'` | `'bothendsreal'` | `'largestimag'` | `'smallestimag'` | `'bothendsimag'` | scalar

Type of eigenvalues, specified as one of the values in the table.

sigma	Description	sigma (R2017a and earlier)
scalar (real or complex, including 0)	The eigenvalues closest to the number `sigma`.	No change
`'largestabs'` (default)	Largest magnitude.	`'lm'`
`'smallestabs'`	Smallest magnitude. Same as `sigma = 0`.	`'sm'`
`'largestreal'`	Largest real.	`'lr'`, `'la'`
`'smallestreal'`	Smallest real.	`'sr'`, `'sa'`
`'bothendsreal'`	Both ends, with `k/2` values with largest and smallest real part respectively (one more from high end if k is odd).	`'be'`

For **nonsymmetric problems**, `sigma` also can be:

sigma	Description	sigma (R2017a and earlier)
`'largestimag'`	Largest imaginary part.	`'li'` if `A` is complex.
`'smallestimag'`	Smallest imaginary part.	`'si'` if `A` is complex.
`'bothendsimag'`	Both ends, with `k/2` values with largest and smallest imaginary part (one more from high end if `k` is odd).	`'li'` if `A` is real.

Example: `eigs(A,k,1)` returns the `k` eigenvalues closest to 1.

Example: `eigs(A,k,'smallestabs')` returns the `k` smallest magnitude eigenvalues.

Data Types: `double` | `char` | `string`

`opts` — Options structure
structure

Options structure, specified as a structure containing one or more of the fields in this table.

Note Use of the options structure to specify options is not recommended. Use name-value pairs instead.

Option Field	Description	Name-Value Pair
`issym`	Symmetry of `Afun` matrix.	`'IsFunctionSymmetric'`
`tol`	Convergence tolerance.	`'Tolerance'`
`maxit`	Maximum number of iterations.	`'MaxIterations'`
`p`	Number of Lanczos basis vectors.	`'SubspaceDimension'`
`v0`	Starting vector.	`'StartVector'`
`disp`	Diagnostic information display level.	`'Display'`

Option Field	Description	Name-Value Pair
`fail`	Treatment of nonconverged eigenvalues in the output.	`'FailureTreatment'`
`spdB`	Is `B` symmetric positive definite?	`'IsSymmetricDefinite'`
`cholB`	Is `B` the Cholesky factor `chol(B)`?	`'IsCholesky'`
`permB`	Specify the permutation vector `permB` if sparse `B` is really `chol(B(permB,permB))`.	`'CholeskyPermutation'`

Example: `opts.issym = 1, opts.tol = 1e-10` creates a structure with values set for the fields `issym` and `tol`.

Data Types: `struct`

`Afun` — Matrix function
function handle

Matrix function, specified as a function handle. The function `y = Afun(x)` must return the proper value depending on the `sigma` input:

- `A*x` — If `sigma` is unspecified or any text option other than `'smallestabs'`.
- `A\x` — If `sigma` is `0` or `'smallestabs'`.
- `(A-sigma*I)\x` — If `sigma` is a nonzero scalar (for standard eigenvalue problem).
- `(A-sigma*B)\x` — If `sigma` is a nonzero scalar (for generalized eigenvalue problem).

For example, the following `Afun` works when calling `eigs` with `sigma = 'smallestabs'`:

```
[L,U,p] = lu(A,'vector');
Afun = @(x) U\(L\(x(p)));
d = eigs(Afun,100,6,'smallestabs')
```

For a generalized eigenvalue problem, add matrix `B` as follows (`B` cannot be represented by a function handle):

```
d = eigs(Afun,100,B,6,'smallestabs')
```

A is assumed to be nonsymmetric unless `'IsFunctionSymmetric'` (or `opts.issym`) specifies otherwise. Setting `'IsFunctionSymmetric'` to `true` ensures that `eigs` calculates real eigenvalues instead of complex ones.

For information on how to provide additional parameters to the `Afun` function, see "Parameterizing Functions".

Tip Call `eigs` with the `'Display'` option turned on to see what output is expected from `Afun`.

---

n — Size of square matrix represented by `Afun`
scalar

Size of square matrix A that is represented by `Afun`, specified as a positive scalar integer.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `d = eigs(A,k,sigma,'Tolerance',1e-10,'MaxIterations',100)` loosens the convergence tolerance and uses fewer iterations.

General Options

`Tolerance` — Convergence tolerance
`1e-14` (default) | positive real scalar

Convergence tolerance, specified as the comma-separated pair consisting of `'Tolerance'` and a positive real numeric scalar.

Example: `s = eigs(A,k,sigma,'Tolerance',1e-3)`

`MaxIterations` — Maximum number of algorithm iterations
`300` (default) | positive integer

Maximum number of algorithm iterations, specified as the comma-separated pair consisting of `'MaxIterations'` and a positive integer.

Example: `d = eigs(A,k,sigma,'MaxIterations',350)`

`SubspaceDimension` — Maximum size of Krylov subspace
`max(2*k,20)` (default) | nonnegative integer

Maximum size of Krylov subspace, specified as the comma-separated pair consisting of `'SubspaceDimension'` and a nonnegative integer. The `'SubspaceDimension'` value must be greater than or equal to `k + 1` for real symmetric problems, and `k + 2` otherwise, where `k` is the number of eigenvalues.

The recommended value is `p >= 2*k`, or for real nonsymmetric problems, `p >= 2*k+1`. If you do not specify a `'SubspaceDimension'` value, then the default algorithm uses at least `20` Lanczos vectors.

For problems where `eigs` fails to converge, increasing the value of `'SubspaceDimension'` can improve the convergence behavior. However, increasing the value too much can cause memory issues.

Example: `d = eigs(A,k,sigma,'SubspaceDimension',25)`

`StartVector` — Initial starting vector
random vector (default) | vector

Initial starting vector, specified as the comma-separated pair consisting of `'StartVector'` and a numeric vector.

The primary reason to specify a different random starting vector is when you want to control the random number stream used to generate the vector.

Note `eigs` selects the starting vectors in a reproducible manner using a private random number stream. Changing the random number seed does *not* affect the starting vector.

Example: `d = eigs(A,k,sigma,'StartVector',randn(m,1))` uses a random starting vector that draws values from the global random number stream.

Data Types: `double`

`FailureTreatment` — Treatment of nonconverged eigenvalues
`'replacenan'` (default) | `'keep'` | `'drop'`

Treatment of nonconverged eigenvalues, specified as the comma-separated pair consisting of `'FailureTreatment'` and one of the options: `'replacenan'`, `'keep'`, or `'drop'`.

The value of `'FailureTreatment'` determines how `eigs` displays nonconverged eigenvalues in the output.

Option	Affect on output
`'replacenan'`	Replace nonconverged eigenvalues with `NaN` values.
`'keep'`	Include nonconverged eigenvalues in the output.
`'drop'`	Remove nonconverged eigenvalues from the output. This option can result in `eigs` returning fewer eigenvalues than requested.

Example: `d = eigs(A,k,sigma,'FailureTreatment','drop')` removes nonconverged eigenvalues from the output.

Data Types: `char` | `string`

`Display` — Toggle for diagnostic information display
`false` or `0` (default) | `true` or `1`

Toggle for diagnostic information display, specified as the comma-separated pair consisting of `'Display'` and a numeric or logical `1` (`true`) or `0` (`false`). Specify a value of `true` or `1` to turn on the display of diagnostic information during the calculation.

Options for Afun

`IsFunctionSymmetric` — Symmetry of `Afun` matrix
`true` or `1` | `false` or `0`

Symmetry of `Afun` matrix, specified as the comma-separated pair consisting of `'IsFunctionSymmetric'` and a numeric or logical `1` (`true`) or `0` (`false`).

This option specifies whether the matrix that `Afun` applies to its input vector is symmetric. Specify a value of `true` or `1` to indicate that `eigs` should use a specialized algorithm for the symmetric matrix and return real eigenvalues.

Options for generalized eigenvalue problem A*V = B*V*D

`IsCholesky` — Cholesky decomposition toggle for `B`

`true` or `1` | `false` or `0`

Cholesky decomposition toggle for `B`, specified as the comma-separated pair consisting of `'IsCholesky'` and a numeric or logical `1` (`true`) or `0` (`false`).

This option specifies whether the input for matrix `B` in the call `eigs(A,B,___)` is actually the Cholesky factor `R` produced by `R = chol(B)`.

Note Do not use this option if `sigma` is `'smallestabs'` or a numeric scalar.

`CholeskyPermutation` — Cholesky permutation vector

`1:n` (default) | vector

Cholesky permutation vector, specified as the comma-separated pair consisting of `'CholeskyPermutation'` and a numeric vector. Specify the permutation vector `permB` if sparse matrix `B` is reordered before factorization according to `chol(B(permB,permB))`.

You also can use the three-output syntax of `chol` for sparse matrices to directly obtain `permB` with `[R,p,permB] = chol(B,'vector')`.

Note Do not use this option if `sigma` is `'smallestabs'` or a numeric scalar.

`IsSymmetricDefinite` — Symmetric-positive-definiteness toggle for `B`

`true` or `1` | `false` or `0`

Symmetric-positive-definiteness toggle for `B`, specified as the comma-separated pair consisting of `'IsSymmetricDefinite'` and a numeric or logical `1` (`true`) or `0` (`false`). Specify `true` or `1` when you know that `B` is symmetric positive definite, that is, it is a symmetric matrix with strictly positive eigenvalues.

If `B` is symmetric positive semi-definite (some eigenvalues are zero), then specifying `'IsSymmetricDefinite'` as `true` or `1` forces `eigs` to use the same specialized algorithm that it uses when `B` is symmetric positive definite.

Note To use this option, the value of `sigma` must be numeric or `'smallestabs'`.

Output Arguments

d — Eigenvalues
column vector

Eigenvalues, returned as a column vector. `d` is sorted differently depending on the value of `sigma`.

Value of `sigma`	Output sorting
`'largestabs'`	Descending order by magnitude
`'largestreal'`	Descending order by real part
`'largestimag'`	Descending order by imaginary part
`'smallestabs'`	Ascending order by magnitude
`'smallestreal'` `'bothendsreal'`	Ascending order by real part
`'smallestimag'`	Ascending order by imaginary part
`'bothendsimag'`	Descending order by absolute value of imaginary part

V — Eigenvectors
matrix

Eigenvectors, returned as a matrix. The columns in `V` correspond to the eigenvalues along the diagonal of `D`. The form and normalization of `V` depends on the combination of input arguments:

- `[V,D] = eigs(A)` returns matrix `V`, whose columns are the eigenvectors of `A` such that `A*V = V*D`. The eigenvectors in `V` are normalized so that the 2-norm of each is 1.

  If `A` is symmetric, then the eigenvectors, `V`, are orthonormal.
- `[V,D] = eigs(A,B)` returns `V` as a matrix whose columns are the generalized eigenvectors that satisfy `A*V = B*V*D`. The 2-norm of each eigenvector is not necessarily 1.

If `B` is symmetric positive definite, then the eigenvectors in `V` are normalized so that the `B`-norm of each is 1. If `A` is also symmetric, then the eigenvectors are `B`-orthonormal.

Different machines, releases of MATLAB, or parameters (such as the starting vector and subspace dimension) can produce different eigenvectors that are still numerically accurate:

- For real eigenvectors, the sign of the eigenvectors can change.
- For complex eigenvectors, the eigenvectors can be multiplied by any complex number of magnitude 1.
- For a multiple eigenvalue, its eigenvectors can be recombined through linear combinations. For example, if $Ax = \lambda x$ and $Ay = \lambda y$, then $A(x+y) = \lambda(x+y)$, so $x+y$ also is an eigenvector of A.

`D` — Eigenvalue matrix
matrix

Eigenvalue matrix, returned as a diagonal matrix with the eigenvalues on the main diagonal.

`flag` — Convergence flag
`0` | `1`

Convergence flag, returned as `0` or `1`. A value of `0` indicates that all the eigenvalues converged. Otherwise, not all of the eigenvalues converged.

Use of this convergence flag output suppresses warnings about failed convergence.

Tips

- `eigs` generates the default starting vector using a private random number stream to ensure reproducibility across runs. Setting the random number generator state using `rng` before calling `eigs` does not affect the output.
- Using `eigs` is not the most efficient way to find a few eigenvalues of small, dense matrices. For such problems, it might be quicker to use `eig(full(A))`. For example, finding three eigenvalues in a 500-by-500 matrix is a relatively small problem that is easily handled with `eig`.

- If `eigs` fails to converge for a given matrix, increase the number of Lanczos basis vectors by increasing the value of `'SubspaceDimension'`. As secondary options, adjusting the maximum number of iterations, `'MaxIterations'`, and the convergence tolerance, `'Tolerance'`, also can help with convergence behavior.

Compatibility Considerations

Behavior and algorithm change

Behavior changed in R2017b

- **Changes to sorting order of output**

  `eigs` now sorts the output according to the value of `sigma`. For example, the command `eigs(A,k,'largestabs')` produces `k` eigenvalues sorted in descending order by magnitude.

  Previously, the sorting order of the output produced by `eigs` was not guaranteed.

- **Reproducibility**

  Calling `eigs` multiple times in succession now produces the same result. Set `'StartVector'` to a random vector to change this behavior.

- **Display**

  A display value of `2` no longer returns timing information. Instead, `eigs` treats a value of `2` the same as a value of `1`. Also, the messages shown by the `'Display'` option have changed. The new messages show the residual in each iteration, instead of the Ritz values.

References

[1] Stewart, G.W. "A Krylov-Schur Algorithm for Large Eigenproblems." *SIAM Journal of Matrix Analysis and Applications*. Vol. 23, Issue 3, 2001, pp. 601–614.

[2] Lehoucq, R.B., D.C. Sorenson, and C. Yang. *ARPACK Users' Guide*. Philadelphia, PA: SIAM, 1998.

Extended Capabilities

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- For the syntax `[___] = eigs(A,B,k,sigma)`, if `B` is sparse and is not diagonal and not triangular, then `sigma` cannot be `'largesttabs'`, `'largestreal'`, `'smallestreal'`, `'bothendsreal'`, , `'largestimag'`, `'smallestimag'`, or `'bothendsimag'`
- For the syntax `[___] = eigs(A,B,k,sigma)`, if `A-sigma*B` is sparse and is not diagonal and not triangular, then `sigma` cannot be ,`'smallestabs'` or numeric.
- For the syntax `[___] = eigs(A,k,sigma)`, if `A` is sparse and is not diagonal and not triangular, then `sigma` cannot be ,`'smallestabs'` or numeric.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`eig` | `svds`

Introduced before R2006a

ellipj

Jacobi elliptic functions

Syntax

```
[SN,CN,DN] = ellipj(U,M)
[SN,CN,DN] = ellipj(U,M,tol)
```

Description

`[SN,CN,DN] = ellipj(U,M)` returns the Jacobi elliptic functions `SN`, `CN`, and `DN` evaluated for corresponding elements of argument `U` and parameter `M`. Inputs `U` and `M` must be the same size, or either `U` or `M` must be scalar.

`[SN,CN,DN] = ellipj(U,M,tol)` computes the Jacobi elliptic functions to accuracy `tol`. The default value of `tol` is `eps`. Increase `tol` for a less accurate but more quickly computed answer.

Examples

Find the Jacobi Elliptic Functions

Find the Jacobi elliptic functions for `U = 0.5` and `M = 0.25`.

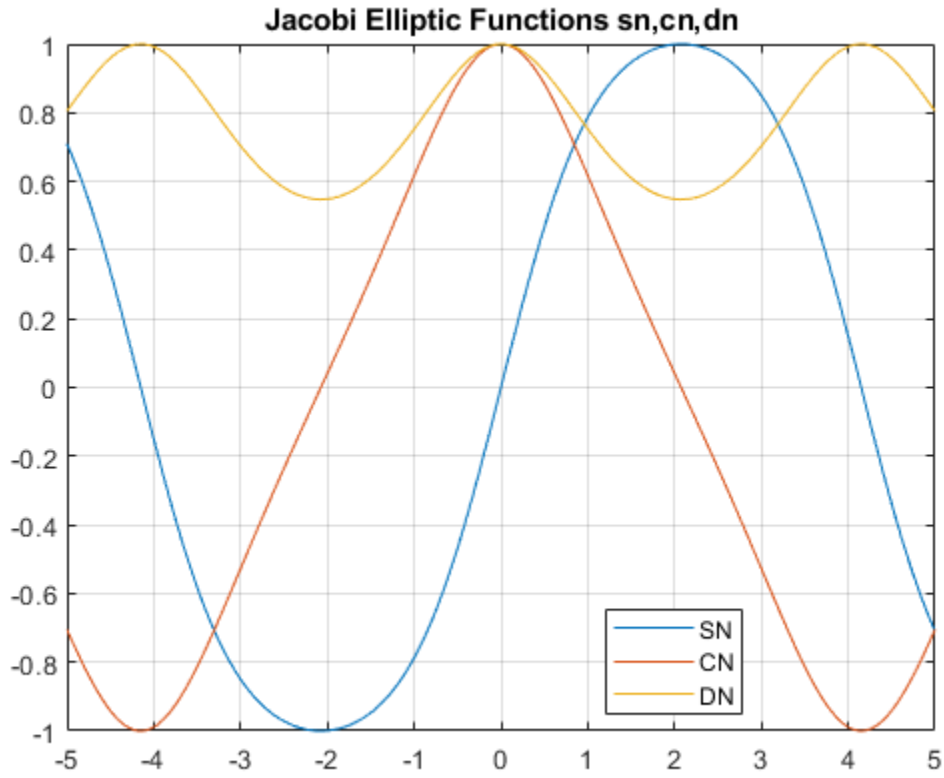
```
[s,c,d] = ellipj(0.5,0.25)
```

```
s = 0.4751

c = 0.8799

d = 0.9714
```

Plot the Jacobi Elliptic Functions

Plot the Jacobi elliptic functions for `-5≤U≤5` and `M = 0.7`.

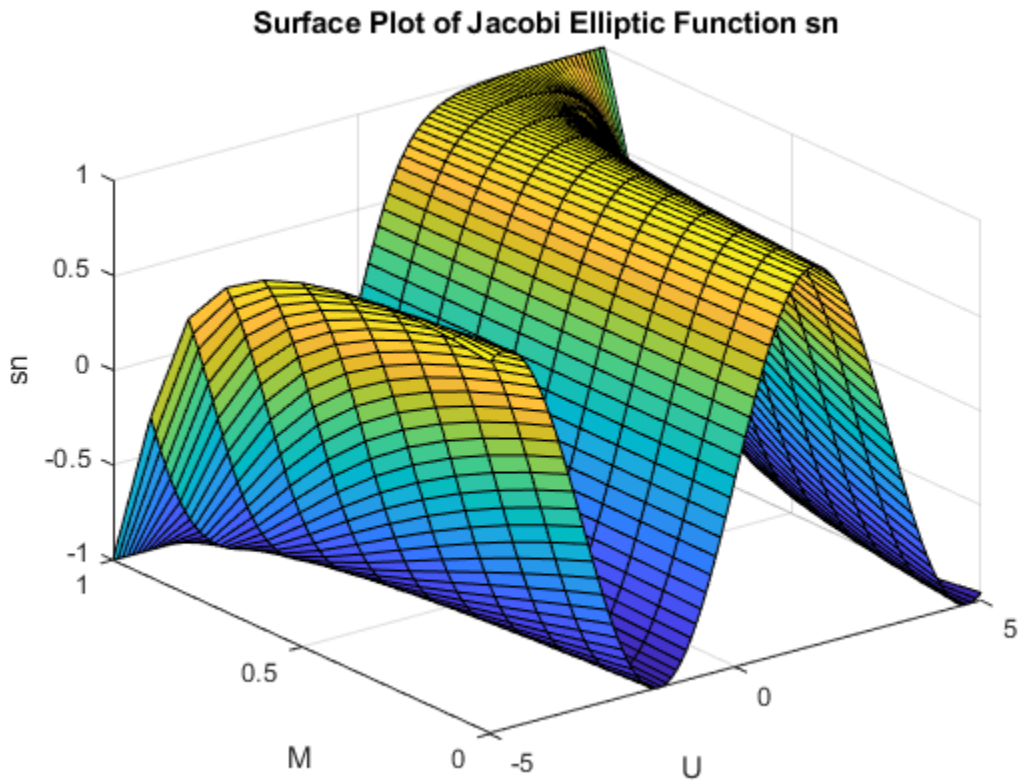
```
M = 0.7;
U = -5:0.01:5;
[S,C,D] = ellipj(U,M);
plot(U,S,U,C,U,D);
legend('SN','CN','DN','Location','best')
grid on
title('Jacobi Elliptic Functions sn,cn,dn')
```



Generate a Surface Plot of the Jacobi Elliptic sn Function

Generate a surface plot of the Jacobi elliptic sn function for the allowed range of `M` and `-5≤U≤5`.

```
[M,U] = meshgrid(0:0.1:1,-5:0.1:5);
S = ellipj(U,M);
surf(U,M,S)
xlabel('U')
ylabel('M')
zlabel('sn')
title('Surface Plot of Jacobi Elliptic Function sn')
```



Faster Calculations of Jacobi Elliptic Integrals by Changing Tolerance

The default value of `tol` is `eps`. Find the run time with the default value for arbitrary `M` using `tic` and `toc`. Increase `tol` by a factor of 1000 and find the run time. Compare the run times.

```
tic
ellipj(0.253,0.937)
```

```
ans = 0.2479
```

```
toc
```

```
Elapsed time is 0.065684 seconds.
```

```
tic
ellipj(0.253,0.937,eps*1000)
```

```
ans = 0.2479
```

```
toc
```

```
Elapsed time is 0.019649 seconds.
```

`ellipj` runs significantly faster when tolerance is significantly increased.

Input Arguments

`U` — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. `U` is limited to real values. If `U` is nonscalar, `M` must be a scalar or a nonscalar of the same size as `U`.

Data Types: `single` | `double`

`M` — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. `M` can take values $0 \leq m \leq 1$. If `M` is a nonscalar, `U` must be a scalar or a nonscalar of the same size as `M`. Map other values of `M` into this range using the transformations described in [1], equations 16.10 and 16.11.

Data Types: `single` | `double`

`tol` — Accuracy of result
`eps` (default) | nonnegative real number

Accuracy of result, specified as a nonnegative real number. The default value is `eps`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`SN` — Jacobi elliptic function sn
scalar | vector | matrix | multidimensional array

Jacobi elliptic function sn, returned as a scalar, vector, matrix, or multidimensional array.

`CN` — Jacobi elliptic function cn
scalar | vector | matrix | multidimensional array

Jacobi elliptic function cn, returned as a scalar, vector, matrix, or multidimensional array.

`DN` — Jacobi elliptic function dn
scalar | vector | matrix | multidimensional array

Jacobi elliptic function dn, returned as a scalar, vector, matrix, or multidimensional array.

More About

Jacobi Elliptic Functions

The Jacobi elliptic functions are defined in terms of the integral

$$u = \int_0^{\phi} \frac{d\theta}{\sqrt{1 - m\sin^2\theta}}.$$

Then

$$sn(u) = \sin\phi,\ cn(u) = \cos\phi,\ dn(u) = \sqrt{1 - m\sin^2\phi}.$$

Some definitions of the elliptic functions use the elliptical modulus k or modular angle α instead of the parameter m. They are related by

$$k^2 = m = \sin^2 a\,.$$

The Jacobi elliptic functions obey many mathematical identities. For a good sample, see [1].

Algorithms

`ellipj` computes the Jacobi elliptic functions using the method of the arithmetic-geometric mean of [1]. It starts with the triplet of numbers

$$a_0 = 1,\ b_0 = \sqrt{1-m},\ c_0 = \sqrt{m}\,.$$

`ellipj` computes successive iterations using

$$a_i = \frac{1}{2}(a_{i-1} + b_{i-1})$$

$$b_i = (^{a_i - 1}$$

$$c_i = \frac{1}{2}(a_{i-1} - b_{i-1})\,.$$

Next, it calculates the amplitudes in radians using

$$\sin(2\phi_{n-1} - \phi_n) = \frac{c_n}{a_n}\sin(\phi_n),$$

being careful to unwrap the phases correctly. The Jacobian elliptic functions are then simply

$$sn(u) = \sin\phi_0$$

$$cn(u) = \cos\phi_0$$

$$dn(u) = \sqrt{1 - m \cdot sn(u)^2}\,.$$

References

[1] Abramowitz, M. and I. A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, 1965, 17.6.

See Also

`ellipke`

Introduced before R2006a

ellipke

Complete elliptic integrals of first and second kind

Syntax

```
K = ellipke(M)
[K,E] = ellipke(M)
[K,E] = ellipke(M,tol)
```

Description

`K = ellipke(M)` returns the complete elliptic integral of the first kind for each element in `M`.

`[K,E] = ellipke(M)` returns the complete elliptic integral of the first and second kind.

`[K,E] = ellipke(M,tol)` computes the complete elliptic integral to accuracy `tol`. The default value of `tol` is `eps`. Increase `tol` for a less accurate but more quickly computed answer.

Examples

Find Complete Elliptic Integrals of First and Second Kind

Find the complete elliptic integrals of the first and second kind for `M = 0.5`.

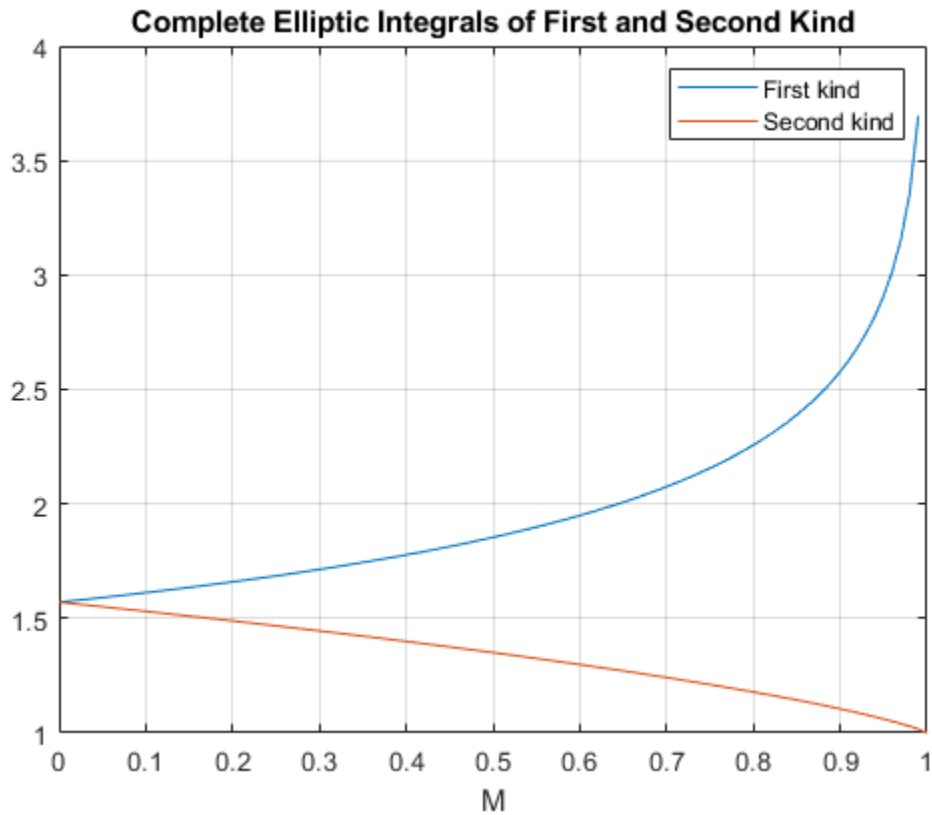
```
M = 0.5;
[K,E] = ellipke(M)
```

```
K = 1.8541

E = 1.3506
```

Plot Complete Elliptic Integrals of First and Second Kind

Plot the complete elliptic integrals of the first and second kind for the allowed range of `M`.

```
M = 0:0.01:1;
[K,E] = ellipke(M);
plot(M,K,M,E)
grid on
xlabel('M')
title('Complete Elliptic Integrals of First and Second Kind')
legend('First kind','Second kind')
```



Faster Calculations of the Complete Elliptic Integrals by Changing the Tolerance

The default value of `tol` is `eps`. Find the runtime with the default value for arbitrary `M` using `tic` and `toc`. Increase `tol` by a factor of thousand and find the runtime. Compare the runtimes.

```
tic
ellipke(0.904561)
```

```
ans = 2.6001
```

```
toc
```

```
Elapsed time is 0.029181 seconds.
```

```
tic
ellipke(0.904561,eps*1000)
```

```
ans = 2.6001
```

```
toc
```

```
Elapsed time is 0.010798 seconds.
```

`ellipke` runs significantly faster when tolerance is significantly increased.

Input Arguments

`M` — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. `M` is limited to values $0 \le m \le 1$.

Data Types: `single` | `double`

`tol` — Accuracy of result
`eps` (default) | nonnegative real number

Accuracy of result, specified as a nonnegative real number. The default value is `eps`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

K — Complete elliptic integral of first kind
scalar | vector | matrix | multidimensional array

Complete elliptic integral of the first kind, returned as a scalar, vector, matrix, or multidimensional array.

E — Complete elliptic integral of second kind
scalar | vector | matrix | multidimensional array

Complete elliptic integral of the second kind, returned as a scalar, vector, matrix, or multidimensional array.

More About

Complete Elliptic Integrals of the First and Second Kind

The complete elliptic integral of the first kind is

$$[K(m)] = \int_0^1 [(1 - t^2)(1 - mt^2)]^{-\frac{1}{2}} dt\,.$$

where m is the first argument of `ellipke`.

The complete elliptic integral of the second kind is

$$E(m) = \int_0^1 (1 - t^2)^{-\frac{1}{2}} (1 - mt^2)^{\frac{1}{2}} dt\,.$$

Some definitions of the elliptic functions use the elliptical modulus k or modular angle α instead of the parameter m. They are related by

$$k^2 = m = \sin^2\alpha\,.$$

References

[1] Abramowitz, M., and I. A. Stegun. *Handbook of Mathematical Functions*. Dover Publications, 1965.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`ellipj`

Introduced before R2006a

ellipsoid

Generate ellipsoid

Syntax

```
[x,y,z] = ellipsoid(xc,yc,zc,xr,yr,zr,n)
[x,y,z] = ellipsoid(xc,yc,zc,xr,yr,zr)
ellipsoid(axes_handle,...)
ellipsoid(...)
```

Description

`[x,y,z] = ellipsoid(xc,yc,zc,xr,yr,zr,n)` generates a surface mesh described by three `n+1`-by-`n+1` matrices, enabling `surf(x,y,z)` to plot an ellipsoid with center `(xc,yc,zc)` and semi-axis lengths `(xr,yr,zr)`.

`[x,y,z] = ellipsoid(xc,yc,zc,xr,yr,zr)` uses `n = 20`.

`ellipsoid(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

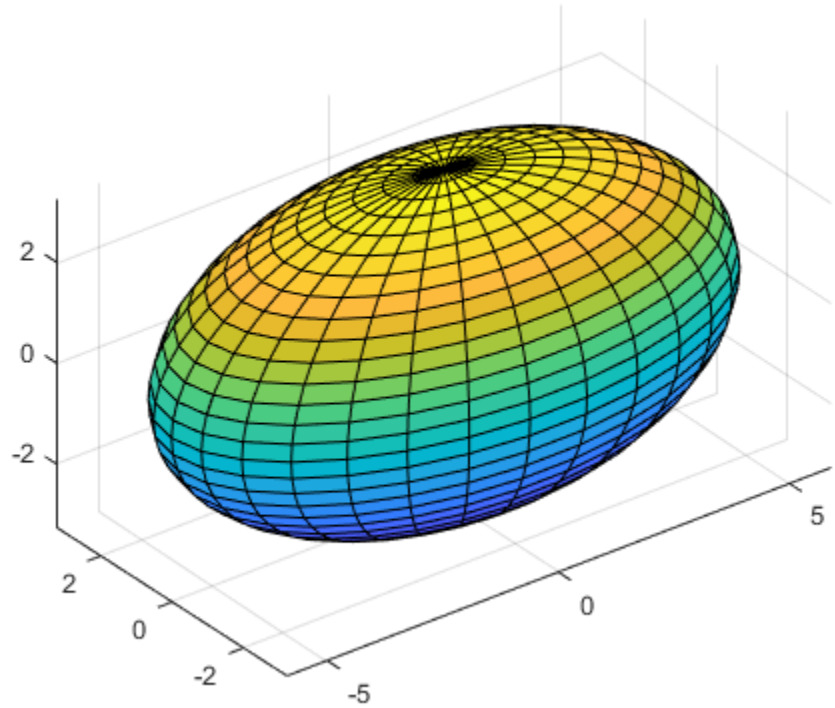
`ellipsoid(...)` with no output arguments plots the ellipsoid as a surface.

Examples

Surface Plot of Ellipsoid

Generate data for an ellipsoid with a center at (0,0,0) and semi-axis lengths (5.9,3.25,3.25). Use `surf` to plot the ellipsoid.

```
[x, y, z] = ellipsoid(0,0,0,5.9,3.25,3.25,30);
figure
surf(x, y, z)
axis equal
```



Algorithms

`ellipsoid` generates the data using the following equation:

$$\frac{(x-xc)^2}{xr^2}+\frac{(y-yc)^2}{yr^2}+\frac{(z-zc)^2}{zr^2}=1$$

Note that `ellipsoid(0,0,0,1,1,1)` is equivalent to a unit sphere.

See Also

`cylinder` | `sphere` | `surf`

Introduced before R2006a

empty

Create empty array of specified class

Syntax

```
A = ClassName.empty
A = ClassName.empty(sz1,...,szN)
A = ClassName.empty(sizeVector)
```

Description

Use *`ClassName`*`.empty` to create a 0-by-0 array of the *`ClassName`* class. Use *`ClassName`*`.empty(m,0)` to create an m-by-0 array of the *`ClassName`* class. This function is useful for creating empty arrays of data types that do not have a special syntax for creating empty arrays, such as `[]` for double arrays.

`A = ClassName.empty` returns an empty 0-by-0 array of the class named by `ClassName`. Replace `ClassName` with the actual name of the class.

`A = ClassName.empty(sz1,...,szN)` returns an empty array with the specified dimensions. At least one of the dimensions must be 0.

`A = ClassName.empty(sizeVector)` returns an empty array with the specified dimensions. At least one of the dimensions must be 0. Use this syntax to define an empty array that is the same size as an existing empty array. Pass the values returned by the `size` function as inputs.

Examples

Create Empty Character Array

This example shows how to create an empty character array using the default dimensions, 0-by-0.

```
A = char.empty
```

```
A =

  0x0 empty char array
```

Empty int16 Array with Nonzero Dimension

This example shows how to create an empty `int16` array with nonzero dimensions. Specify the 5-by-0 dimensions as inputs to the `empty` method.

```
Aint = int16.empty(5,0)
```

```
Aint =

  5x0 empty int16 matrix
```

Use Size of Existing Array

Use the size of an existing empty array to create an array of the same size.

```
Aint = int16.empty(5,0);
Bdouble = double.empty(size(Aint))
```

```
Bdouble =

  5x0 empty double matrix
```

Input Arguments

`sz1,...,szN` — Dimensions of array
0 (default) | integers

Dimensions of array, specified as integers. At least one dimension must be `0`. Negative values are treated as `0`. Trailing dimensions of `1` are not included in the size of the array

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`sizeVector` — Vector of dimensions
0 (default) | row vector

Vector of dimensions, specified as a row vector of nonnegative integers. At least one element must be `0`. Negative values are treated as `0`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Output Arguments

`A` — Empty array
empty array

Empty array, returned as an empty array of the specified dimensions and of the class used in the method invocation.

More About

Class of Empty Object Array

The `empty` method enables you to initialize arrays of a specific class:

```
C = char.empty(0,7)
```

```
C =

   Empty matrix: 0-by-7
```

```
disp(class(C))
char
```

Initializing an array with empty brackets (`[]`) produces an empty array of class `double`:

```
a = [];
```

```
disp(class(a))
double
```

You can initialize an empty array of a user-defined class. For example, the empty static method is a hidden method of the `RGBColor` class defined here.

```
classdef ColorInRGB
   properties
      Color (1,3) = [1,0,0];
   end
   methods
      function obj = ColorInRGB(c)
         if nargin > 0
            obj.Color = c;
         end
      end
   end
end
```

To create an empty 0-by-5 array of class `ColorInRGB`, call the empty method:

```
A = ColorInRGB.empty(0,5);
```

Identify Empty Arrays

You can use the `isempty`, `size`, and `length` functions to identify empty object arrays. For example, create an empty array of the `ColorInRGB` class defined in the previous section.

```
A = ColorInRGB.empty(0,5);
isempty(A)
```

```
ans =

  logical

   1
```

```
size(A)
```

```
ans =

     0     5
```

```
length(A)
```

```
ans =

     0
```

Concatenation and Indexing of Empty Arrays

Empty arrays follow array concatenation behavior. For example, create an empty array of the `ColorInRGB` class defined in the previous section and for a new array by concatenating instances into another array.

```
A = ColorInRGB.empty(0,5);
B = [A A]
```

```
B =

  0×10 ColorInRGB array with properties:

    Color
```

You cannot index into an empty array.

```
B(0,3)
```

```
Index in position 1 is invalid. Array indices must be positive integers or logical values.
```

Tips

- `empty` is a hidden, public, static method of all nonabstract MATLAB classes. You can override the `empty` method in class definitions.

See Also

`isempty` | `length` | `size`

Topics

"Empty Arrays"

Introduced in R2008a

enableNETfromNetworkDrive

Enable access to .NET commands from network drive

Syntax

```
enableNETfromNetworkDrive
```

Description

`enableNETfromNetworkDrive` adds an entry for the MATLAB interface to .NET module to the security policy on your machine. You must have administrative privileges to change your configuration.

Note Use `enableNETfromNetworkDrive` for MATLAB® releases R2012b or earlier, which support installed versions 2.0, 3.0, and 3.5 of the Microsoft® .NET Framework.

See Also

Topics

"Troubleshooting Security Policy Settings from Network Drives"

Introduced in R2009b

enableservice

Enable, disable, or report status of MATLAB Automation server

Syntax

```
state = enableservice('AutomationServer',enable)
state = enableservice('AutomationServer')
```

Description

`state = enableservice('AutomationServer',enable)` enables or disables the MATLAB Automation server. If `enable` is `true` (logical `1`), `enableservice` converts an existing MATLAB session into an Automation server. If `enable` is `false` (logical `0`), `enableservice` disables the MATLAB Automation server. `state` indicates the previous state of the Automation server. If `state = 1`, MATLAB was an Automation server. If `state = 0`, MATLAB was not an Automation server.

`state = enableservice('AutomationServer')` returns the current state of the Automation server. If `state` is logical `1` (`true`), MATLAB is an Automation server.

COM functions are available on Microsoft Windows systems only.

Examples

Enable the Automation server in the current MATLAB session:

```
state = enableservice('AutomationServer',true);
```

Show the current state of the MATLAB session. MATLAB displays `true`:

```
state = enableservice('AutomationServer')
```

Enable the Automation server and show the previous state. MATLAB displays `true`. The previous state can be the same as the current state:

```
state = enableservice('AutomationServer',true)
```

To enable the Automation server every time you run MATLAB, see “Manually Create Automation Server”.

See Also

`actxserver`

Topics

“MATLAB COM Automation Server Interface”

Introduced before R2006a

enableDefaultInteractivity

Enable built-in axes interactions

Syntax

```
enableDefaultInteractivity(ax)
```

Description

`enableDefaultInteractivity(ax)` enables the built-in interactions on page 1-3782 for the specified axes, if they are not already enabled. You can use this function to enable the default set of interactions or a custom set of interactions.

Examples

Disable and Reenable Default Axes Interactions

Create a plot. Some interactions are enabled by default, such as scrolling to zoom. Disable these default interactions.

```
plot(magic(10))
ax = gca;
disableDefaultInteractivity(ax)
```

Then, reenable the interactions.

```
enableDefaultInteractivity(ax)
```

Disable and Reenable Custom Axes Interactions

Create a plot, and replace the default set of axes interactions with just the rotate and zoom interactions. Then disable the interactions.

```
plot(magic(10))
ax = gca;
ax.Interactions = [rotateInteraction zoomInteraction];
disableDefaultInteractivity(ax)
```

Reenable the interactions you created.

```
enableDefaultInteractivity(ax)
```

Input Arguments

ax — Axes
`Axes` object | `PolarAxes` object | `GeographicAxes` object

Axes, specified as an `Axes`, `PolarAxes`, or `GeographicAxes` object.

More About

Built-In Axes Interactions

Built-in axes interactions allow you to explore charts using gestures, for example dragging to pan or scrolling to zoom. These interactions are built into the parent axes and are available without having to select any of the buttons in the axes toolbar. Most types of axes include a default set of built-in interactions. You can customize the default set of interactions on axes with an `Interactions` property.

If you do not want any of the interactions enabled, use the `disableDefaultInteractivity` function to disable them. To enable them if they are not already enabled, use the `enableDefaultInteractivity` function.

See Also

`disableDefaultInteractivity`

Topics

"Control Chart Interactivity"

Introduced in R2018b

end

Terminate block of code or indicate last array index

Syntax

end

Description

end is a keyword that terminates for, while, switch, try, if, and parfor statements. Without an end statement, for, while, switch, try, if, and parfor wait for further input. Each instance of end pairs with the closest previous unpaired for, while, switch, try, if, or parfor statement.

end also terminates a declared function and is required when:

- A function in a file contains one or more nested functions, in which case, every function in the file must be terminated with an end statement.
- A local function is within a function file, and all other local functions are terminated with an end statement.
- A local function is within a script file.

end also represents the last index of an array. For example, X(end) is the last element of X, and X(3:end) selects the third through final elements of X.

Examples

Terminate Conditional Statement and for Loop

Use end to close an if statement and a for loop. The first instance of end pairs with the if statement, and the second pairs with the for statement.

```
a = [0 0 1 1 0 0 0 1 0];
for k = 1:length(a)
```

```
    if a(k) == 0
        a(k) = 2;
    end
end
```

Terminate Switch Block

Use `end` to terminate a `switch` block.

```
choice = 1;

switch choice
    case 1
        disp('Vote for no. 1')
    case 2
        disp('Vote for no. 2')
    otherwise
        disp('Abstain')
end
```

```
Vote for no. 1
```

Declare Function

Declare a function in a file named `average.m` and save it in the current folder. Use `end` to terminate the function.

```
function ave = average(x)
    ave = sum(x(:))/numel(x);
end
```

The function accepts an input array, calculates the average of its elements, and returns a scalar. Call the function from the command line.

```
z = 1:99;
ave = average(z)
```

```
ave =

    50
```

Access Elements of Vector

Create a vector X.

```
X = 1:15;
```

Access the fifth through final elements of X.

```
X(5:end)
```

```
ans = 1×11

     5     6     7     8     9    10    11    12    13    14    15
```

Access the odd-index elements of X.

```
X(1:2:end)
```

```
ans = 1×8

     1     3     5     7     9    11    13    15
```

Access Last Row of Matrix

Access the last row of a matrix A using end.

```
A = magic(3)
```

```
A = 3×3

     8     1     6
     3     5     7
```

```
     4     9     2
```

```
B = A(end,:)
```

```
B = 1×3
     4     9     2
```

Tips

- Classes can overload the `end` function to implement specialized behavior. For more information, see “end as Object Index”.
- If an array `X` already exists, you can use `end` to grow the array. For example, `X(end +1) = 5` increases the length of `X` by 1 and adds a new element to the end of `X`.
- Although `end` is sometimes optional in a function file, use it for better code readability.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`break` | `for` | `function` | `if` | `parfor` | `return` | `switch` | `try` | `while`

Topics

“Types of Functions”

Introduced before R2006a

EndInvoke

Retrieve result of asynchronous call initiated by .NET System.Delegate BeginInvoke method

Syntax

```
result = EndInvoke(asyncResult)
[res0,...,resN] = EndInvoke(res0,...,resN,asyncResult)
```

Description

`result = EndInvoke(asyncResult)` retrieves result of asynchronous call initiated by `BeginInvoke` method.

`[res0,...,resN] = EndInvoke(res0,...,resN,asyncResult)` for methods with `out` and/or `ref` parameters.

Input Arguments

`asyncResult`

.NET System.IAsyncResult object returned by `BeginInvoke`.

`res0,...,resN`

For methods with `out` and/or `ref` parameters, results of the asynchronous call. The number of arguments is the sum of:

- Number of return values (`0` or `1`).
- Number of `out` and `ref` arguments.

Output Arguments

result

Results of the asynchronous call.

res0,...,resN

For methods with `out` and/or `ref` parameters, results of the asynchronous call,

Examples

The following examples show how to call delegates with various input and output arguments. Each example contains:

1. The C# delegate signature. In order to execute the MATLAB code, build the delegate code into an assembly named `SignatureExamples` and load it into MATLAB. For information, see "Build a .NET Application for MATLAB Examples".
2. An example MATLAB function to use with the delegate, which must exist on your path.
3. The `BeginInvoke` and `EndInvoke` signatures MATLAB creates. To display the signatures, create a delegate instance, `myDel`, and call the `methodsview` function.
4. Simple MATLAB example.

This example shows how to use a delegate that has no return value.

1. C# delegate:

   ```
   public delegate void delint(Int32 arg);
   ```

2. MATLAB function to call:

   ```
   % Display input argument
   function dispfnc(A)
   % A = number
   ['Input is ' num2str(A)]
   end
   ```

3. MATLAB creates the following signatures. For `BeginInvoke`:

   ```
   System.IAsyncResult RetVal
       BeginInvoke    (
   ```

```
        SignatureExamples.delint this,
        int32 scalar arg,
        System.AsyncCallback callback,
        System.Object object)
```

The `EndInvoke` signature:

```
    EndInvoke    (
        SignatureExamples.delint this,
        System.IAsyncResult result)
```

4 Call `dispfnc`:

```
myDel = SignatureExamples.delint(@dispfnc);
asyncRes = myDel.BeginInvoke(6, [], []);
while asyncRes.IsCompleted ~= true
    pause(0.05) % Use pause() to let MATLAB process event
end
myDel.EndInvoke(asyncRes)
```

```
Input is 6
```

This example shows how to use a delegate with a return value. The delegate does not have `out` or `ref` parameters.

1 C# delegate:

```
public delegate Int32 del2int(Int32 arg1, Int32 arg2);
```

2 MATLAB function to call:

```
% Add input arguments
function res = addfnc(A, B)
% A and B are numbers
res = A + B;
end
```

3 MATLAB creates the following signatures. For `BeginInvoke`:

```
System.IAsyncResult RetVal
    BeginInvoke    (
        SignatureExamples.del2int this,
        int32 scalar arg1,
        int32 scalar arg2,
        System.AsyncCallback callback,
        System.Object object)
```

The `EndInvoke` signature:

```
int32 scalar RetVal
    EndInvoke    (
        SignatureExamples.del2int this,
        System.IAsyncResult result)
```

4 Call `addfnc`.

```
myDel = SignatureExamples.del2int(@addfnc);
asyncRes = myDel.BeginInvoke(6,8,[],[]);
while asyncRes.IsCompleted ~= true
    pause(0.05) % Use pause() to let MATLAB process event
end
result = myDel.EndInvoke(asyncRes)
```

```
result =
          14
```

This example shows how to use a delegate with a `ref` parameter, `refArg`, and no return value.

1 C# delegate:

```
public delegate void delrefvoid(ref Double refArg);
```

2 MATLAB maps the `ref` argument as both RHS and LHS arguments. MATLAB function to call.

```
% Increment input argument
function res = incfnc(A)
% A = number
res = A + 1;
end
```

3 MATLAB creates the following signatures. For `BeginInvoke`:

```
[System.IAsyncResult RetVal,
double scalar refArg]
    BeginInvoke    (
        SignatureExamples.delrefvoid this,
        double scalar refArg,
        System.AsyncCallback callback,
        System.Object object)
```

The `EndInvoke` signature:

```
double scalar refArg
    EndInvoke    (
```

```
        SignatureExamples.delrefvoid this,
        double scalar refArg,
        System.IAsyncResult result)
```

4 Call `incfnc`.

```
x = 6;
myDel = SignatureExamples.delrefvoid(@incfnc);
asyncRes = myDel.BeginInvoke(x,[],[]);
while asyncRes.IsCompleted ~= true
    pause(0.05) % Use pause() to let MATLAB process event
end
myRef = 0;
result = myDel.EndInvoke(myRef,asyncRes);
disp(['Increment of ' num2str(x) ' = ' num2str(result)]);
```

```
Increment of 6 = 7
```

This example shows how to use a delegate with an `out` parameter, `argOut`, and one return value.

1 C# delegate:

```
public delegate Single deloutsingle(Single argIn, out Single argOut);
```

2 MATLAB maps the `out` argument as a return value for a total of two return values. MATLAB function to call.

```
% Double input argument
function [res1,res2] = times2fnc(A)
res1 = A*2;
res2 = res1;
end
```

3 MATLAB creates the following signatures. For `BeginInvoke`.

```
[System.IAsyncResult RetVal,
single scalar argOut]
    BeginInvoke    (
        SignatureExamples.deloutsingle this,
        single scalar argIn,
        System.AsyncCallback callback,
        System.Object object)
```

The `EndInvoke` signature is:

```
[single scalar RetVal,
single scalar argOut]
```

```
    EndInvoke    (
        SignatureExamples.deloutsingle this,
        System.IAsyncResult result)
```

4 Call `times2fnc`.

```
myDel = SignatureExamples.deloutsingle(@times2fnc);
asyncRes = myDel.BeginInvoke(6,[],[]);
while asyncRes.IsCompleted ~= true
    pause(0.05) % Use pause() to let MATLAB process event
end
[a1,a2] = myDel.EndInvoke(asyncRes);
a1
```

```
a1 =
    12
```

Tips

- If the delegate contains `out` or `ref` parameters, the signature for the `EndInvoke` method follows the MATLAB mapping rules. For information, see ".NET Delegates With out and ref Type Arguments".

See Also

`BeginInvoke`

Topics

"Calling .NET Methods Asynchronously"

External Websites

MSDN Calling Synchronous Methods Asynchronously

Introduced in R2011a

endsWith

Determine if strings end with pattern

Syntax

```
TF = endsWith(str,pattern)
TF = endsWith(str,pattern,'IgnoreCase',true)
```

Description

`TF = endsWith(str,pattern)` returns `1` (`true`) if `str` ends with the specified pattern, and returns `0` (`false`) otherwise.

If `pattern` is an array containing multiple patterns, then `endsWith` returns `1` if it finds that `str` ends with any element of `pattern`.

`TF = endsWith(str,pattern,'IgnoreCase',true)` ignores case when determining if `str` ends with `pattern`.

Examples

Determine If String Ends with Pattern

Create a string array that contains file names. Determine which file names end with the `.gz` extension.

Starting in R2017a, you can create strings using double quotes.

```
str = ["abstract.docx","data.tar.gz","mycode.m"; ...
       "data-analysis.ppt","results.ptx","temp-archive.gz"]
```

```
str = 2x3 string array
    "abstract.docx"        "data.tar.gz"    "mycode.m"
    "data-analysis.ppt"    "results.ptx"    "temp-archive.gz"
```

Return a logical array where the position of each element equal to `1` corresponds to the position of a string in `str` that ends with `.gz`.

```
pattern = ".gz";
TF = endsWith(str,pattern)
```

```
TF = 2x3 logical array

   0   1   0
   0   0   1
```

Display the file names that end with `.gz`. Index back into `str` using `TF`.

```
str(TF)
```

```
ans = 2x1 string array
    "data.tar.gz"
    "temp-archive.gz"
```

Test End of String Against Multiple Patterns

Create a string array that contains file names. Determine which file names end with the `.docx`, `.xlsx`, or `.gz` extensions.

Starting in R2017a, you can create strings using double quotes.

```
str = ["data.tar.gz","mycode.m","outputs.xlsx","results.pptx"]
```

```
str = 1x4 string array
    "data.tar.gz"    "mycode.m"    "outputs.xlsx"    "results.pptx"
```

```
pattern = [".docx",".xlsx",".gz"];
TF = endsWith(str,pattern)
```

```
TF = 1x4 logical array

   1   0   1   0
```

Display the file names that end with `.docx`, `.xlsx`, or `.gz`. Index back into `str` using `TF`.

```
str(TF)
```

```
ans = 1x2 string array
    "data.tar.gz"    "outputs.xlsx"
```

Ignore Case When Testing End of String

Create a string array that contains file names. Determine which file names end with the `.gz` extension, ignoring case.

Starting in R2017a, you can create strings using double quotes.

```
str = ["DATA.TAR.GZ","mycode.m","SUMMARY.PPT","tmp.gz"]
```

```
str = 1x4 string array
    "DATA.TAR.GZ"    "mycode.m"    "SUMMARY.PPT"    "tmp.gz"
```

```
pattern = ".gz";
TF = endsWith(str,pattern,'IgnoreCase',true)
```

```
TF = 1x4 logical array

   1   0   0   1
```

Display the file names that end with `.gz`. Index back into `str` using `TF`.

```
str(TF)
```

```
ans = 1x2 string array
    "DATA.TAR.GZ"    "tmp.gz"
```

Determine if Character Vector Ends With Pattern

Create a character vector that contains the name of a file. Determine whether the name ends with specified extensions.

```
chr = 'MyLatestPaper.docx'
```

```
chr =
'MyLatestPaper.docx'
```

```
TF = endsWith(chr,'docx')
```

```
TF = logical
   1
```

```
TF = endsWith(chr,'gz')
```

```
TF = logical
   0
```

Input Arguments

`str` — Input text
string array | character vector | cell array of character vectors

Input text, specified as a string array, a character vector, or a cell array of character vectors.

`pattern` — Search pattern
string array | character vector | cell array of character vectors

Search pattern, specified as a string array, a character vector, or a cell array of character vectors.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `str` and `pattern` must be a string scalar, a character vector, or a cell array containing not more than one character vector.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`contains` | `find` | `regexp` | `startsWith` | `strcmp` | `strfind`

Topics

“Create String Arrays”
“Search and Replace Text”
“Compare Text”
“Test for Empty Strings and Missing Values”

Introduced in R2016b

eomday

Last day of month

Syntax

```
E = eomday(Y,M)
```

Description

`E = eomday(Y,M)` returns the last day of the year and month given by corresponding elements of the numeric arrays `Y` and `M`.

Examples

Show the end of month for January through September for the year 1900:

```
eomday(1900, 1:9)
ans =
    31    28    31    30    31    30    31    31    30
```

Find the number of days during that period:

```
sum(eomday(1900, 1:9))
ans =
   273
```

Because 1996 is a leap year, the statement `eomday(1996,2)` returns `29`. To show all the leap years in the twentieth century, try:

```
y = 1900:1999;
E = eomday(y,2);
y(E == 29)

ans =
  Columns 1 through 6
     1904     1908     1912     1916     1920     1924
```

```
  Columns 7 through 12
     1928     1932     1936     1940     1944     1948

  Columns 13 through 18
     1952     1956     1960     1964     1968     1972

  Columns 19 through 24
     1976     1980     1984     1988     1992     1996
```

See Also

datenum | datevec | weekday

Introduced before R2006a

enumeration

Class enumeration members and names

Syntax

```
enumeration ClassName
enumeration(obj)
m = enumeration( ___ )
[m,s] = enumeration( ___ )
```

Description

`enumeration ClassName` displays the names of the enumeration members for the MATLAB class with the name `ClassName`.

`enumeration(obj)` displays the names of the enumeration members for the class of `obj`.

`m = enumeration( ___ )` returns the enumeration members in the column vector `m`.

`[m,s] = enumeration( ___ )` returns the enumeration members in the column vector `m` and the member names in the cell array `s`. The names in `s` correspond element-wise to the enumeration members in `m`.

Examples

Display Enumeration Member Names

Display the enumeration member names for the matlab.lang.OnOffSwitchState class.

```
enumeration matlab.lang.OnOffSwitchState
```

```
Enumeration members for class 'matlab.lang.OnOffSwitchState':
```

```
    off
    on
```

Display Enumeration Member Names from Object

Display the enumeration member names for the matlab.lang.OnOffSwitchState class using an enumeration member object.

```
e = matlab.lang.OnOffSwitchState.off;
enumeration(e)
```

```
Enumeration members for class 'matlab.lang.OnOffSwitchState':

    off
    on
```

Get Enumeration Members

Return the enumeration members for the matlab.lang.OnOffSwitchState class in a column vector.

```
m = enumeration('matlab.lang.OnOffSwitchState')
```

```
m 
  2x1 OnOffSwitchState enumeration array

    off
    on
```

Get Enumeration Members and Names

Return the enumeration members of the matlab.lang.OnOffSwitchState class in a column vector and the enumeration names in a cell array.

```
[m,s] = enumeration('matlab.lang.OnOffSwitchState')
```

```
m =
  2x1 OnOffSwitchState enumeration array

    off
    on


s = 2x1 cell array
    {'off'}
    {'on' }
```

Input Arguments

`ClassName` — Enumeration class name
character vector | string

Enumeration class name, specified as a character vector or string scalar.

Data Types: `char` | `string`

`obj` — Instance of enumeration class
object

Instance of enumeration class.

Output Arguments

`m` — Enumeration members
column vector

Enumeration members, returned as a column vector.

`s` — Enumeration member names
cell array

Enumeration member names, returned as a cell array of character vectors.

More About

Behavior Description

Behavior of the `enumeration` function.

- An enumeration class that derives from a built-in class can specify more than one name for a given enumeration member.
- When you call the `enumeration` function with no output arguments, MATLAB displays only the first name for each enumeration member (as specified in the class definition). To see all available enumeration members and their names, use the two output arguments. For example, `[m,s] = enumeration(obj);`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`classdef`

Topics

"Define Enumeration Classes"

Introduced in R2009b

eps

Floating-point relative accuracy

Syntax

```
d = eps
d = eps(x)
d = eps(datatype)
```

Description

`d = eps` returns the distance from `1.0` to the next larger double-precision number, that is, 2^{-52}.

`d = eps(x)`, where `x` has data type `single` or `double`, returns the positive distance from `abs(x)` to the next larger floating-point number of the same precision as `x`. If `x` has type `duration`, then `eps(x)` returns the next larger `duration` value. The command `eps(1.0)` is equivalent to `eps`.

`d = eps(datatype)` returns `eps` according to the data type specified by `datatype`, which can be either `'double'` or `'single'`. The syntax `eps('double')` (default) is equivalent to `eps`, and `eps('single')` is equivalent to `eps(single(1.0))`.

Examples

Accuracy in Double Precision

Display the distance from `1.0` to the next largest double-precision number.

```
d = eps
```

```
d = 2.2204e-16
```

`eps` is equivalent to `eps(1.0)` and `eps('double')`.

Compute `log2(eps)`.

```
d = log2(eps)
```

```
d = -52
```

In base 2, `eps` is equal to `2^-52`.

Find the distance from `10.0` to the next largest double-precision number.

```
d = eps(10.0)
```

```
d = 1.7764e-15
```

Accuracy in Single Precision

Display the distance from `1.0` to the next largest single-precision number.

```
d = eps('single')
```

```
d = single
    1.1921e-07
```

`eps('single')` is equivalent to `eps(single(1.0))`.

Compute `log2(eps('single'))`.

```
d = log2(eps('single'))
```

```
d = single
    -23
```

In base 2, single-precision `eps` is equal to `2^-23`.

Find the distance from the single-precision representation of 10.0 to the next largest single-precision number.

```
d = eps(single(10.0))
```

```
d = single
    9.5367e-07
```

Input Arguments

x — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. `d` is the same size as `x`. For all `x`, `eps(x) = eps(-x) = eps(abs(x))`. If `x` is complex, `d` is the distance to the next larger floating-point number in magnitude. If `x` is `Inf` or `NaN`, then `eps(x)` returns `NaN`.

Data Types: `single` | `double` | `duration`
Complex Number Support: Yes

datatype — Output data type
`'double'` (default) | `'single'`

Output data type, specified as `'double'` or `'single'`.

- `eps('double')` is equivalent to `eps` and `eps(1.0)`.
- `eps('single')` is equivalent to `eps(single(1.0))` and `single(2^-23)`.

Data Types: `char`

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`double` | `duration` | `intmax` | `realmax` | `realmin` | `single`

Topics

Floating-Point Numbers

Introduced before R2006a

eq, ==

Determine equality

Syntax

```
A == B
eq(A,B)
```

Description

`A == B` returns a logical array with elements set to logical `1` (`true`) where arrays `A` and `B` are equal; otherwise, the element is logical `0` (`false`). The test compares both real and imaginary parts of numeric arrays. `eq` returns logical `0` (`false`) where `A` or `B` have NaN or undefined categorical elements.

`eq(A,B)` is an alternative way to execute `A == B`, but is rarely used. It enables operator overloading for classes.

Examples

Equality of Two Vectors

Create two vectors containing both real and imaginary numbers, then compare the vectors for equality.

```
A = [1+i 3 2 4+i];
B = [1 3+i 2 4+i];
A == B
```

```
ans = 1x4 logical array

   0   0   1   1
```

The `eq` function tests both real and imaginary parts for equality, and returns logical `1` (`true`) only where both parts are equal.

Find Characters

Create a character vector.

```
M = 'masterpiece';
```

Test for the presence of a specific character using `==`.

```
M == 'e'
```

```
ans = 1x11 logical array

   0   0   0   0   1   0   0   0   1   0   1
```

The value of logical `1` (`true`) indicates the presence of the character `'e'`.

Find Values in Categorical Array

Create a categorical array with two values: `'heads'` and `'tails'`.

```
A = categorical({'heads' 'heads' 'tails'; 'tails' 'heads' 'tails'})
```

```
A = 2x3 categorical array
     heads      heads      tails
     tails      heads      tails
```

Find all values in the `'heads'` category.

```
A == 'heads'
```

```
ans = 2x3 logical array

   1   1   0
   0   1   0
```

A value of logical `1` (`true`) indicates a value in the category.

Compare the rows of `A` for equality.

```
A(1,:) == A(2,:)
```

```
ans = 1x3 logical array

   0   1   1
```

A value of logical `1` (`true`) indicates where the rows have equal category values.

Compare Floating-Point Numbers

Many numbers expressed in decimal text cannot be represented exactly as binary floating numbers. This leads to small differences in results that the == operator reflects.

Perform a few subtraction operations on numbers expressed in decimal and store the result in `C`.

```
C = 0.5-0.4-0.1
```

```
C = -2.7756e-17
```

With exact decimal arithmetic, `C` should be equal to *exactly* `0`. Its small value is due to the nature of binary floating-point arithmetic.

Compare `C` to `0` for equality.

```
C == 0
```

```
ans = logical
   0
```

Compare floating-point numbers using a tolerance, `tol`, instead of using ==.

```
tol = eps(0.5);
abs(C-0) < tol
```

```
ans = logical
   1
```

The two numbers, `C` and `0`, are closer to one another than two consecutive floating-point numbers near `0.5`. In many situations, `C` may act like `0`.

Compare Datetime Values

Compare the elements of two `datetime` arrays.

Create two `datetime` arrays in different time zones.

```
t1 = [2014,04,14,9,0,0;2014,04,14,10,0,0];
A = datetime(t1,'TimeZone','America/Los_Angeles');
A.Format = 'd-MMM-y HH:mm:ss Z'
```

```
A = 2x1 datetime array
   14-Apr-2014 09:00:00 -0700
   14-Apr-2014 10:00:00 -0700
```

```
t2 = [2014,04,14,12,0,0;2014,04,14,12,30,0];
B = datetime(t2,'TimeZone','America/New_York');
B.Format = 'd-MMM-y HH:mm:ss Z'
```

```
B = 2x1 datetime array
   14-Apr-2014 12:00:00 -0400
   14-Apr-2014 12:30:00 -0400
```

Check where elements in `A` and `B` are equal.

```
A==B
```

```
ans = 2x1 logical array

   1
   0
```

Input Arguments

A, B — Operands
scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Numeric or string inputs A and B must either be the same size or have sizes that are compatible (for example, A is an M-by-N matrix and B is a scalar or 1-by-N row vector). For more information, see “Compatible Array Sizes for Basic Operations”.

You can compare numeric inputs of any type, and the comparison does not suffer loss of precision due to type conversion.

If A and B are categorical, datetime, or duration arrays, then they must be the same size unless one is a scalar.

- If one input is a categorical array, the other input can be a categorical array, a cell array of character vectors, or a single character vector. A single character vector expands into a cell array of character vectors of the same size as the other input. If both inputs are ordinal categorical arrays, they must have the same sets of categories, including their order. If both inputs are categorical arrays that are not ordinal, they can have different sets of categories. See “Compare Categorical Array Elements” for more details.
- If one input is a datetime array, the other input can be a datetime array, a character vector, or a cell array of character vectors.
- If one input is a duration array, the other input can be a duration array or a numeric array. The operator treats each numeric value as a number of standard 24-hour days.
- If one input is a string array, the other input can be a string array, a character vector, or a cell array of character vectors. The corresponding elements of A and B are compared lexicographically.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `categorical` | `datetime` | `duration`
Complex Number Support: Yes

Tips

- When comparing handle objects, use == to test whether objects have the same handle. Use `isequal` to determine if objects with different handles have equal property values.

Compatibility Considerations

Implicit expansion change affects arguments for operators

Behavior changed in R2016b

Starting in R2016b with the addition of implicit expansion, some combinations of arguments for basic operations that previously returned errors now produce results. For example, you previously could not add a row and a column vector, but those operands are now valid for addition. In other words, an expression like `[1 2] + [1; 2]` previously returned a size mismatch error, but now it executes.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a `try/catch` block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see “Compatible Array Sizes for Basic Operations”.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support using `eq` to test equality between an enumeration member and a string array, a character array, or a cell array of character arrays.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`ge` | `gt` | `le` | `lt` | `ne`

Topics

“Array Comparison with Relational Operators”
“MATLAB Operators and Special Characters”

Introduced before R2006a

equilibrate

Matrix scaling for improved conditioning

Syntax

```
[P,R,C] = equilibrate(A)
```

Description

`[P,R,C] = equilibrate(A)` permutes and rescales matrix `A` such that the new matrix `B = R*P*A*C` has a diagonal with entries of magnitude 1, and its off-diagonal entries are not greater than 1 in magnitude.

Examples

Equilibrate Matrix for Improved Conditioning

Equilibrate a matrix with a large condition number to improve the efficiency and stability of a linear system solution with the iterative solver `gmres`.

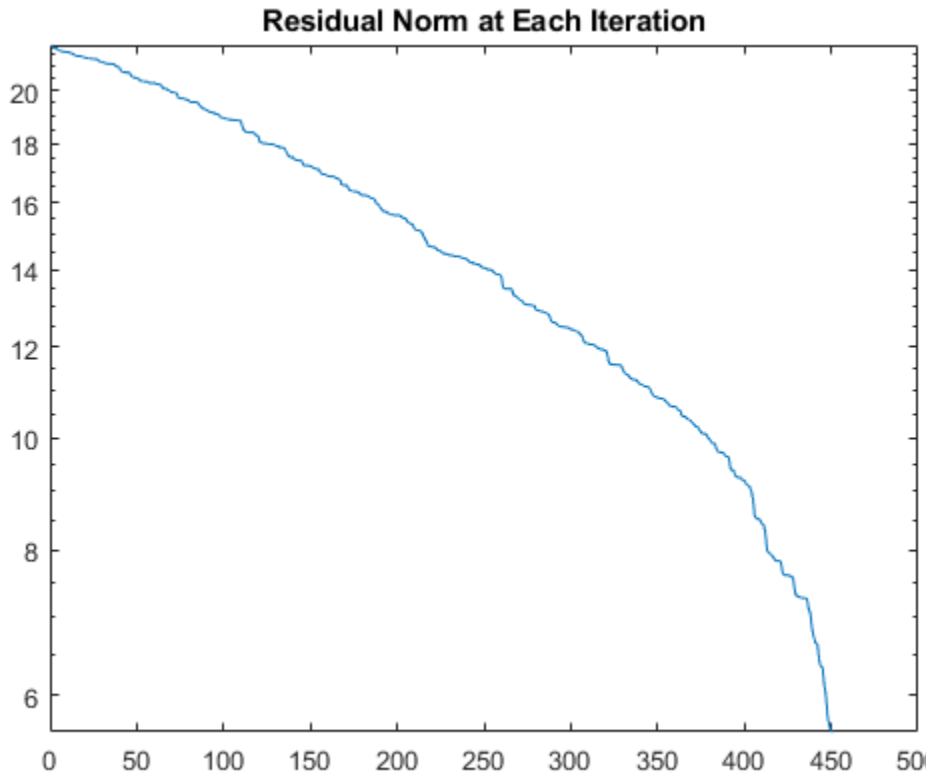
Load the `west0479` matrix, which is a real-valued 479-by-479 sparse matrix. Use `condest` to calculate the estimated condition number of the matrix.

```
load west0479
A = west0479;
c1 = condest(A)
```

```
c1 = 1.4244e+12
```

Try to solve the linear system Ax = *b* using `gmres` with 450 iterations and a tolerance of `1e-11`. Specify five outputs so that `gmres` returns the residual norms of the solution at each iteration (using ~ to suppress unneeded outputs). Plot the residual norms in a semilog plot. The plot shows that `gmres` is not able to achieve a reasonable residual norm, and so the calculated solution for *x* is not reliable.

```
b = ones(size(A,1),1);
tol = 1e-11;
maxit = 450;
[x,flx,~,~,rvx] = gmres(A,b,[],tol,maxit);
semilogy(rvx)
title('Residual Norm at Each Iteration')
```



Use `equilibrate` to permute and rescale `A`. Create a new matrix `B = R*P*A*C`, which has a better condition number and diagonal entries of only 1 and -1.

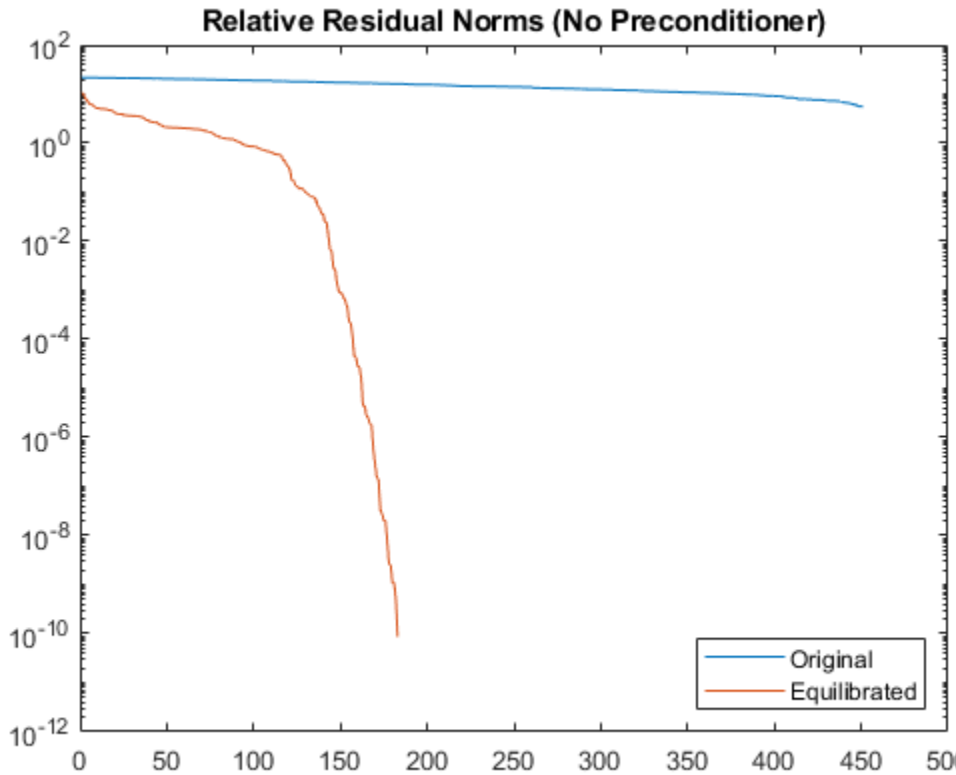
```
[P,R,C] = equilibrate(A);
B = R*P*A*C;
c2 = condest(B)
```

```
c2 = 5.1036e+04
```

Using the outputs returned by `equilibrate`, you can reformulate the problem Ax = *b* into By = *d*, where *B* = RPAC and *d* = RPb. In this form you can recover the solution to the original system with *x* = Cy.

Use `gmres` to solve By = *d* for *y*, and then replot the residual norms at each iteration. The plot shows that equilibrating the matrix improves the stability of the problem, with `gmres` converging to the desired tolerance of `1e-11` in fewer than 200 iterations.

```
d = R*P*b;
[y,fly,~,~,rvy] = gmres(B,d,[],tol,maxit);
hold on
semilogy(rvy)
legend('Original', 'Equilibrated', 'Location', 'southeast')
title('Relative Residual Norms (No Preconditioner)')
hold off
```

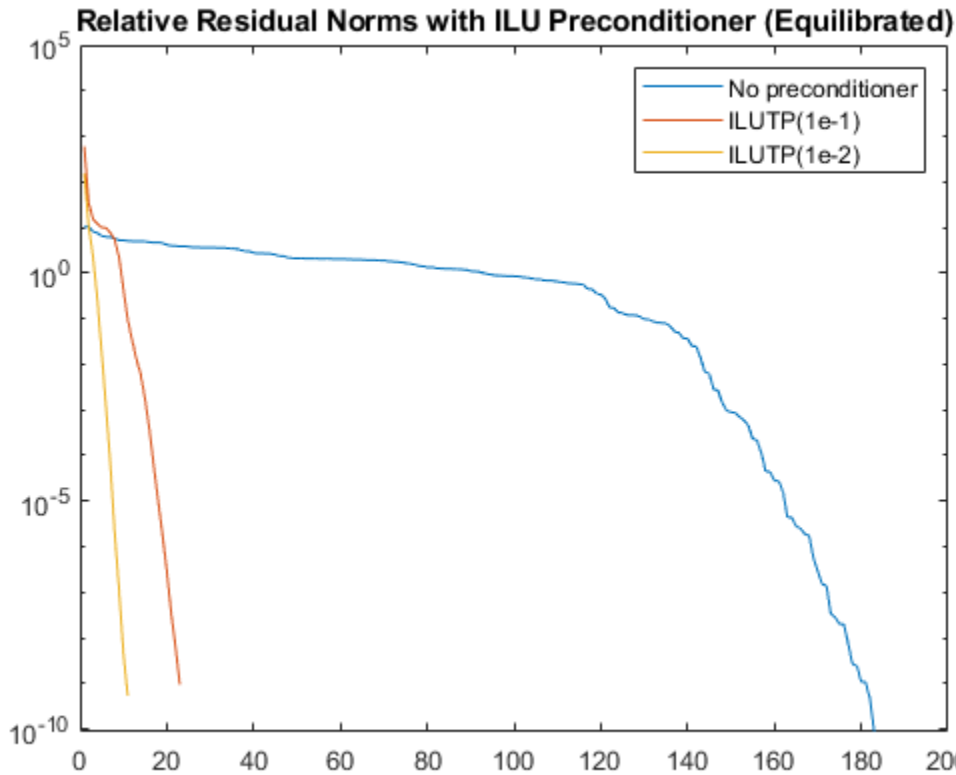


Improve Solution with Preconditioner

After you obtain the matrix `B`, you can improve the stability of the problem even further by calculating a preconditioner for use with `gmres`. The numerical properties of `B` are better than those of the original matrix `A`, so you should use the equilibrated matrix to calculate the preconditioner.

Calculate two different preconditioners with `ilu`, and use these as inputs to `gmres` to solve the problem again. Plot the residual norms at each iteration on the same plot as the equilibrated norms for comparison. The plot shows that calculating preconditioners with the equilibrated matrix greatly increases the stability of the problem, with `gmres` achieving the desired tolerance in fewer than 30 iterations.

```
semilogy(rvy)
hold on

[L1,U1] = ilu(B,struct('type','ilutp','droptol',1e-1,'thresh',0));
[yp1,flyp1,~,~,rvyp1] = gmres(B,d,[],tol,maxit,L1,U1);
semilogy(rvyp1)

[L2,U2] = ilu(B,struct('type','ilutp','droptol',1e-2,'thresh',0));
[yp2,flyp2,~,~,rvyp2] = gmres(B,d,[],tol,maxit,L2,U2);
semilogy(rvyp2)

legend('No preconditioner', 'ILUTP(1e-1)', 'ILUTP(1e-2)')
title('Relative Residual Norms with ILU Preconditioner (Equilibrated)')
hold off
```



Input Arguments

A — Input matrix
square matrix

Input matrix, specified as a square matrix. `A` can be dense or sparse, but must be structurally nonsingular, as determined by `sprank`.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

P — Permutation matrix
sparse matrix

Permutation matrix, returned as a sparse matrix. `P*A` is the permutation of `A` that maximizes the absolute value of the product of its diagonal elements.

R — Row scaling
sparse diagonal matrix

Row scaling, returned as a sparse diagonal matrix. The diagonal entries in `R` and `C` are real and positive.

C — Column scaling
sparse diagonal matrix

Column scaling, returned as a sparse diagonal matrix. The diagonal entries in `R` and `C` are real and positive.

See Also

`balance` | `gmres` | `ichol` | `ilu` | `matchpairs` | `sprank`

Introduced in R2019a

erase

Delete substrings within strings

Syntax

```
newStr = erase(str,match)
```

Description

`newStr = erase(str,match)` deletes all occurrences of `match` in `str`. The `erase` function returns the remaining text as `newStr`.

If `match` is a string array or a cell array of character vectors, then `erase` deletes every occurrence of every element of `match` in `str`. The `str` and `match` arguments do not need to be the same size.

Examples

Delete Substrings from String Array

Create a string array and delete substrings from it. Starting in R2017a, you can create strings using double quotes.

```
str = ["the quick brown fox jumps";
       "over the lazy dog"]
```

```
str = 2x1 string array
    "the quick brown fox jumps"
    "over the lazy dog"
```

Delete the substring `"the "` from `str`. The `erase` function deletes both instances.

```
newStr = erase(str,"the ")
```

```
newStr = 2x1 string array
    "quick brown fox jumps"
    "over lazy dog"
```

Delete multiple substrings from `str`.

```
match = ["the ","quick ","lazy "];
newStr = erase(str,match)
```

```
newStr = 2x1 string array
    "brown fox jumps"
    "over dog"
```

Delete Substring from Character Vector

Create a character vector. Delete the substring, `' World'`, including the space character.

```
chr = 'Hello World'
```

```
chr =
'Hello World'
```

```
newChr = erase(chr,' World')
```

```
newChr =
'Hello'
```

Input Arguments

`str` — Input text
string array | character vector | cell array of character vectors

Input text, specified as a string array, a character vector, or a cell array of character vectors.

`match` — Text to delete
string array | character vector | cell array of character vectors

Text to delete, specified as a string array, a character vector, or a cell array of character vectors.

Tips

- To delete multiple occurrences of a match when the occurrences overlap, use the `strrep` function. `erase` only deletes the first occurrence when occurrences overlap.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `str` and `match` must be a string scalar, a character vector, or a cell array containing not more than one character vector.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`count` | `eraseBetween` | `extractAfter` | `extractBefore` | `extractBetween` | `insertAfter` | `insertBefore` | `join` | `pad` | `replace` | `replaceBetween` | `size` | `split` | `strlength` | `strsplit`

Topics

"Create String Arrays"
"Search and Replace Text"
"Test for Empty Strings and Missing Values"

Introduced in R2016b

eraseBetween

Delete substrings between start and end points

Syntax

```
newStr = eraseBetween(str,startStr,endStr)
newStr = eraseBetween(str,startPos,endPos)
newStr = eraseBetween( ___ ,'Boundaries',bounds)
```

Description

`newStr = eraseBetween(str,startStr,endStr)` deletes all characters from `str` that occur between the substrings `startStr` and `endStr`, but does not delete `startStr` and `endStr` themselves. `eraseBetween` returns the remaining text as `newStr`.

If `str` is a string array or a cell array of character vectors, then `eraseBetween` deletes characters from each element of `str`. The output argument `newStr` has the same data type as `str`.

`newStr = eraseBetween(str,startPos,endPos)` deletes all characters from `str` that occur between the positions `startPos` and `endPos`, including the characters at those positions.

`newStr = eraseBetween( ___ ,'Boundaries',bounds)` forces the starts and ends specified in any of the previous syntaxes to be either inclusive or exclusive. They are inclusive when `bounds` is `'inclusive'`, and exclusive when `bounds` is `'exclusive'`. For example, `eraseBetween(str,startStr,endStr,'Boundaries','inclusive')` deletes `startStr`, `endStr`, and all the text between them.

Examples

Delete Text Between Substrings

Create string arrays. Then delete text that occurs between substrings.

Create a string. Starting in R2017a, you can create strings using double quotes.

```
str = "The quick brown fox"
```

```
str = 
"The quick brown fox"
```

Delete the text that occurs between the substrings `"quick"` and `" fox"`. The `eraseBetween` function deletes the text but does not delete `"quick"` or `" fox"`.

```
newStr = eraseBetween(str,"quick"," fox")
```

```
newStr = 
"The quick fox"
```

Delete substrings from each element of a string array. When you specify different substrings as start and end indicators, they must be contained in a string array or a cell array of character vectors that is the same size as `str`.

```
str = ["The quick brown fox jumps";"over the lazy dog"]
```

```
str = 2x1 string array
    "The quick brown fox jumps"
    "over the lazy dog"
```

```
startPos = ["quick";"the"];
endPos = [" fox";" dog"];
newStr = eraseBetween(str,startPos,endPos)
```

```
newStr = 2x1 string array
    "The quick fox jumps"
    "over the dog"
```

Delete Substrings Between Start and End Positions

Create string arrays and delete substrings between start and end positions that are specified as numbers.

Create a string that contains a name. Starting in R2017a, you can create strings using double quotes.

```
str = "Edgar Allen Poe"
```

```
str =
"Edgar Allen Poe"
```

Delete a substring. To delete the middle name and one of the space characters, specify the sixth and 11th positions in the string. The deleted substring includes the sixth and 11th characters.

```
newStr = eraseBetween(str,6,11)
```

```
newStr =
"Edgar Poe"
```

Delete substrings from each element of a string array. When you specify different start and end positions with numeric arrays, they must be the same size as the input string array.

```
str = ["Edgar Allen Poe";"Louisa May Alcott"]
```

```
str = 2x1 string array
    "Edgar Allen Poe"
    "Louisa May Alcott"
```

```
startsPos = [6;7];
endPos = [11;10];
newStr = eraseBetween(str,startsPos,endPos)
```

```
newStr = 2x1 string array
    "Edgar Poe"
    "Louisa Alcott"
```

Delete Text with Inclusive and Exclusive Boundaries

Delete text from string arrays with boundaries that are forced to be inclusive or exclusive. `eraseBetween` deletes the boundaries when they are inclusive. `eraseBetween` returns the boundaries as part of the output string array when they are exclusive.

Create a string array. Starting in R2017a, you can create strings using double quotes.

```
str = "small|medium|large"
```

```
str =
"small|medium|large"
```

Delete the text between sixth and 13th positions, but do not delete the characters at those positions.

```
newStr = eraseBetween(str,6,13,'Boundaries','exclusive')
```

```
newStr =
"small||large"
```

Delete the text between two substrings, and also the substrings themselves.

```
str = "The quick brown fox jumps over the lazy dog"
```

```
str =
"The quick brown fox jumps over the lazy dog"
```

```
newStr = eraseBetween(str," brown","lazy",'Boundaries','inclusive')
```

```
newStr =
"The quick dog"
```

Delete Text Between Positions in Character Vector

Create a character vector and delete text between start and end positions.

```
chr = 'mushrooms, peppers, and onions'
```

```
chr =
'mushrooms, peppers, and onions'
```

```
newChr = eraseBetween(chr,10,19)
```

```
newChr =
'mushrooms and onions'
```

Delete text between substrings.

```
newChr = eraseBetween(chr,'mushrooms',' and')
```

```
newChr =
'mushrooms and onions'
```

Input Arguments

`str` — Input text
string array | character vector | cell array of character vectors

Input text, specified as a string array, a character vector, or a cell array of character vectors.

Data Types: `string` | `char` | `cell`

`startStr` — String that indicates start of substring to delete
string array | character vector | cell array of character vectors

String that indicates the start of the substring to delete, specified as a string array, a character vector, or a cell array of character vectors.

If `str` is a string array or a cell array of character vectors, then `startStr` can be a character vector, a string scalar, or a string array or a cell array of the same size as `str`.

Example: `eraseBetween(str,"AB","YZ")` deletes all characters between `AB` and `YZ` in each element of `str`.

Example: If `str` is a 2-by-1 string array, then `eraseBetween(str,["AB";"FG"],["YZ";"ST"])` deletes all characters between `AB` and `YZ` in `str(1)`, and between `FG` and `ST` in `str(2)`.

Data Types: `string` | `char` | `cell`

`endStr` — String that indicates end of substring to delete
string array | character vector | cell array of character vectors

String that indicates the end of the substring to delete, specified as a string array, a character vector, or a cell array of character vectors.

If `str` is a string array or a cell array of character vectors, then `endStr` can be a character vector, a string scalar, or a string array or a cell array of the same size as `str`.

Example: `eraseBetween(str,"AB","YZ")` deletes all characters between `AB` and `YZ` in each element of `str`.

Example: If `str` is a 2-by-1 string array, then `eraseBetween(str,["AB";"FG"],["YZ";"ST"])` deletes all characters between `AB` and `YZ` in `str(1)`, and between `FG` and `ST` in `str(2)`.

Data Types: `string` | `char` | `cell`

`startPos` — Start position of substring to delete
numeric array

Start position of substring to delete, specified as a numeric array.

If `str` is a string array or a cell array of character vectors, then `startPos` can be a numeric scalar or a numeric array of the same size as `str`.

Example: `eraseBetween(str,5,9)` deletes all characters from the fifth through the ninth positions in each element of `str`.

Example: If `str` is a 2-by-1 string array, then `eraseBetween(str,[5;10],[9;21])` deletes all characters from the fifth through the ninth positions in `str(1)`, and from the 10th through the 21st positions in `str(2)`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`endPos` — End position of substring to delete
numeric array

End position of substring to delete, specified as a numeric array.

If `str` is a string array or a cell array of character vectors, then `endPos` can be a numeric scalar or a numeric array of the same size as `str`.

Example: `eraseBetween(str,5,9)` deletes all characters from the fifth through the ninth positions in each element of `str`.

Example: If `str` is a 2-by-1 string array, then `eraseBetween(str,[5;10],[9;21])` deletes all characters from the fifth through the ninth positions in `str(1)`, and from the 10th through the 21st positions in `str(2)`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`newStr` — Output text
string array | character vector | cell array of character vectors

Output text, returned as a string array, a character vector, or a cell array of character vectors. `str` and `newStr` have the same data type.

Data Types: `string` | `char` | `cell`

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `str`, `startStr`, and `endStr` must be a string scalar, a character vector, or a cell array containing not more than one character vector.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`count` | `erase` | `extractAfter` | `extractBefore` | `extractBetween` | `insertAfter` | `insertBefore` | `join` | `replace` | `replaceBetween` | `size` | `split` | `strlength`

Topics

“Create String Arrays”
“Search and Replace Text”
“Test for Empty Strings and Missing Values”

Introduced in R2016b

erf

Error function

Syntax

```
erf(x)
```

Description

`erf(x)` returns the "Error Function" on page 1-3841 evaluated for each element of `x`.

Examples

Find Error Function

Find the error function of a value.

```
erf(0.76)
```

```
ans = 0.7175
```

Find the error function of the elements of a vector.

```
V = [-0.5 0 1 0.72];
erf(V)
```

```
ans = 1×4

   -0.5205         0    0.8427    0.6914
```

Find the error function of the elements of a matrix.

```
M = [0.29 -0.11; 3.1 -2.9];
erf(M)
```

```
ans = 2×2

    0.3183   -0.1236
    1.0000   -1.0000
```

Find Cumulative Distribution Function of Normal Distribution

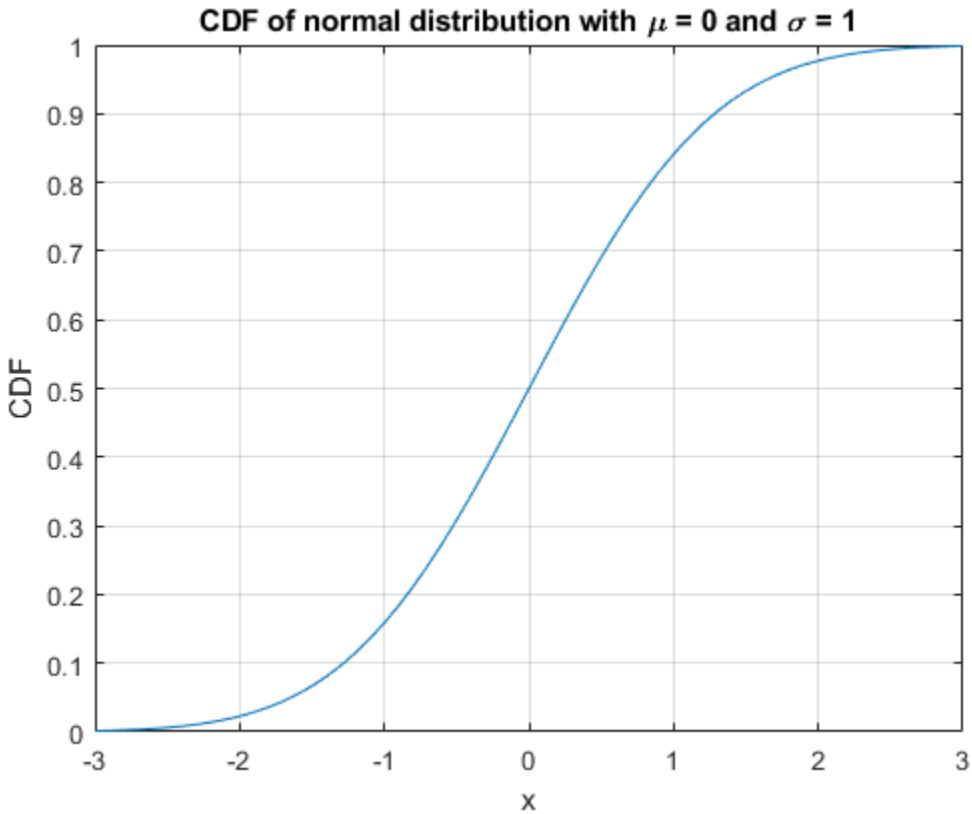
The cumulative distribution function (CDF) of the normal, or Gaussian, distribution with standard deviation σ and mean μ is

$$\phi(x) = \frac{1}{2}\left(1 + \mathrm{erf}\left(\frac{x - \mu}{\sigma\sqrt{2}}\right)\right).$$

Note that for increased computational accuracy, you can rewrite the formula in terms of `erfc` . For details, see “Tips” on page 1-3841.

Plot the CDF of the normal distribution with $\mu = 0$ and $\sigma = 1$.

```
x = -3:0.1:3;
y = (1/2)*(1+erf(x/sqrt(2)));
plot(x,y)
grid on
title('CDF of normal distribution with \mu = 0 and \sigma = 1')
xlabel('x')
ylabel('CDF')
```



Calculate Solution of Heat Equation with Initial Condition

Where $u(x, t)$ represents the temperature at position x and time t, the heat equation is

$$\frac{\partial u}{\partial t} = c\frac{\partial^2 u}{\partial x^2},$$

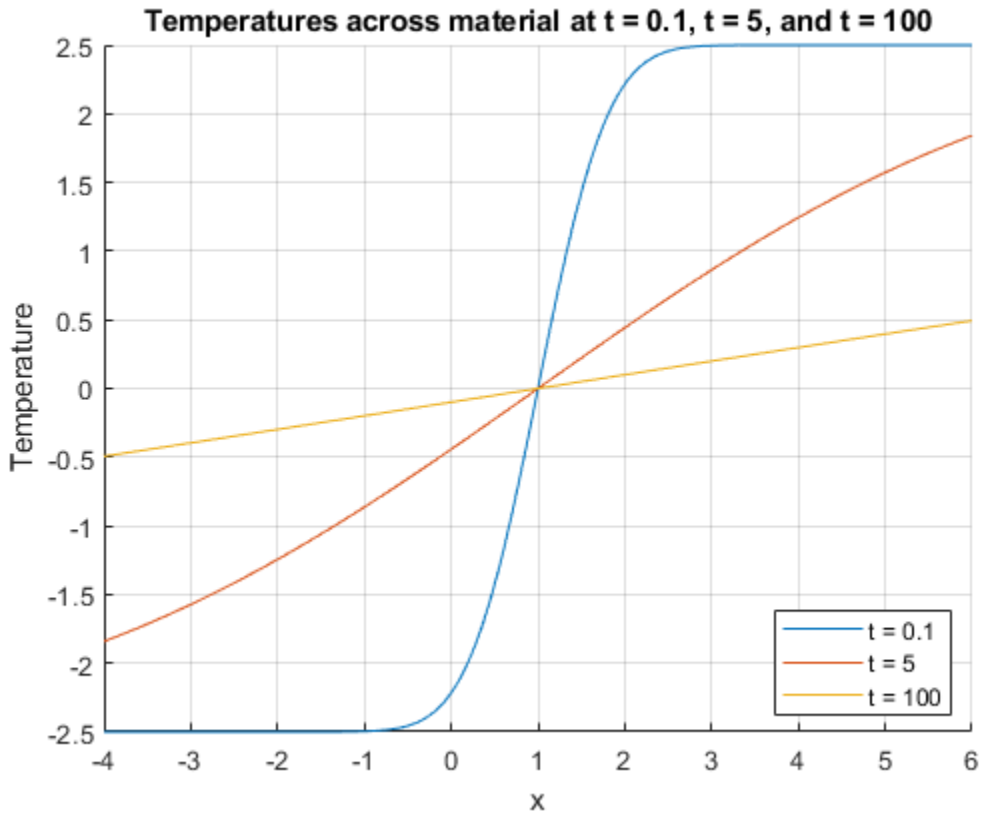
where c is a constant.

For a material with heat coefficient k, and for the initial condition $u(x, 0) = a$ for $x > b$ and $u(x, 0) = 0$ elsewhere, the solution to the heat equation is

$$u(x,t) = \frac{a}{2}\left(\mathrm{erf}\left(\frac{\mathrm{x} - \mathrm{b}}{\sqrt{4\mathrm{kt}}}\right)\right).$$

For `k = 2`, `a = 5`, and `b = 1`, plot the solution of the heat equation at times `t = 0.1`, `5`, and `100`.

```
x = -4:0.01:6;
t = [0.1 5 100];
a = 5;
k = 2;
b = 1;
figure(1)
hold on
for i = 1:3
    u(i,:) = (a/2)*(erf((x-b)/sqrt(4*k*t(i))));
    plot(x,u(i,:))
end
grid on
xlabel('x')
ylabel('Temperature')
legend('t = 0.1','t = 5','t = 100','Location','best')
title('Temperatures across material at t = 0.1, t = 5, and t = 100')
```



Input Arguments

x — Input

real number | vector of real numbers | matrix of real numbers | multidimensional array of real numbers

Input, specified as a real number, or a vector, matrix, or multidimensional array of real numbers. x cannot be sparse.

Data Types: `single` | `double`

More About

Error Function

The error function erf of x is

$$\mathrm{erf}(x)=\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt.$$

Tips

- You can also find the standard normal probability distribution using the Statistics and Machine Learning Toolbox™ function `normcdf`. The relationship between the error function `erf` and `normcdf` is

  $$\mathrm{normcdf}(x) = \frac{1}{2}\left(1 - \mathrm{erf}\left(\frac{-x}{\sqrt{2}}\right)\right).$$

- For expressions of the form `1 - erf(x)`, use the complementary error function `erfc` instead. This substitution maintains accuracy. When `erf(x)` is close to `1`, then `1 - erf(x)` is a small number and might be rounded down to `0`. Instead, replace `1 - erf(x)` with `erfc(x)`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Strict single-precision calculations are not supported. In the generated code, single-precision inputs produce single-precision outputs. However, variables inside the function might be double-precision.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`erfc` | `erfcinv` | `erfcx` | `erfinv`

Introduced before R2006a

erfc

Complementary error function

Syntax

```
erfc(x)
```

Description

`erfc(x)` returns the “Complementary Error Function” on page 1-3846 evaluated for each element of `x`. Use the `erfc` function to replace `1 - erf(x)` for greater accuracy when `erf(x)` is close to `1`.

Examples

Find Complementary Error Function

Find the complementary error function of a value.

```
erfc(0.35)
```

```
ans = 0.6206
```

Find the complementary error function of the elements of a vector.

```
V = [-0.5 0 1 0.72];
erfc(V)
```

```
ans = 1×4

    1.5205    1.0000    0.1573    0.3086
```

Find the complementary error function of the elements of a matrix.

```
M = [0.29 -0.11; 3.1 -2.9];
erfc(M)
```

```
ans = 2×2

    0.6817    1.1236
    0.0000    2.0000
```

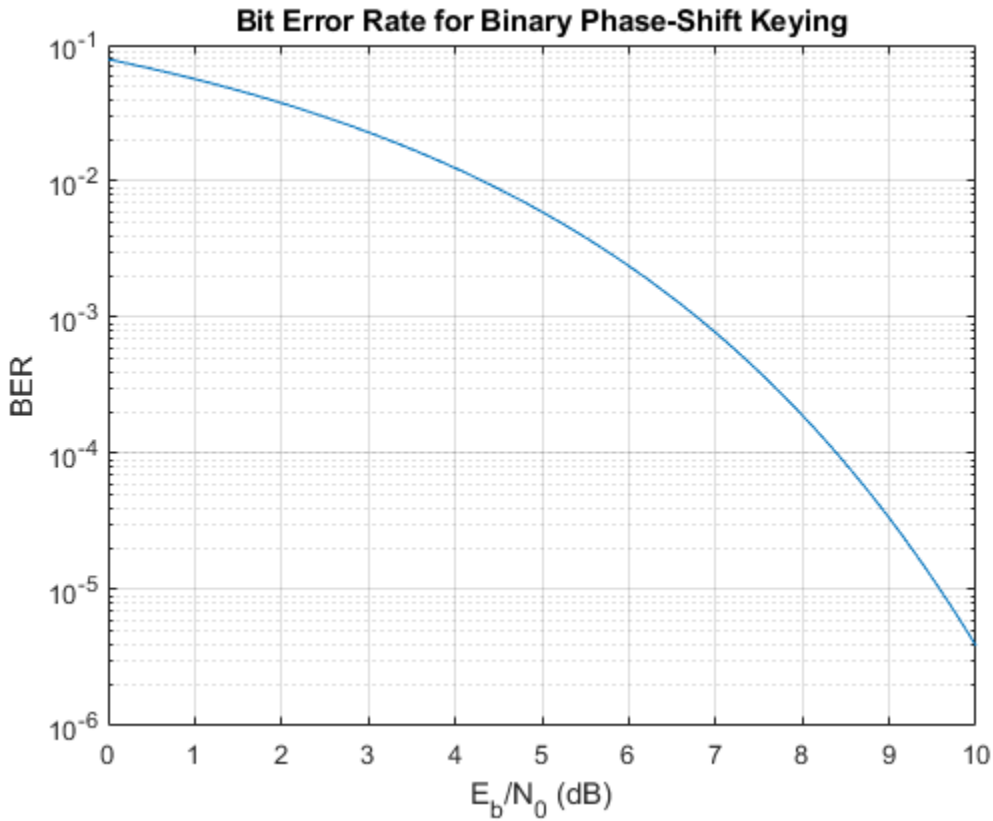
Find Bit Error Rate of Binary Phase-Shift Keying

The bit error rate (BER) of binary phase-shift keying (BPSK), assuming additive white Gaussian noise (AWGN), is

$$P_b = \frac{1}{2}\mathrm{erfc}\left(\sqrt{\frac{\mathrm{E_b}}{\mathrm{N_0}}}\right).$$

Plot the BER for BPSK for values of E_b/N_0 from `0dB` to `10dB`.

```
EbN0_dB = 0:0.1:10;
EbN0 = 10.^(EbN0_dB/10);
BER = 1/2.*erfc(sqrt(EbN0));
semilogy(EbN0_dB,BER)
grid on
ylabel('BER')
xlabel('E_b/N_0 (dB)')
title('Bit Error Rate for Binary Phase-Shift Keying')
```



Avoid Roundoff Errors Using Complementary Error Function

You can use the complementary error function `erfc` in place of `1 - erf(x)` to avoid roundoff errors when `erf(x)` is close to `1`.

Show how to avoid roundoff errors by calculating `1 - erf(10)` using `erfc(10)`. The original calculation returns `0` while `erfc(10)` returns the correct result.

```
1 - erf(10)
```

```
ans = 0
```

```
erfc(10)
```

```
ans = 2.0885e-45
```

Input Arguments

x — Input
real number | vector of real numbers | matrix of real numbers | multidimensional array of real numbers

Input, specified as a real number, or a vector, matrix, or multidimensional array of real numbers. `x` cannot be sparse.

Data Types: `single` | `double`

More About

Complementary Error Function

The complementary error function of x is defined as

$$\begin{aligned}\operatorname{erfc}(x) &= \frac{2}{\sqrt{\pi}}\int_x^{\infty} e^{-t^2}dt \\ &= 1 - \operatorname{erf}(x)\,.\end{aligned}$$

It is related to the error function as

$$\operatorname{erfc}(x) = 1 - \operatorname{erf}(x)\,.$$

Tips

- You can also find the standard normal probability distribution using the Statistics and Machine Learning Toolbox function `normcdf`. The relationship between the error function `erfc` and `normcdf` is

$$\operatorname{normcdf}(x) = \left(\frac{1}{2}\right) \times \operatorname{erfc}\left(\frac{-x}{\sqrt{2}}\right)$$

- For expressions of the form `1 - erfc(x)`, use the error function `erf` instead. This substitution maintains accuracy. When `erfc(x)` is close to `1`, then `1 - erfc(x)` is a small number and might be rounded down to `0`. Instead, replace `1 - erfc(x)` with `erf(x)`.
- For expressions of the form `exp(x^2)*erfc(x)`, use the scaled complementary error function `erfcx` instead. This substitution maintains accuracy by avoiding roundoff errors for large values of `x`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Strict single-precision calculations are not supported. In the generated code, single-precision inputs produce single-precision outputs. However, variables inside the function might be double-precision.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`erf` | `erfcinv` | `erfcx` | `erfinv`

Introduced before R2006a

erfcinv

Inverse complementary error function

Syntax

```
erfcinv(x)
```

Description

`erfcinv(x)` returns the value of the “Inverse Complementary Error Function” on page 1-3851 for each element of `x`. For inputs outside the interval `[0 2]`, `erfcinv` returns `NaN`. Use the `erfcinv` function to replace expressions containing `erfinv(1-x)` for greater accuracy when `x` is close to `1`.

Examples

Find Inverse Complementary Error Function

```
erfcinv(0.3)
```

```
ans = 0.7329
```

Find the inverse complementary error function of the elements of a vector.

```
V = [-10 0 0.5 1.3 2 Inf];
erfcinv(V)
```

```
ans = 1×6

       NaN       Inf    0.4769   -0.2725      -Inf       NaN
```

Find the inverse complementary error function of the elements of a matrix.

```
M = [0.1 1.2; 1 0.9];
erfcinv(M)
```

```
ans = 2×2

    1.1631   -0.1791
         0    0.0889
```

Avoid Roundoff Errors Using Inverse Complementary Error Function

You can use the inverse complementary error function `erfcinv` in place of `erfinv(1-x)` to avoid roundoff errors when `x` is close to `0`.

Show how to avoid roundoff by calculating `erfinv(1-x)` using `erfcinv(x)` for `x = 1e-100`. The original calculation returns `Inf` while `erfcinv(x)` returns the correct result.

```
x = 1e-100;
erfinv(1-x)
```

```
ans = Inf
```

```
erfcinv(x)
```

```
ans = 15.0656
```

Input Arguments

x — Input

real number | vector of real numbers | matrix of real numbers | multidimensional array of real numbers

Input, specified as a real number, or a vector, matrix, or multidimensional array of real numbers. `x` cannot be sparse.

Data Types: `single` | `double`

More About

Inverse Complementary Error Function

The inverse complementary error function `erfcinv(x)` is defined as $\text{erfcinv}(\text{erfc}(x)) = x$.

Tips

- You can also find the inverse standard normal probability distribution using the Statistics and Machine Learning Toolbox function `norminv`. The relationship between the inverse complementary error function `erfcinv` and `norminv` is

  $$\text{norminv}(p) = (-\sqrt{2}) \times \text{erfcinv}(2p).$$

- For expressions of the form `erfcinv(1-x)`, use the inverse error function `erfinv` instead. This substitution maintains accuracy. When `x` is close to `1`, then `1 - x` is a small number and might be rounded down to `0`. Instead, replace `erfcinv(1-x)` with `erfinv(x)`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Strict single-precision calculations are not supported. In the generated code, single-precision inputs produce single-precision outputs. However, variables inside the function might be double-precision.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`erf` | `erfc` | `erfcx` | `erfinv`

Introduced before R2006a

erfcx

Scaled complementary error function

Syntax

```
erfcx(x)
```

Description

`erfcx(x)` returns the value of the “Scaled Complementary Error Function” on page 1-3855 for each element of `x`. Use the `erfcx` function to replace expressions containing `exp(x^2)*erfc(x)` to avoid underflow or overflow errors.

Examples

Find Scaled Complementary Error Function

```
erfcx(5)
```

```
ans = 0.1107
```

Find the scaled complementary error function of the elements of a vector.

```
V = [-Inf -1 0 1 10 Inf];
erfcx(V)
```

```
ans = 1×6

       Inf    5.0090    1.0000    0.4276    0.0561         0
```

Find the scaled complementary error function of the elements of a matrix.

```
M = [-0.5 15; 3.2 1];
erfcx(M)
```

```
ans = 2×2

    1.9524    0.0375
    0.1687    0.4276
```

Avoid Roundoff Errors Using Scaled Complementary Error Function

You can use the scaled complementary error function `erfcx` in place of `exp(x^2)*erfc(x)` to avoid underflow or overflow errors.

Show how to avoid roundoff errors by calculating `exp(35^2)*erfc(35)` using `erfcx(35)`. The original calculation returns `NaN` while `erfcx(35)` returns the correct result.

```
x = 35;
exp(x^2)*erfc(x)
```

```
ans = NaN
```

```
erfcx(x)
```

```
ans = 0.0161
```

Input Arguments

x — Input
real number | vector of real numbers | matrix of real numbers | multidimensional array of real numbers

Input, specified as a real number, or a vector, matrix, or multidimensional array of real numbers. `x` cannot be sparse.

Data Types: `single` | `double`

More About

Scaled Complementary Error Function

The scaled complementary error function `erfcx(x)` is defined as

$$\mathrm{erfcx}(x) = e^{x^2}\mathrm{erfc}(x).$$

For large `X`, `erfcx(X)` is approximately $\left(\frac{1}{\sqrt{\pi}}\right)\frac{1}{x}$.

Tips

- For expressions of the form `exp(-x^2)*erfcx(x)`, use the complementary error function `erfc` instead. This substitution maintains accuracy by avoiding roundoff errors for large values of `x`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Strict single-precision calculations are not supported. In the generated code, single-precision inputs produce single-precision outputs. However, variables inside the function might be double-precision.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`erf` | `erfc` | `erfcinv` | `erfinv`

Introduced before R2006a

erfinv

Inverse error function

Syntax

```
erfinv(x)
```

Description

`erfinv(x)` returns the "Inverse Error Function" on page 1-3861 evaluated for each element of `x`. For inputs outside the interval `[-1 1]`, `erfinv` returns `NaN`.

Examples

Find Inverse Error Function of Value

```
erfinv(0.25)
```

```
ans = 0.2253
```

For inputs outside `[-1,1]`, `erfinv` returns `NaN`. For `-1` and `1`, `erfinv` returns `-Inf` and `Inf`, respectively.

```
erfinv([-2 -1 1 2])
```

```
ans = 1×4

   NaN  -Inf   Inf   NaN
```

Find the inverse error function of the elements of a matrix.

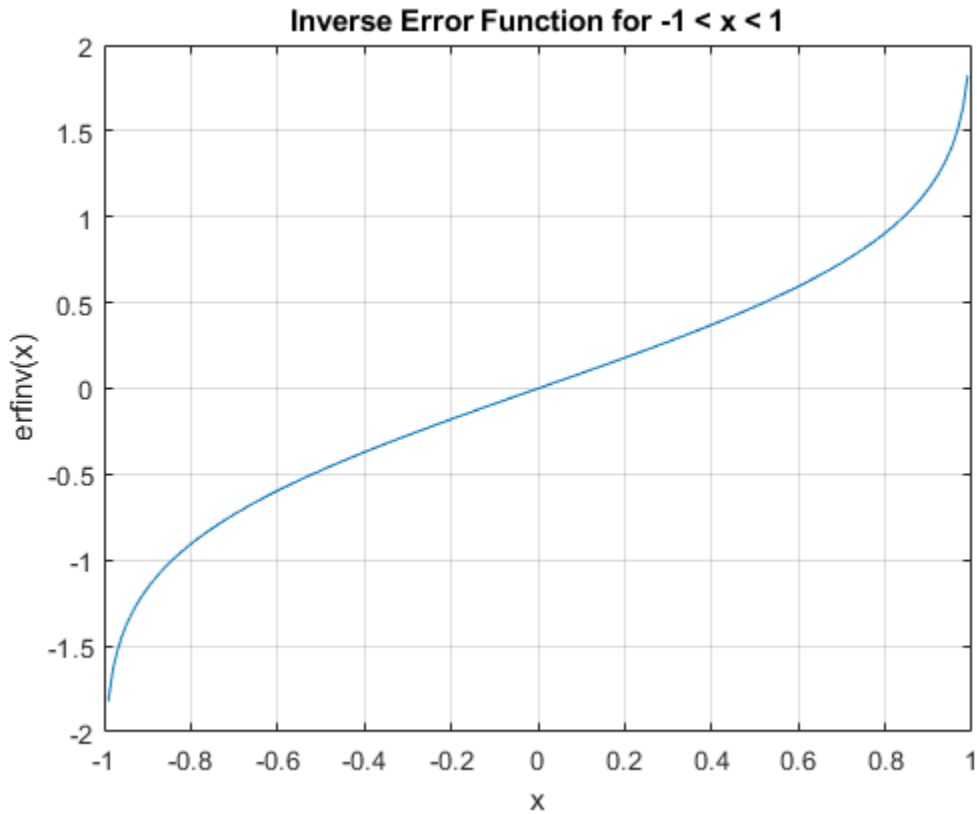
```
M = [0 -0.5; 0.9 -0.2];
erfinv(M)
```

```
ans = 2×2

         0   -0.4769
    1.1631   -0.1791
```

Plot the Inverse Error Function

Plot the inverse error function for -1 < x < 1.

```
x = -1:0.01:1;
y = erfinv(x);
plot(x,y)
grid on
xlabel('x')
ylabel('erfinv(x)')
title('Inverse Error Function for -1 < x < 1')
```



Generate Gaussian Distributed Random Numbers

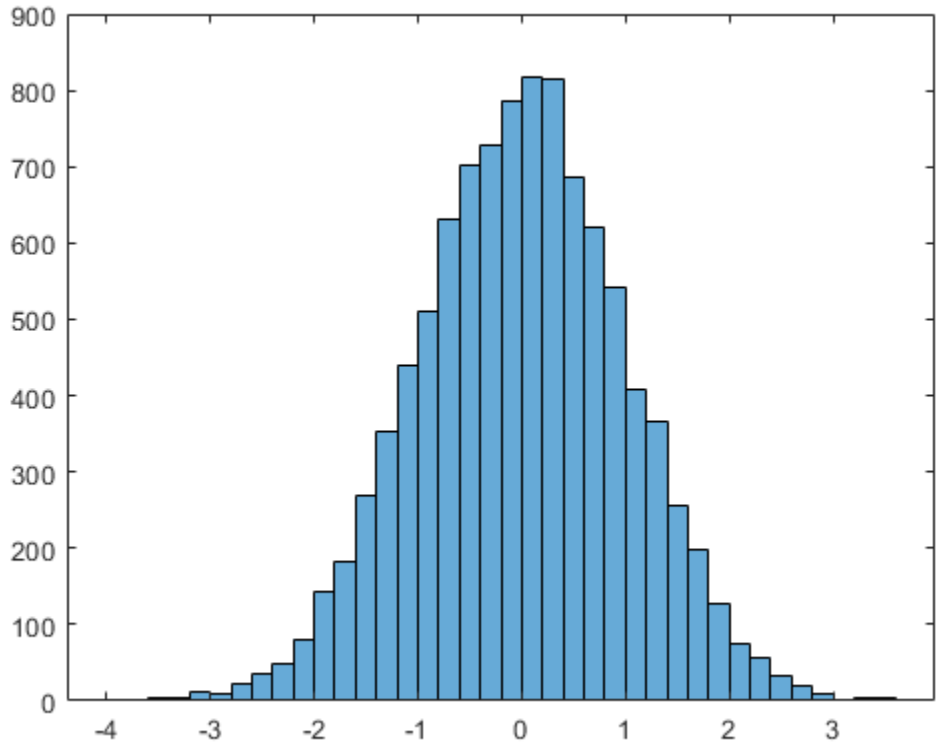
Generate Gaussian distributed random numbers using uniformly distributed random numbers. To convert a uniformly distributed random number x to a Gaussian distributed random number y, use the transform

$$y = \sqrt{2}\mathrm{erf}^{-1}(x).$$

Note that because `x` has the form `-1 + 2*rand(1,10000)`, you can improve accuracy by using `erfcinv` instead of `erfinv`. For details, see “Tips” on page 1-3861.

Generate 10,000 uniformly distributed random numbers on the interval `[-1,1]`. Transform them into Gaussian distributed random numbers. Show that the numbers follow the form of the Gaussian distribution using a histogram plot.

```
rng('default')
x = -1 + 2*rand(1,10000);
y = sqrt(2)*erfinv(x);
h = histogram(y);
```



Input Arguments

x — Input
real number | vector of real numbers | matrix of real numbers | multidimensional array of real numbers

Input, specified as a real number, or a vector, matrix, or multidimensional array of real numbers. `x` cannot be sparse.

Data Types: `single` | `double`

More About

Inverse Error Function

The inverse error function erfinv is defined as the inverse of the error function, such that

$$\mathrm{erfinv}(\mathrm{erf}(x)) = x\,.$$

Tips

- For expressions of the form `erfinv(1-x)`, use the complementary inverse error function `erfcinv` instead. This substitution maintains accuracy. When `x` is close to `1`, then `1 - x` is a small number and may be rounded down to `0`. Instead, replace `erfinv(1-x)` with `erfcinv(x)`.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Strict single-precision calculations are not supported. In the generated code, single-precision inputs produce single-precision outputs. However, variables inside the function might be double-precision.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`erf` | `erfc` | `erfcinv` | `erfcx`

Introduced before R2006a

error

Throw error and display message

Syntax

```
error(msg)
error(msg,A1,...,An)
error(msgID, ___ )

error(errorStruct)

error(correction, ___ )
```

Description

`error(msg)` throws an error and displays an error message.

`error(msg,A1,...,An)` displays an error message that contains formatting conversion characters, such as those used with the MATLAB `sprintf` function. Each conversion character in `msg` is converted to one of the values `A1,...,An`.

`error(msgID, ___ )` includes an error identifier on the exception. The identifier enables you to distinguish errors and to control what happens when MATLAB encounters the errors. You can include any of the input arguments in the previous syntaxes.

`error(errorStruct)` throws an error using the fields in a scalar structure.

`error(correction, ___ )` provides a suggested fix for the exception. You can include any of the input arguments in the previous syntaxes.

Examples

Throw Error

```
msg = 'Error occurred.';
error(msg)
```

```
Error occurred.
```

Throw Error with Formatted Message

Throw a formatted error message with a line break. You must specify more than one input argument with `error` if you want MATLAB to convert special characters (such as `\n`) in the error message. Include information about the class of variable `n` in the error message.

```
n = 7;
if ~ischar(n)
   error('Error. \nInput must be a char, not a %s.',class(n))
end
```

```
Error.
Input must be a char, not a double.
```

If you only use one input argument with `error`, then MATLAB does not convert `\n` to a line break.

```
if ~ischar(n)
   error('Error. \nInput must be a char.')
end
```

```
Error. \nInput must be a char.
```

Throw an error with an identifier.

```
if ~ischar(n)
   error('MyComponent:incorrectType',...
       'Error. \nInput must be a char, not a %s.',class(n))
end
```

```
Error.
Input must be a char, not a double.
```

Use the `MException.last` to view the last uncaught exception.

```
exception = MException.last
```

```
exception =

  MException with properties:

    identifier: 'MyComponent:incorrectType'
       message: 'Error.
Input must be a char, not a double.'
         cause: {0x1 cell}
         stack: [0x1 struct]
```

Throw Error Using Structure

Create structure with message and identifier fields. To keep the example simple, do not use the stack field.

```
errorStruct.message = 'Data file not found.';
errorStruct.identifier = 'MyFunction:fileNotFound';
```

```
errorStruct =

       message: 'Data file not found.'
    identifier: 'MyFunction:fileNotFound'
```

Throw the error.

```
error(errorStruct)
```

```
Data file not found.
```

Throw Error with Suggested Fix

Create a function `hello` that requires one input argument. Add a suggested input argument `"world"` to the error message.

```
function hello(audience)
if nargin < 1
    aac = matlab.lang.correction.AppendArgumentsCorrection('"world"');
    error(aac, 'MATLAB:notEnoughInputs', 'Not enough input arguments.')
end
fprintf("Hello, %s!\n", audience)
end
```

Call the function without an argument.

```
hello
```

```
Error using hello (line 4)
Not enough input arguments.

Did you mean:
>> hello("world")
```

Input Arguments

msg — Information about error

character vector | string scalar

Information about the error, specified as a character vector or string scalar. This message displays as the error message. To format the message, use escape sequences, such as `\t` or `\n`. You also can use any format specifiers supported by the `sprintf` function, such as `%s` or `%d`. Specify values for the conversion specifiers via the `A1,...,An` input arguments. For more information, see “Formatting Text”.

Note You must specify more than one input argument with `error` if you want MATLAB to convert special characters (such as `\t`, `\n`, `%s`, and `%d`) in the error message.

Example: `'File not found.'`

msgID — Identifier for error

character vector | string scalar

Identifier for the error, specified as a character vector or string scalar. Use the error identifier to help identify the source of the error or to control a selected subset of the errors in your program.

The error identifier includes a `component` and `mnemonic`. The identifier must always contain a colon and follows this simple format: `component:mnemonic`. The `component` and `mnemonic` fields must each begin with a letter. The remaining characters can be alphanumerics (A–Z, a–z, 0–9) and underscores. No whitespace characters can appear anywhere in `msgID`. For more information, see “Message Identifiers”.

Example: `'MATLAB:singularMatrix'`

Example: `'MATLAB:narginchk:notEnoughInputs'`

`A1,...,An` — Values
character vector | string scalar | numeric scalar

Values that replace the conversion specifiers in `msg`, specified as a character vector, string scalar, or numeric scalar.

`errorStruct` — Error reporting information
scalar structure

Error reporting information, specified as a scalar structure. The structure must contain at least one of these fields.

`message`	Error message. For more information, see `msg`.
`identifier`	Error message identifier. For more information, see `msgID`.
`stack`	Stack field for the error. When `errorStruct` includes a `stack` field, `error` uses it to set the stack field of the error. When you specify `stack`, use the absolute file name and the entire sequence of functions that nests the function in the stack frame. This character vector is the same as the one returned by `dbstack('-completenames')`.

`correction` — Suggested fix for this exception
`matlab.lang.correction.AppendArgumentsCorrection` object | `matlab.lang.correction.ConvertToFunctionNotationCorrection` object | `matlab.lang.correction.ReplaceIdentifierCorrection` object

Suggested fix for this exception, specified as a `matlab.lang.correction.AppendArgumentsCorrection`, `matlab.lang.correction.ConvertToFunctionNotationCorrection`, or `matlab.lang.correction.ReplaceIdentifierCorrection` object.

Tips

- When you throw an error, MATLAB captures information about it and stores it in a data structure that is an object of the `MException` class. You can access information in the exception object by using `try/catch`. Or, if your program terminates because of an exception and returns control to the Command Prompt, you can use `MException.last`.

- MATLAB does not cease execution of a program if an error occurs within a `try` block. In this case, MATLAB passes control to the `catch` block.
- If all inputs to `error` are empty, MATLAB does not throw an error.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Has no effect in standalone code even when run-time error detection is enabled. See "Run-Time Error Detection and Reporting in Standalone C/C++ Code" (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

See Also

`MException` | `MException.last` | `assert` | `dbstack` | `errordlg` | `try` | `warning`

Topics

"Capture Information About Exceptions"

Introduced before R2006a

errorbar

Line plot with error bars

Syntax

```
errorbar(y,err)
errorbar(x,y,err)
errorbar(x,y,neg,pos)
errorbar( ___ ,ornt)

errorbar(x,y,yneg,ypos,xneg,xpos)

errorbar( ___ ,linespec)
errorbar( ___ ,Name,Value)
errorbar(ax, ___ )
e = errorbar( ___ )
```

Description

`errorbar(y,err)` creates a line plot of the data in `y` and draws a vertical error bar at each data point. The values in `err` determine the lengths of each error bar above and below the data points, so the total error bar lengths are double the `err` values.

`errorbar(x,y,err)` plots `y` versus `x` and draws a vertical error bar at each data point.

`errorbar(x,y,neg,pos)` draws a vertical error bar at each data point, where `neg` determines the length below the data point and `pos` determines the length above the data point, respectively.

`errorbar( ___ ,ornt)` sets the orientation of the error bars. Specify `ornt` as `'horizontal'` for horizontal error bars or `'both'` for both horizontal and vertical error bars. The default for `ornt` is `'vertical'`, which draws vertical error bars. Use this option after any of the previous input argument combinations.

`errorbar(x,y,yneg,ypos,xneg,xpos)` plots `y` versus `x` and draws both horizontal and vertical error bars. The `yneg` and `ypos` inputs set the lower and upper lengths of the

vertical error bars, respectively. The `xneg` and `xpos` inputs set the left and right lengths of the horizontal error bars.

`errorbar( ___ ,linespec)` sets the line style, marker symbol, and color. For example, `'--ro'` plots a dashed, red line with circle markers. The line style affects only the line and not the error bars.

`errorbar( ___ ,Name,Value)` modifies the appearance of the line and error bars using one or more name-value pair arguments. For example, `'CapSize',10` sets the lengths of the caps at the end of each error bar to 10 points.

`errorbar(ax, ___ )` creates the plot in the axes specified by `ax` instead of in the current axes. Specify the axes as the first input argument.

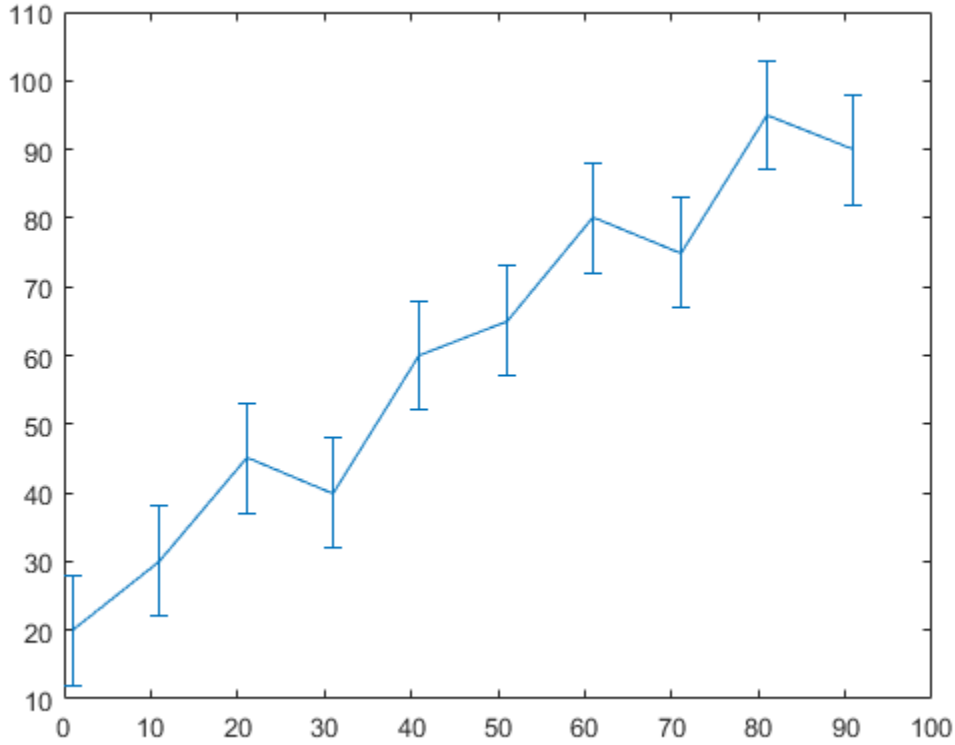
`e = errorbar( ___ )` returns one `ErrorBar` object when `y` is a vector. If `y` is a matrix, then it returns one `ErrorBar` object per column in `y`. Use `e` to modify properties of a specific `ErrorBar` object after it is created. For a list of properties, see Errorbar.

Examples

Plot Vertical Error Bars of Equal Length

Create vectors `x` and `y`. Plot `y` versus `x`. At each data point, display vertical error bars that are equal in length.

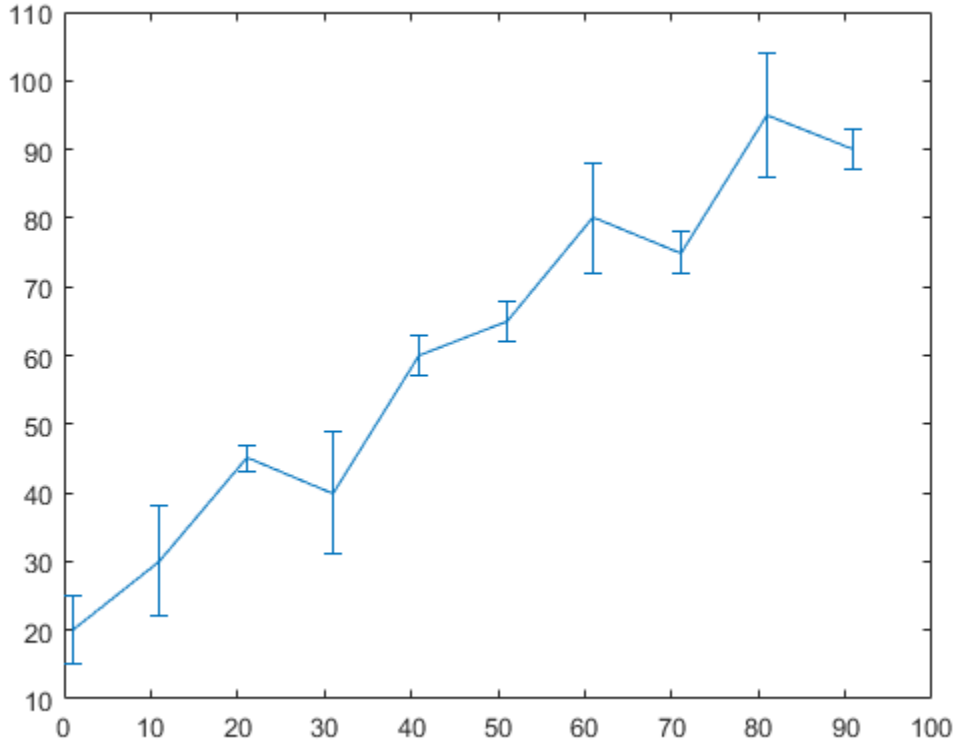
```
x = 1:10:100;
y = [20 30 45 40 60 65 80 75 95 90];
err = 8*ones(size(y));
errorbar(x,y,err)
```



Plot Vertical Error Bars that Vary in Length

Create a line plot with error bars at each data point. Vary the lengths of the error bars.

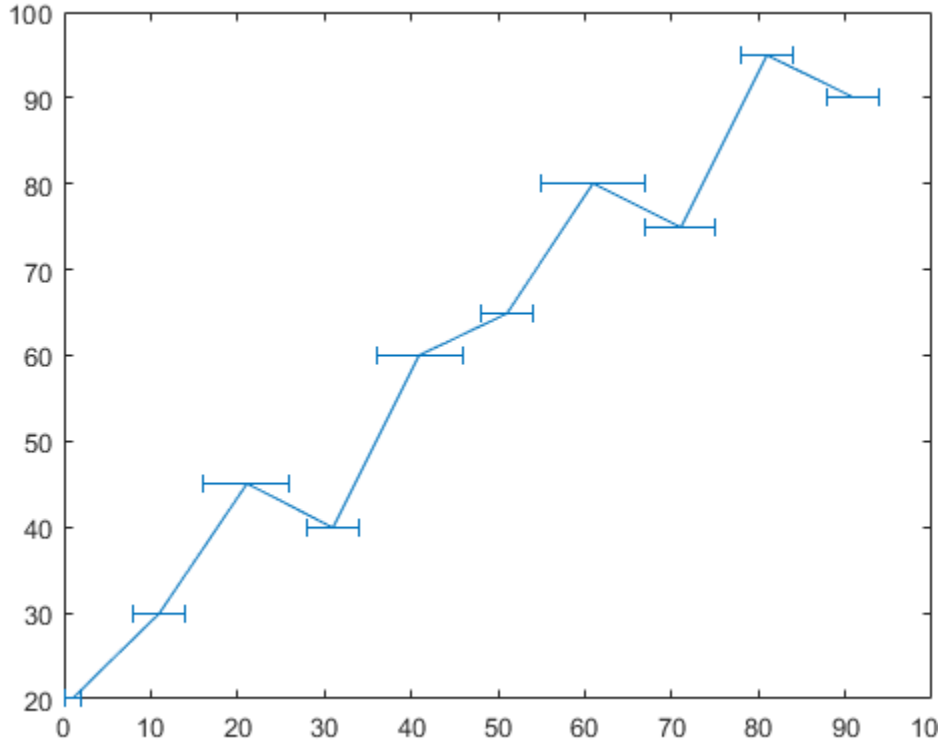
```
x = 1:10:100;
y = [20 30 45 40 60 65 80 75 95 90];
err = [5 8 2 9 3 3 8 3 9 3];
errorbar(x,y,err)
```



Plot Horizontal Error Bars

Create a line plot with horizontal error bars at each data point.

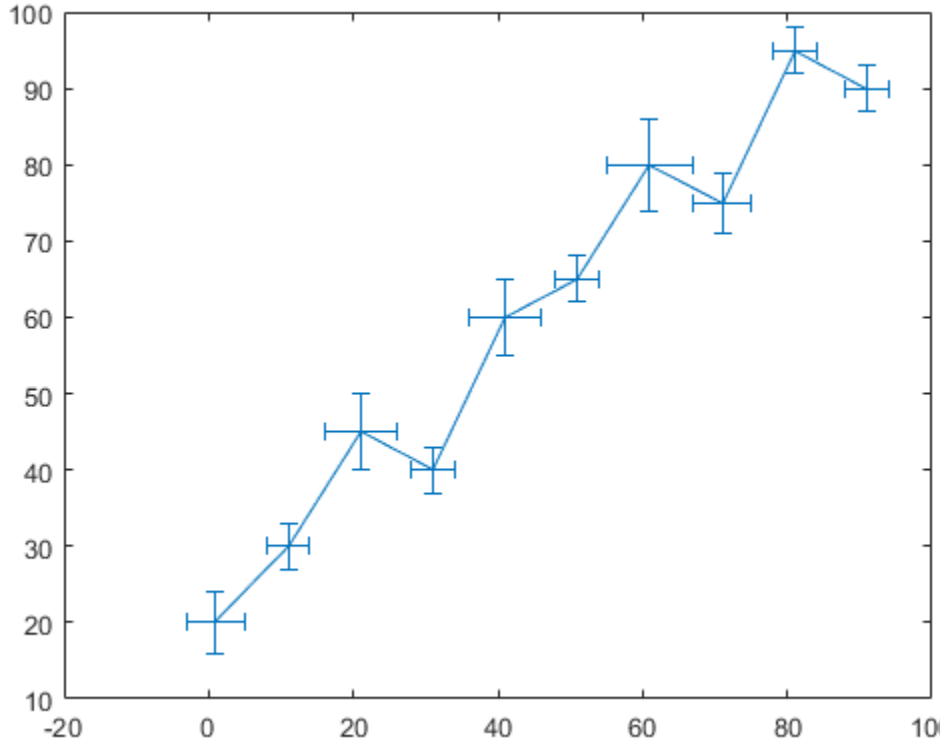
```
x = 1:10:100;
y = [20 30 45 40 60 65 80 75 95 90];
err = [1 3 5 3 5 3 6 4 3 3];
errorbar(x,y,err,'horizontal')
```



Plot Vertical and Horizontal Error Bars

Create a line plot with both vertical and horizontal error bars at each data point.

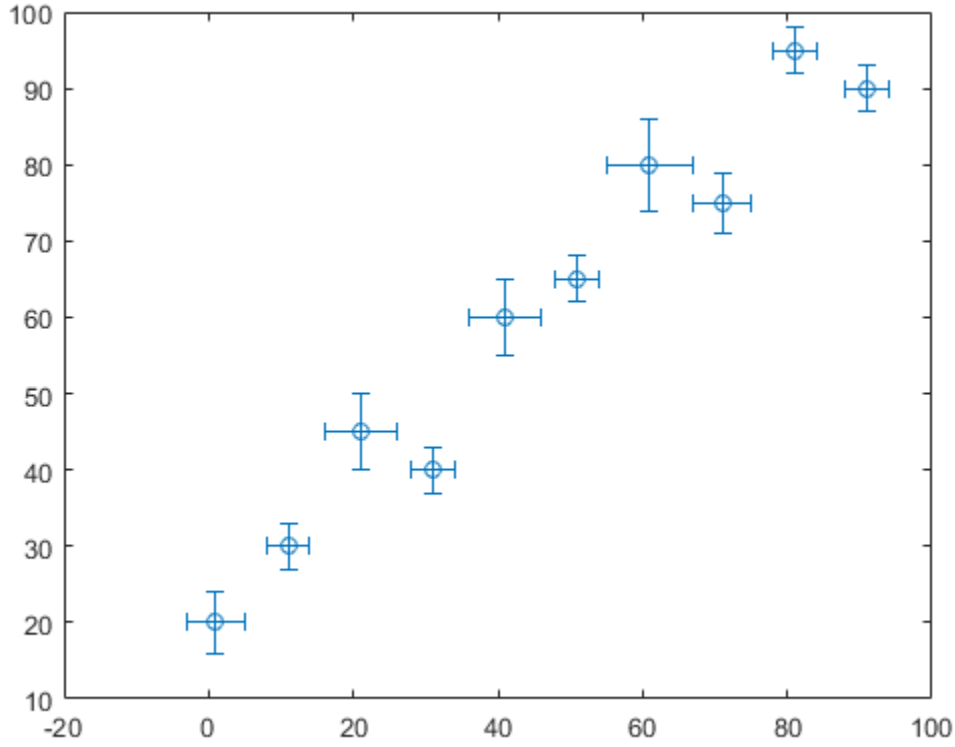
```
x = 1:10:100;
y = [20 30 45 40 60 65 80 75 95 90];
err = [4 3 5 3 5 3 6 4 3 3];
errorbar(x,y,err,'both')
```



Plot Error Bars with No Line

Plot vectors `y` versus `x`. At each data point, display a circle marker with both vertical and horizontal error bars. Do not display the line that connects the data points by omitting the line style option for the `linespec` input argument.

```
x = 1:10:100;
y = [20 30 45 40 60 65 80 75 95 90];
err = [4 3 5 3 5 3 6 4 3 3];
errorbar(x,y,err,'both','o')
```

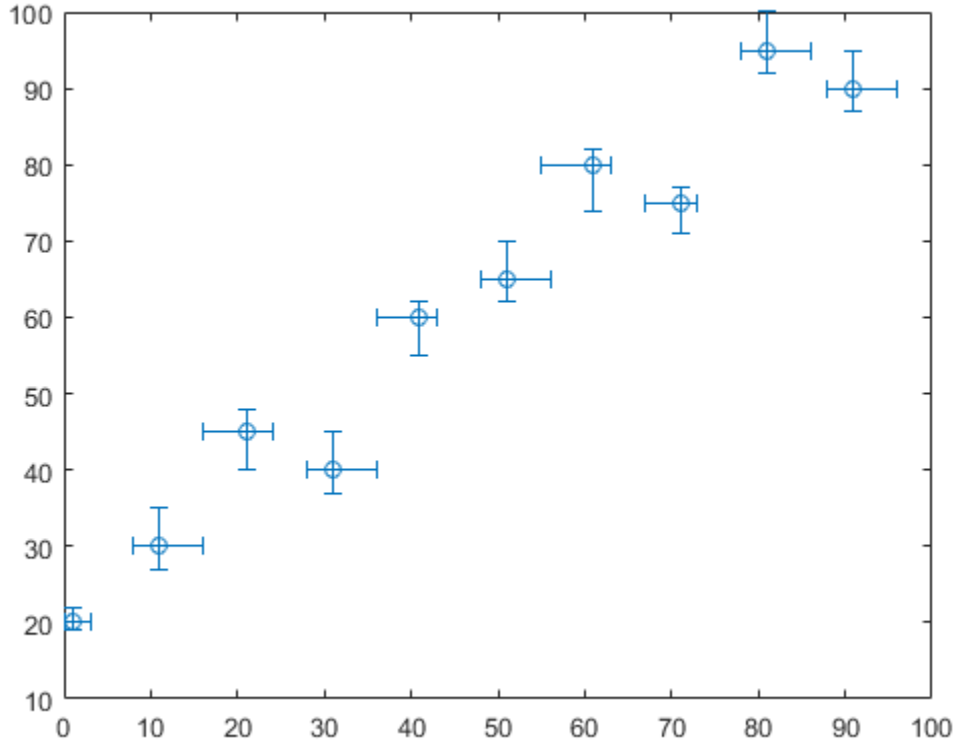


Control Error Bars Lengths in All Directions

Display both vertical and horizontal error bars at each data point. Control the lower and upper lengths of the vertical error bars using the `yneg` and `ypos` input argument options, respectively. Control the left and right lengths of the horizontal error bars using the `xneg` and `xpos` input argument options, respectively.

```
x = 1:10:100;
y = [20 30 45 40 60 65 80 75 95 90];
yneg = [1 3 5 3 5 3 6 4 3 3];
ypos = [2 5 3 5 2 5 2 2 5 5];
```

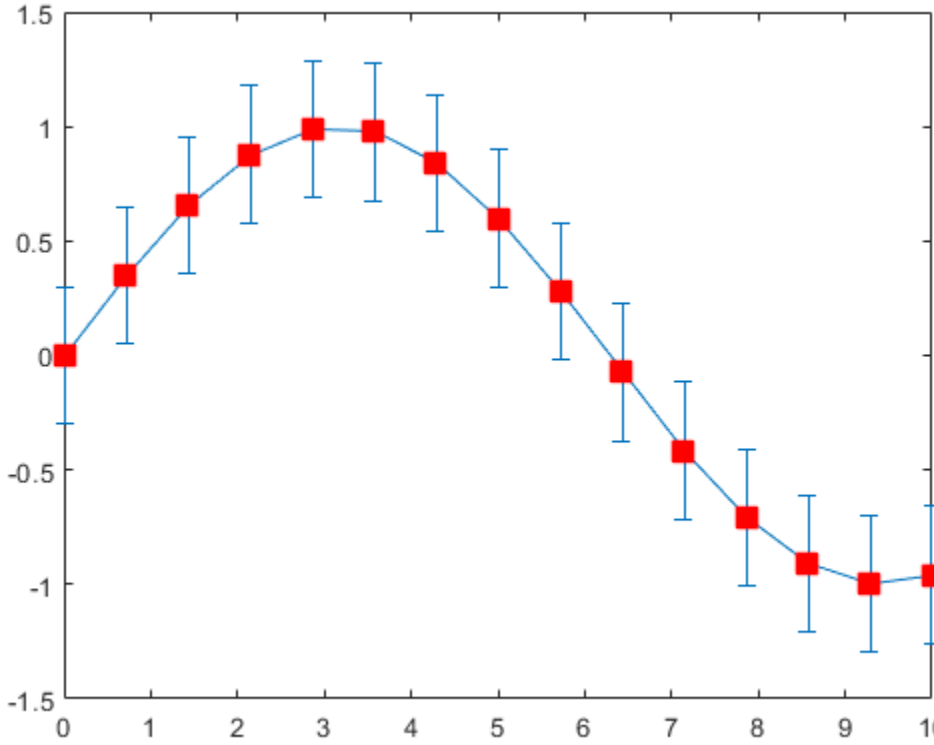
```
xneg = [1 3 5 3 5 3 6 4 3 3];
xpos = [2 5 3 5 2 5 2 2 5 5];
errorbar(x,y,yneg,ypos,xneg,xpos,'o')
```



Add Colored Markers to Each Data Point

Create a line plot with error bars. At each data point, display a marker. Control the appearance of the marker using name-value pair arguments. Use `MarkerSize` to specify the marker size in points. Use `MarkerEdgeColor` and `MarkerFaceColor` to specify the marker outline and interior colors, respectively. Set the colors to either a character vector of a color name, such as `'red'`, or an RGB triplet.

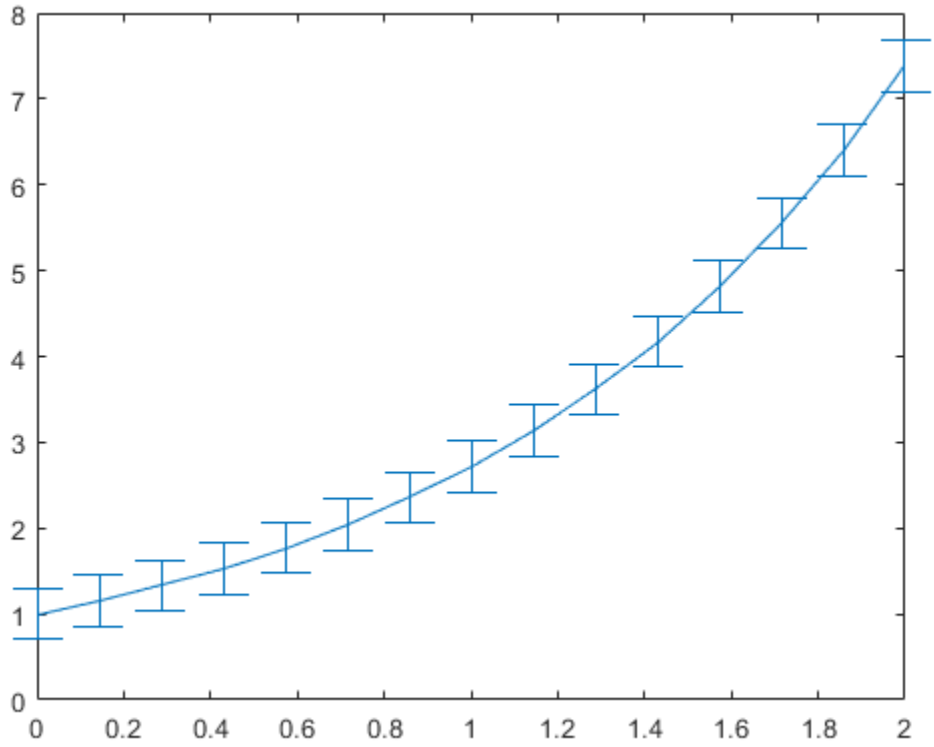
```
x = linspace(0,10,15);
y = sin(x/2);
err = 0.3*ones(size(y));
errorbar(x,y,err,'-s','MarkerSize',10,...
    'MarkerEdgeColor','red','MarkerFaceColor','red')
```



Control Error Bar Cap Size

Control the size of the caps at the end of each error bar by setting the `CapSize` property to a positive value in points.

```
x = linspace(0,2,15);
y = exp(x);
err = 0.3*ones(size(y));
errorbar(x,y,err,'CapSize',18)
```

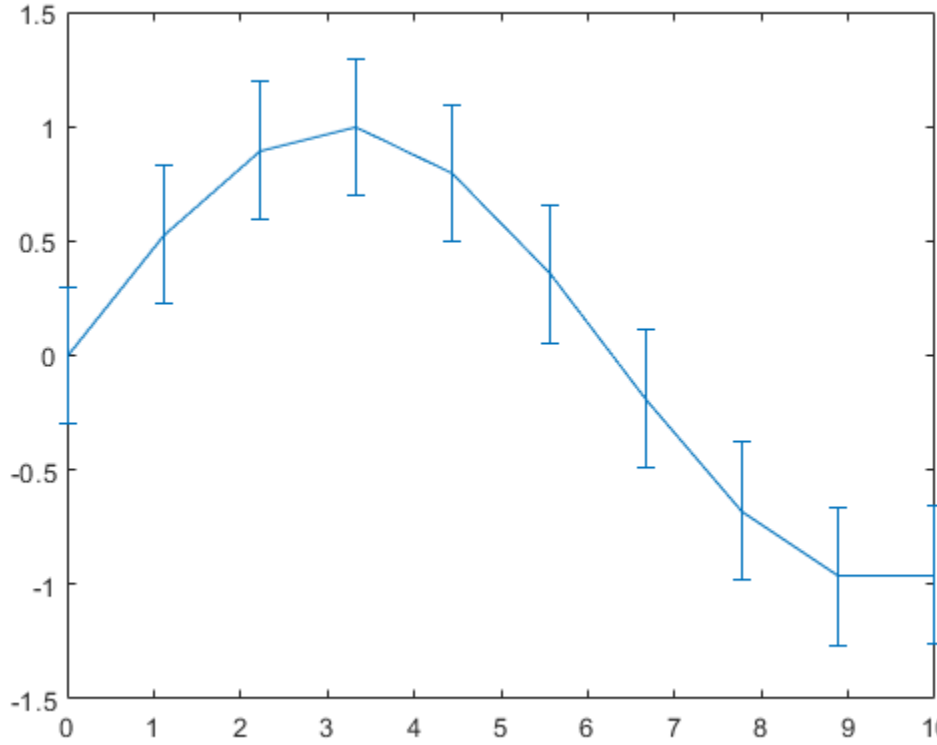


Modify Error Bars After Creation

Create a line plot with error bars. Assign the errorbar object to the variable e.

```
x = linspace(0,10,10);
y = sin(x/2);
```

```
err = 0.3*ones(size(y));
e = errorbar(x,y,err)
```

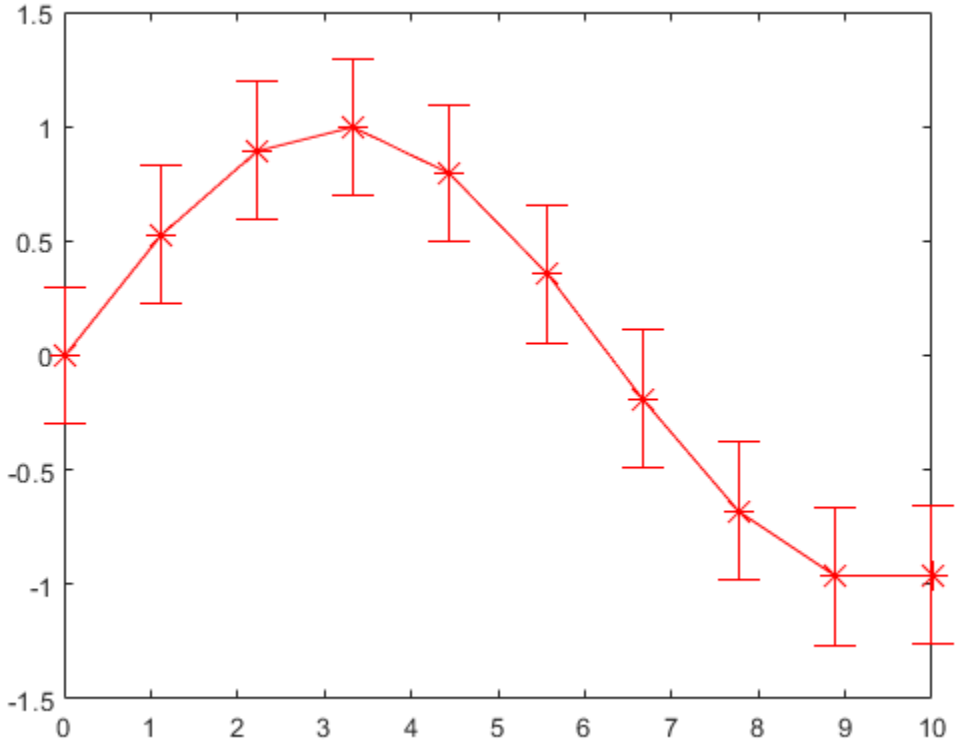



```
e =
  ErrorBar with properties:

              Color: [0 0.4470 0.7410]
          LineStyle: '-'
          LineWidth: 0.5000
             Marker: 'none'
              XData: [1x10 double]
              YData: [1x10 double]
      XNegativeDelta: [1x0 double]
      XPositiveDelta: [1x0 double]
```

```
    YNegativeDelta: [1x10 double]
    YPositiveDelta: [1x10 double]

  Show all properties
```

Use `e` to access properties of the errorbar object after it is created. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead.

```
e.Marker = '*';
e.MarkerSize = 10;
e.Color = 'red';
e.CapSize = 15;
```



Input Arguments

y — *y* values
vector | matrix

y values, specified as a vector or a matrix.

- If `y` is a vector, then `errorbar` plots one line.
- If `y` is a matrix, then `errorbar` plots a separate line for each column in `y`.

Example: `y = [4 3 5 2 2 4];`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

x — *x* values
vector | matrix

x values, specified as a vector or a matrix. `x` must be the same size as `y`.

Example: `x = 0:10:100;`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

err — Error bar lengths for symmetrical error bars
vector | matrix

Error bar lengths for symmetrical error bars, specified as a vector or a matrix. `err` must be the same size as `y`. If you do not want to draw an error bar at a particular data point, then specify the length as `NaN`.

Example: `err = [.4 .3 .5 .2 .4 .5];`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

neg — Error bar lengths in negative direction
vector | matrix | `[]`

Error bar lengths in the negative direction, specified as a vector or matrix the same size as `y` or as an empty array `[]`.

- For vertical error bars, `neg` sets the length of the error bars below the data points.
- For horizontal error bars, `neg` sets the length of the error bars to the left of the data points.

If you do not want to draw the lower part of the error bar at a *particular* data point, then specify the length as `NaN`. If you do not want to draw the lower part of the error bar at *any* data point, then set `neg` to an empty array.

Example: `neg = [.4 .3 .5 .2 .4 .5];`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

pos — Error bar lengths in positive direction
vector | matrix | `[]`

Error bar lengths in the positive direction, specified as a vector or matrix the same size as `y` or as an empty array `[]`.

- For vertical error bars, `pos` sets the length of the error bars above the data points.
- For horizontal error bars, `pos` sets the length of the error bars to the right of the data points.

If you do not want to draw the upper part of the error bar at a *particular* data point, then specify the length as `NaN`. If you do not want to draw the upper part of the error bar at *any* data point, then set `pos` to an empty array.

Example: `pos = [.4 .3 .5 .2 .4 .5];`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

yneg — Vertical error bar lengths below data points
vector | matrix | `[]`

Vertical error bar lengths below the data points, specified as a vector or matrix the same size as `y` or as an empty array `[]`. Specify the values in data units.

If you do not want to draw the lower part of the error bar at a *particular* data point, then specify the value as `NaN`. If you do not want to draw the lower part of the error bar at *any* data point, then set `yneg` to an empty array.

Example: `yneg = [.4 .3 .5 .2 .4 .5];`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

ypos — Vertical error bar lengths above data points
vector | matrix | `[]`

Vertical error bar lengths above the data points, specified as a vector or matrix the same size as `y` or as an empty array `[]`. Specify the values in data units.

If you do not want to draw the upper part of the error bar at a *particular* data point, then specify the length as `NaN`. If you do not want to draw the upper part of the error bar at *any* data point, then set `ypos` to an empty array.

Example: `ypos = [.4 .3 .5 .2 .4 .5];`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

xneg — Horizontal error bar lengths to left of data points
vector | matrix | `[]`

Horizontal error bar lengths to the left of the data points, specified as a vector or matrix the same size as `y` or as an empty array `[]`. Specify the values in data units.

If you do not want to draw the left part of the error bar at a *particular* data point, then specify the length as `NaN`. If you do not want to draw the left part of the error bar at *any* data point, then set `xneg` to an empty array.

Example: `xneg = [.4 .3 .5 .2 .4 .5];`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

xpos — Horizontal error bar lengths to right of data points
vector | matrix | `[]`

Horizontal error bar lengths to the right of the data points, specified as a vector or matrix the same size as `y` or as an empty array `[]`. Specify the values in data units.

If you do not want to draw the right part of the error bar at a *particular* data point, then specify the length as `NaN`. If you do not want to draw the right part of the error bar at *any* data point, then set `xpos` to an empty array.

Example: `xpos = [.4 .3 .5 .2 .4 .5];`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ornt` — Error bar orientation
`'vertical'` (default) | `'horizontal'` | `'both'`

Error bar orientation, specified as one of these values:

- `'vertical'` — Vertical error bars
- `'horizontal'` — Horizontal error bars
- `'both'` — Vertical and horizontal error bars

Example: `errorbar(x,y,err,'horizontal')`

`linespec` — Line style, marker symbol, and color
character vector | string

Line style, marker symbol, and color, specified as a character vector or string containing symbols. The symbols can appear in any order. You do not need to specify all three characteristics (line style, marker symbol, and color). For example, if you omit the line style and specify the marker, then the plot shows only the markers and no line. The line style affects only the line and not the error bars.

Example: `errorbar(x,y,err,'--or')` plots a red, dashed line with circle markers and red error bars at the data points.

Line Style	Description
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Marker	Description
`o`	Circle
`+`	Plus sign
`*`	Asterisk
`.`	Point

Marker	Description
x	Cross
s	Square
d	Diamond
^	Upward-pointing triangle
v	Downward-pointing triangle
>	Right-pointing triangle
<	Left-pointing triangle
p	Pentagram
h	Hexagram

Color	Description
y	yellow
m	magenta
c	cyan
r	red
g	green
b	blue
w	white
k	black

ax — Axes object
current axes (default) | axes object

Axes object. If you do not specify the axes, then `errorbar` plots into the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The properties listed here are only a subset. For a complete list, see Errorbar.

Example: `errorbar(y,err,'LineWidth',2)` specifies a line width of 2 points.

`CapSize` — Length of caps at end of error bars
`6` (default) | positive value in points

Length of caps at end of error bars, specified as a positive value in points.

Example: `errorbar(x,y,err,'CapSize',10)`

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions
`bar` | `corrcoef` | `plot` | `std`

Properties
Errorbar

Introduced before R2006a

ErrorBar Properties

Error bar chart appearance and behavior

Description

`ErrorBar` properties control the appearance and behavior of an `ErrorBar` object. By changing property values, you can modify certain aspects of the error bar chart.

Starting in R2014b, you can use dot notation to query and set properties.

```
e = errorbar(...);
s = e.LineStyle;
e.LineStyle = ':';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Color and Styling

`Color` — Line color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`ColorMode` — Control how `Color` is set

`'auto'` (default) | `'manual'`

Control how the `Color` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the `Color` property by selecting a color from the `ColorOrder` property of the axes.
- `'manual'` — You control the value of the `Color` property manually, either by setting the value of the `Color` property directly on the object, or by including a color in the `LineSpec` argument when you call a plotting function.

If you change the value of the `Color` property manually, MATLAB changes the value of the `ColorMode` property to `'manual'`.

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineStyleMode` — Control how `LineStyle` is set
`'auto'` (default) | `'manual'`

Control how the `LineStyle` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the `LineStyle` property by selecting a line style from the `LineStyleOrder` property of the axes.
- `'manual'` — You control the value of the `LineStyle` property manually, either by setting the value of the `LineStyle` property directly on the object, or by specifying the `LineSpec` argument when you call a plotting function.

If you change the value of the `LineStyle` property manually, MATLAB changes the value of the `LineStyleMode` property to `'manual'`.

`LineWidth` — Line width

`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

`CapSize` — Length of caps at end of error bars

`6` (default) | positive value in points

Length of caps at end of error bars, specified as a positive value in points.

Example: `errorbar(x,y,err,'CapSize',10)`

`AlignVertexCenters` — Sharp vertical and horizontal lines

`'on'` (default) | `'off'`

Sharp vertical and horizontal lines, specified as `'on'` or `'off'`.

If the associated figure has a `GraphicsSmoothing` property set to `'on'` and a `Renderer` property set to `'opengl'`, then the figure applies a smoothing technique to plots. In some cases, this smoothing technique can cause vertical and horizontal lines to appear uneven in thickness or color. Use the `AlignVertexCenters` property to eliminate the uneven appearance.

- `'on'` — Sharpen vertical and horizontal lines to eliminate an uneven appearance.
- `'off'` — Do not sharpen vertical or horizontal lines. The lines might appear uneven in thickness or color.

Note You must have a graphics card that supports this feature. To see if the feature is supported, call the `rendererinfo` function. If it is supported, `rendererinfo` returns value of `1` for `info.Details.SupportsAlignVertexCenters`.

Markers

`Marker` — Marker symbol

`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | ...

Marker symbol, specified as one of the values listed in this table. By default, the object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerMode` — Control how `Marker` is set
`'auto'` (default) | `'manual'`

Control how the `Marker` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the object's `Marker` property.
- `'manual'` — You control the value of the `Marker` property manually, either by setting the value of the `Marker` property directly on the object, or by including a marker in the `LineSpec` argument when you call a plotting function.

If you change the value of the `Marker` property manually, MATLAB changes the value of the `MarkerMode` property to `'manual'`.

`MarkerSize` — Marker size
6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

`MarkerEdgeColor` — Marker outline color
`'auto'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'auto'` uses the same color as the `Color` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` option uses the same color as the `Color` property of the parent axes. If you specify `'auto'` and the axes plot box is invisible, the marker fill color is the color of the figure.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Line Data

`XData` — *x* values

`[]` (default) | vector

x values, specified as a vector. The input argument `X` to the `errorbar` function sets the *x* values. If you do not specify `X`, then `errorbar` uses the indices of `YData` as the *x* values. `XData` and `YData` must have equal lengths.

Example: `1:10`

`XDataMode` — Selection mode for `XData`

`'auto'` (default) | `'manual'`

Selection mode for `XData`, specified as one of these values:

- `'auto'` — Use the indices of the values in `YData`.
- `'manual'` — Use manually specified values. To specify the values, set the `XData` property or specify the input argument `X` to the plotting function.

XDataSource — Variable linked to XData
'' (default) | character vector | string

Variable linked to XData, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the XData.

By default, there is no linked variable so the value is an empty character vector, ''. If you link a variable, then MATLAB does not update the XData values immediately. To force an update of the data values, use the refreshdata function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: 'x'

YData — *y* values
[ ] (default) | vector

y values, specified as a vector. The input argument Y to the errorbar function sets the *y* values. XData and YData must have equal lengths.

YDataSource — Variable linked to YData
'' (default) | character vector | string

Variable linked to YData, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the YData.

By default, there is no linked variable so the value is an empty character vector, ''. If you link a variable, then MATLAB does not update the YData values immediately. To force an update of the data values, use the refreshdata function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: 'y'

Error Bar Data

`YNegativeDelta` — Vertical error bar lengths below data points

vector | `[]`

Vertical error bar lengths below the data points, specified as a vector the same length as `YData` or as an empty array `[]`. Specify the values in data units.

- If you do not want to draw the lower part of the error bar at a *particular* data point, then specify the value as `NaN`.
- If you do not want to draw the lower part of the error bar at *any* data point, then set the property to an empty array.

Example: `e.YNegativeDelta = [.4 .3 .5 .2 .4 .5];`

`YNegativeDeltaSource` — Variable linked to `YNegativeDelta`

`''` (default) | character vector or string containing MATLAB workspace variable

Variable linked to `YNegativeDelta`, specified as a character vector or string containing a MATLAB workspace variable. MATLAB evaluates the variable to generate the `YNegativeDelta` values.

By default, there is no linked variable, so the value is an empty character vector, `''`. When you change the variable for this property, MATLAB does not update the `YNegativeDelta` values. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning. To render the graph, you must change all data source properties to appropriate values.

`YPositiveDelta` — Vertical error bar lengths above data points

vector | `[]`

Vertical error bar lengths above the data points, specified as a vector the same length as `YData` or as an empty array `[]`. Specify the values in data units.

- If you do not want to draw the upper part of the error bar at a particular data point, then specify the value as `NaN`.

- If you do not want to draw the upper part of the error bar at any data point, then set the property to an empty array.

Example: `e.YPositiveDelta = [.4 .3 .5 .2 .4 .5];`

`YPositiveDeltaSource` — Variable linked to `YPositiveDelta`
`''` (default) | character vector or string containing MATLAB workspace variable

Variable linked to `YPositiveDelta`, specified as a character vector or string containing a MATLAB workspace variable. MATLAB evaluates the variable to generate the `YPositiveDelta` values.

By default, there is no linked variable, so the value is an empty character vector, `''`. When you change the variable for this property, MATLAB does not update the `YPositiveDelta` values. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning. To render the graph, you must change all data source properties to appropriate values.

`XNegativeDelta` — Horizontal error bar lengths to left of data points
vector | `[]`

Horizontal error bar lengths to the left of the data points, specified as a vector the same length as `YData` or as an empty array `[]`. Specify the values in data units.

- If you do not want to draw the left part of the error bar at a *particular* data point, then specify the value as `NaN`.
- If you do not want to draw the left part of the error bar at *any* data point, then set the property to an empty array.

Example: `e.XNegativeDelta = [.4 .3 .5 .2 .4 .5];`

`XNegativeDeltaSource` — Variable linked to `XNegativeDelta`
`''` (default) | character vector or string containing MATLAB workspace variable

Variable linked to `XNegativeDelta`, specified as a character vector or string containing a MATLAB workspace variable. MATLAB evaluates the variable to generate the `XNegativeDelta` values.

By default, there is no linked variable, so the value is an empty character vector, `''`. When you change the variable for this property, MATLAB does not update the `XNegativeDelta` values. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning. To render the graph, you must change all data source properties to appropriate values.

`XPositiveDelta` — Horizontal error bar lengths to right of data points

vector | `[]`

Horizontal error bar lengths to the right of the data points, specified as a vector the same length as `YData` or as an empty array `[]`. Specify the values in data units.

- If you do not want to draw the right part of the error bar at a *particular* data point, then specify the value as `NaN`.
- If you do not want to draw the right part of the error bar at *any* data point, then set the property to an empty array.

Example: `e.XPositiveDelta = [.4 .3 .5 .2 .4 .5];`

`XPositiveDeltaSource` — Variable linked to `XPositiveDelta`

`''` (default) | character vector or string containing MATLAB workspace variable

Variable linked to `XPositiveDelta`, specified as a character vector or string containing a MATLAB workspace variable. MATLAB evaluates the variable to generate the `XPositiveDelta` values.

By default, there is no linked variable, so the value is an empty character vector, `''`. When you change the variable for this property, MATLAB does not update the `XPositiveDelta` values. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning. To render the graph, you must change all data source properties to appropriate values.

LData — Error bar lengths below data points (not recommended)

[ ] (default) | vector

Note This property is not recommended. Use the `YNegativeDelta` property instead.

Errorbar lengths below the data points, specified as a vector with length equal to `XData` and `YData`. Specify the values in data units.

Example: `1:10`

LDataSource — Variable linked to LData (not recommended)

`''` (default) | character vector or string containing MATLAB workspace variable

Note This property is not recommended. Use the `YNegativeDeltaSource` property instead.

Variable linked to `LData`, specified as a character vector or string containing a MATLAB workspace variable. MATLAB evaluates the variable to generate the `LData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you change the variable for this property, then MATLAB does not update the `LData` values. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

UData — Error bar lengths above data points (not recommended)

[ ] (default) | vector

Note This property is not recommended. Use the `YPositiveDelta` property instead.

Error bar lengths above the data points, specified as a vector with length equal to `XData` and `YData`. Specify the values in data units.

Example: `1:10`

`UDataSource` — Variable linked to `UData` (not recommended)

`''` (default) | character vector or string containing MATLAB workspace variable

---

Note This property is not recommended. Use the `YPositiveDeltaSource` property instead.

Variable linked to `UData`, specified as a character vector or string containing a MATLAB workspace variable. MATLAB evaluates the variable to generate the `UData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you change the variable for this property, then MATLAB does not update the `UData` values. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Legend

`DisplayName` — Legend label

`''` (default) | character vector | string scalar

Legend label, specified as a character vector or string scalar. The legend does not display until you call the `legend` command. If you do not specify the text, then `legend` sets the label using the form `'dataN'`.

`Annotation` — Control for including or excluding object from legend

`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`DataTipTemplate` — Data tip content

`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

`Clipping` — Clipping of object to axes limits

`'on'` (default) | `'off'`

Clipping of the object to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the object that are outside the axes limits.
- `'off'` — Display the entire object, even if parts of it appear outside the axes limits. Parts of the object might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the object so that it is larger than the original plot.

The `Clipping` property of the axes that contains the object must be set to `'on'`. Otherwise, this property has no effect. For more information about the clipping behavior, see the `Clipping` property of the axes.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see "Callback Definition".

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you

do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks when visible. The `Visible` property must be set to `'on'` and you must click a part of the `ErrorBar` object that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The `HitTest` property determines if the `ErrorBar` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `ErrorBar` object passes the click to the object below it in the current view of the figure window. The `HitTest` property of the `ErrorBar` object has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `ErrorBar` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `ErrorBar` object that has one of these:

- `HitTest` property set to `'on'`
- `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

Note The `PickableParts` property determines if the `ErrorBar` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent

`Axes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `Group`, or `Transform` object.

`Children` — Children

empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the chart object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'errorbar'`

This property is read-only.

Type of graphics object, returned as `'errorbar'`. Use this property to find all objects of a given type within a plotting hierarchy, such as searching for the type using `findobj`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`errorbar`

Topics

“Access Property Values”
“Graphics Object Properties”

Introduced before R2006a

errordlg

Create error dialog box

Note **If you are using App Designer or creating apps with the `uifigure` function, then use `uialert` instead.** For more information, see “GUIDE Migration Strategies”.

Syntax

```
f = errordlg(msg)
f = errordlg(msg,title)
f = errordlg(msg,title,opts)
f = errordlg
```

Description

`f = errordlg(msg)` creates a nonmodal on page 1-3917 error dialog box with the specified error message and returns the dialog box figure object `f`. The message text wraps to fit the dialog box. The dialog box title is **Error Dialog**.

`f = errordlg(msg,title)` specifies a custom dialog box title.

`f = errordlg(msg,title,opts)` specifies the window style when `opts` is set to `'non-modal'`, `'modal'`, or `'replace'`. It specifies the window style and an interpreter for the `msg` argument when `opts` is a structure array.

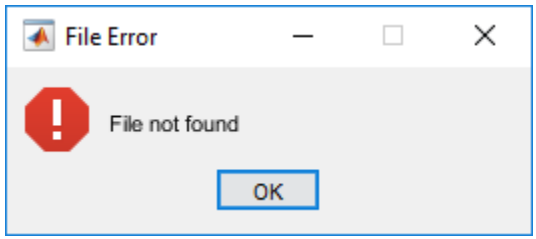
`f = errordlg` creates an error dialog box with a default title and message as follows.

- Default title: **Error Dialog**
- Default message: **This is the default error**

Examples

Specify Message and Title for Error Dialog Box

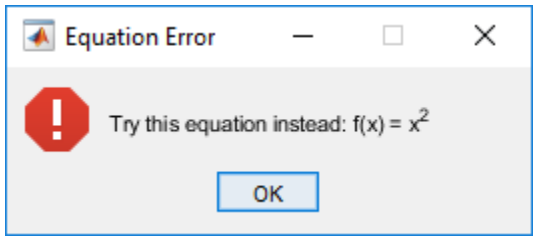
```
f = errordlg('File not found','File Error');
```



Interpret Message for Modal Error Dialog Box

Create a structure,`opts`, to specify a modal on page 1-3917 window style and the TeX interpreter. Then, create an error dialog box specifying the `opts` structure as an input argument. The TeX interpreter renders the `^2` characters in the message text as a superscript.

```
opts = struct('WindowStyle','modal',...
              'Interpreter','tex');
f = errordlg('Try this equation instead: f(x) = x^2',...
             'Equation Error', opts);
```



Input Arguments

`msg` — Error message
`'This is the default error'` (default) | character vector | cell array of character vectors | string array

Error message, specified as a character vector, cell array of character vectors, or a string array.

- If you specify the message as a character vector, then MATLAB wraps the text to fit the dialog box.
- If you specify the message a cell array, then MATLAB wraps the text after each cell array element. MATLAB wraps the text of long cell array elements to fit the dialog box.

Example: 'Input must be a scalar value.'

`title` — Dialog box title
`'Error Dialog'` (default) | character vector | string scalar

Dialog box title, specified as a character vector or string scalar.

Example: 'Input Error'

`opts` — Dialog box settings
'non-modal' | 'modal' | 'replace' | structure array

Dialog box settings specified as a window style only or a structure. The structure specifies the window style and an interpreter for the `msg` argument.

To specify the window style only, set `opts` to one of the values in this table.

Value	Description
`'non-modal'`	Create an error dialog box that is nonmodal on page 1-3917. This dialog box has no effect on other open dialog boxes.
`'modal'`	Specify an error dialog box that is modal on page 1-3917. If other error dialog boxes have the same title, then MATLAB modifies the most recently active one with the current specifications. MATLAB deletes all other open error, message, and warning dialog boxes with the same dialog box title as the most recently active error dialog box. The affected dialog boxes can be modal or nonmodal. Message and warning dialog boxes are created with the `msgbox`, and `warndlg` functions, respectively.

Value	Description
`'replace'`	Specify an error dialog box that is nonmodal. If other error dialog boxes have the same title, then MATLAB modifies the most recently active one with the current specifications. MATLAB deletes all other open error, message, or warning dialog boxes with the same title as the most recently active error dialog box. The affected dialog boxes can be modal or nonmodal. Message and warning dialog boxes are created with the `msgbox`, and `warndlg` functions, respectively.

To specify the window style and an interpreter for the error dialog box message, create an `opts` structure with the fields `WindowStyle` and `Interpreter`. This table shows valid values for the fields. The structure must include both fields.

Field	Values
`WindowStyle`	`'non-modal'`,`'modal'`, or `'replace'`.
`Interpreter`	`'none'` or `'tex'`. If set to `'tex'`, then MATLAB renders the message using the TeX interpreter. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the message text.

Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they modify only the next character or the characters within the curly braces. When you set the `Interpreter` property to `'tex'`, the supported modifiers are as follows.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`

Modifier	Description	Example
`\rm`	Normal font	`'\rm text'`
`\fontname{`*`specifier`*`}`	Font name — Replace *`specifier`* with the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{`*`specifier`*`}`	Font size —Replace *`specifier`* with a numeric scalar value in point units.	`'\fontsize{15} text'`
`\color{`*`specifier`*`}`	Font color — Replace *`specifier`* with one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Replace *`specifier`* with a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	β	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ϵ	`\Delta`	Δ	`\spadesuit`	♠

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\zeta	ζ	\Theta	Θ	\leftrightarrow	↔
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	µ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

Example: `opts = 'modal'`

Example: `opts.WindowStyle = 'non-modal'; opts.Interpreter = 'tex';`

More About

Modal Dialog Box

A modal dialog box prevents a user from interacting with other MATLAB windows before responding to the modal dialog box.

Nonmodal Dialog Box

A nonmodal dialog box enables a user to interact with other MATLAB windows before responding to the nonmodal dialog box. A nonmodal dialog box is also referred to as `normal`.

Tips

- MATLAB program execution continues even when a modal Error dialog box is active. To block program execution until the user closes the dialog box, use the `uiwait` function.

See Also

`dialog` | `helpdlg` | `msgbox` | `warndlg`

Introduced before R2006a

etime

Time elapsed between date vectors

Syntax

```
e = etime(t2,t1)
```

Description

`e = etime(t2,t1)` returns the number of seconds between two date vectors or matrices of date vectors, `t1` and `t2`.

Examples

Compute Elapsed Time

Compute the time elapsed between a specific time and the current time, to 0.01-second accuracy.

Define the initial date and time and convert to date vector form.

```
format shortg
str = 'March 28, 2012 11:51:00';
t1 = datevec(str,'mmmm dd, yyyy HH:MM:SS')
```

```
t1 = 1×6

        2012           3          28          11          51           0
```

Determine the current date and time.

```
t2 = clock
```

```
t2 = 1×6
```

```
        2019            8           27            2           24        7.297
```

The `clock` function returns the current date and time as a date vector.

Use `etime` to compute the number of seconds between `t1` and `t2`.

```
e = etime(t2,t1)
```

```
e =
   2.3394e+08
```

Input Arguments

`t2,t1` — Date vectors
1-by-6 vector | `m`-by-6 matrix

Date vectors, specified as 1-by-6 vectors or `m`-by-6 matrices containing `m` full date vectors in the format:`[Year Month Day Hour Minute Second]`.

Example: `[2012 03 27 11 50 01]`

Data Types: `double`

Tips

- To time the duration of an event, use the `timeit` or `tic` and `toc` functions instead of `clock` and `etime`. The `clock` function is based on the system time, which can be adjusted periodically by the operating system, and thus might not be reliable in time comparison operations.

Algorithms

`etime` does not account for the following:

- Leap seconds.
- Daylight savings time adjustments.

- Differences in time zones.

See Also

`clock` | `cputime` | `now` | `tic` | `timeit` | `toc`

Introduced before R2006a

etree

Elimination tree

Syntax

```
p = etree(A)
p = etree(A,'col')
p = etree(A,'sym')
[p,q] = etree(...)
```

Description

`p = etree(A)` returns an elimination tree for the square symmetric matrix whose upper triangle is that of `A`. `p(j)` is the parent of column `j` in the tree, or `0` if `j` is a root.

`p = etree(A,'col')` returns the elimination tree of `A'*A`.

`p = etree(A,'sym')` is the same as `p = etree(A)`.

`[p,q] = etree(...)` also returns a postorder permutation `q` of the tree.

See Also

`etreeplot` | `treelayout` | `treeplot`

Introduced before R2006a

etreeplot

Plot elimination tree

Syntax

```
etreeplot(A)
etreeplot(A,nodeSpec,edgeSpec)
```

Description

`etreeplot(A)` plots the elimination tree of `A` (or `A+A'`, if non-symmetric).

`etreeplot(A,nodeSpec,edgeSpec)` allows optional parameters `nodeSpec` and `edgeSpec` to set the node or edge color, marker, and linestyle. Use `''` to omit one or both.

See Also

`etree` | `treelayout` | `treeplot`

Introduced before R2006a

eval

Execute MATLAB expression in text

Syntax

```
eval(expression)
[output1,...,outputN] = eval(expression)
```

Description

`eval(expression)` evaluates the MATLAB code represented by `expression`. If you use `eval` within an anonymous function, nested function, or function that contains a nested function, the evaluated `expression` cannot create a variable.

`[output1,...,outputN] = eval(expression)` stores output from `expression` in the specified variables.

Input Arguments

`expression`

Character vector or string scalar that contains a valid MATLAB expression.

To include a numeric value in the expression, convert it to a character vector or string scalar.

Output Arguments

`output1,...,outputN`

Outputs from the evaluated `expression`.

Examples

Variable Name Evaluation

Select a matrix to plot at runtime.

This example requires that you have a matrix in the current workspace. For example:

```
aMatrix = magic(5);
```

Interactively request the name of a matrix to plot, and call `eval` to use its value.

```
expression = input('Enter the name of a matrix: ','s');
if (exist(expression,'var'))
    mesh(eval(expression))
end
```

If you type `aMatrix` at the input prompt, this code creates a mesh plot of `magic(5)`.

Tips

- Many common uses of the `eval` function are less efficient and are more difficult to read and debug than other MATLAB functions and language constructs. For more information, see “Alternatives to the eval Function”.
- Whenever possible, do not include output arguments within the input to the `eval` function, such as `eval(['output = ',expression])`. The preferred syntax,

  ```
  output = eval(expression)
  ```

  allows the MATLAB parser to perform stricter checks on your code, preventing untrapped errors and other unexpected behavior.

See Also

`assignin` | `evalc` | `evalin` | `feval` | `try`

Topics

“Alternatives to the eval Function”

“Variables in Nested and Anonymous Functions”

Introduced before R2006a

evalc

Evaluate MATLAB expression with capture

Syntax

```
T = evalc(expression)
[T,output1,...,outputN] = evalc(expression)
```

Description

`T = evalc(expression)` is the same as `eval(expression)` except that anything that would normally be written to the command window, except for error messages, is captured and returned in the character array `T` (lines in `T` are separated by `\n` characters).

`[T,output1,...,outputN] = evalc(expression)` is the same as `[output1,...,outputN] = eval(expression)` except that any output is captured into `T`.

Input Arguments

`expression`

Character vector or string scalar that contains a valid MATLAB expression.

To include a numeric value in the expression, convert it to a character vector or string scalar.

Output Arguments

T

Output normally written to the command window during the evaluation of `expression`, except for error messages, returned in a character array. The lines in `T` are separated by `\n` characters.

`output1,...,outputN`

Outputs from the evaluated `expression`.

Tips

When you are using `evalc`, functions `diary`, `more`, and `input` are disabled.

See Also

`assignin` | `diary` | `eval` | `evalin` | `feval` | `input` | `more`

Introduced before R2006a

evalin

Execute MATLAB expression in specified workspace

Syntax

```
evalin(ws, expression)
[a1, a2, a3, ...] = evalin(ws, expression)
```

Description

`evalin(ws, expression)` executes *`expression`*, a character vector or string scalar containing any valid MATLAB expression using variables in the workspace `ws`. `ws` can have a value of `'base'` or `'caller'` to denote the MATLAB base workspace or the workspace of the caller function.

`[a1, a2, a3, ...] = evalin(ws, expression)` executes *`expression`* and returns the results in the specified output variables. Using the `evalin` output argument list is recommended over including the output arguments in the expression:

```
evalin(ws,'[a1, a2, a3, ...] = function(var)')
```

The above syntax avoids strict checking by the MATLAB parser and can produce untrapped errors and other unexpected behavior.

Examples

This example extracts the value of the variable `var` in the MATLAB base workspace and captures the value in the local variable `v`:

```
v = evalin('base', 'var');
```

Limitation

`evalin` cannot be used recursively to evaluate an expression. For example, a sequence of the form `evalin('caller', 'evalin(''caller'', ''x'')')` doesn't work.

Tips

The MATLAB base workspace is the workspace that is seen from the MATLAB command line (when not in the debugger). The caller workspace is the workspace of the function that called the currently running function. Note that the base and caller workspaces are equivalent in the context of a function that is invoked from the MATLAB command line.

`evalin('caller', expression)` finds only *variables* in the caller's workspace; it does not find *functions* in the caller. For this reason, you cannot use `evalin` to construct a handle to a function that is defined in the caller.

If you use `evalin('caller', expression)` in the MATLAB debugger after having changed your local workspace context with `dbup` or `dbdown`, MATLAB evaluates the expression in the context of the function that is one level up in the stack from your current workspace context.

See Also

`assignin` | `eval` | `evalc` | `feval` | `try`

Introduced before R2006a

event.DynamicPropertyEvent class

Package: `event`
Superclasses: `event.EventData`

Event data for dynamic property events

Description

The `event.DynamicPropertyEvent` class defines the event data passed to listeners of the `dynamicprops` `PropertyAdded` and `PropertyRemoved` events.

`event.DynamicPropertyEvent` is a subclass of `event.EventData`. The `event.DynamicPropertyEvent` class is sealed (cannot be subclassed) and its constructor is private.

Properties

`PropertyName` — Name of dynamic property
property name

Name of dynamic property that is added or removed

Attributes:

`GetAccess`	`public`
`SetAccess`	`private`
`GetObservable`	`true`
`SetObservable`	`true`

Data Types: `char`

`Source` — Event source
object

Object that is the source of the event

Attributes:

GetAccess	public
SetAccess	private
GetObservable	true
SetObservable	true

EventName — Name of event
PropertyAdded | PropertyRemoved

Name of event triggered on the source object

Attributes:

GetAccess	public
SetAccess	private
GetObservable	true
SetObservable	true

Data Types: char

Events

ObjectBeingDestroyed	Triggered when object is destroyed.

Attributes

Sealed	true
HandleCompatible	true

For information on class attributes, see “Class Attributes”.

See Also

event.EventData

Topics

"Dynamic Property Events"
Class Attributes
Property Attributes

Introduced in R2016a

event.EventData class

Package: `event`
Superclasses: `handle`

Base class for event data

Description

The `event.EventData` class is the base class for all data objects passed to listeners. When you trigger an event using the `notify` handle class method, MATLAB assigns values to the properties of an `event.EventData` object and passes that object to the listener callback function (the event handler).

To provide additional information to event listeners, subclass `event.EventData`. For more information on custom event data, see “Define Event-Specific Data”.

Note Subclasses of `event.EventData` must set the class `ConstructOnLoad` attribute to `true`.

The `event.EventData` class is a `handle` class.

Class Attributes

`ConstructOnLoad`	`true`
`HandleCompatible`	`true`

For information on class attributes, see “Class Attributes”.

Creation

The `notify` handle class method creates an `event.EventData` object when called to trigger an event. The `event.EventData` constructor accepts no input arguments so subclasses of `event.EventData` cannot pass arguments to the superclass constructor.

Properties

Source — Event source

object

Event source object, specified as a handle to the object that triggered the event.

Attributes:

GetAccess	public
SetAccess	private
GetObservable	true
SetObservable	true

EventName — Name of event

character vector

Name of the event, specified as a character vector.

Attributes:

GetAccess	public
SetAccess	private
GetObservable	true
SetObservable	true

Data Types: `char`

Examples

Access Event Data

Get the event source object handle and the event name from the `event.EventData` object passed to this callback function when the event is triggered.

```
function myCallbk(s,evtData)
   eventSource = evtData.Source;
   eventName = evtData.EventName;
```

```
    ...
end
```

See Also

`event.DynamicPropertyEvent` | `event.listener`

Topics

"Listener Callback Syntax"
"Define Event-Specific Data"

Introduced in R2008a

event.hasListener

Determine if listeners exist for event

Syntax

```
tf = event.hasListener(src,EventName)
```

Description

`tf = event.hasListener(src,EventName)` returns `true` if listeners exist for the specified event on the object `src`. Otherwise, it returns `false`. When `src` is an array, `event.hasListener` returns a logical array the same size as `src`.

The class of `src` must define or inherit the specified event. If `src` is a heterogeneous array, the specified event must exist on the class of the array. `event.hasListener` cannot find listeners for events that are defined by some, but not all objects in the heterogeneous array.

`event.hasListener` respects the value of the `event.listener` object `Enabled` property. If you set the listener `Enabled` property to `false`, `event.hasListener` returns `false` for that listener.

Examples

Determine If There Are Listeners

Determine if there are there listeners for `MyEvent` on object `src`:

```
tf = event.hasListener(src,'MyEvent');
```

If `tf` is `true`, then one or more listeners are attached to `src` for event `MyEvent`.

Input Arguments

`src` — Event source
object array

Event source, specified as a scalar or array of objects. The event source is the object to which the listener is attached.

`EventName` — Event name
`char` array

Event name, specified as a `char` array.

Output Arguments

`tf` — `true` if listeners exist
`logical` array

`true` if listeners exist for the specified event on the specified objects. `false` if listeners do not exist.

Data Types: `logical`

Limitations

- `event.hasListener` must have `NotifyAccess` for the event in question.
- `event.hasListener` does not work with the predefined property events, `PreSet`, `PostSet`, `PreGet`, `PostGet`.
- `event.hasListener` does not work with the predefined dynamic property events `PropertyAdded` and `PropertyRemoved`.
- Use `event.hasListener` only with user-defined events.

See Also

`addlistener` | `notify`

Topics

"Determine If Event Has Listeners"
"Event Attributes"

Introduced in R2016a

event.listener class

Package: `event`

Class defining listener objects

Description

The `event.listener` class defines listener objects. Listener objects respond to a specific event by executing a callback function when the event is triggered. You can use the `event.listener` class to construct a listener object. Also, you can create listeners using the handle `addlistener` and `listener` methods.

Use the handle `notify` method to trigger an event.

The `event.listener` class is a `handle` class.

Class Attributes

`ConstructOnLoad`	`true`
`HandleCompatible`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Description

`eListener = event.listener(eventSource,eventName,callbackFcn)` creates a listener for the specified event name on the specified source objects and identifies a function handle to the callback function.

If `eventSource` is an array of object handles, the listener responds to the named event on any of the objects in the array.

Input Arguments

`eventSource` — Event source
handle object | cell array of handle objects

Event source, specified as a handle object array or a cell array of object handles. Use a cell array when the source objects cannot form an array because their classes differ. All source objects must define the specified event.

`eventName` — Event name
character vector | string scalar

Event name, specified as the literal name of the event.

`callbackFcn` — Callback function
function handle

Callback function, specified by a function handle. For more information, see "Listener Callback Syntax"

Properties

`Source` — Event source objects
handle object array or a cell array of handle objects.

Event source objects, specified as the handles of the objects that this listener responds to when the event is triggered.

Attributes:

```
GetAccess                                          public
SetAccess                                          public
GetObservable                                      true
SetObservable                                      true
```

Data Types: `handle object` | `cell array`

`EventName` — Name of event
character vector | string scalar

Name of the event that the listener responds to when triggered on the specified source objects.

Attributes:

GetAccess	public
SetAccess	public
GetObservable	true
SetObservable	true

Data Types: `char` | `string`

`Callback` — Event callback
function handle

Event callback, specified as a function handle. The function executes when the event is triggered.

Attributes:

GetAccess	public
SetAccess	public
GetObservable	true
SetObservable	true

Data Types: `function_handle`

`Enabled` — Enable or disable listener
true (default) | false

If `Enabled` is set to `true` (the default), the callback executes when the event occurs. To disable callback execution for this listener, set `Enabled` to `false`.

Attributes:

GetAccess	public
SetAccess	public
GetObservable	true
SetObservable	true

Data Types: `logical`

`Recursive` — Execute callback recursively
false (default) | true

When `false` (the default), the listener does not execute its callback recursively. Therefore, if the callback triggers its own event, the listener does not respond again.

When `true`, the listener callback can cause the same event that triggered the callback. This scheme can lead to infinite recursion, which ends when the MATLAB recursion limit eventually throws an error.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`
`GetObservable`	`true`
`SetObservable`	`true`

Data Types: `logical`

Examples

Define Listener

Define a listener for an event named `EOL` with a callback function named `EOLCallback` that is triggered on an object in the array `textReader`.

```
listenerHandle = event.listener(textReader,"EOL",@EOLCallback);
```

More About

Listener Callback Signature

The listener callback function must accept at least two input arguments.

```
function CallbackFunction(source,eventData)
   ...
end
```

- `source` is the object that is the source of the event.
- `eventData` is an `event.EventData` object or an instance of a subclass of `event.EventData`.

For more information about listener callbacks, see “Listener Callback Syntax”, “Callback Execution”, and “Define Custom Event Data”.

Limiting Listener Lifecycle

You can create listener object using the `event.listener` class constructor, or using the handle class `addlistener` or `listener` method.

When you create a listener using `addlistener`, the event source object holds a reference to the listener. When the source is destroyed, MATLAB also destroys the listener. You do not need to store a reference to the listener object to manage its lifecycle.

When you create a listener using `event.listener` or the `listener` method, the listener's lifecycle is not coupled to the event source. Because the event source object does not hold a reference to the listener, you have more control over the listener lifecycle. However, if the listener object goes out of scope, the listener no longer exists.

For more information on listener lifecycle, see “Listener Lifecycle”.

Removing a Listener

If you call `delete(lh)` on the listener object, the listener ceases to exist, which means the event no longer causes the listener callback function to execute.

Disabling a Listener

You can enable or disable a listener by setting the value of the listener `Enabled` property.

See Also

`event.proplistener` | `handle.addlistener` | `handle.delete` | `handle.listener`

Topics

“Restore Listeners”
“Events”

Introduced in R2008a

event.PropertyEvent class

Package: `event`

Data for property events

Description

The `event.PropertyEvent` class defines the event data objects passed to listeners of these predefined property events:

- `PreGet`
- `PostGet`
- `PreSet`
- `PostSet`

Predefined property events enable listeners to respond to changes made to property values. For more information, see “Listen for Changes to Property Values”.

The `event.PropertyEvent` class is a sealed subclass of `event.EventData` (that is, you cannot subclass `event.PropertyEvent`). The class constructor is private. MATLAB creates an `event.PropertyEvent` object to pass to listeners of property events.

The `event.PropertyEvent` class is a `handle` class.

Class Attributes

```
Sealed                                                    true
ConstructOnLoad                                           true
HandleCompatible                                          true
RestrictsSubclassing                                      true
```

For information on class attributes, see “Class Attributes”.

Properties

`AffectedObject` — Object whose property is affected

handle object

Object whose property is affected, specified as the object handle.

Attributes:

GetAccess	public
SetAccess	private
GetObservable	true
SetObservable	true

Data Types: `handle object`

`Source` — Property that triggers the event

`meta.property`

Property that triggers the event, specified as the `meta.property` object for the property.

Attributes:

GetAccess	public
SetAccess	private
GetObservable	true
SetObservable	true

Data Types: `meta.property`

`EventName` — Name of property event

`PreGet` | `PostGet` | `PreSet` | `PostSet`

Name of the property event, specified as one of the four event names.

Attributes:

GetAccess	public
SetAccess	private
GetObservable	true
SetObservable	true

Data Types: `char`

Examples

Listen for Property Event

Define the `propEventClass` class with the `PropOne` `GetObservable` and `SetObservable` attributes enabling observation of property events. The class constructor adds listeners for these events.

```
classdef propEventClass < handle
   % Class to observe property events
   properties (GetObservable,SetObservable)
      PropOne string = "default"
   end
   methods
      function obj = propEventClass
         addlistener(obj,'PropOne','PreGet',@propEventHandler);
         addlistener(obj,'PropOne','PostSet',@propEventHandler);
      end
   end
end
```

The `propEventHandler` function serves as the callback for the `PreGet` and `PostSet` events.

The `event.PropertyEvent` object `Source` property contains the `meta.property` object for `PropOne`. Access the `meta.property` `Name` property to get the name of the property on which the event is triggered. Switch on the property name when the callback handles multiple properties.

The `event.PropertyEvent` object `EventName` property contains the name of the event. To handle multiple property events from the callback, switch on the event name.

```
function propEventHandler(~,eventData)
   switch eventData.Source.Name % Get property name
      case 'Prop1'
         switch eventData.EventName % Get the event name
            case 'PreGet'
               fprintf('%s\n','***PreGet triggered***')
            case 'PostSet'
               fprintf('%s\n','***PostSet triggered***')
               disp(eventData.AffectedObject.(eventData.Source.Name));
         end
```

```
    end
end
```

Referencing the `PropOne` property value results in a response from the `propEventHandler` to the `PreGet` event.

```
obj = propEventClass;
obj.PropOne
```

```
***PreGet triggered***

ans =

    "default"
```

Assigning to the `PropOne` property results in a response from the `propEventHandler` to the `PostSet` event.

Because the callback gets the property value to display the new value after the `PostSet` event, the `PreGet` event is triggered. Also, because the assignment statement is not terminated by a semicolon, MATLAB gets the property value to display the object in the command window, which triggers the `PreGet` event again.

```
obj.PropOne = "New string"
```

```
***PostSet triggered***
***PreGet triggered***
New string

obj =

***PreGet triggered***
  propEventClass with properties:

    PropOne: "New string"
```

See Also

`event.EventData` | `meta.property`

Topics

“Listen for Changes to Property Values”

Introduced in R2008a

event.proplistener

Define listener object for property events

Syntax

```
lh =
event.proplistener(Hobj,Properties,'PropEvent',@CallbackFunction)
```

Description

`lh = event.proplistener(Hobj,Properties,'PropEvent',@CallbackFunction)` creates a property listener object for one or more properties on the specified object.

- `Hobj` — handle of object whose property or properties are to be listened to. If `Hobj` is an array, the listener responds to the named event on all objects in the array.
- `Properties` — an object array or a cell array of `meta.property` object handles representing the properties to which you want to listen.
- `PropEvent` — must be one of the following: `PreSet`, `PostSet`, `PreGet`, `PostGet`
- `@CallbackFunction` — function handle to the callback function that executes when the event occurs.

The `event.proplistener` class defines property event listener objects. It is a subclass of the `event.listener` class and adds one property to those defined by `event.listener`:

- `Object` — Cell array of objects whose property events are being listened to.

You can call the `event.proplistener` constructor instead of calling `addlistener` to create a property listener. However, when you do not use `addlistener`, the listener's lifecycle is not tied to the object(s) being listened to.

The `event.proplistener` class is a `handle` class. The `event.proplistener` and the `event.listener` classes are part of the same heterogeneous hierarchy. Therefore, you

can create arrays that contain objects of both classes. The class of an array containing both classes of objects is `event.listener`.

See “Listen for Changes to Property Values”.

See “Get Information About Properties” for more information on using `meta.property` objects.

See Also

`event.listener` | `handle.addlistener` | `handle.listener`

Topics

“Restore Listeners”

Introduced in R2008a

eventlisteners

List event handler functions associated with COM object events

Syntax

```
info = eventlisteners(c)
```

Description

`info = eventlisteners(c)` lists the events and their event handler routines registered with a COM object.

Input Arguments

`c` — COM object
function handle

COM object, specified as a function handle.

Output Arguments

`info` — List of event handler functions
cell array of character vectors | empty cell array

List of event handler functions, returned as a cell array of character vectors. Each row contains the name of a registered event and the handler routine for that event. If the object has no registered events, `eventlisteners` returns an empty cell array.

Limitations

- COM functions are available on Microsoft Windows systems only.

See Also

`events (COM)` | `isevent` | `registerevent` | `unregisterallevents` | `unregisterevent`

Introduced before R2006a

events

Event names

Syntax

```
events(ClassName)
events(obj)
e = events( ___ )
```

Description

`events(ClassName)` displays the names of the non-hidden, public events for the MATLAB class *`classname`*, including events inherited from superclasses.

`events(obj)` displays the names of the public events for the class of `obj`, where `obj` is an instance of a MATLAB class. `obj` can be a scalar object or an array of objects.

`e = events( ___ )` returns the event names in a cell array.

Examples

Get Event Names from Class Name

Get the names of the public events of the handle class and store the result in a cell array of character vectors.

```
eventNames = events('handle');
```

List Object Events

List the events defined by the containers.Map class from an instance of this class.

```
m = containers.Map('May',70);
events(m)
```

```
Events for class containers.Map:

    ObjectBeingDestroyed
```

Input Arguments

`ClassName` — Class name
character vector | string scalar

Class name, specified as a character vector or string.

Data Types: `char` | `string`

`obj` — Object
MATLAB object

MATLAB object specified as a scalar object or an object array.

Output Arguments

`e` — Event names
cell array

Event names contained in a cell array of character vectors.

More About

Public Events

An event is public when the value of its `ListenAccess` attribute is `public` and its `Hidden` attribute value is `false` (default values for both attributes). See “Event Attributes” for a complete list of attributes.

Events Keyword

`events` is also a MATLAB class-definition keyword. See `classdef` for more information on class definition keywords.

See Also

`classdef` | `methods` | `properties`

Topics

"Events"

Introduced in R2008a

exceltime

Convert MATLAB datetime to Excel date number

Syntax

```
e = exceltime(t)
e = exceltime(t,dateType)
```

Description

`e = exceltime(t)` returns a `double` array containing Excel serial date numbers equivalent to the datetime values in `t`. Excel serial date numbers are the number of days and fractional days since 0-January-1900 00:00:00, and do not take into account time zone and leap seconds.

`e = exceltime(t,dateType)` returns the type of Excel serial date numbers specified by `dateType`. For example, you can convert datetime values to the number of days since 1-January-1904 00:00:00.

Examples

Convert Datetime Array to Excel Date Numbers

Create a `datetime` array. Then, convert the dates to the equivalent Excel® serial date numbers.

```
t = datetime('now') + calmonths(1:3)
```

```
t = 1x3 datetime array
   26-Sep-2019 23:35:09   26-Oct-2019 23:35:09   26-Nov-2019 23:35:09
```

```
e = exceltime(t)
```

```
e = 1×3
10^4 ×
```

```
    4.3735    4.3765    4.3796
```

Input Arguments

t — Input date and time
`datetime` array

Input date and time, specified as a `datetime` array.

dateType — Type of Excel serial date numbers
`'1900'` (default) | `'1904'`

Type of Excel serial date numbers, specified as either `'1900'` or `'1904'`.

- If `dateType` is `'1900'`, then `exceltime` converts the datetime values in `t` to the equivalent the number of days and fractional days since 0-January-1900 00:00:00.
- If `dateType` is `'1904'`, then `exceltime` converts the datetime values in `t` to the equivalent the number of days and fractional days since 1-January-1904 00:00:00.

`exceltime` does not account for time zone.

Output Arguments

e — Excel serial date numbers
scalar | vector | matrix | multidimensional array

Excel serial date numbers, returned as a scalar, vector, matrix, or multidimensional array of type `double`. Excel serial date numbers are not defined prior to their epoch (0-January-1900 or 1-January-1904). Excel serial date numbers treat 1900 as a leap year. Therefore, dates after February 28, 1900 are offset by one day relative to MATLAB serial date numbers, and there is a discontinuity of one day between February 28, 1900 and March 1, 1900.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`datenum` | `datetime` | `juliandate` | `posixtime` | `yyyymmdd`

Introduced in R2014b

Execute

Execute MATLAB command in Automation server

Syntax

IDL Method Signature

```
BSTR Execute([in] BSTR command)
```

Microsoft Visual Basic Client

```
Execute(command As String) As String
```

MATLAB Client

```
result = Execute(h,'command')
```

Description

The `Execute` function executes the MATLAB statement specified by `command` in the MATLAB Automation server attached to `h`. To call a function with arguments, use `Feval`.

The server returns output from the command and any MATLAB warning or error messages in `result`. If you terminate the MATLAB command with a semicolon and there are no warnings or error messages, `result` might be empty.

COM functions are available on Microsoft Windows systems only.

Examples

Modify Matrix in MATLAB® Workspace from Visual Basic® .NET Client

This example shows how to execute a command on a MATLAB matrix and return the result to the Visual Basic® .NET client. The example creates a matrix in the client and modifies the contents in the MATLAB workspace.

```
type comexecute.vb
```

```
Dim Matlab As Object
Dim data(6) As Double
Dim B As Object
B = Nothing
Matlab = CreateObject("matlab.application")
For i = 0 To 6
    data(i) = i * 15
Next i
Matlab.PutWorkspaceData("A", "base", data)
Matlab.Execute("A = A.*2;")
Matlab.GetWorkspaceData("A", "base", B)
MsgBox("Doubled second value of A = " & B(0, 1))
```

Modify Matrix in MATLAB® Workspace from VBA Client

This example shows how to execute a command on a MATLAB matrix and return the result to the VBA client. The example creates a matrix in the client and modifies the contents in the MATLAB workspace.

```
type comexecute.vba
```

```
Dim Matlab As Object
Dim MATLAB_version As String
Dim data(6) As Double
Set Matlab = CreateObject("matlab.application")
For i = 0 To 6
    data(i) = i * 15
Next i
x = Matlab.PutWorkspaceData("A", "base", data)
Matlab.Execute ("A = A.*2;")
y = Matlab.GetWorkspaceData("A", "base", B)
MsgBox ("Doubled second value of A = " & B(0, 1))
```

Tips

- To display output from `Execute` in the client window, specify an output variable, `result`.
- If there is an error, the `Execute` function returns the MATLAB error message with the characters `???` prepended to the text.

See Also

`Feval` | `GetFullMatrix` | `PutFullMatrix`

Topics

"Conversion of COM Types to MATLAB Types"
"Conversion of MATLAB Types to COM Types"

Introduced before R2006a

exist

Check existence of variable, script, function, folder, or class

Syntax

```
exist name
exist name searchType
A = exist( ___ )
```

Description

`exist name` returns the type of `name` as a number. This list describes the type associated with each value:

- 0 — `name` does not exist or cannot be found for other reasons. For example, if `name` exists in a restricted folder to which MATLAB does not have access, `exist` returns 0.
- 1 — `name` is a variable in the workspace.
- 2 — `name` is a file with extension `.m`, `.mlx`, or `.mlapp`, or `name` is the name of a file with a non-registered file extension (`.mat`, `.fig`, `.txt`).
- 3 — `name` is a MEX-file on your MATLAB search path.
- 4 — `name` is a loaded Simulink model or a Simulink model or library file on your MATLAB search path.
- 5 — `name` is a built-in MATLAB function. This does not include classes.
- 6 — `name` is a P-code file on your MATLAB search path.
- 7 — `name` is a folder.
- 8 — `name` is a class. (`exist` returns 0 for Java classes if you start MATLAB with the `-nojvm` option.)

MATLAB searches starting at the top of the search path, and moving down until a result is found or the last folder on the path is reached. If more than one `name` exists in a folder, MATLAB displays the first instance of `name`, according to the “Function Precedence Order”. Folders are an exception to the function precedence rules. They have precedence over all types except for variables and built-in functions.

For example, if `name` matches both a file with a `.m` extension and a P-code file, then `exist` returns 6, identifying it as a P-code file. If `name` matches both a variable and a P-code file, exists returns 1, identifying it as a variable. If `name` matches both a folder and a MATLAB function, `exist` returns `7`, identifying it as a folder.

`exist name searchType` returns the type of `name`, restricting results to the specified type, `searchType`. If `name` of type `searchType` does not exist, MATLAB returns `0`.

`A = exist( ___ )` returns the type of `name` to `A`.

Examples

Check Existence of Workspace Variable

Create a variable named `testresults`, and then confirm its existence in the workspace.

```
testresults = magic(5);
exist testresults
```

```
ans = 1
```

A variable named `testresults` exists in the workspace.

Check Existence of Folder

Create the folder `myfolder`, and then check its existence as a folder.

```
mkdir myfolder;
exist myfolder dir
```

```
ans = 7
```

If you specify the type as `file`, MATLAB® searches for both files and folders, therefore returning the same result.

```
exist myfolder file
```

```
ans = 7
```

Check if MATLAB Function is Built-In Function

Check whether the `plot` function is a built-in function or a file.

```
A = exist('plot')
```

```
A = 5
```

This indicates that `plot` is a built-in MATLAB function.

Input Arguments

`name` — name of variable, script, function, folder, or class
character vector | string scalar

Name of variable, script, function, folder, or class, specified as a character vector or string scalar.

`name` can include a partial path, but must be one of these:

- A folder on the search path
- In a folder on the search path
- The current folder
- In the current folder

Subfolders of folders on the path are not searched.

Otherwise, `name` must include a full path.

If `name` specifies a file with a non-registered file extension (`.mat`, `.fig`, `.txt`), include the extension. You can also include an extension to prevent conflict with other similar file names. For example, `exist file.txt` or `exist("file.txt")`.

Note MATLAB does not examine the contents or internal structure of a file and relies solely on the file extension for classification.

Data Types: `char` | `string`

searchType — Type of results to search for
`builtin` | `class` | `dir` | `file` | `var`

Type of results to search for, specified as one of these values:

searchType	Description	Possible Return Values
`builtin`	Checks only for built-in functions.	5, 0
`class`	Checks only for classes.	8, 0
`dir`	Checks only for folders.	7, 0
`file`	Checks only for files or folders.	2, 3, 4, 6, 7, 0
`var`	Checks only for variables.	1, 0

Alternative Functionality

- To check the existence of a file or folder, you also can use the `isfolder` or `isfile` functions. `exist` searches for files and folders on the search path, which can lead to unexpected results. `isfolder` and `isfile` search for files or folders only on the specified path or in the current folder, which can lead to clearer and faster results.

See Also

`dir` | `inmem` | `is*` | `isfile` | `isfolder` | `what` | `which` | `who`

Topics

"Function Precedence Order"
"What Is the MATLAB Search Path?"

Introduced before R2006a

exit

Terminate MATLAB program (same as quit)

Syntax

```
exit
```

Description

`exit` terminates the current session of MATLAB. This function is equivalent to the `quit` function and takes the same options. For more information, see `quit`..

See Also

`quit`

Introduced before R2006a

exp

Exponential

Syntax

```
Y = exp(X)
```

Description

`Y = exp(X)` returns the exponential e^x for each element in array `X`. For complex elements $z = x + iy$, it returns the complex exponential

$$e^z = e^x(\cos y + i\sin y) \ .$$

Use `expm` to compute a matrix exponential.

Examples

Numeric Representation of e

Calculate the exponential of 1, which is Euler's number, e.

```
exp(1)
```

```
ans = 2.7183
```

Euler's Identity

Euler's identity is the equality $e^{i\pi} + 1 = 0$.

Compute the value of $e^{i\pi}$.

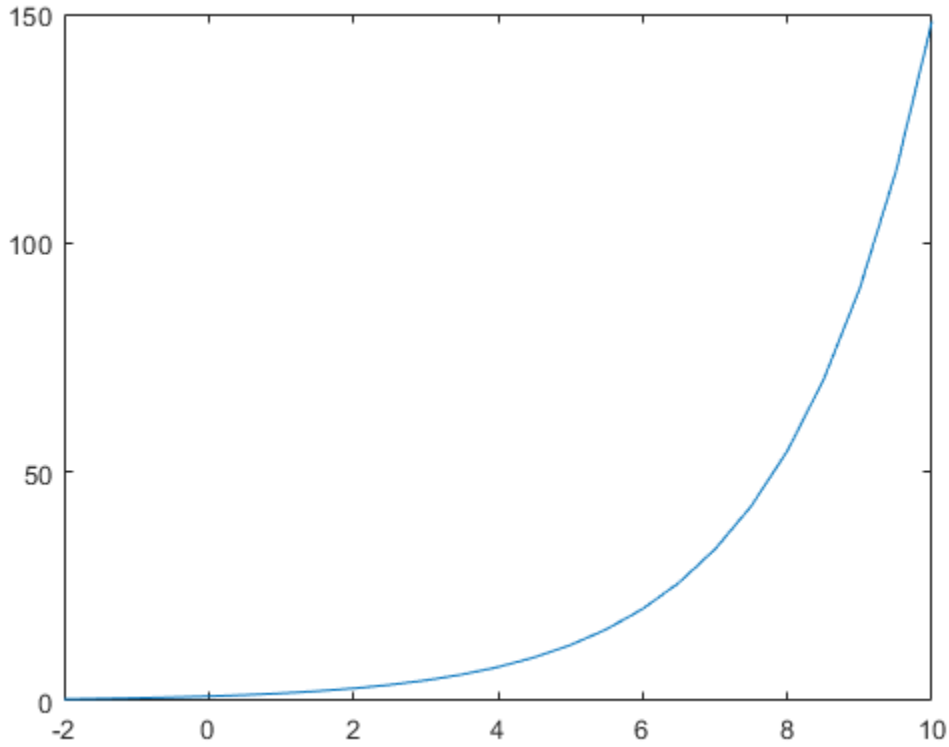
```
Y = exp(1i*pi)
```

```
Y = -1.0000 + 0.0000i
```

Plot Exponential Function

Plot $y = e^{x/2}$ for x values in the range $[-2, 10]$.

```
X = -2:0.5:10;
Y = exp(X/2);
plot(X,Y)
```



Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Exponential values
scalar | vector | matrix | multidimensional array

Exponential values, returned as a scalar, vector, matrix, or multidimensional array.

For real values of `X` in the interval (`-Inf`, `Inf`), `Y` is in the interval (`0`,`Inf`). For complex values of `X`, `Y` is complex. The data type of `Y` is the same as that of `X`.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`expint` | `expm` | `expm1` | `log` | `log10` | `mpower` | `power`

Introduced before R2006a

expint

Exponential integral

Syntax

```
Y = expint(X)
```

Description

`Y = expint(X)` evaluates the exponential integral for each element of `X`.

More About

Exponential Integral

The exponential integral computed by this function is defined as

$$E_1(x) = \int_x^\infty e^{-t}/t \, dt$$

Another common definition of the exponential integral function is the Cauchy principal value integral

$$\mathrm{Ei}(x) = \int_{-\infty}^{x} e^{t}/t \, dt$$

which, for real positive x, is related to `expint` as

$$E_1(-x) = -\mathrm{Ei}(x) - i\pi$$

References

[1] Abramowitz, M. and I. A. Stegun. *Handbook of Mathematical Functions*. Chapter 5, New York: Dover Publications, 1965.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

Introduced before R2006a

expm

Matrix exponential

Syntax

```
Y = expm(X)
```

Description

`Y = expm(X)` computes the matrix exponential of `X`. Although it is not computed this way, if `X` has a full set of eigenvectors `V` with corresponding eigenvalues `D`, then `[V,D] = eig(X)` and

```
expm(X) = V*diag(exp(diag(D)))/V
```

Use `exp` for the element-by-element exponential.

Examples

Compare Exponentials

Compute and compare the exponential of `A` with the *matrix* exponential of `A`.

```
A = [1 1 0; 0 0 2; 0 0 -1];
exp(A)
```

```
ans = 3×3

    2.7183    2.7183    1.0000
    1.0000    1.0000    7.3891
    1.0000    1.0000    0.3679
```

```
expm(A)
```

```
ans = 3×3

    2.7183    1.7183    1.0862
         0    1.0000    1.2642
         0         0    0.3679
```

Notice that the diagonal elements of the two results are equal, which is true for any triangular matrix. The off-diagonal elements, including those below the diagonal, are different.

Input Arguments

X — Input matrix
square matrix

Input matrix, specified as a square matrix.

Data Types: `single` | `double`
Complex Number Support: Yes

Algorithms

The algorithm `expm` uses is described in [1] and [2].

Note The files, `expmdemo1.m`, `expmdemo2.m`, and `expmdemo3.m` illustrate the use of Padé approximation, Taylor series approximation, and eigenvalues and eigenvectors, respectively, to compute the matrix exponential. References [3] and [4] describe and compare many algorithms for computing a matrix exponential.

References

[1] Higham, N. J., "The Scaling and Squaring Method for the Matrix Exponential Revisited," *SIAM J. Matrix Anal. Appl.*, 26(4) (2005), pp. 1179–1193.

[2] Al-Mohy, A. H. and N. J. Higham, "A new scaling and squaring algorithm for the matrix exponential," *SIAM J. Matrix Anal. Appl.*, 31(3) (2009), pp. 970–989.

[3] Golub, G. H. and C. F. Van Loan, *Matrix Computation*, p. 384, Johns Hopkins University Press, 1983.

[4] Moler, C. B. and C. F. Van Loan, “Nineteen Dubious Ways to Compute the Exponential of a Matrix,” *SIAM Review 20*, 1978, pp. 801–836. Reprinted and updated as “Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later,” *SIAM Review 45*, 2003, pp. 3–49.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`eig` | `exp` | `expm1` | `funm` | `logm` | `sqrtm`

Introduced before R2006a

expm1

Compute exp(x)-1 accurately for small values of x

Syntax

```
y = expm1(x)
```

Description

`y = expm1(x)` computes `exp(x)-1`, compensating for the roundoff in `exp(x)`.

For small `x`, `expm1(x)` is approximately `x`, whereas `exp(x)-1` can be zero.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`exp` | `expm` | `log1p`

Introduced before R2006a

export2wsdlg

Create dialog box for exporting variables to workspace

Syntax

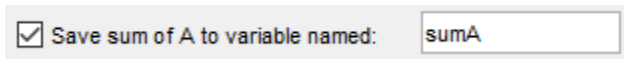
```
export2wsdlg(labels,vars,vals)
export2wsdlg(labels,vars,vals,title)
export2wsdlg(labels,vars,vals,title,defs)
export2wsdlg(labels,vars,vals,title,defs,helpfcn)
export2wsdlg(labels,vars,vals,title,defs,helpfcn,flist)
f = export2wsdlg( ___ )
[f,tf] = export2wsdlg( ___ )
```

Description

`export2wsdlg(labels,vars,vals)` creates a modal on page 1-3985 dialog box with a series of check boxes and edit fields. For each check box, there is a corresponding edit field. The arguments are:

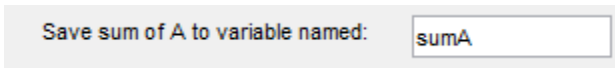
- `labels` – the labels for the check boxes.

  The number of labels determines how many check boxes and edit fields appear in the dialog box. One edit field appears after each check box label; for example:

- `vars` – the default variable names that appear in the edit fields, such as `sumA` shown in the preceding image.
- `vals` – the values to store in the variables.

The `labels`, `vars`, and `vals` must specify the same number of items. If `labels`, `vars`, and `vals` each specify one item, then the dialog box displays a label and an edit field only. For example:



`export2wsdlg(labels,vars,vals,title)` specifies a title for the dialog box.

`export2wsdlg(labels,vars,vals,title,defs)` specifies which check boxes are selected when the dialog box opens.

`export2wsdlg(labels,vars,vals,title,defs,helpfcn)` adds a **Help** button to the dialog box. The `helpfcn` is a callback that displays help.

`export2wsdlg(labels,vars,vals,title,defs,helpfcn,flist)` specifies a cell array of functions and optional arguments that calculate, and then return the values to export to `vars`. When you specify an `flist`, MATLAB uses it instead of `vals`. However, you must specify the `vals` argument as a syntax placeholder. The `flist` must be the same length as `labels`.

`f = export2wsdlg( ___ )` returns the `Figure` object in which the dialog box displays to `f`. You can request this output with any of the input argument combinations in the previous syntaxes.

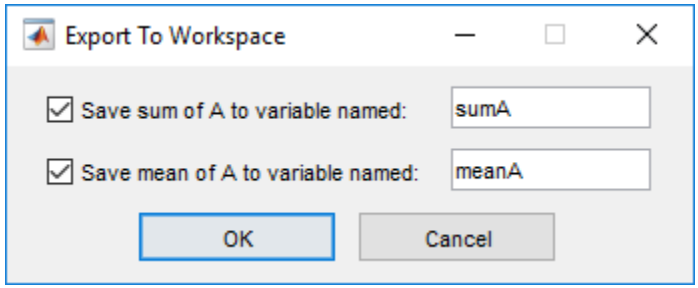
`[f,tf] = export2wsdlg( ___ )` returns `tf` as `1` (true) if the user clicks **OK**, or `0` if the user closes the dialog box by clicking **Cancel** or the close button (X) in the dialog box title bar. When `tf` is 0, `f` is returned as an empty array (`[]`). The `export2swdlg` function does not return until the user closes the dialog box.

Examples

Save Variables to Base Workspace

Create a dialog box that enables the user to save the variables `sumA`, or `meanA`, or both to the base workspace.

```
A = randn(10,1);
labels = {'Save sum of A to variable named:' ...
          'Save mean of A to variable named:'};
vars = {'sumA','meanA'};
values = {sum(A),mean(A)};
export2wsdlg(labels,vars,values);
```

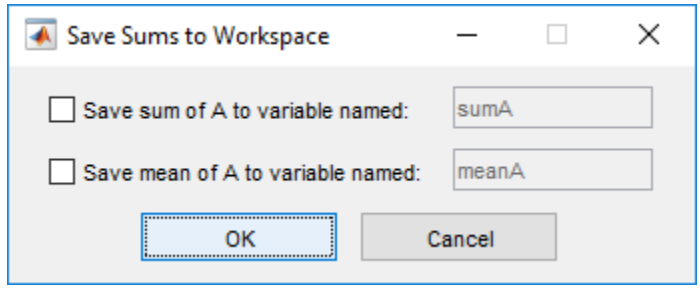


If the user enters an invalid variable name, such as 2 and clicks **OK**, then MATLAB automatically returns an error dialog box. After clicking **OK** in the error dialog box, the user has an opportunity to enter a valid variable name in the Export to Workspace dialog box.

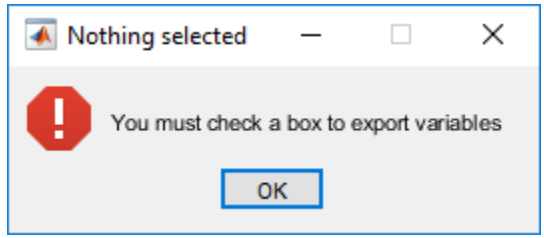
Specify Default Check Box Selections

Define the input variable, `defs`, to specify that no check box is selected when the dialog box opens.

```
A = randn(10,1);
labels = {'Save sum of A to variable named:' ...
          'Save mean of A to variable named:'};
vars = {'sumA','meanA'};
vals = {sum(A),mean(A)};
title = ('Save Sums to Workspace');
defs = logical([0 0]);
export2wsdlg(labels,vars,vals,...
             title,defs);
```



If the user clicks **OK** when no check boxes are selected, MATLAB automatically returns an error dialog box. After clicking **OK** in the error dialog box, the user has the opportunity to correct the error in the Export to Workspace dialog box.



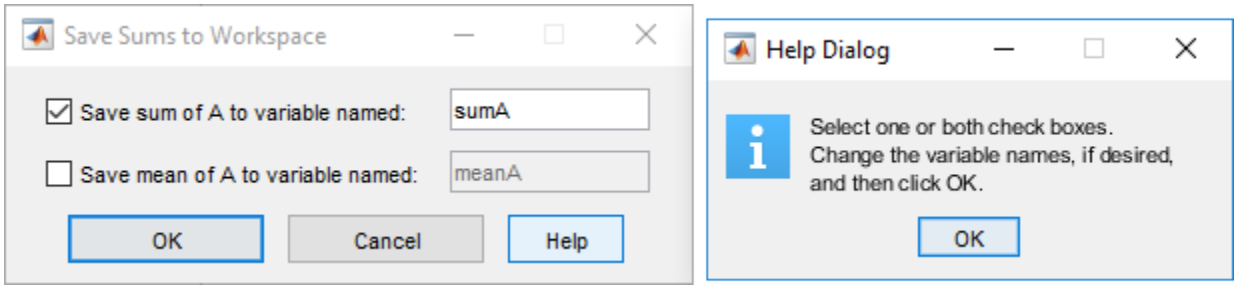
Add Help Button and Callback

Add a **Help** button to the dialog box by specifying a callback input argument. Clicking the **Help** button runs the callback. In this code, the callback function is `smhelp`, which creates a Help dialog box. For more information, see “Write Callbacks for Apps Created Programmatically”.

```
function expsm
A = randn(10,1);
labels = {'Save sum of A to variable named:' ...
          'Save mean of A to variable named:'};
vars = {'sumA','meanA'};
vals = {sum(A),mean(A)};
title = ('Save Sums to Workspace');
defs = logical([1 0]);
export2wsdlg(labels,vars,vals,...
             title,defs,{@smhelp});
```

```
    function smhelp
        helpdlg({'Select one or both check boxes.',...
                 'Change the variable names, if desired,',...
                 'and then click OK.'});
    end
end
```

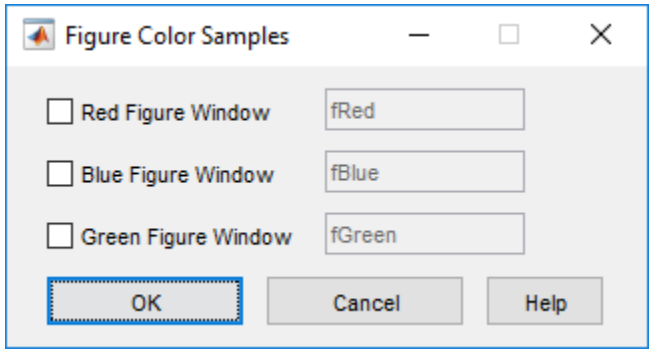
To run this example, copy and paste the preceding code into a text editor. Save the file as `exspm.m` to a folder on your MATLAB path. When you run the code and click the **Help** button, the Help Dialog box opens on top of the Save Sums to Workspace dialog box. The following image shows the two dialog boxes side by side.



Specify Function List

Create a dialog box to create and export figure objects to the workspace. Specify a function list to create figure windows in the specified colors when the user selects the corresponding check boxes, and then clicks **OK**. The `vals` input argument is a syntax placeholder only. Therefore, the cell array items specified by the value of `vals` are unused.

```
labels = {'Red Figure Window',...
          'Blue Figure Window',...
          'Green Figure Window'};
vars = {'fRed','fBlue','fGreen'};
vals = {0,0,0};
title = 'Figure Color Samples';
defs = [false false false];
flist = {{@figure,'color','r'},...
         {@figure,'color','b'},...
         {@figure,'color','g'}};
export2wsdlg(labels,vars,vals,title,defs,{@doc,'figure'},flist);
```



After the user makes check box selections and clicks **OK**, MATLAB does the following for the selected check boxes:

- Runs the corresponding functions (each of which creates a figure)
- Returns the figure objects to the `vars` variables
- Exports the variables to the base workspace.

Input Arguments

`labels` — Check box labels

cell array of character vectors | string array

Check box labels, specified as a cell array of character vectors or a string array. If there is only one item in the array, then `export2wsdlg` creates an edit field instead of a check box. The lengths of `labels`, `vars`, `vals`, and `defs` must all be equal.

Example: `{'Temperature (Celsius)','Mass (Grams)'}`

`vars` — Edit field variable names

cell array of character vectors | string array

Edit field variable names that appear in the edit fields by default when the dialog box opens, specified as a cell array of character vectors or a string array. The number of array elements is the same as the number specified in `labels`. The lengths of `labels`, `vars`, `vals`, and `defs` must all be equal.

The variable names in `vars` must be unique.

The dialog box user can edit the variable names displayed in the edit fields. If the user specifies the same name in multiple edit fields, then MATLAB creates a structure using that name. It then uses the `defaultvars` items as field names for that structure.

Example: `{'Longitude','Latitude'}`

`vals` — Variable values
cell array

Variable values, specified as a cell array. The lengths of `labels`, `vars`, `vals`, and `defs` must all be equal.

Example: `{100,200}`

Example: `{sin(A),cos(A)}`

`title` — Dialog box title
`'Export to Workspace'` (default) | character vector | string scalar

Dialog box title, specified as a character vector or string scalar.

Example: `'Save to Workspace'`

`defs` — Default check box selections
logical array

Default check box selections, specified as a logical array. The lengths of `labels`, `vars`, `vals`, and `defs` must all be equal.

By default all check boxes are selected when the dialog box opens.

Example: `[true, false]` specifies that when the dialog box opens the first check box is selected and the second is not.

`helpfcn` — Help button callback
function handle | cell array | character vector

Help button callback specified as a function handle, cell array, or character vector (not recommended). When you specify a **Help** button callback, MATLAB adds a Help button to the dialog box. For more information, see “Write Callbacks for Apps Created Programmatically”

Example: `@myfun` specifies the callback function as a function handle.

Example: `{@myfun,x}` specifies the callback function as a cell array. In this case, the function accepts the input argument `x`.

`flist` — List of function calls
cell array

A list of function calls, specified as a cell array of function names and optional arguments that calculate and then return the value to export. `flist` must be the same length as `labels`.

Example: `{{@myfun1,x},{@myfun2,x,y}}`

Output Arguments

`f` — Figure object
Figure object | [ ]

`Figure` object in which the dialog box is displayed. If two return arguments are requested, then `f` is returned as an empty array `[]` and the `export2wsdlg` function does not return values until the user closes the dialog box.

`tf` — True or false result
1 | 0

True or false result, returned as `1` or `0`. The function returns `1` (true) if the user clicks the **OK** button; otherwise, it returns `0`. The `export2wsdlg` function does not return values until the user closes the dialog box.

More About

Modal Dialog Box

A modal dialog box prevents a user from interacting with other MATLAB windows before responding to the modal dialog box.

Tips

- MATLAB program execution continues even when a modal Export to Workspace dialog box is active. To block program execution until the user closes the dialog box, use the `uiwait` function.

See Also

Workspace Browser

Introduced in R2006b

exportsetupdlg

Open figure Export Setup dialog box

Syntax

```
exportsetupdlg(f)
exportsetupdlg
```

Description

`exportsetupdlg(f)` displays the export settings dialog box. MATLAB applies your selections to the figure, `f`.

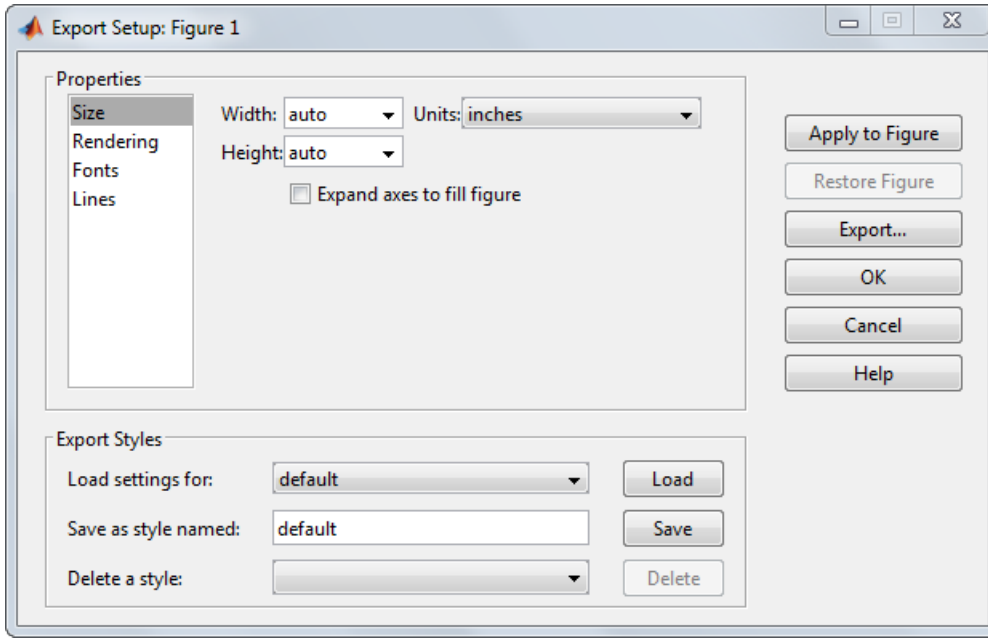
`exportsetupdlg` applies your selections to the current figure. If no figure exists, MATLAB creates a new figure.

Examples

Export Setting for a Figure

Create a figure and display the Export Setup dialog.

```
f = figure;
exportsetupdlg(f);
```



Input Arguments

f — Target figure

figure object

Target figure, specified as a figure object.

See Also

`printdlg` | `printpreview`

Topics

"Customize Figure Before Saving"

Introduced in R2006b

extractAfter

Extract substrings after specified positions

Syntax

```
newStr = extractAfter(str,startStr)
newStr = extractAfter(str,startPos)
```

Description

`newStr = extractAfter(str,startStr)` extracts the substring that begins after `startStr` and ends with the last character of `str`. If `startStr` occurs multiple times in `str`, then `newStr` is `str` from the first occurrence of `startStr` to the end.

If `str` is a string array or a cell array of character vectors, then `extractAfter` extracts substrings from each element of `str`. The output argument `newStr` has the same data type as `str`.

`newStr = extractAfter(str,startPos)` extracts the substring that begins after the position specified by `startPos` and ends with the last character of `str`.

Examples

Select Text After Substring

Create string arrays and select text that occurs after substrings.

Starting in R2017a, you can create strings using double quotes.

```
str = "The quick brown fox"
```

```
str =
"The quick brown fox"
```

Extract the substring that occurs after the substring `"quick "`. The `extractAfter` function selects the new text but does not include `"quick "` in the output.

```
newStr = extractAfter(str,"quick ")
```

```
newStr =
"brown fox"
```

Create a new string array from the elements of a string array. When you specify different substrings as positions, they must be contained in a string array or a cell array that is the same size as the input string array.

```
str = ["The quick brown fox jumps";"over the lazy dog"]
```

```
str = 2x1 string array
    "The quick brown fox jumps"
    "over the lazy dog"
```

```
newStr = extractAfter(str,["quick ";"the "])
```

```
newStr = 2x1 string array
    "brown fox jumps"
    "lazy dog"
```

You also can specify one substring as a position that is applied to all elements of the input string array.

Select Substrings After Position

Create strings after specified positions.

Starting in R2017a, you can create strings using double quotes.

```
str = "Edgar Allen Poe"
```

```
str =
"Edgar Allen Poe"
```

Select the substring after the 12th character.

```
newStr = extractAfter(str,12)
```

```
newStr =
"Poe"
```

Select substrings from each element of a string array. When you specify different positions with numeric arrays, they must be the same size as the input string array.

```
str = ["Edgar Allen Poe";"Louisa May Alcott"]
```

```
str = 2x1 string array
    "Edgar Allen Poe"
    "Louisa May Alcott"
```

```
newStr = extractAfter(str,[12;11])
```

```
newStr = 2x1 string array
    "Poe"
    "Alcott"
```

Select substrings from each element and specify the same position.

```
newStr = extractAfter(str,6)
```

```
newStr = 2x1 string array
    "Allen Poe"
    " May Alcott"
```

Select Text After Position in Character Vector

Create a character vector. Then create new character vectors that are substrings of `chr`.

```
chr = 'peppers and onions'
```

```
chr =
'peppers and onions'
```

Select text after the 12th position.

```
newChr = extractAfter(chr,12)
```

```
newChr =
'onions'
```

Select text after a substring.

```
newChr = extractAfter(chr,'and ')
```

```
newChr =
'onions'
```

Input Arguments

str — Input text

string array | character vector | cell array of character vectors

Input text, specified as a string array, a character vector, or a cell array of character vectors.

Data Types: `string` | `char` | `cell`

startStr — String that indicates start of substring to extract

string array | character vector | cell array of character vectors

String that indicates the start of the substring to extract, specified as a string array, a character vector, or a cell array of character vectors. `extractAfter` excludes `startStr` from the substring to extract.

If `str` is a string array or cell array of character vectors, then `startStr` can be a character vector, a string scalar, or a string array or a cell array of the same size as `str`.

Data Types: `string` | `char` | `cell`

startPos — Start position of substring to extract

numeric array

Start position of substring to extract, specified as a numeric array. `extractAfter` excludes the character at `start` from the substring to extract.

If `str` is a string array or cell array of character vectors, then `startPos` can be a numeric scalar or a numeric array of the same size as `str`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`newStr` — Output text
string array | character vector | cell array of character vectors

Output text, returned as a string array, a character vector, or a cell array of character vectors.

Data Types: `string` | `char` | `cell`

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `str` and `startStr` must be a string scalar, a character vector, or a cell array containing not more than one character vector.

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`count` | `erase` | `eraseBetween` | `extractBefore` | `extractBetween` | `insertAfter` | `insertBefore` | `join` | `replace` | `replaceBetween` | `size` | `split` | `strlength`

Topics

"Create String Arrays"
"Search and Replace Text"
"Test for Empty Strings and Missing Values"

Introduced in R2016b

extractBefore

Extract substrings before specified positions

Syntax

```
newStr = extractBefore(str,endStr)
newStr = extractBefore(str,endPos)
```

Description

`newStr = extractBefore(str,endStr)` extracts the substring that begins with the first character of `str` and ends before `endStr`. If `endStr` occurs multiple times in `str`, then `newStr` is `str` from the start of `str` up to the first occurrence of `endStr`

If `str` is a string array or a cell array of character vectors, then `extractBefore` extracts substrings from each element of `str`. The output argument `newStr` has the same data type as `str`.

`newStr = extractBefore(str,endPos)` extracts the substring that begins with the first character of `str` and ends before the position specified by `endPos`.

Examples

Select Text Before Substring

Create string arrays and select text that occurs before substrings.

Starting in R2017a, you can create strings using double quotes.

```
str = "The quick brown fox"
```

```
str =
"The quick brown fox"
```

Extract the substring that occurs before the substring `" brown"`. The `extractBefore` function selects the text but does not include `" brown"` in the output.

```
newStr = extractBefore(str," brown")
```

```
newStr =
"The quick"
```

Create a new string array from the elements of a string array. When you specify different substrings as positions, they must be contained in a string array or a cell array that is the same size as `str`.

```
str = ["The quick brown fox jumps";"over the lazy dog"]
```

```
str = 2x1 string array
    "The quick brown fox jumps"
    "over the lazy dog"
```

```
newStr = extractBefore(str,[" brown";" dog"])
```

```
newStr = 2x1 string array
    "The quick"
    "over the lazy"
```

You also can specify one substring as a position that is applied to all elements of the input string array.

Select Substrings Before Position

Create strings before specified positions.

Starting in R2017a, you can create strings using double quotes.

```
str = "Edgar Allen Poe"
```

```
str =
"Edgar Allen Poe"
```

Select the substring before the sixth character.

```
newStr = extractBefore(str,6)
```

```
newStr =
"Edgar"
```

Select substrings from each element of a string array. When you specify different positions with numeric arrays, they must be the same size as the input string array.

```
str = ["Edgar Allen Poe";"Louisa May Alcott"]
```

```
str = 2x1 string array
    "Edgar Allen Poe"
    "Louisa May Alcott"
```

```
newStr = extractBefore(str,[6;7])
```

```
newStr = 2x1 string array
    "Edgar"
    "Louisa"
```

Select substrings from each element and specify the same position.

```
newStr = extractBefore(str,12)
```

```
newStr = 2x1 string array
    "Edgar Allen"
    "Louisa May "
```

Select Text Before Position in Character Vector

Create a character vector. Then create new character vectors that are substrings of `chr`.

```
chr = 'peppers and onions'
```

```
chr =
'peppers and onions'
```

Select the substring before the eighth position.

```
newChr = extractBefore(chr,8)
```

```
newChr =
'peppers'
```

Select text before a substring.

```
newChr = extractBefore(chr,' and')
```

```
newChr =
'peppers'
```

Input Arguments

str — Input text
string array | character vector | cell array of character vectors

Input text, specified as a string array, a character vector, or a cell array of character vectors.

Data Types: `string` | `char` | `cell`

endStr — String that indicates end of substring to extract
string array | character vector | cell array of character vectors

String that indicates end of substring to extract, specified as a string array, a character vector, or a cell array of character vectors. `extractBefore` excludes `endStr` from the substring to extract.

If `str` is a string array or cell array of character vectors, then `endStr` can be a character vector, a string scalar, or a string array or a cell array of the same size as `str`.

Data Types: `string` | `char` | `cell`

endPos — End position of substring to extract
numeric array

End position of substring to extract, specified as a numeric array.

If `str` is a string array or cell array of character vectors, then `endPos` can be a numeric scalar or a numeric array of the same size as `str`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`newStr` — Output text
string array | character vector | cell array of character vectors

Output text, returned as a string array, a character vector, or a cell array of character vectors.

Data Types: `string` | `char` | `cell`

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `str` and `endStr` must be a string scalar, a character vector, or a cell array containing not more than one character vector.

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`count` | `erase` | `eraseBetween` | `extractAfter` | `extractBetween` | `insertAfter` | `insertBefore` | `join` | `replace` | `replaceBetween` | `size` | `split` | `strlength`

Topics

"Create String Arrays"
"Search and Replace Text"
"Test for Empty Strings and Missing Values"

Introduced in R2016b

extractBetween

Extract substrings between start and end points

Syntax

```
newStr = extractBetween(str,startStr,endStr)
newStr = extractBetween(str,startPos,endPos)
newStr = extractBetween( ___ ,'Boundaries',bounds)
```

Description

`newStr = extractBetween(str,startStr,endStr)` extracts the substring from `str` that occurs between the substrings `startStr` and `endStr`. The extracted substring does not include `startStr` and `endStr`.

`newStr` is a string array if `str` is a string array. Otherwise, `newStr` is a cell array of character vectors.

If `str` is a string array or a cell array of character vectors, then `extractBetween` extracts substrings from each element of `str`.

`newStr = extractBetween(str,startPos,endPos)` extracts the substring from `str` that occurs between the positions `startPos` and `endPos`, including the characters at those positions. `extractBetween` returns the substring as `newStr`.

`newStr = extractBetween( ___ ,'Boundaries',bounds)` forces the starts and ends specified in any of the previous syntaxes to be either inclusive or exclusive. They are inclusive when `bounds` is `'inclusive'`, and exclusive when `bounds` is `'exclusive'`. For example, `extractBetween(str,startStr,endStr,'Boundaries','inclusive')` returns `startStr`, `endStr`, and all the text between them as `newStr`.

Examples

Select Text Between Substrings

Create string arrays and select text that occurs between substrings.

Starting in R2017a, you can create strings using double quotes.

```
str = "The quick brown fox"
```

```
str =
"The quick brown fox"
```

Select the text that occurs between the substrings `"quick "` and `" fox"`. The `extractBetween` function selects the text but does not include `"quick "` or `" fox"` in the output.

```
newStr = extractBetween(str,"quick "," fox")
```

```
newStr =
"brown"
```

Select substrings from each element of a string array. When you specify different substrings as start and end indicators, they must be contained in a string array or a cell array that is the same size as `str`.

```
str = ["The quick brown fox jumps";"over the lazy dog"]
```

```
str = 2x1 string array
    "The quick brown fox jumps"
    "over the lazy dog"
```

```
newStr = extractBetween(str,["quick ";"the "],[" fox";" dog"])
```

```
newStr = 2x1 string array
    "brown"
    "lazy"
```

Select Substrings Between Start and End Positions

Create string arrays and select substrings between start and end positions that are specified as numbers.

Starting in R2017a, you can create strings using double quotes.

```
str = "Edgar Allen Poe"
```

```
str = 
"Edgar Allen Poe"
```

Select the middle name. Specify the seventh and 11th positions in the string.

```
newStr = extractBetween(str,7,11)
```

```
newStr = 
"Allen"
```

Select substrings from each element of a string array. When you specify different start and end positions with numeric arrays, they must be the same size as the input string array.

```
str = ["Edgar Allen Poe";"Louisa May Alcott"]
```

```
str = 2x1 string array
    "Edgar Allen Poe"
    "Louisa May Alcott"
```

```
newStr = extractBetween(str,[7;8],[11;10])
```

```
newStr = 2x1 string array
    "Allen"
    "May"
```

Select Text with Inclusive and Exclusive Boundaries

Select text from string arrays with boundaries that are forced to be inclusive or exclusive. `extractBetween` includes the boundaries with the selected text when the boundaries are inclusive. `extractBetween` does not include the boundaries with the selected text when the boundaries are exclusive.

Create a string array. Starting in R2017a, you can create strings using double quotes.

```
str1 = "small|medium|large"
```

```
str1 =
"small|medium|large"
```

Select the text between sixth and 13th positions, but do not include the characters at those positions.

```
newStr = extractBetween(str1,6,13,'Boundaries','exclusive')
```

```
newStr =
"medium"
```

Select the text between two substrings, and also the substrings themselves.

```
str2 = "The quick brown fox jumps over the lazy dog"
```

```
str2 =
"The quick brown fox jumps over the lazy dog"
```

```
newStr = extractBetween(str2," brown","jumps",'Boundaries','inclusive')
```

```
newStr =
" brown fox jumps"
```

Select Text Between Positions in Character Vector

Create a character vector and select text between start and end positions.

```
chr = 'mushrooms, peppers, and onions'
```

```
chr =
'mushrooms, peppers, and onions'
```

```
newChr = extractBetween(chr,12,18)
```

```
newChr = 1x1 cell array
    {'peppers'}
```

Select text between substrings.

```
newChr = extractBetween(chr,'mushrooms, ',', and')
```

```
newChr = 1x1 cell array
    {'peppers'}
```

Input Arguments

`str` — Input text
string array | character vector | cell array of character vectors

Input text, specified as a string array, a character vector, or a cell array of character vectors.

Data Types: `string` | `char` | `cell`

`startStr` — String that indicates start of substring to extract
string array | character vector | cell array of character vectors

String that indicates the start of the substring to extract, specified as a string array, a character vector, or a cell array of character vectors.

If `str` is a string array or cell array of character vectors, then `startStr` can be a character vector, a string scalar, or a string array or a cell array of the same size as `str`.

Example: `extractBetween(str,"AB","YZ")` extracts the substrings between `AB` and `YZ` in each element of `str`.

Example: If `str` is a 2-by-1 string array, then `extractBetween(str,["AB";"FG"],["YZ";"ST"])` extracts the substrings between `AB` and `YZ` in `str(1)`, and between `FG` and `ST` in `str(2)`.

Data Types: `string` | `char` | `cell`

`endStr` — String that indicates end of substring to extract
string array | character vector | cell array of character vectors

String that indicates the end of the substring to extract, specified as a string array, a character vector, or a cell array of character vectors.

If `str` is a string array or cell array of character vectors, then `endStr` can be a character vector, a string scalar, or a string array or a cell array of the same size as `str`.

Example: `extractBetween(str,"AB","YZ")` extracts the substrings between `AB` and `YZ` in each element of `str`.

Example: If `str` is a 2-by-1 string array, then `extractBetween(str,["AB";"FG"],["YZ";"ST"])` extracts the substrings between `AB` and `YZ` in `str(1)`, and between `FG` and `ST` in `str(2)`.

Data Types: `string` | `char` | `cell`

`startPos` — Start position of substring to extract
numeric array

Start position of substring to extract, specified as a numeric array.

If `str` is an array with multiple pieces of text, then `startPos` can be a numeric scalar or a numeric array of the same size as `str`.

Example: `extractBetween(str,5,9)` extracts the substrings from the fifth through the ninth positions in each element of `str`.

Example: If `str` is a 2-by-1 string array, then `extractBetween(str,[5;10],[9;21])` extracts the substring from the fifth through the ninth positions in `str(1)`, and from the 10th through the 21st positions in `str(2)`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`endPos` — End position of substring to extract
numeric array

End position of substring to extract, specified as a numeric array.

If `str` is an array with multiple pieces of text, then `endPos` can be a numeric scalar or a numeric array of the same size as `str`.

Example: `extractBetween(str,5,9)` extract the substrings from the fifth through the ninth positions in each element of `str`.

Example: If `str` is a 2-by-1 string array, then `extractBetween(str,[5;10],[9;21])` extracts the substrings from the fifth through the ninth positions in `str(1)`, and from the 10th through the 21st positions in `str(2)`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`newStr` — Output text
string array | cell array of character vectors

Output text, returned as a string array or a cell array of character vectors.

Data Types: `string` | `cell`

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitation:

Expansion in the first dimension is not supported with tall arrays.

For more information, see “Tall Arrays for Out-of-Memory Data”.

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also
`count` | `erase` | `eraseBetween` | `extractAfter` | `extractBefore` | `insertAfter` | `insertBefore` | `join` | `replace` | `replaceBetween` | `size` | `split` | `strlength`

Topics
“Create String Arrays”
“Search and Replace Text”

"Test for Empty Strings and Missing Values"

Introduced in R2016b

Find Local Extrema

Find local maxima and minima in the Live Editor

Description

The **Find Local Extrema** task lets you interactively find local maxima and local minima in data. The task automatically generates MATLAB code for your live script.

Using this task, you can:

- Find local maxima, minima, or both in data from a workspace variable.
- Adjust parameters to locate fewer or more extrema.
- Automatically visualize the detected extrema.

Open the Task

To add the **Find Local Extrema** task to a live script in the MATLAB Editor:

- On the **Live Editor** tab, select **Task > Find Local Extrema**.
- In a code block in the script, type a relevant keyword, such as `extrema` or `find`. Select **Find Local Extrema** from the suggested command completions.

Parameters

`Flat selection` — Flat region indicator

`Center` (default) | `First` | `Last` | `All`

Specify a flat region indicator when you have local consecutive extrema values.

Method	Description
`Center`	Indicates only the center element of a flat region as the local extrema.

Method	Description
`First`	Indicates only the first element of a flat region as the local extrema.
`Last`	Indicates only the last element of a flat region as the local extrema.
`All`	Indicates all the elements of a flat region as the local extrema.

`Min. prominence` — Minimum prominence of extrema
`0` (default) | positive scalar | ...

Specify a minimum prominence to return only local extrema whose prominence is at least that value. Higher prominence values return fewer detected extrema.

The prominence of a local maximum (or peak) is a measure of how the peak stands out with respect to its height and location relative to other peaks. The prominence of a local minimum (or valley) is a measure of how the valley stands out with respect to its depth and location relative to other valleys.

`Prominence window` — Window for computing prominence
`Centered` (default) | `Asymmetric`

Specify a prominence window type and size to define a window of neighboring points for which to compute the prominence for each local extrema.

Window	Description
`Centered`	Specified window length centered about the current point.
`Asymmetric`	Specified window containing the number of elements before the current point and the number of elements after the current point.

Window sizes are relative to the **X-axis** variable units.

See Also

Functions
`islocalmax` | `islocalmin`

Live Editor Tasks
Clean Missing Data | Clean Outlier Data | Find Change Points | Remove Trends | Smooth Data

Topics

"Add Interactive Tasks to a Live Script"
Clean Messy Data and Locate Extrema Using Live Editor Tasks

Introduced in R2019b

eye

Identity matrix

Syntax

```
I = eye
I = eye(n)
I = eye(n,m)
I = eye(sz)

I = eye(___,typename)
I = eye(___,'like',p)
```

Description

`I = eye` returns the scalar `1`.

`I = eye(n)` returns an `n`-by-`n` identity matrix with ones on the main diagonal and zeros elsewhere.

`I = eye(n,m)` returns an `n`-by-`m` matrix with ones on the main diagonal and zeros elsewhere.

`I = eye(sz)` returns an array with ones on the main diagonal and zeros elsewhere. The size vector, `sz`, defines `size(I)`. For example, `eye([2,3])` returns a 2-by-3 array with ones on the main diagonal and zeros elsewhere.

`I = eye(___,typename)` also specifies the data type (class) of `I` for any of the previous syntaxes. For example, `eye(5,'int8')` returns a 5-by-5 identity matrix consisting of 8-bit integers.

`I = eye(___,'like',p)` specifies that `I` has the same data type, sparsity, and complexity (real or complex) as the numeric variable `p`.

Examples

Square Identity Matrix

Create a 4-by-4 identity matrix.

```
I = eye(4)
```

```
I = 4×4

     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
```

Rectangular Matrix

Create a 2-by-3 identity matrix.

```
I = eye(2,3)
```

```
I = 2×3

     1     0     0
     0     1     0
```

Identity Vector

Create a 3-by-1 identity vector.

```
sz = [3,1];
I = eye(sz)
```

```
I = 3×1

     1
```

```
     0
     0
```

Nondefault Numeric Data Type

Create a 3-by-3 identity matrix whose elements are 32-bit unsigned integers.

```
I = eye(3,'uint32'),
```

```
I = 3x3 uint32 matrix

   1   0   0
   0   1   0
   0   0   1
```

```
class(I)
```

```
ans =
'uint32'
```

Complex Identity Matrix

Create a 2-by-2 identity matrix that is not real valued, but instead is complex like an existing array.

Define a complex vector.

```
p = [1+2i 3i];
```

Create an identity matrix that is complex like `p`.

```
I = eye(2,'like',p)
```

```
I = 2×2 complex

   1.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i   1.0000 + 0.0000i
```

Sparse Identity Matrix

Define a 5-by-5 sparse matrix.

```
p = sparse(5,5,pi);
```

Create a 5-by-5 identity matrix that is sparse like P.

```
I = eye(5,'like',p)
```

```
I =
   (1,1)        1
   (2,2)        1
   (3,3)        1
   (4,4)        1
   (5,5)        1
```

Size and Numeric Data Type Defined by Existing Array

Define a 2-by-2 matrix of single precision.

```
p = single([1 3 ; 2 4]);
```

Create an identity matrix that is the same size and data type as P.

```
I = eye(size(p),'like',p),
```

```
I = 2x2 single matrix

     1     0
     0     1
```

```
class(I)
```

```
ans =
'single'
```

Input Arguments

n — Size of first dimension of I

integer value

Size of first dimension of I, specified as an integer value.

- If n is the only integer input argument, then I is a square n-by-n identity matrix.
- If n is 0, then I is an empty matrix.
- If n is negative, then it is treated as 0.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

m — Size of second dimension of I

integer value

Size of second dimension of I, specified as an integer value.

- If m is 0, then I is an empty matrix.
- If m is negative, then it is treated as 0.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

sz — Size of I

row vector of no more than two integer values

Size of I, specified as a row vector of no more than two integer values.

- If an element of sz is 0, then I is an empty matrix.
- If an element of sz is negative, then the element is treated as 0.

Example: sz = [2,3] defines I as a 2-by-3 matrix.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

typename — Output class

'double' (default) | 'single' | 'logical' | 'int8' | 'uint8' | ...

Output class, specified as `'double'`, `'single'`, `logical`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, `'int64'`, or `'uint64'`.

p — Prototype
numeric variable

Prototype, specified as a numeric variable.

Data Types: `double` | `single` | `logical` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`
Complex Number Support: Yes

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `typename` must be a built-in MATLAB numeric type. Does not invoke the static `eye` method for other classes. For example, `eye(m, n, 'myclass')` does not invoke `myclass.eye(m,n)`.
- Size arguments must have a fixed size.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- `typename` must be a built-in MATLAB numeric type. Does not invoke the static `eye` method for other classes. For example, `eye(m, n, 'myclass')` does not invoke `myclass.eye(m,n)`.
- Size arguments must have a fixed size.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `eye` in the Parallel Computing Toolbox documentation.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `eye` in the Parallel Computing Toolbox documentation.

See Also

`ones` | `speye` | `zeros`

Topics

"Class Support for Array-Creation Functions"

Introduced before R2006a

ezcontour

(Not recommended) Easy-to-use contour plotter

Note `ezcontour` is not recommended. Use `fcontour` instead.

Syntax

```
ezcontour(fun)
ezcontour(fun,domain)
ezcontour(...,n)
ezcontour(axes_handle,...)
h = ezcontour(...)
```

Description

`ezcontour(fun)` plots the contour lines of `fun(x,y)` using the `contour` function. `fun` is plotted over the default domain: -2π < x < 2π, -2π < y < 2π.

`fun` can be a function handle for a MATLAB file function or an anonymous function (see "Create Function Handle" and "Anonymous Functions"), a character vector, or a string (see Tips on page 1-4021).

`ezcontour(fun,domain)` plots `fun(x,y)` over the specified `domain`. `domain` can be either a 4-by-1 vector `[xmin, xmax, ymin, ymax]` or a 2-by-1 vector `[min, max]` (where `min` < x < `max`, `min` < y < `max`).

`ezcontour(...,n)` plots `fun` over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezcontour(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = ezcontour(...)` returns the handle to a contour object in `h`.

`ezcontour` automatically adds a title and axis labels.

Examples

Create Contour Plot of Mathematical Expression

This mathematical expression defines a function of two variables, x and y.

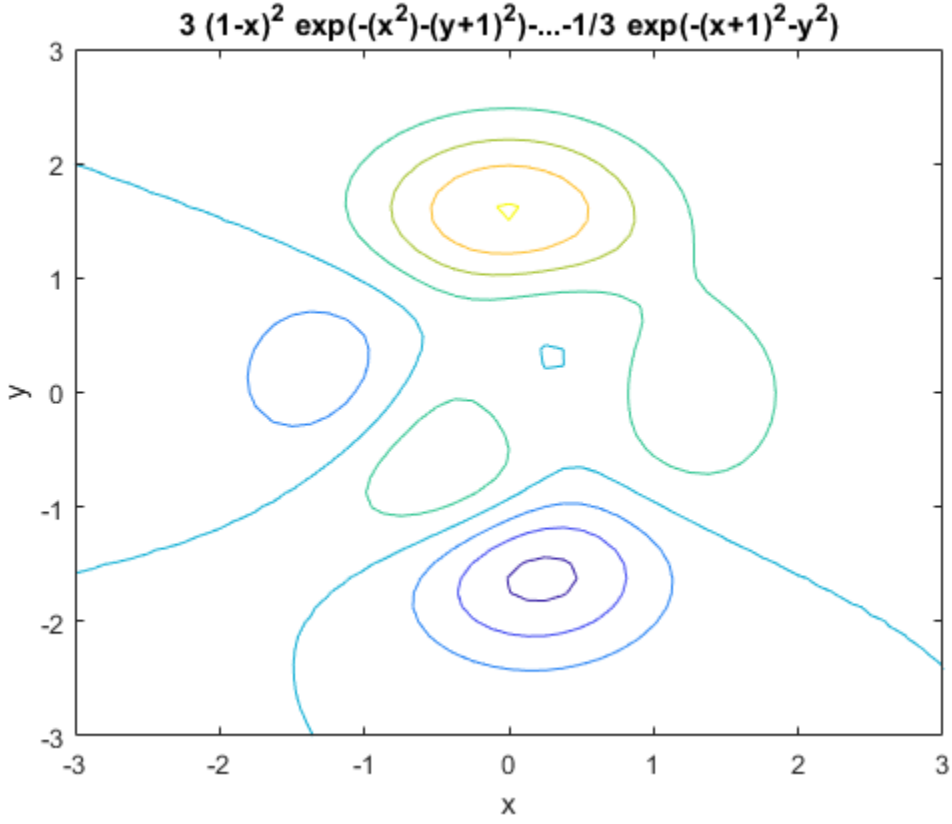
$$f(x,y) = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}$$

The `ezcontour` function requires a function handle argument. Write this mathematical expression in MATLAB® syntax as an anonymous function with handle `f`. You can define an anonymous function in the command window without creating a separate file. For convenience, write the function on three lines.

```
f = @(x,y) 3*(1-x).^2.*exp(-(x.^2) - (y+1).^2) ...
   - 10*(x/5 - x.^3 - y.^5).*exp(-x.^2-y.^2) ...
   - 1/3*exp(-(x+1).^2 - y.^2);
```

Pass the function handle, `f`, to `ezcontour`. Specify a domain from -3 to 3 in both the x-direction and y-direction and use a 49-by-49 computational grid.

```
ezcontour(f,[-3,3],49)
```



In this particular case, the title is too long to fit at the top of the graph so MATLAB® abbreviates it.

Tips

Passing the Function as a Character Vector or String

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezcontour`. For example, the MATLAB syntax for a contour plot of the expression

```
sqrt(x.^2 + y.^2)
```

is written as

```
ezcontour('sqrt(x^2 + y^2)')
```

That is, `x^2` is interpreted as `x.^2` in the character vector or string you pass to `ezcontour`.

If the function to be plotted is a function of the variables *u* and *v* (rather than *x* and *y*), the domain endpoints `umin`, `umax`, `vmin`, and `vmax` are sorted alphabetically. Thus, `ezcontour('u^2 - v^3',[0,1],[3,6])` plots the contour lines for $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezcontour`.

```
fh = @(x,y) sqrt(x.^2 + y.^2);
ezcontour(fh)
```

When using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezcontour` does not alter the syntax, as in the case with character vector or string inputs.

Passing Additional Arguments

If your function has additional parameters, for example, `k` in `myfun`:

```
function z = myfun(x,y,k)
z = x.^k - y.^k - 1;
```

then use an anonymous function to specify that parameter:

```
ezcontour(@(x,y)myfun(x,y,2))
```

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`contour` | `fcontour` | `fmesh` | `fplot` | `fplot3` | `fsurf`

Introduced before R2006a

ezcontourf

(Not recommended) Easy-to-use filled contour plotter

Note `ezcontourf` is not recommended. Use `fcontour` instead.

Syntax

```
ezcontourf(fun)
ezcontourf(fun,domain)
ezcontourf(...,n)
ezcontourf(axes_handle,...)
h = ezcontourf(...)
```

Description

`ezcontourf(fun)` plots the contour lines of `fun(x,y)` using the `contourf` function. `fun` is plotted over the default domain: -2π < `x` < 2π, -2π < `y` < 2π.

`fun` can be a function handle, a character vector, or a string (see Tips on page 1-4026).

`ezcontourf(fun,domain)` plots `fun(x,y)` over the specified `domain`. `domain` can be either a 4-by-1 vector `[xmin, xmax, ymin, ymax]` or a 2-by-1 vector `[min, max]`, where `min` < `x` < `max`, `min` < `y` < `max`).

`ezcontourf(...,n)` plots `fun` over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezcontourf(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = ezcontourf(...)` returns the handle to a contour object in `h`.

`ezcontourf` automatically adds a title and axis labels.

Examples

Create Filled Contour Plot of Mathematical Expression

This mathematical expression defines a function of two variables, x and y.

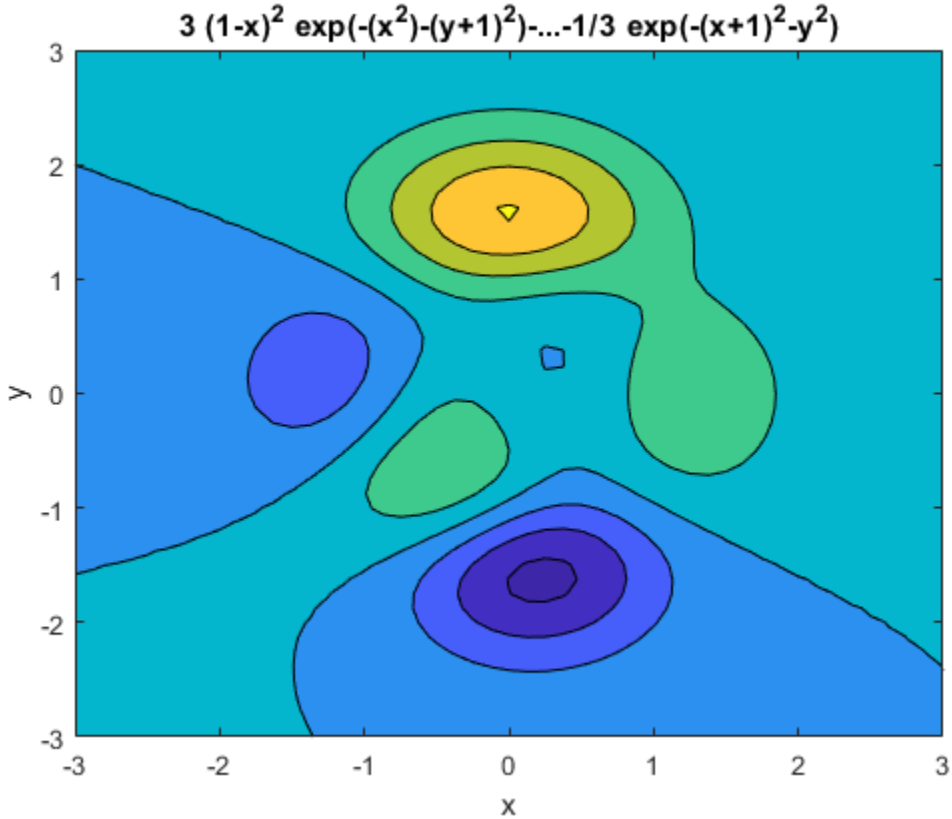
$$f(x,y) = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}$$

The `ezcontourf` function requires a function handle argument. Write this mathematical expression in MATLAB® syntax as an anonymous function with handle `f`. You can define an anonymous function in the command window without creating a separate file. For convenience, write the function on three lines.

```
f = @(x,y) 3*(1-x).^2.*exp(-(x.^2) - (y+1).^2) ...
   - 10*(x/5 - x.^3 - y.^5).*exp(-x.^2-y.^2) ...
   - 1/3*exp(-(x+1).^2 - y.^2);
```

Pass the function handle, `f`, to `ezcontourf`. Specify a domain from -3 to 3 in both the x-direction and y-direction and use a 49-by-49 computational grid.

```
ezcontourf(f,[-3,3],49)
```



In this particular case, the title is too long to fit at the top of the graph so MATLAB® abbreviates it.

Tips

Passing the Function as a Character Vector or String

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezcontourf`. For example, the MATLAB syntax for a filled contour plot of the expression

```
sqrt(x.^2 + y.^2);
```

is written as

```
ezcontourf('sqrt(x^2 + y^2)')
```

That is, `x^2` is interpreted as `x.^2` in the character vector or string you pass to `ezcontourf`.

If the function to be plotted is a function of the variables *u* and *v* (rather than *x* and *y*), then the domain endpoints `umin`, `umax`, `vmin`, and `vmax` are sorted alphabetically. Thus, `ezcontourf('u^2 - v^3',[0,1],[3,6])` plots the contour lines for $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezcontourf`.

```
fh = @(x,y) sqrt(x.^2 + y.^2);
ezcontourf(fh)
```

When using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezcontourf` does not alter the syntax, as in the case with character vector or string inputs.

Passing Additional Arguments

If your function has additional parameters, for example, `k` in `myfun`:

```
function z = myfun(x,y,k)
z = x.^k - y.^k - 1;
```

then you can use an anonymous function to specify that parameter:

```
ezcontourf(@(x,y)myfun(x,y,2))
```

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`contourf` | `fcontour` | `fmesh` | `fplot` | `fplot3` | `fsurf`

Topics

Anonymous Functions

Introduced before R2006a

ezmesh

(Not recommended) Easy-to-use 3-D mesh plotter

Note `ezmesh` is not recommended. Use `fmesh` instead.

Syntax

```
ezmesh(fun)
ezmesh(fun,domain)
ezmesh(funx,funy,funz)
ezmesh(funx,funy,funz,[smin,smax,tmin,tmax])
ezmesh(funx,funy,funz,[min,max])
ezmesh(...,n)
ezmesh(...,'circ')
ezmesh(axes_handle,...)
h = ezmesh(...)
```

Description

`ezmesh(fun)` creates a graph of `fun(x,y)` using the `mesh` function. `fun` is plotted over the default domain: -2π < `x` < 2π, -2π < `y` < 2π.

`fun` can be a function handle, a character vector, or a string (see the Tips on page 1-4032 section).

`ezmesh(fun,domain)` plots `fun` over the specified `domain`. `domain` can be either a 4-by-1 vector [`xmin`, `xmax`, `ymin`, `ymax`] or a 2-by-1 vector [`min`, `max`] (where `min` < `x` < `max`, `min` < `y` < `max`).

`ezmesh(funx,funy,funz)` plots the parametric surface `funx(s,t)`, `funy(s,t)`, and `funz(s,t)` over the square: -2π < `s` < 2π, -2π < `t` < 2π.

`ezmesh(funx,funy,funz,[smin,smax,tmin,tmax])` or `ezmesh(funx,funy,funz,[min,max])` plots the parametric surface using the specified domain.

`ezmesh(...,n)` plots `fun` over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezmesh(...,'circ')` plots `fun` over a disk centered on the domain.

`ezmesh(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

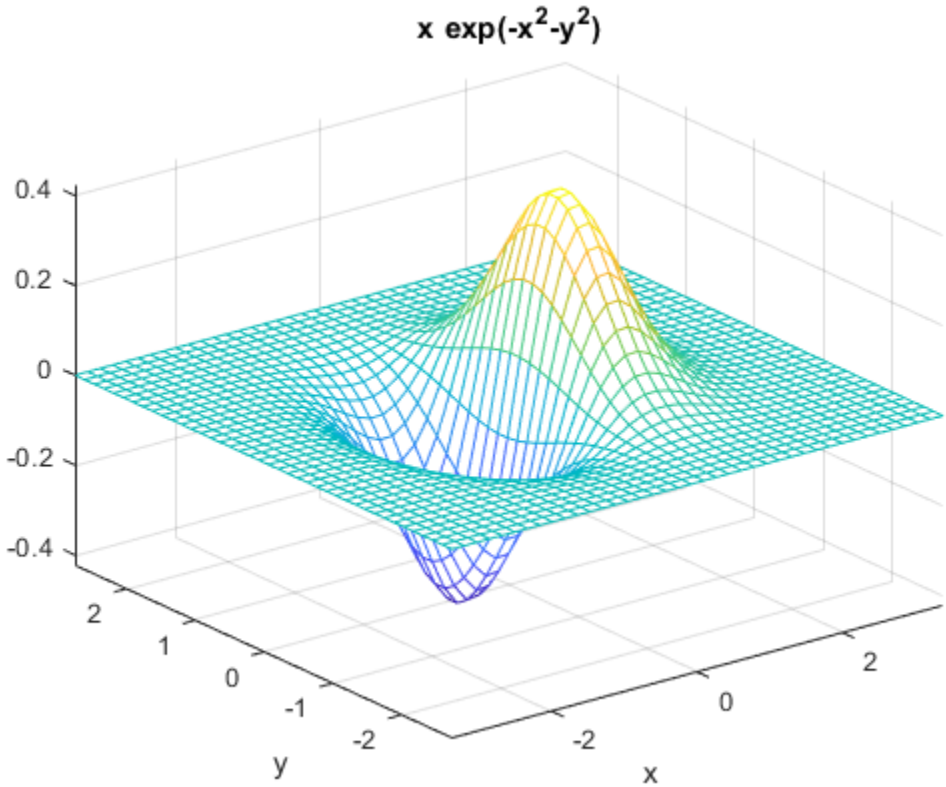
`h = ezmesh(...)` returns the handle to a surface object in `h`.

Examples

Mesh Plot of Mathematical Function

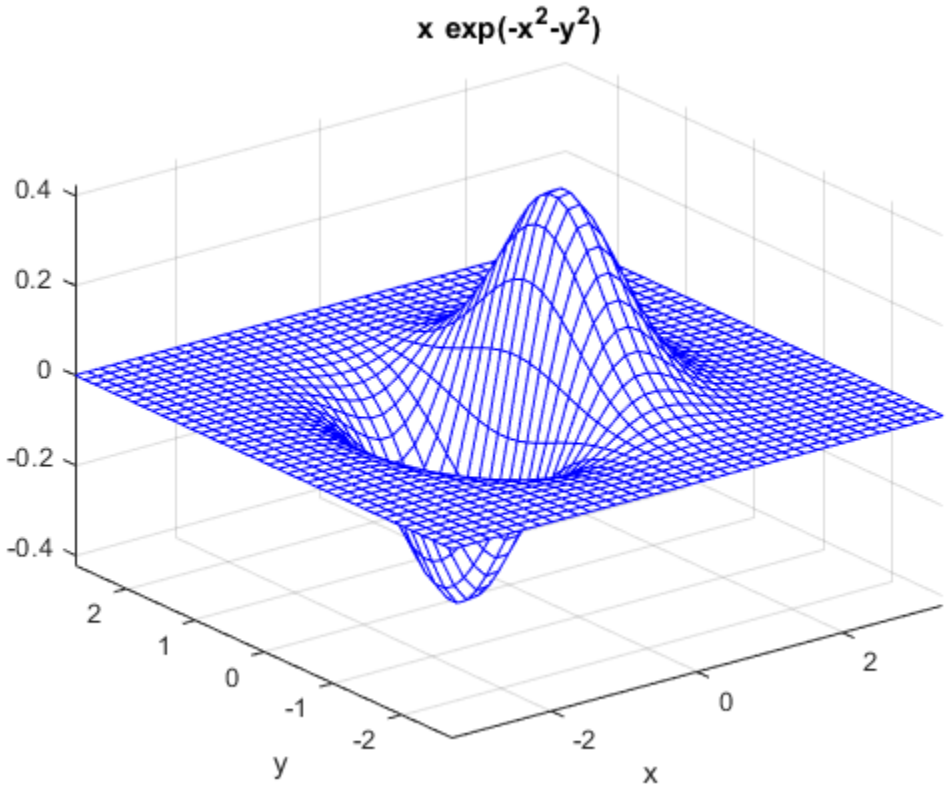
Create a mesh plot of the function $f(x, y) = xe^{-x^2 - y^2}$ over a 40-by-40 grid.

```
fh = @(x,y) x.*exp(-x.^2-y.^2);
ezmesh(fh,40)
```



Set the mesh lines to a uniform blue color by setting the colormap to a single color.

```
colormap([0 0 1])
```



Tips

Passing the Function as a Character Vector or String

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezmesh`. For example, the MATLAB syntax for a mesh plot of the expression

```
sqrt(x.^2 + y.^2);
```

is written as

```
ezmesh('sqrt(x^2 + y^2)')
```

That is, `x^2` is interpreted as `x.^2` in the character vector or string you pass to `ezmesh`.

If the function to be plotted is a function of the variables *u* and *v* (rather than *x* and *y*), then the domain endpoints `umin`, `umax`, `vmin`, and `vmax` are sorted alphabetically. Thus, `ezmesh('u^2 - v^3',[0,1],[3,6])` plots $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezmesh`.

```
fh = @(x,y) sqrt(x.^2 + y.^2);
ezmesh(fh)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezmesh` does not alter the syntax, as in the case with character vector or string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k` in `myfun`:

```
function z = myfun(x,y,k)
z = x.^k - y.^k - 1;
```

then you can use an anonymous function to specify that parameter:

```
ezmesh(@(x,y)myfun(x,y,2))
```

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`fcontour` | `fmesh` | `fplot` | `fplot3` | `fsurf` | `mesh`

Topics

Anonymous Functions

Introduced before R2006a

ezmeshc

(Not recommended) Easy-to-use combination mesh/contour plotter

Note `ezmeshc` is not recommended. Use `fmesh` instead.

Syntax

```
ezmeshc(fun)
ezmeshc(fun,domain)
ezmeshc(funx,funy,funz)
ezmeshc(funx,funy,funz,[smin,smax,tmin,tmax])
ezmeshc(funx,funy,funz,[min,max])
ezmeshc(...,n)
ezmeshc(...,'circ')
ezmesh(axes_handle,...)
h = ezmeshc(...)
```

Description

`ezmeshc(fun)` creates a graph of `fun(x,y)` using the `meshc` function. `fun` is plotted over the default domain -2π < `x` < 2π, -2π < `y` < 2π.

`fun` can be a function handle, a character vector, or a string (see the Tips on page 1-4037 section).

`ezmeshc(fun,domain)` plots `fun` over the specified `domain`. `domain` can be either a 4-by-1 vector [`xmin`, `xmax`, `ymin`, `ymax`] or a 2-by-1 vector [`min`, `max`] (where `min` < `x` < `max`, `min` < `y` < `max`).

`ezmeshc(funx,funy,funz)` plots the parametric surface `funx(s,t)`, `funy(s,t)`, and `funz(s,t)` over the square: -2π < `s` < 2π, -2π < `t` < 2π.

`ezmeshc(funx,funy,funz,[smin,smax,tmin,tmax])` or `ezmeshc(funx,funy,funz,[min,max])` plots the parametric surface using the specified domain.

`ezmeshc(...,n)` plots `fun` over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezmeshc(...,'circ')` plots `fun` over a disk centered on the domain.

`ezmesh(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

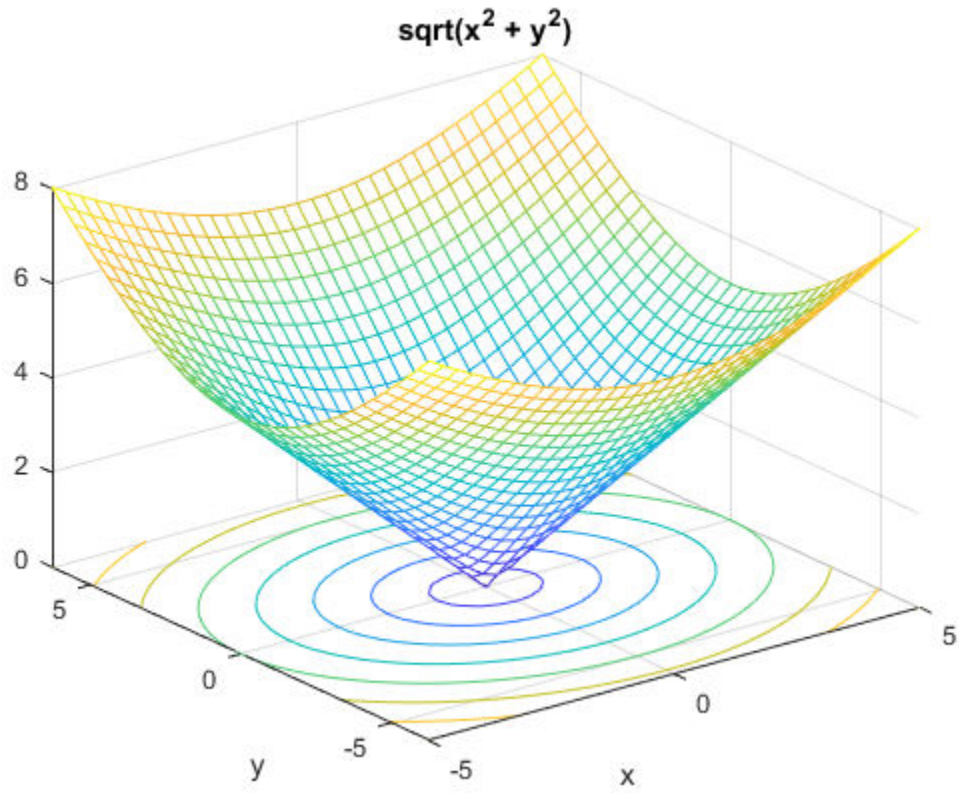
`h = ezmeshc(...)` returns the handle to a surface object in `h`.

Examples

Mesh and Contour Plot of Mathematical Function

Create a mesh/contour plot of the expression $f(x, y) = \sqrt{x^2 + y^2}$ over the domain $-5 < x < 5$ and $-2\pi < y < 2\pi$ with a computational grid size of 35-by-35.

```
ezmeshc('sqrt(x^2 + y^2)',[-5,5,-2*pi,2*pi],35)
```



Tips

Passing the Function as a Character Vector or String

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezmeshc`. For example, the MATLAB syntax for a mesh/contour plot of the expression

```
sqrt(x.^2 + y.^2);
```

is written as

```
ezmeshc('sqrt(x^2 + y^2)')
```

That is, `x^2` is interpreted as `x.^2` in the character vector or string you pass to `ezmeshc`.

If the function to be plotted is a function of the variables *u* and *v* (rather than *x* and *y*), then the domain endpoints `umin`, `umax`, `vmin`, and `vmax` are sorted alphabetically. Thus, `ezmeshc('u^2 - v^3',[0,1],[3,6])` plots $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezmeshc`.

```
fh = @(x,y) sqrt(x.^2 + y.^2);
ezmeshc(fh)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezmeshc` does not alter the syntax, as in the case with character vector or string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k` in `myfun`:

```
function z = myfun(x,y,k)
z = x.^k - y.^k - 1;
```

then you can use an anonymous function to specify that parameter:

```
ezmeshc(@(x,y)myfun(x,y,2))
```

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`fcontour` | `fmesh` | `fplot` | `fplot3` | `fsurf` | `mesh`

Topics

Anonymous Functions

ezplot

(Not recommended) Easy-to-use function plotter

Note `ezplot` is not recommended. Use `fplot` instead.

Syntax

```
ezplot(fun)
ezplot(fun,[xmin,xmax])
ezplot(fun2)
ezplot(fun2,[xymin,xymax])
ezplot(fun2,[xmin,xmax,ymin,ymax])
ezplot(funx,funy)
ezplot(funx,funy,[tmin,tmax])
ezplot(...,fig)
ezplot(ax,...)
h = ezplot(...)
```

Description

`ezplot(fun)` plots the expression `fun(x)` over the default domain -2π < x < 2π, where `fun(x)` is an explicit function of only `x`.

`fun` can be a function handle, a character vector, or a string.

`ezplot(fun,[xmin,xmax])` plots `fun(x)` over the domain: `xmin` < `x` < `xmax`.

For an implicit function, `fun2(x,y)`:

`ezplot(fun2)` plots `fun2(x,y) = 0` over the default domain -2π < x < 2π, -2π < y < 2π.

`ezplot(fun2,[xymin,xymax])` plots `fun2(x,y) = 0` over `xymin` < `x` < `xymax` and `xymin` < `y` < `xymax`.

`ezplot(fun2,[xmin,xmax,ymin,ymax])` plots `fun2(x,y) = 0` over `xmin` < `x` < `xmax` and `ymin` < `y` < `ymax`.

`ezplot(funx,funy)` plots the parametrically defined planar curve `funx(t)` and `funy(t)` over the default domain 0 < `t` < 2π.

`ezplot(funx,funy,[tmin,tmax])` plots `funx(t)` and `funy(t)` over `tmin` < `t` < `tmax`.

`ezplot(...,fig)` plots into the figure window identified by `fig`. Use any of the input argument combinations in the previous syntaxes that include a domain. The domain options are `[xmin xmax]`, `[xymin xymax]`, `[xmin xmax ymin ymax]`, and `[tmin tmax]`.

`ezplot(ax,...)` plots into the axes `ax` instead of the current axes (`gca`).

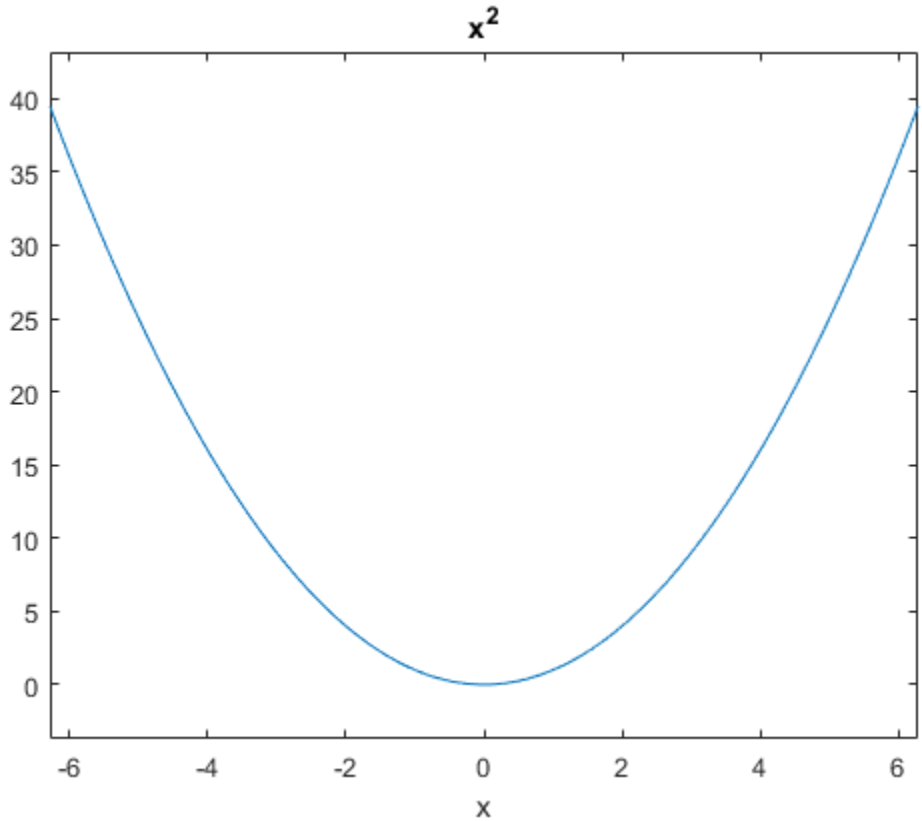
`h = ezplot(...)` returns either a chart line or contour object.

Examples

Plot an Explicit Function

Plot the explicit function x^2 over the domain $[-2\pi, 2\pi]$.

```
ezplot('x^2')
```

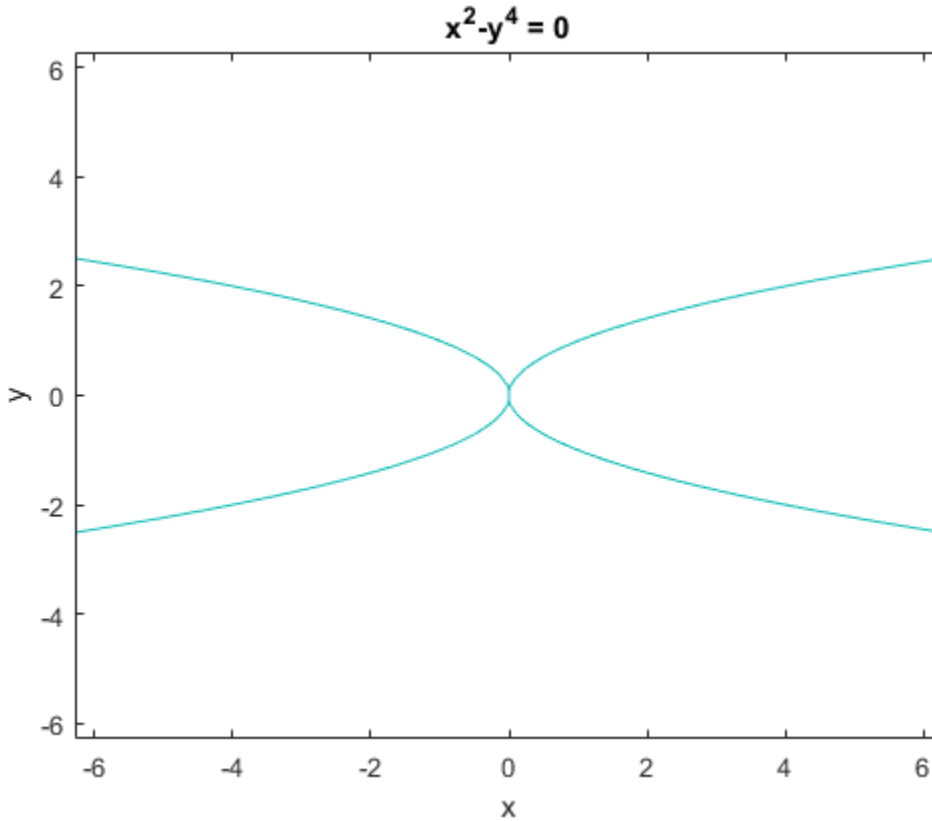


The default domain is $[-2\pi, 2\pi]$.

Plot an Implicit Function

Plot the implicitly defined function $x^2 - y^4 = 0$ over the domain $[-2\pi, 2\pi]$.

```
ezplot('x^2-y^4')
```



The default domain is $[-2\pi, 2\pi]$.

Tips

Passing the Function as a Character Vector or String

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezplot`. For example, the MATLAB syntax for a plot of the expression

```
x.^2 - y.^2
```

which represents an implicitly defined function, is written as

```
ezplot('x^2 - y^2')
```

That is, `x^2` is interpreted as `x.^2` in the character vector or string you pass to `ezplot`.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezplot`.

```
fh = @(x,y) x.^2 + y.^3 - 2*y - 1;
ezplot(fh)
axis equal
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezplot` does not alter the syntax, as in the case with character vector or string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k` in `myfun`:

```
function z = myfun(x,y,k)
z = x.^k - y.^k - 1;
```

then you can use an anonymous function to specify that parameter:

```
ezplot(@(x,y)myfun(x,y,2))
```

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`fcontour` | `fmesh` | `fplot` | `fplot3` | `fsurf` | `plot`

Topics

Anonymous Functions

Introduced before R2006a

ezplot3

(Not recommended) Easy-to-use 3-D parametric curve plotter

Note `ezplot3` is not recommended. Use `fplot3` instead.

Syntax

```
ezplot3(funx,funy,funz)
ezplot3(funx,funy,funz,[tmin,tmax])
ezplot3(...,'animate')
ezplot3(axes_handle,...)
h = ezplot3(...)
```

Description

`ezplot3(funx,funy,funz)` plots the spatial curve `funx(t)`, `funy(t)`, and `funz(t)` over the default domain $0 < t < 2\pi$.

`funx`, `funy`, and `funz` can be function handles, character vectors, or strings (see the Tips on page 1-4048 section).

`ezplot3(funx,funy,funz,[tmin,tmax])` plots the curve `funx(t)`, `funy(t)`, and `funz(t)` over the domain `tmin` < `t` < `tmax`.

`ezplot3(...,'animate')` produces an animated trace of the spatial curve.

`ezplot3(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = ezplot3(...)` returns the handle to the plotted objects in `h`.

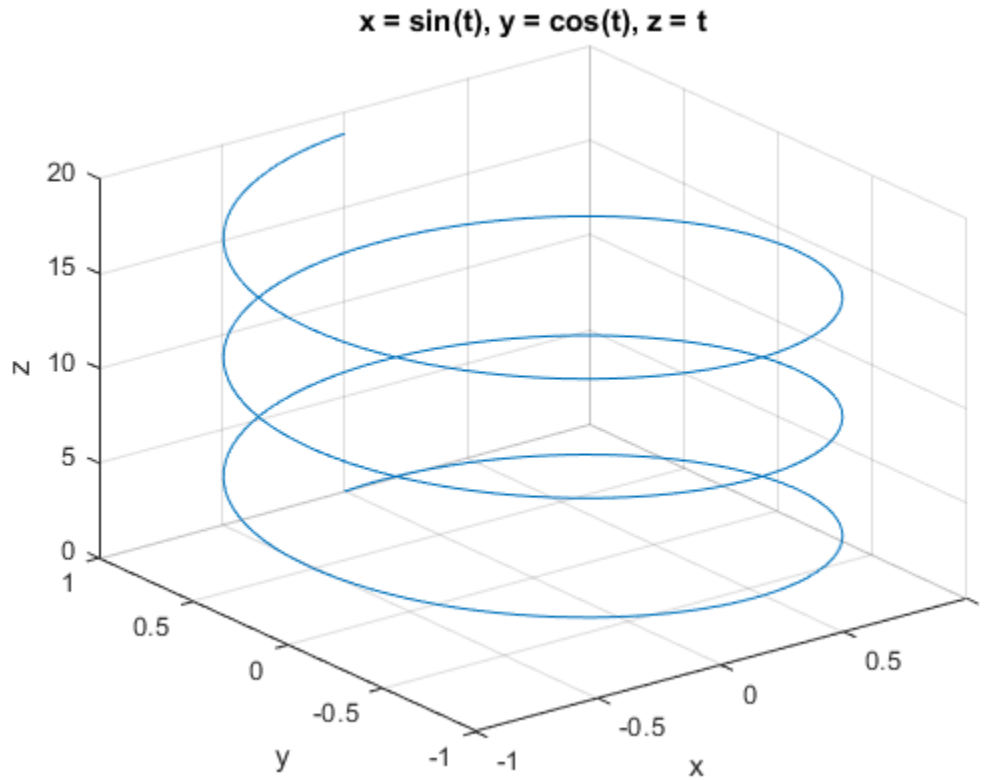
Examples

Plot a Parametric Curve

Plot this parametric curve over the domain $[0, 6\pi]$.

$$x = \sin(t), \quad y = \cos(t), \quad z = t$$

```
ezplot3('sin(t)','cos(t)','t',[0,6*pi])
```



Tips

Passing the Function as a Character Vector or String

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezplot3`. For example, the MATLAB syntax for a plot of the expression

```
x = s./2, y = 2.*s, z = s.^2;
```

which represents a parametric function, is written as

```
ezplot3('s/2','2*s','s^2')
```

That is, `s/2` is interpreted as `s./2` in the character vector or string you pass to `ezplot3`.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezplot3`.

```
fh1 = @(s) s./2; fh2 = @(s) 2.*s; fh3 = @(s) s.^2;
ezplot3(fh1,fh2,fh3)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezplot3` does not alter the syntax, as in the case with character vector or string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k` in `myfuntk`:

```
function s = myfuntk(t,k)
s = t.^k.*sin(t);
```

then you can use an anonymous function to specify that parameter:

```
ezplot3(@cos,@(t)myfuntk(t,1),@sqrt)
```

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`fcontour` | `fmesh` | `fplot` | `fplot3` | `fsurf` | `plot3`

Topics

Anonymous Functions

Introduced before R2006a

ezpolar

Easy-to-use polar coordinate plotter

Syntax

```
ezpolar(fun)
ezpolar(fun,[a,b])
ezpolar(axes_handle,...)
h = ezpolar(...)
```

Description

`ezpolar(fun)` plots the polar curve `rho = fun(theta)` over the default domain 0 < `theta` < 2π.

`fun` can be a function handle, a character vector, or a string (see the Tips on page 1-4051 section).

`ezpolar(fun,[a,b])` plots `fun` for `a` < `theta` < `b`.

`ezpolar(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

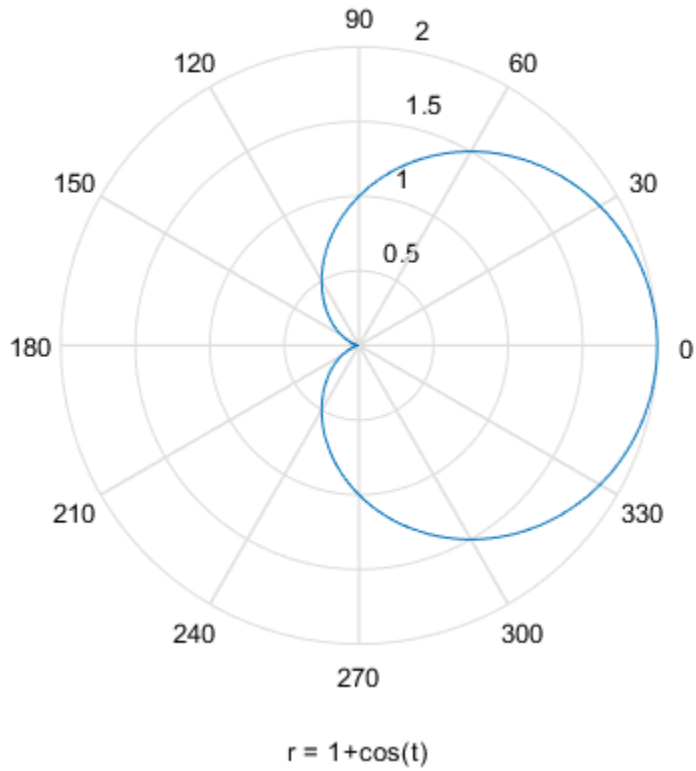
`h = ezpolar(...)` returns the handle to a line object in `h`.

Examples

Polar Plot of Mathematical Function

Plot the function $1 + \cos(t)$ over the domain $[0, 2\pi]$.

```
figure
ezpolar('1+cos(t)')
```



Tips

Passing the Function as a Character Vector or String

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezpolar`. For example, the MATLAB syntax for a plot of the expression

```
t.^2.*cos(t)
```

which represents an implicitly defined function, is written as

```
ezpolar('t^2*cos(t)')
```

That is, `t^2` is interpreted as `t.^2` in the character vector or string you pass to `ezpolar`.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezpolar`.

```
fh = @(t) t.^2.*cos(t);
ezpolar(fh)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezpolar` does not alter the syntax, as in the case with character vector or string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k1` and `k2` in `myfun`:

```
function s = myfun(t,k1,k2)
s = sin(k1*t).*cos(k2*t);
```

then you can use an anonymous function to specify the parameters:

```
ezpolar(@(t)myfun(t,2,3))
```

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`fcontour` | `fmesh` | `fplot` | `fplot3` | `fsurf` | `polarplot`

Topics

Anonymous Functions

Introduced before R2006a

ezsurf

(Not recommended) Easy-to-use 3-D colored surface plotter

Note `ezsurf` is not recommended. Use `fsurf` instead.

Syntax

```
ezsurf(fun)
ezsurf(fun,domain)
ezsurf(funx,funy,funz)
ezsurf(funx,funy,funz,[smin,smax,tmin,tmax])
ezsurf(funx,funy,funz,[min,max])
ezsurf(...,n)
ezsurf(...,'circ')
ezsurf(axes_handle,...)
h = ezsurf(...)
```

Description

`ezsurf(fun)` creates a graph of `fun(x,y)` using the `surf` function. `fun` is plotted over the default domain: -2π < `x` < 2π, -2π < `y` < 2π.

`fun` can be a function handle, a character vector, or a string (see the Tips on page 1-4057 section).

`ezsurf(fun,domain)` plots `fun` over the specified `domain`. `domain` must be a vector. See the "Algorithms" on page 1-4058 section for details on vector inputs vs axes limit outputs.

`ezsurf(funx,funy,funz)` plots the parametric surface `funx(s,t)`, `funy(s,t)`, and `funz(s,t)` over the square: -2π < `s` < 2π, -2π < `t` < 2π.

`ezsurf(funx,funy,funz,[smin,smax,tmin,tmax])` or `ezsurf(funx,funy,funz,[min,max])` plots the parametric surface using the specified domain.

`ezsurf(...,n)` plots `fun` over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezsurf(...,'circ')` plots `fun` over a disk centered on the domain.

`ezsurf(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

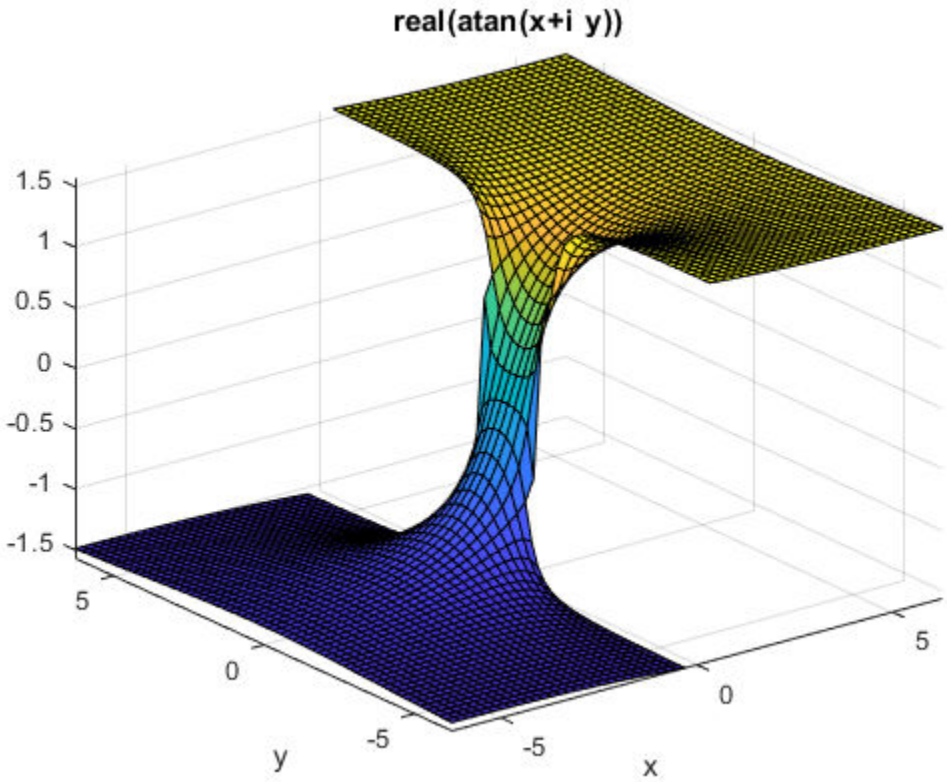
`h = ezsurf(...)` returns the handle to a surface object in `h`.

Examples

Surface Plot of Mathematical Function

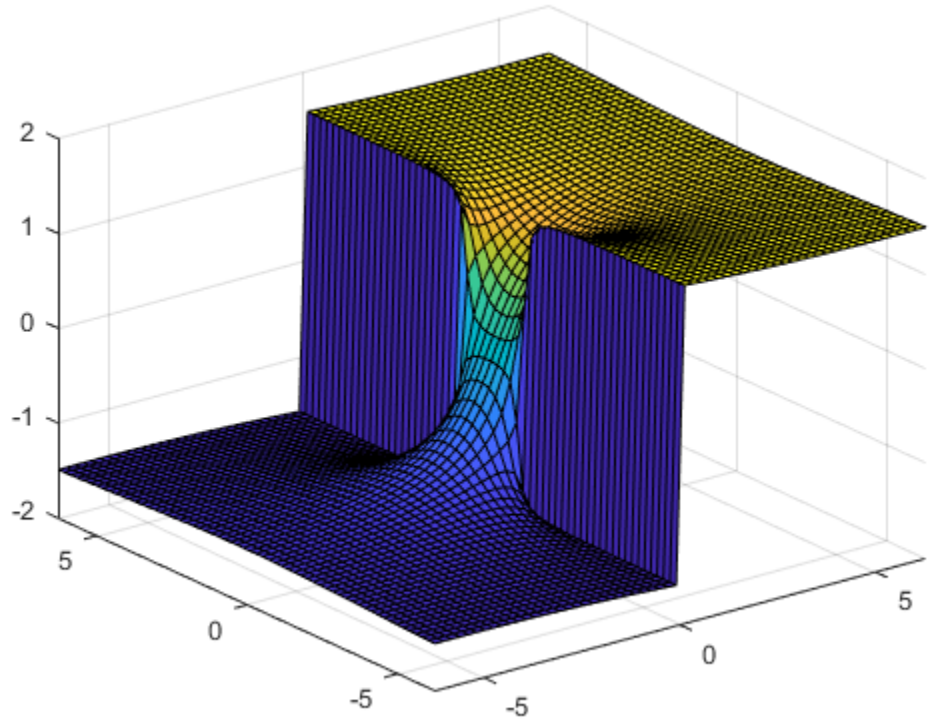
Plot the function $f(x, y) = real(atan(x + iy))$ over the domain $-2\pi < x < 2\pi$ and $-2\pi < y < 2\pi$. The `ezsurf` function does not plot points where the mathematical function is not defined. These points are set to `NaN` so that they do not plot.

```
figure
ezsurf('real(atan(x+i*y))')
```



Use `surf` to plot the same data without filtering discontinuities.

```
figure
[x,y] = meshgrid(linspace(-2*pi,2*pi,60));
z = real(atan(x+1i.*y));
surf(x,y,z)
```



Tips

`ezsurf` and `ezsurfc` do not accept complex inputs.

Passing the Function as a Character Vector or String

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezsurf`. For example, the MATLAB syntax for a surface plot of the expression

```
sqrt(x.^2 + y.^2);
```

is written as

```
ezsurf('sqrt(x^2 + y^2)')
```

That is, `x^2` is interpreted as `x.^2` in the character vector or string you pass to `ezsurf`.

If the function to be plotted is a function of the variables *u* and *v* (rather than *x* and *y*), then the domain endpoints `umin`, `umax`, `vmin`, and `vmax` are sorted alphabetically. Thus, `ezsurf('u^2 - v^3',[0,1],[3,6])` plots $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezsurf`.

```
fh = @(x,y) sqrt(x.^2 + y.^2);
ezsurf(fh)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezsurf` does not alter the syntax, as in the case with character vector or string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k` in `myfun`:

```
function z = myfun(x,y,k1,k2,k3)
z = x.*(y.^k1)./(x.^k2 + y.^k3);
```

then you can use an anonymous function to specify that parameter:

```
ezsurf(@(x,y)myfun(x,y,2,2,4))
```

Algorithms

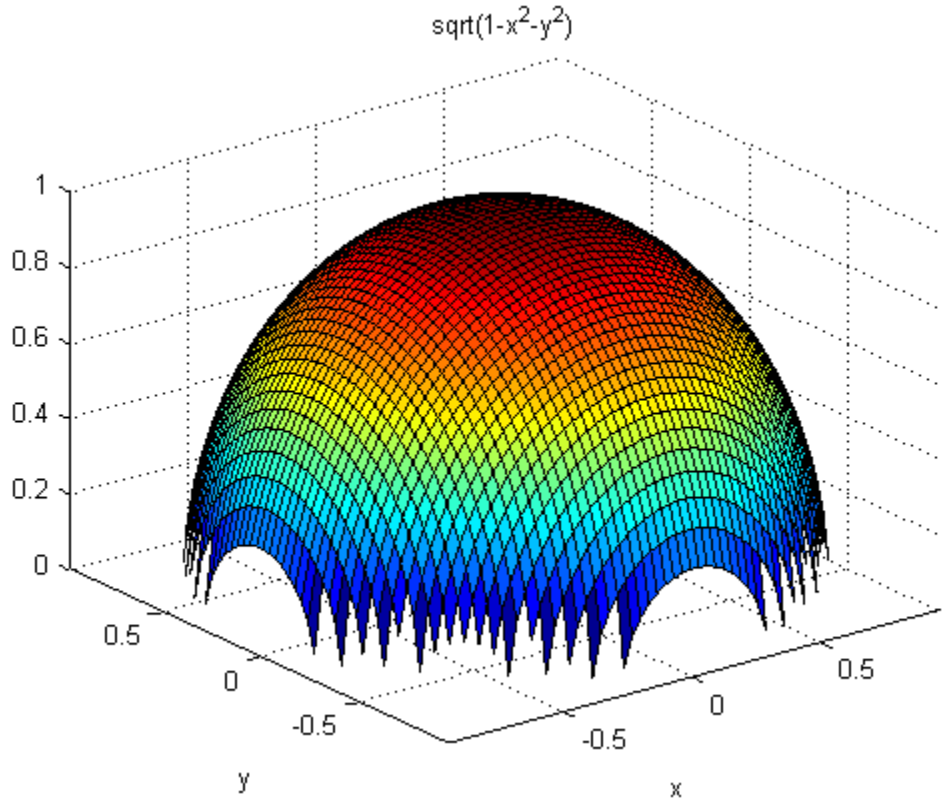
`ezsurf` determines the *x*- and *y*-axes limits in different ways depending on how you input the domain (if at all). In the following table, `R` is the vector [`xmin`, `xmax`, `ymin`, `ymax`] and `v` is the manually entered domain vector.

Number of domain values specified:	Resulting domain vector:
`v = [ ];`	`R = [-2*pi, 2*pi, -2*pi, 2*pi];`
`v = [ v(1) ];`	`R = double([-abs(v),abs(v),-abs(v),abs(v)]);`
`v = [ v(1) v(2) ];`	`R = double([v(1),v(2),v(1),v(2)]);`
`v = [ v(1) v(2) v(3) ];`	`R = double([-v(1),v(2),-abs(v(3)),abs(v(3))]);`
`v = [ v(1) v(2) v(3) v(4) ];`	`R = double(v);`
`v = [ v(1)..v(n) ];` `n>4`	`R = double([-abs(v(1)), abs(v(1)), -abs(v(1)), abs(v(1))]);`

If you specify a single number in non-vector format (without square brackets, [ ]), `ezsurf` interprets it as the `n`, the number of points desired between the axes `max` and `min` values.

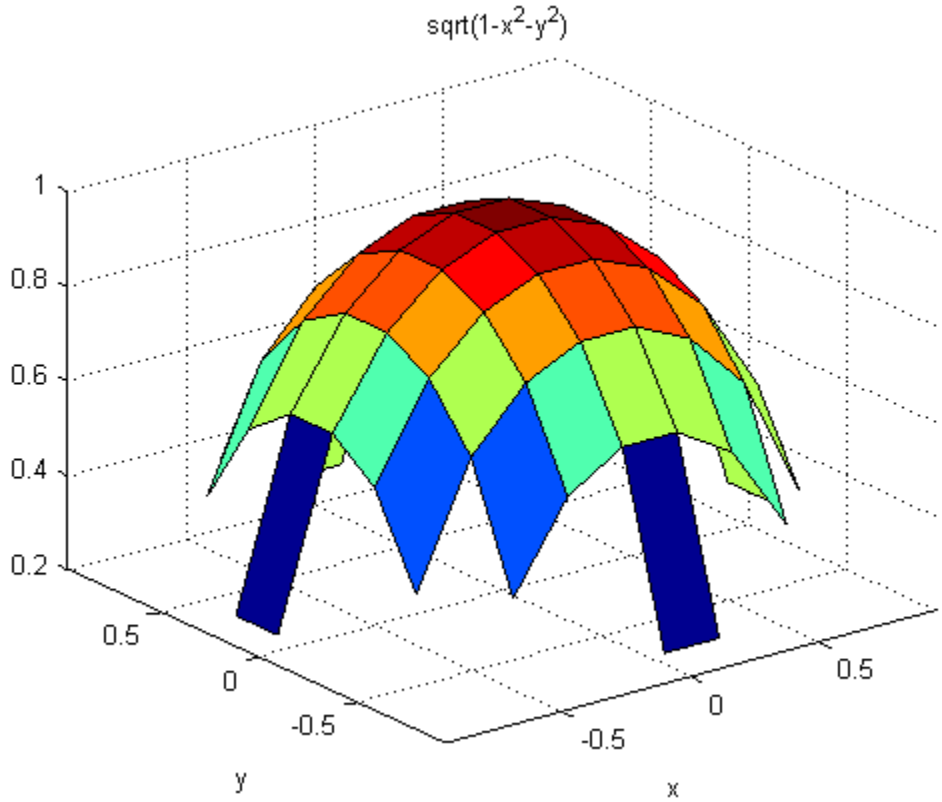
By default, `ezsurf` uses 60 points between the `max` and `min` values of an axes. When the `min` and `max` values are the default values (`R = [-2*pi, 2*pi, -2*pi, 2*pi];`), `ezsurf` ensures the 60 points fall within the non-complex range of the specified equation. For example, $\sqrt{1 - x^2 - y^2}$ is only real when $x^2 - y^2 \leq 1$. The default graph of this function looks like this:

```
ezsurf('sqrt(1-x^2-y^2)')
```



You can see that there are 60 points between the minimum and maximum values for which $\sqrt{1-x^2-y^2}$ has real values. However, when you specify the domain values to be the same as the default (`R = [-2*pi, 2*pi, -2*pi, 2*pi];`), a different result appears:

```
ezsurf('sqrt(1-x^2-y^2)',[-2*pi 2*pi])
```



In this case, the graphic limits are the same, but `ezsurf` used 60 points between the user-defined limits instead of checking to see if all those points would have real answers.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`fcontour` | `fmesh` | `fplot` | `fplot3` | `fsurf` | `surf`

Topics

Anonymous Functions

Introduced before R2006a

ezsurfc

(Not recommended) Easy-to-use combination surface/contour plotter

Note `ezsurfc` is not recommended. Use `fsurf` instead.

Syntax

```
ezsurfc(fun)
ezsurfc(fun,domain)
ezsurfc(funx,funy,funz)
ezsurfc(funx,funy,funz,[smin,smax,tmin,tmax])
ezsurfc(funx,funy,funz,[min,max])
ezsurfc(...,n)
ezsurfc(...,'circ')
ezsurfc(axes_handle,...)
h = ezsurfc(...)
```

Description

`ezsurfc(fun)` creates a graph of `fun(x,y)` using the `surfc` function. The function `fun` is plotted over the default domain: -2π < `x` < 2π, -2π < `y` < 2π.

`fun` can be a function handle, a character vector, or a string (see the Tips on page 1-4065 section).

`ezsurfc(fun,domain)` plots `fun` over the specified `domain`. `domain` can be either a 4-by-1 vector [`xmin`, `xmax`, `ymin`, `ymax`] or a 2-by-1 vector [`min`, `max`] (where `min` < `x` < `max`, `min` < `y` < `max`).

`ezsurfc(funx,funy,funz)` plots the parametric surface `funx(s,t)`, `funy(s,t)`, and `funz(s,t)` over the square: -2π < `s` < 2π, -2π < `t` < 2π.

`ezsurfc(funx,funy,funz,[smin,smax,tmin,tmax])` or `ezsurfc(funx,funy,funz,[min,max])` plots the parametric surface using the specified domain.

`ezsurfc(...,n)` plots *f* over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezsurfc(...,'circ')` plots *f* over a disk centered on the domain.

`ezsurfc(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

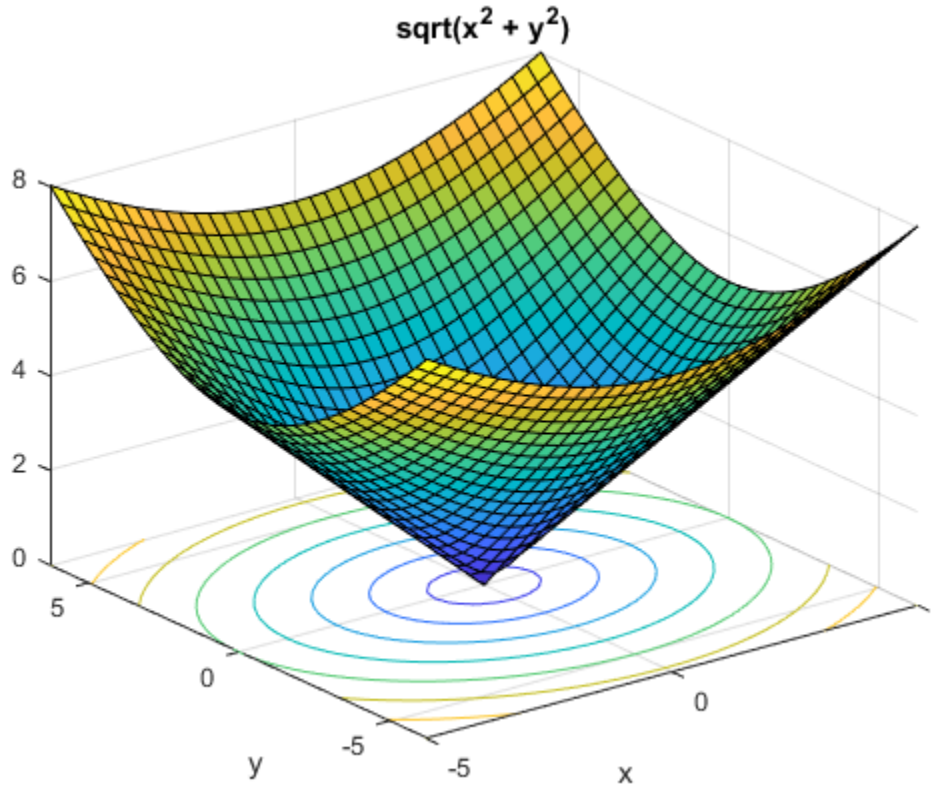
`h = ezsurfc(...)` returns the handles to the graphics objects in `h`.

Examples

Surface and Contour Plot of Mathematical Function

Create a surface/contour plot of the expression $f(x, y) = \sqrt{x^2 + y^2}$ over the domain $-5 < x < 5$ and $-2\pi < y < 2\pi$ with a computational grid size of 35-by-35.

```
ezsurfc('sqrt(x^2 + y^2)',[-5,5,-2*pi,2*pi],35)
```



Tips

`ezsurf` and `ezsurfc` do not accept complex inputs.

Passing the Function as a Character Vector or String

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezsurfc`. For example, the MATLAB syntax for a surface/contour plot of the expression

```
sqrt(x.^2 + y.^2);
```

is written as

```
ezsurfc('sqrt(x^2 + y^2)')
```

That is, `x^2` is interpreted as `x.^2` in the character vector or string you pass to `ezsurfc`.

If the function to be plotted is a function of the variables *u* and *v* (rather than *x* and *y*), then the domain endpoints `umin`, `umax`, `vmin`, and `vmax` are sorted alphabetically. Thus, `ezsurfc('u^2 - v^3',[0,1],[3,6])` plots $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezsurfc`.

```
fh = @(x,y) sqrt(x.^2 + y.^2);
ezsurf(fh)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezsurfc` does not alter the syntax, as in the case with character vector or string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k` in `myfun`:

```
function z = myfun(x,y,k1,k2,k3)
z = x.*(y.^k1)./(x.^k2 + y.^k3);
```

then you can use an anonymous function to specify that parameter:

```
ezsurfc(@(x,y)myfun(x,y,2,2,4))
```

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`fcontour` | `fmesh` | `fplot` | `fplot3` | `fsurf` | `surfc`

Topics

Anonymous Functions

Introduced before R2006a

faceNormals

Class: `TriRep`

(Not recommended) Unit normals to specified triangles

Note `faceNormals(TriRep)` is not recommended. Use `faceNormal(triangulation)` instead.

`TriRep` is not recommended. Use `triangulation` instead.

Syntax

```
FN = faceNormals(TR, TI)
```

Description

`FN = faceNormals(TR, TI)` returns the unit normal vector to each of the specified triangles TI.

Note This query is only applicable to triangular surface meshes.

Input Arguments

`TR`	Triangulation representation.
`TI`	Column vector of indices that index into the triangulation matrix `TR.Triangulation`.

Output Arguments

FN	m-by-3 matrix. `m = length(TI)`, the number of triangles to be queried. Each row `FN(i,:)` represents the unit normal vector to triangle `TI(i)`. If TI is not specified the unit normal information for the entire triangulation is returned, where the normal associated with triangle i is the `i`'th row of `FN`.

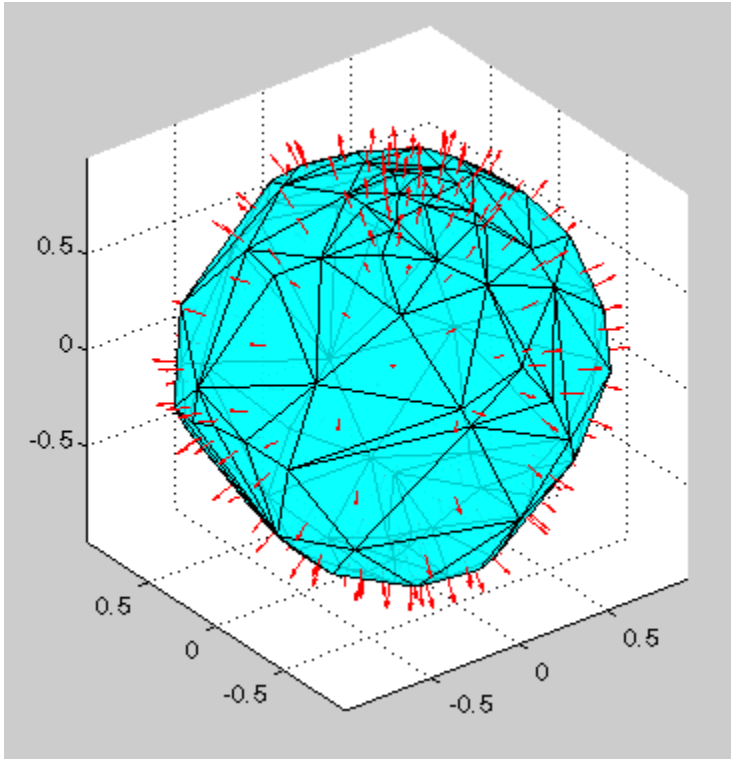
Examples

Triangulate a sample of random points on the surface of a sphere and use the `TriRep` to compute the normal to each triangle:

```
numpts = 100;
thetha = rand(numpts,1)*2*pi;
phi = rand(numpts,1)*pi;
x = cos(thetha).*sin(phi);
y = sin(thetha).*sin(phi);
z = cos(phi);
dt = DelaunayTri(x,y,z);
[tri Xb] = freeBoundary(dt);
tr = TriRep(tri, Xb);
P = incenters(tr);
fn = faceNormals(tr);
trisurf(tri,Xb(:,1),Xb(:,2),Xb(:,3), ...
     'FaceColor', 'cyan', 'faceAlpha', 0.8);
axis equal;
hold on;
```

Display the result using a quiver plot:

```
quiver3(P(:,1),P(:,2),P(:,3), ...
     fn(:,1),fn(:,2),fn(:,3),0.5, 'color','r');
hold off;
```



See Also

`delaunayTriangulation` | `freeBoundary` | `triangulation`

factor

Prime factors

Syntax

```
f = factor(n)
```

Description

`f = factor(n)` returns a row vector containing the prime factors of `n`. Vector `f` is of the same data type as `n`.

Examples

Prime Factors of Double Integer Value

```
f = factor(200)
```

```
f = 1×5

     2     2     2     5     5
```

Multiply the elements of `f` to reproduce the input value.

```
prod(f)
```

```
ans = 200
```

Prime Factors of Unsigned Integer Value

```
n = uint16(138);
f = factor(n)
```

```
f = 1x3 uint16 row vector

    2    3   23
```

Multiply the elements of `f` to reproduce `n`.

```
prod(f)
```

```
ans = 138
```

Input Arguments

`n` — Input value
real, nonnegative integer scalar

Input value, specified as a real, nonnegative integer scalar.

Example: `10`

Example: `int16(64)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The maximum double precision input is `2^33`.
- The maximum single precision input is `2^25`.
- The input `n` cannot have type `int64` or `uint64`.

See Also

`isprime` | `primes`

Introduced before R2006a

factorial

Factorial of input

Syntax

```
f = factorial(n)
```

Description

`f = factorial(n)` returns the product of all positive integers less than or equal to `n`, where `n` is a nonnegative integer value. If `n` is an array, then `f` contains the factorial of each value of `n`. The data type and size of `f` is the same as that of `n`.

The factorial of `n` is commonly written in math notation using the exclamation point character as *n!*. Note that `n!` is not a valid MATLAB syntax for calculating the factorial of `n`.

Examples

10!

```
f = factorial(10)
```

```
f = 3628800
```

22!

```
format long
f = factorial(22)
```

```
f =
     1.124000727777608e+21
```

In this case, `f` is accurate up to 15 digits, `1.12400072777760e+21`, because double-precision numbers are only accurate up to 15 digits.

Reset the output format to the default.

```
format
```

Factorial of Array Elements

```
n = [0 1 2; 3 4 5];
f = factorial(n)
```

```
f = 2×3

     1     1     2
     6    24   120
```

Factorial of Unsigned Integer Values

```
n = uint64([5 10 15 20]);
f = factorial(n)
```

```
f = 1x4 uint64 row vector

                   120               3628800         1307674368000   2432902008176640000
```

Input Arguments

n — Input values
scalar, vector, or array of real, nonnegative integer values

Input values, specified as a scalar, vector, or array of real, nonnegative integers.

Example: `5`

Example: `[0 1 2 3 4]`

Example: `int16([10 15 20])`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Tips

Limitations

- For double-precision inputs, the result is exact when `n` is less than or equal to `21`. Larger values of `n` produce a result that has the correct order of magnitude and is accurate for the first 15 digits. This is because double-precision numbers are only accurate up to 15 digits.
- For single-precision inputs, the result is exact when `n` is less than or equal to `13`. Larger values of `n` produce a result that has the correct order of magnitude and is accurate for the first 8 digits. This is because single-precision numbers are only accurate up to 8 digits.

Saturation

- The table below describes the saturation behavior of each data type when used with the `factorial` function. The values in the last column indicate the saturation point; that is, the first positive integer whose actual factorial is larger than the maximum representable value in the middle column. For `single` and `double`, all values larger than the maximum value are returned as `Inf`. For the integer data types, the saturation value is equal to the maximum value in the middle column.

Data type	Maximum Value	Factorial Saturation Threshold
`double`	`realmax`	`factorial(171)`
`single`	`realmax('single')`	`factorial(single(35))`
`uint64`	2^{64}-1	`factorial(uint64(21))`
`int64`	2^{63}-1	`factorial(int64(21))`
`uint32`	2^{32}-1	`factorial(uint32(13))`

Data type	Maximum Value	Factorial Saturation Threshold
`int32`	2^{31}-1	`factorial(int32(13))`
`uint16`	2^{16}-1	`factorial(uint16(9))`
`int16`	2^{15}-1	`factorial(int16(8))`
`uint8`	2^{8}-1	`factorial(uint8(6))`
`int8`	2^{7}-1	`factorial(int8(6))`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- 64-bit integers are not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- 64-bit integers are not supported.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`prod`

Introduced before R2006a

FactoryGroup

Group of factory settings and factory subgroup objects

Description

A `FactoryGroup` is an object within the factory settings tree. At the top of the tree is the root `FactoryGroup`. Each `FactoryGroup` can contain a collection of other `FactoryGroup` and `FactorySetting` objects. For more information about creating a factory settings tree, see “Create Factory Settings for Toolboxes”.

Creation

Create the `FactoryGroup` root object using the `matlab.settings.createToolboxGroup` function. For example:

```
myToolboxSettings = matlab.settings.FactoryGroup.createToolboxGroup('mytoolbox','Hidden',false);
```

Properties

`Name` — Name of factory group
character vector | string

Name of factory group, specified as a character vector or string.

`Hidden` — Hidden state
`true` (default) | `false`

Hidden state, specified as `true` or `false`.

When set to `true`, the factory group, including all factory groups and factory settings within the group do not display, although they remain accessible.

`ValidationFcn` — Function to validate settings in factory group
function handle

Function to validate factory settings in group, specified as a function handle. When specified, the function is used to validate the values of all factory settings within the group that do not specify their own validation functions. This includes settings in subgroups, as long as the subgroup or settings do not specify their own validation functions.

Object Functions

addGroup	Add new factory settings group
addSetting	Add new factory setting
matlab.settings.FactoryGroup.createToolboxGroup	Create FactoryGroup root object for toolbox

Examples

Create Toolbox Factory Tree

Create the factory settings tree for the toolbox `mytoolbox`.

Create the root factory group `mytoolbox`.

```
myToolboxFactoryTree = matlab.settings.FactoryGroup.createToolboxGroup('mytoolbox', ...
    'Hidden',false);
```

Add the `font` group to the tree and add two settings for font size and font color.

```
toolboxFontGroup = addGroup(myToolboxFactoryTree,'font','Hidden',false);
addSetting(toolboxFontGroup,'FontSize','FactoryValue',11,'Hidden',false, ...
    'ValidationFcn',@matlab.settings.mustBeNumericScalar);
addSetting(toolboxFontGroup,'FontColor','FactoryValue','Black', ...
    'Hidden',false,'ValidationFcn',@matlab.settings.mustBeStringScalar);
```

See Also

`FactorySetting` | `addGroup` | `addSetting` | `matlab.settings.FactoryGroup.createToolboxGroup`

Topics

“Create Factory Settings for Toolboxes”
“Access and Modify Settings”

Introduced in R2019b

FactorySetting

Factory settings object

Description

A `FactorySetting` object represents an individual setting within the factory settings tree.

Creation

To create a factory setting, first, create the `FactoryGroup` root object using the `matlab.settings.createToolboxGroup` function. For example:

```
myToolboxSettings = matlab.settings.FactoryGroup.createToolboxGroup('mytoolbox','Hidden',false);
```

Then, create `FactorySetting` objects within the `FactoryGroup` root object using the `addSetting` function. For example:

```
myFactorySetting = addSetting(myToolboxSettings,'MySetting','Hidden',false,'FactoryValue',10);
```

Properties

`Name` — Name of factory setting
character vector | string

Name of factory setting, specified as a character vector or string.

`Hidden` — Hidden state
`true` (default) | `false`

Hidden state, specified as `true` or `false`.

When set to `true`, the factory setting does not display, although it remains accessible.

`ReadOnly` — Read-only state
`false` (default) | `true`

Read-only state, specified as `true` or `false`. When `true`, the personal or temporary value of the setting cannot be specified.

`FactoryValue` — Factory value of setting
MATLAB data

Factory value of setting, specified as MATLAB data of any type except for handle types. Data containers such as cell arrays, structs, and objects that include handles are also not supported. You must specify either a factory value or a factory value function for the setting, but not both.

`FactoryValueFcn` — Function to set the setting factory value
function handle

Function to set the setting factory value, specified as a function handle. The factory value of the setting is set to the output of the specified function.

The function handle must point to a function on the MATLAB path. Anonymous or nested function handles are not supported.

You can specify either a factory value or a factory value function for the setting, but not both.

`ValidationFcn` — Function to validate setting
function handle

Function to validate setting, specified as a function handle. When specified, the function is used to validate the value of the factory setting.

The function handle must be associated with a function that accepts the potential setting value as an input argument, has no output arguments, and throws an error if the validation fails.

The function handle must point to a function on the MATLAB path. Anonymous or nested function handles are not supported.

Examples

Add New Factory Setting

Create the root factory group for the toolbox `mytoolbox` and then add a new setting to the tree.

Create the root factory group `mytoolbox`.

```
myToolboxFactoryTree = matlab.settings.FactoryGroup.createToolboxGroup('mytoolbox', ...
    'Hidden',false);
```

Add the setting `FontSize` and give it a value. MATLAB returns the new setting as a `FactorySetting` object.

```
fontSizeSetting = addSetting(myToolboxFactoryTree,'FontSize','FactoryValue',11,'Hidden',false);
```

See Also

`FactoryGroup` | `addGroup` | `addSetting` | `matlab.settings.FactoryGroup.createToolboxGroup`

Topics

"Create Factory Settings for Toolboxes"
"Access and Modify Settings"

Introduced in R2019b

false

Logical 0 (false)

Syntax

```
false
F = false(n)
F = false(sz)
F = false(sz1,...,szN)
F = false( ___ ,'like',p)
```

Description

`false` is shorthand for the logical value `0`.

`F = false(n)` is an `n`-by-`n` array of logical zeros.

`F = false(sz)` is an array of logical zeros where the size vector, `sz`, defines `size(F)`. For example, `false([2 3])` returns a 2-by-3 array of logical zeros.

`F = false(sz1,...,szN)` is a `sz1`-by-`...`-by-`szN` array of logical zeros where `sz1,...,szN` indicates the size of each dimension. For example, `false(2,3)` returns a 2-by-3 array of logical zeros.

`F = false( ___ ,'like',p)` returns an array of logical zeros of the same sparsity as the logical variable `p` using any of the previous size syntaxes.

Examples

Generate Square Matrix of Logical Zeros

Use `false` to generate a 3-by-3 square matrix of logical zeros.

```
A = false(3)
```

```
A = 3x3 logical array

   0   0   0
   0   0   0
   0   0   0
```

```
class(A)
```

```
ans =
'logical'
```

The result is of class `logical`.

Generate Array of Logical Zeros with Arbitrary Dimensions

Use `false` to generate a 3-by-2-by-2 array of logical zeros.

```
false(3,2,2)
```

```
ans = 3x2x2 logical array
ans(:,:,1) =

   0   0
   0   0
   0   0


ans(:,:,2) =

   0   0
   0   0
   0   0
```

Alternatively, use a size vector to specify the size of the matrix.

```
false([3 2 2])
```

```
ans = 3x2x2 logical array
ans(:,:,1) =

   0   0
```

```
   0   0
   0   0


ans(:,:,2) =

   0   0
   0   0
   0   0
```

Note that specifying multiple vector inputs returns an error.

Execute Logic Statement

`false` along with `true` can be used to execute logic statements.

Test the logical statement

`~(A and B) = (~A) or (~B)`

for `A = false` and `B = true`.

```
~(false & true) == (~false) | (~true)
```

```
ans = logical
   1
```

The result is logical 1 (true), since the logical statements on both sides of the equation are equivalent. This logical statement is an instance of De Morgan's Law.

Generate Logical Array of Selected Sparsity

Generate a logical array of the same data type and sparsity as the selected array.

```
A = logical(sparse(5,3));
whos A
```

```
  Name      Size            Bytes  Class     Attributes

  A         5x3                41  logical   sparse
```

```
F = false(4,'like',A);
whos F
```

```
  Name      Size            Bytes  Class     Attributes

  F         4x4                49  logical   sparse
```

The output array `F` has the same `sparse` attribute as the specified array `A`.

Input Arguments

n — Size of square matrix

integer

Size of square matrix, specified as an integer. `n` sets the output array size to `n`-by-`n`. For example, `false(3)` returns a 3-by-3 array of logical zeros.

- If `n` is `0`, then `F` is an empty matrix.
- If `n` is negative, then it is treated as `0`.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

sz — Size vector

row vector of integers

Size vector, specified as a row vector of integers. For example, `false([2 3])` returns a 2-by-3 array of logical zeros.

- If the size of any dimension is `0`, then `F` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- If any trailing dimensions greater than `2` have a size of `1`, then the output, `F`, does not include those dimensions.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

sz1,...,szN — Size inputs

comma-separated list of integers

Size inputs, specified by a comma-separated list of integers. For example, `false(2,3)` returns a 2-by-3 array of logical zeros.

- If the size of any dimension is `0`, then `F` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- If any trailing dimensions greater than `2` have a size of `1`, then the output, `F`, does not include those dimensions.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

p — Prototype
logical variable

Prototype, specified as a logical variable.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

F — Output of logical zeros
scalar | vector | matrix | N-D array

Output of logical zeros, returned as a scalar, vector, matrix, or N-D array.

Data Types: `logical`

Tips

- `false(n)` is much faster and more memory efficient than `logical(zeros(n))`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Dimensions must be real, nonnegative, integers.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `false` in the Parallel Computing Toolbox documentation.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `false` in the Parallel Computing Toolbox documentation.

See Also

`logical` | `true`

Topics

"Class Support for Array-Creation Functions"

Introduced before R2006a

fclose

Close one or all open files

Syntax

```
fclose(fileID)
fclose('all')
status = fclose( ___ )
```

Description

`fclose(fileID)` closes an open file.

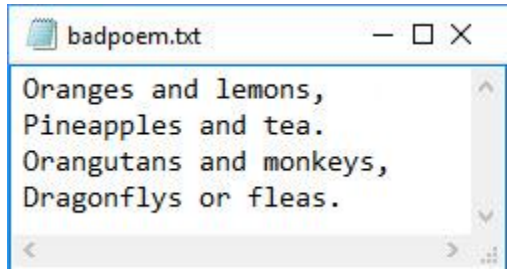
`fclose('all')` closes all open files.

`status = fclose( ___ )` returns a `status` of `0` when the close operation is successful. Otherwise, it returns `-1`. You can use this syntax with any of the input arguments of the previous syntaxes.

Examples

Open and Close File

Open the `badpoem.txt` file, read the first line, and then close the file.



Use `fopen` to open the file. This function assigns a unique file id to use for reading and writing to the file.

```
fid = fopen('badpoem.txt')
```

```
fid = 3
```

Read the first line from the file using `fgetl`.

```
fgetl(fid)
```

```
ans =
'Oranges and lemons,'
```

Close the file.

```
fclose(fid);
```

Input Arguments

`fileID` — File identifier
integer

File identifier of an open file, specified as an integer. Before closing a file with `fclose`, you must use `fopen` to open the file and obtain its `fileID`.

Data Types: `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`feof` | `ferror` | `fopen` | `fprintf` | `fread` | `frewind` | `fscanf` | `fseek` | `ftell` | `fwrite`

Topics

"Import Text Data Files with Low-Level I/O"

Introduced before R2006a

fclose (serial)

(Not recommended) Disconnect serial port object from device

Note This `serial` object function is not recommended. Use `serialport` object functions instead. See “Compatibility Considerations”.

Syntax

```
fclose(obj)
```

Description

`fclose(obj)` disconnects `obj` from the device, where `obj` is a serial port object or an array of serial port objects.

Examples

This example creates the serial port object `s` on a Windows platform, connects `s` to the device, writes and reads text data, and then disconnects `s` from the device using `fclose`.

```
s = serial('COM1');
fopen(s)
fprintf(s, '*IDN?')
idn = fscanf(s);
fclose(s)
```

At this point, the device is available to be connected to a serial port object. If you no longer need `s`, you should remove from memory with the `delete` function, and remove it from the workspace with the `clear` command.

Tips

If `obj` was successfully disconnected, then the `Status` property is configured to `closed` and the `RecordStatus` property is configured to `off`. You can reconnect `obj` to the device using the `fopen` function.

An error is returned if you issue `fclose` while data is being written asynchronously. In this case, you should abort the write operation with the `stopasync` function, or wait for the write operation to complete.

Compatibility Considerations

`serial` object interface is not recommended

Not recommended starting in R2019b

Use of this function with a `serial` object is not recommended. To access a serial port device, use a `serialport` object with its functions and properties instead.

See “Transition Your Code to serialport Interface” for more information about using the recommended functionality.

See Also

Functions

`fopen` | `stopasync`

Introduced before R2006a

fcontour

Plot contours

Syntax

```
fcontour(f)
fcontour(f,xyinterval)

fcontour( ___ ,LineSpec)
fcontour( ___ ,Name,Value)
fcontour(ax, ___ )
fc = fcontour( ___ )
```

Description

`fcontour(f)` plots the contour lines of the function `z = f(x,y)` for constant levels of `z` over the default interval `[-5 5]` for `x` and `y`.

`fcontour(f,xyinterval)` plots over the specified interval. To use the same interval for both `x` and `y`, specify `xyinterval` as a two-element vector of the form `[min max]`. To use different intervals, specify a four-element vector of the form `[xmin xmax ymin ymax]`.

`fcontour( ___ ,LineSpec)` sets the line style and color for the contour lines. For example, `'-r'` specifies red lines. Use this option after any of the previous input argument combinations.

`fcontour( ___ ,Name,Value)` specifies line properties using one or more name-value pair arguments.

`fcontour(ax, ___ )` plots into the axes specified by `ax` instead of the current axes.

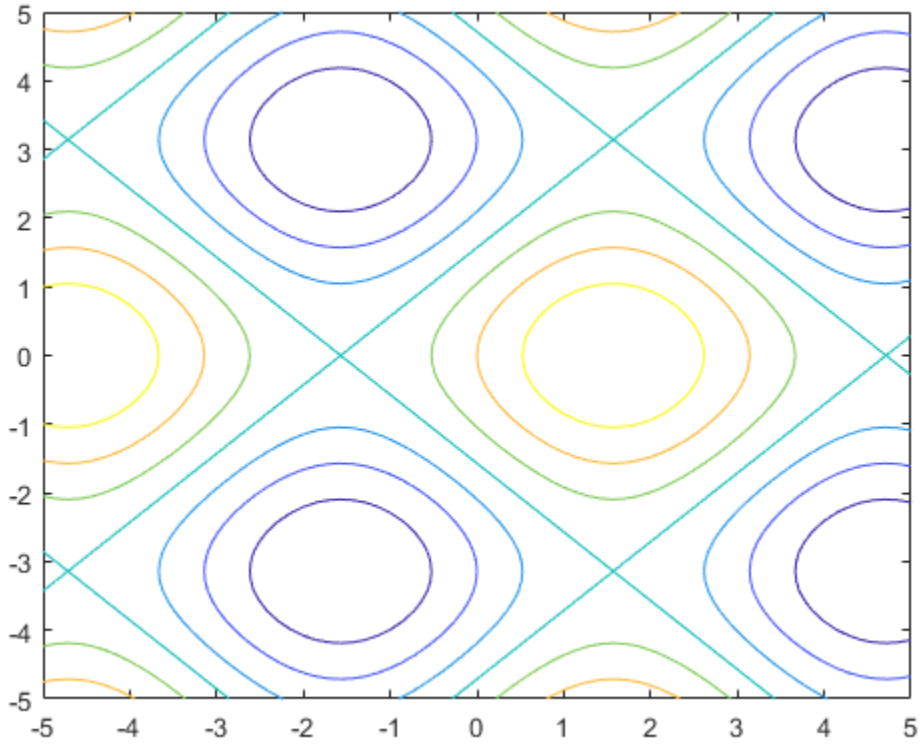
`fc = fcontour( ___ )` returns a `FunctionContour` object. Use `fc` to query and modify properties of a specific `FunctionContour` object. For a list of properties, see FunctionContour.

Examples

Plot Contours of Function

Plot the contours of $f(x, y) = \sin(x) + \cos(y)$ over the default interval of $-5 < x < 5$ and $-5 < y < 5$.

```
f = @(x,y) sin(x) + cos(y);
fcontour(f)
```



Specify Plotting Interval and Plot Piecewise Contour Plot

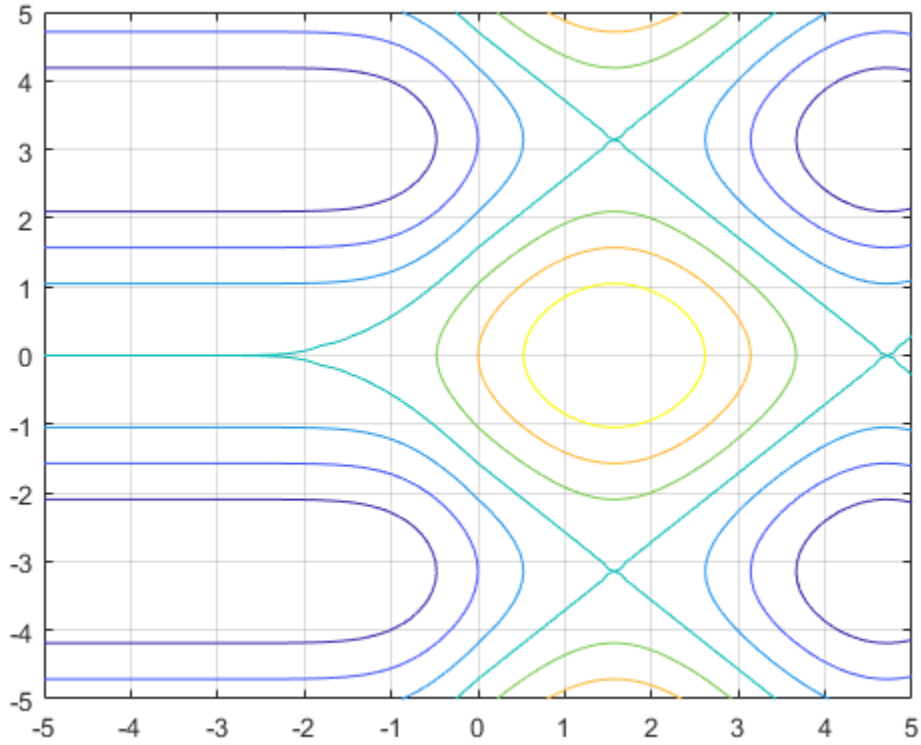
Specify the plotting interval as the second argument of `fcontour`. When you plot multiple inputs over different intervals in the same axes, the axis limits adjust to display all the data. This behavior lets you plot piecewise inputs.

Plot the piecewise input

$$\begin{array}{ll} \mathrm{erf}(x) + \cos(y) & -5 < x < 0 \\ \sin(x) + \cos(y) & 0 < x < 5 \end{array}$$

over $-5 < y < 5$.

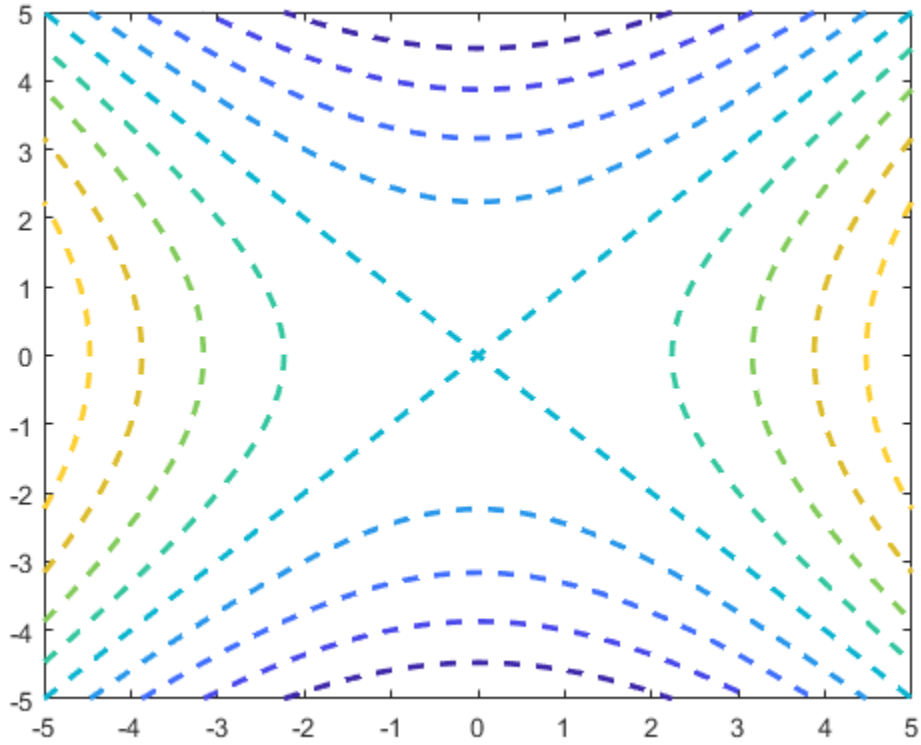
```
fcontour(@(x,y) erf(x) + cos(y),[-5 0 -5 5])
hold on
fcontour(@(x,y) sin(x) + cos(y),[0 5 -5 5])
hold off
grid on
```



Change Line Style and Width

Plot the contours of $x^2 - y^2$ as dashed lines with a line width of 2.

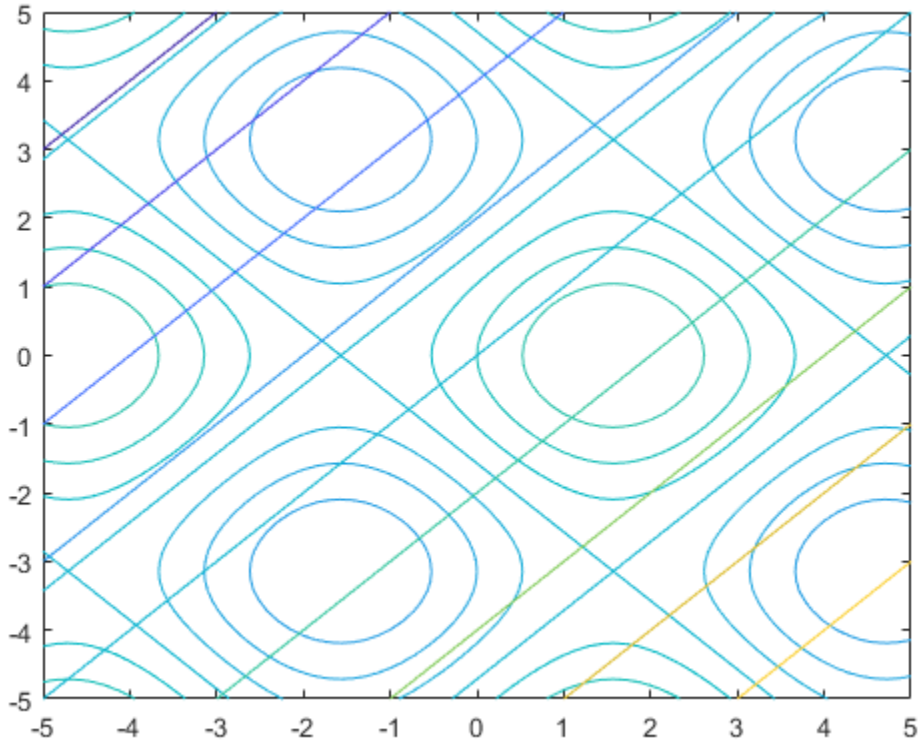
```
f = @(x,y) x.^2 - y.^2;
fcontour(f,'--','LineWidth',2)
```



Plot Multiple Contour Plots

Plot $\sin(x) + \cos(y)$ and $x - y$ on the same axes by using `hold on`.

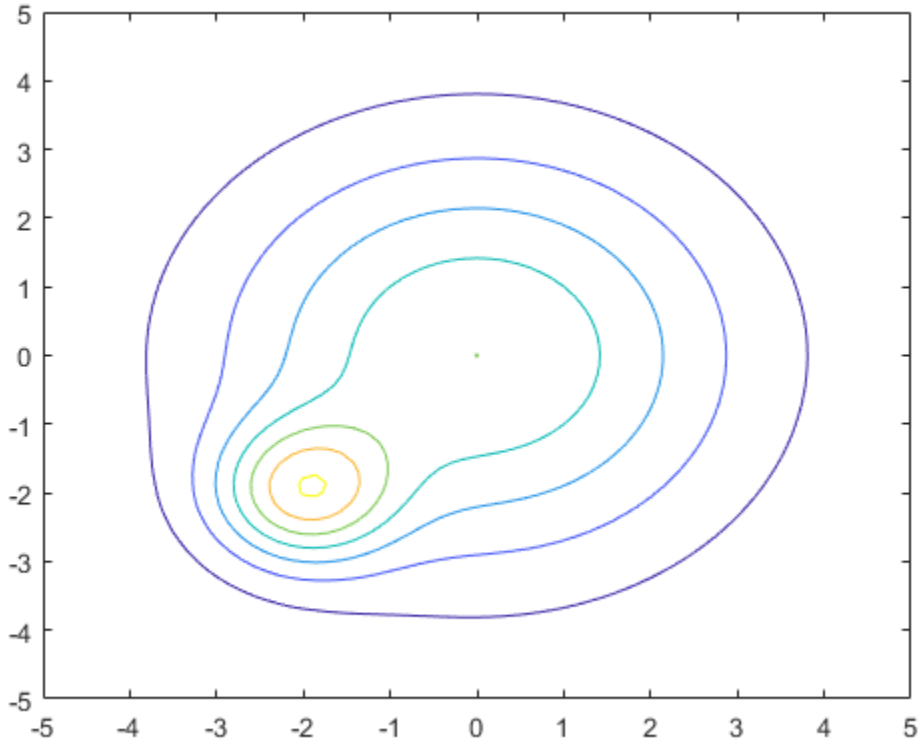
```
fcontour(@(x,y) sin(x)+cos(y))
hold on
fcontour(@(x,y) x-y)
hold off
```



Modify Contour Plot After Creation

Plot the contours of $e^{-(x/3)^2-(y/3)^2}+e^{-(x+2)^2-(y+2)^2}$. Assign the function contour object to a variable.

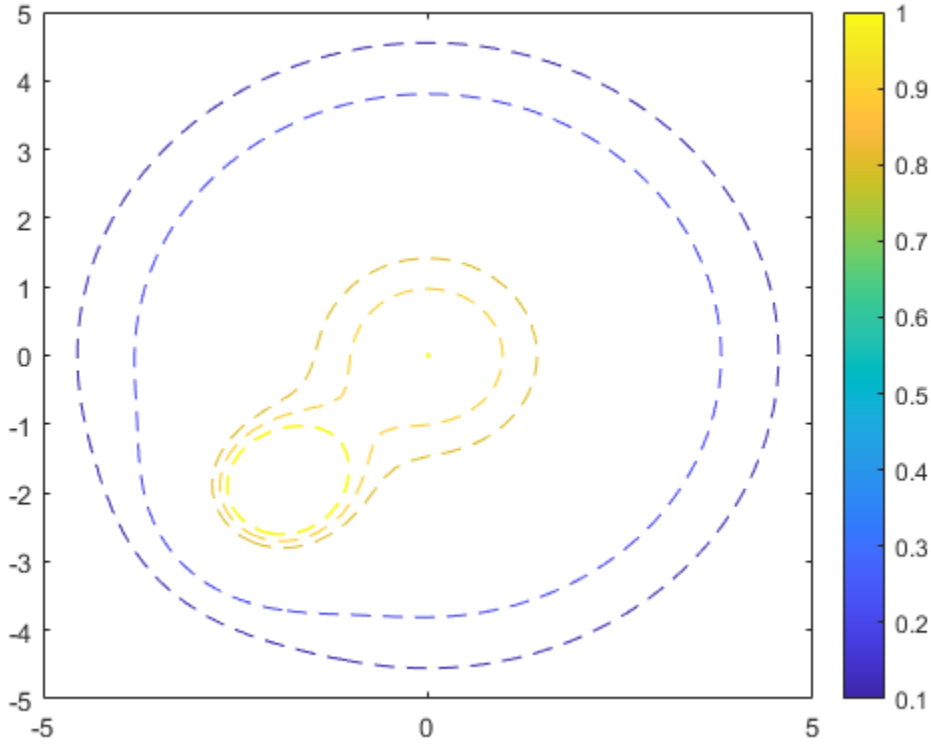
```
f = @(x,y) exp(-(x/3).^2-(y/3).^2) + exp(-(x+2).^2-(y+2).^2);
fc = fcontour(f)
```




```
fc =
  FunctionContour with properties:

     Function: @(x,y)exp(-(x/3).^2-(y/3).^2)+exp(-(x+2).^2-(y+2).^2)
    LineColor: 'flat'
    LineStyle: '-'
    LineWidth: 0.5000
         Fill: 'off'
    LevelList: [0.2000 0.4000 0.6000 0.8000 1 1.2000 1.4000]

  Show all properties
```

Change the line width to `1` and the line style to a dashed line by using dot notation to set properties of the function contour object. Show contours close to `0` and `1` by setting the `LevelList` property. Add a colorbar.

```
fc.LineWidth = 1;
fc.LineStyle = '--';
fc.LevelList = [1 0.9 0.8 0.2 0.1];
colorbar
```

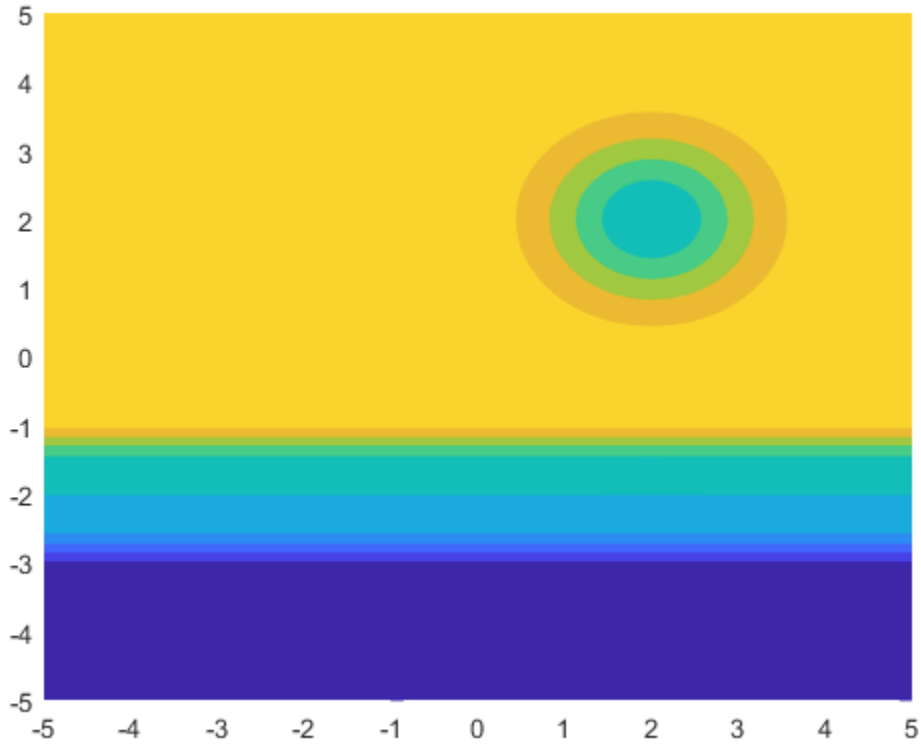


Fill Area Between Contours

Create a plot that looks like a sunset by filling the area between the contours of

$$\mathrm{erf}((y+2)^3) - e^{(-0.65((x-2)^2+(y-2)^2))}.$$

```
f = @(x,y) erf((y+2).^3) - exp(-0.65*((x-2).^2+(y-2).^2));
fcontour(f,'Fill','on');
```

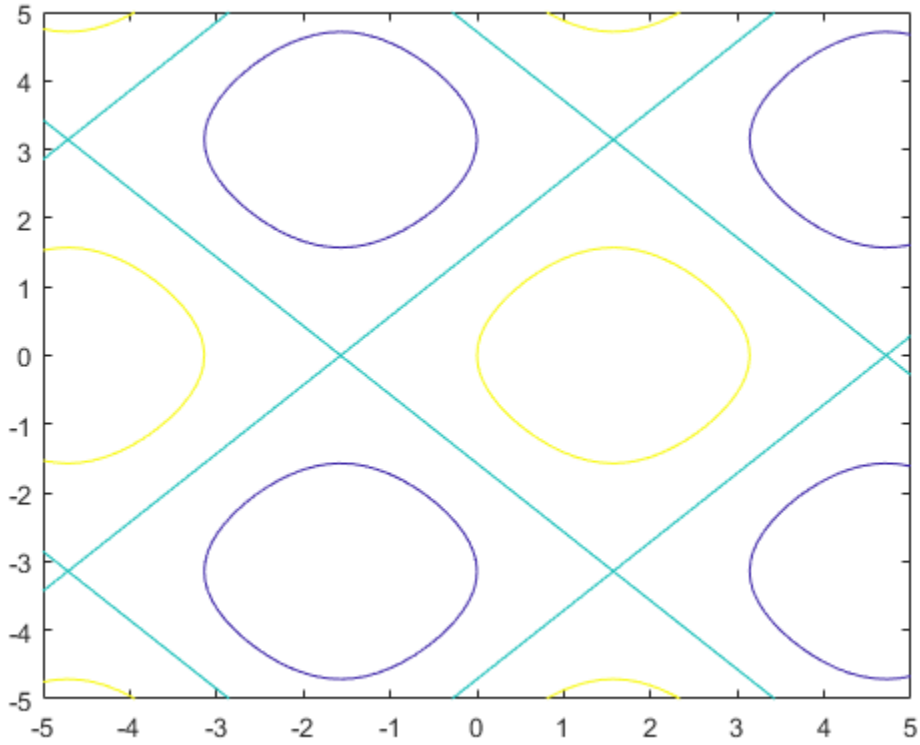


If you want interpolated shading instead, use the `fsurf` function and set its `'EdgeColor'` option to `'none'` followed by the command `view(0,90)`.

Specify Levels for Contour Lines

Set the values at which `fcontour` draws contours by using the `'LevelList'` option.

```
f = @(x,y) sin(x) + cos(y);
fcontour(f,'LevelList',[-1 0 1])
```



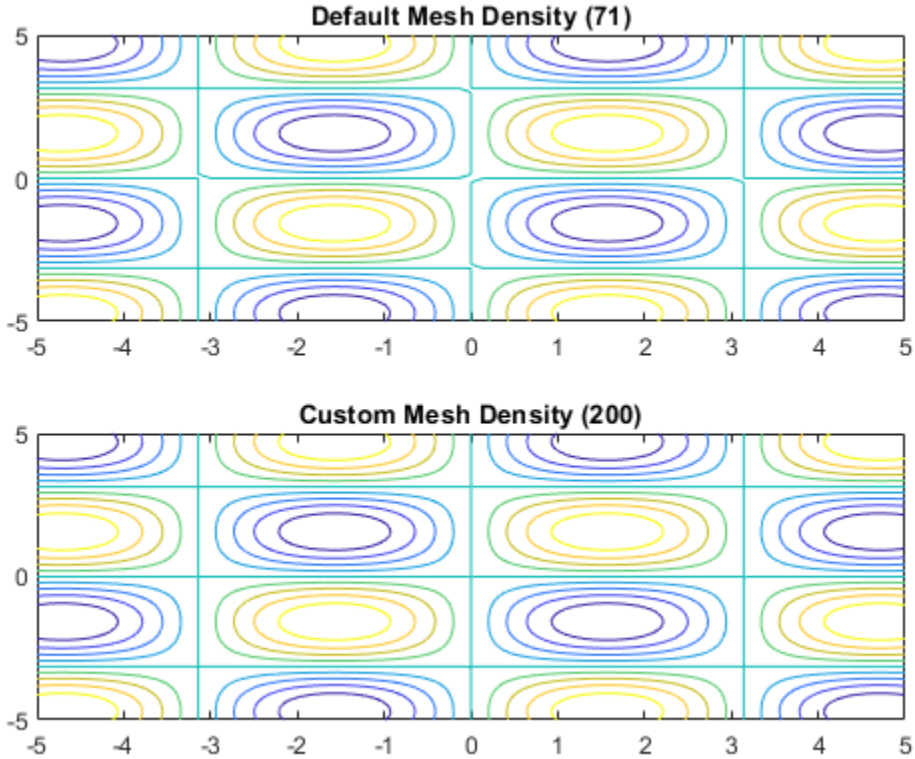
Control Resolution of Contour Lines

Control the resolution of contour lines by using the `'MeshDensity'` option. Increasing `'MeshDensity'` can make smoother, more accurate plots, while decreasing it can increase plotting speed.

Divide a figure into two using `subplot`. In the first subplot, plot the contours of $\sin(x)\sin(y)$. The corners of the squares do not meet. To fix this issue, increase

`'MeshDensity'` to `200` in the second subplot. The corners now meet, showing that by increasing `'MeshDensity'` you increase the resolution.

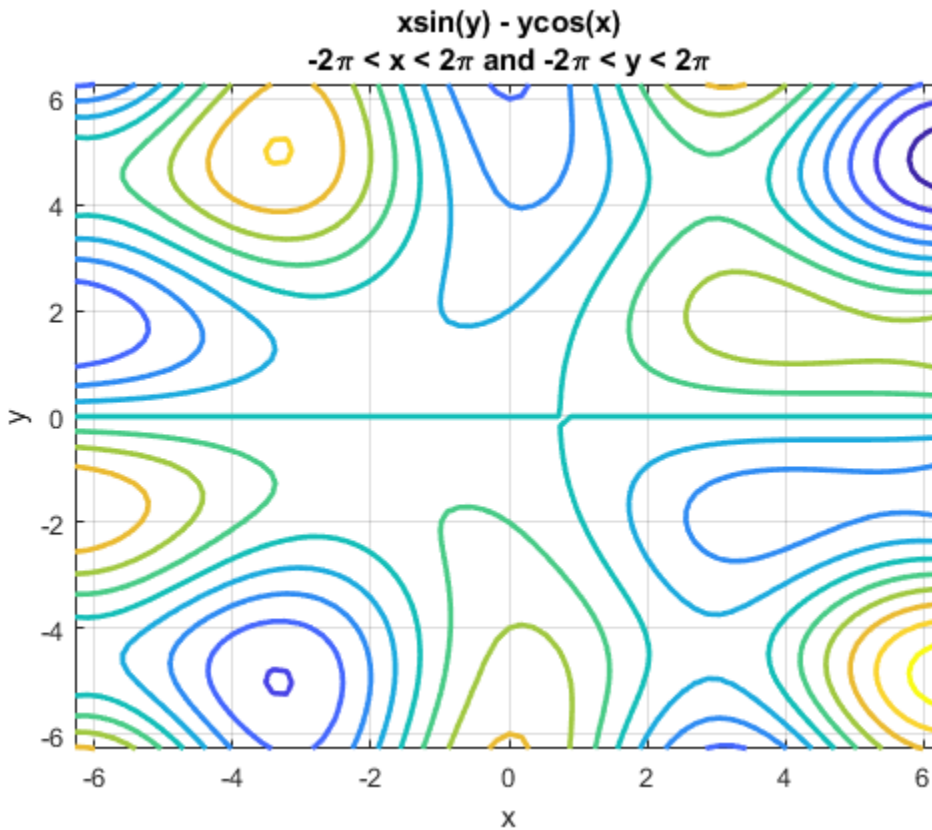
```
f = @(x,y) sin(x).*sin(y);
subplot(2,1,1)
fcontour(f)
title('Default Mesh Density (71)')

subplot(2,1,2)
fcontour(f,'MeshDensity',200)
title('Custom Mesh Density (200)')
```



Add Title and Axis Labels and Format Ticks

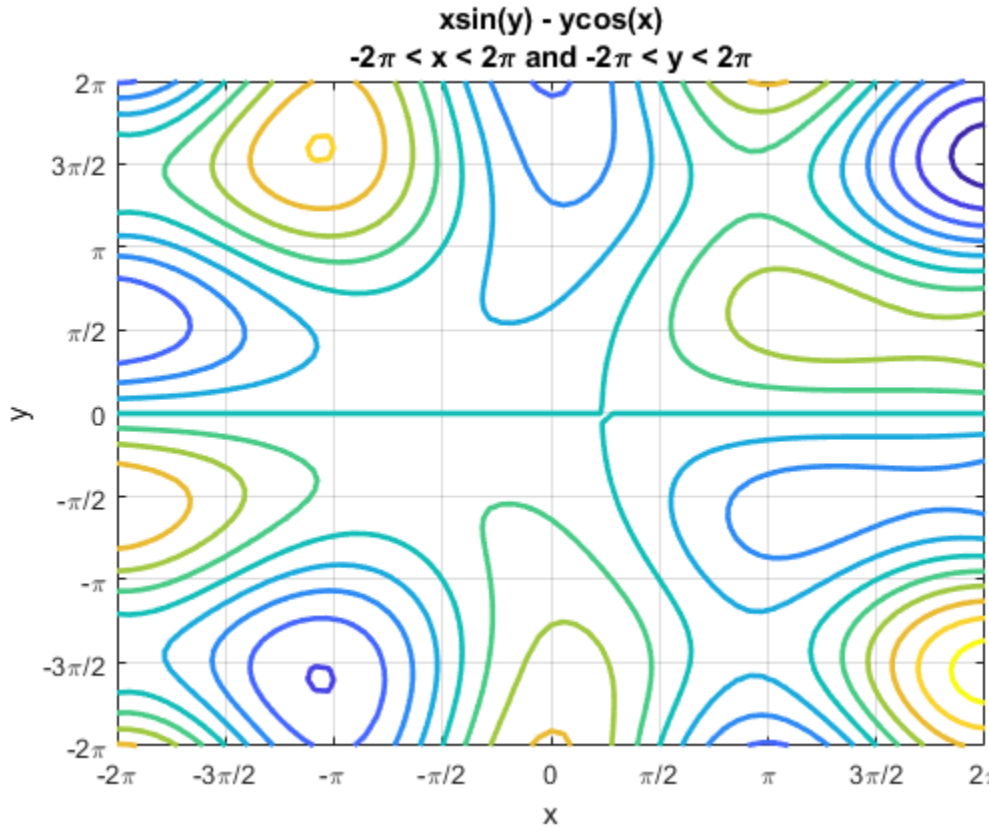
Plot $x\sin(y) - y\cos(x)$. Display the grid lines, add a title, and add axis labels.

```
fcontour(@(x,y) x.*sin(y) - y.*cos(x), [-2*pi 2*pi], 'LineWidth', 2);
grid on
title({'xsin(y) - ycos(x)','-2\pi < x < 2\pi and -2\pi < y < 2\pi'})
xlabel('x')
ylabel('y')
```



Set the x-axis tick values and associated labels by setting the `XTickLabel` and `XTick` properties of the axes object. Access the axes object using `gca`. Similarly, set the y-axis tick values and associated labels.

```
ax = gca;
ax.XTick = ax.XLim(1):pi/2:ax.XLim(2);
ax.XTickLabel = {'-2\pi','-3\pi/2','-\pi','-\pi/2','0','\pi/2','\pi','3\pi/2','2\pi'};
ax.YTick = ax.YLim(1):pi/2:ax.YLim(2);
ax.YTickLabel = {'-2\pi','-3\pi/2','-\pi','-\pi/2','0','\pi/2','\pi','3\pi/2','2\pi'};
```



Input Arguments

f — Function to plot
function handle

Function to plot, specified as a function handle to a named or anonymous function.

Specify a function of the form `z = f(x,y)`. The function must accept two matrix input arguments and return a matrix output argument of the same size. Use array operators instead of matrix operators for the best performance. For example, use `.*` (`times`) instead of `*` (`mtimes`).

Example: `f = @(x,y) sin(x) + cos(y);`

`xyinterval` — Plotting interval for `x` and `y`

`[-5 5 -5 5]` (default) | vector of form `[min max]` | vector of form `[xmin xmax ymin ymax]`

Plotting interval for `x` and `y`, specified in one of these forms:

- Vector of form `[min max]` — Use the interval `[min max]` for both `x` and `y`.
- Vector of form `[xmin xmax ymin ymax]` — Use the interval `[xmin xmax]` for `x` and `[ymin ymax]` for `y`.

`ax` — Axes object

axes object

Axes object. If you do not specify an axes object, then the `fcontour` uses the current axes.

`LineSpec` — Line style and color

character vector | string

Line style and color, specified as a character vector or string containing a line style specifier, a color specifier, or both.

Example: `'--r'` specifies red dashed lines

These two tables list the line style and color options.

Line Style Specifier	Description
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Color Specifier	Description
y	yellow
m	magenta
c	cyan
r	red
g	green
b	blue
w	white
k	black

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'MeshDensity',30`

The properties listed here are only a subset. For a full list, see FunctionContour.

`MeshDensity` — Number of evaluation points per direction
71 (default) | number

Number of evaluation points per direction, specified as a number. The default is `71`. Because `fcontour` uses adaptive evaluation, the actual number of evaluation points is greater.

Example: `30`

`Fill` — Fill between contour lines
`'off'` (default) | `'on'`

Fill between contour lines, specified as one of these values:

- `'off'` — Do not fill the spaces between contour lines with a color.
- `'on'` — Fill the spaces between contour lines with color.

`LevelList` — Contour levels
vector of z values

Contour levels, specified as a vector of z values. By default, the `fcontour` function chooses values that span the range of values in the `ZData` property.

Setting this property sets the associated mode property to manual.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`LevelStep` — Spacing between contour lines
`0` (default) | scalar numeric value

Spacing between contour lines, specified as a scalar numeric value. For example, specify a value of `2` to draw contour lines at increments of 2. By default, `LevelStep` is determined by using the `ZData` values.

Setting this property sets the associated mode property to `'manual'`.

Example: `3.4`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`LineColor` — Color of contour lines
`'flat'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of contour lines, specified as `'flat'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. To use a different color for each contour line, specify `'flat'`. The color is determined by the contour value of the line, the colormap, and the scaling of data values into the colormap. For more information on color scaling, see `caxis`.

To use the same color for all the contour lines, specify an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to

F. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`LineWidth` — Line width

`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

Output Arguments

`fc` — One or more `FunctionContour` objects
scalar | vector

One or more `FunctionContour` objects, returned as a scalar or a vector. You can use these objects to query and modify the properties of a specific contour plot. For a list of properties, see FunctionContour.

See Also

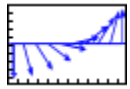
Functions
`fmesh` | `fplot` | `fplot3` | `fsurf` | `title`

Properties
FunctionContour

Introduced in R2016a

feather

Plot velocity vectors



Syntax

```
feather(U,V)
feather(Z)
feather(...,LineSpec)
feather(axes_handle,...)
h = feather(...)
```

Description

A feather plot displays vectors emanating from equally spaced points along a horizontal axis. You express the vector components relative to the origin of the respective vector.

`feather(U,V)` displays the vectors specified by `U` and `V`, where `U` contains the *x* components as relative coordinates, and `V` contains the *y* components as relative coordinates.

`feather(Z)` displays the vectors specified by the complex numbers in `Z`. This is equivalent to `feather(real(Z),imag(Z))`.

`feather(...,LineSpec)` draws a feather plot using the line type, marker symbol, and color specified by `LineSpec`.

`feather(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = feather(...)` returns the handles to line objects in `h`.

Examples

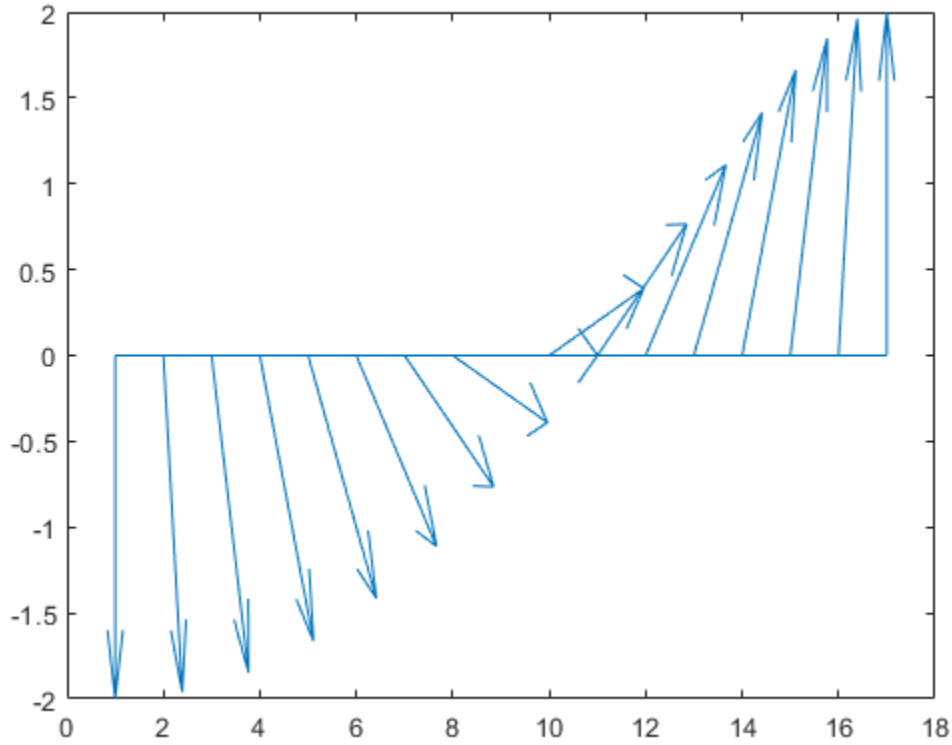
Create Feather Plot

Define `theta` as values between -2π and 2π. Define `r` as a vector the same size as `theta`.

```
theta = -pi/2:pi/16:pi/2;
r = 2*ones(size(theta));
```

Create a feather plot showing the direction of `theta`. Since `feather` uses Cartesian coordinates, convert `theta` and `r` to Cartesian coordinates using `pol2cart`.

```
[u,v] = pol2cart(theta,r);
feather(u,v)
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`LineSpec` | `compass` | `rose`

Introduced before R2006a

featureEdges

Class: `TriRep`

(Not recommended) Sharp edges of surface triangulation

Note `featureEdges(TriRep)` is not recommended. Use `featureEdges(triangulation)` instead.

`TriRep` is not recommended. Use `triangulation` instead.

Syntax

```
FE = featureEdges(TR, filterangle)
```

Description

`FE = featureEdges(TR, filterangle)` returns an edge matrix `FE`. This method is typically used to extract the sharp edges in the surface mesh for the purpose of display. Edges that are shared by only one triangle and edges that are shared by more than two triangles are considered to be feature edges by default.

Note This query is only applicable to triangular surface meshes.

Input Arguments

`TR`	Triangulation representation.
`filterangle`	The threshold angle in radians. Must be in the range $(0, \pi)$. `featureEdges` will return adjacent triangles that have a dihedral angle that deviates from π by an angle greater than `filterangle`.

Output Arguments

FE	Edges of the triangulation. FE is of size m-by-2 where m is the number of computed feature edges in the mesh. The vertices of the edges index into the array of points representing the vertex coordinates, TR.X.

Examples

Create a surface triangulation:

```
x = [0 0 0 0 0 3 3 3 3 3 3 6 6 6 6 6 9 9 9 9 9 9]';
y = [0 2 4 6 8 0 1 3 5 7 8 0 2 4 6 8 0 1 3 5 7 8]';
dt = DelaunayTri(x,y);
tri = dt(:,:);
```

Elevate the 2-D mesh to create a surface:

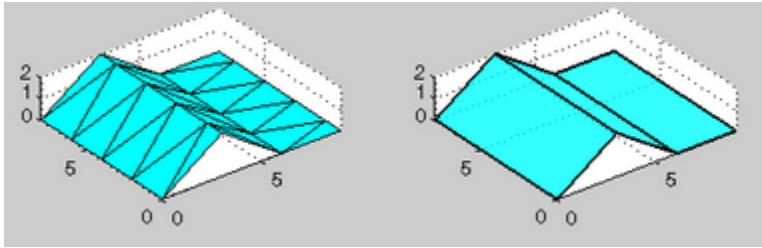
```
z = [0 0 0 0 0 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0]';
subplot(1,2,1);
trisurf(tri,x,y,z, 'FaceColor', 'cyan');
axis equal;
% TRISURF display of surface mesh
% showing mesh edges
```

Compute the feature edges using a filter angle of pi/6:

```
tr = TriRep(tri, x,y,z);
fe = featureEdges(tr,pi/6)';
subplot(1,2,2);
trisurf(tr, 'FaceColor', 'cyan', 'EdgeColor','none', ...
     'FaceAlpha', 0.8); axis equal;
```

Add the feature edges to the plot:

```
hold on;
plot3(x(fe), y(fe), z(fe), 'k', 'LineWidth',1.5);
hold off;
% TRISURF display of surface mesh
% suppressing mesh edges
% and showing feature edges
```



See Also

delaunayTriangulation | edges | triangulation

feof

Test for end of file

Syntax

```
status = feof(fileID)
```

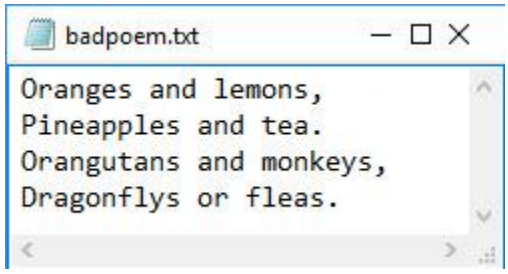
Description

`status = feof(fileID)` returns the status of the end-of-file indicator. The `feof` function returns a `1` if a previous operation set the end-of-file indicator for the specified file. Otherwise, `feof` returns a `0`.

Examples

Read to End of File

From this `badpoem.txt` file, read one line at a time to the end of the file.



Use `fopen` to open the file. This function assigns a unique file id to use for reading and writing to the file.

```
fid = fopen('badpoem.txt');
```

Read and display one line at a time until you reach the end of the file.

```
while ~feof(fid)
    tline = fgetl(fid);
    disp(tline)
end
```

```
Oranges and lemons,
Pineapples and tea.
Orangutans and monkeys,
Dragonflys or fleas.
```

Close the file.

```
fclose(fid);
```

Input Arguments

`fileID` — File identifier
integer

File identifier of an open file, specified as an integer. Before testing for the end-of-file status, you must use `fopen` to open the file and obtain a valid file identifier `fileID`.

Data Types: `double`

Tips

- Opening an empty file does *not* set the end-of-file indicator. Read operations, and other operations like `fseek` and `frewind`, move the file position indicator.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`fclose` | `ferror` | `fopen` | `frewind` | `fseek` | `ftell`

Topics

"Import Text Data Files with Low-Level I/O"

Introduced before R2006a

ferror

File I/O error information

Syntax

```
message = ferror(fileID)
[message,errnum] = ferror(fileID)
[message,errnum] = ferror(fileID,'clear')
```

Description

`message = ferror(fileID)` returns the error message for the most recent file I/O operation on the specified file.

`[message,errnum] = ferror(fileID)` returns the error number that is associated with the error message.

`[message,errnum] = ferror(fileID,'clear')` clears the error indicator for the specified file. Specify the literal `clear` as a character vector or a string scalar. When you clear the error indicator, succeeding calls to `ferror` behave as if the most recent I/O operation is successful.

Examples

Get Most Recent Error Message

Return the details for the most recent file I/O error on a file that you specify.

Open the file to read.

```
fid = fopen('outages.csv','r');
```

Set the read position to `-5` bytes from the beginning of the file. Since no data exists before the beginning of the file, `fseek` returns `-1` indicating that the operation was unsuccessful.

```
status = fseek(fid,-5,'bof')
```

```
status = -1
```

Get the details on the most recent error message on the file using its file identifier `fid`.

```
error = ferror(fid)
```

```
error =
'Offset is bad - before beginning-of-file.'
```

Close the file.

```
fclose(fid);
```

Input Arguments

`fileID` — File identifier
integer

File identifier of an open file, specified as an integer. `fileID` can be one of these values:

- `N` — Input or output from a file with a valid file identifier generated by the `fopen` function. `N` is equal to or greater than `3`.
- `0` — Input from a standard input device.
- `1` — Output to a standard output device.
- `2` — Standard error.

Data Types: `double`

Output Arguments

`message` — Error message
character vector

Error message, returned as character vector containing the error message associated with the most recent I/O operation. If the most recent I/O operation is successful or if you clear it manually, then `message` is an empty character vector.

`errnum` — Error number
integer

Error number, returned as an integer code associated with the error from most recent file I/O operation. If the most recent file I/O operation is successful, then `errnum` is `0`. A nonzero `errnum` indicates an error:

- Negative values correspond to MATLAB error messages.
- Positive values correspond to C library error messages for your system.

See Also

`fclose` | `feof` | `fopen` | `fprintf` | `fread` | `fscanf` | `fseek` | `ftell` | `fwrite`

Introduced before R2006a

feval

Evaluate function

Syntax

```
[y1,...,yN] = feval(fun,x1,...,xM)
```

Description

`[y1,...,yN] = feval(fun,x1,...,xM)` evaluates a function using its name or its handle, and using the input arguments `x1,...,xM`.

The `feval` function follows the same scoping and precedence rules as calling a function handle directly. For more information, see “Create Function Handle”.

Examples

Evaluate Function with Function Name as Character Vector

Round the value of `pi` to the nearest integer using the name of the function.

```
fun = 'round';
x1 = pi;
y = feval(fun,x1)
```

```
y = 3
```

Round the value of `pi` to two digits to the right of the decimal point.

```
x2 = 2;
y = feval(fun,x1,x2)
```

```
y = 3.1400
```

Input Arguments

`fun` — Function to evaluate
function name | function handle

Function to evaluate, specified as a function name or a handle to a function. The function accepts `M` input arguments, and returns `N` output arguments. To specify `fun` as a function name, do not include path information.

Invoking `feval` with a function handle is equivalent to invoking the function handle directly.

Example: `fun = 'cos'`

Example: `fun = @sin`

`x1,...,xM` — Inputs to evaluated function
depends on function

Inputs to the evaluated function. The types of the inputs depend on the function, `fun`.

Output Arguments

`y1,...,yN` — Outputs from evaluated function
depends on function

Outputs from evaluated function. The types of the outputs depend on the function, `fun`.

Tips

- If you have a function handle, it is not necessary to use `feval` because you can invoke the function handle directly. The results of the following statements are equivalent.

  ```
  fh = @eig;
  [V,D] = fh(A)
  ```

  ```
  [V,D] = feval(@eig,A)
  ```
- To evaluate a nested or local function using `feval`, use a function handle instead of the function name. For more information, see “Call Local Functions Using Function Handles”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The code generator automatically interprets `feval` as an extrinsic function. For more information, see “Extrinsic Functions” (MATLAB Coder).
- The code generator does not support the use of `feval` to call local functions.
- The code generator does not support the use of `feval` to call functions that are located in a private folder.

See Also

`assignin` | `builtin` | `eval` | `evalin` | `functions`

Topics

“Create Function Handle”

Introduced before R2006a

fewerbins

Package: `matlab.graphics.chart.primitive`

Decrease number of histogram bins

Syntax

```
N = fewerbins(h)
N = fewerbins(h,direction)
```

Description

`N = fewerbins(h)` decreases the number of bins in histogram `h` by 10% (rounded down to the nearest integer) and returns the new number of bins.

For bivariate histograms, this decreases the bin count in both the *x* and *y* directions.

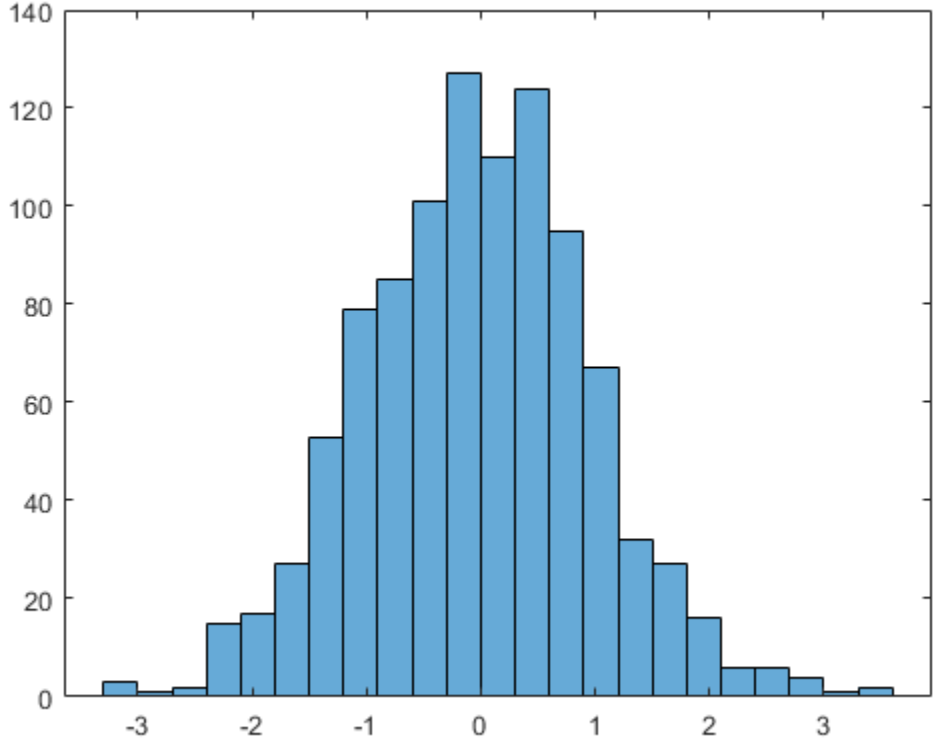
`N = fewerbins(h,direction)`, where `h` must be a `histogram2` object, only decreases the number of bins in the dimension specified by `direction`. The `direction` option can be `'x'`, `'y'`, or `'both'`. The default value is `'both'`.

Examples

Decrease Number of Histogram Bins

Plot a histogram of 1,000 random numbers and return a handle to the histogram object.

```
x = randn(1000,1);
h = histogram(x)
```

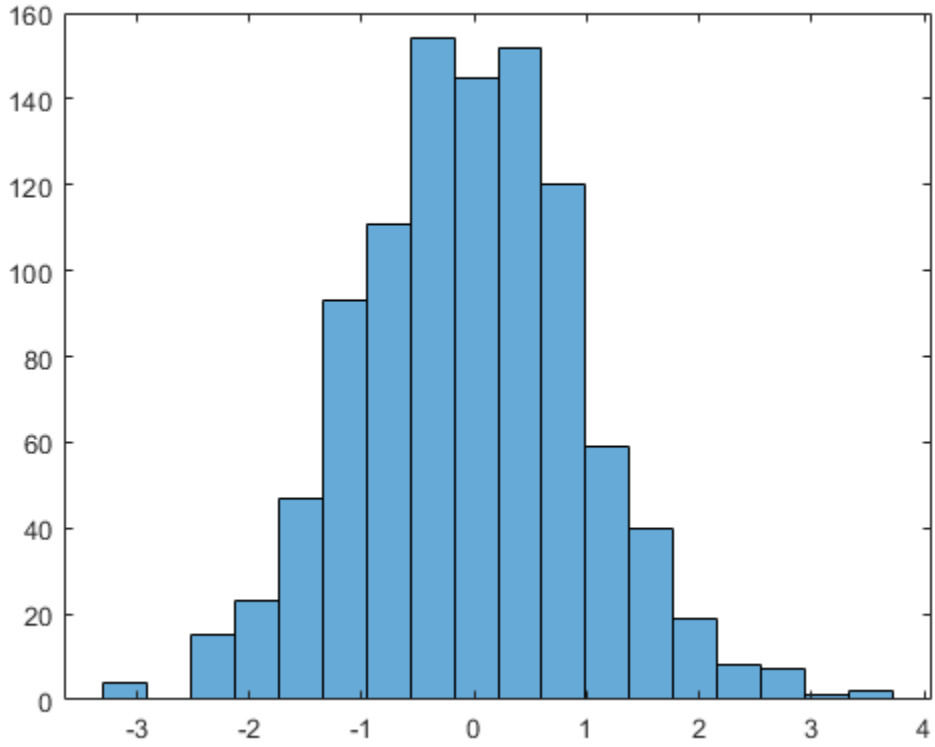



```
h =
  Histogram with properties:

            Data: [1000x1 double]
          Values: [1x23 double]
         NumBins: 23
        BinEdges: [1x24 double]
        BinWidth: 0.3000
       BinLimits: [-3.3000 3.6000]
   Normalization: 'count'
       FaceColor: 'auto'
       EdgeColor: [0 0 0]
```

```
Show all properties
```

Use `fewerbins` to decrease the number of bins in the histogram.

```
fewerbins(h);
fewerbins(h)
```




```
ans = 18
```

Input Arguments

h — Input histogram
`histogram` object | `histogram2` object

Input histogram, specified as a `histogram` or `histogram2` object.

`h` cannot be a categorical histogram.

`direction` — Direction to decrease number of bins
`'both'` (default) | `'x'` | `'y'`

Direction to decrease the number of bins, specified as `'x'`, `'y'`, or `'both'`. Specify `'x'` or `'y'` to only decrease the number of bins in that direction while leaving the number of bins in the other direction constant.

Output Arguments

`N` — Number of bins
scalar | vector

Number of bins, returned as a scalar or vector. `N` is the new number of bins for the histogram after increase. For bivariate histogram plots, `N` is a two-element vector, `[nx ny]`.

See Also

`histcounts` | `histcounts2` | `histogram` | `histogram2` | `morebins`

Introduced in R2014b

Feval (COM)

Execute MATLAB function in Automation server

Syntax

IDL Method Signature

```
HRESULT Feval([in] BSTR functionname, [in] long nargout,
[out] VARIANT* result, [in, optional] VARIANT arg1, arg2, ...)
```

Microsoft Visual Basic Client

```
Feval(String functionname, long numout,
arg1, arg2, ...) As Object
```

MATLAB Client

```
result = Feval(h,'functionName',numout,arg1,arg2,...)
```

Description

`result = Feval(h,'functionName',numout,arg1,arg2,...)` executes MATLAB function `functionName` in the Automation server attached to `h`. The function name is case-sensitive. If `functionName` does not have input arguments, consider calling `Execute` instead.

COM functions are available on Microsoft Windows systems only.

Indicate the number of outputs returned by the function in a `1`-by-`1` `double` array, `numout`. The server returns output from the function in the cell array, `result`.

You can specify as many as 32 input arguments to be passed to the function. These arguments follow `numout` in the `Feval` argument list. The following table shows ways to pass an argument.

Passing Mechanism	Description
Pass the value itself	To pass any numeric or character value, specify the value in the `Feval` argument list: `a = Feval(h,'sin',1,-pi:0.01:pi);`

Pass a client variable	To pass an argument assigned to a variable in the client, specify the variable name alone:
	
	`x = -pi:0.01:pi;`
	`a = Feval(h,'sin',1,x);`

Reference a server variable	To reference a variable defined in the server, specify the variable name followed by an equals (=) sign:
	
	`PutWorkspaceData(h,'x','base',-pi:0.01:pi);`
	`a = Feval(h,'sin',1,'x=');`
	
	MATLAB does not reassign the server variable.

Examples

Call `strcat` from Visual Basic .NET Client

This example shows how to pass string arguments to the MATLAB `strcat` command using `Feval`.

Create a Visual Basic .NET application with the following code.

```
type fevalPassingArguments.vb
```

```
Dim Matlab As Object
Dim out As Object
out = Nothing
Matlab = CreateObject("matlab.application")
Matlab.Feval("strcat",1,out,"hello"," world")
Dim clistr As String
clistr = " world"
Matlab.Feval("strcat",1,out,"hello",clistr)
Matlab.PutCharArray("srvstr","base"," world")
Matlab.Feval("strcat",1,out,"hello","srvstr=")
```

Defining Feval Return Values

This example shows how to return the `filepath`, `name`, and `ext` arguments from the `fileparts` function.

`Feval` returns data from the evaluated function in a cell array. The cell array has one row for every return value. You control the number of return values using the `Feval` `numout` argument.

Create a Visual Basic .NET client with the following code.

```
type fevalDefiningFevalReturnValues.vb
```

```
Dim Matlab As Object
Dim out As Object
Matlab = CreateObject("matlab.application")
Matlab.Feval("fileparts",3,out,"d:\work\ConsoleApp.cpp")
```

Modified Server Variables

This example shows how to return a modified server variable.

Create a matrix, `A`, in the server.

Reshape `A`.MATLAB interprets `A` in the expression `'A='` as a server variable name.

The `reshape` function does not modify variable `A`. `A` is unchanged.

To get the result of the `reshape` function, use the `numout` argument to assign the value to `C`.

Create a Visual Basic .NET application with the following code.

```
type fevalModifiedServerVariables.vb
```

```
Dim Matlab As Object
Dim rows As Double
Dim cols As Double
Dim out As Object
out = Nothing
```

```
Dim data(7) As Double
For i = 0 To 7
    data(i) = i * 15
Next i
Matlab = CreateObject("matlab.application")
Matlab.PutWorkspaceData("A", "base", data)
rows = 4
cols = 2
Matlab.Feval("reshape", 1, out, "A=", rows, cols)
```

Tips

- To display the output from `Feval` in the client window, assign a return value.

See Also

`Execute` | `GetCharArray` | `GetFullMatrix` | `PutCharArray` | `PutFullMatrix`

Topics

"Conversion of COM Types to MATLAB Types"
"Conversion of MATLAB Types to COM Types"

Introduced before R2006a

fft

Fast Fourier transform

Syntax

```
Y = fft(X)
Y = fft(X,n)
Y = fft(X,n,dim)
```

Description

`Y = fft(X)` computes the discrete Fourier transform on page 1-4151 (DFT) of `X` using a fast Fourier transform (FFT) algorithm.

- If `X` is a vector, then `fft(X)` returns the Fourier transform of the vector.
- If `X` is a matrix, then `fft(X)` treats the columns of `X` as vectors and returns the Fourier transform of each column.
- If `X` is a multidimensional array, then `fft(X)` treats the values along the first array dimension whose size does not equal 1 as vectors and returns the Fourier transform of each vector.

`Y = fft(X,n)` returns the `n`-point DFT. If no value is specified, `Y` is the same size as `X`.

- If `X` is a vector and the length of `X` is less than `n`, then `X` is padded with trailing zeros to length `n`.
- If `X` is a vector and the length of `X` is greater than `n`, then `X` is truncated to length `n`.
- If `X` is a matrix, then each column is treated as in the vector case.
- If `X` is a multidimensional array, then the first array dimension whose size does not equal 1 is treated as in the vector case.

`Y = fft(X,n,dim)` returns the Fourier transform along the dimension `dim`. For example, if `X` is a matrix, then `fft(X,n,2)` returns the n-point Fourier transform of each row.

Examples

Noisy Signal

Use Fourier transforms to find the frequency components of a signal buried in noise.

Specify the parameters of a signal with a sampling frequency of 1 kHz and a signal duration of 1.5 seconds.

```
Fs = 1000;            % Sampling frequency
T = 1/Fs;             % Sampling period
L = 1500;             % Length of signal
t = (0:L-1)*T;        % Time vector
```

Form a signal containing a 50 Hz sinusoid of amplitude 0.7 and a 120 Hz sinusoid of amplitude 1.

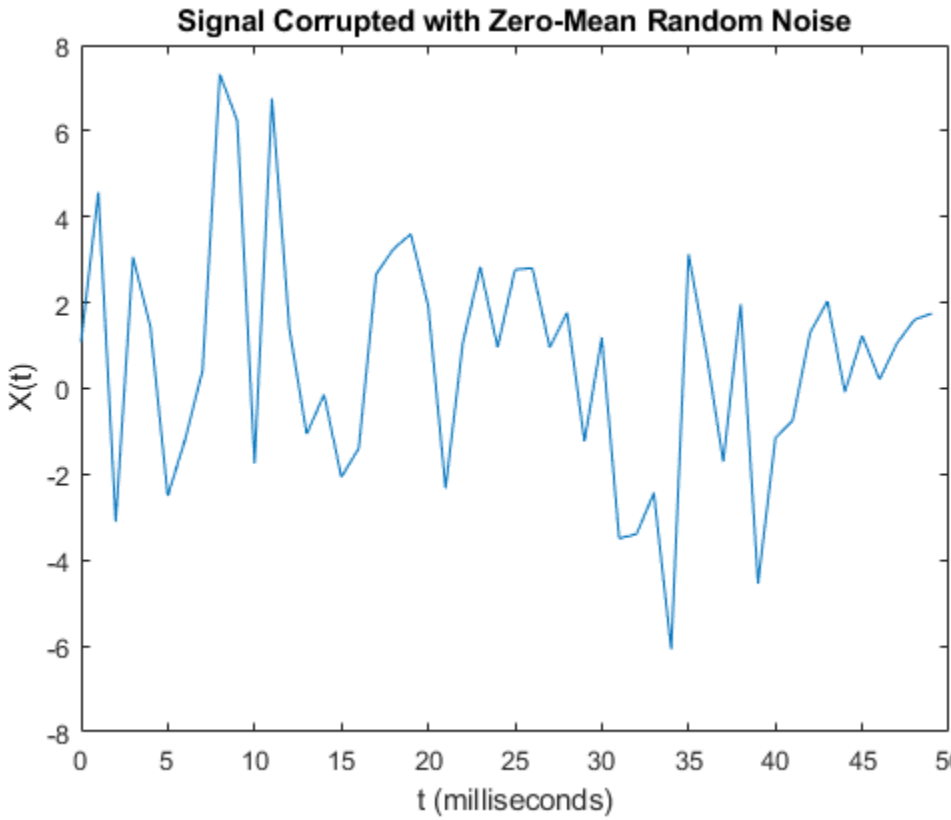
```
S = 0.7*sin(2*pi*50*t) + sin(2*pi*120*t);
```

Corrupt the signal with zero-mean white noise with a variance of 4.

```
X = S + 2*randn(size(t));
```

Plot the noisy signal in the time domain. It is difficult to identify the frequency components by looking at the signal `X(t)`.

```
plot(1000*t(1:50),X(1:50))
title('Signal Corrupted with Zero-Mean Random Noise')
xlabel('t (milliseconds)')
ylabel('X(t)')
```



Compute the Fourier transform of the signal.

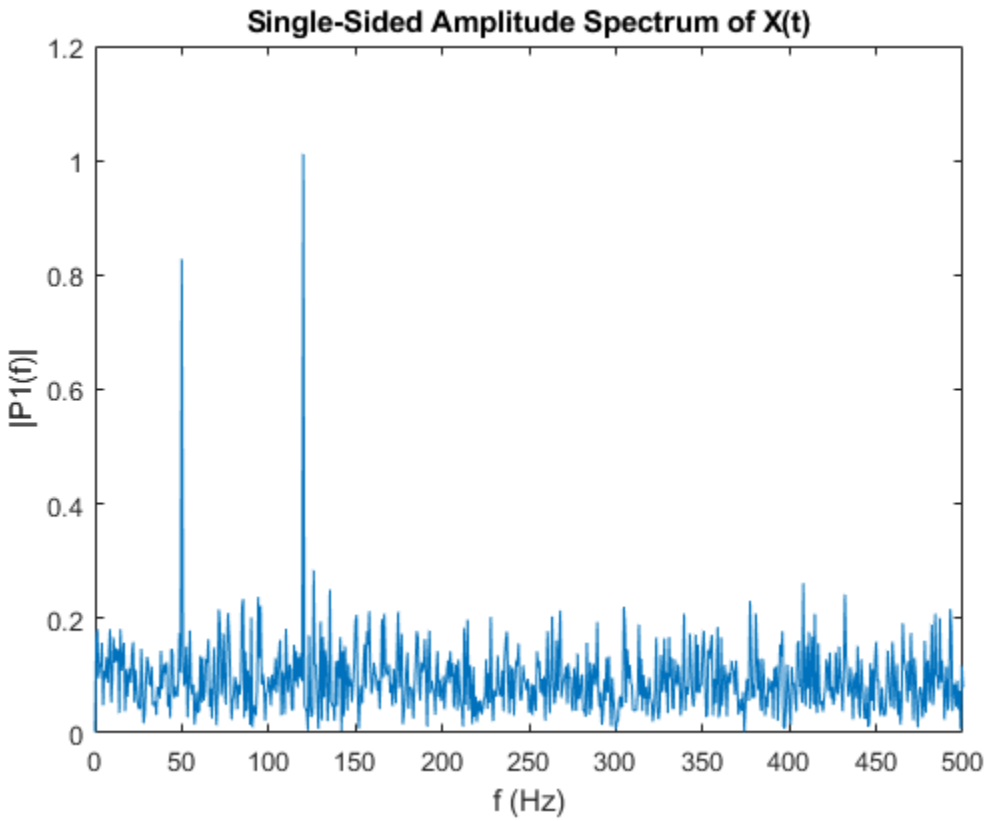
```
Y = fft(X);
```

Compute the two-sided spectrum P2. Then compute the single-sided spectrum P1 based on P2 and the even-valued signal length L.

```
P2 = abs(Y/L);
P1 = P2(1:L/2+1);
P1(2:end-1) = 2*P1(2:end-1);
```

Define the frequency domain f and plot the single-sided amplitude spectrum P1. The amplitudes are not exactly at 0.7 and 1, as expected, because of the added noise. On average, longer signals produce better frequency approximations.

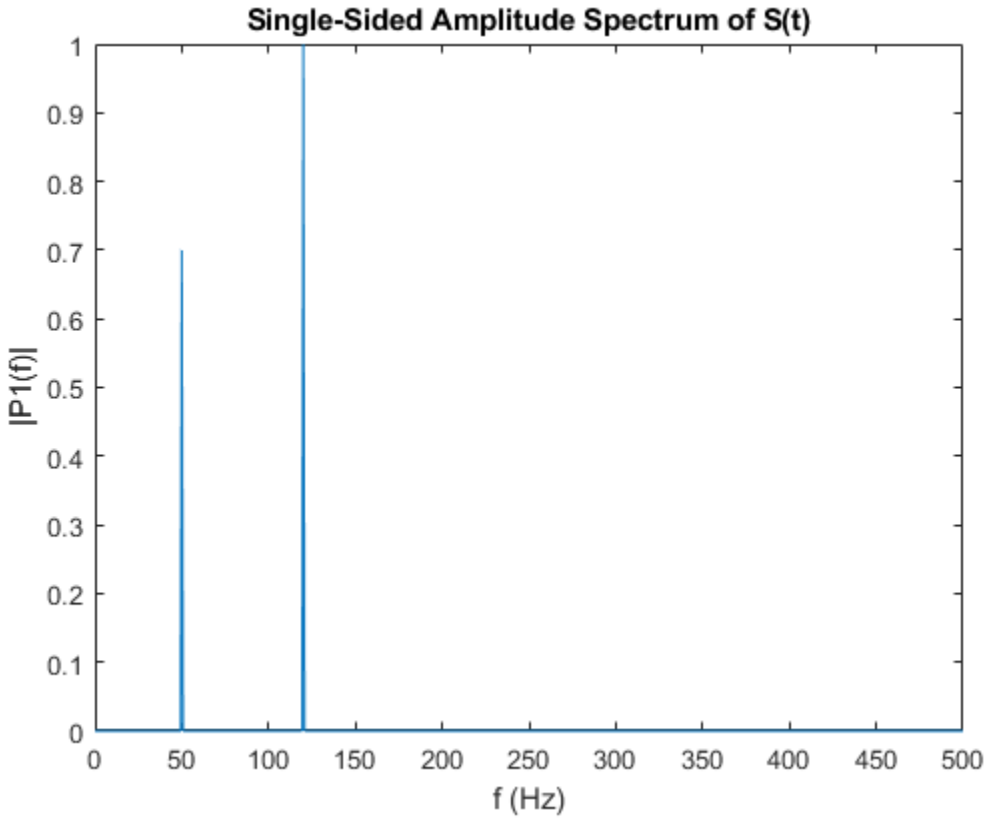
```
f = Fs*(0:(L/2))/L;
plot(f,P1)
title('Single-Sided Amplitude Spectrum of X(t)')
xlabel('f (Hz)')
ylabel('|P1(f)|')
```



Now, take the Fourier transform of the original, uncorrupted signal and retrieve the exact amplitudes, 0.7 and 1.0.

```
Y = fft(S);
P2 = abs(Y/L);
P1 = P2(1:L/2+1);
P1(2:end-1) = 2*P1(2:end-1);
```

```
plot(f,P1)
title('Single-Sided Amplitude Spectrum of S(t)')
xlabel('f (Hz)')
ylabel('|P1(f)|')
```



Gaussian Pulse

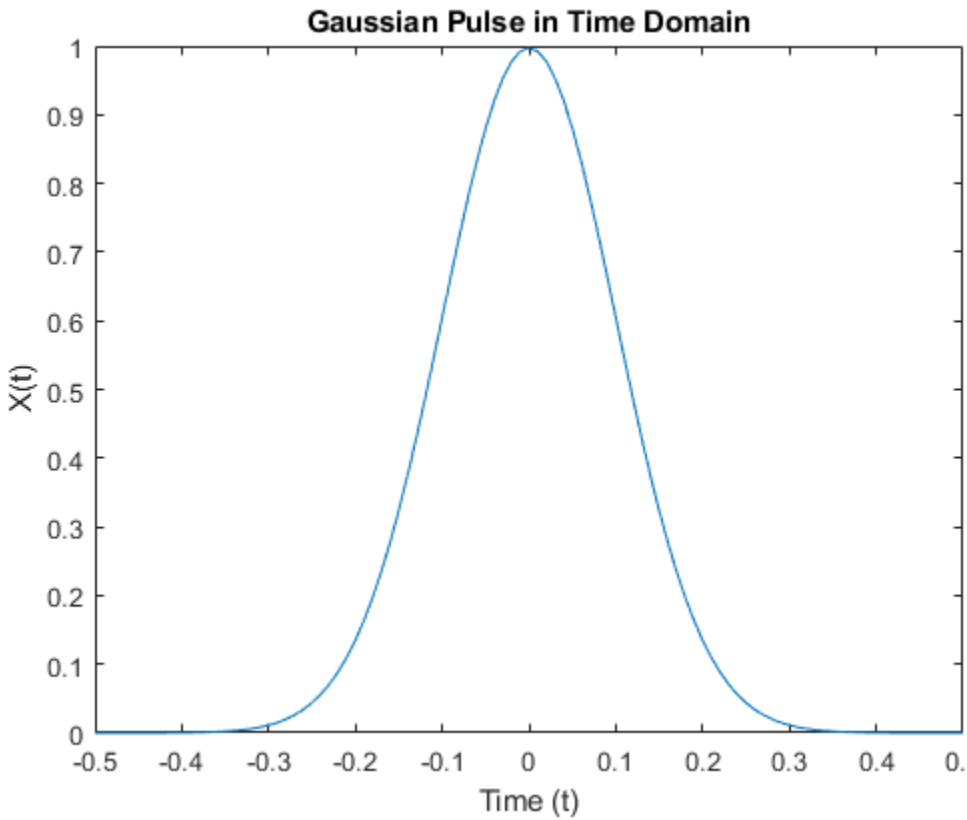
Convert a Gaussian pulse from the time domain to the frequency domain.

Define signal parameters and a Gaussian pulse, X.

```
Fs = 100;            % Sampling frequency
t = -0.5:1/Fs:0.5;   % Time vector
L = length(t);       % Signal length

X = 1/(4*sqrt(2*pi*0.01))*(exp(-t.^2/(2*0.01)));
```

Plot the pulse in the time domain.

```
plot(t,X)
title('Gaussian Pulse in Time Domain')
xlabel('Time (t)')
ylabel('X(t)')
```



To use the `fft` function to convert the signal to the frequency domain, first identify a new input length that is the next power of 2 from the original signal length. This will pad the signal X with trailing zeros in order to improve the performance of `fft`.

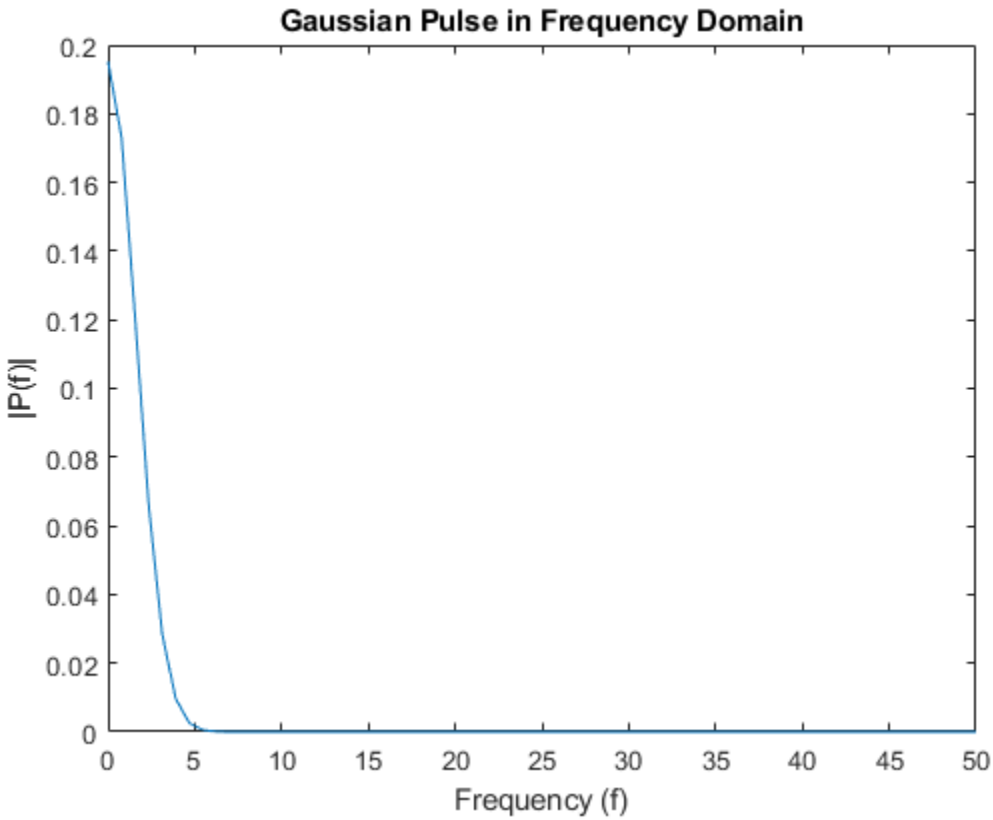
```
n = 2^nextpow2(L);
```

Convert the Gaussian pulse to the frequency domain.

```
Y = fft(X,n);
```

Define the frequency domain and plot the unique frequencies.

```
f = Fs*(0:(n/2))/n;
P = abs(Y/n);

plot(f,P(1:n/2+1))
title('Gaussian Pulse in Frequency Domain')
xlabel('Frequency (f)')
ylabel('|P(f)|')
```



Cosine Waves

Compare cosine waves in the time domain and the frequency domain.

Specify the parameters of a signal with a sampling frequency of 1kHz and a signal duration of 1 second.

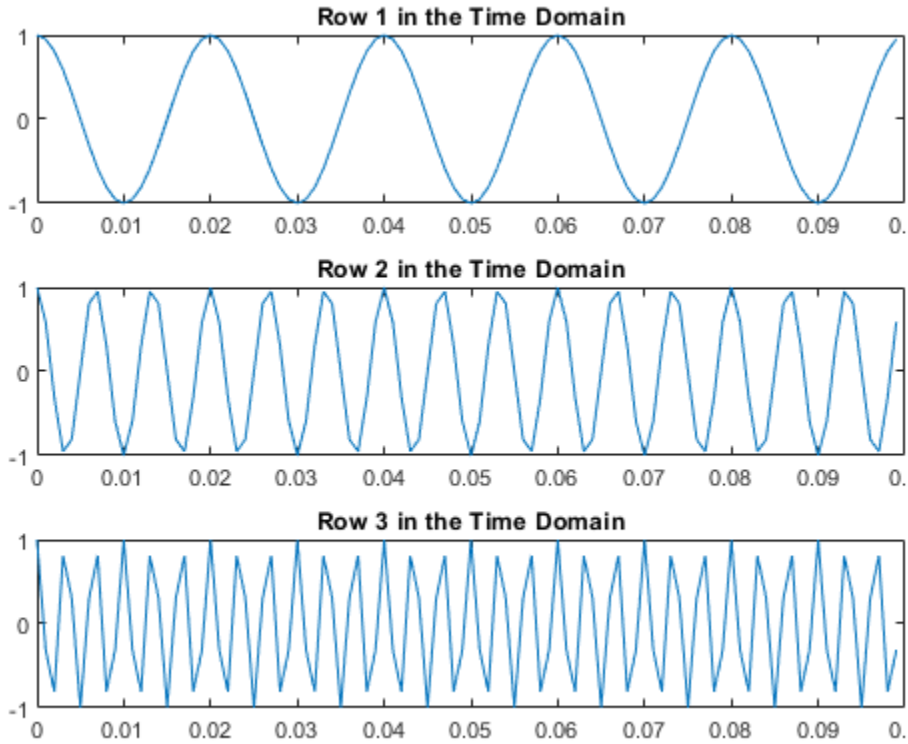
```
Fs = 1000;                    % Sampling frequency
T = 1/Fs;                     % Sampling period
L = 1000;                     % Length of signal
t = (0:L-1)*T;                % Time vector
```

Create a matrix where each row represents a cosine wave with scaled frequency. The result, X, is a 3-by-1000 matrix. The first row has a wave frequency of 50, the second row has a wave frequency of 150, and the third row has a wave frequency of 300.

```
x1 = cos(2*pi*50*t);          % First row wave
x2 = cos(2*pi*150*t);         % Second row wave
x3 = cos(2*pi*300*t);         % Third row wave

X = [x1; x2; x3];
```

Plot the first 100 entries from each row of X in a single figure in order and compare their frequencies.

```
for i = 1:3
    subplot(3,1,i)
    plot(t(1:100),X(i,1:100))
    title(['Row ',num2str(i),' in the Time Domain'])
end
```



For algorithm performance purposes, `fft` allows you to pad the input with trailing zeros. In this case, pad each row of X with zeros so that the length of each row is the next higher power of 2 from the current length. Define the new length using the `nextpow2` function.

```
n = 2^nextpow2(L);
```

Specify the `dim` argument to use `fft` along the rows of X, that is, for each signal.

```
dim = 2;
```

Compute the Fourier transform of the signals.

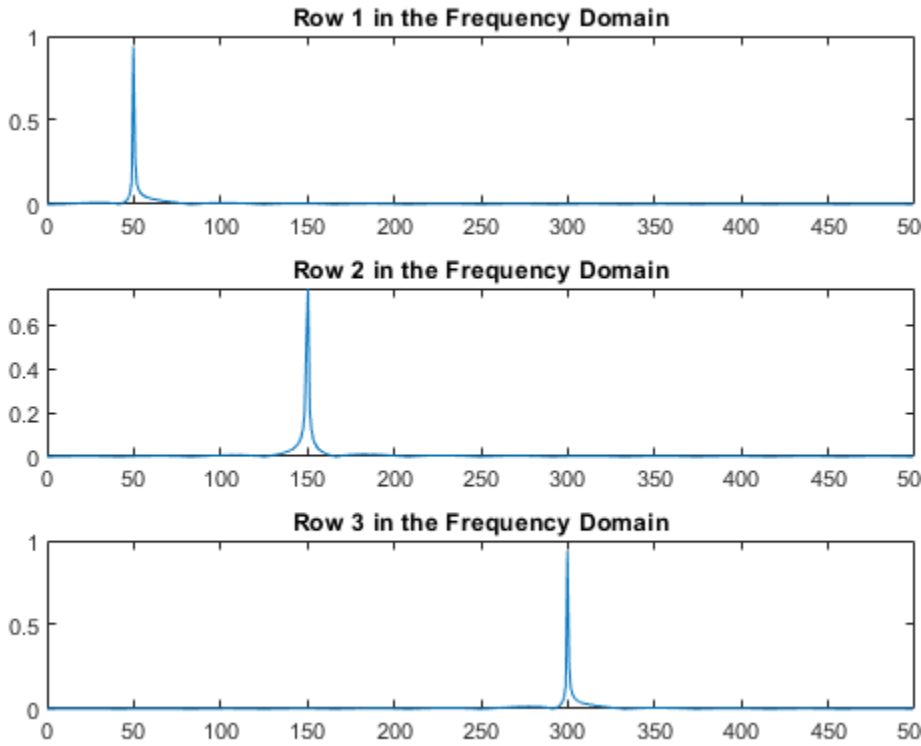
```
Y = fft(X,n,dim);
```

Calculate the double-sided spectrum and single-sided spectrum of each signal.

```
P2 = abs(Y/L);
P1 = P2(:,1:n/2+1);
P1(:,2:end-1) = 2*P1(:,2:end-1);
```

In the frequency domain, plot the single-sided amplitude spectrum for each row in a single figure.

```
for i=1:3
    subplot(3,1,i)
    plot(0:(Fs/n):(Fs/2-Fs/n),P1(i,1:n/2))
    title(['Row ',num2str(i),' in the Frequency Domain'])
end
```



Input Arguments

X — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

If X is an empty 0-by-0 matrix, then `fft(X)` returns an empty 0-by-0 matrix.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`
Complex Number Support: Yes

n — Transform length

`[]` (default) | nonnegative integer scalar

Transform length, specified as `[]` or a nonnegative integer scalar. Specifying a positive integer scalar for the transform length can increase the performance of `fft`. The length is typically specified as a power of 2 or a value that can be factored into a product of small prime numbers. If n is less than the length of the signal, then `fft` ignores the remaining signal values past the nth entry and returns the truncated result. If n is `0`, then `fft` returns an empty matrix.

Example: `n = 2^nextpow2(size(X,1))`

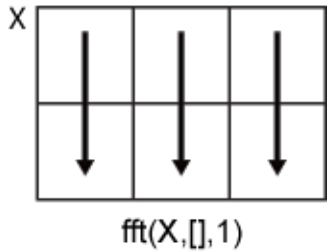
Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

dim — Dimension to operate along

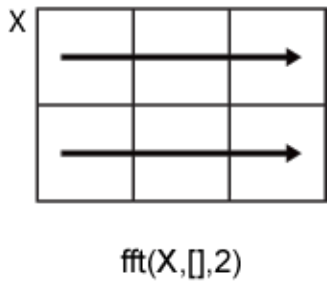
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

- `fft(X,[],1)` operates along the columns of X and returns the Fourier transform of each column.



- `fft(X,[],2)` operates along the rows of `X` and returns the Fourier transform of each row.



If `dim` is greater than `ndims(X)`, then `fft(X,[],dim)` returns `X`. When `n` is specified, `fft(X,n,dim)` pads or truncates `X` to length `n` along dimension `dim`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

Output Arguments

Y — Frequency domain representation
vector | matrix | multidimensional array

Frequency domain representation returned as a vector, matrix, or multidimensional array.

If `X` is of type `single`, then `fft` natively computes in single precision, and `Y` is also of type `single`. Otherwise, `Y` is returned as type `double`.

The size of `Y` is as follows:

- For `Y = fft(X)` or `Y = fft(X,[],dim)`, the size of `Y` is equal to the size of `X`.

- For `Y = fft(X,n,dim)`, the value of `size(Y,dim)` is equal to `n`, while the size of all other dimensions remains as in `X`.

If `X` is real, then `Y` is conjugate symmetric, and the number of unique points in `Y` is `ceil((n+1)/2)`.

Data Types: `double` | `single`

More About

Discrete Fourier Transform of Vector

`Y = fft(X)` and `X = ifft(Y)` implement the Fourier transform and inverse Fourier transform, respectively. For `X` and `Y` of length `n`, these transforms are defined as follows:

$$Y(k) = \sum_{j=1}^{n} X(j)\, W_n^{(j-1)(k-1)}$$

$$X(j) = \frac{1}{n} \sum_{k=1}^{n} Y(k)\, W_n{}^{-(j-1)(k-1)},$$

where

$$W_n = e^{(-2\pi i)/n}$$

is one of n roots of unity.

Tips

- The execution time for `fft` depends on the length of the transform. Transform lengths that have only small prime factors are significantly faster than those that are prime or have large prime factors.
- For most values of `n`, real-input DFTs require roughly half the computation time of complex-input DFTs. However, when `n` has large prime factors, there is little or no speed difference.
- You can potentially increase the speed of `fft` using the utility function, `fftw`. This function controls the optimization of the algorithm used to compute an FFT of a particular size and dimension.

Algorithms

The FFT functions (`fft`, `fft2`, `fftn`, `ifft`, `ifft2`, `ifftn`) are based on a library called FFTW [1] [2].

References

[1] FFTW (`http://www.fftw.org`)

[2] Frigo, M., and S. G. Johnson. "FFTW: An Adaptive Software Architecture for the FFT." *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*. Vol. 3, 1998, pp. 1381-1384.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For limitations related to variable-size data, see "Variable-Sizing Restrictions for Code Generation of Toolbox Functions" (MATLAB Coder).
- For MEX output, MATLAB Coder™ uses the library that MATLAB uses for FFT algorithms. For standalone C/C++ code, by default, the code generator produces code for FFT algorithms instead of producing FFT library calls. To generate calls to a specific installed FFTW library, provide an FFT library callback class. For more information about an FFT library callback class, see `coder.fftw.StandaloneFFTW3Interface`.
- For simulation of a MATLAB Function block, the simulation software uses the library that MATLAB uses for FFT algorithms. For C/C++ code generation, by default, the code generator produces code for FFT algorithms instead of producing FFT library calls. To generate calls to a specific installed FFTW library, provide an FFT library callback class. For more information about an FFT library callback class, see `coder.fftw.StandaloneFFTW3Interface`.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The output `Y` is always complex even if all the imaginary parts are zero.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- For distributed arrays, instead of using a parallel FFT algorithm, `fft` gathers vectors on a single worker to perform prime length FFTs. For large prime-length vector FFTs, out-of-memory errors can result.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`fft2` | `fftn` | `fftshift` | `fftw` | `ifft`

Topics

“Fourier Transforms”

Introduced before R2006a

fft2

2-D fast Fourier transform

Syntax

```
Y = fft2(X)
Y = fft2(X,m,n)
```

Description

`Y = fft2(X)` returns the two-dimensional Fourier transform on page 1-4158 of a matrix using a fast Fourier transform algorithm, which is equivalent to computing `fft(fft(X).').'`. If `X` is a multidimensional array, then `fft2` takes the 2-D transform of each dimension higher than 2. The output `Y` is the same size as `X`.

`Y = fft2(X,m,n)` truncates `X` or pads `X` with trailing zeros to form an `m`-by-`n` matrix before computing the transform. `Y` is `m`-by-`n`. If `X` is a multidimensional array, then `fft2` shapes the first two dimensions of `X` according to `m` and `n`.

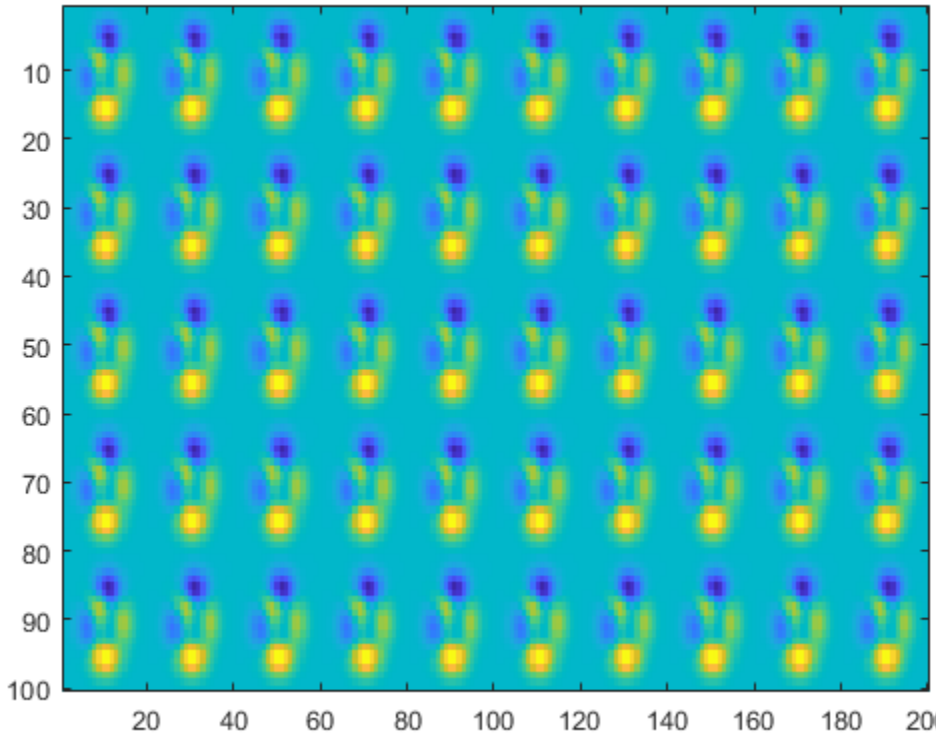
Examples

2-D Transform

The 2-D Fourier transform is useful for processing 2-D signals and other 2-D data such as images.

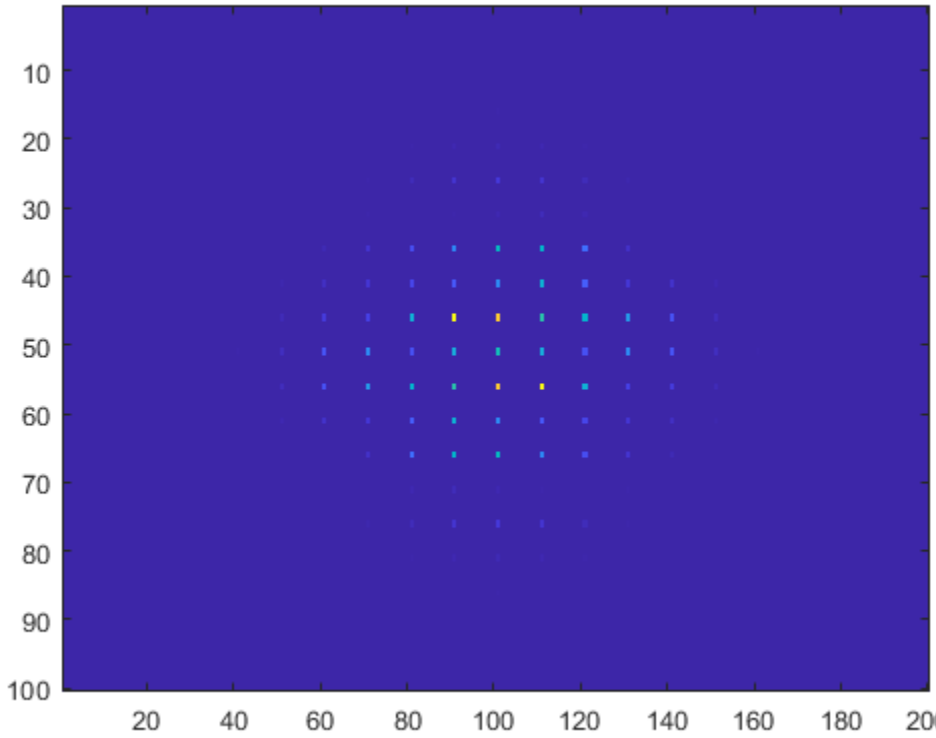
Create and plot 2-D data with repeated blocks.

```
P = peaks(20);
X = repmat(P,[5 10]);
imagesc(X)
```



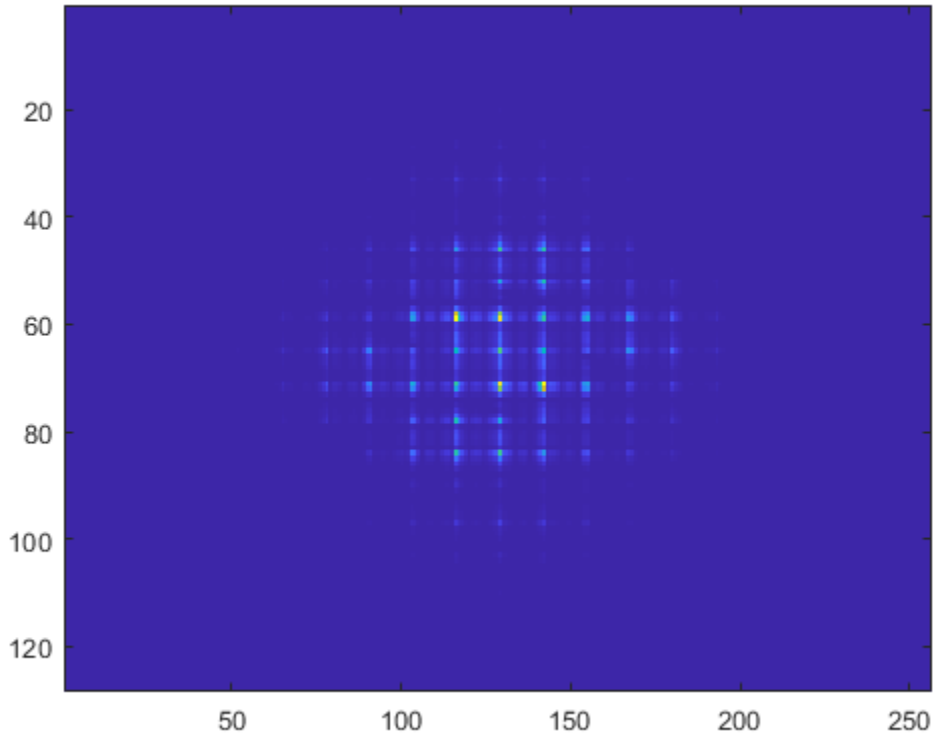
Compute the 2-D Fourier transform of the data. Shift the zero-frequency component to the center of the output, and plot the resulting 100-by-200 matrix, which is the same size as X.

```
Y = fft2(X);
imagesc(abs(fftshift(Y)))
```



Pad X with zeros to compute a 128-by-256 transform.

```
Y = fft2(X,2^nextpow2(100),2^nextpow2(200));
imagesc(abs(fftshift(Y)));
```



Input Arguments

X — Input array
matrix | multidimensional array

Input array, specified as a matrix or a multidimensional array. If X is of type `single`, then `fft2` natively computes in single precision, and Y is also of type `single`. Otherwise, Y is returned as type `double`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

Complex Number Support: Yes

m — Number of transform rows
positive integer scalar

Number of transform rows, specified as a positive integer scalar.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

n — Number of transform columns
positive integer scalar

Number of transform columns, specified as a positive integer scalar.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

More About

2-D Fourier Transform

This formula defines the discrete Fourier transform Y of an m-by-n matrix X:

$$Y_{p+1,q+1} = \sum_{j=0}^{m-1} \sum_{k=0}^{n-1} \omega_m^{jp} \omega_n^{kq} X_{j+1,k+1}$$

ω_m and ω_n are complex roots of unity:

$$\omega_m = e^{-2\pi i/m}$$
$$\omega_n = e^{-2\pi i/n}$$

i is the imaginary unit. p and j are indices that run from 0 to m–1, and q and k are indices that run from 0 to n–1. This formula shifts the indices for X and Y by 1 to reflect matrix indices in MATLAB.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For MEX output, MATLAB Coder uses the library that MATLAB uses for FFT algorithms. For standalone C/C++ code, by default, the code generator produces code for FFT algorithms instead of producing FFT library calls. To generate calls to a specific installed FFTW library, provide an FFT library callback class. For more information about an FFT library callback class, see `coder.fftw.StandaloneFFTW3Interface`.
- For simulation of a MATLAB Function block, the simulation software uses the library that MATLAB uses for FFT algorithms. For C/C++ code generation, by default, the code generator produces code for FFT algorithms instead of producing FFT library calls. To generate calls to a specific installed FFTW library, provide an FFT library callback class. For more information about an FFT library callback class, see `coder.fftw.StandaloneFFTW3Interface`.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The output `Y` is always complex even if all the imaginary parts are zero.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`fft` | `fftn` | `fftw` | `ifft2`

Introduced before R2006a

fftn

N-D fast Fourier transform

Syntax

```
Y = fftn(X)
Y = fftn(X,sz)
```

Description

`Y = fftn(X)` returns the multidimensional Fourier transform on page 1-4163 of an N-D array using a fast Fourier transform algorithm. The N-D transform is equivalent to computing the 1-D transform along each dimension of `X`. The output `Y` is the same size as `X`.

`Y = fftn(X,sz)` truncates `X` or pads `X` with trailing zeros before taking the transform according to the elements of the vector `sz`. Each element of `sz` defines the length of the corresponding transform dimensions. For example, if `X` is a 5-by-5-by-5 array, then `Y = fftn(X,[8 8 8])` pads each dimension with zeros resulting in an 8-by-8-by-8 transform `Y`.

Examples

3-D Transform

You can use the `fftn` function to compute a 1-D fast Fourier transform in each dimension of a multidimensional array.

Create a 3-D signal `X`. The size of `X` is 20-by-20-by-20.

```
x = (1:20)';
y = 1:20;
z = reshape(1:20,[1 1 20]);
X = cos(2*pi*0.01*x) + sin(2*pi*0.02*y) + cos(2*pi*0.03*z);
```

Compute the 3-D Fourier transform of the signal, which is also a 20-by-20-by-20 array.

```
Y = fftn(X);
```

Pad X with zeros to compute a 32-by-32-by-32 transform.

```
m = nextpow2(20);
Y = fftn(X,[2^m 2^m 2^m]);
size(Y)
```

```
ans = 1×3

    32    32    32
```

Input Arguments

X — Input array
matrix | multidimensional array

Input array, specified as a matrix or a multidimensional array. If X is of type `single`, then `fftn` natively computes in single precision, and Y is also of type `single`. Otherwise, Y is returned as type `double`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`
Complex Number Support: Yes

`sz` — Length of transform dimensions
vector of positive integers

Length of the transform dimensions, specified as a vector of positive integers. The elements of `sz` correspond to the transformation lengths of the corresponding dimensions of X. `length(sz)` must be at least `ndims(X)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

More About

N-D Fourier Transform

The discrete Fourier transform Y of an N-D array X is defined as

$$Y_{p_1, p_2, \ldots, p_N} = \sum_{j_1 = 0}^{m_1 - 1} \omega_{m_1}^{p_1 j_1} \sum_{j_2 = 0}^{m_2 - 1} \omega_{m_2}^{p_2 j_2} \ldots \sum_{j_N}^{m_N - 1} \omega_{m_N}^{p_N j_N} X_{j_1, j_2, \ldots, j_N}.$$

Each dimension has length m_k for $k = 1,2,\ldots,N$, and $\omega_{m_k} = e^{-2\pi i/m_k}$ are complex roots of unity where i is the imaginary unit.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The `sz` argument must have a fixed size.
- For MEX output, MATLAB Coder uses the library that MATLAB uses for FFT algorithms. For standalone C/C++ code, by default, the code generator produces code for FFT algorithms instead of producing FFT library calls. To generate calls to a specific installed FFTW library, provide an FFT library callback class. For more information about an FFT library callback class, see `coder.fftw.StandaloneFFTW3Interface`.
- For simulation of a MATLAB Function block, the simulation software uses the library that MATLAB uses for FFT algorithms. For C/C++ code generation, by default, the code generator produces code for FFT algorithms instead of producing FFT library calls. To generate calls to a specific installed FFTW library, provide an FFT library callback class. For more information about an FFT library callback class, see `coder.fftw.StandaloneFFTW3Interface`.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- The `sz` argument must have a fixed size.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The output `Y` is always complex even if all the imaginary parts are zero.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`fft` | `fft2` | `fftw` | `ifftn`

Introduced before R2006a

fftshift

Shift zero-frequency component to center of spectrum

Syntax

```
Y = fftshift(X)
Y = fftshift(X,dim)
```

Description

`Y = fftshift(X)` rearranges a Fourier transform `X` by shifting the zero-frequency component to the center of the array.

- If `X` is a vector, then `fftshift` swaps the left and right halves of `X`.
- If `X` is a matrix, then `fftshift` swaps the first quadrant of `X` with the third, and the second quadrant with the fourth.
- If `X` is a multidimensional array, then `fftshift` swaps half-spaces of `X` along each dimension.

`Y = fftshift(X,dim)` operates along the dimension `dim` of `X`. For example, if `X` is a matrix whose rows represent multiple 1-D transforms, then `fftshift(X,2)` swaps the halves of each row of `X`.

Examples

Shift Vector Elements

Swap the left and right halves of a row vector. If a vector has an odd number of elements, then the middle element is considered part of the left half of the vector.

```
Xeven = [1 2 3 4 5 6];
fftshift(Xeven)
```

```
ans = 1×6

     4     5     6     1     2     3
```

```
Xodd = [1 2 3 4 5 6 7];
fftshift(Xodd)
```

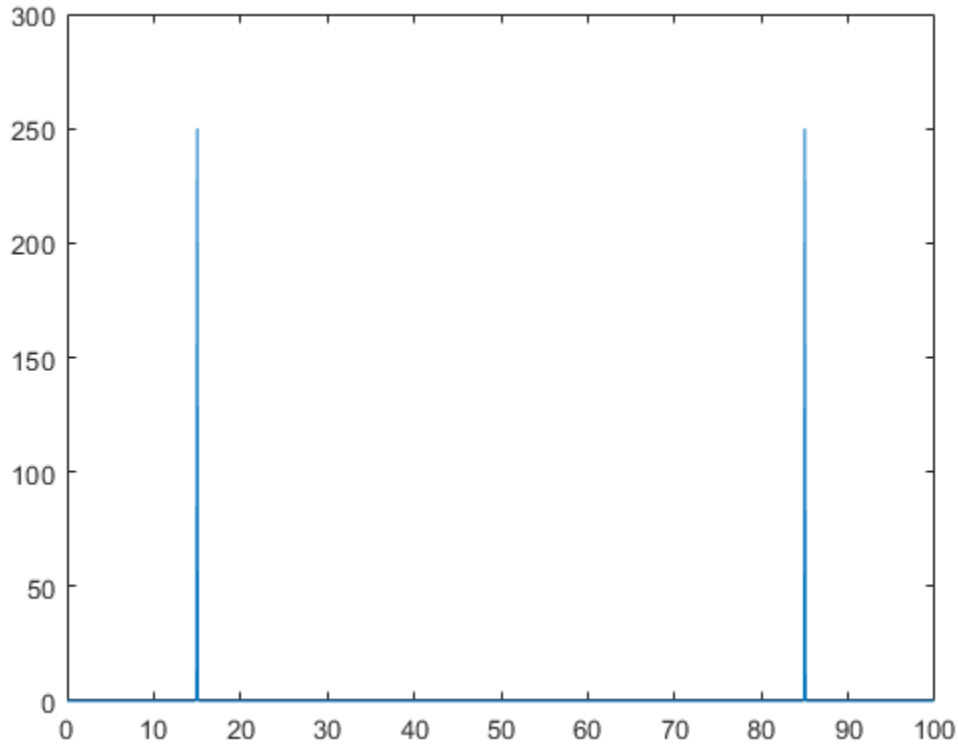
```
ans = 1×7

     5     6     7     1     2     3     4
```

Shift 1-D Signal

When analyzing the frequency components of signals, it can be helpful to shift the zero-frequency components to the center.

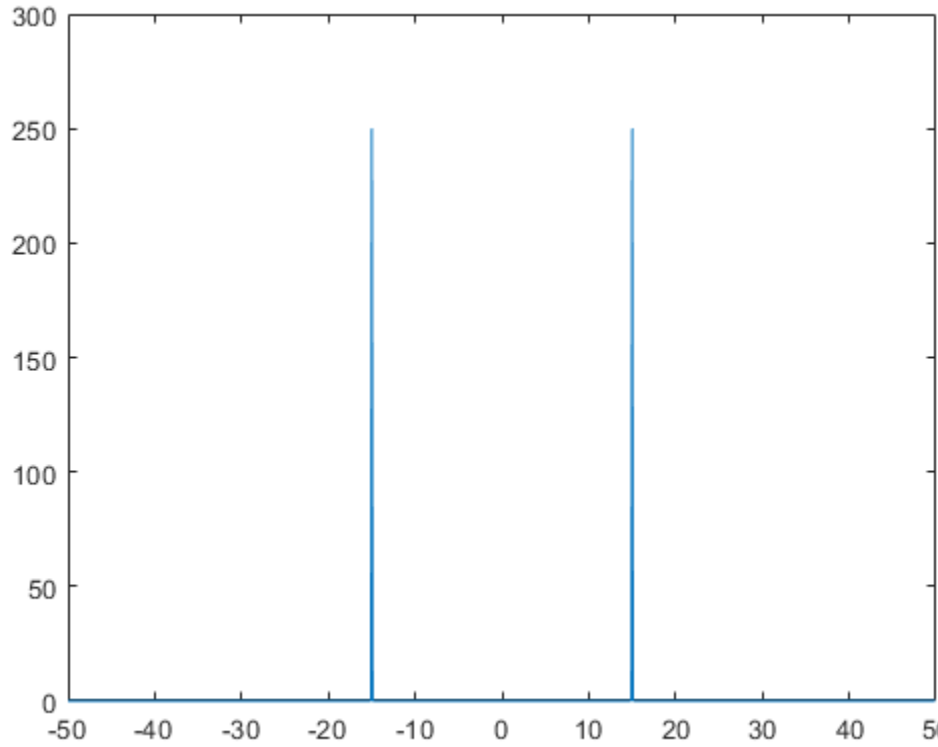
Create a signal S, compute its Fourier transform, and plot the power.

```
fs = 100;                % sampling frequency
t = 0:(1/fs):(10-1/fs); % time vector
S = cos(2*pi*15*t);
n = length(S);
X = fft(S);
f = (0:n-1)*(fs/n);      %frequency range
power = abs(X).^2/n;     %power
plot(f,power)
```



Shift the zero-frequency components and plot the zero-centered power.

```
Y = fftshift(X);
fshift = (-n/2:n/2-1)*(fs/n); % zero-centered frequency range
powershift = abs(Y).^2/n;     % zero-centered power
plot(fshift,powershift)
```



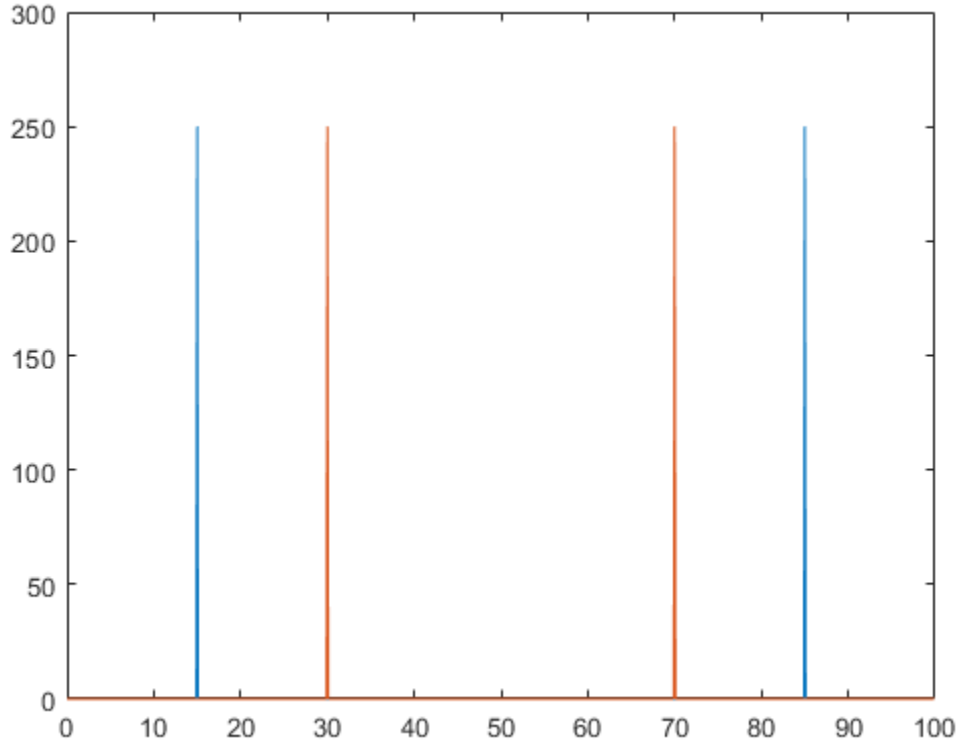
Shift Signals in Matrix

You can process multiple 1-D signals by representing them as rows in a matrix. Then use the dimension argument to compute the Fourier transform and shift the zero-frequency components for each row.

Create a matrix A whose rows represent two 1-D signals, and compute the Fourier transform of each signal. Plot the power for each signal.

```
fs = 100;                % sampling frequency
t = 0:(1/fs):(10-1/fs); % time vector
```

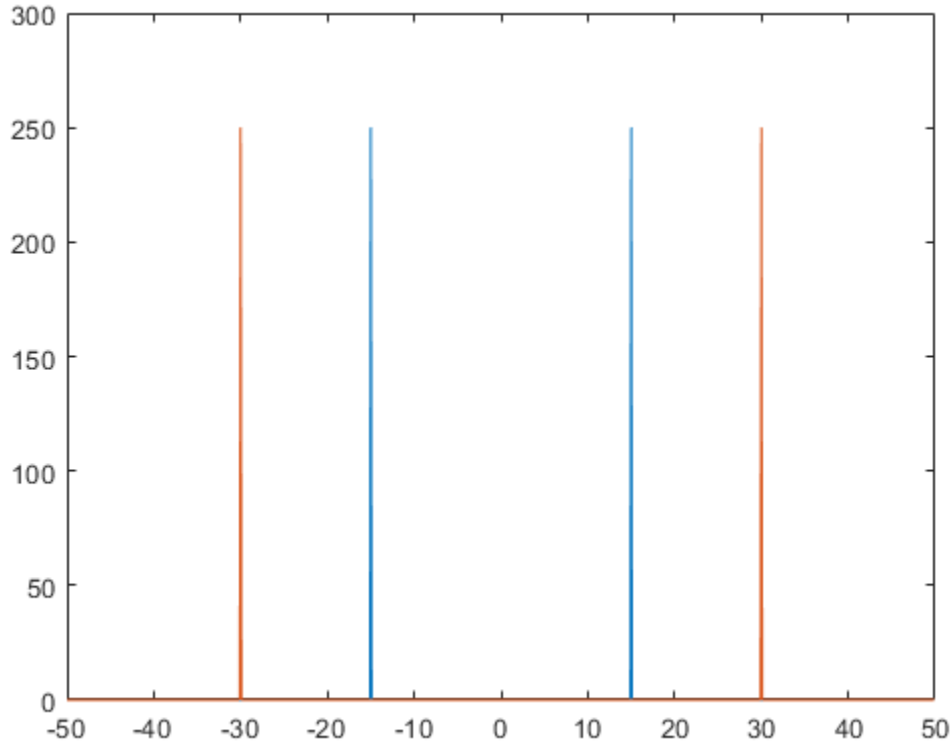
```
S1 = cos(2*pi*15*t);
S2 = cos(2*pi*30*t);
n = length(S1);
A = [S1; S2];
X = fft(A,[],2);
f = (0:n-1)*(fs/n);     % frequency range
power = abs(X).^2/n;    % power
plot(f,power(1,:),f,power(2,:))
```



Shift the zero-frequency components, and plot the zero-centered power of each signal.

```
Y = fftshift(X,2);
fshift = (-n/2:n/2-1)*(fs/n); % zero-centered frequency range
```

```
powershift = abs(Y).^2/n;     % zero-centered power
plot(fshift,powershift(1,:),fshift,powershift(2,:))
```



Input Arguments

X — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, a matrix, or a multidimensional array.

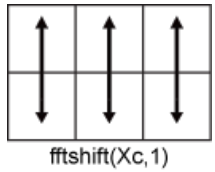
Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Complex Number Support: Yes

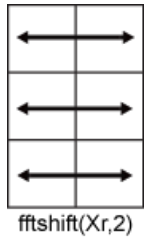
`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then `fftshift` swaps along all dimensions.

- Consider an input matrix `Xc`. The operation `fftshift(Xc,1)` swaps halves of each column of `Xc`.



- Consider a matrix `Xr`. The operation `fftshift(Xr,2)` swaps halves of each row of `Xr`.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`fft` | `fft2` | `fftn` | `fftw` | `ifftshift`

Introduced before R2006a

fftw

Define method for determining FFT algorithm

Syntax

```
method = fftw('planner')
previous = fftw('planner',method)

fftinfo = fftw(wisdom)
previous = fftw(wisdom,fftinfo)
```

Description

`method = fftw('planner')` returns the method that the fast Fourier transform functions `fft`, `fft2`, `fftn`, `ifft`, `ifft2`, and `ifftn` use to determine a transform algorithm. The default method is `'estimate'`, which determines the algorithm based on the size of the data.

`previous = fftw('planner',method)` sets the method and optionally returns the previous method. For example, `fftw('planner','measure')` chooses the fastest algorithm based on the size and type of the data.

`fftinfo = fftw(wisdom)` returns the optimal transform parameters established in the current MATLAB session for the precision specified in `wisdom`. For example, `fftinfo = fftw('dwisdom')` saves the current double-precision transform parameters.

`previous = fftw(wisdom,fftinfo)` applies the parameters in `fftinfo` to the transform algorithm for the precision specified in `wisdom`.

Examples

Optimal Fourier Transform Workflow

The `fftw` function can improve the performance of Fourier transform functions by trying to optimize computation based on the size and type of the data.

Clear any existing double-precision transform parameters.

```
fftw('dwisdom',[]);
```

Set the method for optimizing Fourier transforms within the current MATLAB® session to `'measure'`.

```
fftw('planner','measure');
```

Define a random vector and use the `fft` function to compute the Fourier transform. `fft` uses an algorithm that tries to optimize computation for the 500-by-1 transform. The first call to `fft` can be slower than normal, but subsequent calls will be significantly faster.

```
X = rand(500,1);
tic; fft(X); toc;
```

```
Elapsed time is 0.017237 seconds.
```

```
tic; fft(X); toc;
```

```
Elapsed time is 0.000164 seconds.
```

Assign the current double-precision transform algorithm parameters to the variable `fftinfo` for later use.

```
fftinfo = fftw('dwisdom');
```

Apply the parameter information stored in the variable `fftinfo` to future transform function calls.

```
fftw('dwisdom',fftinfo);
```

Input Arguments

`method` — Method for setting transform parameters
`'estimate'` (default) | `'measure'` | `'patient'` | `'exhaustive'` | `'hybrid'`

Method for setting transform parameters, specified as one of the following:

- `'estimate'` — Determine a best-guess transform algorithm based on the size of problem.
- `'measure'` — Find a better algorithm by computing multiple transforms and measuring the run times.
- `'patient'` — Run a wider range of testing compared to `'measure'`, resulting in a better transform algorithm, but at the expense of higher computational cost to determine the parameters.
- `'exhaustive'` — Runs all tests including ones unlikely to be optimal, resulting in the best algorithm, but with the highest computational cost to determine the parameters.
- `'hybrid'` — Use a combination of `'measure'` for transforms with dimension length 8192 or smaller and `'estimate'` for transforms with dimension length larger than 8192.

Data Types: `char` | `string`

`wisdom` — Transform parameter precision
`'swisdom'` | `'dwisdom'`

Transform parameter precision, specified as either `'swisdom'` or `'dwisdom'`. The option `'swisdom'` specifies transform parameters for single-precision computation, and `'dwisdom'` specifies parameters for double-precision computation.

Data Types: `char`

`fftinfo` — Transform parameters
character array | `[]`

Transform parameters, specified as a character array of transform parameters returned by `fftw(wisdom)` or the empty array `[]`. The empty array `[]` clears the database.

Data Types: `char`

Tips

- Transform parameters returned by `fftw` are most useful when saved and applied on one computer using the same version of MATLAB each session. Transferring saved parameters between computers can result in degraded performance, even if both computers have the same operating system and MATLAB version.

Algorithms

- For transform lengths that are powers of 2 between 2^{14} and 2^{22}, MATLAB uses pre-loaded information from its internal database to optimize the transform computation. Tuning is not executed for exact powers of 2 unless you clear the database using the command `fftw(wisdom,[])`.

References

[1] FFTW (http://www.fftw.org)

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation with MATLAB Coder supports `fftw` only for MEX output. For standalone C/C++ code, to select a planning method for FFT library calls, implement a `getPlanMethod` method in an FFT library callback class. For more information about an FFT library callback class, see `coder.fftw.StandaloneFFTW3Interface`.
- In a MATLAB Function block, `fftw` is supported only for simulation. For C/C++ code generation, to select a planning method for FFT library calls, implement a `getPlanMethod` method in an FFT library callback class. For more information about an FFT library callback class, see `coder.fftw.StandaloneFFTW3Interface`.

See Also

`fft` | `fft2` | `fftn` | `ifft` | `ifft2` | `ifftn`

Introduced before R2006a

fgetl

Read line from file, removing newline characters

Syntax

```
tline = fgetl(fileID)
```

Description

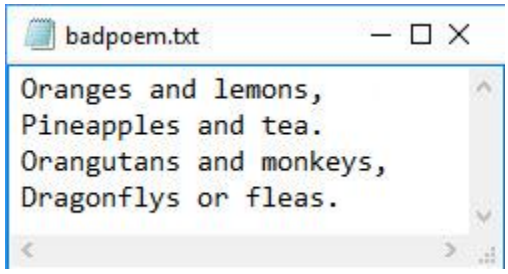
`tline = fgetl(fileID)` returns the next line of the specified file, removing the newline characters.

- If the file is nonempty, then `fgetl` returns `tline` as a character vector.
- If the file is empty and contains only the end-of-file marker, then `fgetl` returns `tline` as a numeric value `-1`.

Examples

Read File One Line at a Time

Read a single line from a file, first excluding newline characters, and then including them. Use the following file.



To read the first line from the file `badpoem.txt`, use `fopen` to open the file. Then read the first line using `fgetl`, which excludes the newline character.

```
fid = fopen('badpoem.txt');
line_ex = fgetl(fid)  % read line excluding newline character
```

```
line_ex =
'Oranges and lemons,'
```

To reread the same line from the file, first reset the read position indicator back to the beginning of the file.

```
frewind(fid);
```

Use the fgets function to read the first line from the file `badpoem.txt`, which reads the line including the newline character.

```
line_in = fgets(fid) % read line including newline character
```

```
line_in =
    'Oranges and lemons,
     '
```

Compare the output by examining the lengths of the lines returned by the `fgetl` and `fgets` functions.

```
length(line_ex)
```

```
ans = 19
```

```
length(line_in)
```

```
ans = 20
```

`fgetl` returns an output that displays in one line, while `fgets` returns an output that includes the newline character and, therefore, displays it in two lines.

```
line_ex
```

```
line_ex =
'Oranges and lemons,'
```

```
line_in
```

```
line_in =
    'Oranges and lemons,
```

'

Close the file.

```
fclose(fid);
```

Input Arguments

`fileID` — File identifier
integer

File identifier of an open file, specified as an integer. Before using `fgetl` to read a line from the file, you must use `fopen` to open the file and obtain its `fileID`.

Data Types: `double`

Tips

- `fgetl` reads characters using the encoding scheme associated with the file. To specify the encoding scheme, use `fopen`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If the function `fgetl` does not read content from the file, then the generated code reports either an error or returns an empty value instead of returning -1.
- If the function `fgetl` reads a null byte, the returned values might be truncated.

See Also

`fclose` | `feof` | `ferror` | `fgets` | `fopen` | `fprintf` | `fread` | `fscanf` | `fwrite` | `textscan`

Topics

"Import Text Data Files with Low-Level I/O"

Introduced before R2006a

fgetl (serial)

(Not recommended) Read line of ASCII text from device and discard terminator

Note This `serial` object function is not recommended. Use `serialport` object functions instead. See “Compatibility Considerations”.

Syntax

```
tline = fgetl(obj)
[tline,count] = fgetl(obj)
[tline,count,msg] = fgetl(obj)
```

Description

`tline = fgetl(obj)` reads one line of ASCII text from the device connected to the serial port object, `obj`, and returns the data to `tline`. This returned data does not include the terminator with the text line. To include the terminator, use `fgets`.

`[tline,count] = fgetl(obj)` returns the number of values read to `count`, including the terminator.

`[tline,count,msg] = fgetl(obj)` returns a warning message to `msg` if the read operation was unsuccessful.

Examples

On a Windows platform, create the serial port object `s`, connect `s` to a Tektronix® TDS 210 oscilloscope, and write the `RS232?` command with the `fprintf` function. `RS232?` instructs the scope to return serial port communications settings.

```
s = serial('COM1');
fopen(s)
fprintf(s,'RS232?')
```

Because the default value for the `ReadAsyncMode` property is `continuous`, data is automatically returned to the input buffer.

```
s.BytesAvailable
```

```
ans =
    17
```

Use `fgetl` to read the data returned from the previous write operation, and discard the terminator.

```
settings = fgetl(s)
```

```
settings =
9600;0;0;NONE;LF
```

```
length(settings)
```

```
ans =
    16
```

Disconnect `s` from the scope, and remove `s` from memory and the workspace.

```
fclose(s)
delete(s)
clear s
```

Tips

Before you can read text from the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read – including the terminator – each time `fgetl` is issued.

Note You cannot use ASCII values larger than 127 characters. The function is limited to 127 binary characters.

Rules for Completing a Read Operation with fgetl

A read operation with `fgetl` blocks access to the MATLAB command line until:

- The terminator specified by the `Terminator` property is reached.
- The time specified by the `Timeout` property passes.
- The input buffer is filled.

Note You cannot use ASCII values larger than 127 characters. The function is limited to 127 binary characters.

Compatibility Considerations

`serial` object interface is not recommended

Not recommended starting in R2019b

Use of this function with a `serial` object is not recommended. To access a serial port device, use a `serialport` object with its functions and properties instead.

See “Transition Your Code to serialport Interface” for more information about using the recommended functionality.

See Also

Functions

`fgets` | `fopen`

Introduced before R2006a

fgets

Read line from file, keeping newline characters

Syntax

```
tline = fgets(fileID)
tline = fgets(fileID,nchar)
[tline,ltout] = fgets( ___ )
```

Description

`tline = fgets(fileID)` reads the next line of the specified file, including the newline characters.

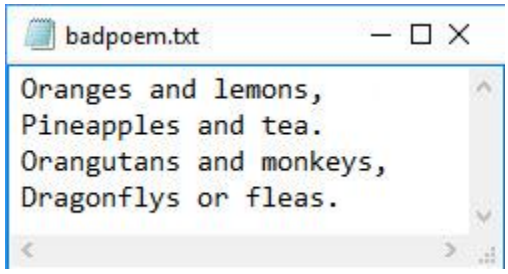
`tline = fgets(fileID,nchar)` returns up to `nchar` characters of the next line.

`[tline,ltout] = fgets( ___ )` also returns the line terminators, if any, in `ltout`.

Examples

Read File One Line at a Time

Read a single line from a file, first excluding newline characters, and then including them. Use the following file.



To read the first line from the file `badpoem.txt`, use `fopen` to open the file. Then read the first line using `fgetl`, which excludes the newline character.

```
fid = fopen('badpoem.txt');
line_ex = fgetl(fid)  % read line excluding newline character
```

```
line_ex =
'Oranges and lemons,'
```

To reread the same line from the file, first reset the read position indicator back to the beginning of the file.

```
frewind(fid);
```

Use the fgets function to read the first line from the file `badpoem.txt`, which reads the line including the newline character.

```
line_in = fgets(fid) % read line including newline character
```

```
line_in =
    'Oranges and lemons,
     '
```

Compare the output by examining the lengths of the lines returned by the `fgetl` and `fgets` functions.

```
length(line_ex)
```

```
ans = 19
```

```
length(line_in)
```

```
ans = 20
```

`fgetl` returns an output that displays in one line, while `fgets` returns an output that includes the newline character and, therefore, displays it in two lines.

```
line_ex
```

```
line_ex =
'Oranges and lemons,'
```

```
line_in
```

```
line_in =
    'Oranges and lemons,
```

'

Close the file.

```
fclose(fid);
```

Input Arguments

`fileID` — File identifier
integer

File identifier of an open file, specified as an integer. Before using `fgets` to read a line from the file, you must use `fopen` to open the file and obtain its `fileID`.

Data Types: `double`

`nchar` — Number of characters
integer

Number of characters to read from the next line, specified as an integer. `fgets` returns at most `nchar` characters of the next line. If the number of characters specified by `nchar` includes characters beyond the newline character or the end-of-file marker, then `fgets` does not return any characters beyond the new line character or the end-of-file marker.

Data Types: `double`

Output Arguments

`tline` — Next line in file
character vector | numeric scalar

Next line in file, returned as a character vector or numeric scalar.

- If the file is nonempty, then `fgets` returns `tline` as a character vector.
- If the file is empty and contains only the end-of-file marker, then `fgets` returns `tline` as a numeric value `-1`.

`ltout` — Line terminators
integer

Line terminators, returned as an integer.

The integers from `0` to `65535` correspond to Unicodecharacters. You can convert integers to their corresponding Unicode representations using the `char` function.

Tips

- `tline` does not include any characters after the newline characters or the end-of-file marker.
- `fgets` reads characters using the encoding scheme associated with the file. To specify the encoding scheme, use `fopen`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If the function `fgets` does not read content from the file, then the generated code reports either an error or returns an empty value instead of returning -1.
- If the function `fgets` reads a null byte, the returned values might be truncated.

See Also

`fclose` | `feof` | `ferror` | `fgetl` | `fopen` | `fprintf` | `fread` | `fscanf` | `fwrite` | `textscan`

Topics

"Import Text Data Files with Low-Level I/O"

Introduced before R2006a

fgets (serial)

(Not recommended) Read line of text from device and include terminator

Note This `serial` object function is not recommended. Use `serialport` object functions instead. See “Compatibility Considerations”.

Syntax

```
tline = fgets(obj)
[tline,count] = fgets(obj)
[tline,count,msg] = fgets(obj)
```

Description

`tline = fgets(obj)` reads one line of text from the device connected to the serial port object, `obj`, and returns the data to `tline`. This returned data includes the terminator with the text line. To exclude the terminator, use `fgetl`.

`[tline,count] = fgets(obj)` returns the number of values read to `count`, including the terminator.

`[tline,count,msg] = fgets(obj)` returns a warning message to `msg` if the read operation was unsuccessful.

Examples

Create the serial port object `s`, connect `s` to a Tektronix TDS 210 oscilloscope, and write the `RS232?` command with the `fprintf` function. `RS232?` instructs the scope to return serial port communications settings.

```
s = serial('COM1');
fopen(s)
fprintf(s,'RS232?')
```

Because the default value for the `ReadAsyncMode` property is `continuous`, data is automatically returned to the input buffer.

```
s.BytesAvailable
```

```
ans =
    17
```

Use `fgets` to read the data returned from the previous write operation, and include the terminator.

```
settings = fgets(s)
```

```
settings =
9600;0;0;NONE;LF
```

```
length(settings)
```

```
ans =
    17
```

Disconnect `s` from the scope, and remove `s` from memory and the workspace.

```
fclose(s)
delete(s)
clear s
```

Tips

Before you can read text from the device, it must be connected to `obj` with the `fopen`function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read – including the terminator – each time `fgets` is issued.

Note You cannot use ASCII values larger than 127 characters. The function is limited to 127 binary characters.

Rules for Completing a Read Operation with fgets

A read operation with `fgets` blocks access to the MATLAB command line until:

- The terminator specified by the `Terminator` property is reached.
- The time specified by the `Timeout` property passes.
- The input buffer is filled.

Compatibility Considerations

`serial` object interface is not recommended

Not recommended starting in R2019b

Use of this function with a `serial` object is not recommended. To access a serial port device, use a `serialport` object with its functions and properties instead.

See "Transition Your Code to serialport Interface" for more information about using the recommended functionality.

See Also

Functions
`fgetl` | `fopen`

Introduced before R2006a

fieldnames

Field names of structure, or public fields of Java or Microsoft COM object

Syntax

```
fields = fieldnames(S)
fields = fieldnames(obj,'-full')
```

Description

`fields = fieldnames(S)` returns the field names of the structure array `S` in a cell array.

`fields = fieldnames(obj,'-full')` returns a cell array of character vectors containing the name, type, attributes, and inheritance of the properties of `obj`. The input argument `obj` is a Java or Microsoft COM object.

Examples

Return Field Names and Values

Create a structure array.

```
S(1,1).x = linspace(0,2*pi);
S(1,1).y = sin(S(1,1).x);
S(1,1).title = 'y = sin(x)';
S(2,1).x = linspace(0,2*pi);
S(2,1).y = cos(S(2,1).x);
S(2,1).title = 'y = cos(x)'
```

```
S=2×3 struct
    x
    y
    title
```

Return the field names in a cell array using the `fieldnames` function.

```
fields = fieldnames(S)
```

```
fields = 3x1 cell array
    {'x'    }
    {'y'    }
    {'title'}
```

To return the values of the fields, use the `struct2cell` function. `struct2cell` and `fieldnames` return the values and the field names in the same order.

```
values = struct2cell(S)
```

```
values=3×2 cell
    {1x100 double}    {1x100 double}
    {1x100 double}    {1x100 double}
    {'y = sin(x)'}    {'y = cos(x)'}
```

Input Arguments

S — Input structure array
structure array

Input structure array.

`obj` — Input object
Java object | Microsoft COM object

Input object, specified as a Java object or Microsoft COM object.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The input must be a structure. Objects are not supported.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`getfield` | `isfield` | `orderfields` | `properties` | `rmfield` | `setfield` | `struct2cell`

Topics

“Generate Field Names from Variables”

Introduced before R2006a

figure

Create figure window

Syntax

```
figure
figure(Name,Value)
f = figure( ___ )

figure(f)
figure(n)
```

Description

`figure` creates a new figure window using default property values. The resulting figure is the current figure on page 1-4203.

`figure(Name,Value)` modifies properties of the figure using one or more name-value pair arguments. For example, `figure('Color','white')` sets the background color to white.

`f = figure( ___ )` returns the `Figure` object. Use `f` to query or modify properties of the figure after it is created.

`figure(f)` makes the figure specified by `f` the current figure and displays it on top of all other figures.

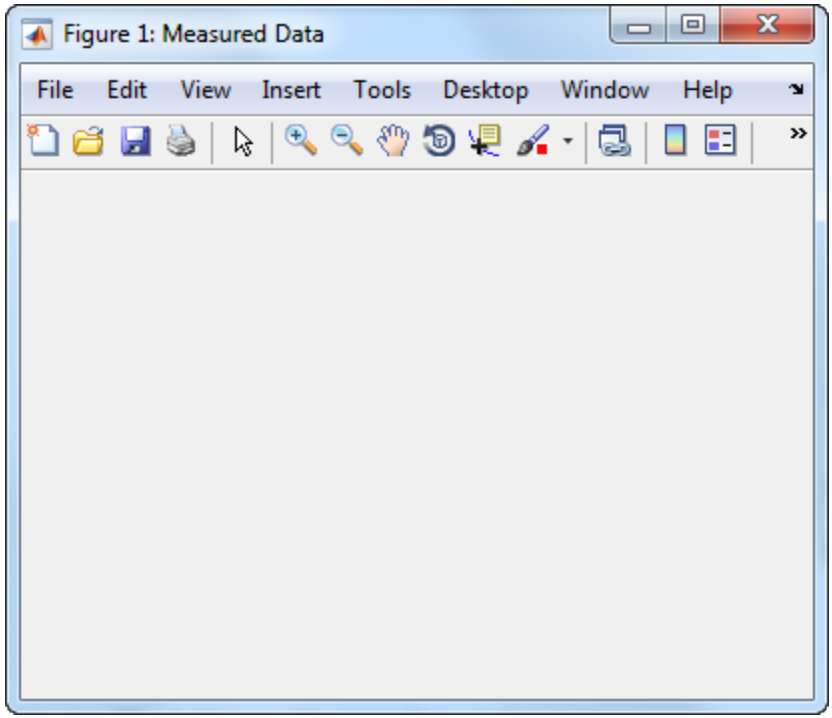
`figure(n)` finds a figure in which the `Number` property is equal to `n`, and makes it the current figure. If no figure exists with that property value, MATLAB creates a new figure and sets its `Number` property to `n`.

Examples

Specify Figure Title

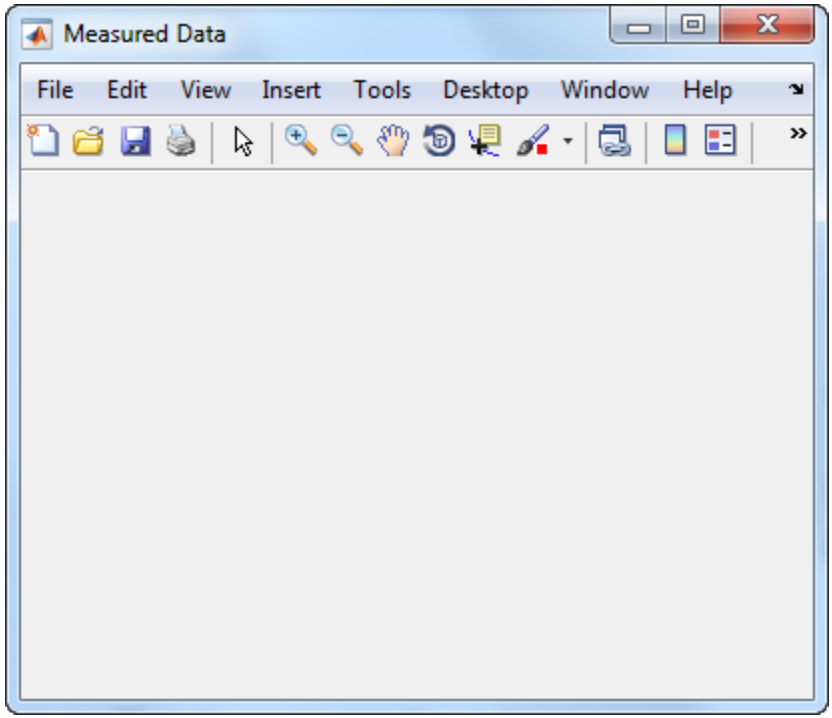
Create a figure, and specify the `Name` property. By default, the resulting title includes the figure number.

```
figure('Name','Measured Data');
```



Specify the `Name` property again, but this time, set the `NumberTitle` property to `'off'`. The resulting title does not include the figure number.

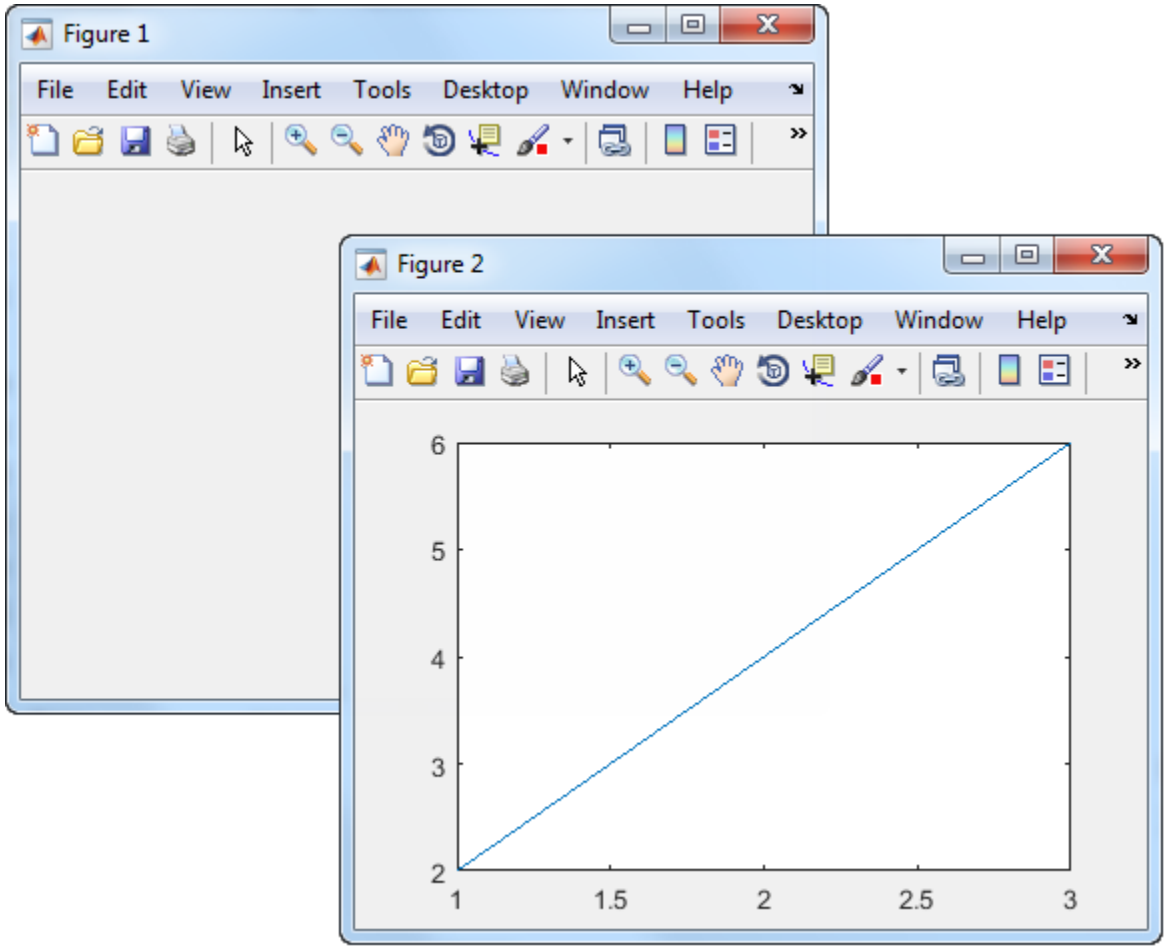
```
figure('Name','Measured Data','NumberTitle','off');
```



Working with Multiple Figures Simultaneously

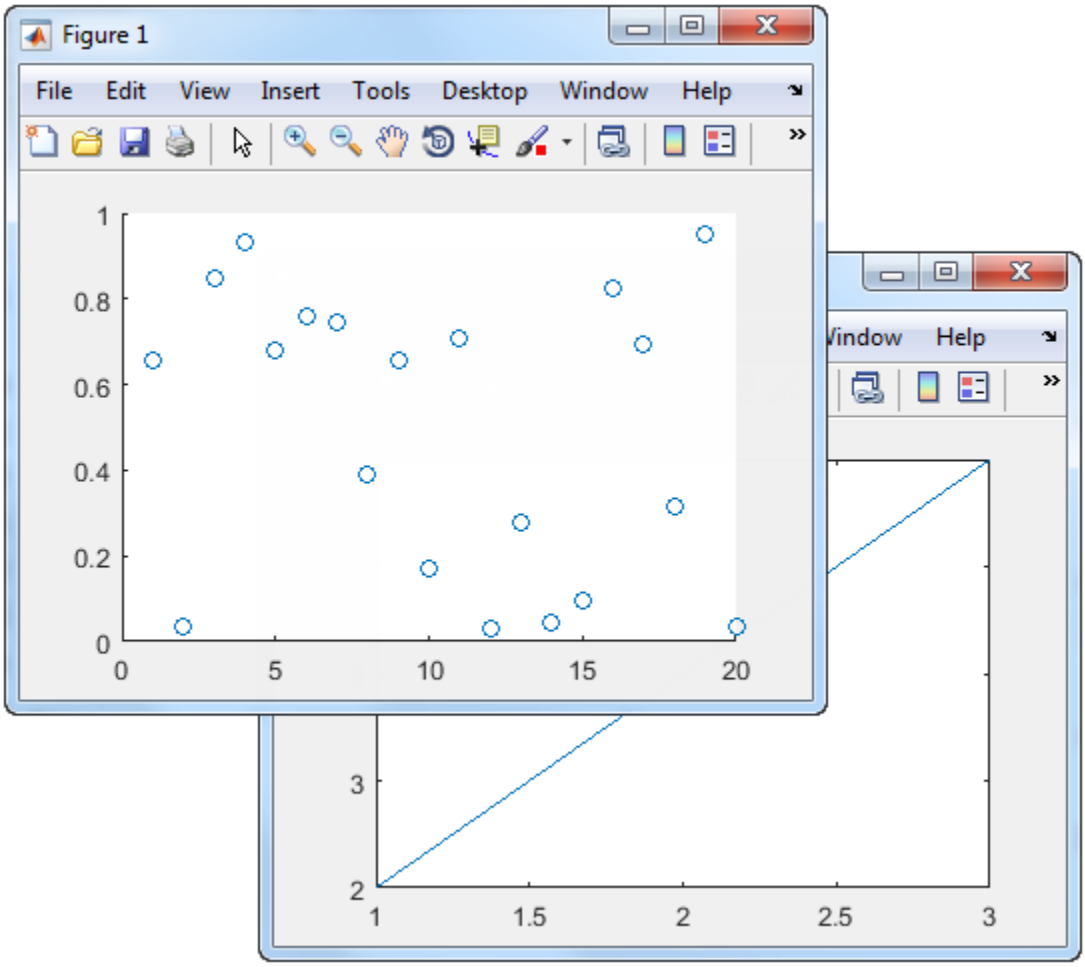
Create two figures, and then create a line plot. By default, the `plot` command targets the current figure.

```
f1 = figure;
f2 = figure;
plot([1 2 3],[2 4 6]);
```



Set the current figure to `f1`, so that it is the target for the next plot. Then create a scatter plot.

```
figure(f1);
scatter((1:20),rand(1,20));
```



Input Arguments

f — Target figure
`Figure` object

Target figure, specified as a `Figure` object.

n — Target figure number
scalar integer value

Target figure number, specified as a scalar integer value. When you specify this argument, MATLAB searches for an existing figure in which the Number property is equal to `n`. If no figure exists with that property value, MATLAB creates a new figure and sets its `Number` property to `n`. By default, the `Number` property value is displayed in the title of the figure.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments as `Name1,Value1,...,NameN,ValueN`.

Example: `figure('Color','white')` creates a figure with a white background.

Note The properties listed here are only a subset. For a full list, see Figure.

`Name` — Name
`''` (default) | character vector | string scalar

Name of the figure, specified as a character vector or a string scalar.

Example: `figure('Name','Results')` sets the name of the figure to `'Results'`.

By default, the name is `'Figure n'`, where *n* is an integer. When you specify the `Name` property, the title of the figure becomes `'Figure n: name'`. If you want only the `Name` value to appear, set `IntegerHandle` or `NumberTitle` to `'off'`.

`Color` — Background color
RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. If you specify `'none'`, the background color appears black on screen, but if you print the figure, the background prints as though the figure window is transparent.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

Data Types: double | char

Position — Location and size of drawable area
[left bottom width height]

Location and size of the drawable area, specified as a vector of the form [left bottom width height]. This area excludes the figure borders, title bar, menu bar, and tool bars.

This table describes each element in the Position vector.

Element	Description
left	Distance from the left edge of the primary display to the inner left edge of the window. This value can be negative on systems that have more than one monitor. If the figure is docked, then this value is relative to the Figure panel within the MATLAB desktop.
bottom	Distance from the bottom edge of the primary display to the inner bottom edge of the window. This value can be negative on systems that have more than one monitor. If the figure is docked, then this value is relative to the Figure panel within the MATLAB desktop.
width	Distance between the right and left inner edges of the figure.
height	Distance between the top and bottom inner edges of the window.

All measurements are in units specified by the Units property.

You cannot specify the figure Position property when the figure is docked.

In MATLAB Online, the bottom and left elements of the Position vector are ignored.

To place the full window, including the borders, title bar, menu bar, tool bars, use the OuterPosition property.

Note The Windows operating system enforces a minimum window width and a maximum window size. If you specify a figure size outside of those limits, the displayed figure will conform to the limits instead of the size you specified.

`Units` — Units of measurement

`'pixels'` (default) | `'normalized'` | `'inches'` | `'centimeters'` | `'points'` | `'characters'`

Units of measurement, specified as one of the values from this table.

Units Value	Description
`'pixels'` (default)	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.
`'normalized'`	These units are normalized with respect to the parent container. The lower-left corner of the container maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'points'`	Points. One point equals 1/72nd of an inch.

Units Value	Description
`'characters'`	These units are based on the default uicontrol font of the graphics root object: • Character width = width of the letter `x`. • Character height = distance between the baselines of two lines of text. To access the default uicontrol font, use `get(groot,'defaultuicontrolFontName')` or `set(groot,'defaultuicontrolFontName')`.

MATLAB measures all units from the lower left corner of the parent object.

This property affects the `Position` property. If you change the `Units` property, consider returning its value to the default value after completing your computation to avoid affecting other functions that assume the default value.

The order in which you specify the `Units` and `Position` properties has these effects:

- If you specify the `Units` before the `Position` property, then MATLAB sets `Position` using the units you specify.
- If you specify the `Units` property after the `Position` property, MATLAB sets the position using the default `Units`. Then, MATLAB converts the `Position` value to the equivalent value in the units you specify.

More About

Current Figure

The current figure is the target for graphics commands such as `axes` and `colormap`. Typically, it is the last figure created or the last figure clicked with the mouse. The `gcf` command returns the current figure.

See Also

Functions
axes | cla | clf | gca | gcf | shg

Properties
Figure

Introduced before R2006a

Figure Properties

Control appearance and behavior of figure window

Note **The properties listed here are valid for figures used in GUIDE or figures created with the `figure` function.** For figures created with the `uifigure` function, see UI Figure Properties.

Description

Figures are containers for graphics or user interface components. Figure properties control the appearance and behavior of a particular instance of a figure. To modify aspects of a figure, change property values.

Starting in R2014b, you can use dot notation to query and set properties.

```
f = figure;
u = f.Units;
f.Units = 'inches';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Window Appearance

`MenuBar` — Figure menu bar display

`'figure'` (default) | `'none'`

Figure menu bar display, specified as `'figure'` or `'none'`. The `MenuBar` property enables you to display or hide the default menus at the top of a figure window. Specify `'figure'` to display the menu bar. Specify `'none'` to hide it.

This property affects only default menus, and does not affect menus defined with the `uimenu` command.

Menu bars do not appear in figures whose `WindowStyle` property is set to `'Modal'`. If a figure containing uimenu children is changed to `'Modal'`, the uimenu children still exist

in the `Children` property of the figure. However, the uimenus do not display while `WindowStyle` is set to `'Modal'`.

Note If you do not want to display the default menus in the figure, then set this property to `'none'` when you create the figure.

`ToolBar` — Figure toolbar display

`'auto'` (default) | `'figure'` | `'none'`

Figure toolbar display, specified as one of the following:

- `'auto'` — Uses the same value as the `MenuBar` property.
- `'figure'` — Toolbar displays.
- `'none'` — Toolbar does not display.

This property affects only the default toolbar. It does not affect other toolbars such as, the Camera Toolbar or Plot Edit Toolbar. Selecting **Figure Toolbar** from the figure **View** menu sets this property to `'figure'`.

Toolbars do not appear in figures whose `WindowStyle` property is set to `'Modal'`. If a figure containing a toolbar is changed to `'Modal'`, the tool bar children still exist in the `Children` property of the figure. However, the toolbar does not display while `WindowStyle` is set to `'Modal'`.

Note If you want to hide the default tool bar, then set this property to `'none'` when you create the figure.

`DockControls` — Interactive figure docking

`'on'` (default) | `'off'`

Interactive figure docking, specified as one of the following:

- `'on'` — Figure can be docked in the MATLAB desktop. The **Desktop** > **Dock Figure** menu item and the Dock Figure button in the menu bar are enabled.
- `'off'` — MATLAB disables the **Desktop** > **Dock Figure** menu item and does not display the figure dock button.

You cannot set the `DockControls` property to `'off'` if the `WindowStyle` is set to `'docked'`.

Setting the `DockControls` property is not supported in MATLAB Online.

`Color` — Background color

RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. If you specify `'none'`, the background color appears black on screen, but if you print the figure, the background prints as though the figure window is transparent.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Data Types: `double` | `char`

`WindowStyle` — Window style
`'normal'` (default) | `'modal'` | `'docked'`

Window style, specified as one of the following:

- `'normal'` — The figure window is independent of other windows, and the other windows are accessible while the figure is displaying.
- `'modal'` — The figure displays on top of all existing figure windows, making them inaccessible as long as the top figure exists and remains modal. However, any new figures created after a modal figure will display.

  When multiple modal windows exist, the most recently created window keeps focus and stays above all other windows until it becomes invisible, or is returned to a normal window style, or is deleted. At that time, focus reverts to the window that last had focus.
- `'docked'` — The figure displays in the desktop or a document window. When the `WindowStyle` property is set to `'docked'`, you cannot set the `DockControls` property to `'off'`. The `'docked'` option is not supported in MATLAB Online.

Note These are some important characteristics of the `WindowStyle` property and some recommended best practices:

- When you create UI windows, always specify the `WindowStyle` property. If you also want to set the `Resize`, `Position`, or `OuterPosition` properties of the figure, then set the `WindowStyle` property first.
- You can change the `WindowStyle` property of a figure at any time, including when the figure is visible and contains children. However on some systems, setting this property might cause the figure to flash or disappear and reappear, depending on the system's implementation of normal and modal windows. For best visual results, set the `WindowStyle` property at creation time or when the figure is invisible.
- Calling `reset` on a figure does not change the value of the `WindowStyle` property.

Modal Window Style Behavior

When `WindowStyle` is set to `'modal'`, the figure window traps all keyboard and mouse actions over all MATLAB windows as long as the windows are visible. Windows belonging to applications other than MATLAB are unaffected.

Typing **Ctrl+C** when a modal figure has focus causes that figure to revert to a `'normal'` `WindowStyle` property setting. This allows the user to type at the command line.

Figures with the `WindowStyle` property set to `'modal'` and the `Visible` property set to `'off'` do not behave modally until MATLAB makes them visible. Therefore, you can hide a modal window for later reuse, instead of destroying it.

Modal figures do not display menu children, built-in menus, or toolbars. But, it is not an error to create menus in a modal figure or to change the `WindowStyle` property setting to `'modal'` on a figure with menu children. The `Menu` objects exist and the figure retains them. If you reset the figure's `WindowStyle` property to `'normal'`, the menus display.

`WindowState` — Window state
`'normal'` (default) | `'minimized'` | `'maximized'` | `'fullscreen'`

Window state, specified as one of these values:

- `'normal'` — The window displays in a normal state.
- `'minimized'` — The window is collapsed, but you can still execute commands to get or set its properties, to add children, or to create plots in the window.
- `'maximized'` — The window fills the screen. The minimize, restore, and close buttons provided by the operating system appear in the corner of the window. If the window has menus and tool bars in the normal state, they are present in this state.

- `'fullscreen'` — The window fills the screen. However, the minimize, restore, and close buttons provided by the operating system are hidden. If the window has menus and tool bars in the normal state, they are present in this state.

Clicking the minimize, maximize, or restore button provided by the operating system sets the `WindowState` property accordingly. Pressing **Ctrl+F11** (Windows and Linux) or **Ctrl +Command+f** (Mac OS) toggles the `'fullscreen'` state.

Setting the `WindowState` property on a docked figure or in MATLAB Online is not supported.

Position

`Position` — Location and size of drawable area

`[left bottom width height]`

Location and size of the drawable area, specified as a vector of the form `[left bottom width height]`. This area excludes the figure borders, title bar, menu bar, and tool bars.

This table describes each element in the `Position` vector.

Element	Description
`left`	Distance from the left edge of the primary display to the inner left edge of the window. This value can be negative on systems that have more than one monitor. If the figure is docked, then this value is relative to the Figure panel within the MATLAB desktop.
`bottom`	Distance from the bottom edge of the primary display to the inner bottom edge of the window. This value can be negative on systems that have more than one monitor. If the figure is docked, then this value is relative to the Figure panel within the MATLAB desktop.
`width`	Distance between the right and left inner edges of the figure.
`height`	Distance between the top and bottom inner edges of the window.

All measurements are in units specified by the `Units` property.

You cannot specify the figure `Position` property when the figure is docked.

In MATLAB Online, the `bottom` and `left` elements of the `Position` vector are ignored.

To place the full window, including the borders, title bar, menu bar, tool bars, use the `OuterPosition` property.

Note The Windows operating system enforces a minimum window width and a maximum window size. If you specify a figure size outside of those limits, the displayed figure will conform to the limits instead of the size you specified.

`Units` — Units of measurement

`'pixels'` (default) | `'normalized'` | `'inches'` | `'centimeters'` | `'points'` | `'characters'`

Units of measurement, specified as one of the values from this table.

Units Value	Description
`'pixels'` (default)	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.
`'normalized'`	These units are normalized with respect to the parent container. The lower-left corner of the container maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'points'`	Points. One point equals 1/72nd of an inch.

Units Value	Description
`'characters'`	These units are based on the default uicontrol font of the graphics root object: • Character width = width of the letter `x`. • Character height = distance between the baselines of two lines of text. To access the default uicontrol font, use `get(groot,'defaultuicontrolFontName')` or `set(groot,'defaultuicontrolFontName')`.

MATLAB measures all units from the lower left corner of the parent object.

This property affects the `Position` property. If you change the `Units` property, consider returning its value to the default value after completing your computation to avoid affecting other functions that assume the default value.

The order in which you specify the `Units` and `Position` properties has these effects:

- If you specify the `Units` before the `Position` property, then MATLAB sets `Position` using the units you specify.
- If you specify the `Units` property after the `Position` property, MATLAB sets the position using the default `Units`. Then, MATLAB converts the `Position` value to the equivalent value in the units you specify.

`InnerPosition` — Location and size of drawable area
`[left bottom width height]`

Location and size of the drawable area, specified as a vector of the form `[left bottom width height]`. This area excludes the figure borders, title bar, menu bar, and tool bars. All measurements are in units specified by the `Units` property.

This property value is identical to the `Position` property value.

`OuterPosition` — Location and size of outer bounds
`[left bottom width height]`

Location and size of the outer bounds, specified as a vector of the form `[left bottom width height]`. This property defines the region enclosed by the figure's outer bounds (which includes the borders, title bar, menu bar, and tool bars).

This table describes each element in the vector.

Element	Description
`left`	Distance from the left edge of the primary display to the outer left edge of the figure window. This value can be negative on systems that have more than one monitor. If the figure is docked, then this value is relative to the Figure panel within the MATLAB desktop.
`bottom`	Distance from the bottom edge of the primary display to the outer bottom edge of the figure window. This value can be negative on systems that have more than one monitor. If the figure is docked, then this value is relative to the Figure panel within the MATLAB desktop.
`width`	Distance between the right and left outer edges of the figure.
`height`	Distance between the top and bottom outer edges of the figure.

All measurements are in units specified by the `Units` property.

You cannot specify the figure `OuterPosition` property when the figure is docked.

Note The Windows operating system enforces a minimum window width and a maximum window size. If you specify a figure size outside of those limits, the displayed figure will conform to the limits instead of the size you specified.

`Clipping` — Clipping of child components (not recommended)

`'on'` (default) | `'off'`

This property has no effect on figures.

`Resize` — Resizable

`'on'` (default) | `'off'`

Resizable figure, specified as `'on'` or `'off'`. When this property is set to `'on'`, the figure is resizable. Otherwise, it is not resizable.

Plotting

`Colormap` — Color map for axes content of figure

parula (default) | `m`-by-`3` array of RGB triplets

Color map for axes content of a figure, specified as an `m`-by-`3` array of RGB (red, green, blue) triplets that define `m` individual colors.

Example: `figure('Colormap',[1 0 1; 0 0 1; 1 1 0])` sets the color map to three colors: magenta, blue, and yellow.

MATLAB accesses these colors by their row number.

Color maps affect the rendering of objects created with the `surface`, `image`, and `patch` functions, but generally do not affect other graphics objects.

When you set the `Colormap` property of the figure, then axes and charts in the figure use the same colormap.

`Alphamap` — Transparency map for `Axes` content

array of 64 values from `0` to `1` (default) | array of finite alpha values from `0` to `1`

Transparency map for `Axes` content, specified as an array of finite alpha values that progress linearly from `0` to `1`. The size of the array can be m-by-1 or 1-by-m. MATLAB accesses alpha values by their index in the array. Alphamaps can be any length.

Alphamaps affect the rendering of objects created with the `surface`, `image`, and `patch` functions, but do not affect other graphics objects.

When you set the `Alphamap` property of the figure, then axes and charts in the figure use the same alpha map.

`NextPlot` — Directive on how to add next plot

`'add'` (default) | `'new'` | `'replace'` | `'replacechildren'`

Directive on how to add next plot, specified as `'add'`, `'new'`, `'replace'`, or `'replacechildren'`.

This table describes the effects of each value.

Property Value	Effect
`'new'`	Creates a new figure and uses it as the current figure.
`'add'`	Adds new graphics objects without clearing or resetting the current figure.
`'replacechildren'`	Removes all axes objects who are not hidden before adding new objects. Does not reset figure properties. Equivalent to using the `clf` command.
`'replace'`	Removes all axes objects and resets figure properties to their defaults before adding new graphics objects. Equivalent to using the `clf reset` command.

Consider using the `newplot` function to handle the `NextPlot` property. For more information, see the axes `NextPlot` property and “Prepare Figures and Axes for Graphs”.

`Renderer` — Renderer
`'opengl'` (default) | `'painters'`

Renderer for screen display and printing, specified as one of these values:

- `'opengl'` — OpenGL® renderer. This option enables MATLAB to access graphics hardware if it is available on your system. The OpenGL renderer displays objects sorted in front to back order, as seen on the monitor. Lines always draw in front of faces when at the same location on the plane of the monitor.
- `'painters'` — Painters renderer. This option works well for axes in a 2-D view. In 2-D, the Painters renderer sorts graphics objects by child order (order specified). In 3-D, the Painters renderer sorts objects in front to back order. However, it might not correctly draw intersecting polygons in 3-D.

Note The `'zbuffer'` option has been removed. Use `'opengl'` or `'painters'` instead.

Setting the `Renderer` property is not supported in MATLAB Online.

OpenGL Hardware and Software Implementations

OpenGL is available on all computers that run MATLAB since a software version of OpenGL is built-into MATLAB. However, if you have graphics hardware that supports a hardware-accelerated version of OpenGL, then MATLAB automatically uses the hardware-accelerated version to increase performance.

In some cases, MATLAB automatically uses software OpenGL even if a hardware version is available. For example, MATLAB uses the software version if it detects graphics hardware with known driver issues or detects that you are using a virtual machine or remote desktop on Windows.

MATLAB issues a warning if it cannot find a usable OpenGL library.

Software OpenGL Selection

To switch from hardware to software OpenGL, do the following:

- On Linux systems, start MATLAB with the command `matlab -softwareopengl`.
- On Windows systems, execute the command `opengl software` in MATLAB or start MATLAB with the command `matlab -softwareopengl`.
- On Macintosh systems, software OpenGL is not supported.

The following software versions are available:

- On Linux systems, MATLAB uses the software implementation of OpenGL that is included in the MATLAB distribution.
- On Windows, OpenGL is available as part of the operating system. If you experience problems with OpenGL, contact your graphics driver vendor to obtain the latest qualified version of OpenGL.
- On Macintosh systems, software OpenGL is not available.

Determine OpenGL Library Version

To determine the version and vendor of the OpenGL library that MATLAB is using on your system, call the `rendererinfo` function. For example, this command gets the information for the current axes and stores it in a structure called `info`.

```
info = rendererinfo(gca)
```

This structure also provides the name of the graphics renderer in the `GraphicsRenderer` field. For example, if MATLAB is using hardware-accelerated

OpenGL, the field returns `'OpenGL Hardware'`. If MATLAB is using software OpenGL, the field returns `'OpenGL Software'`.

XServer Connection Lost

When using Linux, if there is a break in the connection to the XServer, MATLAB can crash with a segmentation violation . If this happens, ensure that the system has the latest XServer installed.

On a Linux system, you also can try upgrading the OpenGL driver or starting MATLAB with software OpenGL using this command:

```
matlab -softwareopengl
```

RendererMode — Renderer selection

`'auto'` (default) | `'manual'`

Renderer selection, specified as:

- `'auto'` — MATLAB selects the rendering method for printing and screen display based on the size and complexity of the graphics objects in the figure.
- `'manual'` — MATLAB uses the renderer specified with the `Renderer` property.

MATLAB sets the `RendererMode` property to `'manual'` if you explicitly set the `Renderer` property to `'painters'` or `'opengl'`.

GraphicsSmoothing — Axes graphics smoothing

`'on'` (default) | `'off'`

Axes graphics smoothing, specified as `'on'` or `'off'`. Smoothing reduces the appearance of jagged lines in an axes graphic. MATLAB applies a smoothing technique to an axes graphic (and the axes rulers) if `GraphicsSmoothing` is set to `'on'`, and either of these conditions is true:

- The `Renderer` property is set to `'painters'`.
- The `Renderer` property is set to `'opengl'` and your hardware card supports OpenGL.

If your axes graphic contains mostly vertical or horizontal lines, consider setting the `GraphicsSmoothing` property to `'on'` and the line or lines `AlignVertexCenters` property to `'on'`. The smoothing technique sacrifices some sharpness for smoothness, which might be particularly noticeable in such graphics.

Note Graphics smoothing has no affect on text. MATLAB smooths text regardless of the value of the `GraphicsSmoothing` property.

Setting the `GraphicsSmoothing` property is not supported in MATLAB Online.

Printing and Exporting

`PaperPosition` — Figure size and location on page when printing or saving

four-element vector of the form `[left bottom width height]`

Figure size and location on page when printing or saving, specified as a four-element vector of the form `[left bottom width height]`.

- `left` and `bottom` values — Control the distance from the lower left corner of the page to the lower left corner of the figure. These values are ignored when saving a figure to a nonpage format, such as a PNG or EPS format.
- `width` and `height` values — Control the figure size. If the width and height values are too large, then the figure might not reach the specified size. If the figure does not reach the specific size, then any UI components on the figure, such as uicontrols or a uitable, might not save or print as expected.

The `PaperUnits` property determines the units of measurement of the `PaperPosition` values. Consider setting the `PaperUnits` property to `'normalized'`. This setting enables MATLAB to automatically size the figure to occupy the same relative amount of the printed page, regardless of the page size.

Example: `figure('PaperPosition',[.25 .25 8 6])` set the figure's size and location for printing to `[.25 .25 8 6]`.

`PaperPositionMode` — Directive to use displayed figure size when printing or saving

`'auto'` (default) | `'manual'`

Directive to use displayed figure size when printing or saving, specified as either `'auto'` or `'manual'`.

- `'auto'` — Printed or saved figure size matches the displayed figure size. The `width` and `height` values of the `PaperPosition` property equal the figure size on the display. The `left` and `bottom` values center the figure. If the figure size changes on the display, the `PaperPosition` property automatically updates to the appropriate size and location values.

- `'manual'` — Printed or saved figure size might not match the displayed figure size. Use this option if you want to print or save the figure using a size that differs from the display, or if you do not want the figure centered on the printed or saved page. Set the `PaperPosition` property to the desired size and location. If the figure size changes on the display, the `PaperPosition` property does not automatically update.

To generate output that has the same size and resolution (DPI) as the displayed figure, set the `PaperPositionMode` property of the figure to `'auto'` and save the figure using `print` with the `-r0` option. The `-r0` option ensures that the output resolution is the same as the display resolution. If the resolutions are different, then the generated output size matches the displayed figure size in measured units (inches, centimeters, points), but not in pixels. For example, if the display resolution is 100 DPI, then a 4-by-5 inch figure is 400-by-500 pixels. If the output resolution is 200 DPI, then the printed or saved figure is the same size in inches, but 800-by-1000 pixels.

Note Starting in R2016a, the default value is `'auto'`. Previously, the default value was `'manual'`.

To change the default value, use one of these techniques.

- Set a print preference. Print preferences persist across MATLAB sessions. You can set the print preference to either `'auto'` or `'manual'`, for example:

  ```
  matlab.graphics.internal.setPrintPreferences('DefaultPaperPositionMode','manual')
  ```

  To query the current print preference value, use the following command. If you set a preference, the command returns `'auto'` or `'manual'`. If you did not set a preference, the command returns `'unset'`.

  ```
  matlab.graphics.internal.getPrintPreferences
  ```

- Set the default value on the root object. This option affects only new figures in the current MATLAB session, for example:

  ```
  set(groot,'defaultFigurePaperPositionMode','manual')
  ```

`PaperSize` — Custom page size

two-element vector of the form `[width height]`

Custom page size when printing the figure or saving it to a paged format (PDF and PostScript® formats), specified as a two-element vector of the form `[width height]`. In

the United States, the default value is `[8.5 11]`. In Europe and Asia, the default value is `[21 29.7]`.

Note If you are saving the figure to a file, the `PaperSize` property only affects PDF and PostScript file formats. Other file formats ignore this property. Use the `PaperPosition` property to control the size of the saved figure.

The `PaperUnits` property determines the units of measurement for the `PaperSize` property. You cannot set the `PaperSize` property if the `PaperUnits` property is set to `'normalized'`. Attempting to do so results in an error.

Specifying the `PaperSize` property sets the `PaperType` property to the corresponding type, if one exists, or to `'custom'` otherwise.

`PaperUnits` — Units used for `PaperSize` and `PaperPosition`
`'inches'` | `'centimeters'` | `'normalized'` | `'points'`

Units used for `PaperSize` and `PaperPosition`, specified as one of these values:

- `'inches'` — Value in inches. This is the default when the locale is the United States.
- `'normalized'` — Normalized units. The lower left corner of the page maps to `(0,0)` and the upper right corner maps to `(1,1)`.
- `'centimeters'` — Value in centimeters. This is the default when the locale is Europe or Asia.
- `'points'` — Value in points. One point equals 1/72 of an inch.

Note If you change the value of the `PaperUnits` property, it is good practice to return the property to its original value after completing your computation so as not to affect other functions that assume the `PaperUnits` property has not changed.

`PaperOrientation` — Orientation of page
`'portrait'` (default) | `'landscape'`

Orientation of page when printing figure or saving it to a paged format (PDF and PostScript formats), specified as one of these values:

- `'portrait'` — Orient the longest page dimension vertically.

- `'landscape'` — Orient the longest page dimension horizontally.

See the `orient` function for more information.

Note When using **File** > **Save As**, only PDF and full-page PS formats use the `PaperOrientation` property. Other formats ignore these values.

PaperType — Standard page sizes
`'usletter'` | `'uslegal'` | `'tabloid'` | `'a0'` | `'a1'` | `'a2'` | `'a3'` | `...`

Standard page sizes when printing the figure or saving it to a paged format (PDF and PostScript formats), specified as one of the values in this table. Specifying the `PaperType` property sets the `PaperSize` property to the corresponding page size.

Value	Page Size (Width x Height)
`'usletter'`	8.5-by-11 in (default in US)
`'uslegal'`	8.5-by-14 in
`'tabloid'`	11-by-17 in
`'a0'`	84.1-by-118.9 cm
`'a1'`	59.4-by-84.1 cm
`'a2'`	42-by-59.4 cm
`'a3'`	29.7-by-42 cm
`'a4'`	21-by-29.7 cm (default in Europe and Asia)
`'a5'`	14.8-by-21 cm
`'b0'`	102.9-by-145.6 cm
`'b1`	72.8-by-102.8 cm
`'b2'`	51.4-by-72.8 cm
`'b3'`	36.4-by-51.4 cm
`'b4'`	25.7-by-36.4 cm
`'b5'`	18.2-by-25.7 cm
`'arch-a'`	9-by-12 in
`'arch-b'`	12-by-18 in

Value	Page Size (Width x Height)
`'arch-c'`	18-by-24 in
`'arch-d'`	24-by-36 in
`'arch-e'`	36-by-48 in
`'a'`	8.5-by-11 in
`'b'`	11-by-17 in
`'c'`	17-by-22 in
`'d'`	22-by-34 in
`'e'`	34-by-43 in
`'custom'`	Custom page size. Specifying a non-standard page size using the `PaperSize` property sets `PaperPosition` to this value.

`InvertHardcopy` — Figure background color when printing or saving
`'on'` (default) | `'off'`

Figure background color when saving or printing, specified as one of these values:

- `'on'` — Change the figure background and axes background colors to white. Text and other aspects of the figure also might change color to improve the appearance of the generated output.
- `'off'` — Use the same colors as the colors on the display. To change the figure background color on the display, use the `Color` property of the figure. To change the axes background color, use the `Color` property of the axes.

Mouse Pointer

`Pointer` — Pointer symbol
`'arrow'` (default) | `'ibeam'` | `'crosshair'` | `'watch'` | `'topl'` | `'custom'` | ...

Pointer symbol, specified as one of the symbol names in the following table or as `'custom'`. The appearance of the symbol is operating-system dependent.

Symbol Name	Resulting Symbol (System Dependent)
`'arrow'`	

Symbol Name	Resulting Symbol (System Dependent)
`'ibeam'`	
`'crosshair'`	
`'watch'` (busy system)	
`'topl'` or `'botr'`	
`'topr'` or `'botl'`	
`'circle'`	
`'cross'`	
`'fleur'`	
`'left'` or `'right'`	
`'top'` or `'bottom'`	
`'hand'`	

Note The `'fullcrosshair'` option was removed in R2014b.

Custom Pointer Symbol

To create a custom pointer symbol, set the `Pointer` property to `'custom'` and use the `PointerShapeCData` property to define the symbol. See the `PointerShapeCData` property for more information.

`PointerShapeCData` — Custom pointer symbol

32-by-32 matrix | 16-by-16 matrix

Custom pointer symbol, specified as a 32-by-32 matrix (for a 32-by-32 pixel pointer) or as a 16-by-16 matrix (for a 16-by-16 pixel pointer). The figure uses this pointer symbol when you set the `Pointer` property to `'custom'`.

Each element in the matrix defines the brightness level for 1 pixel in the pointer. Element `(1,1)` of the matrix corresponds to the pixel in the upper left corner in the pointer. Set the matrix elements to one of these values:

- `1` — Black pixel.
- `2` — White pixel.
- `NaN` — Transparent pixel, such that underlying screen shows through.

`PointerShapeHotSpot` — Active pixel of pointer

`[1 1]` (default) | two-element vector

Active pixel of the pointer, specified as a two-element vector. The vector contains the row and column indices of a particular element in the `PointerShapeCData` matrix that corresponds to the desired active pixel. The default value of `[1 1]` corresponds to the pixel in the upper left corner of the pointer.

If you specify a value outside the range of the `PointerShapeCData` matrix, then the pointer uses the default active pixel of `[1 1]` instead.

This property applies only when the `Pointer` property is set to `'custom'`.

Interactivity

`CurrentAxes` — Target axes in current figure

`Axes` object | `PolarAxes` object | graphics object

Target axes in the current figure, specified as an `Axes` object, a `PolarAxes` object, or a graphics object such as a `HeatmapChart`.

In all figures for which axes children exist, there is always a current axes. The current axes does not have to be the topmost axes, and setting an axes to be the current axes does not restack it above all other axes. If a figure contains no axes, the `get(gcf,'CurrentAxes')` command returns an empty array.

Query the `CurrentAxes` property to get the current axes object without forcing the creation of an axes if one does not exist.

`CurrentObject` — Current object
child object of figure

Current object, returned as a child object of the figure. MATLAB sets the `CurrentObject` property to the last object clicked in the figure. This object is the front-most object in the view.

The `HitTest` property of the child object controls whether that object can become the `CurrentObject`.

Clicking an object whose `HandleVisibility` property is `'off'` sets the `CurrentObject` property to an empty `GraphicsPlaceholder` array. To avoid returning the empty array when users click a hidden object, set `HitTest` property of the hidden object to `'off'`.

Use the `gco` command as an alternative for getting the value of this property.

`CurrentPoint` — Current point
two-element vector

Current point, returned as a two-element vector. The vector contains the (*x*, *y*) coordinates of the mouse pointer, measured from the lower-left corner of the figure. The values are in units specified by the `Units` property.

The coordinates update when you do any of the following:

- Press the mouse button within the figure.
- Release the mouse button after pressing it within the figure.
- Press the mouse button within the figure, and then release it outside the figure.
- Rotate the scroll wheel within the figure.
- Move the mouse within the figure (without pressing any buttons), provided that the `WindowButtonMotionFcn` property is not empty.

If the figure has a callback that responds to mouse interactions, and you trigger that callback faster than the system can execute the code, the coordinates might not reflect the actual location of the pointer. Instead, they are the location when the callback began execution.

If you use the `CurrentPoint` property to plot points, the coordinate values might contain rounding error.

`CurrentCharacter` — Current character
`''` (default) | character

Current character, returned as the character of the last key pressed. This property updates when the figure has focus while the user presses a key.

`Selected` — Selection state (not recommended)
`'off'` (default) | `'on'`

Note The behavior of the `Selected` property changed in R2014b, and it is not recommended. It no longer has any effect on `Figure` objects. This property might be removed in a future release.

`SelectionHighlight` — Display of selection handles (not recommended)
`'on'` (default) | `'off'`

Note Use of the `SelectionHighlight` property is not recommended. This property might be removed in a future release.

This property has no effect on figures.

`SelectionType` — Mouse selection type
`'normal'` (default) | `'extend'` | `'alt'` | `'open'`

Mouse selection type, returned as `'normal'`, `'extend'`, `'alt'`, or `'open'`. This property provides information about the last mouse button press that occurred in the figure.

This table lists the possible `SelectionType` values and the user actions that produce those values.

Value	Corresponding Action
`'normal'`	Click the left mouse button.
`'extend'`	Any of the following: • **Shift**-click the left mouse button. • Click the middle mouse button. • Click both left and right mouse buttons.
`'alt'`	Either of the following: • **Control**-click the left mouse button. • Click the right mouse button.
`'open'`	Double-click any mouse button.

`UIContextMenu` — Context menu
empty `GraphicsPlaceholder` array (default) | uicontextmenu object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when the user right-clicks on a UI component. Create the context menu using the `uicontextmenu` function.

`Visible` — Figure visibility
`'on'` (default) | `'off'`

Figure visibility, specified as `'on'` or `'off'`. The `Visible` property determines whether the figure displays on the screen. If the `Visible` property of a figure is set to `'off'`, the entire figure is invisible, but you can still specify and access its properties.

Changing the size of an invisible figure triggers the `SizeChangedFcn` callback when the figure becomes visible.

Note Changing the `Visible` property of a figure does *not* change the `Visible` property of its child components even though hiding the figure prevents its children from displaying.

Common Callbacks

`ButtonDownFcn` — Button down callback
`''` (default) | function handle | cell array | character vector

Button down callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes whenever the user clicks a blank area of the figure.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

Use the `SelectionType` property to determine whether the user pressed modifier keys.

`CreateFcn` — Component creation function
`''` (default) | function handle | cell array | character vector

Component creation function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

This property specifies a callback function to execute when MATLAB creates the component. MATLAB initializes all component property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Use the `gcbo` function in your `CreateFcn` code to get the component object that is being created.

Setting the `CreateFcn` property on an existing component object has no effect.

`DeleteFcn` — Component deletion function
`''` (default) | function handle | cell array | character vector

Component deletion function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

The `DeleteFcn` property specifies a callback function to execute when MATLAB deletes the component (for example, when the user closes the window). MATLAB executes the `DeleteFcn` callback before destroying the properties of the component object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

Use the `gcbo` function in your `DeleteFcn` code to get the component object that is being deleted.

Keyboard Callbacks

`KeyPressFcn` — Key-press callback

`''` (default) | function handle | cell array | character vector

Key-press callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback function executes when the user presses a key while the figure or a child object has focus. If the user presses a key on a `UIControl` or `Table` component, the callback does not execute unless the `Enable` property is set to `'off'`.

If the user presses multiple keys at approximately the same time, MATLAB detects all the keys.

The `KeyPressFcn` callback can access specific information about the user’s interaction with the keyboard. MATLAB passes this information in a `KeyData` object as the second argument to your callback function. This table lists the properties of this object.

Property	Description
`Character`	The character that appears as a result of pressing the key or keys. Pressing certain keys or modifying keys with the **Ctrl** key might put unprintable characters in the `Character` property. Pressing certain keys alone (such as **Ctrl**, **Alt**, **Shift**) does not generate `Character` data.
`Modifier`	Cell array containing the names of the modifier keys pressed, such as **control**, **alt**, or **shift**. If no modifier keys are pressed, the cell array is empty.
`Key`	The key pressed, identified by the lowercase label on the key, or a descriptive word such as `'space'`.
`Source`	The object that has focus when the user presses the key.
`EventName`	`'KeyPress'`

For more information about specifying callback property values and using callback arguments, see “How to Specify Callback Property Values”.

The `CurrentCharacter` property also returns character information.

`KeyReleaseFcn` — Key-release callback

`''` (default) | function handle | cell array | character vector

Key-release callback, specified as one of these values

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback function executes when the user presses a key while the figure or a child object has focus. If the user releases a key on a `UIControl` or `Table` component, the callback does not execute unless the `Enable` property is set to `'off'`.

The `KeyReleaseFcn` callback can access specific information about the user’s interaction with the keyboard. MATLAB passes this information in a `KeyData` object as

the second argument to your callback function. This table lists the properties of this object.

Property	Description
`Character`	Character that corresponds to the key or keys that are released. Certain keys or key combinations might return unprintable characters in this property. Pressing and releasing certain keys alone (such as **Ctrl**, **Alt**, **Shift**) does not generate `Character` data.
`Modifier`	Cell array containing the names of the modifier keys released, such as **control**, **alt**, or **shift**. If no modifier keys are released, the cell array is empty.
`Key`	Key released, identified by the (lowercase) label on the key, or a descriptive word such as `'space'`.
`Source`	Object that has focus when the key is released.
`EventName`	`'KeyRelease'`

For more information about specifying callback property values and using callback arguments, see “How to Specify Callback Property Values”.

Window Callbacks

`CloseRequestFcn` — Close request callback

`'closereq'` (default) | function handle | cell array | character vector

Close request callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

This callback executes whenever a user attempts to close a figure window. You can, for example, display a dialog box to ask a user to confirm or cancel the close operation or to prevent users from closing a figure that contains a UI.

The basic mechanism is:

1. A user issues the `close` or `close all` command from the command line, closes the figure from the computer window manager menu, or closes the figure by quitting MATLAB.
2. The close operation executes the function defined by the figure `CloseRequestFcn` property. The default function is `closereq`.

   The `closereq` function unconditionally deletes the current figure, destroying the window. The `closereq` function takes advantage of the fact that the `close` command makes each figure specified as an argument the current figure before calling its respective close request function.

The `closereq` function honors the `ShowHiddenHandles` property setting during figure deletion and does not delete hidden figures.

Unless the close request function calls the `delete` or `close` function, MATLAB never closes the figure. (You can call `delete(f)` from the command line if you have created a window with a nondestructive close request function.)

Example: Code CloseRequestFcn to Display Dialog Box

This example shows how to code the close request function to display a question dialog box asking the user to confirm the close operation. Save the code to a writable folder on your system.

```
function my_closereq(src,callbackdata)
% Close request function
% to display a question dialog box
   selection = questdlg('Close This Figure?',...
      'Close Request Function',...
      'Yes','No','Yes');
   switch selection
      case 'Yes'
         delete(gcf)
      case 'No'
      return
   end
end
```

Now, create a figure specifying `my_closereq` for the `CloseRequestFcn`:

```
figure('CloseRequestFcn',@my_closereq)
```

Close the figure window and the question dialog box displays.

`SizeChangedFcn` — Size changed callback

`''` (default) | function handle | cell array | character vector

Size changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

Define this callback to customize the app layout when the size of this container changes (e.g., when the user resizes the window).

The `SizeChangedFcn` callback executes when:

- This container becomes visible for the first time.
- This container is visible while its size changes.
- This container becomes visible for the first time after its size changes. This situation occurs when the size changes while the container is invisible, and then it becomes visible later.

Other important points to consider when defining a `SizeChangedFcn` callback:

- Consider delaying the display of this container until after all the variables that the `SizeChangedFcn` uses are defined. This practice can prevent the `SizeChangedFcn` callback from returning an error. To delay the display of the container, set its `Visible` property to `'off'`. Then, set the `Visible` property to `'on'` after you define the variables that your `SizeChangedFcn` callback uses.
- If your app contains nested containers, they resize from the inside out.
- To access the container that is resizing from within the `SizeChangedFcn`, refer to the source object (the first input argument in the callback) or use the `gcbo` function.

Tip As an easy alternative to specifying a `SizeChangedFcn` callback, you can set the `Units` property of all the objects you put inside a container to `'normalized'`. Doing so makes those components scale proportionally with the container.

Example: Edit Field With Constant Height

Use the `SizeChangedFcn` callback to constrain the size of UI components. For instance, the following code creates a UI that has an edit field at the top of the window. As the user resizes the window, the `sbar` function restricts its height to 20 pixels. The function also sets the edit field's width to the width of the figure.

If you are using R2014b or later, use dot notation to set and query properties:

```
function myui
f = figure('Visible','off','SizeChangedFcn',@sbar);
u = uicontrol('Style','edit','Tag','StatusBar');
f.Visible = 'on';
    function sbar(src,callbackdata)
       old_units = src.Units;
       src.Units = 'pixels';
       sbar_units = u.Units;
       u.Units = 'pixels';
       fpos = src.Position;
       upos = [1 fpos(4) - 20 fpos(3) 20];
       u.Position = upos;
       u.Units = sbar_units;
       src.Units = old_units;
       u.Visible = 'on';
    end
end
```

If you are using R2014a or an earlier release, use this code instead.

```
function myui
f = figure('Visible','off','SizeChangedFcn',@sbar);
u = uicontrol('Style','edit','Tag','StatusBar');
set(f,'Visible','on');
    function sbar(src,callbackdata)
       old_units = get(src,'Units');
       set(src,'Units','pixels');
       sbar_units = get(u,'Units');
       set(u,'Units','pixels');
       fpos = get(src,'Position');
```

```
        upos = [1 fpos(4) - 20 fpos(3) 20];
        set(u,'Position',upos);
        set(u,'Units',sbar_units);
        set(src,'Units',old_units);
        set(u,'Visible','on');
    end
end
```

`WindowButtonDownFcn` — Window button down callback

`''` (default) | function handle | cell array | character vector

Window button down callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when the user clicks anywhere in the figure or one of its child objects. If the user clicks a `UIControl` or `Table` component, the callback does not execute unless the `Enable` property is set to `'off'`.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

`WindowButtonMotionFcn` — Window button motion callback

`''` (default) | function handle | cell array | character vector

Window button motion callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

Typically, this callback function executes whenever the user moves the pointer within the figure. However, if the user clicks and holds the mouse button and moves the pointer outside the figure, the `WindowButtonMotionFcn` callback executes while the pointer is outside the figure.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

Example: Simple Drawing App

This example shows how to create a drawing app that uses the `WindowButtonMotionFcn` callback. Copy and paste the following code into the editor and run it. Then use the mouse to draw a set of connected line segments:

- Click and drag inside the axes to draw the first line segment.
- Click again to define an anchor point for the line segment.
- Move the mouse to add another line segment.
- Right-click to end drawing mode.

```
function drawing_app
figure('WindowButtonDownFcn',@wbdcb)
ah = axes('SortMethod','childorder');
axis ([1 10 1 10])
title('Click and drag')
   function wbdcb(src,callbackdata)
     seltype = src.SelectionType;
     % This code uses dot notation to set properties
     % Dot notation runs in R2014b and later.
     % For R2014a and earlier: seltype = get(src,'SelectionType');
     if strcmp(seltype,'normal')
        src.Pointer = 'circle';
        cp = ah.CurrentPoint;
        % For R2014a and earlier:
        % set(src,'Pointer','circle');
        % cp = get(ah,'CurrentPoint');
        xinit = cp(1,1);
        yinit = cp(1,2);
        hl = line('XData',xinit,'YData',yinit,...
        'Marker','p','color','b');
        src.WindowButtonMotionFcn = @wbmcb;
        src.WindowButtonUpFcn = @wbucb;
        % For R2014a and earlier:
        % set(src,'WindowButtonMotionFcn',@wbmcb);
        % set(src,'WindowButtonUpFcn',@wbucb);

     end

        function wbmcb(src,callbackdata)
```

```
            cp = ah.CurrentPoint;
            % For R2014a and earlier:
            % cp = get(ah,'CurrentPoint');
            xdat = [xinit,cp(1,1)];
            ydat = [yinit,cp(1,2)];
            hl.XData = xdat;
            hl.YData = ydat;
            % For R2014a and earlier:
            % set(hl,'XData',xdat);
            % set(hl,'YData',ydat);
            drawnow
         end

         function wbucb(src,callbackdata)
            last_seltype = src.SelectionType;
            % For R2014a and earlier:
            % last_seltype = get(src,'SelectionType');
            if strcmp(last_seltype,'alt')
               src.Pointer = 'arrow';
               src.WindowButtonMotionFcn = '';
               src.WindowButtonUpFcn = '';
               % For R2014a and earlier:
               % set(src,'Pointer','arrow');
               % set(src,'WindowButtonMotionFcn','');
               % set(src,'WindowButtonUpFcn','');
            else
               return
            end
         end
   end
end
```

`WindowButtonUpFcn` — Window button-up callback

`''` (default) | function handle | cell array | character vector

Window button-up callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when the user releases the mouse button anywhere in the figure or one of its child objects. If the user releases the mouse button on a `UIControl` or `Table` component, the callback does not execute unless the `Enable` property is set to `'off'`.

The button-up actions are associated with button-down actions in the figure. The mouse pointer must be inside the figure when the button-down action occurs, but it can be outside the figure when the button-up action occurs.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

`WindowKeyPressFcn` — Window key-press callback

`''` (default) | function handle | cell array | character vector

Window key-press callback, specified as one of these values

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback function executes when the user presses a key while the figure or a child object has focus.

This callback function can access specific information about the user’s interaction with the keyboard. MATLAB passes this information in a `KeyData` object as the second argument to your callback function. This table lists the properties of this object.

Property	Description
`Character`	The character that appears as a result of pressing the key or keys. Pressing certain keys or modifying keys with the **Ctrl** key might return unprintable characters in this property. Pressing and releasing certain keys alone (such as **Ctrl**, **Alt**, **Shift**) does not generate `Character` data.
`Modifier`	Cell array containing the names of the modifier keys released, such as **control**, **alt**, or **shift**. If no modifier keys are released, the cell array is empty.

Property	Description
`Key`	Key released, identified by the (lowercase) label on the key, or a descriptive word such as `'space'`.
`Source`	Object that has focus when the key is released.
`EventName`	`'KeyRelease'`

For more information about specifying callback property values and using callback arguments, see “How to Specify Callback Property Values”.

`WindowKeyReleaseFcn` — Window key-release callback
`''` (default) | function handle | cell array | character vector

Window key-release callback, specified as one of these values

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback function executes when the user releases a key while the figure or a child object has focus.

This callback function can access specific information about the user’s interaction with the keyboard. MATLAB passes this information in a `KeyData` object as the second argument to your callback function. This table lists the properties of this object.

Property	Description
`Character`	Character that corresponds to the key or keys that are released. Certain keys or key combinations might return unprintable characters in this property. Pressing and releasing certain keys alone (such as **Ctrl**, **Alt**, **Shift**) does not generate `Character` data.
`Modifier`	Cell array containing the names of the modifier keys released, such as **control**, **alt**, or **shift**. If no modifier keys are released, the cell array is empty.

Property	Description
`Key`	Key released, identified by the (lowercase) label on the key, or a descriptive word such as `'space'`.
`Source`	Object that has focus when the key is released.
`EventName`	`'KeyRelease'`

For more information about specifying callback property values and using callback arguments, see “How to Specify Callback Property Values”.

`WindowScrollWheelFcn` — Window scroll wheel callback
`''` (default) | function handle | cell array | character vector

Window scroll wheel callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when the user moves the scroll wheel while the figure or any if its children have focus. However, other objects can capture scroll wheel movement and interfere with the execution of this callback.

Executing this callback has no effect on the `CurrentObject` and `SelectionType` properties.

The `WindowScrollWheelFcn` callback can access specific information when the user rotates the scroll wheel. MATLAB passes this information in a `ScrollWheelData` object as the second argument to your callback function. This table lists the properties of the `ScrollWheelData` object.

Property	Contents
`VerticalScrollCount`	A positive or negative number that indicates the direction and number of scroll wheel clicks. Scrolling down returns a positive value. Scrolling up returns a negative value. The vertical scroll count is the sum of all scroll wheel clicks that occurred since the last time the callback executed. Typically, the value is `1` or `-1`. However, the scroll count can have a larger magnitude if the `WindowScrollWheelFcn` callback takes a long time to return, or if the user spins the scroll wheel very fast.

Property	Contents
`VerticalScrollAmount`	The number of lines scrolled for each click of the scroll wheel. Typically, the value is 3, but some systems might return a different value depending on how the mouse is configured.
`Source`	Object that executes the callback.

Contents r o p e r t y
'WindowScrollWheel' v e n t N a m e

For more information about specifying callback property values and using callback arguments, see “How to Specify Callback Property Values”.

Example: App for Changing x-Axis Limits

This example shows how to create an app that displays a plot. The user can scroll to change the limits of the *x*-axis. Copy and paste the following code into the editor and run it.

```
function scroll_wheel
% Shows how to use WindowScrollWheelFcn property
%
   f = figure('WindowScrollWheelFcn',@figScroll,'Name','Scroll Wheel Demo');
   x = [0:.1:40];
   y = 4.*cos(x)./(x+2);
   a = axes;
   h = plot(x,y);
   title('Rotate the scroll wheel')

   function figScroll(src,callbackdata)
      if callbackdata.VerticalScrollCount > 0
         xd = h.XData;
         % This code uses dot notation to set properties
         % Dot notation runs in R2014b and later.
         % For R2014a and earlier: xd = get(h,'XData');
         inc = xd(end)/20;
         x = [0:.1:xd(end)+inc];
         re_eval(x)
```

```
      elseif callbackdata.VerticalScrollCount < 0
         xd = h.XData;
         % For R2014a and earlier: xd = get(h,'XData');
         inc = xd(end)/20;
         x = [0:.1:xd(end)-inc+.1]; % Don't let xd = 0;
         re_eval(x)
      end
   end

   function re_eval(x)
      y = 4.*cos(x)./(x+2);
      h.YData = y;
      h.XData = x;
      a.XLim = [0 x(end)];
      % For R2014a and earlier:
      % set(h,'YData',y);
      % set(h,'XData',x);
      % set(a,'XLim',[0 x(end)]);
      drawnow
   end
end
```

ResizeFcn — Resize callback function (not recommended)

`''` (default) | function handle | cell array | character vector

Resize callback function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

Note Use of the `ResizeFcn` property is not recommended. It might be removed in a future release. Use `SizeChangedFcn` instead.

Data Types: `function_handle` | `cell` | `char`

Callback Execution Control

Interruptible — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed:

- A value of `'on'` allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- A value of `'off'` blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn`, or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.
- MATLAB does not save the state of properties or the display when an interruption occurs. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

See “Interrupt Callback Execution” for an example that shows how the `Interruptible` and `BusyAction` properties affect the behavior of a program.

BusyAction — Callback queuing
'queue' (default) | 'cancel'

Callback queuing specified as 'queue' (default) or 'cancel'. The BusyAction property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

The BusyAction property of the source of the interrupting callback determines how MATLAB handles its execution. The BusyAction property has these values:

- 'queue' — Put the interrupting callback in a queue to be processed after the running callback finishes execution.
- 'cancel' — Do not execute the interrupting callback.

Whenever MATLAB invokes a callback, that callback always attempts to interrupt an executing callback. The Interruptible property of the object whose callback is running determines if interruption is allowed. If Interruptible is set to:

- on — Interruption occurs at the next point where MATLAB processes the queue. This is the default.
- off — The BusyAction property (of the object owning the interrupting callback) determines if MATLAB enqueues or ignores the interrupting callback.

See “Interrupt Callback Execution” for an example that shows how the BusyAction and Interruptible properties affect the behavior of a program.

HitTest — Ability to become current object
'on' (default) | 'off'

Ability to become current object, specified as 'on' or 'off':

- 'on' — Sets the current object to the Figure when the user clicks the component in the running app. Both the CurrentObject property of the Figure and the gco function return the Figure as the current object.
- 'off' — The current object does not update when the user clicks the figure in the running app.

BeingDeleted — Deletion status
'off' | 'on'

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Figure parent

root object

Figure parent, returned as a root object.

`Children` — Children of figure

empty `GraphicsPlaceholder` array (default) | 1-D array of objects

Children of the `Figure`, returned as an empty `GraphicsPlaceholder` or a 1-D array of objects.

You cannot add or remove children using the `Children` property. Use this property to view the list of children or to reorder the children. The order of the children reflects the front-to-back order (stacking order) of the components on the screen. MATLAB might not allow you to change the order of certain objects. For example, `UIControl` and `Legend` objects are always in front of `Axes` objects.

To add a child to this list, set the `Parent` property of the child component to be the `Figure` object.

Objects with the `HandleVisibility` property set to `'off'` do not list in the `Children` property.

`HandleVisibility` — Visibility of figure object

`'on'` (default) | `'callback'` | `'off'`

Visibility of figure object, specified as `'on'`, `'callback'`, or `'off'`.

This property determines whether a figure is in its parent's (the root's) list of children. `HandleVisibility` is useful for preventing command-line users from accidentally drawing into, or deleting a figure that contains only user interface components (such as a dialog box).

If an object is not in its parent's list of children, functions that find objects by searching the object hierarchy or querying properties cannot return that object. Such functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When the `HandleVisibility` property value is restricted using the `'callback'` or `'off'` settings, the object does not appear in the parent object `Children` property, figures do not appear in the root `CurrentFigure` property, objects do not appear in the root `CallbackObject` property or in the figure `CurrentObject` property, and axes do not appear in their parent `CurrentAxes` property.

Set the root `ShowHiddenHandles` property to `'on'` to make all objects visible, regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Identifiers

`Name` — Name
`''` (default) | character vector | string scalar

Name of the figure, specified as a character vector or a string scalar.

Example: `figure('Name','Results')` sets the name of the figure to `'Results'`.

By default, the name is `'Figure n'`, where *n* is an integer. When you specify the `Name` property, the title of the figure becomes `'Figure n: name'`. If you want only the `Name` value to appear, set `IntegerHandle` or `NumberTitle` to `'off'`.

`Number` — Number
integer | `[]`

This property is read-only.

Number of the figure, returned as an integer or empty array. You can refer to a figure using this value. For example, `figure(2)` makes the figure with a `Number` property value of `2` the current figure.

If the `IntegerHandle` property is set to `'off'`, the `Number` property is empty.

If `IntegerHandle` is `'on'`, the `Number` property is an integer. When you delete the figure, MATLAB reuses the number for the next figure.

`NumberTitle` — Use number title
`'on'` (default) | `'off'`

Number title, specified as `'on'` or `'off'`. When you set this property to `'on'`, the title of the figure includes the phrase `'Figure n'`, where `n` is the value of the `Number` property.

Both the `NumberTitle` and `IntegerHandle` must be set to `'on'` to show to show the number in the title.

`IntegerHandle` — Use integer handle
`'on'` (default) | `'off'`

Use integer handle, specified as `'on'` or `'off'`.

If you set the `IntegerHandle` property to `'on'`, MATLAB finds the lowest integer value that is not used by an existing figure and sets the `Number` property to that value. If you delete a figure, MATLAB can reuse its number on a new figure.

If you set the `IntegerHandle` property to `'off'`, MATLAB does not assign an integer value to the figure, and it sets the `Number` property to an empty array.

To show the figure number in the title bar, both `IntegerHandle` and `NumberTitle` must be set to `'on'`.

`FileName` — File name
character vector | string scalar

File name for saving the figure, specified as a character vector or a string scalar. GUIDE uses this property to store the name of the UI layout file that it saves.

Example: `figure('FileName','myfile.fig')` sets the file name to `myfile.fig`.

Type — Type of Figure object
`'figure'`

This property is read-only.

Type of Figure object, returned as `'figure'`. Use this property to find all objects of a given type within a plotting hierarchy.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object

elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[ ]` (default) | array

User data, specified as any array. Specifying `UserData` can be useful for sharing data within apps. See “Share Data Among Callbacks” for more information.

See Also

`figure` | `gcf`

Topics

“Access Property Values”
“Default Property Values”

Introduced before R2006a

figurepalette

Show or hide **Figure Palette**

Syntax

```
figurepalette('show')
figurepalette('hide')
figurepalette('toggle')
figurepalette
figurepalette(figure_handle,...)
```

Description

`figurepalette('show')` displays the Figure Palette on the current figure.

`figurepalette('hide')` hides the Figure Palette on the current figure.

`figurepalette('toggle')` toggles the visibility of the Figure Palette on the current figure.

`figurepalette` is the same as `figurepalette('show')`.

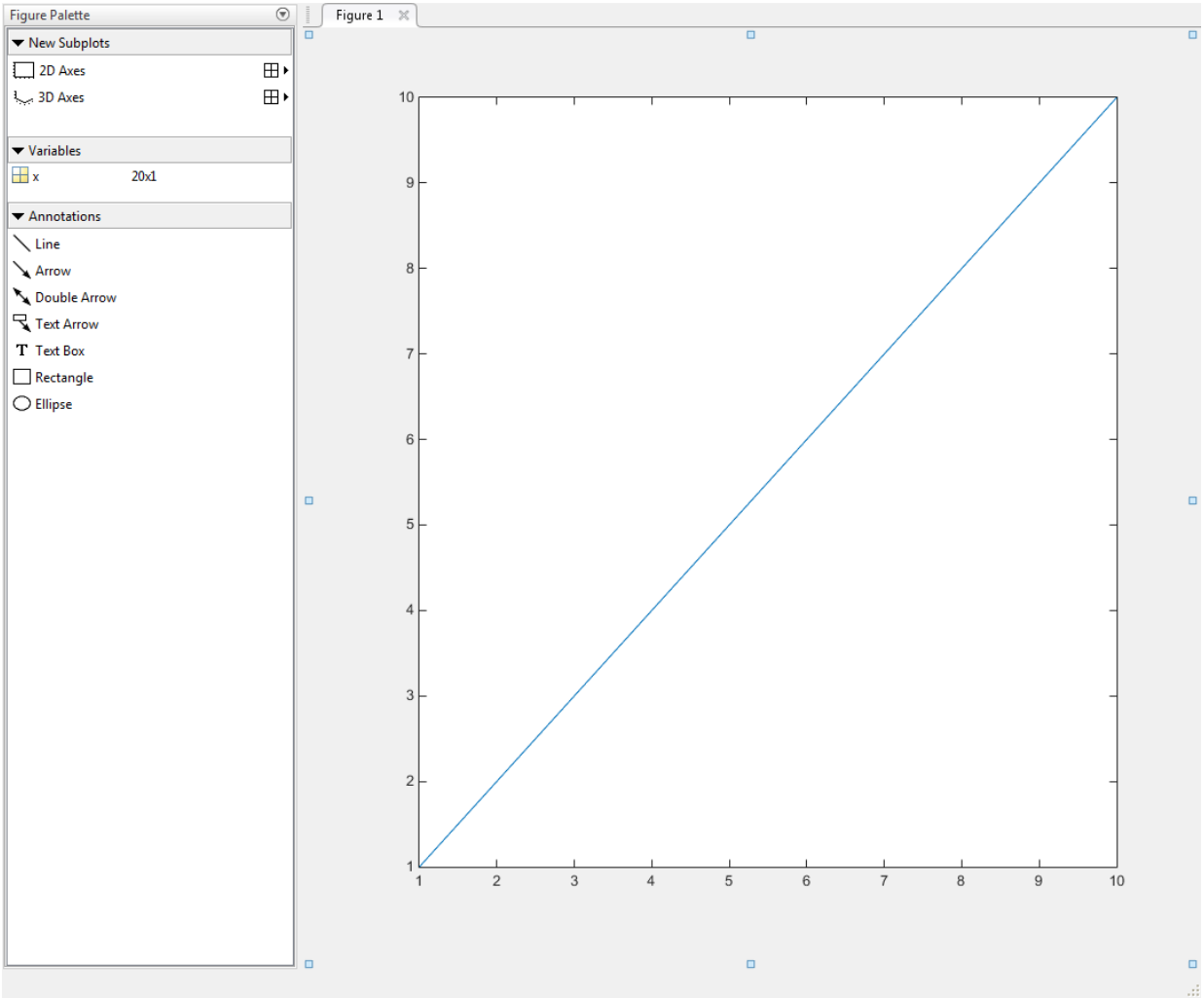
`figurepalette(figure_handle,...)` shows or hides the palette on the figure specified by `figure_handle`.

Examples

Open Figure Palette

Plot a line and open the figure palette.

```
plot(1:10)
figurepalette('show')
```



Tips

If you call `figurepalette` in a MATLAB program and subsequent lines depend on the Figure Palette being fully initialized, follow it by `drawnow` to ensure complete initialization.

Alternatives

Open or close the **Figure Palette** tool from the figure's **View** menu.

See Also

`plotbrowser` | `plottools` | `propertyeditor`

Introduced before R2006a

fileattrib

Set or get attributes of file or folder

Syntax

```
fileattrib
fileattrib filename

fileattrib filename attribs
fileattrib filename attribs users
fileattrib filename attribs users s

[status,values] = fileattrib(filename)
[status,msg,msgID] = fileattrib(filename,attribs, ___ )
```

Description

`fileattrib` lists the attribute values for the current folder. The values are listed using this structure:

```
Name
archive
system
hidden
directory
UserRead
UserWrite
UserExecute
GroupRead
GroupWrite
GroupExecute
OtherRead
OtherWrite
OtherExecute
```

The first field, `Name`, displays the file or folder name. The remaining fields display a value of `0` if the attribute is off, `1` if the attribute is on, and `NaN` if the attribute does not apply.

`fileattrib` is similar to the DOS `attrib` command, or the UNIX `chmod` command.

Note In Windows, setting the write access attribute (`'w'`) to read-only does not necessarily prevent write access. Therefore, the value for `UserWrite` may differ from what is expected.

`fileattrib filename` lists the attribute values for the named file or folder.

`fileattrib filename attribs` sets the specified attributes for the named file or folder.

`fileattrib filename attribs users` sets the file or folder attributes for the specified subset of users.

`fileattrib filename attribs users s` sets the specified attributes for the specified users for the contents of the named folder.

`[status,values] = fileattrib(filename)` returns the status and the last *successfully* set attribute values for the named file or folder. If the file exists, `status` is `1`. Otherwise, `status` is `0`.

`[status,msg,msgID] = fileattrib(filename,attribs, ___ )` sets the specified file attributes and returns the status of the operation as well as an error message and error message identifier if the set operation is unsuccessful.

Examples

View Current Folder Attributes on Windows

View attributes of the current folder on a Windows system, assuming the current folder is `C:\my_MATLAB_files`. The attributes indicate that you have read, write, and execute permissions for the current folder.

```
fileattrib
```

```
            Name: 'C:\my_MATLAB_files'
         archive: 0
          system: 0
          hidden: 0
       directory: 1
        UserRead: 1
       UserWrite: 1
     UserExecute: 1
       GroupRead: NaN
      GroupWrite: NaN
    GroupExecute: NaN
       OtherRead: NaN
      OtherWrite: NaN
    OtherExecute: NaN
```

View File Attributes on Windows

View attributes of the file `myfile.m` on a Windows system. The attributes indicate that the specified item is a file. You can read and execute the file, but cannot update it. The file is archived.

```
fileattrib myfile.m
```

```
            Name: 'C:\my_MATLAB_files\myfile.m'
         archive: 1
          system: 0
          hidden: 0
       directory: 0
        UserRead: 1
       UserWrite: 0
     UserExecute: 1
       GroupRead: NaN
      GroupWrite: NaN
    GroupExecute: NaN
       OtherRead: NaN
      OtherWrite: NaN
    OtherExecute: NaN
```

View Folder Attributes on Windows

View attributes for the folder `C:\my_MATLAB_files\doc`. The attributes indicate that you have read, write, and execute permissions for the folder.

```
fileattrib C:\my_MATLAB_files\doc
```

```
ans =

            Name: 'C:\my_MATLAB_files\doc'
         archive: 0
          system: 0
          hidden: 0
       directory: 1
        UserRead: 1
       UserWrite: 1
     UserExecute: 1
       GroupRead: NaN
      GroupWrite: NaN
    GroupExecute: NaN
       OtherRead: NaN
      OtherWrite: NaN
    OtherExecute: NaN
```

View Folder Attributes on UNIX

View attributes for the folder `/public` on a UNIX system. The attributes indicate that you have read, write, and execute permissions for the folder. In addition, users in your UNIX group and all others have read and execute permissions for the folder, but not write permissions.

```
fileattrib /public
```

```
ans =

            Name: '/public'
         archive: NaN
          system: NaN
          hidden: NaN
       directory: 1
        UserRead: 1
       UserWrite: 1
     UserExecute: 1
```

```
       GroupRead: 1
      GroupWrite: 0
    GroupExecute: 1
       OtherRead: 1
      OtherWrite: 0
    OtherExecute: 1
```

Set File Attributes on Windows

Make `myfile.m` writable.

```
fileattrib('myfile.m','+w')
```

Set File Attributes for All Users on UNIX

Make the folder `/home/work/results` a read-only folder for *all users* on UNIX platforms. The minus (`-`) preceding the write attribute, `w`, removes the write access, making the file read-only.

```
fileattrib('/home/work/results','-w','a')
```

Set Attributes for Folder and Its Contents on Windows

Make the folder `D:\work\results` and all its contents read-only and hidden. Because a value for the `users` argument is not applicable on Windows systems, `users` is specified as an empty character vector, `''`. The `'s'` argument applies the hidden and write access attributes to the contents of the folder and to the folder itself.

```
fileattrib('D:\work\results','+h -w','','s')
```

Get Attributes Structure for a Folder on Windows

Get the attributes for the folder `results` and return them as a structure. A `status` value of 1 indicates that the operation is successful. The structure `values` contains the attributes of the folder.

```
[status,values] = fileattrib('results')
```

```
status =
     1

values =
            Name: 'D:\work\results'
         archive: 0
          system: 0
          hidden: 0
       directory: 1
        UserRead: 1
       UserWrite: 1
     UserExecute: 1
       GroupRead: NaN
      GroupWrite: NaN
    GroupExecute: NaN
       OtherRead: NaN
      OtherWrite: NaN
    OtherExecute: NaN
```

Access the name attribute value in the structure. MATLAB returns the path for `results`.

```
values.Name
```

```
ans =
D:\work\results
```

Get Attributes Structure for Multiple Files on Windows

Get the attributes for all files in the current folder with names that begin with `new`. The returned `1x3` structure array `values` indicates that there are three matching files.

```
[status,values] = fileattrib('new*')
```

```
status =
     1

values =
1x3 struct array with fields:
    Name
    archive
    system
    hidden
    directory
```

```
    UserRead
    UserWrite
    UserExecute
    GroupRead
    GroupWrite
    GroupExecute
    OtherRead
    OtherWrite
    OtherExecute
```

View the file names.

```
values.Name
```

```
ans =
D:\work\results\newname.m

ans =
D:\work\results\newone.m

ans =
D:\work\results\newtest.m
```

View just the second file name.

```
values(2).Name
```

```
ans =
D:\work\results\newname.m
```

Successfully Set Attributes for a File and Get Messages on Windows

Show output that results when an attempt to set file attributes is successful. The `status` value of `1` indicates that the set operation was successful. Therefore, no error `msg` or `msgID` is returned.

```
[status,msg,msgID] = fileattrib('C:\my_MATLAB_files\doc',...
'+h -w','','s')
```

```
status =

     1
```

```
msg =

     ''

msgID =

     ''
```

Unsuccessfully Set Attributes for a File and Get Messages on Windows

Show output that results when an attempt to set file attributes is unsuccessful. The `status` value of `0` indicates that the set operation was unsuccessful. The minus sign incorrectly appears after `w`, instead of before it. `msg` describes the error that occurred and `msgID` contains the message identifier for the error that occurred.

```
[status,msg,msgID] = fileattrib('C:\my_MATLAB_files\doc',...
'+h w-','','s')
```

```
status =

     0


msg =

Illegal file mode characters on the current platform.


msgID =

MATLAB:FILEATTRIB:ModeSyntaxError
```

Input Arguments

`filename` — File or folder name
character vector | string scalar

File or folder name, specified as a character vector or string scalar. You can specify an absolute or relative path. `filename` can include wildcards (*).

Example: `fileattrib('myfile.m')`

Data Types: `char` | `string`

`attribs` — File or folder attribute values
character vector | string scalar

File or folder attribute values, specified as a character vector or string scalar consisting of one or more of these values separated by spaces:

Value	Description
`'a'`	Archive (Microsoft Windows platform only).
`'h'`	Hidden file (Windows platform only).
`'s'`	System file (Windows platform only).
`'w'`	Write access (Windows and UNIX platforms). Results differ by platform and application. For example, even though `fileattrib` disables the “write” privilege for a folder, making it read-only, files in the folder could be writable for some platforms or applications.
`'x'`	Executable (UNIX platform only).

Use the plus (+) qualifier before an attribute to set it, and the minus (-) qualifier before an attribute to clear it.

Example: `fileattrib('myfile.m', '+w -h')`

Data Types: `char` | `string`

`users` — Subset of users
`'a'` | `'g'` | `'o'` | `'u'` | `''`

Subset of users (on UNIX platforms only), specified as one of these values:

Value for UNIX Systems	Description
`'a'`	All users
`'g'`	Group of users
`'o'`	All other users
`'u'`	Current user

Specify an empty value `''` for all platforms other than UNIX. This value is not returned by `fileattrib` get operations.

Example: `fileattrib('/home/work/results','-w','a')`

Output Arguments

`status` — Indication of whether attempt to set or get attributes was successful
`0` | `1`

Indication of whether attempt to set or get attributes was successful, specified as `0` or `1`. If the attempt to set or get attributes was successful, `status` is `1`. Otherwise, `status` is `0`.

`values` — Attribute structure
structure array

Attribute structure, specified as a structure array containing these fields and possible values:

Field name	Possible Values
`Name`	Character vector containing name of file or folder
`archive`	`0` (not set), `1` (set), or `NaN` (not applicable)
`system`	`0` (not set), `1` (set), or `NaN` (not applicable)
`hidden`	`0` (not set), `1` (set), or `NaN` (not applicable)
`directory`	`0` (not set), `1` (set), or `NaN` (not applicable)
`UserRead`	`0` (not set), `1` (set), or `NaN` (not applicable)
`UserWrite`	`0` (not set), `1` (set), or `NaN` (not applicable)
`UserExecute`	`0` (not set), `1` (set), or `NaN` (not applicable)
`GroupRead`	`0` (not set), `1` (set), or `NaN` (not applicable)
`GroupWrite`	`0` (not set), `1` (set), or `NaN` (not applicable)
`GroupExecute`	`0` (not set), `1` (set), or `NaN` (not applicable)
`OtherRead`	`0` (not set), `1` (set), or `NaN` (not applicable)
`OtherWrite`	`0` (not set), `1` (set), or `NaN` (not applicable)
`OtherExecute`	`0` (not set), `1` (set), or `NaN` (not applicable)

Note On Windows systems, setting the write access attribute (`'w'`) to read-only does not necessarily prevent write access. Therefore, the value for `UserWrite` may differ from what is expected.

`msg` — Error message
character vector

Error message, specified as a character vector. If `status` is `0`, `msg` contains the message text of the error. If `status` is `1`, `msg` is empty, `''`.

`msgID` — Error message identifier
character vector

Error message identifier, specified as a character vector. If `status` is `0`, `msgID` contains the message id of the error. If `status` is `1`, `msgID` is empty, `''`.

See Also

`cd` | `copyfile` | `delete` | `dir` | `ls` | `mkdir` | `movefile` | `rmdir`

Introduced before R2006a

Current Folder Browser

Open Current Folder browser

Description

The Current Folder browser enables you to interactively manage files and folders in MATLAB. Use the Current Folder browser to view, create, open, move, and rename files and folders in the current folder.

Open the Current Folder Browser

To open the Current Folder browser if it is not currently visible, do one of the following:

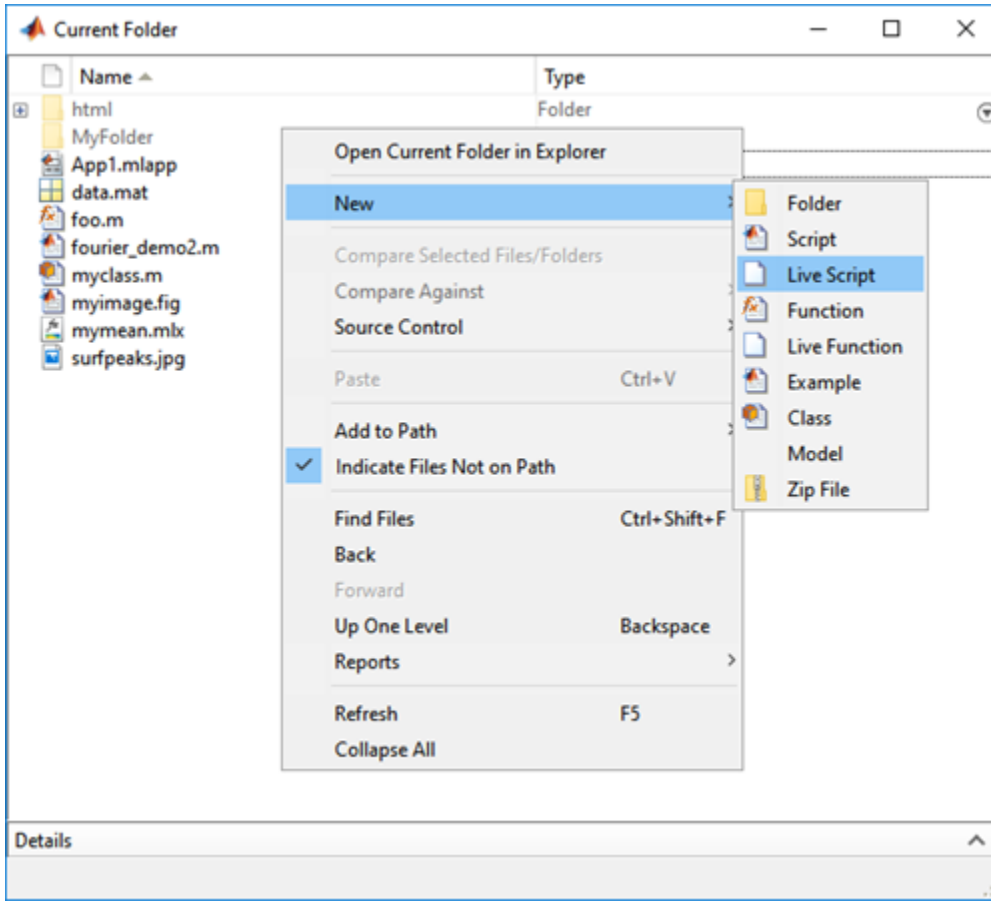
- MATLAB Toolstrip: On the **Home** tab, in the **Environment** section, click **Layout**. Then, in the **Show** section, select **Current Folder**.
- MATLAB command prompt: Enter `filebrowser`.

Examples

Create New Live Script

Create a new live script in the current folder.

In the Current Folder browser, right-click in white space, and then select **New Live Script**.



MATLAB creates and selects a live script named `Untitled`*n*`.mlx` in the current folder.

Enter a name for the live script and press **Enter**. For example, type `myLiveScript` and then press **Enter**. MATLAB renames the live script to `myLiveScript.mlx`

Programmatic Use

`filebrowser` opens the Current Folder browser. If the Current Folder browser is already open, MATLAB selects the tool.

See Also

`cd` | `ls` | `pwd`

Topics

"Manage Files and Folders"
"Enter Statements in Command Window"

Introduced before R2006a

filemarker

Character to separate file name from local or nested function name

Syntax

```
m = filemarker
```

Description

`m = filemarker` returns the character (> by default) used to separate the file name from the local or nested function name when referring to the function.

Examples

Get Help For Local Function

Get the help text for the local function `validateSizes` defined in `imwrite.m`.

```
help(['imwrite' filemarker 'validateSizes'])
```

```
  How many bytes does each element occupy in memory?
```

You can also use the filemarker character with the same result.

```
help('imwrite>validateSizes')
```

```
  How many bytes does each element occupy in memory?
```

Stop at Local and Nested Functions in File

Set a breakpoint in a program at the first executable line of a local function and of a nested function.

Create a file, `myfile.m`, that contains these statements

```
function n = myfile(x)
n = myfunction(x);
myfunction2;

    function z = myfunction2(x)
        z = x + 1;
    end

end

function y = myfunction(x)
y = x + 1;
end
```

Set a breakpoint at the local function `myfunction` using the `filemarker` character.

```
dbstop in myfile>myfunction
```

Set a breakpoint at the nested function `myfunction2` using the `filemarker` character.

```
dbstop in myfile>myfunction2
```

See Also

`dbstop` | `filesep`

Introduced in R2006a

fileparts

Get parts of file name

Syntax

```
[filepath,name,ext] = fileparts(filename)
```

Description

`[filepath,name,ext] = fileparts(filename)` returns the path name, file name, and extension for the specified file.

`fileparts` only parses the specified `filename`. It does not verify that the file exists.

Examples

Get Parts of File Name for Windows

```
file = 'H:\user4\matlab\myfile.txt';
[filepath,name,ext] = fileparts(file)
```

```
filepath =
'H:\user4\matlab'

name =
'myfile'

ext =
'.txt'
```

Get Parts of Dotfile Name for Linux

Get the parts of a user `.cshrc` file name for a Linux® system.

`fileparts` interprets the entire file name as an extension because it begins with a period.

```
[filepath,name,ext] = fileparts('/home/jsmith/.cshrc')
```

```
filepath =
'/home/jsmith'

name =

  1x0 empty char array

ext =
'.cshrc'
```

Input Arguments

`filename` — File name

string scalar | character vector

File name, specified as a string scalar or character vector. `filename` can include a path and file extension.

On Microsoft Windows systems, you can use either forward slashes (`/`) or backslashes (`\`) as path delimiters, even within the same file name. On UNIX and Macintosh systems, use only `/` as a delimiter.

Data Types: `char` | `string`

Output Arguments

`filepath` — File path

string scalar | character vector

File path, returned as a string scalar or character vector. `filepath` has the same data type as the input argument `filename`. If the name of the file to parse does not specify a path, `filepath` is empty (`''`).

Data Types: `char` | `string`

name — File name
string scalar | character vector

File name, returned as a string scalar or character vector. `name` has the same data type as the input argument `filename`.

The extension is not included. `fileparts` interprets all characters following the rightmost delimiter as the name of the file plus extension.

Data Types: `char` | `string`

ext — File extension
string scalar | character vector

File extension, returned as a string scalar or character vector. `ext` has the same data type as the input argument `filename`.

`ext` begins with a period (`.`). If the name of the file to parse does not specify an extension, `ext` is empty (`''`).

Data Types: `char` | `string`

Tips

- To reconstruct a file name from the output of `fileparts`, use `strcat` to concatenate the file name and the extension that begins with a period (.) without a path separator. Then, use `fullfile` to build the file name with the platform-dependent file separators where necessary. For example, `fullfile(filepath, strcat(name,ext))`.
- To specify a folder name only, be sure that the rightmost character in `filename` is a delimiter (/ or \). Otherwise, `fileparts` parses the trailing portion of `filename` as the name of the file and returns it in `name` instead of in `filepath`.

See Also

`filesep` | `fullfile` | `pathsep`

Topics

"Specify File Names"

Introduced before R2006a

fileread

Read contents of file as text

Syntax

```
text = fileread(filename)
```

Description

`text = fileread(filename)` returns contents of the file `filename` as a character vector.

Examples

Search File for Text

Read a file and search it for text of interest.

First, read the file `fileread.m` into a character vector.

```
filetext = fileread('fileread.m');
```

Then, define the text to search for.

```
expr = '[^\n]*fileread[^\n]*';
```

Find and return all lines that contain the text `'fileread'`.

```
matches = regexp(filetext,expr,'match');
```

Display the first matching line.

```
disp(matches{1})
```

```
function out=fileread(filename)
```

Input Arguments

`filename` — Name of file to read
character vector | string scalar

Name of the file to read, specified as a character vector or string scalar.

Example: `'myFile.dat'`

Data Types: `char` | `string`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If the function `fileread` reads the entire file, then all the data must fit in the largest array that is available for code generation.

See Also

`fgetl` | `fgets` | `fread` | `fscanf` | `importdata` | `textscan` | `type`

Introduced before R2006a

filesep

File separator for current platform

Syntax

```
f = filesep
```

Description

`f = filesep` returns the platform-specific file separator character. The file separator is the character that separates individual folder and file names in a path.

Examples

Create Path for Folder on Windows

Create a path to the `iofun` folder on a Microsoft® Windows® platform.

```
iofun_dir = ['toolbox' filesep 'matlab' filesep 'iofun']
```

```
iofun_dir =

    'toolbox\matlab\iofun'
```

Create Path for Folder on Linux

Create a path to the `iofun` folder on a Linux® platform.

```
iofun_dir = ['toolbox' filesep 'matlab' filesep 'iofun']
```

```
iofun_dir =
'toolbox\matlab\iofun'
```

See Also

`fileparts` | `fullfile` | `pathsep`

Introduced before R2006a

fill

Filled 2-D polygons

Syntax

```
fill(X,Y,C)
fill(X,Y,ColorSpec)
fill(X1,Y1,C1,X2,Y2,C2,...)
fill(...,'PropertyName',PropertyValue)
fill(ax,...)
h = fill(...)
```

Description

The `fill` function creates colored polygons.

`fill(X,Y,C)` creates filled polygons from the data in `X` and `Y` with vertex color specified by `C`. `C` is a vector or matrix used as an index into the colormap. If `C` is a row vector, `length(C)` must equal `size(X,2)` and `size(Y,2)`; if `C` is a column vector, `length(C)` must equal `size(X,1)` and `size(Y,1)`. If necessary, `fill` closes the polygon by connecting the last vertex to the first. The values in `X` and `Y` can be numeric, datetime, duration, or categorical values.

`fill(X,Y,ColorSpec)` fills two-dimensional polygons specified by `X` and `Y` with the color specified by `ColorSpec`.

`fill(X1,Y1,C1,X2,Y2,C2,...)` specifies multiple two-dimensional filled areas.

`fill(...,'PropertyName',PropertyValue)` allows you to specify property names and values for a patch graphics object.

`fill(ax,...)` creates the polygons in the axes specified by `ax` instead of in the current axes (`gca`). The option `ax` can precede any of the input argument combinations in the previous syntaxes.

`h = fill(...)` returns a vector of patch objects.

Examples

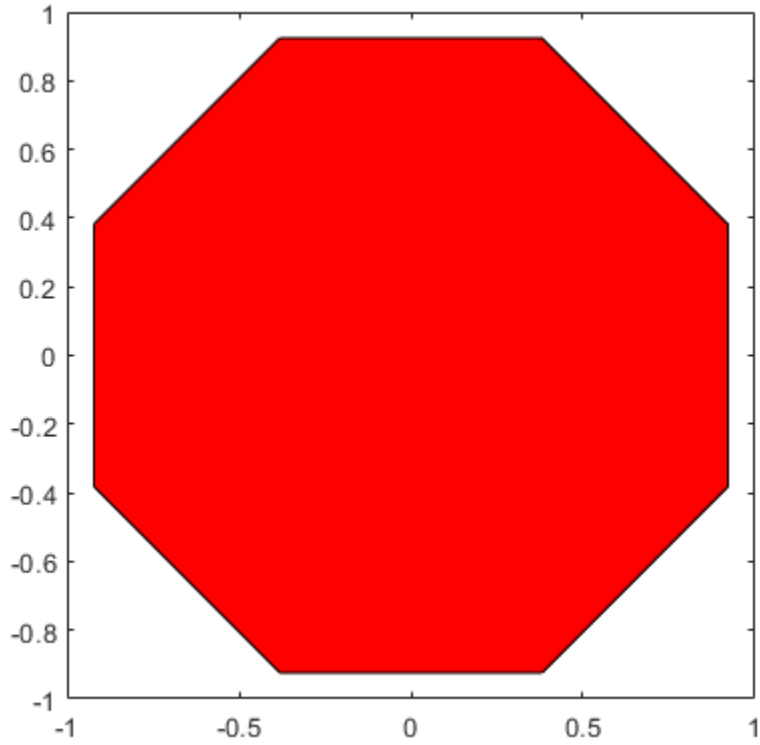
Create Red Octagon

Define the data.

```
t = (1/16:1/8:1)'*2*pi;
x = cos(t);
y = sin(t);
```

Create a red octagon using the `fill` function.

```
fill(x,y,'r')
axis square
```



Tips

If `X` or `Y` is a matrix, and the other is a column vector with the same number of elements as rows in the matrix, `fill` replicates the column vector argument to produce a matrix of the required size. `fill` forms a vertex from corresponding elements in `X` and `Y` and creates one polygon from the data in each column.

If `X` or `Y` contains one or more `NaN` values, then `fill` does not fill the polygons.

The type of color shading depends on how you specify color in the argument list. If you specify color using `ColorSpec`, `fill` generates flat-shaded polygons by setting the patch object's `FaceColor` property to the corresponding RGB triplet.

If you specify color using `C`, `fill` scales the elements of `C` by the values specified by the axes property `CLim`. After scaling `C`, `C` indexes the current colormap.

If `C` is a row vector, `fill` generates flat-shaded polygons where each element determines the color of the polygon defined by the respective column of the `X` and `Y` matrices. Each patch object's `FaceColor` property is set to `'flat'`. Each row element becomes the `CData` property value for the `n`th patch object, where *n* is the corresponding column in `X` or `Y`.

If `C` is a column vector or a matrix, `fill` uses a linear interpolation of the vertex colors to generate polygons with interpolated colors. It sets the patch graphics object `FaceColor` property to `'interp'` and the elements in one column become the `CData` property value for the respective patch object. If `C` is a column vector, `fill` replicates the column vector to produce the required sized matrix.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see "Run MATLAB Functions on a GPU" (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions
`ColorSpec` | `axis` | `caxis` | `colormap` | `fill3` | `patch`

Properties
Patch

Topics

“Plot Dates and Durations”
“Plot Categorical Data”

Introduced before R2006a

fill3

Filled 3-D polygons

Syntax

```
fill3(X,Y,Z,C)
fill3(X,Y,Z,ColorSpec)
fill3(X1,Y1,Z1,C1,X2,Y2,Z2,C2,...)
fill3(...,'PropertyName',PropertyValue)
fill3(ax,...)
h = fill3(...)
```

Description

The `fill3` function creates flat-shaded and Gouraud-shaded polygons.

`fill3(X,Y,Z,C)` fills three-dimensional polygons. `X`, `Y`, and `Z` triplets specify the polygon vertices. If `X`, `Y`, or `Z` is a matrix, `fill3` creates *n* polygons, where *n* is the number of columns in the matrix. `fill3` closes the polygons by connecting the last vertex to the first when necessary. The values in `X`, `Y`, and `Z` can be numeric, datetime, duration, or categorical values.

`C` specifies color, where `C` is a vector or matrix of indices into the current colormap. If `C` is a row vector, `length(C)` must equal `size(X,2)` and `size(Y,2)`; if `C` is a column vector, `length(C)` must equal `size(X,1)` and `size(Y,1)`.

`fill3(X,Y,Z,ColorSpec)` fills three-dimensional polygons defined by `X`, `Y`, and `Z` with color specified by `ColorSpec`.

`fill3(X1,Y1,Z1,C1,X2,Y2,Z2,C2,...)` specifies multiple filled three-dimensional areas.

`fill3(...,'PropertyName',PropertyValue)` allows you to set values for specific patch properties.

`fill3(ax,...)` creates the polygons in the axes specified by `ax` instead of in the current axes (`gca`). The option `ax` can precede any of the input argument combinations in the previous syntaxes.

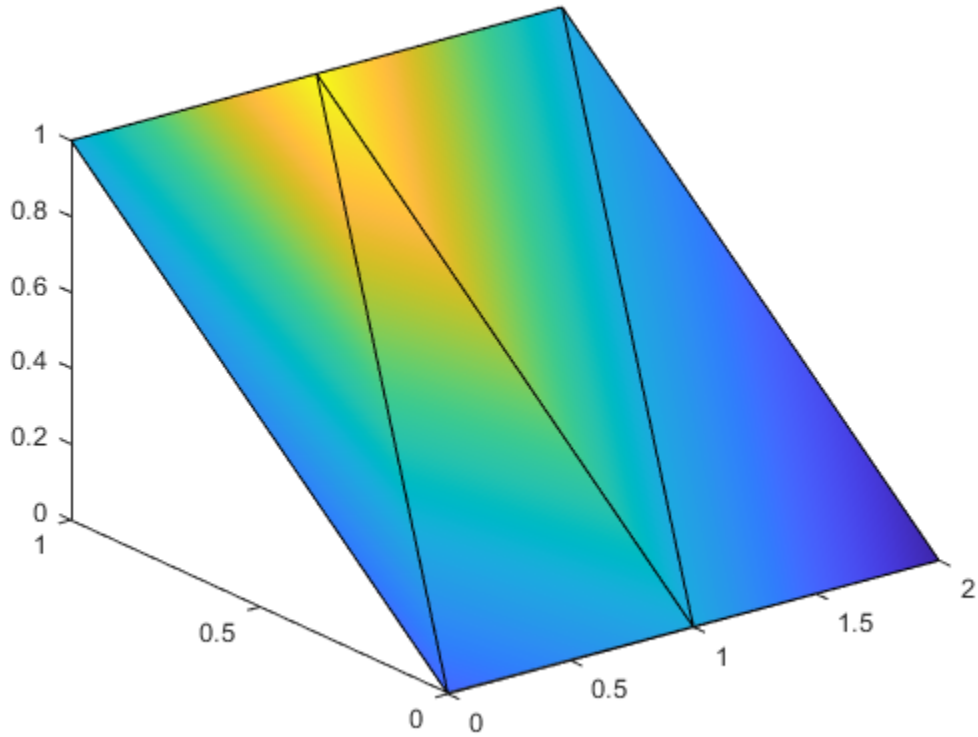
`h = fill3(...)` returns a vector of patch objects.

Examples

Create Filled 3-D Polygon

Create four triangles with interpolated colors.

```
X = [0 1 1 2; 1 1 2 2; 0 0 1 1];
Y = [1 1 1 1; 1 0 1 0; 0 0 0 0];
Z = [1 1 1 1; 1 0 1 0; 0 0 0 0];
C = [0.5000 1.0000 1.0000 0.5000;
     1.0000 0.5000 0.5000 0.1667;
     0.3330 0.3330 0.5000 0.5000];

figure
fill3(X,Y,Z,C)
```



Algorithms

If X, Y, and Z are matrices of the same size, `fill3` forms a vertex from the corresponding elements of X, Y, and Z (all from the same matrix location), and creates one polygon from the data in each column.

If X, Y, or Z is a matrix, `fill3` replicates any column vector argument to produce matrices of the required size.

If you specify color using `ColorSpec`, `fill3` generates flat-shaded polygons and sets the patch object `FaceColor` property to an RGB triplet.

If you specify color using `C`, `fill3` scales the elements of `CLim` by the axes property `CLim`, which specifies the color axis scaling parameters, before indexing the current colormap.

If `C` is a row vector, `fill3` generates flat-shaded polygons and sets the `FaceColor` property of the patch objects to `'flat'`. Each element becomes the `CData` property value for the respective patch object.

If `C` is a column vector or a matrix, `fill3` generates polygons with interpolated colors and sets the patch object `FaceColor` property to `'interp'`. `fill3` uses a linear interpolation of the vertex colormap indices when generating polygons with interpolated colors. The elements in one column become the `CData` property value for the respective patch object. If `C` is a column vector, `fill3` replicates the column vector to produce the required sized matrix.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions
`ColorSpec` | `axis` | `caxis` | `colormap` | `fill` | `patch`

Properties
Patch

Topics

“Plot Dates and Durations”
“Plot Categorical Data”

Introduced before R2006a

fillmissing

Fill missing values

Syntax

```
F = fillmissing(A,'constant',v)
F = fillmissing(A,method)
F = fillmissing(A,movmethod,window)
F = fillmissing( ___ ,dim)
F = fillmissing( ___ ,Name,Value)
[F,TF] = fillmissing( ___ )
```

Description

`F = fillmissing(A,'constant',v)` fills missing entries of an array or table with the constant value `v`. If `A` is a matrix or multidimensional array, then `v` can be either a scalar or a vector. When `v` is a vector, each element specifies the fill value in the corresponding column of `A`. If `A` is a table or timetable, then `v` can also be a cell array.

Missing values are defined according to the data type of `A`:

- `NaN` — `double`, `single`, `duration`, and `calendarDuration`
- `NaT` — `datetime`
- `<missing>` — `string`
- `<undefined>` — `categorical`
- `' '` — `char`
- `{''}` — `cell` of character arrays

If `A` is a table, then the data type of each column defines the missing value for that column.

`F = fillmissing(A,method)` fills missing entries using the method specified by `method`. For example, `fillmissing(A,'previous')` fills missing entries with the previous non-missing entry of `A`.

`F = fillmissing(A,movmethod,window)` fills missing entries using a moving window mean or median with window length `window`. For example, `fillmissing(A,'movmean',5)` fills data with a moving average using a window length of 5.

`F = fillmissing( ___ ,dim)` specifies the dimension of `A` to operate along. By default, `fillmissing` operates along the first dimension whose size does not equal 1. For example, if `A` is a matrix, then `fillmissing(A,2)` operates across the columns of `A`, filling missing data row by row.

`F = fillmissing( ___ ,Name,Value)` specifies additional parameters for filling missing values using one or more name-value pair arguments. For example, if `t` is a vector of time values, then `fillmissing(A,'linear','SamplePoints',t)` interpolates the data in `A` relative to the times in `t`.

`[F,TF] = fillmissing( ___ )` also returns a logical array corresponding to the entries of `A` that were filled.

Examples

Vector with NaN Values

Create a vector that contains `NaN` values and replace each `NaN` with the previous non-missing value.

```
A = [1 3 NaN 4 NaN NaN 5];
F = fillmissing(A,'previous')
```

```
F = 1×7

     1     3     3     4     4     4     5
```

Interpolate Missing Data

Use interpolation to replace `NaN` values in non-uniformly sampled data.

Define a vector of non-uniform sample points and evaluate the sine function over the points.

```
x = [-4*pi:0.1:0, 0.1:0.2:4*pi];
A = sin(x);
```

Inject `NaN` values into `A`.

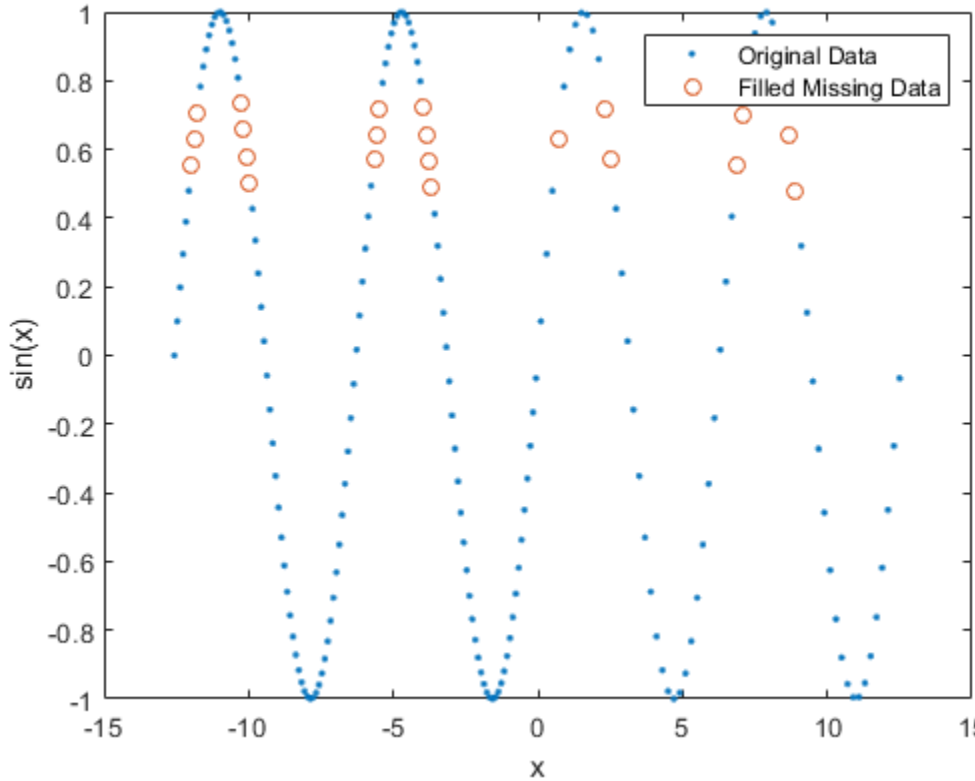
```
A(A < 0.75 & A > 0.5) = NaN;
```

Fill the missing data using linear interpolation, and return the filled vector `F` and the logical vector `TF`. The value 1 (`true`) in entries of `TF` corresponds to the values of `F` that were filled.

```
[F,TF] = fillmissing(A,'linear','SamplePoints',x);
```

Plot the original data and filled data.

```
plot(x,A,'.', x(TF),F(TF),'o')
xlabel('x');
ylabel('sin(x)')
legend('Original Data','Filled Missing Data')
```



Replace NaN with Moving Median

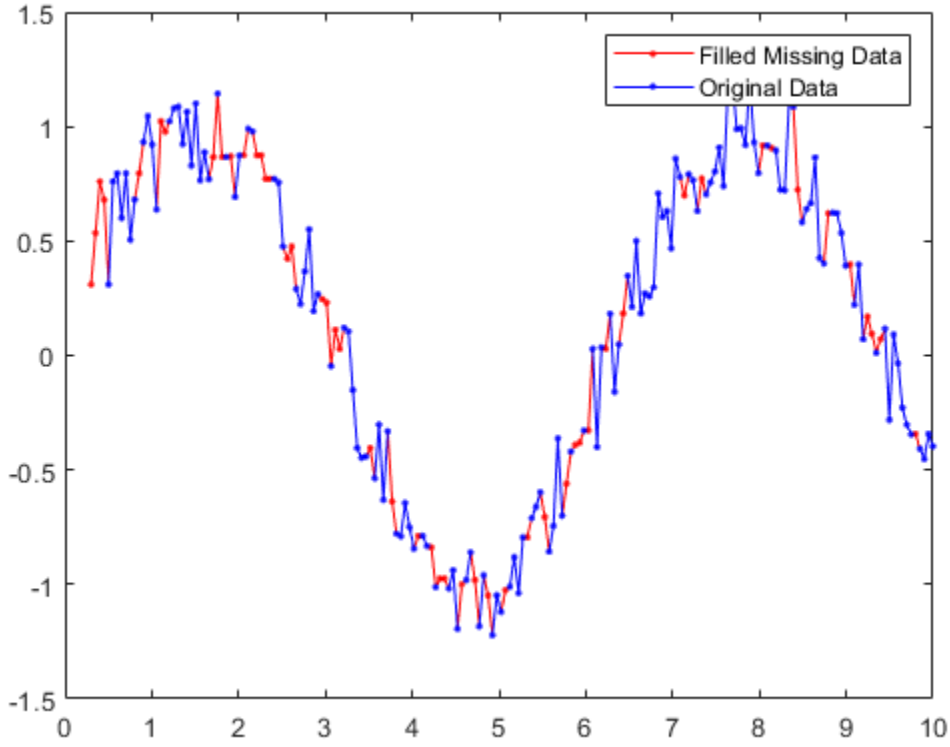
Use a moving median to fill missing numeric data.

Create a vector of sample points `x` and a vector of data `A` that contains missing values.

```
x = linspace(0,10,200);
A = sin(x) + 0.5*(rand(size(x))-0.5);
A([1:10 randi([1 length(x)],1,50)]) = NaN;
```

Replace `NaN` values in `A` using a moving median with a window of length 10, and plot both the original data and the filled data.

```
F = fillmissing(A,'movmedian',10);
plot(x,F,'r.-',x,A,'b.-')
legend('Filled Missing Data','Original Data')
```



Matrix with Missing Endpoints

Create a matrix with missing entries and fill across the columns (second dimension) one row at a time using linear interpolation. For each row, fill leading and trailing missing values with the nearest non-missing value in that row.

```
A = [NaN NaN 5 3 NaN 5 7 NaN 9 NaN;
     8 9 NaN 1 4 5 NaN 5 NaN 5;
     NaN 4 9 8 7 2 4 1 1 NaN]
```

```
A = 3×10

   NaN   NaN     5     3   NaN     5     7   NaN     9   NaN
     8     9   NaN     1     4     5   NaN     5   NaN     5
   NaN     4     9     8     7     2     4     1     1   NaN
```

```
F = fillmissing(A,'linear',2,'EndValues','nearest')
```

```
F = 3×10

     5     5     5     3     4     5     7     8     9     9
     8     9     5     1     4     5     5     5     5     5
     4     4     9     8     7     2     4     1     1     1
```

Table with Multiple Data Types

Fill missing values for table variables with different data types.

Create a table whose variables include `categorical`, `double`, and `char` data types.

```
A = table(categorical({'Sunny';'Cloudy';''}),[66;NaN;54],{'';'N';'Y'},[37;39;NaN],...
    'VariableNames',{'Description' 'Temperature' 'Rain' 'Humidity'})
```

```
A=3×4 table
    Description    Temperature       Rain       Humidity
    ___________    ___________    __________    ________

    Sunny               66        {0x0 char}       37
    Cloudy             NaN        {'N'     }       39
    <undefined>         54        {'Y'     }      NaN
```

Replace all missing entries with the value from the previous entry. Since there is no previous element in the `Rain` variable, the missing character vector is not replaced.

```
F = fillmissing(A,'previous')
```

```
F=3×4 table
    Description    Temperature       Rain       Humidity
    ___________    ___________    __________    ________

      Sunny            66         {0x0 char}       37
      Cloudy           66         {'N'     }       39
      Cloudy           54         {'Y'     }       39
```

Replace the `NaN` values from the `Temperature` and `Humidity` variables in `A` with 0.

```
F = fillmissing(A,'constant',0,'DataVariables',{'Temperature','Humidity'})
```

```
F=3×4 table
    Description    Temperature       Rain       Humidity
    ___________    ___________    __________    ________

    Sunny              66         {0x0 char}       37
    Cloudy              0         {'N'     }       39
    <undefined>        54         {'Y'     }        0
```

Alternatively, use the `isnumeric` function to identify the numeric variables to operate on.

```
F = fillmissing(A,'constant',0,'DataVariables',@isnumeric)
```

```
F=3×4 table
    Description    Temperature       Rain       Humidity
    ___________    ___________    __________    ________

    Sunny              66         {0x0 char}       37
    Cloudy              0         {'N'     }       39
    <undefined>        54         {'Y'     }        0
```

Input Arguments

A — Input data

vector | matrix | multidimensional array | table | timetable

Input data, specified as a vector, matrix, multidimensional array, table, or timetable.

If A is a timetable, then only table values are filled. If the associated vector of row times contains a NaT or NaN value, then fillmissing produces an error. Row times must be unique and listed in ascending order.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char | string | cell | table | timetable | categorical | datetime | duration | calendarDuration

v — Fill constant
scalar | vector | cell array

Fill constant, specified as a scalar, vector, or cell array. v can be a vector when A is a matrix or multidimensional array. v can be a cell array when A is a table or timetable.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char | cell | categorical | datetime | duration

method — Fill method
'previous' | 'next' | 'nearest' | 'linear' | 'spline' | 'pchip' | 'makima'

Fill method, specified as one of the following:

Method	Description
'previous'	previous non-missing value
'next'	next non-missing value
'nearest'	nearest non-missing value
'linear'	linear interpolation of neighboring, non-missing values (numeric, duration, and datetime data types only)
'spline'	piecewise cubic spline interpolation (numeric, duration, and datetime data types only)
'pchip'	shape-preserving piecewise cubic spline interpolation (numeric, duration, and datetime data types only)
'makima'	modified Akima cubic Hermite interpolation (numeric, duration, and datetime data types only)

movmethod — Moving method
'movmean' | 'movmedian'

Moving method to fill missing data, specified as one of the following:

Method	Description
'movmean'	Moving average over a window of length window (numeric data types only)
'movmedian'	Moving median over a window of length window (numeric data types only)

window — Window length
positive integer scalar | two-element vector of positive integers | positive duration scalar | two-element vector of positive durations

Window length, specified as a positive integer scalar, a two-element vector of positive integers, a positive duration scalar, or a two-element vector of positive durations.

When window is a positive integer scalar, then the window is centered about the current element and contains window-1 neighboring elements. If window is even, then the window is centered about the current and previous elements. If window is a two-element vector of positive integers [b f], then the window contains the current element, b elements backward, and f elements forward.

When A is a timetable or 'SamplePoints' is specified as a datetime or duration vector, window must be of type duration, and the windows are computed relative to the sample points.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | duration

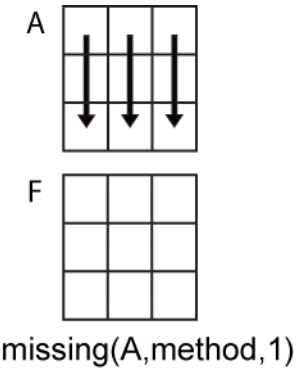
dim — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

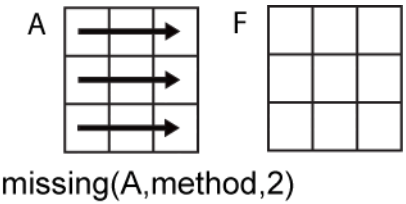
When A is a table or timetable, dim is not supported. fillmissing operates along each table or timetable variable separately.

Consider a two-dimensional input array, A.

- If `dim=1`, then `fillmissing` fills `A` column by column.



- If `dim=2`, then `fillmissing` fills `A` row by row.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `fillmissing(A,'DataVariables',{'Temperature','Altitude'})` fills only the columns corresponding to the `Temperature` and `Altitude` variables of an input table

`EndValues` — Method for handling endpoints
`'extrap'` (default) | `'previous'` | `'next'` | `'nearest'` | `'none'` | scalar

Method for handling endpoints, specified as the comma-separated pair consisting of `'EndValues'` and one of `'extrap'`, `'previous'`, `'next'`, `'nearest'`, `'none'`, or a constant scalar value. The endpoint fill method handles leading and trailing missing values based on the following definitions:

Method	Description
`'extrap'`	same as `method`
`'previous'`	previous non-missing value
`'next'`	next non-missing value
`'nearest'`	nearest non-missing value
`'none'`	no fill value
scalar	constant value (numeric, `duration`, and `datetime` data types only)

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `datetime` | `duration`

`SamplePoints` — Sample points
vector

Sample points for fill method, specified as the comma-separated pair consisting of `'SamplePoints'` and a vector. The sample points represent the location of the data in `A`, and must be sorted and contain unique elements. Sample points do not need to be uniformly sampled. If `A` is a timetable, then the default sample points vector is the vector of row times. Otherwise, the default vector is `[1 2 3 ...]`.

Moving windows are defined relative to the sample points. For example, if `t` is a vector of times corresponding to the input data, then `fillmissing(rand(1,10),'movmean',3,'SamplePoints',t)` has a window that represents the time interval between `t(i)-1.5` and `t(i)+1.5`.

When the sample points vector has data type `datetime` or `duration`, then the moving window length must have type `duration`.

This name-value pair is not supported when the input data is a timetable.

Data Types: `double` | `single` | `datetime` | `duration`

`DataVariables` — Table variables to fill
variable name | cell array of variable names | numeric vector | logical vector | function handle

Table variables to fill, specified as the comma-separated pair consisting of `'DataVariables'` and a variable name, a cell array of variable names, a numeric vector, a logical vector, or a function handle. The `'DataVariables'` value indicates which columns of the input table to fill, and can be one of the following:

- A character vector specifying a single table variable name
- A cell array of character vectors where each element is a table variable name
- A vector of table variable indices
- A logical vector whose elements each correspond to a table variable, where `true` includes the corresponding variable and `false` excludes it
- A function handle that returns a logical scalar, such as `@isnumeric`

Example: `'Age'`

Example: `{'Height','Weight'}`

Example: `@iscategorical`

Data Types: `char` | `cell` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `function_handle`

`MissingLocations` — Known missing indicator
vector | matrix | multidimensional array

Known missing indicator, specified as the comma-separated pair consisting of `'MissingLocations'` and a logical vector, matrix, or multidimensional array of the same size as `A`. The indicator elements can be `true` to indicate a missing value in the corresponding location of `A` or `false` otherwise.

Data Types: `logical`

Output Arguments

`F` — Filled data
vector | matrix | multidimensional array | table | timetable

Filled data, returned as a vector, matrix, multidimensional array, table, or timetable. F is the same size as A.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char | string | cell | table | timetable | categorical | datetime | duration | calendarDuration

TF — Filled data indicator
vector | matrix | multidimensional array

Filled data indicator, returned as a vector, matrix, or multidimensional array. TF is a logical array where 1 (true) corresponds to entries in F that were filled and 0 (false) corresponds to unchanged entries. TF is the same size as A and F.

Data Types: logical

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- The 'spline' and 'makima' methods are not supported.
- The 'SamplePoints' and 'MissingLocations' name-value pairs are not supported.
- The 'DataVariables' name-value pair cannot specify a function handle.
- The 'EndValues' name-value pair can only specify 'extrap'.
- The syntax fillmissing(A,movmethod,window) is not supported when A is a tall timetable.
- The syntax fillmissing(A,'constant',v) must specify a scalar value for v.
- The syntax fillmissing(A,___) does not support character vector variables when A is a tall table or tall timetable.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The `'makima'` option is not supported.
- Tables are not supported.

See Also

Clean Missing Data | `filloutliers` | `ismissing` | `isnan` | `missing` | `rmmissing` | `standardizeMissing`

Topics

Clean Messy Data and Locate Extrema Using Live Editor Tasks
"Missing Data in MATLAB"

Introduced in R2016b

filloutliers

Detect and replace outliers in data

Syntax

```
B = filloutliers(A,fillmethod)
B = filloutliers(A,fillmethod,findmethod)
B = filloutliers(A,fillmethod,'percentiles',threshold)
B = filloutliers(A,fillmethod,movmethod,window)
B = filloutliers( ___ ,dim)
B = filloutliers( ___ ,Name,Value)
[B,TF,L,U,C] = filloutliers( ___ )
```

Description

`B = filloutliers(A,fillmethod)` finds outliers in `A` and replaces them according to `fillmethod`. For example, `filloutliers(A,'previous')` replaces outliers with the previous non-outlier element. By default, an outlier is a value that is more than three scaled median absolute deviations (MAD) on page 1-4317 away from the median. If `A` is a matrix or table, then `filloutliers` operates on each column separately. If `A` is a multidimensional array, then `filloutliers` operates along the first dimension whose size does not equal 1.

`B = filloutliers(A,fillmethod,findmethod)` specifies a method for detecting outliers. For example, `filloutliers(A,'previous','mean')` defines an outlier as an element of `A` more than three standard deviations from the mean.

`B = filloutliers(A,fillmethod,'percentiles',threshold)` defines outliers as points outside of the percentiles specified in `threshold`. The `threshold` argument is a two-element row vector containing the lower and upper percentile thresholds, such as `[10 90]`.

`B = filloutliers(A,fillmethod,movmethod,window)` specifies a moving method for detecting local outliers according to a window length defined by `window`. For example, `filloutliers(A,'previous','movmean',5)` identifies outliers as elements

more than three local standard deviations away from the local mean within a five-element window.

`B = filloutliers( ___ ,dim)` operates along dimension `dim` of `A` for any of the previous syntaxes. For example, `filloutliers(A,'linear',2)` operates on each row of a matrix `A`.

`B = filloutliers( ___ ,Name,Value)` specifies additional parameters for detecting and replacing outliers using one or more name-value pair arguments. For example, `filloutliers(A,'previous','SamplePoints',t)` detects outliers in `A` relative to the corresponding elements of a time vector `t`.

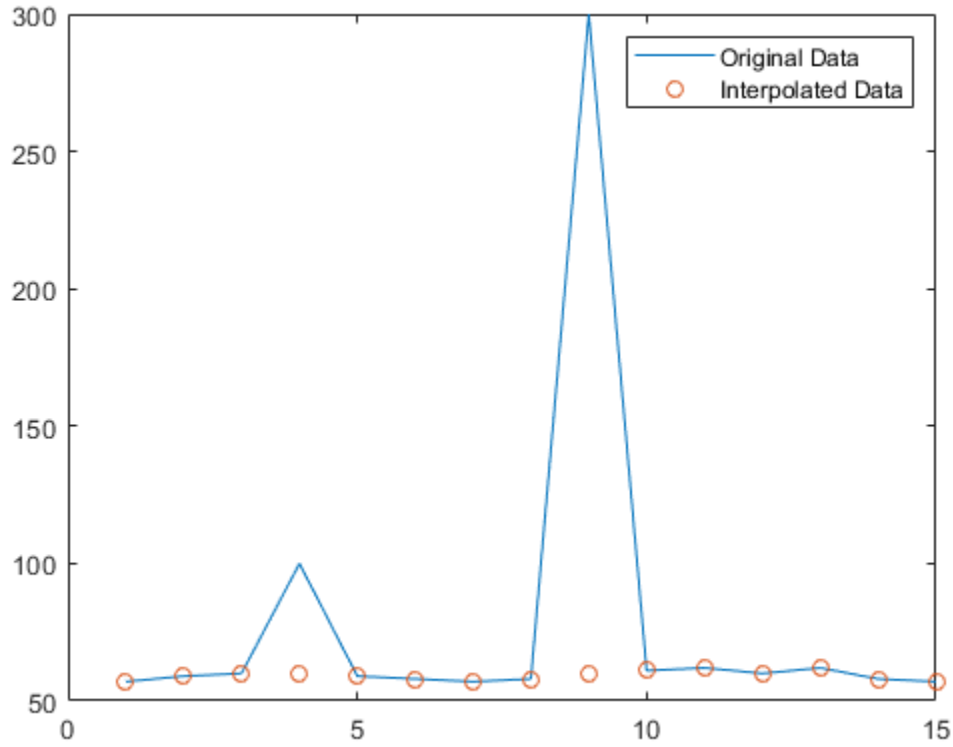
`[B,TF,L,U,C] = filloutliers( ___ )` also returns information about the position of the outliers and thresholds computed by the detection method. `TF` is a logical array indicating the location of the outliers in `A`. The `L`, `U`, and `C` arguments represent the lower and upper thresholds and the center value used by the outlier detection method.

Examples

Interpolate Outliers

Create a vector of data containing an outlier, and use linear interpolation to replace the outlier. Plot the original and filled data.

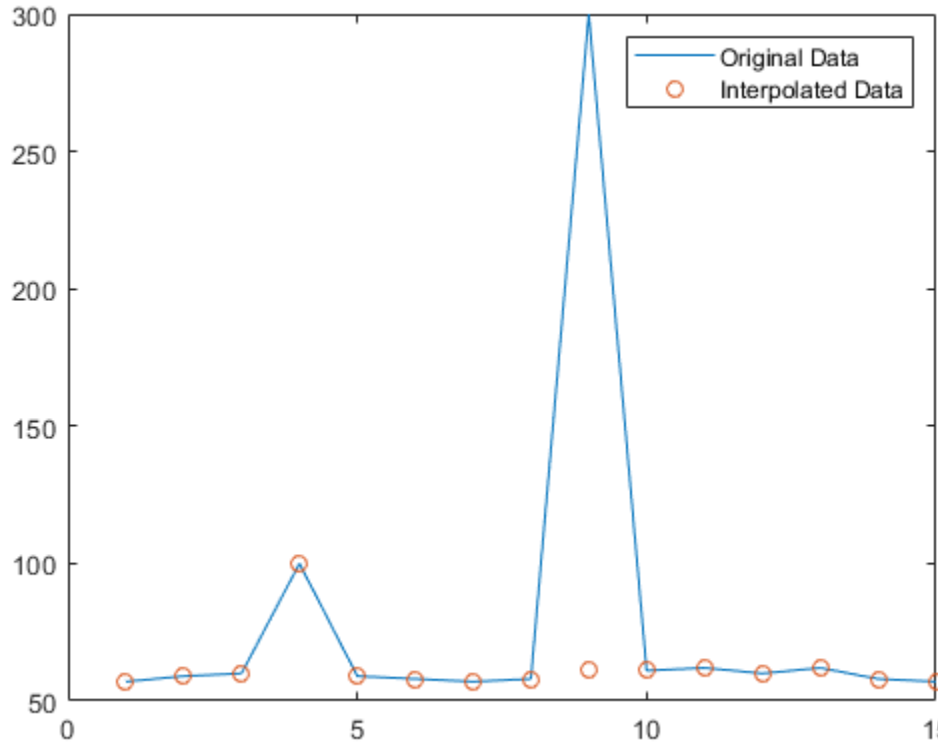
```
A = [57 59 60 100 59 58 57 58 300 61 62 60 62 58 57];
B = filloutliers(A,'linear');
plot(1:15,A,1:15,B,'o')
legend('Original Data','Interpolated Data')
```



Determine Outliers with Mean

Create a vector containing an outlier, and define outliers as points outside three standard deviations from the mean. Replace the outlier with the nearest element that is not an outlier, and plot the original data and the interpolated data.

```
A = [57 59 60 100 59 58 57 58 300 61 62 60 62 58 57];
B = filloutliers(A,'nearest','mean');
plot(1:15,A,1:15,B,'o')
legend('Original Data','Interpolated Data')
```



Determine Outliers with Sliding Window

Use a moving median to find local outliers within a sine wave that corresponds to a time vector.

Create a vector of data containing a local outlier.

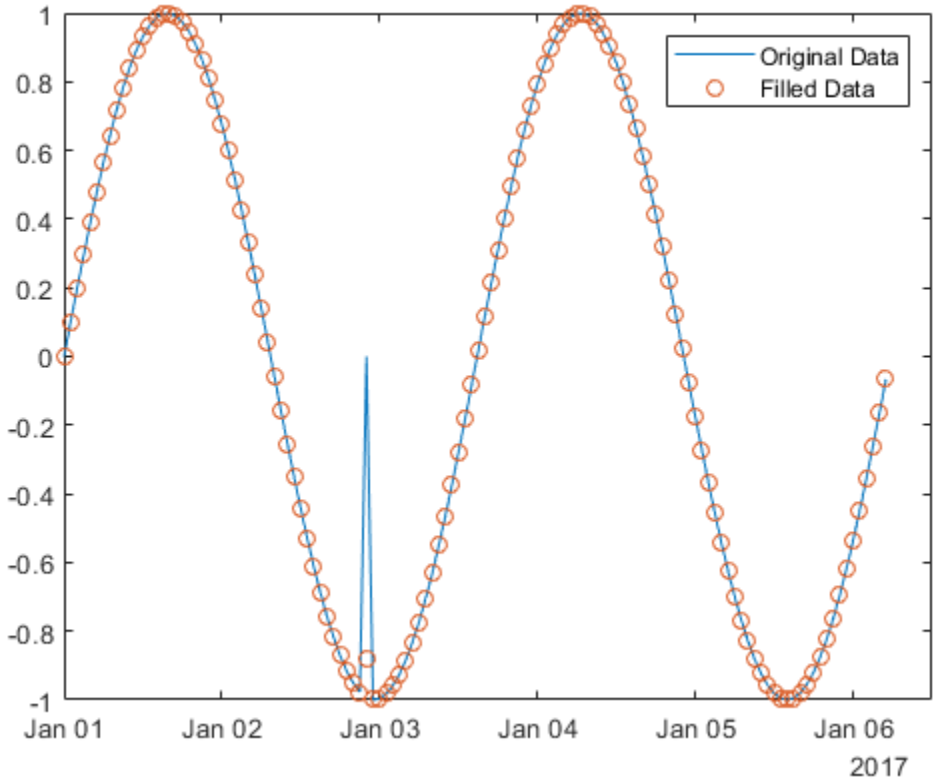
```
x = -2*pi:0.1:2*pi;
A = sin(x);
A(47) = 0;
```

Create a time vector that corresponds to the data in `A`.

```
t = datetime(2017,1,1,0,0,0) + hours(0:length(x)-1);
```

Define outliers as points more than three local scaled MAD away from the local median within a sliding window. Find the location of the outlier in `A` relative to the points in `t` with a window size of 5 hours. Fill the outlier with the computed threshold value using the method `'clip'`, and plot the original and filled data.

```
[B,TF,U,L,C] = filloutliers(A,'clip','movmedian',hours(5),'SamplePoints',t);
plot(t,A,t,B,'o')
legend('Original Data','Filled Data')
```



Display the threshold value that replaced the outlier.

```
L(TF)
```

```
ans = -0.8779
```

Matrix of Data

Fill outliers for each row of a matrix.

Create a matrix of data containing outliers along the diagonal.

```
A = randn(5,5) + diag(1000*ones(1,5))
```

```
A = 5×5
10^3 ×

    1.0005   -0.0013   -0.0013   -0.0002    0.0007
    0.0018    0.9996    0.0030   -0.0001   -0.0012
   -0.0023    0.0003    1.0007    0.0015    0.0007
    0.0009    0.0036   -0.0001    1.0014    0.0016
    0.0003    0.0028    0.0007    0.0014    1.0005
```

Fill outliers with zeros based on the data in each row, and display the new values.

```
[B,TF,lower,upper,center] = filloutliers(A,0,2);
B
```

```
B = 5×5

         0   -1.3077   -1.3499   -0.2050    0.6715
    1.8339         0    3.0349   -0.1241   -1.2075
   -2.2588    0.3426         0    1.4897    0.7172
    0.8622    3.5784   -0.0631         0    1.6302
    0.3188    2.7694    0.7147    1.4172         0
```

You can directly access the detected outlier values and their filled values using `TF` as an index vector.

```
[A(TF) B(TF)]
```

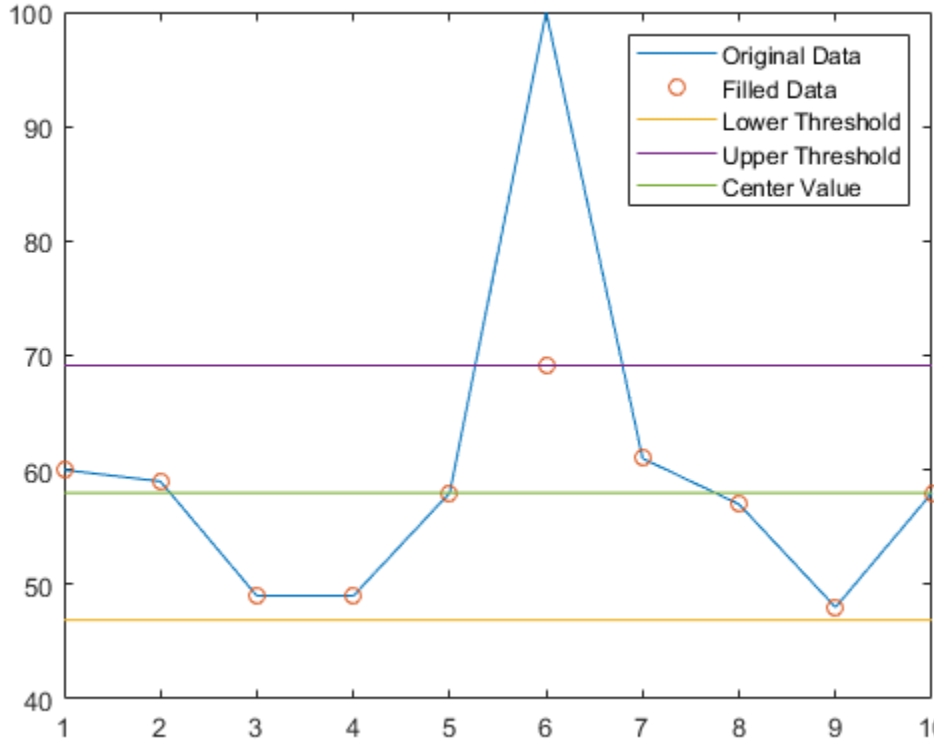
```
ans = 5×2
10^3 ×

    1.0005         0
```

```
    0.9996         0
    1.0007         0
    1.0014         0
    1.0005         0
```

Outlier Thresholds

Find the outlier in a vector of data, and replace it using the `'clip'` method. Plot the original data, the filled data, and the thresholds and center value determined by the detection method. `'clip'` replaces the outlier with the upper threshold value.

```
x = 1:10;
A = [60 59 49 49 58 100 61 57 48 58];
[B,TF,lower,upper,center] = filloutliers(A,'clip');
plot(x,A,x,B,'o',x,lower*ones(1,10),x,upper*ones(1,10),x,center*ones(1,10))
legend('Original Data','Filled Data','Lower Threshold','Upper Threshold','Center Value
```



Input Arguments

A — Input data
vector | matrix | multidimensional array | table | timetable

Input data, specified as a vector, matrix, multidimensional array, table, or timetable.

If `A` is a table, then its variables must be of type `double` or `single`, or you can use the `'DataVariables'` name-value pair to list `double` or `single` variables explicitly. Specifying variables is useful when you are working with a table that contains variables with data types other than `double` or `single`.

If A is a timetable, then `filloutliers` operates only on the table elements. Row times must be unique and listed in ascending order.

Data Types: `double` | `single` | `table` | `timetable`

`fillmethod` — Fill method
numeric scalar | `'center'` | `'clip'` | `'previous'` | `'next'` | `'nearest'` | `'linear'` | `'spline'` | `'pchip'` | `'makima'`

Fill method for replacing outliers, specified as a numeric scalar or one of the following:

Fill Method	Description
Numeric scalar	Fills with specified scalar value
`'center'`	Fills with the center value determined by `findmethod`
`'clip'`	Fills with the lower threshold value for elements smaller than the lower threshold determined by `findmethod`. Fills with the upper threshold value for elements larger than the upper threshold determined by `findmethod`
`'previous'`	Fills with the previous non-outlier value
`'next'`	Fills with the next non-outlier value
`'nearest'`	Fills with the nearest non-outlier value
`'linear'`	Fills using linear interpolation of neighboring, non-outlier values
`'spline'`	Fills using piecewise cubic spline interpolation
`'pchip'`	Fills using shape-preserving piecewise cubic spline interpolation
`'makima'`	modified Akima cubic Hermite interpolation (numeric, `duration`, and `datetime` data types only)

Data Types: `double` | `single` | `char`

`findmethod` — Method for detecting outliers
`'median'` (default) | `'mean'` | `'quartiles'` | `'grubbs'` | `'gesd'`

Method for detecting outliers, specified as one of the following:

Method	Description
`'median'`	Outliers are defined as elements more than three scaled MAD from the median. The scaled MAD is defined as `c*median(abs(A-median(A)))`, where `c=-1/(sqrt(2)*erfcinv(3/2))`.
`'mean'`	Outliers are defined as elements more than three standard deviations from the mean. This method is faster but less robust than `'median'`.
`'quartiles'`	Outliers are defined as elements more than 1.5 interquartile ranges above the upper quartile (75 percent) or below the lower quartile (25 percent). This method is useful when the data in `A` is not normally distributed.
`'grubbs'`	Outliers are detected using Grubbs's test, which removes one outlier per iteration based on hypothesis testing. This method assumes that the data in `A` is normally distributed.
`'gesd'`	Outliers are detected using the generalized extreme Studentized deviate test for outliers. This iterative method is similar to `'grubbs'`, but can perform better when there are multiple outliers masking each other.

`threshold` — Percentile thresholds

two-element row vector

Percentile thresholds, specified as a two-element row vector whose elements are in the interval [0,100]. The first element indicates the lower percentile threshold and the second element indicates the upper percentile threshold. For example, a threshold of `[10 90]` defines outliers as points below the 10th percentile and above the 90th percentile. The first element of `threshold` must be less than the second element.

movmethod — Moving method
'movmedian' | 'movmean'

Moving method for detecting outliers, specified as one of the following:

Method	Description
'movmedian'	Outliers are defined as elements more than three local scaled MAD from the local median over a window length specified by window.
'movmean'	Outliers are defined as elements more than three local standard deviations from the local mean over a window length specified by window.

window — Window length
positive integer scalar | two-element vector of positive integers | positive duration scalar | two-element vector of positive durations

Window length, specified as a positive integer scalar, a two-element vector of positive integers, a positive duration scalar, or a two-element vector of positive durations.

When window is a positive integer scalar, the window is centered about the current element and contains window-1 neighboring elements. If window is even, then the window is centered about the current and previous elements.

When window is a two-element vector of positive integers [b f], the window contains the current element, b elements backward, and f elements forward.

When A is a timetable or 'SamplePoints' is specified as a datetime or duration vector, window must be of type duration, and the windows are computed relative to the sample points.

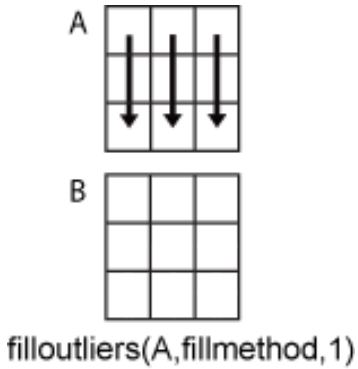
Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | duration

dim — Dimension to operate along
positive integer scalar

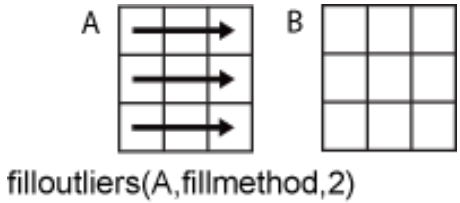
Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a matrix `A`.

`filloutliers(A,fillmethod,1)` fills outliers according to the data in each column.



`filloutliers(A,fillmethod,2)` fills outliers according to the data in each row.



When `A` is a table or timetable, `dim` is not supported. `filloutliers` operates along each table or timetable variable separately.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `filloutliers(A,'center','mean','ThresholdFactor',4)`

`ThresholdFactor` — Detection threshold factor
nonnegative scalar

Detection threshold factor, specified as the comma-separated pair consisting of `'ThresholdFactor'` and a nonnegative scalar.

For methods `'median'` and `'movmedian'`, the detection threshold factor replaces the number of scaled MAD, which is 3 by default.

For methods `'mean'` and `'movmean'`, the detection threshold factor replaces the number of standard deviations from the mean, which is 3 by default.

For methods `'grubbs'` and `'gesd'`, the detection threshold factor is a scalar ranging from 0 to 1. Values close to 0 result in a smaller number of outliers and values close to 1 result in a larger number of outliers. The default detection threshold factor is 0.5.

For the `'quartiles'` method, the detection threshold factor replaces the number of interquartile ranges, which is 1.5 by default.

This name-value pair is not supported when the specified method is `'percentiles'`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`SamplePoints` — Sample points
vector

Sample points, specified as the comma-separated pair consisting of `'SamplePoints'` and a vector. The sample points represent the location of the data in `A`, and must be sorted and contain unique elements. Sample points do not need to be uniformly sampled. If `A` is a timetable, then the default sample points vector is the vector of row times. Otherwise, the default vector is `[1 2 3 ...]`.

Moving windows are defined relative to the sample points. For example, if `t` is a vector of times corresponding to the input data, then `filloutliers(rand(1,10),'previous','movmean',3,'SamplePoints',t)` has a window that represents the time interval between `t(i)-1.5` and `t(i)+1.5`.

When the sample points vector has data type `datetime` or `duration`, then the moving window length must have type `duration`.

Data Types: `single` | `double` | `datetime` | `duration`

`DataVariables` — Table variables
variable name | cell array of variable names | numeric vector | logical vector | function handle

Table variables, specified as the comma-separated pair consisting of `'DataVariables'` and a variable name, a cell array of variable names, a numeric vector, a logical vector, or a function handle. The `'DataVariables'` value indicates which columns of the input table to detect outliers in, and can be one of the following:

- A character vector specifying a single table variable name
- A cell array of character vectors where each element is a table variable name
- A vector of table variable indices
- A logical vector whose elements each correspond to a table variable, where `true` includes the corresponding variable and `false` excludes it
- A function handle that takes the table as input and returns a logical scalar

Example: `'Age'`

Example: `{'Height','Weight'}`

Example: `@isnumeric`

Data Types: `char` | `cell` | `double` | `single` | `logical` | `function_handle`

`MaxNumOutliers` — Maximum outlier count
positive scalar

Maximum outlier count, for the `'gesd'` method only, specified as the comma-separated pair consisting of `'MaxNumOutliers'` and a positive scalar. The `'MaxNumOutliers'` value specifies the maximum number of outliers returned by the `'gesd'` method. For example, `filloutliers(A,'linear','gesd','MaxNumOutliers',5)` returns no more than five outliers.

The default value for `'MaxNumOutliers'` is the integer nearest to 10 percent of the number of elements in `A`. Setting a larger value for the maximum number of outliers can ensure that all outliers are detected, but at the cost of reduced computational efficiency.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`OutlierLocations` — Known outlier indicator
vector | matrix | multidimensional array

Known outlier indicator, specified as the comma-separated pair consisting of `'OutlierLocations'` and a logical vector, matrix, or multidimensional array of the same size as `A`. The known outlier indicator elements can be `true` to indicate an outlier in the corresponding location of `A` or `false` otherwise. Specifying `'OutlierLocations'` turns off the default outlier detection method, and uses only the elements of the known outlier indicator to define outliers.

The `'OutlierLocations'` name-value pair cannot be specified when `findmethod` is specified.

The output `TF` is the same as the `'OutlierLocations'` value.

Data Types: `logical`

Output Arguments

B — Filled outlier array

vector | matrix | multidimensional array | table | timetable

Filled outlier array, returned as a vector, matrix, multidimensional array, table, or timetable. The elements of `B` are the same as those of `A`, but with all outliers replaced according to `fillmethod`.

Data Types: `double` | `single` | `table` | `timetable`

TF — Outlier indicator

vector | matrix | multidimensional array

Outlier indicator, returned as a vector, matrix, or multidimensional array. An element of `TF` is `true` when the corresponding element of `A` is an outlier and `false` otherwise. `TF` is the same size as `A`.

Data Types: `logical`

L — Lower threshold

scalar | vector | matrix | multidimensional array | table | timetable

Lower threshold used by the outlier detection method, returned as a scalar, vector, matrix, multidimensional array, table, or timetable. For example, the lower value of the default outlier detection method is three scaled MAD below the median of the input data. `L` has the same size as `A` in all dimensions except for the operating dimension where the length is 1.

Data Types: `double` | `single` | `table` | `timetable`

`U` — Upper threshold
scalar | vector | matrix | multidimensional array | table | timetable

Upper threshold used by the outlier detection method, returned as a scalar, vector, matrix, multidimensional array, table, or timetable. For example, the upper value of the default outlier detection method is three scaled MAD above the median of the input data. `U` has the same size as `A` in all dimensions except for the operating dimension where the length is 1.

Data Types: `double` | `single` | `table` | `timetable`

`C` — Center value
scalar | vector | matrix | multidimensional array | table | timetable

Center value used by the outlier detection method, returned as a scalar, vector, matrix, multidimensional array, table, or timetable. For example, the center value of the default outlier detection method is the median of the input data. `C` has the same size as `A` in all dimensions except for the operating dimension where the length is 1.

Data Types: `double` | `single` | `table` | `timetable`

More About

Median Absolute Deviation

For a random variable vector A made up of N scalar observations, the median absolute deviation (MAD) is defined as

$$\text{MAD} = \text{median}(|A_i - \text{median}(A)|)$$

for $i = 1,2,...,N$.

The scaled MAD is defined as `c*median(abs(A-median(A)))` where `c=-1/(sqrt(2)*erfcinv(3/2))`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- The `'percentiles'`, `'grubs'`, and `'gesd'` methods are not supported.
- The `'movmedian'` and `'movmean'` methods do not support tall timetables.
- The `'SamplePoints'` and `'MaxNumOutliers'` name-value pairs are not supported.
- The value of `'DataVariables'` cannot be a function handle.
- Computation of `filloutliers(A,fillmethod)`, `filloutliers(A,fillmethod,'median',…)` or `filloutliers(A,fillmethod,'quartiles',…)` along the first dimension is only supported when `A` is a tall column vector.
- The syntaxes `filloutliers(A,'spline',…)` and `filloutliers(A,'makima',…)` are not supported.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The `'movmean'` and `'movmedian'` methods do not support the `'SamplePoints'` name-value pair argument.
- To use the `'spline'` and `'pchip'` fill methods, you must enable support for variable-size arrays.
- String and character array inputs must be constant.
- The `'makima'` options is not supported.
- Tables are not supported.

See Also

Clean Outlier Data | `fillmissing` | `ismissing` | `isoutlier` | `rmoutliers`

Topics

Clean Messy Data and Locate Extrema Using Live Editor Tasks
"Data Smoothing and Outlier Detection"

Introduced in R2017a

filter

1-D digital filter

Syntax

```
y = filter(b,a,x)
y = filter(b,a,x,zi)
y = filter(b,a,x,zi,dim)
[y,zf] = filter( ___ )
```

Description

`y = filter(b,a,x)` filters the input data `x` using a rational transfer function on page 1-4329 defined by the numerator and denominator coefficients `b` and `a`.

If `a(1)` is not equal to `1`, then `filter` normalizes the filter coefficients by `a(1)`. Therefore, `a(1)` must be nonzero.

- If `x` is a vector, then `filter` returns the filtered data as a vector of the same size as `x`.
- If `x` is a matrix, then `filter` acts along the first dimension and returns the filtered data for each column.
- If `x` is a multidimensional array, then `filter` acts along the first array dimension whose size does not equal 1.

`y = filter(b,a,x,zi)` uses initial conditions `zi` for the filter delays. The length of `zi` must equal `max(length(a),length(b))-1`.

`y = filter(b,a,x,zi,dim)` acts along dimension `dim`. For example, if `x` is a matrix, then `filter(b,a,x,zi,2)` returns the filtered data for each row.

`[y,zf] = filter( ___ )` also returns the final conditions `zf` of the filter delays, using any of the previous syntaxes.

Examples

Moving-Average Filter

A moving-average filter is a common method used for smoothing noisy data. This example uses the `filter` function to compute averages along a vector of data.

Create a 1-by-100 row vector of sinusoidal data that is corrupted by random noise.

```
t = linspace(-pi,pi,100);
rng default  %initialize random number generator
x = sin(t) + 0.25*rand(size(t));
```

A moving-average filter slides a window of length *windowSize* along the data, computing averages of the data contained in each window. The following difference equation defines a moving-average filter of a vector x:

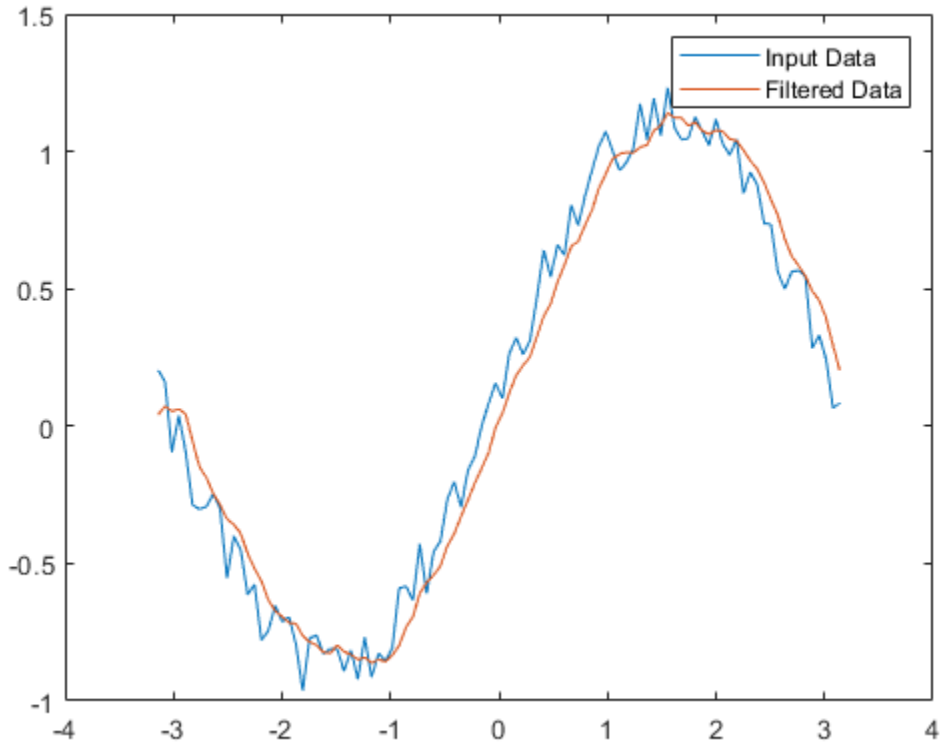
$$y(n) = \frac{1}{windowSize}(x(n) + x(n-1) + \ . \ . \ . \ + x(n-(windowSize-1)))\,.$$

For a window size of 5, compute the numerator and denominator coefficients for the rational transfer function.

```
windowSize = 5;
b = (1/windowSize)*ones(1,windowSize);
a = 1;
```

Find the moving average of the data and plot it against the original data.

```
y = filter(b,a,x);

plot(t,x)
hold on
plot(t,y)
legend('Input Data','Filtered Data')
```



Filter Matrix Rows

This example filters a matrix of data with the following rational transfer function.

$$H(z) = \frac{b(1)}{a(1) + a(2)z^{-1}} = \frac{1}{1 - 0.2z^{-1}}$$

Create a 2-by-15 matrix of random input data.

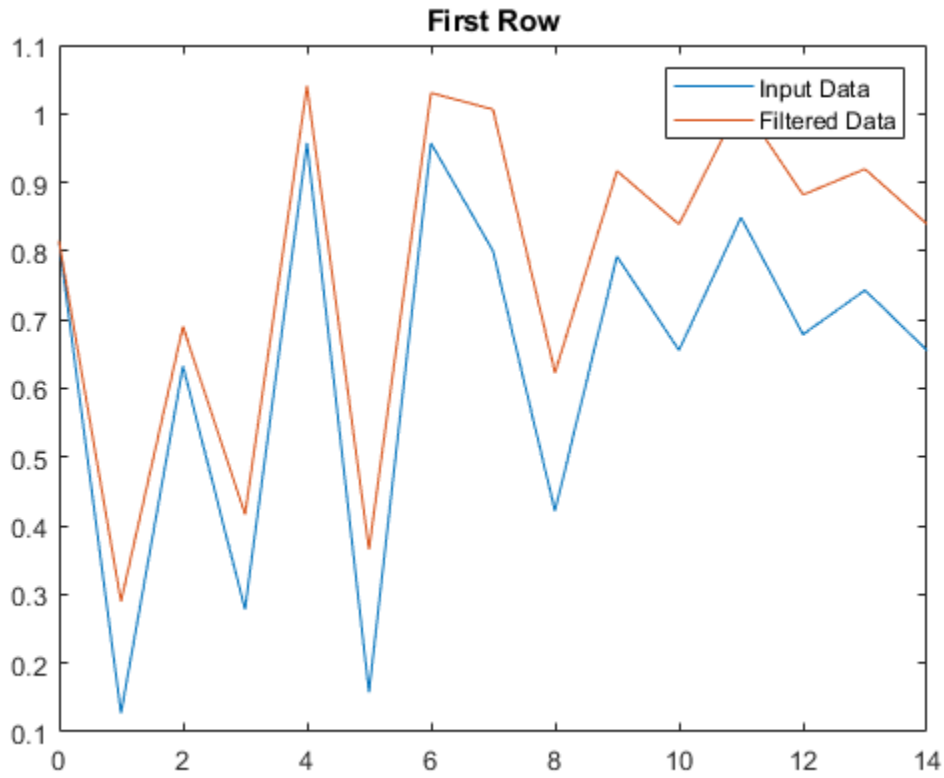
```
rng default  %initialize random number generator
x = rand(2,15);
```

Define the numerator and denominator coefficients for the rational transfer function.

```
b = 1;
a = [1 -0.2];
```

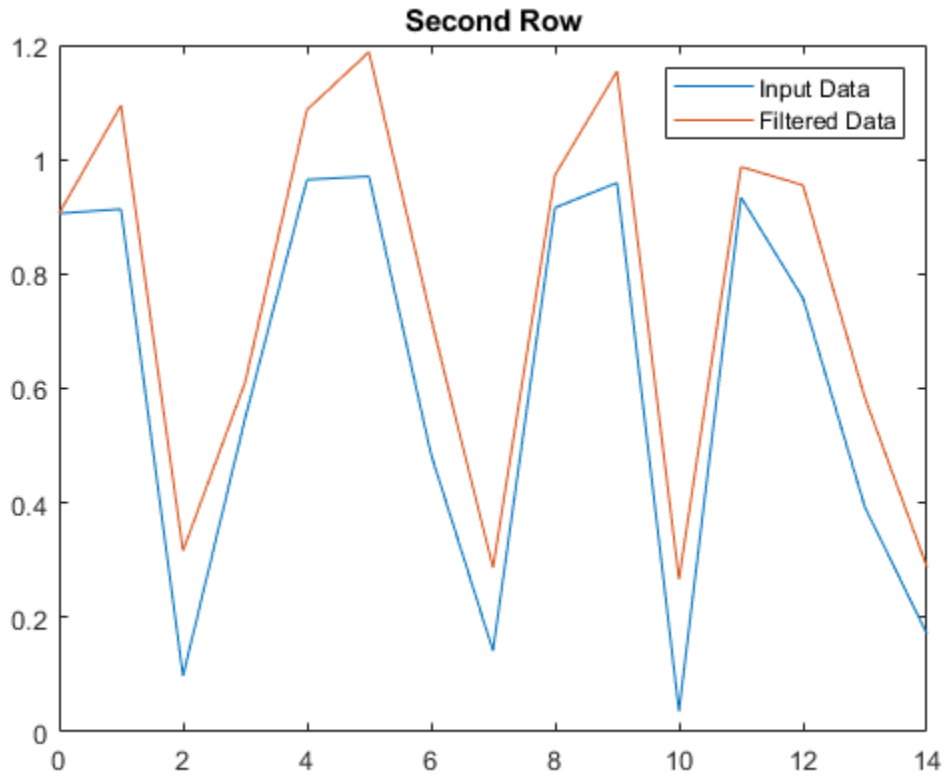
Apply the transfer function along the second dimension of x and return the 1-D digital filter of each row. Plot the first row of original data against the filtered data.

```
y = filter(b,a,x,[],2);

t = 0:length(x)-1;  %index vector

plot(t,x(1,:))
hold on
plot(t,y(1,:))
legend('Input Data','Filtered Data')
title('First Row')
```



Plot the second row of input data against the filtered data.

```
figure
plot(t,x(2,:))
hold on
plot(t,y(2,:))
legend('Input Data','Filtered Data')
title('Second Row')
```



Filter Data in Sections

Use initial and final conditions for filter delays to filter data in sections, especially if memory limitations are a consideration.

Generate a large random data sequence and split it into two segments, `x1` and `x2`.

```
x = randn(10000,1);

x1 = x(1:5000);
x2 = x(5001:end);
```

The whole sequence, x, is the vertical concatenation of x1 and x2.

Define the numerator and denominator coefficients for the rational transfer function,

$$H(z) = \frac{b(1) + b(2)z^{-1}}{a(1) + a(2)z^{-1}} = \frac{2 + 3z^{-1}}{1 + 0.2z^{-1}}.$$

```
b = [2,3];
a = [1,0.2];
```

Filter the subsequences x1 and x2 one at a time. Output the final conditions from filtering x1 to store the internal status of the filter at the end of the first segment.

```
[y1,zf] = filter(b,a,x1);
```

Use the final conditions from filtering x1 as initial conditions to filter the second segment, x2.

```
y2 = filter(b,a,x2,zf);
```

y1 is the filtered data from x1, and y2 is the filtered data from x2. The entire filtered sequence is the vertical concatenation of y1 and y2.

Filter the entire sequence simultaneously for comparison.

```
y = filter(b,a,x);
```

```
isequal(y,[y1;y2])
```

```
ans = logical
   1
```

Input Arguments

b — Numerator coefficients of rational transfer function
vector

Numerator coefficients of the rational transfer function on page 1-4329, specified as a vector.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

a — Denominator coefficients of rational transfer function
vector

Denominator coefficients of the rational transfer function on page 1-4329, specified as a vector.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

x — Input data
vector | matrix | multidimensional array

Input data, specified as a vector, matrix, or multidimensional array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

`zi` — Initial conditions for filter delays
`[]` (default) | vector | matrix | multidimensional array

Initial conditions for filter delays, specified as a vector, matrix, or multidimensional array.

- If `zi` is a vector, then its length must be `max(length(a),length(b))-1`.
- If `zi` is a matrix or multidimensional array, then the size of the leading dimension must be `max(length(a),length(b))-1`. The size of each remaining dimension must match the size of the corresponding dimension of `x`. For example, consider using `filter` along the second dimension (`dim = 2`) of a 3-by-4-by-5 array `x`. The array `zi` must have size [`max(length(a),length(b))-1`]-by-3-by-5.

The default value, specified by `[]`, initializes all filter delays to zero.

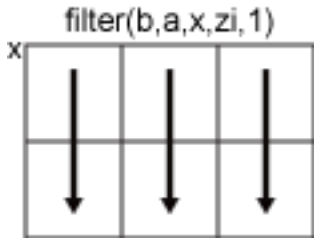
Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

`dim` — Dimension to operate along
positive integer scalar

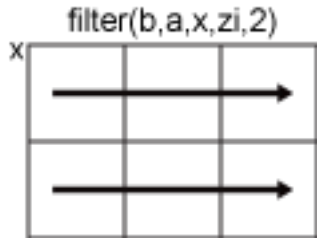
Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a two-dimensional input array, `x`.

- If `dim = 1`, then `filter(b,a,x,zi,1)` works along the rows of `x` and returns the filter applied to each column.



- If `dim = 2`, then `filter(b,a,x,zi,2)` works along the columns of `x` and returns the filter applied to each row.



If `dim` is greater than `ndims(x)`, then `filter` returns `x`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Output Arguments

y — Filtered data

vector | matrix | multidimensional array

Filtered data, returned as a vector, matrix, or multidimensional array of the same size as the input data, `x`.

If `x` is of type `single`, then `filter` natively computes in single precision, and `y` is also of type `single`. Otherwise, `y` is returned as type `double`.

Data Types: `double` | `single`

`zf` — Final conditions for filter delays
vector | matrix | multidimensional array

Final conditions for filter delays, returned as a vector, matrix, or multidimensional array.

- If `x` is a vector, then `zf` is a column vector of length `max(length(a),length(b))-1`.
- If `x` is a matrix or multidimensional array, then `zf` is an array of column vectors of length `max(length(a),length(b))-1`, such that the number of columns in `zf` is equivalent to the number of columns in `x`. For example, consider using `filter` along the second dimension (`dim = 2`) of a 3-by-4-by-5 array `x`. The array `zf` has size [`max(length(a),length(b))-1`]-by-3-by-5.

Data Types: `double` | `single`

More About

Rational Transfer Function

The input-output description of the `filter` operation on a vector in the Z-transform domain is a rational transfer function. A rational transfer function is of the form

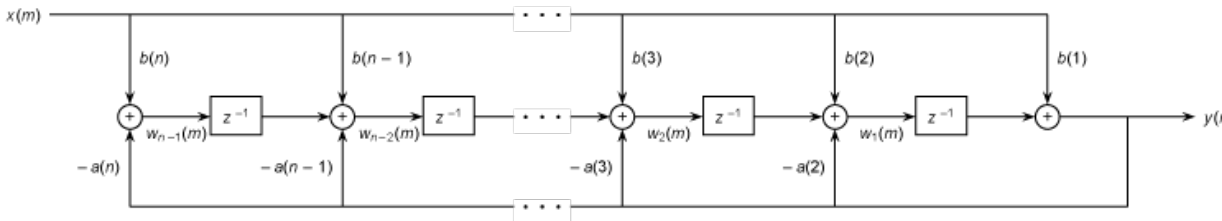
$$Y(z) = \frac{b(1) + b(2)z^{-1} + \ldots + b(n_b + 1)z^{-n_b}}{1 + a(2)z^{-1} + \ldots + a(n_a + 1)z^{-n_a}} X(z),$$

which handles both FIR and IIR filters [1]. n_a is the feedback filter order, and n_b is the feedforward filter order. Due to normalization, assume $a(1) = 1$.

You also can express the rational transfer function as the difference equation

$$\begin{aligned} a(1)y(n) = {} & b(1)x(n) + b(2)x(n-1) + \ldots + b(n_b + 1)x(n - n_b) \\ & -a(2)y(n-1) - \ldots - a(n_a + 1)y(n - n_a)\,. \end{aligned}$$

Furthermore, you can represent the rational transfer function using its direct-form II transposed implementation, as in the following diagram. Here, $n_a = n_b$.



The operation of `filter` at sample m is given by the time-domain difference equations

$$
\begin{aligned}
y(m) &= b(1)x(m) + w_1(m-1) \\
w_1(m) &= b(2)x(m) + w_2(m-1) - a(2)y(m) \\
\vdots \quad &= \quad \vdots \qquad\qquad \vdots \\
w_{n-2}(m) &= b(n-1)x(m) + w_{n-1}(m-1) - a(n-1)y(m) \\
w_{n-1}(m) &= b(n)x(m) - a(n)y(m)\,.
\end{aligned}
$$

Tips

- If you have Signal Processing Toolbox™, use `y = filter(d,x)` to filter an input signal `x` with a `digitalFilter` object `d`. To generate `d` based on frequency-response specifications, use `designfilt`.
- To use the `filter` function with the `b` coefficients from an FIR filter, use `y = filter(b,1,x)`.
- See Digital Filtering (Signal Processing Toolbox) for more on filtering functions.

References

[1] Oppenheim, Alan V., Ronald W. Schafer, and John R. Buck. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1999.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

The two-output syntax `[y,zf] = filter(___)` is not supported when `dim > 1`.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If supplied, `dim` must be a constant.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Supported syntaxes:

  ```
  Y = filter(B,A,X)
  ```

  ```
  Y = filter(B,A,X,[],dim)
  ```

  ```
  [Y,zf] = filter(B,A,X,zi)
  ```

  ```
  [Y,zf] = filter(B,A,X,zi,dim)
  ```
- If `A` is not scalar, then `A` and `B` must each have no more than 10 elements.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`conv` | `filter2`

Topics

“Filter Data”

Introduced before R2006a

filter2

2-D digital filter

Syntax

```
Y = filter2(H,X)
Y = filter2(H,X,shape)
```

Description

`Y = filter2(H,X)` applies a finite impulse response filter to a matrix of data `X` according to coefficients in a matrix `H`.

`Y = filter2(H,X,shape)` returns a subsection of the filtered data according to `shape`. For example, `Y = filter2(H,X,'valid')` returns only filtered data computed without zero-padded edges.

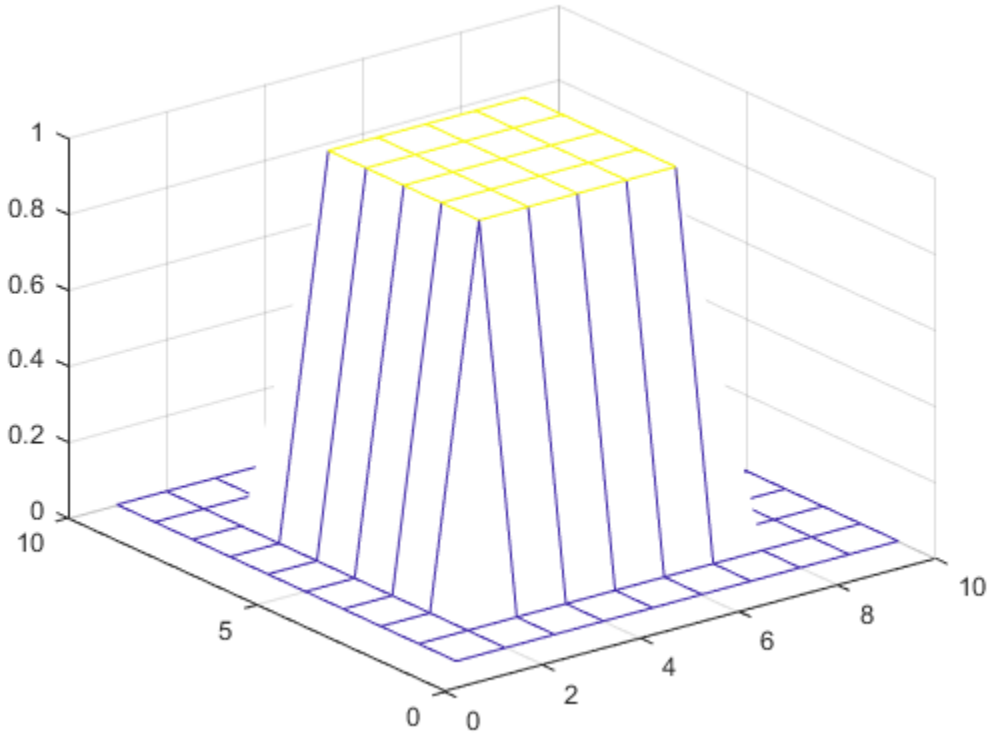
Examples

2-D Pedestal

You can digitally filter images and other 2-D data using the `filter2` function, which is closely related to the `conv2` function.

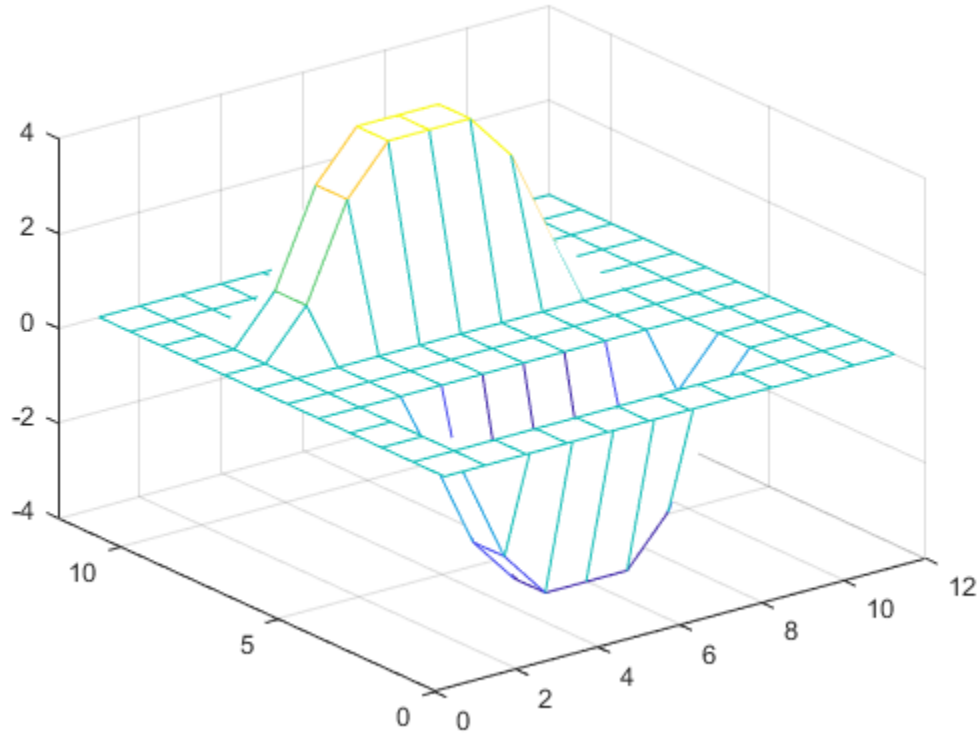
Create and plot a 2-D pedestal with interior height equal to one.

```
A = zeros(10);
A(3:7,3:7) = ones(5);
mesh(A)
```



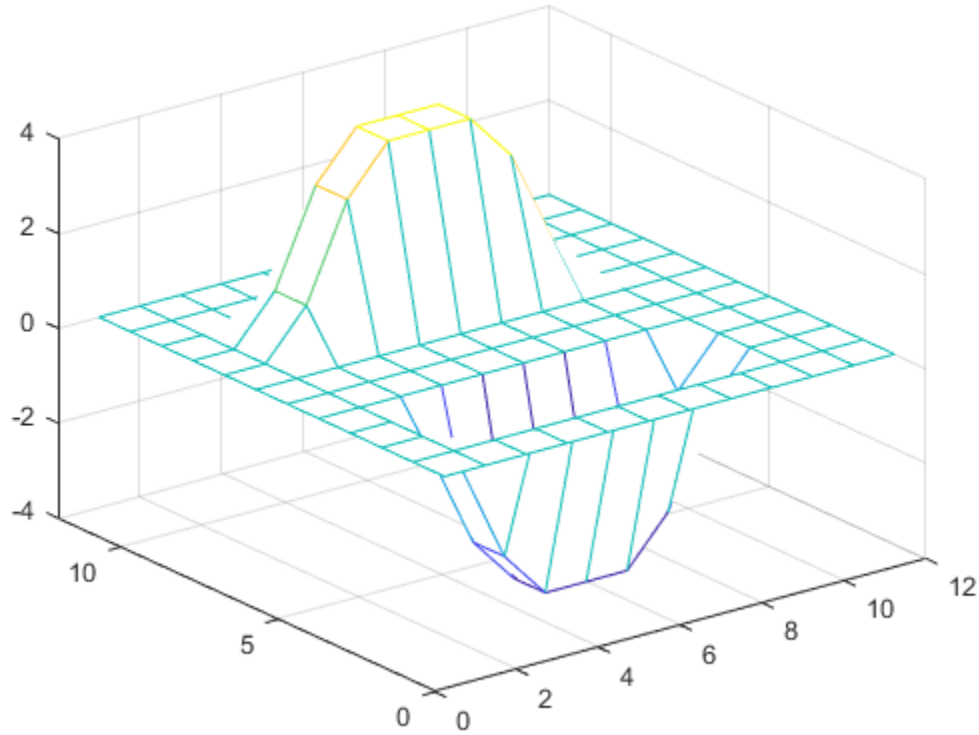
Filter the data in A according to a filter coefficient matrix H, and return the full matrix of filtered data.

```
H = [1 2 1; 0 0 0; -1 -2 -1];
Y = filter2(H,A,'full');
mesh(Y)
```



Rotate H 180 degrees and convolve the result with A. The output is equivalent to filtering the data in A with the coefficients in H.

```
C = conv2(A,rot90(H,2));
mesh(C)
```



Input Arguments

`H` — Coefficients of rational transfer function
matrix

Coefficients of the rational transfer function, specified as a matrix.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

X — Input data
matrix

Input data, specified as a matrix. If one or both of X and H are of type `single`, then the output is also of type `single`. Otherwise, `filter2` returns type `double`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

shape — Subsection of filtered data
`'same'` (default) | `'full'` | `'valid'`

Subsection of the filtered data, specified as one of these values:

- `'same'` — Return the central part of the filtered data, which is the same size as X.
- `'full'` — Return the full 2-D filtered data.
- `'valid'` — Return only parts of the filtered data that are computed without zero-padded edges.

Algorithms

The `filter2` function filters data by taking the 2-D convolution of the input X and the coefficient matrix H rotated 180 degrees. Specifically, `filter2(H,X,shape)` is equivalent to `conv2(X,rot90(H,2),shape)`.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The inputs H and X must be single or double arrays. Both real and complex types are supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

conv2 | filter | rot90

Introduced before R2006a

fimplicit

Plot implicit function

Syntax

```
fimplicit(f)
fimplicit(f,interval)
fimplicit(ax, ___ )

fimplicit( ___ ,LineSpec)
fimplicit( ___ ,Name,Value)
fp = fimplicit( ___ )
```

Description

`fimplicit(f)` plots the implicit function defined by `f(x,y) = 0` over the default interval `[-5 5]` for `x` and `y`.

`fimplicit(f,interval)` specifies the plotting interval for `x` and `y`.

`fimplicit(ax, ___ )` plots into the axes specified by `ax` instead of into the current axes. Specify the axes as the first input argument, prior to any of the previous input arguments.

`fimplicit( ___ ,LineSpec)` specifies the line style, marker symbol, and line color. For example, `'-r'` plots a red line.

`fimplicit( ___ ,Name,Value)` specifies line properties using one or more name-value pair arguments. For example, `'LineWidth',2` specifies a line width of 2 points.

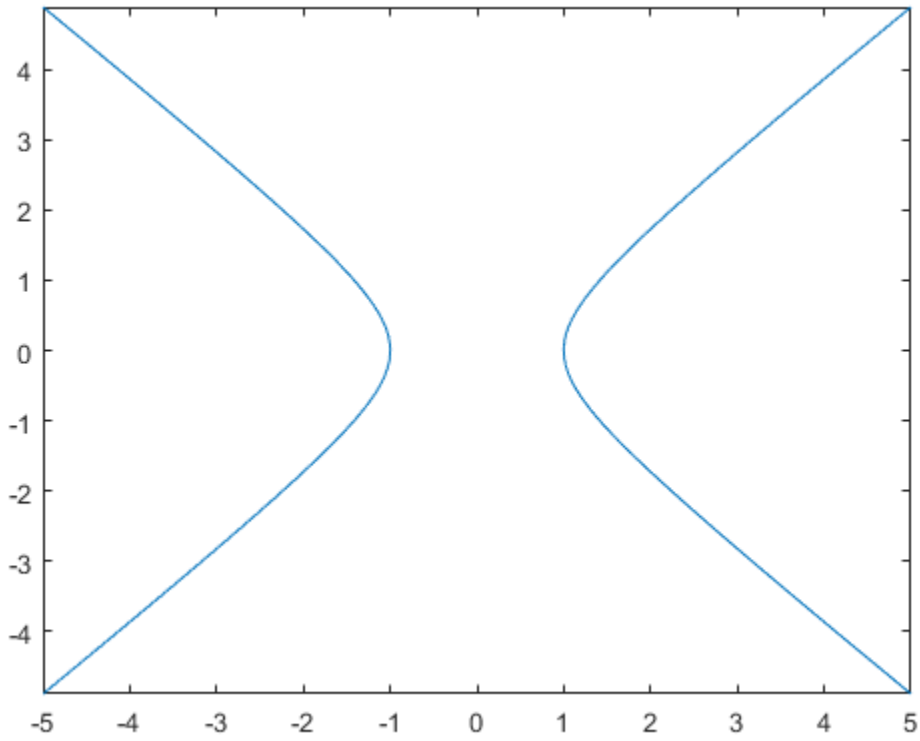
`fp = fimplicit( ___ )` returns the `ImplicitFunctionLine` object. Use `fp` to access and modify properties of the line after it is created. For a list of properties, see ImplicitFunctionLine.

Examples

Plot Implicit Function

Plot the hyperbola described by the function $x^2 - y^2 - 1 = 0$ over the default interval of `[-5 5]` for x and y.

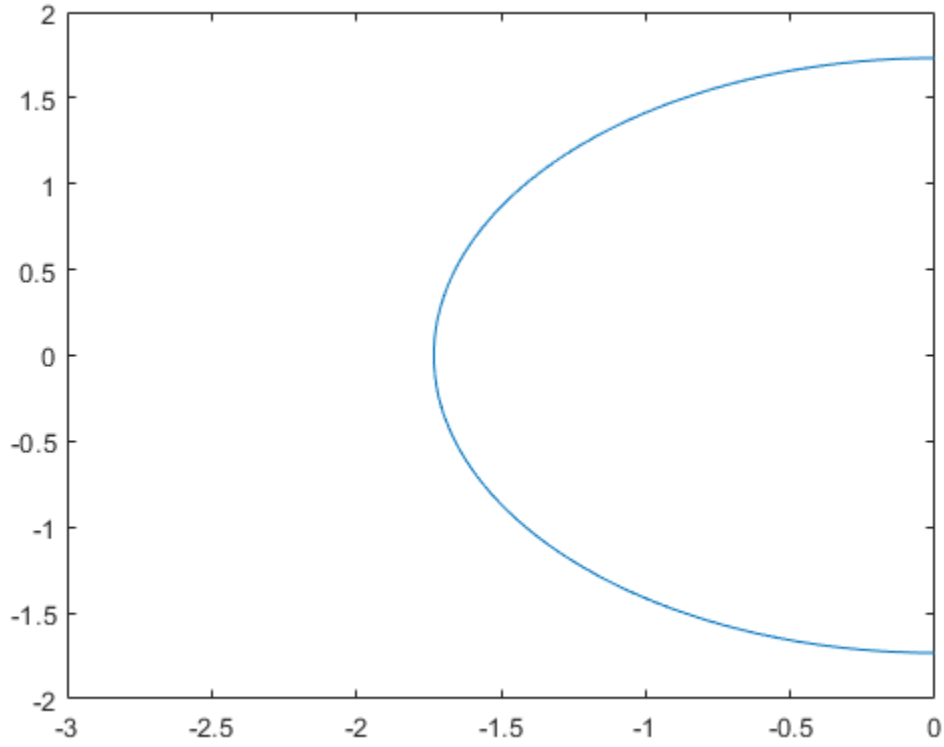
```
fimplicit(@(x,y) x.^2 - y.^2 - 1)
```



Specify Plotting Interval

Plot the function $x^2 + y^2 - 3 = 0$ over the intervals `[-3 0]` for `x` and `[-2 2]` for `y`.

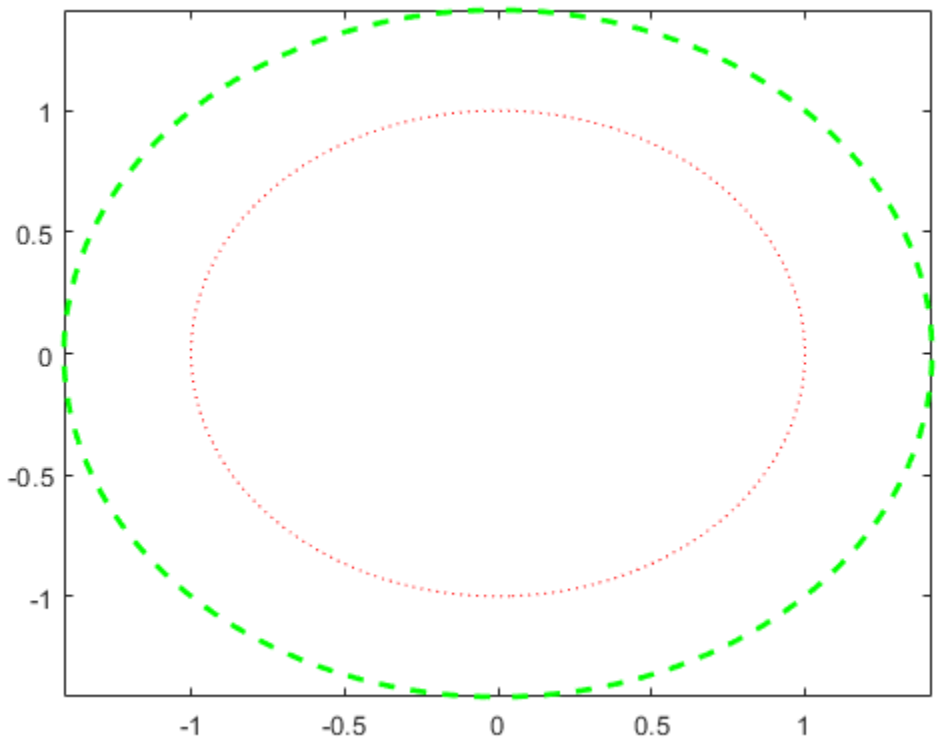
```
f = @(x,y) x.^2 + y.^2 - 3;
fimplicit(f,[-3 0 -2 2])
```



Modify Appearance of Implicit Plot

Plot two circles centered at `(0,0)` with different radius values. For the first circle, use a dotted, red line. For the second circle, use a dashed, green line with a line width of 2 points.

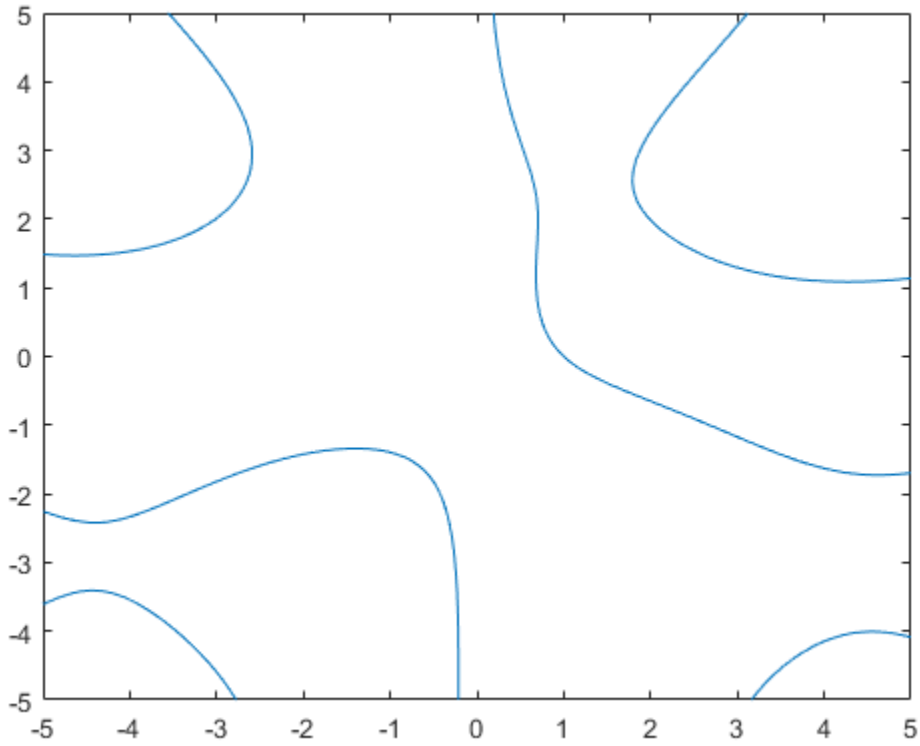
```
f1 = @(x,y) x.^2 + y.^2 - 1;
fimplicit(f1,':r')

hold on
f2 = @(x,y) x.^2 + y.^2 - 2;
fimplicit(f2,'--g','LineWidth',2)
hold off
```



Modify Implicit Plot After Creation

Plot the implicit function $y\sin(x) + x\cos(y) - 1 = 0$ and assign the implicit function line object to the variable `fp`.

```
fp = fimplicit(@(x,y) y.*sin(x) + x.*cos(y) - 1)
```

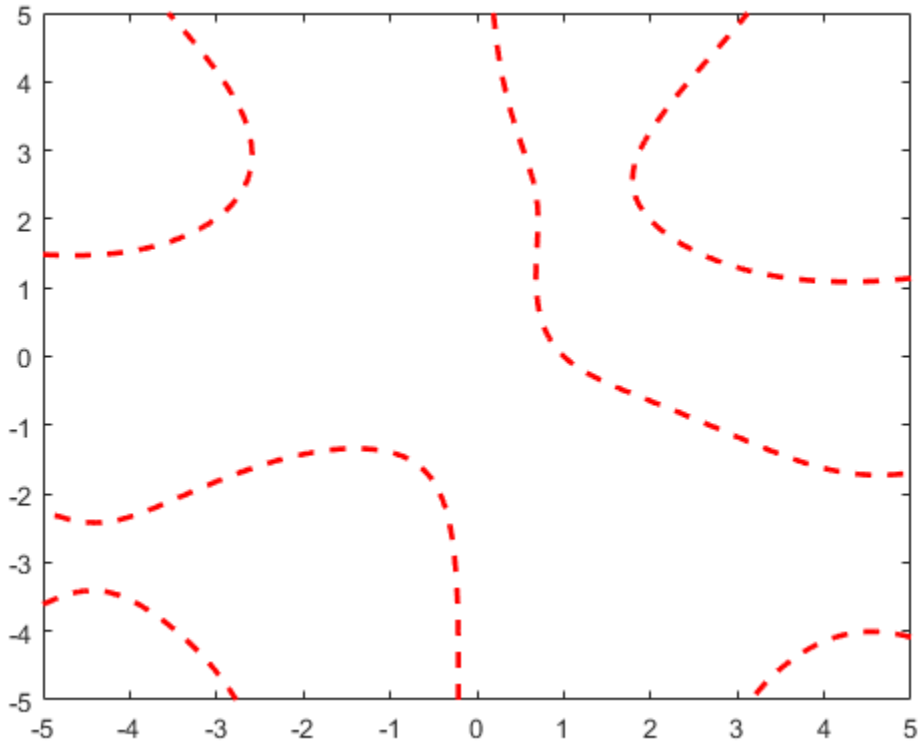



```
fp =
  ImplicitFunctionLine with properties:

     Function: @(x,y)y.*sin(x)+x.*cos(y)-1
        Color: [0 0.4470 0.7410]
```

```
    LineStyle: '-'
    LineWidth: 0.5000

  Show all properties
```

Use `fp` to access and modify properties of the implicit function line object after it is created. For example, change the color, line style, and line width.

```
fp.Color = 'r';
fp.LineStyle = '--';
fp.LineWidth = 2;
```



Input Arguments

`f` — Implicit function to plot

function handle

Implicit function to plot, specified as a function handle to a named or anonymous function.

Specify a function of the form `z = f(x,y)`. The function must accept two matrix input arguments and return a matrix output argument of the same size. Use array operators instead of matrix operators for the best performance. For example, use `.*` (`times`) instead of `*` (`mtimes`).

Example: `fimplicit(@(x,y) x.^2 - y.^2 + 1)`

`interval` — Plotting interval for `x` and `y`

`[-5 5]` (default) | two-element vector | four-element vector

Plotting interval for `x` and `y`, specified in one of these forms:

- Two-element vector of the form `[min max]` — Use the same plotting interval of `[min max]` for both `x` and `y`.
- Four-element vector of the form `[xmin xmax ymin ymax]` — Use different plotting intervals for `x` and `y`. Plot over the interval `[xmin xmax]` for `x` and `[ymin ymax]` for `y`.

Example: `fimplicit(f,[-2 3 -5 0])`

`LineSpec` — Line style, marker symbol, and color

character vector | string

Line style, marker symbol, and color, specified as a character vector or string containing symbols. The symbols can appear in any order. You do not need to specify all three characteristics (line style, marker symbol, and color). For example, if you omit the line style and specify the marker, then the plot shows only the markers and no line.

Example: `fimplicit(f,'--or')` plots a red, dashed line with circle markers.

Line Style	Description
`-`	Solid line (default)
`--`	Dashed line

Line Style	Description
`:`	Dotted line
`-.`	Dash-dot line

Marker	Description
`o`	Circle
`+`	Plus sign
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Pentagram
`h`	Hexagram

Color	Description
`y`	yellow
`m`	magenta
`c`	cyan
`r`	red
`g`	green
`b`	blue
`w`	white
`k`	black

ax — Axes object
axes object

Axes object. If you do not specify the axes, then `fimplicit` uses the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `fimplicit(f,'MeshDensity',50,'LineWidth',2)` specifies the number of evaluation points and the line width.

The `ImplicitFunctionLine` properties listed here are only a subset. For a complete list, see ImplicitFunctionLine.

`MeshDensity` — Number of evaluation points per direction
`151` (default) | scalar

Number of evaluation points per direction, specified as a scalar.

`Color` — Line color
`[0 0 1]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

`Marker` — Marker symbol
`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | `'x'` | `...`

Marker symbol, specified as one of the markers listed in this table. By default, the line does not have markers. Specifying a marker symbol adds markers at selected points along the line.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle

Value	Description
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'+'`

Example: `'diamond'`

`MarkerSize` — Marker size
6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

`MarkerEdgeColor` — Marker outline color
`'auto'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'auto'` uses the same color as the `Color` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` value uses the same color as the `MarkerEdgeColor` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Example: `'#D2F9A7'`

Tips

- Use element-wise operators for the best performance and to avoid a warning message. For example, use `x.*y` instead of `x*y`. For more information, see "Array vs. Matrix Operations".
- When you zoom in on the chart, `fimplicit` recalculates the data, which can reveal hidden details.

See Also

Functions
`fcontour` | `fimplicit3` | `fplot` | `fplot3` | `fsurf`

Properties
ImplicitFunctionLine

Introduced in R2016b

fimplicit3

Plot 3-D implicit function

Syntax

```
fimplicit3(f)
fimplicit3(f,interval)
fimplicit3(ax, ___ )

fimplicit3( ___ ,LineSpec)
fimplicit3( ___ ,Name,Value)
fs = fimplicit3( ___ )
```

Description

`fimplicit3(f)` plots the 3-D implicit function defined by `f(x,y,z) = 0` over the default interval `[-5 5]` for `x`, `y`, and `z`.

`fimplicit3(f,interval)` specifies the plotting interval for `x`, `y`, and `z`.

`fimplicit3(ax, ___ )` plots into the axes specified by `ax` instead of into the current axes. Specify the axes as the first input argument, prior to any of the previous input arguments.

`fimplicit3( ___ ,LineSpec)` specifies the line style, marker symbol, and line color. For example, `'-r'` specifies red lines.

`fimplicit3( ___ ,Name,Value)` specifies surface properties using one or more name-value pair arguments. For example, `'FaceAlpha',0.6` specifies a transparency value of `0.6` for a semi-transparent surface.

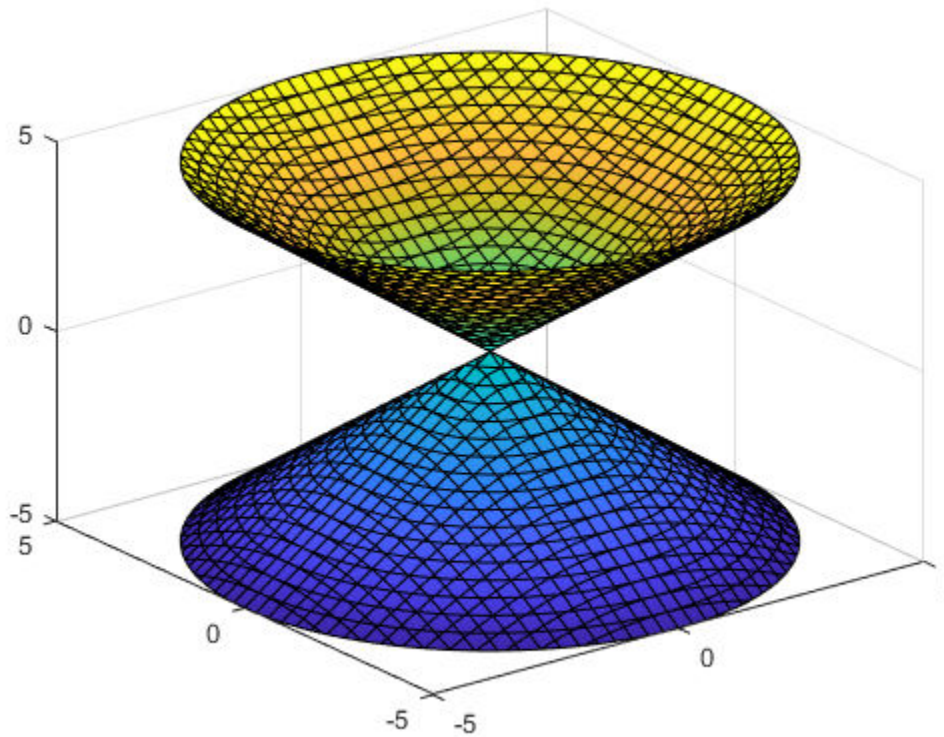
`fs = fimplicit3( ___ )` returns the `ImplicitFunctionSurface` object. Use `fs` to access and modify properties of the surface after it is created. For a list of properties, see ImplicitFunctionSurface.

Examples

Plot 3-D Implicit Function

Plot the hyperboloid $x^2 + y^2 - z^2 = 0$ over the default interval of $[-5, 5]$ for x, y, and z.

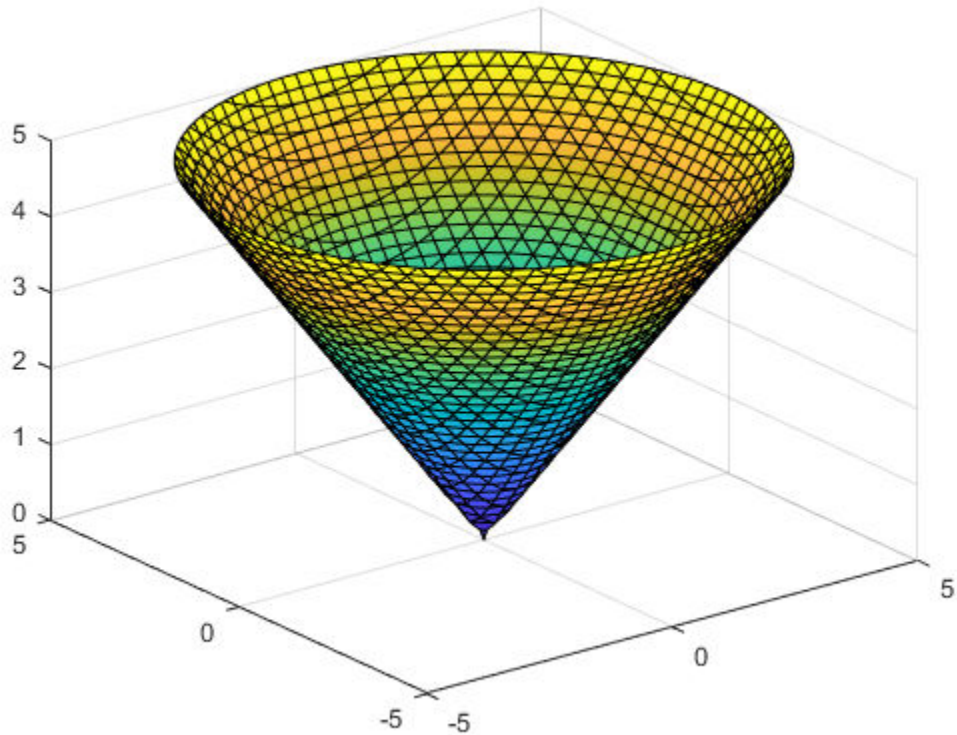
```
f = @(x,y,z) x.^2 + y.^2 - z.^2;
fimplicit3(f)
```



Specify Plotting Interval

Plot the upper half of the hyperboloid $x^2 + y^2 - z^2 = 0$ by specifying the plotting interval as [0 5] for z. For x and y, use the default interval `[-5 5]`.

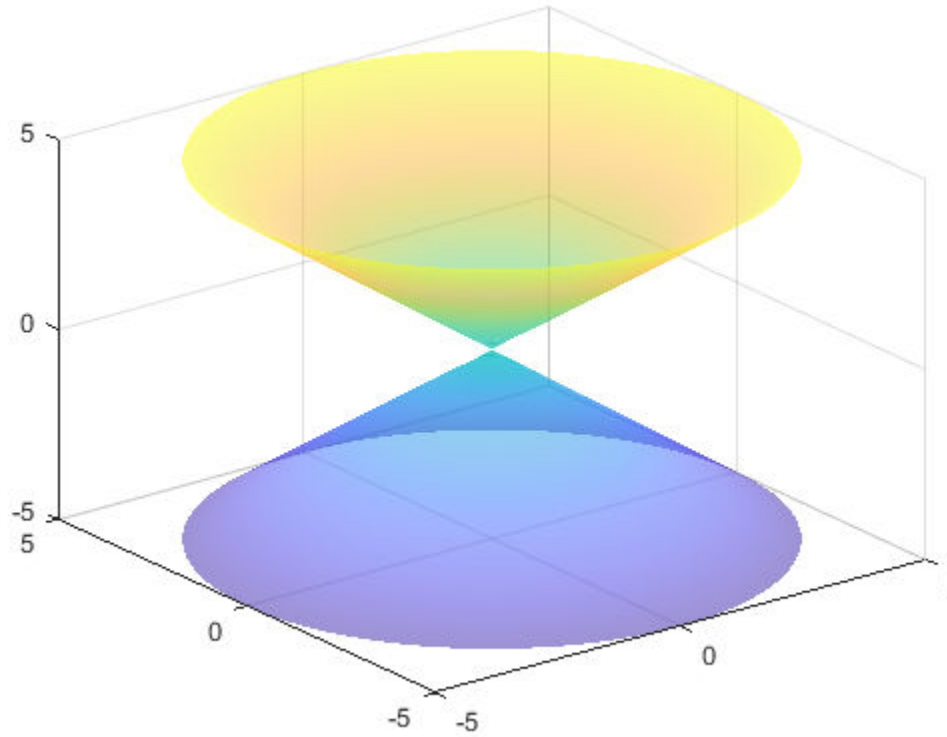
```
f = @(x,y,z) x.^2 + y.^2 - z.^2;
interval = [-5 5 -5 5 0 5];
fimplicit3(f,interval)
```



Modify Appearance of Implicit Surface

Plot the implicit surface $x^2 + y^2 - z^2 = 0$. Remove the lines by setting the `EdgeColor` property to `'none'`. Add transparency by setting the `FaceAlpha` property to a value between 0 and 1.

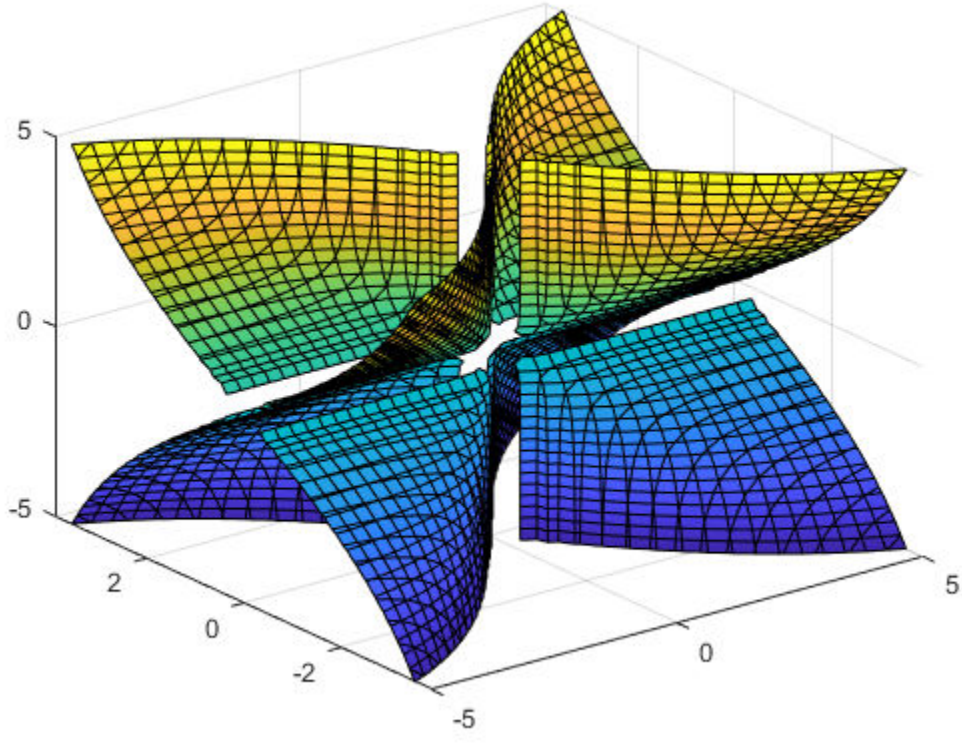
```
f = @(x,y,z) x.^2 + y.^2 - z.^2;
fimplicit3(f,'EdgeColor','none','FaceAlpha',.5)
```



Modify Implicit Surface After Creation

Plot an implicit surface and assign the implicit surface object to the variable `fs`.

```
f = @(x,y,z) 1./x.^2 - 1./y.^2 + 1./z.^2;
fs = fimplicit3(f)
```

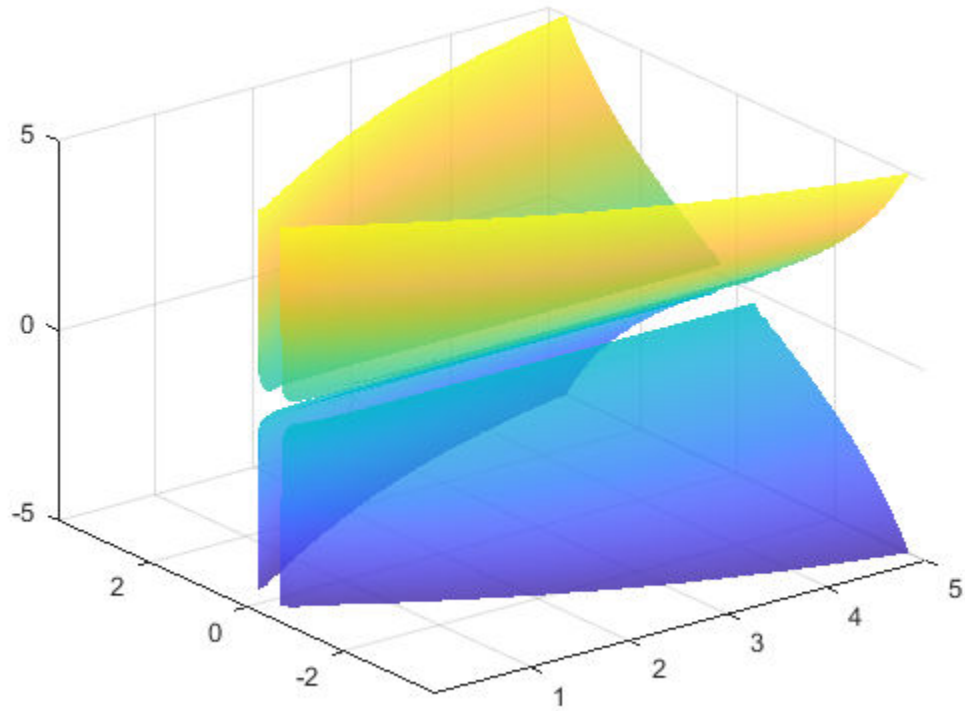



```
fs =
  ImplicitFunctionSurface with properties:

     Function: @(x,y,z)1./x.^2-1./y.^2+1./z.^2
    EdgeColor: [0 0 0]
    LineStyle: '-'
    FaceColor: 'interp'
```

```
Show all properties
```

Use `fs` to access and modify properties of the implicit surface after it is created. For example, show only the positive *x* values by setting the `XRange` property to `[0 5]`. Remove the lines by setting the `EdgeColor` property to `'none'`. Add transparency by setting the `FaceAlpha` property to `0.8`.

```
fs.XRange = [0 5];
fs.EdgeColor = 'none';
fs.FaceAlpha = 0.8;
```



Input Arguments

f — 3-D implicit function to plot

function handle

3-D implicit function to plot, specified as a function handle to a named or anonymous function.

Specify a function of the form `w = f(x,y,z)`. The function must accept three 3-D array input arguments and return a 3-D array output argument of the same size. Use array operators instead of matrix operators for the best performance. For example, use `.*` (`times`) instead of `*` (`mtimes`).

Example: `fimplicit3(@(x,y,z) x.^2 + y.^2 - z.^2)`

interval — Plotting interval for x, y, and z

`[-5 5]` (default) | two-element vector | six-element vector

Plotting interval for `x`, `y`, and `z`, specified in one of these forms:

- Two-element vector of form `[min max]` — Use the same plotting interval of `[min max]` for `x`, `y`, and `z`.
- Six-element vector of form `[xmin xmax ymin ymax zmin zmax]` — Use different plotting intervals for `x`, `y`, and `z`. Plot over the interval `[xmin xmax]` for `x`, over `[ymin ymax]` for `y`, and over `[zmin zmax]` for `z`.

Example: `fimplicit3(f,[-2 3 -4 5 -3 3])`

LineSpec — Line style, marker symbol, and color

character vector | string

Line style, marker symbol, and color, specified as a character vector or string containing symbols. The symbols can appear in any order. You do not need to specify all three characteristics (line style, marker symbol, and color). For example, if you omit the line style and specify the marker, then the plot shows only the markers and no line.

Example: `fimplicit3(f,'--or')` plots a red, dashed line with circle markers.

Line Style	Description
-	Solid line (default)
- -	Dashed line

Line Style	Description
`:`	Dotted line
`-.`	Dash-dot line

Marker	Description
`o`	Circle
`+`	Plus sign
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Pentagram
`h`	Hexagram

Color	Description
`y`	yellow
`m`	magenta
`c`	cyan
`r`	red
`g`	green
`b`	blue
`w`	white
`k`	black

ax — Axes object
axes object

Axes object. If you do not specify the axes, then `fimplicit3` uses the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `fimplicit3(f,'MeshDensity',50,'FaceAlpha',0.5)` specifies the number of evaluation points and a transparency value.

The `ImplicitFunctionSurface` properties listed here are only a subset. For a complete list, see ImplicitFunctionSurface.

`MeshDensity` — Number of evaluation points per direction
35 (default) | scalar

Number of evaluation points per direction, specified as a scalar.

`FaceAlpha` — Face transparency
1 (default) | scalar in range `[0 1]`

Face transparency, specified as a scalar in the range `[0,1]`. Use uniform transparency across all of the faces. A value of `1` is fully opaque and `0` is completely transparent. Values between `0` and `1` are semitransparent.

`FaceColor` — Face color
`'interp'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Face color, specified as `'interp'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'interp'` interpolates the colors based on the `ZData` values.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to

F. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`EdgeColor` — Line color

`[0 0 0]` (default) | `'interp'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as `'interp'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default RGB triplet value of `[0 0 0]` corresponds to black. The `'interp'` value colors the edges based on the `ZData` values.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

Tips

- Use element-wise operators for the best performance and to avoid a warning message. For example, use `x.*y` instead of `x*y`. For more information, see “Array vs. Matrix Operations”.
- When you zoom in on the chart, `fimplicit3` recalculates the data, which can reveal hidden details.

See Also

Functions
`fcontour` | `fimplicit` | `fplot` | `fplot3` | `fsurf`

Properties
ImplicitFunctionSurface

Introduced in R2016b

find

Find indices and values of nonzero elements

Syntax

```
k = find(X)
k = find(X,n)
k = find(X,n,direction)

[row,col] = find( ___ )
[row,col,v] = find( ___ )
```

Description

`k = find(X)` returns a vector containing the linear indices on page 1-4375 of each nonzero element in array `X`.

- If `X` is a vector, then `find` returns a vector with the same orientation as `X`.
- If `X` is a multidimensional array, then `find` returns a column vector of the linear indices of the result.
- If `X` contains no nonzero elements or is empty, then `find` returns an empty array.

`k = find(X,n)` returns the first `n` indices corresponding to the nonzero elements in `X`.

`k = find(X,n,direction)`, where `direction` is `'last'`, finds the last `n` indices corresponding to nonzero elements in `X`. The default for `direction` is `'first'`, which finds the first `n` indices corresponding to nonzero elements.

`[row,col] = find( ___ )` returns the row and column subscripts of each nonzero element in array `X` using any of the input arguments in previous syntaxes.

`[row,col,v] = find( ___ )` also returns vector `v`, which contains the nonzero elements of `X`.

Examples

Zero and Nonzero Elements in Matrix

Find the nonzero elements in a 3-by-3 matrix.

```
X = [1 0 2; 0 1 1; 0 0 4]
```

```
X = 3×3

     1     0     2
     0     1     1
     0     0     4
```

```
k = find(X)
```

```
k = 5×1

     1
     5
     7
     8
     9
```

Use the logical `not` operator on `X` to locate the zeros.

```
k2 = find(~X)
```

```
k2 = 4×1

     2
     3
     4
     6
```

Elements Satisfying a Condition

Find the first five elements that are less than 10 in a 4-by-4 magic square matrix.

```
X = magic(4)
```

```
X = 4×4

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

```
k = find(X<10,5)
```

```
k = 5×1

     2
     3
     4
     5
     7
```

View the corresponding elements of X.

```
X(k)
```

```
ans = 5×1

     5
     9
     4
     2
     7
```

Elements Equal to Specific Values

To find a specific integer value, use the == operator. For instance, find the element equal to 13 in a 1-by-10 vector of odd integers.

```
x = 1:2:20
```

```
x = 1×10
```

```
     1     3     5     7     9    11    13    15    17    19
```

```
k = find(x==13)
```

```
k = 7
```

To find a noninteger value, use a tolerance value based on your data. Otherwise, the result is sometimes an empty matrix due to floating-point roundoff error.

```
y = 0:0.1:1
```

```
y = 1×11

         0    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000    0.7000    0.8
```

```
k = find(y==0.3)
```

```
k =

  1x0 empty double row vector
```

```
k = find(abs(y-0.3) < 0.001)
```

```
k = 4
```

Last Several Nonzero Elements

Create a 6-by-6 magic square matrix with all of the odd-indexed elements equal to zero.

```
X = magic(6);
X(1:2:end) = 0
```

```
X = 6×6

     0     0     0     0     0     0
     3    32     7    21    23    25
     0     0     0     0     0     0
     8    28    33    17    10    15
     0     0     0     0     0     0
     4    36    29    13    18    11
```

Locate the *last* four nonzeros.

```
k = find(X,4,'last')
```

```
k = 4×1

    30
    32
    34
    36
```

Elements Satisfying Multiple Conditions

Find the first three elements in a 4-by-4 matrix that are greater than `0` and less than `10`. Specify two outputs to return the row and column subscripts to the elements.

```
X = [18 3 1 11; 8 10 11 3; 9 14 6 1; 4 3 15 21]
```

```
X = 4×4

    18     3     1    11
     8    10    11     3
     9    14     6     1
     4     3    15    21
```

```
[row,col] = find(X>0 & X<10,3)
```

```
row = 3×1

     2
     3
     4


col = 3×1

     1
     1
     1
```

The first instance is `X(2,1)`, which is `8`.

Subscripts and Values for Nonzero Elements

Find the nonzero elements in a 3-by-3 matrix. Specify three outputs to return the row subscripts, column subscripts, and element values.

```
X = [3 2 0; -5 0 7; 0 0 1]
```

```
X = 3×3

     3     2     0
    -5     0     7
     0     0     1
```

```
[row,col,v] = find(X)
```

```
row = 5×1

     1
     2
     1
     2
     3


col = 5×1

     1
     1
     2
     3
     3


v = 5×1

     3
    -5
     2
     7
```

```
     1
```

Subscripts of Multidimensional Array

Find the nonzero elements in a 4-by-2-by-3 array. Specify two outputs, `row` and `col`, to return the row and column subscripts of the nonzero elements. When the input is a multidimensional array (`N > 2`), `find` returns `col` as a linear index over the `N-1` trailing dimensions of `X`.

```
X = zeros(4,2,3);
X([1 12 19 21]) = 1
```

```
X =
X(:,:,1) =

     1     0
     0     0
     0     0
     0     0


X(:,:,2) =

     0     0
     0     0
     0     0
     1     0


X(:,:,3) =

     0     1
     0     0
     1     0
     0     0
```

```
[row,col] = find(X)
```

```
row = 4×1
```

```
     1
     4
     3
     1


col = 4×1

     1
     3
     5
     6
```

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. If X is an empty array or has no nonzero elements, then k is an empty array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

n — Number of nonzeros to find
positive integer scalar

Number of nonzeros to find, specified as a positive integer scalar. By default, `find(X,n)` looks for the first n nonzero elements in X.

`direction` — Search direction
`'first'` (default) | `'last'`

Search direction, specified as the string `'first'` or `'last'`. Look for the *last* n nonzero elements in X using `find(X,n,'last')`.

Output Arguments

k — Indices to nonzero elements
vector

Indices to nonzero elements, returned as a vector. If X is a row vector, then k is also a row vector. Otherwise, k is a column vector. k is an empty array when X is an empty array or has no nonzero elements.

You can return the nonzero values in X using `X(k)`.

`row` — Row subscripts
vector

Row subscripts, returned as a vector. Together, `row` and `col` specify the `X(row,col)` subscripts corresponding to the nonzero elements in X.

`col` — Column subscripts
vector

Column subscripts, returned as a vector. Together, `row` and `col` specify the `X(row,col)` subscripts corresponding to the nonzero elements in X.

If X is a multidimensional array with `N > 2`, then `col` is a linear index over the `N-1` trailing dimensions of X. This preserves the relation `X(row(i),col(i)) == v(i)`.

v — Nonzero elements of X
vector

Nonzero elements of X, returned as a vector.

More About

Linear Indices

A linear index allows use of a single subscript to index into an array, such as `A(k)`. MATLAB treats the array as a single column vector with each column appended to the bottom of the previous column. Thus, linear indexing numbers the elements in the columns from top to bottom, left to right.

For example, consider a 3-by-3 matrix. You can reference the `A(2,2)` element with `A(5)`, and the `A(2,3)` element with `A(8)`. The linear index changes depending on the size of the array; `A(5)` returns a differently located element for a 3-by-3 matrix than it does for a 4-by-4 matrix.

The `sub2ind` and `ind2sub` functions are useful in converting between subscripts and linear indices.

Tips

- To find array elements that meet a condition, use `find` in conjunction with a relational expression. For example, `find(X<5)` returns the linear indices to the elements in `X` that are less than `5`.
- To directly find the elements in `X` that satisfy the condition `X<5`, use `X(X<5)`. Avoid function calls like `X(find(X<5))`, which unnecessarily use `find` on a logical matrix.
- When you execute `find` with a relational operation like `X>1`, it is important to remember that the result of the relational operation is a logical matrix of ones and zeros. For example, the command `[row,col,v] = find(X>1)` returns a column vector of logical `1` (`true`) values for `v`.
- The row and column subscripts, `row` and `col`, are related to the linear indices in `k` by `k = sub2ind(size(X),row,col)`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- `X` must be a tall column vector.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If a variable-size input becomes a row vector at run time, then code generation ends with an error. This limitation does not apply when the input is scalar or is a variable-length row vector.
- For variable-size inputs, the shape of empty outputs (0-by-0, 0-by-1, or 1-by-0) depends on the upper bounds of the size of the input. When the input array is a scalar or [] at run time, the output might not match MATLAB. If the input is a variable-length row vector, then the size of an empty output is 1-by-0. Otherwise, the size is 0-by-1.
- The generated code always returns a variable-length vector. Even when you provide the output vector `k`, the output is not fixed-size because the output can contain fewer than `k` elements. For example, `find(x,1)` returns a variable-length vector with one or zero elements.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`Logical Operators: Short-Circuit` | `ind2sub` | `ismember` | `nonzeros` | `strfind` | `sub2ind`

Topics

"Find Array Elements That Meet a Condition"
"Array Indexing"
"Relational Operations"
"Sparse Matrices"

Introduced before R2006a

findall

Find all graphics objects

Syntax

```
object_handles = findall(handle_list)
object_handles = findall(handle_list,'property','value',...)
```

Description

`object_handles = findall(handle_list)` returns the handles, including hidden handles, of all objects in the hierarchy under the objects identified in `handle_list`.

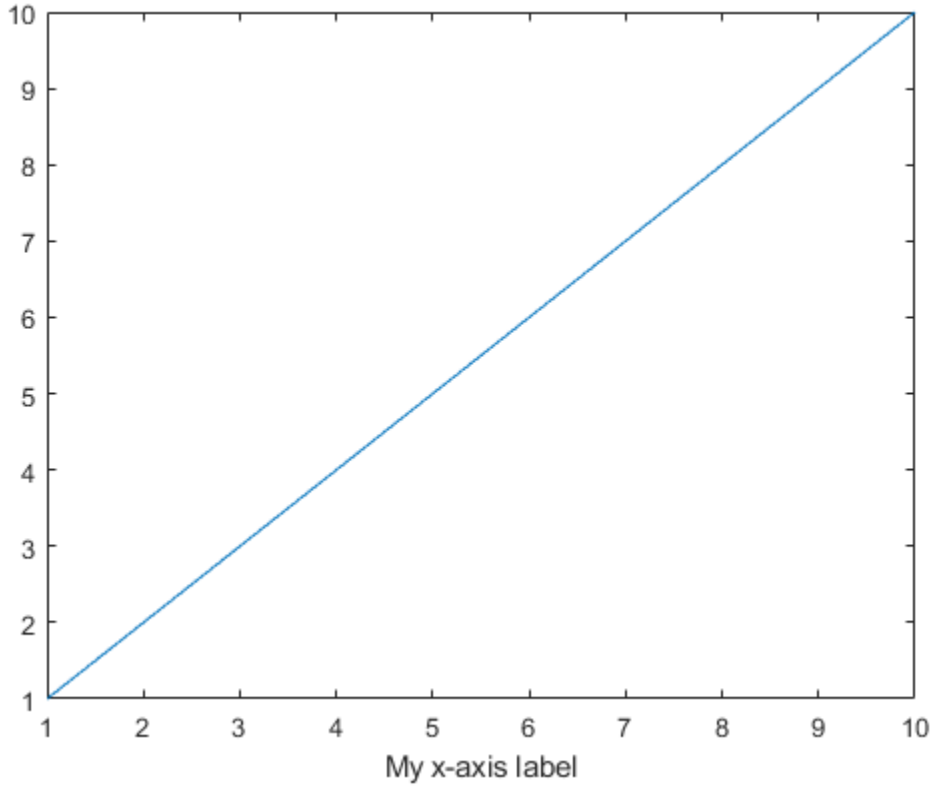
`object_handles = findall(handle_list,'property','value',...)` returns the handles of all objects in the hierarchy under the objects identified in `handle_list` that have the specified properties set to the specified values.

Examples

Find Object with Hidden Handle

Use `findall` to find the text object for the *x*-axis label. Unlike `findobj`, the `findall` function finds objects even if the `HandleVisibility` property is set to `'off'`.

```
plot(1:10)
txt = xlabel('My x-axis label');
```




```
txt.HandleVisibility
```

```
ans =
'off'
```

```
a = findall(gcf,'Type','text')
```

```
a =
  3x1 Text array:

  Text    (My x-axis label)
  Text
  Text
```

```
d = findobj(gcf,'Type','text')
```

```
d = 
  0x0 empty GraphicsPlaceholder array.
```

Tips

`findall` is similar to `findobj`, except that it finds objects even if their `HandleVisibility` is set to `off`.

See Also

`allchild` | `findobj`

Introduced before R2006a

findfigs

Find visible offscreen figures

Syntax

```
findfigs
```

Description

`findfigs` finds all visible figure windows whose display area is off the screen and positions them on the screen.

A window appears to the MATLAB software to be offscreen when its display area (the area not covered by the window's title bar, menu bar, and toolbar) does not appear on the screen.

This function is useful when you are bringing an application from a larger monitor to a smaller one (or one with lower resolution). Windows visible on the larger monitor may appear offscreen on a smaller monitor. Using `findfigs` ensures that all windows appear on the screen.

See Also

Introduced before R2006a

findgroups

Find groups and return group numbers

Syntax

```
G = findgroups(A)
G = findgroups(A1,...,AN)

[G,ID] = findgroups(A)
[G,ID1,...,IDN] = findgroups(A1,...,AN)

G = findgroups(T)
[G,TID] = findgroups(T)
```

Description

`G = findgroups(A)` returns `G`, a vector of group numbers created from the grouping variable `A`. The output argument `G` contains integer values from 1 to `N`, indicating `N` distinct groups for the `N` unique values in `A`. For example, if `A` is `{'b','a','a','b'}`, then `findgroups` returns `G` as `[2 1 1 2]`. You can use `G` to split groups of data out of other variables. Use `G` as an input argument to `splitapply` in the “Split-Apply-Combine Workflow” on page 1-4391.

`findgroups` treats empty character vectors and `NaN`, `NaT`, and undefined categorical values in `A` as missing values and returns `NaN` as the corresponding elements of `G`.

`G = findgroups(A1,...,AN)` creates group numbers from `A1,...,AN`. The `findgroups` function defines groups as the unique combinations of values across `A1,...,AN`. For example, if `A1` is `{'a','a','b','b'}` and `A2` is `[0 1 0 0]`, then `findgroups(A1,A2)` returns `G` as `[1 2 3 3]`, because the combination `'b' 0` occurs twice.

`[G,ID] = findgroups(A)` also returns the unique values for each group in `ID`. For example, if `A` is `{'b','a','a','b'}`, then `findgroups` returns `G` as `[2 1 1 2]` and `ID` as `{'a','b'}`. The arguments `A` and `ID` are the same data type, but need not be the same size.

`[G,ID1,...,IDN] = findgroups(A1,...,AN)` also returns the unique values for each group across `ID1,...,IDN`. The values across `ID1,...,IDN` define the groups. For example, if `A1` is `{'a','a','b','b'}` and `A2` is `[0 1 0 0]`, then `findgroups(A1,A2)` returns `G` as `[1 2 3 3]`, and `ID1` and `ID2` as `{'a','a','b'}` and `[0 1 0]`.

`G = findgroups(T)` returns `G`, a vector of group numbers created from the variables in table `T`. The `findgroups` function treats all the variables in `T` as grouping variables.

`[G,TID] = findgroups(T)` also returns `TID`, a table that contains the unique values for each group. `TID` contains the unique combinations of values across the variables of `T`. The variables in `T` and `TID` have the same names, but the tables need not have the same number of rows.

Examples

Use Group Numbers to Split Data

Use group numbers to split patient height measurements into groups by gender. Then calculate the mean height for each group.

Load patient heights and genders from the data file `patients.mat`.

```
load patients
whos Gender Height
```

```
  Name          Size            Bytes  Class     Attributes

  Gender      100x1             12212  cell
  Height      100x1               800  double
```

Specify groups by gender with `findgroups`.

```
G = findgroups(Gender);
```

Compare the first five elements of `Gender` and `G`. Where `Gender` contains `'Female'`, `G` contains `1`. Where `Gender` contains `'Male'`, `G` contains `2`.

```
Gender(1:5)
```

```
ans = 5x1 cell array
    {'Male'  }
```

```
    {'Male'  }
    {'Female'}
    {'Female'}
    {'Female'}
```

```
G(1:5)
```

```
ans = 5×1

     2
     2
     1
     1
     1
```

Split the `Height` variable into two groups of heights using `G`. Apply the `mean` function. The groups contain the mean heights of female and male patients, respectively.

```
splitapply(@mean,Height,G)
```

```
ans = 2×1

   65.1509
   69.2340
```

Use Two Grouping Variables to Split Data

Calculate mean blood pressures for groups of patients from measurements grouped by gender and status as a smoker.

Load blood pressure readings, gender, and smoking data for patients from the data file `patients.mat`.

```
load patients
whos Systolic Diastolic Gender Smoker
```

```
  Name             Size            Bytes  Class      Attributes

  Diastolic      100x1               800  double
```

```
  Gender         100x1             12212  cell
  Smoker         100x1               100  logical
  Systolic       100x1               800  double
```

Specify groups using gender and smoking information about the patients. `G` contains integers from one to four because there are four possible combinations of values from `Smoker` and `Gender`.

```
G = findgroups(Smoker,Gender);
G(1:10)
```

```
ans = 10×1

     4
     2
     1
     1
     1
     1
     3
     2
     2
     1
```

Calculate the mean blood pressure for each group.

```
meanSystolic = splitapply(@mean,Systolic,G);
meanDiastolic = splitapply(@mean,Diastolic,G);
mBP = [meanSystolic,meanDiastolic]
```

```
mBP = 4×2

  119.4250   79.0500
  119.3462   79.8846
  129.0000   89.2308
  129.5714   90.3333
```

Use Unique Group Values in Output Table

Calculate the median heights for groups of patients, and display the results in a table. To define the groups of patients, use the additional output argument from `findgroups`.

Load patient heights and genders from the data file `patients.mat`.

```
load patients
whos Gender Height
```

```
  Name          Size            Bytes  Class     Attributes

  Gender      100x1             12212  cell
  Height      100x1               800  double
```

Specify groups by gender with `findgroups`. The values in the output argument `gender` define the groups that `findgroups` finds in the grouping variable.

```
[G,gender] = findgroups(Gender);
```

Calculate the median heights. Create a table that contains the median heights.

```
medianHeight = splitapply(@median,Height,G);
T = table(gender,medianHeight)
```

```
T=2×2 table
      gender      medianHeight
    __________    ____________

    {'Female'}         65
    {'Male'  }         69
```

Use Unique Values from Two Grouping Variables

Calculate mean blood pressures for groups of patients, and display the results in a table. To define the groups of patients, use the additional output arguments from `findgroups`.

Load blood pressure readings, gender, and smoking data for 100 patients from the data file `patients.mat`.

```
load patients
whos Systolic Diastolic Gender Smoker
```

```
  Name             Size            Bytes  Class      Attributes

  Diastolic      100x1               800  double
```

```
  Gender         100x1             12212  cell
  Smoker         100x1               100  logical
  Systolic       100x1               800  double
```

Specify groups using gender and smoking information about the patients. Calculate mean blood pressure for each group. The values across the output arguments `gender` and `smoker` define the groups that `findgroups` finds in the grouping variables.

```
[G,gender,smoker] = findgroups(Gender,Smoker);
meanSystolic = splitapply(@mean,Systolic,G);
meanDiastolic = splitapply(@mean,Diastolic,G);
```

Create a table with the mean blood pressure for each group of patients.

```
T = table(gender,smoker,meanSystolic,meanDiastolic)
```

```
T=4×4 table
      gender      smoker    meanSystolic    meanDiastolic
    __________    ______    ____________    _____________

    {'Female'}    false        119.42           79.05
    {'Female'}    true            129          89.231
    {'Male'  }    false        119.35          79.885
    {'Male'  }    true         129.57          90.333
```

Group by Table Variables

Calculate mean blood pressures for patients using grouping variables that are in a table.

Load gender and smoking data for 100 patients into a table.

```
load patients
T = table(Gender,Smoker);
T(1:5,:)
```

```
ans=5×2 table
      Gender      Smoker
    __________    ______

    {'Male'  }    true
    {'Male'  }    false
```

```
    {'Female'}    false
    {'Female'}    false
    {'Female'}    false
```

Specify groups of patients using the `Gender` and `Smoker` variables in `T`.

```
G = findgroups(T);
```

Calculate mean blood pressures from the data variables `Systolic` and `Diastolic`.

```
meanSystolic = splitapply(@mean,Systolic,G);
meanDiastolic = splitapply(@mean,Diastolic,G);
mBP = [meanSystolic,meanDiastolic]
```

```
mBP = 4×2

  119.4250   79.0500
  129.0000   89.2308
  119.3462   79.8846
  129.5714   90.3333
```

Group from Table and Create Output Table

Create a table of mean blood pressures for patients grouped by gender and status as a smoker or nonsmoker.

Load gender and smoking data for patients into a table.

```
load patients
T = table(Gender,Smoker);
```

Specify groups of patients using the `Gender` and `Smoker` variables in `T`. The output table `TID` identifies the groups.

```
[G,TID] = findgroups(T);
TID
```

```
TID=4×2 table
      Gender      Smoker
    __________    ______
```

```
    {'Female'}    false
    {'Female'}    true
    {'Male'  }    false
    {'Male'  }    true
```

Calculate mean blood pressures from the data variables `Systolic` and `Diastolic`. Append mean blood pressures to `TID`.

```
TID.meanSystolic = splitapply(@mean,Systolic,G);
TID.meanDiastolic = splitapply(@mean,Diastolic,G)
```

```
TID=4×4 table
      Gender      Smoker    meanSystolic    meanDiastolic
    __________    ______    ____________    _____________

    {'Female'}    false        119.42           79.05
    {'Female'}    true            129          89.231
    {'Male'  }    false        119.35          79.885
    {'Male'  }    true         129.57          90.333
```

Input Arguments

A — Grouping variable
vector | cell array of character vectors | string array

Grouping variable, specified as a vector, a cell array of character vectors, or a string array. The unique values in `A` identify groups.

If `A` is a vector, then it can be numeric or of data type `categorical`, `calendarDuration`, `datetime`, `duration`, `logical`, or `string`.

T — Grouping variables
table

Grouping variables, specified as a table. `findgroups` treats each table variable as a separate grouping variable. The variables can be numeric or of data type `categorical`, `calendarDuration`, `datetime`, `duration`, `logical`, or `string`.

Output Arguments

`G` — Group numbers
vector of positive integers

Group numbers, returned as a vector of positive integers. For `N` groups identified in the grouping variables, every integer between 1 and `N` specifies a group. `G` contains `NaN` where any grouping variable contains an empty character vector or a `NaN`, `NaT`, or undefined categorical value.

- If the grouping variables are vectors, then `G` and the grouping variables all are the same size.
- If the grouping variables are in a table, the length of `G` is equal to the number of rows of the table.

`ID` — Values that identify each group
vector of unique values | cell array of character vectors

Values that identify each group, returned as a vector or cell array of character vectors. The values of `ID` are the sorted unique values of `A`.

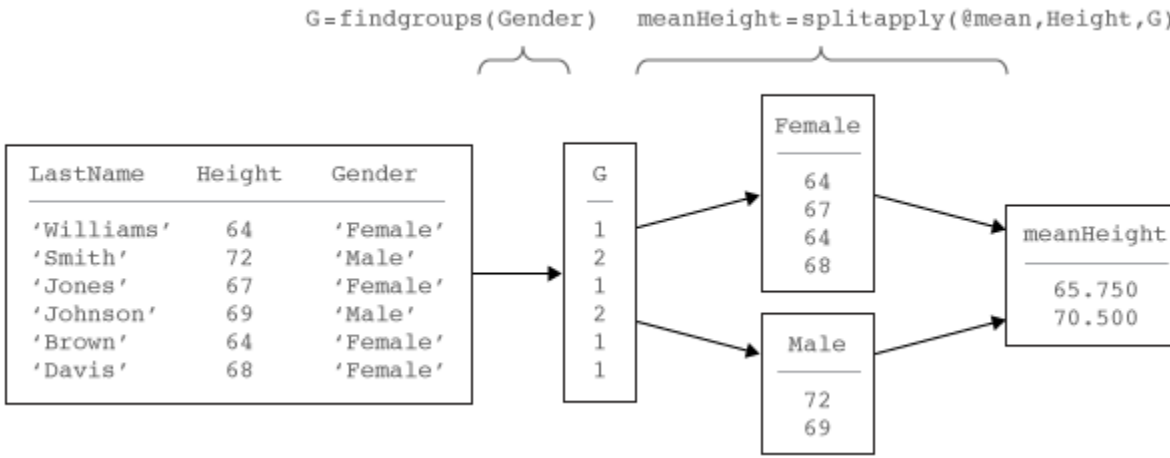
`TID` — Table of unique values that identify each group
table

The unique values that identify each group, returned as a table. The variables of `TID` have the sorted unique values from the corresponding variables of `T`. However, `TID` and `T` need not have the same number of rows.

More About

Split-Apply-Combine Workflow

The *Split-Apply-Combine* workflow is common in data analysis. In this workflow, the analyst splits the data into groups, applies a function to each group, and combines the results. The diagram shows a typical example of the workflow and the parts of the workflow implemented by `findgroups` and `splitapply`.



Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

- Tall tables are not supported.
- The order of the group numbers in `G` might be different compared to in-memory `findgroups` calculations.

For more information, see “Tall Arrays for Out-of-Memory Data”.

See Also

`accumarray` | `arrayfun` | `discretize` | `groupsummary` | `histcounts` | `ismember` | `rowfun` | `splitapply` | `unique` | `varfun`

Topics

“Split Table Data Variables and Apply Functions”

“Split Data into Groups and Calculate Statistics”
“Calculations on Tables”
“Grouping Variables To Split Data”

Introduced in R2015b

findobj

Locate graphics objects with specific properties

Syntax

```
findobj
h = findobj
h = findobj('PropertyName',PropertyValue,...)
h = findobj('PropertyName',PropertyValue,'-logicaloperator',
'PropertyName',PropertyValue,...)
h = findobj('-regexp','PropertyName','expression',...)
h = findobj('-property','PropertyName')
h = findobj(objhandles,...)
h = findobj(objhandles,'-depth',d,...)
h = findobj(objhandles,'flat','PropertyName',PropertyValue,...)
```

Description

`findobj` returns handles of the root object and all its descendants without assigning the result to a variable.

`h = findobj` returns handles of the root object and all its descendants.

`h = findobj('PropertyName',PropertyValue,...)` returns handles of all graphics objects having the property *`PropertyName`*, set to the value `PropertyValue`. You can specify more than one property/value pair, in which case, `findobj` returns only those objects having all specified values.

`h = findobj('PropertyName',PropertyValue,'-logicaloperator', 'PropertyName',PropertyValue,...)` applies the logical operator to the property value matching. Possible values for *`-logicaloperator`* are:

- `-and`
- `-or`
- `-xor`

- `-not`

For more information on logical operators, see “Logical Operations”.

`h = findobj('-regexp','PropertyName','expression',...)` matches objects by evaluating the specified regular expression using the value of the specified property. Objects with property values satisfying the regular expression are returned.

`h = findobj('-property','PropertyName')` finds all objects having the specified property.

`h = findobj(objhandles,...)` restricts the search to objects listed in `objhandles` and their descendants.

`h = findobj(objhandles,'-depth',d,...)` specifies the depth of the search. The depth argument `d` controls how many levels under the handles in `objhandles` MATLAB traverses. Specify `d` as `inf` to get the default behavior of all levels. Specify `d` as `0` to get the same behavior as using the `flat` argument.

`h = findobj(objhandles,'flat','PropertyName',PropertyValue,...)` restricts the search to those objects listed in `objhandles` and does not search descendants.

`findobj` returns an error if a handle refers to a nonexistent graphics object.

`findobj` correctly matches any legal property value. For example,

```
findobj('Color','r')
```

finds all objects having a `Color` property set to `red`, `r`, or `[1 0 0]`.

When a graphics object is a descendant of more than one object identified in `objhandles`, MATLAB searches the object each time `findobj` encounters its handle. Therefore, implicit references to a graphics object can result in multiple returns of its handle.

Note `findobj` does not return graphics objects that have the `HandleVisibility` property set to `'off'`. For more information, see the `HandleVisibility` property description.

Examples

Find all line objects in the current axes:

```
h = findobj(gca,'Type','line')
```

Find all objects having a `Label` set to `'foo'` and a `String` property set to `'bar'`:

```
h = findobj('Label','foo','-and','String','bar');
```

Find all objects whose `String` property is not `'foo'` and is not `'bar'`:

```
h = findobj('-not','String','foo','-not','String','bar');
```

Find all objects having a `String` property set to `'foo'` and a `Tag` property set to `'button one'` and whose `Color` property is not `'red'` or `'blue'`:

```
h = findobj('String','foo','-and','Tag','button one',...
    '-and','-not',{'Color','red','-or','Color','blue'})
```

Find all objects for which you have assigned a value to the `Tag` property (that is, the value is not the empty character vector `''`):

```
h = findobj('-regexp','Tag','[^'']')
```

Find all children of the current figure that have their `BackgroundColor` property set to a certain shade of gray (`[.7 .7 .7]`). This statement also searches the current figure for the matching property value pair.

```
h = findobj(gcf,'-depth',1,'BackgroundColor',[.7 .7 .7])
```

See Also

`copyobj` | `findall` | `findobj` | `gca` | `gcbo` | `gcf` | `gco` | `get` | `regexp` | `set`

Topics

"Find Objects"

Introduced before R2006a

findstr

(Not recommended) Find string within another, longer string

Note `findstr` is not recommended. Use `contains` or `strfind` instead.

Syntax

```
k = findstr(str1, str2)
```

Description

`k = findstr(str1, str2)` searches the longer of the two input arguments for any occurrences of the shorter argument, returning the starting index of each such occurrence in the double array `k`. If no occurrences are found, then `findstr` returns the empty array, `[]`. The input arguments `str1` and `str2` can be character vectors or string scalars.

The search performed by `findstr` is case sensitive. Any leading and trailing blanks in either input argument are explicitly included in the comparison.

Unlike the `contains` or `strfind` functions, the order of the input arguments to `findstr` is not important. This can be useful if you are not certain which of the two input arguments is the longer one.

Examples

```
s = 'Find the starting indices of the shorter character vector.';

findstr(s, 'the')
ans =
     6    30

findstr('the', s)
ans =
     6    30
```

See Also

`contains` | `regexp` | `regexpi` | `regexprep` | `strcmp` | `strcmpi` | `strfind` | `strncmp` | `strncmpi` | `strtok`

Introduced before R2006a

finish

User-defined termination script for MATLAB

Syntax

```
finish
```

Description

`finish` executes user-specified commands when exiting. MATLAB calls the `finish.m` script, if defined, whenever you do one of the following.

- Click the close button on the MATLAB desktop on Microsoft Windows platforms or the equivalent on UNIX platforms.
- Type `quit` or `exit` at the command prompt.

If an error occurs in the `finish` script, then `quit` is canceled so that you can correct your `finish.m` file without losing your workspace.

Examples

Save Workspace to MAT-File

Use the MATLAB sample file `finishsav.m` to save the workspace to a MAT-file when MATLAB quits.

To view the contents or edit the file, open the `finishsav.m` file in the editor.

```
edit(fullfile(matlabroot,'toolbox','local','finishsav.m'));
```

Copy the file to the search path and rename it to `finish.m`.

```
copyfile(fullfile(matlabroot,'toolbox','local','finishsav.m'),fullfile(userpath,'finish.m'))
```

Display Quit Dialog Box

Use the MATLAB sample file `finishdlg.m` to display a custom dialog box allowing you to cancel when MATLAB quits.

To view the contents or edit the file, open the `finishdlg.m` file in the editor.

```
edit(fullfile(matlabroot,'toolbox','local','finishsav.m'));
```

Copy the file to the search path and rename it to `finish.m`.

```
copyfile(fullfile(matlabroot,'toolbox','local','finishdlg.m'),fullfile(userpath,'finish.m'))
```

To display this confirmation dialog box, set the **Confirm before exiting MATLAB** option in “General Preferences”.

Tips

- When using graphics objects in `finish.m`, call `uiwait`, `waitfor`, or `drawnow` so that figures are visible.

See Also

`exit` | `quit` | `startup`

Topics

“Exit MATLAB”

Introduced before R2006a

fitsdisp

Display FITS metadata

Syntax

```
fitsdisp(filename)
fitsdisp(filename,Name,Value)
```

Description

`fitsdisp(filename)` displays metadata for all the Header/Data Units (HDUs) found in the FITS file specified by `filename`.

`fitsdisp(filename,Name,Value)` displays metadata for all the Header/Data Units (HDUs) found in the FITS file with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`filename`

Name of an existing FITS file specified as a character vector or string scalar.

Default:

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`Index`

Positive scalar value or vector specifying the HDUs.

Default:

`Mode`

Specified as one of the following:

- `'standard'` – Display standard keywords
- `'min'` – Display only HDU types and sizes
- `'full'` – Display all HDU keywords

Default: `'standard'`

Examples

Display metadata in the 2nd HDU in the FITS file.

```
fitsdisp('tst0012.fits','Index',2);
```

Display the metadata in the 1st, 3rd, and 5th HDUs in a file.

```
fitsdisp('tst0012.fits','Index',[1 3 5]);
```

Display all metadata in the 5th HDU in a file

```
fitsdisp('tst0012.fits','Index',5,'Mode','full');
```

References

For copyright information, see the `cfitsiocopyright.txt` file.

See Also

`fitsinfo` | `fitsread` | `fitswrite`

Introduced in R2012a

fitsinfo

Information about FITS file

Syntax

```
info = fitsinfo(filename)
```

Description

`info = fitsinfo(filename)` returns the structure, `info`, with fields that contain information about the contents of a Flexible Image Transport System (FITS) file. `filename` is a character vector or string scalar that specifies the name of the FITS file.

The `info` structure contains the following fields, listed in the order they appear in the structure. In addition, the info structure can also contain information about any number of optional file components, called *extensions* in FITS terminology. For more information, see “FITS File Extensions” on page 1-4404.

Field Name	Description	Return Type
`Filename`	Name of the file	Character vector
`FileModDate`	File modification date	Character vector
`FileSize`	Size of the file in bytes	`double`
`Contents`	List of extensions in the file in the order that they occur	Cell array of character vectors
`PrimaryData`	Information about the primary data in the FITS file	Structure array

PrimaryData

The `PrimaryData` field is a structure that describes the primary data in the file. The following table lists the fields in the order they appear in the structure.

Field Name	Description	Return Type
`DataType`	Precision of the data	Character vector
`Size`	Size of each dimension. The number of rows correspond to the value of the NAXIS2 keyword, while the number of columns correspond to the value of the NAXIS1 keyword. Any further dimensions correspond to NAXIS3, NAXIS4, and so on.	`double` array
`DataSize`	Size of the primary data in bytes	`double`
`MissingDataValue`	Value used to represent undefined data	`double`
`Intercept`	Value, used with `Slope`, to calculate actual pixel values from the array pixel values, using the equation: `actual_value = Slope*array_value + Intercept`	`double`
`Slope`	Value, used with `Intercept`, to calculate actual pixel values from the array pixel values, using the equation: `actual_value = Slope*array_value + Intercept`	`double`
`Offset`	Number of bytes from beginning of the file to the location of the first data value	`double`
`Keywords`	A number-of-keywords-by-3 cell array containing keywords, values, and comments of the header in each column	Cell array of character vectors

FITS File Extensions

A FITS file can also include optional extensions. If the file contains any of these extensions, the `info` structure can contain these additional fields.

- AsciiTable on page 1-4405 — Numeric information in tabular format, stored as ASCII characters
- BinaryTable on page 1-4406 — Numeric information in tabular format, stored in binary representation

- Image on page 1-4407 — A multidimensional array of pixels
- Unknown on page 1-4408 — Nonstandard extension

AsciiTable Extension

The `AsciiTable` structure contains the following fields, listed in the order they appear in the structure.

Field Name	Description	Return Type
`Rows`	Number of rows in the table	`double`
`RowSize`	Number of characters in each row	`double`
`NFields`	Number of fields in each row	`double` array
`FieldFormat`	A 1-by-`NFields` cell containing formats in which each field is encoded. The formats are FORTRAN-77 format codes.	Cell array of character vectors
`FieldPrecision`	A 1-by-`NFields` cell containing precision of the data in each field	Cell array of character vectors
`FieldWidth`	A 1-by-`NFields` array containing the number of characters in each field	`double` array
`FieldPos`	A 1-by-`NFields` array of numbers representing the starting column for each field	`double` array
`DataSize`	Size of the data in the table in bytes	`double`
`MissingDataValue`	A 1-by-`NFields` array of numbers used to represent undefined data in each field	Cell array of character vectors
`Intercept`	A 1-by-`NFields` array of numbers used along with `Slope` to calculate actual data values from the array data values using the equation: `actual_value = Slope*array_value+Intercept`	`double` array
`Slope`	A 1-by-`NFields` array of numbers used with `Intercept` to calculate true data values from the array data values using the equation: `actual_value = Slope*array_value+Intercept`	`double` array

Field Name	Description	Return Type
`Offset`	Number of bytes from beginning of the file to the location of the first data value in the table	`double`
`Keywords`	A number-of-keywords-by-3 cell array containing all the Keywords, Values and Comments in the ASCII table header	Cell array of character vectors

BinaryTable Extension

The `BinaryTable` structure contains the following fields, listed in the order they appear in the structure.

Field Name	Description	Return Type
`Rows`	Number of rows in the table	`double`
`RowSize`	Number of bytes in each row	`double`
`NFields`	Number of fields in each row	`double`
`FieldFormat`	A 1-by-`NFields` cell array containing the data type of the data in each field. The data type is represented by a FITS binary table format code.	Cell array of character vectors
`FieldPrecision`	A 1-by-`NFields` cell containing precision of the data in each field	Cell array of character vectors
`FieldSize`	A 1-by-`NFields` array, where each element contains the number of values in the Nth field	`double` array
`DataSize`	Size of the data in the Binary Table, in bytes. Includes any data past the main table.	`double`
`MissingDataValue`	An 1-by-`NFields` array of numbers used to represent undefined data in each field	Cell array of `double`

Field Name	Description	Return Type
`Intercept`	A 1-by-`NFields` array of numbers used along with Slope to calculate actual data values from the array data values using the equation: `actual_value = slope*array_value+Intercept`	`double` array
`Slope`	A 1-by-`NFields` array of numbers used with Intercept to calculate true data values from the array data values using the equation: `actual_value = Slope*array_value+Intercept`	`double` array
`Offset`	Number of bytes from beginning of the file to the location of the first data value	`double`
`ExtensionSize`	Size of any data past the main table, in bytes	`double`
`ExtensionOffset`	Number of bytes from the beginning of the file to any data past the main table	`double`
`Keywords`	A number-of-keywords-by-3 cell array containing all the Keywords, values, and comments in the Binary Table header	Cell array of character vectors

Image Extension

The `Image` structure contains the following fields, listed in the order they appear in the structure.

Field Name	Description	Return Type
`DataType`	Precision of the data	Character vector
`Size`	Size of each dimension. The number of rows correspond to the value of the NAXIS2 keyword, while the number of columns correspond to the value of the NAXIS1 keyword. Any further dimensions correspond to NAXIS3, NAXIS4, and so on.	`double` array

Field Name	Description	Return Type
`DataSize`	Size of the data in the Image extension in bytes	`double`
`Offset`	Number of bytes from the beginning of the file to the first data value	`double`
`MissingDataValue`	Value used to represent undefined data	`double`
`Intercept`	Value, used with `Slope`, to calculate actual pixel values from the array pixel values, using the equation: `actual_value = Slope*array_value +Intercept`	`double`
`Slope`	Value, used with `Intercept`, to calculate actual pixel values from the array pixel values, using the equation: `actual_value = Slope*array_value + Intercept`	`double`
`Keywords`	A number-of-keywords-by-3 cell array containing all the Keywords, values, and comments in the Binary Table header	Cell array of character vectors

Unknown Structure

The `Unknown` structure contains the following fields, listed in the order they appear in the structure.

Field Name	Description	Return Type
`DataType`	Precision of the data	Character vector
`Size`	Sizes of each dimension	`double` array
`DataSize`	Size of the data in nonstandard extensions, in bytes	`double`
`Offset`	Number of bytes from beginning of the file to the first data value	`double`
`MissingDataValue`	Representation of undefined data	`double`

Field Name	Description	Return Type
`Intercept`	Value, used with `Slope`, to calculate actual data values from the array data values, using the equation: `actual_value = Slope*array_value+Intercept`	`double`
`Slope`	Value, used with `Intercept`, to calculate actual data values from the array data values, using the equation: `actual_value = Slope*array_value+Intercept`	`double`
`Keywords`	A number-of-keywords-by-3 cell array containing all the Keywords, values, and comments in the Binary Table header	Cell array of character vectors

Examples

Get Information About FITS File

Use `fitsinfo` to obtain information about the FITS file `tst0012.fits`. The returned structure contains `Filename`, `FileModDate`, `FileSize`, `Contents`, and `PrimaryData`. In addition to its primary data, the file also contains an example of the extensions `BinaryTable`, `Unknown`, `Image`, and `AsciiTable`.

```
S = fitsinfo('tst0012.fits');
```

View the `PrimaryData` field.

```
S.PrimaryData
```

```
ans = struct with fields:
            DataType: 'single'
                Size: [109 102]
            DataSize: 44472
    MissingDataValue: []
           Intercept: 0
               Slope: 1
              Offset: 2880
```

```
            Keywords: {25x3 cell}
```

The `PrimaryData` field describes the data in the file. For example, the `Size` field indicates the data is a 109-by-102 matrix.

View the `AsciiTable` field.

```
S.AsciiTable
```

```
ans = struct with fields:
                Rows: 53
             RowSize: 59
             NFields: 8
         FieldFormat: {1x8 cell}
      FieldPrecision: {1x8 cell}
          FieldWidth: [9 6.2000 3 10.4000 20.1500 5 1 4]
            FieldPos: [1 11 18 22 33 54 54 55]
            DataSize: 3127
    MissingDataValue: {'*'  '---.--'  '  *'  []  '*'  '*'  '*'  ''}
           Intercept: [0 0 -70.2000 0 0 0 0 0]
               Slope: [1 1 2.1000 1 1 1 1 1]
              Offset: 103680
            Keywords: {65x3 cell}
```

The `AsciiTable` field describes the `AsciiTable` extension. For example, using the `FieldWidth` and `FieldPos` fields you can determine the length and location of each field within a row.

See Also

`fitsdisp` | `fitsread` | `fitswrite`

Introduced before R2006a

fitsread

Read data from FITS file

Syntax

```
data = fitsread(filename)
data = fitsread(filename,extname)
data = fitsread(filename,extname,index)
data = fitsread(filename,Name,Value)
```

Description

`data = fitsread(filename)` reads the primary data of the Flexible Image Transport System (FITS) file. `filename` is specified as a character vector or string scalar. The function replaces undefined data values with `NaN` and scales numeric data by the slope and intercept values, always returning double precision values.

`data = fitsread(filename,extname)` reads data from the FITS file extension specified by `extname`.

`data = fitsread(filename,extname,index)` reads data from the FITS file extension specified by `extname` . If there is more than one of the specified extensions in the file, `index` specifies the one to read.

`data = fitsread(filename,Name,Value)` reads data from the FITS file with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`filename`

Character vector or string scalar specifying the name of a FITS file.

Default:

extname

The name of a data array or extension in the FITS file, specified as one of the character vectors or string scalars in the table that follows. To determine the contents of a FITS, view the `Contents` field of the structure returned by `fitsinfo`.

Data Arrays or Extensions

Extname	Description
`'primary'`	Read data from the primary data array.
`'asciitable'`	Read data from the ASCII Table extension. The return value, `data`, is a 1-D cell array.
`'binarytable'`	Read data from the Binary Table extension. The return value, `data`, is a 1-D cell array.
`'image'`	Read data from the Image extension.
`'unknown'`	Read data from the Unknown extension.

Default:

index

Numeric value specifying which extension to read, if more than one exists in the file.

Default:

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

info

`info` structure returned by `fitsinfo` specifying the location of data to read.

Note Using the `info` structure returned by `fitsinfo` to specify the location of data in a FITS file can significantly improve performance, especially when reading multiple images from the file.

Default:

`PixelRegion`

Cell array `{rows,cols,...}` specifying the boundaries of a subimage region to read from the file. Each dimension (`rows`, `cols`) is a vector of 1-based indices given either as `START`, `[START STOP]`, or `[START INCREMENT STOP]`. This parameter is valid only for primary or image extensions.

Default:

`raw`

Specifies that `fitsread` should not scale the data read from the file or replace undefined values with `NaN`. Data read from the file is the same class as it is stored in the file.

Default:

`TableColumns`

Vector of 1-based indices specifying the columns to read from the ASCII or Binary table extension. This vector should contain unique and valid indices into the table data specified in increasing order. This parameter is valid only for ASCII or Binary extensions.

Default:

`TableRows`

Vector of 1-based indices specifying the rows to read from the ASCII or Binary table extension. This vector should contain unique and valid indices into the table data specified in increasing order. This parameter is valid only for ASCII or Binary extensions.

Default:

Output Arguments

`data`

Data returned from the FITS file.

Examples

Read primary data from FITS file

```
data = fitsread('tst0012.fits');

Name        Size             Bytes  Class     Attributes

data      109x102            88944  double
```

Inspect available extensions, read 'image' extension using the `extname` option.

```
info = fitsinfo('tst0012.fits');
 % List of contents, includes any extensions if present.
 disp(info.Contents);
 imageData = fitsread('tst0012.fits','image');
```

Subsample the fifth plane of 'image' extension by 2.

```
info        = fitsinfo('tst0012.fits');
 rowend      = info.Image.Size(1);
 colend      = info.Image.Size(2);
 primaryData = fitsread('tst0012.fits','image',...
               'Info', info,...
               'PixelRegion',{[1 2 rowend], [1 2 colend], 5 });
```

Read every other row from an ASCII table.

```
info      = fitsinfo('tst0012.fits');
rowend    = info.AsciiTable.Rows;
tableData = fitsread('tst0012.fits','asciitable',...
                    'Info',info,...
                    'TableRows',[1:2:rowend]);
```

Read all data for the first, second and fifth columns of the Binary table.

```
info      = fitsinfo('tst0012.fits');
rowend    = info.BinaryTable.Rows;
tableData = fitsread('tst0012.fits','binarytable',...
                    'Info',info,...
                    'TableColumns',[1 2 5]);
```

More About

extension

A FITS file contains primary data and can optionally contain any number of optional components, called extensions in FITS terminology.

See Also

`fitsdisp` | `fitsinfo` | `fitswrite`

Topics

"Importing Flexible Image Transport System (FITS) Files"

Introduced before R2006a

fitswrite

Write image to FITS file

Syntax

```
fitswrite(imagedata,filename)
fitswrite(imagedata,filename,Name,Value)
```

Description

`fitswrite(imagedata,filename)` writes `imagedata` to the FITS file specified by `filename`. If `filename` does not exist, `fitswrite` creates the file as a simple FITS file. If `filename` exists, `fitswrite` overwrites the file or appends the image to the end of the file, depending on the value of the `writemode` argument.

`fitswrite(imagedata,filename,Name,Value)` writes `imagedata` to the FITS file specified by `filename` with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`imagedata`

Image array.

Default:

`filename`

Character vector or string scalar specifying the name of an existing FITS file or the name you want to assign to a new FITS file.

Default:

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`WriteMode`

Specified as one of the following:

- `'overwrite'`
- `'append'`

Default: `'overwrite'`

`Compression`

Specified as one of the following:

- `'none'`
- `'gzip'`
- `'rice'`
- `'hcompress'`
- `'plio'`

Default: `'none'`

Examples

Create a FITS file containing the red channel of an RGB image.

```
X = imread('ngc6543a.jpg');
R = X(:,:,1);
fitswrite(R,'myfile.fits');
fitsdisp('myfile.fits');
```

Create a FITS file with three images constructed from the channels of an RGB image.

```
X = imread('ngc6543a.jpg');
R = X(:,:,1);  G = X(:,:,2);  B = X(:,:,3);
```

```
fitswrite(R,'myfile.fits');
fitswrite(G,'myfile.fits','writemode','append');
fitswrite(B,'myfile.fits','writemode','append');
fitsdisp('myfile.fits');
```

References

For copyright information, see the `cfitsiocopyright.txt` file.

See Also

`fitsinfo` | `fitsread`

Introduced in R2012a

fix

Round toward zero

Syntax

```
Y = fix(X)
```

Description

`Y = fix(X)` rounds each element of `X` to the nearest integer toward zero. For positive `X`, the behavior of `fix` is the same as `floor`. For negative `X`, the behavior of `fix` is the same as `ceil`.

Examples

Round Matrix Elements Toward Zero

```
X = [-1.9 -3.4; 1.6 2.5; -4.5 4.5]
```

```
X = 3×2

   -1.9000   -3.4000
    1.6000    2.5000
   -4.5000    4.5000
```

```
Y = fix(X)
```

```
Y = 3×2

    -1    -3
     1     2
    -4     4
```

Round Complex Numbers Toward Zero

```
X = [1.4+2.3i 3.1-2.2i -5.3+10.9i]
```

```
X = 1×3 complex

   1.4000 + 2.3000i   3.1000 - 2.2000i  -5.3000 +10.9000i
```

```
Y = fix(X)
```

```
Y = 1×3 complex

   1.0000 + 2.0000i   3.0000 - 2.0000i  -5.0000 +10.0000i
```

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. For complex `X`, `fix` treats the real and imaginary parts independently.

`fix` converts logical and `char` elements of `X` into `double` values.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `logical`
Complex Number Support: Yes

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support `char` or `logical` data types for `X`.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Code generation does not support `char` or `logical` data types for `X`.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`ceil` | `floor` | `round`

Topics

“Integers”
“Floating-Point Numbers”

Introduced before R2006a

matlab.unittest.fixtures Package

Summary of classes in MATLAB Fixtures Interface

Description

Fixtures ease creation of setup and teardown code. The `matlab.unittest.fixtures` package consists of the following customized MATLAB fixtures.

Classes

matlab.unittest.fixtures.CurrentFolderFixture	Fixture for changing current working folder
matlab.unittest.fixtures.Fixture	Interface class for test fixtures
matlab.unittest.fixtures.PathFixture	Fixture for adding a folder to the MATLAB path
matlab.unittest.fixtures.ProjectFixture	Fixture for loading project
matlab.unittest.fixtures.SuppressedWarningsFixture	Fixture to suppress display of warnings
matlab.unittest.fixtures.TemporaryFolderFixture	Fixture for creating a temporary folder
matlab.unittest.fixtures.WorkingFolderFixture	Fixture for creating and changing to temporary working folder

See Also

Topics

"Write Tests Using Shared Fixtures"
"Create Basic Custom Fixture"
"Create Advanced Custom Fixture"

Introduced in R2013a

matlab.unittest.fixtures.CurrentFolderFixture class

Package: `matlab.unittest.fixtures`

Fixture for changing current working folder

Description

The `CurrentFolderFixture` class provides a fixture for changing the current working folder. When the test framework sets up the fixture, it changes the working folder. When the test framework tears down the fixture, it restores the working folder to its previous state.

Construction

`matlab.unittest.fixtures.CurrentFolderFixture(folder)` constructs a fixture for changing the current working folder to `folder`.

Input Arguments

`folder` — Folder to make the current working folder
character vector

Folder to make the current working folder, specified as a character vector. MATLAB throws an error if `folder` does not exist.

Properties

`Folder`

Folder to make the current working folder, specified as a character vector in the `folder` input argument.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Create Fixture to Change Current Working Folder

Create the following `changeFolderFixtureTest` class definition on your MATLAB path. This example assumes that the subfolder `helperFiles` exists in your working folder. Create the `changeToFolder`in your working folder if it does not exist.

The `test1` function includes a call to `pwd` to demonstrate the current path changed to the `helperFiles` folder.

```
classdef changeFolderFixtureTest < matlab.unittest.TestCase
    methods(Test)
        function test1(testCase)
            import matlab.unittest.fixtures.CurrentFolderFixture

            changeToFolder = 'helperFiles';
            testCase.applyFixture(CurrentFolderFixture ...
                (changeToFolder));
            pwd
        end
    end
end
```

At the command prompt, run the test. For the purposes of this example, call `pwd` before and after `run` to show the fixture was properly torn down and the path returned to the pre-test state.

```
currentFolderBeforeTest = pwd
run(changeFolderFixtureTest);
currentFolderAfterTest = pwd
```

```
currentFolderBeforeTest =

H:\Documents\doc_examples

Running changeFolderFixtureTest
```

```
ans =

H:\Documents\doc_examples\helperFiles

.
Done changeFolderFixtureTest
__________


currentFolderAfterTest =

H:\Documents\doc_examples
```

See Also

applyFixture | matlab.unittest.fixtures | matlab.unittest.fixtures.PathFixture | matlab.unittest.fixtures.WorkingFolderFixture

matlab.unittest.fixtures.ProjectFixture class

Package: `matlab.unittest.fixtures`
Superclasses: `matlab.unittest.fixtures.Fixture`

Fixture for loading project

Description

`ProjectFixture` is a fixture for loading a project during execution of a test suite.

The test framework constructs this class. When the test framework sets up the fixture, it loads the project if not already loaded. When the test framework tears down the fixture, the project is closed if the fixture loaded it, otherwise the project remains loaded.

The `matlab.unittest.fixtures.ProjectFixture` class is a `handle` class.

Class Attributes

`HandleCompatible`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Instances of this class are created only by the testing framework.

Properties

`ProjectFolder` — Project root folder
path to project root folder

Project root folder, specified as a string scalar. `ProjectFolder` contains the absolute path to the root folder of the project that is loaded when the fixture is set up.

Example: `C:/projects/project1/`

Attributes:

GetAccess	public
SetAccess	immutable

Data Types: string

Limitations

- You cannot use the fromProject method in compiled applications created with the MATLAB Compiler™. For more information, see “Functions Not Supported for Compilation by MATLAB Compiler and MATLAB Compiler SDK” (MATLAB Compiler).

See Also

applyFixture | matlab.unittest.fixtures

Introduced in R2019a

matlab.unittest.fixtures.Fixture class

Package: `matlab.unittest.fixtures`

Interface class for test fixtures

Description

The `Fixture` interface class is the means by which test authors create custom fixtures. Fixtures configure the environment state required for tests.

Classes deriving from the `Fixture` interface must implement the `setup` method. This method executes the changes to the environment. A fixture should restore the environment to its initial state when it is torn down. To restore the environment, use the `addTeardown` method in the `setup` method or implement the fixture's `teardown` method.

Subclasses can set the `SetupDescription` and `TeardownDescription` properties in their constructors to provide descriptions for the actions performed by the `setup` and `teardown` methods. The testing framework can display these descriptions when setting up and tearing down the fixture.

A class that derives from `Fixture` must implement the `isCompatible` method if its constructor accepts any input arguments or is otherwise configurable. `Fixture` subclasses use this method to define a notion of interchangeability of fixtures. Two `matlab.unittest.fixtures` instances of the same class are considered to be interchangeable if the `isCompatible` method returns `true`. The `TestRunner` uses the result of `isCompatible` to determine whether two fixture instances of the same class correspond to the same shared test fixture state.

Properties

`SetupDescription`

Description of fixture setup actions, specified as a character vector. The `SetupDescription` property describes the actions the fixture performs when the testing framework invokes the fixture's `setup` method.

`TeardownDescription`

Description of fixture teardown actions, specified as a character vector. The `TeardownDescription` property describes the actions the fixture performs when the testing framework invokes the fixture's `teardown` method.

Methods

applyFixture	Set up fixture to delegate work to another fixture
log	Record diagnostic information
onFailure	Dynamically add diagnostics for test failures
setup	Set up fixture
teardown	Tear down fixture
addTeardown	Dynamically add teardown routine
isCompatible	Determine if two fixtures of the same class are interchangeable

Events

`AssertionFailed`	Triggered upon failing assertion. A `QualificationEventData` object is passed to listener callback functions.
`AssertionPassed`	Triggered upon passing assertion. A `QualificationEventData` object is passed to listener callback functions.
`AssumptionFailed`	Triggered upon failing assumption. A `QualificationEventData` object is passed to listener callback functions.
`AssumptionPassed`	Triggered upon passing assumption. A `QualificationEventData` object is passed to listener callback functions.
`FatalAssertionFailed`	Triggered upon failing fatal assertion. A `QualificationEventData` object is passed to listener callback functions.

`FatalAssertionPassed`	Triggered upon passing fatal assertion. A `QualificationEventData` object is passed to listener callback functions.
`ExceptionThrown`	Triggered by the `TestRunner` when an exception is thrown. An `ExceptionEventData` object is passed to listener callback functions.
`DiagnosticLogged`	Triggered by the `TestRunner` upon a call to the `log` method. A `LoggedDiagnosticEventData` object is passed to the listener callback functions.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

See Also

`addTeardown` | `applyFixture` | `getSharedTestFixtures` | `matlab.unittest.diagnostics.LoggedDiagnosticEventData` | `matlab.unittest.fixtures` | `matlab.unittest.qualifications.ExceptionEventData` | `matlab.unittest.qualifications.QualificationEventData`

Topics

"Create Basic Custom Fixture"
"Create Advanced Custom Fixture"

applyFixture

Class: `matlab.unittest.fixtures.Fixture`
Package: `matlab.unittest.fixtures`

Set up fixture to delegate work to another fixture

Syntax

```
applyFixture(fixture1,fixture2)
f = applyFixture(fixture1,fixture2)
```

Description

`applyFixture(fixture1,fixture2)` sets up `fixture2` for use with `fixture1`. To delegate work to `fixture2`, call this method from the `setup` method of `fixture1`.

The `applyFixture` method ties the lifecycle of `fixture2` to the lifecycle of `fixture1`. When the testing framework tears down `fixture1`, it also tears down `fixture2`.

`f = applyFixture(fixture1,fixture2)` also returns `fixture2` as an output once it has been set up.

Input Arguments

`fixture1` — Primary fixture
instance of `matlab.unittest.fixtures.Fixture`

Primary fixture, specified as an instance of `matlab.unittest.fixtures.Fixture`.

`fixture2` — Fixture applied to primary fixture
instance of `matlab.unittest.fixtures.Fixture`

Fixture for setting up the primary fixture, specified as an instance of `matlab.unittest.fixtures.Fixture`. The primary fixture delegates work to this fixture.

Examples

Delegate Work to Suppress Display of Warning

Construct a fixture that removes a folder from the search path and suppresses the warning that occurs when the folder you try to remove is not on the path. The fixture uses a `SuppressedWarningsFixture` fixture to disable a specified warning. As part of the `RemoveFolderFromPathFixture` teardown, the testing framework also tears down the `SuppressedWarningsFixture` at the appropriate time, which restores the state of warning to its previous value.

```
classdef RemoveFolderFromPathFixture < matlab.unittest.fixtures.Fixture
    properties (SetAccess = immutable)
        Folder (1,1) string
    end
    methods
        function fixture = RemoveFolderFromPathFixture(folder)
            fixture.Folder = folder;
        end
        function setup(fixture)
            import matlab.unittest.fixtures.SuppressedWarningsFixture

            % Delegate to SuppressedWarningsFixture to suppress display of warnings.
            fixture.applyFixture(SuppressedWarningsFixture('MATLAB:rmpath:DirNotFound'

            % Remove the folder from the path.
            originalPath = path;
            fixture.addTeardown(@()path(originalPath));
            rmpath(fixture.Folder);
        end
    end
    methods (Access = protected)
        function bool = isCompatible(fixture1,fixture2)
            bool = fixture1.Folder == fixture2.Folder;
        end
    end
end
```

Delegate Work to Create Temporary Text File

Construct a fixture that creates a temporary text file. The fixture uses a `TemporaryFolderFixture` fixture to create a temporary folder, and then creates a text file within that folder. As part of the `TemporaryTextFileFixture` teardown, the framework also tears down the `TemporaryFolderFixture` at the appropriate time, which deletes the folder and its contents.

```
classdef TemporaryTextFileFixture < matlab.unittest.fixtures.Fixture
    properties (SetAccess = private)
        File
    end
    methods
        function setup(fixture)
            import matlab.unittest.fixtures.TemporaryFolderFixture

            % Delegate to TemporaryFolderFixture to create a temporary folder.
            tempFixture = fixture.applyFixture(TemporaryFolderFixture);

            fixture.File = fullfile(tempFixture.Folder,'file.txt');

            fid = fopen(fixture.File,'wt');
            fixture.fatalAssertNotEqual(fid,-1);

            status = fclose(fid);
            fixture.fatalAssertEqual(status,0);
        end
    end
end
```

See Also

`applyFixture` | `matlab.unittest.fixtures`

Introduced in R2016a

log

Class: `matlab.unittest.fixtures.Fixture`
Package: `matlab.unittest.fixtures`

Record diagnostic information

Syntax

```
log(f,diagnostic)
log(f,v,diagnostic)
```

Description

`log(f,diagnostic)` logs the supplied diagnostic. The log method provides a means for tests to log information during fixture setup and teardown routines. The testing framework displays logged messages only if you configure it to do so by adding an appropriate plugin, such as the `matlab.unittest.plugins.LoggingPlugin`.

`log(f,v,diagnostic)` logs the diagnostic at the specified verbosity level, `v`.

Input Arguments

`f` — Instance of fixture
`matlab.unittest.fixtures.Fixture` instance

Instance of fixture, specified as a `matlab.unittest.fixtures.Fixture`.

`diagnostic` — Diagnostic information to display upon a failure
string | character vector | function handle | `matlab.unittest.diagnostics.Diagnostic` instance

Diagnostic information to display upon a failure, specified as a string, character vector, function handle, or `matlab.unittest.diagnostics.Diagnostic` instance.

v — Verbosity level
2 (default) | 1 | 3 | 4 | `matlab.unittest.Verbosity` enumeration

Verbosity level, specified as an integer value between 1 and 4 or a `matlab.unittest.Verbosity` enumeration object. The default verbosity level for diagnostic messages is `Concise`. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration.

Numeric Representation	Enumeration Member Name	Verbosity Description
1	`Terse`	Minimal information
2	`Concise`	Moderate amount of information
3	`Detailed`	Some supplemental information
4	`Verbose`	Lots of supplemental information

Examples

Log Diagnostic Information

In a file, `FormatHexFixture.m`, in your current working folder, create the following fixture.

```
classdef FormatHexFixture < matlab.unittest.fixtures.Fixture
    properties (Access=private)
        OriginalFormat
    end
    methods
        function setup(fixture)
            fixture.OriginalFormat = get(0,'Format');
            fixture.log(['The previous format setting was ',...
                fixture.OriginalFormat])
            log(fixture,'Setting Format')
            set(0,'Format','hex')
            log(fixture,3,'Format Set')
        end
        function teardown(fixture)
```

```
            log(fixture,'Resetting Format')
            set(0,'Format',fixture.OriginalFormat)
            log(fixture,3,'Original Format Restored')
        end
    end
end
```

In a file, `SampleTest.m`, in your current working folder, create the following test class.

```
classdef SampleTest < matlab.unittest.TestCase
    methods (Test)
        function test1(testCase)
            testCase.applyFixture(FormatHexFixture);
            actStr = getColumnForDisplay([1;2;3], 'Small Integers');
            expStr = ['Small Integers  '
                '3ff0000000000000'
                '4000000000000000'
                '4008000000000000'];
            testCase.verifyEqual(actStr, expStr)
        end
    end
end

function str = getColumnForDisplay(values, title)
elements = cell(numel(values)+1, 1);
elements{1} = title;
for idx = 1:numel(values)
    elements{idx+1} = displayNumber(values(idx));
end
str = char(elements);
end

function str = displayNumber(n)
str = strtrim(evalc('disp(n);'));
end
```

Run the test.

```
result = run(SampleTest);
```

```
Running SampleTest
.
Done SampleTest
__________
```

None of the logged messages are displayed because the default test runner has a verbosity level of 1 (`Terse`) and the default log message is at level 2 (`Concise`).

Create a test runner to report the diagnostics at levels 1, 2, and 3 and rerun the test.

```
import matlab.unittest.TestRunner
import matlab.unittest.plugins.LoggingPlugin

ts = matlab.unittest.TestSuite.fromClass(?SampleTest);
runner = TestRunner.withNoPlugins;
p = LoggingPlugin.withVerbosity(3);
runner.addPlugin(p);

results = runner.run(ts);
```

```
 [Concise] Diagnostic logged (2014-04-23T13:17:35): The previous format setting was sho
 [Concise] Diagnostic logged (2014-04-23T13:17:35): Setting Format
[Detailed] Diagnostic logged (2014-04-23T13:17:35): Format Set
 [Concise] Diagnostic logged (2014-04-23T13:17:35): Resetting Format
[Detailed] Diagnostic logged (2014-04-23T13:17:35): Original Format Restored
```

See Also

`matlab.unittest.Verbosity` | `matlab.unittest.plugins.LoggingPlugin`

Introduced in R2014b

onFailure

Class: `matlab.unittest.fixtures.Fixture`
Package: `matlab.unittest.fixtures`

Dynamically add diagnostics for test failures

Note In release R2019a, `onFailure` method has protected access.

Syntax

```
onFailure(fixture,failureDiag)
onFailure(fixture,failureDiag,'IncludingAssumptionFailures',tf)
```

Description

`onFailure(fixture,failureDiag)` adds diagnostics for failures during fixture setup and teardown routines. If the test framework encounters a failure, then it executes the diagnostics. By default, these diagnostics execute upon verification failures, assertion failures, fatal assertion failures, and uncaught exceptions.

`onFailure(fixture,failureDiag,'IncludingAssumptionFailures',tf)` indicates if the test framework also executes diagnostics upon assumption failures. To execute diagnostics upon assumption failures also, set `tf` to `true`.

Input Arguments

`fixture` — Instance of fixture
`matlab.unittest.fixtures.Fixture` instance

Instance of fixture, specified as a `matlab.unittest.fixtures.Fixture`.

`failureDiag` — Diagnostic information
character vector | string array | function handle | array of `matlab.unittest.diagnostics.Diagnostic` instances

Diagnostic information to display upon a failure, specified as a character vector, string array, function handle, or array of `matlab.unittest.diagnostics.Diagnostic` instances.

Example: `@() disp('Failure Detected')`

Example: `matlab.unittest.diagnostics.ScreenshotDiagnostic`

`tf` — React to assumption failures
`false` (default) | `true`

React to assumption failures, specified as `false` (`logical` 0) or `true` (`logical` 1). By default, this value is `false` and the test framework executes diagnostics upon verification failures, assertion failures, fatal assertion failures, and uncaught exceptions. However, the test framework does not execute diagnostics upon assumption failures. To execute additional diagnostics upon assumption failures, specify this value as `true`.

Examples

Add Diagnostics for Fixture Failure

In your current working folder, create the `FormatHexFixture` fixture. The test framework encounters an assertion failure during fixture setup and displays the diagnostic message upon failure.

```
classdef FormatHexFixture < matlab.unittest.fixtures.Fixture
    properties (Access=private)
        OriginalFormat
    end
    methods
        function setup(fixture)
            fixture.OriginalFormat = get(0,'Format');
            set(0,'Format','hex')
            fixture.addTeardown(@set,0,'Format',fixture.OriginalFormat)

            fixture.onFailure(['Failure detected in ' mfilename('class')])
            fixture.assertEqual(get(0,'Format'),'short')
        end
    end
end
```

Create the following test class, `SampleTest.m`, in your current working folder.

```
classdef SampleTest < matlab.unittest.TestCase
    methods (Test)
        function test1(testCase)
```

```
            testCase.applyFixture(FormatHexFixture);
            actStr = getColumnForDisplay([1;2;3], 'Small Integers');
            expStr = ['Small Integers  '
                '3ff0000000000000'
                '4000000000000000'
                '4008000000000000'];
            testCase.verifyEqual(actStr, expStr)
        end
    end
end

function str = getColumnForDisplay(values, title)
elements = cell(numel(values)+1, 1);
elements{1} = title;
for idx = 1:numel(values)
    elements{idx+1} = displayNumber(values(idx));
end
str = char(elements);
end

function str = displayNumber(n)
str = strtrim(evalc('disp(n);'));
end
```

Run the test. Note the `onFailure` diagnostics appear under the "Additional Diagnostics" heading.

```
results = runtests('SampleFixtureTest');
```

```
Running SampleFixtureTest

================================================================================
Assertion failed in SampleFixtureTest/test1 and it did not run to completion.

    ---------------------
    Framework Diagnostic:
    ---------------------
    assertEqual failed.
    --> The character arrays are not equal.

    Actual char:
        hex
    Expected char:
        short

    ----------------------
    Additional Diagnostic:
    ----------------------
    Failure Detected in FormatHexFixture

    -----------------
```

```
    Stack Information:
    ------------------
    In C:\work\FormatHexFixture.m (FormatHexFixture.setup) at 11
================================================================================
.
Done SampleFixtureTest
__________

Failure Summary:

     Name                      Failed  Incomplete  Reason(s)
    ===============================================================
     SampleFixtureTest/test1    X         X       Failed by assertion.
```

Compatibility Considerations

Access changed to protected

Behavior changed in R2019a

In release R2019a, the `onFailure` method `Access` attribute is changed from `public` to `protected`. This change restricts the use of `onFailure` to classes derived from `matlab.unittest.fixtures.Fixture`.

See Also

`matlab.unittest.diagnostics`

Introduced in R2017b

setup

Class: `matlab.unittest.fixtures.Fixture`
Package: `matlab.unittest.fixtures`

Set up fixture

Syntax

```
setup(f)
```

Description

`setup(f)` sets up a fixture by performing the defined environment modifications. Classes deriving from the `Fixture` interface must implement the `setup` method. This method executes the changes to the environment. A fixture should restore the environment to its initial state when it is torn down. To restore the environment, use the `addTeardown` method in the `setup` method or implement the fixture's `teardown` method.

Input Arguments

f

`matlab.unittest.fixtures.Fixture` instance

See Also

`addTeardown` | `matlab.unittest.fixtures.Fixture` | `teardown`

Topics

"Create Basic Custom Fixture"
"Create Advanced Custom Fixture"

teardown

Class: `matlab.unittest.fixtures.Fixture`
Package: `matlab.unittest.fixtures`

Tear down fixture

Syntax

```
teardown(f)
```

Description

`teardown(f)` tears down a fixture by performing the defined actions to restore the environment to the initial state.

Input Arguments

f

`matlab.unittest.fixtures.Fixture` instance

Alternatives

Instead of defining a `teardown` method, you can define teardown actions within the `setup` method by implementing the `addTeardown` method.

See Also

`addTeardown` | `matlab.unittest.fixtures.Fixture` | `setup`

Topics

"Create Basic Custom Fixture"

“Create Advanced Custom Fixture”

addTeardown

Class: `matlab.unittest.fixtures.Fixture`
Package: `matlab.unittest.fixtures`

Dynamically add teardown routine

Syntax

```
addTeardown(fixture,tearDownFcn)
addTeardown(fixture,tearDownFcn,arg1,...,argN)
```

Description

`addTeardown(fixture,tearDownFcn)` adds the `tearDownFcn` function handle that defines fixture teardown code to the `fixture` instance. The teardown code is executed in the reverse order to which it is added. This is known as LIFO (or last-in, first-out).

`addTeardown(fixture,tearDownFcn,arg1,...,argN)` provides input arguments to the `tearDownFcn`.

Input Arguments

`fixture`

`matlab.unittest.fixtures.Fixture` instance

`tearDownFcn`

Function that defines the fixture teardown code, specified as a function handle.

`arg1,...,argN`

Input arguments required by `tearDownFcn`, specified by any type. The argument type is specified by the `tearDownFcn` function argument list.

Alternatives

Instead of defining teardown actions within the `setup` method by implementing the `addTeardown` method, you can implement the `teardown` method.

See Also

`matlab.unittest.fixtures.Fixture` | `setup` | `teardown`

Topics

"Create Advanced Custom Fixture"

isCompatible

Class: `matlab.unittest.fixtures.Fixture`
Package: `matlab.unittest.fixtures`

Determine if two fixtures of the same class are interchangeable

Syntax

```
TF = isCompatible(f1, f2)
```

Description

`TF = isCompatible(f1, f2)` determines if two fixtures of the same class are interchangeable. The `isCompatible` method returns either logical `1` (`true`) or logical `0` (`false`).

A class that derives from `Fixture` must implement the `isCompatible` method if its constructor accepts any input arguments or is otherwise configurable. `Fixture` subclasses use this method to define a notion of interchangeability of fixtures. Two `matlab.unittest.fixtures` instances of the same class are considered to be interchangeable if the `isCompatible` method returns `true`. The test framework uses the result of `isCompatible` to determine whether two fixture instances of the same class correspond to the same shared test fixture state. The test framework always calls the `isCompatible` method with two fixture instances of the same class, so the fixture author does not need to implement code to handle the case where the second fixture is a different `fixtures` class.

Input Arguments

f

`matlab.unittest.fixtures.Fixture` instance

Attributes

`Access`	`Protected`

To learn about attributes of methods, see Method Attributes.

See Also

`matlab.unittest.fixtures.Fixture`

Topics

"Create Advanced Custom Fixture"

matlab.unittest.fixtures.PathFixture class

Package: `matlab.unittest.fixtures`

Fixture for adding a folder to the MATLAB path

Description

The `PathFixture` class provides a fixture for adding a folder to the MATLAB path. When the test framework sets up the fixture, it adds the specified folder to the path. When the test framework tears down the fixture, it restores the MATLAB path to its previous state.

Construction

`matlab.unittest.fixtures.PathFixture(folder)` constructs a fixture for adding a folder to the MATLAB path. When the test framework sets up the fixture, it adds `folder` to the path. When it tears down the fixture, it restores the MATLAB path to its previous state.

`matlab.unittest.fixtures.PathFixture(folder,Name,Value)` constructs a fixture with additional options specified by one or more `Name,Value` pair arguments. For example, `matlab.unittest.fixtures.PathFixture('myFolder','IncludingSubfolders',true)` constructs a fixture that adds `myFolder` and any of its subfolders to the path.

Input Arguments

`folder` — Folder to add to the MATLAB path
character vector

Folder to add to the MATLAB path, specified as a character vector. If `folder` does not exist, MATLAB throws an error.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`IncludingSubfolders` — Setting to include subfolders on path
`false` (default) | `true`

Setting to include subfolders of `folder` on the path, specified as `false` or `true` (`logical 0` or `1`). This value is false by default. If you specify it as true, the test framework includes subfolders of `folder` on the path. Package, class, and private folders are not added to the path, even when this property is `true`.

`Position` — Location of where on the path to add the folder
`'begin'` (default) | `'end'`

Location of where on the path to add the folder, specified as `'begin'` or `'end'`. The default value is `'begin'`, which adds `folder` to the beginning (top) of the path.

If you use this option with `IncludingSubfolders`, the fixture adds the folder and its subfolders to the top or bottom of the path as a single block with `folder` on the top.

Properties

`Folder`

Folder to add to the MATLAB path, specified as a character vector in the `folder` input argument.

`IncludeSubfolders`

Indicator to include subfolders of `folder` on the path, specified as `false` or `true` (`logical 0` or `1`). This property is read only. It is false by default, but you can specify it as true during construction.

`Position`

Indicator of where on the path to add folder, specified as `'begin'` or `'end'`. This property is read only. It is `'begin'` by default, but you can specify it as `'end'` during construction.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Add Folder to MATLAB Path for Testing

Create the following `addPathFixtureTest` class definition on your MATLAB path. This example assumes that the subfolder,`helperFiles`, exists in your working folder. If it does not, define `addFolder` to be a folder that exists within your current folder.

```
classdef addPathFixtureTest < matlab.unittest.TestCase
    methods(Test)
        function test1(testCase)
            import matlab.unittest.fixtures.PathFixture

            addFolder = 'helperFiles';
            f = testCase.applyFixture(PathFixture(addFolder));
            disp(['Added to path: ' f.Folder])
        end
    end
end
```

At the command prompt, run the test.

```
run(addPathFixtureTest);
```

```
Running addPathFixtureTest
Added to path: H:\Documents\doc_examples\helperFiles
.
Done addPathFixtureTest
__________
```

After the tests finish running, the framework removes the folder from the path.

Add Folder to Path Using Shared Test Fixture

Create the following `sharedAddPathFixtureTest` class definition on your MATLAB path. This example assumes that the subfolder, `helperFiles`, exists in your working folder.

```
classdef (SharedTestFixtures={ ...
        matlab.unittest.fixtures.PathFixture('helperFiles')}) ...
        sharedAddPathFixtureTest < matlab.unittest.TestCase
```

```
    methods(Test)
        function test1(testCase)
            f = testCase.getSharedTestFixtures;
            disp(['Added to path: ' f.Folder])
        end
    end
end
```

At the command prompt, run the test.

```
run(sharedAddPathFixtureTest);
```

```
Setting up PathFixture
Done setting up PathFixture: Added 'H:\Documents\doc_examples\helperFiles' to the path.
__________

Running sharedAddPathFixtureTest
Added to path: H:\Documents\doc_examples\helperFiles
.
Done sharedAddPathFixtureTest
__________

Tearing down PathFixture
Done tearing down PathFixture: Restored the path to its original state.
__________
```

After the tests finish running, the framework removes the folder from the path.

See Also

applyFixture | matlab.unittest.fixtures | matlab.unittest.fixtures.CurrentFolderFixture | matlab.unittest.fixtures.WorkingFolderFixture

matlab.unittest.fixtures.SuppressedWarningsFixture class

Package: `matlab.unittest.fixtures`

Fixture to suppress display of warnings

Description

The `SuppressedWarningsFixture` class provides a fixture to suppress the display of warnings. When set up, `SuppressedWarningsFixture` disables one or more specified warnings. When torn down, the fixture restores the states of warnings to their previous values.

Construction

`matlab.unittest.fixtures.SuppressedWarningsFixture(warnIDs)` constructs a fixture to suppress the display of one or more warnings.

Input Arguments

`warnIDs` — Identifier for warnings disabled when the fixture is set up
character vector | cell array of character vectors

Warning identifiers for the warnings to be suppressed, specified as a character vector or cell array of character vectors.

Properties

`Warnings`

Warning identifiers describing warnings to suppress specified as a cell array of character vectors in the `warnings` input argument.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Create Fixture to Suppress Warnings

Suppress the warning that occurs when you try to remove a folder from the search path that is not on the search path.

Remove the folder, `folderthatisnotonpath` from your path, assuming it does not exist.

```
rmpath('folderthatisnotonpath')
```

```
Warning: "folderthatisnotonpath" not found in path.
> In rmpath at 58
```

A warning appears because `rmpath` cannot find the folder.

Suppress the warning during testing by creating the following `suppressWarningsTest` class definition on your MATLAB path.

```
classdef suppressWarningsTest < matlab.unittest.TestCase
    methods(Test)
        function test1(testCase)
            import matlab.unittest.fixtures.SuppressedWarningsFixture

            testCase.applyFixture(...
                SuppressedWarningsFixture('MATLAB:rmpath:DirNotFound'));

            % would otherwise cause warning
            rmpath('folderthatisnotonpath')
        end
    end
end
```

At the command prompt, run the test. For the purposes of this example, call `rmpath` before and after running the test to show the warning is not suppressed outside execution of the test.

```
rmpath('folderthatisnotonpath')
run(suppressWarningsTest);
rmpath('folderthatisnotonpath')
```

```
Warning: "folderthatisnotonpath" not found in path.
> In rmpath at 58
Running suppressWarningsTest
.
Done suppressWarningsTest
__________

Warning: "folderthatisnotonpath" not found in path.
> In rmpath at 58
```

Note that the call to `rmpath` within `suppressWarningsTest` does not result in a warning.

See Also

`applyFixture` | `matlab.unittest.fixtures` | `warning`

matlab.unittest.fixtures.TemporaryFolderFixture class

Package: `matlab.unittest.fixtures`

Fixture for creating a temporary folder

Description

The `matlab.unittest.fixtures.TemporaryFolderFixture` provides a fixture to create a temporary folder. When the testing framework sets up the fixture, it creates the temporary folder. When it tears down the fixture, it deletes the folder and all its contents. Before it deletes the folder, the fixture clears from memory the definitions of any MATLAB-files, P-files, and MEX-files that are defined in the temporary folder.

Both the `TemporaryFolderFixture` and `WorkingFolderFixture` fixtures create a temporary folder. Unlike the `WorkingFolderFixture`, the `TemporaryFolderFixture` does not set the folder as the current working folder.

Construction

`matlab.unittest.fixtures.TemporaryFolderFixture` constructs a fixture for creating a temporary folder.

`matlab.unittest.fixtures.TemporaryFolderFixture(Name,Value)` constructs a fixture for creating a temporary folder with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`PreservingOnFailure` — Preservation state of temporary folder and contents after test failure
`false` (default) | `true`

Indicator of whether the temporary folder and its contents are preserved in the event of a test failure, specified as `false` or `true` (`logical 0` or `1`). This property is `false` by default. You can specify it as `true` during fixture construction.

Data Types: `logical`

`WithSuffix` — Suffix for temporary folder name
character vector (default)

Suffix for temporary folder name, specified as a character vector.

Properties

`Folder`

Absolute path of the folder created by the fixture, specified as a character vector.

`PreserveOnFailure`

Indicator of whether the temporary folder and its contents are preserved in the event of a test failure. This property is `logical(0)` or `logical(1)`. It is `logical(0)` by default but is set to `logical(1)` if the `'PreservingOnFailure'` input value is set to `true` during fixture construction.

`Suffix`

Suffix used for temporary folder, specified as a character vector in the `Name,Value` pair argument, `'WithSuffix'`.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Create Temporary Folder Fixture

Create the following `tempFolderFixtureTest` class definition on your MATLAB path.

```
classdef tempFolderFixtureTest < matlab.unittest.TestCase
    methods(Test)
        function test1(testCase)
            import matlab.unittest.fixtures.TemporaryFolderFixture

            tempFixture = testCase.applyFixture(TemporaryFolderFixture);

            disp(['The temporary folder: ' tempFixture.Folder])
        end
    end
end
```

At the command prompt, run the test.

```
run(tempFolderFixtureTest);
```

```
Running tempFolderFixtureTest
The temporary folder: C:\Temp\tpfb1ae2cf_c9de_4de3_9557_00d52bfcc1b2
.
Done tempFolderFixtureTest
__________
```

The name of the temporary folder varies.

Create Temporary Folder Fixture Persisting Through Test Failure

Create the following `anotherTempFolderFixtureTest` class definition on your MATLAB path. For the purposes of this example, the `test1` function contains an assertion that causes test failure.

```
classdef anotherTempFolderFixtureTest < matlab.unittest.TestCase
    methods(Test)
        function test1(testCase)
            import matlab.unittest.fixtures.TemporaryFolderFixture

            testCase.applyFixture(TemporaryFolderFixture( ...
                'PreservingOnFailure',true,'WithSuffix','TestData'));

            % Failed assertion, preserved temporary folder
            testCase.assertEqual(1,2)
        end
    end
end
```

At the command prompt, run the test.

```
run(anotherTempFolderFixtureTest);
```

```
Running anotherTempFolderFixtureTest

================================================================================
Assertion failed in anotherTempFolderFixtureTest/test1 and it did not run to completion.

    ---------------------
    Framework Diagnostic:
    ---------------------
    assertEqual failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
                Actual    Expected    Error    RelativeError
                ______    ________    _____    _____________

                1         2           -1       -0.5

    Actual double:
             1
    Expected double:
             2

    ------------------
    Stack Information:
    ------------------
    In C:\Documents\anotherTempFolderFixtureTest.m (anotherTempFolderFixtureTest.test1) at 10
================================================================================
   [Terse] Diagnostic logged (2014-04-01T13:50:51):
Because of a failure in the test using the TemporaryFolderFixture, the following folder will not be deleted:
C:\Temp\tp9f5aa9f1_ead1_4462_91f2_08bbe7d0316cTestData


.
Done anotherTempFolderFixtureTest
__________

Failure Summary:

     Name                                Failed  Incomplete  Reason(s)
    ===============================================================================
     anotherTempFolderFixtureTest/test1    X         X       Failed by assertion.
```

The test failed and the temporary folder persists. You can open the temporary folder, shown here as `C:\Temp\tp9f5aa9f1_ead1_4462_91f2_08bbe7d0316cTestData`, and examine any contents.

See Also

`applyFixture` | `matlab.unittest.fixtures` | `matlab.unittest.fixtures.CurrentFolderFixture` |

```
matlab.unittest.fixtures.PathFixture|
matlab.unittest.fixtures.WorkingFolderFixture
```

Introduced in R2013b

matlab.unittest.fixtures.WorkingFolderFixture class

Package: `matlab.unittest.fixtures`

Fixture for creating and changing to temporary working folder

Description

The `matlab.unittest.fixtures.WorkingFolderFixture` creates a temporary folder and sets it as the current working folder. The test or the product under test can create files and modify the contents of the folder without affecting the source code or test folder structure.

When the testing framework sets up the fixture, it adds the current folder to the path. Then, the fixture creates a temporary folder, and changes the current working folder to the temporary folder. When the testing framework tears down the fixture, by default, it deletes the temporary folder and all folder contents. The testing framework restores the current working folder to its previous state.

Both the `WorkingFolderFixture` and `TemporaryFolderFixture` fixtures create a temporary folder. However, the `WorkingFolderFixture` also sets the folder as the current working folder.

Construction

`matlab.unittest.fixtures.WorkingFolderFixture` constructs a fixture for creating and changing to a temporary working folder.

`matlab.unittest.fixtures.WorkingFolderFixture(Name,Value)` constructs a fixture with additional options specified by one or more `Name,Value` pair arguments. For example, `matlab.unittest.fixtures.WorkingFolderFixture('PreservingOnFailure', true)` constructs a fixture that does not delete the temporary folder in the event of an error.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`PreservingOnFailure` — Setting to preserve temporary folder and contents after test failure
`false` (default) | `true`

Setting to preserve the temporary folder and contents after test failure, specified as `false` or `true` (`logical 0` or `1`). This value is `false` by default. You can specify it as `true` during fixture construction.

If you specify `'PreservingOnFailure'` as `true` and a test using the fixture encounters a failure, the testing framework displays a message in the Command Window and does not delete the folder. Failures include verification, assertion, or fatal assertion qualification failures and uncaught errors within the tests that use the fixture. Preserving the temporary folder and its contents can aid in investigation of the cause of the test failure.

Data Types: `logical`

`WithSuffix` — Suffix for temporary folder name
character vector

Suffix for temporary folder name, specified as a character vector. The value of this parameter is appended to the name of the temporary folder.

Example: `WorkingFolderFixture('WithSuffix','_ProductA')`

Properties

`Folder` — Absolute path of the folder created by the fixture
character vector

Absolute path of the folder created by the fixture, specified as a character vector.

`PreserveOnFailure` — Indicator to preserve temporary folder
`false` (default) | `true`

Indicator to preserve the temporary folder and its contents in the event of a test failure, specified as `false` or `true`. Set this property through the constructor via the name-value pair argument `'PreservingOnFailure'`.

`Suffix` — Suffix for temporary folder name
character vector

Suffix for the temporary folder name, specified as a character vector. Set this property through the constructor via the name-value pair argument `'WithSuffix'`.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Create Temporary Working Folder

Create the following `ExampleTest` class definition on your MATLAB path.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function test1(testCase)
            import matlab.unittest.fixtures.WorkingFolderFixture;

            testCase.applyFixture(WorkingFolderFixture);

            x = 1:10;

            % Save a file in the temporary folder
            save('data.mat','x');

            disp(['The temporary working folder: ' pwd])
            ls
        end
    end
end
```

At the command prompt, run the test.

```
run(ExampleTest);
```

```
Running ExampleTest
The temporary working folder: C:\AppData\Local\Temp\tp6ff2cadf_9eed_4e90_88c1_5ff9ee8ab

.         ..        data.mat

.
Done ExampleTest
__________
```

The name of the temporary folder varies.

Specify Suffix for Temporary Working Folder Name

Create the following `ExampleTest2` class definition on your MATLAB path.

```
classdef ExampleTest2 < matlab.unittest.TestCase
    methods(Test)
        function test1(testCase)
            import matlab.unittest.fixtures.WorkingFolderFixture;

            f = WorkingFolderFixture('WithSuffix','_ProductA');
            testCase.applyFixture(f);

            x = 1:10;

            % Save a file in the temporary folder
            save('data.mat','x');

            disp(['The temporary working folder: ' pwd])
            ls
        end
    end
end
```

At the command prompt, run the test.

```
run(ExampleTest2);
```

```
Running ExampleTest2
The temporary working folder: C:\AppData\Local\Temp\tp72c6ce7c_a380_4f5e_be3b_4f7191a6
```

```
.         ..        data.mat

.
Done ExampleTest2
__________
```

The name of the temporary folder varies, but it ends with `_ProductA`.

See Also

`applyFixture` | `matlab.unittest.fixtures` | `matlab.unittest.fixtures.CurrentFolderFixture` | `matlab.unittest.fixtures.PathFixture` | `matlab.unittest.fixtures.TemporaryFolderFixture`

Introduced in R2016a

flag

Flag colormap array

Syntax

```
c = flag
c = flag(m)
```

Description

`c = flag` returns the flag colormap as a three-column array with the same number of rows as the colormap for the current figure. If no figure exists, then the number of rows is equal to the default length of 256. Each row in the array contains the red, green, and blue intensities for a specific color. The intensities are in the range [0,1], and the color scheme looks like this image.

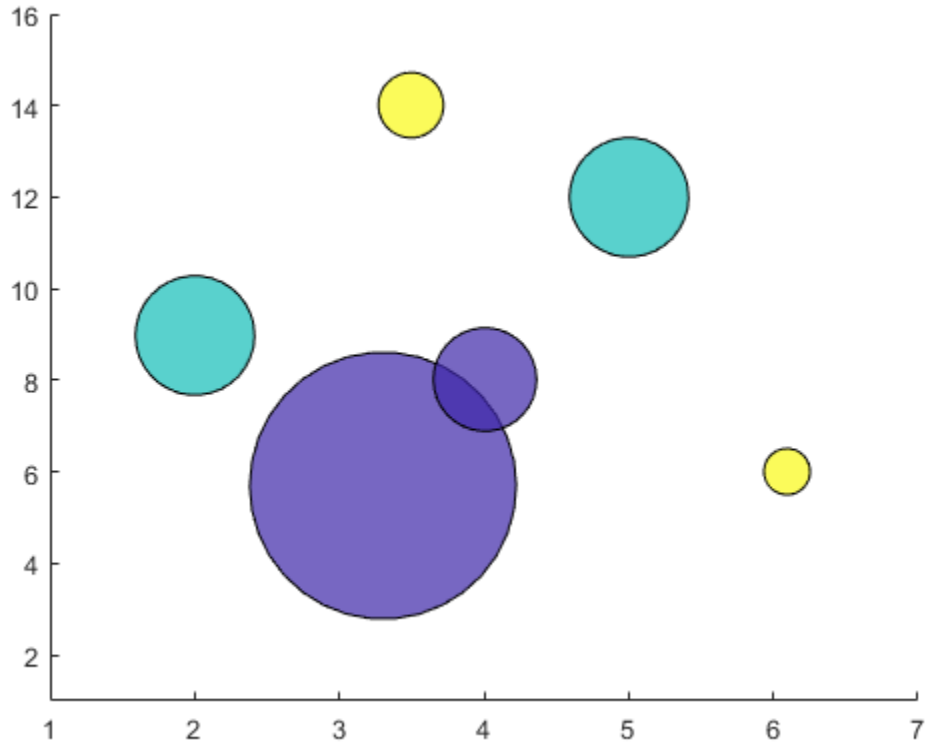
`c = flag(m)` returns the colormap with `m` colors.

Examples

Downsample the Flag Colormap

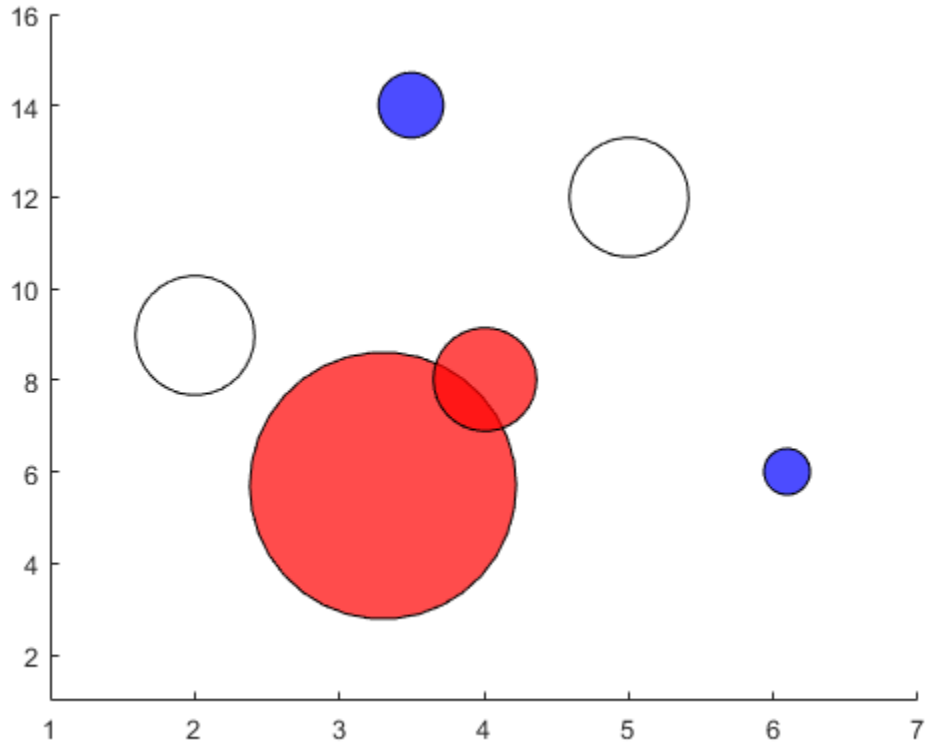
Create a scatter plot using the default colors.

```
x = [3.5 3.3 5 6.1 4 2];
y = [14 5.7 12 6 8 9];
sz = 100*[6 100 20 3 15 20];
c = [3 1 2 3 1 2];
scatter(x,y,sz,c,'filled','MarkerEdgeColor','k','MarkerFaceAlpha',.7);
xlim([1 7]);
ylim([1 16]);
```



Get the flag colormap array with three entries. Then replace the colormap in the scatter plot.

```
c = flag(3);
colormap(c);
```



Input Arguments

`m` — Number of colors
`256` (default) | scalar integer value

Number of colors, specified as a scalar integer value. The default value of `m` is equal to the length of the colormap for the current figure. If no figure exists, the default value is 256.

Data Types: `single` | `double`

Compatibility Considerations

Colormaps Have 256 Colors by Default

Behavior changed in R2019b

Starting in R2019b, colormaps have 256 colors by default.

In R2019a and previous releases, the default size is 64. If you have code that depends on a colormap having 64 colors, specify the number of colors when you set the colormap for the figure, axes, or chart. For example, `colormap(flag(64))` sets the figure's colormap to the 64-color `flag` colormap.

Alternatively, you can change the default colormap for all figures within your MATLAB session:

```
set(groot,'defaultFigureColormap',flag(64))
```

See Also

`colormap`

Topics

"Change Color Scheme Using a Colormap"

Introduced before R2006a

flintmax

Largest consecutive integer in floating-point format

Syntax

```
f = flintmax
f = flintmax(precision)
```

Description

`f = flintmax` returns the largest consecutive integer in IEEE double precision, which is `2^53`. Above this value, double-precision format does not have integer precision, and not all integers can be represented exactly.

`f = flintmax(precision)` returns the largest consecutive integer in IEEE single or double precision. `flintmax` returns `single(2^24)` for single precision and `2^53` for double precision.

Examples

Double Precision

Return the largest consecutive integer in IEEE® double precision, `2^53`.

```
format long e
f = flintmax
```

```
f =
     9.007199254740992e+15
```

Single Precision

Return the largest consecutive integer in IEEE® single precision, `single(2^24)`.

```
f = flintmax('single')
```

```
f = single
    16777216
```

Check the class of `f`.

```
class(f)
```

```
ans =
'single'
```

Limit of Integer Single Precision

Above the value returned by `flintmax('single')`, not all integers can be represented exactly with single precision.

Return the largest consecutive integer in IEEE® single precision, `single(2^24)`.

```
f = flintmax('single')
```

```
f = single
    16777216
```

Add `1` to the value returned from `flintmax`.

```
f1 = f+1
```

```
f1 = single
    16777216
```

`f1` is the same as `f`.

```
isequal(f,f1)
```

```
ans = logical
   1
```

Add 2 to the value returned from `flintmax`. The number `16777218` is represented exactly in single precision while `16777217` is not.

```
f2 = f+2
```

```
f2 = single
    16777218
```

Input Arguments

`precision` — Floating-point precision type
`'double'` (default) | `'single'`

Floating-point precision type, specified as `'double'` or `'single'`.

Data Types: `char`

Output Arguments

`f` — Largest consecutive integer in floating-point format
scalar constant

Largest consecutive integer in floating-point format returned as a scalar constant. This constant is `2^53` for double precision and `single(2^24)` for single precision.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`eps` | `format` | `intmax` | `realmax`

Topics

"Floating-Point Numbers"

Introduced in R2013a

flip

Flip order of elements

Syntax

```
B = flip(A)
B = flip(A,dim)
```

Description

`B = flip(A)` returns array `B` the same size as `A`, but with the order of the elements reversed. The dimension that is reordered in `B` depends on the shape of `A`:

- If `A` is vector, then `flip(A)` reverses the order of the elements along the length of the vector.
- If `A` is a matrix, then `flip(A)` reverses the elements in each column.
- If `A` is an N-D array, then `flip(A)` operates on the first dimension of `A` in which the size value is not `1`.

`B = flip(A,dim)` reverses the order of the elements in `A` along dimension `dim`. For example, if `A` is a matrix, then `flip(A,1)` reverses the elements in each column, and `flip(A,2)` reverses the elements in each row.

Examples

Flip Character Vector

```
A = 'no word, no bond, row on.';
B = flip(A)
```

```
B =
'.no wor ,dnob on ,drow on'
```

Flip Column Vector

```
A = [1;2;3];
B = flip(A)
```

```
B = 3×1

     3
     2
     1
```

Flip Matrix

Create a diagonal matrix, `A`.

```
A = diag([100 200 300])
```

```
A = 3×3

   100     0     0
     0   200     0
     0     0   300
```

Flip `A` without specifying the `dim` argument.

```
B = flip(A)
```

```
B = 3×3

     0     0   300
     0   200     0
   100     0     0
```

Now, flip `A` along the second dimension.

```
B = flip(A,2)
```

```
B = 3×3
```

```
     0     0   100
     0   200     0
   300     0     0
```

Flip N-D Array

Create a 1-by-3-by-2 array.

```
A = zeros(1,3,2);
A(:,:,1) = [1 2 3];
A(:,:,2) = [4 5 6];
A
```

```
A =
A(:,:,1) =

     1     2     3


A(:,:,2) =

     4     5     6
```

Flip `A` without specifying the `dim` argument.

```
B = flip(A)
```

```
B =
B(:,:,1) =

     3     2     1


B(:,:,2) =

     6     5     4
```

Now, flip `A` along the third dimension.

```
B = flip(A,3)
```

```
B =
B(:,:,1) =

     4     5     6


B(:,:,2) =

     1     2     3
```

Flip Cell Array

Create a 3-by-2 cell array.

```
A = {'foo',1000; 999,true; 'aaa','bbb'}
```

```
A=3×2 cell
    {'foo'}    {[1000]}
    {[999]}    {[   1]}
    {'aaa'}    {'bbb' }
```

Flip `A` without specifying the `dim` argument.

```
B = flip(A)
```

```
B=3×2 cell
    {'aaa'}    {'bbb' }
    {[999]}    {[   1]}
    {'foo'}    {[1000]}
```

Now, flip `A` along the second dimension.

```
B = flip(A,2)
```

```
B=3×2 cell
    {[1000]}    {'foo'}
    {[   1]}    {[999]}
    {'bbb' }    {'aaa'}
```

Input Arguments

A — Input array
vector | matrix | multidimensional array | table | timetable

Input array, specified as a vector, matrix, multidimensional array, table, or timetable.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `cell` | `table` | `timetable` | `categorical` | `datetime` | `duration` | `calendarDuration`

Example: `[1 2 3 4]`

Example: `['abcde']`

Example: `[1 2; 3 4]`

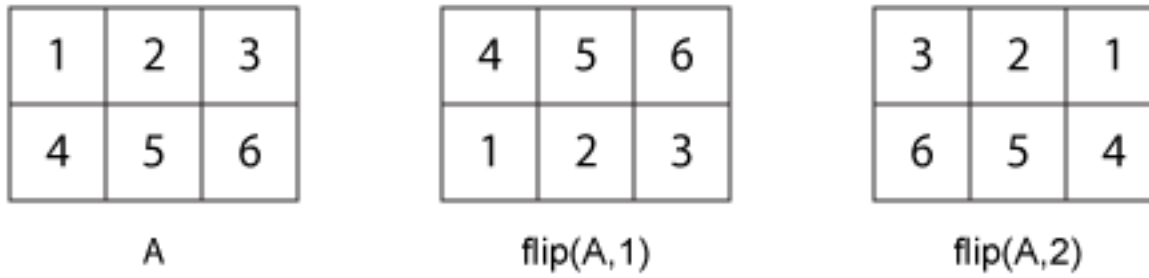
Example: `{'abcde',[1 2 3]}`

Example: `table(rand(1,5),rand(1,5))`

`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

The following illustration shows the difference between `dim=1` and `dim=2` when `A` is a matrix.



Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- If specified, the `dim` argument must be greater than `1`.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Does not support cell arrays for the first argument.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`fliplr` | `flipud` | `permute` | `rot90` | `transpose`

Introduced in R2013b

flipdim

(Not recommended) Flip array along specified dimension

Note `flipdim` is not recommended. Use `flip` instead.

Syntax

```
B = flipdim(A,dim)
```

Description

`B = flipdim(A,dim)` returns `A` with dimension `dim` flipped.

When the value of `dim` is `1`, the array is flipped row-wise down. When `dim` is `2`, the array is flipped columnwise left to right. `flipdim(A,1)` is the same as `flipud(A)`, and `flipdim(A,2)` is the same as `fliplr(A)`.

Examples

`flipdim(A,1)` where

```
A =

     1     4
     2     5
     3     6
```

produces

```
     3     6
     2     5
     1     4
```

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- If specified, the `dim` argument must be greater than `1`.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Does not support cell arrays for the first argument.

See Also

`flip` | `fliplr` | `flipud` | `permute` | `rot90`

Introduced before R2006a

fliplr

Flip array left to right

Syntax

```
B = fliplr(A)
```

Description

`B = fliplr(A)` returns `A` with its columns flipped in the left-right direction (that is, about a vertical axis).

If `A` is a row vector, then `fliplr(A)` returns a vector of the same length with the order of its elements reversed. If `A` is a column vector, then `fliplr(A)` simply returns `A`. For multidimensional arrays, `fliplr` operates on the planes formed by the first and second dimensions.

Examples

Flip Row Vector

Create a row vector.

```
A = 1:10
```

```
A = 1×10

     1     2     3     4     5     6     7     8     9    10
```

Use `fliplr` to flip the elements of `A` in the horizontal direction.

```
B = fliplr(A)
```

```
B = 1×10
```

```
    10     9     8     7     6     5     4     3     2     1
```

The order of the elements in B is reversed compared to A.

Flip Cell Array of Characters

Create a 3-by-3 cell array of characters.

```
A = {'a' 'b' 'c'; 'd' 'e' 'f'; 'g' 'h' 'i'}
```

```
A = 3x3 cell array
    {'a'}    {'b'}    {'c'}
    {'d'}    {'e'}    {'f'}
    {'g'}    {'h'}    {'i'}
```

Change the order of the columns in the horizontal direction by using `fliplr`.

```
B = fliplr(A)
```

```
B = 3x3 cell array
    {'c'}    {'b'}    {'a'}
    {'f'}    {'e'}    {'d'}
    {'i'}    {'h'}    {'g'}
```

The order of the first and third columns of A is switched in B, while the second column remains unchanged.

Flip Multidimensional Array

Create a multidimensional array.

```
A = cat(3, [1 2; 3 4], [5 6; 7 8])
```

```
A =
A(:,:,1) =

     1     2
```

```
     3     4


A(:,:,2) =

     5     6
     7     8
```

A is an array of size 2-by-2-by-2.

Flip the elements on each page of A in the horizontal direction.

```
B = fliplr(A)
```

```
B =
B(:,:,1) =

     2     1
     4     3


B(:,:,2) =

     6     5
     8     7
```

The result, B, is the same size as A, but the horizontal order of the elements is flipped. The operation flips the elements on each page independently.

Input Arguments

A — Input array
vector | matrix | multidimensional array | table | timetable

Input array, specified as a vector, matrix, multidimensional array, table, or timetable.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char | string | struct | cell | table | timetable | categorical | datetime | duration | calendarDuration
Complex Number Support: Yes

Tips

- `fliplr(A)` is equivalent to `flip(A,2)`.
- Use the `flipud` function to flip arrays in the vertical direction (that is, about a horizontal axis).
- The `flip` function can flip arrays in any direction.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Does not support cell arrays.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`flip` | `flipud` | `rot90`

Introduced before R2006a

flipud

Flip array up to down

Syntax

```
B = flipud(A)
```

Description

`B = flipud(A)` returns `A` with its rows flipped in the up-down direction (that is, about a horizontal axis).

If `A` is a column vector, then `flipud(A)` returns a vector of the same length with the order of its elements reversed. If `A` is a row vector, then `flipud(A)` simply returns `A`. For multidimensional arrays, `flipud` operates on the planes formed by the first and second dimensions.

Examples

Flip Column Vector

Create a column vector.

```
A=(1:10)'
```

```
A = 10×1

     1
     2
     3
     4
     5
     6
     7
     8
```

```
     9
    10
```

Use `flipud` to flip the elements of `A` in the vertical direction.

```
B = flipud(A)
```

```
B = 10×1

    10
     9
     8
     7
     6
     5
     4
     3
     2
     1
```

The order of the elements in `B` is reversed compared to `A`.

Flip Cell Array of Characters

Create a 3-by-3 cell array of characters.

```
A = {'a' 'b' 'c'; 'd' 'e' 'f'; 'g' 'h' 'i'}
```

```
A = 3x3 cell array
    {'a'}    {'b'}    {'c'}
    {'d'}    {'e'}    {'f'}
    {'g'}    {'h'}    {'i'}
```

Change the order of the rows in the vertical direction by using `flipud`.

```
B = flipud(A)
```

```
B = 3x3 cell array
    {'g'}    {'h'}    {'i'}
    {'d'}    {'e'}    {'f'}
```

```
    {'a'}    {'b'}    {'c'}
```

The order of the first and third rows of A is switched in B, while the second row remains unchanged.

Flip Multidimensional Array

Create a multidimensional array.

```
A = cat(3, [1 2; 3 4], [5 6; 7 8])
```

```
A =
A(:,:,1) =

     1     2
     3     4


A(:,:,2) =

     5     6
     7     8
```

A is an array of size 2-by-2-by-2.

Flip the elements on each page of A in the vertical direction.

```
B = flipud(A)
```

```
B =
B(:,:,1) =

     3     4
     1     2


B(:,:,2) =

     7     8
     5     6
```

The result, B, is the same size as A, but the vertical order of the elements is flipped. The operation flips the elements on each page independently.

Input Arguments

A — Input array
vector | matrix | multidimensional array | table | timetable

Input array, specified as a vector, matrix, multidimensional array, table, or timetable.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `cell` | `table` | `timetable` | `categorical` | `datetime` | `duration` | `calendarDuration`
Complex Number Support: Yes

Tips

- `flipud(A)` is equivalent to `flip(A,1)`.
- Use the `fliplr` function to flip arrays in the horizontal direction (that is, about a vertical axis).
- The `flip` function can flip arrays in any direction.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Does not support cell arrays.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`flip` | `fliplr` | `rot90`

Introduced before R2006a

floor

Round toward negative infinity

Syntax

```
Y = floor(X)
```

```
Y = floor(t)
Y = floor(t,unit)
```

Description

`Y = floor(X)` rounds each element of `X` to the nearest integer less than or equal to that element.

`Y = floor(t)` rounds each element of the `duration` array `t` to the nearest number of seconds less than or equal to that element.

`Y = floor(t,unit)` rounds each element of `t` to the nearest number of the specified unit of time less than or equal to that element.

Examples

Round Matrix Elements Toward Negative Infinity

```
X = [-1.9  -0.2  3.4;  5.6  7.0  2.4+3.6i];
Y = floor(X)
```

```
Y = 2×3 complex

  -2.0000 + 0.0000i  -1.0000 + 0.0000i   3.0000 + 0.0000i
   5.0000 + 0.0000i   7.0000 + 0.0000i   2.0000 + 3.0000i
```

Round Duration Values Toward Negative Infinity

Round each value in a `duration` array to the nearest number of seconds less than or equal to that value.

```
t = hours(8) + minutes(29:31) + seconds(1.23);
t.Format = 'hh:mm:ss.SS'
```

```
t = 1x3 duration array
   08:29:01.23   08:30:01.23   08:31:01.23
```

```
Y1 = floor(t)
```

```
Y1 = 1x3 duration array
   08:29:01.00   08:30:01.00   08:31:01.00
```

Round each value in `t` to the nearest number of hours less than or equal to that value.

```
Y2 = floor(t,'hours')
```

```
Y2 = 1x3 duration array
   08:00:00.00   08:00:00.00   08:00:00.00
```

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. For complex `X`, `floor` treats the real and imaginary parts independently.

`floor` converts logical and `char` elements of `X` into `double` values.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `logical`
Complex Number Support: Yes

`t` — Input duration
`duration` array

Input duration, specified as a `duration` array.

`unit` — Unit of time
`'seconds'` (default) | `'minutes'` | `'hours'` | `'days'` | `'years'`

Unit of time, specified as `'seconds'`, `'minutes'`, `'hours'`, `'days'`, or `'years'`. A duration of 1 year is equal to exactly 365.2425 24-hour days.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support `char` or `logical` data types for `X`.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Code generation does not support `char` or `logical` data types for `X`.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`ceil` | `fix` | `round`

Topics

“Integers”
“Floating-Point Numbers”

Introduced before R2006a

flow

Simple function of three variables

Syntax

```
v = flow
v = flow(n)
v = flow(x,y,z)
[x,y,z,v] = flow(...)
```

Description

`flow`, a function of three variables, generates fluid-flow data that is useful for demonstrating `slice`, `interp3`, and other functions that visualize scalar volume data.

`v = flow` produces a 25-by-50-by-25 array.

`v = flow(n)` produces a `n`-by-`2n`-by-`n` array.

`v = flow(x,y,z)` evaluates the speed profile at the points `x`, `y`, and `z`.

`[x,y,z,v] = flow(...)` returns the coordinates as well as the volume data.

See Also

`interp3` | `slice`

Topics

"Slicing Fluid Flow Data"

Introduced before R2006a

fmesh

Plot 3-D mesh

Syntax

```
fmesh(f)
fmesh(f,xyinterval)

fmesh(funx,funy,funz)
fmesh(funx,funy,funz,uvinterval)

fmesh( ___ ,LineSpec)
fmesh( ___ ,Name,Value)
fmesh(ax, ___ )
fs = fmesh( ___ )
```

Description

`fmesh(f)` creates a mesh plot of the expression `z = f(x,y)` over the default interval `[-5 5]` for `x` and `y`.

`fmesh(f,xyinterval)` plots over the specified interval. To use the same interval for both `x` and `y`, specify `xyinterval` as a two-element vector of the form `[min max]`. To use different intervals, specify a four-element vector of the form `[xmin xmax ymin ymax]`.

`fmesh(funx,funy,funz)` plots the parametric mesh defined by `x = funx(u,v)`, `y = funy(u,v)`, `z = funz(u,v)` over the default interval `[-5 5]` for `u` and `v`.

`fmesh(funx,funy,funz,uvinterval)` plots the parametric mesh over the specified interval. To use the same interval for both `u` and `v`, specify `uvinterval` as a two-element vector of the form `[min max]`. To use different intervals, specify a four-element vector of the form `[umin umax vmin vmax]`.

`fmesh( ___ ,LineSpec)` sets the line style, marker symbol, and color of the mesh. For example, `'-r'` specifies red lines. Use this option after any of the previous input argument combinations.

`fmesh( ___ ,Name,Value)` specifies properties of the mesh using one or more name-value pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

`fmesh(ax, ___ )` plots into the axes specified by `ax` instead of the current axes `gca`.

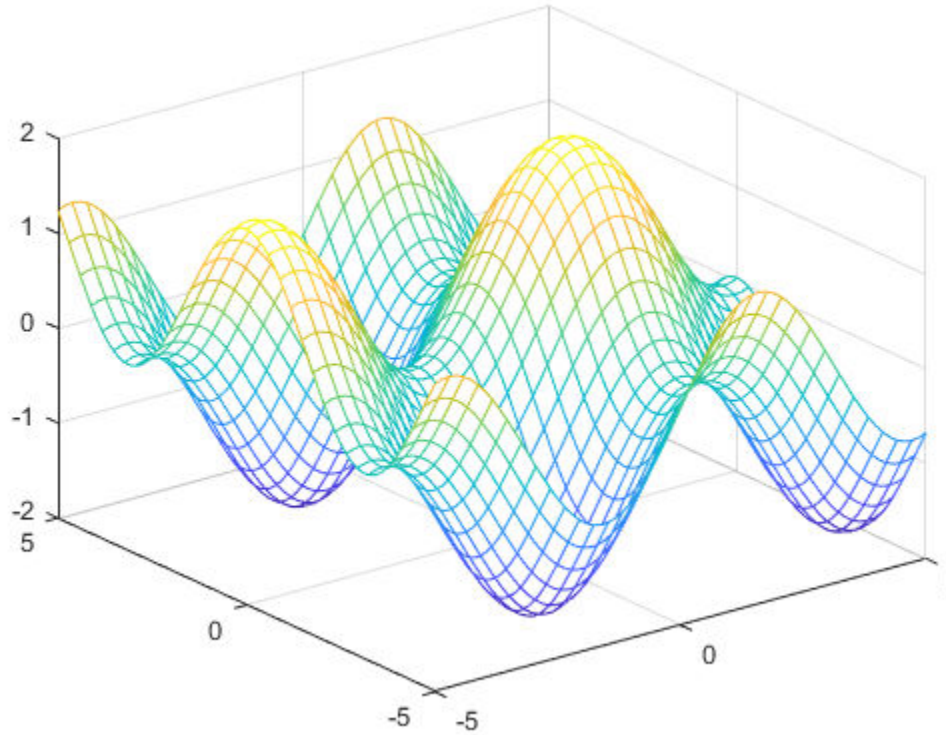
`fs = fmesh( ___ )` returns a `FunctionSurface` object or a `ParameterizedFunctionSurface` object, depending on the inputs. Use `fs` to query and modify properties of a specific surface. For a list of properties, see FunctionSurface or ParameterizedFunctionSurface.

Examples

3-D Mesh Plot of Expression

Plot a mesh of the input $\sin(x) + \cos(y)$ over the default interval $-5 < x < 5$ and $-5 < y < 5$.

```
fmesh(@(x,y) sin(x)+cos(y))
```



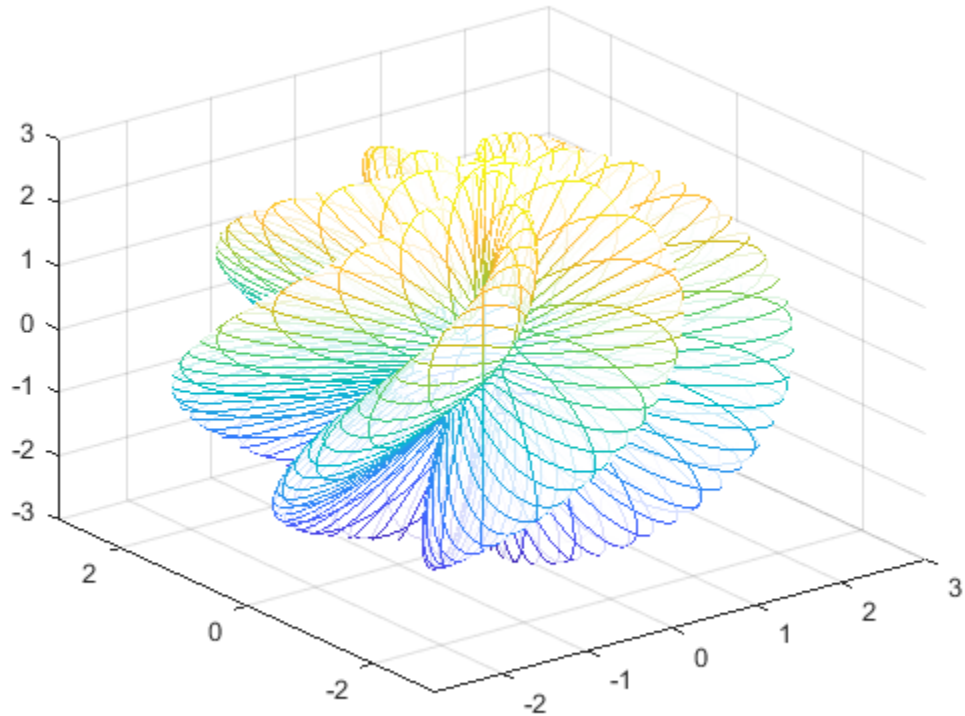
Parameterized Mesh Plot

Plot the parameterized mesh

$$x = r\cos(s)\sin(t)$$
$$y = r\sin(s)\sin(t)$$
$$z = r\cos(t)$$
$$\text{where} \quad r = 2 + \sin(7s + 5t)$$

for $0 < s < 2\pi$ and $0 < t < \pi$. Make the mesh partially transparent using `alpha`.

```
r = @(s,t) 2 + sin(7.*s + 5.*t);
x = @(s,t) r(s,t).*cos(s).*sin(t);
y = @(s,t) r(s,t).*sin(s).*sin(t);
z = @(s,t) r(s,t).*cos(t);
fmesh(x,y,z,[0 2*pi 0 pi])
```




```
alpha(0.8)
```

Specify Interval of Mesh Plot and Plot Piecewise Input

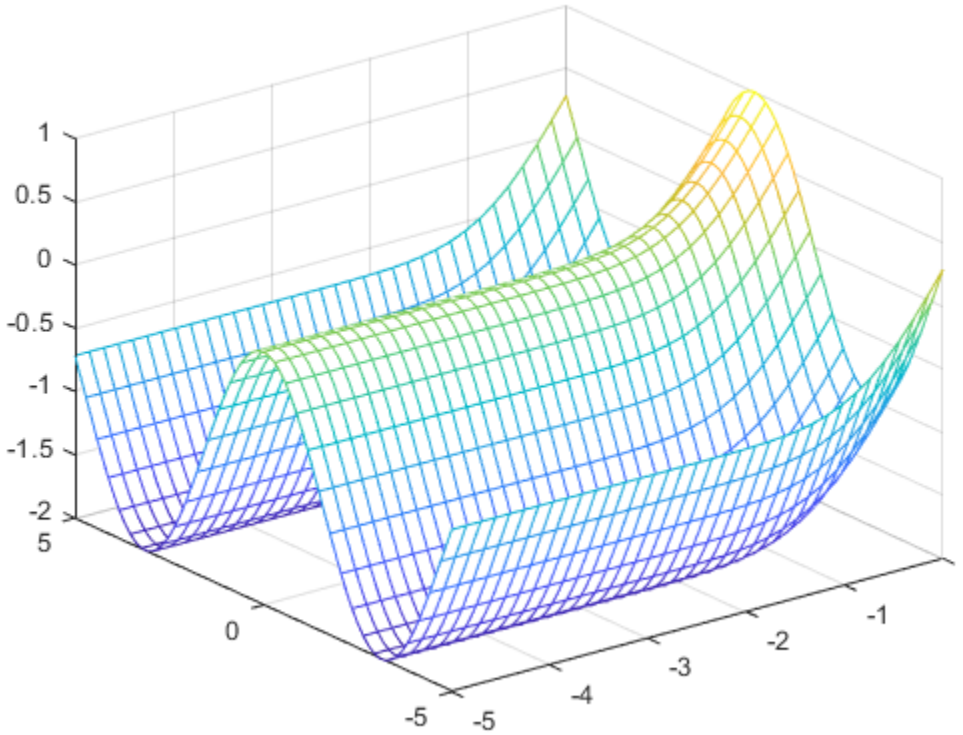
Plot the piecewise input

$$\mathrm{erf}(x) + \cos(y) \quad -5 < x < 0$$
$$\sin(x) + \cos(y) \quad 0 < x < 5$$

over the interval $-5 < y < 5$.

Specify the plotting interval as the second argument of `fmesh`. When you plot multiple inputs over different intervals in the same axes, the axis limits adjust to include all the data.

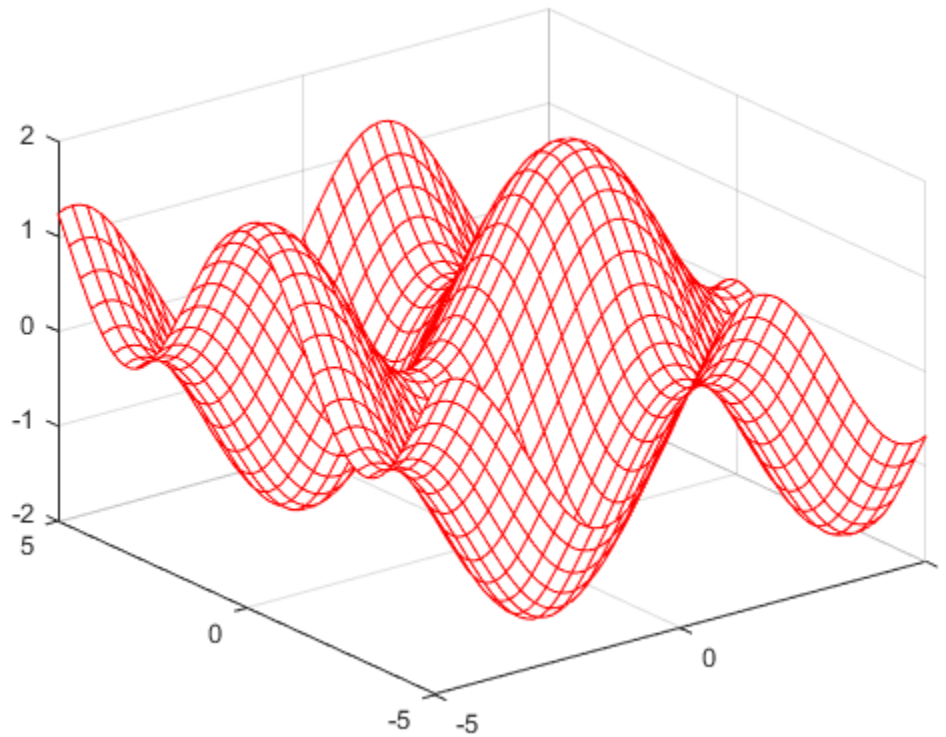
```
fmesh(@(x,y) erf(x)+cos(y),[-5 0 -5 5])
hold on
fmesh(@(x,y) sin(x)+cos(y),[0 5 -5 5])
hold off
```



Specify Mesh Plot Properties

Create a mesh plot using red lines.

```
fmesh(@(x,y) sin(x)+cos(y),'EdgeColor','red')
```



Input Arguments

f — 3-D function to plot
function handle

3-D function to plot, specified as a function handle to a named or anonymous function.

Specify a function of the form `z = f(x,y)`. The function must accept two matrix input arguments and return a matrix output argument of the same size. Use array operators instead of matrix operators for the best performance. For example, use `.*` (`times`) instead of `*` (`mtimes`).

Example: `f = @(x,y) sin(x) + cos(y);`

`xyinterval` — Plotting interval for `x` and `y`
`[-5 5 -5 5]` (default) | vector of form `[min max]` | vector of form `[xmin xmax ymin ymax]`

Plotting interval for `x` and `y`, specified in one of these forms:

- Vector of form `[min max]` — Use the interval `[min max]` for both `x` and `y`
- Vector of form `[xmin xmax ymin ymax]` — Use the interval `[xmin xmax]` for `x` and `[ymin ymax]` for `y`.

`funx` — Parametric function for *x* coordinates
function handle

Parametric function for *x* coordinates, specified as a function handle to a named or anonymous function.

Specify a function of the form `x = funx(u,v)`. The function must accept two matrix input arguments and return a matrix output argument of the same size. Use array operators instead of matrix operators for the best performance. For example, use `.*` (`times`) instead of `*` (`mtimes`).

Example: `funx = @(u,v) u.*sin(v);`

`funy` — Parametric function for *y* coordinates
function handle

Parametric function for *y* coordinates, specified as a function handle to a named or anonymous function.

Specify a function of the form `y = funy(u,v)`. The function must accept two matrix input arguments and return a matrix output argument of the same size. Use array operators instead of matrix operators for the best performance. For example, use `.*` (`times`) instead of `*` (`mtimes`).

Example: `funy = @(t) @(u,v) -u.*cos(v);`

`funz` — Parametric function for *z* coordinates
function handle

Parametric function for *z* coordinates, specified as a function handle to a named or anonymous function.

Specify a function of the form `z = funz(u,v)`. The function must accept two matrix input arguments and return a matrix output argument of the same size. Use array operators instead of matrix operators for the best performance. For example, use `.*` (`times`) instead of `*` (`mtimes`).

Example: `funz = @(u,v) v;`

`uvinterval` — Plotting interval for `u` and `v`
`[-5 5 -5 5]` (default) | vector of form `[min max]` | vector of form `[umin umax vmin vmax]`

Plotting interval for `u` and `v`, specified in one of these forms:

- Vector of form `[min max]` — Use the interval `[min max]` for both `u` and `v`.
- Vector of form `[umin umax vmin vmax]` — Use the interval `[umin umax]` for `u` and `[vmin vmax]` for `v`.

`ax` — Axes object
axes object

Axes object. If you do not specify an axes object, then `fmesh` uses the current axes.

`LineSpec` — Line specification
character vector | string

Line specification, specified as a character vector or string with a line style, marker, and color. The elements can appear in any order, and you can omit one or more options. To show only markers with no connecting lines, specify a marker and omit the line style.

Example: `'r--o'` specifies a red color, a dashed line, and circle markers

Line Style Specifier	Description
-	Solid line (default)

Line Style Specifier	Description
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Marker Specifier	Description
`o`	Circle
`+`	Plus sign
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Pentagram
`h`	Hexagram

Color Specifier	Description
`y`	yellow
`m`	magenta
`c`	cyan
`r`	red
`g`	green
`b`	blue
`w`	white
`k`	black

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Marker','o','MarkerFaceColor','red'`

The properties listed here are only a subset. For a full list, see FunctionSurface.

`MeshDensity` — Number of evaluation points per direction
35 (default) | number

Number of evaluation points per direction, specified as a number. The default is `35`. Because `fmesh` objects use adaptive evaluation, the actual number of evaluation points is greater.

Example: `100`

`ShowContours` — Display contour plot under plot
`'off'` (default) | `'on'`

Display contour plot under plot, specified as `'off'` (default) or `'on'`.

`EdgeColor` — Line color
`'interp'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as `'interp'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'interp'` colors the edges based on the `ZData` property values.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

`Marker` — Marker symbol
`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | `'x'` | `'s'` | `'d'` | `...`

Marker symbol, specified as one of the values in this table. By default, a line does not have markers. Add markers at selected points along the line by specifying a marker.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle

Value	Description
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerEdgeColor` — Marker outline color

`'auto'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'auto'` uses the same color as the `EdgeColor` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: [0.5 0.5 0.5]

Example: 'blue'

Example: '#D2F9A7'

MarkerFaceColor — Marker fill color

'none' (default) | 'auto' | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...

Marker fill color, specified as 'auto', an RGB triplet, a hexadecimal color code, a color name, or a short name. The 'auto' value uses the same color as the MarkerEdgeColor property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to

F. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Example: `'#D2F9A7'`

`MarkerSize` — Marker size
6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

Output Arguments

`fs` — One or more `FunctionSurface` or `ParameterizedFunctionSurface` objects
scalar | vector

One or more `FunctionSurface` or `ParameterizedFunctionSurface` objects, returned as a scalar or a vector.

- If you use the `fmesh(f)` syntax or a variation of this syntax, then `fmesh` returns `FunctionSurface` objects.
- If you use the `fmesh(funx,funy,funz)` syntax or a variation of this syntax, then `fmesh` returns `ParameterizedFunctionSurface` objects.

You can use these objects to query and modify properties of a specific mesh. For a list of properties, see FunctionSurface and ParameterizedFunctionSurface.

See Also

Functions
`fcontour` | `fplot` | `fplot3` | `fsurf`

Properties
FunctionSurface | ParameterizedFunctionSurface

Introduced in R2016a

fminbnd

Find minimum of single-variable function on fixed interval

Syntax

```
x = fminbnd(fun,x1,x2)
x = fminbnd(fun,x1,x2,options)
x = fminbnd(problem)
[x,fval] = fminbnd( ___ )
[x,fval,exitflag] = fminbnd( ___ )
[x,fval,exitflag,output] = fminbnd( ___ )
```

Description

`fminbnd` is a one-dimensional minimizer that finds a minimum for a problem specified by

$$\min_x f(x) \text{ such that } x_1 < x < x_2 .$$

x, x_1, and x_2 are finite scalars, and $f(x)$ is a function that returns a scalar.

`x = fminbnd(fun,x1,x2)` returns a value `x` that is a local minimizer of the scalar valued function that is described in `fun` in the interval `x1 < x < x2`.

`x = fminbnd(fun,x1,x2,options)` minimizes with the optimization options specified in `options`. Use `optimset` to set these options.

`x = fminbnd(problem)` finds the minimum for `problem`, where `problem` is a structure.

`[x,fval] = fminbnd( ___ )`, for any input arguments, returns the value of the objective function computed in `fun` at the solution `x`.

`[x,fval,exitflag] = fminbnd( ___ )` additionally returns a value `exitflag` that describes the exit condition.

`[x,fval,exitflag,output] = fminbnd( ___ )` additionally returns a structure `output` that contains information about the optimization.

Examples

Minimum of `sin`

Find the point where the sin(x) function takes its minimum in the range $0 < x < 2\pi$.

```
fun = @sin;
x1 = 0;
x2 = 2*pi;
x = fminbnd(fun,x1,x2)
```

```
x = 4.7124
```

To display precision, this is the same as the correct value $x = 3\pi/2$.

```
3*pi/2
```

```
ans = 4.7124
```

Minimize a Function Specified by a File

Minimize a function that is specified by a separate function file. A function accepts a point `x` and returns a real scalar representing the value of the objective function at `x`.

Write the following function as a file, and save the file as `scalarobjective.m` on your MATLAB® path.

```
function f = scalarobjective(x)
f = 0;
for k = -10:10
    f = f + (k+1)^2*cos(k*x)*exp(-k^2/2);
end
```

Find the `x` that minimizes `scalarobjective` on the interval `1 <= x <= 3`.

```
x = fminbnd(@scalarobjective,1,3)
```

```
x =

    2.0061
```

Minimize with Extra Parameter

Minimize a function when there is an extra parameter. The function $\sin(x - a)$ has a minimum that depends on the value of the parameter a. Create an anonymous function of x that includes the value of the parameter a. Minimize this function over the interval $0 < x < 2\pi$.

```
a = 9/7;
fun = @(x)sin(x-a);
x = fminbnd(fun,1,2*pi)
```

```
x = 5.9981
```

This answer is correct; the theoretical value is

```
3*pi/2 + 9/7
```

```
ans = 5.9981
```

For more information about including extra parameters, see “Parameterizing Functions”.

Monitor Iterations

Monitor the steps `fminbnd` takes to minimize the $\sin(x)$ function for $0 < x < 2\pi$.

```
fun = @sin;
x1 = 0;
x2 = 2*pi;
options = optimset('Display','iter');
x = fminbnd(fun,x1,x2,options)
```

```
 Func-count     x          f(x)         Procedure
```

```
    1        2.39996      0.67549        initial
    2        3.88322     -0.67549        golden
    3        4.79993    -0.996171        golden
    4        5.08984    -0.929607        parabolic
    5        4.70582    -0.999978        parabolic
    6         4.7118           -1        parabolic
    7        4.71239           -1        parabolic
    8        4.71236           -1        parabolic
    9        4.71242           -1        parabolic

Optimization terminated:
 the current x satisfies the termination criteria using OPTIONS.TolX of 1.000000e-04

x = 4.7124
```

Find Minimum Location and Function Value

Find the location of the minimum of sin(x) and the value of the minimum for $0 < x < 2\pi$.

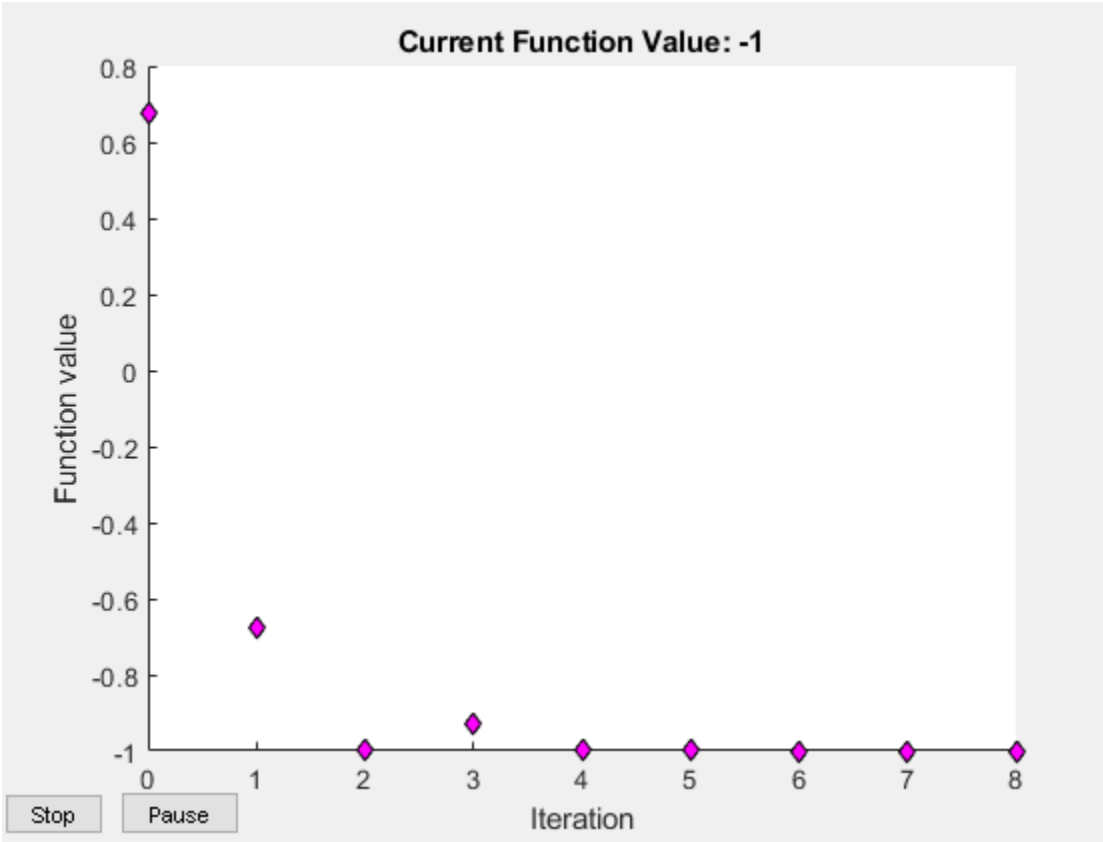
```
fun = @sin;
[x,fval] = fminbnd(fun,1,2*pi)
```

```
x = 4.7124

fval = -1.0000
```

Obtain All Information

Return all information about the `fminbnd` solution process by requesting all outputs. Also, monitor the solution process using a plot function.

```
fun = @sin;
x1 = 0;
x2 = 2*pi;
options = optimset('PlotFcns',@optimplotfval);
[x,fval,exitflag,output] = fminbnd(fun,x1,x2,options)
```




```
x = 4.7124

fval = -1.0000

exitflag = 1

output = struct with fields:
    iterations: 8
     funcCount: 9
     algorithm: 'golden section search, parabolic interpolation'
       message: 'Optimization terminated:...'
```

Input Arguments

fun — Function to minimize
function handle | function name

Function to minimize, specified as a function handle or function name. `fun` is a function that accepts a real scalar `x` and returns a real scalar `f` (the objective function evaluated at `x`).

Specify `fun` as a function handle for a file:

```
x = fminbnd(@myfun,x1,x2)
```

where `myfun` is a MATLAB function such as

```
function f = myfun(x)
f = ...            % Compute function value at x
```

You can also specify `fun` as a function handle for an anonymous function:

```
x = fminbnd(@(x)norm(x)^2,x1,x2);
```

Example: `fun = @(x)-x*exp(-3*x)`

Data Types: `char` | `function_handle` | `string`

x1 — Lower bound
finite real scalar

Lower bound, specified as a finite real scalar.

Example: `x1 = -3`

Data Types: `double`

x2 — Upper bound
finite real scalar

Upper bound, specified as a finite real scalar.

Example: `x2 = 5`

Data Types: `double`

options — Optimization options
structure such as `optimset` returns

Optimization options, specified as a structure such as `optimset` returns. You can use `optimset` to set or change the values of these fields in the options structure. See “Set Options” for detailed information.

`Display`	Level of display (see “Iterative Display”): • `'notify'` (default) displays output only if the function does not converge. • `'off'` or `'none'` displays no output. • `'iter'` displays output at each iteration. • `'final'` displays just the final output.
`FunValCheck`	Check whether objective function values are valid. The default `'off'` allows `fminbnd` to proceed when the objective function returns a value that is `complex` or `NaN`. The `'on'` setting throws an error when the objective function returns a value that is `complex` or `NaN`.
`MaxFunEvals`	Maximum number of function evaluations allowed, a positive integer. The default is `500`. See “Tolerances and Stopping Criteria”.
`MaxIter`	Maximum number of iterations allowed, a positive integer. The default is `500`. See “Tolerances and Stopping Criteria”.
`OutputFcn`	Specify one or more user-defined functions that an optimization function calls at each iteration, either as a function handle or as a cell array of function handles. The default is none (`[]`). See “Output Functions”.
`PlotFcns`	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none (`[]`). • `@optimplotx` plots the current point • `@optimplotfunccount` plots the function count • `@optimplotfval` plots the function value For information on writing a custom plot function, see “Plot Functions”.

`TolX`	Termination tolerance on `x`, a positive scalar. The default is `1e-4`. See “Tolerances and Stopping Criteria”.

Example: `options = optimset('Display','iter')`

Data Types: `struct`

`problem` — Problem structure
structure

Problem structure, specified as a structure with the following fields.

Field Name	Entry
`objective`	Objective function
`x1`	Left endpoint
`x2`	Right endpoint
`solver`	`'fminbnd'`
`options`	Options structure such as returned by `optimset`

The simplest way to obtain a `problem` structure is to export the problem from the Optimization app.

Data Types: `struct`

Output Arguments

`x` — Solution
real scalar

Solution, returned as a real scalar. Typically, `x` is a local solution to the problem when `exitflag` is positive.

`fval` — Objective function value at solution
real number

Objective function value at the solution, returned as a real number. Generally, `fval = fun(x)`.

`exitflag` — Reason `fminbnd` stopped
integer

Reason `fminbnd` stopped, returned as an integer.

`1`	Function converged to a solution `x`.
`0`	Number of iterations exceeded `options.MaxIter` or number of function evaluations exceeded `options.MaxFunEvals`.
`-1`	Stopped by an output function or plot function.
`-2`	The bounds are inconsistent, meaning `x1 > x2`.

`output` — Information about the optimization process
structure

Information about the optimization process, returned as a structure with fields:

`iterations`	Number of iterations taken
`funcCount`	Number of function evaluations
`algorithm`	`'golden section search, parabolic interpolation'`
`message`	Exit message

Limitations

- The function to be minimized must be continuous.
- `fminbnd` might only give local solutions.
- `fminbnd` can exhibit slow convergence when the solution is on a boundary of the interval.

Algorithms

`fminbnd` is a function file. The algorithm is based on golden section search and parabolic interpolation. Unless the left endpoint x_1 is very close to the right endpoint x_2, `fminbnd` never evaluates `fun` at the endpoints, so `fun` need only be defined for x in the interval $x_1 < x < x_2$.

If the minimum actually occurs at x_1 or x_2, `fminbnd` returns a point `x` in the interior of the interval (x_1,x_2) that is close to the minimizer. In this case, the distance of `x` from the minimizer is no more than `2*(TolX + 3*abs(x)*sqrt(eps))`. See [1] or [2] for details about the algorithm.

References

[1] Forsythe, G. E., M. A. Malcolm, and C. B. Moler. *Computer Methods for Mathematical Computations*. Englewood Cliffs, NJ: Prentice Hall, 1976.

[2] Brent, Richard. P. *Algorithms for Minimization without Derivatives*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For C/C++ code generation:

- `fminbnd` does not support the problem structure argument.
- `fminbnd` ignores the `Display` option and does not give iterative display or an exit message. To check solution quality, examine the exit flag.
- The output structure does not include the `algorithm` or `message` fields.
- `fminbnd` ignores the `OutputFcn` and `PlotFcns` options.

See Also

`fminsearch` | `fzero` | `optimset`

Topics

"Minimizing Functions of One Variable"
"Create Function Handle"
"Anonymous Functions"

Introduced before R2006a

fminsearch

Find minimum of unconstrained multivariable function using derivative-free method

Syntax

```
x = fminsearch(fun,x0)
x = fminsearch(fun,x0,options)
x = fminsearch(problem)
[x,fval] = fminsearch( ___ )
[x,fval,exitflag] = fminsearch( ___ )
[x,fval,exitflag,output] = fminsearch( ___ )
```

Description

Nonlinear programming solver. Searches for the minimum of a problem specified by

$$\min_{x} f(x)$$

$f(x)$ is a function that returns a scalar, and x is a vector or a matrix.

`x = fminsearch(fun,x0)` starts at the point `x0` and attempts to find a local minimum `x` of the function described in `fun`.

`x = fminsearch(fun,x0,options)` minimizes with the optimization options specified in the structure `options`. Use `optimset` to set these options.

`x = fminsearch(problem)` finds the minimum for `problem`, where `problem` is a structure.

`[x,fval] = fminsearch( ___ )`, for any previous input syntax, returns in `fval` the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag] = fminsearch( ___ )` additionally returns a value `exitflag` that describes the exit condition.

`[x,fval,exitflag,output] = fminsearch( ___ )` additionally returns a structure `output` with information about the optimization process.

Examples

Minimize Rosenbrock's Function

Minimize Rosenbrock's function, a notoriously difficult optimization problem for many algorithms:

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 .$$

The function is minimized at the point `x = [1,1]` with minimum value `0`.

Set the start point to `x0 = [-1.2,1]` and minimize Rosenbrock's function using `fminsearch`.

```
fun = @(x)100*(x(2) - x(1)^2)^2 + (1 - x(1))^2;
x0 = [-1.2,1];
x = fminsearch(fun,x0)
```

```
x = 1×2

    1.0000    1.0000
```

Monitor Optimization Process

Set options to monitor the process as `fminsearch` attempts to locate a minimum.

Set options to plot the objective function at each iteration.

```
options = optimset('PlotFcns',@optimplotfval);
```

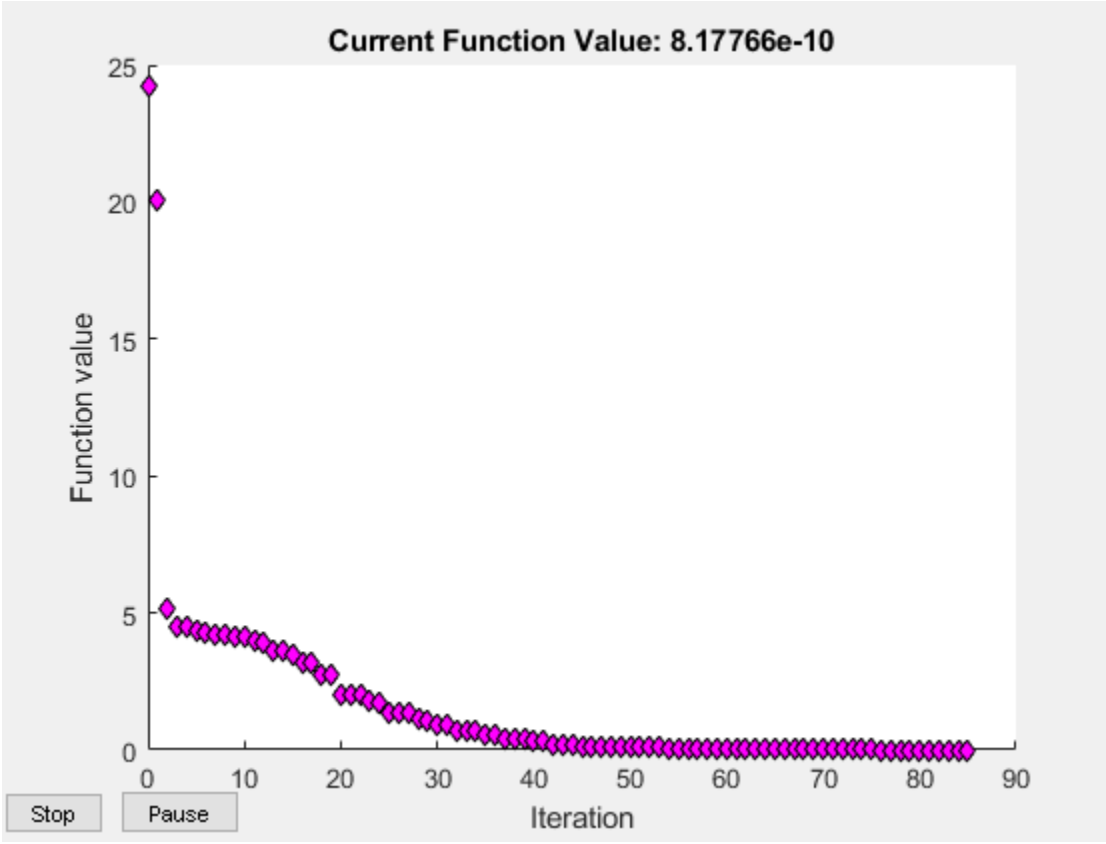
Set the objective function to Rosenbrock's function,

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 .$$

The function is minimized at the point `x = [1,1]` with minimum value `0`.

Set the start point to `x0 = [-1.2,1]` and minimize Rosenbrock's function using `fminsearch`.

```
fun = @(x)100*(x(2) - x(1)^2)^2 + (1 - x(1))^2;
x0 = [-1.2,1];
x = fminsearch(fun,x0,options)
```




```
x = 1×2

    1.0000    1.0000
```

Minimize a Function Specified by a File

Minimize an objective function whose values are given by executing a file. A function file must accept a real vector `x` and return a real scalar that is the value of the objective function.

Copy the following code and include it as a file named `objectivefcn1.m` on your MATLAB® path.

```
function f = objectivefcn1(x)
f = 0;
for k = -10:10
    f = f + exp(-(x(1)-x(2))^2 - 2*x(1)^2)*cos(x(2))*sin(2*x(2));
end
```

Start at `x0 = [0.25,-0.25]` and search for a minimum of `objectivefcn`.

```
x0 = [0.25,-0.25];
x = fminsearch(@objectivefcn1,x0)
```

```
x =

   -0.1696   -0.5086
```

Minimize with Extra Parameters

Sometimes your objective function has extra parameters. These parameters are not variables to optimize, they are fixed values during the optimization. For example, suppose that you have a parameter `a` in the Rosenbrock-type function

$$f(x,a) = 100(x_2 - x_1^2)^2 + (a - x_1)^2 .$$

This function has a minimum value of 0 at $x_1 = a$, $x_2 = a^2$. If, for example, $a = 3$, you can include the parameter in your objective function by creating an anonymous function.

Create the objective function with its extra parameters as extra arguments.

```
f = @(x,a)100*(x(2) - x(1)^2)^2 + (a-x(1))^2;
```

Put the parameter in your MATLAB® workspace.

```
a = 3;
```

Create an anonymous function of `x` alone that includes the workspace value of the parameter.

```
fun = @(x)f(x,a);
```

Solve the problem starting at `x0 = [-1,1.9]`.

```
x0 = [-1,1.9];
x = fminsearch(fun,x0)
```

```
x = 1×2

    3.0000    9.0000
```

For more information about using extra parameters in your objective function, see "Parameterizing Functions".

Find Minimum Location and Value

Find both the location and value of a minimum of an objective function using `fminsearch`.

Write an anonymous objective function for a three-variable problem.

```
x0 = [1,2,3];
fun = @(x)-norm(x+x0)^2*exp(-norm(x-x0)^2 + sum(x));
```

Find the minimum of `fun` starting at `x0`. Find the value of the minimum as well.

```
[x,fval] = fminsearch(fun,x0)
```

```
x = 1×3

    1.5359    2.5645    3.5932
```

```
fval = -5.9565e+04
```

Inspect Optimization Process

Inspect the results of an optimization, both while it is running and after it finishes.

Set options to provide iterative display, which gives information on the optimization as the solver runs. Also, set a plot function to show the objective function value as the solver runs.

```
options = optimset('Display','iter','PlotFcns',@optimplotfval);
```

Set an objective function and start point.

```
function f = objectivefcn1(x)
f = 0;
for k = -10:10
    f = f + exp(-(x(1)-x(2))^2 - 2*x(1)^2)*cos(x(2))*sin(2*x(2));
end
```

Include the code for `objectivefcn1` as a file on your MATLAB® path.

```
x0 = [0.25,-0.25];
fun = @objectivefcn1;
```

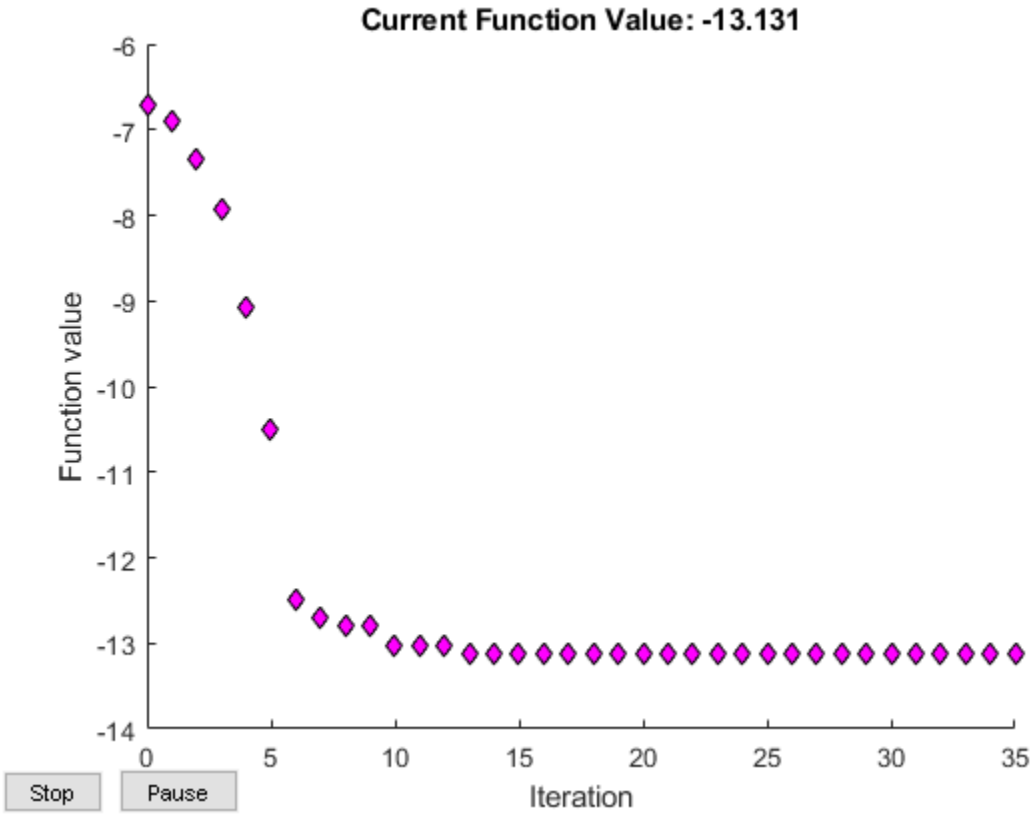
Obtain all solver outputs. Use these outputs to inspect the results after the solver finishes.

```
[x,fval,exitflag,output] = fminsearch(fun,x0,options)
```

```
 Iteration   Func-count     min f(x)         Procedure
     0            1         -6.70447
     1            3         -6.89837         initial simplex
     2            5         -7.34101         expand
     3            7         -7.91894         expand
     4            9         -9.07939         expand
     5           11         -10.5047         expand
     6           13         -12.4957         expand
     7           15         -12.6957         reflect
     8           17         -12.8052         contract outside
```

```
     9           19         -12.8052         contract inside
    10           21         -13.0189         expand
    11           23         -13.0189         contract inside
    12           25         -13.0374         reflect
    13           27          -13.122         reflect
    14           28          -13.122         reflect
    15           29          -13.122         reflect
    16           31          -13.122         contract outside
    17           33         -13.1279         contract inside
    18           35         -13.1279         contract inside
    19           37         -13.1296         contract inside
    20           39         -13.1301         contract inside
    21           41         -13.1305         reflect
    22           43         -13.1306         contract inside
    23           45         -13.1309         contract inside
    24           47         -13.1309         contract inside
    25           49          -13.131         reflect
    26           51          -13.131         contract inside
    27           53          -13.131         contract inside
    28           55          -13.131         contract inside
    29           57          -13.131         contract outside
    30           59          -13.131         contract inside
    31           61          -13.131         contract inside
    32           63          -13.131         contract inside
    33           65          -13.131         contract outside
    34           67          -13.131         contract inside
    35           69          -13.131         contract inside

Optimization terminated:
 the current x satisfies the termination criteria using OPTIONS.TolX of 1.000000e-04
 and F(X) satisfies the convergence criteria using OPTIONS.TolFun of 1.000000e-04


x =

   -0.1696   -0.5086


fval =

  -13.1310


exitflag =
```

```
     1


output =

  struct with fields:

    iterations: 35
     funcCount: 69
     algorithm: 'Nelder-Mead simplex direct search'
       message: 'Optimization terminated:...'
```



The value of `exitflag` is `1`, meaning `fminsearch` likely converged to a local minimum.

The `output` structure shows the number of iterations. The iterative display and the plot show this information as well. The `output` structure also shows the number of function evaluations, which the iterative display shows, but the chosen plot function does not.

Input Arguments

`fun` — Function to minimize
function handle | function name

Function to minimize, specified as a function handle or function name. `fun` is a function that accepts a vector or array `x` and returns a real scalar `f` (the objective function evaluated at `x`).

Specify `fun` as a function handle for a file:

```
x = fminsearch(@myfun,x0)
```

where `myfun` is a MATLAB function such as

```
function f = myfun(x)
f = ...            % Compute function value at x
```

You can also specify `fun` as a function handle for an anonymous function:

```
x = fminsearch(@(x)norm(x)^2,x0);
```

Example: `fun = @(x)-x*exp(-3*x)`

Data Types: `char` | `function_handle` | `string`

`x0` — Initial point
real vector | real array

Initial point, specified as a real vector or real array. Solvers use the number of elements in, and size of, `x0` to determine the number and size of variables that `fun` accepts.

Example: `x0 = [1,2,3,4]`

Data Types: `double`

`options` — Optimization options
structure such as `optimset` returns

Optimization options, specified as a structure such as `optimset` returns. You can use `optimset` to set or change the values of these fields in the options structure. See “Set Options” for detailed information.

`Display`	Level of display (see “Iterative Display”): • `'notify'` (default) displays output only if the function does not converge. • `'final'` displays just the final output. • `'off'` or `'none'` displays no output. • `'iter'` displays output at each iteration.
`FunValCheck`	Check whether objective function values are valid. `'on'` displays an error when the objective function returns a value that is `complex` or `NaN`. The default `'off'` displays no error.
`MaxFunEvals`	Maximum number of function evaluations allowed, a positive integer. The default is `200*numberOfVariables`. See “Tolerances and Stopping Criteria”.
`MaxIter`	Maximum number of iterations allowed, a positive integer. The default value is `200*numberOfVariables`. See “Tolerances and Stopping Criteria”.
`OutputFcn`	Specify one or more user-defined functions that an optimization function calls at each iteration, either as a function handle or as a cell array of function handles. The default is none (`[]`). See “Output Functions”.
`PlotFcns`	Plots various measures of progress while the algorithm executes. Select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none (`[]`): • `@optimplotx` plots the current point. • `@optimplotfunccount` plots the function count. • `@optimplotfval` plots the function value. For information on writing a custom plot function, see “Plot Functions”.

`TolFun`	Termination tolerance on the function value, a positive scalar. The default is `1e-4`. See “Tolerances and Stopping Criteria”. Unlike other solvers, `fminsearch` stops when it satisfies *both* `TolFun` and `TolX`.
`TolX`	Termination tolerance on `x`, a positive scalar. The default value is `1e-4`. See “Tolerances and Stopping Criteria”. Unlike other solvers, `fminsearch` stops when it satisfies *both* `TolFun` and `TolX`.

Example: `options = optimset('Display','iter')`

Data Types: `struct`

`problem` — Problem structure
structure

Problem structure, specified as a structure with the following fields.

Field Name	Entry
`objective`	Objective function
`x0`	Initial point for `x`
`solver`	`'fminsearch'`
`options`	Options structure such as returned by `optimset`

Data Types: `struct`

Output Arguments

`x` — Solution
real vector | real array

Solution, returned as a real vector or real array. The size of `x` is the same as the size of `x0`. Typically, `x` is a local solution to the problem when `exitflag` is positive.

`fval` — Objective function value at solution
real number

Objective function value at the solution, returned as a real number. Generally, `fval = fun(x)`.

`exitflag` — Reason `fminsearch` stopped
integer

Reason `fminsearch` stopped, returned as an integer.

`1`	The function converged to a solution `x`.
`0`	Number of iterations exceeded `options.MaxIter` or number of function evaluations exceeded `options.MaxFunEvals`.
`-1`	The algorithm was terminated by the output function.

`output` — Information about the optimization process
structure

Information about the optimization process, returned as a structure with fields:

`iterations`	Number of iterations
`funcCount`	Number of function evaluations
`algorithm`	`'Nelder-Mead simplex direct search'`
`message`	Exit message

Tips

- `fminsearch` only minimizes over the real numbers, that is, the vector or array x must only consist of real numbers and $f(x)$ must only return real numbers. When x has complex values, split x into real and imaginary parts.
- Use `fminsearch` to solve nondifferentiable problems or problems with discontinuities, particularly if no discontinuity occurs near the solution.

Algorithms

`fminsearch` uses the simplex search method of Lagarias et al. [1]. This is a direct search method that does not use numerical or analytic gradients as in `fminunc`. The algorithm is described in detail in “fminsearch Algorithm”. The algorithm is not guaranteed to converge to a local minimum.

References

[1] Lagarias, J. C., J. A. Reeds, M. H. Wright, and P. E. Wright. “Convergence Properties of the Nelder-Mead Simplex Method in Low Dimensions.” *SIAM Journal of Optimization*. Vol. 9, Number 1, 1998, pp. 112–147.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For C/C++ code generation:

- `fminsearch` ignores the `Display` option and does not give iterative display or an exit message. To check solution quality, examine the exit flag.
- The output structure does not include the `algorithm` or `message` fields.
- `fminsearch` ignores the `OutputFcn` and `PlotFcns` options.

See Also

`fminbnd` | `optimset`

Topics

“Minimizing Functions of Several Variables”
“Curve Fitting via Optimization”
“Create Function Handle”
“Anonymous Functions”

Introduced before R2006a

matlab.fonts Settings

Code font settings

The default font that MATLAB uses for a particular tool depends upon its content. You can change the default font using the `matlab.fonts` settings. Access `matlab.fonts` using the root `SettingsGroup` object returned by the `settings` function. For example, set the temporary value for the code font name to `'Times New Roman'` (where the default is `'Monospaced'`).

```
s = settings;
s.matlab.fonts.codefont.Name.TemporaryValue = 'Times New Roman'
```

For more information about settings, see “Access and Modify Settings”.

Settings

matlab.fonts.codefont

`Name` — Code font name
`'Monospaced'` (default) | character vector

Code font name, specified as a character vector.

Example: `s.matlab.fonts.codefont.Name.TemporaryValue = 'Arial'`

`Style` — Code font style
`0` (default) | `1` | `2` | `3`

Code font style, specified as the positive integer `0`, `1`, `2`, or `3`.

The values are define as follows:

- `0` — Plain
- `1` — Bold
- `2` — Italic
- `3` — Bold Italic

Example: `s.matlab.fonts.codefont.Style.TemporaryValue = 1`

`Size` — Code font size

10 (default) | positive integer

Code font size, specified as a positive integer.

Example: `s.matlab.fonts.codefont.Size.TemporaryValue = 12`

matlab.fonts.editor.codefont

`Size` — App Designer code font size

14 (default) | positive integer

App designer code font size that displays in App Designer **Code View**, specified as a positive integer between 14 and 48.

Example: `s.matlab.fonts.editor.codefont.Size.TemporaryValue = 20`

See Also

`settings`

Topics

"Access and Modify Settings"
"Change Fonts"

Introduced in R2018a

fopen

Open file, or obtain information about open files

Syntax

```
fileID = fopen(filename)
fileID = fopen(filename,permission)
fileID = fopen(filename,permission,machinefmt,encodingIn)
[fileID,errmsg] = fopen( ___ )

fIDs = fopen('all')

filename = fopen(fileID)
[filename,permission,machinefmt,encodingOut] = fopen(fileID)
```

Description

`fileID = fopen(filename)` opens the file, `filename`, for binary read access, and returns an integer file identifier equal to or greater than 3. MATLAB reserves file identifiers `0`, `1`, and `2` for standard input, standard output (the screen), and standard error, respectively.

If `fopen` cannot open the file, then `fileID` is `-1`.

`fileID = fopen(filename,permission)` opens the file with the type of access specified by `permission`.

`fileID = fopen(filename,permission,machinefmt,encodingIn)` additionally specifies the order for reading or writing bytes or bits in the file using the `machinefmt` argument. The optional `encodingIn` argument specifies the character encoding scheme associated with the file.

`[fileID,errmsg] = fopen( ___ )` additionally returns a system-dependent error message if `fopen` fails to open the file. Otherwise, `errmsg` is an empty character vector. You can use this syntax with any of the input arguments of the previous syntaxes.

`fIDs = fopen('all')` returns a row vector containing the file identifiers of all open files. The identifiers reserved for standard input, output, and error are not included. The number of elements in the vector is equal to the number of open files.

`filename = fopen(fileID)` returns the file name that a previous call to `fopen` used when it opened the file specified by `fileID`. The output filename is resolved to the full path. The `fopen` function does not read information from the file to determine the output value.

`[filename,permission,machinefmt,encodingOut] = fopen(fileID)` additionally returns the permission, machine format, and encoding that a previous call to `fopen` used when it opened the specified file. If the file was opened in binary mode, `permission` includes the letter `'b'`. The `encodingOut` output is a standard encoding scheme name. `fopen` does not read information from the file to determine these output values. An invalid `fileID` returns empty character vectors for all output arguments.

Examples

Open File and Pass Identifier to File I/O Function

Open a file and pass the file identifier to the `fgetl` function to read data.

Open the file, `tsunamis.txt`, and obtain the file identifier.

```
fileID = fopen('tsunamis.txt');
```

Pass the `fileID` to the `fgetl` function to read one line from the file.

```
tline = fgetl(fileID)
```

```
tline =

  'A global tsunami data set in xlsx format, comprising the following file:'
```

Close the file.

```
fclose(fileID)
```

Request Name of File to Open

Create a prompt to request the name of a file to open. If `fopen` cannot open the file, display the relevant error message.

```
fileID = -1;
errmsg = '';
while fileID < 0
   disp(errmsg);
   filename = input('Open file: ', 's');
   [fileID,errmsg] = fopen(filename);
end
```

Open File for Writing and Specify Access Type, Writing Order, Character Encoding

Open a file to write to a file using the Shift-JIS character encoding.

```
fileID = fopen('japanese_out.txt','w','n','Shift_JIS');
```

The `'w'` input specifies write access, the `'n'` input specifies native byte ordering, and `'Shift_JIS'` specifies the character encoding scheme.

Get Information About Open Files

Suppose you previously opened a file using `fopen`.

```
fileID = fopen('tsunamis.txt');
```

Get the file identifiers of all open files.

```
fIDs = fopen('all')
```

```
fIDs =

     3
```

Get the file name and character encoding for the open file. Use ~ in place of output arguments you want to omit.

```
[filename,~,~,encoding] = fopen(fileID)
```

```
filename =

    'matlabroot\toolbox\matlab\demos\tsunamis.txt'


encoding =

    'windows-1252'
```

The output shown here is representative. Your results might differ.

Input Arguments

`filename` — Name of file to open
character vector or string scalar

Name of the file to open, including the file extension, specified as a character row vector or a string scalar. If the file is not in the current folder, `filename` must include a full or a relative path.

On UNIX systems, if `filename` begins with `'~/'` or `'~username/'`, the `fopen` function expands the path to the current or specified user's home directory, respectively.

- If you open a file with read access and the file is not in the current folder, then `fopen` searches along the MATLAB search path.
- If you open a file with write or append access and the file is not in the current folder, then `fopen` creates a file in the current directory.

Example: `'myFile.txt'`

Data Types: `char` | `string`

`permission` — File access type
`'r'` (default) | `'w'` | `'a'` | `'r+'` | `'w+'` | `'a+'` | `'A'` | `'W'` | ...

File access type, specified as a character vector or a string scalar. You can open a file in binary mode or in text mode. On UNIX systems, both translation modes have the same effect. To open a file in binary mode, specify one of the following.

`'r'`	Open file for reading.

`'w'`	Open or create new file for writing. Discard existing contents, if any.
`'a'`	Open or create new file for writing. Append data to the end of the file.
`'r+'`	Open file for reading and writing.
`'w+'`	Open or create new file for reading and writing. Discard existing contents, if any.
`'a+'`	Open or create new file for reading and writing. Append data to the end of the file.
`'A'`	Open file for appending without automatic flushing of the current output buffer.
`'W'`	Open file for writing without automatic flushing of the current output buffer.

To open files in text mode, attach the letter `'t'` to the `permission` argument, such as `'rt'` or `'wt+'`.

On Windows systems, in text mode:

- Read operations that encounter a carriage return followed by a newline character (`'\r\n'`) remove the carriage return from the input.
- Write operations insert a carriage return before any newline character in the output.

Open or create a new file in text mode if you want to write to it in MATLAB and then open it in Microsoft Notepad, or any text editor that does not recognize `'\n'` as a newline sequence. When writing to the file, end each line with `'\r\n'`. For an example, see `fprintf`. Otherwise, open files in binary mode for better performance.

To read and write to the same file:

- Open the file with a value for `permission` that includes a plus sign, `'+'`.
- Call `fseek` or `frewind` between read and write operations. For example, do not call `fread` followed by `fwrite`, or `fwrite` followed by `fread`, unless you call `fseek` or `frewind` between them.

Data Types: `char` | `string`

`machinefmt` — Order for reading or writing bytes or bits
`'n'` (default) | `'b'` | `'l'` | `'s'` | `'a'` | ...

Order for reading or writing bytes or bits in the file, specified as one of the following character vectors or string scalars.

`'n'` or `'native'`	Your system byte ordering (default)
`'b'` or `'ieee-be'`	Big-endian ordering
`'l'` or `'ieee-le'`	Little-endian ordering
`'s'` or `'ieee-be.l64'`	Big-endian ordering, 64-bit long data type
`'a'` or `'ieee-le.l64'`	Little-endian ordering, 64-bit long data type

By default, all currently supported platforms use little-endian ordering for new files. Existing binary files can use either big-endian or little-endian ordering.

Data Types: `char` | `string`

`encodingIn` — Character encoding
`'UTF-8'` | `'ISO-8859-1'` | `'windows-1251'` | `'windows-1252'` | ...

Character encoding to use for subsequent read and write operations, including `fscanf`, `fprintf`, `fgetl`, `fgets`, `fread`, and `fwrite`, specified as a character vector or a string scalar. The character vector or string scalar must contain a standard character encoding scheme name such as the following.

`'Big5'`	`'ISO-8859-1'`	`'windows-874'`
`'Big5-HKSCS'`	`'ISO-8859-2'`	`'windows-949'`
`'CP949'`	`'ISO-8859-3'`	`'windows-1250'`
`'EUC-KR'`	`'ISO-8859-4'`	`'windows-1251'`
`'EUC-JP'`	`'ISO-8859-5'`	`'windows-1252'`
`'EUC-TW'`	`'ISO-8859-6'`	`'windows-1253'`
`'GB18030'`	`'ISO-8859-7'`	`'windows-1254'`
`'GB2312'`	`'ISO-8859-8'`	`'windows-1255'`
`'GBK'`	`'ISO-8859-9'`	`'windows-1256'`
`'IBM866'`	`'ISO-8859-11'`	`'windows-1257'`
`'KOI8-R'`	`'ISO-8859-13'`	`'windows-1258'`
`'KOI8-U'`	`'ISO-8859-15'`	`'US-ASCII'`

	'Macintosh'	'UTF-8'
	'Shift_JIS'	

If you do not specify an encoding scheme, `fopen` opens files for processing using the default encoding for your system. For more information, see “Opening Files with Different Character Encodings”.

If you specify a value for encoding that is not in the list of supported values, MATLAB issues a warning. Specifying other encoding names sometimes (but not always) produces correct results.

Data Types: `char` | `string`

`fileID` — File identifier of an open file
integer

File identifier of an open file, specified as an integer.

Data Types: `double`

Tips

- In most cases, it is not necessary to open a file in text mode. MATLAB import functions, all UNIX applications, and Microsoft Word and WordPad recognize `'\n'` as a newline indicator.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support:
  - The input arguments `machinefmt`, `encodingIn`, or `fileID`.

  - The output argument `errmsg`.
  - The syntax `fopen('all')`.
  - Opening a file in text mode. That is the `permission` argument must not contain the letter `t`. For example, the value cannot be `'rt'`.
- The `permission` argument can contain three characters at most. The characters must be unique.
- If you disable extrinsic calls, then you cannot return file identifiers created with `fopen` to MATLAB functions or extrinsic functions. Use these file identifiers only internally.
- When generating C/C++ executables, static libraries, or dynamic libraries, you can open up to 20 files.
- The generated code does not report errors from invalid file identifiers. Write your own file open error handling in your MATLAB code. Test whether `fopen` returns `-1`, which indicates that the file open failed. For example:

```
...
fid = fopen(filename, 'r');
if fid == -1
        % fopen failed

else
% fopen successful, okay to call fread
A = fread(fid);
...
```

- The behavior of the generated code for `fread` is compiler-dependent when you:

  1. Open a file using `fopen` with a `permission` of `a+`.
  2. Read the file using `fread` before calling an I/O function, such as `fseek` or `frewind`, that sets the file position indicator.

See Also

`fclose` | `feof` | `ferror` | `fprintf` | `fread` | `frewind` | `fscanf` | `fseek` | `ftell` | `fwrite`

Introduced before R2006a

fopen (serial)

(Not recommended) Connect serial port object to device

Note This `serial` object function is not recommended. Use `serialport` object functions instead. See “Compatibility Considerations”.

Syntax

`fopen(obj)`

Description

`fopen(obj)` connects the serial port object, `obj` to the device.

Examples

This example creates the serial port object `s`, connects `s` to the device using `fopen`, writes and reads text data, and then disconnects `s` from the device. This example works on a Windows platform.

```
s = serial('COM1');
fopen(s)
fprintf(s,'*IDN?')
idn = fscanf(s);
fclose(s)
```

Tips

Before you can perform a read or write operation, `obj` must be connected to the device with the `fopen` function. When `obj` is connected to the device:

- Data remaining in the input buffer or the output buffer is flushed.

- The `Status` property is set to `open`.
- The `BytesAvailable`, `ValuesReceived`, `ValuesSent`, and `BytesToOutput` properties are set to 0.

An error is returned if you attempt to perform a read or write operation while `obj` is not connected to the device. You can connect only one serial port object to a given device.

Some properties are read-only while the serial port object is open (connected), and must be configured before using `fopen`. Examples include `InputBufferSize` and `OutputBufferSize`. Refer to the property reference pages to determine which properties have this constraint.

The values for some properties are verified only after `obj` is connected to the device. If any of these properties are incorrectly configured, then an error is returned when `fopen` is issued and `obj` is not connected to the device. Properties of this type include `BaudRate`, and are associated with device settings.

Compatibility Considerations

`serial` object interface is not recommended

Not recommended starting in R2019b

Use of this function with a `serial` object is not recommended. To access a serial port device, use a `serialport` object with its functions and properties instead.

See “Transition Your Code to serialport Interface” for more information about using the recommended functionality.

See Also

Functions

`fclose`

Introduced before R2006a

for

`for` loop to repeat specified number of times

Syntax

```
for index = values
   statements
end
```

Description

`for index = values, statements, end` executes a group of statements in a loop for a specified number of times. *`values`* has one of the following forms:

- *`initVal`*:*`endVal`* — Increment the *`index`* variable from *`initVal`* to *`endVal`* by `1`, and repeat execution of *`statements`* until *`index`* is greater than *`endVal`*.
- *`initVal`*:*`step`*:*`endVal`* — Increment *`index`* by the value *`step`* on each iteration, or decrements *`index`* when *`step`* is negative.
- *`valArray`* — Create a column vector, *`index`*, from subsequent columns of array *`valArray`* on each iteration. For example, on the first iteration, *`index = valArray(:,1)`*. The loop executes a maximum of *`n`* times, where *`n`* is the number of columns of *`valArray`*, given by `numel(valArray(1,:))`. The input *`valArray`* can be of any MATLAB data type, including a character vector, cell array, or struct.

Examples

Assign Matrix Values

Create a Hilbert matrix of order 10.

```
s = 10;
H = zeros(s);
```

```
for c = 1:s
    for r = 1:s
        H(r,c) = 1/(r+c-1);
    end
end
```

Decrement Values

Step by increments of `-0.2`, and display the values.

```
for v = 1.0:-0.2:0.0
   disp(v)
end
```

```
     1

    0.8000

    0.6000

    0.4000

    0.2000

     0
```

Execute Statements for Specified Values

```
for v = [1 5 8 17]
   disp(v)
end
```

```
     1

     5

     8

    17
```

Repeat Statements for Each Matrix Column

```
for I = eye(4,3)
    disp('Current unit vector:')
    disp(I)
end
```

```
Current unit vector:
     1
     0
     0
     0

Current unit vector:
     0
     1
     0
     0

Current unit vector:
     0
     0
     1
     0
```

Tips

- To programmatically exit the loop, use a `break` statement. To skip the rest of the instructions in the loop and begin the next iteration, use a `continue` statement.
- Avoid assigning a value to the *`index`* variable within the loop statements. The `for` statement overrides any changes made to *`index`* within the loop.
- To iterate over the values of a single column vector, first transpose it to create a row vector.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Suppose that the loop end value is equal to or close to the maximum or minimum value for the loop index data type. In the generated code, the last increment or decrement of the loop index might cause the index variable to overflow. The index overflow might result in an infinite loop. See “Loop Index Overflow” (MATLAB Coder).

See Also

`break` | `colon` | `continue` | `end` | `if` | `parfor` | `return` | `switch`

Introduced before R2006a

format

Set Command Window output display format

Syntax

```
format style
format
```

Description

`format style` changes the output display format in the Command Window to the format specified by `style`.

`format`, by itself, resets the output format to the default, which is the short, fixed-decimal format for floating-point notation and loose line spacing for all output lines.

Numeric formats affect only how numbers appear in Command Window output, not how MATLAB computes or saves them.

Examples

Long Format

Set the output format to the long fixed-decimal format and display the value of `pi`.

```
format long
pi
```

```
ans =
   3.141592653589793
```

Reset Format to Default

Set the output format to the short engineering format with compact line spacing, and then reset the format to the default.

```
format shortEng
format compact
x = rand(3)
```

```
x = 3×3

   814.7237e-003   913.3759e-003   278.4982e-003
   905.7919e-003   632.3592e-003   546.8815e-003
   126.9868e-003    97.5404e-003   957.5068e-003
```

```
format
x
```

```
x = 3×3

    0.8147    0.9134    0.2785
    0.9058    0.6324    0.5469
    0.1270    0.0975    0.9575
```

Hexadecimal Format

Display the maximum values for integers and real numbers in hexadecimal format.

```
format hex
intmax('uint64')
```

```
ans = uint64
   ffffffffffffffff
```

```
realmax
```

```
ans =
   7fefffffffffffff
```

Short and Long Engineering Notation

Display the difference between `shortEng` and `longEng` formats.

Set the output format to `shortEng`.

```
format shortEng
```

Create a variable and increase its value by a multiple of 10 each time through a `for` loop.

```
A = 5.123456789;
for k = 1:10
   disp(A)
   A = A*10;
end
```

```
     5.1235e+000

    51.2346e+000

   512.3457e+000

     5.1235e+003

    51.2346e+003

   512.3457e+003

     5.1235e+006

    51.2346e+006

   512.3457e+006

     5.1235e+009
```

The values display with 4 digits after the decimal point and an exponent that is a multiple of 3.

Set the output format to the long engineering format and view the same values.

```
format longEng
```

```
A = 5.123456789;
for k = 1:10
   disp(A)
   A = A*10;
end
```

```
    5.12345678900000e+000

    51.2345678900000e+000

    512.345678900000e+000

    5.12345678900000e+003

    51.2345678900000e+003

    512.345678900000e+003

    5.12345678900000e+006

    51.2345678900000e+006

    512.345678900000e+006

    5.12345678900000e+009
```

The values display with 15 digits and an exponent that is a multiple of 3.

Large Data Range Format

Use the `shortG` format when some of the values in an array are short numbers and some have large exponents. The `shortG` format picks whichever short fixed-decimal format or short scientific notation has the most compact display.

Create a variable and display output in the `short` format, which is the default.

```
x = [25 56.31156 255.52675 9876899999];
format short
x
```

```
x = 1×4
10^9 ×
```

```
    0.0000    0.0000    0.0000    9.8769
```

Set the format to `shortG` and redisplay the values.

```
format shortG
x
```

```
x = 1×4

           25       56.312       255.53   9.8769e+09
```

Get Current Format

Get the current numeric format.

```
f = get(0,'Format')
```

```
f =

shortG
```

Get the current line spacing, which can be set to `loose` or `compact`.

```
S = get(0,'FormatSpacing')
```

```
S =

loose
```

Input Arguments

`style` — Output display format
`short` (default) | `long` | `shortE` | `longE` | ...

Output display format, specified as one of these options.

Numeric Format

These styles control the output display format for numeric variables.

Style	Result	Example
`short (default)`	Short, fixed-decimal format with 4 digits after the decimal point.	`3.1416`
`long`	Long, fixed-decimal format with 15 digits after the decimal point for `double` values, and 7 digits after the decimal point for `single` values.	`3.141592653589793`
`shortE`	Short scientific notation with 4 digits after the decimal point.	`3.1416e+00`
`longE`	Long scientific notation with 15 digits after the decimal point for `double` values, and 7 digits after the decimal point for `single` values.	`3.141592653589793e+00`
`shortG`	Short, fixed-decimal format or scientific notation, whichever is more compact, with a total of 5 digits.	`3.1416`
`longG`	Long, fixed-decimal format or scientific notation, whichever is more compact, with a total of 15 digits for `double` values, and 7 digits for `single` values.	`3.14159265358979`
`shortEng`	Short engineering notation (exponent is a multiple of 3) with 4 digits after the decimal point.	`3.1416e+000`
`longEng`	Long engineering notation (exponent is a multiple of 3) with 15 significant digits.	`3.14159265358979e+000`
`+`	Positive/Negative format with `+`, `-`, and blank characters displayed for positive, negative, and zero elements.	`+`
`bank`	Currency format with 2 digits after the decimal point.	`3.14`
`hex`	Hexadecimal representation of a binary double-precision number.	`400921fb54442d18`
`rat`	Ratio of small integers.	`355/113`

Line Spacing Format

Style	Result	Example

`compact`	Suppress excess blank lines to show more output on a single screen.	`theta = pi/2`
		`theta =`
		`  1.5708`

`loose`	Add blank lines to make output more readable.	`theta = pi/2`
		`theta =`
		`  1.5708`

Tips

- The specified format applies only to the current MATLAB session. To maintain a format across sessions, choose a **Numeric format** or **Numeric display** option in the Command Window Preferences.
- You can insert a space between `short` or `long` and the presentation type, for instance, `format short E`.
- MATLAB always displays integer data types to the appropriate number of digits for the data type. For example, MATLAB uses 3 digits to display `int8` data types (for instance, -128:127). Setting the output format to `short` or `long` does not affect the display of integer-type variables.
- Integer-valued, floating-point numbers with a maximum of 9 digits do not display in scientific notation.
- If you are displaying a matrix with a wide range of values, consider using `shortG`. See “Large Data Range Format” on page 1-4558.

See Also

`disp` | `fprintf` | `rat`

Topics

“Format Output”

Introduced before R2006a

fplot

Plot expression or function

Syntax

```
fplot(f)
fplot(f,xinterval)

fplot(funx,funy)
fplot(funx,funy,tinterval)

fplot( ___ ,LineSpec)
fplot( ___ ,Name,Value)
fplot(ax, ___ )
fp = fplot( ___ )
[x,y] = fplot( ___ )
```

Description

`fplot(f)` plots the curve defined by the function `y = f(x)` over the default interval `[-5 5]` for `x`.

`fplot(f,xinterval)` plots over the specified interval. Specify the interval as a two-element vector of the form `[xmin xmax]`.

`fplot(funx,funy)` plots the curve defined by `x = funx(t)` and `y = funy(t)` over the default interval `[-5 5]` for `t`.

`fplot(funx,funy,tinterval)` plots over the specified interval. Specify the interval as a two-element vector of the form `[tmin tmax]`.

`fplot( ___ ,LineSpec)` specifies the line style, marker symbol, and line color. For example, `'-r'` plots a red line. Use this option after any of the input argument combinations in the previous syntaxes.

`fplot( ___ ,Name,Value)` specifies line properties using one or more name-value pair arguments. For example, `'LineWidth',2` specifies a line width of 2 points.

`fplot(ax, ___ )` plots into the axes specified by`ax` instead of the current axes (`gca`). Specify the axes as the first input argument.

`fp = fplot( ___ )` returns a `FunctionLine` object or a `ParameterizedFunctionLine` object, depending on the inputs. Use `fp` to query and modify properties of a specific line. For a list of properties, see FunctionLine or ParameterizedFunctionLine.

`[x,y] = fplot( ___ )` returns the abscissas and ordinates for the function without creating a plot. This syntax will be removed in a future release. Use the `XData` and `YData` properties of the line object, `fp`, instead.

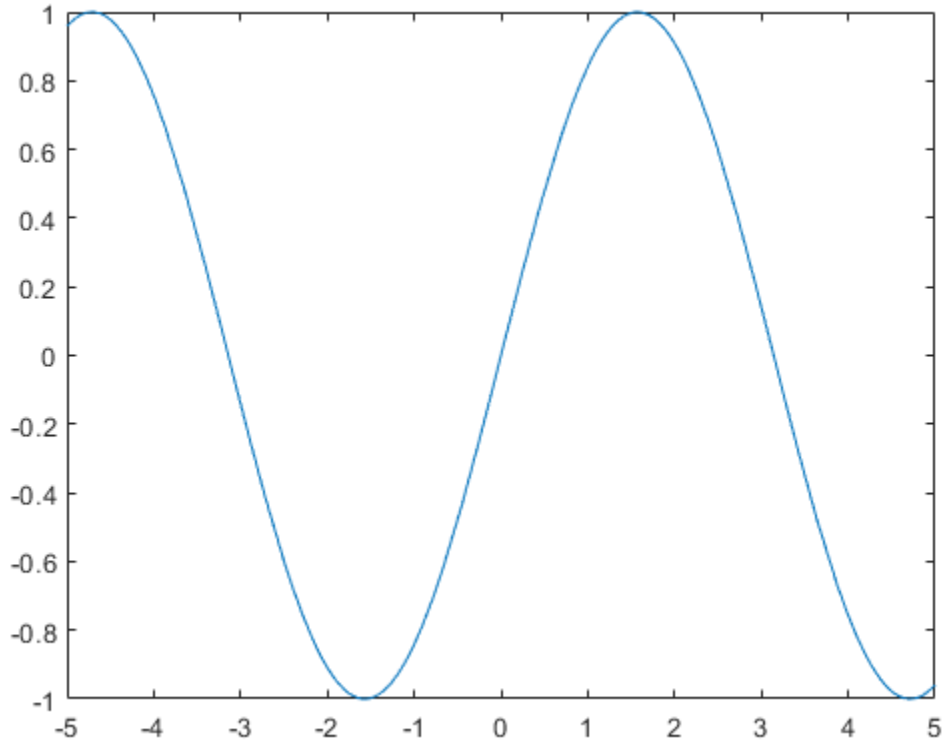
Note `fplot` no longer supports input arguments for specifying the error tolerance or the number of evaluation points. To specify the number of evaluation points, use the `MeshDensity` property.

Examples

Plot Expression

Plot `sin(x)` over the default `x` interval `[-5 5]`.

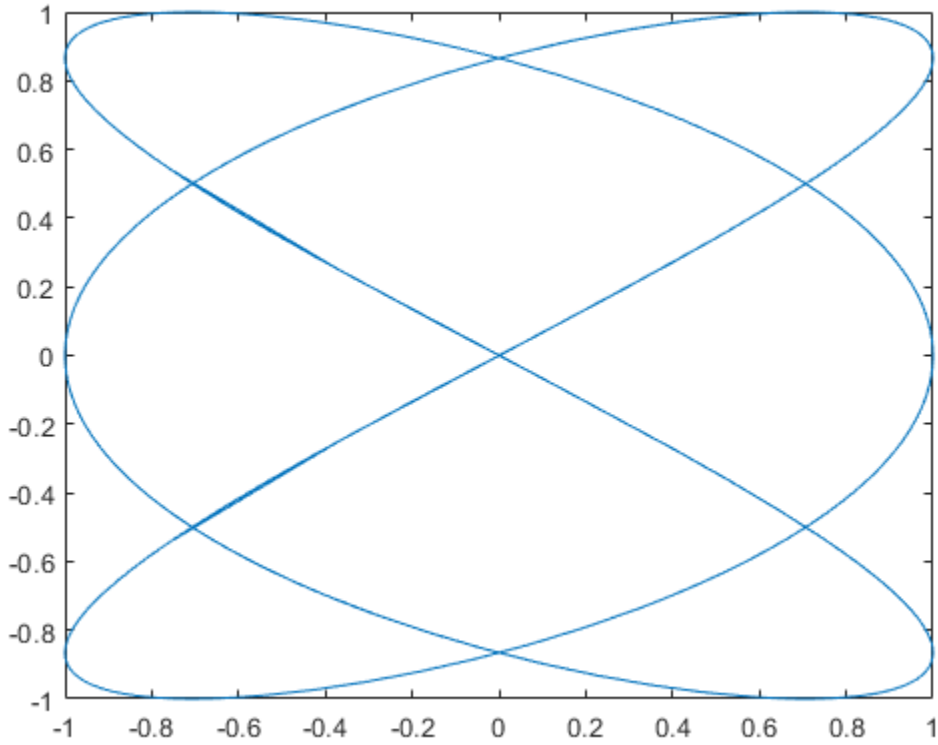
```
fplot(@(x) sin(x))
```



Plot Parametric Curve

Plot the parametric curve $x = cos(3t)$ and $y = sin(2t)$.

```
xt = @(t) cos(3*t);
yt = @(t) sin(2*t);
fplot(xt,yt)
```



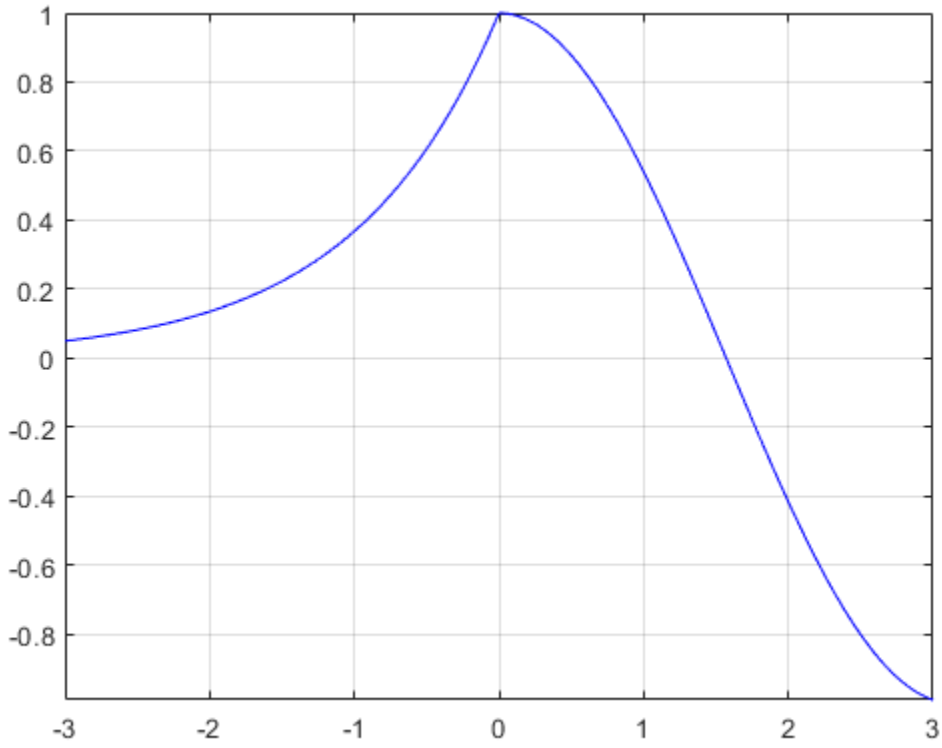
Specify Plotting Interval and Plot Piecewise Functions

Plot the piecewise function

$$\begin{matrix} e^x & -3 < x < 0 \\ \cos(x) & 0 < x < 3 \end{matrix}.$$

Plot multiple lines using `hold on`. Specify the plotting intervals using the second input argument of `fplot`. Specify the color of the plotted lines as blue using `'b'`. When you plot multiple lines in the same axes, the axis limits adjust to incorporate all the data.

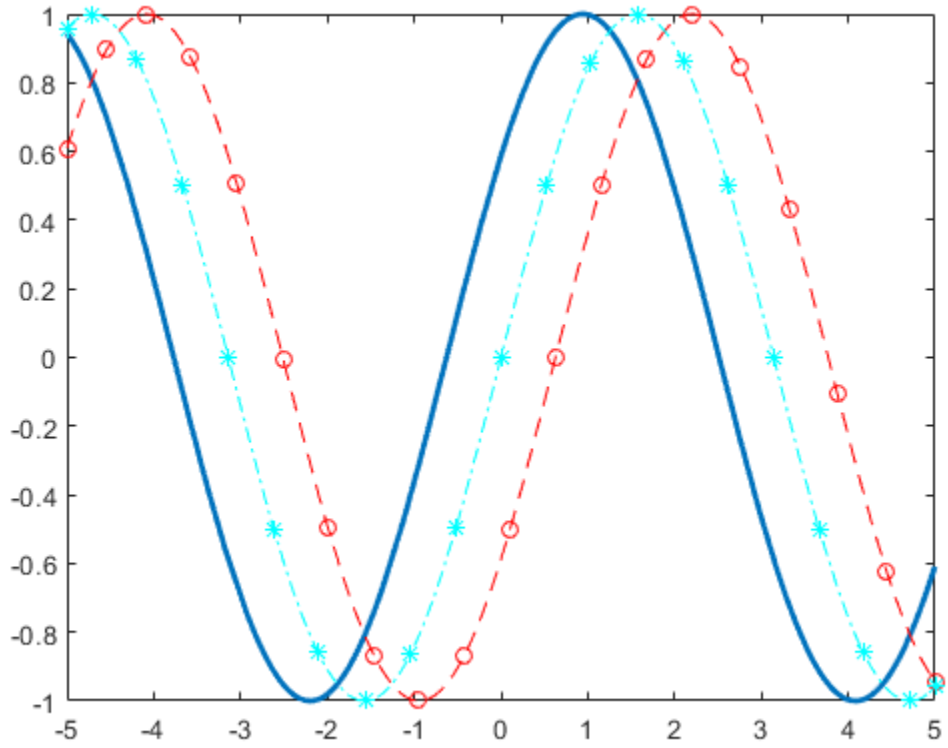
```
fplot(@(x) exp(x),[-3 0],'b')
hold on
fplot(@(x) cos(x),[0 3],'b')
hold off
grid on
```



Specify Line Properties and Display Markers

Plot three sine waves with different phases. For the first, use a line width of 2 points. For the second, specify a dashed red line style with circle markers. For the third, specify a cyan, dash-dotted line style with asterisk markers.

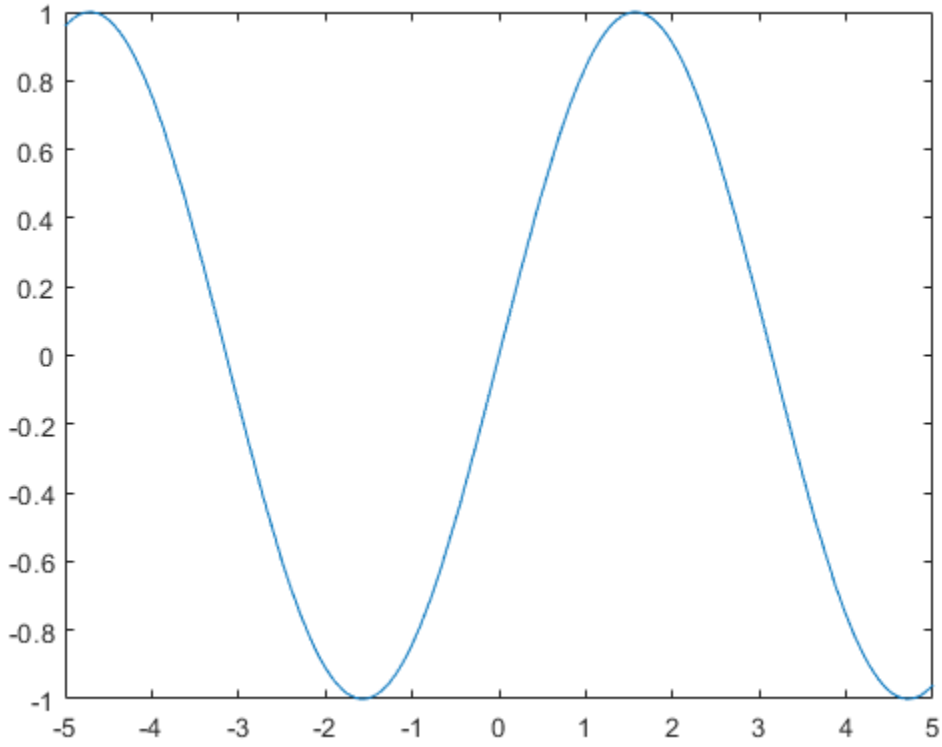
```
fplot(@(x) sin(x+pi/5),'Linewidth',2);
hold on
fplot(@(x) sin(x-pi/5),'--or');
fplot(@(x) sin(x),'-.*c')
hold off
```



Modify Line Properties After Creation

Plot `sin(x)` and assign the function line object to a variable.

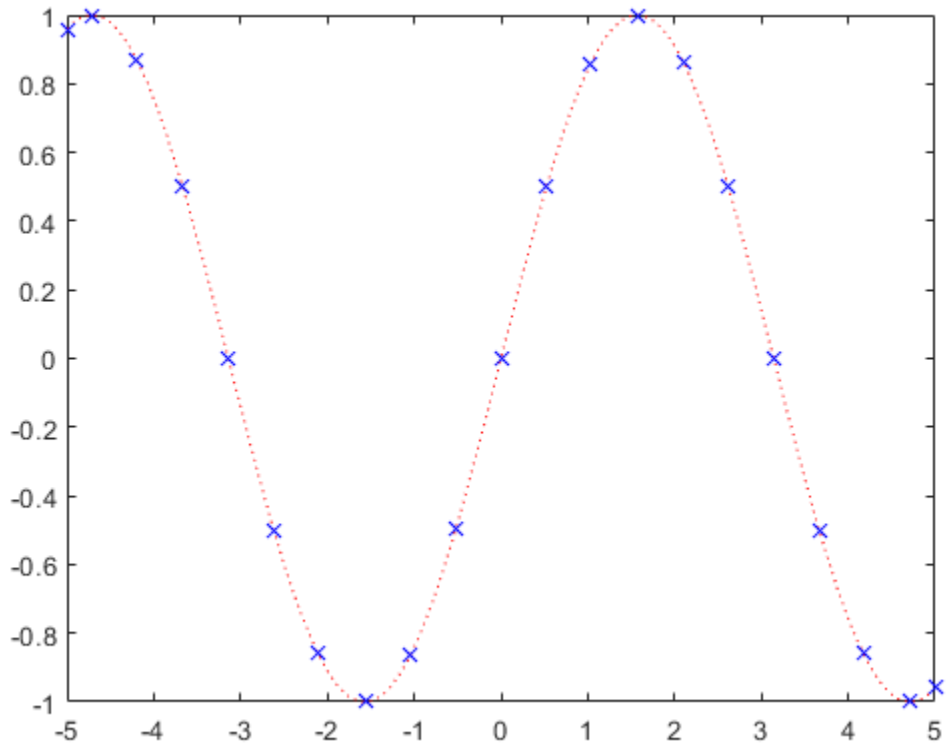
```
fp = fplot(@(x) sin(x))
```




```
fp =
  FunctionLine with properties:

     Function: @(x)sin(x)
        Color: [0 0.4470 0.7410]
    LineStyle: '-'
    LineWidth: 0.5000

  Show all properties
```

Change the line to a dotted red line by using dot notation to set properties. Add cross markers and set the marker color to blue.

```
fp.LineStyle = ':';
fp.Color = 'r';
fp.Marker = 'x';
fp.MarkerEdgeColor = 'b';
```

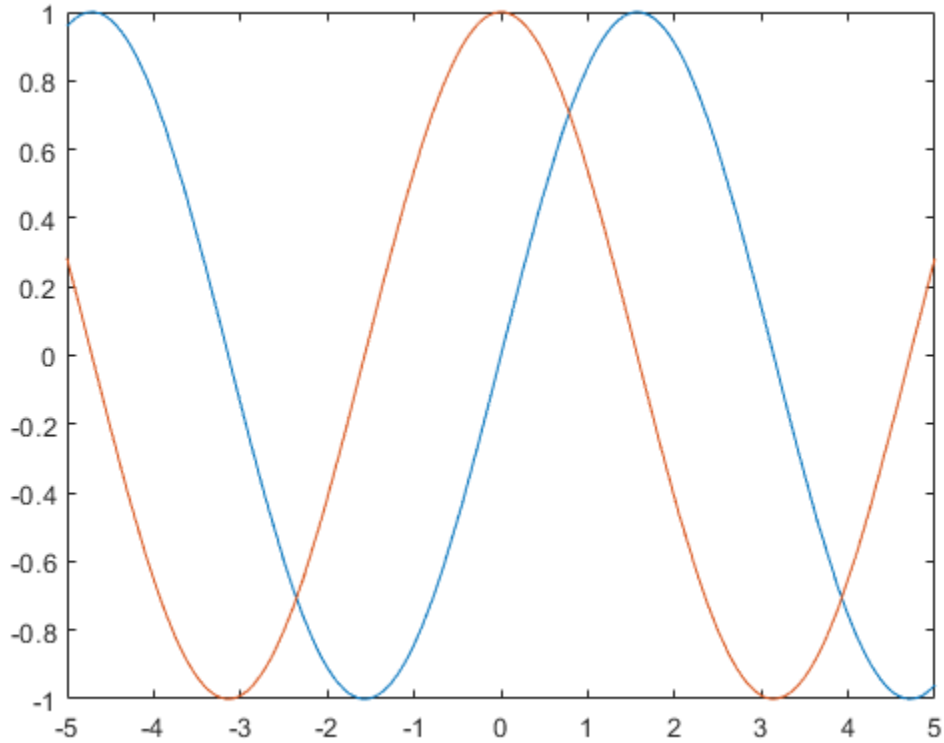


Plot Multiple Lines in Same Axes

Plot two lines using `hold on`.

```
fplot(@(x) sin(x))
hold on
```

```
fplot(@(x) cos(x))
hold off
```

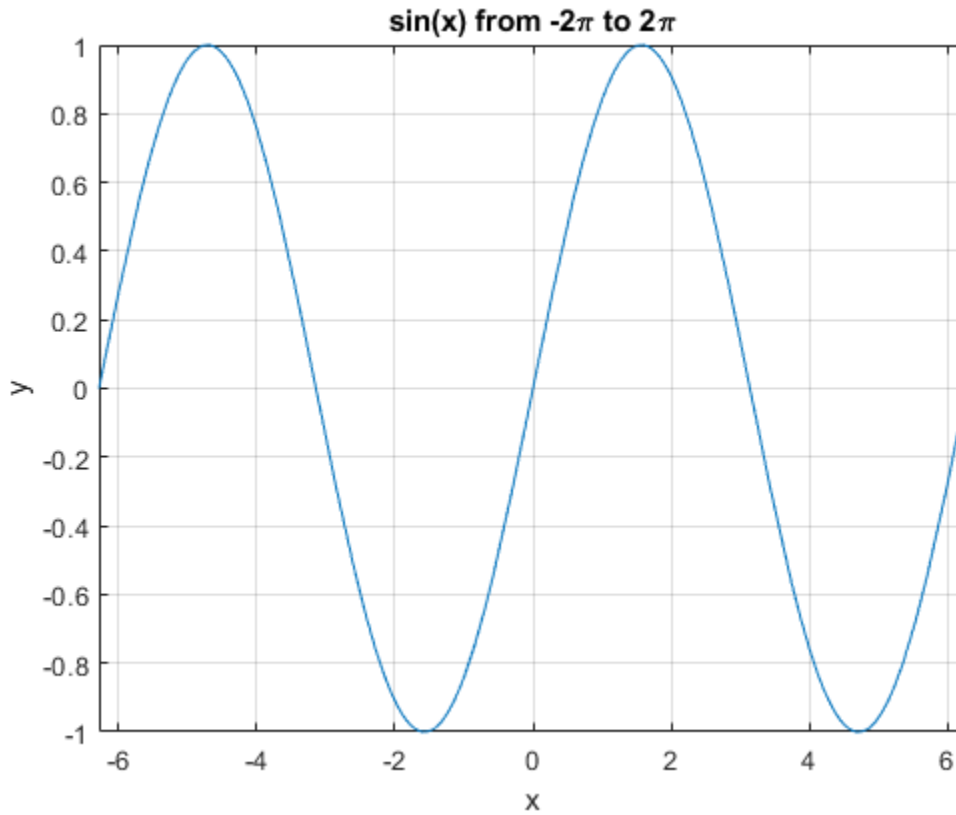


Add Title and Axis Labels and Format Ticks

Plot sin(x) from -2π to 2π using a function handle. Display the grid lines. Then, add a title and label the x-axis and y-axis.

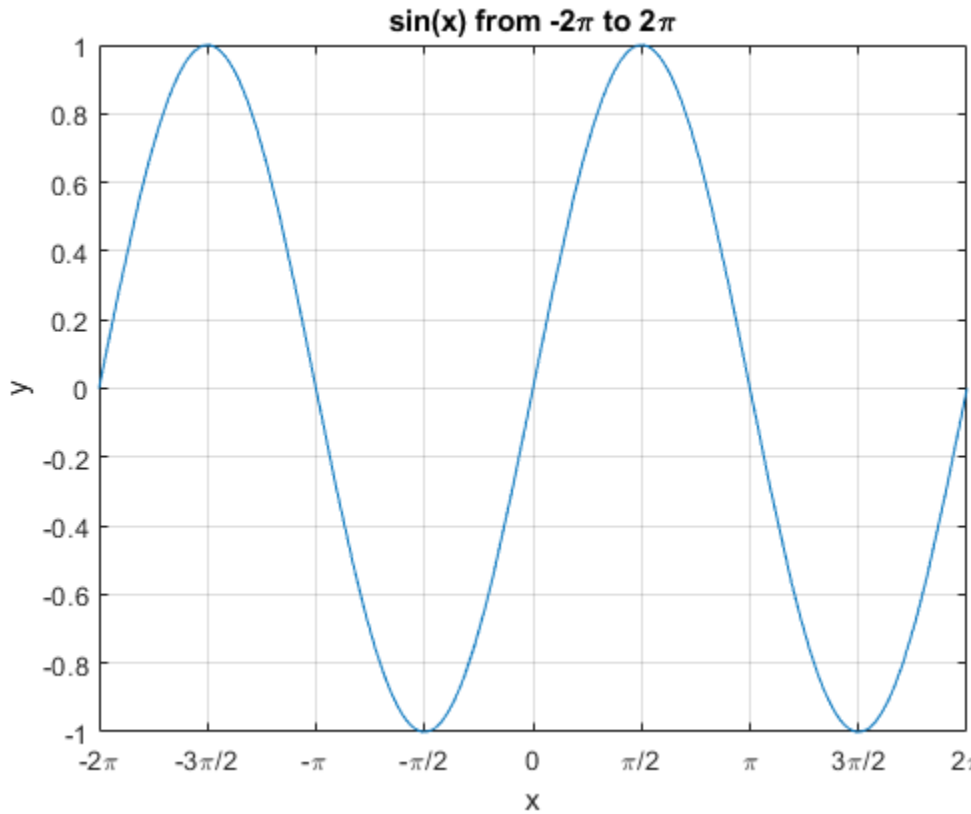
```
fplot(@sin,[-2*pi 2*pi])
grid on
title('sin(x) from -2\pi to 2\pi')
```

```
xlabel('x');
ylabel('y');
```



Use `gca` to access the current axes object. Display tick marks along the *x*-axis at intervals of $\pi/2$. Format the *x*-axis tick values by setting the `XTick` and `XTickLabel` properties of the axes object. Similar properties exist for the *y*-axis.

```
ax = gca;
ax.XTick = -2*pi:pi/2:2*pi;
ax.XTickLabel = {'-2\pi','-3\pi/2','-\pi','-\pi/2','0','\pi/2','\pi','3\pi/2','2\pi'};
```



Input Arguments

f — Function to plot
function handle

Function to plot, specified as a function handle to a named or anonymous function.

Specify a function of the form `y = f(x)`. The function must accept a vector input argument and return a vector output argument of the same size. Use array operators instead of matrix operators for the best performance. For example, use `.*` (`times`) instead of `*` (`mtimes`).

Note Support for character vector inputs will be removed in a future release. Use function handles instead.

Example: `f = @(x) sin(x);`

`xinterval` — Interval for `x`
`[-5 5]` (default) | two-element vector of form `[xmin xmax]`

Interval for `x`, specified as a two-element vector of the form `[xmin xmax]`.

`funx` — Parametric function for *x* coordinates
function handle

Parametric function for *x* coordinates, specified as a function handle to a named or anonymous function.

Specify a function of the form `x = funx(t)`. The function must accept a vector input argument and return a vector output argument of the same size. Use array operators instead of matrix operators for the best performance. For example, use `.*` (`times`) instead of `*` (`mtimes`).

Example: `funx = @(t) sin(2*t);`

`funy` — Parametric function for *y* coordinates
anonymous function | function handle

Parametric function for *y* coordinates, specified as a function handle to a named or anonymous function.

Specify a function of the form `y = funy(t)`. The function must accept a vector input argument and return a vector output argument of the same size. Use array operators instead of matrix operators for the best performance. For example, use `.*` (`times`) instead of `*` (`mtimes`).

Example: `funy = @(t) cos(3*t);`

`tinterval` — Interval for `t`
`[-5 5]` (default) | two-element vector of form `[tmin tmax]`

Interval for `t`, specified as a two-element vector of the form `[tmin tmax]`.

`ax` — Axes object
axes object

Axes object. If you do not specify an axes object, then `fplot` uses the current axes (`gca`).

`LineSpec` — Line specification
character vector | string

Line specification, specified as a character vector or string with a line style, marker, and color. The elements can appear in any order, and you can omit one or more options. To show only markers with no connecting lines, specify a marker and omit the line style.

Example: `'r--o'` specifies a red color, a dashed line, and circle markers

Line Style Specifier	Description
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Marker Specifier	Description
`o`	Circle
`+`	Plus sign
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Pentagram
`h`	Hexagram

Color Specifier	Description
y	yellow
m	magenta
c	cyan
r	red
g	green
b	blue
w	white
k	black

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The properties listed here are only a subset. For a complete list, see FunctionLine or ParameterizedFunctionLine.

Example: `'Marker','o','MarkerFaceColor','red'`

`MeshDensity` — Number of evaluation points
23 (default) | number

Number of evaluation points, specified as a number. The default is `23`. Because `fplot` uses adaptive evaluation, the actual number of evaluation points is greater.

`Color` — Line color
`[0 0.4470 0.7410]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width

`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

`Marker` — Marker symbol

`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | `'x'` | `'s'` | `'d'` | `...`

Marker symbol, specified as one of the values in this table. By default, a line does not have markers. Add markers at selected points along the line by specifying a marker.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross

Value	Description
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerEdgeColor` — Marker outline color

`'auto'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'auto'` uses the same color as the `Color` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` value uses the same color as the `MarkerEdgeColor` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Example: `'#D2F9A7'`

`MarkerSize` — Marker size
6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

Output Arguments

`fp` — One or more `FunctionLine` or `ParameterizedFunctionLine` objects
scalar | vector

One or more `FunctionLine` or `ParameterizedFunctionLine` objects, returned as a scalar or a vector.

- If you use the `fplot(f)` syntax or a variation of this syntax, then `fplot` returns `FunctionLine` objects.
- If you use the `fplot(funx,funy)` syntax or a variation of this syntax, then `fplot` returns `ParameterizedFunctionLine` objects.

You can use these objects to query and modify properties of a specific line. For a list of properties, see FunctionLine and ParameterizedFunctionLine.

Tips

- Use element-wise operators for the best performance and to avoid a warning message. For example, use `x.*y` instead of `x*y`. For more information, see “Array vs. Matrix Operations”.
- When you zoom in on the chart, `fplot` replots the data, which can reveal hidden details.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions

`fcontour` | `fimplicit` | `fmesh` | `fplot3` | `fsurf` | `hold` | `title`

Properties

FunctionLine | ParameterizedFunctionLine

Introduced before R2006a

fplot3

3-D parametric curve plotter

Syntax

```
fplot3(funx,funy,funz)
fplot3(funx,funy,funz,tinterval)

fplot3( ___ ,LineSpec)
fplot3( ___ ,Name,Value)
fplot3(ax, ___ )
fp = fplot3( ___ )
```

Description

`fplot3(funx,funy,funz)` plots the parametric curve defined by `x = funx(t)`, `y = funy(t)`, and `z = funz(t)` over the default interval `[-5,5]` for `t`.

`fplot3(funx,funy,funz,tinterval)` plots over the specified interval. Specify the interval as a two-element vector of the form `[tmin tmax]`.

`fplot3( ___ ,LineSpec)` sets the line style, marker symbol, and line color. For example, `'-r'` specifies a red line. Use this option after any of the previous input argument combinations.

`fplot3( ___ ,Name,Value)` specifies line properties using one or more name-value pair arguments. For example, `'LineWidth',2` specifies a line width of 2 points.

`fplot3(ax, ___ )` plots into the axes specified by `ax` instead of the current axes. Specify the axes as the first input argument.

`fp = fplot3( ___ )` returns a `ParameterizedFunctionLine` object. Use the object to query and modify properties of a specific line. For a list of properties, see ParameterizedFunctionLine.

Examples

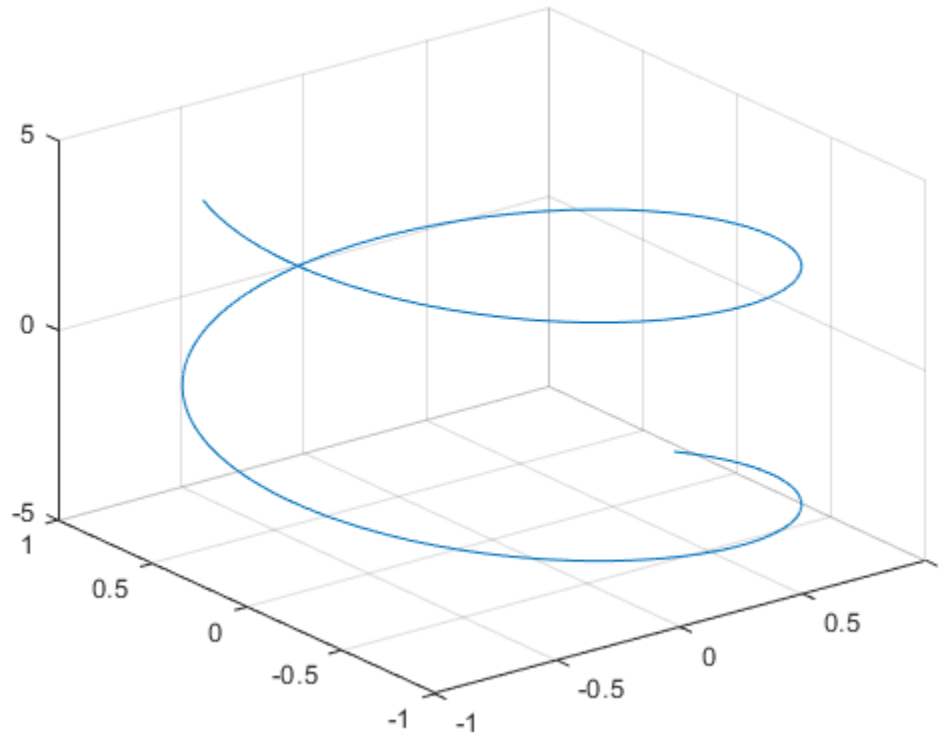
Plot 3-D Parametric Line

Plot the 3-D parametric line

$$x = \sin(t)$$
$$y = \cos(t)$$
$$z = t$$

over the default parameter range `[-5 5]`.

```
xt = @(t) sin(t);
yt = @(t) cos(t);
zt = @(t) t;
fplot3(xt,yt,zt)
```



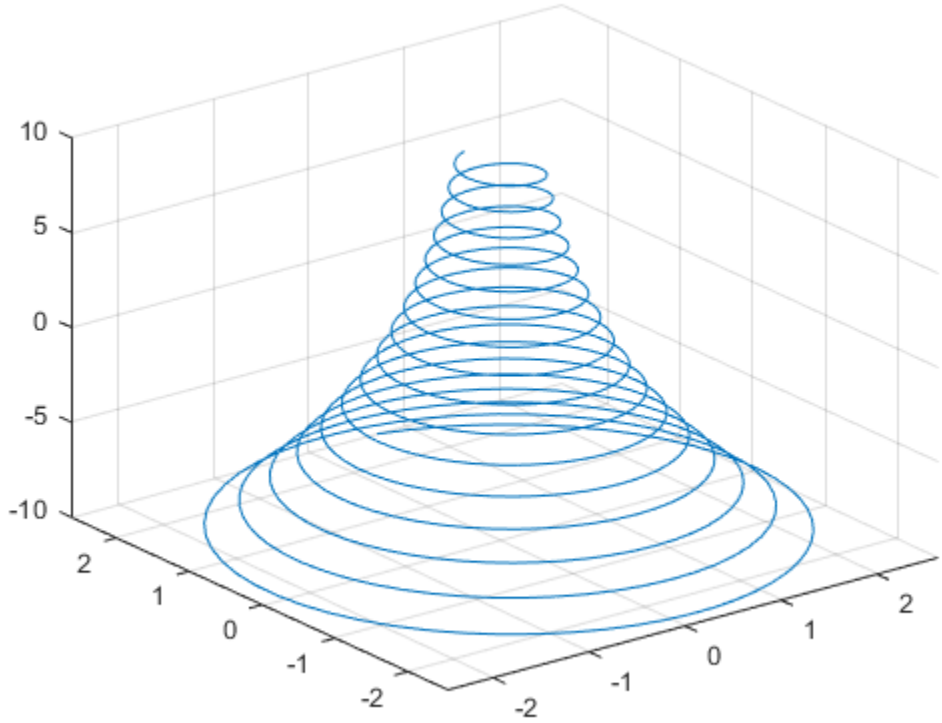
Specify Parameter Range

Plot the parametric line

$$x = e^{-t/10}\sin(5t)$$
$$y = e^{-t/10}\cos(5t)$$
$$z = t$$

over the parameter range `[-10 10]` by specifying the fourth input argument of `fplot3`.

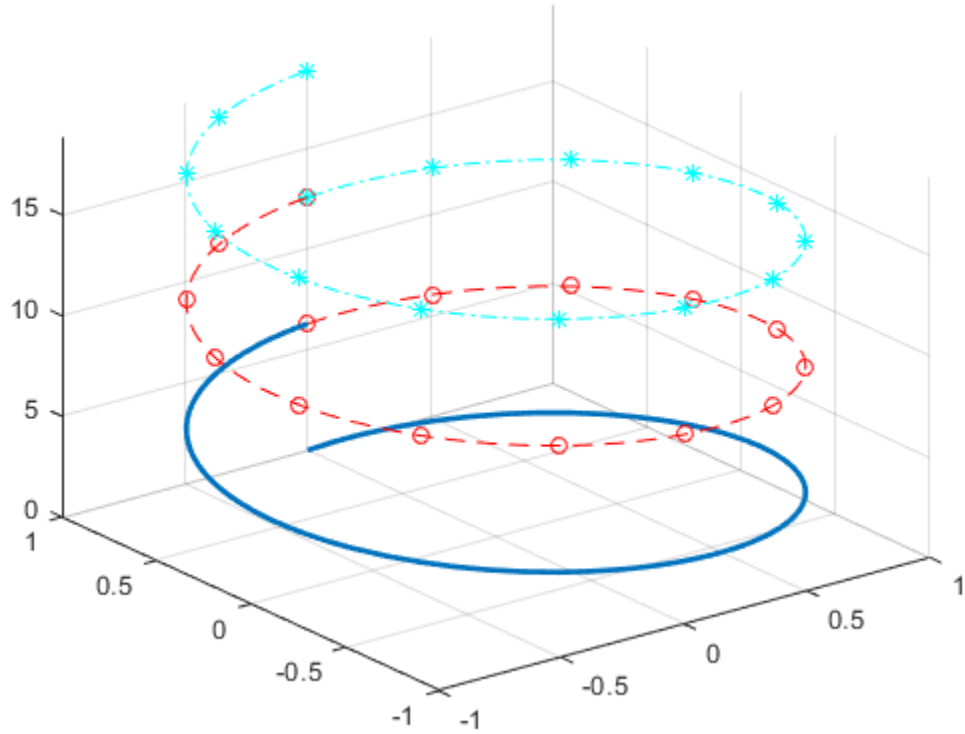
```
xt = @(t) exp(-t/10).*sin(5*t);
yt = @(t) exp(-t/10).*cos(5*t);
zt = @(t) t;
fplot3(xt,yt,zt,[-10 10])
```



Specify Line Properties and Display Markers

Plot the same 3-D parametric curve three times over different intervals of the parameter. For the first interval, use a line width of 2 points. For the second, specify a dashed red line style with circle markers. For the third, specify a cyan, dash-dotted line style with asterisk markers.

```
fplot3(@(t)sin(t), @(t)cos(t), @(t)t, [0 2*pi], 'LineWidth', 2)
hold on
fplot3(@(t)sin(t), @(t)cos(t), @(t)t, [2*pi 4*pi], '--or')
fplot3(@(t)sin(t), @(t)cos(t), @(t)t, [4*pi 6*pi], '-.*c')
hold off
```

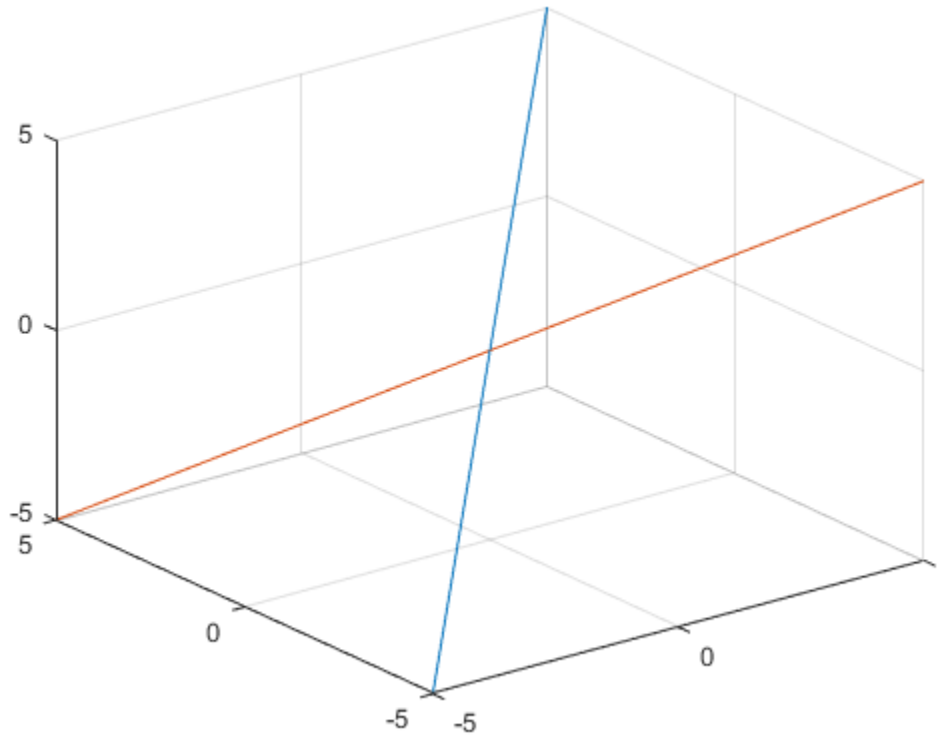


Plot Multiple Lines in Same Axes

Plot multiple lines in the same axes using `hold on`.

```
fplot3(@(t)t, @(t)t, @(t)t)
hold on
```

```
fplot3(@(t)-t, @(t)t, @(t)-t)
hold off
```



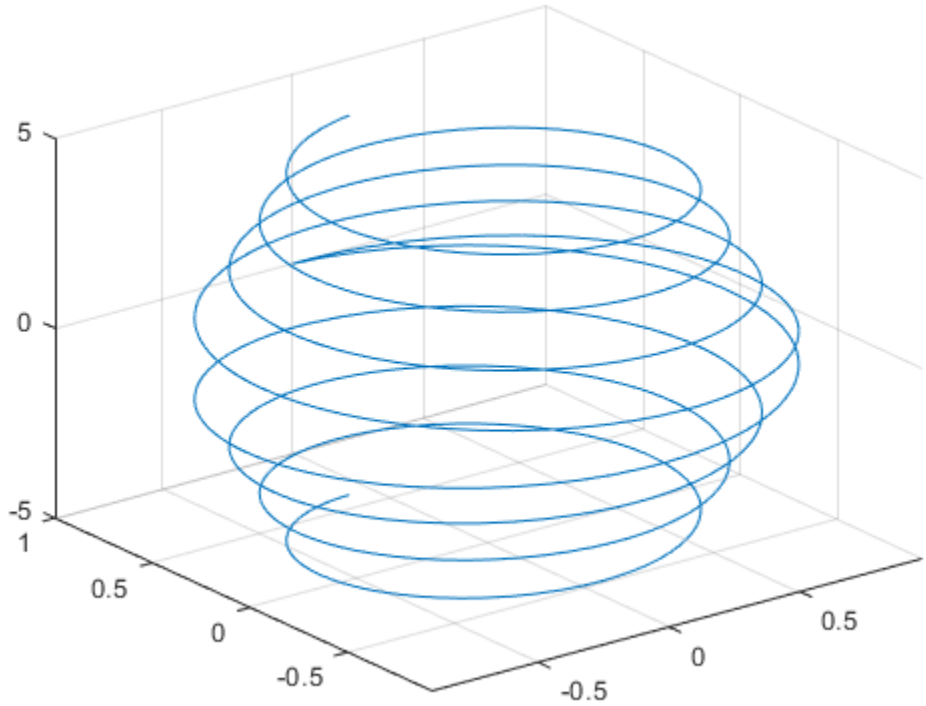
Modify 3-D Parametric Line After Creation

Plot the parametric line

$$x = e^{-|t|/10}\sin(5|t|)$$
$$y = e^{-|t|/10}\cos(5|t|)$$
$$z = t\,.$$

Assign the parameterized function line object to a variable.

```
xt = @(t)exp(-abs(t)/10).*sin(5*abs(t));
yt = @(t)exp(-abs(t)/10).*cos(5*abs(t));
zt = @(t)t;
fp = fplot3(xt,yt,zt)
```

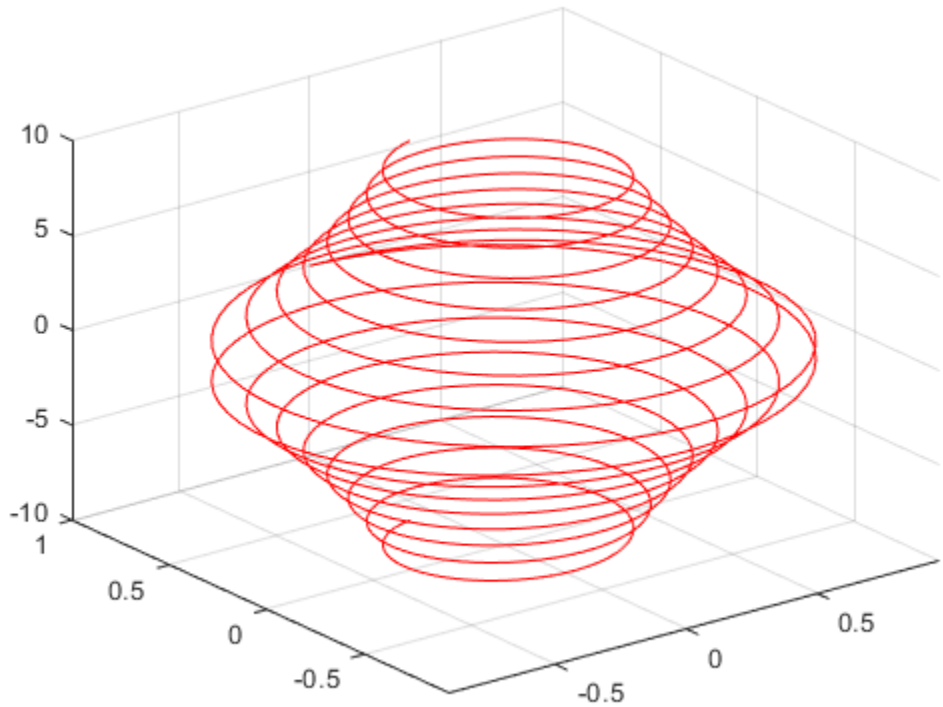



```
fp =
  ParameterizedFunctionLine with properties:

    XFunction: @(t)exp(-abs(t)/10).*sin(5*abs(t))
    YFunction: @(t)exp(-abs(t)/10).*cos(5*abs(t))
    ZFunction: @(t)t
        Color: [0 0.4470 0.7410]
```

```
    LineStyle: '-'
    LineWidth: 0.5000

  Show all properties
```

Change the range of parameter values to `[-10 10]` and change the line color to red.

```
fp.TRange = [-10 10];
fp.Color = 'r';
```



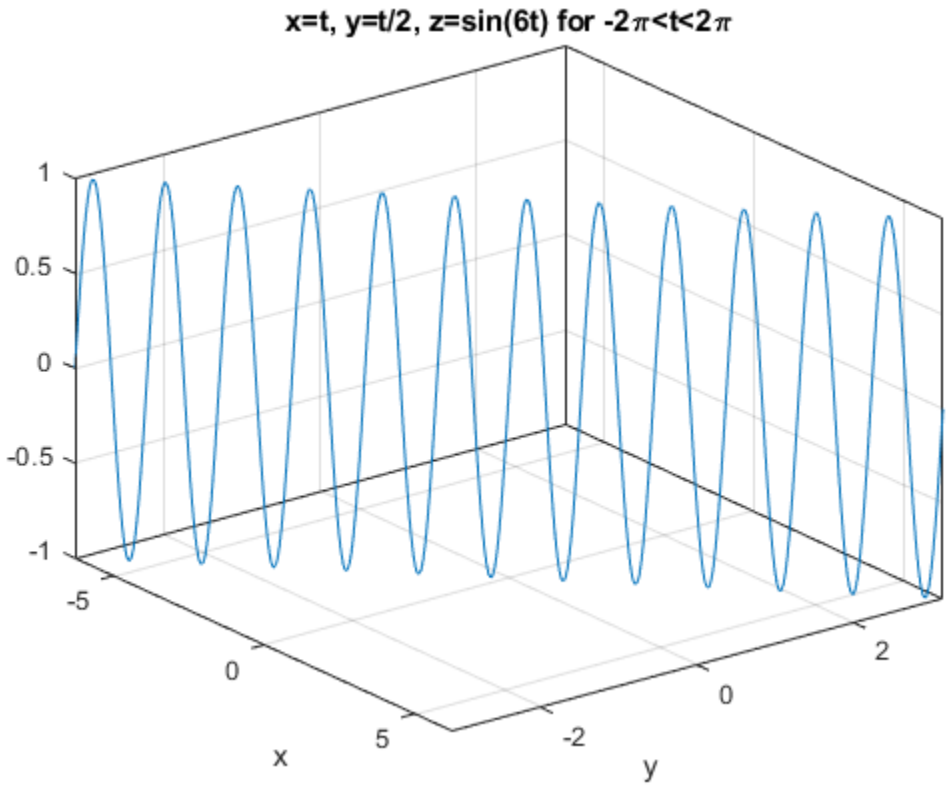
Add Title and Axis Labels and Format Ticks

For t values in the range -2π to 2π, plot the parametric line

$$\begin{aligned} x &= t \\ y &= t/2 \\ z &= \sin(6t)\,. \end{aligned}$$

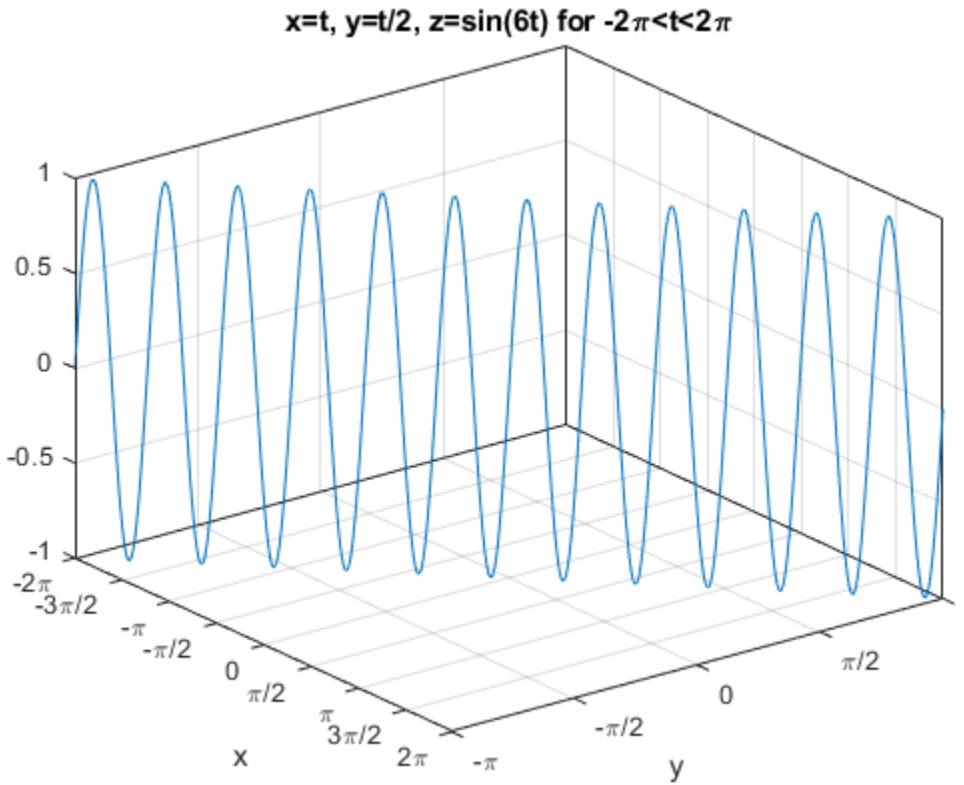
Add a title, x-axis label, and y-axis label. Additionally, change the view of the axes and display the axes box outline.

```
xt = @(t)t;
yt = @(t)t/2;
zt = @(t)sin(6*t);
fplot3(xt,yt,zt,[-2*pi 2*pi],'MeshDensity',30,'LineWidth',1);

title('x=t, y=t/2, z=sin(6t) for -2\pi<t<2\pi')
xlabel('x');
ylabel('y');
view(52.5,30)
box on
```



Access the axes object using `gca`. Specify the *x*-axis tick values and associated labels using the `XTick` and `XTickLabel` properties of the axes object. Similarly, specify the *y*-axis tick values and associated labels.

```
ax = gca;
ax.XTick = -2*pi:pi/2:2*pi;
ax.XTickLabel = {'-2\pi','-3\pi/2','-\pi','-\pi/2','0','\pi/2','\pi','3\pi/2','2\pi'};
ax.YTick = -pi:pi/2:pi;
ax.YTickLabel = {'-\pi','-\pi/2','0','\pi/2','\pi'};
```



Input Arguments

`funx` — Parametric function for *x* coordinates
function handle

Parametric function for *x* coordinates, specified as a function handle to a named or anonymous function.

Specify a function of the form `x = funx(t)`. The function must accept a vector input argument and return a vector output argument of the same size. Use array operators

instead of matrix operators for the best performance. For example, use `.*` (`times`) instead of `*` (`mtimes`).

Example: `funx = @(t) sin(2*t);`

`funy` — Parametric function for *y* coordinates
function handle

Parametric function for *y* coordinates, specified as a function handle to a named or anonymous function.

Specify a function of the form `y = funy(t)`. The function must accept a vector input argument and return a vector output argument of the same size. Use array operators instead of matrix operators for the best performance. For example, use `.*` (`times`) instead of `*` (`mtimes`).

Example: `funy = @(t) cos(2*t);`

`funz` — Parametric function for *z* coordinates
function handle

Parametric function for *z* coordinates, specified as a function handle to a named or anonymous function.

Specify a function of the form `z = funz(t)`. The function must accept a vector input argument and return a vector output argument of the same size. Use array operators instead of matrix operators for the best performance. For example, use `.*` (`times`) instead of `*` (`mtimes`).

Example: `funz = @(t) t;`

`tinterval` — Interval for parameter `t`
`[-5 5]` (default) | two-element vector of form `[tmin tmax]`

Interval for parameter `t`, specified as a two-element vector of the form `[tmin tmax]`.

`ax` — Axes object
axes object

Axes object. If you do not specify an axes object, then `fplot3` uses the current axes (`gca`).

`LineSpec` — Line specification
character vector | string

Line specification, specified as a character vector or string with a line style, marker, and color. The elements can appear in any order, and you can omit one or more options. To show only markers with no connecting lines, specify a marker and omit the line style.

Example: `'r--o'` specifies a red color, a dashed line, and circle markers

Line Style Specifier	Description
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Marker Specifier	Description
`o`	Circle
`+`	Plus sign
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Pentagram
`h`	Hexagram

Color Specifier	Description
`y`	yellow
`m`	magenta
`c`	cyan
`r`	red

Color Specifier	Description
g	green
b	blue
w	white
k	black

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Marker','o','MarkerFaceColor','red'`

The properties listed here are only a subset. For a complete list, see ParameterizedFunctionLine.

`MeshDensity` — Number of evaluation points
23 (default) | number

Number of evaluation points, specified as a number. The default is `23`. Because `fplot3` uses adaptive evaluation, the actual number of evaluation points is greater.

`Color` — Line color
`[0 0.4470 0.7410]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width

`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

`Marker` — Marker symbol

`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | `'x'` | `'s'` | `'d'` | `...`

Marker symbol, specified as one of the values in this table. By default, a line does not have markers. Add markers at selected points along the line by specifying a marker.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle

Value	Description
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerEdgeColor` — Marker outline color

`'auto'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'auto'` uses the same color as the `Color` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` value uses the same color as the `MarkerEdgeColor` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Example: `'#D2F9A7'`

`MarkerSize` — Marker size
6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

Output Arguments

`fp` — One or more `ParameterizedFunctionLine` objects
scalar | vector

One or more `ParameterizedFunctionLine` objects, returned as a scalar or a vector. You can use these objects to query and modify properties of a specific `ParameterizedFunctionLine` object. For details, see ParameterizedFunctionLine.

See Also

Functions
`fcontour` | `fimplicit` | `fimplicit3` | `fmesh` | `fplot` | `fsurf` | `hold` | `title`

Properties
ParameterizedFunctionLine

Introduced in R2016a

fprintf

Write data to text file

Syntax

```
fprintf(fileID,formatSpec,A1,...,An)
fprintf(formatSpec,A1,...,An)

nbytes = fprintf( ___ )
```

Description

`fprintf(fileID,formatSpec,A1,...,An)` applies the `formatSpec` to all elements of arrays `A1,...An` in column order, and writes the data to a text file. `fprintf` uses the encoding scheme specified in the call to `fopen`.

`fprintf(formatSpec,A1,...,An)` formats data and displays the results on the screen.

`nbytes = fprintf( ___ )` returns the number of bytes that `fprintf` writes, using any of the input arguments in the preceding syntaxes.

Examples

Print Literal Text and Array Values

Print multiple numeric values and literal text to the screen.

```
A1 = [9.9, 9900];
A2 = [8.8,  7.7 ; ...
      8800, 7700];
formatSpec = 'X is %4.2f meters or %8.3f mm\n';
fprintf(formatSpec,A1,A2)
```

```
X is 9.90 meters or 9900.000 mm
X is 8.80 meters or 8800.000 mm
X is 7.70 meters or 7700.000 mm
```

`%4.2f` in the `formatSpec` input specifies that the first value in each line of output is a floating-point number with a field width of four digits, including two digits after the decimal point. `%8.3f` in the `formatSpec` input specifies that the second value in each line of output is a floating-point number with a field width of eight digits, including three digits after the decimal point. `\n` is a control character that starts a new line.

Print Double-Precision Values as Integers

Explicitly convert double-precision values with fractions to integer values.

```
a = [1.02 3.04 5.06];
fprintf('%d\n',round(a));
```

```
1
3
5
```

`%d` in the `formatSpec` input prints each value in the vector, `round(a)`, as a signed integer. `\n` is a control character that starts a new line.

Write Tabular Data to Text File

Write a short table of the exponential function to a text file called `exp.txt`.

```
x = 0:.1:1;
A = [x; exp(x)];

fileID = fopen('exp.txt','w');
fprintf(fileID,'%6s %12s\n','x','exp(x)');
fprintf(fileID,'%6.2f %12.8f\n',A);
fclose(fileID);
```

The first call to `fprintf` prints header text `x` and `exp(x)`, and the second call prints the values from variable `A`.

If you plan to read the file with Microsoft Notepad, use `'\r\n'` instead of `'\n'` to move to a new line. For example, replace the calls to `fprintf` with the following:

```
fprintf(fileID,'%6s %12s\r\n','x','exp(x)');
fprintf(fileID,'%6.2f %12.8f\r\n',A);
```

MATLAB import functions, all UNIX applications, and Microsoft Word and WordPad recognize `'\n'` as a newline indicator.

View the contents of the file with the `type` command.

```
type exp.txt
```

```
     x       exp(x)
  0.00   1.00000000
  0.10   1.10517092
  0.20   1.22140276
  0.30   1.34985881
  0.40   1.49182470
  0.50   1.64872127
  0.60   1.82211880
  0.70   2.01375271
  0.80   2.22554093
  0.90   2.45960311
  1.00   2.71828183
```

Get Number of Bytes Written to File

Write data to a file and return the number of bytes written.

Write an array of data, `A`, to a file and get the number of bytes that `fprintf` writes.

```
A = magic(4);

fileID = fopen('myfile.txt','w');
nbytes = fprintf(fileID,'%5d %5d %5d %5d\n',A)
```

```
nbytes = 96
```

The `fprintf` function wrote 96 bytes to the file.

Close the file.

```
fclose(fileID);
```

View the contents of the file with the `type` command.

```
type('myfile.txt')
```

```
   16     5     9     4
    2    11     7    14
    3    10     6    15
   13     8    12     1
```

Display Hyperlinks in Command Window

Display a hyperlink (The MathWorks Web Site) on the screen.

```
url = 'https://www.mathworks.com';
sitename = 'The MathWorks Web Site';

fprintf('<a href = "%s">%s</a>\n',url,sitename)
```

`%s` in the `formatSpec` input indicates that the values of the variables `url` and `sitename`, should be printed as text.

Input Arguments

`fileID` — File identifier
1 (default) | 2 | scalar

File identifier, specified as one of the following:

- A file identifier obtained from `fopen`.
- `1` for standard output (the screen).
- `2` for standard error.

Data Types: `double`

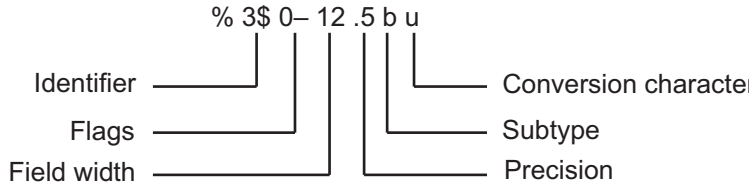
`formatSpec` — Format of output fields
formatting operators

Format of the output fields, specified using formatting operators. `formatSpec` also can include ordinary text and special characters.

`formatSpec` can be a character vector in single quotes, or, starting in R2016b, a string scalar.

Formatting Operator

A formatting operator starts with a percent sign, `%`, and ends with a conversion character. The conversion character is required. Optionally, you can specify identifier, flags, field width, precision, and subtype operators between `%` and the conversion character. (Spaces are invalid between operators and are shown here only for readability).



Conversion Character

This table shows conversion characters to format numeric and character data as text.

Value Type	Conversion	Details
Integer, signed	`%d` or `%i`	Base 10
Integer, unsigned	`%u`	Base 10
	`%o`	Base 8 (octal)
	`%x`	Base 16 (hexadecimal), lowercase letters `a-f`
	`%X`	Same as `%x`, uppercase letters `A-F`
Floating-point number	`%f`	Fixed-point notation (Use a precision operator to specify the number of digits after the decimal point.)
	`%e`	Exponential notation, such as `3.141593e+00` (Use a precision operator to specify the number of digits after the decimal point.)

Value Type	Conversion	Details
	`%E`	Same as `%e`, but uppercase, such as `3.141593E+00` (Use a precision operator to specify the number of digits after the decimal point.)
	`%g`	The more compact of `%e` or `%f`, with no trailing zeros (Use a precision operator to specify the number of significant digits.)
	`%G`	The more compact of `%E` or `%f`, with no trailing zeros (Use a precision operator to specify the number of significant digits.)
Characters or strings	`%c`	Single character
	`%s`	Character vector or string array. The type of the output text is the same as the type of `formatSpec`.

Optional Operators

The optional identifier, flags, field width, precision, and subtype operators further define the format of the output text.

- **Identifier**

  Order for processing the function input arguments. Use the syntax *n*`$`, where *n* represents the positions of the other input arguments in the function call.

  Example: `('%3$s %2$s %1$s %2$s','A','B','C')` prints input arguments `'A'`, `'B'`, `'C'` as follows: `C B A B`.

  Note: If an input argument is an array, you cannot use identifiers to specify particular array elements from that input argument.

- **Flags**

`'-'`	Left-justify. **Example:** `%-5.2f` **Example:** `%-10s`

`'+'`	Always print a sign character (+ or –) for any numeric value. **Example:** `%+5.2f` Right-justify text. **Example:** `%+10s`
`' '`	Insert a space before the value. **Example:** `% 5.2f`
`'0'`	Pad to field width with zeros before the value. **Example:** `%05.2f`
`'#'`	Modify selected numeric conversions: • For `%o`, `%x`, or `%X`, print `0`, `0x`, or `0X` prefix. • For `%f`, `%e`, or `%E`, print decimal point even when precision is 0. • For `%g` or `%G`, do not remove trailing zeros or decimal point. **Example:** `%#5.0f`

- **Field Width**

  Minimum number of characters to print. The field width operator can be a number, or an asterisk (*) to refer to an input argument.

  When you specify * as the field width operator, the other input arguments must provide both a width and a value to be printed. Widths and values can be pairs of arguments or pairs within a numeric array. With * as the field width operator, you can print different values with different widths.

  Example: The input arguments `('%12d',intmax)` are equivalent to `('%*d',12,intmax)`.

  Example: The input arguments `('%*d',[2 10 5 100])` return `'10 100'`, with two spaces allocated for `10` and five spaces for `100`. As an alternative, you also can specify the field widths and values as multiple arguments, as in `('%*d',2,10,5,100)`.

  The function pads to field width with spaces before the value unless otherwise specified by flags.

- **Precision**

For `%f`, `%e`, or `%E`	Number of digits to the right of the decimal point **Example:** `'%.4f'` prints `pi` as `'3.1416'`

For `%g` or `%G`	Number of significant digits **Example:** `'%.4g'` prints `pi` as `'3.142'`

The precision operator can be a number, or an asterisk (*) to refer to an argument.

When you specify * as the field precision operator, the other input arguments must provide both a precision and a value to be printed. Precisions and values can be pairs of arguments, or pairs within a numeric array. With * as the precision operator, you can print different values to different precisions.

When you specify `*.*` as field width and precision operators, you must specify field widths, precisions, and values as triplets.

Example: The input arguments `('%.4f',pi)` are equivalent to `('%.*f',4,pi)`.

Example: The input arguments `('%6.4f',pi)` are equivalent to `('%.*f',6,4,pi)`.

Example: The input arguments `('%*.*f',6,4,pi,9,6,exp(1))` return `'3.1416 2.718282'`, with `9` and `6` as the field width and precision for the output of `exp(1)`.

Note If you specify a precision operator for floating-point values that exceeds the precision of the input numeric data type, the results might not match the input values to the precision you specified. The result depends on your computer hardware and operating system.

- **Subtypes**

  You can use a subtype operator to print a floating-point value as its octal, decimal, or hexadecimal value. The subtype operator immediately precedes the conversion character. This table shows the conversions that can use subtypes.

Input Value Type	Subtype and Conversion Character	Output Value Type

Floating-point number	`%bx` or `%bX`
	`%bo`
	`%bu`
		Double-precision hexadecimal, octal, or decimal value
		Example: `%bx` prints `pi` as `400921fb54442d18`

Input Value Type	Subtype and Conversion Character	Output Value Type

	`%tx` or `%tX`
	`%to`
	`%tu`
		Single-precision hexadecimal, octal, or decimal value
		Example: `%tx` prints `pi` as `40490fdb`

Text Before or After Formatting Operators

`formatSpec` can also include additional text before a percent sign, `%`, or after a conversion character. The text can be:

- Ordinary text to print.
- Special characters that you cannot enter as ordinary text. This table shows how to represent special characters in `formatSpec`.

Special Character	Representation
Single quotation mark	`''`
Percent character	`%%`
Backslash	`\\`
Alarm	`\a`
Backspace	`\b`
Form feed	`\f`
New line	`\n`
Carriage return	`\r`
Horizontal tab	`\t`
Vertical tab	`\v`

Character whose Unicode numeric value can be represented by the hexadecimal number, `N`	`\xN`
	
	Example:
	`sprintf('\x5A')` returns `'Z'`

Special Character	Representation
Character whose Unicode numeric value can be represented by the octal number, `N`	`\N` **Example:** `sprintf('\132')` returns `'Z'`

Notable Behavior of Conversions with Formatting Operators

- Numeric conversions print only the real component of complex numbers.
- If you specify a conversion that does not fit the data, such as a text conversion for a numeric value, MATLAB overrides the specified conversion, and uses `%e`.

  Example: `'%s'` converts `pi` to `3.141593e+00`.
- If you apply a text conversion (either `%c` or `%s`) to integer values, MATLAB converts values that correspond to valid character codes to characters.

  Example: `'%s'` converts `[65 66 67]` to `ABC`.

Data Types: `char` | `string`

`A1,...,An` — Numeric or character arrays
scalar | vector | matrix | multidimensional array

Numeric or character arrays, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`

Output Arguments

`nbytes` — Number of bytes
scalar

Number of bytes that `fprintf` writes, returned as a scalar. When writing to a file, `nbytes` is determined by the character encoding. When printing data to the screen, `nbytes` is the number of characters displayed on the screen.

Tips

- Format specifiers for the reading functions `sscanf` and `fscanf` differ from the formats for the writing functions `sprintf` and `fprintf`. The reading functions do not support a precision field. The width field specifies a minimum for writing, but a maximum for reading.

References

[1] Kernighan, B. W., and D. M. Ritchie, *The C Programming Language*, Second Edition, Prentice-Hall, Inc., 1988.

[2] ANSI specification X3.159-1989: “Programming Language C,” ANSI, 1430 Broadway, New York, NY 10018.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The `formatSpec` parameter must be constant.
- In `formatSpec`, hexadecimal numbers must be in the range [0 7F] and octal numbers must be in the range [0 177].
- If `fileID` has a constant value of `1` or `2` and extrinsic calls are not possible, the code generator produces a C `printf` call. Extrinsic calls are not possible when extrinsic calls are disabled or when `fprintf` is called inside a `parfor` loop.
- The behavior of `fprintf` in the generated code matches the C compiler behavior instead of the MATLAB behavior in these cases:
  - The format specifier has a corresponding C format specifier, for example, `%e` or `%E`.
  - The `fprintf` call is inside a `parfor` loop.
  - Extrinsic calls are disabled.

- These options and capabilities are not supported:
  - The `n$` position identifier for reordering input values
  - Printing arrays
  - Using subtypes to print a floating-point number as its octal, decimal, or hexadecimal value
- When you call `fprintf` with the format specifier `%s`, you cannot put a null character in the middle of the input character vector. To write a null character, use `fprintf(fid, '%c', char(0))`.
- Input argument types must match their format types. For example, if `n` is a double, code generation does not allow the following code:

  ```
  str = sprintf('%d',n)
  ```

  For code generation, first cast `n` to a signed integer type such as `int8`.

  ```
  str = sprintf('%d',int8(n))
  ```
- When you call `fprintf` with an integer format specifier, the type of the integer argument must be a type that the target hardware can represent as a native C type. For example, if you call `fprintf('%d', int64(n))`, then the target hardware must have a native C type that supports a 64-bit integer.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`disp` | `fclose` | `ferror` | `fopen` | `fread` | `fscanf` | `fseek` | `ftell` | `fwrite` | `sprintf`

Topics

“Export Cell Array to Text File”
“Append To or Overwrite Existing Text Files”
“Formatting Text”

Introduced before R2006a

fprintf (serial)

(Not recommended) Write text to device

Note This `serial` object function is not recommended. Use `serialport` object functions instead. See “Compatibility Considerations”.

Syntax

```
fprintf(obj,'cmd')
fprintf(obj,'format','cmd')
fprintf(obj,'cmd','mode')
fprintf(obj,'format','cmd','mode')
```

Description

`fprintf(obj,'cmd')` writes the string `cmd` to the device connected to the serial port object, `obj`. The default format is `%s\n`. The write operation is synchronous and blocks the command line until execution completes. The `cmd` can be either a SCPI command you provide, or a command you provide based on instructions from your device vendor.

`fprintf(obj,'format','cmd')` writes the string using the format specified by *`format`*.

`fprintf(obj,'cmd','mode')` writes the string with command-line access specified by *`mode`*. *`mode`* specifies if `cmd` is written synchronously or asynchronously.

`fprintf(obj,'format','cmd','mode')` writes the string using the specified format. *`format`* is a C language conversion specification.

You need an open connection from the serial port object, `obj`, to the device before performing read or write operations.

To open a connection to the device, use the `fopen` function. When `obj` has an open connection to the device, it has a `Status` property value of `open`. Refer to “Troubleshooting Common Errors” for `fprintf` errors.

To understand the use of `fprintf`, refer to “Completing a Write Operation with fprintf” and “Rules for Writing the Terminator”.

Input Arguments

`format`

ANSI C conversion specification includes these conversion characters.

Specifier	Description
`%c`	Single character
`%d` or `%i`	Decimal notation (signed)
`%e`	Exponential notation (using lowercase `e` as in `3.1415e+00`)
`%E`	Exponential notation (using uppercase `E` as in `3.1415E+00`)
`%f`	Fixed-point notation
`%g`	The more compact of `%e` or `%f`. Insignificant zeros do not print.
`%G`	Same as `%g`, but using uppercase `E`
`%o`	Octal notation (unsigned)
`%s`	String of characters
`%u`	Decimal notation (unsigned)
`%x`	Hexadecimal notation (using lowercase letters `a–f`)
`%X`	Hexadecimal notation (using uppercase letters `A–F`)

`cmd`

Specifies the string `cmd`, which can be either a SCPI command you provide, or a command you provide based on instructions from your device vendor.

`mode`

Specifies whether the string `cmd` is written synchronously or asynchronously:

- `sync`: `cmd` is written synchronously and the command line is blocked.
- `async`: `cmd` is written asynchronously and the command line is not blocked.

If *`mode`* is not specified, the write operation is synchronous.

If you specify asynchronous *mode*, when the write operation occurs:

- The `BytesToOutput` property value continuously updates to reflect the number of bytes in the output buffer.
- The MATLAB file callback function specified for the `OutputEmptyFcn` property is executed when the output buffer is empty.

To determine whether an asynchronous write operation is in progress, use the `TransferStatus` property.

For more information on synchronous and asynchronous write operations, see “Write and Read Data”.

Examples

Create a serial port object `s` and connect it to a Tektronix TDS 210 oscilloscope. Write the `RS232?` command with `fprintf`. `RS232?` instructs the scope to return serial port communications settings. This example works on a Windows platform.

```
s = serial('COM1');
fopen(s)
fprintf(s,'RS232?')
```

Specify a format for the data that does not include the terminator, or configure the terminator to empty.

```
s = serial('COM1');
fopen(s)
fprintf(s,'%s','RS232?')
```

The default format for `fprintf` is `%s\n`. Therefore, the terminator specified by the `Terminator` property is automatically written. However, sometimes you might want to suppress writing the terminator.

Specify an array of formats and commands:

```
s = serial('COM1');
fopen(s)
fprintf(s,['ch:%d scale:%d'],[1 20e-3],'sync')
```

Compatibility Considerations

`serial` object interface is not recommended

Not recommended starting in R2019b

Use of this function with a `serial` object is not recommended. To access a serial port device, use a `serialport` object with its functions and properties instead.

See “Transition Your Code to serialport Interface” for more information about using the recommended functionality.

See Also

Functions

`fopen` | `fwrite` | `stopasync`

Topics

Writing and Reading Data

Introduced before R2006a

frame2im

Return image data associated with movie frame

Syntax

```
RGB = frame2im(F)
[X,map] = frame2im(F)
```

Description

`RGB = frame2im(F)` returns the truecolor (RGB) image from the single movie frame `F`.

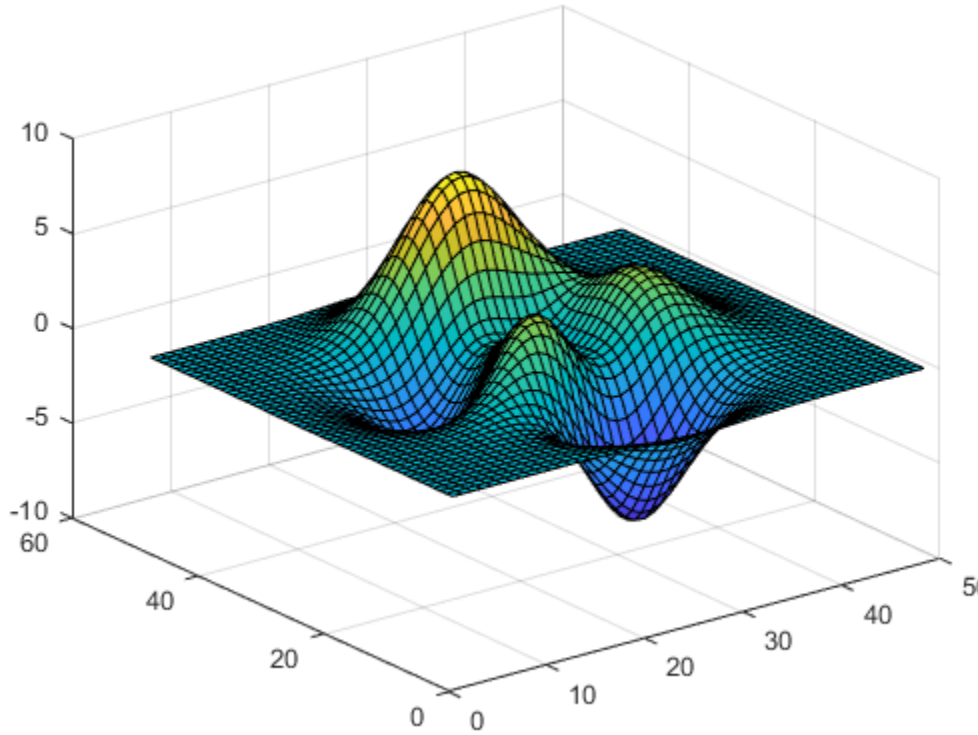
`[X,map] = frame2im(F)` returns the indexed image data `X` and associated colormap `map` from the single movie frame `F`.

Examples

Convert Movie Frame to Image Data

Create a surface plot. The data tip gives the x-, y- and z-coordinate of points along the surface.

```
surf(peaks)
```



Use `getframe` to capture the plot as a movie frame. The `'colormap'` field is empty, therefore the movie frame contains a truecolor (RGB) image.

```
F = getframe
```

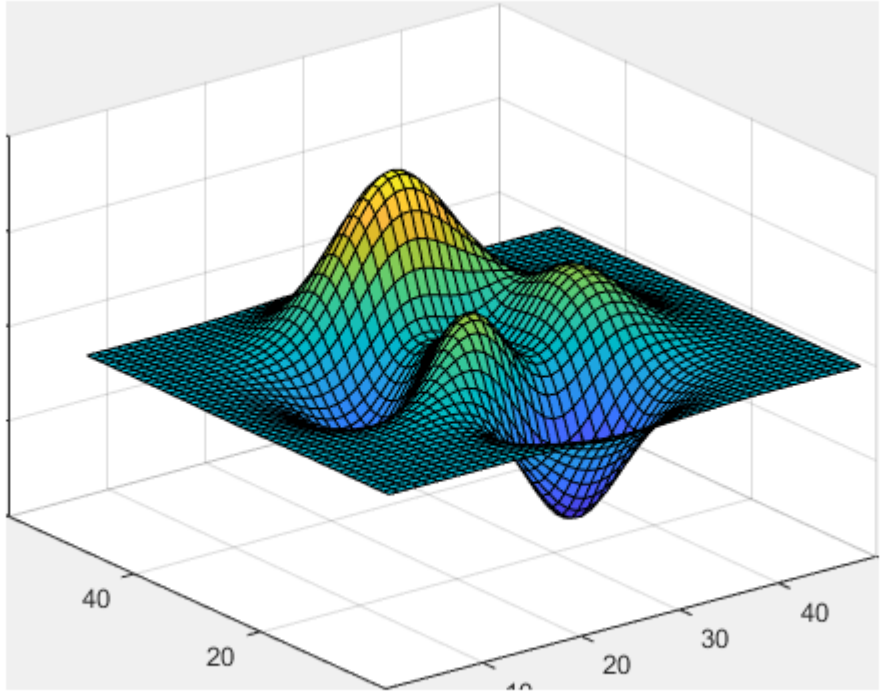
```
F = 

  struct with fields:

      cdata: [344x436x3 uint8]
   colormap: []
```

Convert the captured movie frame to image data.

```
RGB = frame2im(F);
```

Display the truecolor image. The data tip gives information about the column and row indices and RGB value of pixels.

```
figure
imshow(RGB)
```



Input Arguments

F — Movie frame
structure

Movie frame, specified as a structure with two fields:

- `cdata` — The image data stored as an array of `uint8` values.
- `colormap` — The colormap. If the movie frame contains a truecolor (RGB) images, then this field is empty (`[]`).

You can create a move frame structure by using the functions `im2frame` and `getframe`.

Output Arguments

`RGB` — Truecolor image
m-by-*n*-by-3 numeric array

Truecolor image, returned as an *m*-by-*n*-by-3 numeric array.

Data Types: `uint8`

`X` — Indexed image
m-by-*n* numeric matrix

Indexed image, returned as an *m*-by-*n* numeric matrix.

Data Types: `uint8`

`map` — Colormap
c-by-3 numeric matrix

Colormap associated with indexed image `X`, returned as a *c*-by-3 numeric matrix with values in the range [0, 1]. Each row of `map` is a three-element RGB triplet that specifies the red, green, and blue components of a single color of the colormap.

Data Types: `double`

See Also

`getframe` | `im2frame` | `ind2rgb` | `movie`

Topics

"Image Types"

Introduced before R2006a

fread

Read data from binary file

Syntax

```
A = fread(fileID)
A = fread(fileID,sizeA)
A = fread(fileID,sizeA,precision)
A = fread(fileID,sizeA,precision,skip)
A = fread(fileID,sizeA,precision,skip,machinefmt)
[A,count] = fread( ___ )
```

Description

`A = fread(fileID)` reads data from an open binary file into column vector `A` and positions the file pointer at the end-of-file marker. The binary file is indicated by the file identifier, `fileID`. Use `fopen` to open the file and obtain the `fileID` value. When you finish reading, close the file by calling `fclose(fileID)`.

`A = fread(fileID,sizeA)` reads file data into an array, `A`, with dimensions, `sizeA`, and positions the file pointer after the last value read. `fread` populates `A` in column order.

`A = fread(fileID,sizeA,precision)` interprets values in the file according to the form and size described by `precision`. The `sizeA` argument is optional.

`A = fread(fileID,sizeA,precision,skip)` skips the number of bytes or bits specified by `skip` after reading each value in the file. The `sizeA` argument is optional.

`A = fread(fileID,sizeA,precision,skip,machinefmt)` additionally specifies the order for reading bytes or bits in the file. The `sizeA` and `skip` arguments are optional.

`[A,count] = fread( ___ )` additionally returns the number of characters that `fread` reads into `A`. You can use this syntax with any of the input arguments of the previous syntaxes.

Examples

Read Entire File of uint8 Data

Write a nine-element vector to a sample file, `nine.bin`.

```
fileID = fopen('nine.bin','w');
fwrite(fileID,[1:9]);
fclose(fileID);
```

Read all the data in the file into a vector of class `double`. By default, `fread` reads a file 1 byte at a time, interprets each byte as an 8-bit unsigned integer (`uint8`), and returns a `double` array.

```
fileID = fopen('nine.bin');
A = fread(fileID)
```

```
A = 9×1

     1
     2
     3
     4
     5
     6
     7
     8
     9
```

`fread` returns a column vector, with one element for each byte in the file.

View information about `A`.

```
whos A
```

```
  Name      Size            Bytes  Class     Attributes

  A         9x1                72  double
```

Close the file.

```
fclose(fileID);
```

Read Entire File of Double-Precision Data

Create a file named `doubledata.bin`, containing nine double-precision values.

```
fileID = fopen('doubledata.bin','w');
fwrite(fileID,magic(3),'double');
fclose(fileID);
```

Open the file, `doubledata.bin`, and read the data in the file into a 3-by-3 array, `A`. Specify that the source data is class `double`.

```
fileID = fopen('doubledata.bin');
A = fread(fileID,[3 3],'double')
```

```
A = 3×3

     8     1     6
     3     5     7
     4     9     2
```

Close the file.

```
fclose(fileID);
```

Read Text File

Read the contents of the file, `fread.m`. Transpose the output array, `A` so that it is a row vector.

```
fileID = fopen('fread.m');
A = fread(fileID,'*char')';
fclose(fileID);
```

`fread` returns the character array, `A`.

Read Selected Rows or Columns from File

Create a file named `nine.bin`, containing the values from 1 to 9. Write the data as `uint16` values.

```
fileID = fopen('nine.bin','w');
fwrite(fileID,[1:9],'uint16');
fclose(fileID);
```

Read the first six values into a 3-by-2 array. Specify that the source data is class `uint16`.

```
fileID = fopen('nine.bin');
A = fread(fileID,[3,2],'uint16')
```

```
A = 3×2

     1     4
     2     5
     3     6
```

`fread` returns an array populated column-wise with the first six values from the file, `nine.bin`.

Return to the beginning of the file.

```
frewind(fileID)
```

Read two values at a time, and skip one value before reading the next values. Specify this format using the `precision` value, `'2*uint16'`. Because the data is class `uint16`, one value is represented by 2 bytes. Therefore, specify the `skip` argument as `2`.

```
precision = '2*uint16';
skip = 2;
B = fread(fileID,[2,3],precision,skip)
```

```
B = 2×3

     1     4     7
     2     5     8
```

`fread` returns a 2-by-3 array populated column-wise with the values from `nine.bin`.

Close the file.

```
fclose(fileID);
```

Read Digits of Binary Coded Decimal Values

Create a file with binary coded decimal (BCD) values.

```
str = ['AB'; 'CD'; 'EF'; 'FA'];

fileID = fopen('bcd.bin','w');
fwrite(fileID,hex2dec(str),'ubit8');
fclose(fileID);
```

Read 1 byte at a time.

```
fileID = fopen('bcd.bin');
onebyte = fread(fileID,4,'*ubit8');
```

Display the BCD values.

```
disp(dec2hex(onebyte))
```

```
AB
CD
EF
FA
```

Return to the beginning of the file using `frewind`. If you read 4 bits at a time on a little-endian system, your results appear in the wrong order.

```
frewind(fileID)

err = fread(fileID,8,'*ubit4');
disp(dec2hex(err))
```

```
B
A
D
C
F
E
A
F
```

Return to the beginning of the file using `frewind`. Read the data 4 bits at a time as before, but specify a big-endian ordering to display the correct results.

```
frewind(fileID)

correct = fread(fileID,8,'*ubit4','ieee-be');
disp(dec2hex(correct))
```

```
A
B
C
D
E
F
F
A
```

Close the file.

```
fclose(fileID);
```

Input Arguments

`fileID` — File identifier

integer

File identifier of an open binary file, specified as an integer. Before reading a file with `fread`, you must use `fopen` to open the file and obtain the `fileID`.

Data Types: `double`

`sizeA` — Dimensions of output array

`Inf` (default) | integer | two-element row vector

Dimensions of the output array, `A`, specified as `Inf`, an integer, or a two-element row vector.

Form of the `sizeA` Input	Dimensions of the output array, `A`
`Inf`	Column vector, with each element containing a value in the file.

Form of the sizeA Input	Dimensions of the output array, A
n	Column vector with n elements.
[m,n]	m-by-n matrix, filled in column order. n can be Inf, but m cannot.

precision — Class and size of values to read
'uint8=>double' (default) | character vector or string scalar

Class and size in bits of the values to read, specified as a character vector or a string scalar in one of the following forms. Optionally the input specifies the class of the output matrix, A.

Form of the precision Input	Description
source	Input values are of the class specified by *source*. Output matrix A is class double. Example: 'int16'
source=>output	Input values are of the class specified by *source*. The class of the output matrix, A, is specified by *output*. Example: 'int8=>char'
**source*	The input values and the output matrix, A, are of the class specified by *source*. For bit*n* or ubit*n* precisions, the output has the smallest class that can contain the input. Example: '*ubit18' This is equivalent to 'ubit18=>uint32'
*N*source* or *N*source=>output*	Read *N* values before skipping the number of bytes specified by the skip argument. Example: '4*int8'

The following table shows possible values for *source* and *output*.

Value Type	Precision	Bits (Bytes)
Integers, unsigned	'uint'	32 (4)
	'uint8'	8 (1)

Value Type	Precision	Bits (Bytes)
	'uint16'	16 (2)
	'uint32'	32 (4)
	'uint64'	64 (8)
	'uchar'	8 (1)
	'unsigned char'	8 (1)
	'ushort'	16 (2)
	'ulong'	32 (4)
	'ubit*n*'	$1 \le n \le 64$
Integers, signed	'int'	32 (4)
	'int8'	8 (1)
	'int16'	16 (2)
	'int32'	32 (4)
	'int64'	64 (8)
	'integer*1'	8 (1)
	'integer*2'	16 (2)
	'integer*4'	32 (4)
	'integer*8'	64 (8)
	'schar'	8 (1)
	'signed char'	8 (1)
	'short'	16 (2)
	'long'	32 (4)
	'bit*n*'	$1 \le n \le 64$
Floating-point numbers	'single'	32 (4)
	'double'	64 (8)
	'float'	32 (4)
	'float32'	32 (4)
	'float64'	64 (8)
	'real*4'	32 (4)

Value Type	Precision	Bits (Bytes)
	`'real*8'`	`64 (8)`
Characters	`'char*1'`	`8 (1)`
	`'char'`	Depends on the encoding scheme associated with the file. Set encoding with `fopen`.

For most values of *`source`*, if `fread` reaches the end of the file before reading a complete value, it does not return a result for the final value. However, if *`source`* is `bit`*n* or `ubit`*n*, then `fread` returns a partial result for the final value.

Note To preserve `NaN` and `Inf` values in MATLAB, read and write data of class `double` or `single`.

Data Types: `char` | `string`

`skip` — Number of bytes to skip
0 (default) | scalar

Number of bytes to skip after reading each value, specified as a scalar. If you specify a `precision` of `bit`*n* or `ubit`*n*, specify `skip` in bits.

Use the `skip` argument to read data from noncontiguous fields in fixed-length records.

`machinefmt` — Order for reading bytes
`'n'` (default) | `'b'` | `'l'` | `'s'` | `'a'` | ...

Order for reading bytes in the file, specified as a character vector or a string scalar. Specify `machinefmt` as one of the values in the table that follows. For `bit`*n* and `ubit`*n* precisions, `machinefmt` specifies the order for reading bits within a byte, but the order for reading bytes remains your system byte ordering.

`'n'` or `'native'`	Your system byte ordering (default)
`'b'` or `'ieee-be'`	Big-endian ordering
`'l'` or `'ieee-le'`	Little-endian ordering

`'s'` or `'ieee-be.l64'`	Big-endian ordering, 64-bit long data type
`'a'` or `'ieee-le.l64'`	Little-endian ordering, 64-bit long data type

By default, all currently supported platforms use little-endian ordering for new files. Existing binary files can use either big-endian or little-endian ordering.

Data Types: `char` | `string`

Output Arguments

`A` — File data
column vector | matrix

File data, returned as a column vector. If you specified the `sizeA` argument, then `A` is a matrix of the specified size. Data in `A` is class `double` unless you specify a different class in the `precision` argument.

`count` — Number of characters read
scalar

Number of characters read, returned as a scalar value.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The input argument `precision` must be a constant.
- The `source` and `output` classes that `precision` specifies cannot have these values: `'long'`, `'ulong'`, `'unsigned long'`, `'bit*n*'`, or `'ubit*n*'`.
- You cannot use the `machinefmt` input.
- If the `source` or `output` that `precision` specifies is a C type, for example, `int`, then the target and production sizes for that type must:

  - Match.
  - Map directly to a MATLAB type.
- The `source` type that `precision` specifies must map directly to a C type on the target hardware.
- If the `fread` call reads the entire file, then all of the data must fit in the largest array available for code generation.
- If `sizeA` is not constant or contains a nonfinite element, then dynamic memory allocation is required.
- The code generator for the `fread` function treats the `char` value for `source` or `output` as a signed 8-bit integer. Use values between `0` and `127` only.
- The generated code does not report file read errors. Therefore, you must write your own file read error handling in your MATLAB code. In your error handling code, consider checking that the number of bytes read matches the number of bytes that you requested. For example:

```
...
N = 100;
[vals, numRead] = fread(fid, N, '*double');
if numRead ~= N
    % fewer elements read than expected
end
...
```

See Also

`fclose` | `fgetl` | `fopen` | `fprintf` | `fscanf` | `fseek` | `ftell` | `fwrite`

Topics

"Import Binary Data with Low-Level I/O"

Introduced before R2006a

fread (serial)

(Not recommended) Read binary data from device

Note This `serial` object function is not recommended. Use `serialport` object functions instead. See “Compatibility Considerations”.

Syntax

```
A = fread(obj)
A = fread(obj,size,'precision')
[A,count] = fread(...)
[A,count,msg] = fread(...)
```

Description

`A = fread(obj)` and `A = fread(obj,size)` read binary data from the device connected to the serial port object, `obj`, and returns the data to `A`. The maximum number of values to read is specified by `size`. If `size` is not specified, the maximum number of values to read is determined by the object's `InputBufferSize` property. Valid options for `size` are:

`n`	Read at most `n` values into a column vector.
`[m,n]`	Read at most m-by-n values filling an m–by–n matrix in column order.

`size` cannot be `inf`, and an error is returned if the specified number of values cannot be stored in the input buffer. You specify the size, in bytes, of the input buffer with the `InputBufferSize` property. A value is defined as a byte multiplied by the *`precision`* (see below).

`A = fread(obj,size,'precision')` reads binary data with precision specified by *`precision`*.

`precision` controls the number of bits read for each value and the interpretation of those bits as integer, floating-point, or character values. If *`precision`* is not specified,

`uchar` (an 8-bit unsigned character) is used. By default, numeric values are returned in double-precision arrays. The supported values for *`precision`* are listed below in Tips on page 1-4637.

`[A,count] = fread(...)` returns the number of values read to `count`.

`[A,count,msg] = fread(...)` returns a warning message to `msg` if the read operation was unsuccessful.

Tips

Before you can read data from the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read, each time `fread` is issued.

Rules for Completing a Binary Read Operation

A read operation with `fread` blocks access to the MATLAB command line until:

- The specified number of values are read.
- The time specified by the `Timeout` property passes.

Note The `Terminator` property is not used for binary read operations.

Supported Precisions

The supported values for *`precision`* are listed below.

Data Type	Precision	Interpretation
Character	`uchar`	8-bit unsigned character

Data Type	Precision	Interpretation
	`schar`	8-bit signed character
	`char`	8-bit signed or unsigned character
Integer	`int8`	8-bit integer
	`int16`	16-bit integer
	`int32`	32-bit integer
	`uint8`	8-bit unsigned integer
	`uint16`	16-bit unsigned integer
	`uint32`	32-bit unsigned integer
	`short`	16-bit integer
	`int`	32-bit integer
	`long`	32- or 64-bit integer
	`ushort`	16-bit unsigned integer
	`uint`	32-bit unsigned integer
	`ulong`	32- or 64-bit unsigned integer
Floating-point	`single`	32-bit floating point
	`float32`	32-bit floating point
	`float`	32-bit floating point
	`double`	64-bit floating point
	`float64`	64-bit floating point

Compatibility Considerations

`serial` object interface is not recommended

Not recommended starting in R2019b

Use of this function with a `serial` object is not recommended. To access a serial port device, use a `serialport` object with its functions and properties instead.

See “Transition Your Code to serialport Interface” for more information about using the recommended functionality.

See Also

Functions
`fgetl` | `fgets` | `fopen` | `fscanf`

Introduced before R2006a

freeBoundary

Class: `TriRep`

(Not recommended) Facets referenced by only one simplex

Note `freeBoundary(TriRep)` is not recommended. Use `freeBoundary(triangulation)` instead.

`TriRep` is not recommended. Use `triangulation` instead.

Syntax

```
FF = freeBoundary(TR)
[FF XF] = freeBoundary(TR)
```

Description

`FF = freeBoundary(TR)` returns a matrix `FF` that represents the free boundary facets of the triangulation. A facet is on the free boundary if it is referenced by only one simplex (triangle/tetrahedron, etc). `FF` is of size `m`-by-`n`, where m is the number of boundary facets and n is the number of vertices per facet. The vertices of the facets index into the array of points representing the vertex coordinates `TR.X`. The array `FF` could be empty as in the case of a triangular mesh representing the surface of a sphere.

`[FF XF] = freeBoundary(TR)` returns a matrix of free boundary facets

Input Arguments

`TR`	Triangulation representation.

Output Arguments

FF	FF that has vertices defined in terms of a compact array of coordinates XF.
XF	XF is of size m-by-ndim where m is the number of free facets, and ndim is the dimension of the space where the triangulation resides

Examples

Example 1

Use TriRep to compute the boundary triangulation of an imported triangulation.

Load a 3-D triangulation:

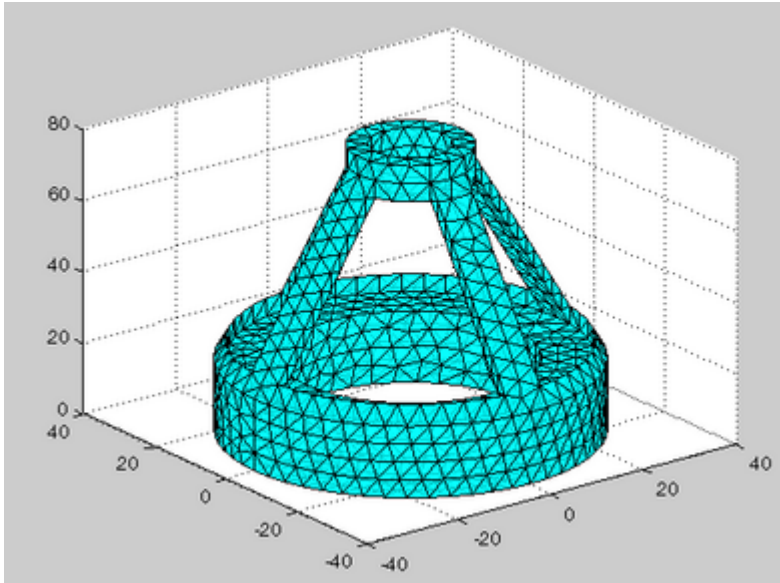
```
load tetmesh;
trep = TriRep(tet, X);
```

Compute the boundary triangulation:

```
 [tri xf] = freeBoundary(trep);
```

Plot the boundary triangulation:

```
trisurf(tri, xf(:,1),xf(:,2),xf(:,3), ...
     'FaceColor','cyan', 'FaceAlpha', 0.8);
```



Example 2

Perform a direct query of a 2-D triangulation created with `DelaunayTri`.

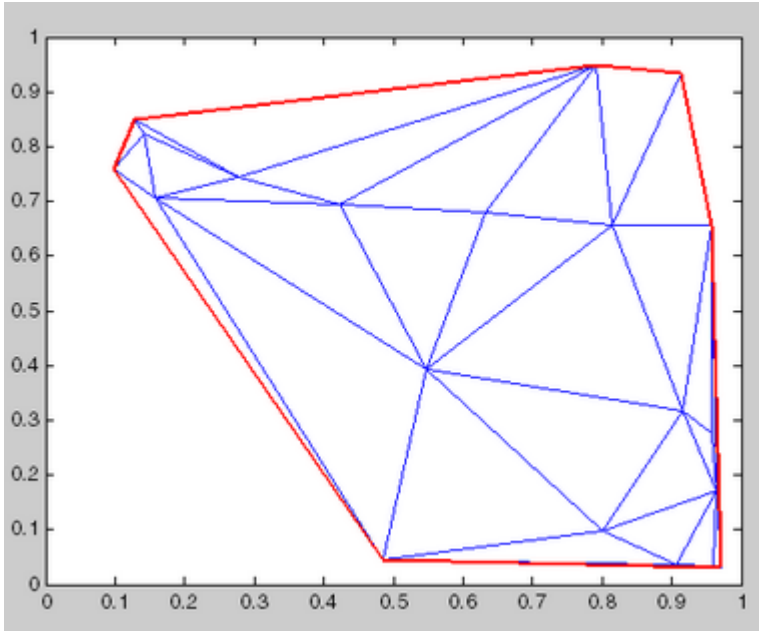
Plot the mesh:

```
x = rand(20,1);
y = rand(20,1);
dt = DelaunayTri(x,y);
fe = freeBoundary(dt)';
triplot(dt);
hold on;
```

Display the free boundary edges in red:

```
plot(x(fe), y(fe), '-r', 'LineWidth',2) ;
hold off;
```

In this instance the free edges correspond to the convex hull of `(x, y)`.



More About

Simplex

A simplex is a triangle/tetrahedron or higher-dimensional equivalent.

Facet

A facet is an edge of a triangle or a face of a tetrahedron.

See Also

`convexHull` | `delaunayTriangulation` | `faceNormal` | `featureEdges` | `triangulation`

freqspace

Frequency spacing for frequency response

Syntax

```
[f1,f2] = freqspace(n)
[f1,f2] = freqspace([m n])
[x1,y1] = freqspace(...,'meshgrid')
f = freqspace(N)
f = freqspace(N,'whole')
```

Description

`freqspace` returns the implied frequency range for equally spaced frequency responses. `freqspace` is useful when creating desired frequency responses for various one- and two-dimensional applications.

`[f1,f2] = freqspace(n)` returns the two-dimensional frequency vectors `f1` and `f2` for an `n`-by-`n` matrix.

For `n` odd, both `f1` and `f2` are `[-n+1:2:n-1]/n`.

For `n` even, both `f1` and `f2` are `[-n:2:n-2]/n`.

`[f1,f2] = freqspace([m n])` returns the two-dimensional frequency vectors `f1` and `f2` for an `m`-by-`n` matrix.

`[x1,y1] = freqspace(...,'meshgrid')` is equivalent to

```
[f1,f2] = freqspace(...);
[x1,y1] = meshgrid(f1,f2);
```

`f = freqspace(N)` returns the one-dimensional frequency vector `f` assuming `N` evenly spaced points around the unit circle. For `N` even or odd, `f` is `(0:2/N:1)`. For `N` even, `freqspace` therefore returns `(N+2)/2` points. For `N` odd, it returns `(N+1)/2` points.

`f = freqspace(N,'whole')` returns `N` evenly spaced points around the whole unit circle. In this case, `f` is `0:2/N:2*(N-1)/N`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`meshgrid`

Introduced before R2006a

frewind

Move file position indicator to beginning of open file

Syntax

```
frewind(fileID)
```

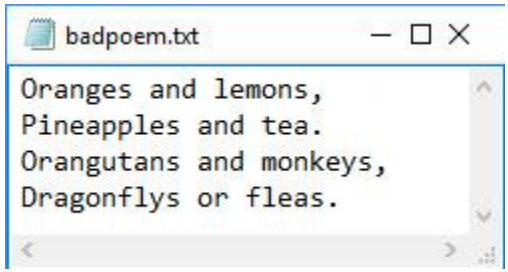
Description

`frewind(fileID)` sets the file position indicator to the beginning of a file.

Examples

Reset File Position Indicator

Open the following file, perform a read operation, and then move the file position indicator back to the beginning of the file.



Use `fopen` to open the file, and then use `fgetl` to read the first two lines.

```
fid = fopen('badpoem.txt');
tline1 = fgetl(fid)  % read first line
```

```
tline1 =
'Oranges and lemons,'
```

```
tline2 = fgetl(fid)  % read second line
```

```
tline2 =
'Pineapples and tea.'
```

The previous two read operations moved the position indicator to the beginning of line 3 in the poem. As a result, the next read operation using `fgetl` returns line 3.

```
tline3 = fgetl(fid)
```

```
tline3 =
'Orangutans and monkeys,'
```

To reread the first line of the file. Reset the position indicator using the `frewind` function, and then perform the read operation.

```
frewind(fid)
fgetl(fid)
```

```
ans =
'Oranges and lemons,'
```

Close the file.

```
fclose(fid);
```

Input Arguments

`fileID` — File identifier
integer

File identifier of an open file, specified as an integer. Before using `frewind` you must use `fopen` to open the file and obtain its `fileID`.

Data Types: `double`

Tips

If the file is on a tape device and the rewind operation fails, `frewind` does not return an error message.

Alternatives

`frewind(fileID)` is equivalent to:

```
fseek(fileID, 0, 'bof');
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`fclose` | `feof` | `ferror` | `fopen` | `fprintf` | `fread` | `fscanf` | `fseek` | `ftell` | `fwrite`

Introduced before R2006a

fscanf

Read data from text file

Syntax

```
A = fscanf(fileID,formatSpec)
A = fscanf(fileID,formatSpec,sizeA)
[A,count] = fscanf( ___ )
```

Description

`A = fscanf(fileID,formatSpec)` reads data from an open text file into column vector `A` and interprets values in the file according to the format specified by `formatSpec`. The `fscanf` function reapplies the format throughout the entire file and positions the file pointer at the end-of-file marker. If `fscanf` cannot match `formatSpec` to the data, it reads only the portion that matches and stops processing.

The text file is indicated by the file identifier, `fileID`. Use `fopen` to open the file, specify the character encoding, and obtain the `fileID` value. When you finish reading, close the file by calling `fclose(fileID)`.

`A = fscanf(fileID,formatSpec,sizeA)` reads file data into an array, `A`, with dimensions, `sizeA`, and positions the file pointer after the last value read. `fscanf` populates `A` in column order. `sizeA` must be a positive integer or have the form `[m n]`, where `m` and `n` are positive integers.

`[A,count] = fscanf( ___ )` additionally returns the number of fields that `fscanf` reads into `A`. For numeric data, this is the number of values read. You can use this syntax with any of the input arguments of the previous syntaxes.

Examples

Read File Contents into Column Vector

Create a sample text file that contains floating-point numbers.

```
x = 100*rand(8,1);
fileID = fopen('nums1.txt','w');
fprintf(fileID,'%4.4f\n',x);
fclose(fileID);
```

View the contents of the file.

```
type nums1.txt
```

```
81.4724
90.5792
12.6987
91.3376
63.2359
9.7540
27.8498
54.6882
```

Open the file for reading, and obtain the file identifier, `fileID`.

```
fileID = fopen('nums1.txt','r');
```

Define the format of the data to read. Use `'%f'` to specify floating-point numbers.

```
formatSpec = '%f';
```

Read the file data, filling output array, `A`, in column order. `fscanf` reapplies the format, `formatSpec`, throughout the file.

```
A = fscanf(fileID,formatSpec)
```

```
A = 8×1

   81.4724
   90.5792
   12.6987
   91.3376
   63.2359
    9.7540
   27.8498
   54.6882
```

A is a column vector containing data from the file.

Close the file.

```
fclose(fileID);
```

Read File Contents into Array

Create a sample text file that contains integers and floating-point numbers.

```
x = 1:1:5;
y = [x;rand(1,5)];
fileID = fopen('nums2.txt','w');
fprintf(fileID,'%d %4.4f\n',y);
fclose(fileID);
```

View the contents of the file.

```
type nums2.txt
```

```
1 0.8147
2 0.9058
3 0.1270
4 0.9134
5 0.6324
```

Open the file for reading, and obtain the file identifier, `fileID`.

```
fileID = fopen('nums2.txt','r');
```

Define the format of the data to read and the shape of the output array.

```
formatSpec = '%d %f';
sizeA = [2 Inf];
```

Read the file data, filling output array, A, in column order. `fscanf` reuses the format, `formatSpec`, throughout the file.

```
A = fscanf(fileID,formatSpec,sizeA)
```

```
A = 2×5

    1.0000    2.0000    3.0000    4.0000    5.0000
```

```
    0.8147    0.9058    0.1270    0.9134    0.6324
```

```
fclose(fileID);
```

Transpose the array so that A matches the orientation of the data in the file.

```
A = A'
```

```
A = 5×2

    1.0000    0.8147
    2.0000    0.9058
    3.0000    0.1270
    4.0000    0.9134
    5.0000    0.6324
```

Skip Specific Characters in File

Skip specific characters in a sample file, and return only numeric data.

Create a sample text file containing temperature values.

```
str = '78°C 72°C 64°C 66°C 49°C';
fileID = fopen('temperature.dat','w');
fprintf(fileID,'%s',str);
fclose(fileID);
```

Read the numbers in the file, skipping the text, °C. Also return the number of values that `fscanf` reads. The extended ASCII code 176 represents the degree sign.

```
fileID = fopen('temperature.dat','r');
degrees = char(176);
[A,count] = fscanf(fileID, ['%d' degrees 'C'])
fclose(fileID);
```

```
A =

    78
    72
    64
    66
```

```
    49


count =

     5
```

A is a vector containing the numeric values in the file. `count` indicates that `fscanf` read five values.

Input Arguments

`fileID` — File identifier

integer

File identifier of an open text file, specified as an integer. Before reading a file with `fscanf`, you must use `fopen` to open the file and obtain the `fileID`.

Data Types: `double`

`formatSpec` — Format of data fields

character vector | string scalar

Format of the data fields in the file, specified as a character vector or string scalar of one or more conversion specifiers. When `fscanf` reads a file, it attempts to match the data to the format specified by `formatSpec`.

Numeric Fields

This table lists available conversion specifiers for numeric inputs. `fscanf` converts values to their decimal (base 10) representation.

Numeric Field Type	Conversion Specifier	Details
Integer, signed	`%d`	Base 10

Numeric Field Type	Conversion Specifier	Details
	`%i`	The values in the file determine the base: • The default is base 10. • If the initial digits are `0x` or `0X`, then the values are hexadecimal (base 16). • If the initial digit is `0`, then values are octal (base 8).
	`%ld` or `%li`	64-bit values, base 10, 8, or 16
Integer, unsigned	`%u`	Base 10
	`%o`	Base 8 (octal)
	`%x`	Base 16 (hexadecimal)
	`%lu`, `%lo`, `%lx`	64-bit values, base 10, 8, or 16
Floating-point number	`%f`	Floating-point fields can contain any of the following (not case sensitive): `Inf`, `-Inf`, `NaN`, or `-NaN`.
	`%e`	
	`%g`	

Character Fields

This table lists available conversion specifiers for character inputs.

Character Field Type	Conversion Specifier	Description
Character vector or string scalar	`%s`	Read all characters excluding white spaces.
	`%c`	Read any single character, including white space. To read multiple characters at a time, specify field width.
Pattern-matching	`%[...]`	Read only characters in the brackets up to the first nonmatching character or white space. Example: `%[mus]` reads `'summer '` as `'summ'`.

If `formatSpec` contains a combination of numeric and character specifiers, then `fscanf` converts each character to its numeric equivalent. This conversion occurs even when the format explicitly skips all numeric values (for example, `formatSpec` is `'%*d %s'`).

Optional Operators

- Fields and Characters to Ignore

  `fscanf` reads all numeric values and characters in your file in sequence, unless you tell it to ignore a particular field or a portion of a field. To skip fields, insert an asterisk (`*`) after the percent sign (`%`). For example, to skip integers, specify `%*d`.

- Field Width

  To specify the maximum number of digits or text characters to read at a time, insert a number after the percent character. For example, `%10c` reads up to 10 characters at a time, including white space. `%4f` reads up to 4 digits at a time, including the decimal point.

- Literal Text to Ignore

  `fscanf` ignores specified text appended to the `formatSpec` conversion specifier.

  Example: `Level%u` reads `'Level1'` as `1`.

  Example: `%uStep` reads `'2Step'` as `2`.

`sizeA` — Dimensions of output array

`Inf` (default) | integer | two-element row vector

Dimensions of the output array, `A`, specified as `Inf`, an integer, or a two-element row vector.

Form of the `sizeA` Input	Description
`Inf`	Read to the end of the file. For numeric data, the output, `A`, is a column vector. For text data, `A` is a character vector.
n	Read at most *n* numeric values or character fields. For numeric data, the output, `A`, is a column vector. For text data, `A`, is a character vector.

Form of the sizeA Input	Description
[*m*,*n*]	Read at most *m**n numeric values or character fields. *n* can be Inf, but *m* cannot. The output, A, is m-by-n, filled in column order.

Output Arguments

A — File data
column vector | matrix | character vector | character array

File data, returned as a column vector, matrix, character vector or character array. The class and size of A depend on the formatSpec input:

- If formatSpec contains only numeric specifiers, then A is numeric. If you specify the sizeA argument, then A is a matrix of the specified size. Otherwise, A is a column vector. If the input contains fewer than sizeA values, then fscanf pads A with zeros.
  - If formatSpec contains only 64-bit signed integer specifiers, then A is of class int64.
  - If formatSpec contains only 64-bit unsigned integer specifiers, then A is of class uint64.
  - Otherwise, A is of class double.
- If formatSpec contains only character or text specifiers (%c or %s), then A is a character array. If you specify sizeA and the input contains fewer characters, then fscanf pads A with char(0).
- If formatSpec contains a combination of numeric and character specifiers, then A is numeric, of class double, and fscanf converts each text characters to its numeric equivalent. This occurs even when formatSpec explicitly skips all numeric fields (for example, formatSpec is '%*d %s').
- If MATLAB cannot match the file data to formatSpec, then A can be numeric or a character array. The class of A depends on the values that fscanf reads before it stops processing.

count — Number of characters read
scalar

Number of characters read, returned as a scalar value.

Tips

- Format specifiers for the reading functions `sscanf` and `fscanf` differ from the formats for the writing functions `sprintf` and `fprintf`. The reading functions do not support a precision field. The width field specifies a minimum for writing but a maximum for reading.

Algorithms

MATLAB reads characters using the encoding scheme associated with the file. You specify the encoding when you open the file using the `fopen` function.

See Also

`fgetl` | `fgets` | `fopen` | `fprintf` | `fread` | `sscanf` | `textscan`

Topics

"Import Text Data Files with Low-Level I/O"

Introduced before R2006a

fscanf (serial)

(Not recommended) Read ASCII data from device, and format as text

Note This `serial` object function is not recommended. Use `serialport` object functions instead. See “Compatibility Considerations”.

Syntax

```
A = fscanf(obj)
A = fscanf(obj,'format')
A = fscanf(obj,'format',size)
[A,count] = fscanf(...)
[A,count,msg] = fscanf(...)
```

Description

`A = fscanf(obj)` reads ASCII data from the device connected to the serial port object, `obj`, and returns it to `A`. The data is converted to text using the `%c` format. For binary data, use `fread`.

`A = fscanf(obj,'format')` reads data and converts it according to *`format`*. *`format`* is a C language conversion specification. Conversion specifications involve the `%` character and the conversion characters d, i, o, u, x, X, f, e, E, g, G, c, and s. Refer to the `sscanf` file I/O format specifications or a C manual for more information.

`A = fscanf(obj,'format',size)` reads the number of values specified by `size`. Valid options for `size` are:

`n`	Read at most `n` values into a column vector.
`[m,n]`	Read at most m-by-n values filling an m–by–n matrix in column order.

`size` cannot be `inf`, and an error is returned if the specified number of values cannot be stored in the input buffer. If `size` is not of the form `[m,n]`, and a character conversion is specified, then `A` is returned as a row vector. You specify the size, in bytes, of the input buffer with the `InputBufferSize` property. An ASCII value is one byte.

`[A,count] = fscanf(...)` returns the number of values read to `count`.

`[A,count,msg] = fscanf(...)` returns a warning message to `msg` if the read operation did not complete successfully.

Examples

Create the serial port object `s` and connect `s` to a Tektronix TDS 210 oscilloscope, which is displaying sine wave. This example works on a Windows platform.

```
s = serial('COM1');
fopen(s)
```

Use the `fprintf` function to configure the scope to measure the peak-to-peak voltage of the sine wave, return the measurement type, and return the peak-to-peak voltage.

```
fprintf(s,'MEASUREMENT:IMMED:TYPE PK2PK')
fprintf(s,'MEASUREMENT:IMMED:TYPE?')
fprintf(s,'MEASUREMENT:IMMED:VALUE?')
```

Because the default value for the `ReadAsyncMode` property is `continuous`, data associated with the two query commands is automatically returned to the input buffer.

```
s.BytesAvailable
```

```
ans =
    21
```

Use `fscanf` to read the measurement type. The operation will complete when the first terminator is read.

```
meas = fscanf(s)
```

```
meas =
PK2PK
```

Use `fscanf` to read the peak-to-peak voltage as a floating-point number, and exclude the terminator.

```
pk2pk = fscanf(s,'%e',14)
```

```
pk2pk =
    2.0200
```

Disconnect `s` from the scope, and remove `s` from memory and the workspace.

```
fclose(s)
delete(s)
clear s
```

Tips

Before you can read data from the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read – including the terminator – each time `fscanf` is issued.

Rules for Completing a Read Operation with fscanf

A read operation with `fscanf` blocks access to the MATLAB command line until:

- The terminator specified by the `Terminator` property is read.
- The time specified by the `Timeout` property passes.
- The number of values specified by `size` is read.
- The input buffer is filled (unless `size` is specified)

Compatibility Considerations

`serial` object interface is not recommended

Not recommended starting in R2019b

Use of this function with a `serial` object is not recommended. To access a serial port device, use a `serialport` object with its functions and properties instead.

See “Transition Your Code to serialport Interface” for more information about using the recommended functionality.

See Also

Functions
`fgetl` | `fgets` | `fopen` | `fread` | `textscan`

Introduced before R2006a

fseek

Move to specified position in file

Syntax

```
fseek(fileID, offset, origin)
status = fseek( ___ )
```

Description

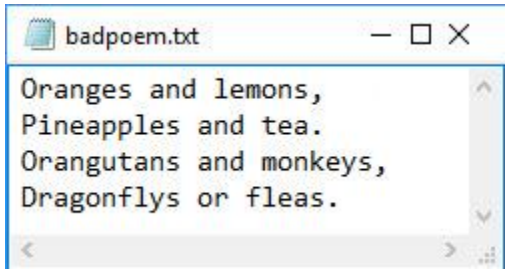
`fseek(fileID, offset, origin)` sets the file position indicator `offset` bytes from `origin` in the specified file.

`status = fseek( ___ )` returns `0` when the operation is successful. Otherwise, `fseek` returns `-1`. Use any of the previous input argument combinations.

Examples

Move to New Position in File

Open the following `badpoem.txt` file and perform read operations (which advance the position pointer) and then use `seek` to move to a new position in the file.



Use `fopen` to open the file. Then, use `ftell` to query the current position.

```
fid = fopen('badpoem.txt');
ftell(fid)
```

```
ans = 0
```

Read the first three lines and query the position in the file after each read. Use `fgetl` to read and `fseek` to examine the current position after the read operation.

```
tline1 = fgetl(fid)  % read the first line
```

```
tline1 =
'Oranges and lemons,'
```

```
ftell(fid)
```

```
ans = 20
```

Read the second line and examine the current position.

```
tline2 = fgetl(fid)  % read the second line
```

```
tline2 =
'Pineapples and tea.'
```

```
ftell(fid)
```

```
ans = 40
```

Read the third line and examine the current position.

```
tline3 = fgetl(fid)  % read the third line
```

```
tline3 =
'Orangutans and monkeys,'
```

```
ftell(fid)
```

```
ans = 64
```

To read line 2, set the position in the file to point to the beginning of line 2. Use `fseek` to set the position, and then perform a read operation.

```
fseek(fid,20,'bof');
fgetl(fid)
```

```
ans =
'Pineapples and tea.'
```

Close the file.

```
fclose(fid);
```

Input Arguments

`fileID` — File identifier
integer

File identifier of an open file, specified as an integer. Before using `fseek`, you must use `fopen` to open the file and obtain its `fileID`.

Data Types: `double`

`offset` — Number of bytes
integer

Number of bytes to move from `origin`, specified as an integer. The value of `offset` can be positive, negative, or zero.

Data Types: `double`

`origin` — Starting location
integer | character vector | string

Starting location in the file, specified as a character vector, string scalar, or a scalar number.

`'bof'` or `-1`	Beginning of file
`'cof'` or `0`	Current position in file
`'eof'` or `1`	End of file

Data Types: `double` | `char` | `string`

Tips

- If a file has `n` bytes of data, then those `n` bytes are in positions `0` through `n-1`.

Alternatives

To move to the beginning of a file, call

```
frewind(fileID)
```

This call is identical to

```
fseek(fileID, 0, 'bof')
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- When the MATLAB behavior differs from the C compiler behavior, the generated code matches the C compiler behavior. Examples include:
  - Seeking past the end of a file.
  - Seeking away from the end of a file opened with append access.
- The offset is passed to the C run-time environment as a signed long data type. Therefore, the offset value must fit in the long data type on the target hardware.

See Also

`fclose` | `feof` | `ferror` | `fopen` | `fprintf` | `fread` | `frewind` | `fscanf` | `ftell` | `fwrite`

Topics

"Import Binary Data with Low-Level I/O"

Introduced before R2006a

fsurf

Plot 3-D surface

Syntax

```
fsurf(f)
fsurf(f,xyinterval)

fsurf(funx,funy,funz)
fsurf(funx,funy,funz,uvinterval)

fsurf( ___ ,LineSpec)
fsurf( ___ ,Name,Value)
fsurf(ax, ___ )
fs = fsurf( ___ )
```

Description

`fsurf(f)` creates a surface plot of the function `z = f(x,y)` over the default interval `[-5 5]` for `x` and `y`.

`fsurf(f,xyinterval)` plots over the specified interval. To use the same interval for both `x` and `y`, specify `xyinterval` as a two-element vector of the form `[min max]`. To use different intervals, specify a four-element vector of the form `[xmin xmax ymin ymax]`.

`fsurf(funx,funy,funz)` plots the parametric surface defined by `x = funx(u,v)`, `y = funy(u,v)`, `z = funz(u,v)` over the default interval `[-5 5]` for `u` and `v`.

`fsurf(funx,funy,funz,uvinterval)` plots over the specified interval. To use the same interval for both `u` and `v`, specify `uvinterval` as a two-element vector of the form `[min max]`. To use different intervals, specify a four-element vector of the form `[umin umax vmin vmax]`.

`fsurf( ___ ,LineSpec)` sets the line style, marker symbol, and surface color. For example, `'-r'` specifies red lines. Use this option after any of the previous input argument combinations.

`fsurf( ___ ,Name,Value)` specifies surface properties using one or more name-value pair arguments. Use this option after any of the input argument combinations in the previous syntaxes.

`fsurf(ax, ___ )` plots into the axes specified by `ax` instead of the current axes (`gca`).

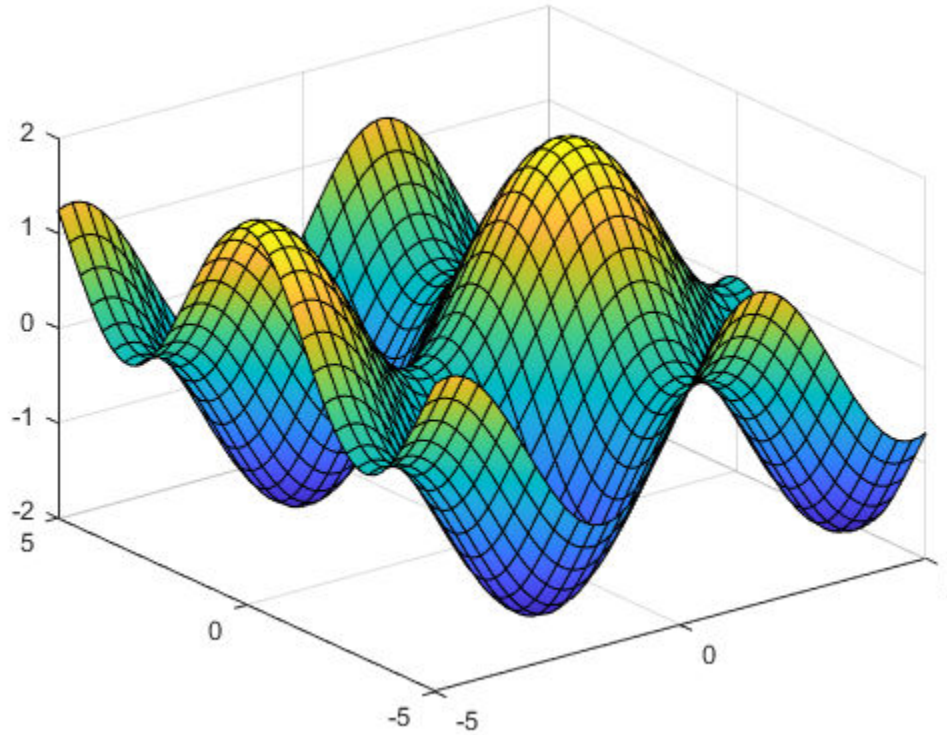
`fs = fsurf( ___ )` returns a `FunctionSurface` object or `ParameterizedFunctionSurface` object, depending on the inputs. Use `fs` to query and modify properties of a specific surface. For a list of properties, see FunctionSurface or ParameterizedFunctionSurface.

Examples

3-D Surface Plot of Expression

Plot the expression $\sin(x) + \cos(y)$ over the default interval $-5 < x < 5$ and $-5 < y < 5$.

```
fsurf(@(x,y) sin(x)+cos(y))
```



Specify Interval of Surface Plot and Plot Piecewise Expression

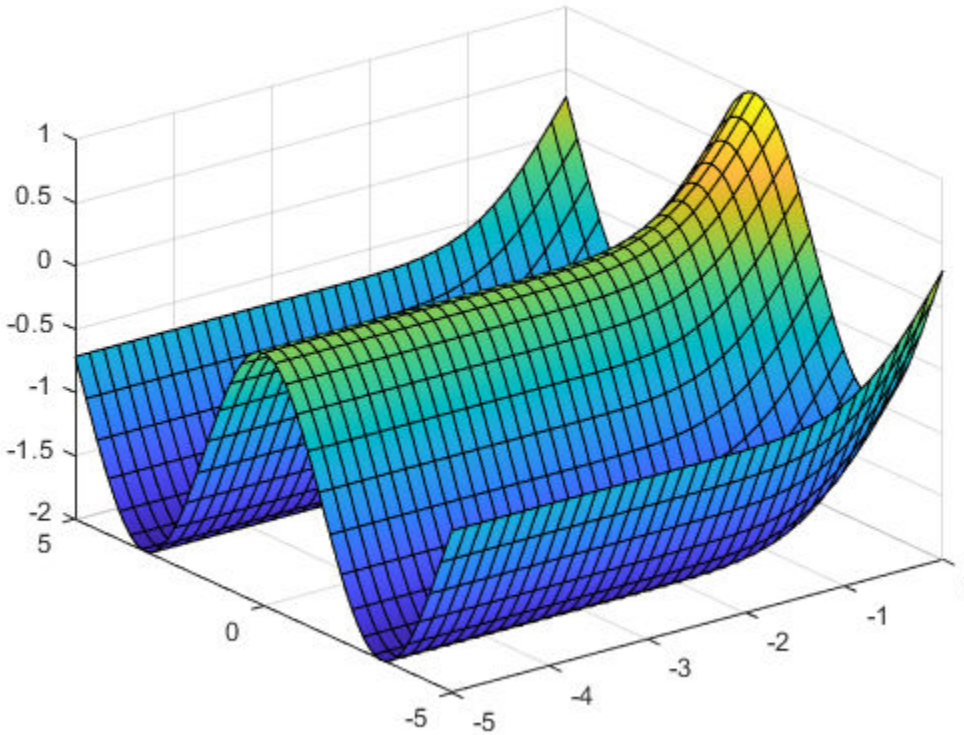
Plot the piecewise expression

$$\begin{array}{ll} \mathrm{erf}(x) + \cos(y) & -5 < x < 0 \\ \sin(x) + \cos(y) & 0 < x < 5 \end{array}$$

over $-5 < y < 5$.

Specify the plotting interval as the second input argument of `fsurf`. When you plot multiple surfaces over different intervals in the same axes, the axis limits adjust to include all the data.

```
f1 = @(x,y) erf(x)+cos(y);
fsurf(f1,[-5 0 -5 5])
hold on
f2 = @(x,y) sin(x)+cos(y);
fsurf(f2,[0 5 -5 5])
hold off
```



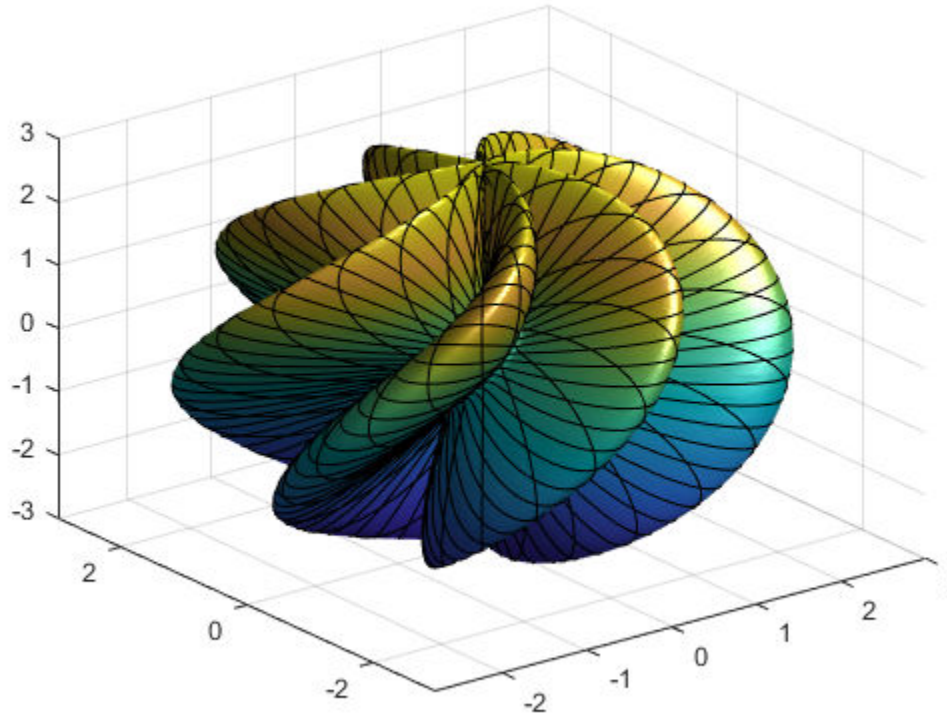
Parameterized Surface Plot

Plot the parameterized surface

$$
\begin{aligned}
x &= r\cos(u)\sin(v) \\
y &= r\sin(u)\sin(v) \\
z &= r\cos(v) \\
\text{where} \quad r &= 2 + \sin(7u + 5v)
\end{aligned}
$$

for $0 < u < 2\pi$ and $0 < v < \pi$. Add light to the surface using `camlight`.

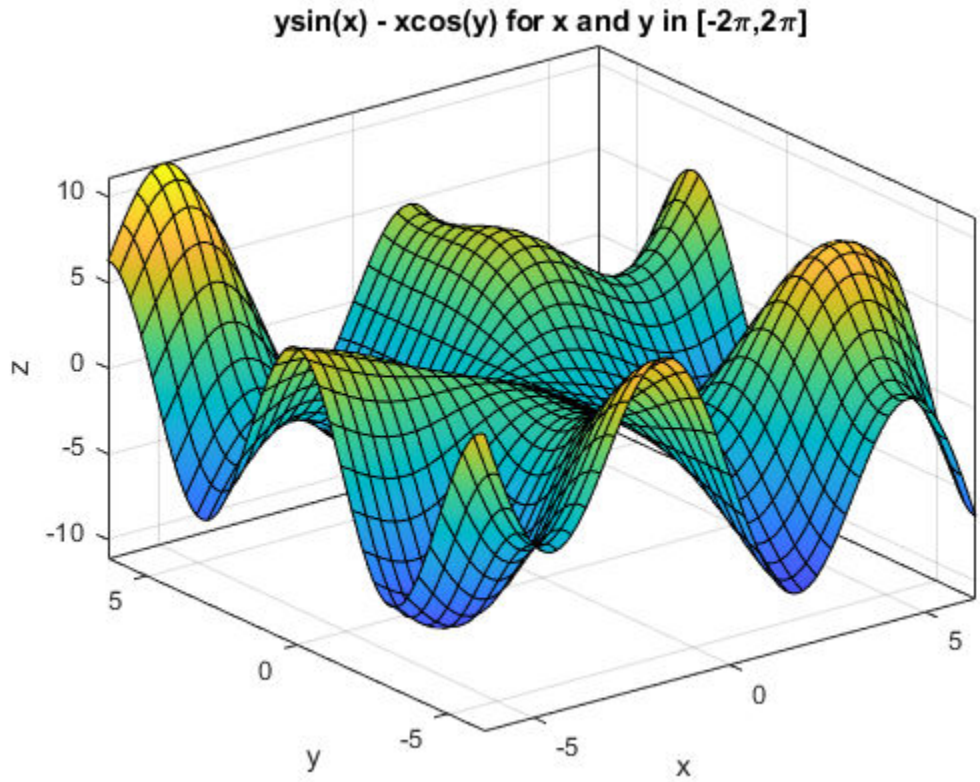
```
r = @(u,v) 2 + sin(7.*u + 5.*v);
funx = @(u,v) r(u,v).*cos(u).*sin(v);
funy = @(u,v) r(u,v).*sin(u).*sin(v);
funz = @(u,v) r(u,v).*cos(v);
fsurf(funx,funy,funz,[0 2*pi 0 pi])
camlight
```



Add Title and Axis Labels and Format Ticks

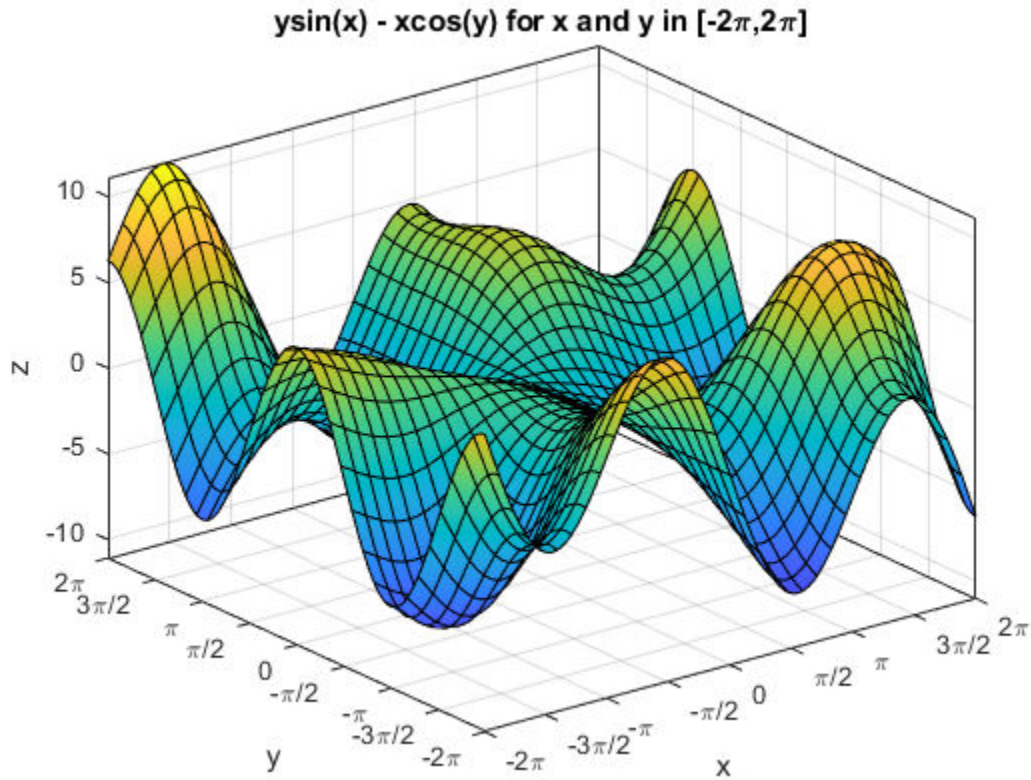
For x and y from -2π to 2π, plot the 3-D surface $y\sin(x) - x\cos(y)$. Add a title and axis labels and display the axes outline.

```
fsurf(@(x,y) y.*sin(x)-x.*cos(y),[-2*pi 2*pi])
title('ysin(x) - xcos(y) for x and y in [-2\pi,2\pi]')
xlabel('x');
ylabel('y');
zlabel('z');
box on
```



Set the *x*-axis tick values and associated labels using the `XTickLabel` and `XTick` properties of axes object. Access the axes object using `gca`. Similarly, set the *y*-axis tick values and associated labels.

```
ax = gca;
ax.XTick = -2*pi:pi/2:2*pi;
ax.XTickLabel = {'-2\pi','-3\pi/2','-\pi','-\pi/2','0','\pi/2','\pi','3\pi/2','2\pi'};

ax.YTick = -2*pi:pi/2:2*pi;
ax.YTickLabel = {'-2\pi','-3\pi/2','-\pi','-\pi/2','0','\pi/2','\pi','3\pi/2','2\pi'};
```

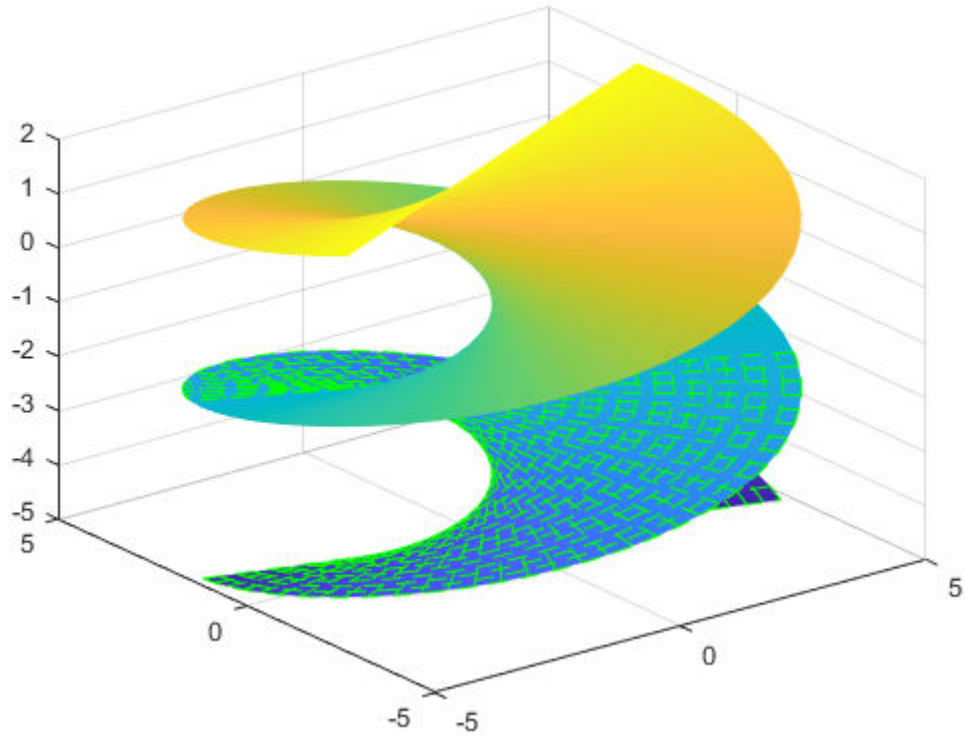


Specify Surface Properties

Plot the parametric surface $x = u\sin(v)$, $y = -u\cos(v)$, $z = v$ with different line styles for different values of v. For $-5 < v < -2$, use a dashed green line for the surface edges. For $-2 < v < 2$, turn off the edges by setting the `EdgeColor` property to `'none'`.

```
funx = @(u,v) u.*sin(v);
funy = @(u,v) -u.*cos(v);
funz = @(u,v) v;

fsurf(funx,funy,funz,[-5 5 -5 -2],'--','EdgeColor','g')
```

```
hold on
fsurf(funx,funy,funz,[-5 5 -2 2],'EdgeColor','none')
hold off
```



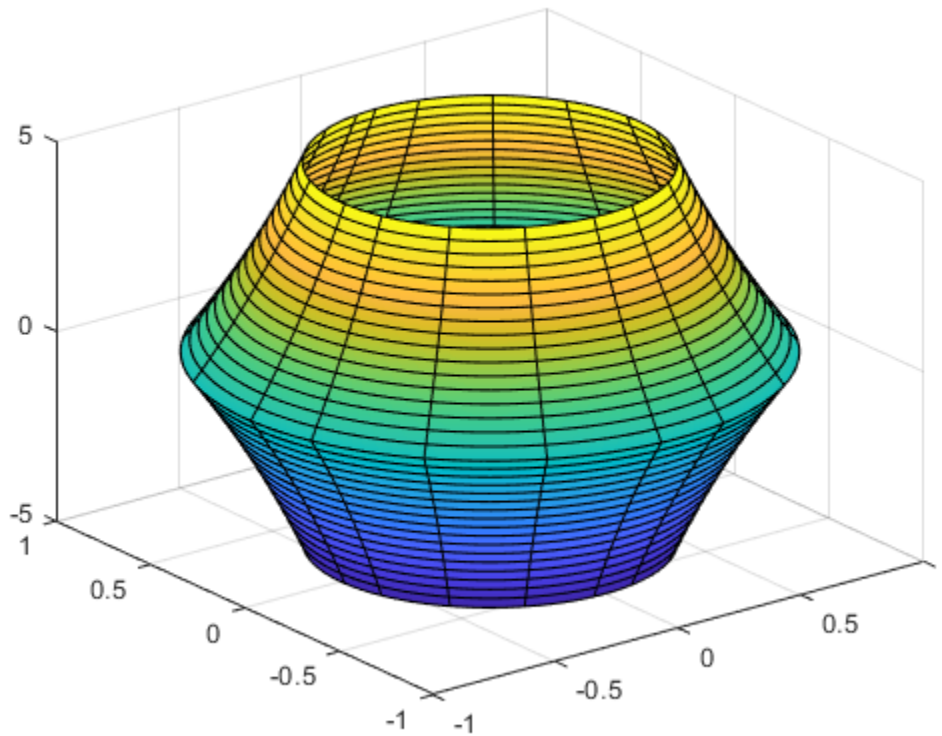
Modify Surface After Creation

Plot the parametric surface

$$x = e^{-|u|/10}\sin(5|v|)$$
$$y = e^{-|u|/10}\cos(5|v|)$$
$$z = u\,.$$

Assign the parameterized function surface object to a variable.

```
x = @(u,v) exp(-abs(u)/10).*sin(5*abs(v));
y = @(u,v) exp(-abs(u)/10).*cos(5*abs(v));
z = @(u,v) u;
fs = fsurf(x,y,z)
```

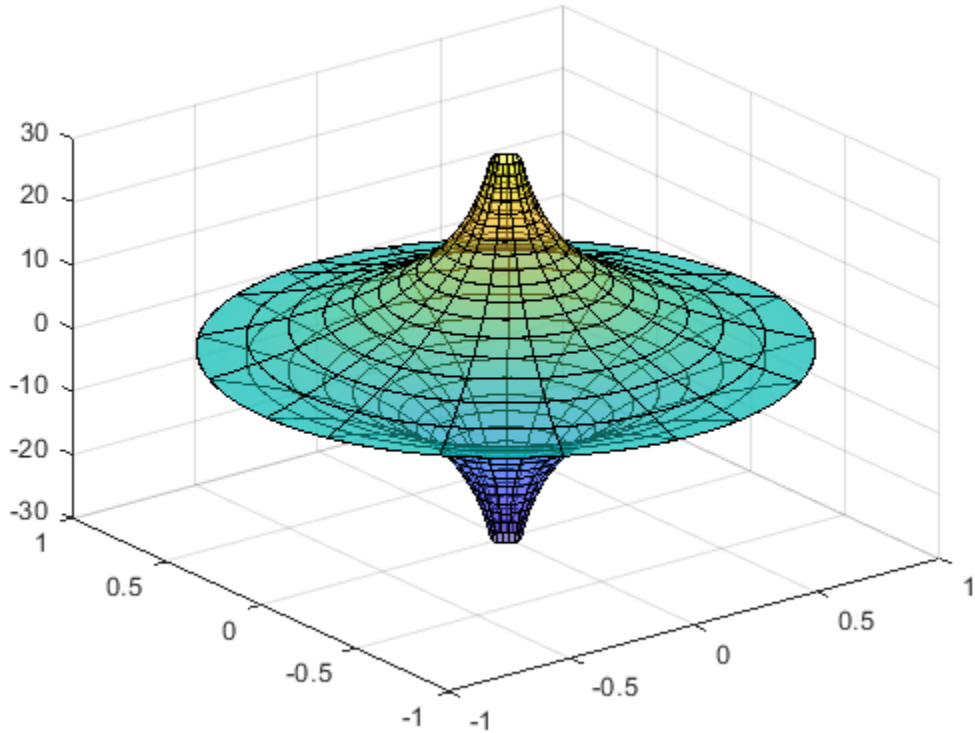



```
fs =
  ParameterizedFunctionSurface with properties:
```

```
    XFunction: @(u,v)exp(-abs(u)/10).*sin(5*abs(v))
    YFunction: @(u,v)exp(-abs(u)/10).*cos(5*abs(v))
    ZFunction: @(u,v)u
    EdgeColor: [0 0 0]
    LineStyle: '-'
    FaceColor: 'interp'

  Show all properties
```

Change the plotting interval for `u` to `[-30 30]` by setting the `URange` property of object. Add transparency to the surface by setting the `FaceAlpha` property to a value between 0 (transparent) and 1 (opaque).

```
fs.URange = [-30 30];
```

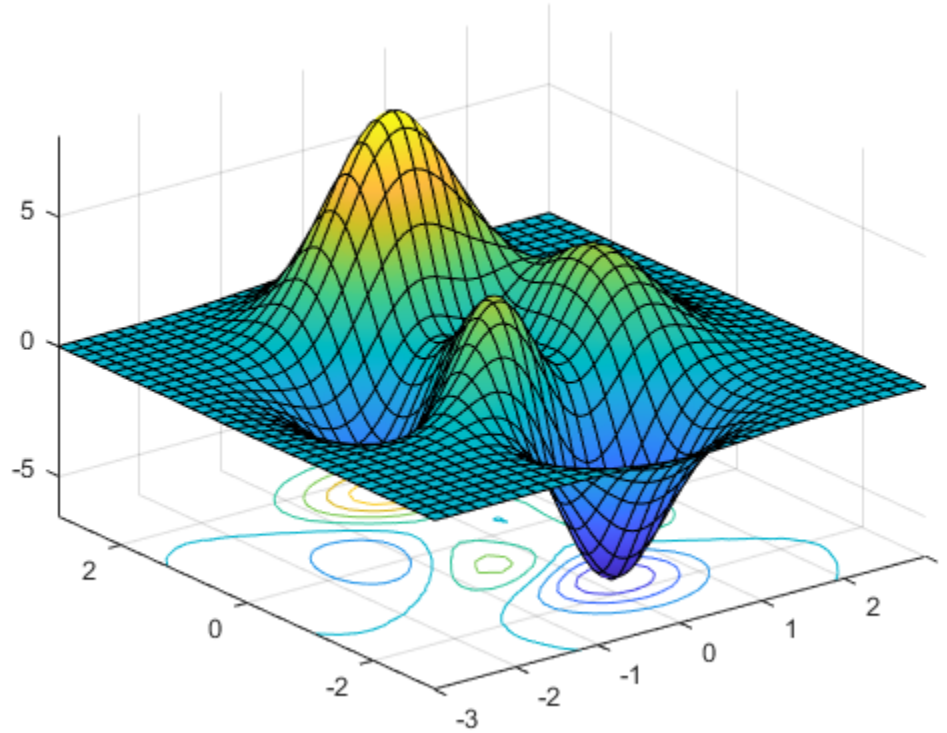



```
fs.FaceAlpha = .5;
```

Show Contours Below Surface Plot

Show contours below a surface plot by setting the `'ShowContours'` option to `'on'`.

```
f = @(x,y) 3*(1-x).^2.*exp(-(x.^2)-(y+1).^2)...
    - 10*(x/5 - x.^3 - y.^5).*exp(-x.^2-y.^2)...
    - 1/3*exp(-(x+1).^2 - y.^2);
fsurf(f,[-3 3],'ShowContours','on')
```

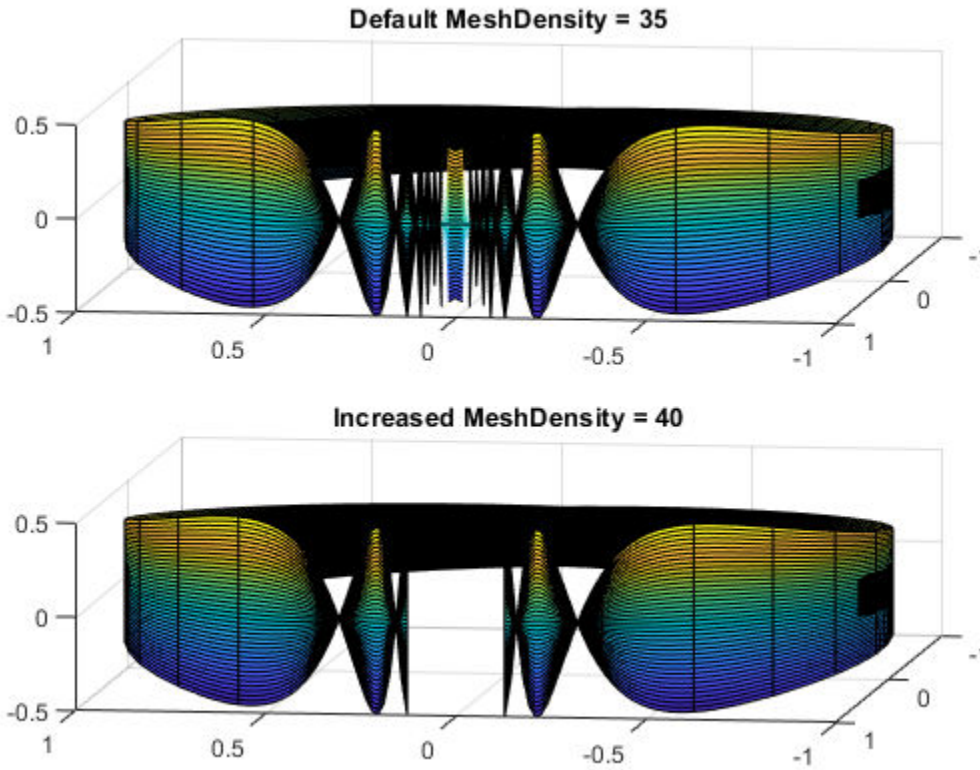


Control Resolution of Surface Plot

Control the resolution of a surface plot using the `'MeshDensity'` option. Increasing `'MeshDensity'` can make smoother, more accurate plots while decreasing it can increase plotting speed.

Divide a figure into two using `subplot`. In the first subplot, plot the parametric surface $x = \sin(s)$, $y = \cos(s)$, $z = (t/10)\sin(1/s)$. The surface has a large gap. Fix this issue by increasing the `'MeshDensity'` to `40` in the second subplot. `fsurf` fills the gap, showing that by increasing `'MeshDensity'` you increased the resolution.

```
subplot(2,1,1)
fsurf(@(s,t) sin(s), @(s,t) cos(s), @(s,t) t/10.*sin(1./s))
view(-172,25)
title('Default MeshDensity = 35')

subplot(2,1,2)
fsurf(@(s,t) sin(s), @(s,t) cos(s), @(s,t) t/10.*sin(1./s),'MeshDensity',40)
view(-172,25)
title('Increased MeshDensity = 40')
```



Input Arguments

f — 3-D function to plot
function handle

3-D function to plot, specified as a function handle to a named or anonymous function.

Specify a function of the form `z = f(x,y)`. The function must accept two matrix input arguments and return a matrix output argument of the same size. Use array operators instead of matrix operators for the best performance. For example, use `.*` (`times`) instead of `*` (`mtimes`).

Example: `f = @(x,y) sin(x) + cos(y);`

`xyinterval` — Plotting interval for `x` and `y`
`[-5 5 -5 5]` (default) | vector of form `[min max]` | vector of form `[xmin xmax ymin ymax]`

Plotting interval for `x` and `y`, specified in one of these forms:

- Vector of form `[min max]` — Use the interval `[min max]` for both `x` and `y`
- Vector of form `[xmin xmax ymin ymax]` — Use the interval `[xmin xmax]` for `x` and `[ymin ymax]` for `y`.

`funx` — Parametric function for *x* coordinates
function handle

Parametric function for *x* coordinates, specified as a function handle to a named or anonymous function.

Specify a function of the form `x = funx(u,v)`. The function must accept two matrix input arguments and return a matrix output argument of the same size. Use array operators instead of matrix operators for the best performance. For example, use `.*` (`times`) instead of `*` (`mtimes`).

Example: `funx = @(u,v) u.*sin(v);`

`funy` — Parametric function for *y* coordinates
function handle

Parametric function for *y* coordinates, specified as a function handle to a named or anonymous function.

Specify a function of the form `y = funy(u,v)`. The function must accept two matrix input arguments and return a matrix output argument of the same size. Use array operators instead of matrix operators for the best performance. For example, use `.*` (`times`) instead of `*` (`mtimes`).

Example: `funy = @(t) @(u,v) -u.*cos(v);`

`funz` — Parametric function for *z* coordinates
function handle

Parametric function for *z* coordinates, specified as a function handle to a named or anonymous function.

Specify a function of the form `z = funz(u,v)`. The function must accept two matrix input arguments and return a matrix output argument of the same size. Use array operators instead of matrix operators for the best performance. For example, use `.*` (`times`) instead of `*` (`mtimes`).

Example: `funz = @(u,v) v;`

`uvinterval` — Plotting interval for `u` and `v`
`[-5 5 -5 5]` (default) | vector of form `[min max]` | vector of form `[umin umax vmin vmax]`

Plotting interval for `u` and `v`, specified in one of these forms:

- Vector of form `[min max]` — Use the interval `[min max]` for both `u` and `v`
- Vector of form `[umin umax vmin vmax]` — Use the interval `[umin umax]` for `u` and `[vmin vmax]` for `v`.

`ax` — Axes object
axes object

Axes object. If you do not specify an axes object, then `fsurf` uses the current axes.

`LineSpec` — Line specification
character vector | string

Line specification, specified as a character vector or string with a line style, marker, and color. The elements can appear in any order, and you can omit one or more options. To show only markers with no connecting lines, specify a marker and omit the line style.

Example: `'r--o'` specifies a red color, a dashed line, and circle markers

Line Style Specifier	Description
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Marker Specifier	Description
`o`	Circle
`+`	Plus sign
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Pentagram
`h`	Hexagram

Color Specifier	Description
`y`	yellow
`m`	magenta
`c`	cyan
`r`	red
`g`	green
`b`	blue
`w`	white
`k`	black

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Marker','o','MarkerFaceColor','red'`

The properties list here are only a subset. For a full list, see FunctionSurface orParameterizedFunctionSurface.

`MeshDensity` — Number of evaluation points per direction
35 (default) | number

Number of evaluation points per direction, specified as a number. The default is `35`. Because `fsurf` objects use adaptive evaluation, the actual number of evaluation points is greater.

Example: `100`

`ShowContours` — Display contour plot under plot
`'off'` (default) | `'on'`

Display contour plot under plot, specified as `'off'` (default) or `'on'`.

`EdgeColor` — Line color
`[0 0 0]` (default) | `'interp'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as `'interp'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default RGB triplet value of `[0 0 0]` corresponds to black. The `'interp'` value colors the edges based on the `ZData` values.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

`Marker` — Marker symbol
`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | `'x'` | `...`

Marker symbol, specified as one of the markers listed in this table. By default, the object does not display markers. Specifying a marker symbol adds markers at the intersection points of mesh lines.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle

Value	Description
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'+'`

Example: `'diamond'`

`MarkerSize` — Marker size
6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

Output Arguments

`fs` — One or more `FunctionSurface` or `ParameterizedFunctionSurface` objects
scalar | vector

One or more `FunctionSurface` or `ParameterizedFunctionSurface` objects, returned as a scalar or a vector.

- If you use the `fsurf(f)` syntax or a variation of this syntax, then `fsurf` returns `FunctionSurface` objects.
- If you use the `fsurf(funx,funy,funz)` syntax or a variation of this syntax, then `fsurf` returns `ParameterizedFunctionSurface` objects.

You can use these objects to query and modify properties of a specific surface. For a list of properties, see FunctionSurface and ParameterizedFunctionSurface.

See Also

Functions
`fcontour` | `fimplicit3` | `fmesh` | `fplot` | `fplot3` | `hold` | `title`

Properties
FunctionSurface | ParameterizedFunctionSurface

Introduced in R2016a

ftell

Current position

Syntax

```
position = ftell(fileID)
```

Description

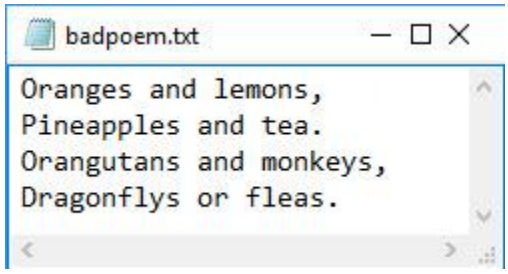
`position = ftell(fileID)` returns the current location of the position pointer in the specified file.

- If the query is successful, then `position` is a zero-based integer that indicates the number of bytes from the beginning of the file.
- If the query is unsuccessful, then `position` is `-1`.

Examples

Find Current Position in File

When you open a file, MATLAB® creates a pointer to indicate the current position within the file. Open the following `badpoem.txt` file and perform a read operation (which advances the position pointer). Then, query the final position in the file after the read operation.



Use `fopen` to open the file. Then, use `ftell` to query the current position.

```
fid = fopen('badpoem.txt');
ftell(fid)
```

```
ans = 0
```

Using `fgetl`, read the first line and examine the current position after the read operation.

```
tline1 = fgetl(fid)  % read the first line
```

```
tline1 =
'Oranges and lemons,'
```

```
ftell(fid)
```

```
ans = 20
```

Read the second line and examine the current position.

```
tline2 = fgetl(fid)  % read the second line
```

```
tline2 =
'Pineapples and tea.'
```

```
ftell(fid)
```

```
ans = 40
```

Close the file.

```
fclose(fid);
```

Input Arguments

`fileID` — File identifier
integer

File identifier of an open file, specified as an integer. To open a file and obtain its valid identifier use the `fopen` function.

Data Types: `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

When the MATLAB behavior differs from the C compiler behavior, the generated code usually matches the C compiler behavior. For example, if you use `fseek` to seek past the end of a file, the behavior of `ftell` in the generated code matches the C compiler behavior.

See Also

`fclose` | `feof` | `ferror` | `fopen` | `fprintf` | `fread` | `frewind` | `fscanf` | `fseek` | `fwrite`

Topics

"Import Binary Data with Low-Level I/O"

Introduced before R2006a

ftp

Object to connect to FTP server and access its files

Description

Connect to an FTP server by calling the `ftp` function, which creates an FTP object. To access a particular FTP account on the server, specify a user name and password. Then use the FTP object to upload and download files. You also can create, delete, and navigate to different folders on the server. To close the connection, use the `close` function.

Note Because FTP is not a secure protocol, the FTP object does not encrypt your user name, your password, or any data you download from or upload to an FTP server. If you require a secure FTP connection, then use an SFTP client provided by your system.

Creation

Syntax

```
ftpobj = ftp(host)
ftpobj = ftp(host,username,password)
ftpobj = ftp(host,username,password,Name,Value)
```

Description

`ftpobj = ftp(host)` opens a connection to the FTP server `host` and stores the connection in FTP object `ftpobj`. To use this syntax, `host` must support anonymous connections.

`ftpobj = ftp(host,username,password)` accesses the FTP account with the specified user name and password.

`ftpobj = ftp(host,username,password,Name,Value)` specifies additional input arguments using one or more name-value pair arguments. For example, you can specify

the value of `'System'` as `'WINDOWS'` to connect to an FTP server that runs a Windows operating system. You also can specify the value of `'LocalDataConnectionMethod'` to change the connection mode from passive to active mode.

Input Arguments

`host` — Name of FTP server

character vector | string scalar

Name of the FTP server, specified as a character vector or string scalar.

To specify an alternate port number for the connection, append a colon (`:`) and the port number to `host`. The default port number for FTP servers is 21.

Typically, the name of the server starts with `'ftp'`, as in `'ftp.example.com'`. However, this practice is a convention, not a technical requirement. If the name of the FTP server starts with a different prefix, then you simply specify `host` as that name. Do not append `'ftp://'` to the name to specify the FTP protocol.

Example: `ftpobj = ftp('ftp.example.com')` opens an anonymous connection to `ftp.example.com`.

Example: `ftpobj = ftp('www.example.com')` opens a connection if the server `www.example.com` is configured to provide FTP service.

Example: `ftpobj = ftp('ftp.example.com:34')` opens a connection using port number 34.

`username` — Name of authorized account

character vector | string scalar

Name of an authorized account on the FTP server, specified as a character vector or string scalar. The FTP object sends `username` as plain text.

`password` — Password for authorized account

character vector | string scalar

Password for an authorized account, specified as a character vector or string scalar. The FTP object sends `password` as plain text.

Example: `ftpobj = ftp('ftp.example.com','nlee','mypassword')` opens a connection for the user `nlee` using the password `mypassword`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `ftpobj = ftp('ftp.example.com','nlee','mypassword','System','WINDOWS')` opens a connection to an FTP server that runs a Windows-based operating system.

`System` — Type of system running on FTP server
`'UNIX'` (default) | `'WINDOWS'`

Type of operating system running on the FTP server, specified as the comma-separated pair consisting of `'System'` and either `'UNIX'` or `'WINDOWS'`.

`LocalDataConnectionMethod` — Connection mode
`'passive'` (default) | `'active'`

Connection mode, specified as the comma-separated pair consisting of `'LocalDataConnectionMethod'` and either `'passive'` or `'active'`.

There are two modes for establishing an FTP connection.

- `'active'` — Your machine establishes a channel for commands, but the FTP server establishes a channel for data. Active mode can be a problem if, for example, your machine is protected by a firewall and does not allow unauthorized session requests from external sources.
- `'passive'` — Your machine establishes both channels. After establishing the command channel, your machine requests that the FTP server start listening on a port, so that your machine can connect to that port.

The default mode is `'passive'` because most modern FTP implementations use passive mode. But to connect to some legacy servers, you might need to specify active mode.

Object Functions

ascii	Set FTP transfer mode to ASCII
binary	Set FTP transfer mode to binary
cd	Change or view current folder on FTP server

close	Close connection to FTP server
delete	Delete file on FTP server
dir	List folder contents on FTP server
mget	Download files from FTP server
mkdir	Make new folder on FTP server
mput	Upload file or folder to FTP server
rename	Rename file on FTP server
rmdir	Remove folder on FTP server

Examples

Download File and List Contents of Folders

To open a connection to an FTP server, create an FTP object. Use the FTP object to download a file and list the contents of subfolders on the server. At the end of the FTP session, close the connection.

First, connect to the National Centers for Environmental Information (NCEI) FTP server.

```
ftpobj = ftp('ftp.ngdc.noaa.gov')
```

```
ftpobj = 

  FTP Object
     host: ftp.ngdc.noaa.gov
     user: anonymous
      dir: /
     mode: binary
```

List the contents of the top-level folder.

```
dir(ftpobj)
```

```
DMSP                          Solid_Earth                   google12c4c939d7b90761.html
INDEX.txt                     coastwatch                    hazards
README.txt                    dmsp4alan                     index.html
STP                           ftp.html                      international
Snow_Ice                      geomag                        ionosonde
```

Download the `README.txt` file from the FTP server. The `mget` function downloads a copy to your current MATLAB® folder.

```
mget(ftpobj,'README.txt');
```

Read the contents of your copy of `README.txt` using the `fileread` function. Split the text into lines using the `splitlines` function and display the first four lines.

```
readme = fileread('README.txt');
readme = splitlines(readme);
readme(1:4)
```

```
ans = 4×1 cell array
    {'                  Welcome to the '                                 }
    {'    NOAA/National Centers for Environmental Information (NCEI), '}
    {'    formerly the National Geophysical Data Center (NGDC)'         }
    {'                     FTP area'                                    }
```

List the contents of a subfolder using the `dir` function.

```
dir(ftpobj,'STP')
```

```
ANOMALIES                   NOAA                        Solid_Earth                 pub
DMSP                        SEIS                        aavso_22nov16               sat
ECLIPSE                     SGD                         aeronomy                    spa
GEOMAGNETIC_DATA            SOLAR_DATA                  cdroms                      spa
GOIN                        SPIDR                       goesr                       swp
GPS_GNSS                    STEP                        ionosonde                   tiv
IONOSPHERE                  SWA                         log.txt
```

Change to a subfolder using the `cd` function. The output from `cd` is the path to the current folder on the FTP server, not your current MATLAB folder.

```
cd(ftpobj,'STP/space-weather')
```

```
ans =
'/STP/space-weather'
```

List the contents of the current folder.

```
dir(ftpobj)
```

```
aurora-airglow           documentation            interplanetary-data      online-publi
```

```
denig-files              geomagnetic-data         ionospheric-data         satellite-da
```

Close the connection to the FTP server. You also can close the connection by deleting the FTP object or letting the connection time out.

```
close(ftpobj)
```

FTP service courtesy of the National Centers for Environmental Information (NCEI). See the NCEI Privacy Policy, Disclaimer, and Copyright for NCEI terms of service.

Tips

- The FTP object does not support proxy server settings.
- While it is unnecessary to specify the `'System',systemKey` name-value pair, the FTP `dir` function might return less information if the FTP object is not configured for the operating system running on the FTP server. In such cases, `dir` might return a structure array with some empty fields. In that case, call `ftp` again and specify the correct value for `systemKey`.

Algorithms

The code for the FTP object is based on code from the Apache™ Project.

In particular, the `ftp` function configures properties for FTP connections as specified by the Apache `FTPClientConfig` class. Any settable property of the `FTPClientConfig` class can be set using the `ftp` function with the corresponding name-value pair argument.

See Also

`sendmail` | `urlread` | `urlwrite` | `webread` | `websave`

Topics

"Perform FTP File Operations"

Introduced before R2006a

full

Convert sparse matrix to full storage

Syntax

```
A = full(S)
```

Description

`A = full(S)` converts sparse matrix `S` to full storage organization, such that `issparse(A)` returns logical `0` (`false`).

Examples

Convert Sparse Matrix to Full Storage

Change the storage format of a matrix and compare the storage requirements.

Create a random sparse matrix. The display of sparse matrices in MATLAB ® omits all zeros and shows the location and value of nonzero elements.

```
rng default %for reproducibility
S = sprand(8,8,0.3)
```

```
S =
   (2,1)       0.0344
   (7,1)       0.4456
   (8,1)       0.7547
   (2,2)       0.4387
   (4,3)       0.7655
   (7,3)       0.6463
   (8,4)       0.2760
   (1,6)       0.9502
   (5,6)       0.1869
   (8,6)       0.6797
```

```
   (3,7)       0.3816
   (4,7)       0.7952
   (8,7)       0.6551
   (6,8)       0.4898
   (7,8)       0.7094
```

Convert the matrix to full storage. The MATLAB display of the matrix reflects the new storage format.

```
A = full(S)
```

```
A = 8×8

         0         0         0         0         0    0.9502         0         0
    0.0344    0.4387         0         0         0         0         0         0
         0         0         0         0         0         0    0.3816         0
         0         0    0.7655         0         0         0    0.7952         0
         0         0         0         0         0    0.1869         0         0
         0         0         0         0         0         0         0    0.4898
    0.4456         0    0.6463         0         0         0         0    0.7094
    0.7547         0         0    0.2760         0    0.6797    0.6551         0
```

Compare the storage requirements of the two formats:

- A uses storage for 64 doubles (8 bytes each), or $64 \cdot 8 = 512$ bytes.
- S uses storage for 15 nonzero elements, as well as 24 integers describing their positions, for a total of $39 \cdot 8 = 312$ bytes.

```
whos
```

```
  Name      Size            Bytes  Class     Attributes

  A         8x8               512  double
  S         8x8               312  double    sparse
```

Input Arguments

S — Sparse matrix to convert
matrix

Sparse matrix to convert, specified as a matrix. If `S` is already a full matrix, then `A` is identical to `S`.

Tips

- If `X` is an `m`-by-`n` matrix with `nz` nonzero elements, then `full(X)` requires space to store `m*n` elements. On the other hand, `sparse(X)` requires space to store `nz` elements and `(nz+n+1)` integers.

  The density of a matrix (`nnz(X)/numel(X)`) determines whether it is more efficient to store the matrix as sparse or full. The exact crossover point depends on the matrix class, as well as the platform. For example, in 32-bit MATLAB, a double sparse matrix with less than about 2/3 density requires less space than the same matrix in full storage. In 64-bit MATLAB, however, double matrices with fewer than half of their elements nonzero are more efficient to store as sparse matrices.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`issparse` | `sparse`

Topics

“Constructing Sparse Matrices”

Introduced before R2006a

fullfile

Build full file name from parts

Syntax

```
f = fullfile(filepart1,...,filepartN)
```

Description

`f = fullfile(filepart1,...,filepartN)` builds a full file specification from the specified folder and file names. `fullfile` inserts platform-dependent file separators where necessary, but does not add a trailing file separator. On Windows platforms, the file separator character is a backslash (\). On other platforms, the file separator might be a different character.

`fullfile` replaces all forward slashes (/) with backslashes (\) on Windows. On UNIX platforms, the backlash (\) character is a valid character in file names and is not replaced.

`fullfile` does not trim leading or trailing separators. `fullfile` collapses inner repeated file separators unless they appear at the beginning of the full file specification. `fullfile` also collapses relative directories indicated by the dot symbol, unless they appear at the end of the full file specification. Relative directories indicated by the double-dot symbol are not collapsed.

Examples

Create a Full File Path on Windows

`fullfile` returns a character vector containing the full path to the file. On Windows® platforms, the file separator character is a backslash (\).

```
f = fullfile('myfolder','mysubfolder','myfile.m')
```

```
f =

    'myfolder\mysubfolder\myfile.m'
```

Create a Full File Path on UNIX

`fullfile` returns a character vector containing the full path to the file. On UNIX® platforms, the file separator character is a forward slash (`/`).

```
f = fullfile('myfolder','mysubfolder','myfile.m')
```

```
f =
'myfolder\mysubfolder\myfile.m'
```

Create Paths to Multiple Files on Windows

`fullfile` returns a cell array containing paths to the files `myfile1.m` and `myfile2.m`.

```
f = fullfile('c:\','myfiles','matlab',{'myfile1.m';'myfile2.m'})
```

```
f =

  2×1 cell array

    'c:\myfiles\matlab\myfile1.m'
    'c:\myfiles\matlab\myfile2.m'
```

Collapse File Separators and Dot Symbols on Windows

Create paths to folders using file separators and dot symbols.

`fullfile` does not trim leading or trailing file separators. `filesep` returns the platform-specific file separator character.

```
f = fullfile('c:\','myfiles','matlab',filesep)
```

```
f =

c:\myfiles\matlab\
```

`fullfile` collapses repeated file separators unless they appear at the beginning of the full file specification.

```
f = fullfile('c:\folder1', '\\\folder2\\')
```

```
f =

c:\folder1\folder2\
```

`fullfile` collapses relative directories indicated by the dot symbol unless they appear at the end of the full file specification. Relative directories indicated by the double-dot symbol are not collapsed.

```
f = fullfile('c:\folder1', '.\folder2', '..\folder3\.')
```

```
f =

c:\folder1\folder2\..\folder3\.
```

Input Arguments

`filepart1,...,filepartN` — Folder or file names
character vectors | string arrays | cell arrays of character vectors

Folder or file names, specified as character vectors, string arrays, or cell arrays of character vectors. Any nonscalar cell arrays and nonscalar strings must be of the same size.

Example: `'folder1'`

Example: `'myfile.m, myfile2.m'`

Example: `{'folder1';'folder2'}`

Data Types: `char` | `cell` | `string`

Output Arguments

`f` — full file specification
character array | string array | cell array of character vectors

Full file specification, returned as a character array, a string array, or a cell array of character vectors. If any input argument is a string array, `f` is a string array. Otherwise, if any input argument is a cell array of character vectors, `f` is a cell array of character vectors. Otherwise, `f` is a character array.

Data Types: `char` | `cell` | `string`

Tips

- To split a full file name into folder parts, use the `split` function.

See Also

`fileparts` | `filesep` | `genpath` | `path` | `pathsep` | `split`

Introduced before R2006a

func2str

Construct character vector from function handle

Syntax

```
c = func2str(fh)
```

Description

`c = func2str(fh)` constructs a character vector, `c`, that contains the name of the function associated with the function handle, `fh`. If `fh` is associated with an anonymous function, `func2str` returns a character vector that represents the anonymous function.

Examples

Convert Function Handle to Character Vector

Create function handles for both the `cos` function and for an anonymous function, and then convert them to character vectors.

```
fh = @cos;
c = func2str(fh)
```

```
c =
'cos'
```

```
fh = @(x,y)sqrt(x.^2+y.^2);
c = func2str(fh);

disp(['Anonymous function: ' c])
```

```
Anonymous function: @(x,y)sqrt(x.^2+y.^2)
```

Programmatically Display Function Handle Name as Character Vector

Create a function that evaluates a function handle for a single input.

Create the following function in a file, `evaluateHandle.m`, in your working folder.

```
function evaluateHandle(fh,x)

y = fh(x);
str = func2str(fh);

disp('For input value: ')
disp(x)
disp(['The function ' str ' evaluates to:'])
disp(y)

end
```

Use a function handle to evaluate the `sin` function at `pi/2`.

```
fh = @sin;
x = pi/2;
evaluateHandle(fh,x)
```

```
For input value:
    1.5708

The function sin evaluates to:
     1
```

Use a function handle to evaluate $x^2 + 7$ for the specified matrix, `A`.

```
fh = @(x) x.^2+7;
A = [1 2;0 1];
evaluateHandle(fh,A)
```

```
For input value:
     1     2
     0     1

The function @(x)x.^2+7 evaluates to:
     8    11
```

7 8

Input Arguments

`fh` — Handle to convert to character vector
function handle

Handle to convert to a character vector, specified as a function handle.

Tips

- You lose variables stored in the function handle when you convert it to a character vector using `func2str`, and then back to a handle using `str2func`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If the input argument is associated with an anonymous function, then the generated code returns the character vector `'@(...)...'`.

See Also

`functions` | `str2func`

Topics

"Create Function Handle"

Introduced before R2006a

function

Declare function name, inputs, and outputs

Syntax

```
function [y1,...,yN] = myfun(x1,...,xM)
```

Description

`function [y1,...,yN] = myfun(x1,...,xM)` declares a function named `myfun` that accepts inputs `x1,...,xM` and returns outputs `y1,...,yN`. This declaration statement must be the first executable line of the function. Valid function names begin with an alphabetic character, and can contain letters, numbers, or underscores.

You can save your function:

- In a function file which contains only function definitions. The name of the file must match the name of the first function in the file.
- In a script file which contains commands and function definitions. Functions must be at the end of the file. Script files cannot have the same name as a function in the file. Functions are supported in scripts in R2016b or later.

Files can include multiple local functions or nested functions. For readability, use the `end` keyword to indicate the end of each function in a file. The `end` keyword is required when:

- Any function in the file contains a nested function.
- The function is a local function within a function file, and any local function in the file uses the `end` keyword.
- The function is a local function within a script file.

Examples

Function with One Output

Define a function in a file named `average.m` that accepts an input vector, calculates the average of the values, and returns a single result.

```
function ave = average(x)
    ave = sum(x(:))/numel(x);
end
```

Call the function from the command line.

```
z = 1:99;
ave = average(z)
```

```
ave =
    50
```

Function with Multiple Outputs

Define a function in a file named `stat.m` that returns the mean and standard deviation of an input vector.

```
function [m,s] = stat(x)
    n = length(x);
    m = sum(x)/n;
    s = sqrt(sum((x-m).^2/n));
end
```

Call the function from the command line.

```
values = [12.7, 45.4, 98.9, 26.6, 53.1];
[ave,stdev] = stat(values)
```

```
ave =
   47.3400
stdev =
   29.4124
```

Function in a Script File

Define a script in a file named `integrationScript.m` that computes the value of the integrand at $2\pi/3$ and computes the area under the curve from 0 to π. Include a local function that defines the integrand, $y = \sin(x)^3$.

Note: Including functions in scripts requires MATLAB® R2016b or later.

```
% Compute the value of the integrand at 2*pi/3.
x = 2*pi/3;
y = myIntegrand(x)

% Compute the area under the curve from 0 to pi.
xmin = 0;
xmax = pi;
f = @myIntegrand;
a = integral(f,xmin,xmax)

function y = myIntegrand(x)
    y = sin(x).^3;
end
```

```
y =

    0.6495


a =

    1.3333
```

Multiple Functions in a Function File

Define two functions in a file named `stat2.m`, where the first function calls the second.

```
function [m,s] = stat2(x)
    n = length(x);
    m = avg(x,n);
    s = sqrt(sum((x-m).^2/n));
end
```

```
function m = avg(x,n)
    m = sum(x)/n;
end
```

Function `avg` is a local function. Local functions are only available to other functions within the same file.

Call function `stat2` from the command line.

```
values = [12.7, 45.4, 98.9, 26.6, 53.1];
[ave,stdev] = stat2(values)
```

```
ave =
   47.3400
stdev =
   29.4124
```

Function with Argument Validation

Define a function that restricts input to a numeric vector that contains no `Inf` or `NaN` elements. This function uses the `arguments` keyword, which is valid for MATLAB versions R2019b and later.

```
function [m,s] = stat3(x)
    arguments
        x (1,:) {mustBeNumeric, mustBeFinite}
    end
    n = length(x);
    m = avg(x,n);
    s = sqrt(sum((x-m).^2/n));
end

function m = avg(x,n)
    m = sum(x)/n;
end
```

In the `arguments` code block, `(1,:)` indicates that `x` must be a vector. The validation functions, `{mustBeNumeric, mustBeFinite}`, restrict the elements in `x` to numeric values that are not `Inf` or `NaN`. For more information, see “Function Argument Validation”.

Calling the function with a vector that contains an element that is `NaN` violates the input argument declaration. This violation results in an error being thrown by the `mustBeFinite` validation function.

```
values = [12.7, 45.4, 98.9, NaN, 53.1];
[ave,stdev] = stat3(values)
```

```
Invalid input argument at position 1. Value must be finite.
```

See Also

`arguments` | `nargin` | `nargout` | `pcode` | `return` | `varargin` | `varargout` | `what` | `which`

Topics

"Create Functions in Files"
"Local Functions"
"Nested Functions"
"Base and Function Workspaces"
"Function Precedence Order"
"Function Argument Validation"
"Indexing into Function Call Results"

Introduced before R2006a

functions

Information about function handle

Syntax

```
s = functions(fh)
```

Description

`s = functions(fh)` returns information about a function handle. This information includes the function name, type, and file name.

Use the `functions` function for querying and debugging purposes only.

Note Do not use `functions` programmatically because its behavior could change in subsequent MATLAB releases.

Examples

Display Information About Named Function Handle

Create a function handle and display its information.

```
fh = @cos;
s = functions(fh)
```

```
s = struct with fields:
    function: 'cos'
        type: 'simple'
        file: ''
```

Display Information About Anonymous Function Handle

Create a function handle to an anonymous function. Display its information and values of required variables.

Create a handle to the function $x^2 + y$, and invoke the function using the handle.

```
y = 7;
fh = @(x)x.^2+y;
z = fh(2)
```

```
z =

    11
```

Display information about the function handle.

```
s = functions(fh)
```

```
s =

            function: '@(x)x.^2+y'
                type: 'anonymous'
                file: ''
           workspace: {[1x1 struct]}
    within_file_path: '__base_function'
```

The function handle contains the required value of `y`.

```
s.workspace{1}
```

```
ans =

    y: 7
```

Display Information About Nested and Local Function Handle

Create a function that returns handles to local and nested functions. Display their information.

Create the following function in a file, `functionsExample.m`, in your working folder. The function returns handles to a nested and local function.

```
function [hNest,hLocal] = functionsExample(v)

hNest = @nestFunction;
hLocal = @localFunction;

    function y = nestFunction(x)
        y = x + v;
    end

end

function y = localFunction(z)
y = z + 1;
end
```

Invoke the function.

```
[hNest,hLocal] = functionsExample(13)
```

```
hNest =

    @functionsExample/nestFunction


hLocal =

    @localFunction
```

Display information about the handle to the nested function.

```
s1 = functions(hNest)
```

```
s1 =

     function: 'functionsExample/nestFunction'
         type: 'nested'
         file: 'C:\work\functionsExample.m'
    workspace: {[1x1 struct]}
```

Display information about the handle to the local function.

```
s2 = functions(hLocal)
```

```
s2 =

     function: 'localFunction'
```

```
         type: 'scopedfunction'
         file: 'C:\work\functionsExample.m'
    parentage: {'localFunction'  'functionsExample'}
```

Input Arguments

fh — Handle to query
function handle

Handle to query, specified as a function handle.

Output Arguments

s — Information about function handle
structure

Information about a function handle, returned as a structure. The structure contains the following fields.

Field Name	Field Description
`function`	Function name. If the function associated with the handle is a nested function, the function name takes the form *`main_function/nested_function`*.
`type`	Function type. For example `'simple'`, `'nested'`, `'scopedfunction'`, or `'anonymous'`.
`file`	Full path to the function with the file extension. • If the function is a local or nested function, then `file` is the full path to the main function. • If the function is built-in MATLAB function or an anonymous function, then `file` is an empty character array (`''`). • If you load a saved function handle, then `file` is an empty character array (`''`).

The structure has additional fields depending on the type of function associated with the handle. For example, a local function has a `parentage` field, and an anonymous function

has a `workspace` field. Use the information in `s` for querying and debugging purposes only.

See Also

`func2str` | `str2func`

Topics

"Create Function Handle"
"Compare Function Handles"

Introduced before R2006a

functiontests

Create array of tests from handles to local functions

Syntax

```
tests = functiontests(f)
```

Description

`tests = functiontests(f)` creates an array of tests, `tests`, from a cell array of handles to local functions, `f`. To apply defined setup and teardown functions, include their function handles in `f`.

Local test functions must include ‘test’ at the beginning or end of the function name. `functiontests` must be called from within a test file.

Examples

Create Test Array

Create the file `exampleTest.m` in your MATLAB® path. In the main function, create a test array. Use local functions to define setup, teardown, and two function tests. Your file should look like this.

```
function tests = exampleTest
tests = functiontests(localfunctions);

function setup(testCase)
function teardown(testCase)
function exampleOneTest(testCase)
function testExampleTwo(testCase)
```

From the command line, call the `exampleTest` function.

```
tests = exampleTest
```

```
tests = 

  1x2 Test array with properties:

    Name
    ProcedureName
    TestClass
    BaseFolder
    Parameterization
    SharedTestFixtures
    Tags

Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

Access the test suite to verify the names of the two function tests.

```
tests.Name
```

```
ans = 

    'exampleTest/exampleOneTest'


ans = 

    'exampleTest/testExampleTwo'
```

Input Arguments

f — Handles to local test functions
cell array of function handles

Handles to local test functions, specified as a cell array. Use `f=localfunctions` in your working file to automatically generate a cell array of function handles for that file. If you want explicit test enumeration, construct `f` by listing individual functions. `f` must include any setup or teardown functions necessary for your test.

Example: `f = localfunctions;`

Example: `f = {@setup,@exampleOneTest,@teardown};`

See Also

`localfunctions` | `matlab.unittest.Test` | `runtests`

Topics

"Write Function-Based Unit Tests"
"Write Simple Test Case Using Functions"
"Write Test Using Setup and Teardown Functions"

Introduced in R2013b

funm

Evaluate general matrix function

Syntax

```
F = funm(A,fun)
F = funm(A,fun,options)
F = funm(A,fun,options,p1,p2,...)
[F,exitflag] = funm(...)
[F,exitflag,output] = funm(...)
```

Description

`F = funm(A,fun)` evaluates the user-defined function `fun` at the square matrix argument `A`. `F = fun(x,k)` must accept a vector `x` and an integer `k`, and return a vector `f` of the same size of `x`, where `f(i)` is the `k`th derivative of the function `fun` evaluated at `x(i)`. The function represented by fun must have a Taylor series with an infinite radius of convergence, except for `fun = @log`, which is treated as a special case.

You can also use `funm` to evaluate the special functions listed in the following table at the matrix `A`.

Function	Syntax for Evaluating Function at Matrix A
`exp`	`funm(A, @exp)`
`log`	`funm(A, @log)`
`sin`	`funm(A, @sin)`
`cos`	`funm(A, @cos)`
`sinh`	`funm(A, @sinh)`
`cosh`	`funm(A, @cosh)`

For matrix square roots, use `sqrtm(A)` instead. For matrix exponentials, which of `expm(A)` or `funm(A, @exp)` is the more accurate depends on the matrix `A`.

The function represented by `fun` must have a Taylor series with an infinite radius of convergence. The exception is `@log`, which is treated as a special case. “Parameterizing Functions” explains how to provide additional parameters to the function `fun`, if necessary.

`F = funm(A,fun,options)` sets the algorithm's parameters to the values in the structure `options`.

The following table lists the fields of `options`.

Field	Description	Values
`options.Display`	Level of display	`'off'` (default), `'on'`, `'verbose'`
`options.TolBlk`	Tolerance for blocking Schur form	Positive scalar. The default is `0.1`.
`options.TolTay`	Termination tolerance for evaluating the Taylor series of diagonal blocks	Positive scalar. The default is `eps`.
`options.MaxTerms`	Maximum number of Taylor series terms	Positive integer. The default is `250`.
`options.MaxSqrt`	When computing a logarithm, maximum number of square roots computed in inverse scaling and squaring method.	Positive integer. The default is `100`.
`options.Ord`	Specifies the ordering of the Schur form `T`.	A vector of length `length(A)`. `options.Ord(i)` is the index of the block into which `T(i,i)` is placed. The default is `[]`.

`F = funm(A,fun,options,p1,p2,...)` passes extra inputs `p1,p2,...` to the function.

`[F,exitflag] = funm(...)` returns a scalar `exitflag` that describes the exit condition of `funm`. `exitflag` can have the following values:

- `0` — The algorithm was successful.
- `1` — One or more Taylor series evaluations did not converge, or, in the case of a logarithm, too many square roots are needed. However, the computed value of `F` might still be accurate.

`[F,exitflag,output] = funm(...)` returns a structure `output` with the following fields:

Field	Description
`output.terms`	Vector for which `output.terms(i)` is the number of Taylor series terms used when evaluating the *i*th block, or, in the case of the logarithm, the number of square roots of matrices of dimension greater than 2.
`output.ind`	Cell array for which the `(i,j)` block of the reordered Schur factor `T` is `T(output.ind{i}, output.ind{j})`.
`output.ord`	Ordering of the Schur form, as passed to `ordschur`
`output.T`	Reordered Schur form

If the Schur form is diagonal then `output = struct('terms',ones(n,1),'ind',{1:n})`.

Examples

Example 1

The following command computes the matrix sine of the 3-by-3 magic matrix.

```
F=funm(magic(3), @sin)

F =

   -0.3850    1.0191    0.0162
    0.6179    0.2168   -0.1844
    0.4173   -0.5856    0.8185
```

Example 2

The statements

```
S = funm(X,@sin);
C = funm(X,@cos);
```

produce the same results to within roundoff error as

```
E = expm(i*X);
C = real(E);
S = imag(E);
```

In either case, the results satisfy `S*S+C*C = I`, where `I = eye(size(X))`.

Example 3

To compute the function `exp(x) + cos(x)` at `A` with one call to `funm`, use

```
F = funm(A,@fun_expcos)
```

where `fun_expcos` is the following function.

```
function f = fun_expcos(x, k)
% Return kth derivative of exp + cos at X.
        g = mod(ceil(k/2),2);
        if mod(k,2)
           f = exp(x) + sin(x)*(-1)^g;
        else
           f = exp(x) + cos(x)*(-1)^g;
        end
```

Algorithms

The algorithm `funm` uses is described in [1].

References

[1] Davies, P. I. and N. J. Higham, "A Schur-Parlett algorithm for computing matrix functions," *SIAM J. Matrix Anal. Appl.*, Vol. 25, Number 2, pp. 464-485, 2003.

[2] Golub, G. H. and C. F. Van Loan, *Matrix Computation*, Third Edition, Johns Hopkins University Press, 1996, p. 384.

[3] Moler, C. B. and C. F. Van Loan, "Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later" *SIAM Review 20*, Vol. 45, Number 1, pp. 1-47, 2003.

See Also

`expm` | `logm` | `sqrtm`

Topics

"Create Function Handle"

Introduced before R2006a

fwrite

Write data to binary file

Syntax

```
fwrite(fileID,A)
fwrite(fileID,A,precision)
fwrite(fileID,A,precision,skip)
fwrite(fileID,A,precision,skip,machinefmt)
```

```
count = fwrite( ___ )
```

Description

`fwrite(fileID,A)` write the elements of array `A` as 8-bit unsigned integers to a binary file in column order. The binary file is indicated by the file identifier, `fileID`. Use `fopen` to open the file and obtain the `fileID` value. When you finish reading, close the file by calling `fclose(fileID)`.

`fwrite(fileID,A,precision)` writes the values in `A` in the form and size described by `precision`.

`fwrite(fileID,A,precision,skip)` skips the number of bytes or bits specified by `skip` before writing each value.

`fwrite(fileID,A,precision,skip,machinefmt)` additionally specifies the order for writing bytes or bits to the file. The `skip` argument is optional.

`count = fwrite( ___ )` returns the number of elements of `A` that `fwrite` successfully writes to the file. You can use this syntax with any of the input arguments of the previous syntaxes.

Examples

Write uint8 Data to Binary File

Open a file named `nine.bin` for writing. Specify write access using `'w'` in the call to `fopen`.

```
fileID = fopen('nine.bin','w');
```

`fopen` returns a file identifier, `fileID`.

Write the integers from 1 to 9 as 8-bit unsigned integers.

```
fwrite(fileID,[1:9]);
```

Close the file.

```
fclose(fileID);
```

Write 4-byte Integers to Binary File

Open a file named `magic5.bin` for writing.

```
fileID = fopen('magic5.bin','w');
```

Write the 25 elements of the 5-by-5 magic square. Use the `precision` argument, `'integer*4'`, to write 4-byte integers.

```
fwrite(fileID,magic(5),'integer*4');
```

Close the file.

```
fclose(fileID);
```

Append Data to Binary File

Write a binary file containing the elements of the 4-by-4 magic square, stored as double-precision floating-point numbers.

```
fileID = fopen('magic4.bin','w');
fwrite(fileID,magic(4),'double');
fclose(fileID);
```

Open the file, `magic4.bin`, with write-access that enables appending to the file. Specify the file-access type, `'a'`, in the call to `fopen`.

```
fileID = fopen('magic4.bin','a');
```

Append a 4-by-4 matrix of zeros to the file. Then, close the file.

```
fwrite(fileID,zeros(4),'double');
fclose(fileID);
```

Write Binary File with Big-Endian Byte Ordering

Write random double-precision numbers to a file named `myfile.bin` for use on a big-endian system. Specify a `machinefmt` value of `'ieee-be'` in the call to `fwrite`, to indicate big-endian byte ordering.

```
fileID = fopen('myfile.bin','w');
fwrite(fileID,rand(4),'double','ieee-be');
fclose(fileID);
```

Input Arguments

`fileID` — File identifier
integer | 1 | 2

File identifier, specified as an integer obtained from `fopen`, 1 for standard output (the screen), or 2 for standard error.

`A` — Data to write
numeric array | character array | string array

Data to write, specified as a numeric, character, or string array.

Example: `[1,2,3;4,5,6]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string`
Complex Number Support: Yes

`precision` — Class and size of values to write

`'uint8'` (default) | character vector | string scalar

Class and size in bits of the values to write, specified as one of the character vectors or string scalars listed in the Precision column.

Value Type	Precision	Bits (Bytes)
Integers, unsigned	`'uint'`	32 (4)
	`'uint8'`	8 (1)
	`'uint16'`	16 (2)
	`'uint32'`	32 (4)
	`'uint64'`	64 (8)
	`'uchar'`	8 (1)
	`'unsigned char'`	8 (1)
	`'ushort'`	16 (2)
	`'ulong'`	32 (4)
	`'ubit`*n*`'`	$1 \le n \le 64$
Integers, signed	`'int'`	32 (4)
	`'int8'`	8 (1)
	`'int16'`	16 (2)
	`'int32'`	32 (4)
	`'int64'`	64 (8)
	`'integer*1'`	8 (1)
	`'integer*2'`	16 (2)
	`'integer*4'`	32 (4)
	`'integer*8'`	64 (8)
	`'schar'`	8 (1)
	`'signed char'`	8 (1)
	`'short'`	16 (2)
	`'long'`	32 (4)
	`'bit`*n*`'`	$1 \le n \le 64$

Value Type	Precision	Bits (Bytes)
Floating-point numbers	`'single'`	`32 (4)`
	`'double'`	`64 (8)`
	`'float'`	`32 (4)`
	`'float32'`	`32 (4)`
	`'float64'`	`64 (8)`
	`'real*4'`	`32 (4)`
	`'real*8'`	`64 (8)`
Characters	`'char*1'`	`8 (1)`
	`'char'`	Depends on the encoding scheme associated with the file. Set encoding with `fopen`.

If you specify a precision of `bit`*n* or `ubit`*n*, then `fwrite` saturates for all values outside the range.

Note To preserve `NaN` and `Inf` values in MATLAB, read and write data of class `double` or `single`.

`skip` — Number of bytes to skip

0 (default) | scalar

Number of bytes to skip before writing each value, specified as a scalar. If you specify a `precision` of `bit`*n* or `ubit`*n*, specify `skip` in bits.

Use the `skip` argument to insert data into noncontiguous fields in fixed-length records.

`machinefmt` — Order for writing bytes

`'n'` (default) | `'b'` | `'l'` | `'s'` | `'a'`

Order for writing bytes within the file, specified as one of the character vectors or string scalars in the table that follows. For `bit`*n* and `ubit`*n* precisions, `machinefmt` specifies the order for writing bits within a byte, but the order for writing bytes remains your system byte ordering.

`'n'` or `'native'`	Your system byte ordering (default)
`'b'` or `'ieee-be'`	Big-endian ordering
`'l'` or `'ieee-le'`	Little-endian ordering
`'s'` or `'ieee-be.l64'`	Big-endian ordering, 64-bit long data type
`'a'` or `'ieee-le.l64'`	Little-endian ordering, 64-bit long data type

By default, all currently supported platforms use little-endian ordering for new files. Existing binary files can use either big-endian or little-endian ordering.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The precision argument must be a constant.
- Does not support precision types `'long'`, `'ulong'`, `'unsigned long'`, `'bit*n*'`, and `'ubit*n*'`.
- Does not support the machine format (order for writing bytes) input argument.
- If the precision is a C type such as `int`, the target and production sizes for that type must:
  - Match.
  - Map directly to a MATLAB integer type.
- Treats a `char` type as a signed 8-bit integer. Use values from 0 through 127 only.
- When appending to a file and using a skip argument, it must be possible for the C run-time `fseek` to seek beyond the end of the file and initialize unwritten bytes to 0. This behavior matches the behavior of POSIX and Windows.

See Also

`fclose` | `ferror` | `fopen` | `fprintf` | `fread` | `fscanf` | `fseek` | `ftell`

Topics

"Export Binary Data with Low-Level I/O"

Introduced before R2006a

fwrite (serial)

(Not recommended) Write binary data to device

Note This `serial` object function is not recommended. Use `serialport` object functions instead. See “Compatibility Considerations”.

Syntax

```
fwrite(obj,A)
fwrite(obj,A,'precision')
fwrite(obj,A,'mode')
fwrite(obj,A,'precision','mode')
```

Description

`fwrite(obj,A)` writes the binary data `A` to the device connected to the serial port object, `obj`.

`fwrite(obj,A,'precision')` writes binary data with precision specified by *`precision`*.

`precision` controls the number of bits written for each value and the interpretation of those bits as integer, floating-point, or character values. If *`precision`* is not specified, `uchar` (an 8-bit unsigned character) is used. The supported values for *`precision`* are listed in Tips on page 1-4735.

`fwrite(obj,A,'mode')` writes binary data with command-line access specified by *`mode`*. If *`mode`* is `sync`, `A` is written synchronously and the command line is blocked. If *`mode`* is `async`, `A` is written asynchronously and the command line is not blocked. If *`mode`* is not specified, the write operation is synchronous.

`fwrite(obj,A,'precision','mode')` writes binary data with precision specified by *`precision`* and command-line access specified by *`mode`*.

Tips

Before you can write data to the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a write operation while `obj` is not connected to the device.

The `ValuesSent` property value is increased by the number of values written each time `fwrite` is issued.

An error occurs if the output buffer cannot hold all the data to be written. You can specify the size of the output buffer with the `OutputBufferSize` property.

If you set the FlowControl property to `hardware` on a serial object, and a hardware connection is not detected, `fwrite` returns an error message. This occurs if a device is not connected, or a connected device is not asserting that is ready to receive data. Check the remote device status and flow control settings to see if hardware flow control is causing errors in MATLAB.

Note If you want to check to see if the device is asserting that it is ready to receive data, set the FlowControl to `none`. Once you connect to the device, check the `PinStatus` structure for `ClearToSend`. If `ClearToSend` is `off`, there is a problem on the remote device side. If `ClearToSend` is `on`, there is a hardware FlowControl device prepared to receive data and you can execute `fwrite`.

Synchronous Versus Asynchronous Write Operations

By default, data is written to the device synchronously and the command line is blocked until the operation completes. You can perform an asynchronous write by configuring the *`mode`* input argument to be `async`. For asynchronous writes:

- The `BytesToOutput` property value is continuously updated to reflect the number of bytes in the output buffer.
- The callback function specified for the `OutputEmptyFcn` property is executed when the output buffer is empty.

You can determine whether an asynchronous write operation is in progress with the `TransferStatus` property.

Synchronous and asynchronous write operations are discussed in more detail in “Write and Read Data”.

Rules for Completing a Write Operation with fwrite

A binary write operation using `fwrite` completes when:

- The specified data is written.
- The time specified by the `Timeout` property passes.

Note The `Terminator` property is not used with binary write operations.

Supported Precisions

The following table shows the supported values for *`precision`*.

Data Type	Precision	Interpretation
Character	`uchar`	8-bit unsigned character
	`schar`	8-bit signed character
	`char`	8-bit signed or unsigned character
Integer	`int8`	8-bit integer
	`int16`	16-bit integer
	`int32`	32-bit integer
	`uint8`	8-bit unsigned integer
	`uint16`	16-bit unsigned integer
	`uint32`	32-bit unsigned integer
	`short`	16-bit integer
	`int`	32-bit integer
	`long`	32- or 64-bit integer
	`ushort`	16-bit unsigned integer
	`uint`	32-bit unsigned integer
	`ulong`	32- or 64-bit unsigned integer

Data Type	Precision	Interpretation
Floating-point	`single`	32-bit floating point
	`float32`	32-bit floating point
	`float`	32-bit floating point
	`double`	64-bit floating point
	`float64`	64-bit floating point

Compatibility Considerations

`serial` object interface is not recommended

Not recommended starting in R2019b

Use of this function with a `serial` object is not recommended. To access a serial port device, use a `serialport` object with its functions and properties instead.

See “Transition Your Code to serialport Interface” for more information about using the recommended functionality.

See Also

Functions

`fopen` | `fprintf`

Introduced before R2006a

fzero

Root of nonlinear function

Syntax

```
x = fzero(fun,x0)
x = fzero(fun,x0,options)

x = fzero(problem)

[x,fval,exitflag,output] = fzero( ___ )
```

Description

`x = fzero(fun,x0)` tries to find a point `x` where `fun(x) = 0`. This solution is where `fun(x)` changes sign—`fzero` cannot find a root of a function such as `x^2`.

`x = fzero(fun,x0,options)` uses `options` to modify the solution process.

`x = fzero(problem)` solves a root-finding problem specified by `problem`.

`[x,fval,exitflag,output] = fzero( ___ )` returns `fun(x)` in the `fval` output, `exitflag` encoding the reason `fzero` stopped, and an output structure containing information on the solution process.

Examples

Root Starting From One Point

Calculate π by finding the zero of the sine function near 3.

```
fun = @sin; % function
x0 = 3; % initial point
x = fzero(fun,x0)
```

```
x = 3.1416
```

Root Starting From an Interval

Find the zero of cosine between 1 and 2.

```
fun = @cos; % function
x0 = [1 2]; % initial interval
x = fzero(fun,x0)
```

```
x = 1.5708
```

Note that cos(1) and cos(2) differ in sign.

Root of a Function Defined by a File

Find a zero of the function $f(x) = x^3 - 2x - 5$.

First, write a file called `f.m`.

```
function y = f(x)
y = x.^3 - 2*x - 5;
```

Save `f.m` on your MATLAB path.

Find the zero of $f(x)$ near 2.

```
fun = @f; % function
x0 = 2; % initial point
z = fzero(fun,x0)
```

```
z =
    2.0946
```

Since `f(x)` is a polynomial, you can find the same real zero, and a complex conjugate pair of zeros, using the `roots` command.

```
roots([1 0 -2 -5])
```

```
   ans =
   2.0946
```

```
-1.0473 + 1.1359i
-1.0473 - 1.1359i
```

Root of Function with Extra Parameter

Find the root of a function that has an extra parameter.

```
myfun = @(x,c) cos(c*x);  % parameterized function
c = 2;                    % parameter
fun = @(x) myfun(x,c);    % function of x alone
x = fzero(fun,0.1)
```

```
x = 0.7854
```

Nondefault Options

Plot the solution process by setting some plot functions.

Define the function and initial point.

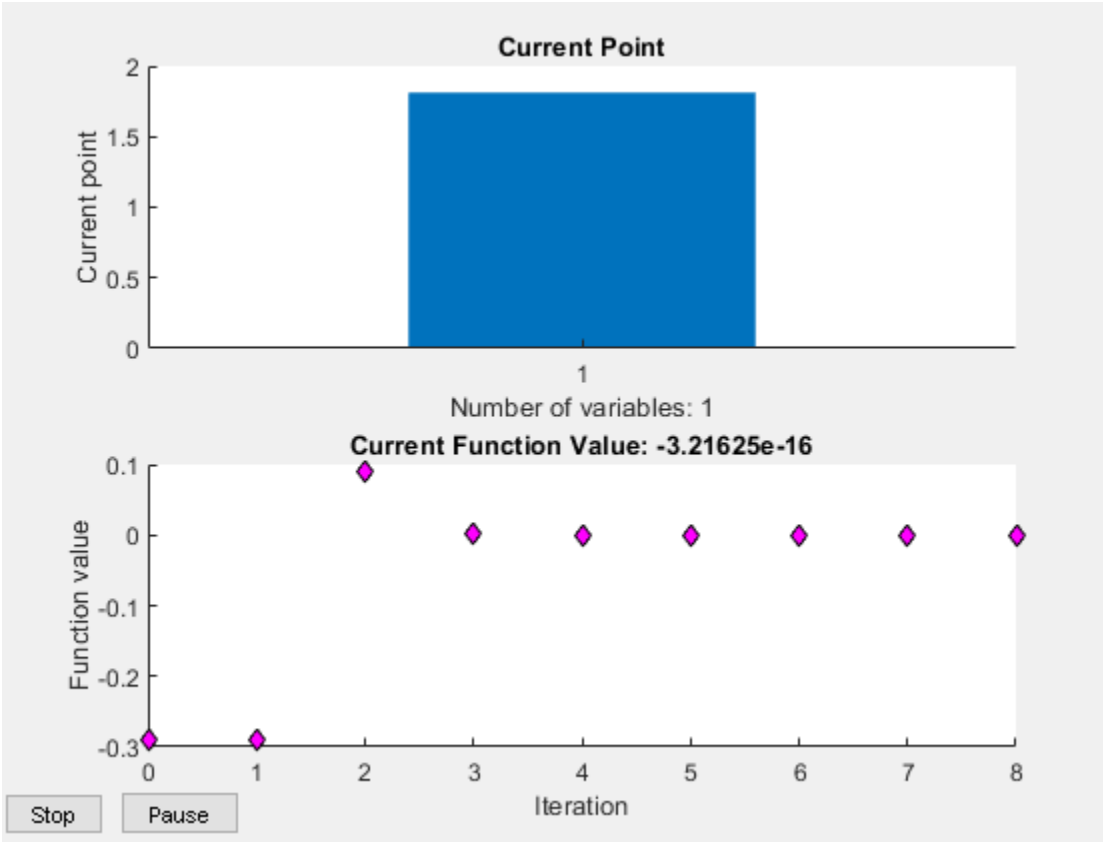
```
fun = @(x)sin(cosh(x));
x0 = 1;
```

Examine the solution process by setting options that include plot functions.

```
options = optimset('PlotFcns',{@optimplotx,@optimplotfval});
```

Run `fzero` including `options`.

```
x = fzero(fun,x0,options)
```




```
x = 1.8115
```

Solve Problem Structure

Solve a problem that is defined by a problem structure.

Define a structure that encodes a root-finding problem.

```
problem.objective = @(x)sin(cosh(x));
problem.x0 = 1;
problem.solver = 'fzero'; % a required part of the structure
problem.options = optimset(@fzero); % default options
```

Solve the problem.

```
x = fzero(problem)
```

```
x = 1.8115
```

More Information from Solution

Find the point where `exp(-exp(-x)) = x`, and display information about the solution process.

```
fun = @(x) exp(-exp(-x)) - x; % function
x0 = [0 1]; % initial interval
options = optimset('Display','iter'); % show iterations
[x fval exitflag output] = fzero(fun,x0,options)
```

```
 Func-count    x          f(x)             Procedure
    2               1     -0.307799        initial
    3        0.544459     0.0153522        interpolation
    4        0.566101    0.00070708        interpolation
    5        0.567143  -1.40255e-08        interpolation
    6        0.567143   1.50013e-12        interpolation
    7        0.567143             0        interpolation

Zero found in the interval [0, 1]

x = 0.5671

fval = 0

exitflag = 1

output = struct with fields:
    intervaliterations: 0
            iterations: 5
             funcCount: 7
             algorithm: 'bisection, interpolation'
               message: 'Zero found in the interval [0, 1]'
```

`fval` = 0 means `fun(x) = 0`, as desired.

Input Arguments

fun — Function to solve

function handle | function name

Function to solve, specified as a handle to a scalar-valued function or the name of such a function. `fun` accepts a scalar `x` and returns a scalar `fun(x)`.

`fzero` solves `fun(x) = 0`. To solve an equation `fun(x) = c(x)`, instead solve `fun2(x) = fun(x) - c(x) = 0`.

To include extra parameters in your function, see the example "Root of Function with Extra Parameter" on page 1-4740 and the section "Parameterizing Functions".

Example: `'sin'`

Example: `@myFunction`

Example: `@(x)(x-a)^5 - 3*x + a - 1`

Data Types: `char` | `function_handle` | `string`

x0 — Initial value

scalar | 2-element vector

Initial value, specified as a real scalar or a 2-element real vector.

- Scalar — `fzero` begins at `x0` and tries to locate a point `x1` where `fun(x1)` has the opposite sign of `fun(x0)`. Then `fzero` iteratively shrinks the interval where `fun` changes sign to reach a solution.
- 2-element vector — `fzero` checks that `fun(x0(1))` and `fun(x0(2))` have opposite signs, and errors if they do not. It then iteratively shrinks the interval where `fun` changes sign to reach a solution. An interval `x0` must be finite; it cannot contain `±Inf`.

Tip Calling `fzero` with an interval (`x0` with two elements) is often faster than calling it with a scalar `x0`.

Example: 3

Example: [2,17]

Data Types: `double`

`options` — Options for solution process

structure, typically created using `optimset`

Options for solution process, specified as a structure. Create or modify the `options` structure using `optimset`. `fzero` uses these `options` structure fields.

`Display`	Level of display: • `'off'` displays no output. • `'iter'` displays output at each iteration. • `'final'` displays just the final output. • `'notify'` (default) displays output only if the function does not converge.
`FunValCheck`	Check whether objective function values are valid. • `'on'` displays an error when the objective function returns a value that is `complex`, `Inf`, or `NaN`. • The default, `'off'`, displays no error.
`OutputFcn`	Specify one or more user-defined functions that an optimization function calls at each iteration, either as a function handle or as a cell array of function handles. The default is none (`[]`). See “Output Functions”.
`PlotFcns`	Plot various measures of progress while the algorithm executes. Select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none (`[]`). • `@optimplotx` plots the current point. • `@optimplotfval` plots the function value. For information on writing a custom plot function, see “Plot Functions”.
`TolX`	Termination tolerance on `x`, a positive scalar. The default is `eps`, 2.2204e–16.

Example: `options = optimset('FunValCheck','on')`

Data Types: `struct`

`problem` — Root-finding problem
structure

Root-finding problem, specified as a structure with all of the following fields.

`objective`	Objective function
`x0`	Initial point for `x`, real scalar or 2-element vector
`solver`	`'fzero'`
`options`	Options structure, typically created using `optimset`

For an example, see “Solve Problem Structure” on page 1-4741.

Data Types: `struct`

Output Arguments

`x` — Location of root or sign change
real scalar

Location of root or sign change, returned as a scalar.

`fval` — Function value at `x`
real scalar

Function value at `x`, returned as a scalar.

`exitflag` — Integer encoding the exit condition
integer

Integer encoding the exit condition, meaning the reason `fzero` stopped its iterations.

`1`	Function converged to a solution `x`.
`-1`	Algorithm was terminated by the output function or plot function.
`-3`	`NaN` or `Inf` function value was encountered while searching for an interval containing a sign change.
`-4`	Complex function value was encountered while searching for an interval containing a sign change.

`-5`	Algorithm might have converged to a singular point.
`-6`	`fzero` did not detect a sign change.

`output` — Information about root-finding process
structure

Information about root-finding process, returned as a structure. The fields of the structure are:

`intervaliterations`	Number of iterations taken to find an interval containing a root
`iterations`	Number of zero-finding iterations
`funcCount`	Number of function evaluations
`algorithm`	`'bisection, interpolation'`
`message`	Exit message

Algorithms

The `fzero` command is a function file. The algorithm, created by T. Dekker, uses a combination of bisection, secant, and inverse quadratic interpolation methods. An Algol 60 version, with some improvements, is given in [1]. A Fortran version, upon which `fzero` is based, is in [2].

References

[1] Brent, R., *Algorithms for Minimization Without Derivatives*, Prentice-Hall, 1973.

[2] Forsythe, G. E., M. A. Malcolm, and C. B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, 1976.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For C/C++ code generation:

- The `fun` input argument must be a function handle, and not a structure or character vector.
- `fzero` ignores all options except for `TolX` and `FunValCheck`.
- `fzero` does not support the fourth output argument, the output structure.

See Also

`fminbnd` | `optimset` | `roots`

Topics

"Roots of Scalar Functions"
"Parameterizing Functions"

Introduced before R2006a

matlab.unittest.FunctionTestCase class

Package: `matlab.unittest`
Superclasses: `matlab.unittest.TestCase`

TestCase used for function-based tests

Description

The `FunctionTestCase` class is a subclass of `TestCase` that allows function-based tests to use qualification functions in the `matlab.unittest.qualifications` package. For each test function, MATLAB creates an instance of the `FunctionTestCase` class and passes it to the test function.

The `functiontests` function constructs `FunctionTestCase` instances, so there is no need for test authors to construct this class directly.

See Also

`functiontests` | `matlab.unittest.TestCase` | `runtests`

Introduced in R2013b

gallery

Test matrices

Syntax

```
[A,B,C,...] = gallery(matname,P1,P2,...)
[A,B,C,...] = gallery(matname,P1,P2,...,classname)
gallery(3)
gallery(5)
```

Description

`[A,B,C,...] = gallery(matname,P1,P2,...)` returns the test matrices specified by `matname`. The `matname` input is the name of a matrix family selected from the table below. `P1,P2,...` are input parameters required by the individual matrix family. The number of optional parameters `P1,P2,...` used in the calling syntax varies from matrix to matrix. The exact calling syntaxes are detailed in the individual matrix descriptions below.

`[A,B,C,...] = gallery(matname,P1,P2,...,classname)` produces a matrix of class `classname`. The `classname` input must be either `'single'` or `'double'` (unless `matname` is `'integerdata'`, in which case `'int8'`, `'int16'`, `'int32'`, `'uint8'`, `'uint16'`, and `'uint32'` are also allowed). If `classname` is not specified, then the class of the matrix is determined from those arguments among `P1,P2,...` that do not specify dimensions or select an option. If any of these arguments is of class `single` then the matrix is `single`; otherwise the matrix is `double`.

`gallery(3)` is a badly conditioned 3-by-3 matrix and `gallery(5)` is an interesting eigenvalue problem.

The gallery holds over fifty different test matrix functions useful for testing algorithms and other purposes.

binomial — Multiple of involutory matrix

`A = gallery('binomial',n)` returns an `n`-by-`n` matrix,with integer entries such that `A^2 = 2^(n-1)*eye(n)`.

Thus, `B = A*2^((1-n)/2)` is involutory, that is, `B^2 = eye(n)`.

cauchy — Cauchy matrix

`C = gallery('cauchy',x,y)` returns an `n`-by-`n` matrix, `C(i,j) = 1/(x(i)+y(j))`. Arguments `x` and `y` are vectors of length `n`. If you pass in scalars for `x` and `y`, they are interpreted as vectors `1:x` and `1:y`.

`C = gallery('cauchy',x)` returns the same as above with `y = x`. That is, the command returns `C(i,j) = 1/(x(i)+x(j))`.

Explicit formulas are known for the inverse and determinant of a Cauchy matrix. The determinant `det(C)` is nonzero if `x` and `y` both have distinct elements. `C` is totally positive if `0 < x(1) <... < x(n)` and `0 < y(1) < ... < y(n)`.

chebspec — Chebyshev spectral differentiation matrix

`C = gallery('chebspec',n,switch)` returns a Chebyshev spectral differentiation matrix of order `n`. Argument `switch` is a variable that determines the character of the output matrix. By default, `switch = 0`.

For `switch = 0` (“no boundary conditions”), `C` is nilpotent (`C^n = 0`) and has the null vector `ones(n,1)`. The matrix `C` is similar to a Jordan block of size `n` with eigenvalue zero.

For `switch = 1`, `C` is nonsingular and well-conditioned, and its eigenvalues have negative real parts.

The eigenvector matrix of the Chebyshev spectral differentiation matrix is ill-conditioned.

chebvand — Vandermonde-like matrix for the Chebyshev polynomials

`C = gallery('chebvand',p)` produces the (primal) Chebyshev Vandermonde matrix based on the vector of points `p`, which define where the Chebyshev polynomial is calculated.

`C = gallery('chebvand',m,p)` where `m` is scalar, produces a rectangular version of the above, with `m` rows.

If `p` is a vector, then $C(i,j) = T_{i-1}(p(j))$ where T_{i-1} is the Chebyshev polynomial of degree $i - 1$. If `p` is a scalar, then `p` equally spaced points on the interval `[0,1]` are used to calculate `C`.

chow — Singular Toeplitz lower Hessenberg matrix

`A = gallery('chow',n,alpha,delta)` returns `A` such that `A = H(alpha) + delta*eye(n)`, where $H_{i,j}(\alpha) = \alpha^{(i-j+1)}$ and argument `n` is the order of the Chow matrix. Default value for scalars `alpha` and `delta` are `1` and `0`, respectively.

`H(alpha)` has `p = floor(n/2)` eigenvalues that are equal to zero. The rest of the eigenvalues are equal to `4*alpha*cos(k*pi/(n+2))^2, k=1:n-p`.

circul — Circulant matrix

`C = gallery('circul',v)` returns the circulant matrix whose first row is the vector `v`.

A circulant matrix has the property that each row is obtained from the previous one by cyclically permuting the entries one step forward. It is a special Toeplitz matrix in which the diagonals “wrap around.”

If `v` is a scalar, then `C = gallery('circul',1:v)`.

The eigensystem of `C` (`n`-by-`n`) is known explicitly: If `t` is an `n`th root of unity, then the inner product of `v` and $w = [1\ t\ t^2\ \ldots\ t^{(n-1)}]$ is an eigenvalue of `C` and `w(n:-1:1)` is an eigenvector.

clement — Tridiagonal matrix with zero diagonal entries

`A = gallery('clement',n,k)` returns an `n`-by-`n` tridiagonal matrix with zeros on its main diagonal and known eigenvalues. It is singular if `n` is odd. About 64 percent of the

entries of the inverse are zero. The eigenvalues include plus and minus the numbers `n-1`, `n-3`, `n-5`, `...`, (1 or 0).

For `k=0` (the default), `A` is nonsymmetric. For `k=1`, `A` is symmetric.

`gallery('clement',n,1)` is diagonally similar to `gallery('clement',n)`.

For odd `N = 2*M+1`, `M+1` of the singular values are the integers `sqrt((2*M+1)^2 - (2*K+1).^2)` for `K = 0:M`.

Note Similar properties hold for `gallery('tridiag',x,y,z)` where `y = zeros(n,1)`. The eigenvalues still come in plus/minus pairs but they are not known explicitly.

---

compar — Comparison matrices

`A = gallery('compar',A,1)` returns `A` with each diagonal element replaced by its absolute value, and each off-diagonal element replaced by minus the absolute value of the largest element in absolute value in its row. However, if `A` is triangular `compar(A,1)` is too.

`gallery('compar',A)` is `diag(B) - tril(B,-1) - triu(B,1)`, where `B = abs(A)`. `compar(A)` is often denoted by M(*A*) in the literature.

`gallery('compar',A,0)` is the same as `gallery('compar',A)`.

condex — Counter-examples to matrix condition number estimators

`A = gallery('condex',n,k,theta)` returns a “counter-example” matrix to a condition estimator. It has order `n` and scalar parameter `theta` (default 100).

The matrix, its natural size, and the estimator to which it applies are specified by `k`:

`k = 1`	4-by-4	LINPACK
`k = 2`	3-by-3	LINPACK
`k = 3`	arbitrary	LINPACK (`rcond`) (independent of `theta`)

`k = 4`	`n >= 4`	LAPACK (RCOND) (default). It is the inverse of this matrix that is a counter-example.

If `n` is not equal to the natural size of the matrix, then the matrix is padded out with an identity matrix to order `n`.

cycol — Matrix whose columns repeat cyclically

`A = gallery('cycol',[m n],k)` returns an `m`-by-`n` matrix with cyclically repeating columns, where one “cycle” consists of `randn(m,k)`. Thus, the rank of matrix `A` cannot exceed `k`, and `k` must be a scalar.

Argument `k` defaults to `round(n/4)`, and need not evenly divide `n`.

`A = gallery('cycol',n,k)`, where `n` is a scalar, is the same as `gallery('cycol', [n n],k)`.

dorr — Diagonally dominant, ill-conditioned, tridiagonal matrix

`[c,d,e] = gallery('dorr',n,theta)` returns the vectors defining an `n`-by-`n`, row diagonally dominant, tridiagonal matrix that is ill-conditioned for small nonnegative values of `theta`. The default value of `theta` is `0.01`. The Dorr matrix itself is the same as `gallery('tridiag',c,d,e)`.

`A = gallery('dorr',n,theta)` returns the matrix itself, rather than the defining vectors.

dramadah — Matrix of zeros and ones whose inverse has large integer entries

`A = gallery('dramadah',n,k)` returns an `n`-by-`n` matrix of `0`'s and `1`'s for which `mu(A) = norm(inv(A),'fro')` is relatively large, although not necessarily maximal. An anti-Hadamard matrix `A` is a matrix with elements `0` or `1` for which `mu(A)` is maximal.

`n` and `k` must both be scalars. Argument `k` determines the character of the output matrix:

k = 1	Default. A is Toeplitz, with `abs(det(A)) = 1`, and `mu(A) > c(1.75)^n`, where `c` is a constant. The inverse of A has integer entries.
k = 2	A is upper triangular and Toeplitz. The inverse of A has integer entries.
k = 3	A has maximal determinant among lower Hessenberg (0,1) matrices. `det(A)` = the nth Fibonacci number. A is Toeplitz. The eigenvalues have an interesting distribution in the complex plane.

fiedler — Symmetric matrix

`A = gallery('fiedler',c)`, where `c` is a length `n` vector, returns the `n`-by-`n` symmetric matrix with elements `abs(n(i)-n(j))`. For scalar `c`, `A = gallery('fiedler',1:c)`.

Matrix A has a dominant positive eigenvalue and all the other eigenvalues are negative.

Explicit formulas for `inv(A)` and `det(A)` are given in [Todd, J., *Basic Numerical Mathematics*, Vol. 2: Numerical Algebra, Birkhauser, Basel, and Academic Press, New York, 1977, p. 159] and attributed to Fiedler. These indicate that `inv(A)` is tridiagonal except for nonzero `(1,n)` and `(n,1)` elements.

forsythe — Perturbed Jordan block

`A = gallery('forsythe',n,alpha,lambda)` returns the `n`-by-`n` matrix equal to the Jordan block with eigenvalue `lambda`, excepting that `A(n,1) = alpha`. The default values of scalars `alpha` and `lambda` are `sqrt(eps)` and `0`, respectively.

The characteristic polynomial of A is given by:

```
det(A-t*I) = (lambda-t)^N - alpha*(-1)^n.
```

frank — Matrix with ill-conditioned eigenvalues

`F = gallery('frank',n,k)` returns the Frank matrix of order `n`. It is upper Hessenberg with determinant `1`. If `k = 1`, the elements are reflected about the anti-diagonal `(1,n)` — `(n,1)`. The eigenvalues of F may be obtained in terms of the zeros of the Hermite polynomials. They are positive and occur in reciprocal pairs; thus if `n` is odd, `1` is an eigenvalue. F has `floor(n/2)` ill-conditioned eigenvalues — the smaller ones.

gcdmat — Greatest common divisor matrix

`A = gallery('gcdmat',n)` returns the `n`-by-`n` matrix with `(i,j)` entry `gcd(i,j)`. Matrix`A` is symmetric positive definite, and `A.^r` is symmetric positive semidefinite for all nonnegative `r`.

gearmat — Gear matrix

`A = gallery('gearmat',n,i,j)` returns the `n`-by-`n` matrix with ones on the sub- and super-diagonals, `sign(i)` in the `(1,abs(i))` position, `sign(j)` in the `(n,n+1-abs(j))` position, and zeros everywhere else. Arguments `i` and `j` default to `n` and `-n`, respectively.

Matrix `A` is singular, can have double and triple eigenvalues, and can be defective.

All eigenvalues are of the form `2*cos(a)` and the eigenvectors are of the form `[sin(w+a), sin(w+2*a), ..., sin(w+n*a)]`, where `a` and `w` are given in Gear, C. W., "A Simple Set of Test Matrices for Eigenvalue Programs," *Math. Comp.*, Vol. 23 (1969), pp. 119-125.

grcar — Toeplitz matrix with sensitive eigenvalues

`A = gallery('grcar',n,k)` returns an `n`-by-`n` Toeplitz matrix with `-1`s on the subdiagonal, `1`s on the diagonal, and `k` superdiagonals of `1`s. The default is `k = 3`. The eigenvalues are sensitive.

hanowa — Matrix whose eigenvalues lie on a vertical line in the complex plane

`A = gallery('hanowa',n,d)` returns an `n`-by-`n` block 2-by-2 matrix of the form:

```
[d*eye(m) -diag(1:m)
diag(1:m) d*eye(m)]
```

Argument `n` is an even integer `n=2*m`. Matrix `A` has complex eigenvalues of the form `d ± k*i`, for `1 <= k <= m`. The default value of `d` is `-1`.

house — Householder matrix

`[v,beta,s] = gallery('house',x,k)` takes `x`, an `n`-element column vector, and returns `V` and `beta` such that `H*x = s*e1`. In this expression, `e1` is the first column of `eye(n)`, `abs(s) = norm(x)`, and `H = eye(n) - beta*V*V'` is a Householder matrix.

`k` determines the sign of `s`:

`k = 0`	`sign(s) = -sign(x(1))` (default)
`k = 1`	`sign(s) = sign(x(1))`
`k = 2`	`sign(s) = 1` (`x` must be real)

If `x` is complex, then `sign(x) = x./abs(x)` when `x` is nonzero.

If `x = 0`, or if `x = alpha*e1` (`alpha >= 0`) and either `k = 1` or `k = 2`, then `V = 0`, `beta = 1`, and `s = x(1)`. In this case, `H` is the identity matrix, which is not strictly a Householder matrix.

`[v, beta] = gallery('house',x)` takes `x`, a scalar or `n`-element column vector, and returns `v` and `beta` such that `eye(n,n) - beta*v*v'` is a Householder matrix. A Householder matrix `H` satisfies the relationship

```
H*x = -sign(x(1))*norm(x)*e1
```

where `e1` is the first column of `eye(n,n)`. Note that if `x` is complex, then `sign(x) = exp(i*arg(x))` (which equals `x./abs(x)` when `x` is nonzero).

If `x = 0`, then `v = 0` and `beta = 1`.

integerdata — Array of arbitrary data from uniform distribution on specified range of integers

`A = gallery('integerdata',imax,[m,n,...],j)` returns an `m`-by-`n`-by-... array `A` whose values are a sample from the uniform distribution on the integers `1:imax`. `j` must be an integer value in the interval `[0, 2^32-1]`. Calling `gallery('integerdata', ...)` with different values of `J` will return different arrays. Repeated calls to `gallery('integerdata',...)` with the same `imax`, size vector and j inputs will always return the same array.

In any call to `gallery('integerdata', ...)` you can substitute individual inputs `m`,`n`,... for the size vector input `[m,n,...]`. For example, `gallery('integerdata',7,[1,2,3,4],5)` is equivalent to `gallery('integerdata',7,1,2,3,4,5)`.

`A = gallery('integerdata',[imin imax],[m,n,...],j)` returns an `m`-by-`n`-by-... array `A` whose values are a sample from the uniform distribution on the integers `imin:imax`.

`[A,B,...] = gallery('integerdata',[imin imax],[m,n,...],j)` returns multiple `m`-by-`n`-by-... arrays `A`, `B`, ..., containing different values.

`A = gallery('integerdata',[imin imax],[m,n,...],j,classname)`produces an array of class `classname`. `classname` must be `'uint8'`, `'uint16'`, `'uint32'`, `'int8'`, `'int16'`, `int32'`, `'single'` or `'double'`.

invhess — Inverse of an upper Hessenberg matrix

`A = gallery('invhess',x,y)`, where `x` is a length `n` vector and `y` is a length `n-1` vector, returns the matrix whose lower triangle agrees with that of `ones(n,1)*x'` and whose strict upper triangle agrees with that of `[1 y]*ones(1,n)`.

The matrix is nonsingular if `x(1) ~= 0` and `x(i+1) ~= y(i)` for all `i`, and its inverse is an upper Hessenberg matrix. Argument `y` defaults to `-x(1:n-1)`.

If `x` is a scalar, `invhess(x)` is the same as `invhess(1:x)`.

invol — Involutory matrix

`A = gallery('invol',n)` returns an `n`-by-`n` involutory `(A*A = eye(n))` and ill-conditioned matrix. It is a diagonally scaled version of `hilb(n)`.

`B = (eye(n)-A)/2` and `B = (eye(n)+A)/2` are idempotent `(B*B = B)`.

ipjfact — Hankel matrix with factorial elements

`[A,d] = gallery('ipjfact',n,k)` returns `A`, an `n`-by-`n` Hankel matrix, and `d`, the determinant of `A`, which is known explicitly. If `k = 0` (the default), then the elements of `A` are `A(i,j) = (i+j)!`. If `k = 1`, then the elements of `A` are `A(i,j) = 1/(i+j)`.

Note that the inverse of `A` is also known explicitly.

jordbloc — Jordan block

`A = gallery('jordbloc',n,lambda)` returns the `n`-by-`n` Jordan block with eigenvalue `lambda`. The default value for `lambda` is `1`.

kahan — Upper trapezoidal matrix

`A = gallery('kahan',n,theta,pert)` returns an upper trapezoidal matrix that has interesting properties regarding estimation of condition and rank.

If `n` is a two-element vector, then `A` is `n(1)`-by-`n(2)`; otherwise, `A` is `n`-by-`n`. The useful range of `theta` is `0 < theta < pi`, with a default value of `1.2`.

To ensure that the QR factorization with column pivoting does not interchange columns in the presence of rounding errors, the diagonal is perturbed by `pert*eps*diag([n:-1:1])`. The default `pert` is `25`, which ensures no interchanges for `gallery('kahan',n)` up to at least `n = 90` in IEEE arithmetic.

kms — Kac-Murdock-Szego Toeplitz matrix

`A = gallery('kms',n,rho)` returns the `n`-by-`n` Kac-Murdock-Szego Toeplitz matrix such that `A(i,j) = rho^(abs(i-j))`, for real `rho`.

For complex `rho`, the same formula holds except that elements below the diagonal are conjugated. `rho` defaults to 0.5.

The KMS matrix `A` has these properties:

- An LDL' factorization with `L = inv(gallery('triw',n,-rho,1))'`, and `D(i,i) = (1-abs(rho)^2)*eye(n)`, except `D(1,1) = 1`.
- Positive definite if and only if `0 < abs(rho) < 1`.
- The inverse `inv(A)` is tridiagonal.

krylov — Krylov matrix

`B = gallery('krylov',A,x,j)` returns the Krylov matrix

```
[x, Ax, A^2x, ..., A^(j-1)x]
```

where `A` is an `n`-by-`n` matrix and `x` is a length `n` vector. The defaults are `x = ones(n,1)`, and `j = n`.

`B = gallery('krylov',n)` is the same as `gallery('krylov',randn(n))`.

lauchli — Rectangular matrix

`A = gallery('lauchli',n,mu)` returns the `(n+1)`-by-`n` matrix

```
[ones(1,n); mu*eye(n)]
```

The Lauchli matrix is a well-known example in least squares and other problems that indicates the dangers of forming `A'*A`. Argument `mu` defaults to `sqrt(eps)`.

lehmer — Symmetric positive definite matrix

`A = gallery('lehmer',n)` returns the symmetric positive definite `n`-by-`n` matrix such that `A(i,j) = i/j` for `j >= i`.

The Lehmer matrix `A` has these properties:

- `A` is totally nonnegative.
- The inverse `inv(A)` is tridiagonal and explicitly known.
- The order `n <= cond(A) <= 4*n*n`.

leslie — Matrix of birth numbers and survival rates

`L = gallery('leslie',a,b)` is the `n`-by-`n` matrix from the Leslie population model with average birth numbers `a(1:n)` and survival rates `b(1:n-1)`. It is zero, apart from the first row (which contains the `a(i)`) and the first subdiagonal (which contains the `b(i)`). For a valid model, the `a(i)` are nonnegative and the `b(i)` are positive and bounded by 1, i.e., `0 < b(i) <= 1`.

`L = gallery('leslie',n)` generates the Leslie matrix with `a = ones(n,1)`, `b = ones(n-1,1)`.

lesp — Tridiagonal matrix with real, sensitive eigenvalues

`A = gallery('lesp',n)` returns an `n`-by-`n` matrix whose eigenvalues are real and smoothly distributed in the interval approximately `[-2*N-3.5, -4.5]`.

The sensitivities of the eigenvalues increase exponentially as the eigenvalues grow more negative. The matrix is similar to the symmetric tridiagonal matrix with the same diagonal entries and with off-diagonal entries 1, via a similarity transformation with `D = diag(1!,2!,...,n!)`.

lotkin — Lotkin matrix

`A = gallery('lotkin',n)` returns the Hilbert matrix with its first row altered to all ones. The Lotkin matrix `A` is nonsymmetric, ill-conditioned, and has many negative eigenvalues of small magnitude. Its inverse has integer entries and is known explicitly.

minij — Symmetric positive definite matrix

`A = gallery('minij',n)` returns the `n`-by-`n` symmetric positive definite matrix with `A(i,j) = min(i,j)`.

The `minij` matrix has these properties:

- The inverse `inv(A)` is tridiagonal and equal to `-1` times the second difference matrix, except its `(n,n)` element is `1`.
- Givens' matrix, `2*A-ones(size(A))`, has tridiagonal inverse and eigenvalues `0.5*sec((2*r-1)*pi/(4*n))^2`, where `r=1:n`.
- `(n+1)*ones(size(A))-A` has elements that are `max(i,j)` and a tridiagonal inverse.

moler — Symmetric positive definite matrix

`A = gallery('moler',n,alpha)` returns the symmetric positive definite `n`-by-`n` matrix `U'*U`, where `U = gallery('triw',n,alpha)`.

For the default `alpha = -1`, `A(i,j) = min(i,j)-2`, and `A(i,i) = i`. One of the eigenvalues of `A` is small.

neumann — Singular matrix from the discrete Neumann problem (sparse)

`C = gallery('neumann',n)` returns the sparse `n`-by-`n` singular, row diagonally dominant matrix resulting from discretizing the Neumann problem with the usual five-

point operator on a regular mesh. Argument `n` is a perfect square integer $n = m^2$ or a two-element vector. `C` is sparse and has a one-dimensional null space with null vector `ones(n,1)`.

normaldata — Array of arbitrary data from standard normal distribution

`A = gallery('normaldata',[m,n,...],j)` returns an `m`-by-`n`-by-... array `A`. The values of `A` are a random sample from the standard normal distribution. `j` must be an integer value in the interval `[0, 2^32-1]`. Calling `gallery('normaldata', ...)` with different values of `j` will return different arrays. Repeated calls to `gallery('normaldata',...)` with the same size vector and `j` inputs will always return the same array.

In any call to `gallery('normaldata', ...)` you can substitute individual inputs `m`,`n`,... for the size vector input `[m,n,...]`. For example, `gallery('normaldata', [1,2,3,4],5)` is equivalent to `gallery('normaldata',1,2,3,4,5)`.

`[A,B,...] = gallery('normaldata',[m,n,...],j)` returns multiple `m`-by-`n`-by-... arrays `A`, `B`, ..., containing different values.

`A = gallery('normaldata',[m,n,...],j, classname)` produces a matrix of class `classname`. `classname` must be either `'single'` or `'double'`.

Generate the arbitrary 6-by-4 matrix of data from the standard normal distribution `N(0, 1)` corresponding to `j = 2`:.

```
x = gallery('normaldata', [6, 4], 2);
```

Generate the arbitrary 1-by-2-by-3 single array of data from the standard normal distribution `N(0, 1)` corresponding to `j = 17`:.

```
y = gallery('normaldata', 1, 2, 3, 17, 'single');
```

orthog — Orthogonal and nearly orthogonal matrices

`Q = gallery('orthog',n,k)` returns the kth type of matrix of order `n`, where `k > 0` selects exactly orthogonal matrices, and `k < 0` selects diagonal scalings of orthogonal matrices. Available types are:

`k = 1`	`Q(i,j) = sqrt(2/(n+1)) * sin(i*j*pi/(n+1))` Symmetric eigenvector matrix for second difference matrix. This is the default.
`k = 2`	`Q(i,j) = 2/(sqrt(2*n+1)) * sin(2*i*j*pi/(2*n+1))` Symmetric.
`k = 3`	`Q(r,s) = exp(2*pi*i*(r-1)*(s-1)/n) / sqrt(n)` Unitary, the Fourier matrix. Q^4 is the identity. This is essentially the same matrix as `fft(eye(n))/sqrt(n)`!
`k = 4`	Helmert matrix: a permutation of a lower Hessenberg matrix, whose first row is `ones(1:n)/sqrt(n)`.
`k = 5`	`Q(i,j) = sin(2*pi*(i-1)*(j-1)/n) + cos(2*pi*(i-1)*(j-1)/n)` Symmetric matrix arising in the Hartley transform.
`k = 6`	`Q(i,j) = sqrt(2/n)*cos((i-1/2)*(j-1/2)*pi/n)` Symmetric matrix arising as a discrete cosine transform.
`k = -1`	`Q(i,j) = cos((i-1)*(j-1)*pi/(n-1))` Chebyshev Vandermonde-like matrix, based on extrema of `T(n-1)`.
`k = -2`	`Q(i,j) = cos((i-1)*(j-1/2)*pi/n)` Chebyshev Vandermonde-like matrix, based on zeros of `T(n)`.

parter — Toeplitz matrix with singular values near pi

`C = gallery('parter',n)` returns the matrix `C` such that `C(i,j) = 1/(i-j+0.5)`.

`C` is a Cauchy matrix and a Toeplitz matrix. Most of the singular values of `C` are very close to `pi`.

pei — Pei matrix

`A = gallery('pei',n,alpha)`, where `alpha` is a scalar, returns the symmetric matrix `alpha*eye(n) + ones(n)`. The default for `alpha` is 1. The matrix is singular for `alpha` equal to either `0` or `-n`.

poisson — Block tridiagonal matrix from Poisson's equation (sparse)

`A = gallery('poisson',n)` returns the block tridiagonal (sparse) matrix of order `n^2` resulting from discretizing Poisson's equation with the 5-point operator on an `n`-by-`n` mesh.

prolate — Symmetric, ill-conditioned Toeplitz matrix

`A = gallery('prolate',n,w)` returns the `n`-by-`n` prolate matrix with parameter `w`. It is a symmetric Toeplitz matrix.

If `0 < w < 0.5` then `A` is positive definite

- The eigenvalues of `A` are distinct, lie in `(0,1)`, and tend to cluster around `0` and `1`.
- The default value of `w` is 0.25.

randcolu — Random matrix with normalized cols and specified singular values

`A = gallery('randcolu',n)` is a random `n`-by-`n` matrix with columns of unit 2-norm, with random singular values whose squares are from a uniform distribution.

`A'*A` is a correlation matrix of the form produced by `gallery('randcorr',n)`.

`gallery('randcolu',x)` where `x` is an `n`-vector (`n` > 1), produces a random `n`-by-`n` matrix having singular values given by the vector `x`. The vector `x` must have nonnegative elements whose sum of squares is `n`.

`gallery('randcolu',x,m)` where `m >= n`, produces an `m`-by-`n` matrix.

`gallery('randcolu',x,m,k)` provides a further option:

`k = 0`	`diag(x)` is initially subjected to a random two-sided orthogonal transformation, and then a sequence of Givens rotations is applied (default).
`k = 1`	The initial transformation is omitted. This is much faster, but the resulting matrix may have zero entries.

For more information, see [4].

randcorr — Random correlation matrix with specified eigenvalues

`gallery('randcorr',n)` is a random `n`-by-`n` correlation matrix with random eigenvalues from a uniform distribution. A correlation matrix is a symmetric positive semidefinite matrix with `1`s on the diagonal (see `corrcoef`).

`gallery('randcorr',x)` produces a random correlation matrix having eigenvalues given by the vector `x`, where `length(x) > 1`. The vector `x` must have nonnegative elements summing to `length(x)`.

`gallery('randcorr',x,k)` provides a further option:

`k = 0`	The diagonal matrix of eigenvalues is initially subjected to a random orthogonal similarity transformation, and then a sequence of Givens rotations is applied (default).
`k = 1`	The initial transformation is omitted. This is much faster, but the resulting matrix may have some zero entries.

For more information, see [3] and [4].

randhess — Random, orthogonal upper Hessenberg matrix

`H = gallery('randhess',n)` returns an `n`-by-`n` real, random, orthogonal upper Hessenberg matrix.

`H = gallery('randhess',x)` if `x` is an arbitrary, real, length `n` vector with `n > 1`, constructs `H` nonrandomly using the elements of `x` as parameters.

Matrix `H` is constructed via a product of `n-1` Givens rotations.

randjorth — Random J-orthogonal matrix

`A = gallery('randjorth', n)`, for a positive integer `n`, produces a random `n`-by-`n` `J`-orthogonal matrix `A`, where

- `J = blkdiag(eye(ceil(n/2)),-eye(floor(n/2)))`
- `cond(A) = sqrt(1/eps)`

`J`-orthogonality means that A'*J*A = J. Such matrices are sometimes called *hyperbolic*.

`A = gallery('randjorth', n, m)`, for positive integers `n` and `m`, produces a random (`n`+`m`)-by-(`n`+`m`) `J`-orthogonal matrix `A`, where

- `J = blkdiag(eye(n),-eye(m))`
- `cond(A) = sqrt(1/eps)`

```
A = gallery('randjorth',n,m,c,symm,method)
```

uses the following optional input arguments:

- `c` — Specifies `cond(A)` to be the scalar `c`.
- `symm` — Enforces symmetry if the scalar `symm` is nonzero.
- `method` — calls `qr` to perform the underlying orthogonal transformations if the scalar `method` is nonzero. A call to `qr` is much faster than the default method for large dimensions

rando — Random matrix composed of elements -1, 0 or 1

`A = gallery('rando',n,k)` returns a random `n`-by-`n` matrix with elements from one of the following discrete distributions:

`k = 1`	`A(i,j) = 0` or `1` with equal probability (default).
`k = 2`	`A(i,j) = -1` or `1` with equal probability.
`k = 3`	`A(i,j) = -1`, `0` or `1` with equal probability.

Argument `n` may be a two-element vector, in which case the matrix is `n(1)`-by-`n(2)`.

randsvd — Random matrix with preassigned singular values

`A = gallery('randsvd',n,kappa,mode,kl,ku)` returns a banded (multidiagonal) random matrix of order `n` with `cond(A) = kappa` and singular values from the distribution `mode`. If `n` is a two-element vector, `A` is `n(1)`-by-`n(2)`.

Arguments `kl` and `ku` specify the number of lower and upper off-diagonals, respectively, in `A`. If they are omitted, a full matrix is produced. If only `kl` is present, `ku` defaults to `kl`.

Distribution `mode` can be:

1	One large singular value.
2	One small singular value.
3	Geometrically distributed singular values (default).
4	Arithmetically distributed singular values.
5	Random singular values with uniformly distributed logarithm.
`< 0`	If `mode` is `-1`, `-2`, `-3`, `-4`, or `-5`, then `randsvd` treats `mode` as `abs(mode)`, except that in the original matrix of singular values the order of the diagonal entries is reversed: small to large instead of large to small.

Condition number `kappa` defaults to `sqrt(1/eps)`. In the special case where `kappa < 0`, `A` is a random, full, symmetric, positive definite matrix with `cond(A) = -kappa` and eigenvalues distributed according to `mode`. Arguments `kl` and `ku`, if present, are ignored.

`A = gallery('randsvd',n,kappa,mode,kl,ku,method)` specifies how the computations are carried out. `method = 0` is the default, while `method = 1` uses an alternative method that is much faster for large dimensions, even though it uses more flops.

redheff — Redheffer's matrix of 1s and 0s

`A = gallery('redheff',n)` returns an `n`-by-`n` matrix of `0`'s and `1`'s defined by `A(i,j) = 1`, if `j = 1` or if `i` divides `j`, and `A(i,j) = 0` otherwise.

The Redheffer matrix has these properties:

- `(n-floor(log2(n)))-1` eigenvalues equal to `1`
- A real eigenvalue (the spectral radius) approximately `sqrt(n)`

- A negative eigenvalue approximately `-sqrt(n)`
- The remaining eigenvalues are provably "small."
- The Riemann hypothesis is true if and only if $\det(A) = O(n^{1/2+\varepsilon})$ for every $\varepsilon >$ `0`.

Barrett and Jarvis conjecture that "the small eigenvalues all lie inside the unit circle `abs(Z) = 1`," and a proof of this conjecture, together with a proof that some eigenvalue tends to zero as `n` tends to infinity, would yield a new proof of the prime number theorem.

riemann — Matrix associated with the Riemann hypothesis

`A = gallery('riemann',n)` returns an `n`-by-`n` matrix for which the Riemann hypothesis is true if and only if

$$\det(A) = O(n!n^{-1/2+\varepsilon})$$

for every $\varepsilon > 0$.

The Riemann matrix is defined by:

```
A = B(2:n+1,2:n+1)
```

where `B(i,j) = i-1` if `i` divides `j`, and `B(i,j) = -1` otherwise.

The Riemann matrix has these properties:

- Each eigenvalue `e(i)` satisfies `abs(e(i)) <= m-1/m`, where `m = n+1`.
- `i <= e(i) <= i+1` with at most `m-sqrt(m)` exceptions.
- All integers in the interval `(m/3, m/2]` are eigenvalues.

ris — Symmetric Hankel matrix

`A = gallery('ris',n)` returns a symmetric `n`-by-`n` Hankel matrix with elements

```
A(i,j) = 0.5/(n-i-j+1.5)
```

The eigenvalues of `A` cluster around $\pi/2$ and $-\pi/2$. This matrix was invented by F.N. Ris.

sampling — Nonsymmetric matrix with ill-conditioned integer eigenvalues.

`A = gallery('sampling',x)`, where `x` is an `n`-vector, is the `n`-by-`n` matrix with `A(i,j) = X(i)/(X(i)-X(j))` for `i ~= j` and `A(j,j)` the sum of the off-diagonal elements in column `j`. A has eigenvalues `0:n-1`. For the eigenvalues 0 and `n`–1, corresponding eigenvectors are `X` and `ones(n,1)`, respectively.

The eigenvalues are ill-conditioned. `A` has the property that `A(i,j) + A(j,i) = 1` for `i ~= j`.

Explicit formulas are available for the left eigenvectors of `A`. For scalar `n`, `gallery('sampling',n)` is the same as `gallery('sampling',1:n)`. A special case of this matrix arises in sampling theory.

smoke — Complex matrix with a 'smoke ring' pseudospectrum

`A = gallery('smoke',n)` returns an `n`-by-`n` matrix with `1`'s on the superdiagonal, `1` in the `(n,1)` position, and powers of roots of unity along the diagonal.

`A = gallery('smoke',n,1)` returns the same except that element `A(n,1)` is zero.

The eigenvalues of `gallery('smoke',n,1)` are the `n`th roots of unity; those of `gallery('smoke',n)` are the `n`th roots of unity times `2^(1/n)`.

toeppd — Symmetric positive definite Toeplitz matrix

`A = gallery('toeppd',n,m,w,theta)` returns an `n`-by-`n` symmetric, positive semi-definite (SPD) Toeplitz matrix composed of the sum of `m` rank 2 (or, for certain `theta`, rank 1) SPD Toeplitz matrices. Specifically,

```
T = w(1)*T(theta(1)) + ... + w(m)*T(theta(m))
```

where `T(theta(k))` has `(i,j)` element `cos(2*pi*theta(k)*(i-j))`.

By default: `m = n`, `w = rand(m,1)`, and `theta = rand(m,1)`.

toeppen — Pentadiagonal Toeplitz matrix (sparse)

`P = gallery('toeppen',n,a,b,c,d,e)` returns the `n`-by-`n` sparse, pentadiagonal Toeplitz matrix with the diagonals: `P(3,1) = a`, `P(2,1) = b`, `P(1,1) = c`, `P(1,2) = d`, and `P(1,3) = e`, where `a`, `b`, `c`, `d`, and `e` are scalars.

By default, `(a,b,c,d,e) = (1,-10,0,10,1)`, yielding a matrix of Rutishauser. This matrix has eigenvalues lying approximately on the line segment `2*cos(2*t) + 20*i*sin(t)`.

tridiag — Tridiagonal matrix (sparse)

`A = gallery('tridiag',c,d,e)` returns the tridiagonal matrix with subdiagonal `c`, diagonal `d`, and superdiagonal `e`. Vectors `c` and `e` must have `length(d)-1`.

`A = gallery('tridiag',n,c,d,e)`, where `c`, `d`, and `e` are all scalars, yields the Toeplitz tridiagonal matrix of order `n` with subdiagonal elements `c`, diagonal elements `d`, and superdiagonal elements `e`. This matrix has eigenvalues

```
d + 2*sqrt(c*e)*cos(k*pi/(n+1))
```

where `k = 1:n`. (see [1].)

`A = gallery('tridiag',n)` is the same as `A = gallery('tridiag',n,-1,2,-1)`, which is a symmetric positive definite M-matrix (the negative of the second difference matrix).

triw — Upper triangular matrix discussed by Wilkinson and others

`A = gallery('triw',n,alpha,k)` returns the upper triangular matrix with ones on the diagonal and `alpha`s on the first `k >= 0` superdiagonals.

Order `n` may be a 2-element vector, in which case the matrix is `n(1)`-by-`n(2)` and upper trapezoidal.

Ostrowski [“On the Spectrum of a One-parametric Family of Matrices,” *J. Reine Angew. Math.,* 1954] shows that

```
cond(gallery('triw',n,2)) = cot(pi/(4*n))^2,
```

and, for large `abs(alpha)`, `cond(gallery('triw',n,alpha))` is approximately `abs(alpha)^n*sin(pi/(4*n-2))`.

Adding `-2^(2-n)` to the `(n,1)` element makes `triw(n)` singular, as does adding `-2^(1-n)` to all the elements in the first column.

uniformdata — Array of arbitrary data from standard uniform distribution

`A = gallery('uniformdata',[m,n,...],j)` returns an `m`-by-`n`-by-... array `A`. The values of `A` are a random sample from the standard uniform distribution. j must be an integer value in the interval `[0, 2^32-1]`. Calling `gallery('uniformdata', ...)` with different values of `j` will return different arrays. Repeated calls to `gallery('uniformdata',...)` with the same size vector and `j` inputs will always return the same array.

In any call to `gallery('uniformdata', ...)` you can substitute individual inputs `m`,`n`,... for the size vector input `[m,n,...]`. For example, `gallery('uniformdata', [1,2,3,4],5)` is equivalent to `gallery('uniformdata',1,2,3,4,5)`.

`[A,B,...] = gallery('uniformdata',[m,n,...],j)` returns multiple `m`-by-`n`-by-... arrays `A`, `B`, ..., containing different values.

`A = gallery('uniformdata',[m,n,...],j, classname)` produces a matrix of class `classname`. `classname` must be either `'single'` or `'double'`.

Generate the arbitrary 6-by-4 matrix of data from the uniform distribution on [0, 1] corresponding to `j = 2`.

```
x = gallery('uniformdata', [6, 4], 2);
```

Generate the arbitrary 1-by-2-by-3 single array of data from the uniform distribution on [0, 1] corresponding to `j = 17`.

```
y = gallery('uniformdata', 1, 2, 3, 17, 'single');
```

wathen — Finite element matrix (sparse, random entries)

`A = gallery('wathen',nx,ny)` returns a sparse, random, `n`-by-`n` finite element matrix where `n = 3*nx*ny + 2*nx + 2*ny + 1`.

Matrix `A` is precisely the “consistent mass matrix” for a regular `nx`-by-`ny` grid of 8-node (serendipity) elements in two dimensions. `A` is symmetric, positive definite for any (positive) values of the “density,” `rho(nx,ny)`, which is chosen randomly in this routine.

`A = gallery('wathen',nx,ny,1)` returns a diagonally scaled matrix such that

```
0.25 <= eig(inv(D)*A) <= 4.5
```

where `D = diag(diag(A))` for any positive integers `nx` and `ny` and any densities `rho(nx,ny)`.

wilk — Various matrices devised or discussed by Wilkinson

`gallery('wilk',n)` returns a different matrix or linear system depending on the value of `n`.

`n = 3`	Upper triangular system `Ux=b` illustrating inaccurate solution.
`n = 4`	Lower triangular system `Lx=b`, ill-conditioned.
`n = 5`	`hilb(6)(1:5,2:6)*1.8144`. A symmetric positive definite matrix.
`n = 21`	`W21+`, a tridiagonal matrix. eigenvalue problem. For more detail, see [2].

References

[1] The *MATLAB* gallery of test matrices is based upon the work of Nicholas J. Higham at the Department of Mathematics, University of Manchester, Manchester, England. Further background can be found in the books *MATLAB Guide, Second Edition*, Desmond J. Higham and Nicholas J. Higham, SIAM, 2005, and *Accuracy and Stability of Numerical Algorithms,* Nicholas J. Higham, SIAM, 1996.

[2] Wilkinson, J. H., *The Algebraic Eigenvalue Problem*, Oxford University Press, London, 1965, p.308.

[3] Bendel, R. B. and M. R. Mickey, “Population Correlation Matrices for Sampling Experiments,” *Commun. Statist. Simulation Comput.*, B7, 1978, pp. 163-182.

[4] Davies, P. I. and N. J. Higham, “Numerically Stable Generation of Correlation Matrices and Their Factors,” *BIT*, Vol. 40, 2000, pp. 640-651.

See Also

`hadamard` | `hilb` | `invhilb` | `magic` | `wilkinson`

Introduced before R2006a

gamma

Gamma function

Syntax

```
Y = gamma(X)
```

Description

`Y = gamma(X)` returns the `gamma` function evaluated at the elements of `X`.

Examples

Evaluate Gamma Function

Evaluate the gamma function with a scalar and a vector.

Evaluate $\Gamma(0.5)$, which is equal to $\sqrt{\pi}$.

```
y = gamma(0.5)
```

```
y = 1.7725
```

Evaluate several values of the gamma function between `[-3.5 3.5]`.

```
x = -3.5:3.5;
y = gamma(x)
```

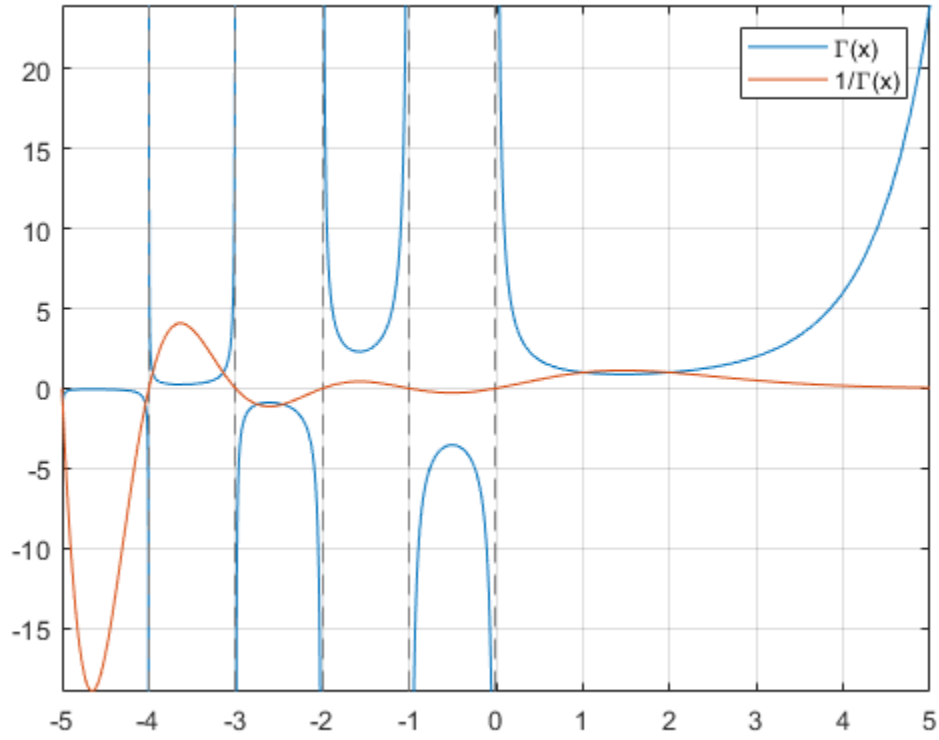
```
y = 1×8

    0.2701   -0.9453    2.3633   -3.5449    1.7725    0.8862    1.3293    3.3234
```

Plot Gamma Function

Plot the gamma function and its inverse.

Use `fplot` to plot the gamma function and its inverse. The gamma function increases quickly for positive arguments and has simple poles at all negative integer arguments (as well as 0). The function does not have any zeros. Conversely, the inverse gamma function has zeros at all negative integer arguments (as well as 0).

```
fplot(@gamma)
hold on
fplot(@(x) 1./gamma(x))
legend('\Gamma(x)','1/\Gamma(x)')
hold off
grid on
```



Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. The elements of X must be real.

Data Types: `single` | `double`

More About

Gamma Function

The `gamma` function is defined for real `x > 0` by the integral:

$$\Gamma(x) = \int_0^\infty e^{-t} t^{x-1} dt$$

The `gamma` function interpolates the `factorial` function. For integer `n`:

```
gamma(n+1) = factorial(n) = prod(1:n)
```

The domain of the `gamma` function extends to negative real numbers by analytic continuation, with simple poles at the negative integers. This extension arises from repeated application of the recursion relation

$$\Gamma(n-1) = \frac{\Gamma(n)}{n-1} \ .$$

Algorithms

The computation of `gamma` is based on algorithms outlined in [1].

References

[1] Cody, J., *An Overview of Software Development for Special Functions*, Lecture Notes in Mathematics, 506, Numerical Analysis Dundee, G. A. Watson (ed.), Springer Verlag, Berlin, 1976.

[2] Abramowitz, M. and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965, sec. 6.5.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`factorial` | `gammainc` | `gammaincinv` | `gammaln` | `psi`

Introduced before R2006a

gammainc

Incomplete gamma function

Syntax

```
Y = gammainc(X,A)
Y = gammainc(X,A,type)
Y = gammainc(X,A,scale)
```

Description

`Y = gammainc(X,A)` returns the lower incomplete gamma function on page 1-4784 evaluated at the elements of `X` and `A`. Both `X` and `A` must be real, and `A` must be nonnegative.

`Y = gammainc(X,A,type)` returns the lower or upper incomplete gamma function. The choices for `type` are `'lower'` (the default) and `'upper'`.

`Y = gammainc(X,A,scale)` scales the resulting lower or upper incomplete gamma function to avoid underflow to zero or loss of accuracy. The choices for `scale` are `'scaledlower'` and `'scaledupper'`.

Examples

Plot Lower Incomplete Gamma Function

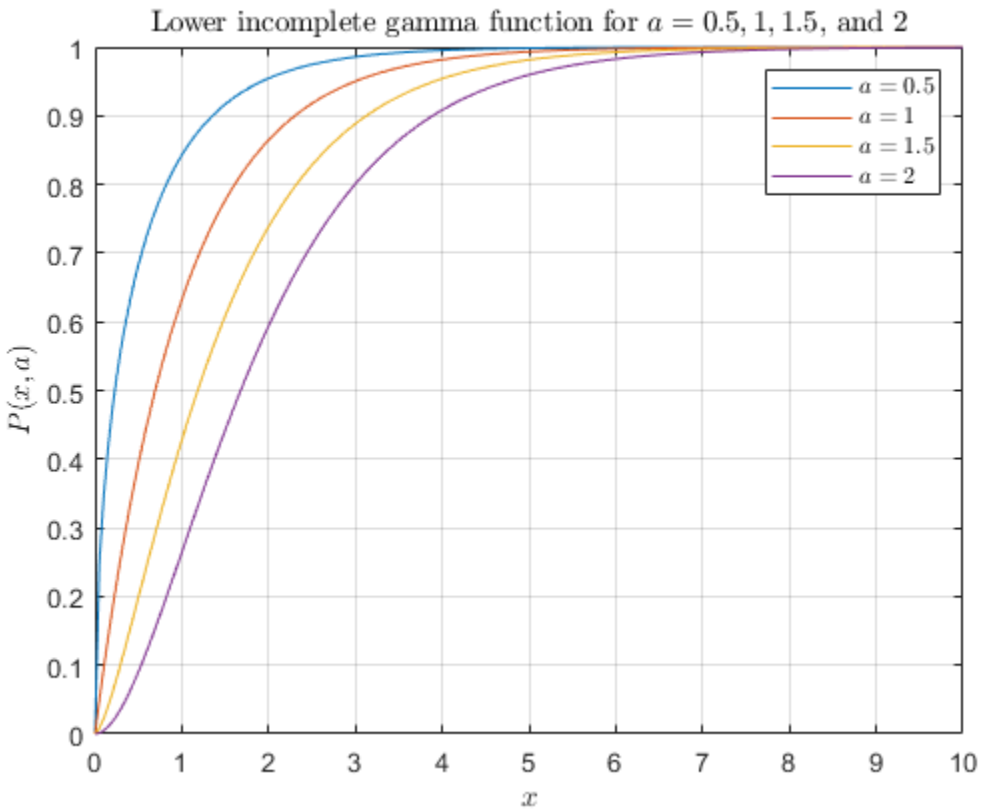
Calculate the lower incomplete gamma function for a = 0.5, 1, 1.5, and 2 within the interval $0 \le x \le 10$. Loop over values of a, evaluate the function at each one, and assign each result to a column of `Y`.

```
A = [0.5 1 1.5 2];
X = 0:0.05:10;
Y = zeros(201,4);
for i = 1:4
```

```
    Y(:,i) = gammainc(X,A(i));
end
```

Plot all of the functions in the same figure.

```
plot(X,Y)
grid on
legend('$a = 0.5$','$a = 1$','$a = 1.5$','$a = 2$','interpreter','latex')
title('Lower incomplete gamma function for $a = 0.5, 1, 1.5,$ and $2$','interpreter','l
xlabel('$x$','interpreter','latex')
ylabel('$P(x,a)$','interpreter','latex')
```



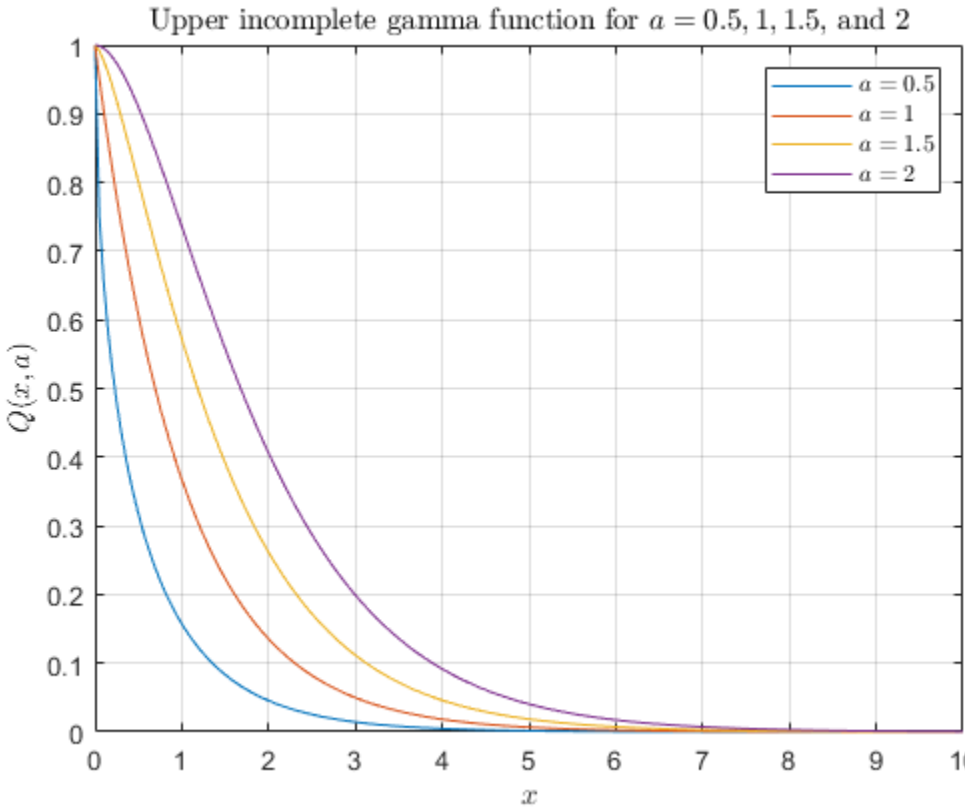
Plot Upper Incomplete Gamma Function

Calculate the upper incomplete gamma function for $a = 0.5$, 1, 1.5, and 2 within the interval $0 \le x \le 10$. Loop over values of a, evaluate the function at each one, and assign each result to a column of Y.

```
A = [0.5 1 1.5 2];
X = 0:0.05:10;
Y = zeros(201,4);
for i = 1:4
    Y(:,i) = gammainc(X,A(i),'upper');
end
```

Plot all of the functions in the same figure.

```
plot(X,Y)
grid on
legend('$a = 0.5$','$a = 1$','$a = 1.5$','$a = 2$','interpreter','latex');
title('Upper incomplete gamma function for $a = 0.5, 1, 1.5,$ and $2$','interpreter','l
xlabel('$x$','interpreter','latex')
ylabel('$Q(x,a)$','interpreter','latex')
```



Plot Scaled Incomplete Gamma Function

Calculate the unscaled lower incomplete gamma function and compare it to the scaled function.

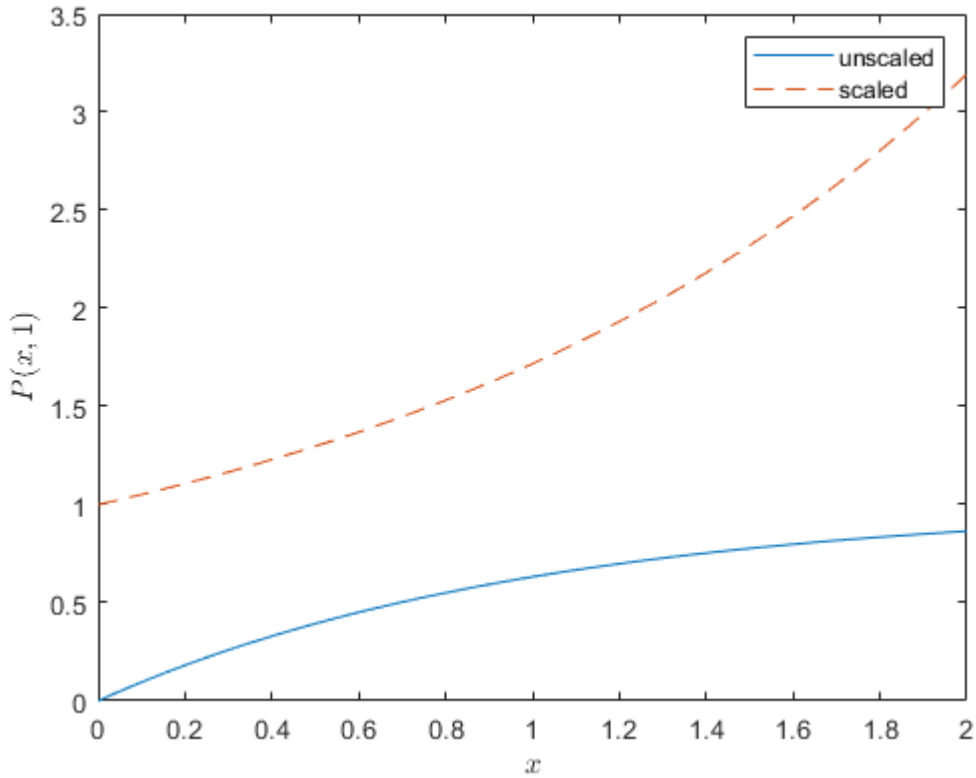
Calculate the unscaled lower incomplete gamma function for $a = 1$ within the interval $0 \le x \le 2$. Plot the function.

```
a = 1;
x = 0:0.001:2;
Y = gammainc(x,a);
```

```
plot(x,Y);
xlabel('$x$','interpreter','latex');
ylabel('$P(x,1)$','interpreter','latex')
hold on
```

Next, calculate the scaled lower incomplete gamma function. Plot the function on the same graph. The scaled function has different asymptotic behavior near 0, which avoids underflow when *x* is close to 0.

```
Ys = gammainc(x,a,'scaledlower');
plot(x,Ys,'--');
legend('unscaled','scaled')
```



Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. The elements of X must be real. X and A must be the same size, or else one of them must be a scalar.

Data Types: `single` | `double`

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. The elements of A must be real and nonnegative. X and A must be the same size, or else one of them must be a scalar.

Data Types: `single` | `double`

type — Type of incomplete gamma function
`'lower'` (default) | `'upper'`

Type of incomplete gamma function, specified as `'lower'` or `'upper'`. If `type` is `'lower'`, then `gammainc` returns the lower incomplete gamma function on page 1-4784. If `type` is `'upper'`, then `gammainc` returns the upper incomplete gamma function on page 1-4784.

scale — Scaling option
`'scaledlower'` | `'scaledupper'`

Scaling option, specified as `'scaledlower'` or `'scaledupper'`. If `scale` is `'scaledlower'` or `'scaledupper'`, then `gammainc` scales the lower or upper incomplete gamma function by a factor of $\Gamma(a + 1)e^x/x^a$, where $\Gamma(a)$ is the `gamma` function. This scaling cancels out the asymptotic behavior of the function near 0, which avoids underflow with small arguments.

Limitations

- When x is negative, the incomplete gamma function can be inaccurate for abs(x) > a +1.

More About

Incomplete Gamma Function

The lower incomplete gamma function P and the upper incomplete gamma function Q are defined by

$$P(x,a) = \frac{1}{\Gamma(a)}\int_0^x t^{a-1}e^{-t}dt,$$

$$Q(x,a) = \frac{1}{\Gamma(a)}\int_x^\infty t^{a-1}e^{-t}dt\,.$$

The gamma function $\Gamma(a)$ is defined by

$$\Gamma(a) = \int_0^\infty t^{a-1}e^{-t}dt\,.$$

MATLAB uses the normalized definition of the incomplete gamma function, where $P(x,a) + Q(x,a) = 1$.

The scaled lower and upper incomplete gamma function are defined by

$$P_s(x,a) = \frac{\Gamma(a+1)}{\Gamma(a)}\frac{e^x}{x^a}\int_0^x t^{a-1}e^{-t}dt,$$

$$Q_s(x,a) = \frac{\Gamma(a+1)}{\Gamma(a)}\frac{e^x}{x^a}\int_x^\infty t^{a-1}e^{-t}dt\,.$$

Some properties of the lower incomplete gamma function are:

- $\lim_{x\to\infty} P(x,a) = 1 \quad \text{for } a \geq 0$
- $\lim_{x,a\to 0} P(x,a) = 1$

Tips

- When the upper incomplete gamma function is close to 0, specifying the `'upper'` option to calculate the function is more accurate than subtracting the lower incomplete gamma function from 1.

References

[1] Olver, F. W. J., A. B. Olde Daalhuis, D. W. Lozier, B. I. Schneider, R. F. Boisvert, C. W. Clark, B. R. Miller, and B. V. Saunders, eds., Chapter 8. Incomplete Gamma and Related Functions, *NIST Digital Library of Mathematical Functions*, Release 1.0.22, Mar. 15, 2018.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Output is always complex.
- Strict single-precision calculations are not supported. In the generated code, single-precision inputs produce single-precision outputs. However, variables inside the function might be double-precision.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The elements of `X` must be nonnegative.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`gamma` | `gammaincinv` | `gammaln` | `psi`

Introduced before R2006a

gammaincinv

Inverse incomplete gamma function

Syntax

```
X = gammaincinv(Y,A)
X = gammaincinv(Y,A,type)
```

Description

`X = gammaincinv(Y,A)` returns the inverse of the lower incomplete gamma function on page 1-4791 evaluated at the elements of `Y` and `A`, such that `Y = gammainc(X,A)`. Both `Y` and `A` must be real. The elements of `Y` must be in the closed interval `[0,1]` and `A` must be nonnegative.

`X = gammaincinv(Y,A,type)` returns the inverse of the lower or upper incomplete gamma function. The choices for `type` are `'lower'` (the default) and `'upper'`.

Examples

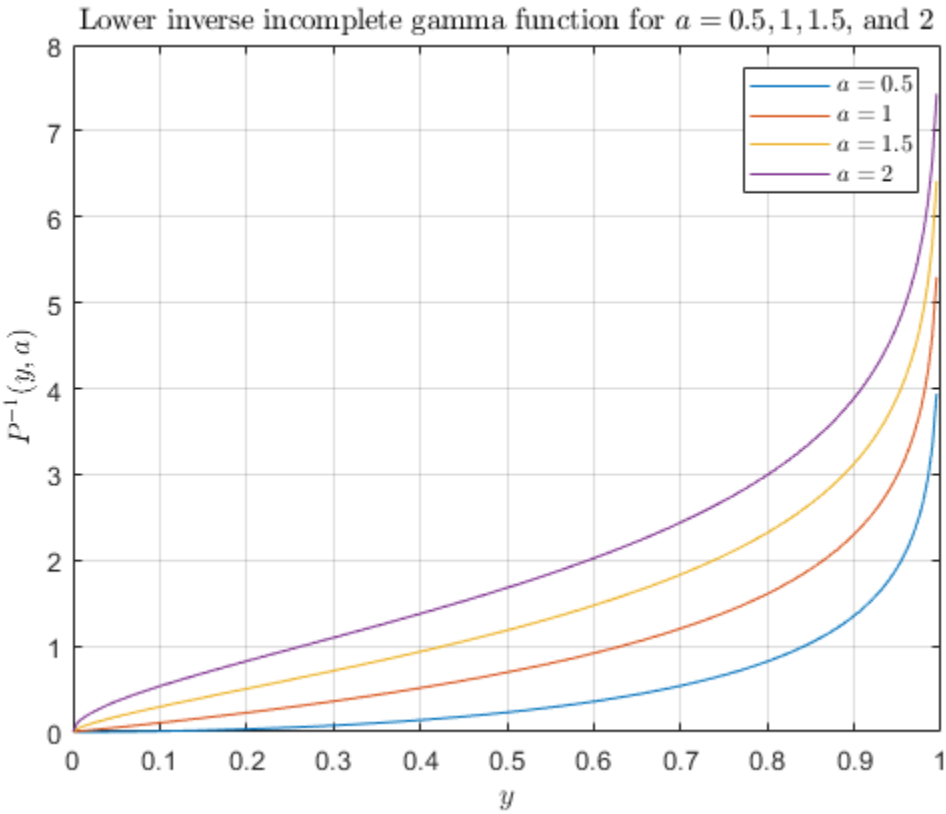
Plot Inverse of Lower Incomplete Gamma Function

Calculate the inverse of the lower incomplete gamma function for a = 0.5, 1, 1.5, and 2 within the interval $0 \le y \le 1$. Loop over values of a, evaluate the inverse function at each one, and assign each result to a column of `X`.

```
A = [0.5 1 1.5 2];
Y = 0:0.005:1;
X = zeros(201,4);
for i = 1:4
    X(:,i) = gammaincinv(Y,A(i));
end
```

Plot all of the inverse functions in the same figure.

```
plot(Y,X)
grid on
legend('$a = 0.5$','$a = 1$','$a = 1.5$','$a = 2$','interpreter','latex')
title('Lower inverse incomplete gamma function for $a = 0.5, 1, 1.5,$ and $2$','interpr
xlabel('$y$','interpreter','latex')
ylabel('$P^{-1}(y,a)$','interpreter','latex')
```



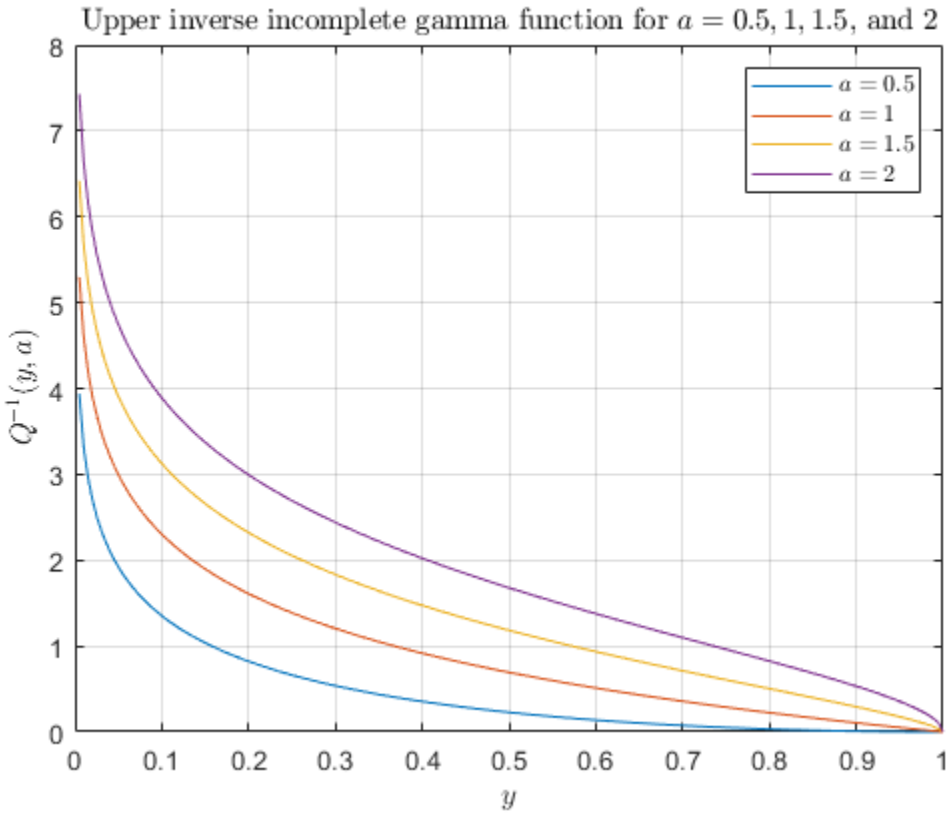
Plot Inverse of Upper Incomplete Gamma Function

Calculate the inverse of the upper incomplete gamma function for $a = 0.5$, 1, 1.5, and 2 within the interval $0 \le y \le 1$. Loop over values of a, evaluate the inverse function at each one, and assign each result to a column of X.

```
A = [0.5 1 1.5 2];
Y = 0:0.005:1;
X = zeros(201,4);
for i = 1:4
    X(:,i) = gammaincinv(Y,A(i),'upper');
end
```

Plot all of the inverse functions in the same figure.

```
plot(Y,X)
grid on
legend('$a = 0.5$','$a = 1$','$a = 1.5$','$a = 2$','interpreter','latex')
title('Upper inverse incomplete gamma function for $a = 0.5, 1, 1.5,$ and $2$','interpr
xlabel('$y$','interpreter','latex')
ylabel('$Q^{-1}(y,a)$','Interpreter','latex')
```



Input Arguments

Y — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. The elements of `Y` must be real and within the closed interval `[0,1]`. `Y` and `A` must be the same size, or else one of them must be a scalar.

Data Types: `single` | `double`

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. The elements of A must be real and nonnegative. Y and A must be the same size, or else one of them must be a scalar.

Data Types: `single` | `double`

type — Type of inverse incomplete gamma function
`'lower'` (default) | `'upper'`

Type of inverse incomplete gamma function, specified as `'lower'` or `'upper'`. If `type` is `'lower'`, then `gammainc` returns the inverse of the lower incomplete gamma function on page 1-4791. If `type` is `'upper'`, then `gammainc` returns the inverse of the upper incomplete gamma function on page 1-4791.

More About

Inverse of Incomplete Gamma Function

The inverse of the lower incomplete gamma function is defined as $x = P^{-1}(y, a)$, such that

$$y = P(x, a) = \frac{1}{\Gamma(a)} \int_0^x t^{a-1} e^{-t} dt .$$

The inverse of the upper incomplete gamma function is defined as $x = Q^{-1}(y, a)$, such that

$$y = Q(x, a) = \frac{1}{\Gamma(a)} \int_x^\infty t^{a-1} e^{-t} dt .$$

The $\Gamma(a)$ term is the gamma function

$$\Gamma(a) = \int_0^\infty t^{a-1} e^{-t} dt .$$

MATLAB uses the normalized definition of the incomplete gamma function, where $P(x, a) + Q(x, a) = 1$.

Some properties of the inverse of the lower incomplete gamma function are:

- $\lim_{y \to 1} P^{-1}(y, a) = \infty \quad \text{for } a > 0$
- $\lim_{\substack{y \to 1 \\ a \to 0}} P^{-1}(y, a) = 0$

Tips

- When the upper incomplete gamma function is close to 0, specifying the `'upper'` option to calculate the upper inverse function is more accurate than subtracting the lower incomplete gamma function from 1 and then taking the lower inverse function.

References

[1] Olver, F. W. J., A. B. Olde Daalhuis, D. W. Lozier, B. I. Schneider, R. F. Boisvert, C. W. Clark, B. R. Miller, and B. V. Saunders, eds., Chapter 8. Incomplete Gamma and Related Functions, *NIST Digital Library of Mathematical Functions*, Release 1.0.22, Mar. 15, 2018.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Output is always complex.

- Strict single-precision calculations are not supported. In the generated code, single-precision inputs produce single-precision outputs. However, variables inside the function might be double-precision.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`gamma` | `gammainc` | `gammaln` | `psi`

gammaln

Logarithm of gamma function

Syntax

```
Y = gammaln(A)
```

Description

`Y = gammaln(A)` returns the logarithm of the `gamma` function, `gammaln(A) = log(gamma(A))`. Input `A` must be nonnegative and real. The `gammaln` command avoids the underflow and overflow that may occur if it is computed directly using `log(gamma(A))`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`gamma` | `gammainc` | `gammaincinv` | `psi`

Introduced before R2006a

gather

Collect tall array into memory after executing queued operations

Syntax

```
Y = gather(X)
[Y1,Y2,Y3,...] = gather(X1,X2,X3,...)
```

Description

`Y = gather(X)` executes all queued operations required to calculate unevaluated tall array `X` and collects the results into memory as `Y`.

MATLAB can run out of memory if the result of the `gather` calculation is too large. If you are unsure whether the result can fit in memory, use `gather(head(X))` or `gather(tail(X))` to perform the full calculation, but bring only a small portion of the result into memory.

Use `gather` sparingly to ensure that extra passes through the data are combined during the calculations whenever possible. For more information, see “Deferred Evaluation of Tall Arrays”.

`[Y1,Y2,Y3,...] = gather(X1,X2,X3,...)` gathers multiple unevaluated tall arrays `X1, X2, X3,...` into the corresponding outputs `Y1, Y2, Y3,...`.

Examples

Evaluate Deferred Tall Array Calculation

Create a datastore for the `airlinesmall.csv` data set. Select a subset of variables to work with, and treat `'NA'` values as missing data so that `datastore` replaces them with `NaN` values. Convert the datastore into a tall table.

```
varnames = {'Year','ArrDelay','UniqueCarrier'};
ds = datastore('airlinesmall.csv', 'TreatAsMissing', 'NA',...
```

```
    'SelectedVariableNames',varnames);
T = tall(ds)
```

```
T =

  Mx3 tall table

    Year    ArrDelay    UniqueCarrier
    ____    ________    _____________

    1987        8          {'PS'}
    1987        8          {'PS'}
    1987       21          {'PS'}
    1987       13          {'PS'}
    1987        4          {'PS'}
    1987       59          {'PS'}
    1987        3          {'PS'}
    1987       11          {'PS'}
     :          :              :
     :          :              :
```

Calculate the size of the tall table.

```
sz = size(T)
```

```
sz =

  1x2 tall double row vector

    ?    ?
```

MATLAB® does not immediately evaluate most operations on tall arrays. Instead, MATLAB remembers the operations you perform as you enter them and optimizes the calculations in the background.

When you use `gather` on an unevaluated tall array, MATLAB executes all of the queued operations using the minimum number of passes through the data. This optimization greatly reduces the execution time of large calculations. For this reason, you should use `gather` only when you need to see a result.

Use `gather` to execute the calculation and collect the result into memory.

```
S = gather(sz)
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 0.96 sec
Evaluation completed in 1.1 sec

S = 1×2

      123523           3
```

Evaluate Multiple Tall Arrays

Use `gather` with several inputs to simultaneously evaluate several tall arrays.

Create a tall array from an in-memory array of random integers between 1 and 1000. Calculate the maximum and minimum values in each column.

```
A = tall(randi(1000,100,7))
```

```
A =

  100x7 tall double matrix

   815   163   645    60   423   583   851
   906   795   379   682    95   541   561
   127   312   812    43   599   870   930
   914   529   533    72   471   265   697
   633   166   351   522   696   319   583
    98   602   940    97   700   120   816
   279   263   876   819   639   940   880
   547   655   551   818    34   646   989
     :     :     :     :     :     :     :
     :     :     :     :     :     :     :
```

```
b = min(A);
c = max(A);
```

Use the results to determine the overall minimum and maximum values in the array. Collect the final result into memory.

```
[mnA,mxA] = gather(min(b),max(c));
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 0.39 sec
Evaluation completed in 0.91 sec
```

```
valRange = [mnA mxA]
```

```
valRange = 1×2

           1        1000
```

Input Arguments

X — Unevaluated tall array
tall array

Unevaluated tall array. An unevaluated tall array is any tall array on which you perform calculations without using `gather` to fully evaluate those calculations.

Output Arguments

Y — In-memory array
array

In-memory array. The data type of `Y` is the same as the underlying data type of the unevaluated tall array `X`.

Tips

- If you have Parallel Computing Toolbox, see `gather` for information about gathering `distributed` and `gpuArray` computations.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `gather` in the Parallel Computing Toolbox documentation.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `gather` in the Parallel Computing Toolbox documentation.

See Also

`head` | `tail` | `tall` | `topkrows`

Topics

"Tall Arrays for Out-of-Memory Data"
"Deferred Evaluation of Tall Arrays"

Introduced in R2016b

gca

Current axes or chart

Syntax

```
ax = gca
```

Description

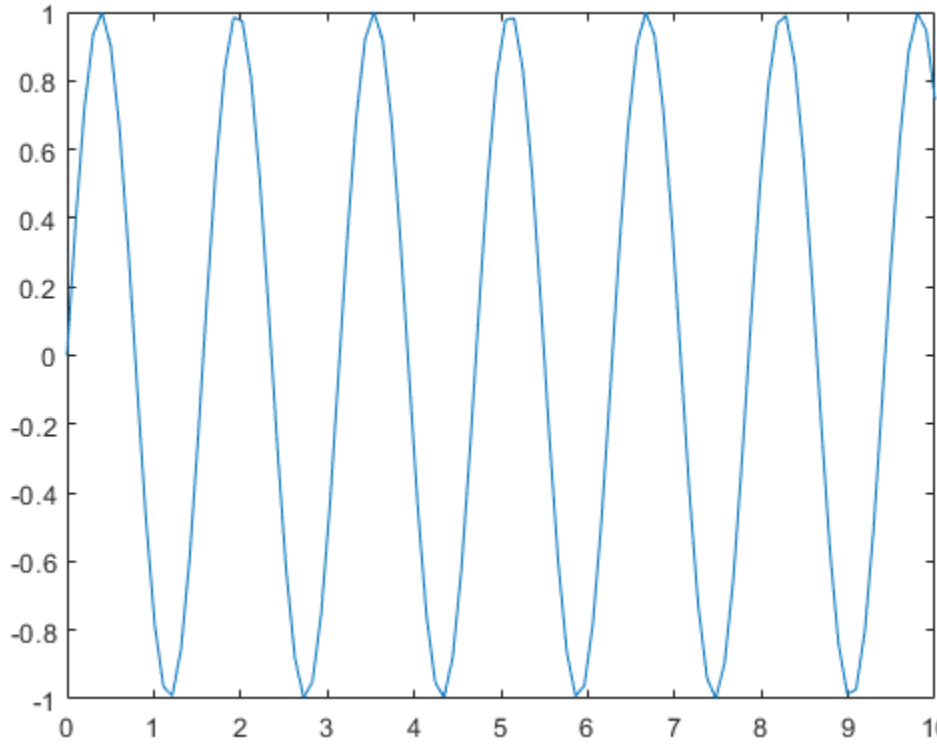
`ax = gca` returns the current axes or chart for the current figure, which is typically the last one created or clicked with the mouse. Graphics functions, such as `title`, target the current axes or chart. Use `ax` to access and modify properties of the axes or chart. If axes or charts do not exist, then `gca` creates Cartesian axes.

Examples

Specify Properties for Current Axes

Plot a sine wave.

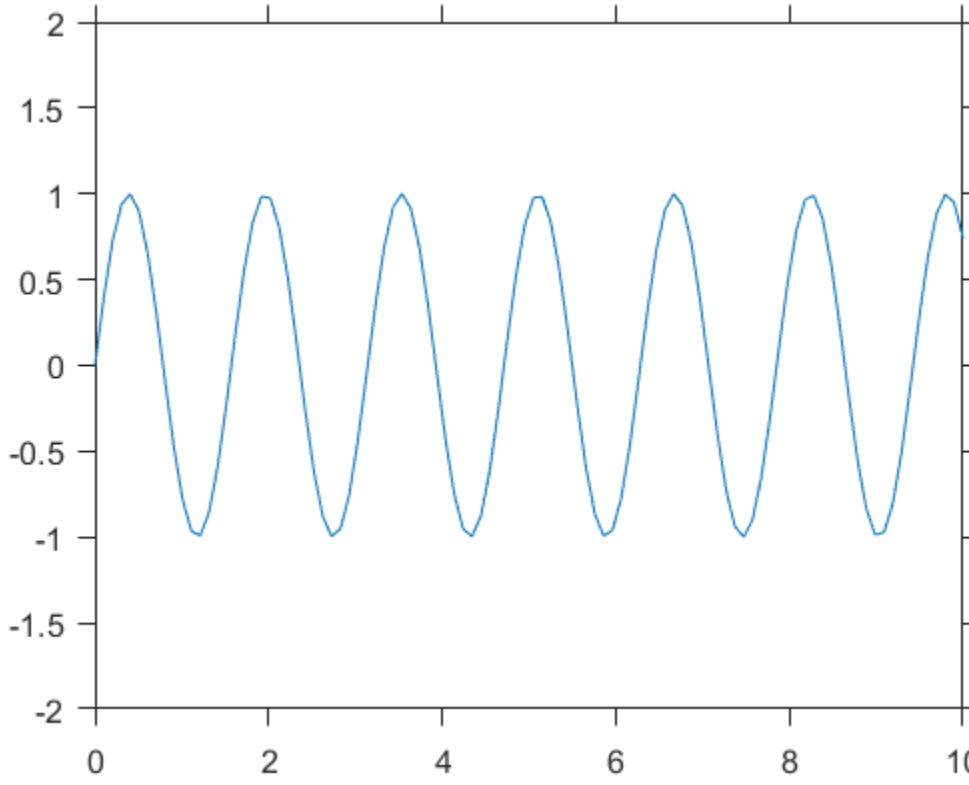
```
x = linspace(0,10);
y = sin(4*x);
plot(x,y)
```



Set the font size, tick direction, tick length, and *y*-axis limits for the current axes. Use `gca` to refer to the current axes.

Note: Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead, such as `set(ax,'FontSize',12)`.

```
ax = gca; % current axes
ax.FontSize = 12;
ax.TickDir = 'out';
ax.TickLength = [0.02 0.02];
ax.YLim = [-2 2];
```



Output Arguments

ax — Current axes or chart

`Axes` object | `PolarAxes` object | `GeographicAxes` object | graphics object

Current axes or chart, returned as an `Axes` object, a `PolarAxes` object, a `GeographicAxes` object, or a graphics object whose parent is a `Figure`, `Tab`, or `Panel` object, instead of an `Axes` object.

For example, a `HeatmapChart` object can be the current chart since the parent is typically a `Figure` object. A `Stem` object cannot be the current chart since the parent is an `Axes` object.

Tips

- User interaction can change the current axes or chart. It is better to assign the axes or chart to a variable when you create it instead of relying on `gca`.
- Changing the current figure also changes the current axes or chart.
- Set axes properties after plotting since some plotting functions reset axes properties.
- To access the current axes or chart without forcing the creation of Cartesian axes, query the figure `CurrentAxes` property. MATLAB returns an empty array if there is no current axes.

  ```
  fig = gcf;
  ax = fig.CurrentAxes;
  ```

  Starting in R2014b, you can query properties using dot notation. If you are using an earlier release, use the `get` function instead, such as `ax = get(fig,'CurrentAxes')`.

See Also

Functions
`axes` | `cla` | `findobj` | `gcf` | `get` | `set` | `shg`

Properties
Axes | Polar Axes

Introduced before R2006a

gcbf

Handle of figure containing object whose callback is executing

Syntax

```
fig = gcbf
```

Description

`fig = gcbf` returns the handle of the figure that contains the object whose callback is currently executing. This object can be the figure itself, in which case, `gcbf` returns the figure's handle.

When no callback is executing, `gcbf` returns the empty matrix, `[]`.

The value returned by `gcbf` is identical to the `figure` output argument returned by `gcbo`.

See Also

`gca` | `gcbo` | `gcf` | `gco`

Introduced before R2006a

gcbo

Handle of object whose callback is executing

Syntax

```
h = gcbo
[h,figure] = gcbo
```

Description

`h = gcbo` returns the handle of the graphics object whose callback is executing.

`[h,figure] = gcbo` returns the handle of the current callback object and the handle of the figure containing this object.

Tips

The MATLAB software stores the handle of the object whose callback is executing in the root `CallbackObject` property. If a callback interrupts another callback, MATLAB replaces the `CallbackObject` value with the handle of the object whose callback is interrupting. When that callback completes, MATLAB restores the handle of the object whose callback was interrupted.

The root `CallbackObject` property is read only, so its value is always valid at any time during callback execution. The root `CurrentFigure` property, and the figure `CurrentAxes` and `CurrentObject` properties (returned by `gcf`, `gca`, and `gco`, respectively) are user settable, so they can change during the execution of a callback, especially if that callback is interrupted by another callback. Therefore, those functions are not reliable indicators of which object's callback is executing.

When you write callback routines for the `CreateFcn` and `DeleteFcn` of any object and the figure `SizeChangedFcn`, you must use `gcbo` since those callbacks do not update the root's `CurrentFigure` property, or the figure's `CurrentObject` or `CurrentAxes` properties; they only update the root's `CallbackObject` property.

When no callbacks are executing, `gcbo` returns `[]` (an empty matrix).

See Also

`gca` | `gcf` | `gco` | `groot`

Introduced before R2006a

gcd

Greatest common divisor

Syntax

```
G = gcd(A,B)
[G,U,V] = gcd(A,B)
```

Description

`G = gcd(A,B)` returns the greatest common divisors of the elements of `A` and `B`. The elements in `G` are always nonnegative, and `gcd(0,0)` returns `0`. This syntax supports inputs of any numeric type.

`[G,U,V] = gcd(A,B)` also returns the Bézout coefficients, `U` and `V`, which satisfy: `A.*U + B.*V = G`. The Bézout coefficients are useful for solving Diophantine equations. This syntax supports double, single, and signed integer inputs.

Examples

Greatest Common Divisors of Double Values

```
A = [-5 17; 10 0];
B = [-15 3; 100 0];
G = gcd(A,B)
```

```
G = 2×2

     5     1
    10     0
```

`gcd` returns positive values, even when the inputs are negative.

Greatest Common Divisors of Unsigned Integers

```
A = uint16([255 511 15]);
B = uint16([15 127 1023]);
G = gcd(A,B)
```

```
G = 1x3 uint16 row vector

   15    1    3
```

Solution to Diophantine Equation

Solve the Diophantine equation, $30x + 56y = 8$ for x and y.

Find the greatest common divisor and a pair of Bézout coefficients for `30` and `56`.

```
[g,u,v] = gcd(30,56)
```

```
g = 2

u = -13

v = 7
```

`u` and `v` satisfy the Bézout's identity, `(30*u) + (56*v) = g`.

Rewrite Bézout's identity so that it looks more like the original equation. Do this by multiplying by `4`. Use `==` to verify that both sides of the equation are equal.

```
(30*u*4) + (56*v*4) == g*4
```

```
ans = logical
   1
```

Calculate the values of x and y that solve the problem.

```
x = u*4
```

```
x = -52
```

```
y = v*4
```

```
y = 28
```

Input Arguments

A,B — Input values
scalars, vectors, or arrays of real integer values

Input values, specified as scalars, vectors, or arrays of real integer values. A and B can be any numeric type, and they can be of different types within certain limitations:

- If A or B is of type `single`, then the other can be of type `single` or `double`.
- If A or B belongs to an integer class, then the other must belong to the same class or it must be a `double` scalar value.

A and B must be the same size or one must be a scalar.

Example: `[20 -3 13],[10 6 7]`

Example: `int16([100 -30 200]),int16([20 15 9])`

Example: `int16([100 -30 200]),20`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

G — Greatest common divisor
real, nonnegative integer values

Greatest common divisor, returned as an array of real nonnegative integer values. G is the same size as A and B, and the values in G are always real and nonnegative. G is returned as the same type as A and B. If A and B are of different types, then G is returned as the nondouble type.

U,V — Bézout coefficients
real integer values

Bézout coefficients, returned as arrays of real integer values that satisfy the equation, `A.*U + B.*V = G`. The data type of `U` and `V` is the same type as that of `A` and `B`. If `A` and `B` are of different types, then `U` and `V` are returned as the nondouble type.

Algorithms

`g = gcd(A,B)` is calculated using the Euclidian algorithm.[1]

`[g,u,v] = gcd(A,B)` is calculated using the extended Euclidian algorithm.[1]

References

[1] Knuth, D. "Algorithms A and X." *The Art of Computer Programming, Vol. 2*, Section 4.5.2. Reading, MA: Addison-Wesley, 1973.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`lcm`

Introduced before R2006a

gcf

Current figure handle

Syntax

```
fig = gcf
```

Description

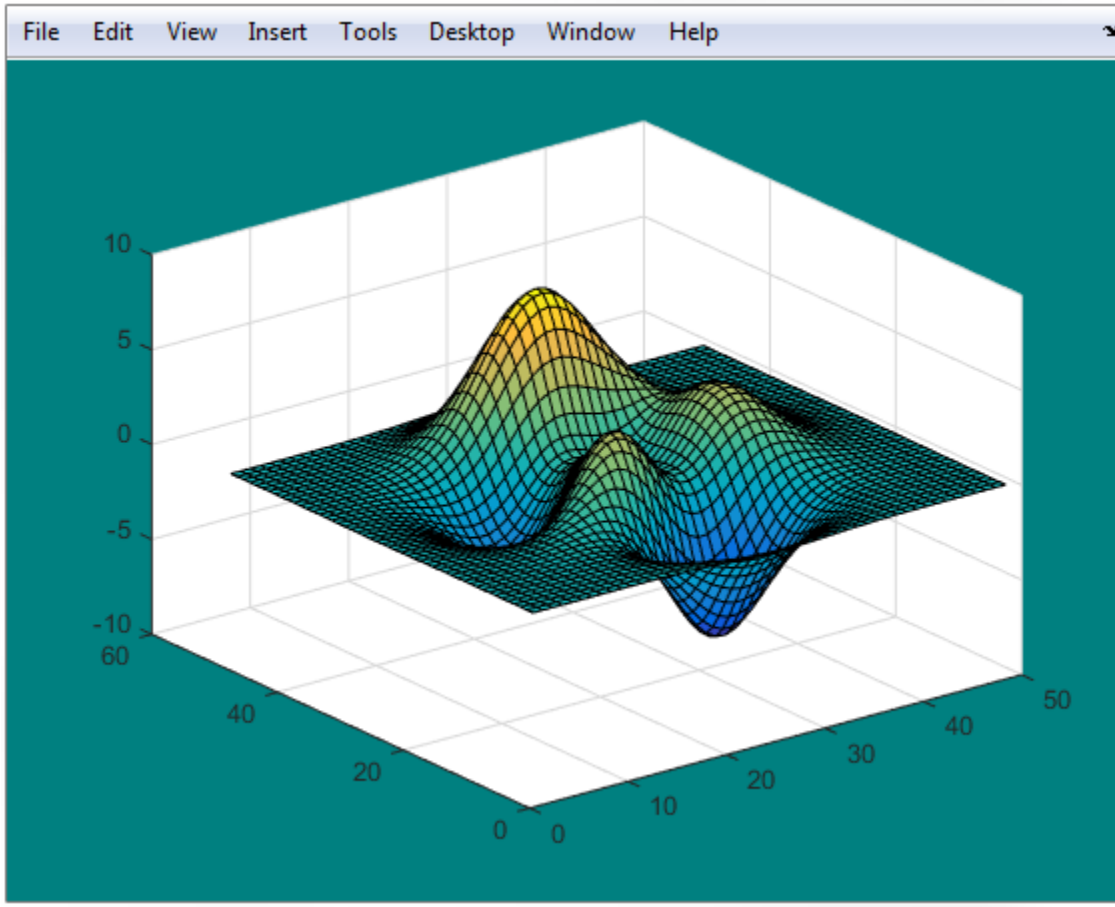
`fig = gcf` returns the current figure on page 1-4814 handle. If a figure does not exist, then `gcf` creates a figure and returns its handle. You can use the figure handle to query and modify figure properties. For more information, see Figure.

Examples

Specify Properties for Current Figure

Set the background color and remove the toolbar for the current figure. Use the `gcf` command to get the current figure handle.

```
surf(peaks)
fig = gcf; % current figure handle
fig.Color = [0 0.5 0.5];
fig.ToolBar = 'none';
```



More About

Current Figure

The current figure is the target for graphics output. It is the figure window in which graphics commands such as `plot`, `title`, and `surf` draw their results. It is typically the last figure created or the last figure clicked with the mouse.

User interaction can change the current figure. If you need to access a specific figure, store the figure handle in your program code when you create the figure and use this handle instead of `gcf`.

Tips

- To get the handle of the current figure without forcing the creation of a figure if one does not exist, query the `CurrentFigure` property on the root object.

  ```
  fig = get(groot,'CurrentFigure');
  ```

  MATLAB returns `fig` as an empty array if there is no current figure.

See Also

Functions
`ancestor` | `clf` | `figure` | `gca` | `get` | `set` | `shg`

Properties
Figure

Introduced before R2006a

gcmr

Get current mapreducer configuration

Syntax

```
mr = gcmr
mr = gcmr('nocreate')
```

Description

`mr = gcmr` returns an object representing the current global execution environment for `mapreduce`.

- If no global execution environment exists, then `gcmr` calls `mapreducer` to set the global execution environment to be the default.
- If a global execution environment currently exists, then `gcmr` returns the last visible MapReducer object created.

  When you create a MapReducer object using `mapreducer`, the object sets the global execution environment. The global execution environment persists even if the object representing it is later deleted.
- If the global execution environment is deleted or invalid, then `gcmr` returns the next visible MapReducer object available. For example, `delete(gcmr)` deletes the current global execution environment.

The `gcmr` function defines the global execution environment for `mapreduce`, and is most likely used with Parallel Computing Toolbox, MATLAB Parallel Server, or MATLAB Compiler.

If you do not specify a configuration to use in your call to `mapreduce`, then by default `mapreduce` uses the configuration returned (or generated) by `gcmr`.

`mr = gcmr('nocreate')` returns the current global execution environment for `mapreduce`, if one already exists. If no global execution environment exists, then `gcmr` returns `[]`.

Tips

- If you have Parallel Computing Toolbox, see the `mapreducer` function reference page for related information.
- If you have MATLAB Compiler, see the `mapreducer` function reference page for related information.

See Also

`mapreduce` | `mapreducer`

Topics

"Speed Up and Deploy MapReduce Using Other Products"

Introduced in R2014b

gco

Handle of current object

Syntax

```
h = gco
h = gco(figure_handle)
```

Description

`h = gco` returns the handle of the current object.

`h = gco(figure_handle)` returns the handle of the current object in the figure specified by `figure_handle`.

Tips

The current object is the last object clicked or selected via keyboard interaction, excluding `uimenu`s. If the mouse click did not occur over a figure child object, the figure becomes the current object. The MATLAB software stores the handle of the current object in the figure's `CurrentObject` property.

An object can become the current object as a result of pressing the space bar to invoke a callback in a dialog when a uicontrol in that dialog has focus (usually the result of using the **Tab** key to change focus).

The `CurrentObject` of the `CurrentFigure` does not always indicate the object whose callback is being executed. Interruptions of callbacks by other callbacks can change the `CurrentObject` or even the `CurrentFigure`. Some callbacks, such as `CreateFcn` and `DeleteFcn`, and uimenu `Callback`, intentionally do not update `CurrentFigure` or `CurrentObject`.

`gcbo` provides the only completely reliable way to retrieve the handle to the object whose callback is executing, at any point in the `callback` function, regardless of the type of callback or of any previous interruptions.

See Also

gca | gcbo | gcf

Introduced before R2006a

ge, >=

Determine greater than or equal to

Syntax

```
A >= B
ge(A,B)
```

Description

`A >= B` returns a logical array with elements set to logical 1 (`true`) where `A` is greater than or equal to `B`; otherwise, the element is logical `0` (`false`). The test compares only the real part of numeric arrays. `ge` returns logical `0` (`false`) where `A` or `B` have NaN or undefined categorical elements.

`ge(A,B)` is an alternate way to execute `A >= B`, but is rarely used. It enables operator overloading for classes.

Examples

Test Vector Elements

Find which vector elements are greater than or equal to a given value.

Create a numeric vector.

```
A = [1 12 18 7 9 11 2 15];
```

Test the vector for elements that are greater than or equal to `11`.

```
A >= 11
```

```
ans = 1x8 logical array
```

```
   0   1   1   0   0   1   0   1
```

The result is a vector with values of logical `1` (`true`) where the elements of `A` satisfy the expression.

Use the vector of logical values as an index to view the values in `A` that are greater than or equal to `11`.

```
A(A >= 11)
```

```
ans = 1×4

    12    18    11    15
```

The result is a subset of the elements in `A`.

Replace Elements of Matrix

Create a matrix.

```
A = magic(4)
```

```
A = 4×4

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Replace all values greater than or equal to `9` with the value `10`.

```
A(A >= 9) = 10
```

```
A = 4×4

    10     2     3    10
     5    10    10     8
    10     7     6    10
     4    10    10     1
```

The result is a new matrix whose largest element is `10`.

Compare Values in Categorical Array

Create an ordinal categorical array.

```
A = categorical({'large' 'medium' 'small'; 'medium' ...
'small' 'large'},{'small' 'medium' 'large'},'Ordinal',1)
```

```
A = 2x3 categorical array
     large       medium      small
     medium      small       large
```

The array has three categories: `'small'`, `'medium'`, and `'large'`.

Find all values greater than or equal to the category `'medium'`.

```
A >= 'medium'
```

```
ans = 2x3 logical array

   1   1   0
   1   0   1
```

A value of logical `1` (`true`) indicates a value greater than or equal to the category `'medium'`.

Compare the rows of `A`.

```
A(1,:) >= A(2,:)
```

```
ans = 1x3 logical array

   1   1   0
```

The function returns logical `1` (`true`) where the first row has a category value greater than or equal to the second row.

Test Complex Numbers

Create a vector of complex numbers.

```
A = [1+i 2-2i 1+3i 1-2i 5-i];
```

Find the values that are greater than or equal to 2.

```
A(A >= 2)
```

```
ans = 1×2 complex

   2.0000 - 2.0000i   5.0000 - 1.0000i
```

`ge` compares only the real part of the elements in `A`.

Use `abs` to find which elements are outside a radius of 2 from the origin.

```
A(abs(A) >= 2)
```

```
ans = 1×4 complex

   2.0000 - 2.0000i   1.0000 + 3.0000i   1.0000 - 2.0000i   5.0000 - 1.0000i
```

The result has more elements since `abs` accounts for the imaginary part of the numbers.

Test Duration Values

Create a `duration` array.

```
d = hours(21:25) + minutes(75)
```

```
d = 1x5 duration array
   22.25 hr   23.25 hr   24.25 hr   25.25 hr   26.25 hr
```

Test the array for elements that are greater than or equal to one standard day.

```
d >= 1
```

```
ans = 1x5 logical array
```

```
0   0   1   1   1
```

Input Arguments

A, B — Operands
scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Numeric or string inputs A and B must either be the same size or have sizes that are compatible (for example, A is an M-by-N matrix and B is a scalar or 1-by-N row vector). For more information, see “Compatible Array Sizes for Basic Operations”.

You can compare numeric inputs of any type, and the comparison does not suffer loss of precision due to type conversion.

If A and B are categorical, datetime, or duration arrays, then they must be the same size unless one is a scalar.

- If one input is an ordinal categorical array, the other input can be an ordinal categorical array, a cell array of character vectors, or a single character vector. A single character vector expands into a cell array of character vectors of the same size as the other input. If both inputs are ordinal categorical arrays, they must have the same sets of categories, including their order. See “Compare Categorical Array Elements” for more details.
- If one input is a datetime array, the other input can be a datetime array, a character vector, or a cell array of character vectors.
- If one input is a duration array, the other input can be a duration array or a numeric array. The operator treats each numeric value as a number of standard 24-hour days.
- If one input is a string array, the other input can be a string array, a character vector, or a cell array of character vectors. The corresponding elements of A and B are compared lexicographically.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `categorical` | `datetime` | `duration`
Complex Number Support: Yes

Tips

- Some floating-point numbers cannot be represented exactly in binary form. This leads to small differences in results that the >= operator reflects. For more information, see "Avoiding Common Problems with Floating-Point Arithmetic".

Compatibility Considerations

Implicit expansion change affects arguments for operators

Behavior changed in R2016b

Starting in R2016b with the addition of implicit expansion, some combinations of arguments for basic operations that previously returned errors now produce results. For example, you previously could not add a row and a column vector, but those operands are now valid for addition. In other words, an expression like `[1 2] + [1; 2]` previously returned a size mismatch error, but now it executes.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a `try/catch` block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see "Compatible Array Sizes for Basic Operations".

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see "Tall Arrays".

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`eq` | `gt` | `le` | `lt` | `ne`

Topics

“Array Comparison with Relational Operators”
“Ordinal Categorical Arrays”
“MATLAB Operators and Special Characters”

Introduced before R2006a

genpath

Generate path name

Syntax

```
p = genpath
p = genpath(folderName)
```

Description

`p = genpath` returns a character vector containing a path name that includes all the folders and subfolders below *`matlabroot`*`/toolbox`, including empty subfolders.

`p = genpath(folderName)` returns a character vector containing a path name that includes `folderName` and multiple levels of subfolders below `folderName`. The path name does not include folders named `private`, folders that begin with the `@` character (class folders), folders that begin with the `+` character (package folders), or subfolders within any of these.

Examples

Add Folder and Subfolders to Search Path

Use `genpath` in conjunction with `addpath` to add a folder and its subfolders to the search path.

Create a folder `myfolder` containing a subfolder `mysubfolder`.

```
mkdir myfolder;
cd myfolder;
mkdir mysubfolder;
cd ..
```

Generate a path that includes `myfolder` and all folders below it.

```
p = genpath('myfolder')
```

```
p =
'myfolder;myfolder\mysubfolder;'
```

Add the folder and its subfolders to the search path.

```
addpath(p)
```

Input Arguments

`folderName` — Folder name
character vector | string scalar

Folder name, specified as a character vector or string scalar.

Example: `'c:/matlab/myfiles'`

Data Types: `char` | `string`

See Also

`addpath` | `path` | `rmpath`

Topics

"What Is the MATLAB Search Path?"

Introduced before R2006a

genvarname

Construct valid variable name from string

Note `genvarname` will be removed in a future release. Use `matlab.lang.makeValidName` and `matlab.lang.makeUniqueStrings` instead.

Syntax

```
varname = genvarname(str)
varname = genvarname(str, exclusions)
```

Description

`varname = genvarname(str)` constructs a string or character vector `varname` that is similar to or the same as the `str` input, and can be used as a valid variable name. `str` can be a string, a string array, a character array, a cell array of character vectors. If `str` is a string array or cell array of character vectors, `genvarname` returns a string array or cell array of character vectors in `varname`. The elements returned by `genvarname` are guaranteed to be different from each other.

`varname = genvarname(str, exclusions)` returns a valid variable name that is different from any name listed in the `exclusions` input. The `exclusions` input can be a string, a string array, a character array, a cell array of character vectors. Specify the function `who` in `exclusions` to create a variable name that will be unique in the current MATLAB workspace (see “Example 4” on page 1-4831, below).

Note `genvarname` does not create a variable in the MATLAB workspace. You cannot, therefore, assign a value to the output of `genvarname`.

Examples

Example 1

Create four similar variable names that do not conflict with each other:

```
v = genvarname({'A', 'A', 'A', 'A'})
v =
    'A'    'A1'    'A2'    'A3'
```

Example 2

Read a column header `hdr` from worksheet `trial2` in Excel spreadsheet myproj_apr23:

```
[data hdr] = xlsread('myproj_apr23.xls', 'trial2');
```

Make a variable name from the text of the column header that will not conflict with other names:

```
v = genvarname(['Column ' hdr{1,3}]);
```

Assign data taken from the spreadsheet to the variable in the MATLAB workspace:

```
eval([v '= data(1:7, 3);']);
```

Example 3

Collect readings from an instrument once every minute over the period of an hour into different fields of a structure. Simulate instrument readings using a random number. `genvarname` not only generates unique fieldnames, but also creates the structure and fields in the MATLAB workspace.

```
for k = 1:60
record.(genvarname(['reading' datestr(clock, 'HHMMSS')])) = rand(1);
pause(60)
end
```

After the program ends, display the recorded data from the workspace:

```
record

record =
```

```
    reading092610: 0.6541
    reading092710: 0.6892
    reading092811: 0.7482
    reading092911: 0.4505
    reading093011: 0.0838
         .
         .
         .
```

Example 4

Generate variable names that are unique in the MATLAB workspace by putting the output from the `who` function in the `exclusions` list.

```
for k = 1:5
   t = clock;
   pause(uint8(rand * 10));
   v = genvarname('time_elapsed', who);
   eval([v ' = etime(clock,t)'])
   end
```

As this code runs, you can see that the variables created by `genvarname` are unique in the workspace:

```
time_elapsed =
    5.0070
time_elapsed1 =
    2.0030
time_elapsed2 =
    7.0010
time_elapsed3 =
    8.0010
time_elapsed4 =
    3.0040
```

After the program completes, use the `who` function to view the workspace variables:

```
who

k           time_elapsed   time_elapsed2  time_elapsed4
t           time_elapsed1  time_elapsed3  v
```

Example 5

If you try to make a variable name from a MATLAB keyword, `genvarname` creates a variable name that capitalizes the keyword and precedes it with the letter `x`:

```
v = genvarname('global')
v =
   xGlobal
```

Example 6

If you enter a character vector that is longer than the value returned by the `namelengthmax` function, `genvarname` truncates the resulting variable name character vector:

```
namelengthmax
ans =
    63

vstr = genvarname(sprintf('%s%s', ...
   'This name truncates because it contains ', ...
   'more than the maximum number of characters'))
vstr =
ThisNameTruncatesBecauseItContainsMoreThanTheMaximumNumberOfCha
```

Tips

A valid MATLAB variable name is a character vector of letters, digits, and underscores, such that the first character is a letter, and the length of the vector is less than or equal to the value returned by the `namelengthmax` function. Any character vector that exceeds `namelengthmax` is truncated in the `varname` output. See “Example 6” on page 1-4832, below.

The variable name returned by `genvarname` is not guaranteed to be different from other variable names currently in the MATLAB workspace unless you use the `exclusions` input in the manner shown in “Example 4” on page 1-4831, below.

If you use `genvarname` to generate a field name for a structure, MATLAB does create a variable for the structure and field in the MATLAB workspace. See “Example 3” on page 1-4830, below.

If the `str` input contains any whitespace characters, `genvarname` removes then and capitalizes the next alphabetic character in `str`. If `str` contains any nonalphanumeric characters, `genvarname` translates these characters into their hexadecimal value.

See Also

`iskeyword` | `isletter` | `isvarname` | `namelengthmax` | `regexp` | `who`

Introduced before R2006a

geoaxes

Create geographic axes

Syntax

```
geoaxes
geoaxes(Name,Value)
geoaxes(parent, ___ )
gx = geoaxes( ___ )
geoaxes(gx)
```

Description

`geoaxes` creates a geographic axes in the current figure using default property values, and makes it the current axes.

A geographic axes displays data in geographic coordinates (latitude/longitude) on a map. The map is *live*, that is, you can pan to view other geographic locations and zoom in and out on the map to view regions in more detail.

`geoaxes(Name,Value)` specifies values for properties of the `GeographicAxes` object using one or more name-value pair arguments.

`geoaxes(parent, ___ )` creates the geographic axes in the figure, panel, or tab specified by `parent`, instead of in the current figure.

`gx = geoaxes( ___ )` returns the `GeographicAxes` object. Use `gx` to modify properties of the axes after you create it.

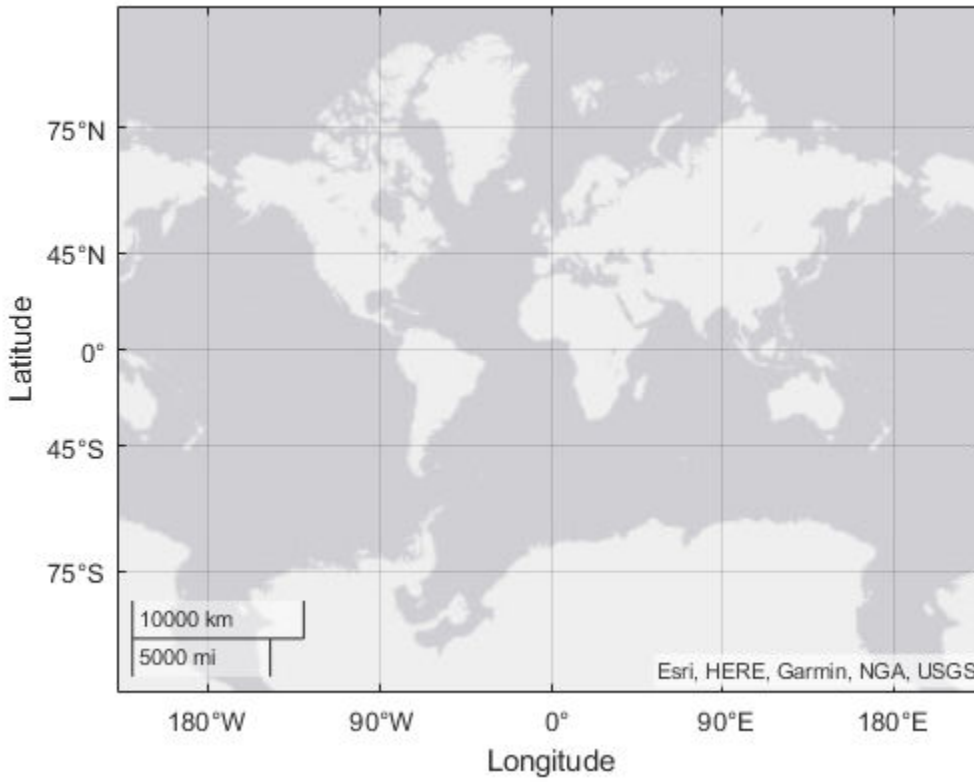
`geoaxes(gx)` makes the `GeographicAxes` object `gx` the current axes.

Examples

Plot Data in Geographic Axes

Create a set of geographic axes.

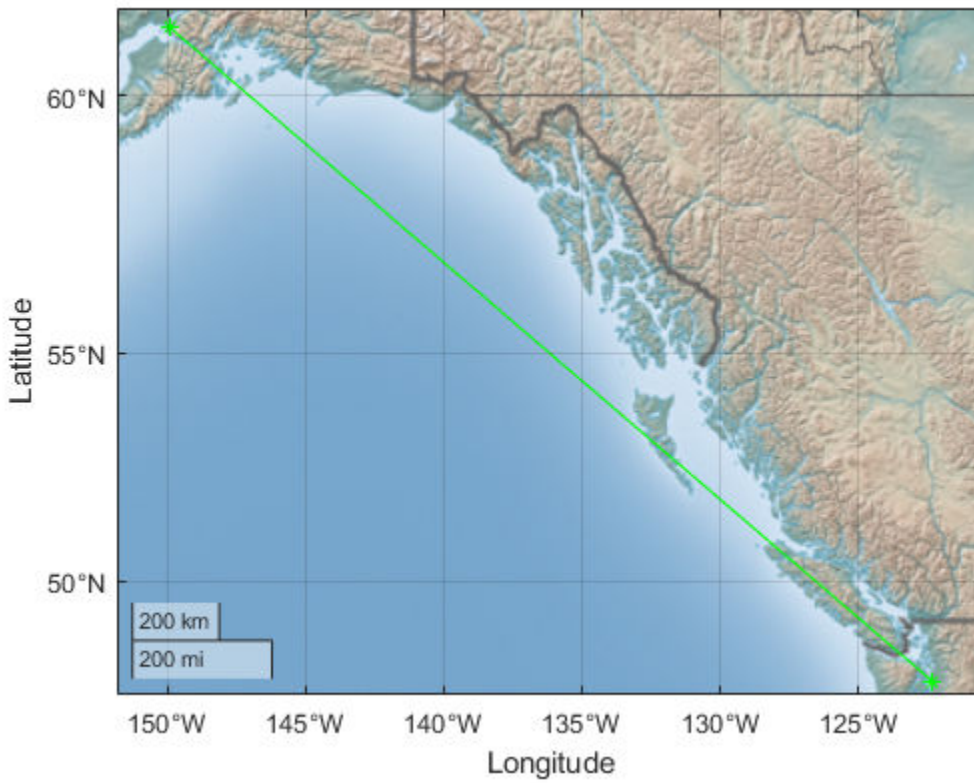
```
gx = geoaxes;
```



Plot data using the `geoplot` function. To modify the geographic axes you created, specify `gx` as an input argument. Customize the appearance of the line using the line specification `'g-*'`. Change the basemap using the `geobasemap` function.

```
latSeattle = 47.62;
lonSeattle = -122.33;
latAnchorage = 61.20;
lonAnchorage = -149.9;
```

```
geoplot(gx,[latSeattle latAnchorage],[lonSeattle lonAnchorage],'g-*')
geobasemap(gx,'colorterrain')
```



Input Arguments

parent — Parent container
`Figure` object | `Panel` object | `Tab` object | `TiledChartLayout` object

Parent container, specified as a `Figure`, `Panel`, `Tab`, or `TiledChartLayout` object.

gx — Geographic axes to be made current axes
`GeographicAxes` object

Geographic axes to be made current axes, specified as a `GeographicAxes` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The properties listed here are only a subset. For a complete list, see GeographicAxes Properties.

Example: `gx = geoaxes('Basemap','colorterrain')`

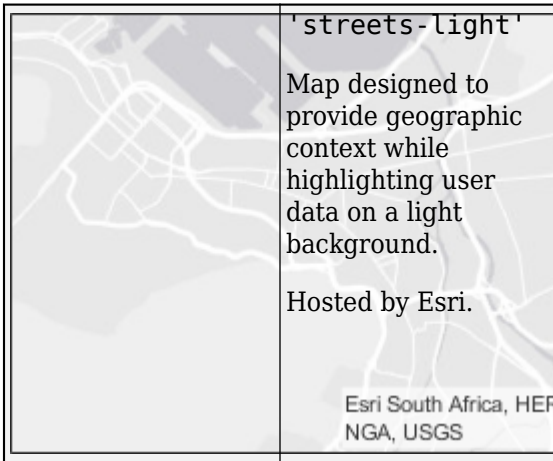
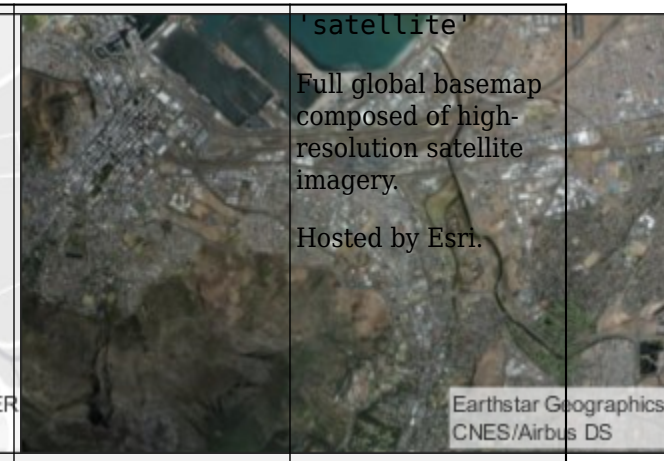
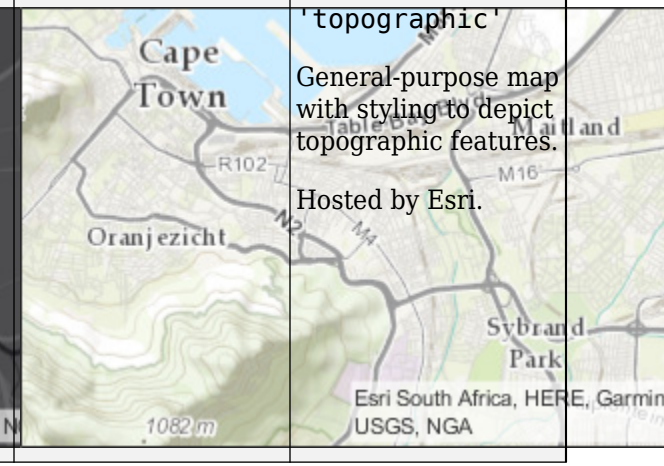
Basemap — Map on which to plot data
`'streets-light'` (default) | `'darkwater'` | `'colorterrain'` | `'satellite'` | ...

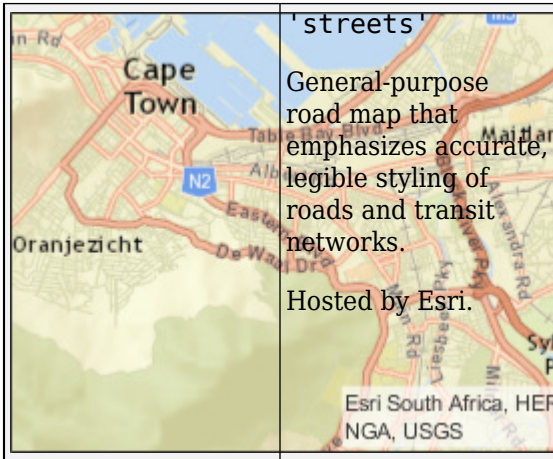
Map on which to plot data, specified as one of the values listed in the table. Six of the basemaps are tiled data sets that MathWorks derives from public-domain data. Five of the basemaps are high-zoom-level maps hosted by Esri®.

MATLAB includes one installed basemap, a two-tone map named `'darkwater'`. Use of this basemap does not require Internet access. Use of the other basemaps, including the default basemap `'streets-light'`, does require Internet access.

If you do not have consistent access to the Internet, you can download the basemaps hosted by MathWorks onto your local system by using the Add-On Explorer. The five high-zoom-level maps are not available for download. For more about downloading basemaps and changing the default basemap on your local system, see “Access Basemaps in MATLAB”.

The five basemaps hosted by Esri include attributions that give credit to the provider of the map data. This attribution can change as you zoom in on a map because different organizations contribute map data at different zoom levels for different localities. Alignment of boundaries and region labels are a presentation of the feature provided by our data vendors and does not imply endorsement by MathWorks.

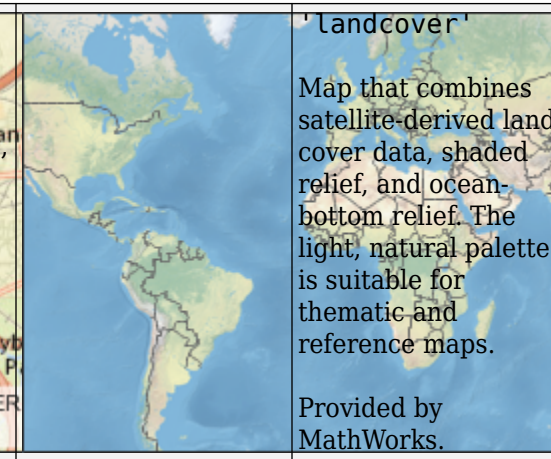
	`'streets-light'` Map designed to provide geographic context while highlighting user data on a light background. Hosted by Esri.		`'satellite'` Full global basemap composed of high-resolution satellite imagery. Hosted by Esri.
	`'streets-dark'` Map designed to provide geographic context while highlighting user data on a dark background. Hosted by Esri.		`'topographic'` General-purpose map with styling to depict topographic features. Hosted by Esri.



`'streets'`

General-purpose road map that emphasizes accurate, legible styling of roads and transit networks.

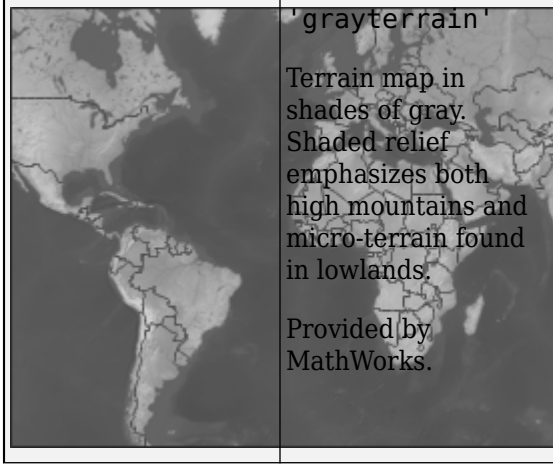
Hosted by Esri.



`'landcover'`

Map that combines satellite-derived land cover data, shaded relief, and ocean-bottom relief. The light, natural palette is suitable for thematic and reference maps.

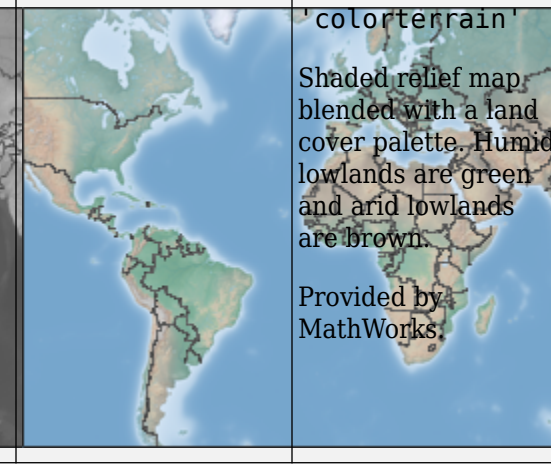
Provided by MathWorks.



`'grayterrain'`

Terrain map in shades of gray. Shaded relief emphasizes both high mountains and micro-terrain found in lowlands.

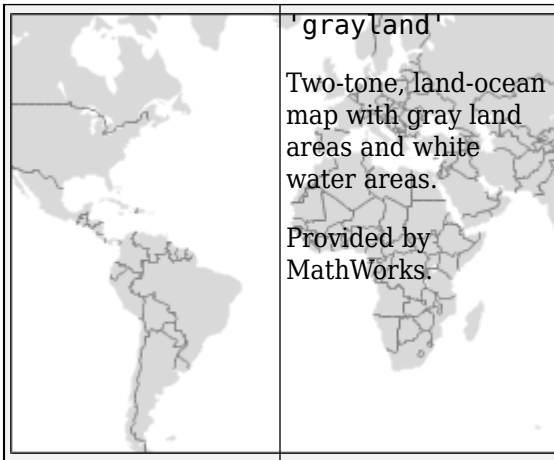
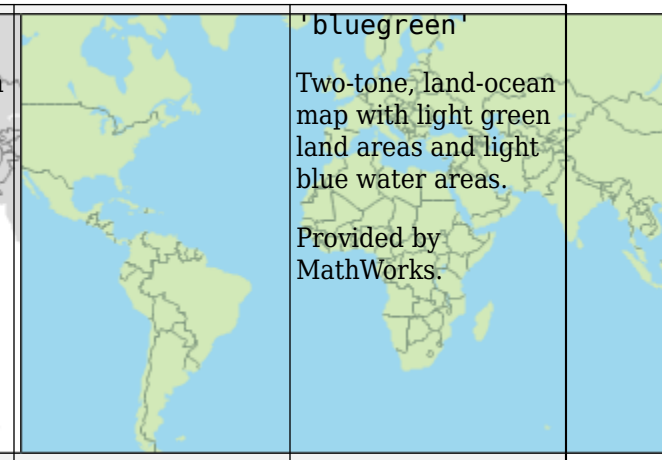
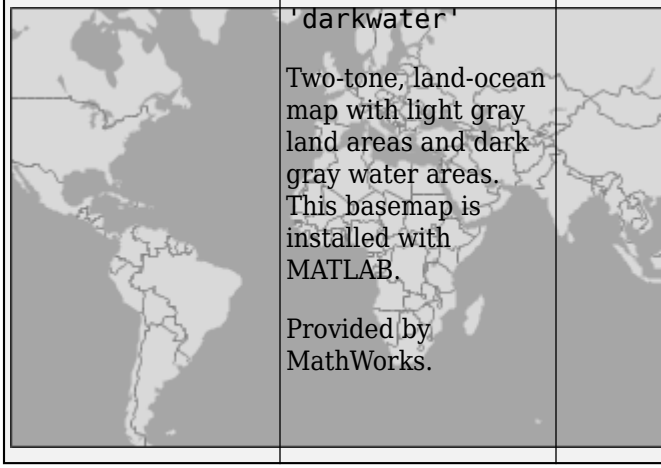
Provided by MathWorks.



`'colorterrain'`

Shaded relief map blended with a land cover palette. Humid lowlands are green and arid lowlands are brown.

Provided by MathWorks.

	`'grayland'` Two-tone, land-ocean map with gray land areas and white water areas. Provided by MathWorks.		`'bluegreen'` Two-tone, land-ocean map with light green land areas and light blue water areas. Provided by MathWorks.
	`'darkwater'` Two-tone, land-ocean map with light gray land areas and dark gray water areas. This basemap is installed with MATLAB. Provided by MathWorks.		`'none'` Blank background that plots your data with a latitude-longitude grid, ticks, and labels.

Example: `gx = geoaxes('Basemap','bluegreen')`

Example: `gx.Basemap = 'bluegreen'`

Data Types: `char` | `string`

`Position` — Size and location, excluding margin for labels
four-element vector

Size and location, excluding margin for labels, specified as a four-element vector of the form `[left bottom width height]`. For more information, see “Control Axes Layout”.

Units — Position units
`'normalized'` (default) | `'inches'` | `'centimeters'` | `'points'` | `'pixels'` | `'characters'`

Position units, specified as one of these values.

Units	Description
`'normalized'` (default)	Normalized with respect to the container, which is typically the figure or a panel. The lower left corner of the container maps to `(0,0)`, and the upper right corner maps to `(1,1)`.
`'inches'`	Inches
`'centimeters'`	Centimeters
`'characters'`	Based on the default uicontrol font of the graphics root object: • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.
`'points'`	Typography points. One point equals 1/72 inch.
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.

When specifying the units as a name-value pair during object creation, you must set the `Units` property before specifying the properties that you want to use these units, such as `OuterPosition`.

Output Arguments

gx — Geographic axes
`GeographicAxes` object

Geographic axes, returned as a `GeographicAxes` object.

Tips

- If you have Mapping Toolbox™, you can specify basemaps of your own choosing by using the `addCustomBasemap` function.
- Some graphics functions reset axes properties when plotting. To plot additional data in a geographic axes, use the `hold on` command before calls to plotting functions.
- You cannot plot data that requires Cartesian axes in a geographic chart.

See Also

Functions
`axes` | `cla` | `figure` | `gca` | `geobasemap` | `geolimits` | `geoplot` | `geoscatter` | `subplot`

Properties
GeographicAxes Properties

Introduced in R2018b

GeographicAxes Properties

Control geographic axes appearance and behavior

Description

`GeographicAxes` properties control the appearance and behavior of a `GeographicAxes` object. By changing property values, you can modify certain aspects of the geographic axes. Set axes properties after plotting since some graphics functions reset axes properties.

Some graphics functions create geographic axes when plotting. Use `gca` to access the newly created axes. To create a geographic axes with default values for all properties, use the `geoaxes` function.

```
gx = geoaxes;
```

Properties

Maps

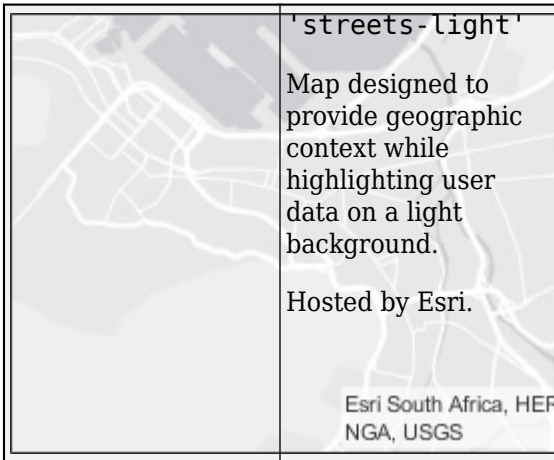
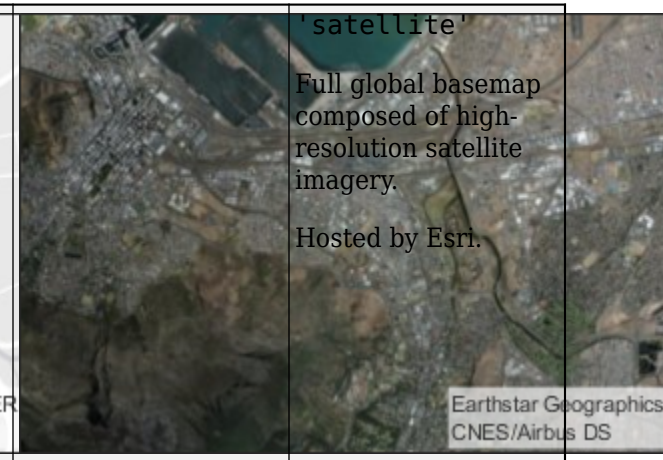
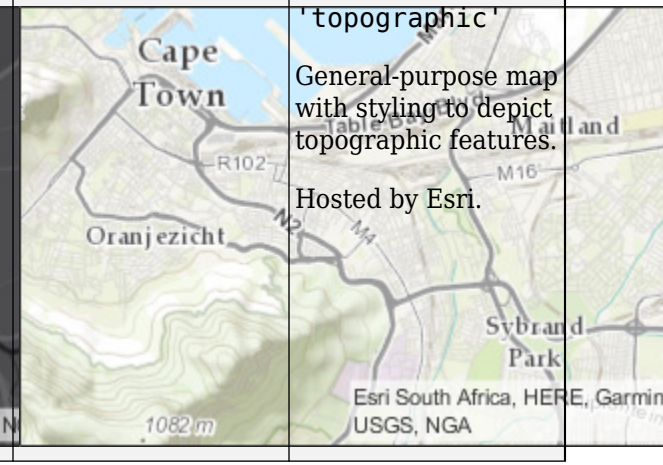
Basemap — Map on which to plot data
`'streets-light'` (default) | `'darkwater'` | `'colorterrain'` | `'satellite'` | ...

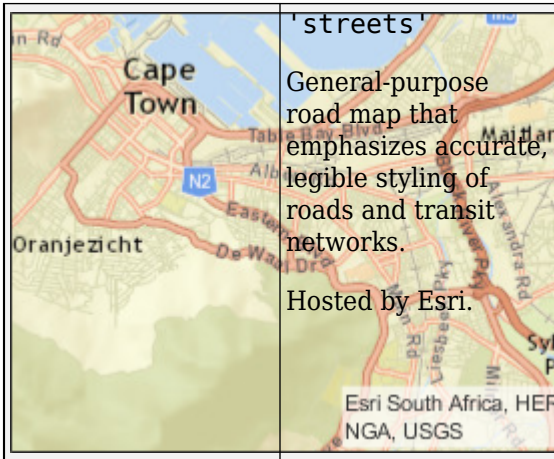
Map on which to plot data, specified as one of the values listed in the table. Six of the basemaps are tiled data sets that MathWorks derives from public-domain data. Five of the basemaps are high-zoom-level maps hosted by Esri.

MATLAB includes one installed basemap, a two-tone map named `'darkwater'`. Use of this basemap does not require Internet access. Use of the other basemaps, including the default basemap `'streets-light'`, does require Internet access.

If you do not have consistent access to the Internet, you can download the basemaps hosted by MathWorks onto your local system by using the Add-On Explorer. The five high-zoom-level maps are not available for download. For more about downloading basemaps and changing the default basemap on your local system, see “Access Basemaps in MATLAB”.

The five basemaps hosted by Esri include attributions that give credit to the provider of the map data. This attribution can change as you zoom in on a map because different organizations contribute map data at different zoom levels for different localities. Alignment of boundaries and region labels are a presentation of the feature provided by our data vendors and does not imply endorsement by MathWorks.

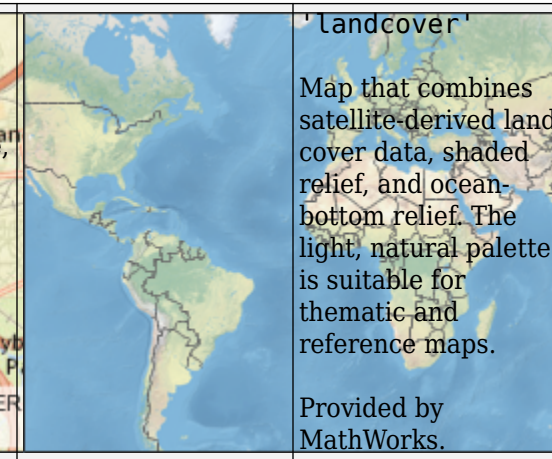
	`'streets-light'` Map designed to provide geographic context while highlighting user data on a light background. Hosted by Esri.		`'satellite'` Full global basemap composed of high-resolution satellite imagery. Hosted by Esri.
	`'streets-dark'` Map designed to provide geographic context while highlighting user data on a dark background. Hosted by Esri.		`'topographic'` General-purpose map with styling to depict topographic features. Hosted by Esri.



`'streets'`

General-purpose road map that emphasizes accurate, legible styling of roads and transit networks.

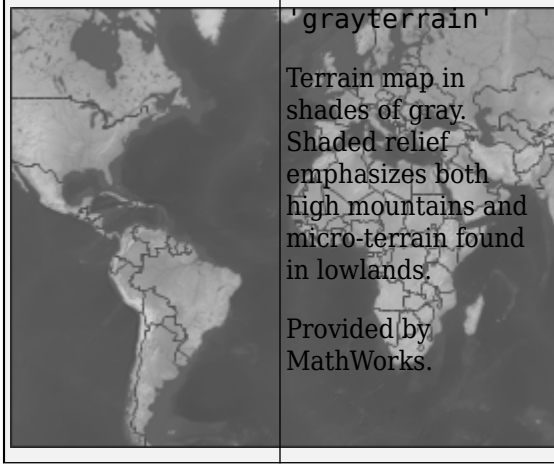
Hosted by Esri.



`'landcover'`

Map that combines satellite-derived land cover data, shaded relief, and ocean-bottom relief. The light, natural palette is suitable for thematic and reference maps.

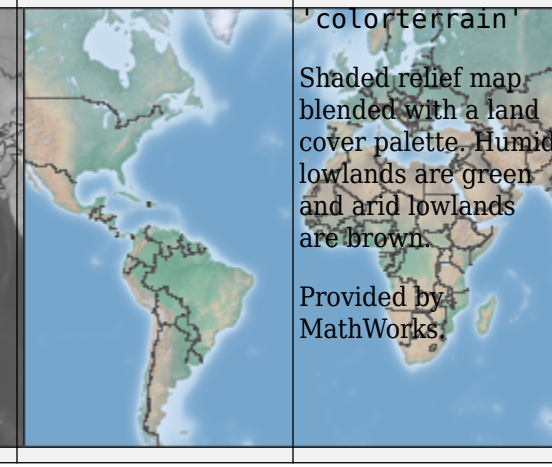
Provided by MathWorks.



`'grayterrain'`

Terrain map in shades of gray. Shaded relief emphasizes both high mountains and micro-terrain found in lowlands.

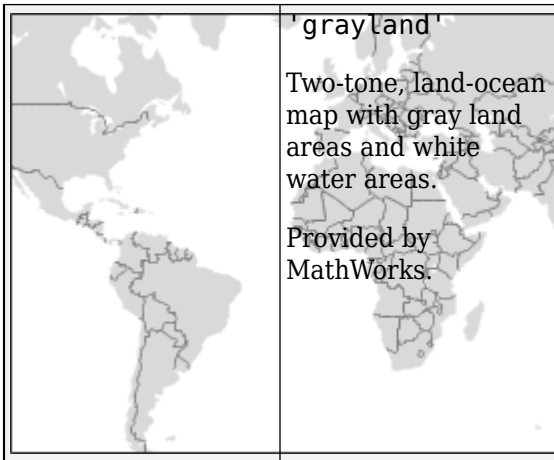
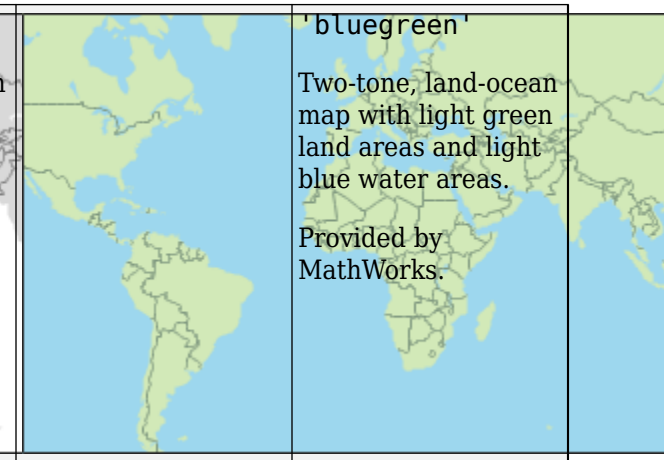
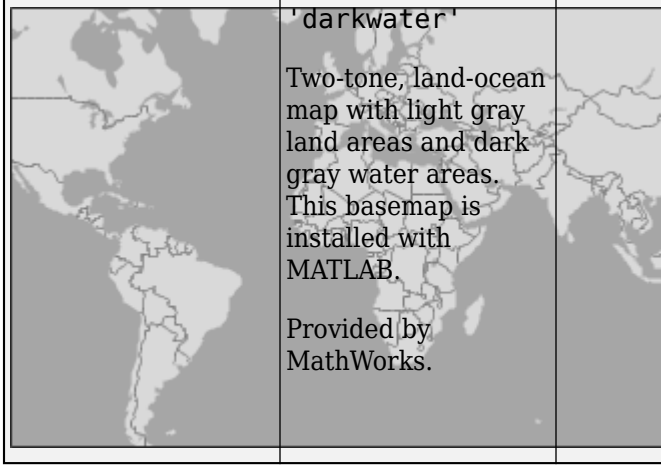
Provided by MathWorks.



`'colorterrain'`

Shaded relief map blended with a land cover palette. Humid lowlands are green and arid lowlands are brown.

Provided by MathWorks.

	`'grayland'` Two-tone, land-ocean map with gray land areas and white water areas. Provided by MathWorks.		`'bluegreen'` Two-tone, land-ocean map with light green land areas and light blue water areas. Provided by MathWorks.
	`'darkwater'` Two-tone, land-ocean map with light gray land areas and dark gray water areas. This basemap is installed with MATLAB. Provided by MathWorks.		`'none'` Blank background that plots your data with a latitude-longitude grid, ticks, and labels.

Example: `gx = geoaxes('Basemap','bluegreen')`

Example: `gx.Basemap = 'bluegreen'`

Data Types: `char` | `string`

`LatitudeLimits` — Latitude limits of map

1-by-2 vector of real, finite values

This property is read-only.

Latitude limits of map, specified as a 1-by-2 vector of real, finite values of the form `[southern_limit northern_limit]` in the range [-90,90]. Use the `geolimits` function to change latitude limits.

Example: `[-85 85]`

`LongitudeLimits` — Longitude limits of map
1-by-2 vector of real, finite values

This property is read-only.

Longitude limits of map, specified as a 1-by-2 vector of real, finite values of the form `[western_limit eastern_limit]`. Values must be in the range `(-Inf, Inf)`. Use the `geolimits` function to change longitude limits.

Example: `[-100 100]`

`MapCenter` — Center point of map in latitude and longitude
two-element numeric vector of real, finite values

Center point of map in latitude and longitude, specified as a two-element vector of real, finite values of the form `[center_latitude center_longitude]`.

Example: `[38.6292 -95.2520]`

`MapCenterMode` — Selection mode for map center
`'auto'` (default) | `'manual'`

Selection mode for the map center, specified as one of these values:

- `'auto'` — Object automatically selects the map center based on the range of data.
- `'manual'` — If you specify a value for `MapCenter`, the object sets this property to `'manual'` automatically.

Example: `gx.MapCenterMode = 'auto'`

`ZoomLevel` — Magnification level of map
real, finite, numeric scalar between 0 and 25, inclusive

Magnification level of map, specified as a real, finite, numeric scalar from 0 through 25, inclusive. The value is a base 2 logarithmic map scale. Increasing the `ZoomLevel` value by one doubles the map scale.

ZoomLevelMode — Selection mode for zoom level
`'auto'` (default) | `'manual'`

Selection mode for zoom level, specified as one of these values:

- `'auto'` — Object selects the zoom level based on the range of data.
- `'manual'` — If you specify a value for `ZoomLevel`, the object sets this property to `'manual'` automatically.

Example: `gx.ZoomLevelMode = 'manual'`

Scalebar — Scale bar showing proportional distances on map
`GeographicScalebar` object

This property is read-only.

Scale bar showing proportional distances on a map, specified as a `GeographicScalebar` object. To modify the appearance and behavior of the scale bar, such as its visibility, use properties of the `GeographicScalebar` object. For more information about these properties, see GeographicScalebar Properties.

Example: `sbar = gx.Scalebar` returns the `GeographicScalebar` object.

Example: `gx.Scalebar.Visible = 'off';` sets the value of the `GeographicScalebar` property.

Font

FontName — Font name
supported font name | `'FixedWidth'`

Font name, specified as a supported font name or `'FixedWidth'`. To display and print text properly, you must choose a font that your system supports. The default font depends on your operating system and locale.

To use a fixed-width font that looks good in any locale, use `'FixedWidth'`. The fixed-width font relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

FontSize — Font size
scalar numeric value

Font size, specified as a scalar numeric value. The font size affects the title, tick labels, legends, colorbars, and scale bar associated with the axes. The default font size depends on the specific operating system and locale. By default, the font size is measured in points. To change the units, set the `FontUnits` property.

MATLAB automatically scales some of the text to a percentage of the axes font size.

- Titles — 110% of the axes font size by default. To control the scaling, use the `TitleFontSizeMultiplier` and `LabelFontSizeMultiplier` properties.
- Legends and colorbars — 90% of the axes font size by default. To specify a different font size, set the `FontSize` property for the `Legend` or `ColorBar` object instead.
- Scale bar — 80% of the axes font size by default.

Example: `gx.FontSize = 12`

`FontSizeMode` — Selection mode for font size
`'auto'` (default) | `'manual'`

Selection mode for the font size, specified as one of these values:

- `'auto'` — Font size specified by MATLAB. If you resize the axes to be smaller than the default size, the font size might scale down to improve readability and layout.
- `'manual'` — Font size specified manually. Do not scale the font size as the axes size changes. To specify the font size, set the `FontSize` property.

`FontWeight` — Character thickness
`'normal'` (default) | `'bold'`

Character thickness, specified as `'normal'` or `'bold'`.

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.

`FontAngle` — Character slant
`'normal'` (default) | `'italic'`

Character slant, specified as `'normal'` or `'italic'`.

Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.

`LabelFontSizeMultiplier` — Scale factor for label font size
`1.1` (default) | numeric value greater than 0

Scale factor for the label font size, specified as a numeric value greater than 0. The scale factor is applied to the value of the `FontSize` property to determine the font size for the label.

Example: `gx.LabelFontSizeMultiplier = 1.75`

`TitleFontSizeMultiplier` — Scale factor for title font size
`1.1` (default) | numeric value greater than 0

Scale factor for the title font size, specified as a numeric value greater than 0. The scale factor is applied to the value of the `FontSize` property to determine the font size for the title.

Example: `gx.TitleFontSizeMultiplier = 1.75`

`TitleFontWeight` — Title character thickness
`'bold'` (default) | `'normal'`

Title character thickness, specified as one of these values:

- `'bold'` — Thicker characters outlines than normal
- `'normal'` — Default weight as defined by the particular font

Example: `gx.TitleFontWeight = 'normal'`

`FontUnits` — Font size units
`'points'` (default) | `'inches'` | `'centimeters'` | `'normalized'` | `'pixels'`

Font size units, specified as one of these values.

Units	Description
`'points'`	Points. One point equals 1/72 inch.
`'inches'`	Inches.
`'centimeters'`	Centimeters.

Units	Description
`'normalized'`	Interpret font size as a fraction of the axes height. If you resize the axes, the font size modifies accordingly. For example, if the `FontSize` is `0.1` in normalized units, then the text is 1/10 of the height value stored in the axes `Position` property.
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems. • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. • On Linux systems, the size of a pixel is determined by your system resolution.

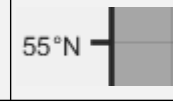
To set both the font size and the font units in a single function call, you first must set the `FontUnits` property so that the `Axes` object correctly interprets the specified font size.

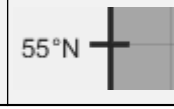
Ticks

`TickDir` — Tick mark direction

`'in'` (default) | `'out'` | `'both'`

Tick mark direction, specified as one of these values.

Value	Description	Example
`'in'`	Direct the tick marks inward from the axis lines.	
`'out'`	Direct the tick marks outward from the axis lines.	

Value	Description	Example
`'both'`	Center the tick marks over the axis lines.	

Example: `gx.TickDir = 'out';`

`TickDirMode` — Selection mode for tick mark direction
`'manual'` (default) | `'auto'`

Selection mode for tick mark direction set by the `TickDir` property, specified as one of these values.

- `'auto'` — Automatically select the tick direction based on the current view.
- `'manual'` — Manually specify the tick direction. To specify the tick direction, set the `TickDir` property.

Example: `gx.TickDirMode = 'auto';`

`TickLength` — Tick mark length
`[0.01 0.025]` (default) | two-element vector

Tick mark length, specified as a two-element vector of the form `[`*`length unused`*`]`. *length* is the tick mark length. Specify the values in units normalized relative to the longest axes dimension. The `GeographicRuler` object uses a two-element vector to be consistent with the value of this property in other ruler objects but the second element is unused.

Note Setting the `TickLength` property automatically sets the `TickLength` property in the `GeographicRuler` objects associated with the `LatitudeAxis` and `LongitudeAxis` properties to the same value. Conversely, setting the `TickLength` property in the `GeographicRuler` objects does not automatically set the same property in the axes object. To prevent the axes property value from overriding the ruler property value, set the axes property value first, and then set the ruler property value.

Example: `gx.TickLength = [0.02 0.0];`

`TickLabelFormat` — Tick label format
`'dms'` (default) | `'dd'` | `'dm'` | `'-dd'` | `'-dm'` | `'-dms'`

Tick label format, specified as one of the following values.

Format	Description	Example
`'dd'`	Decimal degrees plus compass direction	`23°N`
`'dm'`	Degrees and decimal minutes plus compass direction	`18°30'W`
`'dms'` (default)	Degrees, minutes, and decimal seconds plus compass direction	`110°06'18.5"E`
`'-dd'`	Decimal degrees with a minus sign (-) to indicate south and west	`-115.25°`
`'-dm'`	Degrees and decimal minutes with a minus sign (-) to indicate south and west	`-5°45.5'`
`'-dms'`	Degrees, minutes, and decimal seconds with a minus sign (-) to indicate south and west	`-3°21'05"`

The default label format includes degrees, minutes, and seconds. However, the minutes and seconds part of the tick label is not included until you zoom in on the map to at least a zoom level of 14.

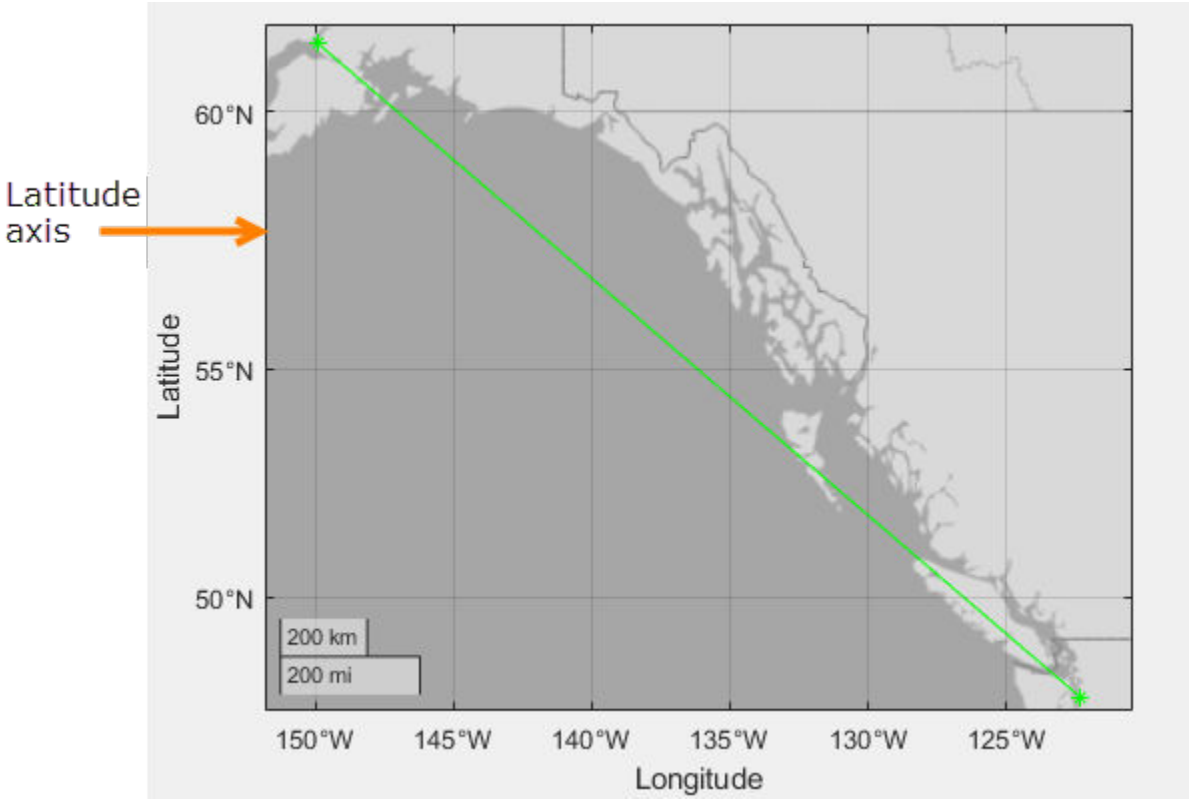
Example: `gx.TickLabelFormat = '-dm';`

Rulers

`LatitudeAxis` — Latitude ruler

`GeographicRuler` object

Latitude ruler, specified as a `GeographicRuler` object. Use properties of the `GeographicRuler` object to control the appearance and behavior of the axis ruler. For more information, see GeographicRuler Properties.

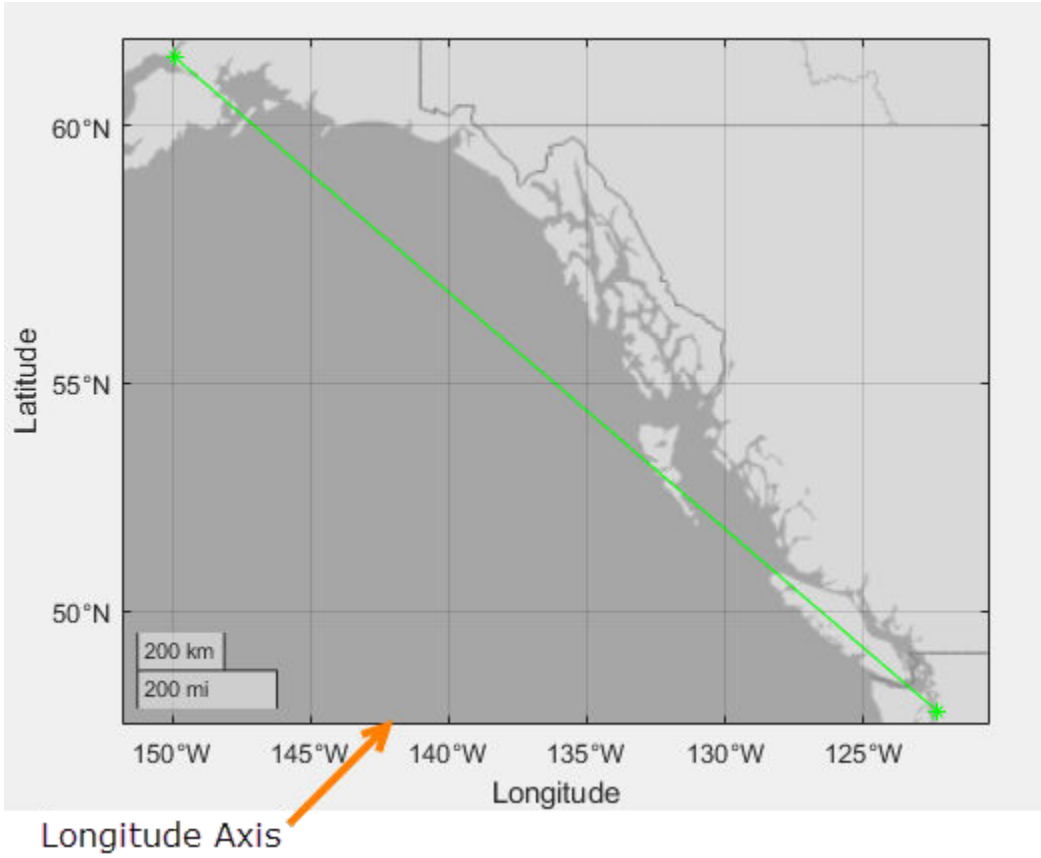


Example: `latruler = gx.LatitudeAxis;`

Example: `gx.LatitudeAxis.TickLabelRotation = 45;`

`LongitudeAxis` — Longitude ruler
`GeographicRuler`

Longitude ruler, specified as a `GeographicRuler` object. Use properties of the `GeographicRuler` object to control the appearance and behavior of the axis ruler. For more information, see GeographicRuler Properties.



Example: `lonruler = gx.LongitudeAxis;`

Example: `gx.LongitudeAxis.TickDirection = 'out';`

`AxisColor` — Color of axis lines, tick values, and labels
`[0.1500 0.1500 0.1500]` (default) | RGB triplet | hexadecimal color code | color name | short color name

Color of axis lines, tick values, and labels, specified as an RGB triplet, hexadecimal color code, color name, or short color name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

Note Setting the `AxisColor` property automatically sets the `Color` property in the `GeographicRuler` and `GeographicScalebar` objects to the same value. The `GeographicRuler` object controls the behavior and appearance of the rulers in the geographic axes. The `GeographicScalebar` object controls the scale bar in the geographic axes. Conversely, setting the `Color` property in the `GeographicRuler` or `GeographicScalebar` object does not automatically set the `AxisColor` property in the axes object. To prevent the axes property value from overriding the ruler or scale bar property value, set the axes property value first, and then set the ruler or scale bar property value.

Example: `gx.AxisColor = [0 0 1];`

Example: `gx.AxisColor = 'b';`

Example: `gx.AxisColor = 'blue';`

Example: `gx.AxisColor = '#0000FF';`

Grids

`Grid` — Visibility of latitude and longitude lines on the map

`'on'` (default) | `'off'`

Visibility of latitude and longitude lines on map, specified as `'on'`, show grid lines, or `'off'`, do not show grid lines.

Example: `gx.Grid = 'off';`

`GridLineStyle` — Line style for grid lines

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style for grid lines, specified as one of the line styles in this table.

Line Style	Description	Resulting Line
'-'	Solid line	

Line Style	Description	Resulting Line
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

To display the grid lines, use the `grid on` command or set the `Grid` property to `'on'`.

Example: `gx.GridLineStyle = '--'`

`GridColor` — Color of grid lines

`[0.15 0.15 0.15]` (default) | RGB triplet | hexadecimal color code | color name | short color name

Background color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short color name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

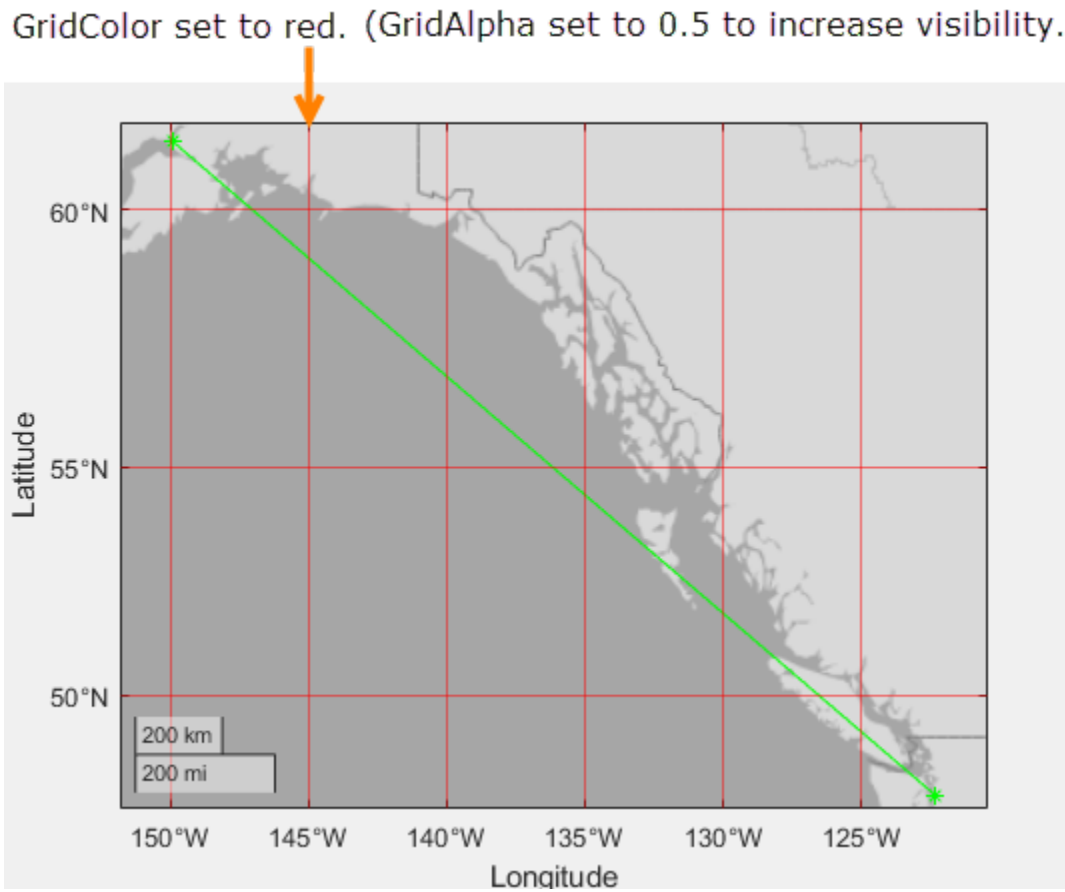
Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	



Example: `gx.GridColor = [0 0 1];`

Example: `gx.GridColor = 'b';`

Example: `gx.GridColor = 'blue';`

Example: `gx.GridColor = '#0000FF';`

`GridColorMode` — Property for setting grid color
`'auto'` (default) | `'manual'`

Property for setting the grid color, specified as one of these values:

- `'auto'` — Object automatically selects the color.
- `'manual'` — To set the grid line color for all directions, use `GridColor`.

`GridAlpha` — Grid-line transparency

`0.15` (default) | value in the range `[0,1]`

Grid-line transparency, specified as a value in the range `[0,1]`. A value of `1` means opaque and a value of `0` means completely transparent.

Example: `gx.GridAlpha = 0.5`

`GridAlphaMode` — Selection mode for `GridAlpha`

`'auto'` (default) | `'manual'`

Selection mode for the `GridAlpha` property, specified as one of these values:

- `'auto'` — Object selects the transparency value.
- `'manual'` — To specify the transparency value, use the `GridAlpha` property.

Example: `gx.GridAlphaMode = 'auto'`

Labels

`Title` — Axes title

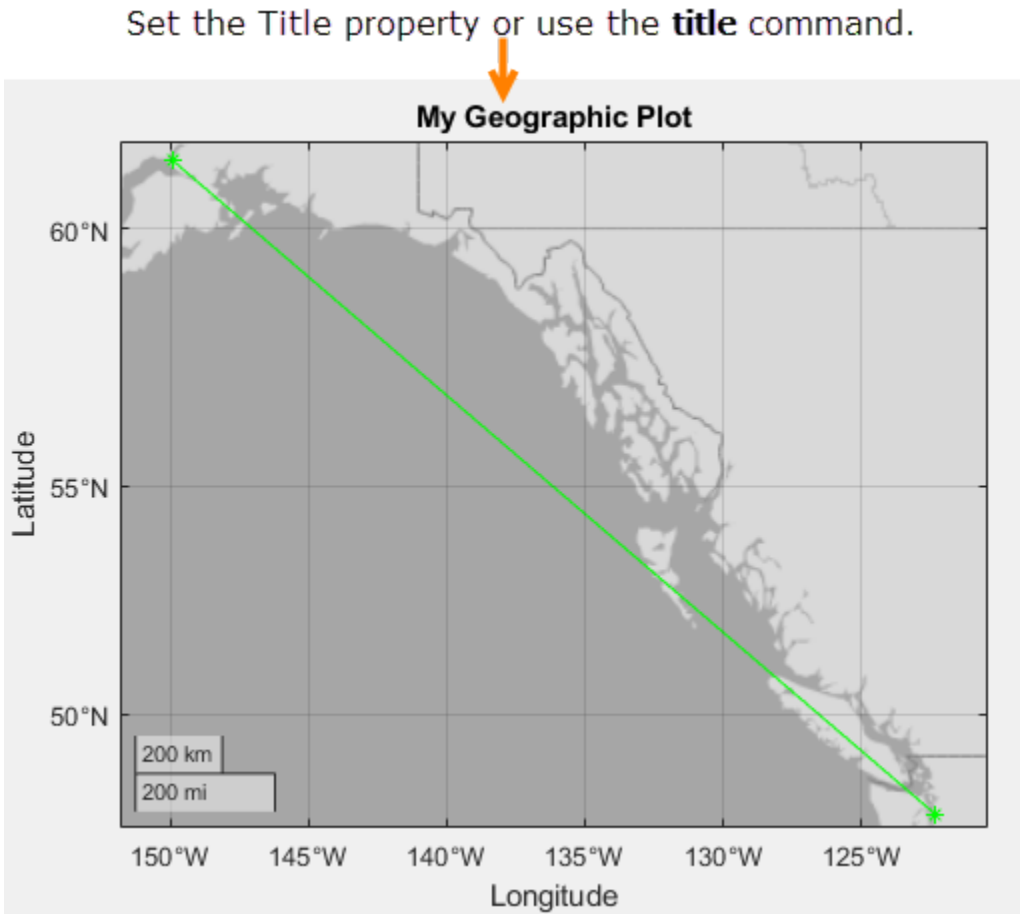
`Text` object | `categorical`

Axes title, specified as a `Text` object or a `categorical` value.

If you use a `Text` object, specify the title as the value of the `String` property of the `Text` object: `gx.Title.String = 'My Geographic Plot'`. To change the title appearance, such as the font style or color, use other `Text` object properties. For a complete list of properties, see Text Properties.

If you specify this property as a categorical array, MATLAB uses the values in the array, not the categories.

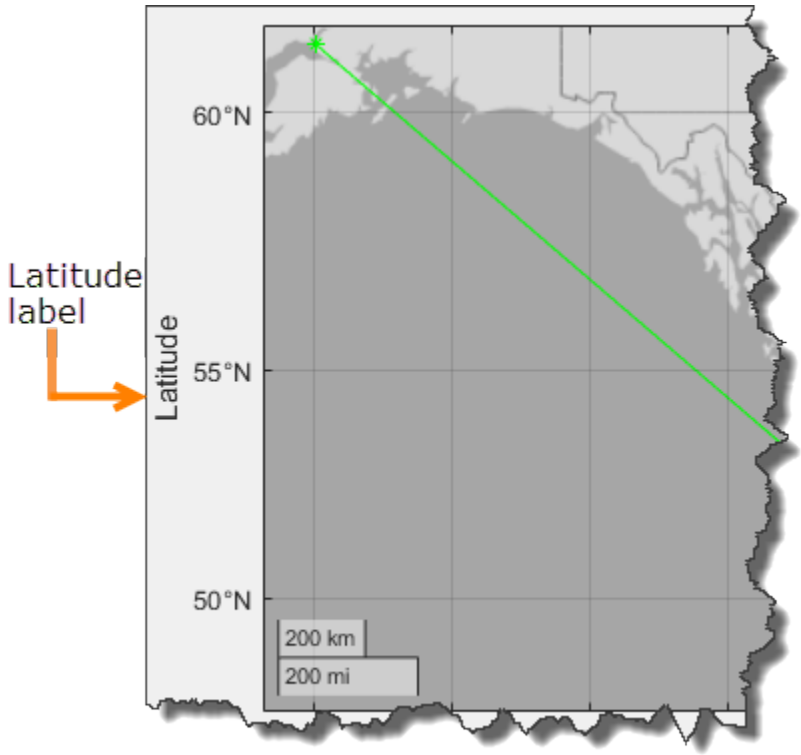
You can also use the `title` function to specify a title.



Example: `gx.Title.String = 'My Geographic Plot'`

`LatitudeLabel` — Latitude axis label
`Text` object

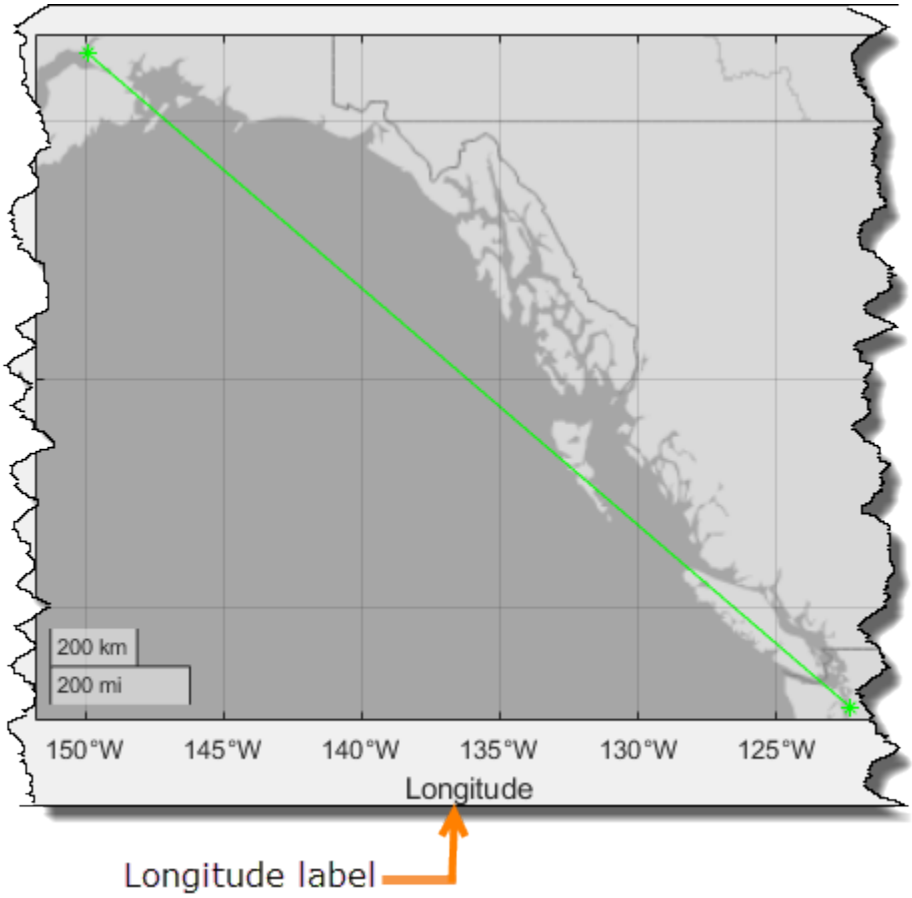
Latitude axis label, specified as a `Text` object. To specify a label, set the `String` property of the `Text` object. To change the label appearance, such as the font style or color, set other `Text` object properties. For a complete list of properties, see Text Properties.



Example: `gx.LatitudeLabel.String = 'My Latitude'`

`LongitudeLabel` — Longitude axis label
`Text` object

Longitude axis label, specified as a `Text` object. To specify a label, set the `String` property of the text object. To change the label appearance, such as the font style or color, set other `Text` object properties. For a complete list of properties, see Text Properties.



Example: `gx.LongitudeLabel.String = 'My Longitude'`

Legend — Legend associated with geographic axes
`empty GraphicsPlaceholder` (default) | `Legend` object

This property is read-only.

Legend associated with a geographic axes, specified as a `Legend` object. To add a legend to the geographic axes, use the `legend` function. Then, you can use this property to modify the legend. For a complete list of properties, see Legend.

```
geoplot(rand(3))
legend({'Line 1','Line 2','Line 3'},'FontSize',12)
gx = gca;
gx.Legend.TextColor = 'red';
```

You also can use this property to determine if the geographic axes has a legend.

```
gx = gca;
lgd = gx.Legend
if ~isempty(lgd)
    disp('Legend Exists')
end
```

Multiple Plots

`ColorOrder` — Color order

seven predefined colors (default) | three-column matrix of RGB triplets

Color order, specified as a three-column matrix of RGB triplets. This property defines the palette of colors MATLAB uses to create plot objects such as `Line`, `Scatter`, and `Bar` objects. Each row of the array is an RGB triplet. An RGB triplet is a three-element vector whose elements specify the intensities of the red, green, and blue components of a color. The intensities must be in the range [0, 1]. This table lists the default colors.

Colors	`ColorOrder` Matrix

	`[ 0 0.4470 0.7410`
	`0.8500 0.3250 0.0980`
	`0.9290 0.6940 0.1250`
	`0.4940 0.1840 0.5560`
	`0.4660 0.6740 0.1880`
	`0.3010 0.7450 0.9330`
	`0.6350 0.0780 0.1840]`

MATLAB assigns colors to objects according to their order of creation. For example, when plotting lines, the first line uses the first color, the second line uses the second color, and so on. If there are more lines than colors, then the cycle repeats.

Changing the Color Order Before or After Plotting

Starting in R2019b, you can change the color order in either of the following ways:

- Call the `colororder` function to change the color order for all the axes in a figure. The colors of existing plots in the figure update immediately. If you place additional axes into the figure, those axes also use the new color order. If you continue to call plotting commands, those commands also use the new colors.
- Set the `ColorOrder` property on the axes, call the `hold` function to set the axes hold state to `'on'`, and then call the desired plotting functions. This is like calling the `colororder` function, but in this case you are setting the color order for the specific axes, not the entire figure. Setting the `hold` state to `'on'` is necessary to ensure that subsequent plotting commands do not reset the axes to use the default color order.

If you are using R2019a or an earlier release, changing the color order matrix does not affect existing plots. To change colors in an existing plot, you must set the `ColorOrder` property, and then set the axes hold state to `'on'` before calling any plotting functions.

The behavior of this property changed in R2019b. For more information, see:

- "Changing ColorOrder or LineStyleOrder on the axes affects existing plots immediately" on page 1-4886
- "Indexing scheme for ColorOrder and LineStyleOrder might change plot colors and line styles" on page 1-4886

`ColorOrderIndex` — Color order index

1 (default) | positive integer

Color order index, specified as a positive integer. This property specifies the next color MATLAB selects from the axes `ColorOrder` property when it creates the next plot object such as a `Line`, `Scatter`, or `Bar` object. For example, if the color order index value is `1`, then the next object added to the axes uses the first color in the `ColorOrder` matrix. If the index value exceeds the number of colors in the `ColorOrder` matrix, then the index value modulo of the number of colors in the `ColorOrder` matrix determines the color of the next object.

When the `NextPlot` property of the axes is set to `'add'`, then the color order index value increases every time you add a new plot to the axes. To start again with first color, set the `ColorOrderIndex` property to `1`.

The behavior of this property changed in R2019b. For more information, see "Indexing scheme for ColorOrder and LineStyleOrder might change plot colors and line styles" on page 1-4886.

`LineStyleOrder` — Line style order

`'-'` solid line (default) | character vector | cell array of character vectors | string array

Line style order, specified as a character vector, a cell array of character vectors, or a string array. This property lists the line styles that MATLAB uses to display multiple plot lines in the axes. MATLAB assigns styles to lines according to their order of creation. It changes to the next line style only after cycling through all the colors in the `ColorOrder` property with the current line style. The default `LineStyleOrder` has only one line style, `'-'`.

To customize the line style order, create a cell array of character vectors or a string array. Specify each element of the array as a line specifier or marker specifier from the following tables. You can combine a line and a marker specifier into a single element, such as `'-*'`.

Line Specifier	Description
`'-'` (default)	Solid line
`'--'`	Dashed line
`':'`	Dotted line
`'-.'`	Dash-dotted line

Marker Specifier	Description
`'+'`	Plus sign markers
`'o'`	Circle markers
`'*'`	Star markers
`'.'`	Point markers
`'x'`	Cross markers
`'s'`	Square markers
`'d'`	Diamond markers
`'^'`	Upward-pointing triangle markers
`'v'`	Downward-pointing triangle markers
`'>'`	Right-pointing triangle markers
`'<'`	Left-pointing triangle markers
`'p'`	Five-pointed star (pentagram) markers

Marker Specifier	Description
`'h'`	Six-pointed star (hexagram) markers

Changing Line Style Order Before or After Plotting

Starting in R2019b, you can change the line style order before or after plotting into the axes. When you set the `LineStyleOrder` property to a new value, MATLAB updates the styles of any lines that are in the axes. If you continue plotting into the axes, your plotting commands continue using the line styles from the updated list.

If you are using R2019a or an earlier release, you must change the line style order before plotting. Set the value of the `LineStyleOrder` property, and then call the `hold` function to set the axes hold state to `'on'` before calling any plotting functions.

The behavior of this property changed in R2019b. For more information, see:

- "Changing ColorOrder or LineStyleOrder on the axes affects existing plots immediately" on page 1-4886
- "Indexing scheme for ColorOrder and LineStyleOrder might change plot colors and line styles" on page 1-4886

`LineStyleOrderIndex` — Line style order index

1 (default) | positive integer

Line style order index, specified as a positive integer. This property specifies the next line style MATLAB selects from the axes `LineStyleOrder` property to create the next plot line. For example, if this property is set to `1`, then the next plot line you add to the axes uses the first item in the `LineStyleOrder` property. If the index value exceeds the number of line styles in the `LineStyleOrder` array, then the index value modulo of the number of elements in the `LineStyleOrder` array determines the style of the next line.

When the `NextPlot` property of the axes is set to `'add'`, MATLAB increments the index value after cycling through all the colors in the `ColorOrder` property with the current line style. To start again with first line style, set the `LineStyleOrderIndex` property to `1`.

The behavior of this property changed in R2019b. For more information, see "Indexing scheme for ColorOrder and LineStyleOrder might change plot colors and line styles" on page 1-4886.

`NextPlot` — Properties to reset

`'replace'` (default) | `'add'` | `'replacechildren'` | `'replaceall'`

Properties to reset when adding a new plot to the axes, specified as one of these values:

- `'add'` — Add new plots to the existing axes. Do not delete existing plots or reset axes properties before displaying the new plot.
- `'replacechildren'` — Delete existing plots before displaying the new plot. Reset the `ColorOrderIndex` and `LineStyleOrderIndex` properties to `1`, but do not reset other axes properties. The next plot added to the axes uses the first color and line style based on the `ColorOrder` and `LineStyle` order properties. This value is similar to using `cla` before every new plot.
- `'replace'` — Delete existing plots and reset axes properties, except `Position` and `Units`, to their default values before displaying the new plot.
- `'replaceall'` — Delete existing plots and reset axes properties, except `Position` and `Units`, to their default values before displaying the new plot. This value is similar to using `cla reset` before every new plot.

Figures also have a `NextPlot` property. Alternatively, you can use the `newplot` function to prepare figures and axes for subsequent graphics commands.

`SortMethod` — Order for rendering objects

`'childorder'` | `'depth'`

Order for rendering objects, specified as one of these values:

- `'depth'` — Draw objects in back-to-front order based on the current view. Use this value to ensure that objects in front of other objects are drawn correctly.
- `'childorder'` — Draw objects in the order in which they are created by graphics functions, without considering the relationship of the objects in three dimensions. This value can result in faster rendering, particularly if the figure is very large, but also can result in improper depth sorting of the objects displayed.

Color and Transparency Maps

`Colormap` — Colormap

`parula` (default) | *m*-by-3 array of RGB triplets

Colormap, specified as an *m*-by-3 array of RGB (red, green, blue) triplets that define *m* individual colors. Alternatively, you can use the `colormap` function to change the color map.

MATLAB accesses these colors by their row number.

Example: `gx.Colormap = [1 0 1; 0 0 1; 1 1 0]` sets the color map to three colors: magenta, blue, and yellow.

`ColorScale` — Scale for color mapping
`'linear'` (default) | `'log'`

Scale for color mapping, specified as one of these values:

- `'linear'` — Linear scale. The tick values along the colorbar also use a linear scale.
- `'log'` — Log scale. The tick values along the colorbar also use a log scale.

Example: `gx.ColorScale = 'log'`

`CLim` — Color limits for colormap
`[0 1]` (default) | two-element vector of the form `[cmin cmax]`

Color limits for the colormap, specified as a two-element vector of the form `[cmin cmax]`.

If the associated mode property is set to `'auto'`, then MATLAB chooses the color limits. If you assign a value to this property, then MATLAB sets the mode to `'manual'` and does not automatically choose the color limits.

`CLimMode` — Selection mode for `CLim`
`'auto'` (default) | `'manual'`

Selection mode for the `CLim` property, specified as one of these values:

- `'auto'` — Automatically select the limits based on the color data of the graphics objects contained in the axes.
- `'manual'` — Manually specify the values. To specify the values, set the `CLim` property. The values do not change when the limits of the axes children change.

`Alphamap` — Transparency map
array of 64 values from `0` to `1` (default) | array of finite alpha values from `0` to `1`

Transparency map, specified as an array of finite alpha values that progress linearly from `0` to `1`. The size of the array can be *m*-by-1 or 1-by-*m*. MATLAB accesses alpha values by their index in the array. An alphamap can be any length.

`AlphaScale` — Scale for transparency mapping
`'linear'` (default) | `'log'`

Scale for transparency mapping, specified as one of these values:

- `'linear'` — Linear scale
- `'log'` — Log scale

Example: `gx.AlphaScale = 'log'`

`ALim` — Alpha limits for alphamap

`[0 1]` (default) | two-element vector of the form `[amin amax]`

Alpha limits for alphamap, specified as a two-element vector of the form `[amin amax]`.

If the associated mode property is set to `'auto'`, then MATLAB chooses the alpha limits. If you set this property, then MATLAB sets the mode to `'manual'` and does not automatically choose the alpha limits.

`ALimMode` — Selection mode for `ALim`

`'auto'` (default) | `'manual'`

Selection mode for the `ALim` property, specified as one of these values:

- `'auto'` — Automatically select the limits based on the `AlphaData` values of the graphics objects contained in the axes.
- `'manual'` — Manually specify the alpha limits. To specify the alpha limits, set the `ALim` property.

Box Styling

`Color` — Background color

`[1 1 1]` (default) | RGB triplet | hexadecimal color code | color name | color short name | ...

Background color, specified as an RGB triplet, a hexadecimal color code, a color name, or a color short name. The background color is only visible when the `Basemap` property is set to `'none'`.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `gx.Color = [0 0 1];`

Example: `gx.Color = 'b';`

Example: `gx.Color = 'blue';`

Example: `gx.Color = '#0000FF';`

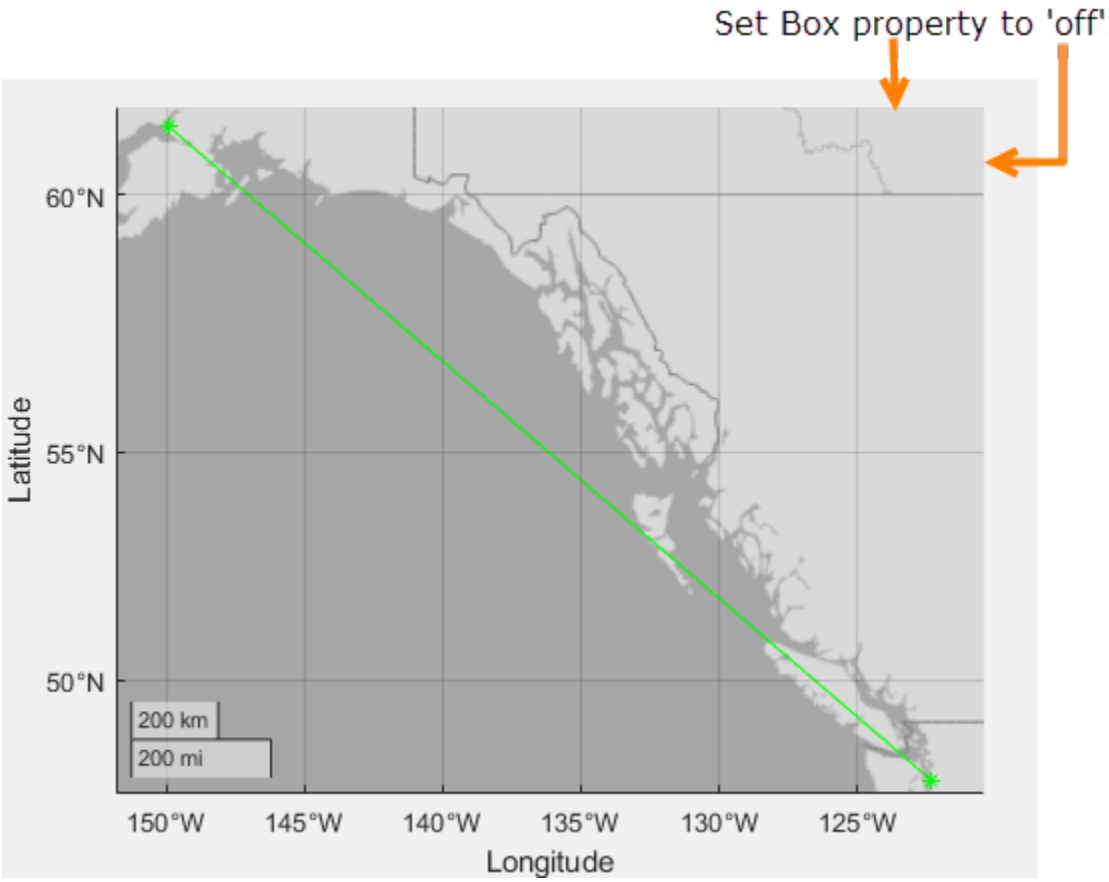
`LineWidth` — Width of lines
`0.5` (default) | positive scalar value

Width of lines, specified as a positive scalar value in point units. One point equals 1/72 inch.

Example: `gx.LineWidth = 1.5`

`Box` — Outline around geographic axes
`'on'` (default) | `'off'`

Outline around the geographic axes, specified as either `'on'` or `'off'`.



Example: `gx.Box = 'off'`

Position

`OuterPosition` — Size and position of geographic axes, including labels and margins

`[0 0 1 1]` (default) | four-element vector

Size and position of the geographic axes, including the labels and margins, specified as a four-element vector of the form `[left bottom width height]`. This vector defines the extents of the rectangle that encloses the outer bounds of the geographic axes. The `left` and `bottom` elements define the distance from the lower-left corner of the figure or panel

that contains the geographic axes to the lower-left corner of the rectangle. The `width` and `height` elements are the rectangle dimensions.

By default, the values are measured in units normalized to the container. To change the units, set the `Units` property. The default value of `[0 0 1 1]` includes the whole interior of the container.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`Position` — Size and position of geographic axes, not including labels or margins

`[0.1300 0.1100 0.7750 0.8150]` (default) | four-element vector

Size and position of the geographic axes, not including labels or margins, specified as a four-element vector of the form `[left bottom width height]`. This vector defines the extents of the tightest bounding rectangle that encloses the geographic axes. The `left` and `bottom` elements define the distance from the lower-left corner of the container to the lower-left corner of the rectangle. The `width` and `height` elements are the rectangle dimensions.

By default, the values are measured in units normalized to the container. To change the units, set the `Units` property.

Example: `gx.Position = [0 0 1 1]` specifies no distance between the lower-left corner of the container to the rectangle and width and height to fill the entire container.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`TightInset` — Margins for text labels

four-element vector of the form `[left bottom right top]`

This property is read-only.

Margins for the text labels, returned as a four-element vector of the form `[left bottom right top]`. This property is read-only.

The elements define the distances between the bounds of the `Position` property and the extent of the geographic axes text labels and title. By default, the values are measured in units normalized to the figure or uipanel that contains the geographic axes. To change the units, set the `Units` property.

The `Position` property and the `TightInset` property define the tightest bounding box that encloses the geographic axes and its labels and title.

`ActivePositionProperty` — Active position property
`'outerposition'` | `'position'`

Position property to hold constant when adding, removing, or changing decorations, specified as one of the following values:

- `'outerposition'` — The `OuterPosition` property remains constant when you add, remove, or change decorations such as a title or an axis label. If any positional adjustments are needed, MATLAB adjusts the `Position` property.
- `'position'` — The `Position` property remains constant when you add, remove, or change decorations such as a title or an axis label. If any positional adjustments are needed, MATLAB adjusts the `OuterPosition` property.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`Units` — Position units
`'normalized'` (default) | `'inches'` | `'centimeters'` | `'points'` | `'pixels'` | `'characters'`

Position units, specified as one of these values.

Units	Description
`'normalized'` (default)	Normalized with respect to the container, which is typically the figure or a panel. The lower left corner of the container maps to `(0,0)` and the upper right corner maps to `(1,1)`.
`'inches'`	Inches
`'centimeters'`	Centimeters

Units	Description
`'characters'`	Based on the default uicontrol font of the graphics root object: • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.
`'points'`	Typography points. One point equals 1/72 inch.
`'pixels'`	Pixels. • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. • On Linux systems, the size of a pixel is determined by your system resolution.

When specifying the units as a `Name,Value` pair during object creation, you must set the `Units` property before specifying the properties that you want to use these units, such as `Position`.

`Layout` — Layout options

empty `LayoutOptions` array (default) | `TiledChartLayoutOptions` object

Layout options, specified as a `TiledChartLayoutOptions` object. This property specifies options when the axes is a child of a tiled chart layout. Use this property to position the axes within the layout by setting the `Tile` and `TileSpan` properties on the `TiledChartLayoutOptions` object.

For example, this code places axes `ax` in the third tile of a tiled chart layout.

```
ax.Layout.Tile = 3;
```

To make the axes span multiple tiles, specify the `TileSpan` property as a two-element vector. For example, this axes spans `2` rows and `3` columns of tiles.

```
ax.Layout.TileSpan = [2 3];
```

If the axes is not a child of a tiled chart layout (for example, if it is a child of a figure or panel), then this property is empty and has no effect.

Interactivity

`Toolbar` — Data exploration toolbar

`AxesToolbar` object

Data exploration toolbar, specified as an `AxesToolbar` object. The toolbar appears at the top-right corner of the geographic axes when you hover over it. The toolbar provides quick access to data exploration tools, such as zooming, restore view, and data tips.

If you do not want the toolbar to appear when you hover over the geographic axes, set the `Visible` property of the `AxesToolbar` object to `'off'`. For more information about the properties of an `AxesToolbar` object, see AxesToolbar Properties.

Example: `gx.Toolbar.Visible = 'off'`

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility of the geographic axes, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`CurrentPoint` — Location of mouse pointer

2-by-3 array

This property is read-only.

Location of mouse pointer, specified as a 2-by-3 array of the form:

```
[lat lon 0
 lat lon 0]
```

The `CurrentPoint` property contains the latitude (`lat`) and longitude (`lon`) coordinates of the mouse pointer with respect to the geographic axes. The `(lat,lon)` points indicate the location of the last mouse click. However, if the figure has a `WindowButtonMotionFcn` callback defined, then the `(lat,lon)` points indicate the last location of the mouse pointer.

The format of the return value is consistent with the return value of the `CurrentPoint` property of the `Axes` object. For geographic axes, the third column of the return value is always zero. The latitude and longitude values in the second row are duplicates of the values in the first row.

Example: `[52.1411 -125.1167 0; 52.1411 -125.1167 0]`

`UIContextMenu` — Context menu
empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state
`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles
`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

Callbacks

`ButtonDownFcn` — Mouse-click callback
`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see "Callback Definition".

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you

do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks only when visible. The `Visible` property must be set to `'on'`. The `HitTest` property determines if the `GeographicAxes` object responds to the click or if an ancestor does.
- `'all'` — Capture mouse clicks regardless of visibility. The `Visible` property can be set to `'on'` or `'off'`. The `HitTest` property determines if the `GeographicAxes` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `GeographicAxes` object passes the click to the object below it in the current view of the figure window, which is typically the axes or the figure. The `HitTest` property has no effect.

If you want an object to be clickable when it is underneath other objects that you do not want to be clickable, then set the `PickableParts` property of the other objects to `'none'` so that the click passes through them.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `GeographicAxes` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `GeographicAxes` object that has one of these:
  - `HitTest` property set to `'on'`
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

Note The `PickableParts` property determines if the `GeographicAxes` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container

`Figure` object | `Panel` object | `Tab` object | `TiledChartLayout` object

Parent container, specified as a `Figure`, `Panel`, `Tab`, or `TiledChartLayout` object.

`Children` — Children

empty `GraphicsPlaceholder` array | array of graphics objects

Children, returned as an array of graphics objects. Use this property to view a list of the children or to reorder the children by setting the property to a permutation of itself.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the child graphics object to the `GeographicAxes` object.

HandleVisibility — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'geoaxes'` (default)

This property is read-only.

Type of graphics object, returned as `'geoaxes'`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

UserData — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

Compatibility Considerations

Changing ColorOrder or LineStyleOrder on the axes affects existing plots immediately

Behavior changed in R2019b

Starting in R2019b, if you change the axes `ColorOrder` or `LineStyleOrder` properties after plotting into the axes, the colors and line styles in your plot update immediately. In R2019a and previous releases, the new colors and line styles affect only subsequent plots, not the existing plots.

To preserve the original behavior, set the axes `ColorOrderIndex` or `LineStyleOrderIndex` property to any value (such as its current value) before changing the `ColorOrder` or `LineStyleOrder` property.

Indexing scheme for ColorOrder and LineStyleOrder might change plot colors and line styles

Behavior changed in R2019b

Starting in R2019b, there is a new indexing scheme that enables you to change the colors and line styles of existing plots by setting the `ColorOrder` or `LineStyleOrder` properties. MATLAB applies this indexing scheme to all objects that have a `ColorMode`, `FaceColorMode`, `MarkerFaceColorMode`, or `CDataMode`. As a result, your code might produce plots that cycle though the colors and line styles differently than in previous releases.

In R2019a and earlier releases, MATLAB uses a different indexing scheme which does not allow you to change the colors of existing plots.

To preserve the way your plots cycle through colors and line styles, set the axes `ColorOrderIndex` or `LineStyleOrderIndex` property to any value (such as its current value) before plotting into the axes.

See Also

`cla` | `gca` | `geoaxes` | `geobasemap` | `geolimits` | `geotickformat`

Topics

"Graphics Object Properties"

Introduced in R2018b

geobasemap

Set or query basemap

Syntax

```
geobasemap basemap
geobasemap(basemap)
geobasemap(gx,basemap)

bmap = geobasemap
bmap = geobasemap(gx)
```

Description

`geobasemap basemap` sets the `Basemap` property for the current geographic axes or chart to the value specified by `basemap`. When you use this syntax, you do not need to enclose the `basemap` argument in quotes, for example `geobasemap topographic`.

`geobasemap(basemap)` sets the basemap for the current geographic axes or chart. When you use this syntax, enclose the `basemap` argument in single or double quotes, for example `geobasemap('topographic')`.

`geobasemap(gx,basemap)` sets the basemap for the geographic axes or chart specified by `gx`.

`bmap = geobasemap` returns the value of the `Basemap` property for the current geographic axes or chart.

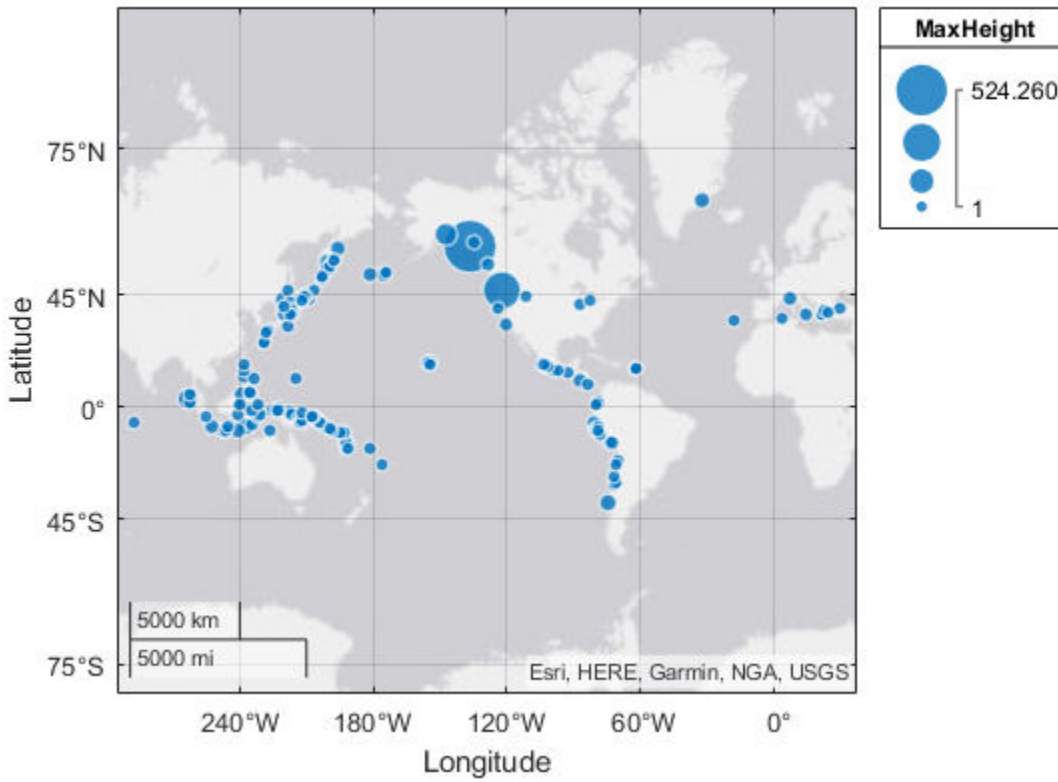
`bmap = geobasemap(gx)` returns the basemap value for the geographic axes or chart specified by `gx`.

Examples

Specify Basemap of Geographic Axes or Chart

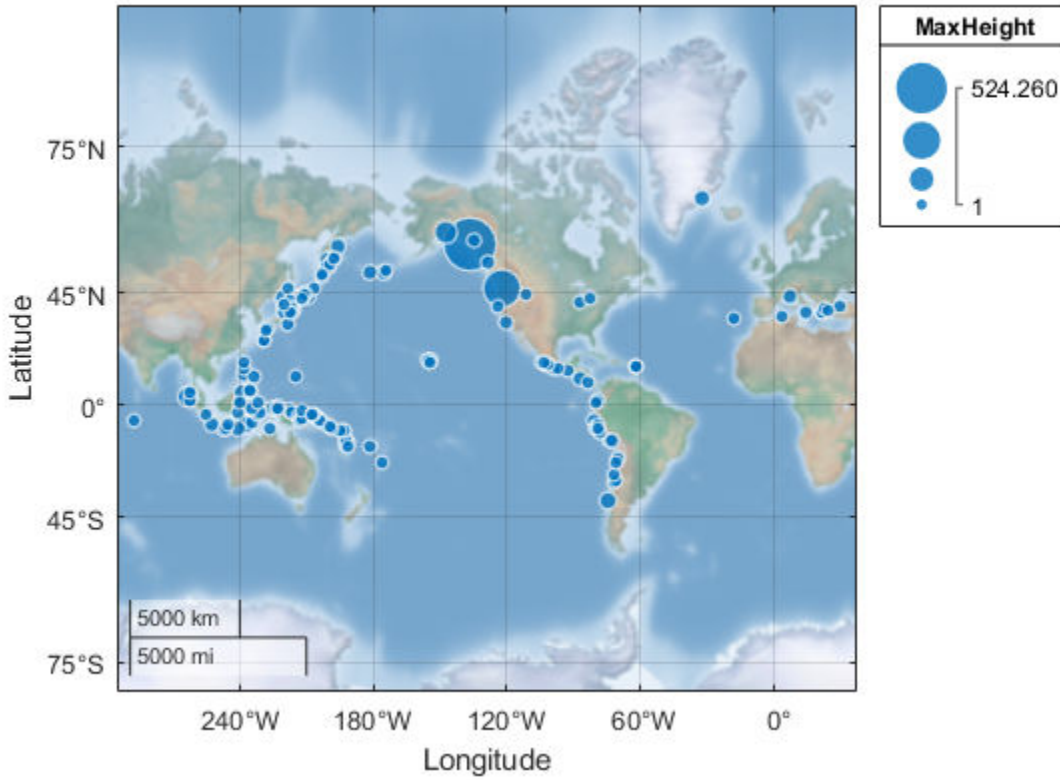
Create a geographic bubble chart. By default, the geographic bubble chart uses the `'streets-light'` basemap.

```
tsunamis = readtable('tsunamis.xlsx');
geobubble(tsunamis,'Latitude','Longitude','SizeVariable','MaxHeight');
```



Change the basemap using the `geobasemap` function.

```
geobasemap colorterrain
```



Input Arguments

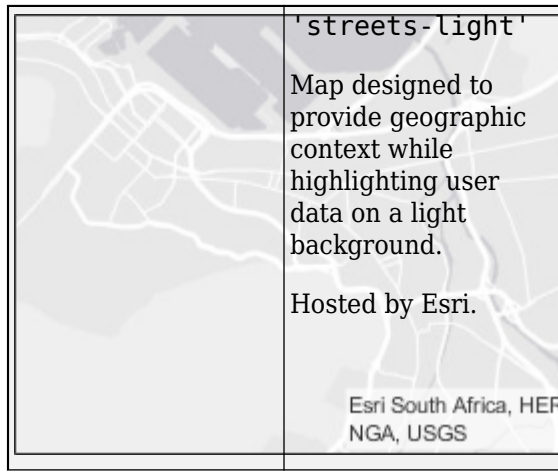
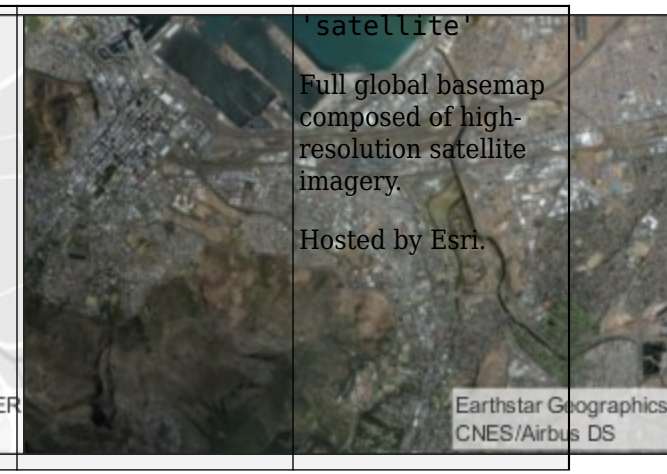
basemap — Map on which to plot data
`'streets-light'` (default) | `'darkwater'` | `'colorterrain'` | `'satellite'` | ...

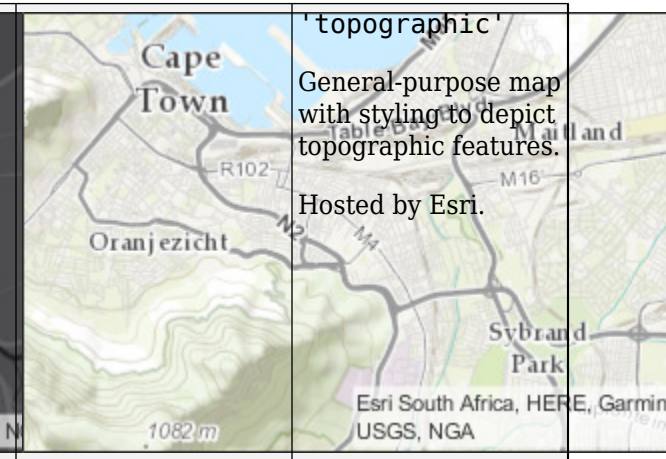
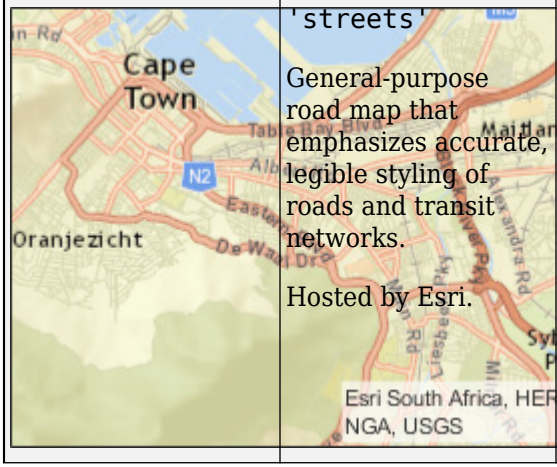
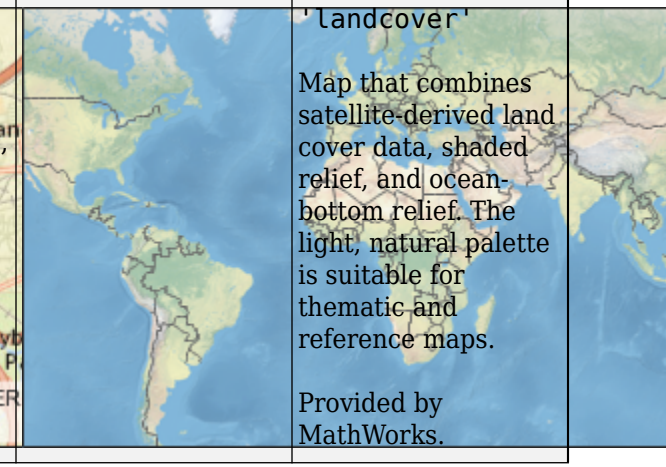
Map on which to plot data, specified as one of the values listed in the table. Six of the basemaps are tiled data sets that MathWorks derives from public-domain data. Five of the basemaps are high-zoom-level maps hosted by Esri.

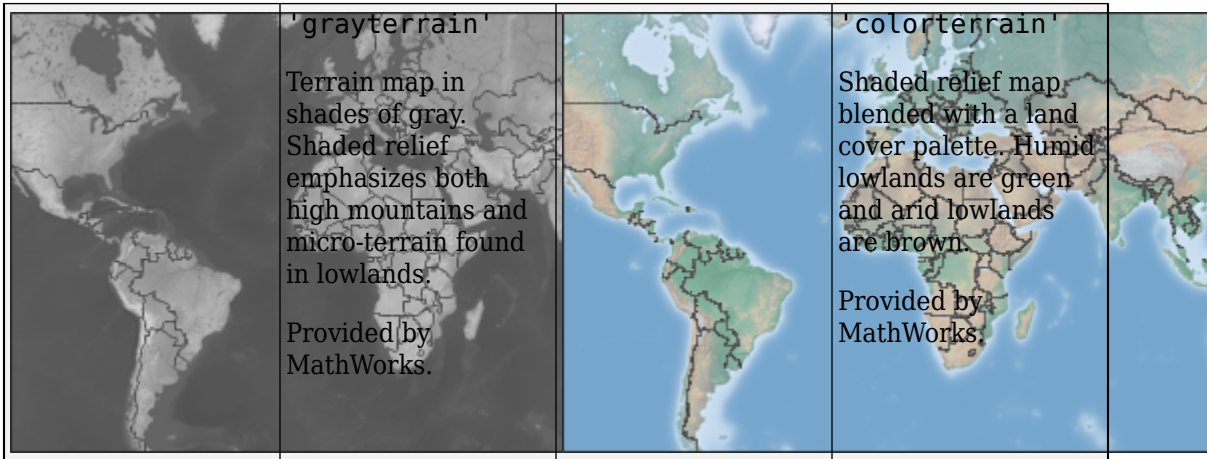
MATLAB includes one installed basemap, a two-tone map named `'darkwater'`. Use of this basemap does not require Internet access. Use of the other basemaps, including the default basemap `'streets-light'`, does require Internet access.

If you do not have consistent access to the Internet, you can download the basemaps hosted by MathWorks onto your local system by using the Add-On Explorer. The five high-zoom-level maps are not available for download. For more about downloading basemaps and changing the default basemap on your local system, see “Access Basemaps in MATLAB”.

The five basemaps hosted by Esri include attributions that give credit to the provider of the map data. This attribution can change as you zoom in on a map because different organizations contribute map data at different zoom levels for different localities. Alignment of boundaries and region labels are a presentation of the feature provided by our data vendors and does not imply endorsement by MathWorks.

	`'streets-light'` Map designed to provide geographic context while highlighting user data on a light background. Hosted by Esri.		`'satellite'` Full global basemap composed of high-resolution satellite imagery. Hosted by Esri.

	`'streets-dark'` Map designed to provide geographic context while highlighting user data on a dark background. Hosted by Esri.		`'topographic'` General-purpose map with styling to depict topographic features. Hosted by Esri.
	`'streets'` General-purpose road map that emphasizes accurate, legible styling of roads and transit networks. Hosted by Esri.		`'landcover'` Map that combines satellite-derived land cover data, shaded relief, and ocean-bottom relief. The light, natural palette is suitable for thematic and reference maps. Provided by MathWorks.



`'grayterrain'`

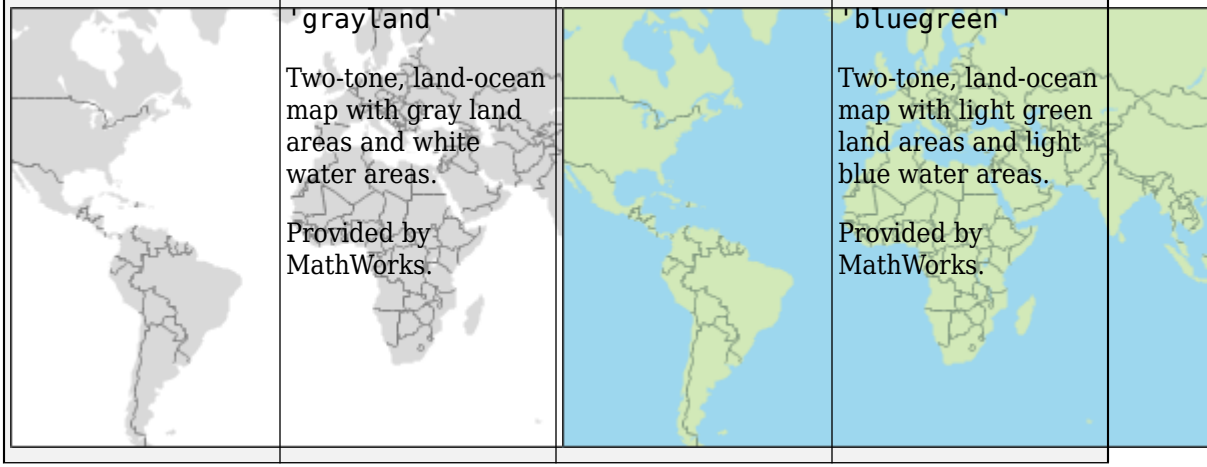
Terrain map in shades of gray. Shaded relief emphasizes both high mountains and micro-terrain found in lowlands.

Provided by MathWorks.

`'colorterrain'`

Shaded relief map blended with a land cover palette. Humid lowlands are green and arid lowlands are brown.

Provided by MathWorks.



`'grayland'`

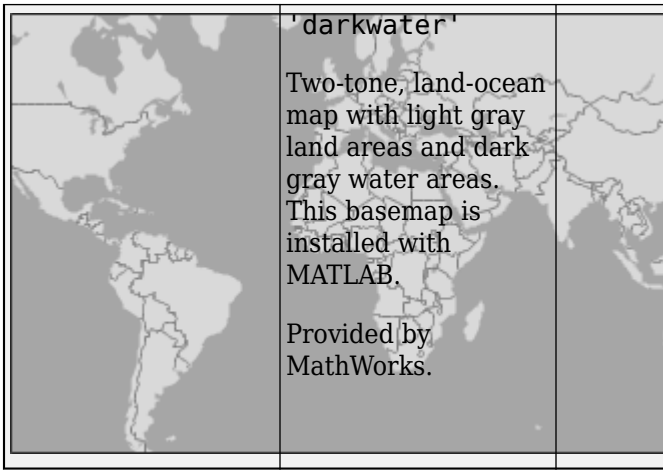
Two-tone, land-ocean map with gray land areas and white water areas.

Provided by MathWorks.

`'bluegreen'`

Two-tone, land-ocean map with light green land areas and light blue water areas.

Provided by MathWorks.

	`'darkwater'` Two-tone, land-ocean map with light gray land areas and dark gray water areas. This basemap is installed with MATLAB. Provided by MathWorks.		`'none'` Blank background that plots your data with a latitude-longitude grid, ticks, and labels.

Example: `gx = geoaxes('Basemap','bluegreen')`

Example: `gx.Basemap = 'bluegreen'`

Data Types: `char` | `string`

gx — Geographic axes or chart
`gca` (default) | `GeographicAxes` object | `GeographicBubbleChart` object

Geographic axes or chart, specified as a `GeographicAxes` object or `GeographicBubbleChart` object.

Tips

Specify custom basemaps using the Mapping Toolbox function `addCustomBasemap`.

See Also

`geoaxes` | `geobubble` | `geodensityplot` | `geoplot` | `geoscatter`

Introduced in R2018b

geobubble

Visualize data values at specific geographic locations

Syntax

```
gb = geobubble(tbl,latvar,lonvar)
gb = geobubble(tbl,latvar,lonvar,Name,Value)

gb = geobubble(lat,lon)
gb = geobubble(lat,lon,sizedata)
gb = geobubble(lat,lon,sizedata,colordata)
gb = geobubble( ___ ,Name,Value)

gb = geobubble(parent, ___ )
```

Description

`gb = geobubble(tbl,latvar,lonvar)` creates a geographic bubble chart with filled circles (“bubbles”) representing the geographic locations specified in the table `tbl` displayed on a map. `latvar` identifies the table variable (column) that specifies bubble latitudes. `lonvar` identifies the table variable that specifies bubble longitudes. By default, the bubbles are all the same size and the same color. The function returns `gb`, a `GeographicBubbleChart` object. To modify this object, use the properties described in GeographicBubbleChart Properties.

The geographic bubble chart displays your data on a map, called a basemap. Initially, the chart sets the geographic limits of the chart to encompass all of your data. The map is *live*, that is, you can pan the basemap to view other geographic locations. You can also zoom in and out on the map to view regions in more detail. The chart updates the map as you pan and zoom. For more information about geographic bubble charts and basemaps, see “Geographic Bubble Charts Overview”.

`gb = geobubble(tbl,latvar,lonvar,Name,Value)` uses additional options specified by one or more name-value pair arguments to set the values of geographic bubble chart properties. Two key properties are `'SizeVariable'` and

`'ColorVariable'`, which specify the table variables that determine the size and color of the bubbles.

`gb = geobubble(lat,lon)` creates a geographic bubble chart where `lat` and `lon` specify the geographic locations. By default, the bubbles are all the same size and color.

`gb = geobubble(lat,lon,sizedata)` scales the areas of the bubbles according to the numeric values in `sizedata`.

`gb = geobubble(lat,lon,sizedata,colordata)` uses the categorical data in `colordata` to determine the color of the bubbles. `geobubble` chooses a color for each category in `colordata`, plus one additional color if any element of `colordata` is undefined. Colors are drawn from an ordered list of 7 standard colors. If there are more than seven categories (more than six, if undefined values are present), the colors repeat cyclically.

`gb = geobubble( ___ ,Name,Value)` specifies additional options for the geographic bubble chart using one or more name-value pair arguments. Specify the options after all other input arguments. For a list of properties, see GeographicBubbleChart Properties.

`gb = geobubble(parent, ___ )` creates the geographic bubble chart in the figure, panel, tab, or tiled chart layout specified by `parent`.

Examples

Display Data from Tsunamis Table on Geographic Bubble Chart

Read data about tsunamis into the workspace as a table. The rows represent individual tsunami occurrences. The columns represent data about a set of variables for each tsunami, such as their locations (latitude and longitude), causes, and wave heights.

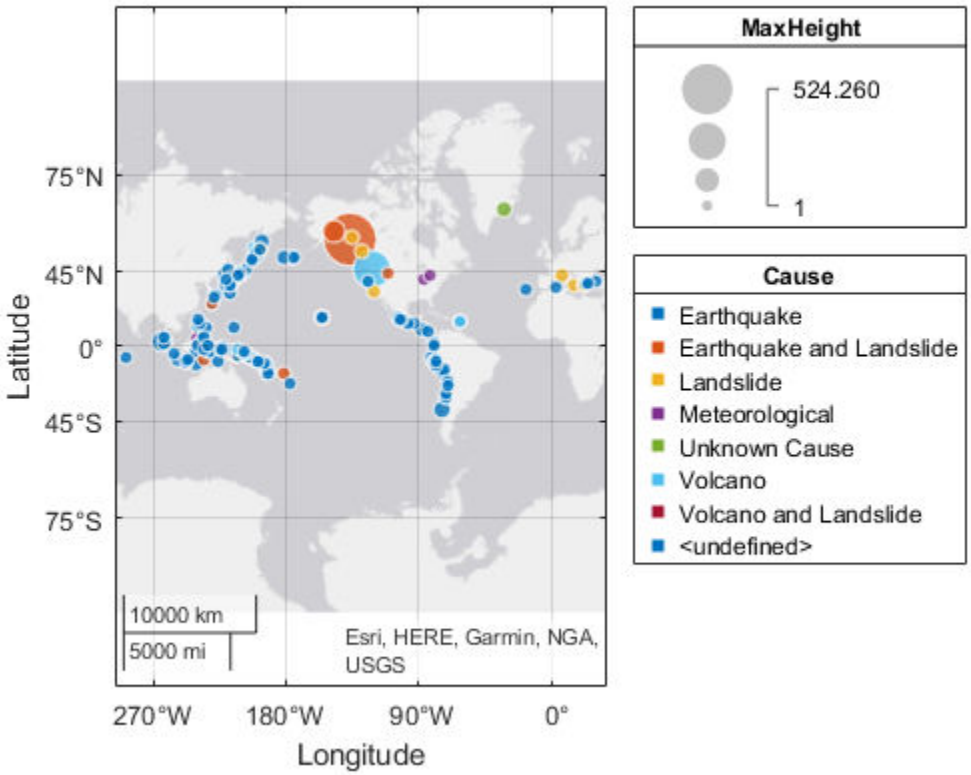
```
tsunamis = readtable('tsunamis.xlsx');
```

Convert one of the table variables into a categorical variable to specify the color of the bubbles. The `Cause` variable attributes a cause to each tsunami, such as, `'Earthquake'`, `'Volcano'`, or `'Earthquake and Landslide'`. Convert the `Cause` variable from a cell array of character vectors into a categorical variable.

```
tsunamis.Cause = categorical(tsunamis.Cause);
```

Create a geographic bubble chart, plotting the locations of the tsunamis on a map. Specify the names of the table variables that hold location information: `Latitude` and `Longitude`. Use the `MaxHeight` table variable to specify the size of the bubbles. The example uses the `Cause` variable, converted earlier into a categorical variable, to specify the color of the bubbles.

```
gb = geobubble(tsunamis,'Latitude','Longitude', ...
    'SizeVariable','MaxHeight','ColorVariable','Cause')
```




```
gb 
  GeographicBubbleChart with properties:

             Basemap: 'streets-light'
           MapLayout: 'normal'
```

```
          SourceTable: [162×20 table]
     LatitudeVariable: 'Latitude'
    LongitudeVariable: 'Longitude'
         SizeVariable: 'MaxHeight'
        ColorVariable: 'Cause'

  Show all properties
```

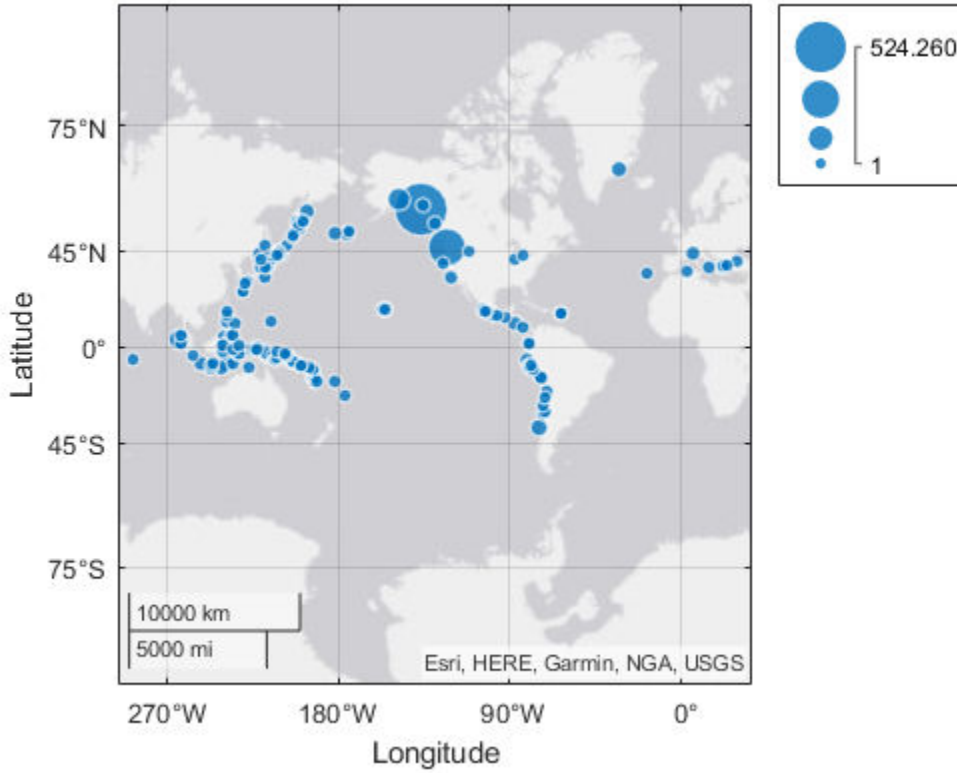
Display Tsunami Data Using Bubble Size to Indicate Tsunami Size

Read data about tsunamis into the workspace as a table. The rows represent individual tsunami occurrences. The columns represent data about a set of variables for each tsunami, such as their locations (latitude and longitude), causes, and wave heights.

```
tsunamis = readtable('tsunamis.xlsx');
```

Create a geographic bubble chart, plotting the locations of the tsunamis on a map. Use the data from the `MaxHeight` variable to specify the size of the bubble. In this example, you pass the data directly to `geobubble`. Alternatively, you can also pass `geobubble` the name of the table and then specify the data by table variable names.

```
gb = geobubble(tsunamis.Latitude,tsunamis.Longitude,tsunamis.MaxHeight)
```




```
gb =
  GeographicBubbleChart with properties:

          Basemap: 'streets-light'
        MapLayout: 'normal'
     LatitudeData: [162×1 double]
    LongitudeData: [162×1 double]
         SizeData: [162×1 double]
        ColorData: []

  Show all properties
```

Display Tsunami Data Using Bubble Color to Indicate Tsunami Cause

Read data about tsunamis into the workspace as a table. The rows represent individual tsunami occurrences. The columns represent data about each occurrence, such as the cause of each tsunami.

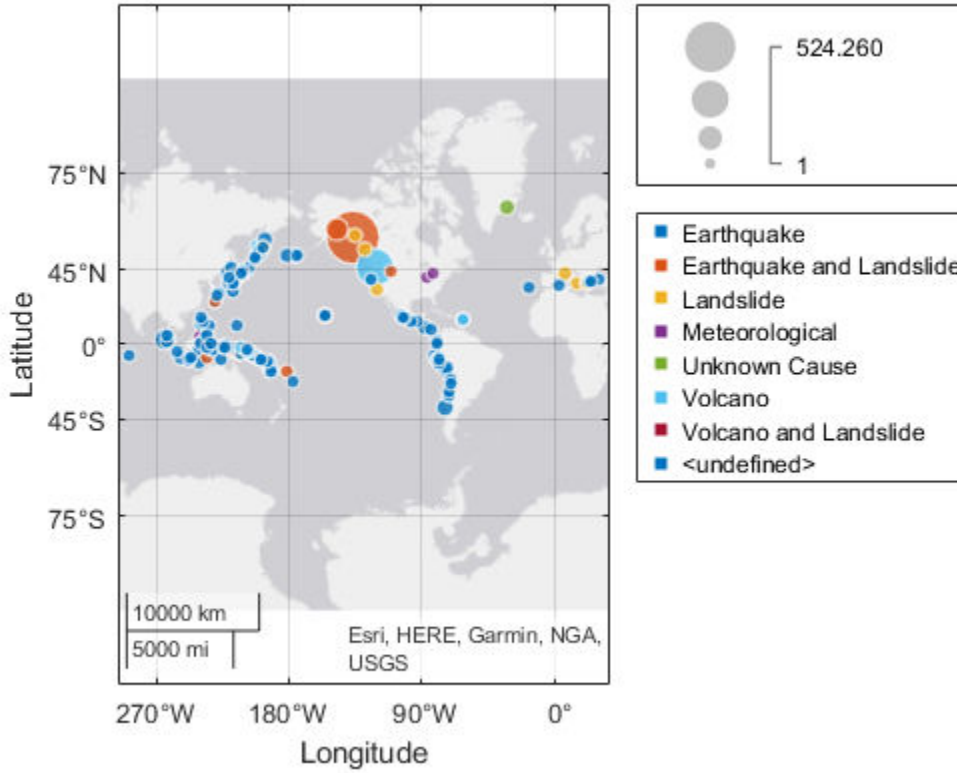
```
tsunamis = readtable('tsunamis.xlsx');
```

Create a categorical variable because the data that controls the color of the bubbles must be a categorical variable. The tsunami table variable `Cause` already categorizes the tsunamis by seven criteria: Earthquake, Earthquake and Landslide, Volcano, Volcano and Landslide, Landslide, Meteorological, and Unknown Cause. Put the `Cause` variable data into a categorical variable.

```
cause = categorical(tsunamis.Cause);
```

Create a geographic bubble chart, plotting the locations of the tsunamis on a map. Use bubble size to indicate the size of the wave, and use bubble color to indicate the cause. When you pass in the data, rather than specifying the names of table variables that contain the data, `geobubble` does not automatically add titles to the size and color legends.

```
gb = geobubble(tsunamis.Latitude,tsunamis.Longitude,tsunamis.MaxHeight,cause)
```




```
gb =
  GeographicBubbleChart with properties:

          Basemap: 'streets-light'
        MapLayout: 'normal'
     LatitudeData: [162×1 double]
    LongitudeData: [162×1 double]
         SizeData: [162×1 double]
        ColorData: [162×1 categorical]

  Show all properties
```

Add Legend Titles Using Geographic Bubble Chart Properties

Read data about tsunamis into the workspace as a table. The rows represent individual tsunami occurrences. The columns represent data about a set of variables for each tsunami, such as its location (latitude and longitude), cause, and maximum height of the wave.

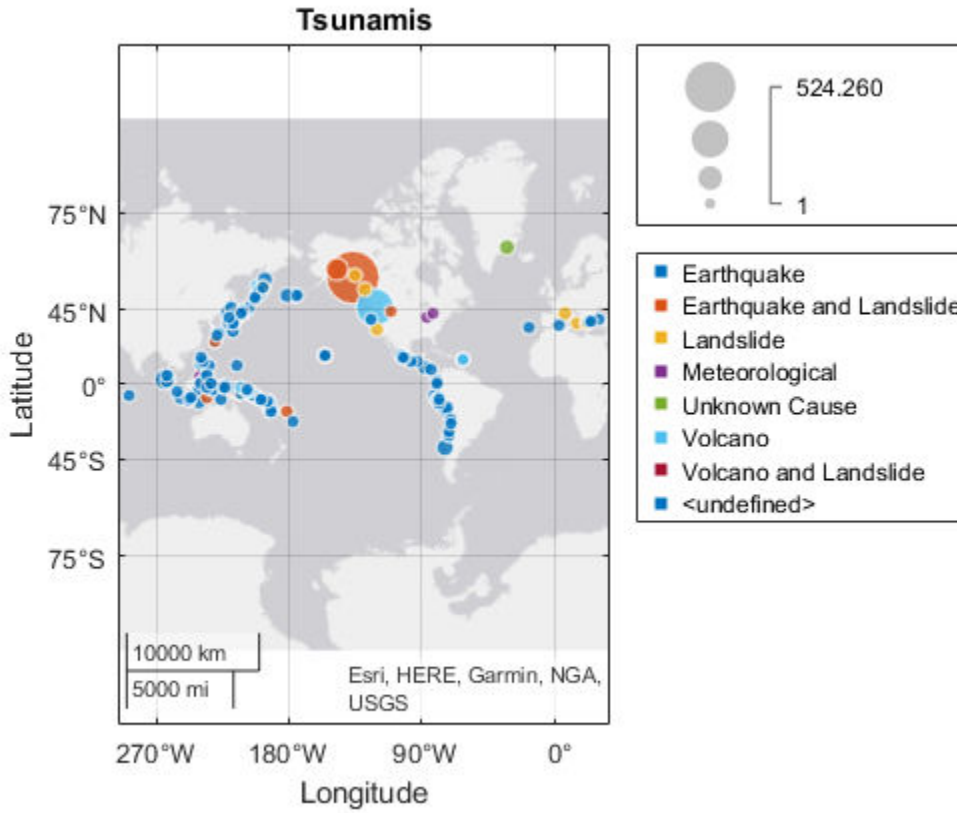
```
tsunamis = readtable('tsunamis.xlsx');
```

Turn one of the table variables into a categorical variable to specify the color of the bubbles. The `Cause` variable attributes a cause to each tsunami, such as, `'Earthquake'`, `'Volcano'`, or `'Earthquake and Landslide'`. Convert the `Cause` variable from a cell array of character vectors into a categorical variable.

```
colordata = categorical(tsunamis.Cause);
```

Create a geographic bubble chart, plotting the locations of the tsunamis on a map. Use bubble size to indicate the size of the wave, and use bubble color to indicate the cause of the tsunami. Also set the basemap and the title using name-value pair arguments.

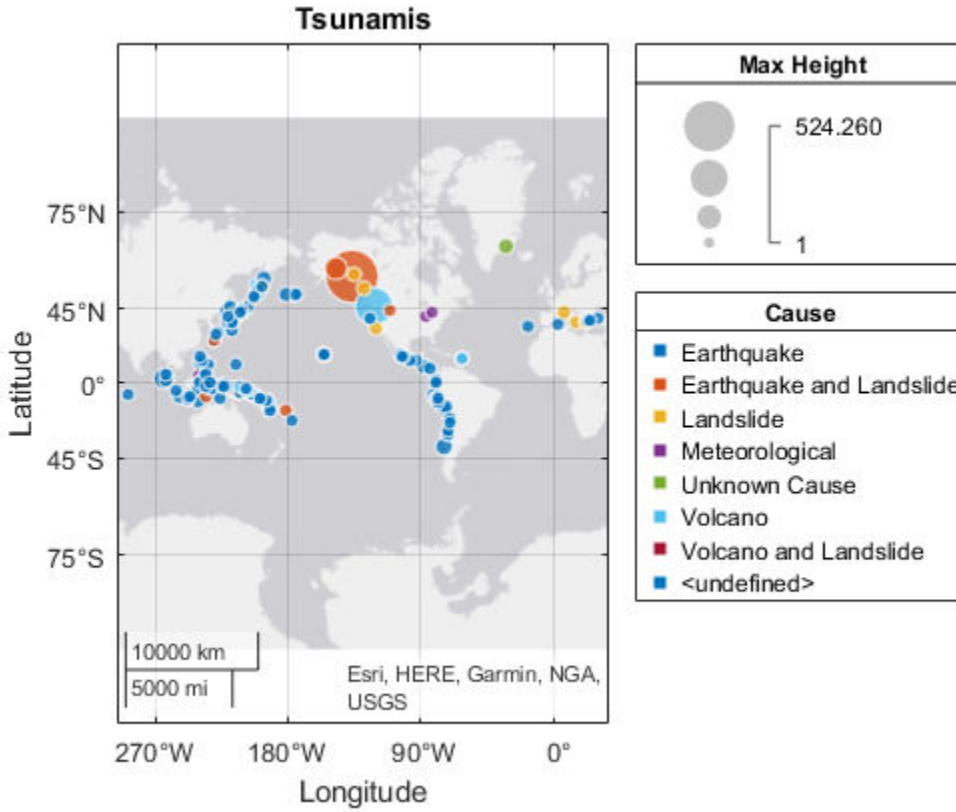
```
gb = geobubble(tsunamis.Latitude,tsunamis.Longitude,tsunamis.MaxHeight,colordata,'Titl
```




```
gb =
  GeographicBubbleChart with properties:

          Basemap: 'streets-light'
        MapLayout: 'normal'
     LatitudeData: [162×1 double]
    LongitudeData: [162×1 double]
         SizeData: [162×1 double]
        ColorData: [162×1 categorical]

  Show all properties
```

Use properties of the geographic bubble chart to modify the chart. For example, add titles to the size and color legends. (If you specify the table as an argument, `geobubble` adds legend titles automatically, using the names of table variables.)

```
gb.SizeLegendTitle = 'Max Height';
gb.ColorLegendTitle = 'Cause'
```




```
gb =
  GeographicBubbleChart with properties:

          Basemap: 'streets-light'
        MapLayout: 'normal'
     LatitudeData: [162×1 double]
    LongitudeData: [162×1 double]
```

```
         SizeData: [162×1 double]
        ColorData: [162×1 categorical]

  Show all properties
```

Input Arguments

`tbl` — Table containing data to be plotted

table | timetable

Table containing data to be plotted, specified as a table or timetable. You can import data as a table using the `readtable` function or create a table from workspace variables using the `table` function. You can create a timetable from workspace variables using the `timetable` function.

The `SourceTable` property of the `GeographicBubbleChart` object stores the table containing data to be plotted.

Data Types: `table`

`latvar` — Table variable for latitude

character vector | string scalar | numeric scalar | logical vector

Table variable for latitude, specified in one of these forms:

- String scalar or character vector specifying one of the table variable names. For example, `geobubble(tbl,'Latitude','Longitude')` selects the variable named `'Latitude'` for `latvar`.
- Numeric scalar indicating the table variable index. For example, `geobubble(tbl,1,2)` selects the first variable in the table for latitude.
- Logical vector containing one `true` element.

The `LatitudeVariable` property of the `GeographicBubbleChart` object stores the selected variable.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string`

`lonvar` — Table variable for longitude

character vector | string scalar | numeric scalar | logical vector

Table variable for longitude, specified in one of these forms:

- String scalar or character vector specifying one of the table variable names. For example, `geobubble(tbl,'Latitude','Longitude')` selects the variable named `'Longitude'` for `lonvar`.
- Numeric scalar indicating the table variable index. For example, `geobubble(tbl,1,2)` selects the second variable in the table for longitude.
- Logical vector containing one `true` element.

The `LongitudeVariable` property of the `GeographicBubbleChart` object stores the selected variable.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string`

`lat` — Latitude coordinates in degrees
real, numeric, finite vector within the range [-90 90]

Latitude coordinates in degrees, specified as a real, numeric, finite vector within the range [-90 90]. The vector can contain embedded `NaN`s. `lat` must be the same size as `lon`.

Example: `[43.0327 38.8921 44.0435]`

Data Types: `single` | `double`

`lon` — Longitude coordinates in degrees
real, numeric, finite vector

Longitude coordinates in degrees, specified as a real, numeric, finite vector. The vector can contain embedded `NaN`s. `lon` must be the same size as `lat`.

Example: `[-107.5556 -77.0269 -72.5565]`

Data Types: `single` | `double`

`sizedata` — Data that determines bubble size
real, numeric, finite vector | real, numeric, finite scalar | `[]`

Data that determines bubble size, specified as a real, numeric, finite vector or scalar, or an empty (`[]`) array. If you specify a vector, `sizedata` must be the same size as `lat` and `lon`. If you specify a scalar value, the geographic bubble chart treats the value with scalar expansion. `sizedata` can contain `NaN`s.

Example: `[99 133 150]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`colordata` — Data categories that determine bubble color
categorical

Data categories that determine bubble color, specified as a categorical variable. The categories are typically bins that data fall into.

Data Types: `categorical`

`parent` — Parent container
`Figure` object | `Panel` object | `Tab` object | `TiledChartLayout` object

Parent container, specified as a `Figure`, `Panel`, `Tab`, or `TiledChartLayout` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `p = geobubble(tbl,latvar,lonvar,'SizeVariable','MaxHeight','ColorVariable','Cause')`

Note The geographic bubble chart properties listed here are frequently used properties. For a complete list, see GeographicBubbleChart Properties.

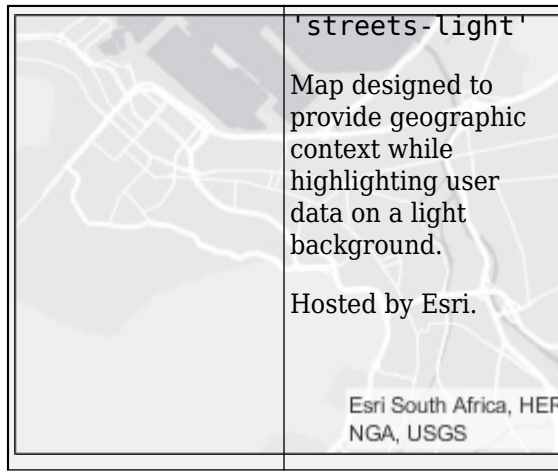
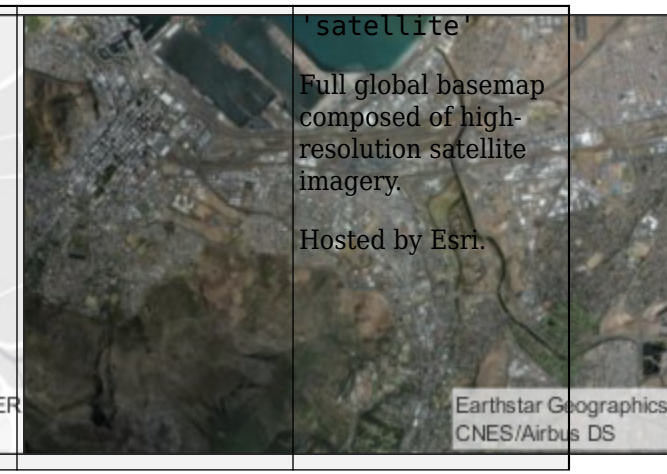
`Basemap` — Map on which to plot data
`'streets-light'` (default) | `'darkwater'` | `'colorterrain'` | `'satellite'` | ...

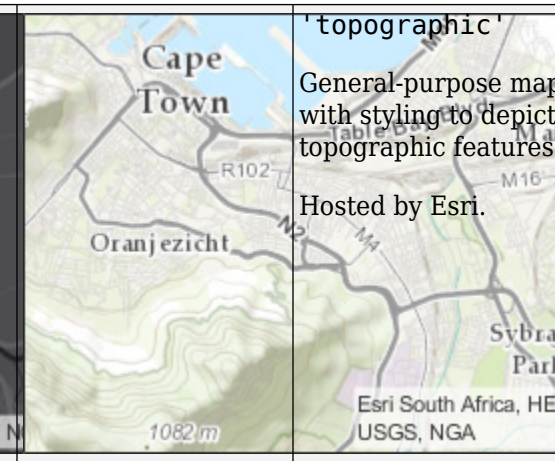
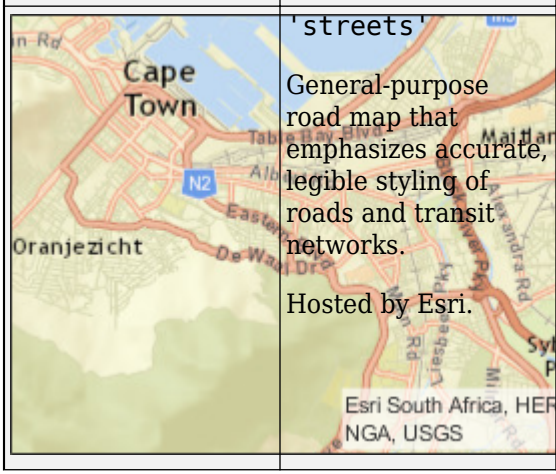
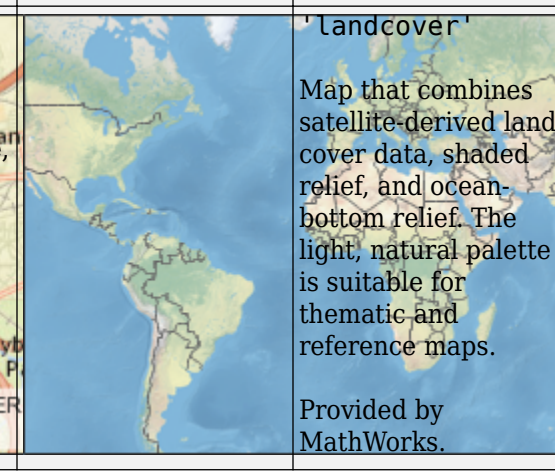
Map on which to plot data, specified as one of the values listed in the table. Six of the basemaps are tiled data sets that MathWorks derives from public-domain data. Five of the basemaps are high-zoom-level maps hosted by Esri.

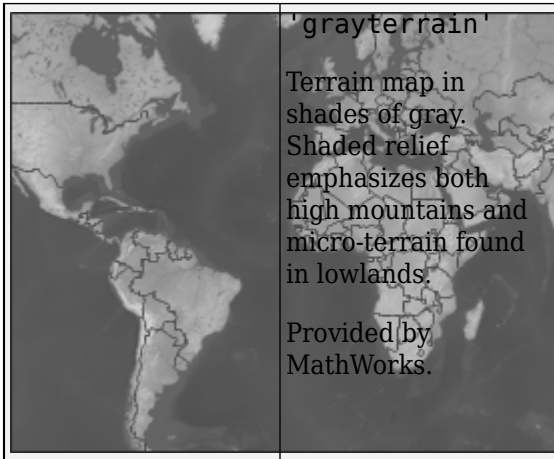
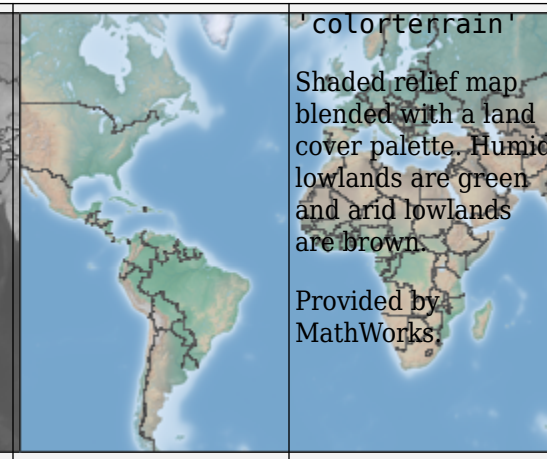
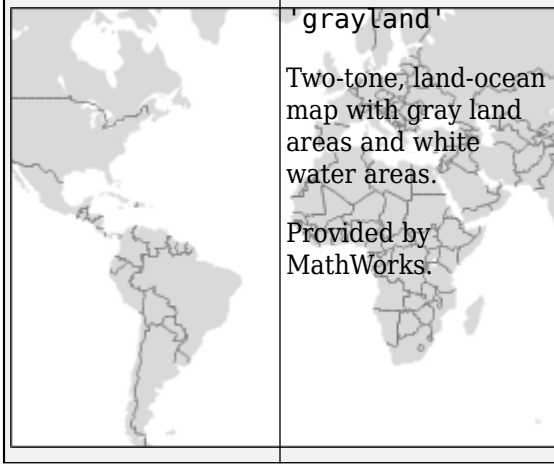
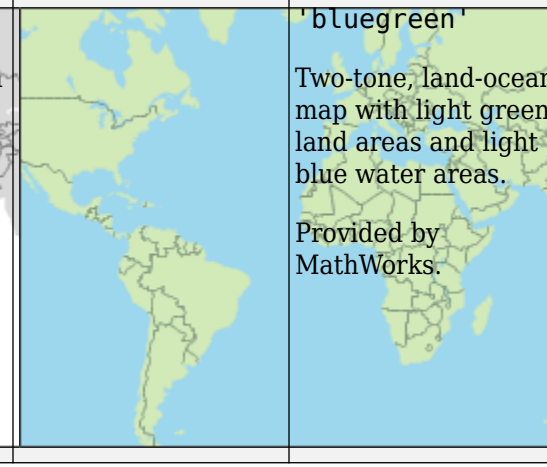
MATLAB includes one installed basemap, a two-tone map named `'darkwater'`. Use of this basemap does not require Internet access. Use of the other basemaps, including the default basemap `'streets-light'`, does require Internet access.

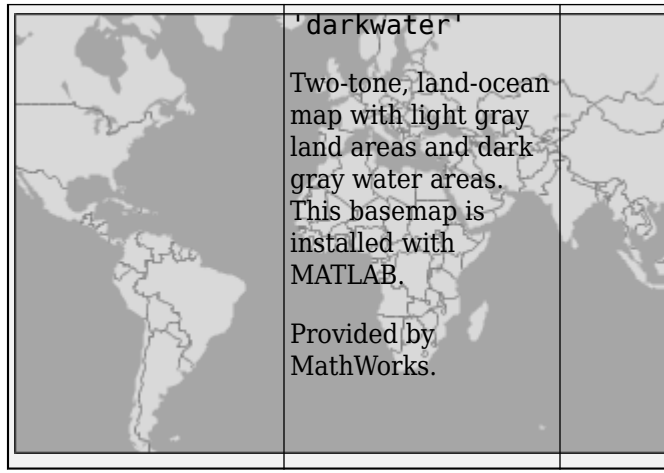
If you do not have consistent access to the Internet, you can download the basemaps hosted by MathWorks onto your local system by using the Add-On Explorer. The five high-zoom-level maps are not available for download. For more about downloading basemaps and changing the default basemap on your local system, see “Access Basemaps in MATLAB”.

The five basemaps hosted by Esri include attributions that give credit to the provider of the map data. This attribution can change as you zoom in on a map because different organizations contribute map data at different zoom levels for different localities. Alignment of boundaries and region labels are a presentation of the feature provided by our data vendors and does not imply endorsement by MathWorks.

	`'streets-light'` Map designed to provide geographic context while highlighting user data on a light background. Hosted by Esri.		`'satellite'` Full global basemap composed of high-resolution satellite imagery. Hosted by Esri.

	`'streets-dark'` Map designed to provide geographic context while highlighting user data on a dark background. Hosted by Esri.		`'topographic'` General-purpose map with styling to depict topographic features. Hosted by Esri.
	`'streets'` General-purpose road map that emphasizes accurate, legible styling of roads and transit networks. Hosted by Esri.		`'landcover'` Map that combines satellite-derived land cover data, shaded relief, and ocean-bottom relief. The light, natural palette is suitable for thematic and reference maps. Provided by MathWorks.

	`'grayterrain'` Terrain map in shades of gray. Shaded relief emphasizes both high mountains and micro-terrain found in lowlands. Provided by MathWorks.		`'colorterrain'` Shaded relief map blended with a land cover palette. Humid lowlands are green and arid lowlands are brown. Provided by MathWorks.
	`'grayland'` Two-tone, land-ocean map with gray land areas and white water areas. Provided by MathWorks.		`'bluegreen'` Two-tone, land-ocean map with light green land areas and light blue water areas. Provided by MathWorks.

	`'darkwater'` Two-tone, land-ocean map with light gray land areas and dark gray water areas. This basemap is installed with MATLAB. Provided by MathWorks.		`'none'` Blank background that plots your data with a latitude-longitude grid, ticks, and labels.

Example: `gx = geoaxes('Basemap','bluegreen')`

Example: `gx.Basemap = 'bluegreen'`

Data Types: `char` | `string`

`ColorVariable` — Table variable used to determine bubble color
character vector | string scalar | numeric scalar | logical vector

Table variable used to determine bubble color, specified in one of these forms:

- A string scalar or character vector specifying the name of the table variable you want to use for color information. For example, `geobubble(__,'ColorVariable','Cause')` specifies the variable named `'Cause'`.
- A numeric scalar indicating the table variable index. For example, `geobubble(__,'ColorVariable',12)` specifies the 12th variable in the table.
- A logical vector containing one `true` element. For example, `sizevar = logical([0 0 0 0 0 0 0 0 0 0 0 1])` specifies the 12th variable in the table.

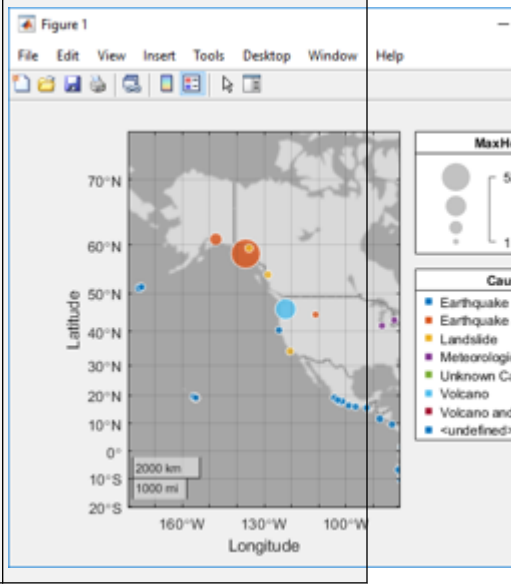
You can use this property only when specifying a table as input. The values associated with this table variable must be categorical. When you specify the color variable, `geobubble` stores the data values associated with this variable in the `ColorData` property and sets the `ColorData` property to read-only.

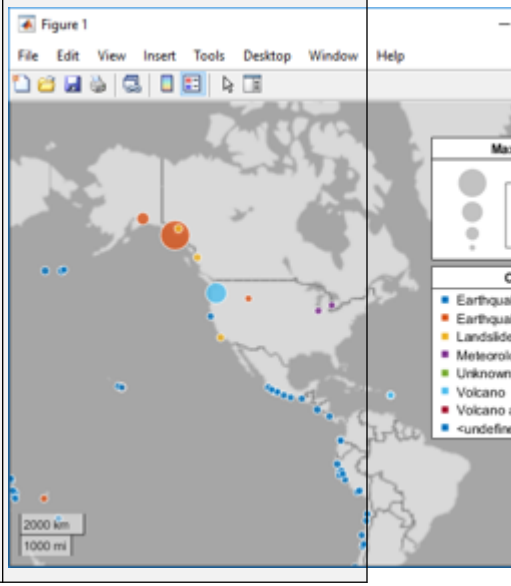
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string`

`MapLayout` — Layout of map

`'normal'` (default) | `'maximized'`

Layout of map, including insets and decorations, specified as either of the following.

Value	Description	Illustration
`'normal'`	Map is inset from the edges of the chart, as defined by its `OuterPosition` property. Axes labels (`'Latitude'` and `'Longitude'`), ticks, and tick labels are visible. If the `Title` property value is set, the chart includes a title. Legends, if present, appear outside and to the right of the map.	

Value	Description	Illustration
`'maximized'`	Map fills the entire space, defined by the `OuterPosition` property. Axes labels, ticks, and tick labels are hidden. The title is hidden, even if the `Title` property is set. The grid is hidden, even if `GridVisible` is set to `'on'`. Legends, if present, appear within the map, toward the upper-right corner.	

Example: `gb = geobubble(__,'MapLayout','maximized')`

Example: `gb.MapLayout = 'maximized'`

Data Types: `char` | `string`

`SizeVariable` — Table variable used to determine bubble size

character vector | string scalar | numeric scalar | logical vector

Table variable used to determine bubble size, specified in one of these forms:

- A string scalar or character vector specifying the name of the table variable you want to use for size information. For example, `geobubble(__,'SizeVariable','MaxHeight')` specifies the variable named `'MaxHeight'`.
- A numeric scalar indicating the table variable index. For example, `geobubble(__,'SizeVariable',16)` specifies the sixteenth variable in the table.
- A logical vector containing one `true` element. For example, `sizevar = logical([0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1])` specifies the 16th variable in the table.

This property can only be used when specifying a table as input. The values associated with this table variable must be of a numeric type. When you specify this variable, `geobubble` stores the data values associated with this variable in the `'SizeData'` property and sets the property to read-only.

Output Arguments

gb — Geographic bubble chart
`GeographicBubbleChart` object

Geographic bubble chart, returned as a `GeographicBubbleChart` object.

Tips

- To bring focus to a geographic bubble chart programmatically, use the `axes` function, `axes(gb)`.

See Also

Functions
`categorical` | `geobasemap` | `geodensityplot` | `geolimits` | `geoplot` | `geoscatter` | `readtable` | `table`

Properties
GeographicBubbleChart Properties

Topics

"Access Data in Tables"
"Create Geographic Bubble Chart from Tabular Data"
"Use Geographic Bubble Chart Properties"
"Deploy Geographic Axes and Charts"
"Access Basemaps in MATLAB"
"Troubleshoot Geographic Axes or Chart Basemap Connection"
"Geographic Bubble Charts Overview"

Introduced in R2017b

geodensityplot

Geographic density plot

Syntax

```
geodensityplot(lat,lon)
geodensityplot(lat,lon,weights)
geodensityplot( ___ ,Name,Value)
geodensityplot(gx, ___ )
dp = geodensityplot( ___ )
```

Description

`geodensityplot(lat,lon)` creates a density plot in a geographic axes from locations specified (in degrees) by the coordinate vectors `lat` and `lon`. `lat` and `lon` must be the same size.

`geodensityplot(lat,lon,weights)` specifies weights for the data points in `weights`.

`geodensityplot( ___ ,Name,Value)` specifies `DensityPlot` properties using one or more Name,Value pair arguments.

`geodensityplot(gx, ___ )` plots in the geographic axes specified by `gx` instead of the current axes.`gx`.

`dp = geodensityplot( ___ )` returns a `DensityPlot` object.

Examples

Create Geographic Density Plot

Set up latitude and longitude data.

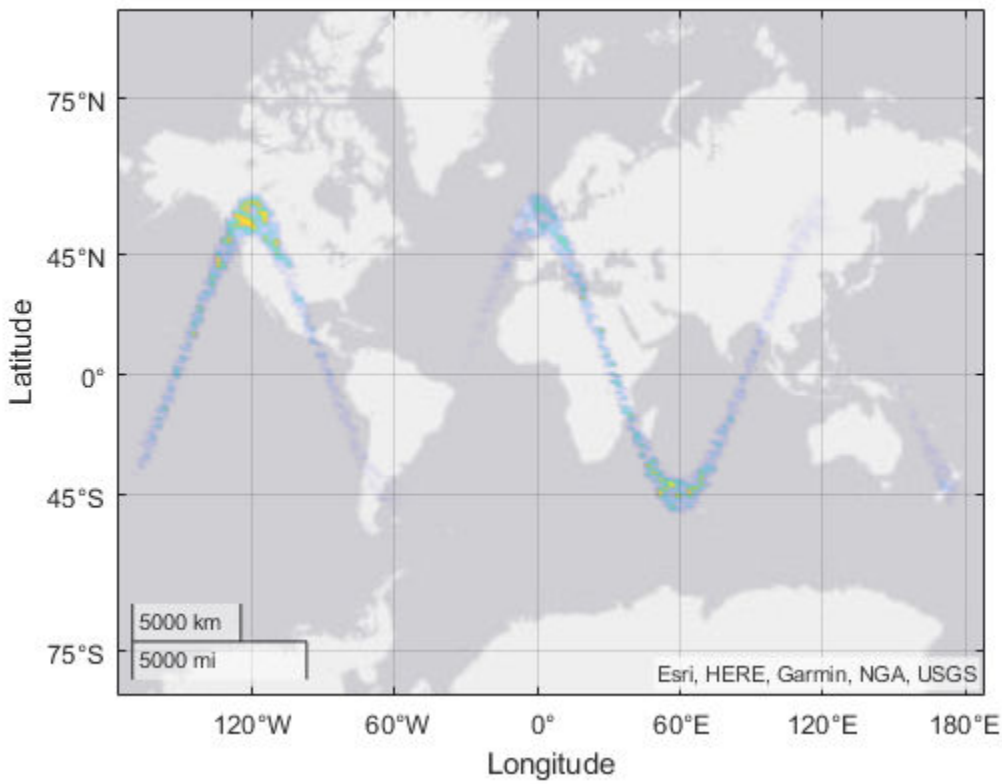
```
lon = linspace(-170,170,3000) + 10*rand(1,3000);
lat = 50 * cosd(3*lon) + 10*rand(size(lon));
```

Specify weights for each data point.

```
weights = 101 + 100*(sind(2*lon));
```

Create the geographic density plot, specifying the colors used with the plot.

```
geodensityplot(lat,lon,weights,'FaceColor','interp')
```



Input Arguments

`lat` — Latitude coordinates of data points

real, numeric, finite vector within the range [-90 90]

Latitude coordinates of data points, specified as a real, numeric, finite vector in degrees, within the range [-90 90]. The vector can contain embedded `NaN`s. `lat` must be the same size as `lon`.

Example: `[43.0327 38.8921 44.0435]`

Data Types: `single` | `double`

`lon` — Longitude coordinates in degrees

real, numeric, finite vector

Longitude coordinates in degrees, specified as a real, numeric, finite vector. The vector can contain embedded `NaN`s. `lon` must be the same size as `lat`.

Longitudes must span less than (or equal to) 360 degrees.

Example: `[-107.5556 -77.0269 -72.5565]`

Data Types: `single` | `double`

`weights` — Weights assigned to data

`[]` (default) | scalar | numeric vector

Weights assigned to data, specified as an empty array, a scalar, or a numeric array. If you specify a numeric vector, the vector must be the same length as the `lat` and `lon` vectors. The `weights` vector is typically additional data you have related to the location data in `lat` and `lon`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`gx` — Target geographic axes

`GeographicAxes` object

Target geographic axes, specified as a `GeographicAxes` object. You can modify the appearance and behavior of a `GeographicAxes` object by setting its properties. For a list of properties, see GeographicAxes Properties.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `dp = geodensityplot(lat,lon,weights,'FaceColor','g')`

Note The density plot properties listed here are frequently used properties. For a complete list, see DensityPlot Properties.

`FaceAlpha` — Face transparency

`'interp'` (default) | scalar in range `[0,1]`

Face transparency, specified as one of these values:

- Scalar in range `[0,1]` — Use uniform transparency across all the faces. A value of `1` is fully opaque and `0` is completely transparent. Values between `0` and `1` are semitransparent.
- `'interp'` — Use interpolated transparency for each face. The transparency varies across each face by interpolating the values at the vertices. Transparency values are chosen from the parent axes alphamap.

`FaceColor` — Face color

`'interp'` | `ColorSpec` value | RGB triplet | ...

Face color, specified as one of the values in this table.

Value	Description
`'interp'`	Use interpolated coloring based on density values. Colors are chosen from the parent axes colormap.
RGB triplet	Three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`.

Value	Description
`ColorSpec`	MATLAB`ColorSpec (Color Specification)`. You can specify some common colors by name.

`Radius` — Radius of influence each point has on density calculation
numeric scalar

Radius of influence each point has on density calculation, specified as a numeric scalar. When used in a `GeographicAxes`, the value is measured in meters.

Output Arguments

`dp` — Geographic density plot
`DensityPlot`

Geographic density plot, returned as a `DensityPlot` object.

Tips

- To customize the geographic axes, modify geographic axes properties. For a list of properties, see GeographicAxes Properties.
- if you have Mapping Toolbox, you can specify basemaps of your own choosing using the `addCustomBasemap` function.

See Also

DensityPlot Properties | `alphamap` | `colormap` | `geoaxes`

Topics

"View Cyclone Track Data in Geographic Density Plot"
"View Density of Cellular Tower Placement"

Introduced in R2018b

GeographicBubbleChart Properties

Control geographic bubble chart appearance and behavior

Description

`GeographicBubbleChart` properties control the appearance and behavior of a `GeographicBubbleChart` object. By changing property values, you can modify aspects of the chart display. Use dot notation to refer to a particular object and property. The following example specifies the name of the size legend by using the `SizeLegendTitle` property.

```
tsunamis = readtable('tsunamis.xlsx');
tsunamis.Cause = categorical(tsunamis.Cause);
figure
gb = geobubble(tsunamis,'Latitude','Longitude', ...
        'SizeVariable','MaxHeight','ColorVariable','Cause', ...
        'Basemap','colorterrain')
geolimits([10 65],[-180 -80])
title 'Tsunamis in North America';
gb.SizeLegendTitle = 'Maximum Height';
```

Properties

Bubble Location

`LatitudeData` — Latitude coordinates of bubble locations

real, finite, numeric vector | [ ]

Latitude coordinates of bubble locations, specified as a real, finite, numeric vector of values in the range [-90,90], or as an empty (`[]`) array. `LatitudeData` must be the same size as `LongitudeData` and can contain `NaN`s.

Bubbles with latitudes outside the approximate limits [-85 85], beyond which the basemap tiles do not extend, are permissible. However these values are not typically seen unless the map extent is controlled manually using the `MapCenter` and `ZoomLevel` properties. Also, bubbles very close to 90 degrees and -90 degrees can never be seen, because they map to infinite or near-infinite *y*-values.

Data Types: `single` | `double`

`LatitudeVariable` — Table variable used for bubble latitude
character vector | string scalar | numeric scalar | logical vector

Table variable used for bubble latitude, specified in one of these forms:

- A string scalar or character vector specifying the name of the table variable you want to use for latitude. For example, `geobubble(__,'LatitudeVariable','Latitude')` specifies the variable named `'Latitude'`.
- A numeric scalar indicating the table variable index. For example, `geobubble(__,'LatitudeVariable',1)` specifies the first variable in the table for latitudes.
- A logical vector containing one `true` element.

The values associated with this table variable must be numeric. You can use this property only when specifying a table as input. `geobubble` stores the value of this variable in the `'LatitudeData'` property and sets the `'LatitudeData'` property to read-only.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string`

`LongitudeData` — Longitude coordinates of bubble locations
real, finite numeric vector | `[]`

Longitude coordinates of bubble locations, specified as a real, finite, numeric vector of values in the range `(-Inf,Inf)`, or as an empty (`[]`) array. `LongitudeData` must be the same size as `LatitudeData` and can contain `NaN`s.

Data Types: `single` | `double`

`LongitudeVariable` — Table variable used for bubble longitude
character vector | string scalar | numeric scalar | logical vector

Table variable used for bubble longitude, specified in one of these forms:

- A string or character vector specifying the name of the table variable you want to use for longitude information. For example, `geobubble(__,'LongitudeVariable','Longitude')` specifies the table variable named `'Longitude'`.

- A numeric scalar indicating the table variable index. For example, `geobubble(__,'LongitudeVariable',16)` specifies the sixteenth variable in the table for longitudes.
- A logical vector containing one `true` element.

The values associated with this table variable must be numeric. You can use this property only when specifying a table as input. `geobubble` stores the value of this variable in the `'LongitudeData'` property and sets the `'LongitudeData'` property to read-only.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string`

Bubble Size

`BubbleWidthRange` — Minimum and maximum width of bubbles

`[5 20]` (default) | numeric scalar | 1-by-2 numeric vector

Minimum and maximum width of bubbles, measured in points, specified as a numeric scalar or a 1-by-2 numeric vector. Values must be non-descending. Use a scalar when you want all the bubbles to have the same (uniform) size. Values must fall within the range [1 100].

Example: `[4 10]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`SizeData` — Data controlling bubble size

numeric vector | numeric scalar | `[]`

Data controlling bubble size, specified as a numeric vector or scalar in the range `(-Inf,Inf)`, or as an empty (`[]`) array. If you specify a vector, `SizeData` must be the same size as `LatitudeData` and `LongitudeData`. If you specify a scalar value, the geographic bubble chart treats the value with scalar expansion. `sizedata` can contain `NaN`s.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`SizeLimits` — Limits for mapping `SizeData` values to bubble width

1-by-2 vector of real, finite values

Limits for mapping `SizeData` values to bubble width, specified as a 1-by-2 vector of real, finite, numeric values, or as an empty (`[]`) matrix. Values must be non-descending. To create bubbles that are all the same size, specify the same value for each element.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`SizeVariable` — Table variable used to determine bubble size
character vector | string scalar | numeric scalar | logical vector

Table variable used to determine bubble size, specified in one of these forms:

- A string scalar or character vector specifying the name of the table variable you want to use for size information. For example, `geobubble(__,'SizeVariable','MaxHeight')` specifies the variable named `'MaxHeight'`.
- A numeric scalar indicating the table variable index. For example, `geobubble(__,'SizeVariable',16)` specifies the sixteenth variable in the table.
- A logical vector containing one `true` element. For example, `sizevar = logical([0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1])` specifies the 16th variable in the table.

This property can only be used when specifying a table as input. The values associated with this table variable must be of a numeric type. When you specify this variable, `geobubble` stores the data values associated with this variable in the `'SizeData'` property and sets the property to read-only.

Bubble Color

`BubbleColorList` — Colors used to display bubbles and color code by category
MATLAB color specification | cell array of strings | string vector of colors | *m*-by-3 RGB color matrix

The `BubbleColorList` property controls the colors used for the bubbles. The value is an *m*-by-3 array where each row is an RGB color triplet, where *m* equals the number categories in the `ColorData` vector, or the number of categories plus 1 if any elements of `ColorData` are undefined, or `1` if `ColorData` is empty. By default, `geobubble` selects colors from an ordered list of 7 standard colors. If *m* is greater than 7, the colors repeat cyclically. To change the colors used, use one of the MATLAB colormap functions, such as `parula` or `jet`, or specify a custom list of your own RGB values.

Data Types: `cell` | `double`

`ColorData` — Data controlling bubble color
categorical vector | `[]`

Data controlling bubble color, specified as a categorical vector or as empty array (`[]`). Bubbles assigned to the same category have the same color on the map. The geographic

bubble chart assigns a color to each category, using the colors listed in the `BubbleColorList` property. The size of `ColorData` must match `LatitudeData` and `LongitudeData`, except when you specify an empty array.

If you use a color legend, the geographic bubble chart displays the category values in the legend. If any of the values contain TeX markup characters, such as an underscore (_), you might see unexpected formatting in your color legend. MATLAB® uses a subset of TeX markup for the text displayed in legends. To use a TeX markup character in regular text, edit the name of the category (using `renamecats`) and insert the TeX escape character, the backslash (\), before the character you want to include. For information about using TeX markup to add superscripts and subscripts, modify font type and color, and include special characters in the text, see the "Interpreter" on page 1-0 property of the text object.

Data Types: `categorical`

`ColorVariable` — Table variable used to determine bubble color
character vector | string scalar | numeric scalar | logical vector

Table variable used to determine bubble color, specified in one of these forms:

- A string scalar or character vector specifying the name of the table variable you want to use for color information. For example, `geobubble(__,'ColorVariable','Cause')` specifies the variable named `'Cause'`.
- A numeric scalar indicating the table variable index. For example, `geobubble(__,'ColorVariable',12)` specifies the 12th variable in the table.
- A logical vector containing one `true` element. For example, `sizevar = logical([0 0 0 0 0 0 0 0 0 0 0 1])` specifies the 12th variable in the table.

You can use this property only when specifying a table as input. The values associated with this table variable must be categorical. When you specify the color variable, `geobubble` stores the data values associated with this variable in the `ColorData` property and sets the `ColorData` property to read-only.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string`

Labels

`Title` — Title of geographic bubble chart

`''` (default) | character array | cell array of character vectors | string scalar | string array | numeric value | `categorical`

Title of geographic bubble chart, specified as a character vector, cell array of character vectors, scalar string, string array, numeric value, or a `categorical` value. If you specify this property as a `categorical` array, MATLAB uses the values in the array, not the categories. You can also use the `title` function to set this value.

By default, MATLAB® supports a subset of TeX markup for the text you specify. To add superscripts and subscripts, modify the font type and color, and include special characters in the text, use TeX markup. To use a TeX markup character in regular text, such as an underscore (_), insert the TeX escape character the backslash (\), before the character you want to include. For more information, see the “Interpreter” on page 1-0 property of the text object.

`ColorLegendTitle` — Text to display as title of color legend

`''` (default) | character vector | string scalar | string array | cell array of character vectors | numeric value | `categorical`

Text to display as title of color legend, specified as a character vector, string scalar, string array, cell array of character vectors, numeric value, or `categorical` value. If you specify this property as a `categorical` array, MATLAB uses the values in the array, not the categories.

By default, MATLAB® supports a subset of TeX markup for the text you specify. To add superscripts and subscripts, modify the font type and color, and include special characters in the text, use TeX markup. To use a TeX markup character in regular text, such as an underscore (_), insert the TeX escape character, the backslash (\), before the character you want to include. For more information, see the “Interpreter” on page 1-0 property of the text object.

Data Types: `char` | `cell` | `string` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`SizeLegendTitle` — Size legend title

`''` (default) | character vector | string scalar | string array | cell array of character vectors | numeric value | `categorical`

Size legend title, specified as a character vector, string scalar, string array, cell array of character vectors, numeric value, or `categorical` value. If you specify this property as a `categorical` array, MATLAB uses the values in the array, not the categories.

By default, MATLAB® supports a subset of TeX markup for the text you specify. To add superscripts and subscripts, modify the font type and color, and include special characters in the text, use TeX markup. To use a TeX markup character in regular text, such as an underscore (_), insert the TeX escape character the backslash (\), before the character you want to include. For more information, see the "Interpreter" on page 1-0 property of the text object.

`LegendVisible` — Visibility of bubble size and color legends

`'on'` (default) | `'off'` | `true` | `false`

Visibility of bubble size and color legends, specified as `'on'` or `'off'` or the logical values `true` or `false`. You can also toggle the visibility of the legends by using the `legend` function.

Data Types: `char` | `string` | `logical`

Font

`FontName` — Font used in the geographic bubble chart

character vector | string scalar | `'FixedWidth'`

Font used in the geographic bubble chart, specified as a string scalar or character vector. To display and print properly, the font name must be a font that your system supports. The default font depends on the specific operating system and locale. To use a fixed-width font that looks good in any locale, use `'FixedWidth'`. The `'FixedWidth'` value relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

Example: `'Cambria'`

Data Types: `char` | `string`

`FontSize` — Font size used in geographic bubble chart

system dependent (default) | real, finite, positive, numeric scalar

Font size used in geographic bubble chart, specified as a real, finite, positive, numeric scalar. The value is in point units, where one point equals 1/72 inches.

Map

Basemap — Map on which to plot data
`'streets-light'` (default) | `'darkwater'` | `'colorterrain'` | `'satellite'` | ...

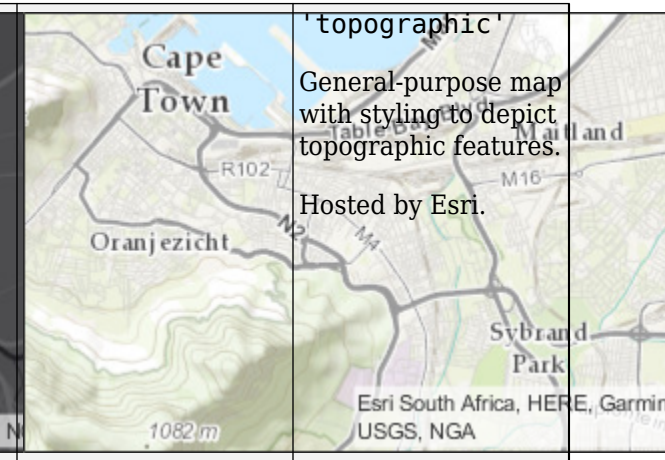
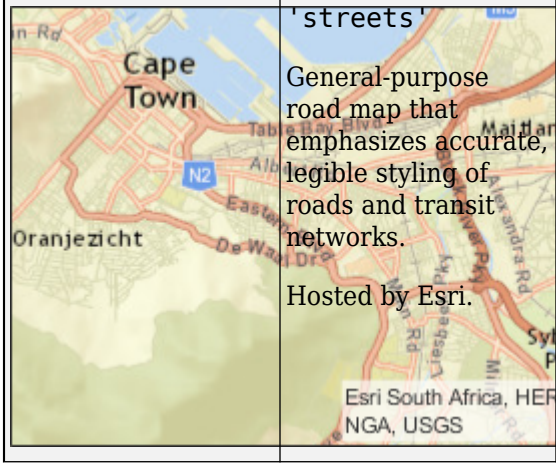
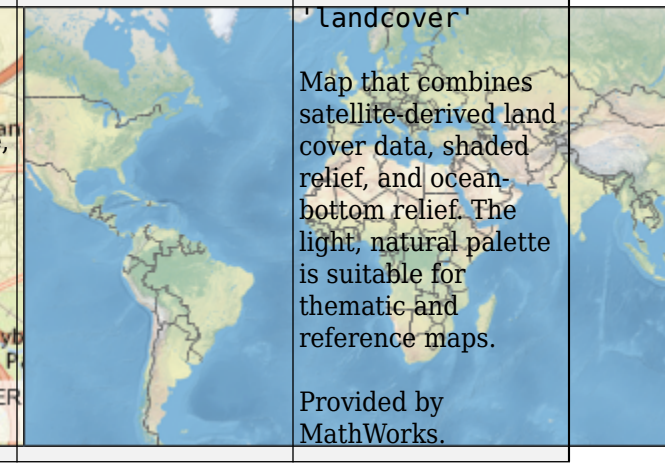
Map on which to plot data, specified as one of the values listed in the table. Six of the basemaps are tiled data sets that MathWorks derives from public-domain data. Five of the basemaps are high-zoom-level maps hosted by Esri.

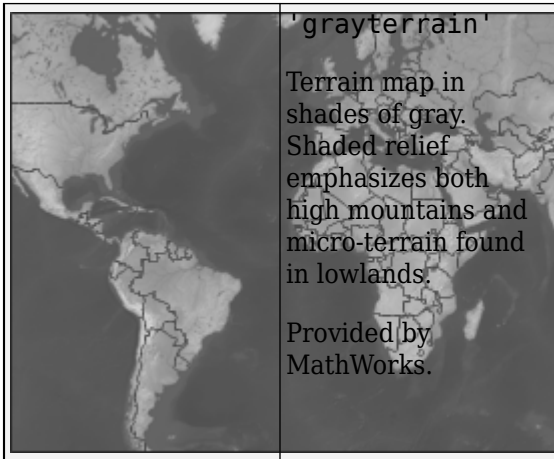
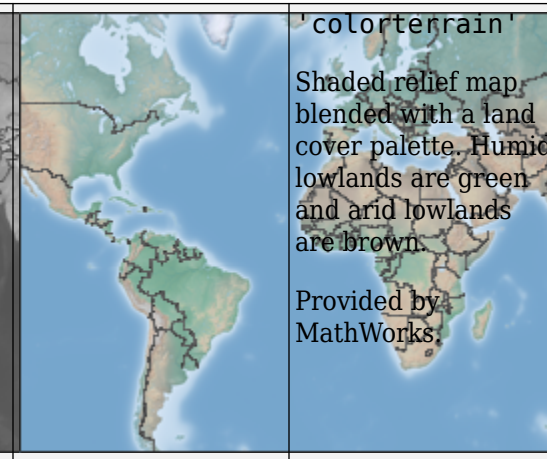
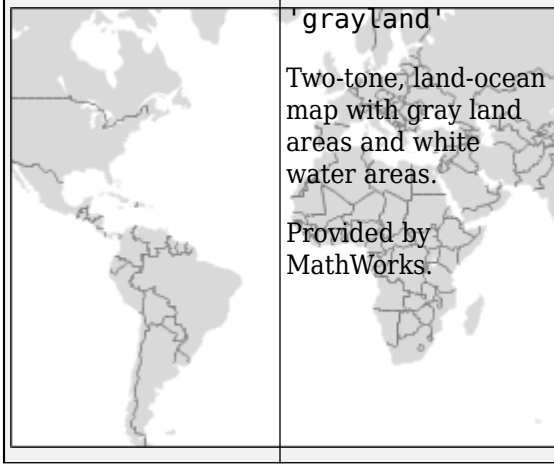
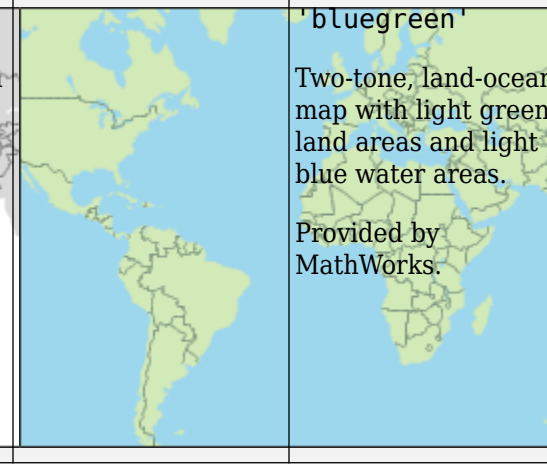
MATLAB includes one installed basemap, a two-tone map named `'darkwater'`. Use of this basemap does not require Internet access. Use of the other basemaps, including the default basemap `'streets-light'`, does require Internet access.

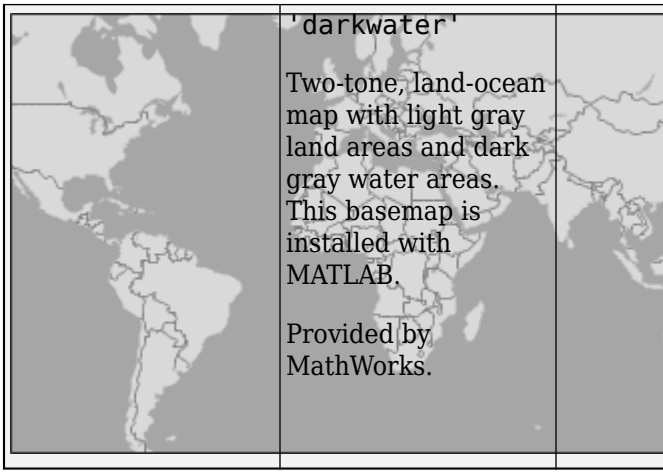
If you do not have consistent access to the Internet, you can download the basemaps hosted by MathWorks onto your local system by using the Add-On Explorer. The five high-zoom-level maps are not available for download. For more about downloading basemaps and changing the default basemap on your local system, see “Access Basemaps in MATLAB”.

The five basemaps hosted by Esri include attributions that give credit to the provider of the map data. This attribution can change as you zoom in on a map because different organizations contribute map data at different zoom levels for different localities. Alignment of boundaries and region labels are a presentation of the feature provided by our data vendors and does not imply endorsement by MathWorks.

	`'streets-light'` Map designed to provide geographic context while highlighting user data on a light background. Hosted by Esri.		`'satellite'` Full global basemap composed of high-resolution satellite imagery. Hosted by Esri.

	`'streets-dark'` Map designed to provide geographic context while highlighting user data on a dark background. Hosted by Esri.		`'topographic'` General-purpose map with styling to depict topographic features. Hosted by Esri.
	`'streets'` General-purpose road map that emphasizes accurate, legible styling of roads and transit networks. Hosted by Esri.		`'landcover'` Map that combines satellite-derived land cover data, shaded relief, and ocean-bottom relief. The light, natural palette is suitable for thematic and reference maps. Provided by MathWorks.

	`'grayterrain'` Terrain map in shades of gray. Shaded relief emphasizes both high mountains and micro-terrain found in lowlands. Provided by MathWorks.		`'colorterrain'` Shaded relief map blended with a land cover palette. Humid lowlands are green and arid lowlands are brown. Provided by MathWorks.
	`'grayland'` Two-tone, land-ocean map with gray land areas and white water areas. Provided by MathWorks.		`'bluegreen'` Two-tone, land-ocean map with light green land areas and light blue water areas. Provided by MathWorks.

	`'darkwater'` Two-tone, land-ocean map with light gray land areas and dark gray water areas. This basemap is installed with MATLAB. Provided by MathWorks.		`'none'` Blank background that plots your data with a latitude-longitude grid, ticks, and labels.

Example: `gx = geoaxes('Basemap','bluegreen')`

Example: `gx.Basemap = 'bluegreen'`

Data Types: `char` | `string`

`SourceTable` — Table containing data to be plotted
table

Table containing data to be plotted, specified as a table.

Data Types: `table`

`GridVisible` — Visibility of the latitude and longitude lines on the map
`'on'` (default) | `'off'` | `true` | `false`

Visibility of the latitude and longitude lines on the map, specified as `'on'` or `'off'`, or the logical values `true` or `false`. You can also use the `grid` function to toggle grid visibility.

Data Types: `logical` | `char` | `string`

`LatitudeLimits` — Latitude limits of map
1-by-2 vector of real, finite values

This property is read-only.

Latitude limits of map, specified as a 1-by-2 vector of real, finite values of the form `[southern_limit northern_limit]` in the range [-90,90]. To set latitude limits use the `geolimits` function.

Data Types: `double`

`LongitudeLimits` — Longitude limits of map
1-by-2 vector of real, finite values

This property is read-only.

Longitude limits of map, specified as a 1-by-2 vector of real, finite values of the form `[western_limit eastern_limit]`. Values must be in the range `(-Inf, Inf)`. To set longitude limits use the `geolimits` function.

Example: `[-100 100]`

Data Types: `double`

`MapCenter` — Center point of map in latitude and longitude
two-element numeric vector of real, finite values

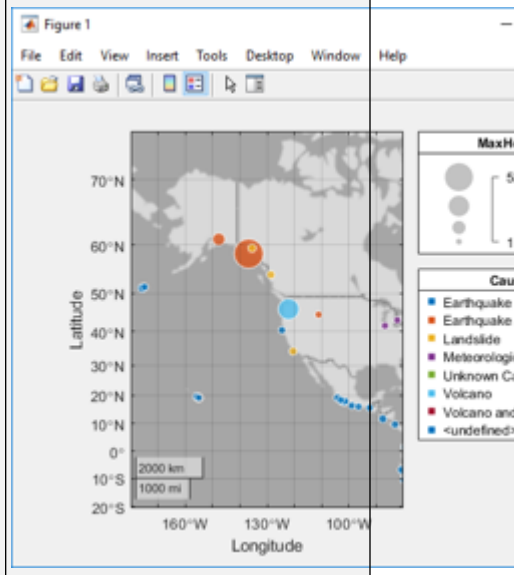
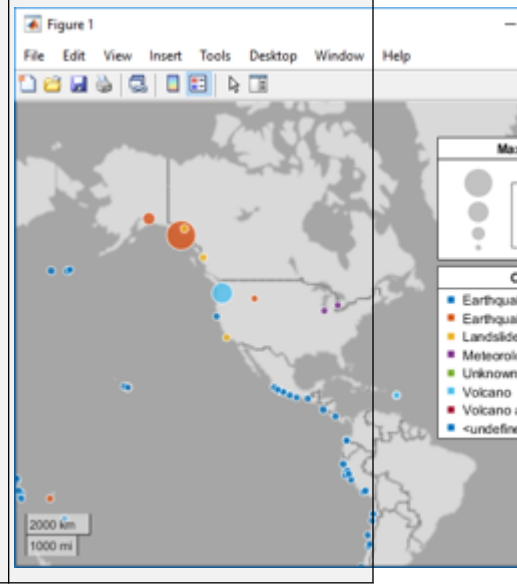
Center point of map in latitude and longitude, specified as a two-element vector of real, finite values of the form `[center_latitude center_longitude]`. Values must be in the range `[(-90,90),(-Inf, Inf)]`.

Example: `[38.6292 -95.2520]`

Data Types: `single` | `double`

`MapLayout` — Layout of map
`'normal'` (default) | `'maximized'`

Layout of map, including insets and decorations, specified as either of the following.

Value	Description	Illustration
`'normal'`	Map is inset from the edges of the chart, as defined by its `OuterPosition` property. Axes labels (`'Latitude'` and `'Longitude'`), ticks, and tick labels are visible. If the `Title` property value is set, the chart includes a title. Legends, if present, appear outside and to the right of the map.	
`'maximized'`	Map fills the entire space, defined by the `OuterPosition` property. Axes labels, ticks, and tick labels are hidden. The title is hidden, even if the `Title` property is set. The grid is hidden, even if `GridVisible` is set to `'on'`. Legends, if present, appear within the map, toward the upper-right corner.	

Example: `gb = geobubble(__,'MapLayout','maximized')`

Example: `gb.MapLayout = 'maximized'`

Data Types: `char` | `string`

`ScalebarVisible` — Visibility of the scale bar on the map
`'on'` (default) | `'off'` | `true` | `false`

Visibility of the scale bar on the map, specified as `'on'` or `'off'`, or the logical values `true` or `false`.

Data Types: `logical` | `char` | `string`

`ZoomLevel` — Magnification level of map
real, finite, numeric scalar between 0 and 25, inclusive

Magnification level of map, specified as a real, finite, numeric scalar between 0 and 25, inclusive. The value is a base 2 logarithmic map scale. Increasing the `ZoomLevel` value by 1 doubles the map scale.

Data Types: `single` | `double`

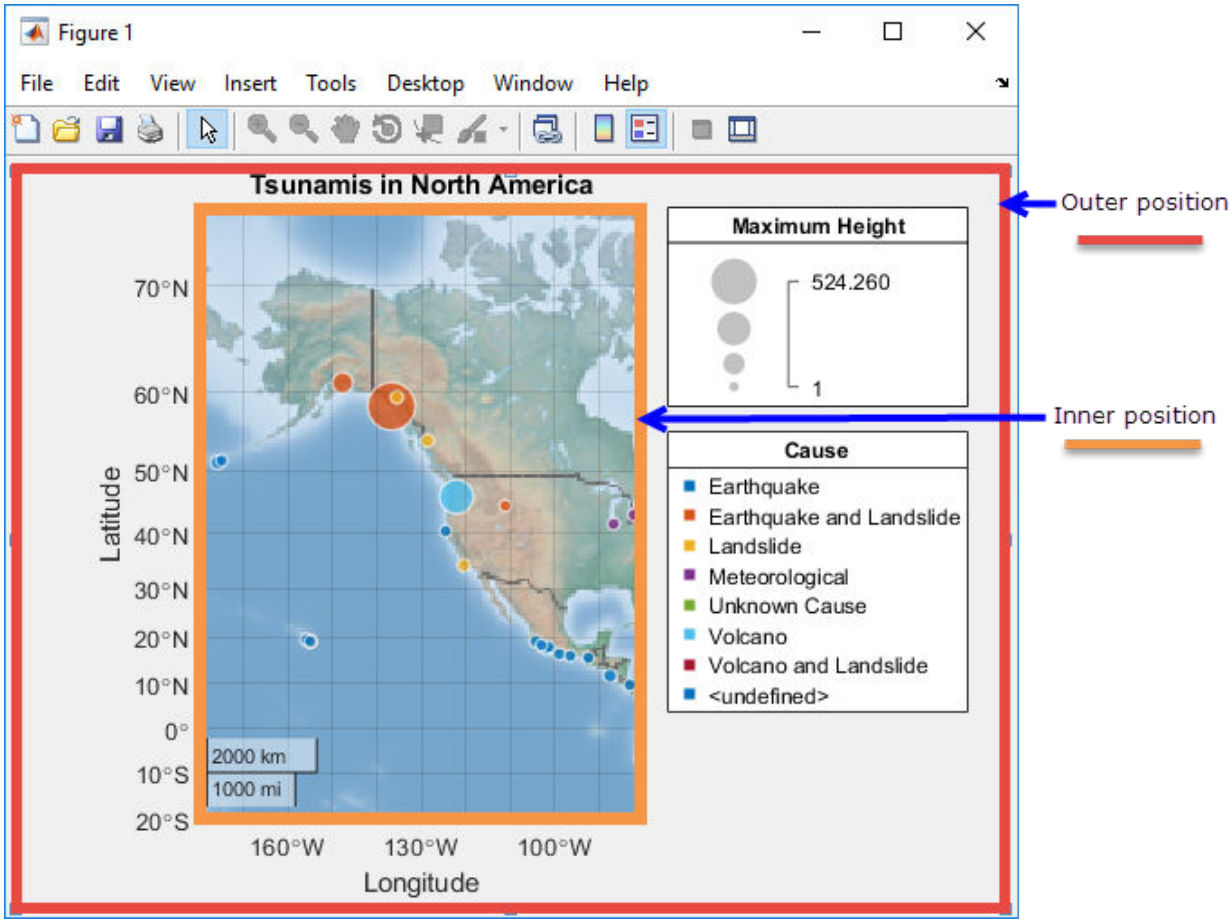
Position

`ActivePositionProperty` — Position property to hold constant
`'outerposition'` | `'innerposition'`

Position property to hold constant when adding, removing, or changing decorations, specified as one of the following values:

- `'outerposition'` — The `OuterPosition` property remains constant when you add, remove, or change decorations such as a title or an axis label. If any positional adjustments are needed, MATLAB adjusts the `InnerPosition` property.
- `'innerposition'` — The `InnerPosition` property remains constant when you add, remove, or change decorations such as a title or an axis label. If any positional adjustments are needed, MATLAB adjusts the `OuterPosition` property.

The following figure shows the `innerposition` and `outerposition` definitions for a geographic bubble chart. `innerposition` does not include the title or axis labels.



Example: `gb.ActivePositionProperty = 'outerposition'`

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`InnerPosition` — Inner size and position

four-element vector

Inner size and position of the geographic bubble chart within the parent container (typically a figure, panel, or tab), returned as a four-element vector of the form `[left bottom width height]`. The inner position does not include the title or axis labels.

- The `left` and `bottom` elements define the distance from the lower left corner of the container to the lower left corner of the geographic bubble chart.
- The `width` and `height` elements are the geographic bubble chart dimensions.

For an illustration, see “ActivePositionProperty” on page 1-0 .

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`OuterPosition` — Size and position of geographic bubble chart within parent

`[0 0 1 1]` (default) | four-element vector

Size and position of the geographic bubble chart within its parent, specified as a four-element numeric vector of the form `[left bottom width height]`. The default value of `[0 0 1 1]` includes the whole interior of the container.

For an illustration, see “ActivePositionProperty” on page 1-0 .

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`Position` — Inner size and position

four-element vector

Inner size and position of the geographic bubble chart within the parent container (typically a figure, panel or tab) returned as a four-element vector of the form `[left bottom width height]`. This property is equivalent to the “InnerPosition” on page 1-0 property.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

Units — Position units
`'normalized'` (default) | `'inches'` | `'centimeters'` | `'points'` | `'pixels'` | `'characters'`

Position units, specified as one of these values.

Units	Description
`'normalized'` (default)	Normalized with respect to the container, which is typically the figure or a panel. The lower left corner of the container maps to `(0,0)`, and the upper right corner maps to `(1,1)`.
`'inches'`	Inches
`'centimeters'`	Centimeters
`'characters'`	Based on the default uicontrol font of the graphics root object: • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.
`'points'`	Typography points. One point equals 1/72 inch.
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.

When specifying the units as a name-value pair during object creation, you must set the `Units` property before specifying the properties that you want to use these units, such as `OuterPosition`.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `TiledChartLayoutOptions` object

Layout options, specified as a `TiledChartLayoutOptions` object. This property specifies options when the chart is a child of a tiled chart layout. Use this property to position the chart within the layout by setting the `Tile` and `TileSpan` properties on the `TiledChartLayoutOptions` object.

For example, this code places chart `c` in the third tile of a tiled chart layout.

```
c.Layout.Tile = 3;
```

To make the chart span multiple tiles, specify the `TileSpan` property as a two-element vector. For example, this chart spans `2` rows and `3` columns of tiles.

```
c.Layout.TileSpan = [2 3];
```

If the chart is not a child of a tiled chart layout (for example, if it is a child of a figure or panel), then this property is empty and has no effect.

`Visible` — Visibility of the geographic bubble chart
`'on'` (default) | `'off'` | `true` | `false`

Visibility of the geographic bubble chart, specified as `'on'` or `'off'` or as the logical values `true` or `false`.

Parent/Child

`Parent` — Parent container
`Figure` object | `Panel` object | `Tab` object | `TiledChartLayout` object

Parent container, specified as a `Figure`, `Panel`, `Tab`, or `TiledChartLayout` object.

See Also

`geobubble` | `geolimits`

Topics

"Create Geographic Bubble Chart from Tabular Data"
"Use Geographic Bubble Chart Properties"
"Deploy Geographic Axes and Charts"
"Access Basemaps in MATLAB"
"Troubleshoot Geographic Axes or Chart Basemap Connection"
"Geographic Bubble Charts Overview"

Introduced in R2017b

geolimits

Set or query geographic limits

Syntax

```
geolimits(latlim,lonlim)
[latitudeLimits,longitudeLimits] = geolimits
geolimits('auto')
geolimits('manual')
[latitudeLimits,longitudeLimits] = geolimits(___)
___ = geolimits(gx,___)
```

Description

`geolimits(latlim,lonlim)` adjusts the geographic limits of the current geographic axes or chart to include latitudes ranging from `latlim(1)` to `latlim(2)` and longitudes from `lonlim(1)` to `lonlim(2)`. If there is no current geographic axes or chart, `geolimits` constructs a default `GeographicAxes` object and sets its limits.

`[latitudeLimits,longitudeLimits] = geolimits` returns the latitude limits and longitude limits of the current geographic axes or chart.

`geolimits('auto')` lets the geographic axes or chart choose its geographic limits based on its data locations.

`geolimits('manual')` requests that the axes or chart preserve its current limits as closely as possible when the map is resized or when its data locations change.

`[latitudeLimits,longitudeLimits] = geolimits(___)` adjusts the geographic limits and returns the actual limits of the map.

Note Typically, the actual limits chosen by `geolimits` are greater in extent than the requested limits because `geolimits` manages the limits to maintain a correct north-south/east-west aspect.

`___ = geolimits(gx, ___)` operates on the geographic axes or chart specified by `gx`.

Examples

Set Geographic Limits of Geographic Bubble Chart

Read tsunami data from a spreadsheet into a table.

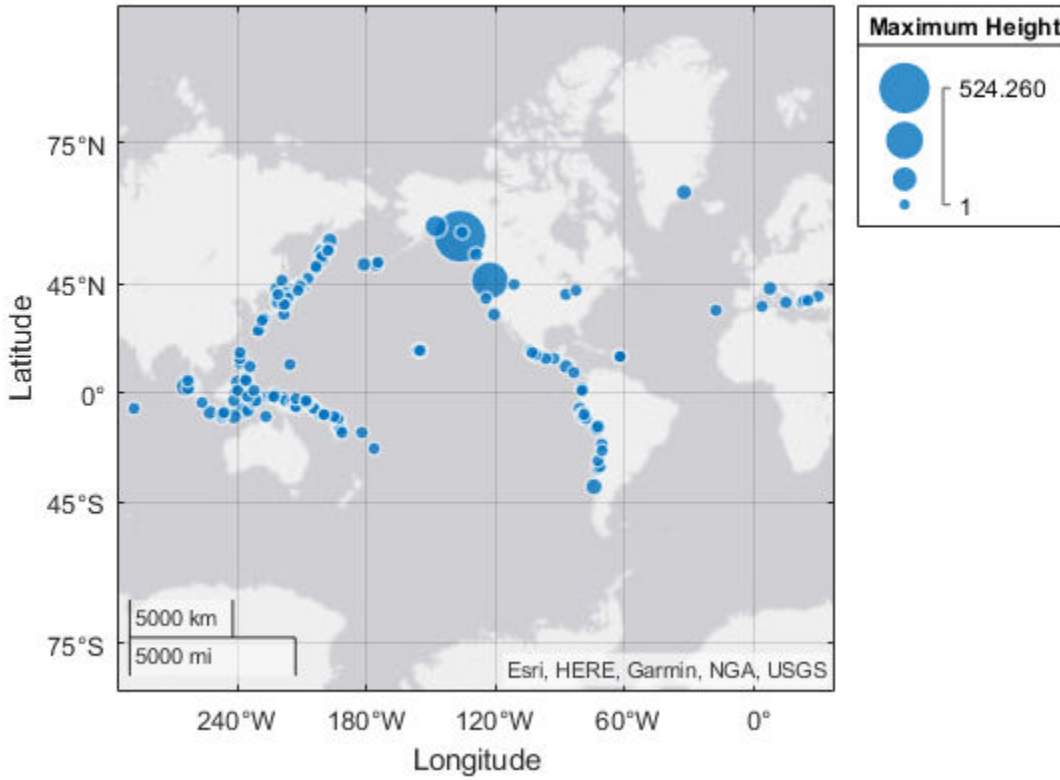
```
tsunamis = readtable('tsunamis.xlsx');
```

Get the latitude and longitude data from the table. This data specifies the locations of the tsunamis. In addition, get data about the size of each tsunami from the table.

```
lat = tsunamis.Latitude;
lon = tsunamis.Longitude;
sizedata = tsunamis.MaxHeight;
```

Create a geographic bubble chart with the tsunami data. Set the title of the size legend by specifying `SizeLegendTitle`.

```
gb = geobubble(lat,lon,sizedata,'SizeLegendTitle','Maximum Height');
```



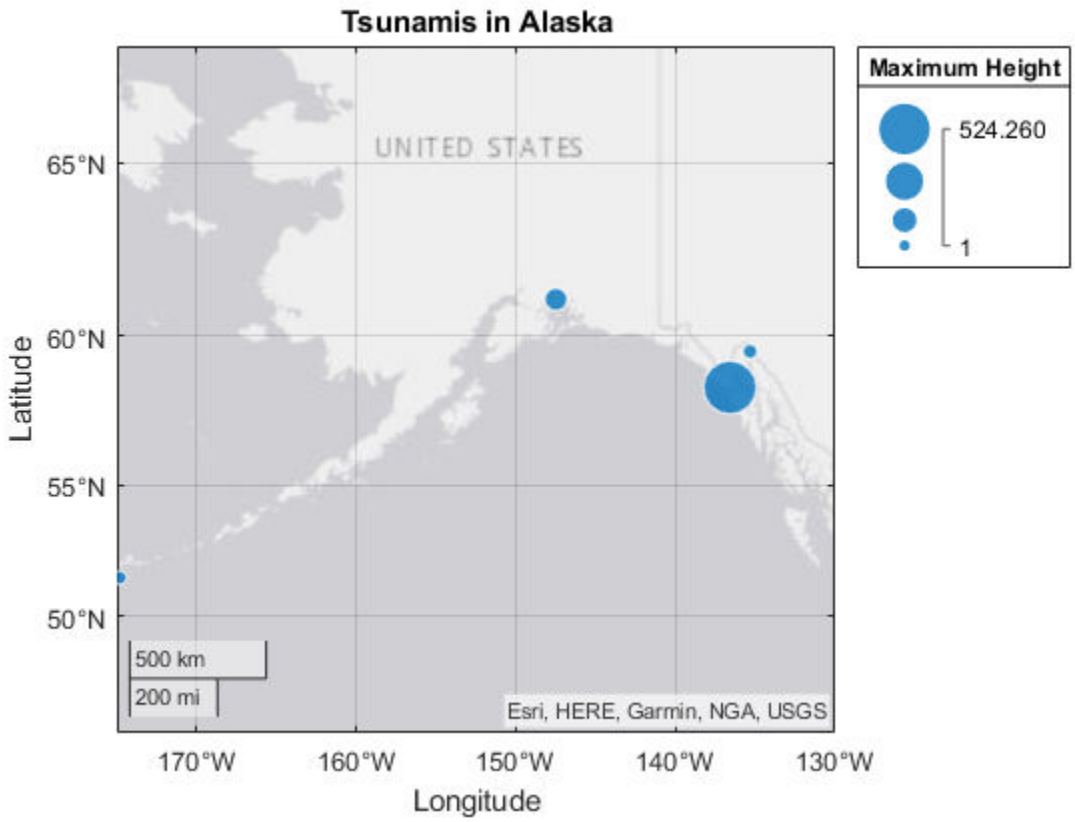
Get the current geographic limits of the chart. By default, `geobubble` sets the map limits large enough to encompass all the locations in the data.

```
[latlim, lonlim] = geolimits
```

```
latlim = 1×2

  -79.8575   85.0511


lonlim = 1×2

 -295.8338   37.8078
```

Modify the geographic limits to get a closer look at tsunami activity in Alaska. The example also specifies a title for the geographic bubble chart.

```
geolimits([50 65],[-175 -130])
gb.Title = 'Tsunamis in Alaska'
```




```
gb =
  GeographicBubbleChart with properties:

          Basemap: []
        MapLayout: 'normal'
     LatitudeData: [162×1 double]
    LongitudeData: [162×1 double]
         SizeData: [162×1 double]
```

```
        ColorData: []

  Show all properties
```

Get the actual geographic limits used by the chart. Due to zoom level quantization and aspect ratio preservation, the actual limits may differ from the requested limits.

```
[latlim, lonlim] = geolimits
```

```
latlim = 1×2

   45.0809   67.9103


lonlim = 1×2

 -175.0000 -130.0000
```

Input Arguments

`latlim` — Limits of the geographic axes or chart in latitude
two-element vector

Limits of the geographic axes or chart in latitude, specified as a two-element vector.

Example: `[50 65]`

Data Types: `single` | `double`

`lonlim` — Limits of the geographic axes or chart in longitude
two-element vector

Limits of the geographic axes or chart in longitude, specified as a two-element vector.

Example: `[-175 -130]`

Data Types: `single` | `double`

gx — Geographic axes or chart
`gca` (default) | `GeographicAxes` object | `GeographicBubbleChart` object

Geographic axes or chart, specified as a `GeographicAxes` object or `GeographicBubbleChart` object.

Output Arguments

`latitudeLimits` — Actual latitude limits of map
two-element vector

Actual latitude limits of map, returned as two-element vector.

`longitudeLimits` — Actual longitude limits of map
two-element vector

Actual longitude limits of map, returned as two-element vector.

See Also

GeographicBubbleChart Properties | `geoaxes` | `geobasemap` | `geobubble`

Topics

"Specify Map Limits with Geographic Axes"
"Plot in Geographic Coordinates"

Introduced in R2017b

geoplot

Plot line in geographic coordinates

Syntax

```
geoplot(lat,lon)
geoplot(lat,lon,LineSpec)
geoplot(lat1,lon1,...,latN,lonN)
geoplot(lat1,lon1,LineSpec1,...,latN,lonN,LineSpecN)

geoplot( ___ ,Name,Value)
geoplot(gx, ___ )
h = geoplot( ___ )
```

Description

`geoplot(lat,lon)` plots a line in a geographic axes with vertices at the latitude-longitude locations specified (in degrees) by the vectors `lat` and `lon`. The `lat` and `lon` inputs must be the same size. By default, `geoplot` draws a solid line, with no markers, using colors specified by the `ColorOrder` property of the Geographic axes object.

`geoplot(lat,lon,LineSpec)` sets the style, marker symbol, and color for the line.

`geoplot(lat1,lon1,...,latN,lonN)` combines the plots specified by several sets of latitude-longitude locations.

`geoplot(lat1,lon1,LineSpec1,...,latN,lonN,LineSpecN)` combines the plots specified by several sets of latitude-longitude locations, with a separate `LineSpec` for each line.

`geoplot( ___ ,Name,Value)` specifies properties of the chart line using one or more `Name,Value` pair arguments.

`geoplot(gx, ___ )` plots in the geographic axes specified by `gx` instead of the current axes.

`h = geoplot( ___ )` returns a column vector of Chart Line objects. Each object corresponds to a plotted line. Use `h` to modify the properties of the Chart line objects.

Examples

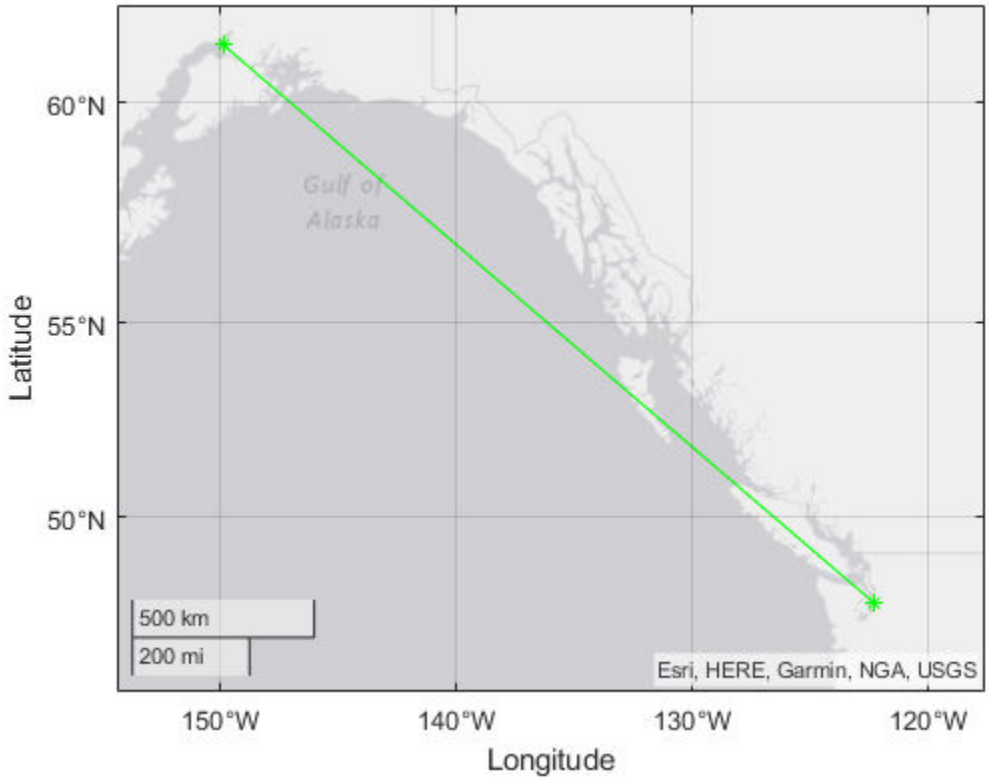
Create Geographic Plot

Plot a straight line between two points on a map. Specify the endpoints of the line using the coordinates of Seattle and Anchorage. Specify latitude and longitude in degrees.

```
latSeattle = 47.62;
lonSeattle = -122.33;
latAnchorage = 61.20;
lonAnchorage = -149.9;
```

Plot the data on a map. Customize the appearance of the line using the line specification `'g-*'`. Adjust the latitude and longitude limits of the map using `geolimits`.

```
geoplot([latSeattle latAnchorage],[lonSeattle lonAnchorage],'g-*')
geolimits([45 62],[-149 -123])
```



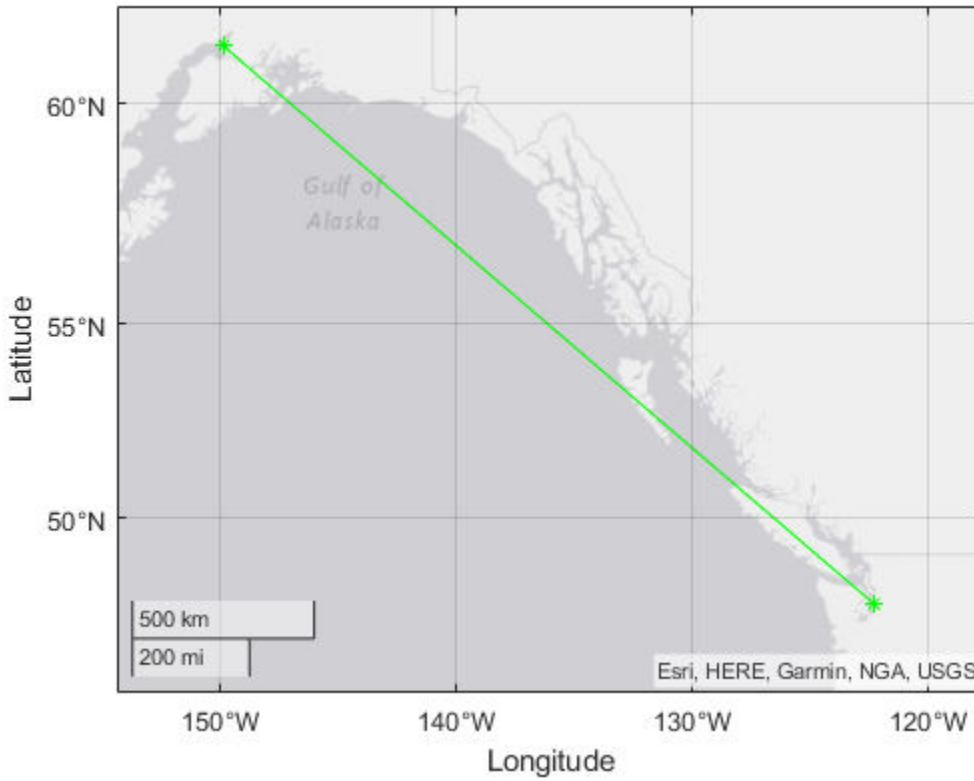
Add Text to Geographic Plot

Plot a straight line between two points on a map. Specify the endpoints of the line using the coordinates of Seattle and Anchorage. Specify latitude and longitude in degrees.

```
latSeattle = 47.62;
lonSeattle = -122.33;
latAnchorage = 61.20;
lonAnchorage = -149.9;
```

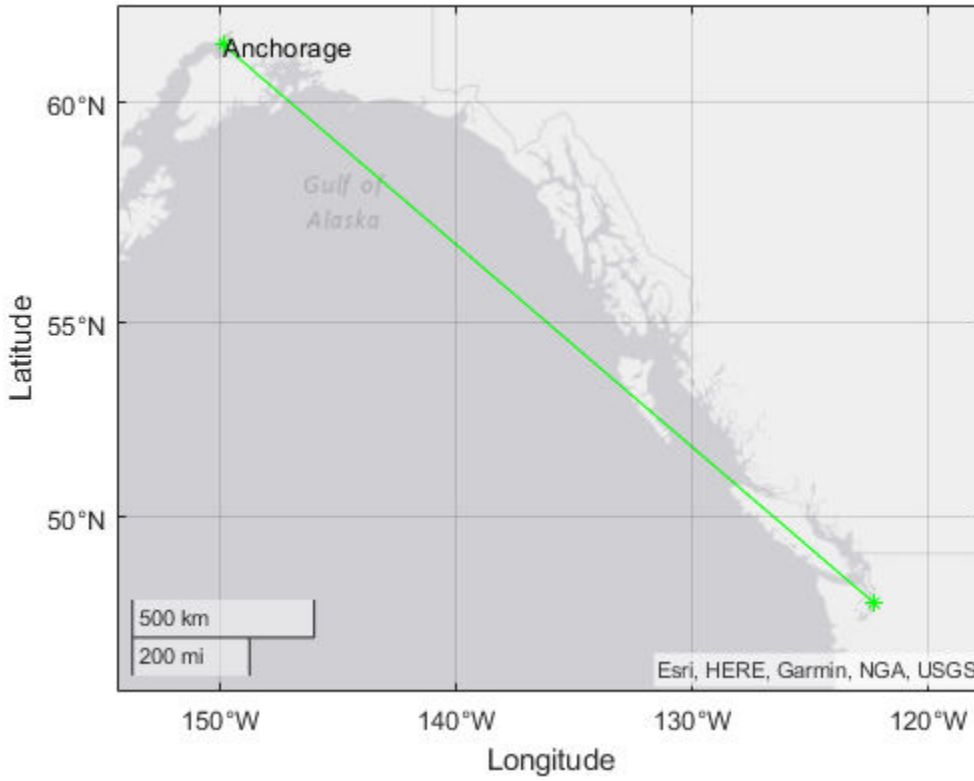
Plot the data using `geoplot`. Customize the appearance of the line using the line specification `'g-*'`. Adjust the latitude and longitude limits of the map using `geolimits`.

```
geoplot([latSeattle latAnchorage],[lonSeattle lonAnchorage],'g-*')
geolimits([45 62],[-149 -123])
```



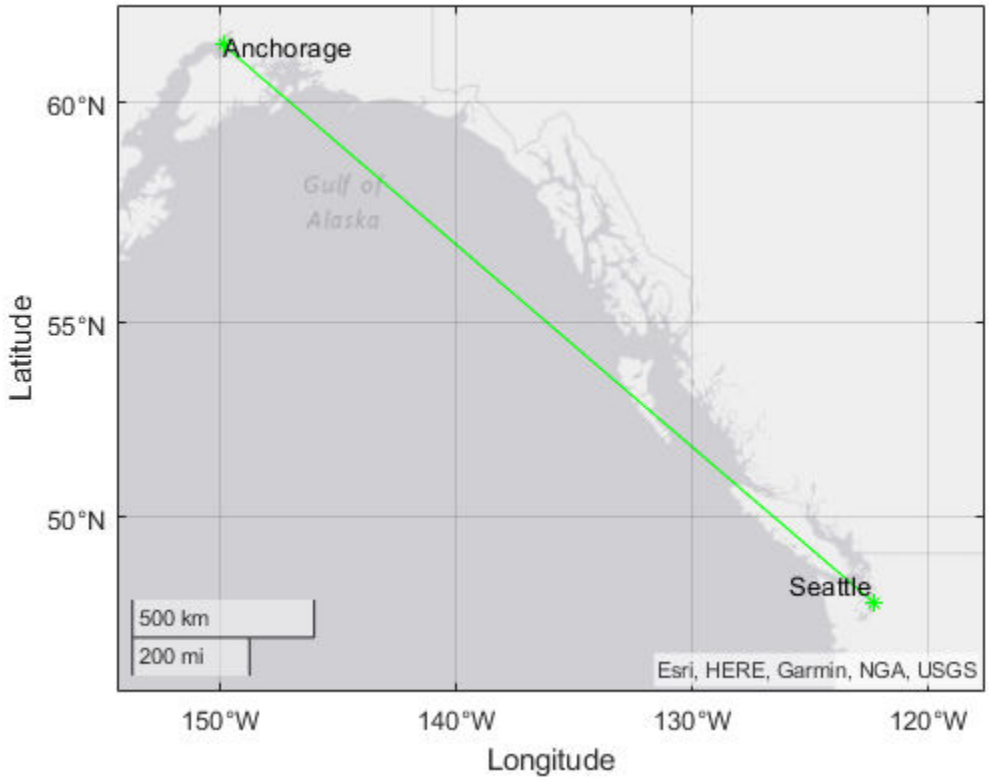
Identify Anchorage using the `text` function.

```
text(latAnchorage,lonAnchorage,'Anchorage');
```



Identify Seattle using the `text` function. Adjust the alignment of the text label by specifying properties of the `Text` object.

```
text(latSeattle,lonSeattle,'Seattle',...
    'HorizontalAlignment','right',...
    'VerticalAlignment','bottom');
```



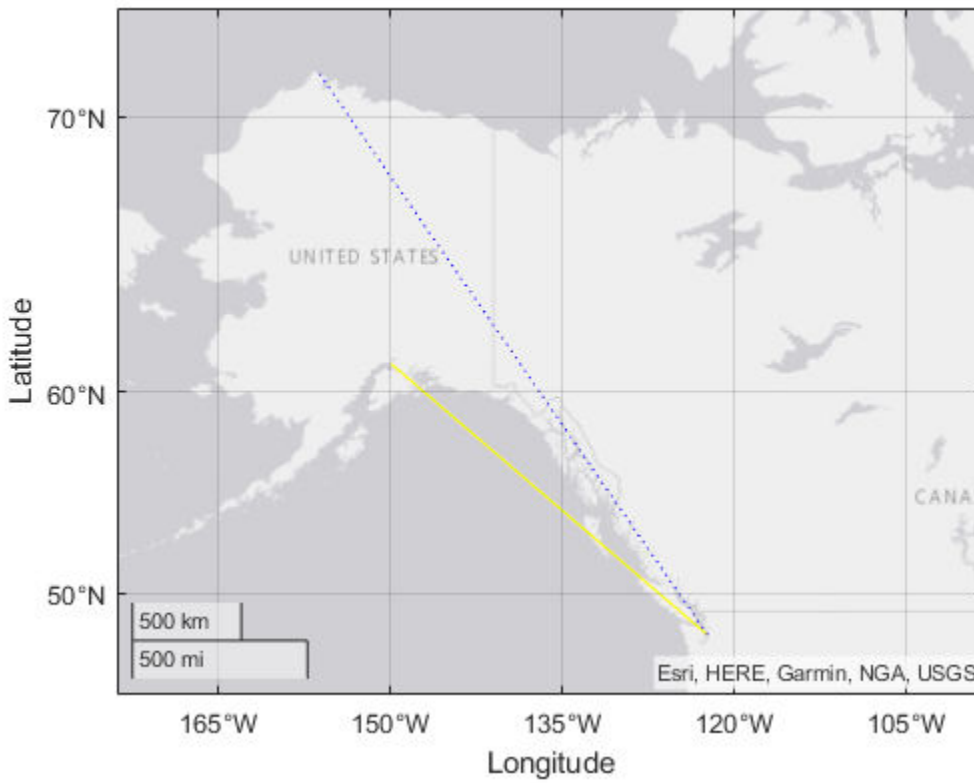
Customize Individual Lines on Map

Specify the latitude and longitude coordinates of Seattle, Anchorage, and Point Barrow.

```
latSeattle = 47.62;
lonSeattle = -122.33;

latAnchorage = 61.20;
lonAnchorage = -149.9;

latPtBarrow = 71.38;
lonPtBarrow = -156.47;
```

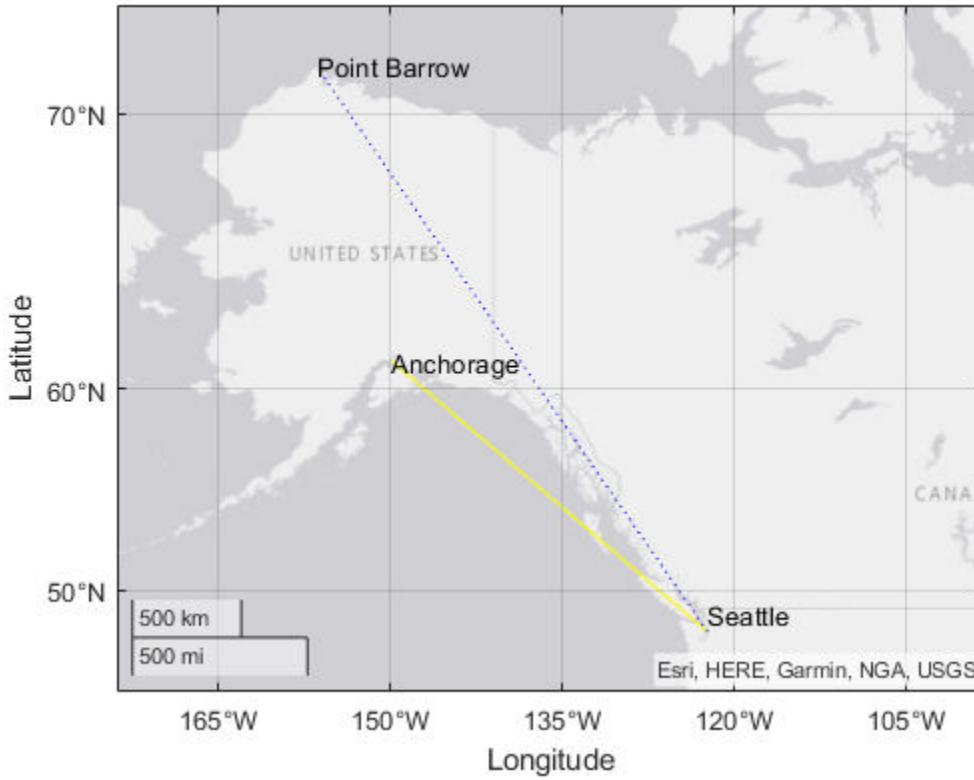
Plot straight lines from Seattle to each of the other two cities. Draw a solid yellow line from Seattle to Anchorage and a dotted blue line between Seattle and Point Barrow. Adjust the latitude and longitude limits of the map using `geolimits`.

```
geoplot([latSeattle latAnchorage],[lonSeattle lonAnchorage],'y-',...
    [latSeattle latPtBarrow],[lonSeattle lonPtBarrow],'b:')
geolimits([44 73],[-149 -123])
```



Label each city on the map using the `text` function.

```
text(latAnchorage,lonAnchorage,'Anchorage');
text(latPtBarrow,lonPtBarrow,'Point Barrow');
text(latSeattle,lonSeattle,'Seattle',...
    'VerticalAlignment','bottom');
```



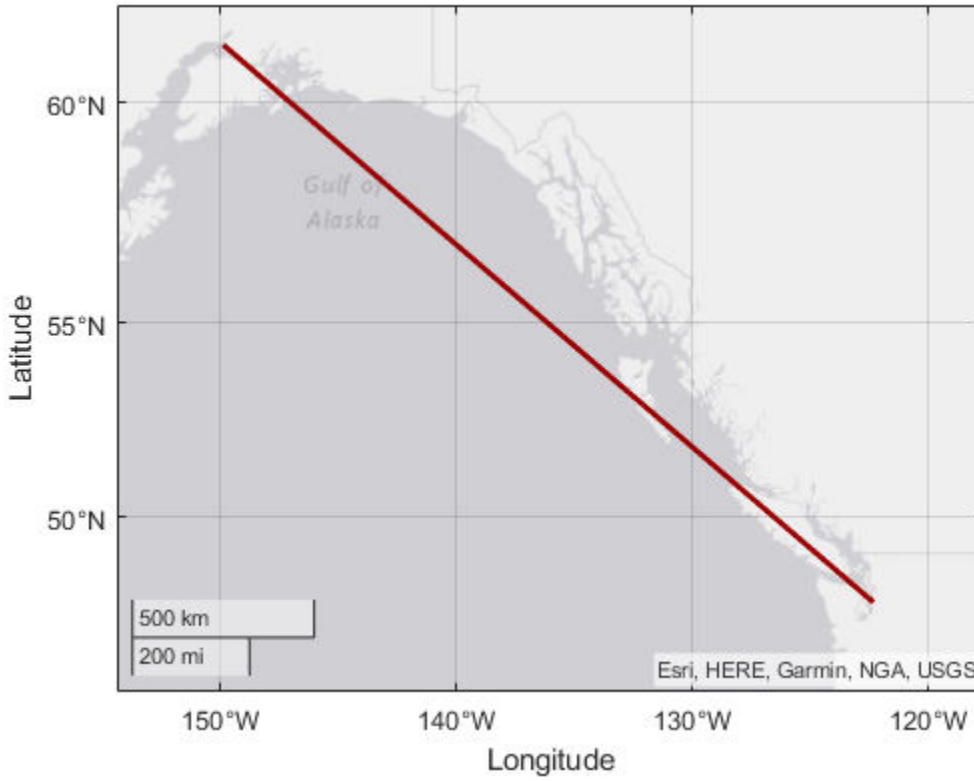
Use Chart Line Properties to Customize Appearance of the Line

Plot a straight line between two points on a map. Specify the endpoints of the line using the coordinates of Seattle and Anchorage. Specify latitude and longitude in degrees.

```
latSeattle = 47.62;
lonSeattle = -122.33;
latAnchorage = 61.20;
lonAnchorage = -149.9;
```

Plot the data using `geoplot`. Adjust the latitude and longitude limits of the map using `geolimits`. Customize the appearance of the line by specifying the `LineWidth` and `Color` properties.

```
geoplot([latSeattle latAnchorage],[lonSeattle lonAnchorage],...
    'LineWidth',2,'Color',[.6 0 0])
geolimits([45 62],[-149 -123])
```



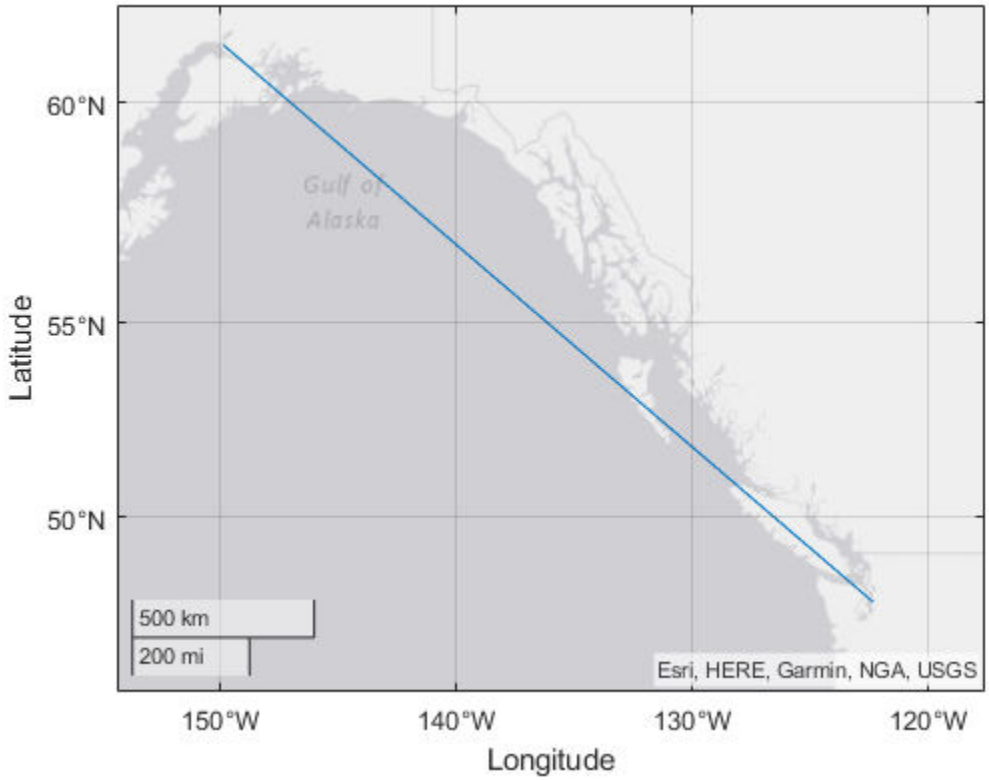
Change Basemap Used in Geographic Plot

Plot a straight line between two points on a map. Specify the endpoints of the line using the coordinates of Seattle and Anchorage. Specify latitude and longitude in degrees.

```
latSeattle = 47.62;
lonSeattle = -122.33;
latAnchorage = 61.20;
lonAnchorage = -149.9;
```

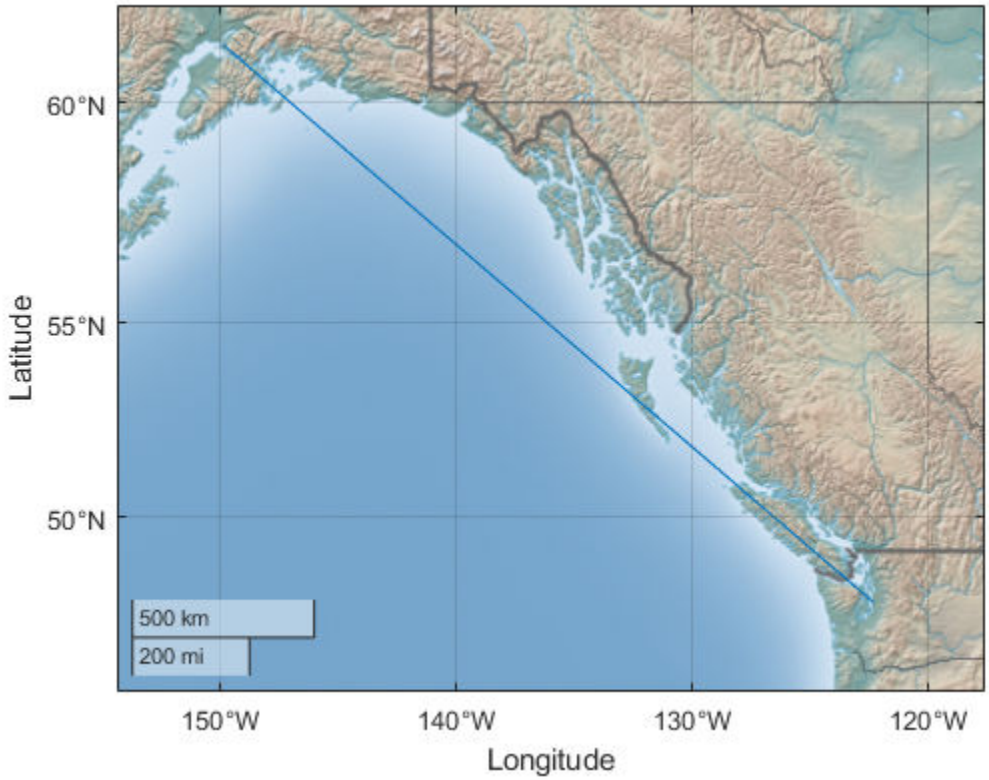
Plot the data using `geoplot`. Adjust the latitude and longitude limits of the map using `geolimits`.

```
geoplot([latSeattle latAnchorage],[lonSeattle lonAnchorage])
geolimits([45 62],[-149 -123])
```



Change the basemap used in the plot using the `geobasemap` function.

```
geobasemap colorterrain
```



Input Arguments

`lat` — Latitude coordinates in degrees
real, numeric, finite vector

Latitude coordinates in degrees, specified as a real, numeric, finite vector. The vector can contain embedded `NaN`s. `lat` must be the same size as `lon`.

Example: `[43.0327 38.8921 44.0435]`

Data Types: `single` | `double`

`lon` — Longitude coordinates in degrees
real, numeric, finite vector

Longitude coordinates in degrees, specified as a real, numeric, finite vector. The vector can contain embedded `NaN`s. `lon` must be the same size as `lat`.

Example: `[-107.5556 -77.0269 -72.5565]`

Data Types: `single` | `double`

`LineSpec` — Line specification
character vector | string

Line specification, specified as a character vector or string containing a line style, marker, and color specifier. The elements can appear in any order, and you can omit one or more options. If you omit the line style and specify the marker, the plot shows only markers with no line connecting them. For more information, see `LineSpec (Line Specification)`.

Example: `'--or'` is a red dashed line with circular markers

Data Types: `char` | `string`

`gx` — Target geographic axes
`GeographicAxes` object

Target geographic axes, specified as a `GeographicAxes` object. You can modify the appearance and behavior of a `GeographicAxes` object by setting its properties. For a list of properties, see GeographicAxes Properties.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Note Property settings apply to all the lines in the plot. To set the properties of an individual line, retrieve the handle to the line in the `geoplot` return value `h` and use dot notation.

The name-value pairs listed here are only a subset. For a full list, see Chart Line.

Example: `h = geoplot(lat,lon,'Color','g')`

`Color` — Line color
`[0 1 0]` (default) | MATLAB `ColorSpec` | ...

Line color, specified as a MATLAB `ColorSpec (Color Specification)`. If specifying an RGB triplet, the intensities must be in the range `[0,1]`.

Example: `'green'`

Example: `'g'`

Example: `[0 1 0]`

`LineStyle` — Line style
`'-'` (default) | MATLAB `LineSpec` | ...

Line style, specified as a MATLAB `LineSpec (Line Specification)`.

Example: `'--'` (dashed line)

Example: `':'` (dotted line)

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

`Marker` — Marker symbol
`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | `'x'` | `...`

Marker symbol, specified as one of the markers in this table. By default, a chart line does not have markers. Add markers at each data point along the line by specifying a marker symbol.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross

Value	Description
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerFaceColor` — Marker face color
`'none'` (default) | `'auto'` | MATLAB `ColorSpec` | ...

Marker face color, specified as `'none'`, `'auto'` or a MATLAB `ColorSpec (Color Specification)`. The `'auto'` value uses the same color as the `Color` name-value pair for the axes. If you specify an RGB triplet, the intensities must be in the range `[0,1]`.

Example: `'green'`

Example: `'g'`

Example: `[0 1 0]`

`MarkerSize` — Marker size
`6` (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

Output Arguments

`h` — Geographic plot
column vector of `Line` objects

Geographic plot, returned as a column vector of `Line` objects. Each object corresponds to a plotted line. Use `h` to modify the properties of the objects after they area created.

Tips

- To customize certain aspects of your plot, use properties of the geographic axes. For a list of properties, see GeographicAxes Properties.
- To plot additional data in the geographic axes, use the `hold on` command.
- You cannot plot data that requires Cartesian axes in a geographic axes.

See Also

Functions
`geoaxes` | `geobasemap` | `geobubble` | `geoscatter` | `plot`

Properties
Chart Line | GeographicAxes Properties

Introduced in R2018b

GeographicRuler Properties

Control appearance and behavior of axis with geographic values

Description

`GeographicRuler` properties control the appearance and behavior of the latitude axis and the longitude axis in a `GeographicAxes`. Each individual axis is a `GeographicRuler` object. By changing property values of the ruler, you can modify certain aspects of a specific axis.

For certain ruler properties, you can customize the appearance of both rulers (latitude axis and longitude axis) by setting properties on the geographic axes instead of setting properties of each individual ruler. For example, using the `GeographicRuler` properties, you can specify a different color for each ruler. Typically, it's preferable, and simpler, to specify the color of both rulers using the `GeographicAxes` property, and keep the rulers consistent in appearance.

Use dot notation to refer to a ruler property. Access the ruler object through the `LatitudeAxis` or `LongitudeAxis` property of the `GeographicAxes` object.

```
% Create a GeographicAxes.
gx = geoaxes;

% View GeographicRuler object.
latruler = gx.LatitudeAxis

latruler =

  GeographicRuler with properties:

             Limits: [-85.0511 85.0511]
         TickValues: [-75 -45 0 45 75]
    TickLabelFormat: 'dms'

  Show all properties
```

Properties

Appearance

`Limits` — Minimum and maximum axis limits

two-element vector of the form `[min max]`

Minimum and maximum axis limits, specified as a two-element vector of the form `[min max]`. `Limits` is a read-only property.

To set the limits of a `GeographicAxes`, use the `geolimits` function.

Note The `Limits` property for the latitude and longitude `GeographicRuler` objects always has the same value as the `LatitudeLimits` property and the `Longitudelimits` properties of the parent `GeographicAxes` object, respectively.

Example: `latlim = gx.LatitudeAxis.Limits;`

`Color` — Color of axis line and labels

`[0.15 0.15 0.15]` (default) | RGB triplet | hexadecimal color code | color name | short color name

Color of the axis line and labels, specified as an RGB triplet, a hexadecimal color code, a color name, or a short color name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Note Setting the `AxisColor` property for the parent `GeographicAxes` sets the `Color` property for the `GeographicRuler` objects to the same value. These ruler objects are associated with the `LatitudeAxis` and `LongitudeAxis` properties of the `GeographicAxes`. Conversely, setting these `GeographicRuler` properties does not set the `GeographicAxes` property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `gx.LatitudeAxis.Color = 'b'`

Example: `gx.LatitudeAxis.Color = 'blue'`

Example: `gx.LatitudeAxis.Color = [0.1 0.7 0.8];`

Example: `gx.LatitudeAxis.Color = '#0000FF';`

`LineWidth` — Width of axis line and tick marks
`0.5` (default) | positive value

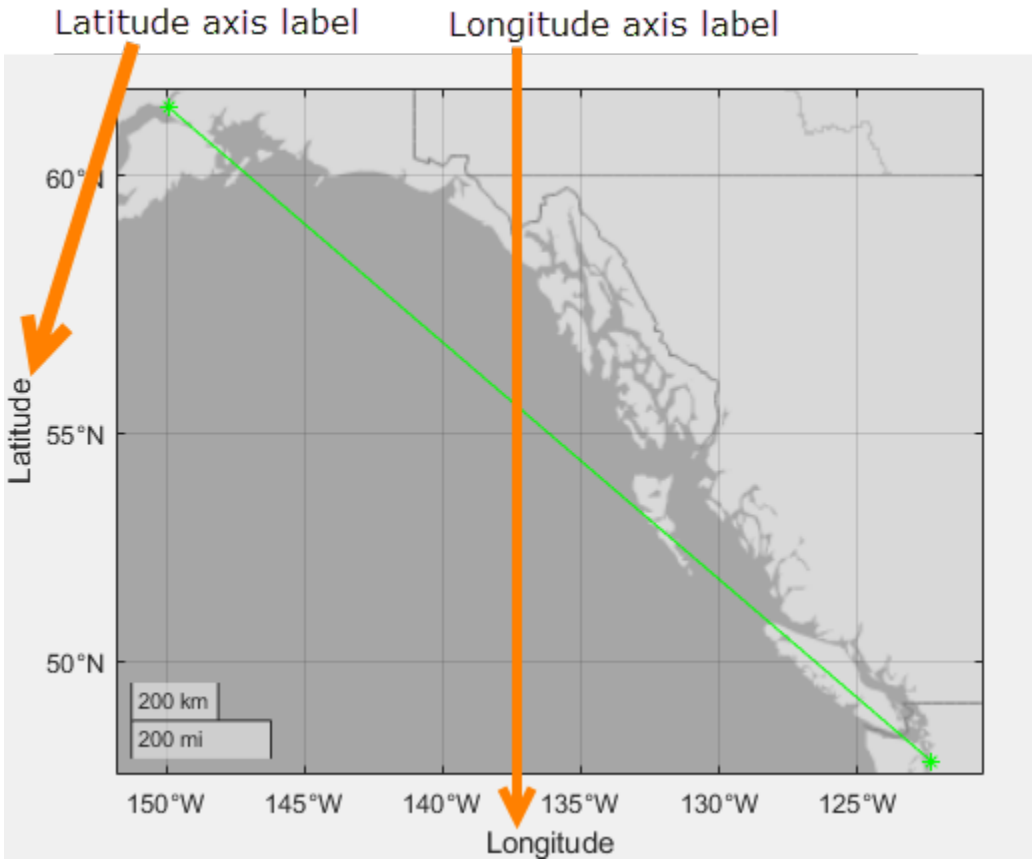
Width of axis line and tick marks, specified as a positive value in point units. One point equals 1/72 inch.

Note Setting the `LineWidth` property for the parent `GeographicAxes` sets the `LineWidth` property for the `GeographicRuler` objects to the same value. These ruler objects are associated with the `LatitudeAxis` and `LongitudeAxis` properties of the `GeographicAxes`. Conversely, setting a ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `gx.LatitudeAxis.LineWidth = 2;`

`Label` — Axis label
`Text` object

Axis label, specified as a `Text` object.



To change existing text, set the `String` property of the `Text` object. Use other `Text` object properties to change the text appearance, such as the font style or color.

```
gx.LatitudeAxis.Label.String = 'Latitude Axis';
gx.LatitudeAxis.Label.FontSize = 16;
```

For a full list of options, see the Text object property reference page.

Note The `Text` object is not a child of the ruler object and, thus, it does not appear in the list of objects returned by `findobj`. This `Text` object does not use default `Text` property values.

`Visible` — Axis ruler visibility

`'on'` (default) | `'off'`

Axis ruler visibility, specified as one of these values:

- `'on'` — Display the axis ruler.
- `'off'` — Hide the axis ruler without deleting it. You still can access properties of an invisible axis ruler using the ruler object.

Example: `gx.LatitudeAxis.Visible = 'off';`

Tick Values and Labels

`TickValues` — Tick mark locations along the axis

vector of increasing values

Tick mark locations along the axis, specified as a vector of increasing values.

If you assign a value to this property, MATLAB sets the `TickValuesMode` property to `'manual'`.

Example: `gx.LatitudeAxis.TickValues = [-15 -10 0 10 15];`

Example: `gx.LongitudeAxis.TickValues = 0:10:100;`

`TickValuesMode` — Selection mode for `TickValues` property

`'auto'` (default) | `'manual'`

Selection mode for the `TickValues` property, specified as one of these values:

- `'auto'` — Automatically select the tick values based on the limits of the map and the position (size) of the axes.
- `'manual'` — Use tick values that you specify. To specify the values, set the `TickValues` property.

`TickLabels` — Tick mark labels

cell array of character vectors | string array

Tick mark labels, specified as a cell array of character vectors or a string array. If you do not specify enough labels for all of the tick values, then the labels repeat. The labels support TeX and LaTeX markup. See the `TickLabelInterpreter` property for more information.

Example: `gx.TickValues = [-66.5 -23.5 0 23.5 66.5];gx.LatitudeAxis.TickLabels = {'Antarctic Circle','Tropic of Capricorn','Equator','Tropic of Cancer','Artic Circle'};`

If you set this property, MATLAB sets the `TickLabelsMode` property to `'manual'`.

Data Types: `char` | `string`

`TickLabelsMode` — Selection mode for `TickLabels` property
`'auto'` (default) | `'manual'`

Selection mode for the `TickLabels` property, specified as one of these values:

- `'auto'` — Automatically select the tick labels.
- `'manual'` — Use tick labels that you specify. To specify the labels, set the `TickLabels` property.

`TickLabelInterpreter` — Interpretation of tick label characters
`'tex'` (default) | `'latex'` | `'none'`

Interpretation of tick label characters, specified as one of these values:

- `'tex'` — Interpret characters using a subset of TeX markup.
- `'latex'` — Interpret characters using LaTeX markup.
- `'none'` — Display literal characters.

For more information about these interpreters, see the `Interpreter` property on the Text Properties reference page.

Example: `gx.LatitudeAxis.TickLabelInterpreter = 'latex';`

`TickLabelFormat` — Tick label format
`'dms'` (default) | `'dd'` | `'dm'` | `-'dd'` | `'-dm'` | `'-dms'`

Tick label format, specified as one of the following values.

Format	Description	Example
`'dd'`	Decimal degrees plus compass direction	`23°N`

Format	Description	Example
`'dm'`	Degrees and (decimal) minutes plus compass direction	`18°30'W`
`'dms'`	Degrees, minutes, and (decimal) seconds plus compass direction	`110°06'18.5"E`
`'-dd'`	Decimal degrees with minus sign (-) to indicate south and west.	`-115.25°`
`'-dm'`	Degrees and (decimal) minutes with minus sign (-) to indicate south and west.	`-5°45.5'`
`'-dms'`	Degrees, minutes, and (decimal) seconds with minus sign (-) to indicate south and west.	`-3°21'05"`

You can also specify the tick label format using the `geotickformat` function.

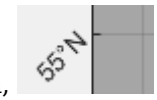
Note Setting the `TickLabelFormat` property for the parent `GeographicAxes` sets the `TickLabelFormat` property for the `GeographicRuler` objects to the same value. These ruler objects are associated with the `LatitudeAxis` and `LongitudeAxis` properties of the `GeographicAxes`. Conversely, setting either of the ruler properties does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `gx.LatitudeAxis.TickLabelFormat = 'dd';`

`TickLabelRotation` — Rotation of tick labels

scalar value in degrees

Rotation of tick labels, specified as a scalar value in degrees. Positive values give counterclockwise rotation. Negative values give clockwise rotation. For example, specify

`gx.LatitudeAxis.TickLabelRotation = 45;` to rotate the tick labels 45 degrees in a counterclockwise direction, 

If you assign a value to this property, then MATLAB sets the `TickLabelRotation` property to `'manual'`.

Example: `gx.LatitudeAxis.TickLabelRotation = -45;`

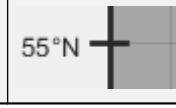
`TickLabelRotationMode` — Selection mode for `TickLabelRotation` property
`'auto'` (default) | `'manual'`

Selection mode for the `TickLabelRotation` property, specified as one of these values:

- `'auto'` — Object automatically selects the tick label rotation.
- `'manual'` — Use a tick label rotation that you specify. To specify the rotation, set the `TickLabelRotation` property.

`TickDirection` — Tick mark direction
`'in'` (default) | `'out'` | `'both'`

Tick mark direction, specified as one of these values:

Value	Description	Example
`'in'`	Direct the tick marks inward from the axis lines.	
`'out'`	Direct the tick marks outward from the axis lines.	
`'both'`	Center the tick marks over the axis lines.	

If you assign a value to this property, MATLAB sets the `TickDirectionMode` property to `'manual'`.

Note Setting the `TickDir` property for the parent `GeographicAxes` sets the `TickDirection` property for the `GeographicRuler` object to the same value. The

`GeographicRuler` object is associated with the `LatitudeAxis` and `LongitudeAxis` properties of the `GeographicAxes`. Conversely, setting one of the ruler properties does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `gx.LatitudeAxis.TickDirection = 'out';`

`TickDirectionMode` — Selection mode for `TickDirection` property
`'auto'` (default) | `'manual'`

Selection mode for the `TickDirection` property, specified as one of these values:

- `'auto'` — Automatically select the tick direction.
- `'manual'` — Use a tick direction that you specify. To specify the tick direction, set the `TickDirection` property.

`TickLength` — Tick mark length
two-element vector

Tick mark length, specified as a two-element vector of the form `[length unused]`. (The `GeographicRuler` object uses a two-element vector to be consistent with the value of this property in other ruler objects.) The first element is the tick mark length. Specify the values in units normalized relative to the longest axes dimension.

Note Setting the `TickLength` property for the parent `GeographicAxes` sets the `TickLength` property for the `GeographicRuler` objects to the same value. The `GeographicRuler` objects are associated with the `LatitudeAxis` and `LongitudeAxis`. Conversely, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `gx.LatitudeAxis.TickLength = [0.04 0];`

Font

`FontName` — Font name
system supported font name | `'FixedWidth'`

Font name, specified as a system supported font name or `'FixedWidth'`. The default font depends on the specific system and locale. To use a fixed-width font that looks good

in any locale, specify `'FixedWidth'`. The actual fixed-width font used depends on the `FixedWidthFontName` property of the root object.

Note Setting the `FontName` property for the parent axes sets the `FontName` property for the `GeographicRuler` objects to the same value. The `GeographicRuler` objects are associated with `LatitudeAxis` and `LongitudeAxis`. Conversely, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `gx.LatitudeAxis.FontName = 'Cambria';`

`FontSize` — Font size
scalar numeric value

Font size, specified as a scalar numeric value. The default font size depends on the specific operating system and locale.

Note Setting the `FontSize` property for the parent axes sets the `FontSize` property for the `GeographicRuler` objects to the same value. The `GeographicRuler` objects are associated with `LatitudeAxis` and `LongitudeAxis` properties. Conversely, setting the ruler properties does not set the axes property. To prevent the axes property value from overriding the value of the ruler properties, set the axes value first, and then set the ruler value.

Example: `gx.LatitudeAxis.FontSize = 12;`

`FontWeight` — Thickness of text characters
`'normal'` (default) | `'bold'`

Thickness of the text characters, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font.
- `'bold'` — Thicker character outlines than normal.

MATLAB uses the `FontWeight` property to select a font from the fonts available on your system. Not all fonts have a bold font weight. Therefore, specifying a bold font weight still can result in the normal font weight.

Note Setting the `FontWeight` property for the parent axes sets the `FontWeight` property for the `GeographicRuler` objects to the same value. The `GeographicRuler` objects are associated with the `LatitudeAxis` and `LongitudeAxis` properties. Conversely, setting the ruler properties does not set the axes property. To prevent the axes property value from overriding the ruler property values, set the axes value first, and then set the ruler value.

Example: `gx.LatitudeAxis.FontWeight = 'bold';`

`FontAngle` — Text character slant

`'normal'` (default) | `'italic'`

Character slant, specified as `'normal'` or `'italic'`. Not all fonts have both font styles. Therefore, the italic font can look the same as the normal font.

Note Setting the `FontAngle` property for the parent axes sets the `FontAngle` property for the `GeographicRuler` objects for both the `LatitudeAxis` and `LongitudeAxis` to the same value. Conversely, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property values, set the axes value first, and then set the ruler value.

Example: `gx.LatitudeAxis.FontAngle = 'italic';`

`FontSmoothing` — Text antialiasing

`'on'` (default) | `'off'`

Text smoothing, specified as one of these values:

- `'on'` — Enable text antialiasing to reduce the jagged appearance of text characters and make the text easier to read. In certain cases, smoothed text blends against the background color and can make the text appear blurry.
- `'off'` — Disable text antialiasing. Use this setting if the text seems blurry.

Example: `gx.LatitudeAxis.FontSmoothing = 'off';`

Parent/Child

`Parent` — Ruler parent

`Axes` object

Ruler parent, specified as an `Axes` object.

Note Ruler objects are not listed in the `Children` property of the parent `Axes` object.

`Children` — Ruler children
empty `GraphicsPlaceholder` array

The ruler has no children. You cannot set this property.

See Also

GeographicAxes Properties

Introduced in R2019a

GeographicScalebar Properties

Control appearance and behavior of scale bar in geographic axes

Description

`GeographicScalebar` properties control the appearance and behavior of the scale bar included in a `GeographicAxes` object.

Access the `GeographicScalebar` object through the `Scalebar` property of the `GeographicAxes` object. Use dot notation to refer to the `GeographicScalebar` properties.

```
% Create a Geographic Axes.
gx = geoaxes;

% Get the GeographicScalebar object.
sb = gx.Scalebar;

% Set a GeographicScalebar object property.
gx.Scalebar.Visible = 'off';
```

Properties

Appearance

`BackgroundAlpha` — Transparency of scale bar background

`0.45` (default) | scalar

Transparency of scale bar background, specified as a scalar between `0` and `1`, inclusive. A value of `1` means the scale bar background is fully opaque and `0` means it is completely transparent (invisible).

Example: `gx.Scalebar.BackgroundAlpha = 0.2;`

`BackgroundColor` — Background color of scale bar

`[1 1 1]` (default) | RGB triplet | hexadecimal color code | color name | short color name

Background color of scale bar, specified as an RGB triplet, a hexadecimal color code, a color name, or a short color name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `gx.Scalebar.BackgroundColor = [0 0 1];`

Example: `gx.Scalebar.BackgroundColor = 'b';`

Example: `gx.Scalebar.BackgroundColor = 'none';`

Example: `gx.Scalebar.BackgroundColor = '#0000FF';`

`EdgeColor` — Color of lines in scale bar

`[0.15 0.15 0.15]` (default) | RGB triplet | hexadecimal color code | color name | short color name

Color of lines in scale bar, specified as an RGB triplet, a hexadecimal color code, a color name, or a short color name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Note Setting the `AxisColor` property for the parent geographic axes sets the `EdgeColor` property for the scale bar to the same value. Conversely, setting the scale bar property does not set the `AxisColor` property. To prevent the axis property value from overriding the scale bar property value, set the axis value first, and then set the scale bar property value.

Example: `gx.Scalebar.EdgeColor = 'b';`

Example: `gx.Scalebar.EdgeColor = 'blue';`

Example: `gx.Scalebar.EdgeColor = [0 0 1];`

Example: `gx.Scalebar.EdgeColor = '#0000FF';`

`LineWidth` — Width of lines in scale bar

`0.5` (default) | positive value

Width of lines in scale bar, specified as a positive value in point units. One point equals 1/72 inch.

Example: `gx.Scalebar.LineWidth = 2;`

Note Setting the `LineWidth` property for the parent geographic axes sets the `LineWidth` property for the scale bar to the same value. Conversely, setting the scale bar property does not set the axes property. To prevent the axes property value from overriding the scale bar property value, set the axes value first, and then set the scale bar property value.

`Visible` — Visibility of scale bar

`'on'` (default) | `'off'`

Visibility of scale bar, specified as one of these values:

- `'on'` — Display the scale bar.
- `'off'` — Hide the scale bar without deleting it. You can still access properties of an invisible scale bar by using the `GeographicScalebar` object.

Example: `gx.Scalebar.Visible = 'off';`

Font

`FontName` — Font name

system-supported font name | `'FixedWidth'`

Font name, specified as a system-supported font name or `'FixedWidth'`. The default font depends on the specific system and locale. To use a fixed-width font that works well in any locale, specify `'FixedWidth'`. The actual fixed-width font used depends on the `FixedWidthFontName` property of the root object.

Note Setting the `FontName` property for the parent geographic axes sets the `FontName` property for the scale bar to the same value. Conversely, setting the scale bar property does not set the axes property. To prevent the axes property value from overriding the

scale bar property value, set the axes value first, and then set the scale bar property value.

Example: `gx.Scalebar.FontName = 'Cambria';`

`FontSize` — Font size
8 (default) | scalar numeric value

Font size, specified as a scalar numeric value.

Note Setting the `FontSize` property for the parent geographic axes sets the `FontSize` property for the scale bar object. Conversely, setting the scale bar property does not set the parent axes property. To prevent the axes property value from overriding the scale bar property value, set the axes value first, and then set the scale bar property value.

When using the font size specified by the parent geographic axes, the scale bar scales the font size to 80% of the parent font size.

Example: `gx.Scalebar.FontSize = 12;`

`FontWeight` — Character thickness
`'normal'` (default) | `'bold'`

Character thickness, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than default weight

MATLAB uses the `FontWeight` property to select a font from those fonts available on your system. Not all fonts have a bold font weight. Therefore, specifying a bold font weight still can result in the normal font weight.

Note Setting the `FontWeight` property for the parent geographic axes sets the `FontWeight` property for the scale bar to the same value. Conversely, setting the scale bar property does not set the parent axes property. To prevent the axes property value from overriding the scale bar property value, set the axes value first, and then set the scale bar property value.

Example: `gx.Scalebar.FontWeight = 'bold';`

`FontColor` — Font color

`[0.15 0.15 0.15]` (default) | RGB triplet | hexadecimal color code | color name | short color name

Font color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short color name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Note Setting the `AxisColor` property for the parent geographic axes sets the `FontColor` property for the scale bar to the same value. Conversely, setting the scale bar property does not set the parent axes property. To prevent the parent axes property value from overriding the scale bar property value, set the axes value first, and then set the scale bar property value.

Example: `gx.Scalebar.FontColor = 'b';`

Example: `gx.Scalebar.FontColor = 'blue';`

Example: `gx.Scalebar.FontColor = [0 0 1];`

Example: `gx.Scalebar.FontColor = '#0000FF';`

`FontAngle` — Character slant

`'normal'` (default) | `'italic'`

Character slant, specified as `'normal'` or `'italic'`. Not all fonts have both font styles. Therefore, the italic font can look the same as the normal font.

Note Setting the `FontAngle` property for the parent geographic axes sets the `FontAngle` property for the scale bar to the same value. Conversely, setting the scale bar property does not set the parent axes property. To prevent the parent axes property value from overriding the scale bar property value, set the axes value first, and then set the scale bar property value.

Example: `gx.Scalebar.FontAngle = 'italic';`

`FontSmoothing` — Character smoothing

`'on'` (default) | `'off'`

Character smoothing, specified as one of these values:

- `'on'` — Enable text antialiasing, or character smoothing, to reduce the jagged appearance of text characters and make the text easier to read. In certain cases, smoothed text blends against the background color and can make the text appear blurry.
- `'off'` — Disable text antialiasing. Use this setting if the text appears blurry.

Example: `gx.Scalebar.FontSmoothing = 'off';`

Parent/Child

`Parent` — Scale bar parent

`GeographicAxes`

This property is read-only.

Scale bar parent, specified as a `GeographicAxes` object.

Note Scale bar objects are not listed in the `Children` property of the parent `GeographicAxes` object.

`Children` — Scale bar children

empty `GraphicsPlaceholder` array

This property is read-only.

Scale bar children, specified as an empty `GraphicsPlaceholder` array. The scale bar has no children.

See Also

GeographicAxes Properties

Introduced in R2019a

geoscatter

Scatter chart in geographic coordinates

Syntax

```
geoscatter(lat,lon)
geoscatter(lat,lon,A)
geoscatter(lat,lon,A,C)

geoscatter( ___ ,M)
geoscatter( ___ ,'filled')
geoscatter( ___ ,Name,Value)
geoscatter(gx, ___ )
s = geoscatter( ___ )
```

Description

`geoscatter(lat,lon)` displays colored circles in a geographic axes at the latitude-longitude locations specified (in degrees) by the vectors `lat` and `lon`. `lat` and `lon` must be the same size.

`geoscatter(lat,lon,A)` uses `A` to specify the area of each marker (in points^2). To draw all the markers with the same size, specify `A` as a scalar. To draw the markers with different sizes, specify `A` as a vector the same length as `lat` and `lon`. If you do not specify `A`, `geoscatter` uses the default size.

`geoscatter(lat,lon,A,C)` uses `C` to specify the color of each marker.

`geoscatter( ___ ,M)` creates a scatter plot where `M` specifies the marker used. By default, `geoscatter` uses circles as the marker.

`geoscatter( ___ ,'filled')` fills the markers.

`geoscatter( ___ ,Name,Value)` specifies properties of the scatter plot using one or more `Name,Value` pair arguments. The property settings apply to all the scatter plots.

`geoscatter(gx, ___ )` plots into the geographic axes specified by `gx` instead of into the current axes.

`s = geoscatter( ___ )` returns the `Scatter` object. Use `S` to modify properties of the object after it is created.

Examples

Create Geographic Scatter Plot Controlling Color, Size, and Shape of Markers

Set up latitude and longitude data.

```
lon = (-170:10:170);
lat = 50 * cosd(3*lon);
```

Define data that controls the area of each marker.

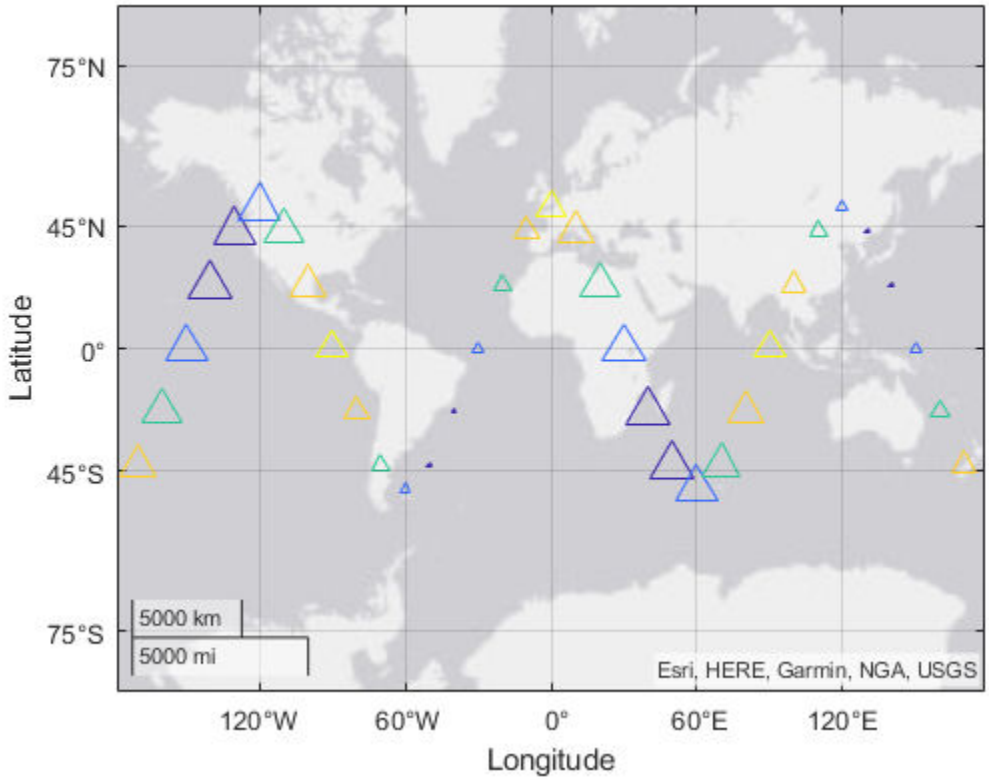
```
A = 101 + 100*(sind(2*lon));
```

Define data to control the color of each marker.

```
C = cosd(4*lon);
```

Plot the data on a geographic scatter plot, specifying the marker size data and the color data. Specify the marker as a triangle, rather than the default circle.

```
geoscatter(lat,lon,A,C,'^')
```



Create Geographic Scatter Plot Specifying Basemap

Set up latitude and longitude data.

```
lon = (-170:10:170);
lat = 50 * cosd(3*lon);
```

Define the data that controls the area of each marker.

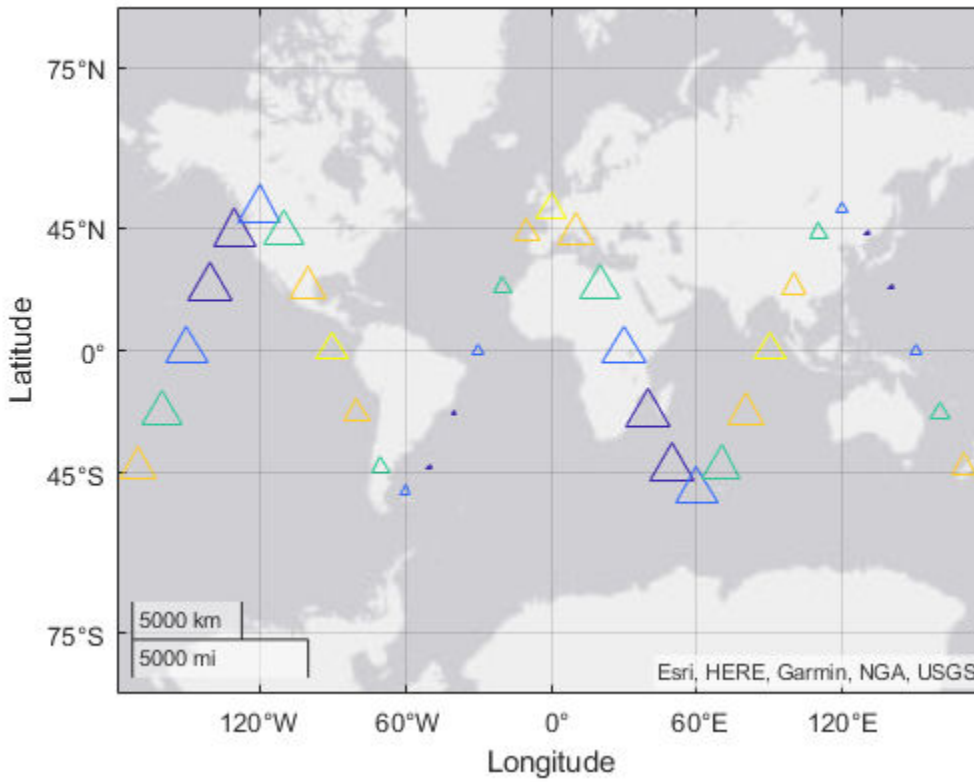
```
A = 101 + 100*(sind(2*lon));
```

Define the data that controls the color of each marker.

```
C = cosd(4*lon);
```

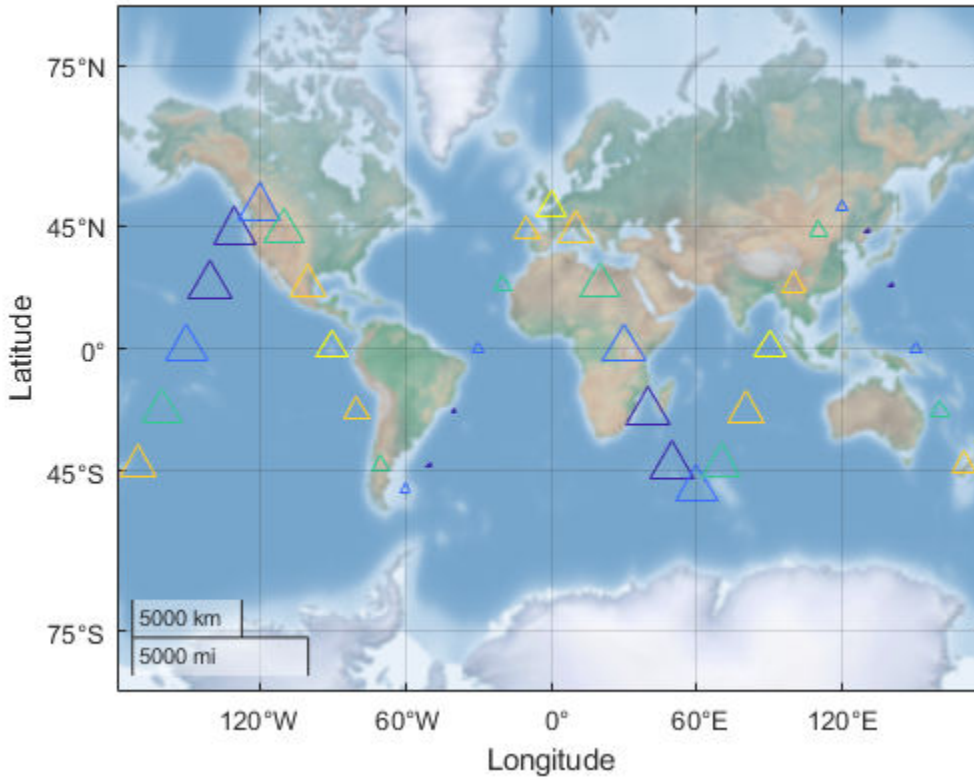
Create the scatter plot on a set of geographic axes, specifying the marker size data and the color data. The example specifies the marker as a triangle, rather than the default circle.

```
geoscatter(lat,lon,A,C,'^')
```



Change the basemap of the geographic scatter plot.

```
geobasemap colorterrain
```



Input Arguments

`lat` — Latitude coordinates in degrees
real, numeric, finite vector within the range [-90 90]

Latitude coordinates in degrees, specified as a real, numeric, finite vector within the range `[-90 90]`. The vector can contain embedded `NaN`s. `lat` must be the same size as `lon`.

Example: `[43.0327 38.8921 44.0435]`

Data Types: `single` | `double`

`lon` — Longitude coordinates in degrees

real, numeric, finite vector

Longitude coordinates in degrees, specified as a real, numeric, finite vector. The vector can contain embedded `NaN`s. `lon` must be the same size as `lat`.

Example: `[-107.5556 -77.0269 -72.5565]`

Data Types: `single` | `double`

`A` — Marker sizes in points squared

`36` (default) | scalar | vector | `[]`

Marker sizes in points squared, specified in one of these forms:

- Scalar — Uniform marker size. For example, `A = 100` creates all markers with an area of 100 points squared.
- Vector — Different marker size for each data point. The vector must be the same length as `lat` and `lon`.
- Empty brackets `[]` — Default marker size with an area of 36 points squared. Use this option if you want to specify the color input argument, but use the default marker area; for example, `geoscatter(lat,lon,[],c)`.

The `SizeData` property of the scatter object stores the marker sizes.

Example: `50`

Example: `[36 25 25 17 46]`

`C` — Marker color

three-column matrix of RGB triplets | vector | RGB triplet | `ColorSpec` value | ...

Marker color, specified as a MATLAB `ColorSpec (Color Specification)` value, a vector of `ColorSpecs`, or a three-column matrix of RGB triplets. Use a vector or three-column matrix to specify different marker colors for each data point. When using a vector, `geoscatter` performs a linear mapping of the values in `C` to colors in the colormap. Whether specifying a vector or three-column matrix, the vector or matrix must be the same length as `lat` and `lon`. If you specify an RGB triplet, the intensities must be in the range `[0,1]`.

Example: `'green'`

Example: `'g'`

Example: `[0 1 0]`

When you specify marker colors, `geoscatter` sets the `MarkerFaceColor` property of the `Scatter` object to `'flat'` and stores the marker colors in the `CData` property.

`M` — Marker symbol
`'o'` (default) | `'+'` | `'*'` | `'.'` | `'x'` | `'s'` | ...

Marker symbol, specified as a character vector or string containing a marker specifier. For more about marker specifiers, see `LineSpec (Line Specification)`.

When you specify marker type, `geoscatter` sets the `Marker` property of the `Scatter` object.

`'filled'` — Option to fill the interior of the markers
`'filled'`

Option to fill the interior of the markers, specified as `'filled'`. Use this option with markers that have a face, for example, `'o'` or `'square'`.

`gx` — Parent geographic axes object
`GeographicAxes` object

Parent geographic axes object, specified as a `GeographicAxes` object. You can modify the appearance and behavior of a `GeographicAxes` object by setting its properties. For a list of properties, see GeographicAxes Properties.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The scatter object properties listed here are only a subset. For a complete list, see Scatter.

Example: `geoscatter(lat,lon,'filled','MarkerFaceAlpha',.5)` creates filled, semi-transparent markers.

`Marker` — Marker type
`'o'` (default) | `'+'` | `'*'` | `'.'` | `'x'` | ...

Marker symbol, specified as a character vector or string containing a marker specifier. For more about marker specifiers, see `LineSpec (Line Specification)`.

`MarkerEdgeColor` — Marker outline color
`'flat'` (default) | `'none'` | RGB triplet | `ColorSpec` value | ...

Marker outline color, specified as `'none'`,`'flat'`, an RGB triplet, or a MATLAB `ColorSpec` value. The default value of `'flat'` uses colors from the `CData` property.

For a custom color, specify an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

When you specify marker edge color, `geoscatter` sets the `MarkerEdgeColor` property of the `Scatter` object and stores the marker colors in the `CData` property.

Example: `'green'`

Example: `'g'`

Example: `[0 1 0]`

`MarkerFaceColor` — Marker fill color
`'none'` (default) | `'flat'` | `'auto'` | RGB triplet | `ColorSpec` value | ...

Marker fill color, specified as `'none'`, `'flat'`, `'auto'`, an RGB triplet, or a MATLAB `ColorSpec (Color Specification)`.

- `'flat'` — `geoscatter` uses the `CData` values.
- `'auto'` — `geoscatter` uses the same color as the `Color` property for the axes.

Example: `[0.3 0.2 0.1]`

Example: `'green'`

`LineWidth` — Width of marker edge
`0.5` (default) | positive value

Width of marker edge, specified as a positive value in point units.

Example: `0.75`

Output Arguments

s — Geographic scatter plot
`Scatter` object

Geographic scatter plot, returned as a `Scatter` object. Use `s` to access and modify properties of the geographic scatter plot after it has been created.

Tips

- To customize the geographic axes containing your scatter object, obtain the object's parent, `gx = s.Parent`, and modify its properties. For a list of properties, see GeographicAxes Properties.
- If you have Mapping Toolbox, you can specify basemaps of your own choosing by using the `addCustomBasemap` function.

See Also

Functions
`geoaxes` | `geobasemap` | `geobubble` | `geolimits` | `geoplot` | `scatter`

Properties
GeographicAxes Properties | Scatter

Introduced in R2018b

geotickformat

Set or query geographic tick label format

Syntax

```
geotickformat fmt
geotickformat(fmt)
geotickformat(gx,fmt)
```

```
tickLabelFormat = geotickformat
tickLabelFormat = geotickformat(gx)
```

Description

`geotickformat fmt` sets the format of the latitude and longitude tick labels for the current geographic axes. When you use this syntax, you do not need to enclose the `fmt` argument in quotes. For example, the command `geotickformat dd` sets the tick label format to decimal degrees.

`geotickformat(fmt)` sets the tick label format for the current geographic axes. When you use this syntax, enclose the `fmt` input in single or double quotes, for example `geotickformat('dd')`.

`geotickformat(gx,fmt)` sets the tick label format for the geographic axes specified by `gx` instead of the current geographic axes.

`tickLabelFormat = geotickformat` returns the tick label format for the current geographic axes.

`tickLabelFormat = geotickformat(gx)` returns the tick label format for the geographic axes specified by `gx`.

Examples

Set Geographic Tick Label Format

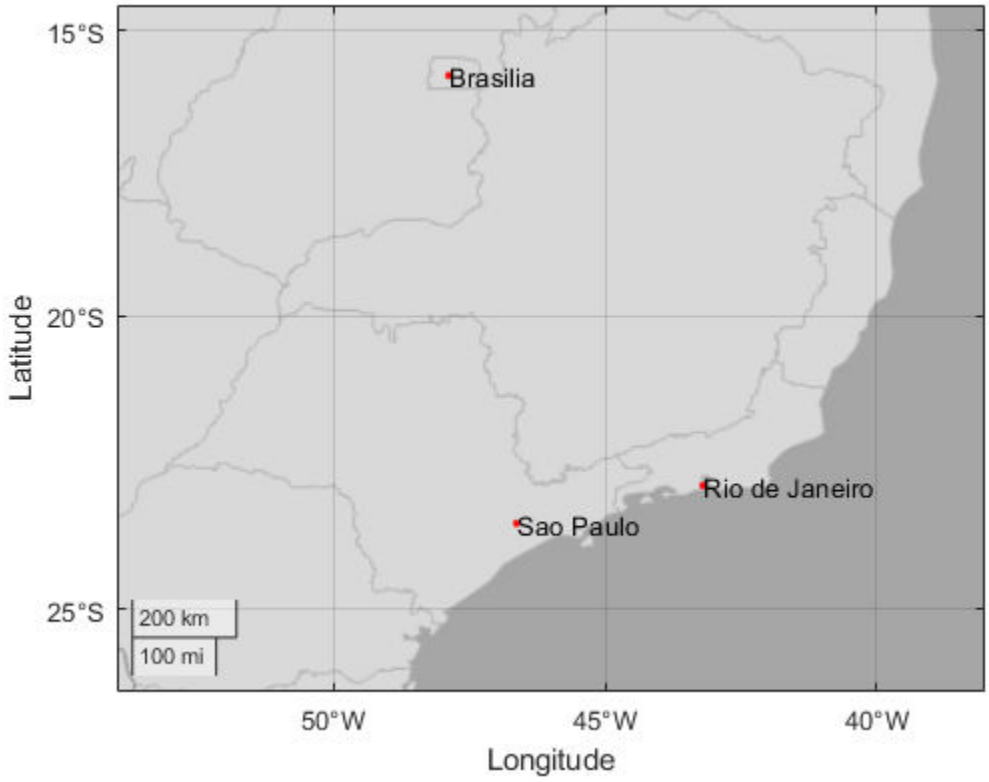
Set the tick label format for a geographic plot using `geotickformat`.

First, specify the coordinates of the three largest cities in Brazil.

```
lat = [-23.5500   -22.9083   -15.7939];
lon = [-46.6333   -43.1964   -47.8828];
```

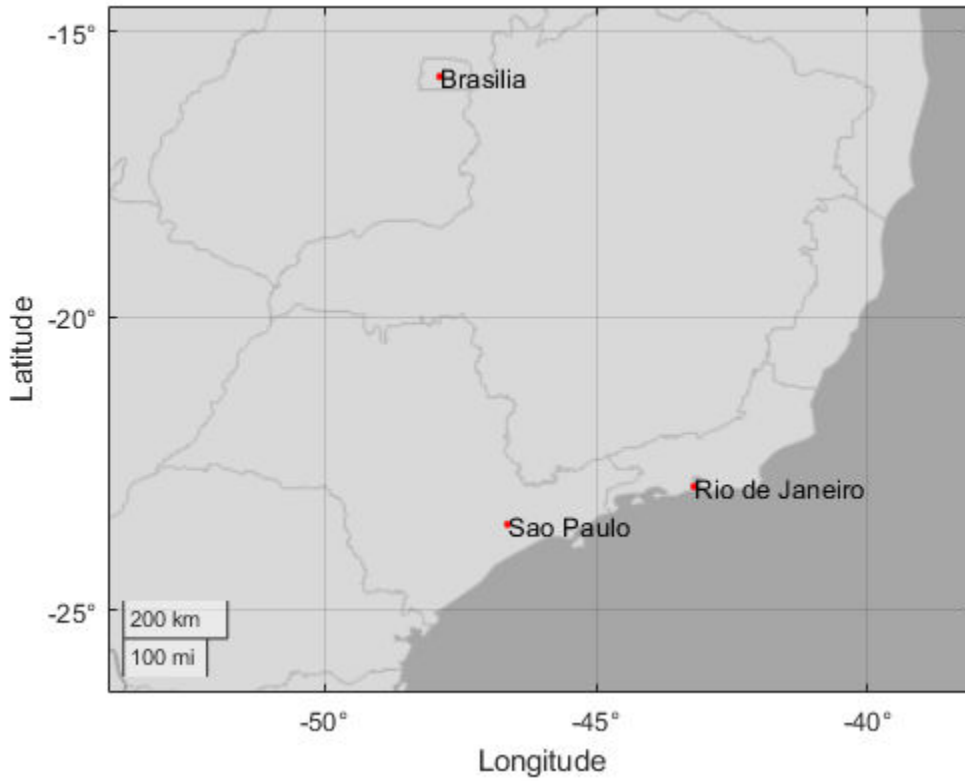
Plot the coordinates and add text labels.

```
geoplot(lat,lon,'r.','MarkerSize',8)
geolimits([-25 -16], [-54 -38])
text(lat,lon,["Sao Paulo","Rio de Janeiro","Brasilia"])
```



Change the tick label format to decimal degrees using `geotickformat`. The cardinal directions of south and west are indicated by a minus sign.

```
geotickformat -dd
```



Input Arguments

`fmt` — Tick label format
`'dms'` (default) | `'dd'` | `'dm'` | `-'dd'` | `'-dm'` | `'-dms'`

Tick label format, specified as one of the following values.

Format	Description	Example
`'dd'`	Decimal degrees plus compass direction	`23.5°N`
`'dm'`	Degrees and decimal minutes plus compass direction	`18°30'W`
`'dms'` (default)	Degrees, minutes, and decimal seconds plus compass direction	`110°06'18.5"E`
`'-dd'`	Decimal degrees with a minus sign (-) to indicate south and west	`-115.25°`
`'-dm'`	Degrees and decimal minutes with a minus sign (-) to indicate south and west	`-5°45.5'`
`'-dms'`	Degrees, minutes, and decimal seconds with a minus sign (-) to indicate south and west	`-3°21'05"`

Data Types: `char` | `string`

gx — Target geographic axes
`GeographicAxes` object

Target geographic axes, specified as a `GeographicAxes` object.

Tips

- The value returned from `geotickformat` is the value of the `TickLabelFormat` property of the geographic axes. You can override this format value on an individual axis by setting the `TickLabelFormat` property of the `GeographicRuler` object associated with the axis. In this case, the value returned by `geotickformat` does not match the format value of the axis or the actual display. Setting the `TickLabelFormat` of an individual axis is not recommended.

See Also

GeographicAxes Properties | GeographicRuler Properties | `geoaxes`

Introduced in R2019a

get

Query graphics object properties

Syntax

```
v = get(h)
v = get(h,propertyName)
v = get(h,propertyArray)
v = get(h,'default')
v = get(h,defaultTypeProperty)
v = get(groot,'factory')
v = get(groot,factoryTypeProperty)
```

Description

Note Do not use the `get` function on Java objects as it will cause a memory leak. For more information, see "Access Public and Private Data".

`v = get(h)` returns all properties and property values for the graphics object identified by `h`. `v` is a structure whose field names are the property names and whose values are the corresponding property values. `h` can be a single object or an `m`-by-`n` array of objects. If `h` is a single object and you do not specify an output argument, then MATLAB displays the information on the screen.

`v = get(h,propertyName)` returns the value for the specific property, `propertyName`. Use single quotes around the property name, for example, `get(h,'Color')`. If you do not specify an output argument, then MATLAB displays the information on the screen.

`v = get(h,propertyArray)` returns an *m*-by-*n* cell array, where *m* is equal to `length(h)` and *n* is equal to the number of property names contained in `propertyArray`.

`v = get(h,'default')` returns all default values currently defined on object `h` in a structure array. The field names are the object property names and the field values are

the corresponding property values. If you do not specify an output argument, MATLAB displays the information on the screen.

`v = get(h,defaultTypeProperty)` returns the current default value for a specific property. The argument `defaultTypeProperty` is the word `default` concatenated with the object type (e.g., `Figure`) and the property name (e.g., `Color`) in single quotes. For example, `get(groot,'defaultFigureColor')`.

`v = get(groot,'factory')` returns the factory-defined values of all user-settable properties in a structure array. The field names are the object property names and the field values are the corresponding property values. If you do not specify an output argument, MATLAB displays the information on the screen.

`v = get(groot,factoryTypeProperty)` returns the factory-defined value for a specific property. The argument `factoryTypeProperty` is the word `factory` concatenated with the object type (e.g., `Figure`) and the property name (e.g., `Color`) in single quotes. For example, `get(groot,'factoryFigureColor')`.

Examples

List All Property Values for Specific Object

Create a line plot and return the chart line object as `p`. List all the properties of the line and the current property values.

```
p = plot(1:10);
get(p)
```

```
    AlignVertexCenters: 'off'
            Annotation: [1x1 matlab.graphics.eventdata.Annotation]
          BeingDeleted: 'off'
            BusyAction: 'queue'
         ButtonDownFcn: ''
              Children: []
              Clipping: 'on'
                 Color: [0.9290 0.6940 0.1250]
             CreateFcn: ''
             DeleteFcn: ''
           DisplayName: ''
      HandleVisibility: 'on'
               HitTest: 'on'
```

```
         Interruptible: 'on'
             LineStyle: '-'
             LineWidth: 0.5000
                Marker: 'none'
       MarkerEdgeColor: 'auto'
       MarkerFaceColor: 'none'
            MarkerSize: 6
                Parent: [1x1 Axes]
         PickableParts: 'visible'
              Selected: 'off'
    SelectionHighlight: 'on'
                   Tag: ''
                  Type: 'line'
         UIContextMenu: []
              UserData: []
               Visible: 'on'
                 XData: [1 2 3 4 5 6 7 8 9 10]
             XDataMode: 'auto'
           XDataSource: ''
                 YData: [1 2 3 4 5 6 7 8 9 10]
           YDataSource: ''
                 ZData: [1x0 double]
           ZDataSource: ''
```

Query Specific Property of Specific Object

Create a line plot and return the chart line object as `p`. Use `get` to return the current value of the `LineWidth` property.

```
p = plot(1:10);
get(p,'LineWidth')
```

```
ans =

    0.5000
```

Query Set of Properties for Specific Object

Create a line plot with circle markers and return the chart line object as `p`. Use `get` to return the current values of the `LineWidth`, `Marker`, and `MarkerSize` properties for the object.

```
p = plot(1:10,'ro-');
props = {'LineWidth','Marker','MarkerSize'};
get(p,props)
```

```
ans =

    [0.5000]    'o'    [6]
```

Query Default Property Value on Root

Return the default value of the `LineWidth` property defined on the root for all line graphics objects.

```
get(groot,'DefaultLineLineWidth')
```

```
ans =

    0.5000
```

See Also

`findobj` | `gca` | `gcf` | `gco` | `set`

Topics

"Graphics Object Properties"

Introduced before R2006a

get

Query property values for `audioplayer` object

Syntax

```
Value = get(obj,Name)
Values = get(obj,{Name1,...,NameN})
Values = get(obj)
get(obj)
```

Description

`Value = get(obj,Name)` returns the value of the specified property for object *`obj`*.

`Values = get(obj,{Name1,...,NameN})` returns the values of the specified properties in a 1-by-*N* cell array.

`Values = get(obj)` returns a scalar structure that contains the values of all properties of *`obj`*. Each field name corresponds to a property name.

`get(obj)` displays all property names and their current values.

Examples

Create an `audioplayer` object from the example file `handel.mat` and query the object properties:

```
load handel.mat;
handelObj = audioplayer(y, Fs);

% Display all properties.
get(handelObj)

% Display only the SampleRate property.
get(handelObj, 'SampleRate')
```

```
% Create a cell array that contains
% values for two properties.
info = get(handelObj, {'BitsPerSample', 'NumChannels'});
```

Alternatives

To access a single property, you can use dot notation. Reference each property as though it is a field of a structure array. For example, find the value of the `TotalSamples` property for an object named `handelObj` (as created in the Example):

```
numSamples = handelObj.TotalSamples;
```

This command is exactly equivalent to:

```
numSamples = get(handelObj, 'TotalSamples');
```

See Also

`audioplayer` | `set`

get

Query property values for `audiorecorder` object

Syntax

```
Value = get(obj,Name)
Values = get(obj,{Name1,...,NameN})
Values = get(obj)
get(obj)
```

Description

`Value = get(obj,Name)` returns the value of the specified property for object *`obj`*.

`Values = get(obj,{Name1,...,NameN})` returns the values of the specified properties in a 1-by-*N* cell array.

`Values = get(obj)` returns a scalar structure that contains the values of all properties of *`obj`*. Each field name corresponds to a property name.

`get(obj)` displays all property names and their current values.

Examples

Create an `audiorecorder` object and query the object properties:

```
recorderObj = audiorecorder;

% Display all properties.
get(recorderObj)

% Display only the SampleRate property.
get(recorderObj, 'SampleRate')

% Create a cell array that contains
```

```
% values for two properties.
info = get(recorderObj, {'BitsPerSample', 'NumChannels'});
```

Alternatives

To access a single property, you can use dot notation. Reference each property as though it is a field of a structure array. For example, find the value of the `TotalSamples` property for an object named `recorderObj` (as created in the Example):

```
numSamples = recorderObj.TotalSamples;
```

This command is exactly equivalent to:

```
numSamples = get(recorderObj, 'TotalSamples');
```

See Also

`audiorecorder` | `set`

get (RandStream)

Random stream properties

Class

`RandStream`

`parallel.gpu.RandStream`

Syntax

```
get(s)
P = get(s)
P = get(s,'PropertyName')
```

Description

`get(s)` prints the list of properties for the random stream `s`.

`P = get(s)` returns all properties of `s` in a scalar structure.

`P = get(s,'PropertyName')` returns the property `'PropertyName'`.

See Also

`RandStream` | `parallel.gpu.RandStream` | `set (RandStream)`

get

Query `tscollection` properties

Syntax

```
tscinfo = get(tsc)
tscinfo = get(tsc,propname)
```

Description

`tscinfo = get(tsc)` returns all properties of the `tscollection` object `tsc`.

`tscinfo = get(tsc,propname)` returns the value of the property `propname`. For a full list of `tscollection` property names, see `tscollection`.

Examples

Display Properties

Create a `tscollection` object from two `timeseries` objects and display its properties.

```
ts1 = timeseries(rand(5,1),'Name','ts1');
ts2 = timeseries(rand(5,1),'Name','ts2');
tsc = tscollection({ts1,ts2});
tscinfo = get(tsc)
```

```
tscinfo = struct with fields:
        Name: 'unnamed'
        Time: [5x1 double]
    TimeInfo: [1x1 tsdata.timemetadata]
      Length: 5
         ts1: [1x1 timeseries]
         ts2: [1x1 timeseries]
```

Display only the `Length` property.

```
tsclength = get(tsc,'Length')
```

```
tsclength = 5
```

Input Arguments

`tsc` — Input `tscollection`
scalar

Input `tscollection`, specified as a scalar.

`propname` — Property name
character vector

Property name, specified as a character vector. For a full list of `tscollection` property names, see `tscollection`.

Data Types: `char`

See Also

`set` | `timeseries` | `tscollection`

Introduced before R2006a

getabstime

Convert `tscollection` time vector to cell array

Syntax

```
tsctime = get(tsc)
```

Description

`tsctime = get(tsc)` returns a cell array whose elements are the dates in the time vector associated with a `tscollection` object `tsin`.

Examples

Absolute Times

Create a `tscollection` object and define its start date.

```
tsc = tscollection(timeseries([3 6 8 0 10]'));
tsc.TimeInfo.StartDate = '10/27/2005 07:05:36';
```

Display the vector of absolute time values.

```
tsctime = getabstime(tsc)
```

```
tsctime = 5x1 cell array
    {'27-Oct-2005 07:05:36'}
    {'27-Oct-2005 07:05:37'}
    {'27-Oct-2005 07:05:38'}
    {'27-Oct-2005 07:05:39'}
    {'27-Oct-2005 07:05:40'}
```

Input Arguments

`tsc` — Input `tscollection`
scalar

Input `tscollection`, specified as a scalar.

Tips

- To define the `tscollection` time vector relative to a calendar date, set the `TimeInfo.StartDate` property to a valid date character vector. Valid dates can have the following formats:

Format	Example
`dd-mmm-yyyy HH:MM:SS`	`01-Mar-2000 15:45:17`
`dd-mmm-yyyy`	`01-Mar-2000`
`mm/dd/yy`	`03/01/00`
`mm/dd`	`03/01`
`HH:MM:SS`	`15:45:17`
`HH:MM:SS PM`	`3:45:17 PM`
`HH:MM`	`15:45`
`HH:MM PM`	`3:45 PM`
`mmm.dd,yyyy HH:MM:SS`	`Mar.01,2000 15:45:17`
`mmm.dd,yyyy`	`Mar.01,2000`
`mm/dd/yyyy`	`03/01/2000`

See Also

`timeseries` | `tscollection`

Introduced before R2006a

getappdata

Retrieve application-defined data

Syntax

```
val = getappdata(obj,name)
vals = getappdata(obj)
```

Description

Use this function to retrieve data stored using the `setappdata` function. Both of these functions provide a convenient way to share data between callbacks or between separate UIs.

`val = getappdata(obj,name)` returns a value stored in the graphics object, `obj`. The name identifier, `name`, uniquely identifies the value to retrieve.

`vals = getappdata(obj)` returns all values stored in the graphics object with their name identifiers.

Examples

Store and Retrieve Date and Time

Create a figure window.

```
f = figure;
```

Get the current date and time as separate variables.

```
dt = fix(clock);
currdate = dt(1:3);
currtime = dt(4:6);
```

Store `currdate` and `currtime` using the `setappdata` function.

```
setappdata(f,'todaysdate',currdate);
setappdata(f,'presenttime',currtime);
```

Retrieve the date information.

```
getappdata(f,'todaysdate')
```

```
ans =

    2014  12    23
```

Retrieve all data associated with figure `f`.

```
getappdata(f)
```

```
ans =

     todaysdate: [2014 12 23]
    presenttime: [16 51 5]
```

Input Arguments

`obj` — Graphics object containing the value
figure | uipanel | uibuttongroup | uicontrol | ...

Graphics object containing the value, specified as any graphics object. This is the same graphics object passed to `setappdata` during the storage operation.

`name` — Name identifier
character vector

Name identifier, specified as a character vector. This is the same name identifier passed to `setappdata` during the storage operation.

Data Types: `char`

Output Arguments

`val` — Stored value
any MATLAB data type

Stored value, returned as the same value and data type that was originally stored.

`vals` — All values stored in the graphics object with name identifiers
structure

All values stored in the graphics object with name identifiers, returned as a structure. Each field in the structure corresponds to a stored value. The field names of the structure correspond to the name identifiers assigned when each value was stored.

See Also

`guidata` | `isappdata` | `rmappdata` | `setappdata`

Topics

"Share Data Among Callbacks"

Introduced before R2006a

getaudiodata

Store recorded audio signal in numeric array

Syntax

```
y = getaudiodata(recorder)
y = getaudiodata(recorder,dataType)
```

Description

`y = getaudiodata(recorder)` returns recorded audio data associated with `audiorecorder` object `recorder` in a `double` array `y`.

`y = getaudiodata(recorder,dataType)` converts the signal data to the specified data type.

Examples

Get Data from Audio Recorder Object

Create a recording using the `audiorecorder` object and then get the audio signal as a numeric array of different data types.

Create an `audiorecorder` object and record a five second audio clip from your microphone.

```
recObj = audiorecorder;
disp('Start speaking.')
```

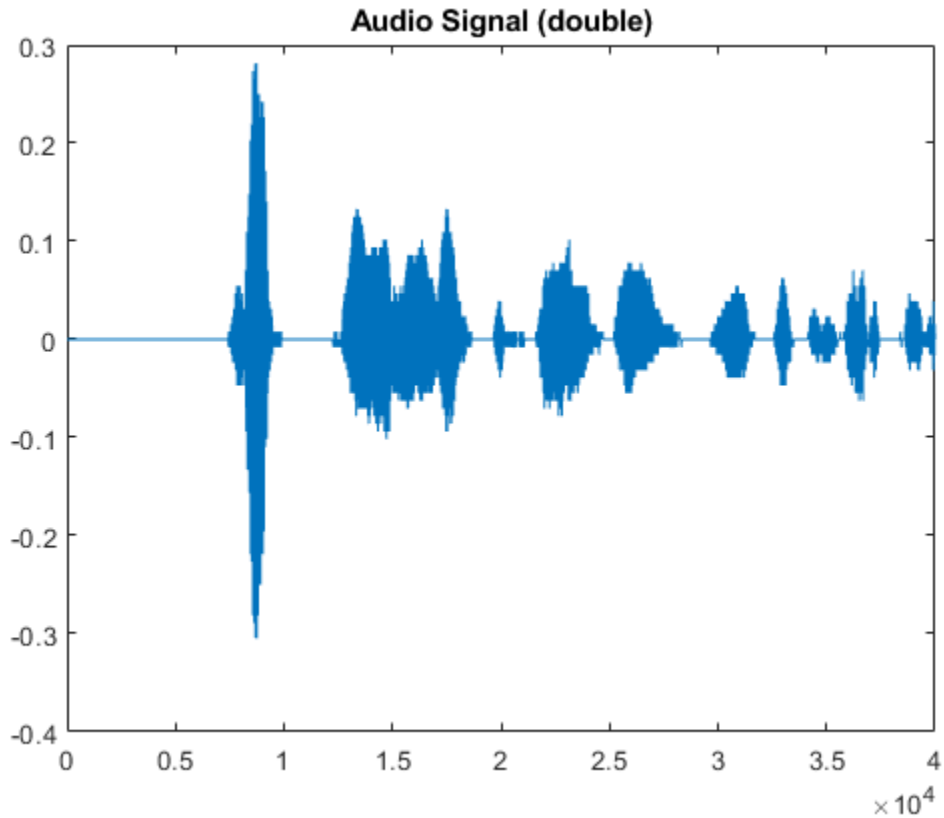
```
Start speaking.
```

```
recordblocking(recObj,5);
disp('End of Recording.');
```

```
End of Recording.
```

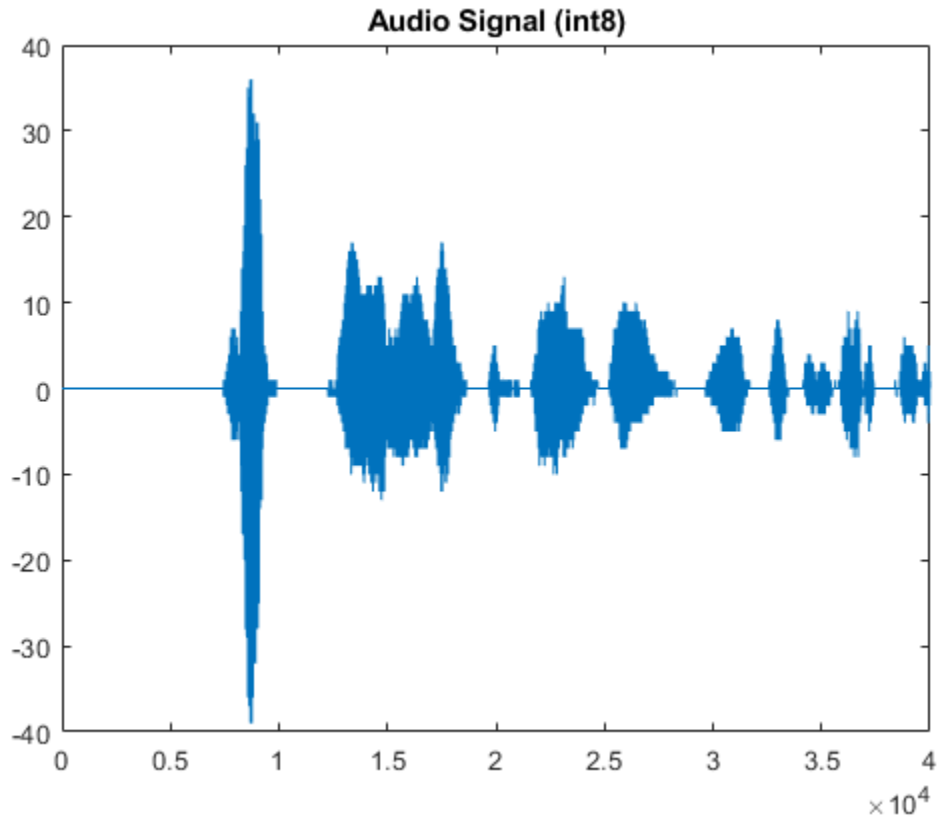
Get the audio signal as a `double` array and plot the data.

```
doubleArray = getaudiodata(recObj);
plot(doubleArray);
title('Audio Signal (double)');
```



Get the audio signal as an `int8` array and plot the data. Based on the data type specified, the same audio signal is returned with a different range of values . In this case, the values in the `int8` array can span between `-128` and `127`.

```
int8Array = getaudiodata(recObj,'int8');
plot(int8Array);
title('Audio Signal (int8)');
```



Input Arguments

`recorder` — Audio recorder object
`audiorecorder` object

Audio recorder object, specified as an `audiorecorder` object. Use the `audiorecorder` function to create the object.

`dataType` — Data type
`'double'` (default) | `'single'` | `'int16'` | `'int8'` | `'uint8'`

Data type of the output audio signal, specified as one of these values: `'double'`, `'single'`, `'int16'`, `'int8'`, or `'uint8'`.

Data Types: `char` | `string`

Output Arguments

y — Audio signal data
numeric array

Audio signal data, returned as a numeric array.

The number of columns in `y` depends on the number of channels in the recording: one for mono and two for stereo.

The value range of `y` depends on the specified `dataType`.

Data Type	Sample Value Range
`int8`	-128 to 127
`uint8`	0 to 255
`int16`	-32768 to 32767
`single`	-1 to 1
`double`	-1 to 1

See Also

`audiorecorder`

Topics

"Record and Play Audio"
"Supported Video and Audio File Formats"

Introduced in R2006a

GetCharArray

Character array from Automation server

Syntax

IDL Method Signature

```
HRESULT GetCharArray([in] BSTR varName, [in] BSTR Workspace,
    [out, retval] BSTR *mlString)
```

Microsoft Visual Basic Client

```
GetCharArray(varname As String, workspace As String) As String
```

MATLAB Client

```
str = GetCharArray(h,'varname','workspace')
```

Description

`str = GetCharArray(h,'varname','workspace')` gets the character array stored in `varname` from the specified *`workspace`* of the server attached to handle `h` and returns it in `str`. The values for *`workspace`* are `base` or `global`.

Examples

Get Character Input in Visual Basic® .NET Client

This example shows how to pass character data between MATLAB® and a Visual Basic® .NET client. Create an application with the following code.

```
type getchararray.vb
```

```
Dim Matlab As Object
Dim S As String
Matlab = CreateObject("matlab.application")
MsgBox("In MATLAB, type" & vbCrLf _
    & "str='new text';")
Try
    S = Matlab.GetCharArray("str", "base")
    MsgBox("str = " & S)
Catch ex As Exception
    MsgBox("You did not set 'str' in MATLAB")
End Try
```

Open the MATLAB window, then type:

```
str='new text';
```

Click **OK**. The Visual Basic `MsgBox` command displays what you typed in MATLAB.

Get Character Input in VBA Client

This example shows how to pass character data between MATLAB® and a VBA client. Create an application with the following code.

```
type getchararray.vba
```

```
Dim Matlab As Object
Dim S As String
Set Matlab = CreateObject("matlab.application")
MsgBox("In MATLAB, type" & vbCrLf & "str='new text';")
S = Matlab.GetCharArray("str", "base")
MsgBox("str = " & S)
```

Open the MATLAB window, then type:

```
str='new text';
```

Click **OK**. The VBA `MsgBox` command displays what you typed in MATLAB.

See Also

`GetVariable` | `GetWorkspaceData` | `PutCharArray`

Topics

"Conversion of COM Types to MATLAB Types"
"Conversion of MATLAB Types to COM Types"

Introduced before R2006a

getenv

Environment variable

Syntax

```
value = getenv(name)
```

Description

`value = getenv(name)` searches the operating system environment variable list for text of the form `name=value`.

On UNIX platforms, the shell you use to start MATLAB determines the operating system environment. For example, starting MATLAB on a Mac platform from the `Applications` folder creates a different shell environment from starting MATLAB from Terminal.

Examples

Display System Root Directory on Windows

```
res = getenv('SYSTEMROOT')
```

```
res = 'C:\windows'
```

Input Arguments

name — Environment variable name
string | character vector

Environment variable name, specified as a string or a character vector.

Example: `'PATH'`

Output Arguments

`value` — Environment variable value
character vector | `[]`

Environment variable value, returned as a character vector. If `name` is not found, then MATLAB returns an empty matrix.

See Also

`computer` | `path` | `pwd` | `setenv` | `ver`

Introduced before R2006a

getfield

Field of structure array

Syntax

```
value = getfield(S,field)
value = getfield(S,field1,...,fieldN)
value = getfield(S,idx,field1,...,fieldN)
value = getfield(S,idx,field1,idx1,...,fieldN,idxN)
```

Description

`value = getfield(S,field)` returns the value in the specified field of the structure `S`. For example, if `S.a = 1`, then `getfield(S,'a')` returns `1`.

As an alternative to `getfield`, use dot notation, `value = S.field`. Dot notation is typically more efficient.

If `S` is nonscalar, then `getfield` returns the value in the first element of the array, equivalent to `S(1).field`.

`value = getfield(S,field1,...,fieldN)` returns the value stored in a nested structure. For example, if `S.a.b.c = 1`, then `getfield(S,'a','b','c')` returns `1`.

`value = getfield(S,idx,field1,...,fieldN)` specifies the element of the structure array. For example, if `S(3,4).a = 1`, then `getfield(S,{3,4},'a')` returns `1`.

`value = getfield(S,idx,field1,idx1,...,fieldN,idxN)` specifies elements of fields. For example, if `S.a(2) = 1`, then `getfield(S,'a',{2})` returns `1`. Similarly, if `S(3,4).a(2).b = 1`, then `getfield(S,{3,4},'a',{2},'b')` returns `1`.

Examples

Access Field of Scalar Structure

Get the value of a field from a structure returned by the `what` function. `what` returns a scalar structure with fields containing the path to the specified folder and the names of various kinds of files in the folder.

```
S = what('C:\Temp')
```

```
S = struct with fields:
        path: 'C:\Temp'
           m: {'testFunc1.m'}
       mlapp: {0×1 cell}
         mlx: {'testFunc2.mlx'}
         mat: {2×1 cell}
         mex: {0×1 cell}
         mdl: {0×1 cell}
         slx: {0×1 cell}
           p: {0×1 cell}
     classes: {0×1 cell}
    packages: {0×1 cell}
```

Return the names of all Live Scripts listed in the `mlx` field. When you use the `getfield` function, you can access a field of the structure returned by a function without using a temporary variable to hold that structure.

```
value = getfield(what('C:\Temp'),'mlx')
```

```
value = 1×1 cell array
    {'testFunc2.mlx'}
```

You also can access a field using dot notation.

```
value = S.mlx
```

```
value = 1×1 cell array
    {'testFunc2.mlx'}
```

Field of Nested Structure

Access a field of a nested structure. In a nested structure, a structure at any level can have fields that are structures, and other fields that are not structures.

First, create a nested structure.

```
S.a.b.c = 1;
S.a.b.d = 'two';
S.a.b.e = struct('f',[3 4],'g','five');
S.h = 50
```

```
S = struct with fields:
    a: [1x1 struct]
    h: 50
```

While `S` is a structure, the fields `S.a`, `S.a.b`, and `S.a.b.e` are also structures.

```
S.a
```

```
ans = struct with fields:
    b: [1x1 struct]
```

```
S.a.b
```

```
ans = struct with fields:
    c: 1
    d: 'two'
    e: [1x1 struct]
```

```
S.a.b.e
```

```
ans = struct with fields:
    f: [3 4]
    g: 'five'
```

Return the value of `S.a.b.d` using the `getfield` function. When you specify a comma-separated list of nested structure names, you must include the structures at every level between the structure at the top and the field name you specify. In this case, the comma-separated list of structure names is `'a','b'` and the field name is `'d'`.

```
value = getfield(S,'a','b','d')
```

```
value =
'two'
```

You also can use dot notation to access the same field.

```
value = S.a.b.d
```

```
value =
'two'
```

Field of Element of Structure Array

Get the value of a field from an element of a structure array returned by the `dir` function. `dir` returns a structure array whose elements each contain information about a file in the specified folder.

Return information about files in the folder `C:\Temp`. There are 5 files in the folder.

```
S = dir('C:\Temp')
```

```
S = 5×1 struct array with fields:
    name
    folder
    date
    bytes
    isdir
    datenum
```

To display information about the 5th file, index into `S`.

```
S(5)
```

```
ans = struct with fields:
       name: 'testFunc2.mlx'
     folder: 'C:\Temp'
       date: '19-Jul-2018 09:43:53'
      bytes: 2385
      isdir: 0
    datenum: 7.3726e+05
```

Return the name of the file described by the 5th element of S using the `getfield` function. When you use `getfield`, specify indices in a cell array.

```
value = getfield(S,{5},'name')
```

```
value =
'testFunc2.mlx'
```

As an alternative, index into the structure array, and then use dot notation to specify a field.

```
value = S(5).name
```

```
value =
'testFunc2.mlx'
```

Indices of Nested Structure Array

Access a field of a nested structure, in which the structures at some levels are structure arrays. In this example, S is a 1-by-2 structure array. The second element, `S(2)`, has a nested structure `a.b`, where b is a 1-by-3 structure array.

First, create a nested structure. After creating the structure using dot notation, create another nonscalar structure array using the `struct` function and add it as a field.

```
S.a = 1;
S(2).a.b = struct('d',{5,10,20});
S
```

```
S=2×1 struct
    a
```

```
S(2).a.b
```

```
ans=3×1 struct
    d
```

Return the value of d from the third element of b using the `getfield` function. You must specify the indices of both S and b using cell arrays.

```
value = getfield(S,{2},'a','b',{3},'d')
```

```
value = 20
```

You also can use dot notation to access the same field.

```
value = S(2).a.b(3).d
```

```
value = 20
```

Elements of Field

Create a structure with a field whose value is an array.

```
S.a = [5 10 15 20 25]
```

```
S = struct with fields:
    a: [5 10 15 20 25]
```

Return elements of the array using the `getfield` function. To return a subarray, specify indices after the name of the field. You must specify the indices within a cell array.

```
value = getfield(S,'a',{[2:4]})
```

```
value = 1×3

    10    15    20
```

You also can use dot notation and array indexing to access the same elements.

```
value = S.a(2:4)
```

```
value = 1×3

    10    15    20
```

Input Arguments

S — Structure array
structure array

Structure array. If S is nonscalar, then each element of S is a structure, and all elements have the same fields with the same names.

`field` — Field name
character vector | string scalar

Field name, specified as a character vector or string scalar.

`idx` — Indices
cell array of numeric values

Indices, specified as a cell array of numeric or logical values. Indices for S and fields 1 through N-1 specify elements of structure arrays. Indices for field N specify elements of the array in that field, which can be of any type.

Example: `getfield(S,{1,2},'a')` is equivalent to `S(1,2).a`.

Example: If `S.a = [5 10 20]`, then `getfield(S,'a',{[2,3]})` returns `[10 20]`.

See Also

`fieldnames` | `isfield` | `orderfields` | `rmfield` | `setfield` | `struct`

Topics

"Access Data in Structure Array"
"Generate Field Names from Variables"

Introduced before R2006a

getframe

Capture axes or figure as movie frame

Syntax

```
F = getframe
F = getframe(ax)
F = getframe(fig)
F = getframe( ___ ,rect)
```

Description

`F = getframe` captures the current axes as it appears on the screen as a movie frame. `F` is a structure containing the image data. `getframe` captures the axes at the same size that it appears on the screen. It does not capture tick labels or other content outside the axes outline.

`F = getframe(ax)` captures the axes identified by `ax` instead of the current axes.

`F = getframe(fig)` captures the figure identified by `fig`. Specify a figure if you want to capture the entire interior of the figure window, including the axes title, labels, and tick marks. The captured movie frame does not include the figure menu and tool bars.

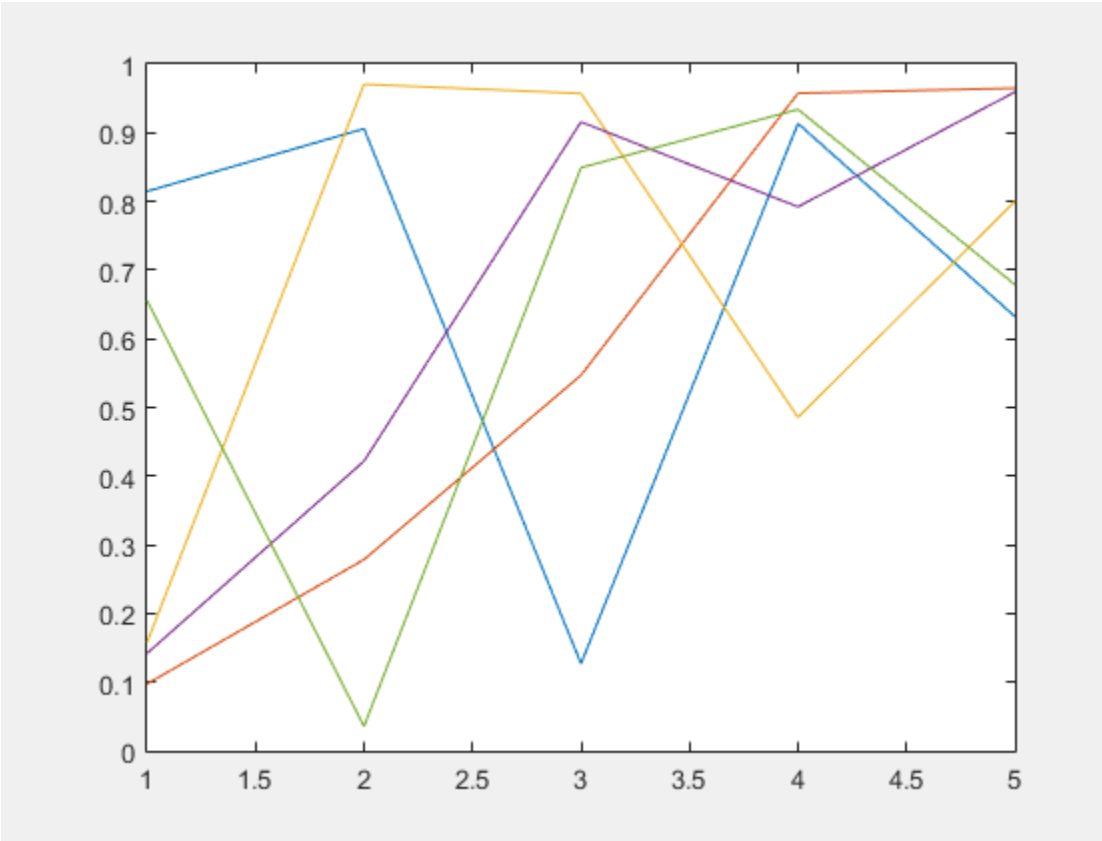
`F = getframe( ___ ,rect)` captures the area within the rectangle defined by `rect`. Specify `rect` as a four-element vector of the form `[left bottom width height]`. Use this option with either the `ax` or `fig` input arguments in the previous syntaxes.

Examples

Capture Contents of Current Axes

Create a plot of random data. Capture the axes and return the image data. `getframe` captures the interior of the axes and the axes outline. It does not capture content that extends beyond the axes outline.

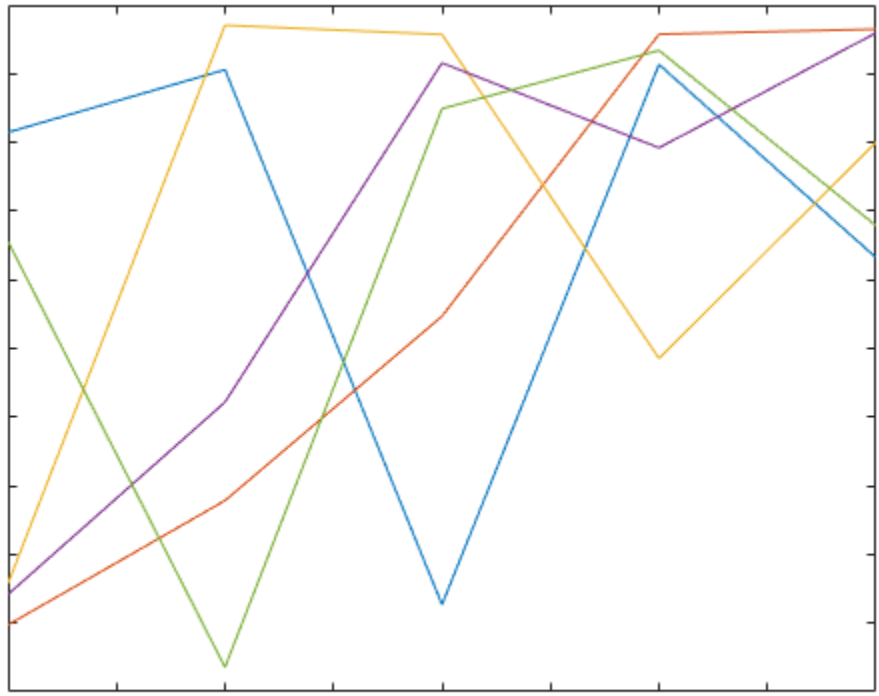
```
plot(rand(5))
F = getframe;
```



`F` is a structure with the field `cdata` that contains the captured image data.

Display the captured image data using `imshow`.

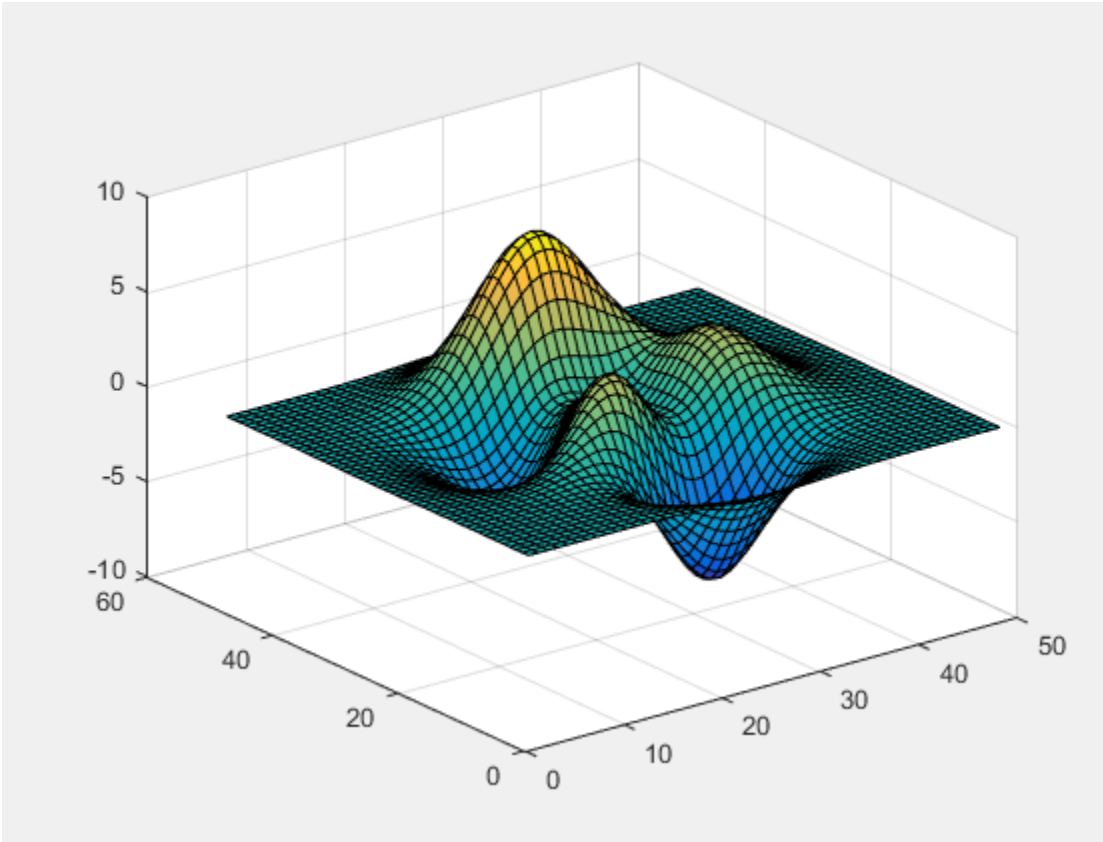
```
figure
imshow(F.cdata)
```



Capture Contents of Figure

Create a surface plot. Capture the interior of the figure window, excluding the menu and tool bars.

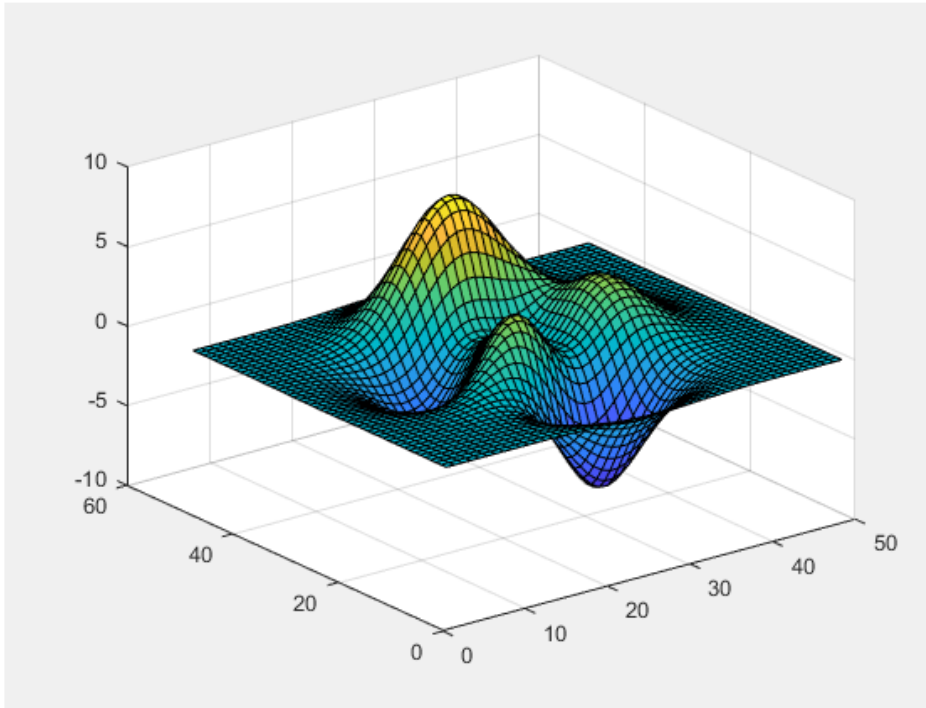
```
surf(peaks)
F = getframe(gcf);
```



F is a structure with the field `cdata` that contains the captured image data.

Display the captured image data using `imshow`.

```
figure
imshow(F.cdata)
```

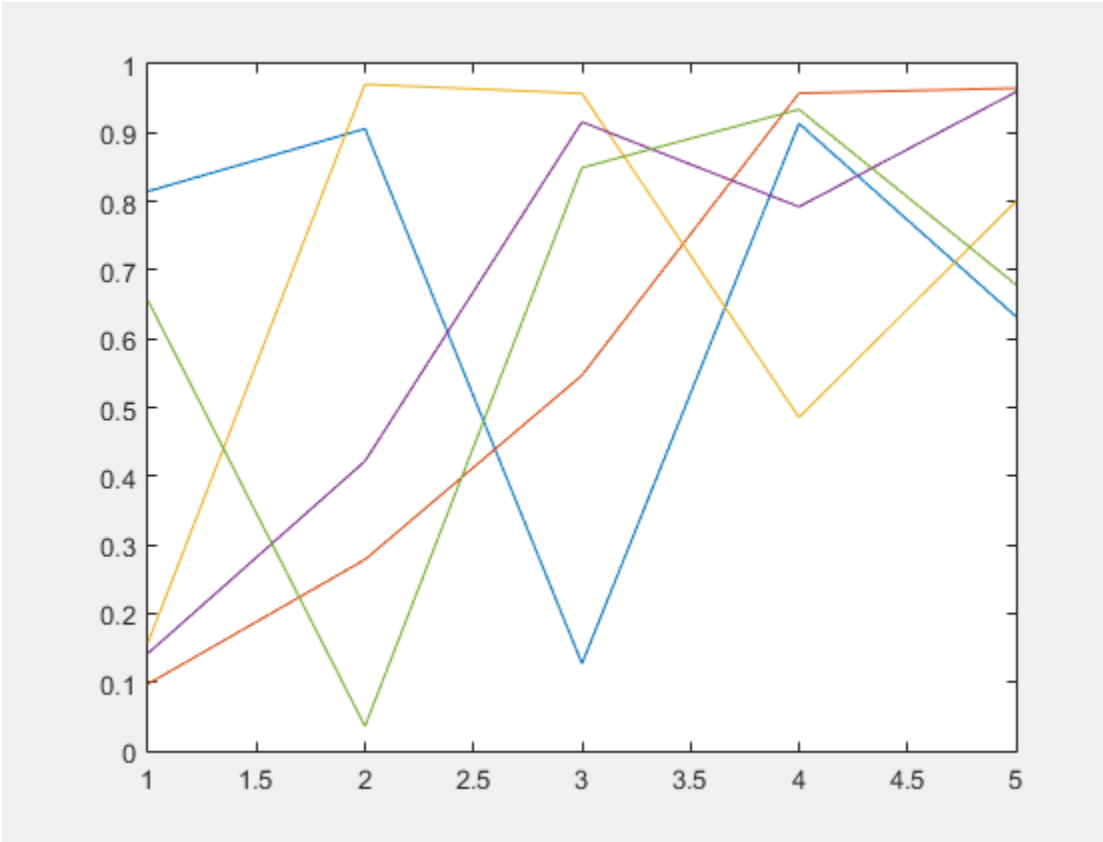


Specify Rectangular Region to Capture

Capture the interior of an axes plus a margin of 30 pixels in each direction. The added margin is necessary to include the tick labels in the capture frame. Depending on the size of the tick labels, the margin might need to be adjusted.

Create a plot of random data.

```
plot(rand(5))
```



Change the axes units to pixels and return the current axes position. The third and fourth elements of the position vector specify the axes width and height in pixels.

```
drawnow
ax = gca;
ax.Units = 'pixels';
pos = ax.Position
```

```
pos =

   73.8000   47.2000  434.0000  342.3000
```

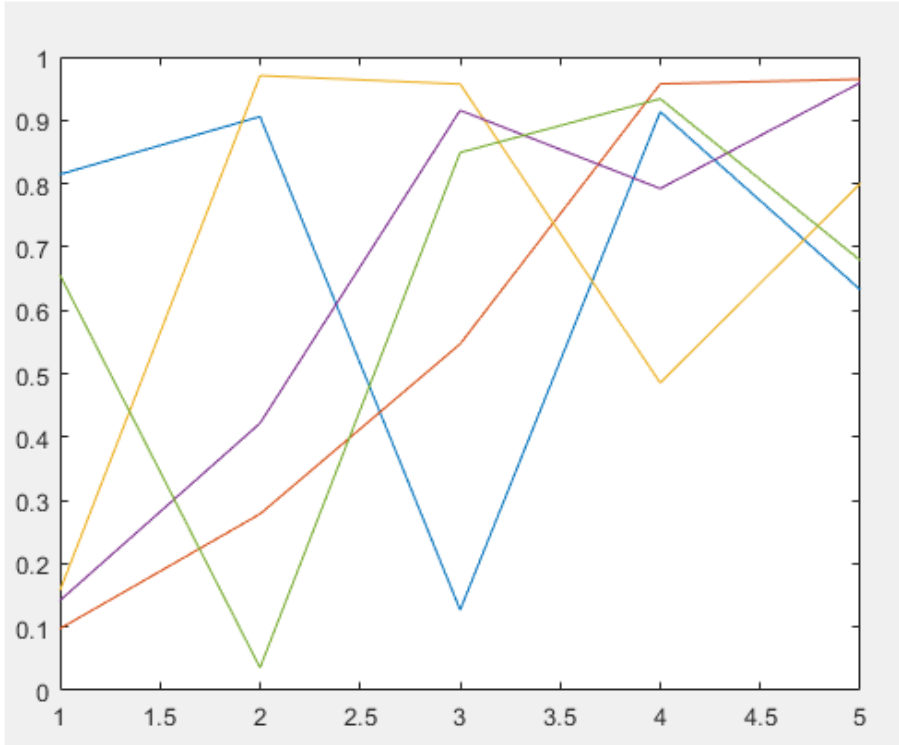
Create a four-element vector, `rect`, that defines a rectangular area covering the axes plus the desired margin. The first two elements of `rect` specify the lower left corner of the

rectangle relative to the lower left corner of the axes. The last two elements of `rect` specify the width and height of the rectangle. Reset the axes units to the default value of `'normalized'`.

```
marg = 30;
rect = [-marg, -marg, pos(3)+2*marg, pos(4)+2*marg];
F = getframe(gca,rect);
ax.Units = 'normalized';
```

Display the captured image data using `imshow`.

```
figure
imshow(F.cdata)
```

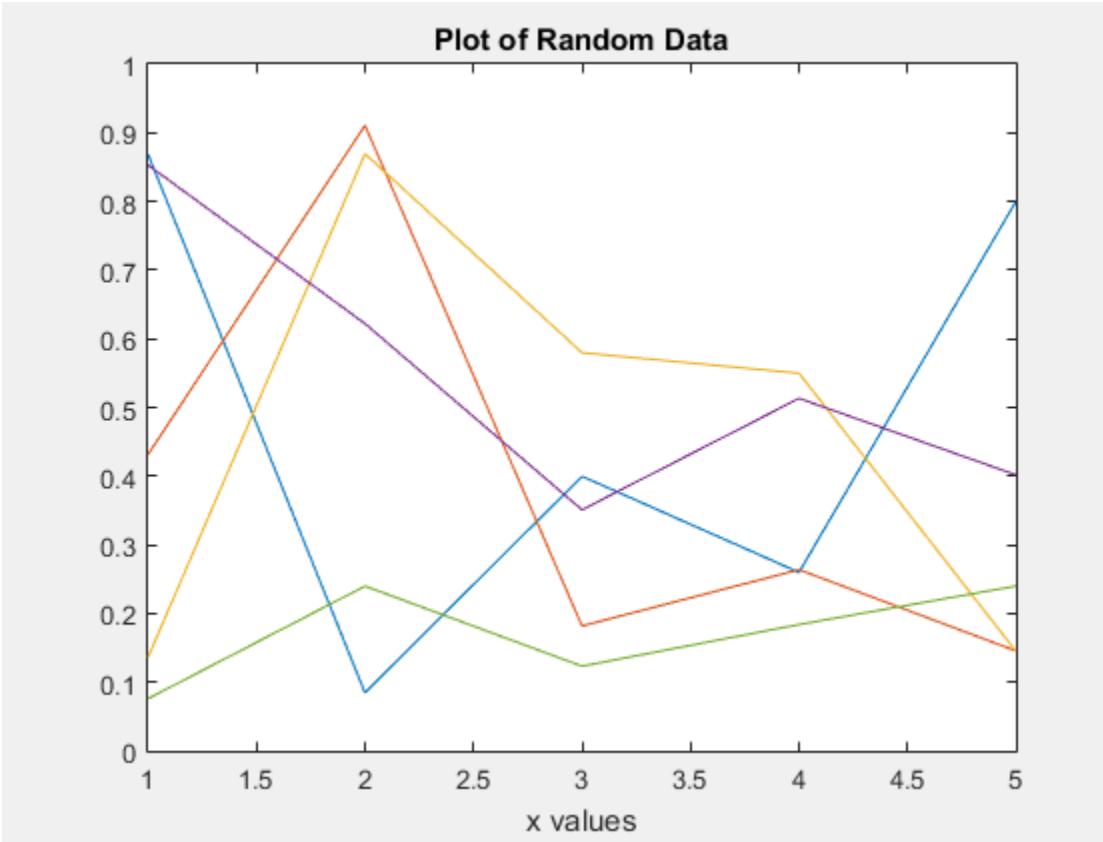


Calculate Region to Include Title and Labels

Calculate a margin around the axes so that the captured image data includes the title, axis labels, and tick labels.

Create a plot with a title and an *x*-axis label.

```
plot(rand(5))
xlabel('x values')
title('Plot of Random Data')
```



Change the axes units to pixels and store the `Position` and `TightInset` property values for the axes. The `TighInset` property is a four-element vector of the form `[left bottom right top]`. The values are the margins used around the axes for the tick values and text labels.

```
drawnow
ax = gca;
ax.Units = 'pixels';
pos = ax.Position;
ti = ax.TightInset;
```

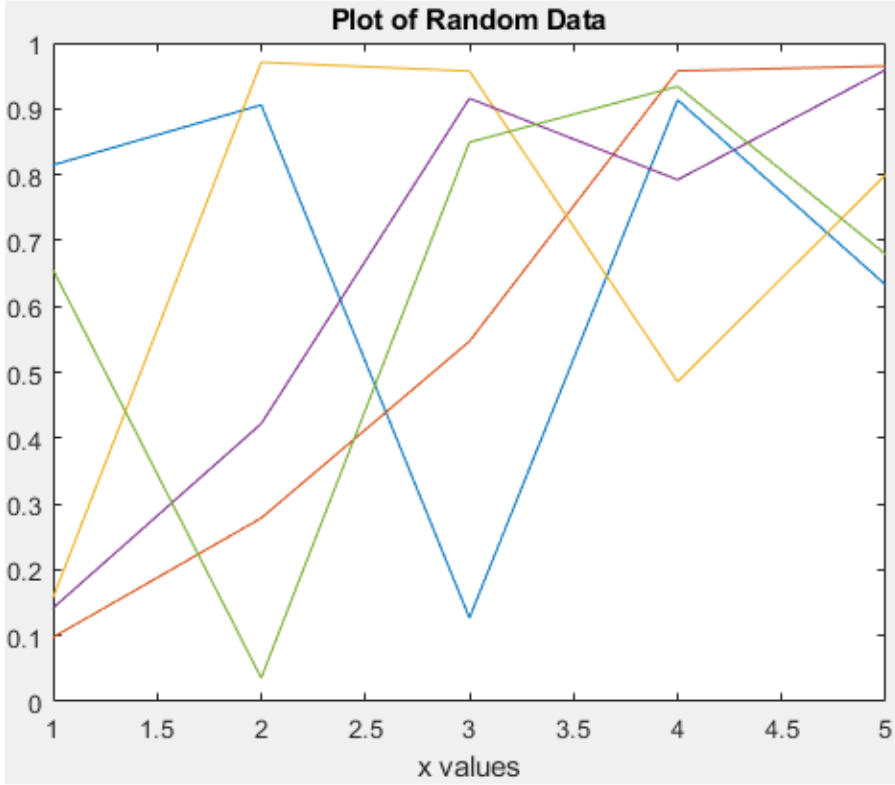
Create a four-element vector, `rect`, that defines a rectangular area covering the axes plus the automatically calculated margin. The first two elements of `rect` specify the lower left

corner of the rectangle relative to the lower left corner of the axes. The last two elements of `rect` specify the width and height of the rectangle.

```
rect = [-ti(1), -ti(2), pos(3)+ti(1)+ti(3), pos(4)+ti(2)+ti(4)];
F = getframe(ax,rect);
```

Display the captured image data using `imshow`.

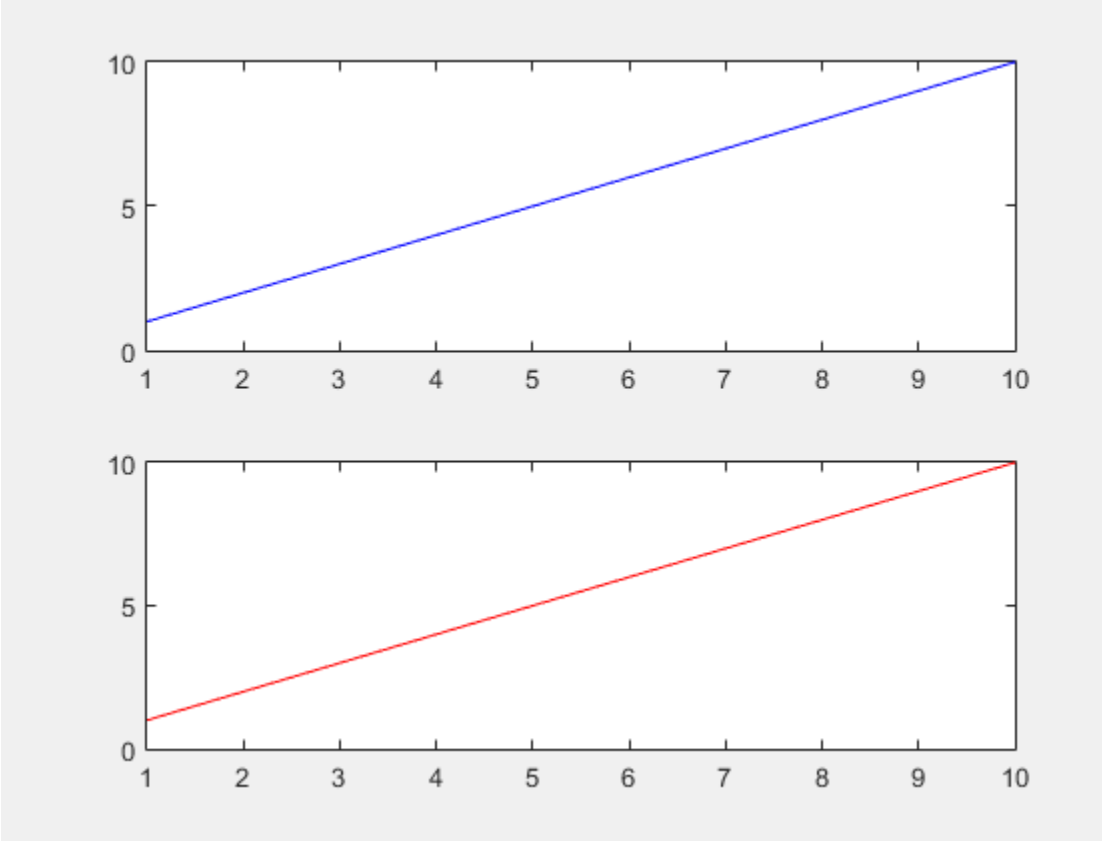
```
figure
imshow(F.cdata)
```



Capture Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Plot a line in each axes.

```
tiledlayout(2,1)
ax1 = nexttile;
plot(1:10,'b')
ax2 = nexttile;
plot(1:10,'r')
```

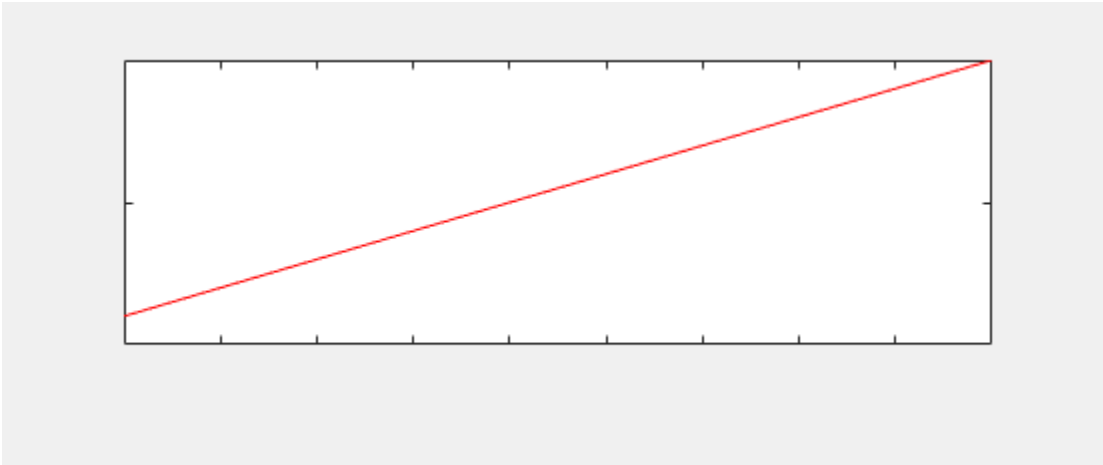


Capture the contents of the lower axes. `getframe` captures the interior and border of the plot. It does not capture tick values or labels that extend beyond the outline of the plot.

```
F = getframe(ax2);
```

Display the captured image data using `imshow`.

```
figure
imshow(F.cdata)
```



Record Frames and Play Movie

Record frames of the `peaks` function vibrating by using `getframe` in a loop. Preallocate an array to store the movie frames.

```
Z = peaks;
surf(Z)
axis tight manual
ax = gca;
ax.NextPlot = 'replaceChildren';

loops = 40;
F(loops) = struct('cdata',[],'colormap',[]);
for j = 1:loops
    X = sin(j*pi/10)*Z;
    surf(X,Z)
    drawnow
    F(j) = getframe(gcf);
end
```

Playback the movie two times.

```
fig = figure;
movie(fig,F,2)
```

Input Arguments

ax — Axes to capture

`Axes` object | `GeographicAxes` object

Axes to capture, specified as an `Axes` object or a `GeographicAxes` object. Use this option if you want to capture an axes that is not the current axes.

`getframe` captures the content within the smallest rectangle that encloses the axes outline. If you want to capture all the tick values and labels, then use the `fig` input argument instead.

Example: `F = getframe(ax);`

fig — Figure to capture

figure object

Figure to capture, specified as a figure object. Use `gcf` to capture the current figure.

Example: `F = getframe(gcf);`

rect — Rectangular area to capture

four-element vector of the form `[left bottom width height]`

Rectangular area to capture, specified as a four-element vector of the form `[left bottom width height]` in pixels on page 1-5044. The `left` and `bottom` elements define the position of the lower left corner of the rectangle. The position is relative to the figure or axes that is specified as the first input argument to `getframe`. The `width` and `height` elements define the dimensions of the rectangle.

Specify a rectangle that is fully contained within the figure window.

Example: `F = getframe(gcf,[0 0 560 420]);`

Output Arguments

F — Movie frame

structure

Movie frame, returned as a structure with two fields:

- `cdata` — The image data stored as an array of `uint8` values. The size of the image data array depends on your screen resolution.
- `colormap` — The colormap. On true color systems, this field is empty.

Note These are some important considerations about the size of `cdata`:

- If you query the size of the region that `getframe` captures (either the figure, the axes, or the region specified by `rect`), the size in pixels on page 1-5044 might not match the number of elements in `cdata`. This difference is because the number of elements in `cdata` depends on your screen resolution (and operating system settings), but pixels in MATLAB might not correspond to the actual pixels on your screen.
- Starting in R2015b, if you are using a high-resolution system, then the size of `cdata` might be larger than in previous releases or on other systems.

More About

Pixels

Distances in pixels are independent of your system resolution on Windows and Macintosh systems:

- On Windows systems, a pixel is 1/96th of an inch.
- On Macintosh systems, a pixel is 1/72nd of an inch.

On Linux systems, the size of a pixel is determined by your system resolution.

Tips

- For the fastest performance when using `getframe`, make sure that the figure is visible on the screen. If the figure is not visible, `getframe` can still capture the figure, but performance can be slower.
- For more control over the resolution of the image data, use the `print` function instead. The `cdata` output argument with print returns the image data. The `resolution` input argument controls the resolution of the image.

See Also

`frame2im` | `im2frame` | `image` | `imshow` | `movie` | `print` | `writeVideo`

Introduced before R2006a

GetFullMatrix

Matrix from Automation server workspace

Syntax

IDL Method Signature

```
GetFullMatrix([in] BSTR varname, [in] BSTR workspace,
    [in, out] SAFEARRAY(double) *pr, [in, out] SAFEARRAY(double) *pi)
```

Microsoft Visual Basic Client

```
GetFullMatrix(varname As String, workspace As String,
    [out] XReal As Double, [out] XImag As Double
```

MATLAB Client

```
[xreal ximag] = GetFullMatrix(h,'varname','workspace',zreal,zimag)
```

Description

`[xreal ximag] = GetFullMatrix(h,'varname','workspace',zreal,zimag)` gets matrix stored in variable `varname` from the specified *`workspace`* of the server attached to handle `h`. The function returns the real part in `xreal` and the imaginary part in `ximag`. The values for *`workspace`* are `base` or `global`.

The `zreal` and `zimag` arguments are matrices of the same size as the real and imaginary matrices (`xreal` and `ximag`) returned from the server. The `zreal` and `zimag` matrices are commonly set to zero.

Use `GetFullMatrix` for 2-D matrices with values of type `double` only. Use `GetVariable` or `GetWorkspaceData` for other types. To use higher dimensional matrices, reshape the 2-D matrix after receiving it from MATLAB.

For VBScript clients, use the `GetWorkspaceData` and `PutWorkspaceData` functions to pass numeric data to and from the MATLAB workspace. These functions use the `variant`

data type instead of the `safearray` data type used by `GetFullMatrix` and `PutFullMatrix`. VBScript does not support `safearray`.

Examples

Read Complex Double Matrix in VBA Client

This example shows how to read a complex double matrix from a MATLAB Automation server.

```
type getfullmatrix.vba
```

```
Dim MatLab As Object
Dim Result As String
Dim XReal(4, 4) As Double
Dim XImag(4, 4) As Double
Dim i, j As Integer

Set MatLab = CreateObject("matlab.application")
Result = MatLab.Execute("M = rand(5);")
MsgBox("In MATLAB, type" & vbCrLf & "M(3,4)")
x = MatLab.GetFullMatrix("M", "base", XReal, XImag)
' Display element (3,4). The array in VBA
' is 0-based.
i = 2
j = 3
MsgBox("XReal(" & i + 1 & "," & j + 1 & ")" & _
    " = " & XReal(i, j))
```

Read Complex Double Matrix in VB .NET Client

This example shows how to read a complex double matrix from a MATLAB Automation server.

```
type getfullmatrix.vb
```

```
Dim MatLab As Object
Dim Result As String
Dim XReal(4, 4) As Double
```

```
Dim XImag(4, 4) As Double
Dim i, j As Integer

MatLab = CreateObject("matlab.application")
Result = MatLab.Execute("M = rand(5);")
MsgBox("In MATLAB, type" & vbCrLf & "M(3,4)")
MatLab.GetFullMatrix("M", "base", XReal, XImag)
' Display element (3,4). The array in VB .NET
' is 0-based.
i = 2
j = 3
MsgBox("XReal(" & i + 1 & "," & j + 1 & ")" & _
    " = " & XReal(i, j))
```

See Also

`Execute` | `GetVariable` | `GetWorkspaceData` | `PutFullMatrix`

Topics

"Conversion of COM Types to MATLAB Types"
"Conversion of MATLAB Types to COM Types"

Introduced before R2006a

getnext

Package: `matlab.mapreduce`

Get next value from ValueIterator

Syntax

```
X = getnext(ValIter)
```

Description

`X = getnext(ValIter)` returns the next available value in `ValIter`. Use the `hasnext` function to confirm availability of values in `ValIter` before calling `getnext`.

Examples

Get Values from ValueIterator in Reduce Function

Use the `hasnext` and `getnext` functions in a `while` loop within the reduce function to iteratively get values from the `ValueIterator` object. For example,

```
function MeanDistReduceFun(sumLenKey, sumLenIter, outKVStore)
    sumLen = [0 0];
    while hasnext(sumLenIter)
        sumLen = sumLen + getnext(sumLenIter);
    end
    add(outKVStore, 'Mean', sumLen(1)/sumLen(2));
end
```

Always call `hasnext` before `getnext` to confirm availability of a value. `mapreduce` returns an error if you call `getnext` with no remaining values in the `ValueIterator` object.

Input Arguments

`ValIter` — Intermediate value iterator
`ValueIterator` object

Intermediate value iterator, specified as a `ValueIterator` object. The `mapreduce` function automatically creates this object during execution. The second input to the reduce function specifies the variable name for the `ValueIterator` object, which is the variable name to use with the `hasnext` and `getnext` functions.

For more information, see `ValueIterator`.

See Also

`hasnext` | `mapreduce`

Topics

`ValueIterator`
"Build Effective Algorithms with MapReduce"

Introduced in R2014b

getpixelposition

Get component position in pixels

Note

Use this function only with GUIDE, or with apps created using the `figure` function.

Syntax

```
position = getpixelposition(handle)
position = getpixelposition(handle,recursive)
```

Description

`position = getpixelposition(handle)` gets the position, in pixels on page 1-5053, of the component specified by `handle`. MATLAB returns the `position` as a four-element vector that specifies the location and size of the component: [distance from left, distance from bottom, width, height].

`position = getpixelposition(handle,recursive)` gets the position as above. If `recursive` is true, the returned position is relative to the parent figure of `handle`.

Use the `getpixelposition` function only to obtain coordinates for children of figures created with the `figure` function, or for child containers of the figure (such as panels or button groups). Results are not reliable for children of axes or other graphics objects.

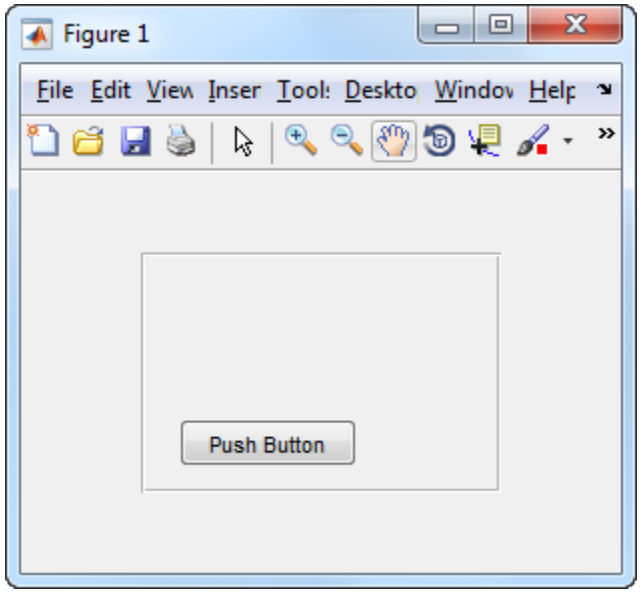
Examples

This example creates a push button within a panel, and then retrieves its position, in pixels, relative to the *panel*.

```
f = figure('Position',[300 300 300 200]);
p = uipanel('Position',[.2 .2 .6 .6]);
```

```
h1 = uicontrol(p,'Style','PushButton',...
     'Units','Normalized',...
     'String','Push Button',...
     'Position',[.1 .1 .5 .2]);
drawnow;
pos1 = getpixelposition(h1)
```

```
pos1 =

   18.6000   12.6000   88.0000   23.2000
```



The following statement retrieves the position of the push button, in pixels, relative to the *figure*.

```
pos1 = getpixelposition(h1,true)
```

```
pos1 =
      78.6000   52.6000   88.0000   23.2000
```

More About

Pixels

Distances in pixels are independent of your system resolution on Windows and Macintosh systems:

- On Windows systems, a pixel is 1/96th of an inch.
- On Macintosh systems, a pixel is 1/72nd of an inch.

On Linux systems, the size of a pixel is determined by your system resolution.

See Also

`figure` | `setpixelposition` | `uicontrol` | `uipanel`

getpoints

Return points that define animated line

Syntax

```
[x,y] = getpoints(an)
[x,y,z] = getpoints(an)
```

Description

`[x,y] = getpoints(an)` returns the *x* and *y* coordinates for the points that define the animated line specified by `an`. Create an animated line with the `animatedline` function.

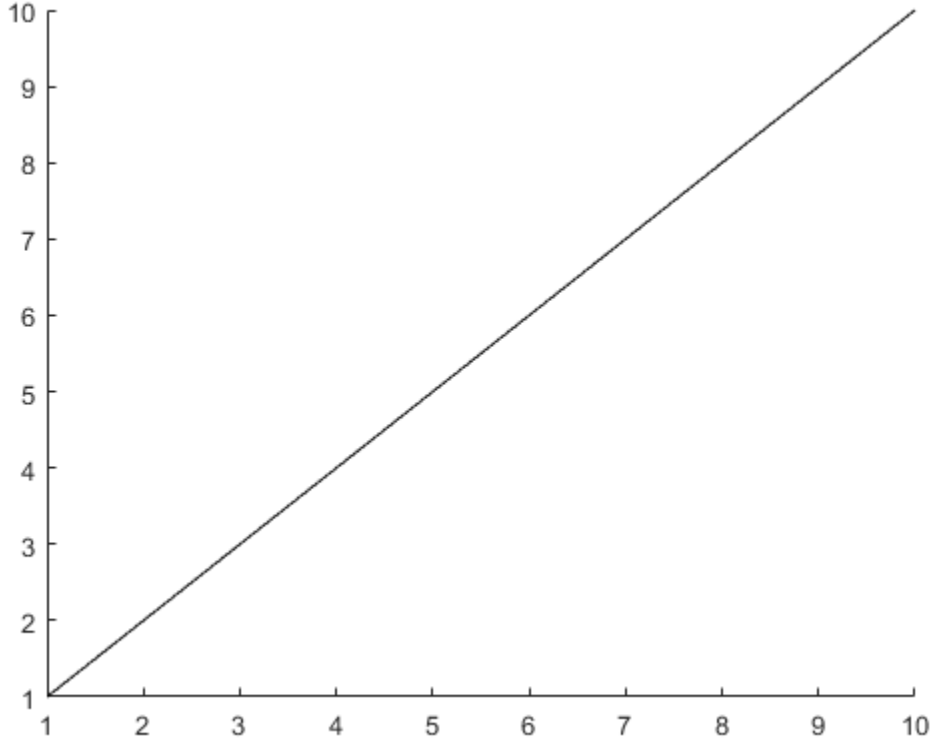
`[x,y,z] = getpoints(an)` returns the coordinates for the 3-D animated line specified by `an`. If the line does not have any *z* values, then `getpoints` returns `z` as a vector of zeros the same length as `x` and `y`.

Examples

Return Points from Animated Line

Create an animated line with 10 points. Then, return the points stored in the animated line.

```
h = animatedline(1:10,1:10);
```




```
[x,y] = getpoints(h)
```

```
x = 1×10

     1     2     3     4     5     6     7     8     9    10


y = 1×10

     1     2     3     4     5     6     7     8     9    10
```

Input Arguments

an — AnimatedLine object
AnimatedLine object

AnimatedLine object. Create an AnimatedLine object using the animatedline function.

Output Arguments

x — First coordinate values
vector

First coordinate values that define the animated line, returned as a vector. For Cartesian axes, the first coordinate is *x*-axis position. For geographic axes, the first coordinate is latitude in degrees.

y — Second coordinate values
vector

Second coordinate values that define the animated line, returned as a vector. For Cartesian axes, the second coordinate is *y*-axis position. For geographic axes, the second coordinate is longitude in degrees.

z — Third coordinate values
vector

Third coordinate values that define the 3-D animated line, returned as a vector. For Cartesian axes, the third coordinate is *z*-axis position.

See Also

Functions
addpoints | animatedline | clearpoints

Properties
AnimatedLine

Introduced in R2014b

getpref

Get custom preference value

Syntax

```
value = getpref(group,pref)
```

Description

`value = getpref(group,pref)` returns the value of the specified preference in the specified group. `value` can be any MATLAB data type, including numeric types, character vectors, cell arrays, structures, and objects. If `pref` specifies multiple preferences, `getpref` returns the values in a cell array. If the specified preference does not exist, MATLAB returns an error.

Examples

Get Value of Existing Preference

Add a preference called `version` to the `mytoolbox` group of preferences and then get the preference value.

```
addpref('mytoolbox','version','1.0')
getpref('mytoolbox','version')
```

```
ans =
    '1.0'
```

Input Arguments

group — Custom preference group name
character vector | string scalar

Custom preference group name, specified as a character vector or a string scalar.

Example: `'mytoolbox'`

Data Types: `char` | `string`

`pref` — Custom preference name
character vector | cell array of character vectors | string array

Custom preference name, specified as a character vector, a cell array of character vectors, or a string array.

Example: `'version'`

Example: `{'version','modifieddate','docpath'}`

Data Types: `char` | `string`

See Also

`addpref` | `ispref` | `rmpref` | `setpref` | `uigetpref` | `uisetpref`

Topics

"Preferences"

Introduced before R2006a

getsampleusingtime

Subset of `tscollection` data

Syntax

```
tscout = getsampleusingtime(tscin,timeval)
tscout = getsampleusingtime(tscin,
timeval,'AllowDuplicateTimes',true)
tscout = getsampleusingtime(tscin,starttime,endtime)
```

Description

`tscout = getsampleusingtime(tscin,timeval)` returns a `tscollection` object that contains the single data sample of an input `tscollection` corresponding to the time `timeval`.

`tscout = getsampleusingtime(tscin,
timeval,'AllowDuplicateTimes',true)` includes multiple data samples with the same time value specified in `timeval`.

`tscout = getsampleusingtime(tscin,starttime,endtime)` includes samples of `tscin` between the times `starttime` and `endtime`.

Examples

Subset of `tscollection`

Create a `tscollection` object from two `timeseries` objects and extract the data samples corresponding to the 2nd through 4th time values.

```
ts1 = timeseries(rand(5,1),'Name','ts1');
ts2 = timeseries(rand(5,1),'Name','ts2');
tscin = tscollection({ts1,ts2});
```

```
tscout = getsampleusingtime(tscin,2,4);
tscout.Time
```

```
ans = 3×1

     2
     3
     4
```

Input Arguments

`tscin` — Input `tscollection`

scalar

Input `tscollection`, specified as a scalar.

`timeval` — Sample time

scalar | vector

Sample time, specified as a numeric scalar, date character vector, or `datenum` scalar. Valid date character vectors can have the following forms:

Format	Example
dd-mmm-yyyy HH:MM:SS	01-Mar-2000 15:45:17
dd-mmm-yyyy	01-Mar-2000
mm/dd/yy	03/01/00
mm/dd	03/01
HH:MM:SS	15:45:17
HH:MM:SS PM	3:45:17 PM
HH:MM	15:45
HH:MM PM	3:45 PM
mmm.dd,yyyy HH:MM:SS	Mar.01,2000 15:45:17
mmm.dd,yyyy	Mar.01,2000
mm/dd/yyyy	03/01/2000

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `cell`

`starttime` — Start time
scalar | vector

Start time, specified as a numeric scalar, date character vector, or `datenum` scalar.

`endtime` — End time
scalar | vector

End time, specified as a numeric scalar, date character vector, or `datenum` scalar.

Tips

- If the time vector in `tscin` is not relative to a calendar date, then `starttime` and `endtime` must be numeric.
- If the time vector in `tscin` is relative to a calendar date, then `starttime` and `endtime` values must be date character vectors or `datenum` values.

See Also

`timeseries` | `tscollection`

Introduced before R2006a

getTag

Value of specified tag

Syntax

```
tagValue = getTag(t,tagID)
```

Description

`tagValue = getTag(t,tagID)` retrieves the value of the tag specified by `tagID` from the TIFF file associated with the `Tiff` object `t`.

Examples

Get Tag Value

Open a TIFF file and get the value of its tags in multiple ways.

Create a `Tiff` object for the file.

```
t = Tiff('example.tif','r');
```

Get the tag value for the `ImageWidth` tag by specifying the tag by its name.

```
tagval = getTag(t,'ImageWidth')
```

```
tagval = 600
```

Alternatively, specify the tag by its numeric identifier.

```
tagval1 = getTag(t,256)
```

```
tagval1 = 600
```

Another way to specify the tag is to use the `Tiff.TagID` structure. The names of the fields of the `Tiff.TagID` structure are valid tag names that contain the corresponding

tag numeric identifiers. For instance the field `Tiff.TagID.ImageWidth` contains the value `256`.

```
tagval2 = getTag(t,Tiff.TagID.ImageWidth)
```

```
tagval2 = 600
```

Close the `Tiff` object.

```
close(t);
```

Input Arguments

`t` — Tiff object
`Tiff` object

`Tiff` object representing a TIFF file. Use the `Tiff` function to create the object.

`tagID` — Tag ID
character vector | string scalar | numeric identifier

Tag ID of a `Tiff` object, specified as a character vector or string scalar, or a numeric identifier.

For example, you can specify `tagId` for the `ImageWidth` tag as any of these:

- Character vector or string scalar containing the tag name `'ImageWidth'`
- Numeric identifier `256` for the `ImageWidth` tag defined by the TIFF specification
- Field of the `Tiff.TagID` structure `Tiff.TagID.ImageWidth`.

The names of the fields of the `Tiff.TagID` structure are valid tag names that contain the corresponding tag numeric identifiers. For instance, the field `Tiff.TagID.ImageWidth` contains the value `256`. To see a list of all the tags along with their numeric identifiers, type `Tiff.TagID` in the command window.

Example: `'ImageWidth'`

Example: `256`

Example: `Tiff.TagID.ImageWidth`

Data Types: `double` | `char` | `string`

Algorithms

References

This function corresponds to the `TIFFGetField` function in the LibTIFF C API. To use this function, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff` | `Tiff.getTagNames` | `setTag`

Topics

"Importing Images"
"Exporting to Images"

Introduced in R2009b

Tiff.getTagNames

List of recognized TIFF tags

Syntax

```
tagNames = Tiff.getTagNames()
```

Description

`tagNames = Tiff.getTagNames()` returns a cell array of TIFF tags recognized by the `Tiff` object.

Examples

Get Tag Names

Retrieve a list of TIFF tags recognized by the `Tiff` object.

```
Tiff.getTagNames
```

```
ans = 73x1 cell array
    {'SubFileType'        }
    {'ImageWidth'         }
    {'ImageLength'        }
    {'BitsPerSample'      }
    {'Compression'        }
    {'Photometric'        }
    {'Thresholding'       }
    {'FillOrder'          }
    {'DocumentName'       }
    {'ImageDescription'   }
    {'Make'               }
    {'Model'              }
    {'StripOffsets'       }
    {'Orientation'        }
```

```
    {'SamplesPerPixel'    }
    {'RowsPerStrip'       }
    {'StripByteCounts'    }
    {'MinSampleValue'     }
    {'MaxSampleValue'     }
    {'XResolution'        }
    {'YResolution'        }
    {'PlanarConfiguration'}
    {'PageName'           }
    {'XPosition'          }
    {'YPosition'          }
    {'Group3Options'      }
    {'Group4Options'      }
    {'ResolutionUnit'     }
    {'PageNumber'         }
    {'TransferFunction'   }
      ⋮
```

See Also

`Tiff` | `getTag` | `setTag`

Topics

"Exporting Image Data and Metadata to TIFF Files"
"Reading Image Data and Metadata from TIFF Files"

Introduced in R2009b

gettimeseriesnames

Names of `timeseries` in `tscollection`

Syntax

```
tsnames = gettimeseriesnames(tsc)
```

Description

`tsnames = gettimeseriesnames(tsc)` returns a cell array of character vectors containing the names of the `timeseries` objects in the `tscollection` object `tsc`.

Examples

`timeseries` Names

Create a `tscollection` object from two `timeseries` objects. Then, display the names of the `timeseries` in the `tscollection`.

```
ts1 = timeseries([1.1 2.9 3.7 4.0 3.0]',1:5,...
                 'Name','Acceleration');
ts2 = timeseries([3.2 4.2 6.2 8.5 1.1]',1:5,...
                 'Name','Speed');
tsc = tscollection({ts1;ts2});
tsnames = gettimeseriesnames(tsc)
```

```
tsnames = 1x2 cell array
    {'Acceleration'}    {'Speed'}
```

Input Arguments

`tsc` — Input `tscollection`
scalar

Input `tscollection`, specified as a scalar.

See Also

`timeseries` | `tscollection`

Introduced before R2006a

gettsafteratevent

Create `timeseries` at or after event

Syntax

```
tsout = gettsafteratevent(tsin,tsevent)
tsout = gettsafteratevent(tsin,tsevent,n)
```

Description

`tsout = gettsafteratevent(tsin,tsevent)` returns a `timeseries` object with samples occurring at and after the event `tsevent` in the input `timeseries` `tsin`. The argument `tsevent` can be either a `tsdata.event` object or a character vector containing the name of the event. When `tsevent` is a `tsdata.event` object, the time defined by `tsevent` specifies the time. When `tsevent` is a character vector, the first `tsdata.event` object in the `Events` property of `tsin` that matches the event name specifies the time.

`tsout = gettsafteratevent(tsin,tsevent,n)` returns a `timeseries` object with samples at and after an event in `tsin`, where `n` is the number of the event occurrence for the event `tsevent`.

Examples

timeseries from Event

Create a `timeseries` object with an event occurring at time sample 2.

```
tsin = timeseries((1:5)');
tsevent = tsdata.event('MyEvent',2);
tsin = addevent(tsin,tsevent);
tsin.Events
```

```
    EventData: []
         Name: 'MyEvent'
```

```
         Time: 2
        Units: 'seconds'
    StartDate: ''
```

Create a new `timeseries` object that contains the data from the event sample of `tsin` and all samples after the event.

```
tsout = gettsafteratevent(tsin,'MyEvent');
tsout.Time
```

```
ans = 3×1

     2
     3
     4
```

Input Arguments

`tsin` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

`tsevent` — Event
scalar | character vector

Event, specified as a scalar `tsdata.event` object or a character vector containing the name of an event.

`n` — Event number
scalar integer

Event number, specified as a scalar integer corresponding to the nth `tsdata.event` of a `timeseries` object.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Tips

- When the input `timeseries` object `tsin` contains date character vectors and `tsevent` uses numeric time, the time selected by `tsevent` is treated as a date that is calculated relative to the `StartDate` property in `tsin.TimeInfo`.
- When `tsin` uses numeric time and `tsevent` uses calendar dates, the time selected by `tsevent` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterevent` | `gettsbeforeevent` | `gettsbetweenevents` | `timeseries` | `tsdata.event`

Introduced before R2006a

gettsafterevent

Create `timeseries` after event

Syntax

```
tsout = gettsafterevent(tsin,tsevent)
tsout = gettsafterevent(tsin,tsevent,n)
```

Description

`tsout = gettsafterevent(tsin,tsevent)` returns a `timeseries` object with samples occurring after the event `tsevent` in the input `timeseries` `tsin`. The argument `tsevent` can be either a `tsdata.event` object or a character vector containing the name of the event. When `tsevent` is a `tsdata.event` object, the time defined by `tsevent` specifies the time. When `tsevent` is a character vector, the first `tsdata.event` object in the `Events` property of `tsin` that matches the event name specifies the time.

`tsout = gettsafterevent(tsin,tsevent,n)` returns a `timeseries` object with samples after an event in `tsin`, where `n` is the number of the event occurrence for the event `tsevent`.

Examples

`timeseries` after Event

Create a `timeseries` object with an event occurring at time sample 2.

```
tsin = timeseries((1:5)');
tsevent = tsdata.event('MyEvent',2);
tsin = addevent(tsin,tsevent);
tsin.Events
```

```
    EventData: []
         Name: 'MyEvent'
```

```
         Time: 2
        Units: 'seconds'
    StartDate: ''
```

Create a new `timeseries` object that contains the data after the event sample.

```
tsout = gettsafterevent(tsin,'MyEvent');
tsout.Time
```

```
ans = 2×1

     3
     4
```

Input Arguments

`tsin` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

`tsevent` — Event
scalar | character vector

Event, specified as a scalar `tsdata.event` object or a character vector containing the name of an event.

`n` — Event number
scalar integer

Event number, specified as a scalar integer corresponding to the nth `tsdata.event` of a `timeseries` object.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Tips

- When the input `timeseries` object `tsin` contains date character vectors and `tsevent` uses numeric time, the time selected by `tsevent` is treated as a date that is calculated relative to the `StartDate` property in `tsin.TimeInfo`.
- When `tsin` uses numeric time and `tsevent` uses calendar dates, the time selected by `tsevent` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafteratevent` | `gettsbeforeevent` | `gettsbetweenevents` | `timeseries` | `tsdata.event`

Introduced before R2006a

gettsatevent

Create `timeseries` at event

Syntax

```
tsout = gettsatevent(tsin,tsevent)
tsout = gettsatevent(tsin,tsevent,n)
```

Description

`tsout = gettsatevent(tsin,tsevent)` returns a `timeseries` object with samples occurring at the event `tsevent` in the input `timeseries` `tsin`. The argument `tsevent` can be either a `tsdata.event` object or a character vector containing the name of the event. When `tsevent` is a `tsdata.event` object, the time defined by `tsevent` specifies the time. When `tsevent` is a character vector, the first `tsdata.event` object in the `Events` property of `tsin` that matches the event name specifies the time.

`tsout = gettsatevent(tsin,tsevent,n)` returns a `timeseries` object with samples occurring at the event in `tsin`, where `n` is the number of the event occurrence for the event `tsevent`.

Examples

timeseries at Event

Create a `timeseries` object with an event occurring at time sample 2.

```
tsin = timeseries((1:5)');
tsevent = tsdata.event('MyEvent',2);
tsin = addevent(tsin,tsevent);
tsin.Events
```

```
    EventData: []
         Name: 'MyEvent'
```

```
         Time: 2
        Units: 'seconds'
    StartDate: ''
```

Create a new `timeseries` object that contains the data at the event sample.

```
tsout = gettsatevent(tsin,'MyEvent');
tsout.Time
```

```
ans = 2
```

Input Arguments

`tsin` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

`tsevent` — Event
scalar | character vector

Event, specified as a scalar `tsdata.event` object or a character vector containing the name of an event.

`n` — Event number
scalar integer

Event number, specified as a scalar integer corresponding to the nth `tsdata.event` of a `timeseries` object.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Tips

- When the input `timeseries` object `tsin` contains date character vectors and `tsevent` uses numeric time, the time selected by `tsevent` is treated as a date that is calculated relative to the `StartDate` property in `tsin.TimeInfo`.

- When `tsin` uses numeric time and `tsevent` uses calendar dates, the time selected by `tsevent` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafteratevent` | `gettsafterevent` | `gettsbeforeevent` | `gettsbetweenevents` | `timeseries` | `tsdata.event`

Introduced before R2006a

gettsbeforeatevent

Create `timeseries` at or before event

Syntax

```
tsout = gettsbeforeatevent(tsin,tsevent)
tsout = gettsbeforeatevent(tsin,tsevent,n)
```

Description

`tsout = gettsbeforeatevent(tsin,tsevent)` returns a `timeseries` object with samples occurring at and before the event `tsevent` in the input `timeseries` `tsin`. The argument `tsevent` can be either a `tsdata.event` object or a character vector containing the name of the event. When `tsevent` is a `tsdata.event` object, the time defined by `tsevent` specifies the time. When `tsevent` is a character vector, the first `tsdata.event` object in the `Events` property of `tsin` that matches the event name specifies the time.

`tsout = gettsbeforeatevent(tsin,tsevent,n)` returns a `timeseries` object with samples at and before an event in `tsin`, where `n` is the number of the event occurrence for the event `tsevent`.

Examples

`timeseries` from Event

Create a `timeseries` object with an event occurring at time sample 2.

```
tsin = timeseries((1:5)');
tsevent = tsdata.event('MyEvent',2);
tsin = addevent(tsin,tsevent);
tsin.Events
```

```
    EventData: []
         Name: 'MyEvent'
```

```
         Time: 2
        Units: 'seconds'
    StartDate: ''
```

Create a new `timeseries` object that contains the data from the event sample of `tsin` and all samples before the event.

```
tsout = gettsbeforeatevent(tsin,'MyEvent');
tsout.Time
```

```
ans = 3×1

     0
     1
     2
```

Input Arguments

`tsin` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

`tsevent` — Event
scalar | character vector

Event, specified as a scalar `tsdata.event` object or a character vector containing the name of an event.

`n` — Event number
scalar integer

Event number, specified as a scalar integer corresponding to the nth `tsdata.event` of a `timeseries` object.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Tips

- When the input `timeseries` object `tsin` contains date character vectors and `tsevent` uses numeric time, the time selected by `tsevent` is treated as a date that is calculated relative to the `StartDate` property in `tsin.TimeInfo`.
- When `tsin` uses numeric time and `tsevent` uses calendar dates, the time selected by `tsevent` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterevent` | `gettsbeforeevent` | `gettsbetweenevents` | `timeseries` | `tsdata.event`

Introduced before R2006a

gettsbeforeevent

Create `timeseries` before event

Syntax

```
tsout = gettsbeforeevent(tsin,tsevent)
tsout = gettsbeforeevent(tsin,tsevent,n)
```

Description

`tsout = gettsbeforeevent(tsin,tsevent)` returns a `timeseries` object with samples occurring before the event `tsevent` in the input `timeseries` `tsin`. The argument `tsevent` can be either a `tsdata.event` object or a character vector containing the name of the event. When `tsevent` is a `tsdata.event` object, the time defined by `tsevent` specifies the time. When `tsevent` is a character vector, the first `tsdata.event` object in the `Events` property of `tsin` that matches the event name specifies the time.

`tsout = gettsbeforeevent(tsin,tsevent,n)` returns a `timeseries` object with samples before an event in `tsin`, where `n` is the number of the event occurrence for the event `tsevent`.

Examples

`timeseries` before Event

Create a `timeseries` object with an event occurring at time sample 2.

```
tsin = timeseries((1:5)');
tsevent = tsdata.event('MyEvent',2);
tsin = addevent(tsin,tsevent);
tsin.Events
```

```
    EventData: []
         Name: 'MyEvent'
```

```
         Time: 2
        Units: 'seconds'
    StartDate: ''
```

Create a new `timeseries` object that contains the data before the event sample.

```
tsout = gettsbeforeevent(tsin,'MyEvent');
tsout.Time
```

```
ans = 2×1

     0
     1
```

Input Arguments

`tsin` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

`tsevent` — Event
scalar | character vector

Event, specified as a scalar `tsdata.event` object or a character vector containing the name of an event.

`n` — Event number
scalar integer

Event number, specified as a scalar integer corresponding to the nth `tsdata.event` of a `timeseries` object.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Tips

- When the input `timeseries` object `tsin` contains date character vectors and `tsevent` uses numeric time, the time selected by `tsevent` is treated as a date that is calculated relative to the `StartDate` property in `tsin.TimeInfo`.
- When `tsin` uses numeric time and `tsevent` uses calendar dates, the time selected by `tsevent` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterevent` | `gettsbeforeatevent` | `gettsbetweenevents` | `timeseries` | `tsdata.event`

Introduced before R2006a

gettsbetweenevents

Create `timeseries` between events

Syntax

```
tsout = gettsbetweenevents(tsin,tsevent1,tsevent2)
tsout = gettsbeforeevent(tsin,tsevent1,tsevent2,n1,n2)
```

Description

`tsout = gettsbetweenevents(tsin,tsevent1,tsevent2)` returns a `timeseries` object with samples occurring between the events `tsevent1` and `tsevent2` in the input `timeseries` `tsin`. The arguments `tsevent1` and `tsevent2` can be either `tsdata.event` objects or character vectors containing the names of the events. When `tsevent1` and `tsevent2` are `tsdata.event` objects, the times defined by the events specify the time. When `tsevent1` and `tsevent2` are character vectors, the first `tsdata.event` object in the `Events` property of `tsin` that matches the event names specifies the time.

`tsout = gettsbeforeevent(tsin,tsevent1,tsevent2,n1,n2)` returns a `timeseries` object with samples between two events in `tsin`, where `n1` is the number of the event occurrence for `tsevent1` and `n2` is the number of the event occurrence for `tsevent2`.

Examples

`timeseries` Between Events

Create a `timeseries` object with events occurring at time samples 1 and 3.

```
tsin = timeseries((1:5)');
tsevent1 = tsdata.event('StartEvent',1);
tsevent2 = tsdata.event('EndEvent',3);
tsin = addevent(tsin,[tsevent1 tsevent2]);
```

Create a new `timeseries` object that contains the data between the two event samples.

```
tsout = gettsbetweenevents(tsin,'StartEvent','EndEvent');
tsout.Time
```

```
ans = 3×1

     1
     2
     3
```

Input Arguments

`tsin` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

`tsevent1` — Start event
scalar | character vector

Start event, specified as a scalar `tsdata.event` object or a character vector containing the name of an event.

`tsevent2` — End event
scalar | character vector

End event, specified as a scalar `tsdata.event` object or a character vector containing the name of an event.

`n1` — Start event number
scalar integer

Start event number, specified as a scalar integer corresponding to the `n1` `tsdata.event` of a `timeseries` object.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

n2 — End event number
scalar integer

End event number, specified as a scalar integer corresponding to the `n2` `tsdata.event` of a `timeseries` object.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Tips

- When the input `timeseries` object `tsin` contains date character vectors and `tsevent1` and `tsevent2` use numeric time, the time selected is treated as a date that is calculated relative to the `StartDate` property in `tsin.TimeInfo`.
- When `tsin` uses numeric time and `tsevent1` and `tsevent2` use calendar dates, the time selected is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterevent` | `gettsbeforeevent` | `timeseries` | `tsdata.event`

Introduced before R2006a

GetVariable

Data from variable in Automation server workspace

Syntax

IDL Method Signature

```
HRESULT GetVariable([in] BSTR varname, [in] BSTR workspace,
    [out, retval] VARIANT* pdata)
```

Microsoft Visual Basic Client

```
GetVariable(varname As String, workspace As String) As Object
```

MATLAB Client

```
D = GetVariable(h,'varname','workspace')
```

Description

`D = GetVariable(h,'varname','workspace')` gets data stored in variable `varname` from the specified `workspace` of the server attached to handle `h` and returns it in output argument `D`. The values for *`workspace`* are `base` or `global`.

Do *not* use `GetVariable` on sparse arrays, structures, or function handles.

If your scripting language requires the explicit return of a result, then use the `GetVariable` function in place of `GetWorkspaceData`, `GetFullMatrix`, or `GetCharArray`.

Examples

Read Cell Array in VB .NET Client

This example shows how to read a cell array from a MATLAB Automation server.

```
type getvariable.vb
```

```
Dim Matlab As Object
Dim Result As String
Dim C2 As Object
Matlab = CreateObject("matlab.application")
Result = Matlab.Execute("C1 = {25.72, 'hello', rand(4)};")
C2 = Matlab.GetVariable("C1", "base")
MsgBox("Second item in cell array: " & C2(0, 1))
```

Read Cell Array in VBA

This example shows how to read a cell array from a MATLAB Automation server.

```
type getvariable.vba
```

```
Dim Matlab As Object
Dim Result As String
Dim C2 As Variant
Set Matlab = CreateObject("matlab.application")
Result = Matlab.Execute("C1 = {25.72, 'hello', rand(4)};")
C2 = Matlab.GetVariable("C1", "base")
MsgBox ("Second item in cell array: " & C2(0, 1))
```

See Also

`Execute` | `GetCharArray` | `GetFullMatrix` | `GetWorkspaceData`

Topics

"Conversion of COM Types to MATLAB Types"
"Conversion of MATLAB Types to COM Types"

Introduced before R2006a

Tiff.getVersion

LibTIFF library version

Syntax

```
versionString = Tiff.getVersion()
```

Description

`versionString = Tiff.getVersion()` returns a character vector containing the version number of the LibTIFF library.

Examples

Get Version

Display the version of LibTIFF library.

```
Tiff.getVersion
```

```
ans =
    'LIBTIFF, Version 4.0.10
     Copyright (c) 1988-1996 Sam Leffler
     Copyright (c) 1991-1996 Silicon Graphics, Inc.'
```

Algorithms

References

This function corresponds to the `TIFFGetVersion` function in the LibTIFF C API. To use this function, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff`

Introduced in R2009b

GetWorkspaceData

Data from Automation server workspace

Syntax

IDL Method Signature

```
HRESULT GetWorkspaceData([in] BSTR varname, [in] BSTR workspace,
    [out] VARIANT* pdata)
```

Microsoft Visual Basic Client

```
GetWorkspaceData(varname As String, workspace As String) As Object
```

MATLAB Client

```
D = GetWorkspaceData(h,'varname','workspace')
```

Description

`D = GetWorkspaceData(h,'varname','workspace')` gets data stored in variable `varname` from the specified `workspace` of the server attached to handle `h` and returns it in output argument `D`. The values for *`workspace`* are `base` or `global`. Use this syntax when writing VBA code.

Use `GetWorkspaceData` instead of `GetFullMatrix` and `GetCharArray` to get numeric and character array data, respectively. Do *not* use `GetWorkspaceData` on sparse arrays, structures, or function handles.

For VBScript clients, use the `GetWorkspaceData` and `PutWorkspaceData` functions to pass numeric data to and from the MATLAB workspace. These functions use the `variant` data type instead of the `safearray` data type used by `GetFullMatrix` and `PutFullMatrix`. VBScript does not support `safearray`.

Examples

Read data from MATLAB Automation Server

This example shows how to use a Visual Basic® .NET client to read data from a MATLAB Automation server.

```
type getworkspacedata.vb
```

```
Dim Matlab As Object
Dim C2 As Object
Dim Result As String
Matlab = CreateObject("matlab.application")
Result = MatLab.Execute("C1 = {25.72, 'hello', rand(4)};")
MsgBox("In MATLAB, type" & vbCrLf & "C1")
Matlab.GetWorkspaceData("C1", "base", C2)
MsgBox("second value of C1 = " & C2(0, 1))
```

Read data from MATLAB Automation Server With VBA

This example shows how to use a VBA client to read data from a MATLAB Automation server.

```
type getworkspacedata.vba
```

```
Dim Matlab As Object
Dim C2 As Variant
Dim Result As String
Set Matlab = CreateObject("matlab.application")
Result = MatLab.Execute("C1 = {25.72, 'hello', rand(4)};")
MsgBox("In MATLAB, type" & vbCrLf & "C1")
X = Matlab.GetWorkspaceData("C1", "base", C2)
MsgBox("second value of C1 = " & C2(0, 1))
```

See Also

`Execute` | `GetCharArray` | `GetFullMatrix` | `GetVariable` | `PutWorkspaceData`

Topics

"Conversion of COM Types to MATLAB Types"
"Conversion of MATLAB Types to COM Types"

Introduced before R2006a

matlab.general Settings

General settings including matfile settings

You can customize MATLAB using the `matlab.general` settings. Access `matlab.general` settings using the root `SettingsGroup` object returned by the `settings` function. For example, set the temporary value for the MAT-file save format to `'v7.3'` (where the default is `'v7'`).

```
s = settings;
s.matlab.general.matfile.SaveFormat.TemporaryValue = 'v7.3'
```

For more information about settings, see “Access and Modify Settings”.

Settings

matlab.general.matfile

`SaveFormat` — MAT-File version

`'v7'` (default) | `'v7.3'` | `'v6'`

MAT-File version to save MAT-files to, specified as `'v7.3'`, `'v7'`, or `'v6'`.

For more information about MAT-file versions, see “MAT-File Versions”.

Example: `s.matlab.general.matfile.SaveFormat.TemporaryValue = 'v7.3'`

See Also

`settings`

Topics

“Access and Modify Settings”
“MAT-File Versions”

Introduced in R2018a

ginput

Identify axes coordinates

Syntax

```
[x,y] = ginput(n)
[x,y] = ginput
[x,y,button] = ginput( ___ )
```

Description

`[x,y] = ginput(n)` allows you to identify the coordinates of `n` points. To choose a point, move your cursor to the desired location and press either a mouse button or a key on the keyboard. Press the **Return** key to stop before all `n` points are selected. MATLAB returns the coordinates of your selected points. If there are no current axes, calling `ginput` creates a set of Cartesian axes.

`[x,y] = ginput` allows you to select an unlimited number of points until you press the **Return** key.

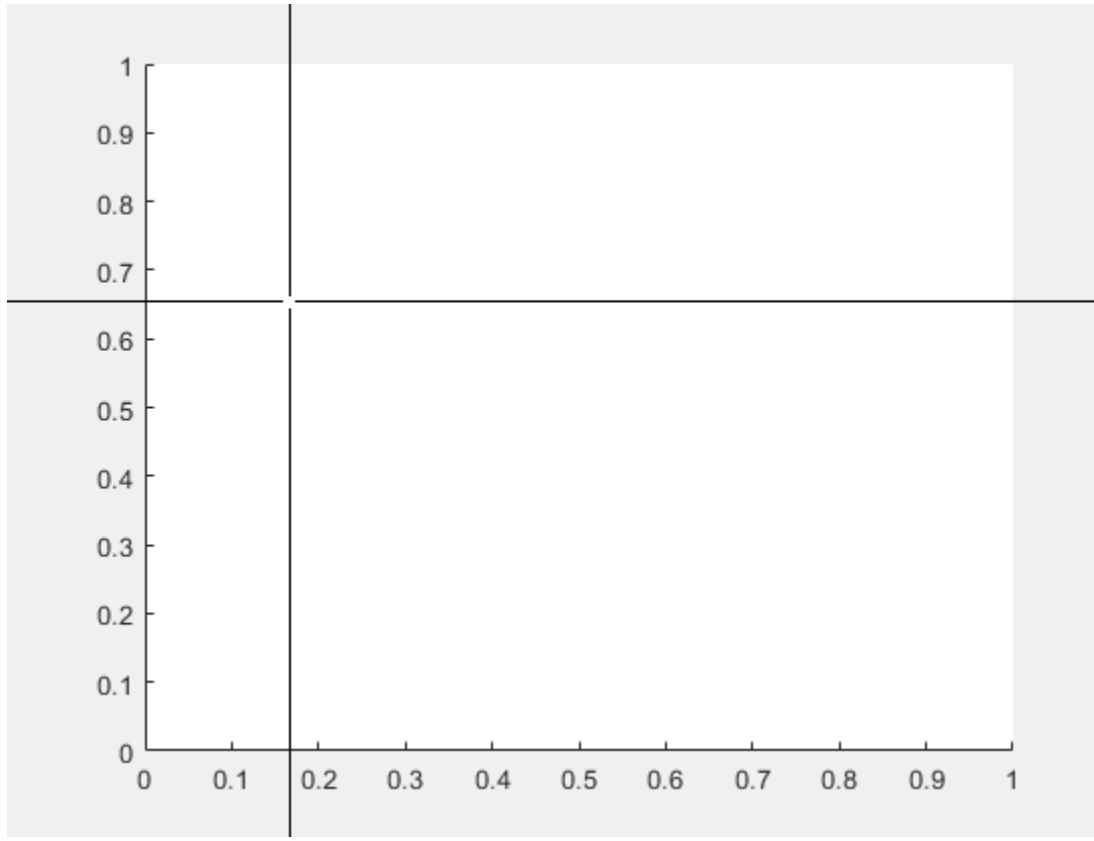
`[x,y,button] = ginput( ___ )` also returns the mouse button or key on the keyboard used to select each point.

Examples

Identify Points and Plot Coordinates

Identify four points in a set of axes using `ginput`. To select each point, move the cursor to your desired location and press a mouse button or key.

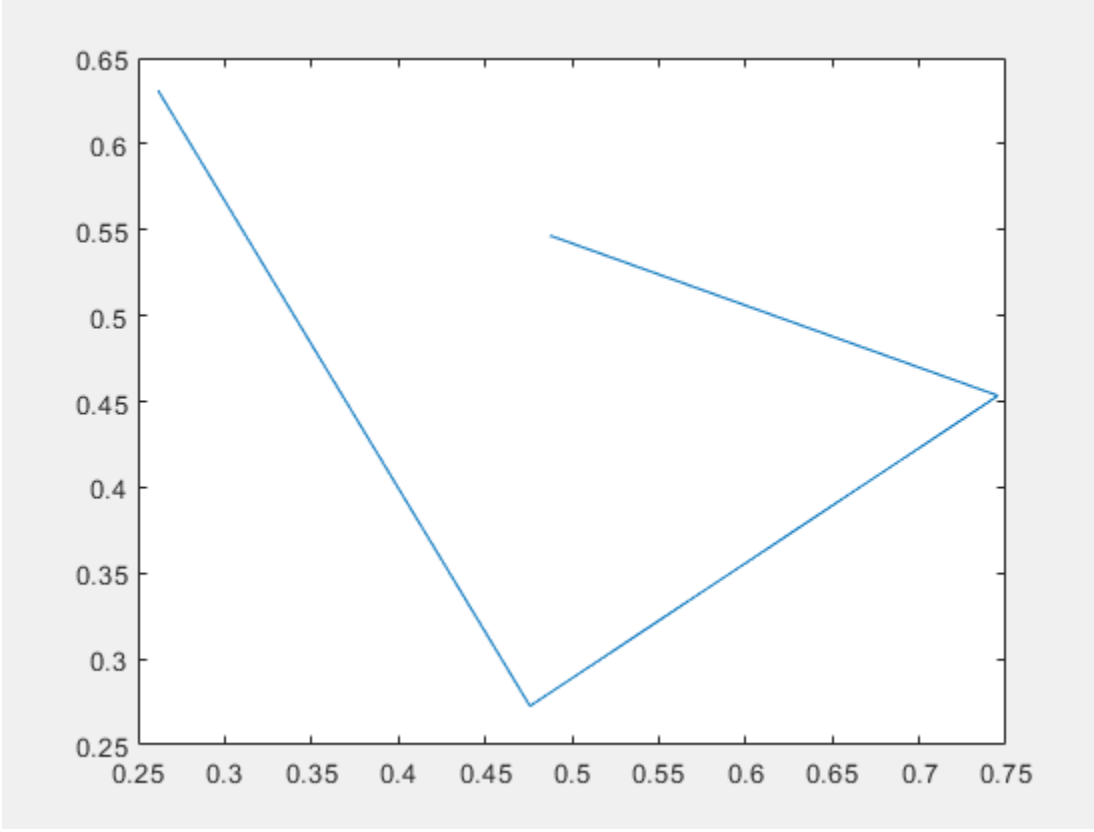
```
[x,y] = ginput(4)
```




```
x =

    0.3699
    0.4425
    0.5450
    0.6130


y =

    0.6690
    0.5605
    0.4719
    0.6025
```

Plot the points.

```
plot(x,y);
```



Return Buttons Used to Select Coordinates

Identify five coordinates in a set of axes using `ginput`. To select each point, move the cursor to your desired location and press a mouse button, lowercase letter, uppercase letter, number, or the space bar. Return the mouse buttons or ASCII numbers of the keys used to select each point.

```
[x,y,button] = ginput(5);
button
```

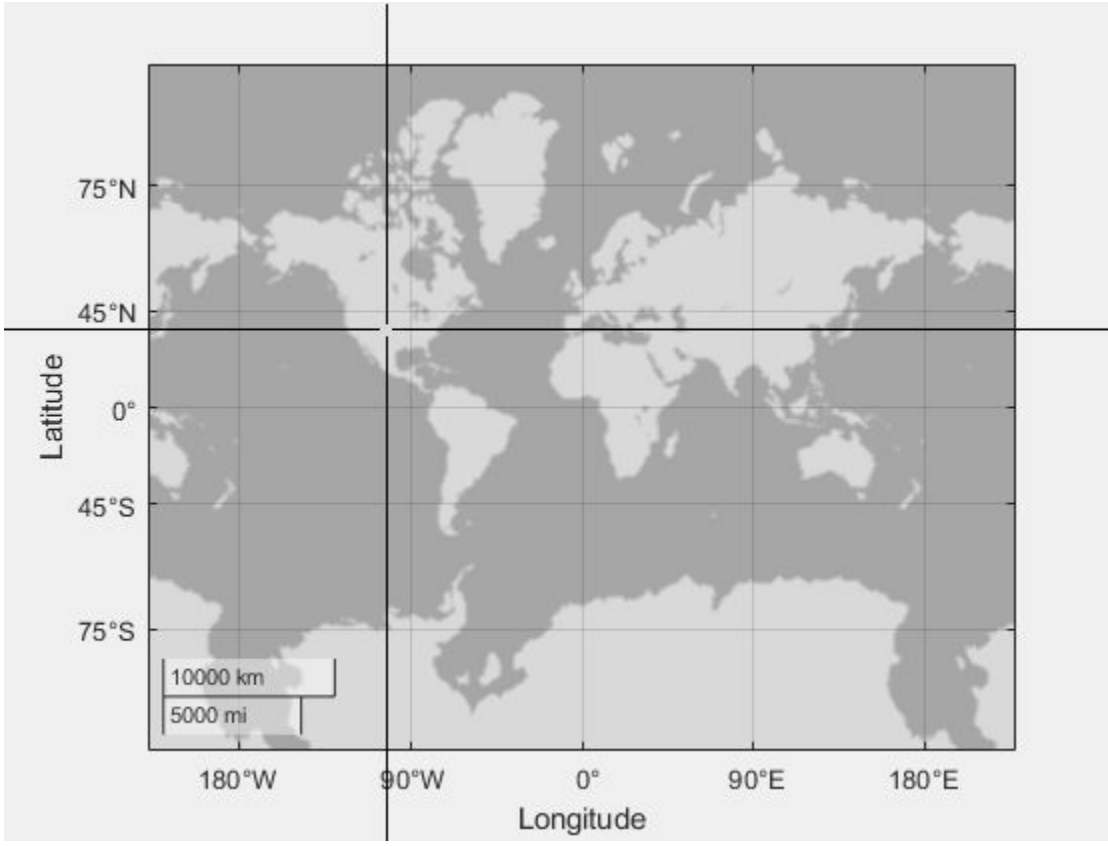
```
button =

     3
   104
    32
    51
    82
```

In this case, the coordinates were identified using the right mouse button (3), the lowercase letter **h** (104), the space bar (32), the number **3** (51), and the uppercase letter **R** (82).

Identify Points on Geographic Axes

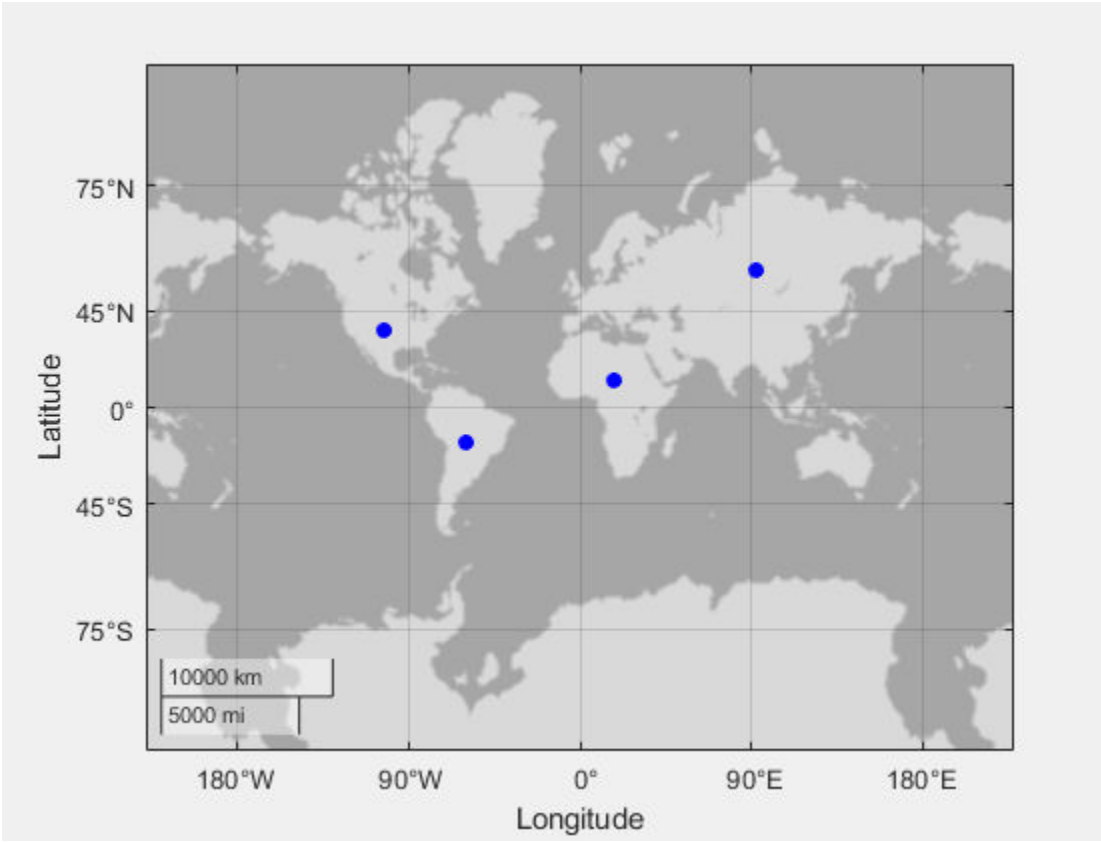
Create geographic axes and identify the latitude and longitude coordinates of four points. Then, plot the points that you identify.

```
geoaxes;
[lat,lon] = ginput(4);
```



Use the `hold on` command and the `geolimits` function to maintain the map limits. Plot the identified points.

```
hold on
geolimits('manual')
geoscatter(lat,lon,'filled','b')
```



Input Arguments

n — Number of points
positive integer

Number of points to identify, specified as a positive integer.

Output Arguments

x — First components of identified coordinates
scalar | column vector

First components of the identified coordinates, returned as a scalar or column vector.

- If the current axes are Cartesian, the values are x-coordinates.
- If the current axes are geographic, the values are latitudes. Positive values correspond to north and negative values correspond to south.

If you choose points outside the axes limits, values are still returned relative to the axes origin.

y — Second components of identified coordinates
scalar | column vector

Second components of the identified coordinates, returned as a scalar or column vector.

- If the current axes are Cartesian, the values are y-coordinates.
- If the current axes are geographic, the values are longitudes. Positive values correspond to locations east of the origin and negative values correspond to locations west of the origin.

If you choose points outside the axes limits, values are still returned relative to the axes origin.

`button` — Keys or mouse buttons used to identify points
scalar | column vector

Keys or mouse buttons used to identify points, returned as a scalar or column vector. Mouse buttons are indicated by 1 for the left button, 2 for the middle, and 3 for the right. Keys on the keyboard are indicated by their corresponding ASCII numbers. Taps on a screen are indicated by 1.

See Also

Functions
`gtext`

Introduced before R2006a

global

Declare variables as global

Syntax

```
global var1 ... varN
```

Description

`global var1 ... varN` declares variables `var1 ... varN` as global in scope.

Ordinarily, each MATLAB function has its own local variables, which are separate from those of other functions and from those of the base workspace. However, if several functions all declare a particular variable name as `global`, then they all share a single copy of that variable. Any change of value to that variable, in any function, is visible to all the functions that declare it as global.

If the global variable does not exist the first time you issue the `global` statement, it is initialized to an empty `0x0` matrix.

If a variable with the same name as the global variable already exists in the current workspace, MATLAB issues a warning and changes the value of that variable and its scope to match the global variable.

Examples

Share Global Variable Between Functions

Create a function in your current working folder that sets the value of a global variable.

```
function setGlobalx(val)
global x
x = val;
```

Create a function in your current working folder that returns the value of a global variable. These two functions have separate function workspaces, but they both can access the global variable.

```
function r = getGlobalx
global x
r = x;
```

Set the value of the global variable, x, and obtain it from a different workspace.

```
setGlobalx(1138)
r = getGlobalx
```

```
r =

        1138
```

Share Global Variable Between Function and Command Line

Assign a value to the global variable using the function that you defined in the previous example.

```
clear all
setGlobalx(42)
```

Display the value of the global variable, x. Even though the variable is global, it is not accessible at the command line.

```
x
```

```
Undefined function or variable 'x'.
```

Declare x as a global variable at the command line, and display its value.

```
global x
x
```

```
x =

    42
```

Change the value of x and use the function that you defined in the previous example to return the global value from a different workspace.

```
x = 1701;
r = getGlobalx
```

```
r =

        1701
```

Tips

- To clear a global variable from all workspaces, use `clear global` *`variable`*.
- To clear a global variable from the current workspace but not other workspaces, use `clear` *`variable`*.

See Also

`clear` | `persistent` | `who`

Topics

"Share Data Between Workspaces"

Introduced before R2006a

gmres

Generalized minimum residual method (with restarts)

Syntax

```
x = gmres(A,b)
gmres(A,b,restart)
gmres(A,b,restart,tol)
gmres(A,b,restart,tol,maxit)
gmres(A,b,restart,tol,maxit,M)
gmres(A,b,restart,tol,maxit,M1,M2)
gmres(A,b,restart,tol,maxit,M1,M2,x0)
[x,flag] = gmres(A,b,...)
[x,flag,relres] = gmres(A,b,...)
[x,flag,relres,iter] = gmres(A,b,...)
[x,flag,relres,iter,resvec] = gmres(A,b,...)
```

Description

`x = gmres(A,b)` attempts to solve the system of linear equations `A*x = b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and should be large and sparse. The column vector `b` must have length `n`. `A` can be a function handle, `afun`, such that `afun(x)` returns `A*x`. For this syntax, `gmres` does not restart; the maximum number of iterations is `min(n,10)`.

“Parameterizing Functions” explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `gmres` converges, a message to that effect is displayed. If `gmres` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`gmres(A,b,restart)` restarts the method every `restart` inner iterations. The maximum number of outer iterations is `min(n/restart,10)`. The maximum number of

total iterations is `restart*min(n/restart,10)`. If `restart` is `n` or `[]`, then `gmres` does not restart and the maximum number of total iterations is `min(n,10)`.

`gmres(A,b,restart,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `gmres` uses the default, `1e-6`.

`gmres(A,b,restart,tol,maxit)` specifies the maximum number of outer iterations, i.e., the total number of iterations does not exceed `restart*maxit`. If `maxit` is `[]` then `gmres` uses the default, `min(n/restart,10)`. If `restart` is `n` or `[]`, then the maximum number of total iterations is `maxit` (instead of `restart*maxit`).

`gmres(A,b,restart,tol,maxit,M)` and `gmres(A,b,restart,tol,maxit,M1,M2)` use preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `gmres` applies no preconditioner. `M` can be a function handle `mfun` such that `mfun(x)` returns `M\x`.

`gmres(A,b,restart,tol,maxit,M1,M2,x0)` specifies the first initial guess. If `x0` is `[]`, then `gmres` uses the default, an all-zero vector.

`[x,flag] = gmres(A,b,...)` also returns a convergence flag:

`flag = 0`	`gmres` converged to the desired tolerance `tol` within `maxit` outer iterations.
`flag = 1`	`gmres` iterated `maxit` times but did not converge.
`flag = 2`	Preconditioner `M` was ill-conditioned.
`flag = 3`	`gmres` stagnated. (Two consecutive iterates were the same.)

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = gmres(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`. The third output, `relres`, is the relative residual of the preconditioned system.

`[x,flag,relres,iter] = gmres(A,b,...)` also returns both the outer and inner iteration numbers at which `x` was computed, where `0 <= iter(1) <= maxit` and `0 <= iter(2) <= restart`.

`[x,flag,relres,iter,resvec] = gmres(A,b,...)` also returns a vector of the residual norms at each inner iteration. These are the residual norms for the preconditioned system.

Examples

Using gmres with a Matrix Input

```
A = gallery('wilk',21);
b = sum(A,2);
tol = 1e-12;
maxit = 15;
M1 = diag([10:-1:1 1 1:10]);

x = gmres(A,b,10,tol,maxit,M1);
```

displays the following message:

```
gmres(10) converged at outer iteration 2 (inner iteration 9) to
a solution with relative residual 3.3e-013
```

Using gmres with a Function Handle

This example replaces the matrix `A` in the previous example with a handle to a matrix-vector product function `afun`, and the preconditioner `M1` with a handle to a backsolve function `mfun`. The example is contained in a function `run_gmres` that

- Calls `gmres` with the function handle `@afun` as its first argument.
- Contains `afun` and `mfun` as nested functions, so that all variables in `run_gmres` are available to `afun` and `mfun`.

The following shows the code for `run_gmres`:

```
function x1 = run_gmres
n = 21;
b = afun(ones(n,1));
tol = 1e-12;  maxit = 15;
x1 = gmres(@afun,b,10,tol,maxit,@mfun);

    function y = afun(x)
```

```
        y = [0; x(1:n-1)] + ...
              [((n-1)/2:-1:0)'; (1:(n-1)/2)'].*x + ...
              [x(2:n); 0];
    end

    function y = mfun(r)
        y = r ./ [((n-1)/2:-1:1)'; 1; (1:(n-1)/2)'];
    end
end
```

When you enter

```
x1 = run_gmres;
```

MATLAB software displays the message

```
gmres(10) converged at outer iteration 2 (inner iteration 10)
to a solution with relative residual 1.1e-013.
```

Using a Preconditioner without Restart

This example demonstrates the use of a preconditioner without restarting `gmres`.

Load `west0479`, a real 479-by-479 nonsymmetric sparse matrix.

```
load west0479;
A = west0479;
```

Set the tolerance and maximum number of iterations.

```
tol = 1e-12;
maxit = 20;
```

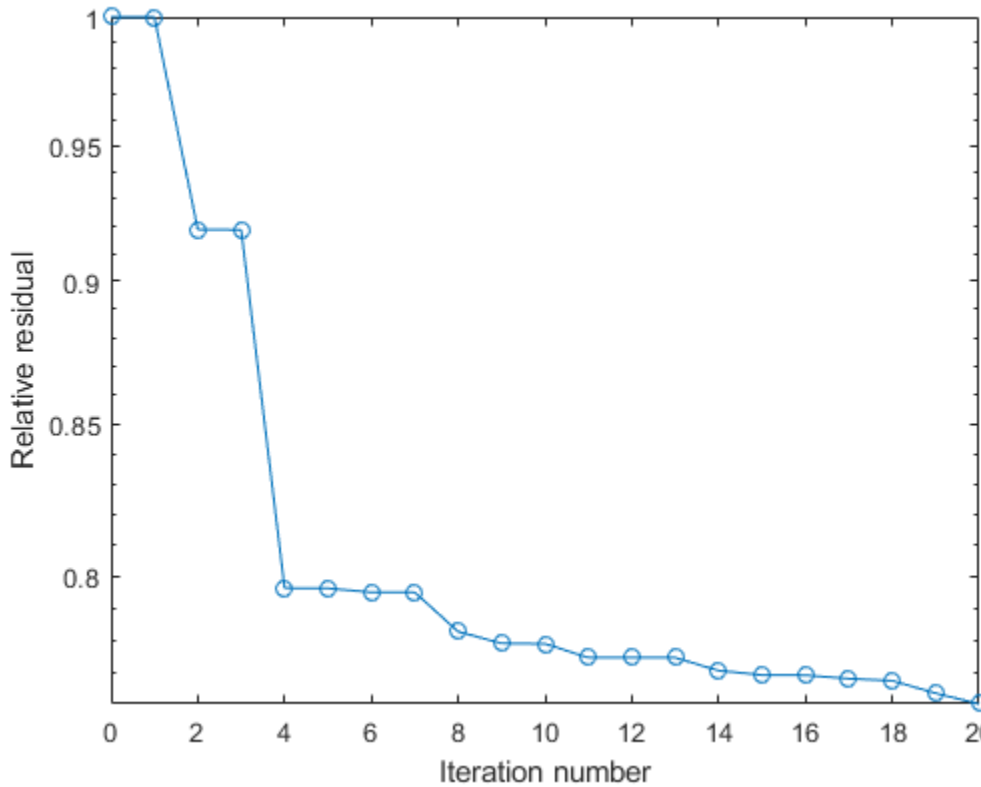
Define `b` so that the true solution is a vector of all ones.

```
b = full(sum(A,2));
[x0,fl0,rr0,it0,rv0] = gmres(A,b,[],tol,maxit);
```

`fl0` is 1 because `gmres` does not converge to the requested tolerance `1e-12` within the requested 20 iterations. The best approximate solution that `gmres` returns is the last one (as indicated by `it0(2) = 20`). MATLAB stores the residual history in `rv0`.

Plot the behavior of `gmres`.

```
semilogy(0:maxit,rv0/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



The plot shows that the solution converges slowly. A preconditioner may improve the outcome.

Use `ilu` to form the preconditioner, since `A` is nonsymmetric.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-5));
```

```
Error using ilu
There is a pivot equal to zero. Consider decreasing
the drop tolerance or consider using the 'udiag' option.
```

Note MATLAB cannot construct the incomplete LU as it would result in a singular factor, which is useless as a preconditioner.

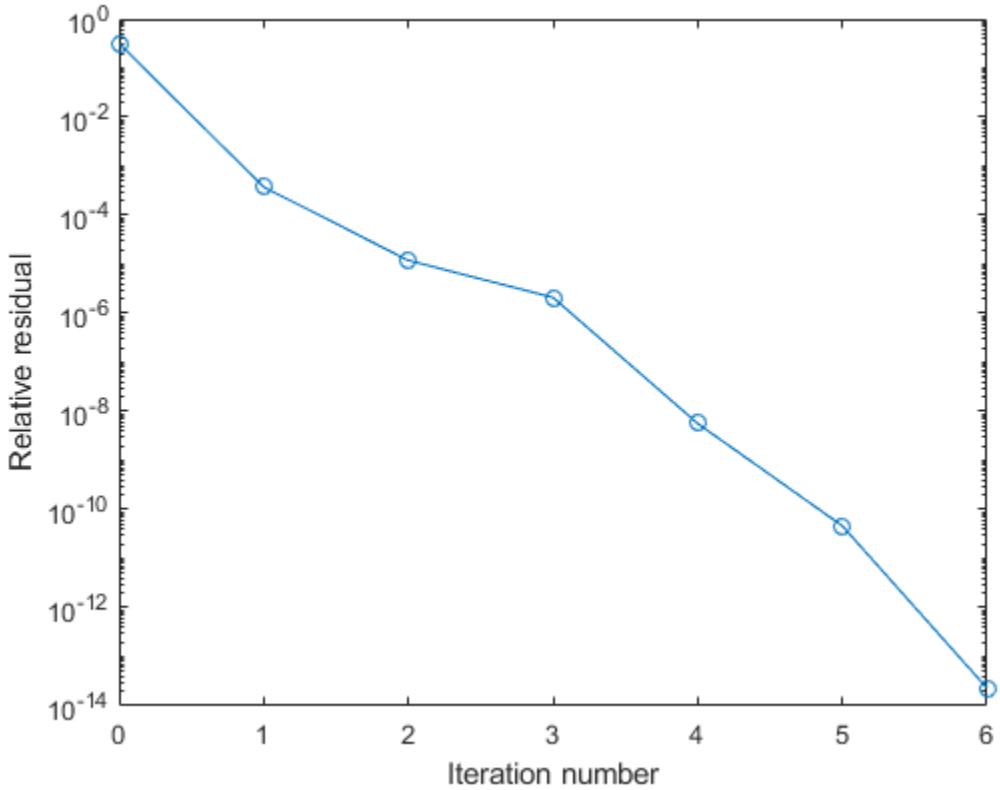
As indicated by the error message, try again with a reduced drop tolerance.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
[x1,fl1,rr1,it1,rv1] = gmres(A,b,[],tol,maxit,L,U);
```

`fl1` is 0 because `gmres` drives the relative residual to `9.5436e-14` (the value of `rr1`). The relative residual is less than the prescribed tolerance of `1e-12` at the sixth iteration (the value of `it1(2)`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. The output, `rv1(1)`, is `norm(M\b)`, where `M = L*U`. The output, `rv1(7)`, is `norm(U\(L\(b-A*x1)))`.

Follow the progress of `gmres` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0).

```
semilogy(0:it1(2),rv1/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



Using a Preconditioner with Restart

This example demonstrates the use of a preconditioner with restarted `gmres`.

Load `west0479`, a real 479-by-479 nonsymmetric sparse matrix.

```
load west0479;
A = west0479;
```

Define `b` so that the true solution is a vector of all ones.

```
b = full(sum(A,2));
```

Construct an incomplete LU preconditioner as in the previous example.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
```

The benefit to using restarted `gmres` is to limit the amount of memory required to execute the method. Without restart, `gmres` requires `maxit` vectors of storage to keep the basis of the Krylov subspace. Also, `gmres` must orthogonalize against all of the previous vectors at each step. Restarting limits the amount of workspace used and the amount of work done per outer iteration. Note that even though preconditioned `gmres` converged in six iterations above, the algorithm allowed for as many as twenty basis vectors and therefore, allocated all of that space up front.

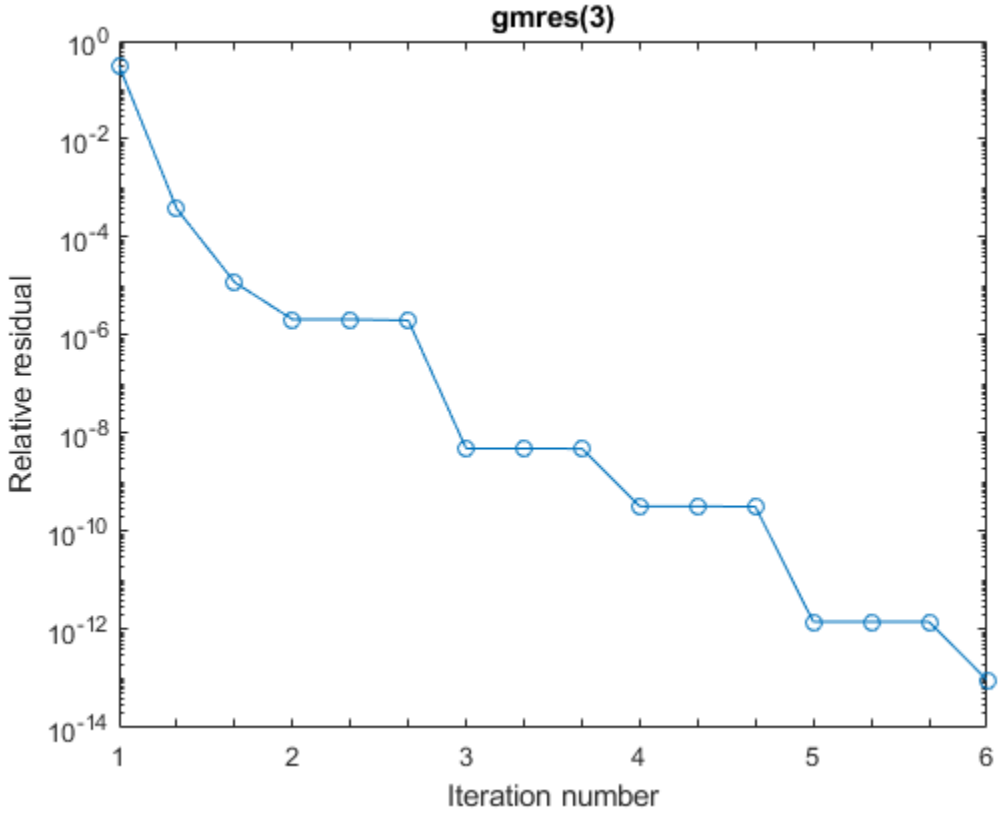
Execute `gmres(3)`, `gmres(4)`, and `gmres(5)`

```
tol = 1e-12;
maxit = 20;
re3 = 3;
[x3,fl3,rr3,it3,rv3] = gmres(A,b,re3,tol,maxit,L,U);
re4 = 4;
[x4,fl4,rr4,it4,rv4] = gmres(A,b,re4,tol,maxit,L,U);
re5 = 5;
[x5,fl5,rr5,it5,rv5] = gmres(A,b,re5,tol,maxit,L,U);
```

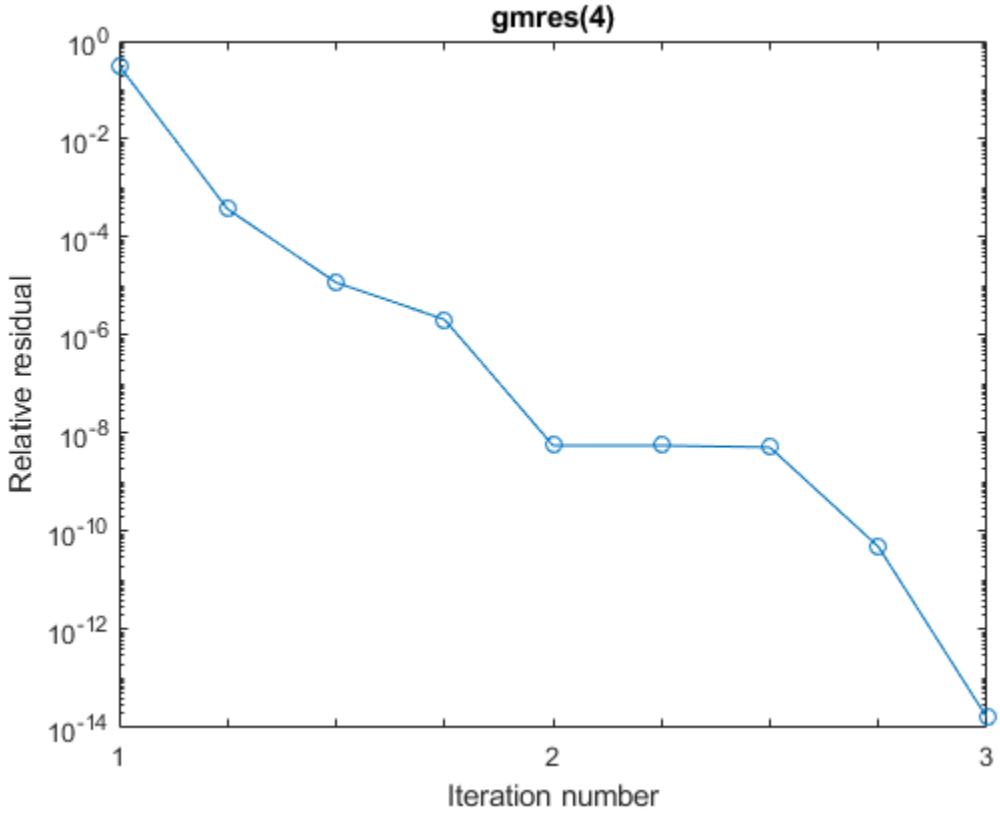
`fl3`, `fl4`, and `fl5` are all 0 because in each case restarted `gmres` drives the relative residual to less than the prescribed tolerance of `1e-12`.

The following plots show the convergence histories of each restarted `gmres` method. `gmres(3)` converges at outer iteration 5, inner iteration 3 (`it3 = [5, 3]`) which would be the same as outer iteration 6, inner iteration 0, hence the marking of 6 on the final tick mark.

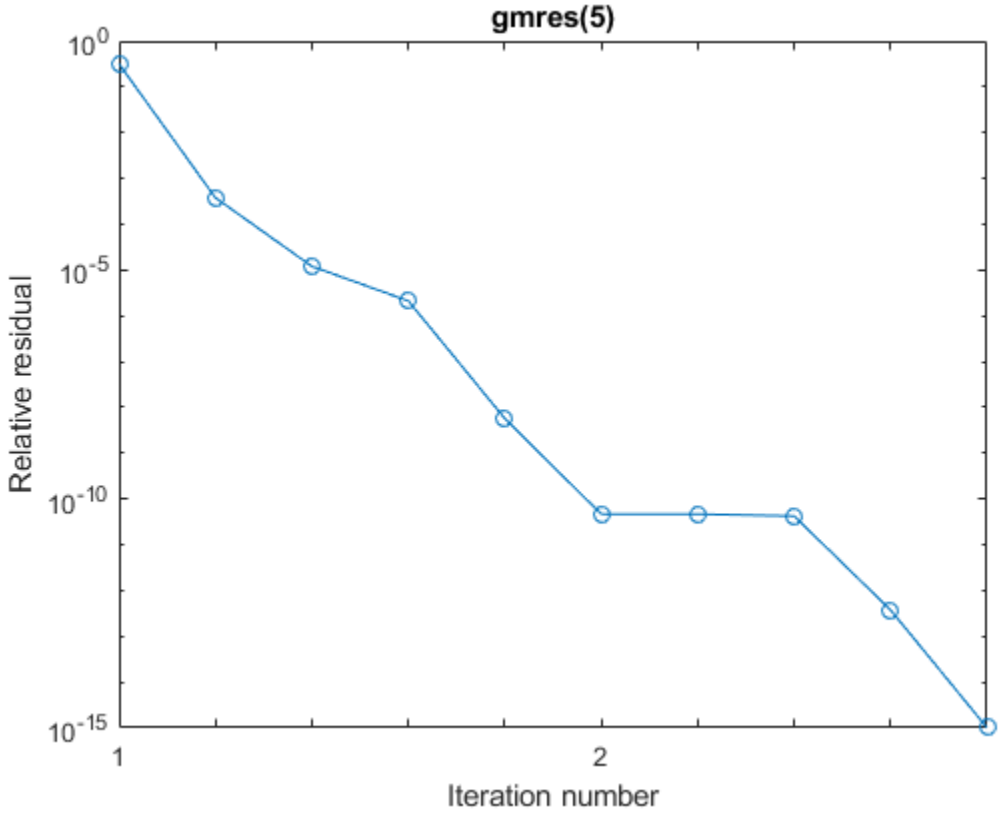
```
figure
semilogy(1:1/3:6,rv3/norm(b),'-o');
h1 = gca;
h1.XTick = [1:1/3:6];
h1.XTickLabel = ['1';' ';' ';'2';' ';' ';'3';' ';' ';'4';' ';' ';'5';' ';' ';'6';];
title('gmres(3)')
xlabel('Iteration number');
ylabel('Relative residual');
```




```
figure
semilogy(1:1/4:3,rv4/norm(b),'-o');
h2 = gca;
h2.XTick = [1:1/4:3];
h2.XTickLabel = ['1';' ';' ';' ';'2';' ';' ';' ';'3'];
title('gmres(4)')
xlabel('Iteration number');
ylabel('Relative residual');
```




```
figure
semilogy(1:1/5:2.8,rv5/norm(b),'-o');
h3 = gca;
h3.XTick = [1:1/5:2.8];
h3.XTickLabel = ['1';' ';' ';' ';' ';'2';' ';' ';' ';' '];
title('gmres(5)')
xlabel('Iteration number');
ylabel('Relative residual');
```



References

Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

Saad, Youcef and Martin H. Schultz, “GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems,” *SIAM J. Sci. Stat. Comput.*, July 1986, Vol. 7, No. 3, pp. 856-869.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When input `A` is a sparse matrix:
  - Only one sparse matrix preconditioner `M` is supported.
  - For a sparse matrix preconditioner, `resvec` returns the residual norms `norm(B-A*X)` without applying the preconditioner.
  - If you use two preconditioners, `M1` and `M2`, then both preconditioners must be functions.
  - For GPU arrays, `gmres` does not detect stagnation (Flag 3). Instead, it reports failure to converge (Flag 1).

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- If `M1` is a function, then it is applied independently to each row.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bicg` | `bicgstab` | `cgs` | `equilibrate` | `ilu` | `lsqr` | `minres` | `mldivide` | `pcg` | `qmr` | `symmlq`

Topics

"Create Function Handle"

Introduced before R2006a

gobjects

Initialize array for graphics objects

Syntax

```
H = gobjects(n)
H = gobjects(s1,...,sn)
H = gobjects(v)
```

```
H = gobjects
H = gobjects(0)
```

Description

`H = gobjects(n)` returns an `n`-by-`n` graphics object array. Use the `gobjects` function instead of the `ones` or `zeros` functions to preallocate an array to store graphics objects.

`H = gobjects(s1,...,sn)` returns an `s1`-by-...-by-`sn` graphics object array, where the list of integers `s1,...,sn` defines the dimensions of the array. For example, `gobjects(2,3)` returns a 2-by-3 array.

`H = gobjects(v)` returns a graphics object array where the elements of the row vector, `v`, define the dimensions of the array. For example, `gobjects([2,3,4])` returns a 2-by-3-by-4 array.

`H = gobjects` returns a 1-by-1 graphics object array.

`H = gobjects(0)` returns an empty graphics object array.

Examples

Specify Array Dimensions

Preallocate a 4-by-1 array to store graphics handles.

```
H = gobjects(4,1)
```

```
H 
  4x1 GraphicsPlaceholder array:

  GraphicsPlaceholder
  GraphicsPlaceholder
  GraphicsPlaceholder
  GraphicsPlaceholder
```

Specify Array Dimensions with Size of Existing Array

Create an array to store graphics handles using the size of an existing array.

Define A as a 3-by-4 array.

```
A = [1,2,3,2; 4,5,6,6; 7,8,9,7];
```

Create an array of graphics handles using the size of A.

```
v = size(A);
H = gobjects(v);
```

The dimensions of the graphics handle array are the same as the dimensions of A.

```
isequal(size(H),size(A))
```

```
ans = logical
   1
```

Return Empty Handle Array

Use the `gobjects` function to return an empty array.

```
H = gobjects(0)
```

```
H 
  0x0 empty GraphicsPlaceholder array.
```

Input Arguments

`n` — defines `n`-by-`n` array
integer value

Size of the object array, specified as an integer value. Negative integers are treated as 0. The array has dimensions `n`-by-`n`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`s1,...,sn` — Size of each array dimension
two or more integer values

Size of each array dimension, specified as a list of two or more integer values. Negative integers are treated as 0.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`v` — Size of each array dimension
row vector of integer values

Size of each array dimension, specified as a row vector of integer values. Negative integers are treated as 0.

Example: [2,4,6,7]

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`0` — Define an empty array
one or more dimensions equal to `0`

Define an empty array by specifying one or more dimension equal to `0`

Output Arguments

`H` — Initialized graphics object array
graphics object array of specified size

Initialized graphics object array of type `GraphicsPlaceholder`. Use this array to contain any type of graphics object.

See Also

Topics

"Graphics Arrays"

Introduced in R2013a

gplot

Plot nodes and edges in adjacency matrix

Syntax

```
gplot(A,XYCoords)
gplot(A,XYCoords,LineSpec)
[x,y] = gplot(A,XYCoords)
```

Description

`gplot(A,XYCoords)` plots a graph of the nodes and edges defined in the adjacency matrix `A` at the coordinates specified in `XYCoords`. The adjacency matrix `A` is an `n`-by-`n` matrix, where `n` is the number of nodes. `XYCoords` is an `n`-by-`2` matrix specifying *xy*-coordinates for each node.

`gplot(A,XYCoords,LineSpec)` additionally uses `LineSpec` to specify the line type, marker symbol, and color to use in the plot. For example, `gplot(A,XY,'r-*')` uses red lines for edges and red asterisks for nodes.

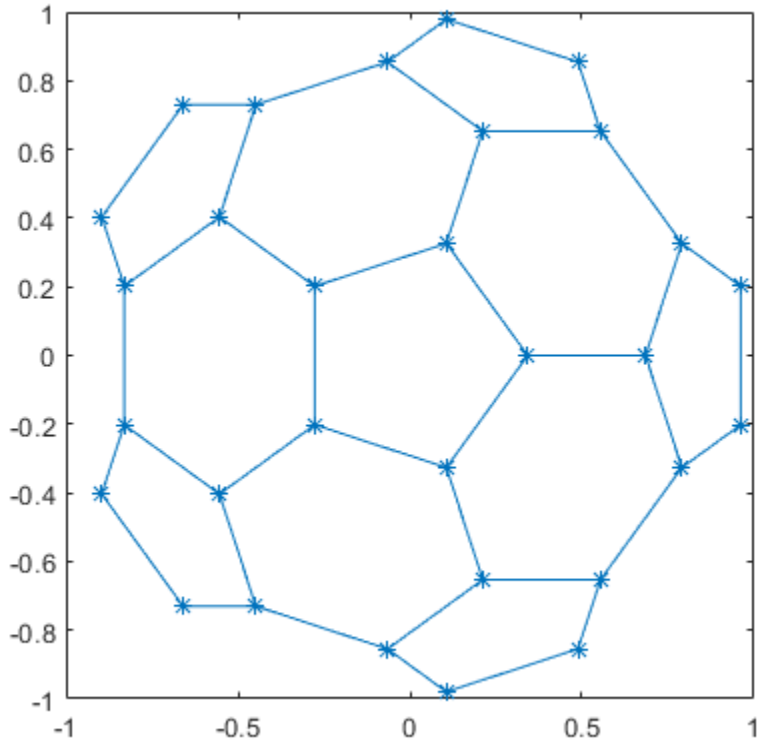
`[x,y] = gplot(A,XYCoords)` returns the `NaN`-delimited vectors `x` and `y` without generating a plot. Use `x` and `y` to generate a plot at a later time using `plot(x,y)`.

Examples

Plot Graph of Nodes Using Asterisks

Plot half of the carbon-60 molecule, placing asterisks at each node.

```
k = 1:30;
[B,XY] = bucky;
gplot(B(k,k),XY(k,[1 2]),'-*')
axis square
```



Input Arguments

A — Adjacency matrix
matrix

Adjacency matrix, specified as a matrix. `A` describes the connections between the nodes in the graph by the location of nonzero values. If node `i` and node `j` are connected, then `A(i,j)` or `A(j,i)` is nonzero; otherwise, `A(i,j)` and `A(j,i)` are zero.

Example: `A = ones(5)` is the adjacency matrix of a graph with five nodes where each node is connected to all the others.

Example: `A = [0 1 1 1; 1 0 0 0; 1 0 0 0; 1 0 0 0]` is the adjacency matrix of a graph with four nodes where one node connects to the other three.

Data Types: `single` | `double`

`XYCoords` — *xy*-coordinates of nodes
N-by-2 matrix

xy-coordinates of nodes, specified as an `N`-by-`2` matrix. Each row in `XYCoords` defines the coordinates for one node in the graph, so `XYCoords(i,:) = [x(i) y(i)]` gives the coordinates for node `i`.

Example: `XYCoords = [1 2; 3 4]` plots one node at (1,2) and a second node at (3,4).

`LineSpec` — Line style, marker, and color
character vector | string

Line style, marker, and color, specified as a character vector or string containing symbols. The symbols can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the nodes and no edges between them.

Example: `'--or'` is a red dashed line with circle markers.

Line Style	Description
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Marker	Description
`o`	Circle
`+`	Plus sign
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square

Marker	Description
d	Diamond
^	Upward-pointing triangle
v	Downward-pointing triangle
>	Right-pointing triangle
<	Left-pointing triangle
p	Pentagram
h	Hexagram

Color	Description
y	Yellow
m	Magenta
c	Cyan
r	Red
g	Green
b	Blue
w	White
k	Black

Output Arguments

x, y — Node coordinates
vectors

Node coordinates, returned as vectors. `x` and `y` contain the same information as `XYCoords`, but in a different format that is suitable for plotting with the command `plot(x,y)`. The line segments defined in `x` and `y` are separated with `NaN` values.

Tips

- Use `graph` and `digraph` objects to work with graph and network algorithms. You can visualize the networks with `plot`.

See Also

`LineSpec` | `digraph` | `graph` | `sparse` | `spy`

Topics

"Graphs and Matrices"
"Graph Plotting and Customization"

Introduced before R2006a

grabcode

Extract MATLAB code from file published to HTML

Syntax

```
grabcode filename
grabcode url
code = grabcode(filename)
```

Description

`grabcode filename` copies the MATLAB code from the HTML file `filename` and pastes it into an untitled document in the Editor. Use `grabcode` to get MATLAB code from published files when the source code is not readily available. `filename` is the HTML file created by publishing a MATLAB code file (`*.m`) or exporting a live script (`*.mlx`) to HTML. In both cases, the MATLAB code from the file appears as HTML comments at the end of `filename`.

`grabcode url` copies MATLAB code from the page or file specified by `url` and pastes it into an untitled document in the Editor.

`code = grabcode(filename)` returns the MATLAB code from `filename` to a character array.

Examples

Get MATLAB Code from HTML File

Use `grabcode` to get MATLAB® code from an existing HTML file.

Extract the MATLAB code from `my_sine_wave.html` into the variable `code`. To view the contents of the file, double-click `my_sine_wave.html` in your current folder.

```
code = grabcode('my_sine_wave.html')
```

```
code =
    '%% Plot Sine Wave
     % Calculate and plot a sine wave.

     %% Calculate and Plot Sine Wave
     % Calculate and plot |y = sin(x)|.

     function sine_wave_f(x)

     y = sin(x);
     plot(x,y)

     %% Modify Plot Properties

     title('Sine Wave', 'FontWeight','bold')
     xlabel('x')
     ylabel('sin(x)')
     set(gca, 'Color', 'w')
     set(gcf, 'MenuBar', 'none') '
```

Input Arguments

`filename` — Name of HTML file

character vector | string scalar

Name of the HTML file, specified as a character vector or string scalar.

Example: `grabcode('name.html')`

`url` — Web page address or location of the HTML file

character vector | string scalar

Web page address or location of HTML file, specified as a character vector or string scalar.

Example: `grabcode('https://www.mathworks.com/matlabcode.html')`

Limitations

- MATLAB Online only supports copying MATLAB code into a character array using the syntax `code = grabcode(filename)`. Copying MATLAB code from an HTML page or file into an untitled document in the Editor is not supported.

See Also

`publish`

Topics

"Publish and Share MATLAB Code"
"Share Live Scripts and Functions"

Introduced before R2006a

gradient

Numerical gradient

Syntax

```
FX = gradient(F)
[FX,FY] = gradient(F)
[FX,FY,FZ,...,FN] = gradient(F)
[ ___ ] = gradient(F,h)
[ ___ ] = gradient(F,hx,hy,...,hN)
```

Description

`FX = gradient(F)` returns the one-dimensional numerical gradient on page 1-5137 of vector `F`. The output `FX` corresponds to $\partial F/\partial x$, which are the differences in the x (horizontal) direction. The spacing between points is assumed to be `1`.

`[FX,FY] = gradient(F)` returns the x and y components of the two-dimensional numerical gradient on page 1-5137 of matrix `F`. The additional output `FY` corresponds to $\partial F/\partial y$, which are the differences in the y (vertical) direction. The spacing between points in each direction is assumed to be `1`.

`[FX,FY,FZ,...,FN] = gradient(F)` returns the `N` components of the numerical gradient on page 1-5137 of `F`, where `F` is an array with `N` dimensions.

`[ ___ ] = gradient(F,h)` uses `h` as a uniform spacing between points in each direction. You can specify any of the output arguments in previous syntaxes.

`[ ___ ] = gradient(F,hx,hy,...,hN)` specifies `N` spacing parameters for the spacing in each dimension of `F`.

Examples

Gradient of Vector

Calculate the gradient of a monotonically increasing vector.

```
x = 1:10
```

```
x = 1×10

     1     2     3     4     5     6     7     8     9    10
```

```
fx = gradient(x)
```

```
fx = 1×10

     1     1     1     1     1     1     1     1     1     1
```

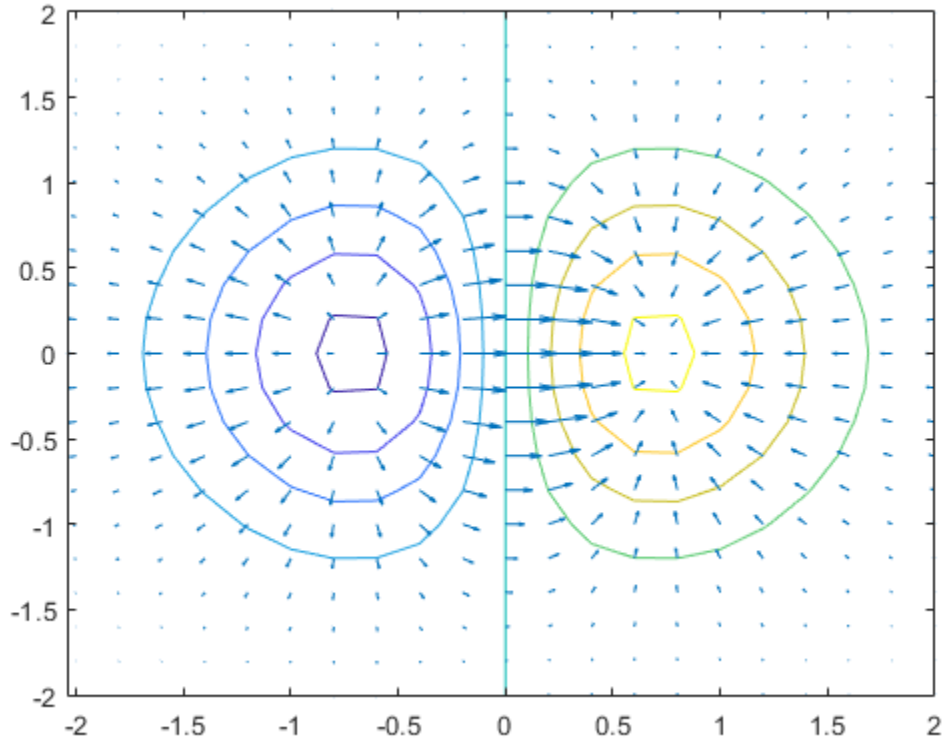
Contour Plot of Vector Field

Calculate the 2-D gradient of $xe^{-x^2 - y^2}$ on a grid.

```
x = -2:0.2:2;
y = x';
z = x .* exp(-x.^2 - y.^2);
[px,py] = gradient(z);
```

Plot the contour lines and vectors in the same figure.

```
figure
contour(x,y,z)
hold on
quiver(x,y,px,py)
hold off
```



Linear Function Approximation

Use the gradient at a particular point to linearly approximate the function value at a nearby point and compare it to the actual value.

The equation for linear approximation of a function value is

$$f(x) \approx f(x_0) + (\nabla f)_{x_0} \cdot (x - x_0) .$$

That is, if you know the value of a function $f(x_0)$ and the slope of the derivative $(\nabla f)_{x_0}$ at a particular point x_0, then you can use this information to approximate the value of the function at a nearby point $f(x) = f(x_0 + \epsilon)$.

Calculate some values of the sine function between -1 and 0.5. Then calculate the gradient.

```
y = sin(-1:0.25:0.5);
yp = gradient(y,0.25);
```

Use the function value and derivative at `x = 0.5` to predict the value of `sin(0.5005)`.

```
y_guess = y(end) + yp(end)*(0.5005 - 0.5)
```

```
y_guess = 0.4799
```

Compute the actual value for comparison.

```
y_actual = sin(0.5005)
```

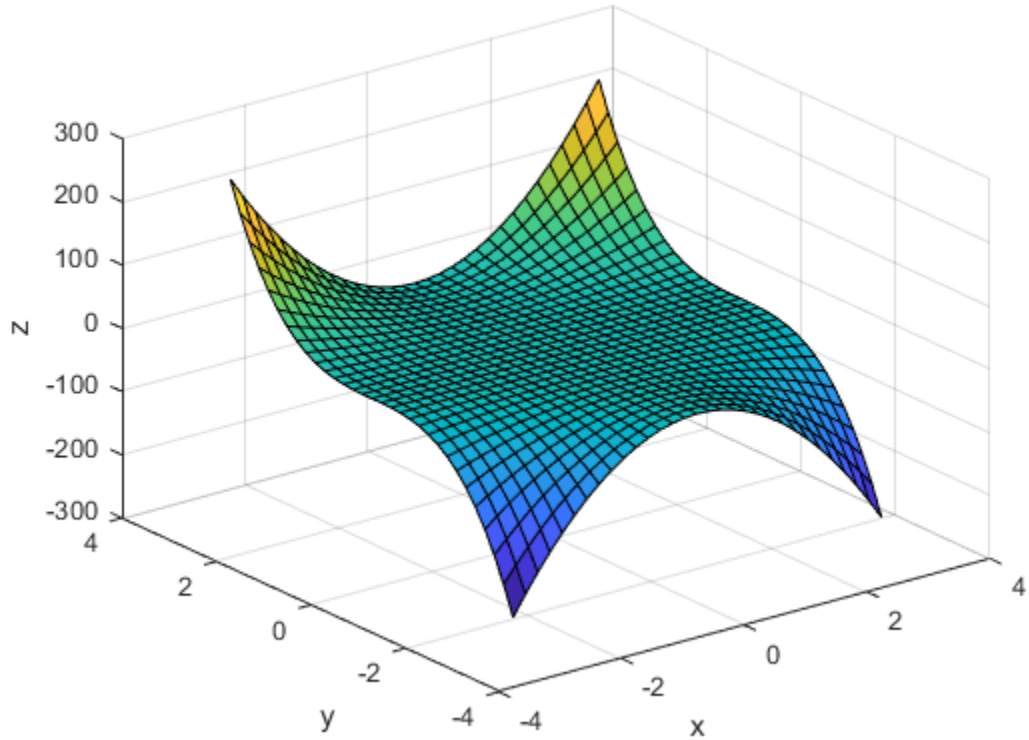
```
y_actual = 0.4799
```

Compute Gradient at Specified Point

Find the value of the gradient of a multivariate function at a specified point.

Consider the multivariate function $f(x, y) = x^2y^3$.

```
x = -3:0.2:3;
y = x';
f = x.^2 .* y.^3;
surf(x,y,f)
xlabel('x')
ylabel('y')
zlabel('z')
```



Calculate the gradient on the grid.

```
[fx,fy] = gradient(f,0.2);
```

Extract the value of the gradient at the point `(1,-2)`. To do this, first obtain the indices of the point you want to work with. Then, use the indices to extract the corresponding gradient values from `fx` and `fy`.

```
x0 = 1;
y0 = -2;
t = (x == x0) & (y == y0);
indt = find(t);
f_grad = [fx(indt) fy(indt)]
```

```
f_grad = 1×2

  -16.0000   12.0400
```

The exact value of the gradient of $f(x, y) = x^2y^3$ at the point (1,-2) is

$$\begin{aligned}(\nabla f)_{(1,\,-2)} &= 2xy^3\hat{i} + 3x^2y^2\hat{j} \\ &= -16\hat{i} + 12\hat{j}.\end{aligned}$$

Input Arguments

F — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

h — Uniform spacing between points
1 (default) | scalar

Uniform spacing between points in all directions, specified as a scalar.

Example: `[FX,FY] = gradient(F,2)`

Data Types: `single` | `double`
Complex Number Support: Yes

hx, hy, hN — Spacing between points (as separate inputs)
1 (default) | scalars | vectors

Spacing between points in each direction, specified as separate inputs of scalars or vectors. The number of inputs must match the number of array dimensions of `F`. Each input can be a scalar or vector:

- A scalar specifies a constant spacing in that dimension.
- A vector specifies the coordinates of the values along the corresponding dimension of `F`. In this case, the length of the vector must match the size of the corresponding dimension.

Example: `[FX,FY] = gradient(F,0.1,2)`

Example: `[FX,FY] = gradient(F,[0.1 0.3 0.5],2)`

Example: `[FX,FY] = gradient(F,[0.1 0.3 0.5],[2 3 5])`

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

FX, FY, FZ, FN — Numerical gradients
arrays

Numerical gradients, returned as arrays of the same size as `F`. The first output `FX` is always the gradient along the 2nd dimension of `F`, going across columns. The second output `FY` is always the gradient along the 1st dimension of `F`, going across rows. For the third output `FZ` and the outputs that follow, the `N`th output is the gradient along the `N`th dimension of `F`.

More About

Numerical Gradient

The *numerical gradient* of a function is a way to estimate the values of the partial derivatives in each dimension using the known values of the function at certain points.

For a function of two variables, $F(x,y)$, the gradient is

$$\nabla F = \frac{\partial F}{\partial x}\hat{i} + \frac{\partial F}{\partial y}\hat{j} \ .$$

The gradient can be thought of as a collection of vectors pointing in the direction of increasing values of F. In MATLAB, you can compute numerical gradients for functions with any number of variables. For a function of N variables, $F(x,y,z, \ldots)$, the gradient is

$$\nabla F = \frac{\partial F}{\partial x}\hat{i} + \frac{\partial F}{\partial y}\hat{j} + \frac{\partial F}{\partial z}\hat{k} + \ldots + \frac{\partial F}{\partial N}\hat{n} \ .$$

Tips

- Use `diff` or a custom algorithm to compute multiple numerical derivatives, rather than calling `gradient` multiple times.

Algorithms

`gradient` calculates the *central difference* for interior data points. For example, consider a matrix with unit-spaced data, `A`, that has horizontal gradient `G = gradient(A)`. The interior gradient values, `G(:,j)`, are

```
G(:,j) = 0.5*(A(:,j+1) - A(:,j-1));
```

The subscript `j` varies between `2` and `N-1`, with `N = size(A,2)`.

`gradient` calculates values along the edges of the matrix with *single-sided differences*:

```
G(:,1) = A(:,2) - A(:,1);
G(:,N) = A(:,N) - A(:,N-1);
```

If you specify the point spacing, then `gradient` scales the differences appropriately. If you specify two or more outputs, then the function also calculates differences along other dimensions in a similar manner. Unlike the `diff` function, `gradient` returns an array with the same number of elements as the input.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`del2` | `diff`

Topics

“Calculate Tangent Plane to Surface”

Introduced before R2006a

addedge

Add new edge to graph

Syntax

```
H = addedge(G,s,t)
H = addedge(G,s,t,w)
H = addedge(G,s,t,EdgeTable)
H = addedge(G,EdgeTable)
```

Description

`H = addedge(G,s,t)` adds an edge to graph `G` between nodes `s` and `t`. If a node specified by `s` or `t` is not present in `G`, then that node is added. The new graph, `H`, is equivalent to `G`, but includes the new edge and any required new nodes.

`H = addedge(G,s,t,w)` also specifies weights, `w`, for the edges between `s` and `t`.

`H = addedge(G,s,t,EdgeTable)` adds edges between the nodes `s` and `t` with attributes specified by the table, `EdgeTable`.

`H = addedge(G,EdgeTable)` adds edges with attributes specified by the table, `EdgeTable`. The `EdgeTable` input must be able to be concatenated with `G.Edges`.

Examples

Add Edges to Graph

Add two new edges to an existing graph. Since one of the new edges references a node that does not exist, `addedge` automatically adds the required fourth node to the graph.

```
G = graph([1 2],[2 3])
```

```
G = 
  graph with properties:
```

```
    Edges: [2x1 table]
    Nodes: [3x0 table]
```

```
G = addedge(G,[2 1],[4 3])
```

```
G = 
  graph with properties:

    Edges: [4x1 table]
    Nodes: [4x0 table]
```

Add Weighted Edges to Graph

Create a directed graph with weighted edges.

```
G = digraph({'A' 'B' 'C'}, {'D' 'C' 'D'}, [10 20 45])
```

```
G = 
  digraph with properties:

    Edges: [3x2 table]
    Nodes: [4x1 table]
```

Add three new weighted edges to the graph. `addedge` also automatically adds any required new nodes to the graph.

```
G = addedge(G, {'A' 'D' 'E'}, {'E' 'B' 'D'}, [5 30 5])
```

```
G = 
  digraph with properties:

    Edges: [6x2 table]
    Nodes: [5x1 table]
```

Add Edges with Attributes to Graph

Create a graph whose edges have the attributes `Weight` and `Power`. Use an edge table to create the graph.

```
EdgeTable = table([1 2; 2 3; 2 4; 2 5; 5 6; 5 7; 5 8], ...
    {'on','off','off','on','on','on','off'}',[10 20 20 10 10 10 20]', ...
    'VariableNames',{'EndNodes','Power','Weight'});
G = graph(EdgeTable)
```

```
G = 
  graph with properties:

    Edges: [7x3 table]
    Nodes: [8x0 table]
```

Add two new edges to the graph by creating a smaller table that can be concatenated to `G.Edges`. Note that this smaller table must use the same order of variables as `G.Edges`.

```
NewEdges = table([5 9; 3 6], {'on' 'off'}', [10 20]', ...
    'VariableNames',{'EndNodes','Power','Weight'});
G = addedge(G,NewEdges)
```

```
G = 
  graph with properties:

    Edges: [9x3 table]
    Nodes: [9x0 table]
```

View the new edge list of the graph, which includes the added edges.

```
G.Edges
```

```
ans=9×3 table
    EndNodes      Power      Weight
    ________     _______     ______

     1    2     {'on' }       10
     2    3     {'off'}       20
     2    4     {'off'}       20
     2    5     {'on' }       10
     3    6     {'off'}       20
     5    6     {'on' }       10
```

```
    5    7     {'on' }      10
    5    8     {'off'}      20
    5    9     {'on' }      10
```

Input Arguments

G — Input graph

graph object | digraph object

Input graph, specified as either a `graph` or `digraph` object. Use `graph` to create an undirected graph or `digraph` to create a directed graph.

Example: `G = graph(1,2)`

Example: `G = digraph([1 2],[2 3])`

s,t — Node pairs (as separate arguments)

node indices | node names

Node pairs, specified as separate arguments of node indices or node names. Similarly located elements in `s` and `t` specify the source and target nodes for edges in the graph.

If you add edges using node names that are not already present in the graph, then `addedge` adds the new node names to the bottom of the `g.Nodes.Name` table. If `s` and `t` are categorical arrays, then the categories of `s` and `t` are used as node names. This can include categories that are not elements in `s` or `t`.

This table shows the different ways to refer to one or more nodes either by their numeric node indices or by their node names.

Form	Single Node	Multiple Nodes
Node index	Scalar **Example:** `1`	Vector **Example:** `[1 2 3]`
Node name	Character vector **Example:** `'A'`	Cell array of character vectors **Example:** `{'A' 'B' 'C'}`

Form	Single Node	Multiple Nodes
	String scalar **Example:** `"A"`	String array **Example:** `["A" "B" "C"]`
	Categorical array **Example:** `categorical("A")`	Categorical array **Example:** `categorical(["A" "B" "C"])`

Example: `G = addedge(G, [1 2], [3 4])` adds two edges to the graph: one from node 1 to node 3, and one from node 2 to node 4.

Example: `G = addedge(G, {'a' 'a'; 'b' 'c'}, {'b' 'c'; 'c' 'e'})` adds four edges to the graph, the first of which goes from `'a'` to `'b'`.

`w` — Edge weights
scalar | vector | matrix

Edge weights, specified as a scalar, vector, or matrix.

- If `w` is a scalar or row vector, then it is scalar expanded to specify a weight for each edge in `s` and `t`.
- If `w` is a column vector, then it must have the same length as `s(:)` and `t(:)`.
- If `w` is a matrix, then it must have the same number of elements as `s(:)` and `t(:)`.

Example: `G = addedge(G, [2 2], [4 5], [1 100]')` adds two edges with weights of 1 and `100`.

Data Types: `single` | `double`
Complex Number Support: Yes

`EdgeTable` — Edge attributes
table

Edge attributes, specified as a table. If you do not specify `s` and `t` to define the graph edges being added, then the first variable in `EdgeTable` is required to be a two-column matrix called `EndNodes` that defines the graph edges being added.

`EdgeTable` must have these general properties:

- For weighted graphs, `EdgeTable` must contain a variable `Weight`.
- If the graph has other edge attributes, then `EdgeTable` must contain all of the same variables as `G.Edges` to ensure compatibility.
- The order of variables in `EdgeTable` must be the same as that of `G.Edges`.

Data Types: `table`

Output Arguments

`H` — Output graph
`graph` object | `digraph` object

Output graph, returned as a `graph` or `digraph` object.

Compatibility Considerations

Change in handling of duplicate edges

Behavior changed in R2018a

`graph`, `digraph`, and `addedge` no longer produce errors when they encounter duplicate edges. Instead, the duplicate edges are added to the graph and the result is a multigraph. The `ismultigraph` function is useful to detect this situation, and `simplify` provides an easy way to remove the extra edges.

See Also

`addnode` | `digraph` | `graph` | `numedges` | `rmedge`

Topics

"Directed and Undirected Graphs"
"Modify Nodes and Edges of Existing Graph"
"Add Graph Node Names, Edge Weights, and Other Attributes"

Introduced in R2015b

addnode

Add new node to graph

Syntax

```
H = addnode(G,nodeIDs)
H = addnode(G,numNodes)
H = addnode(G,NodeProps)
```

Description

`H = addnode(G,nodeIDs)` adds the nodes specified by `nodeIDs` to graph `G`. The node names in `nodeIDs` must not refer to nodes already present in `G`.

`H = addnode(G,numNodes)` adds a number of new nodes to `G` equal to `numNodes`. If `G` contains nodes with names, then the new nodes are assigned sequential names indicating their row placement in `G.Nodes.Name`. For example, `'Node5'` is located at `G.Nodes.Name(5)`.

`H = addnode(G,NodeProps)` adds new nodes to `G` with the node properties in `NodeProps`. One node is added for each row in `NodeProps`. The `NodeProps` table must be able to be concatenated to `G.Nodes`, so that the result is `H.Nodes = [G.Nodes; NodeProps]`.

Examples

Add Nodes to Graph

Add two nodes to a graph that does not have node names.

```
G = graph([1 2 3],[2 3 4])
```

```
G = 
  graph with properties:
```

```
    Edges: [3x1 table]
    Nodes: [4x0 table]
```

```
G = addnode(G,2)
```

```
G =
  graph with properties:

    Edges: [3x1 table]
    Nodes: [6x0 table]
```

Add node names to the graph, and then add five additional new nodes. The auto-generated names for the new nodes indicate their placement in `G.Nodes.Name`.

```
G.Nodes.Name = {'A' 'B' 'C' 'D' 'E' 'F'}'
```

```
G =
  graph with properties:

    Edges: [3x1 table]
    Nodes: [6x1 table]
```

```
G = addnode(G,5);
G.Nodes
```

```
ans=11×1 table
       Name
    __________

    {'A'     }
    {'B'     }
    {'C'     }
    {'D'     }
    {'E'     }
    {'F'     }
    {'Node7' }
    {'Node8' }
    {'Node9' }
    {'Node10'}
    {'Node11'}
```

Add Named Nodes to Graph

Create a directed graph with named nodes, and then add two named nodes to the graph.

```
G = digraph({'A' 'B' 'C'},{'D' 'C' 'D'})
```

```
G = 
  digraph with properties:

    Edges: [3x1 table]
    Nodes: [4x1 table]
```

```
G = addnode(G,{'E' 'F'})
```

```
G = 
  digraph with properties:

    Edges: [3x1 table]
    Nodes: [6x1 table]
```

If the graph does not already have node names, then adding named nodes to the graph automatically generates names for the other nodes.

Create a directed graph without node names, and then add two named nodes to the graph.

```
H = digraph([1 2 3],[4 3 4])
```

```
H = 
  digraph with properties:

    Edges: [3x1 table]
    Nodes: [4x0 table]
```

```
H = addnode(H,{'E','F'});
H.Nodes
```

```
ans=6×1 table
      Name
    _________
```

```
    {'Node1'}
    {'Node2'}
    {'Node3'}
    {'Node4'}
    {'E'    }
    {'F'    }
```

Add Nodes with Attributes to Graph

Create a graph whose nodes represent airports.

```
G = graph({'JFK' 'LAX'}, {'LAX' 'DEN'})
```

```
G = 
  graph with properties:

    Edges: [2x1 table]
    Nodes: [3x1 table]
```

Add a node attribute to indicate whether each airport has free WIFI.

```
G.Nodes.WIFI = [false true true]';
G.Nodes
```

```
ans=3×2 table
     Name      WIFI
    _______    _____

    {'JFK'}    false
    {'LAX'}    true 
    {'DEN'}    true 
```

Add two new nodes to the graph by creating a table, `NodeProps`, containing the node name and WIFI status of each new node. Use `addnode` to concatenate `NodeProps` to `G.Nodes`.

```
NodeProps = table({'ATL' 'ANC'}', [false true]', ...
    'VariableNames', {'Name' 'WIFI'});
G = addnode(G, NodeProps);
```

View the modified node table.

```
G.Nodes
```

```
ans=5×2 table
     Name      WIFI
    _______    _____

    {'JFK'}    false
    {'LAX'}    true
    {'DEN'}    true
    {'ATL'}    false
    {'ANC'}    true
```

Input Arguments

G — Input graph
graph object | digraph object

Input graph, specified as either a `graph` or `digraph` object. Use `graph` to create an undirected graph or `digraph` to create a directed graph.

Example: `G = graph(1,2)`

Example: `G = digraph([1 2],[2 3])`

nodeIDs — Node names
one or more node names

Node names, specified as one or more node names in one of these forms:

- **Single Node** — Character vector `'A'` or string scalar `"A"`.
- **Multiple Nodes** — Cell array of character vectors `{'A' 'B' 'C'}` or string array `["A" "B" "C"]`.

Example: `H = addnode(G,'A')`

Example: `H = addnode(G,["A" "B" "C"])`

Data Types: `char` | `cell` | `string`

numNodes — Number of nodes to add
nonnegative numeric scalar

Number of nodes to add, specified as a nonnegative numeric scalar.

NodeProps — Node attributes
table

Node attributes, specified as a table. `NodeProps` can contain any number of variables to describe attributes of the graph nodes, but must be able to be concatenated to `G.Nodes`, so that the result is `H.Nodes = [G.Nodes; NodeProps]`. For node names, use the variable `Name`, since this variable name is used by some graph functions. If there is a variable `Name`, then it must be a cell array of character vectors or string array specifying a unique name in each row. See `table` for more information on constructing a table.

Data Types: `table`

Output Arguments

H — Output graph
`graph` object | `digraph` object

Output graph, returned as a `graph` or `digraph` object.

See Also

`addedge` | `digraph` | `graph` | `numnodes` | `rmnode`

Topics

"Directed and Undirected Graphs"
"Modify Nodes and Edges of Existing Graph"
"Add Graph Node Names, Edge Weights, and Other Attributes"

Introduced in R2015b

adjacency

Graph adjacency matrix

Syntax

```
A = adjacency(G)

A = adjacency(G,'weighted')
A = adjacency(G,weights)
```

Description

`A = adjacency(G)` returns the sparse adjacency matrix for graph `G`. If `(i,j)` is an edge in `G`, then `A(i,j) = 1`. Otherwise, `A(i,j) = 0`.

`A = adjacency(G,'weighted')` returns a weighted adjacency matrix, where for each edge `(i,j)`, the value `A(i,j)` contains the weight of the edge. If the graph has no edge weights, then `A(i,j)` is set to 1. For this syntax, `G` must be a simple graph such that `ismultigraph(G)` returns `false`.

`A = adjacency(G,weights)` returns a weighted adjacency matrix with edge weights given by the vector `weights`. For each edge `(i,j)` in `G`, the adjacency matrix has value `A(i,j) = weights(findedge(G,i,j))`. For this syntax, `G` must be a simple graph such that `ismultigraph(G)` returns `false`.

Examples

Adjacency Matrix of Graph

Create a directed graph using an edge list, and then find the equivalent adjacency matrix representation of the graph. The adjacency matrix is returned as a sparse matrix.

```
s = [1 1 1 2 2 3];
t = [2 3 4 5 6 7];
G = digraph(s,t)
```

```
G = 
  digraph with properties:

    Edges: [6x1 table]
    Nodes: [7x0 table]
```

```
A = adjacency(G)
```

```
A = 
   (1,2)        1
   (1,3)        1
   (1,4)        1
   (2,5)        1
   (2,6)        1
   (3,7)        1
```

Adjacency Matrix of Weighted Graph

Create an undirected graph using an upper triangular adjacency matrix. When constructing a graph with an adjacency matrix, the nonzero values in the matrix correspond to edge weights.

```
A = [0 5 3 0;0 0 1 2; 0 0 0 11; 0 0 0 0]
```

```
A = 4×4

     0     5     3     0
     0     0     1     2
     0     0     0    11
     0     0     0     0
```

```
G = graph(A,'upper')
```

```
G = 
  graph with properties:
```

```
    Edges: [5x2 table]
    Nodes: [4x0 table]
```

```
G.Edges
```

```
ans=5×2 table
    EndNodes    Weight
    ________    ______

     1    2        5
     1    3        3
     2    3        1
     2    4        2
     3    4       11
```

Use `adjacency` to return the adjacency matrix of the graph. Regardless of the form of adjacency matrix used to construct the graph, the `adjacency` function always returns a symmetric and sparse adjacency matrix containing only 1s and 0s.

```
B = adjacency(G)
```

```
B =
   (2,1)        1
   (3,1)        1
   (1,2)        1
   (3,2)        1
   (4,2)        1
   (1,3)        1
   (2,3)        1
   (4,3)        1
   (2,4)        1
   (3,4)        1
```

Reconstruct Weighted Adjacency Matrix

Create a weighted graph.

```
G = digraph([1 1 1 2 3 4],[2 3 4 4 2 3],[5 6 7 8 9 10]);
G.Edges
```

```
ans=6×2 table
    EndNodes    Weight
    ________    ______

     1    2        5
     1    3        6
     1    4        7
     2    4        8
     3    2        9
     4    3       10
```

Find the adjacency matrix of the graph.

```
A = adjacency(G)
```

```
A =
   (1,2)        1
   (3,2)        1
   (1,3)        1
   (4,3)        1
   (1,4)        1
   (2,4)        1
```

This form of the adjacency matrix does not include the edge weights. Use the `'weighted'` option to include the edge weights in the adjacency matrix.

```
A = adjacency(G,'weighted')
```

```
A =
   (1,2)        5
   (3,2)        9
   (1,3)        6
   (4,3)       10
   (1,4)        7
   (2,4)        8
```

Preview a full storage version of the matrix. Since `G` is a directed graph, the adjacency matrix is not symmetric. However, the adjacency matrix *is* symmetric for undirected graphs.

```
B = full(A)
```

```
B = 4×4

     0     5     6     7
     0     0     0     8
     0     9     0     0
     0     0    10     0
```

Input Arguments

G — Input graph
graph object | digraph object

Input graph, specified as either a graph or digraph object. Use graph to create an undirected graph or digraph to create a directed graph.

Example: G = graph(1,2)

Example: G = digraph([1 2],[2 3])

weights — Edge weights
vector

Edge weights, specified as a vector.

Example: A = adjacency(G,[1 2 3 4])

Data Types: double | logical
Complex Number Support: Yes

Output Arguments

A — Adjacency matrix
sparse matrix

Adjacency matrix, returned as a sparse matrix. The size of A is numnodes(G)-by-numnodes(G).

Tips

- Edges with weight zero are not visible in the sparse adjacency matrix returned by `adjacency`. This means that a weighted adjacency matrix can represent a weighted graph only if there are no edges of weight zero.

See Also

`digraph` | `graph` | `incidence` | `laplacian`

Introduced in R2015b

bctree

Block-cut tree graph

Syntax

```
tree = bctree(G)
[tree,ind] = bctree(G)
```

Description

`tree = bctree(G)` returns the block-cut tree of graph `G`, such that each node in `tree` represents either a biconnected component on page 1-5163 or cut vertex on page 1-5165 of `G`. A node representing a cut vertex is connected to all nodes representing biconnected components that contain that cut vertex.

`[tree,ind] = bctree(G)` also returns a vector of numeric node indices mapping the nodes of `G` into the nodes of `tree`.

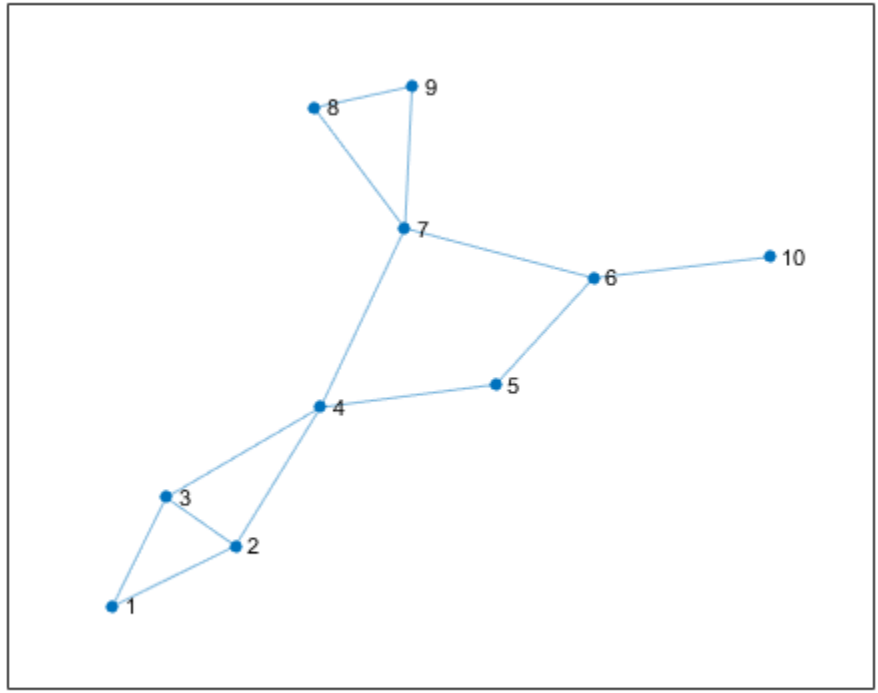
Examples

Compute Block-Cut Tree of Graph

Compute the block-cut tree of a graph, view the resulting node properties, and then highlight the cut vertices in the graph plot.

Create and plot a graph.

```
s = [1 1 2 2 3 4 4 5 6 6 7 7 8];
t = [2 3 3 4 4 5 7 6 7 10 8 9 9];
G = graph(s,t);
p = plot(G);
```



Compute the block-cut tree of the graph and view the node properties.

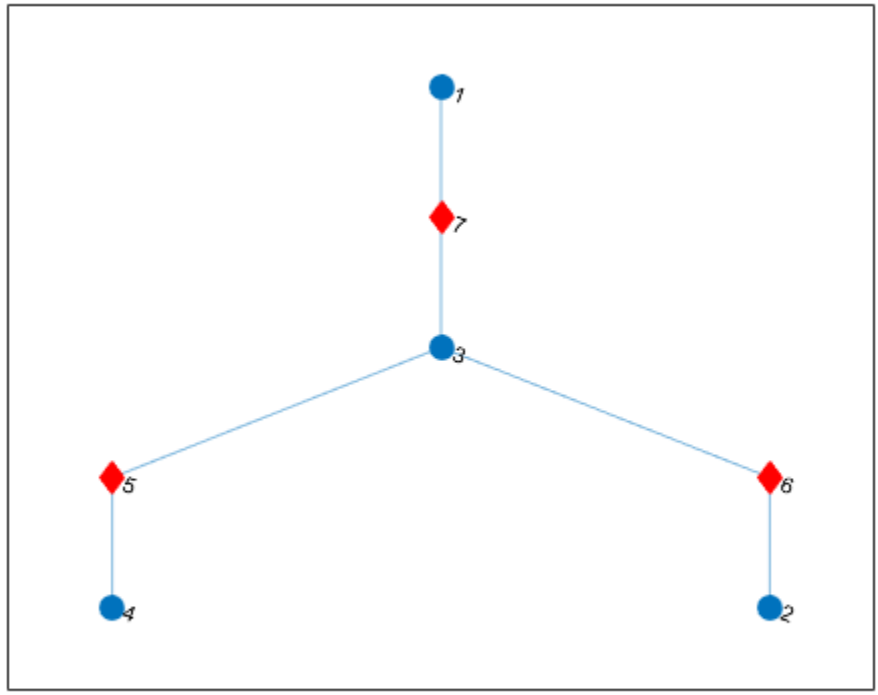
```
tree = bctree(G);
tree.Nodes
```

```
ans=7×3 table
    IsComponent    ComponentIndex    CutVertexIndex
    ___________    ______________    ______________

       true              1                 0
       true              2                 0
       true              3                 0
       true              4                 0
       false             0                 4
```

```
false              0                  6
false              0                  7
```

Plot the block-cut tree using red diamond markers for the nodes that represent cut vertices. The circular nodes represent the biconnected components in the original graph.

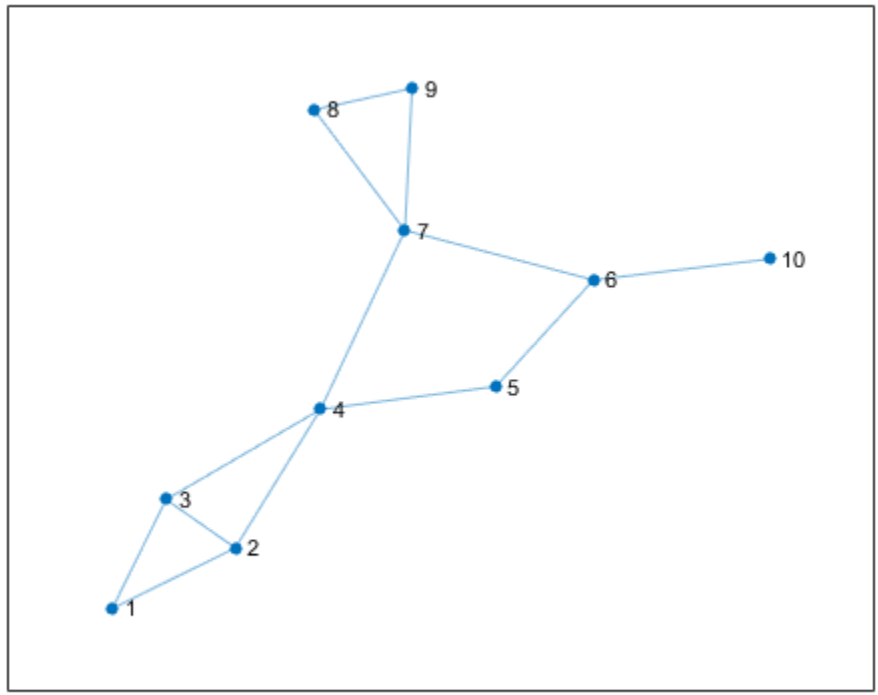
```
p2 = plot(tree,'MarkerSize',9);
highlight(p2,5:7,'Marker','d','NodeColor','r')
```



Return Node Indices for Block-Cut Tree

Create and plot a graph.

```
s = [1 1 2 2 3 4 4 5 6 6 7 7 8];
t = [2 3 3 4 4 5 7 6 7 10 8 9 9];
G = graph(s,t);
p = plot(G);
```



Compute the block-cut tree `tr` of the graph, and specify a second output `ix` to return the node indices.

```
[tr,ix] = bctree(G)
```

```
tr = 
  graph with properties:

    Edges: [6x1 table]
    Nodes: [7x3 table]


ix = 1×10

     4     4     4     5     3     6     7     1     1     2
```

Each index `ix(j)` indicates the node in the block-cut tree that represents node `j` in the input graph. For example, node 4 in `tr` represents a component in `G` that contains nodes 1, 2, and 3, so the first three entries in `ix` are all 4.

Input Arguments

`G` — Input graph
`graph` object

Input graph, specified as a `graph` object. Use `graph` to create an undirected graph object.

Example: `G = graph(1,2)`

Output Arguments

`tree` — Block-cut tree graph
`graph` object

Block-cut tree graph, returned as a `graph` object. `tree` contains a node for each cut vertex in `G` and a node for each biconnected component in `G`. The nodes table `tree.Nodes` contains additional node attributes to describe what each node represents:

- `tree.Nodes.IsComponent(i)` — Value equal to logical `1` (`true`) if node `i` represents a biconnected component. Otherwise, the value is equal to and logical `0` (`false`).
- `tree.Nodes.ComponentIndex(i)` — Index indicating the component represented by node `i`. The value is zero if node `i` represents a cut vertex.

- `tree.Nodes.CutVertexIndex(i)` — Index indicating the cut vertex represented by node `i`. The value is zero if node `i` represents a biconnected component.

`ind` — Node indices
vector

Node indices, returned as a numeric vector. `ind(i)` is the node in the output graph `tree` that represents node `i` in the input graph `G`:

- If node `i` is a cut vertex in graph `G`, then `ind(i)` is the associated node in `tree`.
- If node `i` is not a cut vertex, but belongs to one of the biconnected components in graph `G`, then `ind(i)` is the node in `tree` representing that biconnected component.
- If node `i` is an isolated node in graph `G`, then `ind(i)` is zero.

More About

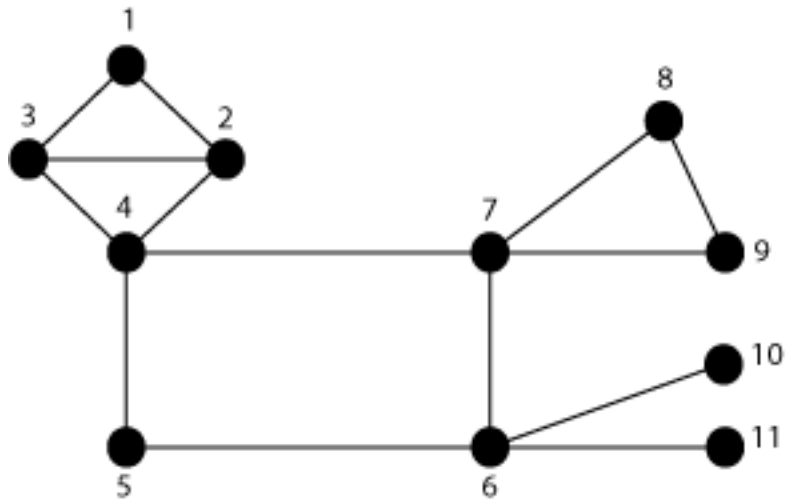
Biconnected Components

A biconnected component of a graph is a maximally biconnected subgraph. A graph is biconnected if it does not contain any cut vertices.

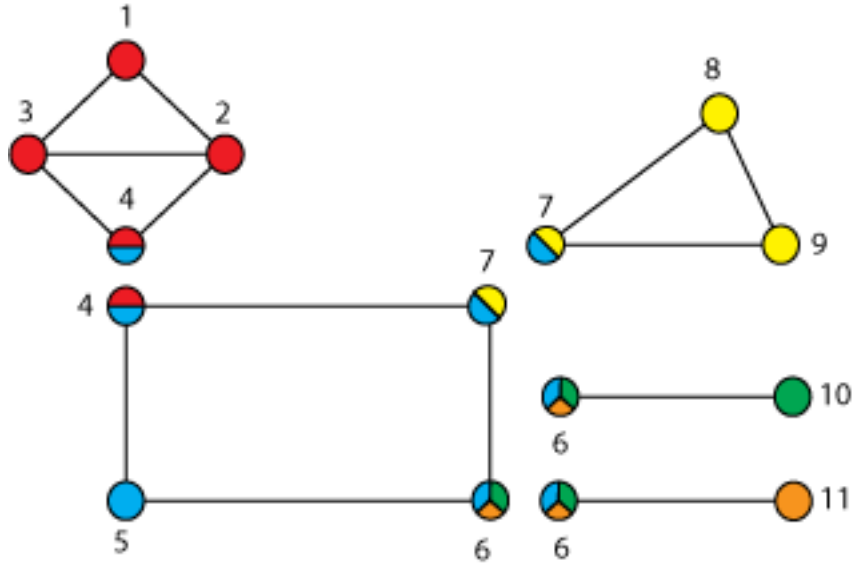
Decomposing a graph into its biconnected components helps to measure how well-connected the graph is. You can decompose any connected graph into a tree of biconnected components, called the *block-cut tree*. The blocks in the tree are attached at shared vertices, which are the cut vertices.

The illustration depicts:

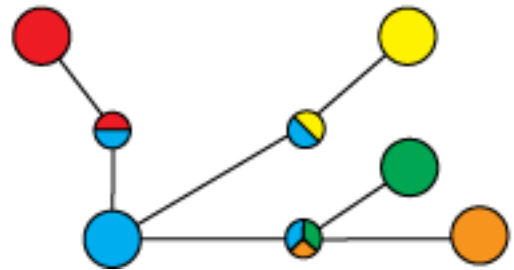
- (a) An undirected graph with 11 nodes.
- (b) Five biconnected components of the graph, with the cut vertices of the original graph colored for each component to which they belong.
- (c) Block-cut tree of the graph, which contains a node for each biconnected component (as large circles) and a node for each cut vertex (as smaller multicolored circles). In the block-cut tree, an edge connects each cut vertex to each component to which it belongs.



(a)



(b)



(c)

1-5164

Cut Vertices

Also known as *articulation points*, cut vertices are graph nodes whose removal increases the number of connected components. In the previous illustration, the cut vertices are those nodes with more than one color: nodes 4, 6, and 7.

See Also

`biconncomp` | `condensation`

Topics

"Directed and Undirected Graphs"
"Visualize Breadth-First and Depth-First Search"

Introduced in R2016b

bfsearch

Breadth-first graph search

Syntax

```
v = bfsearch(G,s)
T = bfsearch(G,s,events)
[T,E] = bfsearch(G,s,events)
[ ___ ] = bfsearch( ___ ,'Restart',tf)
```

Description

`v = bfsearch(G,s)` applies breadth-first search on page 1-5182 to graph `G` starting at node `s`. The result is a vector of node IDs in order of their discovery.

`T = bfsearch(G,s,events)` customizes the output of the breadth-first search by flagging one or more search events. For example, `T = bfsearch(G,s,'allevents')` returns a table containing all flagged events, and `X = bfsearch(G,s,'edgetonew')` returns a matrix or cell array of edges.

`[T,E] = bfsearch(G,s,events)` additionally returns a vector of edge indices `E` when `events` is set to `'edgetonew'`, `'edgetodiscovered'`, or `'edgetofinished'`. The edge indices are for unique identification of edges in a multigraph.

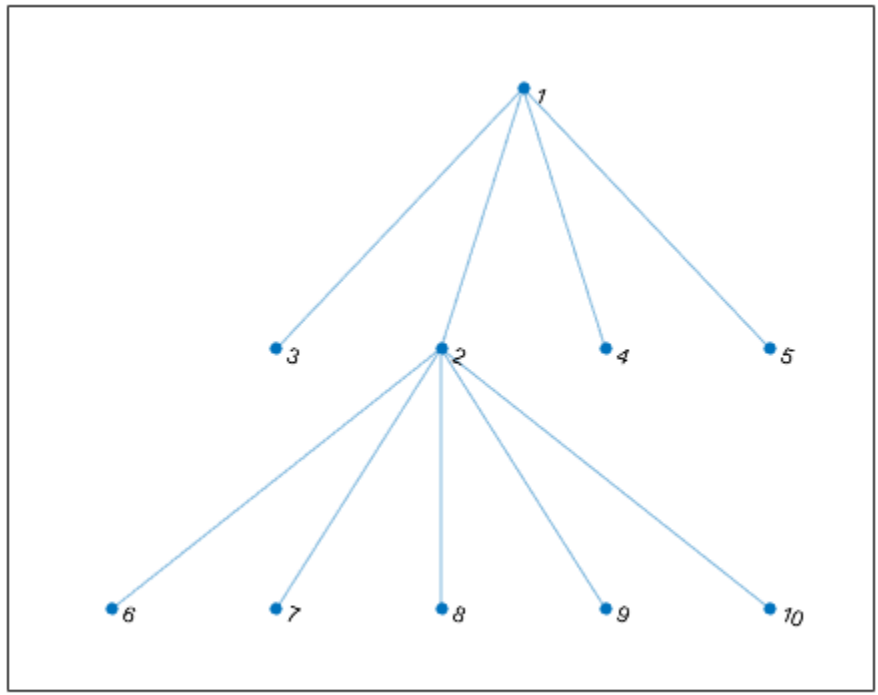
`[ ___ ] = bfsearch( ___ ,'Restart',tf)`, where `tf` is `true`, restarts the search if no new nodes are reachable from the discovered nodes. You can use any of the input or output argument combinations in previous syntaxes. This option ensures that the breadth-first search reaches all nodes and edges in the graph, even if they are not reachable from the starting node, `s`.

Examples

Perform Breadth-First Graph Search

Create and plot a graph.

```
s = [1 1 1 1 2 2 2 2 2];
t = [3 5 4 2 6 10 7 9 8];
G = graph(s,t);
plot(G)
```



Perform a breadth-first search of the graph starting at node 2. The result indicates the order of node discovery.

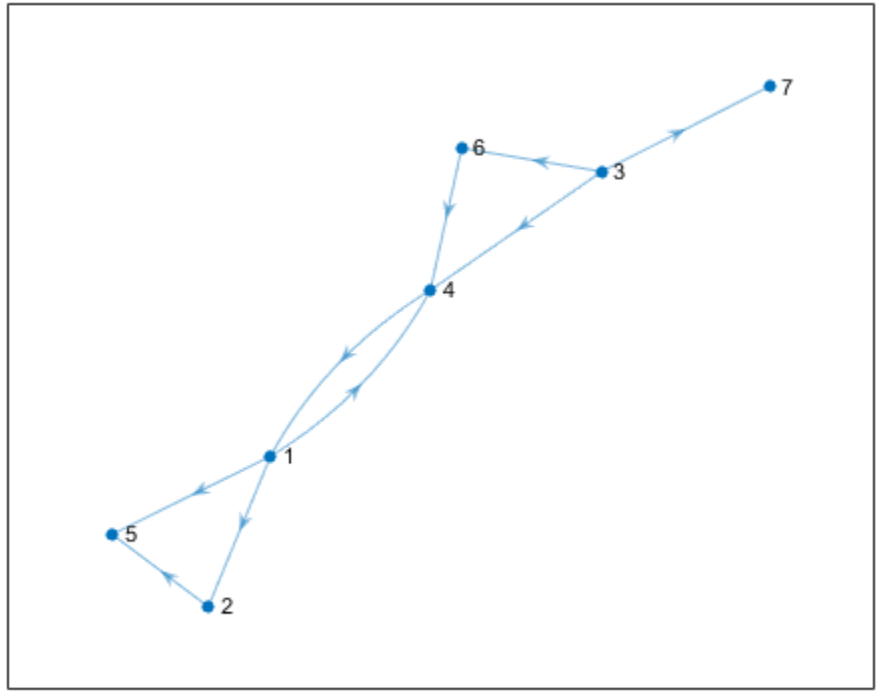
```
v = bfsearch(G,2)
```

```
v = 10×1

     2
     1
     6
     7
     8
     9
    10
     3
     4
     5
```

Breadth-First Graph Search with All Events

Create and plot a directed graph.

```
s = [1 1 1 2 3 3 3 4 6];
t = [2 4 5 5 6 7 4 1 4];
G = digraph(s,t);
plot(G)
```



Perform a breadth-first search on the graph starting at node 1. Specify `'allevents'` to return a table containing all of the events in the algorithm.

```
T = bfsearch(G,1,'allevents')
```

```
T=14×4 table
         Event           Node       Edge       EdgeIndex
    ________________     ____    __________    _________

    startnode              1     NaN    NaN       NaN
    discovernode           1     NaN    NaN       NaN
    edgetonew            NaN       1      2         1
    discovernode           2     NaN    NaN       NaN
    edgetonew            NaN       1      4         2
```

```
    discovernode          4     NaN    NaN       NaN
    edgetonew           NaN       1      5         3
    discovernode          5     NaN    NaN       NaN
    finishnode            1     NaN    NaN       NaN
    edgetodiscovered    NaN       2      5         4
    finishnode            2     NaN    NaN       NaN
    edgetofinished      NaN       4      1         8
    finishnode            4     NaN    NaN       NaN
    finishnode            5     NaN    NaN       NaN
```

To follow the steps in the algorithm, read the events in the table from top to bottom. For example:

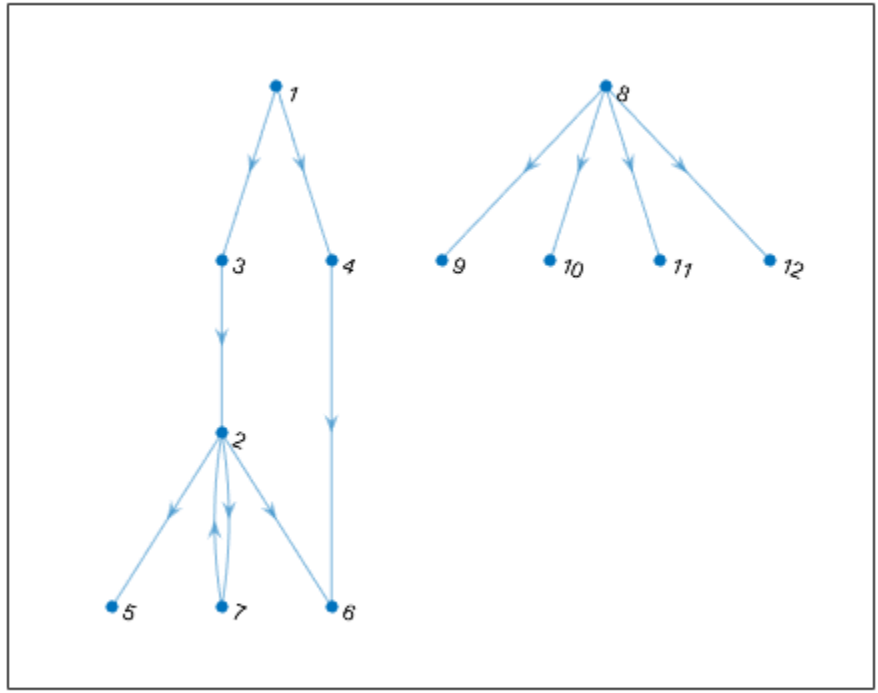
1. The algorithm begins at node 1
2. An edge is discovered between node 1 and node 2
3. Node 2 is discovered
4. and so on...

Breadth-First Graph Search with Multiple Components

Perform a breadth-first search of a graph with multiple components, and then highlight the graph nodes and edges based on the search results.

Create and plot a directed graph. This graph has two weakly connected components.

```
s = [1 1 2 2 2 3 4 7 8 8 8 8];
t = [3 4 7 5 6 2 6 2 9 10 11 12];
G = digraph(s,t);
p = plot(G,'Layout','layered');
```




```
c = conncomp(G,'Type','weak')
```

```
c = 1×12

     1     1     1     1     1     1     1     2     2     2     2     2
```

Perform a breadth-first search of the graph starting at node 2, and flag the `'edgetonew'`, `'edgetofinished'`, and `'startnode'` events. Specify `Restart` as `true` to make the search restart whenever there are remaining nodes that cannot be reached.

```
events = {'edgetonew','edgetofinished','startnode'};
T = bfsearch(G,2,events,'Restart',true)
```

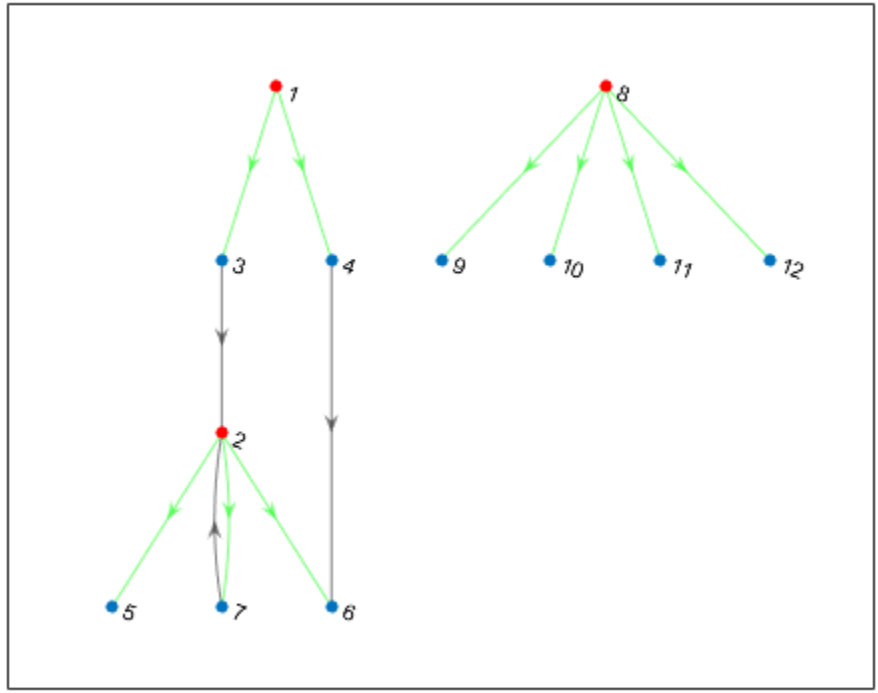
```
T=15×4 table
        Event          Node       Edge       EdgeIndex
    _______________    ____    __________    _________

    startnode            2     NaN    NaN       NaN
    edgetonew          NaN       2      5         3
    edgetonew          NaN       2      6         4
    edgetonew          NaN       2      7         5
    edgetofinished     NaN       7      2         8
    startnode            1     NaN    NaN       NaN
    edgetonew          NaN       1      3         1
    edgetonew          NaN       1      4         2
    edgetofinished     NaN       3      2         6
    edgetofinished     NaN       4      6         7
    startnode            8     NaN    NaN       NaN
    edgetonew          NaN       8      9         9
    edgetonew          NaN       8     10        10
    edgetonew          NaN       8     11        11
    edgetonew          NaN       8     12        12
```

When `Restart` is `true`, the `'startnode'` event returns information about where and when the algorithm restarts the search.

Highlight the graph based on event:

- Color the starting nodes red.
- Green edges are for `'edgetonew'`
- Black edges are for `'edgetofinished'`

```
highlight(p, 'Edges', T.EdgeIndex(T.Event == 'edgetonew'), 'EdgeColor', 'g')
highlight(p, 'Edges', T.EdgeIndex(T.Event == 'edgetofinished'), 'EdgeColor', 'k')
highlight(p,T.Node(~isnan(T.Node)),'NodeColor','r')
```

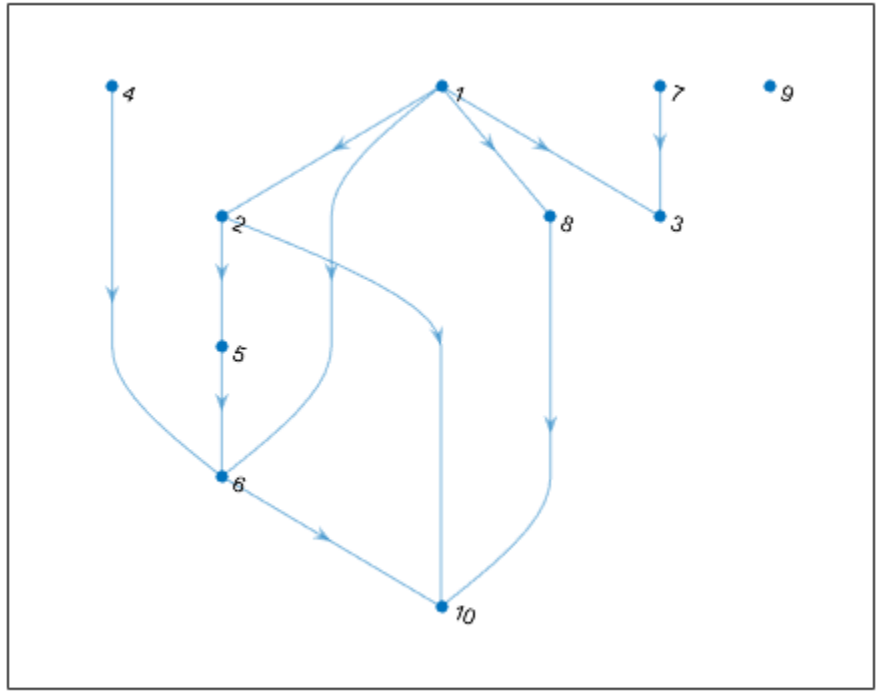


Determine if Graph is Bipartite

Use breadth-first search to determine that a graph is bipartite, and return the relevant partitions. A bipartite graph is a graph that has nodes you can divide into two sets, A and B, with each edge in the graph connecting a node in A to a node in B.

Create and plot a directed graph.

```
s = [1 1 1 1 2 2 4 5 6 7 8];
t = [2 3 6 8 5 10 6 6 10 3 10];
g = digraph(s,t);
plot(g);
```



Use a breadth-first search on the graph to determine if it is bipartite, and if so, return the relevant partitions.

```
events = {'edgetonew', 'edgetodiscovered', 'edgetofinished'};
T = bfsearch(g, 1, events, 'Restart', true);
partitions = false(1, numnodes(g));
is_bipart = true;
is_edgetonew = T.Event == 'edgetonew';
ed = T.Edge;

for ii=1:size(T, 1)
    if is_edgetonew(ii)
        partitions(ed(ii, 2)) = ~partitions(ed(ii, 1));
    else
```

```
        if partitions(ed(ii, 1)) == partitions(ed(ii, 2))
            is_bipart = false;
            break;
        end
    end
end
```

```
is_bipart
```

```
is_bipart = logical
   1
```

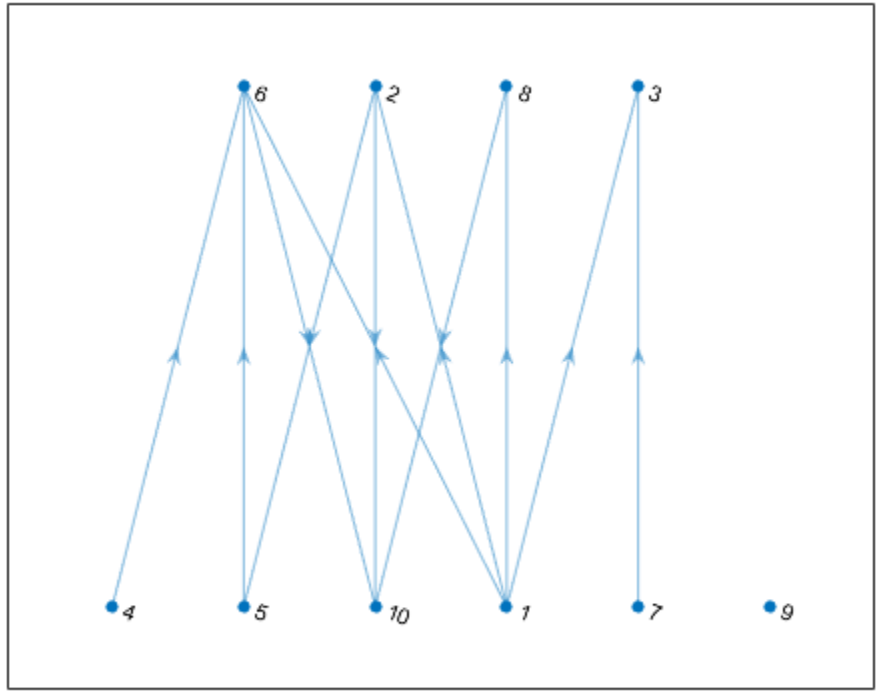
Since `g` is bipartite, the `partitions` variable contains the information about which partition each node belongs to.

Plot the bipartite graph with the `'layered'` layout, using the `partitions` variable to specify the source nodes that appear in the first layer.

```
partitions
```

```
partitions = 1x10 logical array

   0   1   1   0   0   1   0   1   0   0
```

```
plot(g, 'Layout', 'layered', 'Source', find(partitions));
```



Input Arguments

G — Input graph
graph object | digraph object

Input graph, specified as either a `graph` or `digraph` object. Use `graph` to create an undirected graph or `digraph` to create a directed graph.

Example: `G = graph(1,2)`

Example: `G = digraph([1 2],[2 3])`

s — Starting node
node index | node name

Starting node, specified as one of the values in this table.

Value	Example
Scalar node index	`1`
Character vector node name	`'A'`
String scalar node name	`"A"`

Example: `bfsearch(G,1)`

events — Flagged search events
`'discovernode'` (default) | `'startnode'` | `'finishnode'` | `'edgetonew'` | `'edgetodiscovered'` | `'edgetofinished'` | `'allevents'` | cell array | string array

Flagged search events, specified as one of the options in the following table.

- To flag single events, use the flag names.
- To flag a subset of events, put two or more flag names into a cell array or string array.
- To flag all events, use `'allevents'`.

Note Depending on the value of `events`, the output of `bfsearch` varies. See the last column in the following table for information about the output returned by each option.

Value of events	Description	Output
`'discovernode'` (default)	A new node has been discovered.	Return a vector of node IDs: • If `s` is a numeric node index, then the vector contains numeric node indices. • If `s` is a node name, then the vector is a cell array containing node names.
`'finishnode'`	All outgoing edges from the node have been visited.	

Value of events	Description	Output
`'startnode'`	This flag indicates the starting node in the search. If `'Restart'` is `true`, then `'startnode'` flags the starting node each time the search restarts.	
`'edgetonew'`	Edge connects to an undiscovered node.	Return a matrix or cell array of size `N`-by-`2` that specifies the end nodes of edges in the graph: • If `s` is a numeric node index, then the matrix contains numeric node indices. • If `s` is a node name, then the matrix is a cell array containing node names. Additionally, you can specify a second output with `[T,E] = bfsearch(…)` that returns a vector of edge indices `E`.
`'edgetodiscovered'`	Edge connects to a previously discovered node.	
`'edgetofinished'`	Edge connects to a finished node.	
cell array	Specify two or more flags in a cell array to only flag those events during the search.	Return a table, `T`, which contains the variables `T.Event`, `T.Node`, `T.Edge`, and `T.EdgeIndex`: • `T.Event` is a categorical vector containing the flags in order of their occurrence. • `T.Node` contains the node ID of the corresponding node for the events

Value of events	Description	Output
`'allevents'`	All events are flagged.	`'discovernode'`, `'finishnode'`, and `'startnode'`. • `T.Edge` contains the corresponding edge for the events `'edgetonew'`, `'edgetodiscovered'`, and `'edgetofinished'`. • `T.EdgeIndex` contains the edge index for the events `'edgetonew'`, `'edgetodiscovered'`, and `'edgetofinished'`. The edge index is for unique identification of repeated edges in a multigraph. • Unused elements of `T.Node` and `T.Edge` are set to `NaN`. • If `s` is a numeric node index, then `T.Node` and `T.Edge` contain numeric node indices. • If `s` is a node name, then `T.Node` and `T.Edge` are cell arrays containing node names.

Example: `v = bfsearch(G,3)` begins the search at the third node and returns a vector, `v`, containing the nodes in order of discovery. This is the same as `v = bfsearch(G,3,'discovernode')`.

Example: `X = bfsearch(G,'A','edgetonew')` begins at the node named `'A'` and returns a cell array, `X`, indicating each of the edges that connects to an undiscovered node during the search.

Example: `T = bfsearch(G,s,{'discovernode','finishnode'})` returns a table, `T`, but only flags when new nodes are discovered or when a node is marked finished.

Example: `T = bfsearch(G,s,'allevents')` flags all search events and returns a table, `T`.

Data Types: `char` | `string` | `cell`

`tf` — Toggle to restart search
`false` (default) | `true`

Toggle to restart search, specified as `false` (default) or `true`. This option is useful if the graph contains nodes that are unreachable from the starting node. If `'Restart'` is `true`, then the search restarts whenever undiscovered nodes remain that are unreachable from the discovered nodes. The new start node is the node with smallest index that is still undiscovered. The restarting process repeats until `bfsearch` discovers all nodes.

`'Restart'` is `false` by default, so that the search only visits nodes that are reachable from the starting node.

When `'Restart'` is `true`, the `'discovernode'` and `'finishnode'` events occur once for each node in the graph. Also, each edge in the graph is flagged once by `'edgetonew'`, `'edgetodiscovered'`, or `'edgetofinished'`. The edges flagged by `'edgetonew'` form one or more trees.

Example: `T = bfsearch(graph([1 3],[2 4]),1,'Restart',true)` searches both of the connected components in the graph.

Data Types: `logical`

Output Arguments

`v` — Node IDs
numeric column vector | cell vector

Node IDs, returned in one of the following formats:

- If you use a numeric node ID to specify the starting node `s`, then `v` is a numeric column vector of node indices.

- If `s` is a character vector or string containing a node name, then `v` is a cell vector containing node names.

The node IDs in `v` reflect the order of discovery by the breadth-first graph search.

T — Search results
table | vector | matrix | cell array of node names

Search results, returned in one of the following formats:

- If `events` is not specified or is `'discovernode'`, `'finishnode'`, or `'startnode'`, then `T` is a vector of node IDs similar to `v`.
- If `events` is `'edgetonew'`, `'edgetodiscovered'`, or `'edgetofinished'`, then `T` is a matrix or cell array of size `N`-by-`2` indicating the source and target nodes for each relevant edge.
- If `events` is a cell array of search events or `'allevents'`, then `T` is a table containing the flagged search events. The table contains the search event flags in `T.Event`, relevant node IDs in `T.Node`, and relevant edges in `T.Edge` and `T.EdgeIndex`.

In all cases:

- The order of the elements or rows of `T` indicates their order of occurrence during the search.
- If you specify `s` as a numeric node ID, then `T` also refers to nodes using their numeric IDs.
- If you specify `s` as a node name, then `T` also refers to nodes using their names.

E — Edge indices
vector

Edge indices, returned as a vector.

Specify this output to get a vector of edge indices for the events `'edgetonew'`, `'edgetodiscovered'`, or `'edgetofinished'`. The `N`-by-`1` vector of edge indices corresponds with `T`, which is a matrix or cell array of size `N`-by-`2` indicating the source and target nodes for each relevant edge.

Example: `[T,E] = bfsearch(G,s,'edgetonew')`

Tips

- `dfsearch` and `bfsearch` treat undirected graphs the same as directed graphs. An undirected edge between nodes `s` and `t` is treated like two directed edges, one from `s` to `t` and one from `t` to `s`.

Algorithms

The Breadth-First search algorithm begins at the starting node, `s`, and inspects all of its neighboring nodes in order of their node index. Then for each of those neighbors, it visits their unvisited neighbors in order. The algorithm continues until all nodes that are reachable from the starting node have been visited.

In pseudo-code, the algorithm can be written as:

```
Event startnode(S)
Event discovernode(S)
NodeList = {S}

WHILE NodeList is not empty

  C = NodeList{1}
  Remove first element from NodeList

  FOR edge E from outgoing edges of node C, connecting to node N
    Event edgetonew(C,E), edgetodiscovered(C,E) or edgetofinished(C,E)
    (depending on the state of node N)
    IF event was edgetonew
      Event discovernode(N)
      Append N to the end of NodeList
    END
  END

  Event finishnode(C)
END
```

`bfsearch` can return flags to describe the different events in the algorithm, such as when a new node is discovered or when all of the outgoing edges of a node have been visited. The event flags are listed here.

Event Flag	Event Description
`'discovernode'`	A new node has been discovered.
`'finishnode'`	All outgoing edges from the node have been visited.
`'startnode'`	This flag indicates the starting node in the search.
`'edgetonew'`	Edge connects to an undiscovered node
`'edgetodiscovered'`	Edge connects to a previously discovered node
`'edgetofinished'`	Edge connects to a finished node

For more information, see the input argument description for `events`.

Note In cases where the input graph contains nodes that are unreachable from the starting node, the `'Restart'` option provides a way to make the search visit every node in the graph. In that case, the `'startnode'` event indicates the starting node each time the search restarts.

See Also

`conncomp` | `dfsearch` | `digraph` | `graph`

Topics

"Visualize Breadth-First and Depth-First Search"

Introduced in R2015b

biconncomp

Biconnected graph components

Syntax

```
bins = biconncomp(G)
bins = biconncomp(G,'OutputForm',form)
[bins,iC] = biconncomp( ___ )
```

Description

`bins = biconncomp(G)` returns the biconnected components on page 1-5191 of graph `G` as bins. The bin numbers indicate which biconnected component each edge in the graph belongs to. Each edge in `G` belongs to a single biconnected component, whereas the nodes in `G` can belong to more than one biconnected component. Two nodes belong to the same biconnected component if removing any one node from the graph does not disconnect them.

`bins = biconncomp(G,'OutputForm',form)`, where `form` is `'cell'`, returns the output as a cell array such that `bins{j}` contains the node IDs of all nodes in component `j`. The default for `form` is `'vector'`.

`[bins,iC] = biconncomp( ___ )` additionally returns the node indices `iC` indicating which nodes are cut vertices on page 1-5194 (also called articulation points).

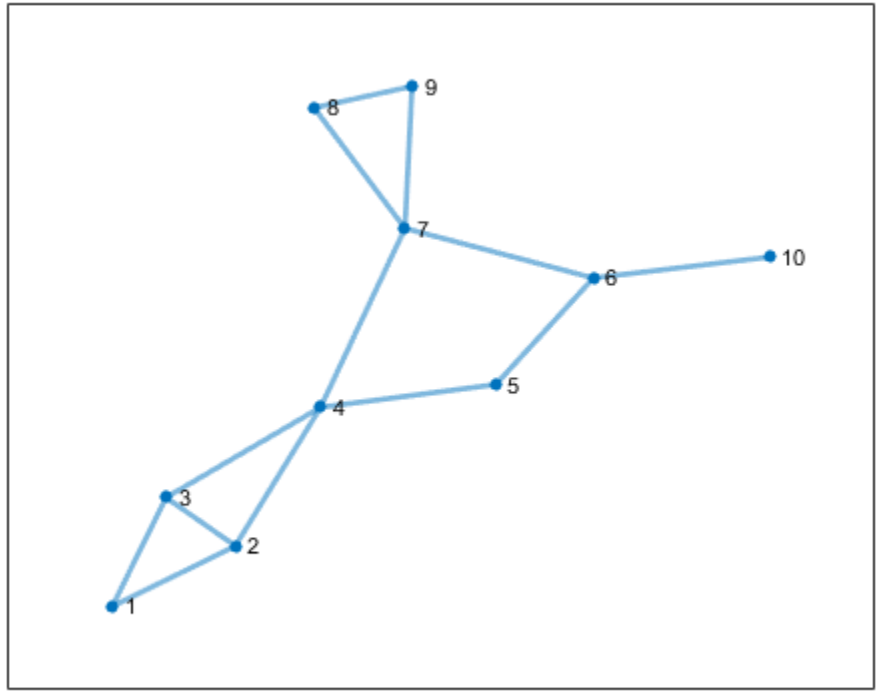
Examples

Highlight Biconnected Components

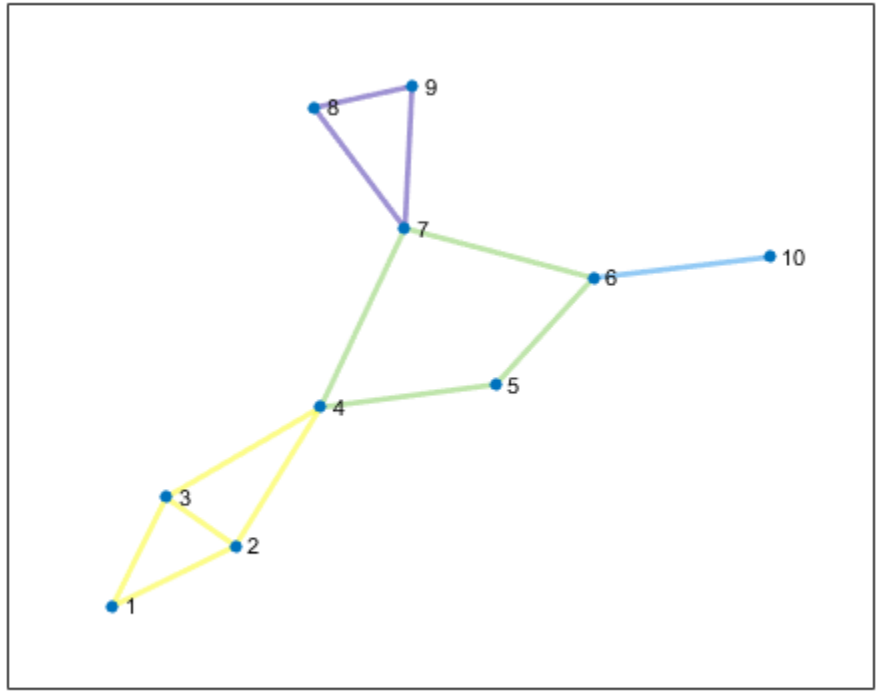
Create and plot a graph. Color the edges based on which biconnected component each edge belongs to.

```
s = [1 1 2 2 3 4 4 5 6 6 7 7 8];
t = [2 3 3 4 4 5 7 6 7 10 8 9 9];
```

```
G = graph(s,t);
p = plot(G,'LineWidth',2);
```




```
p.EdgeCData =  biconncomp(G);
```

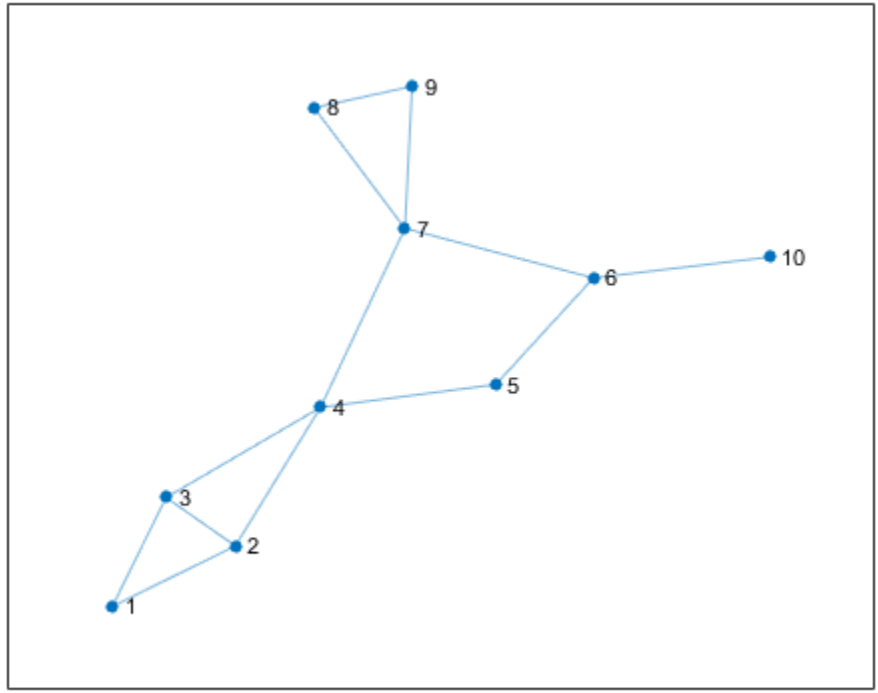


Extract Biconnected Components as Subgraphs

This example shows how to extract the biconnected components from a graph as subgraphs, and then label the nodes in each subgraph using the node indices in the original graph.

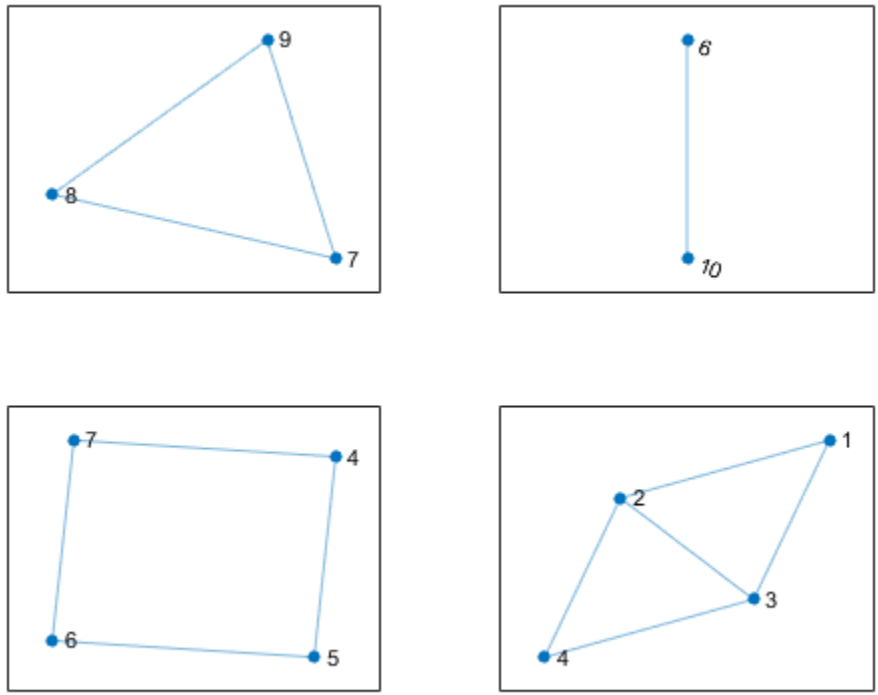
Create and plot a graph.

```
s = [1 1 2 2 3 4 4 5 6 6 7 7 8];
t = [2 3 3 4 4 5 7 6 7 10 8 9 9];
G = graph(s,t);
plot(G)
```



Group the graph nodes into bins based on which biconnected component(s) each node belongs to. Then, loop through each of the bins and extract a subgraph for each biconnected component. Label the nodes in each subgraph using their original node indices.

```
bincell = biconncomp(G, 'OutputForm', 'cell');
n = length(bincell);

for ii = 1:n
    subplot(2,2,ii)
    plot(subgraph(G, bincell{ii}), 'NodeLabel', bincell{ii});
end
```

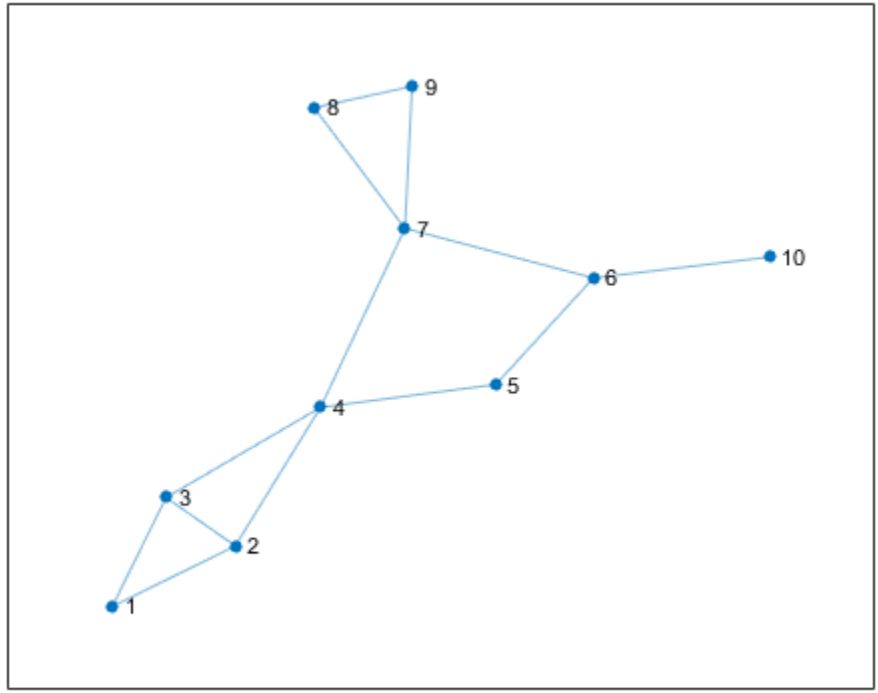


Identify Cut Vertices

Identify the cut vertices in a graph and then highlight those vertices in the graph plot.

Create and plot a graph. Calculate which biconnected component each graph edge belongs to, and specify a second output to return a vector identifying the cut vertices.

```
s = [1 1 2 2 3 4 4 5 6 6 7 7 8];
t = [2 3 3 4 4 5 7 6 7 10 8 9 9];
G = graph(s,t);
p = plot(G);
```

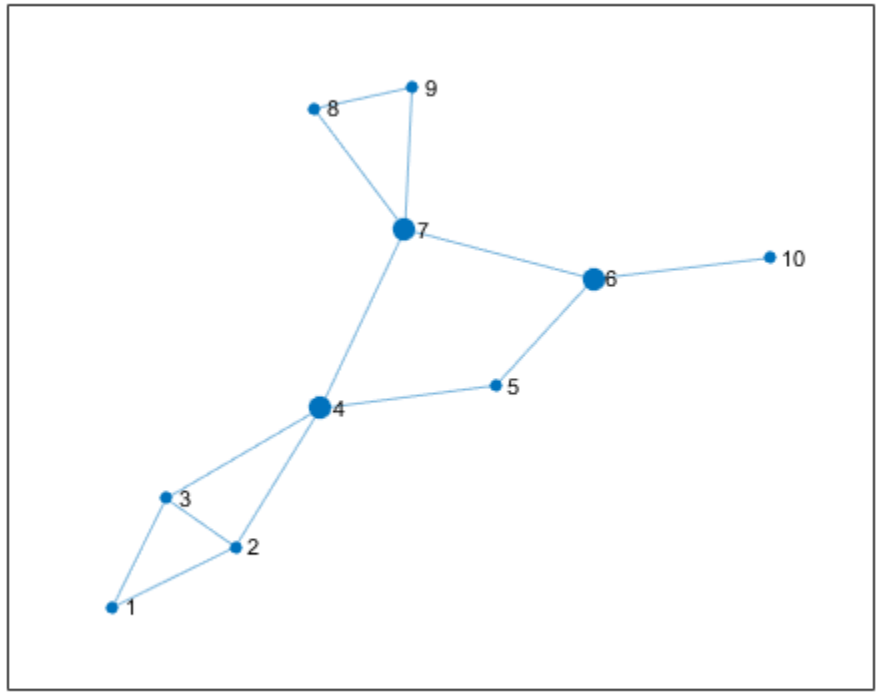



```
[edgebins,iC] = biconncomp(G)
```

```
edgebins = 1×13

     4     4     4     4     4     3     3     3     3     2     1     1     1


iC = 1×3

     4     6     7
```

Nodes 4, 6, and 7 are the cut vertices of graph `G`. Use `highlight` to enlarge the cut vertices referenced in `iC`.

```
highlight(p, iC)
```



Input Arguments

G — Input graph
graph object

Input graph, specified as a graph object. Use graph to create an undirected graph object.

Example: G = graph(1,2)

`form` — Type of output
`'vector'` (default) | `'cell'`

Type of output, specified as one of these values:

Option	Output
`'vector'` (default)	`bins` is a numeric vector indicating which biconnected component each edge belongs to.
`'cell'`	`bins` is a cell array, and `bins{j}` contains the node IDs for all nodes that belong to component `j`.

Output Arguments

`bins` — Biconnected components
vector | cell array

Biconnected components, returned as a vector or cell array. The bin numbers assign each edge or node in the graph to a biconnected component:

- If `OutputForm` is `'vector'` (default), then `bins` is a numeric vector indicating which connected component (bin) each edge belongs to.
- If `OutputForm` is `'cell'`, then `bins` is a cell array, with `bins{j}` containing the node IDs for all nodes that belong to component `j`.

`iC` — Indices of cut vertices
vector

Indices of cut vertices, returned as a vector of numeric node IDs.

More About

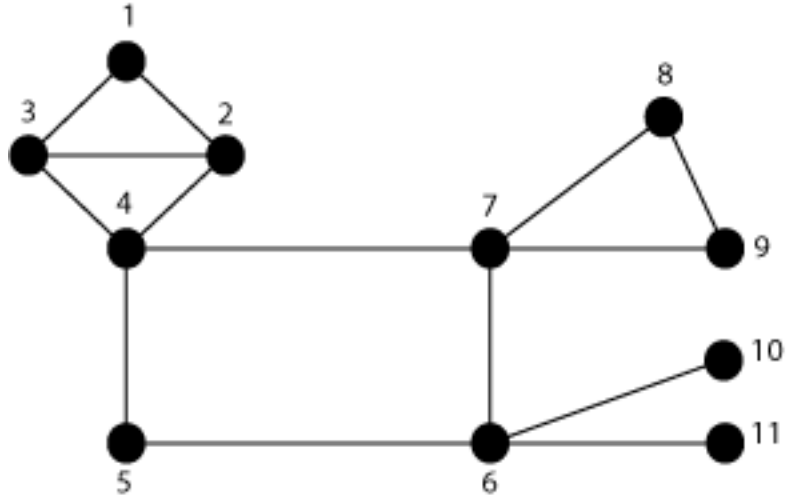
Biconnected Components

A biconnected component of a graph is a maximally biconnected subgraph. A graph is biconnected if it does not contain any cut vertices.

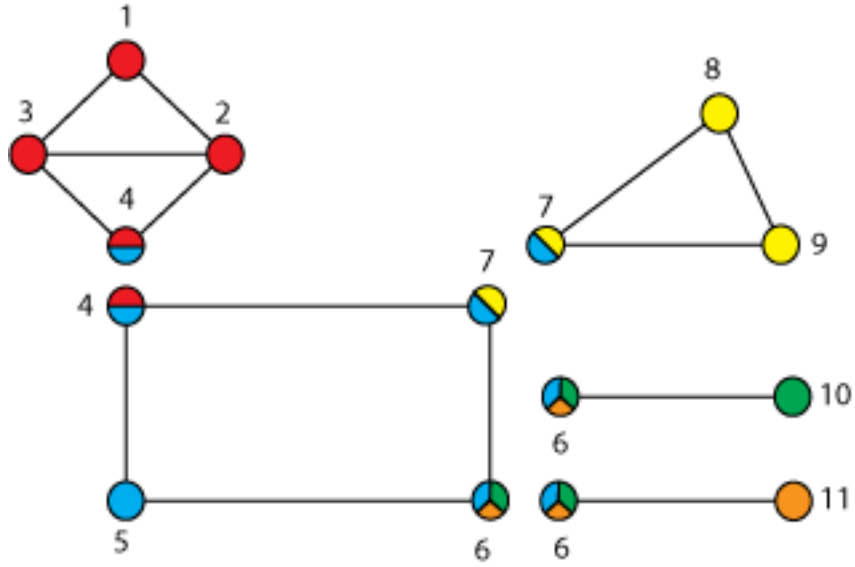
Decomposing a graph into its biconnected components helps to measure how well-connected the graph is. You can decompose any connected graph into a tree of biconnected components, called the *block-cut tree*. The blocks in the tree are attached at shared vertices, which are the cut vertices.

The illustration depicts:

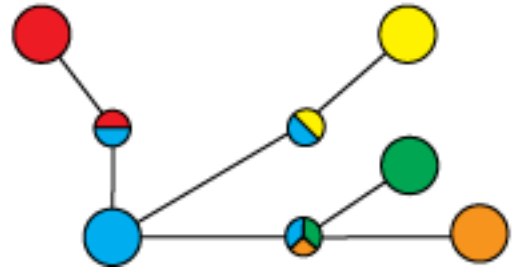
- (a) An undirected graph with 11 nodes.
- (b) Five biconnected components of the graph, with the cut vertices of the original graph colored for each component to which they belong.
- (c) Block-cut tree of the graph, which contains a node for each biconnected component (as large circles) and a node for each cut vertex (as smaller multicolored circles). In the block-cut tree, an edge connects each cut vertex to each component to which it belongs.



(a)



(b)



(c)

Cut Vertices

Also known as *articulation points*, cut vertices are graph nodes whose removal increases the number of connected components. In the previous illustration, the cut vertices are those nodes with more than one color: nodes 4, 6, and 7.

See Also

`bctree` | `condensation` | `conncomp`

Topics

"Directed and Undirected Graphs"
"Visualize Breadth-First and Depth-First Search"

Introduced in R2016b

centrality

Measure node importance

Syntax

```
C = centrality(G,type)
C = centrality( ___ ,Name,Value)
```

Description

`C = centrality(G,type)` computes the node centrality specified by `type` for each node in the graph.

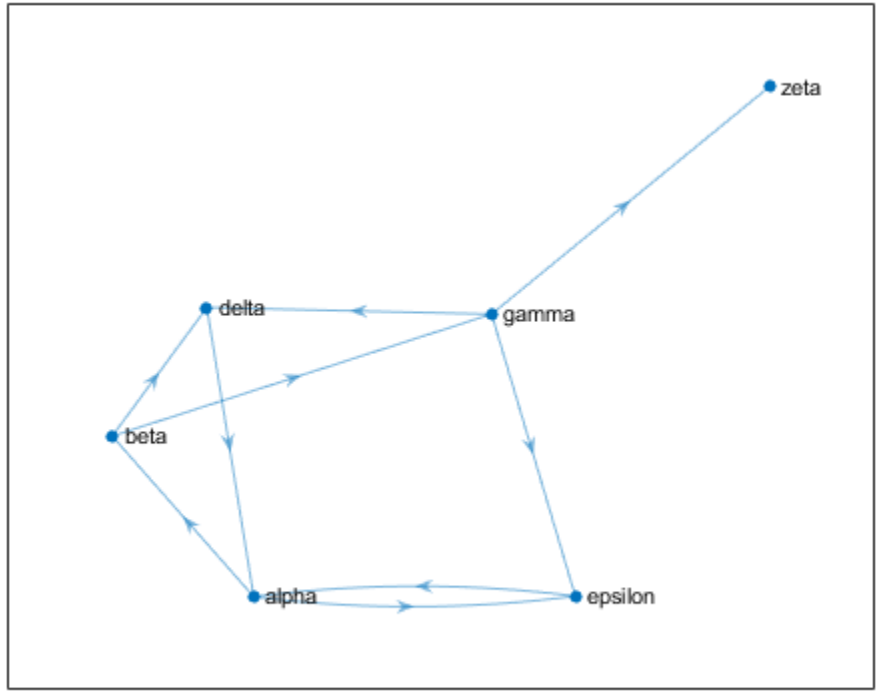
`C = centrality( ___ ,Name,Value)` uses additional options specified by one or more Name-Value pair arguments. For example, `centrality(G,'closeness','Cost',c)` specifies the cost of traversing each edge.

Examples

Page Rank of 6 Websites

Create and plot a graph containing six fictitious websites.

```
s = [1 1 2 2 3 3 3 4 5];
t = [2 5 3 4 4 5 6 1 1];
names = {'http://www.example.com/alpha', 'http://www.example.com/beta', ...
         'http://www.example.com/gamma', 'http://www.example.com/delta', ...
         'http://www.example.com/epsilon', 'http://www.example.com/zeta'};
G = digraph(s,t,[],names);
plot(G,'NodeLabel',{'alpha','beta','gamma','delta','epsilon','zeta'})
```



Calculate the page rank of each website using the `centrality` function. Append this information to the `Nodes` table of the graph as an attribute of the graph nodes.

```
pg_ranks = centrality(G,'pagerank')
```

```
pg_ranks = 6×1

    0.3210
    0.1706
    0.1066
    0.1368
    0.2008
    0.0643
```

```
G.Nodes.PageRank = pg_ranks;
G.Nodes
```

```
ans=6×2 table
                    Name                    PageRank
    ____________________________________    ________

    {'http://www.example.com/alpha'  }    0.32098
    {'http://www.example.com/beta'   }    0.17057
    {'http://www.example.com/gamma'  }    0.10657
    {'http://www.example.com/delta'  }    0.13678
    {'http://www.example.com/epsilon'}    0.20078
    {'http://www.example.com/zeta'   }    0.06432
```

Also determine which nodes are hubs and authorities using `centrality` and append the scores to the `Nodes` table.

```
hub_ranks = centrality(G,'hubs');
auth_ranks = centrality(G,'authorities');
G.Nodes.Hubs = hub_ranks;
G.Nodes.Authorities = auth_ranks;
```

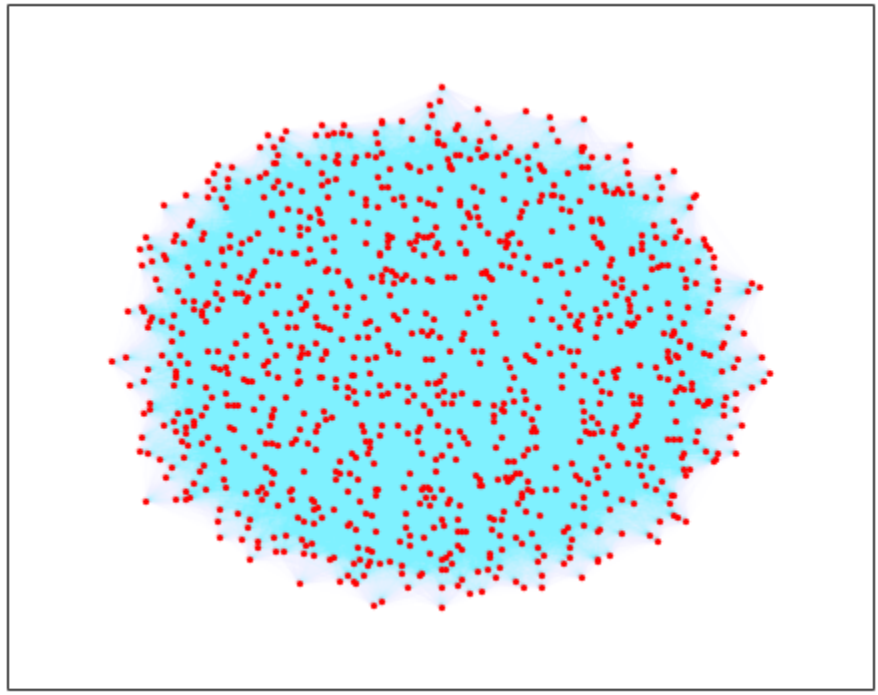
```
G.Nodes
```

```
ans=6×4 table
                    Name                    PageRank       Hubs       Authorities
    ____________________________________    ________    __________    ___________

    {'http://www.example.com/alpha'  }    0.32098        0.24995    7.3237e-05
    {'http://www.example.com/beta'   }    0.17057        0.24995      0.099993
    {'http://www.example.com/gamma'  }    0.10657        0.49991      0.099993
    {'http://www.example.com/delta'  }    0.13678     9.1536e-05       0.29998
    {'http://www.example.com/epsilon'}    0.20078     9.1536e-05       0.29998
    {'http://www.example.com/zeta'   }    0.06432              0       0.19999
```

Degree Centrality of Random Graph

Create and plot a weighted graph using a random sparse adjacency matrix. Since there are a lot of edges, use a very small value for `EdgeAlpha` to make the edges mostly transparent.

```
A = sprand(1000,1000,0.15);
A = A + A';
G = graph(A,'omitselfloops');
p = plot(G,'Layout','force','EdgeAlpha',0.005,'NodeColor','r');
```



Calculate the degree centrality of each node. Specify the importance of each edge using the edge weights.

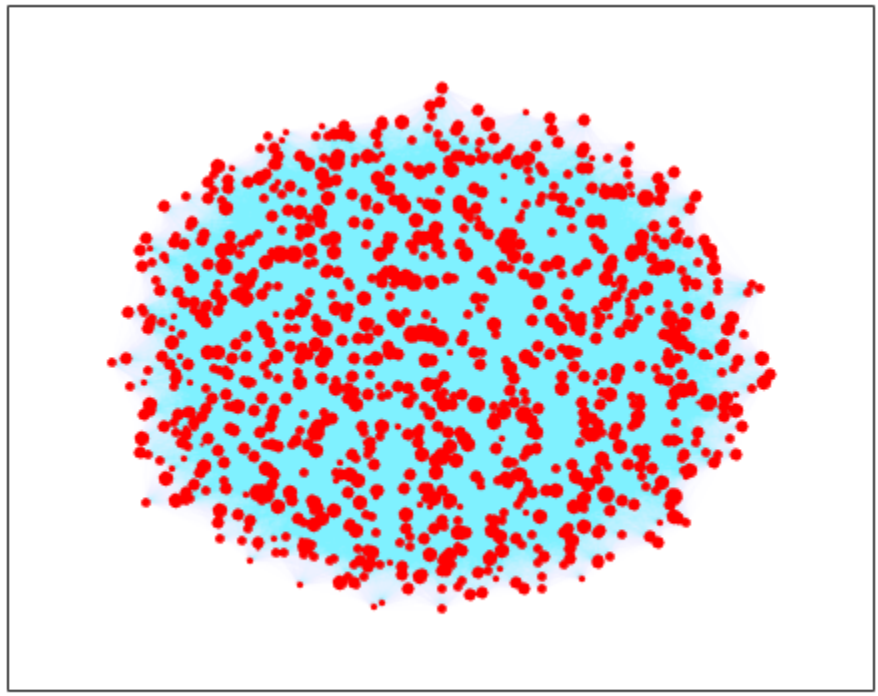
```
deg_ranks = centrality(G,'degree','Importance',G.Edges.Weight);
```

Use `discretize` to place the nodes into 7 equally-spaced bins based on their centrality scores.

```
edges = linspace(min(deg_ranks),max(deg_ranks),7);
bins = discretize(deg_ranks,edges);
```

Make the size of each node in the plot proportional to its centrality score. The marker size of each node is equal to the bin number (1-7).

```
p.MarkerSize = bins;
```



Closeness and Betweenness of Minnesota Roads

Load the data in `minnesota.mat`, which contains a graph object `G` representing the network of roads in Minnesota. The graph nodes have *xy* coordinates contained in the `XCoord` and `YCoord` variables of the `G.Nodes` table.

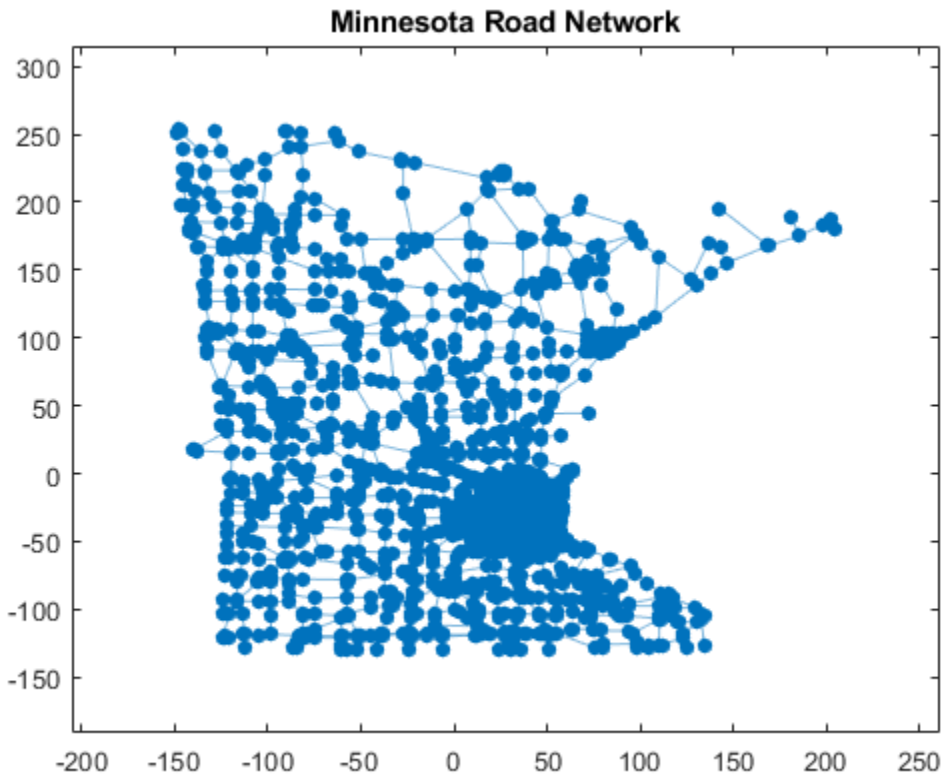
```
load minnesota.mat
xy = [G.Nodes.XCoord G.Nodes.YCoord];
```

Add edge weights to the graph that roughly correspond to the length of the roads, calculated using the Euclidean distance between the *xy* coordinates of the end nodes of each edge.

```
[s,t] = findedge(G);
G.Edges.Weight = hypot(xy(s,1)-xy(t,1), xy(s,2)-xy(t,2));
```

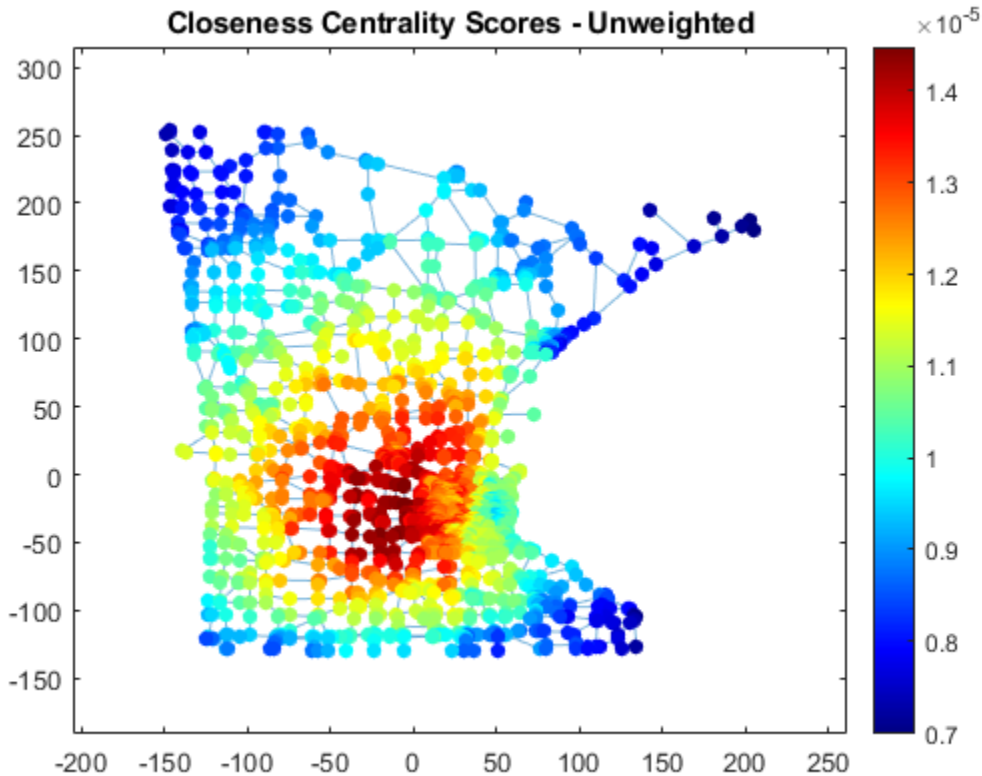
Plot the graph using the *xy* coordinates for the nodes.

```
p = plot(G,'XData',xy(:,1),'YData',xy(:,2),'MarkerSize',5);
title('Minnesota Road Network')
```



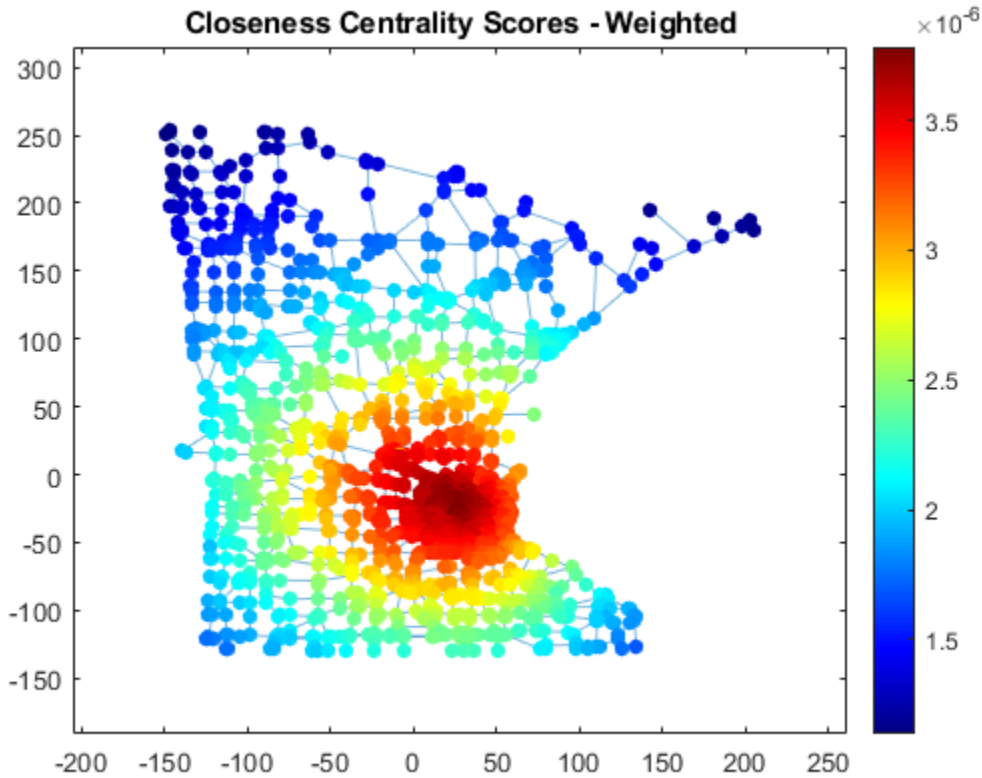
Compute the closeness centrality of each node. Scale the node color `NodeCData` to be proportional to the centrality score.

```
ucc = centrality(G,'closeness');
p.NodeCData = ucc;
colormap jet
colorbar
title('Closeness Centrality Scores - Unweighted')
```



Also compute the weighted closeness centrality score, using the edge weights as the cost of traversing each edge. Using the road lengths as edge weights improves the score quality, since distances are now measured as the sum of the lengths of all traveled edges, rather than the number of edges traveled.

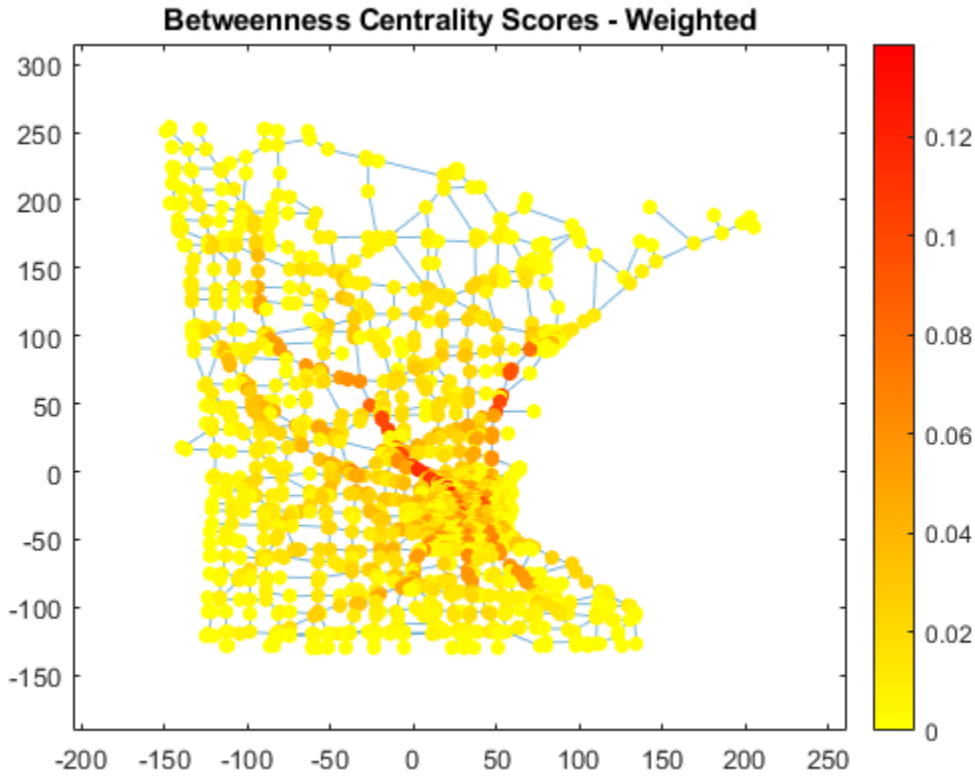
```
wcc = centrality(G,'closeness','Cost',G.Edges.Weight);
p.NodeCData = wcc;
title('Closeness Centrality Scores - Weighted')
```



Compute the weighted betweenness centrality scores for the graph to determine the roads most often found on the shortest path between two nodes. Normalize the centrality scores with the factor $\frac{(n-2)(n-1)}{2}$ so that the score represents the probability that a traveler along a shortest path between two random nodes will travel through a given node. The plot indicates that there are a few very important roads leading into and out of the city.

```
wbc = centrality(G,'betweenness','Cost',G.Edges.Weight);
n = numnodes(G);
```

```
p.NodeCData = 2*wbc./((n-2)*(n-1));
colormap(flip(autumn,1));
title('Betweenness Centrality Scores - Weighted')
```



Input Arguments

G — Input graph
graph object | digraph object

Input graph, specified as either a graph or digraph object. Use graph to create an undirected graph or digraph to create a directed graph.

Example: `G = graph(1,2)`

Example: `G = digraph([1 2],[2 3])`

type — Type of node centrality
`'degree'` | `'outdegree'` | `'indegree'` | `'closeness'` | `'incloseness'` | `'outcloseness'` | `'betweenness'` | `'pagerank'` | `'eigenvector'` | `'hubs'` | `'authorities'`

Type of node centrality, specified as one of the options in the table. The table also lists the compatible Name-Value pairs that work with each type. Each variety of node centrality offers a different measure of node importance in a graph.

Option	Graph type	Description	Name-Value Pairs
`'degree'`	Undirected	The `'degree'`, `'outdegree'`, and `'indegree'` centrality types are based on the number of edges connecting to each node: • `'degree'` — Number of edges connecting to each node. A self-loop counts as two edges connecting to the node. • `'indegree'` — Number of incoming edges to each node. A self-loop counts as one incoming edge. • `'outdegree'` — Number of outgoing edges	`'Importance'`

Option	Graph type	Description	Name-Value Pairs

`'indegree'`
 	
`'outdegree'`
	Directed	from each node. A self-loop counts as one outgoing edge.
		
		If you specify `'Importance'` edge weights, then the algorithm uses the sum of the edge weights rather than the number of connecting edges.	

`'closeness'`	Undirected	The `'closeness'`, `'incloseness'`, and `'outcloseness'` centrality types use the inverse sum of the distance from a node to all other nodes in the graph. If not all nodes are reachable, then the centrality of node `i` is: $$c(i) = \left(\frac{A_i}{N-1}\right)^2 \frac{1}{C_i}.$$ A_i is the number of reachable nodes from node `i` (not counting `i`), N is the number of nodes in `G`, and C_i is the sum	`'Cost'`

Option	Graph type	Description	Name-Value Pairs

`'incloseness'`
 	
`'outcloseness'`
	Directed	of distances from node `i` to all reachable nodes.
		• If no nodes are reachable from node `i`, then `c(i)` is zero.
		• For `'incloseness'`, the distance measure is from all nodes to node `i`.
		• `'Cost'` edge weights specify the length of the edges.	

Option	Graph type	Description	Name-Value Pairs
`'betweenness'`	Undirected or Directed	The `'betweenness'` centrality type measures how often each graph node appears on a shortest path between two nodes in the graph. Since there can be several shortest paths between two graph nodes `s` and `t`, the centrality of node `u` is: $$c(u) = \sum_{s,t \neq u} \frac{n_{st}(u)}{N_{st}} .$$ $n_{st}(u)$ is the number of shortest paths from `s` to `t` that pass through node `u`, and N_{st} is the total number of shortest paths from `s` to `t`. • If the graph is undirected, then the paths from `s` to `t` and from `t` to `s` count only as one path (divide the formula by two). • `'Cost'` edge weights specify the length of the	`'Cost'`

Option	Graph type	Description	Name-Value Pairs
		edges and help determine the shortest paths between nodes `s` and `t`.	

Option	Graph type	Description	Name-Value Pairs

`'pagerank'`	Undirected or Directed	The `'pagerank'` centrality type results from a random walk of the network. At each node in the graph, the next node is chosen with probability `'FollowProbability'` from the set of successors of the current node (neighbors for the undirected case). Otherwise, or when a node has no successors, the next node is chosen from all nodes. The centrality score is the average time spent at each node during the random walk.
		• If a node has a self-loop, then there is a chance that the algorithm traverses it. Therefore self-loops increase the pagerank centrality score of the node they attach to.
			`'Importance'`
			`'FollowProbability'`
			`'Tolerance'`
			`'MaxIterations'`

Option	Graph type	Description	Name-Value Pairs
		• In multigraphs with multiple edges between the same two nodes, nodes with multiple edges are more likely to be chosen. • `'Importance'` edge weights affect how the algorithm chooses successors. Nodes with higher importance are more likely to be chosen.	

Option	Graph type	Description	Name-Value Pairs

`'eigenvector'`	Undirected	The `'eigenvector'` centrality type uses the eigenvector corresponding to the largest eigenvalue of the graph adjacency matrix. The scores are normalized such that the sum of all centrality scores is 1.
		• If there are several disconnected components, then the algorithm computes the eigenvector centrality individually for each component, then scales the scores according to the percentage of graph nodes in that component.
		• The centrality score of disconnected nodes is `1/numnodes(G)`.
		• Specify `'Importance'` edge weights to use a weighted adjacency matrix in the calculation.
			`'Importance'`
			`'Tolerance'`
			`'MaxIterations'`

Option	Graph type	Description	Name-Value Pairs

`'hubs'`
 	
`'authorities'`
	Directed	The `'hubs'` and `'authorities'` centrality scores are two linked centrality measures that are recursive. The hubs score of a node is the sum of the authorities scores of all its successors. Similarly, the authorities score is the sum of the hubs scores of all its predecessors. The sum of all hubs scores is 1 and the sum of all authorities scores is 1.
		
		• These scores can be interpreted as the left (hubs) and right (authorities) singular vectors corresponding to the largest singular value of the adjacency matrix.
		• The centrality score of disconnected nodes is `1/numnodes(G)`.
		• Specify `'Importance'`
			`'Importance'`
			
			`'Tolerance'`
			
			`'MaxIterations'`

Option	Graph type	Description	Name-Value Pairs
		edge weights to use a weighted sum, rather than the simple sum of all successor/ predecessor scores. This is equivalent to using the singular vectors of the weighted adjacency matrix. • If there are several disconnected components (in the weakly connected sense), then the algorithm computes the hubs and authorities scores individually for each component. The scores are then rescaled according to the percentage of graph nodes in that component so that the overall sum is still 1.	

Note The `centrality` function assumes all edge weights are equal to 1. To change this, specify edge weights for use with the `'Cost'` or `'Importance'` name-value pairs.

Example: `centrality(G,'degree')`

Example: `centrality(G,'hubs','Tolerance',tol)`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `C = centrality(G,'closeness','Cost',edgeCosts)` computes the closeness centrality using `edgeCosts` as the cost (weight) of traversing each edge in the graph.

`Cost` — Cost of edge traversal
vector

Cost of edge traversal, specified as the comma-separated pair consisting of `'Cost'` and a vector of positive edge weights. The ith edge weight specifies the cost associated with traversing the edge `findedge(G,i)`. All edge weights must be greater than zero.

`'Cost'` edge weights are smaller when the connection is shorter, or faster, or cheaper. Some examples of `'Cost'` edge weights are:

- Length of a path
- Travel time
- Cost of a ticket

Note `'Cost'` only applies to the `'closeness'`, `'outcloseness'`, `'incloseness'`, and `'betweenness'` centrality types.

Example: `centrality(G,'closeness','Cost',c)`

`FollowProbability` — Probability of selecting a successor node
`0.85` (default) | scalar between 0 and 1

Probability of selecting a successor node, specified as the comma-separated pair consisting of `'FollowProbability'` and a scalar between 0 and 1. The follow probability is the probability that the next node selected in the traversal by the pagerank

algorithm is chosen among the successors of the current node, and not at random from all nodes. For websites, this probability corresponds to clicking a link on the current web page instead of surfing to another random web page.

Note `'FollowProbability'` only applies to the `'pagerank'` centrality type.

Example: `centrality(G,'pagerank','FollowProbability',0.5)`

`Importance` — Edge importance
vector

Edge importance, specified as the comma-separated pair consisting of `'Importance'` and a vector of nonnegative edge weights. The ith edge weight specifies the importance of the edge `findedge(G,i)`. An edge weight of zero is equivalent to removing that edge from the graph.

For multigraphs with multiple edges between two nodes, `centrality` adds the multiple edges together and treats them as a single edge with the combined weight.

`'Importance'` edge weights are larger when the connection is stronger. Some examples of `'Importance'` edge weights are:

- Number of travellers per day
- Number of clicks on a link
- Number of papers published together

Note `'Importance'` only applies to the `'degree'`, `'outdegree'`, `'indegree'`, `'pagerank'`, `'eigenvector'`, `'hubs'`, and `'authorities'` centrality types.

Example: `centrality(G,'degree','Importance',x)`

`MaxIterations` — Maximum number of iterations
`100` (default) | scalar

Maximum number of iterations, specified as the comma-separated pair consisting of `'MaxIterations'` and a scalar. The `centrality` algorithm runs until the tolerance is met or the maximum number of iterations is reached, whichever comes first.

Note `'MaxIterations'` only applies to the `'pagerank'`, `'eigenvector'`, `'hubs'`, and `'authorities'` centrality types.

Example: `centrality(G,'pagerank','MaxIterations',250)`

`Tolerance` — Stopping criterion for iterative solvers
`1e-4` (default) | scalar

Stopping criterion for iterative solvers, specified as the comma-separated pair consisting of `'Tolerance'` and a scalar. The `centrality` algorithm runs until the tolerance is met or the maximum number of iterations is reached, whichever comes first.

Note `'Tolerance'` only applies to the `'pagerank'`, `'eigenvector'`, `'hubs'`, and `'authorities'` centrality types.

Example: `centrality(G,'pagerank','Tolerance',1e-5)`

Output Arguments

`C` — Node centrality scores
column vector

Node centrality scores, returned as a column vector. `C(i)` is the centrality score of node `i`. The interpretation of the node centrality score depends on the type of centrality computation selected. The more central a node is, the larger its centrality score.

See Also

`digraph` | `graph`

Introduced in R2016a

conncomp

Connected graph components

Syntax

```
bins = conncomp(G)
bins = conncomp(G,Name,Value)
[bins,binsizes] = conncomp( ___ )
```

Description

`bins = conncomp(G)` returns the connected components on page 1-5226 of graph `G` as bins. The bin numbers indicate which component each node in the graph belongs to.

- If `G` is an undirected graph, then two nodes belong to the same component if there is a path connecting them.
- If `G` is a directed graph, then two nodes belong to the same strong component only if there is a path connecting them in both directions.

`bins = conncomp(G,Name,Value)` uses additional options specified by one or more Name-Value pair arguments. For example, `conncomp(G,'OutputForm','cell')` returns a cell array to describe the connected components.

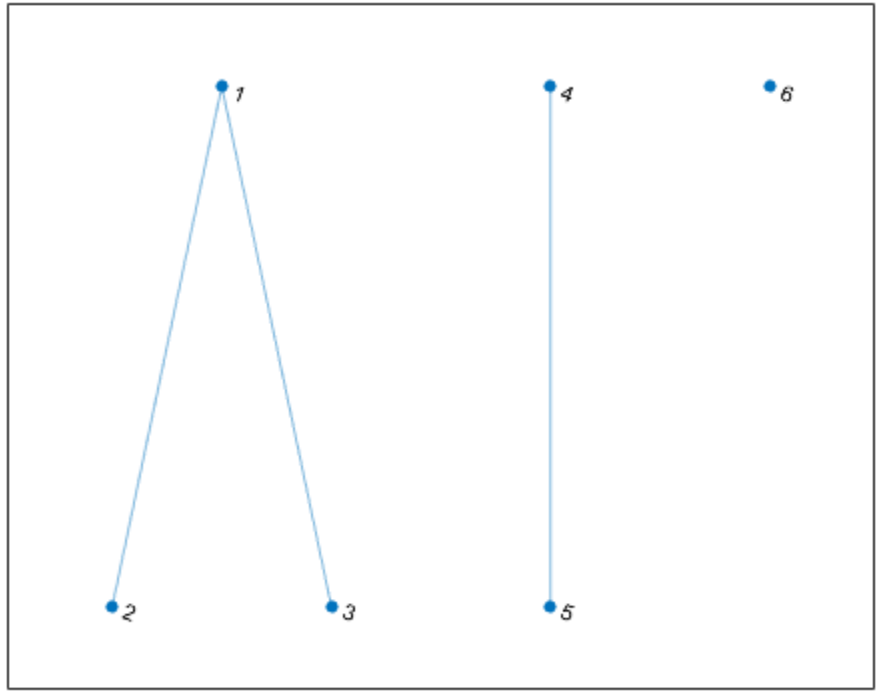
`[bins,binsizes] = conncomp( ___ )` also returns the size of the connected components. `binsizes(i)` gives the number of nodes in component `i`.

Examples

Find Graph Components

Create and plot an undirected graph with three connected components. Use `conncomp` to determine which component each node belongs to.

```
G = graph([1 1 4],[2 3 5],[1 1 1],6);
plot(G)
```



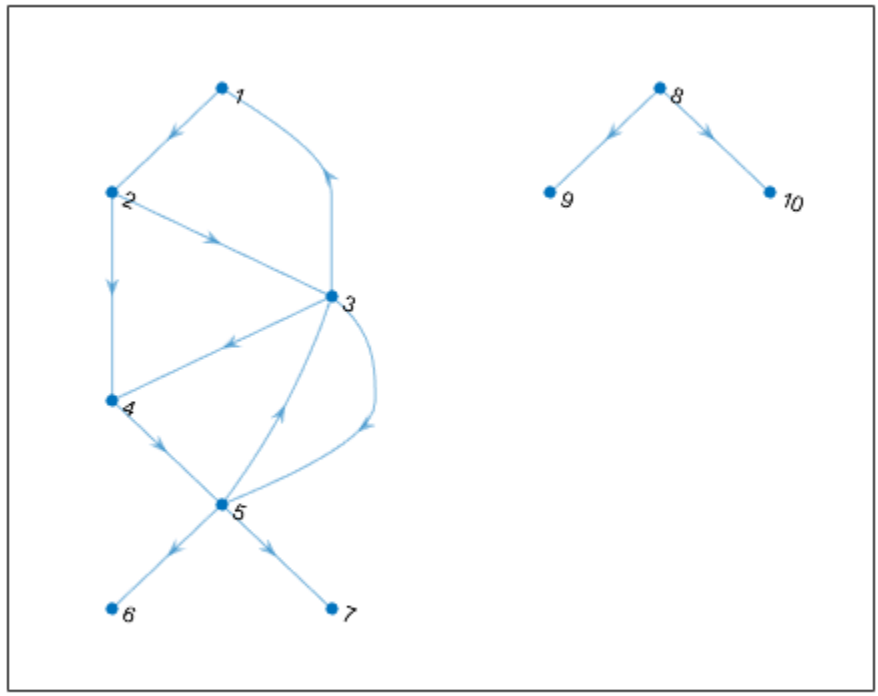

```
bins = conncomp(G)
```

```
bins = 1×6

     1     1     1     2     2     3
```

Strong and Weak Graph Components

Create and plot a directed graph, and then compute the strongly connected components and weakly connected components. Weakly connected components ignore the direction of connecting edges.

```
s = [1 2 2 3 3 3 4 5 5 5 8 8];
t = [2 3 4 1 4 5 5 3 6 7 9 10];
G = digraph(s,t);
plot(G,'Layout','layered')
```




```
str_bins = conncomp(G)
```

```
str_bins = 1×10
```

```
     4     4     4     4     4     6     5     1     3     2
```

```
weak_bins = conncomp(G,'Type','weak')
```

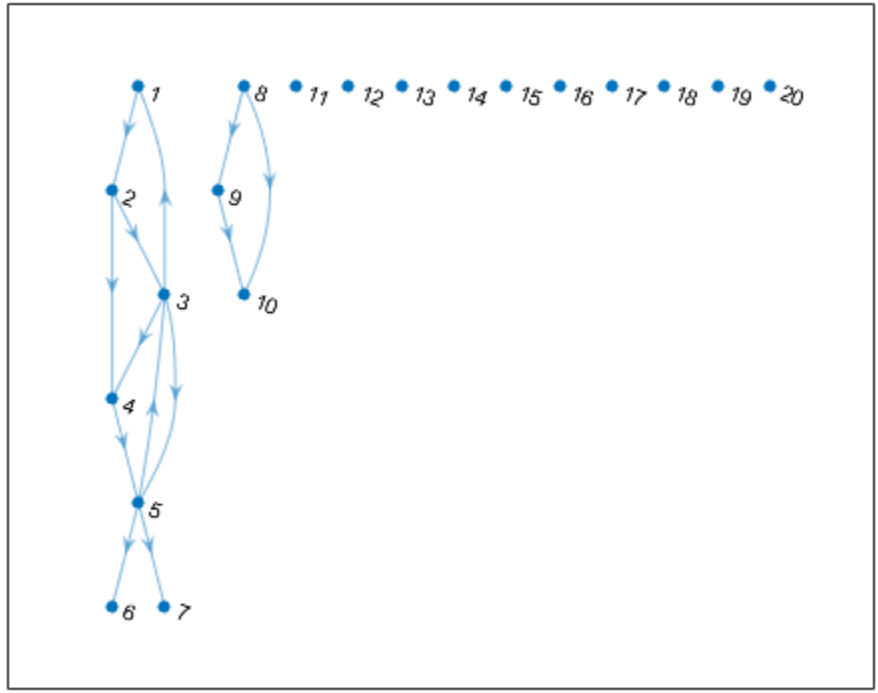
```
weak_bins = 1×10

     1     1     1     1     1     1     1     2     2     2
```

Discard Graph Components Based on Size

Use the second output of `conncomp` to extract the largest component of a graph or to remove components below a certain size.

Create and plot a directed graph. The graph has one large component, one small component, and several components that contain only a single node.

```
s = [1 2 2 3 3 3 4 5 5 5 8 8 9];
t = [2 3 4 1 4 5 5 3 6 7 9 10 10];
G = digraph(s,t,[],20);
plot(G,'Layout','layered')
```



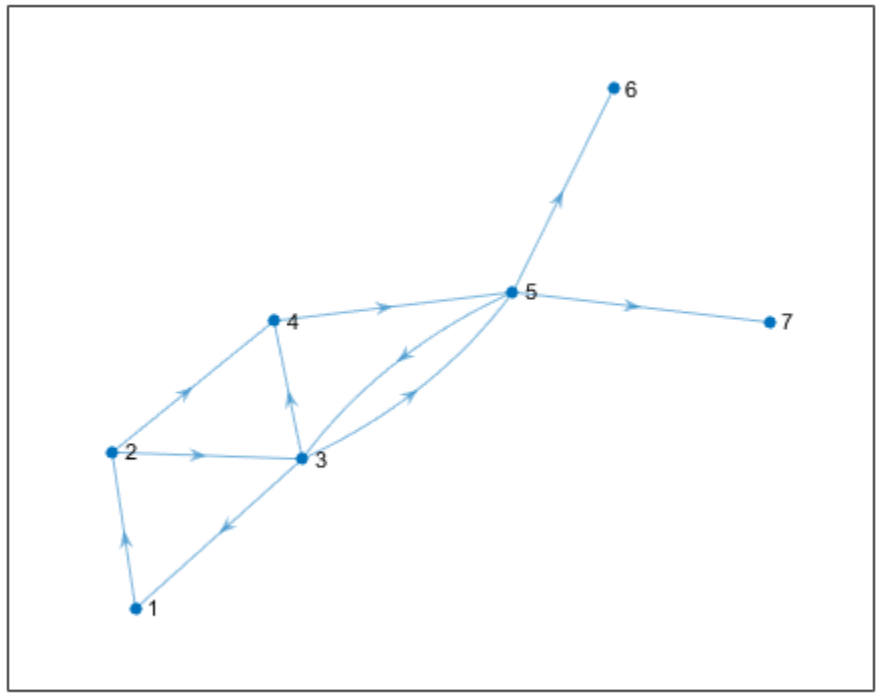
Calculate the weakly connected components and specify two outputs to `conncomp` to get the size of each component.

```
[bin,binsize] = conncomp(G,'Type','weak')
```

```
bin = 1×20

     1     1     1     1     1     1     1     2     2     2     3     4     5     6


binsize = 1×12

     7     3     1     1     1     1     1     1     1     1     1     1
```

Use `binsize` to extract the largest component from the graph. `idx` is a logical index indicating whether each node belongs to the largest component. The `subgraph` function extracts the nodes selected by `idx` from `G`.

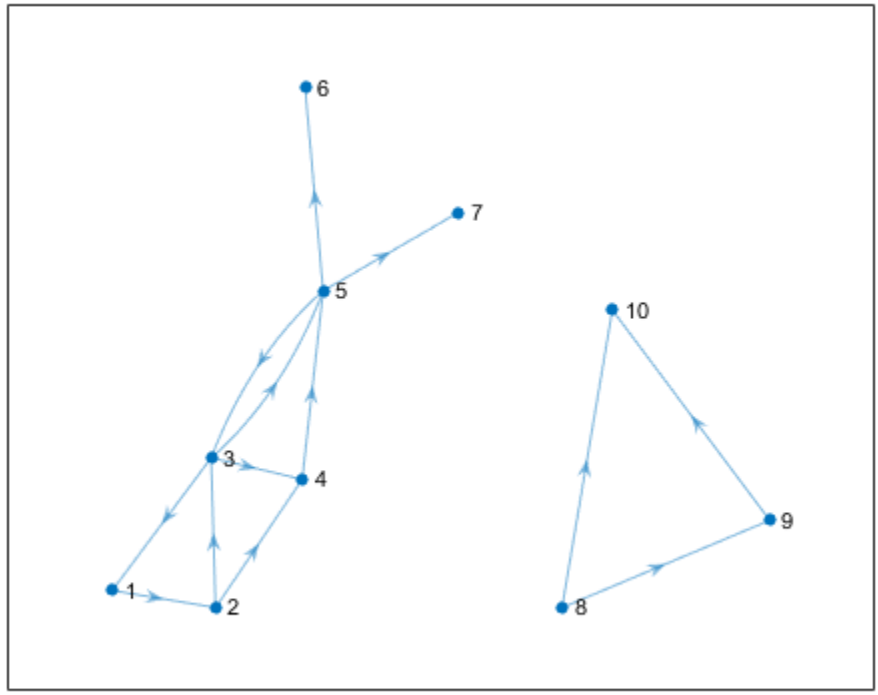
```
idx = binsize(bin) == max(binsize);
SG = subgraph(G, idx);
plot(SG)
```



A similar use of `binsizes` is to filter out components based on size. The procedure is similar to extracting the largest component, however in this case each node can belong to any component that meets the size requirement.

Filter out any components in `G` that have fewer than 3 nodes. `idx` is a logical index indicating whether each node belongs to a component with 3 or more nodes.

```
idx = binsize(bin) >= 3;
SG = subgraph(G, idx);
plot(SG)
```



Input Arguments

G — Input graph
graph object | digraph object

Input graph, specified as either a graph or digraph object. Use graph to create an undirected graph or digraph to create a directed graph.

Example: `G = graph(1,2)`

Example: `G = digraph([1 2],[2 3])`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `bins = conncomp(G,'OutputForm','cell')`

`OutputForm` — Type of output
`'vector'` (default) | `'cell'`

Type of output, specified as the comma-separated pair consisting of `'OutputForm'` and either `'vector'` or `'cell'`.

Option	Output
`'vector'` (default)	`bins` is a numeric vector indicating which connected component each node belongs to.
`'cell'`	`bins` is a cell array, and `bins{j}` contains the node IDs for all nodes that belong to component `j`.

Type — Type of connected components
`'strong'` (default) | `'weak'`

Note The `'Type'` option is supported only for directed graphs created using `digraph`.

Type of connected components, specified as the comma-separated pair consisting of `'Type'` and either `'strong'` (default) or `'weak'`.

Option	Result
`'strong'` (default)	Two nodes belong to the same connected component only if there is a path connecting them in *both* directions.
`'weak'`	Two nodes belong to the same connected component if there is a path connecting them, ignoring edge directions.

Example: `bins = conncomp(G,'Type','weak')` computes the weakly connected components of directed graph `G`.

Output Arguments

`bins` — Connected components
vector | cell array

Connected components, returned as a vector or cell array. The bin numbers assign each node in the graph to a connected component:

- If `OutputForm` is `'vector'` (default), then `bins` is a numeric vector indicating which connected component (bin) each node belongs to.
- If `OutputForm` is `'cell'`, then `bins` is a cell array, with `bins{j}` containing the node IDs for all nodes that belong to component `j`.

`binsizes` — Size of each connected component
vector

Size of each connected component, returned as a vector. `binsizes(i)` gives the number of elements in component `i`. The length of `binsizes` is equal to the number of connected components, `max(bins)`.

More About

Weakly Connected Components

Two nodes belong to the same weakly connected component if there is a path connecting them (ignoring edge direction). There are no edges between two weakly connected components.

The concepts of strong and weak components apply only to directed graphs, as they are equivalent for undirected graphs.

Strongly Connected Components

Two nodes belong to the same strongly connected component if there are paths connecting them in both directions. There can be edges between two strongly connected components, but these connecting edges are never part of a cycle.

The bin numbers of strongly connected components are such that any edge connecting two components points from the component of smaller bin number to the component with a larger bin number.

The concepts of strong and weak components apply only to directed graphs, as they are equivalent for undirected graphs.

See Also

`digraph` | `graph` | `subgraph`

Introduced in R2015b

degree

Degree of graph nodes

Syntax

```
D = degree(G)
D = degree(G,nodeIDs)
```

Description

`D = degree(G)` returns the degree of each node in graph `G`. The degree is the number of edges connected to each node.

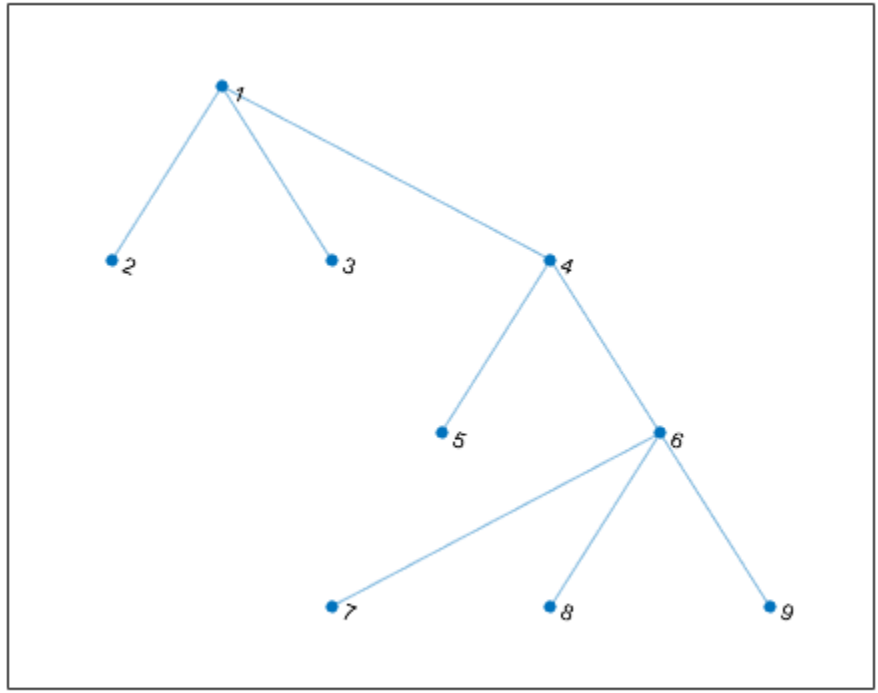
`D = degree(G,nodeIDs)` returns the degree of the nodes specified by `nodeIDs`.

Examples

Degree of All Graph Nodes

Create and plot a graph, and then use `degree` to find the degree of each node in the graph.

```
s = [1 1 1 4 4 6 6 6];
t = [2 3 4 5 6 7 8 9];
G = graph(s,t);
plot(G)
```




```
deg = degree(G)
```

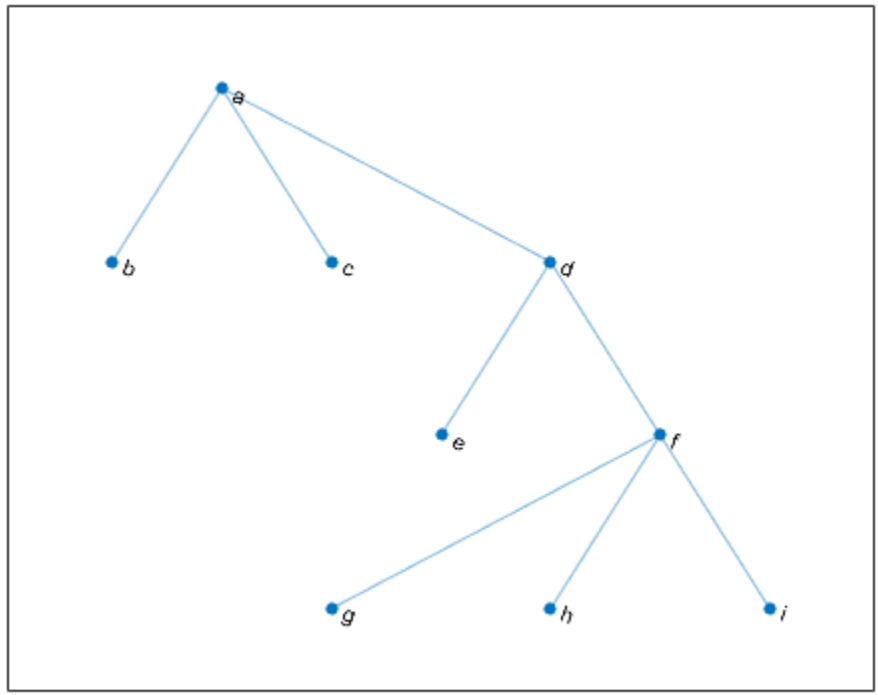
```
deg = 9×1

     3
     1
     1
     3
     1
     4
     1
     1
     1
```

`deg(j)` indicates the degree of node `j`.

Degree of Subset of Graph Nodes

Create and plot a graph, and then find the degree of the first, third, and fifth nodes.

```
s = {'a' 'a' 'a' 'd' 'd' 'f' 'f' 'f'};
t = {'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i'};
G = graph(s,t);
plot(G)
```



```
nodeIDs = {'a' 'c' 'e'}';
deg = degree(G,nodeIDs)
```

```
deg = 3×1

     3
     1
     1
```

`deg(j)` indicates the degree of node `nodeIDs(j)`.

Input Arguments

`G` — Input graph
`graph` object

Input graph, specified as a `graph` object. Use `graph` to create an undirected graph object.

Example: `G = graph(1,2)`

`nodeIDs` — Node identifiers
node indices | node names

Node identifiers, specified as one or more node indices or node names.

This table shows the different ways to refer to one or more nodes either by their numeric node indices or by their node names.

Form	Single Node	Multiple Nodes
Node index	Scalar **Example:** `1`	Vector **Example:** `[1 2 3]`
Node name	Character vector **Example:** `'A'`	Cell array of character vectors **Example:** `{'A' 'B' 'C'}`

Form	Single Node	Multiple Nodes
	String scalar **Example:** `"A"`	String array **Example:** `["A" "B" "C"]`

Example: `D = degree(G,[3 4])`

Example: `D = degree(G,{'LAX','ALB'})`

Output Arguments

`D` — Degree of nodes
array

Degree of nodes, returned as a numeric array. `D` is a column vector unless you specify `nodeIDs`, in which case `D` has the same size as `nodeIDs`.

A node that is connected to itself by an edge (a self-loop) is listed as its own neighbor only once, but the self-loop adds 2 to the total degree of the node.

See Also

`graph` | `inedges` | `neighbors` | `outedges`

Introduced in R2015b

dfsearch

Depth-first graph search

Syntax

```
v = dfsearch(G,s)
T = dfsearch(G,s,events)
[T,E] = dfsearch(G,s,events)
[ ___ ] = dfsearch( ___ ,'Restart',tf)
```

Description

`v = dfsearch(G,s)` applies depth-first search on page 1-5248 to graph `G` starting at node `s`. The result is a vector of node IDs in order of their discovery.

`T = dfsearch(G,s,events)` customizes the output of the depth-first search by flagging one or more search events. For example, `T = dfsearch(G,s,'allevents')` returns a table containing all flagged events, and `X = dfsearch(G,s,'edgetonew')` returns a matrix or cell array of edges.

`[T,E] = dfsearch(G,s,events)` additionally returns a vector of edge indices `E` when `events` is set to `'edgetonew'`, `'edgetodiscovered'`, or `'edgetofinished'`. The edge indices are for unique identification of edges in a multigraph.

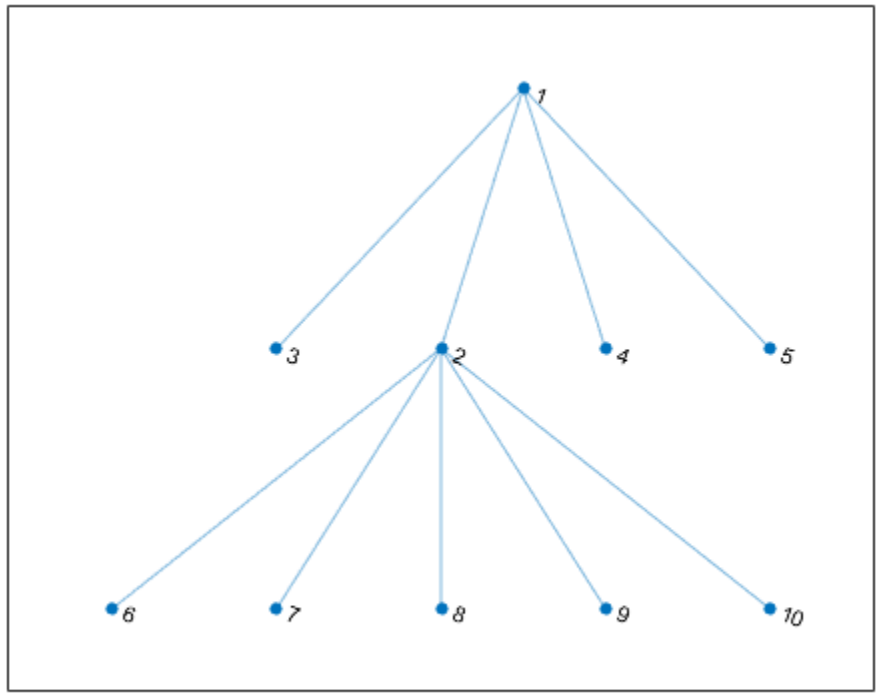
`[ ___ ] = dfsearch( ___ ,'Restart',tf)`, where `tf` is `true`, restarts the search if no new nodes are reachable from the discovered nodes. You can use any of the input or output argument combinations in previous syntaxes. This option ensures that the depth-first search reaches all nodes and edges in the graph, even if they are not reachable from the starting node, `s`.

Examples

Perform Depth-First Graph Search

Create and plot a graph.

```
s = [1 1 1 1 2 2 2 2 2];
t = [3 5 4 2 6 10 7 9 8];
G = graph(s,t);
plot(G)
```



Perform a depth-first search of the graph starting at node 7. The result indicates the order of node discovery.

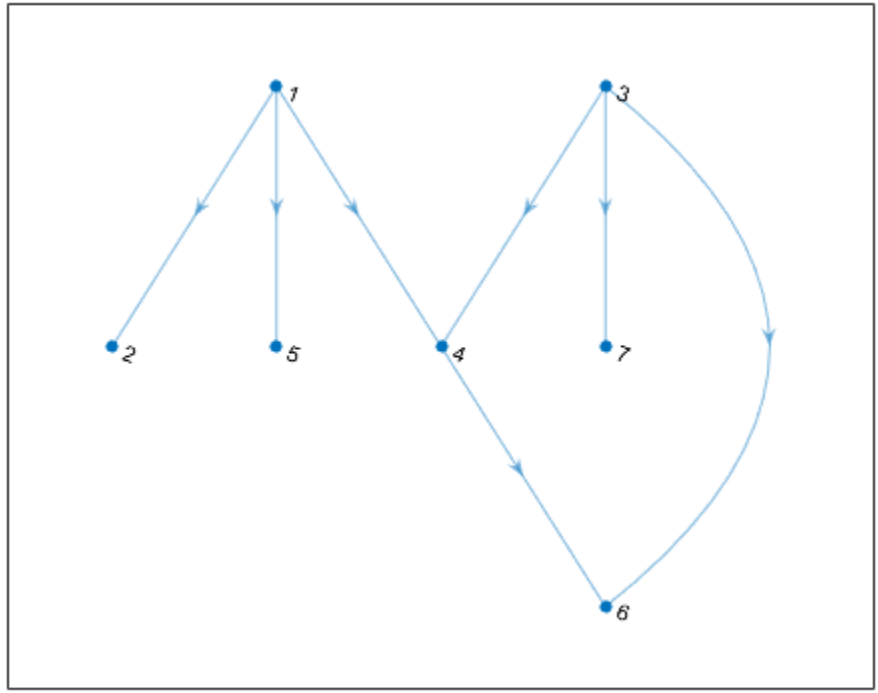
```
v = dfsearch(G,7)
```

```
v = 10×1

     7
     2
     1
     3
     4
     5
     6
     8
     9
    10
```

Depth-First Graph Search with All Events

Create and plot a directed graph.

```
A = [0 1 0 1 1 0 0;
     0 0 0 0 0 0 0;
     0 0 0 1 0 1 1;
     0 0 0 0 0 1 0;
     0 0 0 0 0 0 0;
     0 0 0 0 0 0 0;
     0 0 0 0 0 0 0];
G = digraph(A);
plot(G)
```



Perform a depth-first search on the graph starting at node 3. Specify `'allevents'` to return a table containing all of the events in the algorithm.

```
T = dfsearch(G,3,'allevents')
```

```
T=13×4 table
        Event          Node       Edge       EdgeIndex
    _______________    ____    __________    _________

    startnode            3     NaN    NaN       NaN
    discovernode         3     NaN    NaN       NaN
    edgetonew          NaN       3      4         4
    discovernode         4     NaN    NaN       NaN
    edgetonew          NaN       4      6         7
```

```
discovernode        6     NaN    NaN       NaN
finishnode          6     NaN    NaN       NaN
finishnode          4     NaN    NaN       NaN
edgetofinished    NaN       3      6         5
edgetonew         NaN       3      7         6
discovernode        7     NaN    NaN       NaN
finishnode          7     NaN    NaN       NaN
finishnode          3     NaN    NaN       NaN
```

To follow the steps in the algorithm, read the events in the table from top to bottom. For example:

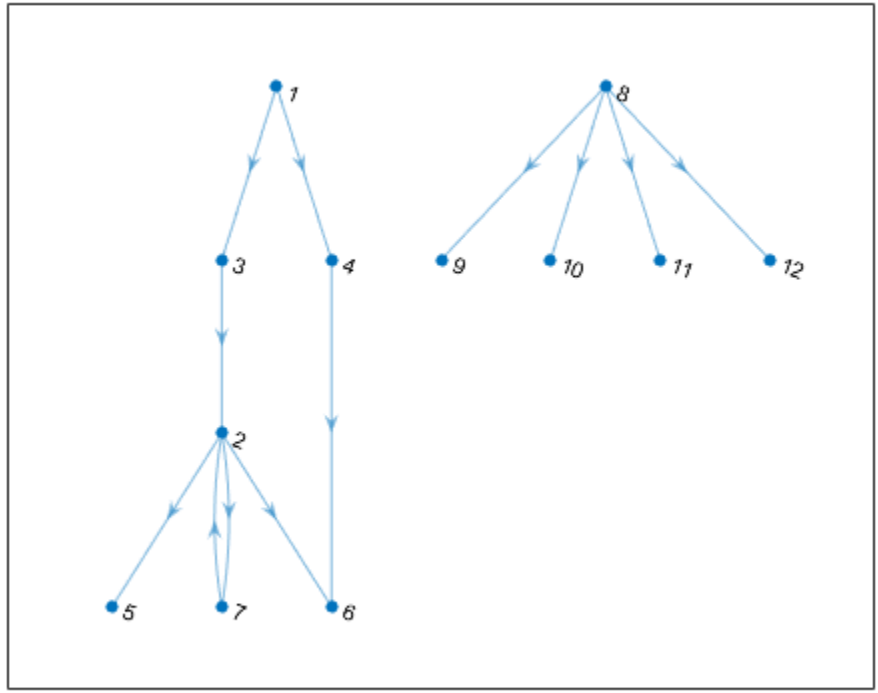
1. The algorithm begins at node 3
2. An edge is discovered between node 3 and node 4
3. Node 4 is discovered
4. and so on...

Depth-First Graph Search with Multiple Components

Perform a depth-first search of a graph with multiple components, and then highlight the graph nodes and edges based on the search results.

Create and plot a directed graph. This graph has two weakly connected components.

```
s = [1 1 2 2 2 3 4 7 8 8 8 8];
t = [3 4 7 5 6 2 6 2 9 10 11 12];
G = digraph(s,t);
p = plot(G,'Layout','layered');
```




```
c = conncomp(G,'Type','weak')
```

```
c = 1×12

     1     1     1     1     1     1     1     2     2     2     2     2
```

Perform a depth-first search of the graph starting at node 4, and flag the `'edgetonew'`, `'edgetodiscovered'`, `'edgetofinished'`, and `'startnode'` events. Specify `Restart` as `true` to make the search restart whenever there are remaining nodes that cannot be reached.

```
events = {'edgetonew','edgetodiscovered','edgetofinished','startnode'};
T = dfsearch(G,4,events,'Restart',true)
```

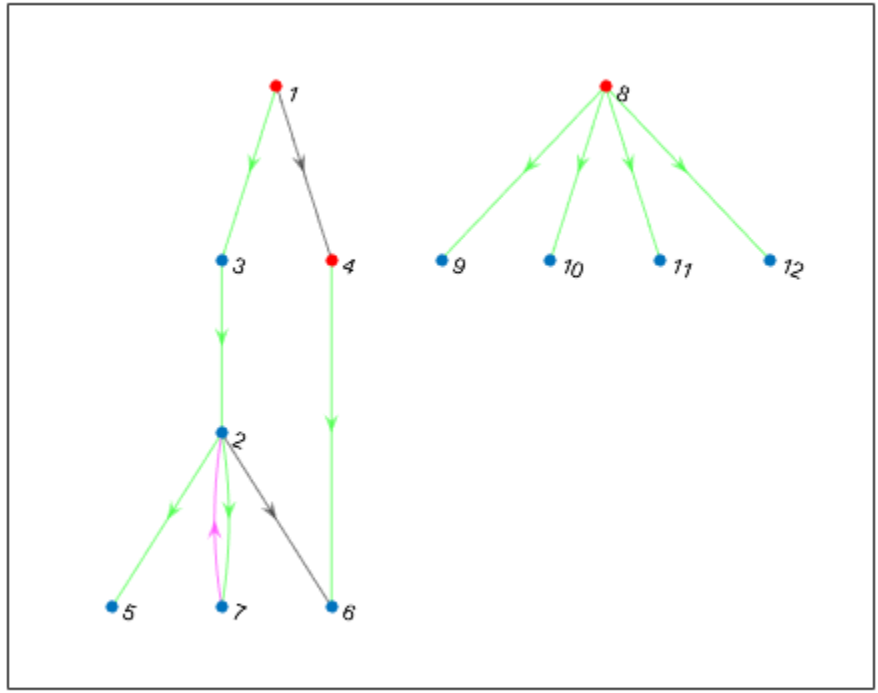
```
T=15×4 table
         Event           Node       Edge       EdgeIndex
    ________________     ____    __________    _________

    startnode              4     NaN    NaN        NaN
    edgetonew            NaN       4      6          7
    startnode              1     NaN    NaN        NaN
    edgetonew            NaN       1      3          1
    edgetonew            NaN       3      2          6
    edgetonew            NaN       2      5          3
    edgetofinished       NaN       2      6          4
    edgetonew            NaN       2      7          5
    edgetodiscovered     NaN       7      2          8
    edgetofinished       NaN       1      4          2
    startnode              8     NaN    NaN        NaN
    edgetonew            NaN       8      9          9
    edgetonew            NaN       8     10         10
    edgetonew            NaN       8     11         11
    edgetonew            NaN       8     12         12
```

When `Restart` is `true`, the `'startnode'` event returns information about where and when the algorithm restarts the search.

Highlight the graph based on event:

- Color the starting nodes red.
- Green edges are for `'edgetonew'`
- Black edges are for `'edgetofinished'`
- Magenta edges are for `'edgetodiscovered'`

```
highlight(p, 'Edges', T.EdgeIndex(T.Event == 'edgetonew'), 'EdgeColor', 'g')
highlight(p, 'Edges', T.EdgeIndex(T.Event == 'edgetofinished'), 'EdgeColor', 'k')
highlight(p, 'Edges', T.EdgeIndex(T.Event == 'edgetodiscovered'), 'EdgeColor', 'm')
highlight(p,T.Node(~isnan(T.Node)),'NodeColor','r')
```

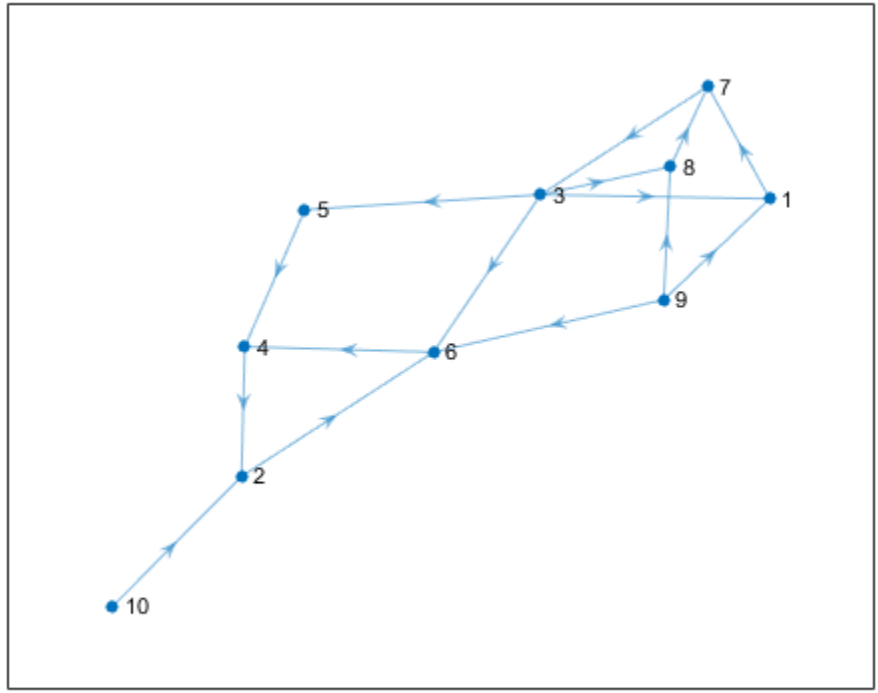


Remove Cycles from Graph

Make a directed graph acyclic by reversing some of its edges.

Create and plot a directed graph.

```
s = [1 2 3 3 3 3 4 5 6 7 8 9 9 9 10];
t = [7 6 1 5 6 8 2 4 4 3 7 1 6 8 2];
g = digraph(s,t);
plot(g, 'Layout', 'force')
```



Perform a depth-first search on the graph, flagging the `'edgetodiscovered'` event. This event corresponds to edges that complete a cycle.

```
[e,edge_indices] = dfsearch(g, 1, 'edgetodiscovered', 'Restart', true)
```

```
e = 3×2

     3     1
     6     4
     8     7


edge_indices = 3×1
```

```
     3
     9
    11
```

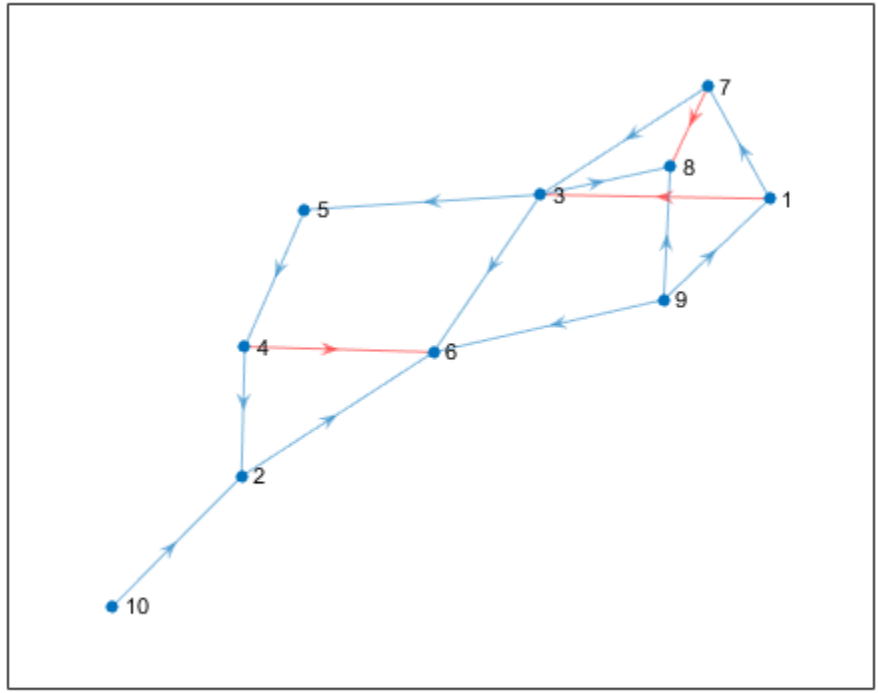
Use `flipedge` to reverse the direction of the flagged edges, so that they no longer complete a cycle. This removes all cycles from the graph. Use `isdag` to confirm that the graph is acyclic.

```
gnew = flipedge(g, edge_indices);
isdag(gnew)
```

```
ans = logical
   1
```

Plot the new graph and highlight the edges that were flipped.

```
p = plot(gnew, 'Layout', 'force');
highlight(p,'Edges',findedge(gnew,e(:,2),e(:,1)),'EdgeColor','r')
```



Input Arguments

G — Input graph
graph object | digraph object

Input graph, specified as either a `graph` or `digraph` object. Use `graph` to create an undirected graph or `digraph` to create a directed graph.

Example: `G = graph(1,2)`

Example: `G = digraph([1 2],[2 3])`

s — Starting node
node index | node name

Starting node, specified as one of the values in this table.

Value	Example
Scalar node index	`1`
Character vector node name	`'A'`
String scalar node name	`"A"`

Example: `dfsearch(G,1)`

events — Flagged search events
`'discovernode'` (default) | `'startnode'` | `'finishnode'` | `'edgetonew'` | `'edgetodiscovered'` | `'edgetofinished'` | `'allevents'` | cell array | string array

Flagged search events, specified as one of the options in the following table.

- To flag single events, use the flag names.
- To flag a subset of events, put two or more flag names into a cell array or string array.
- To flag all events, use `'allevents'`.

Note Depending on the value of `events`, the output of `dfsearch` varies. See the last column in the following table for information about the output returned by each option.

Value of events	Description	Output
`'discovernode'` (default)	A new node has been discovered.	Return a vector of node IDs: • If `s` is a numeric node index, then the vector contains numeric node indices. • If `s` is a node name, then the vector is a cell array containing node names.
`'finishnode'`	All outgoing edges from the node have been visited.	

Value of events	Description	Output
`'startnode'`	This flag indicates the starting node in the search. If `'Restart'` is `true`, then `'startnode'` flags the starting node each time the search restarts.	
`'edgetonew'`	Edge connects to an undiscovered node.	Return a matrix or cell array of size `N`-by-`2` that specifies the end nodes of edges in the graph: • If `s` is a numeric node index, then the matrix contains numeric node indices. • If `s` is a node name, then the matrix is a cell array containing node names. Additionally, you can specify a second output with `[T,E] = dfsearch(…)` that returns a vector of edge indices `E`.
`'edgetodiscovered'`	Edge connects to a previously discovered node.	
`'edgetofinished'`	Edge connects to a finished node.	
cell array	Specify two or more flags in a cell array to only flag those events during the search.	Return a table, `T`, which contains the variables `T.Event`, `T.Node`, `T.Edge`, and `T.EdgeIndex`: • `T.Event` is a categorical vector containing the flags in order of their occurrence. • `T.Node` contains the node ID of the corresponding node for the events

Value of events	Description	Output
`'allevents'`	All events are flagged.	`'discovernode'`, `'finishnode'`, and `'startnode'`. • `T.Edge` contains the corresponding edge for the events `'edgetonew'`, `'edgetodiscovered'`, and `'edgetofinished'`. • `T.EdgeIndex` contains the edge index for the events `'edgetonew'`, `'edgetodiscovered'`, and `'edgetofinished'`. The edge index is for unique identification of repeated edges in a multigraph. • Unused elements of `T.Node` and `T.Edge` are set to `NaN`. • If `s` is a numeric node index, then `T.Node` and `T.Edge` contain numeric node indices. • If `s` is a node name, then `T.Node` and `T.Edge` are cell arrays containing node names.

Example: `v = dfsearch(G,3)` begins the search at the third node and returns a vector, `v`, containing the nodes in order of discovery. This is the same as `v = dfsearch(G,3,'discovernode')`.

Example: `X = dfsearch(G,'A','edgetonew')` begins at the node named `'A'` and returns a cell array of character vectors, `X`, indicating each of the edges that connects to an undiscovered node during the search.

Example: `T = dfsearch(G,s,{'discovernode','finishnode'})` returns a table, `T`, but only flags when new nodes are discovered or when a node is marked finished.

Example: `T = dfsearch(G,s,'allevents')` flags all search events and returns a table, `T`.

Data Types: `char` | `string` | `cell`

`tf` — Toggle to restart search
`false` (default) | `true`

Toggle to restart search, specified as `false` (default) or `true`. This option is useful if the graph contains nodes that are unreachable from the starting node. If `'Restart'` is `true`, then the search restarts whenever undiscovered nodes remain that are unreachable from the discovered nodes. The new start node is the node with smallest index that is still undiscovered. The restarting process repeats until `dfsearch` discovers all nodes.

`'Restart'` is `false` by default, so that the search only visits nodes that are reachable from the starting node.

When `'Restart'` is `true`, the `'discovernode'` and `'finishnode'` events occur once for each node in the graph. Also, each edge in the graph is flagged once by `'edgetonew'`, `'edgetodiscovered'`, or `'edgetofinished'`. The edges flagged by `'edgetonew'` form one or more trees.

Example: `T = dfsearch(graph([1 3],[2 4]),1,'Restart',true)` searches both of the connected components in the graph.

Data Types: `logical`

Output Arguments

`v` — Node IDs
numeric column vector | cell vector

Node IDs, returned in one of the following formats:

- If you use a numeric node ID to specify the starting node, `s`, then `v` is a numeric column vector of node indices.

- If `s` is a character vector or string containing a node name, then `v` is a cell vector containing node names.

The node IDs in `v` reflect the order of discovery by the depth-first graph search.

T — Search results
table | vector | matrix | cell array of node names

Search results, returned in one of the following formats:

- If `events` is not specified or is `'discovernode'`, `'finishnode'`, or `'startnode'`, then `T` is a vector of node IDs similar to `v`.
- If `events` is `'edgetonew'`, `'edgetodiscovered'`, or `'edgetofinished'`, then `T` is a matrix or cell array of size `N`-by-`2` indicating the source and target nodes for each relevant edge.
- If `events` is a cell array of search events or `'allevents'`, then `T` is a table containing the flagged search events. The table contains the search event flags in `T.Event`, relevant node IDs in `T.Node`, and relevant edges in `T.Edge` and `T.EdgeIndex`.

In all cases:

- The order of the elements or rows of `T` indicates their order of occurrence during the search.
- If you specify `s` as a numeric node ID, then `T` also refers to nodes using their numeric IDs.
- If you specify `s` as a node name, then `T` also refers to nodes using their names.

E — Edge indices
vector

Edge indices, returned as a vector.

Specify this output to get a vector of edge indices for the events `'edgetonew'`, `'edgetodiscovered'`, or `'edgetofinished'`. The `N`-by-`1` vector of edge indices corresponds with `T`, which is a matrix or cell array of size `N`-by-`2` indicating the source and target nodes for each relevant edge.

Example: `[T,E] = dfsearch(G,s,'edgetonew')`

Tips

- `dfsearch` and `bfsearch` treat undirected graphs the same as directed graphs. An undirected edge between nodes `s` and `t` is treated like two directed edges, one from `s` to `t` and one from `t` to `s`.

Algorithms

The Depth-First search algorithm begins at the starting node, `s`, and inspects the neighbor of `s` that has the smallest node index. Then for that neighbor, it inspects the next undiscovered neighbor with the lowest index. This continues until the search encounters a node whose neighbors have all been visited. At that point, the search backtracks along the path to the nearest previously discovered node that has an undiscovered neighbor. This process continues until all nodes that are reachable from the starting node have been visited.

In pseudo-code, the (recursive) algorithm can be written as:

```
Event startnode(S)
Call DFS(S)

function DFS(C)

  Event discovernode(C)

  FOR edge E from outgoing edges of node C, connecting to node N
    Event edgetonew(C,E), edgetodiscovered(C,E) or edgetofinished(C,E)
    (depending on the state of node N)
    IF event was edgetonew
      Call DFS(N)
    END
  END

Event finishnode(C)

END
```

`dfsearch` can return flags to describe the different events in the algorithm, such as when a new node is discovered or when all of the outgoing edges of a node have been visited. The event flags are listed here.

Event Flag	Event Description
`'discovernode'`	A new node has been discovered.
`'finishnode'`	All outgoing edges from the node have been visited.
`'startnode'`	This flag indicates the starting node in the search.
`'edgetonew'`	Edge connects to an undiscovered node
`'edgetodiscovered'`	Edge connects to a previously discovered node
`'edgetofinished'`	Edge connects to a finished node

For more information, see the input argument description for `events`.

Note In cases where the input graph contains nodes that are unreachable from the starting node, the `'Restart'` option provides a way to make the search visit every node in the graph. In that case, the `'startnode'` event indicates the starting node each time the search restarts.

See Also

`bfsearch` | `conncomp` | `digraph` | `graph`

Topics

"Visualize Breadth-First and Depth-First Search"

Introduced in R2015b

distances

Shortest path distances of all node pairs

Syntax

```
d = distances(G)
d = distances(G,s)
d = distances(G,s,t)
d = distances( ___ ,'Method',algorithm)
```

Description

`d = distances(G)` returns a matrix, `d`, where `d(i,j)` is the length of the shortest path between node `i` and node `j`. If the graph is weighted (that is, `G.Edges` contains a variable `Weight`), then those weights are used as the distances along the edges in the graph. Otherwise, all edge distances are taken to be `1`.

`d = distances(G,s)` restricts the source nodes to the nodes defined by `s`, such that `d(i,j)` is the distance from node `s(i)` to node `j`.

`d = distances(G,s,t)` additionally restricts the target nodes to the nodes defined by `t`, such that `d(i,j)` is the distance from node `s(i)` to node `t(j)`.

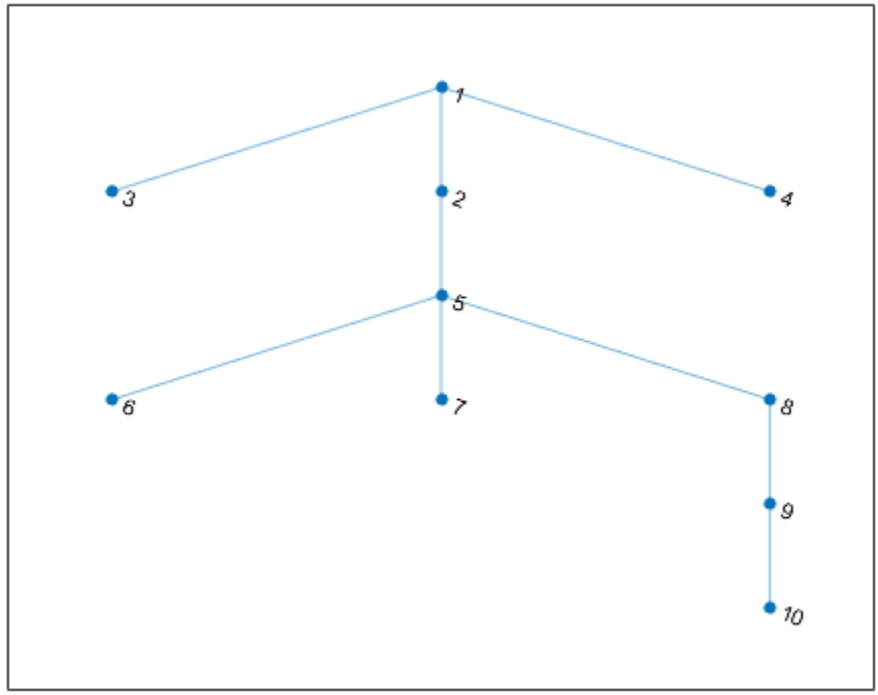
`d = distances( ___ ,'Method',algorithm)` optionally specifies the algorithm to use in computing the shortest path using any of the input arguments in previous syntaxes. For example, if `G` is a weighted graph, then `distances(G,'Method','unweighted')` ignores the edge weights in `G` and instead treats all edge weights as `1`.

Examples

Shortest Path Distance for All Node Pairs

Create and plot a graph.

```
s = [1 1 1 2 5 5 5 8 9];
t = [2 3 4 5 6 7 8 9 10];
G = graph(s,t);
plot(G)
```



Calculate the shortest path distance between all node pairs in the graph. Since the graph edges do not have weights, all edge distances are taken to be 1.

```
d = distances(G)
```

```
d = 10×10

     0     1     1     1     2     3     3     3     4     5
     1     0     2     2     1     2     2     2     3     4
```

```
     1     2     0     2     3     4     4     4     5     6
     1     2     2     0     3     4     4     4     5     6
     2     1     3     3     0     1     1     1     2     3
     3     2     4     4     1     0     2     2     3     4
     3     2     4     4     1     2     0     2     3     4
     3     2     4     4     1     2     2     0     1     2
     4     3     5     5     2     3     3     1     0     1
     5     4     6     6     3     4     4     2     1     0
```

`d` is symmetric because `G` is an undirected graph. In general `d(i,j)` is the length of the shortest path between node `i` and node `j`, and for undirected graphs this is equivalent to `d(j,i)`.

For example, find the length of the shortest path between node 1 and node 10.

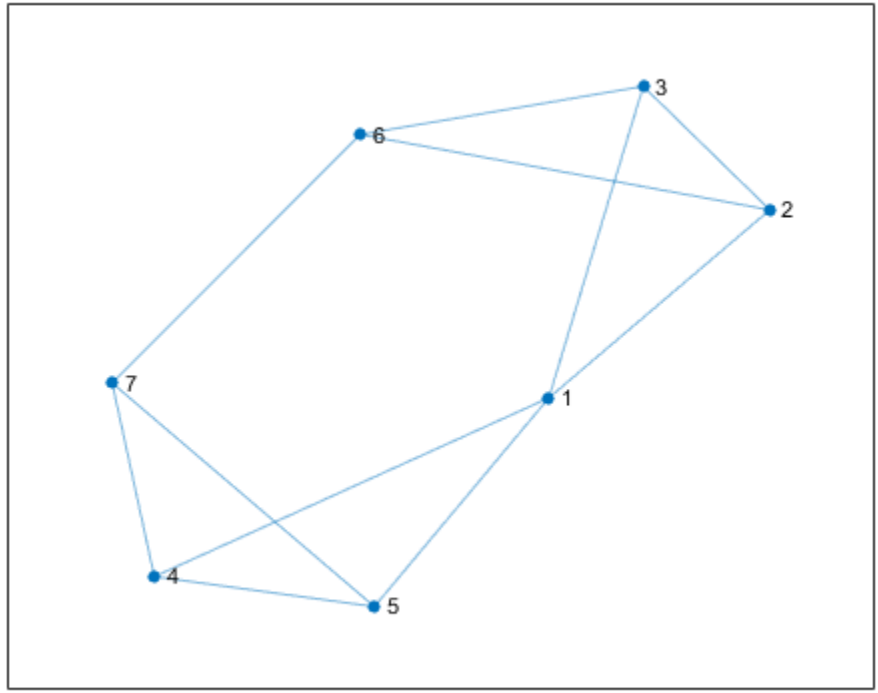
```
d(1,10)
```

```
ans = 5
```

Shortest Path Distances from Specified Sources

Create and plot a graph.

```
s = [1 1 1 1 2 2 3 4 4 5 6];
t = [2 3 4 5 3 6 6 5 7 7 7];
G = graph(s,t);
plot(G)
```



Find the shortest path distances from node 1, node 2, and node 3 to all other nodes in the graph.

```
d = distances(G,[1 2 3])
```

```
d = 3×7

     0     1     1     1     1     2     2
     1     0     1     2     2     1     2
     1     1     0     2     2     1     2
```

Use `d` to find the shortest path distance from node 1 to node 7.

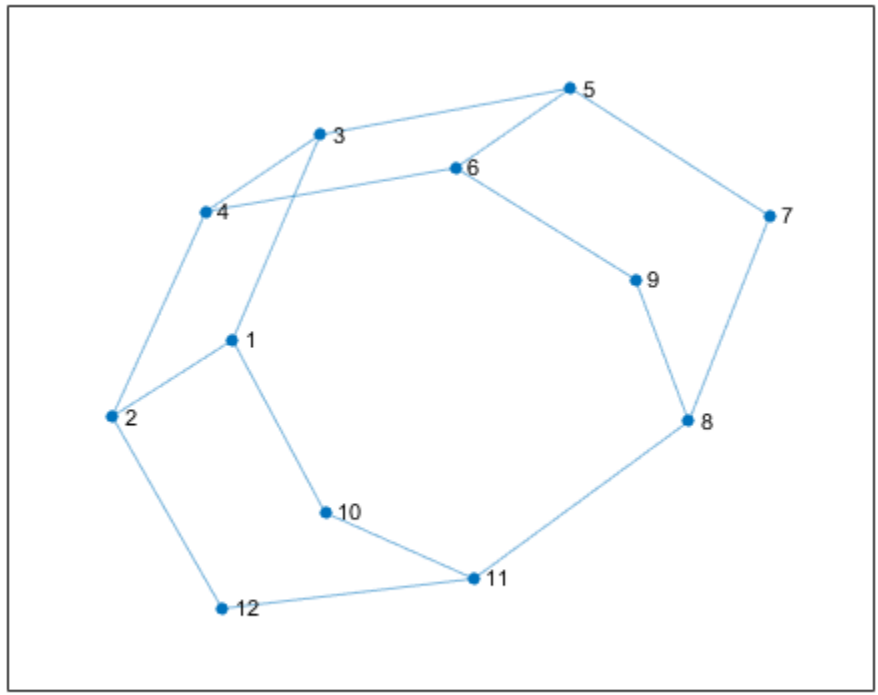
```
d(1,7)
```

```
ans = 2
```

Shortest Path Distances to Specified Targets

Create and plot a graph.

```
s = [1 1 1 2 2 3 3 4 5 5 6 7 8  8 10 11];
t = [2 3 10 4 12 5 4 6 6 7 9 8 9 11 11 12];
G = graph(s,t);
plot(G)
```



Find the shortest path distances from nodes 5 and 7 to nodes 2 and 3.

```
sources = [5 7];
targets = [2 3];
d = distances(G,sources,targets)
```

```
d = 2×2

     3     1
     4     2
```

Use `d` to find the shortest path distance between node 7 and node 3. In this case, `d(i,j)` is the distance from node `sources(i)` to node `targets(j)`.

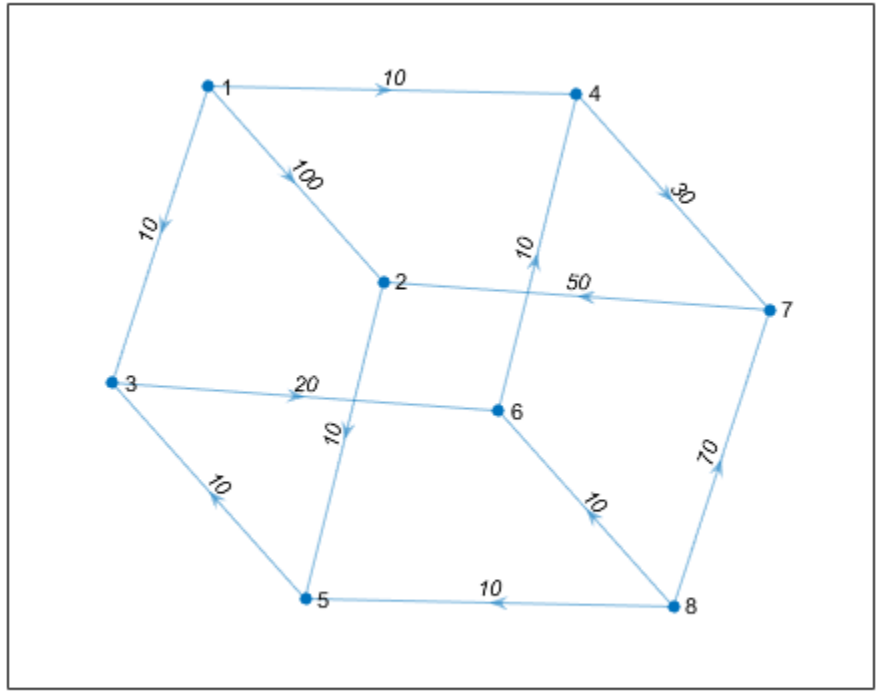
```
d(2,2)
```

```
ans = 2
```

Ignore Edge Weights

Create and plot a directed graph with weighted edges.

```
s = [1 1 1 2 5 3 6 4 7 8 8 8];
t = [2 3 4 5 3 6 4 7 2 6 7 5];
weights = [100 10 10 10 10 20 10 30 50 10 70 10];
G = digraph(s,t,weights);
plot(G,'EdgeLabel',G.Edges.Weight)
```



Find the shortest path distance between all pairs of graph nodes.

```
d = distances(G)
```

```
d = 8×8

     0    90    10    10   100    30    40   Inf
   Inf     0    20    50    10    40    80   Inf
   Inf   110     0    30   120    20    60   Inf
   Inf    80   100     0    90   120    30   Inf
   Inf   120    10    40     0    30    70   Inf
   Inf    90   110    10   100     0    40   Inf
   Inf    50    70   100    60    90     0   Inf
```

```
   Inf   100    20    20    10    10    50     0
```

Since `G` is a directed graph, `d` is not symmetric, and `d(i,j)` corresponds to the distance between nodes `i` and `j`. The `Inf` values in `d` correspond to nodes that are unreachable. For example, since node 1 has no predecessors, it is not possible to reach node 1 from any other node in the graph. So the first column of `d` contains many `Inf` values to reflect that node 1 is unreachable.

By default, `distances` uses the edge weights to compute the distances. Specify `'Method'` as `'unweighted'` to ignore the edge weights and treat all edge distances as 1.

```
d1 = distances(G,'Method','unweighted')
```

```
d1 = 8×8

     0     1     1     1     2     2     2   Inf
   Inf     0     2     4     1     3     5   Inf
   Inf     4     0     2     5     1     3   Inf
   Inf     2     4     0     3     5     1   Inf
   Inf     5     1     3     0     2     4   Inf
   Inf     3     5     1     4     0     2   Inf
   Inf     1     3     5     2     4     0   Inf
   Inf     2     2     2     1     1     1     0
```

Input Arguments

G — Input graph
`graph` object | `digraph` object

Input graph, specified as either a `graph` or `digraph` object. Use `graph` to create an undirected graph or `digraph` to create a directed graph.

Example: `G = graph(1,2)`

Example: `G = digraph([1 2],[2 3])`

s — Source nodes
`'all'` (default) | node indices | node names

Source nodes, specified as one or more node indices or node names, or `'all'` to select all source nodes.

This table shows the different ways to refer to one or more nodes either by their numeric node indices or by their node names.

Form	Single Node	Multiple Nodes
Node index	Scalar **Example:** `1`	Vector **Example:** `[1 2 3]`
Node name	Character vector **Example:** `'A'`	Cell array of character vectors **Example:** `{'A' 'B' 'C'}`
	String scalar **Example:** `"A"`	String array **Example:** `["A" "B" "C"]`

`s` and `t` must not specify nodes named `'all'` or `'Method'`, since these node names conflict with option names. Use `findnode` to instead pass in the node index for these cases.

Example: `distances(G,[1 2])`

Example: `distances(G,'all',[1 3 5])`

`t` — Target nodes
`'all'` (default) | node indices | node names

Target nodes, specified as one or more node indices or node names, or `'all'` to select all target nodes.

`s` and `t` must not specify nodes named `'all'` or `'Method'`, since these node names conflict with option names. Use `findnode` to instead pass in the node index for these cases.

Example: `distances(G,[1 2])`

Example: `distances(G,'all',[1 3 5])`

`algorithm` — Shortest path algorithm
`'auto'` (default) | `'unweighted'` | `'positive'` | `'mixed'`

Shortest path algorithm, specified as one of the options in the table.

Option	Description
`'auto'` (default)	The `'auto'` option automatically selects the algorithm: • `'unweighted'` is used for `graph` and `digraph` inputs with no edge weights. • `'positive'` is used for all `graph` inputs that have edge weights, and requires the weights to be nonnegative. This option is also used for `digraph` inputs with nonnegative edge weights. • `'mixed'` is used for `digraph` inputs whose edge weights contain some negative values. The graph cannot have negative cycles.
`'unweighted'`	Breadth-First computation that treats all edge weights as `1`.
`'positive'`	Dijkstra algorithm that requires all edge weights to be nonnegative.
`'mixed'` (only for `digraph`)	Bellman-Ford algorithm for directed graphs that requires the graph to have no negative cycles. While `'mixed'` is slower than `'positive'` for the same problem, `'mixed'` is more versatile as it allows some edge weights to be negative.

Note For most graphs, `'unweighted'` is the fastest algorithm, followed by `'positive'`, and `'mixed'`.

Example: `distances(G,s,t,'Method','unweighted')`

Output Arguments

d — Shortest path distances between node pairs
matrix

Shortest path distances between node pairs, returned as a matrix. The size of `d` is (# source nodes)-by-(# target nodes). A value of `Inf` indicates a path that does not exist.

Tips

- The `shortestpath`, `shortestpathtree`, and `distances` functions do not support undirected graphs with negative edge weights, or more generally any graph containing a negative cycle, for these reasons:
  - A *negative cycle* is a path that leads from a node back to itself, with the sum of the edge weights on the path being negative. If a negative cycle is on a path between two nodes, then no shortest path exists between the nodes, since a shorter path can always be found by traversing the negative cycle.
  - A single negative edge weight in an undirected graph creates a negative cycle.

See Also

`digraph` | `graph` | `nearest` | `shortestpath` | `shortestpathtree`

Introduced in R2015b

edgecount

Number of edges between two nodes

Syntax

```
C = edgecount(G,s,t)
```

Description

`C = edgecount(G,s,t)` returns the number of edges between the source and target node pairs `s` and `t` in graph `G`.

Examples

Number of Edges Between Nodes

Create a graph and display the edges table for reference. Find the number of edges between node 1 and node 2.

```
G = graph([1 1 1 1 2 2],[2 2 2 3 4 5]);
G.Edges
```

```
ans=6×1 table
    EndNodes
    ________

     1    2
     1    2
     1    2
     1    3
     2    4
     2    5
```

```
N = edgecount(G,1,2)
```

```
N = 3
```

You also can specify a single source node and several target nodes. Find the number of edges between node 1 and all other nodes (including self-loops).

```
N = edgecount(G,1,1:numnodes(G))
```

```
N = 5×1

     0
     3
     1
     0
     0
```

Input Arguments

G — Input graph
graph object | digraph object

Input graph, specified as either a graph or digraph object. Use graph to create an undirected graph or digraph to create a directed graph.

Example: G = graph(1,2)

Example: G = digraph([1 2],[2 3])

s,t — Node pairs (as separate arguments)
node indices | node names

Node pairs, specified as separate arguments of node indices or node names. Similarly located elements in s and t specify the source and target nodes for edges in the graph. s and t must specify the same number of nodes, unless one of them specifies a single node.

This table shows the different ways to refer to one or more nodes either by their numeric node indices or by their node names.

Form	Single Node	Multiple Nodes
Node index	Scalar **Example:** `1`	Vector **Example:** `[1 2 3]`
Node name	Character vector **Example:** `'A'`	Cell array of character vectors **Example:** `{'A' 'B' 'C'}`
	String scalar **Example:** `"A"`	String array **Example:** `["A" "B" "C"]`

Example: `edgecount(G,1,2)`

Example: `edgecount(G,1,1:5)`

Example: `edgecount(G,{'a' 'b' 'c'},'a')`

See Also

`digraph` | `findedge` | `graph` | `simplify`

Introduced in R2018a

findedge

Locate edge in graph

Syntax

```
[sOut,tOut] = findedge(G)
[sOut,tOut] = findedge(G,idx)
idxOut = findedge(G,s,t)
[idxOut,m] = findedge(G,s,t)
```

Description

`[sOut,tOut] = findedge(G)` returns the source and target node IDs, `sOut` and `tOut`, for all of the edges in graph `G`.

`[sOut,tOut] = findedge(G,idx)` finds the source and target nodes of the edges specified by `idx`.

`idxOut = findedge(G,s,t)` returns the numeric edge indices, `idxOut`, for the edges specified by the source and target node pairs `s` and `t`. The edge indices correspond to the rows `G.Edges.Edge(idxOut,:)` in the `G.Edges` table of the graph. If there are multiple edges between `s` and `t`, then all their indices are returned. An edge index of `0` indicates an edge that is not in the graph.

`[idxOut,m] = findedge(G,s,t)` additionally returns a vector `m` indicating which node pair `(s,t)` is associated with each edge index in `idxOut`. This is useful when there are multiple edges between the same two nodes.

Examples

Locate Edges with Specified End Nodes

Create a graph, and then determine the edge index for the (1,2) and (3,5) edges.

```
s = [1 1 2 2 2 3 3 3];
t = [2 3 3 4 5 6 7 5];
G = graph(s,t)
```

```
G = 
  graph with properties:

    Edges: [8x1 table]
    Nodes: [7x0 table]
```

```
idxOut = findedge(G,[1 3],[2 5])
```

```
idxOut = 2×1

     1
     6
```

`idxOut` contains the row index into `G.Edges.EndNodes` for each specified edge.

End Nodes of All Graph Edges

Create a graph, and then determine the end nodes of all edges in the graph.

```
s = {'a' 'a' 'b' 'b' 'c' 'c'};
t = {'b' 'c' 'd' 'e' 'f' 'g'};
G = graph(s,t);
G.Edges
```

```
ans=6×1 table
       EndNodes
    _____________

    {'a'}    {'b'}
    {'a'}    {'c'}
    {'b'}    {'d'}
    {'b'}    {'e'}
    {'c'}    {'f'}
    {'c'}    {'g'}
```

```
[sOut,tOut] = findedge(G)
```

```
sOut = 6×1

     1
     1
     2
     2
     3
     3


tOut = 6×1

     2
     3
     4
     5
     6
     7
```

Locate Edges with Specified Edge Indices

Create a graph, and then determine the end nodes for the edges whose indices are 3 and 7.

```
s = [1 1 1 1 2 2 3 3 4 4];
t = [2 3 4 5 6 7 8 9 10 11];
G = digraph(s,t)
```

```
G = 
  digraph with properties:

    Edges: [10x1 table]
    Nodes: [11x0 table]
```

```
[sOut,tOut] = findedge(G,[3 7])
```

```
sOut = 2×1

     1
     3
```

```
tOut = 2×1

     4
     8
```

Determine Weight of Specified Edge

Create a graph.

```
s = [1 1 2 3];
t = [2 3 3 4];
weights = [10 20 30 40];
G = graph(s,t,weights)
```

```
G = 
  graph with properties:

    Edges: [4x2 table]
    Nodes: [4x0 table]
```

Find the weight of the (1,3) edge, using `findedge` to retrieve the index.

```
G.Edges.Weight(findedge(G,1,3))
```

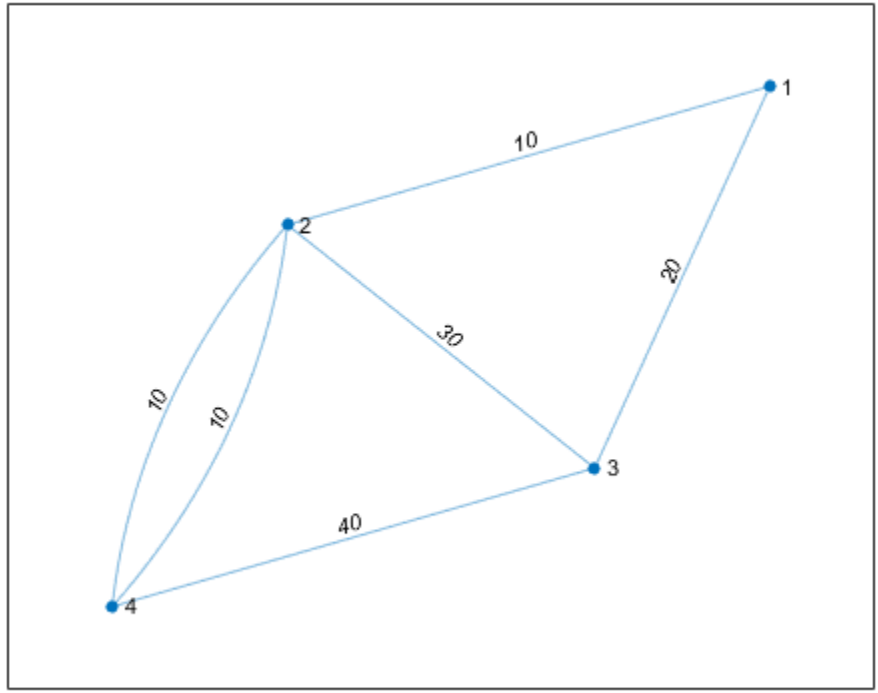
```
ans = 20
```

Change Weights of Multigraph Edges

Use `findedge` to change the weights of several multigraph edges.

Create and plot a multigraph. This graph has two edges between node 2 and node 4.

```
s = [1 1 2 3 2 2];
t = [2 3 3 4 4 4];
weights = [10 20 30 40 10 10];
G = graph(s,t,weights);
plot(G,'EdgeLabel',G.Edges.Weight)
```



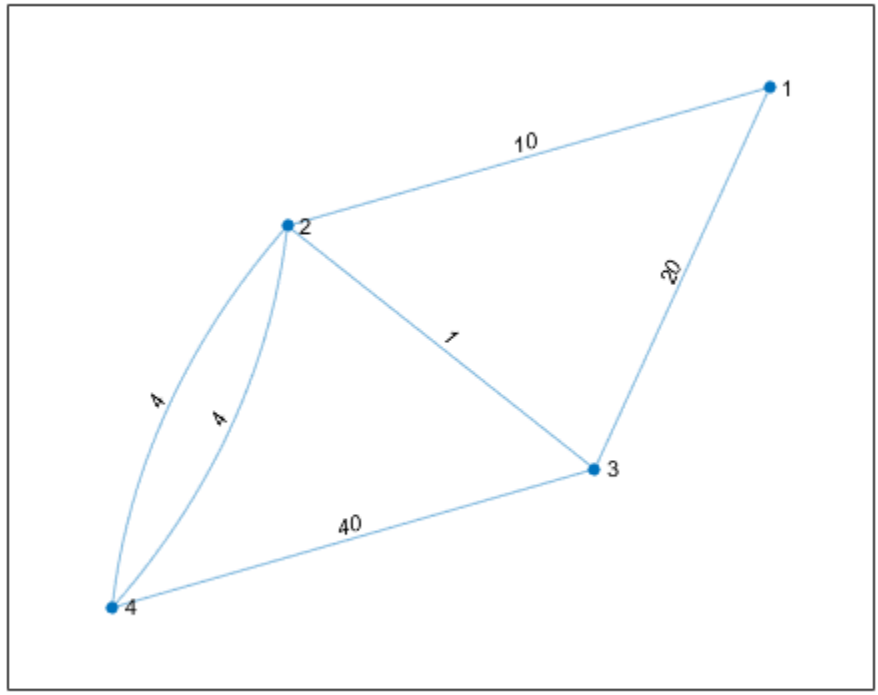
Change the weights of the edges between nodes (3,2) and (2,4). Specify two outputs to `findedge` to get the end-node indices, `m`. This output is useful when there are multiple edges between two nodes, since `idxOut` can have more elements than the number of node pairs in `s` and `t`. The edge `idxOut(1) = 3` connects the node pair `(s(1),t(1)) = (3,2)`, and the edges `idxOut(2) = 4` and `idxOut(3) = 5` connect the edge `(s(2),t(2)) = (2,4)`.

```
s = [3 2];
t = [2 4];
w = [1 4];
[idxOut, m] = findedge(G, s, t)
```

```
idxOut = 3×1
```

```
     3
     4
     5


m = 3×1

     1
     2
     2
```

```
G.Edges.Weight(idxOut) = w(m);
plot(G,'EdgeLabel',G.Edges.Weight)
```



Input Arguments

G — Input graph
graph object | digraph object

Input graph, specified as either a graph or digraph object. Use graph to create an undirected graph or digraph to create a directed graph.

Example: G = graph(1,2)

Example: G = digraph([1 2],[2 3])

s,t — Node pairs (as separate arguments)
node indices | node names

Node pairs, specified as separate arguments of node indices or node names. Similarly located elements in s and t specify the source and target nodes for edges in the graph.

This table shows the different ways to refer to one or more nodes either by their numeric node indices or by their node names.

Form	Single Node	Multiple Nodes
Node index	Scalar **Example:** 1	Vector **Example:** [1 2 3]
Node name	Character vector **Example:** 'A'	Cell array of character vectors **Example:** {'A' 'B' 'C'}
	String scalar **Example:** "A"	String array **Example:** ["A" "B" "C"]
	Categorical array **Example:** categorical("A")	Categorical array **Example:** categorical(["A" "B" "C"])

Example: G = findedge(G,[1 2],[3 4])

Example: G = findedge(G,{'a' 'a'; 'b' 'c'},{'b' 'c'; 'c' 'e'})

`idx` — Edge indices
scalar | vector

Edge indices, specified as a scalar or vector of positive integers. The edge index corresponds to a row in the `G.Edges` table of the graph, `G.Edges(idx,:)`.

Output Arguments

`idxOut` — Edge indices
scalar | vector

Edge indices, returned as a scalar or vector of nonnegative integers. The edge indices correspond to rows in the `G.Edges` table of the graph, `G.Edges(idxOut,:)`. An edge index of `0` indicates an edge that is not in the graph.

The length of `idxOut` corresponds to the number of node pairs in the input, unless the input graph is a multigraph.

`m` — End node indices
vector

End node indices, returned as a vector. The values in `m` connect the edge indices in `idxOut` to the input node pairs `(s,t)`. The edge `idxOut(j)` connects the node pair with index `m(j)`.

`sOut,tOut` — Node IDs
scalars | vectors

Node IDs, returned as separate scalars or vectors of positive integers. Similarly located elements in `sOut` and `tOut` specify the source and target nodes that form the edges `G.Edges(idx,:)`.

See Also

`digraph` | `findnode` | `graph` | `numedges`

Topics

"Directed and Undirected Graphs"
"Modify Nodes and Edges of Existing Graph"

"Add Graph Node Names, Edge Weights, and Other Attributes"

Introduced in R2015b

findnode

Locate node in graph

Syntax

```
k = findnode(G,nodeID)
```

Description

`k = findnode(G,nodeID)` returns the numeric node ID, `k`, of the node in graph `G` whose name or index is `nodeID`. The numeric node ID is zero if the node is not in the graph.

Examples

Determine Index of Named Node

Create a graph, and then determine the numeric node index for the nodes named `'AB'` and `'BC'`.

```
s = {'AA' 'AA' 'AA' 'AB' 'AC' 'BB'};
t = {'BA' 'BB' 'BC' 'BA' 'AB' 'BC'};
G = graph(s,t)
```

```
G = 
  graph with properties:

    Edges: [6x1 table]
    Nodes: [6x1 table]
```

```
k = findnode(G,{'AB' 'BC'})
```

```
k = 2×1
```

```
     5
     4
```

k contains the row index into `G.Nodes.Name` for each specified node.

Input Arguments

G — Input graph
graph object | digraph object

Input graph, specified as either a `graph` or `digraph` object. Use `graph` to create an undirected graph or `digraph` to create a directed graph.

Example: `G = graph(1,2)`

Example: `G = digraph([1 2],[2 3])`

nodeID — Node identifiers
node indices | node names

Node identifiers, specified as one or more node indices or node names.

This table shows the different ways to refer to one or more nodes either by their numeric node indices or by their node names.

Form	Single Node	Multiple Nodes
Node index	Scalar **Example:** `1`	Vector **Example:** `[1 2 3]`
Node name	Character vector **Example:** `'A'`	Cell array of character vectors **Example:** `{'A' 'B' 'C'}`
	String scalar **Example:** `"A"`	String array **Example:** `["A" "B" "C"]`

Example: `k = findnode(G,'Chicago')` returns the numeric node ID for the node in graph `G` with the name `'Chicago'`.

Data Types: `char` | `cell` | `string`

Output Arguments

k — Numeric node IDs
scalar | column vector

Numeric node IDs, returned as a scalar or column vector. The values of `k` are indices into `G.Nodes(k,:)`.

See Also

`digraph` | `findedge` | `graph` | `numnodes`

Topics

"Directed and Undirected Graphs"
"Modify Nodes and Edges of Existing Graph"
"Add Graph Node Names, Edge Weights, and Other Attributes"

Introduced in R2015b

incidence

Graph incidence matrix

Syntax

```
I = incidence(G)
```

Description

`I = incidence(G)` returns the sparse incidence matrix for graph `G`. If `s` and `t` are the node IDs of the source and target nodes of the `j`th edge in `G`, then `I(s,j) = -1` and `I(t,j) = 1`. That is, each column of `I` indicates the source and target nodes for a single edge in `G`.

Examples

Graph Incidence Matrix

Create a graph using an edge list, and then calculate the graph incidence matrix.

```
s = [1 1 1 1 1];
t = [2 3 4 5 6];
G = graph(s,t);
I = incidence(G)
```

```
I = 
   (1,1)       -1
   (2,1)        1
   (1,2)       -1
   (3,2)        1
   (1,3)       -1
   (4,3)        1
   (1,4)       -1
   (5,4)        1
   (1,5)       -1
```

```
   (6,5)        1
```

Each column in `I` contains two nonzero entries, which indicate the end nodes of a single edge in `G`.

Calculate the graph Laplacian matrix, `L`, and confirm the relation `L = I*I'` for undirected graphs.

```
L = laplacian(G);
L - I*I'
```

```
ans =
   All zero sparse: 6x6
```

Incidence Matrix of Directed Graph

Create a directed graph using an edge list, and then calculate the incidence matrix.

```
s = [1 2 1 3 2 3 3 3];
t = [2 1 3 1 3 4 5 6];
G = digraph(s,t)
```

```
G = 
  digraph with properties:

    Edges: [8x1 table]
    Nodes: [6x0 table]
```

```
I = incidence(G)
```

```
I = 
   (1,1)       -1
   (2,1)        1
   (1,2)       -1
   (3,2)        1
   (1,3)        1
   (2,3)       -1
   (2,4)       -1
   (3,4)        1
   (1,5)        1
```

```
   (3,5)       -1
   (3,6)       -1
   (4,6)        1
   (3,7)       -1
   (5,7)        1
   (3,8)       -1
   (6,8)        1
```

Each column in `I` represents the source and target nodes of a single edge in `G`.

Input Arguments

`G` — Input graph
`graph` object | `digraph` object

Input graph, specified as either a `graph` or `digraph` object. Use `graph` to create an undirected graph or `digraph` to create a directed graph.

Example: `G = graph(1,2)`

Example: `G = digraph([1 2],[2 3])`

Output Arguments

`I` — Incidence matrix
sparse matrix

Incidence matrix, returned as a sparse matrix. The size of `I` is `numnodes(G)`-by-`numedges(G)`. The graph incidence matrix is undefined for graphs with self-loops.

Tips

- The `incidence` function calculates the variety of incidence matrix commonly known as a *signed* or *oriented* incidence matrix. The signed incidence matrix of an undirected graph, `I`, is related to the graph Laplacian matrix, `L`, such that `L == I*I'`.

See Also

`adjacency` | `digraph` | `graph` | `laplacian`

Introduced in R2015b

isomorphism

Compute isomorphism between two graphs

Syntax

```
P = isomorphism(G1,G2)
P = isomorphism( ___ ,Name,Value)
[P,edgeperm] = isomorphism( ___ )
```

Description

`P = isomorphism(G1,G2)` computes a graph isomorphism on page 1-5288 equivalence relation between graphs `G1` and `G2`, if one exists. If no isomorphism exists, then `P` is an empty array.

`P = isomorphism( ___ ,Name,Value)` specifies additional options with one or more name-value pair arguments. For example, you can specify `'NodeVariables'` and a list of node variables to indicate that the isomorphism must preserve these variables to be valid.

`[P,edgeperm] = isomorphism( ___ )` additionally returns a vector of edge permutations, `edgeperm`. This output enables you to preserve edge variables when working with multigraphs.

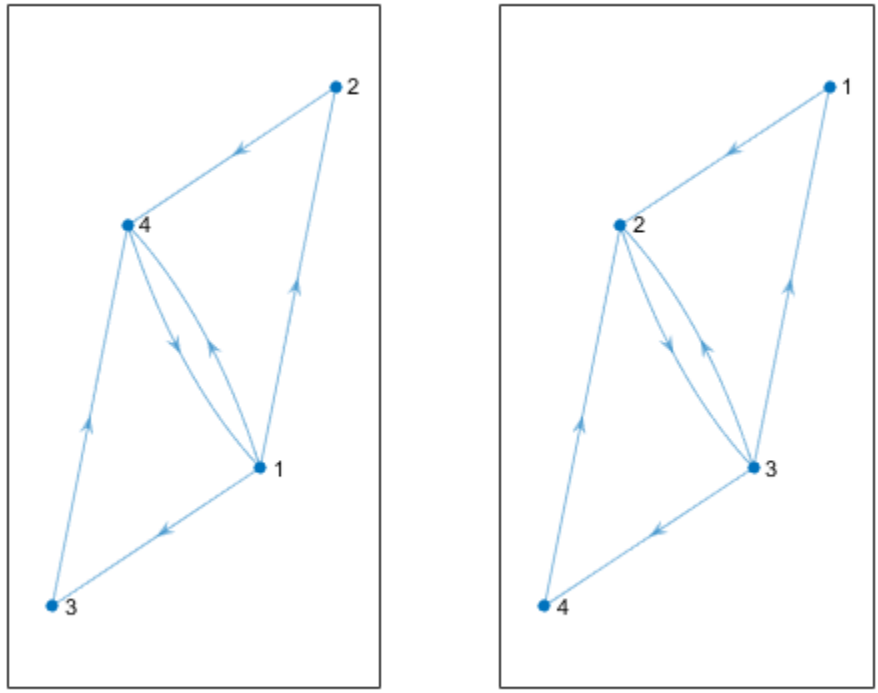
Examples

Compute Isomorphism Permutation

Create and plot two directed graphs, and then calculate the isomorphism relation between them.

```
G1 = digraph([1 1 1 2 3 4],[2 3 4 4 4 1]);
G2 = digraph([3 3 3 2 1 4],[1 4 2 3 2 2]);
subplot(1,2,1)
plot(G1)
```

```
subplot(1,2,2)
plot(G2)
```




```
p = isomorphism(G1,G2)
```

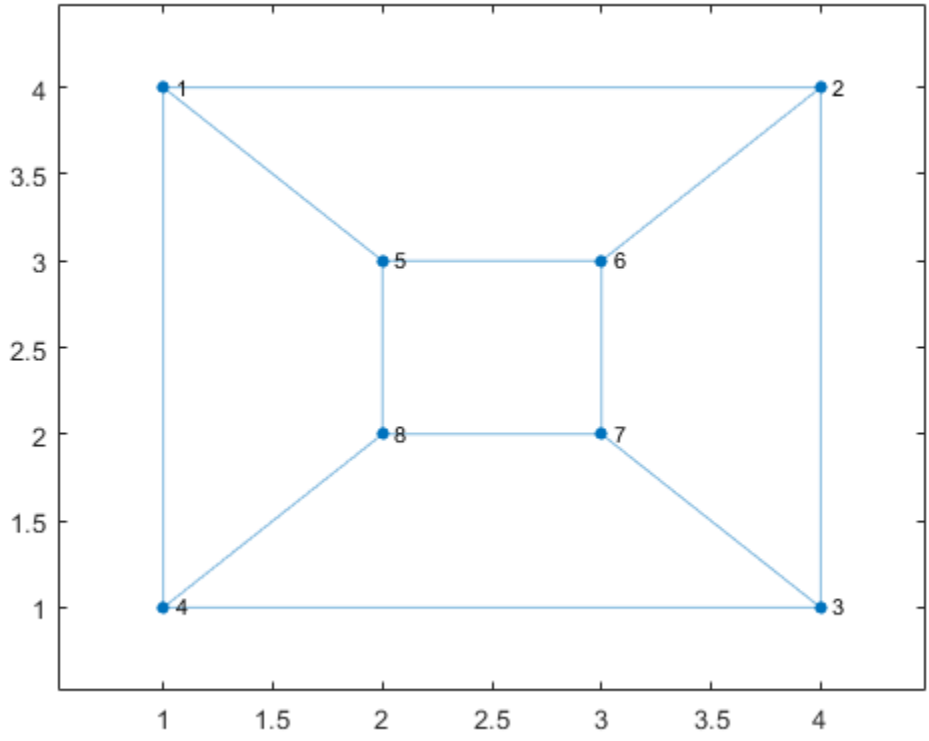
```
p = 4×1

     3
     1
     4
     2
```

The result indicates that `reordernodes(G2,p)` has the same structure as `G1`.

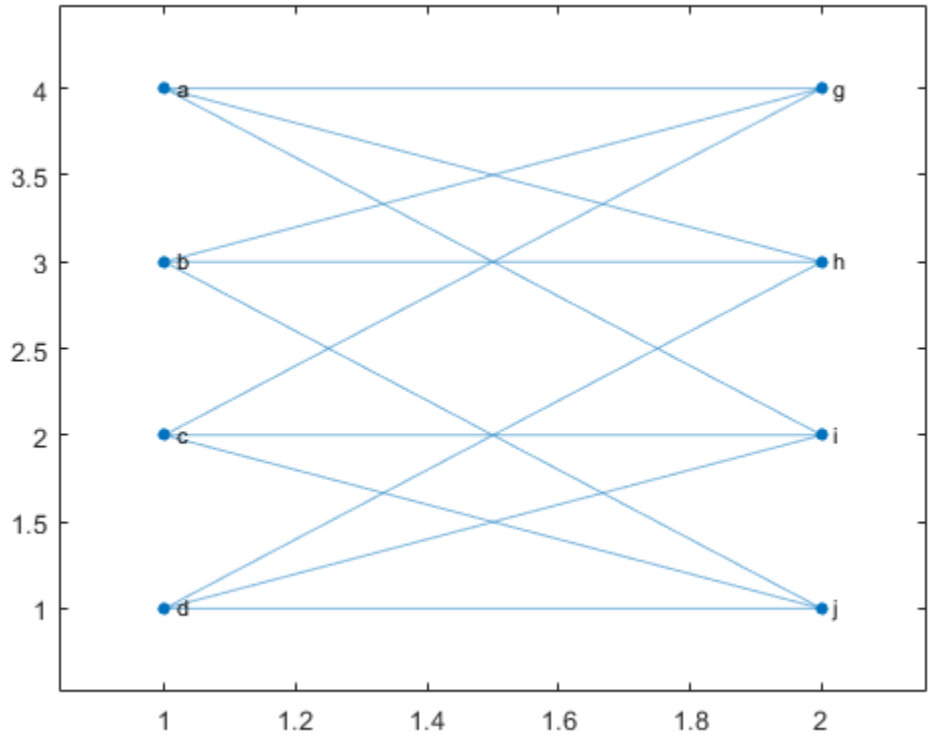
Compute Isomorphism Between Graphs with Different Labels and Layouts

Create and plot two graphs, G1 and G2.

```
G1 = graph([1 1 1 2 2 3 3 4 5 5 7 7],[2 4 5 3 6 4 7 8 6 8 6 8]);
plot(G1,'XData',[1 4 4 1 2 3 3 2],'YData',[4 4 1 1 3 3 2 2])
```




```
G2 = graph({'a' 'a' 'a' 'b' 'b' 'b' 'c' 'c' 'c' 'd' 'd' 'd'}, ...
    {'g' 'h' 'i' 'g' 'h' 'j' 'g' 'i' 'j' 'h' 'i' 'j'});
plot(G2,'XData',[1 2 2 2 1 2 1 1],'YData',[4 4 3 2 3 1 2 1])
```



Compute the isomorphism relation between the graphs, if one exists. The result indicates that the graph nodes can be permuted to represent the same graph despite their different labels and layouts.

```
p = isomorphism(G1,G2)
```

```
p = 8×1

     1
     2
     5
     3
     4
     7
```

```
6
8
```

Compute Isomorphism and Preserve Node Properties

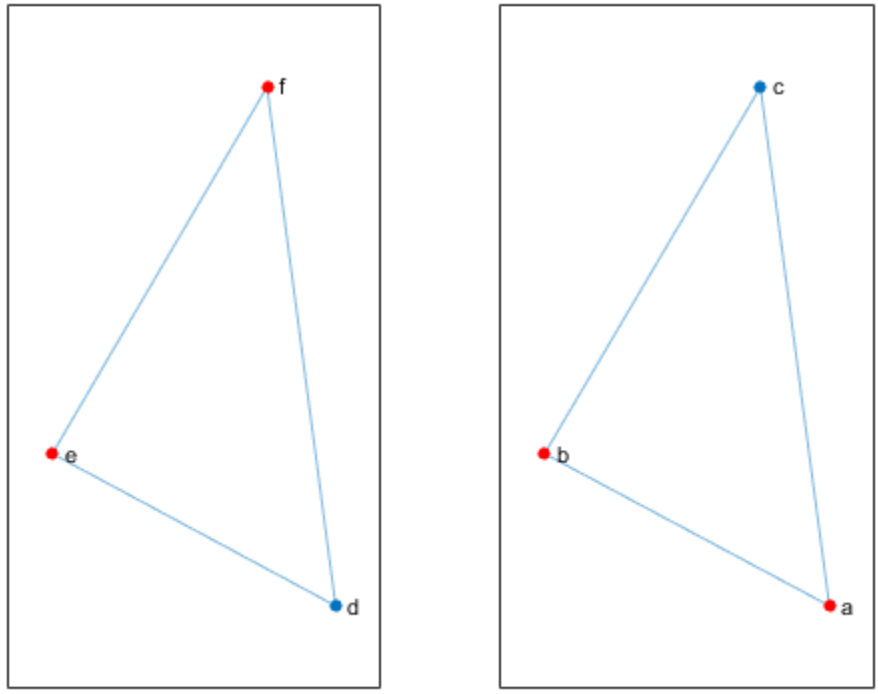
Compute two different isomorphism relations between two graphs. One of the relations preserves a node property, while the other ignores it.

Create two similar graphs. Add a node property `Color` to each of the graphs.

```
G1 = graph({'d' 'e' 'f'},{'e' 'f' 'd'});
G1.Nodes.Color = {'blue' 'red' 'red'}';

G2 = graph({'a' 'b' 'c'},{'b' 'c' 'a'});
G2.Nodes.Color = {'red' 'red' 'blue'}';
```

Plot the graphs side-by-side in the same figure. Color the nodes red that have `Color = 'red'`.

```
subplot(1,2,1)
p1 = plot(G1);
highlight(p1,{'e' 'f'},'NodeColor','r')

subplot(1,2,2)
p2 = plot(G2);
highlight(p2,{'a' 'b'},'NodeColor','r')
```



Compute the isomorphism between the graphs, ignoring the `Color` property.

```
p = isomorphism(G1,G2)
```

```
p = 3×1

     1
     2
     3
```

Compute the isomorphism again, but this time preserve the value of the `Color` property in the comparison. `isomorphism` returns a different permutation that preserves the `Color` property.

```
p = isomorphism(G1,G2,'NodeVariables','Color')
```

```
p = 3×1

     3
     1
     2
```

View the nodes in G1 and G2 that the isomorphism matches together.

```
[G1.Nodes.Name, G2.Nodes.Name(p)]
```

```
ans = 3x2 cell array
    {'d'}    {'c'}
    {'e'}    {'a'}
    {'f'}    {'b'}
```

Input Arguments

G1,G2 — Input graphs (as separate arguments)
graph objects | digraph objects

Input graphs, specified as separate arguments of graph or digraph objects. Use graph to create an undirected graph or digraph to create a directed graph.

G1 and G2 must be both graph objects or both digraph objects.

Example: G1 = graph(1,2)

Example: G1 = digraph([1 2],[2 3])

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: P = isomorphism(G1,G2,'NodeVariables',{'Var1' 'Var2'})

`EdgeVariables` — Edge variables to preserve
character vector | string scalar | cell array of character vectors | string array

Edge variables to preserve, specified as the comma-separated pair consisting of `'EdgeVariables'` and a character vector, string scalar, cell array of character vectors, or string array. Use this option to specify one or more edge variables that are in both `G1.Edges` and `G2.Edges`. The isomorphism must preserve the specified edge variables in order to be valid.

If `G` is a multigraph, then you can specify the second output `edgeperms` to enable reordering edge variables.

Data Types: `char` | `string` | `cell`

`NodeVariables` — Node variables to preserve
character vector | string scalar | cell array of character vectors | string array

Node variables to preserve, specified as the comma-separated pair consisting of `'NodeVariables'` and a character vector, string scalar, cell array of character vectors, or string array. Use this option to specify one or more node variables that are in both `G1.Nodes` and `G2.Nodes`. The isomorphism must preserve the specified node variables in order to be valid.

Data Types: `char` | `string` | `cell`

Output Arguments

`P` — Permutation vector for isomorphism
column vector | `[]`

Permutation vector for isomorphism, returned as a column vector when an isomorphism exists or as the empty array `[]` when an isomorphism does not exist. If `P` is not empty, then `reordernodes(G2,P)` has the same structure as `G1`.

`edgeperm` — Edge permutation
column vector

Edge permutation, returned as a column vector. When working with multigraphs, the edge permutation vector enables you to preserve edge variables specified by the `'EdgeVariables'` name-value pair. Use these commands to reorder the edge variables of repeated edges:

```
[p,edgeperm] = isomorphism(g1,g2,'EdgeVariables',edgevars);
g2perm = reordernodes(g2, p);
g2perm.Edges(:, 2:end) = g2perm.Edges(edgeperm, 2:end);
```

More About

Graph Isomorphism

Two graphs, `G1` and `G2`, are isomorphic if there exists a permutation of the nodes `P` such that `reordernodes(G2,P)` has the same structure as `G1`.

Two graphs that are isomorphic have similar structure. For example, if a graph contains one cycle, then all graphs isomorphic to that graph also contain one cycle.

See Also

`digraph` | `graph` | `isisomorphic` | `reordernodes`

Topics

"Directed and Undirected Graphs"

Introduced in R2016b

isisomorphic

Determine whether two graphs are isomorphic

Syntax

```
tf = isisomorphic(G1,G2)
tf = isisomorphic(G1,G2,Name,Value)
```

Description

`tf = isisomorphic(G1,G2)` returns logical `1` (`true`) if a graph isomorphism on page 1-5296 exists between graphs `G1` and `G2`; otherwise, it returns logical `0` (`false`).

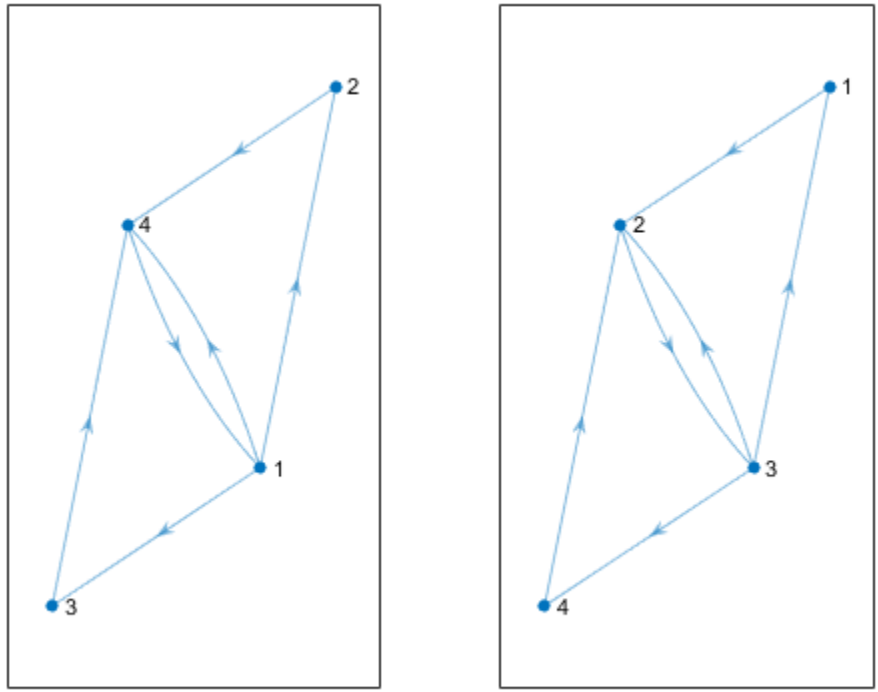
`tf = isisomorphic(G1,G2,Name,Value)` specifies additional options with one or more name-value pair arguments. For example, you can specify `'NodeVariables'` and a list of node variables to indicate that the isomorphism must preserve these variables to be valid.

Examples

Compare Graphs

Create and plot two directed graphs, and then determine if they are isomorphic.

```
G1 = digraph([1 1 1 2 3 4],[2 3 4 4 4 1]);
G2 = digraph([3 3 3 2 1 4],[1 4 2 3 2 2]);
subplot(1,2,1)
plot(G1)
subplot(1,2,2)
plot(G2)
```



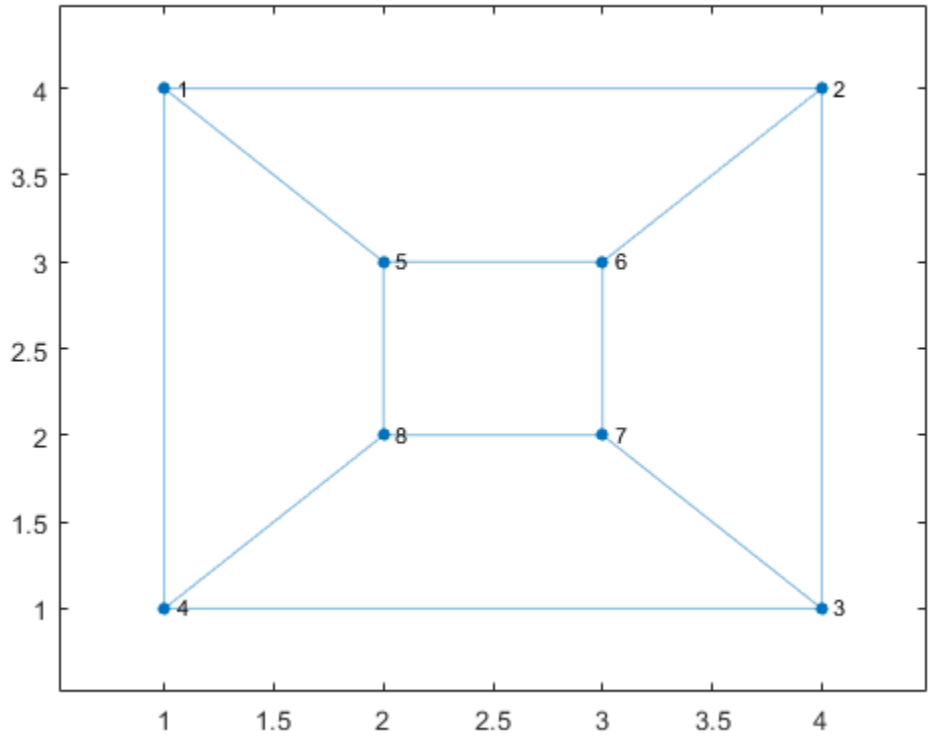

```
isisomorphic(G1,G2)
```

```
ans = logical
   1
```

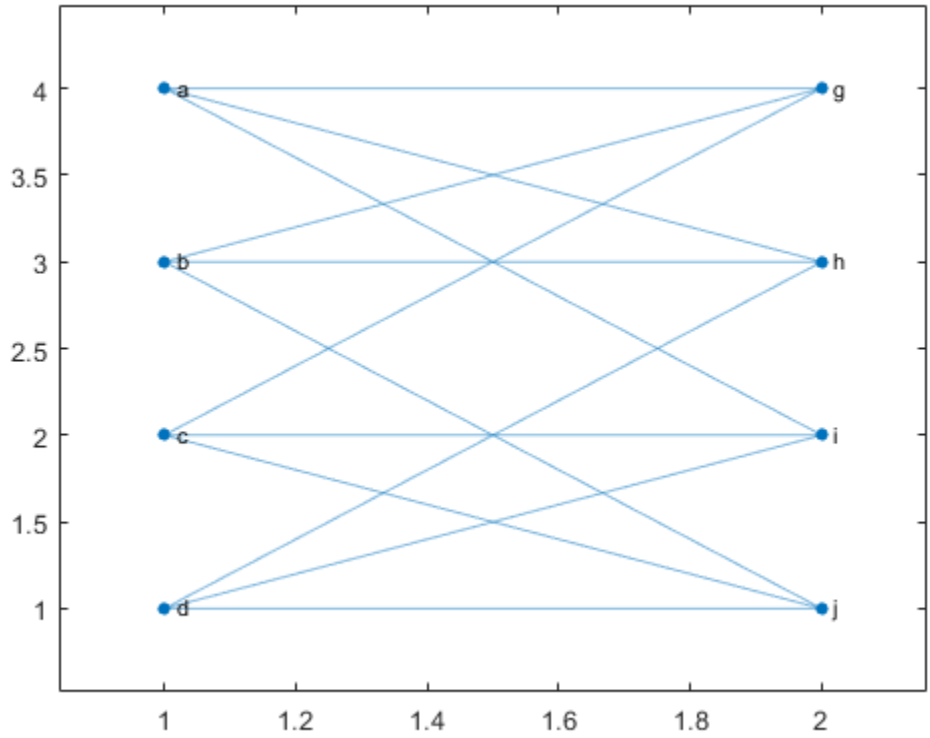
Compare Graphs with Different Labels and Layouts

Create and plot two graphs, G1 and G2.

```
G1 = graph([1 1 1 2 2 3 3 4 5 5 7 7],[2 4 5 3 6 4 7 8 6 8 6 8]);
plot(G1,'XData',[1 4 4 1 2 3 3 2],'YData',[4 4 1 1 3 3 2 2])
```




```
G2 = graph({'a' 'a' 'a' 'b' 'b' 'b' 'c' 'c' 'c' 'd' 'd' 'd'}, ...
    {'g' 'h' 'i' 'g' 'h' 'j' 'g' 'i' 'j' 'h' 'i' 'j'});
plot(G2,'XData',[1 2 2 2 1 2 1 1],'YData',[4 4 3 2 3 1 2 1])
```



Determine whether an isomorphism exists for `G1` and `G2`. The result indicates that the graphs are structurally the same despite their different labels and layouts.

```
tf = isisomorphic(G1,G2)
```

```
tf = logical
   1
```

Preserve Node Properties in Isomorphism Comparison

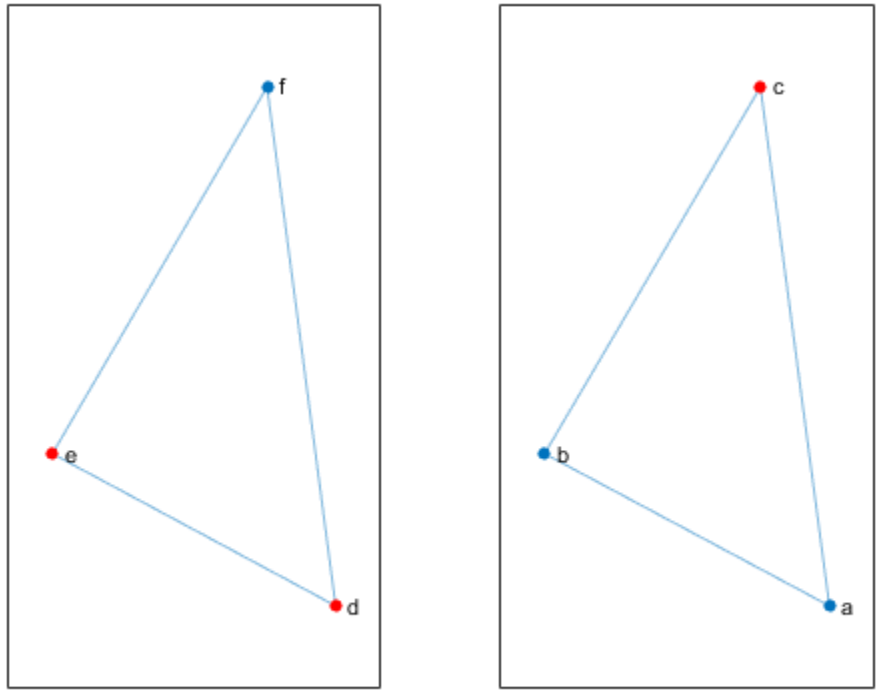
Use two different comparisons to determine if there is an isomorphism relation between two graphs. One of the comparisons preserves a node property, while the other ignores it.

Create two similar graphs. Add a node property `Color` to each of the graphs.

```
G1 = graph({'d' 'e' 'f'},{'e' 'f' 'd'});
G1.Nodes.Color = {'red' 'red' 'blue'}';

G2 = graph({'a' 'b' 'c'},{'b' 'c' 'a'});
G2.Nodes.Color = {'blue' 'blue' 'red'}';
```

Plot the graphs side-by-side in the same figure. Color the nodes red that have `Color = 'red'`.

```
subplot(1,2,1)
p1 = plot(G1);
highlight(p1,{'d' 'e'},'NodeColor','r')

subplot(1,2,2)
p2 = plot(G2);
highlight(p2,'c','NodeColor','r')
```



Determine if the graphs are isomorphic, ignoring the `Color` property.

```
tf = isisomorphic(G1,G2)
```

```
tf = logical
   1
```

Determine if the graphs are isomorphic and preserve the value of the `Color` property in the comparison. In this case, there is no isomorphism since the `Color` property of each graph contains different numbers of `'red'` and `'blue'` values.

```
tf = isisomorphic(G1,G2,'NodeVariables','Color')
```

```
tf = logical
   0
```

Input Arguments

G1,G2 — Input graphs (as separate arguments)
graph objects | digraph objects

Input graphs, specified as separate arguments of graph or digraph objects. Use graph to create an undirected graph or digraph to create a directed graph.

G1 and G2 must be both graph objects or both digraph objects.

Example: G1 = graph(1,2)

Example: G1 = digraph([1 2],[2 3])

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: tf = isisomorphic(G1,G2,'NodeVariables',{'Var1' 'Var2'})

EdgeVariables — Edge variables to preserve
character vector | string scalar | cell array of character vectors | string array

Edge variables to preserve, specified as the comma-separated pair consisting of 'EdgeVariables' and a character vector, string scalar, cell array of character vectors, or string array. Use this option to specify one or more edge variables that are in both G1.Edges and G2.Edges. The isomorphism comparison must preserve the specified edge variables in order to be valid. For multigraphs with multiple edges between the same two nodes, the ordering of the edge variables for the same node pair is irrelevant.

Data Types: char | string | cell

NodeVariables — Node variables to preserve
character vector | string scalar | cell array of character vectors | string array

Node variables to preserve, specified as the comma-separated pair consisting of `'NodeVariables'` and a character vector, string scalar, cell array of character vectors, or string array. Use this option to specify one or more node variables that are in both `G1.Nodes` and `G2.Nodes`. The isomorphism comparison must preserve the specified node variables in order to be valid.

Data Types: `char` | `string` | `cell`

More About

Graph Isomorphism

Two graphs, `G1` and `G2`, are isomorphic if there exists a permutation of the nodes `P` such that `reordernodes(G2,P)` has the same structure as `G1`.

Two graphs that are isomorphic have similar structure. For example, if a graph contains one cycle, then all graphs isomorphic to that graph also contain one cycle.

See Also

`digraph` | `graph` | `isomorphism` | `reordernodes`

Topics

"Directed and Undirected Graphs"

Introduced in R2016b

ismultigraph

Determine whether graph has multiple edges

Syntax

```
tf = ismultigraph(G)
```

Description

`tf = ismultigraph(G)` returns logical `1` (`true`) if `G` has multiple edges between any two nodes. Otherwise, it returns logical `0` (`false`).

By convention, `ismultigraph` returns logical `0` (`false`) for graphs that contain self-loops, but no repeated edges. However, a graph with multiple self-loops is considered to be a multigraph.

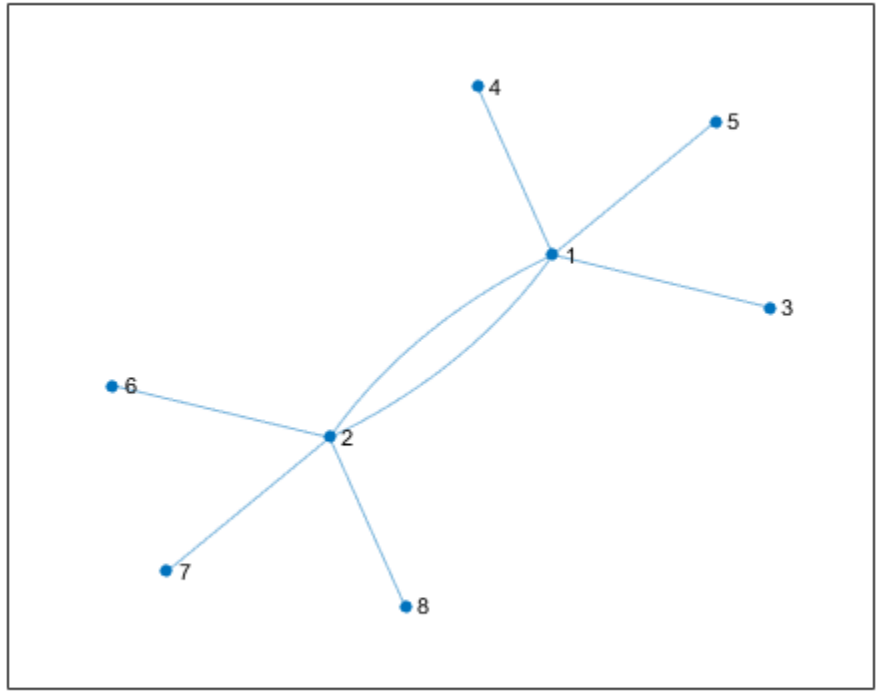
Examples

Test for Multiple Edges

Use `ismultigraph` to determine whether an input graph has multiple edges between two nodes.

Create a graph.

```
G = graph([1 1 1 1 1 2 2 2],[2 2 3 4 5 6 7 8]);
plot(G)
```



Check to see if `G` is a multigraph. The result is logical `1` (`true`) because there are two edges between nodes 1 and 2.

```
tf = ismultigraph(G)
```

```
tf = logical
   1
```

Remove Multiple Edges from Graph

Use `ismultigraph` to determine whether a graph needs to be simplified.

It is common to encounter duplicate edges when you create an empty graph and programmatically add edges to it with `addedge`. The data used for the edges needs to be unique to avoid duplicates.

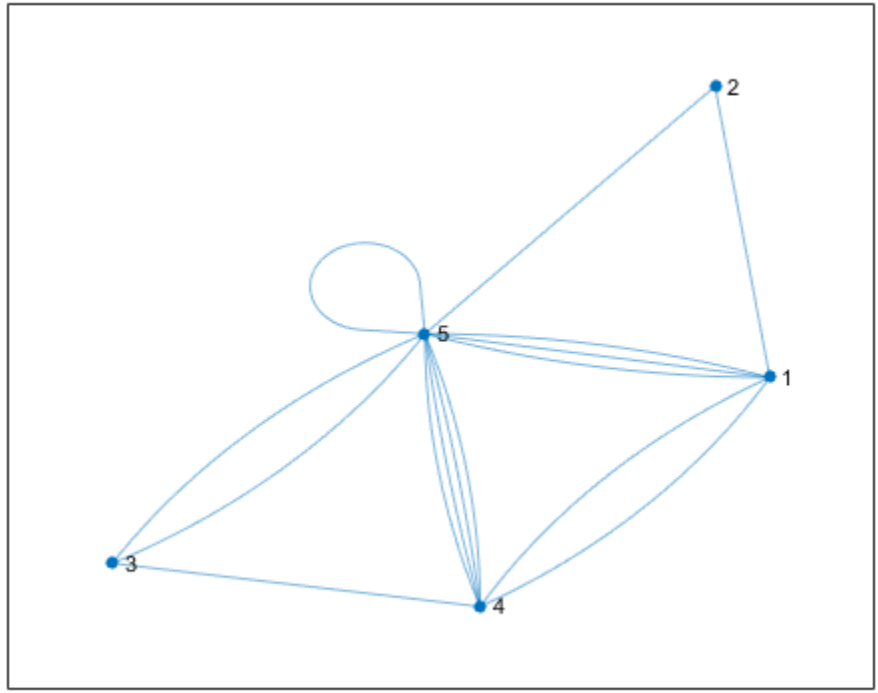
To demonstrate this, create an empty graph and a matrix with two columns of random numbers. Since the random numbers are only between 1 and 5, this data produces multiple edges.

```
G = graph;
rng default % for reproducibility
X = randi(5,15,2)
```

```
X = 15×2

     5     1
     5     3
     1     5
     5     4
     4     5
     1     4
     2     1
     3     5
     5     5
     5     4
       ⋮
```

Instead of cleaning the source data to ensure the rows are unique, add all of the edges to the graph. Plot the graph for reference.

```
G = addedge(G,X(:,1),X(:,2));
plot(G)
```

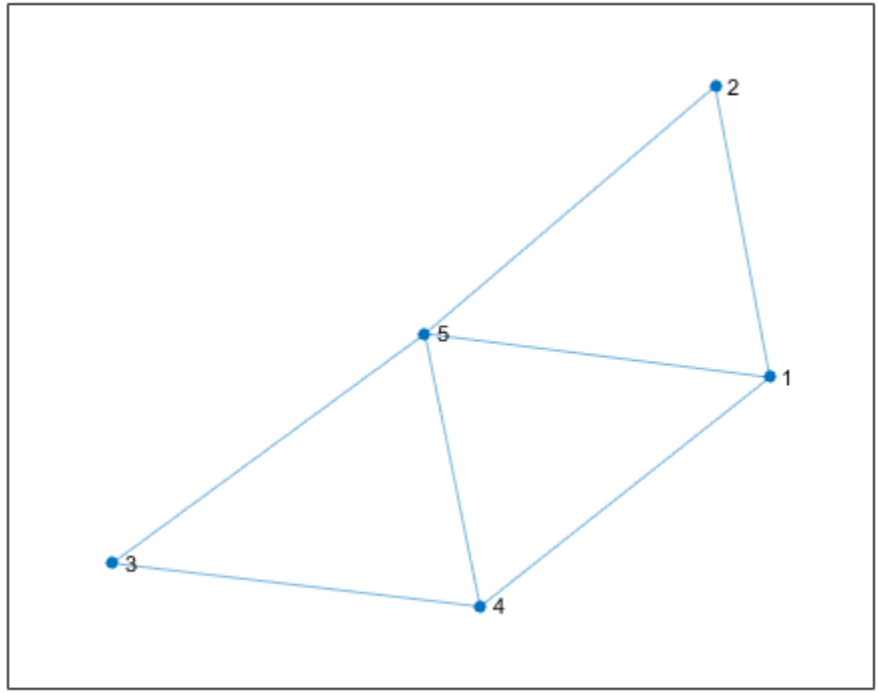


Test to see if the graph is a multigraph and, if it is, use `simplify` to remove repeated edges and self-loops.

```
if ismultigraph(G)
    G = simplify(G);
end
```

Plot the resulting graph.

```
plot(G)
```



Input Arguments

G — Input graph
graph object | digraph object

Input graph, specified as either a `graph` or `digraph` object. Use `graph` to create an undirected graph or `digraph` to create a directed graph.

Example: `G = graph(1,2)`

Example: `G = digraph([1 2],[2 3])`

See Also

`digraph` | `edgecount` | `graph` | `simplify`

Topics

"Directed and Undirected Graphs"

Introduced in R2018a

laplacian

Graph Laplacian matrix

Syntax

```
L = laplacian(G)
```

Description

`L = laplacian(G)` returns the graph Laplacian matrix, `L`. Each diagonal entry, `L(j,j)`, is given by the degree of node `j`, `degree(G,j)`. The off-diagonal entries of `L` represent the edges in `G` such that `L(i,j) = L(j,i) = -1` if there is an edge between nodes `i` and `j`; otherwise, `L(i,j) = L(j,i) = 0`. The input graph `G` cannot be a multigraph or contain self-loops, and edge weights are ignored.

Examples

Graph Laplacian Matrix

Create a graph using an edge list, and then calculate the graph Laplacian matrix.

```
s = [1 1 1 1 1];
t = [2 3 4 5 6];
G = graph(s,t);
L = laplacian(G)
```

```
L = 
   (1,1)        5
   (2,1)       -1
   (3,1)       -1
   (4,1)       -1
   (5,1)       -1
   (6,1)       -1
   (1,2)       -1
   (2,2)        1
```

```
   (1,3)       -1
   (3,3)        1
   (1,4)       -1
   (4,4)        1
   (1,5)       -1
   (5,5)        1
   (1,6)       -1
   (6,6)        1
```

The diagonal elements of `L` indicate the degree of the nodes, such that `L(j,j)` is the degree of node `j`.

Calculate the graph incidence matrix, `I`, and confirm the relation `L = I*I'`.

```
I = incidence(G);
L - I*I'
```

```
ans =
   All zero sparse: 6x6
```

Input Arguments

`G` — Input graph
`graph` object

Input graph, specified as a `graph` object. Use `graph` to create an undirected graph object.

Example: `G = graph(1,2)`

Output Arguments

`L` — Laplacian matrix
matrix

Laplacian matrix. `L` is a square, symmetric, sparse matrix of size `numnodes(G)`-by-`numnodes(G)`. The graph Laplacian matrix is undefined for graphs with self-loops.

See Also

`adjacency` | `graph` | `incidence`

Introduced in R2015b

maxflow

Maximum flow in graph

Syntax

```
mf = maxflow(G,s,t)
mf = maxflow(G,s,t,algorithm)
[mf,GF] = maxflow( ___ )
[mf,GF,cs,ct] = maxflow( ___ )
```

Description

`mf = maxflow(G,s,t)` returns the maximum flow on page 1-5315 between nodes `s` and `t`. If graph `G` is unweighted (that is, `G.Edges` does not contain the variable `Weight`), then `maxflow` treats all graph edges as having a weight equal to 1.

`mf = maxflow(G,s,t,algorithm)` specifies the maximum flow algorithm to use. This syntax is only available if `G` is a directed graph.

`[mf,GF] = maxflow( ___ )` also returns a directed graph object, `GF`, using any of the input arguments in previous syntaxes. `GF` is formed using only the edges in `G` that have nonzero flow values.

`[mf,GF,cs,ct] = maxflow( ___ )` additionally returns the source and target node IDs, `cs` and `ct`, representing the minimum cut on page 1-5316 associated with the maximum flow.

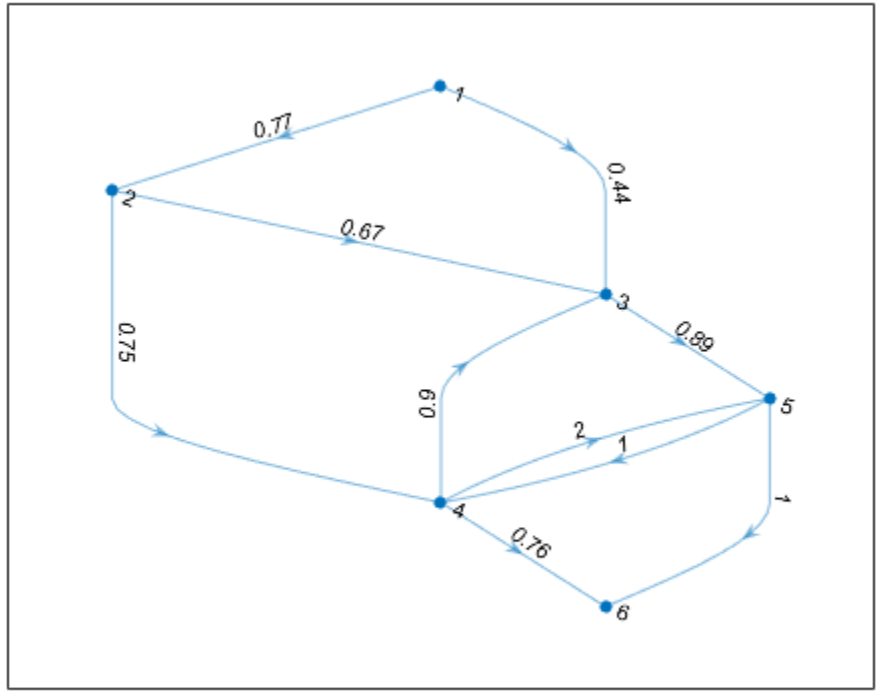
Examples

Maximum Flow in Graph

Create and plot a weighted graph. The weighted edges represent flow capacities.

```
s = [1 1 2 2 3 4 4 4 5 5];
t = [2 3 3 4 5 3 5 6 4 6];
```

```
weights = [0.77 0.44 0.67 0.75 0.89 0.90 2 0.76 1 1];
G = digraph(s,t,weights);
plot(G,'EdgeLabel',G.Edges.Weight,'Layout','layered');
```



Determine the maximum flow from node 1 to node 6.

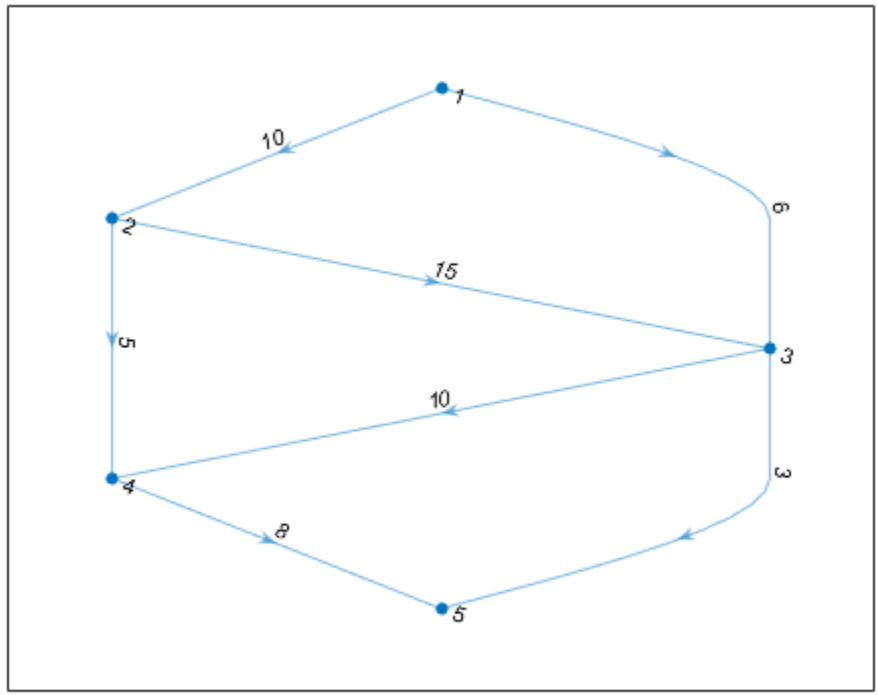
```
mf = maxflow(G,1,6)
```

```
mf = 1.2100
```

Maximum Flow with Specified Algorithm

Create and plot a graph. The weighted edges represent flow capacities.

```
s = [1 1 2 2 3 3 4];
t = [2 3 3 4 4 5 5];
weights = [10 6 15 5 10 3 8];
G = digraph(s,t,weights);
H = plot(G,'EdgeLabel',G.Edges.Weight);
```



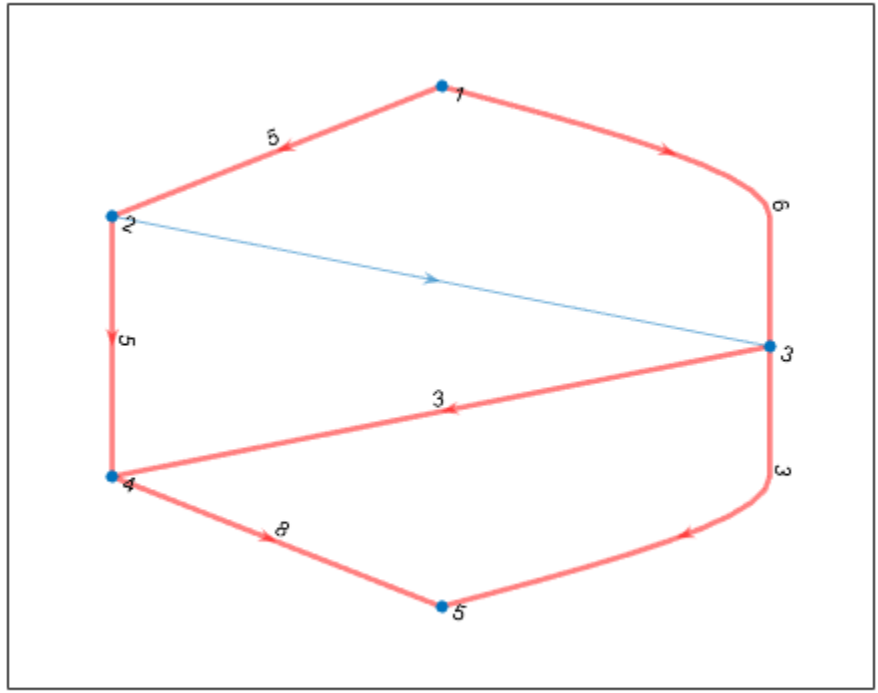
Find the maximum flow value between node 1 and node 5. Specify `'augmentpath'` to use the Ford-Fulkerson algorithm, and use two outputs to return a graph of the nonzero flows.

```
[mf,GF] = maxflow(G,1,5,'augmentpath')
```

```
mf = 11

GF = 
  digraph with properties:

    Edges: [6x2 table]
    Nodes: [5x0 table]
```

Highlight and label the graph of nonzero flows.

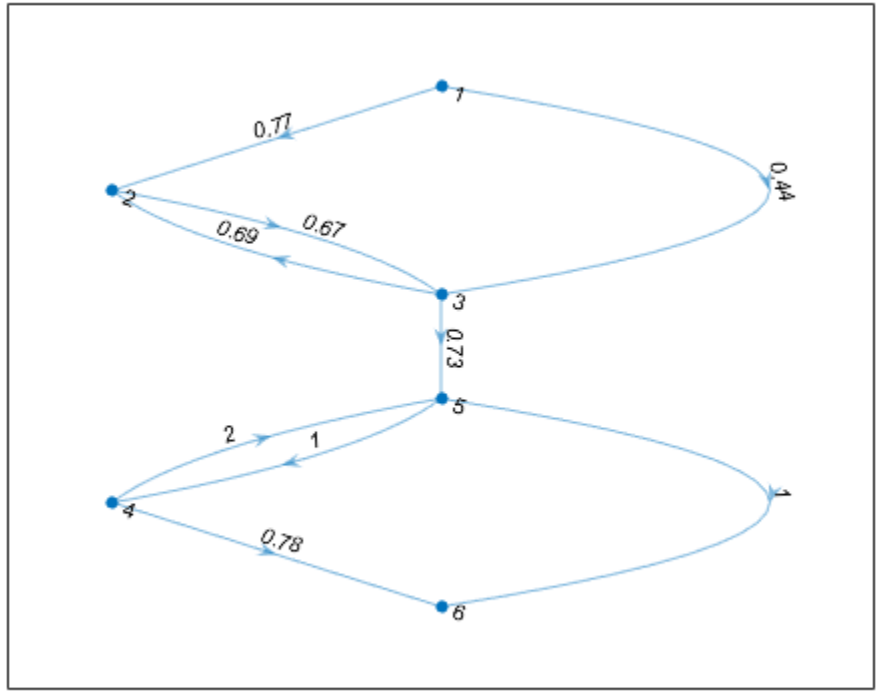
```
H.EdgeLabel = {};
highlight(H,GF,'EdgeColor','r','LineWidth',2);
st = GF.Edges.EndNodes;
labeledge(H,st(:,1),st(:,2),GF.Edges.Weight);
```



Minimum Cut Computation

Create and plot a weighted graph. The edge weights represent flow capacities.

```
s = [1 1 2 3 3 4 4 5 5];
t = [2 3 3 2 5 5 6 4 6];
weights = [0.77 0.44 0.67 0.69 0.73 2 0.78 1 1];
G = digraph(s,t,weights);
plot(G,'EdgeLabel',G.Edges.Weight,'Layout','layered')
```



Find the maximum flow and minimum cut of the graph.

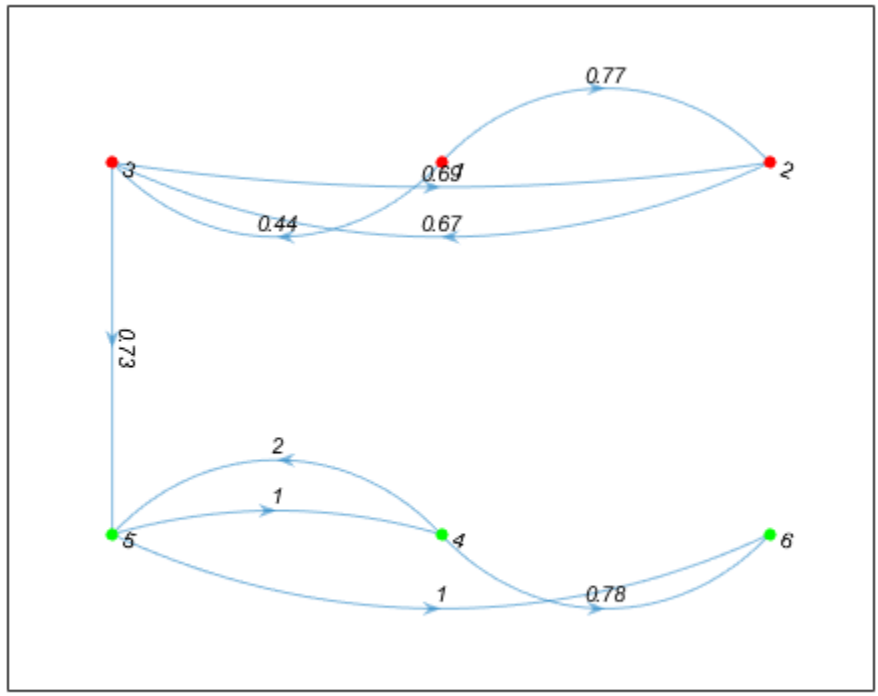
```
[mf,~,cs,ct] = maxflow(G,1,6)
```

```
mf = 0.7300

cs = 3×1

     1
     2
     3


ct = 3×1
```

```
4
5
6
```

Plot the minimum cut, using the `cs` nodes as sources and the `ct` nodes as sinks. Highlight the `cs` nodes as red and the `ct` nodes as green. Note that the weight of the edge that connects these two sets of nodes is equal to the maximum flow.

```
H = plot(G,'Layout','layered','Sources',cs,'Sinks',ct, ...
    'EdgeLabel',G.Edges.Weight);
highlight(H,cs,'NodeColor','red')
highlight(H,ct,'NodeColor','green')
```



Input Arguments

`G` — Input graph

`graph` object | `digraph` object

Input graph, specified as either a `graph` or `digraph` object. Use `graph` to create an undirected graph or `digraph` to create a directed graph.

Example: `G = graph(1,2)`

Example: `G = digraph([1 2],[2 3])`

`s,t` — Node pair (as separate arguments)

node indices | node names

Node pair, specified as separate arguments of node indices or node names to indicate the source node and target node. This table shows the different ways to refer to nodes either by their node indices or by their node names.

Value	Example
Scalar node index	`1`
Character vector node name	`'A'`
String scalar node name	`"A"`

Example: `mf = maxflow(G,'A','B')`

Example: `mf = maxflow(G,1,10)`

Data Types: `double` | `char` | `string`

`algorithm` — Maximum flow algorithm

`'searchtrees'` (default) | `'augmentpath'` | `'pushrelabel'`

Maximum flow algorithm, specified as one of the entries in the table.

Note You can only specify nondefault `algorithm` options with a directed graph.

Option	Description
`'searchtrees'` (default)	Uses the Boykov-Kolmogorov algorithm. Computes the maximum flow by constructing two search trees associated with nodes `s` and `t`.
`'augmentpath'`	Uses the Ford-Fulkerson algorithm. Computes the maximum flow iteratively by finding an augmenting path in a residual directed graph. The directed graph cannot have any parallel edges of opposite direction between the same two nodes, unless the weight of one of those edges is zero. So if the graph contains edge `[i j]`, then it can contain the reverse edge `[j i]` only if the weight of `[i j]` is zero and/or the weight of `[j i]` is zero.
`'pushrelabel'`	Computes the maximum flow by pushing a node's excess flow to its neighbors and then relabeling the node. The directed graph cannot have any parallel edges of opposite direction between the same two nodes, unless the weight of one of those edges is zero. So if the graph contains edge `[i j]`, then it can contain the reverse edge `[j i]` only if the weight of `[i j]` is zero and/or the weight of `[j i]` is zero.

Example: `mf = maxflow(G,'A','D','augmentpath')`

Output Arguments

`mf` — Maximum flow
scalar

Maximum flow, returned as a scalar.

`GF` — Directed graph of flows
`digraph` object

Directed graph of flows, returned as a `digraph` object. `GF` contains the same nodes as `G`, but only contains those edges of `G` that have a nonzero flow. For multigraphs with multiple edges between the same two nodes, `GF` contains a single edge reflecting the flow through the multiple edges.

`cs` — Minimum cut source node IDs
node indices | node names

Minimum cut source node IDs, returned as node indices or node names.

- If `s` and `t` specify numeric node indices, then `cs` and `ct` also contain node indices.
- If `s` and `t` specify node names, then `cs` and `ct` also contain node names.

`ct` — Minimum cut target node IDs
scalar | vector | character vector | cell array of character vectors

Minimum cut target node IDs, returned as node indices or node names.

- If `s` and `t` specify numeric node indices, then `cs` and `ct` also contain node indices.
- If `s` and `t` specify node names, then `cs` and `ct` also contain node names.

More About

Maximum Flow

In the context of maximum flow, the edges in a graph are considered to have a *capacity* as represented by the edge weight. The capacity of an edge is the amount of flow that can pass through that edge. Therefore, the maximum flow between two nodes in a graph maximizes the amount of flow passing from the source node, `s`, to the target node, `t`, based on the capacities of the connecting edges.

Minimum Cut

A minimum cut partitions the directed graph nodes into two sets, `cs` and `ct`, such that the sum of the weights of all edges connecting `cs` and `ct` (weight of the cut) is minimized. The weight of the minimum cut is equal to the maximum flow value, `mf`.

The entries in `cs` and `ct` indicate the nodes of `G` associated with nodes `s` and `t`, respectively. `cs` and `ct` satisfy `numel(cs) + numel(ct) = numnodes(G)`.

See Also

`digraph` | `graph`

Introduced in R2015b

minspantree

Minimum spanning tree of graph

Syntax

```
T = minspantree(G)
T = minspantree(G,Name,Value)
[T,pred] = minspantree( ___ )
```

Description

`T = minspantree(G)` returns the minimum spanning tree on page 1-5324, `T`, for graph `G`.

`T = minspantree(G,Name,Value)` uses additional options specified by one or more Name-Value pair arguments. For example, `minspantree(G,'Method','sparse')` uses Kruskal's algorithm for calculating the minimum spanning tree.

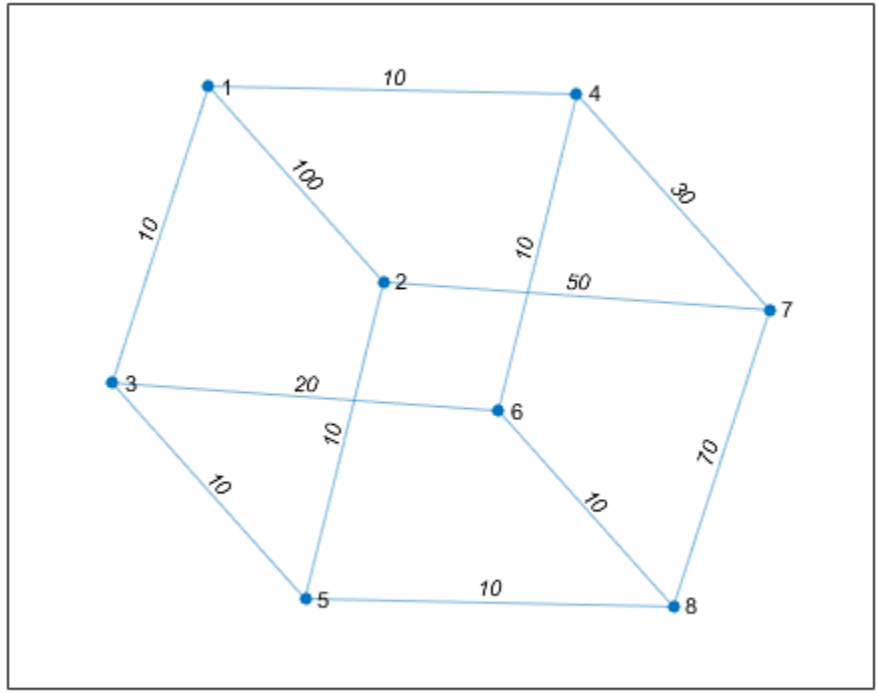
`[T,pred] = minspantree( ___ )` also returns a vector of predecessor nodes, `pred`, using any of the input arguments in previous syntaxes.

Examples

Minimum Spanning Tree of Cube Graph

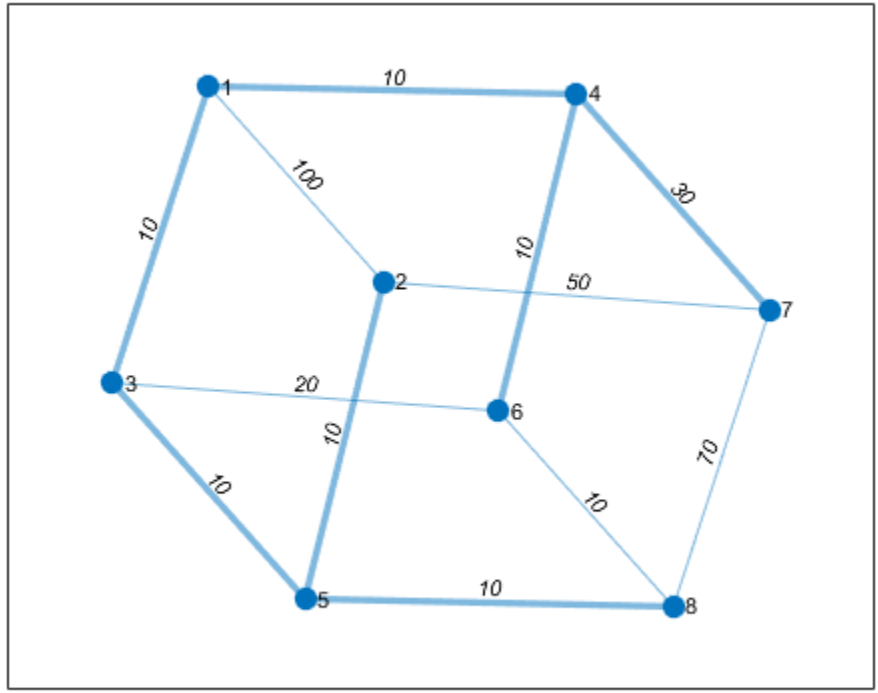
Create and plot a cube graph with weighted edges.

```
s = [1 1 1 2 5 3 6 4 7 8 8 8];
t = [2 3 4 5 3 6 4 7 2 6 7 5];
weights = [100 10 10 10 10 20 10 30 50 10 70 10];
G = graph(s,t,weights);
p = plot(G,'EdgeLabel',G.Edges.Weight);
```



Calculate and plot the minimum spanning tree of the graph on top of the graph. `T` contains the same nodes as `G`, but a subset of the edges.

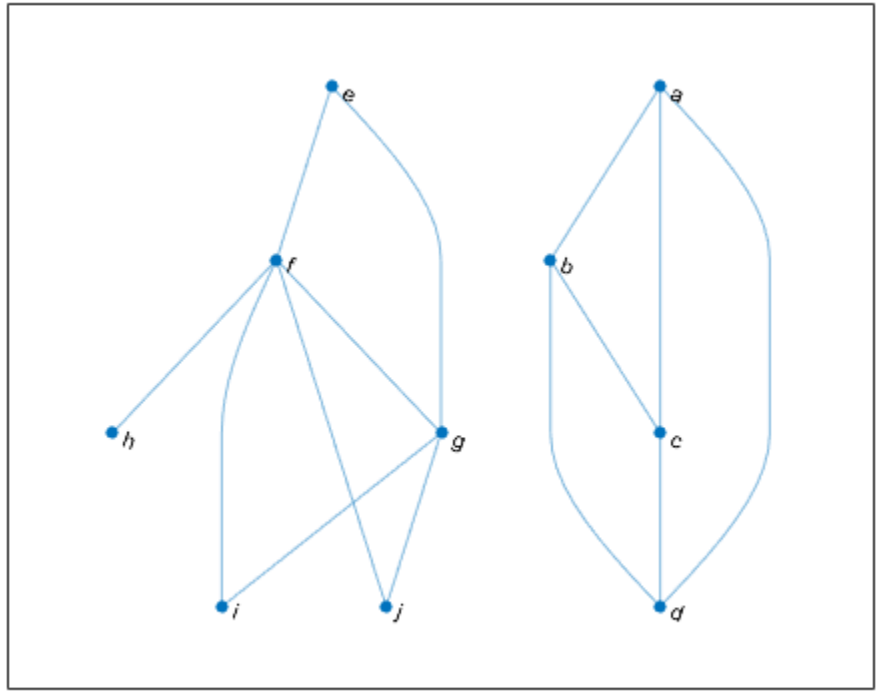
```
[T,pred] = minspantree(G);
highlight(p,T)
```



Minimum Spanning Forest from Specified Root Node

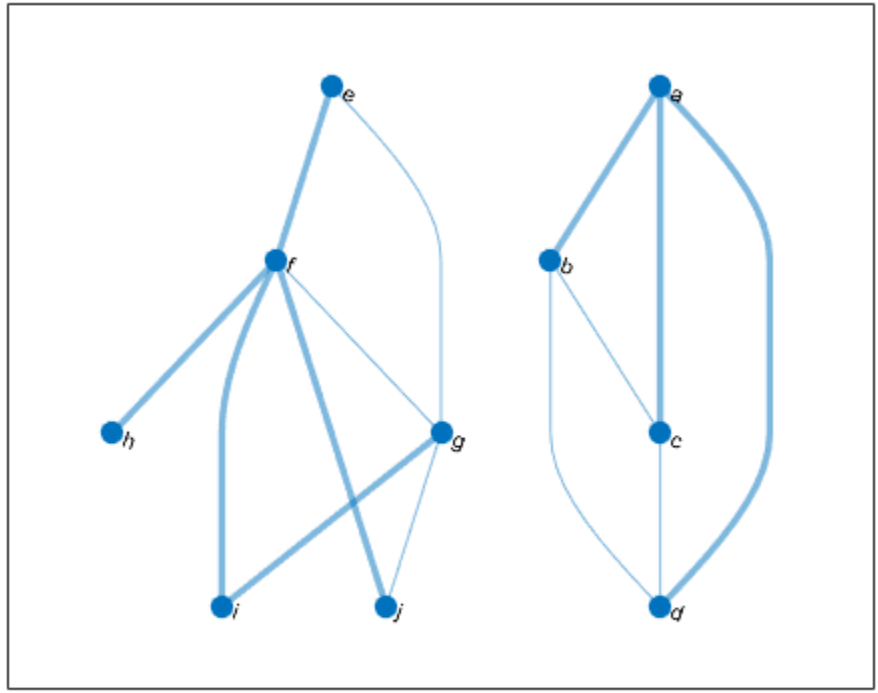
Create and plot a graph that has multiple components.

```
s = {'a' 'a' 'a' 'b' 'b' 'c' 'e' 'e' 'f' 'f' 'f' 'f' 'g' 'g'};
t = {'b' 'c' 'd' 'c' 'd' 'd' 'f' 'g' 'g' 'h' 'i' 'j' 'i' 'j'};
G = graph(s,t);
p = plot(G,'Layout','layered');
```



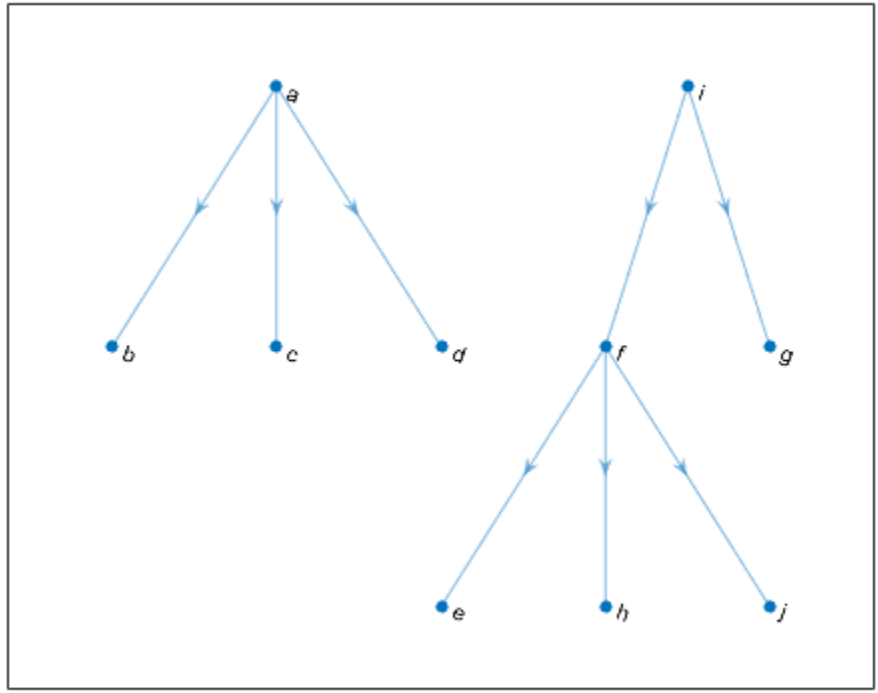
Find the minimum spanning forest for the graph, starting at node `i`. Highlight the resulting forest in the plot. The graph node names are carried over into the minimum spanning tree graph.

```
[T,pred] = minspantree(G,'Type','forest','Root',findnode(G,'i'));
highlight(p,T)
```



Use the vector of predecessor nodes, `pred`, to create a directed version of the minimum spanning forest. All of the edges in this tree are directed away from the root nodes in each component (nodes `i` and `a`).

```
rootedTree = digraph(pred(pred~=0),find(pred~=0),[],G.Nodes.Name);
plot(rootedTree)
```



Input Arguments

G — Input graph
graph object

Input graph, specified as a graph object. Use graph to create an undirected graph object.

Example: G = graph(1,2)

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[T,pred] = minspantree(G,'Method','sparse')`

`Method` — Minimum spanning tree algorithm
`'dense'` (default) | `'sparse'`

Minimum spanning tree algorithm, specified as the comma-separated pair consisting of `'Method'` and one of the options in the table.

Option	Description
`'dense'` (default)	Prim's algorithm. This algorithm starts at the root node and adds edges to the tree while traversing the graph.
`'sparse'`	Kruskal's algorithm. This algorithm sorts all of the edges by weight, and then adds them to the tree if they do not create a cycle.

`Root` — Root node
1 (default) | node index | node name

Root node, specified as the comma-separated pair consisting of `'Root'` and a node index or node name. The default root node is `1`.

- If `'Method'` is `'dense'` (default), then the root node is the starting node.
- If `'Method'` is `'sparse'`, then the root node is used only to compute `pred`, the vector of predecessor nodes.

You can specify the root node in any of these formats:

Value	Example
Scalar node index	`1`
Character vector node name	`'A'`
String scalar node name	`"A"`

Type — Type of minimum spanning tree
`'tree'` (default) | `'forest'`

Type of minimum spanning tree, specified as the comma-separated pair consisting of `'Type'` and one of the options in the table.

Option	Description
`'tree'`	Only a single tree is returned. The tree contains the root node.
`'forest'`	A forest of minimum spanning trees is returned. In other words, specify `'forest'` to calculate the minimum spanning tree of all connected components in the graph.

Output Arguments

T — Minimum spanning tree
`graph` object

Minimum spanning tree, returned as a `graph` object.

pred — Predecessor nodes
vector

Predecessor nodes, returned as a vector of node indices. `pred(I)` is the node index of the predecessor of node `I`. By convention, `pred(rootNode) = 0`. If `Type` is `'tree'`, then `pred(I) = NaN` for all nodes `I` that are not in the same component as the root node.

`pred` specifies a directed version of the minimum spanning tree, with all edges directed away from the root node.

More About

Minimum Spanning Tree

For connected graphs, a spanning tree is a subgraph that connects every node in the graph, but contains no cycles. There can be many spanning trees for any given graph. By

assigning a weight to each edge, the different spanning trees are assigned a number for the total weight of their edges. The minimum spanning tree is then the spanning tree whose edges have the least total weight.

For graphs with equal edge weights, all spanning trees are minimum spanning trees, since traversing `n` nodes requires `n-1` edges.

See Also

`conncomp` | `graph` | `shortestpath`

Introduced in R2015b

nearest

Nearest neighbors within radius

Syntax

```
nodeIDs = nearest(G,s,d)
nodeIDs = nearest(G,s,d,Name,Value)
[nodeIDs,dist] = nearest( ___ )
```

Description

`nodeIDs = nearest(G,s,d)` returns all nodes in graph `G` that are within distance `d` from node `s`. If the graph is weighted (that is, if `G.Edges` contains a variable `Weight`), then those weights are used as the distances along the edges in the graph. Otherwise, all edge distances are taken to be `1`.

`nodeIDs = nearest(G,s,d,Name,Value)` uses additional options specified by one or more name-value pair arguments. For example, if `G` is a weighted graph, then `nearest(G,s,d,'Method','unweighted')` ignores the edge weights in graph `G` and instead treats all edge weights as `1`.

`[nodeIDs,dist] = nearest( ___ )` additionally returns the distance to each of the nearest neighbors, such that `dist(j)` is the distance from source node `s` to the node `nodeIDs(j)`. You can use any of the input argument combinations in previous syntaxes.

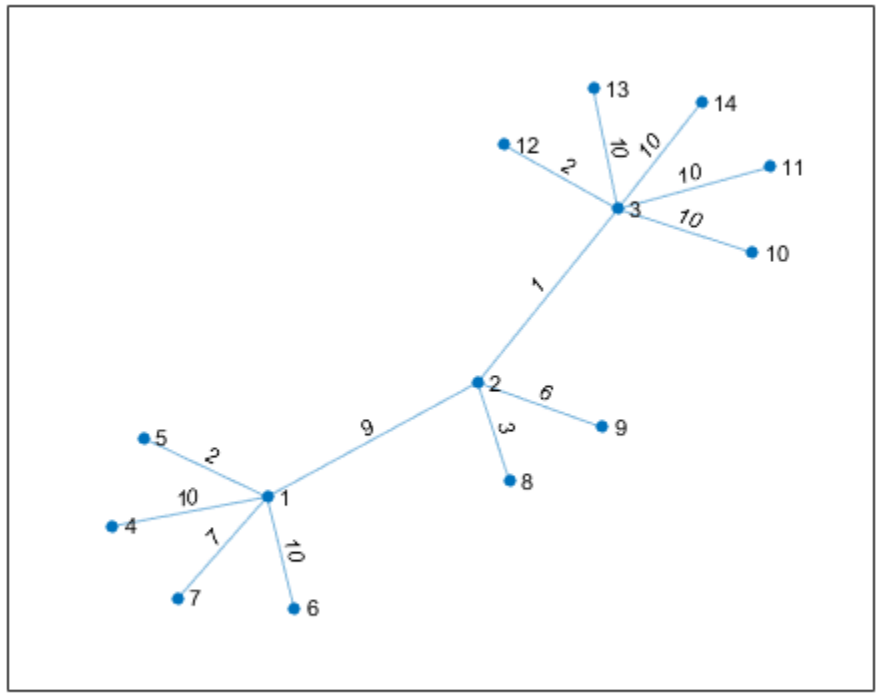
Examples

Nearest Nodes

Create and plot a graph with weighted edges.

```
s = [1 1 1 1 1 2 2 2 3 3 3 3 3];
t = [2 4 5 6 7 3 8 9 10 11 12 13 14];
weights = randi([1 10],1,13);
```

```
G = graph(s,t,weights);
p = plot(G,'Layout','force','EdgeLabel',G.Edges.Weight);
```



Determine which nodes are within a radius of 15 from node 1.

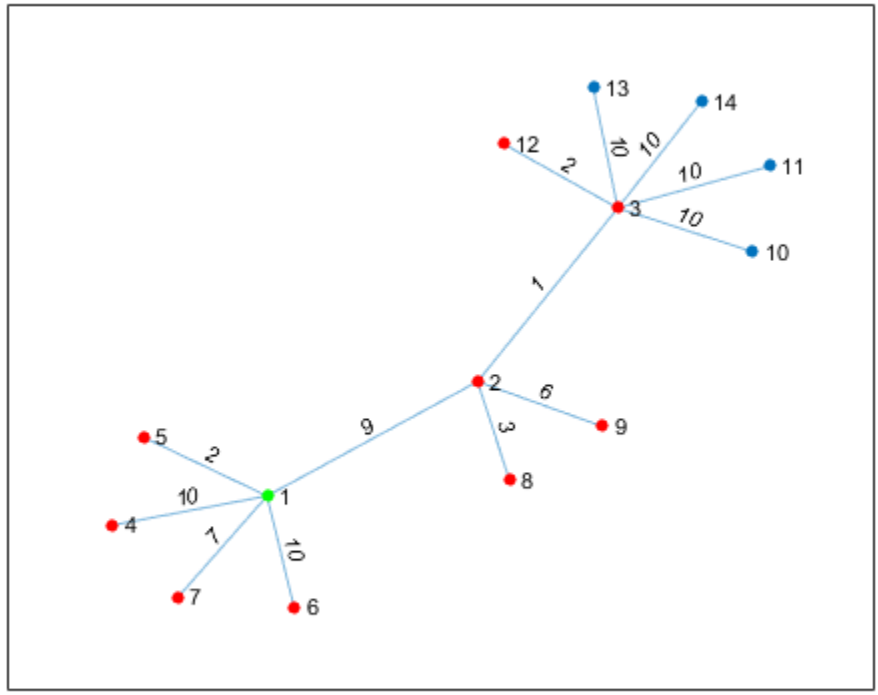
```
nn = nearest(G,1,15)
```

```
nn = 9×1

     5
     7
     2
     3
     4
     6
```

```
     8
    12
     9
```

Highlight the source node as green and the nearest neighbors as red.

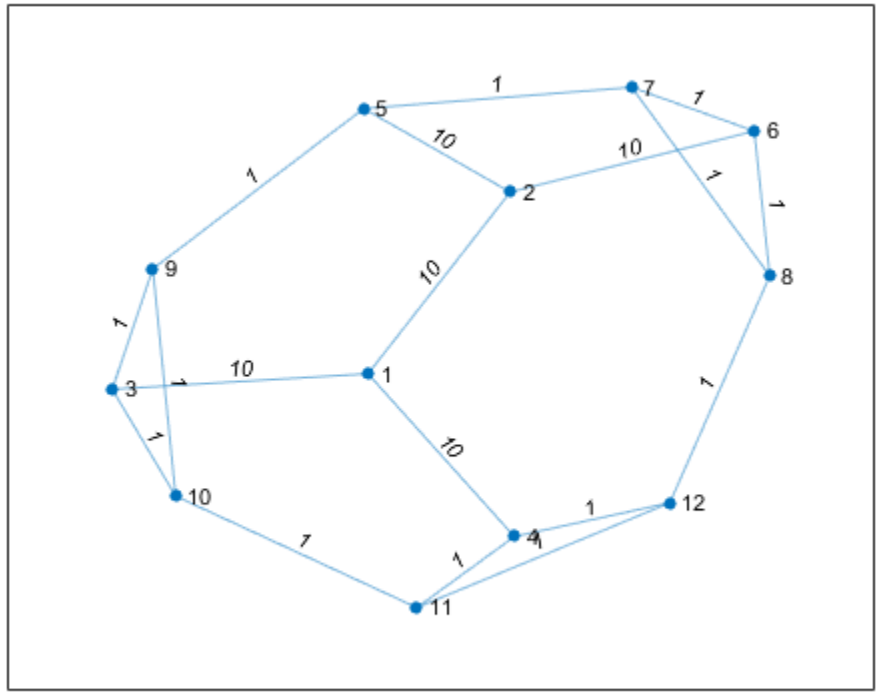
```
highlight(p,1,'NodeColor','g')
highlight(p,nn,'NodeColor','r')
```



Distances of Nearest Nodes

Create and plot a graph with weighted edges.

```
s = [1 1 1 2 2 6 6 7 7 3 3 9 9 4 4 11 11 8];
t = [2 3 4 5 6 7 8 5 8 9 10 5 10 11 12 10 12 12];
weights = [10 10 10 10 10 1 1 1 1 1 1 1 1 1 1 1 1 1];
G = graph(s,t,weights);
plot(G,'EdgeLabel',G.Edges.Weight)
```



Determine which nodes are within a radius of 5 from node 3, and return the distance to each node.

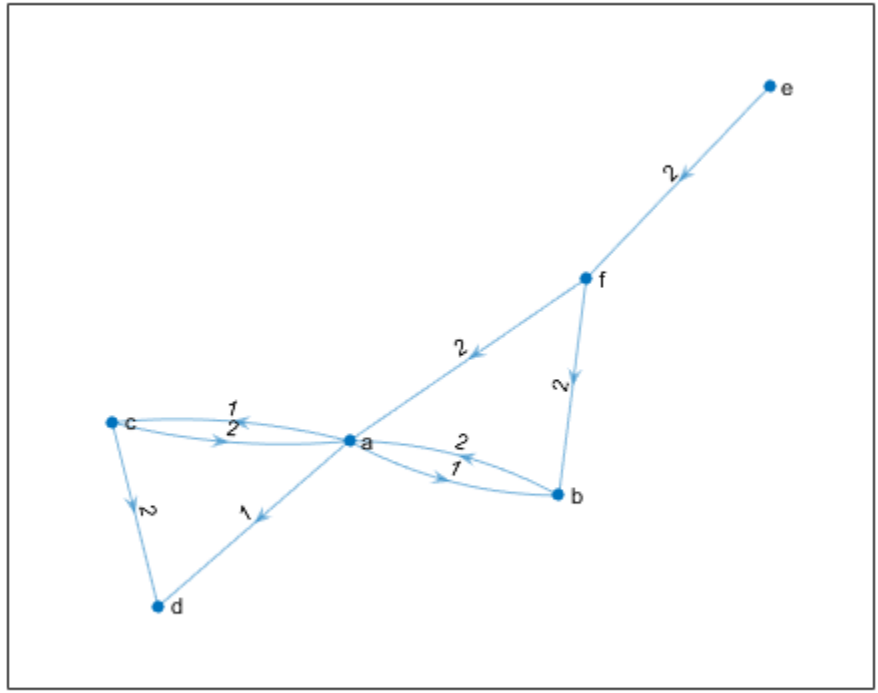
```
[nn,dist] = nearest(G,3,5)
```

```
nn = 9×1

     9
    10
     5
    11
     4
     7
    12
     6
     8


dist = 9×1

     1
     1
     2
     2
     3
     3
     3
     4
     4
```

Incoming Neighbor Distances in Directed Graph

Create and plot a directed graph with weighted edges.

```
s = {'a' 'a' 'a' 'b' 'c' 'c' 'e' 'f' 'f'};
t = {'b' 'c' 'd' 'a' 'a' 'd' 'f' 'a' 'b'};
weights = [1 1 1 2 2 2 2 2 2];
G = digraph(s,t,weights);
plot(G,'EdgeLabel',G.Edges.Weight)
```



Determine the nearest nodes within a radius of 1 from node `'a'`, measured by outgoing path distance from node `'a'`.

```
nn_out = nearest(G,'a',1)
```

```
nn_out = 3x1 cell array
    {'b'}
    {'c'}
    {'d'}
```

Determine all of the nodes that have incoming paths leading to node `'a'` by specifying the radius as `Inf`.

```
nn_in = nearest(G,'a',Inf,'Direction','incoming')
```

```
nn_in = 4x1 cell array
    {'b'}
    {'c'}
    {'f'}
    {'e'}
```

Input Arguments

G — Input graph
graph object | digraph object

Input graph, specified as either a graph or digraph object. Use graph to create an undirected graph or digraph to create a directed graph.

Example: G = graph(1,2)

Example: G = digraph([1 2],[2 3])

s — Source node
node index | node name

Source node, specified as a node index or a node name in one of the forms in this table.

Value	Example
Scalar node index	1
Character vector node name	'A'
String scalar node name	"A"

Example: nearest(G,3,1)

Example: nearest(G,'a',5)

d — Neighbor distance radius
scalar

Neighbor distance radius, specified as a numeric scalar.

Example: nearest(G,3,1)

Example: nearest(G,'a',2.5)

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[nodeIDs,dist] = nearest(G,s,5,'Method','unweighted','Direction','incoming')`

`Direction` — Direction of distance measurement

`'outgoing'` (default) | `'incoming'`

Note The `'Direction'` option can only be specified with directed graphs.

Direction of distance measurement, specified as the comma-separated pair consisting of `'Direction'` and one of the options in this table.

Option	Description
`'outgoing'` (default)	Distances are calculated using paths going *out* from source node `s`.
`'incoming'`	Distances are calculated using paths coming *in* to source node `s`.

Example: `nearest(G,s,d,'Direction','incoming')`

`Method` — Shortest path algorithm

`'auto'` (default) | `'unweighted'` | `'positive'` | `'mixed'`

Shortest path algorithm, specified as the comma-separated pair consisting of `'Method'` and one of the options in this table.

Option	Description
`'auto'` (default)	The `'auto'` option automatically selects the algorithm: • `'unweighted'` is used for `graph` and `digraph` inputs with no edge weights. • `'positive'` is used for all `graph` inputs that have edge weights, and requires the weights to be nonnegative. This option is also used for `digraph` inputs with nonnegative edge weights. • `'mixed'` is used for `digraph` inputs whose edge weights contain some negative values. The graph cannot have negative cycles.
`'unweighted'`	Breadth-first computation that treats all edge weights as `1`.
`'positive'`	Dijkstra algorithm that requires all edge weights to be nonnegative.
`'mixed'` (only for `digraph`)	Bellman-Ford algorithm for directed graphs that requires the graph to have no negative cycles. While `'mixed'` is slower than `'positive'` for the same problem, `'mixed'` is more versatile as it allows some edge weights to be negative.

Note For most graphs, `'unweighted'` is the fastest algorithm, followed by `'positive'` and `'mixed'`.

Example: `nearest(G,s,d,'Method','positive')`

Output Arguments

`nodeIDs` — Nearest neighbor node IDs
node indices | node names

Nearest neighbor node IDs, returned as node indices if `s` is numeric, or as node names if `s` is a node name. The nodes are sorted from nearest to furthest. `nodeIDs` is empty if no nodes are within the specified distance. `nodeIDs` never contains the source node `s` even if the graph has self-loops.

Use `H = subgraph(G,[s; nodeIDs])` to extract a subgraph of the nearest neighbors from the original graph `G`.

`dist` — Neighbor distances
vector

Neighbor distances, returned as a vector. `dist(j)` is the distance from source node `s` to neighboring node `nodeIDs(j)`.

See Also

`distances` | `neighbors` | `predecessors` | `shortestpath` | `shortestpathtree` | `successors`

Topics

`graph`
`digraph`

Introduced in R2016a

neighbors

Neighbors of graph node

Syntax

```
N = neighbors(G,nodeID)
```

Description

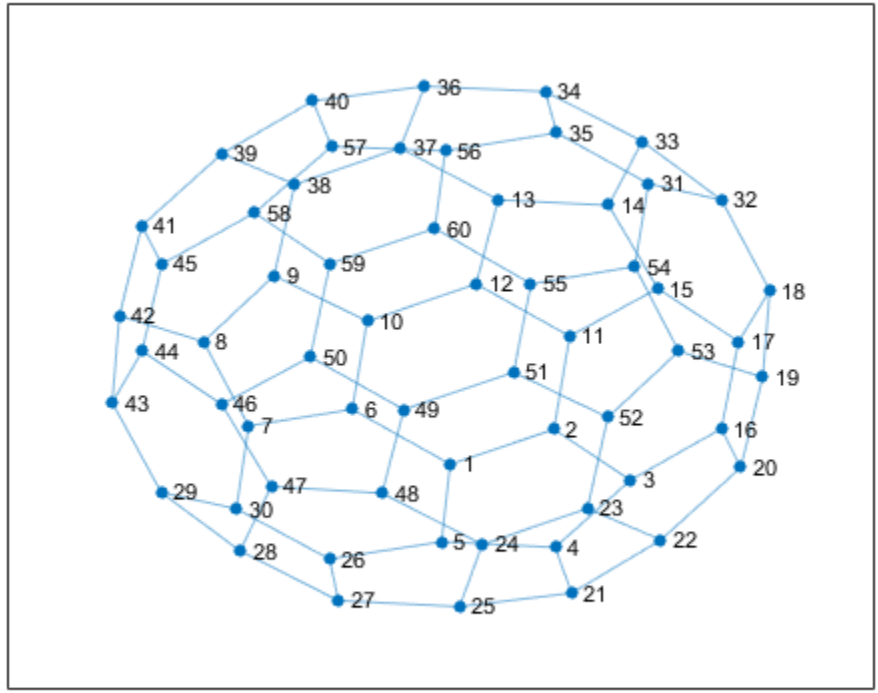
`N = neighbors(G,nodeID)` returns the node IDs of all nodes connected by an edge to the node specified by `nodeID`.

Examples

Neighboring Graph Nodes

Create and plot a graph, and then determine the neighbors of node `10`.

```
G = graph(bucky);
plot(G)
```




```
N = neighbors(G,10)
```

```
N = 3×1

     6
     9
    12
```

Input Arguments

G — Input graph
graph object

Input graph, specified as a graph object. Use graph to create an undirected graph object.

Example: G = graph(1,2)

nodeID — Node identifier
node index | node name

Node identifier, specified as one of the values in this table.

Value	Example
Scalar node index	1
Character vector node name	'A'
String scalar node name	"A"

Example: N = neighbors(G,3)

Example: N = neighbors(G,'A')

Output Arguments

N — Neighboring nodes
node indices | node names

Neighboring nodes, returned as node indices if nodeID is numeric, or as node names if nodeID is a node name. A node that is connected to itself by an edge (a self-loop) is listed as its own neighbor only once.

Compatibility Considerations

Self-loop counting change

Behavior changed in R2018a

`neighbors` counts self-loops only once. In previous releases, if node `u` had a self-loop, then `neighbors(g,u)` listed `u` twice in the output. `neighbors(g,u)` now returns only one instance of `u`.

See Also

`degree` | `graph` | `nearest` | `outedges`

Introduced in R2015b

numedges

Number of edges in graph

Syntax

```
N = numedges(G)
```

Description

`N = numedges(G)` returns the number of edges in graph `G`.

Examples

Number of Graph Edges

Use the `bucky` adjacency matrix to create a directed graph, and then determine how many edges the graph contains.

```
G = digraph(bucky)
```

```
G = 
  digraph with properties:

    Edges: [180x2 table]
    Nodes: [60x0 table]
```

```
N = numedges(G)
```

```
N = 180
```

Input Arguments

G — Input graph
graph object | digraph object

Input graph, specified as either a graph or digraph object. Use graph to create an undirected graph or digraph to create a directed graph.

Example: G = graph(1,2)

Example: G = digraph([1 2],[2 3])

Output Arguments

N — Number of edges
scalar

Number of edges, returned as a scalar.

See Also

addedge | digraph | graph | numnodes | rmedge

Introduced in R2015b

numnodes

Number of nodes in graph

Syntax

```
N = numnodes(G)
```

Description

`N = numnodes(G)` returns the number of nodes in graph `G`.

Examples

Number of Graph Nodes

Use the `bucky` adjacency matrix to create a directed graph, and then determine how many nodes the graph contains.

```
G = digraph(bucky)
```

```
G = 
  digraph with properties:

    Edges: [180x2 table]
    Nodes: [60x0 table]
```

```
N = numnodes(G)
```

```
N = 60
```

Input Arguments

G — Input graph

graph object | digraph object

Input graph, specified as either a graph or digraph object. Use graph to create an undirected graph or digraph to create a directed graph.

Example: G = graph(1,2)

Example: G = digraph([1 2],[2 3])

Output Arguments

N — Number of nodes

scalar

Number of nodes, returned as a scalar.

See Also

addnode | digraph | graph | numedges | rmnode

Introduced in R2015b

outedges

Outgoing edges from node

Syntax

```
eid = outedges(G,nodeID)
[eid,nid] = outedges(G,nodeID)
```

Description

`eid = outedges(G,nodeID)` returns the indices of all outgoing edges from node `nodeID` in graph `G`.

`[eid,nid] = outedges(G,nodeID)` additionally returns the successor nodes `nid` that are connected to `nodeID` by the edges `eid`.

Examples

Outgoing Edges of Selected Node

Create an undirected multigraph with three nodes and four edges. Find the outgoing edges of node 2.

```
G = graph([1 1 1 2],[2 2 3 3]);
G.Edges
```

```
ans=4×1 table
    EndNodes
    ________

     1    2
     1    2
     1    3
     2    3
```

```
eid = outedges(G,2)
```

```
eid = 3×1

     1
     2
     4
```

The vector `eid` contains indices to rows in the `G.Edges` table. Use the vector to index into `G.Edges`.

```
G.Edges(eid,:)
```

```
ans=3×1 table
    EndNodes
    ________

     1    2
     1    2
     2    3
```

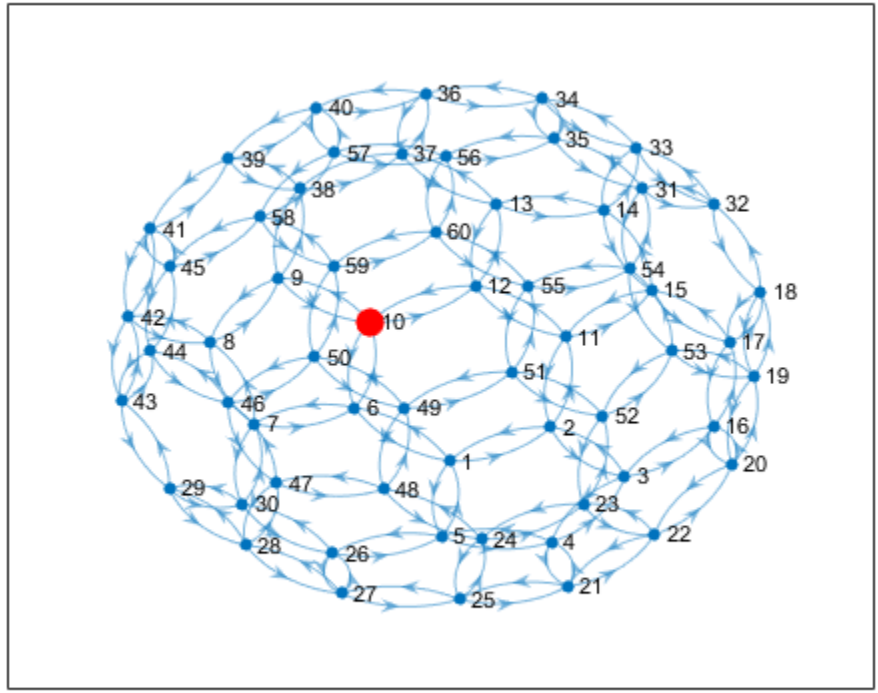
For undirected graphs, the edges (1,2) and (2,1) are the same.

Find Outgoing Edges and Node Successors

Plot a graph and highlight the outgoing edges and successors of a selected node.

Create and plot a directed graph using the `bucky` adjacency matrix. Highlight node 10 for reference.

```
G = digraph(bucky);
p = plot(G);
highlight(p,10,'NodeColor','r','MarkerSize',10)
```



Determine the outgoing edges and successors of node 10. Highlight these nodes and edges.

```
[eid,nid] = outedges(G,10)
```

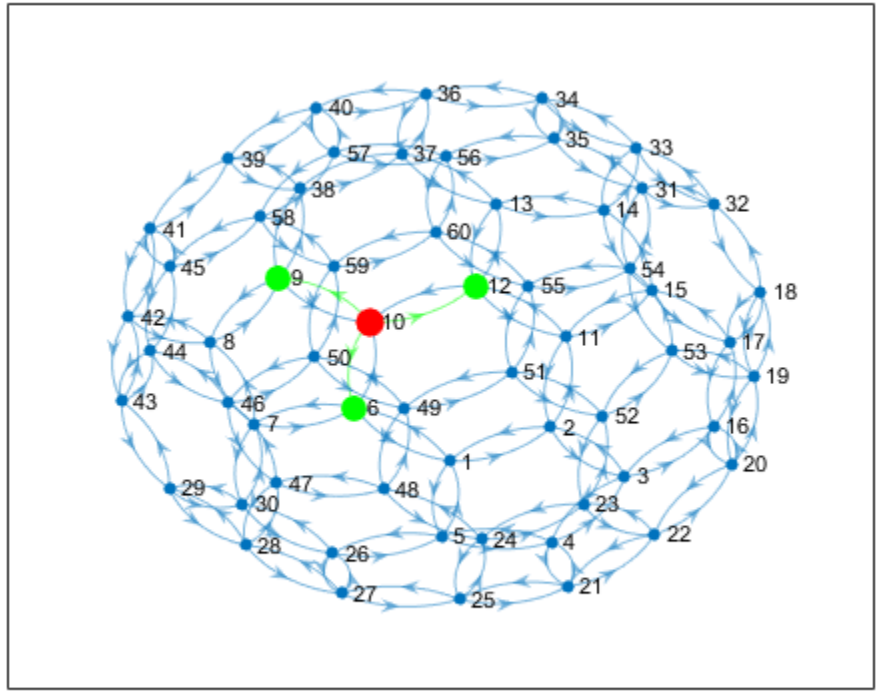
```
eid = 3×1

    28
    29
    30


nid = 3×1
```

```
     6
     9
    12
```

```
X = G.Edges(eid,:)
```

```
X=3×2 table
    EndNodes    Weight
    ________    ______

    10     6      1
    10     9      1
    10    12      1
```

```
highlight(p,nid,'NodeColor','g','MarkerSize',9)
highlight(p,'Edges',eid,'EdgeColor','g')
```



Input Arguments

G — Input graph

graph object | digraph object

Input graph, specified as either a `graph` or `digraph` object. Use `graph` to create an undirected graph or `digraph` to create a directed graph.

Example: `G = graph(1,2)`

Example: `G = digraph([1 2],[2 3])`

`nodeID` — Node identifier
node index | node name

Node identifier, specified as one of the values in this table.

Value	Example
Scalar node index	`1`
Character vector node name	`'A'`
String scalar node name	`"A"`

Example: `outedges(G,1)`

Example: `outedges(G,'A')`

Output Arguments

`eid` — Edge indices
column vector

Edge indices, returned as a column vector. You can use the edge indices to index into the edges table of the graph with `G.Edges(eid,:)`.

`nid` — Node IDs of successors
node indices | node names

Node IDs of successors, returned as node indices if `nodeID` is numeric, or as node names if `nodeID` is a node name. Use `findnode(G,nid)` to convert node names into node indices. You can use node indices to index into the nodes table of the graph with `G.Nodes(nid,:)`.

The node IDs in `nid` are the same as those returned by the `successors` function. However, if there are multiple outgoing edges to the same node, this node is listed more than once in `nid`.

Tips

- By convention, for undirected graphs, all edges incident to a node are considered to be outgoing edges. Use `inedges` to find incoming edges in a directed graph.

- For graphs with multiple edges, `outedges` and `successors` can return arrays of different lengths, since there can be multiple outgoing edges to some of the successors.

See Also

`digraph` | `graph` | `inedges` | `successors`

Introduced in R2018a

highlight

Package: `matlab.graphics.chart.primitive`

Highlight nodes and edges in plotted graph

Syntax

```
highlight(H,nodeIDs)
highlight(H,G)
highlight(H,s,t)
highlight( ___ ,Name,Value)
```

Description

`highlight(H,nodeIDs)` highlights the nodes specified by `nodeIDs` by increasing the sizes of their markers.

`highlight(H,G)` highlights the nodes and edges of graph `G` by increasing their node marker size and edge line width, respectively. `G` must have the same nodes and a subset of the edges of the underlying graph of `H`. If `G` contains repeated edges, then they are all highlighted. Isolated nodes with degree 0 are not highlighted.

`highlight(H,s,t)` highlights all edges between the specified source and target node pairs in `s` and `t` by increasing their edge line widths. If the edge indices are available instead of the node pairs (s,t), use `highlight(H,'Edges',idx)` instead.

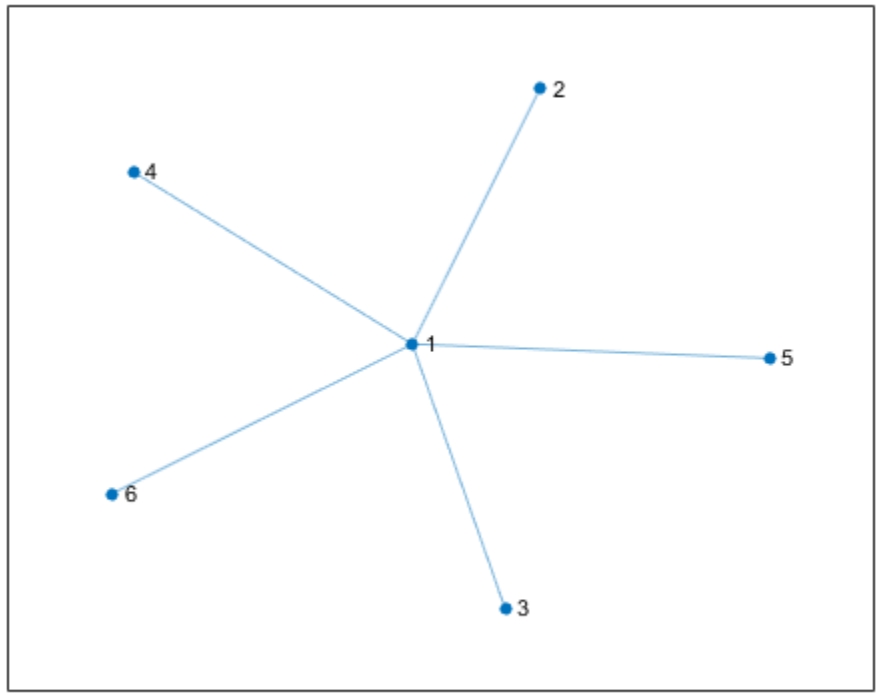
`highlight( ___ ,Name,Value)` uses additional options specified by one or more Name-Value pair arguments using any of the input argument combinations in previous syntaxes. For example, `highlight(H,nodes,'NodeColor','g')` highlights a subset of nodes by changing their color to green, instead of increasing their marker size.

Examples

Highlight Graph Nodes

Create and plot a graph. Return a handle to the `GraphPlot` object, h.

```
s = 1;
t = 2:6;
G = graph(s,t);
h = plot(G,'Layout','force')
```

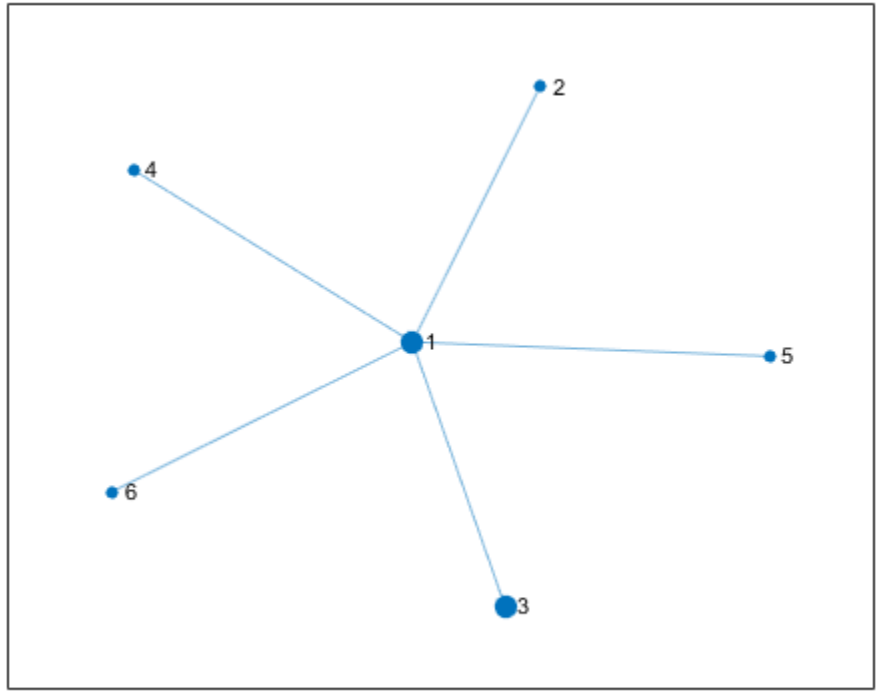



```
h =
  GraphPlot with properties:

     NodeColor: [0 0.4470 0.7410]
    MarkerSize: 4
```

```
       Marker: 'o'
    EdgeColor: [0 0.4470 0.7410]
    LineWidth: 0.5000
    LineStyle: '-'
    NodeLabel: {'1'  '2'  '3'  '4'  '5'  '6'}
    EdgeLabel: {}
        XData: [3.8317e-04 0.6403 0.4648 -1.3929 1.7883 -1.5009]
        YData: [9.6820e-04 1.6734 -1.7296 1.1251 -0.0922 -0.9777]
        ZData: [0 0 0 0 0 0]

  Show all properties
```

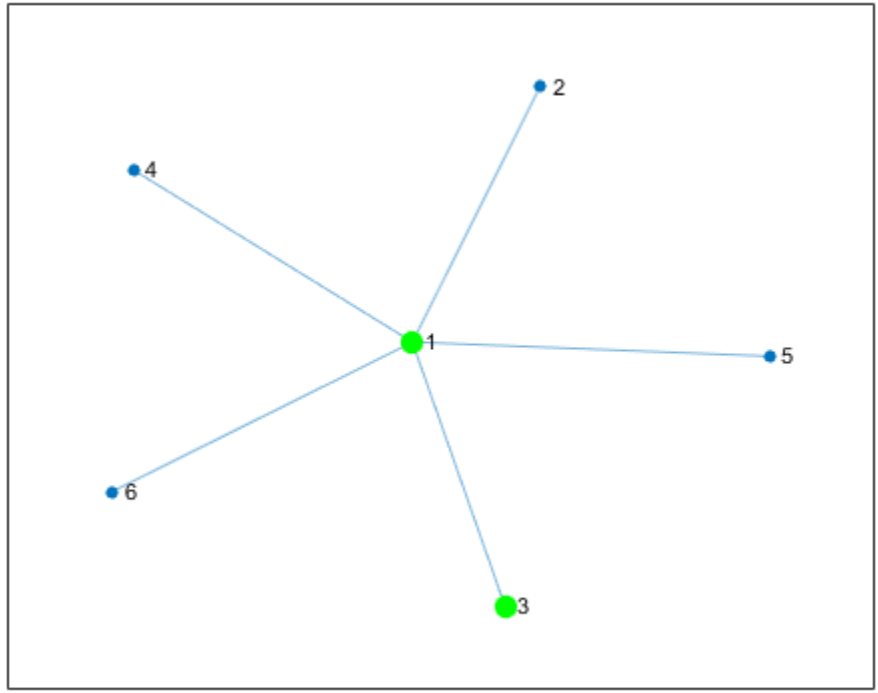
Highlight nodes 1 and 3 by increasing their marker size.

```
highlight(h,[1 3])
```



Highlight nodes 1 and 3 by changing their color.

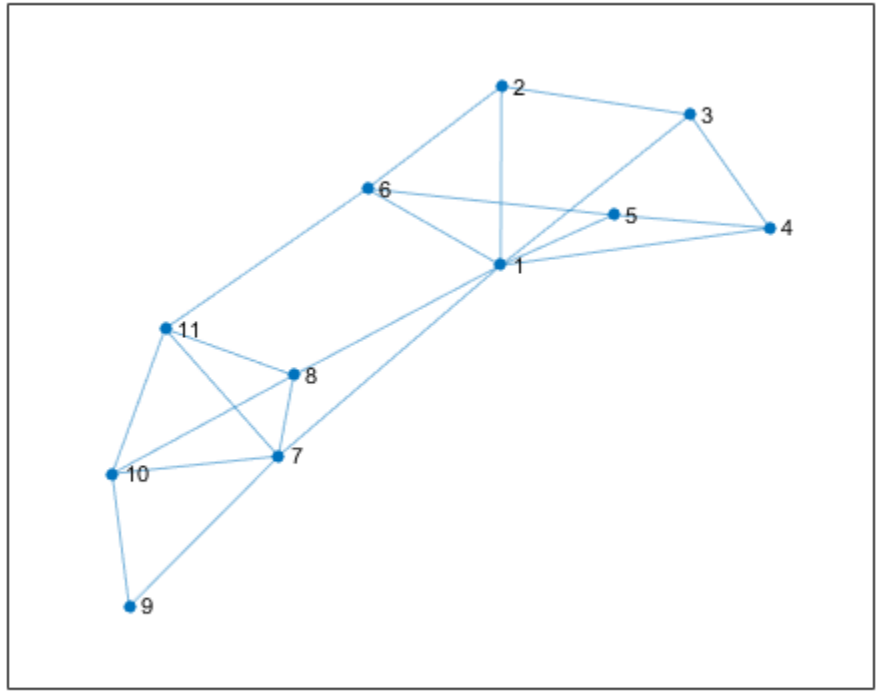
```
highlight(h,[1 3],'NodeColor','g')
```



Highlight Minimum Spanning Tree of Graph

Create and plot a graph. Return a handle to the `GraphPlot` object, `h`.

```
s = [1 1 1 1 1 1 2 3 4 5 6 7 7 7 7 8 9 10 11 8 6];
t = [2 3 4 5 6 7 3 4 5 6 2 8 9 10 11 10 10 11 8 1 11];
G = graph(s,t);
h = plot(G)
```

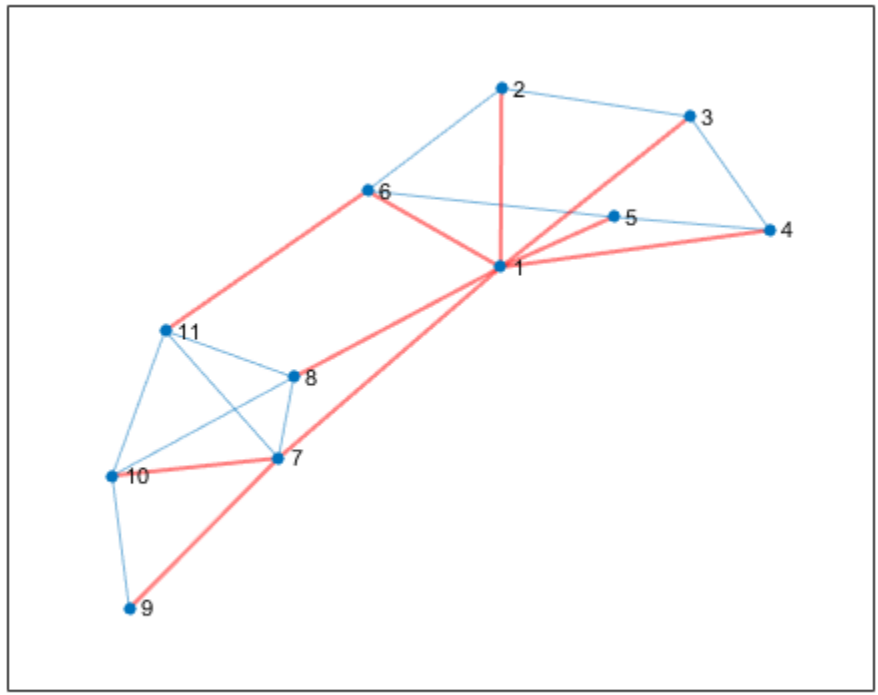



```
h =
  GraphPlot with properties:

     NodeColor: [0 0.4470 0.7410]
    MarkerSize: 4
        Marker: 'o'
     EdgeColor: [0 0.4470 0.7410]
     LineWidth: 0.5000
     LineStyle: '-'
     NodeLabel: {'1'  '2'  '3'  '4'  '5'  '6'  '7'  '8'  '9'  '10'  '11'}
     EdgeLabel: {}
         XData: [1x11 double]
         YData: [1x11 double]
         ZData: [0 0 0 0 0 0 0 0 0 0 0]
```

Show all properties

Calculate the minimum spanning tree of the graph. Highlight the minimum spanning tree subgraph in the plot by increasing the line width and changing the color of the edges in the tree.

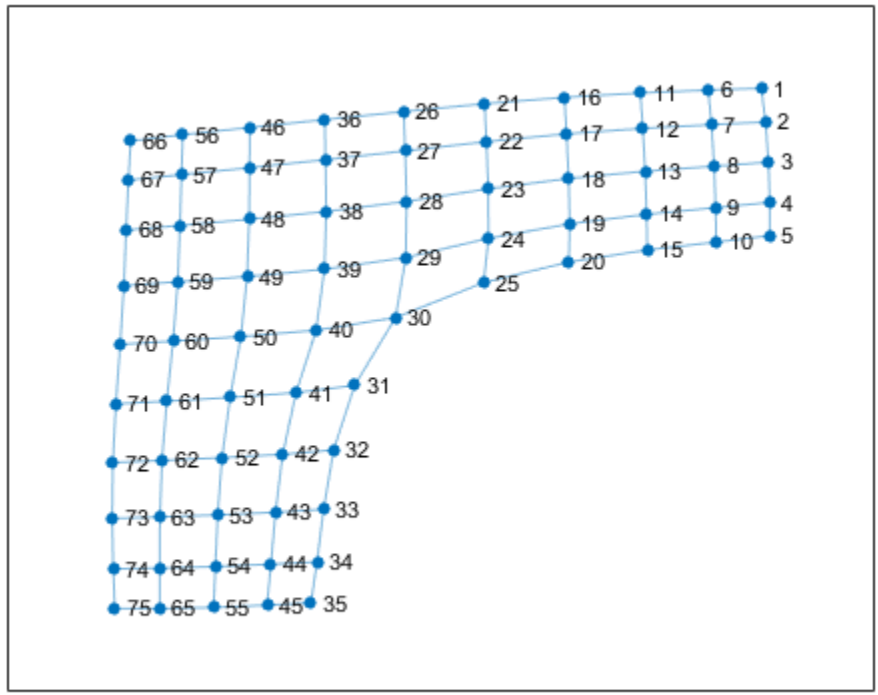
```
[T,p] = minspantree(G);
highlight(h,T,'EdgeColor','r','LineWidth',1.5)
```



Highlight Shortest Path

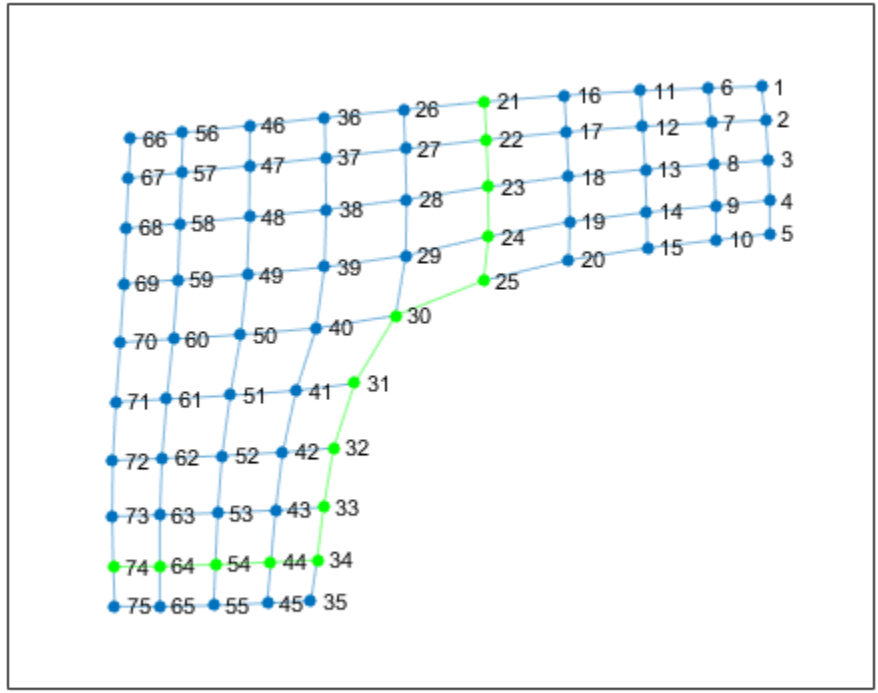
Create and plot a graph. Return a handle to the `GraphPlot` object, `h`.

```
n = 10;
A = delsq(numgrid('L',n+2));
G = graph(A,'omitselfloops');
G.Edges.Weight = ones(numedges(G),1);
h = plot(G);
```



Highlight the shortest path between nodes 74 and 21 by changing the color of the nodes and edges along the path to green.

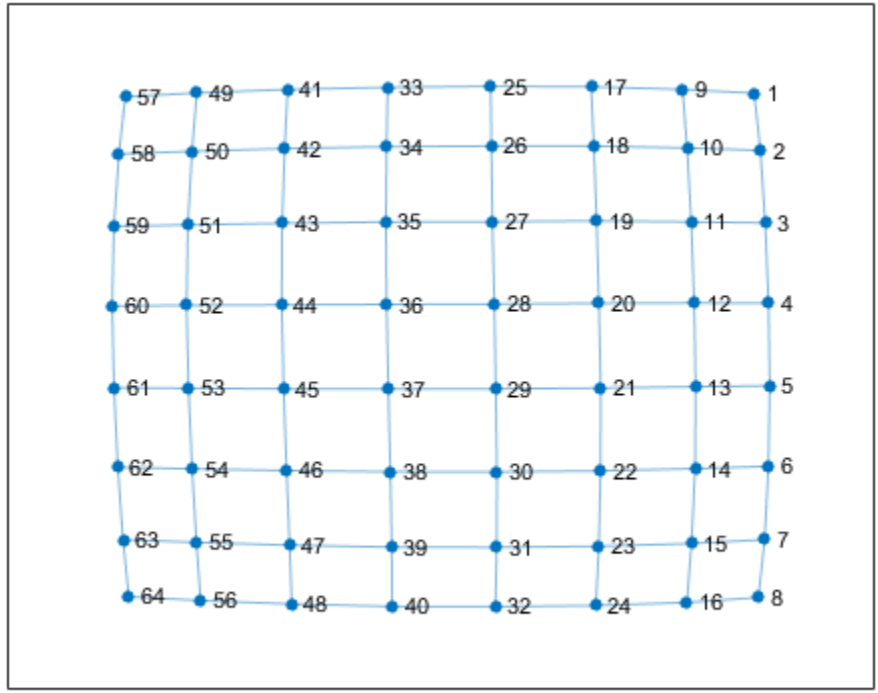
```
path = shortestpath(G,74,21);
highlight(h,path,'NodeColor','g','EdgeColor','g')
```



Highlight Node Neighbors

Create a graph representing a square grid with a side of 8 nodes. Plot the graph and return a handle to the `GraphPlot` object, `p`.

```
n = 8;
A = delsq(numgrid('S',n+2));
G = graph(A,'omitselfloops');
p = plot(G);
```



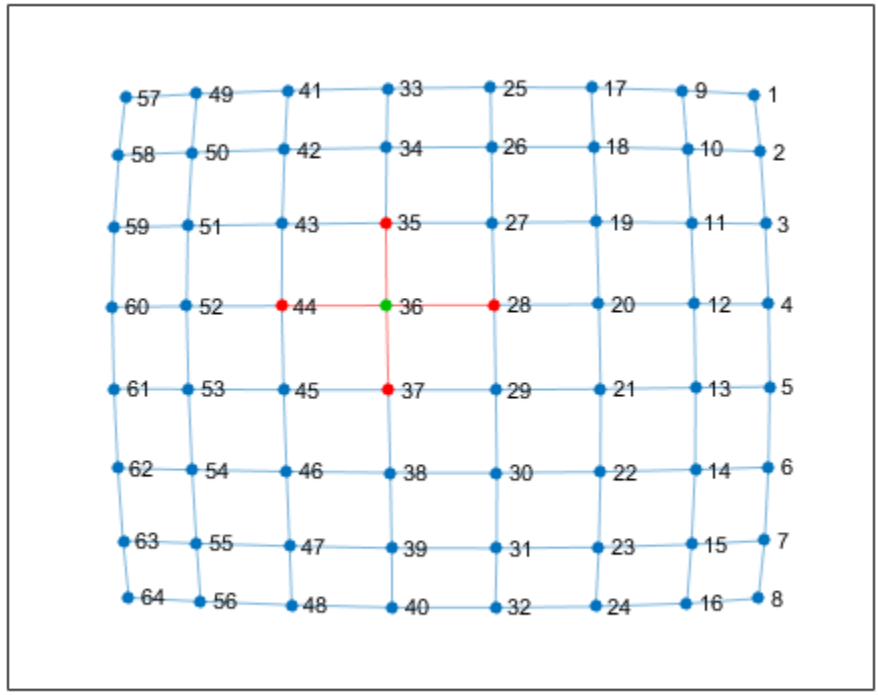
Find the neighbors of node 36.

```
n36 = neighbors(G,36)
```

```
n36 = 4×1

    28
    35
    37
    44
```

Use `highlight` to change the color of node 36 to green, and the color of its neighbors and their connecting edges to red.

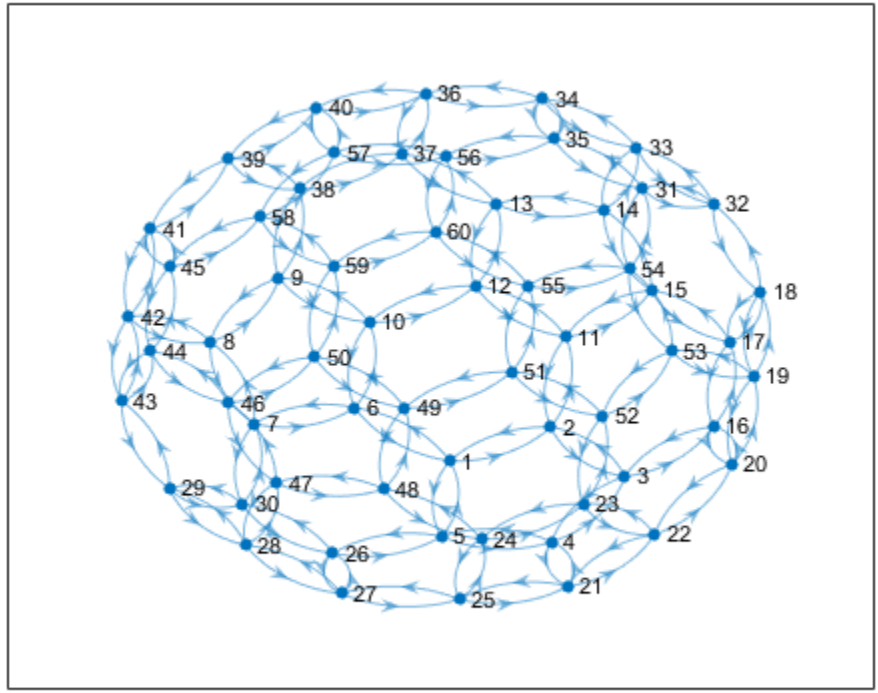
```
highlight(p,36,'NodeColor',[0 0.75 0])
highlight(p,n36,'NodeColor','red')
highlight(p,36,n36,'EdgeColor','red')
```



Highlight Path of Maximum Flow

Create and plot a directed graph. Return a handle to the `GraphPlot` object, `h`.

```
G = digraph(bucky);
h = plot(G);
```



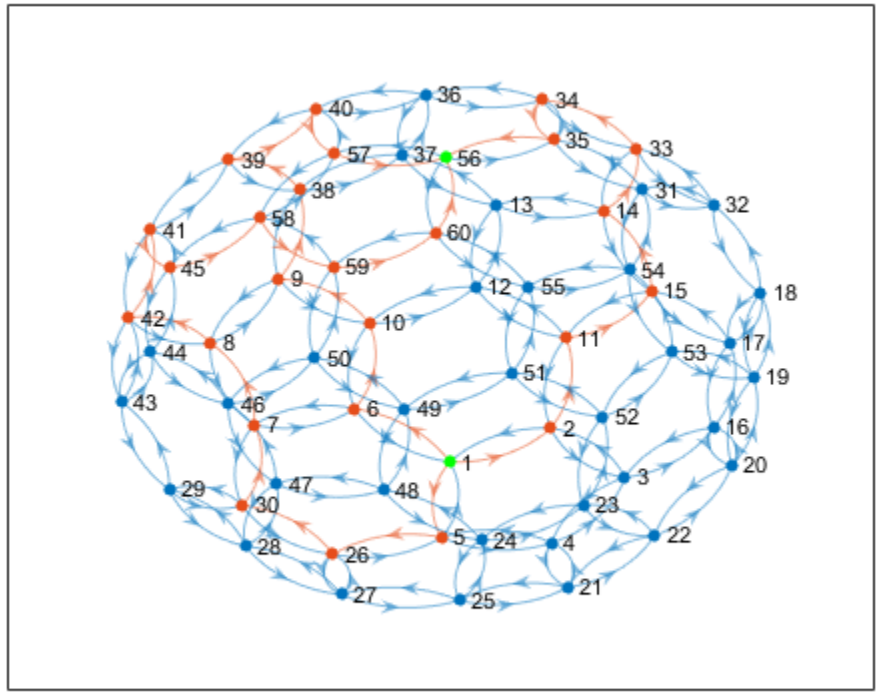
Compute the maximum flow between nodes 1 and 56. Specify two outputs to `maxflow` to return a directed graph of the nonzero flows, `GF`.

```
[mf,GF] = maxflow(G,1,56)
```

```
mf = 3

GF = 
  digraph with properties:

    Edges: [28x2 table]
    Nodes: [60x0 table]
```

Use `highlight` to change the color of the edges that contain nonzero flow values. Also change the color of source node 1 and target node 56 to green.

```
highlight(h,GF,'EdgeColor',[0.9 0.3 0.1],'NodeColor',[0.9 0.3 0.1])
highlight(h,[1 56],'NodeColor','g')
```

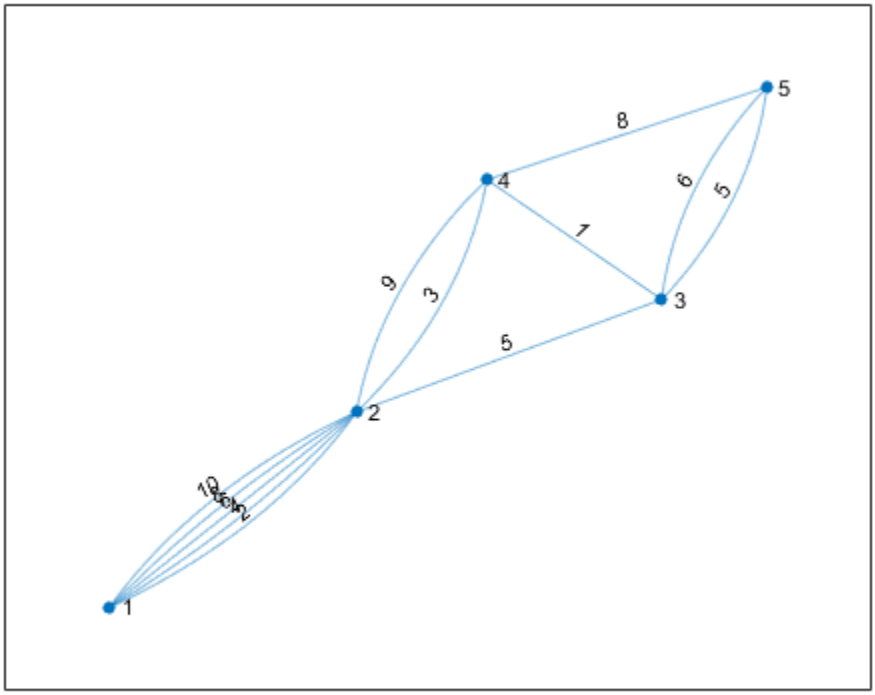


Shortest Path in Multigraph

Plot the shortest path between two nodes in a multigraph and highlight the specific edges that are traversed.

Create a weighted multigraph with five nodes. Several pairs of nodes have more than one edge between them. Plot the graph for reference.

```
G = graph([1 1 1 1 1 2 2 3 3 3 4 4],[2 2 2 2 2 3 4 4 5 5 5 2],[2 4 6 8 10 5 3 1 5 6 8 9
p = plot(G,'EdgeLabel',G.Edges.Weight);
```



Find the shortest path between node 1 and node 5. Since several of the node pairs have more than one edge between them, specify three outputs to `shortestpath` to return the specific edges that the shortest path traverses.

```
[P,d,edgepath] = shortestpath(G,1,5)
```

P = *1×5*

```
     1     2     4     3     5


d = 11

edgepath = 1×4

     1     7     9    10
```

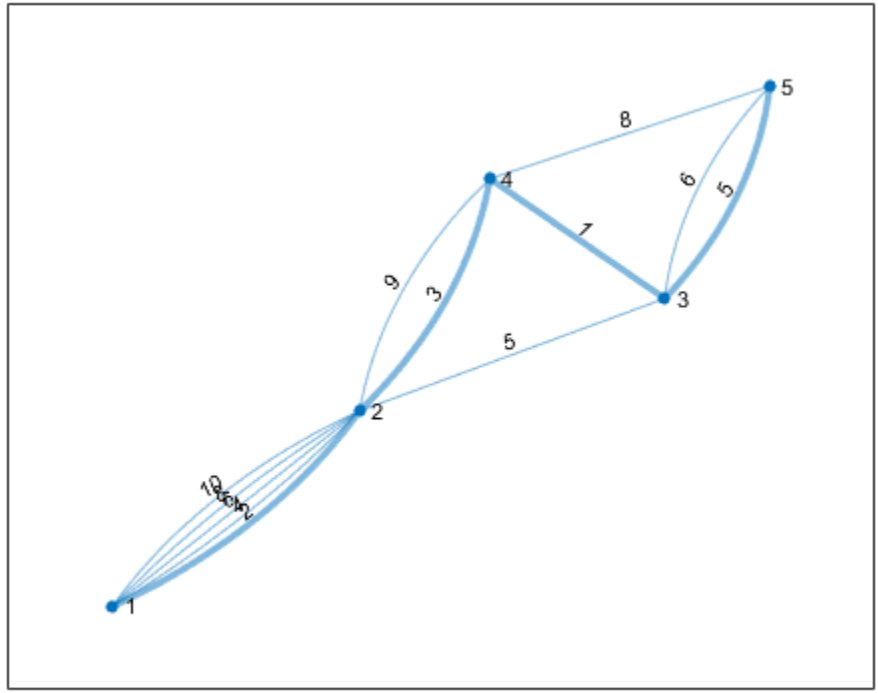
The results indicate that the shortest path has a total length of 11 and follows the edges given by `G.Edges(edgepath,:)`.

```
G.Edges(edgepath,:)
```

```
ans=4×2 table
    EndNodes    Weight
    ________    ______

     1    2       2
     2    4       3
     3    4       1
     3    5       5
```

Highlight this edge path by using the `highlight` function with the `'Edges'` name-value pair to specify the indices of the edges traversed.

```
highlight(p,'Edges',edgepath)
```



Input Arguments

H — Input graph plot
`GraphPlot` object

Input graph plot, specified as a `GraphPlot` object. Use the `graph` or `digraph` functions to create a graph, and then use `plot` with an output argument to return a `GraphPlot` object.

Example: `H = plot(G)`

`nodeIDs` — Nodes to highlight
logical vector | node indices | node names

Nodes to highlight, specified as a logical vector, or as one or more node indices or node names. If `nodeIDs` is a logical vector, then it must have length `numnodes(G)`.

This table shows the different ways to refer to one or more nodes either by their numeric node indices or by their node names.

Form	Single Node	Multiple Nodes
Node index	Scalar **Example:** `1`	Vector **Example:** `[1 2 3]`
Node name	Character vector **Example:** `'A'`	Cell array of character vectors **Example:** `{'A' 'B' 'C'}`
	String scalar **Example:** `"A"`	String array **Example:** `["A" "B" "C"]`

`nodeIDs` must not specify node names that conflict with any of the optional parameter names for `highlight`, such as `'Edges'` or `'EdgeColor'`. Use `findnode` to instead pass in the node index for these cases.

`G` — Graph to highlight
`graph` object | `digraph` object

Graph to highlight, specified as a `graph` or `digraph` object. `G` must have the same nodes and a subset of the edges of the underlying graph of `H`. Isolated nodes with degree `0` are not highlighted.

`s,t` — Node pairs (as separate arguments)
node indices | node names

Node pairs, specified as separate arguments of node indices or node names. Similarly located elements in `s` and `t` specify the source and target nodes for edges in the graph.

`s` and `t` must not specify node names that conflict with any of the optional parameter names for `highlight`, such as `'Edges'` or `'EdgeColor'`. Use `findnode` to instead pass in the node index for these cases.

Example: `highlight(H,[1 2],[3 3])` highlights the graph edges `(1,3)` and `(2,3)`.

Example: `highlight(H,'a','b')` highlights all edges from `'a'` to `'b'`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `highlight(H,nodes,'NodeColor','y')`

Edges to Highlight

Edges — Edges to highlight
scalar | vector

Edges to highlight, specified as the comma-separated pair consisting of `'Edges'` and a scalar edge index, vector of edge indices, or logical vector. Use this name-value pair to highlight a specific edge between nodes when multiple edges exist between the same two nodes.

The value of this name-value pair can be the third output from `shortestpath` or `shortestpathtree`, such as `[path,d,edgepath] = shortestpath(…)`.

Example: `highlight(p,'Edges',edgepath)`

Edge Properties

`EdgeColor` — Edge color
`[0 0.4470 0.7410]` (default) | RGB triplet | hexadecimal color code | color name

Edge color, specified as the comma-separated pair consisting of `'EdgeColor'` and an RGB triplet, hexadecimal color code, or color name.

- RGB triplets and hexadecimal color codes are useful for specifying custom colors.
  - An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
  - A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from

0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `plot(G,'EdgeColor','r')` creates a graph plot with red edges.

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as the comma-separated pair consisting of `'LineStyle'` and one of the line styles listed in this table.

Character(s)	Line Style	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Edge line width
`0.5` (default) | positive value

Edge line width, specified as the comma-separated pair consisting of `'LineWidth'` and a positive value in point units.

Example: `0.75`

`ArrowSize` — Arrow size
positive value

Arrow size, specified as a positive value in point units. The default value of `ArrowSize` is 7 for graphs with 100 or fewer nodes, and 4 for graphs with more than 100 nodes.

`ArrowSize` is used only for directed graphs.

Example: 15

`ArrowPosition` — Position of arrow along edge

`0.5` (default) | scalar

Position of arrow along edge, specified as a value in `[0 1]`. A value near 0 places arrows closer to the source node, and a value near 1 places arrows closer to the target node. The default value is `0.5` so that the arrows are halfway between the source and target nodes.

`ArrowPosition` is used only for directed graphs.

Node Properties

`NodeColor` — Node color

`[0 0.4470 0.7410]` (default) | RGB triplet | hexadecimal color code | color name

Node color, specified as the comma-separated pair consisting of `'NodeColor'` and an RGB triplet, hexadecimal color code, or color name.

- RGB triplets and hexadecimal color codes are useful for specifying custom colors.
  - An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
  - A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

  Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `plot(G,'NodeColor','k')` creates a graph plot with black nodes.

`Marker` — Node marker symbol

`'o'` (default) | character vector

Node marker symbol, specified as the comma-separated pair consisting of `'Marker'` and one of the character vectors listed in this table. The default is to use circular markers for the graph nodes.

Value	Description
'o'	Circle
'+'	Plus sign

Value	Description
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'+'`

Example: `'diamond'`

`MarkerSize` — Node marker size
positive value

Node marker size, specified as the comma-separated pair consisting of `'MarkerSize'` and a positive value in point units. The default marker size is 4 for graphs with 100 or fewer nodes, and 2 for graphs with more than 100 nodes.

Example: `10`

Node and Edge Labels

`NodeLabelColor` — Color of node labels
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | color name

Node label color, specified as an RGB triplet, hexadecimal color code, or color name.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `plot(G,'NodeLabel',C,'NodeLabelColor','m')` creates a graph plot with magenta node labels.

`EdgeLabelColor` — Color of edge labels

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | color name

Edge label color, specified as an RGB triplet, hexadecimal color code, or color name.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `plot(G,'EdgeLabel',C,'EdgeLabelColor','m')` creates a graph plot with magenta edge labels.

Font

`NodeFontName` — Font name for node labels
`'Helvetica'` (default) | supported font name | `'FixedWidth'`

Font name for node labels, specified as a supported font name or `'FixedWidth'`. To display and print properly, you must choose a font that your system supports. The default font depends on the specific operating system and locale. For example, Windows and Linuxsystems in English localization use the Helvetica font by default.

To use a fixed-width font that looks good in any locale, specify `'FixedWidth'`.

Example: `'Cambria'`

`NodeFontSize` — Font size for node labels
8 (default) | positive number

Font size for node labels, specified as a positive number.

`NodeFontWeight` — Thickness of text in node labels
`'normal'` (default) | `'bold'`

Thickness of text in node labels, specified as `'normal'` or `'bold'`:

- `'bold'` — **Thicker character outlines than normal**
- `'normal'` — Normal weight as defined by the particular font

Not all fonts have a bold font weight.

Data Types: `cell` | `char` | `string`

`NodeFontAngle` — Character slant of text in node labels
`'normal'` (default) | `'italic'`

Character slant of text in node labels, specified as `'normal'` or `'italic'`:

- `'italic'` — *Slanted characters*
- `'normal'` — No character slant

Not all fonts have both font styles.

Data Types: `cell` | `char` | `string`

`EdgeFontName` — Font name for edge labels
`'Helvetica'` (default) | supported font name | `'FixedWidth'`

Font name for edge labels, specified as a supported font name or `'FixedWidth'`. To display and print properly, you must choose a font that your system supports. The default font depends on the specific operating system and locale. For example, Windows and Linuxsystems in English localization use the Helvetica font by default.

To use a fixed-width font that looks good in any locale, specify `'FixedWidth'`.

Example: `'Cambria'`

`EdgeFontSize` — Font size for edge labels
8 (default) | positive number

Font size for edge labels, specified as a positive number.

`EdgeFontWeight` — Thickness of text in edge labels
`'normal'` (default) | `'bold'`

Thickness of text in edge labels, specified as `'normal'` or `'bold'`:

- `'bold'` — **Thicker character outlines than normal**
- `'normal'` — Normal weight as defined by the particular font

Not all fonts have a bold font weight.

Data Types: `cell` | `char` | `string`

`EdgeFontAngle` — Character slant of text in edge labels
`'normal'` (default) | `'italic'`

Character slant of text in edge labels, specified as `'normal'` or `'italic'`:

- `'italic'` — *Slanted characters*
- `'normal'` — No character slant

Not all fonts have both font styles.

Data Types: `cell` | `char` | `string`

See Also

Functions
`labeledge` | `labelnode` | `layout` | `plot`

Objects
`GraphPlot`

Topics
"Graph Plotting and Customization"

Introduced in R2015b

labeledge

Package: `matlab.graphics.chart.primitive`

Label graph edges

Syntax

```
labeledge(H,s,t,Labels)
labeledge(H,idx,Labels)
```

Description

`labeledge(H,s,t,Labels)` labels the edges specified by `(s,t)` node pairs with the character vectors or strings contained in `Labels`. The lengths of `s`, `t`, and `Labels` must be equal, or `Labels` can be scalar. If there are multiple edges between `s` and `t`, then the same label is applied to all of them.

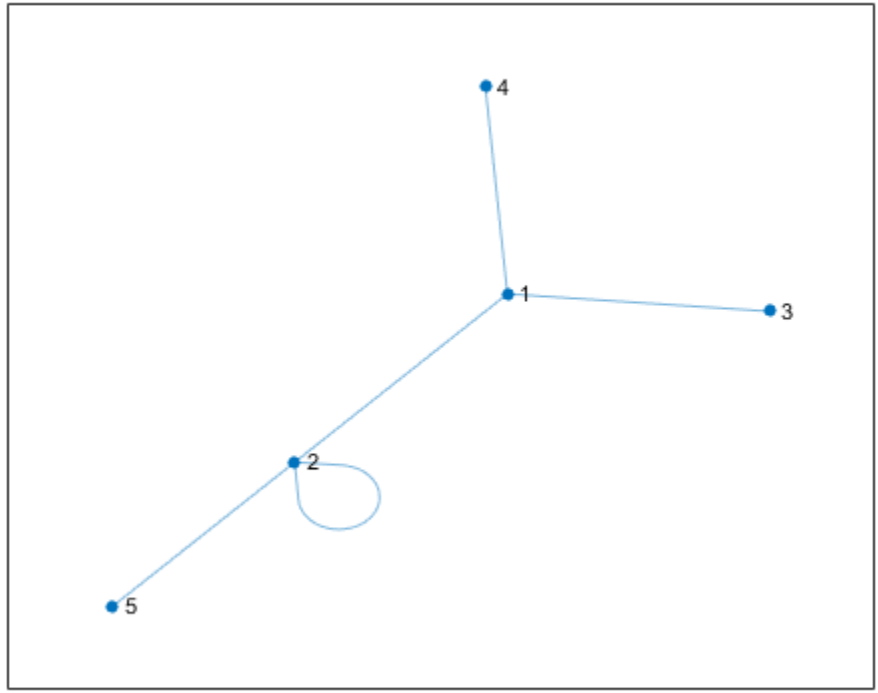
`labeledge(H,idx,Labels)` specifies the edges to label using logical or numeric edge indices.

Examples

Label Graph Edge Weight

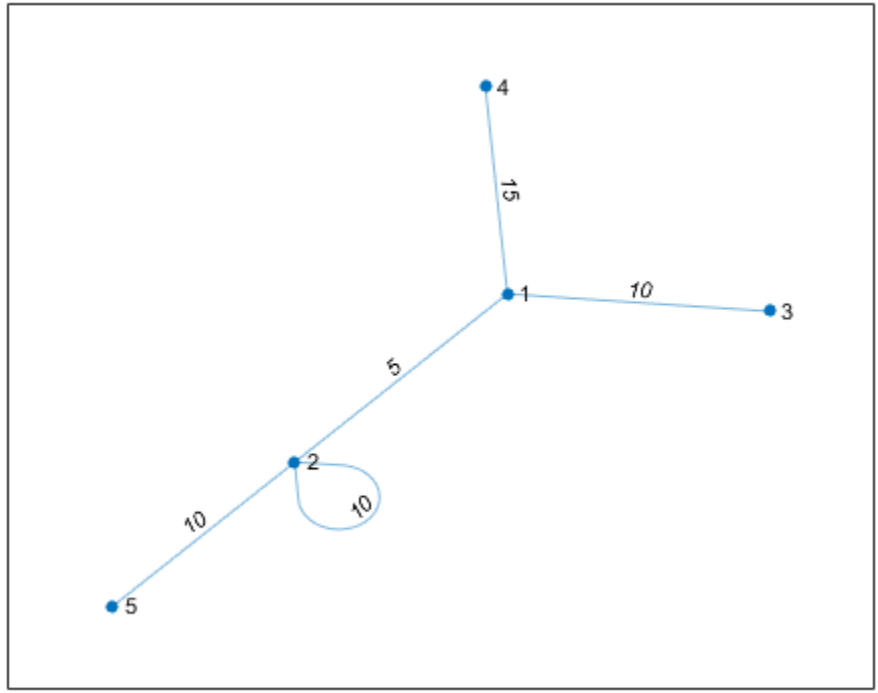
Create and plot a graph.

```
s = [1 1 1 2 2];
t = [2 3 4 2 5];
G = graph(s,t);
h = plot(G);
```



Label the edge weights using `labeledge`.

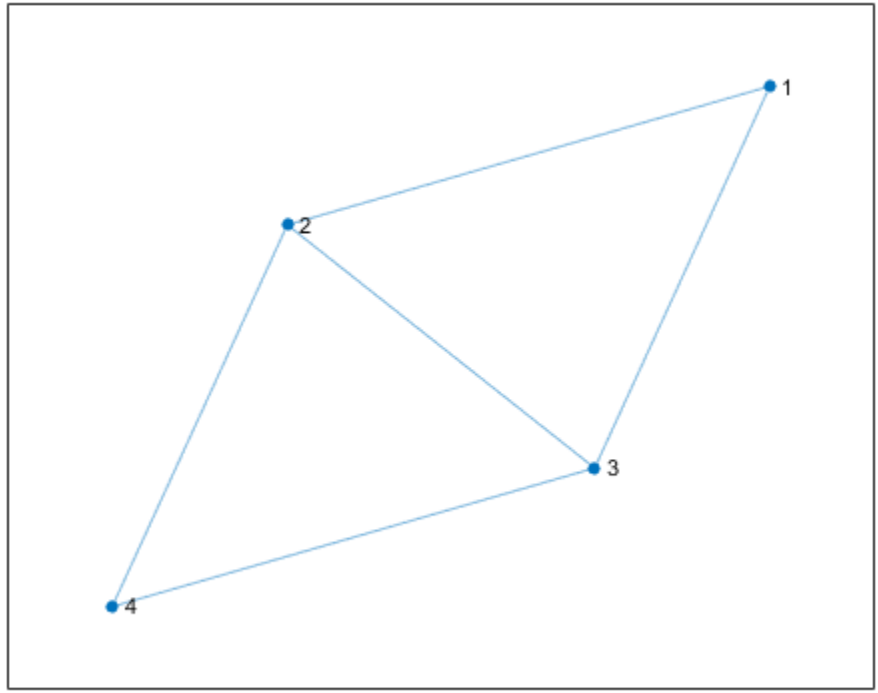
```
weights = [5 10 15 10 10];
labeledge(h,1:numedges(G),weights)
```



Add Text Labels to Graph Edges

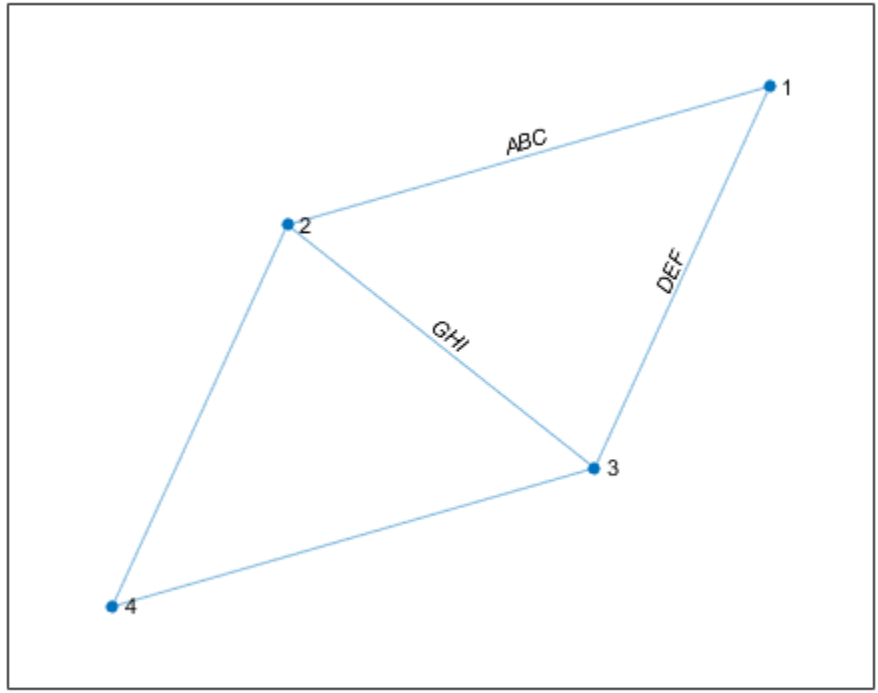
Create and plot a graph.

```
s = [1 1 2 2 3];
t = [2 3 3 4 4];
G = graph(s,t);
h = plot(G);
```



Add text labels to three of the five graph edges.

```
labelText = {'ABC' 'DEF' 'GHI'};
labeledge(h,[1 1 2],[2 3 3],labelText)
```



Input Arguments

H — Input graph plot
`GraphPlot` object

Input graph plot, specified as a `GraphPlot` object. Use the `graph` or `digraph` functions to create a graph, and then use `plot` with an output argument to return a `GraphPlot` object.

Example: `H = plot(G)`

`s,t` — Node pairs (as separate arguments)

node indices | node names

Node pairs, specified as separate arguments of node indices or node names. Similarly located elements in `s` and `t` specify the source and target nodes for edges in the graph.

This table shows the different ways to refer to one or more nodes either by their numeric node indices or by their node names.

Form	Single Node	Multiple Nodes
Node index	Scalar **Example:** `1`	Vector **Example:** `[1 2 3]`
Node name	Character vector **Example:** `'A'`	Cell array of character vectors **Example:** `{'A' 'B' 'C'}`
	String scalar **Example:** `"A"`	String array **Example:** `["A" "B" "C"]`

Example: `labeledge(H,[1 2 2],[2 3 4],{'label1' 'label2' 'label3'})`

Example: `labeledge(H,{'a' 'b' 'b'},{'b' 'c' 'd'},{'label1' 'label2' 'label3'})`

`idx` — Edge indices

scalar | vector

Edge indices, specified as a scalar, vector of positive integers, or logical vector.

- A vector of positive integers specifies edge indices for a graph `G` corresponding to rows in the table `G.Edges`, such that `G.Edges(idx,:)` returns the end nodes of the edge. In this form, the maximum value for `idx` is `numedges(G)`, and the number of selected nodes is equal to the number of indices in `idx`.
- A logical vector specifies whether each edge is labeled. In this form, the length of the logical vector must be `numedges(G)`, and the number of selected nodes is equal to the number of nonzero indices in `idx`.

Example: `labeledge(H,[1 3 5 7],{'label1' 'label2' 'label3' 'label4'})`

`Labels` — Edge labels
character vector | string scalar | cell array of character vectors | string array

Edge labels, specified as a character vector, string scalar, cell array of character vectors, or string array.

- If `Labels` is a character vector or string scalar, then `labeledge` uses that label for each specified edge.
- If `Labels` is a cell array or string array, then it must contain a label for each specified edge.

Data Types: `char` | `cell` | `string`

See Also

Functions
`highlight` | `labelnode` | `layout` | `plot`

Objects
`GraphPlot`

Topics
"Graph Plotting and Customization"

Introduced in R2015b

labelnode

Package: `matlab.graphics.chart.primitive`

Label graph nodes

Syntax

`labelnode(H,nodeIDs,Labels)`

Description

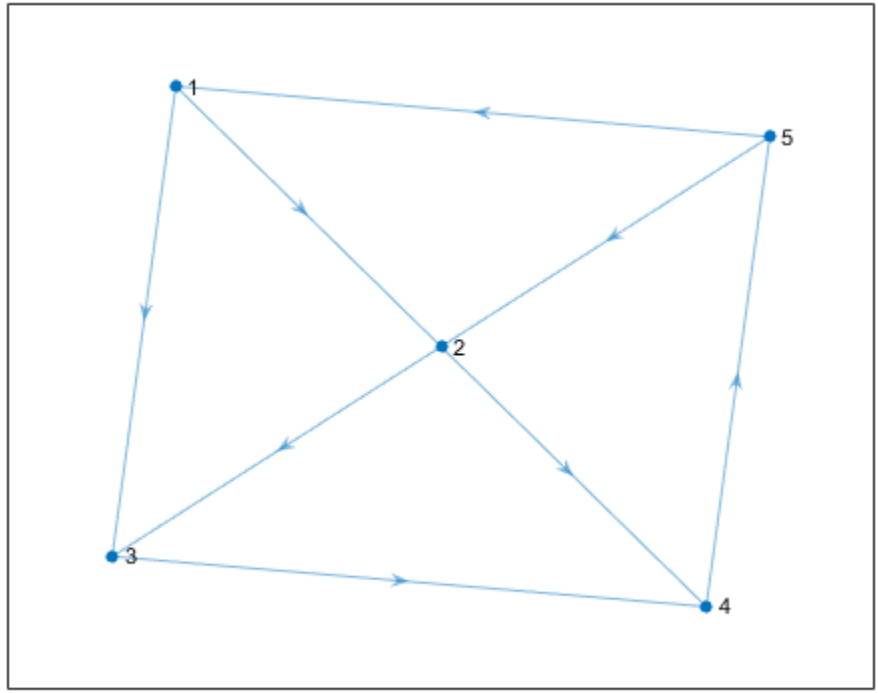
`labelnode(H,nodeIDs,Labels)` labels the nodes specified by `nodeIDs` with the character vectors or strings contained in `Labels`.

Examples

Label Graph Nodes

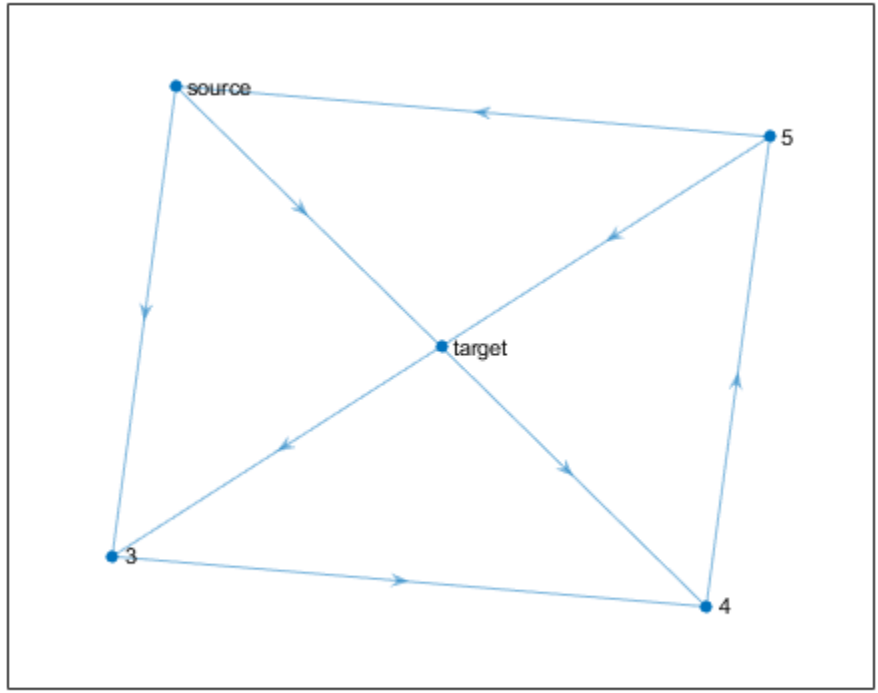
Create and plot a graph.

```
s = [1 1 2 2 3 4 5 5];
t = [2 3 3 4 4 5 1 2];
G = digraph(s,t);
h = plot(G);
```



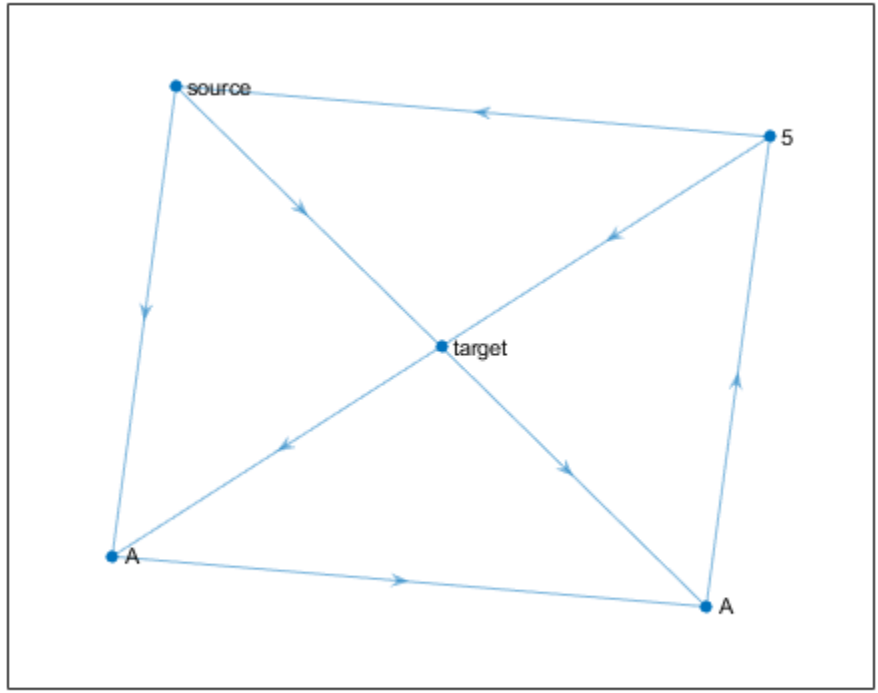
The `plot` function automatically labels the graph nodes with their node indices (or with their node names, if available). Use text labels for some of the graph nodes instead.

```
labelnode(h,[1 2],{'source' 'target'})
```



Label node 3 and node 4 as `'A'`. Since the node labels do not change the underlying graph, `G`, the labels can contain duplicate names (the official node names in `G` must always be unique).

```
labelnode(h,[3 4],'A')
```



Input Arguments

H — Input graph plot
`GraphPlot` object

Input graph plot, specified as a `GraphPlot` object. Use the `graph` or `digraph` functions to create a graph, and then use `plot` with an output argument to return a `GraphPlot` object.

Example: `H = plot(G)`

`nodeIDs` — Node identifiers
logical vector | node indices | node names

Node identifiers, specified as a logical vector or as one or more node indices or node names. If `nodeIDs` is a logical vector, then its length must match the number of nodes in the graph.

This table shows the different ways to refer to one or more nodes either by their numeric node indices or by their node names.

Form	Single Node	Multiple Nodes
Node index	Scalar **Example:** `1`	Vector **Example:** `[1 2 3]`
Node name	Character vector **Example:** `'A'`	Cell array of character vectors **Example:** `{'A' 'B' 'C'}`
	String scalar **Example:** `"A"`	String array **Example:** `["A" "B" "C"]`

`Labels` — Node labels
character vector | string scalar | cell array of character vectors | string array

Node labels, specified as a character vector or a cell array of character vectors.

- If `Labels` is a character vector or string scalar, then `labelnode` uses that label for each node specified by `nodeIDs`.
- If `Labels` is a cell array or string array, then it must contain a label for each node specified by `nodeIDs`.

Data Types: `char` | `cell` | `string`

See Also

Functions
`highlight` | `labeledge` | `layout` | `plot`

Objects
`GraphPlot`

Topics
"Graph Plotting and Customization"

Introduced in R2015b

layout

Package: `matlab.graphics.chart.primitive`

Change layout of graph plot

Syntax

```
layout(H)
layout(H,method)
layout(H,method,Name,Value)
```

Description

`layout(H)` changes the layout of graph plot `H` by using an automatic choice of layout method based on the structure of the graph. The `layout` function modifies the `XData` and `YData` properties of `H`.

`layout(H,method)` optionally specifies the layout method. `method` can be `'circle'`, `'force'`, `'layered'`, `'subspace'`, `'force3'`, or `'subspace3'`.

`layout(H,method,Name,Value)` uses additional options specified by one or more name-value pair arguments. For example, `layout(H,'force','Iterations',N)` specifies the number of iterations to use in computing the force layout, and `layout(H,'layered','Sources',S)` uses a layered layout with source nodes `S` included in the first layer.

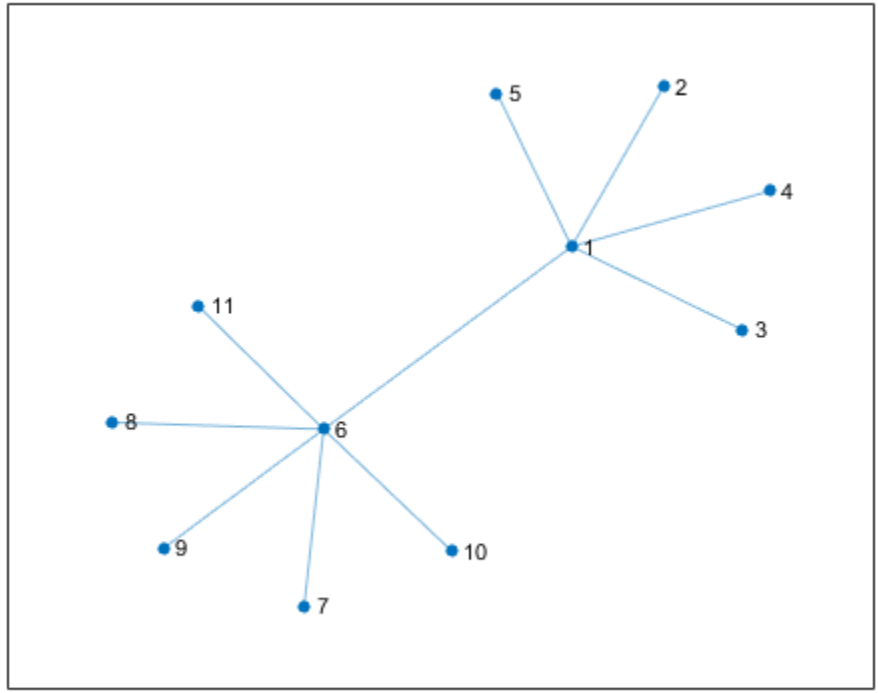
Examples

Graph Layout Based on Structure

Create and plot a graph using the `'force'` layout.

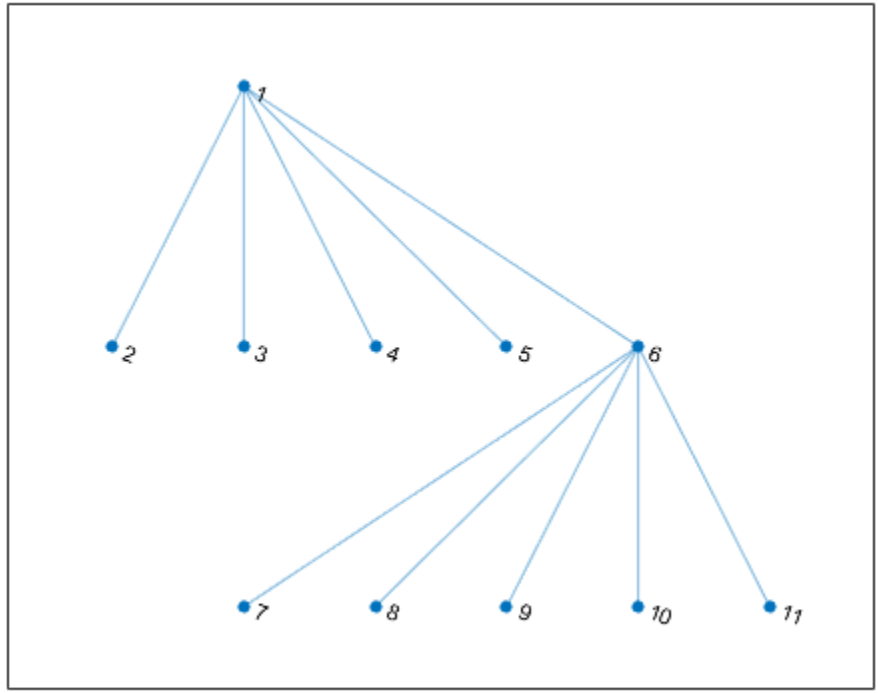
```
s = [1 1 1 1 1 6 6 6 6 6];
t = [2 3 4 5 6 7 8 9 10 11];
```

```
G = graph(s,t);
h = plot(G,'Layout','force');
```



Change the layout to be the default that `plot` determines based on the structure and properties of the graph. The result is the same as using `plot(G)`.

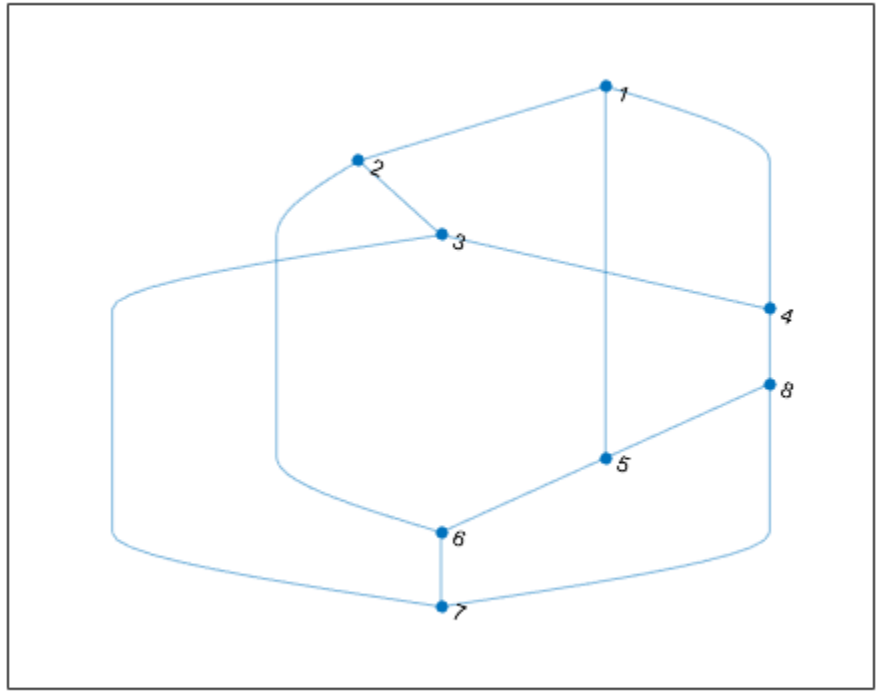
```
layout(h)
```



Change Layout of Graph

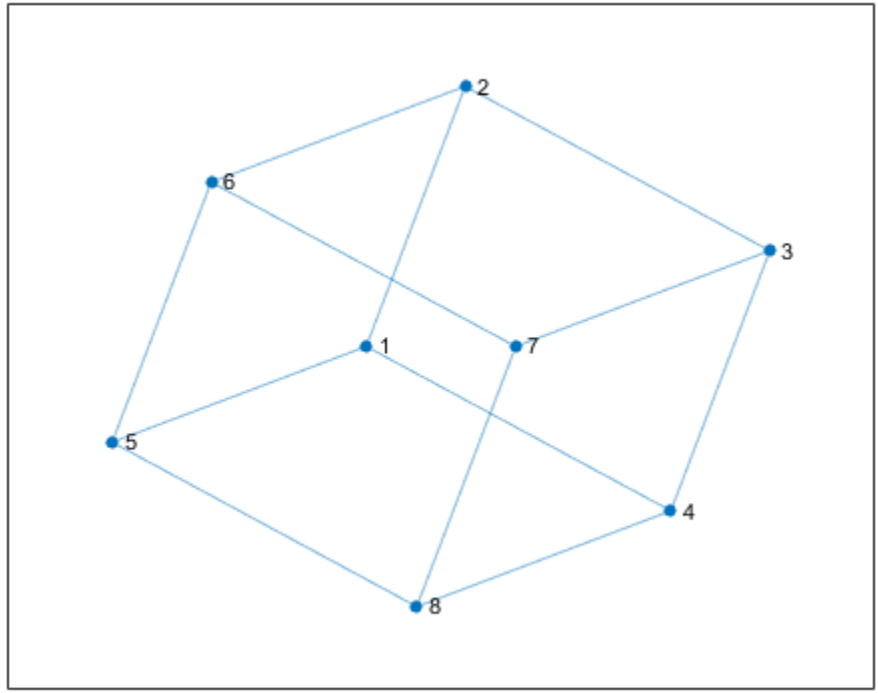
Create and plot a graph using the `'layered'` layout.

```
s = [1 1 1 2 2 3 3 4 5 5 6 7];
t = [2 4 5 3 6 4 7 8 6 8 7 8];
G = graph(s,t);
h = plot(G,'Layout','layered');
```



Change the layout of the graph to use the `'subspace'` method.

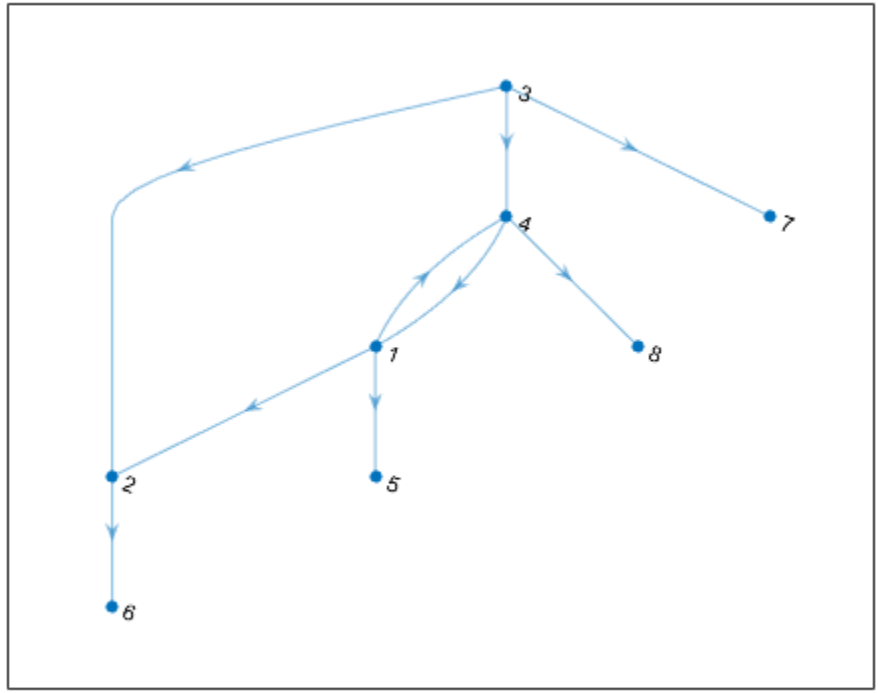
```
layout(h,'subspace')
```



Refine Layout Method of Graph

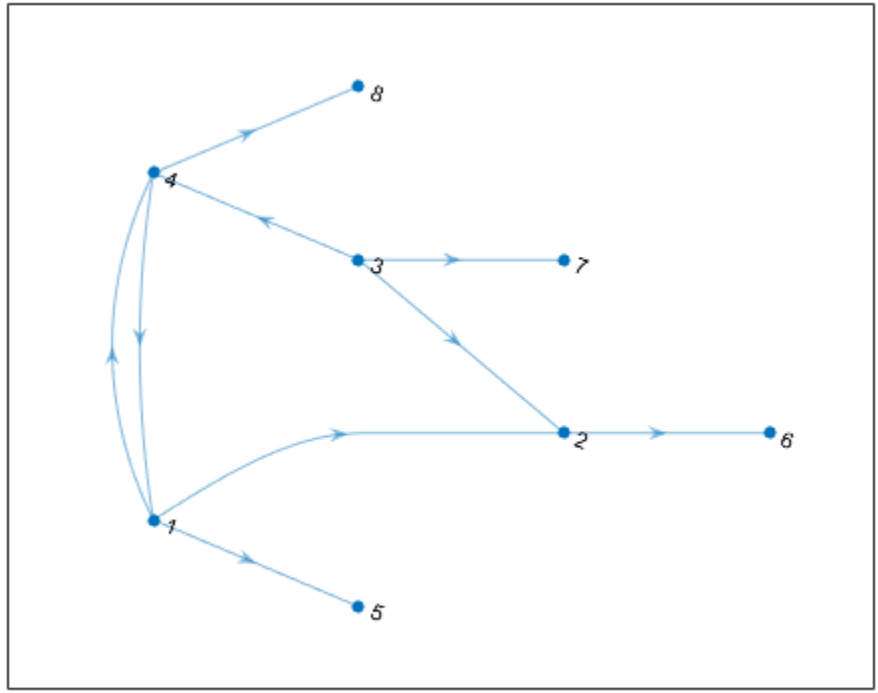
Create and plot a graph using the `'layered'` layout method.

```
s = [1 1 1 2 3 3 3 4 4];
t = [2 4 5 6 2 4 7 8 1];
G = digraph(s,t);
h = plot(G,'Layout','layered');
```



Use the `layout` function to refine the hierarchical layout by specifying source nodes and a horizontal orientation.

```
layout(h,'layered','Direction','right','Sources',[1 4])
```

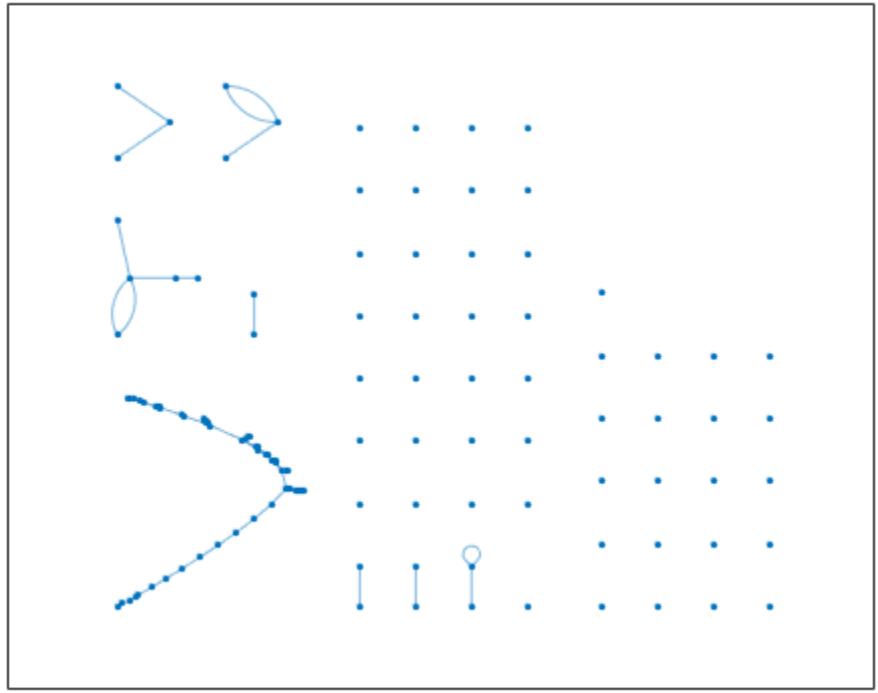


Graph Layout with Multiple Components

Plot a graph that has multiple components, and then show how to use the `'UseGravity'` option to improve the visualization.

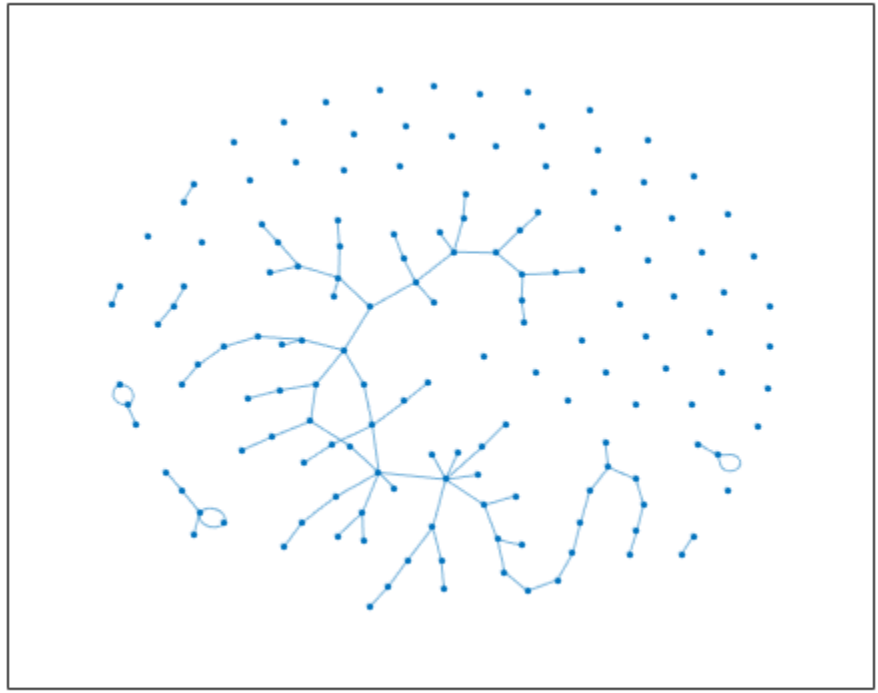
Create and plot a graph that has 150 nodes separated into many disconnected components. MATLAB® lays the graph components out on a grid.

```
s = [1 3 5 7 7 10:100];
t = [2 4 6 8 9 randi([10 100],1,91)];
G = graph(s,t,[],150);
h = plot(G);
```



Update the layout coordinates of the graph object, and specify `'UseGravity'` as `true` so that the components are layed out radially around the origin, with more space allotted for the larger components.

```
layout(h,'force','UseGravity',true)
```

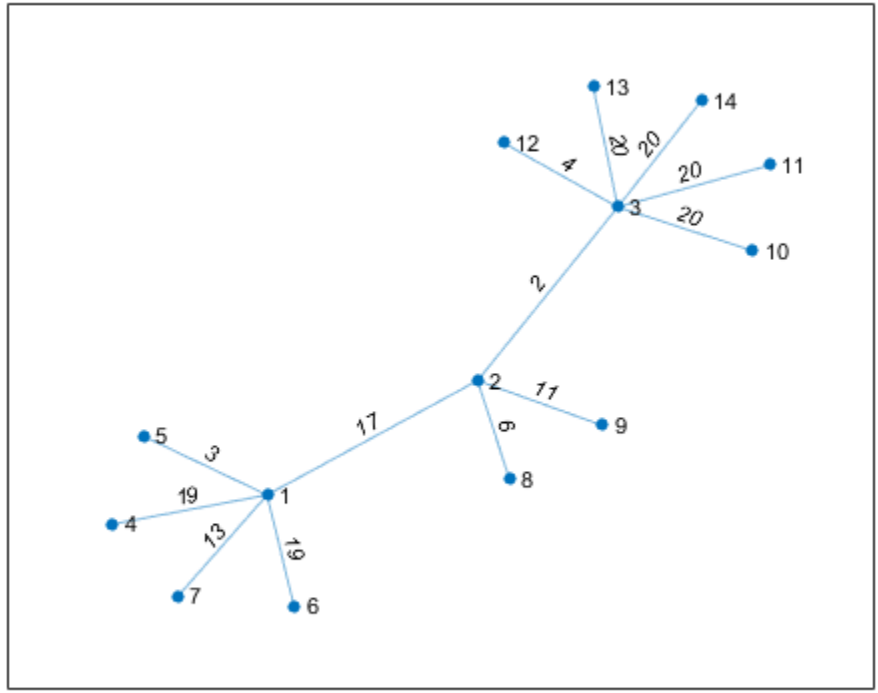


Graph Layout Based on Edge Weights

Plot a graph using the `'WeightEffect'` name-value pair to make the length of graph edges proportional to their weights.

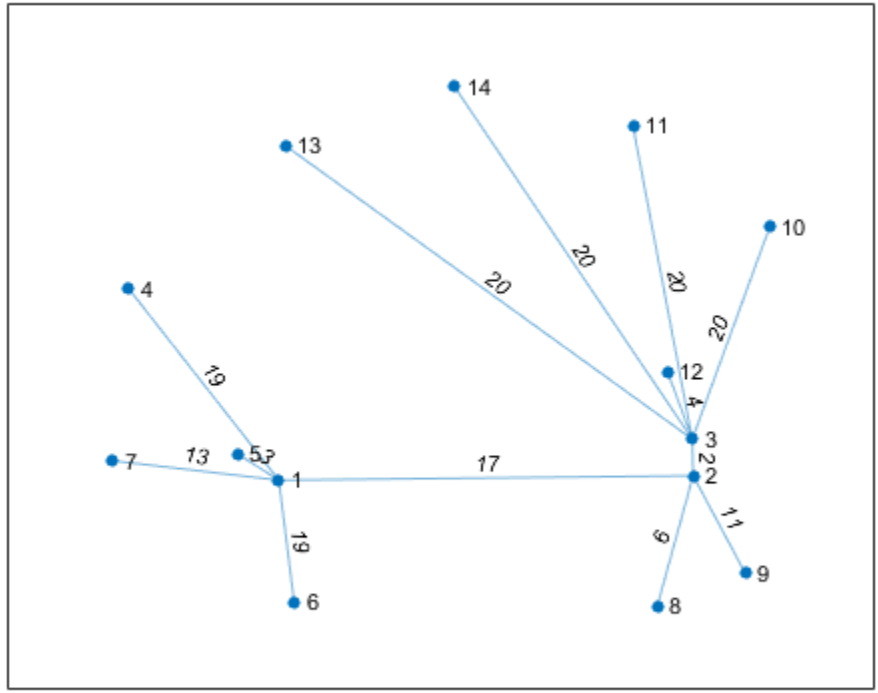
Create and plot a directed graph with weighted edges.

```
s = [1 1 1 1 1 2 2 2 3 3 3 3 3];
t = [2 4 5 6 7 3 8 9 10 11 12 13 14];
weights = randi([1 20],1,13);
G = graph(s,t,weights);
p = plot(G,'Layout','force','EdgeLabel',G.Edges.Weight);
```



Recompute the layout of the graph using the `'WeightEffect'` name-value pair, so that the length of each edge is proportional to its weight. This makes it so that the edges with the largest weights are the longest.

```
layout(p,'force','WeightEffect','direct')
```



Input Arguments

H — Input graph plot
`GraphPlot` object

Input graph plot, specified as a `GraphPlot` object. Use the `graph` or `digraph` functions to create a graph, and then use `plot` with an output argument to return a `GraphPlot` object.

Example: `H = plot(G)`

method — Layout method
`'auto'` (default) | `'circle'` | `'force'` | `'layered'` | `'subspace'` | `'force3'` | `'subspace3'`

Layout method, specified as one of the options in the table. The table also lists compatible name-value pairs to further refine each layout method.

Option	Description	Layout-Specific Name-Value Pairs
`'auto'` (default)	Automatic choice of layout method based on the size and structure of the graph.	—
`'circle'`	Circular layout. Places the graph nodes on a circle centered at the origin with radius 1.	`'Center'` — Center node in circular layout
`'force'`	Force-directed layout [1]. Uses attractive forces between adjacent nodes and repulsive forces between distant nodes.	`'Iterations'` — Number of force-directed layout iterations `'WeightEffect'` — Effect of edge weights on layout `'UseGravity'` — Gravity toggle for layouts with multiple components `'XStart'` — Starting *x*-coordinates for nodes `'YStart'` — Starting *y*-coordinates for nodes

Option	Description	Layout-Specific Name-Value Pairs
`'layered'`	Layered layout [2], [3], [4]. Places the graph nodes into a set of layers, revealing hierarchical structure. By default the layers progress downwards (the arrows of a directed acyclic graph point down).	`'Direction'` — Direction of layers `'Sources'` — Nodes to include in the first layer `'Sinks'` — Nodes to include in the last layer `'AssignLayers'` — Layer assignment method
`'subspace'`	Subspace embedding layout [5]. Plots the graph nodes in a high-dimensional embedded subspace, and then projects the positions back into 2-D. By default the subspace dimension is either 100 or the total number of nodes, whichever is smaller.	`'Dimension'` — Dimension of embedded subspace

Option	Description	Layout-Specific Name-Value Pairs
`'force3'`	3-D force-directed layout.	`'Iterations'` — Number of force-directed layout iterations `'WeightEffect'` — Effect of edge weights on layout `'UseGravity'` — Gravity toggle for layouts with multiple components `'XStart'` — Starting *x*-coordinates for nodes `'YStart'` — Starting *y*-coordinates for nodes `'ZStart'` — Starting *z*-coordinates for nodes
`'subspace3'`	3-D subspace embedding layout.	`'Dimension'` — Dimension of embedded subspace

Example: `layout(H,'layered')`

Example: `layout(H,'force3','Iterations',10)`

Example: `layout(H,'subspace','Dimension',50)`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `layout(H,'subspace','Dimension',200)`

Force

`Iterations` — Number of force-directed layout iterations
`100` (default) | positive scalar integer

Number of force-directed layout iterations, specified as the comma-separated pair consisting of `'Iterations'` and a positive scalar integer.

This option is available only when `method` is `'force'` or `'force3'`.

Example: `layout(H,'force','Iterations',250)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`WeightEffect` — Effect of edge weights on layout
`'none'` (default) | `'direct'` | `'inverse'`

Effect of edge weights on layout, specified as the comma-separated pair consisting of `'WeightEffect'` and one of the values in this table. If there are multiple edges between two nodes (as in a directed graph with an edge in each direction, or a multigraph), then the weights are summed before computing `'WeightEffect'`.

This option is available only when `method` is `'force'` or `'force3'`.

Value	Description
`'none'` (default)	Edge weights do not affect the layout.
`'direct'`	Edge length is proportional to the edge weight, `G.Edges.Weight`. Larger edge weights produce longer edges.
`'inverse'`	Edge length is inversely proportional to the edge weight, `1./G.Edges.Weight`. Larger edge weights produce shorter edges.

Example: `layout(H,'force','WeightEffect','inverse')`

`UseGravity` — Gravity toggle for layouts with multiple components
`'on'` | `'off'` | `true` | `false`

Gravity toggle for layouts with multiple components, specified as the comma-separated pair consisting of `'UseGravity'` and `'on'`, `'off'`, `true`, or `false`.

By default, MATLAB lays out graphs with multiple components on a grid. The grid can obscure the details of larger components since they are given the same amount of space as smaller components. With `'UseGravity'` set to `'on'` or `true`, multiple components are instead layed out radially around the origin. This layout spreads out the components in a more natural way, and provides more space for larger components.

This option is available only when `method` is `'force'` or `'force3'`.

Example: `layout(H,'force','UseGravity',true)`

Data Types: `char` | `logical`

`XStart` — Starting x-coordinates for nodes
vector

Starting x-coordinates for nodes, specified as the comma-separated pair consisting of `'XStart'` and a vector of node coordinates. Use this option together with `'YStart'` to specify 2-D starting coordinates (or with `'YStart'` and `'ZStart'` to specify 3-D starting coordinates) before iterations of the force-directed algorithm change the node positions.

This option is available only when `method` is `'force'` or `'force3'`.

Example: `layout(H,'force','XStart',x,'YStart',y)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`YStart` — Starting y-coordinates for nodes
vector

Starting y-coordinates for nodes, specified as the comma-separated pair consisting of `'YStart'` and a vector of node coordinates. Use this option together with `'XStart'` to specify 2-D starting coordinates (or with `'XStart'` and `'ZStart'` to specify 3-D starting coordinates) before iterations of the force-directed algorithm change the node positions.

This option is available only when `method` is `'force'` or `'force3'`.

Example: `layout(H,'force','XStart',x,'YStart',y)`

Example: `layout(H,'force','XStart',x,'YStart',y,'ZStart',z)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ZStart` — Starting z-coordinates for nodes
vector

Starting z-coordinates for nodes, specified as the comma-separated pair consisting of `'ZStart'` and a vector of node coordinates. Use this option together with `'XStart'` and `'YStart'` to specify the starting *x*, *y*, and *z* node coordinates before iterations of the force-directed algorithm change the node positions.

This option is available only when `method` is `'force3'`.

Example: `layout(H,'force','XStart',x,'YStart',y,'ZStart',z)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Layered

`Direction` — Direction of layers
`'down'` (default) | `'up'` | `'left'` | `'right'`

Direction of layers, specified as the comma-separated pair consisting of `'Direction'` and either `'down'`, `'up'`, `'left'` or `'right'`. For directed acyclic (DAG) graphs, the arrows point in the indicated direction.

This option is available only when `method` is `'layered'`.

Example: `layout(H,'layered','Direction','up')`

`Sources` — Nodes to include in the first layer
node indices | node names

Nodes to include in the first layer, specified as the comma-separated pair consisting of `'Sources'` and one or more node indices or node names.

This table shows the different ways to refer to one or more nodes either by their numeric node indices or by their node names.

Form	Single Node	Multiple Nodes
Node index	Scalar **Example:** `1`	Vector **Example:** `[1 2 3]`
Node name	Character vector **Example:** `'A'`	Cell array of character vectors **Example:** `{'A' 'B' 'C'}`

Form	Single Node	Multiple Nodes
	String scalar **Example:** `"A"`	String array **Example:** `["A" "B" "C"]`

This option is available only when `method` is `'layered'`.

Example: `layout(H,'layered','Sources',[1 3 5])`

`Sinks` — Nodes to include in the last layer
node indices | node names

Nodes to include in the last layer, specified as the comma-separated pair consisting of `'Sinks'` and one or more node indices or node names.

This option is available only when `method` is `'layered'`.

Example: `layout(H,'layered','Sinks',[2 4 6])`

`AssignLayers` — Layer assignment method
`'auto'` (default) | `'asap'` | `'alap'`

Layer assignment method, specified as the comma-separated pair consisting of `'AssignLayers'` and one of the options in this table.

Option	Description
`'auto'` (default)	Node assignment uses either `'asap'` or `'alap'`, whichever is more compact.
`'asap'`	As soon as possible. Each node is assigned to the first possible layer, given the constraint that all its predecessors must be in earlier layers.
`'alap'`	As late as possible. Each node is assigned to the last possible layer, given the constraint that all its successors must be in later layers.

This option is available only when `method` is `'layered'`.

Example: `layout(H,'layered','AssignLayers','alap')`

Subspace

`Dimension` — Dimension of embedded subspace
positive scalar integer

Dimension of embedded subspace, specified as the comma-separated pair consisting of `'Dimension'` and a positive scalar integer.

- The default integer value is `min([100, numnodes(G)])`.
- For the `'subspace'` layout, the integer must be greater than or equal to 2.
- For the `'subspace3'` layout, the integer must be greater than or equal to 3.
- In both cases, the integer must be less than the number of nodes.

This option is available only when `method` is `'subspace'` or `'subspace3'`.

Example: `layout(H,'subspace','Dimension',d)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Circle

`Center` — Center node in circular layout
node index | node name

Center node in circular layout, specified as the comma-separated pair consisting of `'Center'` and one of the values in this table.

Value	Example
Scalar node index	`1`
Character vector node name	`'A'`
String scalar node name	`"A"`

This option is available only when `method` is `'circle'`.

Example: `layout(H,'circle','Center',3)` places node three at the center.

Example: `layout(H,'circle','Center','Node1')` places the node named `'Node1'` at the center.

Tips

- Use the `Layout` name-value pair to change the layout of a graph when you plot it. For example, `plot(G,'Layout','circle')` plots the graph `G` with a circular layout.
- When using the `'force'` or `'force3'` layout methods, a best practice is to use more iterations with the algorithm instead of using `XStart`, `YStart`, and `ZStart` to restart the algorithm using previous outputs. The result of executing the algorithm with 100 iterations is different in comparison to executing 50 iterations, and then restarting the algorithm from the ending positions to execute 50 more iterations.

References

[1] Fruchterman, T., and E. Reingold,. “Graph Drawing by Force-directed Placement.” *Software — Practice & Experience*. Vol. 21 (11), 1991, pp. 1129–1164.

[2] Gansner, E., E. Koutsofios, S. North, and K.-P Vo. “A Technique for Drawing Directed Graphs.” *IEEE Transactions on Software Engineering*. Vol.19, 1993, pp. 214–230.

[3] Barth, W., M. Juenger, and P. Mutzel. “Simple and Efficient Bilayer Cross Counting.” *Journal of Graph Algorithms and Applications*. Vol.8 (2), 2004, pp. 179–194.

[4] Brandes, U., and B. Koepf. “Fast and Simple Horizontal Coordinate Assignment.” *LNCS*. Vol. 2265, 2002, pp. 31–44.

[5] Y. Koren. “Drawing Graphs by Eigenvectors: Theory and Practice.” *Computers and Mathematics with Applications*. Vol. 49, 2005, pp. 1867–1888.

See Also

Functions
`digraph` | `graph` | `highlight` | `labeledge` | `labelnode` | `plot`

Objects
`GraphPlot`

Topics
“Graph Plotting and Customization”

Introduced in R2015b

plot

Plot graph nodes and edges

Syntax

```
plot(G)
plot(G,LineSpec)
plot( ___ ,Name,Value)
plot(ax, ___ )
h = plot( ___ )
```

Description

`plot(G)` plots the nodes and edges in graph `G`.

`plot(G,LineSpec)` sets the line style, marker symbol, and color. For example, `plot(G,'-or')` uses red circles for the nodes and red lines for the edges.

`plot( ___ ,Name,Value)` uses additional options specified by one or more Name-Value pair arguments using any of the input argument combinations in previous syntaxes. For example, `plot(G,'Layout','circle')` plots a circular ring layout of the graph, and `plot(G,'XData',X,'YData',Y,'ZData',Z)` specifies the `(X,Y,Z)` coordinates of the graph nodes.

`plot(ax, ___ )` plots into the axes specified by `ax` instead of into the current axes (`gca`). The option, `ax`, can precede any of the input argument combinations in previous syntaxes.

`h = plot( ___ )` returns a `GraphPlot` object. Use this object to inspect and adjust the properties of the plotted graph.

Examples

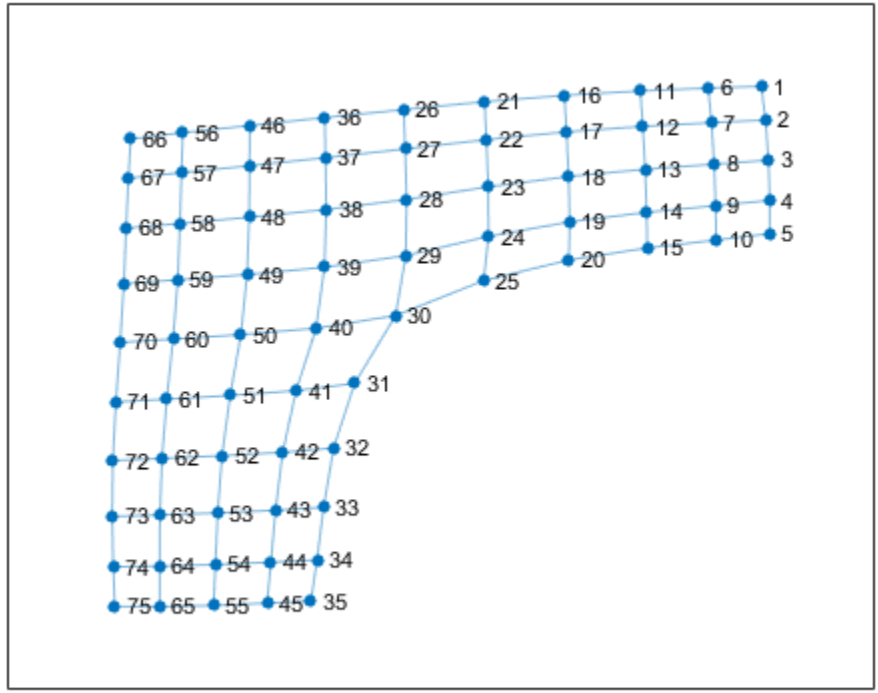
Plot Graph

Create a graph using a sparse adjacency matrix, and then plot the graph.

```
n = 10;
A = delsq(numgrid('L',n+2));
G = graph(A,'omitselfloops')
```

```
G 
  graph with properties:

    Edges: [130x2 table]
    Nodes: [75x0 table]
```

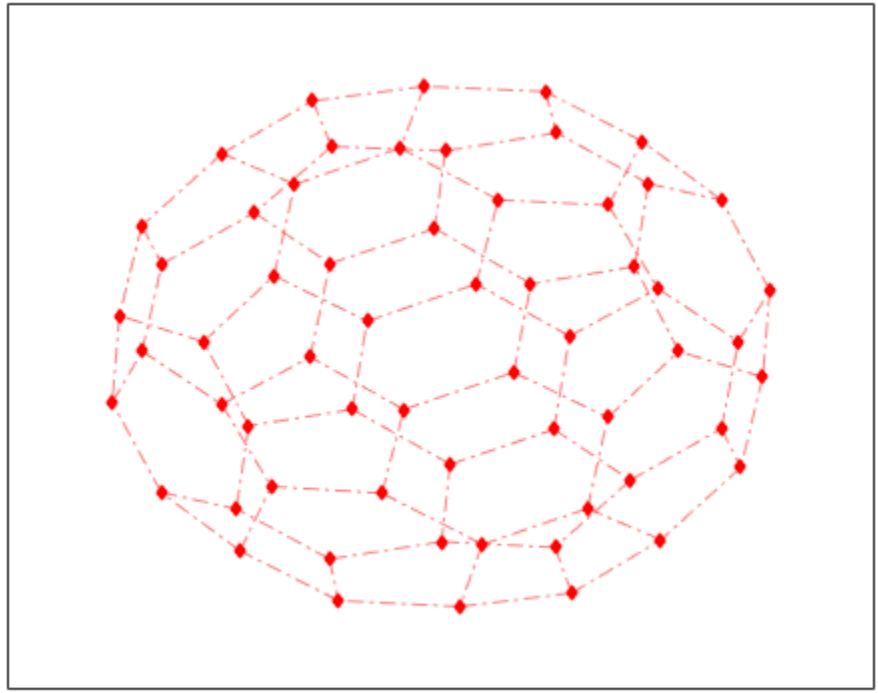
```
plot(G)
```



Plot Graph Using Line Specifier

Create and plot a graph. Specify the `LineSpec` input to change the `Marker`, `NodeColor`, and/or `LineStyle` of the graph plot.

```
G = graph(bucky);
plot(G,'-.dr','NodeLabel',{})
```



Plot Graph with Specified Layout

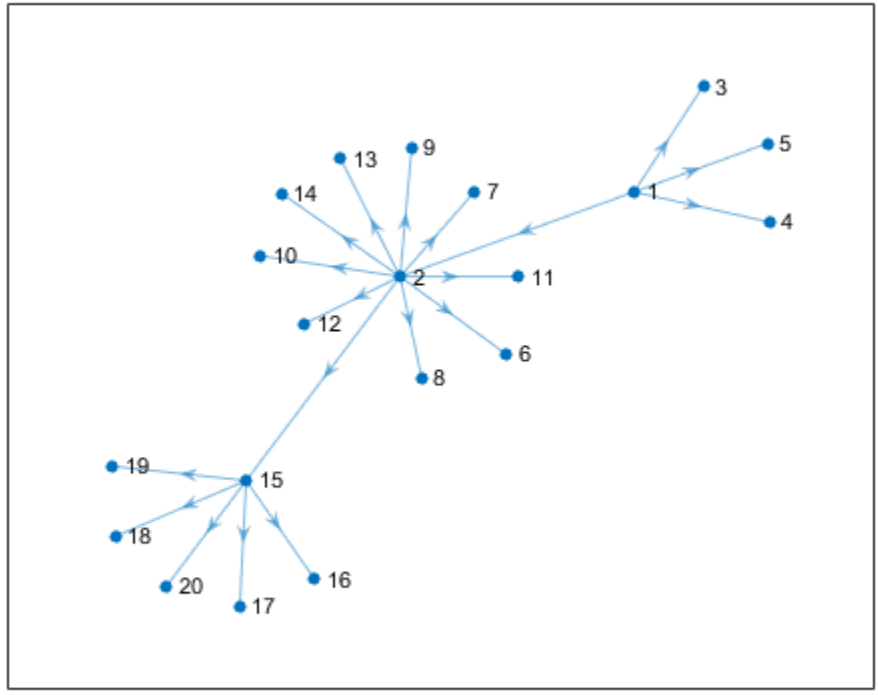
Create a directed graph, and then plot the graph using the `'force'` layout.

```
G = digraph(1,2:5);
G = addedge(G,2,6:15);
G = addedge(G,15,16:20)
```

```
G = 
  digraph with properties:

    Edges: [19x1 table]
```

```
    Nodes: [20x0 table]
```

```
plot(G,'Layout','force')
```



Custom Graph Node Coordinates

Create a weighted graph.

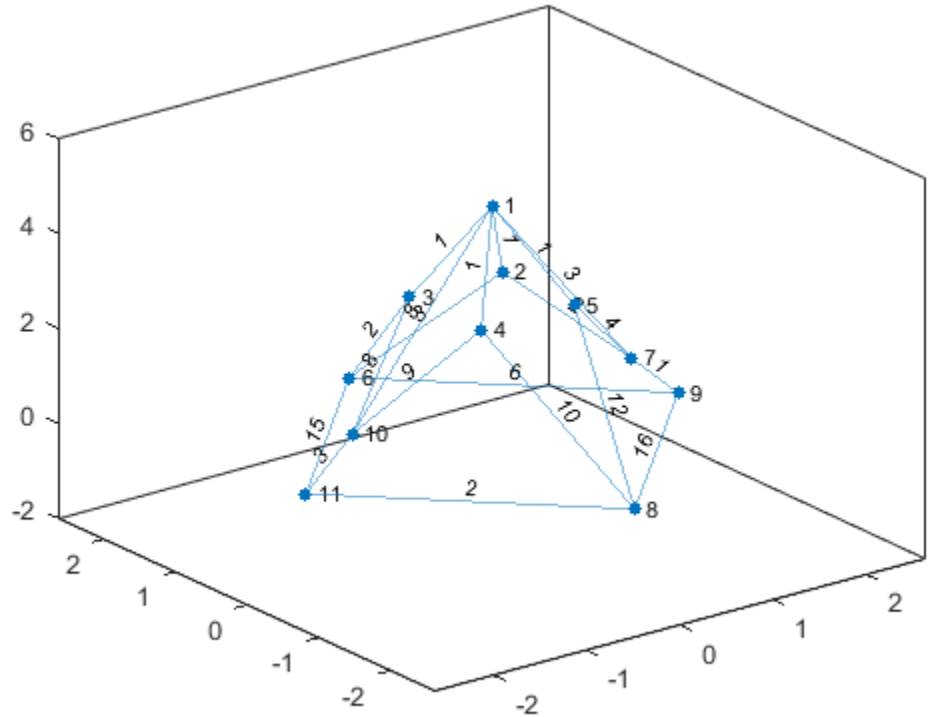
```
s = [1 1 1 1 1 2 2 7 7 9 3 3 1 4 10 8 4 5 6 8];
t = [2 3 4 5 7 6 7 5 9 6 6 10 10 10 11 11 8 8 11 9];
```

```
weights = [1 1 1 1 3 3 2 4 1 6 2 8 8 9 3 2 10 12 15 16];
G = graph(s,t,weights)
```

```
G 
  graph with properties:

    Edges: [20x2 table]
    Nodes: [11x0 table]
```

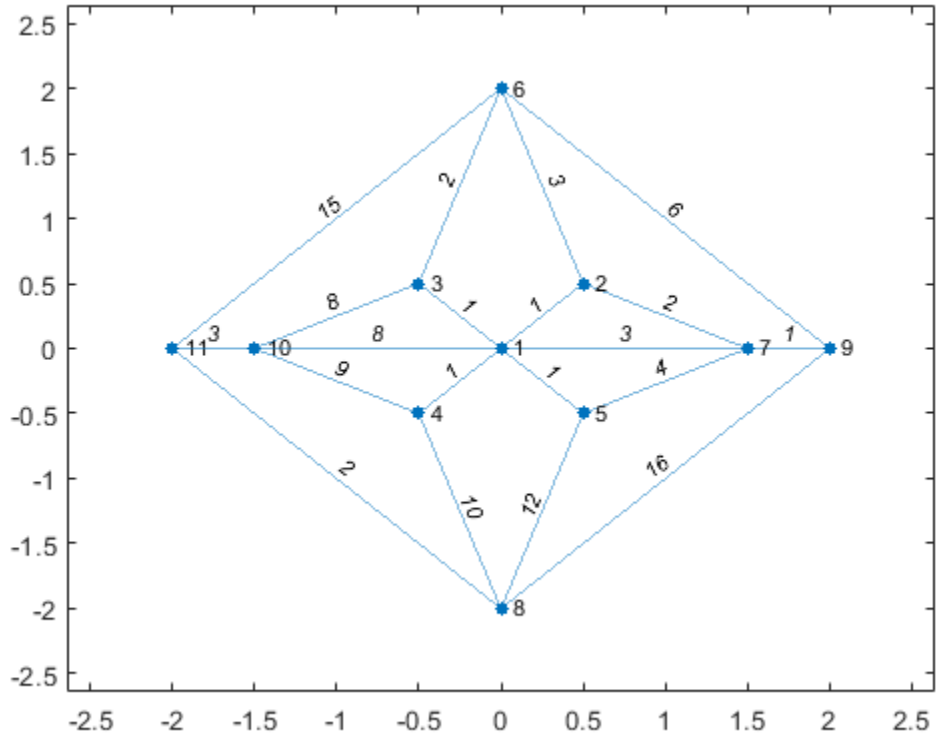
Plot the graph using custom coordinates for the nodes. The x-coordinates are specified using `XData`, the y-coordinates are specified using `YData`, and the z-coordinates are specified using `ZData`. Use `EdgeLabel` to label the edges using the edge weights.

```
x = [0 0.5 -0.5 -0.5 0.5 0 1.5 0 2 -1.5 -2];
y = [0 0.5 0.5 -0.5 -0.5 2 0 -2 0 0 0];
z = [5 3 3 3 3 0 1 0 0 1 0];
plot(G,'XData',x,'YData',y,'ZData',z,'EdgeLabel',G.Edges.Weight)
```



View the graph from above.

```
view(2)
```



Edge Line Width Proportional to Edge Weight

Create a weighted graph.

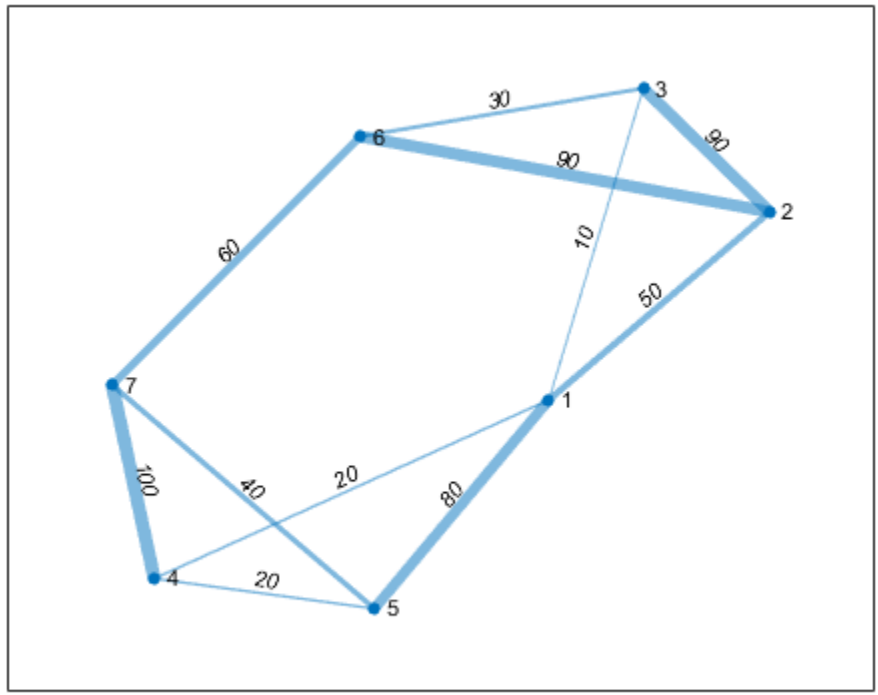
```
s = [1 1 1 1 2 2 3 4 4 5 6];
t = [2 3 4 5 3 6 6 5 7 7 7];
weights = [50 10 20 80 90 90 30 20 100 40 60];
G = graph(s,t,weights)
```

```
G =
  graph with properties:
```

```
Edges: [11x2 table]
Nodes: [7x0 table]
```

Plot the graph, labeling the edges with their weights, and making the width of the edges proportional to their weights. Use a rescaled version of the edge weights to determine the width of each edge, such that the widest line has a width of 5.

```
LWidths = 5*G.Edges.Weight/max(G.Edges.Weight);
plot(G,'EdgeLabel',G.Edges.Weight,'LineWidth',LWidths)
```



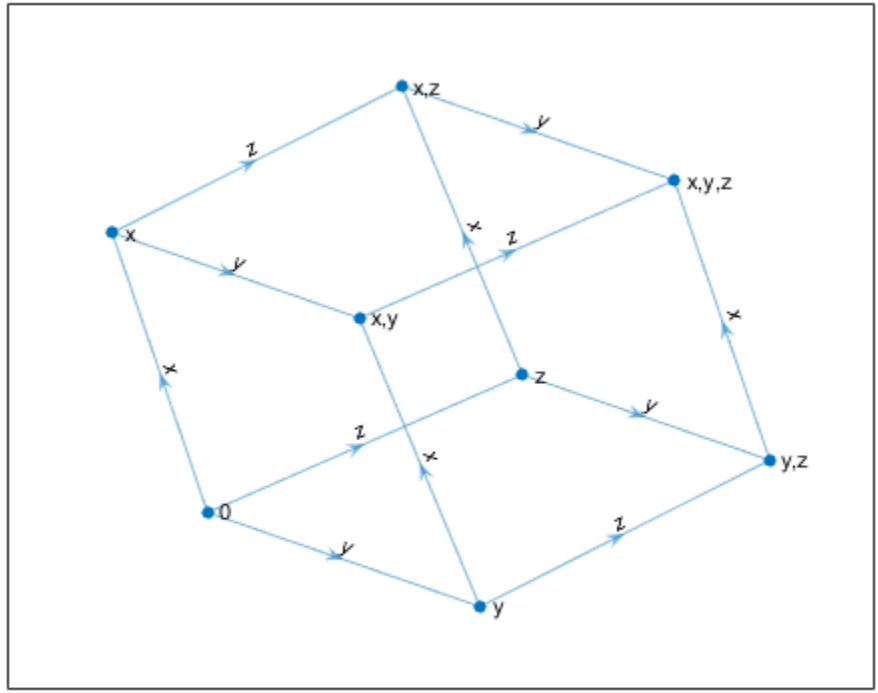
Label Graph Nodes and Edges

Create a directed graph. Plot the graph with custom labels for the nodes and edges.

```
s = [1 1 1 2 2 3 3 4 4 5 6 7];
t = [2 3 4 5 6 5 7 6 7 8 8 8];
G = digraph(s,t)
```

```
G 
  digraph with properties:

    Edges: [12x1 table]
    Nodes: [8x0 table]
```

```
eLabels = {'x' 'y' 'z' 'y' 'z' 'x' 'z' 'x' 'y' 'z' 'y' 'x'};
nLabels = {'{0}','{x}','{y}','{z}','{x,y}','{x,z}','{y,z}','{x,y,z}'};
plot(G,'Layout','force','EdgeLabel',eLabels,'NodeLabel',nLabels)
```



Adjust GraphPlot Properties

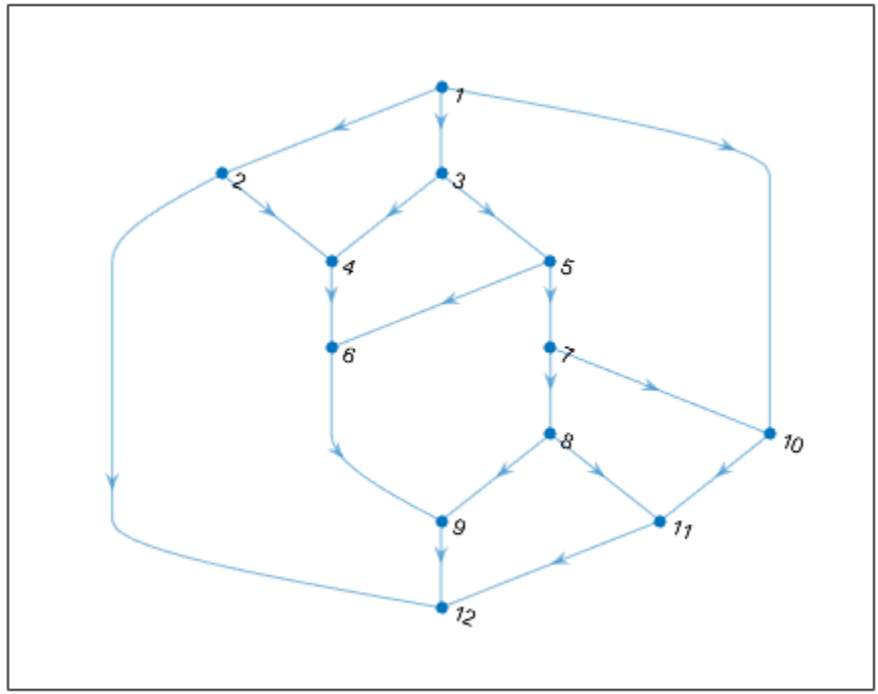
Create and plot a directed graph. Specify an output argument to `plot` to return a handle to the `GraphPlot` object.

```
s = [1 1 1 2 2 3 3 4 5 5 6 7 7 8 8 9 10 11];
t = [2 3 10 4 12 4 5 6 6 7 9 8 10 9 11 12 11 12];
G = digraph(s,t)
```

```
G = 
  digraph with properties:
```

```
    Edges: [18x1 table]
    Nodes: [12x0 table]
```

```
p = plot(G)
```

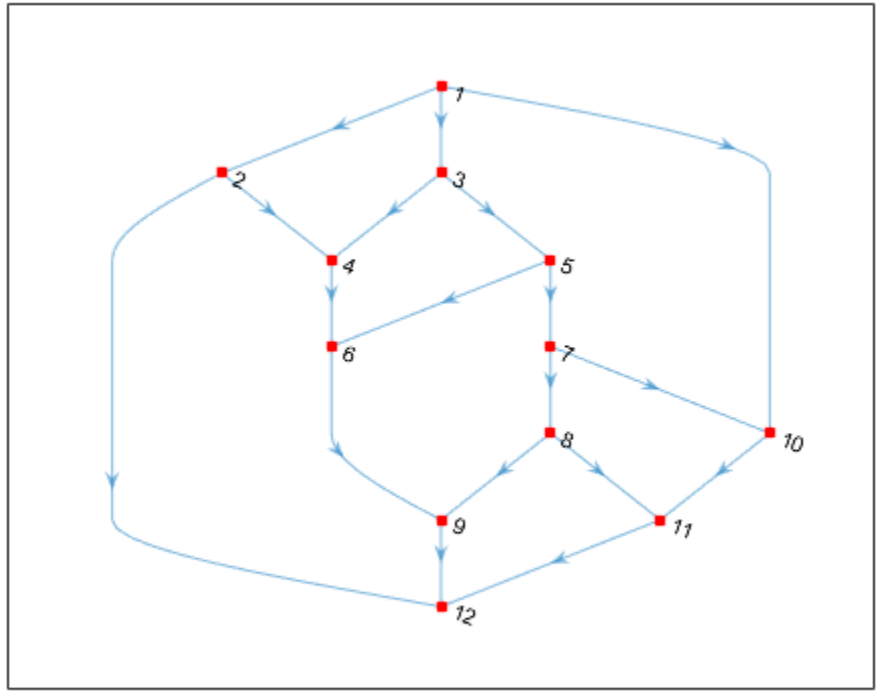



```
p =
  GraphPlot with properties:

     NodeColor: [0 0.4470 0.7410]
    MarkerSize: 4
        Marker: 'o'
     EdgeColor: [0 0.4470 0.7410]
```

```
    LineWidth: 0.5000
    LineStyle: '-'
    NodeLabel: {1x12 cell}
    EdgeLabel: {}
        XData: [2.5000 1.5000 2.5000 2 3 2 3 3 2.5000 4 3.5000 2.5000]
        YData: [7 6 6 5 5 4 4 3 2 3 2 1]
        ZData: [0 0 0 0 0 0 0 0 0 0 0 0]

  Show all properties
```

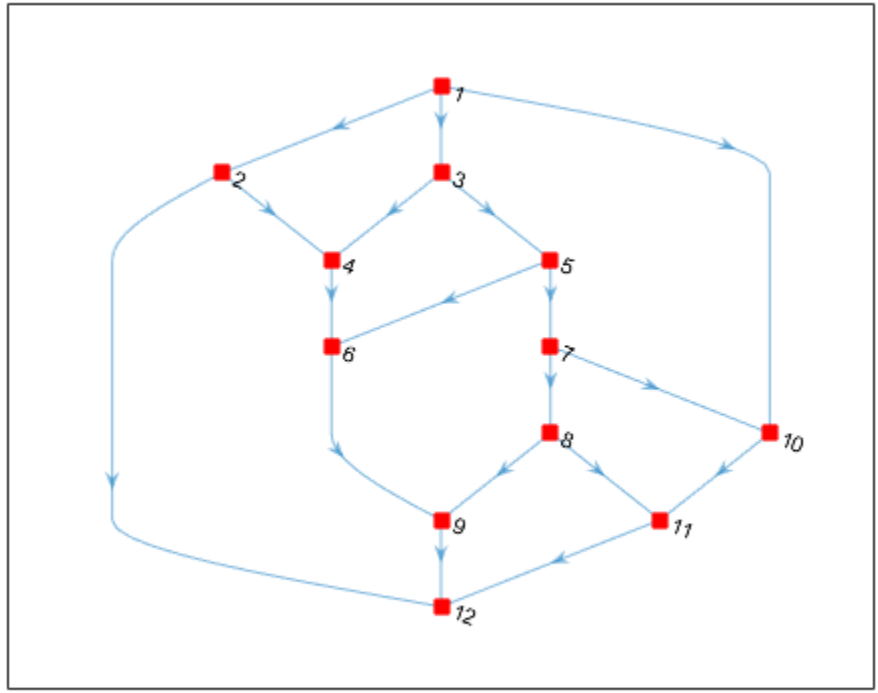
Change the color and marker of the nodes.

```
p.Marker = 's';
p.NodeColor = 'r';
```



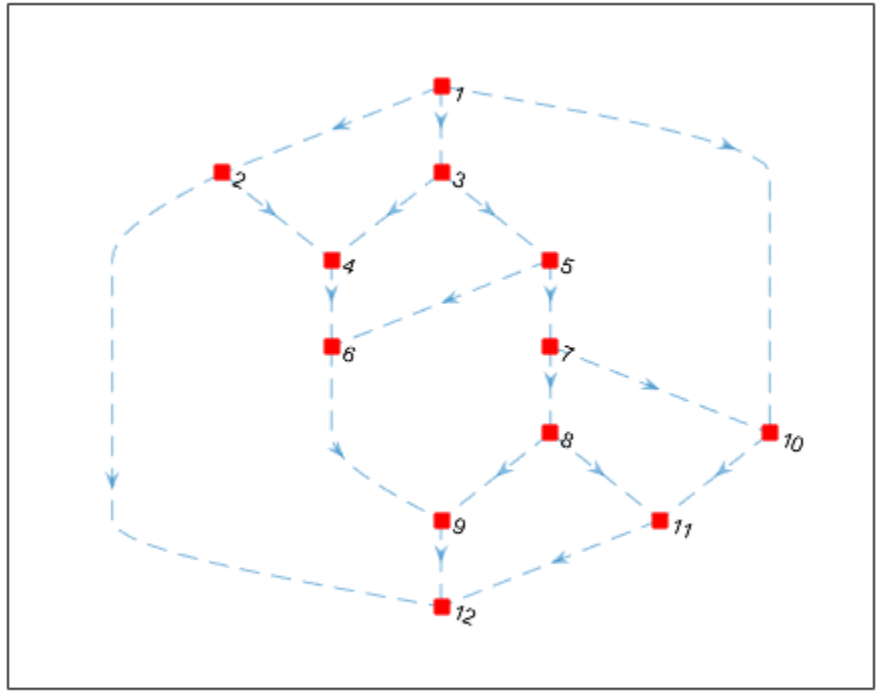
Increase the size of the nodes.

```
p.MarkerSize = 7;
```



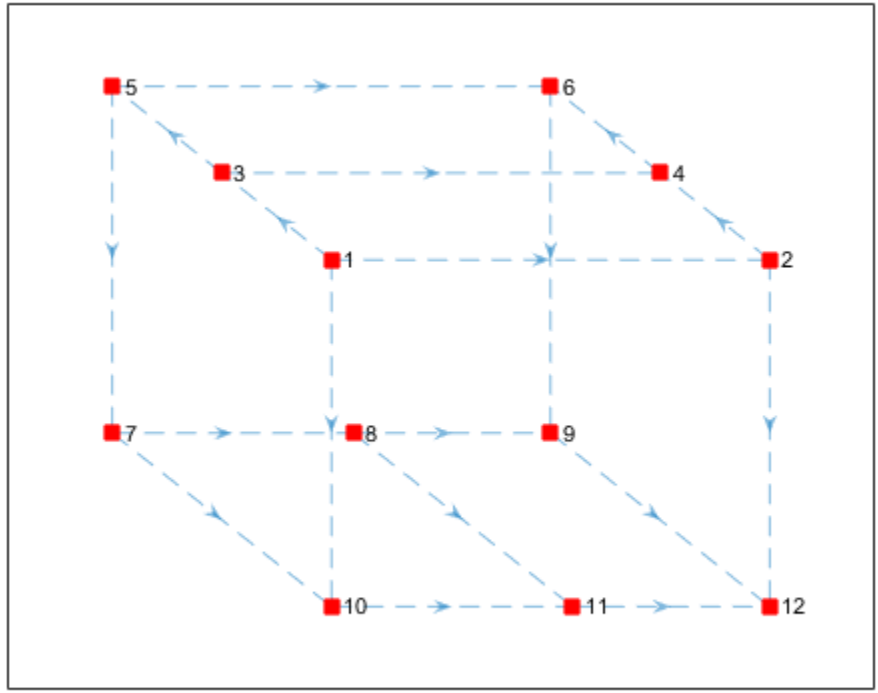
Change the line style of the edges.

```
p.LineStyle = '--';
```



Change the *x* and *y* coordinates of the nodes.

```
p.XData = [2 4 1.5 3.5 1 3 1 2.1 3 2 3.1 4];
p.YData = [3 3 3.5 3.5 4 4 2 2 2 1 1 1];
```



Input Arguments

G — Input graph
graph object | digraph object

Input graph, specified as either a `graph` or `digraph` object. Use `graph` to create an undirected graph or `digraph` to create a directed graph.

Example: `G = graph(1,2)`

Example: `G = digraph([1 2],[2 3])`

`LineSpec` — Line style, marker symbol, and color
character vector | string vector

Line style, marker symbol, and color, specified as a character vector or string vector of symbols. The symbols can appear in any order, and you can omit one or more of the characteristics. If you omit the line style, then the plot shows solid lines for the graph edges.

Example: `'--or'` uses red circle node markers and red dashed lines as edges.

Example: `'r*'` uses red asterisk node markers and solid red lines as edges.

Symbol	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Symbol	Marker
`o`	Circle
`+`	Plus sign
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Pentagram
`h`	Hexagram

Symbol	Color
y	yellow
m	magenta
c	cyan
r	red
g	green
b	blue
w	white
k	black

ax — Axes object
object

Axes object. If you do not specify an axes object, then `plot` uses the current axes (`gca`).

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `p = plot(G,'EdgeColor','r','NodeColor','k','LineStyle','--')`

The graph properties listed here are only a subset. For a complete list, see GraphPlot.

`ArrowSize` — Arrow size
positive value

Note `ArrowSize` only affects the display of directed graphs created using `digraph`.

Arrow size, specified as the comma-separated pair consisting of `'ArrowSize'` and a positive value in point units. The default value of `ArrowSize` is 7 for graphs with 100 or fewer nodes, and 4 for graphs with more than 100 nodes.

Example: 15

EdgeCData — Color data of edge lines
vector

Color data of edge lines, specified as the comma-separated pair consisting of `'EdgeCData'` and a vector with length equal to the number of edges in the graph. The values in `EdgeCData` map linearly to the colors in the current colormap, resulting in different colors for each edge in the plotted graph.

EdgeColor — Edge color
`[0 0.4470 0.7410]` (default) | RGB triplet | hexadecimal color code | color name | matrix | `'flat'` | `'none'`

Edge color, specified as the comma-separated pair consisting of `'EdgeColor'` and one of these values:

- `'none'` — Edges are not drawn.
- `'flat'` — Color of each edge depends on the value of `EdgeCData`.
- matrix — Each row is an RGB triplet representing the color of one edge. The size of the matrix is `numedges(G)`-by-3.
- RGB triplet, hexadecimal color code, or color name — Edges use the specified color.

  RGB triplets and hexadecimal color codes are useful for specifying custom colors.

  - An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
  - A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

  Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `plot(G,'EdgeColor','r')` creates a graph plot with red edges.

`EdgeLabel` — Edge labels

`{}` (default) | vector | cell array of character vectors | string array

Edge labels, specified as the comma-separated pair consisting of `'EdgeLabel'` and a numeric vector, cell array of character vectors, or string array. The length of `EdgeLabel`

must be equal to the number of edges in the graph. By default `EdgeLabel` is an empty cell array (no edge labels are displayed).

Example: `{'A', 'B', 'C'}`

Example: `[1 2 3]`

Example: `plot(G,'EdgeLabel',G.Edges.Weight)` labels the graph edges with their weights.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `cell` | `string`

`Layout` — Graph layout method
`'auto'` (default) | `'circle'` | `'force'` | `'layered'` | `'subspace'` | `'force3'` | `'subspace3'`

Graph layout method, specified as the comma-separated pair consisting of `'Layout'` and one of the options in the table. The table also lists compatible name-value pairs to further refine each layout method. See the `layout` reference page for more information on these layout-specific name-value pairs.

Option	Description	Layout-Specific Name-Value Pairs
`'auto'` (default)	Automatic choice of layout method based on the size and structure of the graph.	—
`'circle'`	Circular layout. Places the graph nodes on a circle centered at the origin with radius 1.	`'Center'` — Center node in circular layout

Option	Description	Layout-Specific Name-Value Pairs
`'force'`	Force-directed layout [1]. Uses attractive forces between adjacent nodes and repulsive forces between distant nodes.	`'Iterations'` — Number of force-directed layout iterations `'WeightEffect'` — Effect of edge weights on layout `'UseGravity'` — Gravity toggle for layouts with multiple components `'XStart'` — Starting *x*-coordinates for nodes `'YStart'` — Starting *y*-coordinates for nodes
`'layered'`	Layered node layout [2], [3], [4]. Places the graph nodes into a set of layers, revealing hierarchical structure. By default the layers progress downwards (the arrows of a directed acyclic graph point down).	`'Direction'` — Direction of layers `'Sources'` — Nodes to include in the first layer `'Sinks'` — Nodes to include in the last layer `'AssignLayers'` — Layer assignment method
`'subspace'`	Subspace embedding node layout [5]. Plots the graph nodes in a high-dimensional embedded subspace, and then projects the positions back into 2-D. By default the subspace dimension is either 100 or the total number of nodes, whichever is smaller.	`'Dimension'` — Dimension of embedded subspace

Option	Description	Layout-Specific Name-Value Pairs
`'force3'`	3-D force-directed layout.	`'Iterations'` — Number of force-directed layout iterations `'WeightEffect'` — Effect of edge weights on layout `'UseGravity'` — Gravity toggle for layouts with multiple components `'XStart'` — Starting *x*-coordinates for nodes `'YStart'` — Starting *y*-coordinates for nodes `'ZStart'` — Starting *z*-coordinates for nodes
`'subspace3'`	3-D subspace embedding layout.	`'Dimension'` — Dimension of embedded subspace

Example: `plot(G,'Layout','force3','Iterations',10)`

Example: `plot(G,'Layout','subspace','Dimension',50)`

Example: `plot(G,'Layout','layered')`

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'` | cell array | string vector

Line style, specified as the comma-separated pair consisting of `'LineStyle'` and one of the line styles listed in this table, or as a cell array or string vector of such values. Specify a cell array of character vectors or string vector to use different line styles for each edge.

Character(s)	Line Style	Resulting Line
`'-'`	Solid line	

Character(s)	Line Style	Resulting Line
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Edge line width
`0.5` (default) | positive value | vector

Edge line width, specified as the comma-separated pair consisting of `'LineWidth'` and a positive value in point units or a vector of such values. Specify a vector to use a different line width for each edge in the graph.

Example: `0.75`

`Marker` — Node marker symbol
`'o'` (default) | character vector | cell array | string vector

Node marker symbol, specified as the comma-separated pair consisting of `'Marker'` and one of the character vectors listed in this table, or as a cell array or string vector of such values. The default is to use circular markers for the graph nodes. Specify a cell array of character vectors or string vector to use different markers for each node.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle

Value	Description
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'+'`

Example: `'diamond'`

`MarkerSize` — Node marker size
positive value | vector

Node marker size, specified as the comma-separated pair consisting of `'MarkerSize'` and a positive value in point units or as a vector of such values. Specify a vector to use different marker sizes for each node in the graph. The default value of `MarkerSize` is 4 for graphs with 100 or fewer nodes, and 2 for graphs with more than 100 nodes.

Example: `10`

`NodeCData` — Color data of node markers
vector

Color data of node markers, specified as the comma-separated pair consisting of `'NodeCData'` and a vector with length equal to the number of nodes in the graph. The values in `NodeCData` map linearly to the colors in the current colormap, resulting in different colors for each node in the plotted graph.

`NodeColor` — Node color
`[0 0.4470 0.7410]` (default) | RGB triplet | hexadecimal color code | color name | matrix | `'flat'` | `'none'`

Node color, specified as the comma-separated pair consisting of `'NodeColor'` and one of these values:

- `'none'` — Nodes are not drawn.
- `'flat'` — Color of each node depends on the value of `NodeCData`.
- matrix — Each row is an RGB triplet representing the color of one node. The size of the matrix is `numnodes(G)`-by-3.

- RGB triplet, hexadecimal color code, or color name — Nodes use the specified color.

  RGB triplets and hexadecimal color codes are useful for specifying custom colors.

  - An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
  - A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

  Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

  Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `plot(G,'NodeColor','k')` creates a graph plot with black nodes.

`NodeLabel` — Node labels

node IDs (default) | vector | cell array of character vectors | string array

Node labels, specified as the comma-separated pair consisting of `'NodeLabel'` and a numeric vector, cell array of character vectors, or string array. The length of `NodeLabel` must be equal to the number of nodes in the graph. By default `NodeLabel` is a cell array containing the node IDs for the graph nodes:

- For nodes without names (that is, `G.Nodes` does not contain a `Name` variable), the node labels are the values `unique(G.Edges.EndNodes)` contained in a cell array.
- For named nodes, the node labels are `G.Nodes.Name'`.

Example: `{'A', 'B', 'C'}`

Example: `[1 2 3]`

Example: `plot(G,'NodeLabel',G.Nodes.Name)` labels the nodes with their names.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `cell` | `string`

`XData` — x-coordinate of nodes

vector

Note `XData` and `YData` must be specified together so that each node has a valid (*x*,*y*) coordinate. Optionally, you can also specify `ZData` for 3-D coordinates.

---

x-coordinate of nodes, specified as the comma-separated pair consisting of `'XData'` and a vector with length equal to the number of nodes in the graph.

`YData` — y-coordinate of nodes
vector

Note `XData` and `YData` must be specified together so that each node has a valid (*x*,*y*) coordinate. Optionally, you can also specify `ZData` for 3-D coordinates.

---

y-coordinate of nodes, specified as the comma-separated pair consisting of `'YData'` and a vector with length equal to the number of nodes in the graph.

`ZData` — z-coordinate of nodes
vector

Note `XData` and `YData` must be specified together so that each node has a valid (*x*,*y*) coordinate. Optionally, you can also specify `ZData` for 3-D coordinates.

---

z-coordinate of nodes, specified as the comma-separated pair consisting of `'ZData'` and a vector with length equal to the number of nodes in the graph.

Output Arguments

h — Graph plot
`GraphPlot` object

Graph plot, returned as an object. For more information, see `GraphPlot`.

Compatibility Considerations

Self-loop display change

Behavior changed in R2018a

Self-loops in the plot of a simple graph are now shaped like a leaf or teardrop. In previous releases, self-loops were displayed as circles.

References

[1] Fruchterman, T., and E. Reingold. “Graph Drawing by Force-directed Placement.” *Software — Practice & Experience*. Vol. 21 (11), 1991, pp. 1129–1164.

[2] Gansner, E., E. Koutsofios, S. North, and K.-P Vo. “A Technique for Drawing Directed Graphs.” *IEEE Transactions on Software Engineering*. Vol.19, 1993, pp. 214–230.

[3] Barth, W., M. Juenger, and P. Mutzel. “Simple and Efficient Bilayer Cross Counting.” *Journal of Graph Algorithms and Applications*. Vol.8 (2), 2004, pp. 179–194.

[4] Brandes, U., and B. Koepf. “Fast and Simple Horizontal Coordinate Assignment.” *LNCS*. Vol. 2265, 2002, pp. 31–44.

[5] Y. Koren. “Drawing Graphs by Eigenvectors: Theory and Practice.” *Computers and Mathematics with Applications*. Vol. 49, 2005, pp. 1867–1888.

See Also

Functions
`digraph` | `graph` | `highlight` | `labeledge` | `labelnode` | `layout`

Objects
`GraphPlot`

Properties
GraphPlot

Topics

“Graph Plotting and Customization”
“Add Node Properties to Graph Plot Data Cursor”

Introduced in R2015b

rmedge

Remove edge from graph

Syntax

```
H = rmedge(G,s,t)
H = rmedge(G,idx)
```

Description

`H = rmedge(G,s,t)` removes the edges specified by the node pairs `s` and `t` from graph `G`. If there are multiple edges specified by `s` and `t`, then they are all removed.

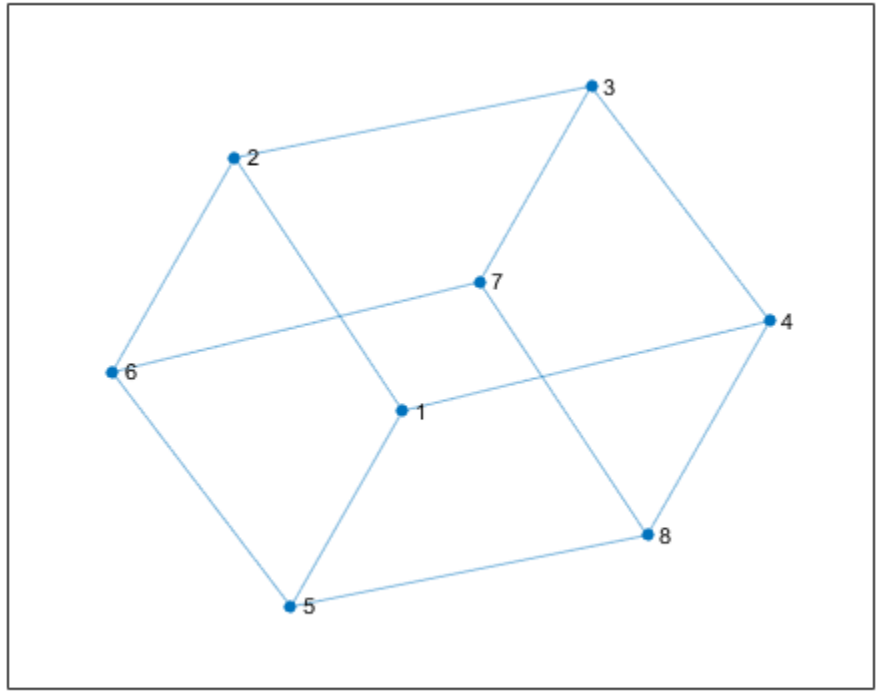
`H = rmedge(G,idx)` specifies which edges to remove with edge indices `idx`. The edge indices are row numbers in the `G.Edges` table.

Examples

Remove Edges with Specified End Nodes

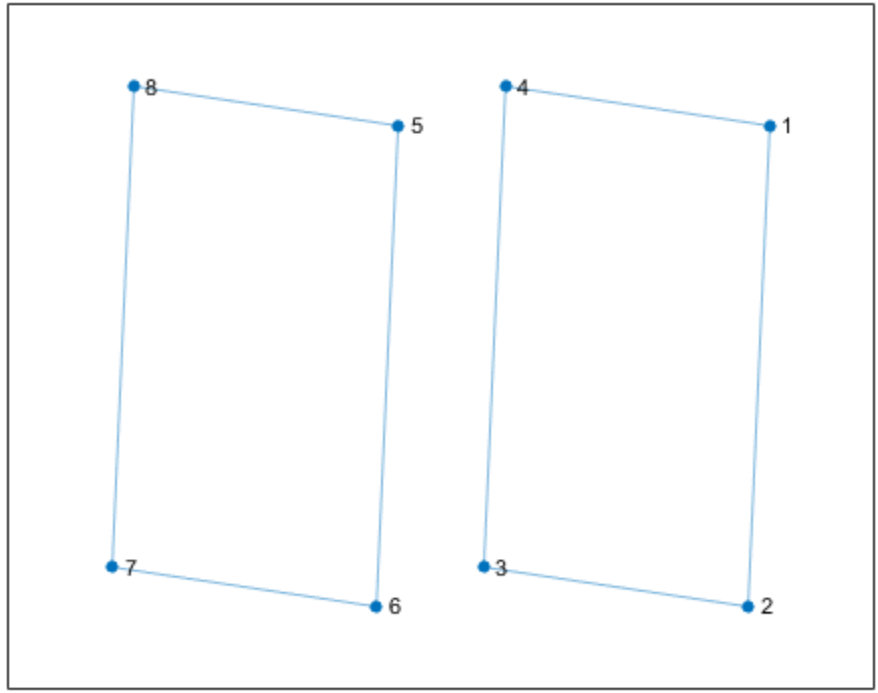
Create and plot a graph.

```
s = [1 1 1 2 2 3 3 4 5 5 6 7];
t = [2 4 5 3 6 4 7 8 6 8 7 8];
G = graph(s,t);
plot(G)
```



Remove several edges from the graph and plot the result.

```
G = rmedge(G,[1 2 3 4],[5 6 7 8]);
plot(G)
```



Remove Edge with Specified Index

Create a graph and view the edge list.

```
s = {'BOS' 'NYC' 'NYC' 'NYC' 'LAX'};
t = {'NYC' 'LAX' 'DEN' 'LAS' 'DCA'};
G = digraph(s,t);
G.Edges
```

```
ans=5×1 table
         EndNodes
    ________________
```

```
    {'BOS'}    {'NYC'}
    {'NYC'}    {'LAX'}
    {'NYC'}    {'DEN'}
    {'NYC'}    {'LAS'}
    {'LAX'}    {'DCA'}
```

Remove the edge between nodes `'NYC'` and `'DEN'` using the edge index.

```
G = rmedge(G,3);
G.Edges
```

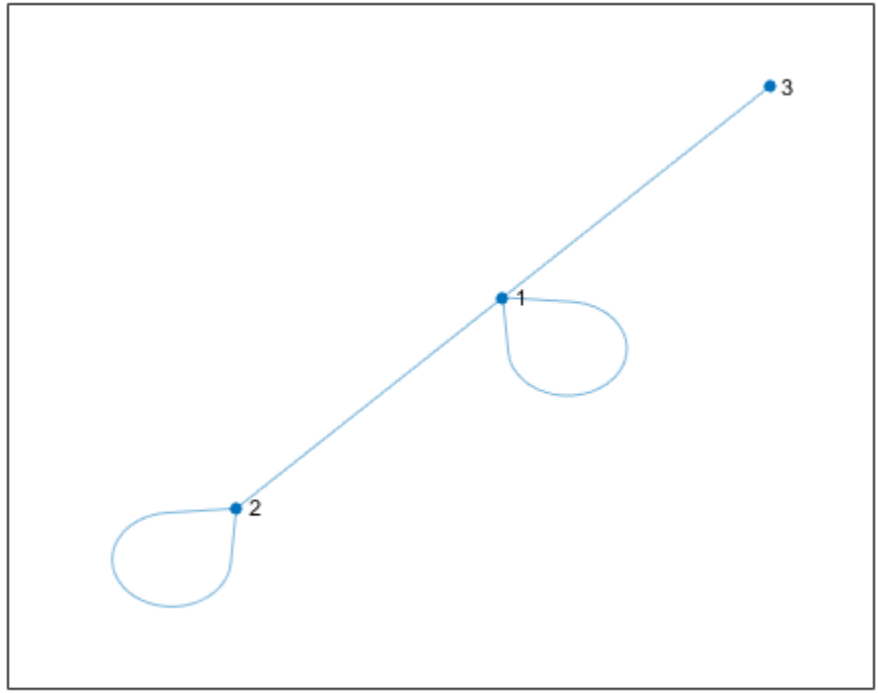
```
ans=4×1 table
         EndNodes
    ________________

    {'BOS'}    {'NYC'}
    {'NYC'}    {'LAX'}
    {'NYC'}    {'LAS'}
    {'LAX'}    {'DCA'}
```

Remove Self-Loops

This example shows how to remove all of the self-loops from a graph. Self-loops are edges that connect a node to itself.

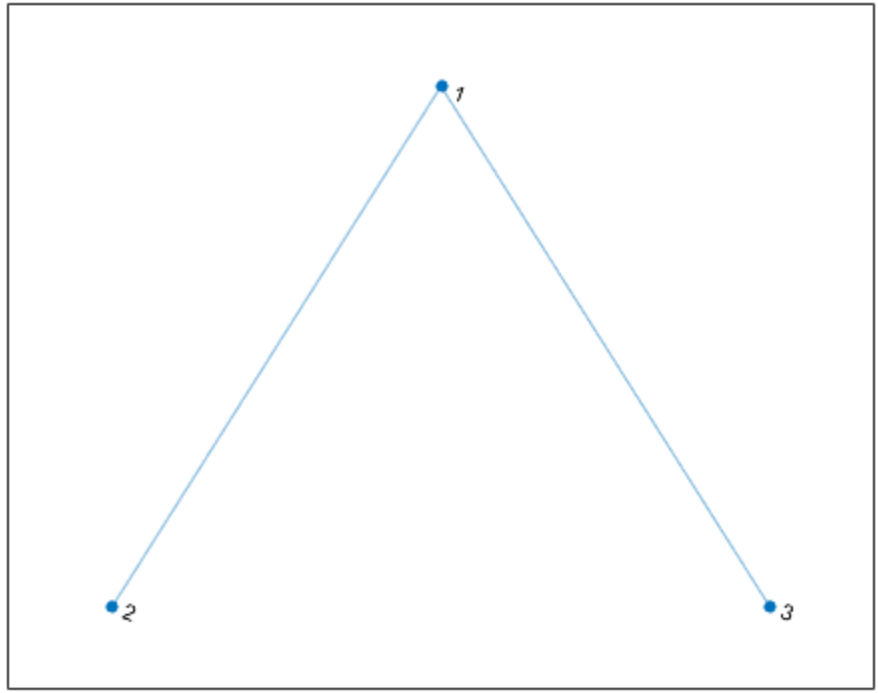
Create a graph that has two self-loops.

```
G = graph([1 1 1 2],[1 2 3 2]);
plot(G)
```



Use `rmedge` to remove all self-loops from the graph. Even though `G` has only two self-loops, this technique removes all self-loops from any directed or undirected graph.

```
G = rmedge(G, 1:numnodes(G), 1:numnodes(G));
plot(G)
```



Input Arguments

G — Input graph
graph object | digraph object

Input graph, specified as either a graph or digraph object. Use graph to create an undirected graph or digraph to create a directed graph.

Example: G = graph(1,2)

Example: G = digraph([1 2],[2 3])

`s,t` — Node pairs (as separate arguments)
node indices | node names

Node pairs, specified as separate arguments of node indices or node names. Similarly located elements in `s` and `t` specify the source and target nodes for edges in the graph.

This table shows the different ways to refer to one or more nodes either by their numeric node indices or by their node names.

Form	Single Node	Multiple Nodes
Node index	Scalar **Example:** `1`	Vector **Example:** `[1 2 3]`
Node name	Character vector **Example:** `'A'`	Cell array of character vectors **Example:** `{'A' 'B' 'C'}`
	String scalar **Example:** `"A"`	String array **Example:** `["A" "B" "C"]`

Example: `G = rmedge(G,1,2)` removes the edge between node 1 and node 2 from graph `G`.

Example: `G = rmedge(G,{'a' 'b'},{'d' 'c'})` removes two edges from graph `G`, the first of which is between node `'a'` and node `'d'`.

`idx` — Edge indices
scalar | vector

Edge indices, specified as a scalar or vector. The edge indices are nonnegative integers that are row numbers in the `G.Edges` table.

Example: `G = rmedge(G,[1 3 5])` removes the first, third, and fifth edges (rows) from `G.Edges`.

Output Arguments

`H` — Output graph
`graph` object | `digraph` object

Output graph, returned as a `graph` or `digraph` object.

See Also

`addedge` | `digraph` | `graph` | `numedges` | `rmnode`

Topics

"Directed and Undirected Graphs"
"Modify Nodes and Edges of Existing Graph"
"Add Graph Node Names, Edge Weights, and Other Attributes"

Introduced in R2015b

rmnode

Remove node from graph

Syntax

```
H = rmnode(G,nodeIDs)
```

Description

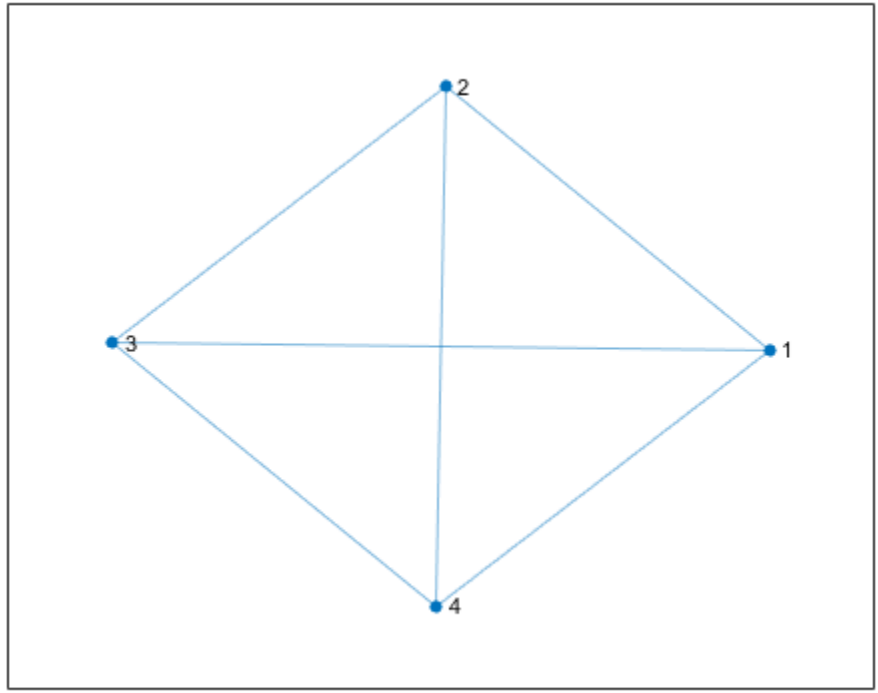
`H = rmnode(G,nodeIDs)` removes the nodes specified by `nodeIDs` from graph `G`. Any edges incident upon the nodes in `nodeIDs` are also removed. `rmnode` refreshes the numbering of the nodes in `H`.

Examples

Remove Node from Graph

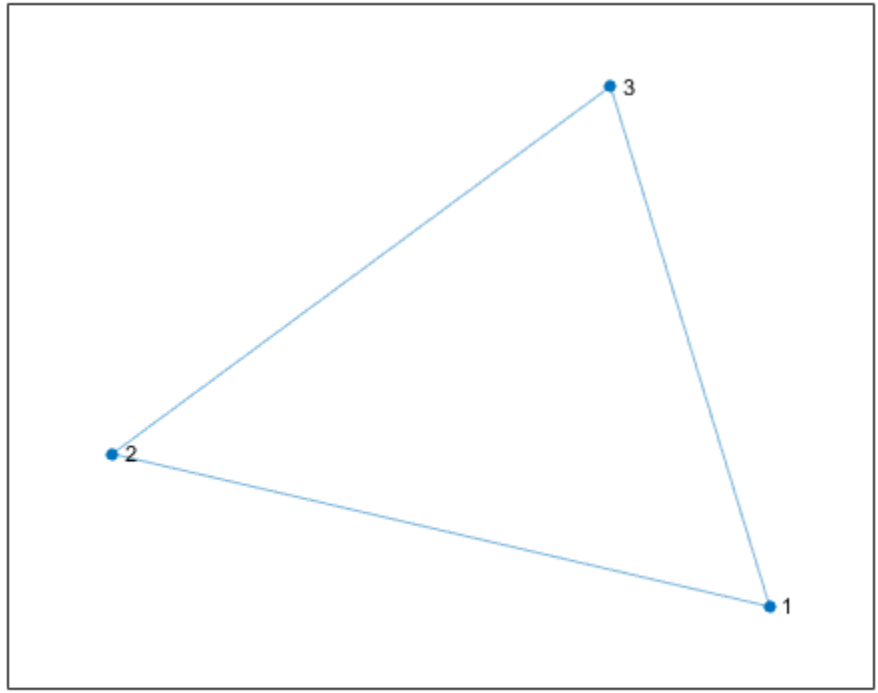
Create and plot a graph.

```
s = [1 1 1 2 2 3];
t = [2 3 4 3 4 4];
G = graph(s,t);
plot(G)
```



Remove node 1 from the graph and plot the result. The nodes in the new graph are automatically renumbered.

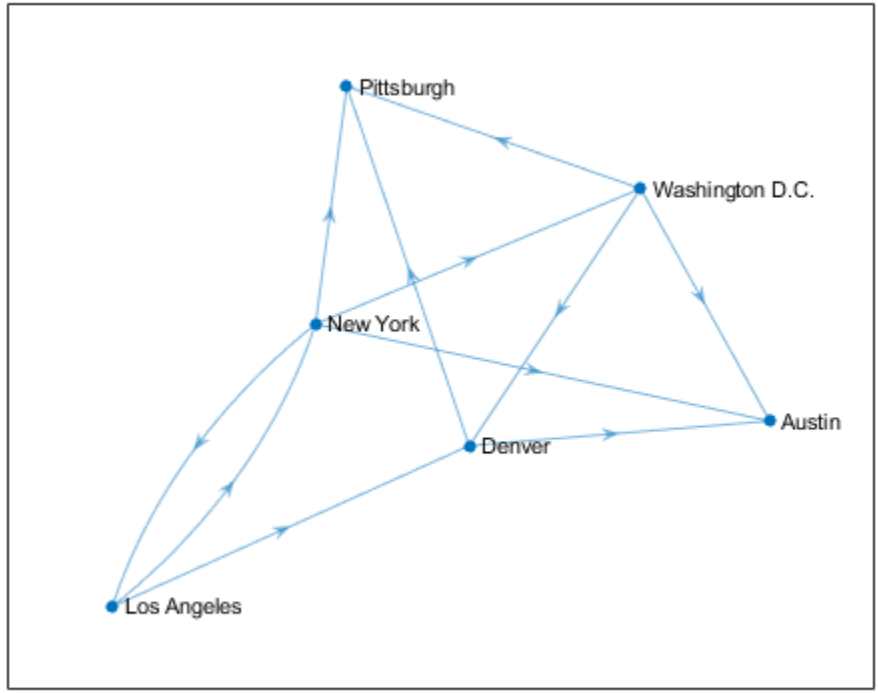
```
G = rmnode(G,1);
plot(G)
```



Remove Several Named Nodes from Graph

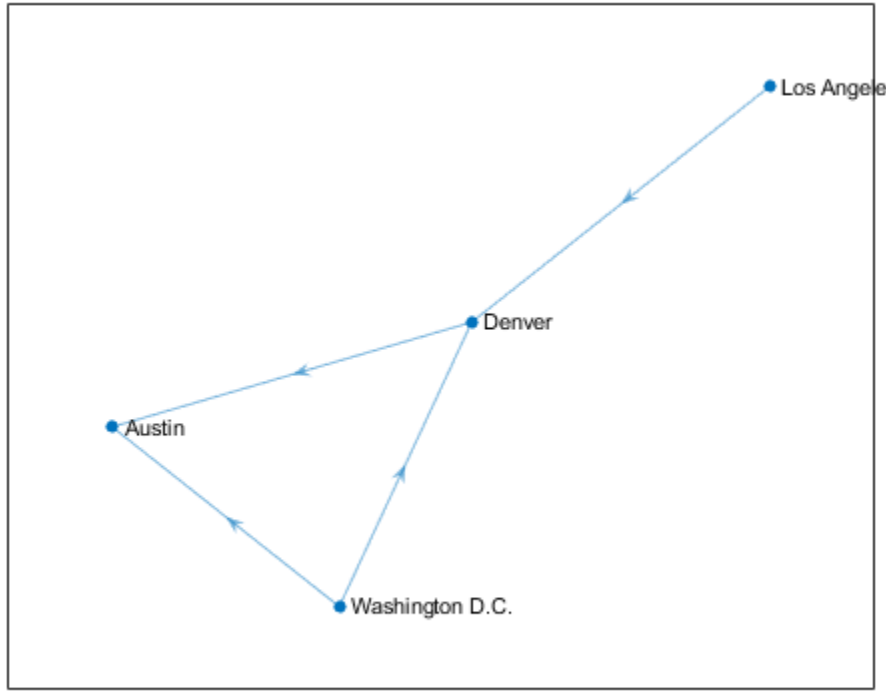
Create and plot a graph with named nodes.

```
s = [1 1 1 1 2 2 3 3 3 5 5];
t = [2 3 4 6 1 5 4 5 6 4 6];
names = {'New York' 'Los Angeles' 'Washington D.C.' 'Pittsburgh' ...
    'Denver' 'Austin'};
G = digraph(s,t,[],names);
plot(G)
```



Remove the nodes `'New York'` and `'Pittsburgh'` from the graph, then replot the result.

```
G = rmnode(G,{'New York' 'Pittsburgh'});
plot(G,'Layout','force')
```



Input Arguments

G — Input graph
graph object | digraph object

Input graph, specified as either a `graph` or `digraph` object. Use `graph` to create an undirected graph or `digraph` to create a directed graph.

Example: `G = graph(1,2)`

Example: `G = digraph([1 2],[2 3])`

`nodeIDs` — Node identifiers
node indices | node names

Node identifiers, specified as one or more node indices or node names.

This table shows the different ways to refer to one or more nodes either by their numeric node indices or by their node names.

Form	Single Node	Multiple Nodes
Node index	Scalar **Example:** `1`	Vector **Example:** `[1 2 3]`
Node name	Character vector **Example:** `'A'`	Cell array of character vectors **Example:** `{'A' 'B' 'C'}`
	String scalar **Example:** `"A"`	String array **Example:** `["A" "B" "C"]`

Example: `G = rmnode(G,[1 2])` removes node 1 and node 2 from graph `G`.

Output Arguments

`H` — Output graph
`graph` object | `digraph` object

Output graph, returned as a `graph` or `digraph` object.

See Also

`addnode` | `digraph` | `graph` | `numnodes` | `rmedge`

Topics

"Directed and Undirected Graphs"
"Modify Nodes and Edges of Existing Graph"
"Add Graph Node Names, Edge Weights, and Other Attributes"

Introduced in R2015b

reordernodes

Reorder graph nodes

Syntax

```
H = reordernodes(G,order)
[H,idx] = reordernodes(G,order)
```

Description

`H = reordernodes(G,order)` reorders the nodes in graph `G` according to `order`. This reordering corresponds to a symmetric permutation of the adjacency matrix of `G`.

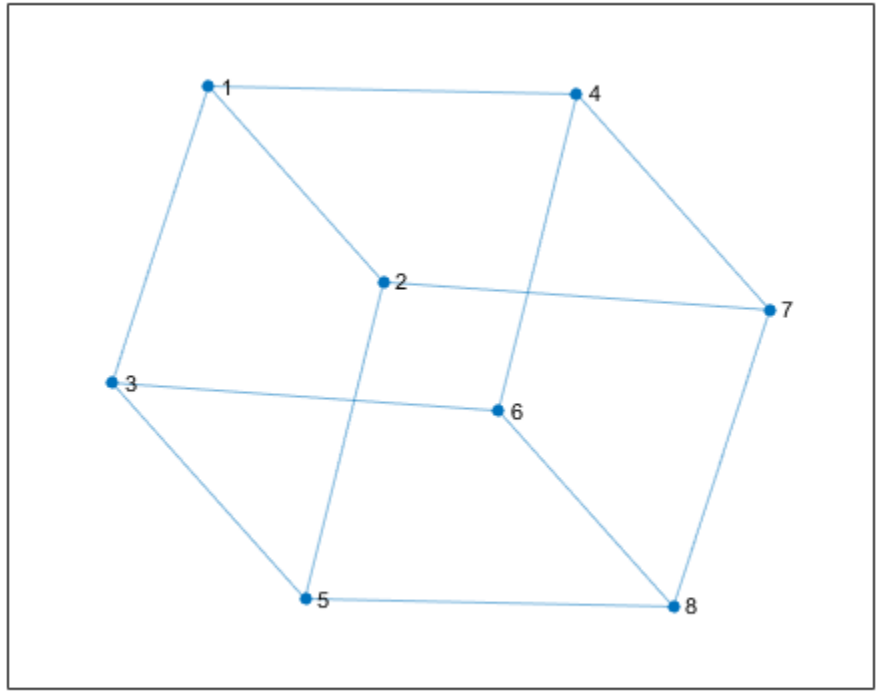
`[H,idx] = reordernodes(G,order)` also returns the permutation vector for the edge indices, `idx`. For example, if `G.Edges` has a variable `Weight`, then `H.Edges.Weight == G.Edges.Weight(idx)`.

Examples

Reorder Graph Nodes

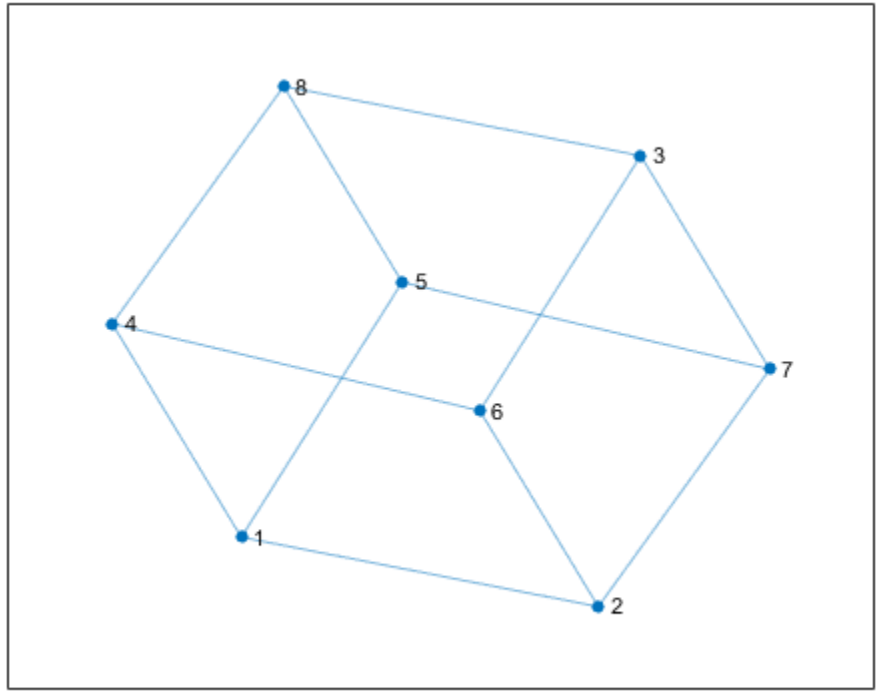
Create and plot a graph.

```
s = [1 1 1 2 5 3 6 4 7 8 8 8];
t = [2 3 4 5 3 6 4 7 2 6 7 5];
G = graph(s,t);
plot(G)
```



Reorder the graph nodes using a permutation vector.

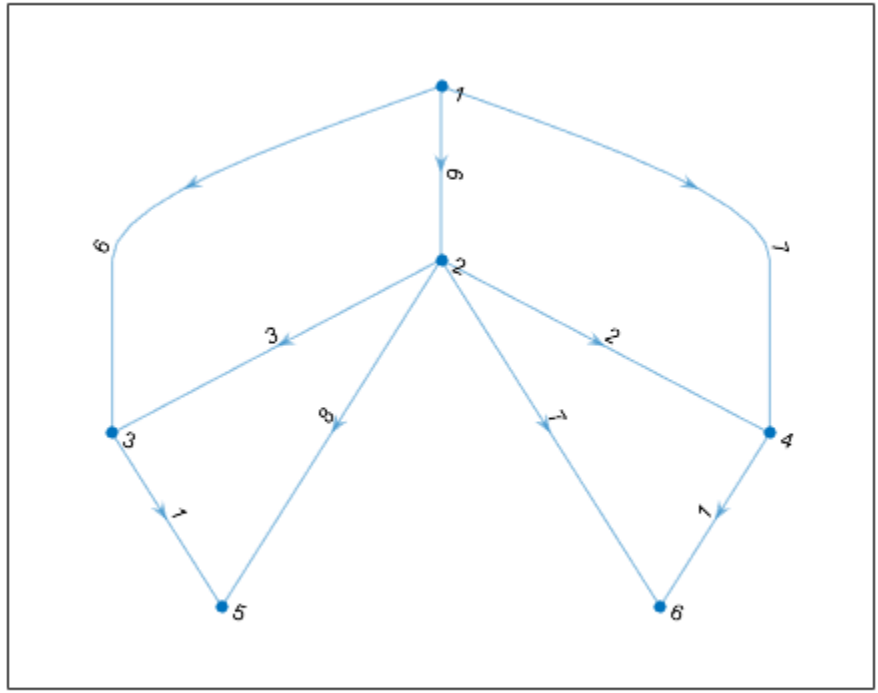
```
order = [7 2 3 4 8 1 5 6];
G = reordernodes(G,order);
plot(G)
```



Reorder Graph Nodes by Degree

Create and plot a weighted graph.

```
s = [1 1 1 2 2 2 2 3 4];
t = [3 4 2 3 4 5 6 5 6];
weights = [6 7 6 3 2 8 7 1 1];
G = digraph(s,t,weights);
plot(G,'EdgeLabel',G.Edges.Weight)
```

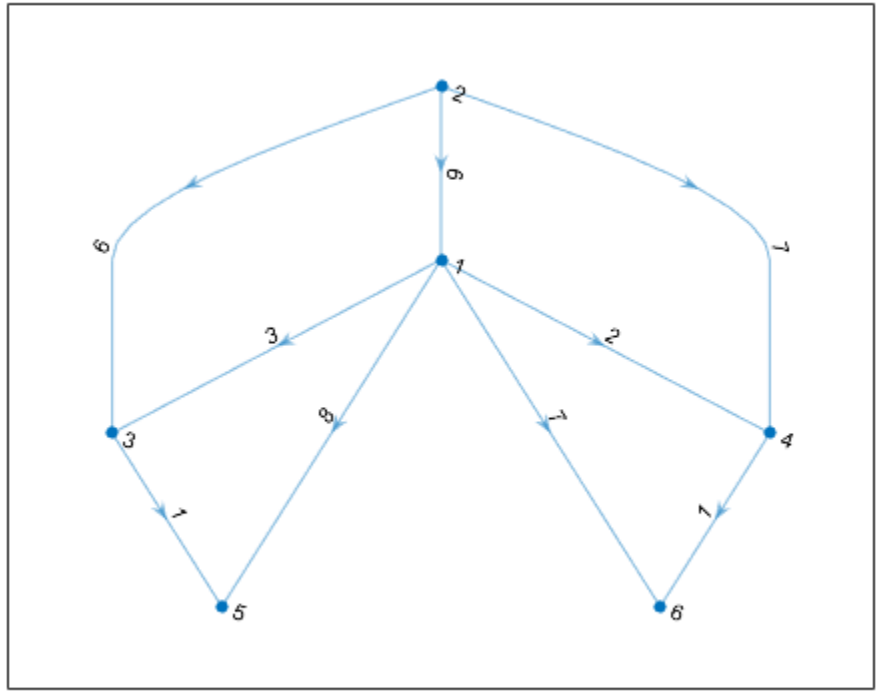


Reorder the graph nodes based on the out-degree, such that node 1 has the largest out-degree.

```
[~,order] = sort(outdegree(G),'descend')
```

```
order = 6×1

     2
     1
     3
     4
     5
     6
```

```
[H,idx] = reordernodes(G,order);
plot(H,'EdgeLabel',H.Edges.Weight)
```



`idx` describes the permutation of the rows in `G.Edges`. Confirm this correspondence using the `Weight` variable.

```
isequal(H.Edges.Weight, G.Edges.Weight(idx))
```

```
ans = logical
   1
```

Input Arguments

G — Input graph
graph object | digraph object

Input graph, specified as either a graph or digraph object. Use graph to create an undirected graph or digraph to create a directed graph.

Example: G = graph(1,2)

Example: G = digraph([1 2],[2 3])

order — Node order
node indices | node names

Node order, specified as node indices or node names. order specifies a symmetric permutation of the adjacency matrix of G. If A = adjacency(G), then A(order,order) produces adjacency(H).

order can be one of:

- Numeric vector of node indices, such as [1 3 2].
- A cell array of character vectors or string array of node names, such as {'A' 'C' 'B'} or ["A" "C" "B"].

Example: H = reordernodes(G,[3 1 2])

Output Arguments

H — Output graph
graph object | digraph object

Output graph, returned as a graph or digraph object. H contains the same Nodes and Edges properties as G, but with the rows of H.Nodes and H.Edges permuted:

- H.Nodes is the same as G.Nodes(order,:).
- H.Edges is similar to G.Edges(idx,:), except that the nodes are renumbered.

idx — Permutation vector of edge indices
vector

Permutation vector of edge indices, returned as a vector. The values in `idx` describe the permutation of the rows in `G.Edges`.

See Also

`addnode` | `digraph` | `findnode` | `graph` | `rmnode` | `subgraph`

Introduced in R2015b

shortestpath

Shortest path between two single nodes

Syntax

```
P = shortestpath(G,s,t)
P = shortestpath(G,s,t,'Method',algorithm)
[P,d] = shortestpath( ___ )
[P,d,edgepath] = shortestpath( ___ )
```

Description

`P = shortestpath(G,s,t)` computes the shortest path starting at source node `s` and ending at target node `t`. If the graph is weighted (that is, `G.Edges` contains a variable `Weight`), then those weights are used as the distances along the edges in the graph. Otherwise, all edge distances are taken to be `1`.

`P = shortestpath(G,s,t,'Method',algorithm)` optionally specifies the algorithm to use in computing the shortest path. For example, if `G` is a weighted graph, then `shortestpath(G,s,t,'Method','unweighted')` ignores the edge weights in `G` and instead treats all edge weights as `1`.

`[P,d] = shortestpath( ___ )` additionally returns the length of the shortest path, `d`, using any of the input arguments in previous syntaxes.

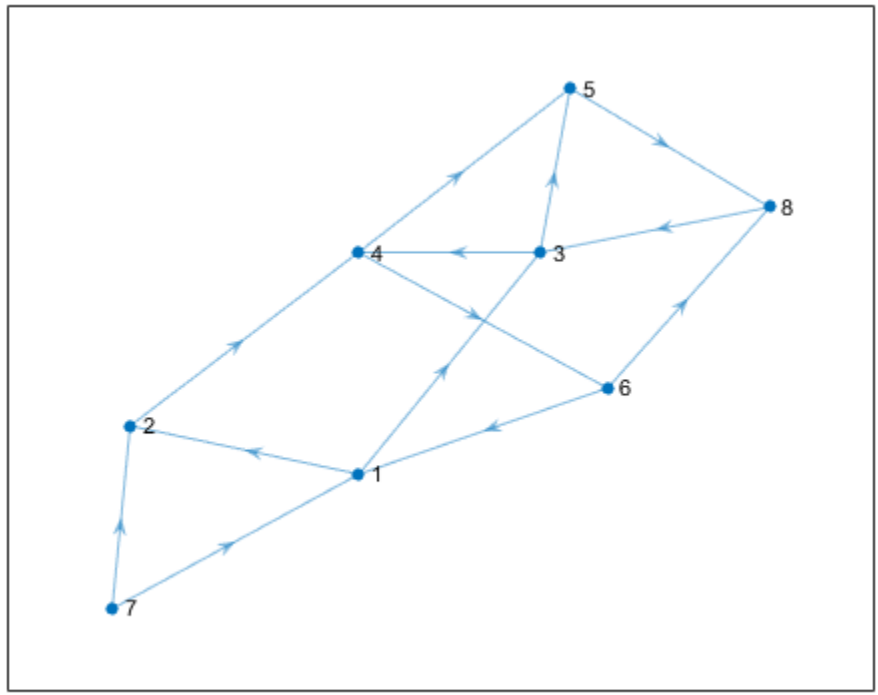
`[P,d,edgepath] = shortestpath( ___ )` additionally returns the edge indices `edgepath` of all edges on the shortest path from `s` to `t`.

Examples

Shortest Path Between Specified Nodes

Create and plot a directed graph.

```
s = [1 1 2 3 3 4 4 6 6 7 8 7 5];
t = [2 3 4 4 5 5 6 1 8 1 3 2 8];
G = digraph(s,t);
plot(G)
```



Calculate the shortest path between nodes 7 and 8.

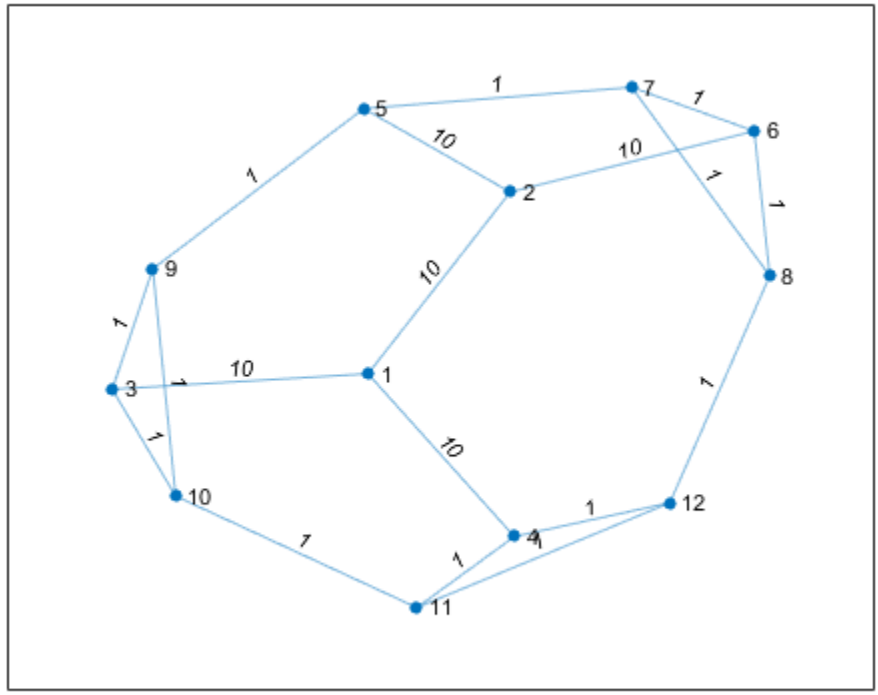
```
P = shortestpath(G,7,8)
```

```
P = 1×5

     7     1     3     5     8
```

Shortest Path in Weighted Graph

Create and plot a graph with weighted edges.

```
s = [1 1 1 2 2 6 6 7 7 3 3 9 9 4 4 11 11 8];
t = [2 3 4 5 6 7 8 5 8 9 10 5 10 11 12 10 12 12];
weights = [10 10 10 10 10 1 1 1 1 1 1 1 1 1 1 1 1 1];
G = graph(s,t,weights);
plot(G,'EdgeLabel',G.Edges.Weight)
```



Find the shortest path between nodes 3 and 8, and specify two outputs to also return the length of the path.

```
[P,d] = shortestpath(G,3,8)
```

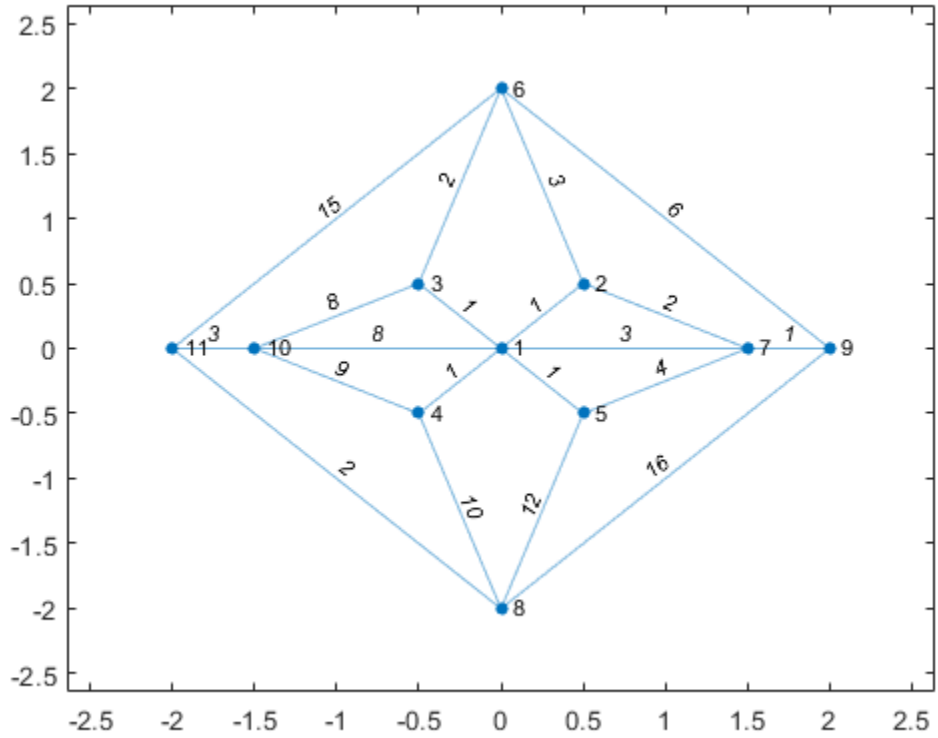
```
P = 1×5

     3     9     5     7     8


d = 4
```

Since the edges in the center of the graph have large weights, the shortest path between nodes 3 and 8 goes around the boundary of the graph where the edge weights are smallest. This path has a total length of 4.

Shortest Path Ignoring Edge Weights

Create and plot a graph with weighted edges, using custom node coordinates.

```
s = [1 1 1 1 1 2 2 7 7 9 3 3 1 4 10 8 4 5 6 8];
t = [2 3 4 5 7 6 7 5 9 6 6 10 10 10 11 11 8 8 11 9];
weights = [1 1 1 1 3 3 2 4 1 6 2 8 8 9 3 2 10 12 15 16];
G = graph(s,t,weights);

x = [0 0.5 -0.5 -0.5 0.5 0 1.5 0 2 -1.5 -2];
y = [0 0.5 0.5 -0.5 -0.5 2 0 -2 0 0 0];
p = plot(G,'XData',x,'YData',y,'EdgeLabel',G.Edges.Weight);
```

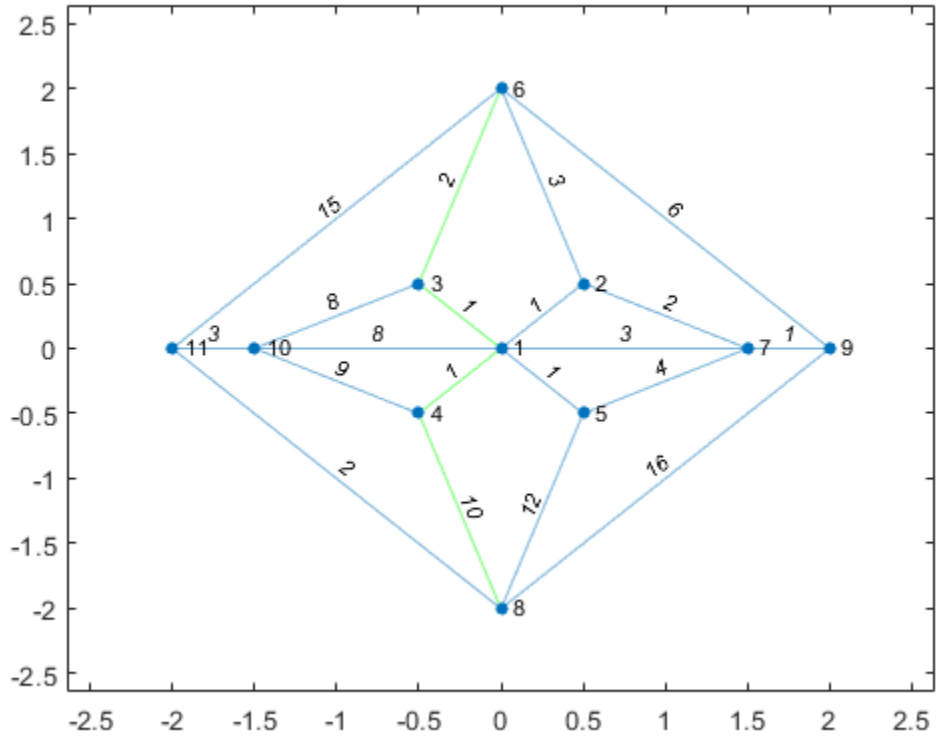


Find the shortest path between nodes 6 and 8 based on the graph edge weights. Highlight this path in green.

```
[path1,d] = shortestpath(G,6,8)
```

```
path1 = 1×5

     6     3     1     4     8


d = 14
```

```
highlight(p,path1,'EdgeColor','g')
```

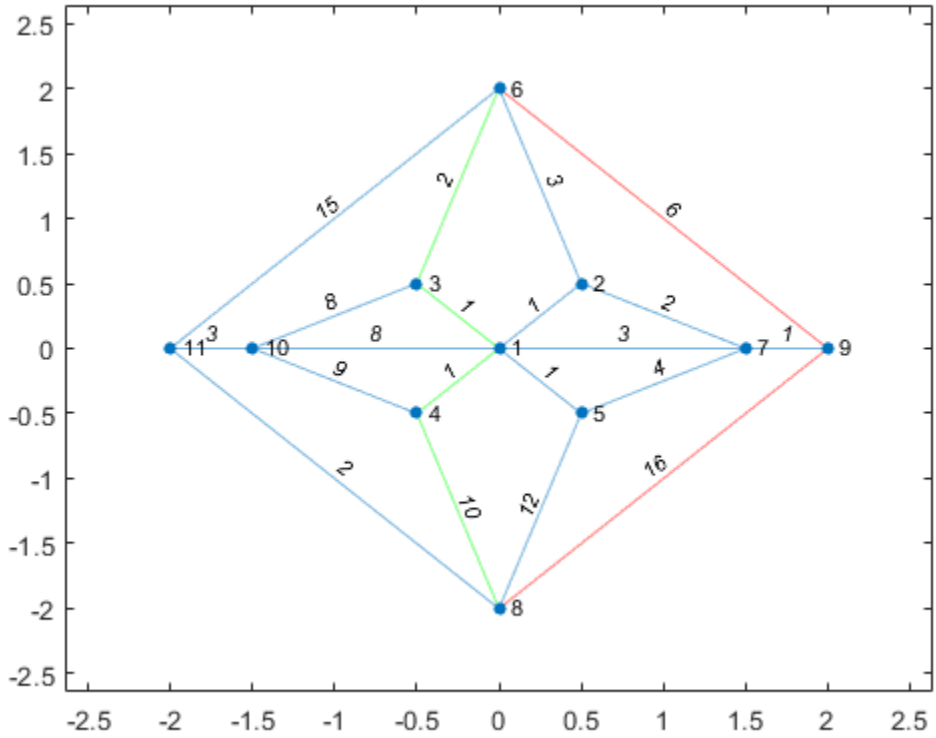


Specify `Method` as `unweighted` to ignore the edge weights, instead treating all edges as if they had a weight of 1. This method produces a different path between the nodes, one that previously had too large of a path length to be the shortest path. Highlight this path in red.

```
[path2,d] = shortestpath(G,6,8,'Method','unweighted')
```

```
path2 = 1×3

     6     9     8


d = 2
```

```
highlight(p,path2,'EdgeColor','r')
```

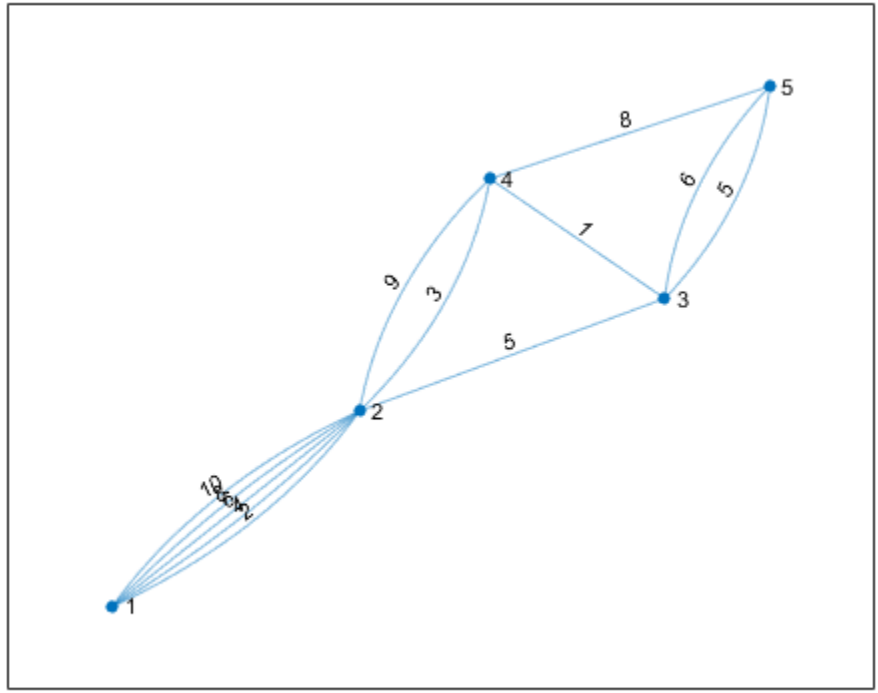


Shortest Path in Multigraph

Plot the shortest path between two nodes in a multigraph and highlight the specific edges that are traversed.

Create a weighted multigraph with five nodes. Several pairs of nodes have more than one edge between them. Plot the graph for reference.

```
G = graph([1 1 1 1 1 2 2 3 3 3 4 4],[2 2 2 2 2 3 4 4 5 5 5 2],[2 4 6 8 10 5 3 1 5 6 8 9
p = plot(G,'EdgeLabel',G.Edges.Weight);
```



Find the shortest path between node 1 and node 5. Since several of the node pairs have more than one edge between them, specify three outputs to `shortestpath` to return the specific edges that the shortest path traverses.

```
[P,d,edgepath] = shortestpath(G,1,5)
```

```
P = 1×5

     1     2     4     3     5


d = 11

edgepath = 1×4
```

```
     1     7     9    10
```

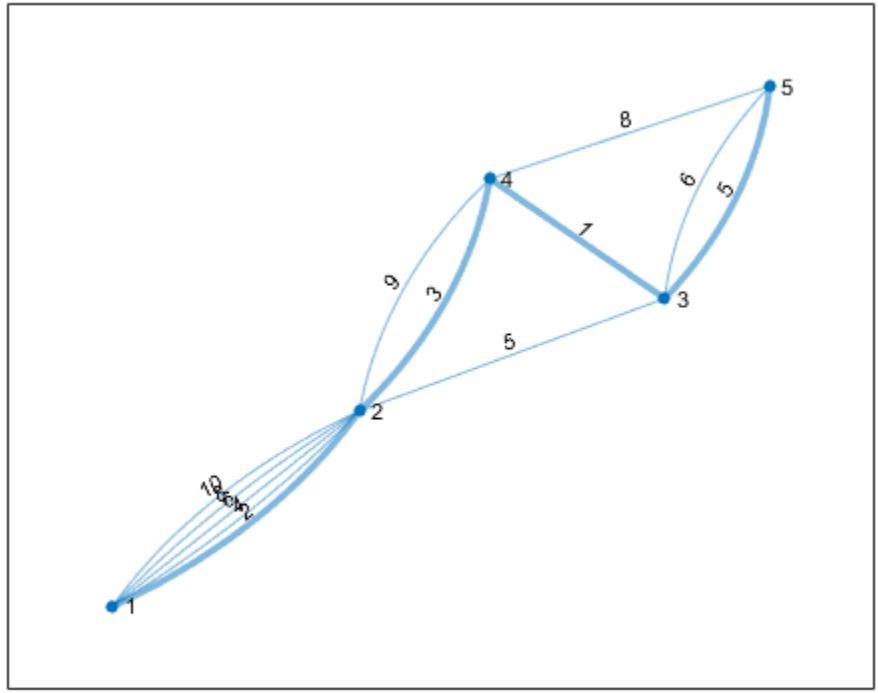
The results indicate that the shortest path has a total length of 11 and follows the edges given by `G.Edges(edgepath,:)`.

```
G.Edges(edgepath,:)
```

```
ans=4×2 table
    EndNodes    Weight
    ________    ______

     1    2       2
     2    4       3
     3    4       1
     3    5       5
```

Highlight this edge path by using the `highlight` function with the `'Edges'` name-value pair to specify the indices of the edges traversed.

```
highlight(p,'Edges',edgepath)
```



Input Arguments

G — Input graph
graph object | digraph object

Input graph, specified as either a graph or digraph object. Use graph to create an undirected graph or digraph to create a directed graph.

Example: G = graph(1,2)

Example: G = digraph([1 2],[2 3])

`s,t` — Source and target node IDs (as separate arguments)
node indices | node names

Source and target node IDs, specified as separate arguments of node indices or node names.

Value	Example
Scalar node index	`1`
Character vector node name	`'A'`
String scalar node name	`"A"`

Example: `shortestpath(G,2,5)` computes the shortest path between node 2 and node 5.

Example: `shortestpath(G,'node1','node2')` computes the shortest path between the named nodes `node1` and `node2`.

`algorithm` — Shortest path algorithm
`'auto'` (default) | `'unweighted'` | `'positive'` | `'mixed'` | `'acyclic'`

Shortest path algorithm, specified as one of the options in the table.

Option	Description
`'auto'` (default)	The `'auto'` option automatically selects the algorithm: • `'unweighted'` is used for `graph` and `digraph` inputs with no edge weights. • `'positive'` is used for all `graph` inputs that have edge weights, and requires the weights to be nonnegative. This option is also used for `digraph` inputs with nonnegative edge weights. • `'mixed'` is used for `digraph` inputs whose edge weights contain some negative values. The graph cannot have negative cycles.

Option	Description
`'unweighted'`	Breadth-First computation that treats all edge weights as 1.
`'positive'`	Dijkstra algorithm that requires all edge weights to be nonnegative.
`'mixed'` (only for `digraph`)	Bellman-Ford algorithm for directed graphs that requires the graph to have no negative cycles. While `'mixed'` is slower than `'positive'` for the same problem, `'mixed'` is more versatile as it allows some edge weights to be negative.
`'acyclic'` (only for `digraph`)	Algorithm designed to improve performance for directed, acyclic graphs (DAGs) with weighted edges. Use `isdag` to confirm if a directed graph is acyclic.

Note For most graphs, `'unweighted'` is the fastest algorithm, followed by `'acyclic'`, `'positive'`, and `'mixed'`.

Example: `shortestpath(G,s,t,'Method','acyclic')`

Output Arguments

P — Shortest path between nodes

node indices | node names

Shortest path between nodes, returned as a vector of node indices or an array of node names. `P` is empty, `{}`, if there is no path between the nodes.

- If `s` and `t` contain numeric node indices, then `P` is a numeric vector of node indices.
- If `s` and `t` contain node names, then `P` is a cell array or string array containing node names.

If there are multiple shortest paths between `s` and `t`, then `P` contains only one of the paths. The path that is returned can change depending on which algorithm `Method` specifies.

`d` — Shortest path distance
scalar

Shortest path distance, returned as a numeric scalar. `d` is the summation of the edge weights between consecutive nodes in `P`. If there is no path between the nodes, then `d` is `Inf`.

`edgepath` — Edges on shortest path
vector of edge indices

Edges on shortest path, returned as a vector of edge indices. For multigraphs, this output indicates which edge between two nodes is on the path. This output is compatible with the `'Edges'` name-value pair of `highlight`, for example: `highlight(p,'Edges',edgepath)`.

Tips

- The `shortestpath`, `shortestpathtree`, and `distances` functions do not support undirected graphs with negative edge weights, or more generally any graph containing a negative cycle, for these reasons:
  - A *negative cycle* is a path that leads from a node back to itself, with the sum of the edge weights on the path being negative. If a negative cycle is on a path between two nodes, then no shortest path exists between the nodes, since a shorter path can always be found by traversing the negative cycle.
  - A single negative edge weight in an undirected graph creates a negative cycle.

See Also

`digraph` | `distances` | `graph` | `nearest` | `shortestpathtree`

Introduced in R2015b

shortestpathtree

Shortest path tree from node

Syntax

```
TR = shortestpathtree(G,s)
TR = shortestpathtree(G,s,t)
TR = shortestpathtree( ___ ,Name,Value)
[TR,D] = shortestpathtree( ___ )
[TR,D,E] = shortestpathtree( ___ )
```

Description

`TR = shortestpathtree(G,s)` returns a directed graph, `TR`, that contains the tree of shortest paths from source node `s` to all other nodes in the graph. If the graph is weighted (that is, `G.Edges` contains a variable `Weight`), then those weights are used as the distances along the edges in the graph. Otherwise, all edge distances are taken to be 1.

`TR = shortestpathtree(G,s,t)` computes the tree of shortest paths between multiple source or target nodes:

- `s` can be a single source node, and `t` can specify multiple target nodes.
- `s` can specify several source nodes, and `t` can specify a single target node.

`TR = shortestpathtree( ___ ,Name,Value)` uses additional options specified by one or more Name-Value pair arguments, using any of the input argument combinations in previous syntaxes. For example, `shortestpathtree(G,s,'OutputForm','vector')` returns a numeric vector that describes the shortest path tree.

`[TR,D] = shortestpathtree( ___ )` additionally returns the shortest path distance between nodes in the tree.

`[TR,D,E] = shortestpathtree( ___ )` additionally returns a logical vector `E` that indicates whether each graph edge is in `TR`.

Examples

Shortest Paths from Specified Source Node

Find the shortest paths from a source node to each of the other reachable nodes in a graph, and plot the results.

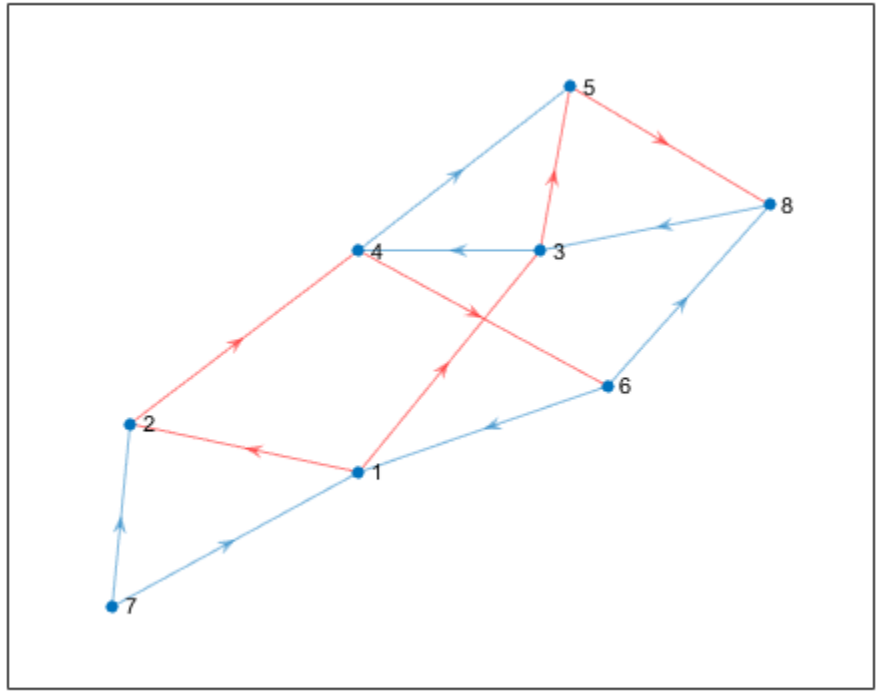
Create a directed graph.

```
s = [1 1 2 3 3 4 4 6 6 7 8 7 5];
t = [2 3 4 4 5 5 6 1 8 1 3 2 8];
G = digraph(s,t)
```

```
G = 
  digraph with properties:

    Edges: [13x1 table]
    Nodes: [8x0 table]
```

Calculate the shortest paths from node 1 to each of the other reachable nodes in the graph. Then, plot the resulting tree on top of the graph.

```
TR = shortestpathtree(G,1);
p = plot(G);
highlight(p,TR,'EdgeColor','r')
```



Since there is no path from node 1 to node 7, node 7 is disconnected from the tree.

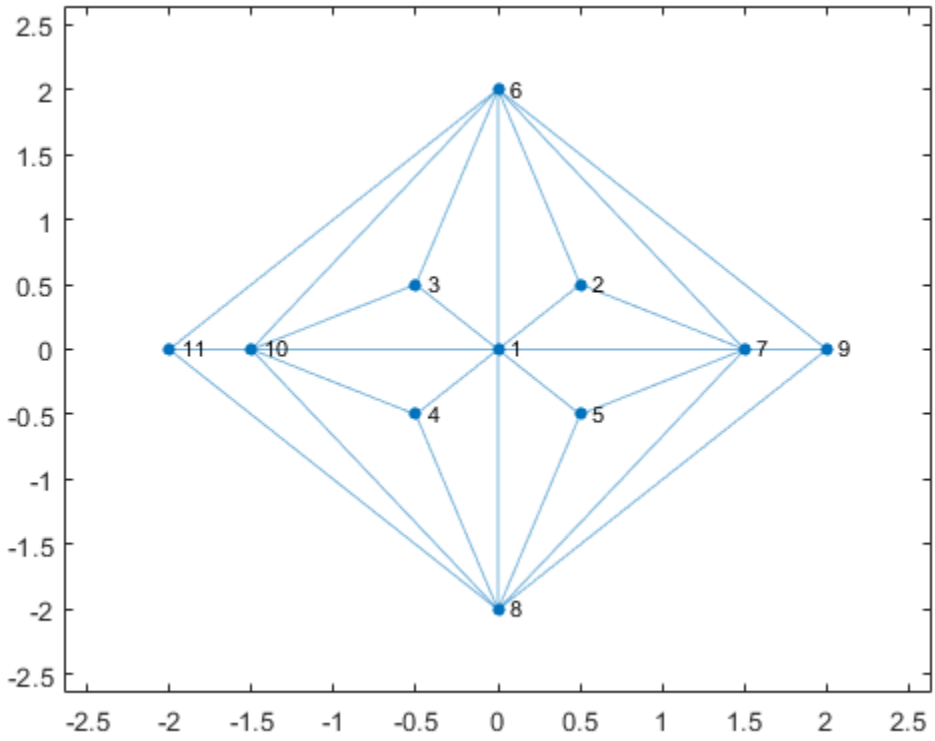
Shortest Paths to Specified Target Node

Find the shortest paths from each node in a graph to a target node, and plot the results.

Create and plot a graph.

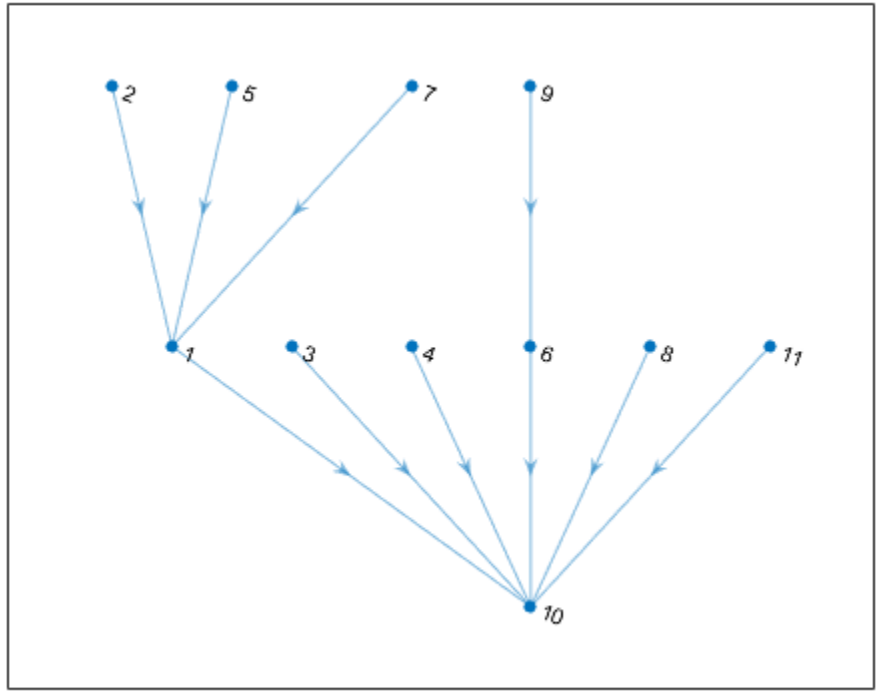
```
s = [1 1 1 1 1 1 1 2 2 7 7 7 7 9 9 3 3 1 6 4 8 10 6 8 4 5];
t = [2 3 4 5 6 8 7 6 7 5 6 8 9 6 8 6 10 10 10 10 10 11 11 11 8 8];
G = graph(s,t);
```

```
x = [0 0.5 -0.5 -0.5 0.5 0 1.5 0 2 -1.5 -2];
y = [0 0.5 0.5 -0.5 -0.5 2 0 -2 0 0 0];
plot(G,'XData',x,'YData',y)
```



Find the shortest paths from each node in the graph to node 10. Plot the resulting tree.

```
TR = shortestpathtree(G,'all',10);
plot(TR)
```

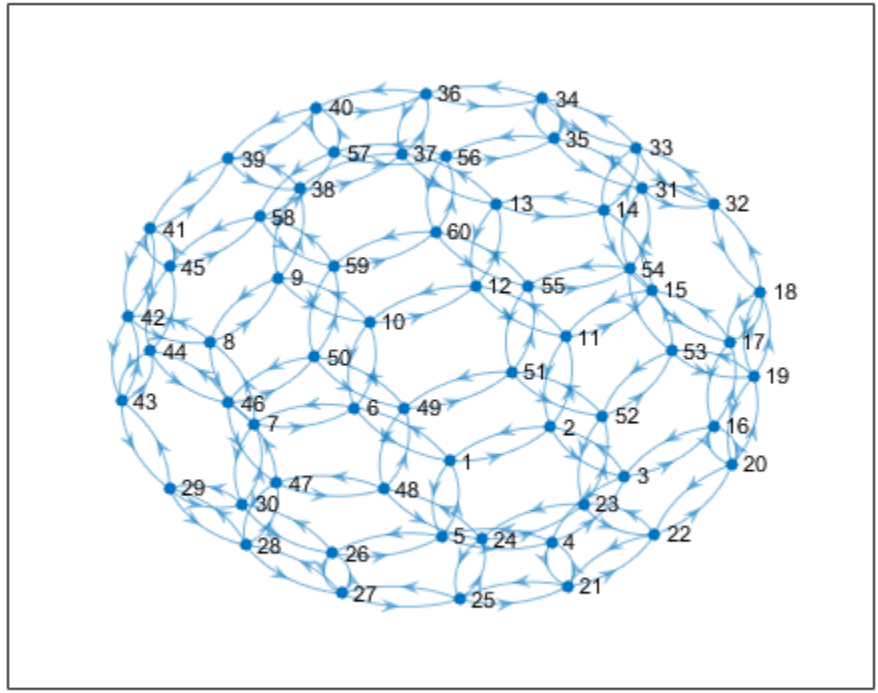


Subset of Shortest Paths with Specified Source Node

Find the shortest paths and path lengths from a single source node to several target nodes.

Create and plot a graph.

```
G = digraph(bucky);
plot(G)
```



Find the shortest paths from node 23 to several other nodes. Specify `OutputForm` as `cell` to return the shortest paths in a cell array. Specify two outputs to also return the shortest path distances.

```
target = [1 5 13 32 44];
[TR,D] = shortestpathtree(G,23,target,'OutputForm','cell')
```

```
TR=5×1 cell
    {1x6 double}
    {1x5 double}
    {1x8 double}
    {1x6 double}
    {1x6 double}
```

```
D = 1×5

     5     4     7     5     5
```

`tree{j}` is the shortest path from node 23 to node `target(j)` with length `D(j)`.

Find the path and path length from node 21 to node 5.

```
path = TR{2}
```

```
path = 1×5

    23    22    21     4     5
```

```
path_length = D(2)
```

```
path_length = 4
```

Input Arguments

G — Input graph
`graph` object | `digraph` object

Input graph, specified as either a `graph` or `digraph` object. Use `graph` to create an undirected graph or `digraph` to create a directed graph.

Example: `G = graph(1,2)`

Example: `G = digraph([1 2],[2 3])`

s — Source node(s)
node indices | node names | `'all'`

Source node(s), specified as one or more node indices or node names, or as all nodes in the graph with `'all'`.

- When used alone, `s` must specify a single source node.
- When used together with `t`, the `s` and `t` inputs must satisfy:
  - `s` can be a single source node, and `t` can specify multiple target nodes.

- `s` can specify several source nodes, and `t` can specify a single target node.

This table shows the different ways to refer to one or more nodes either by their numeric node indices or by their node names.

Form	Single Node	Multiple Nodes
Node index	Scalar **Example:** `1`	Vector **Example:** `[1 2 3]`
Node name	Character vector **Example:** `'A'`	Cell array of character vectors **Example:** `{'A' 'B' 'C'}`
	String scalar **Example:** `"A"`	String array **Example:** `["A" "B" "C"]`

`s` must not specify the node name `'all'`, since this node name conflicts with an option name. Use `findnode` to instead pass in the node index for this case.

Example: `shortestpathtree(G,'a')`

Example: `shortestpathtree(G,[1 2 3],8)`

`t` — Target node(s)
`'all'` (default) | node indices | node names

Target node(s), specified as one or more node indices or node names, or as all nodes in the graph with `'all'`.

The `s` and `t` inputs must satisfy:

- `s` can be a single source node, and `t` can specify multiple target nodes.
- `s` can specify several source nodes, and `t` can specify a single target node.

`t` must not specify nodes named `'all'`, `'Method'`, or `'OutputForm'`, since these node names conflict with option names. Use `findnode` to instead pass in the node index for these cases.

Example: `shortestpathtree(G,[1 2 3],8)`

Example: `shortestpathtree(G,{'a','b','c'},{'f'})`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[TR,D] = shortestpathtree(G,s,t,'Method','unweighted','OutputForm','vector')`

`OutputForm` — Format of output
`'tree'` (default) | `'cell'` | `'vector'`

Format of output, specified as the comma-separated pair consisting of `'OutputForm'` and one of the options in the table.

Option	Description
`'tree'` (default)	`TR` is a directed graph representing the shortest path tree. If specified, the third output `E` is a logical vector indicating whether each edge is in `TR`.
`'cell'`	`TR` is a cell array, and `TR{k}` contains the path from `s` to `t(k)` or from `s(k)` to `t`. If there is no path between the nodes, then `TR{k}` is empty. If `s` and `t` are node names, then `TR{k}` is a cell array of character vectors. Otherwise, `TR{k}` is a numeric vector. If specified, the third output `E` is a cell array indicating the edges on each corresponding path in `TR`.

Option	Description
`'vector'`	`TR` is a vector that describes the tree: • If `s` contains a single source node, then `TR(k)` is the ID of the node that precedes node `k` on the path from `s` to `k`. By convention, `TR(s) = 0`. • If `s` contains multiple source nodes, then `TR(k)` is the ID of the node that succeeds node `k` on the path from `k` to `t`. By convention, `TR(t) = 0`. In each case `TR(k)` is `NaN` if node `k` is not part of the tree. If specified, the third output `E` is a vector where `E(k)` gives the index of the edge of the shortest path tree connecting node `k` and node `TR(k)`.

Example: `shortestpathtree(G,s,'OutputForm','vector')`

`Method` — Shortest path algorithm
`'auto'` (default) | `'unweighted'` | `'positive'` | `'mixed'` | `'acyclic'`

Shortest path algorithm, specified as the comma-separated pair consisting of `'Method'` and one of the options in the table.

Option	Description
`'auto'` (default)	The `'auto'` option automatically selects the algorithm: • `'unweighted'` is used for `graph` and `digraph` inputs with no edge weights. • `'positive'` is used for all `graph` inputs that have edge weights, and requires the weights to be nonnegative. This option is also used for `digraph` inputs with nonnegative edge weights. • `'mixed'` is used for `digraph` inputs whose edge weights contain some negative values. The graph cannot have negative cycles.
`'unweighted'`	Breadth-First computation that treats all edge weights as `1`.
`'positive'`	Dijkstra algorithm that requires all edge weights to be nonnegative.
`'mixed'` (only for `digraph`)	Bellman-Ford algorithm for directed graphs that requires the graph to have no negative cycles. While `'mixed'` is slower than `'positive'` for the same problem, `'mixed'` is more versatile as it allows some edge weights to be negative.
`'acyclic'` (only for `digraph`)	Algorithm designed to improve performance for directed, acyclic graphs (DAGs) with weighted edges. Use `isdag` to confirm if a directed graph is acyclic.

Note For most graphs, `'unweighted'` is the fastest algorithm, followed by `'acyclic'`, `'positive'`, and `'mixed'`.

Example: `shortestpath(G,s,t,'Method','acyclic')`

Output Arguments

TR — Shortest path tree
`digraph` object (default) | cell array | vector

Shortest path tree, returned as a `digraph` object, cell array, or vector, depending on the value of `'OutputForm'`. Use the `highlight` function to visualize the shortest path tree on top of a plot of the graph, or use `plot(TR)` to visualize the shortest path tree on its own.

If there are multiple shortest paths between two nodes, then `TR` contains only one of the paths. The path that is returned can change depending on which algorithm is specified by `Method`. The `TR` output is a graph with zero edges if there are no paths connecting any of the specified nodes.

D — Distance between source and target nodes
vector

Distance between source and target nodes, returned as a vector. A value of `Inf` indicates there is no path between two nodes.

E — Edges in tree or on path
logical vector (default) | cell array | vector

Edges in tree or on path, returned as a logical vector, cell array, or vector, depending on the value of `'OutputForm'`:

- If you don't specify `'OutputForm'` or specify a value of `'tree'`, then `E` is a logical vector indicating whether each graph edge is in directed graph `TR`. This output is compatible with the `'Edges'` name-value pair of `highlight`, for example: `highlight(p,'Edges',E)`.
- If `'OutputForm'` is `'cell'`, then `E` is a cell array containing the edges on the corresponding paths in `TR`.
- If `'OutputForm'` is `'vector'`, then `E` is a vector which, for each node, gives the index of the edge connecting it to its parent node in the shortest path tree.

Tips

- The `shortestpath`, `shortestpathtree`, and `distances` functions do not support undirected graphs with negative edge weights, or more generally any graph containing a negative cycle, for these reasons:
  - A *negative cycle* is a path that leads from a node back to itself, with the sum of the edge weights on the path being negative. If a negative cycle is on a path between two nodes, then no shortest path exists between the nodes, since a shorter path can always be found by traversing the negative cycle.
  - A single negative edge weight in an undirected graph creates a negative cycle.

See Also

`digraph` | `distances` | `graph` | `nearest` | `shortestpath`

Introduced in R2015b

simplify

Reduce multigraph to simple graph

Syntax

```
H = simplify(G)
H = simplify(G,pickmethod)
H = simplify(G,aggregatemethod)
H = simplify( ___ ,selfloopflag)
H = simplify( ___ ,Name,Value)
[H,eind,ecount] = simplify( ___ )
```

Description

`H = simplify(G)` returns a graph without multiple edges or self-loops. When several edges occur between the same two nodes, only the first edge (as defined in `G.Edges`) is kept. Edge properties are preserved.

`H = simplify(G,pickmethod)` specifies a method to choose between multiple edges. Edge properties are preserved. `pickmethod` can be `'first'` (default), `'last'`, `'min'`, or `'max'`.

`H = simplify(G,aggregatemethod)` specifies a method to combine the edge weights of multiple edges into the weight of a single new edge. All other edge properties in `G` are dropped. `aggregatemethod` can be `'sum'` or `'mean'`.

`H = simplify( ___ ,selfloopflag)` specifies whether to preserve or remove self-loops from the graph using any of the input argument combinations in previous syntaxes. For example, `'keepselfloops'` specifies that nodes with one or more self-loops will have one self-loop in the simplified graph.

`H = simplify( ___ ,Name,Value)` specifies additional options with one or more name-value pair arguments. For example, you can specify `'PickVariable'` and a variable in `G.Edges` to use that variable with the `'min'` or `'max'` selection methods.

`[H,eind,ecount] = simplify( ___ )` additionally returns edge indices `eind` and edge counts `ecount`:

- `H.Edges(eind(i),:)` is the edge in `H` that represents edge `i` in `G`.
- `ecount(j)` is the number of edges in `G` that correspond to edge `j` in `H`.

Examples

Simplify Multigraph to Simple Graph

Create a weighted, undirected multigraph with several edges between node 1 and node 2.

```
G = graph([1 1 1 1 2 3],[2 2 2 3 3 4], 1:6);
G.Edges
```

```
ans=6×2 table
    EndNodes    Weight
    ________    ______

     1    2       1
     1    2       2
     1    2       3
     1    3       4
     2    3       5
     3    4       6
```

Simplify the multigraph into a simple graph, such that there is only one edge between node 1 and node 2. `simplify` keeps the first edge between those two nodes, `G.Edges(1,:)`, and drops the others.

```
G = simplify(G);
G.Edges
```

```
ans=4×2 table
    EndNodes    Weight
    ________    ______

     1    2       1
     1    3       4
     2    3       5
```

```
     3    4        6
```

Pick or Combine Multiple Graph Edges

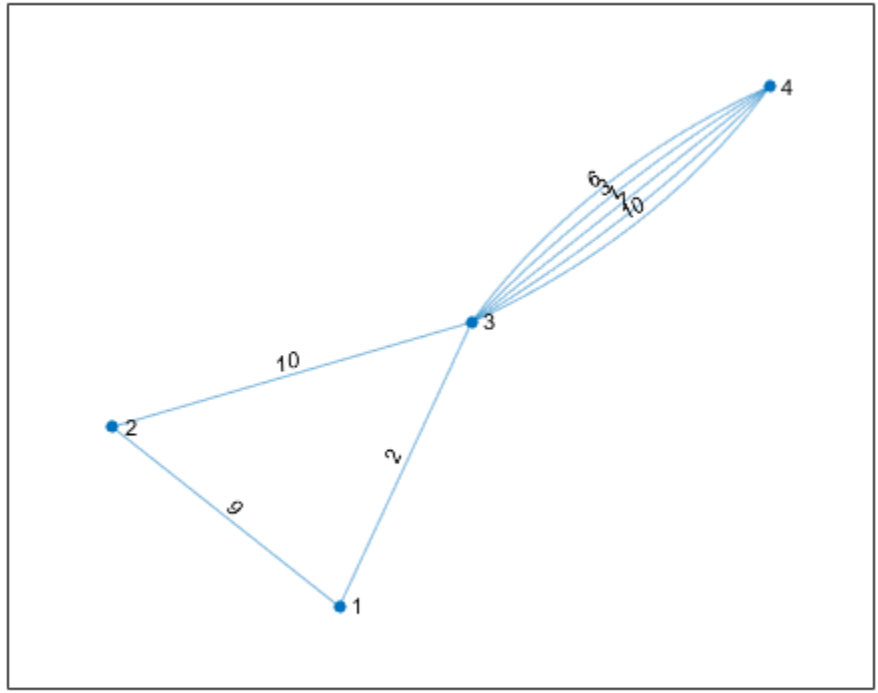
Use the second input of `simplify` to select a method that picks between multiple edges or combines multiple edges into one.

Create a weighted multigraph. In this graph, five edges occur between node 3 and node 4, but the edges have random weights. View the edges table and plot the graph for reference.

```
G = graph([1 2 3 3 3 3 3 3 ],[2 3 1 4 4 4 4 4],randi(10,1,8));
G.Edges
```

```
ans=8×2 table
    EndNodes    Weight
    ________    ______

     1    2        9
     1    3        2
     2    3       10
     3    4       10
     3    4        7
     3    4        1
     3    4        3
     3    4        6
```

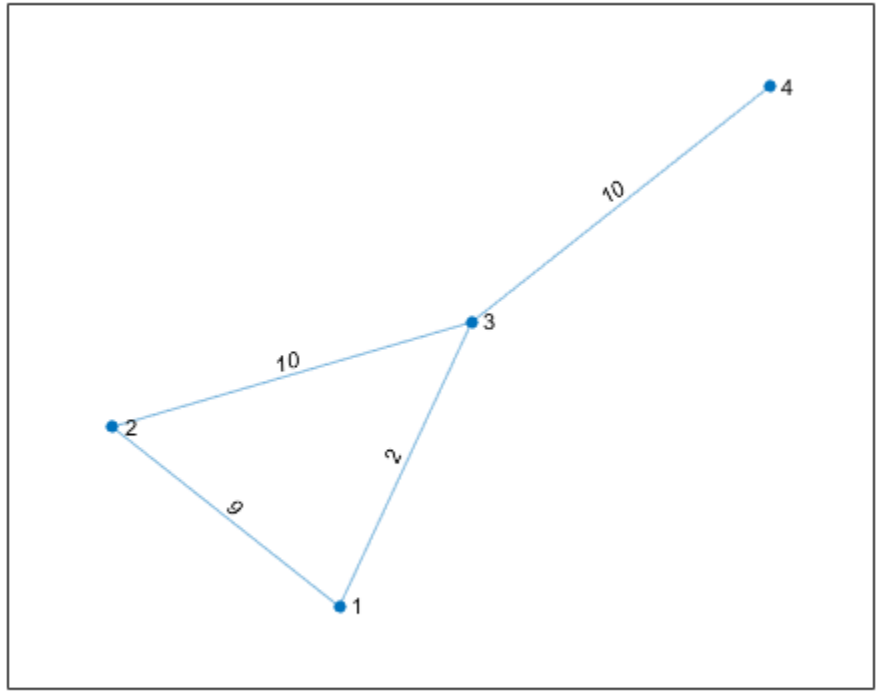
```
plot(G,'EdgeLabel',G.Edges.Weight)
```



The command `simplify(G)` keeps the first of the repeated edges. However, you can specify a different pick/combine method with the second input.

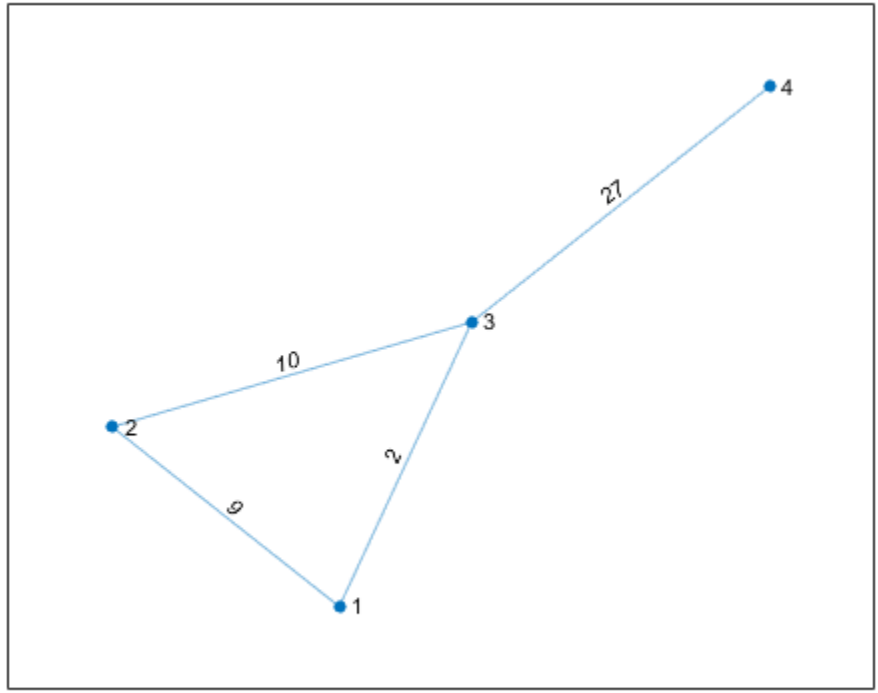
The options for picking between multiple edges are: `'first'` (default), `'last'`, `'min'`, and `'max'`. Keep the repeated edge with maximum weight.

```
H_pick = simplify(G,'max');
plot(H_pick,'EdgeLabel',H_pick.Edges.Weight)
```



The options for combining multiple edges into one are: `'sum'` and `'mean'`. Sum repeated edges together to produce a single edge with a larger weight.

```
H_comb = simplify(G,'sum');
plot(H_comb,'EdgeLabel',H_comb.Edges.Weight)
```

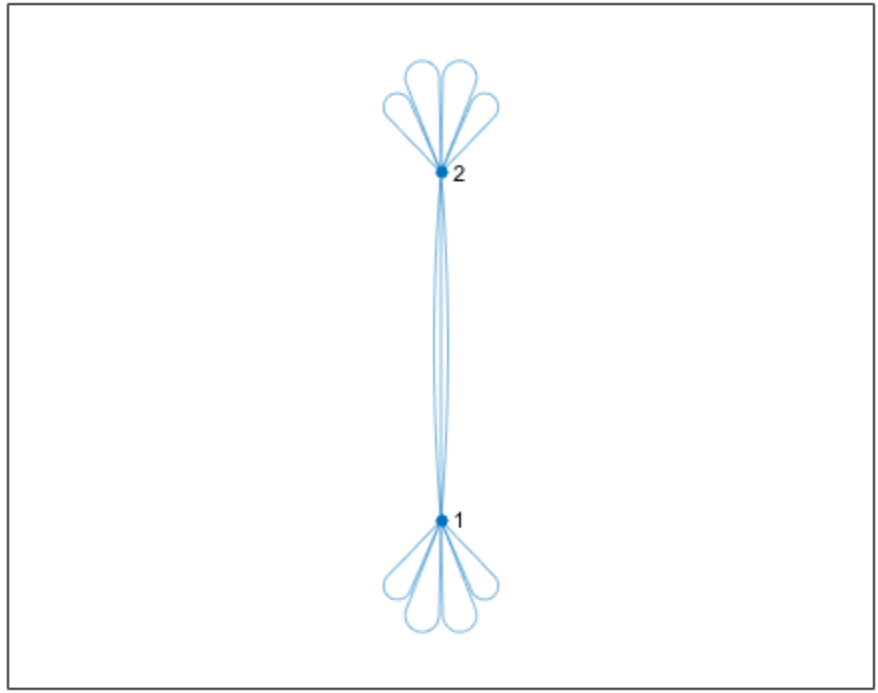


Preserve Self-Loops in Graph

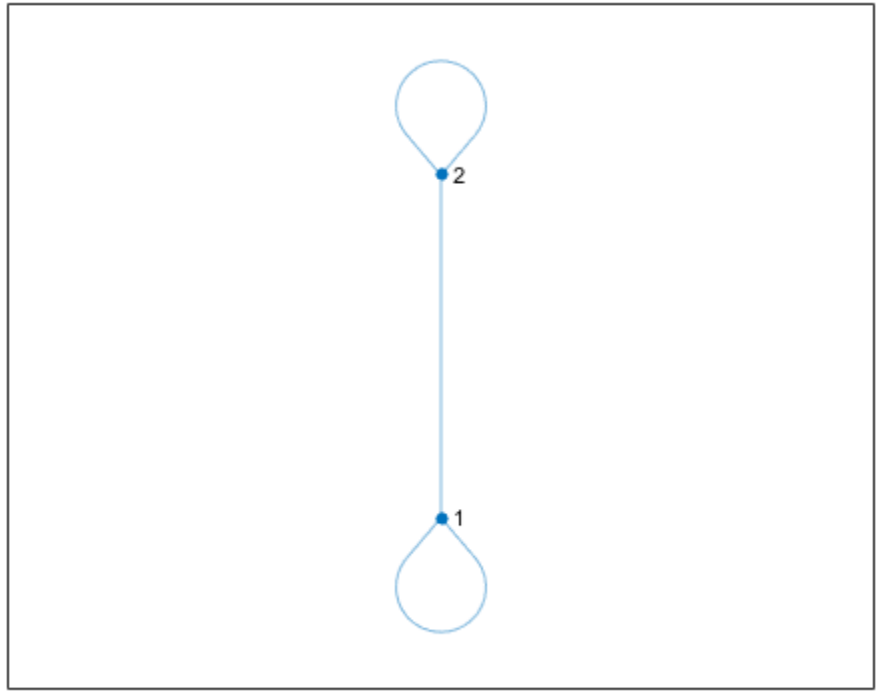
Simplify a graph while preserving self-loops using the `'keepselfloops'` option.

Create a multigraph with two nodes and several self-loops. Simplify the graph and preserve self-loops.

```
G = graph([1 1 1 1 1 1 1 2 2 2 2],[1 1 1 1 2 2 2 2 2 2 2 ]);
plot(G)
axis equal
```




```
G = simplify(G,'keepselfloops');
plot(G)
axis equal
```



Edge Indices and Counts of Repeated Edges

Use the second and third outputs of `simplify` to get information about how many (and which) edges are combined.

Create an undirected multigraph with three nodes and four edges.

```
G = graph([1 1 1 2],[2 2 3 3]);
G.Edges
```

```
ans=4×1 table
    EndNodes
```

```
    ________

     1    2
     1    2
     1    3
     2    3
```

Simplify the graph and specify three outputs to get additional information about the combined edges.

```
[G,ei,ec] = simplify(G)
```

```
G = 
  graph with properties:

    Edges: [3x1 table]
    Nodes: [3x0 table]


ei = 4×1

     1
     1
     2
     3


ec = 3×1

     2
     1
     1
```

`ei(i)` is the edge in the simplified graph that represents edge `i` in the old graph. Since the first two edges are repeated, `ei(1) = ei(2) = 1`. Also, `ec(1) = 2`, since there are two edges in the new graph corresponding to edge 1 in the old graph.

Simplify Graph Using Specific Edge Variables

Show how to simplify a multigraph using the `'PickVariable'` and `'AggregationVariables'` name-value pairs.

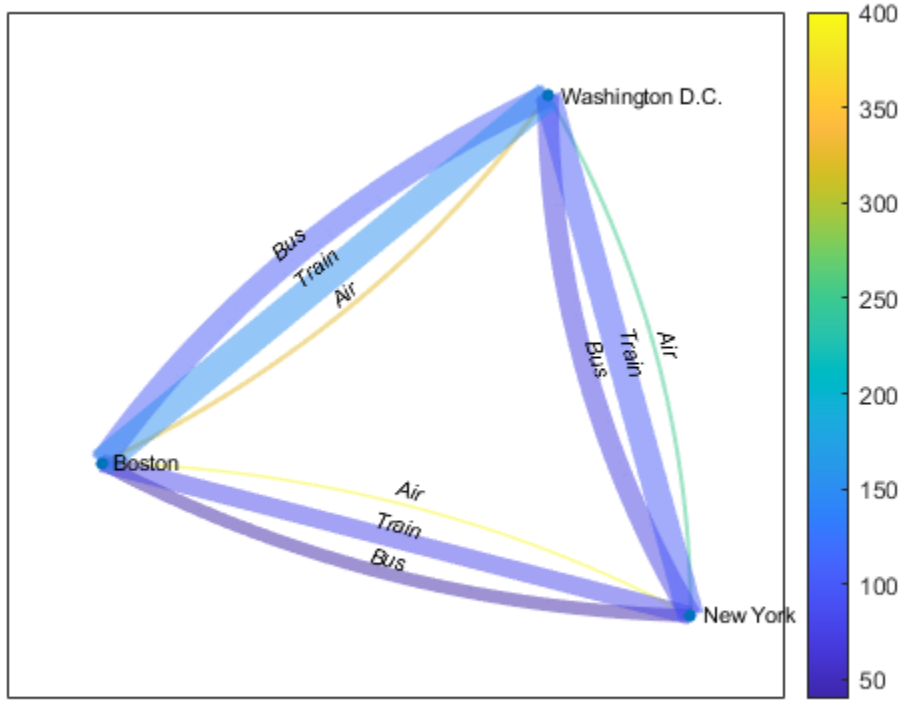
Create a multigraph where the nodes represent locations and the edges represent modes of transport. The edges have properties that reflect the cost and time of each mode of transportation. Preview the edges table.

```
G = graph([1 1 1 1 1 1 2 2 2],[2 2 2 3 3 3 3 3 3],[],{'New York', 'Boston', 'Washingto
G.Edges.Mode = categorical([1 2 3 1 2 3 1 2 3],[1 2 3],{'Air' 'Train' 'Bus'})';
G.Edges.Cost = [400 80 40 250 100 75 325 150 100]';
G.Edges.Time = [1 7 5 1.5 10 8 1.75 11 9]';
G.Edges
```

```
ans=9×4 table
                 EndNodes                    Mode     Cost    Time
    ____________________________________    _____    ____    ____

    {'New York'}    {'Boston'         }    Air      400        1
    {'New York'}    {'Boston'         }    Train     80        7
    {'New York'}    {'Boston'         }    Bus       40        5
    {'New York'}    {'Washington D.C.'}    Air      250      1.5
    {'New York'}    {'Washington D.C.'}    Train    100       10
    {'New York'}    {'Washington D.C.'}    Bus       75        8
    {'Boston'  }    {'Washington D.C.'}    Air      325     1.75
    {'Boston'  }    {'Washington D.C.'}    Train    150       11
    {'Boston'  }    {'Washington D.C.'}    Bus      100        9
```

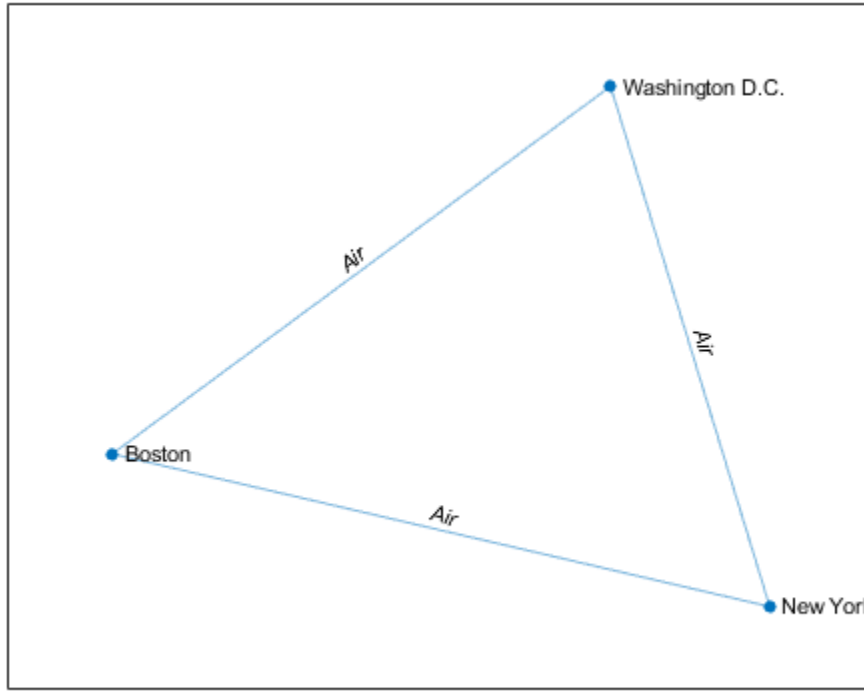
Plot the graph for reference. Label the transportation mode on each edge, make the edge line widths proportional to the time, and the color of each edge proportional to the cost.

```
plot(G,'EdgeLabel',cellstr(G.Edges.Mode),'LineWidth',G.Edges.Time./min(G.Edges.Time),'E
colorbar
```



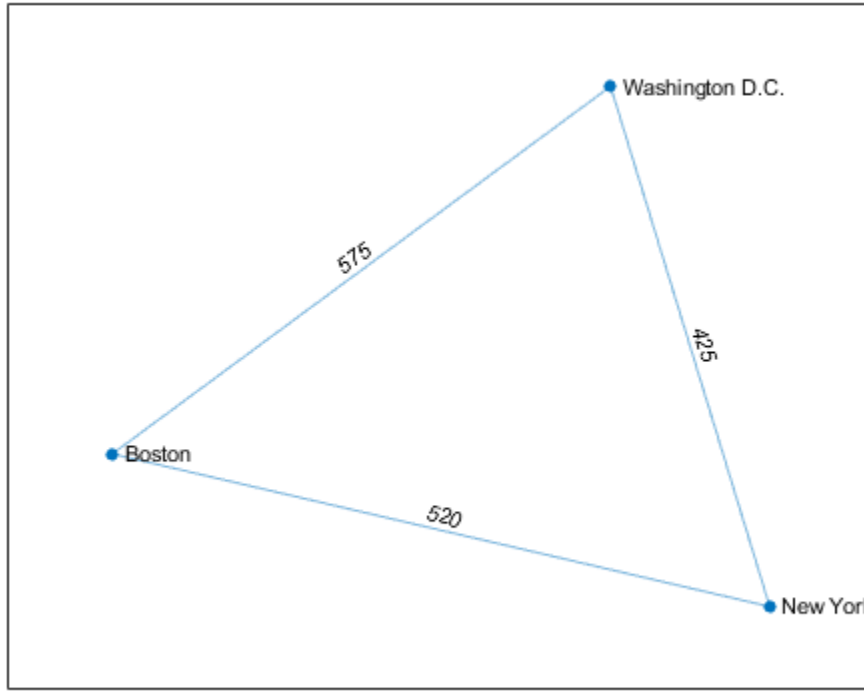
Use the `'min'` selection method and specify the value of `'PickVariable'` as the `'Time'` variable to find the quickest mode of transport between each set of nodes.

```
t = simplify(G,'min','PickVariable','Time');
plot(t,'EdgeLabel',cellstr(t.Edges.Mode))
```



Use the `'sum'` aggregation method and specify the value of `'AggregationVariables'` as `'Cost'` to compute how much money is made on each connection.

```
c = simplify(G,'sum','AggregationVariables','Cost');
plot(c,'EdgeLabel',c.Edges.Cost)
```



Input Arguments

G — Input graph
graph object | digraph object

Input graph, specified as either a `graph` or `digraph` object. Use `graph` to create an undirected graph or `digraph` to create a directed graph.

Example: `G = graph(1,2)`

Example: `G = digraph([1 2],[2 3])`

pickmethod — Edge picking method
`'first'` (default) | `'last'` | `'min'` | `'max'`

Edge picking method, specified as `'first'`, `'last'`, `'min'`, or `'max'`. The edge picking method provides a way to choose which of several edges to preserve when more than one edge exists between the same two nodes.

- If the method is `'first'` or `'last'`, then `simplify` preserves only the first or last edge that occurs in the edges table `G.Edges`.
- If the method is `'min'` or `'max'`, then `simplify` preserves only the edge with minimum or maximum weight. The `Weight` variable must exist in `G.Edges`, unless you use the `'PickVariable'` name-value pair to base the selection on a different variable.

Example: `simplify(G,'last')`

aggregatemethod — Aggregation method
`'sum'` | `'mean'`

Aggregation method, specified as either `'sum'` or `'mean'`. The aggregation method provides a way to combine several edges into a single edge when there is more than one edge between the same two nodes.

By default, `simplify` only sums or averages the edge weights in the graph and drops all other edge properties. However, you can use the `'AggregationVariables'` name-value pair to specify which numeric variables in `G.Edges` to preserve and aggregate.

Example: `simplify(G,'sum')`

selfloopflag — Toggle to keep self-loops
`'omitselfloops'` (default) | `'keepselfloops'`

Toggle to keep self-loops, specified as either:

- `'omitselfloops'` — Remove all self-loops from the graph. This is the default.
- `'keepselfloops'` — Nodes with one or more self-loops have a single self-loop in the simplified graph.

Example: `simplify(G,'sum','keepselfloops')`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `H = simplify(G,'sum','AggregationVariables',{'Var1' 'Var2'})`

`PickVariable` — Variable to base edge selection on
`'Weight'` (default) | variable name | numeric index

Variable to base edge selection on, specified as the comma-separated pair consisting of `'PickVariable'` and a variable name or numeric index. Use this name-value pair to select an edge variable in `G.Edges` other than `'Weight'` to use with the `'min'` or `'max'` picking methods. `simplify` preserves only the edge with the minimum or maximum value of the selected variable when there are several edges between the same two nodes.

Example: `simplify(G,'min','PickVariable',3)`

Example: `simplify(G,'min','PickVariable','var3')`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`AggregationVariables` — Variables to aggregate
`'Weight'` (default) | variable name | cell array of variable names | string array | numeric vector | logical vector | function handle

Variables to aggregate, specified as the comma-separated pair consisting of `'AggregationVariables'` and a variable name, cell array of variable names, string array, numeric vector, logical vector, or function handle. Use this name-value pair to select one or more edge variables in `G.Edges` that can be combined with the `'sum'` or `'mean'` aggregation methods. `simplify` combines the values of these variables into a single value for one new edge when there are several edges between the same two nodes. The value of `'AggregationVariables'` can be one of the following:

- Character vector specifying a single table variable name
- Cell array of character vectors where each element is a table variable name
- String array specifying one or more variable names
- Vector of table variable indices

- Logical vector whose elements correspond to table variables, where `true` includes the corresponding variable and `false` excludes it
- A function handle that takes the `G.Edges` table as input and returns a logical scalar, such as `@isnumeric`

Example: `simplify(G,'sum','AggregationVariables',[4 5 6])`

Example: `simplify(G,'mean','AggregationVariables',{'var5 var7'})`

Example: `simplify(G,'mean','AggregationVariables',@isnumeric)`

Data Types: `single` | `double` | `logical` | `function_handle` | `char` | `string` | `cell`

Output Arguments

`H` — Simplified graph
`graph` object | `digraph` object

Simplified graph, returned as a `graph` or `digraph` object. `H` does not contain any repeated edges between the same two nodes, such that `ismultigraph(H)` returns logical `0` (`false`). Self-loops also are removed, unless you specify the `'keepselfloops'` option.

`eind` — Edge indices
vector

Edge indices, returned as a vector. The edge in `H` that represents edge `i` in `G` is given by `H.Edges(eind(i),:)`. If edge `i` in `G` is a self-loop that was removed, then `eind(i)` is `0`.

`ecount` — Edge counts
vector

Edge counts, returned as a vector. `ecount(i)` is the number of edges in `G` that correspond to edge `i` in `H`.

See Also

`digraph` | `graph` | `ismultigraph`

Introduced in R2018a

subgraph

Extract subgraph

Syntax

```
H = subgraph(G,nodeIDs)
H = subgraph(G,idx)
```

Description

`H = subgraph(G,nodeIDs)` returns a subgraph of `G` that contains only the nodes specified by `nodeIDs`.

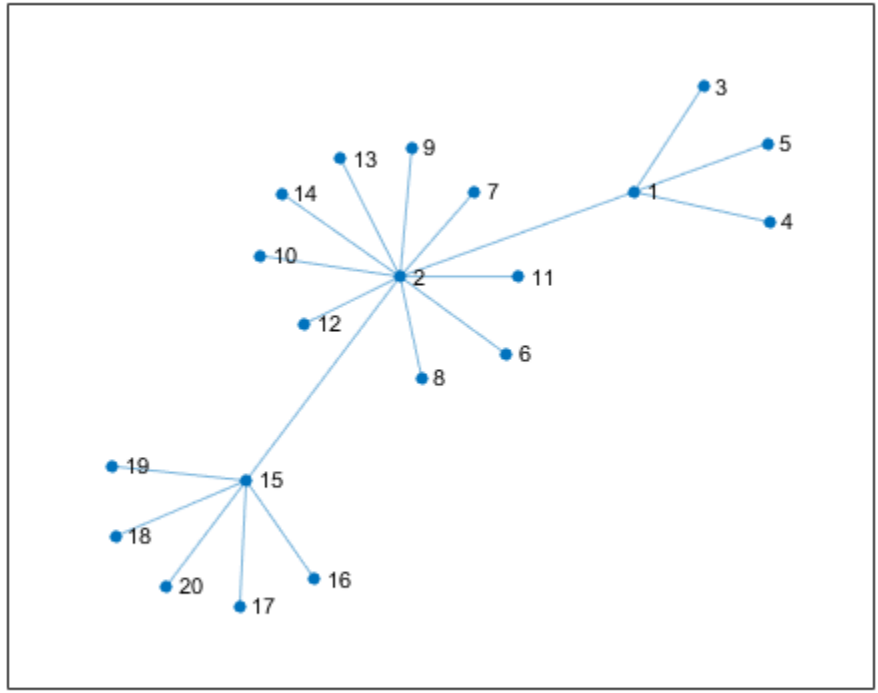
`H = subgraph(G,idx)` specifies the subgraph nodes using a logical vector.

Examples

Extract Subgraph

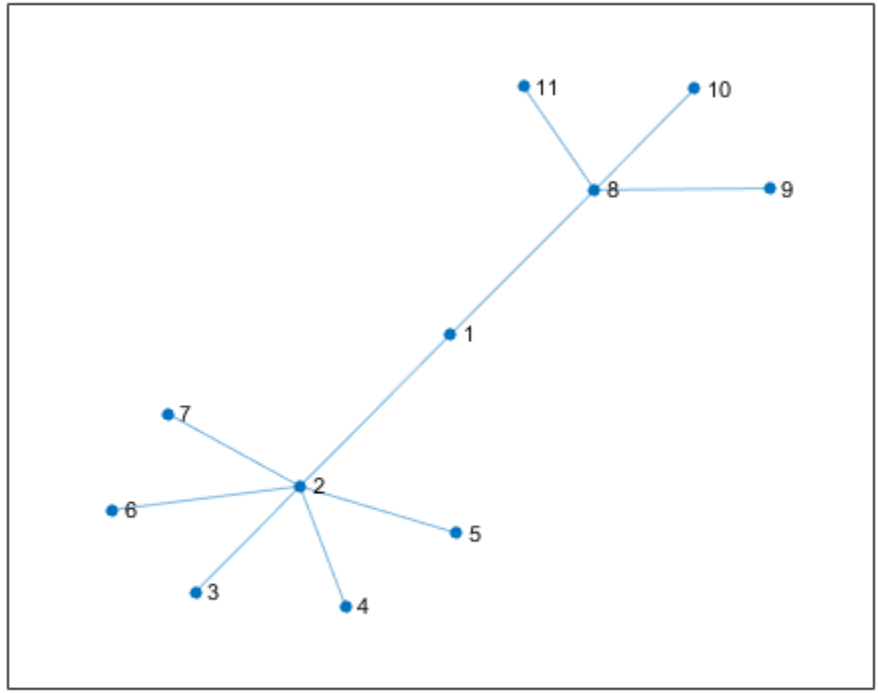
Create and plot a graph.

```
s = [1 1 1 1 2 2 2 2 2 2 2 2 2 2 15 15 15 15 15];
t = [3 5 4 2 14 6 11 12 13 10 7 9 8 15 16 17 19 18 20];
G = graph(s,t);
plot(G,'Layout','force')
```



Extract a subgraph from G by specifying which nodes to include. The node numbering in the subgraph is reset.

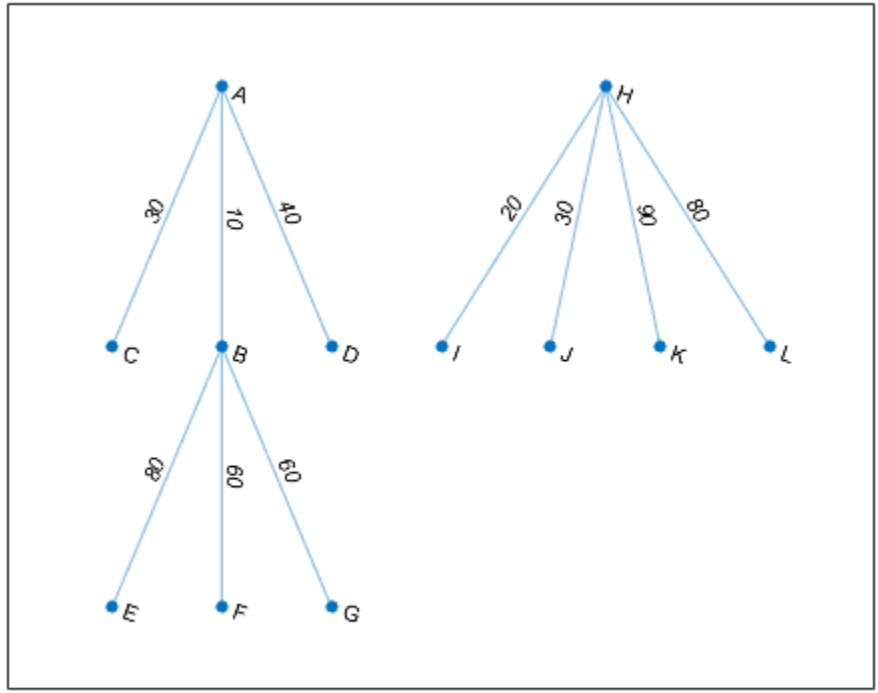
```
idx = [2 15 16 17 18 19 20 1 3 4 5];
H = subgraph(G,idx);
plot(H,'Layout','force')
```



Subgraph with Node and Edge Properties

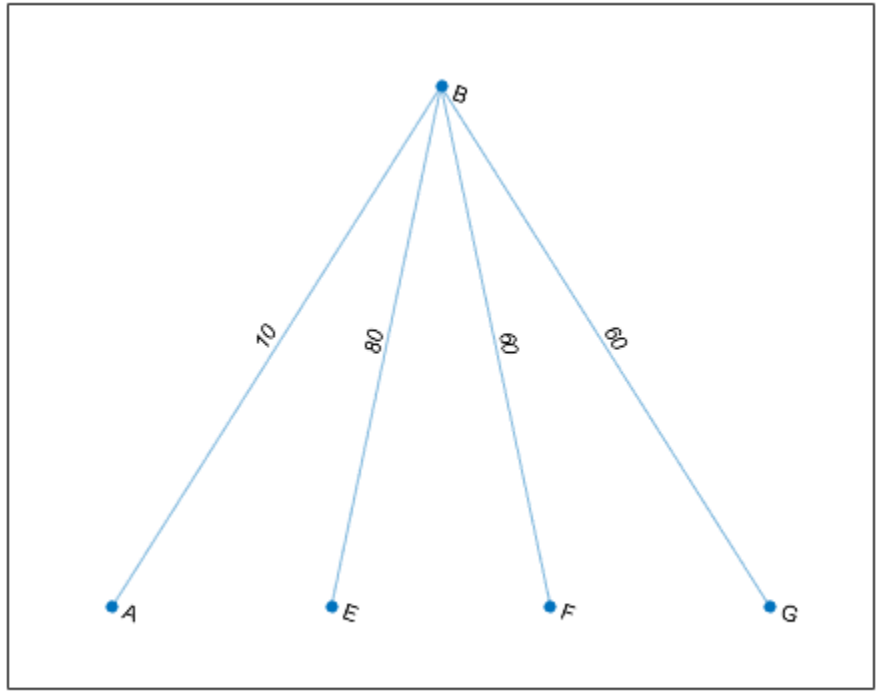
Create and plot a weighted graph with named nodes.

```
s = [1 1 1 2 2 2 8 8 8 8];
t = [2 3 4 5 6 7 9 10 11 12];
weights = [10 30 40 80 60 60 20 30 90 80];
names = {'A' 'B' 'C' 'D' 'E' 'F' 'G' 'H' 'I' 'J' 'K' 'L'};
G = graph(s,t,weights,names);
plot(G,'EdgeLabel',G.Edges.Weight)
```



Extract a subgraph that contains node `'B'` and all of its neighbors. `subgraph` preserves the node names and edge weights. However, the numeric node IDs in `H` are renumbered compared to `G`.

```
N = neighbors(G,'B');
H = subgraph(G, ['B'; N]);
plot(H,'EdgeLabel',H.Edges.Weight)
```



Input Arguments

G — Input graph
graph object | digraph object

Input graph, specified as either a `graph` or `digraph` object. Use `graph` to create an undirected graph or `digraph` to create a directed graph.

Example: `G = graph(1,2)`

Example: `G = digraph([1 2],[2 3])`

`nodeIDs` — Node identifiers
node indices | node names

Node identifiers, specified as one or more node indices or node names. `nodeIDs` selects a subset of the nodes in `G` to generate the subgraph, `H`.

This table shows the different ways to refer to one or more nodes either by their numeric node indices or by their node names.

Form	Single Node	Multiple Nodes
Node index	Scalar **Example:** `1`	Vector **Example:** `[1 2 3]`
Node name	Character vector **Example:** `'A'`	Cell array of character vectors **Example:** `{'A' 'B' 'C'}`
	String scalar **Example:** `"A"`	String array **Example:** `["A" "B" "C"]`

Example: `H = subgraph(G,[1 2 5])`

Example: `H = subgraph(G,{'A' 'B' 'E'})`

`idx` — Node selection vector
vector

Node selection vector, specified as a logical vector. The subgraph contains only the nodes `J` for which `idx(J)` is logical `1` (`true`). The index of node `J` in `H` is `I(J)`, where `I = find(idx)`.

Example: `subgraph(G,degree(G)>2)`

Data Types: `logical`

Output Arguments

`H` — Subgraph
`graph` object | `digraph` object

Subgraph, returned as a `graph` or `digraph` object. `H` contains only the nodes that were selected with `nodeIDs` or `idx`. Other nodes in `G` (and the edges connecting to those nodes) are discarded. The node properties and edge properties of the selected nodes and edges are carried over from `G` into `H`.

See `graph` or `digraph` for more information about graph objects.

See Also

`digraph` | `graph` | `reordernodes`

Topics

"Directed and Undirected Graphs"
"Modify Nodes and Edges of Existing Graph"
"Add Graph Node Names, Edge Weights, and Other Attributes"

Introduced in R2015b

graph

Graph with undirected edges

Description

`graph` objects represent undirected graphs, which have direction-less edges connecting the nodes. After you create a graph object, you can learn more about the graph by using object functions to perform queries against the object. For example, you can add or remove nodes or edges, determine the shortest path between two nodes, or locate a specific node or edge.

```
G = graph([1 1], [2 3]);
e = G.Edges
G = addedge(G,2,3)
G = addnode(G,4)
plot(G)
```

Creation

Syntax

```
G = graph
```

```
G = graph(A)
G = graph(A,nodenames)
G = graph(A,NodeTable)
G = graph(A,___,type)
G = graph(A,___,'omitselfloops')
```

```
G = graph(s,t)
G = graph(s,t,weights)
G = graph(s,t,weights,nodenames)
G = graph(s,t,weights,NodeTable)
G = graph(s,t,weights,num)
G = graph(s,t,___,'omitselfloops')
```

```
G = graph(s,t,EdgeTable, ___ )

G = graph(EdgeTable)
G = graph(EdgeTable,NodeTable)
G = graph(EdgeTable, ___ ,'omitselfloops')
```

Description

`G = graph` creates an empty undirected graph object, `G`, which has no nodes or edges.

`G = graph(A)` creates a weighted graph using a square, symmetric adjacency matrix, `A`. The location of each nonzero entry in `A` specifies an edge for the graph, and the weight of the edge is equal to the value of the entry. For example, if `A(2,1) = 10`, then `G` contains an edge between node 2 and node 1 with a weight of 10.

`G = graph(A,nodenames)` additionally specifies node names. The number of elements in `nodenames` must be equal to `size(A,1)`.

`G = graph(A,NodeTable)` specifies node names (and possibly other node attributes) using a table, `NodeTable`. The table must have the same number of rows as `A`. Specify node names using the table variable `Name`.

`G = graph(A, ___ ,type)` specifies a triangle of the adjacency matrix to use in constructing the graph. You must specify `A` and optionally can specify `nodenames` or `NodeTable`. To use only the upper or lower triangle of `A` to construct the graph, `type` can be either `'upper'` or `'lower'`.

`G = graph(A, ___ ,'omitselfloops')` ignores the diagonal elements of `A` and returns a graph without any self-loops. You can use any of the input argument combinations in previous syntaxes.

`G = graph(s,t)` specifies graph edges `(s,t)` in node pairs. `s` and `t` can specify node indices or node names.

`G = graph(s,t,weights)` also specifies edge weights with the array `weights`.

`G = graph(s,t,weights,nodenames)` specifies node names using the cell array of character vectors or string array, `nodenames`. `s` and `t` cannot contain node names that are not in `nodenames`.

`G = graph(s,t,weights,NodeTable)` specifies node names (and possibly other node attributes) using a table, `NodeTable`. Specify node names using the `Name` table variable. `s` and `t` cannot contain node names that are not in `NodeTable`.

`G = graph(s,t,weights,num)` specifies the number of nodes in the graph with the numeric scalar `num`.

`G = graph(s,t, ___ ,'omitselfloops')` does not add any self-loops to the graph. That is, any `k` that satisfies `s(k) == t(k)` is ignored. You can use any of the input argument combinations in previous syntaxes.

`G = graph(s,t,EdgeTable, ___ )` uses a table to specify edge attributes instead of specifying `weights`. The `EdgeTable` input must be a table with a row for each corresponding pair of elements in `s` and `t`. Specify edge weights using the table variable `Weight`.

`G = graph(EdgeTable)` uses the table `EdgeTable` to define the graph. With this syntax, the first variable in `EdgeTable` must be named `EndNodes`, and it must be a two-column array defining the edge list of the graph.

`G = graph(EdgeTable,NodeTable)` additionally specifies the names (and possibly other attributes) of the graph nodes using a table, `NodeTable`.

`G = graph(EdgeTable, ___ ,'omitselfloops')` does not add self-loops to the graph. That is, any `k` that satisfies `EdgeTable.EndNodes(k,1) == EdgeTable.EndNodes(k,2)` is ignored. You must specify `EdgeTable` and optionally can specify `NodeTable`.

Input Arguments

`A` — Adjacency matrix

matrix

Adjacency matrix, specified as a full or sparse, numeric matrix. The entries in `A` specify the network of connections (edges) between the nodes of the graph. The location of each nonzero entry in `A` specifies an edge between two nodes. The value of that entry provides the edge weight. A logical adjacency matrix results in an unweighted graph.

Nonzero entries on the main diagonal of `A` specify *self-loops*, or nodes that are connected to themselves with an edge. Use the `'omitselfloops'` input option to ignore diagonal entries.

A must be symmetric unless the `type` input is specified. Use `issymmetric` to confirm matrix symmetry. For triangular adjacency matrices, specify `type` to use only the upper or lower triangle.

Example: `A = [0 1 5; 1 0 0; 5 0 0]` describes a graph with three nodes and two edges. The edge between node 1 and node 2 has a weight of `1`, and the edge between node 1 and node 3 has a weight of `5`.

Data Types: `single` | `double` | `logical`

`nodenames` — Node names
cell array of character vectors | string array

Node names, specified as a cell array of character vectors or string array. `nodenames` must have length equal to `numnodes(G)` so that it contains a nonempty, unique name for each node in the graph.

Example: `G = graph(A,{'n1','n2','n3'})` specifies three node names for a 3-by-3 adjacency matrix, `A`.

Data Types: `cell` | `string`

`type` — Type of adjacency matrix
`'upper'` | `'lower'`

Type of adjacency matrix, specified as either `'upper'` or `'lower'`.

Example: `G = graph(A,'upper')` uses only the upper triangle of `A` to construct the graph, `G`.

`s,t` — Node pairs (as separate arguments)
node indices | node names

Node pairs, specified as node indices or node names. `graph` creates edges between the corresponding nodes in `s` and `t`, which must both be numeric, or both be character vectors, cell arrays of character vectors, string arrays, or categorical arrays. In all cases, `s` and `t` must have the same number of elements.

- If `s` and `t` are numeric, then they correspond to indices of graph nodes. Numeric node indices must be positive integers greater than or equal to 1.
- If `s` and `t` are character vectors, cell arrays of character vectors, or string arrays, then they specify names for the nodes. The `Nodes` property of the graph is a table containing a `Name` variable with the node names, `G.Nodes.Name`.

- If `s` and `t` are categorical arrays, then the categories in `s` and `t` are used as the node names in the graph. This can include categories that are not elements in `s` or `t`.
- If `s` and `t` specify multiple edges between the same two nodes, then the result is a multigraph.

This table shows the different ways to refer to one or more nodes either by their numeric node indices or by their node names.

Form	Single Node	Multiple Nodes
Node index	Scalar **Example:** `1`	Vector **Example:** `[1 2 3]`
Node name	Character vector **Example:** `'A'`	Cell array of character vectors **Example:** `{'A' 'B' 'C'}`
	String scalar **Example:** `"A"`	String array **Example:** `["A" "B" "C"]`
	Categorical array **Example:** `categorical("A")`	Categorical array **Example:** `categorical(["A" "B" "C"])`

Example: `G = graph([1 2 3],[2 4 5])` creates a graph with five nodes and three edges.

Example: `G = graph({'Boston' 'New York' 'Washington D.C.'},{'New York' 'New Jersey' 'Pittsburgh'})` creates a graph with five named nodes and three edges.

`weights` — Edge weights

scalar | vector | matrix | multidimensional array | `[]`

Edge weights, specified as a scalar, vector, matrix, or multidimensional array. `weights` must be a scalar or an array with the same number of elements as `s` and `t`.

`graph` stores the edge weights as a `Weight` variable in the `G.Edges` property table. To add or change weights after creating a graph, you can modify the table variable directly, for example, `G.Edges.Weight = [25 50 75]'`.

If you specify `weights` as an empty array `[]`, then it is ignored.

Example: `G = graph([1 2],[2 3],[100 200])` creates a graph with three nodes and two edges. The edges have weights of `100` and `200`.

Data Types: `single` | `double`

`num` — Number of graph nodes
positive scalar integer

Number of graph nodes, specified as a positive scalar integer. `num` must be greater than or equal to the largest elements in `s` and `t`.

Example: `G = graph([1 2],[2 3],[],5)` creates a graph with three connected nodes and two isolated nodes.

`EdgeTable` — Table of edge information
table

Table of edge information. If you do not specify `s` and `t`, then the first variable in `EdgeTable` is required to be a two-column matrix, cell array of character vectors, or string array called `EndNodes` that defines the graph edges. For edge weights, use the variable `Weight`, since this table variable name is used by some graph functions. If there is a variable `Weight`, then it must be a numeric column vector. See `table` for more information on constructing a table.

After creating a graph, query the edge information table using `G.Edges`.

Example: `EdgeTable = table([1 2; 2 3; 3 5; 4 5],'VariableNames', {'EndNodes'})`

Data Types: `table`

`NodeTable` — Table of node information
table

Table of node information. `NodeTable` can contain any number of variables to describe attributes of the graph nodes. For node names, use the variable `Name`, since this variable name is used by some graph functions. If there is a variable `Name`, then it must be a cell array of character vectors or string array specifying a unique name in each row. See `table` for more information on constructing a table.

After the graph is created, query the node information table using `G.Nodes`.

Example: `NodeTable = table({'a'; 'b'; 'c'; 'd'},'VariableNames', {'Name'})`

Data Types: `table`

Output Arguments

`G` — Undirected graph
`graph` object

Undirected graph, returned as a `graph` object.

Properties

Edges — Edges of graph
table

Edges of graph, returned as a table. By default this is an `M`-by-`1` table, where `M` is the number of edges in the graph.

- To add new edge properties to the graph, create a new variable in the `Edges` table.
- To add or remove edges from the graph, use the `addedge` or `rmedge` object functions.

Example: `G.Edges` returns a table listing the edges in the graph

Example: `G.Edges.Weight` returns a numeric vector of the edge weights.

Example: `G.Edges.Weight = [10 20 30 55]'` specifies new edge weights for the graph.

Example: `G.Edges.NormWeight = G.Edges.Weight/sum(G.Edges.Weight)` adds a new edge property to the table containing the normalized weights of the edges.

Data Types: `table`

Nodes — Nodes of graph
table

Nodes of graph, returned as a table. By default this is an empty `N`-by-`0` table, where `N` is the number of nodes in the graph.

- To add new node properties to the graph, create a new variable in the `Nodes` table.

- To add or remove nodes from the graph, use the `addnode` or `rmnode` object functions.

Example: `G.Nodes` returns a table listing the node properties of the graph. This table is empty by default.

Example: `G.Nodes.Names = {'Montana', 'New York', 'Washington', 'California'}'` adds node names to the graph by adding the variable `Names` to the `Nodes` table.

Example: `G.Nodes.WiFi = logical([1 0 0 1 1]')` adds the variable `WiFi` to the `Nodes` table. This property specifies that certain airports have wireless internet coverage.

Data Types: `table`

Object Functions

Access and Modify Nodes and Edges

addedge	Add new edge to graph
rmedge	Remove edge from graph
addnode	Add new node to graph
rmnode	Remove node from graph
findedge	Locate edge in graph
findnode	Locate node in graph
numedges	Number of edges in graph
numnodes	Number of nodes in graph
edgecount	Number of edges between two nodes
reordernodes	Reorder graph nodes
subgraph	Extract subgraph

Search and Structure

bfsearch	Breadth-first graph search
dfsearch	Depth-first graph search
centrality	Measure node importance
conncomp	Connected graph components
biconncomp	Biconnected graph components
bctree	Block-cut tree graph
maxflow	Maximum flow in graph
minspantree	Minimum spanning tree of graph
isisomorphic	Determine whether two graphs are isomorphic

isomorphism	Compute isomorphism between two graphs
ismultigraph	Determine whether graph has multiple edges
simplify	Reduce multigraph to simple graph

Shortest Path

shortestpath	Shortest path between two single nodes
shortestpathtree	Shortest path tree from node
distances	Shortest path distances of all node pairs

Matrix Representation

adjacency	Graph adjacency matrix
incidence	Graph incidence matrix
laplacian	Graph Laplacian matrix

Node Information

degree	Degree of graph nodes
neighbors	Neighbors of graph node
nearest	Nearest neighbors within radius
outedges	Outgoing edges from node

Visualization

plot	Plot graph nodes and edges

Examples

Create and Modify Graph Object

Create a `graph` object with three nodes and two edges. One edge is between node 1 and node 2, and the other edge is between node 1 and node 3.

```
G = graph([1 1],[2 3])
```

```
G = 
  graph with properties:

    Edges: [2x1 table]
```

```
    Nodes: [3x0 table]
```

View the edge table of the graph.

```
G.Edges
```

```
ans=2×1 table
    EndNodes
    ________

     1    2
     1    3
```

Add node names to the graph, and then view the new node and edge tables. The end nodes of each edge are now expressed using their node names.

```
G.Nodes.Name = {'A' 'B' 'C'}';
G.Nodes
```

```
ans=3×1 table
    Name
    _____

    {'A'}
    {'B'}
    {'C'}
```

```
G.Edges
```

```
ans=2×1 table
       EndNodes
    _____________

    {'A'}    {'B'}
    {'A'}    {'C'}
```

You can add or modify extra variables in the `Nodes` and `Edges` tables to describe attributes of the graph nodes or edges. However, you cannot directly change the number of nodes or edges in the graph by modifying these tables. Instead, use the `addedge`, `rmedge`, `addnode`, or `rmnode` functions to modify the number of nodes or edges in a graph.

For example, add an edge to the graph between nodes 2 and 3 and view the new edge list.

```
G = addedge(G,2,3)
```

```
G = 
  graph with properties:

    Edges: [3x1 table]
    Nodes: [3x1 table]
```

```
G.Edges
```

```
ans=3×1 table
       EndNodes
    _____________

    {'A'}    {'B'}
    {'A'}    {'C'}
    {'B'}    {'C'}
```

Adjacency Matrix Graph Construction

Create a symmetric adjacency matrix, A, that creates a complete graph of order 4. Use a logical adjacency matrix to create a graph without weights.

```
A = ones(4) - diag([1 1 1 1])
```

```
A = 4×4

     0     1     1     1
     1     0     1     1
     1     1     0     1
     1     1     1     0
```

```
G = graph(A~=0)
```

```
G = 
  graph with properties:

    Edges: [6x1 table]
```

```
    Nodes: [4x0 table]
```

View the edge list of the graph.

```
G.Edges
```

```
ans=6×1 table
    EndNodes
    ________

     1    2
     1    3
     1    4
     2    3
     2    4
     3    4
```

Adjacency Matrix Construction with Node Names

Create an upper triangular adjacency matrix.

```
A = triu(magic(4))
```

```
A = 4×4

    16     2     3    13
     0    11    10     8
     0     0     6    12
     0     0     0     1
```

Create a graph with named nodes using the adjacency matrix. Specify `'omitselfloops'` to ignore the entries on the diagonal of `A`, and specify `type` as `'upper'` to indicate that `A` is upper-triangular.

```
names = {'alpha' 'beta' 'gamma' 'delta'};
G = graph(A,names,'upper','omitselfloops')
```

```
G =
  graph with properties:
```

```
    Edges: [6x2 table]
    Nodes: [4x1 table]
```

View the edge and node information.

```
G.Edges
```

```
ans=6×2 table
           EndNodes           Weight
    ______________________    ______

    {'alpha'}    {'beta' }       2
    {'alpha'}    {'gamma'}       3
    {'alpha'}    {'delta'}      13
    {'beta' }    {'gamma'}      10
    {'beta' }    {'delta'}       8
    {'gamma'}    {'delta'}      12
```

```
G.Nodes
```

```
ans=4×1 table
      Name
    _________

    {'alpha'}
    {'beta' }
    {'gamma'}
    {'delta'}
```

Edge List Graph Construction

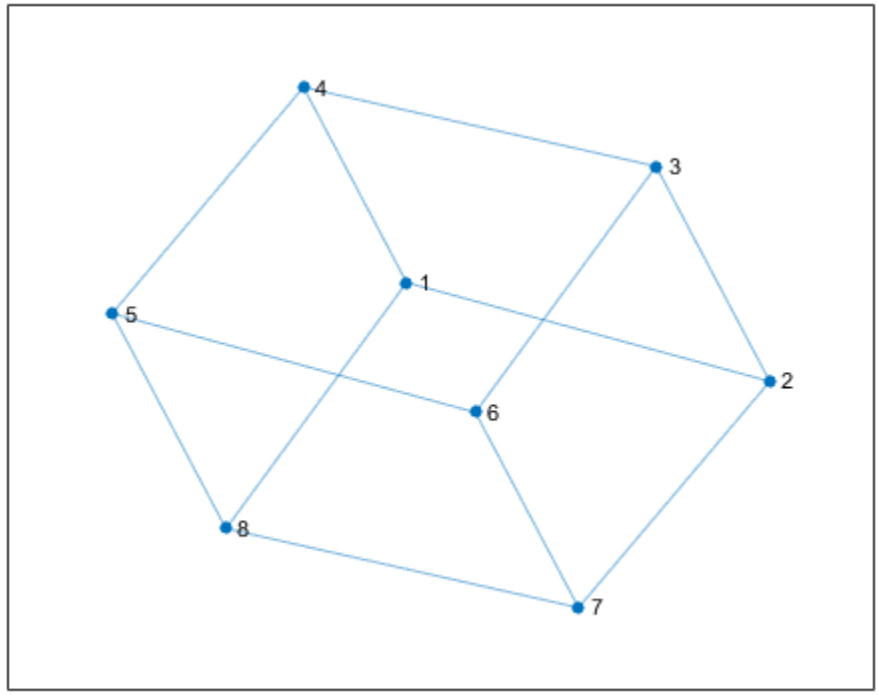
Create and plot a cube graph using a list of the end nodes of each edge.

```
s = [1 1 1 2 2 3 3 4 5 5 6 7];
t = [2 4 8 3 7 4 6 5 6 8 7 8];
G = graph(s,t)
```

```
G =
  graph with properties:
```

```
    Edges: [12x1 table]
    Nodes: [8x0 table]
```

```
plot(G)
```



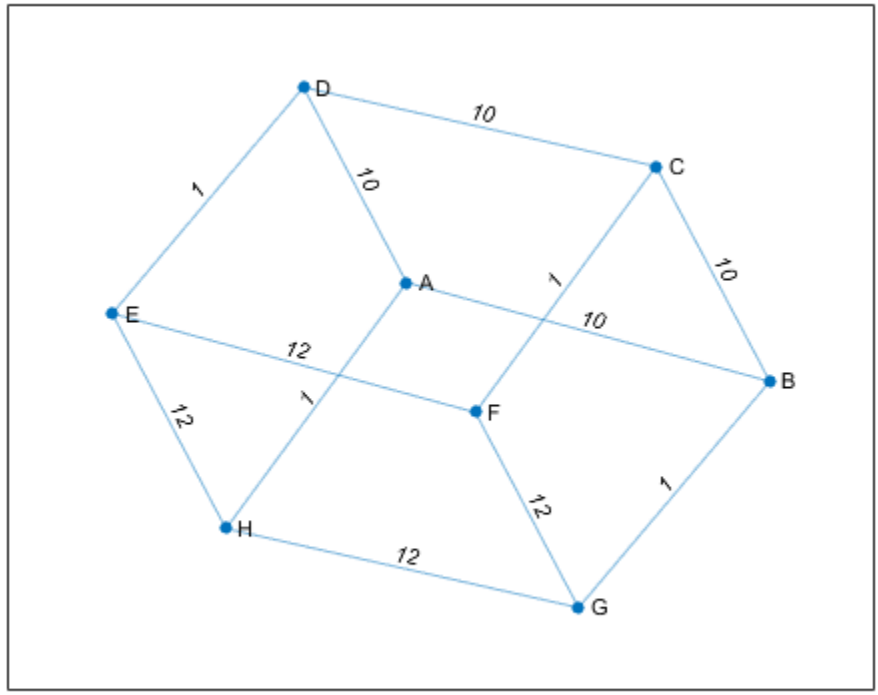
Edge List Graph Construction with Node Names and Edge Weights

Create and plot a cube graph using a list of the end nodes of each edge. Specify node names and edge weights as separate inputs.

```
s = [1 1 1 2 2 3 3 4 5 5 6 7];
t = [2 4 8 3 7 4 6 5 6 8 7 8];
weights = [10 10 1 10 1 10 1 1 12 12 12 12];
names = {'A' 'B' 'C' 'D' 'E' 'F' 'G' 'H'};
G = graph(s,t,weights,names)
```

```
G =
  graph with properties:

    Edges: [12x2 table]
    Nodes: [8x1 table]
```

```
plot(G,'EdgeLabel',G.Edges.Weight)
```



Edge List Construction with Extra Nodes

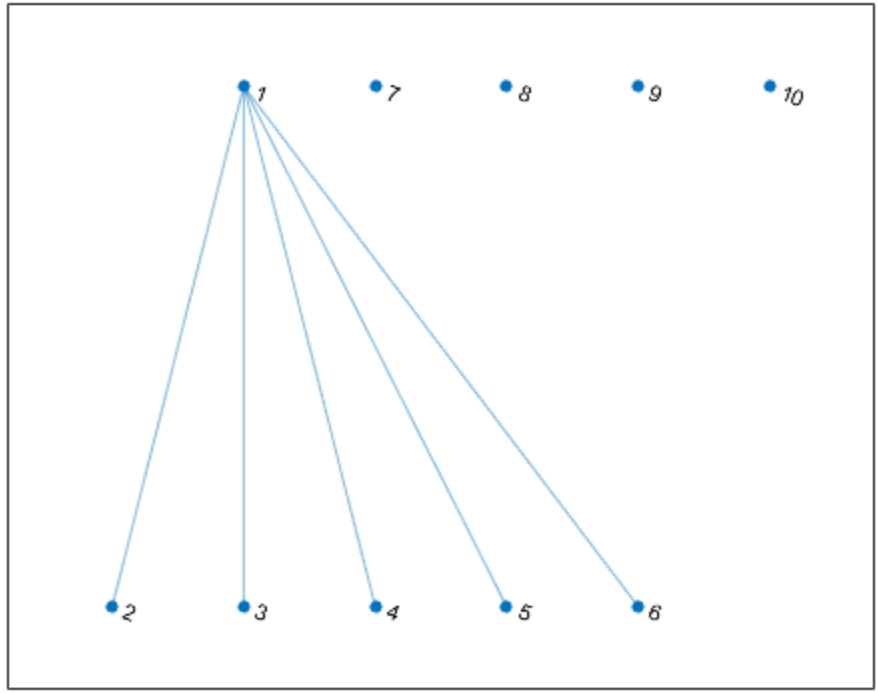
Create a weighted graph using a list of the end nodes of each edge. Specify that the graph should contain a total of 10 nodes.

```
s = [1 1 1 1 1];
t = [2 3 4 5 6];
weights = [5 5 5 6 9];
G = graph(s,t,weights,10)
```

```
G = 
  graph with properties:

    Edges: [5x2 table]
    Nodes: [10x0 table]
```

Plot the graph. The extra nodes are disconnected from the primary connected component.

```
plot(G)
```



Add Nodes and Edges to Empty Graph

Create an empty graph object, `G`.

```
G = graph;
```

Add three nodes and three edges to the graph. The corresponding entries in `s` and `t` define the end nodes of the graph edges. `addedge` automatically adds the appropriate nodes to the graph if they are not already present.

```
s = [1 2 1];
t = [2 3 3];
G = addedge(G,s,t)
```

```
G = 
  graph with properties:

    Edges: [3x1 table]
    Nodes: [3x0 table]
```

View the edge list. Each row describes an edge in the graph.

```
G.Edges
```

```
ans=3×1 table
    EndNodes
    ________

     1    2
     1    3
     2    3
```

For the best performance, construct graphs all at once using a single call to `graph`. Adding nodes or edges in a loop can be slow for large graphs.

Graph Construction with Tables

Create an edge table that contains the variables `EndNodes`, `Weight`, and `Code`. Then create a node table that contains the variables `Name` and `Country`. The variables in each table specify properties of the graph nodes and edges.

```
s = [1 1 1 2 3];
t = [2 3 4 3 4];
weights = [6 6.5 7 11.5 17]';
code = {'1/44' '1/49' '1/33' '44/49' '49/33'}';
EdgeTable = table([s' t'],weights,code, ...
    'VariableNames',{'EndNodes' 'Weight' 'Code'})
```

```
EdgeTable=5×3 table
    EndNodes    Weight      Code
```

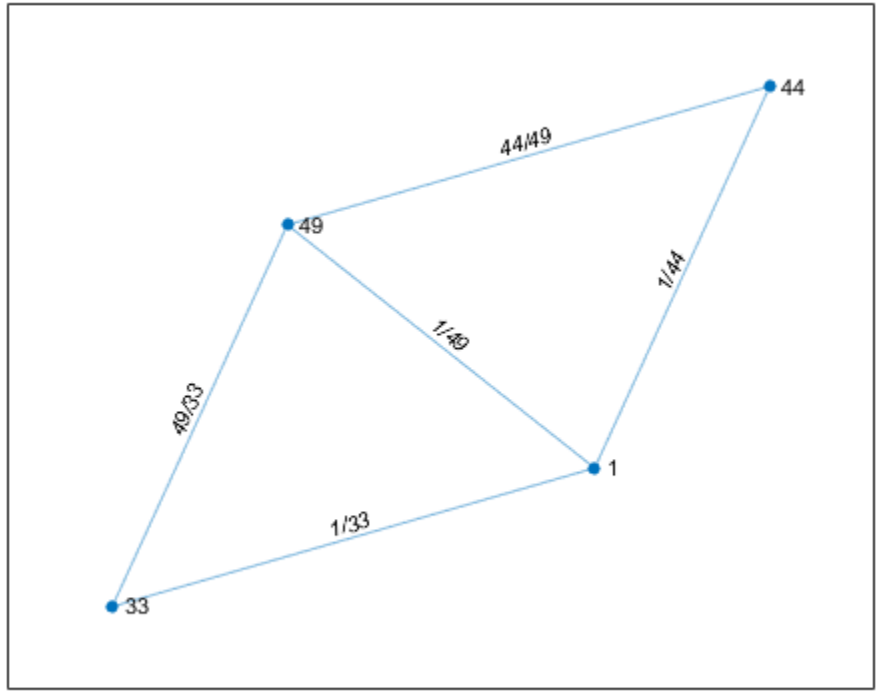
```
    ________    ______    _________

     1    2         6     {'1/44' }
     1    3       6.5     {'1/49' }
     1    4         7     {'1/33' }
     2    3      11.5     {'44/49'}
     3    4        17     {'49/33'}
```

```
names = {'USA' 'GBR' 'DEU' 'FRA'}';
country_code = {'1' '44' '49' '33'}';
NodeTable = table(names,country_code,'VariableNames',{'Name' 'Country'})
```

```
NodeTable=4×2 table
     Name      Country
    _______    _______

    {'USA'}    {'1' }
    {'GBR'}    {'44'}
    {'DEU'}    {'49'}
    {'FRA'}    {'33'}
```

Create a graph using the node and edge tables. Plot the graph using the country codes as node and edge labels.

```
G = graph(EdgeTable,NodeTable);
plot(G,'NodeLabel',G.Nodes.Country,'EdgeLabel',G.Edges.Code)
```



Compatibility Considerations

Change in handling of duplicate edges

Behavior changed in R2018a

`graph`, `digraph`, and `addedge` no longer produce errors when they encounter duplicate edges. Instead, the duplicate edges are added to the graph and the result is a multigraph. The `ismultigraph` function is useful to detect this situation, and `simplify` provides an easy way to remove the extra edges.

See Also

`digraph` | `subgraph`

Topics

"Build Watts-Strogatz Small World Graph Model"
"Use PageRank Algorithm to Rank Websites"
"Directed and Undirected Graphs"
"Modify Nodes and Edges of Existing Graph"
"Add Graph Node Names, Edge Weights, and Other Attributes"
"Graph Plotting and Customization"

Introduced in R2015b

GraphPlot

Graph plot for directed and undirected graphs

Description

Graph plots are the primary way to visualize graphs and networks created using the `graph` and `digraph` functions. After you create a `GraphPlot` object, you can modify aspects of the plot by changing its property values. This is particularly useful for modifying the display of the graph nodes or edges.

Creation

To create a `GraphPlot` object, specify an output argument with the `plot` function. For example:

```
G = graph([1 1 1 1 5 5 5 5],[2 3 4 5 6 7 8 9]);
h = plot(G)
```

Properties

GraphPlot	Graph plot appearance and behavior

Object Functions

layout	Change layout of graph plot
highlight	Highlight nodes and edges in plotted graph
labelnode	Label graph nodes
labeledge	Label graph edges

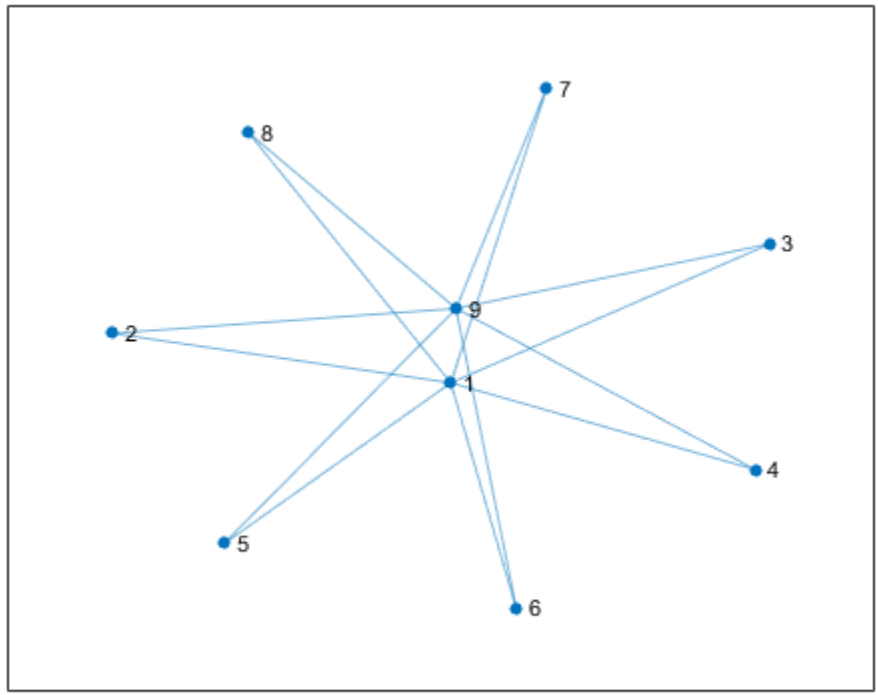
Examples

Adjust Properties of GraphPlot Object

Create a `GraphPlot` object, and then show how to adjust the properties of the object to affect the output display.

Create and plot a graph.

```
s = [1 1 1 1 1 1 1 9 9 9 9 9 9 9];
t = [2 3 4 5 6 7 8 2 3 4 5 6 7 8];
G = graph(s,t);
h = plot(G)
```

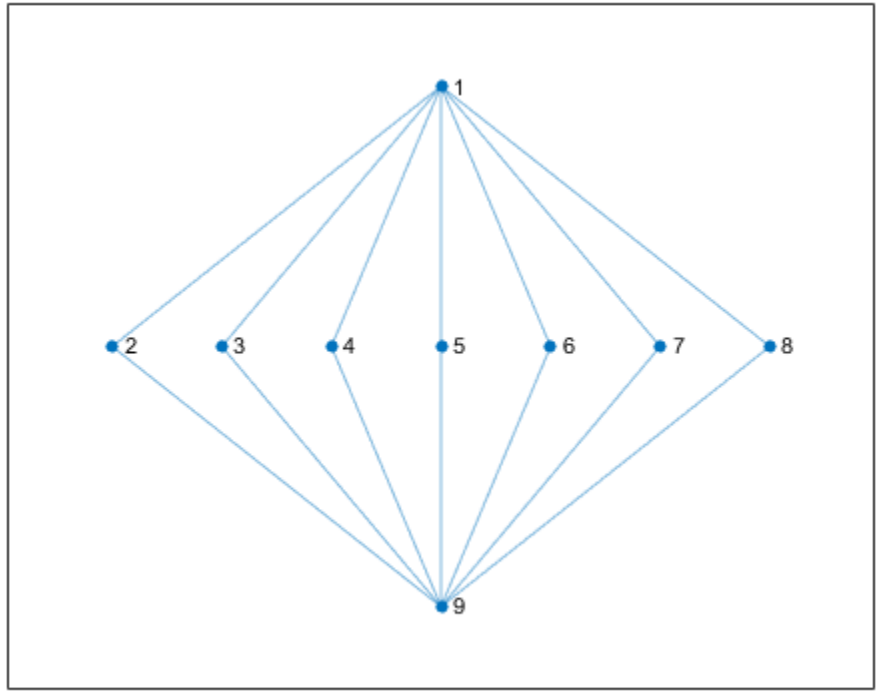



```
h =
  GraphPlot with properties:
```

```
     NodeColor: [0 0.4470 0.7410]
    MarkerSize: 4
        Marker: 'o'
     EdgeColor: [0 0.4470 0.7410]
     LineWidth: 0.5000
     LineStyle: '-'
     NodeLabel: {'1'  '2'  '3'  '4'  '5'  '6'  '7'  '8'  '9'}
     EdgeLabel: {}
         XData: [1x9 double]
         YData: [1x9 double]
         ZData: [0 0 0 0 0 0 0 0 0]

  Show all properties
```

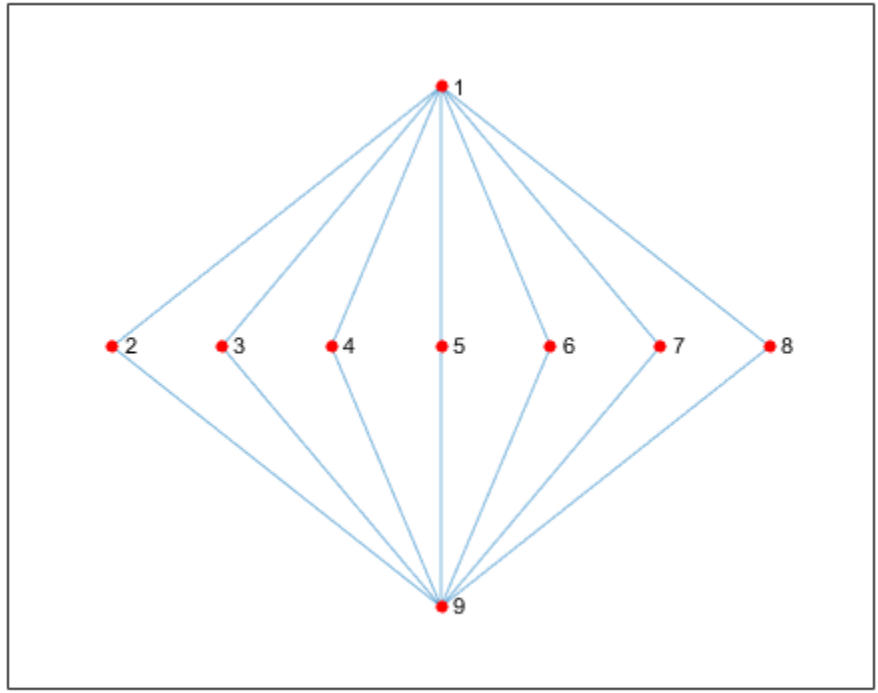
Use custom node coordinates for the graph nodes.

```
h.XData = [0 -3 -2 -1 0 1 2 3 0];
h.YData = [2 0 0 0 0 0 0 0 -2];
```



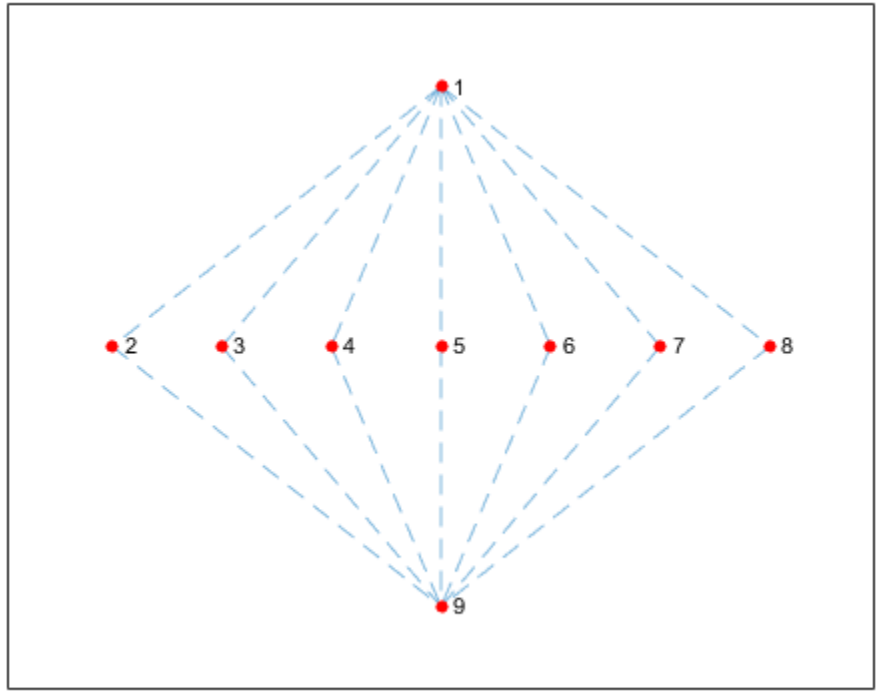
Make the graph nodes red.

```
h.NodeColor = 'r';
```



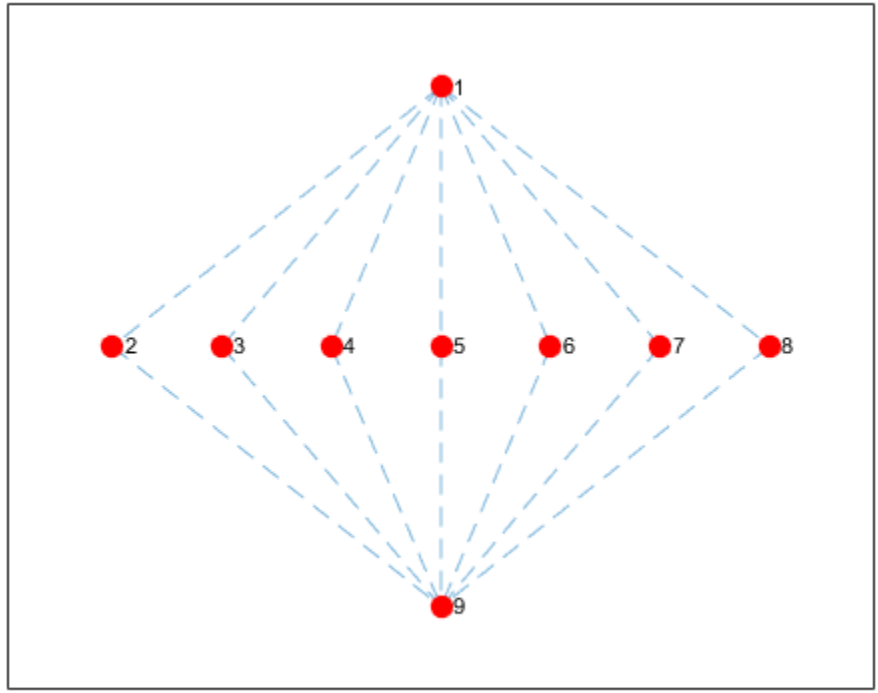
Use dashed lines for the graph edges.

```
h.LineStyle = '--';
```



Increase the size of the nodes.

```
h.MarkerSize = 8;
```



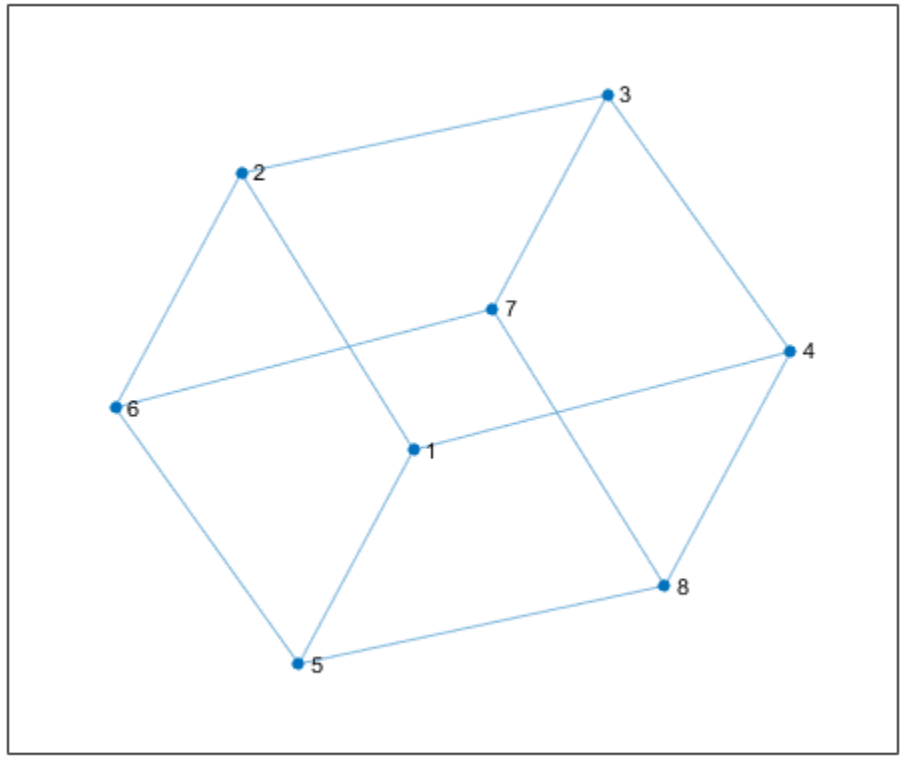
Saving and Loading GraphPlot Objects

Use the `savefig` function to save a graph plot figure.

```
s = [1 1 1 2 2 3 3 4 5 5 6 7];
t = [2 4 5 3 6 4 7 8 6 8 7 8];
G = graph(s,t);
h = plot(G);
savefig('cubegraph.fig');
clear all
close all
```

Use `openfig` to load the graph plot figure back into MATLAB. `openfig` also returns a handle to the figure, `y`.

```
y = openfig('cubegraph.fig');
```



Use the `findobj` function to locate the correct object handle using one of the property values. Using `findobj` allows you to continue manipulating the original `GraphPlot` object used to generate the figure.

```
h = findobj('Marker','o')
```

```
h =
  GraphPlot with properties:

     NodeColor: [0 0.4470 0.7410]
    MarkerSize: 4
        Marker: 'o'
     EdgeColor: [0 0.4470 0.7410]
     LineWidth: 0.5000
     LineStyle: '-'
     NodeLabel: {'1'  '2'  '3'  '4'  '5'  '6'  '7'  '8'}
     EdgeLabel: {}
         XData: [-0.2052 -1.1020 0.8023 1.7577 -0.8023 -1.7577 0.2052 1.1020]
         YData: [-0.4749 1.3919 1.9185 0.1892 -1.9185 -0.1892 0.4749 -1.3919]
         ZData: [0 0 0 0 0 0 0 0]

  Show all properties
```

Compatibility Considerations

Change to default text interpreter

Behavior changed in R2018b

The new `GraphPlot` property `Interpreter` has a default value of `'tex'`. In previous releases, graph node and edge labels displayed text as the literal characters instead of interpreting the text using TeX markup. If you do not want node and edge labels to use TeX markup, then set the `Interpreter` property to `'none'`.

Self-loop display change

Behavior changed in R2018a

Self-loops in the plot of a simple graph are now shaped like a leaf or teardrop. In previous releases, self-loops were displayed as circles.

See Also

`digraph` | `graph`

Topics

GraphPlot
"Graph Plotting and Customization"
"Add Node Properties to Graph Plot Data Cursor"

Introduced in R2015b

GraphPlot Properties

Graph plot appearance and behavior

Description

`GraphPlot` properties control the appearance and behavior of plotted graphs. By changing property values, you can modify aspects of the graph display. Use dot notation to refer to a particular object and property:

```
G = graph([1 1 1 1 5 5 5 5],[2 3 4 5 6 7 8 9]);
h = plot(G);
c = h.EdgeColor;
h.EdgeColor = 'k';
```

Properties

Nodes

`NodeColor` — Node color
`[0 0.4470 0.7410]` (default) | RGB triplet | hexadecimal color code | color name | matrix | `'flat'` | `'none'`

Node color, specified as one of these values:

- `'none'` — Nodes are not drawn.
- `'flat'` — Color of each node depends on the value of `NodeCData`.
- matrix — Each row is an RGB triplet representing the color of one node. The size of the matrix is `numnodes(G)`-by-3.
- RGB triplet, hexadecimal color code, or color name — All nodes use the specified color.

  RGB triplets and hexadecimal color codes are useful for specifying custom colors.

  - An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `plot(G,'NodeColor','k')` creates a graph plot with black nodes.

`Marker` — Node marker symbol

`'o'` (default) | character vector | cell array | string vector

Node marker symbol, specified as one of the values listed in this table, or as a cell array or string vector of such values. The default is to use circular markers for the graph nodes. Specify a cell array of character vectors or string vector to use different markers for each node.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'+'`

Example: `'diamond'`

`MarkerSize` — Node marker size

positive value | vector

Node marker size, specified as a positive value in point units or as a vector of such values. Specify a vector to use different marker sizes for each node in the graph. The default value of `MarkerSize` is 4 for graphs with 100 or fewer nodes, and 2 for graphs with more than 100 nodes.

Example: `10`

`NodeCData` — Color data of node markers
vector

Color data of node markers, specified as a vector with length equal to the number of nodes in the graph. The values in `NodeCData` map linearly to the colors in the current colormap, resulting in different colors for each node in the plotted graph.

Edges

`EdgeColor` — Edge color
`[0 0.4470 0.7410]` (default) | RGB triplet | hexadecimal color code | color name | matrix | `'flat'` | `'none'`

Edge color, specified as one of these values:

- `'none'` — Edges are not drawn.
- `'flat'` — Color of each edge depends on the value of `EdgeCData`.
- matrix — Each row is an RGB triplet representing the color of one edge. The size of the matrix is `numedges(G)`-by-3.
- RGB triplet, hexadecimal color code, or color name — All edges use the specified color.

  RGB triplets and hexadecimal color codes are useful for specifying custom colors.

  - An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
  - A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

  Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `plot(G,'EdgeColor','r')` creates a graph plot with red edges.

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'` | cell array | string vector

Line style, specified as one of the line styles listed in this table, or as a cell array or string vector of such values. Specify a cell array of character vectors or string vector to use different line styles for each edge.

Character(s)	Line Style	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Edge line width
`0.5` (default) | positive value | vector

Edge line width, specified as a positive value in point units, or as a vector of such values. Specify a vector to use a different line width for each edge in the graph.

Example: `0.75`

`EdgeAlpha` — Transparency of graph edges
`0.5` (default) | scalar value between `0` and `1` inclusive

Transparency of graph edges, specified as a scalar value between `0` and `1` inclusive. A value of `1` means fully opaque and `0` means completely transparent (invisible).

Example: `0.25`

`EdgeCData` — Color data of edge lines
vector

Color data of edge lines, specified as a vector with length equal to the number of edges in the graph. The values in `EdgeCData` map linearly to the colors in the current colormap, resulting in different colors for each edge in the plotted graph.

`ArrowSize` — Arrow size
positive value | vector of positive values

Arrow size, specified as a positive value in point units or as a vector of such values. As a vector, `ArrowSize` specifies the size of the arrow for each edge in the graph. The default

value of `ArrowSize` is `7` for graphs with 100 or fewer nodes, and `4` for graphs with more than 100 nodes.

`ArrowSize` only affects directed graphs.

Example: `15`

`ArrowPosition` — Position of arrow along edge
`0.5` (default) | scalar | vector

Position of arrow along edge, specified as a value in the range `[0 1]` or as a vector of such values with length equal to the number of edges. A value near 0 places arrows closer to the source node, and a value near 1 places arrows closer to the target node. The default value is `0.5` so that the arrows are halfway between the source and target nodes.

`ArrowPosition` only affects directed graphs.

`ShowArrows` — Toggle display of arrows on directed edges
`'on'` | `'off'`

Toggle display of arrows on directed edges, specified as `'on'` or `'off'`. For directed graphs the default value is `'on'` so that arrows are displayed, but you can specify a value of `'off'` to hide the arrows on the directed edges. For undirected graphs `ShowArrows` is always `'off'`.

Position

`XData` — x-coordinate of nodes
vector

Note `XData` and `YData` must be specified together so that each node has a valid (*x*,*y*) coordinate. Optionally, you can specify `ZData` for 3-D coordinates.

---

x-coordinate of nodes, specified as a vector with length equal to the number of nodes in the graph.

`YData` — y-coordinate of nodes
vector

Note `XData` and `YData` must be specified together so that each node has a valid (*x*,*y*) coordinate. Optionally, you can specify `ZData` for 3-D coordinates.

---

y-coordinate of nodes, specified as a vector with length equal to the number of nodes in the graph.

ZData — z-coordinate of nodes
vector

Note `XData` and `YData` must be specified together so that each node has a valid (*x*,*y*) coordinate. Optionally, you can specify `ZData` for 3-D coordinates.

---

z-coordinate of nodes, specified as a vector with length equal to the number of nodes in the graph.

Node and Edge Labels

NodeLabel — Node labels
node IDs (default) | vector | cell array of character vectors

Node labels, specified as a numeric vector or cell array of character vectors. The length of `NodeLabel` must be equal to the number of nodes in the graph. By default `NodeLabel` is a cell array containing the node IDs for the graph nodes:

- For nodes without names (that is, `G.Nodes` does not contain a `Name` variable), the node labels are the values `unique(G.Edges.EndNodes)` contained in a cell array.
- For named nodes, the node labels are `G.Nodes.Name'`.

Example: `{'A', 'B', 'C'}`

Example: `[1 2 3]`

Example: `plot(G,'NodeLabels',G.Nodes.Name)` labels the nodes with their names.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `cell`

NodeLabelMode — Selection mode for node labels
`'auto'` (default) | `'manual'`

Selection mode for node labels, specified as `'auto'` (default) or `'manual'`. Specify `NodeLabelMode` as `'auto'` to populate `NodeLabel` with the node IDs for the graph nodes (numeric node indices or node names). Specifying `NodeLabelMode` as `'manual'` does not change the values in `NodeLabel`.

`NodeLabelColor` — Color of node labels

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | color name | matrix

Node label color, specified as one of these values:

- matrix — Each row is an RGB triplet representing the color of one node label. The size of the matrix is `numnodes(G)`-by-3.
- RGB triplet, hexadecimal color code, or color name — All node labels use the specified color.

  RGB triplets and hexadecimal color codes are useful for specifying custom colors.

  - An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
  - A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

  Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

  Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `plot(G,'NodeLabel',C,'NodeLabelColor','m')` creates a graph plot with magenta node labels.

`EdgeLabel` — Edge labels
`{}` (default) | vector | cell array of character vectors

Edge labels, specified as a numeric vector or cell array of character vectors. The length of `EdgeLabel` must be equal to the number of edges in the graph. By default `EdgeLabel` is an empty cell array (no edge labels are displayed).

Example: `{'A', 'B', 'C'}`

Example: `[1 2 3]`

Example: `plot(G,'EdgeLabels',G.Edges.Weight)` labels the graph edges with their weights.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `cell`

`EdgeLabelMode` — Selection mode for edge labels
`'manual'` (default) | `'auto'`

Selection mode for edge labels, specified as `'manual'` (default) or `'auto'`. Specify `EdgeLabelMode` as `'auto'` to populate `EdgeLabel` with the edge weights in

`G.Edges.Weight` (if available), or the edge indices `G.Edges(k,:)` (if no weights are available). Specifying `EdgeLabelMode` as `'manual'` does not change the values in `EdgeLabel`.

`EdgeLabelColor` — Color of edge labels

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | color name | matrix

Edge label color, specified as one of these values:

- matrix — Each row is an RGB triplet representing the color of one edge label. The size of the matrix is `numedges(G)`-by-3.
- RGB triplet, hexadecimal color code, or color name — All edge labels use the specified color.

  RGB triplets and hexadecimal color codes are useful for specifying custom colors.

  - An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
  - A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

  Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `plot(G,'EdgeLabel',C,'EdgeLabelColor','m')` creates a graph plot with magenta edge labels.

`Interpreter` — Interpretation of text characters
`'tex'` (default) | `'latex'` | `'none'`

Interpretation of text characters, specified as one of these values:

- `'tex'` — Interpret characters using a subset of TeX markup.
- `'latex'` — Interpret characters using LaTeX markup.
- `'none'` — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text.

Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they modify only the next character or the characters within the curly braces. When you set the `Interpreter` property to `'tex'`, the supported modifiers are as follows.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{`*`specifier`*`}`	Font name — Replace *`specifier`* with the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{`*`specifier`*`}`	Font size —Replace *`specifier`* with a numeric scalar value in point units.	`'\fontsize{15} text'`
`\color{`*`specifier`*`}`	Font color — Replace *`specifier`* with one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`

Modifier	Description	Example
`\color[rgb] {specifier}`	Custom font color — Replace *specifier* with a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	β	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ϵ	`\Delta`	Δ	`\spadesuit`	♠
`\zeta`	ζ	`\Theta`	Θ	`\leftrightarrow`	↔
`\eta`	η	`\Lambda`	Λ	`\leftarrow`	←
`\theta`	θ	`\Xi`	Ξ	`\Leftarrow`	⇐
`\vartheta`	ϑ	`\Pi`	Π	`\uparrow`	↑
`\iota`	ι	`\Sigma`	Σ	`\rightarrow`	→
`\kappa`	κ	`\Upsilon`	ϒ	`\Rightarrow`	⇒
`\lambda`	λ	`\Phi`	Φ	`\downarrow`	↓
`\mu`	µ	`\Psi`	Ψ	`\circ`	º
`\nu`	ν	`\Omega`	Ω	`\pm`	±
`\xi`	ξ	`\forall`	∀	`\geq`	≥
`\pi`	π	`\exists`	∃	`\propto`	∝

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. Use dollar symbols around the text, for example, use `'$\int_1^{20} x^2 dx$'` for inline mode or `'$$ \int_1^{20} x^2 dx$$'` for display mode.

The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup.

The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters.

For more information about the LaTeX system, see The LaTeX Project website at https://www.latex-project.org/.

Font

NodeFontName — Font name for node labels
'Helvetica' (default) | supported font name | 'FixedWidth'

Font name for node labels, specified as a supported font name or 'FixedWidth'. For labels to display and print properly, you must choose a font that your system supports. The default font depends on the specific operating system and locale. For example, Windows and Linuxsystems in English localization use the Helvetica font by default.

To use a fixed-width font that looks good in any locale, specify 'FixedWidth'.

Example: 'Cambria'

NodeFontSize — Font size for node labels
8 (default) | positive number | vector of positive numbers

Font size for node labels, specified as a positive number or a vector of positive numbers. If NodeFontSize is a vector, then each element specifies the font size of one node label.

NodeFontWeight — Thickness of text in node labels
'normal' (default) | 'bold' | vector | cell array

Thickness of text in node labels, specified as 'normal', 'bold', or as a string vector or cell array of character vectors specifying 'normal' or 'bold' for each node.

- 'bold' — **Thicker character outlines than normal**
- 'normal' — Normal weight as defined by the particular font

Not all fonts have a bold font weight.

Data Types: cell | char | string

NodeFontAngle — Character slant of text in node labels
'normal' (default) | 'italic' | vector | cell array

Character slant of text in node labels, specified as 'normal', 'italic', or as a string vector or cell array of character vectors specifying 'normal' or 'italic' for each node.

- 'italic' — *Slanted characters*
- 'normal' — No character slant

Not all fonts have both font styles.

Data Types: `cell` | `char` | `string`

`EdgeFontName` — Font name for edge labels
`'Helvetica'` (default) | supported font name | `'FixedWidth'`

Font name for edge labels, specified as a supported font name or `'FixedWidth'`. For labels to display and print properly, you must choose a font that your system supports. The default font depends on the specific operating system and locale. For example, Windows and Linuxsystems in English localization use the Helvetica font by default.

To use a fixed-width font that looks good in any locale, specify `'FixedWidth'`.

Example: `'Cambria'`

`EdgeFontSize` — Font size for edge labels
8 (default) | positive number | vector of positive numbers

Font size for edge labels, specified as a positive number or a vector of positive numbers. If `EdgeFontSize` is a vector, then each element specifies the font size of one edge label.

`EdgeFontWeight` — Thickness of text in edge labels
`'normal'` (default) | `'bold'` | vector | cell array

Thickness of text in edge labels, specified as `'normal'`, `'bold'`, or as a string vector or cell array of character vectors specifying `'normal'` or `'bold'` for each edge.

- `'bold'` — **Thicker character outlines than normal**
- `'normal'` — Normal weight as defined by the particular font

Not all fonts have a bold font weight.

Data Types: `cell` | `char` | `string`

`EdgeFontAngle` — Character slant of text in edge labels
`'normal'` (default) | `'italic'` | vector | cell array

Character slant of text in edge labels, specified as `'normal'`, `'italic'`, or as a string vector or cell array of character vectors specifying `'normal'` or `'italic'` for each edge.

- `'italic'` — *Slanted characters*
- `'normal'` — No character slant

Not all fonts have both font styles.

Data Types: `cell` | `char` | `string`

Legend

`DisplayName` — Text used by legend

`''` (default) | character vector

Text used by the legend, specified as a character vector. The text appears next to an icon of the GraphPlot.

Example: `'Text Description'`

For multiline text, create the character vector using `sprintf` with the new line character `\n`.

Example: `sprintf('line one\nline two')`

Alternatively, you can specify the legend text using the `legend` function.

- If you specify the text as an input argument to the `legend` function, then the legend uses the specified text and sets the `DisplayName` property to the same value.
- If you do not specify the text as an input argument to the `legend` function, then the legend uses the text in the `DisplayName` property. If the `DisplayName` property does not contain any text, then the legend generates a character vector. The character vector has the form `'dataN'`, where `N` is the number assigned to the GraphPlot object based on its location in the list of legend entries.

If you edit interactively the character vector in an existing legend, then MATLAB updates the `DisplayName` property to the edited character vector.

`Annotation` — Legend icon display style

`Annotation` object

This property is read-only.

Legend icon display style, returned as an `Annotation` object. Use this object to include or exclude the GraphPlot from a legend.

1. Query the `Annotation` property to get the `Annotation` object.
2. Query the `LegendInformation` property of the `Annotation` object to get the `LegendEntry` object.

3. Specify the `IconDisplayStyle` property of the `LegendEntry` object to one of these values:
   - `'on'` — Include the GraphPlot object in the legend as one entry (default).
   - `'off'` — Do not include the GraphPlot object in the legend.
   - `'children'` — Include only children of the GraphPlot object as separate entries in the legend.

If a legend already exists and you change the `IconDisplayStyle` setting, then you must call `legend` to update the display.

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the GraphPlot.
- `'off'` — Hide the GraphPlot without deleting it. You still can access the properties of an invisible GraphPlot object.

`DataTipTemplate` — Data tip content

`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

`UIContextMenu` — Context menu

uicontextmenu object

Context menu, specified as a uicontextmenu object. Use this property to display a context menu when you right-click the GraphPlot. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the GraphPlot when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the GraphPlot.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the GraphPlot. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The GraphPlot object — You can access properties of the GraphPlot object from within the callback function.

- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`CreateFcn` — Creation callback

`''` (default) | function handle | cell array | character vector

Creation callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you create the GraphPlot. Setting the `CreateFcn` property on an existing GraphPlot has no effect. You must define a default value for this property, or define this property using a `Name,Value` pair during GraphPlot creation. MATLAB executes the callback after creating the GraphPlot and setting all of its properties.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The GraphPlot object — You can access properties of the GraphPlot object from within the callback function. You also can access the GraphPlot object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

`DeleteFcn` — Deletion callback

`''` (default) | function handle | cell array | character vector

Deletion callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you delete the GraphPlot. MATLAB executes the callback before destroying the GraphPlot so that the callback can access its property values.

If you specify this callback using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- The GraphPlot object — You can access properties of the GraphPlot object from within the callback function. You also can access the GraphPlot object through the `CallbackObject` property of the root, which can be queried using the `gcbo` function.
- Event data — This argument is empty for this property. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Example: `@myCallback`

Example: `{@myCallback,arg3}`

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

Note There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the GraphPlot is the running callback, then the `Interruptible` property determines if it another callback can interrupt it:

- `'on'` — Interruptible. Interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes. For more information, see “Interrupt Callback Execution”.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Not interruptible. MATLAB finishes executing the running callback without any interruptions.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks.

Note There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is allowed. If interruption is not allowed, then the `BusyAction`

property of the object owning the interrupting callback determines if it is discarded or put in the queue.

If the `ButtonDownFcn` callback of the GraphPlot tries to interrupt a running callback that cannot be interrupted, then the `BusyAction` property determines if it is discarded or put in the queue. Specify the `BusyAction` property as one of these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution. This is the default behavior.
- `'cancel'` — Discard the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Can capture mouse clicks only when visible. The `Visible` property must be set to `'on'`. The `HitTest` property determines if the GraphPlot responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the GraphPlot passes the click to the object below it in the current view of the figure window. The `HitTest` property of the GraphPlot has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the GraphPlot. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the GraphPlot that has a `HitTest` property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

Note The `PickableParts` property determines if the GraphPlot object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status of GraphPlot
`'off'` (default) | `'on'`

This property is read-only.

Deletion status of GraphPlot, returned as `'on'` or `'off'`. MATLAB sets the `BeingDeleted` property to `'on'` when the delete function of the GraphPlot begins execution (see the `DeleteFcn` property). The `BeingDeleted` property remains set to `'on'` until the GraphPlot no longer exists.

Check the value of the `BeingDeleted` property to verify that the GraphPlot is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent of GraphPlot

axes object | group object | transform object

Parent of GraphPlot, specified as an axes, group, or transform object.

`Children` — Children

empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the chart object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of GraphPlot object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — The GraphPlot object handle is always visible.
- `'off'` — The GraphPlot object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — The GraphPlot object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the GraphPlot at the command-line, but allows callback functions to access it.

If the GraphPlot object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'graphplot'`

This property is read-only.

Type of graphics object, returned as `'graphplot'`. Use this property to find all objects of a given type within a plotting hierarchy, such as searching for the type using `findobj`.

Tag — Tag to associate with GraphPlot

`''` (default) | character vector

Tag to associate with the GraphPlot, specified as a character vector. Tags provide a way to identify graphics objects. Use this property to find all objects with a specific tag within a plotting hierarchy, for example, searching for the tag using `findobj`.

Example: `'January Data'`

Data Types: `char`

UserData — Data to associate with GraphPlot

`[]` (default) | scalar, vector, or matrix | cell array | character array | table | structure

Data to associate with the GraphPlot object, specified as a scalar, vector, matrix, cell array, character array, table, or structure. MATLAB does not use this data.

To associate multiple sets of data or to attach a field name to the data, use the `getappdata` and `setappdata` functions.

Example: `1:100`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell`

See Also

`digraph` | `graph` | `plot`

Topics

"Access Property Values"
`GraphPlot`

Introduced in R2015b

gray

Gray colormap array

Syntax

```
c = gray
c = gray(m)
```

Description

`c = gray` returns the gray colormap as a three-column array with the same number of rows as the colormap for the current figure. If no figure exists, then the number of rows is equal to the default length of 256. Each row in the array contains the red, green, and blue intensities for a specific color. The intensities are in the range [0,1], and the color scheme looks like this image.

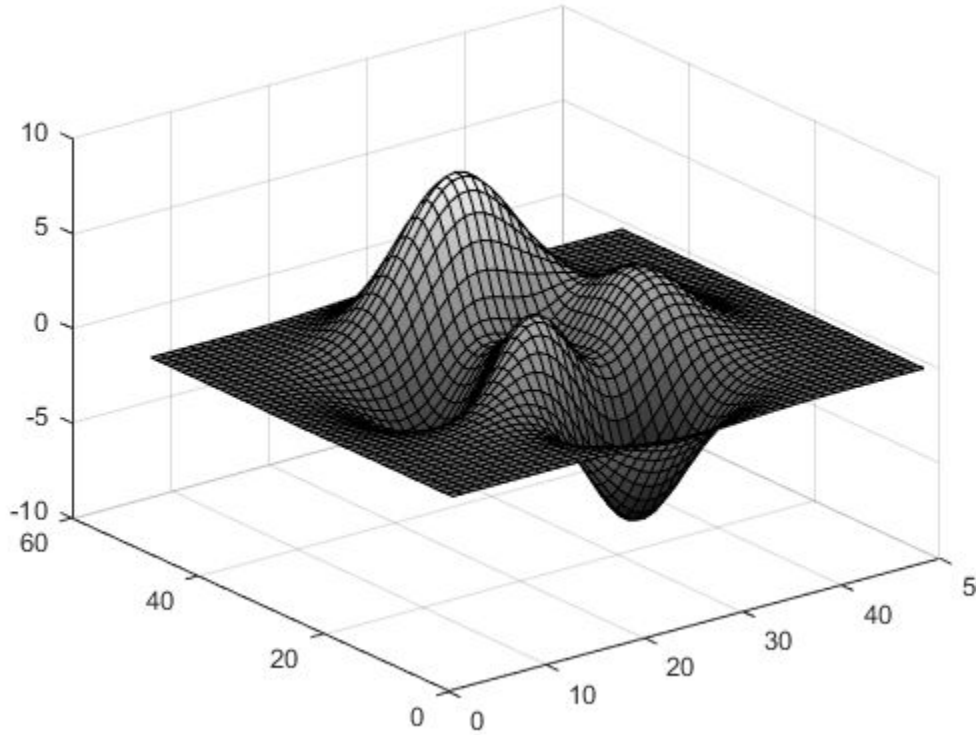
`c = gray(m)` returns the colormap with `m` colors.

Examples

Reverse the Colormap

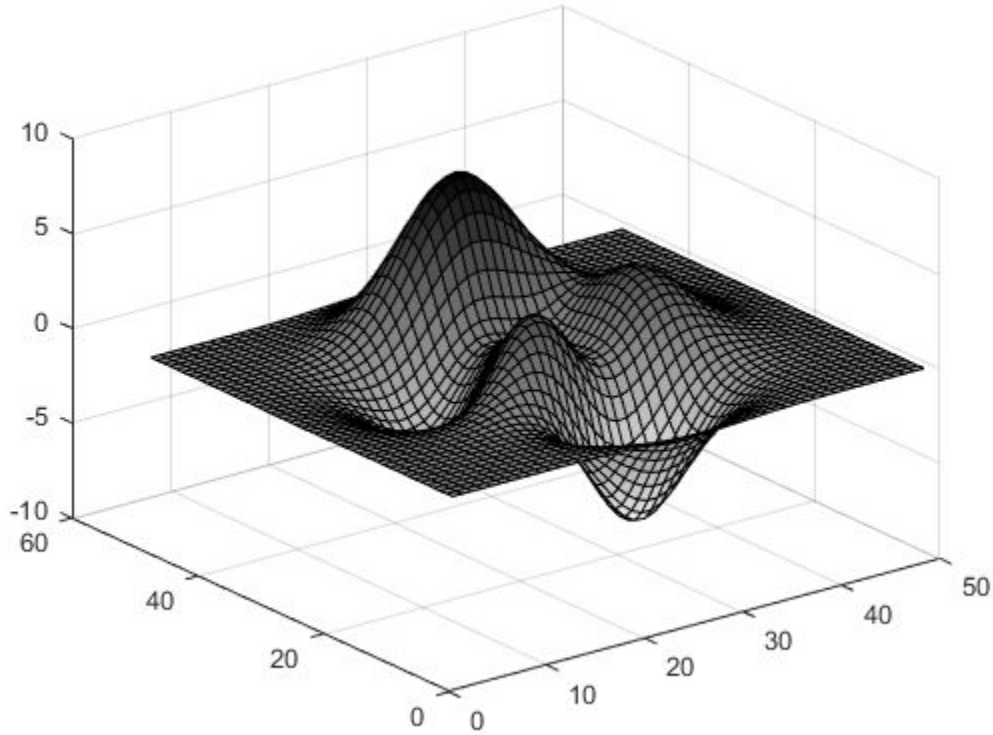
Plot a surface and assign the gray colormap.

```
surf(peaks);
colormap('gray');
```



Get the gray colormap array and reverse the order. Then apply the modified colormap to the surface.

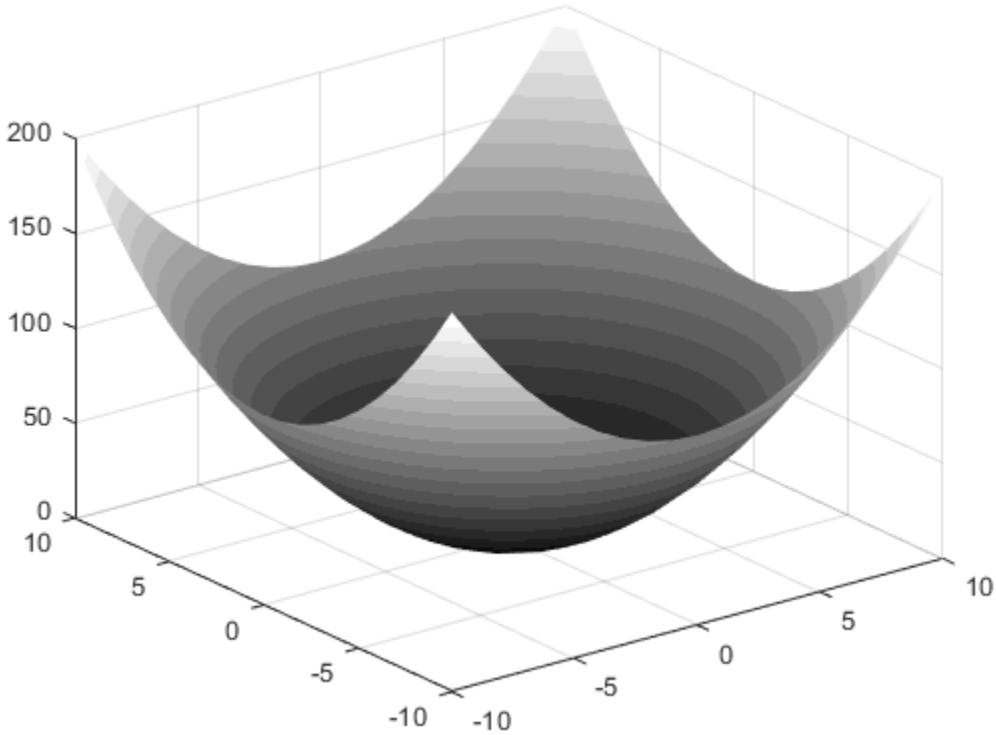
```
c = gray;
c = flipud(c);
colormap(c);
```



Downsample the Gray Colormap

Get a downsampled version of the gray colormap containing only twenty colors. Then display the contours of a paraboloid by applying the colormap and interpolated shading.

```
c = gray(20);
[X,Y] = meshgrid(-10:1:10);
Z = X.^2 + Y.^2;
surf(X,Y,Z);
colormap(c);
shading interp;
```



Input Arguments

`m` — Number of colors

256 (default) | scalar integer value

Number of colors, specified as a scalar integer value. The default value of `m` is equal to the length of the colormap for the current figure. If no figure exists, the default value is 256.

Data Types: `single` | `double`

Compatibility Considerations

Colormaps Have 256 Colors by Default

Behavior changed in R2019b

Starting in R2019b, colormaps have 256 colors by default.

In R2019a and previous releases, the default size is 64. If you have code that depends on a colormap having 64 colors, specify the number of colors when you set the colormap for the figure, axes, or chart. For example, `colormap(gray(64))` sets the figure's colormap to the 64-color `gray` colormap.

Alternatively, you can change the default colormap for all figures within your MATLAB session:

```
set(groot,'defaultFigureColormap',gray(64))
```

See Also

`colormap`

Topics

"Change Color Scheme Using a Colormap"

Introduced before R2006a

matlab.graphics.Graphics class

Package: `matlab.graphics`

Common base class for graphics objects

Description

The `matlab.graphics.Graphics` class is the base class of all graphics objects. Because graphics objects are part of a heterogeneous hierarchy, you can create arrays of mixed classes (for example, an array can contain lines, surfaces, axes, and other graphics objects).

The class of an array of mixed objects is `matlab.graphics.Graphics` because this class is common to all graphics object.

Attributes

`Abstract`	`true`
`HandleCompatible`	`true`

For information on class attributes, see “Class Attributes”.

See Also

`matlab.graphics.GraphicsPlaceholder`

Topics

Class Attributes
Property Attributes

matlab.graphics.GraphicsPlaceholder class

Package: `matlab.graphics`
Superclasses: `matlab.graphics.Graphics`

Default graphics object

Description

The `matlab.graphics.GraphicsPlaceholder` class defines the default graphics object. Instances of this class appear as:

- Elements of pre-allocated arrays created with `gobjects`.
- Unassigned array element placeholders
- Graphics object properties that hold object handles, but are set to empty values
- Empty values returned by functions that return object handles (for example, `findobj`).

Attributes

`Sealed`	`true`
`ConstructOnLoad`	`true`
`HandleCompatible`	`true`

For information on class attributes, see “Class Attributes”.

Examples

Test for Current Figure

```
if isempty(get(groot,'CurrentFigure')
   ... % There is no current figure
end
```

See Also

Topics

"Graphics Object Programming"
Class Attributes
Property Attributes

graymon

Set default figure properties for grayscale monitors

Note `graymon` is not recommended.

Syntax

```
graymon
```

Description

`graymon` sets defaults for graphics properties to produce more legible displays for grayscale monitors.

See Also

`axes` | `figure`

Introduced before R2006a

grid

Display or hide axes grid lines

Syntax

```
grid on
grid off
grid
grid minor
grid(target, ___ )
```

Description

`grid on` displays the major grid lines for the current axes or chart returned by the `gca` command. Major grid lines extend from each tick mark.

`grid off` removes all grid lines from the current axes or chart.

`grid` toggles the visibility of the major grid lines.

`grid minor` toggles the visibility of the minor grid lines. Minor grid lines lie between the tick marks. Not all types of charts support minor grid lines.

`grid(target, ___ )` uses the axes or chart specified by `target` instead of the current axes or chart. Specify `target` as the first input argument. Use single quotes around other input arguments, for example, `grid(target,'on')`.

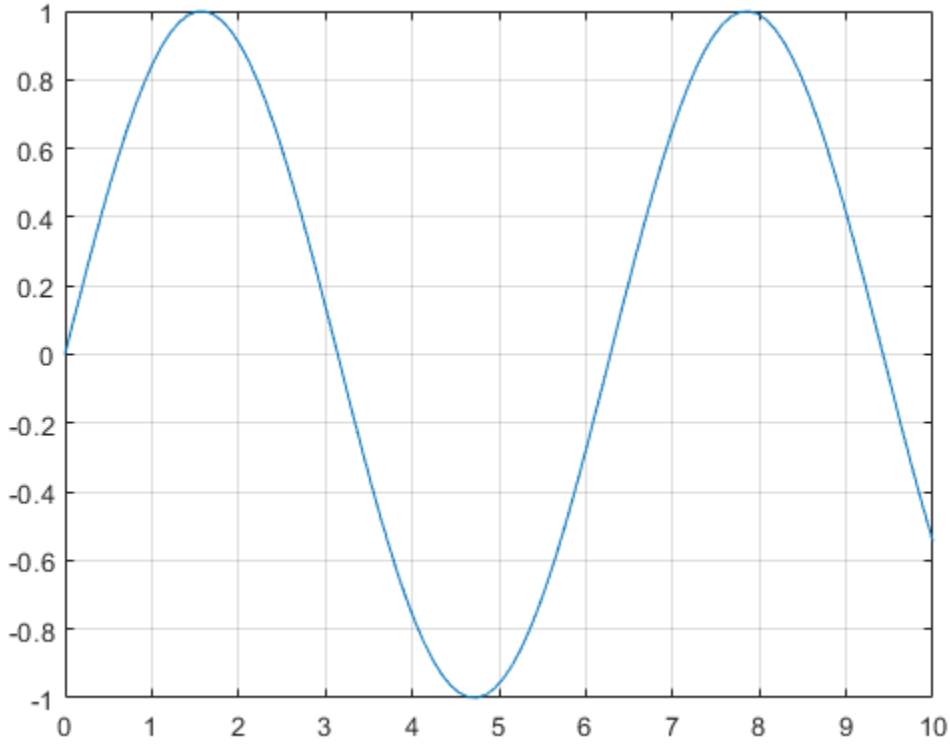
Examples

Display Grid Lines

Display the grid lines for a sine plot.

```
x = linspace(0,10);
y = sin(x);
```

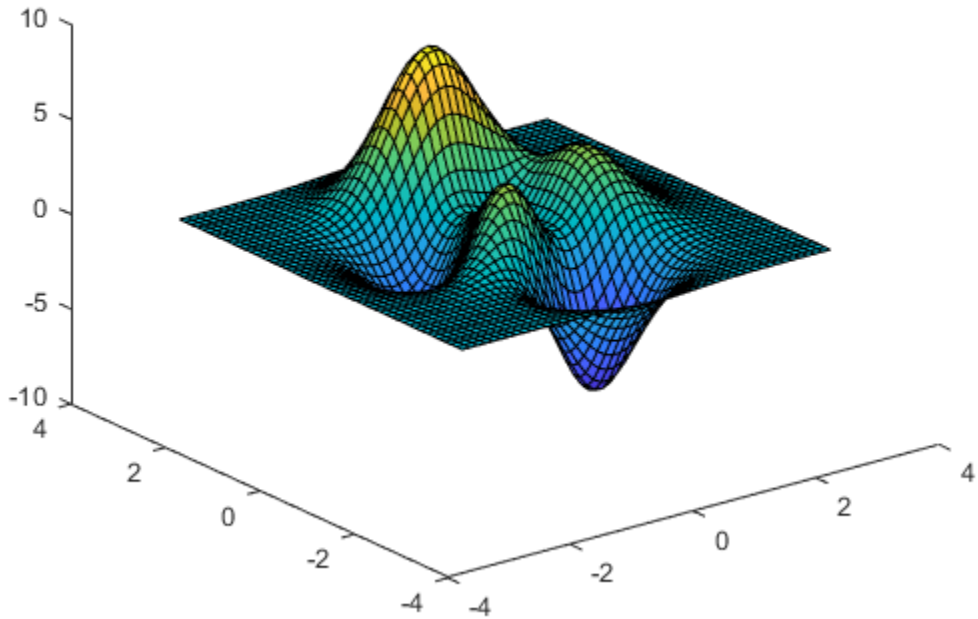
```
plot(x,y)
grid on
```



Remove Grid Lines

Create a surface plot and remove the grid lines.

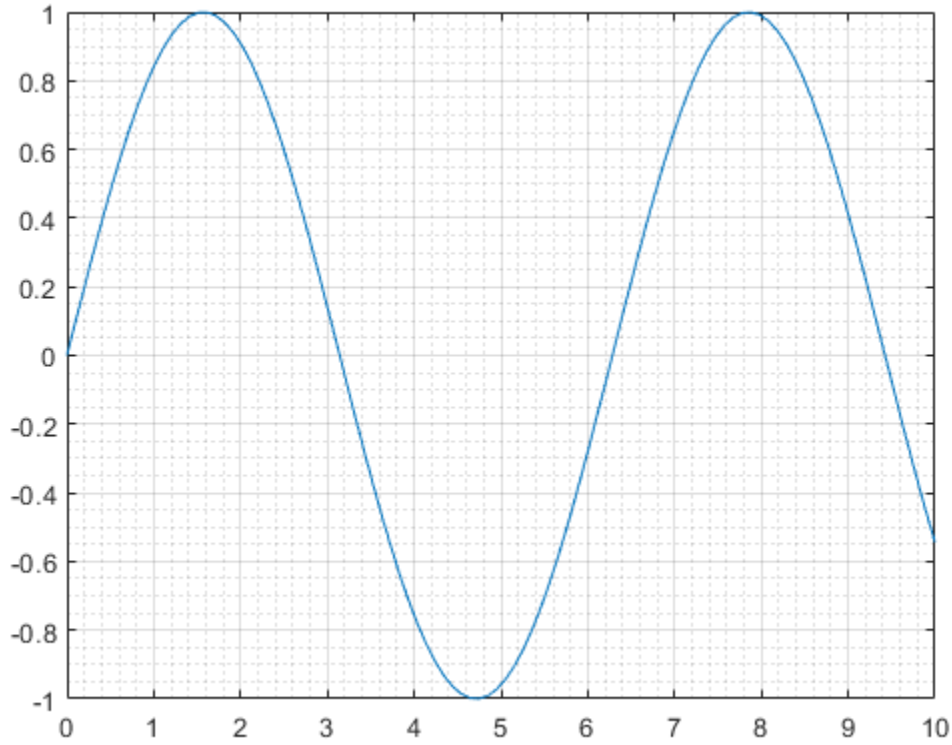
```
[X,Y,Z] = peaks;
surf(X,Y,Z)
grid off
```



Display Major and Minor Grid Lines

Display the major and minor grid lines for a sine plot.

```
x = linspace(0,10);
y = sin(x);
plot(x,y)
grid on
grid minor
```

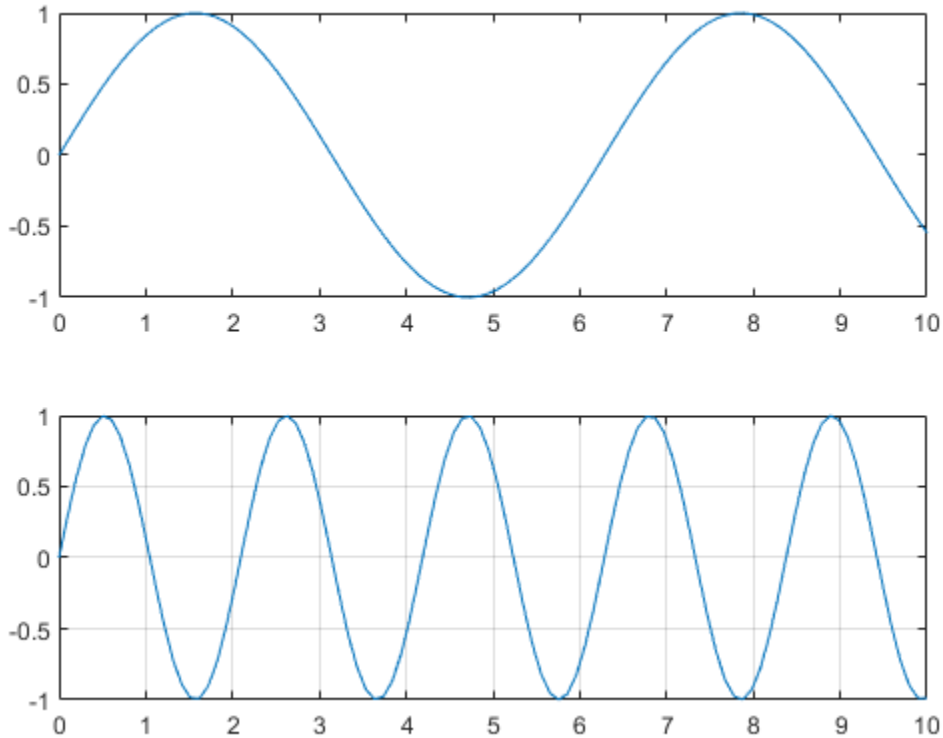


Display Grid Lines on Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Plot data into each axes. Then display grid lines in the bottom plot by passing `ax2` to the `grid` function.

```
x = linspace(0,10);
y1 = sin(x);
y2 = sin(3*x);
tiledlayout(2,1)
```

```
% Top plot
ax1 = nexttile;
plot(ax1,x,y1)

% Bottom plot
ax2 = nexttile;
plot(ax2,x,y2)
grid(ax2,'on')
```



Input Arguments

target — Target axes or chart

Axes object | PolarAxes object | GeographicAxes object | graphics object

Target axes or chart, specified as an Axes object, a PolarAxes object, a GeographicAxes object, or a graphics object that has a GridVisible property. For example, you can add or remove grid lines from a HeatmapChart object using the grid function.

If you do not specify the target, then the grid function affects the graphics object returned by the gca command.

Tips

- When working with Cartesian axes, some axes properties affect the appearance of the grid lines. This table lists a subset of axes properties related to the grid lines.

Axes Property	Description
XTick, YTick, ZTick	Location of tick marks and major grid lines for each axis direction
XGrid, YGrid, ZGrid	Display of major grid lines for each axis direction
XMinorGrid, YMinorGrid, ZMinorGrid	Display of minor grid lines for each axis direction
LineWidth	Line width of grid lines, axes box outline, and tick marks
GridLineStyle	Major grid line style
MinorGridLineStyle	Minor grid line style
GridColor	Major grid line color
MinorGridColor	Minor grid line color
GridAlpha	Major grid line transparency
MinorGridAlpha	Minor grid line transparency

Axes Property	Description
`Layer`	Location of grid lines in relation to the plotted data

- When working with polar axes, some polar axes properties affect the appearance of the grid lines. This table lists a subset of polar axes properties related to the grid lines.

PolarAxes Property	Description
`ThetaTick`, `RTick`	Location of tick marks and major grid lines for each axis direction
`ThetaGrid`, `RGrid`	Display of major grid lines for each axis direction
`ThetaMinorGrid`, `RMinorGrid`	Display of minor grid lines for each axis direction
`LineWidth`	Width of outline, tick marks, and grid lines
`GridLineStyle`	Major grid line style
`MinorGridLineStyle`	Minor grid line style
`GridColor`	Major grid line color
`MinorGridColor`	Minor grid line color
`GridAlpha`	Major grid line transparency
`MinorGridAlpha`	Minor grid line transparency
`Layer`	Location of grid lines in relation to the plotted data

- When working with geographic axes, some geographic axes properties affect the appearance of the grid lines. This table lists a subset of geographic axes properties related to the grid lines. Note that `GeographicAxes` objects do not support minor grid lines.

GeographicAxes Property	Description
`Grid`	Display of latitude and longitude grid lines
`LineWidth`	Line width of grid lines, box outline, and tick marks

GeographicAxes Property	Description
`GridLineStyle`	Grid line style
`GridColor`	Color of grid lines
`GridAlpha`	Grid line transparency

Algorithms

The `grid` function sets these graphics object properties to either `'on'` or `'off'`:

- `XGrid`, `YGrid`, and `ZGrid` when working with Cartesian `Axes` objects.
- `ThetaGrid` and `RGrid` when working with `PolarAxes` objects.
- `Grid` when working with `GeographicAxes` objects.
- `GridVisible` when working with other types of graphics objects, such as a `HeatmapChart` object.

See Also

Functions
`axis` | `box` | `legend` | `title` | `xlabel` | `ylabel`

Properties
Axes | GeographicAxes | PolarAxes

Topics

"Add Grid Lines and Edit Placement"

Introduced before R2006a

griddata

Interpolate 2-D or 3-D scattered data

Syntax

```
vq = griddata(x,y,v,xq,yq)
vq = griddata(x,y,z,v,xq,yq,zq)
vq = griddata( ___ ,method)
```

Description

`vq = griddata(x,y,v,xq,yq)` fits a surface of the form $v = f(x,y)$ to the scattered data in the vectors `(x,y,v)`. The `griddata` function interpolates the surface at the query points specified by `(xq,yq)` and returns the interpolated values, `vq`. The surface always passes through the data points defined by `x` and `y`.

`vq = griddata(x,y,z,v,xq,yq,zq)` fits a hypersurface of the form $v = f(x,y,z)$.

`vq = griddata( ___ ,method)` specifies the interpolation method used to compute `vq` using any of the input arguments in the previous syntaxes. `method` can be `'linear'`, `'nearest'`, `'natural'`, `'cubic'`, or `'v4'`. The default method is `'linear'`.

Note Qhull-specific options are no longer supported. Remove the `options` argument from all instances in your code that pass it to `griddata`.

In a future release, `griddata` will not accept any input vectors of mixed orientation. In addition, the following syntaxes will be removed:

```
[Xq,Yq,Vq] = griddata(x,y,v,xq,yq)
[Xq,Yq,Vq] = griddata(x,y,v,xq,yq, method)
```

To specify a grid of query points, construct a full grid with `ndgrid` or `meshgrid` before calling `griddata`.

Examples

Interpolate Scattered Data Over Uniform Grid

Interpolate randomly scattered data on a uniform grid of query points.

Sample a function at 200 random points between `-2.5` and `2.5`.

```
xy = -2.5 + 5*gallery('uniformdata',[200 2],0);
x = xy(:,1);
y = xy(:,2);
v = x.*exp(-x.^2-y.^2);
```

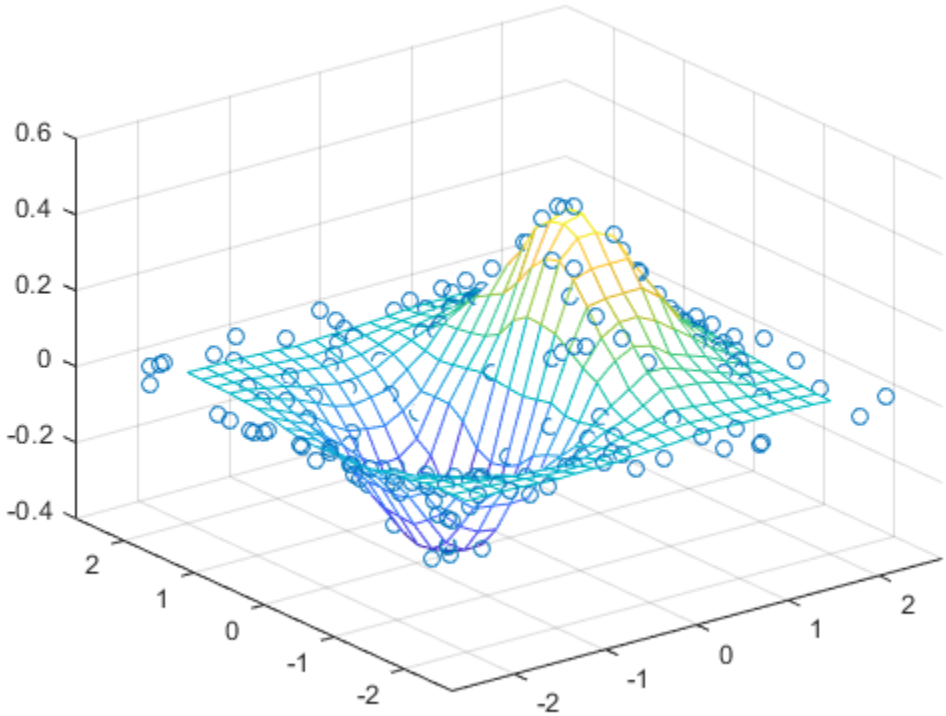
`x`, `y`, and `v` are vectors containing scattered (nonuniform) sample points and data.

Define a regular grid and interpolate the scattered data over the grid.

```
[xq,yq] = meshgrid(-2:.2:2, -2:.2:2);
vq = griddata(x,y,v,xq,yq);
```

Plot the gridded data as a mesh and the scattered data as dots.

```
mesh(xq,yq,vq)
hold on
plot3(x,y,v,'o')
xlim([-2.7 2.7])
ylim([-2.7 2.7])
```



Interpolate 4-D Data Set Over Grid

Interpolate a 3-D slice of a 4-D function that is sampled at randomly scattered points.

Sample a 4-D function $v(x, y, z)$ at 2500 random points between `-1` and `1`. The vectors `x`, `y`, and `z` contain the nonuniform sample points.

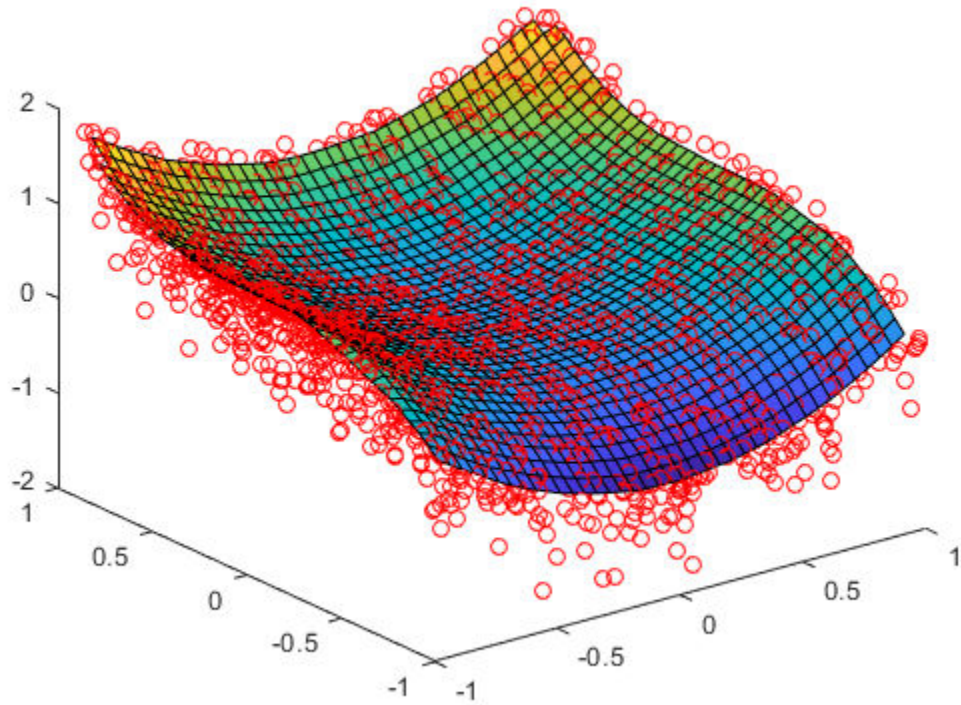
```
x = 2*rand(2500,1) - 1;
y = 2*rand(2500,1) - 1;
z = 2*rand(2500,1) - 1;
v = x.^2 + y.^3 - z.^4;
```

Define a regular grid with *xy* points in the range [-1, 1], and set $z = 0$. Interpolating on this grid of 2-D query points `(xq,yq,0)` produces a 3-D interpolated slice `(xq,yq,0,vq)` of the 4-D data set `(x,y,z,v)`.

```
d = -1:0.05:1;
[xq,yq,zq] = meshgrid(d,d,0);
```

Interpolate the scattered data on the grid. Plot the results.

```
vq = griddata(x,y,z,v,xq,yq,zq);
plot3(x,y,v,'ro')
hold on
surf(xq,yq,vq)
```



Comparison of Scattered Data Interpolation Methods

Compare the results of several different interpolation algorithms offered by `griddata`.

Create a sample data set of 50 scattered points. The number of points is artificially small to highlight the differences between the interpolation methods.

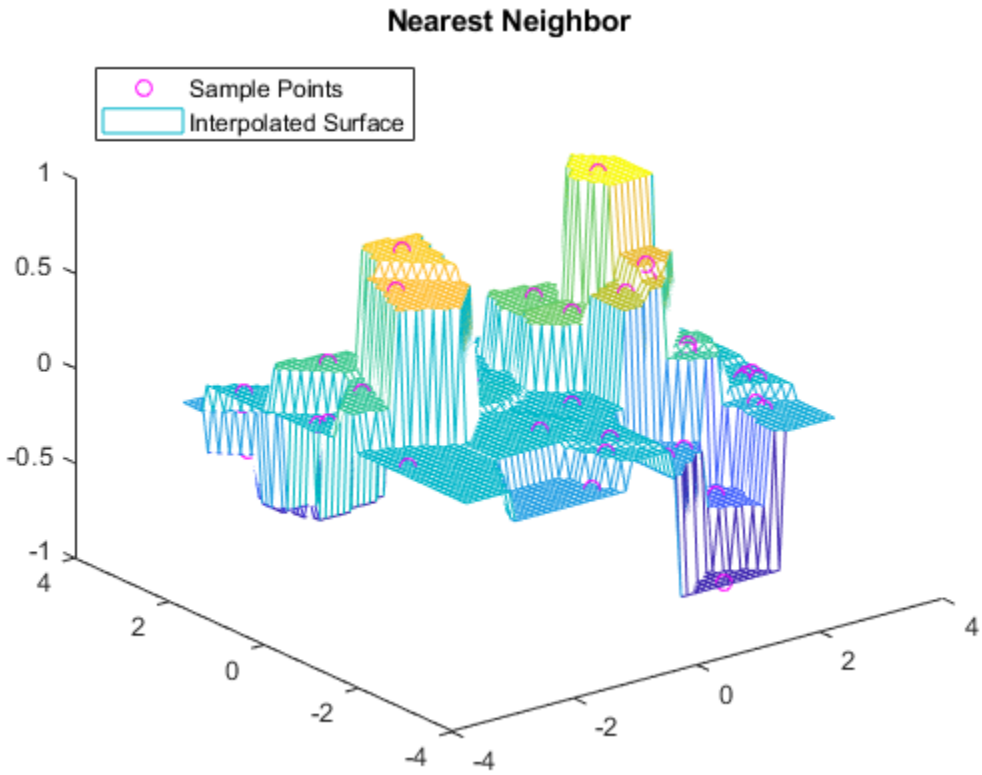
```
x = -3 + 6*rand(50,1);
y = -3 + 6*rand(50,1);
v = sin(x).^4 .* cos(y);
```

Create a grid of query points.

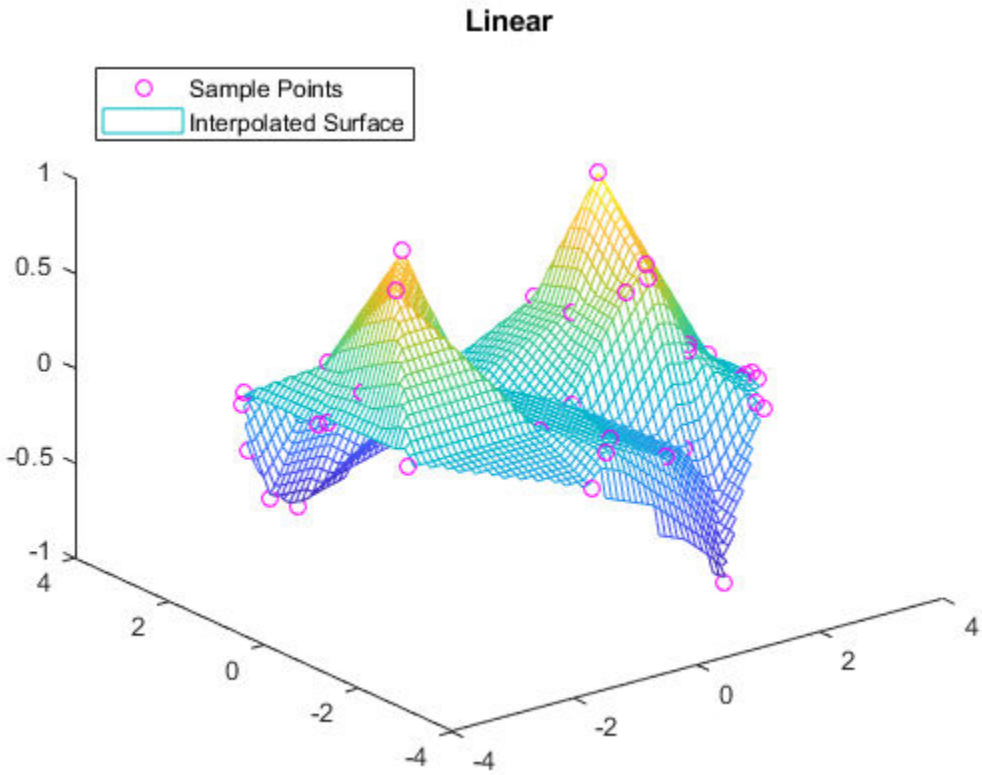
```
[xq,yq] = meshgrid(-3:0.1:3);
```

Interpolate the sample data using the `'nearest'`, `'linear'`, `'natural'`, and `'cubic'` methods. Plot the results for comparison.

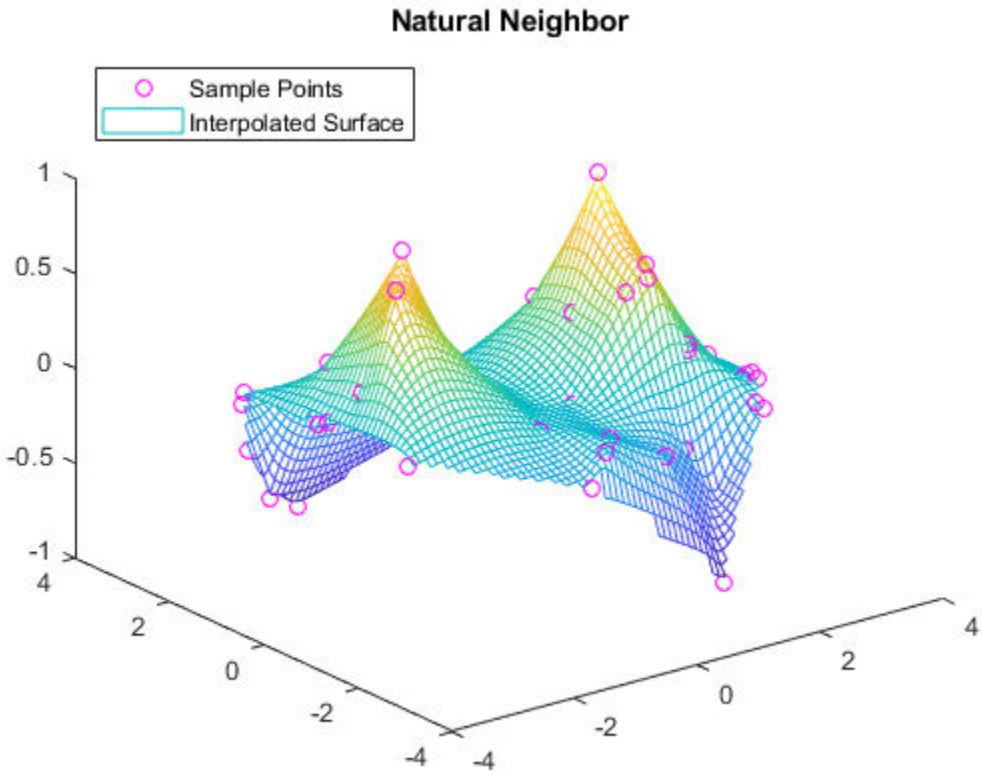
```
z1 = griddata(x,y,v,xq,yq,'nearest');
plot3(x,y,v,'mo')
hold on
mesh(xq,yq,z1)
title('Nearest Neighbor')
legend('Sample Points','Interpolated Surface','Location','NorthWest')
```



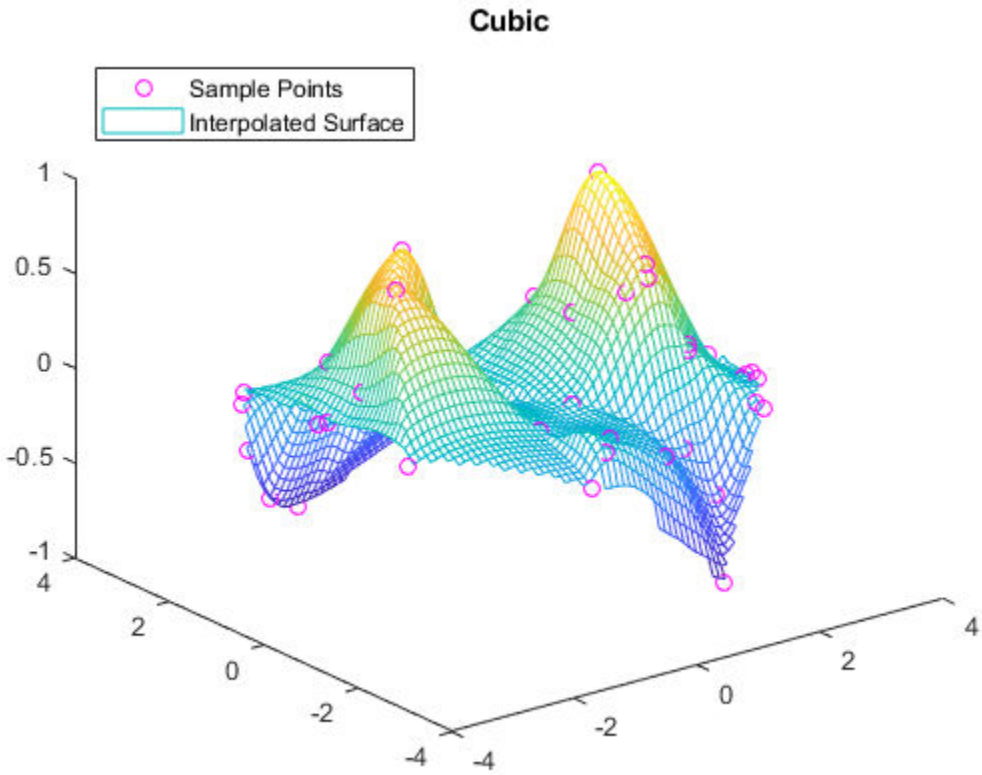

```
z2 = griddata(x,y,v,xq,yq,'linear');
figure
plot3(x,y,v,'mo')
hold on
mesh(xq,yq,z2)
title('Linear')
legend('Sample Points','Interpolated Surface','Location','NorthWest')
```




```
z3 = griddata(x,y,v,xq,yq,'natural');
figure
plot3(x,y,v,'mo')
hold on
mesh(xq,yq,z3)
title('Natural Neighbor')
legend('Sample Points','Interpolated Surface','Location','NorthWest')
```

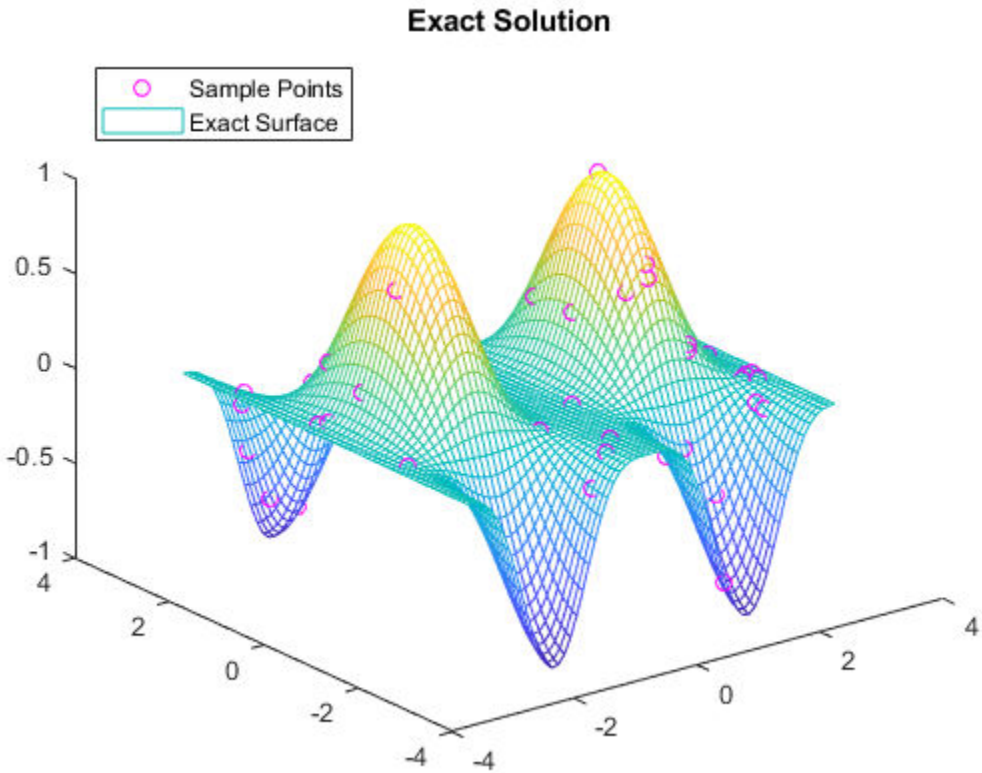



```
z4 = griddata(x,y,v,xq,yq,'cubic');
figure
plot3(x,y,v,'mo')
hold on
mesh(xq,yq,z4)
title('Cubic')
legend('Sample Points','Interpolated Surface','Location','NorthWest')
```



Plot the exact solution.

```
figure
plot3(x,y,v,'mo')
hold on
mesh(xq,yq,sin(xq).^4 .* cos(yq))
title('Exact Solution')
legend('Sample Points','Exact Surface','Location','NorthWest')
```



Input Arguments

x, y, z — Sample point coordinates
vectors

Sample point coordinates, specified as vectors. Corresponding elements in x, y, and z specify the *xyz* coordinates of points where the sample values v are known. The sample points must be unique.

Data Types: `single` | `double`

v — Sample values
vector

Sample values, specified as a vector. The sample values in `v` correspond to the sample points in `x`, `y`, and `z`.

If `v` contains complex numbers, then `griddata` interpolates the real and imaginary parts separately.

Data Types: `single` | `double`
Complex Number Support: Yes

xq, yq, zq — Query points
vector | array

Query points, specified as vectors or arrays. Corresponding elements in the vectors or arrays specify the *xyz* coordinates of the query points. The query points are the locations where `griddata` performs interpolation.

- Specify arrays if you want to pass a grid of query points. Use `ndgrid` or `meshgrid` to construct the arrays.
- Specify vectors if you want to pass a collection of scattered points.

The specified query points must lie inside the convex hull of the sample data points. `griddata` returns `NaN` for query points outside of the convex hull.

Data Types: `single` | `double`

method — Interpolation method
`'linear'` (default) | `'nearest'` | `'natural'` | `'cubic'` | `'v4'`

Interpolation method, specified as one of the methods in this table.

Method	Description	Continuity
`'linear'`	Triangulation-based linear interpolation (default) supporting 2-D and 3-D interpolation.	C^0
`'nearest'`	Triangulation-based nearest neighbor interpolation supporting 2-D and 3-D interpolation.	Discontinuous

Method	Description	Continuity
`'natural'`	Triangulation-based natural neighbor interpolation supporting 2-D and 3-D interpolation. This method is an efficient tradeoff between linear and cubic.	C^1 except at sample points
`'cubic'`	Triangulation-based cubic interpolation supporting 2-D interpolation only.	C^2
`'v4'`	Biharmonic spline interpolation (MATLAB 4 `griddata` method) supporting 2-D interpolation only. Unlike the other methods, this interpolation is not based on a triangulation.	C^2

Data Types: `char`

Output Arguments

`vq` — Interpolated values
vector | array

Interpolated values, returned as a vector or array. The size of `vq` depends on the size of the query point inputs `xq`, `yq`, and `zq`:

- For 2-D interpolation, where `xq` and `yq` specify an `m`-by-`n` grid of query points, `vq` is an `m`-by-`n` array.
- For 3-D interpolation, where `xq`, `yq`, and `zq` specify an `m`-by-`n`-by-`p` grid of query points, `vq` is an `m`-by-`n`-by-`p` array.
- If `xq`, `yq`, (and `zq` for 3-D interpolation) are vectors that specify scattered points, then `vq` is a vector of the same length.

For all interpolation methods other than `'v4'`, the output `vq` contains `NaN` values for query points outside the convex hull of the sample data. The `'v4'` method performs the same calculation for all points regardless of location.

Tips

- Scattered data interpolation with `griddata` uses a Delaunay triangulation of the data, so can be sensitive to scaling issues in `x`, `y`, and `z`. When this occurs, you can use `normalize` to rescale the data and improve the results. See “Normalize Data with Differing Magnitudes” for more information.

See Also

`delaunay` | `griddatan` | `interpn` | `meshgrid` | `ndgrid` | `scatteredInterpolant`

Introduced before R2006a

griddatan

Interpolate N-D scattered data

Syntax

```
vq = griddatan(x,v,xq)
vq = griddatan(x,v,xq,method)
vq = griddatan(x,v,xq,method,options)
```

Description

`vq = griddatan(x,v,xq)` fits a hypersurface of the form $v = f(x)$ to the sample points `x` with values `v`. The `griddatan` function interpolates the surface at the query points specified by `xq` and returns the interpolated values, `vq`. The surface always passes through the data points defined by `x` and `v`.

`vq = griddatan(x,v,xq,method)` specifies the interpolation method used to compute `vq`. Options are `'linear'` or `'nearest'`.

`vq = griddatan(x,v,xq,method,options)` specifies a cell array of character vectors, `options`, to be used in Qhull via `delaunayn`.

Examples

Fit a Hypersurface

Interpolate a 4-D scattered data set and visualize a 3-D isosurface of the interpolated data.

Create a scattered set of sample points.

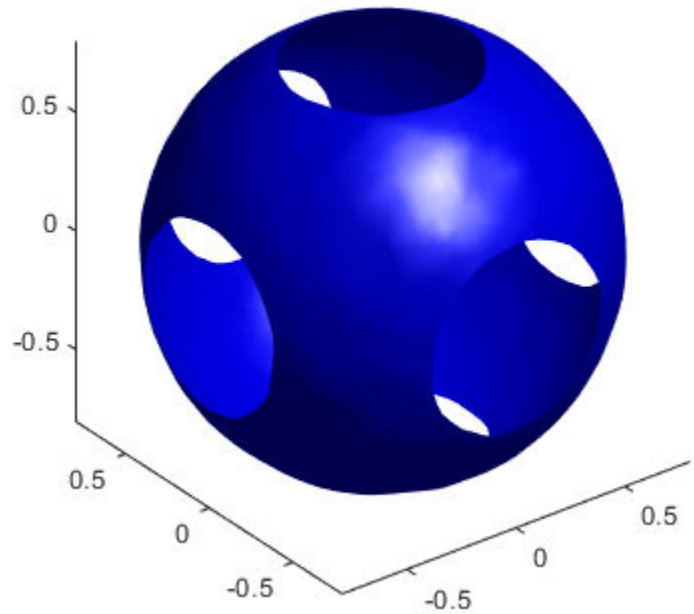
```
X = 2*gallery('uniformdata',[5000 3],0)-1;
Y = sum(X.^2,2);
```

Create x, y, and z grids to use as the 3-D set of query points, and interpolate the scattered data at these points.

```
d = -0.8:0.05:0.8;
[y0,x0,z0] = ndgrid(d,d,d);
XI = [x0(:) y0(:) z0(:)];
YI = griddatan(X,Y,XI);
```

Since it is difficult to visualize 4-D data sets, use isosurface at 0.8 to visualize the interpolation result.

```
YI = reshape(YI, size(x0));
p = patch(isosurface(x0,y0,z0,YI,0.8));
isonormals(x0,y0,z0,YI,p)
p.FaceColor = 'blue';
p.EdgeColor = 'none';
view(3)
axis equal
camlight
lighting phong
```



Nearest Neighbor Interpolation

Use nearest neighbor interpolation on a 3-D data set.

Create a sample 3-D data set. The matrix X contains the *xyz* locations of the observed data, and v contains the (randomly generated) observed data. This type of data set could represent, for example, oxygen levels in ocean water at these locations.

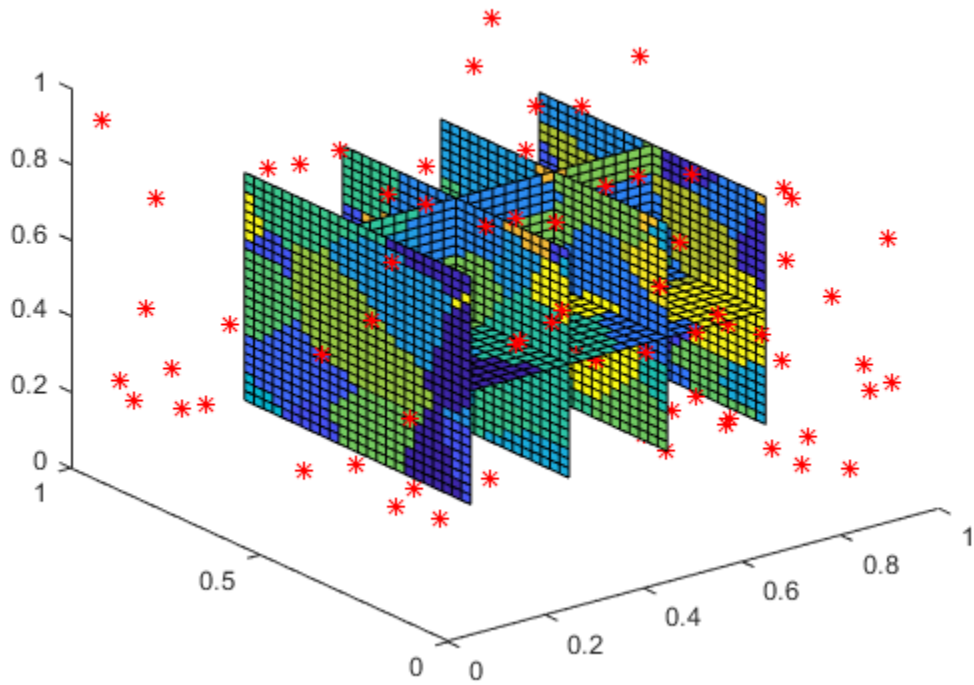
```
X = [rand(100,1) rand(100,1) rand(100,1)];
v = rand(100,1);
```

Use nearest neighbor interpolation to approximate the value of the underlying function at some query points.

```
[xx,yy,zz] = meshgrid(0.2:0.025:0.8);
xq = [xx(:) yy(:) zz(:)];
vq = griddatan(X,v,xq,'nearest');
```

Plot slices of the result on top of the sample point locations.

```
vq = reshape(vq,size(xx));
plot3(X(:,1),X(:,2),X(:,3),'r*')
hold on
slice(xx,yy,zz,vq,[0.2 0.4 0.6 0.8],0.5,0.5)
```



Input Arguments

x — Sample point coordinates
matrix

Sample point coordinates, specified as a matrix. Specify `x` as an `m`-by-`n` matrix to represent `m` points in `n`-dimensional space. The sample points must be unique.

Data Types: `single` | `double`

v — Sample values
vector

Sample values, specified as a vector. Specify `v` as a vector of length `m`, with one value for each sample point (row) specified in `x`.

If `v` contains complex numbers, then `griddatan` interpolates the real and imaginary parts separately.

Data Types: `single` | `double`
Complex Number Support: Yes

xq — Query points
matrix

Query points, specified as a matrix. Specify `xq` as a `p`-by-`n` matrix to represent `p` points in `n`-dimensional space. `xq` is usually created from a uniform grid produced by `ndgrid`.

Data Types: `single` | `double`

method — Interpolation method
`'linear'` (default) | `'nearest'` | `[]`

Interpolation method, specified as one of the values in this table. These methods are both based on a Delaunay triangulation of the input data.

Option	Description	Continuity
`'linear'` (default)	Triangulation-based linear interpolation.	C^0
`'nearest'`	Nearest neighbor interpolation.	Discontinuous

If `method` is `[]`, then `griddatan` uses the default `'linear'` method.

Data Types: `char`

`options` — Qhull-specific options
cell array | `[]` | `{''}`

Qhull-specific options, specified as a cell array. For a list of supported options, see Qhull Quick Reference.

If `options` is `[]`, then `griddatan` uses the default options:

- `{'Qt' 'Qbb' 'Qc'}` for 2-D and 3-D interpolations.
- `{'Qt' 'Qbb' 'Qc' 'Qx'}` for interpolations in 4+ dimensions.

If `options` is `{''}`, then `griddatan` does not use any options, not even the defaults.

Data Types: `cell`

Output Arguments

`vq` — Interpolated values
vector

Interpolated values, returned as a vector of length `p`. The interpolated values in `vq` correspond to the query points (rows) in `xq`.

Tips

- It is not practical to use `griddatan` for interpolation in dimensions higher than about 6-D, because the memory required by the underlying triangulation grows exponentially with the number of dimensions.
- Scattered data interpolation with `griddatan` uses a Delaunay triangulation of the data, so can be sensitive to scaling issues in `x`. When this occurs, you can use `normalize` to rescale the data and improve the results. See “Normalize Data with Differing Magnitudes” for more information.

See Also

`delaunayn` | `griddata` | `meshgrid`

Introduced before R2006a

griddedInterpolant

Gridded data interpolation

Description

Use `griddedInterpolant` to perform interpolation on a 1-D, 2-D, 3-D, or N-D gridded data on page 1-5628 set. `griddedInterpolant` returns the interpolant on page 1-5628 `F` for the given dataset. You can evaluate `F` at a set of query points, such as `(xq,yq)` in 2-D, to produce interpolated values `vq = F(xq,yq)`.

Use `scatteredInterpolant` to perform interpolation with scattered data on page 1-5629.

Creation

Syntax

```
F = griddedInterpolant
F = griddedInterpolant(x,v)
F = griddedInterpolant(X1,X2,...,Xn,V)
F = griddedInterpolant(V)
F = griddedInterpolant(gridVecs,V)
F = griddedInterpolant( ___ ,Method)
F = griddedInterpolant( ___ ,Method,ExtrapolationMethod)
```

Description

`F = griddedInterpolant` creates an empty gridded data interpolant object.

`F = griddedInterpolant(x,v)` creates a 1-D interpolant from a vector of sample points `x` and corresponding values `v`.

`F = griddedInterpolant(X1,X2,...,Xn,V)` creates a 2-D, 3-D, or N-D interpolant using a full grid on page 1-5629 of sample points passed as a set of `n`-dimensional arrays

`X1,X2,...,Xn`. The `V` array contains the sample values associated with the point locations in `X1,X2,...,Xn`. Each of the arrays `X1,X2,...,Xn` must be the same size as `V`.

`F = griddedInterpolant(V)` uses the default grid to create the interpolant. When you use this syntax, `griddedInterpolant` defines the grid as a set of points whose spacing is `1` and range is `[1, size(V,i)]` in the `i`th dimension. Use this syntax when you want to conserve memory and are not concerned about the absolute distances between points.

`F = griddedInterpolant(gridVecs,V)` specifies a cell array `gridVecs` that contains `n` grid vectors on page 1-5629 to describe an `n`-dimensional grid of sample points. Use this syntax when you want to use a specific grid and also conserve memory.

`F = griddedInterpolant(___,Method)` specifies an alternative interpolation method: `'linear'`, `'nearest'`, `'next'`, `'previous'`, `'pchip'`, `'cubic'`, `'makima'`, or `'spline'`. You can specify `Method` as the last input argument in any of the previous syntaxes.

`F = griddedInterpolant(___,Method,ExtrapolationMethod)` specifies both the interpolation and extrapolation methods. `griddedInterpolant` uses `ExtrapolationMethod` to estimate the value when your query points fall outside the domain of your sample points.

Input Arguments

x — Sample points
vector

Sample points, specified as a vector. `x` and `v` must be the same size. The sample points in `x` must be unique.

Data Types: `single` | `double`

v — Sample values
vector

Sample values, specified as a vector. `x` and `v` must be the same size.

Data Types: `single` | `double`

X1, X2, Xn — Sample points in full grid form
arrays

Sample points in full grid form, specified as separate `n`-dimensional arrays. The sample points must be unique and sorted. You can create the arrays `X1,X2,...,Xn` using the `ndgrid` function. These arrays are all the same size, and each one is the same size as `V`.

Data Types: `single` | `double`

`gridVecs` — Sample points in grid vector form
cell array of grid vectors

Sample points in grid vector form, specified as a cell array of grid vectors `{xg1,xg2,...,xgn}`. The sample points must be unique and sorted. The vectors must specify a grid that is the same size as `V`. In other words, `size(V) = [length(xg1) length(xg2),...,length(xgn)]`. Use this form as an alternative to the full grid to save memory when your grid is very large.

Data Types: `single` | `double`

`V` — Sample values
array

Sample values, specified as an array. The elements of `V` are the values that correspond to the sample points. The size of `V` must be the size of the full grid of sample points.

- If you specify the sample points as a full grid consisting of N-D arrays, then `V` must be the same size as any one of `X1,X2,...,Xn`.
- If you specify the sample points as grid vectors, then `size(V) = [length(xg1) length(xg2) ... length(xgn)]`.

Data Types: `single` | `double`

`Method` — Interpolation method
`'linear'` (default) | `'nearest'` | `'next'` | `'previous'` | `'pchip'` | `'cubic'` | `'spline'` | `'makima'`

Interpolation method, specified as one of the options in this table.

Method	Description	Continuity	Comments
`'linear'` (default)	Linear interpolation. The interpolated value at a query point is based on linear interpolation of the values at neighboring grid points in each respective dimension.	C^0	• Requires at least 2 grid points in each dimension • Requires more memory than `'nearest'`
`'nearest'`	Nearest neighbor interpolation. The interpolated value at a query point is the value at the nearest sample grid point.	Discontinuous	• Requires 2 grid points in each dimension • Fastest computation with modest memory requirements
`'next'`	Next neighbor interpolation (for 1-D only). The interpolated value at a query point is the value at the next sample grid point.	Discontinuous	• Requires at least 2 points • Same memory requirements and computation time as `'nearest'`
`'previous'`	Previous neighbor interpolation (for 1-D only). The interpolated value at a query point is the value at the previous sample grid point.	Discontinuous	• Requires at least 2 points • Same memory requirements and computation time as `'nearest'`
`'pchip'`	Shape-preserving piecewise cubic interpolation (for 1-D only). The interpolated value at a query point is based on a shape-preserving piecewise cubic interpolation of the values at neighboring grid points.	C^1	• Requires at least 4 points • Requires more memory and computation time than `'linear'`

Method	Description	Continuity	Comments
`'cubic'`	Cubic interpolation. The interpolated value at a query point is based on a cubic interpolation of the values at neighboring grid points in each respective dimension. The interpolation is based on a cubic convolution.	C^1	• Grid must have uniform spacing, although the spacing in each dimension does not have to be the same • Requires at least 4 points in each dimension • Requires more memory and computation time than `'linear'`
`'makima'`	Modified Akima cubic Hermite interpolation. The interpolated value at a query point is based on a piecewise function of polynomials with degree at most three evaluated using the values of neighboring grid points in each respective dimension. The Akima formula is modified to avoid overshoots.	C^1	• Requires at least 2 points in each dimension • Produces fewer undulations than `'spline'`, but does not flatten as aggressively as `'pchip'` • Computation is more expensive than `'pchip'`, but typically less than `'spline'` • Memory requirements are similar to those of `'spline'`
`'spline'`	Cubic spline interpolation. The interpolated value at a query point is based on a cubic interpolation of the values at neighboring grid points in each respective dimension. The interpolation is based on a cubic spline using not-a-knot end conditions.	C^2	• Requires 4 points in each dimension • Requires more memory and computation time than `'cubic'`

`ExtrapolationMethod` — Extrapolation method

`'linear'` (default) | `'nearest'` | `'next'` | `'previous'` | `'pchip'` | `'cubic'` | `'spline'` | `'makima'` | `'none'`

Extrapolation method, specified as `'linear'`, `'nearest'`, `'next'`, `'previous'`, `'pchip'`, `'cubic'`, `'spline'`, or `'makima'`. In addition, you can specify `'none'` if you want queries outside the domain of your grid to return `NaN` values.

If you omit `ExtrapolationMethod`, the default is the value you specify for `Method`. If you omit both the `Method` and `ExtrapolationMethod` arguments, both values default to `'linear'`.

Properties

`GridVectors` — Grid vectors
cell array

Grid vectors, specified as a cell array `{xg1,xg2,...,xgn}`. These vectors specify the grid points (locations) for the values in `Values`. The grid points must be unique.

Index-based editing of the properties of `F` is not supported. Instead, completely replace the `GridVectors` or `Values` arrays as necessary.

Data Types: `cell`

`Values` — Function values at sample points
array

Function values at sample points, specified as an array of values associated with the grid points in `GridVectors`.

Index-based editing of the properties of `F` is not supported. Instead, completely replace the `GridVectors` or `Values` arrays as necessary.

Data Types: `single` | `double`

`Method` — Interpolation method
`'linear'` (default) | `'nearest'` | `'next'` | `'previous'` | `'pchip'` | `'cubic'` | `'spline'` | `'makima'`

Interpolation method, specified as a character vector. `Method` can be: `'linear'`, `'nearest'`, `'next'`, `'previous'`, `'pchip'`, `'cubic'`, `'spline'`, or `'makima'`. See `Method` for descriptions of these methods.

Data Types: `char`

ExtrapolationMethod — Extrapolation method
`'linear'` | `'nearest'` | `'next'` | `'previous'` | `'pchip'` | `'cubic'` | `'spline'` | `'makima'` | `'none'`

Extrapolation method, specified as a character vector. `ExtrapolationMethod` can be: `'linear'`, `'nearest'`, `'next'`, `'previous'`, `'pchip'`, `'cubic'`, `'spline'`, `'makima'`, or `'none'`. A value of `'none'` indicates that extrapolation is disabled. The default value is the value of `Method`.

Data Types: `char`

Evaluate Interpolant at Specific Points

Use `griddedInterpolant` to create the interpolant on page 1-5628, `F`. Then you can evaluate `F` at specific points using any of the following syntaxes:

```
F = griddedInterpolant
F = griddedInterpolant(x,v)
F = griddedInterpolant(X1,X2,...,Xn,V)
F = griddedInterpolant(V)
F = griddedInterpolant(gridVecs,V)
F = griddedInterpolant(___,Method)
F = griddedInterpolant(___,Method,ExtrapolationMethod)
Vq = F(Xq)
Vq = F(xq1,xq2,...,xqn)
Vq = F(Xq1,Xq2,...,Xqn)
Vq = F({xgq1,xgq2,...,xgqn})
```

- `Vq = F(Xq)` specifies the query points in the matrix `Xq`. Each row of `Xq` contains the coordinates of a query point.
- `Vq = F(xq1,xq2,...,xqn)` specifies the query points `xq1,xq2,...,xqn` as column vectors of length `m` representing `m` points scattered in `n`-dimensional space.
- `Vq = F(Xq1,Xq2,...,Xqn)` specifies the query points using the `n`-dimensional arrays `Xq1,Xq2,...,Xqn`, which define a full grid on page 1-5629 of points.
- `Vq = F({xgq1,xgq2,...,xgqn})` specifies the query points as grid vectors on page 1-5629. Use this syntax to conserve memory when you want to query a large grid of points.

Examples

1-D Interpolation

Use `griddedInterpolant` to interpolate a 1-D data set.

Create a vector of scattered sample points `v`. The points are sampled at random 1-D locations between 0 and 20.

```
x = sort(20*rand(100,1));
v = besselj(0,x);
```

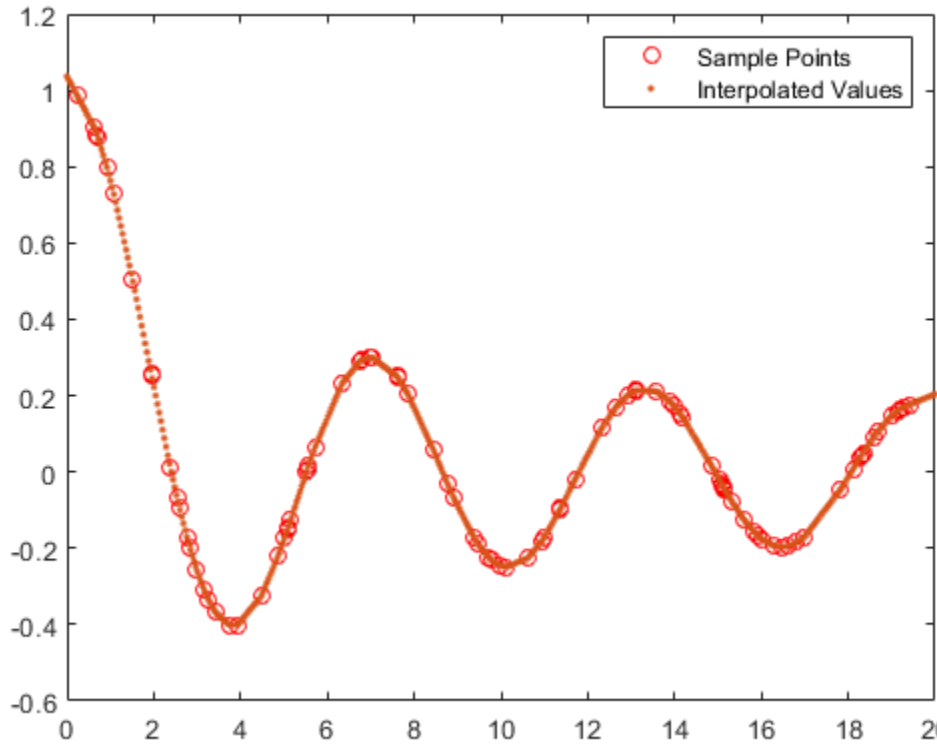
Create a gridded interpolant object for the data. By default, `griddedInterpolant` uses the `'linear'` interpolation method.

```
F = griddedInterpolant(x,v)
```

```
F = 
  griddedInterpolant with properties:

            GridVectors: {[100x1 double]}
                 Values: [100x1 double]
                 Method: 'linear'
    ExtrapolationMethod: 'linear'
```

Query the interpolant `F` at 500 uniformly spaced points between 0 and 20. Plot the interpolated results `(xq,vq)` on top of the original data `(x,v)`.

```
xq = linspace(0,20,500);
vq = F(xq);
plot(x,v,'ro')
hold on
plot(xq,vq,'.')
legend('Sample Points','Interpolated Values')
```

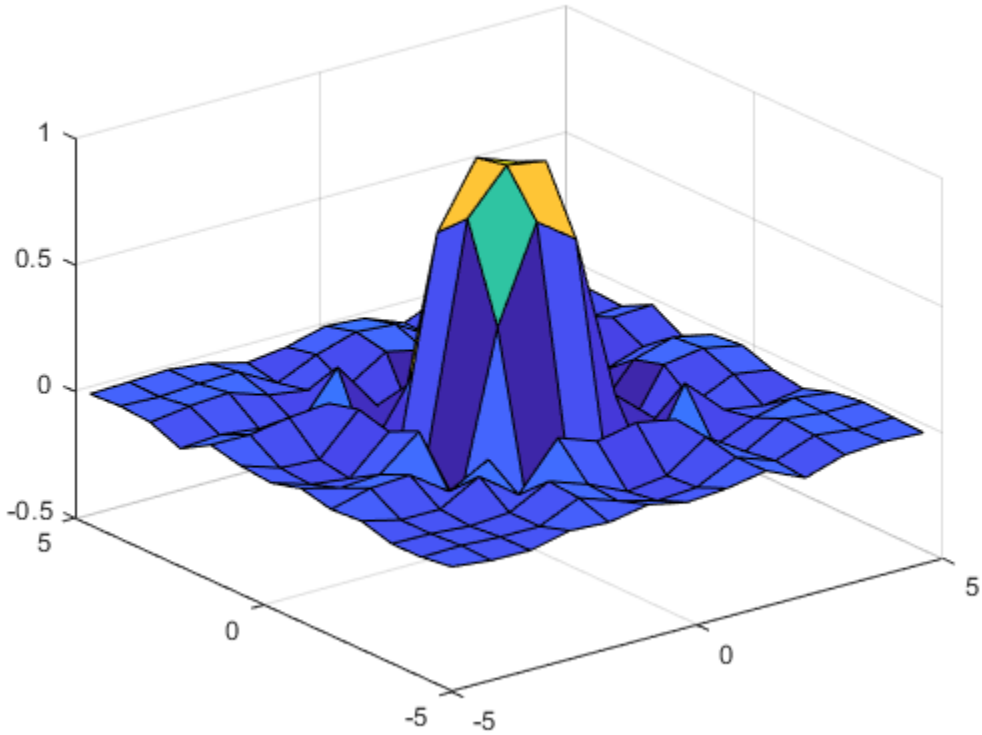


3-D Interpolation Using Full Grid vs. Grid Vectors

Interpolate 3-D data using two methods to specify the query points.

Create and plot a 3-D data set representing the function $z(x, y) = \frac{\sin\left(x^2 + y^2\right)}{x^2 + y^2}$ evaluated at a set of gridded sample points in the range [-5,5].

```
[x,y] = ndgrid(-5:0.8:5);
z = sin(x.^2 + y.^2) ./ (x.^2 + y.^2);
surf(x,y,z)
```

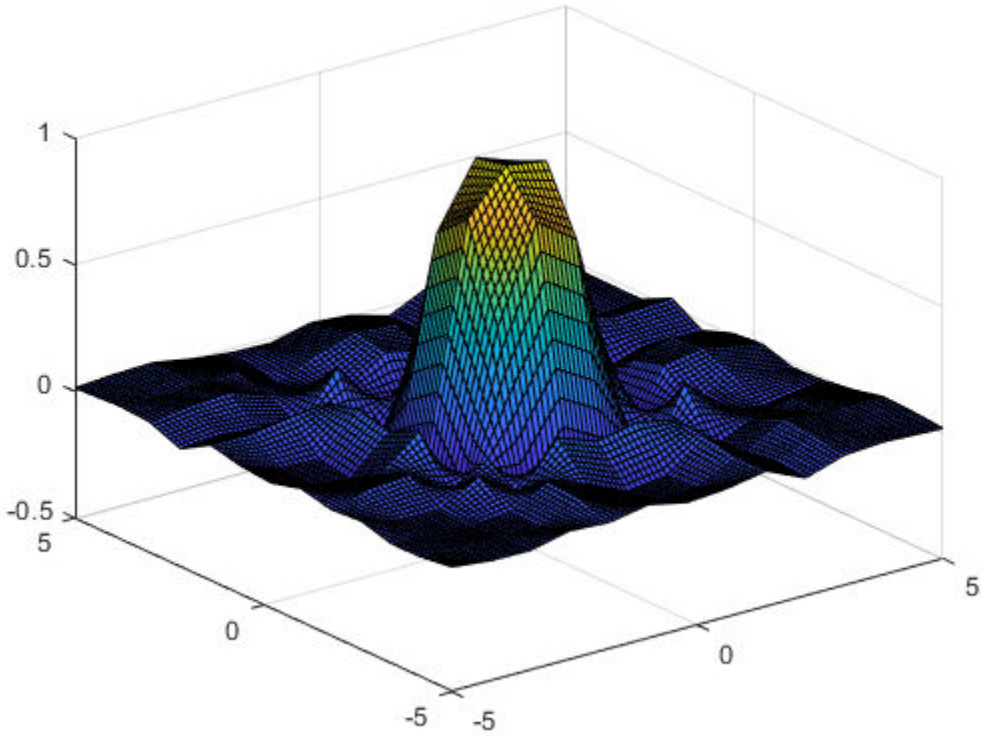


Create a gridded interpolant object for the data.

```
F = griddedInterpolant(x,y,z);
```

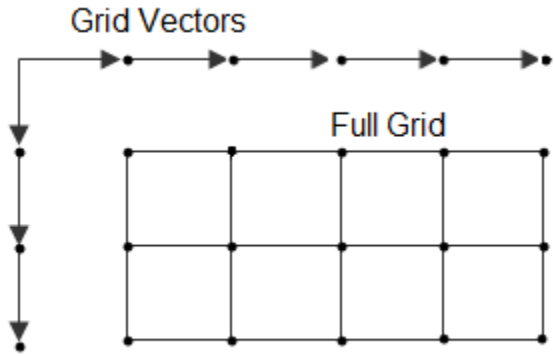
Use a finer mesh to query the interpolant and improve the resolution.

```
[xq,yq] = ndgrid(-5:0.1:5);
vq = F(xq,yq);
surf(xq,yq,vq)
```



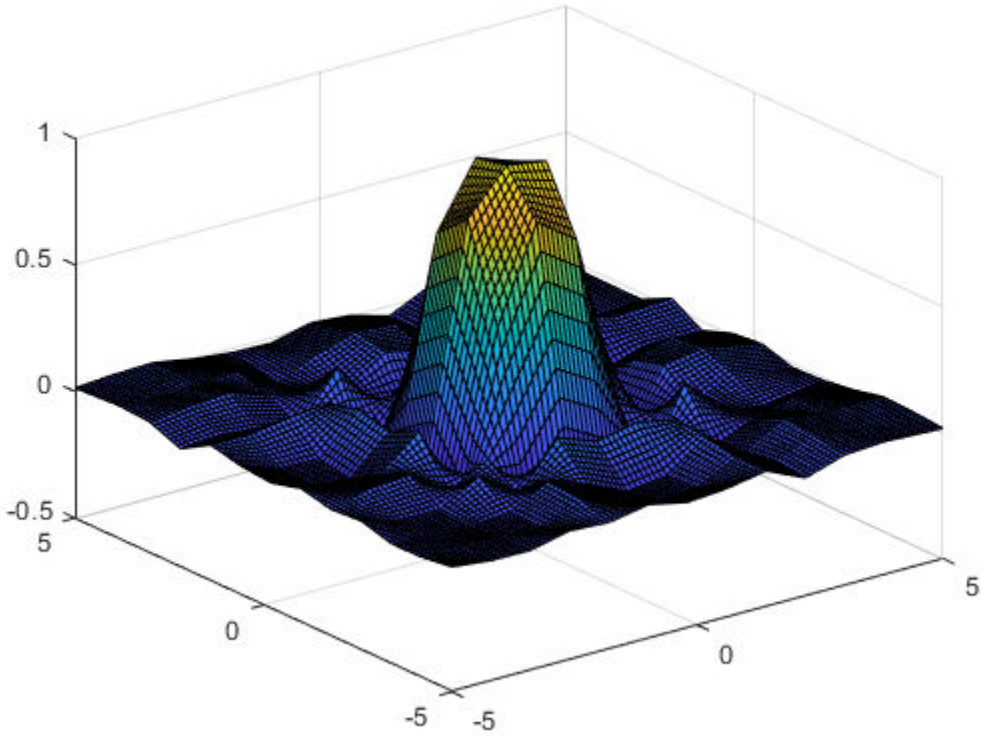
In cases where there are a lot of sample points or query points, and where memory usage becomes a concern, you can use *grid vectors* to improve memory usage.

- When you specify grid vectors instead of using `ndgrid` to create the full grid, `griddedInterpolant` avoids forming the full query grid to carry out the calculations.
- When you pass grid vectors, they are normally grouped together as cells in a cell array, `{xg1, xg2, ..., xgn}`. The grid vectors are a compact way to represent the points of the full grid.



Alternatively, execute the previous commands using grid vectors.

```
x = -5:0.8:5;
y = x';
z = sin(x.^2 + y.^2) ./ (x.^2 + y.^2);
F = griddedInterpolant({x,y},z);
xq = -5:0.1:5;
yq = xq';
vq = F({xq,yq});
surf(xq,yq,vq)
```



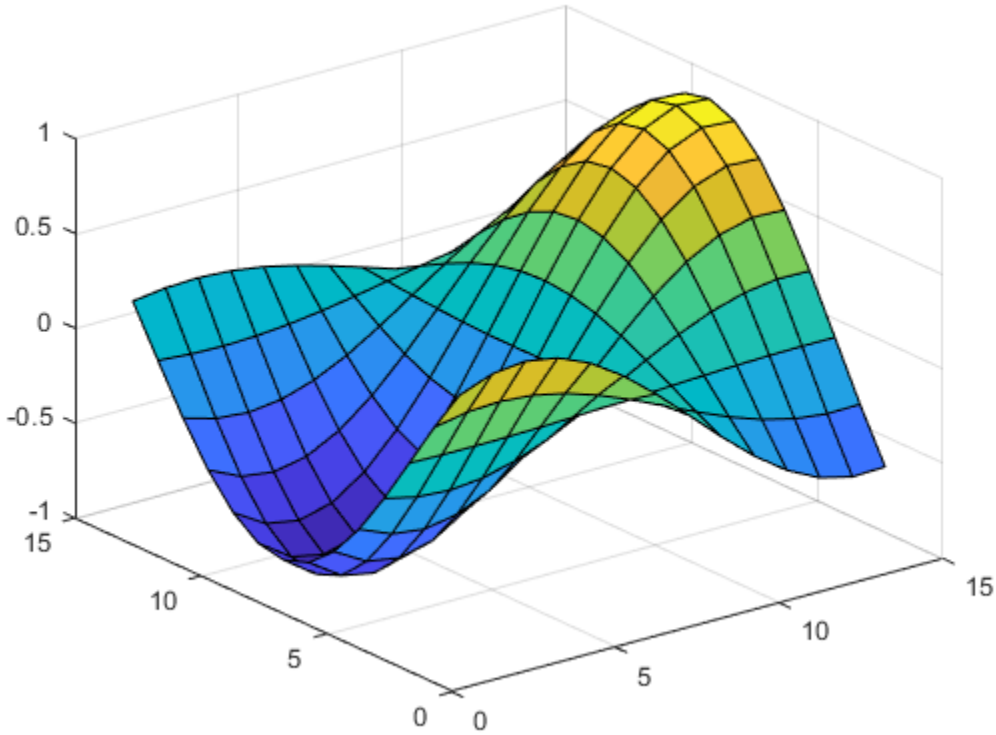
Interpolation with Default Grid

Use the default grid to perform a quick interpolation on a set of sample points. The default grid uses unit-spaced points, so this interpolation is useful when the exact *xy* spacing between the sample points is not important.

Create a matrix of sample function values and plot them against the default grid.

```
x = (1:0.3:5)';
y = x';
V = cos(x) .* sin(y);
```

```
n = length(x);
surf(1:n,1:n,V)
```



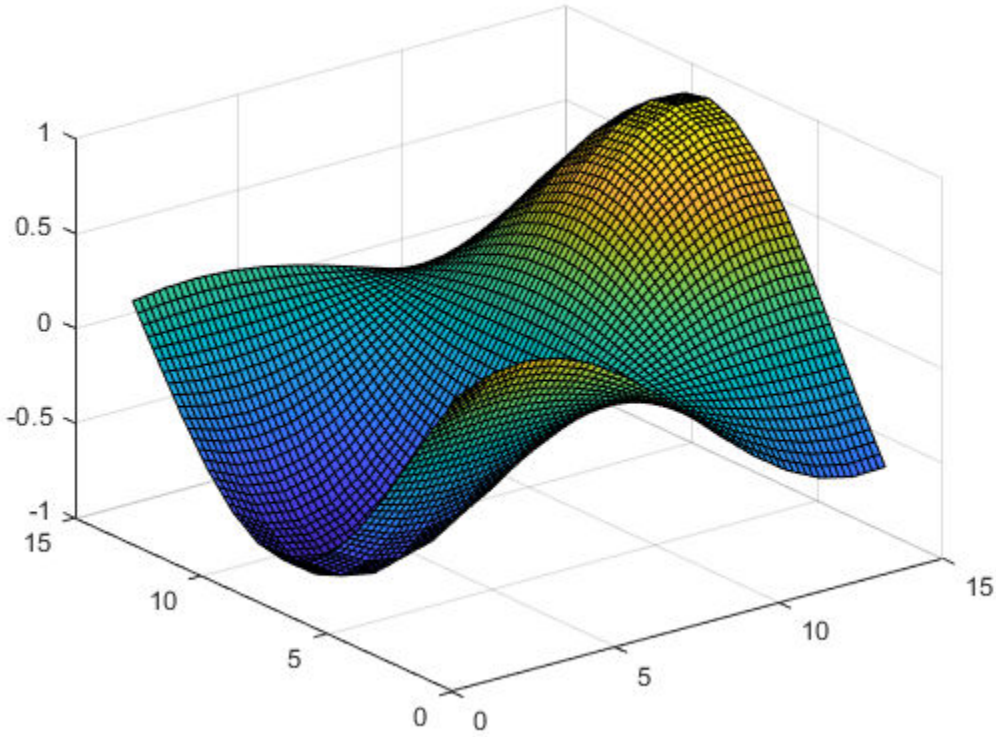
Interpolate the data using the default grid.

```
F = griddedInterpolant(V)
```

```
F =
  griddedInterpolant with properties:

            GridVectors: {[1 2 3 4 5 6 7 8 9 10 11 12 13 14]  [1x14 double]}
                 Values: [14x14 double]
                 Method: 'linear'
    ExtrapolationMethod: 'linear'
```

Query the interpolant and plot the results.

```
[xq,yq] = ndgrid(1:0.2:n);
Vq = F(xq,yq);
surf(xq',yq',Vq)
```



2-D Interpolation Over Finer Grid

Interpolate coarsely sampled data using a full grid with spacing of `0.5`.

Define the sample points as a full grid with range [1, 10] in both dimensions.

```
[X,Y] = ndgrid(1:10,1:10);
```

Sample $f(x, y) = x^2 + y^2$ at the grid points.

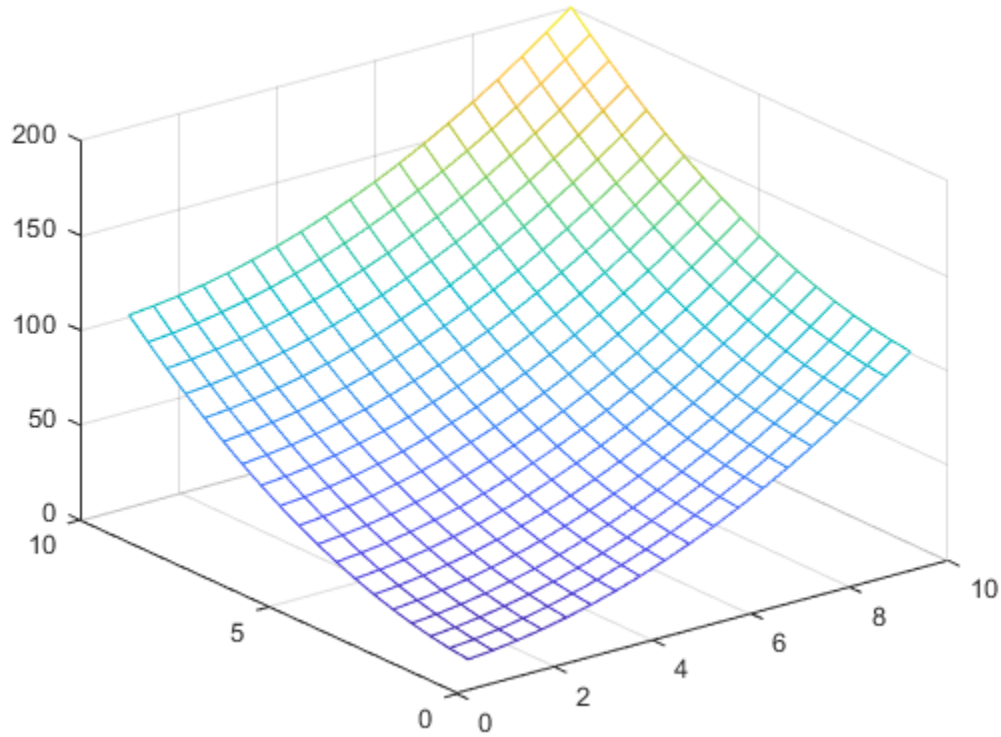
```
V = X.^2 + Y.^2;
```

Create the interpolant, specifying cubic interpolation.

```
F = griddedInterpolant(X,Y,V,'cubic');
```

Define a full grid of query points with `0.5` spacing and evaluate the interpolant at those points. Then plot the result.

```
[Xq,Yq] = ndgrid(1:0.5:10,1:0.5:10);
Vq = F(Xq,Yq);
mesh(Xq,Yq,Vq);
```



1-D Extrapolation

Compare results of querying the interpolant outside the domain of F using the `'pchip'` and `'nearest'` extrapolation methods.

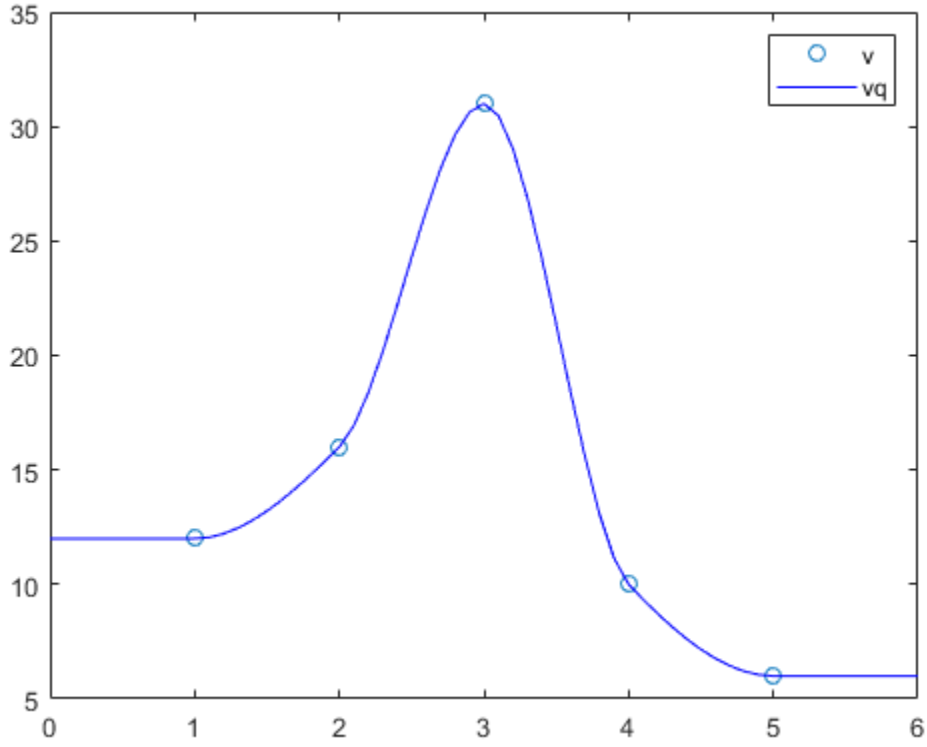
Create the interpolant, specifying `'pchip'` as the interpolation method and `'nearest'` as the extrapolation method.

```
x = [1 2 3 4 5];
v = [12 16 31 10 6];
F = griddedInterpolant(x,v,'pchip','nearest')
```

```
F =
  griddedInterpolant with properties:

            GridVectors: {[1 2 3 4 5]}
                 Values: [12 16 31 10 6]
                 Method: 'pchip'
    ExtrapolationMethod: 'nearest'
```

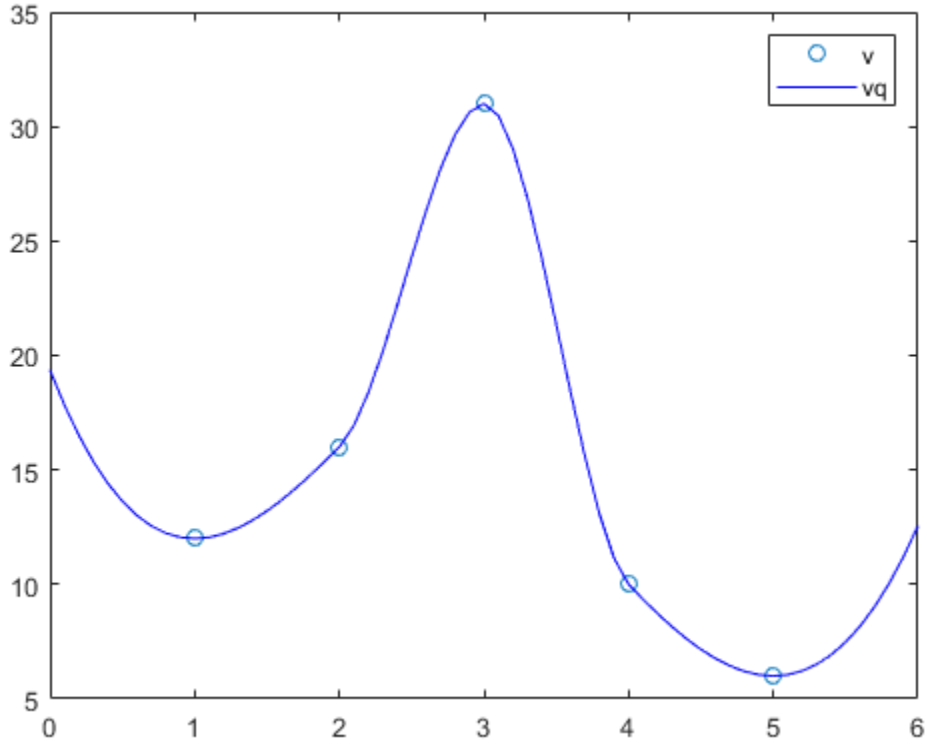
Query the interpolant, and include points outside the domain of F.

```
xq = 0:0.1:6;
vq = F(xq);
figure
plot(x,v,'o',xq,vq,'-b');
legend ('v','vq')
```



Query the interpolant at the same points again, this time using the pchip extrapolation method.

```
F.ExtrapolationMethod = 'pchip';
figure
vq = F(xq);
plot(x,v,'o',xq,vq,'-b');
legend ('v','vq')
```



More About

Interpolant

Interpolating function that you can evaluate at query points.

Gridded Data

A set of points that are axis-aligned and ordered.

Scattered Data

A set of points that have no structure among their relative locations.

Full Grid

A grid represented as a set of arrays. For example, you can create a full grid using `ndgrid`.

Grid Vectors

A set of vectors that serve as a compact representation of a grid in `ndgrid` format.

For example, `[X,Y] = ndgrid(xg,yg)` returns a full grid in the matrices `X` and `Y`. You can represent the same grid using the grid vectors `xg` and `yg`.

Tips

- It is quicker to evaluate a `griddedInterpolant` object `F` at many different sets of query points than it is to compute the interpolations separately using `interp1`, `interp2`, `interp3`, or `interpn`. For example:

```
% Fast to create interpolant F and evaluate multiple times
F = griddedInterpolant(X1,X2,V)
v1 = F(Xq1)
v2 = F(Xq2)

% Slower to compute interpolations separately using interp2
v1 = interp2(X1,X2,V,Xq1)
v2 = interp2(X1,X2,V,Xq2)
```

See Also

`fillmissing` | `filloutliers` | `interp1` | `interp2` | `interp3` | `interpn` | `meshgrid` | `ndgrid` | `scatteredInterpolant`

Topics

"Resample Image with Gridded Interpolation"

"Interpolating Gridded Data"

Introduced in R2011b

GridLayout Properties

Control behavior of grid layout manager

Description

Grid layout managers position UI components along the rows and columns of an invisible grid that spans the entire figure or a container within the figure. By changing property values of a grid layout, you can modify certain aspects of its behavior. Use dot notation to refer to a specific object and property:

```
uf = uifigure;
g = uigridlayout(uf);
g.ColumnWidth = {100,'1x'};
```

Properties

Grid

`ColumnWidth` — Column width

`{'1x','1x'}` (default) | cell array | string array | numeric array

Column width, specified as a cell array containing either `'fit'`, numbers, or numbers paired with `'x'` characters. You can specify any combination of values. The number of elements in the cell array controls the number of columns in the grid. For example, to create a 4-column grid, specify a 1-by-4 cell array. Column width can be specified as a string array or numeric array, only if the elements specified are of the same type, like `["1x" "2x" "1x"]` or `[100 200 50]`.

There are three different types of column widths:

- Fit width — Specify `'fit'`. Column width automatically adjusts to fit its contents. For text-based components, `'fit'` width adjusts with font properties to show the whole text. For non text-based components, `'fit'` width is based on the default size of the component and other factors. Use `'fit'` width if you want to avoid hard-coding the column width to fit components, or if your app is translated to another language or runs on different platforms.

- Fixed width in pixels — Specify a number. The column width is fixed at the number of pixels you specify. When the parent container resizes, the column width does not change.
- Variable width — Specify a number paired with an `'x'` character (for example, `'1x'`). When the parent container resizes, the column width grows or shrinks. Variable-width columns fill the remaining horizontal space that the fixed-width columns do not use. The number you pair with the `'x'` character is a weight for dividing up the remaining space among all the variable-width columns. If the grid has only one variable-width column, then it uses all the remaining space regardless of the number. If there are multiple variable-width columns that use the same number, then they share the space equally. Otherwise, the amount of space is proportional to the number.

For example, `{'fit',200,'2x','1x'}` specifies that the width of the first column is sized to fit its content, the second column is fixed at 200 pixels, and the last two columns share the remaining horizontal space. The third column uses twice as much space as the fourth column.

Changing certain aspects of a layout can affect the value of this property. For example, adding more components to a fully populated grid changes the size of the grid to accommodate the new components.

Changing the `ColumnWidth` property on a grid layout that already contains components does not change the layout of the components. For example, if you try to dynamically delete a column that contains components, the `ColumnWidth` property does not change until you move those components out of that column.

`RowHeight` — Row height

`{'1x','1x'}` (default) | cell array | string array | numeric array

Row height, specified as a cell array containing either `'fit'`, numbers, or numbers paired with `'x'` characters. You can specify any combination of values. The number of elements in the cell array controls the number of rows in the grid. For example, to create a grid that has 4 rows, specify a 1-by-4 cell array. Row height can be specified as a string array or numeric array, only if the elements specified are of the same type, like `["1x" "2x" "1x"]` or `[100 200 50]`.

There are three different types of row heights:

- Fit height — Specify `'fit'`. Row height automatically adjusts to fit its contents. For text-based components, `'fit'` height adjusts with font properties to show the whole text. For non text-based components, `'fit'` height is based on the default size of the

component and other factors. Use `'fit'` height if you want to avoid hard-coding the row height to fit components, or if your app is translated to another language or runs on different platforms.

- Fixed height in pixels — Specify a number. The row height is fixed at the number of pixels you specify. When the parent container resizes, the row height does not change.
- Variable height — Specify a number paired with an `'x'` character (for example, `'1x'`). When the parent container resizes, the row grows or shrinks. Variable-height rows fill the remaining vertical space that the fixed-height rows do not use. The number you pair with the `'x'` character is a weight for dividing up the remaining space among all the variable-height rows. If the grid has only one variable-height row, then it uses all the remaining space regardless of the number. If there are multiple variable-height rows that use the same number, then they share the space equally. Otherwise, the amount of space is proportional to the number.

For example, `{'fit',200,'2x','1x'}` specifies that the height of the first row is sized to fit its content, the second row is fixed at 200 pixels, and the last two rows share the remaining vertical space. The third row uses twice as much space as the fourth row.

Changing certain aspects of a layout can affect the value of this property. For example, adding more components to a fully populated grid changes the size of the grid to accommodate the new components.

Changing the `RowHeight` property on a grid layout that already contains components does not change the layout of the components. For example, if you try to dynamically delete a row that contains components, the `RowHeight` property does not change until you move those components out of that row.

`ColumnSpacing` — Column spacing
`10` (default) | number

Column spacing, specified as a scalar number of pixels between adjacent columns in the grid. The number you specify applies to all columns.

`RowSpacing` — Row spacing
`10` (default) | number

Row spacing, specified as a scalar number of pixels between adjacent rows in the grid. The number you specify applies to all rows.

`Padding` — Padding
`[10 10 10 10]` (default) | `[left bottom right top]`

Padding around the outer perimeter of the grid, specified as a vector of the form `[left bottom right top]`. The elements of the vector are described in the table below.

Vector Element	Description
`left`	Distance in pixels between the inner left edge of the parent container and the left edge of the grid.
`bottom`	Distance in pixels between the inner bottom edge of the parent container and the bottom edge of the grid.
`right`	Distance in pixels between the inner right edge of the parent container and the right edge of the grid.
`top`	Distance in pixels between the inner top edge of the parent container and the top edge of the grid. The inner top edge of the parent container starts below all decorations such as titles, tab labels, or menu bars.

Interactivity

`Visible` — Visibility of children

`'on'` (default) | `'off'`

Visibility of children, specified as `'on'` or `'off'`. Set this property to `'off'` to hide all child components in the grid and their descendants. The children and their descendants are hidden regardless of the value of their `Visible` properties. When components are hidden, you can get and set their properties even though they do not appear in the app.

When you set this property to `'on'`, the children and their descendants are visible only if their `Visible` properties are also set to `'on'`.

Setting the `Visible` property on the grid does not change the values of the `Visible` properties of its descendants.

`Scrollable` — Ability to scroll

`'off'` (default) | `'on'`

Ability to scroll, specified as `'off'` or `'on'`. Setting this property to `'on'` enables scrolling within the grid. In order to scroll, these conditions must also be met:

- The sum of the values specified for the `'RowHeight'` property of the grid must be larger than the height of the parent container.

- The sum of the values specified for the `'ColumnWidth'` property of the grid must be larger than the width of the parent container.
- At least one row or column of the grid must be set to a fixed pixel height or width.
- The grid must contain components.

Certain types of charts and axes do not support scrollable containers. However, you can place the chart or axes in a nonscrollable panel, and then place the panel in the scrollable container. For more information, see “Displaying Graphics in App Designer”.

Position

`Layout` — Layout options

empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for a nested grid layout container. If the grid layout is not a child of another grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the grid layout is a child of another grid layout, you can place that child grid in the desired row and column of the parent grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code nests `grid2` in the third row and second column of `grid1`.

```
grid1 = uigridlayout([4 3]);
grid2 = uigridlayout(grid1);
grid2.Layout.Row = 3;
grid2.Layout.Column = 2;
```

To make the child grid span multiple rows or columns of its parent grid, specify the `Row` or `Column` property as a two-element vector. For example, this command spans `grid2` over columns `2` through `3` of `grid1`:

```
grid2.Layout.Column = [2 3];
```

Callbacks

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.

- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.

- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`Children` — Children

empty `GraphicsPlaceholderArray` | array of UI component objects

Children, returned as an array of UI component objects. Use this property to view the list of children or to reorder the children by setting the property to a permutation of itself. You cannot add or remove children using this property. To add a child to this list, set the `Parent` property of the child UI component.

Reordering the children has no effect on the location of the components in the grid. To change the location of a component in a grid, set its `Layout` property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.

HandleVisibility Value	Description
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uigridlayout'`

This property is read-only.

Type of graphics object, returned as `'uigridlayout'`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`appdesigner` | `scroll` | `uigridlayout`

Topics

"Use Grid Layout Managers"

Introduced in R2018b

groot

Graphics root object

Syntax

```
groot
r = groot
```

Description

`groot` refers to the graphics root object. Use `groot` to access root properties. For a list of properties, see Root.

`r = groot` stores the graphics root object handle. To set root properties using dot notation, you must store the handle first.

Examples

View Root Property Values

View a list of graphics root properties and their current values.

```
get(groot)
```

```
         CallbackObject: [0x0 GraphicsPlaceholder]
               Children: [0x0 GraphicsPlaceholder]
          CurrentFigure: [0x0 GraphicsPlaceholder]
     FixedWidthFontName: 'Courier New'
       HandleVisibility: 'on'
       MonitorPositions: [1 1 1280 1024]
                 Parent: [0x0 GraphicsPlaceholder]
         PointerLocation: [1 1]
             ScreenDepth: 32
    ScreenPixelsPerInch: 96
             ScreenSize: [1 1 1280 1024]
```

```
      ShowHiddenHandles: 'off'
                    Tag: ''
                   Type: 'root'
                  Units: 'pixels'
               UserData: []
```

Set Root Property Values

Set graphics root property values by storing the object handle and using dot notation.

```
r = groot;
r.ShowHiddenHandles = 'on';
```

Tips

- Use the graphics root object to set default values on the root level for other types of objects. For example, set the default colormap for all future figures to the `summer` colormap.

  ```
  set(groot,'DefaultFigureColormap',summer)
  ```

  To restore a property to its original MATLAB default, use the `'remove'` keyword.

  ```
  set(groot,'DefaultFigureColormap','remove')
  ```

  For more information on setting default values, see “Default Property Values”.

See Also

Functions
`gca` | `gcf`

Properties
Root

Introduced in R2014b

Group Properties

Group object appearance and behavior

Description

`Group` object properties control the behavior of `Group` objects. By changing property values, you can modify certain aspects of the `Group` object.

Starting in R2014b, you can use dot notation to query and set properties.

```
g = hggroup;
c = g.Children;
g.Visible = 'off';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Legend

`DisplayName` — Legend label
`''` (default) | character vector | string scalar

Legend label, specified as a character vector or string scalar. The legend does not display until you call the `legend` command. If you do not specify the text, then `legend` sets the label using the form `'dataN'`.

`Annotation` — Control for including or excluding object from legend
`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend.

- `'off'` — Do not include the object in the legend. (default)
- `'children'` — Include only children of the object as separate items in the legend.

Interactivity

`Visible` — Visibility

`'on'` (default) | `'off'`

Visibility, specified as one of these values:

- `'on'` — Display all objects in the `Group` object.
- `'off'` — Hide all objects in the `Group` object. You still can access the properties of invisible `Group` objects. Setting the `Visible` property for the `Group` object does not change the `Visible` property for objects in the `Group` object.

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.

- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Children that can capture mouse clicks
`'visible'` (default) | `'none'`

Children that can capture mouse clicks, specified as one of these values:

- `'visible'` — Any child object can capture a mouse click, depending on the `PickableParts` property value of the child.
- `'none'` — No child objects can capture mouse clicks, regardless of the `PickableParts` property value of the child.

`HitTest` — Response to mouse clicks captured by children
`'on'` (default) | `'off'`

Response to mouse clicks captured by children, specified one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of `Group` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Do not trigger the callbacks of the `Group` object. Instead, trigger the callbacks for the nearest ancestor that has a `HitTest` property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

A `Group` object cannot capture mouse clicks. However, if you click a child of the `Group` object and if the child has a `HitTest` property set to `'off'`, then the child passes the click to the `Group` object.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent
`Axes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `Group`, or `Transform` object.

`Children` — Children
empty `GraphicsPlaceholder` array | array of graphics objects

Children, returned as an array of graphics objects. Use this property to view a list of the children or to reorder the children by setting the property to a permutation of itself.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the child graphics object to the `Group` object.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'hggroup'`

This property is read-only.

Type of graphics object, returned as `'hggroup'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`hggroup`

Topics

“Access Property Values”
“Graphics Object Properties”

Introduced before R2006a

groupcounts

Number of group elements

Syntax

```
G = groupcounts(T,groupvars)
G = groupcounts(T,groupvars,groupbins)
G = groupcounts( ___ ,Name,Value)

B = groupcounts(A)
B = groupcounts(A,groupbins)
B = groupcounts( ___ ,Name,Value)
[B,BG] = groupcounts(A, ___ )
```

Description

`G = groupcounts(T,groupvars)` computes the number of elements in each group of data in a table or timetable, and returns a table containing the groups and their counts. A group is the set of unique combinations of grouping variables in `groupvars`. For example, `G = groupcounts(T,'Gender')` returns the number of `Male` elements and the number of `Female` elements in the variable `Gender`.

`G = groupcounts(T,groupvars,groupbins)` specifies how to bin the data in `groupvars`. For example, `G = groupcounts(T,'SaleDate','year')` gives the yearly sales counts according to `SaleDate`.

`G = groupcounts( ___ ,Name,Value)` specifies additional grouping properties using one or more name-value pairs for any of the previous syntaxes. For example, `G = groupcounts(T,'Category1','IncludeMissingGroups',false)` excludes the group made from missing `categorical` data indicated by `<undefined>`.

`B = groupcounts(A)` returns a vector containing the group counts for each unique combination of grouping vectors represented in a column vector, matrix, or cell array of column vectors `A`.

`B = groupcounts(A,groupbins)` bins the data according to `groupbins`.

`B = groupcounts( ___ ,Name,Value)` specifies additional grouping properties using one or more name-value pairs.

`[B,BG] = groupcounts(A, ___ )` also returns the groups corresponding to the counts in `B`.

Examples

Group Table Variables

Compute the number of group elements from table data.

Create a table containing information about five individuals.

```
Gender = ["male";"female";"male";"female";"male"];
Smoker = logical([1;0;1;0;1]);
Weight = [176;163;131;133;119];
T = table(Gender,Smoker,Weight)
```

```
T=5×3 table
     Gender     Smoker    Weight
    ________    ______    ______

    "male"      true       176
    "female"    false      163
    "male"      true       131
    "female"    false      133
    "male"      true       119
```

Count the number of elements in each group by gender.

```
G1 = groupcounts(T,'Gender')
```

```
G1=2×2 table
     Gender     GroupCount
    ________    __________

    "female"        2
    "male"          3
```

Count the number of elements in each group by gender and smoker status. By default, `groupcounts` suppresses groups with zero elements.

```
G2 = groupcounts(T,{'Gender','Smoker'})
```

```
G2=2×3 table
     Gender     Smoker    GroupCount
    ________    ______    __________

    "female"    false         2
    "male"      true          3
```

To count all groups, including those with zero elements, specify the `'IncludeEmptyGroups'` parameter with value `true`.

```
G3 = groupcounts(T,{'Gender','Smoker'},'IncludeEmptyGroups',true)
```

```
G3=4×3 table
     Gender     Smoker    GroupCount
    ________    ______    __________

    "female"    false         2
    "female"    true          0
    "male"      false         0
    "male"      true          3
```

Specify Group Bins

Group data according to specified bins.

Create a timetable containing sales information for days within a single month.

```
TimeStamps = datetime([2017 3 4; 2017 3 2; 2017 3 15; 2017 3 10;...
                       2017 3 14; 2017 3 31; 2017 3 25;...
                       2017 3 29; 2017 3 21; 2017 3 18]);
Profit = [2032 3071 1185 2587 1998 2899 3112 909 2619 3085]';
TotalItemsSold = [14 13 8 5 10 16 8 6 7 11]';
TT = timetable(TimeStamps,Profit,TotalItemsSold)
```

```
TT=10×3 timetable
    TimeStamps     Profit    TotalItemsSold
```

```
    ___________    ______    _______________

    04-Mar-2017    2032            14
    02-Mar-2017    3071            13
    15-Mar-2017    1185             8
    10-Mar-2017    2587             5
    14-Mar-2017    1998            10
    31-Mar-2017    2899            16
    25-Mar-2017    3112             8
    29-Mar-2017     909             6
    21-Mar-2017    2619             7
    18-Mar-2017    3085            11
```

Compute the group counts by the total items sold, binning the groups into intervals of item numbers.

```
G = groupcounts(TT,'TotalItemsSold',[0 4 8 12 16])
```

```
G=3×2 table
    disc_TotalItemsSold    GroupCount
    ___________________    __________

         [4, 8)                3
         [8, 12)               4
         [12, 16]              3
```

Compute the group counts grouped by day of the week.

```
G = groupcounts(TT,'TimeStamps','dayname')
```

```
G=5×2 table
    dayname_TimeStamps    GroupCount
    __________________    __________

        Tuesday               2
        Wednesday             2
        Thursday              1
        Friday                2
        Saturday              3
```

Multiple Grouping Vectors for Vector Input

Compute the group counts for four groups based on their gender and smoker status.

Store patient information as three vectors of different types.

```
Gender = ["male";"female";"male";"female";"male"];
Smoker = logical([1;0;1;0;1]);
Weight = [176;163;131;133;119];
```

Grouping by gender and smoker status, compute the group counts. B contains the counts for each group. BG is a cell array containing two vectors that describe the groups as you look at their elements rowwise. For instance, the first row of BG{1} says that the patients in the first group are female, and the first row of BG{2} says that they are nonsmokers. The count for that group is 2, found in the corresponding row of B.

```
[B,BG] = groupcounts({Gender,Smoker},'IncludeEmptyGroups',true);
B
```

```
B = 4×1

     2
     0
     0
     3
```

```
BG{1}
```

```
ans = 4x1 string array
    "female"
    "female"
    "male"
    "male"
```

```
BG{2}
```

```
ans = 4x1 logical array

   0
   1
   0
   1
```

Input Arguments

T — Input data
table | timetable

Input data, specified as a table or timetable.

A — Input vectors
column vector | matrix | cell array of column vectors

Input array, specified as a column vector, matrix, or cell array of column vectors representing grouping vectors. When A is a matrix, the grouping vectors are columnwise.

groupvars — Grouping variables
scalar | vector | matrix | cell array | function handle

Grouping variables for table input, specified as a scalar, vector, matrix, cell array, or function handle.

groupvars indicates which columns of the input table to use to compute the groups, and can be one of the following:

- A character vector or scalar string specifying a single table variable name
- A cell array of character vectors or a string array, where each element is a table variable name
- A vector of table variable indices
- A logical vector whose elements each correspond to a table variable, where true includes the corresponding variable and false excludes it
- A function handle that takes a table variable as input and returns a logical scalar

Example: 'Age'

Example: {'Height','Weight'}

groupbins — Binning scheme
'none' (default) | character vector | scalar | vector | cell array

Binning scheme, specified as one of the following options:

- `'none'`, indicating the groups are returned according to the specified grouping variables only
- A list of bin edges, specified as a numeric vector, or a `datetime` vector for `datetime` grouping variables or vectors
- A number of bins, specified as an integer scalar
- A time duration, specified as a scalar of type `duration` or `calendarDuration`, indicating bin widths (for `datetime` or `duration` grouping variables or vectors only)
- A cell array listing binning rules for each grouping variable or vector
- A time bin for `datetime` and `duration` grouping variables or vectors only, specified as one of the following character vectors:

Value	Description	Data Type
`'second'`	Each bin is 1 second.	`datetime` and `duration`
`'minute'`	Each bin is 1 minute.	`datetime` and `duration`
`'hour'`	Each bin is 1 hour.	`datetime` and `duration`
`'day'`	Each bin is 1 calendar day. This value accounts for Daylight Saving Time shifts.	`datetime` and `duration`
`'week'`	Each bin is 1 calendar week.	`datetime` only
`'month'`	Each bin is 1 calendar month.	`datetime` only
`'quarter'`	Each bin is 1 calendar quarter.	`datetime` only
`'year'`	Each bin is 1 calendar year. This value accounts for leap days.	`datetime` and `duration`
`'decade'`	Each bin is 1 decade (10 calendar years).	`datetime` only
`'century'`	Each bin is 1 century (100 calendar years).	`datetime` only
`'secondofminute'`	Bins are seconds from 0 to 59.	`datetime` only

Value	Description	Data Type
`'minuteofhour'`	Bins are minutes from 0 to 59.	`datetime` only
`'hourofday'`	Bins are hours from 0 to 23.	`datetime` only
`'dayofweek'`	Bins are days from 1 to 7. The first day of the week is Sunday.	`datetime` only
`'dayname'`	Bins are full day names such as `'Sunday'`.	`datetime` only
`'dayofmonth'`	Bins are days from 1 to 31.	`datetime` only
`'dayofyear'`	Bins are days from 1 to 366.	`datetime` only
`'weekofmonth'`	Bins are weeks from 1 to 6.	`datetime` only
`'weekofyear'`	Bins are weeks from 1 to 54.	`datetime` only
`'monthname'`	Bins are full month names such as `'January'`.	`datetime` only
`'monthofyear'`	Bins are months from 1 to 12.	`datetime` only
`'quarterofyear'`	Bins are quarters from 1 to 4.	`datetime` only

When multiple grouping variables or vectors are specified, you can provide a single binning rule that is applied to all grouping variables, or a cell array containing a binning method for each grouping variable, such as `{'none',[0 2 4 Inf]}`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `G = groupcounts(T,groupvars,groupbins,'IncludedEdge','right')`

`IncludedEdge` — Included bin edge
`'left'` (default) | `'right'`

Included bin edge, specified as either `'left'` or `'right'`, indicating which end of the bin interval is inclusive.

This name-value pair can only be specified when `groupbins` is specified, and the value is applied to all binning schemes for all grouping variables or vectors.

`IncludeMissingGroups` — Missing groups indicator
`true` or `1` (default) | `false` or `0`

Missing groups indicator, specified as a numeric or logical `1` (`true`) or `0` (`false`). If the parameter value is `true`, then `groupcounts` displays groups made up of missing values, such as `NaN`. If the parameter value is `false`, then `groupcounts` does not display the missing value groups.

`IncludeEmptyGroups` — Empty groups indicator
`false` or `0` (default) | `true` or `1`

Empty groups indicator, specified as a numeric or logical `0` (`false`) or `1` (`true`). If the parameter value is `false`, then `groupcounts` does not display groups with zero elements. If the parameter value is `true`, then `groupcounts` displays the empty groups.

Output Arguments

`G` — Output table
table

Output table, returned as a table containing the computed groups and the number of elements in each group. For a single grouping variable, the output groups are sorted according to the order returned by the `unique` function with the `'sorted'` option.

`B` — Group counts
column vector

Group counts for non-table input data, returned as a column vector containing the number of elements in each group.

BG — Groups
column vector | cell array of column vectors

Groups for non-table input data, returned as a column vector or cell array of column vectors. For a single grouping vector, the output groups are sorted according to the order returned by the `unique` function with the `'sorted'` option.

When you provide more than one input vector, `BG` is a cell array containing column vectors of equal length. The group information can be found by looking at the elements rowwise across all vectors in `BG`. The count for each group is contained in the corresponding row of the first output argument `B`.

Tips

- When making many calls to `groupcounts`, consider converting grouping variables to type `categorical` or `logical` when possible for improved performance. For example, if you have a grouping variable of type `char` (such as `Gender` with elements `'Male'` and `'Female'`), you can convert it to a categorical variable using the command `categorical(Gender)`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- The first input argument does not support cell arrays.
- The `groupvars` argument does not support function handles.
- The `'IncludeEmptyGroups'` name-value pair is not supported.
- The order of the groups might be different compared to in-memory `groupcounts` calculations.
- When grouping by discretized datetime arrays, the categorical group names are different compared to in-memory `groupcounts` calculations.

For more information, see “Tall Arrays”.

See Also

`discretize` | `findgroups` | `groupfilter` | `groupsummary` | `grouptransform` | `rowfun` | `splitapply` | `varfun`

Introduced in R2019a

groupfilter

Filter by group

Syntax

```
G = groupfilter(T,groupvars,method)
G = groupfilter(T,groupvars,groupbins,method)
G = groupfilter( ___ ,datavars)
G = groupfilter(T,groupvars,groupbins,method, ___ ,'IncludedEdge',LR)

B = groupfilter(A,groupvars,method)
B = groupfilter(A,groupvars,groupbins,method)
B = groupfilter(A,groupvars,groupbins,method,'IncludedEdge',LR)
[B,BG] = groupfilter(A, ___ )
```

Description

`G = groupfilter(T,groupvars,method)` filters the data in a table or timetable according to the computation in `method`, grouping by the unique combinations of grouping variables specified in `groupvars`. The `method` argument is a function handle that operates on each nongrouping variable. The output `G` is a table or timetable containing only the rows of `T` that satisfy the condition in `method`. For example, `G = groupfilter(T,'Trial',@(x) numel(x) > 5)` groups the data in `T` by `Trial`, keeping the rows that belong to groups with more than five trials.

`G = groupfilter(T,groupvars,groupbins,method)` bins the data according to `groupbins` before applying the filter, horizontally concatenating the groups to the output table. For example, `G = groupfilter(T,'SaleDate','year',@(x) numel(x) > 5)` bins the data in `SaleDate` by year, and keeps the rows whose bin has more than five elements.

`G = groupfilter( ___ ,datavars)` specifies the table variables to apply the filter to for either of the previous syntaxes. For example, `G = groupfilter(T,'Trial',@(x) x == max(x),'Height')` keeps the rows of `T` corresponding to the maximum height for each trial.

`G = groupfilter(T,groupvars,groupbins,method, ___ ,'IncludedEdge',LR)` specifies the included bin edge as `'left'` or `'right'` to indicate which end of the bin interval is inclusive when `groupbins` is specified.

`B = groupfilter(A,groupvars,method)` filters the data according to the unique combinations of grouping vectors in `groupvars` when `A` is a vector or matrix. `groupvars` can be a column vector, matrix, or cell array of column vectors. The output `B` is a matrix containing the rows of `A` that satisfy the condition defined by `method`.

`B = groupfilter(A,groupvars,groupbins,method)` bins the data according to `groupbins`.

`B = groupfilter(A,groupvars,groupbins,method,'IncludedEdge',LR)` specifies whether to include the left or right edge in each bin when `groupbins` is specified.

`[B,BG] = groupfilter(A, ___ )` also returns the rows of the grouping vectors corresponding to the unfiltered rows in `A`.

Examples

Filter Two-Variable Table

Create a table containing two variables.

```
groupID = [1 1 1 2 2 3]';
sample = [3 1 2 9 8 5]';
T = table(groupID,sample)
```

```
T=6×2 table
    groupID    sample
    _______    ______

       1          3
       1          1
       1          2
       2          9
       2          8
       3          5
```

Group by ID number, and return only rows corresponding to groups with more than two samples.

```
Gnumel = groupfilter(T,'groupID',@(x) numel(x) > 2)
```

```
Gnumel=3×2 table
    groupID    sample
    _______    ______

       1         3
       1         1
       1         2
```

Return only rows whose group samples are between 0 and 6.

```
Gvals = groupfilter(T,'groupID',@(x) min(x) > 0 && max(x) < 6)
```

```
Gvals=4×2 table
    groupID    sample
    _______    ______

       1         3
       1         1
       1         2
       3         5
```

Group by Largest Values

Create a table containing two variables that represent a day number and temperature.

```
daynum = [1 1 1 1 2 2 2 2]';
temp = [67 65 71 55 61 79 58 78]';
T = table(daynum,temp)
```

```
T=8×2 table
    daynum    temp
    ______    ____

      1        67
      1        65
      1        71
```

```
      1        55
      2        61
      2        79
      2        58
      2        78
```

Group by day number, and return the largest two temperatures for each day.

```
G = groupfilter(T,'daynum',@(x) ismember(x,maxk(x,2)))
```

```
G=4×2 table
    daynum    temp
    ______    ____

      1        67
      1        71
      2        79
      2        78
```

Group by Month

Create a table of dates and corresponding profits.

```
timeStamps = datetime([2017 3 4; 2017 3 2; 2017 3 15; 2017 4 10;...
                       2017 4 14; 2017 4 30; 2017 5 25;...
                       2017 5 29; 2017 5 21]);
profit = [2032 3071 1185 2587 1998 2899 3112 909 2619]';
T = table(timeStamps,profit)
```

```
T=9×2 table
    timeStamps     profit
    ___________    ______

    04-Mar-2017     2032
    02-Mar-2017     3071
    15-Mar-2017     1185
    10-Apr-2017     2587
    14-Apr-2017     1998
    30-Apr-2017     2899
    25-May-2017     3112
```

```
    29-May-2017      909
    21-May-2017     2619
```

Group the dates by month, and return only rows that correspond to the maximum profit for that month.

```
Gmax = groupfilter(T,'timeStamps','month',@(x) x == max(x))
```

```
Gmax=3×3 table
    timeStamps     profit    month_timeStamps
    ___________    ______    ________________

    02-Mar-2017     3071         Mar-2017
    30-Apr-2017     2899         Apr-2017
    25-May-2017     3112         May-2017
```

Return rows whose month had an average profit greater than $2300.

```
Gavg = groupfilter(T,'timeStamps','month',@(x) mean(x) > 2300)
```

```
Gavg=3×3 table
    timeStamps     profit    month_timeStamps
    ___________    ______    ________________

    10-Apr-2017     2587         Apr-2017
    14-Apr-2017     1998         Apr-2017
    30-Apr-2017     2899         Apr-2017
```

Filter Three-Variable Table

Create a table containing three variables.

```
groupID = [1 2 3 1 2 3 1 2 3]';
height = [62 61 59 66 70 72 57 67 71]';
gender = ["M" "F" "F" "M" "M" "F" "M" "M" "M"]';
T = table(groupID,height,gender)
```

```
T=9×3 table
    groupID    height    gender
    _______    ______    ______
```

```
    1         62       "M"
    2         61       "F"
    3         59       "F"
    1         66       "M"
    2         70       "M"
    3         72       "F"
    1         57       "M"
    2         67       "M"
    3         71       "M"
```

Group by ID number, and return rows whose group members all have a minimum height of 60.

```
G1 = groupfilter(T,'groupID',@(x) min(x) >= 60,'height')
```

```
G1=3×3 table
    groupID    height    gender
    _______    ______    ______

       2         61       "F"
       2         70       "M"
       2         67       "M"
```

Return rows whose group members are all male.

```
G2 = groupfilter(T,'groupID',@(x) all(x == "M"),'gender')
```

```
G2=3×3 table
    groupID    height    gender
    _______    ______    ______

       1         62       "M"
       1         66       "M"
       1         57       "M"
```

Group by ID number and gender, and return rows containing the maximum height for males and females in each group.

```
G3 = groupfilter(T,{'groupID','gender'},@(x) x == max(x))
```

```
G3=5×3 table
    groupID    height    gender
```

```
    _______    ______    ______

       2         61       "F"
       1         66       "M"
       2         70       "M"
       3         72       "F"
       3         71       "M"
```

Filter with Vector Data

Create a vector of dates and a vector of corresponding profit values.

```
timeStamps = datetime([2017 3 4; 2017 3 2; 2017 3 15; 2017 3 10; ...
                       2017 3 14; 2017 3 31; 2017 3 25; ...
                       2017 3 29; 2017 3 21; 2017 3 18]);
profit = [2032 3071 1185 2587 1998 2899 3112 909 2619 3085]';
```

Group by day of the week, and compute the maximum profit for each group. Display the maximum profits and their corresponding groups.

```
[maxDailyProfit,dayOfWeek] = groupfilter(profit,timeStamps,'dayname',@(x) x == max(x))
```

```
maxDailyProfit = 5×1

        3071
        1185
        2899
        3112
        2619


dayOfWeek = 5x1 categorical array
     Thursday
     Wednesday
     Friday
     Saturday
     Tuesday
```

Input Arguments

`T` — Input data
table | timetable

Input data, specified as a table or timetable.

`A` — Input array
vector | matrix

Input array, specified as a vector or matrix.

`groupvars` — Grouping variables or vectors
scalar | vector | matrix | cell array | function handle

Grouping variables or vectors, specified as a scalar, vector, matrix, cell array, or function handle.

For table or timetable input, `groupvars` indicates which columns to use to compute the groups, and can be one of the following:

- A character vector or string scalar specifying a single table variable name
- A cell array of character vectors or a string array, where each element is a table variable name
- A vector of table variable indices
- A logical vector whose elements each correspond to a table variable, where `true` includes the corresponding variable and `false` excludes it
- A function handle that takes a table variable as input and returns a logical scalar

For array input, `groupvars` can be either a column vector with the same number of rows as `A` or a group of column vectors arranged in a matrix or a cell array.

Example: `'Age'`

Example: `{'Height','Weight'}`

`method` — Computation method
function handle

Computation method, specified as a function handle.

`method` defines the function used to filter out members from each group. The function must return a logical scalar or a logical column vector with the same number of rows as the input data indicating which group members to select. If the function returns a logical scalar, then either all members of the group are filtered out (when the value is `false`) or none are filtered out (when the value is `true`). If the function returns a logical vector, then members of groups are filtered out when the corresponding element is `false`. Members are kept when the corresponding element is `true`.

To define the function handle, use a syntax of the form `@(inputargs) myfunc` where `myfunc` depends on `inputargs`. For example, `@(x) mean(x)` designates the variable `x` as the input argument to the `mean` function. You can then define a condition by which to filter, such as `@(x) mean(x) > 10`, which returns a logical scalar. This computation keeps rows whose group has a mean larger than 10. For more information, see “Create Function Handle” and “Anonymous Functions”.

When `groupfilter` applies the method to more than one nongrouping variable at a time, the method must return `true` for each variable in order to keep the row.

`groupbins` — Binning scheme

`'none'` (default) | character vector | scalar | vector | cell array

Binning scheme, specified as one of the following options:

- `'none'`, indicating no binning
- A list of bin edges, specified as a numeric vector, or a `datetime` vector for `datetime` grouping variables
- A number of bins, specified as an integer scalar
- A time duration, specified as a scalar of type `duration` or `calendarDuration`, indicating bin widths (for `datetime` or `duration` grouping variables only)
- A cell array listing binning rules for each grouping variable
- A time bin for `datetime` and `duration` grouping variables only, specified as one of the following character vectors:

Value	Description	Data Type
`'second'`	Each bin is 1 second.	`datetime` and `duration`
`'minute'`	Each bin is 1 minute.	`datetime` and `duration`
`'hour'`	Each bin is 1 hour.	`datetime` and `duration`

Value	Description	Data Type
`'day'`	Each bin is 1 calendar day. This value accounts for Daylight Saving Time shifts.	`datetime` and `duration`
`'week'`	Each bin is 1 calendar week.	`datetime` only
`'month'`	Each bin is 1 calendar month.	`datetime` only
`'quarter'`	Each bin is 1 calendar quarter.	`datetime` only
`'year'`	Each bin is 1 calendar year. This value accounts for leap days.	`datetime` and `duration`
`'decade'`	Each bin is 1 decade (10 calendar years).	`datetime` only
`'century'`	Each bin is 1 century (100 calendar years).	`datetime` only
`'secondofminute'`	Bins are seconds from 0 to 59.	`datetime` only
`'minuteofhour'`	Bins are minutes from 0 to 59.	`datetime` only
`'hourofday'`	Bins are hours from 0 to 23.	`datetime` only
`'dayofweek'`	Bins are days from 1 to 7. The first day of the week is Sunday.	`datetime` only
`'dayname'`	Bins are full day names such as `'Sunday'`.	`datetime` only
`'dayofmonth'`	Bins are days from 1 to 31.	`datetime` only
`'dayofyear'`	Bins are days from 1 to 366.	`datetime` only

Value	Description	Data Type
`'weekofmonth'`	Bins are weeks from 1 to 6.	`datetime` only
`'weekofyear'`	Bins are weeks from 1 to 54.	`datetime` only
`'monthname'`	Bins are full month names such as `'January'`.	`datetime` only
`'monthofyear'`	Bins are months from 1 to 12.	`datetime` only
`'quarterofyear'`	Bins are quarters from 1 to 4.	`datetime` only

When multiple grouping variables are specified, you can provide a single binning rule that is applied to all grouping variables, or a cell array containing a binning method for each grouping variable such as `{'none',[0 2 4 Inf]}`.

LR — Included bin edge
`'left'` (default) | `'right'`

Included bin edge, specified as either `'left'` or `'right'`, indicating which end of the bin interval is inclusive.

This argument can be specified only when `groupbins` is specified and the value applies all binning schemes for all grouping variables or vectors.

`datavars` — Data variables
scalar | vector | cell array | function handle

Data variables for table or timetable input, specified as a scalar, vector, cell array, or function handle. `datavars` indicates which columns of the input table to apply the methods to, and can be one of the following options:

- A character vector or string scalar specifying a single table variable name
- A cell array of character vectors or string array where each element is a table variable name
- A vector of table variable indices
- A logical vector whose elements each correspond to a table variable, where `true` includes the corresponding variable and `false` excludes it

- A function handle that takes a table variable as input and returns a logical scalar

When `datavars` is not specified, `groupfilter` applies the computations to each nongrouping variable.

Example: `'Profit'`

Example: `{'Income','Expenses'}`

Example: `@isnumeric`

Output Arguments

`G` — Output table
table

Output table, returned as a table containing the unfiltered rows of data.

`B` — Output array
vector | matrix

Output array, returned as a vector or matrix containing the unfiltered data.

`BG` — Grouping vectors
column vector | cell array of column vectors

Grouping vectors for array input data, returned as a column vector or cell array of column vectors. `BG` contains the rows of the grouping vectors corresponding to the unfiltered rows in `A`.

See Also

`discretize` | `findgroups` | `groupcounts` | `groupsummary` | `grouptransform` | `rowfun` | `splitapply` | `varfun`

Introduced in R2019b

groupsummary

Group summary computations

Syntax

```
G = groupsummary(T,groupvars)
G = groupsummary(T,groupvars,method)
G = groupsummary(T,groupvars,method,datavars)
G = groupsummary(T,groupvars,groupbins)
G = groupsummary(T,groupvars,groupbins,method)
G = groupsummary(T,groupvars,groupbins,method,datavars)
G = groupsummary( ___ ,Name,Value)
```

```
B = groupsummary(A,groupvars,method)
B = groupsummary(A,groupvars,groupbins,method)
B = groupsummary( ___ ,Name,Value)
[B,BG] = groupsummary(A, ___ )
[B,BG,BC] = groupsummary(A, ___ )
```

Description

`G = groupsummary(T,groupvars)` returns a table containing the computed groups and the number of elements in each group for data in a table or timetable `T`. A group contains the unique combinations of grouping variables in `groupvars`. For example, `G = groupsummary(T,'Gender')` returns the number of `Male` elements and the number of `Female` elements in the variable `Gender`.

`G = groupsummary(T,groupvars,method)` also returns the computations specified in `method`. For example, `G = groupsummary(T,'Gender','median')` returns the median of all nongrouping variables in `T` for both genders, in addition to the number of elements in each group.

`G = groupsummary(T,groupvars,method,datavars)` specifies the variables to apply the computations to.

`G = groupsummary(T,groupvars,groupbins)` specifies how to bin the data in `groupvars`. For example, `G = groupsummary(T,'SaleDate','year')` gives the

group counts for all sales in T within each year according to the grouping variable SaleDate.

G = groupsummary(T,groupvars,groupbins,method) bins the data according to groupbins for the computations specified in method.

G = groupsummary(T,groupvars,groupbins,method,datavars) bins the data according to groupbins and specifies the variables to apply the computations to.

G = groupsummary(___ ,Name,Value) specifies additional grouping properties using one or more name-value pairs for any of the previous syntaxes. For example, G = groupsummary(T,'Category1','IncludeMissingGroups',false) excludes the group made from missing categorical data indicated by <undefined>.

B = groupsummary(A,groupvars,method) returns the computations according to the unique combinations of grouping vectors in groupvars when A is a vector or matrix. groupvars can be a column vector, matrix, or cell array of column vectors.

B = groupsummary(A,groupvars,groupbins,method) bins the data according to groupbins.

B = groupsummary(___ ,Name,Value) specifies additional grouping properties using one or more name-value pairs for either of the previous array syntaxes.

[B,BG] = groupsummary(A, ___) also returns the groups for each grouping vector.

[B,BG,BC] = groupsummary(A, ___) also returns the group counts for each group.

Examples

Summary Statistics

Compute summary statistics on table variables.

Create a table T that contains information about five individuals.

```
Gender = ["male";"female";"female";"male";"male"];
Age = [38;43;38;40;49];
Height = [71;69;64;67;64];
Weight = [176;163;131;133;119];
T = table(Gender,Age,Height,Weight)
```

```
T=5×4 table
     Gender      Age    Height    Weight
    ________     ___    ______    ______

    "male"       38       71       176
    "female"     43       69       163
    "female"     38       64       131
    "male"       40       67       133
    "male"       49       64       119
```

Compute the counts of males and females by specifying `'Gender'` as the grouping variable.

```
G = groupsummary(T,'Gender')
```

```
G=2×2 table
     Gender      GroupCount
    ________     __________

    "female"         2
    "male"           3
```

Compute the mean age, height, and weight of females and males separately.

```
G = groupsummary(T,'Gender','mean')
```

```
G=2×5 table
     Gender      GroupCount    mean_Age    mean_Height    mean_Weight
    ________     __________    ________    ___________    ___________

    "female"         2            40.5         66.5            147
    "male"           3           42.333       67.333         142.67
```

Still grouping by gender, compute the median height only.

```
G = groupsummary(T,'Gender','median','Height')
```

```
G=2×3 table
     Gender      GroupCount    median_Height
    ________     __________    _____________

    "female"         2             66.5
```

```
    "male"          3                67
```

Multiple Grouping Variables

Group table data using two grouping variables.

Create a table T that contains information about five individuals.

```
Gender = ["male";"female";"male";"female";"male"];
Smoker = logical([1;0;1;0;1]);
Weight = [176;163;131;133;119];
T = table(Gender,Smoker,Weight)
```

```
T=5×3 table
     Gender     Smoker    Weight
    ________    ______    ______

    "male"      true       176
    "female"    false      163
    "male"      true       131
    "female"    false      133
    "male"      true       119
```

Compute the mean weight, grouped by gender and smoking status. By default, two combinations of gender and smoking status are not represented in the output because they are empty groups.

```
G = groupsummary(T,{'Gender','Smoker'},'mean','Weight')
```

```
G=2×4 table
     Gender     Smoker    GroupCount    mean_Weight
    ________    ______    __________    ___________

    "female"    false         2             148
    "male"      true          3             142
```

Set the `'IncludeEmptyGroups'` parameter value to `true` in order to see all group combinations, including the empty ones.

```
G = groupsummary(T,{'Gender','Smoker'},'mean','Weight','IncludeEmptyGroups',true)
```

```
G=4×4 table
     Gender      Smoker    GroupCount    mean_Weight
    ________     ______    __________    ___________

    "female"     false         2             148
    "female"     true          0             NaN
    "male"       false         0             NaN
    "male"       true          3             142
```

Specify Group Bins

Group data according to specified bins.

Create a timetable containing sales information for days within a single month.

```
TimeStamps = datetime([2017 3 4; 2017 3 2; 2017 3 15; 2017 3 10;...
                       2017 3 14; 2017 3 31; 2017 3 25;...
                       2017 3 29; 2017 3 21; 2017 3 18]);
Profit = [2032 3071 1185 2587 1998 2899 3112 909 2619 3085]';
TotalItemsSold = [14 13 8 5 10 16 8 6 7 11]';
TT = timetable(TimeStamps,Profit,TotalItemsSold)
```

```
TT=10×2 timetable
    TimeStamps     Profit    TotalItemsSold
    __________     ______    ______________

    04-Mar-2017     2032           14
    02-Mar-2017     3071           13
    15-Mar-2017     1185            8
    10-Mar-2017     2587            5
    14-Mar-2017     1998           10
    31-Mar-2017     2899           16
    25-Mar-2017     3112            8
    29-Mar-2017      909            6
    21-Mar-2017     2619            7
    18-Mar-2017     3085           11
```

Compute the mean profit grouped by the total items sold, binning the groups into intervals of item numbers.

```
format shorte
G = groupsummary(TT,'TotalItemsSold',[0 4 8 12 16],'mean','Profit')
```

```
G=3×3 table
    disc_TotalItemsSold    GroupCount    mean_Profit
    ___________________    __________    ___________

         [4, 8)            3.0000e+00    2.0383e+03
         [8, 12)           4.0000e+00    2.3450e+03
         [12, 16]          3.0000e+00    2.6673e+03
```

Compute the mean profit grouped by day of the week.

```
G = groupsummary(TT,'TimeStamps','dayname','mean','Profit')
```

```
G=5×3 table
    dayname_TimeStamps    GroupCount    mean_Profit
    __________________    __________    ___________

        Tuesday           2.0000e+00    2.3085e+03
        Wednesday         2.0000e+00    1.0470e+03
        Thursday          1.0000e+00    3.0710e+03
        Friday            2.0000e+00    2.7430e+03
        Saturday          3.0000e+00    2.7430e+03
```

Group Operations with Vector Data

Create a vector of dates and a vector of corresponding profit values.

```
timeStamps = datetime([2017 3 4; 2017 3 2; 2017 3 15; 2017 3 10; ...
                       2017 3 14; 2017 3 31; 2017 3 25; ...
                       2017 3 29; 2017 3 21; 2017 3 18]);
profit = [2032 3071 1185 2587 1998 2899 3112 909 2619 3085]';
```

Compute the mean profit by day of the week. Display the means, the group names, and the number of members in each group.

```
format shorte
[meanDailyProfit,dayOfWeek,dailyCounts] = groupsummary(profit,timeStamps,'dayname','mea
```

```
meanDailyProfit = 5×1
```

```
   2.3085e+03
   1.0470e+03
   3.0710e+03
   2.7430e+03
   2.7430e+03


dayOfWeek = 5x1 categorical array
     Tuesday
     Wednesday
     Thursday
     Friday
     Saturday


dailyCounts = 5×1

     2
     2
     1
     2
     3
```

Multiple Grouping Vectors for Vector Input

Compute the mean weights for four groups based on their gender and smoker status.

Store patient information as three vectors of different types.

```
Gender = ["male";"female";"male";"female";"male"];
Smoker = logical([1;0;1;0;1]);
Weight = [176;163;131;133;119];
```

Grouping by gender and smoker status, compute the mean weights. `B` contains the mean for each group (`NaN` for empty groups). `BG` is a cell array containing two vectors that describe the groups as you look at their elements rowwise. For instance, the first row of `BG{1}` says that the patients in the first group are female, and the first row of `BG{2}` says that they are nonsmokers. Finally, `BC` contains the number of members in each group for the corresponding groups in `BG`.

```
[B,BG,BC] = groupsummary(Weight,{Gender,Smoker},'mean','IncludeEmptyGroups',true);
B
```

```
B = 4×1

   148
   NaN
   NaN
   142
```

```
BG{1}
```

```
ans = 4x1 string array
    "female"
    "female"
    "male"
    "male"
```

```
BG{2}
```

```
ans = 4x1 logical array

   0
   1
   0
   1
```

```
BC
```

```
BC = 4×1

     2
     0
     0
     3
```

Input Arguments

T — Input data
table | timetable

Input data, specified as a table or timetable.

`A` — Input array
vector | matrix

Input array, specified as a vector or matrix.

`groupvars` — Grouping variables or vectors
scalar | vector | matrix | cell array | function handle

Grouping variables or vectors, specified as a scalar, vector, matrix, cell array, or function handle.

For table or timetable input data, `groupvars` indicates which columns to use to compute the groups, and can be one of the following:

- A character vector or scalar string specifying a single table variable name
- A cell array of character vectors or a string array, where each element is a table variable name
- A vector of table variable indices
- A logical vector whose elements each correspond to a table variable, where `true` includes the corresponding variable and `false` excludes it
- A function handle that takes a table variable as input and returns a logical scalar

For array input, `groupvars` can be either a column vector with the same number of rows as `A` or a group of column vectors arranged in a matrix or cell array.

Example: `'Age'`

Example: `{'Height','Weight'}`

`method` — Computation method
`'sum'` | `'mean'` | `'median'` | `'mode'` | `'var'` | `'std'` | `'min'` | `'max'` | `'range'` | `'nummissing'` | `'nnz'` | `'all'` | function handle | cell array

Computation method, specified as one of the following:

- `'sum'` — sum
- `'mean'` — mean
- `'median'` — median

- `'mode'` — mode
- `'var'` — variance
- `'std'` — standard deviation
- `'min'` — minimum
- `'max'` — maximum
- `'range'` — maximum minus minimum
- `'nummissing'` — number of missing elements
- `'nnz'` — number of nonzero and non-`NaN` elements
- `'all'` — all computations previously listed

You can also specify a function handle that returns one entity per group whose first dimension has length 1. To specify multiple computations at a time, list the options in a cell array, such as `{'mean','median'}`.

`datavars` — Data variables

scalar | vector | cell array | function handle

Data variables for table or timetable input, specified as a scalar, vector, cell array, or function handle. `datavars` indicates which variables of the input table to apply the methods to, and can be one of the following options:

- A character vector or scalar string specifying a single table variable name
- A cell array of character vectors or string array where each element is a table variable name
- A vector of table variable indices
- A logical vector whose elements each correspond to a table variable, where `true` includes the corresponding variable and `false` excludes it
- A function handle that takes a table variable as input and returns a logical scalar

When `datavars` is not specified, `groupsummary` applies the computations to each nongrouping variable.

Example: `'Profit'`

Example: `{'Income','Expenses'}`

Example: `@isnumeric`

groupbins — Binning scheme
`'none'` (default) | vector | scalar | cell array

Binning scheme, specified as one of the following options:

- `'none'`, indicating no binning
- A list of bin edges, specified as a numeric vector, or a `datetime` vector for `datetime` grouping variables or vectors
- A number of bins, specified as an integer scalar
- A time duration, specified as a scalar of type `duration` or `calendarDuration` indicating bin widths (for `datetime` or `duration` grouping variables or vectors only)
- A time bin for `datetime` and `duration` grouping variables or vectors only, specified as one of the following character vectors:

Value	Description	Data Type
`'second'`	Each bin is 1 second.	`datetime` and `duration`
`'minute'`	Each bin is 1 minute.	`datetime` and `duration`
`'hour'`	Each bin is 1 hour.	`datetime` and `duration`
`'day'`	Each bin is 1 calendar day. This value accounts for Daylight Saving Time shifts.	`datetime` and `duration`
`'week'`	Each bin is 1 calendar week.	`datetime` only
`'month'`	Each bin is 1 calendar month.	`datetime` only
`'quarter'`	Each bin is 1 calendar quarter.	`datetime` only
`'year'`	Each bin is 1 calendar year. This value accounts for leap days.	`datetime` and `duration`
`'decade'`	Each bin is 1 decade (10 calendar years).	`datetime` only
`'century'`	Each bin is 1 century (100 calendar years).	`datetime` only

Value	Description	Data Type
`'secondofminute'`	Bins are seconds from 0 to 59.	`datetime` only
`'minuteofhour'`	Bins are minutes from 0 to 59.	`datetime` only
`'hourofday'`	Bins are hours from 0 to 23.	`datetime` only
`'dayofweek'`	Bins are days from 1 to 7. The first day of the week is Sunday.	`datetime` only
`'dayname'`	Bins are full day names such as `'Sunday'`.	`datetime` only
`'dayofmonth'`	Bins are days from 1 to 31.	`datetime` only
`'dayofyear'`	Bins are days from 1 to 366.	`datetime` only
`'weekofmonth'`	Bins are weeks from 1 to 6.	`datetime` only
`'weekofyear'`	Bins are weeks from 1 to 54.	`datetime` only
`'monthname'`	Bins are full month names such as `'January'`.	`datetime` only
`'monthofyear'`	Bins are months from 1 to 12.	`datetime` only
`'quarterofyear'`	Bins are quarters from 1 to 4.	`datetime` only

- A cell array listing binning rules for each grouping variable or vector

When multiple grouping variables are specified, you can provide a single binning rule that is applied to all grouping variables, or a cell array containing a binning method for each grouping variable such as `{'none',[0 2 4 Inf]}`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `G = groupsummary(T,groupvars,groupbins,'IncludedEdge','right')`

`IncludedEdge` — Included bin edge
`'left'` (default) | `'right'`

Included bin edge, specified as either `'left'` or `'right'`, indicating which end of the bin interval is inclusive.

This name-value pair can only be specified when `groupbins` is specified, and the value is applied to all binning schemes for all grouping variables or vectors.

`IncludeMissingGroups` — Missing groups indicator
`true` (default) | `false`

Missing groups indicator, specified as `true` or `false`. When the parameter value is `true`, `groupsummary` displays groups made up of missing values, such as `NaN`. When the parameter value is `false`, `groupsummary` does not display the missing groups.

Data Types: `logical`

`IncludeEmptyGroups` — Empty groups indicator
`false` (default) | `true`

Empty groups indicator, specified as `true` or `false`. When the parameter value is `false`, `groupsummary` does not display groups with zero elements. When the parameter value is `true`, `groupsummary` displays the empty groups.

Data Types: `logical`

Output Arguments

`G` — Output table
table

Output table, returned as a table containing the specified computations for each group.

B — Output array
vector | matrix

Output array, returned as a vector or matrix containing the group computations. When you specify multiple methods, `groupsummary` horizontally concatenates the computations in the order that they were listed.

BG — Groups
column vector | cell array of column vectors

Groups for array input data, returned as a column vector or cell array of column vectors each corresponding to a grouping vector.

When you provide more than one grouping vector, `BG` is a cell array containing column vectors of equal length. The group information can be found by looking at the elements rowwise across all vectors in `BG`. Each group maps to the corresponding row of the output array `B`.

BC — Group counts
column vector

Group counts for array input data, returned as a column vector containing the number of elements in each group. The length of `BC` is the same as the length of the group column vectors returned in `BG`.

Tips

- When making many calls to `groupsummary`, consider converting grouping variables to type `categorical` or `logical` when possible for improved performance. For example, if you have a grouping variable of type `char` (such as `Gender` with elements `'Male'` and `'Female'`), you can convert it to a categorical value using the command `categorical(Gender)`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- If `A` and `groupvars` are both tall matrices, then they must have the same number of rows.
- If the first input is a tall matrix, then `groupvars` can be a cell array containing tall grouping vectors.
- The `groupvars` and `datavars` arguments do not support function handles.
- The `'IncludeEmptyGroups'` name-value pair is not supported.
- The `'median'` and `'mode'` methods are not supported.
- For tall datetime arrays, the `'std'` method is not supported.
- If the `method` argument is a function handle, then it must be a valid input for `splitapply` operating on a tall array.
- The order of the groups might be different compared to in-memory `groupsummary` calculations.
- When grouping by discretized datetime arrays, the categorical group names are different compared to in-memory `groupsummary` calculations.
- For table inputs, all variables being operated on must be tall vectors.

For more information, see “Tall Arrays”.

See Also

`discretize` | `findgroups` | `groupcounts` | `groupfilter` | `grouptransform` | `rowfun` | `splitapply` | `varfun`

Introduced in R2018a

grouptransform

Transform by group

Syntax

```
G = grouptransform(T,groupvars,method)
G = grouptransform(T,groupvars,groupbins,method)
G = grouptransform( ___ ,datavars)
G = grouptransform( ___ ,Name,Value)

B = grouptransform(A,groupvars,method)
B = grouptransform(A,groupvars,groupbins,method)
B = grouptransform( ___ ,Name,Value)
[B,BG] = grouptransform(A, ___ )
```

Description

`G = grouptransform(T,groupvars,method)` transforms the data in a table or timetable using the computation in `method`, grouping by the variables specified in `groupvars`. The output `G` is a table or timetable containing the transformed data in place of the nongrouping variables from `T`. For example, `G = grouptransform(T,'Gender','norm')` normalizes the data in `T` by gender using the 2-norm.

`G = grouptransform(T,groupvars,groupbins,method)` bins the transformed data according to `groupbins`, placing the groups at the end of the output table as additional variables. For example, `G = grouptransform(T,'SaleDate','year','rescale')` scales the data in `T` to the range [0,1] and bins by sale year.

`G = grouptransform( ___ ,datavars)` specifies the table variables to transform for either of the previous syntaxes.

`G = grouptransform( ___ ,Name,Value)` specifies additional grouping properties using one or more name-value pairs. For example, `G = grouptransform(T,'Temp','linearfill','ReplaceValues',false)` appends

the filled data as an additional variable of T instead of replacing the nongrouping variables.

`B = grouptransform(A,groupvars,method)` returns the transformed data according to the unique combinations of grouping vectors in `groupvars` when A is a vector or matrix. `groupvars` can be a column vector, matrix, or cell array of column vectors.

`B = grouptransform(A,groupvars,groupbins,method)` bins the transformed data according to `groupbins`.

`B = grouptransform( ___ ,Name,Value)` specifies additional grouping properties using one or more name-value pairs for either of the previous array syntaxes.

`[B,BG] = grouptransform(A, ___ )` also returns the rows of the grouping vectors.

Examples

Fill Missing Data by Group

Create a timetable containing a progress status for 3 teams.

```
timeStamp = days([1 1 1 2 2 2 3 3 3]');
teamNumber = [1 2 3 1 2 3 1 2 3]';
percentComplete = [14.2 28.1 11.5 NaN NaN 19.3 46.1 51.2 30.3]';
T = timetable(timeStamp,teamNumber,percentComplete)
```

```
T=9×3 timetable
    timeStamp    teamNumber    percentComplete
    _________    __________    _______________

    1 day            1              14.2
    1 day            2              28.1
    1 day            3              11.5
    2 days           1               NaN
    2 days           2               NaN
    2 days           3              19.3
    3 days           1              46.1
    3 days           2              51.2
    3 days           3              30.3
```

Fill missing status percentages (NaN) for each group using linear interpolation.

```
G = grouptransform(T,'teamNumber','linearfill','percentComplete')
```

```
G=9×3 timetable
    timeStamp    teamNumber    percentComplete
    _________    __________    _______________

    1 day            1               14.2
    1 day            2               28.1
    1 day            3               11.5
    2 days           1              30.15
    2 days           2              39.65
    2 days           3               19.3
    3 days           1               46.1
    3 days           2               51.2
    3 days           3               30.3
```

To append the filled data to the original table instead of replacing the `percentComplete` variable, use the `'ReplaceValues'` parameter.

```
Gappend = grouptransform(T,'teamNumber','linearfill','percentComplete','ReplaceValues',
```

```
Gappend=9×4 timetable
    timeStamp    teamNumber    percentComplete    linearfill_percentComplete
    _________    __________    _______________    __________________________

    1 day            1              14.2                      14.2
    1 day            2              28.1                      28.1
    1 day            3              11.5                      11.5
    2 days           1               NaN                     30.15
    2 days           2               NaN                     39.65
    2 days           3              19.3                      19.3
    3 days           1              46.1                      46.1
    3 days           2              51.2                      51.2
    3 days           3              30.3                      30.3
```

Normalize Data by Day Name

Create a table of dates and corresponding profits.

```
timeStamps = datetime([2017 3 4; 2017 3 2; 2017 3 15; 2017 3 10;...
                       2017 3 14; 2017 3 31; 2017 3 25;...
```

```
                        2017 3 29; 2017 3 21; 2017 3 18]);
profit = [2032 3071 1185 2587 1998 2899 3112 909 2619 3085]';
T = table(timeStamps,profit)
```

```
T=10×2 table
    timeStamps     profit
    ___________    ______

    04-Mar-2017     2032
    02-Mar-2017     3071
    15-Mar-2017     1185
    10-Mar-2017     2587
    14-Mar-2017     1998
    31-Mar-2017     2899
    25-Mar-2017     3112
    29-Mar-2017      909
    21-Mar-2017     2619
    18-Mar-2017     3085
```

Grouping by day name, normalize the profits by the 2-norm.

```
G = grouptransform(T,'timeStamps','dayname','norm')
```

```
G=10×3 table
    timeStamps     profit     dayname_timeStamps
    ___________    _______    __________________

    04-Mar-2017    0.42069        Saturday
    02-Mar-2017          1        Thursday
    15-Mar-2017    0.79344        Wednesday
    10-Mar-2017    0.66582        Friday
    14-Mar-2017    0.60654        Tuesday
    31-Mar-2017    0.74612        Friday
    25-Mar-2017    0.64428        Saturday
    29-Mar-2017    0.60864        Wednesday
    21-Mar-2017    0.79506        Tuesday
    18-Mar-2017    0.63869        Saturday
```

Group Operations with Vector Data

Create a vector of dates and a vector of corresponding profit values.

```
timeStamps = datetime([2017 3 4; 2017 3 2; 2017 3 15; 2017 3 10; ...
                       2017 3 14; 2017 3 31; 2017 3 25; ...
                       2017 3 29; 2017 3 21; 2017 3 18]);
profit = [2032 3071 1185 2587 1998 2899 3112 909 2619 3085]';
```

Grouping by day of the week, normalize the profit values by the 2-norm. Display the transformed data and which group it corresponds to.

```
[normDailyProfit,dayOfWeek] = grouptransform(profit,timeStamps,'dayname','norm')
```

```
normDailyProfit = 10×1

    0.4207
    1.0000
    0.7934
    0.6658
    0.6065
    0.7461
    0.6443
    0.6086
    0.7951
    0.6387


dayOfWeek = 10x1 categorical array
     Saturday
     Thursday
     Wednesday
     Friday
     Tuesday
     Friday
     Saturday
     Wednesday
     Tuesday
     Saturday
```

Input Arguments

T — Input data
table | timetable

Input data, specified as a table or timetable.

A — Input array
vector | matrix

Input array, specified as a vector or matrix.

groupvars — Grouping variables or vectors
scalar | vector | matrix | cell array | function handle

Grouping variables or vectors, specified as a scalar, vector, matrix, cell array, or function handle.

For table or timetable input data, `groupvars` indicates which columns to use to compute the groups, and can be one of the following:

- A character vector or string scalar specifying a single table variable name
- A cell array of character vectors or a string array, where each element is a table variable name
- A vector of table variable indices
- A logical vector whose elements each correspond to a table variable, where `true` includes the corresponding variable and `false` excludes it
- A function handle that takes a table variable as input and returns a logical scalar

For array input, `groupvars` can be either a column vector with the same number of rows as `A` or a group of column vectors arranged in a matrix or cell array.

Example: `'Age'`

Example: `{'Height','Weight'}`

method — Computation method
`'zscore'` | `'norm'` | `'meancenter'` | `'rescale'` | `'meanfill'` | `'linearfill'` | function handle

Computation method, specified as one of the following:

Method	Description
`'zscore'`	Normalize data to have mean 0 and standard deviation 1
`'norm'`	Normalize data by 2-norm
`'meancenter'`	Normalize data to have mean 0

Method	Description
`'rescale'`	Rescale range to [0,1]
`'meanfill'`	Fill missing values with the mean of the group data
`'linearfill'`	Fill missing values by linear interpolation of nonmissing group data

You can also specify a function handle that returns a scalar or a column vector with the same number of rows as the input data. If the function returns a scalar, `grouptransform` repeats the scalar for each element of the transformed output.

`datavars` — Data variables

scalar | vector | cell array | function handle

Data variables for table or timetable input, specified as a scalar, vector, cell array, or function handle. `datavars` indicates which columns of the input table to apply the methods to, and can be one of the following options:

- A character vector or string scalar specifying a single table variable name
- A cell array of character vectors or string array where each element is a table variable name
- A vector of table variable indices
- A logical vector whose elements each correspond to a table variable, where `true` includes the corresponding variable and `false` excludes it
- A function handle that takes a table variable as input and returns a logical scalar

When `datavars` is not specified, `grouptransform` applies the computation to each nongrouping variable.

Example: `'Profit'`

Example: `{'Income','Expenses'}`

Example: `@isnumeric`

`groupbins` — Binning scheme

`'none'` (default) | character vector | scalar | vector | cell array

Binning scheme, specified as one of the following options:

- `'none'`, indicating no binning
- A list of bin edges, specified as a numeric vector, or a `datetime` vector for `datetime` grouping variables
- A number of bins, specified as an integer scalar
- A time duration, specified as a scalar of type `duration` or `calendarDuration` indicating bin widths (for `datetime` or `duration` grouping variables only)
- A cell array listing binning rules for each grouping variable
- A time bin for `datetime` and `duration` grouping variables only, specified as one of the following character vectors:

Value	Description	Data Type
`'second'`	Each bin is 1 second.	`datetime` and `duration`
`'minute'`	Each bin is 1 minute.	`datetime` and `duration`
`'hour'`	Each bin is 1 hour.	`datetime` and `duration`
`'day'`	Each bin is 1 calendar day. This value accounts for Daylight Saving Time shifts.	`datetime` and `duration`
`'week'`	Each bin is 1 calendar week.	`datetime` only
`'month'`	Each bin is 1 calendar month.	`datetime` only
`'quarter'`	Each bin is 1 calendar quarter.	`datetime` only
`'year'`	Each bin is 1 calendar year. This value accounts for leap days.	`datetime` and `duration`
`'decade'`	Each bin is 1 decade (10 calendar years).	`datetime` only
`'century'`	Each bin is 1 century (100 calendar years).	`datetime` only
`'secondofminute'`	Bins are seconds from 0 to 59.	`datetime` only

Value	Description	Data Type
`'minuteofhour'`	Bins are minutes from 0 to 59.	`datetime` only
`'hourofday'`	Bins are hours from 0 to 23.	`datetime` only
`'dayofweek'`	Bins are days from 1 to 7. The first day of the week is Sunday.	`datetime` only
`'dayname'`	Bins are full day names such as `'Sunday'`.	`datetime` only
`'dayofmonth'`	Bins are days from 1 to 31.	`datetime` only
`'dayofyear'`	Bins are days from 1 to 366.	`datetime` only
`'weekofmonth'`	Bins are weeks from 1 to 6.	`datetime` only
`'weekofyear'`	Bins are weeks from 1 to 54.	`datetime` only
`'monthname'`	Bins are full month names such as `'January'`.	`datetime` only
`'monthofyear'`	Bins are months from 1 to 12.	`datetime` only
`'quarterofyear'`	Bins are quarters from 1 to 4.	`datetime` only

When multiple grouping variables are specified, you can provide a single binning rule that is applied to all grouping variables, or a cell array containing a binning method for each grouping variable such as `{'none',[0 2 4 Inf]}`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `G = grouptransform(T,groupvars,groupbins,'zscore','IncludedEdge','right')`

`IncludedEdge` — Included bin edge
`'left'` (default) | `'right'`

Included bin edge, specified as either `'left'` or `'right'`, indicating which end of the bin interval is inclusive.

This name-value pair can only be specified when `groupbins` is specified, and the value is applied to all binning schemes for all grouping variables or vectors.

`ReplaceValues` — Transform placement indicator
`true` or `1` (default) | `false` or `0`

Transform placement indicator, specified as a numeric or logical `1` (`true`) or `0` (`false`). When the parameter value is `true`, `grouptransform` outputs a table or array with the transformed data in place of the nongrouping variables or vectors from the input. When the parameter value is `false`, `grouptransform` appends the transformed data as additional matrix columns or table variables to the input data.

Output Arguments

`G` — Output table
table

Output table, returned as a table containing the transformed data for each group.

`B` — Output array
vector | matrix

Output array, returned as a vector or matrix containing the transformed data.

`BG` — Grouping vectors
column vector | cell array of column vectors

Grouping vectors for array input data, returned as a column vector or cell array of column vectors.

Tips

- When making many calls to `grouptransform`, consider converting grouping variables to type `categorical` or `logical` when possible for improved performance. For example, if you have a grouping variable of type `char` (such as `Gender` with elements `'Male'` and `'Female'`), you can convert it to a categorical value using the command `categorical(Gender)`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- If `A` and `groupvars` are both tall matrices, then they must have the same number of rows.
- If the first input is a tall matrix, then `groupvars` can be a cell array containing tall grouping vectors.
- The `groupvars` and `datavars` arguments do not support function handles.
- If the `method` argument is a function handle, then it must be a valid input for `splitapply` operating on a tall array.
- When grouping by discretized datetime arrays, the categorical group names are different compared to in-memory `grouptransform` calculations.

For more information, see “Tall Arrays”.

See Also

`discretize` | `findgroups` | `groupcounts` | `groupfilter` | `groupsummary` | `rowfun` | `splitapply` | `varfun`

Introduced in R2018b

gsvd

Generalized singular value decomposition

Syntax

```
[U,V,X,C,S] = gsvd(A,B)
[U,V,X,C,S] = gsvd(A,B,0)
sigma = gsvd(A,B)
```

Description

`[U,V,X,C,S] = gsvd(A,B)` returns unitary matrices `U` and `V`, a (usually) square matrix `X`, and nonnegative diagonal matrices `C` and `S` so that

```
A = U*C*X'
B = V*S*X'
C'*C + S'*S = I
```

`A` and `B` must have the same number of columns, but may have different numbers of rows. If `A` is `m`-by-`p` and `B` is `n`-by-`p`, then `U` is `m`-by-`m`, `V` is `n`-by-`n`, `X` is `p`-by-`q`, `C` is `m`-by-`q` and `S` is `n`-by-`q`, where `q = min(m+n,p)`.

The nonzero elements of `S` are always on its main diagonal. The nonzero elements of `C` are on the diagonal `diag(C,max(0,q-m))`. If `m >= q`, this is the main diagonal of `C`.

`[U,V,X,C,S] = gsvd(A,B,0)`, where `A` is `m`-by-`p` and `B` is `n`-by-`p`, produces the “economy-sized“ decomposition where the resulting `U` and `V` have at most `p` columns, and `C` and `S` have at most `p` rows. The generalized singular values are `diag(C)./diag(S)` so long as `m >= p` and `n >= p`.

If `A` is `m`-by-`p` and `B` is `n`-by-`p`, then `U` is `m`-by-`min(q,m)`, `V` is `n`-by-`min(q,n)`, `X` is `p`-by-`q`, `C` is `min(q,m)`-by-`q` and `S` is `min(q,n)`-by-`q`, where `q = min(m+n,p)`.

`sigma = gsvd(A,B)` returns the vector of generalized singular values, `sqrt(diag(C'*C)./diag(S'*S))`. When `B` is square and nonsingular, the generalized singular values, `gsvd(A,B)`, correspond to the ordinary singular values, `svd(A/B)`, but they are sorted in the opposite order. Their reciprocals are `gsvd(B,A)`.

The vector `sigma` has length `q` and is in non-decreasing order.

Examples

Example 1

The matrices have at least as many rows as columns.

```
A = reshape(1:15,5,3)
B = magic(3)
A =
         1     6    11
         2     7    12
         3     8    13
         4     9    14
         5    10    15
B =
         8     1     6
         3     5     7
         4     9     2
```

The statement

```
[U,V,X,C,S] = gsvd(A,B)
```

produces a 5-by-5 orthogonal `U`, a 3-by-3 orthogonal `V`, a 3-by-3 nonsingular `X`,

```
X =
        2.8284   -9.3761   -6.9346
       -5.6569   -8.3071  -18.3301
        2.8284   -7.2381  -29.7256
```

and

```
C =
        0.0000         0         0
             0    0.3155         0
             0         0    0.9807
             0         0         0
             0         0         0
S =
        1.0000         0         0
```

```
             0    0.9489         0
             0         0    0.1957
```

Since A is rank deficient, the first diagonal element of C is zero.

The economy sized decomposition,

```
[U,V,X,C,S] = gsvd(A,B,0)
```

produces a 5-by-3 matrix U and a 3-by-3 matrix C.

```
U =
        0.5700   -0.6457   -0.4279
       -0.7455   -0.3296   -0.4375
       -0.1702   -0.0135   -0.4470
        0.2966    0.3026   -0.4566
        0.0490    0.6187   -0.4661

C =
        0.0000         0         0
             0    0.3155         0
             0         0    0.9807
```

The other three matrices, V, X, and S are the same as those obtained with the full decomposition.

The generalized singular values are the ratios of the diagonal elements of C and S.

```
sigma = gsvd(A,B)
sigma =
        0.0000
        0.3325
        5.0123
```

These values are a reordering of the ordinary singular values

```
svd(A/B)
ans =
        5.0123
        0.3325
        0.0000
```

Example 2

The matrices have at least as many columns as rows.

```
A = reshape(1:15,3,5)
B = magic(5)
A =

     1     4     7    10    13
     2     5     8    11    14
     3     6     9    12    15

B =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

The statement

```
[U,V,X,C,S] = gsvd(A,B)
```

produces a 3-by-3 orthogonal `U`, a 5-by-5 orthogonal `V`, a 5-by-5 nonsingular `X` and

```
C =

         0         0    0.0000         0         0
         0         0         0    0.0439         0
         0         0         0         0    0.7432

S =

    1.0000         0         0         0         0
         0    1.0000         0         0         0
         0         0    1.0000         0         0
         0         0         0    0.9990         0
         0         0         0         0    0.6690
```

In this situation, the nonzero diagonal of `C` is `diag(C,2)`. The generalized singular values include three zeros.

```
sigma = gsvd(A,B)
sigma =

         0
         0
    0.0000
```

```
    0.0439
    1.1109
```

Reversing the roles of A and B reciprocates these values, producing two infinities.

```
gsvd(B,A)
ans =

   1.0e+16 *

    0.0000
    0.0000
    8.8252
       Inf
       Inf
```

Tips

- In this formulation of the `gsvd`, no assumptions are made about the individual ranks of A or B. The matrix X has full rank if and only if the matrix `[A;B]` has full rank. In fact, `svd(X)` and `cond(X)` are equal to `svd([A;B])` and `cond([A;B])`. Other formulations, eg. G. Golub and C. Van Loan [1], require that `null(A)` and `null(B)` do not overlap and replace X by `inv(X)` or `inv(X')`.

  Note, however, that when `null(A)` and `null(B)` do overlap, the nonzero elements of C and S are not uniquely determined.

Algorithms

The generalized singular value decomposition uses the C-S decomposition described in [1], as well as the built-in `svd` and `qr` functions. The C-S decomposition is implemented in a local function in the `gsvd` program file.

References

[1] Golub, Gene H. and Charles Van Loan, *Matrix Computations*, Third Edition, Johns Hopkins University Press, Baltimore, 1996

See Also

qr | svd

Introduced before R2006a

gt, >

Determine greater than

Syntax

```
A > B
gt(A,B)
```

Description

`A > B` returns a logical array with elements set to logical 1 (`true`) where `A` is greater than `B`; otherwise, the element is logical `0` (`false`). The test compares only the real part of numeric arrays. `gt` returns logical `0` (`false`) where `A` or `B` have NaN or undefined categorical elements.

`gt(A,B)` is an alternate way to execute `A > B`, but is rarely used. It enables operator overloading for classes.

Examples

Test Vector Elements

Determine if vector elements are greater than a given value.

Create a numeric vector.

```
A = [1 12 18 7 9 11 2 15];
```

Test the vector for elements that are greater than `10`.

```
A > 10
```

```
ans = 1x8 logical array
```

```
   0   1   1   0   0   1   0   1
```

The result is a vector with values of logical `1` (`true`) where the elements of `A` satisfy the expression.

Use the vector of logical values as an index to view the values in `A` that are greater than `10`.

```
A(A > 10)
```

```
ans = 1×4

    12    18    11    15
```

The result is a subset of the elements in `A`.

Replace Elements of Matrix

Create a matrix.

```
A = magic(4)
```

```
A = 4×4

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Replace all values greater than `9` with the value `10`.

```
A(A > 9) = 10
```

```
A = 4×4

    10     2     3    10
     5    10    10     8
     9     7     6    10
     4    10    10     1
```

The result is a new matrix whose largest element is 10.

Compare Values in Categorical Array

Create an ordinal categorical array.

```
A = categorical({'large' 'medium' 'small'; 'medium' ...
'small' 'large'},{'small' 'medium' 'large'},'Ordinal',1)
```

```
A = 2x3 categorical array
     large       medium      small
     medium      small       large
```

The array has three categories: 'small', 'medium', and 'large'.

Find all values greater than the category 'medium'.

```
A > 'medium'
```

```
ans = 2x3 logical array

   1   0   0
   0   0   1
```

A value of logical 1 (true) indicates a value greater than the category 'medium'.

Compare the rows of A.

```
A(1,:) > A(2,:)
```

```
ans = 1x3 logical array

   1   1   0
```

The function returns logical 1 (true) where the first row has a category value greater than the second row.

Test Complex Numbers

Create a vector of complex numbers.

```
A = [1+i 2-2i 1+3i 1-2i 5-i];
```

Find the values that are greater than 2.

```
A(A > 2)
```

```
ans = 5.0000 - 1.0000i
```

`gt` compares only the real part of the elements in `A`.

Use `abs` to find which elements are outside a radius of 2 from the origin.

```
A(abs(A) > 2)
```

```
ans = 1×4 complex

   2.0000 - 2.0000i   1.0000 + 3.0000i   1.0000 - 2.0000i   5.0000 - 1.0000i
```

The result has more elements since `abs` accounts for the imaginary part of the numbers.

Compare Dates

Create a vector of dates.

```
A = datetime([2014,05,01;2014,05,31])
```

```
A = 2x1 datetime array
   01-May-2014
   31-May-2014
```

Find the dates that occur after May 10, 2014.

```
A(A > '2014-05-10')
```

```
ans = datetime
   31-May-2014
```

Input Arguments

A, B — Operands
scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Numeric or string inputs A and B must either be the same size or have sizes that are compatible (for example, A is an M-by-N matrix and B is a scalar or 1-by-N row vector). For more information, see “Compatible Array Sizes for Basic Operations”.

You can compare numeric inputs of any type, and the comparison does not suffer loss of precision due to type conversion.

If A and B are categorical, datetime, or duration arrays, then they must be the same size unless one is a scalar.

- If one input is an ordinal categorical array, the other input can be an ordinal categorical array, a cell array of character vectors, or a single character vector. A single character vector expands into a cell array of character vectors of the same size as the other input. If both inputs are ordinal categorical arrays, they must have the same sets of categories, including their order. See “Compare Categorical Array Elements” for more details.
- If one input is a datetime array, the other input can be a datetime array, a character vector, or a cell array of character vectors.
- If one input is a duration array, the other input can be a duration array or a numeric array. The operator treats each numeric value as a number of standard 24-hour days.
- If one input is a string array, the other input can be a string array, a character vector, or a cell array of character vectors. The corresponding elements of A and B are compared lexicographically.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `categorical` | `datetime` | `duration`
Complex Number Support: Yes

Compatibility Considerations

Implicit expansion change affects arguments for operators

Behavior changed in R2016b

Starting in R2016b with the addition of implicit expansion, some combinations of arguments for basic operations that previously returned errors now produce results. For example, you previously could not add a row and a column vector, but those operands are now valid for addition. In other words, an expression like `[1 2] + [1; 2]` previously returned a size mismatch error, but now it executes.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a `try/catch` block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see “Compatible Array Sizes for Basic Operations”.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`eq` | `ge` | `le` | `lt` | `ne`

Topics

“Array Comparison with Relational Operators”
“Ordinal Categorical Arrays”
“MATLAB Operators and Special Characters”

Introduced before R2006a

gtext

Add text to figure using mouse

Syntax

```
gtext(str)
gtext(str,Name,Value)
```

```
t = gtext( ___ )
```

Description

`gtext(str)` inserts the text, `str`, at the location you select with the mouse. When you hover over the figure window, the pointer becomes a crosshair. `gtext` is waiting for you to select a location. Move the pointer to the location you want and either click the figure or press any key, except **Enter**.

`gtext(str,Name,Value)` specifies text properties using one or more name-value pair arguments. For example, `'FontSize',14` specifies a 14-point font.

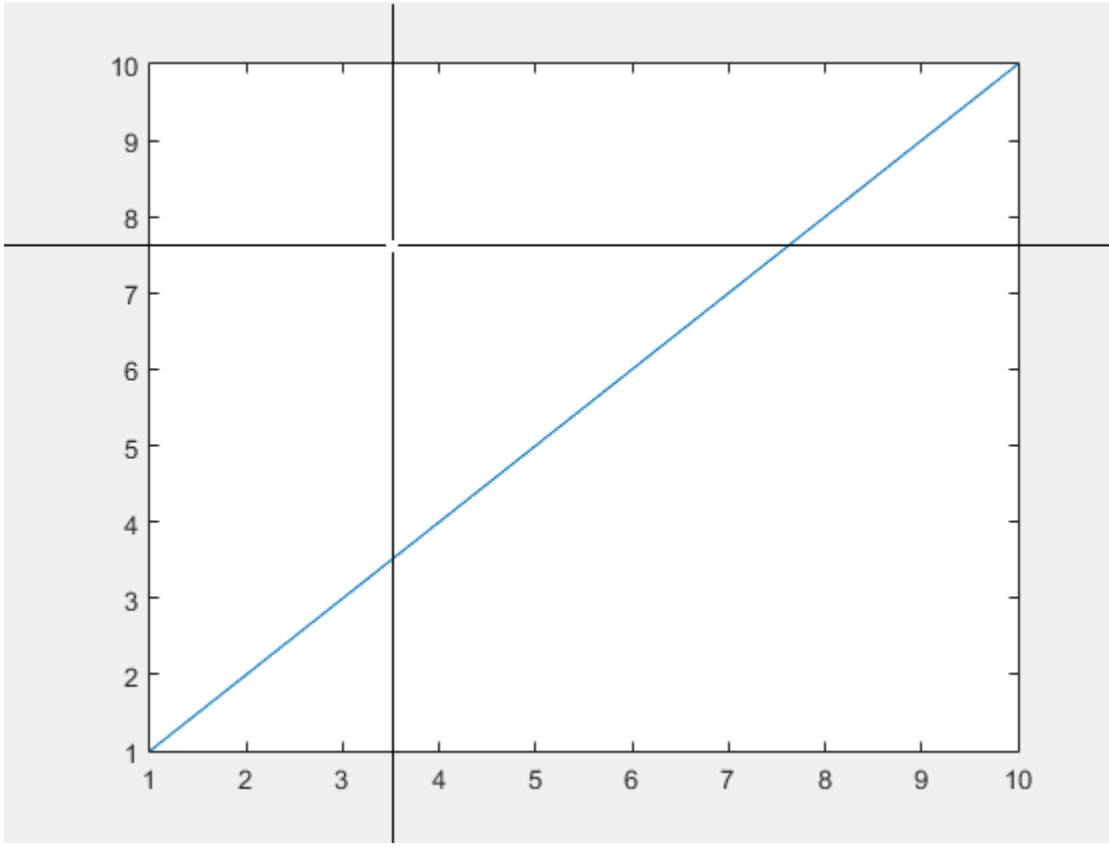
`t = gtext( ___ )` returns an array of text objects created by `gtext`. Use `t` to modify properties of the text objects after they are created. For a list of properties and descriptions, see Text. You can return an output argument using any of the arguments from the previous syntaxes.

Examples

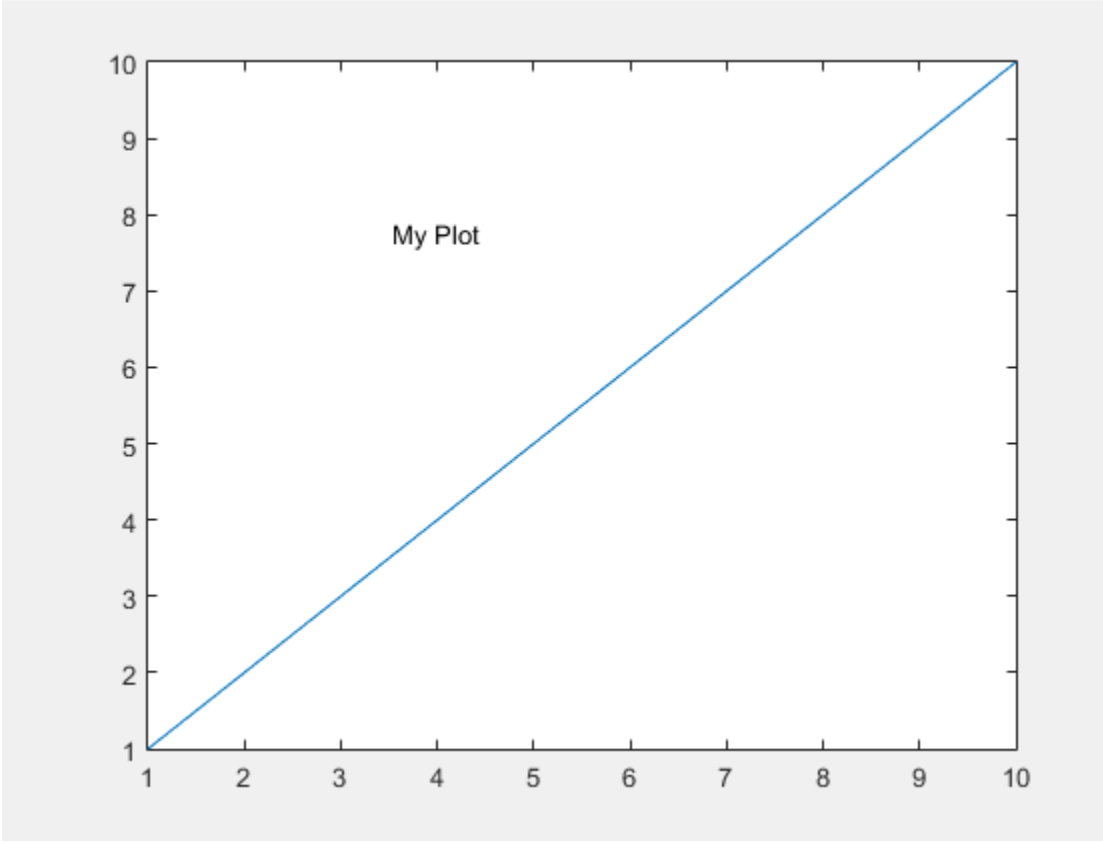
Add Text to Figure Using Mouse

Create a simple line plot and use `gtext` to add text to the figure using the mouse.

```
plot(1:10)
gtext('My Plot')
```



Click the figure to place the text at the selected location.



Specify Font Size and Color

Create a simple line plot and add text to the figure using the mouse. Use a red, 14-point font.

```
plot(1:10)
gtext('My Plot','Color','red','FontSize',14)
```

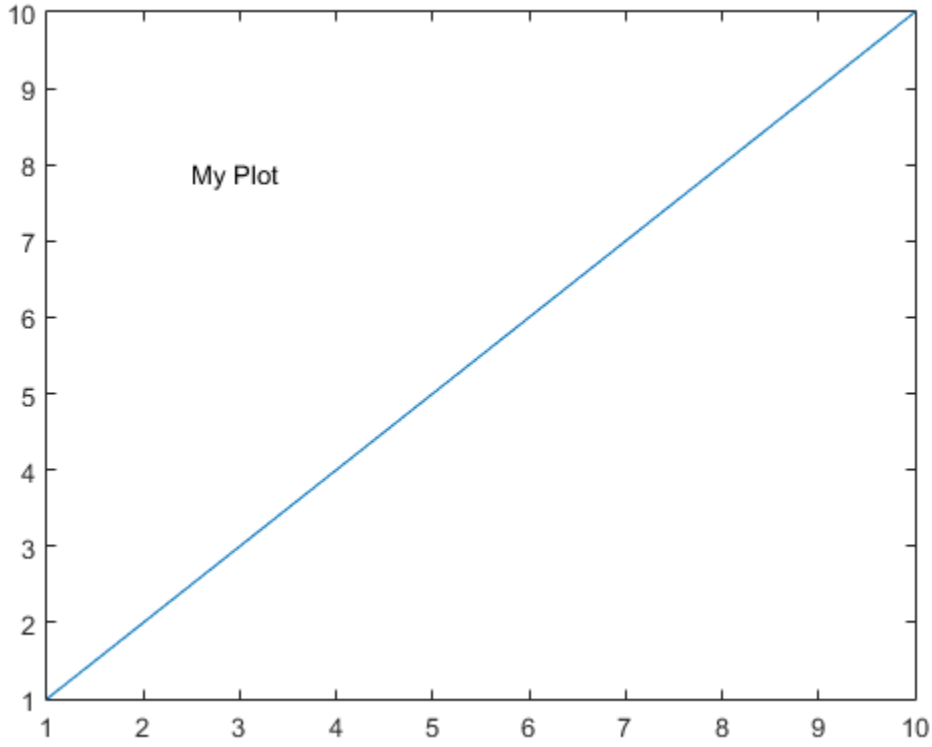
Click the figure where you want to display the text.

Modify Text After Creation

Create a simple line plot and add text to the figure using the mouse. Return the text object created, `t`.

```
plot(1:10)
t = gtext('My Plot')
```

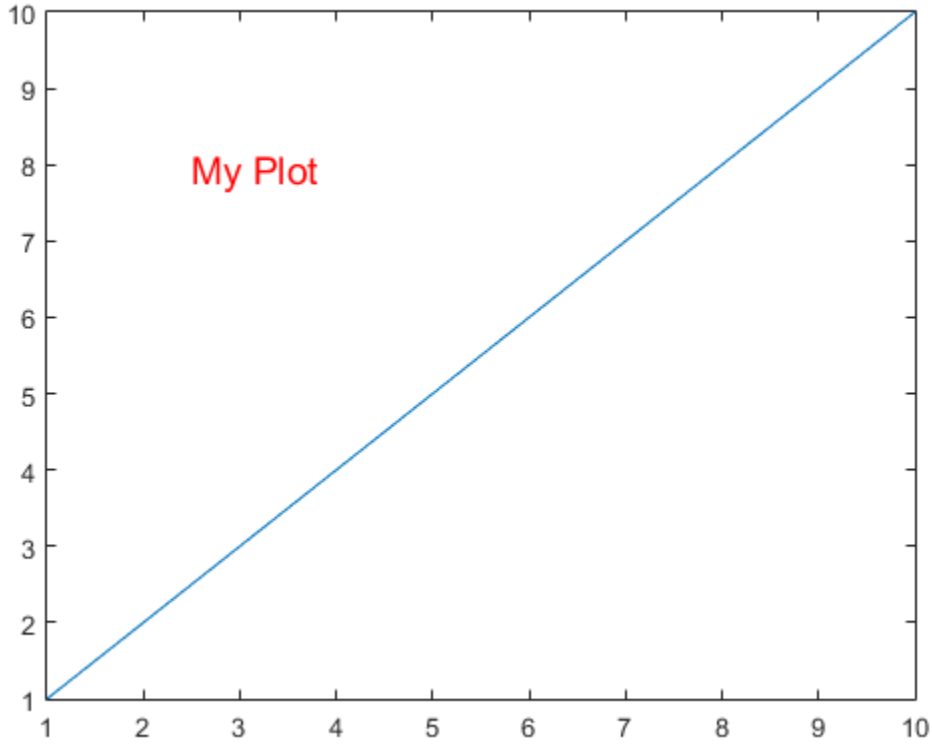
Click the figure to place the text and create the text object.




```
t =

  Text (My Plot) with properties:

                String: 'My Plot'
              FontSize: 10
            FontWeight: 'normal'
              FontName: 'Helvetica'
                 Color: [0 0 0]
  HorizontalAlignment: 'left'
              Position: [4.3906 5.3950 0]
                 Units: 'data'

  Show all properties
```

Use `t` to change the font size and color of the text by setting text properties. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead, such as `set(t,'Color','red')`.

```
t.Color = 'red';
t.FontSize = 14;
```



Input Arguments

`str` — Text to display

character vector | cell array of character vectors | character array | string array

Text to display, specified in one of these forms:

- Character vector — Display the text with one click, for example:

  ```
  gtext('my text')
  ```

- Cell array of character vectors — Display one row of the array with each click. For example, this code displays `first` with the first click and `second` with the second click.

  ```
  gtext({'first';'second'})
  ```

  To display multiline text, specify more than one character vector per row.

  ```
  gtext({'first','new line';'second','new line'})
  ```

- Character array — Display one row of the array with each click. Each row must contain the same number of characters, for example:

  ```
  gtext(['one';'two'])
  ```

- String array — Display one row of the array with each click. For example:

  ```
  gtext(["one";"two"])
  ```

To include special characters, such as superscripts, subscripts, Greek letters, or mathematical symbols, use TeX markup. For a list of supported markup, see the `Interpreter` property.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `text(.5,.5,'my text','FontSize',14,'Color','red')`

The properties listed here are only a subset. For a full list, see Text.

Note You cannot specify the `Position` text property as a name-value pair during text creation. `gtext` ignores the specified value.

`FontSize` — Font size
scalar value greater than zero

Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals `1/72` inch. To change the font units, use the `FontUnits` property.

Example: 12

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`FontWeight` — Character thickness
`'normal'` (default) | `'bold'`

Character thickness, specified as `'normal'` or `'bold'`.

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.

`FontName` — Font name
supported font name | `'FixedWidth'`

Font name, specified as a supported font name or `'FixedWidth'`. To display and print text properly, you must choose a font that your system supports. The default font depends on your operating system and locale.

To use a fixed-width font that looks good in any locale, use `'FixedWidth'`. The fixed-width font relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`Color` — Text color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Text color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`HorizontalAlignment` — Horizontal alignment of text with respect to position point

`'left'` (default) | `'center'` | `'right'`

Horizontal alignment of the text with respect to the `x` value in the `Position` property, specified as one of the values in this table. The vertical line indicates where the `x` value lies in relation to the text.

Value	Result
`'left'` (default)	
`'center'`	
`'right'`	

`Interpreter` — Interpretation of text characters

`'tex'` (default) | `'latex'` | `'none'`

Interpretation of text characters, specified as one of these values:

- `'tex'` — Interpret characters using a subset of TeX markup.
- `'latex'` — Interpret characters using LaTeX markup.
- `'none'` — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text.

Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they modify only the next character or the characters within the curly

braces. When you set the `Interpreter` property to `'tex'`, the supported modifiers are as follows.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{`*`specifier`*`}`	Font name — Replace *`specifier`* with the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{`*`specifier`*`}`	Font size —Replace *`specifier`* with a numeric scalar value in point units.	`'\fontsize{15} text'`
`\color{`*`specifier`*`}`	Font color — Replace *`specifier`* with one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Replace *`specifier`* with a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\alpha	α	\upsilon	υ	\sim	~
\angle	∠	\phi	ϕ	\leq	≤
\ast	*	\chi	χ	\infty	∞
\beta	β	\psi	ψ	\clubsuit	♣
\gamma	γ	\omega	ω	\diamondsuit	♦
\delta	δ	\Gamma	Γ	\heartsuit	♥
\epsilon	ϵ	\Delta	Δ	\spadesuit	♠
\zeta	ζ	\Theta	Θ	\leftrightarrow	↔
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	µ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\otimes`	⊗	`\subseteq`	⊆	`\oslash`	∅
`\cap`	∩	`\in`	∈	`\supseteq`	⊇
`\supset`	⊃	`\lceil`	⌈	`\subset`	⊂
`\int`	∫	`\cdot`	·	`\o`	o
`\rfloor`	⌋	`\neg`	¬	`\nabla`	∇
`\lfloor`	⌊	`\times`	x	`\ldots`	...
`\perp`	⊥	`\surd`	√	`\prime`	´
`\wedge`	∧	`\varpi`	ϖ	`\0`	∅
`\rceil`	⌉	`\rangle`	〉	`\mid`	|
`\vee`	∨	`\langle`	〈	`\copyright`	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. Use dollar symbols around the text, for example, use `'$\int_1^{20} x^2 dx$'` for inline mode or `'$$ \int_1^{20} x^2 dx$$'` for display mode.

The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup.

The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters. For multiline text, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project website at https://www.latex-project.org/.

Algorithms

`gtext` uses the `ginput` and `text` functions.

See Also

Functions
`annotation` | `ginput` | `text`

Properties
Text

Topics
"Add Text to Chart"

Introduced before R2006a

guidata

Store or retrieve UI data

Note

Use this function only with GUIDE, or with apps created using the `figure` function.

Syntax

```
guidata(obj,data)
data = guidata(obj)
```

Description

`guidata(obj,data)` stores the specified data in the application data of `obj` if it is a figure, or the parent figure of `obj` if it is another component. For more information, see "How guidata Manages Data" on page 1-5732.

`data = guidata(obj)` returns previously stored data, or an empty matrix if nothing is stored.

Examples

Store Data in Programmatic UI

Create a programmatic UI that stores and retrieves counter data when you click on it.

First, create a program file called `progCounter.m`. Within the program file:

- Create a figure.
- Create a structure with a field value initialized to zero.

- Store the data in the figure.
- Define a callback function that retrieves the data from the figure, changes the data, and stores the new data in the figure again.

Run the program and click inside the figure. The updated data appears in the Command Window.

```
f = figure;
data.numberOfClicks = 0;
guidata(f,data)
f.ButtonDownFcn = @My_Callback;

function My_Callback(src,event)
data = guidata(src);
data.numberOfClicks = data.numberOfClicks + 1;
guidata(src,data)
data
end
```

```
data =

  struct with fields:

    numberOfClicks: 1
```

Using `guidata` in GUIDE

Create a button in GUIDE, and store and access data when the button is pressed. To do this, first add a field to the `handles` structure and use `guidata` to update and manage it. (Make sure to add the data to `handles` rather than overwriting it. For more information, see “How GUIDE Uses guidata” on page 1-5732.) Then, configure a callback that uses `guidata` to access and store data when the button is pressed.

To do this, first, open GUIDE and add a push button to the layout area. Then, click on the Editor icon to open the program file for editing. Find the `OpeningFcn` that was automatically created by GUIDE and add a new field to `handles` called `numberOfClicks`.

```
% --- Executes just before countClicks is made visible.
function countClicks_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
```

```
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to countClicks (see VARARGIN)

% Choose default command line output for countClicks
handles.output = hObject;

handles.numberOfClicks = 0;

% Update handles structure
guidata(hObject, handles);
```

Next, find the push button callback function that GUIDE created. Program it to access the data stored in `handles`, modify it, save the changed data to the structure, and display the new data in the Command Window. Notice that in GUIDE you use `hObject`, in place of `src`, to refer to the object whose callback is executing.

```
% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

handles.numberOfClicks = handles.numberOfClicks + 1;
guidata(hObject,handles)
data = guidata(hObject)
```

```
>> countClicks

data =

  struct with fields:

           figure1: [1×1 Figure]
       pushbutton1: [1×1 UIControl]
            output: [1×1 Figure]
    numberOfClicks: 1
```

Input Arguments

`obj` — Graphics object

graphics object

Graphics object, such as a `Figure`, `Axes`, `Illustration`, or UI object. Use this argument to specify the figure that stores `data`. If the specified object is not a figure, then the parent figure of the object will be used to store `data`.

`data` — Data
any MATLAB data

Data to store in the figure, specified as any MATLAB data. Typically, `data` is specified as a structure, which enables you to add new fields as needed. For example, create a data structure with a field called `Category`, store the data from the field in the structure, and display the stored data in the Command Window:

```
data.Category = 'Projected Growth';
guidata(gcf,data);
data = guidata(gcf)
```

Algorithms

How `guidata` Manages Data

`guidata` can manage only one variable per parent figure at any time. Subsequent calls to `guidata(obj,data)` overwrite the previously stored data. Store additional data by creating a structure with multiple fields.

How GUIDE Uses `guidata`

GUIDE uses `guidata` to store and maintain the structure called `handles`. In a GUIDE code file, do not overwrite the `handles` structure or your program will no longer work. If you need to store other data, you can do so by adding new fields to the `handles` structure.

See Also

`getappdata` | `guide` | `guihandles` | `setappdata` | `struct`

Introduced before R2006a

guide

(To be removed) Create or edit UI file in GUIDE

Note The GUIDE environment will be removed in a future release. Use “App Designer” instead. For more information, see “Compatibility Considerations”.

Syntax

```
guide
guide(filename)
guide(figs)
```

Description

`guide` opens GUIDE, a UI design environment. From the Quick Start dialog box, you can choose to open a previously created UI or create a new one using one of the templates.

`guide(filename)` opens the specified MATLAB figure file for editing in GUIDE. If the figure file is not on the MATLAB path, specify the full path. Only one `filename` can be opened at a time.

`guide(figs)` opens each of the `Figure` objects in `figs` in a separate copy of the GUIDE design environment. Use this syntax if you want to edit one or more preexisting figures in GUIDE that have been saved to variables.

Examples

Open Figure in GUIDE

Open an existing figure from your MATLAB path.

```
guide("feedback.fig")
```

Input Arguments

`filename` — File name or path of saved figure

character vector | string scalar

File name or path of saved figure, specified as a character vector or string scalar. Specify the full path if the figure is not on your MATLAB path. Including `.fig` in the file name is optional.

Example: `guide("output.fig")`

Example: `guide('testModule')`

Example: `guide("C:\Documents\Controls\power.fig")`

`figs` — Existing `Figure` objects

single `Figure` object | array of `Figure` objects

Existing `Figure` objects, specified as a single `Figure` object or an array of `Figure` objects. Use this argument to open one or more previously saved figures in separate instances of GUIDE. GUIDE creates figures with the `figure` function.

Example: `guide([f1 f2]);`

Compatibility Considerations

GUIDE will be removed

Not recommended starting in R2019b

The GUIDE environment and the `guide` function will be removed in a future release.

After GUIDE is removed, existing GUIDE apps will continue to run in MATLAB but will not be editable using the drag-and-drop environment in GUIDE. To continue editing an existing GUIDE app and help maintain its compatibility with future MATLAB releases, use one of the suggested migration strategies listed in the table.

App Development	Migration Strategy	How to Migrate
Frequent or ongoing development	Migrate your app to App Designer	Use the GUIDE to App Designer Migration Tool for MATLAB on `mathworks.com`
Minimal or occasional editing	Export your app to a single MATLAB file to manage your app layout and code using MATLAB functions	Open the app in GUIDE and select **File** > **Export to MATLAB-file**

App Designer is the recommended app development environment in MATLAB. To create new apps, use “App Designer” and the `appdesigner` function instead.

To learn more about migrating apps, see “GUIDE Migration Strategies”.

See Also

`appdesigner` | `uifigure`

Topics

“Create and Run a Simple App Using App Designer”
“App Designer Components”
“GUIDE Migration Strategies”
“Displaying Graphics in App Designer”

Introduced before R2006a

guihandles

Create structure containing all child objects of `Figure`

Note Use this function only with GUIDE, or with apps created using the `figure` function. App Designer is the recommended environment for building apps. For more information, see “GUIDE Migration Strategies”.

Syntax

```
handles = guihandles(object_handle)
handles = guihandles
```

Description

`handles = guihandles(object_handle)` returns a structure containing all child objects in a `Figure`. The `object_handle` argument can be a `Figure` object or any child component of a `Figure`. Any UI or graphics components you placed in the `Figure` are listed with field names that match their `Tag` property.

The `handles` structure typically contains one field for every child object, but there are some exceptions:

- Objects that have an empty `Tag` property are not listed.
- Objects with hidden handles are listed.
- When multiple objects have the same `Tag` property, that field in the structure contains a vector of objects.

`handles = guihandles` returns a structure containing all child objects for the current `Figure`.

See Also

`getappdata` | `guidata` | `guide` | `setappdata`

Introduced before R2006a

gunzip

Extract contents of GNU zip file

Syntax

```
gunzip(gzipfilenames)
gunzip(gzipfilenames,outputfolder)
filenames = gunzip( ___ )
```

Description

`gunzip(gzipfilenames)` extracts the archived contents of each file in `gzipfilenames` to the folder containing `gzipfilenames`. `gunzip` does not delete the original GNU zip files.

`gunzip` recursively extracts the content in folders. `gunzip` can extract files from your local system or files from an Internet URL.

`gunzip(gzipfilenames,outputfolder)` extracts `gzipfilenames` into `outputfolder`. If `outputfolder` does not exist, MATLAB creates it.

`filenames = gunzip( ___ )` returns a cell array of character vectors containing the relative path names of all resulting files. You can use this syntax with any of the input argument combinations in the previous syntaxes.

Examples

Extract GNU Zip Files in Current Folder

Use the `gunzip` function to extract all the GNU zip files in the current folder.

```
files = gunzip('*.gz')
```

```
files = 1x1 cell array
    {'membrane.m'}
```

Extract Archive File from URL

Download and extract an archive file from a URL to a local folder.

Suppose you have the archive file `example.tar.gz` stored at the URL `http://example.com/example.tar.gz`. Download and extract the file to the `example` folder.

```
url = 'http://example.com/example.tar.gz';
gunzip(url, 'example');
untar('example/example.tar','example');
```

Input Arguments

`gzipfilenames` — Names of the GNU zip files
character vector | cell array of character vectors | string array

Names of the GNU zip files to extract from, specified as a character vector, a cell array of character vectors, or a string array. File names must include a path relative to the current folder or an absolute path.

If `gzipfilenames` is a URL, `gzipfilenames` must include the protocol type (for example, `http://`). MATLAB downloads the URL to the temporary folder on your system, and then it deletes the URL on cleanup.

Data Types: `char` | `string`

`outputfolder` — Target folder
character vector | string scalar

Target folder for the extracted files, specified as a character vector or a string scalar.

Data Types: `char` | `string`

See Also

`gzip` | `tar` | `untar` | `unzip`

Introduced before R2006a

gzip

Compress files into GNU zip files

Syntax

```
gzip(filenames)
gzip(filenames,outputfolder)
entrynames = gzip( ___ )
```

Description

`gzip(filenames)` compresses the contents of the specified files and folders into GNU zip files with the file extension `.gz`. `gzip` recursively compresses the content in folders. `gzip` places each output file to the same folder as the input file.

`gzip(filenames,outputfolder)` places the resulting GNU zip files into `outputfolder`. If `outputfolder` does not exist, MATLAB creates it.

`entrynames = gzip( ___ )` returns a cell array of character vectors containing the relative path names of all resulting files. You can use this syntax with any of the input argument combinations in the previous syntaxes.

Examples

Compress Select Files Into GNU Zip Files

Compress all files with a `.m` and `.mlx` extension in the current folder and store the results in the `archive` folder.

```
compressedfiles = gzip({'*.m','*.mlx'},'archive');
compressedfiles
```

```
compressedfiles = 1x1 cell array
    {'archive\CompressSelectFilesExample.mlx.gz'}
```

Create GNU Zip File of a Folder

Create a GNU zip file of a folder including all subfolders, and store the relative paths in the file.

Create a folder `myfolder` containing a subfolder `mysubfolder` and the files `membrane.m` and `logo.m`.

```
mkdir myfolder;
movefile('membrane.m','myfolder');
movefile('logo.m','myfolder');
cd myfolder;
mkdir mysubfolder;
cd ..
```

Create a GNU zip file of the contents of `myfolder`, including all subfolders.

```
gzippedfiles = gzip('myfolder');
```

Input Arguments

`filenames` — Names of files or folders

character vector | cell array of character vectors | string array

Names of files or folders to compress, specified as a character vector, a cell array of character vectors, or a string array.

Files that are on the MATLAB path can include a partial path. Otherwise, files must include a path relative to the current folder or an absolute path.

Folders must include a path relative to the current folder or an absolute path. On UNIX systems, folders also can start with `~/` or `~username/`, which expands to the current user's home folder or the specified user's home folder, respectively. You can use the wildcard character * when specifying files or folders, except when relying on the MATLAB path to resolve a file name or partial path name.

Data Types: `char` | `string`

`outputfolder` — Target folder
character vector | string scalar

Target folder for the compressed files, specified as a character vector or a string scalar.

Data Types: `char` | `string`

See Also

`gunzip` | `tar` | `untar` | `unzip` | `zip`

Introduced before R2006a

h5create

Create HDF5 data set

Syntax

```
h5create(filename,datasetname,size,Name,Value)
```

Description

`h5create(filename,datasetname,size,Name,Value)` creates an HDF5 data set in the file specified by `filename`.

Input Arguments

`filename`

Character vector or string scalar specifying the name of an HDF5 file. If `filename` does not already exist, `h5create` creates it, with additional options specified by one or more `Name,Value` pair arguments.

Default:

`datasetname`

Character vector or string scalar specifying the name of the data set you want to create. If `datasetname` is a full path name, `h5create` creates all intermediate groups, if they don't already exist.

Default:

`size`

Array specifying the extents of the dataset. To specify an unlimited extent, set the corresponding element of `size` to `Inf`.

Default:

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`Datatype`

Any of the following MATLAB datatypes.

`'double'`	`'uint64'`	`'uint32'`	`'uint16'`	`'uint8'`
`'single'`	`'int64'`	`'int32'`	`'int16'`	`'int8'`

Default: `'double'`

`ChunkSize`

Defines chunking layout.

Default: Not chunked

`Deflate`

Defines gzip compression level (0-9).

Default: 0

`FillValue`

Defines the fill value for numeric data sets.

Default:

`Fletcher32`

Turns on the Fletcher32 checksum filter.

Default: `false`

Shuffle

Turns on the Shuffle filter.

Default: `false`

TextEncoding

Character encoding, specified as `'system'` or `'UTF-8'`.

- `'system'` — Use the system default encoding to create the dataset name.
- `'UTF-8'` — Use `UTF-8` encoding to create the dataset name.

Default: `'system'`

Examples

Create Fixed-Size Data Set

Create a fixed-size 100-by-200 data set.

```
h5create('myfile.h5','/myDataset1',[100 200])
h5disp('myfile.h5')
```

```
HDF5 myfile.h5
Group '/'
    Dataset 'myDataset1'
        Size:  100x200
        MaxSize:  100x200
        Datatype:   H5T_IEEE_F64LE (double)
        ChunkSize:  []
        Filters:  none
        FillValue:  0.000000
```

Create Data Set with Compression

Create a single-precision 1000-by-2000 data set with a chunk size of 50-by-80. Apply the highest level of compression.

```
h5create('myfile.h5','/myDataset2',[1000 2000],'Datatype','single', ...
          'ChunkSize',[50 80],'Deflate',9)
h5disp('myfile.h5')
```

```
HDF5 myfile.h5
Group '/'
    Dataset 'myDataset2'
        Size:  1000x2000
        MaxSize:  1000x2000
        Datatype:   H5T_IEEE_F32LE (single)
        ChunkSize:  50x80
        Filters:  deflate(9)
        FillValue:  0.000000
```

Create Data Set with Unlimited Dimension

Create a two-dimensional data set that is unlimited along the second extent.

```
h5create('myfile.h5','/myDataset3',[200 Inf],'ChunkSize',[20 20])
h5disp('myfile.h5')
```

```
HDF5 myfile.h5
Group '/'
    Dataset 'myDataset3'
        Size:  200x0
        MaxSize:  200xInf
        Datatype:   H5T_IEEE_F64LE (double)
        ChunkSize:  20x20
        Filters:  none
        FillValue:  0.000000
```

See Also

`h5disp` | `h5info` | `h5read` | `h5write`

Topics

"Exporting to HDF5 Files"
"Working with Non-ASCII Characters in HDF5 Files"

Introduced in R2011a

h5disp

Display contents of HDF5 file

Syntax

```
h5disp(filename)
h5disp(filename,location)
h5disp(filename,location,mode)
h5disp( ___ ,'TextEncoding',encoding)
```

Description

`h5disp(filename)` displays the structure (metadata) of the entire HDF5 file, `filename`.

`h5disp(filename,location)` displays the metadata for the specified location.

`h5disp(filename,location,mode)` displays the file metadata according to the value of `mode`.

`h5disp( ___ ,'TextEncoding',encoding)` specifies the text encoding to use for reading objects in the HDF5 file.

Input Arguments

`filename`

Character vector or string scalar specifying the name of an HDF5 file.

Default:

`location`

Character vector or string scalar specifying the full path to a location in an HDF5 file. To display the metadata for the entire file, specify `'/'` as the value of `location`. If `location` is a group, `h5disp` displays all objects below the group.

Default:

`mode`

Specified as one of these values:

Value	Description
`'min'`	Minimal, display only group and data set names.
`'simple'`	Display data set metadata and attribute values, if the attribute is an integer, floating point, or a scalar string.

Default: `'simple'`

`encoding`

Character encoding, specified as `'system'` or `'UTF-8'`.

- `'system'` — Use the system default encoding for names of objects and attributes.
- `'UTF-8'` — Use `UTF-8` encoding for names of objects and attributes.

Default: `'system'`

Examples

Display the entire contents of an HDF5 file.

```
h5disp('example.h5')
```

Display metadata for one data set in an HDF5 file.

```
h5disp('example.h5','/g4/world');
```

See Also

`h5info`

Topics

"Using the High-Level HDF5 Functions to Import Data"
"Working with Non-ASCII Characters in HDF5 Files"

Introduced in R2011a

h5info

Return information about HDF5 file

Syntax

```
info = h5info(filename)
info = h5info(filename,location)
info = h5info( ___ ,'TextEncoding',encoding)
```

Description

`info = h5info(filename)` returns information about the entire HDF5 file, specified by `filename`.

`info = h5info(filename,location)` returns information about the group, data set, or named datatype specified by `location` in the HDF5 file, `filename`.

`info = h5info( ___ ,'TextEncoding',encoding)` specifies the text encoding to use for reading objects in the HDF5 file.

Input Arguments

`filename`

Character vector or string scalar specifying the name of an HDF5 file.

Default:

`location`

Character vector or string scalar specifying the location of a group, data set, or named datatype in an HDF5 file.

Default:

encoding

Character encoding, specified as `'system'` or `'UTF-8'`.

- `'system'` — Use the system default encoding for names of objects and attributes.
- `'UTF-8'` — Use `UTF-8` encoding for names of objects and attributes.

Default: `'system'`

Output Arguments

info

A structure containing information about the HDF5 file. The set of fields in the structure depends on the `location` parameter. The first field is always `'Filename'`. Other fields that might be present in the `info` structure are as follows.

Location Type	Field	Description
Files and Groups		
	`Name`	Character vector specifying name of the group. If you specify only a file name, this value is `'/'`.
	`Groups`	Array of structures describing subgroups.
	`Datasets`	Array of structures describing data sets.
	`Datatypes`	Array of structures describing named datatypes.
	`Links`	Array of structures describing soft, external, user-defined, and certain hard links.
	`Attributes`	Array of structures describing group attributes.
Data sets		

Location Type	Field	Description
	`Name`	Character vector specifying the name of the data set.
	`Datatype`	Structure describing the datatype.
	`Dataspace`	Structure describing the size of the dataset.
	`ChunkSize`	Extents of the data set's chunk size, if defined.
	`FillValue`	Data set's fill value, if defined.
	`Filter`	Array of structures describing any defined filters such as compression.
	`Attributes`	Array of structures describing data set attributes.
Named Datatypes		
	`Name`	Character vector specifying the name of the datatype object.
	`Class`	HDF5 class of the named datatype.
	`Type`	Character vector or struct further describing the datatype.
	`Size`	Size of the named datatype in bytes.

Examples

Return all information.

```
info = h5info('example.h5');
```

Return information about a group and all data sets contained within the group.

```
info = h5info('example.h5','/g4');
```

Return information about a specific dataset.

```
info = h5info('example.h5','/g4/time');
```

See Also

`h5disp`

Topics

"Using the High-Level HDF5 Functions to Import Data"
"Working with Non-ASCII Characters in HDF5 Files"

Introduced in R2011a

h5readatt

Read attribute from HDF5 file

Syntax

```
attval = h5readatt(filename,location,attr)
```

Description

`attval = h5readatt(filename,location,attr)` retrieves the value for the named attribute `attr` from the given `location` in the HDF5 file `filename`.

Input Arguments

`filename`

Character vector or string scalar specifying the name of an HDF5 file.

Default:

`location`

Character vector or string scalar specifying the full path of the attribute in an HDF5 file. `location` can refer to either a group or a data set.

Default:

`attr`

Character vector or string scalar specifying the name of an attribute in an HDF5 file.

Default:

Output Arguments

attval

Value of the attribute.

Examples

Read Group Attribute

```
attval = h5readatt('example.h5','/','attr2')
```

```
attval = 2x2 int32 matrix

   0   2
   1   3
```

Read Data Set Attribute

Read the `units` attribute.

```
attval = h5readatt('example.h5','/g4/lon','units')
```

```
attval =
'degrees_east'
```

See Also

`h5info` | `h5writeatt`

Topics

"Using the High-Level HDF5 Functions to Import Data"

Introduced in R2011a

h5write

Write to HDF5 data set

Syntax

```
h5write(filename,datasetname,data)
h5write(filename,datasetname,data,start,count)
h5write(filename,datasetname,data,start,count,stride)
```

Description

`h5write(filename,datasetname,data)` writes `data` to an entire data set, `datasetname`, in the HDF5 file, `filename`.

`h5write(filename,datasetname,data,start,count)` writes a subset of the `data` to a data set, `datasetname`, in the HDF5 file, `filename`. `start` is a one-based index value that specifies the first element to be written. `count` specifies the number of elements to write along each dimension. `h5write` extends an extendable data set along any unlimited dimensions, if necessary.

`h5write(filename,datasetname,data,start,count,stride)` writes a hyperslab of data, where `stride` specifies the inter-element spacing along each dimension.

Input Arguments

`filename`

Character vector or string scalar specifying the name of an HDF5 file.

Default:

`datasetname`

Character vector or string scalar specifying the name of a data set in the HDF5 file.

Default:

`data`

Data to be written to the HDF5 file. You can specify only non-sparse floating-point and integer data sets.

Default:

`start`

Numeric index value specifying where in the data set to start writing to the file.

Default:

`count`

Numeric value specifying how much data to write to the file.

Default:

`stride`

Numeric value specifying the interelement spacing of data to write to the file.

Default: Vector of ones.

Examples

Write to Entire Data Set

Write random values to a data set named `DS1`.

```
h5create('myfile.h5','/DS1',[10 20])
mydata = rand(10,20);
h5write('myfile.h5', '/DS1', mydata)
```

Display the contents of the file.

```
h5disp('myfile.h5')
```

```
HDF5 myfile.h5
Group '/'
    Dataset 'DS1'
        Size:  10x20
        MaxSize:  10x20
        Datatype:   H5T_IEEE_F64LE (double)
        ChunkSize:  []
        Filters:  none
        FillValue:  0.000000
```

Write to Block of Data Set

Write a hyperslab of data to the last 5-by-7 block of a data set.

```
h5create('myfile.h5','/DS2',[10 20])
mydata = rand(5,7);
h5write('myfile.h5','/DS2',mydata,[6 14],[5 7])
```

Append Data to Unlimited Data Set

```
h5create('myfile.h5','/DS3',[20 Inf],'ChunkSize',[5 5]);
for j = 1:10
      data = j*ones(20,1);
      start = [1 j];
      count = [20 1];
      h5write('myfile.h5','/DS3',data,start,count);
end
h5disp('myfile.h5');
```

```
HDF5 myfile.h5
Group '/'
    Dataset 'DS3'
        Size:  20x10
        MaxSize:  20xInf
        Datatype:   H5T_IEEE_F64LE (double)
        ChunkSize:  5x5
        Filters:  none
        FillValue:  0.000000
```

Limitations

- `h5write` supports only floating point and integer data sets. To write to string data sets, you must use the H5D package.

More About

Hyperslab

A hyperslab is a collection of points in a data space. The points can be contiguous or form a regular pattern of points or blocks in a data space.

See Also

`H5D.create` | `H5D.write` | `h5create` | `h5disp` | `h5read` | `h5writeatt`

Topics

"Exporting to HDF5 Files"

Introduced in R2011a

h5writeatt

Write HDF5 attribute

Syntax

```
h5writeatt(filename,location,attname,attvalue)
h5writeatt(filename,location,attname,attvalue,'TextEncoding',encodin
g)
```

Description

`h5writeatt(filename,location,attname,attvalue)` writes the attribute named `attname` with the value `attvalue` to the HDF5 file `filename`. The parent object `location` can be either a group or variable. `location` is the complete path name of the group or variable to which you want to associate the attribute.

`h5writeatt(filename,location,attname,attvalue,'TextEncoding',encodin
g)` additionally specifies the text encoding to use when writing the attribute name or any attribute values that are of the text data type.

Input Arguments

`filename`

Character vector or string scalar specifying the name of an HDF5 file.

Default:

`location`

Character vector or string scalar specifying the full path identifying a group or variable in an HDF5 file.

Default:

`attname`

Character vector or string scalar specifying the name of an attribute in an HDF5 file. If the attribute does not exist, `h5writeatt` creates the attribute with the name specified.

If the specified attribute already exists but does not have a datatype or dataspace consistent with `attvalue`, `h5writeatt` deletes the attribute and recreates it. String attributes are created with a scalar dataspace.

Default:

`attvalue`

Value to be written to the attribute in an HDF5 file.

Default:

`encoding`

Character encoding, specified as `'system'` or `'UTF-8'`.

- `'system'` — Use the system default encoding for the attribute name and any attribute values that are of the text data type.
- `'UTF-8'` — Use `UTF-8` encoding for the attribute name and any attribute values that are of the text data type.

Default: `'system'`

Examples

Create a root group attribute whose value is the current time.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
h5writeatt('myfile.h5','/','creation_date',datestr(now));
```

Create a double-precision data set attribute.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
```

```
attData = [0 1 2 3];
h5writeatt('myfile.h5','/g4/world','attr',attData);
h5disp('myfile.h5','/g4/world');
```

See Also

h5disp | h5readatt | h5write

Topics

"Exporting to HDF5 Files"
"Working with Non-ASCII Characters in HDF5 Files"

Introduced in R2011a

H5.close

Close HDF5 library

Syntax

```
H5.close()
```

Description

`H5.close()` closes the HDF5 library.

See Also

`H5.open`

H5.garbage_collect

Free unused memory in HDF5 library

Syntax

```
H5.garbage_collect()
```

Description

`H5.garbage_collect()` frees unused memory in the HDF5 library.

H5.get_libversion

Version of HDF5 library

Syntax

```
[majnum,minnum,relnum] = H5.get_libversion()
```

Description

`[majnum,minnum,relnum] = H5.get_libversion()` returns the version of the HDF5 library in use.

H5.open

Open HDF5 library

Syntax

```
H5.open()
```

Description

`H5.open()` opens the HDF5 library.

See Also

`H5.close`

H5.set_free_list_limits

Set size limits on free lists

Syntax

```
H5.set_free_list_limits(reg_global_lim,reg_list_lim,arr_global_lim,a
rr_list_lim,blk_global_lim,blk_list_lim)
```

Description

`H5.set_free_list_limits(reg_global_lim,reg_list_lim,arr_global_lim,arr_list_lim,blk_global_lim,blk_list_lim)` sets size limits on all types of free lists.

H5A.close

Close specified attribute

Syntax

```
H5A.close(attr_id)
```

Description

`H5A.close(attr_id)` terminates access to the attribute specified by `attr_id`, releasing the identifier.

See Also

`H5A.open`

H5A.create

Create attribute

Syntax

```
attr_id = H5A.create(loc_id,name,type_id,space_id,acpl_id)
attr_id = H5A.create(loc_id,name,type_id,space_id,acpl_id,aapl_id)
```

Description

`attr_id = H5A.create(loc_id,name,type_id,space_id,acpl_id)` creates the attribute name that is attached to the object specified by `loc_id`. `loc_id` is a group, dataset, or named datatype identifier. The datatype and dataspace identifiers of the attribute, `type_id` and `space_id`, respectively, are created with the H5T and H5S interfaces. The attribute property list, `acpl_id`, is currently unused and should be set to `'H5P_DEFAULT'`. This interface corresponds to the 1.6.x version of `H5Acreate`.

`attr_id = H5A.create(loc_id,name,type_id,space_id,acpl_id,aapl_id)` creates the attribute with the additional attribute access property list identifier `aapl_id`. `aapl_id` should currently be set to `'H5P_DEFAULT'`. This interface corresponds to the 1.8.x version of `H5Acreate`.

Examples

```
acpl_id = H5P.create('H5P_ATTRIBUTE_CREATE');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
space_id = H5S.create('H5S_SCALAR');
fid = H5F.create('myfile.h5');
attr_id = H5A.create(fid,'my_attr',type_id,space_id,acpl_id);
H5A.close(attr_id);
H5F.close(fid);
```

See Also

`H5A.close` | `H5P.create`

H5A.delete

Delete attribute

Syntax

```
H5A.delete(loc_id,name)
```

Description

`H5A.delete(loc_id,name)` removes the attribute specified by `name` from the dataset, group, or named datatype specified by `loc_id`.

Examples

Delete a root group attribute.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
fid = H5F.open('myfile.h5','H5F_ACC_RDWR','H5P_DEFAULT');
gid = H5G.open(fid,'/');
H5A.delete(gid,'attr1');
H5G.close(gid);
H5F.close(fid);
```

H5A.get_info

Information about attribute

Syntax

```
info = H5A.get_info(attr_id)
```

Description

`info = H5A.get_info(attr_id)` returns information about an attribute specified by `attr_id`.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/');
attr_id = H5A.open(gid,'attr1');
info = H5A.get_info(attr_id);
H5A.close(attr_id);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5A.open`

H5A.get_name

Attribute name

Syntax

```
attr_name = H5A.get_name(attr_id)
attr_name = H5A.get_name(attr_id,'TextEncoding',encoding)
```

Description

`attr_name = H5A.get_name(attr_id)` returns the name of the attribute specified by `attr_id`.

`attr_name = H5A.get_name(attr_id,'TextEncoding',encoding)` specifies the text encoding to use to interpret the attribute name. Specify `encoding` as `'system'`(default) or `'UTF-8'`.

- `'system'` — Use the system default encoding to interpret the attribute name.
- `'UTF-8'` — Use `UTF-8` encoding to interpret the attribute name.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g1/g1.1');
idx_type = 'H5_INDEX_NAME';
order = 'H5_ITER_INC';
attr_id = H5A.open_by_idx(gid,'dset1.1.1',idx_type,order,0);
name = H5A.get_name(attr_id);
H5A.close(attr_id);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5A.open_by_idx`

Topics

"Working with Non-ASCII Characters in HDF5 Files"

H5A.get_space

Copy of attribute data space

Syntax

```
dspace_id = H5A.get_space(attr_id)
```

Description

`dspace_id = H5A.get_space(attr_id)` returns a copy of the data space for the attribute specified by `attr_id`.

Examples

Retrieve the dimensions of an attribute data space.

```
fid = H5F.open('example.h5');
attr_id = H5A.open(fid,'attr2');
space = H5A.get_space(attr_id);
[~,dims] = H5S.get_simple_extent_dims(space);
H5A.close(attr_id);
H5F.close(fid);
```

See Also

`H5A.open` | `H5S.close`

H5A.get_type

Copy of attribute data type

Syntax

```
type_id = H5A.get_type(attr_id)
```

Description

`type_id = H5A.get_type(attr_id)` returns a copy of the data type for the attribute specified by `attr_id`.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/');
attr_id = H5A.open(gid,'attr1');
type_id = H5A.get_type(attr_id);
H5T.close(type_id);
H5A.close(attr_id);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5A.open` | `H5T.close`

H5A.iterate

Execute function for attributes attached to object

Syntax

```
[status,idx_stop,cdata_out] =
H5A.iterate(obj_id,idx_type,order,idx_start,iter_func,cdata_in)
H5A.iterate(loc_id,attr_idx,iterator_func)
```

Description

`[status,idx_stop,cdata_out] = H5A.iterate(obj_id,idx_type,order,idx_start,iter_func,cdata_in)` executes the specified function `iter_func` for each attribute connected to an object. `obj_id` identifies the object to which attributes are attached. `idx_type` is the type of index and valid values include the following.

`'H5_INDEX_NAME'`	An alpha-numeric index by attribute name
`'H5_INDEX_CRT_ORDER'`	An index by creation order

`order` specifies the index traversal order. Valid values include the following.

`'H5_ITER_INC'`	Iteration from beginning to end
`'H5_ITER_DEC'`	Iteration from end to beginning
`'H5_ITER_NATIVE'`	Iteration in the fastest available order

`idx_start` specifies the starting point of the iteration. `idx_stop` returns the point at which iteration was stopped. This allows an interrupted iteration to be resumed.

The callback function, `iter_func`, must have the following signature:

```
[status,cdata_out] = iter_func(obj_id,attr_name,info,cdata_in)
```

`cdata_in` is a user-defined value or structure and is passed to the first step of the iteration in the `iter_func` `cdata_in` parameter. The `cdata_out` of an iteration step

forms the `cdata_in` for the next iteration step. Then, the final `cdata_out` at the end of the iteration is returned to the caller as `cdata_out`. This form of `H5A.iterate` corresponds to the `H5Aiterate2` function in the HDF5 C API.

`status` value returned by `iter_func` is interpreted as follows.

zero	Continues with the iteration or returns zero status value to the caller if all members have been processed
positive	Stops the iteration and returns the positive status value to the caller
negative	Stops the iteration and throws an error indicating failure

`H5A.iterate(loc_id,attr_idx,iterator_func)` executes the specified function for each attribute of the group, dataset, or named datatype specified by `loc_id`. The `attr_idx` argument specifies where the iteration begins. `iterator_func` must be a function handle.

The iterator function must have the following signature:

```
status = iterator_func(loc_id,attr_name)
```

`loc_id` still specifies the group, dataset, or named data type passed into `H5A.iterate`, and `attr_name` specifies the current attribute. This form of `H5A.iterate` corresponds to `H5Aiterate1` function in the HDF5 C API.

H5A.open

Open attribute

Syntax

```
attr_id = H5A.open(obj_id,attr_name)
attr_id = H5A.open(obj_id,attr_name,aapl_id)
```

Description

`attr_id = H5A.open(obj_id,attr_name)` opens an attribute for an object specified by a parent object identifier and attribute name.

`attr_id = H5A.open(obj_id,attr_name,aapl_id)` opens an attribute with an attribute access property list identifier, `aapl_id`. The only currently valid value for `aapl_id` is `'H5P_DEFAULT'`.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/');
attr_id = H5A.open(gid,'attr1');
H5A.close(attr_id);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5A.close` | `H5A.open_by_idx` | `H5A.open_by_name`

H5A.open_by_idx

Open attribute specified by index

Syntax

```
attr_id = H5A.open_by_idx(loc_id,obj_name,idx_type,order,n)
attr_id =
H5A.open_by_idx(loc_id,obj_name,idx_type,order,n,aapl_id,lapl_id)
```

Description

`attr_id = H5A.open_by_idx(loc_id,obj_name,idx_type,order,n)` opens an existing attribute at index `n` attached to an object specified by its location, `loc_id`, and name, `obj_name`.

`idx_type` is the type of index and valid values include the following.

`'H5_INDEX_NAME'`	An alpha-numeric index by attribute name
`'H5_INDEX_CRT_ORDER'`	An index by creation order

`order` specifies the index traversal order. Valid values include the following.

`'H5_ITER_INC'`	Iteration from beginning to end
`'H5_ITER_DEC'`	Iteration from end to beginning
`'H5_ITER_NATIVE'`	Iteration in the fastest available order

`attr_id = H5A.open_by_idx(loc_id,obj_name,idx_type,order,n,aapl_id,lapl_id)` opens an attribute with attribute access property list, `aapl_id`, and link access property list, `lapl_id`. The `aapl_id` argument must currently be specified as `'H5P_DEFAULT'`. Also, `lapl_id` can be specified by `'H5P_DEFAULT'`.

Examples

Loop through a set of dataset attributes in reverse alphabetical order.

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g1/g1.1');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
info = H5O.get_info(dset_id);
for idx = 0:info.num_attrs-1
        attr_id =H5A.open_by_idx(gid,'dset1.1.1','H5_INDEX_NAME','H5_ITER_DEC',idx);
        fprintf('attribute name:  %s\n',H5A.get_name(attr_id));
        H5A.close(attr_id);
end
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5A.close` | `H5A.open` | `H5A.open_by_name`

H5A.open_by_name

Open attribute specified by name

Syntax

```
attr_id = H5A.open_by_name(loc_id,obj_name,attr_name)
attr_id =
H5A.open_by_name(loc_id,obj_name,attr_name,aapl_id,lapl_id)
```

Description

`attr_id = H5A.open_by_name(loc_id,obj_name,attr_name)` opens an existing attribute `attr_name` attached to an object specified by its location `loc_id` and name `obj_name`.

`attr_id = H5A.open_by_name(loc_id,obj_name,attr_name,aapl_id,lapl_id)` opens an existing attribute with the attribute access property list `aapl_id` and link access property list `lacpl_id`. `aapl_id` must be specified as `'H5P_DEFAULT'`. `lapl_id` may also be specified by `'H5P_DEFAULT'`.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g1/g1.1');
attr_id = H5A.open_by_name(gid,'dset1.1.1','attr1');
H5A.close(attr_id);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5A.close` | `H5A.open` | `H5A.open_by_idx`

H5A.read

Read attribute

Syntax

```
attr = H5A.read(attr_id)
attr = H5A.read(attr_id, mem_type_id)
```

Description

`attr = H5A.read(attr_id)` reads the attribute specified by `attr_id`. MATLAB will determine the appropriate memory datatype.

`attr = H5A.read(attr_id, mem_type_id)` reads the attribute specified by `attr_id`. `mem_type_id` specifies the attribute's memory datatype and should usually be given as `'H5ML_DEFAULT'`, which specifies that MATLAB will determine the appropriate memory datatype.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. If the HDF5 library reports the attribute size as 3-by-4-by-5, then the corresponding MATLAB array size is 5-by-4-by-3. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/');
attr_id = H5A.open(gid,'attr1');
data = H5A.read(attr_id);
H5A.close(attr_id);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5A.open` | `H5A.write`

H5A.write

Write attribute

Syntax

```
H5A.write(attr_id,type_id,buf)
```

Description

`H5A.write(attr_id,type_id,buf)` writes the data in buf into the attribute specified by `attr_id`. `type_id` specifies the attribute's memory datatype. The memory datatype should be `'H5ML_DEFAULT'`, which specifies that MATLAB should determine the appropriate memory datatype.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. If the MATLAB array size is 5-by-4-by-3, then the HDF5 library should be reporting the attribute size as 3-by-4-by-5. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

Write a scalar double precision attribute.

```
acpl = H5P.create('H5P_ATTRIBUTE_CREATE');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
space_id = H5S.create('H5S_SCALAR');
fid = H5F.create('myfile.h5');
attr_id = H5A.create(fid,'my_attr',type_id,space_id,acpl);
H5A.write(attr_id,'H5ML_DEFAULT',10.0)
H5A.close(attr_id);
H5F.close(fid);
H5T.close(type_id);
```

See Also

`H5A.read`

H5D.close

Close dataset

Syntax

```
H5D.close(dataset_id)
```

Description

`H5D.close(dataset_id)` ends access to a dataset specified by `dataset_id` and releases resources used by it.

See Also

`H5D.create` | `H5D.open`

H5D.create

Create new dataset

Syntax

```
dataset_id = H5D.create(loc_id,name,type_id,space_id,plist_id)
dataset_id =
H5D.create(loc_id,name,type_id,space_id,lcpl_id,dcpl_id,dapl_id)
```

Description

`dataset_id = H5D.create(loc_id,name,type_id,space_id,plist_id)` creates the data set specified by `name` in the file or in the group specified by `loc_id`. `type_id` and `space_id` identify the datatype and dataspace, respectively. `plist_id` identifies the dataset creation property list. This interface corresponds to the `H5Dcreate1` function in the HDF5 library C 1.6 API.

`dataset_id =`
`H5D.create(loc_id,name,type_id,space_id,lcpl_id,dcpl_id,dapl_id)` creates the data set with three distinct property lists:

`lcpl_id`	link creation property list
`dcpl_id`	dataset creation property list
`dapl_id`	dataset access property list

This interface corresponds to the `H5Dcreate` function in the HDF5 library C 1.8 API.

Examples

Create a 10x5 double precision dataset with default property list settings.

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [10 5];
```

```
h5_dims = fliplr(dims);
h5_maxdims = h5_dims;
space_id = H5S.create_simple(2,h5_dims,h5_maxdims);
dcpl = 'H5P_DEFAULT';
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5S.close(space_id);
H5T.close(type_id);
H5D.close(dset_id);
H5F.close(fid);
h5disp('myfile.h5');
```

Create a 6x3 fixed length string dataset. Each string will have a length of 4 characters.

```
fid = H5F.create('myfile_strings.h5');
type_id = H5T.copy('H5T_C_S1');
H5T.set_size(type_id,4);
dims = [6 3];
h5_dims = fliplr(dims);
h5_maxdims = h5_dims;
space_id = H5S.create_simple(2,h5_dims,h5_maxdims);
dcpl = 'H5P_DEFAULT';
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5S.close(space_id);
H5T.close(type_id);
H5D.close(dset_id);
H5F.close(fid);
h5disp('myfile_strings.h5');
```

See Also

`H5D.close` | `H5S.close` | `H5S.create_simple` | `H5T.copy`

H5D.get_access_plist

Copy of dataset access property list

Syntax

```
plist_id = H5D.get_access_plist(dataset_id)
```

Description

`plist_id = H5D.get_access_plist(dataset_id)` returns a copy of the dataset access property list used to open the specified dataset.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
dapl = H5D.get_access_plist(dset_id);
H5P.close(dapl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5D.get_create_plist` | `H5P.close`

H5D.get_create_plist

Copy of dataset creation property list

Syntax

```
plist_id = H5D.get_create_plist(dataset_id)
```

Description

`plist_id = H5D.get_create_plist(dataset_id)` returns the identifier to a copy of the dataset creation property list for the dataset specified by `dataset_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
dcpl = H5D.get_create_plist(dset_id);
H5P.close(dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5D.get_access_plist` | `H5P.close`

H5D.get_offset

Location of dataset in file

Syntax

```
offset = H5D.get_offset(dataset_id)
```

Description

`offset = H5D.get_offset(dataset_id)` returns the location in the file of the dataset specified by `dataset_id`. The location is expressed as an offset, in bytes, from the beginning of the file.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
offset = H5D.get_offset(dset_id);
H5D.close(dset_id);
H5F.close(fid);
```

H5D.get_space

Copy of dataset data space

Syntax

```
dspace_id = H5D.get_space(dataset_id)
```

Description

`dspace_id = H5D.get_space(dataset_id)` returns an identifier for a copy of the data space for a dataset.

Examples

Retrieve the dimensions of an attribute data space.

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
space = H5D.get_space(dset_id);
[~,dims] = H5S.get_simple_extent_dims(space);
H5S.close(space);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5D.open` | `H5S.close`

H5D.get_space_status

Determine if space is allocated

Syntax

```
status = H5D.get_space_status(dataset_id)
```

Description

`status = H5D.get_space_status(dataset_id)` determines whether space has been allocated for the dataset specified by `dataset_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
status = H5D.get_space_status(dset_id);
switch(status)
    case H5ML.get_constant_value('H5D_SPACE_STATUS_NOT_ALLOCATED')
        fprintf('Not allocated.\n');
    case H5ML.get_constant_value('H5D_SPACE_STATUS_ALLOCATED')
        fprintf('Allocated.\n');
    case H5ML.get_constant_value('H5D_SPACE_STATUS_PART_ALLOCATED')
        fprintf('Part allocated.\n');
end
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5D.get_space`

H5D.get_storage_size

Determine required storage size

Syntax

```
dataset_size = H5D.get_storage_size(dataset_id)
```

Description

`dataset_size = H5D.get_storage_size(dataset_id)` returns the amount of storage that is required for the dataset specified by `dataset_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
dataset_size = H5D.get_storage_size(dset_id);
H5D.close(dset_id);
H5F.close(fid);
```

H5D.get_type

Copy of datatype

Syntax

```
type_id = H5D.get_type(dataset_id)
```

Description

`type_id = H5D.get_type(dataset_id)` returns an identifier for a copy of the data type for the dataset specified by `dataset_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
type_id = H5D.get_type(dset_id);
H5T.close(type_id);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5T.close`

H5D.open

Open specified dataset

Syntax

```
dataset_id = H5D.open(loc_id,name)
dataset_id = H5D.open(loc_id,name,dapl_id)
```

Description

`dataset_id = H5D.open(loc_id,name)` opens the dataset specified by `name` in the file or group specified by `loc_id`.

`dataset_id = H5D.open(loc_id,name,dapl_id)` opens the dataset specified by `name` in the file or group specified by `loc_id`. The dataset access property list, `dapl_id`, provides information regarding access to the dataset.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g2');
dset_id = H5D.open(gid,'dset2.2');
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5D.close`

H5D.read

Read data from HDF5 dataset

Syntax

```
data = H5D.read(dataset_id)
data =
H5D.read(dataset_id,mem_type_id,mem_space_id,file_space_id,dxpl)
```

Description

`data = H5D.read(dataset_id)` reads the entire dataset specified by `dataset_id`.

`data = H5D.read(dataset_id,mem_type_id,mem_space_id,file_space_id,dxpl)` reads the dataset specified by `dataset_id`. The `mem_type_id` input specifies the memory data type and should usually be `'H5ML_DEFAULT'` to allow MATLAB to determine the appropriate value. `mem_space_id` describes how the data is to be arranged in memory and should usually be `'H5S_ALL'`. The `file_space_id` input describes how the data is to be selected from the file. It also can be given as `'H5S_ALL'`, but this results in the entire dataset being read into memory. `dxpl` is the dataset transfer property list identifier and should usually be `'H5P_DEFAULT'`.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

Read an entire dataset.

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
data = H5D.read(dset_id);
```

```
H5D.close(dset_id);
H5F.close(fid);
```

Read the 2x3 hyperslab starting in the 4th row and 5th column of the example dataset.

```
plist = 'H5P_DEFAULT';
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
dims = fliplr([2 3]);
mem_space_id = H5S.create_simple(2,dims,[]);
file_space_id = H5D.get_space(dset_id);
offset = fliplr([3 4]);
block = fliplr([2 3]);
H5S.select_hyperslab(file_space_id,'H5S_SELECT_SET',offset,[],[],block);
data = H5D.read(dset_id,'H5ML_DEFAULT',mem_space_id,file_space_id,plist);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5D.open` | `H5D.write` | `H5S.create_simple`

H5D.set_extent

Change size of dataset dimensions

Syntax

```
H5D.set_extent(dset_id,h5_extents)
```

Description

`H5D.set_extent(dset_id,h5_extents)` changes the dimensions of the dataset `dset_id` to the sizes specified in `h5_extents`.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_extents` parameter assumes C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

Extend an unlimited one-dimensional dataset from a length of 10 to a length of 20.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
fid = H5F.open('myfile.h5','H5F_ACC_RDWR','H5P_DEFAULT');
dset_id = H5D.open(fid,'/g4/time');
H5D.set_extent(dset_id,20);
H5D.close(dset_id);
H5F.close(fid);
```

H5D.vlen_get_buf_size

Determine variable length storage requirements

Syntax

```
size = H5D.vlen_get_buf_size(dataset_id,type_id,space_id)
```

Description

`size = H5D.vlen_get_buf_size(dataset_id,type_id,space_id)` determines the number of bytes required to store the VL data from the dataset, using the `space_id` for the selection in the dataset on disk and the `type_id` for the memory representation of the VL data in memory.

H5D.write

Write data to HDF5 dataset

Syntax

```
H5D.write(dataset_id,mem_type_id,mem_space_id,file_space_id,plist_id
,buf)
```

Description

`H5D.write(dataset_id,mem_type_id,mem_space_id,file_space_id,plist_id,buf)` writes the dataset specified by `dataset_id` from the application memory buffer `buf` into the file. `plist_id` specifies the data transfer properties. `mem_type_id` identifies the memory datatype of the dataset. `mem_space_id` and `file_space_id` define the part of the dataset to write. The memory datatype should usually be `'H5ML_DEFAULT'`, which specifies that MATLAB should determine the appropriate memory datatype.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

Write to the entire 36-by-19 `/g4/world` example dataset.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
plist = 'H5P_DEFAULT';
fid = H5F.open('myfile.h5','H5F_ACC_RDWR',plist);
dset_id = H5D.open(fid,'/g4/world');
dims = [36 19];
data = rand(dims);
```

```
H5D.write(dset_id,'H5ML_DEFAULT','H5S_ALL','H5S_ALL',plist,data);
H5D.close(dset_id);
H5F.close(fid);
```

Write to the entire two-element `/g3/VLstring` dataset.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
h5disp('myfile.h5','/g3/VLstring');
plist = 'H5P_DEFAULT';
fid = H5F.open('myfile.h5','H5F_ACC_RDWR',plist);
dset_id = H5D.open(fid,'/g3/VLstring');
data = {'dogs'; 'dogs and cats'};
H5D.write(dset_id,'H5ML_DEFAULT','H5S_ALL','H5S_ALL',plist,data);
H5D.close(dset_id);
H5F.close(fid);
data_out = h5read('myfile.h5','/g3/VLstring');
```

Write a 10-by-5 block of data to the location starting at row index 15 and column index 5 of the same dataset. Recall that indexing is zero-based.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
plist = 'H5P_DEFAULT';
fid = H5F.open('myfile.h5','H5F_ACC_RDWR',plist);
dset_id = H5D.open(fid,'/g4/world');
start = [15 5];
h5_start = fliplr(start);
block = [10 5];
h5_block = fliplr(block);
mem_space_id = H5S.create_simple(2,h5_block,[]);
file_space_id = H5D.get_space(dset_id);
H5S.select_hyperslab(file_space_id,'H5S_SELECT_SET',h5_start,[],[],h5_block);
data = rand(block);
H5D.write(dset_id,'H5ML_DEFAULT',mem_space_id,file_space_id,plist,data);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5D.read`

H5DS.attach_scale

Attach dimension scale to specific dataset dimension

Syntax

```
H5DS.attach_scale(dataset_id,dimscale_id,idx)
```

Description

`H5DS.attach_scale(dataset_id,dimscale_id,idx)` attaches a dimension scale `dimscale_id` to dimension `idx` of the dataset `dataset_id`.

Note The ordering of the dimension scale indices are the same as the HDF5 library C API.

Examples

Add the `'lon'` and `'lat'` dimension scales to the `'world'` dataset.

```
plist = 'H5P_DEFAULT';
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
fid = H5F.open('myfile.h5','H5F_ACC_RDWR',plist);
world_dset_id = H5D.open(fid,'/g4/world',plist);
lat_dset_id = H5D.open(fid,'/g4/lat',plist);
lon_dset_id = H5D.open(fid,'/g4/lon',plist);
H5DS.attach_scale(world_dset_id,lat_dset_id,0);
H5DS.attach_scale(world_dset_id,lon_dset_id,1);
H5D.close(lat_dset_id);
H5D.close(lon_dset_id);
H5D.close(world_dset_id);
H5F.close(fid);
```

See Also

`H5DS.detach_scale`

H5DS.detach_scale

Detach dimension scale from specific dataset dimension

Syntax

```
H5DS.detach_scale(dataset_id,dimscale_id,idx)
```

Description

`H5DS.detach_scale(dataset_id,dimscale_id,idx)` detaches dimension scale `dimscale_id` from dimension `idx` of the dataset `dataset_id`.

Note The ordering of the dimension scale indices are the same as the HDF5 library C API.

See Also

`H5DS.attach_scale`

H5DS.get_label

Retrieve label from specific dataset dimension

Syntax

```
label = H5DS.get_label(dataset_id,idx)
```

Description

`label = H5DS.get_label(dataset_id,idx)` retrieves the label for dimension `idx` of the dataset `dataset_id`.

Note The ordering of the dimension scale indices are the same as the HDF5 library C API.

Examples

```
fid = H5F.open('example.h5');
world_dset_id = H5D.open(fid,'/g4/world');
label = H5DS.get_label(world_dset_id,0);
H5D.close(world_dset_id);
H5F.close(fid);
```

See Also

`H5DS.set_label`

H5DS.get_num_scales

Number of scales attached to dataset dimension

Syntax

```
num_scales = H5DS.get_num_scales(dataset_id,idx)
```

Description

`num_scales = H5DS.get_num_scales(dataset_id,idx)` determines the number of dimension scales that are attached to dimension `idx` of the dataset `dataset_id`.

Examples

```
fid = H5F.open('example.h5');
world_dset_id = H5D.open(fid,'/g4/world');
num_scales = H5DS.get_num_scales(world_dset_id,0);
H5D.close(world_dset_id);
H5F.close(fid);
```

H5DS.get_scale_name

Name of dimension scale

Syntax

```
name = H5DS.get_scale_name(dimscale_id)
```

Description

`name = H5DS.get_scale_name(dimscale_id)` retrieves the name of the dimension scale `dimscale_id`.

Examples

```
fid = H5F.open('example.h5');
lat_dset_id = H5D.open(fid,'/g4/lat');
scale_name = H5DS.get_scale_name(lat_dset_id);
H5D.close(lat_dset_id);
H5F.close(fid);
```

See Also

`H5DS.set_scale`

H5DS.is_scale

Determine if dataset is a dimension scale

Syntax

```
bool = H5DS.is_scale(dataset_id)
```

Description

`bool = H5DS.is_scale(dataset_id)` determines whether the dataset `dataset_id` is a dimension scale.

Examples

```
fid = H5F.open('example.h5');
lat_dset_id = H5D.open(fid,'/g4/lat');
if H5DS.is_scale(lat_dset_id)
    fprintf('/g4/lat is a dimension scale.\n');
else
    fprintf('/g4/lat is not a dimension scale.\n');
end
H5D.close(lat_dset_id);
H5F.close(fid);
```

H5DS.iterate_scales

Iterate on scales attached to dataset dimension

Syntax

```
[status,idx_out,opdata_out] =
H5DS.iterate_scales(dset_id,dim,idx_in,iter_func,opdata_in)
```

Description

`[status,idx_out,opdata_out] = H5DS.iterate_scales(dset_id,dim,idx_in,iter_func,opdata_in)` iterates over the scales attached to dimension `dim` of the dataset `dset_id` to perform a common operation whose function handle is `iter_func`.

`idx_in` specifies the starting point of the iteration. `idx_out` returns the point at which iteration was stopped. This allows an interrupted iteration to be resumed. If `idx_in` is `[]`, then the iterator starts at the first member.

The callback function `iter_func` must have the following signature:

```
function [status,opdata_out] =
iter_func(dset_id,dim,dimscale_id,opdata_in)
```

`opdata_in` is a user-defined value or structure and is passed to the first step of the iteration in the `iter_func` `opdata_in` parameter. The `opdata_out` of an iteration step forms the `opdata_in` for the next iteration step. The final `opdata_out` at the end of the iteration is then returned to the caller as `opdata_out`.

`dimscale_id` specifies the current dimension scale dataset identifier and `dim` is the associated dimension.

status value returned by `iter_func` is interpreted as follows:

zero	Continues with the iteration or returns zero status value to the caller if all members have been processed
positive	Stops the iteration and returns the positive status value to the caller
negative	Stops the iteration and throws an error indicating failure

H5DS.set_label

Set label for dataset dimension

Syntax

```
H5DS.set_label(dataset_id,idx,label)
```

Description

`H5DS.set_label(dataset_id,idx,label)` sets a label for dimension `idx` of the dataset `dataset_id`.

Note The ordering of the dimension scale indices are the same as the HDF5 library C API.

Examples

```
plist = 'H5P_DEFAULT';
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
fid = H5F.open('myfile.h5','H5F_ACC_RDWR',plist);
world_dset_id = H5D.open(fid,'/g4/world',plist);
H5DS.set_label(world_dset_id,0,'latitude');
H5DS.set_label(world_dset_id,1,'longitude');
H5D.close(world_dset_id);
H5F.close(fid);
```

See Also

`H5DS.get_label`

H5DS.set_scale

Convert dataset to dimension scale

Syntax

```
H5DS.set_scale(dataset_id,dim_name)
```

Description

`H5DS.set_scale(dataset_id,dim_name)` converts the dataset, `dataset_id`, to a dimension scale with name `dim_name`.

Examples

Create a dimension scale with name `'xdim'`. The dataset has the name, `'x'`.

```
fid = H5F.create('myfile.h5');
space_id = H5S.create_simple(1,10,10);
dtype = 'H5T_NATIVE_INT';
dcpl = 'H5P_DEFAULT';
dset_id = H5D.create(fid,'x',dtype,space_id,dcpl);
H5DS.set_scale(dset_id,'xdim');
H5S.close(space_id);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5DS.get_scale_name`

H5E.clear

Clear error stack

Syntax

```
H5E.clear()
```

Description

`H5E.clear()` clears the error stack for the current thread.

H5E.get_major

Description of major error number

Syntax

```
err_string = H5E.get_major(major_number)
```

Description

`err_string = H5E.get_major(major_number)` returns a character vector describing an error specified by the major error number, `major_number`.

The HDF5 group has deprecated the use of this function.

See Also

`H5E.get_minor`

H5E.get_minor

Description of minor error number

Syntax

```
err_string = H5E.get_minor(minor_number)
```

Description

`err_string = H5E.get_minor(minor_number)` returns a character vector describing an error specified by the minor error number, `minor_number`.

The HDF5 group has deprecated the use of this function.

See Also

`H5E.get_major`

H5E.walk

Walk error stack

Syntax

```
H5E.walk(direction,func)
```

Description

`H5E.walk(direction,func)` walks the error stack for the current thread and calls the specified function for each error along the way. `func` is a function handle. `direction` specifies how the error stack is traversed and can be given by one of the following character vectors or the numeric equivalent.

```
'H5E_WALK_UPWARD'
'H5E_WALK_DOWNWARD'
```

The specified function must have the following signature:

```
status = func(n,error_struct)
```

where `n` is the indexed position of the error in the stack and `error_struct` is a structure with the following fields:

`maj_num`	Major error number
`min_num`	Minor error number
`func_name`	Function in which the error occurred
`file_name`	File in which the error occurred
`line`	Line in file where error occurs
`desc`	Optional supplied description

This function corresponds to the `H5Ewalk1` function in the HDF5 library C API.

See Also

`H5ML.get_constant_value`

H5F.close

Close HDF5 file

Syntax

```
H5F.close(file_id)
```

Description

`H5F.close(file_id)` terminates access to HDF5 file identified by `file_id`, flushing all data to storage.

See Also

`H5F.open`

H5F.create

Create HDF5 file

Syntax

```
file_id = H5F.create(filename)
file_id = H5F.create(name,flags,fcpl_id,fapl_id)
```

Description

`file_id = H5F.create(filename)` creates the file specified by `filename` with default library properties if the file does not already exist.

`file_id = H5F.create(name,flags,fcpl_id,fapl_id)` creates the file specified by `name`. `flags` specifies whether to truncate the file, if it already exists, or to fail if the file already exists. `flags` can be specified by one of the following character vectors or string scalars, or their numeric equivalent:

`'H5F_ACC_TRUNC'`	overwrite any existing file by the same name
`'H5F_ACC_EXCL'`	do not overwrite an existing file

`fcpl_id` is the file creation property list identifier. `fapl_id` is the file access property list identifier. A value of `'H5P_DEFAULT'` for either property list indicates that the library should use default values for the appropriate property list.

Examples

Create an HDF5 file called `'myfile.h5'`.

```
fid = H5F.create('myfile.h5');
H5F.close(fid);
```

Create an HDF5 file called `'myfile.h5'`, overwriting any existing file by the same name. Default file access and file creation properties shall apply.

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

H5F.close | H5ML.get_constant_value | H5P.create

H5F.flush

Flush buffers to disk

Syntax

```
H5F.flush(object_id,scope)
```

Description

`H5F.flush(object_id,scope)` causes all buffers associated with a file to be immediately flushed to disk without removing the data from the cache. `object_id` can be any object associated with the file, including the file itself, a dataset, a group, an attribute, or a named data type. `scope` specifies whether the scope of the flushing action is global or local. Specify `scope` as one of these character vectors or string scalars:

```
'H5F_SCOPE_GLOBAL'
'H5F_SCOPE_LOCAL'
```

H5F.get_access_plist

File access property list

Syntax

```
fapl_id = H5F.get_access_plist(file_id)
```

Description

`fapl_id = H5F.get_access_plist(file_id)` returns the file access property list identifier of the file specified by `file_id`.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
H5P.close(fapl);
H5F.close(fid);
```

See Also

`H5F.get_create_plist`

H5F.get_create_plist

File creation property list

Syntax

```
fcpl_id = H5F.get_create_plist(file_id)
```

Description

`fcpl_id = H5F.get_create_plist(file_id)` returns a file creation property list identifier identifying the creation properties used to create the file specified by `file_id`.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
H5P.close(fcpl);
H5F.close(fid);
```

See Also

`H5F.get_access_plist`

H5F.get_filesize

Size of HDF5 file

Syntax

```
size = H5F.get_filesize(file_id)
```

Description

`size = H5F.get_filesize(file_id)` returns the size of the HDF5 file specified by `file_id`.

H5F.get_freespace

Amount of free space in file

Syntax

```
free_space = H5F.get_freespace(file_id)
```

Description

`free_space = H5F.get_freespace(file_id)` returns the amount of space that is unused by any objects in the file specified by `file_id`.

H5F.get_info

Global information about file

Syntax

```
file_info = H5F.get_info(obj_id)
```

Description

`file_info = H5F.get_info(obj_id)` returns global information for the file associated with the object identifier `obj_id`. For details about the fields of the `file_info` structure, please refer to the HDF5 documentation.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'g2');
info = H5F.get_info(gid);
H5G.close(gid);
H5F.close(fid);
```

H5F.get_mdc_config

Metadata cache configuration

Syntax

```
config_struct = H5F.get_mdc_config(file_id)
```

Description

`config_struct = H5F.get_mdc_config(file_id)` returns the current metadata cache configuration for the target file.

Examples

```
fid = H5F.open('example.h5');
config = H5F.get_mdc_config(fid);
H5F.close(fid);
```

See Also

`H5F.set_mdc_config`

H5F.get_mdc_hit_rate

Metadata cache hit-rate

Syntax

```
hitRate = H5F.get_mdc_hit_rate(file_id)
```

Description

`hitRate = H5F.get_mdc_hit_rate(file_id)` queries the metadata cache of the target file to obtain its hit-rate since the last time hit-rate statistics were reset. If the cache has not been accessed since the last time the hit-rate statistics were reset, the hit-rate is defined to be 0.0. The hit-rate is calculated as

```
(cache hits / (cache hits + cache misses))
```

Examples

```
fid = H5F.open('example.h5');
hit_rate = H5F.get_mdc_hit_rate(fid);
H5F.close(fid);
```

See Also

`H5F.get_mdc_config`

H5F.get_mdc_size

Metadata cache size data

Syntax

```
[max_sz,min_clean_sz,cursz,num_cur_entries] =
H5F.get_mdc_size(fileId)
```

Description

`[max_sz,min_clean_sz,cursz,num_cur_entries] = H5F.get_mdc_size(fileId)` queries the metadata cache of the target file to obtain current metadata cache size information.

Examples

```
fid = H5F.open('example.h5');
[maxsz,minsz,cursz,nent] = H5F.get_mdc_size(fid);
H5F.close(fid);
```

See Also

`H5F.get_mdc_config`

H5F.get_name

Name of HDF5 file

Syntax

```
name = H5F.get_name(obj_id)
```

Description

`name = H5F.get_name(obj_id)` returns the name of the file to which the object `obj_id` belongs. The object can be a group, dataset, attribute, or named data type.

Examples

```
fid = H5F.open('example.h5');
name = H5F.get_name(fid);
H5F.close(fid);
```

See Also

`H5A.get_name` | `H5I.get_name`

H5F.get_obj_count

Number of open objects in HDF5 file

Syntax

```
obj_count = H5F.get_obj_count(file_id,types)
```

Description

`obj_count = H5F.get_obj_count(file_id,types)` returns the number of open object identifiers for the file specified by `file_id` for the specified type. Specify `types` as one of the following character vectors or string scalars.

```
'H5F_OBJ_FILE'
'H5F_OBJ_DATASET'
'H5F_OBJ_GROUP'
'H5F_OBJ_DATATYPE'
'H5F_OBJ_ATTR'
'H5F_OBJ_ALL'
'H5F_OBJ_LOCAL'
```

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g2');
obj_count = H5F.get_obj_count(fid,'H5F_OBJ_GROUP');
H5G.close(gid);
H5F.close(fid);
```

See Also

H5F.get_obj_ids

H5F.get_obj_ids

List of open HDF5 file objects

Syntax

```
[num_obj_ids,obj_id_list] = H5F.get_obj_ids(file_id,types,max_objs)
```

Description

`[num_obj_ids,obj_id_list] = H5F.get_obj_ids(file_id,types,max_objs)` returns a list of all open identifiers for HDF5 objects of the type specified by `types` in the file specified by `file_id`. The `max_objs` input specifies the maximum number of object identifiers to return. `num_obj_ids` is the total number of objects in the list. Specify `types` as one of these character vectors or string scalars:

```
'H5F_OBJ_FILE'
'H5F_OBJ_DATASET'
'H5F_OBJ_GROUP'
'H5F_OBJ_DATATYPE'
'H5F_OBJ_ATTR'
'H5F_OBJ_ALL'
'H5F_OBJ_LOCAL'
```

Note If the number of objects of the specified type, that are open, exceeds `max_objs`, then `num_obj_ids` will be greater than `max_objs`.

Examples

```
fid = H5F.open('example.h5');
gid1 = H5G.open(fid,'/g1');
gid2 = H5G.open(fid,'/g2');
```

```
gid3 = H5G.open(fid,'/g3');
gid4 = H5G.open(fid,'/g4');
[num_obj_ids,objs] = H5F.get_obj_ids(fid,'H5F_OBJ_GROUP',3);
H5G.close(gid1);
H5G.close(gid2);
H5G.close(gid3);
H5G.close(gid4);
H5F.close(fid);
```

See Also

H5F.get_obj_count

H5F.is_hdf5

Determine if file is HDF5

Syntax

```
value = H5F.is_hdf5(name)
```

Description

`value = H5F.is_hdf5(name)` returns a positive number if the file specified by `name` is in the HDF5 format, and zero if it is not. A negative return value indicates failure.

Examples

```
value = H5F.is_hdf5('example.tif');
if value > 0
    fprintf('example.tif is an HDF5 file\n');
else
    fprintf('example.tif is not an HDF5 file\n');
end
```

H5F.mount

Mount HDF5 file onto specified location

Syntax

```
H5F.mount(loc_id,name,child_id,plist_id)
```

Description

`H5F.mount(loc_id,name,child_id,plist_id)` mounts the file specified by `child_id` onto the group specified by `loc_id` and `name`, using the mount properties specified by `plist_id`.

Examples

Mount one file with a dataset onto a group in a second file and access the dataset via the second file.

```
plist = 'H5P_DEFAULT';
fid2 = H5F.create('file2.h5','H5F_ACC_TRUNC',plist,plist);
gid2 = H5G.create(fid2,'g2',plist,plist,plist);
fid1 = H5F.create('file1.h5','H5F_ACC_TRUNC','H5P_DEFAULT',...
        'H5P_DEFAULT');
space_id = H5S.create('H5S_SCALAR');
dset_id = H5D.create(fid1,'DS1','H5T_NATIVE_DOUBLE',space_id,plist);
H5S.close(space_id);
H5D.close(dset_id);
H5F.mount(fid2,'g2',fid1,plist);
dset_id1 = H5D.open(fid1,'/g2/DS1',plist);
H5D.close(dset_id1);
H5F.unmount(fid1,'g2');
H5G.close(gid2);
H5F.close(fid1);
H5F.close(fid2);
```

See Also

`H5F.unmount`

H5F.open

Open HDF5 file

Syntax

```
file_id = H5F.open(filename)
file_id = H5F.open(name,flags,fapl_id)
```

Description

`file_id = H5F.open(filename)` opens the file specified by `filename` for read-only access and returns the file identifier, `file_id`.

`file_id = H5F.open(name,flags,fapl_id)` opens the file specified by `name`, returning the file identifier, `file_id`. Specify the file access flags `flags` as one of the these character vectors or string scalars, or their numeric equivalents:

`'H5F_ACC_RDWR'`	read-write mode
`'H5F_ACC_RDONLY'`	read-only mode

The file access property list, `fapl_id`, may be specified as `'H5P_DEFAULT'`, in which case the default I/O settings are used.

Examples

Open a file in read-only mode with default file access properties.

```
fid = H5F.open('example.h5');
H5F.close(fid);
```

Open a file in read-write mode.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
```

```
fid = H5F.open('myfile.h5','H5F_ACC_RDWR','H5P_DEFAULT');
H5F.close(fid);
```

See Also

H5F.close | H5ML.get_constant_value

H5F.reopen

Reopen HDF5 file

Syntax

```
new_file_id = H5F.reopen(file_id)
```

Description

`new_file_id = H5F.reopen(file_id)` returns a new file identifier for the already open HDF5 file specified by `file_id`.

See Also

`H5F.open`

H5F.set_mdc_config

Configure HDF5 file metadata cache

Syntax

```
H5F.set_mdc_config(fileId,config)
```

Description

`H5F.set_mdc_config(fileId,config)` attempts to configure the file's metadata cache according to the supplied configuration structure. Before using this function, you should retrieve the current configuration using `H5F.get_mdc_config`.

See Also

`H5F.get_mdc_config`

H5F.unmount

Unmount file or group from mount point

Syntax

```
H5F.unmount(loc_id,name)
```

Description

`H5F.unmount(loc_id,name)` dissassociates the file or group specified by `loc_id` from the mount point specified by `name`. `loc_id` can be a file or group identifier.

See Also

`H5F.mount`

H5G.close

Close group

Syntax

```
H5G.close(group_id)
```

Description

`H5G.close(group_id)` releases resources used by the group specified by `group_id`. `group_id` was returned by either `H5G.create` or `H5G.open`.

See Also

`H5G.create` | `H5G.open`

H5G.create

Create group

Syntax

```
group_id = H5G.create(loc_id,name,size_hint)
group_id = H5G.create(loc_id,name,lcpl_id,gcpl_id,gapl_id)
```

Description

`group_id = H5G.create(loc_id,name,size_hint)` creates a new group with the name specified by `name` at the location specified by `loc_id`. `loc_id` can be a file or group identifier. `size_hint` specifies the number of bytes to reserve for the names that will appear in the group. This interface corresponds to the 1.6 version of `H5Gcreate`.

`group_id = H5G.create(loc_id,name,lcpl_id,gcpl_id,gapl_id)` creates a new group with link creation, group creation, and group access property lists `lcpl_id`, `gcpl_id`, and `gapl_id`. This interface corresponds to the 1.8 version of `H5Gcreate`.

Examples

Create an HDF5 file `'myfile.h5'` with a group `'my_group'` with default property list settings.

```
fid = H5F.create('myfile.h5');
plist = 'H5P_DEFAULT';
gid = H5G.create(fid,'my_group',plist,plist,plist);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5G.close` | `H5G.open`

H5G.get_info

Information about group

Syntax

```
info = H5G.get_info(group_id)
```

Description

`info = H5G.get_info(group_id)` retrieves information about the group specified by `group_id`.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g2');
info = H5G.get_info(gid);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5G.create` | `H5G.open`

H5G.open

Open specified group

Syntax

```
group_id = H5G.open(loc_id,name)
group_id = H5G.open(loc_id,name,gapl_id)
```

Description

`group_id = H5G.open(loc_id,name)` opens the group specified by `name` at the location specified by `loc_id`. `loc_id` is a file or group identifier. This interface corresponds to the 1.6 version of `H5Gopen`.

`group_id = H5G.open(loc_id,name,gapl_id)` opens the group with an additional group access property list, `gapl_id`. This interface corresponds to the 1.8 version of `H5Gopen`.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g2');
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5G.close` | `H5P.create`

H5I.dec_ref

Decrement reference count

Syntax

```
ref_count = H5I.dec_ref(obj_id)
```

Description

`ref_count = H5I.dec_ref(obj_id)` decrements the reference count of the object identified by `obj_id` and returns the new count.

See Also

`H5I.get_ref` | `H5I.inc_ref`

H5I.get_file_id

File identifier for specified object

Syntax

```
file_id = H5I.get_file_id(obj_id)
```

Description

`file_id = H5I.get_file_id(obj_id)` returns the identifier of the file associated with the object referenced by `obj_id`.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g4');
fid2 = H5I.get_file_id(gid);
name = H5F.get_name(fid2);
fprintf('The filename is %s.\n',name);
H5G.close(gid);
H5F.close(fid);
H5F.close(fid2);
```

H5I.get_name

Name of object

Syntax

```
name = H5I.get_name(obj_id)
name = H5I.get_name(obj_id,'TextEncoding',encoding)
```

Description

`name = H5I.get_name(obj_id)` returns the name of the object specified by `obj_id`. If no name is attached to the object, an empty character vector is returned.

`name = H5I.get_name(obj_id,'TextEncoding',encoding)` additionally specifies the text encoding to use to interpret the object name. Specify `encoding` as `'system'`(default) or `'UTF-8'`.

- `'system'` — Use the system default encoding to interpret the object name.
- `'UTF-8'` — Use `UTF-8` encoding to interpret the object name.

Examples

Display the names of all the objects in the `/g3` group in the example file by alphabetical order.

```
idx_type = 'H5_INDEX_NAME';
order = 'H5_ITER_INC';
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g3');
info = H5G.get_info(gid);
for j = 1:info.nlinks
    obj_id = H5O.open_by_idx(fid,'g3',idx_type,order,j-1,'H5P_DEFAULT');
    name = H5I.get_name(obj_id);
    fprintf('Object %d: ''%s''.\n',j-1,name);
    H5O.close(obj_id);
```

```
end
H5G.close(gid);
H5F.close(fid);
```

See Also

H5A.get_name | H5F.get_name

Topics

"Working with Non-ASCII Characters in HDF5 Files"

H5I.get_ref

Reference count of object

Syntax

```
refcount = H5I.get_ref(obj_id)
```

Description

`refcount = H5I.get_ref(obj_id)` returns the reference count of the object specified by `obj_id`.

See Also

`H5I.dec_ref` | `H5I.inc_ref`

H5I.get_type

Type of object

Syntax

```
obj_type = H5I.get_type(obj_id)
```

Description

`obj_type = H5I.get_type(obj_id)` returns the type of the object specified by `obj_id`. `obj_type` corresponds to one of the following enumerated values.

```
H5I_FILE
H5I_GROUP
H5I_DATATYPE
H5I_DATASPACE
H5I_DATASET
H5I_ATTR
H5I_BADID
```

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g3');
dset_id = H5D.open(fid,'/g4/world');
[~,objs] = H5F.get_obj_ids(fid,'H5F_OBJ_ALL',3);
for j = 1:numel(objs)
    name = H5I.get_name(objs(j));
    fprintf('object ''%s'':  ==> ',name);
    type = H5I.get_type(objs(j));
    switch(type)
        case H5ML.get_constant_value('H5I_FILE')
            fprintf('FILE identifier.\n');
```

```
        case H5ML.get_constant_value('H5I_GROUP')
            fprintf('GROUP identifier.\n');
        case H5ML.get_constant_value('H5I_DATASET')
            fprintf('DATASET identifier.\n');
        otherwise
            fprintf('unknown identifier type.\n');
    end
end
H5G.close(gid);
H5F.close(fid);
```

See Also

H5ML.get_constant_value

H5I.inc_ref

Increment reference count of specified object

Syntax

```
ref_count = H5I.inc_ref(obj_id)
```

Description

`ref_count = H5I.inc_ref(obj_id)` increments the reference count of the object specified by `obj_id` and returns the new count.

See Also

`H5I.dec_ref` | `H5I.get_ref`

H5I.is_valid

Determine if specified identifier is valid

Syntax

```
tf = H5I.is_valid(obj_id)
```

Description

`tf = H5I.is_valid(obj_id)` determines whether the identifier `obj_id` is valid.

Examples

```
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.close(fapl);
if H5I.is_valid(fapl);
    fprintf('File access property list is valid.\n');
else
    fprintf('File access property list is not valid.\n');
end
```

H5L.copy

Copy link from source location to destination location

Syntax

```
H5L.copy(src_loc_id,src_name,dest_loc_id,dest_name,lcpl_id,lapl_id)
```

Description

`H5L.copy(src_loc_id,src_name,dest_loc_id,dest_name,lcpl_id,lapl_id)` copies the link specified by `src_name` from the file or group specified by `src_loc_id` to the destination `dest_loc_id`. The new copy of the link is created with the name `dest_name`.

`dest_loc_id` must refer to either the current file or a group in the current file. If `dest_loc_id` is the file identifier, the copy is placed in the file's root group.

The new link is created with the creation and access property lists specified by `lcpl_id` and `lapl_id`.

Examples

```
plist_id = 'H5P_DEFAULT';
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',plist_id,plist_id);
g1 = H5G.create(fid,'g1',plist_id);
g2 = H5G.create(fid,'g2',plist_id);
g11 = H5G.create(g1,'g3',plist_id);
H5L.copy(g1,'g3',g2,'g4',plist_id,plist_id);
```

H5L.create_external

Create soft link to external object

Syntax

```
H5L.create_external(filename,objname,link_loc_id,link_name,lcpl_id,l
apl_id)
```

Description

`H5L.create_external(filename,objname,link_loc_id,link_name,lcpl_id,l apl_id)` creates a soft link to an object in a different file. `filename` identifies the target file containing the target object. `obj_name` specifies the path to the target object within that file. `obj_name` must start at the target file's root group but is not interpreted until lookup time.

`link_loc_id` and `link_name` specify the location and name, respectively, of the new link. `link_name` is interpreted relative to `link_loc_id`.

`lcpl_id` and `lapl_id` are the link creation and access property lists associated with the new link.

Examples

```
plist_id = 'H5P_DEFAULT';
fid1 = H5F.create('myfile1.h5');
g1 = H5G.create(fid1,'g1',plist_id,plist_id,plist_id);
H5G.close(g1);
H5F.close(fid1);
fid2 = H5F.create('myfile2.h5');
H5L.create_external('myfile1.h5','g1',fid2,'g2',plist_id,plist_id);
```

H5L.create_hard

Create hard link

Syntax

```
H5L.create_hard(obj_loc_id,obj_name,link_loc_id,link_name,lcpl_id,lapl_id)
```

Description

`H5L.create_hard(obj_loc_id,obj_name,link_loc_id,link_name,lcpl_id,lapl_id)` creates a new hard link to a pre-existing object in an HDF5 file. The new link may be one of many that point to that object. `obj_loc_id` and `obj_name` specify the location and name, respectively, of the target object, i.e., the object to which the new hard link points.

`link_loc_id` and `link_name` specify the location and name, respectively, of the new link. `link_name` is interpreted relative to `link_loc_id`.

`lcpl_id` and `lapl_id` are the link creation and access property lists associated with the new link.

Examples

```
fid = H5F.create('myfile.h5');
gid1 = H5G.create(fid,'/g1',0);
gid2 = H5G.create(gid1,'g2',0);
gid3 = H5G.create(gid2,'g3',0);
lcpl = 'H5P_DEFAULT';
lapl = 'H5P_DEFAULT';
H5L.create_hard(gid2,'g3',gid1,'g4',lcpl,lapl);
H5G.close(gid3);
H5G.close(gid2);
H5G.close(gid1);
H5F.close(fid);
```

See Also

`H5L.create_soft`

H5L.create_soft

Create soft link

Syntax

```
H5L.create_soft(target_path,link_loc_id,link_name,lcpl_id,lapl_id)
```

Description

`H5L.create_soft(target_path,link_loc_id,link_name,lcpl_id,lapl_id)` creates a new soft link to an object in an HDF5 file. The new link may be one of many that point to that object. `target_path` specifies the path to the target object, i.e., the object that the new soft link points to. `target_path` can be anything and is interpreted at lookup time. This `target_path` may be absolute in the file or relative to `link_loc_id`.

`link_loc_id` and `link_name` specify the location and name, respectively, of the new link. `link_name` is interpreted relative to `link_loc_id`.

`lcpl_id` and `lapl_id` are the link creation and access property lists associated with the new link.

Examples

```
plist_id = 'H5P_DEFAULT';
fid = H5F.create('myfile.h5');
gid1 = H5G.create(fid,'/g1',0);
gid3 = H5G.create(gid1,'g3',0);
gid2 = H5G.create(fid,'/g2',0);
lcpl = 'H5P_DEFAULT';
lapl = 'H5P_DEFAULT';
H5L.create_soft('/g1/g3',gid2,'g4',lcpl,lapl);
H5G.close(gid3);
H5G.close(gid2);
H5G.close(gid1);
H5F.close(fid);
```

See Also

`H5L.create_hard`

H5L.delete

Remove link

Syntax

```
H5L.delete(loc_id,name,lapl_id)
```

Description

`H5L.delete(loc_id,name,lapl_id)` removes the link specified by name from the location `loc_id`. `lapl_id` is a link access property list identifier.

Examples

Remove the only link to the `'/g3'` group in `example.h5`.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
fid = H5F.open('myfile.h5','H5F_ACC_RDWR','H5P_DEFAULT');
H5L.delete(fid,'g3','H5P_DEFAULT');
H5F.close(fid);
```

See Also

`H5L.move`

H5L.exists

Determine if link exists

Syntax

```
bool = H5L.exists(loc_id,name,lapl_id)
```

Description

`bool = H5L.exists(loc_id,name,lapl_id)` checks if a link specified by the pairing of an object id and name exists within a group. `lapl_id` is a link access property list identifier.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g1/g1.2/g1.2.1');
if H5L.exists(gid,'slink','H5P_DEFAULT')
    fprintf('link exists\n');
else
    fprintf('link does not exist\n');
end
```

H5L.get_info

Information about link

Syntax

```
linkStruct = H5L.get_info(location_id,link_name,lapl_id)
```

Description

`linkStruct = H5L.get_info(location_id,link_name,lapl_id)` returns information about a link.

A file or group identifier, `location_id`, specifies the location of the link. `link_name`, interpreted relative to `link_id`, specifies the link being queried.

Examples

```
fid = H5F.open('example.h5');
info = H5L.get_info(fid,'g3','H5P_DEFAULT');
H5F.close(fid);
```

H5L.get_name_by_idx

Information about link specified by index

Syntax

```
name =
H5L.get_name_by_idx(loc_id,group_name,idx_type,order,n,lapl_id)
name =
H5L.get_name_by_idx(loc_id,group_name,idx_type,order,n,lapl_id,'Text
Encoding',encoding)
```

Description

`name = H5L.get_name_by_idx(loc_id,group_name,idx_type,order,n,lapl_id)` retrieves information about a link at index `n`, present in group `group_name`, at location `loc_id`. The `lapl_id` input specifies the link access property list for querying the group.

Specify the type of index `idx_type` as one of these character vectors or string scalars.

`'H5_INDEX_NAME'`	Alpha-numeric index on name
`'H5_INDEX_CRT_ORDER'`	Index on creation order

Specify the index traversal order `order` as one of these character vectors or string scalars.

`'H5_ITER_INC'`	Iteration from beginning to end
`'H5_ITER_DEC'`	Iteration from end to beginning
`'H5_ITER_NATIVE'`	Iteration in the fastest available order

`name = H5L.get_name_by_idx(loc_id,group_name,idx_type,order,n,lapl_id,'Text Encoding',encoding)` additionally specifies the text encoding to use to interpret the link name. Specify `encoding` as `'system'`(default) or `'UTF-8'`.

- `'system'` — Use the system default encoding to interpret the link name.
- `'UTF-8'` — Use `UTF-8` encoding to interpret the link name.

Examples

```
fid = H5F.open('example.h5');
idx_type = 'H5_INDEX_NAME';
order = 'H5_ITER_DEC';
lapl_id = 'H5P_DEFAULT';
name = H5L.get_name_by_idx(fid,'g3',idx_type,order,0,lapl_id);
H5F.close(fid);
```

See Also

Topics

"Working with Non-ASCII Characters in HDF5 Files"

H5L.get_val

Value of symbolic link

Syntax

```
linkval = H5L.get_val(link_loc_id,link_name,lapl_id)
linkval =
H5L.get_val(link_loc_id,link_name,lapl_id,'TextEncoding',encoding)
```

Description

`linkval = H5L.get_val(link_loc_id,link_name,lapl_id)` returns the value of a symbolic link.

`link_loc_id` is a file or group identifier. `link_name` identifies a symbolic link and is defined relative to `link_loc_id`. Symbolic links include soft and external links and some user-defined links.

In the case of soft links, `linkval` is a cell array containing the path to which the link points.

In the case of external links, `linkval` is a cell array consisting of the name of the target file and the object name.

This function corresponds to the `H5L.get_val` and `H5Lunpack_elink_val` functions in the HDF5 1.8 C API.

`linkval = H5L.get_val(link_loc_id,link_name,lapl_id,'TextEncoding',encoding)` additionally specifies the text encoding to use to interpret the link value. Specify `encoding` as `'system'`(default) or `'UTF-8'`.

- `'system'` — Use the system default encoding to interpret the link value.
- `'UTF-8'` — Use `UTF-8` encoding to interpret the link value.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g1/g1.2/g1.2.1');
linkval = H5L.get_val(gid,'slink','H5P_DEFAULT');
H5G.close(gid);
H5F.close(fid);
```

See Also

Topics

"Working with Non-ASCII Characters in HDF5 Files"

H5L.iterate

Iterate over links

Syntax

```
[status,idx_out,opdata_out] =
H5L.iterate(group_id,index_type,order,idx_in,iter_func,opdata_in)
```

Description

`[status,idx_out,opdata_out] = H5L.iterate(group_id,index_type,order,idx_in,iter_func,opdata_in)` iterates through the links in a group, specified by `group_id`, to perform a common function whose function handle is `iter_func`. `H5L.iterate` does not recursively follow links into subgroups of the specified group.

`index_type` and `order` establish the iteration. `index_type` specifies the index to be used. If the links have not been indexed by the index type, they will first be sorted by that index then the iteration will begin. If the links have been so indexed, the sorting step will be unnecessary, so the iteration may begin more quickly. Specify `index_type` as one of these character vectors or string scalars.

`'H5_INDEX_NAME'`	Alpha-numeric index on name
`'H5_INDEX_CRT_ORDER'`	Index on creation order

`order` specifies the order in which objects are to be inspected along the index specified in `index_type`. Specify `order` as one of these character vectors or string scalars.

`'H5_ITER_INC'`	Increasing order
`'H5_ITER_DEC'`	Decreasing order
`'H5_ITER_NATIVE'`	Fastest available order

`idx_in` specifies the starting point of the iteration. `idx_out` returns the point at which iteration was stopped. This allows an interrupted iteration to be resumed.

The callback function `iter_func` must have the following signature:

```
function [status,opdata_out] = iter_func(group_id,name,opdata_in)
```

`opdata_in` is a user-defined value or structure and is passed to the first step of the iteration in the `iter_func` `opdata_in` parameter. The `opdata_out` of an iteration step forms the `opdata_in` for the next iteration step. The final `opdata_out` at the end of the iteration is then returned to the caller as `opdata_out`.

`status` value returned by `iter_func` is interpreted as follows:

zero	Continues with the iteration or returns zero status value to the caller if all members have been processed
positive	Stops the iteration and returns the positive status value to the caller
negative	Stops the iteration and throws an error indicating failure

H5L.iterate_by_name

Iterate through links in group specified by name

Syntax

```
[status,idx_out,opdata_out] =
H5L.iterate_by_name(loc_id,group_name,index_type,order,idx_in,iter_f
unc,opdata_in,lapl_id)
```

Description

`[status,idx_out,opdata_out] = H5L.iterate_by_name(loc_id,group_name,index_type,order,idx_in,iter_func,opdata_in,lapl_id)` iterates through the links in a group to perform a common function whose function handle is `iter_func`. The starting point of the iteration is pairing of a specified by the location id and a relative group name. `H5L.iterate_by_name` does not recursively follow links into subgroups of the specified group. A link access property list, `lapl_id`, may affect the outcome depending upon the type of link being traversed.

`index_type` and `order` establish the iteration. `index_type` specifies the index to be used. If the links have not been indexed by the index type, they will first be sorted by that index then the iteration will begin. If the links have been so indexed, the sorting step will be unnecessary, so the iteration may begin more quickly. Specify `index_type` as one of these character vectors or string scalars.

`'H5_INDEX_NAME'`	Alpha-numeric index on name
`'H5_INDEX_CRT_ORDER'`	Index on creation order

`order` specifies the order in which objects are to be inspected along the index specified in `index_type`. Specify `order` as one of these character vectors or string scalars.

`'H5_ITER_INC'`	Increasing order
`'H5_ITER_DEC'`	Decreasing order

`'H5_ITER_NATIVE'`	Fastest available order

`idx_in` specifies the starting point of the iteration. `idx_out` returns the point at which iteration was stopped. This allows an interrupted iteration to be resumed.

The callback function `iter_func` must have the following signature:

```
function [status,opdata_out] = iter_func(group_id,name,opdata_in)
```

`opdata_in` is a user-defined value or structure and is passed to the first step of the iteration in the `iter_func` `opdata_in` parameter. The `opdata_out` of an iteration step forms the `opdata_in` for the next iteration step. The final opdata_out at the end of the iteration is then returned to the caller as `opdata_out`.

`status` value returned by `iter_func` is interpreted as follows:

zero	Continues with the iteration or returns zero status value to the caller if all members have been processed
positive	Stops the iteration and returns the positive status value to the caller
negative	Stops the iteration and throws an error indicating failure

H5L.move

Rename link

Syntax

```
H5L.move(src_loc_id,src_name,dest_loc_id,dest_name,lcpl_id,lapl_id)
```

Description

`H5L.move(src_loc_id,src_name,dest_loc_id,dest_name,lcpl_id,lapl_id)` renames a link within an HDF5 file. The original link, `src_name`, is removed from the group graph and the new link, `dest_name`, is inserted; this change is accomplished as an atomic operation.

`src_loc_id` and `src_name` identify the existing link. `src_loc_id` is either a file or group identifier; `src_name` is the path to the link and is interpreted relative to `src_loc_id`.

`dest_loc_id` and `dest_name` identify the new link. `dest_loc_id` is either a file or group identifier; `dest_name` is the path to the link and is interpreted relative to `dest_loc_id`.

`lcpl_id` and `lapl_id` are the link creation and link access property lists, respectively, associated with the new link, `dest_name`.

Examples

Rename the `'/g2'` group to `'/g2/g3'`.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
fid = H5F.open('myfile.h5','H5F_ACC_RDWR','H5P_DEFAULT');
g2id = H5G.open(fid,'g2');
H5L.move(fid,'g3',g2id,'g3','H5P_DEFAULT','H5P_DEFAULT');
```

```
H5G.close(g2id);
H5F.close(fid);
```

See Also

H5L.delete

H5L.visit

Recursively iterate through links in group specified by group identifier

Syntax

```
[status opdata_out] =
H5L.visit(group_id,index_type,order,iter_func,opdata_in)
```

Description

`[status opdata_out] = H5L.visit(group_id,index_type,order,iter_func,opdata_in)` recursively iterates through all links in and below a group, specified by `group_id`, to perform a common function whose function handle is `iter_func`.

`index_type` and order establish the iteration. `index_type` specifies the index to be used. If the links have not been indexed by the index type, they are first sorted by that index, and then the iteration will begin. If the links have been so indexed, the sorting step is unnecessary, so the iteration can begin more quickly. Specify `index_type` as one of these character vectors or string scalars.

`'H5_INDEX_NAME'`	Alpha-numeric index on name
`'H5_INDEX_CRT_ORDER'`	Index on creation order

Note that the index type passed in `index_type` is a best effort setting. If the application passes in a value indicating iteration in creation order and a group is encountered that was not tracked in creation order, that group will be iterated over in alpha-numeric order by name, or name order. (Name order is the native order used by the HDF5 Library and is always available.)

`order` specifies the order in which objects are to be inspected along the index specified in `index_type`. Specify `order` as one of these character vectors or string scalars.

`'H5_ITER_INC'`	Increasing order

`'H5_ITER_DEC'`	Decreasing order
`'H5_ITER_NATIVE'`	Fastest available order

The callback function `iter_func` must have the following signature:

```
function [status,opdata_out] = iter_func(group_id,name,opdata_in)
```

`opdata_in` is a user-defined value or structure and is passed to the first step of the iteration in the `iter_func` `opdata_in` parameter. The `opdata_out` of an iteration step forms the `opdata_in` for the next iteration step. The final `opdata_out` at the end of the iteration is then returned to the caller as `opdata_out`.

`status` value returned by `iter_func` is interpreted as follows.

zero	Continues with the iteration or returns zero status value to the caller if all members have been processed
positive	Stops the iteration and returns the positive status value to the caller
negative	Stops the iteration and throws an error indicating failure

H5L.visit_by_name

Recursively iterate through links in group specified by location and group name

Syntax

```
[status,opdata_out] =
H5L.visit_by_name(loc_id,group_name,index_type,order,iter_func,opdat
a_in,lapl_id)
```

Description

`[status,opdata_out] = H5L.visit_by_name(loc_id,group_name,index_type,order,iter_func,opdata_in,lapl_id)` recursively iterates though all links in and below a group to perform a common function whose function handle is `iter_func`. The starting point of the iteration is specified by the pairing of a location id and a relative group name. A link access property list, `lapl_id`, may affect the outcome depending upon the type of link being traversed.

`index_type` and `order` establish the iteration. `index_type` specifies the index to be used. If the links have not been indexed by the index type, they are first sorted by that index, and then the iteration will begin. If the links have been so indexed, the sorting step is unnecessary, so the iteration can begin more quickly. Specify `index_type` as one of these character vectors or string scalars.

`'H5_INDEX_NAME'`	Alpha-numeric index on name
`'H5_INDEX_CRT_ORDER'`	Index on creation order

Note that the index type passed in `index_type` is a best effort setting. If the application passes in a value indicating iteration in creation order and a group is encountered that was not tracked in creation order, that group will be iterated over in alpha-numeric order by name, or name order. (Name order is the native order used by the HDF5 Library and is always available.)

`order` specifies the order in which objects are to be inspected along the index specified in `index_type`. Specify `order` as one of these character vectors or string scalars.

`'H5_ITER_INC'`	Increasing order
`'H5_ITER_DEC'`	Decreasing order
`'H5_ITER_NATIVE'`	Fastest available order

The callback function `iter_func` must have the following signature:

```
function [status,opdata_out] = iter_func(group_id,name,opdata_in)
```

`opdata_in` is a user-defined value or structure and is passed to the first step of the iteration in the `iter_func` `opdata_in` parameter. The `opdata_out` of an iteration step forms the `opdata_in` for the next iteration step. The final `opdata_out` at the end of the iteration is then returned to the caller as `opdata_out`.

`status` value returned by `iter_func` is interpreted as follows.

zero	Continues with the iteration or returns zero status value to the caller if all members have been processed
positive	Stops the iteration and returns the positive status value to the caller
negative	Stops the iteration and throws an error indicating failure

H5ML.compare_values

Numerically compare two HDF5 values

Syntax

```
bEqual = H5ML.compare_values(value1,value2)
```

Description

`bEqual = H5ML.compare_values(value1,value2)` compares two values, where either or both values may be specified by their text representation (for example, `'H5T_NATIVE_INT'`). The values are compared numerically.

Function parameters:

`bEqual`	A logical value indicating whether the two values are equal
`value1`	The first value to be compared
`value2`	The second value to be compared

Examples

```
val = H5ML.get_constant_value('H5T_NATIVE_INT');
H5ML.compare_values(val,'H5T_NATIVE_INT')
```

H5ML.get_constant_names

Constants known by HDF5 library

Syntax

```
names = H5ML.get_constant_names()
```

Description

`names = H5ML.get_constant_names()` returns a list of known library constants, definitions, and enumerations. When these names are supplied as actual parameters to HDF5 functions, they are automatically be converted to the appropriate numeric value.

Function parameters.

`names`	An alphabetized cell array of character vectors containing the names.

H5ML.get_constant_value

Value corresponding to a string

Syntax

```
value = H5ML.get_constant_value(constant)
```

Description

`value = H5ML.get_constant_value(constant)` returns the numerical value corresponding to a given HDF5 enumeration or identifier. `constant` is specified as a character vector or string scalar that contains an enumeration (for example, `'H5_ENUM_T'`) or a predefined identifier (for example, `'H5T_NATIVE_INT'`). Since the value corresponding to a given identifier is not guaranteed to remain the same, it is almost always preferable to use the `H5ML.compare_values()` function instead.

Function parameters:

`value`	The numerical value corresponding to the supplied HDF5 enumeration or identifier.
`constant`	A character vector or string scalar containing a HDF5 enumeration or defined value.

Examples

```
a = H5ML.get_constant_value('H5T_NATIVE_INT');
```

H5ML.get_function_names

Functions provided by HDF5 library

Syntax

```
names = H5ML.get_function_names()
```

Description

`names = H5ML.get_function_names()` returns a list of supported library functions.

Function parameters:

`names`	An alphabetized cell array of names

H5ML.get_mem_datatype

Data type for dataset ID

Syntax

```
DTYPE_ID = H5ML.get_mem_datatype(LOCATION_ID)
```

Description

`DTYPE_ID = H5ML.get_mem_datatype(LOCATION_ID)` returns the ID of an HDF5 memory datatype for the dataset or attribute identified by `LOCATION_ID`. This HDF5 memory datatype is the default used by `H5D.read` or `H5D.write` when you specify `'H5ML_DEFAULT'` as a value of the memory data type parameter.

The identifier returned by `H5ML.get_mem_datatype` should eventually be closed by calling `H5T.close` to release resources.

Examples

```
file_id = H5F.open('example.h5','H5F_ACC_RDONLY','H5P_DEFAULT');
dset_id = H5D.open(file_id,'/g1/g1.1/dset1.1.1');
datatype_id = H5ML.get_mem_datatype(dset_id)
H5T.close(datatype_id);
H5D.close(dset_id);
H5F.close(file_id);
```

H5ML.hoffset

Determine the offset of a field within a structure

Note `H5ML.hoffset` is not recommended. Use `H5T` instead.

Syntax

Description

This function is used to determine the offset (in bytes) of a structure, `H5T.insert(file_type,'a', offset(1), dtype(1));`, within a field. It is used when constructing an HDF5 COMPOUND type. It is designed to correspond to the HDF5 HOFFSET macro. For more details about the operation of the HOFFSET macro, please consult the HDF5 documentation.

Function parameters:

`offset`	The byte offset of the field within the structure.
`structure`	The structure which contains the specified fieldname.
`fieldname`	The field for which the offset is determined.

Examples

This function is deprecated. It can only be used in workflows that do not include a field that is itself an HDF5 COMPOUND or of variable length. To handle these cases, the offsets should be computed directly. For example, in the case above, a file dataspace for such a compound could be created with:

```
dtype(1) = H5T.copy('H5T_NATIVE_INT');
dtype(2) = H5T.copy('H5T_NATIVE_DOUBLE');
```

```
dtype(3) = H5T.copy('H5T_NATIVE_FLOAT');

for j = 1:3, sz(j,1) = H5T.get_size(dtype(j)); end

% The first offset would always be zero and the size of the last
% field does not matter.
offset(1) = 0;
offset(2:3) = cumsum(sz(1:2));

file_type = H5T.create('H5T_COMPOUND',sum(sz));

H5T.insert(file_type,'a', offset(1), dtype(1));
H5T.insert(file_type,'b', offset(2), dtype(2));
H5T.insert(file_type,'c', offset(3), dtype(3));
```

See Also

H5T.get_size

H5ML.sizeof

Return the size (in bytes) of a variable as stored on disk

Note `H5ML.sizeof` is not recommended. Use `H5T` instead.

Syntax

Description

This function is used to determine the size (in bytes) of a structure or other (simple) variable. It is designed to correspond to the C sizeof() operator as it is used during the creation of HDF5 datatypes, especially the HDF5 COMPOUND type.

Function parameters:

`size`	The size (in bytes) of the variable as it would be stored on disk.
`arg`	The variable for which the size is being sought.

Examples

This function is deprecated. It can only be used in workflows that do not include a field that is itself an HDF5 COMPOUND or of variable length. To handle these cases, the offsets should be computed directly. For example, in the case above, a file dataspace for such a compound could be created with:

```
dtype(1) = H5T.copy('H5T_NATIVE_INT');
dtype(2) = H5T.copy('H5T_NATIVE_DOUBLE');
dtype(3) = H5T.copy('H5T_NATIVE_FLOAT');
```

```
for j = 1:3, sz(j,1) = H5T.get_size(dtype(j)); end

% The first offset would always be zero and the size of the last
% field does not matter.
offset(1) = 0;
offset(2:3) = cumsum(sz(1:2));

file_type = H5T.create('H5T_COMPOUND',sum(sz));

H5T.insert(file_type,'a', offset(1), dtype(1));
H5T.insert(file_type,'b', offset(2), dtype(2));
H5T.insert(file_type,'c', offset(3), dtype(3));
```

See Also

H5T.get_size

H5O.close

Close object

Syntax

```
H5O.close(obj_id)
```

Description

`H5O.close(obj_id)` closes the object specified by `obj_id`. `obj_id` cannot be a dataspace, attribute, property list, or file.

H5O.copy

Copy object from source location to destination location

Syntax

```
H5O.copy(src_loc_id,src_name,dst_loc_id,dst_name,ocpypl_id,lcpl_id)
```

Description

`H5O.copy(src_loc_id,src_name,dst_loc_id,dst_name,ocpypl_id,lcpl_id)` copies the group, dataset or named datatype specified by `src_name` from the file or group specified by `src_loc_id` to the destination location `dst_loc_id`.

The destination location, as specified in `dst_loc_id`, may be a group in the current file or a location in a different file. If `dst_loc_id` is a file identifier, the copy will be placed in that file's root group.

The new copy will be created with the name `dst_name`. `dst_name` must not pre-exist in the destination location. If `dst_name` already exists at the location `dst_loc_id`, the operation will fail.

The new copy of the object is created with the object creation property and link creation property lists `ocpypl_id` and `lcpl_id`, respectively.

Examples

Copy the group `'/g3'` and all its datasets to a new group `'/g3.5'`.

```
srcFile = [matlabroot '/toolbox/matlab/demos/example.h5'];
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
ocpl = H5P.create('H5P_OBJECT_COPY');
lcpl = H5P.create('H5P_LINK_CREATE');
H5P.set_create_intermediate_group(lcpl,true);
fid = H5F.open('myfile.h5','H5F_ACC_RDWR','H5P_DEFAULT');
```

```
gid = H5G.open(fid,'/');
H5O.copy(gid,'g3',gid,'g3.5',ocpl,lcpl);
H5G.close(gid);
H5P.close(ocpl);
H5P.close(lcpl);
H5F.close(fid);
```

H5O.get_comment

Get comment for object specified by object identifier

Syntax

```
comment = H5O.get_comment(obj_id)
```

Description

`comment = H5O.get_comment(obj_id)` retrieves the comment for the object specified by `obj_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'g4/world');
comment = H5O.get_comment(dset_id);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5O.get_comment_by_name` | `H5O.set_comment` | `H5O.set_comment_by_name`

H5O.get_comment_by_name

Get comment for object specified by location and object name

Syntax

```
comment = H5O.get_comment_by_name(loc_id,name,lapl_id)
```

Description

`comment = H5O.get_comment_by_name(loc_id,name,lapl_id)` retrieves a comment where a location id and name together specify the object. A link access property list can affect the outcome if a link is traversed to access the object.

Examples

```
fid = H5F.open('example.h5','H5F_ACC_RDONLY','H5P_DEFAULT');
comment = H5O.get_comment_by_name(fid,'g4/world','H5P_DEFAULT');
H5F.close(fid);
```

See Also

`H5O.get_comment` | `H5O.set_comment` | `H5O.set_comment_by_name`

H5O.get_info

Object metadata

Syntax

```
info = H5O.get_info(obj_id)
```

Description

`info = H5O.get_info(obj_id)` retrieves the metadata for an object specified by `obj_id`. For details about the object metadata, please refer to the HDF5 documentation.

Examples

Determine the number of attributes for a dataset.

```
fid = H5F.open('example.h5','H5F_ACC_RDONLY','H5P_DEFAULT');
dsetId = H5D.open(fid,'/g1/g1.1/dset1.1.1');
info = H5O.get_info(dsetId);
info.num_attrs
```

Determine the type of objects in the root group.

```
plist = 'H5P_DEFAULT';
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/');
root_info = H5G.get_info(gid);
idx_type = 'H5_INDEX_NAME';
order = 'H5_ITER_DEC';
for j = 0:root_info.nlinks-1
   obj_id = H5O.open_by_idx(fid,'/',idx_type,order,j,plist);
   obj_info = H5O.get_info(obj_id);
   switch(obj_info.type)
       case H5ML.get_constant_value('H5G_LINK')
           fprintf('Object #%d is a link.\n',j);
       case H5ML.get_constant_value('H5G_GROUP')
```

```
            fprintf('Object #%d is a group.\n',j);
        case H5ML.get_constant_value('H5G_DATASET')
            fprintf('Object #%d is a dataset.\n',j);
        case H5ML.get_constant_value('H5G_TYPE')
            fprintf('Object #%d is a named datatype.\n',j);
    end
    H5O.close(obj_id);
end
H5G.close(gid);
H5F.close(fid);
```

See Also

H5D.open | H5F.open | H5G.open | H5T.open

H5O.link

Create hard link to specified object

Syntax

```
H5O.link(obj_id,new_loc_id,new_link_name,lcpl_id,lapl_id)
```

Description

`H5O.link(obj_id,new_loc_id,new_link_name,lcpl_id,lapl_id)` creates a hard link to an object, where `new_loc_id` and `new_link_name` specify the location. `lcpl_id` and `lapl_id` are the link creation and access property lists associated with the new link.

`H5O.link` is designed to add additional structure to an existing file so that, for example, an object can be shared among multiple groups.

Examples

Create a hard link from group `'/g2'` to the dataset `'/g1/ds1'`.

```
plist_id = 'H5P_DEFAULT';
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',plist_id,plist_id);
gid1 = H5G.create(fid,'/g1',plist_id);
space_id = H5S.create_simple(1,10,[]);
ds1 = H5D.create(gid1,'ds1','H5T_NATIVE_INT',space_id,plist_id);
gid2 = H5G.create(fid,'/g2',plist_id);
H5O.link(ds1,gid2,'ds2',plist_id,plist_id);
H5D.close(ds1);
H5S.close(space_id);
H5G.close(gid2); H5G.close(gid1);
H5F.close(fid);
```

See Also

`H5L.create_hard` | `H5L.create_soft`

H5O.open

Open specified object

Syntax

```
obj_id = H5O.open(loc_id,relname,lapl_id)
```

Description

`obj_id = H5O.open(loc_id,relname,lapl_id)` opens an object specified by location identifier and relative path name. `lapl_id` is the link access property list associated with the link pointing to the object. If default link access properties are appropriate, this can be passed in as `'H5P_DEFAULT'`.

Examples

```
fid = H5F.open('example.h5');
obj_id = H5O.open(fid,'g3','H5P_DEFAULT');
H5O.close(obj_id);
H5F.close(fid);
```

See Also

`H5O.close` | `H5O.open_by_idx`

H5O.open_by_idx

Open object specified by index

Syntax

```
obj_id = H5O.open_by_idx(loc_id, group_name, idx_type, order, n,
lapl_id)
```

Description

`obj_id = H5O.open_by_idx(loc_id, group_name, idx_type, order, n, lapl_id)` opens the `n`-th object in the group specified by `loc_id` and `group_name`. `loc_id` specifies a file or group. `group_name` specifies the group relative to `loc_id` in which the object can be found.

Two parameters are used to establish the iteration: `index_type` and `order`. `index_type` specifies the type of index by which objects are ordered. Specify `index_type` as one of these character vectors or string scalars.

`'H5_INDEX_NAME'`	Alpha-numeric index on name
`'H5_INDEX_CRT_ORDER'`	Index on creation order

`order` specifies the order in which the links are to be referenced for the purposes of this function. Specify `order` as one of these character vectors or string scalars.

`'H5_ITER_INC'`	Increasing order
`'H5_ITER_DEC'`	Decreasing order
`'H5_ITER_NATIVE'`	Fastest available order

`n` specifies the zero-based position of the object within the index. `lapl_id` specifies the link access property list to be used in accessing the object.

Examples

```
fid = H5F.open('example.h5');
idx_type = 'H5_INDEX_NAME';
order = 'H5_ITER_DEC';
obj_id = H5O.open_by_idx(fid,'g3',idx_type,order,0,'H5P_DEFAULT');
H5O.close(obj_id);
H5F.close(fid);
```

See Also

H5O.close | H5O.open

H5O.set_comment

Set comment for object specified by object identifier

Syntax

```
H5O.set_comment(obj_id,comment)
```

Description

`H5O.set_comment(obj_id,comment)` sets a comment for the object specified by `obj_id`.

Examples

```
plist = 'H5P_DEFAULT';
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',plist,plist);
gid = H5G.create(fid,'/g1',plist);
H5O.set_comment(gid,'This is a group comment.');
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5O.get_comment` | `H5O.get_comment_by_name` | `H5O.set_comment_by_name`

H5O.set_comment_by_name

Set comment for object specified by location and object name

Syntax

```
H5O.set_comment_by_name(loc_id,rel_name,comment,lapl_id)
```

Description

`H5O.set_comment_by_name(loc_id,rel_name,comment,lapl_id)` sets a comment for an object specified by a location ID and a relative name. `lapl_id` is a link access property list identifier that can affect the outcome if links are traversed.

Examples

```
plist = 'H5P_DEFAULT';
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',plist,plist);
gid = H5G.create(fid,'/g1',plist);
H5O.set_comment_by_name(fid,'g1','This is a group comment.',plist);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5O.get_comment` | `H5O.get_comment_by_name` | `H5O.set_comment`

H5O.visit

Visit objects specified by object identifier

Syntax

```
[status,opdata_out] =
H5O.visit(obj_id,index_type,order,iter_func,opdata_in)
```

Description

`[status,opdata_out] = H5O.visit(obj_id,index_type,order,iter_func,opdata_in)` is a recursive iteration function to visit the object `object_id` and, if `object_id` is a group, all objects in and below it in an HDF5 file. This provides a mechanism for an application to perform a common set of operations across all of those objects or a dynamically selected subset.

If `object_id` is a group identifier, that group serves as the root of a recursive iteration. If `object_id` is a file identifier, that file's root group serves as the root of the recursive iteration. If `object_id` is any other type of object, such as a dataset or named data type, there is no iteration.

Two parameters are used to establish the iteration: `index_type` and `order`. The `index_type` parameter specifies the index used. If the links in a group have not been indexed by the index type, they are first sorted by that index, and then the iteration will begin. If the links have been so indexed, the sorting step is unnecessary, so the iteration can begin more quickly. Specify `index_type` as one of these character vectors or string scalars.

`'H5_INDEX_NAME'`	Alpha-numeric index on name
`'H5_INDEX_CRT_ORDER'`	Index on creation order

Note that the index type passed in `index_type` is a best effort setting. If the application passes in a value indicating iteration in creation order and a group is encountered that was not tracked in creation order, that group will be iterated over in alpha-numeric order by name, or name order. (Name order is the native order used by the HDF5 Library and is

always available.) `order` specifies the order in which objects are to be inspected along the index specified in `index_type`. Specify `order` as one of these character vectors or string scalars.

`'H5_ITER_INC'`	Increasing order
`'H5_ITER_DEC'`	Decreasing order
`'H5_ITER_NATIVE'`	Fastest available order

The callback function `iter_func` must have the following signature:

```
function [status,opdata_out] = iter_func(group_id,name,opdata_in)
```

`opdata_in` is a user-defined value or structure and is passed to the first step of the iteration in the `iter_func` `opdata_in` parameter. The `opdata_out` of an iteration step forms the `opdata_in` for the next iteration step. The final `opdata_out` at the end of the iteration is then returned to the caller as `opdata_out`.

`status` value returned by `iter_func` is interpreted as follows.

zero	Continues with the iteration or returns zero status value to the caller if all members have been processed
positive	Stops the iteration and returns the positive status value to the caller
negative	Stops the iteration and throws an error indicating failure

See Also

`H5O.visit_by_name`

H5O.visit_by_name

Visit objects specified by location and object name

Syntax

```
[status,opdata_out] =
H5O.visit_by_name(loc_id,obj_name,index_type,order,iter_func,opdata_
in,lapl_id)
```

Description

`[status,opdata_out] = H5O.visit_by_name(loc_id,obj_name,index_type,order,iter_func,opdata_in,lapl_id)` specifies the object by the pairing of the location identifier and object name. `loc_id` specifies a file or an object in a file and `obj_name` specifies an object in the file with either an absolute name or relative to `loc_id`. A link access property list can affect the outcome if links are involved.

Two parameters are used to establish the iteration: `index_type` and `order`. The `index_type` parameter specifies the index to be used. If the links in a group have not been indexed by the index type, they are first sorted by that index, and then the iteration will begin; if the links have been so indexed, the sorting step is unnecessary, so the iteration can begin more quickly. Specify `index_type` as one of these character vectors or string scalars.

`'H5_INDEX_NAME'`	Alpha-numeric index on name
`'H5_INDEX_CRT_ORDER'`	Index on creation order

Note that the index type passed in `index_type` is a best effort setting. If the application passes in a value indicating iteration in creation order and a group is encountered that was not tracked in creation order, that group will be iterated over in alpha-numeric order by name, or name order. (Name order is the native order used by the HDF5 Library and is always available.) `order` specifies the order in which objects are to be inspected along the index specified in `index_type`. Specify `order` as one of these character vectors or string scalars.

`'H5_ITER_INC'`	Increasing order
`'H5_ITER_DEC'`	Decreasing order
`'H5_ITER_NATIVE'`	Fastest available order

The callback function `iter_func` must have the following signature:

```
function [status,opdata_out] = iter_func(group_id,name,opdata_in)
```

`opdata_in` is a user-defined value or structure and is passed to the first step of the iteration in the `iter_func` `opdata_in` parameter. The `opdata_out` of an iteration step forms the `opdata_in` for the next iteration step. The final `opdata_out` at the end of the iteration is then returned to the caller as `opdata_out`.

`lapl_id` is a link access property list. When default link access properties are acceptable, `'H5P_DEFAULT'` can be used.

`status` value returned by `iter_func` is interpreted as follows.

zero	Continues with the iteration or returns zero status value to the caller if all members have been processed
positive	Stops the iteration and returns the positive status value to the caller
negative	Stops the iteration and throws an error indicating failure

See Also

`H5O.visit`

H5P.close

Close property list

Syntax

```
H5P.close(plist_id)
```

Description

`H5P.close(plist_id)` terminates access to the property list specified by `plist_id`.

See Also

`H5P.create`

H5P.copy

Copy of property list

Syntax

```
plist_copy = H5P.copy(plist_id)
```

Description

`plist_copy = H5P.copy(plist_id)` returns a copy of the property list specified by `plist_id`.

Examples

Make a copy of the file creation property list for `'example.h5'`.

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
fcpl2 = H5P.copy(fcpl);
```

H5P.create

Create new property list

Syntax

```
plist = H5P.create(class_id)
```

Description

`plist = H5P.create(class_id)` creates a new property list as an instance of the property list class specified by `class_id`. Specify `class_id` as one of the following character vectors or string scalars, or their corresponding constant value.

```
'H5P_ATTRIBUTE_CREATE'
'H5P_DATASET_ACCESS'
'H5P_DATASET_CREATE'
'H5P_DATASET_XFER'
'H5P_DATATYPE_CREATE'
'H5P_DATATYPE_ACCESS'
'H5P_FILE_MOUNT'
'H5P_FILE_CREATE'
'H5P_FILE_ACCESS'
'H5P_GROUP_CREATE'
'H5P_GROUP_ACCESS'
'H5P_LINK_CREATE'
'H5P_LINK_ACCESS'
'H5P_OBJECT_COPY'
'H5P_OBJECT_CREATE'
'H5P_STRING_CREATE'
```

`class_id` can also be an instance of a property list class.

Examples

```
fapl = H5P.create('H5P_FILE_ACCESS');
fid = H5F.open('example.h5','H5F_ACC_RDONLY',fapl);
```

See Also

H5ML.get_constant_value | H5P.close | H5P.get_class

H5P.get_class

Property list class

Syntax

```
plist_class = H5P.get_class(plist_id)
```

Description

`plist_class = H5P.get_class(plist_id)` returns the property list class for the property list specified by `plist_id`.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
pclass = H5P.get_class(fcpl);
name = H5P.get_class_name(pclass);
```

See Also

`H5P.get_class_name`

H5P.close_class

Close property list class

Syntax

```
H5P.close_class(class)
```

Description

`H5P.close_class(class)` closes the property list class specified by `pclass_id`.

H5P.equal

Determine equality of property lists

Syntax

```
value = H5P.equal(plist1_id, plist2_id)
```

Description

`value = H5P.equal(plist1_id, plist2_id)` returns a positive number if the two property lists specified are equal, and zero if they are not. A negative value indicates failure.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
fcpl = H5F.get_create_plist(fid);
if H5P.equal(fapl,fcpl)
    fprintf('property lists are equal\n');
else
    fprintf('property lists are not equal\n');
end
```

H5P.exist

Determine if specified property exists in property list

Syntax

```
value = H5P.exist(prop_id,name)
```

Description

`value = H5P.exist(prop_id,name)` returns a positive value if the property specified by `name` exists within the property list or class specified by `prop_id`. Specify `name` as a character vector or string scalar.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
if H5P.exist(fapl,'sieve_buf_size')
    fprintf('sieve buffer size property exists\n');
else
    fprintf('sieve buffer size property does not exist\n');
end
```

H5P.get

Value of specified property in property list

Syntax

```
value = H5P.get(plist_id,name)
```

Description

`value = H5P.get(plist_id,name)` retrieves a copy of the value of the property specified by `name` in the property list specified by `plist_id`. Specify `name` as a character vector or string scalar. The `H5P.get` function returns the property as an array of `uint8` values. You might need to cast the value to an appropriate data type to get a meaningful result.

Examples

```
plist = H5P.create('H5P_FILE_ACCESS');
val = H5P.get(plist, 'rdcc_w0');
rdcc_w0 = typecast(val,'double');
```

It is recommended to use alternative functions like `H5P.get_chunk`, `H5P.get_layout`, `H5P.get_size` etc., where available, to get values for the common property names.

See Also

`H5P.set` | `typecast`

H5P.get_class_name

Name of property list class

Syntax

```
classname = H5P.get_class_name(pclass_id)
```

Description

`classname = H5P.get_class_name(pclass_id)` retrieves the name of the generic property list class. `classname` is a character vector. If no class is found, an empty character vector is returned.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
pclass = H5P.get_class(fcpl);
name = H5P.get_class_name(pclass);
```

See Also

`H5P.get_class`

H5P.get_class_parent

Identifier for parent class

Syntax

```
pclass_obj_id = H5P.get_class_parent(pclass_id)
```

Description

`pclass_obj_id = H5P.get_class_parent(pclass_id)` returns an identifier to the parent class object of the property class specified by `pclass_id`.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
fcpl_class = H5P.get_class(fcpl);
parent_class = H5P.get_class_parent(fcpl_class);
name = H5P.get_class_name(parent_class);
```

See Also

`H5P.get_class` | `H5P.get_class_name`

H5P.get_nprops

Query number of properties in property list or class

Syntax

```
nprops = H5P.get_nprops(id)
```

Description

`nprops = H5P.get_nprops(id)` returns the number of properties in the property list or class specified by `id`.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
nprops = H5P.get_nprops(fcpl);
```

H5P.get_size

Query size of property value in bytes

Syntax

```
sz = H5P.get_size(id,name)
```

Description

`sz = H5P.get_size(id,name)` returns the size, in bytes, of the property specified by `name` in the property list or property class specified by `id`. Specify `name` as a character vector or string scalar.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
sz = H5P.get_size(fapl,'sieve_buf_size');
```

H5P.isa_class

Determine if property list is member of class

Syntax

```
output = H5P.isa_class(plist_id, pclass_id)
```

Description

`output = H5P.isa_class(plist_id, pclass_id)` returns a positive number if the property list specified by `plist_id` is a member of the class specified by `pclass_id`, zero if it is not, and a negative value to indicate an error.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
if H5P.isa_class(fcpl,'H5P_FILE_ACCESS')
    fprintf('fcpl is a file access property list\n');
else
    fprintf('fcpl is not a file access property list\n');
end
```

See Also

`H5P.get_class`

H5P.iterate

Iterate over properties in property list

Syntax

```
[output,idx_out] = H5P.iterate(id,idx_in,iter_func)
```

Description

`[output,idx_out] = H5P.iterate(id,idx_in,iter_func)` executes the operation `iter_func` on each property in the property object specified in `id`. The `id` input can be a property list or a property class. `idx_in` specifies the index of the next property to be processed. `output` is the value returned by the last call to `iter_func`. `idx_out` is the index of the last property processed. `iter_func` is a function handle.

The iterator function must have the following signature:

```
status = iter_func(id,prop_name)
```

`id` still identifies the property object passed into `H5P.iterate`, but `name` identifies the name of the current property.

H5P.set

Set property list value

Syntax

```
H5P.set(plist_id,name,value)
```

Description

`H5P.set(plist_id,name,value)` sets the value of the property specified by `name` in the property list specified by `plist_id` to the value specified in `value`. Specify `name` as a character vector or string scalar. The datatype of `value` must be `uint8`.

Examples

```
plist = H5P.create('H5P_FILE_ACCESS');
H5P.set(plist, 'rdcc_w0', typecast(0.8, 'uint8'));
```

It is recommended to use alternative functions like `H5P.set_chunk`, `H5P.set_layout`, `H5P.set_size`, etc., where available, to set values for the common property names.

See Also

`typecast`

H5P.get_btree_ratios

B-tree split ratios

Syntax

```
[left,middle,right] = H5P.get_btree_ratios(plist_id)
```

Description

`[left,middle,right] = H5P.get_btree_ratios(plist_id)` returns the B-tree split ratios for the dataset transfer property list specified by `plist_id`. The `left` output specifies the B-tree split ratio for left-most nodes. `right` corresponds to the right-most nodes and lone nodes, and `middle` corresponds to all other nodes.

Examples

```
dxpl = H5P.create('H5P_DATASET_XFER');
[left,middle,right] = H5P.get_btree_ratios(dxpl);
```

See Also

`H5P.set_btree_ratios`

H5P.get_chunk_cache

Raw data chunk cache parameters

Syntax

```
[rdcc_nslots,rdcc_nbytes,rdcc_w0] = H5P.get_chunk_cache(dapl_id)
```

Description

`[rdcc_nslots,rdcc_nbytes,rdcc_w0] = H5P.get_chunk_cache(dapl_id)` retrieves the number of chunk slots in the raw data chunk cache hash table (`rdcc_nslots`), the maximum possible number of bytes in the raw data chunk cache (`rdcc_nbytes`), and the preemption policy value (`rdcc_w0`) on a dataset access property list.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/vlen3D');
dapl = H5D.get_access_plist(dset_id);
[rrdcc_nslots,rdcc_nbytes,rdcc_w0] = H5P.get_chunk_cache(dapl);
H5P.close(dapl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.set_chunk_cache`

H5P.get_edc_check

Determine if error detection is enabled

Syntax

```
check = H5P.get_edc_check(plist_id)
```

Description

`check = H5P.get_edc_check(plist_id)` queries the dataset transfer property list, specified by `plist_id`, to determine whether error detection is enabled for data read operations. Returns either `H5Z_ENABLE_EDC` or `H5Z_DISABLE_EDC`.

Examples

```
dxpl = H5P.create('H5P_DATASET_XFER');
check = H5P.get_edc_check(dxpl);
switch(check)
    case H5ML.get_constant_value('H5Z_ENABLE_EDC')
        fprintf('error detection enabled\n');
    case H5ML.get_constant_value('H5Z_DISABLE_EDC');
        fprintf('error detection disabled\n');
end
```

See Also

`H5P.set_edc_check`

H5P.get_hyper_vector_size

Number of I/O vectors

Syntax

```
sz = H5P.get_hyper_vector_size(dxpl_id)
```

Description

`sz = H5P.get_hyper_vector_size(dxpl_id)` returns the number of I/O vectors to be read/written in hyperslab I/O.

Examples

```
dxpl = H5P.create('H5P_DATASET_XFER');
sz = H5P.get_hyper_vector_size(dxpl);
```

See Also

`H5P.set_hyper_vector_size`

H5P.set_btree_ratios

Set B-tree split ratios for dataset transfer

Syntax

```
H5P.set_btree_ratios(plist_id,left,middle,right)
```

Description

`H5P.set_btree_ratios(plist_id,left,middle,right)` sets the B-tree split ratios for the dataset transfer property list specified by `plist_id`. The `left` argument specifies the B-tree split ratio for left-most nodes. `right` specifies the B-tree split ratio for right-most nodes and lone nodes. `middle` specifies the B-tree split ratio for all other nodes.

Examples

```
dxpl = H5P.create('H5P_DATASET_XFER');
H5P.set_btree_ratios(dxpl, 0.2, 0.6, 0.95);
```

See Also

`H5P.get_btree_ratios`

H5P.set_chunk_cache

Set raw data chunk cache parameters

Syntax

```
H5P.set_chunk_cache(dapl_id, rdcc_nslots, rdcc_nbytes, rdcc_w0)
```

Description

`H5P.set_chunk_cache(dapl_id, rdcc_nslots, rdcc_nbytes, rdcc_w0)` sets the number of elements (`rdcc_nslots`), the total number of bytes (`rdcc_nbytes`), and the preemption policy value (`rdcc_w0`) in the raw data chunk cache.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/vlen3D');
dapl = H5D.get_access_plist(dset_id);
H5P.set_chunk_cache(dapl,500,1e6,0.7);
H5P.close(dapl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.get_chunk_cache`

H5P.set_edc_check

Enable error detection for dataset transfer

Syntax

```
H5P.set_edc_check(plist_id,check)
```

Description

`H5P.set_edc_check(plist_id,check)` sets the dataset transfer property list specified by `plist_id` to enable or disable error detection when reading data. Specify `check` as a character vector or string scalar containing `'H5Z_ENABLE_EDC'` or `'H5Z_DISABLE_EDC'`.

Examples

Disable error detection for a default dataset transfer property list.

```
dxpl = H5P.create('H5P_DATASET_XFER');
H5P.set_edc_check(dxpl,'H5Z_DISABLE_EDC');
```

See Also

`H5P.get_edc_check`

H5P.set_hyper_vector_size

Set number of I/O vectors for hyperslab I/O

Syntax

```
H5P.set_hyper_vector_size(dxpl_id,size)
```

Description

`H5P.set_hyper_vector_size(dxpl_id,size)` sets the number of I/O vectors to be accumulated in memory before being issued to the lower levels of the HDF5 library for reading or writing the actual data. `dxpl_id` is a dataset transfer property list identifier. `size` specifies the number of I/O vectors to accumulate in memory for I/O operations.

Examples

```
dxpl = H5P.create('H5P_DATASET_XFER');
H5P.set_hyper_vector_size(dxpl,2048);
```

See Also

`H5P.get_hyper_vector_size`

H5P.all_filters_avail

Determine availability of all filters

Syntax

```
value = H5P.all_filters_avail(dcpl_id)
```

Description

`value = H5P.all_filters_avail(dcpl_id)` returns a positive value if all of the filters set in the dataset creation property list `dcpl_id` are currently available, and zero if they are not. A negative value indicates failure.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/float');
dcpl = H5D.get_create_plist(dset_id);
if H5P.all_filters_avail(dcpl)
    fprintf('all filters available\n');
else
    fprintf('all filters not available\n');
end
H5P.close(dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.set_filter`

H5P.fill_value_defined

Determine if fill value is defined

Syntax

```
fvstatus = H5P.fill_value_defined(plist_id)
```

Description

`fvstatus = H5P.fill_value_defined(plist_id)` determines whether a fill value is defined in the dataset creation property list `plist_id`. The `fvstatus` output can have any of the following values: `H5D_FILL_VALUE_UNDEFINED`, `H5D_FILL_VALUE_DEFAULT`, or `H5D_FILL_VALUE_USER_DEFINED`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/float');
dcpl = H5D.get_create_plist(dset_id);
fvstatus = H5P.fill_value_defined(dcpl);
switch(fvstatus)
    case H5ML.get_constant_value('H5D_FILL_VALUE_UNDEFINED')
        fprintf('fill value undefined\n');
    case H5ML.get_constant_value('H5D_FILL_VALUE_DEFAULT')
        fprintf('fill value set to default\n');
    case H5ML.get_constant_value('H5D_FILL_VALUE_USER_DEFINED')
        fprintf('fill value is user defined\n');
end
```

See Also

`H5P.get_fill_value` | `H5P.set_fill_value`

H5P.get_alloc_time

Return timing of storage space allocation

Syntax

```
alloc_time = H5P.get_alloc_time(plist_id)
```

Description

`alloc_time = H5P.get_alloc_time(plist_id)` retrieves the timing for storage space allocation from the dataset creation property list specified by `plist_id`. The `alloc_time` output can have any of the following values: `H5D_ALLOC_TIME_DEFAULT`, `H5D_ALLOC_TIME_EARLY`, `H5D_ALLOC_TIME_INCR`, or `H5D_ALLOC_TIME_LATE`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
dcpl = H5D.get_create_plist(dset_id);
alloc_time = H5P.get_alloc_time(dcpl);
switch(alloc_time)
    case H5ML.get_constant_value('H5D_ALLOC_TIME_DEFAULT')
        fprintf('allocation time is default\n');
    case H5ML.get_constant_value('H5D_ALLOC_TIME_EARLY')
        fprintf('allocation time is dataset creation time\n');
    case H5ML.get_constant_value('H5D_ALLOC_TIME_INCR')
        fprintf('allocation time is incremental\n');
    case H5ML.get_constant_value('H5D_ALLOC_TIME_LATE')
        fprintf('allocation time is when data is first written\n');
end
```

H5P.get_chunk

Return size of chunks

Syntax

```
[rank,h5_chunk_dims] = H5P.get_chunk(plist_id)
```

Description

`[rank,h5_chunk_dims] = H5P.get_chunk(plist_id)` retrieves the size of chunks for the raw data of a chunked layout dataset for the dataset creation property list specified by `plist_id`.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_chunk_dims` parameter assumes C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g4/time');
dcpl = H5D.get_create_plist(dset_id);
[rank,chunk_dims] = H5P.get_chunk(dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.set_chunk`

H5P.get_external

Return information about external file

Syntax

```
[name,offset,size] = H5P.get_external(plist_id,idx)
```

Description

`[name,offset,size] = H5P.get_external(plist_id,idx)` returns information about the external file specified by the dataset creation property list `plist_id`. The `idx` specifies the external file index, which is a number from zero to `N-1`, where `N` is the value returned by `H5P.get_external_count`. The `name` output returns the name of the external file (limited by 2048 characters). The `offset` output returns the location in bytes, from the beginning of the external file, where the data starts. The `size` output returns the size of the external data.

See Also

`H5P.get_external_count`

H5P.get_external_count

Return count of external files

Syntax

```
num_files = H5P.get_external_count(plist_id)
```

Description

`num_files = H5P.get_external_count(plist_id)` returns the number of external files for the dataset creation property list, `plist_id`.

See Also

`H5P.get_external`

H5P.get_fill_time

Return time when fill values are written to dataset

Syntax

```
fill_time = H5P.get_fill_time(plist_id)
```

Description

`fill_time = H5P.get_fill_time(plist_id)` returns the time when fill values are written to the dataset specified by the dataset creation property list `plist_id`. The `fill_time` output is one of the following values: `H5D_FILL_TIME_IFSET`, `H5D_FILL_TIME_ALLOC`, or `H5D_FILL_TIME_NEVER`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
dcpl = H5D.get_create_plist(dset_id);
fill_time = H5P.get_fill_time(dcpl);
switch(fill_time)
    case H5ML.get_constant_value('H5D_FILL_TIME_IFSET')
        fprintf('upon allocation if and only if fill value set by user\n');
    case H5ML.get_constant_value('H5D_FILL_TIME_ALLOC')
        fprintf('written when storage space is allocated\n');
    case H5ML.get_constant_value('H5D_FILL_TIME_NEVER')
        fprintf('fill values are never written\n');
end
```

See Also

`H5P.get_fill_time` | `H5P.set_fill_value`

H5P.get_fill_value

Return dataset fill value

Syntax

```
value = H5P.get_fill_value(plist_id,type_id)
```

Description

`value = H5P.get_fill_value(plist_id,type_id)` returns the dataset fill value defined in the dataset creation property list `plist_id`. The `type_id` input specifies the datatype of the returned fill value.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
dcpl = H5D.get_create_plist(dset_id);
type_id = H5T.copy('H5T_NATIVE_INT');
fill_value = H5P.get_fill_value(dcpl,type_id);
```

See Also

`H5P.get_fill_time` | `H5P.set_fill_time` | `H5P.set_fill_value`

H5P.get_filter

Return information about filter in pipeline

Syntax

```
[filter,flags,cd_values,name] = H5P.get_filter(plist_id,index)
[filter,flags,cd_values,name,filter_config] =
H5P.get_filter(plist_id,index)
```

Description

`[filter,flags,cd_values,name] = H5P.get_filter(plist_id,index)` returns information about the filter, specified by its filter index, in the filter pipeline, specified by the property list with which it is associated. This interface corresponds to the 1.6 version of `H5Pget_filter` in the HDF5 library.

`[filter,flags,cd_values,name,filter_config] = H5P.get_filter(plist_id,index)` returns information about the filter, specified by its filter index, in the filter pipeline, specified by the property list with which it is associated. It also returns information about the filter. Consult the HDF5 documentation for `H5Zget_filter_info` for information about `filter_config`. This interface corresponds to the 1.8 version of `H5Pget_filter` in the HDF5 library.

See Also

`H5P.get_filter_by_id` | `H5P.get_nfilters` | `H5P.modify_filter` | `H5P.remove_filter`

H5P.get_filter_by_id

Return information about specified filter

Syntax

```
[flags,cd_values,name,filter_config] =
H5P.get_filter_by_id(plist_id,idx)
```

Description

`[flags,cd_values,name,filter_config] = H5P.get_filter_by_id(plist_id,idx)` returns information about the filter specified by the filter id, `idx`.

See Also

`H5P.get_filter` | `H5P.get_nfilters` | `H5P.modify_filter` | `H5P.remove_filter`

H5P.get_layout

Determine layout of raw data for dataset

Syntax

```
layout = H5P.get_layout(dcpl)
```

Description

`layout = H5P.get_layout(dcpl)` returns the layout of the raw data for the dataset specified by the dataset creation property list, `dcpl`. Possible values are: `H5D_COMPACT`, `H5D_CONTIGUOUS`, or `H5D_CHUNKED`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
dcpl = H5D.get_create_plist(dset_id);
layout = H5P.get_layout(dcpl);
switch(layout)
    case H5ML.get_constant_value('H5D_COMPACT')
        fprintf('layout is compact\n');
    case H5ML.get_constant_value('H5D_CONTIGUOUS')
        fprintf('layout is contiguous\n');
    case H5ML.get_constant_value('H5D_CHUNKED')
        fprintf('layout is chunked\n');
end
```

See Also

`H5P.set_layout`

H5P.get_nfilters

Return number of filters in pipeline

Syntax

```
num_filters = H5P.get_nfilters(plist_id)
```

Description

`num_filters = H5P.get_nfilters(plist_id)` returns the number of filters defined in the filter pipeline associated with the dataset creation property list, `plist_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g4/world');
dcpl = H5D.get_create_plist(dset_id);
num_filters = H5P.get_nfilters(dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.get_filter` | `H5P.get_filter_by_id` | `H5P.modify_filter` | `H5P.remove_filter`

H5P.modify_filter

Modify filter in pipeline

Syntax

```
H5P.modify_filter(plist_id,filter_id,flags,cd_values)
```

Description

`H5P.modify_filter(plist_id,filter_id,flags,cd_values)` modifies the specified filter in the filter pipeline. `plist_id` is a property list identifier. `flags` is a bit vector specifying certain general properties of the filter. `cd_values` specifies auxiliary data for the filter.

See Also

`H5P.get_filter` | `H5P.get_filter_by_id` | `H5P.get_nfilters` | `H5P.remove_filter`

H5P.remove_filter

Remove filter from property list

Syntax

```
H5P.remove_filter(plist_id,filter)
```

Description

`H5P.remove_filter(plist_id,filter)` removes the specified filter from the filter pipeline. `plist_id` is the dataset creation property list identifier.

See Also

`H5P.get_filter` | `H5P.get_filter_by_id` | `H5P.get_nfilters` | `H5P.modify_filter`

H5P.set_alloc_time

Set timing for storage space allocation

Syntax

```
H5P.set_alloc_time(plist_id,alloc_time)
```

Description

`H5P.set_alloc_time(plist_id,alloc_time)` sets the timing for the allocation of storage space for a dataset's raw data. `plist_id` is a dataset creation property list. Specify `alloc_time` as one of these character vectors or string scalars: `'H5D_ALLOC_TIME_DEFAULT'`, `'H5D_ALLOC_TIME_EARLY'`, `'H5D_ALLOC_TIME_INC'`, or `'H5D_ALLOC_TIME_LATE'`.

Examples

Create a 1000x500 double precision dataset with late allocation time.

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [1000 500];
h5_dims = fliplr(dims);
h5_maxdims = h5_dims;
space_id = H5S.create_simple(2,h5_dims,h5_maxdims);
dcpl = H5P.create('H5P_DATASET_CREATE');
alloc_time = H5ML.get_constant_value('H5D_ALLOC_TIME_LATE');
H5P.set_alloc_time(dcpl,alloc_time);
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.get_alloc_time`

H5P.set_chunk

Set chunk size

Syntax

```
H5P.set_chunk(plist_id,h5_chunk_dims)
```

Description

`H5P.set_chunk(plist_id,h5_chunk_dims)` sets the size of the chunks used to store a chunked layout dataset. `plist_id` is a dataset creation property list identifier. `h5_chunk_dims` is an array specifying the size, in dataset elements, of each chunk.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_chunk_dims` parameter assumes C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

Create a two dimensional double precision dataset that has an initial size of [512 1024], but is also unlimited in both dimensions and has a chunk size of [512 1024].

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
unlimited = H5ML.get_constant_value('H5S_UNLIMITED');
dims = [512 1024];
h5_dims = fliplr(dims);
h5_maxdims = [unlimited unlimited];
space_id = H5S.create_simple(2,[1024 512],h5_maxdims);
dcpl = H5P.create('H5P_DATASET_CREATE');
chunk_dims = [512 1024];
h5_chunk_dims = fliplr(chunk_dims);
H5P.set_chunk(dcpl,h5_chunk_dims);
```

```
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

H5P.get_chunk

H5P.set_deflate

Set compression method and compression level

Syntax

```
H5P.set_deflate(plist_id,level)
```

Description

`H5P.set_deflate(plist_id,level)` sets the compression method for the dataset creation property list specified by `plist_id` to `H5D_COMPRESS_DEFLATE`. `level` specifies the compression level as a value from 0 and 9, inclusive. Lower values results in less compression.

Examples

Create a two dimensional double precision dataset that has an initial size of [512 1024], but is also unlimited in both dimensions and has a chunk size of [512 1024] and a compression level of 5.

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
unlimited = H5ML.get_constant_value('H5S_UNLIMITED');
dims = [512 1024];
h5_dims = fliplr(dims);
h5_maxdims = [unlimited unlimited];
space_id = H5S.create_simple(2,[1024 512],h5_maxdims);
dcpl = H5P.create('H5P_DATASET_CREATE');
chunk_dims = [512 1024];
h5_chunk_dims = fliplr(chunk_dims);
H5P.set_chunk(dcpl,h5_chunk_dims);
H5P.set_deflate(dcpl,5);
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

H5P.set_external

Add additional file to external file list

Syntax

```
H5P.set_external(plist_id,name,offset,nbytes)
```

Description

`H5P.set_external(plist_id,name,offset,nbytes)` adds the external file specified by `name` to the list of external files in the dataset creation property list, `plist_id`. The `offset` argument specifies the location, in bytes, where the data starts relative to the beginning of the file. `nbytes` is the number of bytes reserved in the file for the data. `nbytes` may also be given as `'H5F_UNLIMITED'`, in which case the external file may be of unlimited size.

Examples

Create a dataset with an unlimited size external file.

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [100 50];
h5_dims = fliplr(dims);
h5_maxdims = h5_dims;
space_id = H5S.create_simple(2,h5_dims,h5_maxdims);
dcpl = H5P.create('H5P_DATASET_CREATE');
H5P.set_external(dcpl,'myexternalfile.dat',0,'H5F_UNLIMITED');
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
data = rand(dims);
dxpl = 'H5P_DEFAULT';
H5D.write(dset_id,'H5ML_DEFAULT','H5S_ALL','H5S_ALL',dxpl,data);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5ML.get_constant_value` | `H5P.get_external`

H5P.set_fill_time

Set time when fill values are written to dataset

Syntax

```
H5P.set_fill_time(plist_id,fill_time)
```

Description

`H5P.set_fill_time(plist_id,fill_time)` sets the timing for writing fill values to a dataset in the dataset creation property list `plist_id`. The timing can be specified by one of the following values: `H5D_FILL_TIME_IFSET`, `H5D_FILL_TIME_ALLOC`, or `H5D_FILL_TIME_NEVER`.

Examples

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [100 50];
h5_dims = fliplr(dims);
h5_maxdims = h5_dims;
space_id = H5S.create_simple(2,h5_dims,h5_maxdims);
dcpl = H5P.create('H5P_DATASET_CREATE');
fill_time = H5ML.get_constant_value('H5D_FILL_TIME_ALLOC');
H5P.set_fill_time(dcpl,fill_time);
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.get_fill_time` | `H5P.get_fill_value` | `H5P.set_fill_value`

H5P.set_fill_value

Set fill value for dataset creation property list

Syntax

```
H5P.set_fill_value(plist_id,type_id,value)
```

Description

`H5P.set_fill_value(plist_id,type_id,value)` sets the fill value for a the dataset creation property list specified by `plist_id`. The `value` argument specifies the fill value. `type_id` specifies the datatype of the fill value. Setting `value` to an empty array indicates that the fill value is to be undefined.

Examples

Create a double precision dataset with a fill value of -999.

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [100 50];
h5_dims = fliplr(dims);
h5_maxdims = h5_dims;
space_id = H5S.create_simple(2,h5_dims,h5_maxdims);
dcpl = H5P.create('H5P_DATASET_CREATE');
fill_time = H5ML.get_constant_value('H5D_FILL_TIME_ALLOC');
H5P.set_fill_time(dcpl,fill_time);
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
H5P.set_fill_value(dcpl,type_id,-999);
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

H5P.set_filter

Add filter to filter pipeline

Syntax

```
H5P.set_filter(plist_id,filter,flags,cd_values)
```

Description

`H5P.set_filter(plist_id,filter,flags,cd_values)` adds the specified filter and corresponding properties to the end of an output filter pipeline. `plist_id` is a property list identifier. Specify `filter` as one of these character vectors or string scalars.

- `'H5P_FILTER_DEFLATE'`
- `'H5P_FILTER_SHUFFLE'`
- `'H5P_FILTER_FLETCHER32'`

`flags` is a bit vector specifying properties of the filter. `cd_values` is an array that contains auxiliary data for the filter.

See Also

`H5P.set_deflate` | `H5P.set_fletcher32` | `H5P.set_shuffle`

H5P.set_fletcher32

Set Fletcher32 checksum filter in dataset creation

Syntax

```
H5P.set_fletcher32(plist_id)
```

Description

`H5P.set_fletcher32(plist_id)` sets the Fletcher32 checksum filter in the dataset creation property list specified by `plist_id`. The dataset creation property list must also have chunking enabled.

Examples

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,dims,[]);
dcpl = H5P.create('H5P_DATASET_CREATE');
chunk_dims = [10 20];
h5_chunk_dims = fliplr(chunk_dims);
H5P.set_chunk(dcpl,h5_chunk_dims);
H5P.set_fletcher32(dcpl);
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.set_deflate` | `H5P.set_shuffle`

H5P.set_layout

Set type of storage for dataset

Syntax

```
H5P.set_layout(dcpl,layout)
```

Description

`H5P.set_layout(dcpl,layout)` sets the type of storage used to store the raw data for the dataset creation property list, `dcpl`. Specify the type of storage layout for raw data `layout` as one of these character vectors or string scalars: `'H5D_COMPACT'`, `'H5D_CONTIGUOUS'`, or `'H5D_CHUNKED'`.

Examples

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,dims,[]);
dcpl = H5P.create('H5P_DATASET_CREATE');
layout = H5ML.get_constant_value('H5D_CONTIGUOUS');
H5P.set_layout(dcpl,layout);
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.get_layout` | `H5P.set_chunk`

H5P.set_nbit

Set N-Bit filter

Syntax

```
H5P.set_nbit(plist_id)
```

Description

`H5P.set_nbit(plist_id)` sets the N-Bit filter, `H5Z_FILTER_NBIT`, in the dataset creation property list `plist_id`.

H5P.set_scaleoffset

Set Scale-Offset filter

Syntax

```
H5P.set_scaleoffset(plistId,scaleType,scaleFactor)
```

Description

`H5P.set_scaleoffset(plistId,scaleType,scaleFactor)` sets the Scale-Offset filter, `H5Z_FILTER_SCALEOFFSET`, for a dataset. For integer data types, the parameter `scaleType` should be set to the enumerated value `H5Z_SO_INT`. For floating-point data types, the `scaleType` should be the enumerated value `H5Z_SO_FLOAT_DSCALE`. Chunking must already be enabled on the dataset creation property list.

See Also

`H5P.set_chunk`

H5P.set_shuffle

Set shuffle filter

Syntax

```
H5P.set_shuffle(plist_id)
```

Description

`H5P.set_shuffle(plist_id)` sets the shuffle filter, `H5Z_FILTER_SHUFFLE`, in the dataset creation property list `plist_id`. Compression must be enabled on the dataset creation property list in order to use the shuffle filter, and best results are usually obtained when the shuffle filter is set immediately prior to setting the deflate filter.

Examples

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,dims,[]);
dcpl = H5P.create('H5P_DATASET_CREATE');
chunk_dims = [10 20];
h5_chunk_dims = fliplr(chunk_dims);
H5P.set_chunk(dcpl,h5_chunk_dims);
H5P.set_shuffle(dcpl);
H5P.set_deflate(dcpl,5);
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.set_deflate`

H5P.get_alignment

Retrieve alignment properties

Syntax

```
[threshold,alignment] = H5P.get_alignment(plist_id)
```

Description

`[threshold,alignment] = H5P.get_alignment(plist_id)` retrieves the current settings for alignment properties from the file access property list specified by `plist_id`.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
[threshold,alignment] = H5P.get_alignment(fapl);
H5P.close(fapl);
H5F.close(fid);
```

H5P.get_driver

Low-level file driver

Syntax

```
driver_id = H5P.get_driver(plist_id)
```

Description

`driver_id = H5P.get_driver(plist_id)` returns the identifier of the low-level file driver associated with the file access property list or data transfer property list specified by `plist_id`. See HDF5 documentation for a list of valid return values.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
driver_id = H5P.get_driver(fapl);
if ( driver_id == H5ML.get_constant_value('H5FD_SEC2') )
    fprintf('File driver is H5FD_SEC2.\n');
end
H5P.close(fapl);
H5F.close(fid);
```

See Also

`H5ML.get_constant_value`

H5P.get_family_offset

Offset for family file driver

Syntax

```
offset = H5P.get_family_offset(fapl_id)
```

Description

`offset = H5P.get_family_offset(fapl_id)` retrieves the value of offset from the file access property list, `fapl_id`. `offset` is the offset of the data in the HDF5 file that is stored on disk in the selected member file in a family of files.

See Also

`H5P.set_family_offset`

H5P.get_fapl_core

Information about core file driver properties

Syntax

```
[increment,backing_store] = H5P.get_fapl_core(fapl_id)
```

Description

`[increment,backing_store] = H5P.get_fapl_core(fapl_id)` queries the `H5FD_CORE` driver properties as set by `H5P.set_fapl_core`. The `fapl_id` argument specifies a file access property list. The return value `increment` specifies the size, in bytes, of memory increments. `backing_store` is a Boolean flag indicating whether to write the file contents to disk when the file is closed.

See Also

`H5P.set_fapl_core`

H5P.get_fapl_family

File access property list information

Syntax

```
[memb_size,memb_fapl_id] = H5P.get_fapl_family(fapl_id)
```

Description

`[memb_size,memb_fapl_id] = H5P.get_fapl_family(fapl_id)` returns the size in bytes of each file member and the identifier of the file access property list for use with the family driver specified by `fapl_id`.

See Also

`H5P.set_fapl_family`

H5P.get_fapl_multi

Information about multifile access property list

Syntax

```
[memb_map,memb_fapl,memb_name,memb_addr,relax] =
H5P.get_fapl_multi(fapl_id)
```

Description

`[memb_map,memb_fapl,memb_name,memb_addr,relax] = H5P.get_fapl_multi(fapl_id)` returns information about the multifile access property list specified by `fapl_id`. The `memb_map` output maps memory usage types to other memory usage types. `memb_fapl` is a property list for each memory usage type. `memb_name` is the name generator for names of member files. `relax` is a Boolean value that, when non-zero, allows read-only access to incomplete file sets.

See Also

`H5P.set_fapl_multi`

H5P.get_fclose_degree

File close degree

Syntax

```
degree = H5P.get_fclose_degree(fapl_id)
```

Description

`degree = H5P.get_fclose_degree(fapl_id)` returns the current setting of the file close degree property `fc_degree` in the file access property list specified by `fapl_id`. Possible return values are: `H5F_CLOSE_DEFAULT`, `H5F_CLOSE_WEAK`, `H5F_CLOSE_SEMI`, or `H5F_CLOSE_STRONG`.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
degree = H5P.get_fclose_degree(fapl);
switch(degree)
    case H5ML.get_constant_value('H5F_CLOSE_DEFAULT')
        fprintf('file close degree is default\n');
    case H5ML.get_constant_value('H5F_CLOSE_WEAK')
        fprintf('file close degree is weak\n');
    case H5ML.get_constant_value('H5F_CLOSE_SEMI')
        fprintf('close degree is semi\n');
    case H5ML.get_constant_value('H5F_CLOSE_STRONG')
        fprintf('close degree is strong\n');
end
H5P.close(fapl);
H5F.close(fid);
```

See Also

`H5P.set_fclose_degree`

H5P.get_libver_bounds

Library version bounds settings

Syntax

```
[low,high] = H5P.get_libver_bounds(fapl_id)
```

Description

`[low,high] = H5P.get_libver_bounds(fapl_id)` gets bounds on library version bounds settings that control the format versions used when creating objects in the file with access property list `fapl_id`.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
[low,high] = H5P.get_libver_bounds(fapl);
```

See Also

`H5F.get_access_plist` | `H5P.set_libver_bounds`

H5P.get_gc_references

Garbage collection references setting

Syntax

```
gc_ref = H5P.get_gc_references(fapl_id)
```

Description

`gc_ref = H5P.get_gc_references(fapl_id)` returns the current setting for the garbage collection references property from the file access property list specified by `fapl_id`. If `gc_ref` is `1`, garbage collection is on; if `0`, garbage collection is off.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
gc_ref = H5P.get_gc_references(fapl);
H5P.close(fapl);
H5F.close(fid);
```

See Also

`H5P.set_gc_references`

H5P.get_mdc_config

Metadata cache configuration

Syntax

```
config_struct = H5P.get_mdc_config(plist_id)
```

Description

`config_struct = H5P.get_mdc_config(plist_id)` returns the current metadata cache configuration from the indicated file access property list.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
config = H5P.get_mdc_config(fapl);
H5P.close(fapl);
H5F.close(fid);
```

See Also

`H5P.set_mdc_config`

H5P.get_meta_block_size

Metadata block size setting

Syntax

```
sz = H5P.get_meta_block_size(fapl_id)
```

Description

`sz = H5P.get_meta_block_size(fapl_id)` returns the current minimum size, in bytes, of new metadata block allocations.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
sz = H5P.get_meta_block_size(fapl);
H5P.close(fapl);
H5F.close(fid);
```

See Also

`H5P.set_meta_block_size`

H5P.get_multi_type

Type of data property for MULTI driver

Syntax

```
type = H5P.get_multi_type(fapl_id)
```

Description

`type = H5P.get_multi_type(fapl_id)` returns the type of data setting from the file access or data transfer property list, `fapl_id`.

This function should only be used with an HDF5 file written as a set of files with the MULTI file driver.

See Also

`H5P.set_multi_type`

H5P.get_sieve_buf_size

Maximum data sieve buffer size

Syntax

```
sz = H5P.get_sieve_buf_size(fapl_id)
```

Description

`sz = H5P.get_sieve_buf_size(fapl_id)` returns the current maximum size of the data sieve buffer.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
sz = H5P.get_sieve_buf_size(fapl);
H5P.close(fapl);
H5F.close(fid);
```

See Also

`H5P.set_sieve_buf_size`

H5P.get_small_data_block_size

Small data block size setting

Syntax

```
sz = H5P.get_small_data_block_size(fapl_id)
```

Description

`sz = H5P.get_small_data_block_size(fapl_id)` returns the current setting for the size of the small data block. `fapl_id` is a file access property list identifier.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
sz = H5P.get_small_data_block_size(fapl);
H5P.close(fapl);
H5F.close(fid);
```

See Also

`H5P.set_small_data_block_size`

H5P.set_alignment

Set alignment properties for file access property list

Syntax

```
H5P.set_alignment(fapl_id,threshold,alignment)
```

Description

`H5P.set_alignment(fapl_id,threshold,alignment)` sets the alignment properties of the file access property list specified by `fapl_id` so that any file object greater than or equal in size to threshold (in bytes) is aligned on an address which is a multiple of alignment.

In most cases the default values of threshold and alignment result in the best performance.

See Also

`H5P.get_alignment`

H5P.set_family_offset

Set offset property for family of files

Syntax

```
H5P.set_family_offset(fapl_id,offset)
```

Description

`H5P.set_family_offset(fapl_id,offset)` sets offset property in the file access property list specified by `fapl_id` for low-level access to a file in a family of files. `offset` identifies a user-determined location from the beginning of the HDF5 file in bytes.

See Also

`H5P.get_family_offset`

H5P.set_fapl_core

Modify file access to use H5FD_CORE driver

Syntax

```
H5P.set_fapl_core(fapl_id,increment,backing_store)
```

Description

`H5P.set_fapl_core(fapl_id,increment,backing_store)` modifies the file access property list to use the H5FD_CORE driver. `increment` specifies the increment by which allocated memory is to be increased each time more memory is required. `backing_store` is a Boolean flag that, when non-zero, indicates the file contents should be written to disk when the file is closed.

Examples

Create a file image in memory only.

```
plist = 'H5P_DEFAULT';
ndatasets = 20;
block_size = 1024*1024;
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_fapl_core(fapl,2^16,false);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',plist,fapl);
space_id = H5S.create_simple(1, block_size, []);
type_id = H5T.copy('H5T_IEEE_F64LE');
data = zeros(block_size,1);
for j = 1:ndatasets
    dsname = sprintf( 'dset%02d', j);
    fprintf( 'Writing dataset %s...\n',dsname);
    dsid = H5D.create(fid,dsname,type_id,space_id,'H5P_DEFAULT');
    H5D.write(dsid,'H5ML_DEFAULT',space_id,space_id,plist,data);
    H5D.close(dsid);
end
H5P.close(fapl);
```

```
H5S.close(space_id);
H5T.close(type_id);
H5F.close(fid);
dir('myfile.h5');
```

See Also

H5P.get_fapl_core

H5P.set_fapl_family

Set file access to use family driver

Syntax

```
H5P.set_fapl_family(fapl_id,memb_size,memb_fapl_id)
```

Description

`H5P.set_fapl_family(fapl_id,memb_size,memb_fapl_id)` sets the file access property list, specified by `fapl_id`, to use the family driver. `memb_size` is the size in bytes of each file member. `memb_fapl_id` is the identifier of the file access property list to be used for each family member.

Examples

```
plist = 'H5P_DEFAULT';
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_fapl_family(fapl, 8192, plist);
fid = H5F.create('family%d.h5','H5F_ACC_TRUNC','H5P_DEFAULT',fapl);
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [50 25];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,[]);
dset_id = H5D.create(fid,'DS',type_id,space_id,plist)
data = reshape(1:prod(dims),dims);
H5D.write(dset_id,'H5ML_DEFAULT','H5S_ALL','H5S_ALL',plist,data);
H5P.close(fapl);
H5T.close(type_id);
H5S.close(space_id);
H5D.close(dset_id);
dir('*.h5');
h5disp('family%d.h5');
```

See Also

H5P.get_fapl_family

H5P.set_fapl_log

Set use of logging driver

Syntax

```
H5P.set_fapl_log(fapl_id,logfile,flags,buf_size)
```

Description

`H5P.set_fapl_log(fapl_id,logfile,flags,buf_size)` modifies the file access property list, `fapl_id`, to use the logging driver H5FD_LOG. `logfile` is the name of the file in which the logging entries are to be recorded. `flags` is a bit mask that specifies the types of activity to log. See the HDF5 documentation for a list of available flag settings. `buf_size` specifies the size of the logging buffer.

H5P.set_fapl_multi

Set use of multifile driver

Syntax

```
H5P.set_fapl_multi(fapl_id,relax)
H5P.set_fapl_multi(fapl_id,memb_map,memb_fapl,memb_name,memb_addr,relax)
```

Description

`H5P.set_fapl_multi(fapl_id,relax)` sets the file access property list, `fapl_id`, to access HDF5 files created with the multi-driver with default values provided by the HDF5 library. `relax` is a Boolean value that allows read-only access to incomplete file sets when set to `1`.

`H5P.set_fapl_multi(fapl_id,memb_map,memb_fapl,memb_name,memb_addr,relax)` sets the file access property list to use the multifile driver. `memb_map` maps memory usage types to other memory usage types. `memb_fapl` contains a property list for each memory usage type. `memb_name` is a name generator for names of member files. `memb_addr` specifies the offsets within the virtual address space at which each type of data storage begins.

See Also

`H5P.get_fapl_multi`

H5P.set_fapl_sec2

Set file access for sec2 driver

Syntax

```
H5P.set_fapl_sec2(fapl_id)
```

Description

`H5P.set_fapl_sec2(fapl_id)` modifies the file access property list, `fapl_id`, to use the H5FD_SEC2 driver.

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_fapl_sec2(fapl);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

H5P.set_fapl_split

Set file access for emulation of split file driver

Syntax

```
H5P.set_fapl_split(fapl_id,meta_ext,meta_plist_id,raw_ext,raw_plist_
id)
```

Description

`H5P.set_fapl_split(fapl_id,meta_ext,meta_plist_id,raw_ext,raw_plist_id)` is a compatibility function that enables the multi-file driver to emulate the split driver from HDF5 Releases 1.0 and 1.2. `meta_ext` is a character vector or string scalar that specifies the metadata filename extension. `meta_plist_id` is a file access property list identifier for the metadata file. `raw_ext` is a character vector or string scalar that specifies the raw data filename extension. `raw_plist_id` is the file access property list identifier for the raw data file.

H5P.set_fapl_stdio

Set file access for standard I/O driver

Syntax

```
H5P.set_fapl_stdio(fapl_id)
```

Description

`H5P.set_fapl_stdio(fapl_id)` modifies the file access property list, `fapl_id`, to use the standard I/O driver, H5FD_STDIO.

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_fapl_stdio(fapl);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

H5P.set_fclose_degree

Set file access for file close degree

Syntax

```
H5P.set_fclose_degree(fapl_id,degree)
```

Description

`H5P.set_fclose_degree(fapl_id,degree)` sets the file close degree property in the file access property list, `fapl_id`, to the value specified by `degree`. Specify `degree` as one of these character vectors or string scalars.

- `'H5F_CLOSE_WEAK'`
- `'H5F_CLOSE_SEMI'`
- `'H5F_CLOSE_STRONG'`
- `'H5F_CLOSE_DEFAULT'`

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_fclose_degree(fapl,'H5F_CLOSE_STRONG');
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

`H5P.get_fclose_degree`

H5P.set_gc_references

Set garbage collection references flag

Syntax

```
H5P.set_gc_references(fapl_id,gc_ref)
```

Description

`H5P.set_gc_references(fapl_id,gc_ref)` sets the flag for garbage collecting references for the file specified by the file access property list identifier, `fapl_id`. The `gc_ref` argument is a flag setting reference garbage collection to on (`1`) or off (`0`).

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_gc_references(fapl,1);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

`H5P.get_gc_references`

H5P.set_libver_bounds

Set library version bounds for objects

Syntax

```
H5P.set_libver_bounds(fapl_id,low,high)
```

Description

`H5P.set_libver_bounds(fapl_id,low,high)` sets bounds on library versions, and indirectly format versions, to be used when creating objects in the file with access property list `fapl_id`. The `low` argument must be set to either of `'H5F_LIBVER_EARLIEST'`, `'H5F_LIBVER_18'` or `'H5F_LIBVER_LATEST'`. The `high` argument must be set to `'H5F_LIBVER_18'` or `'H5F_LIBVER_LATEST'`.

Examples

Create an HDF5 file where objects are created using the latest available format for each object.

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_libver_bounds(fapl,'H5F_LIBVER_LATEST','H5F_LIBVER_LATEST');
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
```

See Also

`H5ML.get_constant_value` | `H5P.get_libver_bounds`

H5P.set_mdc_config

Set initial metadata cache configuration

Syntax

```
H5P.set_mdc_config(plist_id,config_struct)
```

Description

`H5P.set_mdc_config(plist_id,config_struct)` sets the initial metadata cache configuration in the indicated file access property list to the supplied values. Before using this function, you should retrieve the current configuration using `H5P.get_mdc_config`.

Many of the fields in the structure, `config_struct`, are intended to be used only in close consultation with the HDF5 Group itself.

See Also

`H5P.get_mdc_config`

H5P.set_meta_block_size

Set minimum metadata block size

Syntax

```
H5P.set_meta_block_size(fapl_id,size)
```

Description

`H5P.set_meta_block_size(fapl_id,size)` sets the minimum metadata block size for the file access property list specified by `fapl_id`. The `size` argument specifies minimum size, in bytes, of metadata block allocations.

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_meta_block_size(fapl,4096);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

`H5P.get_meta_block_size`

H5P.set_multi_type

Specify type of data accessed with MULTI driver

Syntax

```
H5P.set_multi_type(fapl_id,type)
```

Description

`H5P.set_multi_type(fapl_id,type)` sets the type of data property in the file access or data transfer property list `fapl_id`. The `type` argument can have any of the following values: `H5FD_MEM_SUPER`, `H5FD_MEM_BTREE`, `H5FD_MEM_DRAW`, `H5FD_MEM_GHEAP`, `H5FD_MEM_LHEAP`, or `H5FD_MEM_OHDR`.

See Also

`H5P.get_multi_type`

H5P.set_sieve_buf_size

Set maximum size of data sieve buffer

Syntax

```
H5P.set_sieve_buf_size(fapl_id,buffer_size)
```

Description

`H5P.set_sieve_buf_size(fapl_id,buffer_size)` sets `buffer_size`, the maximum size in bytes of the data sieve buffer, which is used by file drivers that are capable of using data sieving. `fapl_id` is a file access property list identifier.

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_sieve_buf_size(fapl,131072);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

`H5P.get_sieve_buf_size`

H5P.set_small_data_block_size

Set size of block reserved for small data

Syntax

```
H5P.set_small_data_block_size(fapl_id,size)
```

Description

`H5P.set_small_data_block_size(fapl_id,size)` sets the maximum size, in bytes, of a contiguous block reserved for small data. `fapl_id` is a file access property list identifier.

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_small_data_block_size(fapl,4096);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

`H5P.get_small_data_block_size`

H5P.get_istore_k

Return 1/2 rank of indexed storage B-tree

Syntax

```
ik = H5P.get_istore_k(plist_id)
```

Description

`ik = H5P.get_istore_k(plist_id)` returns the chunked storage B-tree 1/2 rank of the file creation property list specified by `plist_id`.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
ik = H5P.get_istore_k(fcpl);
H5P.close(fcpl);
H5F.close(fid);
```

See Also

`H5P.set_istore_k`

H5P.get_sizes

Return size of offsets and lengths

Syntax

```
[sizeof_addr,sizeof_size] = H5P.get_sizes(fcpl)
```

Description

`[sizeof_addr,sizeof_size] = H5P.get_sizes(fcpl)` returns the size of the offsets and lengths used in an HDF5 file. `fcpl` specifies a file creation property list.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
[soaddr, sosize] = H5P.get_sizes(fcpl);
H5P.close(fcpl);
H5F.close(fid);
```

See Also

`H5P.set_sizes`

H5P.get_sym_k

Return size of B-tree 1/2 rank and leaf node 1/2 size

Syntax

```
[ik,lk] = H5P.get_sym_k(plist_id)
```

Description

`[ik,lk] = H5P.get_sym_k(plist_id)` returns the size of the symbol table B-tree 1/2 rank, `ik`, and the symbol table leaf node 1/2 size, `lk`. The `plist_id` argument is a file creation property list identifier.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
[ik, lk] = H5P.get_sym_k(fcpl);
H5P.close(fcpl);
H5F.close(fid);
```

See Also

`H5P.set_sym_k`

H5P.get_userblock

Return size of user block

Syntax

```
sz = H5P.get_userblock(plist_id)
```

Description

`sz = H5P.get_userblock(plist_id)` returns the size of a user block in a file creation property list. `plist_id` is a property list identifier.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
sz = H5P.get_userblock(fcpl);
H5P.close(fcpl);
H5F.close(fid);
```

See Also

`H5P.set_userblock`

H5P.get_version

Return version information for file creation property list

Syntax

```
[superblock,freelist,stab,shhdr] = H5P.get_version(fcpl)
```

Description

`[superblock,freelist,stab,shhdr] = H5P.get_version(fcpl)` returns the version of the super block, the global freelist, the symbol table, and the shared object header. Retrieving this information requires the file creation property list.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
[super,freelist,stab,shhdr] = H5P.get_version(fcpl);
H5P.close(fcpl);
H5F.close(fid);
```

H5P.set_istore_k

Set size of parameter for indexing chunked datasets

Syntax

```
H5P.set_istore_k(plist_id,ik)
```

Description

`H5P.set_istore_k(plist_id,ik)` sets the size of the parameter used to control the B-trees for indexing chunked datasets for the file creation property list specified by `plist_id`. The `ik` argument is one half the rank of a tree that stores chunked raw data.

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_istore_k(fcpl,64);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

`H5P.get_istore_k`

H5P.set_sizes

Set byte size of offsets and lengths

Syntax

```
H5P.set_sizes(plist_id,sizeof_addr,sizeof_size)
```

Description

`H5P.set_sizes(plist_id,sizeof_addr,sizeof_size)` sets the byte size of the offsets and lengths used to address objects in an HDF5 file. `plist_id` is a file creation property list.

See Also

`H5P.get_sizes`

H5P.set_sym_k

Set size of parameters used to control symbol table nodes

Syntax

```
H5P.set_sym_k(plist_id,ik,lk)
```

Description

`H5P.set_sym_k(plist_id,ik,lk)` sets the size of parameters used to control the symbol table nodes for the file access property list, `plist_id`. The `ik` argument is one half the rank of a tree that stores a symbol table for a group. `lk` is one half of the number of symbols that can be stored in a symbol table node.

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_sym_k(fcpl,32,8);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

`H5P.get_sym_k`

H5P.set_userblock

Set user block size

Syntax

```
H5P.set_userblock(plist_id,size)
```

Description

`H5P.set_userblock(plist_id,size)` sets the user block size of the file creation property list, `plist_id`.

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_userblock(fcpl,4096);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

`H5P.get_userblock`

H5P.get_attr_creation_order

Return tracking order and indexing settings

Syntax

```
crt_order_flags = H5P.get_attr_creation_order(ocpl_id)
```

Description

`crt_order_flags = H5P.get_attr_creation_order(ocpl_id)` retrieves tracking and indexing settings for attribute creation order. If `crt_order_flags` is zero, then the attribute creation order is neither tracked or indexed. Otherwise the creation order flags should be one of the following constant values:

```
H5P_CRT_ORDER_TRACKED
H5P_CRT_ORDER_INDEXED
```

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
dcpl = H5D.get_create_plist(dset_id);
flags = H5P.get_attr_creation_order(dcpl);
switch ( flags )
    case 0
        fprintf('neither tracked nor indexed\n');
    case H5ML.get_constant_value('H5P_CRT_ORDER_TRACKED')
        fprintf('tracked\n');
    case H5ML.get_constant_value('H5P_CRT_ORDER_INDEXED')
        fprintf('indexed\n');
end
H5P.close(dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5ML.get_constant_value` | `H5P.set_attr_creation_order`

H5P.get_attr_phase_change

Retrieve attribute phase change thresholds

Syntax

```
[max_compact,min_dense] = H5P.get_attr_phase_change(ocpl_id)
```

Description

`[max_compact,min_dense] = H5P.get_attr_phase_change(ocpl_id)` retrieves attribute phase change thresholds for the dataset or group with creation property list `ocpl_id`.

`max_compact` is the maximum number of attributes to be stored in compact storage (default is 8).

`min_dense` is the minimum number of attributes to be stored in dense storage (default is 6).

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
dcpl = H5D.get_create_plist(dset_id);
[max_compact,min_dense] = H5P.get_attr_phase_change(dcpl);
H5P.close(dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.set_attr_phase_change`

H5P.get_copy_object

Return properties to be used when object is copied

Syntax

```
copy_options = H5P.get_copy_object(ocpl_id)
```

Description

`copy_options = H5P.get_copy_object(ocpl_id)` retrieves the properties currently specified in the object copy property list `ocpl_id`, which will be invoked when a new copy is made of an existing object.

Examples

```
ocpl = H5P.create('H5P_OBJECT_COPY');
options = H5P.get_copy_object(ocpl);
```

See Also

`H5P.set_copy_object`

H5P.set_attr_creation_order

Set tracking of attribute creation order

Syntax

```
H5P.set_attr_creation_order(gcplId,crt_order_flags)
```

Description

`H5P.set_attr_creation_order(gcplId,crt_order_flags)` sets tracking and indexing of attribute creation order. The creation order flags should be either `H5P_CRT_ORDER_TRACKED` or a bitwise-or of `H5P_CRT_ORDER_TRACKED` and `H5P_CRT_ORDER_INDEXED`.

The default behavior is that attribute creation order is neither tracked nor indexed.

Examples

```
dcpl = H5P.create('H5P_DATASET_CREATE');
order = H5ML.get_constant_value('H5P_CRT_ORDER_TRACKED');
H5P.set_attr_creation_order(dcpl,order);
```

See Also

`H5ML.get_constant_value` | `H5P.get_attr_creation_order` | `bitor`

H5P.set_attr_phase_change

Set attribute storage phase change thresholds

Syntax

```
H5P.set_attr_phase_change(ocpl_id,max_compact,min_dense)
```

Description

`H5P.set_attr_phase_change(ocpl_id,max_compact,min_dense)` sets attribute storage phase change thresholds for the group or dataset with creation order property list `ocpl_id`.

`max_compact` is the maximum number of attributes to be stored in compact storage (default is 8).

`min_dense` is the minimum number of attributes to be stored in dense storage (default is 6).

See Also

`H5P.get_attr_phase_change`

H5P.set_copy_object

Set properties to be used when objects are copied

Syntax

```
H5P.set_copy_object(ocp_plist_id,copy_options)
```

Description

`H5P.set_copy_object(ocp_plist_id,copy_options)` sets the properties in the object copy property list `ocp_plist_id` that will be invoked when a new copy is made of an existing object. `ocp_plist_id` is the object copy property list and specifies the properties governing the copying of the object.

Specify `copy_options` as a character vector or string scalar containing one of these flag names, or its equivalent numerical value.

`H5O_COPY_SHALLOW_HIERARCHY_FLAG`	Copy only immediate members of a group. Default behavior, without flag: Recursively copy all objects below the group.
`H5O_COPY_EXPAND_SOFT_LINK_FLAG`	Expand soft links into new objects. Default behavior, without flag: Keep soft links as they are.
`H5O_COPY_EXPAND_EXT_LINK_FLAG`	Expand external link into new objects. Default behavior, without flag: Keep external links as they are.
`H5O_COPY_EXPAND_REFERENCE_FLAG`	Copy objects that are pointed to by references. Default behavior, without flag: Update only the values of object references.
`H5O_COPY_WITHOUT_ATTR_FLAG`	Copy object without copying attributes. Default behavior, without flag: Copy object along with all its attributes.

Examples

```
ocp_plist_id = H5P.create ('H5P_OBJECT_COPY');
option1 = H5ML.get_constant_value('H5O_COPY_EXPAND_SOFT_LINK_FLAG');
option2 = H5ML.get_constant_value('H5O_COPY_EXPAND_REFERENCE_FLAG');
copy_options = bitor(option1,option2);
H5P.set_copy_object(ocp_plist_id,copy_options);
```

H5P.get_create_intermediate_group

Determine creation of intermediate groups

Syntax

```
bool = H5P.get_create_intermediate_group(lcpl_id)
```

Description

`bool = H5P.get_create_intermediate_group(lcpl_id)` determines whether the link creation property list `lcpl_id` is set to enable creating missing intermediate groups.

Examples

```
lcpl = H5P.create('H5P_LINK_CREATE');
if H5P.get_create_intermediate_group(lcpl)
    fprintf('set to enable creating intermediate groups\n');
else
    fprintf('not set to enable creating intermediate groups\n');
end
```

See Also

`H5P.set_create_intermediate_group`

H5P.get_link_creation_order

Query if link creation order is tracked

Syntax

```
crt_order_flags = H5P.get_link_creation_order(gcpl_id)
```

Description

`crt_order_flags = H5P.get_link_creation_order(gcpl_id)` queries whether link creation order is tracked or indexed in a group with creation property list identifier `gcpl_id`. The creation order flags should be one of the following constant values:

```
H5P_CRT_ORDER_TRACKED
H5P_CRT_ORDER_INDEXED
```

Examples

```
tracked = H5ML.get_constant_value('H5P_CRT_ORDER_TRACKED');
indexed = H5ML.get_constant_value('H5P_CRT_ORDER_INDEXED');
gcpl = H5P.create('H5P_GROUP_CREATE');
order = H5P.get_link_creation_order(gcpl);
if bitand(order,tracked)
    fprintf('order is tracked\n');
end
if bitand(order,indexed)
    fprintf('order is indexed\n');
end
```

See Also

`H5ML.get_constant_value` | `H5P.set_link_creation_order` | `bitand`

H5P.get_link_phase_change

Query settings for conversion between groups

Syntax

```
[max_compact,min_dense] = H5P.get_link_phase_change(gcpl_id)
```

Description

`[max_compact,min_dense] = H5P.get_link_phase_change(gcpl_id)` retrieves the settings for conversion between compact and dense groups.

`max_compact` is the maximum number of links to store as header messages in the group header before converting the group to the dense format. Groups that are in the compact format and exceed this number of links are automatically converted to the dense format.

`min_dense` is the minimum number of links to store in the dense format. Groups which are in dense format and in which the number of links falls below this number are automatically converted back to the compact format.

Examples

```
gcpl = H5P.create('H5P_GROUP_CREATE');
[max_compact, min_dense] = H5P.get_link_phase_change(gcpl);
```

See Also

`H5P.set_link_phase_change`

H5P.set_create_intermediate_group

Set creation of intermediate groups

Syntax

```
H5P.set_create_intermediate_group(lcpl_id,flag)
```

Description

`H5P.set_create_intermediate_group(lcpl_id,flag)` specifies in the link creation property list `lcpl_id` whether to create missing intermediate groups.

Examples

Enable the creation of intermediate groups.

```
fid = H5F.create('myfile.h5');
lcpl = H5P.create('H5P_LINK_CREATE');
H5P.set_create_intermediate_group(lcpl,1);
gid = H5G.create(fid,'/a/b/c/d',lcpl,'H5P_DEFAULT','H5P_DEFAULT');
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5P.get_create_intermediate_group`

H5P.set_link_creation_order

Set creation order tracking and indexing

Syntax

```
H5P.set_link_creation_order(gcplId,crt_order_flags)
```

Description

`H5P.set_link_creation_order(gcplId,crt_order_flags)` sets creation order tracking and indexing for links in the group with group creation property list `gcpl_id`.

The creation order flags should be one of the following constant values:

```
H5P_CRT_ORDER_TRACKED
H5P_CRT_ORDER_INDEXED
```

If only `H5P_CRT_ORDER_TRACKED` is set, HDF5 will track link creation order in any group created with the group creation property list `gcpl_id`. If both `H5P_CRT_ORDER_TRACKED` and `H5P_CRT_ORDER_INDEXED` are set, HDF5 will track link creation order in the group and index links on that property.

Examples

```
tracked = H5ML.get_constant_value('H5P_CRT_ORDER_TRACKED');
indexed = H5ML.get_constant_value('H5P_CRT_ORDER_INDEXED');
order = bitor(tracked,indexed);
gcpl = H5P.create('H5P_GROUP_CREATE');
H5P.set_link_creation_order(gcpl,order);
```

See Also

`H5ML.get_constant_value` | `H5P.get_link_creation_order`

H5P.set_link_phase_change

Set parameters for group conversion

Syntax

```
H5P.set_link_phase_change(gcpl_id,max_compact,min_dense)
```

Description

`H5P.set_link_phase_change(gcpl_id,max_compact,min_dense)` sets the parameters for conversion between compact and dense groups.

`max_compact` is the maximum number of links to store as header messages in the group header before converting the group to the dense format. Groups that are in the compact format and exceed this number of links are automatically converted to the dense format.

`min_dense` is the minimum number of links to store in the dense format. Groups which are in dense format and in which the number of links falls below this number are automatically converted back to the compact format.

Examples

```
gcpl = H5P.create('H5P_GROUP_CREATE');
H5P.set_link_phase_change(gcpl,10,8);
```

See Also

`H5P.get_link_phase_change`

H5P.get_char_encoding

Return character encoding

Syntax

```
encoding = H5P.get_char_encoding(propertyList)
```

Description

`encoding = H5P.get_char_encoding(propertyList)` retrieves the character encoding used to encode strings or object names that are created with the property list `propertyList`. The values returned correspond to either `'H5T_CSET_ASCII'` or `'H5T_CSET_UTF8'`.

See Also

`H5ML.get_constant_value` | `H5P.set_char_encoding`

H5P.set_char_encoding

Set character encoding used to encode strings

Syntax

```
H5P.set_char_encoding(propList,encoding)
```

Description

`H5P.set_char_encoding(propList,encoding)` sets the character encoding used to encode strings or object names that are created with the property list `propList`. The values of `encoding` should either be `'H5T_CSET_ASCII'` or `'H5T_CSET_UTF8'`.

See Also

`H5ML.get_constant_value` | `H5P.get_char_encoding`

H5R.create

Create reference

Syntax

```
ref = H5R.create(loc_id,name,ref_type,space_id)
```

Description

`ref = H5R.create(loc_id,name,ref_type,space_id)` creates the reference, `ref`, of the type specified in `ref_type`, pointing to the object specified by name located at `loc_id`. The `ref_type` argument can be either `'H5R_OBJECT'`, or `'H5R_DATASET_REGION'`. The `space_id` argument should be `-1`, if `ref_type` is `'H5R_OBJECT'`.

Examples

Create a double-precision dataset and a reference dataset.

```
fid = H5F.create('myfile.h5');
type1_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [10 5];
h5_dims = fliplr(dims);
h5_maxdims = h5_dims;
space1_id = H5S.create_simple(2,h5_dims,h5_maxdims);
dcpl = 'H5P_DEFAULT';
dset1_id = H5D.create(fid,'my_double',type1_id,space1_id,dcpl);
type2_id = 'H5T_STD_REF_OBJ';
space2_id = H5S.create('H5S_SCALAR');
dset2_id = H5D.create(fid,'my_ref',type2_id,space2_id,dcpl);
ref_data = H5R.create(fid,'my_double','H5R_OBJECT',-1);
dxpl = 'H5P_DEFAULT';
H5D.write(dset2_id,'H5ML_DEFAULT','H5S_ALL','H5S_ALL',dxpl,ref_data);
H5D.close(dset1_id);
H5D.close(dset2_id);
H5F.close(fid);
```

See Also

`H5D.create`

H5R.dereference

Open object specified by reference

Syntax

```
output = H5R.dereference(dataset,ref_type,ref)
```

Description

`output = H5R.dereference(dataset,ref_type,ref)` returns an identifier to the object specified by `ref` in the dataset specified by `dataset`.

Examples

```
plist = 'H5P_DEFAULT';
space = 'H5S_ALL';
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/reference');
ref_data = H5D.read(dset_id,'H5T_STD_REF_OBJ',space,space,plist);
deref_dset_id = H5R.dereference(dset_id,'H5R_OBJECT',ref_data(:,1));
H5D.close(dset_id);
H5D.close(deref_dset_id);
H5F.close(fid);
```

See Also

`H5I.get_name` | `H5R.create`

H5R.get_name

Name of referenced object

Syntax

```
name = H5R.get_name(loc_id,ref_type,ref)
name = H5R.get_name(loc_id,ref_type,ref,'TextEncoding',encoding)
```

Description

`name = H5R.get_name(loc_id,ref_type,ref)` retrieves the name for the object identified by `ref`. The `loc_id` argument is the identifier for the dataset containing the reference or for the group containing that dataset. `ref_type` specifies the type of the reference `ref`. Valid values for `ref_type` are `'H5R_OBJECT'` or `'H5R_DATASET_REGION'`.

`name = H5R.get_name(loc_id,ref_type,ref,'TextEncoding',encoding)` additionally specifies the text encoding to use to interpret the reference name. Specify `encoding` as `'system'`(default) or `'UTF-8'`.

- `'system'` — Use the system default encoding to interpret the reference name.
- `'UTF-8'` — Use `UTF-8` encoding to interpret the reference name.

Examples

```
plist = 'H5P_DEFAULT';
space = 'H5S_ALL';
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/reference');
ref_data = H5D.read(dset_id,'H5T_STD_REF_OBJ',space,space,plist);
name = H5R.get_name(dset_id,'H5R_OBJECT',ref_data(:,1));
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5I.get_name`

Topics

"Working with Non-ASCII Characters in HDF5 Files"

H5R.get_obj_type

Type of referenced object

Syntax

```
obj_type = H5R.get_obj_type(id,ref_type,ref)
```

Description

`obj_type = H5R.get_obj_type(id,ref_type,ref)` returns the type of object that an object reference points to. Valid values for `ref_type` are: `'H5R_OBJECT'` or `'H5R_DATASET_REGION'`. The function returns one of these values.

`'H5O_TYPE_GROUP'`	Object is a group.
`'H5O_TYPE_DATASET'`	Object is a dataset.
`'H5O_TYPE_NAMED_DATATYPE'`	Object is a named datatype.

This function corresponds to the 1.8 interface version of `H5Rget_obj_type` in the HDF5 library C API.

Examples

```
plist = 'H5P_DEFAULT';
space = 'H5S_ALL';
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/reference');
ref_data = H5D.read(dset_id,'H5T_STD_REF_OBJ',space,space,plist);
obj_type = H5R.get_obj_type(fid,'H5R_OBJECT',ref_data(:,1));
switch(obj_type)
    case H5ML.get_constant_value('H5O_TYPE_GROUP')
        fprintf('group\n');
    case H5ML.get_constant_value('H5O_TYPE_DATASET')
        fprintf('dataset\n');
    case H5ML.get_constant_value('H5O_TYPE_NAMED_DATATYPE')
```

```
        fprintf('named datatype\n');
end
H5D.close(dset_id);
H5F.close(fid);
```

See Also

H5ML.get_constant_value

H5R.get_region

Copy of data space of specified region

Syntax

```
space_id = H5R.get_region(dataset,ref_type,ref)
```

Description

`space_id = H5R.get_region(dataset,ref_type,ref)` returns a data space with the specified region selected. `dataset` is used to identify the file containing the referenced region and can be any identifier for any object in the file.

Examples

```
space = 'H5S_ALL';
plist = 'H5P_DEFAULT';
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/region_reference');
ref_data = H5D.read(dset_id,'H5T_STD_REF_DSETREG',space,space,plist);
space_id = H5R.get_region(fid,'H5R_DATASET_REGION',ref_data(:,1));
H5S.close(space_id);
H5D.close(dset_id);
H5F.close(fid);
```

h5read

Read data from HDF5 data set

Syntax

```
data = h5read(filename,datasetname)
data = h5read(filename,datasetname,start,count)
data = h5read(filename,datasetname,start,count,stride)
```

Description

`data = h5read(filename,datasetname)` reads all the data from the dataset `datasetname` contained in the HDF5 file `filename`.

`data = h5read(filename,datasetname,start,count)` reads a subset of data from the dataset beginning at the location specified in `start`. The `count` argument specifies the number of elements to read along each dimension.

`data = h5read(filename,datasetname,start,count,stride)` returns a subset of data with the interval between the indices of each dimension of the dataset specified by `stride`.

Examples

Read Entire HDF5 Dataset

Get the metadata for a dataset from the HDF5 file and then read the dataset.

Display the metadata for a dataset `/g4/lat` from the HDF5 file `example.h5`.

```
h5disp('example.h5','/g4/lat')
```

```
HDF5 example.h5
Dataset 'lat'
```

```
    Size:  19
    MaxSize:  19
    Datatype:   H5T_IEEE_F64LE (double)
    ChunkSize:  []
    Filters:  none
    FillValue:  0.000000
    Attributes:
        'units':  'degrees_north'
        'CLASS':  'DIMENSION_SCALE'
        'NAME':  'lat'
```

Read the dataset.

```
data = h5read('example.h5','/g4/lat')
```

```
data = 19×1

   -90
   -80
   -70
   -60
   -50
   -40
   -30
   -20
   -10
     0
     ⋮
```

Read Subset of HDF5 Dataset

Get the metadata for a dataset from the HDF5 file and then read a subset of the dataset.

Display the metadata for a dataset `/g4/world` from the HDF5 file `example.h5`.

```
h5disp('example.h5','/g4/world')
```

```
HDF5 example.h5
Dataset 'world'
    Size:  36x19
    MaxSize:  36x19
```

```
    Datatype:   H5T_IEEE_F64LE (double)
    ChunkSize:  []
    Filters:  none
    FillValue:  0.000000
```

Starting from the beginning of data, read a 5-by-3 subset of the data from the dataset.

```
start = [1 1];
count = [5 3];
data = h5read('example.h5','/g4/world',start,count)
```

```
data = 5×3

     0     0     0
     0     0     0
     0     0     0
     0     0     0
     0     0     0
```

Read HDF5 Dataset with Specified Spacing Between Indices

Read data from a dataset, where the data is sampled at a specified spacing between the dataset indices along each dimension.

First, display the metadata for a dataset `/g4/lon` from the HDF5 file `example.h5`. The variable in the dataset has one dimension with 36 elements.

```
h5disp('example.h5','/g4/lon')
```

```
HDF5 example.h5
Dataset 'lon'
    Size:  36
    MaxSize:  36
    Datatype:   H5T_IEEE_F64LE (double)
    ChunkSize:  []
    Filters:  none
    FillValue:  0.000000
    Attributes:
        'units':  'degrees_east'
        'CLASS':  'DIMENSION_SCALE'
        'NAME':  'lon'
```

Start reading from the location in `startLoc` and read variable data at intervals specified in `stride`. A value of 1 in `stride`, accesses adjacent values in the corresponding dimension. Whereas, a value of 2 accesses every other value in the corresponding dimension, and so on.

```
startLoc = 1;
count  = 18;
stride = 2;
subsetData  = h5read('example.h5','/g4/lon',startLoc,count,stride);
```

Examine the output variable `subsetData`.

```
whos subsetData
```

```
  Name             Size            Bytes  Class     Attributes

  subsetData      18x1               144  double
```

Input Arguments

`filename` — File name
character vector | string scalar

File name, specified as a character vector or string scalar containing the name of a HDF5 file.

Data Types: `char` | `string`

`datasetname` — Dataset name
character vector | string scalar

Dataset name, specified as a character vector or string scalar containing the name of the dataset in the HDF5 file. An HDF5 dataset is a multidimensional array of data elements, together with supporting metadata.

Data Types: `char` | `string`

`start` — Starting location
vector of ones (default) | numeric vector

Starting location, specified as a numeric vector of positive integers. For an N-dimensional dataset, `start` is a vector of length *N* containing 1-based indices.

If you do not specify `start`, then the `h5read` function starts reading the dataset from the first index along each dimension.

Data Types: `double`

`count` — Number of elements
vector of `Inf`'s (default) | numeric vector

Number of elements to read, specified as a numeric vector of positive integers. For an `N`-dimensional dataset, `count` is a vector of length `N`, specifying the number of elements to read along each dimension. If any element of `count` is `Inf`, then `h5read` reads until the end of the corresponding dimension.

If you do not specify `count`, then the `h5read` function reads data until end of each dimension.

Data Types: `double`

`stride` — Space between indices
vector of ones (default) | numeric vector

Space between the indices along each dimension of the dataset, specified as a numeric vector of integers. For an `N`-dimensional variable in the dataset, `stride` is vector of length `N`. The elements of the `stride` vector correspond, in order, to the variable dimensions. A value of `1` accesses adjacent values of the variable in the corresponding dimension. Whereas, a value of `2` accesses every other value of the variable in the corresponding dimension, and so on.

If you do not specify `stride`, then the `h5read` function reads the data with a default spacing of `1` along each dimension.

Data Types: `double`

See Also

`h5disp` | `h5readatt` | `h5write` | `h5writeatt`

Topics

"Using the High-Level HDF5 Functions to Import Data"

Introduced in R2011a

H5S.copy

Create copy of data space

Syntax

```
output = H5S.copy(space_id)
```

Description

`output = H5S.copy(space_id)` creates a new data space, which is an exact copy of the dataspace identified by `space_id`. The `output` argument is a data space identifier.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g2/dset2.1');
space1_id = H5D.get_space(dset_id);
space2_id = H5S.copy(space1_id);
[~,dims1] = H5S.get_simple_extent_dims(space1_id)
[~,dims2] = H5S.get_simple_extent_dims(space2_id)
```

See Also

`H5D.get_space` | `H5S.get_simple_extent_dims`

H5S.create

Create new data space

Syntax

```
space_id = H5S.create(space_type)
```

Description

`space_id = H5S.create(space_type)` creates a new dataspace of the type `space_type`. Specify `space_type` as one of these character vectors or string scalars.

- `'H5S_SCALAR'`
- `'H5S_SIMPLE'`
- `'H5S_NULL'`

`space_id` is the identifier for the new dataspace.

Examples

Create a scalar dataspace.

```
space_id = H5S.create('H5S_SCALAR');
numpoints = H5S.get_simple_extent_npoints(space_id);
```

See Also

`H5S.get_simple_extent_npoints`

H5S.close

Close data space

Syntax

```
H5S.close(space_id)
```

Description

`H5S.close(space_id)` releases and terminates access to a data space. `space_id` is a data space identifier.

See Also

`H5A.get_space` | `H5D.get_space`

H5S.create_simple

Create new simple data space

Syntax

```
space_id = H5S.create_simple(rank,h5_dims,h5_maxdims)
```

Description

`space_id = H5S.create_simple(rank,h5_dims,h5_maxdims)` creates a new simple data space and opens it for access. `rank` is the number of dimensions used in the data space. `h5_dims` is an array specifying the size of each dimension of the dataset. `h5_maxdims` is an array specifying the upper limit on the size of each dimension. `space_id` is a data space identifier.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_dims` and `h5_maxdims` parameters assume C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

Create a data space for a dataset with 10 rows and 5 columns.

```
dims = [10 5];
h5_dims = fliplr(dims);
h5_maxdims = h5_dims;
space_id = H5S.create_simple(2,h5_dims,h5_maxdims);
```

Create a data space for a dataset with 10 rows and 5 columns such that the dataset is extendible along the column dimension.

```
dims = [10 5];
h5_dims = fliplr(dims);
```

```
maxdims = [10 H5ML.get_constant_value('H5S_UNLIMITED')];
h5_maxdims = fliplr(maxdims);
space_id = H5S.create_simple(2,h5_dims,h5_maxdims);
```

See Also

H5ML.get_constant_value | H5S.close | H5S.create

H5S.extent_copy

Copy extent from source to destination data space

Syntax

```
H5S.extent_copy(dst_id,src_id)
```

Description

`H5S.extent_copy(dst_id,src_id)` copies the extent from the source data space, `src_id`, to the destination data space, `dst_id`.

Examples

```
space_id1 = H5S.create('H5S_SIMPLE');
dims = [100 200];
h5_dims = fliplr(dims);
maxdims = [100 H5ML.get_constant_value('H5S_UNLIMITED')];
h5_maxdims = fliplr(maxdims);
H5S.set_extent_simple(space_id1,2,h5_dims,h5_maxdims);
space_id2 = H5S.create('H5S_SIMPLE');
H5S.extent_copy(space_id2,space_id1);
```

See Also

`H5S.create` | `H5S.get_simple_extent_dims` | `H5S.set_extent_simple`

H5S.get_select_bounds

Bounding box of data space selection

Syntax

```
[start,finish] = H5S.get_select_bounds(space_id)
```

Description

`[start,finish] = H5S.get_select_bounds(space_id)` returns the coordinates of the bounding box containing the current selection. `start` contains the starting coordinates of the bounding box and `finish` contains the coordinates of the diagonally opposite corner.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_start`, `h5_stride`, `h5_count` and `h5_block` parameters assume C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
start = fliplr([10 20]); block = fliplr([20 30]);
H5S.select_hyperslab(space_id,'H5S_SELECT_SET',start,[],[],block);
start = fliplr([30 40]); block = fliplr([20 30]);
H5S.select_hyperslab(space_id,'H5S_SELECT_OR',start,[],[],block);
[start, finish] = H5S.get_select_bounds(space_id);
matlab_start = fliplr(start);
matlab_finish = fliplr(finish);
```

See Also

`H5S.create_simple` | `H5S.select_hyperslab`

H5S.get_select_elem_npoints

Number of element points in selection

Syntax

```
numpoints = H5S.get_select_elem_npoints(space_id)
```

Description

`numpoints = H5S.get_select_elem_npoints(space_id)` returns the number of element points in the current data space selection.

Examples

Select the corner points of a data space.

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
coords = [0 0; 0 199; 99 0; 99 199];
coords = fliplr(coords);
coords = coords';
H5S.select_elements(space_id,'H5S_SELECT_SET',coords)
numpoints = H5S.get_select_elem_npoints(space_id);
```

See Also

`H5S.select_elements`

H5S.get_select_elem_pointlist

Element points in data space selection

Syntax

```
points =
H5S.get_select_elem_pointlist(space_id,startpoint,numpoints)
```

Description

`points = H5S.get_select_elem_pointlist(space_id,startpoint,numpoints)` returns the list of element points in the current data space selection. `startpoint` specifies the element point to start with and `numpoints` specifies the total number of points.

`points` is a two-dimensional array of 0-based values specifying the coordinates of the elements. If `m` is the rank of the dataspace, then `points` will have size [`m` x `numpoints`].

Note The ordering of the coordinate points is the same as the HDF5 library C API.

Examples

Determine the first two points in the current selection.

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
coords = [0 0; 0 199; 99 0; 99 199];
coords = fliplr(coords);
coords = coords';
H5S.select_elements(space_id,'H5S_SELECT_SET',coords);
points = H5S.get_select_elem_pointlist(space_id,0,2);
```

H5S.get_select_hyper_blocklist

List of hyperslab blocks

Syntax

```
blocklist =
H5S.get_select_hyper_blocklist(space_id,startblock,numblocks)
```

Description

`blocklist = H5S.get_select_hyper_blocklist(space_id,startblock,numblocks)` returns a list of the hyperslab blocks currently selected. `space_id` is a dataspace identifier. `startblock` specifies the block to start with and `numblocks` specifies the number of hyperslab blocks to retrieve.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_start`, `h5_stride`, `h5_count` and `h5_block` parameters assume C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
start = fliplr([10 20]); block = fliplr([20 25]);
H5S.select_hyperslab(space_id,'H5S_SELECT_SET',start,[],[],block);
start = fliplr([20 30]); block = fliplr([20 25]);
H5S.select_hyperslab(space_id,'H5S_SELECT_NOTB',start,[],[],block);
nblocks = H5S.get_select_hyper_nblocks(space_id);
for j = 1:nblocks
    hblocks{j} = H5S.get_select_hyper_blocklist(space_id,j-1,1);
end
```

See Also

`H5S.get_select_hyper_nblocks` | `H5S.select_hyperslab`

H5S.get_select_hyper_nblocks

Number of hyperslab blocks

Syntax

```
num_blocks = H5S.get_select_hyper_nblocks(space_id)
```

Description

`num_blocks = H5S.get_select_hyper_nblocks(space_id)` returns the number of hyperslab blocks in the current data space selection.

Examples

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
start = fliplr([10 20]); block = fliplr([20 25]);
H5S.select_hyperslab(space_id,'H5S_SELECT_SET',start,[],[],block);
start = fliplr([20 30]); block = fliplr([20 25]);
H5S.select_hyperslab(space_id,'H5S_SELECT_NOTB',start,[],[],block);
nblocks = H5S.get_select_hyper_nblocks(space_id);
```

See Also

`H5S.get_select_hyper_blocklist` | `H5S.select_hyperslab`

H5S.get_select_npoints

Number of elements in data space selection

Syntax

```
num_points = H5S.get_select_npoints(space_id)
```

Description

`num_points = H5S.get_select_npoints(space_id)` returns the number of elements in the current data space selection.

Examples

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
op = 'H5S_SELECT_SET';
start = fliplr([10 20]); block = fliplr([20 30]);
H5S.select_hyperslab(space_id,'H5S_SELECT_SET',start,[],[],block);
n = H5S.get_select_npoints(space_id);
```

See Also

`H5S.create_simple` | `H5S.select_hyperslab`

H5S.get_select_type

Type of data space selection

Syntax

```
sel_type = H5S.get_select_type(space_id)
```

Description

`sel_type = H5S.get_select_type(space_id)` returns the data space selection type. Valid return values correspond to the following enumerated constants:

```
H5S_SEL_NONE
H5S_SEL_POINTS
H5S_SEL_HYPERSLABS
H5S_SEL_ALL
```

Examples

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
start = fliplr([10 20]); block = fliplr([20 30]);
H5S.select_hyperslab(space_id,'H5S_SELECT_SET',start,[],[],block);
sel_type = H5S.get_select_type(space_id);
switch(sel_type)
    case H5ML.get_constant_value('H5S_SEL_NONE')
        fprintf('no selection\n');
    case H5ML.get_constant_value('H5S_SEL_POINTS');
        fprintf('point selection\n');
    case H5ML.get_constant_value('H5S_SEL_HYPERSLABS');
        fprintf('hyperslab selection\n');
end
```

See Also

`H5ML.get_constant_value` | `H5S.select_elements` | `H5S.select_hyperslab`

H5S.get_simple_extent_dims

Data space size and maximum size

Syntax

```
[numdims,h5_dims,h5_maxdims] = H5S.get_simple_extent_dims(space_id)
```

Description

`[numdims,h5_dims,h5_maxdims] = H5S.get_simple_extent_dims(space_id)` returns the number of dimensions in the data space, the size of each dimension, and the maximum size of each dimension.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_dims` and `h5_maxdims` assume C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g2/dset2.2');
space_id = H5D.get_space(dset_id);
[ndims,h5_dims] = H5S.get_simple_extent_dims(space_id);
matlab_dims = fliplr(h5_dims);
```

H5S.get_simple_extent_ndims

Data space rank

Syntax

```
output = H5S.get_simple_extent_ndims(space_id)
```

Description

`output = H5S.get_simple_extent_ndims(space_id)` returns the dimensionality (also called the rank) of a data space.

H5S.get_simple_extent_npoints

Number of elements in data space

Syntax

```
output = H5S.get_simple_extent_npoints(space_id)
```

Description

`output = H5S.get_simple_extent_npoints(space_id)` returns the number of elements in the data space specified by `space_id`.

H5S.get_simple_extent_type

Data space class

Syntax

```
space_type = H5S.get_simple_extent_type(space_id)
```

Description

`space_type = H5S.get_simple_extent_type(space_id)` returns the data space class of the data space specified by `space_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
space_id = H5D.get_space(dset_id);
space_type = H5S.get_simple_extent_type(space_id);
switch(space_type)
    case H5ML.get_constant_value('H5S_SCALAR')
        fprintf('scalar\n');
    case H5ML.get_constant_value('H5S_SIMPLE')
        fprintf('simple\n');
    case H5ML.get_constant_value('H5S_NULL')
        fprintf('none\n');
end
```

See Also

`H5D.get_space` | `H5ML.get_constant_value` | `H5S.create`

H5S.is_simple

Determine if data space is simple

Syntax

```
output = H5S.is_simple(space_id)
```

Description

`output = H5S.is_simple(space_id)` returns a positive value if the data space specified by `space_id` is a simple data space, zero if it is not, and a negative value to indicate failure.

Examples

Create a new data space and verify that it is simple.

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
val = H5S.is_simple(space_id);
```

Create a null data space and verify that it is not simple.

```
space_id = H5S.create('H5S_NULL');
val = H5S.is_simple(space_id);
```

See Also

`H5S.create` | `H5S.create_simple`

H5S.offset_simple

Set offset of simple data space

Syntax

```
H5S.offset_simple(space_id,offset)
```

Description

`H5S.offset_simple(space_id,offset)` specifies the offset of the simple data space specified by `space_id`. This function allows the same shaped selection to be moved to different locations within a data space without requiring it to be redefined.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_start`, `h5_stride`, `h5_count` and `h5_block` parameters assume C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
start = fliplr([10 20]); block = fliplr([20 30]);
H5S.select_hyperslab(space_id,'H5S_SELECT_SET',start,[],[],block);
offset = fliplr([3 5]);
H5S.offset_simple(space_id,offset)
[start,finish] = H5S.get_select_bounds(space_id);
start = fliplr(start);
finish = fliplr(finish);
```

See Also

`H5S.get_select_bounds` | `H5S.select_hyperslab`

H5S.select_all

Select entire extent of data space

Syntax

```
H5S.select_all(space_id)
```

Description

`H5S.select_all(space_id)` selects the entire extent of the data space specified by `space_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
space_id = H5D.get_space(dset_id);
num_points1 = H5S.get_select_npoints(space_id);
H5S.select_none(space_id);
num_points2 = H5S.get_select_npoints(space_id);
H5S.select_all(space_id);
num_points3 = H5S.get_select_npoints(space_id);
```

H5S.select_elements

Specify coordinates to include in selection

Syntax

```
H5S.select_elements(space_id,op,h5_coord)
```

Description

`H5S.select_elements(space_id,op,h5_coord)` selects the array elements to be included in the selection for the data space specified by `space_id`. The `op` argument determines how the new selection is to be combined with the previously existing selection for the data space. Specify `op` as one of these character vectors or string scalars.

`'H5S_SELECT_SET'`

`'H5S_SELECT_APPEND'`

`'H5S_SELECT_PREPEND'`

`h5_coord` is a two-dimensional array of 0-based values specifying the coordinates of the elements being selected. If `m` is the rank of the data space and if `n` is the number of points, then `h5_coord` should be an `m`-by-`n` array.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_coord` parameter assumes coordinates have C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

Select the corner points of a data space. In this case, `h5_coord` should have size 2x4.

```
dims = [100 200];
h5_dims = fliplr(dims);
```

```
space_id = H5S.create_simple(2,h5_dims,h5_dims);
coords = [0 0; 0 199; 99 0; 99 199];
h5_coords = fliplr(coords);
h5_coords = h5_coords';
H5S.select_elements(space_id,'H5S_SELECT_SET',h5_coords);
```

See Also

H5S.create_simple | H5S.get_select_elem_npoints | H5S.get_select_elem_pointlist

H5S.select_hyperslab

Select hyperslab region

Syntax

```
H5S.select_hyperslab(space_id,op,h5_start,h5_stride,h5_count,h5_block)
```

Description

`H5S.select_hyperslab(space_id,op,h5_start,h5_stride,h5_count,h5_block)` selects a hyperslab region to add to the current selected region for the data space specified by `space_id`. The `op` argument determines how the new selection is to be combined with the previously existing selection for the data space. Specify `op` as one of these character vectors or string scalars: `'H5S_SELECT_SET'`, `'H5S_SELECT_OR'`, `'H5S_SELECT_AND'`, `'H5S_SELECT_XOR'`, `'H5S_SELECT_NOTA'`, or `'H5S_SELECT_NOTB'`.

The `h5_start` array determines the starting coordinates of the hyperslab to select. The `h5_count` array determines how many blocks to select from the data space, in each dimension. The `h5_stride` array specifies how many elements to move in each dimension. The `h5_block` array determines the size of the element block selected from the data space.

If `h5_stride` is specified as `[]`, then a contiguous hyperslab is selected, as if each value in `h5_stride` were set to 1. If `h5_count` is specified as `[]`, the number of blocks selected along each dimension defaults to 1. If `h5_block` is specified as `[]`, then the block size defaults to a single element in each dimension, as if each value in the block array were set to 1.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_start`, `h5_stride`, `h5_count` and `h5_block` parameters assume C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
start = fliplr([10 20]); block = fliplr([20 30]);
H5S.select_hyperslab(space_id,'H5S_SELECT_SET',start,[],[],block);
```

See Also

H5S.create_simple

H5S.select_none

Reset selection region to include no elements

Syntax

```
H5S.select_none(space_id)
```

Description

`H5S.select_none(space_id)` resets the selection region for the data space `space_id` to include no elements.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
space_id = H5D.get_space(dset_id);
num_points1 = H5S.get_select_npoints(space_id);
H5S.select_none(space_id);
num_points2 = H5S.get_select_npoints(space_id);
```

H5S.select_valid

Determine validity of selection

Syntax

```
output = H5S.select_valid(space_id)
```

Description

`output = H5S.select_valid(space_id)` returns a positive value if the selection of the data space `space_id` is within the extent of that data space, and zero if it is not. A negative value indicates failure.

Examples

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
start = fliplr([90 190]); count = [11 11];
H5S.select_hyperslab(space_id,'H5S_SELECT_SET',start,[],count,[]);
valid = H5S.select_valid(space_id);
```

See Also

`H5S.create_simple` | `H5S.select_hyperslab`

H5S.set_extent_none

Remove extent from data space

Syntax

```
H5S.set_extent_none(space_id)
```

Description

`H5S.set_extent_none(space_id)` removes the extent from a data space and sets the type to `H5S_NO_CLASS`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer2D');
space_id = H5D.get_space(dset_id);
H5S.set_extent_none(space_id);
extent_type = H5S.get_simple_extent_type(space_id);
switch(extent_type)
    case H5ML.get_constant_value('H5S_SCALAR')
        fprintf('scalar\n');
    case H5ML.get_constant_value('H5S_SIMPLE')
        fprintf('simple\n');
    case H5ML.get_constant_value('H5S_NO_CLASS')
        fprintf('no class\n');
end
```

See Also

`H5S.get_simple_extent_dims`

H5S.set_extent_simple

Set size of data space

Syntax

```
H5S.set_extent_simple(space_id,rank,h5_dims,h5_maxdims)
```

Description

`H5S.set_extent_simple(space_id,rank,h5_dims,h5_maxdims)` sets the size of the data space identified by `space_id`. The `rank` argument is the number of dimensions used in the data space. `h5_dims` is an array specifying the size of each dimension of the dataset. `h5_maxdims` is an array specifying the upper limit on the size of each dimension.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_dims` and `h5_maxdims` parameters assume C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
space_id = H5S.create('H5S_SIMPLE');
dims = [100 200];
h5_dims = fliplr(dims);
maxdims = [100 H5ML.get_constant_value('H5S_UNLIMITED')];
h5_maxdims = fliplr(maxdims);
H5S.set_extent_simple(space_id,2,h5_dims, h5_maxdims);
```

See Also

`H5ML.get_constant_value` | `H5S.create` | `H5S.get_simple_extent_dims`

H5T.close

Close data type

Syntax

```
H5T.close(type_id)
```

Description

`H5T.close(type_id)` releases the data type specified by `type_id`.

See Also

`H5A.get_type` | `H5D.get_type`

H5T.commit

Commit transient data type

Syntax

```
H5T.commit(loc_id,name,type_id)
H5T.commit(loc_id,name,type_id,lcpl_id,tcpl_id,tapl_id)
```

Description

`H5T.commit(loc_id,name,type_id)` commits a transient data type to a file, creating a new named data type. `loc_id` is a file or group identifier. `name` is the name of the data type and `type_id` is the data type id. This interface corresponds to the 1.6.x version of `H5Tcommit`.

`H5T.commit(loc_id,name,type_id,lcpl_id,tcpl_id,tapl_id)` commits a transient data type to a file, creating a new named data type. `loc_id` is a file or group identifier. `name` is the name of the data type and `type_id` is the data type id. `lcpl_id`, `tcpl_id`, and `tapl_id` are link creation, data type creation, and data type access property list identifiers. This interface corresponds to the 1.8.x version of `H5Tcommit`.

Examples

Create a named variable-length data type.

```
plist_id = 'H5P_DEFAULT';
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',plist_id,plist_id);
base_type_id = H5T.copy('H5T_NATIVE_DOUBLE');
vlen_type_id = H5T.vlen_create(base_type_id);
H5T.commit(fid,'MyVlen',vlen_type_id);
H5T.close(vlen_type_id);
H5T.close(base_type_id);
H5F.close(fid);
```

See Also

`H5T.close` | `H5T.committed`

H5T.committed

Determine if data type is committed

Syntax

```
output = H5T.committed(type_id)
```

Description

`output = H5T.committed(type_id)` returns a positive value to indicate that the data type has been committed, and zero to indicate that it has not. A negative value indicates failure.

Examples

```
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
is_committed = H5T.committed(type_id);
```

See Also

`H5T.commit`

H5T.copy

Copy data type

Syntax

```
output_type_id = H5T.copy(type_id)
```

Description

`output_type_id = H5T.copy(type_id)` copies the existing data type identifier, a dataset identifier specified by `type_id`, or a predefined data type such as `'H5T_NATIVE_DOUBLE'`. `output_type_id` is a data type identifier.

Examples

```
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
type_size = H5T.get_size(type_id);
```

See Also

`H5T.get_size`

H5T.create

Create new data type

Syntax

```
output = H5T.create(class_id,size)
```

Description

`output = H5T.create(class_id,size)` creates a new data type of the class specified by `class_id`, with the number of bytes specified by `size`. The `output` argument is a data type identifier.

Examples

Create a signed 32-bit enumerated data type.

```
type_id = H5T.create('H5T_ENUM',4);
H5T.set_order(type_id,'H5T_ORDER_LE');
H5T.set_sign(type_id,'H5T_SGN_2');
H5T.enum_insert(type_id,'black',0);
H5T.enum_insert(type_id,'white',1);
```

See Also

`H5T.set_order` | `H5T.set_sign`

H5T.detect_class

Determine of data type contains specific class

Syntax

```
output = H5T.detect_class(type_id,class_id)
```

Description

`output = H5T.detect_class(type_id,class_id)` returns a positive value if the data type specified in `type_id` contains any data types of the data type class specified in `class_id`, or zero to indicate that it does not. A negative value indicates failure.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/vlen');
type_id = H5D.get_type(dset_id);
has_double = H5T.detect_class(type_id,'H5T_FLOAT');
```

See Also

`H5D.get_type`

H5T.equal

Determine equality of data types

Syntax

```
output = H5T.equal(type1_id,type2_id)
```

Description

`output = H5T.equal(type1_id,type2_id)` returns a positive number if the data type identifiers refer to the same data type, and zero to indicate that they do not. A negative value indicates failure. Either of the input values could be a character vector or string scalar specifying an HDF5 data type.

Examples

Determine if the data type of a dataset is a 32-bit little endian integer.

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer2D');
dtype_id = H5D.get_type(dset_id);
if H5T.equal(dtype_id,'H5T_STD_I32LE')
    fprintf('32-bit little endian integer\n');
end
```

See Also

`H5D.get_type`

H5T.get_class

Data type class identifier

Syntax

```
class_id = H5T.get_class(type_id)
```

Description

`class_id = H5T.get_class(type_id)` returns the data type class identifier of the data type specified by `type_id`.

Valid class identifiers include:

```
'H5T_INTEGER'
'H5T_FLOAT'
'H5T_STRING'
'H5T_BITFIELD'
'H5T_OPAQUE'
'H5T_COMPOUND'
'H5T_ENUM'
'H5T_VLEN'
'H5T_ARRAY'
```

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/enum');
type_id = H5D.get_type(dset_id);
class_id = H5T.get_class(type_id);
switch(class_id)
    case H5ML.get_constant_value('H5T_INTEGER')
```

```
        fprintf('Integer\n');
    case H5ML.get_constant_value('H5T_FLOAT')
        fprintf('Floating point\n');
    case H5ML.get_constant_value('H5T_STRING')
        fprintf('String\n');
    case H5ML.get_constant_value('H5T_BITFIELD')
        fprintf('Bitfield\n');
    case H5ML.get_constant_value('H5T_OPAQUE')
        fprintf('Opaque\n');
    case H5ML.get_constant_value('H5T_COMPOUND')
        fprintf('Compound'\n');
    case H5ML.get_constant_value('H5T_ENUM')
        fprintf('Enumerated\n');
    case H5ML.get_constant_value('H5T_VLEN')
        fprintf('Variable length\n');
    case H5ML.get_constant_value('H5T_ARRAY')
        fprintf('Array\n');
end
```

See Also

H5ML.get_constant_value

H5T.get_create_plist

Copy of data type creation property list

Syntax

```
plist_id = H5T.get_create_plist(datatype_id)
```

Description

`plist_id = H5T.get_create_plist(datatype_id)` returns a property list identifier for the data type creation property list associated with the data type specified by `datatype_id`.

See Also

`H5D.get_create_plist` | `H5F.get_create_plist`

H5T.get_native_type

Native data type of dataset data type

Syntax

```
output = H5T.get_native_type(type_id,direction)
```

Description

`output = H5T.get_native_type(type_id,direction)` returns the equivalent native data type for the dataset data type specified in `type_id`. The `direction` argument indicates the order in which the library searches for a native data type match and must be either `'H5T_DIR_ASCEND'` or `'H5T_DIR_DESCEND'`.

H5T.get_size

Size of data type in bytes

Syntax

```
type_size = H5T.get_size(type_id)
```

Description

`type_size = H5T.get_size(type_id)` returns the size of a data type in bytes. `type_id` is a data type identifier.

Examples

Determine the size of the data type for a specific dataset.

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/bitfield2D');
type_id = H5D.get_type(dset_id);
type_size = H5T.get_size(type_id);
```

See Also

`H5D.get_type` | `H5T.set_size`

H5T.get_super

Base data type

Syntax

```
super_type_id = H5T.get_super(type_id)
```

Description

`super_type_id = H5T.get_super(type_id)` returns the base data type from which the data type type specified by `type_id` is derived.

Examples

Retrieve the base data type for an enumerated dataset.

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/enum');
dtype_id = H5D.get_type(dset_id);
super_type_id = H5T.get_super(dtype_id);
```

H5T.lock

Lock data type

Syntax

```
H5T.lock(type_id)
```

Description

`H5T.lock(type_id)` locks the data type specified by `type_id`, making it read-only and non-destructible.

H5T.open

Open named data type

Syntax

```
type_id = H5T.open(loc_id,name)
```

Description

`type_id = H5T.open(loc_id,name)` opens a named data type at the location specified by `loc_id` and returns an identifier for the data type. `loc_id` is either a file or group identifier.

This function corresponds to the `H5Topen1` function in the HDF5 library C API.

See Also

`H5A.open` | `H5D.open` | `H5G.open` | `H5O.open` | `H5T.close`

H5T.array_create

Create array data type object

Syntax

```
array_type_id = H5T.array_create(base_id,h5_dims)
array_type_id = H5T.array_create(base_id,rank,h5_dims,perms)
```

Description

`array_type_id = H5T.array_create(base_id,h5_dims)` creates a new array data type object. This interface corresponds to the 1.8 library version of `H5Tarray_create`.

`array_type_id = H5T.array_create(base_id,rank,h5_dims,perms)` creates a new array data type object. This interface corresponds to the 1.6 library version of `H5Tarray_create`. The `perms` parameter is not used at this time and can be omitted.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_dims` parameter assumes C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

Create a 100-by-200 double precision array data type.

```
base_type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [100 200];
h5_dims = fliplr(dims);
array_type = H5T.array_create(base_type_id,h5_dims);
```

See Also

`H5T.get_array_dims` | `H5T.get_array_ndims`

H5T.get_array_dims

Array dimension extents

Syntax

```
dimsizes = H5T.get_array_dims(type_id)
[ndims,dimsizes,perm] = H5T.get_array_dims(type_id)
```

Description

`dimsizes = H5T.get_array_dims(type_id)` returns the sizes of the dimensions and the dimension permutations of the specified array data type object. This interface corresponds to the 1.8 version of `H5Tget_array_dims`.

`[ndims,dimsizes,perm] = H5T.get_array_dims(type_id)` corresponds to the 1.6 version of the interface. It is strongly deprecated.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/array2D');
type_id = H5D.get_type(dset_id);
h5_dims = H5T.get_array_dims(type_id);
dims = fliplr(h5_dims);
```

See Also

`H5T.array_create` | `H5T.get_array_ndims`

H5T.get_array_ndims

Rank of array data type

Syntax

```
output = H5T.get_array_ndims(type_id)
```

Description

`output = H5T.get_array_ndims(type_id)` returns the rank, the number of dimensions, of an array data type object.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/array2D');
type_id = H5D.get_type(dset_id);
ndims = H5T.get_array_ndims(type_id);
```

See Also

`H5T.get_array_dims`

H5T.get_cset

Character set of string data type

Syntax

```
cset = H5T.get_cset(type_id)
```

Description

`cset = H5T.get_cset(type_id)` returns the character set type of the data type specified by `type_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/string');
type_id = H5D.get_type(dset_id);
cset = H5T.get_cset(type_id);
switch(cset)
  case H5ML.get_constant_value('H5T_CSET_ASCII')
      fprintf('ASCII\n');
  case H5ML.get_constant_value('H5T_CSET_UTF8')
      fprintf('UTF-8\n');
end
```

See Also

`H5T.set_cset`

H5T.get_ebias

Exponent bias of floating-point type

Syntax

```
output = H5T.get_ebias(type_id)
```

Description

`output = H5T.get_ebias(type_id)` returns the exponent bias of a floating-point type. `type_id` is data type identifier.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/float');
type_id = H5D.get_type(dset_id);
ebias = H5T.get_ebias(type_id);
```

See Also

`H5T.set_ebias`

H5T.get_fields

Floating-point data type bit field information

Syntax

```
[spos,epos,esize,mpos,msize] = H5T.get_fields(type_id)
```

Description

`[spos,epos,esize,mpos,msize] = H5T.get_fields(type_id)` returns information about the locations of the various bit fields of a floating point data type. `type_id` is a data type identifier. `spos` is the floating-point sign bit. `epos` is the exponent bit-position. `esize` is the size of the exponent in bits. `mpos` is the mantissa bit-position. `msize` is the size of the mantissa in bits.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/float');
type_id = H5D.get_type(dset_id);
[spos, epos, esize, mpos, msize] = H5T.get_fields(type_id);
```

H5T.get_inpad

Internal padding type for floating-point data types

Syntax

```
pad_type = H5T.get_inpad(type_id)
```

Description

`pad_type = H5T.get_inpad(type_id)` returns the internal padding type for unused bits in floating-point data types. `type_id` is a data type identifier. `pad_type` can be `H5T_PAD_ZERO`, `H5T_PAD_ONE`, or `H5T_PAD_BACKGROUND`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/float');
type_id = H5D.get_type(dset_id);
pad_type = H5T.get_inpad(type_id);
switch(pad_type)
    case H5ML.get_constant_value('H5T_PAD_ZERO')
        fprintf('pad zero\n');
    case H5ML.get_constant_value('H5T_PAD_ONE');
        fprintf('pad one\n');
    case H5ML.get_constant_value('H5T_PAD_BACKGROUND')
        fprintf('pad background\n');
end
```

See Also

`H5T.set_inpad`

H5T.get_norm

Mantissa normalization type

Syntax

```
norm_type = H5T.get_norm(type_id)
```

Description

`norm_type = H5T.get_norm(type_id)` returns the mantissa normalization of a floating-point data type. `type_id` is a data type identifier. `norm_type` can be `H5T_NORM_IMPLIED`, `H5T_NORM_MSBSET`, or `H5T_NORM_NONE`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/float');
type_id = H5D.get_type(dset_id);
norm_type = H5T.get_norm(type_id);
switch(norm_type)
    case H5ML.get_constant_value('H5T_NORM_IMPLIED')
        fprintf('MSB of mantissa is not stored, always 1\n');
    case H5ML.get_constant_value('H5T_NORM_MSBSET');
        fprintf('MSB of mantissa is always 1\n');
    case H5ML.get_constant_value('H5T_NORM_NONE')
        fprintf('mantissa is not normalized\n');
end
```

See Also

`H5T.set_norm`

H5T.get_offset

Bit offset of first significant bit

Syntax

```
offset = H5T.get_offset(type_id)
```

Description

`offset = H5T.get_offset(type_id)` returns the offset of the first significant bit. `type_id` is a data type identifier.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/float');
type_id = H5D.get_type(dset_id);
offset = H5T.get_offset(type_id);
```

See Also

`H5T.set_offset`

H5T.get_order

Byte order of atomic data type

Syntax

```
output = H5T.get_order(type_id)
```

Description

`output = H5T.get_order(type_id)` returns the byte order of an atomic data type. `type_id` is a data type identifier. Possible return values are the constant values corresponding to the following character vectors:

`'H5T_ORDER_LE'`

`'H5T_ORDER_BE'`

`'H5T_ORDER_VAX'`

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g2/dset2.2');
type_id = H5D.get_type(dset_id);
switch(H5T.get_order(type_id))
    case H5ML.get_constant_value('H5T_ORDER_LE')
        fprintf('little endian\n');
    case H5ML.get_constant_value('H5T_ORDER_BE')
        fprintf('big endian\n');
    case H5ML.get_constant_value('H5T_ORDER_VAX')
        fprintf('vax\n');
end
```

See Also

`H5ML.get_constant_value` | `H5T.set_order`

H5T.get_pad

Padding type of least and most-significant bits

Syntax

```
[lsb,msb] = H5T.get_pad(type_id)
```

Description

`[lsb,msb] = H5T.get_pad(type_id)` returns the padding type of the least and most-significant bit padding. `type_id` is a data type identifier. `lsb` is the least-significant bit padding type. `msb` is the most-significant bit padding type. Values for `lsb` and `msb` can be `H5T_PAD_ZERO`, `H5T_PAD_ONE`, or `H5T_PAD_BACKGROUND`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
type_id = H5D.get_type(dset_id);
[lsb,msb] = H5T.get_pad(type_id);
switch(lsb)
    case H5ML.get_constant_value('H5T_PAD_ZERO')
        fprintf('lsb pad type is zeros\n');
    case H5ML.get_constant_value('H5T_PAD_ONE');
        fprintf('lsb pad type is ones\n');
    case H5ML.get_constant_value('H5T_PAD_BACKGROUND')
        fprintf('lsb pad type is background\n');
end
switch(msb)
    case H5ML.get_constant_value('H5T_PAD_ZERO')
        fprintf('msb pad type is zeros\n');
    case H5ML.get_constant_value('H5T_PAD_ONE');
        fprintf('msb pad type is ones\n');
    case H5ML.get_constant_value('H5T_PAD_BACKGROUND')
        fprintf('msb pad type is background\n');
end
```

See Also

`H5T.set_pad`

H5T.get_precision

Precision of atomic data type

Syntax

```
output = H5T.get_precision(type_id)
```

Description

`output = H5T.get_precision(type_id)` returns the precision of an atomic data type. `type_id` is a data type identifier.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
type_id = H5D.get_type(dset_id);
numbits = H5T.get_precision(type_id);
```

See Also

`H5T.set_precision`

H5T.get_sign

Sign type for integer data type

Syntax

```
sign_type = H5T.get_sign(type_id)
```

Description

`sign_type = H5T.get_sign(type_id)` returns the sign type for an integer type. `type_id` is a data type identifier. Valid types are: `H5T_SGN_NONE` or `H5T_SGN_2`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
type_id = H5D.get_type(dset_id);
sign_type = H5T.get_sign(type_id);
switch(sign_type)
    case H5ML.get_constant_value('H5T_SGN_NONE')
        fprintf('Unsigned integer type.\n');
    case H5ML.get_constant_value('H5T_SGN_2');
        fprintf('Signed integer type.\n');
end
```

See Also

`H5T.set_sign`

H5T.get_strpad

Storage mechanism for string data type

Syntax

```
output = H5T.get_strpad(type_id)
```

Description

`output = H5T.get_strpad(type_id)` returns the storage mechanism (padding type) for a string data type. Possible values are:

`'H5T_STR_NULLPAD'`	Pad with zeros
`'H5T_STR_NULLTERM'`	Null-terminate
`'H5T_STR_SPACEPAD'`	Pad with spaces

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/string');
type_id = H5D.get_type(dset_id);
padding = H5T.get_strpad(type_id);
switch(padding)
    case H5ML.get_constant_value('H5T_STR_NULLTERM')
        fprintf('null-terminated\n');
    case H5ML.get_constant_value('H5T_STR_NULLPAD')
        fprintf('padded with zeros\n');
    case H5ML.get_constant_value('H5T_STR_SPACEPAD')
        fprintf('padded with spaces\n');
end
```

See Also

`H5T.set_strpad`

H5T.set_cset

Set character dataset for string data type

Syntax

```
H5T.set_cset(type_id,cset)
```

Description

`H5T.set_cset(type_id,cset)` sets the character encoding used to create strings. Specify `cset` as `'H5T_CSET_ASCII'`, or its equivalent numerical value.

Examples

```
type_id = H5T.copy('H5T_C_S1');
H5T.set_size(type_id,10);
encoding = H5ML.get_constant_value('H5T_CSET_ASCII');
H5T.set_cset(type_id,encoding);
```

See Also

`H5T.get_cset`

H5T.set_ebias

Set exponent bias of floating-point data type

Syntax

```
H5T.set_ebias(type_id,ebias)
```

Description

`H5T.set_ebias(type_id,ebias)` sets the exponent bias of a floating-point type. `type_id` is a data type identifier. `ebias` is an exponent bias value.

Examples

```
type_id = H5T.copy('H5T_NATIVE_FLOAT');
H5T.set_size(type_id,32);
H5T.set_ebias(type_id,99);
```

See Also

`H5T.get_ebias`

H5T.set_fields

Set sizes and locations of floating-point bit fields

Syntax

```
H5T.set_fields(type_id,spos,epos,esize,mpos,msize)
```

Description

`H5T.set_fields(type_id,spos,epos,esize,mpos,msize)` sets the locations and sizes of the various floating-point bit fields. `spos` is the sign position. `epos` is the exponent in bits. `esize` is the size of exponent in bits. `mpos` is the mantissa bit position. `msize` is the size of the mantissa in bits.

Examples

```
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
H5T.set_fields(type_id,30,24,6,0,2);
```

See Also

`H5T.get_fields`

H5T.set_inpad

Specify how unused internal bits are to be filled

Syntax

```
H5T.set_inpad(type_id,pad_type)
```

Description

`H5T.set_inpad(type_id,pad_type)` sets how unused internal bits of a floating point type are filled. `type_id` is the identifier of the data type. Specify how to fill the bits in `pad_type` which can be one of these character vectors or string scalars, or its equivalent numerical value:
`H5T_PAD_ZERO`
`H5T_PAD_ONE`
`H5T_PAD_BACKGROUND` (leave background alone)

Examples

```
type_id = H5T.copy('H5T_NATIVE_FLOAT');
pad_type = H5ML.get_constant_value('H5T_PAD_ZERO');
H5T.set_inpad(type_id,pad_type);
```

See Also

`H5T.get_inpad`

H5T.set_norm

Set mantissa normalization of floating-point data type

Syntax

```
H5T.set_norm(type_id,norm)
```

Description

`H5T.set_norm(type_id,norm)` sets the mantissa normalization of a floating-point data type. Specify `norm` as one of these character vectors or string scalars: `'H5T_NORM_IMPLIED'`, `'H5T_NORM_MSBSET'`, or `'H5T_NORM_NONE'`.

Examples

```
type_id = H5T.copy('H5T_NATIVE_FLOAT');
norm_type = H5ML.get_constant_value('H5T_NORM_MSBSET');
H5T.set_norm(type_id,norm_type);
```

See Also

`H5T.get_norm`

H5T.set_offset

Set bit offset of first significant bit

Syntax

```
H5T.set_offset(type_id,offset)
```

Description

`H5T.set_offset(type_id,offset)` sets the bit offset of the first significant bit. `type_id` is the identifier of the data type. `offset` specifies the number of bits of padding that appear.

Examples

```
type_id = H5T.copy('H5T_NATIVE_INT');
H5T.set_offset(type_id,16);
```

See Also

`H5T.get_offset`

H5T.set_order

Set byte ordering of atomic data type

Syntax

```
H5T.set_order(type_id,type_order)
```

Description

`H5T.set_order(type_id,type_order)` sets the byte ordering of an atomic data type. Specify `type_order` as one of these character vectors or string scalars.

- `'H5T_ORDER_LE'`
- `'H5T_ORDER_BE'`
- `'H5T_ORDER_VAX'`

Examples

Create a big endian 32-bit integer type.

```
type_id = H5T.copy('H5T_NATIVE_INT');
order = H5ML.get_constant_value('H5T_ORDER_BE');
H5T.set_order(type_id,order);
```

See Also

`H5ML.get_constant_value` | `H5T.get_order`

H5T.set_pad

Set padding type for least and most significant bits

Syntax

```
H5T.set_pad(type_id,lsb,msb)
```

Description

`H5T.set_pad(type_id,lsb,msb)` sets the padding type for the least and most-significant bits. `type_id` is the identifier of the data type. `lsb` specifies the padding type for least-significant bits; `msb` for most-significant bits. Specify padding types as one of these character vectors or string scalars: `'H5T_PAD_ZERO'`, `'H5T_PAD_ONE'`, or `'H5T_PAD_BACKGROUND'` (leave background alone).

Examples

```
type_id = H5T.copy('H5T_NATIVE_INT');
lsb = H5ML.get_constant_value('H5T_PAD_ONE');
msb = H5ML.get_constant_value('H5T_PAD_ZERO');
H5T.set_pad(type_id,lsb,msb);
```

See Also

`H5T.get_pad`

H5T.set_precision

Set precision of atomic data type

Syntax

```
H5T.set_precision(type_id,prec)
```

Description

`H5T.set_precision(type_id,prec)` sets the precision of an atomic data type. `type_id` is a data type identifier. `prec` specifies the number of bits of precision for the data type.

H5T.set_sign

Set sign property for integer data type

Syntax

```
H5T.set_sign(type_id,sign)
```

Description

`H5T.set_sign(type_id,sign)` sets the sign property for an integer type. `type_id` is a data type identifier. Specify the sign type `sign` as `'H5T_SGN_NONE'` or `'H5T_SGN_2'`.

Examples

```
type_id = H5T.copy('H5T_NATIVE_LONG');
sgn = H5ML.get_constant_value('H5T_SGN_NONE');
H5T.set_sign(type_id,sgn);
```

See Also

`H5T.get_sign`

H5T.set_size

Set size of data type in bytes

Syntax

```
H5T.set_size(type_id,type_size)
```

Description

`H5T.set_size(type_id,type_size)` sets the total size in bytes for the data type specified by `type_id`. If a variable length string is desired then `type_size` can be specified as `'H5T_VARIABLE'`.

Examples

Create a variable length string with null termination.

```
type_id = H5T.copy('H5T_C_S1');
H5T.set_size(type_id,'H5T_VARIABLE');
H5T.set_strpad(type_id,'H5T_STR_NULLTERM');
```

See Also

`H5T.get_size`

H5T.set_strpad

Set storage mechanism for string data type

Syntax

```
H5T.set_strpad(type_id,storage_type)
```

Description

`H5T.set_strpad(type_id,storage_type)` defines the storage mechanism for the string data type identified by `type_id`. The storage type may be one of the following values.

`'H5T_STR_NULLTERM'`	Null terminated
`'H5T_STR_NULLPAD'`	Padded with zeros
`'H5T_STR_SPACEPAD'`	Padded with spaces

Examples

Create a ten-character string data type with space padding.

```
type_id = H5T.copy('H5T_C_S1');
H5T.set_size(type_id,10);
H5T.set_strpad(type_id,'H5T_STR_SPACEPAD');
```

See Also

`H5T.get_strpad`

H5T.get_member_class

Data type class for compound data type member

Syntax

```
output = H5T.get_member_class(type_id,membno)
```

Description

`output = H5T.get_member_class(type_id,membno)` returns the data type class of the compound data type member specified by `membno`. The `type_id` argument is the data type identifier of a compound object.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/compound');
type_id = H5D.get_type(dset_id);
member_name = H5T.get_member_name(type_id,0);
member_class = H5T.get_member_class(type_id,0);
```

See Also

`H5T.get_member_name`

H5T.get_member_index

Index of compound or enumeration type member

Syntax

```
idx = H5T.get_member_index(type_id,name)
```

Description

`idx = H5T.get_member_index(type_id,name)` returns the index of a field of a compound data type or an element of an enumeration data type. `type_id` is a data type identifier and name is a character vector or string scalar that specifies the target field or element.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/compound');
type_id = H5D.get_type(dset_id);
idx = H5T.get_member_index(type_id,'b');
```

See Also

`H5T.get_member_name`

H5T.get_member_name

Name of compound or enumeration type member

Syntax

```
name = H5T.get_member_name(type_id,membno)
```

Description

`name = H5T.get_member_name(type_id,membno)` returns the name of a field of a compound data type or an element of an enumeration data type. `type_id` is a data type identifier. `membno` is a zero-based index of the field or element whose name is to be retrieved.

Examples

Determine the name of the first field of a compound dataset.

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/compound');
dtype_id = H5D.get_type(dset_id);
member_name = H5T.get_member_name(dtype_id,0);
```

See Also

`H5T.get_member_index`

H5T.get_member_offset

Offset of field of compound data type

Syntax

```
output = H5T.get_member_offset(type_id,membno)
```

Description

`output = H5T.get_member_offset(type_id,membno)` returns the byte offset of the field specified by `membno` in the compound data type specified by `type_id`. Note that zero (0) is a valid offset.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/compound');
type_id = H5D.get_type(dset_id);
idx = H5T.get_member_offset(type_id,1);
```

See Also

`H5T.get_member_name`

H5T.get_member_type

Data type of specified member

Syntax

```
type_id = H5T.get_member_type(type_id,membno)
```

Description

`type_id = H5T.get_member_type(type_id,membno)` returns the data type of the member specified by `membno` in the data type specified by `type_id`.

Examples

Get the size of the data type of the first member of a compound data type.

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/compound');
compound_type_id = H5D.get_type(dset_id);
member_type_id = H5T.get_member_type(compound_type_id,0);
type_size = H5T.get_size(member_type_id);
```

See Also

`H5D.get_type`

H5T.get_nmembers

Number of elements in enumeration type

Syntax

```
output = H5T.get_nmembers(type_id)
```

Description

`output = H5T.get_nmembers(type_id)` retrieves the number of fields in a compound data type or the number of members of an enumeration data type. `type_id` is a data type identifier.

Examples

Determine the number of fields in a compound dataset.

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/compound');
dtype_id = H5D.get_type(dset_id);
nmembers = H5T.get_nmembers(dtype_id);
```

H5T.insert

Add member to compound data type

Syntax

```
H5T.insert(type_id,name,offset,member_datatype)
```

Description

`H5T.insert(type_id,name,offset,member_datatype)` adds another member to the compound data type specified by `type_id`. The `name` argument is a character vector or string scalar that specifies the name of the new member, which must be unique in the compound data type. `offset` specifies where you want to insert the new member and `member_datatype` specifies the data type identifier of the new member.

Examples

```
type_id = H5T.create('H5T_COMPOUND',16);
H5T.insert(type_id,'first',0,'H5T_NATIVE_DOUBLE');
H5T.insert(type_id,'second',8,'H5T_NATIVE_INT');
H5T.insert(type_id,'third',12,'H5T_NATIVE_UINT');
```

See Also

`H5T.create`

H5T.pack

Recursively remove padding from compound data type

Syntax

```
H5T.pack(type_id)
```

Description

`H5T.pack(type_id)` recursively removes padding from within a compound data type to make it more efficient (space-wise) to store that data. `type_id` is a data type identifier.

H5T.enum_create

Create new enumeration data type

Syntax

```
output = H5T.enum_create(parent_id)
```

Description

`output = H5T.enum_create(parent_id)` creates a new enumeration data type based on the specified base data type, `parent_id`, which must be an integer type. `output` is a data type identifier for the new enumeration data type.

Examples

```
parent_id = H5T.copy('H5T_NATIVE_UINT');
type_id = H5T.enum_create(parent_id);
H5T.enum_insert(type_id,'red',1);
H5T.enum_insert(type_id,'green',2);
H5T.enum_insert(type_id,'blue',3);
H5T.close(type_id);
H5T.close(parent_id);
```

See Also

`H5T.enum_insert`

H5T.enum_insert

Insert enumeration data type member

Syntax

```
H5T.enum_insert(type_id,name,value)
```

Description

`H5T.enum_insert(type_id,name,value)` inserts a new enumeration data type member into the enumeration data type specified by `type_id`. The `name` argument is a character vector or string scalar that specifies the name of the new member of the enumeration and `value` is the value of the member.

Examples

```
parent_id = H5T.copy('H5T_NATIVE_UINT');
type_id = H5T.enum_create(parent_id);
H5T.enum_insert(type_id,'red',1);
H5T.enum_insert(type_id,'green',2);
H5T.enum_insert(type_id,'blue',3);
H5T.close(type_id);
H5T.close(parent_id);
```

See Also

`H5T.enum_create`

H5T.enum_nameof

Name of enumeration data type member

Syntax

```
name = H5T.enum_nameof(type_id,value)
```

Description

`name = H5T.enum_nameof(type_id,value)` returns the symbol name corresponding to a member of an enumeration data type. `type_id` specifies the enumeration data type. `value` identifies the member of the enumeration.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/enum');
type_id = H5D.get_type(dset_id);
name0 = H5T.enum_nameof(type_id,int32(0));
name1 = H5T.enum_nameof(type_id,int32(1));
```

See Also

`H5T.enum_valueof`

H5T.enum_valueof

Value of enumeration data type member

Syntax

```
value = H5T.enum_valueof(type_id,member_name)
```

Description

`value = H5T.enum_valueof(type_id,member_name)` returns the value corresponding to a specified member of an enumeration data type. `type_id` specifies the enumeration data type and `member_name` specifies the member.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/enum');
type_id = H5D.get_type(dset_id);
num_members = H5T.get_nmembers(type_id);
for j = 1:num_members
    member_name{j} = H5T.get_member_name(type_id,j-1);
    member_value(j) = H5T.enum_valueof(type_id,member_name{j});
end
```

See Also

`H5T.get_member_name` | `H5T.get_nmembers`

H5T.get_member_value

Value of enumeration data type member

Syntax

```
value = H5T.get_member_value(type_id,membno)
```

Description

`value = H5T.get_member_value(type_id,membno)` returns the value of the enumeration data type member specified by `membno`. The `type_id` argument is the data type identifier for the enumeration data type.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/enum');
type_id = H5D.get_type(dset_id);
num_members = H5T.get_nmembers(type_id);
for j = 1:num_members
    member_name{j} = H5T.get_member_name(type_id,j-1);
    member_value(j) = H5T.get_member_value(type_id,j-1);
end
```

See Also

`H5T.get_member_name` | `H5T.get_nmembers`

H5T.get_tag

Tag associated with opaque data type

Syntax

```
tag = H5T.get_tag(type_id)
```

Description

`tag = H5T.get_tag(type_id)` returns the tag associated with the opaque data type specified by `type_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/opaque');
dtype_id = H5D.get_type(dset_id);
tag = H5T.get_tag(dtype_id);
```

See Also

`H5T.set_tag`

H5T.set_tag

Tag opaque data type with description

Syntax

```
H5T.set_tag(type_id,tag)
```

Description

`H5T.set_tag(type_id,tag)` tags the opaque data type specified by `type_id`, with the descriptive text, `tag`. Specify `tag` as a character vector or string scalar.

Examples

Create an opaque data type with a length of 4 bytes and a particular tag.

```
type_id = H5T.create('H5T_OPAQUE',4);
H5T.set_tag(type_id,'Created by MATLAB.');
```

See Also

`H5T.create` | `H5T.get_tag`

H5T.is_variable_str

Determine if data type is variable-length string

Syntax

```
output = H5T.is_variable_str(type_id)
```

Description

`output = H5T.is_variable_str(type_id)` returns a positive value if the data type specified by `type_id` is a variable-length string and zero if it is not. A negative value indicates failure.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/VLstring2D');
type_id = H5D.get_type(dset_id);
if H5T.is_variable_str(type_id) > 0
    fprintf('variable length string\n');
end
```

See Also

`H5D.get_type` | `H5T.get_size` | `H5T.vlen_create`

H5T.vlen_create

Create new variable-length data type

Syntax

```
vlen_type_id = H5T.vlen_create(base_id)
```

Description

`vlen_type_id = H5T.vlen_create(base_id)` creates a new variable-length (VL) data type. `base_id` specifies the base type of the data type to create.

Examples

Create a variable length data type for 64-bit floating-point numbers.

```
base_type_id = H5T.copy('H5T_NATIVE_DOUBLE');
vlen_type_id = H5T.vlen_create(base_type_id);
```

See Also

`H5T.is_variable_str`

H5Z.filter_avail

Determine if filter is available

Syntax

```
output = H5Z.filter_avail(filter_id)
```

Description

`output = H5Z.filter_avail(filter_id)` determines whether the filter specified by the filter identifier is available to the application. `filter_id` can be specified by one of the following character vectors or string scalars, or its numeric equivalent.

```
'H5Z_FILTER_DEFLATE'
'H5Z_FILTER_SHUFFLE'
'H5Z_FILTER_FLETCHER32'
'H5Z_FILTER_SZIP'
'H5Z_FILTER_NBIT'
'H5Z_FILTER_SCALEOFFSET'
```

Examples

Determine if the shuffle filter is available.

```
bool = H5Z.filter_avail('H5Z_FILTER_SHUFFLE');
```

See Also

`H5ML.get_constant_value`

H5Z.get_filter_info

Information about filter

Syntax

```
filter_config_flags = H5Z.get_filter_info(filter)
```

Description

`filter_config_flags = H5Z.get_filter_info(filter)` retrieves information about the filter specified by its identifier. At present, the information returned is the filter's configuration flags, indicating whether the filter is configured to decode data, to encode data, neither, or both. `filter_config_flags` should be used with the HDF5 constant values `'H5Z_FILTER_CONFIG_ENCODE_ENABLED'` and `'H5Z_FILTER_CONFIG_DECODE_ENABLED'` in a bitwise AND operation. If the resulting value is 0, then the encode or decode functionality is not available.

Examples

Determine if encoding is enabled for the deflate filter.

```
flags = H5Z.get_filter_info('H5Z_FILTER_DEFLATE');
functionality = H5ML.get_constant_value('H5Z_FILTER_CONFIG_ENCODE_ENABLED');
enabled = bitand(flags,functionality) > 0;
```

See Also

`H5ML.get_constant_value` | `H5Z.filter_avail` | `bitand`

hadamard

Hadamard matrix

Syntax

```
H = hadamard(n)
H = hadamard(n,classname)
```

Description

`H = hadamard(n)` returns the "Hadamard Matrix" on page 1-6133 of order `n`.

`H = hadamard(n,classname)` returns a matrix of class `classname`, which can be either `'single'` or `'double'`.

Examples

Compute Hadamard Matrix

Compute the 4-by-4 Hadamard matrix.

```
H = hadamard(4)
```

```
H = 4×4

     1     1     1     1
     1    -1     1    -1
     1     1    -1    -1
     1    -1    -1     1
```

Input Arguments

n — Matrix order
scalar, nonnegative integer

Matrix order, specified as a scalar, nonnegative integer.

Example: `hadamard(5)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`classname` — Matrix class
`'double'` (default) | `'single'`

Matrix class, specified as either `'double'` or `'single'`.

Example: `hadamard(5,'single')`

Data Types: `char`

Limitations

- An `n`-by-`n` Hadamard matrix with `n > 2` exists only if `rem(n,4) = 0`. This function handles only the cases where `n`, `n/12`, or `n/20` is a power of 2.

More About

Hadamard Matrix

Hadamard matrices are matrices of `1`'s and `-1`'s whose columns are orthogonal,

```
H'*H = n*I
```

where `[n n]=size(H)` and `I = eye(n,n)`.

They have applications in several different areas, including combinatorics, signal processing, and numerical analysis [1], [2].

References

[1] Ryser, Herbert J. *Combinatorial Mathematics*. Mathematical Association of America, 1963.

[2] Pratt, William K. *Digital Signal Processing*. New York, NY: John Wiley and Sons, 1978.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `n` must be a fixed-size scalar.

See Also

`compan` | `hankel` | `toeplitz`

Introduced before R2006a

matlab.io.datastore.HadoopFileBased class

Package: `matlab.io.datastore`

(Not recommended) Add Hadoop file support to datastore

Note `matlab.io.datastore.HadoopFileBased` is not recommended. Use `matlab.io.datastore.HadoopLocationBased` instead.

Description

`matlab.io.datastore.HadoopFileBased` is an abstract mixin class that adds Hadoop support to your custom datastore.

To use this mixin class, you must inherit from the `matlab.io.datastore.HadoopFileBased` class in addition to inheriting from the `matlab.io.Datastore` base class. Type the following syntax as the first line of your class definition file:

```
classdef MyDatastore < matlab.io.Datastore & ...
                            matlab.io.datastore.HadoopFileBased
    ...
end
```

To add Hadoop support along with parallel processing support, use these lines in your class definition file:

```
classdef MyDatastore < matlab.io.Datastore & ...
                            matlab.io.datastore.Partitionable & ...
                            matlab.io.datastore.HadoopFileBased
    ...
end
```

To add support for Hadoop to your custom datastore, you must:

- Inherit from the additional class `matlab.io.datastore.HadoopFileBased`
- Define these additional methods: `getLocation`, `initializeDatastore`, and `isfullfile`

For more details and steps to create your custom datastore with support for Hadoop, see “Develop Custom Datastore”.

Methods

getLocation	(Not recommended) Location of files in Hadoop
initializeDatastore	(Not recommended) Initialize datastore with information from Hadoop
isfullfile	(Not recommended) Check if datastore reads full files

Attributes

`Sealed`	`false`

For information on class attributes, see “Class Attributes”.

Examples

Build Datastore with Hadoop Support

Implement a datastore with parallel processing and Hadoop support and use it to bring your data from the Hadoop server into MATLAB. Then use the `tall` and `gather` functions on this data.

Create a new `.m` class definition file that contains the code implementing your custom datastore. You must save this file in your working folder or in a folder that is on the MATLAB path. The name of the `.m` file must be the same as the name of your object constructor function. For example, if you want your constructor function to have the name `MyDatastoreHadoop`, then the name of the script file must be `MyDatastoreHadoop.m`. The `.m` class definition file must contain these steps:

- Step 1: Inherit from the datastore classes.
- Step 2: Define the constructor and the required methods.
- Step 3: Define your custom file reading function.

This code shows the three steps in a sample implementation of a custom datastore that can read binary files from a Hadoop server.

```
%% STEP 1: INHERIT FROM DATASTORE CLASSES
classdef MyDatastoreHadoop < matlab.io.Datastore & ...
        matlab.io.datastore.Partitionable & ...
        matlab.io.datastore.HadoopFileBased

    properties (Access = private)
        CurrentFileIndex double
        FileSet matlab.io.datastore.DsFileSet
    end


%% STEP 2: DEFINE THE CONSTRUCTOR AND THE REQUIRED METHODS
    methods
        % Define your datastore constructor
        function myds = MyDatastoreHadoop(location,altRoots)
            myds.FileSet = matlab.io.datastore.DsFileSet(location,...
                'FileExtensions','.bin', ...
                'FileSplitSize',8*1024);
            myds.CurrentFileIndex = 1;

            if nargin == 2
                 myds.AlternateFileSystemRoots = altRoots;
            end

            reset(myds);
        end

        % Define the hasdata method
        function tf = hasdata(myds)
            % Return true if more data is available
            tf = hasfile(myds.FileSet);
        end

        % Define the read method
        function [data,info] = read(myds)
            % Read data and information about the extracted data
            % See also: MyFileReader()
            if ~hasdata(myds)
                error(sprintf(['No more data to read.\nUse the reset ',...
                     'method to reset the datastore to the start of ' ,...
                     'the data. \nBefore calling the read method, ',...
                     'check if data is available to read ',...
                     'by using the hasdata method.']))
            end

            fileInfoTbl = nextfile(myds.FileSet);
            data = MyFileReader(fileInfoTbl);
            info.Size = size(data);
            info.FileName = fileInfoTbl.FileName;
            info.Offset = fileInfoTbl.Offset;

            % Update CurrentFileIndex for tracking progress
            if fileInfoTbl.Offset + fileInfoTbl.SplitSize >= ...
                    fileInfoTbl.FileSize
                myds.CurrentFileIndex = myds.CurrentFileIndex + 1 ;
```

```
        end
    end

    % Define the reset method
    function reset(myds)
        % Reset to the start of the data
        reset(myds.FileSet);
        myds.CurrentFileIndex = 1;
    end


    % Define the partition method
    function subds = partition(myds,n,ii)
        subds = copy(myds);
        subds.FileSet = partition(myds.FileSet,n,ii);
        reset(subds);
    end
end


methods (Hidden = true)

    % Define the progress method
    function frac = progress(myds)
        % Determine percentage of data read from datastore
        if hasdata(myds)
           frac = (myds.CurrentFileIndex-1)/...
                          myds.FileSet.NumFiles;
        else
           frac = 1;
        end
    end

    % Define the initializeDatastore method
    function initializeDatastore(myds,hadoopInfo)
        import matlab.io.datastore.DsFileSet;
        myds.FileSet = DsFileSet(hadoopInfo,...
            'FileSplitSize',myds.FileSet.FileSplitSize,...
            'IncludeSubfolders',true, ...
            'FileExtensions','.bin');
        reset(myds);
    end

    % Define the getLocation method
    function loc = getLocation(myds)
        loc = myds.FileSet;
    end

    % Define the isfullfile method
    function tf = isfullfile(~)
        tf = isequal(myds.FileSet.FileSplitSize,'file');
    end

end
```

```
    methods (Access = protected)
        % If you use the  FileSet property in the datastore,
        % then you must define the copyElement method. The
        % copyElement method allows methods such as readall
        % and preview to remain stateless
        function dscopy = copyElement(ds)
            dscopy = copyElement@matlab.mixin.Copyable(ds);
            dscopy.FileSet = copy(ds.FileSet);
        end

        % Define the maxpartitions method
        function n = maxpartitions(myds)
            n = maxpartitions(myds.FileSet);
        end
    end
end

%% STEP 3: IMPLEMENT YOUR CUSTOM FILE READING FUNCTION
function data = MyFileReader(fileInfoTbl)
% create a reader object using FileName
reader = matlab.io.datastore.DsFileReader(fileInfoTbl.FileName);

% seek to the offset
seek(reader,fileInfoTbl.Offset,'Origin','start-of-file');

% read fileInfoTbl.SplitSize amount of data
data = read(reader,fileInfoTbl.SplitSize);
end
```

This step completes the implementation of your custom datastore.

Next, create a datastore object using your custom datastore constructor. If your data is located at *hdfs:///path_to_files*, then you can use this code.

```
setenv('HADOOP_HOME','/path/to/hadoop/install');
ds = MyDatastoreHadoop('hdfs:///path_to_files');
```

To use `tall` arrays and the `gather` function on Apache Spark™ with parallel cluster configuration, set the `mapreducer` and attach `MyDatastoreHadoop.m` to the cluster.

```
mr = mapreducer(cluster);
mr.Cluster.AttachedFiles = 'MyDatastoreHadoop.m';
```

Create tall array from datastore.

```
t = tall(ds);
```

Gather the head of the tall array.

```
hd = gather(head(t));
```

See Also

mapreduce | matlab.io.Datastore | matlab.io.datastore.DsFileSet | matlab.io.datastore.Partitionable | tall

Topics

"Add Support for Hadoop"
"Use Tall Arrays on a Spark Enabled Hadoop Cluster" (Parallel Computing Toolbox)
"Big Data Workflow Using Tall Arrays and Datastores" (Parallel Computing Toolbox)

Introduced in R2017b

getLocation

Class: `matlab.io.datastore.HadoopFileBased`
Package: `matlab.io.datastore`

(Not recommended) Location of files in Hadoop

Note `matlab.io.datastore.HadoopFileBased.getLocation` is not recommended. Use `matlab.io.datastore.HadoopLocationBased.getLocation` instead.

Syntax

```
location = getLocation(ds)
```

Description

`location = getLocation(ds)` returns the location of the files in Hadoop for the datastore object `ds`.

Input Arguments

`ds` — Input datastore
`matlab.io.Datastore` object

Input datastore, specified as a `matlab.io.Datastore` object. To create a `Datastore` object, see `matlab.io.Datastore`.

Output Arguments

`location` — Location of files in Hadoop
cell array of character vectors | `matlab.io.datastore.DsFileSet` object

Location of files in Hadoop, returned as cell array of character vectors or a `matlab.io.datastore.DsFileSet` object. To create a `DsFileSet` object, see `matlab.io.datastore.DsFileSet`.

Attributes

`Abstract`	`true`
`Hidden`	`true`

To learn about attributes of methods, see Method Attributes.

See Also

`initializeDatastore` | `isfullfile` | `matlab.io.datastore.DsFileSet` | `matlab.io.datastore.HadoopFileBased`

Topics

"Add Support for Hadoop"

Introduced in R2017b

initializeDatastore

Class: `matlab.io.datastore.HadoopFileBased`
Package: `matlab.io.datastore`

(Not recommended) Initialize datastore with information from Hadoop

---

Note `matlab.io.datastore.HadoopFileBased.initializeDatastore` is not recommended. Use `matlab.io.datastore.HadoopLocationBased.initializeDatastore` instead.

Syntax

```
initializeDatastore(ds,info)
```

Description

`initializeDatastore(ds,info)` initializes the datastore `ds` with the information from Hadoop.

Input Arguments

`ds` — Input datastore
`matlab.io.Datastore` object

Input datastore, specified as a `matlab.io.Datastore` object. To create a `Datastore` object, see `matlab.io.Datastore`.

`info` — File information
structure

File information, specified as a structure. The structure contains the following fields: `FileName`, `Offset`, and `Size`. The `FileName` field is of the type `char`. The `Offset` and the `Size` fields are of the type `double`.

Attributes

`Abstract`	`true`
`Hidden`	`true`

To learn about attributes of methods, see Method Attributes.

See Also

`getLocation` | `isfullfile` | `matlab.io.datastore.HadoopFileBased`

Topics

"Add Support for Hadoop"

Introduced in R2017b

isfullfile

Class: `matlab.io.datastore.HadoopFileBased`
Package: `matlab.io.datastore`

(Not recommended) Check if datastore reads full files

Note `matlab.io.datastore.HadoopFileBased.isfullfile` is not recommended. Use `matlab.io.datastore.HadoopLocationBased.isfullfile` instead.

Syntax

```
tf = isfullfile(ds)
```

Description

`tf = isfullfile(ds)` returns logical `1` (`true`) if the datastore object specified by `ds` reads an entire file with each call to the `read` method. Otherwise, it returns logical `0` (`false`).

Input Arguments

ds — Input datastore
`matlab.io.Datastore` object

Input datastore, specified as a `matlab.io.Datastore` object. To create a `Datastore` object, see `matlab.io.Datastore`.

Attributes

Abstract	true
Hidden	true

To learn about attributes of methods, see Method Attributes.

See Also

`getLocation` | `initializeDatastore` | `matlab.io.datastore.HadoopFileBased`

Topics

"Add Support for Hadoop"

Introduced in R2017b

matlab.io.datastore.HadoopLocationBased class

Package: `matlab.io.datastore`

Add Hadoop support to datastore

Description

`matlab.io.datastore.HadoopLocationBased` is an abstract mixin class that adds Hadoop support for data stored in files, or other non-file-based data sources such as databases.

To use this mixin class, you must inherit from the `matlab.io.datastore.HadoopLocationBased` class in addition to inheriting from the `matlab.io.Datastore` base class. Type the following syntax as the first line of your class definition file:

```
classdef MyDatastore < matlab.io.Datastore & ...
                            matlab.io.datastore.HadoopLocationBased
    ...
end
```

To add Hadoop support along with parallel processing support, use these lines in your class definition file:

```
classdef MyDatastore < matlab.io.Datastore & ...
                            matlab.io.datastore.Partitionable & ...
                            matlab.io.datastore.HadoopLocationBased
    ...
end
```

To add support for Hadoop to your custom datastore, you must:

- Inherit from the additional class `matlab.io.datastore.HadoopLocationBased`
- Define these additional methods: `getLocation` and `initializeDatastore`.

For more details and steps to create your custom datastore with support for Hadoop, see “Develop Custom Datastore”.

Methods

getLocation	Location in Hadoop
initializeDatastore	Initialize datastore with information from Hadoop
isfullfile	Check if datastore reads full files

Attributes

`Sealed`	`false`

For information on class attributes, see “Class Attributes”.

Examples

Build Datastore with Hadoop Support

Implement a datastore with parallel processing and Hadoop support and use it to bring your data from the Hadoop server into MATLAB. Then use the `tall` and `gather` functions on this data.

Create a new `.m` class definition file that contains the code implementing your custom datastore. You must save this file in your working folder or in a folder that is on the MATLAB path. The name of the `.m` file must be the same as the name of your object constructor function. For example, if you want your constructor function to have the name `MyDatastoreHadoop`, then the name of the script file must be `MyDatastoreHadoop.m`. The `.m` class definition file must contain these steps:

- Step 1: Inherit from the datastore classes.
- Step 2: Define the constructor and the required methods.
- Step 3: Define your custom file reading function.

This code shows the three steps in a sample implementation of a custom datastore that can read binary files from a Hadoop server.

```
%% STEP 1: INHERIT FROM DATASTORE CLASSES
classdef MyDatastoreHadoop < matlab.io.Datastore & ...
        matlab.io.datastore.Partitionable & ...
        matlab.io.datastore.HadoopLocationBased
```

```
    properties (Access = private)
        CurrentFileIndex double
        FileSet matlab.io.datastore.DsFileSet
    end


%% STEP 2: DEFINE THE CONSTRUCTOR AND THE REQUIRED METHODS
    methods
        % Define your datastore constructor
        function myds = MyDatastoreHadoop(location,altRoots)
            myds.FileSet = matlab.io.datastore.DsFileSet(location,...
                'FileExtensions','.bin', ...
                'FileSplitSize',8*1024);
            myds.CurrentFileIndex = 1;

            if nargin == 2
                 myds.AlternateFileSystemRoots = altRoots;
            end

            reset(myds);
        end

        % Define the hasdata method
        function tf = hasdata(myds)
            % Return true if more data is available
            tf = hasfile(myds.FileSet);
        end

        % Define the read method
        function [data,info] = read(myds)
            % Read data and information about the extracted data
            % See also: MyFileReader()
            if ~hasdata(myds)
                error(sprintf(['No more data to read.\nUse the reset ',...
                     'method to reset the datastore to the start of ' ,...
                     'the data. \nBefore calling the read method, ',...
                     'check if data is available to read ',...
                     'by using the hasdata method.']))
            end

            fileInfoTbl = nextfile(myds.FileSet);
            data = MyFileReader(fileInfoTbl);
            info.Size = size(data);
            info.FileName = fileInfoTbl.FileName;
            info.Offset = fileInfoTbl.Offset;

            % Update CurrentFileIndex for tracking progress
            if fileInfoTbl.Offset + fileInfoTbl.SplitSize >= ...
                    fileInfoTbl.FileSize
                myds.CurrentFileIndex = myds.CurrentFileIndex + 1 ;
            end
        end

        % Define the reset method
        function reset(myds)
```

```
        % Reset to the start of the data
        reset(myds.FileSet);
        myds.CurrentFileIndex = 1;
    end


    % Define the partition method
    function subds = partition(myds,n,ii)
        subds = copy(myds);
        subds.FileSet = partition(myds.FileSet,n,ii);
        reset(subds);
    end
end


methods (Hidden = true)

    % Define the progress method
    function frac = progress(myds)
        % Determine percentage of data read from datastore
        if hasdata(myds)
           frac = (myds.CurrentFileIndex-1)/...
                          myds.FileSet.NumFiles;
        else
           frac = 1;
        end
    end

    % Define the initializeDatastore method
    function initializeDatastore(myds,hadoopInfo)
        import matlab.io.datastore.DsFileSet;
        myds.FileSet = DsFileSet(hadoopInfo,...
            'FileSplitSize',myds.FileSet.FileSplitSize,...
            'IncludeSubfolders',true, ...
            'FileExtensions','.bin');
        reset(myds);
    end

    % Define the getLocation method
    function loc = getLocation(myds)
        loc = myds.FileSet;
    end

    % Define the isfullfile method
    function tf = isfullfile(~)
        tf = isequal(myds.FileSet.FileSplitSize,'file');
    end

end

methods (Access = protected)
    % If you use the  FileSet property in the datastore,
    % then you must define the copyElement method. The
    % copyElement method allows methods such as readall
    % and preview to remain stateless
```

```
        function dscopy = copyElement(ds)
            dscopy = copyElement@matlab.mixin.Copyable(ds);
            dscopy.FileSet = copy(ds.FileSet);
        end

        % Define the maxpartitions method
        function n = maxpartitions(myds)
            n = maxpartitions(myds.FileSet);
        end
    end
end

%% STEP 3: IMPLEMENT YOUR CUSTOM FILE READING FUNCTION
function data = MyFileReader(fileInfoTbl)
% create a reader object using FileName
reader = matlab.io.datastore.DsFileReader(fileInfoTbl.FileName);

% seek to the offset
seek(reader,fileInfoTbl.Offset,'Origin','start-of-file');

% read fileInfoTbl.SplitSize amount of data
data = read(reader,fileInfoTbl.SplitSize);
end
```

This step completes the implementation of your custom datastore.

Next, create a datastore object using your custom datastore constructor. If your data is located at *`hdfs:///path_to_files`*, then you can use this code.

```
setenv('HADOOP_HOME','/path/to/hadoop/install');
ds = MyDatastoreHadoop('hdfs:///path_to_files');
```

To use `tall` arrays and the `gather` function on Apache Spark with parallel cluster configuration, set the `mapreducer` and attach `MyDatastoreHadoop.m` to the cluster.

```
mr = mapreducer(cluster);
mr.Cluster.AttachedFiles = 'MyDatastoreHadoop.m';
```

Create tall array from datastore.

```
t = tall(ds);
```

Gather the head of the tall array.

```
hd = gather(head(t));
```

See Also

mapreduce | matlab.io.Datastore | matlab.io.datastore.DsFileSet | matlab.io.datastore.Partitionable | tall

Topics

"Add Support for Hadoop"
"Use Tall Arrays on a Spark Enabled Hadoop Cluster" (Parallel Computing Toolbox)
"Big Data Workflow Using Tall Arrays and Datastores" (Parallel Computing Toolbox)

Introduced in R2019a

getLocation

Class: `matlab.io.datastore.HadoopLocationBased`
Package: `matlab.io.datastore`

Location in Hadoop

Syntax

```
location = getLocation(ds)
```

Description

`location = getLocation(ds)` returns the location of the data in Hadoop for the datastore object `ds`.

Input Arguments

`ds` — Input datastore
`matlab.io.Datastore` object

Input datastore, specified as a `matlab.io.Datastore` object. To create a `Datastore` object, see `matlab.io.Datastore`.

Output Arguments

`location` — Location of data in Hadoop
cell array of character vectors | `DsFileSet` object | `table`

Location of data in Hadoop, returned as a cell array of character vectors, `DsFileSet` object, or a `table`.

- Cell array of character vectors — List of files.

- `DsFileSet` object — To create a `DsFileSet` object, see `matlab.io.datastore.DsFileSet`.
- `table` — The table must contain a variable named `Hostname`. The `Hostname` variable must be one of these data types.
  - String array containing one hostname per row.
  - Cell array containing `0` or more hostnames per row.

Attributes

`Abstract`	`true`
`Hidden`	`true`

To learn about attributes of methods, see Method Attributes.

See Also

`initializeDatastore` | `isfullfile` | `matlab.io.datastore.DsFileSet` | `matlab.io.datastore.HadoopLocationBased`

Topics

"Add Support for Hadoop"

Introduced in R2019b

initializeDatastore

Class: `matlab.io.datastore.HadoopLocationBased`
Package: `matlab.io.datastore`

Initialize datastore with information from Hadoop

Syntax

```
initializeDatastore(ds,info)
```

Description

`initializeDatastore(ds,info)` initializes the datastore `ds` with the information from Hadoop.

Input Arguments

`ds` — Input datastore
`matlab.io.Datastore` object

Input datastore, specified as a `matlab.io.Datastore` object. To create a `Datastore` object, see `matlab.io.Datastore`.

`info` — Hadoop data information
table

Hadoop data information, specified as a table.

- When data is file-based, `info` is a single row table containing the following variables: `FileName`, `Offset`, and `Size`. The `FileName` variable is of the type `string`. The `Offset` and the `Size` variables are of the type `double`.
- When data is non-file-based, `info` is one row of the output from the `getLocation` method.

Attributes

`Abstract`	`true`
`Hidden`	`true`

To learn about attributes of methods, see Method Attributes.

See Also

`getLocation` | `isfullfile` | `matlab.io.datastore.HadoopLocationBased`

Topics

"Add Support for Hadoop"

Introduced in R2019a

isfullfile

Class: `matlab.io.datastore.HadoopLocationBased`
Package: `matlab.io.datastore`

Check if datastore reads full files

Syntax

```
tf = isfullfile(ds)
```

Description

`tf = isfullfile(ds)` returns `true` if the datastore object specified by `ds` reads an entire file with each call to the `read` method. Otherwise, it returns `false`.

Input Arguments

ds — Input datastore
`matlab.io.Datastore` object

Input datastore, specified as a `matlab.io.Datastore` object. To create a `Datastore` object, see `matlab.io.Datastore`.

Attributes

Abstract	true
Hidden	true

To learn about attributes of methods, see Method Attributes.

Tips

- `isfullfile` is not required for non-file-based data.

See Also

`getLocation` | `initializeDatastore` | `matlab.io.datastore.HadoopLocationBased`

Topics

"Add Support for Hadoop"

Introduced in R2019a

handle class

Superclass of all handle classes

Description

The `handle` class is the superclass for all classes that follow handle semantics. A handle is a variable that refers to an object of a handle class. Multiple variables can refer to the same object.

The `handle` class is an abstract class, so you cannot create an instance of this class directly. You use the `handle` class to derive other classes, which can be concrete classes whose instances are handle objects.

To define a handle class, derive your class from `handle` using the syntax in the following `classdef` code.

```
classdef MyHandleClass < handle
    ...
end
```

Deriving from the `handle` class enables your subclass to:

- Inherit several useful methods (“Handle Class Methods”)
- Define events and listeners (“Events and Listeners Syntax”)
- Define dynamic properties (“Dynamic Properties — Adding Properties to an Instance”)
- Implement set and get methods (“Implement Set/Get Interface for Properties”)
- Customize copy behavior (“Implement Copy for Handle Classes”)

For information about using handle classes, see “Handle Classes”

Class Attributes

`Abstract`	`true`
`ConstructOnLoad`	`true`
`HandleCompatible`	`true`

For information on class attributes, see “Class Attributes”.

Methods

Public Methods

addlistener	Create event listener bound to event source
listener	Create event listener without binding to event source
notify	Notify listeners that event is occurring
findobj	Find handle objects
findprop	Find meta.property object
delete	Delete handle object
isvalid	Determine valid handles
relationaloperators	Determine equality or sort handle objects
empty	Create empty array of specified class

Events

Event Name	Trigger	Event Data	Event Attributes

`ObjectBeingDestroyed`	Triggered when the handle object is about to be destroyed, but before calling the `delete` method.	`event.EventData`	`NotifyAccess: private`
			
			`ListenAccess: public`

Examples

Derive Class from `handle`

The `MySubclass` class derives from the `handle` class. The property set method does not return the object passed to the method, as is required by a value class:

```
classdef MySubclass < handle
   properties
      Client tcpclient
```

```
    end
    methods
       function set.Client(obj,c)
          if isa(c,'tcpclient')
             obj.Client = c;
          end
       end
    end
end
```

Create an object of `MySubclass` and assign a `tcpclient` object to the `Client` property.

```
t = MySubclass;
t.Client = tcpclient('www.mathworks.com', 80);
```

More About

Handle Object Copy Behavior

Copying a handle object does not copy the underlying data associated with the object. The copy is another handle referring to the same object. Therefore, if a function modifies a handle object passed as an input argument, the modification affects the original input object in the caller's workspace.

In contrast, nonhandle objects (that is, value objects) associate data with a particular variable. Functions must return modified value objects to change the object outside of the function's workspace.

For information on passing objects to functions, see “Object Modification” and “Handle Object Behavior”.

Destroying Handle Objects

MATLAB destroys handle objects when there are no references to the object. You can explicitly remove a handle object by calling its `delete` method. The `handle` class enables you to control what happens when handle objects are destroyed, either implicitly when no references exist or explicitly when you delete the object.

For more information, see “Handle Class Destructor”.

Create Listener for `ObjectBeingDestroyed` Event

Any code can respond to the pending deletion of a handle object by defining a listener for that object's `ObjectBeingDestroyed` event. MATLAB triggers this event before calling the object's `delete` method.

For more information on using events and listeners, see "Events and Listeners Syntax".

See Also

Topics

"Handle Classes"
"Events"
"Handle Object Behavior"

Introduced in R2008a

addlistener

Class: `handle`

Create event listener bound to event source

Syntax

```
el = addlistener(hSource,EventName,callback)
el = addlistener(hSource,PropertyName,EventName,callback)
```

Description

`el = addlistener(hSource,EventName,callback)` creates a listener for the event `EventName` when triggered on the source object, `hSource`.

If `hSource` is an array, the listener responds to the named event on any object in the `hSource` array. `callback` is a function handle referencing the callback function.

`el = addlistener(hSource,PropertyName,EventName,callback)` creates a listener for one of the predefined property events. There are four predefined property events:

Event Name	Event Occurs
`PreSet`	Immediately before the property value is set, before calling its set access method
`PostSet`	Immediately after the property value is set
`PreGet`	Immediately before a property value query is serviced, before calling its get access method
`PostGet`	Immediately after returning the property value to the query

Input Arguments

`hSource` — Event source
handle array

Event source is the object that is source of the event, or an array of source objects, specified as a handle array.

`EventName` — Name of event
character vector | string scalar

Name of event that is triggered on the source objects, specified as case-sensitive, quoted text. For property events, the event name is one of the four predefined property events.

Data Types: `char` | `string`

`PropertyName` — Name of property
character vector or `meta.property`

Name of the property whose property event triggers your listener, specified as one of these values:

- A character vector or a cell array of character vectors, where each character vector is the name of a property defined for the objects in `hSource`
- A scalar `meta.property` object
- An array of `meta.property` objects

You can attach listeners to property events on dynamic properties only when `hSource` is scalar. If `hSource` is non-scalar, then the properties must belong to the class of `hSource` and cannot include dynamic properties (which are not part of the class definition).

The class defining the source property must set the `GetObservable` and `SetObservable` property attributes to enable you to listen to the property events.

`callback` — Listener callback
function handle

Listener callback specified as a function handle

Data Types: `function_handle`

Output Arguments

el — Listener object
`event.listener` or `event.proplistener`

Listener object created by `addlistener`, specified as the handle to an `event.listener` or an `event.proplistener` object.

Examples

Create Listener

Create a property listener for the `Color` property of a graphics `figure` window.

```
fig = figure;
propListener = addlistener(fig,'Color','PostSet',@(src,evnt)disp('Color changed'));
```

Set the value of the `Color` property to `yellow`. Setting the property triggers the `PostSet` property event on the figure. The event source object is the specific figure referenced by the handle `fig`.

```
set(fig,'Color','yellow')
```

Tips

- To remove a listener, delete the listener object returned by `addlistener`. For example,

  ```
  delete(el)
  ```

  calls the handle class delete method to delete the object from the workspace and remove the listener.
- Redefining or clearing the variable containing the handle of the listener (for example, `el`) does not delete the listener. The event object (`hSource`) still has a reference to the `event.listener` object. `addlistener` ties the listener's lifecycle to the object that is the source of the event.
- To define a listener that is not tied to the event object, use the `event.listener` constructor directly to create the listener.

Alternatives

When you need the lifecycle of the listener object to be independent of the source object lifecycle, use `handle.listener` to create listeners.

See Also

`event.listener` | `handle.listener` | `handle.notify`

Topics

"Events"
"Overview Events and Listeners"
"Create Property Listeners"
"Listener Lifecycle"
"Restore Listeners"

Introduced in R2008a

listener

Class: `handle`

Create event listener without binding to event source

Syntax

```
eL = listener(hSource,EventName,callback)
eL = listener(hSource,PropertyName,EventName,callback)
```

Description

`eL = listener(hSource,EventName,callback)` creates a listener for the event named `EventName`. `hSource` is a handle object that is the source of the event. `callback` is a function handle that MATLAB invokes when the event is triggered.

If `hSource` is an array of event source objects, the listener responds to the named event on any object in the array that is not in a deleted state.

`eL = listener(hSource,PropertyName,EventName,callback)` creates a listener for one of the predefined property events. There are four predefined property events:

Event Name	Event Occurs
`PreSet`	Immediately before the property value is set, before calling its set access method
`PostSet`	Immediately after the property value is set
`PreGet`	Immediately before a property value query is serviced, before calling its get access method
`PostGet`	Immediately after returning the property value to the query

Input Arguments

`hSource` — Handle object that is the source of the event
handle array

Handle object that is the source of the event, specified as a single object or an array of objects.

`EventName` — Name of the event that is triggered on the source objects
character vector or string scalar

Name of the event that is triggered on the source objects, specified as case-sensitive, quoted text. For property events, the event name is one of the four predefined property events.

Data Types: `char` | `string`

`PropertyName` — Name of property
character vector or `meta.property` object.

Name of the property whose property event triggers the listener, specified as one of these values:

- A character vector or a cell array of character vectors, where each character vector is the name of a property defined for the objects in `hSource`
- A scalar `meta.property` object or an array of `meta.property` objects corresponding to properties defined for the objects in `hSource`

You can attach listeners to the property events of dynamic properties only when `hSource` is scalar. If `hSource` is non-scalar, then the properties must belong to the class of `hSource` and cannot include dynamic properties (which are not part of the class definition).

The class defining the source property must set the `GetObservable` and `SetObservable` property attributes to enable you to listen to the property events.

`callback` — Listener callback
function handle

Listener callback specified as a function handle

Data Types: `function_handle`

Output Arguments

el — Listener object
`event.listener` or `event.proplistener`

Listener object, returned as the handle to an `event.listener` or an `event.proplistener` object.

Examples

Create Listener

Create a property listener for the `Color` property of a graphics `figure` window.

```
fig = figure;
propListener = listener(fig,'Color','PostSet',@(src,evnt)disp('Color changed'));
```

Set the value of the `Color` property to `yellow`. Setting the property triggers the `PostSet` property event on the figure. The event source object is the specific figure referenced by the handle `fig`.

```
set(fig,'Color','yellow')
```

Delete the listener object.

```
delete(propListener)
```

Tips

Listener Lifecycle

To remove a listener, delete the listener object returned by `listener`. For example, this statement calls the handle class `handle.delete` method to remove the listener.

```
delete(el)
```

Calling delete on the listener object destroys the listener and, therefore, the event no longer causes the callback function to execute.

The `listener` method does not bind the listener's lifecycle to the object that is the source of the event. Destroying the event source object does not affect the lifecycle of the listener object.

You must explicitly destroy listeners created with the `listener` method independently of the source object. Calling the handle delete method on the listener variable (for example, `delete(el)`) explicitly destroys the listener. Redefining or clearing the variable containing the listener can delete the listener if there are no other references to it. To bind the lifecycle of the listener to the lifecycle of the event source object, use `handle.addlistener`.

Alternatives

Use `addlistener` when you want MATLABto manage the listener lifecycle.

See Also

`event.listener` | `handle.notify`

Topics

"Events"
"Overview Events and Listeners"
"Create Property Listeners"
"Listener Lifecycle"
"Restore Listeners"

Introduced in R2017b

delete

Class: `handle`

Delete handle object

Syntax

```
delete(H)
```

Description

`delete(H)` deletes the handle object. After calling `delete`, `H` is an invalid object (the object is destroyed). If `H` is an array of handle objects, then MATLAB calls the `delete` method on each object in the array.

Subclasses of the `handle` class can implement a method named `delete` to perform cleanup tasks just before destroying objects of the class. MATLAB calls the `delete` method of any `handle` object automatically when the object is destroyed. The subclass `delete` method must meet certain criteria to be called by MATLAB during object destruction. For information on implementing a `delete` method for a subclass of `handle`, see “Handle Class Destructor”.

Input Arguments

`H` — Handle object
handle object or array of handle objects

Handle object to delete, specified as the handle variable.

Attributes

Access	public

To learn about attributes of methods, see Method Attributes.

See Also

`isvalid`

Topics

“Handle Class Destructor”
“Handle Object Behavior”

findobj

Class: `handle`

Find handle objects

Syntax

```
Hmatch = findobj(H)
Hmatch = findobj(H,property,value,...,property,value)
Hmatch = findobj(H,'-not',property,value)
Hmatch = findobj(H,'-regexp',property,value)
Hmatch = findobj(H,property,value,'-logicaloperator',property,value)
Hmatch = findobj(H,'-function',fh)
Hmatch = findobj(H,'-function',property,fh)
Hmatch = findobj(H,'-class',class)
Hmatch = findobj(H,'-isa',class)
Hmatch = findobj(H,'-property',property)
Hmatch = findobj(H,'-method',methodname)
Hmatch = findobj(H,'-event',eventname)
Hmatch = findobj(H,'-depth',d, ___ )
```

Description

`Hmatch = findobj(H)` returns the objects listed in `H` and all of their descendants.

`Hmatch = findobj(H,property,value,...,property,value)` finds `handle` objects that have the specified property set to the specified value.

`Hmatch = findobj(H,'-not',property,value)` inverts the expression in the following `property` `value` pair. That is, find objects whose specified property is not equal to `value`.

`Hmatch = findobj(H,'-regexp',property,value)` treats the content of the `value` argument as regular expressions.

`Hmatch = findobj(H,property,value,'-logicaloperator',property,value)` applies the logical operator to the name/value pairs. Supported logical operators include:

- `'-or'` (default if you do not specify an operator)
- `'-and'`
- `'-xor'`

`Hmatch = findobj(H,'-function',fh)` calls the function handle `fh` on the objects in `H` and returns the objects for which the function returns `true`.

`Hmatch = findobj(H,'-function',property,fh)` calls the function handle `fh` on the specified property's value for the objects in `H` and returns the objects for which the function returns `true`. The function must return a scalar logical value.

`Hmatch = findobj(H,'-class',class)` finds all objects belonging to the specified class.

`Hmatch = findobj(H,'-isa',class)` finds all objects belonging to the specified class.

`Hmatch = findobj(H,'-property',property)` finds all object in `H` having the named property.

`Hmatch = findobj(H,'-method',methodname)` finds objects that have the specified method name.

`Hmatch = findobj(H,'-event',eventname)` finds objects that have the specified event name.

`Hmatch = findobj(H,'-depth',d, ___ )` specifies how many levels in the instance hierarchies under the objects in `H` to search.

Specify all `-options` as character vectors or string scalars.

Input Arguments

`H` — Objects to search from
handle array

Objects to search from, specified as an array of object handles. Unless the you specify the `'-depth'` option, `findobj` searches the objects in the input array `H` and child objects in the instance hierarchy.

`property` — Property name
character vector | string scalar

Property name, specified as case-sensitive, quoted text.

Data Types: `char` | `string`

`class` — Class of object to find
character vector | string scalar

Class of object to find, specified as case-sensitive, quoted text.

Data Types: `char` | `string`

`value` — Property value
any value

Property value, specified as a value or MATLAB expression.

`methodname` — Method name
character vector | string scalar

Method name, specified as case-sensitive quoted text.

Data Types: `char` | `string`

`eventname` — Event name
character vector | string scalar

Event name, specified as case-sensitive quoted text.

Data Types: `char` | `string`

`d` — Depth of search
integer >= 0

Depth of search, specified as an integer indicating the number of levels below any given object in the input array `H`.

- `d = n` — Search `n` levels of the hierarchy below each object in `H`

- `d = 0` — Search only the same level as the objects in `H`.
- `d = inf` — Search all levels below objects in `H`. This is the default.

`fh` — Function handle
function handle

Function handle, specifying the function that is evaluated for each object in the input array `H`. This function must return a scalar, logical value indicating whether there is a match (`true`) or not (`false`).

Output Arguments

`Hmatch` — Objects found by search
handle array

Objects found by search, returned as a handle array.

Attributes

Access	public

To learn about attributes of methods, see Method Attributes.

Examples

Object with Specific Property Value

Find the object with a specific property value. Given the `handle` class, `BasicHandle`:

```
classdef BasicHandle < handle
   properties
      Prop1
   end
   methods
      function obj = BasicHandle(val)
         if nargin > 0
            obj.Prop1 = val;
```

```
         end
      end
   end
end
```

Create an array of `BasicHandle` objects:

```
h(1) = BasicHandle(7);
h(2) = BasicHandle(11);
h(3) = BasicHandle(27);
```

Find the handle of the object whose `Prop1` property has a value of 7:

```
h7 = findobj(h,'Prop1',7);
h7.Prop1
```

```
ans =

     7
```

Object with Specific Property Name

Find the object with a specific dynamic property. Given the `button` class:

```
classdef button < dynamicprops
   properties
      UiHandle
   end
   methods
      function obj = button(pos)
         if nargin > 0
            if length(pos) == 4
               obj.UiHandle = uicontrol('Position',pos,...
                'Style','pushbutton');
            else
               error('Improper position')
            end
         end
      end
   end
end
```

Create an array of button objects, only one element of which defines a dynamic property. Use `findobj` to get the handle of the object with the dynamic property named `ButtonCoord`:

```
b(1) = button([20 40 80 20]);
addprop(b(1),'ButtonCoord');
b(1).ButtonCoord = [2,3];
b(2) = button([120 40 80 20]);
b(3) = button([220 40 80 20]);

h =  findobj(b,'-property','ButtonCoord');
h.ButtonCoord
```

```
ans =

     2     3
```

Tips

- `findobj` has access only to public members.
- If there are no matches, `findobj` returns an empty array of the same class as the input array `H`.
- Logical operator precedence follows MATLAB precedence rules. For more information, see “Operator Precedence”.
- Control precedence by grouping within cell arrays

See Also

`handle.findprop`

Introduced in R2008a

findprop

Class: `handle`

Find `meta.property` object

Syntax

```
mp = findprop(h,property)
```

Description

`mp = findprop(h,property)` returns the `meta.property` object associated with the named property of the object `h`. `property` can be a property defined by the class of `h` or a dynamic property defined only for the object `h`.

Input Arguments

`h` — handle object
handle

Handle object, specified as a scalar handle.

`property` — Name of property
character vector | string scalar

Name of property, specified as case-sensitive, quoted text.

Data Types: `char` | `string`

Output Arguments

`mp` — `meta.property` object
`meta.property`

`meta.property` object that is associated with the named property. If `findprop` does not find the property on the object `h`, `findprop` returns a 0-by-1 empty `meta.property` object.

Attributes

`Access`	`public`

To learn about attributes of methods, see Method Attributes.

Examples

Display Property Attributes

Display the current attribute values of the containers.Map Count property.

```
mp = findprop(containers.Map,'Count');
disp(mp)
```

```
  property with properties:

                  Name: 'Count'
           Description: 'Number of pairs in the collection'
    DetailedDescription: ''
             GetAccess: 'public'
             SetAccess: 'private'
             Dependent: 1
              Constant: 0
              Abstract: 0
             Transient: 1
                Hidden: 0
         GetObservable: 0
         SetObservable: 0
              AbortSet: 0
           NonCopyable: 1
             GetMethod: []
             SetMethod: []
            HasDefault: 0
            Validation: [0x0 meta.Validation]
         DefiningClass: [1x1 meta.class]
```

See Also

`handle.findobj` | `meta.property`

Topics

"Get Information About Properties"
"Dynamic Properties — Adding Properties to an Instance"

isvalid

Class: `handle`

Determine valid handles

Syntax

```
B = isvalid(H)
```

Description

`B = isvalid(H)` returns a logical array in which each element is `true` if the corresponding element in `H` is a valid handle. A handle variable becomes invalid if the object has been deleted. Within a class `delete` method, `isvalid` always returns `false`.

You cannot override the `isvalid` method in `handle` subclasses.

Input Arguments

`H` — Input array
handle array

Input array, specified as an array of object handles.

Output Arguments

`B` — Result of validity test
logical

Result of validity test, returned as a logical array the same size as `H` in which each element is `true` if the corresponding element in `H` is a valid handle.

Attributes

Access	public
Sealed	true

To learn about attributes of methods, see Method Attributes.

Examples

Test for Valid Handles

This example tests a handle array for valid members:

```
H = plot(rand(5));
delete(H(3:4))
B = isvalid(H)
```

```
B =

     1
     1
     0
     0
     1
```

See Also

handle.delete | isgraphics

Topics

"Test Handle Validity"
"Object Lifecycle"
"Handle Object During delete Method Execution"

notify

Class: `handle`

Notify listeners that event is occurring

Syntax

```
notify(H,EventName)
notify(H,EventName,data)
```

Description

`notify(H,EventName)` notifies listeners that the named event is taking place on the handle objects in `H`.

`notify(H,EventName,data)` includes user-defined event data.

Input Arguments

`H` — Event source
handle array

Event source, specified as a handle array. All of the objects in `H` must define the named event.

`EventName` — Name of event
character vector | string scalar

Name of event, specified as case-sensitive, quoted text that is defined by the class of `H`.

`data` — User-defined event data
subclass of `event.EventData`

User-defined event data, specified as an object that is a subclass of the `event.EventData` class. For information on defining event data, see “Define Event-Specific Data”.

Note Listener callbacks should not reuse the same event data object in subsequent calls to `notify`. Always create a new event data object to pass to `notify`.

Attributes

`Access`	`public`

To learn about attributes of methods, see Method Attributes.

See Also

`handle.addlistener` | `handle.listener`

Topics

“Events and Listeners Syntax”
“Events”

Introduced in R2008a

relationaloperators

Class: `handle`

Determine equality or sort handle objects

Syntax

```
tf = eq(H1,H2)
tf = ne(H1,H2)
tf = lt(H1,H2)
tf = le(H1,H2)
tf = gt(H1,H2)
tf = ge(H1,H2)
```

Description

`tf = eq(H1,H2)` Equal. (`H1 == H2`)

`tf = ne(H1,H2)` Not equal. (`H1 ~= H2`)

`tf = lt(H1,H2)` Less than. (`H1 < H2`)

`tf = le(H1,H2)` Less than or equal. (`H1 <= H2`)

`tf = gt(H1,H2)` Greater than. (`H1 > H2`)

`tf = ge(H1,H2)` Greater than or equal. (`H1 >= H2`)

For each pair of input arrays (`H1` and `H2`), the operation returns a logical array of the same size. Each element in the returned array is an element-wise equality or comparison test result. These methods perform scalar expansion in the same way as the MATLAB built-in relational operators. For general information on relational operators, see “Relational Operations”.

The following guidelines apply to handle comparison:

- Copies of a handle variable always compare as equal.
- The repeated comparison of any two handles always yields the same result in the same MATLAB session.
- Different handles are always not equal.
- The order of handle values is purely arbitrary and has no connection to the state of the handle objects being compared.
- If the input arrays belong to different classes (including the case where one input array belongs to a non-handle class such as `double`) then the comparison is always false.
- If you make a comparison between a handle object and an object of a dominant class, the method of the dominant class is invoked. You should generally test only like objects because a dominant class might not define one of these methods.
- An error occurs if the input arrays are not the same size and neither is scalar.

Use `isequal` when you want to determine if different handle objects have the same data in all object properties. Use == when you want to determine if handle variables refer to the same object.

When comparing objects that contain dynamic properties, `isequal` always returns `false`.

Input Arguments

`H1` — Left argument to operator
handle array

Left argument to operator, specified as a handle array.

`H2` — Right argument to operator
handle array

Right argument to operator, specified as a handle array.

Output Arguments

`tf` — Result of comparison
logical array

Result of comparison, returned as a logical array of the same size as the input arrays, where each element is an element-wise equality or comparison test result

- `1` — relationship is true
- `0` — relationship is not true

Attributes

`Access`	`public`

To learn about attributes of methods, see Method Attributes.

See Also

`isequal`

Topics

"Determining Equality of Objects"

hankel

Hankel matrix

Syntax

```
H = hankel(c)
H = hankel(c,r)
```

Description

`H = hankel(c)` returns the square Hankel matrix whose first column is `c` and whose elements are zero below the first anti-diagonal.

`H = hankel(c,r)` returns a Hankel matrix whose first column is `c` and whose last row is `r`. If the last element of `c` differs from the first element of `r`, the last element of `c` prevails.

Examples

Compute Hankel Matrix

Create the Hankel matrix whose first column is `c` and whose last row is `r`.

```
c = 1:3;
r = 3:6;
h = hankel(c,r)
```

```
h = 3×4

     1     2     3     4
     2     3     4     5
     3     4     5     6
```

More About

Hankel Matrix

A Hankel matrix is a matrix that is symmetric and constant across the anti-diagonals, and has elements `h(i,j) = p(i+j-1)`, where vector `p = [c r(2:end)]` completely determines the Hankel matrix.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`hadamard` | `kron` | `toeplitz`

Introduced before R2006a

hasFactoryValue

Package: `matlab.settings`

Determine whether the setting has a factory value set

Syntax

```
hasFactoryValue(s)
```

Description

`hasFactoryValue(s)` returns `1` if the setting has a factory value set. Otherwise, `hasFactoryValue` returns `0`.

Examples

Determine if Setting Has Factory Value

Check whether the setting for the maximum column width for comments in MATLAB has a factory value.

```
s = settings;
hasFactoryValue(s.matlab.editor.language.matlab.comments.MaxWidth)
```

```
ans =
  logical
   1
```

Input Arguments

s — Setting
`setting` object

Setting, specified as a `setting` object. Use the `settings` function to access the root settings group object and all the available settings in the tree.

See Also

`hasPersonalValue` | `hasTemporaryValue` | `hasTemporaryValue` | `settings`

Topics

"Access and Modify Settings"

Introduced in R2018a

hasGroup

Package: `matlab.settings`

Determine if settings group exists

Syntax

`hasGroup(parentgroup,name)`

Description

`hasGroup(parentgroup,name)` returns `1` if the settings group `parentgroup` contains a settings group called `name`. Otherwise, `hasGroup` returns `0`.

Examples

Check Before Adding Settings Group

Check whether a settings group exists before adding it.

Use the `settings` function to access the root of the settings tree and then create the settings group `mysettings`.

```
s = settings;
addGroup(s,'mysettings');
```

Check whether a settings group called `myothersettings` exists before adding it to `mysettings`.

```
if hasGroup(s.mysettings,'myothersettings') == 0
    addGroup(s.mysettings,'myothersettings');
end
s.mysettings
```

```
ans =
  SettingsGroup 'mysettings' with properties:
      myothersettings: [1×1 SettingsGroup]
```

Input Arguments

parentgroup — Parent settings group
SettingsGroup object

Parent settings group, specified as a SettingsGroup object. Use the settings function to access the root settings group object and all the available settings groups.

name — Name of settings group to check for
character vector | string scalar

Name of the settings group to check for, specified as a character vector or string scalar.

See Also

addGroup | hasSetting | removeGroup | settings

Topics

"Access and Modify Settings"
"Create Custom Settings"

Introduced in R2019b

hasnext

Package: `matlab.mapreduce`

Determine if ValueIterator has one or more values available

Syntax

```
tf = hasnext(ValIter)
```

Description

`tf = hasnext(ValIter)` returns logical `1` (`true`) if `ValueIter` has one or more values available; otherwise, it returns logical `0` (`false`).

Examples

Get Values from ValueIterator in Reduce Function

Use the `hasnext` and `getnext` functions in a `while` loop within the reduce function to iteratively get values from the `ValueIterator` object. For example,

```
function MeanDistReduceFun(sumLenKey, sumLenIter, outKVStore)
    sumLen = [0 0];
    while hasnext(sumLenIter)
        sumLen = sumLen + getnext(sumLenIter);
    end
    add(outKVStore, 'Mean', sumLen(1)/sumLen(2));
end
```

Always call `hasnext` before `getnext` to confirm availability of a value. `mapreduce` returns an error if you call `getnext` with no remaining values in the `ValueIterator` object.

Input Arguments

`ValIter` — Intermediate value iterator
`ValueIterator` object

Intermediate value iterator, specified as a `ValueIterator` object. The `mapreduce` function automatically creates this object during execution. The second input to the reduce function specifies the variable name for the `ValueIterator` object, which is the variable name to use with the `hasnext` and `getnext` functions.

For more information, see `ValueIterator`.

See Also

`getnext` | `mapreduce`

Topics

`ValueIterator`
"Build Effective Algorithms with MapReduce"

Introduced in R2014b

hasPersonalValue

Package: `matlab.settings`

Determine whether the setting has a personal value set

Syntax

```
hasPersonalValue(s)
```

Description

`hasPersonalValue(s)` returns `1` if the setting has a personal value set. Otherwise, `hasPersonalValue` returns `0`.

Examples

Clear Setting Personal Value

Check whether the setting for the maximum column width for comments in MATLAB has a personal value. If the setting has a personal value, clear the value.

```
s = settings;

if(hasPersonalValue(s.matlab.editor.language.matlab.comments.MaxWidth))
    clearPersonalValue(s.matlab.editor.language.matlab.comments.MaxWidth)
end
```

Input Arguments

s — Setting
`setting` object

Setting, specified as a `setting` object. Use the `settings` function to access the root settings group object and all the available settings in the tree.

See Also

`addSetting` | `hasFactoryValue` | `hasTemporaryValue` | `settings`

Topics

"Access and Modify Settings"

Introduced in R2018a

hasSetting

Package: `matlab.settings`

Determine if setting exists in settings group

Syntax

`hasSetting(parentgroup,name)`

Description

`hasSetting(parentgroup,name)` returns `1` if the specified settings group contains a setting called `name`. Otherwise, `hasSetting` returns `0`.

Examples

Check Before Adding Setting

Check whether a setting exists before adding it.

Use the `settings` function to access the root of the settings tree and then create the settings group `mysettings`.

```
s = settings;
addGroup(s,'mysettings');
```

Check whether the `MyWorkAddress` setting exists before adding it to `mysettings`.

```
if hasSetting(s.mysettings,'MyWorkAddress') == 0
    addSetting(s.mysettings,'MyWorkAddress','PersonalValue','3 Apple Hill Drive');
end
s.mysettings
```

```
ans =
  SettingsGroup 'mysettings' with properties:
        MyWorkAddress: [1×1 Setting]
```

Input Arguments

parentgroup — Parent settings group
SettingsGroup object

Parent settings group, specified as a SettingsGroup object. Use the settings function to access the root settings group object and all the available settings groups in the tree.

name — Name of setting to check for
character vector | string scalar

Name of the setting to check for, specified as a character vector or string scalar.

See Also

addSetting | hasGroup | removeSetting | settings

Topics

"Access and Modify Settings"
"Create Custom Settings"

Introduced in R2019b

hasTemporaryValue

Package: `matlab.settings`

Determine whether the setting has a temporary value set

Syntax

`hasTemporaryValue(s)`

Description

`hasTemporaryValue(s)` returns `1` if the setting has a temporary value set. Otherwise, `hasTemporaryValue` returns `0`.

Examples

Clear Setting Temporary Value

Check whether the setting for the maximum column width for comments in MATLAB has a temporary value. If the setting has a temporary value, clear the value.

```
s = settings;

if(hasTemporaryValue(s.matlab.editor.language.matlab.comments.MaxWidth))
    clearTemporaryValue(s.matlab.editor.language.matlab.comments.MaxWidth)
end
```

Input Arguments

s — Setting
`setting` object

Setting, specified as a `setting` object. Use the `settings` function to access the root settings group object and all the available settings in the tree.

See Also

`addSetting` | `hasFactoryValue` | `hasPersonalValue` | `settings`

Topics

"Access and Modify Settings"

Introduced in R2018a

hdf5info

Information about HDF5 file

Note `hdf5info` will be removed in a future version. Use `h5info` instead.

Syntax

```
fileinfo = hdf5info(filename)
fileinfo = hdf5info(...,'ReadAttributes',BOOL)
[...] = hdf5info(..., 'V71Dimensions', BOOL)
```

Description

`fileinfo = hdf5info(filename)` returns a structure `fileinfo` whose fields contain information about the contents of the HDF5 file `filename`. `filename` is a character vector or string scalar that specifies the name of the HDF5 file.

`fileinfo = hdf5info(...,'ReadAttributes',BOOL)` specifies whether `hdf5info` returns the values of the attributes or just information describing the attributes. By default, `hdf5info` reads in attribute values (`BOOL = true`).

`[...] = hdf5info(..., 'V71Dimensions', BOOL)` specifies whether to report the dimensions of data sets and attributes as they were returned in previous versions of `hdf5info` (MATLAB 7.1 [R14SP3] and earlier). If `BOOL` is true, `hdf5info` swaps the first two dimensions of the data set. This behavior was intended to account for the difference in how HDF5 and MATLAB express array dimensions. HDF5 describes data set dimensions in row-major order; MATLAB stores data in column-major order. However, swapping these dimensions may not correctly reflect the intent of the data in the file and may invalidate metadata. When `BOOL` is false (the default), `hdf5info` returns data dimensions that correctly reflect the data ordering as it is written in the file—each dimension in the output variable matches the same dimension in the file.

Note If you use the `'V71Dimensions'` parameter and intend on passing the `fileinfo` structure returned to the `hdf5read` function, you should also specify the `'V71Dimensions'` parameters with `hdf5read`. If you do not, `hdf5read` uses the new behavior when reading the data set and certain metadata returned by `hdf5info` does not match the actual data returned by `hdf5read`.

Examples

```
fileinfo = hdf5info('example.h5');
```

To get more information about the contents of the HDF5 file, look at the `GroupHierarchy` field in the `fileinfo` structure returned by `hdf5info`.

```
toplevel = fileinfo.GroupHierarchy

toplevel =

      Filename: [1x64 char]
          Name: '/'
        Groups: [1x2 struct]
      Datasets: []
     Datatypes: []
         Links: []
    Attributes: [1x2 struct]
```

To probe further into the file hierarchy, keep examining the `Groups` field.

See also

`hdf5read`, `hdf5writeh5info`

hdf5read

Read HDF5 file

Note `hdf5read` will be removed in a future version. Use `h5read` instead.

Syntax

```
data = hdf5read(filename,datasetname)
attr = hdf5read(filename,attributename)
[data, attr] = hdf5read(...,'ReadAttributes',BOOL)
data = hdf5read(hinfo)
[...] = hdf5read(..., 'V71Dimensions', BOOL)
```

Description

`data = hdf5read(filename,datasetname)` reads all the data in the data set `datasetname` that is stored in the HDF5 file `filename` and returns it in the variable `data`. To determine the names of data sets in an HDF5 file, use the `hdf5info` function.

The return value, `data`, is a multidimensional array. `hdf5read` maps HDF5 data types to native MATLAB data types, whenever possible. If it cannot represent the data using MATLAB data types, `hdf5read` uses one of the HDF5 data type objects. For example, if an HDF5 file contains a data set made up of an enumerated data type, `hdf5read` uses the `hdf5.h5enum` object to represent the data in the MATLAB workspace. The `hdf5.h5enum` object has data members that store the enumerations (names), their corresponding values, and the enumerated data.

Note `hdf5read` performs best when reading numeric datasets. If you need to read string, compound, or variable length datasets, MathWorks strongly recommends that you use the low-level HDF5 interface function, `H5D.read`. To read a subset of a dataset, you must use the low-level interface.

`attr = hdf5read(filename,attributename)` reads all the metadata in the attribute `attributename`, stored in the HDF5 file `filename`, and returns it in the variable `attr`. To determine the names of attributes in an HDF5 file, use the `hdf5info` function.

`[data, attr] = hdf5read(...,'ReadAttributes',BOOL)` reads all the data, as well as all of the associated attribute information contained within that data set. By default, `BOOL` is false.

`data = hdf5read(hinfo)` reads all of the data in the data set specified in the structure `hinfo` and returns it in the variable `data`. The `hinfo` structure is extracted from the output returned by `hdf5info`, which specifies an HDF5 file and a specific data set.

`[...] = hdf5read(..., 'V71Dimensions', BOOL)` specifies whether to change the majority of data sets read from the file. If `BOOL` is true, `hdf5read` permutes the first two dimensions of the data set, as it did in previous releases (MATLAB 7.1 [R14SP3] and earlier). This behavior was intended to account for the difference in how HDF5 and MATLAB express array dimensions. HDF5 describes data set dimensions in row-major order; MATLAB stores data in column-major order. However, permuting these dimensions may not correctly reflect the intent of the data and may invalidate metadata. When `BOOL` is false (the default), the data dimensions correctly reflect the data ordering as it is written in the file — each dimension in the output variable matches the same dimension in the file.

Examples

Use `hdf5info` to get information about an HDF5 file and then use `hdf5read` to read a data set, using the information structure (`hinfo`) returned by `hdf5info` to specify the data set.

```
hinfo = hdf5info('example.h5');
dset = hdf5read(hinfo.GroupHierarchy.Groups(2).Datasets(1));
```

See Also

`h5read` | `hdf5info` | `hdf5write`

Introduced before R2006a

hdf5write

Write data to file in HDF5 format

Note `hdf5write` will be removed in a future version. Use `h5write` instead.

Syntax

```
hdf5write(filename,location,dataset)
hdf5write(filename,details,dataset)
hdf5write(filename,details,attribute)
hdf5write(filename, details1, dataset1, details2, dataset2,...)
hdf5write(filename,...,'WriteMode',mode,...)
hdf5write(..., 'V71Dimensions', BOOL)
```

Description

`hdf5write(filename,location,dataset)` writes the data `dataset` to the HDF5 file, `filename`. If `filename` does not exist, `hdf5write` creates it. If `filename` exists, `hdf5write` overwrites the existing file, by default, but you can also append data to an existing file using an optional syntax.

`location` defines where to write the data set in the file. HDF5 files are organized in a hierarchical structure similar to a UNIX directory structure. `location` is specified as a character vector or string scalar that resembles a UNIX path.

`hdf5write` maps the data in `dataset` to HDF5 data types according to rules outlined below.

`hdf5write(filename,details,dataset)` writes `dataset` to `filename` using the values in the `details` structure. For a data set, the `details` structure can contain the following fields.

Field Name	Description	Data Type
`Location`	Location of the data set in the file	Character vector or string scalar
`Name`	Name to attach to the data set	Character vector or string scalar

`hdf5write(filename,details,attribute)` writes the metadata `attribute` to `filename` using the values in the `details` structure. For an attribute, the `details` structure can contain following fields.

Field Name	Description	Data Type
`AttachedTo`	Location of the object this attribute modifies	Structure array
`AttachType`	Identifies what kind of object this attribute modifies; possible values are `'group'` and `'dataset'`	Character vector or string scalar
`Name`	Name to attach to the data set	Character vector or string scalar

`hdf5write(filename, details1, dataset1, details2, dataset2,...)` writes multiple data sets and associated attributes to `filename` in one operation. Each data set and attribute must have an associated `details` structure.

`hdf5write(filename,...,'WriteMode',mode,...)` specifies whether `hdf5write` overwrites the existing file (the default) or appends data sets and attributes to the file. Possible values for mode are `'overwrite'` and `'append'`.

`hdf5write(..., 'V71Dimensions', BOOL)` specifies whether to change the majority of data sets written to the file. If `BOOL` is true, `hdf5write` permutes the first two dimensions of the data set, as it did in previous releases (MATLAB 7.1 [R14SP3] and earlier). This behavior was intended to account for the difference in how HDF5 and MATLAB express array dimensions. HDF5 describes data set dimensions in row-major order; MATLAB stores data in column-major order. However, permuting these dimensions may not correctly reflect the intent of the data and may invalidate metadata. When `BOOL` is false (the default), the data written to the file correctly reflects the data ordering of the data sets — each dimension in the file's data sets matches the same dimension in the corresponding MATLAB variable.

Data Type Mappings

The following table lists how `hdf5write` maps the data type from the workspace into an HDF5 file. If the data in the workspace that is being written to the file is a MATLAB data type, `hdf5write` uses the following rules when translating MATLAB data into HDF5 data objects.

MATLAB Data Type	HDF5 Data Set or Attribute
Numeric	Corresponding HDF5 native data type. For example, if the workspace data type is `uint8`, the `hdf5write` function writes the data to the file as 8-bit integers. The size of the HDF5 dataspace is the same size as the MATLAB array.
Character vector or string scalar	Single, null-terminated string
Cell array of character vectors or string array	Multiple, null-terminated strings, each the same length. Length is determined by the length of the longest string in the cell array. The size of the HDF5 dataspace is the same size as the cell array.
Cell array of numeric data	Numeric array, the same dimensions as the cell array. The elements of the array must all have the same size and type. The data type is determined by the first element in the cell array.
Structure array	HDF5 compound type. Individual fields in the structure employ the same data translation rules for individual data types. For example, a cell array of strings becomes a multiple, null-terminated strings.
HDF5 objects	If the data being written to the file is composed of HDF5 objects, `hdf5write` uses the same data type when writing to the file. For all HDF5 objects, except `HDF5.h5enum` objects, the dataspace has the same dimensions as the array of HDF5 objects passed to the function. For `HDF5.h5enum` objects, the size and dimensions of the data set in the HDF5 file is the same as the object's Data field.

Examples

Write a 5-by-5 data set of `uint8` values to the root group.

```
hdf5write('myfile.h5', '/dataset1', uint8(magic(5)))
```

Write a 2-by-2 data set of text entries into a subgroup.

```
dataset = {'north', 'south'; 'east', 'west'};
hdf5write('myfile2.h5', '/group1/dataset1.1', dataset);
```

Write a data set and attribute to an existing group.

```
dset = single(rand(10,10));
dset_details.Location = '/group1/dataset1.2';
dset_details.Name = 'Random';

attr = 'Some random data';
attr_details.Name = 'Description';
attr_details.AttachedTo = '/group1/dataset1.2/Random';
attr_details.AttachType = 'dataset';

hdf5write('myfile2.h5', dset_details, dset, ...
           attr_details, attr, 'WriteMode', 'append');
```

Write a data set using objects.

```
dset = hdf5.h5array(magic(5));
hdf5write('myfile3.h5', '/g1/objects', dset);
```

See Also

`hdf5info` | `hdf5read`

Introduced before R2006a

hdfan

Gateway to HDF multifile annotation (AN) interface

Syntax

```
[out1,...,outN] = hdfan(funcstr,input1,...,inputN)
```

Description

`hdfan` is the MATLAB gateway to the HDF multifile annotation (AN) interface.

`[out1,...,outN] = hdfan(funcstr,input1,...,inputN)` returns one or more outputs corresponding to the AN function in the HDF library specified by `functstr`.

There is a one-to-one correspondence between AN functions in the HDF library and valid values for `funcstr`. For example, `hdfan('endaccess',annot_id)` corresponds to the C library call `ANendaccess(annot_id)`.

Access Functions

Access functions initialize the interface and provide and terminate access to annotations.

Value of `funcstr`	Function Syntax	Description
`'start'`	`AN_id = hdfan('start',file_id)`	Initializes the multifile annotation interface.
`'select'`	`annot_id = hdfan('select',AN_id, index,annot_type)`	Selects and returns the identifier for the annotation identified by the given index value and annotation type.
`'end'`	`status = hdfan('end',AN_id)`	Terminates access to the multifile annotation interface.

Value of funcstr	Function Syntax	Description
'create'	annot_id = hdfan('create',AN_id,tag,ref,annot_type)	Creates a data annotation for the object identified by the specified tag and reference number. annot_type can be 'data_label' or 'data_desc'.
'createf'	annot_id = hdfan('createf',AN_id,annot_type)	Creates a file label or file description annotation. annot_type can be 'file_label' or 'file_desc'.
'endaccess'	status = hdfan('endaccess',annot_id)	Terminates access to an annotation.

Read/Write Functions

Read/write functions read and write file or object annotations.

Value of funcstr	Function Syntax	Description
'writeann'	status = hdfan('writeann',annot_id,annot_string)	Writes the annotation corresponding to the given annotation identifier.
'readann'	[annot_string,status] = hdfan('readann',annot_id)	Reads the annotation corresponding to the given annotation identifier;
	[annot_string,status] = hdfan('readann',annot_id,max_str_length)	Reads the annotation corresponding to the given annotation identifier. annot_string will not be longer than max_str_length.

General Inquiry Functions

General inquiry functions return information about the annotations in a file.

Value of `funcstr`	Function Syntax	Description
`'numann'`	`num_annot = hdfan('numann',AN_id,annot_type,tag,ref)`	Gets number of annotations of specified type corresponding to given tag/ref pair.
`'annlist'`	`[ann_list,status] = hdfan('annlist',AN_id,annot_type,tag,ref)`	Gets the list of annotations of given type in the file corresponding to a given tag/ref pair.
`'annlen'`	`length = hdfan('annlen',annot_id)`	Gets the length of annotation corresponding to the given annotation identifier.
`'fileinfo'`	`[nfl,nfd,ndl,ndd,status] = hdfan('fileinfo',AN_id)`	Gets number of file label, file description, data label, and data description annotations in the file corresponding to `AN_id`.
`'get_tagref'`	`[tag,ref,status] = hdfan('get_tagref',AN_id,index,annot_type)`	Gets the tag/ref pair for the specified annotation type and index.
`'id2tagref'`	`[tag,ref,status] = hdfan('id2tagref',annot_id)`	Gets the tag/ref pair corresponding to the specified annotation identifier.
`'tagref2id'`	`annot_id = hdfan('tagref2id',AN_id,tag,ref)`	Gets the annotation identifier corresponding to the specified tag/ref pair.
`'atype2tag'`	`tag = hdfan('atype2tag',annot_type)`	Gets the tag corresponding to the specified annotation type.
`'tag2atype'`	`annot_type = hdfan('tag2atype',tag)`	Gets the annotation type corresponding to the specified tag.

Input/Output Arguments

A `status` or identifier output of -1 indicates that the operation failed.

In general, the input argument `annot_type` can be one of these values:

- `'file_label'`
- `'file_desc'`
- `'data_label'`
- `'data_desc'`

`AN_id` refers to the multifile annotation interface identifier.

`annot_id` refers to an individual annotation identifier.

You must terminate access to all opened identifiers using either `hdfan('end',AN_id)` or `hdfan('endaccess',annot_id)`. Otherwise, the HDF library might not properly write all data to the file.

See Also

`hdfdf24` | `hdfdfr8` | `hdfh` | `hdfhd` | `hdfhe` | `hdfhx` | `hdfml` | `hdfv` | `hdfvf` | `hdfvh` | `hdfvs` | `matlab.io.hdf4.sd`

Introduced before R2006a

hdfdf24

Gateway to HDF 24-bit raster image (DF24) interface

Syntax

```
[out1,...,outN] = hdfdf24(funcstr,input1,...,inputN)
```

Description

`hdfdf24` is the MATLAB gateway to the HDF 24-bit raster image interface.

`[out1,...,outN] = hdfdf24(funcstr,input1,...,inputN)` returns one or more outputs corresponding to the DF24 function in the HDF library specified by `functstr`.

There is a one-to-one correspondence between DF24 functions in the HDF library and valid values for `funcstr`. For example, `hdfdf24('lastref')` corresponds to the C library call `DF24lastref()`.

Write Functions

Write functions create raster image sets and store them in new files or append them to existing files.

Value of `funcstr`	Function Syntax	Description
`'addimage'`	`status = hdfdf24('addimage',filename,RGB)`	Appends a 24-bit raster image to a file.
`'putimage'`	`status = hdfdf24('putimage',filename,RGB)`	Writes a 24-bit raster image to file by overwriting all existing data.

Value of funcstr	Function Syntax	Description
'setcompress'	status = hdfdf24('setcompress',compress_type,...)	Sets the compress method for the next raster image written to the file. compress_type can be 'none', 'rle', 'jpeg', or 'imcomp'. If compress_type is 'jpeg', then two additional parameters must be specified: quality (a scalar between 0 and 100) and force_baseline (either 0 or 1). Other compression types do not have additional parameters.
'setdims'	status = hdfdf24('setdims',width,height)	Sets the dimensions for the next raster image written to the file.
'setil'	status = hdfdf24('setil',interlace)	Sets the interlace format of the next raster image written to the file. interlace can be 'pixel', 'line', or 'component'.
'lastref'	ref = hdfdf24('lastref')	Reports the last reference number assigned to a 24-bit raster image.

Read Functions

Read functions determine the dimensions and interlace format of an image set, read the actual image data, and provide sequential or random read access to any raster image set.

Value of funcstr	Function Syntax	Description
'getdims'	[width,height,interlace,status] = hdfdf24('getdims',filename)	Retrieves the dimensions before reading the next raster image. interlace can be 'pixel', 'line', or 'component'.

Value of `funcstr`	Function Syntax	Description
`'getimage'`	`[RGB,status] = hdfdf24('getimage',filename)`	Reads the next 24-bit raster image.
`'reqil'`	`status = hdfdf24('reqil',interlace)`	Specifies the interlace format before reading the next raster image. `interlace` can be `'pixel'`, `'line'`, or `'component'`.
`'readref'`	`status = hdfdf24('readref',filename,ref)`	Reads 24-bit raster image with the specified raster number.
`'restart'`	`status = hdfdf24('restart')`	Returns to the first 24-bit raster image in the file.
`'nimages'`	`num_images = hdfdf24('nimages',filename)`	Reports the number of 24-bit raster images in a file.

Input/Output Arguments

A `status` or identifier output of -1 indicates that the operation failed.

HDF uses C-style ordering of elements, in which elements along the last dimension vary fastest. MATLAB uses FORTRAN-style ordering, in which elements along the first dimension vary fastest. `hdfdf24` does not automatically convert from C-style ordering to MATLAB style ordering, which means that MATLAB image arrays need to be permuted when using `hdfdf24` to read or write from HDF files. The exact permutation depends on the interlace format specified by, for example, `hdfdf24('setil',...)`. The following calls to `permute` converts HDF arrays to MATLAB arrays, according to the specified interlace format.

`RGB = permute(RGB,[3 2 1]);`	`'pixel'` interlace
`RGB = permute(RGB,[3 1 2]);`	`'line'` interlace
`RGB = permute(RGB,[2 1 3]);`	`'component'` interlace

See Also

`hdfdf24` | `hdfdfr8` | `hdfh` | `hdfhd` | `hdfhe` | `hdfhx` | `hdfml` | `hdfv` | `hdfvf` | `hdfvh` | `hdfvs` | `matlab.io.hdf4.sd`

Introduced before R2006a

hdfdfr8

Gateway to HDF 8-bit raster image (DFR8) interface

Syntax

```
[out1,...,outN] = hdfdfr8(funcstr,input1,...,inputN)
```

Description

`hdfdfr8` is the MATLAB gateway to the HDF 8-bit raster image (DFR8) interface.

`[out1,...,outN] = hdfdfr8(funcstr,input1,...,inputN)` returns one or more outputs corresponding to the DFR8 function in the HDF library specified by `functstr`.

There is a one-to-one correspondence between DFR8 functions in the HDF library and valid values for `funcstr`. For example, `hdfdfr8('setpalette',map)` corresponds to the C library call `DFR8setpalette(map)`.

Write Functions

Write functions create raster image sets and store them in new files or append them to existing files.

Value of `funcstr`	Function Syntax	Description
`'writeref'`	`status = hdfdfr8('writeref',filename,ref)`	Stores the raster image using the specified reference number.
`'setpalette'`	`status = hdfdfr8('setpalette',colormap)`	Sets palette for multiple 8-bit raster images.
`'addimage'`	`status = hdfdfr8('addimage',filename,X,compress)`	Appends an 8-bit raster image to a file. `compress` can be `'none'`, `'rle'`, `'jpeg'`, or `'imcomp'`.

Value of funcstr	Function Syntax	Description
'putimage'	status = hdfdfr8('putimage',filename,X,compress)	Writes an 8-bit raster image to an existing file or creates the file. compress can be 'none', 'rle', 'jpeg', or 'imcomp'.
'setcompress'	status = hdfdfr8('setcompress',compress_type,...)	Sets the compression type. compress_type can be 'none', 'rle', 'jpeg', or 'imcomp'. If compress_type is 'jpeg', then two additional parameters must be passed in: quality (a scalar between 0 and 100) and force_baseline (either 0 or 1). Other compression types do not have additional parameters.

Read Functions

Read functions determine the dimension and palette assignment for an image set, read the actual image data, and provide sequential or random read access to any raster image set.

Value of funcstr	Function Syntax	Description
'getdims'	[width,height,hasmap,status] = hdfdfr8('getdims',filename)	Retrieves dimensions for an 8-bit raster image.
'getimage'	[X,map,status] = hdfdfr8('getimage',filename)	Retrieves an 8-bit raster image and its palette.
'readref'	status = hdfdfr8('readref',filename,ref)	Gets the next raster image with the specified reference number.
'restart'	status = hdfdfr8('restart')	Ignores information about last file accessed and restarts from beginning.

Value of `funcstr`	Function Syntax	Description
`'nimages'`	`num_images = hdfdfr8('nimages',filename)`	Returns number of raster images in a file.
`'lastref'`	`ref = hdfdfr8('lastref')`	Returns reference number of last element accessed.

Input/Output Arguments

A `status` or identifier output of -1 indicates that the operation failed.

HDF uses C-style ordering of elements, in which elements along the last dimension vary fastest.MATLAB uses FORTRAN-style ordering, in which elements along the first dimension vary fastest. `hdfdfr8` does not automatically convert from C-style ordering to MATLAB style ordering, which means that MATLAB image and colormap matrices must be transposed when using `hdfdfr8` to read or write from HDF files.

Functions in `hdfdfr8` that read and write palette information expect to use `uint8` data in the range [0,255], while MATLAB colormaps contain double-precision values in the range [0,1]. Therefore, HDF palettes must be converted to `double` and scaled to be used as MATLAB colormaps.

See Also

`hdfdf24` | `hdfdfr8` | `hdfh` | `hdfhd` | `hdfhe` | `hdfhx` | `hdfml` | `hdfv` | `hdfvf` | `hdfvh` | `hdfvs` | `matlab.io.hdf4.sd`

Introduced before R2006a

hdfh

Gateway to HDF H interface

Syntax

```
[out1,...,outN] = hdfh(funcstr,input1,...,inputN)
```

Description

`hdfh` is the MATLAB gateway to the HDF H interface.

`[out1,...,outN] = hdfh(funcstr,input1,...,inputN)` returns one or more outputs corresponding to the H function in the HDF library specified by `functstr`.

There is a one-to-one correspondence between H functions in the HDF library and valid values for `funcstr`. For example, `hdfh('close',file_id)` corresponds to the C library call `Hclose(file_id)`.

Functions

Value of `funcstr`	Function Syntax	Description
`'appendable'`	`status = hdfh('appendable',access_id)`	Specifies that the element can be appended to.
`'close'`	`status = hdfh('close',file_id)`	Closes the access path to the file.
`'deldd'`	`status = hdfh('deldd',file_id,tag,ref)`	Deletes a tag and reference number from the data descriptor list.
`'dupdd'`	`status = hdfh('dupdd',file_id,tag,ref,old_tag,old_ref)`	

Value of `funcstr`	Function Syntax	Description
`'endaccess'`	`status = hdfh('endaccess',access_id)`	Terminates access to a data object by disposing of the access identifier.
`'fidinquire'`	`[filename,access_mode,attach,status] = hdfh('fidinquire',file_id)`	Returns information about specified file.
`'find'`	`[tag,ref,offset,length,status] = hdfh('find',file_id,... search_tag,search_ref,search_type,dir)`	Locates the next object to be searched for in an HDF file. `search_type` can be `'new'` or `'continue'`. The `dir` input can be `'forward'` or `'backward'`.
`'getelement'`	`[data,status] = hdfh('getelement',file_id,tag,ref)`	Reads the data element for the specified tag and reference number.
`'getfileversion'`	`[major,minor,release,info,status] = hdfh('getfileversion',file_id)`	Returns version information for an HDF file.
`'getlibversion'`	`[major,minor,release,info,status] = hdfh('getlibversion')`	Returns version information for the current HDF library.
`'inquire'`	`[file_id,tag,ref,length,offset,position,access,... special,status] = hdfh('inquire',access_id)`	Returns access information about a data element.
`'ishdf'`	`tf = hdfh('ishdf',filename)`	Determines if a file is an HDF file.
`'length'`	`length = hdfh('length',file_id,tag,ref)`	Returns the length of a data object specified by the tag and reference number.

Value of `funcstr`	Function Syntax	Description
`'newref'`	`ref = hdfh('newref',file_id)`	Returns a reference number that can be used with any tag to product a unique tag/reference number pair.
`'nextread'`	`status = hdfh('nextread',access_id,tag,ref,origin)`	Searches for the next data descriptor that matches the specified tag and reference number. `origin` can be `'start'` or `'current'`.
`'number'`	`num = hdfh('number',file_id,tag)`	Returns the number of instances of a tag in a file.
`'offset'`	`offset = hdfh('offset',file_id,tag,ref)`	Returns the offset of a data element in the file.
`'open'`	`file_id = hdfh('open',filename,access,n_dds)`	Provides an access path to an HDF file by reading all the data descriptor blocks into memory.
`'putelement'`	`count = hdfh('putelement',file_id,tag,ref,X)`	Writes a data element or replaces an existing data element in an HDF file. `X` must be a `uint8` array.
`'read'`	`X = hdfh('read',access_id,length)`	Reads the next segment in a data element.
`'seek'`	`status = hdfh('seek',access_id,offset,origin)`	Sets the access pointer to an offset within a data element. `origin` can be `'start'` or `'current'`.
`'startread'`	`access_id = hdfh('startread',file_id,tag,ref)`	

Value of `funcstr`	Function Syntax	Description
`'startwrite'`	`access_id = hdfh('startwrite',file_id,tag,ref,length)`	
`'sync'`	`status = hdfh('sync',file_id)`	
`'trunc'`	`length = hdfh('trunc',access_id,trunc_len)`	Truncates the specified data object to the given length.
`'write'`	`count = hdfh('write',access_id,X)`	Writes the next data segment to a specified data element. `X` must be a `uint8` array.

Output Arguments

A `status` or identifier output of -1 indicates that the operation failed.

Limitations

- `hdfh` does not support these functions in the NCSA H interface:
  - `Hcache`
  - `Hendbitaccess`
  - `Hexist`
  - `Hflushdd`
  - `Hgetbit`
  - `Hputbit`
  - `Hsetlength`
  - `Hshutdown`
  - `Htagnewref`

See Also

`hdfan` | `hdfdf24` | `hdfdfr8` | `hdfhd` | `hdfhe` | `hdfhx` | `hdfml` | `hdfv` | `hdfvf` | `hdfvh` | `hdfvs` | `matlab.io.hdf4.sd`

Introduced before R2006a

hdfhd

Gateway to HDF HD interface

Syntax

```
[out1,...,outN] = hdfhd(funcstr,input1,...,inputN)
```

Description

`hdfhd` is the MATLAB gateway to the HDF HD interface.

`[out1,...,outN] = hdfhd(funcstr,input1,...,inputN)` returns one or more outputs corresponding to the HD function in the HDF library specified by `functstr`.

There is a one-to-one correspondence between HD functions in the HDF library and valid values for `funcstr`.

Functions

Value of `funcstr`	Function Syntax	Description
`'gettagsname'`	`tag_name = hdfhd('gettagsname',tag)`	Gets the name of the specified tag.

Output Arguments

A `status` or identifier output of -1 indicates that the operation failed.

See Also

`hdfan` | `hdfdf24` | `hdfdfr8` | `hdfh` | `hdfhd` | `hdfhe` | `hdfml` | `hdfv` | `hdfvf` | `hdfvh` | `hdfvs` | `matlab.io.hdf4.sd`

Introduced before R2006a

hdfhe

Gateway to HDF HE interface

Syntax

```
[out1,...,outN] = hdfhe(funcstr,input1,...,inputN)
```

Description

`hdfhe` is the MATLAB gateway to the HDF HE interface.

This is a stub page.

`[out1,...,outN] = hdfhe(funcstr,input1,...,inputN)` returns one or more outputs corresponding to the HE function in the HDF library specified by `functstr`.

There is a one-to-one correspondence between HE functions in the HDF library and valid values for `funcstr`.

Functions

Value of `funcstr`	Function Syntax	Description
`'clear'`	`hdfhe('clear')`	Clears all information on reported errors from the error stack.
`'print'`	`hdfhe('print',level)`	Prints information in error stack. If `level` is 0, then the entire error stack is printed.
`'string'`	`error_text = hdfhe('string',error_code)`	Returns the error message associated with the specified error code.

Value of `funcstr`	Function Syntax	Description
`'value'`	`error_code = hdfhe('value',stack_offset)`	Returns an error code from the specified level of the error stack. A `stack_offset` value of 1 gets the most recent error code.

Output Arguments

A `status` or identifier output of -1 indicates that the operation failed.

Limitations

- `hdfhe` does not support these functions:
  - `HEpush`
  - `HEreport`

See Also

`hdfan` | `hdfdf24` | `hdfdfr8` | `hdfh` | `hdfhd` | `hdfhe` | `hdfml` | `hdfv` | `hdfvf` | `hdfvh` | `hdfvs` | `matlab.io.hdf4.sd`

Introduced before R2006a

hdfhx

Gateway to HDF external data (HX) interface

Syntax

```
[out1,...,outN] = hdfhx(funcstr,input1,...,inputN)
```

Description

`hdfhx` is the MATLAB gateway to the HDF interface for manipulating linked and external data elements.

`[out1,...,outN] = hdfhx(funcstr,input1,...,inputN)` returns one or more outputs corresponding to the HX function in the HDF library specified by `functstr`.

There is a one-to-one correspondence between HX functions in the HDF library and valid values for `funcstr`. For example, `hdfhx('setdir',pathname);` corresponds to the C library call `HXsetdir(pathname)`.

Functions

Value of `funcstr`	Function Syntax	Description
`'create'`	`access_id = hdfhx('create',file_id,tag,ref,extern_name,offset,length)`	Creates a new external file special data element.
`'setcreatedir'`	`status = hdfhx('setcreatedir',pathname);`	Sets the directory location for writing external file.
`'setdir'`	`status = hdfhx('setdir',pathname);`	Sets the directory for locating external files. `pathname` can contain multiple directories separated by vertical bars.

Input/Output Arguments

A `status` or identifier output of -1 indicates that the operation failed.

In cases where the HDF C library accepts `NULL` for certain inputs, an empty matrix (`[]` or `''`) can be used.

See Also

`hdfan` | `hdfdf24` | `hdfdfr8` | `hdfh` | `hdfhd` | `hdfhe` | `hdfml` | `hdfv` | `hdfvf` | `hdfvh` | `hdfvs` | `matlab.io.hdf4.sd`

hdfinfo

Information about HDF4 or HDF-EOS file

Syntax

```
S = hdfinfo(filename)
S = hdfinfo(filename,mode)
```

Description

`S = hdfinfo(filename)` returns a structure `S` whose fields contain information about the contents of an HDF4 or HDF-EOS file. Specify `filename` as a character vector or string scalar containing the name of the HDF4 file.

`S = hdfinfo(filename,mode)` reads the file as an HDF4 file, if `mode` is `'hdf'`, or as an HDF-EOS file, if `mode` is `'eos'`. If `mode` is `'eos'`, only HDF-EOS data objects are queried. To retrieve information on the entire contents of a file containing both HDF4 and HDF-EOS objects, `mode` must be `'hdf'`.

Note `hdfinfo` can be used on Version 4.x HDF files or Version 2.x HDF-EOS files. To get information about an HDF5 file, use `hdf5info`.

The set of fields in the returned structure `S` depends on the individual file. Fields that can be present in the `S` structure are shown in the following table.

Mode	Field Name	Description	Return Type
`HDF`	`Attributes` on page 1-6235	Attributes of the data set	Structure array
	`Description`	Annotation description	Cell array
	`Filename`	Name of the file	Character vector
	`Label`	Annotation label	Cell array

Mode	Field Name	Description	Return Type
	`Raster8` on page 1-6235	Description of 8-bit raster images	Structure array
	`Raster24` on page 1-6235	Description of 24-bit raster images	Structure array
	`SDS` on page 1-6236	Description of scientific data sets	Structure array
	`Vdata` on page 1-6236	Description of Vdata sets	Structure array
	`Vgroup` on page 1-6236	Description of Vgroups	Structure array
`EOS`	`Filename`	Name of the file	Character vector
	`Grid` on page 1-6237	Grid data	Structure array
	`Point` on page 1-6237	Point data	Structure array
	`Swath` on page 1-6238	Swath data	Structure array

Those fields in the table above that contain structure arrays are further described in the tables shown below.

Fields Common to Returned Structure Arrays

Structure arrays returned by `hdfinfo` contain some common fields. These are shown in the table below. Not all structure arrays will contain all of these fields.

Field Name	Description	Data Type
`Attributes`	Data set attributes. Contains fields `Name` and `Value`.	Structure array
`Description`	Annotation description	Cell array
`Filename`	Name of the file	Character vector
`Label`	Annotation label	Cell array
`Name`	Name of the data set	Character vector

Field Name	Description	Data Type
`Rank`	Number of dimensions of the data set	Double
`Ref`	Data set reference number	Double
`Type`	Type of HDF or HDF-EOS object	Character vector

Fields Specific to Certain Structures

Structure arrays returned by `hdfinfo` also contain fields that are unique to each structure. These are shown in the tables below.

Fields of the Attribute Structure

Field Name	Description	Data Type
`Name`	Attribute name	Character vector
`Value`	Attribute value or description	Numeric or Text

Fields of the Raster8 and Raster24 Structures

Field Name	Description	Data Type
`HasPalette`	`1` (`true`) if the image has an associated palette, otherwise `0` (`false`) (8-bit only)	Logical
`Height`	Height of the image, in pixels	Number
`Interlace`	Interlace mode of the image (24-bit only)	Character vector
`Name`	Name of the image	Character vector
`Width`	Width of the image, in pixels	Number

Fields of the SDS Structure

Field Name	Description	Data Type
`DataType`	Data precision	Character vector
`Dims`	Dimensions of the data set. Contains fields `Name`, `DataType`, `Size`, `Scale`, and `Attributes`. `Scale` is an array of numbers to place along the dimension and demarcate intervals in the data set.	Structure array
`Index`	Index of the SDS	Number

Fields of the Vdata Structure

Field Name	Description	Data Type
`DataAttributes`	Attributes of the entire data set. Contains fields `Name` and `Value`.	Structure array
`Class`	Class name of the data set	Character vector
`Fields`	Fields of the Vdata. Contains fields `Name` and `Attributes`.	Structure array
`NumRecords`	Number of data set records	Double
`IsAttribute`	`1` (`true`) if Vdata is an attribute, otherwise `0` (`false`)	Logical

Fields of the Vgroup Structure

Field Name	Description	Data Type
`Class`	Class name of the data set	Character vector
`Raster8`	Description of the 8-bit raster image	Structure array
`Raster24`	Description of the 24-bit raster image	Structure array
`SDS`	Description of the Scientific Data sets	Structure array
`Tag`	Tag of this Vgroup	Number
`Vdata`	Description of the Vdata sets	Structure array
`Vgroup`	Description of the Vgroups	Structure array

Fields of the Grid Structure

Field Name	Description	Data Type
`Columns`	Number of columns in the grid	Number
`DataFields`	Description of the data fields in each `Grid` field of the grid. Contains fields `Name`, `Rank`, `Dims`, `NumberType`, `FillValue`, and `TileDims`.	Structure array
`LowerRight`	Lower right corner location, in meters	Number
`Origin Code`	Origin code for the grid	Number
`PixRegCode`	Pixel registration code	Number
`Projection`	Projection code, zone code, sphere code, and projection parameters of the grid. Contains fields `ProjCode`, `ZoneCode`, `SphereCode`, and `ProjParam`.	Structure
`Rows`	Number of rows in the grid	Number
`UpperLeft`	Upper left corner location, in meters	Number

Fields of the Point Structure

Field Name	Description	Data Type
`Level`	Description of each level of the point. Contains fields `Name`, `NumRecords`, `FieldNames`, `DataType`, and `Index`.	Structure

Fields of the Swath Structure

Field Name	Description	Data Type
`DataFields`	Data fields in the swath. Contains fields `Name`, `Rank`, `Dims`, `NumberType`, and `FillValue`.	Structure array
`GeolocationFields`	Geolocation fields in the swath. Contains fields `Name`, `Rank`, `Dims`, `NumberType`, and `FillValue`.	Structure array
`IdxMapInfo`	Relationship between indexed elements of the geolocation mapping. Contains fields `Map` and `Size`.	Structure
`MapInfo`	Relationship between data and geolocation fields. Contains fields `Map`, `Offset`, and `Increment`.	Structure

Examples

To retrieve information about the file `example.hdf`,

```
fileinfo = hdfinfo('example.hdf')

fileinfo = 
    Filename: 'example.hdf'
         SDS: [1x1 struct]
       Vdata: [1x1 struct]
```

And to retrieve information from this about the scientific data set in `example.hdf`,

```
sds_info = fileinfo.SDS

sds_info = 
       Filename: 'example.hdf'
           Type: 'Scientific Data Set'
           Name: 'Example SDS'
           Rank: 2
       DataType: 'int16'
     Attributes: []
           Dims: [2x1 struct]
          Label: {}
```

```
    Description: {}
          Index: 0
```

See Also

hdfread

Introduced before R2006a

hdfml

Utilities for working with MATLAB HDF gateway functions

Syntax

```
hdfml('closeall')
hdfml('listinfo')
tag = hdfml('tagnum',tagname)
nbytes = hdfml('sizeof',data_type)
hdfml('defaultchartype',char_type)
```

Description

`hdfml('closeall')` closes all open registered HDF file and data object identifiers.

`hdfml('listinfo')` prints information about all open registered HDF file and data object identifiers.

`tag = hdfml('tagnum',tagname)` returns the tag number corresponding to the tag name specified by `tagname`.

`nbytes = hdfml('sizeof',data_type)` returns size in bytes of specified data type.

`hdfml('defaultchartype',char_type)` defines the HDF data type for text data in MATLAB. Valid values for `char_type` are `'char8'` or `'uchar8'`. The change persists until the MATLAB HDF gateway function is cleared from memory. Text data in MATLAB are mapped to `char8` by default.

The MATLAB HDF gateway functions maintain lists of certain HDF file and data object identifiers so that, for example, HDF objects and files can be properly closed when a user issues the command:

```
clear mex
```

These lists are updated whenever these identifiers are created or closed.

See Also

`hdfan` | `hdfdf24` | `hdfdfr8` | `hdfh` | `hdfhd` | `hdfhe` | `hdfml` | `hdfv` | `hdfvf` | `hdfvh` | `hdfvs` | `matlab.io.hdf4.sd`

Introduced before R2006a

hdfpt

Interface to HDF-EOS Point object

Syntax

```
[out1,...,outN] = hdfpt(funcstr,input1,...,inputN)
```

Description

`hdfpt` is the MATLAB gateway to the Point functions in the HDF-EOS C library, which is developed and maintained by EOSDIS (Earth Observing System Data and Information System). A Point data set comprises a series of data records taken at (possibly) irregular time intervals and at scattered geographic locations. Each data record consists of a set of one or more data values representing the state of a point in time and/or space.

`[out1,...,outN] = hdfpt(funcstr,input1,...,inputN)` returns one or more outputs corresponding to the Point function in the HDF-EOS library specified by `functstr`.

There is a one-to-one correspondence between PT functions in the HDF-EOS C library and valid values for `funcstr`. For example, `hdfpt('detach',point_id)` corresponds to the C library call `PTdetach(point_id)`.

Programming Model

The programming model for accessing a point data set through `hdfpt` is as follows:

1. Open the file and initialize the PT interface by obtaining a file id from a file name.
2. Open or create a point data set by obtaining a point id from a point name.
3. Perform desired operations on the data set.
4. Close the point data set by disposing of the point id.
5. Terminate point access to the file by disposing of the file id.

To access a single point data set that already exists in an HDF-EOS file, use the following MATLAB commands:

```
fileid = hdfpt('open',filename,access);
pointid = hdfpt('attach',fileid,pointname);

% Optional operations on the data set...

status = hdfpt('detach',pointid);
status = hdfpt('close',fileid);
```

To access several files at the same time, obtain a separate file identifier for each file to be opened. To access more than one point data set, obtain a separate point id for each data set.

It is important to properly dispose of point id's and file id's so that buffered operations are written completely to disk. If you quit MATLAB or clear all MEX-files with PT identifiers still open, MATLAB issues a warning and automatically disposes of them.

Note that file identifiers returned by `hdfpt` are not interchangeable with file identifiers returned by any other HDF-EOS or HDF function.

Access Routines

Access routines initialize and terminate access to the PT interface and point data sets (including opening and closing files).

Value of `funcstr`	Function Syntax	Description
`'open'`	`file_id = hdfpt('open',filename,access)`	Given the filename and desired access mode, opens or creates an HDF file in order to create, read, or write a point. `access` can be `'read'`, `'readwrite'`, or `'create'`. `file_id` is -1 if the operation fails.

Value of `funcstr`	Function Syntax	Description
`'create'`	`point_id = hdfpt('create',file_id,pointname)`	Creates a point data set with the specified name. `pointname` is a character vector or string scalar containing the name of the point data set. `point_id` is -1 if the operation fails.
`'attach'`	`point_id = hdfpt('attach',file_id,pointname)`	Attaches to an existing point data set within the file. `point_id` is -1 if the operation fails.
`'detach'`	`status = hdfpt('detach',point_id)`	Detaches from point data set.
`'close'`	`status = hdfpt('close',file_id)`	Closes file.

Definition Routines

Definition routines allow the user to set key features of a point data set.

Value of `funcstr`	Function Syntax	Description
`'deflevel'`	`status = hdfpt('deflevel',point_id,levelname,... fieldList,fieldTypes,fieldOrders)`	Defines a new level within a point data set. `levelname` is the name of the level to be defined. `fieldList` is a cell array of character vectors or string array containing field names in the new level. `fieldTypes` is also a cell array of character vectors or string array containing the number type for each field in the `fieldList`. Valid number types include `'uchar8'`, `'uchar'`, `'char8'`, `'char'`, `'double'`, `'uint8'`, `'uint16'`, `'uint32'`, `'float'`, `'int8'`, `'int16'`, and `'int32'`. `fieldOrders` is a vector containing the order for each field.
`'deflinkage'`	`status = hdfpt('deflinkage',point_id,parent,child,linkfield)`	Defines a linkfield between two adjacent levels. `parent` is the name of the parent level. `child` is the name of the child level. `linkfield` is the name of a field that is defined at both levels.

Basic I/O Routines

Basic I/O routines read and write data and metadata to a point data set.

Value of funcstr	Function Syntax	Description
'writelevel'	status = hdfpt('writelevel',point_id,level,data)	Appends new records to the specified level in a point data set. level is the desired level index (zero-based). data must be a P-by-1 cell array where P is the number of fields defined for the specified level. Each cell of data must contain an M(k)-by-N matrix of data, where M(k) is the order of the k-th field (the number of scalar values in the field) and N is the number of records. The MATLAB class of the cells must match the HDF data type defined for the corresponding fields. Text data in MATLAB is automatically converted to match any of the HDF char types. Other data types must match exactly.
'readlevel'	[data,status] = hdfpt('readlevel',point_id,... level,fieldList,records)	Reads data from a given level in a point data set. level is the index (zero-based) of the desired level. fieldList is a cell array of character vectors or string array specifying the list of the fields to read. records is a vector containing the indices (zero-based) of the records to read. data is a P-by-1 cell array where P is the number of requested fields. Each cell of data contains an M(k)-by-N matrix of data where M(k) is the order of the k-th field and N is the number of records, or length(records).

Value of `funcstr`	Function Syntax	Description
`'updatelevel'`	`status = hdfpt('updatelevel',point_id,... level,fieldList,records,data)`	Updates (corrects) data in a particular level of a point data set. `level` is the index (zero-based) of the desired level. `fieldList` is a cell array of character vectors or string array specifying the list of field names to update. `records` is a vector containing the indices (zero-based) of the records to update. `data` is a `P`-by-1 cell array where `P` is the number of specified fields. Each cell of `data` must contain an `M(k)`-by-`N` matrix of data, where `M(k)` is the order of the `k`-th field (the number of scalar values in the field) and `N` is the number of records, or `length(records)`. The MATLAB class of the cells must match the HDF data type defined for the corresponding fields. Text data in MATLAB is automatically converted to match any of the HDF char types. Other data types must match exactly.
`'writeattr'`	`status = hdfpt('writeattr',point_id,attrname,data)`	Writes or updates the point data set attribute with the specified name. If the attribute does not already exist, it is created.
`'readattr'`	`[data,status] = hdfpt('readattr',point_id,attrname)`	Reads the attribute data from the specified attribute.

Inquiry Routines

Inquiry routines return information about data contained in a point data set.

Value of `funcstr`	Function Syntax	Description
`'nlevels'`	`nlevels = hdfpt('nlevels',point_id)`	Returns the number of levels in a point data set. `nlevels` is -1 if the operation fails.
`'nrecs'`	`nrecs = hdfpt('nrecs',point_id,level)`	Returns the number of records in the specified level. `nrecs` is -1 if the operation fails.
`'nfields'`	`[numfields,strbufsize] = hdfpt('nfields',point_id,level)`	Returns the number of fields in the specified level. `strbufsize` is the length of array containing the field names. `numfields` is -1 and `strbufsize` is `[]` if the operation fails.
`'levelinfo'`	`[numfields,fieldList,field Type,fieldOrder] = ... hdfpt('levelinfo',point_id,level)`	Returns information on fields for a specified level. `fieldList` is a cell array of character vectors or string array containing the field names. `fieldType` is a cell array of character vectors that defined the data type for each field. `fieldOrder` is a vector containing the order (number of scalar values) associated with each field. If the operation fails, `numfields` is -1 and the other outputs are empty.
`'levelindx'`	`level = hdfpt('levelindx',point_id,levelname)`	Returns the level index (zero-based) of the level with the specified name. `level` is -1 if the operation fails.
`'bcklinkinfo'`	`[linkfield,status] = hdfpt('bcklinkinfo',point_id,level)`	Returns the linkfield to the previous level. `status` is -1 and `linkfield` is `[]` if the operation fails.

Value of `funcstr`	Function Syntax	Description
`'fwdlinkinfo'`	`[linkfield,status] = hdfpt('fwdlinkinfo',point_id,level)`	Returns the linkfield to the following level. `status` is -1 and `linkfield` is `[]` if the operation fails.
`'getlevelname'`	`[levelname,status] = hdfpt('getlevelname',point_id,level)`	Returns the name of a level given the level index. `status` is -1 and `levelname` is `[]` if the operation fails.
`'sizeof'`	`[byteSize,fieldLevels] = hdfpt('sizeof',point_id,fieldList)`	Returns the size in bytes and field levels of the specified fields. `fieldList` is a cell array of character vectors or string array containing the field names. `byteSize` is the total size of bytes of the specified fields, and `fieldLevels` is a vector containing the level index corresponding to each field. `byteSize` is -1 and `fieldLevels` is `[]` if the operation fails.
`'attrinfo'`	`[numberType,count,status] = ... hdfpt('attrinfo',point_id,attrname)`	Returns the number type and size in bytes of the specified attribute. `attrname` is the name of the attribute. `numberType` is a character vector containing the name of the corresponding HDF data type of the attribute. `count` is the number of bytes used by the attribute data. `status` is -1 and `numberType` and `count` are `[]` if the operation fails.

Value of funcstr	Function Syntax	Description
'inqattrs'	[nattrs,attrnames] = hdfpt('inqattrs',point_id)	Retrieve information about attributes defined in a point data set. nattrs and attrnames are the number and names of all the defined attributes, respectively. If the operation fails, nattrs is -1 and attrnames is [].
'inqpoint'	[numpoints,pointnames] = hdfpt('inqpoint',filename)	Retrieve number and names of point data sets defined in an HDF-EOS file. pointnames is a cell array of character vectors containing a the point names. numpoints is -1 and pointnames is [] if the operation fails.

Utility Routines

Placeholder.

Value of funcstr	Function Syntax	Description
'getrecnums'	[outRecords,status] = hdfpt('getrecnums',... point_id,inLevel,outLevel,inRecords)	Returns the record numbers in outLevel corresponding the group of records specified by inRecords in level inLevel. The inLevel and outLevel arguments are zero-based level indices. inRecords is a vector of zero-based record indices. status is -1 and outRecords is [] if the operation fails.

Subset Routines

Subset routines allow reading of data from a specified geographic region.

Value of `funcstr`	Function Syntax	Description
`'defboxregion'`	`region_id = hdfpt('defboxregion',point_id,cornerLon,cornerLat)`	Defines a longitude-latitude box region for a point. `cornerLon` is a two-element vector containing the longitudes of opposite box corners. `cornerLat` is a two-element vector containing the latitudes of opposite box corners. `region_id`is -1 if the operation fails.
`'defvrtregion'`	`period_id = hdfpt('defvrtregion',point_id,region_id,... vert_field,range)`	Defines a vertical region for a point. `vert_field` is the name of the field to subset. `range` is a two-element vector containing the minimum and maximum vertical values. `period_id` is -1 if the operation fails.
`'regioninfo'`	`[byteSize,status] = hdfpt('regioninfo',point_id,... region_id,level,fieldList)`	Returns the data size in bytes of the subset period of the specified level. `fieldlist` is a cell array of character vectors or string array specifying the list of fields to extract. `status` and `byteSize` are -1 if the operation fails.
`'regionrecs'`	`[numRec,recNumbers,status] = hdfpt('regionrecs',... point_id,region_id,level)`	Returns the records numbers within the subsetted region of the specified level. `status` and `numrec` are -1 and `recNumbers` is `[]` if the operation fails.

Value of `funcstr`	Function Syntax	Description
`'extractregion'`	`[data,status] = hdfpt('extractregion',point_id,... region_id,level,fieldList)`	Reads data from the specified subset region. `fieldList` is a cell array of character vectors or string array specifying the list of requested fields. `data` is a `P`-by-1 cell array where `P` is the number of requested fields. Each cell of `data` contains an `M(k)`-by-`N` matrix of data where `M(k)` is the order of the `k`-th field and `N` is the number of records. `status` is -1 and `data` is `[]` if the operation fails.
`'deftimeperiod'`	`period_id = hdfpt('deftimeperiod',point_id,startTime,stopTime)`	Defines a time period for a point data set. `period_id` is -1 if the operation fails.
`'periodinfo'`	`[byteSize,status] = hdfpt('periodinfo',point_id,... period_id,level,fieldList)`	Retrieves the size in bytes of the subsetted period. `fieldList` is a cell array of character vectors or string array specifying the list of field names. `byteSize` and `status` are -1 if the operation fails.
`'periodrecs'`	`[numRec,recNumbers,status] = hdfpt('periodrecs',... point_id,period_id,level)`	Returns the records numbers within the subsetted time period of the specified level. `numRec` and `status` are -1 if the operation fails.

Value of funcstr	Function Syntax	Description
'extractperiod'	[data,status] = hdfpt('extractperiod',... point_id,period_id,level, fieldList)	Reads data from the specified subsetted time period. fieldList is a cell array of character vectors or string array specifying the list of requested fields. data is a P-by-1 cell array where P is the number of requested fields. Each cell of data contains an M(k)-by-N matrix of data where M(k) is the order of the k-th field and N is the number of records. status is -1 and data is [] if the operation fails.

Input/Output Arguments

Most routines return the flag, status, which is 0 when the routine succeeds and -1 when the routine fails. Routines with syntaxes which do not contain status return failure information in one of its outputs as notated in the function syntaxes.

levelName is a character vector or string scalar.

Some of the C library functions accept input values that are defined in terms of C macros. For example, the C PTopen() function requires an access mode input that can be DFACC_READ, DFACC_RDWR, or DFACC_CREATE, where these symbols are defined in the appropriate C header file. Where macro definitions are used in the C library, the equivalent MATLAB syntaxes use text derived from the macro names. You can either use text containing the entire macro name, or you can omit the common prefix. You can use either upper or lower case. For example, this C function call:

```
status = PTopen("PointFile.hdf",DFACC_CREATE)
```

is equivalent to these MATLAB function calls:

```
status = hdfpt('open','PointFile.hdf','DFACC_CREATE')
status = hdfpt('open','PointFile.hdf','dfacc_create')
status = hdfpt('open','PointFile.hdf','CREATE')
status = hdfpt('open','PointFile.hdf','create')
```

In cases where a C function returns a value with a macro definition, the equivalent MATLAB function returns the value as text containing the lowercase short form of the macro.

HDF number types are specified as: `'uchar8'`, `'uchar'`, `'char8'`, `'char'`, `'double'`, `'uint8'`, `'uint16'`, `'uint32'`, `'float'`, `'int8'`, `'int16'`, and `'int32'`.

In cases where the HDF-EOS library accepts `NULL`, use an empty matrix (`[]`).

See Also

`matlab.io.hdfeos.sw`

Introduced before R2006a

hdfread

Read data from HDF4 or HDF-EOS file

Syntax

```
data = hdfread(filename, datasetname)
data = hdfread(hinfo)
data = hdfread(...,param,value,...)
data = hdfread(filename,EOSname,param,value,...)
[data,map] = hdfread(...)
```

Description

`data = hdfread(filename, datasetname)` returns all the data in the data set specified by `datasetname` from the HDF4 or HDF-EOS file specified by `filename`. To determine the name of a data set in an HDF4 file, use the `hdfinfo` function.

Note `hdfread` can be used on Version 4.x HDF files or Version 2.x HDF-EOS files. To read data from an HDF5 file, use `h5read`.

`data = hdfread(hinfo)` returns all the data in the data set specified by the structure`hinfo`, returned by the `hdfinfo` function. Specify the field in the `hinfo` structure that relates to a particular type of data set, and use indexing to specify which data set, when there are more than one. See “Specify data set to read” on page 1-6263 for more information.

`data = hdfread(...,param,value,...)` returns subsets of the data according to the specified parameter and value pairs. See the tables below to find the valid parameters and values for different types of data sets.

`data = hdfread(filename,EOSname,param,value,...)` subsets the data field from the HDF-EOS point, grid, or swath specified by `EOSname`.

`[data,map] = hdfread(...)` returns the image `data` and the colormap `map` for an 8-bit raster image.

Subsetting Parameters

The following tables show the subsetting parameters that can be used with the `hdfread` function for certain types of HDF4 data. These data types are

- HDF Scientific Data (SD) on page 1-6256
- HDF Vdata (V) on page 1-6257
- HDF-EOS Grid Data on page 1-6258
- HDF-EOS Point Data on page 1-6260
- HDF-EOS Swath Data on page 1-6260

Note the following:

- If a parameter requires multiple values, use a cell array to store the values. For example, the `'Index'` parameter requires three values: `start`, `stride`, and `edge`. Enclose these values in curly braces as a cell array.

  ```
  hdfread(..., 'Index', {start,stride,edge})
  ```
- All values that are indices are 1-based.

Subsetting Parameters for HDF Scientific Data (SD) Data Sets

When you are working with HDF SD files, `hdfread` supports the parameters listed in this table.

Parameter	Description
`'Index'`	Three-element cell array, `{start,stride,edge}`, specifying the location, range, and values to be read from the data set • `start` — A 1-based array specifying the position in the file to begin reading Default: `1`, start at the first element of each dimension. The values specified must not exceed the size of any dimension of the data set. • `stride` — A 1-based array specifying the interval between the values to read Default: `1`, read every element of the data set. • `edge` — A 1-based array specifying the length of each dimension to read Default: An array containing the lengths of the corresponding dimensions

For example, this code reads the data set `Example SDS` from the HDF file `example.hdf`. The `'Index'` parameter specifies that `hdfread` start reading data at the beginning of each dimension, read until the end of each dimension, but only read every other data value in the first dimension.

```
data = hdfread('example.hdf','Example SDS','Index',{[],[2 1],[]})
```

Subsetting Parameters for HDF Vdata Sets

When you are working with HDF Vdata files, `hdfread` supports these parameters.

Parameter	Description
`'Fields'`	Character vector or string scalar specifying the name of the field to be read. When specifying multiple field names, use a cell array of character vectors or a string array.
`'FirstRecord'`	1-based number specifying the record from which to begin reading
`'NumRecords'`	Number specifying the total number of records to read

For example, this code reads the Vdata set `Example Vdata` from the HDF file `example.hdf`.

```
data = hdfread('example.hdf','Example Vdata','FirstRecord', 2,'NumRecords', 5)
```

Subsetting Parameters for HDF-EOS Grid Data

When you are working with HDF-EOS grid data, `hdfread` supports three types of parameters:

- Required parameters
- Optional parameters
- Mutually exclusive parameters — You can only specify one of these parameters in a call to `hdfread`, and you cannot use these parameters in combination with any optional parameter.

Parameter	Description
Required Parameter	
`'Fields'`	Character vector or string scalar specifying the field to be read. You can specify only one field name for a Grid data set.
Mutually Exclusive Optional Parameters	
`'Index'`	Three-element cell array, `{start,stride,edge}`, specifying the location, range, and values to be read from the data set `start` — An array specifying the position in the file to begin reading Default: `1`, start at the first element of each dimension. The values must not exceed the size of any dimension of the data set. `stride` — An array specifying the interval between the values to read Default: `1`, read every element of the data set. `edge` — An array specifying the length of each dimension to read Default: An array containing the lengths of the corresponding dimensions
`'Interpolate'`	Two-element cell array, `{longitude,latitude}`, specifying the longitude and latitude points that define a region for bilinear interpolation. Each element is an N-length vector specifying longitude and latitude coordinates.

Parameter	Description
`'Pixels'`	Two-element cell array, `{longitude,latitude}`, specifying the longitude and latitude coordinates that define a region. Each element is an N-length vector specifying longitude and latitude coordinates. This region is converted into pixel rows and columns with the origin in the upper left corner of the grid. Note: This is the pixel equivalent of reading a `'Box'` region.
`'Tile'`	Vector specifying the coordinates of the tile to read, for HDF-EOS Grid files that support tiles
Optional Parameters	
`'Box'`	Two-element cell array, `{longitude,latitude}`, specifying the longitude and latitude coordinates that define a region. `longitude` and `latitude` are each two-element vectors specifying longitude and latitude coordinates.
`'Time'`	Two-element cell array, `[start stop]`, where `start` and `stop` are numbers that specify the start and end-point for a period of time
`'Vertical'`	Two-element cell array, `{dimension, range}` `dimension` — Character vector or string scalar specifying the name of the data set field to be read from. You can specify only one field name for a Grid data set. `range` — Two-element array specifying the minimum and maximum range for the subset. If `dimension` is a dimension name, then `range` specifies the range of elements to extract. If `dimension` is a field name, then `range` specifies the range of values to extract. `'Vertical'` subsetting can be used alone or in conjunction with `'Box'` or `'Time'`. To subset a region along multiple dimensions, vertical subsetting can be used up to eight times in one call to `hdfread`.

For example,

```
data = hdfread('grid.hdf','PolarGrid','Fields','ice_temp','Index', {[5 10],[],[15 20]})
```

Subsetting Parameters for HDF-EOS Point Data

When you are working with HDF-EOS Point data, `hdfread` has two required parameters and three optional parameters.

Parameter	Description
Required Parameters	
`'Fields'`	Character vector or string scalar containing the name of the data set field to be read. For multiple field names, use a cell array of character vectors or a string array.
`'Level'`	1-based number specifying which level to read from in an HDF-EOS Point data set
Mutually Exclusive Optional Parameters	
`'Box'`	Two-element cell array, `{longitude,latitude}`, specifying the longitude and latitude coordinates that define a region. `longitude` and `latitude` are each two-element vectors specifying longitude and latitude coordinates.
`'RecordNumbers'`	Vector specifying the record numbers to read
`'Time'`	Two-element cell array, `[start stop]`, where `start` and `stop` are numbers that specify the start and endpoint for a period of time

For example,

```
hdfread(...,'Fields',{field1, field2},...
            'Level',level,'RecordNumbers',[1:50, 200:250])
```

Subsetting Parameters for HDF-EOS Swath Data

When you are working with HDF-EOS Swath data, `hdfread` supports three types of parameters:

- Required parameters
- Optional parameters
- Mutually exclusive

You can only use one of the mutually exclusive parameters in a call to `hdfread`, and you cannot use these parameters in combination with any optional parameter.

Parameter	Description
Required Parameter	
`'Fields'`	Character vector or string scalar containing the name of the data set field to be read. You can specify only one field name for a Swath data set.
Mutually Exclusive Optional Parameters	
`'Index'`	Three-element cell array, `{start,stride,edge}`, specifying the location, range, and values to be read from the data set • `start` — An array specifying the position in the file to begin reading Default: `1`, start at the first element of each dimension. The values must not exceed the size of any dimension of the data set. • `stride` — An array specifying the interval between the values to read Default: `1`, read every element of the data set. • `edge` — An array specifying the length of each dimension to read Default: An array containing the lengths of the corresponding dimensions
`'Time'`	Three-element cell array, `{start, stop, mode}`, where `start` and `stop` specify the beginning and the endpoint for a period of time, and `mode` is a character vector or string scalar which defines the criterion for the inclusion of a cross track in a region. The cross track is within a region if any of these conditions is met: • Its midpoint is within the box (`mode='midpoint'`). • Either endpoint is within the box (`mode='endpoint'`).
Optional Parameters	

Parameter	Description
'Box'	Three-element cell array, {longitude, latitude, mode} specifying the longitude and latitude coordinates that define a region. longitude and latitude are two-element vectors that specify longitude and latitude coordinates. mode is a character vector or string scalar which defines the criterion for the inclusion of a cross track in a region. The cross track is within a region if any of these conditions is met: • Its midpoint is within the box (mode='midpoint'). • Either endpoint is within the box (mode='endpoint'). • Any point is within the box (mode='anypoint').
'Vertical'	Two-element cell array, {dimension, range} • dimension is a character vector or string scalar specifying either a dimension name or field name to subset the data by. • range is a two-element vector specifying the minimum and maximum range for the subset. If dimension is a dimension name, then range specifies the range of elements to extract. If dimension is a field name, then range specifies the range of values to extract. 'Vertical' subsetting can be used alone or in conjunction with 'Box' or 'Time'. To subset a region along multiple dimensions, vertical subsetting can be used up to eight times in one call to hdfread.

For example,

```
hdfread('swath.hdf', 'Example Swath', 'Fields', 'Temperature', ...
         'Time', {5000, 6000, 'midpoint'})
```

Examples

Read Data Set in HDF File

Specify the name of the HDF file and the name of the data set. This example reads a data set named temperature from a sample HDF file.

```
data = hdfread('sd.hdf','temperature');
```

Specify data set to read

Call `hdfinfo` to retrieve information about the contents of the HDF file.

```
fileinfo = hdfinfo('sd.hdf')
```

```
fileinfo = struct with fields:
      Filename: 'B:\matlab\toolbox\matlab\imagesci\sd.hdf'
    Attributes: [1x1 struct]
           SDS: [1x2 struct]
         Vdata: [1x1 struct]
```

Extract the structure containing information about the particular data set you want to import from the data returned by `hdfinfo`. This example uses the structure in the `SDS` field to retrieve a scientific data set.

```
sds_info = fileinfo.SDS(2)
```

```
sds_info = struct with fields:
       Filename: 'B:\matlab\toolbox\matlab\imagesci\sd.hdf'
           Type: 'Scientific Data Set'
           Name: 'temperature'
           Rank: 2
       DataType: 'double'
     Attributes: [1x11 struct]
           Dims: [2x1 struct]
          Label: {}
    Description: {}
          Index: 1
```

Pass this structure to `hdfread` to import the data in the data set.

```
data = hdfread(sds_info);
```

Read Data from HDF-EOS Grid Field

Read data from the HDF-EOS global grid field, `TbOceanRain`, in the example file, `example.hdf`.

```
data1 = hdfread('example.hdf','MonthlyRain','Fields','TbOceanRain');
```

Read data for the northern hemisphere for the same field. Use the `Box` parameter to specify the longitude and latitude coordinates for that region.

```
data2 = hdfread('example.hdf','MonthlyRain', ...
'Fields','TbOceanRain', ...
'Box', {[0 360],[0 90]});
```

Read Subset of Data in Data Set

Retrieve info about the example file, `example.hdf`.

```
fileinfo = hdfinfo('example.hdf');
```

Retrieve information about Scientific Data Set in `example.hdf`.

```
data_set_info = fileinfo.SDS;
```

Check the size of the data set.

```
data_set_info.Dims.Size
```

```
ans = 16

ans = 5
```

Read a subset of the data in the data set using the `'index'` parameter with `hdfread`. This example specifies a starting index of `[3 3]`, an interval of 1 between values (`[]` meaning the default value of 1), and a length of 10 rows and 2 columns.

```
data = hdfread(data_set_info,'Index',{[3 3],[],[10 2]});
data(:,1)
```

```
ans = 10x1 int16 column vector

    7
    8
    9
   10
   11
   12
   13
```

```
   14
   15
   16
```

```
data(:,2)
```

```
ans = 10x1 int16 column vector

    8
    9
   10
   11
   12
   13
   14
   15
   16
   17
```

Access Data in Fields of Vdata

Use the `Vdata` field from the information returned by `hdfinfo` to read three fields of the data, `Idx`, `Temp`, and `Dewpt`.

```
s = hdfinfo('example.hdf');
data = hdfread(s.Vdata(1),'Fields',{'Idx','Temp','Dewpt'})
```

```
data=3×1 cell
    {1x10 int16}
    {1x10 int16}
    {1x10 int16}
```

See Also

`hdfinfo`

Introduced before R2006a

hdftool

(Not recommended) Browse and import data from HDF4 or HDF-EOS files

Syntax

Note `hdftool` is not recommended. Use `hdfread` instead.

Syntax

```
hdftool
hdftool(filename)
h = hdftool( ___ )
```

Description

`hdftool` starts the HDF Import Tool, a graphical user interface used to browse the contents of HDF4 and HDF-EOS files and import data and subsets of data from these files. To open an HDF4 or HDF-EOS file, select **Open** from the **Home** tab. You can open multiple files in the HDF Import Tool by selecting **Open** from the **Home** tab.

`hdftool(filename)` opens the HDF4 or HDF-EOS file specified by `filename` in the HDF Import Tool.

`h = hdftool( ___ )` returns a handle `h` to the HDF Import Tool. To close the tool from the command line, use `close(h)`.

Examples

```
hdftool('example.hdf');
```

See Also

`hdfinfo` | `hdfread` | `uiimport`

Introduced before R2006a

hdfv

Gateway to HDF Vgroup (V) interface

Syntax

```
[out1,...,outN] = hdfv(funcstr,input1,...,inputN)
```

Description

`hdfv` is the MATLAB gateway to the HDF Vgroup (V) interface.

`[out1,...,outN] = hdfv(funcstr,input1,...,inputN)` returns one or more outputs corresponding to the V function in the HDF library specified by `functstr`.

There is a one-to-one correspondence between V functions in the HDF library and valid values for `funcstr`. For example, `hdfv('nattrs',vgroup_id)` corresponds to the C library call `Vnattrs(vgroup_id)`.

Access Functions

Access functions open files, initialize the Vgroup interface, and access individual groups. They also terminate access to vgroups and the Vgroup interface and close HDF files.

Value of `funcstr`	Function Syntax	Description
`'start'`	`status = hdfv('start',file_id)`	Initializes the V interface.
`'attach'`	`vgroup_id = hdfv('attach',file_id,vgroup_ref,access)`	Establishes access to a vgroup. `access` can be `'r'` or `'w'`.
`'detach'`	`status = hdfv('detach',vgroup_id)`	Terminates access to a vgroup.
`'end'`	`status = hdfv('end',file_id)`	Terminates access to the V interface.

Create Functions

Create functions organize, label, and add data objects to vgroups.

Value of `funcstr`	Function Syntax	Description
`'setclass'`	`status = hdfv('setclass',vgroup_id,class)`	Assigns a class to a vgroup.
`'setname'`	`status = hdfv('setname',vgroup_id,name)`	Assigns a name to a vgroup.
`'insert'`	`ref = hdfv('insert',vgroup_id,id)`	Adds a vgroup or vdata to an existing group. `id` can be a vdata id or a vgroup id.
`'addtagref'`	`status = hdfv('addtagref',vgroup_id,tag,ref)`	Adds any HDF data object to an existing vgroup.
`'setattr'`	`status = hdfv('setattr',vgroup_id,name,A)`	Sets the attribute of a vgroup.

File Inquiry Functions

File inquiry functions return information about how vgroups are stored in a file. They are useful for locating vgroups in a file.

Value of `funcstr`	Function Syntax	Description
`'lone'`	`[refs,count] = hdfv('lone',file_id,maxsize)`	Returns the reference numbers of vgroups not included in other vgroups.
`'getid'`	`next_ref = hdfv('getid',file_id,vgroup_ref)`	Returns the reference number for the next vgroup in the HDF file.

Value of `funcstr`	Function Syntax	Description
`'find'`	`vgroup_ref = hdfv('find',file_id,vgroup_name)`	Returns the reference number of the vgroup with the specified name if successful and zero otherwise.
`'findclass'`	`vgroup_ref = hdfv('findclass',file_id,class)`	Returns the reference number of the vgroup with the specified class.

Vgroup Inquiry Functions

Vgroup inquiry functions provide specific information about a specific vgroup. This information includes the class, name, member count, and additional member information.

Value of `funcstr`	Function Syntax	Description
`'getclass'`	`[class_name,status] = hdfv('getclass',vgroup_id)`	Returns the name of the class of the specified group.
`'getname'`	`[vgroup_name,status] = hdfv('getname',vgroup_id)`	Returns the name of the specified group.
`'inquire'`	`[num_entries,name,status] = hdfv('inquire',vgroup_id)`	Returns the number of entries and the name of a vgroup.
`'isvg'`	`status = hdfv('isvg',vgroup_id,ref)`	Checks if the object specified by ref refers to a child vgroup of the vgroup specified by `vgroup_id`.
`'isvs'`	`status = hdfv('isvs',vgroup_id,vdata_ref)`	Checks if the object specified by `vdata_ref` refers to a child vdata of the vgroup specified by `vgroup_id`.
`'gettagref'`	`[tag,ref,status] = hdfv('gettagref',vgroup_id,index)`	Retrieves a tag/reference number pair for a data object in the specified vgroup.

Value of funcstr	Function Syntax	Description
'ntagrefs'	count = hdfv('ntagrefs',vgroup_id)	Returns the number of tag/reference number pairs contained in the specified vgroup.
'gettagrefs'	[tag,refs,count] = hdfv('gettagrefs',vgroup_id,maxsize)	Retrieves the tag/reference pairs of all the data objects within a vgroup.
'inqtagref'	tf = hdfv('inqtagref',vgroup_id,tag,ref)	Checks if an object belongs to a vgroup.
'getversion'	version = hdfv('getversion',vgroup_id)	Queries the vgroup version of a given vgroup.
'nattrs'	count = hdfv('nattrs',vgroup_id)	Queries the total number of vgroup attributes.
'attrinfo'	[name,data_type,count,nbytes,status] = hdfv('attrinfo',vgroup_id,... attr_index)	Queries information on a given vgroup attribute.
'getattr'	[values,status] = hdfv('getattr',vgroup_id,attr_index)	Queries the values of a given attribute.
'Queryref'	ref = hdfv('Queryref',vgroup_id)	Returns the reference number of the specified vgroup.
'Querytag'	tag = hdfv('Querytag',vgroup_id)	Returns the tag of the specified vgroup.
'flocate'	vdata_ref = hdfv('flocate',vgroup_id,field)	Returns the reference number of the vdata containing the specified field name in the specified vgroup.

Value of `funcstr`	Function Syntax	Description
`'nrefs'`	`count = hdfv('nrefs',vgroup_id,tag)`	Returns the number of data objects with the specified tag in the specified vgroup.

Output Arguments

A `status` or identifier output of -1 indicates that the operation failed.

See Also

`hdfdf24` | `hdfdfr8` | `hdfh` | `hdfhd` | `hdfhe` | `hdfhx` | `hdfml` | `hdfv` | `hdfvf` | `hdfvh` | `hdfvs` | `matlab.io.hdf4.sd`

Introduced before R2006a

hdfvf

Gateway to VF functions in HDF Vdata interface

Syntax

```
[out1,...,outN] = hdfvf(funcstr,input1,...,inputN)
```

Description

`hdfvf` is the MATLAB gateway to the VF unctions in the HDF Vdata interface.

`[out1,...,outN] = hdfvf(funcstr,input1,...,inputN)` returns one or more outputs corresponding to the VF function in the HDF library specified by `functstr`.

There is a one-to-one correspondence between VF functions in the HDF library and valid values for `funcstr`. For example, `hdfvf('nfields',vdata_id)` corresponds to the C library call `VFnfields(vdata_id)`.

Field Inquiry Functions

Field inquiry functions provide specific information about the fields in a given vdata, including the field's size, name, order, type, and number of fields in the vdata.

Value of `funcstr`	Function Syntax	Description
`'fieldesize'`	`fsize = hdfvf('fieldesize',vdata_id,field_index)`	Retrieves the field size (as stored in a file) of a specified field.
`'fieldisize'`	`fsize = hdfvf('fieldisize',vdata_id,field_index)`	Retrieves the field size (as stored in memory) of a specified field.

Value of `funcstr`	Function Syntax	Description
`'fieldname'`	`name = hdfvf('fieldname',vdata_id,field_index)`	Retrieves the name of the specified field in the given vdata.
`'fieldorder'`	`order = hdfvf('fieldorder',vdata_id,field_index)`	Retrieves the order of the specified field in the given vdata.
`'fieldtype'`	`data_type = hdfvf('fieldtype',vdata_id,field_index)`	Retrieves the data type for the specified field in the given vdata.
`'nfields'`	`count = hdfvf('nfields',vdata_id)`	Retrieves the total number of fields in the specified vdata.

Output Arguments

A `status` or identifier output of -1 indicates that the operation failed.

See Also

`hdfdf24` | `hdfdfr8` | `hdfh` | `hdfhd` | `hdfhe` | `hdfhx` | `hdfml` | `hdfv` | `hdfvf` | `hdfvh` | `hdfvs` | `matlab.io.hdf4.sd`

Introduced before R2006a

hdfvh

Gateway to VH functions in HDF Vdata interface

Syntax

```
[out1,...,outN] = hdfvh(funcstr,input1,...,inputN)
```

Description

`hdfvh` is the MATLAB gateway to VH functions in the HDF Vdata interface.

`[out1,...,outN] = hdfvh(funcstr,input1,...,inputN)` returns one or more outputs corresponding to the VH function in the HDF library specified by `functstr`.

There is a one-to-one correspondence between VH functions in the HDF library and valid values for `funcstr`.

High-level Vdata Functions

High-level Vdata functions write data to single-field vdatas.

Value of `funcstr`	Function Syntax	Description
`'makegroup'`	`vgroup_ref = hdfvh('makegroup',file_id ,tags,refs,.. vgroup_name,vgroup_class)`	Groups a collection of data objects within a vgroup.
`'storedata'`	`count = hdfvh('storedata',file_id ,fieldname,data,... vdata_name,vdata_class)`	Creates vdatas containing records limited to one field with one component per field.

Value of `funcstr`	Function Syntax	Description
`'storedatam'`	`count = hdfvh('storedatam',file_id,fieldname,data,... vdata_name,vdata_class)`	Creates vdatas containing records with one field containing one or more components.

Output Arguments

A `status` or identifier output of -1 indicates that the operation failed.

See Also

`hdfdf24` | `hdfdfr8` | `hdfh` | `hdfhd` | `hdfhe` | `hdfhx` | `hdfml` | `hdfv` | `hdfvf` | `hdfvh` | `hdfvs` | `matlab.io.hdf4.sd`

Introduced before R2006a

hdfvs

Gateway to VS functions in HDF Vdata interface

Syntax

```
[out1,...,outN] = hdfvs(funcstr,input1,...,inputN)
```

Description

`hdfvs` is the MATLAB gateway to the VS functions in the HDF Vdata interface.

`[out1,...,outN] = hdfvs(funcstr,input1,...,inputN)` returns one or more outputs corresponding to the VS function in the HDF library specified by `functstr`.

There is a one-to-one correspondence between VS functions in the HDF library and valid values for `funcstr`. For example, `hdfvs('detach',vdata_id)` corresponds to the C library call `VSdetach(vdata_id)`.

Access Functions

Access functions attach, or allow access, to vdatas. Data transfer can only occur after a vdata has been accessed. These routines also detach from, or properly terminate access to, vdatas when data transfer has been completed.

Value of `funcstr`	Function Syntax	Description
`'attach'`	`vdata_id = hdfvs('attach',file_id,vdata_ref,access)`	Establishes access to a specified vdata. `access` can be `'r'` or `'w'`.
`'detach'`	`status = hdfvs('detach',vdata_id)`	Terminates access to a specified vdata.

Read and Write Functions

Read and write functions read and write the contents of a vdata.

Value of `funcstr`	Function Syntax	Description
`'fdefine'`	`status = hdfvs('fdefine',vdata_id,fieldname,data_type,order)`	Defines a new vdata field. `data_type` is character vector or string scalar containing the HDF number type. Specify `data_type` as one of these values: `'uchar8'`, `'uchar'`, `'char8'`, `'char'`, `'double'`, `'uint8'`, `'uint16'`, `'uint32'`, `'float'`, `'int8'`, `'int16'`, or `'int32'`.
`'setclass'`	`status = hdfvs('setclass',vdata_id,class)`	Assigns a class to a vdata.
`'setfields'`	`status = hdfvs('setfields',vdata_id,fields)`	Specifies the vdata fields to be written.
`'setinterlace'`	`status = hdfvs('setinterlace',vdata_id,interlace)`	Sets the interlace mode for a vdata. `interlace` can be `'full'` or `'no'`.
`'setname'`	`status = hdfvs('setname',vdata_id,name)`	Assigns a name to a vdata.
`'write'`	`count = hdfvs('write', vdata_id, data)`	Writes to a vdata. data must be an `nfields`-by-1 cell array. Each cell must contain an `order(i)`-by-`n` vector of data where `order(i)` is the number of scalar values in each field. The types of the data must match the field types set via `hdfvs('setfields')` or the fields in an already existing vdata.

Value of `funcstr`	Function Syntax	Description
`'read'`	`[data,count] = hdfvs('read',vdata_id,n)`	Reads from a vdata. Data is returned in a `nfields`-by-1 cell array. Each cell contains a `order(i)`-by-`n` vector of data where `order` is the number of scalar values in each field. The fields are returned in the same order as specified in `hdfvs('setfields',...)`.
`'seek'`	`pos = hdfvs('seek',vdata_id,record)`	Seeks to a specified record in a vdata.
`'setattr'`	`status = hdfvs('setattr',vdata_id,field_index,name,A)`	Sets the attribute of a vdata field or vdata.
`'setexternalfile'`	`status = hdfvs('setexternalfile',vdata_id,filename,offset)`	Stores vdata information in an external file.
`'getattr'`	`[value,status] = hdfvs('getattr',vdata_id,field_index,attr_index)`	Reads the value of an attribute attached to a vdata or a vdata field. Set `field_index` to `'vdata'` to retrieve an attribute attached to the field itself. Set `field_index` to the numerical index of the field to retrieve an attribute attached to a vdata field.
`'setattr'`	`status = hdfvs('setattr',vdata_id,field_index,name,A)`	Sets the attribute of a vdata field or vdata.`field_index` can be an index number or `'vdata'`.

File Inquiry Functions

File inquiry functions provide information about how vdatas are stored in a file. They are useful for locating vdatas in a file.

Value of `funcstr`	Function Syntax	Description
`'find'`	`vdata_ref = hdfvs('find',file_id,vdata_name)`	Searches for a given vdata name in the specified HDF file.
`'findclass'`	`vdata_ref = hdfvs('findclass',file_id,vdata_class)`	Returns the reference number of the first vdata corresponding to the specified vdata class.
`'getid'`	`next_ref = hdfvs('getid',file_id,vdata_ref)`	Returns the identifier of the next vdata in the file.
`'lone'`	`[refs,count] = hdfvs('lone',file_id,maxsize)`	Returns the reference numbers of the vdatas that are not linked into vgroups.

Vdata Inquiry Functions

Vdata inquiry functions provide specific information about a given vdata, including the vdata's name, class, number of fields, number of records, tag and reference pairs, interlace mode, and size.

Value of `funcstr`	Function Syntax	Description
`'fexist'`	`status = hdfvs('fexist',vdata_id,fields)`	Tests for the existence of fields in the specified vdata.
`'inquire'`	`[n,interlace,fields,nbytes,vdata_name,status] = ... hdfvs('inquire',vdata_id)`	Returns information about the specified vdata.
`'elts'`	`count = hdfvs('elts',vdata_id)`	Returns the number of records in the specified vdata.

Value of `funcstr`	Function Syntax	Description
`'getclass'`	`[class_name,status] = hdfvs('getclass',vdata_id)`	Returns the HDF class of the specified vdata.
`'getfields'`	`[field_names,count] = hdfvs('getfields',vdata_id)`	Returns all field names within the specified vdata.
`'getinterlace'`	`[interlace,status] = hdfvs('getinterlace',vdata_id)`	Retrieves the interlace mode of the specified vdata.
`'getname'`	`[vdata_name,status] = hdfvs('getname',vdata_id)`	Retrieves the name of the specified vdata.
`'getversion'`	`version = hdfvs('getversion',vdata_id)`	Returns the version number of a vdata.
`'sizeof'`	`nbytes = hdfvs('sizeof',vdata_id,fields)`	Returns the fields sizes of the specified vdata.
`'Queryfields'`	`[fields,status] = hdfvs('Queryfields',vdata_id)`	Returns the field names of the specified vdata.
`'Queryname'`	`[name,status] = hdfvs('Queryname',vdata_id)`	Returns the name of the specified vdata.
`'Queryref'`	`ref = hdfvs('Queryref',vdata_id)`	Retrieves the reference number of the specified vdata.
`'Querytag'`	`tag = hdfvs('Querytag',vdata_id)`	Retrieves the tag of the specified vdata.
`'Querycount'`	`[count,status] = hdfvs('Querycount',vdata_id)`	Returns the number of records in the specified vdata.

Value of `funcstr`	Function Syntax	Description
`'Queryinterlace'`	`[interlace,status] = hdfvs('Queryinterlace',vdata_id)`	Returns the interlace mode of the specified vdata.
`'Queryvsize'`	`vsize = hdfvs('Queryvsize',vdata_id)`	Retrieves the local size in bytes of the specified vdata record.
`'findex'`	`[field_index,status] = hdfvs('findex',vdata_id,fieldname)`	Queries the index of a vdata field given the field name.
`'nattrs'`	`count = hdfvs('nattrs',vdata_id)`	Returns the number of attributes of the specified vdata and the vdata fields contained in it.
`'fnattrs'`	`count = hdfvs('fnattrs',vdata_id,field_index)`	Queries the total number of vdata attributes.
`'findattr'`	`attr_index = hdfvs('findattr',vdata_id,field_index,attr_name)`	Retrieves the index of an attribute given the attribute name.
`'isattr'`	`tf = hdfvs('isattr',vdata_id)`	Determines if the given vdata is an attribute.
`'attrinfo'`	`[name,data_type,count,nbytes,status] = hdfvs('attrinfo',... vdata_id,field_index,attr_index)`	Returns the name, data type, number of values, and the size of the values of the specified attributes of the specified vdata field or vdata.

Output Arguments

A `status` or identifier output of -1 indicates that the operation failed.

See Also

`hdfdf24` | `hdfdfr8` | `hdfh` | `hdfhd` | `hdfhe` | `hdfhx` | `hdfml` | `hdfv` | `hdfvf` | `hdfvh` | `hdfvs` | `matlab.io.hdf4.sd`

Introduced before R2006a

heatmap

Create heatmap chart

Syntax

```
h = heatmap(tbl,xvar,yvar)
h = heatmap(tbl,xvar,yvar,'ColorVariable',cvar)

h = heatmap(cdata)
h = heatmap(xvalues,yvalues,cdata)

h = heatmap( ___ ,Name,Value)
h = heatmap(parent, ___ )
```

Description

`h = heatmap(tbl,xvar,yvar)` creates a heatmap from the table `tbl` and returns the `HeatmapChart` object. The `xvar` input indicates the table variable to display along the *x*-axis. The `yvar` input indicates the table variable to display along the *y*-axis. The default colors are based on a count aggregation, which totals the number of times each pair of *x* and *y* values appears together in the table. Use `h` to modify the heatmap after it is created. For a list of properties, see HeatmapChart.

`h = heatmap(tbl,xvar,yvar,'ColorVariable',cvar)` uses the table variable specified by `cvar` to calculate the color data. The default calculation method is a mean aggregation.

`h = heatmap(cdata)` creates a heatmap from matrix `cdata`. The heatmap has one cell for each value in `cdata`.

`h = heatmap(xvalues,yvalues,cdata)` specifies the labels for the values that appear along the *x*-axis and *y*-axis.

`h = heatmap( ___ ,Name,Value)` specifies additional options for the heatmap using one or more name-value pair arguments. Specify the options after all other input arguments. For a list of properties, see HeatmapChart.

`h = heatmap(parent, ___ )` creates the heatmap in the figure, panel, or tab specified by `parent`.

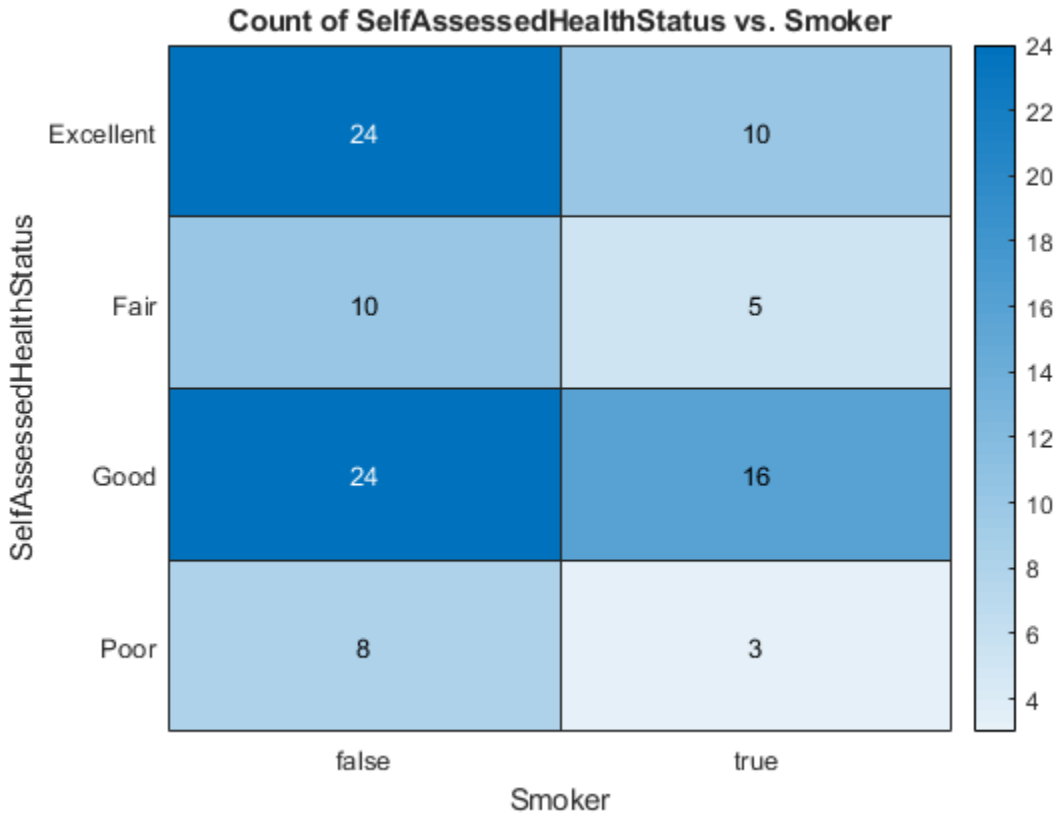
Examples

Create Heatmap from Tabular Data

Create a heatmap from a table of data for medical patients.

Load the `patients` data set and create a table from a subset of the variables loaded into the workspace. Then create a heatmap that counts the total number of patients with the same set of `Smoker` and `SelfAssessedHealthStatus` values.

```
load patients
tbl = table(LastName,Age,Gender,SelfAssessedHealthStatus,...
    Smoker,Weight,Location);
h = heatmap(tbl,'Smoker','SelfAssessedHealthStatus');
```

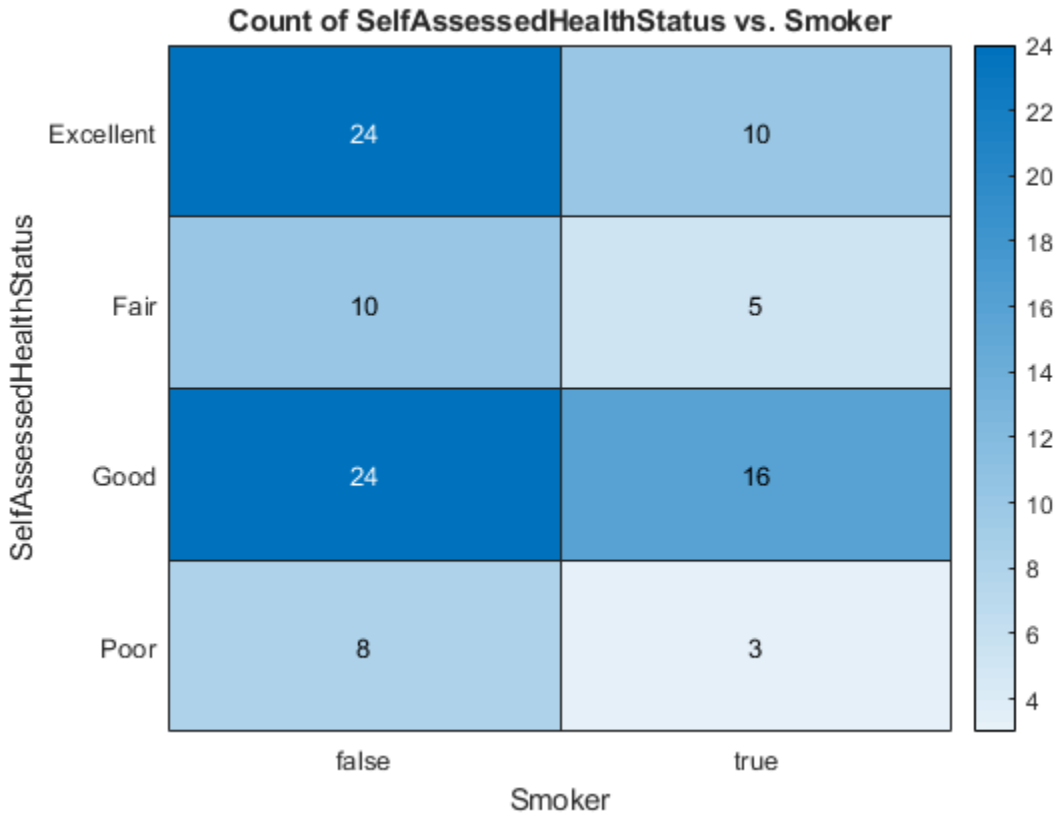


Reorder Heatmap Labels

Create a heatmap and reorder the labels along the *y*-axis.

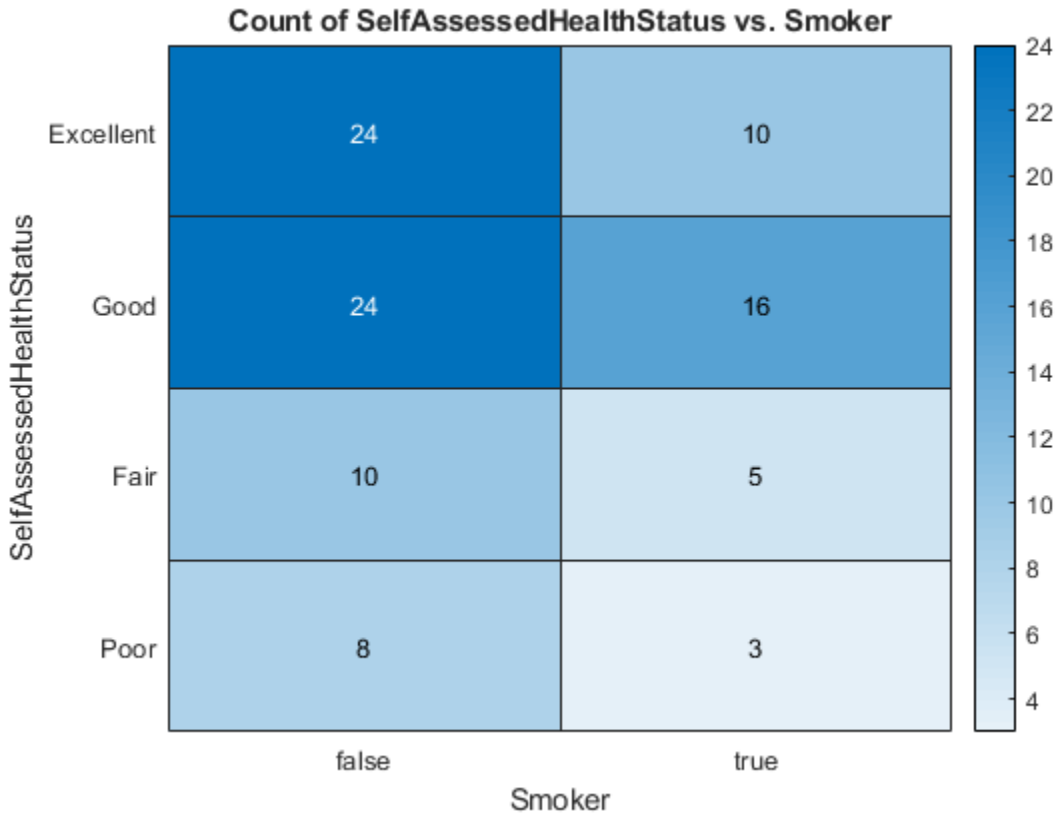
Load the `patients` data set and create a heatmap from the data. Assign the `HeatmapChart` object to the variable `h`.

```
load patients
tbl = table(LastName,Age,Gender,SelfAssessedHealthStatus,...
    Smoker,Weight,Location);
h = heatmap(tbl,'Smoker','SelfAssessedHealthStatus');
```



Reorder the labels along the *y*-axis.

```
h.YDisplayData = {'Excellent','Good','Fair','Poor'};
```



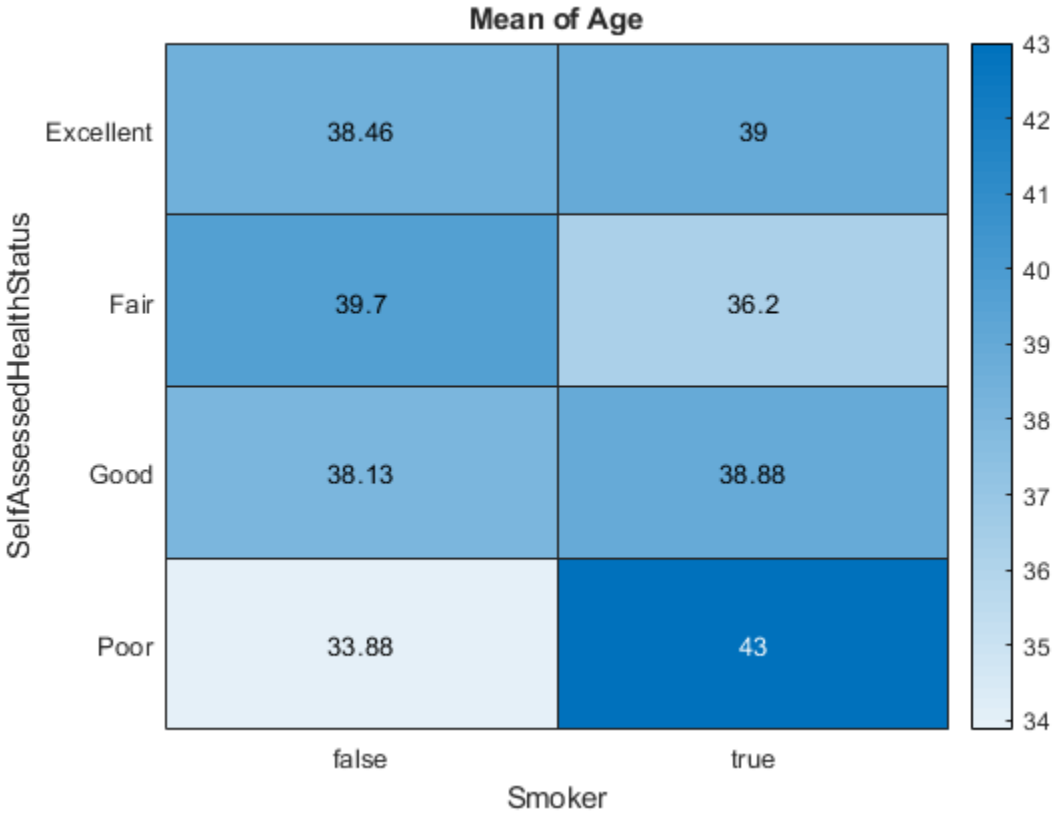
Alternatively, you can reorder the labels by changing the data to categorical data and then reordering the categories using the `reordercats` function. Similarly, you can add, remove, or rename the heatmap labels using the `addcats`, `removecats`, or `renamecats` functions, respectively.

Specify Table Variable for Heatmap Colors

Create a heatmap and specify the table variable to use when determining the heatmap cell colors.

Load the `patients` data set and create a heatmap from the data. Color each cell using the average age of patients with a particular pair of `Smoker` and `SelfAssessedHealthStatus` values by setting the `ColorVariable` option to `'Age'`.

```
load patients
tbl = table(LastName,Age,Gender,SelfAssessedHealthStatus,...
    Smoker,Weight,Location);
h = heatmap(tbl,'Smoker','SelfAssessedHealthStatus','ColorVariable','Age');
```

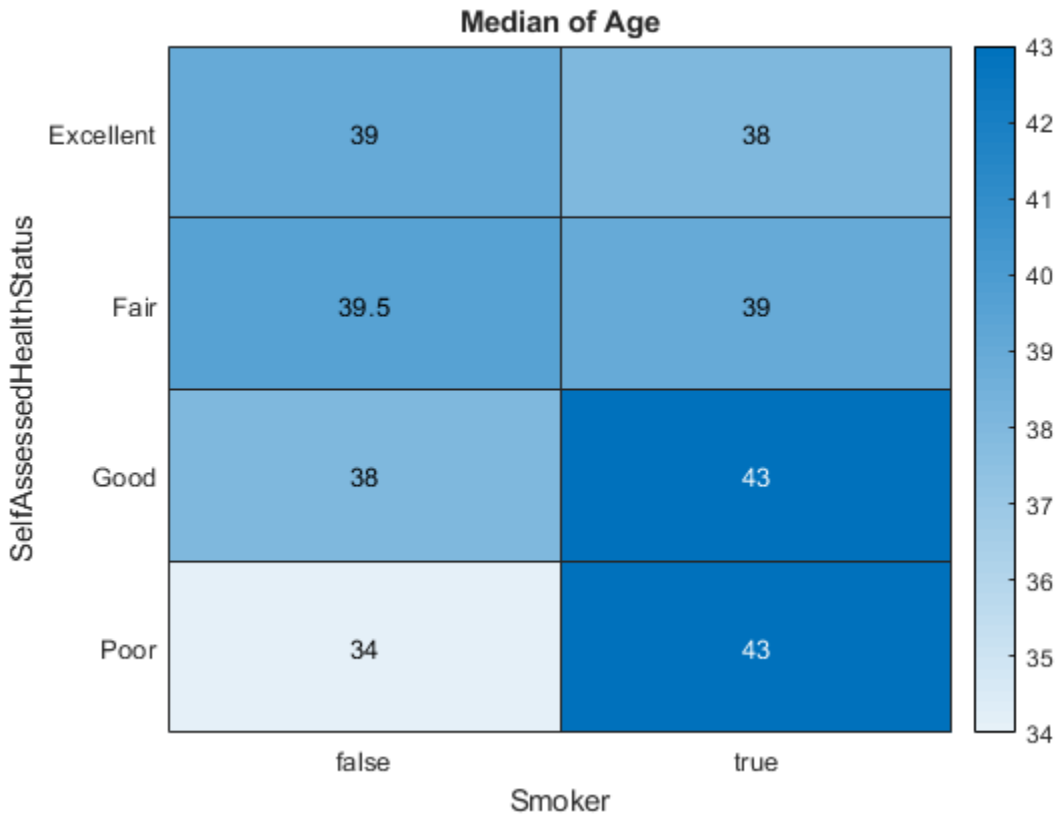


Specify Calculation Method for Color Data

Create a heatmap and specify the table variable and calculation method to use when determining the heatmap cell colors.

Load the patients data set and create a heatmap from the data. Color each cell using the median age of patients with a particular pair of `Smoker` and `SelfAssessedHealthStatus` values. Specify the `ColorVariable` option as `'Age'` and the `ColorMethod` option as `'median'`.

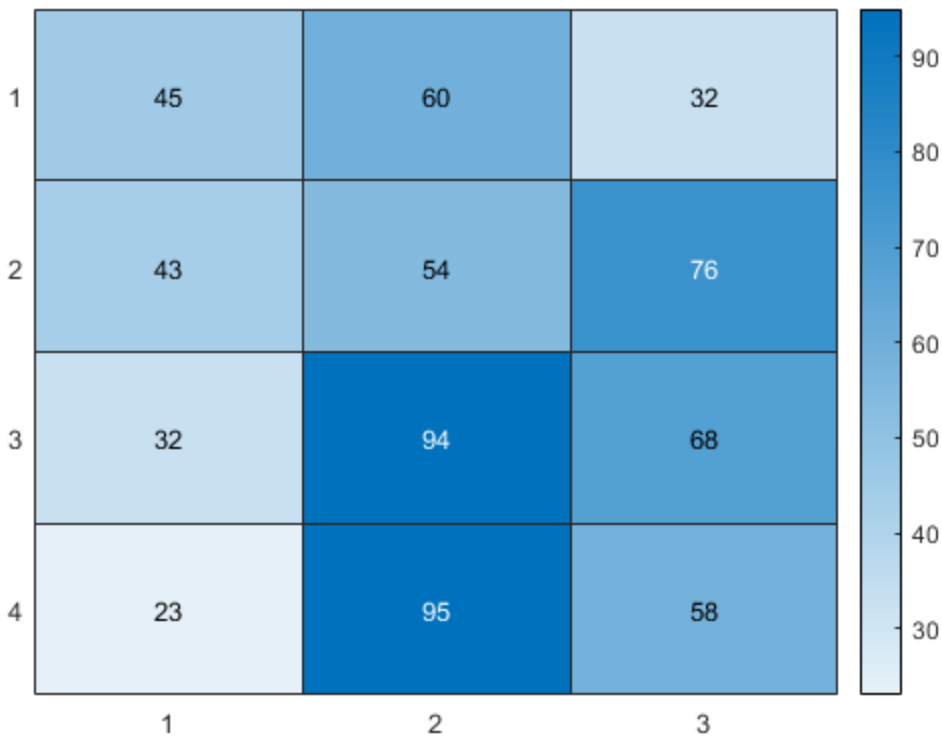
```
load patients
tbl = table(LastName,Age,Gender,SelfAssessedHealthStatus,...
    Smoker,Weight,Location);
h = heatmap(tbl,'Smoker','SelfAssessedHealthStatus','ColorVariable','Age','ColorMethod
```



Create Heatmap from Matrix Data

Create a matrix of data. Then create a heatmap of the matrix values. The default labels along the *x*-axis and *y*-axis appear as 1, 2, 3, and so on.

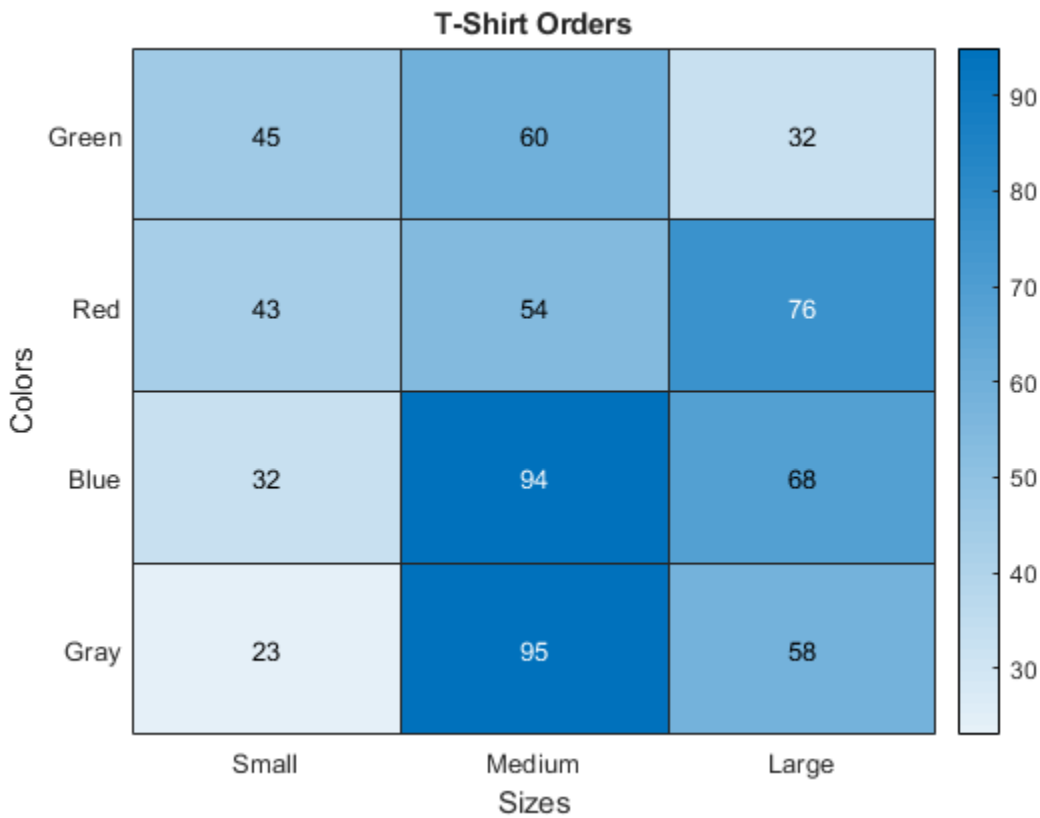
```
cdata = [45 60 32; 43 54 76; 32 94 68; 23 95 58];
h = heatmap(cdata);
```



Create Heatmap from Matrix Data Using Custom Axis Labels

Create a matrix of data. Then create a heatmap of the matrix values. Use custom labels along the *x*-axis and *y*-axis by specifying the first two input arguments as the labels you want. Specify the title and axis labels by setting properties of the `HeatmapChart` object.

```
cdata = [45 60 32; 43 54 76; 32 94 68; 23 95 58];
xvalues = {'Small','Medium','Large'};
yvalues = {'Green','Red','Blue','Gray'};
h = heatmap(xvalues,yvalues,cdata);

h.Title = 'T-Shirt Orders';
h.XLabel = 'Sizes';
h.YLabel = 'Colors';
```



Normalize Colors Along Each Row or Column

Create a heatmap and normalize the colors along each column or row by setting the `ColorScaling` property.

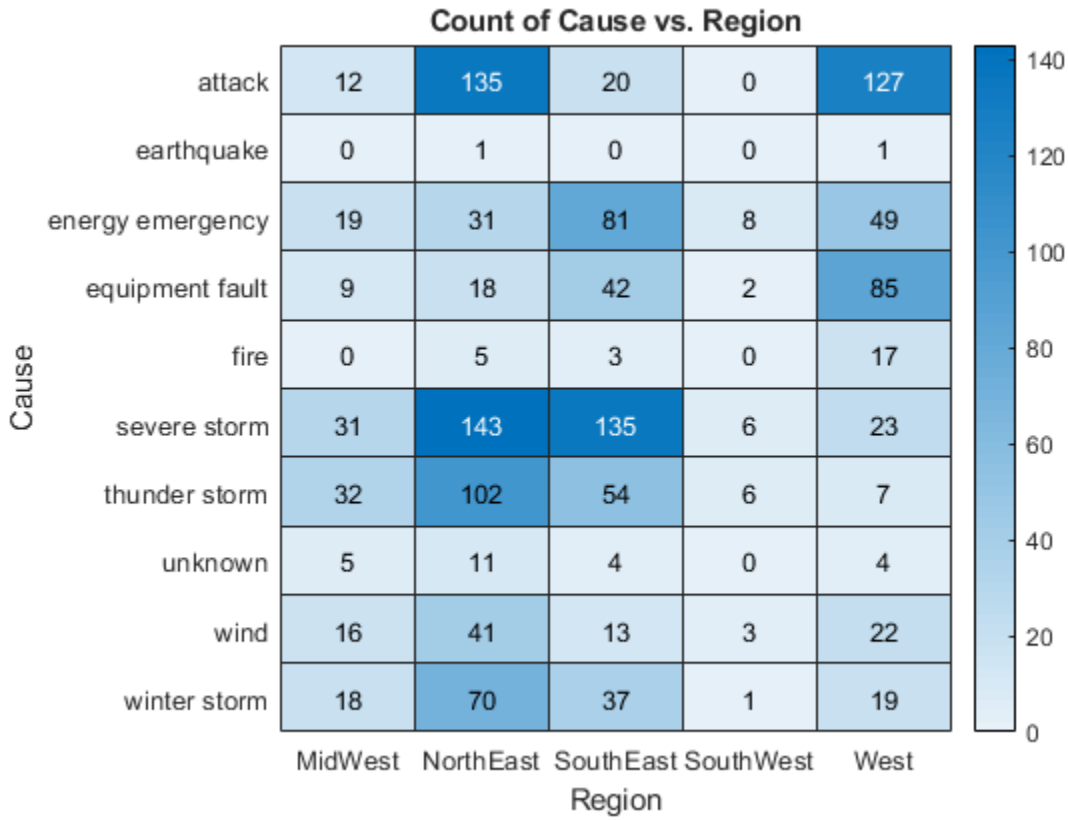
Read the sample file `outages.csv` into a table. The sample file contains data representing electric utility outages in the Unites States. The table contains six columns: `Region`, `OutageTime`, `Loss`, `Customers`, `RestorationTime`, and `Cause`. Display the first five rows of each column.

```
T = readtable('outages.csv');
T(1:5,:)
```

```
ans=5×6 table
       Region           OutageTime        Loss     Customers     RestorationTime
    _____________    ________________    ______    __________    ________________    _

    {'SouthWest'}    2002-02-01 12:18    458.98    1.8202e+06    2002-02-07 16:50    {
    {'SouthEast'}    2003-01-23 00:49    530.14    2.1204e+05                 NaT    {
    {'SouthEast'}    2003-02-07 21:15     289.4    1.4294e+05    2003-02-17 08:14    {
    {'West'     }    2004-04-06 05:44    434.81    3.4037e+05    2004-04-06 06:10    {
    {'MidWest'  }    2002-03-16 06:18    186.44    2.1275e+05    2002-03-18 23:23    {
```

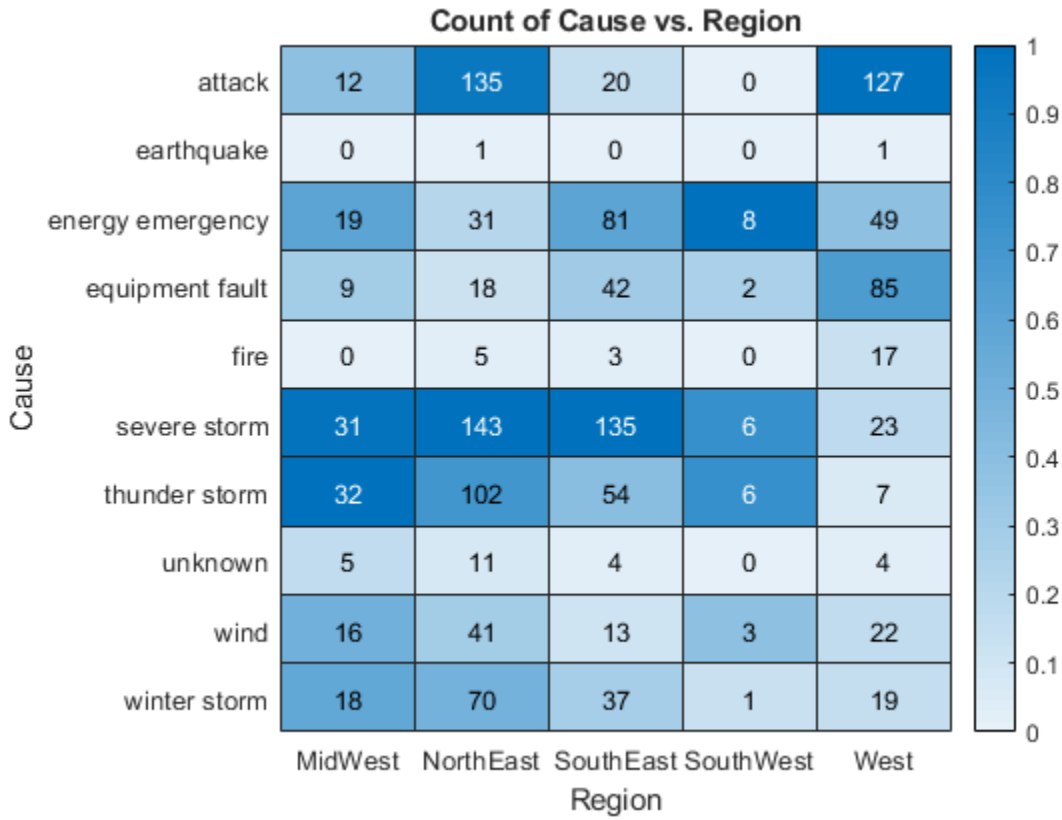
Create a heatmap that shows the different regions along the *x*-axis and the different outage causes along the *y*-axis. In each cell, show how many times each region experienced a power outage due to a particular cause.

```
h = heatmap(T,'Region','Cause');
```



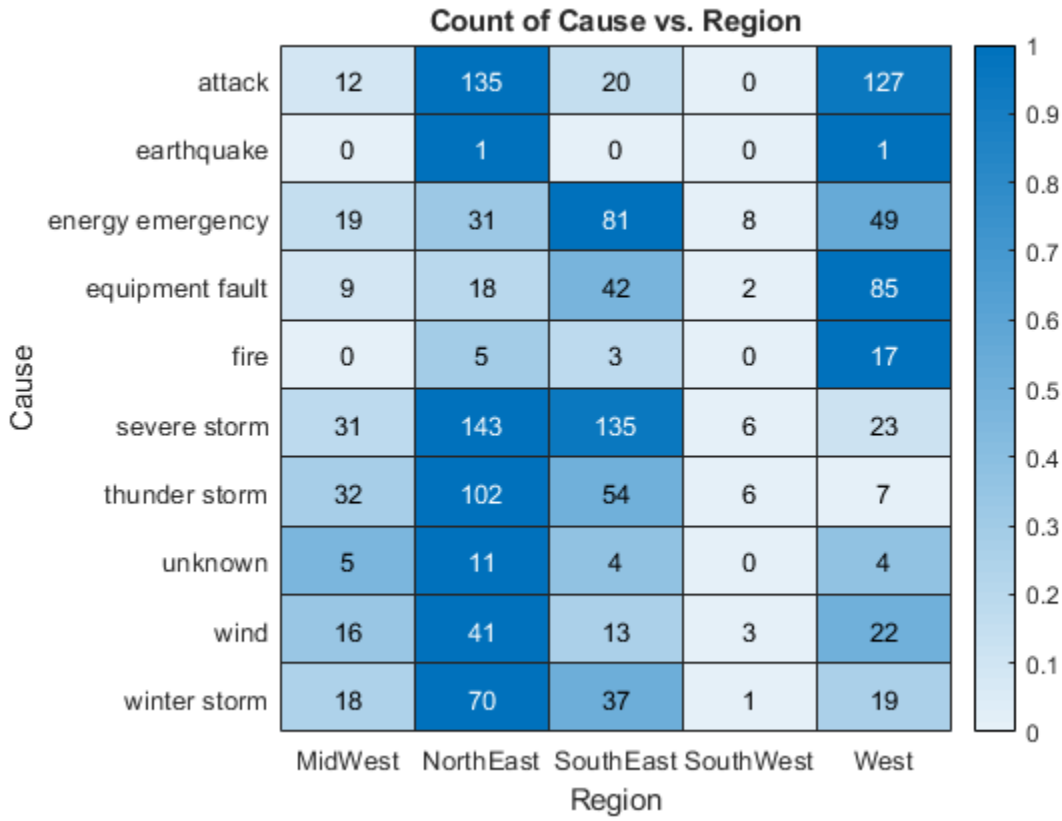
Normalize the colors along each column. The smallest value in each column maps to the first color in the colormap and the largest value maps to the last color. The last color indicates the cause that caused the most power outages for each region.

```
h.ColorScaling = 'scaledcolumns';
```



Normalize the colors along each row instead. The smallest value in each row maps to the first color in the colormap and the largest value maps to the last color. The last color indicates the region that experienced the most power outages due to each cause.

```
h.ColorScaling = 'scaledrows';
```



Compute Color Data Using Custom Aggregation Method

Create a heatmap and color the cells using data computed with a custom aggregation method. Use the `accumarray` function to compute the color data.

Read the sample file `Temperature.csv` into a table. The file contains three columns: `Month`, `Year`, and `TemperatureF`.

```
tbl = readtable(fullfile(matlabroot,'examples','graphics','TemperatureData.csv'));
```

Create categorical arrays from the `Month` and `Year` columns of the table. Then determine the unique months and years to use as labels along the *x*-axis and *y*-axis.

```
months = categorical(tbl.Month);
years = categorical(tbl.Year);
xlabels = categories(months);
ylabels = categories(years);
```

Determine the final size of the resulting color data based on the number of unique months and years.

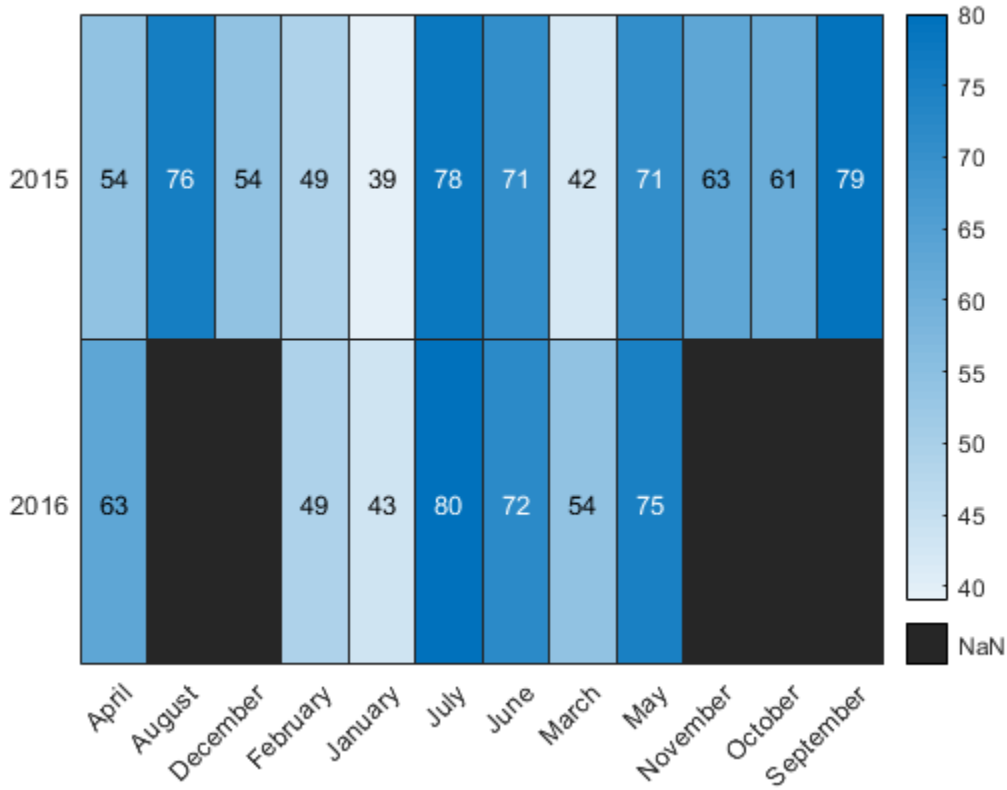
```
nummonths = numel(xlabels);
numyears = numel(ylabels);
```

Convert the categorical `months` and `years` arrays into numeric indices to use with the `accumarray` function. Compute the color data as the maximum temperature for each month and year combination using the `accumarray` function. Use `NaN` for missing month and year combinations.

```
x = double(months);
y = double(years);
temps = tbl.TemperatureF;
cdata = accumarray([y,x],temps,[numyears,nummonths],@max,NaN);
```

Create the heatmap. Label the *x*-axis and *y*-axis with the months and years, respectively. Color the heatmap cells using the computed matrix data.

```
h = heatmap(xlabels,ylabels,cdata);
```



Note: You can use the `reordercats` function for categorical arrays to reorder the axis labels.

Input Arguments

`tbl` — Source table
table | timetable

Source table, specified as a table or timetable.

You can create a table from workspace variables using the `table` function, or you can import data as a table using the `readtable` function. You can create a timetable from workspace variables using the `timetable` function.

The `SourceTable` property of the `HeatmapChart` object stores the source table.

`xvar` — Table variable for *x*-axis
character vector | string scalar | numeric scalar | logical vector

Table variable for *x*-axis, specified in one of these forms:

- Character vector or string scalar indicating one of the variable names. For example, `heatmap(tbl,'HealthStatus','Gender')` selects the variable named `'HealthStatus'` for the *x*-axis.
- Numeric scalar indicating the table variable index. For example, `heatmap(tbl,2,3)` selects the second variable in the table for the *x*-axis.
- Logical vector containing one `true` element.

The values associated with your table variable must be grouped into a finite set of discrete categories that the `categorical` function accepts. If the values are not grouped into a finite set of categories, use the `discretize` function to group them.

The labels that appear along the *x*-axis are in alphabetical order. You can customize the labels using categorical arrays. For an example, see “Create Heatmap from Tabular Data”.

The `XVariable` property of the `HeatmapChart` object stores the selected variable.

`yvar` — Table variable for *y*-axis
character vector | string scalar | numeric scalar | logical vector

Table variable for *y*-axis, specified in one of these forms:

- Character vector or string scalar of one of the variable names. For example, `heatmap(tbl,'HealthStatus','Gender')` selects the variable named `'HealthStatus'` for the *y*-axis.
- Numeric scalar indicating the table variable index. For example, `heatmap(tbl,2,3)` selects the third variable in the table for the *y*-axis.
- Logical vector containing one `true` element.

The values associated with your table variable must be grouped into a finite set of discrete categories that the `categorical` function accepts. If the values are not grouped into a finite set of categories, use the `discretize` function to group them.

The labels that appear along the *y*-axis are in alphabetical order. You can customize the labels using categorical arrays. For an example, see “Create Heatmap from Tabular Data”.

The `YVariable` property of the `HeatmapChart` object stores the selected variable.

`cvar` — Table variable for color data
character vector | string scalar | numeric scalar | logical vector

Table variable for color data, specified in one of these forms:

- A character vector or string scalar of one of the variable names. For example, `heatmap(__,'ColorVariable,'HealthStatus')` selects the variable named `'HealthStatus'` for the *y*-axis.
- A numeric scalar indicating the table variable index. For example, `heatmap(__,'ColorVariable',1)` selects the third variable in the table for the *y*-axis.
- A logical vector containing one `true` element.

The values associated with your table variable must be of a numeric type or `logical`.

The property value is empty `[]` when using matrix data. The `ColorVariable` property of the `HeatmapChart` object stores the selected variable. The `ColorData` property populates with data based on the selected variable.

By default, `heatmap` calculates the color data based on a mean aggregation. To change the calculation method, set the `ColorMethod` property.

`cdata` — Color data
matrix

Color data for the heatmap cells, specified as a matrix.

The `ColorData` property of the `HeatmapChart` object stores the values.

Example: `[40 24 68; 68 37 58; 49 23 46]`

`xvalues` — Values appearing along *x*-axis
categorical array | string array | numeric array | cell array of character vectors

Values appearing along the *x*-axis, specified as a categorical array, string array, numeric array, or cell array of character vectors.

Example: `{'small','medium','large'}`

Example: `categorical({'small','medium','large'})`

`yvalues` — Values appearing along *y*-axis
categorical array | string array | numeric array | cell array of character vectors

Values appearing along *y*-axis, specified as a categorical array, string array, numeric array, or cell array of character vectors.

Example: `{'small','medium','large'}`

Example: `categorical({'small','medium','large'})`

`parent` — Parent container
`Figure` object | `Panel` object | `Tab` object | `TiledChartLayout` object

Parent container, specified as a `Figure`, `Panel`, `Tab`, or `TiledChartLayout` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
`heatmap(tbl,xvar,yvar,'ColorVariable','Temperatures','ColorMethod','median')`

Note The properties listed here are only a subset. For a complete list, see HeatmapChart.

`Title` — Chart title
character vector | cell array of character vectors | string array | categorical array

Chart title, specified as a character vector, cell array of character vectors, string array, or categorical array.

Example: `h = heatmap(rand(10,10),'Title','My Title Text')`

Example: `h.Title = 'My Title Text';`

To create a multiline title, specify a cell array of character vectors or a string array. Each element in the array corresponds to a line of text.

Example: `h.Title = {'My','Title'};`

If you specify the title as a categorical array, MATLAB uses the values in the array, not the categories.

If you create the chart using tabular data, the default chart has an autogenerated title. If you do not want a title, specify `''`.

By default, heatmaps support a subset of TeX markup for the text you specify. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text. If you want a TeX markup character in regular text, such as an underscore (_), then insert a backslash (\) before the character you want to include. The backslash is the TeX escape character. For more information, see the "Interpreter" on page 1-0 property of the text object.

`ColorMethod` — Method to calculate color data
`'count'` | `'mean'` | `'median'` | `'sum'` | `'none'`

Method to calculate the color data values (stored in `ColorData`), specified as `'count'`, `'mean'`, `'median'`, `'sum'`, or `'none'`.

If you do not want to use a third variable from the table for the color data, then specify the method in this table.

Method	Description
`'count'`	Count the number of times each pair of *x* and *y* values appears in the source table. The heatmap does not use the `ColorVariable` property. This value is the default value when you are using tabular data and do not specify the `ColorVariable` parameter.

If you want to use a third variable from the table for the color data, then set the `ColorVariable` property to the variable you want and specify the `ColorMethod` property as one of the methods listed in this table. For each pair of *x* and *y* values, the

methods use the corresponding values in the `ColorVariable` column of the source table to calculate the data.

Method	Description
`'mean'`	Calculate the average value. This value is the default value when you specify the `ColorVariable` property.
`'median'`	Calculate the median value.
`'sum'`	Sum the values.
`'none'`	Use the value exactly. The table cannot contain more than one instance of each pair of *x* and *y* values. This value is the default value when using matrix data.

If you want to compute your own matrix of aggregated data, use the `accumarray` function. Specify the matrix as input to the `heatmap` function.

Example: `h = heatmap(__,'ColorMethod','median')`

Example: `h.ColorMethod = 'median'`

Tips

- To interactively explore the data in your heatmap, use these options.
  - Zoom — Use the scroll wheel or the **+** and **-** keys to zoom.
  - Pan — Click and drag the heatmap or use the arrow keys to pan across the rows or columns.
  - Data tips — Hover over the heatmap to display a data tip.
  - Rearrange rows and columns — Click and drag a row or column label to move it to a different position.
  - Sort values — Click the icon that appears when you hover over the row or column label. Click once to sort the values in ascending order, twice to sort the values in descending order, and a third time to reset the order.

Compatibility Considerations

Heatmaps interpret text using TeX markup

Behavior changed in R2019a

Starting in R2019a, heatmaps interpret text using TeX markup instead of displaying the literal characters. If you want to use a TeX markup character in regular text, such as an underscore (_), then insert a backslash (\) before the character you want to include. The backslash is the TeX escape character. For more information on using TeX markup, see the "Interpreter" on page 1-0 property of the text object.

See Also

Functions

`categorical` | `readtable` | `sortx` | `sorty` | `table`

Properties

HeatmapChart

Topics

"Create Heatmap from Tabular Data"
"Access Data in Tables"

Introduced in R2017a

HeatmapChart Properties

Heatmap chart appearance and behavior

Description

`HeatmapChart` properties control the appearance and behavior of a `HeatmapChart` object. By changing property values, you can modify certain aspects of the heatmap chart. For example, you can add a title:

```
h = heatmap([1 3 5; 2 4 6]);
h.Title = 'My Heatmap Title';
```

Properties

Labels

`Title` — Chart title

character vector | cell array of character vectors | string array | categorical array

Chart title, specified as a character vector, cell array of character vectors, string array, or categorical array.

Example: `h = heatmap(rand(10,10),'Title','My Title Text')`

Example: `h.Title = 'My Title Text';`

To create a multiline title, specify a cell array of character vectors or a string array. Each element in the array corresponds to a line of text.

Example: `h.Title = {'My','Title'};`

If you specify the title as a categorical array, MATLAB uses the values in the array, not the categories.

If you create the chart using tabular data, the default chart has an autogenerated title. If you do not want a title, specify `''`.

By default, heatmaps support a subset of TeX markup for the text you specify. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include

special characters in the text. If you want a TeX markup character in regular text, such as an underscore (_), then insert a backslash (\) before the character you want to include. The backslash is the TeX escape character. For more information, see the “Interpreter” on page 1-0 property of the text object.

`XLabel` — Label for *x*-axis

character vector | cell array of character vectors | string array | categorical array

Label for the *x*-axis, specified as a character vector, cell array of character vectors, string array, or categorical array.

Example: `h = heatmap(rand(10,10),'XLabel','My Label')`

Example: `h.XLabel = 'My Label';`

To create a multiline label, specify a cell array of character vectors or a string array. Each element in the array corresponds to a line of text.

Example: `h.XLabel = {'My','Label'};`

If you specify the label as a categorical array, MATLAB uses the values in the array, not the categories.

If you create the chart using tabular data, the default chart has an autogenerated label. If you do not want a label, specify `''`.

By default, heatmaps support a subset of TeX markup for the text you specify. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text. If you want a TeX markup character in regular text, such as an underscore (_), then insert a backslash (\) before the character you want to include. The backslash is the TeX escape character. For more information, see the “Interpreter” on page 1-0 property of the text object.

`YLabel` — Label for *y*-axis

character vector | cell array of character vectors | string array | categorical array

Label for the *y*-axis, specified as a character vector, cell array of character vectors, string array, or categorical array.

Example: `h = heatmap(rand(10,10),'YLabel','My Label')`

Example: `h.YLabel = 'My Label';`

To create a multiline label, specify a cell array of character vectors or a string array. Each element in the array corresponds to a line of text.

Example: `h.YLabel = {'My','Label'};`

If you specify the label as a categorical array, MATLAB uses the values in the array, not the categories.

If you create the chart using tabular data, the default chart has an autogenerated label. If you do not want a label, specify `''`.

By default, heatmaps support a subset of TeX markup for the text you specify. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text. If you want a TeX markup character in regular text, such as an underscore (_), then insert a backslash (\) before the character you want to include. The backslash is the TeX escape character. For more information, see the "Interpreter" on page 1-0 property of the text object.

`MissingDataLabel` — Label for missing data icon

`'NaN'` (default) | character vector

Label for missing data icon that displays below the colorbar, specified as a character vector. If you do not want a label to display, use an empty character vector `''`.

By default, heatmaps support a subset of TeX markup for the text you specify. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text. If you want a TeX markup character in regular text, such as an underscore (_), then insert a backslash (\) before the character you want to include. The backslash is the TeX escape character. For more information, see the "Interpreter" on page 1-0 property of the text object.

Example: `h = heatmap(__,'MissingDataLabel','No data')`

Example: `h.MissingDataLabel = 'No data'`

Color and Styling

`Colormap` — Colormap for coloring heatmap cells

predefined colormap name | `m`-by-3 array of RGB triplets

Colormap for coloring heatmap cells, specified as a predefined colormap name or an `m`-by-3 array of RGB (red, green, blue) triplets that define `m` individual colors. You can specify one of the predefined colormaps, or you can create a custom one.

- Predefined colormaps — Specify the colormap name in command form, such as `parula` or `summer`. For a full list of options, see `colormap`.

- Custom colormap — Specify an m-by-3 array of RGB triplets.

Example: `h = heatmap(__,'Colormap',summer)`

Example: `h.Colormap = parula`

`ColorMethod` — Method to calculate color data
`'count'` | `'mean'` | `'median'` | `'sum'` | `'none'`

Method to calculate the color data values (stored in `ColorData`), specified as `'count'`, `'mean'`, `'median'`, `'sum'`, or `'none'`.

If you do not want to use a third variable from the table for the color data, then specify the method in this table.

Method	Description
`'count'`	Count the number of times each pair of *x* and *y* values appears in the source table. The heatmap does not use the `ColorVariable` property. This value is the default value when you are using tabular data and do not specify the `ColorVariable` parameter.

If you want to use a third variable from the table for the color data, then set the `ColorVariable` property to the variable you want and specify the `ColorMethod` property as one of the methods listed in this table. For each pair of *x* and *y* values, the methods use the corresponding values in the `ColorVariable` column of the source table to calculate the data.

Method	Description
`'mean'`	Calculate the average value. This value is the default value when you specify the `ColorVariable` property.
`'median'`	Calculate the median value.
`'sum'`	Sum the values.

Method	Description
`'none'`	Use the value exactly. The table cannot contain more than one instance of each pair of *x* and *y* values. This value is the default value when using matrix data.

If you want to compute your own matrix of aggregated data, use the `accumarray` function. Specify the matrix as input to the `heatmap` function.

Example: `h = heatmap(__,'ColorMethod','median')`

Example: `h.ColorMethod = 'median'`

`ColorScaling` — Mapping of color data to colormap colors
`'scaled'` (default) | `'scaledcolumns'` | `'scaledrows'` | `'log'`

Mapping of color data to colormap colors, specified as one of these values:

- `'scaled'` — Map values in the `ColorData` property that are less than or equal to the minimum color limit to the first color in the colormap. Map values greater than or equal to the maximum color limit to the last color. The `ColorLimits` property contains the color limit values.
- `'scaledcolumns'` — Normalize each column in the `ColorData` property to values from 0 through 1. Map the smallest value to the first color in the colormap and the largest value to the last color. If all the values in a column are the same, then the heatmap uses the middle color of the colormap.
- `'scaledrows'` — Normalize each row in the `ColorData` property to values from 0 to 1. Map the smallest value to the first color in the colormap and the largest value to the last color. If all the values in a row are the same, then the heatmap uses the middle color of the colormap.
- `'log'` — Calculate the log of each value in the `ColorData` property before mapping the values to colors in the colormap. Negative values appear as missing data. However, if all the values are negative, then this option uses `-log(-value)`.

The heatmap ignores `NaN`, `Inf`, and `-Inf` values when determining the color scaling.

When `ColorScaling` is set to `'scaledcolumns'` or `'scaledrows'`, the default tick labels on the colorbar range from 0 to 1. The smallest value within a column or row of the heatmap chart maps to 0 on the colorbar. The largest value within a column or row maps to 1 on the colorbar.

Example: `h = heatmap(__,'ColorScaling','scaledcolumns')`

Example: `h.ColorScaling = 'scaledcolumns'`

`ColorLimits` — Color limits
two-element vector

Color limits, specified as a two-element vector of the form `[min max]`. The color limits indicate the color data values that map to the first and last colors in the colormap.

Note The default values of `min` and `max` reflect the range of your data after the `ColorScaling` option is applied.

Example: `h = heatmap(__,'ColorLimits',[0 10])`

Example: `h.ColorLimits = [0 10]`

`MissingDataColor` — Color for cells with no data value
`[0.1500 0.1500 0.1500]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color for cells with no data value, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `h = heatmap(__,'MissingDataColor',[0.8 0.8 0.8])`

Example: `h.MissingDataColor = [0.8 0.8 0.8]`

`ColorbarVisible` — Display of colorbar
`'on'` (default) | `'off'`

Display of colorbar, specified as `'on'` or `'off'`.

Example: `h = heatmap(__,'ColorbarVisible','off')`

Example: `h.ColorbarVisible = 'off'`

`GridVisible` — Display of grid lines
`'on'` (default) | `'off'`

Display of grid lines, specified as `'on'` or `'off'`.

Example: `h = heatmap(__,'GridVisible','off')`

Example: `h.GridVisible = 'off'`

`CellLabelColor` — Text color for data labels

`'auto'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Text color for data labels, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'auto'` chooses an appropriate text color, depending on the color of each heatmap cell. If you do not want the labels to display, specify `'none'`.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

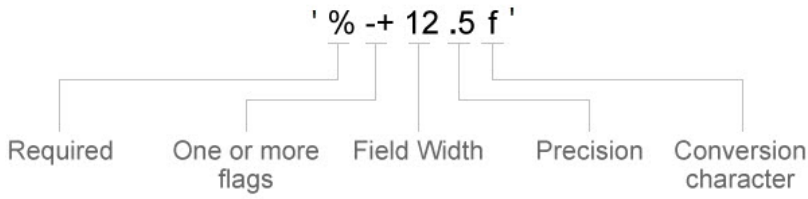
Example: `h = heatmap(__,'CellLabelColor','blue')`

Example: `h.CellLabelColor = 'blue'`

`CellLabelFormat` — Format for data labels

`'%0.4g'` (default) | character vector of format options

Format for data labels, specified as a character vector of format options.



Most elements of the character vector are optional, except the percent sign and conversion character. Construct the character vector in this order:

- One or more flags — Options. For example, add a plus sign before positive values. For a full list of options, see the table of Optional Flags.
- Field width — Minimum number of characters to print in the tick label. Specify the field width as an integer value. If the number of significant digits in the tick value is smaller than the field width, then the label is padded with spaces.

- Precision — Number of digits to the right of the decimal point or the number of significant digits, depending on the conversion character. Specify the precision as an integer value.
- Conversion character — Value type. For a full list of options, see the table of Conversion Characters. If you specify a conversion that does not fit the data, then MATLAB overrides the specified conversion, and uses `%e`.

Also, you can specify literal text at the beginning or end of the format. To print a single quotation mark, use `''`. To print a percent character, use `%%`.

Example: `h.CellLabelFormat = '%.2f'` displays the values using fixed-point notation with two decimal places.

Example: `h.CellLabelFormat ='$%.2f'` displays a dollar sign before each value.

Example: `h.CellLabelFormat = '%.2f lbs'` displays `lbs` after each value.

Optional Flags

Identifier	Description	Example of Numeric Format
`+`	Print the sign character (+) for positive values, such as `'+100'`.	`'%+4.4g'`
`0`	Pad the field width with leading zeros instead of spaces, such as `'0100'`.	`'%04.4g'`
`-`	Left justify, which pads the end of the value with spaces instead of the beginning. For example, if the field width is 4, then this flag formats the label as `'100 '` instead of `' 100'`.	`'%-4.4g'`
`#`	For the `%f`, `%e`, and `%g` conversion characters, print the decimal point even when the precision is 0, such as `'100.'`. For `%g`, do not remove trailing zeros.	`'%#4.4g'`

Conversion Characters

Identifier	Description	Example
`d` or `i`	Signed integer with base 10. The precision value indicates the number of significant digits.	`'%.4d'` displays π as `0003`.
`f`	Fixed-point notation. The precision value indicates the number of decimal places.	`'%.4f'` displays π as `3.1416`.
`e`	Exponential notation. The precision value indicates the number of decimal places.	`'%.4e'` displays π as `3.1416x10`^{0}.
`g`	The more compact version of `e` or `f`, with no trailing zeros. The precision value indicates the maximum number of decimal places.	`'%.4g'` displays π as `3.1416`.

`FontColor` — Text color for title, axis labels, and tick labels

`[0.1500 0.1500 0.1500]` (default) | color name | RGB triplet

Text color for title, axis labels, and tick labels, specified as a color name or an RGB triplet.

Example: `h = heatmap(__,'FontColor','blue')`

Example: `h.FontColor = 'blue'`

Font

`FontName` — Font name

system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

Example: `h = heatmap(__,'FontName','Cambria')`

Example: `h.FontName = 'Cambria'`

`FontSize` — Font size

scalar numeric value

Font size used for the title, axis labels, and cell labels, specified as a scalar value. The default font depends on the specific operating system and locale.

The title and axis labels use a slightly larger font size (scaled up by 10%). If there is not enough room to display the text within each cell, then the text might use a smaller font size or the text might not appear.

Example: `h = heatmap(__,'FontSize',12)`

Example: `h.FontSize = 12`

Table Data

`SourceTable` — Source table

table | timetable

Source table, specified as a table or a timetable.

You can create a table from workspace variables using the `table` function, or you can import data as a table using the `readtable` function. You can create a timetable from workspace variables using the `timetable` function.

Note The property is ignored and read-only when you use matrix data.

`XVariable` — Table variable for *x*-axis

character vector | string | numeric scalar | logical vector

Table variable for *x*-axis, specified in one of these forms:

- Character vector or string indicating one of the variable names
- Numeric scalar indicating the table variable index
- Logical vector containing one `true` element

The values associated with your table variable must be grouped into a finite set of discrete categories that the `categorical` function accepts. If the values are not grouped into a finite set of categories, use the `discretize` function to group them.

If you set the `XVariable` property value, then the `XData` and `XDisplayData` properties automatically update to appropriate values.

The labels that appear along the *x*-axis are in alphabetical order. You can customize the labels using `XDisplayData`, `yDisplayData`, or categorical arrays. For an example, see "Create Heatmap from Tabular Data".

Note The property is ignored and read-only when you use matrix data.

Example: `h.XVariable = 'Location'` specifies the variable named `'Location'`.

`YVariable` — Table variable for *y*-axis

character vector | string | numeric scalar | logical vector

Table variable for *y*-axis, specified in one of these forms:

- Character vector or string indicating one of the variable names
- Numeric scalar indicating the table variable index
- Logical vector containing one `true` element

The values associated with your table variable must be grouped into a finite set of discrete categories that the `categorical` function accepts. If the values are not grouped into a finite set of categories, use the `discretize` function to group them.

If you set the `YVariable` property value, then the `YData` and `YDisplayData` properties automatically update to appropriate values.

The labels that appear along the *y*-axis are in alphabetical order. You can customize the labels using categorical arrays. For an example, see "Create Heatmap from Tabular Data".

Note The property is ignored and read-only when you use matrix data.

Example: `h.YVariable = 'Location'` specifies the variable named `'Location'`.

`ColorVariable` — Table variable for color data

character vector | string | numeric scalar | logical vector

Table variable for color data, specified in one of these forms:

- Character vector or string indicating one of the variable names.

- Numeric scalar indicating the table variable index.
- Logical vector containing one `true` element.

The values associated with your table variable must be of a numeric type or `logical`.

When you specify the color variable, MATLAB updates the `ColorData` property values. Also, the `ColorMethod` property changes to `'mean'`, unless you previously specified a different value.

Note This property is ignored and read-only when you use matrix data. It is also ignored when the `ColorMethod` property is set to `'count'`.

Example: `h = heatmap(__,'ColorVariable','Temperature')`

Example: `h.ColorVariable = 'Temperature'`

Matrix Data

`ColorData` — Data to color each heatmap cell

matrix of numeric values

Data to color each heatmap cell, specified as a matrix of numeric values.

If you are using tabular data, you cannot set this property. The `ColorData` values automatically populate based on the table variable you select with the `ColorVariable` property.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`XData` — *x* values

cell array of character vectors | string array | categorical array

x values associated with the color data matrix columns, specified as a cell array of character vectors, a string array, or a categorical array. The `XDisplayData` property controls the order in which the values appear along the *x*-axis in the chart.

If you change the `XData` property value, then the `XDisplayData` property automatically updates to appropriate values.

If you are using tabular data, you cannot set this property. The `XData` values automatically populate based on the table variable you select with the `XVariable` property.

Example: `h.XData = {'small','large','medium'}`

Data Types: `char` | `string` | `cell` | `categorical`

`YData` — *y* values
cell array of character vectors | string array | categorical array

y values associated with the color data matrix rows, specified as a cell array of character vectors, a string array, or a categorical array. The `YDisplayData` property controls the order in which the values appear along the *y*-axis in the chart.

If you change the `YData` property value, then the `YDisplayData` property automatically updates to appropriate values.

If you are using tabular data, you cannot set this property. The `YData` values automatically populate based on the table variable you select with the `YVariable` property.

Example: `h.YData = {'small','large','medium'}`

Data Types: `char` | `string` | `cell` | `categorical`

Displayed Data

`ColorDisplayData` — Sorted color data
matrix

This property is read-only.

Sorted color data, returned as a matrix of values as they appear in the heatmap chart. The values are sorted based on the `XDisplayData` and `YDisplayData` properties.

Data Types: `double`

`XDisplayData` — Display order of *x*-axis values
cell array of character vectors | string array | categorical array

Display order of *x*-axis values, specified as a cell array of character vectors, a string array, or a categorical array. You can set this property to a subset, superset, or permutation of the values in `XData`. By default, the values are the same as the `XData` values.

If you specify a value that is not in `XData`, then the heatmap shows a row of either `NaN` values or zeros.

- If the `ColorMethod` property is `'sum'` or `'count'`, then the heatmap shows a row of zeros.
- If the `ColorMethod` property is `'mean'`, `'median'`, or `'none'`, then the heatmap shows a row of `NaN` values.

If you want to specify the `XDisplayData` as a name-value pair during object creation, you must specify the `XDisplayData` property before specifying the `XDisplayLabels` or `XLimits` properties. Use the `XDisplayLabels` property to change the displayed text.

Example: `h.XDisplayData = {'small','medium','large'}`

Data Types: `char` | `string` | `cell` | `categorical`

`YDisplayData` — Display order of *y*-axis values
cell array of character vectors | string array | categorical array

Display order of *y*-axis values, specified as a cell array of character vectors, a string array, or a categorical array. You can set this property to a subset, superset, or permutation of the values in `YData`. By default, the values are the same as the `YData` values.

If you specify a value that is not in `YData`, then the heatmap shows a row of either `NaN` values or zeros.

- If the `ColorMethod` property is `'sum'` or `'count'`, then the heatmap shows a row of zeros.
- If the `ColorMethod` property is `'mean'`, `'median'`, or `'none'`, then the heatmap shows a row of `NaN` values.

If you want to specify `YDisplayData` as a name-value pair during object creation, you must specify the `YDisplayData` property before specifying the `YDisplayLabels` or `YLimits` properties. Use the `YDisplayLabels` property to change the displayed text.

Example: `h.YDisplayData = {'small','medium','large'}`

Data Types: `char` | `string` | `cell` | `categorical`

`XDisplayLabels` — Labels for *x*-axis values
cell array of character vectors | string array | categorical array

Labels for the *x*-axis values, specified as a cell array of character vectors, string array, or categorical array. The array must be a column vector the same size as the `XDisplayData`

vector. Specify one label for each value in `XDisplayData`. By default, the values are the same as the `XDisplayData` values. When you specify this property as a categorical array, MATLAB uses the values in the array, not the categories.

If you add a value, delete a value, or rearrange the values in the `XDisplayData` property, then this property updates accordingly to maintain the pairings of values and labels. If you want to specify both `XDisplayLabels` and `XDisplayData` as name-value pairs during object creation, then specify the `XDisplayData` property first.

By default, heatmaps support a subset of TeX markup for the text you specify. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text. If you want a TeX markup character in regular text, such as an underscore (_), then insert a backslash (\) before the character you want to include. The backslash is the TeX escape character. For more information, see the “Interpreter” on page 1-0 property of the text object.

Example: `h.XDisplayLabels = {'SM','MED','LG'}`

Data Types: `char` | `string` | `cell` | `categorical`

`YDisplayLabels` — Labels for *y*-axis values

cell array of character vectors | string array | categorical array

Labels for the *y*-axis values, specified as a cell array of character vectors, string array, or categorical array. The array must be a column vector the same size as the `YDisplayData` vector. Specify one label for each value in `YDisplayData`. By default, the values are the same as the `YDisplayData` values. When you specify this property as a categorical array, MATLAB uses the values in the array, not the categories.

If you add a value, delete a value, or rearrange the values in the `YDisplayData` property, then this property updates accordingly to maintain the pairings of values and labels. If you want to specify both `YDisplayLabels` and `YDisplayData` as name-value pairs during object creation, then specify the `YDisplayData` property first.

By default, heatmaps support a subset of TeX markup for the text you specify. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text. If you want a TeX markup character in regular text, such as an underscore (_), then insert a backslash (\) before the character you want to include. The backslash is the TeX escape character. For more information, see the “Interpreter” on page 1-0 property of the text object.

Example: `h.YDisplayLabels = {'SM','MED','LG'}`

Data Types: `char` | `string` | `cell` | `categorical`

`XLimits` — *x*-axis limits
two-element row vector

x-axis limits, specified as a two-element row vector of values from `XDisplayData`.

If you want to specify both `XLimits` and `XDisplayData` as name-value pairs during object creation, then specify the `XDisplayData` property first.

Example: `h.XLimits = {'small','medium'}`

Data Types: `char` | `string` | `cell` | `categorical`

`YLimits` — *y*-axis limits
two-element row vector

y-axis limits, specified as a two-element row vector of values from `YDisplayData`.

If you want to specify both `YLimits` and `YDisplayData` as name-value pairs during object creation, then specify the `YDisplayData` property first.

Example: `h.YLimits = {'small','medium'}`

Data Types: `char` | `string` | `cell` | `categorical`

Position

`ActivePositionProperty` — Position property to hold constant
`'outerposition'` | `'innerposition'`

Position property to hold constant when adding, removing, or changing decorations, specified as one of the following values:

- `'outerposition'` — The `OuterPosition` property remains constant when you add, remove, or change decorations such as a title or an axis label. If any positional adjustments are needed, MATLAB adjusts the `InnerPosition` property.
- `'innerposition'` — The `InnerPosition` property remains constant when you add, remove, or change decorations such as a title or an axis label. If any positional adjustments are needed, MATLAB adjusts the `OuterPosition` property.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`OuterPosition` — Outer size and location

`[0 0 1 1]` (default) | four-element vector

Outer size and location of the heatmap within the parent container (typically a figure, panel, or tab), specified as a four-element vector of the form `[left bottom width height]`. The outer position includes the colorbar, title, and axis labels.

- The `left` and `bottom` elements define the distance from the lower left corner of the container to the lower left corner of the heatmap.
- The `width` and `height` elements are the heatmap dimensions, which include the heatmap cells, plus a margin for the surrounding text and colorbar.

The default value of `[0 0 1 1]` covers the whole interior of the container. The units are normalized relative to the size of the container. To change the units, set the `Units` property.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`InnerPosition` — Inner size and location

four-element vector

Inner size and location of the heatmap within the parent container (typically a figure, panel, or tab) returned as a four-element vector of the form `[left bottom width height]`. The inner position does not include the colorbar, title, or axis labels.

- The `left` and `bottom` elements define the distance from the lower left corner of the container to the lower left corner of the heatmap.
- The `width` and `height` elements are the heatmap dimensions, which include only the heatmap cells.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`Position` — Inner size and location

four-element vector

Inner size and location of the heatmap within the parent container (typically a figure, panel, or tab) returned as a four-element vector of the form `[left bottom width height]`. This property is equivalent to the `InnerPosition` property.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`Units` — Position units

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'points'` | `'pixels'` | `'characters'`

Position units, specified as one of these values.

Units	Description
`'normalized'` (default)	Normalized with respect to the container, which is typically the figure or a panel. The lower left corner of the container maps to `(0,0)`, and the upper right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the default uicontrol font of the graphics root object: • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.
`'points'`	Typography points. One point equals 1/72 inch.

Units	Description
'pixels'	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.

When specifying the units as a name-value pair during object creation, you must set the `Units` property before specifying the properties that you want to use these units, such as `OuterPosition`.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `TiledChartLayoutOptions` object

Layout options, specified as a `TiledChartLayoutOptions` object. This property specifies options when the chart is a child of a tiled chart layout. Use this property to position the chart within the layout by setting the `Tile` and `TileSpan` properties on the `TiledChartLayoutOptions` object.

For example, this code places chart `c` in the third tile of a tiled chart layout.

```
c.Layout.Tile = 3;
```

To make the chart span multiple tiles, specify the `TileSpan` property as a two-element vector. For example, this chart spans `2` rows and `3` columns of tiles.

```
c.Layout.TileSpan = [2 3];
```

If the chart is not a child of a tiled chart layout (for example, if it is a child of a figure or panel), then this property is empty and has no effect.

`Visible` — State of visibility
`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the heatmap.
- `'off'` — Hide the heatmap without deleting it. You still can access the properties of an invisible `HeatmapChart` object.

Parent/Child

`Parent` — Parent container
`Figure` object | `Panel` object | `Tab` object | `TiledChartLayout` object

Parent container, specified as a `Figure`, `Panel`, `Tab`, or `TiledChartLayout` object.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'off'` | `'callback'`

Visibility of `HeatmapChart` object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. To temporarily hide the handle during the execution of that function, set the `HandleVisibility` to `'off'`.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles, regardless of their `HandleVisibility` property setting.

See Also

`heatmap`

Topics

"Access Property Values"

Introduced in R2017a

head

Get top rows of table, timetable, or tall array

Syntax

```
B = head(A)
B = head(A,k)
```

Description

`B = head(A)` returns the first eight rows of table or timetable `A`.

`B = head(A,k)` returns the first `k` rows of `A`.

Examples

Preview Table

Create a table that contains 100 rows and five variables.

```
load patients
T = table(LastName,Gender,Age,Height,Weight);
size(T)
```

```
ans = 1×2

   100     5
```

Preview the first eight rows.

```
T2 = head(T)
```

```
T2=8×5 table
      LastName        Gender      Age    Height    Weight
```

```
    ____________    __________    ___    ______    ______

    {'Smith'   }    {'Male'  }    38       71       176
    {'Johnson' }    {'Male'  }    43       69       163
    {'Williams'}    {'Female'}    38       64       131
    {'Jones'   }    {'Female'}    40       67       133
    {'Brown'   }    {'Female'}    49       64       119
    {'Davis'   }    {'Female'}    46       68       142
    {'Miller'  }    {'Female'}    33       64       142
    {'Wilson'  }    {'Male'  }    40       68       180
```

Preview Contents of Tall Table

Create a tall table and preview the first few rows of data.

Create a tall table for the `airlinesmall.csv` data set. Select a subset of the variables to work with. Use `head` to extract the first few rows of data.

```
varnames = {'Year','Month','ArrDelay','DepDelay','UniqueCarrier'};
ds = datastore('airlinesmall.csv','TreatAsMissing','NA',...
    'SelectedVariableNames',varnames);
T = tall(ds)
```

```
T =

  Mx5 tall table

    Year    Month    ArrDelay    DepDelay    UniqueCarrier
    ____    _____    ________    ________    _____________

    1987     10          8          12          {'PS'}
    1987     10          8           1          {'PS'}
    1987     10         21          20          {'PS'}
    1987     10         13          12          {'PS'}
    1987     10          4          -1          {'PS'}
    1987     10         59          63          {'PS'}
    1987     10          3          -2          {'PS'}
    1987     10         11          -1          {'PS'}
     :        :          :           :              :
     :        :          :           :              :
```

```
tt = head(T)
```

```
tt =

  8x5 tall table

    Year    Month    ArrDelay    DepDelay    UniqueCarrier
    ____    _____    ________    ________    _____________

    1987     10          8          12          {'PS'}
    1987     10          8           1          {'PS'}
    1987     10         21          20          {'PS'}
    1987     10         13          12          {'PS'}
    1987     10          4          -1          {'PS'}
    1987     10         59          63          {'PS'}
    1987     10          3          -2          {'PS'}
    1987     10         11          -1          {'PS'}
```

Collect the results into memory to view the data.

```
t8 = gather(tt)
```

```
t8=8×5 table
    Year    Month    ArrDelay    DepDelay    UniqueCarrier
    ____    _____    ________    ________    _____________

    1987     10          8          12          {'PS'}
    1987     10          8           1          {'PS'}
    1987     10         21          20          {'PS'}
    1987     10         13          12          {'PS'}
    1987     10          4          -1          {'PS'}
    1987     10         59          63          {'PS'}
    1987     10          3          -2          {'PS'}
    1987     10         11          -1          {'PS'}
```

Retrieve Specified Number of Rows in Tall Array

Preview the first 20 rows of data in a tall table.

Create a tall table for the `airlinesmall.csv` data set. Select a subset of the variables to work with, and treat `'NA'` values as missing data so that `datastore` replaces them with `NaN` values. Use `head` to view the first 20 rows of data.

```
varnames = {'Year','Month','ArrDelay','DepDelay','UniqueCarrier'};
ds = datastore('airlinesmall.csv','TreatAsMissing','NA',...
    'SelectedVariableNames',varnames);
T = tall(ds)
```

```
T =

  Mx5 tall table

    Year    Month    ArrDelay    DepDelay    UniqueCarrier
    ____    _____    ________    ________    _____________

    1987     10          8          12          {'PS'}
    1987     10          8           1          {'PS'}
    1987     10         21          20          {'PS'}
    1987     10         13          12          {'PS'}
    1987     10          4          -1          {'PS'}
    1987     10         59          63          {'PS'}
    1987     10          3          -2          {'PS'}
    1987     10         11          -1          {'PS'}
     :        :          :           :              :
     :        :          :           :              :
```

```
tt = head(T,20)
```

```
tt =

  20x5 tall table

    Year    Month    ArrDelay    DepDelay    UniqueCarrier
    ____    _____    ________    ________    _____________

    1987     10          8          12          {'PS'}
    1987     10          8           1          {'PS'}
    1987     10         21          20          {'PS'}
    1987     10         13          12          {'PS'}
    1987     10          4          -1          {'PS'}
    1987     10         59          63          {'PS'}
    1987     10          3          -2          {'PS'}
    1987     10         11          -1          {'PS'}
     :        :          :           :              :
     :        :          :           :              :
```

Collect the results into memory to view the data.

```
t20 = gather(tt)
```

```
t20=20×5 table
    Year    Month    ArrDelay    DepDelay    UniqueCarrier
    ____    _____    ________    ________    _____________

    1987     10          8          12          {'PS'}
    1987     10          8           1          {'PS'}
    1987     10         21          20          {'PS'}
    1987     10         13          12          {'PS'}
    1987     10          4          -1          {'PS'}
    1987     10         59          63          {'PS'}
    1987     10          3          -2          {'PS'}
    1987     10         11          -1          {'PS'}
    1987     10          3           3          {'PS'}
    1987     10          2           1          {'PS'}
    1987     10         16          15          {'PS'}
    1987     10          3           9          {'PS'}
    1987     10         39          15          {'PS'}
    1987     10         57          32          {'TW'}
    1987     10          0          -3          {'TW'}
    1987     10        -14           0          {'TW'}
       ⋮
```

Input Arguments

A — Input array
table | timetable

Input array, specified as a table or timetable.

Data Types: `table` | `timetable`

k — Number of rows to extract
scalar

Number of rows to extract, specified as a positive scalar integer. If `A` has fewer than `k` rows, then `head` returns all of `A`.

Output Arguments

B — Requested rows
table | timetable

Requested rows, returned as a table or timetable. The data type of B is the same as A.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

You can use `head` and `tail` with tall arrays of any valid underlying data type (`single`, `double`, `int8`, `datetime`, `table`, and so on).

If you are unsure whether the result returned by `gather(A)` will fit in memory, then use `gather(head(A))` or `gather(tail(A))`. These commands still fully evaluate the tall array `A`, but only return a small subset of the result in memory.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`gather` | `summary` | `table` | `tail` | `tall` | `timetable` | `topkrows`

Topics

"Index and View Tall Array Elements"

Introduced in R2016b

height

Number of table rows

Syntax

```
H = height(T)
```

Description

`H = height(T)` returns the number of rows in the table, `T`.

`height(T)` is equivalent to `size(T,1)`.

Examples

Number of Table Rows

Create a table, `T`.

```
LastName = {'Smith';'Johnson';'Williams';'Jones';'Brown'};
Age = [38;43;38;40;49];
Height = [71;69;64;67;64];
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];

T = table(Age,Height,Weight,BloodPressure,'RowNames',LastName)
```

```
T=5×4 table
                Age    Height    Weight    BloodPressure
                ___    ______    ______    _____________

    Smith       38       71       176       124     93
    Johnson     43       69       163       109     77
    Williams    38       64       131       125     83
    Jones       40       67       133       117     75
```

```
    Brown       49       64       119       122     80
```

Find the number of rows in table `T`.

```
H = height(T)
```

```
H = 5
```

`T` contains five rows; `height` does not count the variable names.

Input Arguments

`T` — Input table
table

Input table, specified as a table.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

This function fully supports code generation. For more information, see “Code Generation for Tables” (MATLAB Coder).

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`numel` | `size` | `width`

Introduced in R2013b

help

Help for functions in Command Window

Syntax

```
help name
help
```

Description

`help name` displays the help text for the functionality specified by `name`, such as a function, method, class, toolbox, or variable.

Note Some help text displays the names of functions in uppercase characters to make them stand out from the rest of the text. When typing these function names, use lowercase. For function names that appear in mixed case (such as `javaObject`), type the names as shown.

`help` displays content relevant to your previous actions.

Examples

Display Function and Overloaded Method Help

Display help for the MATLAB `delete` function.

```
help delete
```

```
delete Delete file or graphics object.
    delete file_name  deletes the named file from disk.  Wildcards
    may be used.  For example, delete *.p deletes all P-files from the
    current directory.
...
```

Because `delete` is the name of a function and of several methods, the help text includes a link to a list of the methods with the same name.

Request help for the `delete` method of the `handle` class.

```
help handle/delete
```

Display Package, Class, and Method Help

Display help for the `containers` package, the `Map` class, and the `isKey` method.

```
help containers
help containers.Map
help containers.Map.isKey
```

Not all packages, classes, and associated methods or events require complete specification. For example, display the help text for the `throwAsCaller` method of the `MException` class.

```
help throwAsCaller
```

Display Variable Help

Display help for a variable of type `datetime`.

```
t = datetime
help t
```

Because `t` is of type `datetime`, the `help` command displays help text for the `datetime` class.

Request help for `Day`, a method of the `datetime` class.

```
help t.Day
```

Display Help for Functions in Folder

List all of the functions in the folder *`matlabroot`*`/toolbox/matlab/timefun` by specifying a partial path.

```
help timefun
```

Input Arguments

name — Functionality name
character vector | string scalar

Functionality name, such as the name of a function, method, class, toolbox, or variable, specified as a character vector or string scalar. `name` also can be an operator symbol (such as `+`).

If `name` is a variable, `help` displays the help text for the class of that variable.

To get help for a method of a class, specify the class name and the method name, separated with a period. For example, to get help for the `methodname` method of the `classname` class, type `help classname.methodname`.

Some classes and other packaged items require that you specify the package name. Events, properties, and some methods require that you specify the class name. Separate the components of the name with periods. For example, to get help for the `propertyname` property of the `classname` class, type `classname.propertyname`. To get help for the `classname` class in the `packagename` package, type `packagename.methodname`. To get help for the `methodname` method of the `classname` class in the `packagename` package, type `packagename.classname.methodname`.

If `name` appears in multiple folders on the MATLAB search path, `help` displays the help text for the first instance of `name` found on the search path.

If `name` is overloaded, `help` displays a link to a list of the methods with the same name.

If `name` specifies the name or partial path of a folder:

- If the folder contains a nonempty `Contents.m` file, the `help` function displays the file. `Contents.m` contains a list of MATLAB program files in the folder and their descriptions.
- If the folder contains an empty `Contents.m` file, the `help` function displays *`name`* `is a folder`.
- If the folder does not contain a `Contents.m` file, the `help` function lists the first line of help text for each program file in the folder.

- If `name` is the name of both a folder and a function, the `help` function displays the associated text for both the folder and the function.

Tips

- To prevent long descriptions from scrolling off the screen before you have time to read them, enter `more on`, and then enter the `help` statement.
- To identify the package name for a class, create an instance of the class, and then call `class(obj)`.

Alternative Functionality

View more extensive help using the `doc` command or the Function Browser.

See Also

`doc` | `lookfor` | `more` | `what` | `which` | `whos`

Topics

"Ways to Get Function Help"
"Add Help for Your Program"

Introduced before R2006a

helpbrowser

Open Help browser to access online documentation

Note `helpbrowser` will be removed in a future release. Use `doc` instead.

Syntax

```
helpbrowser
```

Description

`helpbrowser` displays the Help browser, open to its default startup page.

See Also

`doc` | `help`

Topics

"Ways to Get Function Help"

Introduced before R2006a

helpdesk

Open Help browser

Note `helpdesk` will be removed in a future release. Use `doc` instead.

Syntax

```
helpdesk
```

Description

`helpdesk` opens the Help browser to the default startup page. In previous releases, `helpdesk` displayed the Help Desk, which was the precursor to the Help browser.

See Also

`doc`

Introduced before R2006a

helpdlg

Create help dialog box

Note If you are using App Designer or creating apps with the `uifigure` function, then use `uialert` instead. For more information, see “GUIDE Migration Strategies”.

Syntax

```
helpdlg
helpdlg(msg)
helpdlg(msg,title)
f = helpdlg( ___ )
```

Description

`helpdlg` creates a nonmodal on page 1-6349 Help dialog box with the default title, `Help Dialog`, and the default message, `This is the default help`.

`helpdlg(msg)` specifies custom message text. If a dialog box with a matching dialog box title already exists, then MATLAB brings it to the front.

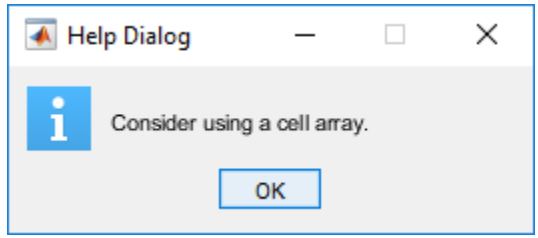
`helpdlg(msg,title)` specifies a custom title for the dialog box.

`f = helpdlg( ___ )` returns the figure object. You can use this syntax with any of the input arguments from the previous syntaxes.

Examples

Specify Custom Message

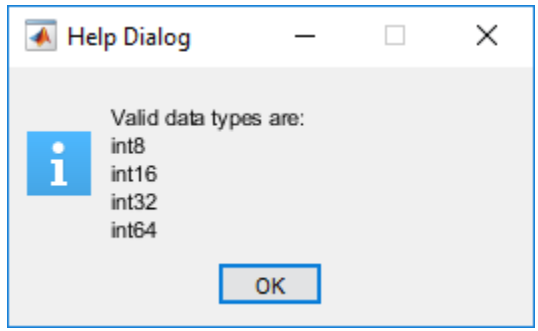
```
helpdlg('Consider using a cell array.');
```



Specify Message with Line Breaks

Specify the help dialog box text using a cell array of character vectors. Line breaks occur after each array element.

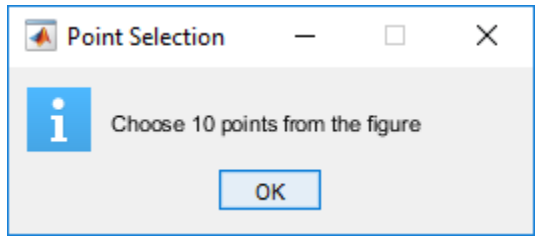
```
helpdlg({'Valid data types are:','int8',...
         'int16','int32','int64'});
```



Specify Custom Title and Message

Create a dialog box with a custom title and message.

```
helpdlg('Choose 10 points from the figure',...
        'Point Selection');
```



If the dialog box is not visible, it might be buried under other windows. Issue the command again to bring it to the front

```
helpdlg('Choose 10 points from the figure',...
        'Point Selection');
```

Now, change the message by calling `helpdlg` again using the same title with a different message.

```
helpdlg('Choose 5 points from the figure',...
        'Point Selection')
```

After you create a dialog box, each time you call `helpdlg` and specify the same `title`, MATLAB brings that dialog box to the front. If the dialog box `title` is unchanged, then MATLAB only creates a new dialog box if you close the dialog box between calls to `helpdlg`.

Input Arguments

`msg` — Help message
`'This is the default help'` (default) | character vector | cell array of character vectors | string array

Help message, specified as a character vector, cell array of character vectors, or string array.

- If you specify the help message as a character vector, then a long message wraps to fit the dialog box.
- If you specify the help message as a cell array of character vectors, then lines breaks occur between each array element. Long elements wrap to fit the dialog box.

Example: `'This value is required.'`

Example: `{'Valid data types are:','int8','int16','int32','int64'}`

`title` — Dialog box title
`'Help Dialog'` (default) | character vector | string array

Dialog box title, specified as a character vector or a string array.

Example: `'Value Specification'`

More About

Nonmodal Dialog Box

A nonmodal dialog box does not prevent the user to interact with other MATLAB windows before responding.

Tips

- MATLAB program execution continues even when a modal Help dialog box is active. To block program execution until the user closes the dialog box, use the `uiwait` function.

See Also

`dialog` | `errordlg` | `msgbox` | `warndlg`

Introduced before R2006a

helpwin

Provide access to help comments for all functions

Note `helpwin` will be removed in a future release. Use `doc` instead.

Syntax

```
helpwin
helpwin topic
```

Description

`helpwin` lists topics for groups of functions in the MATLAB Help browser. It shows brief descriptions of the topics and provides links to display help comments for the functions. You cannot follow links in the `helpwin` list of functions if the MATLAB software is busy (for example, running a program).

`helpwin topic` displays help information for the topic. If `topic` is a folder, it displays all functions in the folder. If `topic` is a function, `helpwin` displays help for that function. From the page, you can access a list of folders (**Default Topics** link) as well as the reference page help for the function (**Go to online doc** link). You cannot follow links in the `helpwin` list of functions if MATLAB is busy (for example, running a program).

Examples

Typing

```
helpwin datafun
```

displays the functions in the `datafun` folder and a brief description of each.

Typing

```
helpwin fft
```

displays the help for the `fft` function.

See Also

`doc` | `help`

Introduced before R2006a

hess

Hessenberg form of matrix

Syntax

```
H = hess(A)
[P,H] = hess(A)
[AA,BB,Q,Z] = hess(A,B)
```

Description

`H = hess(A)` finds `H`, the Hessenberg form of matrix `A`.

`[P,H] = hess(A)` produces a Hessenberg matrix `H` and a unitary matrix `P` so that `A = P*H*P'` and `P'*P = eye(size(A))`.

`[AA,BB,Q,Z] = hess(A,B)` for square matrices `A` and `B`, produces an upper Hessenberg matrix `AA`, an upper triangular matrix `BB`, and unitary matrices `Q` and `Z` such that `Q*A*Z = AA` and `Q*B*Z = BB`.

Examples

`H` is a 3-by-3 eigenvalue test matrix:

```
H =
   -149    -50   -154
    537    180    546
    -27     -9    -25
```

Its Hessenberg form introduces a single zero in the (3,1) position:

```
hess(H) =
   -149.0000    42.2037   -156.3165
   -537.6783   152.5511   -554.9272
           0     0.0728      2.4489
```

More About

Hessenberg Matrix

A Hessenberg matrix contains zeros below the first subdiagonal. If the matrix is symmetric or Hermitian, then the form is tridiagonal. This matrix has the same eigenvalues as the original, but less computation is needed to reveal them.

See Also

`eig` | `qz` | `schur`

Introduced before R2006a

matlab.mixin.Heterogeneous class

Package: `matlab.mixin`

Superclass for heterogeneous array formation

Description

`matlab.mixin.Heterogeneous` is an abstract class that supports forming heterogeneous arrays. A heterogeneous array is an array of objects that differ in their specific class, but are all derived from or are instances of a root class. The root class derives directly from `matlab.mixin.Heterogeneous`.

Class Attributes

`Abstract`	`true`
`HandleCompatible`	`true`

For information on class attributes, see “Class Attributes”.

Methods

Public Methods

cat	Concatenation for heterogeneous arrays
horzcat	Horizontal concatenation for heterogeneous arrays
vertcat	Vertical concatenation for heterogeneous arrays

Protected Methods

matlab.mixin.Heterogeneous.getDefaultScalarElement	Return default object for heterogeneous array operations

More About

Heterogeneous Hierarchy

Use `matlab.mixin.Heterogeneous` to define hierarchies of classes whose instances you can combine into heterogeneous arrays.

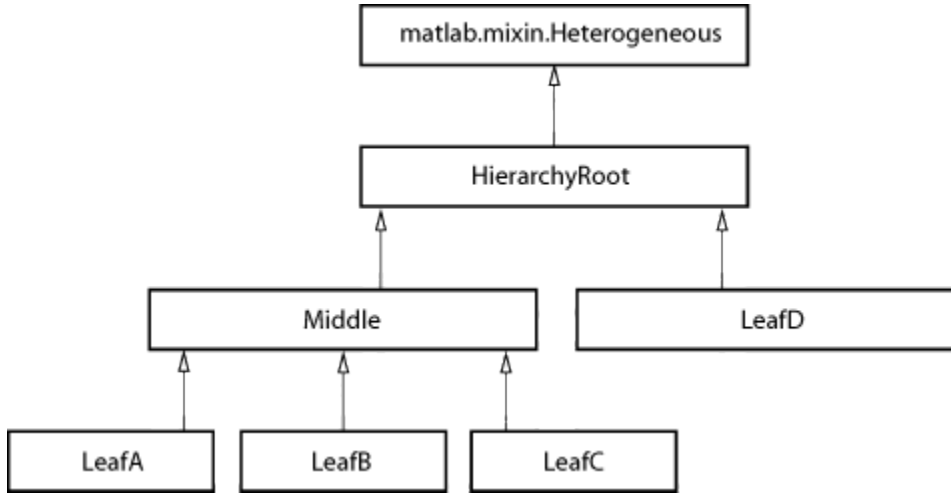
The following class definition enables the formation of heterogeneous arrays that combine instances of any classes derived from `HierarchyRoot`.

```
classdef HierarchyRoot < matlab.mixin.Heterogeneous
   % HierarchyRoot is a direct subclass of matlab.mixin.Heterogeneous.
   % HierarchyRoot is the root of this heterogeneous hierarchy.
end
```

Deriving the `HierarchyRoot` class directly from `matlab.mixin.Heterogeneous` enables the `HierarchyRoot` class to become the root of a hierarchy of classes. You can combine instances of the members of this hierarchy into a heterogeneous array. Only instances of classes derived from the same root class can combine to form a valid heterogeneous array.

Class of a Heterogeneous Array

The class of a heterogeneous array is always the class of the most specific superclass common to all objects in the array. For example, suppose you define the following class hierarchy:



Forming an array containing an instance of `LeafA` with an instance of `LeafB` creates an array of class `Middle`.

```
harray = [LeafA LeafB];
class(harray)
```

```
ans =
Middle
```

Forming an array containing an instance of `LeafC` with an instance of `LeafD` creates an array of class `HierarchyRoot`.

```
harray = [LeafC LeafD];
class(harray)
```

```
ans =
HierarchyRoot
```

Forming an array containing an instance of `LeafA` with another instance of `LeafA` creates a homogeneous array of class `LeafA`.

```
harray = [LeafA LeafA];
class(harray)
```

```
ans =
LeafA
```

Restrictions on Heterogeneous Array Formation

You can form heterogeneous arrays only with objects that are derived from the same hierarchy root (for example, the `HierarchyRoot` class in the hierarchy shown previously).

You can form heterogeneous arrays with objects that derive from multiple superclasses, but only one branch in the hierarchy can define a heterogeneous root.

Forming a Heterogeneous Array

Heterogeneous arrays are the result of operations that produce arrays containing instances of two or more classes from the heterogeneous hierarchy. Usually, the operation is concatenation or indexed assignment. For example, these statements form a heterogeneous array using indexed assignment.

```
harray(1) = LeafA;
harray(2) = LeafC;
class(harray)
```

```
ans =
Middle
```

Growing the Array Can Change Its Class

If an array contains objects derived from `matlab.mixin.Heterogeneous`, assigning new objects into it can change the class of the array. For example, consider a homogeneous array containing objects only of the `LeafA` class.

```
harray = [LeafA,LeafA,LeafA];
class(harray)
```

```
ans =
LeafA
```

Adding an object of a different class derived from the same root to a homogeneous array converts the array's class to the most specific superclass.

```
harray(4) = LeafB;
class(harray)
```

```
ans =
Middle
```

Method Dispatching

When MATLAB invokes a method for which the dominant argument is a heterogeneous array:

- The method must be defined for the class of the heterogeneous array, either directly by the class of the array or by inheritance from a superclass.
- The method must be `Sealed = true` (that is, cannot be overridden by a subclass). If you need to seal an inherited method, see Sealing Inherited Methods.

As with a homogeneous array, the class of the heterogeneous array determines which class method executes for any given method invocation. MATLAB does not consider the class of individual elements in the array when dispatching to methods.

Sealing Inherited Methods

The requirement that methods called on a heterogeneous array be `Sealed = true` ensures correct and predictable behavior with all array elements.

You must override methods that are inherited from outside the heterogeneous hierarchy if these methods are not `Sealed = true` and you want to call these methods on heterogeneous arrays.

For example, suppose you define a heterogeneous array by subclassing `matlab.mixin.SetGet`, in addition to `matlab.mixin.Heterogeneous`. Override the `set` method to call the `matlab.mixin.SetGet` superclass method as required by your class design.

```
classdef HeterogeneousSetGet < matlab.mixin.SetGet & matlab.mixin.Heterogeneous
    properties
        P
    end
    methods(Sealed)
        function varargout = set(obj,varargin)
            [varargout{1:nargout}] = set@matlab.mixin.SetGet(obj,varargin{:});
        end
    end
end
```

Method implementations can take advantage of the fact that, given a heterogeneous array `harray` and a scalar index `n`, the expression

```
harray(n)
```

is not a heterogeneous array. Therefore, when invoking a method on a single element of a heterogeneous array, special requirements for heterogeneous arrays do not apply.

Defining the Default Object

When working with object arrays (both heterogeneous and homogeneous), MATLAB creates default objects to fill in missing array elements by calling the class constructor with no arguments. Filling in missing array elements becomes necessary in cases such as these:

- Indexed assignment creating an array with gaps. For example, if `harray` is not previously defined:

  ```
  harray(5) = LeafA;
  ```

- Loading a heterogeneous array from a MAT-file when MATLAB cannot find the class definition of a specific object.

The `matlab.mixin.Heterogeneous` class provides a default implementation of a method called `getDefaultScalarElement`. This method returns an instance of the root class of the heterogeneous hierarchy, unless the root class is abstract.

If the root class is abstract or is not an appropriate default object for the classes in the heterogeneous hierarchy, you can override the `getDefaultScalarElement` method to return an instance of class that is derived from the root class.

Defining the getDefaultScalarElement Method

Specify the class of the default object by overriding the `matlab.mixin.Heterogeneous` method called `getDefaultScalarElement` in the root class of the heterogeneous hierarchy. You can override `getDefaultScalarElement` only in the root class.

The `getDefaultScalarElement` method has the following signature:

```
methods (Static, Sealed, Access = protected)
   function defaultObject = getDefaultScalarElement
   ...
   end
end
```

The `getDefaultScalarElement` method must satisfy these criteria:

- Static — MATLAB calls this method without an object.
- Protected — MATLAB calls this method; object users do not.
- Sealed (not required) — Seal this method to ensure users of the heterogeneous hierarchy do not change the intended behavior of the class.

- It must return a scalar object
- Its returned value must pass the `isa` test for the root class, that is

  `(isa(getDefaultScalarElement,'HierarchyRoot')`

  where *`HierarchyRoot`* is the name of the heterogeneous hierarchy root class. This means the default object can be an instance of any class derived from the root class.

Cannot Redefine Indexing or Concatenation

Heterogeneous arrays require consistent indexing and concatenation behaviors. Therefore, subclasses of `matlab.mixin.Heterogeneous` cannot change their default indexed-reference, indexed-assignment, or concatenation behavior.

You cannot override the following methods in your subclasses:

- `cat`
- `horzcat`
- `vertcat`
- `subsref`
- `subsasgn`

In cases involving multiple inheritance in which your subclass inherits from superclasses in addition to `matlab.mixin.Heterogeneous`, the superclasses cannot define any of these methods.

Default Concatenation Behavior

Statements of the form

```
a = [obj1 obj2 ...];
```

create an array, `a`, containing the objects listed in brackets.

Concatenating `Heterogeneous` objects of the same specific class preserves the class of the objects and does not form a heterogeneous array.

Concatenating `Heterogeneous` objects that are derived from the same root superclass, but that are of different specific classes, yields a heterogeneous array. MATLAB does not attempt to convert the class of any array members if all are part of the same root hierarchy.

Indexed Assignment Behavior

Statements of the form

```
a(m:n) = [objm ... objn];
```

assign the right-hand side objects to the array elements (`m:n`), specified on the left side of the assignment.

Indexed assignment to a heterogeneous array can do any of these:

- Increase or decrease the size of the array.
- Overwrite existing array elements.
- Change property values of objects within the array.
- Change the class of the array.
- Change whether the array is heterogeneous.

Indexed Reference Behavior

Statements of the form

```
a = harray(m:n);
```

assign the elements of `harray` referenced by indices `m:n`, to array `a`.

Indexed reference on a heterogeneous array returns a sub-range of the original array. Depending on the specific elements within that sub-range (`m:n`), the result might have a different class than the original array, and might not be heterogeneous.

Converting Nonmember Objects

If you attempt to form a heterogeneous array with objects that are not derived from the same root class, MATLAB calls the `convertObject` method, if it exists, to convert objects to the dominant class. Implementing a `convertObject` method enables the formation of heterogeneous arrays containing objects that are not part of the heterogeneous hierarchy.

When Is Conversion Necessary

Suppose there are two classes `A` and `B`, where `B` is not derived from `matlab.mixin.Heterogeneous`, or where `A` and `B` are derived from different root classes that are derived from `matlab.mixin.Heterogeneous`.

MATLAB attempts to call the `convertObject` method implemented by the root class of `A` in the following cases:

- Indexed assignment `A(k) = B`
- Horizontal and vertical concatenations `[A B] and [A;B]`

  `[A,B] and [A;B]`

Implement a `convertObject` method if you want to support conversion of objects whose class is not defined in your heterogeneous hierarchy. You do not need to implement this method if your class design does not require this conversion.

Implementing convertObject

Only the root class of the heterogeneous hierarchy can implement a `convertObject` method.

The `convertObject` method must have the following signature.

```
methods (Static, Sealed, Access = protected)
   function cobj = convertObject(DomClass,objToConvert)
   ...
   end
end
```

Where for indexed assignment `A(k) = B` and concatenation `[A B]`:

- `DomClass` is the name of the class of the array `A`.
- `objToConvert` is the object to be converted, `B` in this case.
- `cobj` is a legal member of the heterogeneous hierarchy to which `A` belongs.

`convertObject` must return a valid object of class `A` or MATLAB issues an error.

Handle Compatibility

The `matlab.mixin.Heterogeneous` class is handle compatible. It can be combined with either handle or value classes when defining a subclass using multiple superclasses. See “Handle Compatible Classes” for information on handle compatibility.

The `matlab.mixin.Heterogeneous` class is a value class. To learn how value classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.

See Also

Topics

“A Class Hierarchy for Heterogeneous Arrays”
“Designing Heterogeneous Class Hierarchies”
“Design Subclass Constructors”

Introduced in R2011a

cat

Class: `matlab.mixin.Heterogeneous`
Package: `matlab.mixin`

Concatenation for heterogeneous arrays

Syntax

```
C = cat(dim,A,B)
```

Description

`C = cat(dim,A,B)` concatenates objects `A` and `B` along the dimension `dim`. The class of object arrays `A` and `B` must be derived from the same root class of a `matlab.mixin.Heterogeneous` hierarchy.

- If `A` and `B` are of the same class, the class of the resulting array is unchanged.
- If `A` and `B` are of different subclasses of a common superclass that is derived from `matlab.mixin.Heterogeneous`, then the result is a heterogeneous array and the array's class is that of the most specific superclass shared by `A` and `B`.

The `cat` method is sealed in the class `matlab.mixin.Heterogeneous` and, therefore, you cannot override it in subclasses.

Input Arguments

`dim`

Scalar dimension along which to concatenate arrays

Default:

A

Object array derived from the same root subclass of `matlab.mixin.Heterogeneous` as `B`

B

Object array derived from the same root subclass of `matlab.mixin.Heterogeneous` as `A`

Output Arguments

C

Array resulting from the specified concatenation. The class of this array is that of the most specific superclass shared by `A` and `B`.

Attributes

`Sealed`	`true`
`Hidden`	`true`

To learn about attributes of methods, see Method Attributes.

See Also

`cat` | `matlab.mixin.Heterogeneous`

matlab.mixin.Heterogeneous.getDefaultScalarElement

Class: `matlab.mixin.Heterogeneous`
Package: `matlab.mixin`

Return default object for heterogeneous array operations

Syntax

```
defaultObject = getDefaultScalarElement
```

Description

`defaultObject = getDefaultScalarElement` returns the default object for a heterogeneous hierarchy. Override this method if the root class is abstract or is not an appropriate default object for the classes in the heterogeneous hierarchy. `getDefaultScalarElement` must return an instance of another member of the heterogeneous hierarchy.

The implementation of `getDefaultScalarElement` inherited from the `matlab.mixin.Heterogeneous` class returns an instance of the root class. If the root class is abstract, you must implement `getDefaultScalarElement` in the root class to return a default object. For more information, see “Root Class” on page 1-6368.

MATLAB calls the `getDefaultScalarElement` method when requiring a default object. For more information on heterogeneous arrays and default objects, see `matlab.mixin.Heterogeneous`.

Output Arguments

`defaultObject`

The default object for heterogeneous array operations.

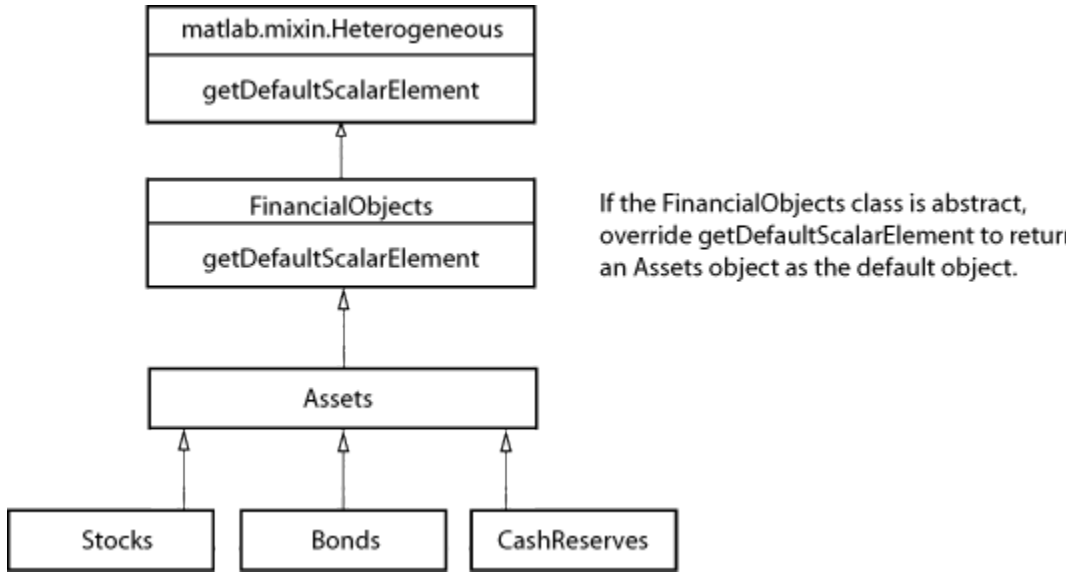
Attributes

`Static`	`true`
`Access`	`Protected`
`Sealed`	`true` not required

To learn about attributes of methods, see Method Attributes.

Examples

This example describes a heterogeneous hierarchy with a root class (`FinancialObjects`) that is an abstract class and cannot, therefore, be used for the default object. The `FinancialObjects` class definition includes an override of the `getDefaultScalarElement` method which returns an instance of the `Assets` class as the default object.



The root class can override the `getDefaultScalarElement` method that is defined in `matlab.mixin.Heterogeneous` class and return an `Assets` object as the default object.

```
classdef FinancialObjects < matlab.mixin.Heterogeneous
   methods (Abstract)
      val = determineCurrentValue(obj)
   end
   methods (Static, Sealed, Access = protected)
      function default_object = getDefaultScalarElement
         default_object = Assets;
      end
   end
end
```

More About

Root Class

Root class – The direct subclass of `matlab.mixin.Heterogeneous` that forms the root of a heterogeneous hierarchy. Classes of objects that you can combine into heterogeneous arrays must derive from this root class.

Tips

- Override `getDefaultScalarElement` only if the root class is not suitable as a default object.
- Override `getDefaultScalarElement` only in the root class of the heterogeneous hierarchy.
- `getDefaultScalarElement` must return a scalar object.
- `getDefaultScalarElement` must be a static method with protected access. While not required by MATLAB, you can seal this method to prevent overriding by other classes.
- MATLAB returns an error if the value returned by `getDefaultScalarElement` is not scalar or is not an instance of a class that is a valid member of the hierarchy.

See Also

`matlab.mixin.Heterogeneous`

horzcat

Class: `matlab.mixin.Heterogeneous`
Package: `matlab.mixin`

Horizontal concatenation for heterogeneous arrays

Syntax

```
C = horzcat(A1,A2,...)
```

Description

`C = horzcat(A1,A2,...)` concatenates the `matlab.mixin.Heterogeneous` objects `A1`, `A2`, and so on, to form the array `C`. All input arrays must have the same number of rows.

The class of object arrays `A1,A2,...` must be derived from the same root class of a `matlab.mixin.Heterogeneous` hierarchy.

MATLAB calls:

```
C = horzcat(A1,A2,...)
```

for the expressions:

```
C = [A1,A2,...]
C = [A1 A2 ...]
```

when `A1` is an array of `matlab.mixin.Heterogeneous` objects.

If all input arguments are of the same specific class, the class of the resulting array is unchanged. If all input arguments are of different subclasses of a common superclass that is derived from `matlab.mixin.Heterogeneous`, then the result is a heterogeneous array. The array's class is that of the most specific superclass shared by all input arguments.

If all input arguments are not members of the same heterogeneous hierarchy, MATLAB calls the `convertObjects` method, if defined by the dominant root class (the first argument or the left-most element in the concatenation if no other class is dominant).

The `horzcat` method is sealed in the class `matlab.mixin.Heterogeneous` and, therefore, you cannot override it in subclasses.

Input Arguments

`A1`

Object array of class `matlab.mixin.Heterogeneous`

`A2`

Object array of class `matlab.mixin.Heterogeneous`

Output Arguments

`C`

Array resulting from the specified concatenation. The class of this array is that of the most specific superclass shared by the input arguments.

Attributes

`Sealed`	`true`
`Hidden`	`true`

To learn about attributes of methods, see Method Attributes.

See Also

`horzcat` | `matlab.mixin.Heterogeneous`

vertcat

Class: `matlab.mixin.Heterogeneous`
Package: `matlab.mixin`

Vertical concatenation for heterogeneous arrays

Syntax

```
C = vertcat(A1,A2,...)
```

Description

`C = vertcat(A1,A2,...)` concatenates the `matlab.mixin.Heterogeneous` objects `A1`, `A2`, and so on, to form the array `C`. All input arrays must have the same number of columns.

The class of object arrays `A1,A2,...` must be derived from the same root class of a `matlab.mixin.Heterogeneous` hierarchy.

MATLAB calls:

```
C = vertcat(A1,A2,...)
```

for the expression:

```
C = [A1;A2;...]
```

when `A1` and `A2`, and so on are arrays of `matlab.mixin.Heterogeneous` objects.

If all input arguments are of the same specific class, the class of the resulting array is unchanged. If all input arguments are of different subclasses of a common superclass that is derived from `matlab.mixin.Heterogeneous`, then the result is a heterogeneous array. The array's class is that of the most specific superclass shared by all input arguments.

If all input arguments are not members of the same heterogeneous hierarchy, MATLAB calls the `convertObjects` method, if defined by the dominant root class (the first argument or the left-most element in the concatenation if no other class is dominant).

The `horzcat` method is sealed in the class `matlab.mixin.Heterogeneous` and, therefore, you cannot override it in subclasses.

Input Arguments

`A1`

Object array of class `matlab.mixin.Heterogeneous`

`A2`

Object array of class `matlab.mixin.Heterogeneous`

Output Arguments

`C`

Array resulting from the specified vertical concatenation. The class of this array is that of the most specific superclass shared by the input arguments.

Attributes

`Sealed`	`true`
`Hidden`	`true`

To learn about attributes of methods, see Method Attributes.

See Also

`matlab.mixin.Heterogeneous` | `vertcat`

hex2dec

Convert text representation of hexadecimal number to decimal number

Syntax

```
D = hex2dec(hexStr)
```

Description

`D = hex2dec(hexStr)` converts `hexStr` to a decimal number and returns it. The input argument `hexStr` is text that represents digits of a hexadecimal number using the characters `0`-`9` and either `A`-`F` or `a`-`f`.

Examples

Convert Text Representing Hexadecimal Value

Convert a character vector that represents a hexadecimal value to a decimal number.

```
hexStr = '3FF';
D = hex2dec(hexStr)
```

```
D = 1023
```

Starting in R2019b, you can write values in hexadecimal format directly without using `hex2dec`. Use the `0x` prefix and do not use quotation marks. MATLAB® stores the value as an integer, not as text.

```
D = 0x3FF
```

```
D = uint16
    1023
```

Return Numeric Array

Create a string array that represents multiple hexadecimal values.

```
hexStr = ["3FF" "7A" "E"]
```

```
hexStr = 1x3 string array
    "3FF"    "7A"    "E"
```

Convert the hexadecimal values and return a numeric array.

```
D = hex2dec(hexStr)
```

```
D = 1×3

        1023         122          14
```

Starting in R2019b, it is recommended that you create a numeric array using hexadecimal literals instead of converting text with `hex2dec`.

```
D = [0x3FF 0x7A 0xE]
```

```
D = 1x3 uint16 row vector

   1023    122     14
```

Input Arguments

`hexStr` — Text representing hexadecimal numbers
character array | cell array of character vectors | string array

Text representing hexadecimal numbers, specified as a character array, cell array of character vectors, or string array.

- If `hexStr` is a character array with multiple rows or a cell array of character vectors, then the output is a numeric column vector.
- If `hexStr` is a string array, then the output is a numeric array that has the same dimensions.

`hexStr` cannot represent a negative number.

Tips

- If the input argument `hexStr` represents a value greater than the value returned by `flintmax`, then `hex2dec` might not return an exact conversion.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Input text must be specified as a character array or string scalar. Cell arrays are not supported.

See Also

`base2dec` | `bin2dec` | `dec2hex` | `flintmax` | `format` | `hex2num` | `sprintf`

Topics

"Hexadecimal and Binary Values"

Introduced before R2006a

hex2num

Convert IEEE hexadecimal format to double-precision number

Syntax

```
X = hex2num(hexStr)
```

Description

`X = hex2num(hexStr)` converts `hexStr` to the double-precision floating-point number that it represents. The input argument `hexStr` has up to 16 characters representing a number in its IEEE format using hexadecimal digits. If `hexStr` has fewer than 16 digits, then `hex2num` pads `hexStr` with zeros to the right.

Unlike the `hex2dec` function, `hex2num` can convert inputs that represent floating-point values with fractional parts. Also, `hex2num` handles `NaN` and `Inf` value and denormal numbers correctly.

If `hexStr` represents multiple numbers, then `hex2dec` returns the numbers as a column vector of double-precision numbers.

Examples

Convert Hexadecimal Format to Number with Fractional Part

Create a character vector that represents a double-precision number in its IEEE® format, using hexadecimal digits.

```
hexStr = '400921fb54442d18';
```

To convert it to the number that it represents, use the `hex2num` function. If `hexStr` represents a number with a fractional part, then `hex2num` converts it correctly.

```
X = hex2num(hexStr)
```

```
X = 3.1416
```

You can convert X back to its hexadecimal representation using the num2hex function.

```
num2hex(X)
```

```
ans =
'400921fb54442d18'
```

Convert Hexadecimal Representations of Negative Number, NaN, and Inf

Create a character vector that represents a negative number. Then convert it.

```
hexStr = 'BFF';
X = hex2num(hexStr)
```

```
X = -1
```

Convert a representation of NaN.

```
hex2num('FFF8')
```

```
ans = NaN
```

Convert a representation of Inf.

```
hex2num('7FF')
```

```
ans = Inf
```

Return Double-Precision Array

Create a string array that represents multiple numbers in their IEEE® format, using hexadecimal digits.

```
hexStr = ["4005bf0b0d7a3c06" "3fc189374bc6a7f0" "402c"]
```

```
hexStr = 1x3 string array
    "4005bf0b0d7a3c06"    "3fc189374bc6a7f0"    "402c"
```

Convert `hexStr` and return it as an array of double-precision numbers.

```
X = hex2num(hexStr)
```

```
X = 1×3

    2.7183    0.1370   14.0000
```

Input Arguments

`hexStr` — Text representing hexadecimal numbers
character array | cell array of character vectors | string array

Text representing hexadecimal numbers, specified as a character array, cell array of character vectors, or string array.

- If `hexStr` is a cell array of character vectors or a string array, then `hex2num` converts each element to a number.
- If `hexStr` is a character array, then `hex2num` converts each row to a number.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If the output of `hex2num` is empty, the generated code returns a `0`-by-`1` array instead of a `0`-by-`0` array.

See Also

`format` | `hex2dec` | `num2hex` | `sprintf`

Topics

"Hexadecimal and Binary Values"

External Websites

IEEE 754

Introduced before R2006a

hgexport

Export figure

Syntax

```
hgexport(fig,filename)
hgexport(fig,'-clipboard')
```

Description

`hgexport(fig,filename)` writes figure `fig` to the EPS file `filename`.

`hgexport(fig,'-clipboard')` writes figure `fig` to the Microsoft Windows clipboard. The format in which the figure is exported is determined by which renderer you use. The Painters renderer generates a metafile. The OpenGL renderer generate a bitmap.

Alternatives

Use the **File > Export Setup** dialog. Use **Edit > Copy Figure** to copy the figure's content to the system clipboard. For details, see "Customize Figure Before Saving" and "Copy Figure to Clipboard from Edit Menu".

See Also

`print`

Introduced before R2006a

hggroup

Create group object

Syntax

```
h = hggroup
h = hggroup(...,'PropertyName',propertyvalue,...)
h = hggroup(ax,...)
```

Properties

For a list of properties, see Group.

Description

`h = hggroup` creates a `Group` object as a child of the current axes and returns its handle, `h`.

`h = hggroup(...,'PropertyName',propertyvalue,...)` creates a `Group` object with the property values specified in the argument list.

`h = hggroup(ax,...)` creates the `Group` object in the axes specified by `ax` instead of in the current axes (`gca`). The option `ax` can precede any of the input argument combinations in the previous syntaxes.

A `Group` object can be the parent of any axes, as well as other `Group` objects. Use `Group` objects to form a group of child objects that can be treated as a single object.

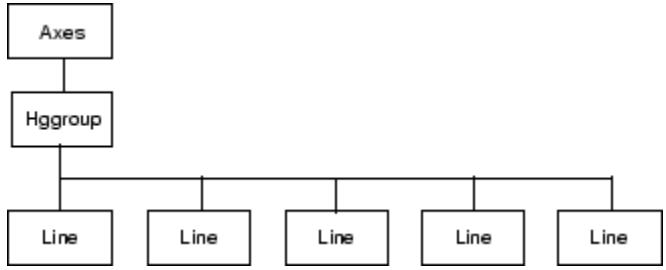
Examples

Plot random data and parented the lines to the `Group` object.

```
hg = hggroup;
plot(randn(5),randn(5),'Parent',hg)
```

Instance Diagram for This Example

The following diagram shows the object hierarchy created by this example.



See Also

`hgtransform`

Topics

"Create Object Groups"

Introduced before R2006a

hgload

Load graphics object hierarchy from file

Syntax

```
h = hgload(filename)
[h,old_prop_values] = hgload(...,property_structure)
```

Description

Note `hgload` is not recommended. Use `openfig` instead.

`h = hgload(filename)` loads graphics object hierarchy from the FIG-file specified by `filename` and returns handles to the top-level objects. Specify `filename` as a character vector or string. If `filename` contains no extension, then MATLAB adds the `.fig` extension.

`[h,old_prop_values] = hgload(...,property_structure)` overrides the properties on the top-level objects stored in the FIG-file with the values in `property_structure`, and returns their previous values in `old_prop_values`.

`property_structure` must be a structure having field names that correspond to property names and values that are the new property values.

`old_prop_values` is a cell array equal in length to `h`, containing the old values of the overridden properties for each object. Each cell contains a structure having field names that are property names, each of which contains the original value of each property that has been changed. Any property specified in `property_structure` that is not a property of a top-level object in the FIG-file is not included in `old_prop_values`.

Nonserializable objects (such as the default toolbars and the default menus) are not saved because they are created when the figure is created. This allows revisions of the default menus and toolbars to occur without affecting existing FIG-files.

Alternatives

Use the **File > Open** on the figure window menu to access figure files with the **Open** dialog.

See Also

hgsave | open

Introduced before R2006a

hgsave

Save graphics object hierarchy to file

Syntax

```
hgsave(filename)
hgsave(h,filename)
hgsave(...,'-v6')
hgsave(...,'-v7.3')
```

Description

Note `hgsave` is not recommended. Use `savefig` instead.

`hgsave(filename)` saves the current figure to a file named `filename`. Specify `filename` as a character vector or string.

`hgsave(h,filename)` saves the objects identified by the array of handles `h` to a file named `filename`. If you do not specify an extension for `filename`, then the extension `.fig` is appended. If `h` is a vector, none of the handles in `h` may be ancestors or descendents of any other handles in `h`.

`hgsave(...,'-v6')` saves the FIG-file in a format that can be loaded by versions prior to MATLAB 7.

`hgsave(...,'-v7.3')` saves the FIG-file in a format that can be loaded only by MATLAB versions 7.3 and above. This format, based on HDF5 files, is intended for saving FIG-files larger than 2 GB.

Backward Compatibility

When creating a figure you want to save and use in a MATLAB version prior to MATLAB 7, use the `'v6'` option with the plotting function and the `'-v6'` option for `hgsave`. Check the reference page for the plotting function you are using for more information.

In MATLAB release R2014b or later, you cannot open a save FIG-file in earlier versions of MATLAB. Use `savefig` to save figures that are compatible with earlier versions of MATLAB.

Alternatives

Use the **File > Export Setup** dialog. Use **Edit > Copy Figure** to copy the figure's content to the system clipboard. For details, see "Customize Figure Before Saving" and "Copy Figure to Clipboard from Edit Menu".

See Also

`hgload` | `open` | `save` | `savefig`

Introduced before R2006a

hgsetget

Abstract class used to derive handle class with set and get methods

Note `hgsetget` will be removed in a future release. Use `matlab.mixin.SetGet` instead.

Syntax

```
classdef myclass < hgsetget
```

Description

`classdef myclass < hgsetget` makes *`myclass`* a subclass of the `hgsetget` class, which is a subclass of the `handle` class.

Use the `hgsetget` class to derive classes that inherit `set` and `get` methods that behave like Handle Graphics® `set` and `get` functions.

Methods

When you derive a class from the `hgsetget` class, your class inherits the following methods.

Method	Purpose
set	Assigns values to the specified properties or returns a cell array of possible values for writable properties.
get	Returns value of specified property or a `struct` with all property values.

Method	Purpose
setdisp	Called when `set` is called with no output arguments and a handle array, but no property name. Override this method to change what set displays.
getdisp	Called when `get` is called with no output arguments and handle array, but no property name. Override this method to change what get displays.

See Also

Topics

"Implement Set/Get Interface for Properties"

hgtransform

Create transform object

Syntax

```
h = hgtransform
h = hgtransform('PropertyName',propertyvalue,...)
h = hgtransform(ax,...)
```

Properties

For a list of properties, see Transform.

Description

`h = hgtransform` creates a `Transform` object and returns its handle.

`h = hgtransform('PropertyName',propertyvalue,...)` creates a `Transform` object with the property value settings specified in the argument list. For a description of the properties, see Transform.

`h = hgtransform(ax,...)` creates the `Transform` object in the axes specified by `ax` instead of in the current axes (`gca`). The option `ax` can precede any of the input argument combinations in the previous syntaxes.

`Transform` objects can contain other objects, which lets you treat the `Transform` object and its children as a single entity with respect to visibility, size, orientation, etc. You can group objects by parenting them to a single `Transform` object (i.e., setting the object's `Parent` property to the transform object's handle):

```
h = hgtransform;
surface('Parent',h,...)
```

The primary advantage of parenting objects to a `Transform` object is that you can perform *transforms* (for example, translation, scaling, rotation, etc.) on the child objects in unison.

The parent of a `Transform` object is either an `Axes` object or another `Transform` object.

Although you cannot see a `Transform` object, setting its `Visible` property to `off` makes all its children invisible as well.

Exceptions and Limitations

- A `Transform` object can be the parent of any number of axes child objects belonging to the same axes, except for light objects.
- `Transform` objects can never be the parent of axes objects and therefore can contain objects only from a single axes.
- `Transform` objects can be the parent of other transform objects within the same axes.
- You cannot transform `Image` objects because images are not true 3-D objects. Texture mapping the image data to a surface `CData` enables you to produce the effect of transforming an image in 3-D space.
- Transforms do not affect text objects unless the text object uses data units. If a `Text` object has a position specified in data units, then the transform moves the lower left corner of the text. The transform does not affect the font size or orientation. To change the font size and orientation, use text properties.

Note Many plotting functions clear the axes (remove axes children) before drawing the graph. Clearing the axes also deletes any `Transform` objects in the axes.

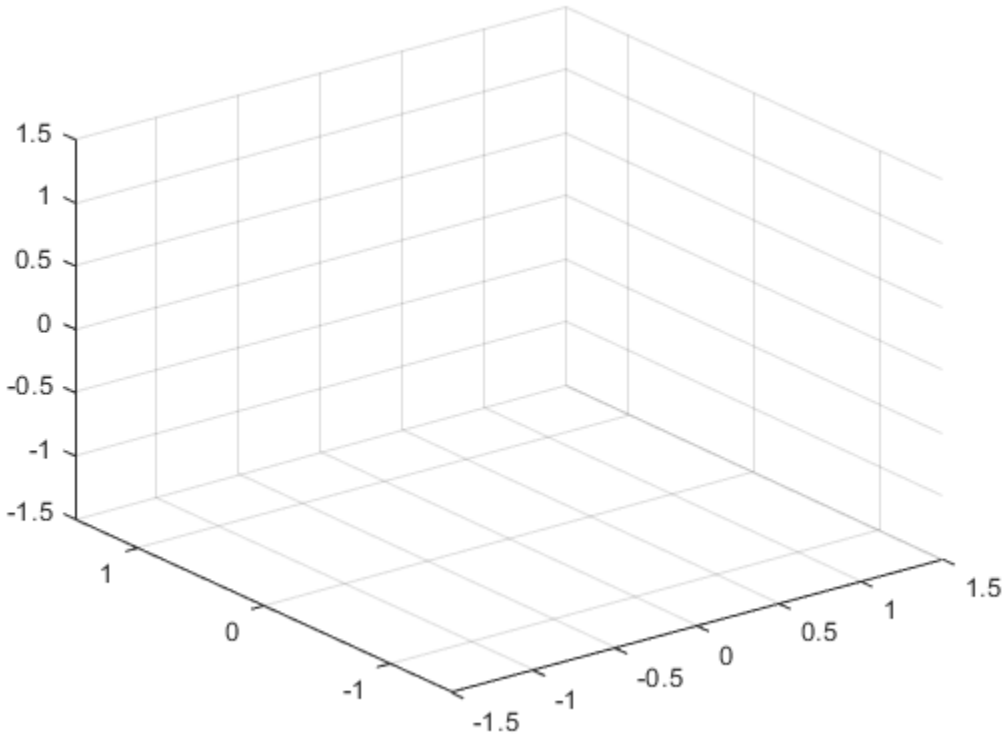
Examples

Transforming a Group of Objects

This example shows how to create a 3-D star with a group of surface objects parented to a single transform object. The transform object then rotates the object about the z-axis while scaling its size.

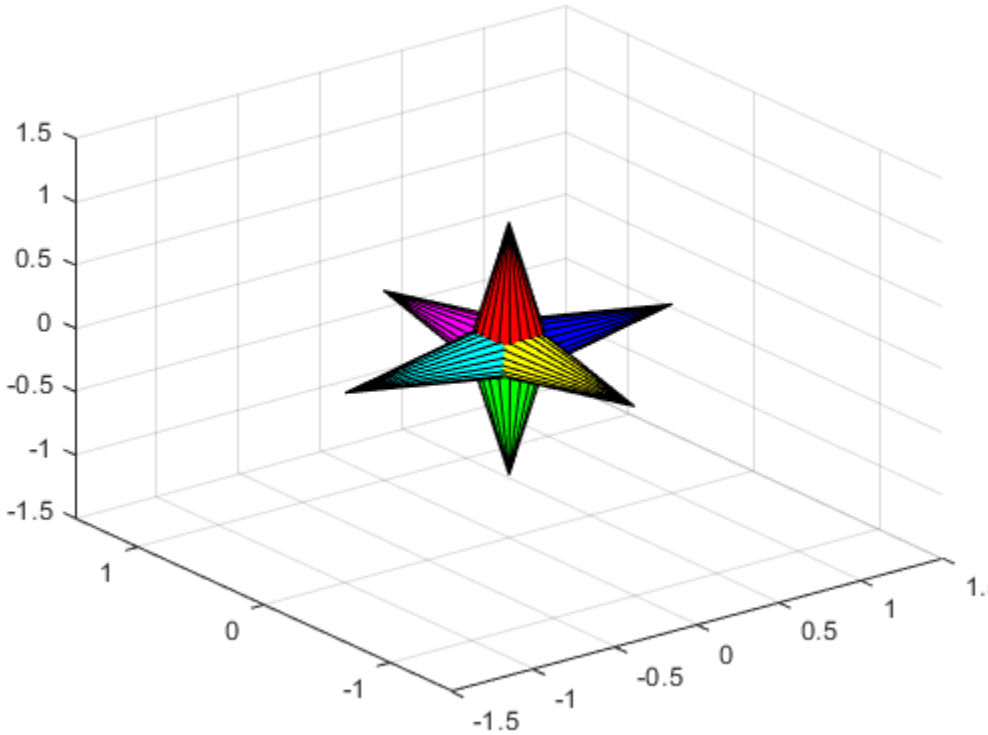
Create an axes and adjust the view. Set the axes limits to prevent auto limit selection during scaling.

```
ax = axes('XLim',[-1.5 1.5],'YLim',[-1.5 1.5],'ZLim',[-1.5 1.5]);
view(3)
grid on
```



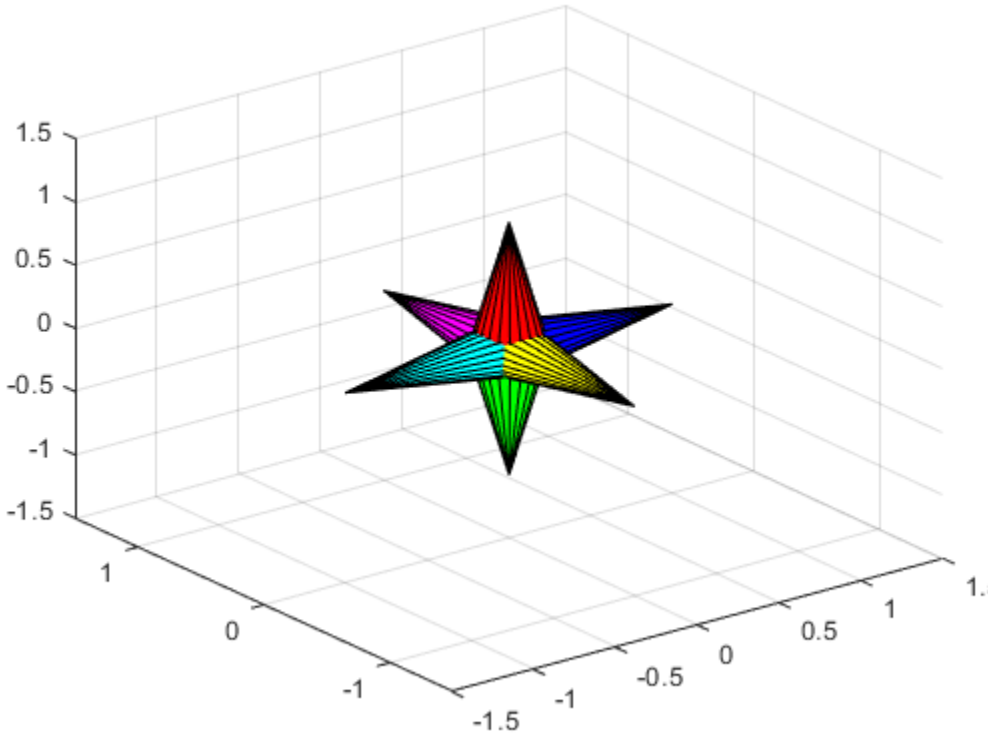
Create the objects you want to parent to the transform object.

```
[x,y,z] = cylinder([.2 0]);
h(1) = surface(x,y,z,'FaceColor','red');
h(2) = surface(x,y,-z,'FaceColor','green');
h(3) = surface(z,x,y,'FaceColor','blue');
h(4) = surface(-z,x,y,'FaceColor','cyan');
h(5) = surface(y,z,x,'FaceColor','magenta');
h(6) = surface(y,-z,x,'FaceColor','yellow');
```



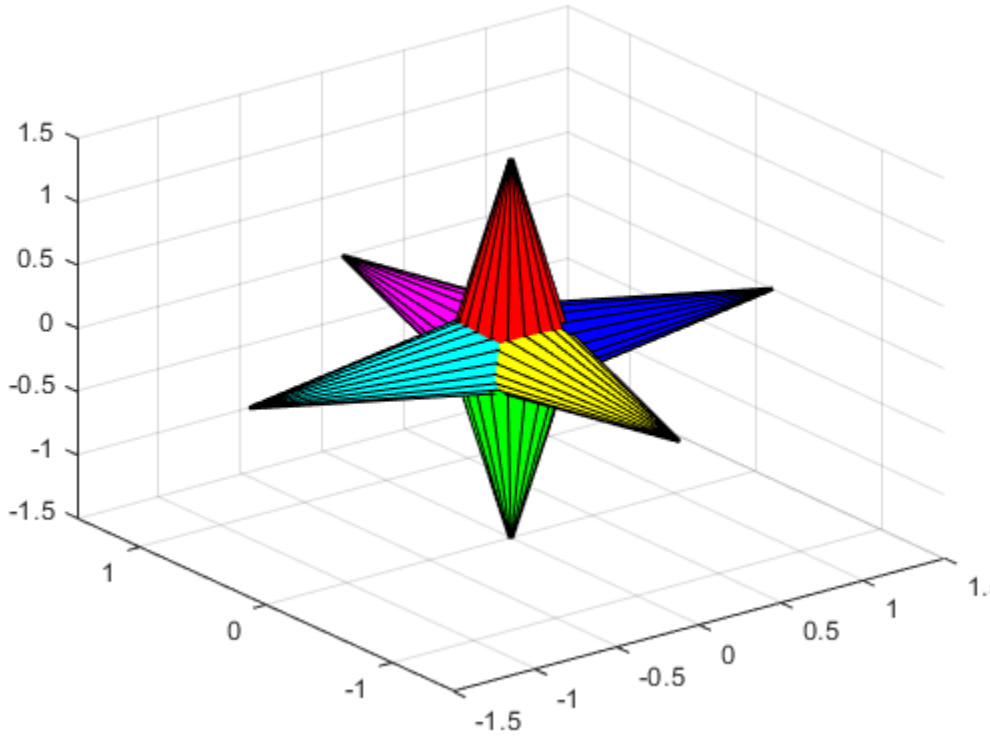
Create a transform object and parent the surface objects to it. Initialize the rotation and scaling matrix to the identity matrix (eye).

```
t = hgtransform('Parent',ax);
set(h,'Parent',t)

Rz = eye(4);
Sxy = Rz;
```



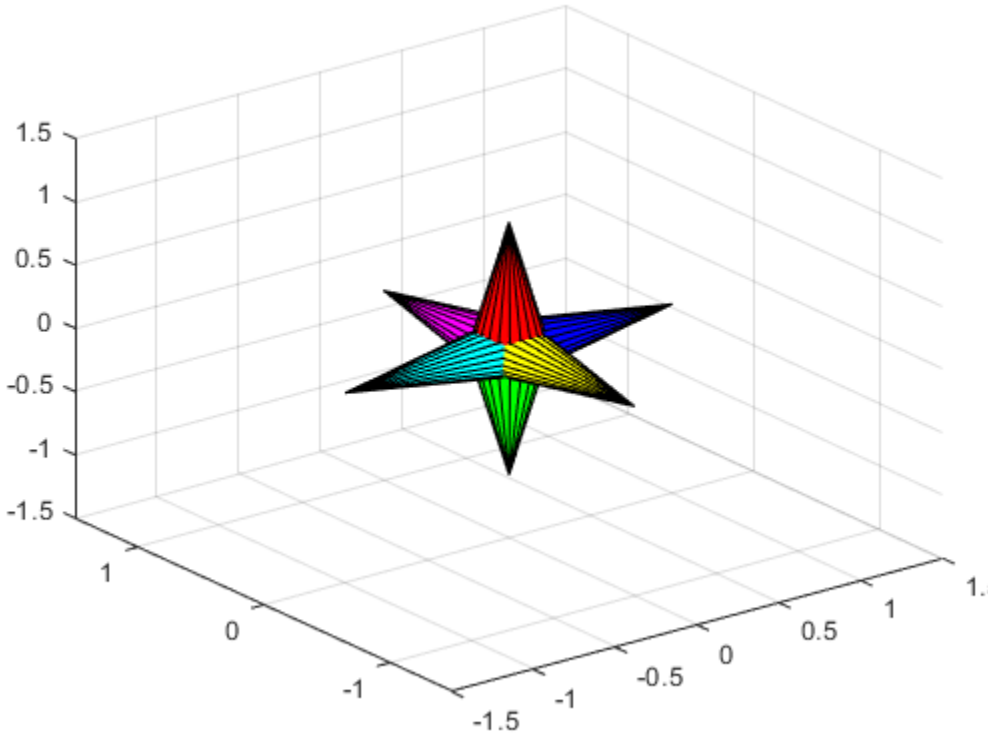
Form the *z*-axis rotation matrix and the scaling matrix. Rotate group and scale by using the increasing values of `r`.

```
for r = 1:.1:2*pi
    % Z-axis rotation matrix
    Rz = makehgtform('zrotate',r);
    % Scaling matrix
    Sxy = makehgtform('scale',r/4);
    % Concatenate the transforms and
    % set the transform Matrix property
    set(t,'Matrix',Rz*Sxy)
    drawnow
end
pause(1)
```



Reset to the original orientation and size using the identity matrix.

```
set(t,'Matrix',eye(4))
```

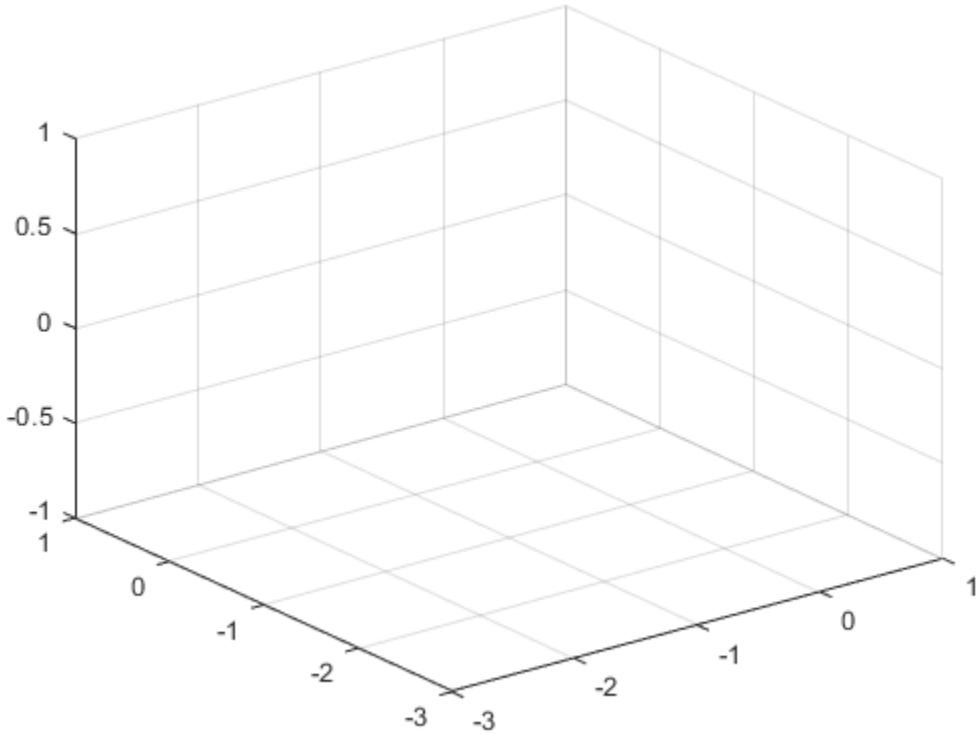


Transforming Objects Independently

This example creates two transform objects to illustrate how to transform each independently within the same axes. A translation transformation moves one transform object away from the origin.

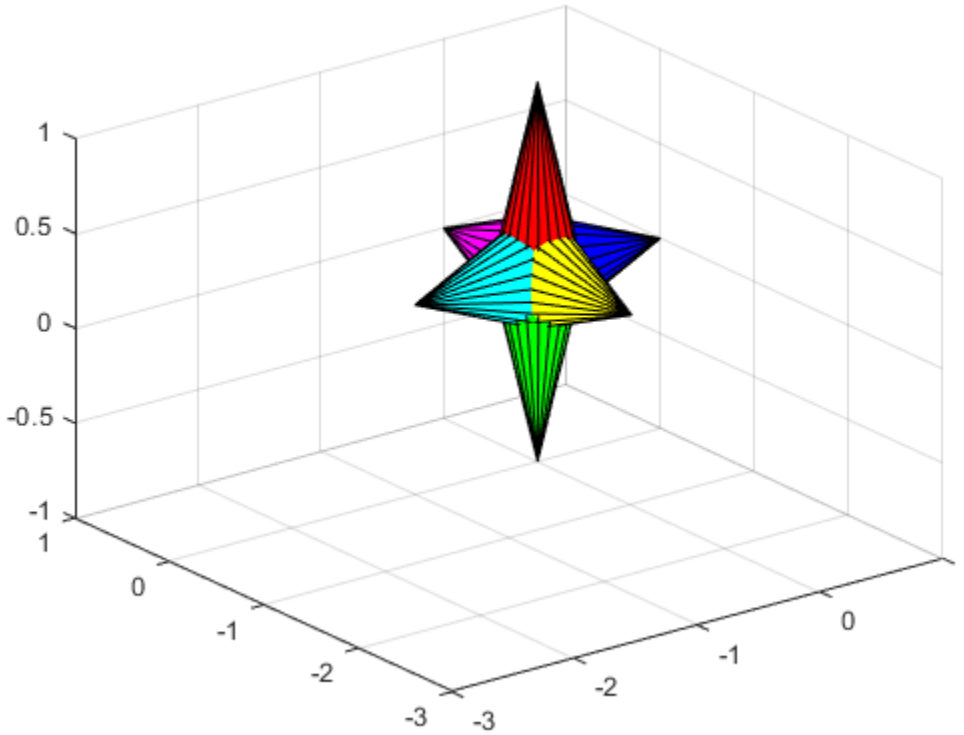
Create and set up the axes object that will be the parent of both transform objects. Set the limits to accommodate the translated object.

```
ax = axes('XLim',[-3 1],'YLim',[-3 1],'ZLim',[-1 1]);
view(3)
grid on
```



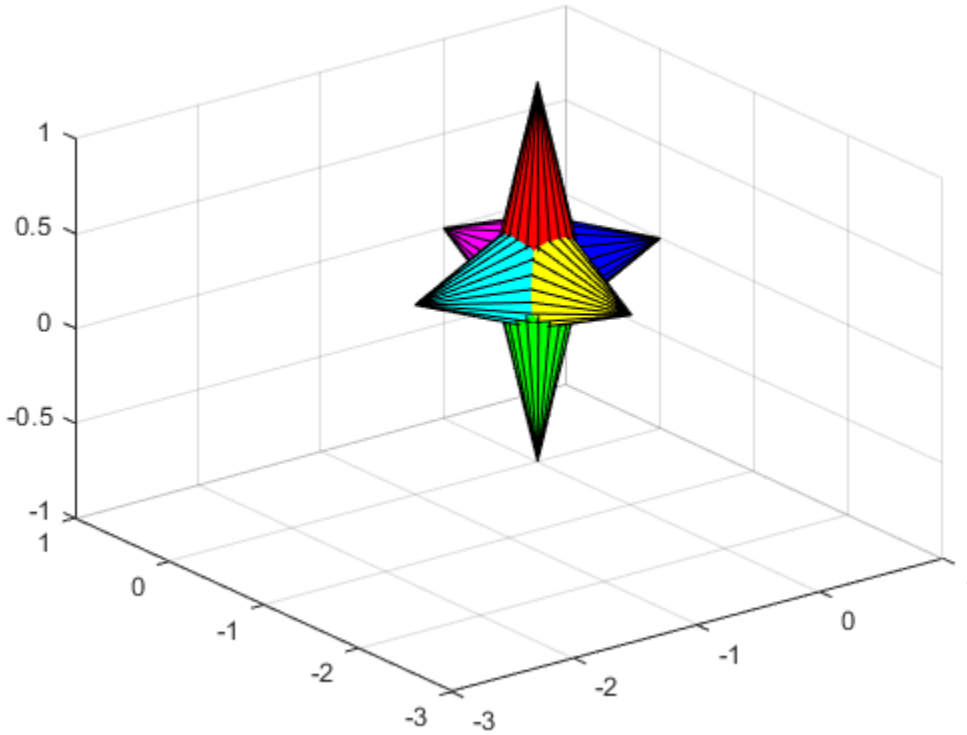
Create the surface objects to group.

```
[x,y,z] = cylinder([.3 0]);
h(1) = surface(x,y,z,'FaceColor','red');
h(2) = surface(x,y,-z,'FaceColor','green');
h(3) = surface(z,x,y,'FaceColor','blue');
h(4) = surface(-z,x,y,'FaceColor','cyan');
h(5) = surface(y,z,x,'FaceColor','magenta');
h(6) = surface(y,-z,x,'FaceColor','yellow');
```



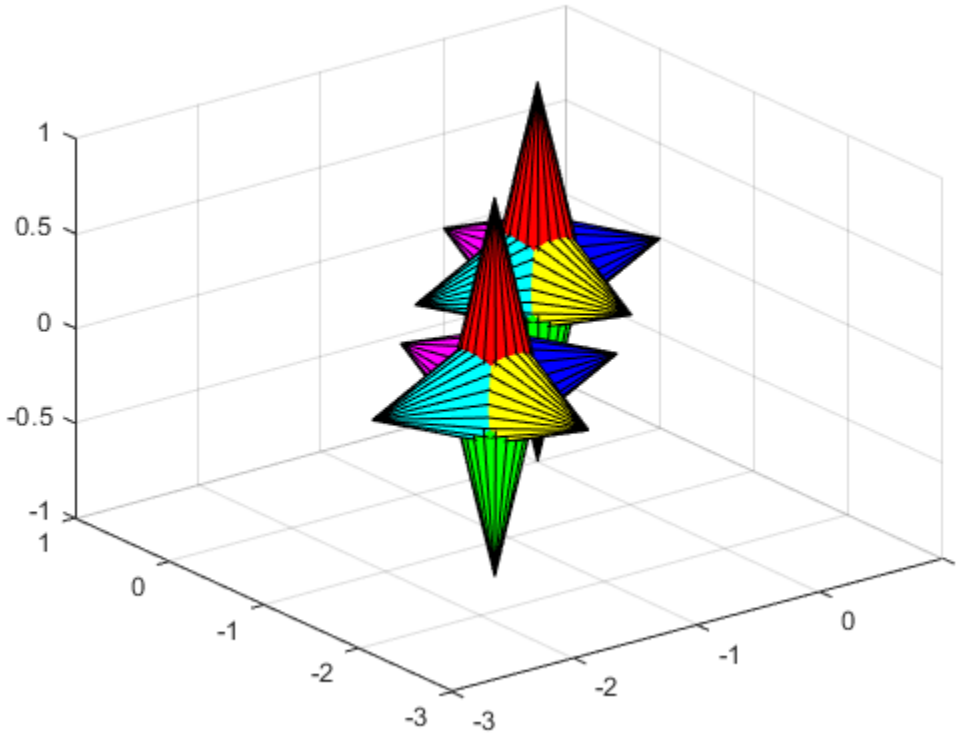
Create the transform objects and parent them to the same axes. Then, parent the surfaces to transform t1. Copy the surface objects and parent the copies to transform t2. This figure should not change.

```
t1 = hgtransform('Parent',ax);
t2 = hgtransform('Parent',ax);

set(h,'Parent',t1)
h2 = copyobj(h,t2);
```



Translate the second transform object away from the first transform object and display the result.

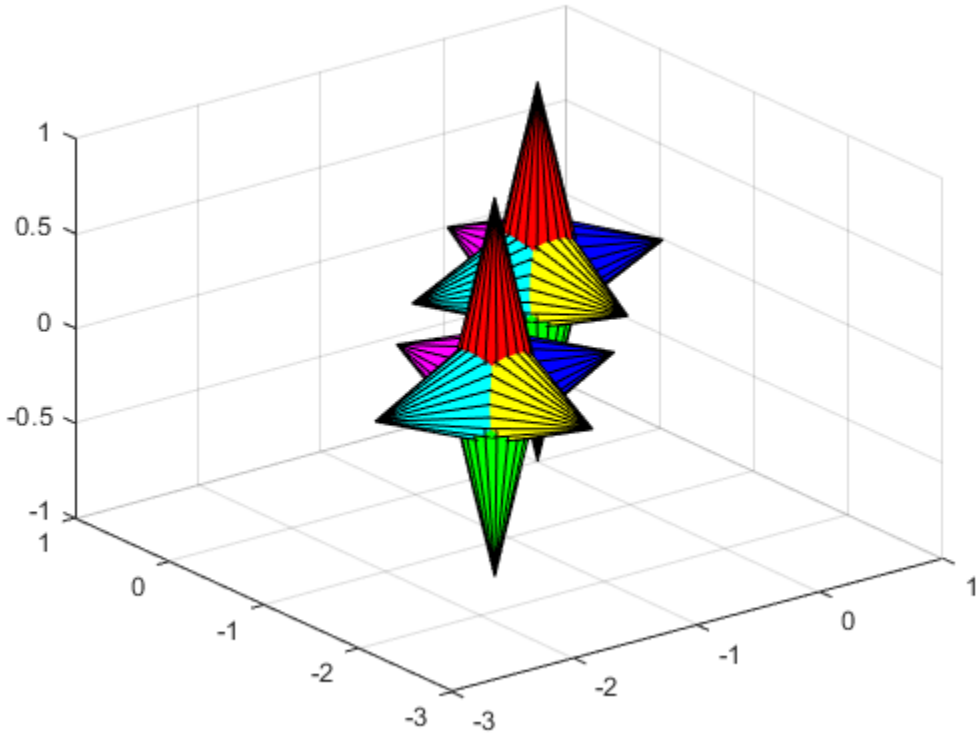
```
Txy = makehgtform('translate',[-1.5 -1.5 0]);
set(t2,'Matrix',Txy)
drawnow
```



Rotate both transform objects in opposite directions.

Rotate 10 times (2pi radians = 1 rotation)

```
for r = 1:.1:20*pi
    % Form z-axis rotation matrix
    Rz = makehgtform('zrotate',r);
    % Set transforms for both transform objects
    set(t1,'Matrix',Rz)
    set(t2,'Matrix',Txy*inv(Rz))
    drawnow
end
```



See Also

`hggroup` | `makehgtform`

Topics

"Create Object Groups"

Introduced before R2006a

hidden

Remove hidden lines from mesh plot

Syntax

```
hidden on
hidden off
hidden
hidden(ax,...)
```

Description

Hidden line removal draws only those lines that are not obscured by other objects in a 3-D view. The `hidden` function only applies to surface plot objects that have a uniform `FaceColor`.

`hidden on` turns on hidden line removal for the current mesh plot so lines in the back of a mesh are hidden by those in front. This is the default behavior.

`hidden off` turns off hidden line removal for the current mesh plot.

`hidden` toggles the hidden line removal state.

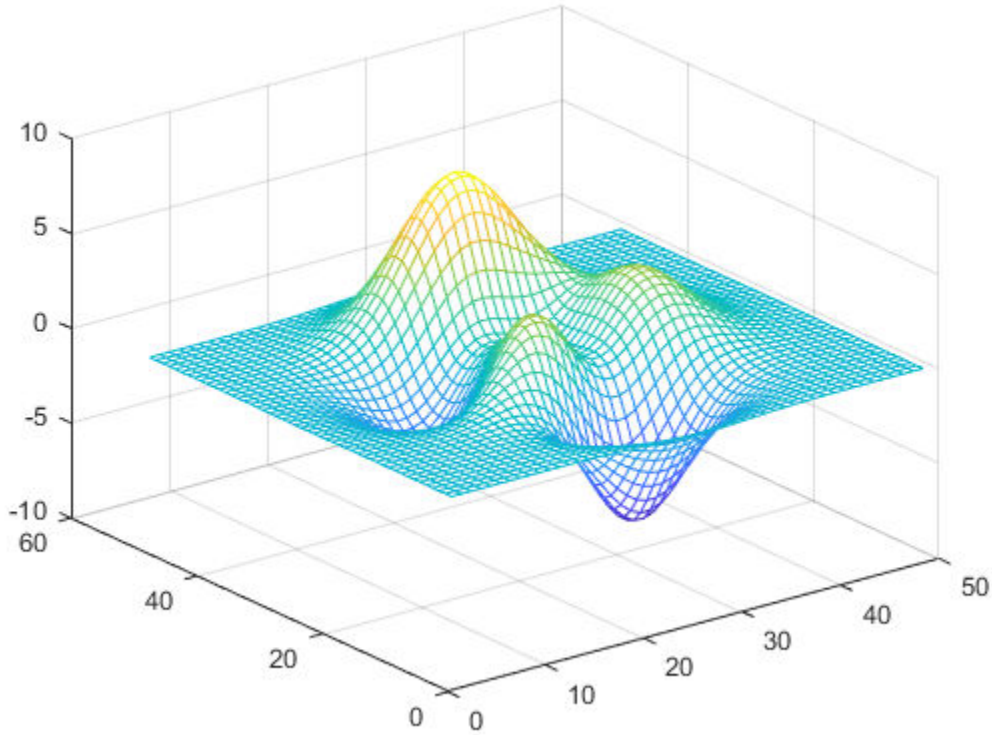
`hidden(ax,...)` modifies surface objects in the axes specified by `ax` instead of the current axes.

Examples

Show Obscured Lines

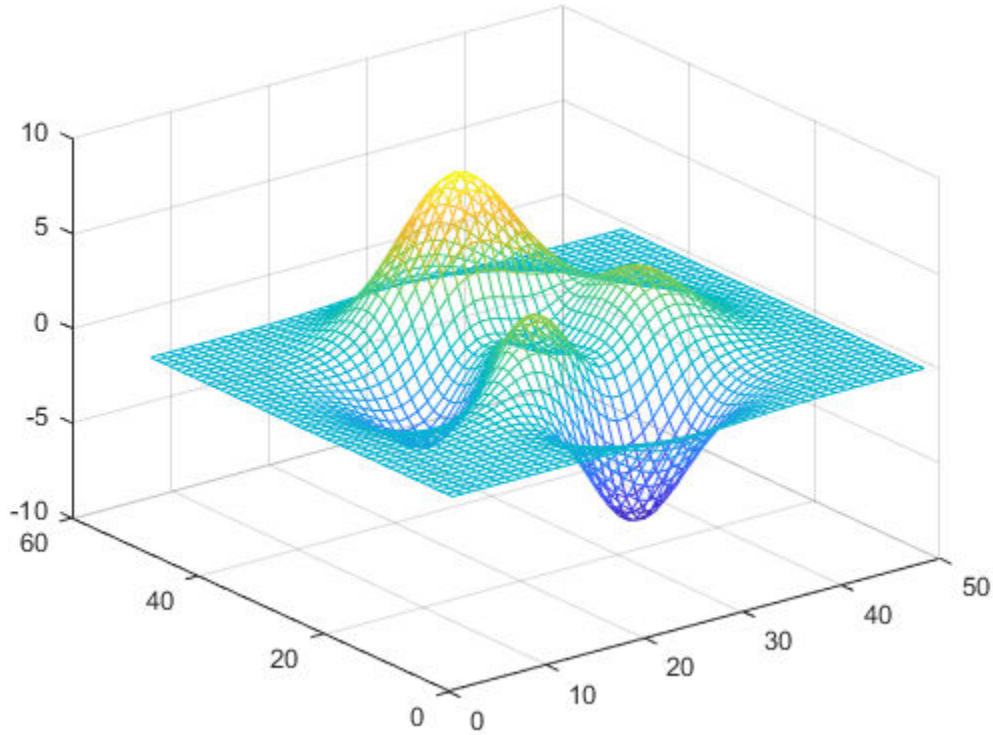
Create a mesh plot of the `peaks` function.

```
figure
mesh(peaks)
```



By default, MATLAB® hides obscured lines from the view. Show the obscured parts of the mesh by setting the hidden line removal to `off`.

```
hidden off
```



Algorithms

When a surface graphics object has a uniform `FaceColor` matching the `Color` property of the axes, `hidden off` sets the `FaceColor` of the surface object to `'none'`.

`hidden on` sets the `FaceColor` property of such surface objects to match the axes `Color` property (or to match that of the figure, if axes `Color` is `'none'`).

See Also

`mesh` | `shading`

Introduced before R2006a

hilb

Hilbert matrix

Syntax

```
H = hilb(n)
H = hilb(n,classname)
```

Description

`H = hilb(n)` returns the Hilbert matrix of order `n`. The Hilbert matrix is a notable example of a poorly conditioned matrix. The elements of Hilbert matrices are given by $H(i,j) = 1/(i + j - 1)$.

`H = hilb(n,classname)` returns a matrix of class `classname`, which can be either `'single'` or `'double'`.

Examples

Fourth-Order Hilbert Matrix

Compute the fourth-order Hilbert matrix and its condition number to see that it is poorly conditioned.

```
H = hilb(4)
```

```
H = 4×4

    1.0000    0.5000    0.3333    0.2500
    0.5000    0.3333    0.2500    0.2000
    0.3333    0.2500    0.2000    0.1667
    0.2500    0.2000    0.1667    0.1429
```

```
cond(H)
```

```
ans = 1.5514e+04
```

Input Arguments

n — Matrix order
scalar, nonnegative integer

Matrix order, specified as a scalar, nonnegative integer.

Example: `hilb(10)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`classname` — Matrix class
`'double'` (default) | `'single'`

Matrix class, specified as either `'double'` or `'single'`.

Example: `hilb(10,'single')`

Data Types: `char`

References

[1] Forsythe, G. E. and C. B. Moler. *Computer Solution of Linear Algebraic Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1967.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`invhilb`

Introduced before R2006a

hist

Histogram plot (not recommended; use `histogram`)

Note `hist` is not recommended. Use `histogram` instead.

For more information, including suggestions on updating code, see “Replace Discouraged Instances of hist and histc”.

Syntax

```
hist(x)
hist(x,nbins)
hist(x,xbins)

hist(ax, ___ )

counts = hist( ___ )
[counts,centers] = hist( ___ )
```

Description

`hist(x)` creates a histogram bar chart of the elements in vector `x`. The elements in `x` are sorted into 10 equally spaced bins along the *x*-axis between the minimum and maximum values of `x`. `hist` displays bins as rectangles, such that the height of each rectangle indicates the number of elements in the bin.

If the input is a multi-column array, `hist` creates histograms for each column of `x` and overlays them onto a single plot.

If the input is of data type `categorical`, each bin is a category of `x`.

`hist(x,nbins)` sorts `x` into the number of bins specified by the scalar `nbins`.

`hist(x,xbins)` sorts `x` into bins with intervals or categories determined by the vector `xbins`.

- If `xbins` is a vector of evenly spaced values, then `hist` uses the values as the bin centers.
- If `xbins` is a vector of unevenly spaced values, then `hist` uses the midpoints between consecutive values as the bin edges.
- If `x` is of data type `categorical`, then `xbins` must be a categorical vector or cell array of character vectors that specifies categories. `hist` plots bars only for those categories.

The length of the vector `xbins` is equal to the number of bins.

`hist(ax, ___ )` plots into the axes specified by `ax` instead of into the current axes (`gca`). The option `ax` can precede any of the input argument combinations in the previous syntaxes.

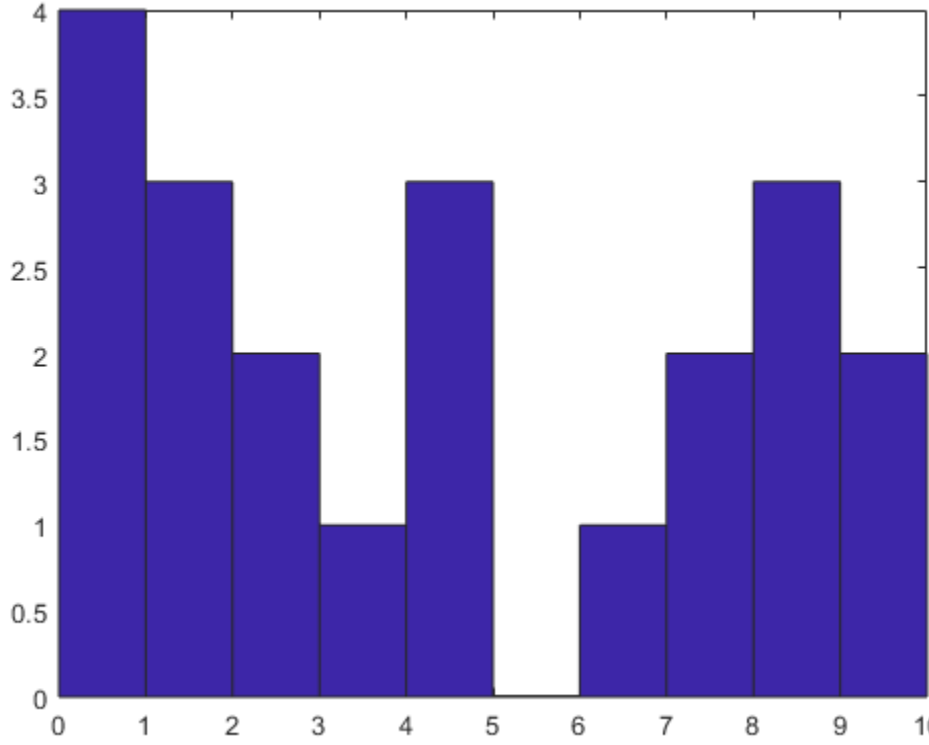
`counts = hist( ___ )` returns a row vector, `counts`, containing the number of elements in each bin.

`[counts,centers] = hist( ___ )` returns an additional row vector, `centers`, indicating the location of each bin center on the *x*-axis.

Examples

Histogram of Vector

```
x = [0 2 9 2 5 8 7 3 1 9 4 3 5 8 10 0 1 2 9 5 10];
hist(x)
```

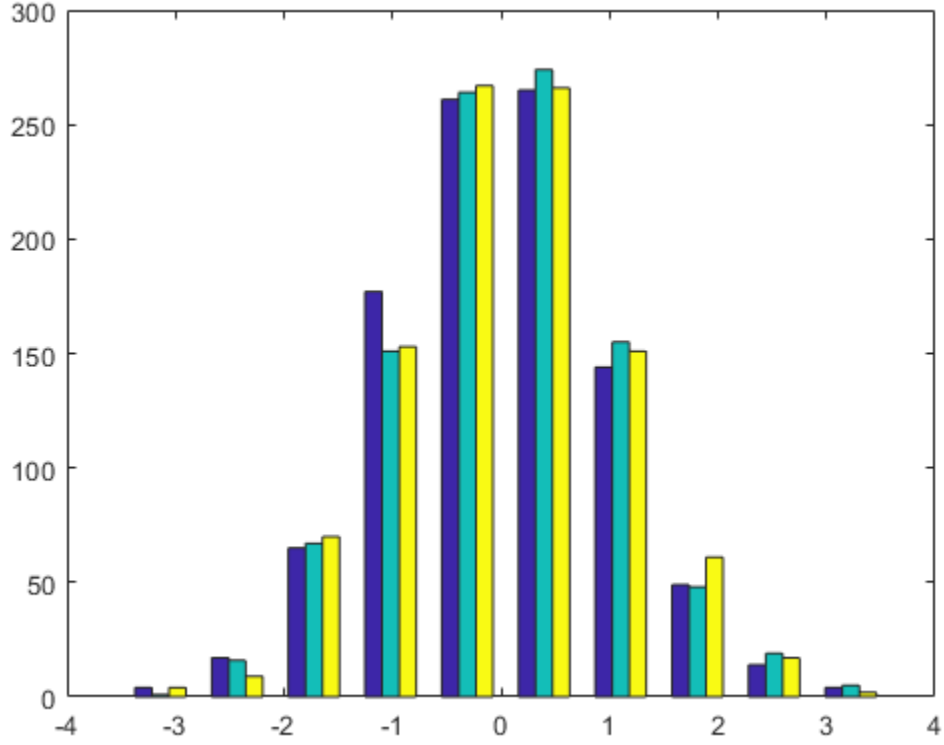


`hist` sorts the values in `x` among 10 equally spaced bins between the minimum and maximum values in the vector, which are 0 and 10 in this example.

Histogram of Multiple Columns

Generate three columns of 1,000 random numbers and plot the three column overlaid histogram.

```
x = randn(1000,3);
hist(x)
```

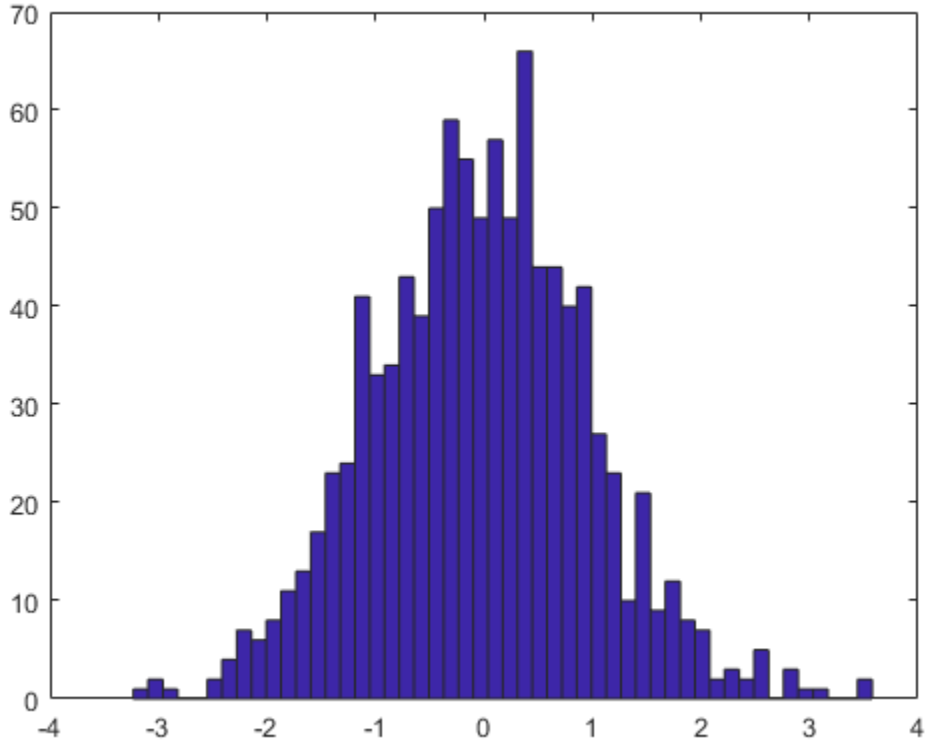


The values in `x` are sorted among 10 equally spaced bins between the minimum and maximum values. `hist` sorts and bins the columns of `x` separately and plots each column with a different color.

Specify Number of Histogram Bins

Plot a histogram of 1,000 random numbers sorted into 50 equally spaced bins.

```
x = randn(1000,1);
nbins = 50;
hist(x,nbins)
```



Use hist to Calculate Only

Generate 1,000 random numbers. Count how many numbers are in each of 10 equally spaced bins. Return the bin counts and bin centers.

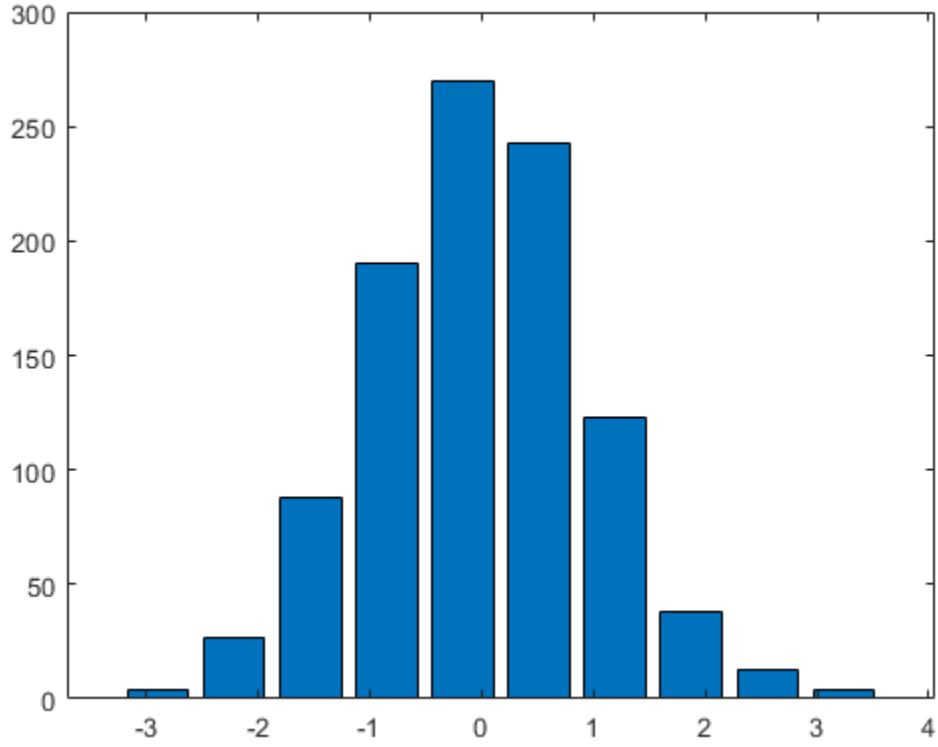
```
x = randn(1000,1);
[counts,centers] = hist(x)
```

```
counts = 1×10

     4    27    88   190   270   243   123    38    13     4
```

```
centers = 1×10

   -2.8915   -2.2105   -1.5294   -0.8484   -0.1673    0.5137    1.1947    1.8758    2.5
```

Use `bar` to plot the histogram.

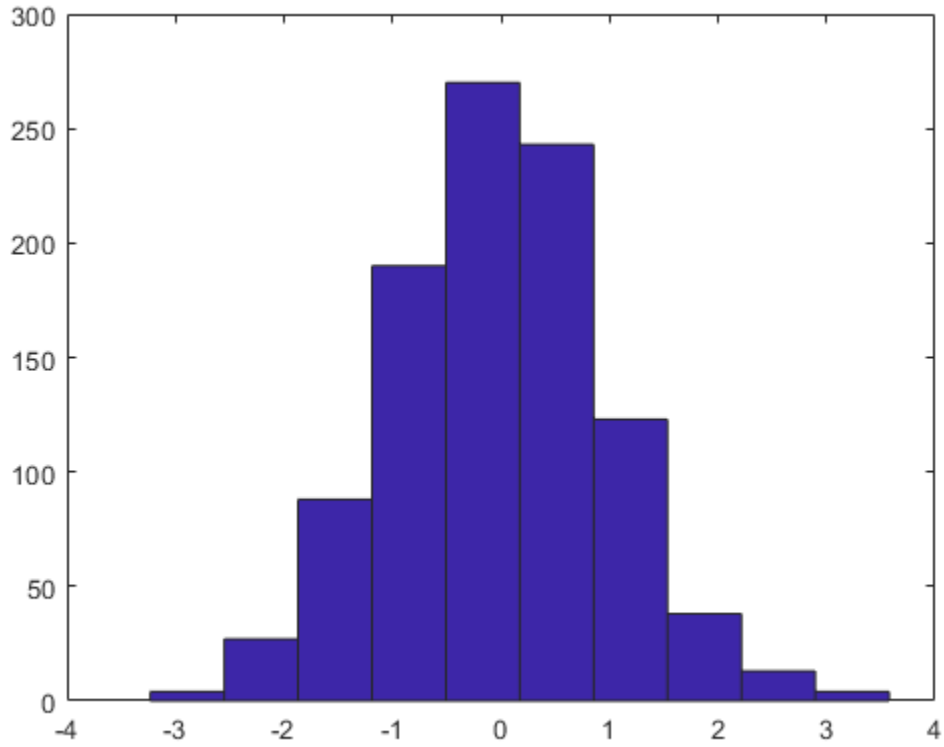
```
bar(centers,counts)
```



Specify Histogram Colors

Generate 1,000 random numbers and create a histogram.

```
data = randn(1000,1);
hist(data)
```

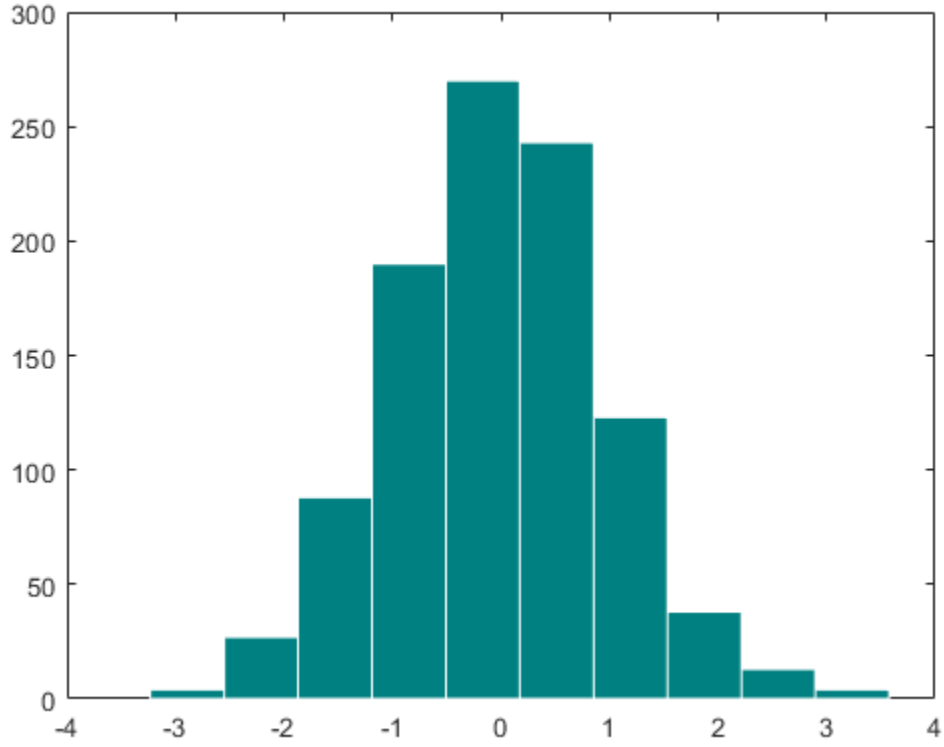


Get the handle to the patch object that creates the histogram plot.

```
h = findobj(gca,'Type','patch');
```

Set the face color of the bars plotted to an RGB triplet value of `[0 0.5 0.5]`. Set the edge color to white.

```
h.FaceColor = [0 0.5 0.5];
h.EdgeColor = 'w';
```



Input Arguments

x — Input array
vector or matrix

Input vector or matrix.

- If `x` is a vector, then `hist` creates one histogram.
- If `x` is a matrix, then `hist` creates a separate histogram for each column and plots the histograms using different colors.

If the input array contains `NaN`s or undefined categorical values, `hist` does not include these values in the bin counts.

If the input array contains the infinite values `-Inf` or `Inf`, then `hist` sorts `-Inf` into the first bin and `Inf` into the last bin. If you do not specify the bin intervals, then `hist` calculates the bin intervals using only the finite values in the input array.

Data Types: `single|double|logical|categorical`

`nbins` — Number of bins
10 (default) | scalar

Number of bins. Input `x` must be numeric, not categorical.

Data Types: `single|double|int8|int16|int32|int64|uint8|uint16|uint32|uint64`

`xbins` — Bin locations or categories
vector

Bin locations or categories, specified as a vector.

If `x` is numeric or logical, then `xbins` must be of type `single` or `double`.

- If the elements in `xbins` are equally spaced, then these elements are the bin centers.
- If the elements in `xbins` are not equally spaced, then these elements are indicated by markers along the *x*-axis, but are not the actual bin centers. Instead, `hist` calculates the bin edges as the midpoints between consecutive elements in vector `xbins`. To specify the bin edges directly, use `histc`.
- `xbins` must contain only finite values. The first and last bins extend to cover the minimum and maximum values in `x`.

If `x` is categorical, then `xbins` must be a categorical vector or cell array of character vectors that specifies categories. `hist` plots bars only for those categories specified by `xbins`.

`ax` — Axes object
axes object

Axes object. Use `ax` to plot the histogram in a specific axes instead of the current axes (`gca`).

Output Arguments

`counts` — Counts of the number of elements in each bin
row vector

Counts of the number of elements in each bin, returned as a row vector.

`centers` — Bin centers or categories
vector

Bin centers or categories, returned as a vector. If used with the syntax `[counts,centers] = hist(x,xbins)`, then the `centers` output has the same elements as the `xbins` input.

- If `x` is numeric or logical, then `centers` is a numeric row vector.
- If `x` is categorical, then `centers` is a cell array of character vectors.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support creation of histogram bar charts. Call `hist` with at least one output argument.
- If you supply `nbins`, then it must be a constant, scalar value.
- The values in the input array must be real.
- The orientation of vector outputs might not match the orientation in MATLAB.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bar` | `histc` | `histcounts` | `histogram` | `mode` | `patch` | `rose` | `stairs`

Introduced before R2006a

histc

Histogram bin counts (not recommended; use `histcounts`)

Note `histc` is not recommended. Use `histcounts` instead.

For more information, including suggestions on updating code, see “Replace Discouraged Instances of hist and histc”.

Syntax

```
bincounts = histc(x,binranges)
bincounts = histc(x,binranges,dim)
[bincounts,ind]= histc( ___ )
```

Description

`bincounts = histc(x,binranges)` counts the number of values in `x` that are within each specified bin range. The input, `binranges`, determines the endpoints for each bin. The output, `bincounts`, contains the number of elements from `x` in each bin.

- If `x` is a vector, then `histc` returns `bincounts` as a vector of histogram bin counts.
- If `x` is a matrix, then `histc` operates along each column of `x` and returns `bincounts` as a matrix of histogram bin counts for each column.

To plot the histogram, use `bar(binranges,bincounts,'histc')`.

`bincounts = histc(x,binranges,dim)` operates along the dimension `dim`.

`[bincounts,ind]= histc( ___ )` returns `ind`, an array the same size as `x` indicating the bin number that each entry in `x` sorts into. Use this syntax with any of the previous input argument combinations.

Examples

Create Histogram Plot

Initialize the random number generator to make the output of `randn` repeatable.

```
rng(0,'twister')
```

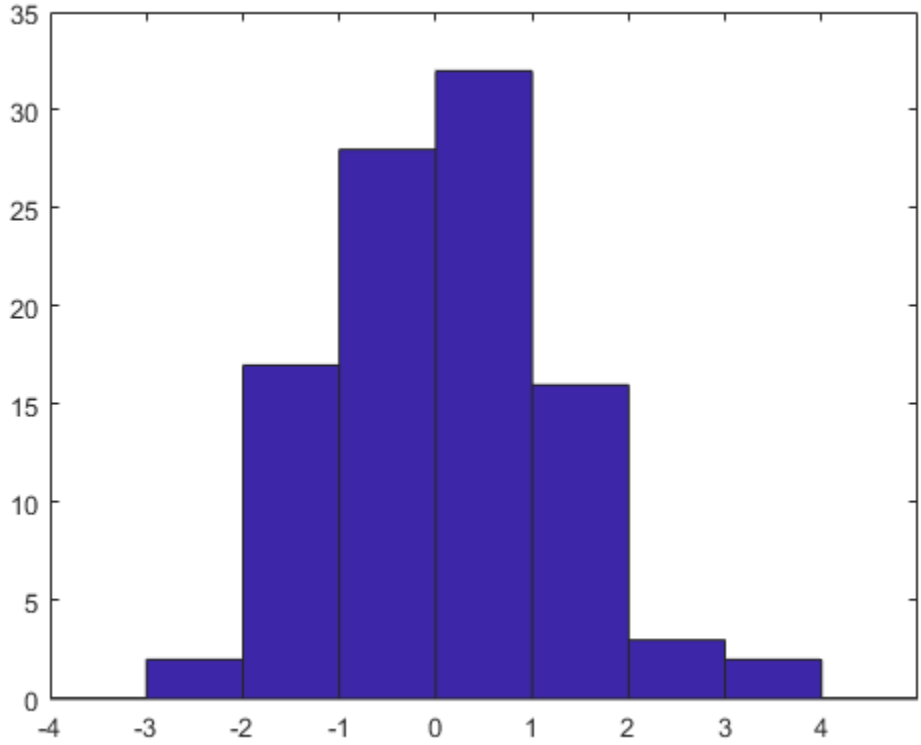
Define `x` as 100 normally distributed random numbers. Define bin ranges between -4 and 4. Determine the number of values in `x` that are within each specified bin range. Return the number of elements in each bin in `bincounts`.

```
x = randn(100,1);
binranges = -4:4;
[bincounts] = histc(x,binranges)
```

```
bincounts = 9×1

     0
     2
    17
    28
    32
    16
     3
     2
     0
```

To plot the histogram, use the `bar` function.

```
figure
bar(binranges,bincounts,'histc')
```



Return Bin Numbers for Histogram

Defined `ages` as a vector of ages. Sort `ages` into bins with varying ranges between 0 and 75.

```
ages = [3,12,24,15,5,74,23,54,31,23,64,75];
binranges = [0,10,25,50,75];

[bincounts,ind] = histc(ages,binranges)
```

```
bincounts = 1×5
```

```
     2     5     1     3     1

ind = 1×12

     1     2     2     2     1     4     2     4     3     2     4     5
```

`bincounts` contains the number of values in each bin. `ind` indicates the bin numbers.

Input Arguments

x — Values to be sorted
vector | matrix

Values to be sorted, specified as a vector or a matrix. The bin counts do not include values in `x` that are `NaN` or that lie outside the specified bin ranges. If `x` contains complex values, then `histc` ignores the imaginary parts and uses only the real parts.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`binranges` — Bin ranges
vector | matrix

Bin ranges, specified as a vector of monotonically nondecreasing values or a matrix of monotonically nondecreasing values running down each successive column. The values in `binranges` determine the left and right endpoints for each bin. If `binranges` contains complex values, then `histc` ignores the imaginary parts and uses only the real parts.

If `binranges` is a matrix, then `histc` determines the bin ranges by using values running down successive columns. Each bin includes the left endpoint, but does not include the right endpoint. The last bin consists of the scalar value equal to last value in `binranges`.

For example, if `binranges` equals the vector `[0,5,10,13]`, then `histc` creates four bins. The first bin includes values greater than or equal to 0 and strictly less than 5. The second bin includes values greater than or equal to 5 and less than 10, and so on. The last bin contains the scalar value 13.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`dim` — Dimension along which to operate
scalar

Dimension along which to operate, specified as a scalar.

Output Arguments

`bincounts` — Number of elements in each bin
vector | matrix

Number of elements in each bin, returned as a vector or a matrix. The last entry in `bincounts` is the number of values in `x` that equal the last entry in `binranges`.

`ind` — Bin index numbers
vector | matrix

Bin index numbers, returned as a vector or a matrix that is the same size as `x`.

Tips

- If values in `x` lie outside the specified bin ranges, then `histc` does not include these values in the bin counts. Start and end the `binranges` vector with `-inf` and `inf` to ensure that all values in `x` are included in the bin counts.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The output of a variable-size array that becomes a column vector at run time is a column-vector, not a row-vector.
- If supplied, `dim` must be a constant.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

See Also

`bar` | `hist` | `histcounts` | `histogram` | `mode`

Introduced before R2006a

histcounts

Histogram bin counts

Syntax

```
[N,edges] = histcounts(X)
[N,edges] = histcounts(X,nbins)
[N,edges] = histcounts(X,edges)
[N,edges,bin] = histcounts( ___ )

N = histcounts(C)
N = histcounts(C,Categories)
[N,Categories] = histcounts( ___ )

[ ___ ] = histcounts( ___ ,Name,Value)
```

Description

`[N,edges] = histcounts(X)` partitions the `X` values into bins, and returns the count in each bin, as well as the bin edges. The `histcounts` function uses an automatic binning algorithm that returns bins with a uniform width, chosen to cover the range of elements in `X` and reveal the underlying shape of the distribution.

`[N,edges] = histcounts(X,nbins)` uses a number of bins specified by the scalar, `nbins`.

`[N,edges] = histcounts(X,edges)` sorts `X` into bins with the bin edges specified by the vector, `edges`. The value `X(i)` is in the `k`th bin if `edges(k)` ≤ `X(i)` < `edges(k+1)`. The last bin also includes the right bin edge, so that it contains `X(i)` if `edges(end-1)` ≤ `X(i)` ≤ `edges(end)`.

`[N,edges,bin] = histcounts( ___ )` also returns an index array, `bin`, using any of the previous syntaxes. `bin` is an array of the same size as `X` whose elements are the bin indices for the corresponding elements in `X`. The number of elements in the `k`th bin is `nnz(bin==k)`, which is the same as `N(k)`.

`N = histcounts(C)`, where `C` is a categorical array, returns a vector, `N`, that indicates the number of elements in `C` whose value is equal to each of `C`'s categories. `N` has one element for each category in `C`.

`N = histcounts(C,Categories)` counts only the elements in `C` whose value is equal to the subset of categories specified by `Categories`.

`[N,Categories] = histcounts( ___ )` also returns the categories that correspond to each count in `N` using either of the previous syntaxes for categorical arrays.

`[ ___ ] = histcounts( ___ ,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments using any of the input or output argument combinations in previous syntaxes. For example, you can specify `'BinWidth'` and a scalar to adjust the width of the bins for numeric data. For categorical data, you can specify `'Normalization'` and either `'count'`, `'countdensity'`, `'probability'`, `'pdf'`, `'cumcount'`, or `'cdf'`.

Examples

Bin Counts and Bin Edges

Distribute 100 random values into bins. `histcounts` automatically chooses an appropriate bin width to reveal the underlying distribution of the data.

```
X = randn(100,1);
[N,edges] = histcounts(X)
```

```
N = 1×7

     2    17    28    32    16     3     2


edges = 1×8

    -3    -2    -1     0     1     2     3     4
```

Specify Number of Bins

Distribute 10 numbers into 6 equally spaced bins.

```
X = [2 3 5 7 11 13 17 19 23 29];
[N,edges] = histcounts(X,6)
```

```
N = 1×6

     2     2     2     2     1     1


edges = 1×7

         0    4.9000    9.8000   14.7000   19.6000   24.5000   29.4000
```

Specify Bin Edges

Distribute 1,000 random numbers into bins. Define the bin edges with a vector, where the first element is the left edge of the first bin, and the last element is the right edge of the last bin.

```
X = randn(1000,1);
edges = [-5 -4 -2 -1 -0.5 0 0.5 1 2 4 5];
N = histcounts(X,edges)
```

```
N = 1×10

     0    24   149   142   195   200   154   111    25     0
```

Normalized Bin Counts

Distribute all of the prime numbers less than 100 into bins. Specify `'Normalization'` as `'probability'` to normalize the bin counts so that `sum(N)` is `1`. That is, each bin count represents the probability that an observation falls within that bin.

```
X = primes(100);
[N,edges] = histcounts(X, 'Normalization', 'probability')
```

```
N = 1×4

    0.4000    0.2800    0.2800    0.0400


edges = 1×5

     0    30    60    90   120
```

Determine Bin Placement

Distribute 100 random integers between -5 and 5 into bins, and specify `'BinMethod'` as `'integers'` to use unit-width bins centered on integers. Specify a third output for `histcounts` to return a vector representing the bin indices of the data.

```
X = randi([-5,5],100,1);
[N,edges,bin] = histcounts(X,'BinMethod','integers');
```

Find the bin count for the third bin by counting the occurrences of the number `3` in the bin index vector, `bin`. The result is the same as `N(3)`.

```
count = nnz(bin==3)
```

```
count = 8
```

Categorical Bin Counts

Create a categorical vector that represents votes. The categories in the vector are `'yes'`, `'no'`, or `'undecided'`.

```
A = [0 0 1 1 1 0 0 0 0 NaN NaN 1 0 0 0 1 0 1 0 1 0 0 0 1 1 1 1];
C = categorical(A,[1 0 NaN],{'yes','no','undecided'})
```

```
C = 1x27 categorical array
  Columns 1 through 9

     no      no      yes      yes      yes      no      no      no      no
```

```
  Columns 10 through 16

     undecided      undecided      yes      no      no      no      yes

  Columns 17 through 25

     no      yes      no      yes      no      no      no      yes      yes

  Columns 26 through 27

     yes      yes
```

Determine the number of elements that fall into each category.

```
[N,Categories] = histcounts(C)
```

```
N = 1×3

    11    14     2


Categories = 1x3 cell array
    {'yes'}    {'no'}    {'undecided'}
```

Input Arguments

X — Data to distribute among bins
vector | matrix | multidimensional array

Data to distribute among bins, specified as a vector, matrix, or multidimensional array. If `X` is not a vector, then `histcounts` treats it as a single column vector, `X(:)`.

`histcounts` ignores all `NaN` values. Similarly, `histcounts` ignores `Inf` and `-Inf` values unless the bin edges explicitly specify `Inf` or `-Inf` as a bin edge.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `datetime` | `duration`

C — Categorical data
categorical array

Categorical data, specified as a categorical array. `histcounts` ignores undefined categorical values.

Data Types: `categorical`

`nbins` — Number of bins
positive integer

Number of bins, specified as a positive integer. If you do not specify `nbins`, then `histcounts` automatically calculates how many bins to use based on the values in `X`.

Example: `[N,edges] = histcounts(X,15)` uses 15 bins.

`edges` — Bin edges
vector

Bin edges, specified as a vector. `edges(1)` is the left edge of the first bin, and `edges(end)` is the right edge of the last bin.

For datetime and duration data, `edges` must be a datetime or duration vector in monotonically increasing order.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `datetime` | `duration`

`Categories` — Categories included in count
all categories (default) | cell vector of character vectors | categorical vector

Categories included in count, specified as a cell vector of character vectors or a categorical vector. By default, `histcounts` uses a bin for each category in categorical array `C`. Use `Categories` to specify a unique subset of the categories instead.

Example: `h = histcounts(C,{'Large','Small'})` counts only the categorical data in the categories `'Large'` and `'Small'`.

Data Types: `cell` | `categorical`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[N,edges] = histcounts(X,'Normalization','probability')` normalizes the bin counts in `N`, such that `sum(N)` is 1.

`BinLimits` — Bin limits
two-element vector

Bin limits, specified as a two-element vector, `[bmin,bmax]`. This option bins only the values in `X` that fall between `bmin` and `bmax` inclusive; that is, `X(X>=bmin & X<=bmax)`.

This option does not apply to categorical data.

Example: `[N,edges] = histcounts(X,'BinLimits',[1,10])` bins only the values in `X` that are between `1` and `10` inclusive.

`BinMethod` — Binning algorithm
`'auto'` (default) | `'scott'` | `'fd'` | `'integers'` | `'sturges'` | `'sqrt'` | ...

Binning algorithm, specified as one of the values in this table.

Value	Description
`'auto'`	The default `'auto'` algorithm chooses a bin width to cover the data range and reveal the shape of the underlying distribution.
`'scott'`	Scott's rule is optimal if the data is close to being normally distributed, but is also appropriate for most other distributions. It uses a bin width of `3.5*std(X(:))*numel(X)^(-1/3)`.
`'fd'`	The Freedman-Diaconis rule is less sensitive to outliers in the data, and may be more suitable for data with heavy-tailed distributions. It uses a bin width of `2*IQR(X(:))*numel(X)^(-1/3)`, where `IQR` is the interquartile range of `X`.

Value	Description
`'integers'`	The integer rule is useful with integer data, as it creates a bin for each integer. It uses a bin width of 1 and places bin edges halfway between integers. To prevent from accidentally creating too many bins, a limit of 65536 bins (2^{16}) can be created with this rule. If the data range is greater than 65536, then wider bins are used instead. **Note** `'integers'` does not support datetime or duration data.
`'sturges'`	Sturges' rule is a simple rule that is popular due to its simplicity. It chooses the number of bins to be `ceil(1 + log2(numel(X)))`.
`'sqrt'`	The Square Root rule is another simple rule widely used in other software packages. It chooses the number of bins to be `ceil(sqrt(numel(X)))`.

`histcounts` does not always choose the number of bins using these exact formulas. Sometimes the number of bins is adjusted slightly so that the bin edges fall on "nice" numbers.

For datetime data, the bin method can be one of these units of time:

`'second'`	`'month'`
`'minute'`	`'quarter'`
`'hour'`	`'year'`
`'day'`	`'decade'`
`'week'`	`'century'`

For duration data, the bin method can be one of these units of time:

`'second'`	`'day'`

'minute'	'year'
'hour'	

If you specify BinMethod with datetime or duration data, then histcounts can use a maximum of 65,536 bins (or 2^{16}). If the specified bin duration requires more bins, then histcounts uses a larger bin width corresponding to the maximum number of bins.

This option does not apply to categorical data.

Example: [N,edges] = histcounts(X,'BinMethod','integers') uses bins centered on integers.

BinWidth — Width of bins
scalar

Width of bins, specified as a scalar. If you specify BinWidth, then histcounts can use a maximum of 65,536 bins (or 2^{16}). If the specified bin width requires more bins, then histcounts uses a larger bin width corresponding to the maximum number of bins.

For datetime and duration data, the value of 'BinWidth' can be a scalar duration or calendar duration.

This option does not apply to categorical data.

Example: [N,edges] = histcounts(X,'BinWidth',5) uses bins with a width of 5.

Normalization — Type of normalization
'count' (default) | 'probability' | 'countdensity' | 'pdf' | 'cumcount' | 'cdf'

Type of normalization, specified as one of the values in this table. For each bin i:

- v_i is the bin value.
- c_i is the number of elements in the bin.
- w_i is the width of the bin.
- N is the number of elements in the input data. This value can be greater than the binned data if the data contains NaN, NaT, or <undefined> values, or if some of the data lies outside the bin limits.

Value	Bin Values	Notes
`'count'` (default)	$v_i = c_i$	• Count or frequency of observations. • Sum of bin values is less than or equal to `numel(X)`. The sum is less than `numel(X)` only when some of the input data is not included in the bins. • For categorical data, sum of bin values is either `numel(X)` or `sum(ismember(X(:),Categories))`.
`'countdensity'`	$v_i = \frac{c_i}{w_i}$	• Count or frequency scaled by width of bin. • For categorical data, this the same as `'count'`. **Note** `'countdensity'` does not support datetime or duration data.

Value	Bin Values	Notes
`'cumcount'`	$v_i = \sum_{j=1}^{i} c_j$	• Cumulative count. Each bin value is the cumulative number of observations in that bin and all previous bins. • The value of the last bin is less than or equal to `numel(X)`. • For categorical data, the value of the last bin is less than or equal to `numel(X)` or `sum(ismember(X(:),Categories))`.
`'probability'`	$v_i = \frac{c_i}{N}$	• Relative probability. • The sum of the bin values is less than or equal to `1`.
`'pdf'`	$v_i = \frac{c_i}{N \cdot w_i}$	• Probability density function estimate. • For categorical data, this is the same as `'probability'`. **Note** `'pdf'` does not support datetime or duration data.
`'cdf'`	$v_i = \sum_{j=1}^{i} \frac{c_j}{N}$	• Cumulative density function estimate. • `N(end)` is less than or equal to `1`.

Example: `[N,edges] = histcounts(X,'Normalization','pdf')` bins the data using the probability density function estimate.

Output Arguments

`N` — Bin counts
row vector

Bin counts, returned as a row vector.

`edges` — Bin edges
vector

Bin edges, returned as a vector. `edges(1)` is the left edge of the first bin, and `edges(end)` is the right edge of the last bin.

`bin` — Bin indices
array

Bin indices, returned as an array of the same size as `X`. Each element in `bin` describes which numbered bin contains the corresponding element in `X`.

A value of `0` in `bin` indicates an element which does not belong to any of the bins (for example, a `NaN` value).

`Categories` — Categories included in count
cell vector of character vectors

Categories included in count, returned as a cell vector of character vectors. `Categories` contains the categories in `C` that correspond to each count in `N`.

Tips

- The behavior of `histcounts` is similar to that of the `discretize` function. Use `histcounts` to find the number of elements in each bin. On the other hand, use `discretize` to find which bin each element belongs to (without counting).

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- Some input options are not supported. The allowed options are:
  - `'BinWidth'`
  - `'BinLimits'`
  - `'Normalization'`
  - `'BinMethod'` — The `'auto'` and `'scott'` bin methods are the same. The `'fd'` bin method is not supported.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.
- If you do not supply bin edges, then code generation might require variable-size arrays and dynamic memory allocation.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- 64-bit integers are not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`discretize` | `histcounts2` | `histogram` | `histogram2`

Topics

"Replace Discouraged Instances of hist and histc"

Introduced in R2014b

histcounts2

Bivariate histogram bin counts

Syntax

```
[N,Xedges,Yedges] = histcounts2(X,Y)
[N,Xedges,Yedges] = histcounts2(X,Y,nbins)
[N,Xedges,Yedges] = histcounts2(X,Y,Xedges,Yedges)
[N,Xedges,Yedges] = histcounts2( ___ ,Name,Value)
[N,Xedges,Yedges,binX,binY] = histcounts2( ___ )
```

Description

`[N,Xedges,Yedges] = histcounts2(X,Y)` partitions the values in `X` and `Y` into 2-D bins, and returns the bin counts, as well as the bin edges in each dimension. The `histcounts2` function uses an automatic binning algorithm that returns uniform bins chosen to cover the range of values in `X` and `Y` and reveal the underlying shape of the distribution.

`[N,Xedges,Yedges] = histcounts2(X,Y,nbins)` specifies the number of bins to use in each dimension.

`[N,Xedges,Yedges] = histcounts2(X,Y,Xedges,Yedges)` partitions `X` and `Y` into bins with the bin edges specified by `Xedges` and `Yedges`.

`N(i,j)` counts the value `[X(k),Y(k)]` if `Xedges(i)` ≤ `X(k)` < `Xedges(i+1)` and `Yedges(j)` ≤ `Y(k)` < `Yedges(j+1)`. The last bins in each dimension also include the last (outer) edge. For example, `[X(k),Y(k)]` falls into the `i`th bin in the last row if `Xedges(end-1)` ≤ `X(k)` ≤ `Xedges(end)` and `Yedges(i)` ≤ `Y(k)` < `Yedges(i+1)`.

`[N,Xedges,Yedges] = histcounts2( ___ ,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments using any of the input arguments in previous syntaxes. For example, you can specify `'BinWidth'` and a two-element vector to adjust the width of the bins in each dimension.

`[N,Xedges,Yedges,binX,binY] = histcounts2( ___ )` also returns index arrays `binX` and `binY`, using any of the previous syntaxes. `binX` and `binY` are arrays of the

same size as X and Y whose elements are the bin indices for the corresponding elements in X and Y. The number of elements in the `(i,j)`th bin is equal to `nnz(binX==i & binY==j)`, which is the same as `N(i,j)` if `Normalization` is `'count'`.

Examples

Bin Counts and Bin Edges

Distribute 100 pairs of random numbers into bins. `histcounts2` automatically chooses an appropriate bin width to reveal the underlying distribution of the data.

```
x = randn(100,1);
y = randn(100,1);
[N,Xedges,Yedges] = histcounts2(x,y)
```

```
N = 7×6

     0     0     0     2     0     0
     1     2    10     4     0     0
     1     4     9     9     5     0
     1     4    10    11     5     1
     1     4     6     3     1     1
     0     0     1     2     0     0
     0     0     1     0     1     0


Xedges = 1×8

    -3    -2    -1     0     1     2     3     4


Yedges = 1×7

    -3    -2    -1     0     1     2     3
```

Specify Number of Bins in Each Dimension

Distribute 10 pairs of numbers into 12 bins. Specify 3 bins in the x-dimension, and 4 bins in the y-dimension.

```
x = [1 1 2 3 2 2 1 1 2 3];
y = [5 6 3 8 9 1 2 7 5 1];
nbins = [3 4];
[N,Xedges,Yedges] = histcounts2(x,y,nbins)
```

```
N = 3×4

     1     0     2     1
     1     1     1     1
     1     0     0     1


Xedges = 1×4

    0.6000    1.4000    2.2000    3.0000


Yedges = 1×5

         0    2.3000    4.6000    6.9000    9.2000
```

Specify Bin Edges

Distribute 1,000 pairs of random numbers into bins. Define the bin edges with two vectors: one each for the x and y dimensions. The first element in each vector specifies the first edge of the first bin, and the last element is the last edge of the last bin.

```
x = randn(1000,1);
y = randn(1000,1);
Xedges = -5:5;
Yedges = [-5 -4 -2 -1 -0.5 0 0.5 1 2 4 5];
N = histcounts2(x,y,Xedges,Yedges)
```

```
N = 10×10

     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     1     1     1     0     0     0
     0     0     5     5     3     5     1     2     0     0
     0     2    19    23    29    25    26    20     5     0
     0    10    36    51    59    71    54    46    10     0
     0     7    43    46    79    64    60    46     9     0
```

```
     0     3    12    18    21    23    19     9     6     0
     0     0     5     3     2     8     2     2     0     0
     0     0     0     1     1     1     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
```

Normalized Bin Counts

Distribute 1,000 pairs of random numbers into bins. Specify `Normalization` as `'probability'` to normalize the bin counts such that `sum(N(:))` is `1`. That is, each bin count represents the probability that an observation falls within that bin.

```
x = randn(1000,1);
y = randn(1000,1);
[N,Xedges,Yedges] = histcounts2(x,y,6,'Normalization','probability')
```

```
N = 6×6

         0         0    0.0020    0.0020         0         0
         0    0.0110    0.0320    0.0260    0.0070    0.0010
    0.0010    0.0260    0.1410    0.1750    0.0430    0.0060
         0    0.0360    0.1620    0.1940    0.0370    0.0040
         0    0.0040    0.0300    0.0370    0.0100    0.0010
         0    0.0030    0.0040    0.0040    0.0010         0


Xedges = 1×7

   -4.0000   -2.7000   -1.4000   -0.1000    1.2000    2.5000    3.8000


Yedges = 1×7

   -4.0000   -2.7000   -1.4000   -0.1000    1.2000    2.5000    3.8000
```

Determine Bin Placement

Distribute 1,000 random integer pairs between -10 and 10 into bins, and specify `BinMethod` as `'integers'` to use unit-width bins centered on integers. Specify five outputs for `histcounts2` to return vectors representing the bin placement of the data.

```
x = randi([-10,10],1000,1);
y = randi([-10,10],1000,1);
[N,Xedges,Yedges,binX,binY] = histcounts2(x,y,'BinMethod','integers');
```

Determine which bin the value `(x(3),y(3))` falls into.

```
[x(3),y(3)]
```

```
ans = 1×2

    -8    10
```

```
bin = [binX(3) binY(3)]
```

```
bin = 1×2

     3    21
```

Input Arguments

`X,Y` — Data to distribute among bins (as separate arguments)
vectors | matrices | multidimensional arrays

Data to distribute among bins, specified as separate arguments of vectors, matrices, or multidimensional arrays. `X` and `Y` must have the same size.

Corresponding elements in `X` and `Y` specify the *x* and *y* coordinates of 2-D data points, `[X(k),Y(k)]`. The data types of `X` and `Y` can be different.

`histcounts2` ignores all `NaN` values. Similarly, `histcounts2` ignores `Inf` and `-Inf` values unless the bin edges explicitly specify `Inf` or `-Inf` as a bin edge.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

nbins — Number of bins in each dimension
scalar | vector

Number of bins in each dimension, specified as a positive scalar integer or two-element vector of positive integers. If you do not specify `nbins`, then `histcounts2` automatically calculates how many bins to use based on the values in `X` and `Y`:

- If `nbins` is a scalar, then `histcounts2` uses that many bins in each dimension.
- If `nbins` is a vector, then `nbins(1)` specifies the number of bins in the *x*-dimension and `nbins(2)` specifies the number of bins in the *y*-dimension.

Example: `[N,Xedges,Yedges] = histcounts2(X,Y,[15 20])` uses 15 bins in the `x`-dimension and 20 bins in the `y`-dimension.

Xedges — Bin edges in *x*-dimension
vector

Bin edges in *x*-dimension, specified as a vector. `Xedges(1)` is the first edge of the first bin in the *x*-dimension, and `Xedges(end)` is the outer edge of the last bin.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Yedges — Bin edges in *y*-dimension
vector

Bin edges in *y*-dimension, specified as a vector. `Yedges(1)` is the first edge of the first bin in the *y*-dimension, and `Yedges(end)` is the outer edge of the last bin.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[N,Xedges,Yedges] = histcounts2(X,Y,'Normalization','probability')` normalizes the bin counts in `N`, such that `sum(N)` is 1.

`BinMethod` — Binning algorithm

`'auto'` (default) | `'scott'` | `'fd'` | `'integers'`

Binning algorithm, specified as one of the values in this table.

Value	Description
`'auto'`	The default `'auto'` algorithm chooses a bin width to cover the data range and reveal the shape of the underlying distribution.
`'scott'`	Scott's rule is optimal if the data is close to being jointly normally distributed. This rule is appropriate for most other distributions, as well. It uses a bin size of `[3.5*std(X(:))*numel(X)^(-1/4), 3.5*std(Y(:))*numel(Y)^(-1/4)]`.
`'fd'`	The Freedman-Diaconis rule is less sensitive to outliers in the data, and might be more suitable for data with heavy-tailed distributions. It uses a bin size of `[2*IQR(X(:))*numel(X)^(-1/4), 2*IQR(Y(:))*numel(Y)^(-1/4)]`, where `IQR` is the interquartile range.
`'integers'`	The integer rule is useful with integer data, as it creates bins centered on pairs of integers. It uses a bin width of 1 for each dimension and places bin edges halfway between integers. To avoid accidentally creating too many bins, you can use this rule to create a limit of 1024 bins (2^{10}). If the data range for either dimension is greater than 1024, then the integer rule uses wider bins instead.

`histcounts2` does not always choose the number of bins using these exact formulas. Sometimes the number of bins is adjusted slightly so that the bin edges fall on "nice" numbers.

Example: `[N,Xedges,Yedges] = histcounts2(X,Y,'BinMethod','integers')` uses 2-D bins centered on each pair of integers.

`BinWidth` — Width of bins in each dimension
vector

Width of bins in each dimension, specified as a two-element vector of positive integers, `[xWidth yWidth]`.

If you specify `BinWidth`, then `histcounts2` can use a maximum of 1024 bins (2^{10}) along each dimension. If the specified bin width requires more bins, then `histcounts2` uses a larger bin width corresponding to the maximum number of bins.

Example: `[N,Xedges,Yedges] = histcounts2(X,Y,'BinWidth',[5 10])` uses bins with size `5` in the `x`-dimension and size `10` in the `y`-dimension.

`XBinLimits` — Bin limits in *x*-dimension
two-element vector

Bin limits in *x*-dimension, specified as a two-element vector, `[xbmin,xbmax]`. The vector indicates the first and last bin edges in the *x*-dimension.

This option only bins data that falls within the bin limits inclusively, `X>=xbmin & X<=xbmax`.

`YBinLimits` — Bin limits in *y*-dimension
two-element vector

Bin limits in *y*-dimension, specified as a two-element vector, `[ybmin,ybmax]`. The vector indicates the first and last bin edges in the *y*-dimension.

This option only bins data that falls within the bin limits inclusively, `Y>=ybmin & Y<=ybmax`.

`Normalization` — Type of normalization
`'count'` (default) | `'probability'` | `'countdensity'` | `'pdf'` | `'cumcount'` | `'cdf'`

Type of normalization, specified as one of the values in this table. For each bin `i`:

- v_i is the bin value.
- c_i is the number of elements in the bin.

- $A_i = w_{xi} \cdot w_{yi}$ is the area of each bin, computed using the x and y bin widths.
- N is the number of elements in the input data. This value can be greater than the binned data if the data contains `NaN` values, or if some of the data lies outside the bin limits.

Value	Bin Values	Notes
`'count'` (default)	$v_i = c_i$	• Count or frequency of observations. • Sum of bin values is less than or equal to `numel(X)` and `numel(y)`. The sum is less than `numel(X)` only when some of the input data is not included in the bins.
`'countdensity'`	$v_i = \frac{c_i}{A_i}$	• Count or frequency scaled by area of bin. • The sum of the bin volumes, (N value * Area of bin), is less than or equal to `numel(X)` and `numel(Y)`.
`'cumcount'`	$v_i = \sum_{j=1}^{i} c_j$	• Cumulative count. Each bin value is the cumulative number of observations in each bin and all previous bins in both the x and y dimensions. • `N(end,end)` is less than or equal to `numel(X)` and `numel(Y)`.
`'probability'`	$v_i = \frac{c_i}{N}$	• Relative probability. • `sum(N(:))` is less than or equal to `1`.

Value	Bin Values	Notes
`'pdf'`	$v_i = \frac{c_i}{N \cdot A_i}$	• Probability density function estimate. • The sum of the bin volumes, (N value * Area of bin), is less than or equal to 1.
`'cdf'`	$v_i = \sum_{j=1}^{i} \frac{c_j}{N}$	• Cumulative density function estimate. • `N(end,end)` is less than or equal to 1.

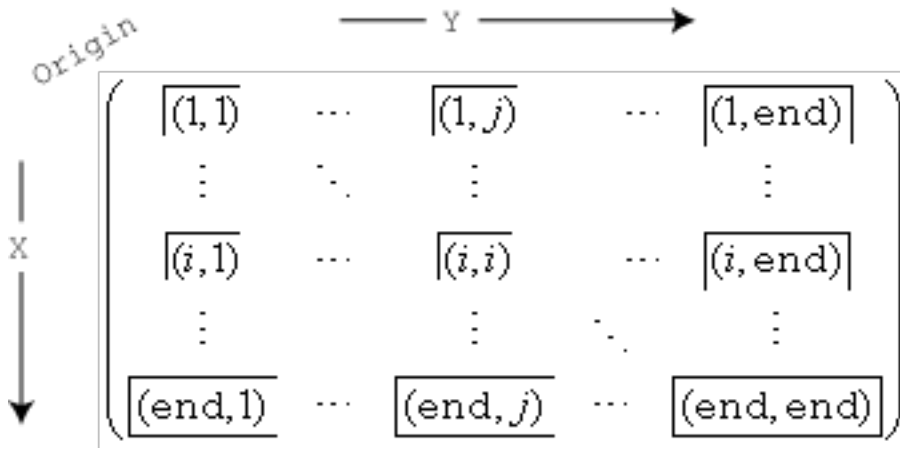
Example: `[N,Xedges,Yedges] = histcounts2(X,Y,'Normalization','pdf')` bins the data using the probability density function estimate for `X` and `Y`.

Output Arguments

`N` — Bin counts
array

Bin counts, returned as a numeric array.

The bin inclusion scheme for the different numbered bins in `N`, as well as their relative orientation to the *x*-axis and *y*-axis, is



For example, the `(1,1)` bin includes values that fall on the first edge in each dimension, and the last bin in the bottom right includes values that fall on any of its edges.

Xedges — Bin edges in *x*-dimension
vector

Bin edges in *x*-dimension, returned as a vector. `Xedges(1)` is the first bin edge in the *x*-dimension and `Xedges(end)` is the last bin edge.

Yedges — Bin edges in *y*-dimension
vector

Bin edges in *y*-dimension, returned as a vector. `Yedges(1)` is the first bin edge in the *y*-dimension and `Yedges(end)` is the last bin edge.

`binX` — Bin index in *x*-dimension
array

Bin index in *x*-dimension, returned as a numeric array of the same size as `X`. Corresponding elements in `binX` and `binY` describe which numbered bin contains the corresponding values in `X` and `Y`. A value of `0` in `binX` or `binY` indicates an element that does not belong to any of the bins (such as a `NaN` value).

For example, `binX(1)` and `binY(1)` describe the bin placement for the value `[X(1),Y(1)]`.

`binY` — Bin index in *y*-dimension
array

Bin index in *y*-dimension, returned as a numeric array of the same size as `Y`. Corresponding elements in `binX` and `binY` describe which numbered bin contains the corresponding values in `X` and `Y`. A value of `0` in `binX` or `binY` indicates an element that does not belong to any of the bins (such as a `NaN` value).

For example, `binX(1)` and `binY(1)` describe the bin placement for the value `[X(1),Y(1)]`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.
- If you do not supply bin edges, then code generation might require variable-size arrays and dynamic memory allocation.

See Also

`discretize` | `fewerbins` | `histcounts` | `histogram` | `histogram2` | `morebins`

Introduced in R2015b

histogram

Histogram plot

Description

Histograms are a type of bar plot for numeric data that group the data into bins. After you create a `Histogram` object, you can modify aspects of the histogram by changing its property values. This is particularly useful for quickly modifying the properties of the bins or changing the display.

Creation

Syntax

```
histogram(X)
histogram(X,nbins)
histogram(X,edges)
histogram('BinEdges',edges,'BinCounts',counts)
```

```
histogram(C)
histogram(C,Categories)
histogram('Categories',Categories,'BinCounts',counts)
```

```
histogram( ___ ,Name,Value)
histogram(ax, ___ )
h = histogram( ___ )
```

Description

`histogram(X)` creates a histogram plot of `X`. The `histogram` function uses an automatic binning algorithm that returns bins with a uniform width, chosen to cover the range of elements in `X` and reveal the underlying shape of the distribution. `histogram` displays the bins as rectangles such that the height of each rectangle indicates the number of elements in the bin.

`histogram(X,nbins)` uses a number of bins specified by the scalar, `nbins`.

`histogram(X,edges)` sorts `X` into bins with the bin edges specified by the vector, `edges`. Each bin includes the left edge, but does not include the right edge, except for the last bin which includes both edges.

`histogram('BinEdges',edges,'BinCounts',counts)` manually specifies bin edges and associated bin counts. `histogram` plots the specified bin counts and does not do any data binning.

`histogram(C)`, where `C` is a categorical array, plots a histogram with a bar for each category in `C`.

`histogram(C,Categories)` plots only the subset of categories specified by `Categories`.

`histogram('Categories',Categories,'BinCounts',counts)` manually specifies categories and associated bin counts. `histogram` plots the specified bin counts and does not do any data binning.

`histogram( ___ ,Name,Value)` specifies additional options with one or more `Name,Value` pair arguments using any of the previous syntaxes. For example, you can specify `'BinWidth'` and a scalar to adjust the width of the bins, or `'Normalization'` with a valid option (`'count'`, `'probability'`, `'countdensity'`, `'pdf'`, `'cumcount'`, or `'cdf'`) to use a different type of normalization. For a list of properties, see Histogram.

`histogram(ax, ___ )` plots into the axes specified by `ax` instead of into the current axes (`gca`). The option `ax` can precede any of the input argument combinations in the previous syntaxes.

`h = histogram( ___ )` returns a `Histogram` object. Use this to inspect and adjust the properties of the histogram. For a list of properties, see Histogram.

Input Arguments

X — Data to distribute among bins

vector | matrix | multidimensional array

Data to distribute among bins, specified as a vector, matrix, or multidimensional array. If `X` is not a vector, then `histogram` treats it as a single column vector, `X(:)`, and plots a single histogram.

`histogram` ignores all `NaN` and `NaT` values. Similarly, `histogram` ignores `Inf` and `-Inf` values, unless the bin edges explicitly specify `Inf` or `-Inf` as a bin edge. Although `NaN`, `NaT`, `Inf`, and `-Inf` values are typically not plotted, they are still included in normalization calculations that include the total number of data elements, such as `'probability'`.

Note If `X` contains integers of type `int64` or `uint64` that are larger than `flintmax`, then it is recommended that you explicitly specify the histogram bin edges. `histogram` automatically bins the input data using double precision, which lacks integer precision for numbers greater than `flintmax`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `datetime` | `duration`

`C` — Categorical data
categorical array

Categorical data, specified as a categorical array. `histogram` does not plot undefined categorical values. However, undefined categorical values are still included in normalization calculations that include the total number of data elements, such as `'probability'`.

Data Types: `categorical`

`nbins` — Number of bins
positive integer

Number of bins, specified as a positive integer. If you do not specify `nbins`, then `histogram` automatically calculates how many bins to use based on the values in `X`.

Example: `histogram(X,15)` creates a histogram with 15 bins.

`edges` — Bin edges
vector

Bin edges, specified as a vector. `edges(1)` is the left edge of the first bin, and `edges(end)` is the right edge of the last bin.

The value `X(i)` is in the `k`th bin if `edges(k)` ≤ `X(i)` < `edges(k+1)`. The last bin also includes the right bin edge, so that it contains `X(i)` if `edges(end-1)` ≤ `X(i)` ≤ `edges(end)`.

For datetime and duration data, `edges` must be a datetime or duration vector in monotonically increasing order.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `datetime` | `duration`

`Categories` — Categories included in histogram
cell array of character vectors | categorical array | string array

Note This option only applies to categorical histograms.

Categories included in histogram, specified as a cell array of character vectors, categorical array, or string array.

- If you specify an input categorical array `C`, then by default, `histogram` plots a bar for each category in `C`. In that case, use `Categories` to specify a unique subset of the categories instead.
- If you specify bin counts, then `Categories` specifies the associated category names for the histogram.

Example: `h = histogram(C,{'Large','Small'})` plots only the categorical data in the categories `'Large'` and `'Small'`.

Example: `histogram('Categories',{'Yes','No','Maybe'},'BinCounts',[22 18 3])` plots a histogram that has three categories with the associated bin counts.

Example: `h.Categories` queries the categories that are in histogram object `h`.

Data Types: `cell` | `categorical` | `string`

`counts` — Bin counts
vector

Bin counts, specified as a vector. Use this input to pass bin counts to `histogram` when the bin counts calculation is performed separately and you do not want `histogram` to do any data binning.

The length of `counts` must be equal to the number of bins.

- For numeric histograms, the number of bins is `length(edges)-1`.
- For categorical histograms, the number of bins is equal to the number of categories.

Example: `histogram('BinEdges',-2:2,'BinCounts',[5 8 15 9])`

Example: `histogram('Categories',{'Yes','No','Maybe'},'BinCounts',[22 18 3])`

ax — Target axes
`Axes` object | `PolarAxes` object

Target axes, specified as an `Axes` object or a `PolarAxes` object. If you do not specify the axes and if the current axes are Cartesian axes, then the `histogram` function uses the current axes (`gca`). To plot into polar axes, specify the `PolarAxes` object as the first input argument or use the `polarhistogram` function.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `histogram(X,'BinWidth',5)`

The histogram properties listed here are only a subset. For a complete list, see Histogram.

`BarWidth` — Relative width of categorical bars
`0.9` (default) | scalar in range `[0,1]`

Note This option only applies to histograms of categorical data.

Relative width of categorical bars, specified as a scalar value in the range `[0,1]`. Use this property to control the separation of categorical bars within the histogram. The default value is `0.9`, which means that the bar width is 90% of the space from the previous bar to the next bar, with 5% of that space on each side.

If you set this property to `1`, then adjacent bars touch.

Example: `0.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

BinLimits — Bin limits
two-element vector

Bin limits, specified as a two-element vector, `[bmin,bmax]`. This option plots a histogram using the values in the input array, `X`, that fall between `bmin` and `bmax` inclusive. That is, `X(X>=bmin & X<=bmax)`.

This option does not apply to histograms of categorical data.

Example: `histogram(X,'BinLimits',[1,10])` plots a histogram using only the values in `X` that are between `1` and `10` inclusive.

BinLimitsMode — Selection mode for bin limits
`'auto'` (default) | `'manual'`

Selection mode for bin limits, specified as `'auto'` or `'manual'`. The default value is `'auto'`, so that the bin limits automatically adjust to the data.

If you explicitly specify either `BinLimits` or `BinEdges`, then `BinLimitsMode` is automatically set to `'manual'`. In that case, specify `BinLimitsMode` as `'auto'` to rescale the bin limits to the data.

This option does not apply to histograms of categorical data.

BinMethod — Binning algorithm
`'auto'` (default) | `'scott'` | `'fd'` | `'integers'` | `'sturges'` | `'sqrt'` | ...

Binning algorithm, specified as one of the values in this table.

Value	Description
`'auto'`	The default `'auto'` algorithm chooses a bin width to cover the data range and reveal the shape of the underlying distribution.
`'scott'`	Scott's rule is optimal if the data is close to being normally distributed. This rule is appropriate for most other distributions, as well. It uses a bin width of `3.5*std(X(:))*numel(X)^(-1/3)`.

Value	Description
`'fd'`	The Freedman-Diaconis rule is less sensitive to outliers in the data, and might be more suitable for data with heavy-tailed distributions. It uses a bin width of `2*IQR(X(:))*numel(X)^(-1/3)`, where `IQR` is the interquartile range of `X`.
`'integers'`	The integer rule is useful with integer data, as it creates a bin for each integer. It uses a bin width of 1 and places bin edges halfway between integers. To avoid accidentally creating too many bins, you can use this rule to create a limit of 65536 bins (2^{16}). If the data range is greater than 65536, then the integer rule uses wider bins instead. **Note** `'integers'` does not support datetime or duration data.
`'sturges'`	Sturges' rule is popular due to its simplicity. It chooses the number of bins to be `ceil(1 + log2(numel(X)))`.
`'sqrt'`	The Square Root rule is widely used in other software packages. It chooses the number of bins to be `ceil(sqrt(numel(X)))`.

`histogram` does not always choose the number of bins using these exact formulas. Sometimes the number of bins is adjusted slightly so that the bin edges fall on "nice" numbers.

For datetime data, the bin method can be one of these units of time:

`'second'`	`'month'`
`'minute'`	`'quarter'`
`'hour'`	`'year'`
`'day'`	`'decade'`

'week'	'century'

For duration data, the bin method can be one of these units of time:

'second'	'day'
'minute'	'year'
'hour'	

If you specify BinMethod with datetime or duration data, then histogram can use a maximum of 65,536 bins (or 2^{16}). If the specified bin duration requires more bins, then histogram uses a larger bin width corresponding to the maximum number of bins.

This option does not apply to histograms of categorical data.

Note If you set the BinLimits, NumBins, BinEdges, or BinWidth property, then the BinMethod property is set to 'manual'.

Example: histogram(X,'BinMethod','integers') creates a histogram with the bins centered on integers.

BinWidth — Width of bins
scalar

Width of bins, specified as a scalar. When you specify BinWidth, then histogram can use a maximum of 65,536 bins (or 2^{16}). If instead the specified bin width requires more bins, then histogram uses a larger bin width corresponding to the maximum number of bins.

For datetime and duration data, the value of 'BinWidth' can be a scalar duration or calendar duration.

This option does not apply to histograms of categorical data.

Example: histogram(X,'BinWidth',5) uses bins with a width of 5.

DisplayOrder — Category display order
'data' (default) | 'ascend' | 'descend'

Category display order, specified as `'ascend'`, `'descend'`, or `'data'`. With `'ascend'` or `'descend'`, the histogram displays with increasing or decreasing bar heights. The default `'data'` value uses the category order in the input data, `C`.

This option only works with categorical data.

`DisplayStyle` — Histogram display style
`'bar'` (default) | `'stairs'`

Histogram display style, specified as either `'bar'` or `'stairs'`. Specify `'stairs'` to display a stairstep plot, which displays the outline of the histogram without filling the interior.

The default value of `'bar'` displays a histogram bar plot.

Example: `histogram(X,'DisplayStyle','stairs')` plots the outline of the histogram.

`EdgeAlpha` — Transparency of histogram bar edges
`1` (default) | scalar value between `0` and `1` inclusive

Transparency of histogram bar edges, specified as a scalar value between `0` and `1` inclusive. A value of `1` means fully opaque and `0` means completely transparent (invisible).

Example: `histogram(X,'EdgeAlpha',0.5)` creates a histogram plot with semi-transparent bar edges.

`EdgeColor` — Histogram edge color
`[0 0 0]` or black (default) | `'none'` | `'auto'` | RGB triplet | hexadecimal color code | color name

Histogram edge color, specified as one of these values:

- `'none'` — Edges are not drawn.
- `'auto'` — Color of each edge is chosen automatically.
- RGB triplet, hexadecimal color code, or color name — Edges use the specified color.

  RGB triplets and hexadecimal color codes are useful for specifying custom colors.

  - An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `histogram(X,'EdgeColor','r')` creates a histogram plot with red bar edges.

`FaceAlpha` — Transparency of histogram bars

`0.6` (default) | scalar value between `0` and `1` inclusive

Transparency of histogram bars, specified as a scalar value between `0` and `1` inclusive. `histogram` uses the same transparency for all the bars of the histogram. A value of `1` means fully opaque and `0` means completely transparent (invisible).

Example: `histogram(X,'FaceAlpha',1)` creates a histogram plot with fully opaque bars.

`FaceColor` — Histogram bar color

`'auto'` (default) | `'none'` | RGB triplet | hexadecimal color code | color name

Histogram bar color, specified as one of these values:

- `'none'` — Bars are not filled.
- `'auto'` — Histogram bar color is chosen automatically (default).
- RGB triplet, hexadecimal color code, or color name — Bars are filled with the specified color.

  RGB triplets and hexadecimal color codes are useful for specifying custom colors.

  - An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
  - A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

  Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

If you specify `DisplayStyle` as `'stairs'`, then `histogram` does not use the `FaceColor` property.

Example: `histogram(X,'FaceColor','g')` creates a histogram plot with green bars.

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Width of bar outlines

`0.5` (default) | positive value

Width of bar outlines, specified as a positive value in point units. One point equals 1/72 inch.

Example: `1.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Normalization` — Type of normalization

`'count'` (default) | `'probability'` | `'countdensity'` | `'pdf'` | `'cumcount'` | `'cdf'`

Type of normalization, specified as one of the values in this table. For each bin `i`:

- v_i is the bin value.
- c_i is the number of elements in the bin.
- w_i is the width of the bin.
- N is the number of elements in the input data. This value can be greater than the binned data if the data contains `NaN`, `NaT`, or `<undefined>` values, or if some of the data lies outside the bin limits.

Value	Bin Values	Notes
`'count'` (default)	$v_i = c_i$	• Count or frequency of observations. • Sum of bin values is less than or equal to `numel(X)`. The sum is less than `numel(X)` only when some of the input data is not included in the bins. • For categorical data, sum of bin values is less than or equal to either `numel(X)` or `sum(ismember(X(:),Categories))`.
`'countdensity'`	$v_i = \frac{c_i}{w_i}$	• Count or frequency scaled by width of bin. • The area (height * width) of each bar is the number of observations in the bin. The sum of the bar areas is less than or equal to `numel(X)`. • For categorical histograms, this is the same as `'count'`. **Note** `'countdensity'` does not support datetime or duration data.

Value	Bin Values	Notes
`'cumcount'`	$v_i = \sum_{j=1}^{i} c_j$	• Cumulative count. Each bin value is the cumulative number of observations in that bin and all previous bins. • The height of the last bar is less than or equal to `numel(X)`. • For categorical histograms, the height of the last bar is less than or equal to `numel(X)` or `sum(ismember(X(:),Categories))`.
`'probability'`	$v_i = \frac{c_i}{N}$	• Relative probability. • The sum of the bar heights is less than or equal to `1`.
`'pdf'`	$v_i = \frac{c_i}{N \cdot w_i}$	• Probability density function estimate. • The area of each bar is the relative number of observations. The sum of the bar areas is less than or equal to `1`. • For categorical histograms, this is the same as `'probability'`. **Note** `'pdf'` does not support datetime or duration data.

Value	Bin Values	Notes
`'cdf'`	$v_i = \sum_{j=1}^{i} \frac{c_j}{N}$	• Cumulative density function estimate. • The height of each bar is equal to the cumulative relative number of observations in the bin and all previous bins. The height of the last bar is less than or equal to 1. • For categorical data, the height of each bar is equal to the cumulative relative number of observations in each category and all previous categories.

Example: `histogram(X,'Normalization','pdf')` plots an estimate of the probability density function for X.

`NumDisplayBins` — Number of categories to display
scalar

Number of categories to display, specified as a scalar. You can change the ordering of categories displayed in the histogram using the `'DisplayOrder'` option.

This option only works with categorical data.

`Orientation` — Orientation of bars
`'vertical'` (default) | `'horizontal'`

Orientation of bars, specified as `'vertical'` or `'horizontal'`.

Example: `histogram(X,'Orientation','horizontal')` creates a histogram plot with horizontal bars.

`ShowOthers` — Toggle summary display of data belonging to undisplayed categories
`'off'` (default) | `'on'`

Toggle summary display of data belonging to undisplayed categories, specified as `'off'` or `'on'`. Set this option to `'on'` to display an additional bar in the histogram with the name `'Others'`. This extra bar counts all elements that do not belong to categories displayed in the histogram.

You can change the number of categories displayed in the histogram, as well as their order, using the `'NumDisplayBins'` and `'DisplayOrder'` options.

This option only works with categorical data.

Output Arguments

h — Histogram
object

Histogram, returned as an object. For more information, see Histogram.

Properties

Histogram	Histogram appearance and behavior

Object Functions

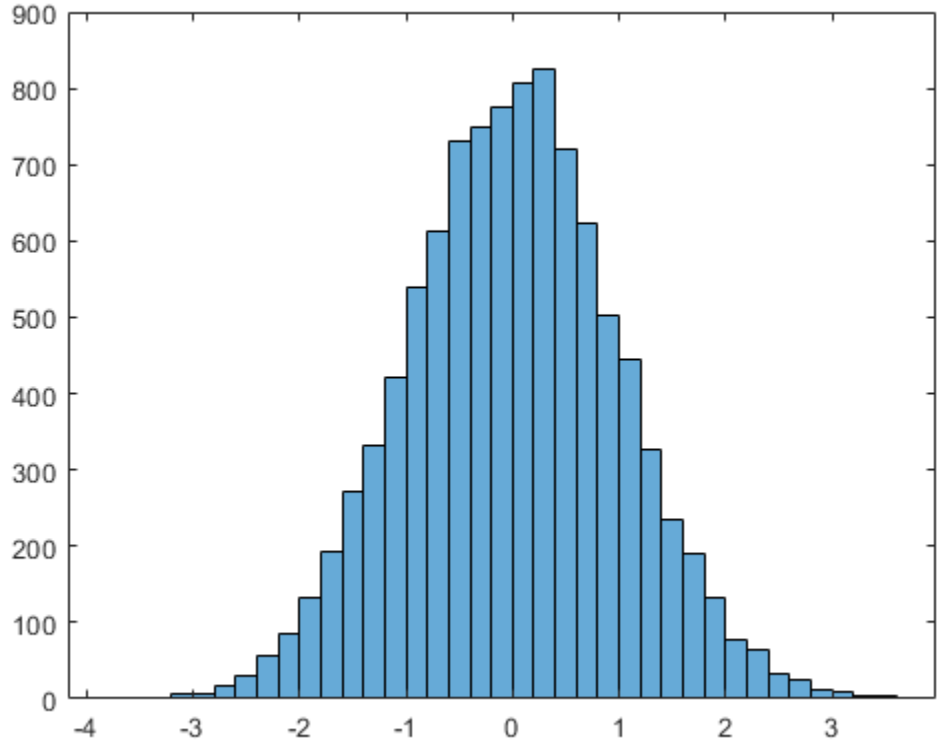
morebins	Increase number of histogram bins
fewerbins	Decrease number of histogram bins

Examples

Histogram of Vector

Generate 10,000 random numbers and create a histogram. The `histogram` function automatically chooses an appropriate number of bins to cover the range of values in `x` and show the shape of the underlying distribution.

```
x = randn(10000,1);
h = histogram(x)
```




```
h =
  Histogram with properties:

            Data: [10000x1 double]
          Values: [1x37 double]
         NumBins: 37
        BinEdges: [1x38 double]
        BinWidth: 0.2000
       BinLimits: [-3.8000 3.6000]
   Normalization: 'count'
       FaceColor: 'auto'
       EdgeColor: [0 0 0]
```

```
  Show all properties
```

When you specify an output argument to the `histogram` function, it returns a histogram object. You can use this object to inspect the properties of the histogram, such as the number of bins or the width of the bins.

Find the number of histogram bins.

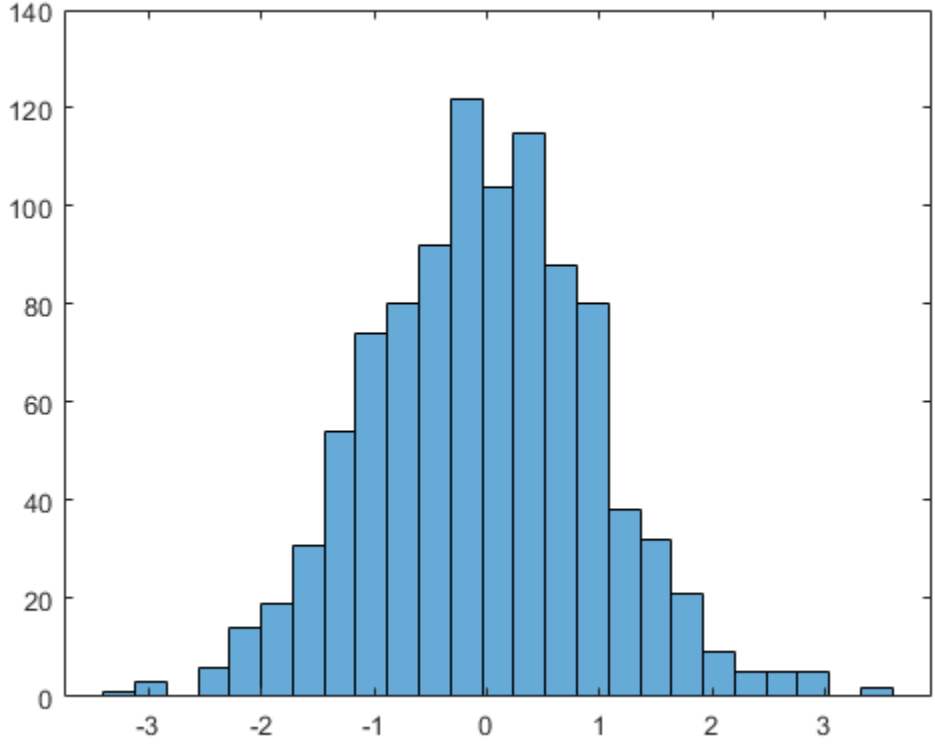
```
nbins = h.NumBins
```

```
nbins = 37
```

Specify Number of Histogram Bins

Plot a histogram of 1,000 random numbers sorted into 25 equally spaced bins.

```
x = randn(1000,1);
nbins = 25;
h = histogram(x,nbins)
```




```
h =
  Histogram with properties:

            Data: [1000x1 double]
          Values: [1x25 double]
         NumBins: 25
        BinEdges: [1x26 double]
        BinWidth: 0.2800
       BinLimits: [-3.4000 3.6000]
   Normalization: 'count'
       FaceColor: 'auto'
       EdgeColor: [0 0 0]
```

```
  Show all properties
```

Find the bin counts.

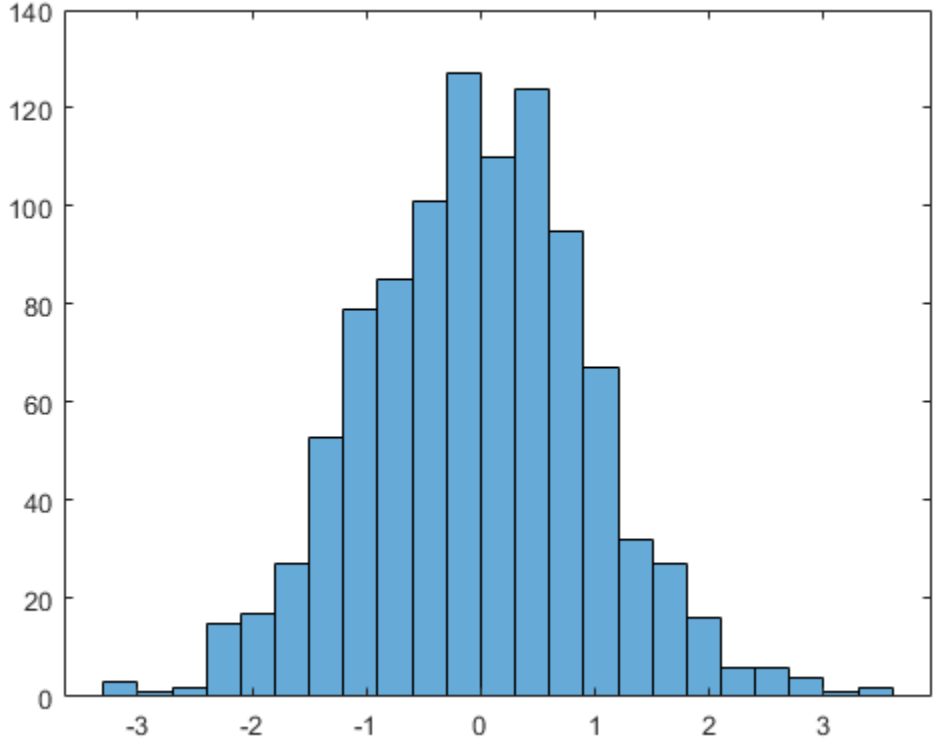
```
counts = h.Values
```

```
counts = 1×25

     1     3     0     6    14    19    31    54    74    80    92   122   104   115
```

Change Number of Histogram Bins

Generate 1,000 random numbers and create a histogram.

```
X = randn(1000,1);
h = histogram(X)
```

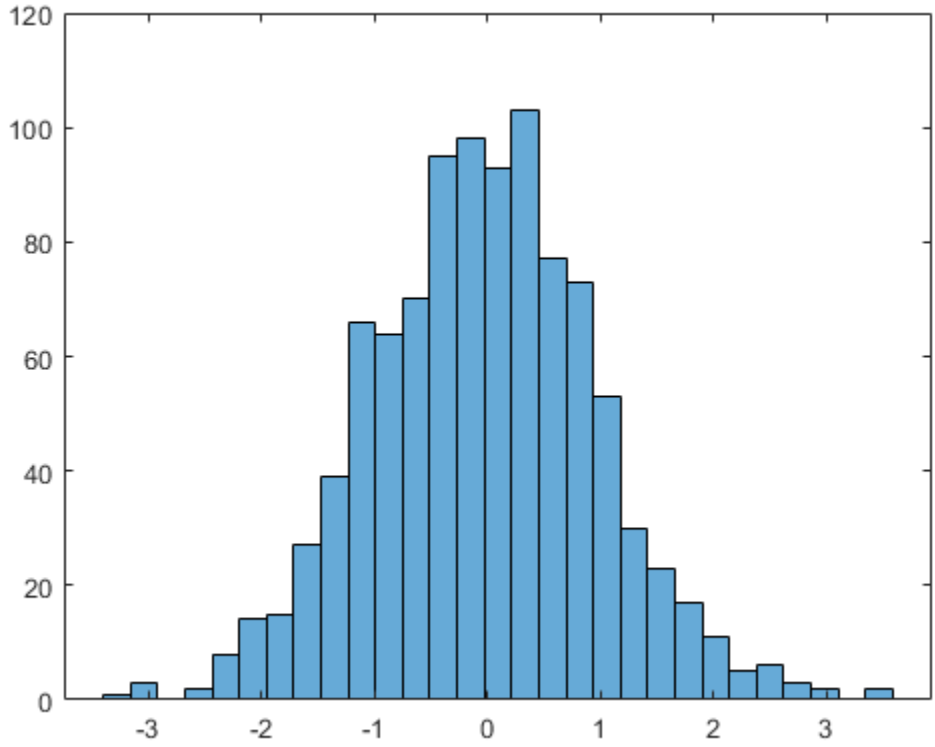



```
h =
  Histogram with properties:

            Data: [1000x1 double]
          Values: [1x23 double]
         NumBins: 23
        BinEdges: [1x24 double]
        BinWidth: 0.3000
       BinLimits: [-3.3000 3.6000]
   Normalization: 'count'
       FaceColor: 'auto'
       EdgeColor: [0 0 0]
```

```
Show all properties
```

Use the `morebins` function to coarsely adjust the number of bins.

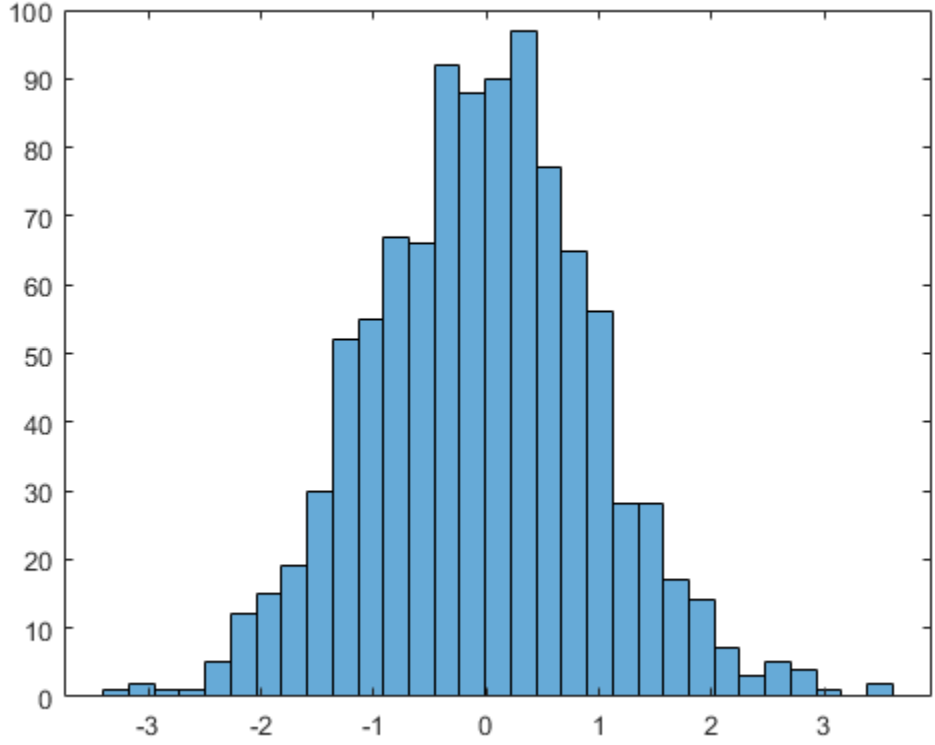
```
Nbins = morebins(h);
Nbins = morebins(h)
```




```
Nbins = 29
```

Adjust the bins at a fine grain level by explicitly setting the number of bins.

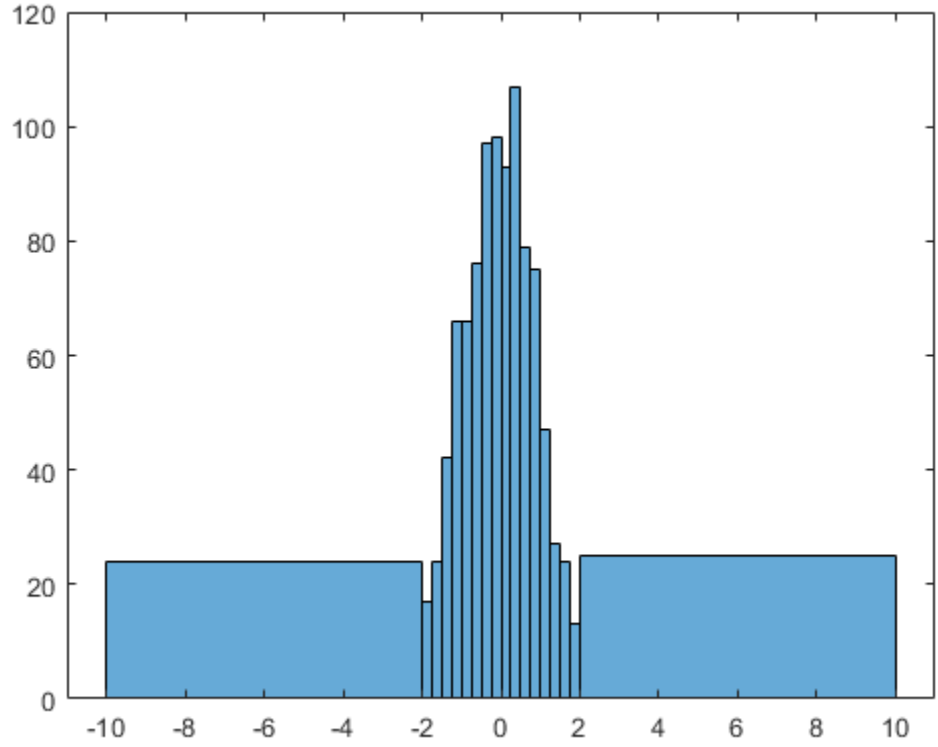
```
h.NumBins = 31;
```



Specify Bin Edges of Histogram

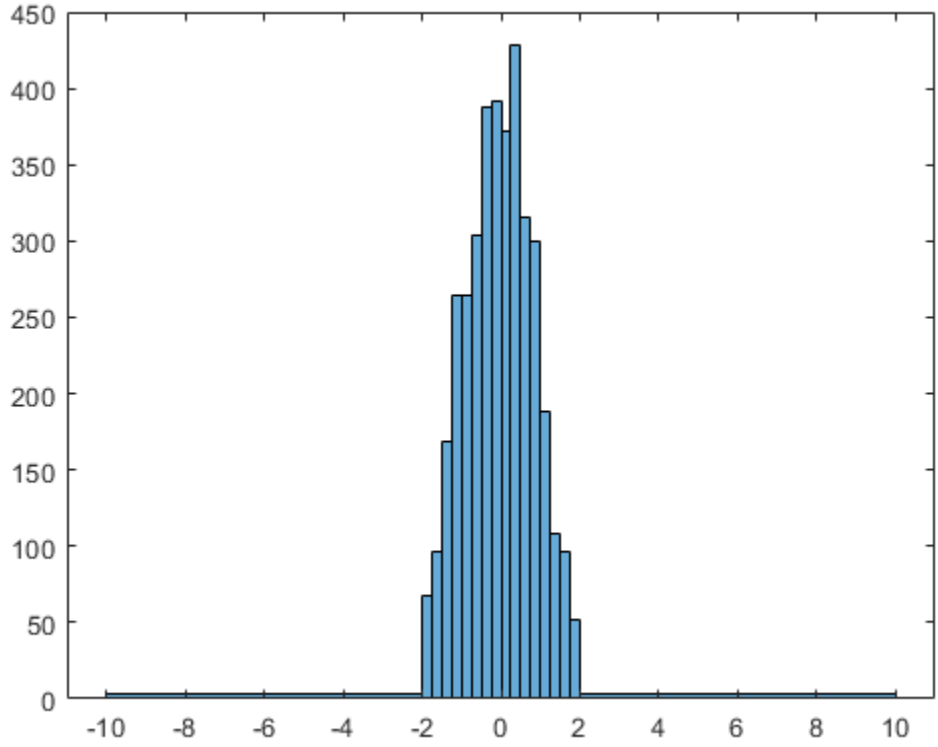
Generate 1,000 random numbers and create a histogram. Specify the bin edges as a vector with wide bins on the edges of the histogram to capture the outliers that do not satisfy $|x| < 2$. The first vector element is the left edge of the first bin, and the last vector element is the right edge of the last bin.

```
x = randn(1000,1);
edges = [-10 -2:0.25:2 10];
h = histogram(x,edges);
```



Specify the `Normalization` property as `'countdensity'` to flatten out the bins containing the outliers. Now, the *area* of each bin (rather than the height) represents the frequency of observations in that interval.

```
h.Normalization = 'countdensity';
```



Plot Categorical Histogram

Create a categorical vector that represents votes. The categories in the vector are `'yes'`, `'no'`, or `'undecided'`.

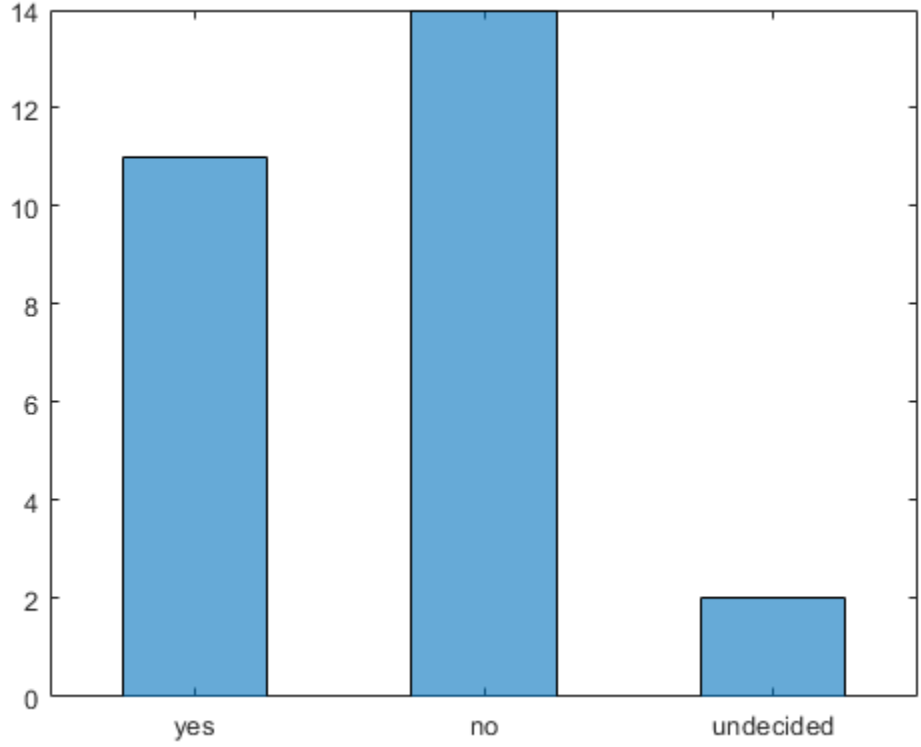
```
A = [0 0 1 1 1 0 0 0 0 NaN NaN 1 0 0 0 1 0 1 0 1 0 0 0 1 1 1 1];
C = categorical(A,[1 0 NaN],{'yes','no','undecided'})
```

```
C = 1x27 categorical array
  Columns 1 through 9
```

```
    no      no      yes      yes      yes      no      no      no      no
  Columns 10 through 16
    undecided      undecided      yes      no      no      no      yes
  Columns 17 through 25
    no      yes      no      yes      no      no      no      yes      yes
  Columns 26 through 27
    yes      yes
```

Plot a categorical histogram of the votes, using a relative bar width of `0.5`.

```
h = histogram(C,'BarWidth',0.5)
```



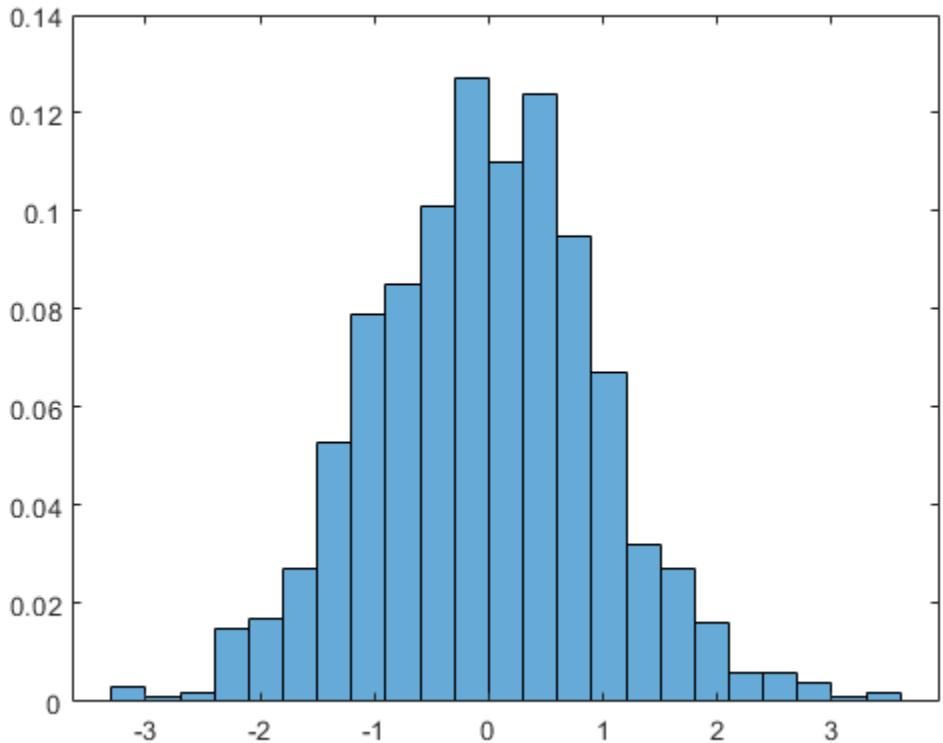

```
h =
  Histogram with properties:

              Data: [1x27 categorical]
            Values: [11 14 2]
    NumDisplayBins: 3
        Categories: {'yes'  'no'  'undecided'}
      DisplayOrder: 'data'
     Normalization: 'count'
      DisplayStyle: 'bar'
         FaceColor: 'auto'
         EdgeColor: [0 0 0]
```

Show all properties

Histogram with Specified Normalization

Generate 1,000 random numbers and create a histogram using the `'probability'` normalization.

```
x = randn(1000,1);
h = histogram(x,'Normalization','probability')
```



```
h =
  Histogram with properties:

             Data: [1000x1 double]
           Values: [1x23 double]
          NumBins: 23
         BinEdges: [1x24 double]
         BinWidth: 0.3000
        BinLimits: [-3.3000 3.6000]
    Normalization: 'probability'
        FaceColor: 'auto'
        EdgeColor: [0 0 0]

  Show all properties
```

Compute the sum of the bar heights. With this normalization, the height of each bar is equal to the probability of selecting an observation within that bin interval, and the height of all of the bars sums to 1.

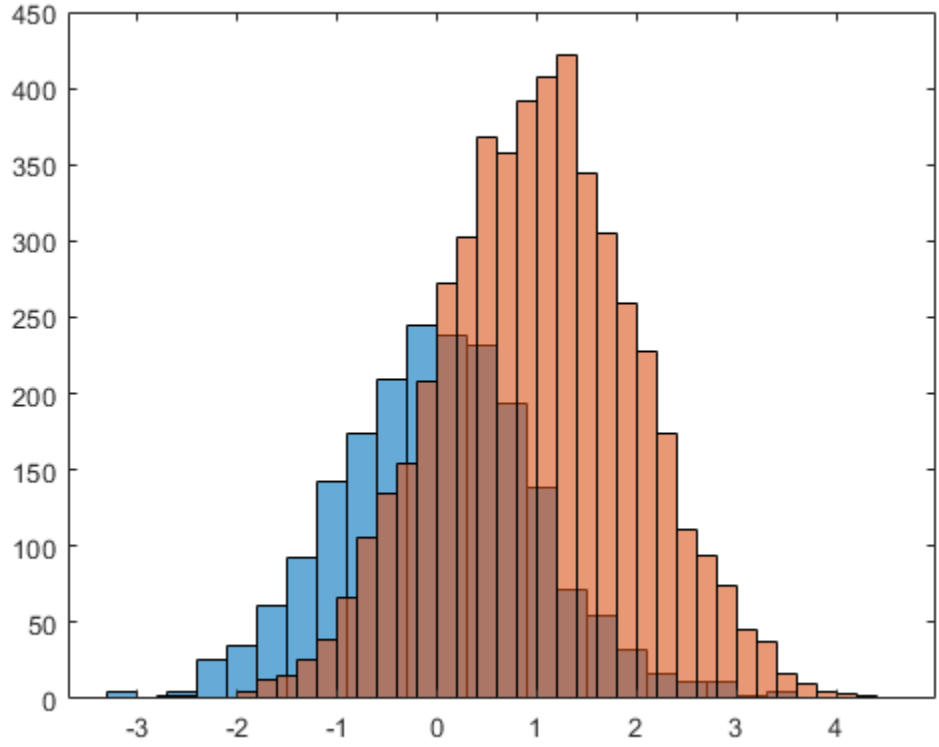
```
S = sum(h.Values)
```

```
S = 1
```

Plot Multiple Histograms

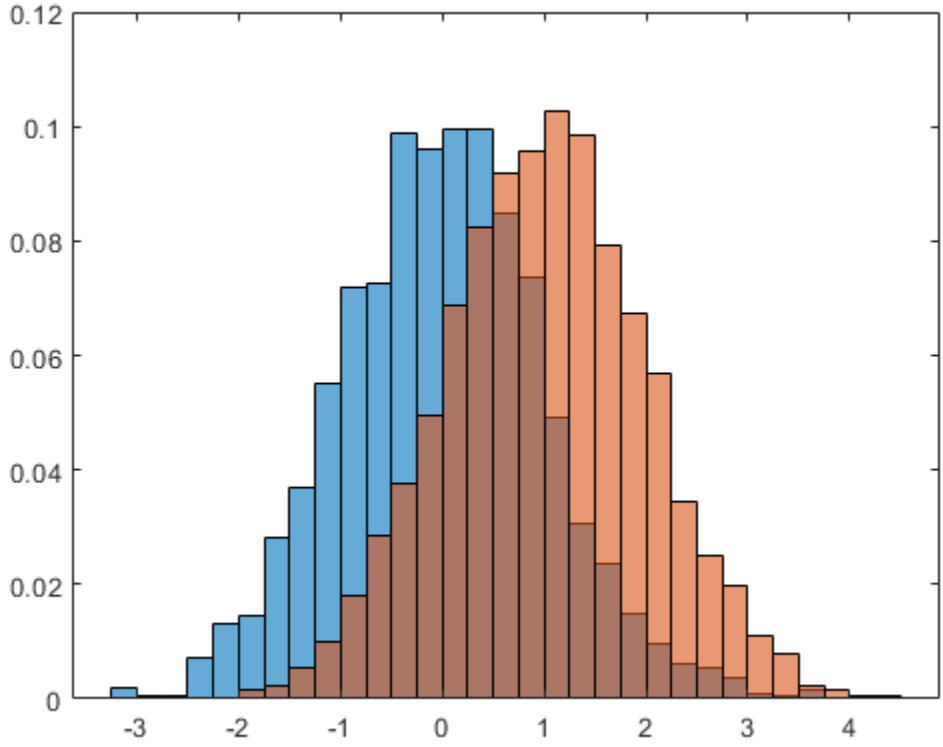
Generate two vectors of random numbers and plot a histogram for each vector in the same figure.

```
x = randn(2000,1);
y = 1 + randn(5000,1);
h1 = histogram(x);
hold on
h2 = histogram(y);
```



Since the sample size and bin width of the histograms are different, it is difficult to compare them. Normalize the histograms so that all of the bar heights add to 1, and use a uniform bin width.

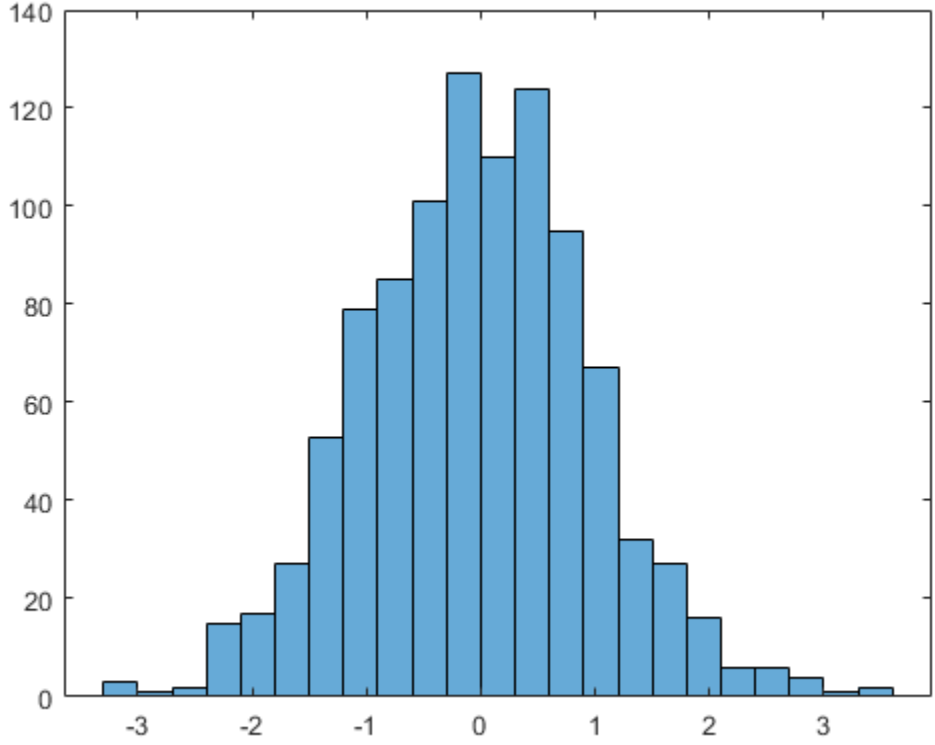
```
h1.Normalization = 'probability';
h1.BinWidth = 0.25;
h2.Normalization = 'probability';
h2.BinWidth = 0.25;
```



Adjust Histogram Properties

Generate 1,000 random numbers and create a histogram. Return the histogram object to adjust the properties of the histogram without recreating the entire plot.

```
x = randn(1000,1);
h = histogram(x)
```

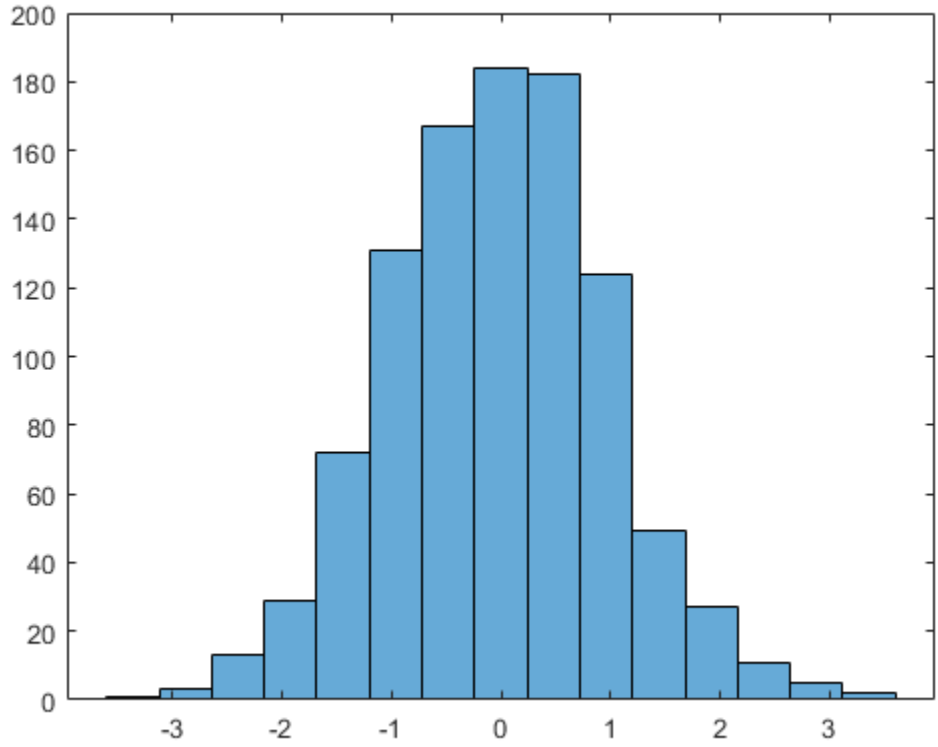



```
h =
  Histogram with properties:

            Data: [1000x1 double]
          Values: [1x23 double]
         NumBins: 23
        BinEdges: [1x24 double]
        BinWidth: 0.3000
       BinLimits: [-3.3000 3.6000]
   Normalization: 'count'
       FaceColor: 'auto'
       EdgeColor: [0 0 0]
```

```
Show all properties
```

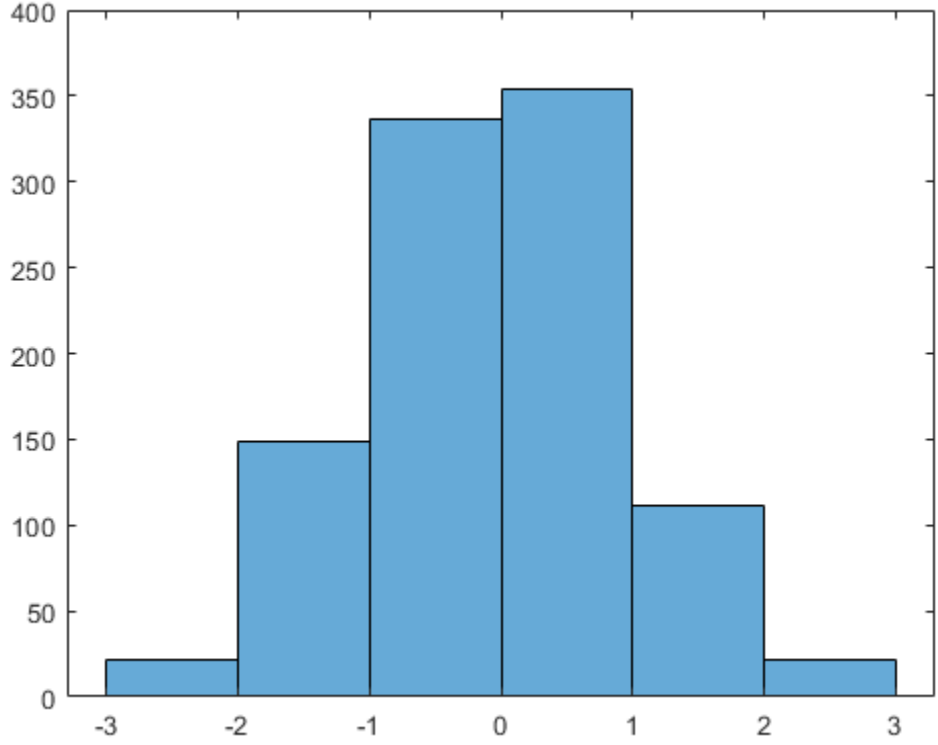
Specify exactly how many bins to use.

```
h.NumBins = 15;
```



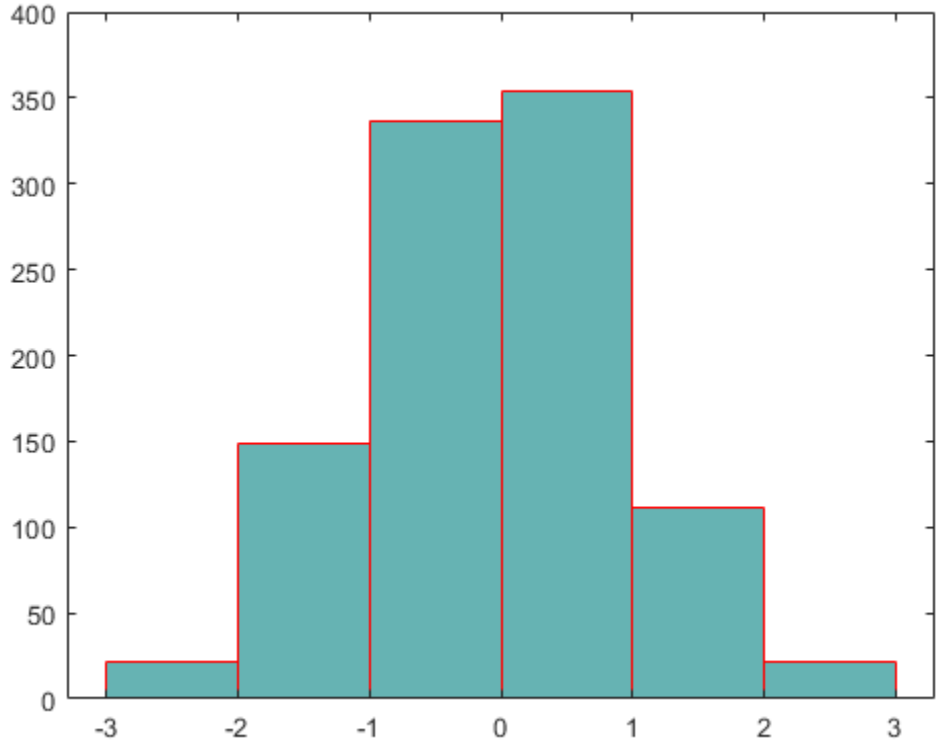
Specify the edges of the bins with a vector. The first value in the vector is the left edge of the first bin. The last value is the right edge of the last bin.

```
h.BinEdges = [-3:3];
```



Change the color of the histogram bars.

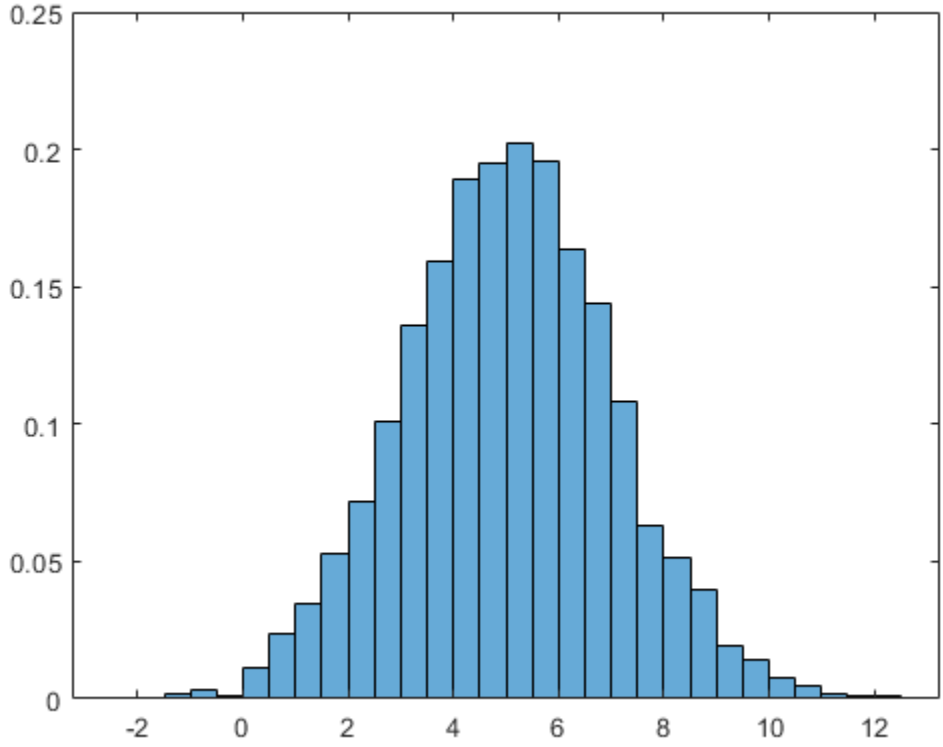
```
h.FaceColor = [0 0.5 0.5];
h.EdgeColor = 'r';
```



Determine Underlying Probability Distribution

Generate 5,000 normally distributed random numbers with a mean of 5 and a standard deviation of 2. Plot a histogram with `Normalization` set to `'pdf'` to produce an estimation of the probability density function.

```
x = 2*randn(5000,1) + 5;
histogram(x,'Normalization','pdf')
```



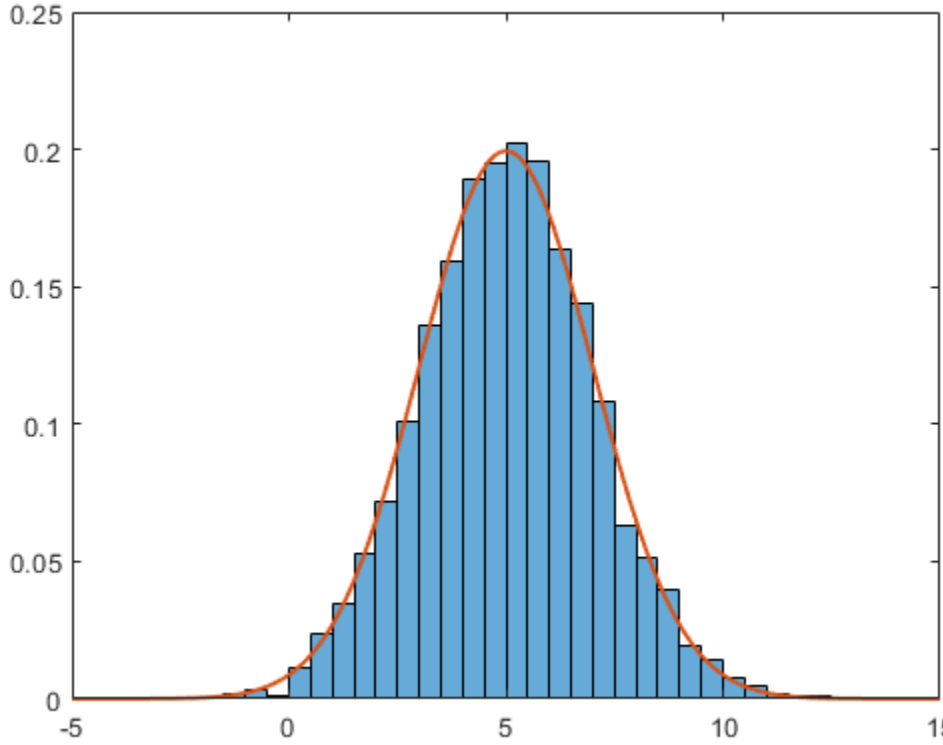
In this example, the underlying distribution for the normally distributed data is known. You can, however, use the `'pdf'` histogram plot to determine the underlying probability distribution of the data by comparing it against a known probability density function.

The probability density function for a normal distribution with mean μ, standard deviation σ, and variance σ^2 is

$$f(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2} \right].$$

Overlay a plot of the probability density function for a normal distribution with a mean of 5 and a standard deviation of 2.

```
hold on
y = -5:0.1:15;
mu = 5;
sigma = 2;
f = exp(-(y-mu).^2./(2*sigma^2))./(sigma*sqrt(2*pi));
plot(y,f,'LineWidth',1.5)
```



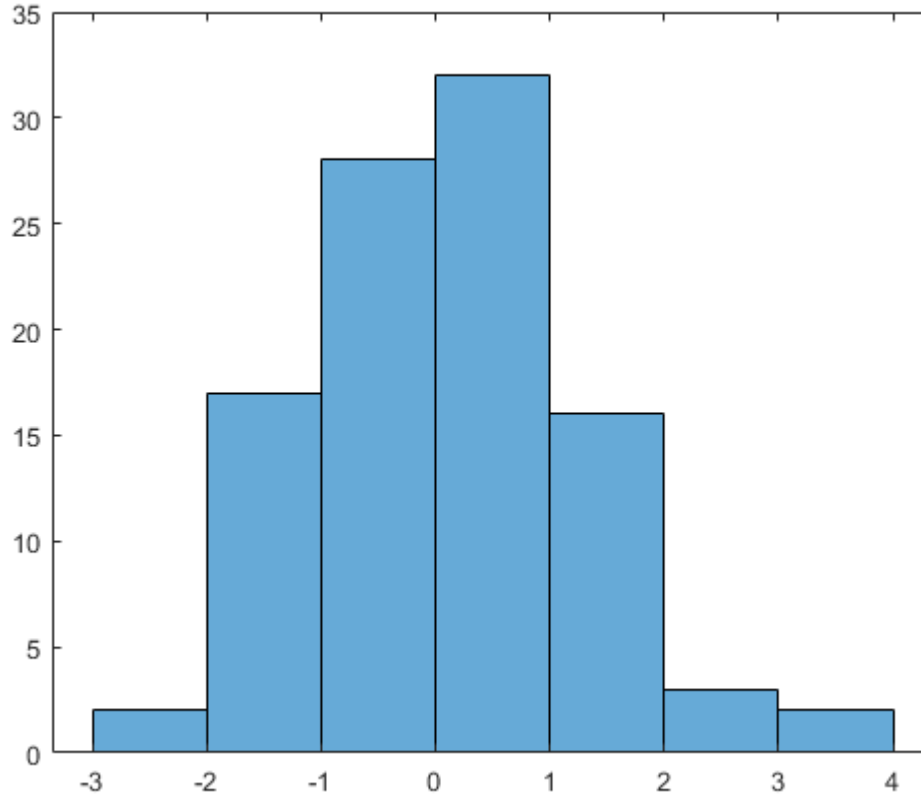
Saving and Loading Histogram Objects

Use the `savefig` function to save a histogram figure.

```
y = histogram(randn(10));
savefig('histogram.fig');
clear all
close all
```

Use `openfig` to load the histogram figure back into MATLAB. `openfig` also returns a handle to the figure, `h`.

```
h = openfig('histogram.fig');
```



Use the `findobj` function to locate the correct object handle from the figure handle. This allows you to continue manipulating the original histogram object used to generate the figure.

```
y = findobj(h, 'type', 'histogram')
```

```
y = 
  Histogram with properties:

             Data: [10x10 double]
           Values: [2 17 28 32 16 3 2]
          NumBins: 7
         BinEdges: [-3 -2 -1 0 1 2 3 4]
         BinWidth: 1
        BinLimits: [-3 4]
    Normalization: 'count'
        FaceColor: 'auto'
        EdgeColor: [0 0 0]

  Show all properties
```

Tips

- Histogram plots created using `histogram` have a context menu in plot edit mode that enables interactive manipulations in the figure window. For example, you can use the context menu to interactively change the number of bins, align multiple histograms, or change the display order.
- When you add data tips to a histogram plot, they display the bin edges and bin count.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

- Some input options are not supported. The allowed options are:
  - `'BinWidth'`
  - `'BinLimits'`
  - `'Normalization'`
  - `'DisplayStyle'`
  - `'BinMethod'` — The `'auto'` and `'scott'` bin methods are the same. The `'fd'` bin method is not supported.
  - `'EdgeAlpha'`
  - `'EdgeColor'`
  - `'FaceAlpha'`
  - `'FaceColor'`
  - `'LineStyle'`
  - `'LineWidth'`
  - `'Orientation'`
- Additionally, there is a cap on the maximum number of bars. The default maximum is 100.
- The `morebins` and `fewerbins` methods are not supported.
- Editing properties of the histogram object that require recomputing the bins is not supported.

For more information, see “Tall Arrays for Out-of-Memory Data”.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Histogram | `discretize` | `fewerbins` | `histcounts` | `histcounts2` | `histogram2` | `morebins`

Topics

“Plot Categorical Data”
“Control Categorical Histogram Display”
“Replace Discouraged Instances of hist and histc”

Introduced in R2014b

Histogram Properties

Histogram appearance and behavior

Description

Histogram properties control the appearance and behavior of the histogram. By changing property values, you can modify aspects of the histogram. Use dot notation to refer to a particular object and property:

```
h = histogram(randn(10,1));
c = h.BinWidth;
h.BinWidth = 2;
```

Properties

Bins

`NumBins` — Number of bins
positive integer

Number of bins, specified as a positive integer. If you do not specify `NumBins`, then `histogram` automatically calculates how many bins to use based on the values in `Data`.

This option does not apply to histograms of categorical data.

`BinWidth` — Width of bins
scalar

Width of bins, specified as a scalar. When you specify `BinWidth`, then `histogram` can use a maximum of 65,536 bins (or 2^{16}). If instead the specified bin width requires more bins, then `histogram` uses a larger bin width corresponding to the maximum number of bins.

For datetime and duration data, the value of `'BinWidth'` can be a scalar duration or calendar duration.

This option does not apply to histograms of categorical data.

Example: `histogram(X,'BinWidth',5)` uses bins with a width of 5.

`BinEdges` — Edges of bins
numeric vector

Edges of bins, specified as a numeric vector. The first vector element specifies the left edge of the first bin. The last element specifies the right edge of the last bin. If you do not specify the bin edges, then `histogram` automatically determines the location of the bin edges.

This option does not apply to histograms of categorical data.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`BinLimits` — Bin limits
two-element vector

Bin limits, specified as a two-element vector, `[bmin,bmax]`. This option plots a histogram using the values in the input array, `X`, that fall between `bmin` and `bmax` inclusive. That is, `X(X>=bmin & X<=bmax)`.

This option does not apply to histograms of categorical data.

Example: `histogram(X,'BinLimits',[1,10])` plots a histogram using only the values in `X` that are between `1` and `10` inclusive.

`BinLimitsMode` — Selection mode for bin limits
`'auto'` (default) | `'manual'`

Selection mode for bin limits, specified as `'auto'` or `'manual'`. The default value is `'auto'`, so that the bin limits automatically adjust to the data.

If you explicitly specify either `BinLimits` or `BinEdges`, then `BinLimitsMode` is automatically set to `'manual'`. In that case, specify `BinLimitsMode` as `'auto'` to rescale the bin limits to the data.

This option does not apply to histograms of categorical data.

`BinMethod` — Binning algorithm
`'auto'` (default) | `'scott'` | `'fd'` | `'integers'` | `'sturges'` | `'sqrt'` | ...

Binning algorithm, specified as one of the values in this table.

Value	Description
`'auto'`	The default `'auto'` algorithm chooses a bin width to cover the data range and reveal the shape of the underlying distribution.
`'scott'`	Scott's rule is optimal if the data is close to being normally distributed. This rule is appropriate for most other distributions, as well. It uses a bin width of `3.5*std(X(:))*numel(X)^(-1/3)`.
`'fd'`	The Freedman-Diaconis rule is less sensitive to outliers in the data, and might be more suitable for data with heavy-tailed distributions. It uses a bin width of `2*IQR(X(:))*numel(X)^(-1/3)`, where `IQR` is the interquartile range of `X`.
`'integers'`	The integer rule is useful with integer data, as it creates a bin for each integer. It uses a bin width of 1 and places bin edges halfway between integers. To avoid accidentally creating too many bins, you can use this rule to create a limit of 65536 bins (2^{16}). If the data range is greater than 65536, then the integer rule uses wider bins instead. **Note** `'integers'` does not support datetime or duration data.
`'sturges'`	Sturges' rule is popular due to its simplicity. It chooses the number of bins to be `ceil(1 + log2(numel(X)))`.
`'sqrt'`	The Square Root rule is widely used in other software packages. It chooses the number of bins to be `ceil(sqrt(numel(X)))`.

`histogram` does not always choose the number of bins using these exact formulas. Sometimes the number of bins is adjusted slightly so that the bin edges fall on "nice" numbers.

For datetime data, the bin method can be one of these units of time:

`'second'`	`'month'`
`'minute'`	`'quarter'`
`'hour'`	`'year'`
`'day'`	`'decade'`
`'week'`	`'century'`

For duration data, the bin method can be one of these units of time:

`'second'`	`'day'`
`'minute'`	`'year'`
`'hour'`	

If you specify `BinMethod` with datetime or duration data, then `histogram` can use a maximum of 65,536 bins (or 2^{16}). If the specified bin duration requires more bins, then `histogram` uses a larger bin width corresponding to the maximum number of bins.

This option does not apply to histograms of categorical data.

Note If you set the `BinLimits`, `NumBins`, `BinEdges`, or `BinWidth` property, then the `BinMethod` property is set to `'manual'`.

Example: `histogram(X,'BinMethod','integers')` creates a histogram with the bins centered on integers.

Categories

`Categories` — Categories included in histogram
cell array of character vectors | categorical array | string array

Note This option only applies to categorical histograms.

Categories included in histogram, specified as a cell array of character vectors, categorical array, or string array.

- If you specify an input categorical array `C`, then by default, `histogram` plots a bar for each category in `C`. In that case, use `Categories` to specify a unique subset of the categories instead.
- If you specify bin counts, then `Categories` specifies the associated category names for the histogram.

Example: `h = histogram(C,{'Large','Small'})` plots only the categorical data in the categories `'Large'` and `'Small'`.

Example: `histogram('Categories',{'Yes','No','Maybe'},'BinCounts',[22 18 3])` plots a histogram that has three categories with the associated bin counts.

Example: `h.Categories` queries the categories that are in histogram object `h`.

Data Types: `cell` | `categorical` | `string`

`DisplayOrder` — Category display order
`'data'` (default) | `'ascend'` | `'descend'`

Category display order, specified as `'ascend'`, `'descend'`, or `'data'`. With `'ascend'` or `'descend'`, the histogram displays with increasing or decreasing bar heights. The default `'data'` value uses the category order in the input data, `C`.

This option only works with categorical data.

`NumDisplayBins` — Number of categories to display
scalar

Number of categories to display, specified as a scalar. You can change the ordering of categories displayed in the histogram using the `'DisplayOrder'` option.

This option only works with categorical data.

`ShowOthers` — Toggle summary display of data belonging to undisplayed categories
`'off'` (default) | `'on'`

Toggle summary display of data belonging to undisplayed categories, specified as `'off'` or `'on'`. Set this option to `'on'` to display an additional bar in the histogram with the name `'Others'`. This extra bar counts all elements that do not belong to categories displayed in the histogram.

You can change the number of categories displayed in the histogram, as well as their order, using the `'NumDisplayBins'` and `'DisplayOrder'` options.

This option only works with categorical data.

Data

`Data` — Data to distribute among bins

vector | matrix | multidimensional array | categorical array

Data to distribute among bins, specified as a vector, matrix, multidimensional array, or categorical array. If `Data` is not a vector, then `histogram` treats it as a single column vector, `Data(:)`, and plots a single histogram.

`histogram` ignores all `NaN`, `NaT`, and undefined categorical values. Similarly, `histogram` ignores `Inf` and `-Inf` values unless the bin edges explicitly specify `Inf` or `-Inf` as a bin edge. Although `NaN`, `NaT`, `Inf`, `-Inf`, and `<undefined>` values are typically not plotted, they are still included in normalization calculations that include the total number of data elements, such as `'probability'`.

If you change the values in the `Data` property of a `histogram` object, then the bin edges are not automatically updated. To recompute the bins, adjust a bin-related property such as `BinMethod` or `NumBins`. You can only specify categorical values for `Data` if the histogram object was originally created using categoricals.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `categorical` | `datetime` | `duration`

`Values` — Bin values

numeric vector

This property is read-only.

Bin values, returned as a numeric vector. If `Normalization` is `'count'` (the default), then the `k`th element in `Values` specifies how many elements of `Data` fall in the kth bin interval (bin counts). The last bin includes values that are on *either* bin edge, but all other bins only include values that fall on the left edge.

Depending on the value of `Normalization`, the `Values` property can instead contain a normalized variant of the bin counts.

`Normalization` — Type of normalization

`'count'` (default) | `'probability'` | `'countdensity'` | `'pdf'` | `'cumcount'` | `'cdf'`

Type of normalization, specified as one of the values in this table. For each bin `i`:

- v_i is the bin value.
- c_i is the number of elements in the bin.
- w_i is the width of the bin.
- N is the number of elements in the input data. This value can be greater than the binned data if the data contains `NaN`, `NaT`, or `<undefined>` values, or if some of the data lies outside the bin limits.

Value	Bin Values	Notes
`'count'` (default)	$v_i = c_i$	• Count or frequency of observations. • Sum of bin values is less than or equal to `numel(X)`. The sum is less than `numel(X)` only when some of the input data is not included in the bins. • For categorical data, sum of bin values is less than or equal to either `numel(X)` or `sum(ismember(X(:),Categories))`.

Value	Bin Values	Notes
`'countdensity'`	$v_i = \frac{c_i}{w_i}$	• Count or frequency scaled by width of bin. • The area (height * width) of each bar is the number of observations in the bin. The sum of the bar areas is less than or equal to `numel(X)`. • For categorical histograms, this is the same as `'count'`. **Note** `'countdensity'` does not support datetime or duration data.
`'cumcount'`	$v_i = \sum_{j=1}^{i} c_j$	• Cumulative count. Each bin value is the cumulative number of observations in that bin and all previous bins. • The height of the last bar is less than or equal to `numel(X)`. • For categorical histograms, the height of the last bar is less than or equal to `numel(X)` or `sum(ismember(X(:),Categories))`.
`'probability'`	$v_i = \frac{c_i}{N}$	• Relative probability. • The sum of the bar heights is less than or equal to `1`.

Value	Bin Values	Notes
`'pdf'`	$v_i = \frac{c_i}{N \cdot w_i}$	• Probability density function estimate. • The area of each bar is the relative number of observations. The sum of the bar areas is less than or equal to 1. • For categorical histograms, this is the same as `'probability'`. **Note** `'pdf'` does not support datetime or duration data.
`'cdf'`	$v_i = \sum_{j=1}^{i} \frac{c_j}{N}$	• Cumulative density function estimate. • The height of each bar is equal to the cumulative relative number of observations in the bin and all previous bins. The height of the last bar is less than or equal to 1. • For categorical data, the height of each bar is equal to the cumulative relative number of observations in each category and all previous categories.

Example: `histogram(X,'Normalization','pdf')` plots an estimate of the probability density function for X.

BinCounts — Bin counts

vector

Bin counts, specified as a vector. Use this input to pass bin counts to `histogram` when the bin counts calculation is performed separately and you do not want `histogram` to do any data binning.

The length of `counts` must be equal to the number of bins.

- For numeric histograms, the number of bins is `length(edges)-1`.
- For categorical histograms, the number of bins is equal to the number of categories.

Compared to the `Values` property, `BinCounts` is not normalized. If `Normalization` is `'count'`, then `Values` and `BinCounts` are equivalent.

Example: `histogram('BinEdges',-2:2,'BinCounts',[5 8 15 9])`

Example: `histogram('Categories',{'Yes','No','Maybe'},'BinCounts',[22 18 3])`

BinCountsMode — Selection mode for bin counts

`'auto'` (default) | `'manual'`

Selection mode for bin counts, specified as `'auto'` or `'manual'`. The default value is `'auto'`, so that the bin counts are automatically computed from `Data` and `BinEdges`.

If you specify `BinCounts`, then `BinCountsMode` is automatically set to `'manual'`. Similarly, if you specify `Data`, then `BinCountsMode` is automatically set to `'auto'`.

Color and Styling

DisplayStyle — Histogram display style

`'bar'` (default) | `'stairs'`

Histogram display style, specified as either `'bar'` or `'stairs'`. Specify `'stairs'` to display a stairstep plot, which displays the outline of the histogram without filling the interior.

The default value of `'bar'` displays a histogram bar plot.

Example: `histogram(X,'DisplayStyle','stairs')` plots the outline of the histogram.

`Orientation` — Orientation of bars

`'vertical'` (default) | `'horizontal'`

Orientation of bars, specified as `'vertical'` or `'horizontal'`.

Example: `histogram(X,'Orientation','horizontal')` creates a histogram plot with horizontal bars.

`BarWidth` — Relative width of categorical bars

`0.9` (default) | scalar in range `[0,1]`

Note This option only applies to histograms of categorical data.

Relative width of categorical bars, specified as a scalar value in the range `[0,1]`. Use this property to control the separation of categorical bars within the histogram. The default value is `0.9`, which means that the bar width is 90% of the space from the previous bar to the next bar, with 5% of that space on each side.

If you set this property to `1`, then adjacent bars touch.

Example: `0.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`FaceColor` — Histogram bar color

`'auto'` (default) | `'none'` | RGB triplet | hexadecimal color code | color name

Histogram bar color, specified as one of these values:

- `'none'` — Bars are not filled.
- `'auto'` — The histogram bar color is chosen automatically (default).
- RGB triplet, hexadecimal color code, or color name — Bars are filled with the specified color.

  RGB triplets and hexadecimal color codes are useful for specifying custom colors.

  - An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

If you specify `DisplayStyle` as `'stairs'`, then `histogram` does not utilize the `FaceColor` property.

Example: `histogram(X,'FaceColor','g')` creates a histogram plot with green bars.

`EdgeColor` — Histogram edge color

`[0 0 0]` or black (default) | `'none'` | `'auto'` | RGB triplet | hexadecimal color code | color name

Histogram edge color, specified as one of these values:

- `'none'` — Edges are not drawn.
- `'auto'` — The color of each edge is chosen automatically.
- RGB triplet, hexadecimal color code, or color name — Edges use the specified color.

  RGB triplets and hexadecimal color codes are useful for specifying custom colors.

  - An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
  - A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

  Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `histogram(X,'EdgeColor','r')` creates a histogram plot with red bar edges.

`FaceAlpha` — Transparency of histogram bars

`0.6` (default) | scalar value between `0` and `1` inclusive

Transparency of histogram bars, specified as a scalar value between `0` and `1` inclusive. `histogram` uses the same transparency for all the bars of the histogram. A value of `1` means fully opaque and `0` means completely transparent (invisible).

Example: `histogram(X,'FaceAlpha',1)` creates a histogram plot with fully opaque bars.

`EdgeAlpha` — Transparency of histogram bar edges
`1` (default) | scalar value between `0` and `1` inclusive

Transparency of histogram bar edges, specified as a scalar value between `0` and `1` inclusive. A value of `1` means fully opaque and `0` means completely transparent (invisible).

Example: `histogram(X,'EdgeAlpha',0.5)` creates a histogram plot with semi-transparent bar edges.

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Width of bar outlines
`0.5` (default) | positive value

Width of bar outlines, specified as a positive value in point units. One point equals 1/72 inch.

Example: `1.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Legend

`DisplayName` — Text used by legend
variable name of `Data` or `''` (default) | character vector

Text used by the legend, specified as a character vector. The text appears next to an icon of the histogram.

Example: `'Text Description'`

For multiline text, create the character vector using `sprintf` with the new line character `\n`.

Example: `sprintf('line one\nline two')`

Alternatively, you can specify the legend text using the `legend` function.

- If you specify the text as an input argument to the `legend` function, then the legend uses the specified text and sets the `DisplayName` property to the same value.
- If you do not specify the text as an input argument to the `legend` function, then the legend uses the text in the `DisplayName` property. The default value of `DisplayName` is one of these values.
  - For numeric inputs, `DisplayName` is a character vector representing the variable name of the input data used to construct the histogram. If the input data does not have a variable name, then `DisplayName` is empty, `''`.
  - For categorical array inputs, `DisplayName` is empty, `''`.

If the `DisplayName` property does not contain any text, then the legend generates a character vector. The character vector has the form `'dataN'`, where `N` is the number assigned to the histogram object based on its location in the list of legend entries.

If you edit interactively the character vector in an existing legend, then MATLAB updates the `DisplayName` property to the edited character vector.

`Annotation` — Control for including or excluding object from legend
`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`DataTipTemplate` — Data tip content

`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.

- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.

- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback.

MATLAB resumes executing the running callback when the interrupting callback completes.

- If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.

- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks only when visible. The `Visible` property must be set to `'on'`. The `HitTest` property determines if the `Histogram` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `Histogram` object passes the click to the object behind it in the current view of the figure window. The `HitTest` property of the `Histogram` object has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `Histogram` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `Histogram` object that has one of these:
  - `HitTest` property set to `'on'`
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

---

Note The `PickableParts` property determines if the `Histogram` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent

`Axes` object | `PolarAxes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `PolarAxes`, `Group`, or `Transform` object.

`Children` — Children

empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the chart object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object
`'histogram'` | `'categoricalhistogram'`

This property is read-only.

Type of graphics object, returned as either `'histogram'` or `'categoricalhistogram'`. Use this property to find all objects of a given type within a plotting hierarchy, such as searching for the type using `findobj`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`histogram`

Topics

“Access Property Values”

Introduced in R2014b

histogram2

Bivariate histogram plot

Description

Bivariate histograms are a type of bar plot for numeric data that group the data into 2-D bins. After you create a `Histogram2` object, you can modify aspects of the histogram by changing its property values. This is particularly useful for quickly modifying the properties of the bins or changing the display.

Creation

Syntax

```
histogram2(X,Y)
histogram2(X,Y,nbins)
histogram2(X,Y,Xedges,Yedges)
histogram2('XBinEdges',Xedges,'YBinEdges',Yedges,'BinCounts',counts)

histogram2( ___ ,Name,Value)
histogram2(ax, ___ )
h = histogram2( ___ )
```

Description

`histogram2(X,Y)` creates a bivariate histogram plot of `X` and `Y`. The `histogram2` function uses an automatic binning algorithm that returns bins with a uniform area, chosen to cover the range of elements in `X` and `Y` and reveal the underlying shape of the distribution. `histogram2` displays the bins as 3-D rectangular bars such that the height of each bar indicates the number of elements in the bin.

`histogram2(X,Y,nbins)` specifies the number of bins to use in each dimension of the histogram.

`histogram2(X,Y,Xedges,Yedges)` specifies the edges of the bins in each dimension using the vectors `Xedges` and `Yedges`.

`histogram2('XBinEdges',Xedges,'YBinEdges',Yedges,'BinCounts',counts)` manually specifies the bin counts. `histogram2` plots the specified bin counts and does not do any data binning.

`histogram2( ___ ,Name,Value)` specifies additional options with one or more `Name,Value` pair arguments using any of the previous syntaxes. For example, you can specify `'BinWidth'` and a two-element vector to adjust the width of the bins in each dimension, or `'Normalization'` with a valid option (`'count'`, `'probability'`, `'countdensity'`, `'pdf'`, `'cumcount'`, or `'cdf'`) to use a different type of normalization. For a list of properties, see Histogram2.

`histogram2(ax, ___ )` plots into the axes specified by `ax` instead of into the current axes (`gca`). The option `ax` can precede any of the input argument combinations in the previous syntaxes.

`h = histogram2( ___ )` returns a `Histogram2` object. Use this to inspect and adjust properties of the bivariate histogram. For a list of properties, see Histogram2.

Input Arguments

X,Y — Data to distribute among bins (as separate arguments)

vectors | matrices | multidimensional arrays

Data to distribute among bins, specified as separate arguments of vectors, matrices, or multidimensional arrays. `X` and `Y` must be the same size. If `X` and `Y` are not vectors, then `histogram2` treats them as single column vectors, `X(:)` and `Y(:)`, and plots a single histogram.

Corresponding elements in `X` and `Y` specify the *x* and *y* coordinates of 2-D data points, `[X(k),Y(k)]`. The data types of `X` and `Y` can be different, but `histogram2` concatenates these inputs into a single `N`-by-2 matrix of the dominant data type.

`histogram2` ignores all `NaN` values. Similarly, `histogram2` ignores `Inf` and `-Inf` values, unless the bin edges explicitly specify `Inf` or `-Inf` as a bin edge. Although `NaN`, `Inf`, and `-Inf` values are typically not plotted, they are still included in normalization calculations that include the total number of data elements, such as `'probability'`.

Note If X or Y contain integers of type `int64` or `uint64` that are larger than `flintmax`, then it is recommended that you explicitly specify the histogram bin edges.`histogram2` automatically bins the input data using double precision, which lacks integer precision for numbers greater than `flintmax`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`nbins` — Number of bins in each dimension
scalar | vector

Number of bins in each dimension, specified as a positive scalar integer or two-element vector of positive integers. If you do not specify `nbins`, then `histogram2` automatically calculates how many bins to use based on the values in X and Y.

- If `nbins` is a scalar, then `histogram2` uses that many bins in each dimension.
- If `nbins` is a vector, then `nbins(1)` specifies the number of bins in the *x*-dimension and `nbins(2)` specifies the number of bins in the *y*-dimension.

Example: `histogram2(X,Y,20)` uses 20 bins in each dimension.

Example: `histogram2(X,Y,[10 20])` uses 10 bins in the x-dimension and 20 bins in the y-dimension.

Xedges — Bin edges in *x*-dimension
vector

Bin edges in *x*-dimension, specified as a vector. `Xedges(1)` is the first edge of the first bin in the *x*-dimension, and `Xedges(end)` is the outer edge of the last bin.

The value `[X(k),Y(k)]` is in the `(i,j)`th bin if `Xedges(i)` ≤ `X(k)` < `Xedges(i+1)` **and** `Yedges(j)` ≤ `Y(k)` < `Yedges(j+1)`. The last bins in each dimension also include the last (outer) edge. For example, `[X(k),Y(k)]` falls into the `i`th bin in the last row if `Xedges(end-1)` ≤ `X(k)` ≤ `Xedges(end)` **and** `Yedges(i)` ≤ `Y(k)` < `Yedges(i+1)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Yedges — Bin edges in *y*-dimension
vector

Bin edges in *y*-dimension, specified as a vector. `Yedges(1)` is the first edge of the first bin in the *y*-dimension, and `Yedges(end)` is the outer edge of the last bin.

The value `[X(k),Y(k)]` is in the `(i,j)`th bin if `Xedges(i)` ≤ `X(k)` < `Xedges(i+1)` **and** `Yedges(j)` ≤ `Y(k)` < `Yedges(j+1)`. The last bins in each dimension also include the last (outer) edge. For example, `[X(k),Y(k)]` falls into the `i`th bin in the last row if `Xedges(end-1)` ≤ `X(k)` ≤ `Xedges(end)` **and** `Yedges(i)` ≤ `Y(k)` < `Yedges(i+1)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`counts` — Bin counts
matrix

Bin counts, specified as a matrix. Use this input to pass bin counts to `histogram2` when the bin counts calculation is performed separately and you do not want `histogram2` to do any data binning.

`counts` must be a matrix of size `[length(XBinEdges)-1 length(YBinEdges)-1]` so that it specifies a bin count for each bin.

Example: `histogram2('XBinEdges',-1:1,'YBinEdges',-2:2,'BinCounts',[1 2 3 4; 5 6 7 8])`

`ax` — Axes object
object

Axes object. If you do not specify an axes, then the `histogram2` function uses the current axes (`gca`).

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `histogram2(X,Y,'BinWidth',[5 10])`

The properties listed here are only a subset. For a complete list, see Histogram2.

`BinMethod` — Binning algorithm
`'auto'` (default) | `'scott'` | `'fd'` | `'integers'`

Binning algorithm, specified as one of the values in this table.

Value	Description
`'auto'`	The default `'auto'` algorithm chooses a bin width to cover the data range and reveal the shape of the underlying distribution.
`'scott'`	Scott's rule is optimal if the data is close to being jointly normally distributed. This rule is appropriate for most other distributions, as well. It uses a bin size of `[3.5*std(X(:))*numel(X)^(-1/4), 3.5*std(Y(:))*numel(Y)^(-1/4)]`.
`'fd'`	The Freedman-Diaconis rule is less sensitive to outliers in the data, and might be more suitable for data with heavy-tailed distributions. It uses a bin size of `[2*IQR(X(:))*numel(X)^(-1/4), 2*IQR(Y(:))*numel(Y)^(-1/4)]`, where `IQR` is the interquartile range.
`'integers'`	The integer rule is useful with integer data, as it creates bins centered on pairs of integers. It uses a bin width of 1 for each dimension and places bin edges halfway between integers. To avoid accidentally creating too many bins, you can use this rule to create a limit of 1024 bins (2^{10}). If the data range for either dimension is greater than 1024, then the integer rule uses wider bins instead.

`histogram2` does not always choose the number of bins using these exact formulas. Sometimes the number of bins is adjusted slightly so that the bin edges fall on "nice" numbers.

Note If you set the `NumBins`, `XBinEdges`, `YBinEdges`, `BinWidth`, or `BinLimits` property, then the `BinMethod` property is set to `'manual'`.

Example: `histogram2(X,Y,'BinMethod','integers')` creates a bivariate histogram with the bins centered on pairs of integers.

`BinWidth` — Width of bins in each dimension
vector

Width of bins in each dimension, specified as a two-element vector of positive integers, `[xWidth yWidth]`.

If you specify `BinWidth`, then `histogram2` can use a maximum of 1024 bins (2^{10}) along each dimension. If instead the specified bin width requires more bins, then `histogram2` uses a larger bin width corresponding to the maximum number of bins.

Example: `histogram2(X,Y,'BinWidth',[5 10])` uses bins with size `5` in the `x`-dimension and size `10` in the `y`-dimension.

`DisplayStyle` — Histogram display style
`'bar3'` (default) | `'tile'`

Histogram display style, specified as either `'bar3'` or `'tile'`. Specify `'tile'` to display the histogram as a rectangular array of tiles with colors indicating the bin values.

The default value of `'bar3'` displays the histogram using 3-D bars.

Example: `histogram2(X,Y,'DisplayStyle','tile')` plots the histogram as a rectangular array of tiles.

`EdgeAlpha` — Transparency of histogram bar edges
`1` (default) | scalar value between `0` and `1` inclusive

Transparency of histogram bar edges, specified as a scalar value between `0` and `1` inclusive. A value of `1` means fully opaque and `0` means completely transparent (invisible).

Example: `histogram2(X,Y,'EdgeAlpha',0.5)` creates a bivariate histogram plot with semi-transparent bar edges.

`EdgeColor` — Histogram edge color
`[0.15 0.15 0.15]` (default) | `'none'` | `'auto'` | RGB triplet | hexadecimal color code | color name

Histogram edge color, specified as one of these values:

- `'none'` — Edges are not drawn.

- `'auto'` — Color of each edge is chosen automatically.
- RGB triplet, hexadecimal color code, or color name — Edges use the specified color.

  RGB triplets and hexadecimal color codes are useful for specifying custom colors.

  - An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
  - A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

  Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

  Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `histogram2(X,Y,'EdgeColor','r')` creates a 3-D histogram plot with red bar edges.

`FaceAlpha` — Transparency of histogram bars

`1` (default) | scalar value between `0` and `1` inclusive

Transparency of histogram bars, specified as a scalar value between `0` and `1` inclusive. `histogram2` uses the same transparency for all the bars of the histogram. A value of `1` means fully opaque and `0` means completely transparent (invisible).

Example: `histogram2(X,Y,'FaceAlpha',0.5)` creates a bivariate histogram plot with semi-transparent bars.

`FaceColor` — Histogram bar color

`'auto'` (default) | `'flat'` | `'none'` | RGB triplet | hexadecimal color code | color name

Histogram bar color, specified as one of these values:

- `'none'` — Bars are not filled.
- `'flat'` — Bar colors vary with height. Bars with different height have different colors. The colors are selected from the figure or axes colormap.
- `'auto'` — Bar color is chosen automatically (default).
- RGB triplet, hexadecimal color code, or color name — Bars are filled with the specified color.

  RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

If you specify `DisplayStyle` as `'stairs'`, then `histogram2` does not use the `FaceColor` property.

Example: `histogram2(X,Y,'FaceColor','g')` creates a 3-D histogram plot with green bars.

`FaceLighting` — Lighting effect on histogram bars
`'lit'` (default) | `'flat'` | `'none'`

Lighting effect on histogram bars, specified as one of the values in this table.

Value	Description
`'lit'`	Histogram bars display a pseudo-lighting effect, where the sides of the bars use darker colors relative to the tops. The bars are unaffected by other light sources in the axes. This is the default value when `DisplayStyle` is `'bar3'`.
`'flat'`	Histogram bars are not lit automatically. In the presence of other light objects, the lighting effect is uniform across the bar faces.
`'none'`	Histogram bars are not lit automatically, and lights do not affect the histogram bars. `FaceLighting` can only be `'none'` when `DisplayStyle` is `'tile'`.

Example: `histogram2(X,Y,'FaceLighting','none')` turns off the lighting of the histogram bars.

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Width of bar outlines

`0.5` (default) | positive value

Width of bar outlines, specified as a positive value in point units. One point equals 1/72 inch.

Example: `1.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Normalization` — Type of normalization

`'count'` (default) | `'probability'` | `'countdensity'` | `'pdf'` | `'cumcount'` | `'cdf'`

Type of normalization, specified as one of the values in this table. For each bin `i`:

- v_i is the bin value.
- c_i is the number of elements in the bin.
- $A_i = w_{xi} \cdot w_{yi}$ is the area of each bin, computed using the x and y bin widths.
- N is the number of elements in the input data. This value can be greater than the binned data if the data contains `NaN` values, or if some of the data lies outside the bin limits.

Value	Bin Values	Notes
`'count'` (default)	$v_i = c_i$	• Count or frequency of observations. • Sum of bin values is less than or equal to `numel(X)` and `numel(y)`. The sum is less than `numel(X)` and `numel(y)` only when some of the input data is not included in the bins.
`'countdensity'`	$v_i = \frac{c_i}{A_i}$	• Count or frequency scaled by area of bin. • The volume (height * area) of each bar is the number of observations in the bin. The sum of the bar volumes is less than or equal to `numel(X)` and `numel(y)`.
`'cumcount'`	$v_i = \sum_{j=1}^{i} c_j$	• Cumulative count. Each bin value is the cumulative number of observations in each bin and all previous bins in both the x and y dimensions. • The height of the last bar is less than or equal to `numel(X)` and `numel(Y)`.
`'probability'`	$v_i = \frac{c_i}{N}$	• Relative probability. • The sum of the bar heights is less than or equal to 1.

Value	Bin Values	Notes
`'pdf'`	$v_i = \frac{c_i}{N \cdot A_i}$	• Probability density function estimate. • The volume of each bar is the relative number of observations. The sum of the bar volumes is less than or equal to 1.
`'cdf'`	$v_i = \sum_{j=1}^{i} \frac{c_j}{N}$	• Cumulative density function estimate. • The height of each bar is equal to the cumulative relative number of observations in each bin and all previous bins in both the *x* and *y* dimensions. The height of the last bar is less than or equal to 1.

Example: `histogram2(X,Y,'Normalization','pdf')` plots an estimate of the probability density function for X and Y.

`ShowEmptyBins` — Toggle display of empty bins
`'off'` (default) | `'on'`

Toggle display of empty bins, specified as either `'off'` or `'on'`. The default value is `'off'`.

Example: `histogram2(X,Y,'ShowEmptyBins','on')` turns on the display of empty bins.

`XBinLimits` — Bin limits in *x*-dimension
vector

Bin limits in *x*-dimension, specified as a two-element vector, `[xbmin,xbmax]`. The vector indicates the first and last bin edges in the *x*-dimension.

`histogram2` only plots data that falls within the bin limits inclusively, `Data(Data(:,1)>=xbmin & Data(:,1)<=xbmax)`.

`XBinLimitsMode` — Selection mode for bin limits in *x*-dimension
`'auto'` (default) | `'manual'`

Selection mode for bin limits in *x*-dimension, specified as `'auto'` or `'manual'`. The default value is `'auto'`, so that the bin limits automatically adjust to the data along the x-axis.

If you explicitly specify either `XBinLimits` or `XBinEdges`, then `XBinLimitsMode` is set automatically to `'manual'`. In that case, specify `XBinLimitsMode` as `'auto'` to rescale the bin limits to the data.

`YBinLimits` — Bin limits in *y*-dimension
vector

Bin limits in *y*-dimension, specified as a two-element vector, `[ybmin,ybmax]`. The vector indicates the first and last bin edges in the *y*-dimension.

`histogram2` only plots data that falls within the bin limits inclusively, `Data(Data(:,2)>=ybmin & Data(:,2)<=ybmax)`.

`YBinLimitsMode` — Selection mode for bin limits in *y*-dimension
`'auto'` (default) | `'manual'`

Selection mode for bin limits in *y*-dimension, specified as `'auto'` or `'manual'`. The default value is `'auto'`, so that the bin limits automatically adjust to the data along the y-axis.

If you explicitly specify either `YBinLimits` or `YBinEdges`, then `YBinLimitsMode` is set automatically to `'manual'`. In that case, specify `YBinLimitsMode` as `'auto'` to rescale the bin limits to the data.

Output Arguments

h — Bivariate histogram
object

Bivariate histogram, returned as an object. For more information, see Histogram2 Properties.

Properties

Histogram2 Histogram2 appearance and behavior

Object Functions

morebins Increase number of histogram bins
fewerbins Decrease number of histogram bins

Examples

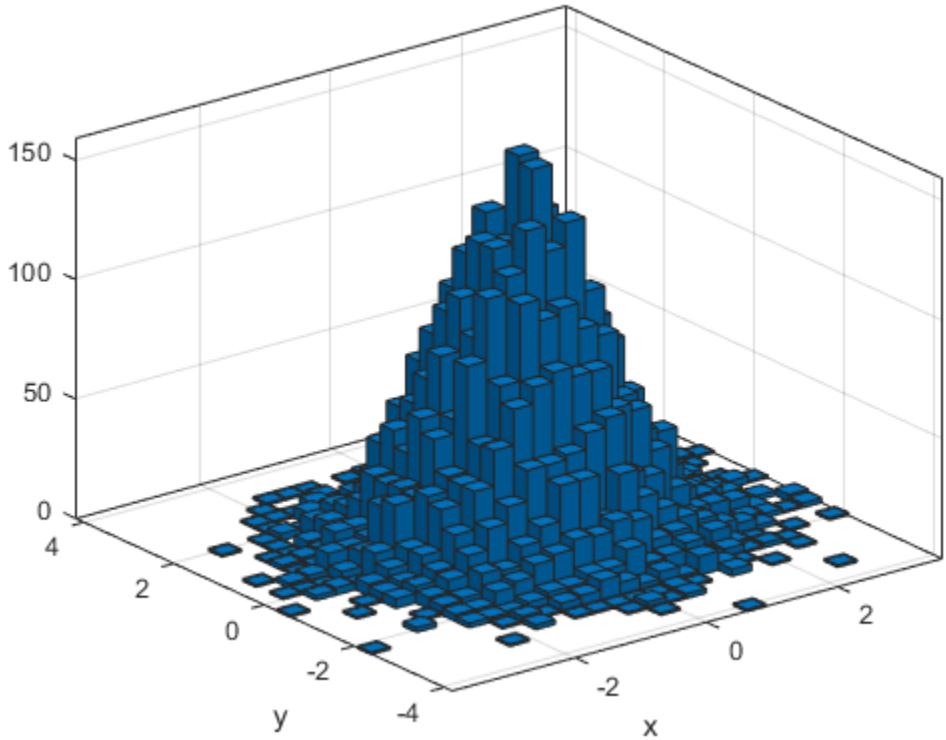
Histogram of Vectors

Generate 10,000 pairs of random numbers and create a bivariate histogram. The `histogram2` function automatically chooses an appropriate number of bins to cover the range of values in `x` and `y` and show the shape of the underlying distribution.

```
x = randn(10000,1);
y = randn(10000,1);
h = histogram2(x,y)
```

```
h = 
  Histogram2 with properties:

             Data: [10000x2 double]
           Values: [25x28 double]
          NumBins: [25 28]
        XBinEdges: [1x26 double]
        YBinEdges: [1x29 double]
         BinWidth: [0.3000 0.3000]
    Normalization: 'count'
        FaceColor: 'auto'
        EdgeColor: [0.1500 0.1500 0.1500]

  Show all properties
```

```
xlabel('x')
ylabel('y')
```



When you specify an output argument to the `histogram2` function, it returns a histogram2 object. You can use this object to inspect the properties of the histogram, such as the number of bins or the width of the bins.

Find the number of histogram bins in each dimension.

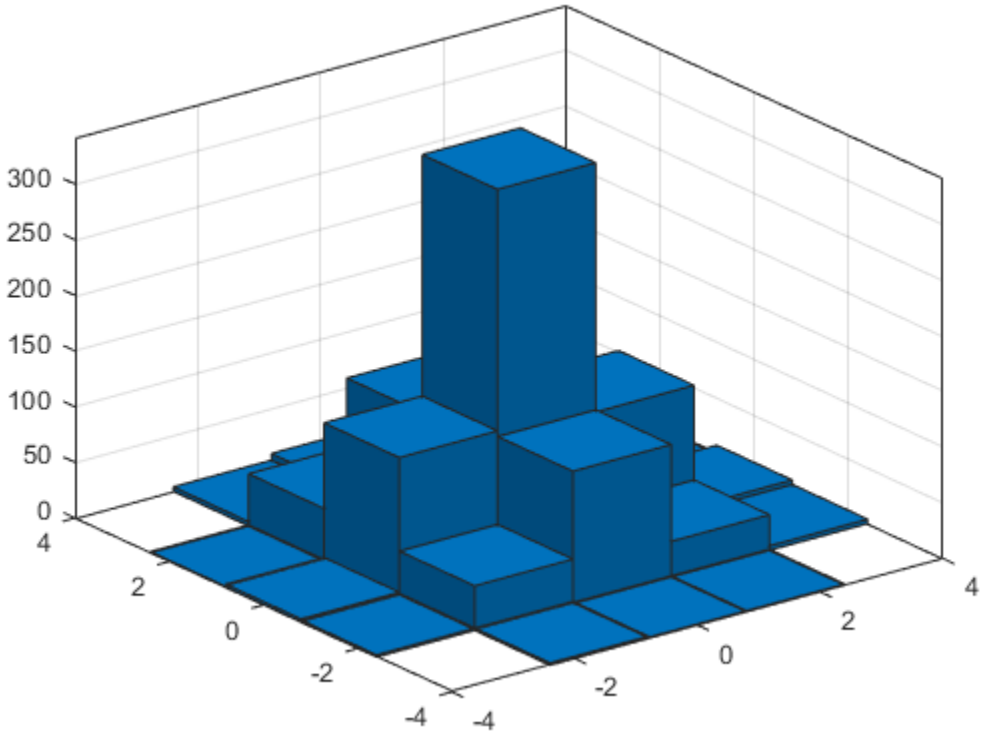
```
nXnY = h.NumBins
```

```
nXnY = 1×2

    25    28
```

Specify Number of Histogram Bins

Plot a bivariate histogram of 1,000 pairs of random numbers sorted into 25 equally spaced bins, using 5 bins in each dimension.

```
x = randn(1000,1);
y = randn(1000,1);
nbins = 5;
h = histogram2(x,y,nbins)
```




```
h =
  Histogram2 with properties:

              Data: [1000x2 double]
```

```
           Values: [5x5 double]
          NumBins: [5 5]
        XBinEdges: [-4 -2.4000 -0.8000 0.8000 2.4000 4]
        YBinEdges: [-4 -2.4000 -0.8000 0.8000 2.4000 4]
         BinWidth: [1.6000 1.6000]
    Normalization: 'count'
        FaceColor: 'auto'
        EdgeColor: [0.1500 0.1500 0.1500]

  Show all properties
```

Find the resulting bin counts.

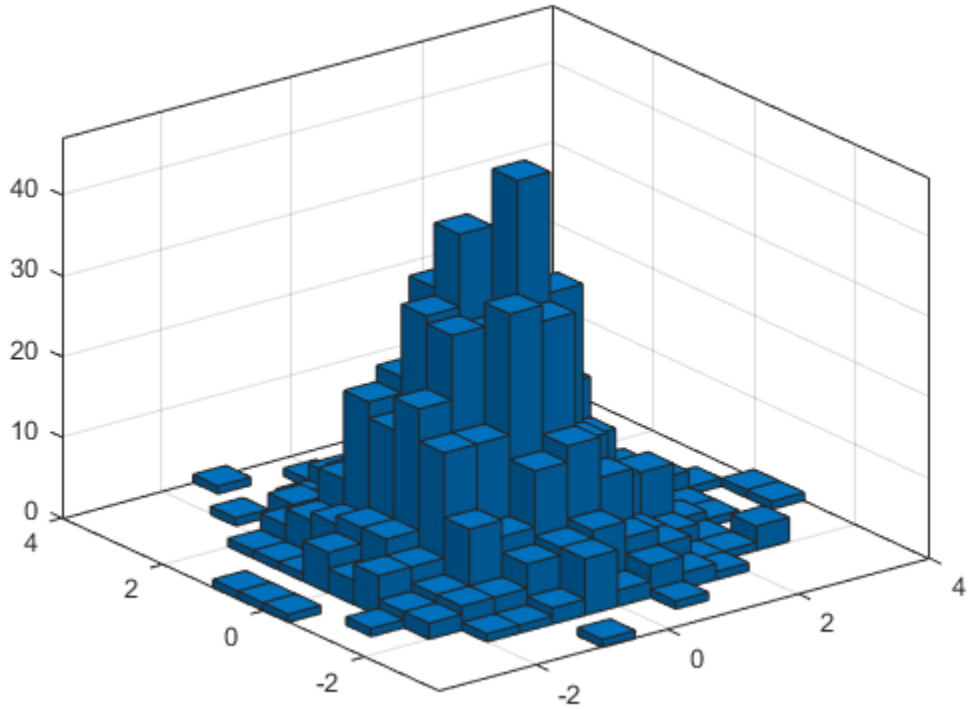
```
counts = h.Values
```

```
counts = 5×5

     0     2     3     1     0
     2    40   124    47     4
     1   119   341   109    10
     1    32   117    33     1
     0     4     8     1     0
```

Adjust Number of Histogram Bins

Generate 1,000 pairs of random numbers and create a bivariate histogram.

```
x = randn(1000,1);
y = randn(1000,1);
h = histogram2(x,y)
```

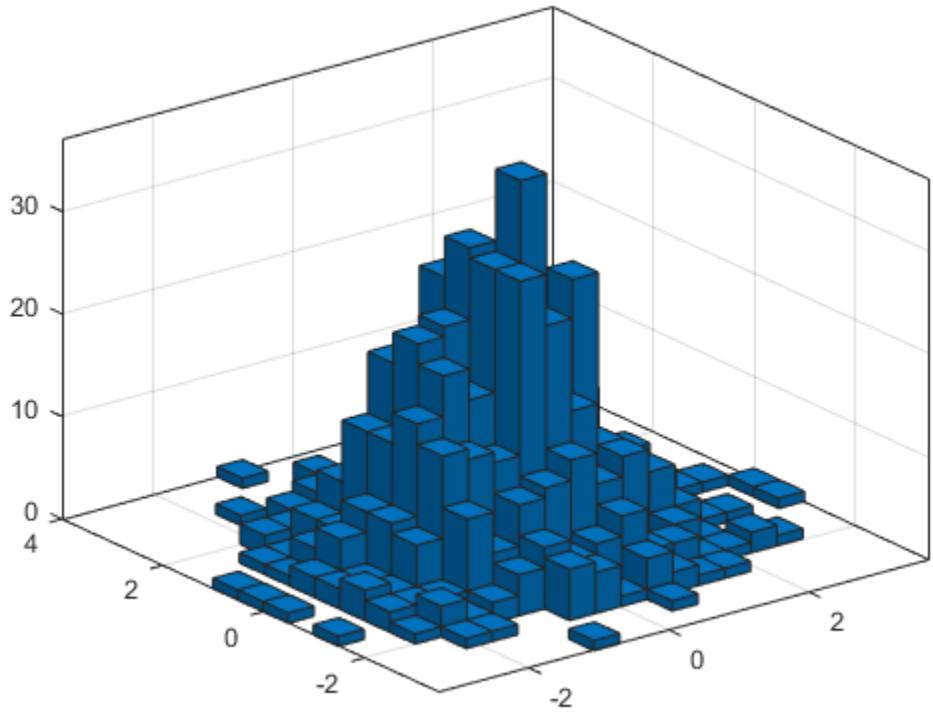



```
h =
  Histogram2 with properties:

             Data: [1000x2 double]
           Values: [15x15 double]
          NumBins: [15 15]
        XBinEdges: [1x16 double]
        YBinEdges: [1x16 double]
         BinWidth: [0.5000 0.5000]
    Normalization: 'count'
        FaceColor: 'auto'
        EdgeColor: [0.1500 0.1500 0.1500]
```

```
Show all properties
```

Use the `morebins` function to coarsely adjust the number of bins in the *x* dimension.

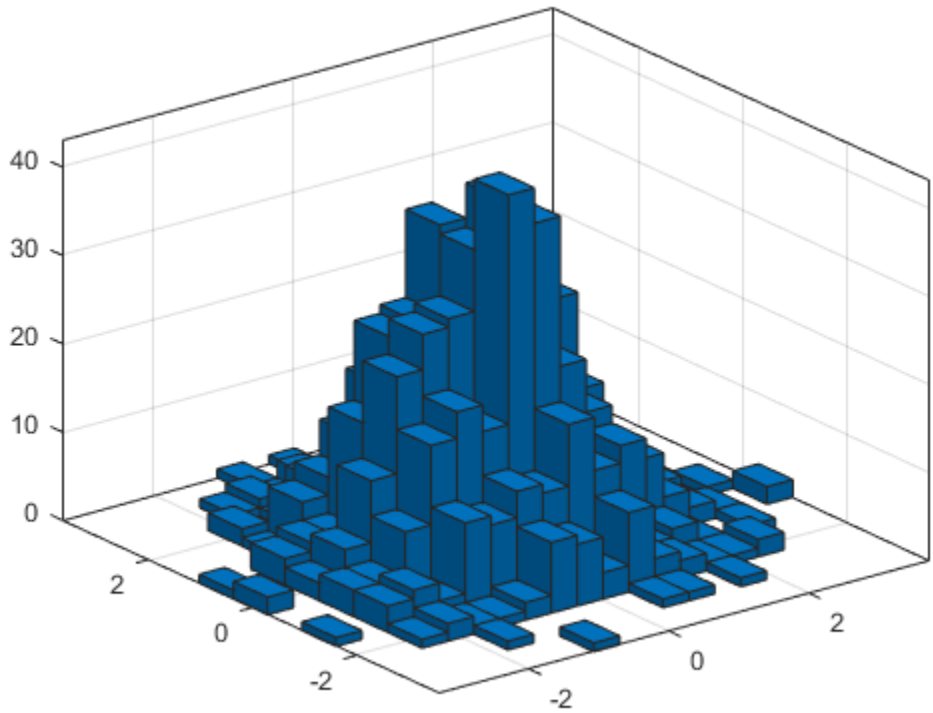
```
nbins = morebins(h,'x');
nbins = morebins(h,'x')
```




```
nbins = 1×2

    19    15
```

Use the `fewerbins` function to adjust the number of bins in the *y* dimension.

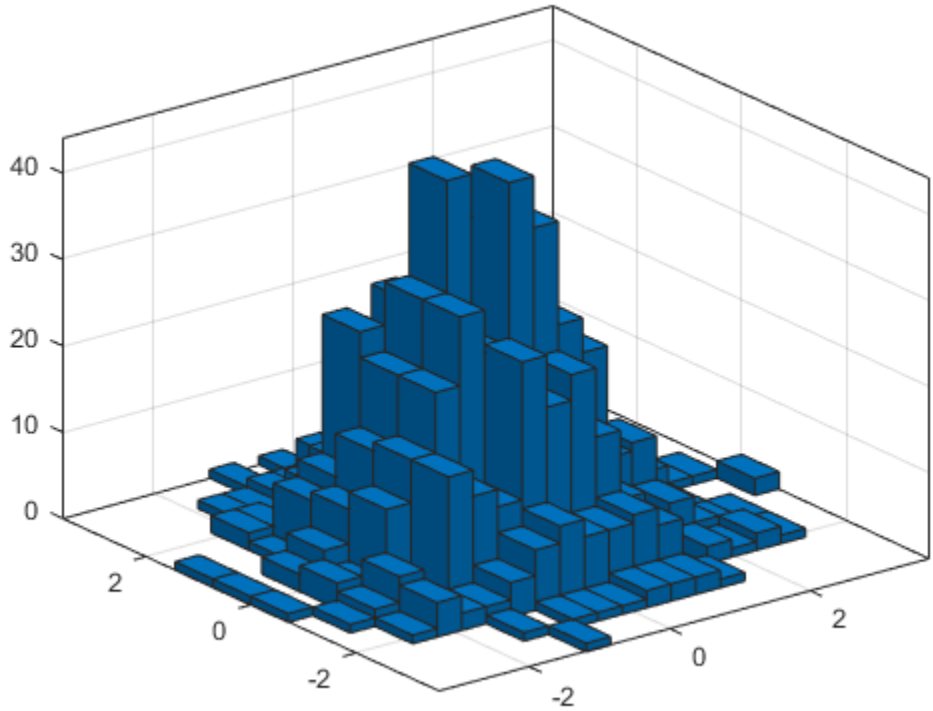
```
nbins = fewerbins(h,'y');
nbins = fewerbins(h,'y')
```




```
nbins = 1×2

    19    11
```

Adjust the number of bins at a fine grain level by explicitly setting the number of bins.

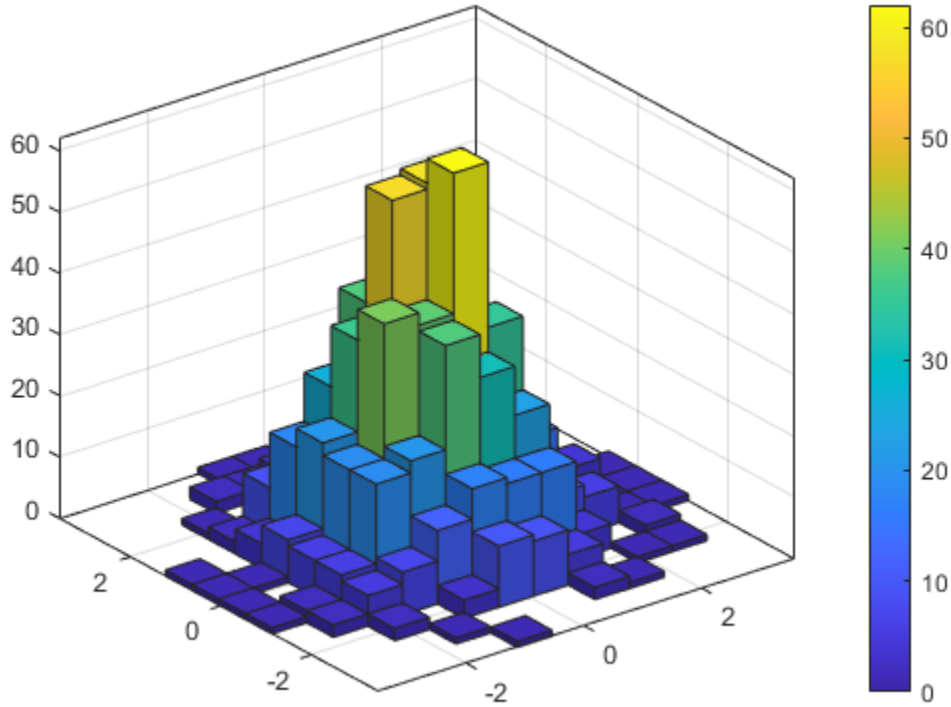
```
h.NumBins = [20 10];
```



Color Histogram Bars by Height

Create a bivariate histogram using 1,000 normally distributed random numbers with 12 bins in each dimension. Specify `FaceColor` as `'flat'` to color the histogram bars by height.

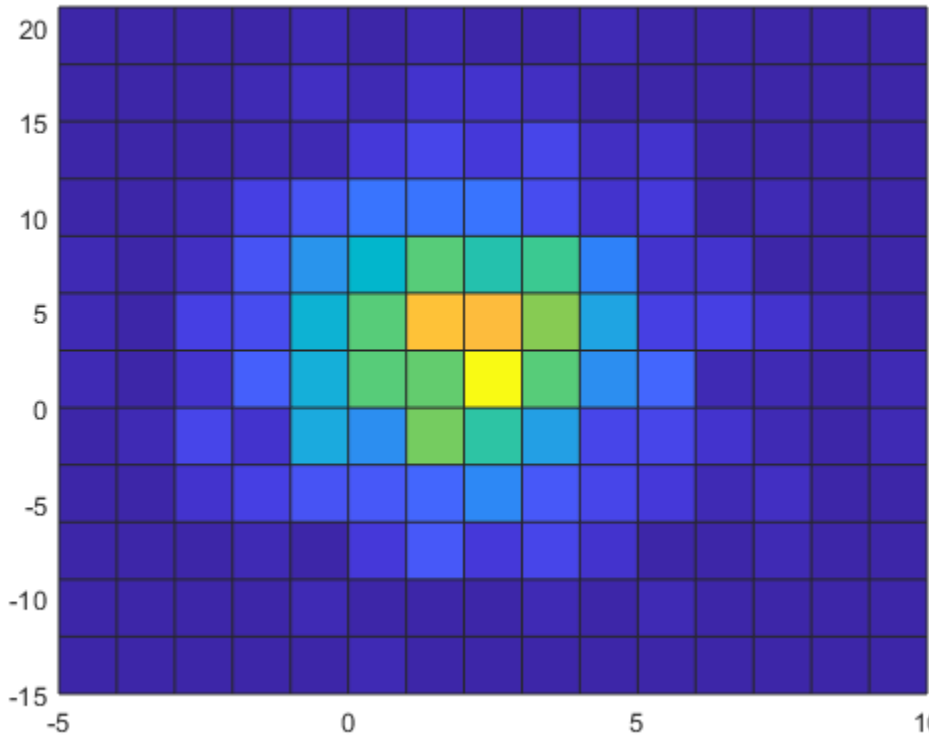
```
h = histogram2(randn(1000,1),randn(1000,1),[12 12],'FaceColor','flat');
colorbar
```



Tiled Histogram View

Generate random data and plot a bivariate tiled histogram. Display the empty bins by specifying `ShowEmptyBins` as `'on'`.

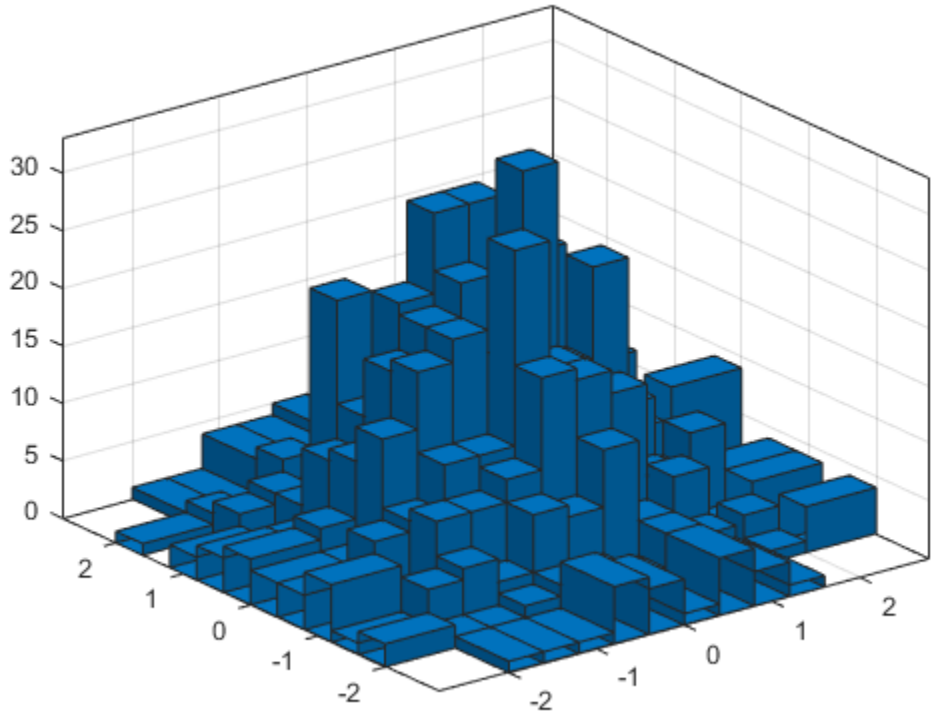
```
x = 2*randn(1000,1)+2;
y = 5*randn(1000,1)+3;
h = histogram2(x,y,'DisplayStyle','tile','ShowEmptyBins','on');
```



Specify Bin Edges of Histogram

Generate 1,000 pairs of random numbers and create a bivariate histogram. Specify the bin edges using two vectors, with infinitely wide bins on the boundary of the histogram to capture all outliers that do not satisfy $|x| < 2$.

```
x = randn(1000,1);
y = randn(1000,1);
Xedges = [-Inf -2:0.4:2 Inf];
Yedges = [-Inf -2:0.4:2 Inf];
h = histogram2(x,y,Xedges,Yedges)
```



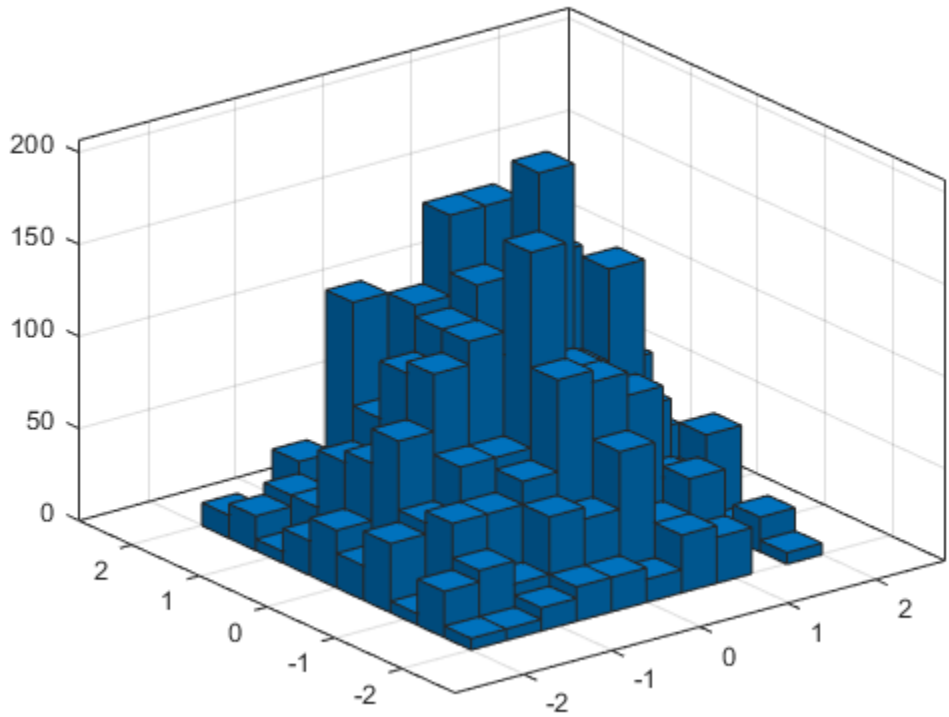

```
h =
  Histogram2 with properties:

             Data: [1000x2 double]
           Values: [12x12 double]
          NumBins: [12 12]
        XBinEdges: [1x13 double]
        YBinEdges: [1x13 double]
         BinWidth: 'nonuniform'
    Normalization: 'count'
        FaceColor: 'auto'
        EdgeColor: [0.1500 0.1500 0.1500]
```

```
Show all properties
```

When the bin edges are infinite, `histogram2` displays each outlier bin (along the boundary of the histogram) as being double the width of the bin next to it.

Specify the `Normalization` property as `'countdensity'` to remove the bins containing the outliers. Now the *volume* of each bin represents the frequency of observations in that interval.

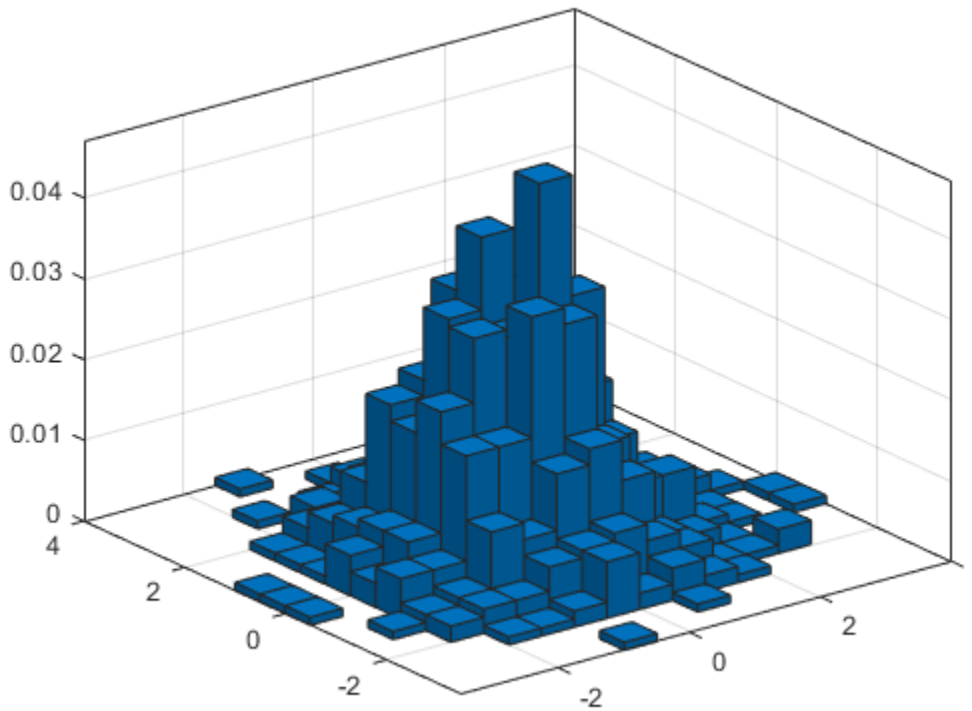
```
h.Normalization = 'countdensity';
```



Normalized Histogram

Generate 1,000 pairs of random numbers and create a bivariate histogram using the `'probability'` normalization.

```
x = randn(1000,1);
y = randn(1000,1);
h = histogram2(x,y,'Normalization','probability')
```




```
h =
  Histogram2 with properties:

             Data: [1000x2 double]
           Values: [15x15 double]
```

```
          NumBins: [15 15]
        XBinEdges: [1x16 double]
        YBinEdges: [1x16 double]
         BinWidth: [0.5000 0.5000]
    Normalization: 'probability'
        FaceColor: 'auto'
        EdgeColor: [0.1500 0.1500 0.1500]

  Show all properties
```

Compute the total sum of the bar heights. With this normalization, the height of each bar is equal to the probability of selecting an observation within that bin interval, and the heights of all of the bars sum to 1.

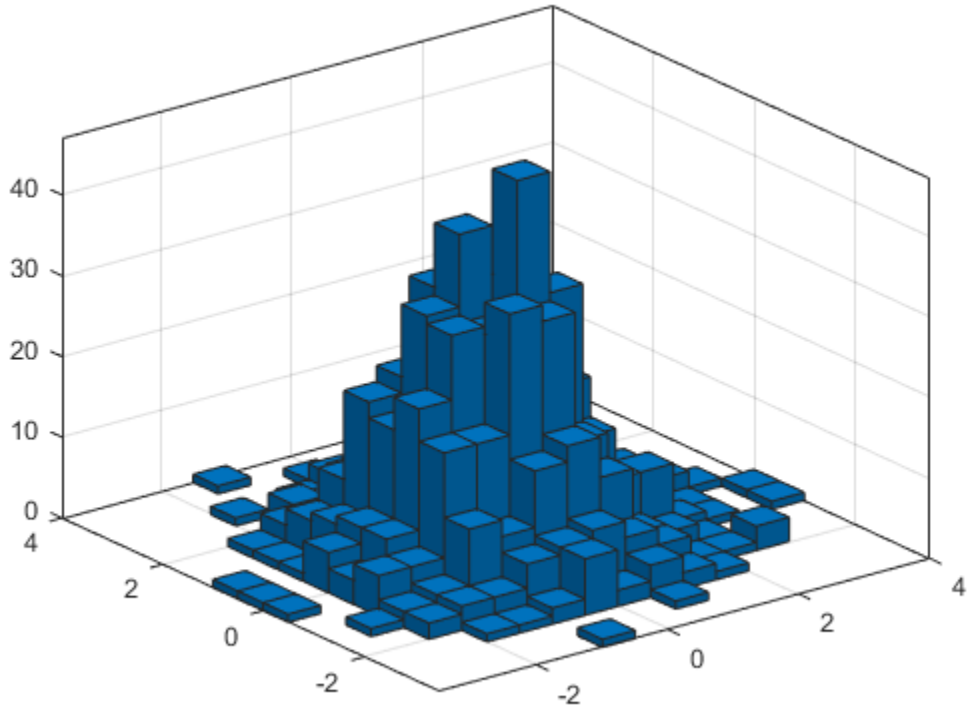
```
S = sum(h.Values(:))
```

```
S = 1.0000
```

Adjust Histogram Properties

Generate 1,000 pairs of random numbers and create a bivariate histogram. Return the histogram object to adjust the properties of the histogram without recreating the entire plot.

```
x = randn(1000,1);
y = randn(1000,1);
h = histogram2(x,y)
```

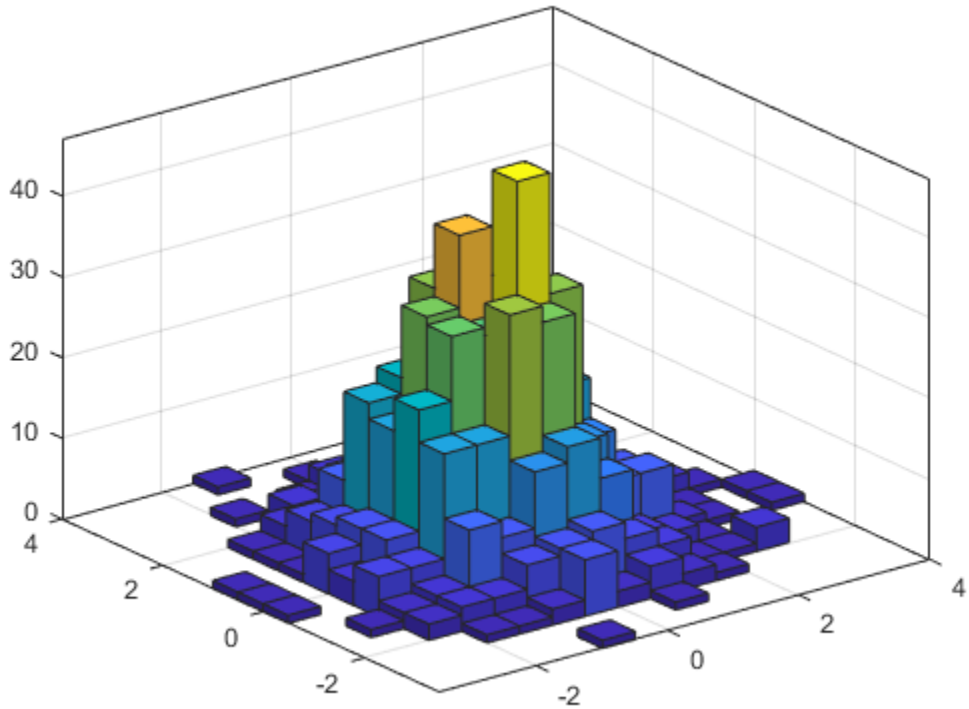



```
h =
  Histogram2 with properties:

             Data: [1000x2 double]
           Values: [15x15 double]
          NumBins: [15 15]
        XBinEdges: [1x16 double]
        YBinEdges: [1x16 double]
         BinWidth: [0.5000 0.5000]
    Normalization: 'count'
        FaceColor: 'auto'
        EdgeColor: [0.1500 0.1500 0.1500]
```

```
Show all properties
```

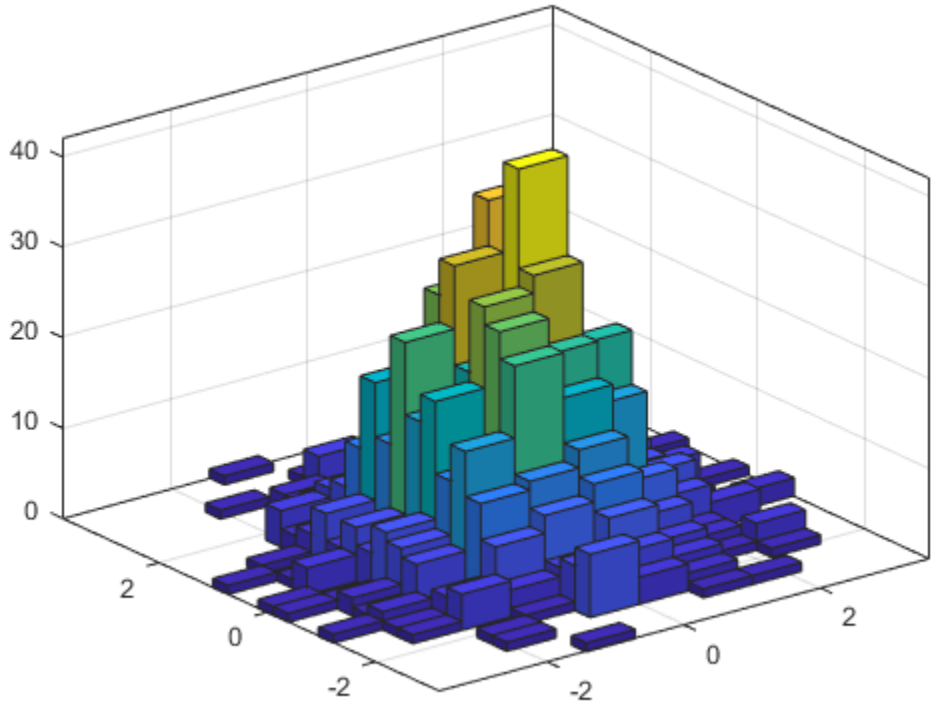
Color the histogram bars by height.

```
h.FaceColor = 'flat';
```



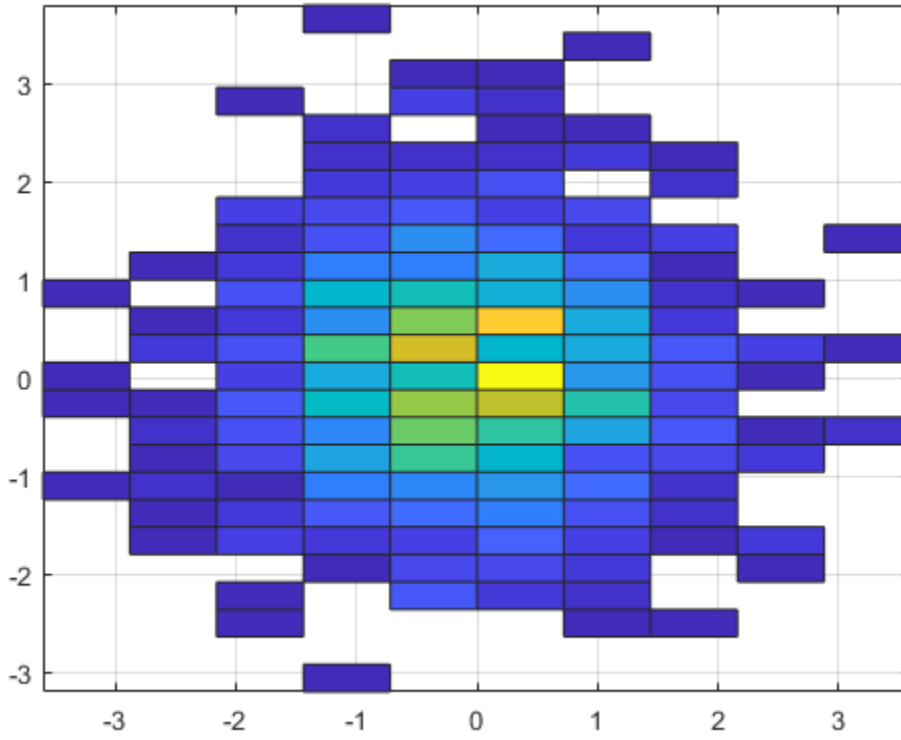
Change the number of bins in each direction.

```
h.NumBins = [10 25];
```



Display the histogram as a tile plot.

```
h.DisplayStyle = 'tile';
view(2)
```



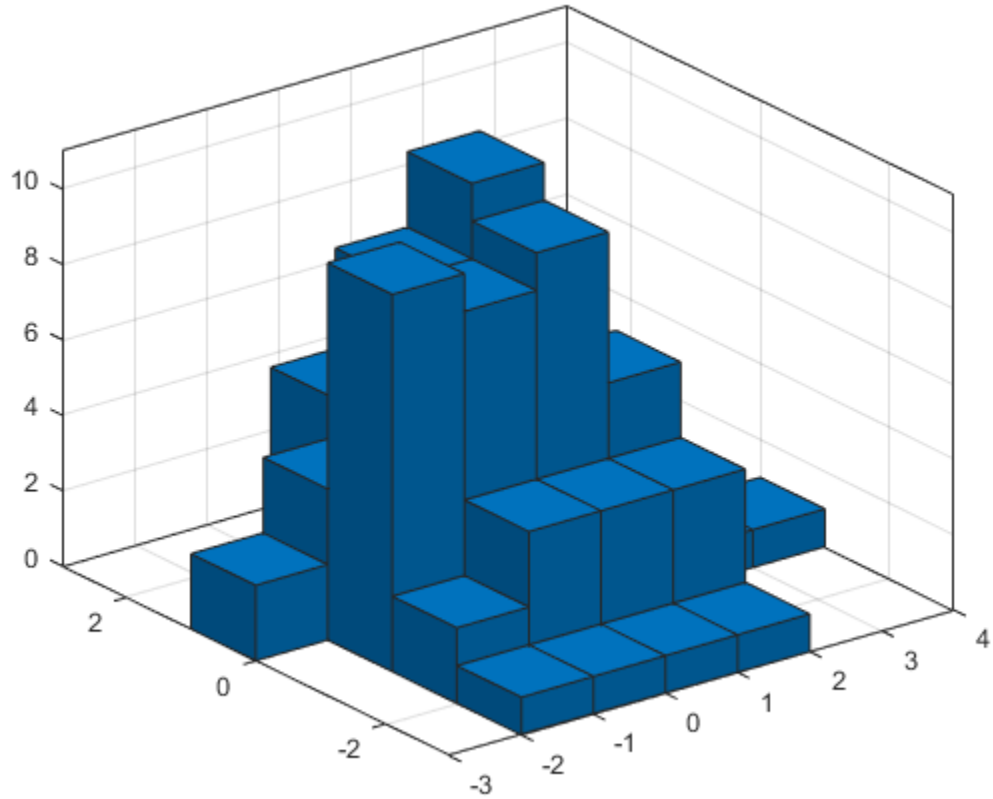
Saving and Loading Histogram2 Objects

Use the `savefig` function to save a histogram2 figure.

```
y = histogram2(randn(100,1),randn(100,1));
savefig('histogram2.fig');
clear all
close all
```

Use `openfig` to load the histogram figure back into MATLAB. `openfig` also returns a handle to the figure, `h`.

```
h = openfig('histogram2.fig');
```



Use the `findobj` function to locate the correct object handle from the figure handle. This allows you to continue manipulating the original histogram object used to generate the figure.

```
y = findobj(h, 'type', 'histogram2')
```

```
y 
  Histogram2 with properties:

              Data: [100x2 double]
```

```
          Values: [7x6 double]
         NumBins: [7 6]
       XBinEdges: [-3 -2 -1 0 1 2 3 4]
       YBinEdges: [-3 -2 -1 0 1 2 3]
        BinWidth: [1 1]
   Normalization: 'count'
       FaceColor: 'auto'
       EdgeColor: [0.1500 0.1500 0.1500]

  Show all properties
```

Tips

- Histogram plots created using `histogram2` have a context menu in plot edit mode that enables interactive manipulations in the figure window. For example, you can use the context menu to interactively change the number of bins, align multiple histograms, or change the display order.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

- Some input options are not supported. The allowed options are:
  - `'BinWidth'`
  - `'XBinLimits'`
  - `'YBinLimits'`
  - `'Normalization'`
  - `'DisplayStyle'`
  - `'BinMethod'` — The `'auto'` and `'scott'` bin methods are the same. The `'fd'` bin method is not supported.

  - `'EdgeAlpha'`
  - `'EdgeColor'`
  - `'FaceAlpha'`
  - `'FaceColor'`
  - `'LineStyle'`
  - `'LineWidth'`
  - `'Orientation'`
- Additionally, there is a cap on the maximum number of bars. The default maximum is 100.
- The `morebins` and `fewerbins` methods are not supported.
- Editing properties of the histogram object that require recomputing the bins is not supported.

For more information, see “Tall Arrays for Out-of-Memory Data”.

See Also

Histogram2 | `bar3` | `discretize` | `fewerbins` | `histcounts` | `histcounts2` | `morebins`

Introduced in R2015b

Histogram2 Properties

Histogram2 appearance and behavior

Description

Histogram2 properties control the appearance and behavior of the histogram. By changing property values, you can modify aspects of the histogram. Use dot notation to refer to a particular object and property:

```
h = histogram2(randn(10,1),randn(10,1));
c = h.NumBins;
h.NumBins = [4 7];
```

Properties

Bins

`NumBins` — Number of bins in each dimension

vector

Number of bins in each dimension, specified as a two-element vector of positive integers, `[nX nY]`. If you do not specify `NumBins`, then `histogram2` automatically calculates how many bins to use based on the values in `X` and `Y`.

Example: `histogram2(X,Y,[10 20])`

Example: `h.NumBins = [10 20]`

`BinWidth` — Width of bins in each dimension

vector

Width of bins in each dimension, specified as a two-element vector. The first element in the vector gives the width of the bins in the *x*-dimension, and the second element gives the width of the bins in the *y*-dimension.

When you specify `BinWidth`, then `histogram2` can use a maximum of 1024 bins (2^{10}) along each dimension. If instead the specified bin width requires more bins, then `histogram2` uses a larger bin width corresponding to the maximum number of bins.

Example: `histogram2(X,Y,'BinWidth',[5 10])` uses bins with size `5` in the `x`-dimension and size `10` in the `y`-dimension.

`XBinEdges` — Bin edges in *x*-dimension
vector

Bin edges in *x*-dimension, specified as a vector. `Xedges(1)` is the first edge of the first bin in the *x*-dimension, and `Xedges(end)` is the outer edge of the last bin.

The value `[X(k),Y(k)]` is in the `(i,j)`th bin if `Xedges(i)` ≤ `X(k)` < `Xedges(i+1)` **and** `Yedges(j)` ≤ `Y(k)` < `Yedges(j+1)`. The last bins in each dimension also include the last (outer) edge. For example, `[X(k),Y(k)]` falls into the `i`th bin in the last row if `Xedges(end-1)` ≤ `X(k)` ≤ `Xedges(end)` **and** `Yedges(i)` ≤ `Y(k)` < `Yedges(i+1)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`YBinEdges` — Bin edges in *y*-dimension
vector

Bin edges in *y*-dimension, specified as a vector. `Yedges(1)` is the first edge of the first bin in the *y*-dimension, and `Yedges(end)` is the outer edge of the last bin.

The value `[X(k),Y(k)]` is in the `(i,j)`th bin if `Xedges(i)` ≤ `X(k)` < `Xedges(i+1)` **and** `Yedges(j)` ≤ `Y(k)` < `Yedges(j+1)`. The last bins in each dimension also include the last (outer) edge. For example, `[X(k),Y(k)]` falls into the `i`th bin in the last row if `Xedges(end-1)` ≤ `X(k)` ≤ `Xedges(end)` **and** `Yedges(i)` ≤ `Y(k)` < `Yedges(i+1)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`XBinLimits` — Bin limits in x-dimension
vector

Bin limits in `x`-dimension, specified as a two-element vector, `[xbmin,xbmax]`. The vector indicates the first and last bin edges in the `x`-dimension.

`histogram2` only plots data that falls within the bin limits inclusively, `Data(Data(:,1)>=xbmin & Data(:,1)<=xbmax)`.

`XBinLimitsMode` — Selection mode for bin limits in x-dimension
`'auto'` (default) | `'manual'`

Selection mode for bin limits in x-dimension, specified as `'auto'` or `'manual'`. The default value is `'auto'`, so that the bin limits automatically adjust to the data along the x-axis.

If you explicitly specify either `XBinLimits` or `XBinEdges`, then `XBinLimitsMode` is set automatically to `'manual'`. In that case, specify `XBinLimitsMode` as `'auto'` to rescale the bin limits to the data.

`YBinLimits` — Bin limits in y-dimension
vector

Bin limits in y-dimension, specified as a two-element vector, `[ybmin,ybmax]`. The vector indicates the first and last bin edges in the y-dimension.

`histogram2` only plots data that falls within the bin limits inclusively, `Data(Data(:,2)>=ybmin & Data(:,2)<=ybmax)`.

`YBinLimitsMode` — Selection mode for bin limits in y-dimension
`'auto'` (default) | `'manual'`

Selection mode for bin limits in y-dimension, specified as `'auto'` or `'manual'`. The default value is `'auto'`, so that the bin limits automatically adjust to the data along the y-axis.

If you explicitly specify either `YBinLimits` or `YBinEdges`, then `YBinLimitsMode` is set automatically to `'manual'`. In that case, specify `YBinLimitsMode` as `'auto'` to rescale the bin limits to the data.

`BinMethod` — Binning algorithm
`'auto'` (default) | `'scott'` | `'fd'` | `'integers'`

Binning algorithm, specified as one of the values in this table.

Value	Description
`'auto'`	The default `'auto'` algorithm chooses a bin width to cover the data range and reveal the shape of the underlying distribution.

Value	Description
`'scott'`	Scott's rule is optimal if the data is close to being jointly normally distributed. This rule is appropriate for most other distributions, as well. It uses a bin size of `[3.5*std(X(:))*numel(X)^(-1/4), 3.5*std(Y(:))*numel(Y)^(-1/4)]`.
`'fd'`	The Freedman-Diaconis rule is less sensitive to outliers in the data, and might be more suitable for data with heavy-tailed distributions. It uses a bin size of `[2*IQR(X(:))*numel(X)^(-1/4), 2*IQR(Y(:))*numel(Y)^(-1/4)]`, where `IQR` is the interquartile range.
`'integers'`	The integer rule is useful with integer data, as it creates a bin for each pair of integers `X` and `Y`. It uses a bin width of 1 for each dimension and places bin edges halfway between integers. To avoid accidentally creating too many bins, you can use this rule to create a limit of 1024 bins (2^{10}). If the data range for either dimension is greater than 1024, then the integer rule uses wider bins instead.

Note If you set the `NumBins`, `XBinEdges`, `YBinEdges`, `BinWidth`, or `BinLimits` property, then the `BinMethod` property is set to `'manual'`.

Example: `histogram2(X,Y,'BinMethod','integers')` creates a bivariate histogram with the bins centered on integers.

`ShowEmptyBins` — Toggle display of empty bins
`'off'` (default) | `'on'`

Toggle display of empty bins, specified as either `'off'` or `'on'`. The default value is `'off'`.

Example: `histogram2(X,Y,'ShowEmptyBins','on')` turns on the display of empty bins.

Data

`Data` — Data to distribute among bins

matrix

Data to distribute among bins, specified as a matrix of size `m`-by-`2`. The `X` and `Y` inputs to `histogram2` correspond to the columns in `Data`, that is, `Data(:,1)` is `X(:)` and `Data(:,2)` is `Y(:)`.

`histogram2` ignores all `NaN` values. Similarly,`histogram2` ignores `Inf` and `-Inf` values, unless the bin edges explicitly specify `Inf` or `-Inf` as a bin edge. Although `NaN`, `Inf`, and `-Inf` values are typically not plotted, they are still included in normalization calculations that include the total number of data elements, such as `'probability'`.

If you change the values in the `Data` property of a `histogram2` object, then the bin edges are not automatically updated. To recompute the bins, adjust a bin-related property such as `BinMethod` or `NumBins`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`Values` — Bin values

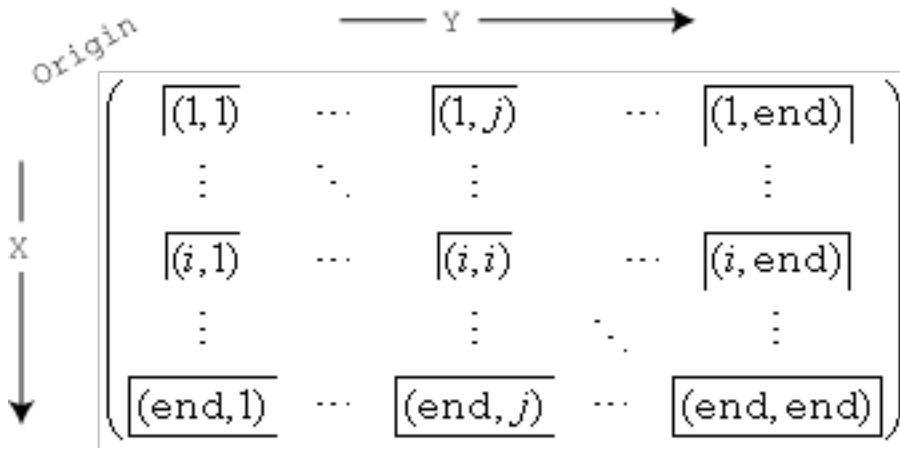
matrix

This property is read-only.

Bin values, returned as a numeric matrix. If `Normalization` is `'count'`, then the `(i,j)`th entry in `Values` specifies the bin count for the bin whose *x* edges are `[Xedges(i), Xedges(i+1)]` and whose *y* edges are `[Yedges(j), Yedges(j+1)]`.

Depending on the value of `Normalization`, the `Values` property instead can contain a normalized variant of the bin counts.

The bin inclusion scheme for the different numbered bins in `Values`, as well as their relative orientation to the *x*-axis and *y*-axis, is



For example, the `(1,1)` bin includes values that fall on the first edge in each dimension, and the last bin in the bottom right includes values that fall on any of its edges.

`Normalization` — Type of normalization
`'count'` (default) | `'probability'` | `'countdensity'` | `'pdf'` | `'cumcount'` | `'cdf'`

Type of normalization, specified as one of the values in the table.

Value	Description
`'count'`	Default normalization scheme. The height of each bar is the number of observations in each bin. The sum of the bar heights is equal to `numel(X)` and `numel(Y)`.
`'probability'`	The height of each bar is the relative number of observations, (Number of observations in bin / Total number of observations). The sum of the bar heights is `1`.
`'countdensity'`	The height of each bar is (Number of observations in bin) / (Area of bin). The volume (Height * Area) of each bar is the number of observations in the bin. The sum of the bar volumes is equal to `numel(X)` and `numel(Y)`.

Value	Description
`'pdf'`	Probability density function estimate. The height of each bar is, (Number of observations in the bin) / (Total number of observations * Area of bin). The volume of each bar is the relative number of observations. The sum of the bar volumes is `1`.
`'cumcount'`	The height of each bar is the cumulative number of observations in each bin and all previous bins in both the *x* and *y* dimensions. The height of the last bar is equal to `numel(X)` and `numel(Y)`.
`'cdf'`	Cumulative density function estimate. The height of each bar is equal to the cumulative relative number of observations in each bin and all previous bins in both the *x* and *y* dimensions. The height of the last bar is `1`.

Example: `histogram2(X,Y,'Normalization','pdf')` plots an estimate of the probability density function for `X` and `Y`.

`BinCounts` — Bin counts
matrix

Bin counts, specified as a matrix. Use this input to pass bin counts to `histogram2` when the bin counts calculation is performed separately and you do not want `histogram2` to do any data binning.

`counts` must be a matrix of size `[nbinsX nbinsY]` so that it specifies a bin count for each bin.

The number of bins in the *x*-dimension is `length(XBinEdges)-1`, and the number of bins in the *y*-dimension is `length(YBinEdges)-1`.

Compared to the `Values` property, `BinCounts` is not normalized. If `Normalization` is `'count'`, then `Values` and `BinCounts` are equivalent.

Example: `histogram2('XBinEdges',-1:1,'YBinEdges',-2:2,'BinCounts',[1 2 3 4; 5 6 7 8])`

BinCountsMode — Selection mode for bin counts

`'auto'` (default) | `'manual'`

Selection mode for bin counts, specified as `'auto'` or `'manual'`. The default value is `'auto'`, so that the bin counts are automatically computed from `Data`, `XBinEdges`, and `YBinEdges`.

If you specify `BinCounts`, then `BinCountsMode` is automatically set to `'manual'`. Similarly, if you specify `Data`, then `BinCountsMode` is automatically set to `'auto'`.

Color and Styling

DisplayStyle — Histogram display style

`'bar3'` (default) | `'tile'`

Histogram display style, specified as either `'bar3'` or `'tile'`. Specify `'tile'` to display the histogram as a rectangular array of tiles with colors indicating the bin values.

The default value of `'bar3'` displays the histogram using 3-D bars.

Example: `histogram2(X,Y,'DisplayStyle','tile')` plots the histogram as a rectangular array of tiles.

FaceColor — Histogram bar color

`'auto'` (default) | `'flat'` | `'none'` | RGB triplet | hexadecimal color code | color name

Histogram bar color, specified as one of these values:

- `'none'` — Bars are not filled.
- `'flat'` — Bar colors vary with height. Bars with different height have different colors. The colors are selected from the figure or axes colormap.
- `'auto'` — Bar color is chosen automatically (default).
- RGB triplet, hexadecimal color code, or color name — Bars are filled with the specified color.

  RGB triplets and hexadecimal color codes are useful for specifying custom colors.

  - An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
  - A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from

`0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

If you specify `DisplayStyle` as `'stairs'`, then `histogram2` does not use the `FaceColor` property.

Example: `histogram2(X,Y,'FaceColor','g')` creates a histogram plot with green bars.

`EdgeColor` — Histogram edge color

`[0.15 0.15 0.15]` (default) | `'none'` | `'auto'` | RGB triplet | hexadecimal color code | color name

Histogram edge color, specified as one of these values:

- `'none'` — Edges are not drawn.
- `'auto'` — Color of each edge is chosen automatically.
- RGB triplet, hexadecimal color code, or color name — Edges use the specified color.

  RGB triplets and hexadecimal color codes are useful for specifying custom colors.

  - An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
  - A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

  Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `histogram2(X,Y,'EdgeColor','r')` creates a histogram plot with red bar edges.

FaceAlpha — Transparency of histogram bars

`1.0` (default) | scalar value between `0` and `1` inclusive

Transparency of histogram bars, specified as a scalar value between `0` and `1` inclusive. `histogram2` uses the same transparency for all the bars of the histogram. A value of `1` means fully opaque and `0` means completely transparent (invisible).

Example: `histogram2(X,Y,'FaceAlpha',0.5)` creates a bivariate histogram plot with semi-transparent bars.

`EdgeAlpha` — Transparency of histogram bar edges
`1` (default) | scalar value between `0` and `1` inclusive

Transparency of histogram bar edges, specified as a scalar value between `0` and `1` inclusive. A value of `1` means fully opaque and `0` means completely transparent (invisible).

Example: `histogram2(X,Y,'EdgeAlpha',0.5)` creates a bivariate histogram plot with semi-transparent bar edges.

`FaceLighting` — Lighting effect on histogram bars
`'lit'` (default) | `'flat'` | `'none'`

Lighting effect on histogram bars, specified as one of the values in the table.

Value	Description
`'lit'`	Histogram bars display a pseudo-lighting effect, where the sides of the bars use darker colors relative to the tops. The bars are unaffected by other light sources in the axes. This is the default value when `DisplayStyle` is `'bar3'`.
`'flat'`	Histogram bars are not lit automatically. In the presence of other light objects, the lighting effect is uniform across the bar faces.
`'none'`	Histogram bars are not lit automatically, and lights do not affect the histogram bars. `FaceLighting` can only be `'none'` when `DisplayStyle` is `'tile'`.

Example: `histogram2(X,Y,'FaceLighting','none')` turns off the lighting of the histogram bars.

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Width of bar outlines
`0.5` (default) | positive value

Width of bar outlines, specified as a positive value in point units. One point equals 1/72 inch.

Example: `1.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Legend

`DisplayName` — Text used by legend
variable names of `Data` or `''` (default) | character vector

Text used by the legend, specified as a character vector. The text appears next to an icon of the histogram2.

Example: `'Text Description'`

For multiline text, create the character vector using `sprintf` with the new line character `\n`.

Example: `sprintf('line one\nline two')`

Alternatively, you can specify the legend text using the `legend` function.

- If you specify the text as an input argument to the `legend` function, then the legend uses the specified text and sets the `DisplayName` property to the same value.

- If you do not specify the text as an input argument to the `legend` function, then the legend uses the text in the `DisplayName` property. By default, `DisplayName` is a character vector representing the variable names of the *x* and *y* input data used to construct the histogram. If one or both of the inputs do not have variable names, then `DisplayName` is empty, `''`.

If the `DisplayName` property does not contain any text, then the legend generates a character vector. The character vector has the form `'dataN'`, where `N` is the number assigned to the histogram2 object based on its location in the list of legend entries.

If you edit interactively the character vector in an existing legend, then MATLAB updates the `DisplayName` property to the edited character vector.

`Annotation` — Control for including or excluding object from legend
`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility
`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`DataTipTemplate` — Data tip content
`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

`UIContextMenu` — Context menu
empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state
`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles
`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.

- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.

- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks only when visible. The `Visible` property must be set to `'on'`. The `HitTest` property determines if the `Histogram2` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `Histogram2` object passes the click to the object behind it in the current view of the figure window. The `HitTest` property of the `Histogram2` object has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `Histogram2` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `Histogram2` object that has one of these:
  - `HitTest` property set to `'on'`
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

Note The `PickableParts` property determines if the `Histogram2` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent

`Axes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `Group`, or `Transform` object.

`Children` — Children

empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the chart object.

HandleVisibility — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'histogram2'` (default)

This property is read-only.

Type of graphics object, returned as `'histogram2'`. Use this property to find all objects of a given type within a plotting hierarchy, such as searching for the type using `findobj`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
[ ] (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`histogram2`

Topics

“Access Property Values”

Introduced in R2015b

hms

Hour, minute, and second numbers of datetime or duration

Syntax

```
[h,m,s] = hms(t)
```

Description

`[h,m,s] = hms(t)` returns the hour, minute, and second values of the datetime or duration values in `t` as separate numeric arrays. The `h` and `m` outputs contain integer values, and the `s` output can contain a fractional part. `h`, `m`, and `s` are the same size as `t`.

Calling `hms` on a datetime array is equivalent to calling the `hour`, `minute`, and `second` functions.

Examples

Find Hour, Minute, and Second Numbers of datetime Array

```
t1 = datetime('now','Format','HH:mm:ss.SSS');
t = t1 + minutes(0:45:135)
```

```
t = 1x4 datetime array
   17:37:49.812   18:22:49.812   19:07:49.812   19:52:49.812
```

```
[h,m,s] = hms(t)
```

```
h = 1×4

    17    18    19    19
```

```
m = 1×4

    37    22     7    52

s = 1×4

   49.8120   49.8120   49.8120   49.8120
```

`hms` returns the hour, minute, and second numbers in separate arrays.

Input Arguments

`t` — Input dates and times
datetime or duration array

Input dates and times, specified as a datetime or duration array.

Data Types: `datetime` | `duration`

Output Arguments

`h` — Hour numbers
numeric array

Hour numbers, returned as a numeric array of integer values from 0 to 23. The `h` output is of type `double` and is the same size as `t`.

`m` — Minute numbers
numeric array

Minute numbers, returned as a numeric array of integer values from 0 to 59. The `m` output is of type `double` and is the same size as `t`.

`s` — Second numbers
numeric array

Second numbers, returned as a numeric array of values from 0 to less than 60, and can include a fractional part. For datetime values whose time zone is `UTCLeapSeconds`, the `s`

output can contain a value between 60 and 61 for times that fall during a leap second occurrence. The `s` output is of type `double` and is the same size as `t`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- In generated code, this function supports duration arrays only. For more information, see “Code Generation for Duration Arrays” (MATLAB Coder) and “Duration Array Limitations for Code Generation” (MATLAB Coder).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`hour` | `minute` | `second` | `ymd`

Introduced in R2014b

hold

Retain current plot when adding new plots

Syntax

```
hold on
hold off
hold all
hold

hold(ax, ___ )
```

Description

`hold on` retains plots in the current axes so that new plots added to the axes do not delete existing plots. New plots use the next colors and line styles based on the `ColorOrder` and `LineStyleOrder` properties of the axes. MATLAB adjusts axes limits, tick marks, and tick labels to display the full range of data. If axes do not exist, then the `hold` command creates them.

`hold off` sets the hold state to off so that new plots added to the axes clear existing plots and reset all axes properties. The next plot added to the axes uses the first color and line style based on the `ColorOrder` and `LineStyleOrder` properties of the axes. This option is the default behavior.

`hold all` is the same as `hold on`. This syntax will be removed in a future release. Use `hold on` instead.

`hold` toggles the hold state between on and off.

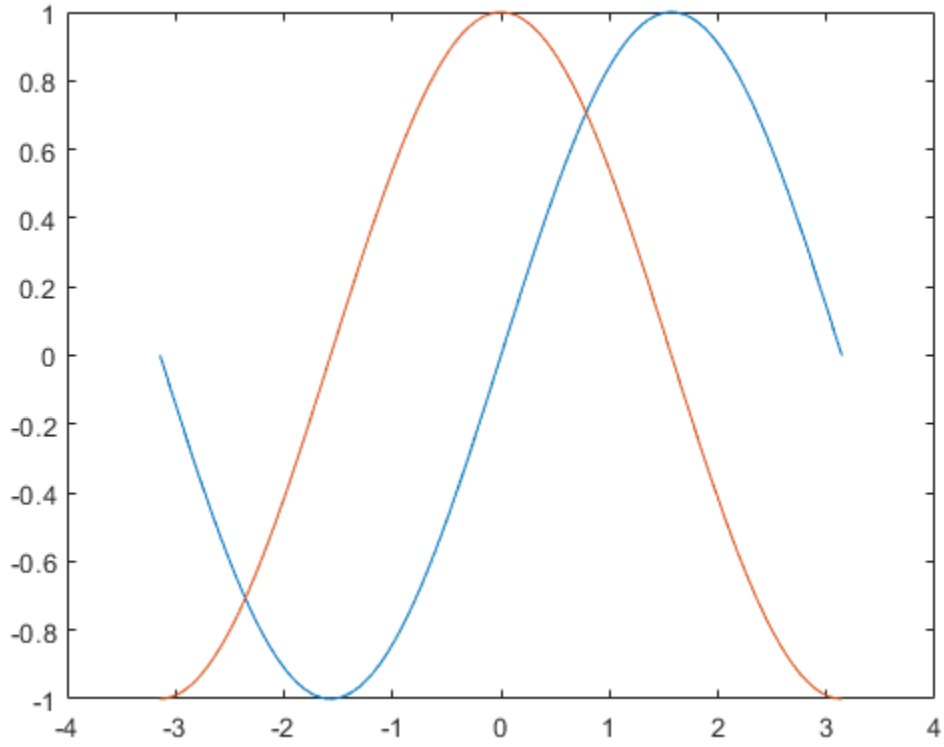
`hold(ax, ___ )` sets the hold state for the axes specified by `ax` instead of the current axes. Specify the axes as the first input argument for any of the previous syntaxes. Use single quotes around the `'on'` and `'off'` inputs, such as `hold(ax,'on')`.

Examples

Add Line Plot to Existing Axes

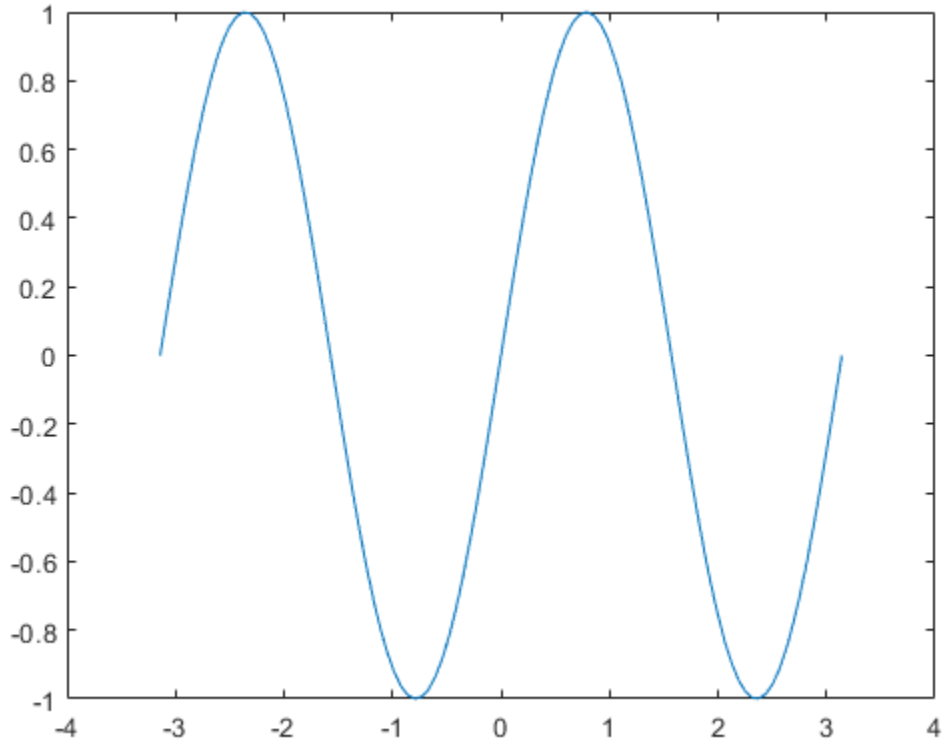
Create a line plot. Use `hold on` to add a second line plot without deleting the existing line plot. The new plot uses the next color and line style based on the `ColorOrder` and `LineStyleOrder` properties of the axes. Then reset the hold state to off.

```
x = linspace(-pi,pi);
y1 = sin(x);
plot(x,y1)

hold on
y2 = cos(x);
plot(x,y2)
hold off
```



When the hold state is off, new plots delete existing plots. New plots start from the beginning of the color order and line style order.

```
y3 = sin(2*x);
plot(x,y3)
```

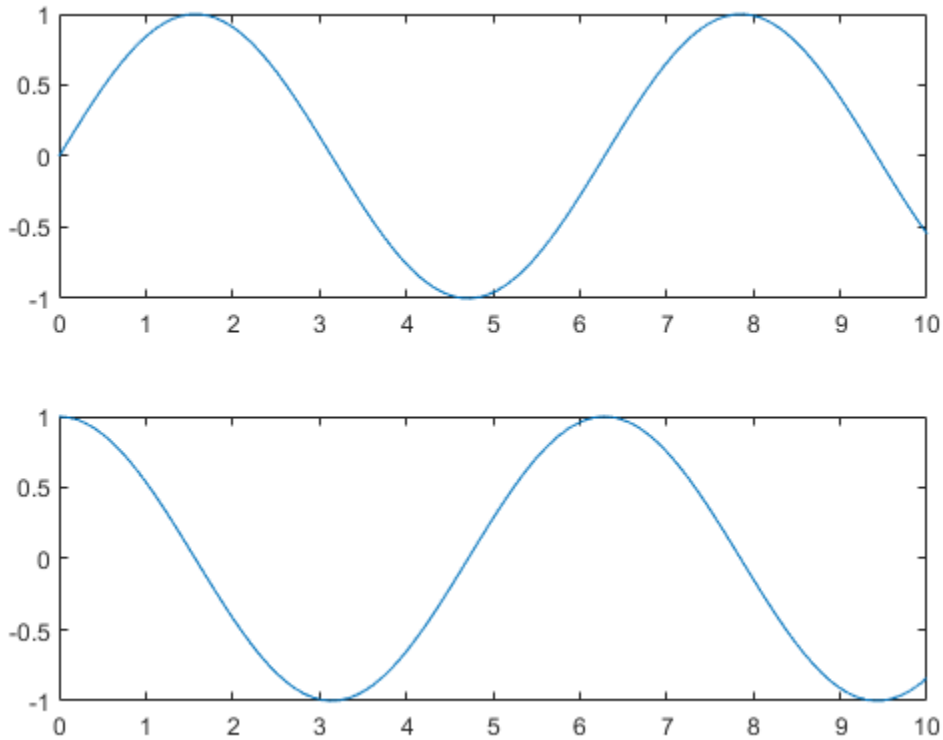


Specify Hold State for Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Plot a sine wave plot in each axes.

```
x = linspace(0,10);
y1 = sin(x);
y2 = cos(x);
tiledlayout(2,1)
```

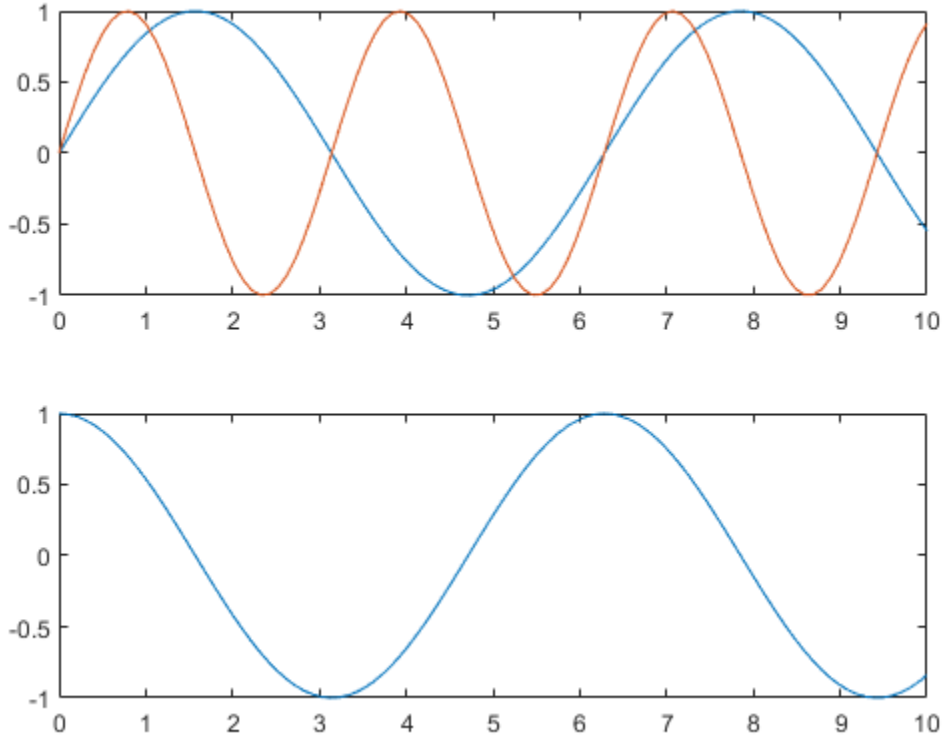
```
% Top plot
ax1 = nexttile;
plot(ax1,x,y1)

% Bottom plot
ax2 = nexttile;
plot(ax2,x,y2)
```



Add a second sine wave to the top axes.

```
hold(ax1,'on')
y3 = sin(2*x);
```

```
plot(ax1,x,y3)
hold(ax1,'off')
```



Input Arguments

ax — Target axes

`Axes` object | `PolarAxes` object | `GeographicAxes` object

Target axes, specified as an `Axes` object, a `PolarAxes` object, or a `GeographicAxes` object. If you do not specify the axes, then `hold` sets the hold state for the current axes.

Tips

- Use the `ishold` function to test the hold state.

Algorithms

The `hold` function sets these properties:

- `NextPlot` axes property - Sets this property of the associated `Axes`, `PolarAxes`, or `GeographicAxes` object to either `'add'` or `'replace'`.
- `NextPlot` figure property - Sets this property of the `Figure` object to `'add'`.

See Also

Functions
`axes` | `cla` | `figure` | `ishold` | `newplot` | `nexttile` | `tiledlayout`

Properties
Axes | PolarAxes

Topics
"Combine Multiple Plots"

Introduced before R2006a

home

Send cursor home

Syntax

```
home
```

Description

`home` moves the cursor to the upper-left corner of the Command Window. `home` also scrolls all visible text in the Command Window out of view, giving the appearance of clearing the screen without deleting any text. After running the `home` command, you can use the scroll bar to view the previously displayed text.

Use `home` in a MATLAB code file to always display output in the same starting position on the screen without clearing the Command Window.

Examples

Move Cursor to Home Position

Use the `magic` function to create and display a 5-by-5 integer matrix in the Command Window.

```
magic(5)
```

```
ans = 5×5

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

Run the `home` function to scroll the displayed matrix and any other visible text out of view and to move the cursor to the upper-left corner of the Command Window.

```
home
```

Use the Command Window scroll bar to scroll back up and see the hidden text.

See Also

`clc`

Introduced before R2006a

horzcat

Concatenate arrays horizontally

Syntax

```
C = horzcat(A,B)
C = horzcat(A1,A2,…,An)
```

Description

`C = horzcat(A,B)` concatenates `B` horizontally to the end of `A` when `A` and `B` have compatible sizes (the lengths of the dimensions match except in the second dimension).

`C = horzcat(A1,A2,…,An)` concatenates `A1`, `A2`, … , `An` horizontally.

`horzcat` is equivalent to using square brackets for horizontally concatenating arrays. For example, `[A,B]` or `[A B]` is equal to `horzcat(A,B)` when `A` and `B` are compatible arrays.

Examples

Two Matrices

Create two matrices and concatenate them horizontally, first by using square bracket notation, and then by using `horzcat`.

```
A = [1 2; 3 4]
```

```
A = 2×2

     1     2
     3     4
```

```
B = [4 5 6; 7 8 9]
```

```
B = 2×3

     4     5     6
     7     8     9
```

```
C = [A,B]
```

```
C = 2×5

     1     2     4     5     6
     3     4     7     8     9
```

```
D = horzcat(A,B)
```

```
D = 2×5

     1     2     4     5     6
     3     4     7     8     9
```

Two Tables

Create a table A with three rows and two variables.

```
A = table([5;6;5],['M';'M';'M'],...
    'VariableNames',{'Age' 'Gender'},...
    'RowNames',{'Thomas' 'Gordon' 'Percy'})
```

```
A=3×2 table
              Age    Gender
              ___    ______

    Thomas     5       M
    Gordon     6       M
    Percy      5       M
```

Create a table B with three rows and three variables.

```
B = table([45;41;40],[45;32;34],{'NY';'CA';'MA'},...
    'VariableNames',{'Height' 'Weight' 'Birthplace'},...
    'RowNames',{'Percy' 'Gordon' 'Thomas'})
```

```
B=3×3 table
              Height    Weight    Birthplace
              ______    ______    __________

    Percy       45        45        {'NY'}
    Gordon      41        32        {'CA'}
    Thomas      40        34        {'MA'}
```

Horizontally concatenate A and B. The order of rows in C matches the order in A.

```
C = horzcat(A,B)
```

```
C=3×5 table
              Age    Gender    Height    Weight    Birthplace
              ___    ______    ______    ______    __________

    Thomas     5       M         40        34        {'MA'}
    Gordon     6       M         41        32        {'CA'}
    Percy      5       M         45        45        {'NY'}
```

Dates with Different Types

Concatenate a date character vector, a string date, and a datetime into a single row of dates. The result is a datetime row vector.

```
chardate = '2016-03-24';
strdate = "2016-04-19";
t = datetime('2016-05-10','InputFormat','yyyy-MM-dd');
C = horzcat(chardate,strdate,t)
```

```
C = 1x3 datetime array
   24-Mar-2016   19-Apr-2016   10-May-2016
```

String Array

Concatenate three string arrays into a single array.

```
A1 = ["str1"; "str2"];
A2 = ["str3"; "str4"];
A3 = ["str5"; "str6"];
C = horzcat(A1,A2,A3)
```

```
C = 2x3 string array
    "str1"    "str3"    "str5"
    "str2"    "str4"    "str6"
```

Matrices in a Cell Array

Create a cell array containing two matrices. Horizontally concatenate the matrices from the cell array into one matrix.

```
M1 = [1 2; 3 4];
M2 = [5 6 7; 8 9 10];
A1 = {M1,M2};
C = horzcat(A1{:})
```

```
C = 2×5

     1     2     5     6     7
     3     4     8     9    10
```

Input Arguments

A — First input
scalar | vector | matrix | multidimensional array | table | timetable

First input, specified as a scalar, vector, matrix, multidimensional array, table, or timetable.

B — Second input
scalar | vector | matrix | multidimensional array | table | timetable

Second input, specified as a scalar, vector, matrix, multidimensional array, table, or timetable.

- The elements of `B` are concatenated to the end of the first input along the second dimension. The sizes of the input arguments must be compatible. For example, if the first input is a matrix of size 3-by-2, then `B` must have 3 rows.
- All table inputs must have unique variable names. When present, row names must be identical, except for order.
- All timetable inputs must have the same row times and all columns must have different names.
- You can concatenate valid combinations of different types. For more information, see "Valid Combinations of Unlike Classes".

`A1,A2,…,An` — List of inputs
comma-separated list

List of inputs, specified as a comma-separated list of elements to concatenate in the order they are specified.

- The inputs must have compatible sizes. For example, if `A1` is a column vector of length *m*, then the remaining inputs must each have *m* rows to concatenate horizontally.
- All table inputs must have unique variable names. When present, row names must be identical, except for order.
- All timetable inputs must have the same row times and all columns must have different names.
- You can concatenate valid combinations of different types. For more information, see "Valid Combinations of Unlike Classes".

Tips

- To construct text by horizontally concatenating strings, character vectors, or cell arrays of character vectors, use the `strcat` function.
- To construct a single piece of delimited text from a cell array of character vectors or a string array, use the `strjoin` function.

Algorithms

For table inputs, `horzcat` concatenates by matching row names when present, or by matching table positions. `horzcat` assigns values for the `Description` and `UserData` properties of the output using the first nonempty values of the corresponding properties of the input.

When concatenating an empty array to a nonempty array, `horzcat` omits the empty array in the output. For example, `horzcat([1 2],[])` returns the row vector `[1 2]`.

If all input arguments are empty and have compatible sizes, then `horzcat` returns an empty array whose size is equal to the output size as when the inputs are nonempty. For example, `horzcat(zeros(0,1),zeros(0,2))` returns a 0-by-3 empty array. If the input sizes are not compatible, then `horzcat` returns a 0-by-0 empty array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cat` | `strcat` | `strjoin` | `vertcat`

Topics

“Creating, Concatenating, and Expanding Matrices”
“Valid Combinations of Unlike Classes”
“Combine Categorical Arrays”
“Concatenating Objects of Different Classes”
“Concatenation Methods”

Introduced before R2006a

horzcat

Horizontally concatenate `tscollection` objects

Syntax

```
tscout = horzcat(tsc1,tsc2,…,tscN)
```

Description

`tscout = horzcat(tsc1,tsc2,…,tscN)` horizontally concatenates the `tscollection` objects `tsc1`, `tsc2`,..., `tscN`, which contains the `timeseries` from the concatenated objects. Each `tscollection` in the list must have the same time vector.

Examples

Horizontal Concatenation

Create two `tscollection` objects and horizontally concatenate them.

```
ts1 = timeseries([1.1 2.9 3.7 4.0 3.0]',1:5,...
                 'Name','Acceleration');
ts2 = timeseries([3.2 4.2 6.2 8.5 1.1]',1:5,...
                 'Name','Speed');
tsc1 = tscollection(ts1);
tsc2 = tscollection(ts2);
tscout = horzcat(tsc1,tsc2)
```

```
Time Series Collection Object: unnamed

Time vector characteristics

      Start time            1 seconds
      End time              5 seconds

Member Time Series Objects:
```

```
Acceleration
Speed
```

Input Arguments

`tsc1,tsc2,…,tscN` — List of `tscollection` objects
comma-separated list

List of `tscollection` objects to concatenate, specified as a comma-separated list of scalar `tscollection` objects.

See Also

`timeseries` | `tscollection` | `vertcat`

Introduced before R2006a

hour

Hour number

Syntax

```
h = hour(t)
```

Description

`h = hour(t)` returns the hour numbers of the datetime values in `t`. The `h` output is a `double` array the same size as `t` and contains integer values from 0 to 23.

The `hour` function returns the hour numbers of datetime values. To assign hour values to a datetime array, `t`, use `t.Hour` and modify the `Hour` property.

Examples

Find Hour Number of Datetime Values

```
t = datetime('today'):hours(8):datetime('tomorrow');
t.Format = 'MMM dd, HH:mm'
```

```
t = 1x4 datetime array
   Aug 27, 00:00   Aug 27, 08:00   Aug 27, 16:00   Aug 28, 00:00
```

```
h = hour(t)
```

```
h = 1×4

     0     8    16     0
```

Input Arguments

t — Input date and time
`datetime` array

Input date and time, specified as a `datetime` array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`hms` | `minute` | `second` | `timeofday`

Introduced in R2014b

hours

Duration in hours

Syntax

```
H = hours(X)
```

Description

`H = hours(X)` returns an array of hours equivalent to the values in `X`.

- If `X` is a numeric array, then `H` is a `duration` array in units of hours.
- If `X` is a `duration` array, then `H` is a `double` array with each element equal to the number of hours in the corresponding element of `X`.

The `hours` function converts between `duration` and `double` values. To display a duration in units of hours, set its `Format` property to `'h'`.

Examples

Create Duration Array of Hours

```
X = magic(4);
H = hours(X)
```

```
H = 4x4 duration array
   16 hr    2 hr    3 hr   13 hr
    5 hr   11 hr   10 hr    8 hr
    9 hr    7 hr    6 hr   12 hr
    4 hr   14 hr   15 hr    1 hr
```

Convert Durations to Numeric Array of Hours

Create a `duration` array.

```
X = hours(2:10:38) + minutes(30)
```

```
X = 1x4 duration array
    2.5 hr   12.5 hr   22.5 hr   32.5 hr
```

Convert each duration in `X` to a number of hours.

```
H = hours(X)
```

```
H = 1×4

    2.5000   12.5000   22.5000   32.5000
```

View the data type of `H`

```
whos H
```

```
  Name      Size            Bytes  Class     Attributes

  H         1x4                32  double
```

Input Arguments

X — Input array
numeric array | duration array | logical array

Input array, specified as a numeric array, duration array, or logical array.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This function fully supports code generation. For more information, see “Code Generation for Duration Arrays” (MATLAB Coder).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`duration`

Topics

“Set Date and Time Display Format”

Introduced in R2014b

hot

Hot colormap array

Syntax

```
c = hot
c = hot(m)
```

Description

`c = hot` returns the hot colormap as a three-column array with the same number of rows as the colormap for the current figure. If no figure exists, then the number of rows is equal to the default length of 256. Each row in the array contains the red, green, and blue intensities for a specific color. The intensities are in the range [0,1], and the color scheme looks like this image.

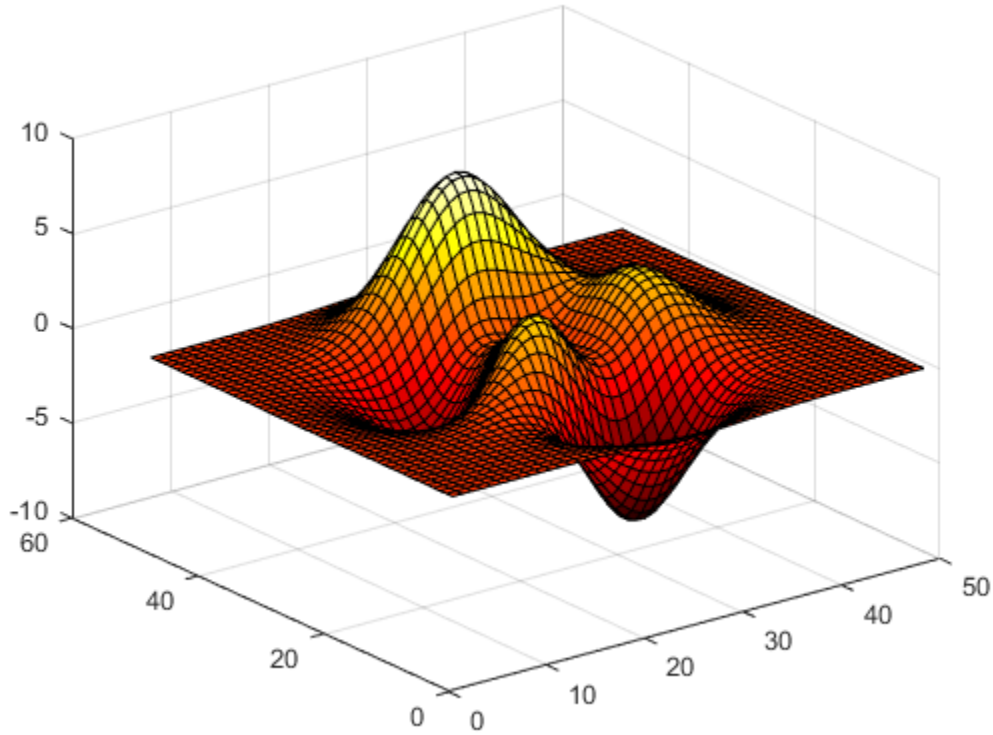
`c = hot(m)` returns the colormap with `m` colors.

Examples

Reverse the Colormap

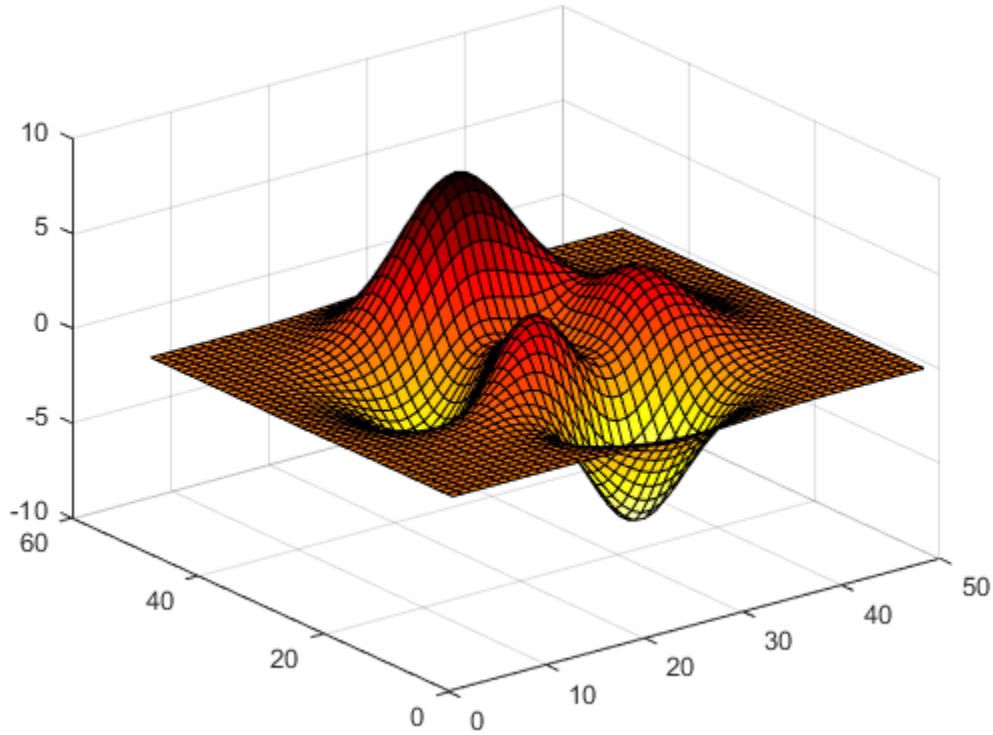
Plot a surface and assign the hot colormap.

```
surf(peaks);
colormap('hot');
```



Get the hot colormap array and reverse the order. Then apply the modified colormap to the surface.

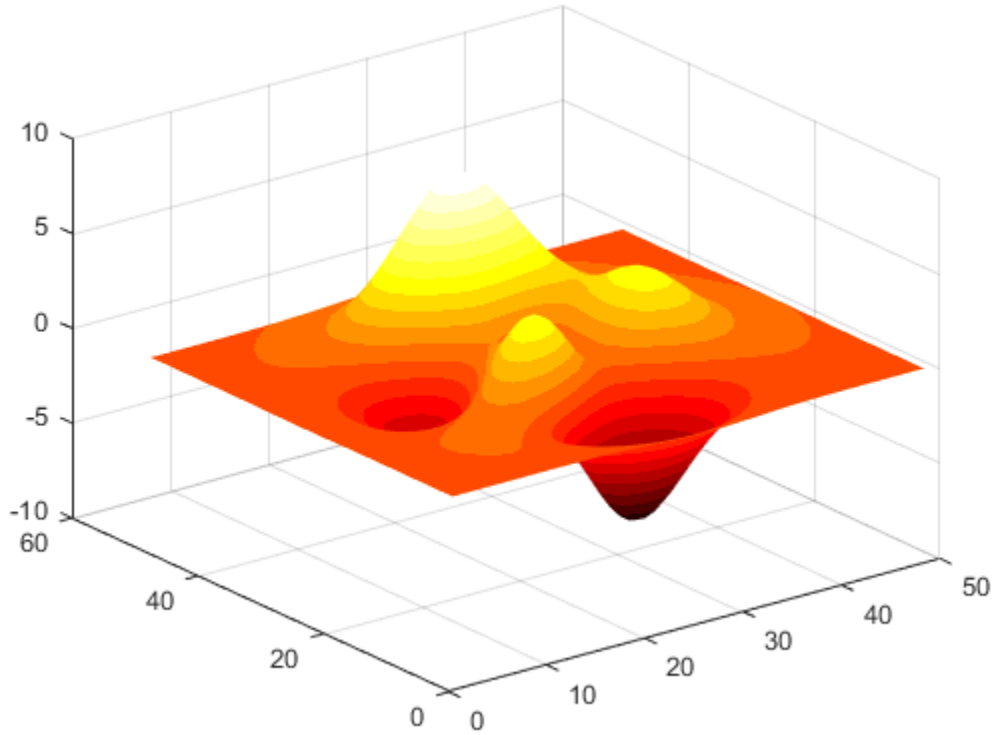
```
c = hot;
c = flipud(c);
colormap(c);
```



Downsample the Hot Colormap

Get a downsampled version of the hot colormap containing only twenty colors. Then display the contours of the peaks function by applying the colormap and interpolated shading.

```
c = hot(20);
surf(peaks);
colormap(c);
shading interp;
```



Input Arguments

m — Number of colors
256 (default) | scalar integer value

Number of colors, specified as a scalar integer value. The default value of `m` is equal to the length of the colormap for the current figure. If no figure exists, the default value is 256.

Data Types: `single` | `double`

Compatibility Considerations

Colormaps Have 256 Colors by Default

Behavior changed in R2019b

Starting in R2019b, colormaps have 256 colors by default.

In R2019a and previous releases, the default size is 64. If you have code that depends on a colormap having 64 colors, specify the number of colors when you set the colormap for the figure, axes, or chart. For example, `colormap(hot(64))` sets the figure's colormap to the 64-color `hot` colormap.

Alternatively, you can change the default colormap for all figures within your MATLAB session:

```
set(groot,'defaultFigureColormap',hot(64))
```

See Also

`colormap`

Topics

"Change Color Scheme Using a Colormap"

Introduced before R2006a

hsv

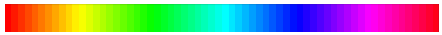
HSV colormap array

Syntax

```
c = hsv
c = hsv(m)
```

Description

`c = hsv` returns the hsv colormap as a three-column array with the same number of rows as the colormap for the current figure. If no figure exists, then the number of rows is equal to the default length of 256. Each row in the array contains the red, green, and blue intensities for a specific color. The intensities are in the range [0,1], and the color scheme looks like this image.



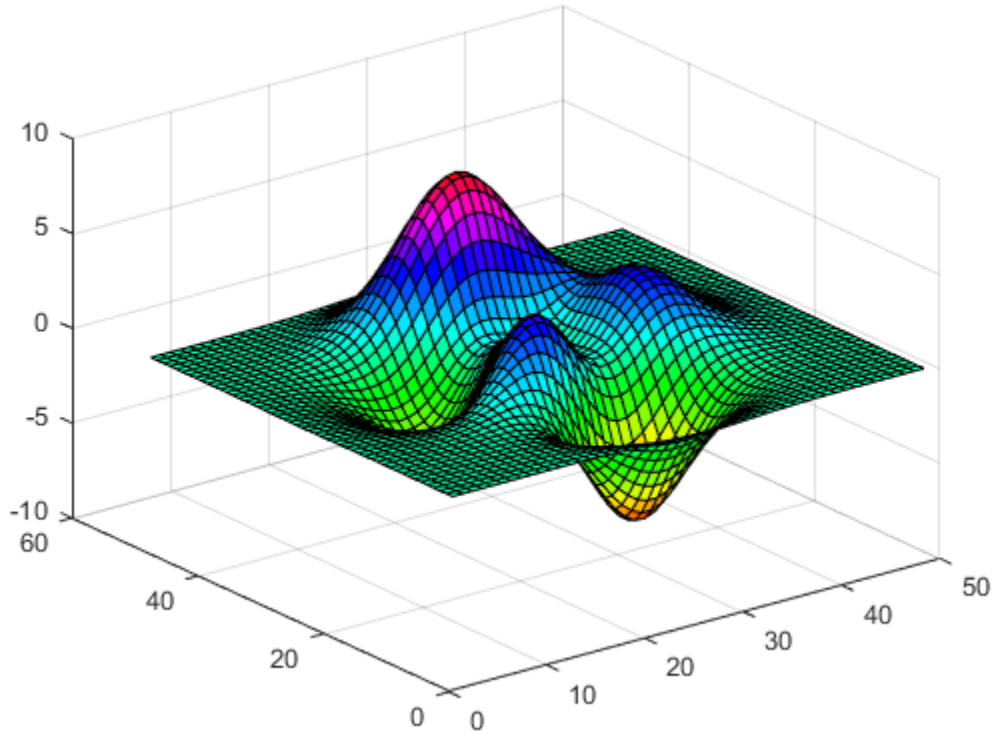
`c = hsv(m)` returns the colormap with `m` colors.

Examples

Reverse the Colormap

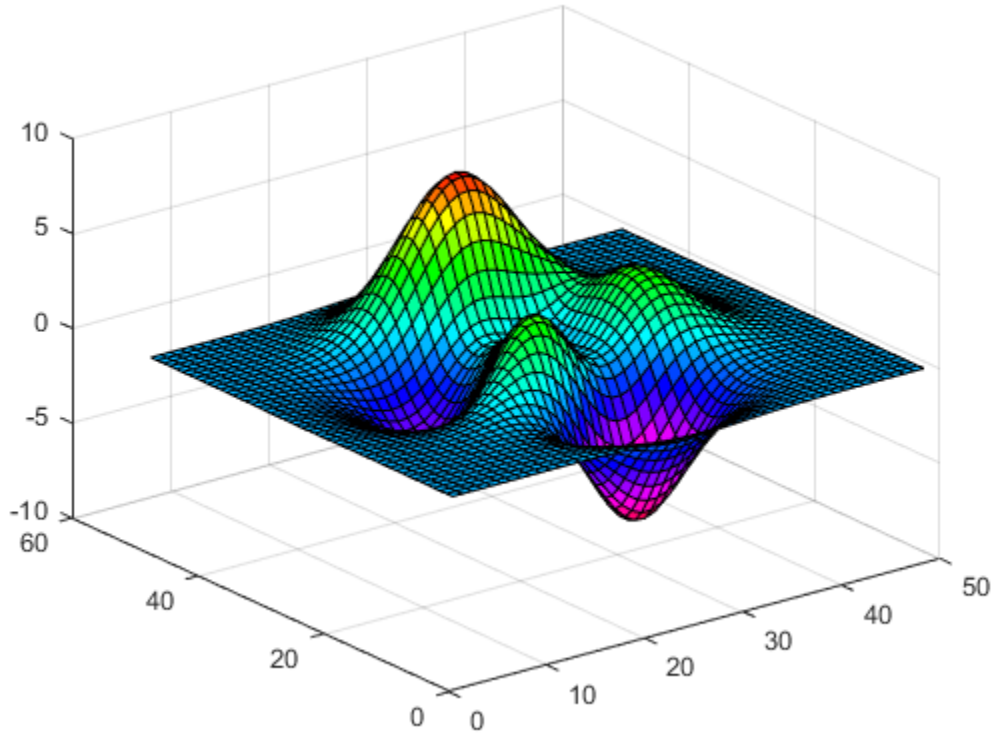
Plot a surface and assign the hsv colormap.

```
surf(peaks);
colormap('hsv');
```



Get the hsv colormap array and reverse the order. Then apply the modified colormap to the surface.

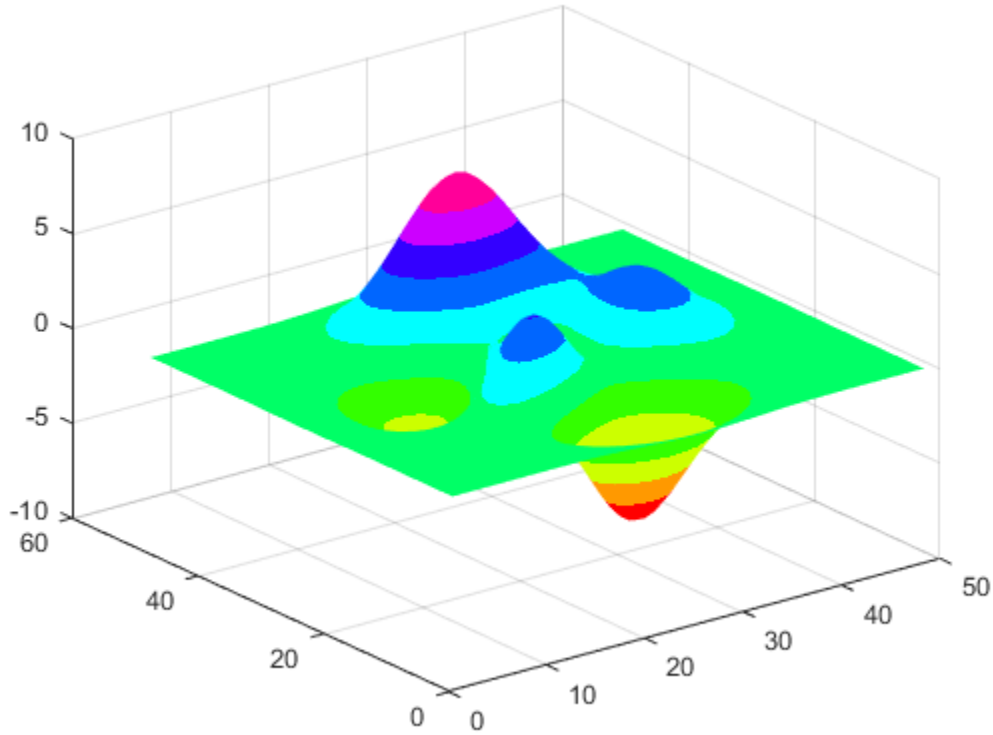
```
c = hsv;
c = flipud(c);
colormap(c);
```



Downsample the hsv Colormap

Get a downsampled version of the hsv colormap containing only ten colors. Then display the contours of the peaks function by applying the colormap and interpolated shading.

```
c = hsv(10);
surf(peaks);
colormap(c);
shading interp;
```



Input Arguments

`m` — Number of colors

`256` (default) | scalar integer value

Number of colors, specified as a scalar integer value. The default value of `m` is equal to the length of the colormap for the current figure. If no figure exists, the default value is 256.

Data Types: `single` | `double`

Compatibility Considerations

Colormaps Have 256 Colors by Default

Behavior changed in R2019b

Starting in R2019b, colormaps have 256 colors by default.

In R2019a and previous releases, the default size is 64. If you have code that depends on a colormap having 64 colors, specify the number of colors when you set the colormap for the figure, axes, or chart. For example, `colormap(hsv(64))` sets the figure's colormap to the 64-color `hsv` colormap.

Alternatively, you can change the default colormap for all figures within your MATLAB session:

```
set(groot,'defaultFigureColormap',hsv(64))
```

See Also

`colormap`

Topics

"Change Color Scheme Using a Colormap"

Introduced before R2006a

hsv2rgb

Convert HSV colors to RGB

Syntax

```
RGB = hsv2rgb(HSV)
rgbmap = hsv2rgb(hsvmap)
```

Description

`RGB = hsv2rgb(HSV)` converts the hue, saturation, and value (HSV) values of an HSV image to red, green, and blue values of an RGB image.

`rgbmap = hsv2rgb(hsvmap)` converts an HSV colormap to an RGB colormap.

Examples

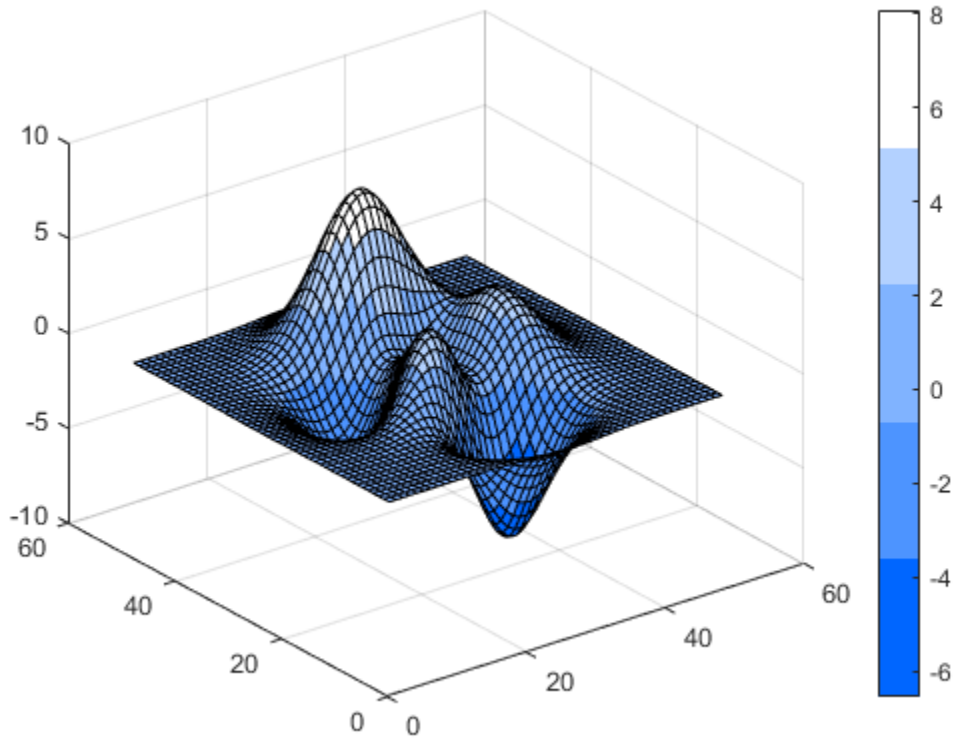
Convert HSV Matrix to a Colormap

Create a three-column HSV matrix that specifies five shades of blue. In this case, hue and value are constant, while saturation varies between `1.0` and `0.0`.

```
hsv = [.6 1 1; .6 .7 1; .6 .5 1; .6 .3 1; .6 0 1];
```

Convert the HSV matrix to a colormap by calling `hsv2rgb`. Then use that colormap in a surface plot.

```
rgb = hsv2rgb(hsv);
surf(peaks);
colormap(rgb);
colorbar
```



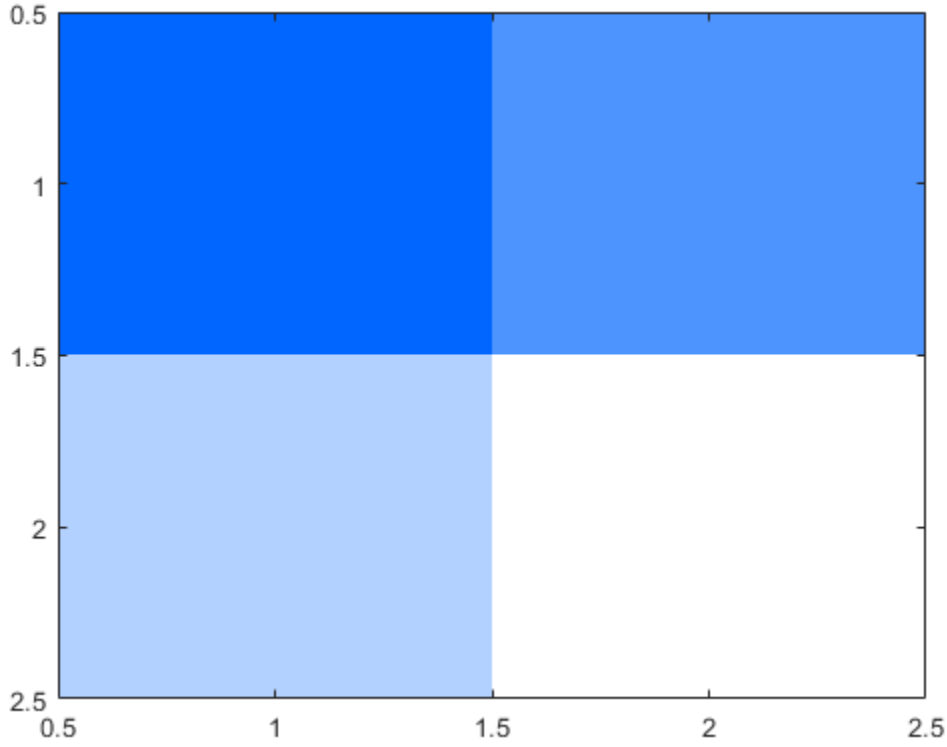
Convert 3-D HSV Array to a Truecolor Image

Create a 2-by2-by-3 HSV array that specifies four shades of blue.

```
hsv(:,:,1) = ones(2,2)*.6;
hsv(:,:,2) = [1 .7; .3 0];
hsv(:,:,3) = ones(2,2);
```

Convert the HSV array to a truecolor image using `hsv2rgb`. Then display the image.

```
rgb = hsv2rgb(hsv);
image(rgb);
```



Input Arguments

HSV — HSV image
m-by-*n*-by-3 numeric array

HSV image, specified as an *m*-by-*n*-by-3 numeric array with values in the range [0, 1]. The third dimension of HSV defines the hue, saturation, and value for each pixel, respectively, as described in the table.

Attribute	Description
Hue	Value from `0` to `1` that corresponds to the color's position on a color wheel. As hue increases from `0` to `1`, the color transitions from red to orange, yellow, green, cyan, blue, magenta, and finally back to red.
Saturation	Amount of hue or departure from neutral. `0` indicates a neutral shade, whereas `1` indicates maximum saturation.
Value	Maximum value among the red, green, and blue components of a specific color.

Data Types: `double` | `single` | `logical`

`hsvmap` — HSV colormap
c-by-3 numeric matrix

HSV colormap, specified as a *c*-by-3 numeric matrix with values in the range [0, 1]. Each row of `hsvmap` is a three-element HSV triplet that specifies the hue, saturation, and value components of a single color of the colormap.

Data Types: `double`

Output Arguments

`RGB` — RGB image
m-by-*n*-by-3 numeric array

RGB image, returned as an *m*-by-*n*-by-3 numeric array with values in the range [0, 1]. The third dimension of `RGB` defines the red, green, and blue intensity of each pixel, respectively. The image has the same data type as the HSV image, `HSV`.

Data Types: `double` | `single`

`rgbmap` — RGB colormap
c-by-3 numeric matrix

RGB colormap, returned as a *c*-by-3 numeric matrix with values in the range [0, 1]. Each row of `rgbmap` is a three-element RGB triplet that specifies the ref, green, and blue components of a single color of the colormap. The colormap has the same data type as the HSV colormap, `hsvmap`.

Data Types: `double` | `single`

References

[1] Smith, A. R. “Color Gamut Transform Pairs”. *SIGGRAPH 78 Conference Proceedings*. 1978, pp. 12–19.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`hsv` | `rgb2hsv`

Introduced before R2006a

HTML Properties

Control HTML UI component appearance and behavior

Description

HTML UI components allow you to display raw HTML text or embed HTML, JavaScript®, or CSS into your app and to interface with third-party JavaScript libraries. The `HTML` properties control the appearance and behavior of an HTML UI component. Use dot notation to refer to a specific object and property.

```
fig = uifigure;
h = uihtml(fig);
h.Position = [100 100 150 100];
h.HTMLSource = '<p style="font-family:sans-serif">This is <mark>marked</mark> text.</p>';
```

Properties

HTML

`HTMLSource` — HTML markup or file

`''` (default) | character vector | string scalar

HTML markup or file, specified as a character vector or string scalar that contains either HTML markup or the path to an HTML file. All HTML markup and files must be well formed. If the specified character vector or string scalar ends with `.html`, then it is assumed to be the path to an HTML file.

When embedding HTML markup, you do not need to specify `<html>` or `<body>` tags. If the markup you specify is supported in the Chromium browser used by MATLAB figures created with the `uifigure` function, then it will render. If you need more structure to your code, then consider using an HTML file instead.

If you set the `HTMLSource` property to the path to an HTML file, the HTML file must be located in a place that your local file system can access. If you are using supporting files, such as JavaScript, CSS, libraries, or images, then put the files in a location that your local file system can access, reference them in an HTML file, and set the `HTMLSource` property to the path to the HTML file. Supporting files cannot be in a relative directory adjacent to or above the HTML file.

Example: `h = uihtml('HTMLSource','CustomCharts.html')` specifies an HTML file.

Example: `h = uihtml('HTMLSource','<p>This is <span style="color:red">red</span> text.</p>')` specifies markup.

`Data` — MATLAB data
any MATLAB data type

MATLAB data, specified as any MATLAB data type. Use this argument when the value of `HTMLSource` is the path to an HTML file defining a JavaScript object. Then, this data can be synchronized between the MATLAB HTML UI component and the JavaScript object.

To synchronize the value of the `Data` property between MATLAB and the third-party content that you are embedding in your app, create a `setup` function in an HTML file that connects a JavaScript object called `htmlComponent` to the HTML UI component in MATLAB. Then, set the `HTMLSource` property value to the path to the file.

Changes in the value of the `Data` property trigger callback events in only one direction. Specifically,

- When the `Data` property is set in MATLAB, the `Data` property of the `htmlComponent` JavaScript object also updates and triggers JavaScript event listeners of the `'DataChanged'` event.
- When `htmlComponent.Data` is set in JavaScript, the `Data` property of the HTML UI component in MATLAB also updates and triggers the `DataChangedFcn` callback.

If you have data like nested cell arrays, arrays of structures, or MATLAB table array data, you might need more information about how data is converted between MATLAB and JavaScript. When the `Data` property of an HTML UI component is set in MATLAB, the data is converted using the `jsonencode` function, synchronized with JavaScript, parsed using JSON.parse(), and finally set to the `Data` property of the `htmlComponent` JavaScript object. Conversions happen in the opposite direction as well by way of JSON.stringify() and the `jsondecode` function. The `Data` property is the only property that synchronizes with JavaScript.

Refer to the `jsonencode` function for more information about the data types it supports. If the `jsonencode` function does not support your data type, then use a data type conversion function, like `num2str`, to convert your data to a supported type. Then, set it as the value of the `Data` property.

For more information about how to write an HTML file that enables data synchronization and the properties of the `htmlComponent` JavaScript object, see “Create HTML File That Can Trigger or Respond to Data Changes”.

Interactivity

`Visible` — Visibility of HTML UI component

`'on'` (default) | `'off'`

Visibility of HTML UI component, specified as `'on'` or `'off'`. The `Visible` property determines whether the HTML UI component is displayed on the screen. If the `Visible` property is set to `'off'`, the entire HTML UI component is hidden, but you can still set and access its properties.

To make your app open faster, set the `Visible` property to `'off'` for all components that do not need to appear at startup.

`Tooltip` — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size of HTML UI component

`[100 100 100 100]` (default) | `[left bottom width height]`

Location and size of HTML UI component relative to the parent container, specified as a four-element vector of the form `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the HTML UI component

Element	Description
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the HTML UI component
`width`	Distance between the right and left outer edges of the HTML UI component
`height`	Distance between the top and bottom outer edges of the HTML UI component

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

All measurements are in pixel units.

`InnerPosition` — Inner location and size of HTML UI component
`[100 100 100 100]` (default) | `[left bottom width height]`

Inner location and size of HTML UI component, specified as a four-element vector of the form `[left bottom width height]`. All measurements are in pixel units. This property value is identical to the `Position` property.

`OuterPosition` — Outer location and size of HTML UI component
`[100 100 100 100]` (default) | `[left bottom width height]`

This property is read-only.

Outer location and size of HTML UI component, returned as a four-element vector of the form `[left bottom width height]`. All measurements are in pixel units. This property value is identical to the `Position` property.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places an HTML UI component in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
h = uihtml(g);
h.HTMLSource = '<body style="background-color:powderblue;"><p>Hello World!</p></body>';
h.Layout.Row = 3;
h.Layout.Column = 2;
```

To make the HTML UI component span contiguous rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this HTML UI component spans columns 2 through 3.

```
h.Layout.Column = [2 3];
```

Callbacks

`DataChangedFcn` — Callback when data changes

`''` (default) | function handle | cell array | character vector

Callback when data changes, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

When the value of the `Data` property changes in the `htmlComponent` JavaScript object, the `Data` property of the MATLAB HTML UI component is updated, and this callback executes.

This callback function can access specific information about changes in data in the JavaScript object. MATLAB passes the information in a `DataChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the `DataChangedData` object properties using dot notation. For example, `event.PreviousData` returns the previous value of `Data` before it was last updated or changed. The `DataChangedData` object is not available to callback functions specified as character vectors.

This table lists the properties of the `DataChangedData` object.

Property	Description
`Data`	Most recent data resulting from the last `DataChanged` event
`PreviousData`	Data before the last `DataChanged` event
`Source`	MATLAB HTML UI component that executes the callback
`EventName`	`'DataChanged'`

For more information about writing callbacks, see “Write Callbacks in App Designer”.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback.

MATLAB resumes executing the running callback when the interrupting callback completes.

- If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.

- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.

HandleVisibility Value	Description
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uihtml'`

This property is read-only.

Type of graphics object, returned as `'uihtml'`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`jsondecode` | `jsonencode` | `uifigure` | `uihtml`

Topics
"Create HTML File That Can Trigger or Respond to Data Changes"

Introduced in R2019b

matlab.net.http Package

Summary of packages and classes in MATLAB HTTP interface

Description

Classes

matlab.net.http.AuthenticationScheme	HTTP Authentication scheme
AuthInfo	Authentication or authorization information in HTTP messages
Cookie	HTTP cookie received from server
CookieInfo	HTTP cookie information
Credentials	Credentials for authenticating HTTP requests
Disposition	Results in HTTP log record
HeaderField	Header field of HTTP message
HTTPException	Exception thrown by HTTP services
HTTPOptions	Options controlling HTTP message exchange
LogRecord	HTTP history log record
MediaType	Internet media type used in HTTP headers
Message	HTTP request or response message
MessageBody	Body of HTTP message
MessageType	HTTP message type
ProgressMonitor	Progress monitor for HTTP message exchange
ProtocolVersion	HTTP protocol version
RequestLine	First line of HTTP request message
RequestMessage	HTTP request message
RequestMethod	HTTP request method
ResponseMessage	HTTP response message
StartLine	First line of HTTP message
StatusClass	Status class of HTTP response
StatusCode	Status code in HTTP response
StatusLine	First line of HTTP response message

Packages

See Also

Introduced in R2016b

matlab.net.http.AuthenticationScheme class

Package: `matlab.net.http`

HTTP Authentication scheme

Description

The `AuthenticationScheme` enumeration class provides identifiers for supported authentication schemes. To specify a scheme, add a Scheme property to a `Credentials` object.

If you send a message to a server that requires authentication, then the server returns a `ResponseMessage` with a StatusCode of 401 or 407. The `AuthenticateField` in the response specifies the required `AuthenticationSchemes` for the request. Choose the strongest of the schemes that you can support, and reissue the request with an `AuthorizationField` containing the appropriate authorization information.

If MATLAB does not implement a scheme automatically and you want to use the scheme, then you must implement your own challenge responses.

Enumeration Member Name	Description
`Basic`	User name and password are transmitted in the header of an HTTP message. This scheme is implemented automatically when you supply the appropriate `Credentials` property in an `HTTPOptions` object when sending a message, and the server requests `Basic` authentication.
`Digest`	User is authenticated with a name and password, but more secure because the password is not transmitted over the connection. This scheme is implemented automatically when you supply the appropriate `Credentials` property in an `HTTPOptions` object when sending a message, and the server requests `Digest` authentication.
`Bearer`	Based on `OAuth`. MATLAB does not support this scheme.

Enumeration Member Name	Description
`HOBA`	Not support by MATLAB.
`Mutual`	Not support by MATLAB.
`Negotiate`	This scheme supports SPNEGO-based Kerberos and NTLM on Windows only. MATLAB implements this scheme automatically when you supply `Credentials` in `HTTPOptions` that name this scheme or which have an empty `Scheme`. A default `HTTPOptions` object contains such a `Credentials` object. To use this scheme, you must be properly logged into a Kerberos or NTLM environment. This scheme ignores the `Username` and `Password` properties of the `Credentials` object.
`NTLM`	MATLAB implements this scheme automatically when you supply `Credentials` in `HTTPOptions` that name this scheme or which have an empty `Scheme`. A default `HTTPOptions` object contains such a `Credentials` object. To use this scheme: • On Windows platforms, you must be properly logged into an NTLM environment, and the `Username` and `Password` properties of the `Credentials` object are ignored. • On Linux and macOS platforms, you must specify the `Username` and `Password` in the `Credentials` object.
`OAuth`	Not support by MATLAB.
`Token`	Not support by MATLAB.

Methods

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`string`	Authentication scheme as string
`char`	Authentication scheme as character vector

See Also

`AuthInfo` | `AuthenticateField` | `AuthorizationField` | `Credentials` | `HTTPOptions` | `ResponseMessage`

Topics

"Server Authentication"
"Proxy Server Authentication"

External Websites

RFC 2617 — HTTP Authentication: Basic and Digest Access Authentication

Introduced in R2016b

matlab.net.http.AuthInfo class

Package: `matlab.net.http`

Authentication or authorization information in HTTP messages

Description

MATLAB automatically handles authentication in request messages when you provide credentials in an `HTTPOptions` object. Use the `AuthInfo` class to examine authentication, to specify authorization information, or to implement an authentication protocol, such as `OAuth`, that is not handled automatically by the MATLAB HTTP interface.

The `AuthInfo` class represents one authentication challenge returned when:

- You call the `AuthenticateField.convert` method for a response message.
- You insert credentials in an `AuthorizationField` header field into a request message. When you store an `AuthInfo` object in an Authorization field, MATLAB automatically encloses values in quotes where required, and inserts escape characters as needed.
- A server returns `auth-info` data in an `AuthenticationInfoField` header field.

The `AuthenticateField` and `AuthorizationField` `convert` methods convert each `name=value` pair attribute in the field to parameter name-and-value pairs in the `Parameters` property. Escape characters and any quotes surrounding values are removed.

For more information, see RFC 7235 Authentication and RFC 2617 Host-Imp Interface (for Basic and Digest authentication).

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Description

`obj = matlab.net.http.AuthInfo(Scheme,paramName,paramValue)` creates an `AuthInfo` object that includes the Scheme property and optional `paramName`,`paramValue` parameters. You can specify several argument pairs in any order as `paramName1,paramValue1,...,paramNameN,paramValueN`.

`obj = matlab.net.http.AuthInfo(paramName,paramValue)` creates an `AuthInfo` object with an empty `Scheme` value.

`obj = matlab.net.http.AuthInfo(pStruct)` copies parameters and values from the fields of `pStruct`.

`obj = matlab.net.http.AuthInfo(pStr)` creates an `AuthInfo` from `pStr`.

Input Arguments

`paramName` — Parameter name
string | character vector

Parameter name, specified as a string or a character vector. The web service defines `paramName,paramValue` pairs that it accepts as part of a request. A `paramName` argument represents parameters that are converted to `name=value` parameters when you convert an `AuthInfo` object to a string.

Data Types: `char` | `string`

`paramValue` — Parameter value
type required by `paramName`

Parameter value, specified as a type required by `paramName`. There is no constraint on `paramValue` types, but values must support a `string` or `char` method.

`pStruct` — Parameters and values
MATLAB structure

Parameters and values, specified as fields in a MATLAB structure. The structure should have at least a `Scheme` field. MATLAB omits fields with empty values.

Data Types: `struct`

`pStr` — Parameters and values
string

Parameters and values, specified as strings. For `pStr` syntax:

- If `pStr` is a credential or challenge, then see RFC 7235, section 2.1.
- If `pStr` is an `auth-info` in an Authentication-Info header, then see RFC 2617, Section 3.2.3.

`pStr` is one of these formats. The values are optionally enclosed in quotes and escape characters are inserted as needed.

```
SCHEME token
SCHEME param1=value1, param2=value2, ...
param1=value1, param2=value2, ...
```

Data Types: `string`

Properties

`Scheme` — Authentication scheme
`AuthenticationScheme.Basic` (default) | `matlab.net.http.AuthenticationScheme` object | string | empty

Authentication scheme, specified as a `matlab.net.http.AuthenticationScheme` object or a string naming the scheme.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Parameters` — Parameter names and values
n-by-2 cell array

Parameter names and values, specified as an n-by-2 cell array. `Parameters{i,1}` is the name of the `i`th parameter and is a string. `Parameters{i,2}` is its value. The type depends on the parameter. To get or set this property, use the `getParameter` or `setParameter` methods.

The `Parameters` property depends on the `Scheme` property and the attributes that follow `Scheme` in the header field. Parameter name matching is case-insensitive. Use the `addParameter`, `setParameter`, and `removeParameter` methods to access the values in the `Parameters` array.

Some parameter properties have special meanings and syntax, which MATLAB enforces based on the `Scheme`. Parameters with special meanings are listed here. All other parameters are `string` objects.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`
`Dependent`	`true`

Methods

Public Methods

getParameter	Return value of AuthInfo parameter
removeParameter	Remove AuthInfo parameter
setParameter	Set value of AuthInfo parameter

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`eq` and `isequal`	Two `AuthInfo` objects are equal if: • They have the same `Scheme` property and parameter names. Parameter name matching is case-insensitive. • The parameter values are equal using `isequal`, regardless of the order of the parameters. Both values must be scalars.

`string`	AuthInfo as string as it appears in an Authorization or Authenticate header field
`char`	AuthInfo as character vector

See Also

`AuthenticateField` | `AuthenticationInfoField` | `AuthenticationScheme` | `AuthorizationField` | `StatusCode` | `matlab.net.URI` | `matlab.net.base64encode`

External Websites

RFC 7235 Authentication
RFC 2617 Host-Imp Interface

Introduced in R2016b

getParameter

Class: `matlab.net.http.AuthInfo`
Package: `matlab.net.http`

Return value of AuthInfo parameter

Syntax

```
value = getParameter(authinfo,name)
```

Description

`value = getParameter(authinfo,name)` returns the value of parameter `name`. If the parameter does not exist, then `getParameter` returns empty (`[]`).

Input Arguments

`authinfo` — AuthInfo
`matlab.net.http.AuthInfo` object

AuthInfo, specified as a `matlab.net.http.AuthInfo` object. If `authinfo` is nonscalar, then `value` is a cell array of values with the same size and shape as `authinfo`.

`name` — Parameter name
string | character vector

Parameter name, specified as a string or a character vector. Name matching is case-insensitive. If `name` is `'Scheme'`, then `getParameter` returns the value of the `Scheme` property.

See Also

Introduced in R2016b

removeParameter

Class: `matlab.net.http.AuthInfo`
Package: `matlab.net.http`

Remove AuthInfo parameter

Syntax

```
newinfo = removeParameter(authinfo,name)
```

Description

`newinfo = removeParameter(authinfo,name)` removes the named parameter and returns the modified `AuthInfo` object. If the parameter does not exist, then `removeParameter` does nothing.

Input Arguments

`authinfo` — AuthInfo
`matlab.net.http.AuthInfo` object

AuthInfo, specified as a `matlab.net.http.AuthInfo` object.

`name` — Parameter
string | character vector

Parameter, specified as a string or a character vector. Name matching is case-insensitive.

See Also

Introduced in R2016b

setParameter

Class: `matlab.net.http.AuthInfo`
Package: `matlab.net.http`

Set value of AuthInfo parameter

Syntax

```
newinfo = setParameter(authinfo,name,value)
```

Description

`newinfo = setParameter(authinfo,name,value)` sets the `value` of the parameter `name` and returns the modified `AuthInfo` object. If a parameter does not exist, then the method adds it.

If you use this method to set the value of the `Scheme` property, the method changes the value of the property instead of adding a `Scheme` parameter. To set the `Scheme`, type `obj.Scheme = value`.

If you have a parameter named `Encoded`, then it must be the only parameter in this object. This usage is for an `AuthorizationField` object only.

Input Arguments

`authinfo` — AuthInfo
`matlab.net.http.AuthInfo` object

AuthInfo, specified as a `matlab.net.http.AuthInfo` object.

`name` — Parameter name
string | character vector

Parameter name, specified as a string or a character vector.

Name matching is not case-sensitive, but `value` is case-sensitive. If `name` matches an existing parameter, but with a different case, then the parameter name is changed to the case of the specified `name`.

`value` — Parameter value
string | character vector | any type that has string or char method | type supported by `name`

Parameter value, specified a string, character vector, any type that has a string or char method, or a type supported by the `name`.

Do not enclose a value with double-quotes or insert any escape characters. When you insert this `AuthInfo` object in an `AuthenticateField`, `AuthenticationInfoField` or `AuthorizationField` object, then the value is converted to a string and quotes and escape characters are automatically inserted as needed. If `value` is an array of strings or a cell array of character vectors, then its members are separated by spaces.

Output Arguments

`newinfo` — Updated parameter value
type specified by parameter

Updated parameter value, returned as any type that was set for the parameter. If `value` is `[]`, then `setParameter` does not change the value of the `name` parameter. However, if `value` is an empty character vector or string, then the method sets `value` to an empty string.

See Also

`matlab.net.http.field.AuthenticateField` | `matlab.net.http.field.AuthenticationInfoField` | `matlab.net.http.field.AuthorizationField`

Introduced in R2016b

matlab.net.http.Cookie class

Package: `matlab.net.http`

HTTP cookie received from server

Description

The `Cookie` class represents an HTTP cookie. You obtain a `Cookie` object from a `SetCookieField` header in a `ResponseMessage` object. Then you send it back to the server in a `RequestMessage` `CookieField` header.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Description

`obj = matlab.net.http.Cookie(name,value)` creates a `Cookie` object with the specified `name` and `value` properties on page 1-6643. Use this constructor for testing.

Properties

`Name` — Cookie name
string

Cookie name, specified as a string.

Example: `"SESSIONID"`

Attributes:

`GetAccess`	`public`
`SetAccess`	`private`

`Value` — Cookie value
string

Cookie value, specified as a string. The value only has meaning to the server that sent the cookie.

Example: `"688412d8ed15f4a736dc6ab3"`

Attributes:

`GetAccess`	`public`
`SetAccess`	`private`

Methods

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`string`	Cookie as string as it appears in a header field
`char`	Cookie as character vector

Examples

Test for Expected Cookie Value

This example assumes a server returns specific values in the variable `response`. You cannot run this code in MATLAB. You can, however, use these coding patterns to test values from your web service responses.

Create a cookie object `test` containing values you expect from a server.

```
test = matlab.net.http.Cookie('cookie1','ID:33445566')
```

```
test = 

  Cookie with properties:

     Name: "cookie1"
    Value: "ID:33445566"
```

Send a message to a server; this example does not show this code. Assume that the server responds to your request message with this information in the `response` message.

```
setCookieFields = response.getFields('Set-Cookie')
```

```
setCookieFields = 
  SetCookieField with properties:

     Name: "Set-Cookie"
    Value: "cookie1=ID:33445566; Path=/"
```

Extract the cookie information.

```
cookieInfos = setCookieFields.convert;
```

Compare the cookie with your test cookie.

```
string(cookieInfos.Cookie) == string(test)
```

```
ans = 

   1
```

The cookies match.

See Also

`CookieField` | `RequestMessage` | `ResponseMessage` | `SetCookieField`

Topics

"Manage Cookies"

External Websites

RFC 6265 HTTP State Management Mechanism

Introduced in R2016b

matlab.net.http.CookieInfo class

Package: `matlab.net.http`

HTTP cookie information

Description

The `CookieInfo` class contains a `Cookie` object and information about the cookie that you can use for cookie management. To obtain a `CookieInfo` object, call the `convert` method of a `SetCookieField` object in a response message.

Unlike browsers, MATLAB does not provide an automatic cookie store. You must save cookies on your own and send them back to servers as needed.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Description

`obj = matlab.net.http.CookieInfo(Name,Value)` creates a `CookieInfo` object with additional properties specified by one, or more name-value pair arguments. `Name` is the property name and `Value` is the corresponding value. You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Unspecified properties are set to their default values.

Use this constructor for testing.

Properties

`Cookie` — Cookie
`matlab.net.http.Cookie` object

Cookie, specified as a `matlab.net.http.Cookie` object. Insert the Cookie into a request message `CookieField` property.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Expires` — Cookie expiration time
`datetime` | NaT

Cookie expiration time, specified as a `datetime` object. If there is an `Expires` attribute in the header field, then the `Expires` property is set. If the `Expires` attribute cannot be parsed, then `Expires` is NaT.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `datetime`

`MaxAge` — Duration of cookie
integer | NaN

Duration of cookie, specified as an integer or NaN. If there is a `Max-Age` attribute in the header field, then the `MaxAge` property is set. If the `Max-Age` attribute cannot be parsed, then `MaxAge` is NaN.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Domain` — Cookie domain
string

Cookie domain, specified as a string. If the Set-Cookie field does not specify a `Domain` attribute, then the `Domain` property is set to the request message `URI.Host` property

and `HostOnly` is set to `true`. `Domain` never begins with a period, as described in RFC 6265 — HTTP State Management Mechanism, Section 5.2.3.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `string`

`Path` — Path attribute in Set-Cookie field

string

Path attribute in a Set-Cookie field, specified as a string. The `Path` property is the value of the `Path` attribute in the Set-Cookie field. If the field does not contain a path attribute, then the `Path` property is set to `default-path` based on the request message `URI.Path` property. For a description of `default-path`, see RFC 6265 — HTTP State Management Mechanism, Section 5.1.4. Paths and Path-Match. `Path` always begins with a slash (`/`) character.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `string`

`Secure` — `Secure` attribute status

`false` (default) | `true`

`Secure` attribute status, where the `Secure` property is specified as `true` or `false`. The `Secure` property indicates that the Set-Cookie field contains a `Secure` attribute.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `logical`

`HttpOnly` — `HttpOnly` attribute status

`false` (default) | `true`

`HttpOnly` attribute status, where the `HttpOnly` property is specified as `true` or `false`. The `HttpOnly` property indicates that the Set-Cookie field contains an `HttpOnly` attribute.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `logical`

`Extensions` — Values of `extension-av` attributes
string

Values of `extension-av` attributes, specified as a string. The entire attribute is a single string, even if its syntax is of the form `name=value`.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `string`

`HostOnly` — `Domain` attribute status
`true` (default) | `false`

`Domain` attribute status, where the `HostOnly` property is specified as `true` or `false`. If the Set-Cookie field contains a `Domain` attribute, then `HostOnly` is true.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `logical`

`ExpirationTime` — Cookie expiration
`datetime`

Cookie expiration, specified as a `datetime` object.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `datetime`

`CreationTime` — Time cookie was created
`datetime`

Time the cookie was created, specified as a `datetime` object. The `SetCookieField.convert` method sets `CreationTime` to the current time.

If a new `CookieInfo` object matches an existing one, then the new object replaces the old one. However, the `CreationTime` property does not change. This behavior is specified in RFC 6265 — HTTP State Management Mechanism, Section 5.3 Storage Model, step 11.3. The `CookieInfo` objects match when the `Cookie.Name`, `CookieInfo.Domain`, and `CookieInfo.Path` (if specified) properties are equal.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `datetime`

Methods

Public Methods

matlab.net.http.CookieInfo.collectFromLog	Latest CookieInfo objects from HTTP response message history

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`string`	Cookie info as string as it appears in a `SetCookieField` header field. This string might not exactly match the string that appeared in the field from which this `CookieInfo` was created, but it has the same semantic meaning.
`char`	Cookie info as character vector

See Also

`SetCookieField` | `datetime` | `duration`

External Websites

RFC 6265 — HTTP State Management Mechanism, Section 5.3 Storage Model
RFC 6265 — HTTP State Management Mechanism, Section 5.2.3. The Domain Attribute
RFC 6265 — HTTP State Management Mechanism, Section 5.1.4 Default Base URI

Introduced in R2016b

matlab.net.http.CookieInfo.collectFromLog

Class: `matlab.net.http.CookieInfo`
Package: `matlab.net.http`

Latest CookieInfo objects from HTTP response message history

Syntax

```
infos = matlab.net.http.CookieInfo.collectFromLog(history)
```

Description

`infos = matlab.net.http.CookieInfo.collectFromLog(history)` returns `CookieInfo` objects for all the valid cookies found in the Set-Cookie header fields of the response message history. Use `collectFromLog` to obtain the latest cookies from a history of messages, such as those exchanged during a transaction involving authentication or redirection. If a server sends multiple versions of the same cookie, then `collectFromLog` returns only the most recent. This method also eliminates cookies that might not be valid for the URI of the server, that is, whose Domain is inconsistent with the request URI.

Input Arguments

`history` — Log records
vector of `matlab.net.http.LogRecord` objects

Log records, specified as a vector of `matlab.net.http.LogRecord` objects. The `RequestMessage.send` method returns a `LogRecord` as an optional argument.

Attributes

Sealed	true

Static	true

Examples

Reduce Message Exchanges

Eliminate message redirection by reusing cookies.

Send a message to a fictional website using redirection. Multiple messages are exchanged. To execute this code, you must provide a valid URI.

```
import matlab.net.http.*
import matlab.net.http.field.*
r = RequestMessage;
[resp,~,history] = r.send('http://www.somewebsite.com');
disp(length(history))
```

```
4
```

Extract the cookies from the message history.

```
cookieInfos = CookieInfo.collectFromLog(history);
if ~isempty(cookieInfos)
    cookies = [cookieInfos.Cookie];
end
```

Apply the cookies to the next request. Only one message is exchanged.

```
r = RequestMessage([],CookieField(cookies));
[resp,~,history] = r.send('http://www.somewebsite.com');
disp(length(history))
```

```
1
```

See Also

matlab.net.http.LogRecord | matlab.net.http.ResponseMessage | matlab.net.http.field.CookieField | send

Introduced in R2016b

matlab.net.http.Credentials class

Package: `matlab.net.http`
Superclasses: `handle`

Credentials for authenticating HTTP requests

Description

The `Credentials` class specifies authentication credentials for sending a request message. Specify a `Credentials` object in an `HTTPOptions` object.

The `RequestMessage.send` method uses credentials to respond to authentication challenges from servers or proxies. The authentication challenge is in an `AuthenticateField` header field and specifies one or more authentication schemes that the server or proxy accepts to satisfy the request.

The behavior depends on the authentication scheme. In general, MATLAB searches the vector of `Credentials` objects for one that applies to the request URI and which supports the specified authentication scheme. MATLAB then resends the original request with the appropriate credentials in an `AuthorizationField` header. If multiple credentials apply, then MATLAB uses the most specific `Credentials` object for the strongest scheme. If duplicate `Credentials` objects exist, then MATLAB uses the first one.

To see what authentication schemes MATLAB implements automatically, see `AuthenticationScheme`. If the server requires other schemes, or you do not supply credentials for the required scheme, then the authentication response message returns a StatusCode of 401 or 407. In this case, you must implement the appropriate response yourself.

For schemes that do not require `Username` and `Password` properties, such as NTLM on Windows, you can use a default `Credentials` object. A default `Credentials` object has default or empty properties and applies to all supported schemes and URIs. Your authorization credentials come from information, such as Kerberos tickets, that are stored in your system when you log in. To constrain the conditions under which a particular scheme is used, you can specify additional properties, such as Scope and Realm. For example, set these values to use NTLM for some URLs and Kerberos for others and reject authentication requests from servers that do not match those URLs and schemes.

For schemes that require `Username` and `Password` properties, if MATLAB authenticates with a `Credentials` object, then MATLAB saves the results in this object. MATLAB can apply these credentials on subsequent requests without waiting for an authentication challenge from the server. To take advantage of shortcut, provide the same `Credentials` object on subsequent requests in the same or other `HTTPOptions` objects.

`Credentials` objects are handle objects and internally accumulate information about prior successful authentications. Therefore, you can reuse the information for subsequent messages. If you insert this object into multiple `HTTPOptions` objects, then the `Credentials` object might be updated upon each use. If you copy `Credentials` using its `copy` method, then MATLAB copies only the visible properties that you set, and not the internal state.

The `matlab.net.http.Credentials` class is a `handle` class.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Description

`obj = matlab.net.http.Credentials(Name,Value)` creates HTTP credentials with additional properties specified by one, or more name-value pair arguments. `Name` is the property name and `Value` is the corresponding value. You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Unspecified properties are set to their default values.

If you call this constructor with no arguments, then the `Credentials` object applies to and thus permits authentication for all URIs and all authentication schemes. However, the object works only for schemes that do not require a user name or password. For example, on Windows, the object enables NTLM and Kerberos authentication using the credentials of the logged-in user.

Properties

Scheme — Authentication schemes for credentials

empty (default) | `AuthenticationScheme.Basic` | `AuthenticationScheme.Digest` | vector of `matlab.net.http.AuthenticationScheme` objects | ...

Authentication schemes for credentials, specified as a vector of `matlab.net.http.AuthenticationScheme` objects. For a complete list of supported schemes per platform, see `AuthenticationScheme`.

If `Scheme` is empty (default), then the credentials apply to all defined authentication schemes.

For more information about authentication, see “Server Authentication” and “Proxy Server Authentication”.

If `Scheme` is set to `Basic` only, then these credentials might be applied to a request whether or not the server requests authentication. A `Basic`-only scheme avoids an extra round trip responding to an authentication challenge. However, if the server does not require `Basic` authentication, this scheme unnecessarily exposes the `Username` and `Password` properties to the server.

If this property includes any scheme besides or in addition to `Basic`, or if `Scheme` is empty (which allow all schemes), then MATLAB does not send any authorization information in the first message to which these credentials apply. MATLAB waits for the server to respond with a challenge, such as a WWW-Authenticate field. The response tells MATLAB what authentication schemes the server accepts. Then MATLAB chooses the strongest scheme listed among the `Credentials`, which also match the `Scope` and `Realm` properties you specify, if any. Subsequent messages to the same server with the same `Scope` include the appropriate authorization information without first requiring a challenge.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Scope — URIs to which credentials apply

empty (default) | vector of `matlab.net.URI` objects | string | character vector

URIs to which credentials apply, specified as a vector of `matlab.net.URI` objects or strings or character vectors. Strings must be acceptable to the `URI` constructor or of the form `host/path/...`.

An empty `Scope` value, or an empty `Host` or `Path` in this vector matches all `Host` or `Path` properties. Do not leave `Scope` empty if `Scheme` is set to `Basic` only, unless you only access trusted servers. This combination of settings sends `Username` and `Password` to any server you access using the `HTTPOptions` containing these `Credentials`.

MATLAB compares the values in `Scope` with the request message URI to determine if these credentials apply. Credentials apply if the request URI refers to the same host at a path at or deeper than one of the URIs in this `Scope`. A `Scope` containing a URI naming a host with no path applies to all paths on that host.

For example, a `mathworks.com` host name in a `Scope` matches a request to `www.mathworks.com` and `anything.mathworks.com`. A `mathworks.com/products/stateflow` URI matches a request to `www.mathworks.com/products/stateflow/features` but not to `www.mathworks.com/products`. The `/products` path is not at or deeper than `/products/stateflow`.

Only the `Host`, `Port`, and `Path` properties of the `Scope` URIs are used. Typically you only specify a `Host` name, such as `www.mathworks.com`. If you know that the credentials are needed only for some paths within a host, then add a `Path` or a portion of a path.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Realm` — Authentication realms for credentials

empty (default) | string array | character vector | cell array of character vectors

Authentication realms for credentials, specified as a string array, character vector, or cell array of character vectors containing regular expressions describing the realms for the credentials. The default value is empty, which matches all realms. If any value in this vector is an empty string, then it only matches an empty or unspecified `Realm`. To anchor the regular expression to the start or end of the authentication `Realm` string, include the `^` or `$` characters as appropriate.

A `Realm` contains text to display so that the user knows what name and password to enter. The server specifies the `Realm` in an `AuthenticateField` of a `ResponseMessage`. In general, leave this property empty. Use a `Realm` when a server

requires different login values for different URIs and you want to specify programmatically different credentials for different realms on the same server. If you prompt for a name and password, do not set this property. Instead, display the `Realm` property from the `AuthenticateField` in your prompt so that the user knows which credentials to enter.

MATLAB compares the expressions in `Realm` against the authentication `Realm` in `AuthenticateField` to determine if these credentials apply. Once MATLAB carries out a successful authentication using one of these realms, MATLAB caches information about the authentication in `Credentials`. A subsequent request to a host and path that applies to these `Credentials` uses this cached information for authentication. This avoids the overhead of an authentication challenge or a call to the `GetCredentialsFcn` function.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Username` — User name for `Basic` or `Digest` authentication schemes

string | character vector | `[]`

User name for authentication schemes, specified as a string or a character vector. This property applies only to schemes that require an explicit `Username` and `Password`, and not for schemes that get your credentials from the system.

If you set the `Username` and `Password` properties to any string (including an empty one), then `Username` is used for authentication to any request for which these credentials apply, unless `GetCredentialsFcn` is specified. If you set this property to `[]`, then you must specify `GetCredentialsFcn` or authentication is not attempted.

If you do not want to embed a user name in your code, then leave this property empty and specify a `GetCredentialsFcn` that prompts the user for a name or obtains it from another source.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Password` — Password for `Basic` or `Digest` authentication schemes

string | character vector | `[]`

Password for authentication schemes, specified as a string or a character vector. This property applies only to schemes that require an explicit `Username` and `Password`, and not for schemes that get your credentials from the system.

If you set the `Username` and `Password` properties to any string (including an empty one), then `Password` is used for authentication to any request for which these credentials apply, unless `GetCredentialsFcn` is specified. If you set this property to `[]` there is no `GetCredentialsFcn` property, then no password is provided.

If you do not want to embed a password in your code, then leave this property empty and specify a `GetCredentialsFcn` that prompts the user for a password or obtains it from another source.

Attributes:

GetAccess	public
SetAccess	public

`GetCredentialsFcn` — Function returning `Username` and `Password`

function handle

Function returning the `Username` and `Password` for authentication, specified as a function handle. This property applies only to schemes requiring you to specify a user name and password.

MATLAB calls the `GetCredentialsFcn` function to obtain the name and password to use for the authentication response. MATLAB ignores the `Username` or `Password` properties in `Credentials`.

The function signature for `GetCredentialsFcn` is:

```
[username,password] = GetCredentialsFcn(cred,req,resp,authInfo,prevUsername,prevPasswd)
```

where the arguments are specified as:

By implementing the `GetCredentialsFcn` function and leaving the `Username` and/or `Password` properties in `Credentials` empty, you can implement a prompt to obtain these values from the user without embedding them in your program. In your prompt, display the request URI or the `authInfo.Realm` parameter. Another convenient pattern is to set the `Username` property and prompt only for the password. Your prompt can display the existing `Username`, or `prevUsername`, if set, and give the user the option to change it.

The `GetCredentialsFcn` function can examine the credentials in the `cred` argument and the header fields in the request and response messages to determine which resource is being accessed. Thus, the function can prompt the user for the correct credentials. In general, the prompt should display `authInfo.Realm` to let the user know the context of the authentication.

Since the `Credentials` is a handle class, the `GetCredentialsFcn` function stores the user name and password in the `cred` argument. You can use that object in future requests without calling the function again. MATLAB saves successful user names and passwords internally to apply them to future requests. However, MATLAB might not always be able to determine whether the same user name and password apply to different requests using these credentials.

If `GetCredentialsFcn` returns an empty array `[]` (not an empty string `''`) for the user name, then authentication should be denied and MATLAB returns the server authentication failure in the response message to the caller of `RequestMessage.send`. This behavior is appropriate if you implement a user prompt and the user clicks cancel in the prompt. If `prevUsername` and `prevPasswd` are identical to the name and password that you would return, then when you programmatically supply the name and password, you must return `[]`. This value indicates that your credentials are not accepted and you have no alternative choice. Otherwise, an infinite loop might occur calling your `GetCredentaislFcn` function repeatedly.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Data Types: `function_handle`

Examples

Create Credentials

Create credentials that are sent to only the appropriate server.

```
import matlab.net.http.Credentials
scope = URI('http://my.server.com');
creds = Credentials('Username','John','Password','secret','Scope',scope);
options = HTTPOptions('Credentials',creds);
```

Send the message. If the server requires authentication, then the transaction involves an exchange of several messages.

```
resp = RequestMessage().send(scope,options);
...
```

Next, reuse the options that contain the same credentials. Since the credentials already have been used successfully, this transaction requires a single message.

```
resp = RequestMessage().send(scope,options)
```

Prompt User for Name and Password

Create a function that prompts for credentials, using the `Username` property from the `Credentials` object as a default. MATLAB calls this function to obtain the name and password to use for the authentication response.

Create the `getMyCredentials` function.

```
function [u,p] = getMyCredentials(cred,req,resp,authInfo)
prompt = ["Username:" "Password:"];
defAns = [cred.Username ""];
title = "Credentials needed for " + getParameter(authInfo,'realm');
answer = inputdlg(prompt, title, [1, 60], defAns, 'on');
if isempty(answer)
    u = [];
    p = [];
else
    u = answer{1};
    p = answer{2};
end
end
```

Create the request message.

```
cred = matlab.net.http.Credentials('GetCredentialsFcn',@getMyCredentials);
options = matlab.net.http.HTTPOptions('Credentials',cred);
req = matlab.net.http.RequestMessage;
```

Send the message to `httpbin.org`.

```
uri = 'httpbin.org/basic-auth/user/passwd';
resp = req.send(uri,options)
```

Enter any text. To quit, select **Cancel**.

See Also

`AuthenticationScheme` | `HTTPOptions` | `RequestMessage` | `StatusCode`

Topics

"Server Authentication"

Introduced in R2016b

matlab.net.http.Disposition class

Package: `matlab.net.http`

Results in HTTP log record

Description

The `Disposition` enumeration class provides results in an HTTP log record.

Enumeration Member Name	Description
`ConversionError`	An error occurred converting the data of the response. The request was received, but there was an error trying to automatically convert the payload of the response. This condition indicates that the `MessageBody.Payload` property of `LogRecord.Response` contains the raw payload and the `Data` property is empty. The `LogRecord.Exception` property contains the exception. This error does not occur if a `ContentConsumer` was involved.
`Done`	A request and response were successfully sent and received. `Done` indicates the log record contains both the `RequestMessage` and `ResponseMessage`. It does not imply anything about the `StatusCode` in the response.
`Interrupt`	The user interrupted the operation, for example, pressing **Ctrl +C**. A `LogRecord` with this `Disposition` appears only if the operation was interrupted after transmission of the `RequestMessage` has begun. `LogRecord.Exception` is empty in this case. `LogRecord` might be partially populated, depending on when the interrupt occurred. If the interrupt occurred after receipt of a response header, then `LogRecord.Response` contains the header, and might also contain partial data processed during receipt of the payload, depending on the particular data converter or `ContentConsumer` that was being used.

Enumeration Member Name	Description
`TransmissionError`	An error occurred sending or receiving the message. The failure could be due to an I/O error such as a disconnect or timeout, a failure in a `ContentProvider` or `ContentConsumer`, or an error trying to convert the outgoing or incoming data from or to MATLAB data. The `LogRecord.Exception` property contains the exception that occurred. If the error occurred sending the request, `LogRecord.Request` contains the completed `RequestMessage` object and the `LogRecord.Response` property is empty. If the error occurred receiving the response, `LogRecord.Response` might be empty if complete headers were not received. If headers were received but the payload could not be read, then the `ResponseMessage` object contains the headers and might contain all or part of the payload that was received.

See Also

`LogRecord` | `MessageBody` | `RequestMessage` | `ResponseMessage` | `StatusCode`

Introduced in R2016b

matlab.net.http.HeaderField class

Package: `matlab.net.http`

Header field of HTTP message

Description

Use the `HeaderField` class to implement a header field for an HTTP message. The class provides conversions between strings in the header and MATLAB objects, arrays, and structures. Although you can set the `HeaderField` properties to arbitrary values, HTTP header fields have constraints on the allowed characters.

The `Name` property defines the header field type. MATLAB provides subclasses for commonly used fields in the `matlab.net.http.field` package. To see a list of supported subclasses, call the `HeaderField.displaySubclasses` method.

Creation

Description

`obj = matlab.net.http.HeaderField(name,value)` creates a header field with the `Name` property set to `name` and the `Value` property set to `value`. Either argument can be an empty double, `[]`. You can specify several argument pairs in any order as `name1,value1,...,nameN,valueN`. If the last `value` argument is missing, then `HeaderField` treats it as empty.

Properties

`Name` — Header field name
string | character vector

Header field name, specified as a string or character vector. `Name` determines the type of the field, which determines valid values for the `Value` property. If you set `Name` to `[]` or an empty string, then `Value` is `[]`.

If this object is an instance of a subclass implementing a specific header field type, then that class enforces constraints on the `Name` property.

Example: `'Content-Type'`

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Value` — Header field value
string | any type valid for `Name`

Header field value, specified as a string or any type valid for the `Name` property.

When you read this property, `Value` is a string representing the value in the field.

When you set this property, `Value` is any type acceptable to the field based on the `Name` property and/or the class of this object. The result is converted to a string. If a field type has a default value, set `Value` to an empty string (`''` or `string('')`). If you specify an empty double, `[]`, then the request message `send` and `complete` methods do not add this field to the message.

Example: `'text/html'`

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`
`Dependent`	`true`

Methods

Public Methods

convert	Convert header field value to MATLAB type
convertLike	Convert header field value like another header field
parse	Parse header field value and return as strings
addFields	Add fields to HeaderField array
removeFields	Remove fields from header field array
changeFields	Change existing values in HeaderField array
replaceFields	Change values in or add fields to array of HeaderFields

getFields	Return header fields matching name or class
eq	Compare two HeaderField arrays
displaySubclasses	Display supported HeaderField subclasses

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`isequal`	Return `true` if two header field arrays are the same size and the corresponding elements are equal as described by the `eq` method.
`string`	Array of header fields as a string, as it appears in a message. Newline characters are inserted between the fields but not at the end of all the fields.
`char`	Array of header fields as a character vector, as described by the `string` method

Examples

Choose Header Field Constructor

To create a Content-Type header field, use either the `HeaderField` class or the `ContentTypeField` class constructor.

When you use the `HeaderField` class constructor, you specify the `Name` property as `'Content-Type'`. However, if you misspell the field name, you might not find out about the error until the server rejects the message. Some servers silently ignore unknown field names.

```
f1 = matlab.net.http.HeaderField('Content-Type','text/plain');
```

Using the `ContentTypeField` class constructor is preferred because you cannot misspell the field name.

```
f2 = matlab.net.http.field.ContentTypeField('text/plain');
```

If the `Value` properties are the same, then the fields are equal, regardless of which constructor you use.

```
f1 == f2
```

```
ans =    1
```

Find Value in HTTP HeaderField Array

This example shows how to locate a specific header field `Cache-Control` in a response from mathworks.com.

Send a message to `mathworks.com`.

```
request = matlab.net.http.RequestMessage;
uri = matlab.net.URI('https://www.mathworks.com');
response = send(request,uri);
```

Search for `Cache-Control` and display the value.

```
field = response.getFields('Cache-Control');
value = field.Value
```

```
value 
"max-age=14400"
```

Tips

- The `HeaderField` constructor creates fields of class `HeaderField`. To create a field of a class defined in the `matlab.net.http.field` package, use the subclass constructor instead. For a list of subclasses, call the `HeaderField.displaySubclasses` method.

  For example, the `matlab.net.http.field.DateField` subclass creates a Date header field. If you use the `HeaderField` class to create a Date field, the `DateField` class interprets and enforces the value, even though `HeaderField` is not an instance of `DateField`. Likewise, if you convert the field value to a MATLAB `datetime` value, the `DateField.convert` method is used.
- If the `HeaderField` constructor rejects the `Name` and `Value` arguments, use the `GenericField` class constructor instead.

See Also

`matlab.net.http.field.GenericField`

Introduced in R2016b

convert

Class: `matlab.net.http.HeaderField`
Package: `matlab.net.http`

Convert header field value to MATLAB type

Syntax

```
value = convert(fields)
```

Description

`value = convert(fields)` converts the `Value` property of each element of `fields` to a MATLAB type.

For each header field in `fields`, MATLAB checks the `matlab.net.http.field` package for a class that supports a name with the same value as the `fields.Name` property. If the package contains a class for this field, then `convert` invokes the `convert` method of that class. To see the classes in the `field` package, call the `matlab.net.http.HeaderField.displaySubclasses` method.

If the conversion fails or if no class supports one of the `fields.Name` properties, then `convert` throws an exception.

This method does not work on heterogeneous arrays. All members of `fields` must be the same class.

Input Arguments

`fields` — Header fields
vector of class objects from the `matlab.net.http.field` package

Header fields, specified as a vector of class objects from the `matlab.net.http.field` package. All members of `fields` must be the same class.

Output Arguments

value — Header field values
vector of MATLAB types

Header field values, returned as a vector of MATLAB types. The types returned depend on the value of the specific class `convert` method.

Examples

Display datetime Value in Header Field

Send a message to the Hubble Heritage website and display the year an image was modified.

```
uri = matlab.net.URI('http://heritage.stsci.edu/2007/14/images/p0714aa.jpg');
req = matlab.net.http.RequestMessage;
r = send(req,uri);
d = convert(getFields(r,'Last-Modified'));
LastModified = d.Year
```

```
LastModified =

        2007
```

See Also

`matlab.net.http.HeaderField.displaySubclasses`

Introduced in R2016b

convertLike

Class: `matlab.net.http.HeaderField`
Package: `matlab.net.http`

Convert header field value like another header field

Syntax

```
value = convertLike(fields,other)
```

Description

`value = convertLike(fields,other)` converts the `Value` property of each element of `fields` to a MATLAB type, using conversion rules of the field `other`.

For each header field in `fields`, MATLAB checks the `matlab.net.http.field` package for a class that supports a name with the same value as the `fields.Name` property. If the package contains a class for this field, then `convert` invokes the `convert` method of that class. To see the classes in the `field` package, call the `matlab.net.http.HeaderField.displaySubclasses` method.

If the conversion fails or if no class supports one of the `fields.Name` properties, then `convertLike` throws an exception.

This method does not support heterogeneous arrays. All members of `fields` must be the same class.

Input Arguments

`fields` — Header fields
vector of `matlab.net.HeaderField` objects

Header fields, specified as a vector of `matlab.net.http.HeaderField` objects. All members of `fields` must be the same class.

`other` — Field to use for conversion rules
string | character vector | `matlab.net.http.HeaderField` object | `meta.class` identifying a `HeaderField` subclass

Field to use for conversion rules, specified as string, character vector, `HeaderField` object, or `meta.class` identifying a `HeaderField` subclass in the `matlab.net.http.field` package.

Output Arguments

`value` — Header field values
vector of MATLAB types

Header field values, returned as a vector of MATLAB types. The types returned depend on the value of the specific class `convert` method.

Examples

Get datetime Value from Created-Date Header Field

The `response` variable is a message that has a `Created-Date` header field. Its value is formatted like an `HTTPDateField` object, which returns a `datetime` object.

```
myField = response.getFields('Created-Date');
date = myField.convertLike(?matlab.net.http.field.HTTPDateField);
```

See Also
`convert` | `matlab.net.http.HeaderField.displaySubclasses`

Introduced in R2016b

matlab.net.http.HeaderField.displaySubclasses

Class: `matlab.net.http.HeaderField`
Package: `matlab.net.http`

Display supported HeaderField subclasses

Syntax

```
matlab.net.http.HeaderField.displaySubclasses
[fields,names] = matlab.net.http.HeaderField.displaySubclasses
```

Description

`matlab.net.http.HeaderField.displaySubclasses` displays all subclasses of the `HeaderField` class in the `matlab.net.http.field` package and the names of the header fields they support. Use these subclasses to construct common header fields.

`[fields,names] = matlab.net.http.HeaderField.displaySubclasses` returns the subclass names in `fields` as an array of strings. The method returns the header field names that the subclasses support in `names`, a cell array of string vectors that the subclasses support. `names{i}` contains the names supported by `fields(i)`. If `fields(i)` has no constraints on supported names, then `names{i}` is empty.

Attributes

Sealed	true
Static	true

Examples

Display Reserved Header Field Names

Display all header field names from classes in the `matlab.net.http.field` package. Use this command when you chose a name for an `IntegerField` or `URIReferenceField` header field object. The `Name` property for these objects cannot be one of the names in this list.

```
matlab.net.http.HeaderField.displaySubclasses
```

```
Available classes in matlab.net.http.field and names they support:
AcceptField              Accept
AuthenticateField        WWW-Authenticate, Proxy-Authenticate
AuthenticationInfoField  Authentication-Info, Proxy-Authentication-Info
AuthorizationField       Authorization, Proxy-Authorization
ConnectionField          Connection
ContentLengthField       Content-Length
ContentLocationField     Content-Location
ContentTypeField         Content-Type
CookieField              Cookie
DateField                Date
GenericField
HTTPDateField            Date, Expires, Retry-After, Accept-Datetime, Last-Modified, If-Modified-Since
HostField                Host
IntegerField
LocationField            Location
SetCookieField           Set-Cookie
URIReferenceField
```

Display Header Field Names Supported by HTTPDateField

Show the header fields that you can create using the `HTTPDateField` class.

```
[fields,names] = matlab.net.http.HeaderField.displaySubclasses;
for i = 1:numel(fields)
    if (strcmp(fields(i),'HTTPDateField'))
        disp(names{i})
        break
    end
end
```

```
  Columns 1 through 4

    "Date"    "Expires"    "Retry-After"    "Accept-Datetime"

  Columns 5 through 6

    "Last-Modified"    "If-Modified-Since"
```

See Also

Topics

Method Attributes

Introduced in R2016b

eq

Class: `matlab.net.http.HeaderField`
Package: `matlab.net.http`

Compare two HeaderField arrays

Syntax

```
field1 == field2
tf = eq(field1,field2)
```

Description

`field1 == field2` compares two `HeaderField` arrays element by element, returning an array of logical values indicating matching elements. The arrays must have the same dimensions, unless one is a scalar.

Two header fields are equal if they are functionally equivalent, even if they are not identical. Functionally equivalent means both of these conditions are true:

- `Name` properties match using a case-insensitive comparison
- `Value` properties match. If the `convert` method is supported for the `HeaderField` type, MATLAB uses the `isequal` method on the results of `convert`. If `convert` is not supported, comparisons are based on a case-sensitive match of the `Value` strings.

The == method compares the `Name` and `Value` properties only. The method ignores the actual classes of `field1` and `field2`, since both are instances of the `HeaderField` class.

`tf = eq(field1,field2)` is an alternative way to execute `field1 == field2`. The method returns a logical scalar or array.

Input Arguments

field1 — Header field
matlab.net.http.HeaderField object

Header field, specified as a `matlab.net.http.HeaderField` object or a vector of `HeaderField` objects.

field2 — Header field
matlab.net.http.HeaderField object

Header field, specified as a `matlab.net.http.HeaderField` object or a vector of `HeaderField` objects.

Examples

Different Header Field Classes Are Equivalent

This comparison evaluates to true, even though one object being compared is a `HeaderField` object and the other is a `DateField` object.

```
import matlab.net.http.HeaderField
import matlab.net.http.field.DateField
dt = datetime('now');
HeaderField('Date',dt) == DateField(dt)
```

```
ans = logical
   1
```

See Also

`convert` | `displaySubclasses`

Introduced in R2016b

parse

Class: `matlab.net.http.HeaderField`
Package: `matlab.net.http`

Parse header field value and return as strings

Syntax

```
value = parse(obj)
```

```
value = parse(obj,fields)
```

```
value = parse( ___ ,Name,Value)
```

Description

`value = parse(obj)` parses the `Value` property of the header field and returns strings. Use this method to process header fields for which there is no class in the `matlab.net.http.field` package. Use the `matlab.net.http.HeaderField.displaySubclasses` method to display classes in the package. For classes in the package, use the corresponding `convert` method to parse the value.

The parsing rules are based on sections 3.2.4-3.2.6 of RFC 7230 Message Syntax and Routing and are augmented to interpret multiple values.

`value = parse(obj,fields)` specifies the names to use for unnamed `struct` fields.

If the Nth field of a `struct` has no name, the corresponding Nth name in `fields` exists and is nonempty. It is used instead of `Arg_N`. Using this syntax forces the returned `value` to be a `struct` (or vector of `struct` objects) with at least as many fields as the length of `fields`. Typically this pattern occurs in header fields that begin with a token followed by attribute pairs.

`value = parse( ___ ,Name,Value)` specifies one or more delimiters. The default delimiters are commas and semicolons. You can use any of the input arguments in the previous syntaxes.

Input Arguments

`obj` — Header field
`matlab.net.http.HeaderField` object

Header field, specified as a `matlab.net.http.HeaderField` object or a vector of `HeaderField` objects.

`fields` — Names of `struct` fields
string vector | character vector | cell array of character vectors

Names of `struct` fields, specified as a string vector, a character vector, or a cell array of character vectors.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`ArrayDelimiters` — Delimiters separating array elements
string vector | character vector | cell vector of regular expressions | `''` | `[]`

Delimiters separating array elements, specified as:

- A string vector, character vector, or cell vector of regular expressions specifying the possible delimiters, interpreted in the order they appear in the vector.
- `''` — Do not parse `obj` as an array. MATLAB inserts quotes and escape characters.
- `[]` — Do not parse `obj` as an array. MATLAB does not insert quotes or escape characters into array elements.

`MemberDelimiters` — Delimiters separating structure fields
string vector | character vector | cell vector of regular expressions | `''` | `[]`

Delimiters separating structure fields, specified as:

- A string vector, character vector, or cell vector of regular expressions specifying the possible delimiters, interpreted in the order they appear in the vector.
- `''` — Do not parse `obj` as a `struct`. MATLAB inserts quotes and escape characters.

- `[]` — Do not parse `obj` as a `struct`. MATLAB does not insert quotes or escape characters into `struct` values.

Output Arguments

`value` — Header field `Value` property
string vector | `struct` array | cell array of `struct`

Header field `Value` property, returned as a string vector, a `struct` array, or a cell array of `struct` values.

MATLAB parses the `Value` property as a list of comma-separated strings. Each string becomes an element of the `value` vector. An element is one of the following:

- `struct` of `name=value` pairs
- `struct` of semicolon-separated values
- string, if the field does not contain a semicolon or an equal sign or does not appear to be a structure.

`parse` converts the name of each `struct` field to a valid MATLAB identifier using `matlab.lang.makeValidName`. For the following `Value` property, `parse` creates field name `x_p1` from `_p1`.

To resolve duplicate names, `parse` calls `matlab.lang.makeUniqueStrings`. For the following `Value` property, `parse` creates field name `p11` from duplicate field name `p1`.

If a `struct` field contains only a `Value`, but not a `name=value` pair, then the field name is `Arg_N`. The `N` is the ordinal position of the field in the `struct`. For the following `Value` property, `parse` creates field name `Arg_2` for the missing name.

Value Property	Output Argument	Description
`"p1=first p2=second"`	`p1: "first"` `p2: "second"`	`parse` returns a `struct` for `name=value` pairs.
`"first;second"`	`Arg_1: "first"` `Arg_2: "second"`	`parse` returns a `struct` and assigns default field names for semicolon-separated values.

Value Property	Output Argument	Description
"first second"	"first second"	parse returns a string if the field does not contain a semicolon or an equal sign or does not appear to be a structure.
"_p1=first p2=second"	x_p1: "first" p2: "second"	parse converts invalid field name _p1 to x_p1.
"p1=first p1=second"	p1: "first" p11: "second"	parse converts duplicate field name p1 to p11.
"p1=first; second"	p1: "first" Arg_2: "second"	parse creates field name Arg_2 for the missing name for Value second.
"p1=first; p3=(a comment here)"	p1: "first" p3: "(a comment here)"	parse retains comments.

If obj is a vector of header fields, then the parse method concatenates the results of parsing each of the fields into a single array. If the values are not of the same type, then value is a cell array.

Value Property of Header Field Vector	Element of Cell Array Output Argument
"p1=first p2=second"	x{1} = p1: "first" p2: "second"
"third"	x{2} = third

Examples

Change Default Field Name

Assume that you receive a header field H in a response message from a server with the Value property media-type; name1=value1; name2=value2. To run this example, create the variable H.

```
H = matlab.net.http.HeaderField('Test-Name','media-type; name1=value1; name2=value2')
```

```
H = 

  HeaderField with properties:

     Name: "Test-Name"
    Value: "media-type; name1=value1; name2=value2"
```

Parse the `Value` property of `H`. MATLAB creates a default field name `Arg_1`.

```
var = parse(H)
```

```
var = 

  struct with fields:

    Arg_1: "media-type"
    name1: "value1"
    name2: "value2"
```

Change the default to a more meaningful name `MediaType`.

```
var = parse(H,'MediaType')
```

```
var = 

  struct with fields:

    MediaType: "media-type"
        name1: "value1"
        name2: "value2"
```

Attributes

`Sealed`	`true`

To learn about attributes of methods, see Method Attributes.

See Also

`matlab.lang.makeUniqueStrings` | `matlab.lang.makeValidName`

External Websites

RFC 7230 Message Syntax and Routing

Introduced in R2016b

addFields

Class: `matlab.net.http.HeaderField`
Package: `matlab.net.http`

Add fields to HeaderField array

Syntax

```
fields = addFields(fields,newFields)
fields = addFields(msg,fName,fValue)
fields = addFields(fields,fName1,fValue1,...,fNameN,fValueN)
fields = addFields(fields,index, ___ )
```

Description

`fields = addFields(fields,newFields)` adds `newFields` to the end of the `HeaderField` array `fields` and returns the updated array. `addFields` does not check for duplicate fields.

`fields = addFields(msg,fName,fValue)` adds field with name `fName` and value `fValue`.

`fields = addFields(fields,fName1,fValue1,...,fNameN,fValueN)` adds fields specified by `fName`, `fValue` pair arguments, in the order specified.

`fields = addFields(fields,index, ___ )` inserts fields at `index` and can include any of the input arguments in previous syntaxes.

Input Arguments

`fields` — Header fields
`matlab.net.http.HeaderField`

Header fields, specified as an array of `matlab.net.http.HeaderField` objects.

`newFields` — Fields to add
`matlab.net.http.HeaderField`

Fields to add, specified as a vector or comma-separated list of one or more `matlab.net.http.HeaderField` objects.

`fName` — Header field name
string

Header field name, specified as a string.

`fValue` — Header field value
string | any type valid for `fName`

Header field value, specified as a string or any type valid for `fName`. To use the default value for the field, set `fValue` to `''`. If the last value is missing, then it is the same as specifying `[]`.

`index` — Location in header field array
integer

Location in header field array, specified as an integer. If `index` is greater than the length of the header or `index` is 0, the method adds fields to the end. If `index` is negative, the method counts from the end of the header.

See Also

Introduced in R2018a

changeFields

Class: `matlab.net.http.HeaderField`
Package: `matlab.net.http`

Change existing values in HeaderField array

Syntax

```
fields = changeFields(fields,
FieldName1,FieldValue1,...,FieldNameN,FieldValueN)
fields = changeFields(fields,newFields)
```

Description

`fields = changeFields(fields, FieldName1,FieldValue1,...,FieldNameN,FieldValueN)` changes the existing fields in each header field with the specified names to the indicated values and returns the updated array.

Name matching is case-insensitive. However, if you specify a name that differs in case from the existing field name, then the field name changes to the specified name. This usage does not change the class of an existing field.

This method throws an error when:

- All the specified fields are not already in the header.
- There is more than one field with the specified name.

`fields = changeFields(fields,newFields)` changes the existing fields in each header field to the names, values, and types specified in `fields`. This syntax might change the class of an existing field if the field name is a case-insensitive match to a name in `fields`.

Input Arguments

`fields` — Header fields
`matlab.net.http.HeaderField`

Header fields, specified as an array of `matlab.net.http.HeaderField` objects.

`FieldName` — Header field name
string

Header field name, specified as a string.

`FieldValue` — Header field value
string | any type valid for `fName`

Header field value, specified as a string or any type valid for `fName`. To use the default value for the field, set `fValue` to `''`. If the last value is missing, then it is the same as specifying `[]`.

`newFields` — Fields to add
`matlab.net.http.HeaderField`

Fields to add, specified as a vector or comma-separated list of one or more `matlab.net.http.HeaderField` objects.

See Also

`replaceFields`

Introduced in R2018a

getFields

Class: `matlab.net.http.HeaderField`
Package: `matlab.net.http`

Return header fields matching name or class

Syntax

```
[fields,indices] = getFields(headers,ids)
```

Description

`[fields,indices] = getFields(headers,ids)` returns fields and their indices in the header fields that match the given IDs.

Input Arguments

`headers` — Header fields
vector of `matlab.net.http.HeaderField` objects

Header fields, specified as a vector of `matlab.net.http.HeaderField` objects.

`ids` — Names to match
strings | `matlab.net.http.HeaderField` objects | `meta.class` objects

Names to match, specified as one of the following.

- String vector, character vector, cell array of character vectors, or comma-separated list of strings or character vectors containing the names of the fields to return. Names are not case-sensitive.
- Vector or comma-separated list of one or more `matlab.net.http.HeaderField` objects whose `Name` properties are used to determine which fields to return. Names are not case-sensitive. Header field values in these objects are ignored.

- Vector or comma-separated list of `meta.class` objects that are subclasses of `HeaderField`.

  The `getFields` method searches for names that match names supported by the specified classes, regardless of the class. If a subclass does not have a specific set of supported names, all header fields of that subclass are matched regardless of their names. Call the `HeaderField.displaySubclasses` method to see supported names.

Output Arguments

`fields` — Fields matching input
array of `matlab.net.http.HeaderField` objects

Fields matching input, returned as an array of `matlab.net.http.HeaderField` objects. If no matches exist, `getFields` returns an empty `HeaderField` array. The name search is case-insensitive.

`indices` — Index values of fields
array of integers | `[]`

Index values of fields, returned as an array of integers, or `[]` if no match.

If `messages` is a vector, then `getFields` returns fields matching `ids` from all messages. `indices` is a cell array of vectors, where `indices{i}` contains the indices of the matching fields in `messages{i}`.

See Also

Introduced in R2018a

removeFields

Class: `matlab.net.http.HeaderField`
Package: `matlab.net.http`

Remove fields from header field array

Syntax

```
fields = removeFields(fields,ids)
```

Description

`fields = removeFields(fields,ids)` removes all header fields matching `ids` and returns the updated array.

Input Arguments

`fields` — Header fields
`matlab.net.http.HeaderField`

Header fields, specified as an array of `matlab.net.http.HeaderField` objects.

`ids` — Names to match
strings | `matlab.net.http.HeaderField` objects | `meta.class` objects

Names to match, specified as one of the following.

- String vector, character vector, cell array of character vectors, or comma-separated list of strings or character vectors containing the names of the fields to remove. Names are not case-sensitive.
- Vector or comma-separated list of one or more `HeaderField` objects whose `Name` properties are used to determine which fields to remove. Names are not case-sensitive. Header field values are ignored.

- Vector or comma-separated list of `meta.class` objects that are subclasses of `HeaderField`.

  The `removeFields` method searches for names that match names supported by the specified classes, regardless of the class. If a subclass does not have a specific set of supported names, all header fields of that subclass are matched regardless of their names. Call the `HeaderField.displaySubclasses` method to see supported names.

See Also

Introduced in R2018a

replaceFields

Class: `matlab.net.http.HeaderField`
Package: `matlab.net.http`

Change values in or add fields to array of HeaderFields

Syntax

```
fields = replaceFields(fields,
FieldName1,FieldValue1,...,FieldNameN,FieldValueN)
fields = replaceFields(fields,newFields)
```

Description

`fields = replaceFields(fields, FieldName1,FieldValue1,...,FieldNameN,FieldValueN)` changes the existing fields in each header field with the specified names to the indicated values and returns the updated array. This method is the same as `changeFields`, but if a field does not exist, then `replaceFields` adds it to the end of fields.

`fields = replaceFields(fields,newFields)` changes the existing fields in each header field to the names, values, and types specified in `fields`. This syntax might change the class of an existing field if the field name is a case-insensitive match to a name in `fields`.

Input Arguments

`fields` — Header fields
`matlab.net.http.HeaderField`

Header fields, specified as an array of `matlab.net.http.HeaderField` objects.

`FieldName` — Header field name
string

Header field name, specified as a string.

`FieldValue` — Header field value
string | any type valid for `fName`

Header field value, specified as a string or any type valid for `fName`. To use the default value for the field, set `fValue` to `''`. If the last value is missing, then it is the same as specifying `[]`.

`newFields` — Fields to add
`matlab.net.http.HeaderField`

Fields to add, specified as a vector or comma-separated list of one or more `matlab.net.http.HeaderField` objects.

See Also

`changeFields`

Introduced in R2018a

matlab.net.http.HTTPException class

Package: `matlab.net.http`
Superclasses: `MException`

Exception thrown by HTTP services

Description

The `HTTPException` class contains information about errors.

The `RequestMessage.send` method throws an HTTP exception when an error occurs *after* it attempts to send a message. For instance, the method throws an exception when:

- Something fails during transmission and the server does not receive the message. Possible causes of failure include a network problem, timeout, or bad URI.
- The server receives the message, but something fails while it is sending a response.
- The server receives the message and sends a response, but the response cannot be converted based on its Content-Type. For example, a bad JSON string is received.

If an error occurs *before* the `RequestMessage.send` method attempts to send the message, then it throws a standard `MException` instead.

The `HTTPException` describes the error in the `MException.cause` property and returns a history of the transaction. The `History` property contains the message that was sent and the message that was received, if any.

Properties

`Request` — Request message as completed
`matlab.net.http.RequestMessage` object

Request message as completed, specified as a `matlab.net.http.RequestMessage` object. `Request` contains the last message that was sent or would have been sent. If this message header was successfully sent, then it is the same as the last entry in the `History.Request` property. Otherwise, the last entry in the `History` property does not contain the message you attempted to send.

Attributes:

GetAccess	public
SetAccess	private

URI — URI for last message
matlab.net.URI object

URI for the last message that was sent or would have been sent, specified as a `matlab.net.URI` object.

Attributes:

GetAccess	public
SetAccess	private

History — History of transaction
vector of `matlab.net.http.LogRecord` objects

History of the transaction, specified as a vector of `matlab.net.http.LogRecord` objects. `History` only contains messages whose headers were successfully sent or received. If an exception occurs when sending or receiving a message header, then `History` does not contain that message. If an error occurs while sending or receiving the payload, or during conversion of the data to or from the payload, then the `MessageBody` `Payload` property and/or `Data` properties might not be set.

Attributes:

GetAccess	public
SetAccess	private

Examples

Response Payload Not Valid Format for Content-Type

Send a message to the website in the `url` variable. This destination returns a message whose `Payload` property contains an invalid JPEG image for Content-Type `image/jpeg`.

```
try
    resp = RequestMessage().send(url);
catch e
```

```
    if isa(e,'matlab.net.http.HTTPException')
        response = e.History(end).Response;
        if ~isempty(response)
            data = response.Body.Data;
            payload = resonse.Body.Payload
        end
    end
end
```

The `payload` variable contains the bytes that were received as a `uint8` vector. Since the Content-Type is not character-based, the `data` variable is empty.

See Also

`LogRecord` | `MException` | `matlab.net.uri` | `send`

Introduced in R2016b

matlab.net.http.HTTPOptions class

Package: `matlab.net.http`

Options controlling HTTP message exchange

Description

Use the `HTTPOptions` class to create options for HTTP request messages. Use this object to specify options that are constant across several requests.

Creation

Description

`obj = matlab.net.http.HTTPOptions` creates HTTP options with default property values.

`obj = matlab.net.http.HTTPOptions(Name,Value)` creates HTTP options with additional properties specified by one or more name-value pair arguments. `Name` is the property name and `Value` is the corresponding value. You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Unspecified properties are set to their default values.

Properties

`Authenticate` — Whether Credentials used for authentication
`true` (default) | `false`

Whether Credentials are used for authentication, specified as `true` or `false`.

If `Authenticate` is true, then implement the supported authentication method requested by the server or proxy. The authentication is based on the `Credentials` property and the proxy user name and password set in MATLAB Web Preferences, if any. MATLAB supports `Basic` and `Digest` authentication only.

The response message contains the server or proxy authentication challenge when any of these conditions exist.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `logical`

`CertificateFilename` — File name of root certificates

`'default'` (default) | string | character vector

File name of root certificates, specified as a string scalar or character vector denoting the location of a file containing certificates. The file is in privacy-enhanced mail (PEM) format. The location must be in the current folder, in a folder on the MATLAB path, or a full or relative path to a file. If you specify the value `'default'`, then system certificates are used.

If you request an HTTPS connection, then the certificate from the server is validated against the certification authority certificates in the PEM file. Standard HTTPS mechanisms use this validation to validate the signature on the server certificate and the entire certificate chain. If verification fails, a connection is not allowed. You can disable the verification in cases where the server's certificate does not match the URI used to access it, by creating a `matlab.net.http.RequestMessage` and setting the `matlab.net.http.HTTPOptions.VerifyServerName` property to `false`. Use this option if you are confident that you are communicating directly with the intended server.

If you need additional certificates, add them to the system certificates. PEM files are ASCII files which are easily modified. Since security of HTTPS connections depends on the integrity of this file, protect it appropriately. MATLAB does not manage certificates or certificate files, but there are third-party tools for managing PEM files.

If `CertificateFilename` is empty, then MATLAB checks if the certificate domain of the server matches the host name of the server and that it is not expired. The signature is not validated.

Set `CertificateFilename` to empty (`''`) only if you cannot establish a connection due to a missing or expired certificate.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `char` | `string`

`ConnectTimeout` — Seconds to wait for initial server connection

10 (default) | integer | `Inf`

Seconds to wait for initial server connection, specified as an integer. If a proxy is involved, the timeout applies to the connection to the proxy; otherwise it applies to the connection to the server.

The default is 10 seconds. If the timeout period is exceeded, then `ConnectTimeout` throws an error. To disable timeouts, set `ConnectTimeout` to `Inf`.

`ConnectTimeout` determines how long to wait to complete a connection attempt with a server or proxy before throwing an error. This timeout does not limit how long it takes to receive a complete response.

When sending a request to a server through a proxy, consider using ResponseTimeout to limit the wait time.

Some operating systems have a maximum timeout enforced by the system. This timeout takes effect even if the value of `ConnectTimeout` is greater than the maximum. For example, on Windows 10, this timeout is 21 seconds.

Attributes:

GetAccess	public
SetAccess	public

`ConvertResponse` — How to process raw payload from server

`true` (default) | `false`

How to process raw payload received from a server in a `ResponseMessage`, specified as `true` or `false`.

If `ConvertResponse` is true, then

If `ConvertResponse` is false, then any specified `ContentConsumer` is ignored and the behavior depends on whether the Content-Type specifies character data.

In all cases, the `Payload` property is deleted unless you also set the `SavePayload` property to `true`.

`ConvertResponse` is ignored if the message was encoded (compressed) and one of these:

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `logical`

`DataTimeout` — Seconds to wait between data packets
`Inf` (default) | integer

Seconds to wait between data packets on the network, specified as an integer. The default value is `Inf`, meaning no timeout. This timeout, enforced once an initial connection is established, is useful for communications with potentially slow servers. If this timeout is exceeded while waiting to send or receive the next expected packet, then MATLAB closes the connection and throws an error. In this case, use the `matlab.net.http.HTTPException` `History` property to obtain any partial data.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`DecodeResponse` — Whether to decode compressed data
`true` (default) | `false`

Whether to decode compressed data, specified as `true` or `false`. Decoding means to decompress (decode) the response payload when the server returns compressed (encoded) data. Decoding occurs before conversion based on the Content-Type field.

A response is encoded when there is a Content-Encoding field that specifies a compression algorithm. MATLAB supports content coding values `gzip`, `x-gzip`, and `deflate`. The value `identity` means that there is no encoding, which is equivalent to the message having no Content-Encoding field. If MATLAB does not support the Content-Encoding type, decoding does not occur even if `DecodeResponse` is true.

If `DecodeResponse` is false and the payload is encoded, then:

Do not set this value to false for compressed responses if you are using a `ContentConsumer` that cannot process compressed data, unless you also set `ConvertResponse` to false to suppress use of the consumer. `FileConsumer` and `BinaryConsumer` are the only consumers provided by MATLAB that can process compressed data.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `logical`

`Credentials` — Authentication credentials

`matlab.net.http.Credentials` (default) | vector of `matlab.net.http.Credentials` objects | empty

Authentication credentials, specified as one or more `matlab.net.http.Credentials` objects. The default value is a default `matlab.net.http.Credentials` object. Use a default `Credentials` object to allow authentication for schemes such as Kerberos and NTLM on Windows. These schemes do not require specifying a user name or password.

`Credentials` are used only if the `Authenticate` property is true. You must specify at least one `Credentials` object for authentication to take place. If you set `Credentials` to empty, then no authentication takes place.

When you access the same server multiple times during a session, for maximum performance specify the same `Credentials` vector or same `HTTPOptions` object for each request. `Credentials` contains cached information that speeds up subsequent authentications.

If you provide `Credentials` for use with a proxy, and you want those `Credentials` to override a different user name and password specified in the MATLAB Web Preferences Panel, then specify the host and port of the proxy in the `ProxyURI` property of this `HTTPOptions` object or clear the **Use a proxy with authentication** option in the preferences panel.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`KeepAliveTimeout` — Seconds to keep server connection open
`Inf` (default) | `0`

Seconds to keep a server connection open after the initial connect, specified as `0` or `Inf`. Use this property to enable multiple successive messages to be sent over the same connection. An `Inf` value (the default) enables persistent connections, keeping the connection open as long as the server is able. A value of `0` closes the connection after each message. Other values are not supported.

The `KeepAliveTimeout` property has no effect on success of an operation. MATLAB always keeps the connection open long enough to get the expected response from the server, unless other timeouts are exceeded. However, a `0` value can seriously affect performance of sending many short messages to the same server.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`MaxRedirects` — Number of redirects allowed
20 (default) | 0 | integer | `Inf`

Number of redirects allowed, specified as an integer for a given request. The default number of redirects is 20. Set to 0 to disable redirection. Set to `Inf` to allow unlimited redirections.

If `MaxRedirects` is nonzero, then cookies received from the server in each redirect response are copied into the redirected message. After `MaxRedirects`, the response message contains the next redirect message.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`ProgressMonitorFcn` — Progress monitor handler
function handle | empty

Progress monitor handler, specified as a function handle to a `matlab.net.http.ProgressMonitor` object. If `UseProgressMonitor` is true, then MATLAB calls the `ProgressMonitor` function to report the progress of a transfer. If `UseProgressMonitor` is false or `ProgressMonitorFcn` is empty, then no progress is reported.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `function_handle`

`ProxyURI` — Proxy server address
empty (default) | `matlab.net.URI` | string

Proxy server address, specified as a `matlab.net.URI` object or a string of the form `host:port` or `//host:port`.

`ProxyURI` is used only if the `UseProxy` property is true. `ProxyURI` overrides the proxy specified in MATLAB Web Preferences and any proxy set in Windows system settings.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`ResponseTimeout` — Seconds to wait to receive initial response
`Inf` (default) | integer

Seconds to wait to receive the initial response (header) from the server after sending the last packet of a request, specified as an integer. The default value is `Inf`, meaning no timeout. If this timeout is exceeded, then MATLAB closes the connection and throws an error.

Use `ResponseTimeout` to limit the wait time when sending a request to a server through a proxy, since `ConnectTimeout` only applies to the proxy connection time.

`ResponseTimeout` is equivalent to the `Timeout` property set by `weboptions`.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

SavePayload — Whether Payload saved
`false` (default) | `true`

Whether Payload is saved, specified as `true` or `false`. The payload is the raw bytes received from or sent to the server, saved in the `MessageBody.Payload` property.

In a request message, setting `SavePayload` to `true` saves the payload after data conversion. In a response message, the bytes are saved before conversion.

Use `SavePayload` as a debugging tool. For example, the server cannot process the body of a request, or there is a failure converting a response body to a MATLAB type. Setting `SavePayload` to `true` might consume a considerable amount of memory because the payload is at least equal to the size of the converted data.

To retrieve the response payload without conversion, set the `ConvertResponse` property to `false` and read `MessageBody.Data` instead.

If an `HTTPException` occurs during message processing, then the payload received up to the point of failure is in `HTTPException.History(end).Response.Body.Payload`.

If `RequestMessage.Body` is a `ContentProvider` object, then MATLAB saves the provider's converted data in `Body.Payload`.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `logical`

`UseProgressMonitor` — Whether to display progress
`false` (default) | `true`

Whether to display progress, specified as `true` or `false`. Set `UseProgressMonitor` to `true` to report progress of a transfer using the function specified by the `ProgressMonitorFcn` property.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `logical`

`UseProxy` — Whether using proxy

`true` (default) | `false`

Whether using a proxy, specified as `true` or `false`.

If `UseProxy` is true, then MATLAB selects the first one of these proxies.

All requests go directly to the destination URI without a proxy when any of the following is true.

MATLAB automatically diverts a message to a proxy when `UseProxy` is true.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `logical`

`VerifyServerName` — Whether server name matches certificate

`true` (default) | `false`

Whether server name matches certificate, specified as `true` or `false`.

In a secure connection using `https` protocol, MATLAB verifies that the name of the server in the certificate matches the `Host` property in the URI of the request, or in the URI of the latest redirect request. This verification ensures that you are communicating with the intended server. To disable the verification in cases where the server certificate does not match the URI used to access it, set this property to false. For example, you want to access the server using an IP address or "localhost" and you are confident that you are communicating directly with the intended server.

Attributes:

GetAccess	public
SetAccess	public

Data Types: logical

Examples

Extend Connection Timeout

Increase connection time out to 20 seconds.

Change the default timeout option for the request message specified in the variable `request` sent to the server specified in the variable `url`.

```
options = matlab.net.http.HTTPOptions('ConnectTimeout',20);
response = request.send(url,options);
```

See Also

ContentConsumer | ContentProvider | Credentials | MessageBody | ProgressMonitor | URI | send

Topics

Web Preferences

Introduced in R2016b

matlab.net.http.LogRecord class

Package: `matlab.net.http`

HTTP history log record

Description

A vector of `LogRecord` objects represents a history of request-response message pairs exchanged between client and server during an HTTP operation, such as `RequestMessage.send`. The `send` method returns history as an output argument. When an error occurs, the `HTTPException` object contains a `History` property.

Use the `LogRecord` class for debugging.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Properties

`URI` — URI of request
`matlab.net.URI`

URI of the request, specified as a `matlab.net.URI` object.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Request` — Request message
`matlab.net.http.RequestMessage` object

Request message, specified as a `matlab.net.http.RequestMessage` object. `Request` contains a value if MATLAB attempted to send the header of the request message, even if

an exception occurs in the process of sending the payload. If an exception occurs, then the `Exception` property contains the `MException`.

The `Request.Body` property contains the body of the request message, if any. The `Request.Body.Payload` property is set only if you specify the `SavePayload` property in an `HTTPOptions` object.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`RequestTime` — Start and end times of request message
pair of `datetime` objects

Start and end times of a request message, specified as a pair of `datetime` objects. `RequestTime` contains a value only if the `Request` property is set.

The `RequestTime` values are an approximation of the times when the first and last bytes, including the payload, of a request message were sent. If an exception occurs during transmission, then the end time is the time of the exception, and the `Exception` property contains the `MException`.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `datetime`

`Response` — Response message
`matlab.net.http.ResponseMessage` object

Response message, specified as a `matlab.net.http.ResponseMessage` object. `Response` contains a value only if the complete header of the response is received successfully, even if an exception occurs receiving the payload. If an exception occurs while trying to send the request or during receipt of the header, then `Response` might be empty. If an exception occurs, then the `Exception` property contains the `MException` object.

The `Response.Body` property is set if you specify the `SavePayload` property in an `HTTPOptions` object and the response has a body. If `Response.Body` is set, then the `Response.Body.Payload` property is also set.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`ResponseTime` — Start and end times of response message
pair of `datetime` objects

Start and end times of the response message, specified as a pair of `datetime` objects. `ResponseTime` contains a value only if the `Response` property is set.

The `ResponseTime` values are an approximation of the times when the first and last bytes of the response message are received. If an exception occurs during receipt, then the end time is the time of the exception, and the `Exception` property contains the `MException` object.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `datetime`

`Disposition` — Disposition of exchange
`matlab.net.http.Disposition` enumeration

Disposition of the exchange, specified as a `matlab.net.http.Disposition` enumeration, indicating the result of the exchange. If the `Disposition` value is `Done`, then the `Exception` property is empty and all fields of the `LogRecord` contain their final values.

Some values of `Disposition` imply that an exception was saved in the `Exception` property. If an exception occurs, the `Response` or `Request` properties might not be set. When these values are set depends on whether the exception occurs before or during the transmission of the headers or the payload.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Exception` — Exception
`MException`

Exception, specified as an `MException` object that occurred during processing of the exchange. If an error occurred during the transmission, receipt, or processing of the response, this property is the `MException` containing the exception. The value of the `Disposition` property determines whether this property is set.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Methods

Public Methods

show Display or return human-readable version of vector of LogRecords

See Also

`Disposition` | `ResponseMessage` | `StartLine` | `matlab.net.URI` | `send`

Introduced in R2016b

show

Class: `matlab.net.http.LogRecord`
Package: `matlab.net.http`

Display or return human-readable version of vector of LogRecords

Syntax

```
show(records)
show(records,maxlength)

strs = show( ___ )
```

Description

`show(records)` displays the entire contents of the `records` array.

`show(records,maxlength)` displays `maxlength` characters of the message bodies. If a message body contains fewer than `maxlength` characters, then the method displays the entire message body.

`strs = show( ___ )` returns the output as a string instead of displaying it and can include any of the input arguments in previous syntaxes. `strs` contains one string for each `LogRecord` object in `records`.

Input Arguments

`records` — Log records
array of `matlab.net.http.LogRecord` objects

Log records, specified as an array of `matlab.net.http.LogRecord` objects.

`maxlength` — Number of bytes
integer

Number of bytes to convert, specified as an integer.

Examples

Display Last Log Record for Message Sent to `mathworks.com`.

```
import matlab.net.*
import matlab.net.http.*
import matlab.net.http.field.*

r = RequestMessage;
uri = URI('https://www.mathworks.com')
[resp,~,hist] = r.send(uri);

show(hist(end),100)
```

```
REQUEST 1 to https://www.mathworks.com/?requestedDomain=www.mathworks.com at 29-Apr-2016 09:30:35 - 29-Apr-2016 09:30:35 (
GET /?requestedDomain=www.mathworks.com HTTP/1.1
Cookie: Bjava-apache-all-ah=2831032236.0.0000; X_MW_DOMAIN=www.mathworks.com; JSESSIONID=236bd3c6939d903ca917331c1b60
User-Agent: MATLAB/9.0.0.379006 (R2016b)
Date: Fri, 29 Apr 2016 13:30:33 GMT
Connection: Close
Host: www.mathworks.com

RESPONSE 1 at 29-Apr-2016 09:30:35 - 29-Apr-2016 09:30:36 (0.141 sec)
HTTP/1.1 200 OK
Date: Fri, 29 Apr 2016 13:30:35 GMT
Server: MathworksServer_08
X-Powered-By: Servlet/3.0 JSP/2.2 (MathworksServer_08 Java/Oracle Corporation/1.7)
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Expires: Sun, 26 Apr 2015 11:56:16 GMT
Content-Type: text/html;charset=utf-8
Via: 1.1 www.mathworks.com
Vary: Accept-Encoding
Transfer-Encoding: chunked
Connection: close



<!DOCTYPE HTML>
<html>
<head>
<title>MathWorks - Makers of MATLAB and Simulink</title>
<meta char

<< 46590 total characters of text/html UTF-8 data >>
```

The information displayed depends on your system.

See Also

Introduced in R2016b

matlab.net.http.MediaType class

Package: `matlab.net.http`

Internet media type used in HTTP headers

Description

The `MediaType` class represents an internet media type, as defined in RFC 7231 — Media Type, section 3.1.1.1. The `MediaType` class also represents the contents of a Content-Type field or an Accept field.

Creation

Description

`obj = matlab.net.http.MediaType(type,paramName,paramValue)` creates a `MediaType` object given by `type`, with additional options specified by one or more `paramName,paramValue` pair arguments. You can specify several `paramName,paramValue` pair arguments in any order as `paramName1,paramValue1,...,paramNameN,paramValueN`.

`obj = matlab.net.http.MediaType(text)` parses `text` to create a media type. Quotes and escape characters within the values of parameters are removed. This constructor does not prevent creating a `MediaType` object with duplicate parameter names.

Input Arguments

`type` — Media type
string

Media type, specified as a string. The type must have the syntax of `type/subtype`.

`paramName` — Parameter name
string

Parameter name, specified as a nonempty string.

`paramValue` — Parameter value
string | type acceptable to the string function

Parameter value, specified as a nonempty string or type acceptable to the `string` function. The `string` function adds quotes and escapes as needed. Do not include quotation or escape characters within `paramValue`.

`text` — Media type
string

Media type, specified as a string.

Properties

Type — Primary type
string

Primary type, specified as a nonempty string.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `string`

Subtype — Subtype
string

Subtype, specified as a nonempty string.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Data Types: `string`

`Parameters` — Parameters
nx2 string matrix

Parameters, specified as an nx2 string matrix. Parameters(i,1) is the name of the ith parameter and Parameters(i,2) is its value. Use the `setParameter` method to add or change parameter values. To reorder or remove parameters, construct a new `MediaType` object with the parameters you want to use from an existing `MediaType` object.

Attributes:

```
GetAccess                                public
SetAccess                                private
```

Data Types: `string`

`MediaInfo` — Media type portion
`matlab.net.http.MediaType` object

Media type portion, specified as a `matlab.net.http.MediaType` object. `MediaInfo` contains the `MediaType` object without the parameters at or following the first `q` parameter.

Attributes:

```
GetAccess                           public
SetAccess                           immutable
Dependent                           true
```

`Weight` — Weight parameter `q`
double | empty | NaN

Weight parameter `q`, specified as double. If there is no `q` parameter, then `Weight` is empty. If the parameter cannot be converted to a double, then `Weight` is NaN. If you set this property, it must have a value in the range 0–1. Setting this value modifies the final `q` parameter in the object or adds one to the end of the parameter list.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

Methods

Public Methods

getParameter	Value of media type parameter
setParameter	Set value of media type parameter

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`eq`	Element by element comparisons between two `MediaType` arrays, returned as a logical array of the same dimensions. If the arrays are not the same dimensions, one must be a scalar, and then scalar-expansion is used. Two `MediaType` arrays are considered equal if they have the same `Type` and `Subtype` properties using a case-insensitive comparison. In addition, they have the same `Parameters` property names and values, where parameter names are case-insensitive and values are case-sensitive, except for 'charset' values, as described in RFC 2046 — Charset Parameter, section 4.1.2. Parameter order is not significant.
`isequal`	Compare two `MediaType` arrays with same dimensions and corresponding elements according to `eq`
`string`	`MediaType` as string
`char`	`MediaType` as character vector

See Also

External Websites

RFC 7231 — Media Type, section 3.1.1.1

Introduced in R2016b

getParameter

Class: `matlab.net.http.MediaType`
Package: `matlab.net.http`

Value of media type parameter

Syntax

```
value = getParameter(obj,name)
```

Description

`value = getParameter(obj,name)` returns the value of a `MediaType` parameter `name`. Parameter name matching is case-insensitive.

Input Arguments

`obj` — Media type
`matlab.net.http.MediaType` object

Media type, specified as a `matlab.net.http.MediaType` object.

`name` — Parameter name
string

Parameter name, specified as a string.

Output Arguments

`value` — Parameter value
array of strings | empty

Parameter value, returned as an array of nonempty strings. Parameter name matching is case-insensitive. If the parameter appears more than once, then `value` contains multiple elements. If the parameter does not exist, then `value` is empty.

See Also

`setParameter`

Introduced in R2016b

setParameter

Class: `matlab.net.http.MediaType`
Package: `matlab.net.http`

Set value of media type parameter

Syntax

```
newType = setParameter(mediaType,name,value)
```

Description

`newType = setParameter(mediaType,name,value)` returns a copy of the specified `mediaType` parameter with the parameter `name` set to `value`, creating one if it does not exist. Parameter name matching is case-insensitive. If a parameter exists with the same name, but different case, the case is changed to the specified `name`. If more than one match is found, only the last value is set.

A new parameter is added to the end of the list of parameters. If a `q` parameter exists in the list, the new parameter is treated as part of the `accept-params` rather than the media-type.

Input Arguments

`mediaType` — Media type
`matlab.net.http.MediaType` object

Media type, specified as a `matlab.net.http.MediaType` object.

`name` — Parameter name
string

Parameter name, specified as a string.

`value` — Parameter value
scalar string | character vector | any type supporting the `string` method

Parameter value, specified as a scalar string, character vector, or any type supporting the `string` method. The resulting string must not be empty.

If `value` is empty or an empty string, parameter `name` is removed.

There is no validation of `value`.

Output Arguments

`newType` — Updated media type
`matlab.net.http.MediaType` object

Updated media type, returned as a `matlab.net.http.MediaType` object.

See Also

`getParameter`

Introduced in R2016b

matlab.net.http.Message class

Package: `matlab.net.http`

HTTP request or response message

Description

The `Message` class is an abstract class used as a base class to create HTTP messages.

Properties

`StartLine` — First line of message

`matlab.net.http.StartLine` object

First line of message, specified as a `matlab.net.http.StartLine` object, if any.

Attributes:

GetAccess	public
SetAccess	public

`Header` — Message header

`[]` (default) | `matlab.net.http.HeaderField` object

Message header, specified as a `matlab.net.http.HeaderField` object or a vector of `HeaderField` objects. When you set the `Header` property, MATLAB checks the fields of the header to ensure that they are appropriate for the message type. The `RequestMessage` `send` and `complete` methods fill in any required header fields for a properly formed request.

Attributes:

GetAccess	public
SetAccess	public

Body — Message body

`[]` (default) | `matlab.net.http.MessageBody` object | `matlab.net.http.io.ContentProvider`

Message body, specified as a `matlab.net.http.MessageBody` object, `matlab.net.http.io.ContentProvider`, or data acceptable to the `MessageBody` constructor. By default, `Body` is empty (set to `[]`). A request message containing a `Body` property normally uses a method such as `'PUT'` or `'POST'`, not the default value `'GET'`, but this convention is not enforced.

In a completed or received message, if the message has a `ContentTypeField` header field, then the `MessageBody.ContentType` property is set to that value. Otherwise, `ContentType` is unchanged or empty.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Completed` — Whether message is complete
`false` (default) | `true`

Whether message is complete, specified as `true` or `false`. A `true` value means that the message is complete.

`RequestMessage` methods that validate messages (`send` and `complete`) set the `Completed` property to `true` when:

If the property is true, then these methods do not modify the message, and the `send` method sends the message without checking it for validity. Any later changes to this message change the value of `Completed` back to `false`.

Methods that return messages set `Completed` to `true` when:

If a request message contains data (`Body.Data` is not empty), then the property is true only if `Body.Payload` contains the raw data.

In a response message, the payload is set only if you set the `HTTPOptions.SavePayload` property to `true`.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Transient                                   true
```

Data Types: `logical`

Methods

Public Methods

addFields	Add fields to message header
changeFields	Change existing fields in message header
getFields	Return message header fields matching name or class
removeFields	Remove fields from message header
replaceFields	Change values in or add fields to message header
show	Display or return formatted version of message

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`string`	Message `Header`, `Body` and `StartLine` properties, as string. For multiple messages, returns a string array. If `Body` contains binary data that cannot be converted to characters, then the method displays a message indicating the length of the data in bytes. The string is an approximate representation of what the message looks like when sent or received. Use for logging, diagnostics, or debugging. For a formatted version of messages, use `show`.
`char`	Same as `string`, except returns message as character vector.

`isequal`	Returns `true` if the visible public properties of all messages in the two message arrays are equal.

Tips

- To send arbitrary headers and data in a request message, set the `Completed` property to `true` to prevent the `send` method from modifying the message. You still can use the `complete` method to validate the message, but the `send` method does not validate the message.

See Also

`ContentTypeField` | `HeaderField` | `MessageBody` | `RequestMessage` | `ResponseMessage`

Introduced in R2016b

addFields

Class: `matlab.net.http.Message`, `matlab.net.http.RequestMessage`, `matlab.net.http.ResponseMessage`
Package: `matlab.net.http`

Add fields to message header

Syntax

```
msg = addFields(msg,fields)
msg = addFields(msg,fName,fValue)
msg = addFields(msg,fName1,fValue1,...,fNameN,fValueN)
msg = addFields(msg,index, ___ )
```

Description

`msg = addFields(msg,fields)` adds `fields` to the end of the header of each message and returns the updated message.

`addFields` does not check for duplicate fields, but the `RequestMessage` `send` and `complete` methods might reject inappropriate duplicates.

To prevent the `send` or `complete` methods from automatically adding a particular header field, call `addFields` for that field with an empty (`[]`) `Value` property.

`msg = addFields(msg,fName,fValue)` adds field with name `fName` and value `fValue`.

`msg = addFields(msg,fName1,fValue1,...,fNameN,fValueN)` adds fields specified by `fName`, `fValue` pair arguments, in the order specified.

`msg = addFields(msg,index, ___ )` inserts fields at `index` and can include any of the input arguments in previous syntaxes.

Input Arguments

`msg` — Message
`matlab.net.http.Message` object

Message, specified as a `matlab.net.http.Message` object.

`fields` — Fields to add
one or more `matlab.net.http.HeaderField` objects

Fields to add, specified as a vector or comma-separated list of one or more `matlab.net.http.HeaderField` objects.

Example: `matlab.net.http.HeaderField('Accept','text/plain')`

`fName` — Header field name
string

Header field name, specified as a string.

Example: `'Accept'`

`fValue` — Header field value
string | any type valid for `fName`

Header field value, specified as a string or any type valid for `fName`. To use the default value for the field, set `fValue` to `''`. If the last value is missing, then it is the same as specifying `[]`.

Example: `'text/plain'`

`index` — Location in message header
integer

Location in the message header, specified as an integer. If `index` is greater than the length of the header or `index` is 0, the method adds fields to the end. If `index` is negative, the method counts from the end of the header.

Example: `-1` inserts `fields` before the last field

Examples

Add HTTP Accept Header Field

Create an Accept header field with value `'text/plain'` and add it to a default request message.

```
field = matlab.net.http.HeaderField('Accept','text/plain');
m = matlab.net.http.RequestMessage('get');
msg = addFields(m,field);
show(msg)
```

```
GET
Accept: text/plain
```

Add Multiple HTTP Header Fields

Add two header fields to a request message.

```
m = matlab.net.http.RequestMessage('get');
msg = addFields(m,'Accept','text/plain','Cache-Control','no-store, no-cache');
show(msg)
```

```
GET
Accept: text/plain
Cache-Control: no-store, no-cache
```

Insert HTTP Header Field

Create a request message with two header fields.

```
m = matlab.net.http.RequestMessage('get');
msg = addFields(m,'Accept','text/plain','Cache-Control','no-store, no-cache');
```

Insert a Content-Type header field before the last header field in a message.

```
f = matlab.net.http.HeaderField('Content-Type','text/plain');
msg = addFields(msg,-1,f);
show(msg)
```

```
GET
Accept: text/plain
```

```
Content-Type: text/plain
Cache-Control: no-store, no-cache
```

See Also

`matlab.net.http.HeaderField` | `matlab.net.http.RequestMessage`

Introduced in R2016b

changeFields

Class: `matlab.net.http.Message`, `matlab.net.http.RequestMessage`, `matlab.net.http.ResponseMessage`
Package: `matlab.net.http`

Change existing fields in message header

Syntax

```
msg = changeFields(msg,fields)
msg = changeFields(msg,
FieldName1,FieldValue1,...,FieldNameN,FieldValueN)
```

Description

`msg = changeFields(msg,fields)` changes the existing fields in each message to the names, values, and types specified in `fields` and returns the updated message. This syntax might change the class of an existing field if the field name is a case-insensitive match to a name in `fields`.

This method throws an error when:

- All the specified fields are not already in the header.
- There is more than one field with the specified name.

`msg = changeFields(msg,
FieldName1,FieldValue1,...,FieldNameN,FieldValueN)` changes fields with the specified names to the indicated values.

Name matching is case-insensitive. However, if you specify a name that differs in case from the existing field name, then the field name changes to the specified name. This usage does not change the class of an existing field.

Input Arguments

msg — Message
`matlab.net.http.Message` object

Message, specified as a `matlab.net.http.Message` object.

fields — Fields to change
`matlab.net.http.HeaderField` objects

Fields to change, specified as a vector or comma-separated list of one or more `matlab.net.http.HeaderField` objects.

Example: `'Accept','text/plain'`

FieldName1,FieldValue1,...,FieldNameN,FieldValueN — Fields defined by name-value pairs
name-value pairs

Fields defined by name-value pairs. `FieldName` is specified as a string or character vector, and `FieldValue` is specified as any type valid for `FieldName`.

To use the default value for the field, set `FieldValue` to `''`.

If the last value is missing, it is the same as specifying empty (`[]`).

Examples

Show How Changing Field Value Changes Field Class

Create two Content-Length fields using the `HeaderField` class and its subclass, `ContentLengthField`. Change the value of the Content-Length field in a message, which changes the class of the field, depending on how you create the original header field.

Create two header files with the same properties. The fields have the same value, but the classes are different.

```
h1 = matlab.net.http.HeaderField('Content-Length',5);
h2 = matlab.net.http.field.ContentLengthField(5);
compareNames = eq(h1.Name,h2.Name)
```

```
compareNames = logical
   1
```

```
compareValues = eq(h1.Value,h2.Value)
```

```
compareValues = logical
   1
```

Create a request message using the `h1` header field and display its class.

```
r = matlab.net.http.RequestMessage;
r.Header= h1;
h1Class = class(r.Header)
```

```
h1Class =
'matlab.net.http.HeaderField'
```

Change the Content-Length header field using the `h2` header field. The header field class changes.

```
r1 = r.changeFields(h2);
h2Class = class(r1.Header)
```

```
h2Class =
'matlab.net.http.HeaderField'
```

Change the Content-Length header field using a name-value pair. The header field class does not change.

```
r1 = r.changeFields('Content-length',5);
class(r1.Header)
```

```
ans =
'matlab.net.http.HeaderField'
```

See Also

`complete` | `matlab.net.http.HeaderField` | `send`

Introduced in R2016b

getFields

Class: `matlab.net.http.Message`, `matlab.net.http.RequestMessage`, `matlab.net.http.ResponseMessage`
Package: `matlab.net.http`

Return message header fields matching name or class

Syntax

```
[fields,indices] = getFields(messages,ids)
```

Description

`[fields,indices] = getFields(messages,ids)` returns fields and their indices in the messages that match the given IDs.

Input Arguments

`messages` — Messages
vector of `matlab.net.http.Message` objects

Messages, specified as a vector of `matlab.net.http.Message` objects.

Example: `getFields([msg1,msg2],'Date')`

`ids` — Names to match
strings | `matlab.net.http.HeaderField` objects | `meta.class` objects

Names to match, specified as one of the following.

- String vector, character vector, cell array of character vectors, or comma-separated list of strings or character vectors containing the names of the fields to return. Names are not case-sensitive.

- Vector or comma-separated list of one or more `matlab.net.http.HeaderField` objects whose `Name` properties are used to determine which fields to return. Names are not case-sensitive. Header field values in these objects are ignored.
- Vector or comma-separated list of `meta.class` objects that are subclasses of `HeaderField`.

  The `getFields` method searches for names that match names supported by the specified classes, regardless of the class. If a subclass does not have a specific set of supported names, all header fields of that subclass are matched regardless of their names. Call the `HeaderField.displaySubclasses` method to see supported names.

Example: `getFields(response,'Date','Content-Type')`

Output Arguments

`fields` — Fields matching input
array of `matlab.net.http.HeaderField` objects

Fields matching input, returned as an array of `matlab.net.http.HeaderField` objects. If no matches exist, `getFields` returns an empty `HeaderField` array. The name search is case-insensitive.

`indices` — Index values of fields
array of integers | `[]`

Index values of fields, returned as an array of integers, or `[]` if no match.

If `messages` is a vector, then `getFields` returns fields matching `ids` from all messages. `indices` is a cell array of vectors, where `indices{i}` contains the indices of the matching fields in `messages{i}`.

Examples

Verify Required Field Added to Request Message

Add a missing Date header field to a request message using the `RequestMessage.complete` method.

Create a request message to get text from the MathWorks website.

```
uri = matlab.net.URI('https://www.mathworks.com');
req = matlab.net.http.RequestMessage;
```

Verify the message contains a Date field. The date value reflects your system information. The value `indices{1} = []` indicates that `req` does not contain a Date field.

```
request = complete(req,uri);
[fields,indices] = getFields([req,request],?matlab.net.http.field.HTTPDateField)
```

```
fields = 
  DateField with properties:

     Name: "Date"
    Value: "Mon, 26 Aug 2019 21:41:01 GMT"


indices=1×2 cell
    {0x0 double}    {[3]}
```

See Also

`matlab.net.http.HeaderField` | `matlab.net.http.HeaderField.displaySubclasses`

Introduced in R2016b

removeFields

Class: `matlab.net.http.Message`, `matlab.net.http.RequestMessage`, `matlab.net.http.ResponseMessage`
Package: `matlab.net.http`

Remove fields from message header

Syntax

```
msg = removeFields(msg,ids)
```

Description

`msg = removeFields(msg,ids)` removes all header fields matching `ids` and returns the updated message.

Input Arguments

`msg` — Message
`matlab.net.http.Message` object

Message, specified as a `matlab.net.http.Message` object.

`ids` — Names to match
strings | `matlab.net.http.HeaderField` objects | `meta.class` objects

Names to match, specified as one of the following.

- String vector, character vector, cell array of character vectors, or comma-separated list of strings or character vectors containing the names of the fields to remove. Names are not case-sensitive.
- Vector or comma-separated list of one or more `HeaderField` objects whose `Name` properties are used to determine which fields to remove. Names are not case-sensitive. Header field values are ignored.

- Vector or comma-separated list of `meta.class` objects that are subclasses of `HeaderField`.

  The `removeFields` method searches for names that match names supported by the specified classes, regardless of the class. If a subclass does not have a specific set of supported names, all header fields of that subclass are matched regardless of their names. Call the `HeaderField.displaySubclasses` method to see supported names.

Example: `getFields(response,'Date','Content-Type')`

Examples

Remove Header Field from Request Message

Create a request message with two header fields, Cache-Control and Content-Type, and display the message.

```
import matlab.net.http.HeaderField

m = matlab.net.http.RequestMessage('get');
f1 = HeaderField('Cache-Control','no-store, no-cache');
f2 = HeaderField('Content-Type','text/html;charset=utf-8');
m = addFields(m,f1,f2);
show(m)
```

```
GET
Cache-Control: no-store, no-cache
Content-Type: text/html;charset=utf-8
```

Remove the Cache-Control field and display the updated message.

```
msg = removeFields(m,'Cache-Control');
show(msg)
```

```
GET
Content-Type: text/html;charset=utf-8
```

Remove Header Field Subclasses

The `MediaRangeField` class has two subclasses, `AcceptField` and `ContentTypeField`. An ID that specifies the `MediaRangeField` class matches all fields of class `MediaRangeField`, `AcceptField`, and `ContentTypeField`, plus any fields with the `Name` property `'Accept'` or `'Content-Type'`.

Create message `m` with a `ContentTypeField` header field.

```
m = matlab.net.http.RequestMessage('get');
f = matlab.net.http.HeaderField('Content-Type','text/html;charset=utf-8');
m = addFields(m,f);
```

Remove all header fields that are subclasses of the `MediaRangeField` class and display the updated message.

```
id = ?matlab.net.http.field.MediaRangeField;
msg = removeFields(m,id);
show(msg)
```

```
GET
```

See Also

`matlab.net.http.HeaderField` | `matlab.net.http.HeaderField.displaySubclasses`

Introduced in R2016b

replaceFields

Class: `matlab.net.http.Message`, `matlab.net.http.RequestMessage`, `matlab.net.http.ResponseMessage`
Package: `matlab.net.http`

Change values in or add fields to message header

Syntax

```
msg = replaceFields(msg,
FieldName1,FieldValue1,...,FieldNameN,FieldValueN)
msg = replaceFields(msg,newFields)
```

Description

`msg = replaceFields(msg, FieldName1,FieldValue1,...,FieldNameN,FieldValueN)` replaces fields with the specified names to the indicated values specified in `FieldName1,FieldValue1,...,FieldNameN,FieldValueN` and returns the updated message.

This method is the same as the `changeFields` method, but if a field does not exist which matches the name or class, then the method adds a new one to the end of the header instead of throwing an error.

`msg = replaceFields(msg,newFields)` replaces the existing fields in each message to the names, values, and types specified in `newFields` and returns the updated message.

Input Arguments

`msg` — Message
`matlab.net.http.Message` object

Message, specified as a `matlab.net.http.Message` object.

`FieldName1,FieldValue1,...,FieldNameN,FieldValueN` — Fields defined by name-value pairs
name-value pairs

Fields defined by name-value pairs. `FieldName` is specified as a string or character vector, and `FieldValue` is specified as any type valid for `FieldName`.

To use the default value for the field, set `FieldValue` to `''`.

If the last value is missing, it is the same as specifying empty (`[]`).

`newFields` — Names to match
strings | `matlab.net.http.HeaderField` objects | `meta.class` objects

Names to match, specified as one of the following.

- String vector, character vector, cell array of character vectors, or comma-separated list of strings or character vectors containing the names of the fields to replace. Names are not case-sensitive.
- Vector or comma-separated list of one or more `HeaderField` objects whose `Name` properties are used to determine which fields to replace. Names are not case-sensitive. Header field values are ignored.
- Vector or comma-separated list of `meta.class` objects that are subclasses of `HeaderField`.

  The `replaceFields` method searches for names that match names supported by the specified classes, regardless of the class. If a subclass does not have a specific set of supported names, all header fields of that subclass are matched regardless of their names. Call the `HeaderField.displaySubclasses` method to see supported names.

See Also

`matlab.net.http.HeaderField` | `matlab.net.http.HeaderField.displaySubclasses`

Introduced in R2016b

show

Class: `matlab.net.http.Message`, `matlab.net.http.RequestMessage`, `matlab.net.http.ResponseMessage`
Package: `matlab.net.http`

Display or return formatted version of message

Syntax

```
show(msg)
show(msg,maxlength)
```

```
str = show( ___ )
```

Description

`show(msg)` displays the whole message in the Command Window, including the `Header` and `Body` properties. If `Body` contains binary data that cannot be converted to characters, then the method displays a message indicating the length of the data in bytes.

Use this syntax for diagnostics or debugging.

`show(msg,maxlength)` displays the first `maxlength` characters of `Body`. If `Body` is longer than `maxlength` characters, then the method displays the total length of the data.

`str = show( ___ )` returns a string containing the information to be displayed, and can include any of the input arguments in previous syntaxes.

Input Arguments

`msg` — Message
`matlab.net.http.Message` object

Message, specified as a `matlab.net.http.Message` object or vector of `Message` objects.

`maxlength` — Number of bytes
integer

Number of bytes, specified as an integer to convert.

Examples

Show Contents of HTTP Request Message Containing Image Data

When a message body contains image/jpeg data, MATLAB displays the number of bytes in the image.

Send a message to the heritage.stsci.edu website requesting an image of Jupiter. Display the response message.

```
request = matlab.net.http.RequestMessage;
uri = matlab.net.URI('http://heritage.stsci.edu/2007/14/images/p0714aa.jpg');
r = send(request,uri);
show(r)
```

```
HTTP/1.1 200 OK
Date: Tue, 09 Aug 2016 19:58:56 GMT
Server: Apache/2.2.15  Red Hat  mod_ssl/2.2.15 OpenSSL/1.0.1e-fips Phusion_Passenger/4
Last-Modified: Thu, 08 Mar 2007 18:56:52 GMT
ETag: "1ca16a7-31265-42b2edeb90500"
Accept-Ranges: bytes
Content-Length: 201317
Connection: close
Content-Type: image/jpeg

<< 201317 bytes of image/jpeg data >>
```

Show 100 Characters of HTTP Message Content

Display the first 100 characters only of the body of a message from `mathworks.com`. MATLAB displays text indicating the total number of characters in the message body.

```
request = matlab.net.http.RequestMessage;
uri = matlab.net.URI('https://www.mathworks.com');
```

```
r = send(request,uri);
show(r,100)
```

```
HTTP/1.1 200 OK
Date: Mon, 02 Oct 2017 12:55:05 GMT
Server: Apache/2.2.15 (CentOS)
Host-ID: wcxm4
X-Content-Type-Options: nosniff
Last-Modified: Sun, 01 Oct 2017 16:57:27 GMT
ETag: "f4b6"
Accept-Ranges: bytes
Cache-Control: max-age=14400
Expires: Mon, 02 Oct 2017 16:55:05 GMT
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 9500
Content-Type: text/html; charset=UTF-8
Connection: close

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-w

<< 62581 total characters of text/html UTF-8 data >>
```

See Also

Introduced in R2016b

matlab.net.http.MessageBody class

Package: `matlab.net.http`

Body of HTTP message

Description

A `MessageBody` object contains the body of an HTTP message. In a request message, set the `Body` property to your data or to a `MessageBody` object containing your data. In a response message, this object contains the received data.

Message data is represented by two properties. The `Payload` property contains the raw bytes sent to or received from the network. The `Data` property contains the `Payload` as a MATLAB type. Often only one of these two properties is set.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

A `MessageBody` object contains the body of an HTTP message. In a request message, set the `Body` property to your data or to a `MessageBody` object containing your data. In a response message, this object contains the received data.

Message data is represented by two properties. The `Payload` property contains the raw bytes sent to or received from the network. The `Data` property contains the `Payload` as a MATLAB type. Often only one of these two properties is set.

Properties

`Data` — Message data

`uint8` vector or array | character array | string | structure array | image data | XML DOM | audio data | table

Message data, specified as one of the following MATLAB data types. For conversion details, see “HTTP Data Type Conversion”.

In a request message, the `Data` property is the MATLAB data before conversion to a `uint8` payload. Data conversion happens when you call the `RequestMessage` `send` or `complete` methods. The conversion depends on the Content-Type you specify in the message and the type of `Data`. If you do not specify a Content-Type, then the `send` and `complete` methods try to deduce the type from the data and add the appropriate ContentTypeField to the request message.

In a response message, `Data` represents the `uint8` payload converted to a MATLAB type based on the Content-Type specified by the server. If conversion fails, then `Data` remains empty and the raw data appears in the `Payload` property.

To suppress automatic conversion of the response `Payload`, set the `HTTPOptions.ConvertResponse` property to `false`. In that case, `Data` contains either a string for character data or a `uint8` vector of bytes.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`
`Dependent`	`true`

`ContentType` — Content-Type of data

`MediaType` object

Content-Type of the `Data` property, specified as a read-only `matlab.net.http.MediaType` object. This property determines how contents of the `Data` property was converted to or from the payload. If the Content-Type has a `MediaType` with a charset attribute, then the charset determines the encoding.

This property usually has the same values as the result of calling `convert` on the Content-Type field in the message containing this `MessageBody`.

You do not set `ContentType` in a request message. When you create a `MessageBody` object, this property is empty. When you copy `MessageBody` into a request message, `ContentType` is set to the value of the `ContentTypeField` in the message, if there is one. The `RequestMessage` `send` and `complete` methods set `ContentType` based on the type of `Data` and the value of the `ContentTypeField` in the request message.

In a response message, `ContentType` is based on the `ContentTypeField` of the message.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Transient                                   true
```

`ContentCoding` — Content-Encoding of payload
string

This property is set to indicate that the `Payload` property of a `ResponseMessage` is encoded. When `ContentCoding` is set, no processing was done on the payload and the `Data` property is empty.

If MATLAB receives a message whose payload is encoded using a compression algorithm that it supports, such as `gzip` or `deflate`, it automatically decodes that payload before attempting any other conversions. If decoding was successful, it optionally stores the decoded payload in `Payload` and the converted payload (if any) in `Data`. In that case, this property is empty to indicate that the `Payload` is not encoded.

If the payload was encoded but decoding was not successful, or you suppressed decoding by setting the `HTTPOptions.DecodePayload` property to `false`, then the unprocessed still-encoded payload is returned in `Payload`, `Data` is left empty, and `ContentCoding` is set to a vector of strings representing the value of the Content-Encoding header field in the response message. In this case, you can save the `Payload` as is (for example, write it to a file), or process it according to the compression algorithms specified in `ContentCoding`. For example, if the value is `gzip`, you can write the data to a file and use the `gunzip` command to process the data.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Transient                                   true
```

Data Types: `string`

`Payload` — Raw bytes sent in message
empty (default) | `uint8` vector | scalar string | character vector

Raw bytes sent in a message, specified as a `uint8` vector, scalar string, or character vector. As a convenience in a request message, you can set `Payload` to a scalar string or character vector. MATLAB converts the value to a `uint8` vector. In a response message, `Payload` is always a `uint8` vector.

In a request message:

In a response message, `Payload` is the raw bytes received in these cases:

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`
`Dependent`	`true`

Methods

Public Methods

show Display or return formatted version of message body data

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`string`	MessageBody.Data property as a string. Use for diagnostics or debugging. If the `Data` property contains text, then the method displays the entire contents. Use the `MessageBody.show` method to limit the amount of displayed text. If `Data` does not contain text, then the method displays a message indicating the length of the data in bytes. If `MessageBody` is an array, then returns an array of the same size.
`char`	`Data` property as a character vector, similar to the `string` method. However, if `MessageBody` is an array, then the method returns a cell array of character vectors of the same size.

See Also

`ContentTypeField` | `MediaType` | MessageBody.Data | `RequestMessage` | `ResponseMessage` | `imread` | `imwrite`

Topics

"HTTP Data Type Conversion"

External Websites

RFC 2616, section 14.17 Content-Type

Introduced in R2016b

show

Class: `matlab.net.http.MessageBody`
Package: `matlab.net.http`

Display or return formatted version of message body data

Syntax

```
show(body)
show(body,maxlength)

str = show( ___ )
```

Description

`show(body)` displays the entire `Data` property of a message body if `Data` is a scalar string or character vector. For other types, the method displays a message indicating the length of the data in bytes, if known. If length is unknown, then the value displayed is the length of the data if `body` were sent in a request message.

The `show` method formats `Data` when possible. For example, the method puts separators between parts of a multipart message and separately converts each part to a string. If `Data` is a string or character vector, then the `string` and `show` methods return the same value.

Use this method for diagnostics or debugging.

`show(body,maxlength)` displays the first `maxlength` characters of `Data`. If `Data` is longer than `maxlength` characters, then the method displays the total length of the data.

`str = show( ___ )` returns a string containing the information to be displayed, and can include any of the input arguments in previous syntaxes.

Input Arguments

body — Message body
`matlab.net.http.MessageBody` object

Message body, specified as a `matlab.net.http.MessageBody` object.

`maxlength` — Number of bytes
integer

Number of bytes to convert, specified as an integer.

Examples

Display Information About Image Data

Display data about an image received from the Hubble Heritage website.

```
req = matlab.net.http.RequestMessage;
uri = matlab.net.URI('http://heritage.stsci.edu/2007/14/images/p0714aa.jpg');
r = send(req,uri);
show(r.Body)
```

```
<< 201317 bytes of image/jpeg data >>
```

Display First 100 Characters of HTML Body

Request the MATLAB `webread` documentation from `mathworks.com`.

```
uri = matlab.net.URI('https://www.mathworks.com/help/matlab/ref/webread.html');
request = matlab.net.http.RequestMessage;
r = send(request,uri);
```

Display the first 100 characters of the message body.

```
show(r.Body,100)
```

```
<!DOCTYPE HTML>
<html>
```

```
<head>
<title>Read content from RESTful web service - MATLAB webread</title>


<< 94866 total characters of text/html UTF-8 data >>
```

See Also

Introduced in R2016b

matlab.net.http.MessageType class

Package: `matlab.net.http`

HTTP message type

Description

The `MessageType` enumeration class identifies the message type.

Enumeration Member Name
`Request`
`Response`

See Also

`ProgressMonitor`

Introduced in R2016b

matlab.io.datastore.Partitionable class

Package: `matlab.io.datastore`

Add parallelization support to datastore

Description

`matlab.io.datastore.Partitionable` is an abstract mixin class that adds parallelization support to your custom datastore for use with Parallel Computing Toolbox and MATLAB Parallel Server.

To use this mixin class, you must inherit from `matlab.io.datastore.Partitionable` class, in addition to inheriting from the `matlab.io.Datastore` base class. Type the following syntax as the first line of your class definition file:

```
classdef MyDatastore < matlab.io.Datastore & ...
                       matlab.io.datastore.Partitionable
    ...
end
```

To add support for parallel processing to your custom datastore, you must:

- Inherit from the additional class `matlab.io.datastore.Partitionable`.
- Define these additional methods: `maxpartitions` and `partition`.

For more details and steps to create your custom datastore with parallel processing support, see “Develop Custom Datastore”.

Methods

maxpartitions	Maximum number of partitions possible
numpartitions	Default number of partitions
partition	Partition a datastore

Attributes

Sealed	false

For information on class attributes, see “Class Attributes”.

Examples

Build Datastore with Parallel Processing Support

Build a datastore with parallel processing support and use it to bring your custom or proprietary data into MATLAB®. Then, process the data in a parallel pool.

Create a `.m` class definition file that contains the code implementing your custom datastore. You must save this file in your working folder or in a folder that is on the MATLAB® path. The name of the `.m` file must be the same as the name of your object constructor function. For example, if you want your constructor function to have the name MyDatastorePar, then the name of the `.m` file must be `MyDatastorePar.m`. The `.m` class definition file must contain the following steps:

- Step 1: Inherit from the datastore classes.
- Step 2: Define the constructor and the required methods.
- Step 3: Define your custom file reading function.

In addition to these steps, define any other properties or methods that you need to process and analyze your data.

```
%% STEP 1: INHERIT FROM DATASTORE CLASSES
classdef MyDatastorePar < matlab.io.Datastore & ...
        matlab.io.datastore.Partitionable

    properties(Access = private)
        CurrentFileIndex double
        FileSet matlab.io.datastore.DsFileSet
    end

    % Property to support saving, loading, and processing of
    % datastore on different file system machines or clusters.
    % In addition, define the methods get.AlternateFileSystemRoots()
```

```
    % and set.AlternateFileSystemRoots() in the methods section.
    properties(Dependent)
        AlternateFileSystemRoots
    end

%% STEP 2: DEFINE THE CONSTRUCTOR AND THE REQUIRED METHODS
    methods
        % Define your datastore constructor
        function myds = MyDatastorePar(location,altRoots)
            myds.FileSet = matlab.io.datastore.DsFileSet(location,...
                'FileExtensions','.bin', ...
                'FileSplitSize',8*1024);
            myds.CurrentFileIndex = 1;

            if nargin == 2
                 myds.AlternateFileSystemRoots = altRoots;
            end

            reset(myds);
        end

        % Define the hasdata method
        function tf = hasdata(myds)
            % Return true if more data is available
            tf = hasfile(myds.FileSet);
        end

        % Define the read method
        function [data,info] = read(myds)
            % Read data and information about the extracted data
            % See also: MyFileReader()
            if ~hasdata(myds)
                msgII = ['Use the reset method to reset the datastore ',...
                         'to the start of the data.'];
                msgIII = ['Before calling the read method, ',...
                          'check if data is available to read ',...
                          'by using the hasdata method.'];
                error('No more data to read.\n%s\n%s',msgII,msgIII);
            end

            fileInfoTbl = nextfile(myds.FileSet);
            data = MyFileReader(fileInfoTbl);
            info.Size = size(data);
            info.FileName = fileInfoTbl.FileName;
```

```
            info.Offset = fileInfoTbl.Offset;

            % Update CurrentFileIndex for tracking progress
            if fileInfoTbl.Offset + fileInfoTbl.SplitSize >= ...
                    fileInfoTbl.FileSize
                myds.CurrentFileIndex = myds.CurrentFileIndex + 1 ;
            end
        end

        % Define the reset method
        function reset(myds)
            % Reset to the start of the data
            reset(myds.FileSet);
            myds.CurrentFileIndex = 1;
        end

        % Define the partition method
        function subds = partition(myds,n,ii)
            subds = copy(myds);
            subds.FileSet = partition(myds.FileSet,n,ii);
            reset(subds);
        end

        % Getter for AlternateFileSystemRoots property
        function altRoots = get.AlternateFileSystemRoots(myds)
            altRoots = myds.FileSet.AlternateFileSystemRoots;
        end

        % Setter for AlternateFileSystemRoots property
        function set.AlternateFileSystemRoots(myds,altRoots)
            try
              % The DsFileSet object manages AlternateFileSystemRoots
              % for your datastore
              myds.FileSet.AlternateFileSystemRoots = altRoots;

              % Reset the datastore
              reset(myds);
            catch ME
              throw(ME);
            end
        end

    end
```

```
    methods (Hidden = true)
        % Define the progress method
        function frac = progress(myds)
            % Determine percentage of data read from datastore
            if hasdata(myds)
               frac = (myds.CurrentFileIndex-1)/...
                              myds.FileSet.NumFiles;
            else
               frac = 1;
            end
        end
    end

    methods(Access = protected)
        % If you use the  FileSet property in the datastore,
        % then you must define the copyElement method. The
        % copyElement method allows methods such as readall
        % and preview to remain stateless
        function dscopy = copyElement(ds)
            dscopy = copyElement@matlab.mixin.Copyable(ds);
            dscopy.FileSet = copy(ds.FileSet);
        end

        % Define the maxpartitions method
        function n = maxpartitions(myds)
            n = maxpartitions(myds.FileSet);
        end
    end
end

%% STEP 3: IMPLEMENT YOUR CUSTOM FILE READING FUNCTION
function data = MyFileReader(fileInfoTbl)
% create a reader object using FileName
reader = matlab.io.datastore.DsFileReader(fileInfoTbl.FileName);

% seek to the offset
seek(reader,fileInfoTbl.Offset,'Origin','start-of-file');

% read fileInfoTbl.SplitSize amount of data
data = read(reader,fileInfoTbl.SplitSize);

end
```

Your custom datastore is now ready. Use your custom datastore to read and process the data in a parallel pool.

Read Data Using Custom Datastore And Process in Parallel Pool

Use custom datastore to preview and read your proprietary data into MATLAB for parallel processing.

This example uses a simple data set to illustrate a workflow using your custom datastore. The data set is a collection of 15 binary (`.bin`) files where each file contains a column (`1` variable) and `10000` rows (records) of unsigned integers.

```
dir('*.bin')
```

```
binary_data01.bin  binary_data05.bin  binary_data09.bin  binary_data13.bin
binary_data02.bin  binary_data06.bin  binary_data10.bin  binary_data14.bin
binary_data03.bin  binary_data07.bin  binary_data11.bin  binary_data15.bin
binary_data04.bin  binary_data08.bin  binary_data12.bin
```

Create a datastore object using the `MyDatastorePar` function. For implementation details of `MyDatastorePar`, see the example **Build Datastore with Parallel Processing Support**.

```
folder = fullfile('*.bin');
ds = MyDatastorePar(folder);
```

Preview the data from the datastore.

```
preview(ds)
```

```
ans = 8x1 uint8 column vector

   113
   180
   251
    91
    29
    66
   254
   214
```

Identify the number of partitions for your datastore. If you have Parallel Computing Toolbox (PCT), then you can use `n = numpartitions(ds,myPool)`, where `myPool` is `gcp` or `parpool`.

```
n = numpartitions(ds);
```

Partition the datastore into n parts and n workers in a parallel pool.

```
parfor ii = 1:n
    subds = partition(ds,n,ii);
      while hasdata(subds)
        data = read(subds);
        % do something
      end
end
```

Process Datastore on Different Platforms

To process your datastore with parallel and distributed computing that involves different platform cloud or cluster machines, you must pre-define `'AlternateFileSystemRoots'` parameter. For instance, create a datastore on your local machine, and analyze a small portion of the data. Then, scale up your analysis to the entire dataset using Parallel Computing Toolbox and MATLAB Parallel Server.

Create a datastore using `MyDatastorePar` and assign a value to the `'AlternateFileSystemRoots'` property. For implementation details of `MyDatastorePar`, see the example `Build Datastore with Parallel Processing Support`.

To set the value for the `'AlternateFileSystemRoots'` property, identify the root paths for your data on the different platforms. The root paths differ based on the machine or file system. For instance, if you access your data using these root paths:

- `"Z:\DataSet"` from the Windows machine.
- `"/nfs-bldg001/DataSet"` from the MATLAB Parallel Server Linux cluster.

Then, associate these root paths using the `AlternateFileSystemRoots` property.

```
altRoots = ["Z:\DataSet","/nfs-bldg001/DataSet"];
ds = MyDatastorePar('Z:\DataSet',altRoots);
```

Analyze a small portion of the data on your local machine. For instance, get a partitioned subset of the data and clean the data by removing any missing entries. Then, examine a plot of the variables.

```
tt = tall(partition(ds,100,1));
summary(tt);
```

```
% analyze your data
tt = rmmissing(tt);
plot(tt.MyVar1,tt.MyVar2)
```

Scale up your analysis to the entire dataset by using MATLAB Parallel Server cluster (Linux cluster). For instance, start a worker pool using the cluster profile, and then perform analysis on the entire dataset by using parallel and distributed computing capabilities.

```
parpool('MyMjsProfile')
tt = tall(ds);
summary(tt);
% analyze your data
tt = rmmissing(tt);
plot(tt.MyVar1,tt.MyVar2)
```

Tips

- For your custom datastore implementation, best practice is not to implement the `numpartitions` method.

See Also

`datastore` | `mapreduce` | `matlab.io.Datastore` | `matlab.io.datastore.HadoopLocationBased`

Topics

"Develop Custom Datastore"
"Tall Arrays for Out-of-Memory Data"
"Partition a Datastore in Parallel" (Parallel Computing Toolbox)

Introduced in R2017b

maxpartitions

Class: `matlab.io.datastore.Partitionable`
Package: `matlab.io.datastore`

Maximum number of partitions possible

Syntax

```
n = maxpartitions(ds)
```

Description

`n = maxpartitions(ds)` returns the maximum number of partitions for datastore `ds`.

Input Arguments

`ds` — Input datastore
`matlab.io.Datastore` object

Input datastore, specified as a `matlab.io.Datastore` object. To create a `Datastore` object, see `matlab.io.Datastore`.

Attributes

`Abstract`	`true`
`Access`	`protected`

To learn about attributes of methods, see Method Attributes.

Examples

Build Datastore with Parallel Processing Support

Build a datastore with parallel processing support and use it to bring your custom or proprietary data into MATLAB®. Then, process the data in a parallel pool.

Create a `.m` class definition file that contains the code implementing your custom datastore. You must save this file in your working folder or in a folder that is on the MATLAB® path. The name of the `.m` file must be the same as the name of your object constructor function. For example, if you want your constructor function to have the name MyDatastorePar, then the name of the `.m` file must be `MyDatastorePar.m`. The `.m` class definition file must contain the following steps:

- Step 1: Inherit from the datastore classes.
- Step 2: Define the constructor and the required methods.
- Step 3: Define your custom file reading function.

In addition to these steps, define any other properties or methods that you need to process and analyze your data.

```
%% STEP 1: INHERIT FROM DATASTORE CLASSES
classdef MyDatastorePar < matlab.io.Datastore & ...
        matlab.io.datastore.Partitionable

    properties(Access = private)
        CurrentFileIndex double
        FileSet matlab.io.datastore.DsFileSet
    end

    % Property to support saving, loading, and processing of
    % datastore on different file system machines or clusters.
    % In addition, define the methods get.AlternateFileSystemRoots()
    % and set.AlternateFileSystemRoots() in the methods section.
    properties(Dependent)
        AlternateFileSystemRoots
    end

%% STEP 2: DEFINE THE CONSTRUCTOR AND THE REQUIRED METHODS
    methods
        % Define your datastore constructor
```

```
function myds = MyDatastorePar(location,altRoots)
    myds.FileSet = matlab.io.datastore.DsFileSet(location,...
        'FileExtensions','.bin', ...
        'FileSplitSize',8*1024);
    myds.CurrentFileIndex = 1;

    if nargin == 2
         myds.AlternateFileSystemRoots = altRoots;
    end

    reset(myds);
end

% Define the hasdata method
function tf = hasdata(myds)
    % Return true if more data is available
    tf = hasfile(myds.FileSet);
end

% Define the read method
function [data,info] = read(myds)
    % Read data and information about the extracted data
    % See also: MyFileReader()
    if ~hasdata(myds)
        msgII = ['Use the reset method to reset the datastore ',...
                 'to the start of the data.'];
        msgIII = ['Before calling the read method, ',...
                  'check if data is available to read ',...
                  'by using the hasdata method.'];
        error('No more data to read.\n%s\n%s',msgII,msgIII);
    end

    fileInfoTbl = nextfile(myds.FileSet);
    data = MyFileReader(fileInfoTbl);
    info.Size = size(data);
    info.FileName = fileInfoTbl.FileName;
    info.Offset = fileInfoTbl.Offset;

    % Update CurrentFileIndex for tracking progress
    if fileInfoTbl.Offset + fileInfoTbl.SplitSize >= ...
            fileInfoTbl.FileSize
        myds.CurrentFileIndex = myds.CurrentFileIndex + 1 ;
    end
end
```

```
        % Define the reset method
        function reset(myds)
            % Reset to the start of the data
            reset(myds.FileSet);
            myds.CurrentFileIndex = 1;
        end

        % Define the partition method
        function subds = partition(myds,n,ii)
            subds = copy(myds);
            subds.FileSet = partition(myds.FileSet,n,ii);
            reset(subds);
        end

        % Getter for AlternateFileSystemRoots property
        function altRoots = get.AlternateFileSystemRoots(myds)
            altRoots = myds.FileSet.AlternateFileSystemRoots;
        end

        % Setter for AlternateFileSystemRoots property
        function set.AlternateFileSystemRoots(myds,altRoots)
            try
              % The DsFileSet object manages AlternateFileSystemRoots
              % for your datastore
              myds.FileSet.AlternateFileSystemRoots = altRoots;

              % Reset the datastore
              reset(myds);
            catch ME
              throw(ME);
            end
        end

    end

    methods (Hidden = true)
        % Define the progress method
        function frac = progress(myds)
            % Determine percentage of data read from datastore
            if hasdata(myds)
               frac = (myds.CurrentFileIndex-1)/...
                              myds.FileSet.NumFiles;
            else
```

```
                frac = 1;
            end
        end
    end

    methods(Access = protected)
        % If you use the  FileSet property in the datastore,
        % then you must define the copyElement method. The
        % copyElement method allows methods such as readall
        % and preview to remain stateless
        function dscopy = copyElement(ds)
            dscopy = copyElement@matlab.mixin.Copyable(ds);
            dscopy.FileSet = copy(ds.FileSet);
        end

        % Define the maxpartitions method
        function n = maxpartitions(myds)
            n = maxpartitions(myds.FileSet);
        end
    end
end

%% STEP 3: IMPLEMENT YOUR CUSTOM FILE READING FUNCTION
function data = MyFileReader(fileInfoTbl)
% create a reader object using FileName
reader = matlab.io.datastore.DsFileReader(fileInfoTbl.FileName);

% seek to the offset
seek(reader,fileInfoTbl.Offset,'Origin','start-of-file');

% read fileInfoTbl.SplitSize amount of data
data = read(reader,fileInfoTbl.SplitSize);

end
```

Your custom datastore is now ready. Use your custom datastore to read and process the data in a parallel pool.

Tips

- If a datastore is based on files and uses the `matlab.io.datastore.DsFileSet` object, then use the `maxpartitions` method of the `DsFileSet` object.

See Also

`matlab.io.datastore.Partitionable` | `numpartitions` | `partition`

Introduced in R2017b

numpartitions

Class: `matlab.io.datastore.Partitionable`
Package: `matlab.io.datastore`

Default number of partitions

Syntax

```
n = numpartitions(ds)
n = numpartitions(ds,pool)
```

Description

`n = numpartitions(ds)` returns the default number of partitions for datastore `ds`.

`n = numpartitions(ds,pool)` returns a reasonable number of partitions to parallelize datastore access over the parallel pool specified by `pool`. To parallelize datastore access, you must have Parallel Computing Toolbox installed.

Input Arguments

`ds` — Input datastore
`matlab.io.Datastore` object

Input datastore, specified as a `matlab.io.Datastore` object. To create a `Datastore` object, see `matlab.io.Datastore`.

`pool` — Parallel pool
parallel pool object

Parallel pool, specified as a parallel pool object.

Example: `gcp`

Attributes

Sealed	true
Abstract	false

To learn about attributes of methods, see Method Attributes.

Examples

Build Datastore with Parallel Processing Support

Build a datastore with parallel processing support and use it to bring your custom or proprietary data into MATLAB®. Then, process the data in a parallel pool.

Create a `.m` class definition file that contains the code implementing your custom datastore. You must save this file in your working folder or in a folder that is on the MATLAB® path. The name of the `.m` file must be the same as the name of your object constructor function. For example, if you want your constructor function to have the name MyDatastorePar, then the name of the `.m` file must be `MyDatastorePar.m`. The `.m` class definition file must contain the following steps:

- Step 1: Inherit from the datastore classes.
- Step 2: Define the constructor and the required methods.
- Step 3: Define your custom file reading function.

In addition to these steps, define any other properties or methods that you need to process and analyze your data.

```
%% STEP 1: INHERIT FROM DATASTORE CLASSES
classdef MyDatastorePar < matlab.io.Datastore & ...
        matlab.io.datastore.Partitionable

    properties(Access = private)
        CurrentFileIndex double
        FileSet matlab.io.datastore.DsFileSet
    end

    % Property to support saving, loading, and processing of
    % datastore on different file system machines or clusters.
```

```
    % In addition, define the methods get.AlternateFileSystemRoots()
    % and set.AlternateFileSystemRoots() in the methods section.
    properties(Dependent)
        AlternateFileSystemRoots
    end

%% STEP 2: DEFINE THE CONSTRUCTOR AND THE REQUIRED METHODS
    methods
        % Define your datastore constructor
        function myds = MyDatastorePar(location,altRoots)
            myds.FileSet = matlab.io.datastore.DsFileSet(location,...
                'FileExtensions','.bin', ...
                'FileSplitSize',8*1024);
            myds.CurrentFileIndex = 1;

            if nargin == 2
                 myds.AlternateFileSystemRoots = altRoots;
            end

            reset(myds);
        end

        % Define the hasdata method
        function tf = hasdata(myds)
            % Return true if more data is available
            tf = hasfile(myds.FileSet);
        end

        % Define the read method
        function [data,info] = read(myds)
            % Read data and information about the extracted data
            % See also: MyFileReader()
            if ~hasdata(myds)
                msgII = ['Use the reset method to reset the datastore ',...
                         'to the start of the data.'];
                msgIII = ['Before calling the read method, ',...
                          'check if data is available to read ',...
                          'by using the hasdata method.'];
                error('No more data to read.\n%s\n%s',msgII,msgIII);
            end

            fileInfoTbl = nextfile(myds.FileSet);
            data = MyFileReader(fileInfoTbl);
            info.Size = size(data);
```

```
            info.FileName = fileInfoTbl.FileName;
            info.Offset = fileInfoTbl.Offset;

            % Update CurrentFileIndex for tracking progress
            if fileInfoTbl.Offset + fileInfoTbl.SplitSize >= ...
                    fileInfoTbl.FileSize
                myds.CurrentFileIndex = myds.CurrentFileIndex + 1 ;
            end
        end

        % Define the reset method
        function reset(myds)
            % Reset to the start of the data
            reset(myds.FileSet);
            myds.CurrentFileIndex = 1;
        end

        % Define the partition method
        function subds = partition(myds,n,ii)
            subds = copy(myds);
            subds.FileSet = partition(myds.FileSet,n,ii);
            reset(subds);
        end

        % Getter for AlternateFileSystemRoots property
        function altRoots = get.AlternateFileSystemRoots(myds)
            altRoots = myds.FileSet.AlternateFileSystemRoots;
        end

        % Setter for AlternateFileSystemRoots property
        function set.AlternateFileSystemRoots(myds,altRoots)
            try
              % The DsFileSet object manages AlternateFileSystemRoots
              % for your datastore
              myds.FileSet.AlternateFileSystemRoots = altRoots;

              % Reset the datastore
              reset(myds);
            catch ME
              throw(ME);
            end
        end

    end
```

```
    methods (Hidden = true)
        % Define the progress method
        function frac = progress(myds)
            % Determine percentage of data read from datastore
            if hasdata(myds)
               frac = (myds.CurrentFileIndex-1)/...
                              myds.FileSet.NumFiles;
            else
               frac = 1;
            end
        end
    end

    methods(Access = protected)
        % If you use the  FileSet property in the datastore,
        % then you must define the copyElement method. The
        % copyElement method allows methods such as readall
        % and preview to remain stateless
        function dscopy = copyElement(ds)
            dscopy = copyElement@matlab.mixin.Copyable(ds);
            dscopy.FileSet = copy(ds.FileSet);
        end

        % Define the maxpartitions method
        function n = maxpartitions(myds)
            n = maxpartitions(myds.FileSet);
        end
    end
end

%% STEP 3: IMPLEMENT YOUR CUSTOM FILE READING FUNCTION
function data = MyFileReader(fileInfoTbl)
% create a reader object using FileName
reader = matlab.io.datastore.DsFileReader(fileInfoTbl.FileName);

% seek to the offset
seek(reader,fileInfoTbl.Offset,'Origin','start-of-file');

% read fileInfoTbl.SplitSize amount of data
data = read(reader,fileInfoTbl.SplitSize);

end
```

Your custom datastore is now ready. Use your custom datastore to read and process the data in a parallel pool.

Tips

- For your custom datastore implementation, we recommend that you do not implement the `numpartitions` method.

Algorithms

In the default implementation, the number of partitions, `n`, is selected as the minimum between these two values:

- Value returned by the `maxpartitions` method on the datastore object.
- Three times the number of workers available.

See Also

`matlab.io.datastore.Partitionable` | `maxpartitions` | `partition`

Introduced in R2017b

partition

Class: `matlab.io.datastore.Partitionable`
Package: `matlab.io.datastore`

Partition a datastore

Syntax

```
subds = partition(ds,n,index)
```

Description

`subds = partition(ds,n,index)` partitions datastore `ds` into the number of parts specified by `n` and returns the partition corresponding to the index `index`. The partitioned datastore `subds` is of the same type as the input datastore `ds`.

Input Arguments

`ds` — Input datastore
`matlab.io.Datastore` object

Input datastore, specified as a `matlab.io.Datastore` object. To create a `Datastore` object, see `matlab.io.Datastore`.

`n` — Number of partitions
positive integer

Number of partitions, specified as a positive integer. To get a reasonable value for `n`, use the `numpartitions` function.

When you specify a value of `n` that is not in the range of partitions available for the datastore, the `partition` method returns an empty datastore. For more information, see “Empty Datastores” on page 1-6782. For instance, if a datastore can hold up to `10` partitions, then the output of the `partition` method depends on the value of `n`.

- If the specified value of `n` is less than or equal to `10`, then the `partition` method returns the partition specified by the `index`. For example, `partition(ds,10,1)` returns a copy of the first partition of the original datastore `ds`.
- If the specified value of `n` is greater than `10`, then the `partition` method returns an empty datastore. For example, `partition(ds,100,11)` returns an empty datastore.

Example: 3

Data Types: `double`

`index` — Index
positive integer

Index, specified as a positive integer.

Example: 1

Data Types: `double`

Attributes

`Abstract`	`true`

To learn about attributes of methods, see Method Attributes.

Examples

Build Datastore with Parallel Processing Support

Build a datastore with parallel processing support and use it to bring your custom or proprietary data into MATLAB®. Then, process the data in a parallel pool.

Create a `.m` class definition file that contains the code implementing your custom datastore. You must save this file in your working folder or in a folder that is on the MATLAB® path. The name of the `.m` file must be the same as the name of your object constructor function. For example, if you want your constructor function to have the name MyDatastorePar, then the name of the `.m` file must be `MyDatastorePar.m`. The `.m` class definition file must contain the following steps:

- Step 1: Inherit from the datastore classes.
- Step 2: Define the constructor and the required methods.
- Step 3: Define your custom file reading function.

In addition to these steps, define any other properties or methods that you need to process and analyze your data.

```
%% STEP 1: INHERIT FROM DATASTORE CLASSES
classdef MyDatastorePar < matlab.io.Datastore & ...
        matlab.io.datastore.Partitionable

    properties(Access = private)
        CurrentFileIndex double
        FileSet matlab.io.datastore.DsFileSet
    end

    % Property to support saving, loading, and processing of
    % datastore on different file system machines or clusters.
    % In addition, define the methods get.AlternateFileSystemRoots()
    % and set.AlternateFileSystemRoots() in the methods section.
    properties(Dependent)
        AlternateFileSystemRoots
    end

%% STEP 2: DEFINE THE CONSTRUCTOR AND THE REQUIRED METHODS
    methods
        % Define your datastore constructor
        function myds = MyDatastorePar(location,altRoots)
            myds.FileSet = matlab.io.datastore.DsFileSet(location,...
                'FileExtensions','.bin', ...
                'FileSplitSize',8*1024);
            myds.CurrentFileIndex = 1;

            if nargin == 2
                 myds.AlternateFileSystemRoots = altRoots;
            end

            reset(myds);
        end

        % Define the hasdata method
        function tf = hasdata(myds)
            % Return true if more data is available
```

```
        tf = hasfile(myds.FileSet);
    end

    % Define the read method
    function [data,info] = read(myds)
        % Read data and information about the extracted data
        % See also: MyFileReader()
        if ~hasdata(myds)
            msgII = ['Use the reset method to reset the datastore ',...
                     'to the start of the data.'];
            msgIII = ['Before calling the read method, ',...
                      'check if data is available to read ',...
                      'by using the hasdata method.'];
            error('No more data to read.\n%s\n%s',msgII,msgIII);
        end

        fileInfoTbl = nextfile(myds.FileSet);
        data = MyFileReader(fileInfoTbl);
        info.Size = size(data);
        info.FileName = fileInfoTbl.FileName;
        info.Offset = fileInfoTbl.Offset;

        % Update CurrentFileIndex for tracking progress
        if fileInfoTbl.Offset + fileInfoTbl.SplitSize >= ...
                fileInfoTbl.FileSize
            myds.CurrentFileIndex = myds.CurrentFileIndex + 1 ;
        end
    end

    % Define the reset method
    function reset(myds)
        % Reset to the start of the data
        reset(myds.FileSet);
        myds.CurrentFileIndex = 1;
    end

    % Define the partition method
    function subds = partition(myds,n,ii)
        subds = copy(myds);
        subds.FileSet = partition(myds.FileSet,n,ii);
        reset(subds);
    end

    % Getter for AlternateFileSystemRoots property
```

```
    function altRoots = get.AlternateFileSystemRoots(myds)
        altRoots = myds.FileSet.AlternateFileSystemRoots;
    end

    % Setter for AlternateFileSystemRoots property
    function set.AlternateFileSystemRoots(myds,altRoots)
        try
          % The DsFileSet object manages AlternateFileSystemRoots
          % for your datastore
          myds.FileSet.AlternateFileSystemRoots = altRoots;

          % Reset the datastore
          reset(myds);
        catch ME
          throw(ME);
        end
    end

end

methods (Hidden = true)
    % Define the progress method
    function frac = progress(myds)
        % Determine percentage of data read from datastore
        if hasdata(myds)
           frac = (myds.CurrentFileIndex-1)/...
                          myds.FileSet.NumFiles;
        else
           frac = 1;
        end
    end
end

methods(Access = protected)
    % If you use the  FileSet property in the datastore,
    % then you must define the copyElement method. The
    % copyElement method allows methods such as readall
    % and preview to remain stateless
    function dscopy = copyElement(ds)
        dscopy = copyElement@matlab.mixin.Copyable(ds);
        dscopy.FileSet = copy(ds.FileSet);
    end

    % Define the maxpartitions method
```

```
        function n = maxpartitions(myds)
            n = maxpartitions(myds.FileSet);
        end
    end
end

%% STEP 3: IMPLEMENT YOUR CUSTOM FILE READING FUNCTION
function data = MyFileReader(fileInfoTbl)
% create a reader object using FileName
reader = matlab.io.datastore.DsFileReader(fileInfoTbl.FileName);

% seek to the offset
seek(reader,fileInfoTbl.Offset,'Origin','start-of-file');

% read fileInfoTbl.SplitSize amount of data
data = read(reader,fileInfoTbl.SplitSize);

end
```

Your custom datastore is now ready. Use your custom datastore to read and process the data in a parallel pool.

More About

Empty Datastores

An empty datastore is a datastore object that does not contain any records. For an empty datastore, your custom datastore methods must satisfy these conditions:

- `hasdata` must return `false`.
- `read` must return an error.
- `numpartitions` and `maxpartitions` must return `0`.
- `partition` must return an empty datastore.
- `preview` and `readall` must return empty data that preserves the non-tall dimensions. For example, if the `read` method on a nonempty datastore returns data that is of size 5-by-15-by-25, then the `preview` and `readall` methods must return empty data of size 0-by-15-by-25.

Non-Tall Dimensions

Dimensions other than the first dimension of the array. For an array of size 5-by-15-by-25, the tall dimension is 5 and the non-tall dimensions are 15 and 25.

Tips

- In your implementation of the `partition` method, you must include these steps.
  - Before creating a partitioned datastore `subds`, create a deep copy of the original datastore `ds`.
  - At the end of the `partition` method, reset the partitioned datastore `subds`.

  For a sample implementation of the `partition` method, see “Add Support for Parallel Processing”.
- When a partition of a datastore contains no readable record, the `read` method must return empty data. The non-tall dimensions of this empty data must match the non-tall dimensions of the `read` method output on a partition with readable records. This requirement ensures that the behavior of the `readall` method matches the behavior of the `gather` function.

See Also

`gather` | `maxpartitions` | `numpartitions` | `readall`

Introduced in R2017b

parula

Parula colormap array

Syntax

```
c = parula
c = parula(m)
```

Description

`c = parula` returns the parula colormap as a three-column array with the same number of rows as the colormap for the current figure. If no figure exists, then the number of rows is equal to the default length of 256. Each row in the array contains the red, green, and blue intensities for a specific color. The intensities are in the range [0,1], and the color scheme looks like this image.

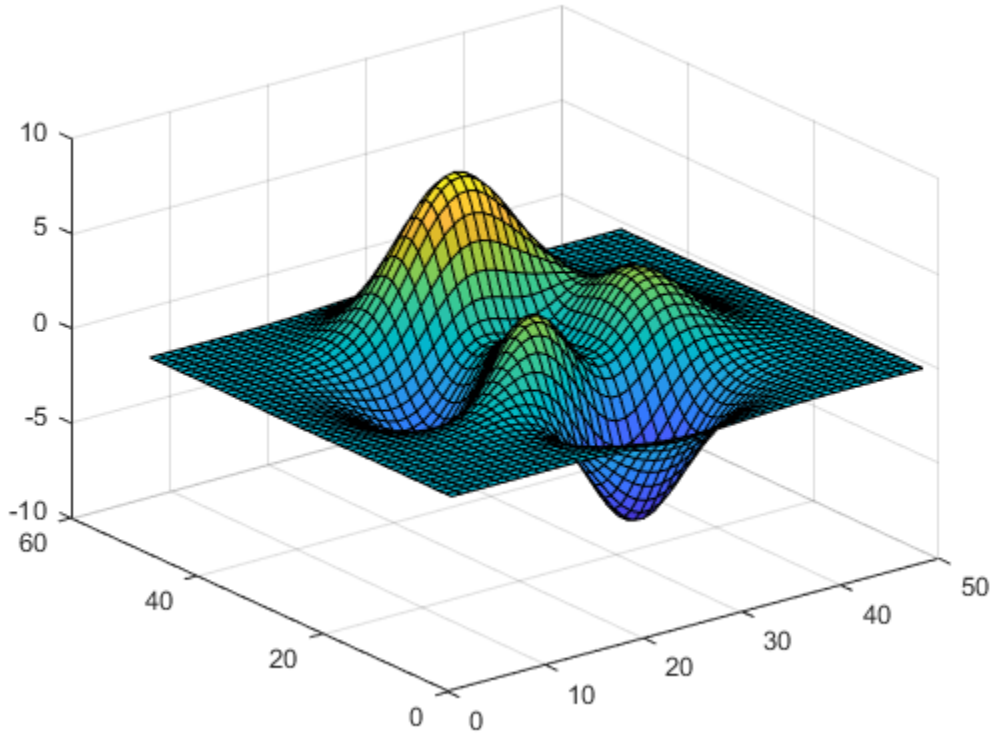
`c = parula(m)` returns the colormap with `m` colors.

Examples

Reverse the Colormap

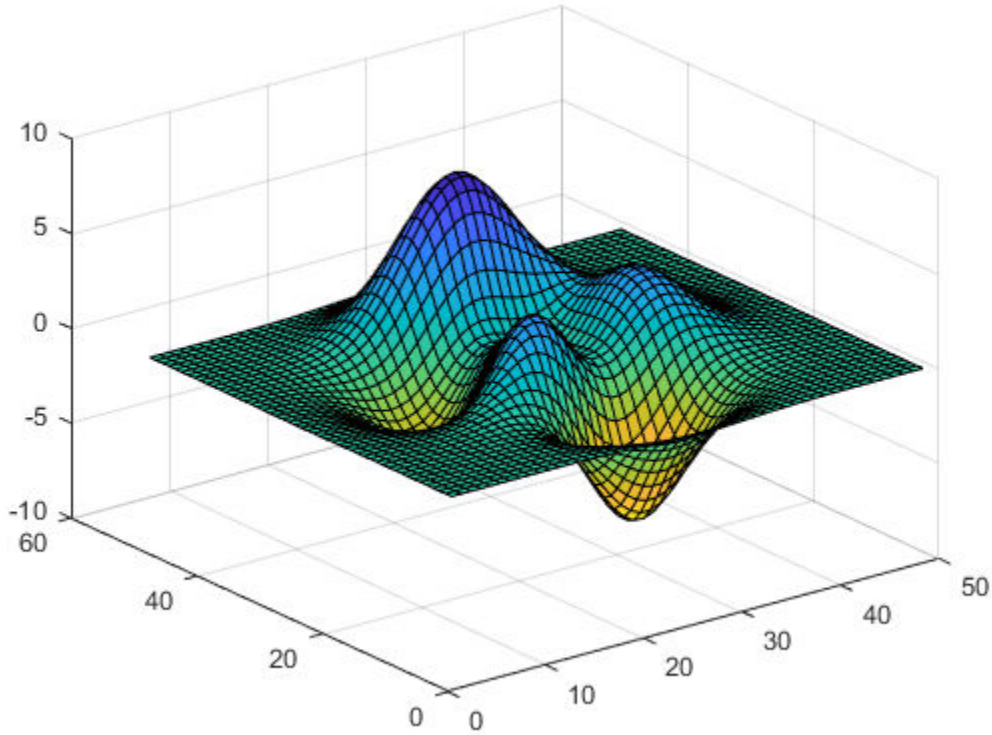
Plot a surface with the default parula colormap.

```
surf(peaks);
```



Get the parula colormap array and reverse the order. Then apply the modified colormap to the surface.

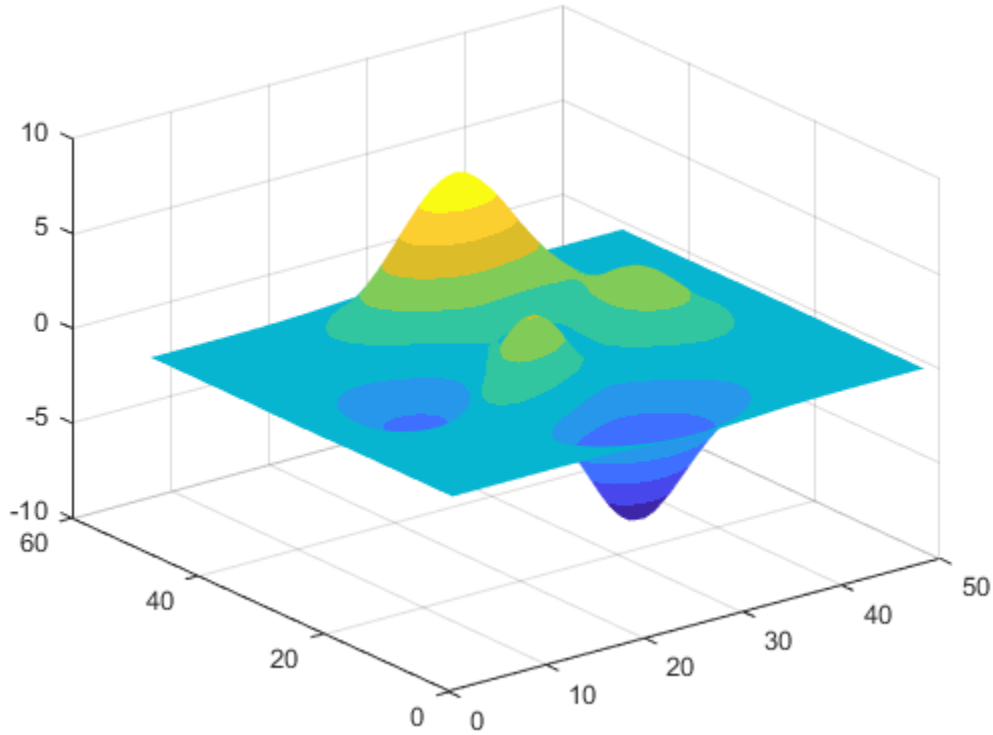
```
c = parula;
c = flipud(c);
colormap(c);
```



Downsample the Parula Colormap

Get a downsampled version of the parula colormap containing only ten colors. Then display the contours of the peaks function by applying the colormap and interpolated shading.

```
c = parula(10);
surf(peaks);
colormap(c);
shading interp;
```



Input Arguments

`m` — Number of colors

256 (default) | scalar integer value

Number of colors, specified as a scalar integer value. The default value of `m` is equal to the length of the colormap for the current figure. If no figure exists, the default value is 256.

Data Types: `single` | `double`

Compatibility Considerations

Colormaps Have 256 Colors by Default

Behavior changed in R2019b

Starting in R2019b, colormaps have 256 colors by default.

In R2019a and previous releases, the default size is 64. If you have code that depends on a colormap having 64 colors, specify the number of colors when you set the colormap for the figure, axes, or chart. For example, `colormap(parula(64))` sets the figure's colormap to the 64-color `parula` colormap.

Alternatively, you can change the default colormap for all figures within your MATLAB session:

```
set(groot,'defaultFigureColormap',parula(64))
```

Enhanced Color Scheme

Behavior changed in R2017a

Starting in R2017a, the colors in the `parula` colormap are slightly different than in previous releases. This table shows a comparison between the R2017a and earlier versions of the colormap. The visual change is subtle; however, you might notice more colorful colors and smoother transitions between colors.

R2017a and later	
R2016b and earlier	

See Also

`colormap`

Topics

“Change Color Scheme Using a Colormap”

Introduced in R2014b

parquetinfo

Get information about Parquet file

Description

The `ParquetInfo` object contains information about a Parquet file such as file size, variable names, variable types, and compression schemes. To get information about a Parquet file, create the `ParquetInfo` object using the `parquetinfo` function.

Creation

Syntax

```
info = parquetinfo(filename)
```

Description

`info = parquetinfo(filename)` returns a `info` object for the Parquet file specified by `filename`.

Input Arguments

`filename` — Name of Parquet file
character vector | string scalar

Name of Parquet file, specified as a character vector or string scalar. `ParquetInfo` works with Parquet 1.0 or Parquet 2.0 files.

Depending on the location of the file, `filename` can take on one of these forms.

<table>
<tr><th>Location</th><th>Form</th></tr>
<tr><td>Current folder or folder on the MATLAB path</td><td>Specify the name of the file in <code>filename</code>.

Example: <code>'data.parquet'</code></td></tr>
<tr><td>File in a folder</td><td>If the file is not in the current folder or in a folder on the MATLAB path, then specify the full or relative path name.

Example: <code>'C:\myFolder\data.parquet'</code>

Example: <code>'myDir\myFile.ext'</code></td></tr>
<tr><td>Remote Location</td><td>If the file is stored at a remote location, then <code>filename</code> must contain the full path of the file specified as a uniform resource locator (URL) of the form:

<code>scheme_name://path_to_file/my_file.ext</code>

Based on your remote location, <code>scheme_name</code> can be one of the values in this table.
<table>
<tr><th>Remote Location</th><th><code>scheme_name</code></th></tr>
<tr><td>Amazon S3</td><td><code>s3</code></td></tr>
<tr><td>Windows Azure Blob Storage</td><td><code>wasb</code>, <code>wasbs</code></td></tr>
<tr><td>HDFS</td><td><code>hdfs</code></td></tr>
</table>
For more information, see “Work with Remote Data”.

Example: <code>'s3://bucketname/path_to_file/data.parquet'</code></td></tr>
</table>

Data Types: `char` | `string`

Properties

`Filename` — Absolute path to Parquet file
string scalar

This property is read-only.

Absolute path to Parquet file, specified as a string scalar.

Data Types: `string`

`FileSize` — File size in bytes
double

This property is read-only.

File size in bytes, specified as `double`.

Data Types: `double`

`NumRowGroups` — Number of row groups
double

This property is read-only.

Number of row groups, specified as a `double`.

Data Types: `double`

`RowGroupHeights` — Number of rows in each row group
double

This property is read-only.

Number of rows in each row group, specified as a `double`.

Data Types: `double`

`VariableNames` — Variable names
string array

This property is read-only.

Variable names, specified as a string array. If the Parquet file contains `N` variables, then `VariableNames` is an array of size `1`-by-`N` containing the names of the variables.

Data Types: `string`

`VariableTypes` — Variable data types
string array

This property is read-only.

Variable data types, specified as a string array. If the Parquet file contains `N` variables, then `VariableTypes` is an array of size `1`-by-`N` containing datatype names for each variable.

Each element in the array is the name of the MATLAB datatype to which the corresponding variable in the Parquet file maps.

Data Types: `string`

`VariableCompression` — Variable compression algorithm
string array

This property is read-only.

Variable compression algorithm, specified as a string array. If the Parquet file contains `N` variables, then `VariableCompression` is an array of size `1`-by-`N` containing compression algorithm names.

Each element in the array corresponds to the compression algorithm used to compress that variable in the Parquet file.

Examples

Get Information About Parquet File

Use the `praquetinfo` function to create a `ParquetInfo` object containing information about the file.

```
info = parquetinfo('outages.parquet')
```

```
info = 
  ParquetInfo with properties:

               Filename: "B:\matlab\toolbox\matlab\demos\outages.parquet"
               FileSize: 44202
           NumRowGroups: 1
        RowGroupHeights: 1468
          VariableNames: [1x6 string]
          VariableTypes: [1x6 string]
    VariableCompression: [1x6 string]
```

Display the name, type, and compression scheme for the third variable in the file.

```
disp([info.VariableNames(3)  info.VariableTypes(3) info.VariableCompression(3)])
```

```
    "Loss"    "double"    "snappy"
```

See Also

parquetDatastore | parquetread | parquetwrite

Introduced in R2019a

parquetread

Read columnar data from a Parquet file

Syntax

```
T = parquetread(filename)
T = parquetread(filename,Name,Value)
```

Description

`T = parquetread(filename)` reads the Parquet file specified in `filename` into a table or timetable `T`.

`T = parquetread(filename,Name,Value)` reads the Parquet file into a table or timetable with additional options specified by one or more name-value pair arguments.

Examples

Read Parquet File into Table

Get information about a Parquet file, read the data from the file into a table, and then read a subset of the variables into a table.

Create a `ParquetInfo` object for the file `outages.parquet`.

```
info = parquetinfo('outages.parquet')
```

```
info = 
  ParquetInfo with properties:

               Filename: "B:\matlab\toolbox\matlab\demos\outages.parquet"
               FileSize: 44202
           NumRowGroups: 1
        RowGroupHeights: 1468
          VariableNames: [1x6 string]
```

```
          VariableTypes: [1x6 string]
    VariableCompression: [1x6 string]
```

Read data from the file into a table and display the first 10 rows.

```
T = parquetread('outages.parquet');
T(1:10,:)
```

```
ans=10×6 table
      Region            OutageTime          Loss     Customers       RestorationTime
    ___________    ____________________    ______    __________    ___________________

    "SouthWest"    01-Feb-2002 12:18:00    458.98    1.8202e+06    07-Feb-2002 16:50:00
    "SouthEast"    23-Jan-2003 00:49:00    530.14    2.1204e+05                     NaT
    "SouthEast"    07-Feb-2003 21:15:00     289.4    1.4294e+05    17-Feb-2003 08:14:00
    "West"         06-Apr-2004 05:44:00    434.81    3.4037e+05    06-Apr-2004 06:10:00
    "MidWest"      16-Mar-2002 06:18:00    186.44    2.1275e+05    18-Mar-2002 23:23:00
    "West"         18-Jun-2003 02:49:00         0             0    18-Jun-2003 10:54:00
    "West"         20-Jun-2004 14:39:00    231.29           NaN    20-Jun-2004 19:16:00
    "West"         06-Jun-2002 19:28:00    311.86           NaN    07-Jun-2002 00:51:00
    "NorthEast"    16-Jul-2003 16:23:00    239.93         49434    17-Jul-2003 01:12:00
    "MidWest"      27-Sep-2004 11:09:00    286.72         66104    27-Sep-2004 16:37:00
```

Select and import the variables Region, OutageTime, and Cause into a table and display the first 10 rows.

```
SelVarNames = {'Region','OutageTime','Cause'};
T_subset = parquetread('outages.parquet','SelectedVariableNames',SelVarNames);
T_subset(1:10,:)
```

```
ans=10×3 table
      Region            OutageTime               Cause
    ___________    ____________________    __________________

    "SouthWest"    01-Feb-2002 12:18:00    "winter storm"
    "SouthEast"    23-Jan-2003 00:49:00    "winter storm"
    "SouthEast"    07-Feb-2003 21:15:00    "winter storm"
    "West"         06-Apr-2004 05:44:00    "equipment fault"
    "MidWest"      16-Mar-2002 06:18:00    "severe storm"
    "West"         18-Jun-2003 02:49:00    "attack"
    "West"         20-Jun-2004 14:39:00    "equipment fault"
    "West"         06-Jun-2002 19:28:00    "equipment fault"
    "NorthEast"    16-Jul-2003 16:23:00    "fire"
```

```
    "MidWest"      27-Sep-2004 11:09:00    "equipment fault"
```

Read Parquet File into Timetable

Read the data from the file into a `timetable`, and then use timetable functions to determine if the timetable regular and sorted.

Read data from `outages.parquet` into a timetable and display the first `10` rows. Use the second variable `OutageTime` in the data as the time vector for the timetable.

```
TT = parquetread('outages.parquet','RowTimes','OutageTime');
TT(1:10,:)
```

```
ans=10×6 timetable
         OutageTime           Region        Loss     Customers       RestorationTime
    ____________________    ___________    ______    __________    ___________________

    01-Feb-2002 12:18:00    "SouthWest"    458.98    1.8202e+06    07-Feb-2002 16:50:00
    23-Jan-2003 00:49:00    "SouthEast"    530.14    2.1204e+05                     NaT
    07-Feb-2003 21:15:00    "SouthEast"     289.4    1.4294e+05    17-Feb-2003 08:14:00
    06-Apr-2004 05:44:00    "West"         434.81    3.4037e+05    06-Apr-2004 06:10:00
    16-Mar-2002 06:18:00    "MidWest"      186.44    2.1275e+05    18-Mar-2002 23:23:00
    18-Jun-2003 02:49:00    "West"              0             0    18-Jun-2003 10:54:00
    20-Jun-2004 14:39:00    "West"         231.29           NaN    20-Jun-2004 19:16:00
    06-Jun-2002 19:28:00    "West"         311.86           NaN    07-Jun-2002 00:51:00
    16-Jul-2003 16:23:00    "NorthEast"    239.93         49434    17-Jul-2003 01:12:00
    27-Sep-2004 11:09:00    "MidWest"      286.72         66104    27-Sep-2004 16:37:00
```

Determine if the timetable is regular and sorted. A regular timetable has the same time interval between consecutive row times and a sorted timetable has a row time vector is in ascending order.

```
isregular(TT)
```

```
ans = logical
   0
```

```
issorted(TT)
```

```
ans = logical
   0
```

Sort the timetable on its row times using the `sortrows` function and display the first `10` rows of the sorted data.

```
TT = sortrows(TT);
TT(1:10,:)
```

```
ans=10×6 timetable
         OutageTime           Region        Loss     Customers       RestorationTime
    ____________________    ___________    ______    __________    ___________________

    01-Feb-2002 12:18:00    "SouthWest"    458.98    1.8202e+06    07-Feb-2002 16:50:00
    05-Mar-2002 17:53:00    "MidWest"      96.563    2.8666e+05    10-Mar-2002 14:41:00
    16-Mar-2002 06:18:00    "MidWest"      186.44    2.1275e+05    18-Mar-2002 23:23:00
    26-Mar-2002 01:59:00    "MidWest"      388.04    5.6422e+05    28-Mar-2002 19:55:00
    20-Apr-2002 16:46:00    "MidWest"       23141           NaN                     NaT
    08-May-2002 20:34:00    "SouthWest"    50.732         34481    08-May-2002 22:21:00
    18-May-2002 11:04:00    "MidWest"      1389.1    1.3447e+05    21-May-2002 01:22:00
    20-May-2002 10:57:00    "NorthEast"    9116.6    2.4983e+06    21-May-2002 15:22:00
    27-May-2002 09:44:00    "SouthEast"    237.28    1.7101e+05    27-May-2002 16:19:00
    02-Jun-2002 16:11:00    "SouthEast"         0             0    05-Jun-2002 05:55:00
```

Input Arguments

`filename` — Name of Parquet file
character vector | string scalar

Name of Parquet file, specified as a character vector or string scalar. `parquetread` works with Parquet 1.0 or Parquet 2.0 files.

Depending on the location of the file, `filename` can take on one of these forms.

Location	Form
Current folder or folder on the MATLAB path	Specify the name of the file in `filename`. **Example:** `'data.parquet'`

<table>
<tr><th>Location</th><th>Form</th></tr>
<tr><td>File in a folder</td><td>If the file is not in the current folder or in a folder on the MATLAB path, then specify the full or relative path name.

Example: <code>'C:\myFolder\data.parquet'</code>

Example: <code>'myDir\myFile.ext'</code></td></tr>
<tr><td>Remote Location</td><td>If the file is stored at a remote location, then <code>filename</code> must contain the full path of the file specified as a uniform resource locator (URL) of the form:

<code>scheme_name://path_to_file/my_file.ext</code>

Based on your remote location, scheme_name can be one of the values in this table.
<table>
<tr><th>Remote Location</th><th>scheme_name</th></tr>
<tr><td>Amazon S3</td><td>s3</td></tr>
<tr><td>Windows Azure Blob Storage</td><td>wasb, wasbs</td></tr>
<tr><td>HDFS</td><td>hdfs</td></tr>
</table>
For more information, see “Work with Remote Data”.

Example: <code>'s3://bucketname/path_to_file/data.parquet'</code></td></tr>
</table>

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'OutputType','table'` imports the data in the Parquet file as a table.

`OutputType` — Output datatype
`'auto'` (default) | `'table'` | `'timeable'`

Output datatype, specified as the comma-separated pair consisting of `'OutputType'` and `'auto'`, `'table'`, or `'timeable'`.

- `'auto'` — Return a table or a timetable. The `parquetread` detects if the output should be a table or a timetable based on other name-value pairs that you specify. For example, when you set timetable related name-value pairs, then `parquetread` infers that the output is a timetable. Setting these name-value pairs indicates that the output is a timetable: `RowTimes`, `StartTime`, `SampleRate`, or `TimeStep`.
- `'table'` — Return a table. For more information on the table datatype, see `table`.
- `'timetable'` — Return a timetable. For more information on the timetable datatype, see `timetable`.

Example: `'OutputType','timetable'`

Data Types: `char` | `string`

`SelectedVariableNames` — Subset of variables to import
character vector | string scalar | cell array of character vectors | string array

Subset of variables to import, specified as the comma-separated pair consisting of `'SelectedVariableNames'` and a character vector, string scalar, cell array of character vectors, or a string array.

- `SelectedVariableNames` must be a subset of variable names contained in the Parquet file. To get the names of all the variables in the file, use the `VariableNames` property of the `ParquetInfo` object.
- If you do not specify the `SelectedVariableNames` name-value pair, `parquetread` reads all the variables from the file.

Data Types: `char` | `string` | `cell`

`RowTimes` — Row times variable
variable name | time vector

Row times variable, specified as the comma-separated pair consisting of `'RowTimes'` and a variable name or a time vector.

- Variable name must be a character vector or string scalar containing the name of any variable in the input table that contains `datetime` or `duration` values. The variable specified by the variable name provides row time labels for the rows. The remaining variables of the input table become the variables of the timetable.

- Time vector must be a `datetime` vector or a `duration` vector. The number of elements of time vector must equal the number of rows of the input table. The time values in the time vector do not need to be unique, sorted, or regular. All the variables of the input table become variables of the timetable.

Data Types: `char` | `string` | `datetime` | `duration`

`StartTime` — Start time of row times
datetime scalar | duration scalar

Start time of the row times, specified as the comma-separated pair consisting of `StartTime` and a datetime scalar or duration scalar.

- If the start time is a datetime, then the row times of `T` are datetime values.
- If the start time is a duration, then the row times of `T` are duration values.
- If the time step is a calendar duration, then the start time must be a datetime value.

`StartTime` is a timetable related parameter. The `parquetread` function uses `StartTime` along with `SampleRate` or `TimeStep` to define the time vector for the output `T`.

Data Types: `datetime` | `duration`

`SampleRate` — Sample rate
positive numeric scalar

Sample rate, specified as the comma-separated pair consisting of `'SampleRate'` and a positive numeric scalar. The sample rate is the number of samples per second (Hz) of the time vector of the output timetable `T`.

`SampleRate` is a timetable related parameter. The `parquetread` function uses `SampleRate` along with other timetable parameters to define the time vector for the output `T`.

Data Types: `double`

`TimeStep` — Time step of time vector
duration | calendarDuration

Time step of time vector, specified as the comma-separated pair consisting of `'TimeStep'` and a duration scalar.

- If you specify the time step as a calendar duration (for example, calendar months), then the vector of row times must be a datetime vector.
- If you specify the time step as a duration (for example, seconds), then the vector of row times can either be a datetime or duration vector.

`TimeStep` is a timetable related parameter. The `parquetread` function uses `TimeStep` along with other timetable parameters to define the time vector for the output `T`.

`PreserveVariableNames` — Flag to preserve variable names
`false` (default) | `true`

Flag to preserve variable names, specified as the comma-separated pair consisting of `PreserveVariableNames` and either `true`, or `false`.

- `true` — Preserve variable names that are not valid MATLAB identifiers such as variable names that include spaces and non-ASCII characters.
- `false` — Convert invalid variable names (as determined by the `isvarname` function) to valid MATLAB identifiers.

Starting in R2019b, variable names and row names can include any characters, including spaces and non-ASCII characters. Also, they can start with any characters, not just letters. Variable and row names do not have to be valid MATLAB identifiers (as determined by the `isvarname` function). To preserve these variable names and row names, set `PreserveVariableNames` to `true`.

Output Arguments

`T` — Output data
table | timetable

Output data, returned as a table or timetable. The output of the `parquetread` function depends on the value of the `OutputType` name-value pair. For more information, see the name-value pair description for `OutputType`.

Limitations

In some cases, `parquetwrite` creates files that do not represent the original array `T` exactly. If you use `parquetread` or `datastore` to read the files, then the result might

not have the same format or contents as the original table. For more information, see “Apache Parquet Data Type Mappings”.

See Also

`parquetDatastore` | `parquetInfo` | `parquetwrite` | `table` | `timetable`

Introduced in R2019a

parquetwrite

Write columnar data to Parquet file

Syntax

```
parquetwrite(filename,T)
parquetwrite(filename,T,'VariableCompression',VariableCompression)
```

Description

`parquetwrite(filename,T)` writes a table or timetable `T` to a Parquet 2.0 file with the filename specified in `filename`.

`parquetwrite(filename,T,'VariableCompression',VariableCompression)` specifies the compression schemes to use when writing variables to the output file. The Parquet file format enables you to specify the compression schemes on a per-variable(column) level allowing very efficient compression and encoding of data.

Examples

Write Table or Timetable to Parquet File

Write tabular data into a Parquet file and compare the size of the same tabular data in `.csv` and `.parquet` file formats.

Read the tabular data from the file `outages.csv` into a table.

```
T = readtable('outages.csv');
```

Write the data to Parquet file format. By default, the `parquetwrite` function uses the `Snappy` compression scheme. To specify other compression schemes see `'VariableCompression'` name-value pair.

```
parquetwrite('outagesDefault.parquet',T)
```

Get the file sizes and compute the ratio of the size of tabular data in the `.csv` format to size of the same data in `.parquet` format.

Get size of `.csv` file.

```
fcsv = dir(which('outages.csv'));
size_csv = fcsv.bytes
```

```
size_csv = 101040
```

Get size of `.parquet` file.

```
fparquet  = dir('outagesDefault.parquet');
size_parquet = fparquet.bytes
```

```
size_parquet = 44881
```

Compute the ratio.

```
sizeRatio = ( size_parquet/size_csv )*100 ;
disp(['Size Ratio = ', num2str(sizeRatio) '% of original size'])
```

```
Size Ratio = 44.419% of original size
```

Input Arguments

`filename` — Name of output Parquet file
character vector | string scalar

Name of output Parquet file, specified as a character vector or string scalar.

Depending on the location you are writing to, `filename` can take on one of these forms.

Location	Form
Current folder	To write to the current folder, specify the name of the file in `filename`. **Example:** `'myData.parquet'`

Location	Form
Other folders	To write to a folder different from the current folder, specify the full or relative path name in `filename`. **Example:** `'C:\myFolder\myData.parquet'` **Example:** `'dataDir\myData.parquet'`
Remote Location	To write to a remote location, `filename` must contain the full path of the file specified as a uniform resource locator (URL) of the form: *`scheme_name`*`://path_to_file/myData.parquet` Based on your remote location, *`scheme_name`* can be one of the values in this table. **Remote Location** — **`scheme_name`** Amazon S3 — `s3` Windows Azure Blob Storage — `wasb`, `wasbs` HDFS — `hdfs` For more information, see “Work with Remote Data”. **Example:** `'s3://bucketname/path_to_file/myData.parquet'`

Data Types: `char` | `string`

T — Input data
table | timetable

Input data, specified as a table or timetable.

`VariableCompression` — Compression scheme names
`'Snappy'` (default) | `'Brotli'` | `'Gzip'` | `'Uncompressed'` | cell array of character vectors | string vector

Compression scheme names, specified as one of these values:

- `'Snappy'`, `'Brotli'`, `'Gzip'`, or `'Uncompressed'`. If you specify one compression algorithm then `parquetwrite` compresses all variables using the same algorithm.
- Alternatively, you can specify a cell array of character vectors or a string vector containing the names of the compression algorithms to use for each variable.

In general, `'Snappy'` has better performance for reading and writing, `'Gzip'` has a higher compression ratio at the cost of more CPU processing time, and `'Brotli'` typically produces the smallest file size at the cost of compression speed.

`parquetwrite` writes Parquet 2.0 files using the Parquet dictionary encoding scheme. This encoding scheme is most efficient when the number of unique values is not too large. If the size of the dictionary or number of unique values grows to be too big, then the encoding automatically falls back to plain encoding.

Example: `parquetwrite('myData.parquet', T, 'VariableCompression', 'Brotli')`

Example: `parquetwrite('myData.parquet', T, 'VariableCompression', {'Brotli' 'Snappy' 'Gzip'})`

Limitations

In some cases, `parquetwrite` creates files that do not represent the original array `T` exactly. If you use `parquetread` or `datastore` to read the files, then the result might not have the same format or contents as the original table. For more information, see “Apache Parquet Data Type Mappings”.

See Also

`parquetDatastore` | `parquetInfo` | `parquetread`

Introduced in R2019a

matlab.net.http.ProgressMonitor class

Package: `matlab.net.http`
Superclasses: `handle`

Progress monitor for HTTP message exchange

Description

Use the `ProgressMonitor` class to implement a progress monitor for an HTTP request message. A progress monitor listens to changes in the properties of this class to implement a display or update of your choice.

To implement a progress monitor, create a subclass of this class. Then create an `HTTPOptions` object, specify that subclass as the `ProgressMonitorFcn` property, and set the `UseProgressMonitor` property to `true`.

An HTTP request starts when you call the `RequestMessage.send` method. The process might involve multiple messages in both directions in the case of redirects and authentication. MATLAB calls the `done` method when all transfers have been completed.

During a transfer:

- MATLAB sets the `Max`, `CancelFcn`, and `Direction` properties when you call `RequestMessage.send`.
- MATLAB sets the `Value` property repeatedly as the body of the request message is sent to indicate the number of bytes transferred.
- When receipt of the `ResponseMessage` begins, MATLAB sets `Direction` to `Response` and again sets `Value` repeatedly.
- To cancel the transfer at any time, call the `CancelFcn` function. This action is the same as interrupting the `send` function in the Command Window.

You can display a graphical progress indicator or other indication of progress after `Direction` changes to `Response`. For each subsequent `Value`, update the indicator to the current `Value`. You can also use this mechanism to monitor progress programmatically.

Construction

`obj = matlab.net.http.ProgressMonitor(Name,Value)` creates a progress monitor with additional properties specified by one, or more name-value pair arguments. `Name` is the property name and `Value` is the corresponding value. You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Unspecified properties are set to their default values.

Properties

`Interval` — Seconds to first call and maximum interval between calls

2 seconds (default) | double

Seconds to first call and maximum interval between calls, specified as double.

`Interval` is the amount of time in seconds after the start of transfer before the first setting of the `Value` property. `Interval` is a suggested maximum amount of time between settings of `Value`, regardless of progress. If the total time to transfer the data is less than `Interval` seconds, then `Value` is not set. If no data has been transferred in `Interval` seconds since the last setting of `Value`, then `Value` might be set again to the same value. In this way, your `ProgressMonitor` object can cancel a transfer (by calling `CancelFcn`) even if there is no progress.

The `Interval` value is a suggested value. There is no guarantee that MATLAB sets `Value` within `Interval` seconds if there has been no progress.

The default interval is two seconds. To specify a different value, set `Interval` in your constructor. The minimum interval between consecutive settings of `Value` when there is no progress is 0.1 seconds. However, if `Value` changes, it might be set more often than this interval.

After setting `Value` the first time, there is no delay in setting `Value` for subsequent messages in the same exchange.

`InUse` — Monitor is in use indicator (read-only)

`true` | `false`

Monitor is in use indicator, specified as `true` or `false`. MATLAB sets this property to indicate whether it is using this `ProgressMonitor` object during a transfer. `InUse` prevents you from reusing this object for more than one transfer at a time.

Data Types: `logical`

`CancelFcn` — Function to call to cancel transfer (read-only)
function handle

Function to call to cancel a transfer, specified as a function handle. MATLAB sets `CancelFcn` to the function your `ProgressMonitor` should call to cancel a transfer. Calling this function has the same effect as interrupting the transfer in the Command Window.

`Max` — Maximum length of the transfer (read-only)
`uint64` | `[]`

Maximum length of the transfer, specified as `uint64`. If the maximum length is not known, `Max` is `[]`. `Max` is the maximum value for your displayed progress indicator.

MATLAB sets `Max` at the beginning of each send and receive operation to the expected number of bytes to be transferred, based on the Content-Length header field.

If the message does not contain a Content-Length field, `Max` is `[]`. In this case, you cannot determine the proportion of the transfer that has been completed. You can, however, still monitor changes in the `Value` property.

`Direction` — Direction of transfer
`matlab.net.http.MessageType` object | empty

Direction of transfer, specified as a `matlab.net.http.MessageType` object. MATLAB sets `Direction` to indicate whether progress is monitored for a request message or a response message. If no transfer is taking place, then `Direction` is empty.

`Value` — Number of bytes transferred so far
`uint64` | empty

Number of bytes transferred so far, specified as `uint64`. MATLAB sets `Value` repeatedly to the total number of bytes transferred for the current message. However, it delays setting this property the first time in an exchange until at least `Interval` seconds have elapsed since the start of the message.

Implement a `set.Value` method for this property to monitor progress of the transfer. You can also implement the ability to cancel the transfer from within the progress monitor in the `set.Value` method.

MATLAB might set `Value` to empty at the end of a given transfer, to indicate that transfer in the current direction has ended. MATLAB always sets `Value` to empty at the conclusion of all transfers before calling the `done` method.

You cannot control the frequency at which MATLAB updates the `Value` property. However, MATLAB might set `Value` at least once every `Interval` seconds, even if there is no progress. This behavior allows you to call the `CancelFcn` if a transfer is not progressing. `Value` might be zero if no bytes have been transferred for `Interval` seconds since transfer began.

Methods

done	Indicate all message transfers completed

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

See Also

Topics

"Display Progress Monitor for HTTP Message"

Introduced in R2016b

done

Class: `matlab.net.http.ProgressMonitor`
Package: `matlab.net.http`

Indicate all message transfers completed

Syntax

```
done(obj)
```

Description

`done(obj)` indicates that all message transfers are completed. MATLAB calls this method when all transfers for a given `RequestMessage.send` operation have been completed. This method indicates that MATLAB is no longer using the `ProgressMonitor` object. Use this method to delete windows or other objects you create to display progress.

See Also

Topics

"Display Progress Monitor for HTTP Message"

Introduced in R2016b

matlab.net.http.ProtocolVersion class

Package: `matlab.net.http`

HTTP protocol version

Description

If you use `'HTTP/1.1'`, you do not need to create a protocol version. To use a different protocol, use the `ProtocolVersion` class to create the protocol version and add it to the request line of a request message.

The server returns a `ProtocolVersion` object in the status line of a response message.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Description

`obj = matlab.net.http.ProtocolVersion(name,major,minor)` creates the protocol version with the specified properties on page 1-6814. This constructor enforces proper syntax of the parameters. For test purposes, create a `ProtocolVersion` object in a request message. Then use the `isequal` method to compare this value with the value returned by the server in the response message.

`obj = matlab.net.http.ProtocolVersion(str)` creates the version from `str`.

Input Arguments

`str` — Protocol version
string | character vector

Protocol version, specified as a string or a character vector acceptable to the `ProtocolVersion` constructor. `str` should have the syntax of name plus major and minor version numbers, but no error occurs if it does not.

Properties

`Name` — Protocol name

`HTTP` (default) | string | character vector

Protocol name, specified as a string or character vector.

Example: `'HTTP'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

`Major` — Major version number

1 (default) | integer 0–9

Major version number, specified as an integer value from 0 through 9.

Example: 1

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

`Minor` — Minor version number

1 (default) | integer 0–9

Minor version number, specified as an integer value from 0 through 9.

Example: 1

Attributes:

GetAccess	public
SetAccess	public
Dependent	true

Methods

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

eq, isequal	true if protocol versions are functionally equivalent. The comparison ignores the case of the Name property and uses numeric comparisons for the Major and Minor properties.
string	Protocol version as string
char	Protocol version as character vector

Examples

Display Protocol Version in Response Message

Send an HTTP request message to mathworks.com using default values. Display the protocol version in the response message.

```
request = matlab.net.http.RequestMessage;
uri = matlab.net.URI('https://www.mathworks.com');
response = send(request,uri);
version = string(response.StatusLine.ProtocolVersion)
```

```
version =
```

```
"HTTP/1.1"
```

See Also

RequestLine | RequestMessage | ResponseMessage | StatusLine

Introduced in R2016b

matlab.net.http.RequestLine class

Package: `matlab.net.http`
Superclasses: `matlab.net.http.StartLine`

First line of HTTP request message

Description

The `RequestLine` class represents the first line of a request message. A request line is automatically created when you send or complete a message. You also can create a request line and pass it to the `RequestMessage.send` method in place of the URI argument.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Description

`obj = matlab.net.http.RequestLine(method,requestTarget,protocolVersion)` creates a request line with the specified properties on page 1-6818. You can omit trailing arguments and use `[]` to specify any placeholders.

`obj = matlab.net.http.RequestLine(str)` creates a request line by parsing `str`.

Input Arguments

`str` — Request line
string | character vector

Request line, specified as a string or a character vector. `str` consists of 1-3 parts, separated by white space, specifying the `Method`, `RequestTarget`, and `ProtocolVersion` properties.

Properties

`Method` — Request method

`[]` (default) | `matlab.net.http.RequestMethod` enumeration | string | character vector

Request method, specified as a `matlab.net.http.RequestMethod` enumeration or a string or character vector representing a request method. To send a message, set the `RequestMessage.Method` property or the `RequestLine.Method` property.

Example: `'GET'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

`ProtocolVersion` — Protocol version

`[]` (default) | `matlab.net.http.ProtocolVersion` object | string

Protocol version, specified as a `matlab.net.http.ProtocolVersion` object or as a string acceptable to the constructor.

Example: `'HTTP/1.1'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

`RequestTarget` — Target URI

`matlab.net.URI` object | string | character vector

Target URI, specified as a `matlab.net.URI` object, or as a string or a character vector acceptable to the constructor. The `RequestTarget` value must be consistent with the URI you specify in a request message `send` or `complete` method.

`RequestTarget` contains at least a leading forward slash (`/`) for the `Path` property, even if the path is relative or empty.

When using a proxy, `RequestTarget` is a full URI with `Scheme` and `Authority` properties. When not using a proxy, `RequestTarget` must be an absolute path with an optional `Query` property. You specify a proxy in an `HTTPOptions` object.

Example: `'http://heritage.stsci.edu/2007/14/images/p0714aa.jpg'`

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`
`Dependent`	`true`

Methods

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`char`	Request line as character vector
`string`	Request line as string

Examples

Create Request Line for Hubble Heritage Project Image

Create an HTTP request line to get an image from the Hubble Heritage Project website.

```
import matlab.net.http.RequestLine
url = 'http://heritage.stsci.edu/2007/14/images/p0714aa.jpg';
reqline = RequestLine('get',url,'HTTP/1.1');
string(reqline)
```

```
ans =
"GET http://heritage.stsci.edu/2007/14/images/p0714aa.jpg HTTP/1.1"
```

See Also

`complete` | `matlab.net.URI` | `matlab.net.http.RequestMethod` | `send`

Introduced in R2016b

matlab.net.http.RequestMessage class

Package: `matlab.net.http`
Superclasses: `matlab.net.http.Message`

HTTP request message

Description

Use the `RequestMessage` class to format HTTP request messages to send to a server for processing. Use the `send` method to send the message, or the `complete` method to validate the message before sending. These methods fill in any necessary header fields and other message properties.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Description

`obj = matlab.net.http.RequestMessage` creates a request message with default values. When you send or complete a message, the default `Method` property is `RequestMethod.GET`.

`obj = matlab.net.http.RequestMessage(method,header,body)` specifies one or more optional message properties on page 1-6822. You can omit trailing arguments and use `[]` to specify any placeholders.

`obj = matlab.net.http.RequestMessage(requestLine,header,body)` sets the `RequestLine` property to `requestLine`. Use this syntax if you need control over the contents of the request line. For example, to send a message explicitly to a proxy, set the `RequestLine.RequestTarget` property to the full URI. Otherwise, MATLAB chooses

the proxy based on your proxy settings, and the `send` method sets the `RequestTarget` to the `Path` property of the URI.

`obj = matlab.net.http.RequestMessage(requestLine,header,provider)` gets the message body data from a `matlab.net.http.io.ContentProvider`.

Properties

`RequestLine` — Request line
`matlab.net.http.RequestLine` object | string | character vector

Request line, specified as a `matlab.net.http.RequestLine` object, or a string or a character vector that contains the method, target, and protocol version. This line is automatically created when you send a message, based on the method and URI you specify. If you set this property explicitly, then its contents are used as the request line. The value might be set to a `RequestLine` object or to a string which is parsed and converted to a `RequestLine` object.

Example: `'GET HTTP/1.1'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

`Method` — Request method
`[]` (default) | `matlab.net.http.RequestMethod` enumeration | string | character vector

Request method, specified as a `matlab.net.http.RequestMethod` enumeration or a string or character vector representing a request method. To send a message, set the `RequestMessage.Method` property or the `RequestLine.Method` property.

Example: `'GET'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

Header — Message header

[] (default) | matlab.net.http.HeaderField object

Message header, specified as a matlab.net.http.HeaderField object or a vector of HeaderField objects. When you set the Header property, MATLAB checks the fields of the header to ensure that they are appropriate for the message type. The RequestMessage send and complete methods fill in any required header fields for a properly formed request.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Body — Message body

[] (default) | matlab.net.http.MessageBody object | matlab.net.http.io.ContentProvider

Message body, specified as a matlab.net.http.MessageBody object, matlab.net.http.io.ContentProvider, or data acceptable to the MessageBody constructor. By default, Body is empty (set to []). A request message containing a Body property normally uses a method such as 'PUT' or 'POST', not the default value 'GET', but this convention is not enforced.

In a completed or received message, if the message has a ContentTypeField header field, then the MessageBody.ContentType property is set to that value. Otherwise, ContentType is unchanged or empty.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Completed — Whether message is complete

false (default) | true

Whether message is complete, specified as true or false. A true value means that the message was completed.

Methods that validate messages (RequestMessage.send and RequestMessage.complete) set the Completed property to true after:

If the property is true, then these methods do not modify the message, and the `send` method sends the message without checking it for validity. Any change to this message after that changes the `Completed` property back to `false`.

To send arbitrary headers and data in a request message, set `Completed` to `true` to prevent the `send` method from modifying the message. You can still use the `complete` method to validate the message, but the `send` method sends it whether it is valid.

If a request message contains data (the `Body.Data` property is not empty), then `Completed` is set to `true` only if `Body.Payload` contains the raw data. In a response message, the payload is set only if you specify the `HTTPOptions.SavePayload` property.

Attributes:

GetAccess	public
SetAccess	public
Transient	true

Data Types: `logical`

`StartLine` — Message start line
`matlab.net.http.StartLine`

Message start line, specified as a `matlab.net.http.StartLine` object.

Attributes:

GetAccess	public
SetAccess	public

Methods

Public Methods

complete	Validate and complete HTTP request message without sending
send	Send HTTP request message and receive response
addFields	Add fields to message header
changeFields	Change existing fields in message header
getFields	Return message header fields matching name or class
removeFields	Remove fields from message header
replaceFields	Change values in or add fields to message header

show Display or return formatted version of message

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`string`	Message `Header`, `Body` and `StartLine` properties, as string. For multiple messages, returns a string array. If `Body` contains binary data that cannot be converted to characters, then the method displays a message indicating the length of the data in bytes. The string is an approximate representation of what the message looks like when sent or received. Use for logging, diagnostics, or debugging. For a formatted version of messages, use `show`.
`char`	Same as `string`, except returns message as character vector.
`isequal`	Returns `true` if the visible public properties of all messages in the two message arrays are equal.

Examples

Create HTTP Request Message

Format an HTTP message requesting a server to add text to a website. This example only formats the message and does not send the data.

Add content to the message body.

```
data = 'Data to send';
body = matlab.net.http.MessageBody(data);
body.show
```

```
Data to send
```

Create a Content-Type header field describing the data type of the body.

```
contentTypeField = matlab.net.http.field.ContentTypeField('text/plain');
```

Create an Accept header field specifying the data types acceptable in the response message.

```
type1 = matlab.net.http.MediaType('text/*');
type2 = matlab.net.http.MediaType('application/json','q','.5');
acceptField = matlab.net.http.field.AcceptField([type1 type2]);
```

Create a request header containing the two header fields.

```
header = [acceptField contentTypeField];
```

Specify that this message is a PUT request.

```
method = matlab.net.http.RequestMethod.PUT;
```

Create the request message and display the contents.

```
request = matlab.net.http.RequestMessage(method,header,body);
show(request)
```

```
PUT
Accept: text/*, application/json; q=.5
Content-Type: text/plain

Data to send
```

See Also

`MessageBody` | `RequestLine` | `RequestMethod` | `ResponseMessage` | `matlab.net.URI` | `webread`

Introduced in R2016b

complete

Class: `matlab.net.http.RequestMessage`
Package: `matlab.net.http`

Validate and complete HTTP request message without sending

Syntax

```
[completedrequest,target] = complete(request,uri)
[completedrequest,target] = complete(request,uri,options)
```

Description

`[completedrequest,target] = complete(request,uri)` adds and validates message header fields and converts data like the `RequestMessage.send` method, but does not send the message. `complete` assumes a default `HTTPOptions` object to determine how to complete and validate the request.

Use the `complete` method to examine the contents of a request message for debugging purposes.

To fill in and validate the `Header` and `RequestLine` properties, this method ignores the `Completed` property in `request`. The method always returns a modified `completedrequest`. If `request` is not completed, then the method errors. You can use this behavior to determine whether a manually completed request is valid.

If `Completed` is not set, then this method always converts `Data` in `request.Body` and stores the result in `completedrequest.Body.Payload`, overwriting any previous contents of `Payload`. This means that both `Data` and `Payload` in `completedrequest.Body` contain values. This is different from the behavior of the `send` which does not save the `Payload` unless `HTTPOptions.SavePayload` is set. If the message contains a large amount of data, then memory usage and conversion time might be a factor.

However, if `request.Body` contains a `ContentProvider`, then `complete` does not call the provider to create data. `completedrequest.Body` contains the same `ContentProvider`.

`[completedrequest,target] = complete(request,uri,options)` provides additional options for validating and completing the request message.

If you intend to send `completedrequest` to avoid the cost of a repeat validation, send it to `target` instead of `uri`, using the same `options`. Time-dependent header fields such as `Date` which are added by the `send` method, are not updated when sent again using `completedrequest`.

Input Arguments

`request` — Request message
`matlab.net.http.RequestMessage` object

Request message, specified as a `matlab.net.http.RequestMessage` object.

`uri` — Message destination
`matlab.net.URI` object | string | character vector

Message destination, specified as a `matlab.net.URI` object or a string or character vector acceptable to the constructor.

`options` — Additional options
`matlab.net.http.HTTPOptions` object

Additional options, specified as a `matlab.net.http.HTTPOptions` object, for processing request and response messages.

Output Arguments

`completedrequest` — Completed and validated request
`matlab.net.http.RequestMessage` object

Completed and validated request, returned as a `matlab.net.http.RequestMessage` object. The `Completed` property is true.

target — Completed URI
matlab.net.URI object

Completed URI, returned as a matlab.net.URI object.

Examples

Validate Request Message

Create a request message for a fictional website. Then, validate and complete the request without sending it.

```
request = matlab.net.http.RequestMessage();
url = 'myschool.edu/campus.jpg';
options = matlab.net.http.HTTPOptions('SavePayload',true);
[request,url] = complete(request,url,options);
show(request)
```

```
GET /campus.jpg HTTP/1.1
Host: myschool.edu
User-Agent: MATLAB/9.0.0.366741 (R2016b)
Date: Wed, 13 Jul 2016 17:21:08 GMT
Connection: close
```

MATLAB displays User-Agent and Date values relevant to your system.

Show the updated URL.

```
string(url)
```

```
ans = http://myschool.edu/campus.jpg
```

Limitations

- A completed request does not add any authorization header fields that might be needed for authentication to a server or proxy, even if the Authenticate property is set in options. It might not be possible to determine what the server requires without sending the message. To see what was sent in an authentication exchange, examine the completedrequest or history arguments returned by the send method.

Tips

- To send the same request message repeatedly, send `completedrequest`. Otherwise, if you send `request`, then MATLAB repeatedly validates the message. Also be sure to specify `target` as the URI and the same `options` input argument. Time-dependent header fields such as Date, which the `send` method adds, are not updated when sending `completedrequest`.
- To complete a message without converting the data, set the `Completed` property to `true` before calling the `complete` method. If `Completed` is true and `request.Body` is a `MessageBody` object, then the `complete` method assumes that the current value of `request.Body.Payload` is the desired one, even if it is empty.

  This behavior differs from the `send` method. If `request.Body.Payload` is empty, then `send` converts and sends nonempty `Body.Data` values, even if `Completed` is true.

See Also

`HTTPOptions` | `MessageBody` | `matlab.net.URI` | `send`

Introduced in R2016b

send

Class: `matlab.net.http.RequestMessage`
Package: `matlab.net.http`

Send HTTP request message and receive response

Syntax

```
[response,completedrequest,history] = send(request,uri)
[response,completedrequest,history] = send(request,uri,options,
consumer)
```

Description

`[response,completedrequest,history] = send(request,uri)` sends the `request` message to the web service specified by `uri` and returns the `response`, if any. If no `request.Method` property is specified, then the `send` method sets the property to `'GET'`.

By default, `send` verifies the semantic correctness of the headers and other parts of the message and completes the `uri`. The method also fills in any required header fields for a properly formed request. If `request.Body` is a `MessageBody` whose `Payload` property is not already set, then `send` calls appropriate conversion functions to convert any `request.Body.Data` to a vector of bytes representing an HTTP payload to be sent, as described for `MessageBody.Data`. Normally, a `'GET'` request does not contain data, but the method sends the `Body` regardless of the `RequestMethod`. If the server returns data in its response and no `consumer` is specified, then `send` converts that data to MATLAB data and saves it in `response.Body.Data`. See `MessageBody.Data` for more information on data conversion.

If `request.Body` is a `ContentProvider`, then MATLAB calls the provider to get the data to be sent.

If the header already contains a field that the method normally adds, then `send` verifies that the field has the expected value. You can override the default behavior as follows.

- To send a message as is without any checking or alteration of the header, set the `request.Completed` property to `true` before sending. If you used the `complete` method to complete the request, then you should specify the same value of `uri` and `options` that you provided to `complete`, or there might be unpredictable results. Even if `Completed` is set, unspecified fields in the `RequestLine` will be filled in with default values.
- To allow the `send` method to check and alter the header, but suppress adding a particular header field that `send` or a `ContentProvider` might add, add that field to `request.Header` with an empty value (`[]`). For example, `send` automatically adds a User-Agent header field. If you do not want this behavior, then add `HeaderField('User-Agent')` to the header. Header fields with empty values are not included in the message. The Host and Connection fields cannot be suppressed.
- To override the value that the `send` method adds for a given header field, add your own instance of that field before sending or completing the message. However, this will not override a header field that a `ContentProvider` might add. However, for some header field types, `send` might still reject the message if the value is not valid. To prevent any checking of the value of a given field, or to override a field that a `ContentProvider` adds, add a field of type `matlab.http.field.GenericField` to the header with the desired name and value. Neither `send` nor a `ContentProvider` will add any header fields with names equal to any `GenericField` headers and will not check their correctness.
- To send raw binary data without conversion, you can insert a `uint8` vector into either `Body.Data` or `Body.Payload`. The only difference is that data in `Body.Data` is subject to conversion based on the Content-Type field in the message, while `Body.Payload` is not. `send` always tries to convert nonempty `Body.Data` if `Body.Payload` is empty, even if `Completed` is already set. See `MessageBody.Data` for conversion rules.

`[response,completedrequest,history] = send(request,uri,options, consumer)` provides additional options for processing the `request` and `response` messages.

Input Arguments

`request` — Request message
`matlab.net.http.RequestMessage` object

Request message, specified as a `matlab.net.http.RequestMessage` object.

`uri` — Message destination
`matlab.net.URI` object | string | character vector

Message destination, specified as a `matlab.net.URI` object or a string or character vector acceptable to the constructor. If the value is a `URI` object, then it must name a Host. If it is a string and it does not include a Scheme, then `'http'` is assumed. For example, `'www.somewebsite.com'` and `'//www.somewebsite.com'` are both treated as `'http://www.somewebsite.com'`.

`options` — Additional options
`matlab.net.http.HTTPOptions` object

Additional options, specified as a `matlab.net.http.HTTPOptions` object, for processing request and response messages. If not specified, or if the value is empty, then `send` uses default options.

`consumer` — Content consumer
`matlab.net.http.io.ContentConsumer` object | function handle

Content consumer to process the returned payload, specified as a `matlab.net.http.io.ContentConsumer` object or a handle to a function that returns a `ContentConsumer`.

The `send` method calls the `ContentConsumer` to process or store buffers of data in real time as the data is being received. The `consumer` can store the data in `response.Body.Data` or handle it in some other way. For example, a consumer can display the data in a figure window or save it in a file. When a consumer is specified, MATLAB does not automatically set `MessageBody.Data`, but it will set `MessageBody.Payload` to the unconverted payload if `options.SavePayload` is true. For example, a `FileConsumer` saves the data to a file, not in `MessageBody.Data`.

Using a `ContentConsumer` provides more flexibility in converting or storing the response data than the default MATLAB response data conversion. For a description of the default conversion of received data, see `MessageBody.Data`. For a list of `ContentConsumer` types provided by MATLAB, type:

```
mp = ?matlab.net.http.io.ContentConsumer;
{mp.ContainingPackage.ClassList.Name}'
```

In addition, software developers can create their own `ContentConsumer` subclasses to process data as it is being received.

The `consumer` is used only if it accepts the message, based on various factors such as the Content-Type header in the `response` and whether `response.StatusCode` is `OK`. Each `consumer` has its own criteria for accepting a message.

If the payload is compressed with a supported encoding, and `options` is unspecified or `options.DecodePayload` is true, then the consumer gets the decompressed data. If payload is compressed and `options.DecodePayload` is false, or the payload is compressed with an unsupported encoding, then the consumer is not used and there is no default processing of the data.

In all cases where the consumer is not used, the payload is processed and converted as if no `consumer` was specified.

If `consumer` is a function handle, the function is called to instantiate a consumer only after MATLAB determines that the response has a payload.

When specifying `consumer` but no `options`, add a placeholder `[]` argument for `options` to use default options.

Output Arguments

`response` — Message received from server
`matlab.net.http.ResponseMessage` object

Message received from a server, returned as a `matlab.net.http.ResponseMessage` object. There might be intermediate requests and responses exchanged between MATLAB and the proxy or server if redirections and/or authentications are involved.

`completedrequest` — Request that was sent
`matlab.net.http.RequestMessage` object

Request that was sent before receiving the `response` argument, returned as a `matlab.net.http.RequestMessage` object. The `send` method augments the `completedrequest` argument with authentication or redirection information.

If `request.Body` is a `ContentProvider`, then `completedrequest.Body` is normally empty because `ContentProvider` payloads are not saved. However, if `options.SavePayload` is true, then the `completedrequest.Body` is a `MessageBody` whose `Payload` has the data sent from the provider as a `uint8` vector. In some cases, when the Content-Type of the request indicates that it is character-based, the `MessageBody.Data` property contains the payload represented as a string.

After sending an HTTP request, examine the `completedrequest` argument to see what was sent. The server might send multiple messages, for example, if there were redirections or an authentication exchange occurred. If there are multiple messages, then `completedrequest` contains the last request. To see the first, or intermediate messages, look at the `history` argument.

To send the same request multiple times, call the `RequestMessage.complete` method:

```
[completedrequest,target] = complete(request,uri)
```

Then, call the `send` method with these output arguments:

```
resp = send(completedrequest,target)
```

`history` — Log of messages
vector of `matlab.net.http.LogRecord` objects

Log of messages, returned as a vector of `matlab.net.http.LogRecord` objects that were exchanged to satisfy this `send` request. If you have a single request and response, then the `history` argument contains one record. In the case of an authentication containing multiple messages, the history can contain multiple log records for each redirection.

Use the history to obtain all Set-Cookie headers from response messages. You can send these headers back to the server in subsequent requests.

The last record in the history contains the same properties as the `completedrequest` and `response` arguments, except for the `Body` property. To log message bodies, specify the `SavePayload` property in the `options` argument.

The history also can be useful for debugging.

Examples

Send HTTP Message

Send an HTTP message to read the MathWorks Contact Support web page and display the message status code.

```
import matlab.net.*
import matlab.net.http.*
```

```
r = RequestMessage;
uri = URI('https://www.mathworks.com/support/contact_us');
resp = send(r,uri);
status = resp.StatusCode
```

```
status =

    OK
```

Do Not Allow HTTP Message Redirects

Prevent message redirects from `mathworks.com` website by setting the HTTP option `MaxRedirects` to zero. Then display status code information.

```
import matlab.net.*
import matlab.net.http.*
r = RequestMessage;
uri = URI('https://www.mathworks.com/support/contact_us');
options = HTTPOptions('MaxRedirects',0);
[resp,~,hist] = send(r,uri,options);
status = getReasonPhrase(resp.StatusCode)
```

```
status =

    'Moved Permanently'
```

Error Handling

Always check the `response` `Status` property to determine whether the request was accepted. Error conditions are:

- `MException` — Message is not well formed and cannot be completed.
- `HTTPException` — Message is completed, but the web service is unreachable or does not respond within the timeout period specified in `options`.

- `Status` property of `response` — Web service responds and returns an HTTP error status. `send` returns normally, setting the `Status` property to the error returned from the server.

See Also

`HTTPException` | `HTTPOptions` | `LogRecord` | `complete` | `matlab.net.URI` | `matlab.net.http.field.GenericField` | `matlab.net.http.io.ContentConsumer` | `matlab.net.http.io.ContentProvider`

Introduced in R2016b

matlab.net.http.RequestMethod class

Package: `matlab.net.http`

HTTP request method

Description

The `RequestMethod` enumeration class provides identifiers for request methods. Use these values for the `Method` property in a request message. The methods supported are those listed in the IANA Hypertext Transfer Protocol (HTTP) Method Registry as of April 14, 2017.

To see a list of all the `RequestMethod` members, type:

```
enumeration matlab.net.http.RequestMethod
```

Examples

Display Method for Default Request Message

```
r = matlab.net.http.RequestMessage;
uri = 'https://www.mathworks.com';
r = complete(r,uri);
reqmethod = r.Method
```

```
reqmethod =
GET
```

See Also

`matlab.net.http.RequestMessage`

External Websites

Hypertext Transfer Protocol (HTTP) Method Registry

Introduced in R2016b

matlab.net.http.ResponseMessage class

Package: `matlab.net.http`
Superclasses: `matlab.net.http.Message`

HTTP response message

Description

The `ResponseMessage` class formats HTTP response messages returned by a server.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

The `RequestMessage.send` method creates a response message.

Properties

`StatusLine` — Status line from server
`StatusLine` object

Status line from server, specified as a `matlab.net.http.StatusLine` object.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`
`Dependent`	`true`

`StatusCode` — HTTP status code
`StatusCode` object

HTTP status code, specified as a `matlab.net.http.StatusCode` object. Same as the `StatusLine.StatusCode` property.

Attributes:

```
GetAccess                           public
SetAccess                           immutable
Dependent                           true
```

`Header` — Message header

`[]` (default) | `matlab.net.http.HeaderField` object

Message header, specified as a `matlab.net.http.HeaderField` object or a vector of `HeaderField` objects. When you set the `Header` property, MATLAB checks the fields of the header to ensure that they are appropriate for the message type. The `RequestMessage` `send` and `complete` methods fill in any required header fields for a properly formed request.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Body` — Message body

`[]` (default) | `matlab.net.http.MessageBody` object | `matlab.net.http.io.ContentProvider`

Message body, specified as a `matlab.net.http.MessageBody` object, `matlab.net.http.io.ContentProvider`, or data acceptable to the `MessageBody` constructor. By default, `Body` is empty (set to `[]`). A request message containing a `Body` property normally uses a method such as `'PUT'` or `'POST'`, not the default value `'GET'`, but this convention is not enforced.

In a completed or received message, if the message has a `ContentTypeField` header field, then the `MessageBody.ContentType` property is set to that value. Otherwise, `ContentType` is unchanged or empty.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Completed` — Whether message is completed

`false` (default) | `true`

Whether message is completed, specified as `true` or `false`. A `true` value means that the message was completed. Methods that return messages set the `Completed` property to `true` when:

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`
`Transient`	`true`

Data Types: `logical`

Methods

Public Methods

complete	Process or reprocess response payload Content-Type
addFields	Add fields to message header
changeFields	Change existing fields in message header
getFields	Return message header fields matching name or class
removeFields	Remove fields from message header
replaceFields	Change values in or add fields to message header
show	Display or return formatted version of message

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`string`	Message `Header`, `Body` and `StartLine` properties, as string. For multiple messages, returns a string array. If `Body` contains binary data that cannot be converted to characters, then the method displays a message indicating the length of the data in bytes. The string is an approximate representation of what the message looks like when sent or received. Use for logging, diagnostics, or debugging. For a formatted version of messages, use `show`.
`char`	Same as `string`, except returns message as character vector.
`isequal`	Returns `true` if the visible public properties of all messages in the two message arrays are equal.

Examples

Request Image from Website

Send a message to the Hubble Heritage website requesting an image of Jupiter.

Format an HTTP request message and display the Content-Type of the response message body. The server returns a JPEG image.

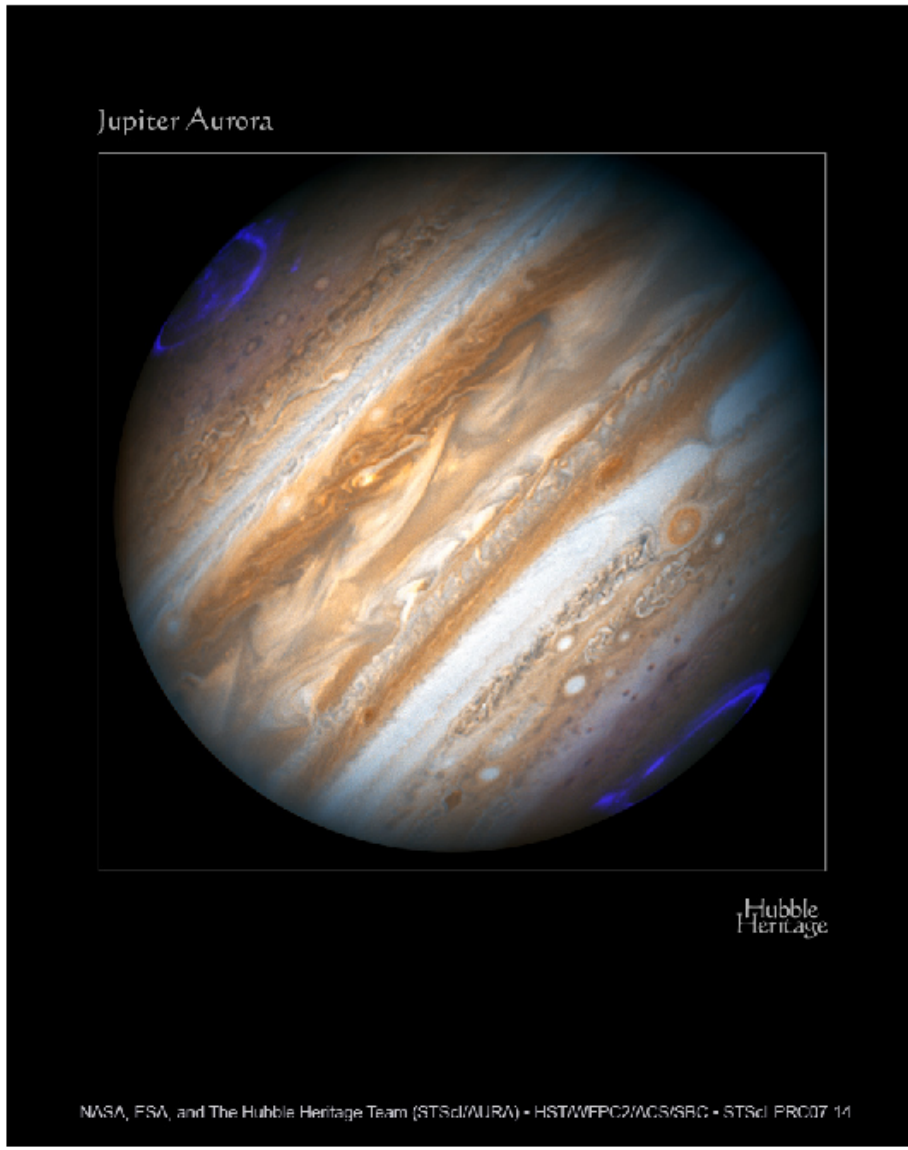
```
request = matlab.net.http.RequestMessage;
uri = matlab.net.URI('http://heritage.stsci.edu/2007/14/images/p0714aa.jpg');
r = send(request,uri);
r.Body.ContentType
```

```
ans =
```

```
  MediaType with properties:

          Type: "image"
       Subtype: "jpeg"
    Parameters: []
     MediaInfo: "image/jpeg"
        Weight: []
```

Display the image using the `imshow` function. MATLAB resizes the image to fit on the screen.

```
warning('off','Images:initSize:adjustingMag');
imshow(r.Body.Data)
```



See Also

`MessageBody` | `RequestMessage` | `StatusCode` | `StatusLine` | `send`

Introduced in R2016b

complete

Class: `matlab.net.http.ResponseMessage`
Package: `matlab.net.http`

Process or reprocess response payload Content-Type

Syntax

```
msg = complete(msg)
msg = complete(msg,consumer)
```

Description

`msg = complete(msg)` converts the `msg.Body.Payload` property to `msg.Body.Data` using the current value of the Content-Type header field in `msg`.

`msg = complete(msg,consumer)` returns a copy of the message with `msg.Body.Payload` processed by a `matlab.net.http.io.ContentConsumer`. The consumer might store its result in `msg.Body.Data` or process it in some other manner.

Use the `complete` method when:

- `Body.Data` was unset or not set properly because the server inserted the wrong Content-Type in the message or the Content-Type was missing.
- You set the ConvertResponse `HTTPOptions.ConvertResponse` property to `false` to prevent conversion of the data when it was originally received.
- You specified the wrong `consumer` when sending the message.

If there was an exception processing the received message, or if you set the `HTTPOptions.SavePayload` property when you sent the request, the `Body.Payload` in this response message contains the original payload (if any). In this case, modify the header of this message to add or correct the Content-Type field. Then call the `complete` method to process the response as if the server had inserted that Content-Type field originally. The result is new contents in `Body.Data` and/or `Data` processed by the specified `consumer`.

If `Body.Payload` is set, then this method ignores the current value of `Body.Data` and reprocesses that payload based on Content-Type. This case occurs for a conversion error or if you specified `SavePayload`. But if conversion of the incoming data succeeded originally, but was incorrect, `Body.Data` is set and `Body.Payload` might be empty. In this case, change the ContentTypeField in the received message to the desired type and then call this method. `complete` attempts to convert the data back to a payload based on the `Body.ContentType` property used to convert it originally. Then it is reconverted using the new Content-Type header in the response message. If `Data` is not empty, then the returned `Body.Payload` is set.

If you specified `SavePayload` when sending the message, `complete` uses the original payload that was preserved in `Body.Payload` instead, with no loss of information.

If the `ResponseMessage.Completed` property is set, `complete` does nothing. In a message that contains a `Body`, this property is normally set only if `msg`.`Body.Payload` has been set.

Input Arguments

`msg` — Response message
`matlab.net.http.ResponseMessage` object

Response message, specified as a `matlab.net.http.ResponseMessage` object.

Output Arguments

`msg` — Completed response message
`matlab.net.http.ResponseMessage` object

Completed response message, returned as a `matlab.net.http.ResponseMessage` object.

Examples

Convert a `text/plain` Content-Type Field

Assume that the server returned a response containing a JSON string but specified a Content-Type field of `text/plain` instead of `application/json`. In this case,

`Body.Payload` is empty and `msg.Body.Data` contains an ASCII string (since the default charset for `text/plain` is `us-ascii`). To process this data and obtain a JSON structure:

```
response = response.changeFields('Content-Type','application/json');
response = response.complete();
data = response.Body.Data;
```

The call to `complete` converts `Body.Data` to `Body.Payload` using `us-ascii` encoding. The method then reconverts `Body.Payload` to `utf-8` before processing it as a JSON string and storing the result in `Body.Data`. This conversion does not retain any non-ASCII characters that were garbled when converting the original payload using `text/plain` but preserves the original ASCII data.

See Also

matlab.net.http.HTTPOptions.ConvertResponse | matlab.net.http.HTTPOptions.SavePayload | matlab.net.http.ResponseMessage.Completed | `matlab.net.http.io.ContentConsumer`

Introduced in R2016b

matlab.net.http.StartLine class

Package: `matlab.net.http`

First line of HTTP message

Description

The `StartLine` class is an abstract class used as a base class for creating `matlab.net.http.RequestLine` and `matlab.net.http.StatusLine` elements.

Methods

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`string`	Start line as string
`char`	Start line as character vector as it would appear in first line of HTTP message

See Also

`RequestLine` | `StatusLine`

Introduced in R2016b

matlab.net.http.StatusClass class

Package: `matlab.net.http`

Status class of HTTP response

Description

The `StatusClass` enumeration class provides identifiers for status classes.

Numeric Representation	Enumeration Member Name
100	`Informational`
200	`Successful`
300	`Redirection`
400	`ClientError`
500	`ServerError`

Methods

Public Methods

getReasonPhrase StatusClass name as text phrase

See Also

`StatusCode`

Introduced in R2016b

getReasonPhrase

Class: `matlab.net.http.StatusClass`
Package: `matlab.net.http`

StatusClass name as text phrase

Syntax

```
meaning = getReasonPhrase(class)
```

Description

`meaning = getReasonPhrase(class)` returns the status class name as an English-language string with spaces between words.

Input Arguments

`class` — Status class
`matlab.net.http.StatusClass` object

Status class, specified as a `matlab.net.http.StatusClass` object.

Examples

Display HTTP Status Code Category

Send a message to `mathworks.com` and display the status code category.

```
uri = matlab.net.URI('https://www.mathworks.com');
req = matlab.net.http.RequestMessage;
resp = send(req, uri);
sc = getClass(resp.StatusCode);
disp(getReasonPhrase(sc))
```

```
Successful
```

See Also

`matlab.net.http.ResponseMessage` | `matlab.net.http.StatusLine`

Introduced in R2016b

matlab.net.http.StatusCode class

Package: `matlab.net.http`

Status code in HTTP response

Description

The `StatusCode` enumeration class provides identifiers for status codes. This list is from the 2018-09-21 version of the IANA HTTP Status Code registry, https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml.

Integer Representation	Enumeration Member Name	Integer Representation	Enumeration Member Name	Integer Representation	Enumeration Member Name
100	Continue	400	BadRequest	500	InternalServerError
101	SwitchingProtocols	401	Unauthorized	501	NotImplemented
102	Processing	402	PaymentRequired	502	BadGateway
103	EarlyHints	403	Forbidden	503	ServiceUnavailable
200	OK	404	NotFound	504	GatewayTimeout
201	Created	405	MethodNotAllowed	505	HTTPVersionNotSupported
202	Accepted	406	NotAcceptable	506	VariantAlsoNegotiates
203	NonAuthoritativeInformation	407	ProxyAuthenticationRequired	507	InsufficientStorage
204	NoContent	408	RequestTimeout	508	LoopDetected

Integer Representation	Enumeration Member Name	Integer Representation	Enumeration Member Name	Integer Representation	Enumeration Member Name
205	ResetContent	409	Conflict	509	Unassigned
206	PartialContent	410	Gone	510	NotExtended
207	MultiStatus	411	LengthRequired	511	HTTPVersionNotSupported
208	AlreadyReported	412	PreconditionFailed	451	UnavailableForLegalReasons
226	IMUsed	413	PayloadTooLarge		
		414	URITooLong		
300	MultipleChoices	415	UnsupportedMediaType		
301	MovedPermanently	416	RangeNotSatisfiable		
302	Found	417	ExpectationFailed		
303	SeeOther	421	MisdirectedRequest		
304	NotModified	422	UnprocessableEntity		
305	UseProxy	423	Locked		
306	SwitchProxy	424	FailedDependency		
307	TemporaryRedirect	426	UpgradeRequired		
308	PermanentRedirect	428	PreconditionRequired		

Integer Representation	Enumeration Member Name	Integer Representation	Enumeration Member Name	Integer Representation	Enumeration Member Name
		429	`TooManyRequests`		
		431	`RequestHeaderFieldsTooLarge`		

Methods

Public Methods

matlab.net.http.StatusCode.fromValue	Create StatusCode object from string or number
getClass	StatusClass for StatusCode
getReasonPhrase	StatusCode identifier as text phrase

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`char`	Name of status code as character vector
`string`	Three-digit value of status code as string

Examples

Display HTTP Error Code Information

Use the status code to provide error information.

Send a PUT message to the `mathworks.com` website.

```
uri = matlab.net.URI('https://www.mathworks.com');
header = matlab.net.http.field.ContentTypeField('text/plain');
req = matlab.net.http.RequestMessage('put',header,'Data');
resp = send(req, uri);
```

The website does not allow PUT methods. Display a user-friendly message.

```
sc = resp.StatusCode;
if sc ~= matlab.net.http.StatusCode.OK
    disp([getReasonPhrase(getClass(sc)),': ',getReasonPhrase(sc)])
    disp(resp.StatusLine.ReasonPhrase)
end
```

```
Client Error: Method Not Allowed
Method Not Allowed
```

How to Choose a Status Code Conversion Method

The `StatusCode` methods - `char`, `string`, `getReasonPhrase`, and `getClass` - provide information about the code and its meaning. Choose a method based on your requirements.

Suppose that your response message contains status code 307. To run this example, create the code.

```
sc = matlab.net.http.StatusCode(307);
```

Use the `char` and the `getReasonPhrase` methods to return text for the status code meaning. The `getReasonPhrase` method creates a phrase you can use in messages.

```
txt = char(sc)
```

```
txt 
'TemporaryRedirect'
```

```
msg = getReasonPhrase(sc)
```

```
msg 
'Temporary Redirect'
```

Use the `string` method to return the integer value of the status code as a string.

```
value = string(sc)
```

```
value =
"307"
```

If your code processes status codes based on status class, use the `getClass` method.

```
class = getClass(sc)
```

```
class =
Redirection
```

Display All HTTP Status Codes

```
enumeration matlab.net.http.StatusCode
```

```
Enumeration members for class 'matlab.net.http.StatusCode':

    Continue
    SwitchingProtocols
    Processing
    EarlyHints
    OK
    Created
    Accepted
    NonAuthoritativeInformation
    NoContent
    ResetContent
    PartialContent
    MultiStatus
    AlreadyReported
    IMUsed
    MultipleChoices
    MovedPermanently
    Found
    SeeOther
    NotModified
    UseProxy
    SwitchProxy
    TemporaryRedirect
    PermanentRedirect
    BadRequest
    Unauthorized
    PaymentRequired
```

```
Forbidden
NotFound
MethodNotAllowed
NotAcceptable
ProxyAuthenticationRequired
RequestTimeout
Conflict
Gone
LengthRequired
PreconditionFailed
PayloadTooLarge
URITooLong
UnsupportedMediaType
RangeNotSatisfiable
ExpectationFailed
MisdirectedRequest
UnprocessableEntity
Locked
FailedDependency
UpgradeRequired
PreconditionRequired
TooManyRequests
RequestHeaderFieldsTooLarge
UnavailableForLegalReasons
InternalServerError
NotImplemented
BadGateway
ServiceUnavailable
GatewayTimeout
HTTPVersionNotSupported
VariantAlsoNegotiates
InsufficientStorage
LoopDetected
Unassigned
NotExtended
NetworkAuthenticationRequired
```

See Also

`ResponseMessage` | `StatusClass`

Introduced in R2016b

matlab.net.http.StatusCode.fromValue

Class: `matlab.net.http.StatusCode`
Package: `matlab.net.http`

Create StatusCode object from string or number

Syntax

```
code = matlab.net.http.StatusCode.fromValue(value)
```

Description

`code = matlab.net.http.StatusCode.fromValue(value)` converts `value` to a `StatusCode` enumeration. Use this method to create a `StatusCode` object for comparisons.

Input Arguments

`value` — Number to convert
positive integer | positive integer string

Number to convert, specified as a positive integer or a positive integer string.

Attributes

`Static`	`true`

Examples

Compare Response Status with Code from External Source

Compare a response status code with a value from an external source that lists status codes as numbers or strings.

Assume that you have the status code from a response message.

```
class(resp.StatusCode)
```

```
ans = matlab.net.http.StatusCode
```

Compare the code with value `401` from the table using the `fromValue` method.

```
theValue = '401';
if resp.StatusCode == matlab.net.http.StatusCode.fromValue(theValue)
    % do something
end
```

See Also

Topics

Method Attributes

Introduced in R2016b

getClass

Class: `matlab.net.http.StatusCode`
Package: `matlab.net.http`

StatusClass for StatusCode

Syntax

```
class = getClass(code)
```

Description

`class = getClass(code)` converts status code to `matlab.net.http.StatusClass` object.

Input Arguments

code — Status code
`matlab.net.http.StatusCode` object

Status code, specified as a `matlab.net.http.StatusCode` object.

Examples

Display Class for HTTP Status Code 503

```
sc = matlab.net.http.StatusCode(503);
res = getClass(sc)
```

```
res 
ServerError
```

See Also

`matlab.net.http.StatusClass`

Introduced in R2016b

getReasonPhrase

Class: `matlab.net.http.StatusCode`
Package: `matlab.net.http`

StatusCode identifier as text phrase

Syntax

```
meaning = getReasonPhrase(code)
```

Description

`meaning = getReasonPhrase(code)` returns the status code identifier as a string (English only) with punctuation and spacing added. The meaning is based on the numeric code. `meaning` is not necessarily the same as the `ReasonPhrase` property that the server inserts into the status line of a response message.

Input Arguments

code — Status code
`matlab.net.http.StatusCode` object

Status code, specified as a `matlab.net.http.StatusCode` object.

Examples

Display Meaning of HTTP Status Code

```
sc = matlab.net.http.StatusCode(500);
res = getReasonPhrase(sc)
```

```
res =
'Internal Server Error'
```

See Also

`matlab.net.http.StatusLine`

Introduced in R2016b

matlab.net.http.StatusLine class

Package: `matlab.net.http`
Superclasses: `matlab.net.http.StartLine`

First line of HTTP response message

Description

The server inserts a `StatusLine` object into every HTTP response message. For information about a status line, see RFC 7230 Status Line, section 3.1.2.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

`obj = StatusLine(protocolVersion,statusCode,reasonPhrase)` creates a status line with the specified properties on page 1-6867. Use constructors for testing.

`obj = StatusLine(str)` creates a status line by parsing `str`.

Input Arguments

`str` — Status line
string | character vector

Status line, specified as a string or a character vector.

Properties

`ProtocolVersion` — Protocol version

`matlab.net.http.ProtocolVersion`

Protocol version, specified as a `matlab.net.http.ProtocolVersion` object.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

`StatusCode` — Status code

`matlab.net.http.StatusCode` enumeration | string | integer

Status code, specified as a `matlab.net.http.StatusCode` enumeration, string, or integer. The value is a `StatusCode` object if the server returns one of its enumeration values. If the server returns a number, then the value is an integer.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

`ReasonPhrase` — Reason phrase from server

string

Reason phrase from server, specified as a string. `ReasonPhrase` is empty if the server does not provide a reason. This value is not necessarily the same as the value returned by the `StatusCode.getReasonPhrase` method.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

Methods

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`string`	Status line as string
`char`	Status line as character vector

See Also

`ProtocolVersion` | `StatusCode` | `getReasonPhrase`

External Websites

RFC 7230 Status Line, section 3.1.2

Introduced in R2016b

matlab.net.http.field Package

Summary of header field classes in MATLAB HTTP interface

Description

Classes

AcceptField	HTTP Accept header field
AuthenticateField	HTTP WWW-Authenticate or Proxy-Authenticate header field
AuthenticationInfoField	HTTP Authentication-Info header field in response message
AuthorizationField	HTTP Authorization or Proxy-Authorization header field
ContentDispositionField	HTTP Content-Disposition header field
ContentLengthField	HTTP Content-Length field
ContentLocationField	HTTP Content-Location header field
ContentTypeField	HTTP Content-Type header field
CookieField	HTTP Cookie header field
DateField	HTTP Date header field
GenericField	HTTP header field with any name and value
GenericParameterizedField	GenericField to support parameterized syntax
HTTPDateField	HTTP header field containing date
IntegerField	Base class for HTTP header fields containing nonnegative integers
LocationField	HTTP Location header field
MediaRangeField	Base class for HTTP Content-Type and Accept header fields
SetCookieField	HTTP Set-Cookie header field
URIReferenceField	Base class for HTTP header fields containing URI components

See Also

Introduced in R2016b

matlab.net.http.field.AcceptField class

Package: `matlab.net.http.field`
Superclasses: `matlab.net.http.HeaderField`, `matlab.net.http.field.MediaRangeField`

HTTP Accept header field

Description

An `AcceptField` object is an HTTP header field in a request message. The field contains one or more media type specifications indicating the type of content acceptable to the client.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Description

`obj = matlab.net.http.field.AcceptField(value)` creates an Accept header field with the `Value` property set to `value`.

Use an Accept field if the server can return different `MediaType` objects and you want to receive only certain types. If you do not specify an Accept field, the server might assume that you are willing to receive any type.

Properties

`Name` — Header field name
`'Accept'`

Header field name, specified as `'Accept'`.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Value` — Media type
vector of `matlab.net.http.MediaType` objects | vector of strings | cell array of character vectors

Media type, specified as a vector of `matlab.net.http.MediaType` objects, a vector of strings, or a cell array of character vectors. All strings must be acceptable to the `MediaType` constructor. Each `Value` represents a media type containing an optional quality (`'q'`) parameter. The `Value` property of the field is a comma-separated list of the `MediaType` objects converted to strings.

Example: `MediaType('application/json','q','.5')`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

Methods

Public Methods

addFields	Add fields to HeaderField array
removeFields	Remove fields from header field array
changeFields	Change existing values in HeaderField array
replaceFields	Change values in or add fields to array of HeaderFields
getFields	Return header fields matching name or class
eq	Compare two HeaderField arrays
convertLike	Convert header field value like another header field
parse	Parse header field value and return as strings
displaySubclasses	Display supported HeaderField subclasses

Specialized for Subclass

`convert`	Parse `AcceptField` as a comma-separated list of values and converts each `Value` property to a `MediaType` object. Each value is a media-type expression as defined in RFC 7231 Media Type, section 3.1.1.1.

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`isequal`	`true` if two header field arrays are the same size and corresponding elements are equal, as described by `eq` method.
`string`	Array of header fields as string, as it appears in a message. Inserts newline characters between fields but not at the end of all fields.
`char`	Array of header fields as character vector, as described by `string` method.

See Also

`MediaType` | `RequestMessage` | `ResponseMessage`

External Websites

RFC 7231 Semantics and Content, section 5.3.2 Accept
RFC 7231 Media Type, section 3.1.1.1

Introduced in R2016b

matlab.net.http.field.AuthenticateField class

Package: `matlab.net.http.field`
Superclasses: `matlab.net.http.HeaderField`

HTTP WWW-Authenticate or Proxy-Authenticate header field

Description

An `AuthenticateField` object contains one or more challenges from a server asking for authentication information. A server or proxy creates an `AuthenticateField` in a response message.

When you send a request message to a server or through a proxy that requires authentication, MATLAB automatically tries to authenticate to the server or proxy when:

- `HTTPOptions.Authenticate` property is true (default)
- `HTTPOptions.Credentials` property contains the necessary names and passwords.

If authentication is successful, then the response message returns an `OK` status and does not contain an authentication field.

If you disable authentication or if authentication failed, then the response message returns an authentication field. In that case, the status code of the response message is either 401 (`Unauthorized`) or 407 (`ProxyAuthenticationRequired`). Examine the `AuthInfo` object and respond by adding the appropriate `AuthorizationField` to the request message containing your credentials. Or resend the request by setting the correct `Credentials` property in `HTTPOptions`.

If the server or proxy requires an authentication scheme that MATLAB does not support, you must implement the authentication protocol yourself. Create a request message with the appropriate credentials and other information.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Description

`obj = matlab.net.http.field.AuthenticateField(name,value)` creates an authentication header field with the `Name` property set to `name` and the `Value` property set to `value`.

A server creates this field in a response message. Use this constructor for test purposes.

Properties

`Name` — Header field name

`'WWW-Authenticate'` | `'Proxy-Authenticate'`

Header field name, specified as `'WWW-Authenticate'` or `'Proxy-Authenticate'`.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Value` — Challenges

vector of `AuthInfo` objects | string

A comma-separated list of challenges, specified as a vector of `matlab.net.http.AuthInfo` objects or a string in the format defined by RFC 7235 and RFC 2617. Use the `AuthenticateField.convert` method to parse this field.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

Methods

Public Methods

addFields	Add fields to HeaderField array
removeFields	Remove fields from header field array
changeFields	Change existing values in HeaderField array
replaceFields	Change values in or add fields to array of HeaderFields
getFields	Return header fields matching name or class
eq	Compare two HeaderField arrays
convertLike	Convert header field value like another header field
parse	Parse header field value and return as strings
displaySubclasses	Display supported HeaderField subclasses

Specialized for Subclass

`convert`	`Value` property of `AuthenticateField` as vector of `AuthInfo` objects, one for each challenge in the header field, in the order they appear in `AuthenticateField`.

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`isequal`	`true` if two header field arrays are the same size and corresponding elements are equal, as described by `eq` method
`string`	Array of header fields as string, as it appears in a message. Inserts newline characters between fields but not at the end of all fields.
`char`	Array of header fields as character vector, as described by `string` method

See Also

`AuthInfo` | `Credentials` | `HTTPOptions` | `RequestMessage` | `ResponseMessage` | `StatusCode` | `matlab.net.http.field.AuthorizationField`

External Websites

RFC 7235 Hypertext Transfer Protocol (HTTP/1.1): Authentication
RFC 2617 HTTP Authentication: Basic and Digest Access Authentication

Introduced in R2016b

matlab.net.http.field.AuthenticationInfoField class

Package: `matlab.net.http.field`
Superclasses: `matlab.net.http.HeaderField`

HTTP Authentication-Info header field in response message

Description

An `AuthenticationInfoField` object contains information about a successful Digest authentication. A server or proxy creates an Authentication-Info header field in a response message.

Since this field appears only in response messages, you do not normally create one of these fields. Use this field only if you are implementing your own authentication protocol or one that MATLAB does not automatically support.

If you use the automatic authentication provided by MATLAB by specifying the `Credentials` property in `HTTPOptions`, you do not need to access this field.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Description

`obj = matlab.net.http.field.AuthenticationInfoField(name)` creates an Authentication-Info header field with the `Name` property set to `name` with no value.

`obj = matlab.net.http.field.AuthenticationInfoField(name,value)` creates a header field with the `Value` property set to `value`.

Properties

`Name` — Header field name
`'Authentication-Info'` | `'Proxy-Authentication-Info'`

Header field name, specified as `'Authentication-Info'` or `'Proxy-Authentication-Info'`.

Attributes:

GetAccess	public
SetAccess	public

`Value` — Header field value
string

Header field value, specified as a comma-separated list of `token=value` strings.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true

Methods

Public Methods

addFields	Add fields to HeaderField array
removeFields	Remove fields from header field array
changeFields	Change existing values in HeaderField array
replaceFields	Change values in or add fields to array of HeaderFields
getFields	Return header fields matching name or class
eq	Compare two HeaderField arrays
convertLike	Convert header field value like another header field
parse	Parse header field value and return as strings

displaySubclasses Display supported HeaderField subclasses

Specialized for Subclass

`convert`	`Value` property of `AuthenticationInfoField` as vector of `AuthInfo` objects containing tokens and values

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`isequal`	`true` if two header field arrays are the same size and corresponding elements are equal, as described by `eq` method
`string`	Array of header fields as string, as it appears in a message. Inserts newline characters between fields but not at the end of all fields.
`char`	Array of header fields as character vector, as described by `string` method

See Also

`AuthInfo` | `HTTPOptions`

External Websites

RFC 7615 — HTTP Authentication-Info and Proxy-Authentication-Info Response Header Fields
RFC 2617 — section 3.2.3 The Authentication-Info Header
RFC 2617 — section 3.6 Proxy-Authentication and Proxy-Authorization

Introduced in R2016b

matlab.net.http.field.AuthorizationField class

Package: `matlab.net.http.field`
Superclasses: `matlab.net.http.HeaderField`

HTTP Authorization or Proxy-Authorization header field

Description

An `AuthorizationField` object contains credentials in a request message in response to a challenge from a server in an `AuthenticateField`. The credentials are in the form of an `AuthInfo` object. For a description of these fields, see RFC 7235 section 4.2 and section 4.4.

MATLAB automatically creates this field when:

- `HTTPOptions.Authenticate` property is `true` (default) in a request message.
- You have specified appropriate credentials in the `HTTPOptions.Credentials` property.
- MATLAB supports the authentication scheme requested by the server.

You create this field explicitly when you disable automatic authentication or implement an unsupported authentication protocol. If you create this field explicitly, then set the `Value` property to a valid authorization string or an `AuthInfo` object.

To see the `AuthorizationField` that was sent to the server for automatic authentication, examine the completed request or history arguments returned by the `RequestMessage.send` method.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Description

`obj = matlab.net.http.field.AuthorizationField(name,value)` creates an authorization header field with the Name property set to `name` and the Value property set to `value`. Create this field if you disabled automatic authentication or to implement an unsupported authentication protocol.

Properties

`Name` — Header field name
`'Authorization'` | `'Proxy-Authorization'`

Header field name, specified as `'Authorization'` or `'Proxy-Authorization'`.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Value` — Authorization
string | `matlab.net.http.AuthInfo` object

Authorization, specified as a valid authorization string or a `matlab.net.http.AuthInfo` object.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

Methods

Public Methods

addFields	Add fields to HeaderField array

removeFields	Remove fields from header field array
changeFields	Change existing values in HeaderField array
replaceFields	Change values in or add fields to array of HeaderFields
getFields	Return header fields matching name or class
eq	Compare two HeaderField arrays
convertLike	Convert header field value like another header field
parse	Parse header field value and return as strings
displaySubclasses	Display supported HeaderField subclasses

Specialized for Subclass

`convert`	`Value` property of `AuthorizationField` as vector of `AuthInfo` objects. There is only one `AuthInfo` object in an `AuthorizationField`. If you pass in a vector of `AuthorizationField` objects, `convert` returns an equal-size vector of `AuthInfo` objects. The parameters in `AuthInfo` correspond to parameters of the credentials in the `AuthorizationField`. This set of parameters varies depending on the `AuthInfo.Scheme` property, which is the first token in the field.

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`isequal`	`true` if two header field arrays are the same size and corresponding elements are equal, as described by `eq` method
`string`	Array of header fields as string, as it appears in a message. Inserts newline characters between fields but not at the end of all fields.
`char`	Array of header fields as character vector, as described by `string` method

Examples

View AuthorizationField Credentials in Request

```
import matlab.net.http.*
creds = Credentials('Username','MyName','Password','MyPassword');
options = HTTPOptions('Credentials', creds);
[response, request] = RequestMessage().send('http://myhost.com',options);
authorizationField = request.getFields('Authorization');
authInfo = authorizationField.convert;
disp(string(authInfo));
```

See Also

External Websites

RFC 7235, section 4.2 Authorization
RFC 7235, section 4.4 Proxy-Authorization

Introduced in R2016b

matlab.net.http.field.ContentDispositionField class

Package: `matlab.net.http.field`
Superclasses: `matlab.net.http.field.GenericParameterizedField`

HTTP Content-Disposition header field

Description

Use a `ContentDispositionField` object to specify a Content-Disposition header field, which is commonly used in multipart form requests.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Description

`obj = matlab.net.http.field.ContentDispositionField(value)` creates a Content-Disposition header field with the Value property set to `value`.

Properties

Name — Header field name
`'Content-Disposition'`

Header field name, specified as `'Content-Disposition'`.

Attributes:

GetAccess	public
SetAccess	public

Value — Media type
matlab.net.http.MediaType object | string

Media type specified as a single matlab.net.http.MediaType object or string acceptable to the MediaType constructor. The value must not contain a quality 'q' parameter.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true

Methods

Public Methods

getParameter	Return value of parameter in field
setParameter	Set value of parameter in field
removeParameter	Remove parameter from field
addFields	Add fields to HeaderField array
removeFields	Remove fields from header field array
changeFields	Change existing values in HeaderField array
replaceFields	Change values in or add fields to array of HeaderFields
getFields	Return header fields matching name or class
eq	Compare two HeaderField arrays
convertLike	Convert header field value like another header field
parse	Parse header field value and return as strings
displaySubclasses	Display supported HeaderField subclasses

Specialized for Subclass

`convert`	`Value` property of `ContentDispositionField` as matrix of parameter names and values. Column 1 contains the name of the parameter and column 2 is its value. For an unnamed parameter, for example the `Type` property, column 1 contains `""`.

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`isequal`	`true` if two header field arrays are the same size and corresponding elements are equal, as described by `eq` method
`string`	Array of header fields as string, as it appears in a message. Inserts newline characters between fields but not at the end of all fields.
`char`	Array of header fields as character vector, as described by `string` method

See Also

`MediaType`

Introduced in R2018a

matlab.net.http.field.ContentLengthField class

Package: `matlab.net.http.field`
Superclasses: `matlab.net.http.HeaderField`, `matlab.net.http.field.IntegerField`

HTTP Content-Length field

Description

A `ContentLengthField` object is an HTTP header field in a request or a response message that specifies the length of the payload in bytes.

MATLAB requires all outbound messages with a payload to contain a Content-Length field. If a message contains a nonempty body, MATLAB creates a Content-Length field based on the length of the data. For more information, see RFC 7231, section 3.3.2. Content-Length.

Response messages might or might not contain this field.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Description

`obj = matlab.net.http.field.ContentLengthField(value)` creates a Content-Length header field with the `Value` property set to `value`.

Properties

Name — Header field name
'Content-Length'

Header field name, specified as 'Content-Length'.

Attributes:

GetAccess	public
SetAccess	public

Value — Length of payload
nonnegative integer | string

Length of payload in bytes, specified as a nonnegative integer or a string that evaluates to a nonnegative integer.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true

Methods

Public Methods

addFields	Add fields to HeaderField array
removeFields	Remove fields from header field array
changeFields	Change existing values in HeaderField array
replaceFields	Change values in or add fields to array of HeaderFields
getFields	Return header fields matching name or class
eq	Compare two HeaderField arrays
convertLike	Convert header field value like another header field
parse	Parse header field value and return as strings
displaySubclasses	Display supported HeaderField subclasses

Specialized for Subclass

`convert`	Return `Value` property of `ContentLengthField` as integer of type `double`

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`isequal`	`true` if two header field arrays are the same size and corresponding elements are equal, as described by `eq` method
`string`	Array of header fields as string, as it appears in a message. Inserts newline characters between fields but not at the end of all fields.
`char`	Array of header fields as character vector, as described by `string` method

See Also

External Websites

RFC 7231, section 3.3.2. Content-Length

Introduced in R2016b

matlab.net.http.field.ContentLocationField class

Package: `matlab.net.http.field`
Superclasses: `matlab.net.http.HeaderField`, `matlab.net.http.field.URIReferenceField`

HTTP Content-Location header field

Description

A `ContentLocationField` object contains a uniform resource identifier (URI). A server inserts a Content-Location field in a response message.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Properties

`Name` — Header field name
`'Content-Location'`

Header field name, specified as `'Content-Location'`.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Value` — URI
`matlab.net.URI` object | string

URI, specified as a `matlab.net.URI` object or a string acceptable to the `URI` constructor. `Value` must not contain a `Fragment` property.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`
`Dependent`	`true`

Methods

Public Methods

addFields	Add fields to HeaderField array
removeFields	Remove fields from header field array
changeFields	Change existing values in HeaderField array
replaceFields	Change values in or add fields to array of HeaderFields
getFields	Return header fields matching name or class
eq	Compare two HeaderField arrays
convertLike	Convert header field value like another header field
parse	Parse header field value and return as strings
displaySubclasses	Display supported HeaderField subclasses

Specialized for Subclass

`convert`	Return `Value` of `ContentLocationField` as `matlab.net.URI` object

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`isequal`	`true` if two header field arrays are the same size and corresponding elements are equal, as described by `eq` method
`string`	Array of header fields as string, as it appears in a message. Inserts newline characters between fields but not at the end of all fields.

`char`	Array of header fields as character vector, as described by `string` method

See Also

`matlab.net.URI`

Introduced in R2016b

matlab.net.http.field.ContentTypeField class

Package: `matlab.net.http.field`
Superclasses: `matlab.net.http.HeaderField`, `matlab.net.http.field.MediaRangeField`

HTTP Content-Type header field

Description

A `ContentTypeField` object is an HTTP header field in a request or response message. The field contains a single media type specification indicating the type of content in the body of the message.

In a request message, a Content-Type field determines how data specified in the `MessageBody.Data` property is converted. For information on this conversion, see "HTTP Data Type Conversion". If a request message contains a nonempty body, but does not contain a Content-Type field, then MATLAB creates a Content-Type field based on the data type.

Class Attributes

`Sealed`	`true`

For information on class attributes, see "Class Attributes".

Creation

Description

`obj = matlab.net.http.field.ContentTypeField(value)` creates a Content-Type header field with the `Value` property set to `value`.

Properties

Name — Header field name
'Content-Type'

Header field name, specified as 'Content-Type'.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true

Value — Media type
matlab.net.http.MediaType object | string

Media type, specified as a matlab.net.http.MediaType object or a string acceptable to the MediaType constructor. Value cannot contain a quality ('q') parameter.

Example: 'text/html;charset=utf-8'

Attributes:

GetAccess	public
SetAccess	public

Methods

Public Methods

addFields	Add fields to HeaderField array
removeFields	Remove fields from header field array
changeFields	Change existing values in HeaderField array
replaceFields	Change values in or add fields to array of HeaderFields
getFields	Return header fields matching name or class
eq	Compare two HeaderField arrays
convertLike	Convert header field value like another header field
parse	Parse header field value and return as strings
displaySubclasses	Display supported HeaderField subclasses

Specialized for Subclass

`convert`	Parse `ContentTypeField` as a comma-separated list of values and converts each `Value` property to a `MediaType` object. Each value is a media-type expression as defined in RFC 7231 Media Type, section 3.1.1.1.

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`isequal`	`true` if two header field arrays are the same size and corresponding elements are equal, as described by `eq` method.
`string`	Array of header fields as string, as it appears in a message. Insert newline characters between fields but not at the end of all fields.
`char`	Array of header fields as character vector, as described by `string` method.

See Also

`MediaType` | `MessageBody` | `RequestMessage` | `ResponseMessage`

Topics

"HTTP Data Type Conversion"

External Websites

RFC 7231 Semantics and Content, section 3.1.1.5. Content-Type
RFC 7231 Media Type, section 3.1.1.1

Introduced in R2016b

matlab.net.http.field.CookieField class

Package: `matlab.net.http.field`
Superclasses: `matlab.net.http.HeaderField`

HTTP Cookie header field

Description

If there are cookies in a Set-Cookie field, then you can send them to a server by including a `CookieField` object in your request message. For more information, see RFC 6265 HTTP State Management Mechanism.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Description

`obj = matlab.net.http.field.CookieField(value)` creates a Cookie header field with the `Value` property set to `value`.

Properties

`Name` — Header field name
`'Cookie'`

Header field name, specified as `'Cookie'`.

Attributes:

GetAccess	public
SetAccess	public

`Value` — Cookie
string

Cookie, specified as a string.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true

Methods

Public Methods

addFields	Add fields to HeaderField array
removeFields	Remove fields from header field array
changeFields	Change existing values in HeaderField array
replaceFields	Change values in or add fields to array of HeaderFields
getFields	Return header fields matching name or class
eq	Compare two HeaderField arrays
convertLike	Convert header field value like another header field
parse	Parse header field value and return as strings
displaySubclasses	Display supported HeaderField subclasses

Specialized for Subclass

`convert`	Vector of `Cookie` objects in `CookieField`

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`isequal`	`true` if two header field arrays are the same size and corresponding elements are equal, as described by `eq` method
`string`	Array of header fields as string, as it appears in a message. Inserts newline characters between fields but not at the end of all fields.
`char`	Array of header fields as character vector, as described by `string` method

Examples

Send All Received HTTP Cookies Back to Server

This example sends all cookies to a server. In practice, you would send only unexpired cookies.

If the initial exchange involves multiple messages for authentication and redirection, you might want to obtain the `CookieInfo` object from the history containing all these messages. For more information, see `CookieInfo.collectFromLog`.

```
r = matlab.net.http.RequestMessage;
resp = send(r,'https://www.mathworks.com');
setCookieFields = resp.getFields('Set-Cookie');
if ~isempty(setCookieFields)
   % fetch all CookieInfos from Set-Cookie fields and add to request
   cookieInfos = setCookieFields.convert;
   r = r.addFields(matlab.net.http.field.CookieField([cookieInfos.Cookie]));
end
resp = r.send('https://www.mathworks.com');
```

See Also

`Cookie` | `CookieInfo` | `RequestMessage` | `ResponseMessage` | `matlab.net.http.field.SetCookieField`

External Websites

RFC 6265 HTTP State Management Mechanism

Introduced in R2016b

matlab.net.http.field.DateField class

Package: `matlab.net.http.field`
Superclasses: `matlab.net.http.HeaderField`, `matlab.net.http.field.HTTPDateField`

HTTP Date header field

Description

A `DateField` object represents an HTTP Date header field. A Date field is an optional field that represents the date and time a request or a response message was originally sent. If you do not specify a Date field in a request message, then MATLAB inserts one when you send it.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Description

`obj = matlab.net.http.field.DateField(value)` creates a Date header field with the `Value` property set to `value`.

`obj = matlab.net.http.field.DateField` sets the field value to the current date and time.

Properties

`Name` — Header field name
`'Date'`

Header field name, specified as `'Date'`.

Attributes:

GetAccess	public
SetAccess	public

`Value` — Date and time
current date and time (default) | `datetime` | string

Date and time, specified as a `datetime` object or a string in a valid HTTP date format. The default is the current date and time. `Value` must not be in the future. If the time zone is missing, then the local time zone is assumed. The `datetime` `Format` property is ignored.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true

Methods

Public Methods

addFields	Add fields to HeaderField array
removeFields	Remove fields from header field array
changeFields	Change existing values in HeaderField array
replaceFields	Change values in or add fields to array of HeaderFields
getFields	Return header fields matching name or class
eq	Compare two HeaderField arrays
convertLike	Convert header field value like another header field
parse	Parse header field value and return as strings
displaySubclasses	Display supported HeaderField subclasses

Specialized for Subclass

`convert`	`Value` property of `DateField` as array of MATLAB `datetime` objects

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`isequal`	`true` if two header field arrays are the same size and corresponding elements are equal, as described by `eq` method
`string`	Array of header fields as string, as it appears in a message. Inserts newline characters between fields but not at the end of all fields.
`char`	Array of header fields as character vector, as described by `string` method

See Also

`datetime`

External Websites

RFC 7231 Semantics and Content, section 7.1.1.2 Date

Introduced in R2016b

matlab.net.http.field.GenericField class

Package: `matlab.net.http.field`
Superclasses: `matlab.net.http.HeaderField`

HTTP header field with any name and value

Description

Use a `GenericField` header field for testing or to work around a built-in restriction that might not be appropriate for your application. The `HeaderField` class and its subclasses validate the `Value` property for the given `Name` property. To override a `Value` that might otherwise be rejected, create a `GenericField` header field.

If a header field in a response message contains a value that is not valid for the field name, then MATLAB creates a `GenericField` object.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

Description

`obj = matlab.net.http.field.GenericField(name,value)` creates an HTTP header field with the `Name` property set to `name` and the `Value` property set to `value`.

Properties

`Name` — Header field name
string | character vector

Header field name, specified as a string or character vector.

Attributes:

GetAccess	public
SetAccess	public

Value — Header value
string | character vector

Header value, specified as a string or character vector.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true

Methods

Public Methods

getParameter	Return value of parameter in field
setParameter	Set value of parameter in field
removeParameter	Remove parameter from field
addFields	Add fields to HeaderField array
removeFields	Remove fields from header field array
changeFields	Change existing values in HeaderField array
replaceFields	Change values in or add fields to array of HeaderFields
getFields	Return header fields matching name or class
eq	Compare two HeaderField arrays
convertLike	Convert header field value like another header field
parse	Parse header field value and return as strings
displaySubclasses	Display supported HeaderField subclasses

Specialized for Subclass

`convert`	`Value` property of header field as MATLAB type. MATLAB checks the `matlab.net.http.field` package for a class with the `GenericField.Name` property. If there is a class for this field, then `convert` invokes the `convert` method of that class. If the conversion fails or if no class supports this `Name` property, then `convert` throws an exception.

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`isequal`	`true` if two header field arrays are the same size and corresponding elements are equal, as described by `eq` method
`string`	Array of header fields as string, as it appears in a message. Inserts newline characters between fields but not at the end of all fields.
`char`	Array of header fields as character vector, as described by `string` method

Examples

Override Content-Length Header Field Value

A Content-Length header field must contain a number.

The following statement generates an error.

```
field = matlab.net.http.field.ContentLengthField('abc');
```

For a character value, create a `GenericField` object.

```
field = matlab.net.http.field.GenericField('Content-Length','abc');
```

See Also

`HeaderField` | `RequestMessage` | `ResponseMessage` | `matlab.net.http.field.ContentLengthField`

Introduced in R2016b

getParameter

Class: `matlab.net.http.field.GenericField`
Package: `matlab.net.http.field`

Return value of parameter in field

Syntax

```
[value,actname] = getParameter(field,name)
```

Description

`[value,actname] = getParameter(field,name)` returns the value of parameter `name`. If the parameter does not exist, then `getParameter` returns empty (`[]`).

Input Arguments

`field` — Header field
`matlab.net.http.field.GenericField` object

Header field, specified as a `matlab.net.http.field.GenericField` object.

`name` — Parameter name
string | character vector | `""` | `''`

Parameter name, specified as a string or a character vector.

The name must be a valid token, containing only characters defined in RFC 7230 Field Value Components, section 3.2.6. To set an unnamed parameter, use `""` or `''`.

Name matching is not case-sensitive, but `value` is case-sensitive. If `name` matches an existing parameter, but with a different case, then the parameter name is changed to the case of the specified `name`.

If there are multiple matching parameters, then all are changed to `value`.

Output Arguments

`value` — Parameter value
string | string vector | empty string

Parameter value returned as a string. If there are multiple matches, then `value` and `actname` are string vectors. If there are no matches, then `value` and `actname` are empty string arrays. The method removes quotes surrounding the parameter value from `value`.

`actname` — Actual parameter name
string | string vector | empty string

Actual parameter name returned as a string, string vector, or empty string.

See Also

Introduced in R2018a

removeParameter

Class: `matlab.net.http.field.GenericField`
Package: `matlab.net.http.field`

Remove parameter from field

Syntax

```
field = removeParameter(field,name)
```

Description

`field = removeParameter(field,name)` removes all instances of the named parameter and its value and returns the modified field. If the parameter does not exist, then `removeParameter` does nothing.

Input Arguments

`field` — Header field
`matlab.net.http.field.GenericField` object

Header field, specified as a `matlab.net.http.field.GenericField` object.

`name` — Parameter name
string | character vector

Parameter name, specified as a string or a character vector.

Name matching is not case-sensitive, but `value` is case-sensitive. If `name` matches an existing parameter, but with a different case, then the parameter name is changed to the case of the specified `name`. If there are multiple matching parameters, then all are removed.

See Also

Introduced in R2018a

setParameter

Class: `matlab.net.http.field.GenericField`
Package: `matlab.net.http.field`

Set value of parameter in field

Syntax

```
field = setParameter(field,name,value)
```

Description

`field = setParameter(field,name,value)` sets the `value` of the parameter `name` and returns the modified `field`.

The returned `field.Value` might be reformatted to remove extraneous whitespace in the original value.

Input Arguments

`field` — Header field
`matlab.net.http.field.GenericField` object

Header field, specified as a `matlab.net.http.field.GenericField` object.

`name` — Parameter name
string | character vector | `""` | `''`

Parameter name, specified as a string or a character vector.

The name must be a valid token, containing only characters defined in RFC 7230 Field Value Components, section 3.2.6. To set an unnamed parameter, use `""` or `''`.

Name matching is not case-sensitive, but `value` is case-sensitive. If `name` matches an existing parameter, but with a different case, then the parameter name is changed to the case of the specified `name`.

If there are multiple matching parameters, then all are changed to `value`.

`value` — Parameter value
string | character vector

Parameter value, specified a string or character vector. `value` can contain any characters. If it contains characters not allowed in a token, and is not already quoted, the method adds quotes. An empty string is inserted as paired double-quotes (`""`).

See Also

Introduced in R2018a

matlab.net.http.field.GenericParameterizedField class

Package: `matlab.net.http.field`
Superclasses: `matlab.net.http.field.GenericField`

GenericField to support parameterized syntax

Description

A `GenericParameterizedField` object is a version of a `GenericField` object that supports the following parameterized syntax.

```
Type; param1=value1; param2=value2; param3=value3; ...
```

`Type` is a token and each `param=value` pair represents the name and value of a parameter. `Type` is optional, though subclasses can require it. Unlike `GenericField`, this field only supports a single set of parameters (collectively called an "element"), not a comma-separated list of elements.

Creation

Description

`obj = GenericParameterizedField(name,value)` creates an HTTP header field with the Name property set to `name` and the Value property set to `value`. If `value` is a string or character vector, then `value` is used as is. Otherwise, MATLAB attempts to convert it to a string.

`value` should contain a type and semicolon-separated list of parameters in the following form, where `type` is the value of the `Type` property and each `param=value` defines a parameter. However, MATLAB does not enforce this syntax of `value`.

```
type; param1=value1; param2=value2; param3=value3; ...
```

If `value` is an Nx2 string matrix, each row of the matrix represents a `param=value` parameter of the field, in the form:

```
""     type
param1 value1
param2 value2
```

Any row with an empty name appears in the field as a value without a name. Normally the first row is the `Type`. When using this form of the constructor, MATLAB checks that the param names and the type are legal tokens. For `param=value` pairs, MATLAB quotes values that contain reserved characters, if they are not already quoted, and escapes double-quotes.

The type is optional. If not set, then it can be set later using the `Type` property or by calling the method `setParameter(obj,"",type)`.

`obj = GenericParameterizedField(name,type,Name,Value)` creates a header field with the Type property set to `type` and parameters with additional options specified by one or more `Name,Value` pair arguments.

This syntax is roughly the same as specifying the following string matrix argument:

```
obj = GenericParameterizedField(NAME, ["" TYPE; PARAM1 VALUE1; PARAM2 VALUE2; ...])
```

Properties

Type — Field Type property

string | string array

Field Type property specified as a string. The value is equal to any token in the field's value that is not part of a name=value pair. If there is more than one, then `Type` is a string array containing all such tokens. There is normally one token at the start of the value that is considered the type of the value.

For example, in the following statement, the `Type` property is `MYTYPE`.

```
MYTYPE; foo=bar; abc=def; hij=klm
```

In the following statement, there is no Type, so the `Type` property is `""`.

```
foo=bar; abc=def; hij=klm
```

To remove all tokens, set `Type` to an empty array or an empty string.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

`Name` — Field name
string | character vector

Field name specified as a string or character vector. This property is inherited from superclass `matlab.net.http.HeaderField`.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Value` — Field value
string | any type valid for `Name`

Field value specified as a string or any type valid for the `Name` property. This property is inherited from superclass `matlab.net.http.HeaderField`.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

Methods

Public Methods

getParameter	Return value of parameter in field
setParameter	Set value of parameter in field
removeParameter	Remove parameter from field
addFields	Add fields to HeaderField array
removeFields	Remove fields from header field array
changeFields	Change existing values in HeaderField array

replaceFields	Change values in or add fields to array of HeaderFields
getFields	Return header fields matching name or class
eq	Compare two HeaderField arrays
convertLike	Convert header field value like another header field
parse	Parse header field value and return as strings
displaySubclasses	Display supported HeaderField subclasses

Specialized for Subclass

`convert`	`Value` property as matrix of parameter names and values. Column 1 contains the name of the parameter and column 2 is its value. For an unnamed parameter, for example the `Type` property, column 1 contains `""`.

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`isequal`	`true` if two header field arrays are the same size and corresponding elements are equal, as described by `eq` method
`string`	Array of header fields as string, as it appears in a message. Inserts newline characters between fields but not at the end of all fields.
`char`	Array of header fields as character vector, as described by `string` method

See Also

Introduced in R2018a

matlab.net.http.field.HTTPDateField class

Package: `matlab.net.http.field`
Superclasses: `matlab.net.http.HeaderField`

HTTP header field containing date

Description

An `HTTPDateField` object is an HTTP header field containing a date. The fields implemented by this class contain a single date in standard HTTP date format.

Creation

Description

`obj = matlab.net.http.field.HTTPDateField(name)` creates an HTTP date header field with the `Name` property set to `name`.

`obj = matlab.net.http.field.HTTPDateField(name,value)` sets the `Value` property to `value`.

Properties

`Name` — Date field name
`'Date'` | `'Expires'` | `'Retry-After'` | `'Accept-Datetime'` | `'Last-Modified'` | `'If-Modified-Since'`

Date field name, specified as a string of one of these values: `'Date'`, `'Expires'`, `'Retry-After'`, `'Accept-Datetime'`, `'Last-Modified'`, or `'If-Modified-Since'`.

To create a `Date` field, use the `matlab.net.http.field.DateField` subclass.

Attributes:

GetAccess	public
SetAccess	public

Value — Date
datetime | string

Date, specified as a `datetime` object or a string in a valid HTTP date format as specified in RFC 7231 Semantics and Content, section 7.1.1.1. Date/Time Formats. If a `datetime` object does not have a time zone, then it is assumed to be local.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true

Methods

Public Methods

addFields	Add fields to HeaderField array
removeFields	Remove fields from header field array
changeFields	Change existing values in HeaderField array
replaceFields	Change values in or add fields to array of HeaderFields
getFields	Return header fields matching name or class
eq	Compare two HeaderField arrays
convertLike	Convert header field value like another header field
parse	Parse header field value and return as strings
displaySubclasses	Display supported HeaderField subclasses

Specialized for Subclass

`convert`	`Value` property of `HTTPDateField` as array of MATLAB `datetime` objects. `convert` interprets the field as a comma-separated list of quoted HTTP dates.

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`isequal`	`true` if two header field arrays are the same size and corresponding elements are equal, as described by `eq` method
`string`	Array of header fields as string, as it appears in a message. Inserts newline characters between fields but not at the end of all fields.
`char`	Array of header fields as character vector, as described by `string` method

See Also

`matlab.net.http.field.DateField`

External Websites

RFC 7231 Semantics and Content, section 7.1.1.1. Date/Time Formats

Introduced in R2016b

matlab.net.http.field.IntegerField class

Package: `matlab.net.http.field`
Superclasses: `matlab.net.http.HeaderField`

Base class for HTTP header fields containing nonnegative integers

Description

Use the `IntegerField` class to construct a header field containing a nonnegative integer, for which there is no existing class in the `matlab.net.http.field` package.

Creation

Description

`obj = matlab.net.http.field.IntegerField(name,value)` creates an HTTP header field with the `Name` property set to `name` and the `Value` property set to `value`.

Properties

`Name` — Header field name
string

Header field name, specified as a string. Any `Name`, except names shown by the `matlab.net.http.HeaderField.displaySubclasses` method, can be used.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Value` — Integer value
string

Integer value, specified as a string representing any real, nonnegative numeric type.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`
`Dependent`	`true`

Methods

Public Methods

addFields	Add fields to HeaderField array
removeFields	Remove fields from header field array
changeFields	Change existing values in HeaderField array
replaceFields	Change values in or add fields to array of HeaderFields
getFields	Return header fields matching name or class
eq	Compare two HeaderField arrays
convertLike	Convert header field value like another header field
parse	Parse header field value and return as strings
displaySubclasses	Display supported HeaderField subclasses

Specialized for Subclass

`convert`	`Value` property of `IntegerField` as integer of type `double`

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`isequal`	`true` if two header field arrays are the same size and corresponding elements are equal, as described by `eq` method
`string`	Array of header fields as string, as it appears in a message. Inserts newline characters between fields but not at the end of all fields.

`char`	Array of header fields as character vector, as described by `string` method

See Also

`matlab.net.http.HeaderField.displaySubclasses`

Introduced in R2016b

matlab.net.http.field.LocationField class

Package: `matlab.net.http.field`
Superclasses: `matlab.net.http.HeaderField`, `matlab.net.http.field.URIReferenceField`

HTTP Location header field

Description

A `LocationField` object contains the message uniform resource identifier (URI). The server adds a Location field to a response message to tell you to redirect messages to another URI. The meaning of this field depends on the status code of the response.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Properties

`Name` — Header field name
`'Location'`

Header field name, specified as `'Location'`.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Value` — URI
`matlab.net.URI` object | string

URI, specified as a `matlab.net.URI` object or a string acceptable to the `URI` constructor.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`
`Dependent`	`true`

Methods

Public Methods

convertLike	Convert header field value like another header field
parse	Parse header field value and return as strings
addFields	Add fields to HeaderField array
removeFields	Remove fields from header field array
changeFields	Change existing values in HeaderField array
replaceFields	Change values in or add fields to array of HeaderFields
getFields	Return header fields matching name or class
eq	Compare two HeaderField arrays
displaySubclasses	Display supported HeaderField subclasses

Specialized for Subclass

`convert`	Return `Value` property of `LocationField` as `matlab.net.URI` object

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`isequal`	`true` if two header field arrays are the same size and corresponding elements are equal, as described by `eq` method
`string`	Array of header fields as string, as it appears in a message. Inserts newline characters between fields but not at the end of all fields.

`char`	Array of header fields as character vector, as described by `string` method

See Also

`matlab.net.URI` | `matlab.net.http.StatusCode`

External Websites

RFC 3986 — Uniform Resource Identifier (URI): Generic Syntax

Introduced in R2016b

matlab.net.http.field.MediaRangeField class

Package: `matlab.net.http.field`
Superclasses: `matlab.net.http.HeaderField`

Base class for HTTP Content-Type and Accept header fields

Description

The `MediaRangeField` class is the base class for Content-Type and Accept header fields. Fields in a `MediaRangeField` object contain a comma-separated list of strings, which the `MediaType` class interprets.

Properties

`Name` — Header field name

string

Header field name, specified as a string.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Value` — Header field value

string

Header field value, specified as a string.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

Methods

Public Methods

addFields	Add fields to HeaderField array
removeFields	Remove fields from header field array
changeFields	Change existing values in HeaderField array
replaceFields	Change values in or add fields to array of HeaderFields
getFields	Return header fields matching name or class
eq	Compare two HeaderField arrays
convertLike	Convert header field value like another header field
parse	Parse header field value and return as strings
displaySubclasses	Display supported HeaderField subclasses

Specialized for Subclass

`convert`	Parse `MediaRangeField` as a comma-separated list of values and converts each `Value` property to a `MediaType` object. Each value is a media-type expression as defined in RFC 7231 Media Type, section 3.1.1.1.

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`isequal`	`true` if two header field arrays are the same size and corresponding elements are equal, as described by `eq` method.
`string`	Array of header fields as string, as it appears in a message. Insert newline characters between fields but not at the end of all fields.
`char`	Array of header fields as character vector, as described by `string` method.

See Also

`matlab.net.http.MediaType` | `matlab.net.http.field.AcceptField` | `matlab.net.http.field.ContentTypeField`

External Websites

RFC 7231 Media Type, section 3.1.1.1

Introduced in R2016b

matlab.net.http.field.SetCookieField class

Package: `matlab.net.http.field`
Superclasses: `matlab.net.http.HeaderField`

HTTP Set-Cookie header field

Description

A `SetCookieField` object appears in a response message created by the server. A message can have one or more Set-Cookie fields. To extract the `Value` property as a `CookieInfo` object, call the `convert` method.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Properties

`Name` — Header field name
`'Set-Cookie'`

Header field name, specified as `'Set-Cookie'`.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Value` — Header field value
string

Header field value, specified as a string.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`
`Dependent`	`true`

Methods

Public Methods

addFields	Add fields to HeaderField array
removeFields	Remove fields from header field array
changeFields	Change existing values in HeaderField array
replaceFields	Change values in or add fields to array of HeaderFields
getFields	Return header fields matching name or class
eq	Compare two HeaderField arrays
convert	Returns CookieInfo object for each Set-Cookie field
convertLike	Convert header field value like another header field
parse	Parse header field value and return as strings
displaySubclasses	Display supported HeaderField subclasses

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`string`	Array of header fields as string, as it appears in a message. Insert newline characters between fields but not at the end of all fields.
`char`	Array of header fields as character vector, as described by `string` method.

Examples

Obtain Information from All Set-Cookie Fields in HTTP Message

```
r = matlab.net.http.RequestMessage();
uri = matlab.net.URI('http://httpbin.org/cookies/set?xxx=zzz&abc=def');
opts = matlab.net.http.HTTPOptions('MaxRedirects',0);
resp = r.send(uri,opts);
setCookieFields = resp.getFields('Set-Cookie');
if ~isempty(setCookieFields)
   cookieInfos = setCookieFields.convert(uri);
   r = r.addFields(matlab.net.http.field.CookieField([cookieInfos.Cookie]));
end
resp = r.send('http://httpbin.org/cookies');
disp(resp.Body.Data.cookies)
```

```
    abc: 'def'
    xxx: 'zzz'
```

See Also

Cookie | CookieInfo | ResponseMessage

Introduced in R2016b

convert

Class: `matlab.net.http.field.SetCookieField`
Package: `matlab.net.http.field`

Returns CookieInfo object for each Set-Cookie field

Syntax

```
value = convert(fields)
value = convert(fields,uri)
```

Description

`value = convert(fields)` returns a `CookieInfo` object for each Set-Cookie field. A `SetCookieField` contains information about one cookie. If the server sends multiple cookies, then there are multiple `SetCookieField` objects.

`value = convert(fields,uri)` compares `uri` against or initializes the `CookieInfo.Domain` property.

Input Arguments

`fields` — Set-Cookie header field
`matlab.net.http.field.SetCookieField` object

Set-Cookie header field, specified as a `matlab.net.http.field.SetCookieField` object.

`uri` — URI
`matlab.net.URI` object

URI, specified as a `matlab.net.URI` object. The URI is that of the request, used to compare against or initialize the `CookieInfo.Domain` property.

If the `SetCookieField` does not contain a Domain attribute, then the `CookieInfo.Domain` property is set to the `URI.Host` property.

If there is a Domain attribute, its value must domain-match the `URI.Host` or no `CookieInfo` is returned for that `SetCookieField`. If it matches, then the `Domain` property is set to the Domain attribute. For information on domain matching, see RFC 6265 section 5.1.3 and section 5.3, step 6.

If you do not specify a URI, then the `CookieInfo.Domain` property is not set or checked. Use this syntax if you do not intend to use the Domain attribute to manage your cookie store.

Output Arguments

`value` — CookieInfo
vector of `matlab.net.http.CookieInfo` objects

CookieInfo object, returned as a vector of `matlab.net.http.CookieInfo` objects corresponding to the Set-Cookie fields in the `fields` vector.

See Also

External Websites

RFC 6265, section 5.1.3 Domain Matching
RFC 6265, section 5.3 Storage Model step 6

Introduced in R2016b

matlab.net.http.field.URIReferenceField class

Package: `matlab.net.http.field`
Superclasses: `matlab.net.http.HeaderField`

Base class for HTTP header fields containing URI components

Description

The `URIReferenceField` class is a base class for creating Location and Content-Location header fields. Use the `URIReferenceField` class to construct different header fields containing a single URI or a portion of a URI.

Creation

`obj = matlab.net.http.field.URIReferenceField(name,value)` creates an HTTP header field with the `Name` property set to `name` and the `Value` property set to `value`.

Properties

`Name` — Header field name
string

Header field name, specified as a string. Any `Name`, except names shown by the `matlab.net.http.HeaderField.displaySubclasses` method, can be used.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Value` — URI
`matlab.net.URI` | string

URI, specified as a `matlab.net.URI` object or a string acceptable to the `URI` constructor.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`
`Dependent`	`true`

Methods

Public Methods

addFields	Add fields to HeaderField array
removeFields	Remove fields from header field array
changeFields	Change existing values in HeaderField array
replaceFields	Change values in or add fields to array of HeaderFields
getFields	Return header fields matching name or class
eq	Compare two HeaderField arrays
convertLike	Convert header field value like another header field
parse	Parse header field value and return as strings
displaySubclasses	Display supported HeaderField subclasses

Specialized for Subclass

`convert`	`Value` property of `URIReferenceField` as `matlab.net.URI` object

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects in this class.

`isequal`	`true` if two header field arrays are the same size and corresponding elements are equal, as described by `eq` method

`string`	Array of header fields as string, as it appears in a message. Inserts newline characters between fields but not at the end of all fields.
`char`	Array of header fields as character vector, as described by `string` method

See Also

`matlab.net.URI` | `matlab.net.http.HeaderField.displaySubclasses` | `matlab.net.http.field.ContentLocationField` | `matlab.net.http.field.LocationField`

External Websites

RFC 3986 — Uniform Resource Identifier (URI): Generic Syntax

Introduced in R2016b

matlab.net.ArrayFormat class

Package: `matlab.net`

Convert arrays in HTTP queries

Description

Use an `ArrayFormat` enumeration with the `matlab.net.QueryParameter` class to control the format used to convert query values representing multiple values.

Enumeration Member Name	Description
`csv`	Format that generates a comma-separated list, such as `parm=1,2,3`
`json`	Format that generates a JSON-like array, such as `parm=[1,2,3]`
`php`	Format that generates a name with brackets and multiple values, such as `parm[]=1&[]=2&[]=3`
`repeating`	Format that generates repeating name/value pairs, such as `parm=1&=2&=3`

A query value is considered to contain multiple values when it is one of the following:

- Nonscalar number, string, logical, or datetime (Each element is a value.)
- m-by-n character array, where each row is interpreted as a string
- Cell vector, where each element is a value

Query values, except for character arrays, with more than one dimension are not supported. In cell vectors, each element must be a scalar or character vector.

Attributes

`Sealed`	`true`

See Also

`matlab.net.QueryParameter` | `matlab.net.URI`

Topics

Class Attributes

Introduced in R2016b

matlab.net.QueryParameter class

Package: `matlab.net`

Parameter in query portion of uniform resource identifier (URI)

Description

Use the `QueryParameter` class to create a URI query string of the form:

```
name1=value1&name2=value2&name3=value3
```

where each `name=value` segment is a `QueryParameter` object, converted to a string using the `string` method. The `string` method on a vector of `QueryParameter` objects joins the results using the & character. The `string` method converts any values to strings and performs all necessary encoding of special characters in the result.

Construction

`obj = matlab.net.QueryParameter` creates an empty query parameter.

`obj = matlab.net.QueryParameter(paramName,paramValue)` creates a query parameter vector of `paramName`,`paramValue` pair arguments. You can specify several argument pairs in any order as `paramName1,paramValue1,...,paramNameN,paramValueN`.

`obj = matlab.net.QueryParameter(qStruct)` creates a query parameter vector from a structure.

`obj = matlab.net.QueryParameter(queryStr)` parses `queryStr` into the query.

`obj = matlab.net.QueryParameter( ___ ,Format)` specifies the format to be used for nonscalar values, and can include any of the input arguments in previous syntaxes.

Input Arguments

paramName — Parameter name
string | character vector

Parameter name, specified as a string or a character vector.

paramValue — Parameter value
type required by paramName

Parameter value, specified as a type required by paramName.

qStruct — Parameter names and values
structure

Parameter names and values, specified as a structure. The fields of qStruct define the parameter names and values.

Data Types: struct

queryStr — Parameter names and values
string | character vector

Parameter names and values, specified as a string or a character vector. The queryStr is a completed, encoded query as it would appear in a URI, with an optional leading ? character.

queryStr is split at the & characters into individual name=value query parameters. The Name property is set to name and the Value property is set to value.

A triplet of characters of the form % and two hex digits represents a percent-encoded byte. A sequence of these bytes is treated as UTF-8 encoded characters, which are decoded to form the Name and Value properties of the QueryParameters. Also, any plus sign '+' in queryStr is treated as a space '%20'. When the QueryParameter is converted back to a string, any required percent-encoding is performed only on characters that should be encoded. This action is done whether or not those characters were originally encoded in queryStr, so the result from the string method might not exactly match queryStr. The meaning, however, is the same when used in a URI.

For example, the UTF-8 encoding for the euro sign € is E2 82 AC.

```
q1 = matlab.net.QueryParameter('V=%e2%82%ac')
```

```
q1 =

  QueryParameter with properties:

      Name: "V"
     Value: "€"
    Format: csv
```

The + and %20 characters are treated as spaces.

```
q2 = matlab.net.QueryParameter('V=a+b%20c')
```

```
q2 =

  QueryParameter with properties:

      Name: "V"
     Value: "a b c"
    Format: csv
```

The `string` method implements percent-encoding on characters that require encoding. For example, the € is encoded.

```
string(q1)
```

```
ans = V=%E2%82%AC
```

However, the characters in the `queryStr` argument `'V=a+b%20c'` do not need encoding.

```
q3 = string(q2)
```

```
q3 = V=a+b+c
```

Although the result from the `string` method does not match `queryStr`, the values are identical when used in a URI.

Properties

Name — Parameter name
string | character vector

Parameter name, specified as a string or a character vector.

`Value` — Parameter value
real number | logical | datetime | string | character vector

Parameter value, specified as a real number, logical, datetime (with value other than NaT), string, character vector, or a vector or cell vector of these values. If `Value` is any other type, then `Value` must support `string` or `char` methods that convert the value to a character vector. If empty, `Value` is treated as an empty string.

`Format` — Encoding format
`matlab.netArrayFormat.csv` (default) | `matlab.netArrayFormat` enumeration

Encoding format, specified as a `matlab.net.ArrayFormat` enumeration, to use for encoding `Value` if it is a vector.

Methods

char	Encoded query parameter as character vector
string	Encoded query parameter as string

Attributes

`Sealed`	`true`

Examples

Create HTTP Query Parameters from Structure

Create a structure field name `this` and set it to the value `that`.

```
qStruct.this = 'that';
QP = matlab.net.QueryParameter(qStruct)
```

```
QP 
  QueryParameter with properties:

      Name: "this"
```

```
     Value: 'that'
    Format: csv
```

Create HTTP Query Parameters from Character Vector

Create a character vector with two queries, `this=that` and `one=2`. The `QueryParameter` method splits `qStr` at the & character into two `QueryParameter` objects.

```
qStr = '?this=that&one=2';
QPs = matlab.net.QueryParameter(qStr);
```

The `name=value` pairs in `qStr` define the `Name` and `Value` properties.

```
name1 = QPs(1).Name
```

```
name1 =
"this"
```

```
value1 = QPs(1).Value
```

```
value1 =
"that"
```

```
name2 = QPs(2).Name
```

```
name2 =
"one"
```

```
value2 = QPs(2).Value
```

```
value2 =
"2"
```

See Also

`matlab.net.ArrayFormat` | `matlab.net.URI`

Topics

Class Attributes

Introduced in R2016b

char

Class: `matlab.net.QueryParameter`
Package: `matlab.net`

Encoded query parameter as character vector

Syntax

```
str = char(obj)
```

Description

`str = char(obj)` returns a `QueryParameter` object as an encoded MATLAB character vector.

Input Arguments

`obj` — Query parameter
`matlab.net.QueryParameter` object

Query parameter, specified as one or more `matlab.net.QueryParameter` objects.

Output Arguments

`str` — Query parameter
character vector

Query parameter, returned as a character vector. For more information, see `string`.

See Also

`string`

Introduced in R2016b

string

Class: `matlab.net.QueryParameter`
Package: `matlab.net`

Encoded query parameter as string

Syntax

```
str = string(obj)
```

Description

`str = string(obj)` returns a `QueryParameter` object as an encoded string.

Input Arguments

`obj` — Query parameter
`matlab.net.QueryParameter` object

Query parameter, specified as one or more `matlab.net.QueryParameter` objects. If `obj` is a vector of `QueryParameter` objects, then the method returns a single string joining the encoded members with & character. If `obj` is empty, then the method returns `""`.

Output Arguments

`str` — Query parameter
string

Query parameter, returned as a string.

`str` is of the form `name=value`, where `name` is the `Name` property and `value` is the `Value` property, represented as a string. If `Value` is a cell array or nonscalar other than a

character vector, it is converted based on the `Format` property. The conversion might result in multiple `name=value` pairs separated by the & character and including other punctuation. Special characters in `Name` or `Value` that are not permitted in a query are percent-encoded (%), except for space, which is encoded as +. For more information on encoding, see the `qStr` argument to the QueryParameter constructor.

See Also

`matlab.net.ArrayFormat` | `matlab.net.QueryParameter`

Introduced in R2016b

matlab.net.URI class

Package: `matlab.net`

Uniform resource identifier (URI)

Description

The `matlab.net.URI` class constructs an internet uniform resource identifier (URI), such as a web address or a URL. An internet URI is a string divided into components. Each component is represented by a property of the `URI` class. The following text shows the properties and their associated punctuation, separated by spaces for clarity. The spaces do not appear in the encoded URI. The associated punctuation is not part of the property value.

```
Scheme: //Authority /Path(1) /Path(2) ... /Path(end) ?Query #Fragment
```

where `Authority` contains these properties:

```
UserInfo@ Host :Port
```

Use the `matlab.net.URI` `string` or `char` methods to create an internet URI. These methods encode the properties by adding punctuation to nonempty properties and by escaping reserved characters.

All properties are optional. However, different uses might require certain properties to be set.

To eliminate a property and its punctuation from the output string, set the property value to `[]`.

Construction

`obj = matlab.net.URI` creates an empty URI.

`obj = matlab.net.URI(destination)` creates the URI specified by `destination`.

`obj = matlab.net.URI(destination,queryVector)` sets the `Query` property to `queryVector`. `Query` values are appended to any query parameters already specified in `destination`.

`obj = matlab.net.URI(destination,queryName,queryValue)` adds one or more `queryName`,`queryValue` parameters to the `Query` property.

`obj = matlab.net.URI(destination,queryVector,queryName,queryValue)` adds `queryVector` and the `queryName`,`queryValue` parameters to the `Query` property.

`obj = matlab.net.URI( ___ ,format)` specifies the format of the output when an array appears in a `queryValue` argument. You can use any of the input arguments in the previous syntaxes.

`obj = matlab.net.URI( ___ ,'literal')` indicates that `destination` is already encoded. Use this option if you copy and paste an already-encoded URI, for example, from the address bar of a browser. When you read properties of this URI directly, you see the decoded version. The `'literal'` option does not permit you to construct an illegal URI. It prevents reencoding of `'%'` characters. Characters that must always be encoded, such as `'\'` and `' '` in the `Host` or `Path`, are still percent-encoded.

This option has no effect on `Query` (`matlab.net.QueryParameter`) arguments.

Input Arguments

`destination` — Destination
string | character vector | `matlab.net.URI` object

Destination, specified as a string or a character vector specifying a URI or portions of one, or a `matlab.net.URI` object. If `destination` is a `matlab.net.URI` object, then `destination` must be the only argument.

Example: `https://user:pwd@www.mathworks.com:8000/product/matlab?abc=def&this=that#xyz` All properties

Example: Host and Scheme properties: `https://www.mathworks.com`

Example: Host only: `//www.mathworks.com`

Example: Host and Path: `//www.mathworks.com/products/matlab/`

Example: Path only: `products/matlab/live-editor`

Example: Host and Query: `//www.mathworks.com/search/site_search.html?q=weboptions`

`queryVector` — Query property
`matlab.net.QueryParameter` object

Query property, specified as a vector of `matlab.net.QueryParameter` objects. A query is of the form:

```
name1=value1&name2=value2&name3=value3
```

Example: `matlab.net.QueryParameter('hl','en','ie','utf8','num',50)`

`queryName` — Query name
string | character vector

Query name, specified as a string or a character vector. The web service defines `queryName`,`queryValue` pairs that it accepts as part of a request. Do not encode characters in `queryName`.

`queryValue` — Query value
character array | numeric | logical | `datetime`

Query value, specified as a character array, or a numeric, logical, or `datetime` value or array. Do not encode characters in `queryValue`.

`format` — Output format
`matlab.net.ArrayFormat` object

Output format, specified as a `matlab.net.ArrayFormat` object when an array appears in a `queryValue` argument. For allowed values, see `ArrayFormat`.

The `format` argument does not affect the format of values in the `queryVector` argument.

Properties

`Scheme` — URI scheme
string | character vector

URI scheme, sometimes called protocol, appearing before the `://` characters, specified as a string or a character vector. `Scheme` always returns a string. If not empty, then

`Scheme` must be `http` or `https`. However, this convention is not enforced. MATLAB does not support other schemes, such as `file`.

Example: `http`

Example: `https`

`UserInfo` — User information
string | character vector

User information, specified as a string or a character vector. `UserInfo` appears before the `Host` property followed by an `@` character. The `string` method percent-encodes special characters. When setting `UserInfo`, do not encode the value.

Example: `name`

Example: `name:password`

`Host` — Host name
string or character vector in DNS name format | IPv4 or IPv6 address

Host name, specified as a string or a character vector. The value is in Domain Name System (DNS) format or as an Internet Protocol version 4 (IPv4) or version 6 (IPv6) address. The `string` method percent-encodes characters that are not allowed in the host portion of a URI. Period characters (`.`) are unchanged. When setting `Host`, do not encode the value.

Example: `www.mathworks.com`

Example: `2222:7344:0db8:0000:0100:8a2e:0370:85a3` IPv6 address

`Port` — Port number
number | string | character vector

Port number, specified as a number, or as a string or a character vector representing a number in the range 0–65535, stored as a `uint16`.

Example: 8000

`Path` — Path segments
string | string vector | character vector | cell array of character vectors

Path segments, specified as a string or string vector or as a character vector or cell array of character vectors. The result is always a vector of strings. To see the value of the encoded path, use the `EncodedPath` property.

A path in a URI is specified by the `EncodedPath` property. `EncodedPath` is a series of segments separated by the `/` character, where each of those segments is a member of `Path`.

```
Path(1)/Path(2)/Path(3)/.../Path(end)
```

The `/` characters do not appear in `Path`, but `EncodedPath` contains them. For example,

```
uri = matlab.net.URI;
uri.Path = {'products' 'matlab'};
P = uri.Path
```

```
P = 
    "products"    "matlab"
```

```
EP = uri.EncodedPath
```

```
EP = products/matlab
```

If you set `Path` to a character vector or scalar string that contains a `/` character, then the value is split into segments at the `/` characters. The result is the same as specifying a vector of strings or cell array of character vectors.

```
uri.Path = 'products/matlab';
P = uri.Path
```

```
P = 
    "products"    "matlab"
```

There is always one more `Path` segment than the number of `/` characters in `EncodedPath`. Any segment can be an empty string. If `Path(1)` is an empty string, then `EncodedPath` begins with `/`. If `Path(end)` is an empty string, then `EncodedPath` ends with `/`.

```
uri.Path = '/products/matlab/';
EP = uri.EncodedPath
```

```
EP = /products/matlab/
```

When setting `Path` to a nonscalar string or cell array, characters not allowed in the path portion of a URI are percent-encoded in `EncodedPath`. To include the `#` character,

```
uri.Path = {'foo#bar'};EP = uri.EncodedPath
```

```
EP = foo%23bar
```

Do not encode the # character. If you do, then the encoded characters are encoded again.

```
uri.Path = {'foo%23Fbar'};
EP = uri.EncodedPath
```

```
EP = foo%2523Fbar
```

`Path` can be relative or absolute. An absolute path is one with more than one segment, whose first segment is empty. It is encoded as a string beginning with `/` character followed by the second string. This definition of absolute path corresponds to `path-absolute`, defined in RFC 3986 section 3.3. A relative path is one whose first string is nonempty. It is encoded without a leading `/`. For example, create an absolute path:

```
uri1 = matlab.net.URI;
uri1.Path = {'' 'products' 'matlab'};
EP = uri1.EncodedPath
```

```
EP = /products/matlab
```

Create a relative path:

```
uri2 = matlab.net.URI;
uri2.Path = {'products' 'matlab'};
EP = uri2.EncodedPath
```

```
EP = products/matlab
```

If the URI contains a `Scheme`, `Host`, `UserInfo` or `Port` property, and `Path` is not empty, then `EncodedPath` has a leading `/`. The `/` character separates `Path` from the other properties. Therefore, the distinction between absolute and relative paths exists only for URIs that do not contain `Scheme`, `Host`, `UserInfo`, or `Port` properties. For example, `uri1` is an absolute path.

```
uri1.EncodedPath
```

```
ans = /products/matlab
```

Set the `Host`:

```
uri1.Host = 'www.mathworks.com';
disp(string(uri1))
```

```
//www.mathworks.com/products/matlab
```

Set `Host` of relative path `uri2`:

```
uri2.EncodedPath
```

```
ans = products/matlab
```

```
uri2.Host = 'www.mathworks.com';
disp(string(uri2))
```

```
//www.mathworks.com/products/matlab
```

To create a URI with a path that points to the root, set `Path` to `string.empty` or `["" ""]`.

```
uri.Path = {'products' 'matlab' ''};
EP = uri.EncodedPath
```

```
EP = products/matlab/
```

To set `Path` to a folder, add an empty string to the end of the vector. This convention adds a trailing `/` to `EncodedPath`.

```
uri.Path = {'products' 'matlab' ''};
EP = uri.EncodedPath
```

```
EP = products/matlab/
```

`Query` — Query of URI
`matlab.net.QueryParameter` object | string

Query of URI, specified as a vector of `matlab.net.QueryParameter` objects or a string containing the encoded query with an optional leading `?` character.

`Fragment` — Direction to secondary resource
string | character vector

Direction to a secondary resource, specified as a string or a character vector. The `string` method percent-encodes characters that are not allowed in the fragment portion of a URI. When setting `Fragment`, do not encode the value.

Example: In the URI `https://www.mathworks.com/help/matlab/ref/weboptions.html#examples`, the `Fragment` property is `examples`.

`Absolute` — Whether URI is absolute (read-only)
`true` | `false`

Whether URI is absolute, specified as `true` or `false`. An absolute URI has a nonempty `Scheme` property. If the URI is not absolute, then it is relative. For a definition of

`absolute-URI`, see RFC 3986 Uniform Resource Identifier (URI): Generic Syntax — Section 4.3 Absolute URI.

The `Path` property in an absolute URI is always treated as an absolute path and the `EncodedPath` property always contains a leading `/` character. To send a message, the URI must be absolute and must also contain a nonempty `Host` property.

Data Types: `logical`

`EncodedAuthority` — Encoded authority portion of URI
string | character vector

Encoded authority portion of a URI, specified as a string or a character vector with associated punctuation appearing only if the property is nonempty. The format of `EncodedAuthority` is `UserInfo@Host:Port`. Setting `EncodedAuthority` is a shortcut to setting the `UserInfo`, `Host`, and `Port` properties, except that you must encode special characters.

Example: In the URI `https://user:pwd@www.mathworks.com:8000/product/matlab?abc=def&this=that#xyz`, the `EncodedAuthority` property is `user:pwd@www.mathworks.com:8000`.

`EncodedPath` — Encoded path
string | character vector

Encoded path, specified as a string or a character vector. Read this property to obtain the `Path` property as an encoded string as it would appear in the encoded URI. If you have an already-encoded path as a string, then set the `EncodedPath` property instead of the `Path` property to prevent further encoding. When reading `EncodedPath`, it has a leading `/` if `Path` is not `[]` and there are nonempty components in the URI before `Path`.

Setting `EncodedPath` to an empty array (`''`, `[]` or `string.empty`) is equivalent to setting `Path` to that value.

If there is no `Path` property in an encoded URI, then `EncodedPath` returns an empty string, `""`. However, `EncodedPath` is never an empty array.

Example: In the URI `https://www.mathworks.com/solutions/robotics`, the `EncodedPath` property is `/solutions/robotics`.

`EncodedQuery` — Encoded query
string | character vector

Encoded query, specified as a string or a character vector. `EncodedQuery` returns the same value as calling the `string` method on the `Query` property. Setting `EncodedQuery` is equivalent to setting the `Query` property.

Example: In the URI `https://www.mathworks.com/support/search_results.html?q=+weboptions+product:"MATLAB+Compiler"`, the `EncodedQuery` property is `q=+weboptions+product:%22MATLAB+Compiler%22`.

`EncodedURI` — Entire encoded URI
string | character vector

Entire encoded URI, specified as a string or a character vector. `EncodedURI` returns the same value as the `matlab.net.URI.string` method. Setting `EncodedURI` is equivalent to calling the `URI` constructor with the `'literal'` argument.

Methods

These methods implement the equivalent MATLAB functionality for this class.

`eq`	Compare URIs for equality. Two URIs are considered equal if they refer to the same resource. An empty string or empty `Path` property is considered equal to `[]`.
`char`	URI as character vector
`string`	URI as string

Examples

Search mathworks.com for Term `weboptions`

Create a URI.

```
U = matlab.net.URI('https://www.mathworks.com');
U.Query = matlab.net.QueryParameter('q','weboptions');
U.Path = 'search/site_search.html';
```

Display the search results containing `weboptions`.

```
web(char(U))
```

Create URI for Simulink Product Web Page

```
U = matlab.net.URI('//www.mathworks.com/products/simulink/');
U.EncodedURI
```

```
ans =

  string

    "//www.mathworks.com/products/simulink/"
```

See Also

QueryParameter | RequestMessage | webread | websave | webwrite

External Websites

RFC 3986 Uniform Resource Identifier (URI): Generic Syntax — Section 3.1. Scheme
RFC 3986 Uniform Resource Identifier (URI): Generic Syntax — Section 4.3 Absolute URI

Introduced in R2016b

matlab.net.base64decode

Base 64 decoding of string

Syntax

```
res = base64decode(V)
```

Description

`res = base64decode(V)` decodes `V`, which was encoded using base 64 encoding, and returns the decoded byte array. For more information, see RFC 4648 — Section 4, Base 64 Encoding. Base 64 encoding is used in some contexts in internet messages where arbitrary data must be transmitted in the form of ASCII characters.

In the input string, any characters not in the set of 65 characters defined for base 64 encoding are ignored. Decoding stops at the end of the string or at the first occurrence of an equal sign character (=).

If you know that the encoded data was a character string, you can convert it back to a string using the MATLAB `native2unicode(res)` function. This function uses the user default encoding in effect at the time the string was encoded.

Input Arguments

`V` — Text encoded using base 64
string | character vector

Text encoded using base 64, specified as a string or character vector.

See Also

`matlab.net.base64encode` | `native2unicode`

External Websites

RFC 4648 — Section 4, Base 64 Encoding

Introduced in R2016b

matlab.net.base64encode

Base 64 encoding of byte string or vector

Syntax

```
res = base64encode(V)
```

Description

`res = base64encode(V)` encodes `V` using base 64 encoding and returns the encoded characters as a string. For more information, see RFC 4648 — Section 4, Base 64 Encoding. Base 64 encoding is used in some contexts in internet messages where data must be transmitted in a limited set of ASCII characters. Use `base64encode` to encode strings containing special characters that might be misinterpreted as control characters by the transmission protocol. `base64encode` can encode arbitrary binary data.

If `V` is a string or character vector, `base64encode` first converts it to bytes using the user default encoding. To use a different character encoding, call the MATLAB `unicode2native` function to convert `V` to a `uint8` vector before passing it to `base64encode`.

Input Arguments

`V` — Data to encode
string | character vector | numeric vector

Data to encode, specified as a string, character vector, or numeric vector.

See Also

`matlab.net.base64decode` | `unicode2native`

External Websites

RFC 4648 — Section 4, Base 64 Encoding

Introduced in R2016b

matlab.net.http.io Package

Streaming content consumers and providers for HTTP messages

Description

Classes

matlab.net.http.io.BinaryConsumer	Consumer for binary data in HTTP messages
matlab.net.http.io.ContentConsumer	Consumer for HTTP message payloads
matlab.net.http.io.ContentProvider	ContentProvider for HTTP message payloads
matlab.net.http.io.FileConsumer	Consumer for files in HTTP messages
matlab.net.http.io.FileProvider	ContentProvider to send files
matlab.net.http.io.FormProvider	ContentProvider that sends form data
matlab.net.http.io.GenericConsumer	Consumer for multiple content types in HTTP messages
matlab.net.http.io.GenericProvider	Generic ContentProvider for HTTP payloads
matlab.net.http.io.ImageConsumer	Consumer for image data in HTTP payloads
matlab.net.http.io.ImageProvider	ContentProvider to send MATLAB image data
matlab.net.http.io.JSONConsumer	Content consumer that converts JSON input into MATLAB data
matlab.net.http.io.JSONProvider	ContentProvider to send MATLAB data as JSON string
matlab.net.http.io.MultipartConsumer	Helper for multipart content types in HTTP messages
matlab.net.http.io.MultipartFormProvider	ContentProvider to send multipart/form-data messages
matlab.net.http.io.MultipartProvider	ContentProvider to send multipart/mixed HTTP messages
matlab.net.http.io.StringConsumer	String consumer for HTTP payloads
matlab.net.http.io.StringProvider	ContentProvider to send MATLAB strings

See Also

Introduced in R2018a

matlab.net.http.io.BinaryConsumer class

Package: `matlab.net.http.io`
Superclasses: `matlab.net.http.io.ContentConsumer`

Consumer for binary data in HTTP messages

Description

Use a `BinaryConsumer` object to copy the raw payload to the `Response.Body.Data` property. A `BinaryConsumer` is the default consumer for a `GenericConsumer` if no specified consumer matches the type.

The `BinaryConsumer` class is provided for subclass authors to examine and process raw binary data while it is being received, possibly converting it to MATLAB array data to be stored in `Response.Body.Data`.

The `matlab.net.http.io.BinaryConsumer` class is a `handle` class.

Creation

`consumer = BinaryConsumer` constructs a consumer to receive binary data from a `ResponseMessage`.

You can create a subclass of this consumer to examine raw data as it is being received, and then delegate to this class to store the data in Response.Body.Data.

Properties

Public Properties

`AllocationLength` — Suggested buffer size
`uint64`

Suggested buffer size, specified as `uint64`. MATLAB sets `AllocationLength` to the anticipated size of buffers of data passed to `putData`. The actual size might be smaller or

larger. To improve performance, the consumer can use this value to preallocate space to handle the data.

MATLAB sets this property before calling the `start` method for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`ContentLength` — Expected length of payload
`uint64` | empty

Expected length of the payload, specified as `uint64`. The property normally is the `Value` property of the `matlab.net.http.field.ContentLengthField` in the `Header` property.

If `ContentLength` is empty, then the length is not known. The payload ends when `putData(uint8.empty)` is called.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses that might benefit from knowing the length of the data.

If this `ContentConsumer` is a delegate of a top-level consumer, then the value of `ContentLength` might be different from the `ContentLength` value of the top-level consumer.

Example: `numel(someData)` where `someData` is type `uint8`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`ContentType` — Media type of payload
`matlab.net.http.MediaType` | empty

Media type of payload, specified as a `matlab.net.http.MediaType` object. The property normally is the `Value` property of the `matlab.net.http.field.ContentTypeField` in the `Header` property. If the `ContentType` property is empty, then the `ContentTypeField` is empty or nonexistent.

MATLAB sets this property before calling `initialize` for the convenience of subclasses that might want to examine the `MediaType`. Subclasses can set this property if they determine from the data that it is of a different `MediaType`.

At the end of the transfer, MATLAB copies this value into the `Response.Body.ContentType` property.

Example: `'application/octet-stream'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Header` — Header of payload currently being processed
`matlab.net.http.HeaderField`

Header of the payload currently being processed, specified as a `matlab.net.http.HeaderField` object.

Consumers use this header to determine how to process the payload that is being sent to them. For a top-level consumer, this value is the same as `Response.Header`. For a delegate, the value might be different. For example, in a multipart message processed by a `MultipartConsumer`, it is the header of the part that this delegate is processing. The delegate can still examine `Response.Header` for headers of the original message.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Request` — Completed `RequestMessage` that was sent
`matlab.net.http.RequestMessage`

The completed `RequestMessage` that was sent, specified as a `matlab.net.http.RequestMessage` object. This is the final `RequestMessage` after all redirections, which is the `completedrequest` return value from the `send` method.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Response — ResponseMessage being processed
`matlab.net.http.ResponseMessage`

The `ResponseMessage` being processed, specified as a `matlab.net.http.ResponseMessage` object.

MATLAB sets the `Response` property before calling `initialize`. The value is the `ResponseMessage` after headers have been received but before receiving any payload. At the start of the response message processing (or the start of a part for multipart messages), the `ResponseMesssage.Body` property is a `MessageBody` object with empty `Data` and `Payload` properties. To store received data, consumers can modify the `Response` and `MessageBody.Data` properties during data transfer. Usually, consumers that process and then store data set `Response.Body.Data` to their processed payload, though this is not required. At the completion of the transfer, MATLAB returns this `Response` to the caller of `send`. Consumers should not modify other `Response` properties, such as `Header` or `StatusLine`, as those changes are returned to the caller of `send`.

The `Response.Body.Payload` property is empty during the transfer and consumers should not attempt to modify it. If the `HTTPOptions.SavePayload` property is set, then MATLAB sets `Payload` to the received payload at the end of the transfer of the message or the part (after the call to `putData(uint8.empty)`) or when an exception occurs.

If an exception occurs in the consumer during message processing, then MATLAB throws an `HTTPException` object. The `History` property contains this `Response` value.

If the consumer is a delegate that is processing part of a multipart message, then `Response.Header` contains the header of the whole message, and the `Payload` and `Data` properties of `Response.Body` are cleared before invoking the `ContentConsumer` for each part. At the conclusion of each part, a new `ResponseMessage` is added to the end of the array of `ResponseMessage` objects in the original response's `Body.Data` containing the `Header` from this object and the `Body` from this property. The next delegate sees a fresh `Response` with an empty `MessageBody`, not the previous delegate's `MessageBody`.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

URI — Destination of request being processed
matlab.net.URI

Destination of the request being processed, specified as a `matlab.net.URI` object. This value is the original destination URI as determined by `send`. It is not the URI of a proxy or the final URI after redirections.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Protected Properties

AppendFcn — Function called by putData to append additional data
function handle

Function, specified as a function handle, called by the `putData` method to append additional data. The `putData` method in this class calls the `AppendFcn` function to append data it receives in its `data` argument to existing data in the response message. The function must have the signature:

```
AppendFcn(consumer,newdata)
```

where `newdata` is the data to be appended to the array at `consumer.Response.Body.Data`. It is the responsibility of this method to update `consumer.CurrentLength` to reflect the new length of `Data`. If `newdata` is empty, which indicates the end of the stream, then the function should update `Response.Body.Data` to its final value.

The default behavior, if this property is empty, uses an internal function that treats `Data` as an array of arbitrary values supporting the `horzcat` function. It efficiently adds `newdata` by preallocating space, maintaining `CurrentLength` to be the actual length of

data stored. At the end of the message, it truncates `Response.Body.Data` to `CurrentLength`.

Subclasses can change this property if `horzcat` is not appropriate for the append process. For example, when a `StringConsumer` builds a scalar string, it would add to the string using the `plus` function instead of `horzcat`.

Subclasses that do not invoke `ContentConsumer.putData` to append data, or which are satisfied with `horzcat` behavior when appending data, can ignore this property.

Example: `@customAppend` where `@customAppend` is defined by the consumer

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`CurrentDelegate` — `ContentConsumer` to which this consumer is delegating
`matlab.net.http.io.ContentConsumer` | `[]`

The `ContentConsumer` to which this consumer is delegating, specified as a `matlab.net.http.io.ContentConsumer` object. The `delegateTo` method of the calling consumer (the delegator) sets the `CurrentDelegate` property. If there is no current delegation, then the value is `[]`.

MATLAB sets `CurrentDelegate` to `[]` before calling `initialize`.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`CurrentLength` — Length of data currently in `Response.Body.Data`
`uint64.empty` (default) | `uint64`

Length of data currently in the `Response.Body.Data` property, specified as `uint64`.

This property is used when `Response.Body.Data` has been preallocated to a size larger than the actual amount of data currently stored, to indicate the length of that stored data. If this property is empty, then it means that all of `Response.Body.Data` contains the stored data or that a `ContentConsumer` subclass is disposing of the data in some way other than storing it in `Response.Body.Data`.

This property is used and set by the `putData` method in this base class when the `AppendFcn` property is empty. It is for the benefit of subclasses that call `putData` and want to examine already-stored data, and/or any implementations of `AppendFcn` that maintain results in `Response.Body.Data`.

Subclasses that use `putData` also can modify this property to reset the position in the buffer where the data is stored. For example, when the default `AppendFcn` function is used, a subclass that processes all of `Response.Body.Data` on each call to `putData` might no longer have a use for the original data, so it can reset the `CurrentLength` property to 1 so that the next `putData` call overwrites the buffer with new data. There is no need to clear elements in the buffer past the end of the new data.

Subclasses that do not call `putData` can use this property to track their own data, or can leave it unset (empty). MATLAB does not place any constraints on the value that can be set here and does not use it for any purpose other than to determine where the default `AppendFcn` should store the next buffer of data, and where to truncate the data at the end of the message. Set this property to empty before the final call to `putData(uint8.empty)` to prevent truncation of the data.

MATLAB sets this property to empty before each call to `initialize`.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`MyDelegator` — `ContentConsumer` that delegated to this consumer
`matlab.net.http.io.ContentConsumer` | empty

The `ContentConsumer` that delegated to this consumer, specified as a `matlab.net.http.io.ContentConsumer` object. If this consumer is a delegate that was invoked by another consumer, such as a `GenericConsumer` or `MultipartConsumer`, then this is the calling consumer. It is empty in a top-level consumer specified in the call to `send`.

Delegates can use this property to access properties of their delegators, for example, to determine which consumer delegated to them.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

Methods

Public Methods

initialize	Prepare consumer for new HTTP payload
start	Start HTTP transfer from BinaryConsumer
putData	Save next buffer of data for BinaryConsumer
delegateTo	Delegate to another consumer

Examples

Process `uint8` Data

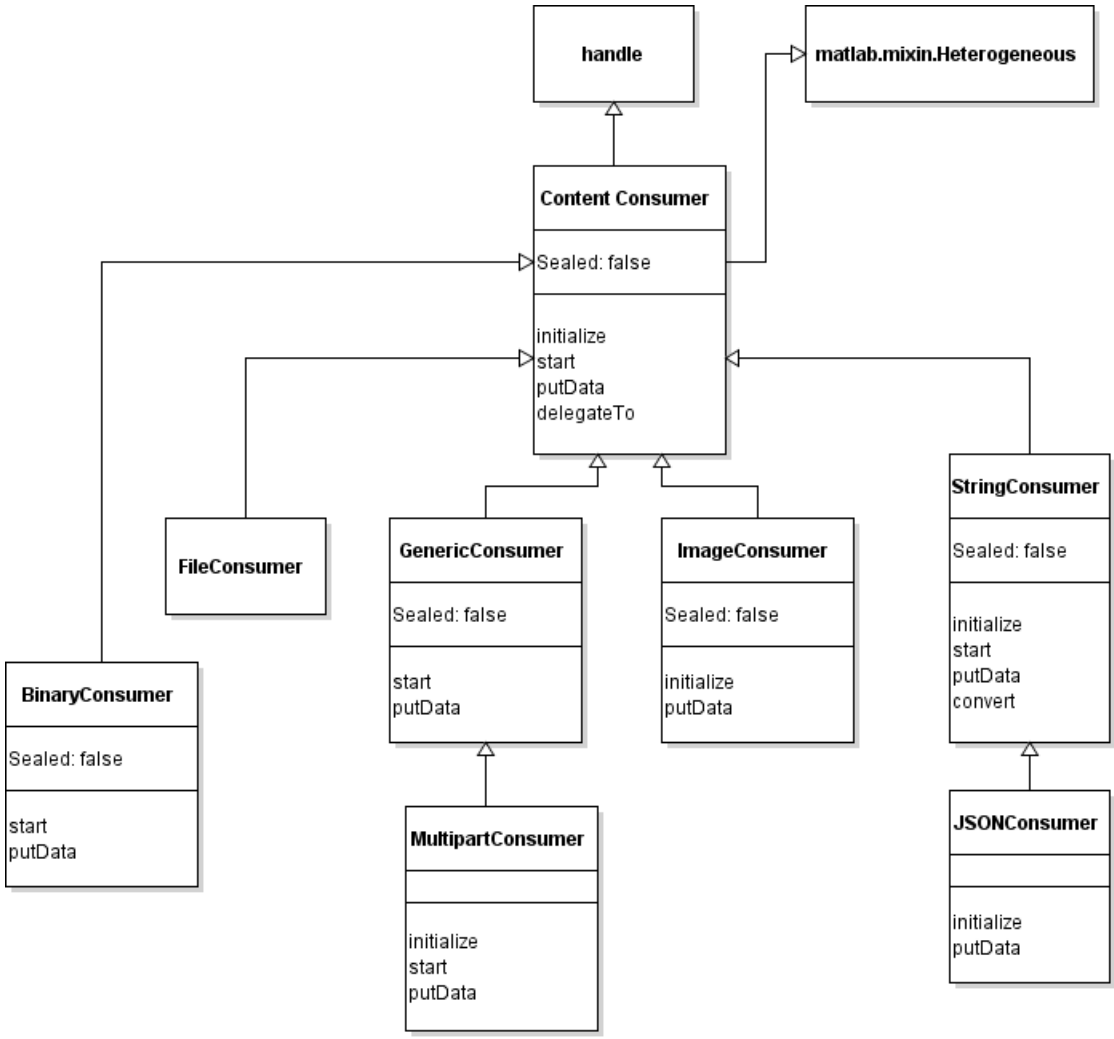
The following consumer combines each pair of `uint8` bytes received into an `int16`, and stores the `int16` array in the response.

```
classdef MyBinaryConsumer < matlab.net.http.io.BinaryConsumer
    properties
        ExtraByte uint8
    end
    methods
        function [len, stop] = putData(obj, data)
            if isempty(data)
                [len, stop] = obj.putData@matlab.net.http.io.BinaryConsumer(data);
            else
                if ~isempty(obj.ExtraByte)
                    data = [obj.ExtraByte; data];
                end
                len = length(data);
                if mod(len,2) > 0
                    obj.ExtraByte = data(end);
                    len = len - 1;
                else
                    obj.ExtraByte = uint8.empty;
                end
                res(1:len/2) = bitshift(uint16(data(1:2:len)),8)+uint16(data(2:2:len));
                [len, stop] = obj.putData@matlab.net.http.io.BinaryConsumer(res);
            end
        end
    end
    methods(Access=protected)
        function bs = start(obj)
            obj.ExtraByte = uint8.empty;
            bs = obj.start@matlab.net.http.io.BinaryConsumer();
        end
```

```
        end
    end
```

More About

Class Hierarchy



See Also

`ContentConsumer` | `GenericConsumer`

Introduced in R2018a

putData

Class: `matlab.net.http.io.BinaryConsumer`
Package: `matlab.net.http.io`

Save next buffer of data for BinaryConsumer

Syntax

```
[len,stop] = putData(consumer,data)
```

Description

`[len,stop] = putData(consumer,data)` stores the next buffer of data. This method is an overridden method of `putData`.

Input Arguments

`consumer` — Content consumer
`matlab.net.http.io.BinaryConsumer`

Content consumer, specified as a `matlab.net.http.io.BinaryConsumer` object.

`data` — Buffer of raw data in `ResponseMessage`
nonempty `uint8` vector | `uint8.empty` | `[]` (empty double)

Buffer of raw data in a `matlab.net.http.ResponseMessage` object, specified as a nonempty `uint8` vector, `uint8.empty`, or `[]`. For more information about these values, see the "data" on page 1-0 input argument for the `ContentConsumer.putData` method.

When MATLAB calls this function, it provides data as a `uint8` vector. This function appends the vector at the position `CurrentLength`+1 in `Response.Body.Data`, increasing the size of `data`, if necessary, to make room for future data.

If you call this method to store your own data, then you can provide data of any type that is compatible with data already in`Response.Body.Data`.

Output Arguments

`len` — Length of data processed
double | [ ] (empty double)

Length of data processed, returned as double or empty double. For more information, see the “size” on page 1-0 argument in `ContentConsumer.putData`.

`stop` — Indicate whether to receive further data
`false` (default) | `true`

Indicate whether to receive further data from this message, returned as `true` or `false`. For more information, see the “stop” on page 1-0 argument in `ContentConsumer.putData`.

Attributes

`Access`	`public`

See Also

ContentConsumer.CurrentLength | ContentConsumer.Response | `putData`

Introduced in R2018a

start

Class: `matlab.net.http.io.BinaryConsumer`
Package: `matlab.net.http.io`

Start HTTP transfer from BinaryConsumer

Syntax

```
bufsize = start(consumer)
```

Description

`bufsize = start(consumer)` starts receipt of data.

This method is an abstract method of `ContentConsumer` that is called by MATLAB. For more information, see the `start` method.

Input Arguments

`consumer` — Content consumer
`matlab.net.http.io.BinaryConsumer`

Content consumer, specified as a `matlab.net.http.io.BinaryConsumer` object.

Output Arguments

`bufsize` — Maximum buffer size
`[]`

Maximum buffer size, returned as `[]` to indicate that there is no preferred buffer size.

Attributes

Access	protected

See Also

start

Introduced in R2018a

matlab.net.http.io.ContentConsumer class

Package: `matlab.net.http.io`
Superclasses: `handle`, `matlab.mixin.Heterogeneous`

Consumer for HTTP message payloads

Description

A ContentConsumer is an object that converts or processes data received in an HTTP `ResponseMessage` object. MATLAB calls the consumer repeatedly during receipt of a response message to process buffers of the payload as it is being received. You can act on or display this streamed data while it is being received. You also can abort the transfer before receiving the entire message. Using a consumer can improve latency (the delay between receiving an instruction and the beginning of the transfer of data) when the time to process the data is comparable to the speed of the network. It also allows you to receive unbounded streamed response messages.

`ContentConsumer` is an abstract class. To process incoming data received in an HTTP response message, use one of the `ContentConsumer` subclasses:

- `FileConsumer`
- `StringConsumer`
- `JSONConsumer`
- `ImageConsumer`
- `MultipartConsumer`
- `BinaryConsumer`
- `GenericConsumer`

You also can write you own subclass of the `ContentConsumer` class or extend one of the subclasses.

The `matlab.net.http.io.ContentConsumer` class is a `handle` class.

Properties

Public Properties

`ContentLength` — Expected length of payload

`uint64` | empty

Expected length of the payload, specified as `uint64`. The property normally is the `Value` property of the `matlab.net.http.field.ContentLengthField` in the `Header` property.

If `ContentLength` is empty, then the length is not known. The payload ends when `putData(uint8.empty)` is called.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses that might benefit from knowing the length of the data.

If this `ContentConsumer` is a delegate of a top-level consumer, then the value of `ContentLength` might be different from the `ContentLength` value of the top-level consumer.

Example: `numel(someData)` where `someData` is type `uint8`

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`ContentType` — Media type of payload

`matlab.net.http.MediaType` | empty

Media type of payload, specified as a `matlab.net.http.MediaType` object. The property normally is the `Value` property of the `matlab.net.http.field.ContentTypeField` in the `Header` property. If the `ContentType` property is empty, then the `ContentTypeField` is empty or nonexistent.

MATLAB sets this property before calling `initialize` for the convenience of subclasses that might want to examine the `MediaType`. Subclasses can set this property if they determine from the data that it is of a different `MediaType`.

At the end of the transfer, MATLAB copies this value into the `Response.Body.ContentType` property.

Example: `'application/octet-stream'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`AllocationLength` — Suggested buffer size

`uint64`

Suggested buffer size, specified as `uint64`. MATLAB sets `AllocationLength` to the anticipated size of buffers of data passed to `putData`. The actual size might be smaller or larger. To improve performance, the consumer can use this value to preallocate space to handle the data.

MATLAB sets this property before calling the `start` method for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`URI` — Destination of request being processed

`matlab.net.URI`

Destination of the request being processed, specified as a `matlab.net.URI` object. This value is the original destination URI as determined by `send`. It is not the URI of a proxy or the final URI after redirections.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Request` — Completed `RequestMessage` that was sent

`matlab.net.http.RequestMessage`

The completed `RequestMessage` that was sent, specified as a `matlab.net.http.RequestMessage` object. This is the final `RequestMessage` after all redirections, which is the `completedrequest` return value from the `send` method.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Response` — `ResponseMessage` being processed
`matlab.net.http.ResponseMessage`

The `ResponseMessage` being processed, specified as a `matlab.net.http.ResponseMessage` object.

MATLAB sets the `Response` property before calling `initialize`. The value is the `ResponseMessage` after headers have been received but before receiving any payload. At the start of the response message processing (or the start of a part for multipart messages), the `ResponseMesssage.Body` property is a `MessageBody` object with empty `Data` and `Payload` properties. To store received data, consumers can modify the `Response` and `MessageBody.Data` properties during data transfer. Usually, consumers that process and then store data set `Response.Body.Data` to their processed payload, though this is not required. At the completion of the transfer, MATLAB returns this `Response` to the caller of `send`. Consumers should not modify other `Response` properties, such as `Header` or `StatusLine`, as those changes are returned to the caller of `send`.

The `Response.Body.Payload` property is empty during the transfer and consumers should not attempt to modify it. If the `HTTPOptions.SavePayload` property is set, then MATLAB sets `Payload` to the received payload at the end of the transfer of the message or the part (after the call to `putData(uint8.empty)`) or when an exception occurs.

If an exception occurs in the consumer during message processing, then MATLAB throws an `HTTPException` object. The `History` property contains this `Response` value.

If the consumer is a delegate that is processing part of a multipart message, then `Response.Header` contains the header of the whole message, and the `Payload` and `Data` properties of `Response.Body` are cleared before invoking the `ContentConsumer` for each part. At the conclusion of each part, a new `ResponseMessage` is added to the end of the array of `ResponseMessage` objects in the original response's `Body.Data` containing the `Header` from this object and the `Body` from this property. The next delegate sees a fresh `Response` with an empty `MessageBody`, not the previous delegate's `MessageBody`.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

`Header` — Header of payload currently being processed
`matlab.net.http.HeaderField`

Header of the payload currently being processed, specified as a `matlab.net.http.HeaderField` object.

Consumers use this header to determine how to process the payload that is being sent to them. For a top-level consumer, this value is the same as `Response.Header`. For a delegate, the value might be different. For example, in a multipart message processed by a `MultipartConsumer`, it is the header of the part that this delegate is processing. The delegate can still examine `Response.Header` for headers of the original message.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Protected Properties

`CurrentLength` — Length of data currently in `Response.Body.Data`
`uint64.empty` (default) | `uint64`

Length of data currently in the `Response.Body.Data` property, specified as `uint64`.

This property is used when `Response.Body.Data` has been preallocated to a size larger than the actual amount of data currently stored, to indicate the length of that stored data. If this property is empty, then it means that all of `Response.Body.Data` contains the stored data or that a `ContentConsumer` subclass is disposing of the data in some way other than storing it in `Response.Body.Data`.

This property is used and set by the `putData` method in this base class when the `AppendFcn` property is empty. It is for the benefit of subclasses that call `putData` and

want to examine already-stored data, and/or any implementations of `AppendFcn` that maintain results in `Response.Body.Data`.

Subclasses that use `putData` also can modify this property to reset the position in the buffer where the data is stored. For example, when the default `AppendFcn` function is used, a subclass that processes all of `Response.Body.Data` on each call to `putData` might no longer have a use for the original data, so it can reset the `CurrentLength` property to 1 so that the next `putData` call overwrites the buffer with new data. There is no need to clear elements in the buffer past the end of the new data.

Subclasses that do not call `putData` can use this property to track their own data, or can leave it unset (empty). MATLAB does not place any constraints on the value that can be set here and does not use it for any purpose other than to determine where the default `AppendFcn` should store the next buffer of data, and where to truncate the data at the end of the message. Set this property to empty before the final call to `putData(uint8.empty)` to prevent truncation of the data.

MATLAB sets this property to empty before each call to `initialize`.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`CurrentDelegate` — `ContentConsumer` to which this consumer is delegating
`matlab.net.http.io.ContentConsumer` | `[]`

The `ContentConsumer` to which this consumer is delegating, specified as a `matlab.net.http.io.ContentConsumer` object. The `delegateTo` method of the calling consumer (the delegator) sets the `CurrentDelegate` property. If there is no current delegation, then the value is `[]`.

MATLAB sets `CurrentDelegate` to `[]` before calling `initialize`.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`MyDelegator` — `ContentConsumer` that delegated to this consumer
`matlab.net.http.io.ContentConsumer` | empty

The `ContentConsumer` that delegated to this consumer, specified as a `matlab.net.http.io.ContentConsumer` object. If this consumer is a delegate that

was invoked by another consumer, such as a `GenericConsumer` or `MultipartConsumer`, then this is the calling consumer. It is empty in a top-level consumer specified in the call to `send`.

Delegates can use this property to access properties of their delegators, for example, to determine which consumer delegated to them.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`AppendFcn` — Function called by `putData` to append additional data
function handle

Function, specified as a function handle, called by the `putData` method to append additional data. The `putData` method in this class calls the `AppendFcn` function to append data it receives in its `data` argument to existing data in the response message. The function must have the signature:

```
AppendFcn(consumer,newdata)
```

where `newdata` is the data to be appended to the array at `consumer.Response.Body.Data`. It is the responsibility of this method to update `consumer.CurrentLength` to reflect the new length of `Data`. If `newdata` is empty, which indicates the end of the stream, then the function should update `Response.Body.Data` to its final value.

The default behavior, if this property is empty, uses an internal function that treats `Data` as an array of arbitrary values supporting the `horzcat` function. It efficiently adds `newdata` by preallocating space, maintaining `CurrentLength` to be the actual length of data stored. At the end of the message, it truncates `Response.Body.Data` to `CurrentLength`.

Subclasses can change this property if `horzcat` is not appropriate for the append process. For example, when a `StringConsumer` builds a scalar string, it would add to the string using the `plus` function instead of `horzcat`.

Subclasses that do not invoke `ContentConsumer.putData` to append data, or which are satisfied with `horzcat` behavior when appending data, can ignore this property.

Example: `@customAppend` where `@customAppend` is defined by the consumer

Attributes:

GetAccess	protected
SetAccess	protected

Methods

Public Methods

initialize	Prepare consumer for new HTTP payload
start	Start HTTP data transfer to ContentConsumer
putData	Process or save next buffer of data for ContentConsumer
delegateTo	Delegate to another consumer

More About

Top-Level Consumer

When you specify a `ContentConsumer` in the call to `RequestMessage.send`, the consumer is a top-level consumer. MATLAB passes the entire payload as it is being received, a buffer at a time. For example:

```
req = RequestMesage;
resp = req.send(url, [], MyConsumer);
```

Delegate

A consumer invoked by another consumer to handle all or part of the data in a message is a delegate.

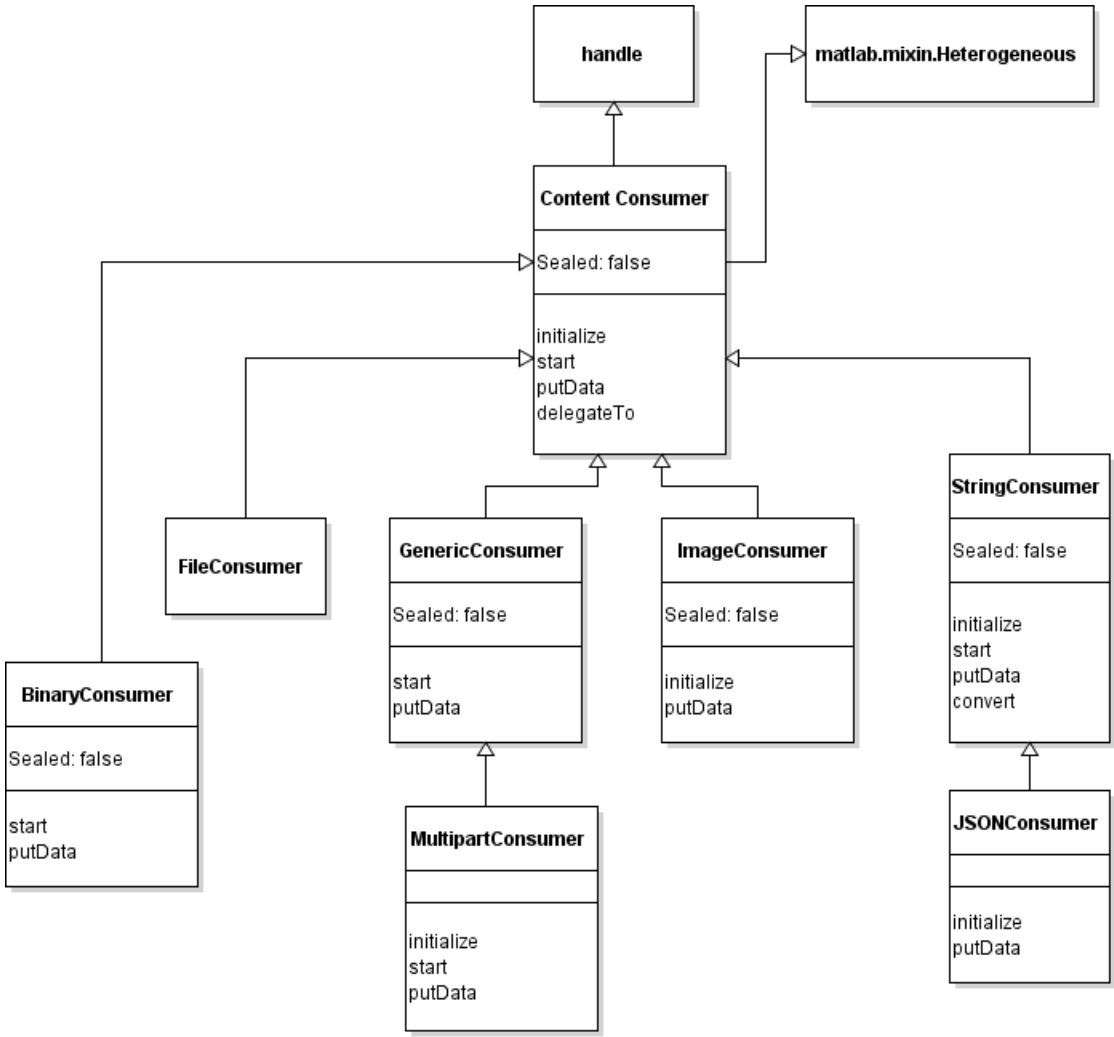
In the following code, `mp` is the top-level consumer that receives the entire payload of a multipart message.

```
mp = MultipartConsumer('image/*', ImageConsumer, 'text/*', StringConsumer);
resp = req.send(url, [], mp);
```

`ImageConsumer` and `StringConsumer` objects are delegate consumers that get only those parts of the payload that are images or text. A `GenericConsumer` also uses delegates. A `ContentConsumer` normally does not care whether it is a top-level

consumer or a delegate. Any consumer in the `matlab.net.http.io` package can work as a delegate.

Class Hierarchy



See Also

`BinaryConsumer` | `ContentLengthField` | `ContentProvider` | `ContentTypeField` | `FileConsumer` | `GenericConsumer` | `HeaderField` | `ImageConsumer` | `JSONConsumer` | `MediaType` | `MultipartConsumer` | `RequestMessage` | `ResponseMessage` | `StringConsumer` | `matlab.net.URI`

Introduced in R2018a

delegateTo

Class: `matlab.net.http.io.ContentConsumer`
Package: `matlab.net.http.io`

Delegate to another consumer

Syntax

```
[OK,bufsize] = delegateTo(consumer,delegate,header)
```

Description

`[OK,bufsize] = delegateTo(consumer,delegate,header)` prepares `delegate` as a consumer to process subsequent payloads based on `header`. `consumer` becomes the delegator and `delegate` becomes the delegate.

To prepare a delegate, `delegateTo` sets the following properties.

- Set `consumer.CurrentDelegate` to `delegate`
- Set `delegate.Header` to the value of `header`
- Set `delegate.MyDelegator` to `consumer`
- Set `delegate` properties, such as `Request`, `Response`, and `URI`, to the corresponding `consumer` properties

`delegateTo` then calls the delegate's `initialize` method. If `initialize` returns `true`, indicating that the delegate accepts the message, then `delegateTo` calls the `start` method.

Do not call `delegateTo` in a different delegate without telling the first delegate that the data has ended. To do this, call `delegate.putData(uint8.empty)`.

The delegate's `initialize` method sets `OK` and its `start` method sets `bufsize`. `bufsize` is valid only if `OK` is `true`. If `OK` is `false`, then the `start` method was not called and this consumer should not invoke `putData` in that delegate.

Input Arguments

consumer — Content consumer
matlab.net.http.io.ContentConsumer

Content consumer, specified as a `matlab.net.http.io.ContentConsumer` object.

delegate — Consumer to process payload
`matlab.net.http.io.ContentConsumer` | function handle

Consumer to process payload, specified as a `matlab.net.http.io.ContentConsumer` object or a handle to a function that returns a `ContentConsumer`. If a function, then it has no input arguments.

header — Header of payload
`matlab.net.http.HeaderField`

Header of the payload currently being processed, specified as a `matlab.net.http.HeaderField` object.

Output Arguments

OK — Indicate whether consumer accepts or rejects message payload
`true` (default) | `false`

Indicate whether the consumer accepts or rejects the payload of the message, returned as `true` or `false`. The default is `true` if the status code of the `ResponseMessage` is `StatusCode.OK`.

If `OK` is `true`, then the caller must call `delegate.putData` explicitly to feed data to the delegate, and, if desired, to copy any data that the delegate inserts in its `Response.Body` back into the caller's `Response`, on each call or before switching to a new delegate or at the end of the message.

If `OK` is `false`, or if the previous delegate has been told that the data has ended, then a consumer can call `delegateTo` to invoke another delegate for subsequent (or the same) data in the same message.

bufsize — Maximum buffer size
double

Maximum buffer size, returned as double.

Attributes

`Access`	`protected`

See Also

ContentConsumer.CurrentDelegate | ContentConsumer.Header | ContentConsumer.MyDelegator | `initialize` | `start`

Introduced in R2018a

initialize

Class: `matlab.net.http.io.ContentConsumer`
Package: `matlab.net.http.io`

Prepare consumer for new HTTP payload

Syntax

```
OK = initialize(consumer)
```

Description

`OK = initialize(consumer)` prepares a `ContentConsumer` for use with a new HTTP payload.

MATLAB calls `initialize` after receipt of the header of a `ResponseMessage` that might contain a payload, to prepare the consumer for that payload. It is not invoked for messages not expected to contain a payload, such as those with an explicit Content-Length of 0, or in error cases where a complete header was not received.

The `delegateTo` method in a delegate consumer also calls `initialize`.

If you subclass a `ContentConsumer`, then you should at least check the ContentType property to verify that the response is of the type you are prepared to handle. You can override this method to, for example:

- Initialize your own properties
- Determine if you want to process the payload
- Process a payload that has a `Response.StatusCode` other than `OK`

It is up to you whether to perform any subsequent initializations in this method or delay them until the `start` method.

Even if `initialize` is called, MATLAB might not call the consumer's `start` method if the message has no payload.

The default implementation returns `true` if the `Response.StatusCode` is `OK` and `false` otherwise. Subclasses that override this method should invoke this superclass method first and check the return value, unless they want to process messages with a status other than `OK`. Subclasses that invoke `putData` in this class must call this method.

Consumer subclasses should be prepared to be reused for subsequent messages. MATLAB calls `initialize` before each message and then `start` for each message that has a nonempty payload. Once MATLAB calls `start`, it does not call `initialize` until the message has ended, an exception was thrown, or an interrupt occurred during message processing. All these cases are indicated by a call to `putData(uint8.empty)`.

Input Arguments

`consumer` — Content consumer
`matlab.net.http.io.ContentConsumer`

Content consumer, specified as a `matlab.net.http.io.ContentConsumer` object.

Output Arguments

`OK` — Indicate whether consumer accepts or rejects message payload
`true` (default) | `false`

Indicate whether the consumer accepts or rejects the payload of the message, returned as `true` or `false`.

- If the status code of the `ResponseMessage` is `StatusCode.OK`, then the default is `true`.
- If `OK` is `true`, then the consumer accepted the message and processes the payload, if any. MATLAB then calls the consumer's `start` method when the first byte of the payload arrives, followed by one or more calls to `putData`, passing a buffer of data on each call.
- If `OK` is `false`, then the consumer does not want to process the message, in which case MATLAB processes the payload as if no consumer had been specified (which might mean default conversion of payload to data).

If you override the `initialize` method and reject the message and you want to abort receipt of the message instead of processing it in a default manner, then throw an error from this method instead of returning `false`.

Attributes

Access	protected

See Also

ContentConsumer.CurrentLength | ContentConsumer.Response | `delegateTo` | `matlab.net.http.RequestMessage` | `matlab.net.http.StatusCode` | `putData` | `start`

Introduced in R2018a

putData

Class: `matlab.net.http.io.ContentConsumer`
Package: `matlab.net.http.io`

Process or save next buffer of data for ContentConsumer

Syntax

```
[size,stop] = putData(consumer,data)
```

Description

`[size,stop] = putData(consumer,data)` provides a buffer of `data` read from the server to the `consumer`.

MATLAB calls `putData`. Subclass consumers can override this method to receive streamed data. Your consumer should return the length of data that it actually processed in `size`, and a true/false indication in `stop` to specify whether it wants to receive further data from this message.

When MATLAB calls `putData` with an empty `data` argument, it means that the message or message part in the case of a multipart message has ended.

If you create a subclass of a consumer that implements this method, your `putData` method can call its superclass `putData` to take advantage of any conversions or processing that the superclass implements.

MATLAB limits the size of `data` buffers to the `bufsize` value returned by the `start` method or an internal buffer size if `bufsize` is `[]`. Also, if the server sends a chunk-encoded message, then a given call to `putData` never provides more than one chunk. This allows the consumer to obtain slowly arriving chunks in a timely manner even if `bufsize` is much larger than the chunk size.

Default behavior of `putData()` in this base class:

Subclass consumers have the option of storing their possibly converted content directly in `Response.Body.Data`, either incrementally or all at once, or disposing of it in some

other way. The subclass consumer does not need to call this `putData` method to store data. As a convenience, consumers that want to store content incrementally in `Response.Body.Data` can call this method to do so. This method appends `data` to `Response.Body.Data` using the `AppendFcn`, attempting to do so efficiently by incrementally allocating capacity. The actual length of stored data is maintained in the `CurrentLength` property, which can be smaller than the actual length of `Response.Body.Data`. At the end of the transfer (for example, when `putData(consumer,[])` or `putData(consumer,uint8.empty)` is called, `Response.Body.Data` is truncated to `CurrentLength`. You can define your own `AppendFcn` to implement an alternative append method.

By default this method always returns `size` equal to the `numel(data)` and `stop` equal to `false`.

If you intend to use this method to store data and you know the maximum length of data to be stored, then you should set `Response.Body.Data` to a vector of the desired size filled with default values (for example, zeros), before calling this method for the first time. This method starts storing data at the beginning of your data area and then truncates it to the length of data at the end of the message, maintaining the length of data stored in `CurrentLength`.

Consumers that call this method in this base class to store data incrementally can provide `data` of any type that supports `horzcat` or `vertcat`, including structures and cell arrays. If you provide a cell array, the existing Data is converted to a cell array if it is not already, and elements of the cell array are inserted into the existing cell array at the linear index beginning at `CurrentLength+1`.

If you call this method in `ContentConsumer` to store data, then you should let this method manage `Response.Body.Data` or `CurrentLength` and not modify them directly.

`ContentConsumers` that call this method in their superclass should be prepared to do any cleanup, such as closing windows or deleting temporary files, if the superclass throws an exception.

Input Arguments

`consumer` — Content consumer
`matlab.net.http.io.ContentConsumer`

Content consumer, specified as a `matlab.net.http.io.ContentConsumer` object.

`data` — Buffer of raw data in `ResponseMessage`
nonempty `uint8` vector | `uint8.empty` | `[]` (empty double)

Buffer of raw data in a `matlab.net.http.ResponseMessage` object, specified as a one of the following:

- nonempty `uint8` vector - A normal buffer of data read from the server.
- `uint8.empty` - End of data. This is the normal way MATLAB indicates that the response message has ended. This is an indication for the consumer to clean up (for example, delete temporary files or truncate response data to current length) and be prepared for a possible future call to initialize for a subsequent message. In response, the consumer should return `stop=true` and `size=0` to indicate that processing was successful with no new bytes processed. If a consumer returns `size=[]`, then the consumer had a problem finalizing the data and MATLAB throws an `HTTPException` back to the caller of `RequestMessage.send`.
- `[]` (empty double) - The server, a network problem, or the user (using **Ctrl+C**) aborted the transfer. The consumer should generally clean up exactly as if `uint8.empty` was received, but some consumers might delete any incomplete data already received. On return from `putData`, MATLAB throws an `HTTPException` whose `History.Response` contains any data that the consumer stored in its `Response` property.

Most consumers that do not care about the difference between `[]` and `uint8.empty` can simply check `isempty(data)` and clean up appropriately. In every case where `data` is empty, consumers must call their superclass `putData` with that same empty value, even if they are not using their superclass `putData` to store data, as that is the only way the superclass knows to clean up. After receiving an empty value of `data`, implementations must ignore subsequent calls to `putData` with empty values, until the next call to `start`. Typically they should return `stop=false` and `size=0` on subsequent calls and not carry out any additional processing.

Output Arguments

`size` — Length of data processed
double | `[]` (empty double)

Length of data processed, returned as double or empty double. If the value of `size` is:

- `size >= 0, size <= length(data)` - The number of bytes of `data` processed by this call to `putData`. The number is used only for the benefit of subclasses of this consumer to know how much data was processed. It has no effect on future calls to `putData`. If `data` is empty, then `size` is ignored.
- `size < 0` - The result of `abs(size)` is the number of bytes processed. Same as the previous case, but in addition, MATLAB silently skips the remainder of the data, making one more call to `putData(uint8.empty)` at the end of the data. If a `MultipartConsumer` is not being used, this is similar to returning `stop=true`, except for the additional call to `putData` at the end. If `MultipartConsumer` is being used, a negative value of `size` only ends the part, and does not affect processing of subsequent parts of the message, so the connection is not closed until the next part or end of message is reached.
- `size = []` (empty double) - The consumer has decided something went wrong with the transfer and further transfers from the server should be terminated. This is similar to `stop=true`, but it is considered an error, causing a standard `HTTPException` to be thrown to the caller of `RequestMessage.send` indicating that the consumer aborted the connection. In this case, the only way the caller can get the partially processed `ResponseMessage` is through `HTTPException.History`.

  As an alternative to returning `size=[]` to throw a standard exception, your `putData` method can directly throw its own exception, which MATLAB wraps as a cause in an `HTTPException` returned to the caller of `RequestMessage.send`.

`stop` — Indicate whether to receive further data
`false` (default) | `true`

Indicate whether to receive further data from this message, specified as `true` or `false`. If `stop` is `true`, then the MATLAB stops processing the rest of the message, including any subsequent parts of a multipart message being processed by `MultipartConsumer`, silently proceeding as if the end of the message has been reached, even if the message has more data. This immediately closes the connection to the server and no error is returned to the caller of `RequestMessage.send`. This is not considered an error condition and is the normal way to terminate receipt of an arbitrary-length stream gracefully. If `stop=true` and `data` is not already empty, then MATLAB makes one additional call to `putData` with empty `data`. `stop=true` might be set whether or not `data` is empty.

Consumers should not normally set `stop=true` at the end of `data`, because, if they are multipart delegates, that would terminate processing for the rest of the message. To terminate processing just for their own part of the message, consumers should return `size < 0` to indicate that they do not want to receive more data for their part.

Attributes

Access	public

See Also

ContentConsumer.AppendFcn | ContentConsumer.CurrentLength | ContentConsumer.Response | `MultipartConsumer` | `matlab.net.http.HTTPException` | `matlab.net.http.HTTPOptions` | `matlab.net.http.RequestMessage` | `matlab.net.http.ResponseMessage` | `start`

Introduced in R2018a

start

Class: `matlab.net.http.io.ContentConsumer`
Package: `matlab.net.http.io`

Start HTTP data transfer to ContentConsumer

Syntax

```
bufsize = start(consumer)
```

Description

`bufsize = start(consumer)` signals the start of an HTTP transfer.

MATLAB calls `start` when it is ready to send data to the consumer, after it calls the `initialize` method that returned `true`. All consumers must implement this abstract method.

The `start` method differs from `initialize` in that it is invoked only if the message contains a payload. MATLAB calls `initialize` as soon as it receives the message header. Hence, it might be better to perform initializations in `start` rather than `initialize`, so that it is not done if the message is empty.

Input Arguments

`consumer` — Content consumer
`matlab.net.http.io.ContentConsumer`

Content consumer, specified as a `matlab.net.http.io.ContentConsumer` object.

Output Arguments

`bufsize` — Maximum buffer size
double | [ ]

Maximum buffer size that MATLAB passes to each call to the `putData` method, returned as double. The method returns the maximum size of the buffer, although MATLAB might pass in a smaller size.

`start` returns `[]` if the consumer can always process all of the data immediately, regardless of the size, to let MATLAB choose the best size. Specifying a smaller size is useful for slowly arriving data, as it allows you to receive data in a more timely manner rather than waiting for a large buffer to be filled.

If the server sends chunked-encoded messages, and if you want to be sure that each call to `putData` contains a whole chunk, then return `[]` or a value larger than the maximum chunk size. MATLAB never provides more than a single chunk at a time in one call to `putData`, but it might provide a part of a chunk if the chunk is larger than `bufsize` or MATLAB's internal buffer size.

Attributes

`Abstract`	`protected`

See Also

ContentConsumer.Response | `initialize` | `putData`

Introduced in R2018a

matlab.net.http.io.FileConsumer class

Package: `matlab.net.http.io`
Superclasses: `matlab.net.http.io.ContentConsumer`

Consumer for files in HTTP messages

Description

The `FileConsumer` class provides a convenient way to download a file from a web service, or to save data received from the web in a file. You can specify the name of the file, or let MATLAB determine the name from information sent by the server or file named in the URI.

The `matlab.net.http.io.FileConsumer` class is a `handle` class.

Creation

Description

`consumer = FileConsumer(filename,permission,machineformat,encoding)` constructs a `FileConsumer` that creates or overwrites a file with the payload of the response from the server. The parameters have the same meaning as those of the `fopen` function, and all are optional.

`consumer = FileConsumer(FID)` sets the `FileIdentifier` property to `FID` and writes to that file. `FID` must be the identifier of a file you opened for writing. MATLAB writes to the file at the current position indicator, so if you open an existing file using `'a+'` permission, for example, MATLAB appends to the file. When transfer is completed, MATLAB leaves the position indicator at the end of the file and does not close the file.

Input Arguments

`filename` — File or folder name
character vector | string scalar

File or folder name, including full path and optional extension, specified as a character vector or a string scalar. To determine the name of the file that MATLAB creates, see the `Filename` property.

If `filename` specifies a file in an existing folder, then MATLAB:

- Opens the file using `fopen(filename,permission,...)`.
- If `permission` is not specified, then opens the file using `fopen(filename,'w+')`.
- If `filename` does not include an extension, then MATLAB adds one based on the Content-Type and/or Content-Disposition header field in the received message or the extension of the file name in the URI of the request, if any.

If `filename` specifies an existing, writable folder, then MATLAB creates a file in the folder with a name derived from the Content-Disposition header field in the response or from the URI, possibly adding an extension based on Content-Type if that name does not contain one.

If `filename` is missing or empty, then MATLAB creates a file in the current folder. This is equivalent to `filename = '.'`. The current folder is the folder at the time this `FileConsumer` was created, not the time this consumer is used in a `send` request.

Example: `'myTextFile.txt'`

Data Types: `char` | `string`

`permission` — File access type

`w+` (default) | `u+` | `u` | `T` | value allowed by `fopen` function

File access type, specified as a string. If `permission` is specified, it must allow write access. The default value is `'w+'`, which opens or creates a file for reading and writing and discards existing contents, if any.

`permission` can be any value allowed by the `fopen` function. The following additional values of `permission` are supported:

`'u+'`, `'u'`	Same as `'w'` and `'w+'`, but if the file exists, then a file is created with a unique name derived from the `filename` argument. The name of the new file has a hyphen and a sequence number appended to the part of its name excluding the extension. For example, if `filename` is `'MyFile.txt'` but the file exists, then MATLAB creates the file `MyFile-1.txt`. The `Filename` property contains the new file's full path name.
`'T'`	When appended to a permission, behaves similarly to text mode `'t'` but uses text mode only if the Content-Type of the data indicates it is character-based. This includes any type specifying a charset parameter or types MATLAB knows to be character-based, such as `"application/json"`.

In all cases, for `'w'` and `'w+'` permissions (or if `permission` is not specified), MATLAB does not overwrite an existing file unless the file name is exactly equal to `filename`.

Example: `'a''w+T'`

Data Types: `char` | `string`

`machineformat` — Order for reading or writing bytes or bits
any value allowed by `fopen`

Order for reading or writing bytes or bits, specified as any value allowed by the `fopen` function.

Data Types: `char` | `string`

`encoding` — Character encoding
any value allowed by `fopen`

Character encoding, specified as any value allowed by the `fopen` function.

Data Types: `char` | `string`

Properties

Public Properties

`FileIdentifier` — File identifier

double

Identifier of the file (FID) being written, specified as double. If the consumer was constructed with an FID argument, then this property is the identifier. Data is written to the current file position indicator associated with this identifier, so subclasses should be careful not to change the position accidentally when using this identifier. At the conclusion of the transfer, the file remains open and the position remains at the end of the file.

If the constructor was called with a `filename` argument, or with no arguments, then this property is the read-only file identifier for that file. This allows subclasses to read the file during transfer without disturbing the position indicator used for writing. At the conclusion of the transfer, this identifier is closed.

Attributes:

```
GetAccess                                   public
SetAccess                                   private
```

`Filename` — File path name

`string.empty` (default) | string

Full path name of the file being written, specified as a string. If the consumer was constructed with an FID argument, then this property is the name of the file. Otherwise, this value might not be set until MATLAB has begun writing to the file during receipt of a response message, since the file name cannot necessarily be determined until all headers have been received. Use this property to determine the file that was written. `Filename` is also stored in the `Response.Body.Data` property.

Attributes:

```
GetAccess                                   public
SetAccess                                   private
```

`AllocationLength` — Suggested buffer size

`uint64`

Suggested buffer size, specified as `uint64`. MATLAB sets `AllocationLength` to the anticipated size of buffers of data passed to `putData`. The actual size might be smaller or larger. To improve performance, the consumer can use this value to preallocate space to handle the data.

MATLAB sets this property before calling the `start` method for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`ContentLength` — Expected length of payload

`uint64` | empty

Expected length of the payload, specified as `uint64`. The property normally is the `Value` property of the `matlab.net.http.field.ContentLengthField` in the `Header` property.

If `ContentLength` is empty, then the length is not known. The payload ends when `putData(uint8.empty)` is called.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses that might benefit from knowing the length of the data.

If this `ContentConsumer` is a delegate of a top-level consumer, then the value of `ContentLength` might be different from the `ContentLength` value of the top-level consumer.

Example: `numel(someData)` where `someData` is type `uint8`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`ContentType` — Media type of payload

`matlab.net.http.MediaType` | empty

Media type of payload, specified as a `matlab.net.http.MediaType` object. The property normally is the `Value` property of the `matlab.net.http.field.ContentTypeField` in the `Header` property. If the `ContentType` property is empty, then the `ContentTypeField` is empty or nonexistent.

MATLAB sets this property before calling `initialize` for the convenience of subclasses that might want to examine the `MediaType`. Subclasses can set this property if they determine from the data that it is of a different `MediaType`.

At the end of the transfer, MATLAB copies this value into the `Response.Body.ContentType` property.

Example: `'application/octet-stream'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Header` — Header of payload currently being processed
`matlab.net.http.HeaderField`

Header of the payload currently being processed, specified as a `matlab.net.http.HeaderField` object.

Consumers use this header to determine how to process the payload that is being sent to them. For a top-level consumer, this value is the same as `Response.Header`. For a delegate, the value might be different. For example, in a multipart message processed by a `MultipartConsumer`, it is the header of the part that this delegate is processing. The delegate can still examine `Response.Header` for headers of the original message.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Request` — Completed `RequestMessage` that was sent
`matlab.net.http.RequestMessage`

The completed `RequestMessage` that was sent, specified as a `matlab.net.http.RequestMessage` object. This is the final `RequestMessage` after all redirections, which is the `completedrequest` return value from the `send` method.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Response — ResponseMessage being processed
matlab.net.http.ResponseMessage

The `ResponseMessage` being processed, specified as a `matlab.net.http.ResponseMessage` object.

MATLAB sets the `Response` property before calling `initialize`. The value is the `ResponseMessage` after headers have been received but before receiving any payload. At the start of the response message processing (or the start of a part for multipart messages), the `ResponseMesssage.Body` property is a `MessageBody` object with empty `Data` and `Payload` properties. To store received data, consumers can modify the `Response` and `MessageBody.Data` properties during data transfer. Usually, consumers that process and then store data set `Response.Body.Data` to their processed payload, though this is not required. At the completion of the transfer, MATLAB returns this `Response` to the caller of `send`. Consumers should not modify other `Response` properties, such as `Header` or `StatusLine`, as those changes are returned to the caller of `send`.

The `Response.Body.Payload` property is empty during the transfer and consumers should not attempt to modify it. If the `HTTPOptions.SavePayload` property is set, then MATLAB sets `Payload` to the received payload at the end of the transfer of the message or the part (after the call to `putData(uint8.empty)`) or when an exception occurs.

If an exception occurs in the consumer during message processing, then MATLAB throws an `HTTPException` object. The `History` property contains this `Response` value.

If the consumer is a delegate that is processing part of a multipart message, then `Response.Header` contains the header of the whole message, and the `Payload` and `Data` properties of `Response.Body` are cleared before invoking the `ContentConsumer` for each part. At the conclusion of each part, a new `ResponseMessage` is added to the end of the array of `ResponseMessage` objects in the original response's `Body.Data` containing the `Header` from this object and the `Body` from this property. The next delegate sees a fresh `Response` with an empty `MessageBody`, not the previous delegate's `MessageBody`.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

URI — Destination of request being processed
matlab.net.URI

Destination of the request being processed, specified as a `matlab.net.URI` object. This value is the original destination URI as determined by `send`. It is not the URI of a proxy or the final URI after redirections.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Protected Properties

`AppendFcn` — Function called by `putData` to append additional data
function handle

Function, specified as a function handle, called by the `putData` method to append additional data. The `putData` method in this class calls the `AppendFcn` function to append data it receives in its `data` argument to existing data in the response message. The function must have the signature:

```
AppendFcn(consumer,newdata)
```

where `newdata` is the data to be appended to the array at `consumer.Response.Body.Data`. It is the responsibility of this method to update `consumer.CurrentLength` to reflect the new length of `Data`. If `newdata` is empty, which indicates the end of the stream, then the function should update `Response.Body.Data` to its final value.

The default behavior, if this property is empty, uses an internal function that treats `Data` as an array of arbitrary values supporting the `horzcat` function. It efficiently adds `newdata` by preallocating space, maintaining `CurrentLength` to be the actual length of

data stored. At the end of the message, it truncates `Response.Body.Data` to `CurrentLength`.

Subclasses can change this property if `horzcat` is not appropriate for the append process. For example, when a `StringConsumer` builds a scalar string, it would add to the string using the `plus` function instead of `horzcat`.

Subclasses that do not invoke `ContentConsumer.putData` to append data, or which are satisfied with `horzcat` behavior when appending data, can ignore this property.

Example: `@customAppend` where `@customAppend` is defined by the consumer

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`CurrentDelegate` — `ContentConsumer` to which this consumer is delegating
`matlab.net.http.io.ContentConsumer` | `[]`

The `ContentConsumer` to which this consumer is delegating, specified as a `matlab.net.http.io.ContentConsumer` object. The `delegateTo` method of the calling consumer (the delegator) sets the `CurrentDelegate` property. If there is no current delegation, then the value is `[]`.

MATLAB sets `CurrentDelegate` to `[]` before calling `initialize`.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`CurrentLength` — Length of data currently in `Response.Body.Data`
`uint64.empty` (default) | `uint64`

Length of data currently in the `Response.Body.Data` property, specified as `uint64`.

This property is used when `Response.Body.Data` has been preallocated to a size larger than the actual amount of data currently stored, to indicate the length of that stored data. If this property is empty, then it means that all of `Response.Body.Data` contains the stored data or that a `ContentConsumer` subclass is disposing of the data in some way other than storing it in `Response.Body.Data`.

This property is used and set by the `putData` method in this base class when the `AppendFcn` property is empty. It is for the benefit of subclasses that call `putData` and want to examine already-stored data, and/or any implementations of `AppendFcn` that maintain results in `Response.Body.Data`.

Subclasses that use `putData` also can modify this property to reset the position in the buffer where the data is stored. For example, when the default `AppendFcn` function is used, a subclass that processes all of `Response.Body.Data` on each call to `putData` might no longer have a use for the original data, so it can reset the `CurrentLength` property to 1 so that the next `putData` call overwrites the buffer with new data. There is no need to clear elements in the buffer past the end of the new data.

Subclasses that do not call `putData` can use this property to track their own data, or can leave it unset (empty). MATLAB does not place any constraints on the value that can be set here and does not use it for any purpose other than to determine where the default `AppendFcn` should store the next buffer of data, and where to truncate the data at the end of the message. Set this property to empty before the final call to `putData(uint8.empty)` to prevent truncation of the data.

MATLAB sets this property to empty before each call to `initialize`.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`MyDelegator` — `ContentConsumer` that delegated to this consumer
`matlab.net.http.io.ContentConsumer` | empty

The `ContentConsumer` that delegated to this consumer, specified as a `matlab.net.http.io.ContentConsumer` object. If this consumer is a delegate that was invoked by another consumer, such as a `GenericConsumer` or `MultipartConsumer`, then this is the calling consumer. It is empty in a top-level consumer specified in the call to `send`.

Delegates can use this property to access properties of their delegators, for example, to determine which consumer delegated to them.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

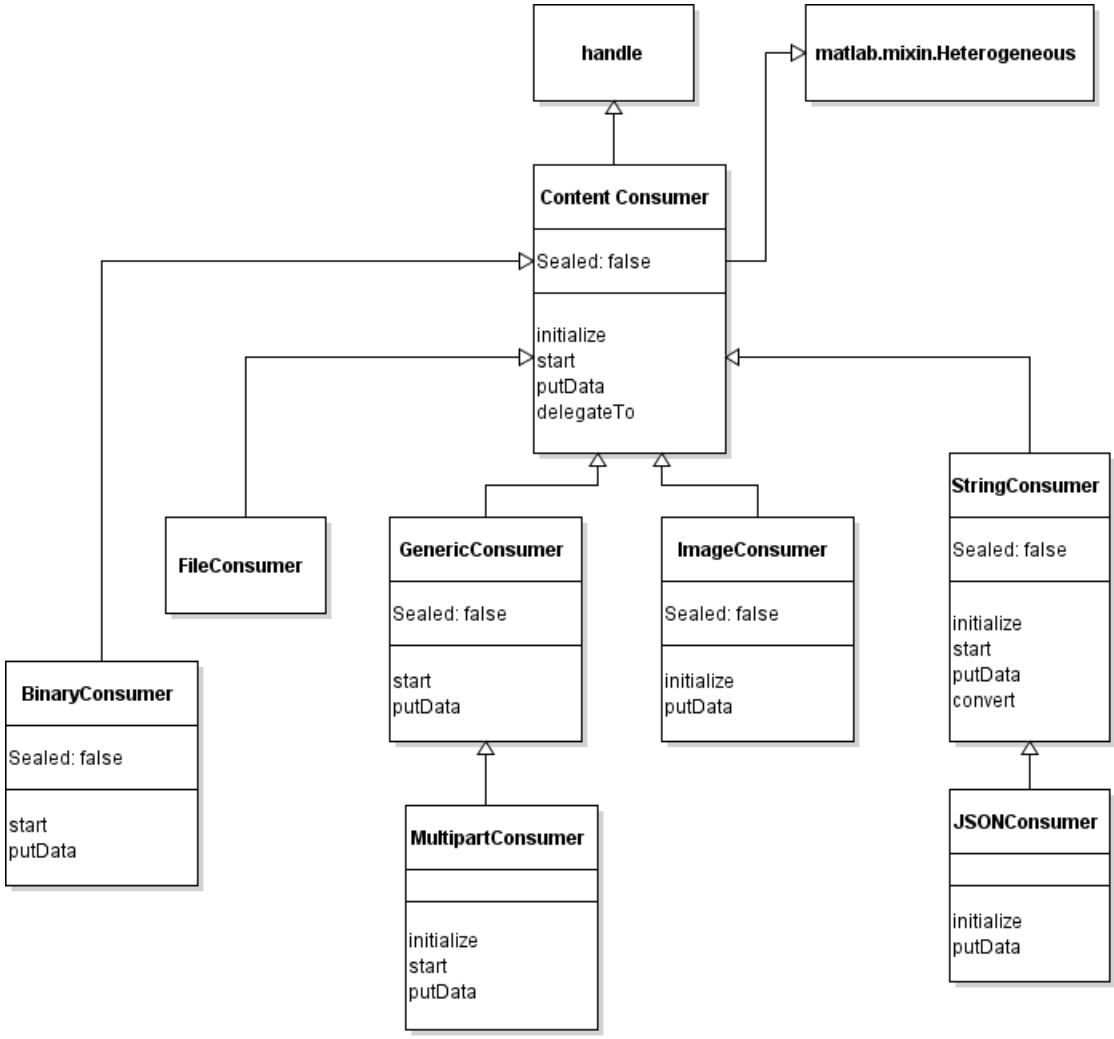
Methods

Public Methods

initialize	Prepare consumer for new HTTP payload
start	Start transfer of file to FileConsumer
putData	Save next buffer of data to file for FileConsumer
delegateTo	Delegate to another consumer

More About

Class Hierarchy



See Also

`ContentConsumer` | `ContentTypeField` | `FileProvider` | `RequestMessage` | `ResponseMessage` | `fopen`

Introduced in R2018a

putData

Class: `matlab.net.http.io.FileConsumer`
Package: `matlab.net.http.io`

Save next buffer of data to file for FileConsumer

Syntax

```
[size,stop] = putData(consumer,data)
```

Description

`[size,stop] = putData(consumer,data)` stores the next buffer of data read from the server, writes `data` to the file, and returns the number of bytes written. This method is an overridden method of `putData` called by MATLAB.

Input Arguments

`consumer` — Content consumer
`matlab.net.http.io.FileConsumer`

Content consumer, specified as a `matlab.net.http.io.FileConsumer` object.

`data` — Buffer of raw data in `ResponseMessage`
nonempty `uint8` vector | `uint8.empty` | `[]` (empty double)

Buffer of raw data in a `matlab.net.http.ResponseMessage` object, specified as a nonempty `uint8` vector, `uint8.empty`, or `[]`. For more information about these values, see the "data" on page 1-0 input argument for the `ContentConsumer.putData` method.

Output Arguments

`size` — Number of bytes written
double | [ ] (empty double)

Number of bytes written to the file, returned as double or empty double. For more information, see the “size” on page 1-0 argument in `putData`.

`stop` — Indicate whether to receive further data
`false` (default) | `true`

Indicate whether to receive further data from this message, returned as `true` or `false`. For more information, see the “stop” on page 1-0 argument in `ContentConsumer.putData`.

Attributes

Access	public

See Also

`putData`

Introduced in R2018a

start

Class: `matlab.net.http.io.FileConsumer`
Package: `matlab.net.http.io`

Start transfer of file to FileConsumer

Syntax

```
bufsize = start(consumer)
```

Description

`bufsize = start(consumer)` signals the start of the receipt of a file.

This method is an abstract method of `ContentConsumer` that MATLAB calls to start the receipt of the data. The method returns the maximum buffer size as a double. For more information, see the `start` method.

Input Arguments

`consumer` — Content consumer
`matlab.net.http.io.FileConsumer`

Content consumer, specified as a `matlab.net.http.io.FileConsumer` object.

Attributes

Access	protected

See Also

`start`

Introduced in R2018a

matlab.net.http.io.GenericConsumer class

Package: `matlab.net.http.io`
Superclasses: `matlab.net.http.io.ContentConsumer`

Consumer for multiple content types in HTTP messages

Description

Use this consumer to handle streaming for multiple content types when you cannot predict in advance which types the server returns.

The `matlab.net.http.io.GenericConsumer` class is a `handle` class.

Creation

Description

`consumer = GenericConsumer` constructs a `GenericConsumer` using default consumers.

`consumer = GenericConsumer(types,consumer)` constructs a `GenericConsumer` to handle the specified `types` using the corresponding `consumer`. You can specify several argument pairs in any order as `types1,consumer1,...,typesN,consumerN`. MATLAB searches the `types` in the order they appear and uses the first match. If there are no matches among the specified types, MATLAB uses the default set of consumers, depending on the type.

`consumer = GenericConsumer(puthandle)` constructs a `ContentConsumer` that calls the function specified by `puthandle` for each call to this consumer's `putData` method.

Input Arguments

`types` — Content types
string array | character vector | cell array of character vectors

Content types, specified as a string array, character vector, or cell array of character vectors. `types` specifies content types using the syntax `"type/subtype"`.

The type and subtype components in each element of `types` are treated as regular expressions, matched against the type/subtype of the `ContentTypeField` in the response, with the addition that a lone '*' for a type or subtype matches any type or subtype, and all searches are anchored to both the start and end of the string. For example,

Type/Subtype	Matches
`text/*`	Type `'text'` and any subtype
`*/.*json.*`	Any type with subtype that contains `'json'`
`*/.*json`	Any type with subtype that ends with `'json'`
`*/*`	Any type or subtype

If the subtype is `'*'`, you can omit the trailing `'/*'`. `text` is the same as `'text/*'`.

`types` are searched in order they appear, and the first match is used. If there are no matches among the specified types, a default set of consumers is used, depending on the type, in this order:

Type/Subtype	Default Consumer
`multipart/*`	`MultipartConsumer`
`image/*`	`ImageConsumer`
`.*/.*json.*`	`JSONConsumer`
`*/*`	`StringConsumer`
`*/*`	`BinaryConsumer`

While both `StringConsumer` and `BinaryConsumer` are used for any type, `StringConsumer` only accepts types for which it can determine a charset, specified as `text/*`, any type with a charset attribute, or one of the types MATLAB knows is character-based, such as `"application/xml"` and `"application/javascript"`. If `StringConsumer` rejects the type, then `BinaryConsumer` accepts the type and stores the unconverted payload in `Response.Body.Data` as a `uint8` vector.

When this consumer chooses a matching delegate based on this search criteria, it invokes the delegate's `initialize` method to see if the delegate accepts the payload. If the method returns `false` to indicate that the delegate does not accept, then this consumer continues searching the list to find the next matching delegate.

`consumer` — Content consumer
`matlab.net.http.io.ContentConsumer` | function handle

Content consumer, specified as a `matlab.net.http.io.ContentConsumer` object that can handle one of the specified types, or a handle to a function returning a `ContentConsumer` that can handle those types.

`puthandle` — putData function
function handle

Handle to a function with the following `ContentConsumer.putData` syntax:

```
[length,stop] = putData(data)
```

where *`data`* a `uint8` array and *`length`* is the length of that array.

Use this syntax to process all input from the server using a single function, when you know the type of data that the server returns. The function does not have access to the `ResponseMessage` or any information about this consumer.

Properties

Public Properties

`AllocationLength` — Suggested buffer size
`uint64`

Suggested buffer size, specified as `uint64`. MATLAB sets `AllocationLength` to the anticipated size of buffers of data passed to `putData`. The actual size might be smaller or larger. To improve performance, the consumer can use this value to preallocate space to handle the data.

MATLAB sets this property before calling the `start` method for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`ContentLength` — Expected length of payload
`uint64` | empty

Expected length of the payload, specified as `uint64`. The property normally is the `Value` property of the `matlab.net.http.field.ContentLengthField` in the `Header` property.

If `ContentLength` is empty, then the length is not known. The payload ends when `putData(uint8.empty)` is called.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses that might benefit from knowing the length of the data.

If this `ContentConsumer` is a delegate of a top-level consumer, then the value of `ContentLength` might be different from the `ContentLength` value of the top-level consumer.

Example: `numel(someData)` where `someData` is type `uint8`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`ContentType` — Media type of payload
`matlab.net.http.MediaType` | empty

Media type of payload, specified as a `matlab.net.http.MediaType` object. The property normally is the `Value` property of the `matlab.net.http.field.ContentTypeField` in the `Header` property. If the `ContentType` property is empty, then the `ContentTypeField` is empty or nonexistent.

MATLAB sets this property before calling `initialize` for the convenience of subclasses that might want to examine the `MediaType`. Subclasses can set this property if they determine from the data that it is of a different `MediaType`.

At the end of the transfer, MATLAB copies this value into the `Response.Body.ContentType` property.

Example: `'application/octet-stream'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Header` — Header of payload currently being processed
`matlab.net.http.HeaderField`

Header of the payload currently being processed, specified as a `matlab.net.http.HeaderField` object.

Consumers use this header to determine how to process the payload that is being sent to them. For a top-level consumer, this value is the same as `Response.Header`. For a delegate, the value might be different. For example, in a multipart message processed by a `MultipartConsumer`, it is the header of the part that this delegate is processing. The delegate can still examine `Response.Header` for headers of the original message.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Request` — Completed `RequestMessage` that was sent
`matlab.net.http.RequestMessage`

The completed `RequestMessage` that was sent, specified as a `matlab.net.http.RequestMessage` object. This is the final `RequestMessage` after all redirections, which is the `completedrequest` return value from the `send` method.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Response` — `ResponseMessage` being processed
`matlab.net.http.ResponseMessage`

The `ResponseMessage` being processed, specified as a `matlab.net.http.ResponseMessage` object.

MATLAB sets the `Response` property before calling `initialize`. The value is the `ResponseMessage` after headers have been received but before receiving any payload. At the start of the response message processing (or the start of a part for multipart messages), the `ResponseMesssage.Body` property is a `MessageBody` object with empty `Data` and `Payload` properties. To store received data, consumers can modify the `Response` and `MessageBody.Data` properties during data transfer. Usually, consumers that process and then store data set `Response.Body.Data` to their processed payload, though this is not required. At the completion of the transfer, MATLAB returns this `Response` to the caller of `send`. Consumers should not modify other `Response` properties, such as `Header` or `StatusLine`, as those changes are returned to the caller of `send`.

The `Response.Body.Payload` property is empty during the transfer and consumers should not attempt to modify it. If the `HTTPOptions.SavePayload` property is set, then MATLAB sets `Payload` to the received payload at the end of the transfer of the message or the part (after the call to `putData(uint8.empty)`) or when an exception occurs.

If an exception occurs in the consumer during message processing, then MATLAB throws an `HTTPException` object. The `History` property contains this `Response` value.

If the consumer is a delegate that is processing part of a multipart message, then `Response.Header` contains the header of the whole message, and the `Payload` and `Data` properties of `Response.Body` are cleared before invoking the `ContentConsumer` for each part. At the conclusion of each part, a new `ResponseMessage` is added to the end of the array of `ResponseMessage` objects in the original response's `Body.Data` containing the `Header` from this object and the `Body` from this property. The next delegate sees a fresh `Response` with an empty `MessageBody`, not the previous delegate's `MessageBody`.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true

URI — Destination of request being processed
`matlab.net.URI`

Destination of the request being processed, specified as a `matlab.net.URI` object. This value is the original destination URI as determined by `send`. It is not the URI of a proxy or the final URI after redirections.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Protected Properties

`PutMethod` — `putData` method

function handle | `[]`

`putData` method of the delegate, specified as a function handle, or specified as `[]` if `CurrentDelegate` is set.

This property is set by `delegateTo`. Subclasses should invoke this function in their `putData` method to send data to the delegate or to end the delegate's portion of the data by sending `uint8.empty`:

```
[len, stop] = obj.PutMethod(data);
```

At the end of the message, after the above call to any delegate to end the message, subclasses should set `PutMethod` to empty and invoke the call `putData(uint8.empty)` in their superclass so that this class knows the message has ended. This `putData` call will leave `PutMethod` empty or set it back to the `puthandle` argument passed to the constructor.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
Dependent                           true
```

`AppendFcn` — Function called by `putData` to append additional data

function handle

Function, specified as a function handle, called by the `putData` method to append additional data. The `putData` method in this class calls the `AppendFcn` function to

append data it receives in its `data` argument to existing data in the response message. The function must have the signature:

```
AppendFcn(consumer,newdata)
```

where `newdata` is the data to be appended to the array at `consumer.Response.Body.Data`. It is the responsibility of this method to update `consumer.CurrentLength` to reflect the new length of `Data`. If `newdata` is empty, which indicates the end of the stream, then the function should update `Response.Body.Data` to its final value.

The default behavior, if this property is empty, uses an internal function that treats `Data` as an array of arbitrary values supporting the `horzcat` function. It efficiently adds `newdata` by preallocating space, maintaining `CurrentLength` to be the actual length of data stored. At the end of the message, it truncates `Response.Body.Data` to `CurrentLength`.

Subclasses can change this property if `horzcat` is not appropriate for the append process. For example, when a `StringConsumer` builds a scalar string, it would add to the string using the `plus` function instead of `horzcat`.

Subclasses that do not invoke `ContentConsumer.putData` to append data, or which are satisfied with `horzcat` behavior when appending data, can ignore this property.

Example: `@customAppend` where `@customAppend` is defined by the consumer

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`CurrentDelegate` — `ContentConsumer` to which this consumer is delegating
`matlab.net.http.io.ContentConsumer` | `[]`

The `ContentConsumer` to which this consumer is delegating, specified as a `matlab.net.http.io.ContentConsumer` object. The `delegateTo` method of the calling consumer (the delegator) sets the `CurrentDelegate` property. If there is no current delegation, then the value is `[]`.

MATLAB sets `CurrentDelegate` to `[]` before calling `initialize`.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`CurrentLength` — Length of data currently in `Response.Body.Data`

`uint64.empty` (default) | `uint64`

Length of data currently in the `Response.Body.Data` property, specified as `uint64`.

This property is used when `Response.Body.Data` has been preallocated to a size larger than the actual amount of data currently stored, to indicate the length of that stored data. If this property is empty, then it means that all of `Response.Body.Data` contains the stored data or that a `ContentConsumer` subclass is disposing of the data in some way other than storing it in `Response.Body.Data`.

This property is used and set by the `putData` method in this base class when the `AppendFcn` property is empty. It is for the benefit of subclasses that call `putData` and want to examine already-stored data, and/or any implementations of `AppendFcn` that maintain results in `Response.Body.Data`.

Subclasses that use `putData` also can modify this property to reset the position in the buffer where the data is stored. For example, when the default `AppendFcn` function is used, a subclass that processes all of `Response.Body.Data` on each call to `putData` might no longer have a use for the original data, so it can reset the `CurrentLength` property to 1 so that the next `putData` call overwrites the buffer with new data. There is no need to clear elements in the buffer past the end of the new data.

Subclasses that do not call `putData` can use this property to track their own data, or can leave it unset (empty). MATLAB does not place any constraints on the value that can be set here and does not use it for any purpose other than to determine where the default `AppendFcn` should store the next buffer of data, and where to truncate the data at the end of the message. Set this property to empty before the final call to `putData(uint8.empty)` to prevent truncation of the data.

MATLAB sets this property to empty before each call to `initialize`.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`MyDelegator` — `ContentConsumer` that delegated to this consumer
`matlab.net.http.io.ContentConsumer` | empty

The `ContentConsumer` that delegated to this consumer, specified as a `matlab.net.http.io.ContentConsumer` object. If this consumer is a delegate that was invoked by another consumer, such as a `GenericConsumer` or `MultipartConsumer`, then this is the calling consumer. It is empty in a top-level consumer specified in the call to `send`.

Delegates can use this property to access properties of their delegators, for example, to determine which consumer delegated to them.

Attributes:

`GetAccess`	`protected`
`SetAccess`	`protected`

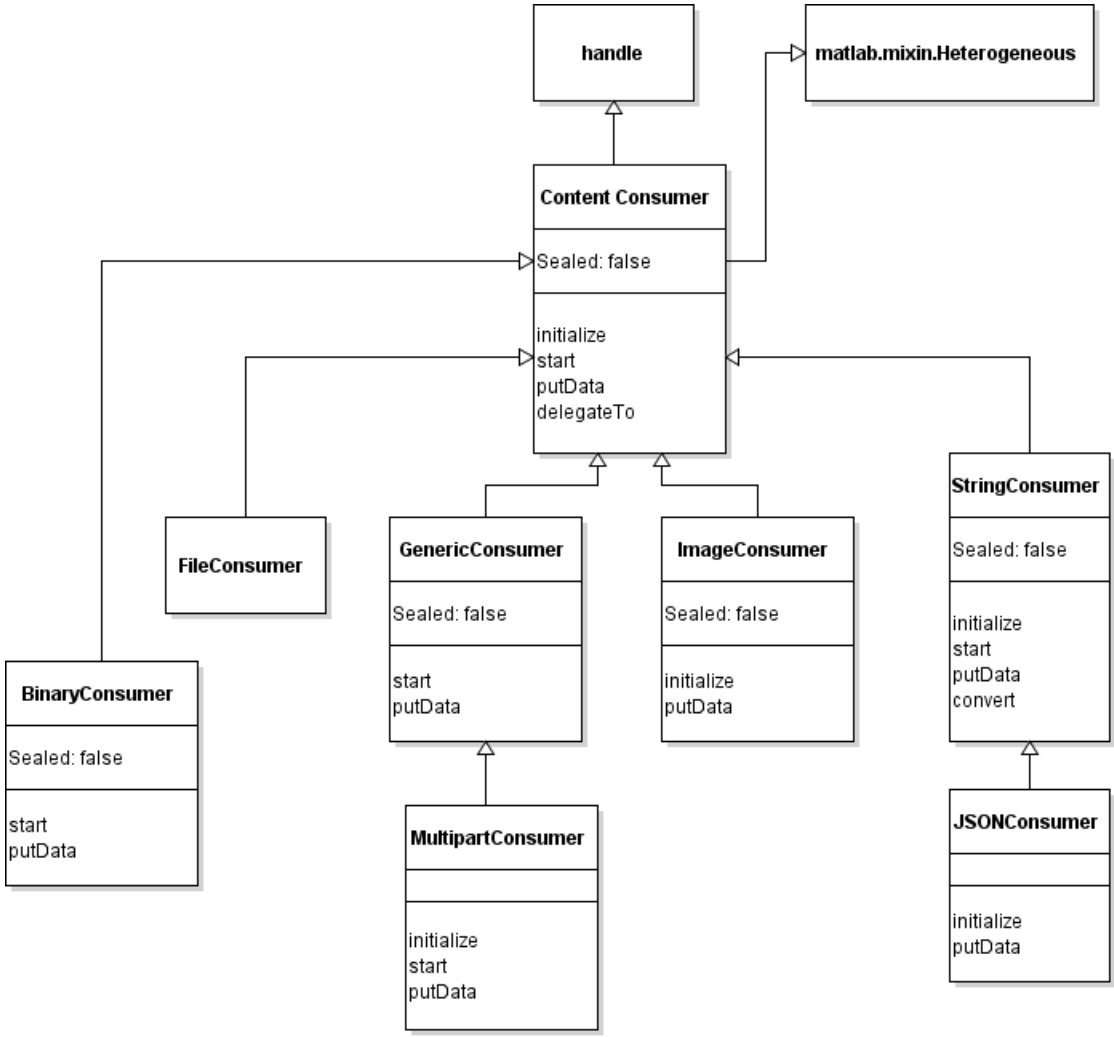
Methods

Public Methods

initialize	Prepare consumer for new HTTP payload
start	Start data transfer to GenericConsumer
putData	Save next buffer of data for GenericConsumer
delegateTo	Delegate to another consumer

More About

Class Hierarchy



See Also

`ContentConsumer` | `ContentTypeField` | `FileProvider` | `RequestMessage` | `ResponseMessage` | `fopen`

Introduced in R2018a

putData

Class: `matlab.net.http.io.GenericConsumer`
Package: `matlab.net.http.io`

Save next buffer of data for GenericConsumer

Syntax

```
[len,stop] = putData(consumer,data)
```

Description

`[len,stop] = putData(consumer,data)` stores the next buffer of data. This method is an overridden method of `putData`.

`putData` determines what method to call based on the following:

- If this `GenericConsumer` was created using the `GenericConsumer(puthandle)` constructor or `delegateTo` returned a delegate that accepted the message, then `putData` calls the function in GenericConsumer.PutMethod with `data` as an argument.
- If there was consumer that accepted the message, then `putData` calls `putData`, which appends `data` to `Response.Body.Data`.

Input Arguments

`consumer` — Content consumer
`matlab.net.http.io.GenericConsumer`

Content consumer, specified as a `matlab.net.http.io.GenericConsumer` object.

`data` — Buffer of raw data in `ResponseMessage`
nonempty `uint8` vector | `uint8.empty` | `[]` (empty double)

Buffer of raw data in a `matlab.net.http.ResponseMessage` object, specified as a nonempty `uint8` vector, `uint8.empty`, or `[]`. For more information about these values, see the "data" on page 1-0 input argument for the `ContentConsumer.putData` method.

If `data` is `[]` to indicate that the message has ended and there was a delegate, then `putData` copies the `Response` from the delegate to this object's `Response`.

Output Arguments

`len` — Length of data processed
double | `[]` (empty double)

Length of data processed, returned as double or empty double. For more information, see the "size" on page 1-0 argument in `ContentConsumer.putData`.

`stop` — Indicate whether to receive further data
`false` (default) | `true`

Indicate whether to receive further data from this message, returned as `true` or `false`. For more information, see the "stop" on page 1-0 argument in `ContentConsumer.putData`.

Attributes

Access	public

See Also

ContentConsumer.Response | GenericConsumer.PutMethod | `delegateTo` | `putData`

Introduced in R2018a

start

Class: `matlab.net.http.io.GenericConsumer`
Package: `matlab.net.http.io`

Start data transfer to GenericConsumer

Syntax

```
bufsize = start(consumer)
```

Description

`bufsize = start(consumer)` returns the maximum buffer size for calls to the `putData` method.

This method is an abstract method of `start` that prepares a `consumer` for receipt of data. If the `GenericConsumer` object was creating using the `putDataHandle` argument, then `start` does nothing and returns `[]`. Otherwise, `start` determines which `ContentConsumer` to delegate to, based on the ContentType property and `types` argument used by the object constructor. If the delegate was specified as a function handle returning a consumer instance, then `start` calls the function to obtain a delegate consumer instance. `start` then calls the `delegateTo` method, passing in the consumer instance, which calls the `initialize` method in that consumer. If `initialize` returns `false` to indicate it does not accept the message, then `start` tries the next delegate in the list. If a delegate accepts, then it calls `start` in that delegate. In that case, the caller of `start` is obligated to send that delegate the data from the message, or terminate the delegate by calling its `putData(uint8.empty)` method. If all delegates reject the message, then `start` throws an exception.

When a delegate accepts the message, `start` saves the delegate instance for that Content-Type. Therefore, if this method is called again with a Content-Type that matches the same `types` entry, then `start` uses the same delegate instance.

For more information, see `start`.

Input Arguments

`consumer` — Generic consumer
`matlab.net.http.io.GenericConsumer`

Generic consumer, specified as a `matlab.net.http.io.GenericConsumer` object.

Output Arguments

`bufsize` — Maximum buffer size
double | `[]`

Maximum buffer size, returned as double. If the GenericConsumer object was creating using the `putDataHandle` argument, then `start` does nothing and returns `[]`.

Attributes

Access	`protected`

See Also

ContentConsumer.ContentType | ContentConsumer.Response | `delegateTo` | `initialize` | `start`

Introduced in R2018a

matlab.net.http.io.ImageConsumer class

Package: `matlab.net.http.io`
Superclasses: `matlab.net.http.io.ContentConsumer`

Consumer for image data in HTTP payloads

Description

This consumer reads image data from the web and converts it to MATLAB image data. It stores the result in the `Body` property of the `ResponseMessage` to which it is applied. Specified directly as a consumer in `RequestMessage.send`, it provides the same functionality for image content types that is provided by the default `send` method when no consumer is specified, saving the converted image, plus a possible colormap and alpha channel, in `Body.Data`, based on the Content-Type of the message. For more information on conversion of image data in a response, see the description for `image/*` Content-Types in “HTTP Data Type Conversion”.

This consumer only accepts data for which it can determine a format based on headers in the response message or the extension of the file name in the URI of the request (if any). You can override this behavior by specifying the expected format in the `Format` property.

This consumer returns the data as a MATLAB image in one of the formats described for the return value of the `imread` function. To store the original data in a file without converting it, use a `FileConsumer`.

The `matlab.net.http.io.ImageConsumer` class is a `handle` class.

Creation

Description

`consumer = ImageConsumer` constructs a consumer for HTTP images.

`consumer = ImageConsumer(fmt)` constructs an `ImageConsumer` using the specified format.

`consumer = ImageConsumer( ___ ,arg1,...,argn)` passes additional arguments used by `imread` to convert the response data.

Input Arguments

`fmt` — Image format
string | character vector

Image format, specified as a string or a character vector, which is accepted by `imread`. This value overrides any format specification in the message header. This constructor sets the `Format` property to the value of `fmt`.

Example: `'png'`

`arg1,...,argn` — `imread` arguments
valid types

One or more input arguments to the `imread` function, specified as valid argument types. Do not specify `filename` or `fmt` arguments.

Example: `'PixelRegion',{[1 100],[4 500]}`

Properties

Public Properties

`Info` — Image information
structure

Image information as required for the `imfinfo` function, specified as a structure. MATLAB sets this property only after a successful conversion. The `Filename` field in this structure is empty.

Attributes:

`GetAccess`	`public`
`SetAccess`	`private`

`PartialData` — Partial image data
uint16

Partial image data, specified as `uint16`. If the conversion failed or the transfer was interrupted, then MATLAB sets `PartialData` to the raw received data, which is a `uint8` vector. `PartialData` might not contain any data during a transfer or in successful cases.

Attributes:

```
GetAccess                                   public
SetAccess                                   private
```

`Format` — Image format

empty (default) | string | character vector

Image format, specified as a string or character vector. This consumer rejects messages whose format is not one of those specified in the EXT column of the `imformats` function. The default value of this property is empty, which attempts to derive the format from the Content-Type field or the extension of the file name in the Content-Disposition field of the response, or the URI of the request. If you want to force this consumer to process the data using a specific format, set this property before applying this consumer to a `RequestMessage.send` method.

If the server does not properly indicate the format, set this property to the correct format.

Attributes:

```
GetAccess                                     public
SetAccess                                     public
```

`AllocationLength` — Suggested buffer size

`uint64`

Suggested buffer size, specified as `uint64`. MATLAB sets `AllocationLength` to the anticipated size of buffers of data passed to `putData`. The actual size might be smaller or larger. To improve performance, the consumer can use this value to preallocate space to handle the data.

MATLAB sets this property before calling the `start` method for the convenience of subclasses.

Attributes:

```
GetAccess                                     public
SetAccess                                     public
```

`ContentLength` — Expected length of payload
`uint64` | empty

Expected length of the payload, specified as `uint64`. The property normally is the `Value` property of the `matlab.net.http.field.ContentLengthField` in the `Header` property.

If `ContentLength` is empty, then the length is not known. The payload ends when `putData(uint8.empty)` is called.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses that might benefit from knowing the length of the data.

If this `ContentConsumer` is a delegate of a top-level consumer, then the value of `ContentLength` might be different from the `ContentLength` value of the top-level consumer.

Example: `numel(someData)` where `someData` is type `uint8`

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`ContentType` — Media type of payload
`matlab.net.http.MediaType` | empty

Media type of payload, specified as a `matlab.net.http.MediaType` object. The property normally is the `Value` property of the `matlab.net.http.field.ContentTypeField` in the `Header` property. If the `ContentType` property is empty, then the `ContentTypeField` is empty or nonexistent.

MATLAB sets this property before calling `initialize` for the convenience of subclasses that might want to examine the `MediaType`. Subclasses can set this property if they determine from the data that it is of a different `MediaType`.

At the end of the transfer, MATLAB copies this value into the `Response.Body.ContentType` property.

Example: `'application/octet-stream'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Header` — Header of payload currently being processed
`matlab.net.http.HeaderField`

Header of the payload currently being processed, specified as a `matlab.net.http.HeaderField` object.

Consumers use this header to determine how to process the payload that is being sent to them. For a top-level consumer, this value is the same as `Response.Header`. For a delegate, the value might be different. For example, in a multipart message processed by a `MultipartConsumer`, it is the header of the part that this delegate is processing. The delegate can still examine `Response.Header` for headers of the original message.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Request` — Completed `RequestMessage` that was sent
`matlab.net.http.RequestMessage`

The completed `RequestMessage` that was sent, specified as a `matlab.net.http.RequestMessage` object. This is the final `RequestMessage` after all redirections, which is the `completedrequest` return value from the `send` method.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Response` — `ResponseMessage` being processed
`matlab.net.http.ResponseMessage`

The `ResponseMessage` being processed, specified as a `matlab.net.http.ResponseMessage` object.

MATLAB sets the `Response` property before calling `initialize`. The value is the `ResponseMessage` after headers have been received but before receiving any payload. At the start of the response message processing (or the start of a part for multipart messages), the `ResponseMesssage.Body` property is a `MessageBody` object with empty `Data` and `Payload` properties. To store received data, consumers can modify the `Response` and `MessageBody.Data` properties during data transfer. Usually, consumers that process and then store data set `Response.Body.Data` to their processed payload, though this is not required. At the completion of the transfer, MATLAB returns this `Response` to the caller of `send`. Consumers should not modify other `Response` properties, such as `Header` or `StatusLine`, as those changes are returned to the caller of `send`.

The `Response.Body.Payload` property is empty during the transfer and consumers should not attempt to modify it. If the `HTTPOptions.SavePayload` property is set, then MATLAB sets `Payload` to the received payload at the end of the transfer of the message or the part (after the call to `putData(uint8.empty)`) or when an exception occurs.

If an exception occurs in the consumer during message processing, then MATLAB throws an `HTTPException` object. The `History` property contains this `Response` value.

If the consumer is a delegate that is processing part of a multipart message, then `Response.Header` contains the header of the whole message, and the `Payload` and `Data` properties of `Response.Body` are cleared before invoking the `ContentConsumer` for each part. At the conclusion of each part, a new `ResponseMessage` is added to the end of the array of `ResponseMessage` objects in the original response's `Body.Data` containing the `Header` from this object and the `Body` from this property. The next delegate sees a fresh `Response` with an empty `MessageBody`, not the previous delegate's `MessageBody`.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true

URI — Destination of request being processed
`matlab.net.URI`

Destination of the request being processed, specified as a `matlab.net.URI` object. This value is the original destination URI as determined by `send`. It is not the URI of a proxy or the final URI after redirections.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Protected Properties

`AppendFcn` — Function called by `putData` to append additional data

function handle

Function, specified as a function handle, called by the `putData` method to append additional data. The `putData` method in this class calls the `AppendFcn` function to append data it receives in its `data` argument to existing data in the response message. The function must have the signature:

```
AppendFcn(consumer,newdata)
```

where `newdata` is the data to be appended to the array at `consumer.Response.Body.Data`. It is the responsibility of this method to update `consumer.CurrentLength` to reflect the new length of `Data`. If `newdata` is empty, which indicates the end of the stream, then the function should update `Response.Body.Data` to its final value.

The default behavior, if this property is empty, uses an internal function that treats `Data` as an array of arbitrary values supporting the `horzcat` function. It efficiently adds `newdata` by preallocating space, maintaining `CurrentLength` to be the actual length of data stored. At the end of the message, it truncates `Response.Body.Data` to `CurrentLength`.

Subclasses can change this property if `horzcat` is not appropriate for the append process. For example, when a `StringConsumer` builds a scalar string, it would add to the string using the `plus` function instead of `horzcat`.

Subclasses that do not invoke `ContentConsumer.putData` to append data, or which are satisfied with `horzcat` behavior when appending data, can ignore this property.

Example: `@customAppend` where `@customAppend` is defined by the consumer

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`CurrentDelegate` — `ContentConsumer` to which this consumer is delegating
`matlab.net.http.io.ContentConsumer` | `[]`

The `ContentConsumer` to which this consumer is delegating, specified as a `matlab.net.http.io.ContentConsumer` object. The `delegateTo` method of the calling consumer (the delegator) sets the `CurrentDelegate` property. If there is no current delegation, then the value is `[]`.

MATLAB sets `CurrentDelegate` to `[]` before calling `initialize`.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`CurrentLength` — Length of data currently in `Response.Body.Data`
`uint64.empty` (default) | `uint64`

Length of data currently in the `Response.Body.Data` property, specified as `uint64`.

This property is used when `Response.Body.Data` has been preallocated to a size larger than the actual amount of data currently stored, to indicate the length of that stored data. If this property is empty, then it means that all of `Response.Body.Data` contains the stored data or that a `ContentConsumer` subclass is disposing of the data in some way other than storing it in `Response.Body.Data`.

This property is used and set by the `putData` method in this base class when the `AppendFcn` property is empty. It is for the benefit of subclasses that call `putData` and want to examine already-stored data, and/or any implementations of `AppendFcn` that maintain results in `Response.Body.Data`.

Subclasses that use `putData` also can modify this property to reset the position in the buffer where the data is stored. For example, when the default `AppendFcn` function is used, a subclass that processes all of `Response.Body.Data` on each call to `putData` might no longer have a use for the original data, so it can reset the `CurrentLength`

property to 1 so that the next `putData` call overwrites the buffer with new data. There is no need to clear elements in the buffer past the end of the new data.

Subclasses that do not call `putData` can use this property to track their own data, or can leave it unset (empty). MATLAB does not place any constraints on the value that can be set here and does not use it for any purpose other than to determine where the default `AppendFcn` should store the next buffer of data, and where to truncate the data at the end of the message. Set this property to empty before the final call to `putData(uint8.empty)` to prevent truncation of the data.

MATLAB sets this property to empty before each call to `initialize`.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`MyDelegator` — `ContentConsumer` that delegated to this consumer
`matlab.net.http.io.ContentConsumer` | empty

The `ContentConsumer` that delegated to this consumer, specified as a `matlab.net.http.io.ContentConsumer` object. If this consumer is a delegate that was invoked by another consumer, such as a `GenericConsumer` or `MultipartConsumer`, then this is the calling consumer. It is empty in a top-level consumer specified in the call to `send`.

Delegates can use this property to access properties of their delegators, for example, to determine which consumer delegated to them.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

Methods

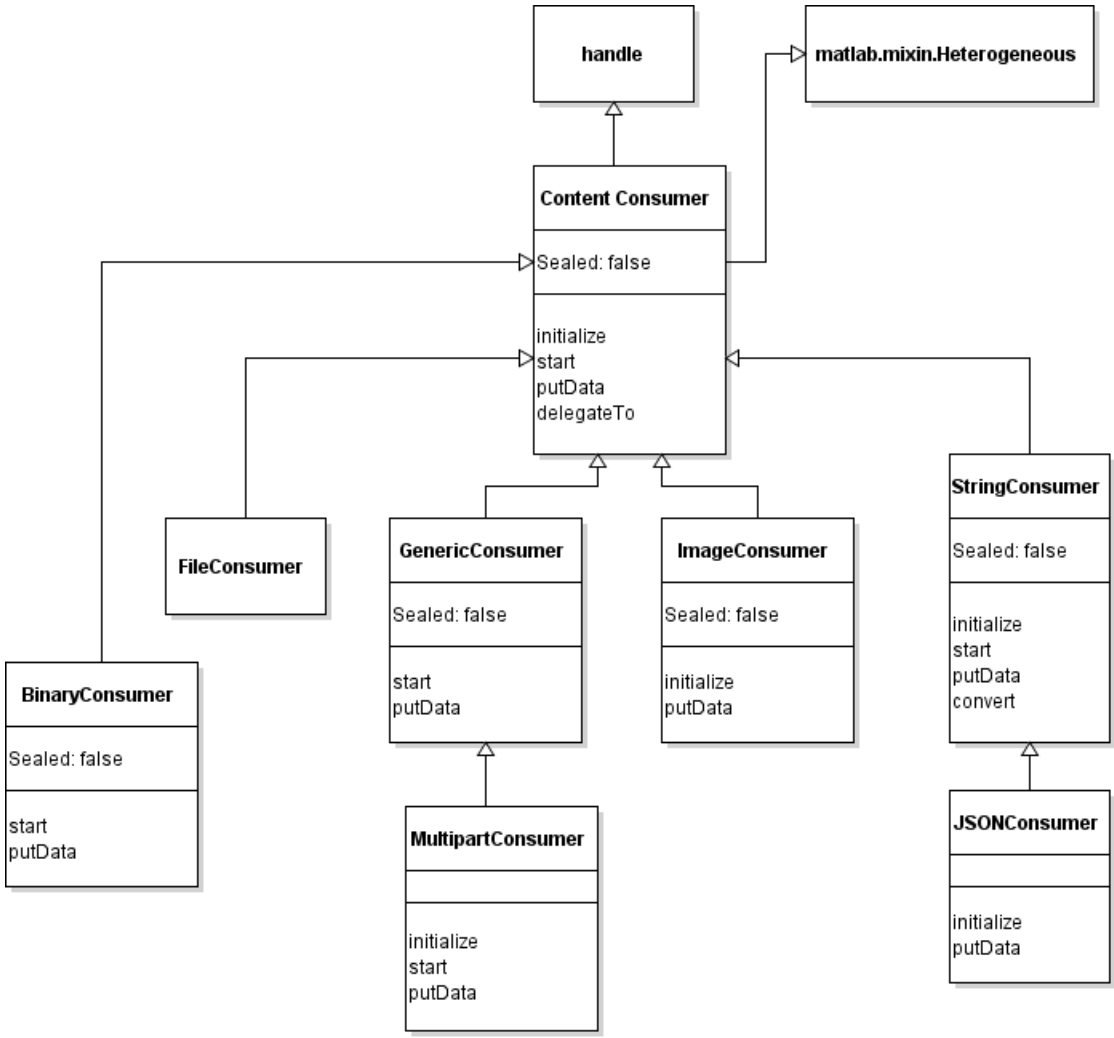
Public Methods

initialize	Prepare ImageConsumer for new image in HTTP message
start	Start HTTP data transfer to ContentConsumer
putData	Save data for ImageConsumer

delegateTo Delegate to another consumer

More About

Class Hierarchy



See Also

`ContentConsumer` | `FileConsumer` | `ImageProvider` | `MessageBody` | `RequestMessage` | `ResponseMessage` | `imfinfo` | `imformats` | `imread` | `matlab.net.URI`

Topics

"HTTP Data Type Conversion"

Introduced in R2018a

initialize

Class: `matlab.net.http.io.ImageConsumer`
Package: `matlab.net.http.io`

Prepare ImageConsumer for new image in HTTP message

Syntax

```
OK = initialize(consumer)
```

Description

`OK = initialize(consumer)` initializes this `ImageConsumer` for receipt of a new image. This method is an overridden method of `initialize` in `ContentConsumer`.

Input Arguments

`consumer` — Content consumer
`matlab.net.http.io.ImageConsumer`

Content consumer, specified as a `matlab.net.http.io.ImageConsumer` object.

Output Arguments

`OK` — Indicate whether consumer accepts or rejects message payload
`true` (default) | `false`

Indicate whether consumer accepts or rejects message payload, returned as `true` or `false`. `initialize` returns `false` if one of the following occurs:

- A Content-Type header field is present and its Type is not `"image"`.
- The Format property is empty and the subtype is not one of those that the `imread` function accepts as a file extension.

Attributes

Access	protected

See Also

ContentConsumer.ContentType | ImageConsumer.Format | `imread` | `initialize`

Introduced in R2018a

putData

Class: `matlab.net.http.io.ImageConsumer`
Package: `matlab.net.http.io`

Save data for ImageConsumer

Syntax

```
[len,stop] = putData(consumer,data)
```

Description

`[len,stop] = putData(consumer,data)` processes buffers of `data`, based on the ContentType property, and returns the result as MATLAB image data in the `Response.Body.Data` property. This method is an overridden method of `putData`. Use this method in subclasses of `ImageConsumer`.

After all the data in the message, the result is RGB data or a cell array containing image data, colormap, and possible transparency, as documented for the `imread` function. For more information on image conversion, see the input conversion section of the matlab.net.http.MessageBody.Data property.

This consumer does not guarantee that `Response.Body.Data` has a useful result until the end of the data is reached (after `putData` is passed empty `data`). Subclasses that override this method, that want to examine the data stream while it is being received, should look at `data`, not `Response.Body.Data`.

Subclasses that only want to see the result after conversion to MATLAB image data can examine `Response.Body.Data` after calling this method with empty `data`.

After the end of data, this method sets the ImageConsumer.Info property.

Input Arguments

`consumer` — Content consumer
`matlab.net.http.io.ImageConsumer`

Content consumer, specified as a `matlab.net.http.io.ImageConsumer` object.

`data` — Buffer of raw data in `ResponseMessage`
nonempty `uint8` vector | `uint8.empty` | `[]` (empty double)

Buffer of raw data in a `matlab.net.http.ResponseMessage` object, specified as a nonempty `uint8` vector, `uint8.empty`, or `[]`. For more information about these values, see the "data" on page 1-0 input argument for the `ContentConsumer.putData` method.

Output Arguments

`len` — Length of data processed
double | `[]` (empty double)

Length of data processed, returned as double or empty double. For more information, see the "size" on page 1-0 argument in `ContentConsumer.putData`.

`stop` — Indicate whether to receive further data
`false` (default) | `true`

Indicate whether to receive further data from this message, returned as `true` or `false`. For more information, see the "stop" on page 1-0 argument in `ContentConsumer.putData`.

Attributes

Access	public

See Also

ContentConsumer.ContentType | ImageConsumer.Info | `imread` | `matlab.net.http.MessageBody` | `putData`

Introduced in R2018a

matlab.net.http.io.JSONConsumer class

Package: `matlab.net.http.io`
Superclasses: `matlab.net.http.io.StringConsumer`

Content consumer that converts JSON input into MATLAB data

Description

This `ContentConsumer` receives messages whose content is JSON. It converts the JSON data to MATLAB and stores the result in the body of the response message.

This consumer should only be applied to incoming content that is JSON (for example, response messages or parts of multipart messages with a Content-Type of `"application/json"`), though it does not check the incoming Content-Type.

By default, MATLAB automatically converts a message with a Content-Type `"application/json"`, so you do not need to specify this consumer for that type, or any other type that obviously indicates JSON. For more information, see input conversions for `MessageBody.Data`. Specify this provider explicitly if you know that the incoming data is JSON even if the Content-Type might not indicate this. For example, sometimes a file containing JSON data has a name with a `.txt` extension. When downloading such a file the server might specify a Content-Type of `"text/plain"` based on that extension, even though it contains JSON data.

If an error occurs converting the data, `Response.Body.Data` in the `HTTPException` thrown on the error contains any intermediate result of the decoding process.

The `matlab.net.http.io.JSONConsumer` class is a `handle` class.

Creation

Description

`consumer = JSONConsumer` creates a consumer that converts a JSON string received in a `ResponseMessage` to MATLAB data using `jsondecode`.

Properties

Public Properties

`Charset` — Character set

empty (default) | string

The character set used to convert the data, specified as a string. This value is initially empty. If you leave it empty, then this value is set when a message is received, based on the specified or default charset in the Content-Type field of the message. If you want to force conversion using a different charset, then you can do so by specifying a charset in the `StringConsumer` constructor, or directly setting this property. Subclasses can set this property at any time, including in the middle of a message. If you change this value after calling `putData`, then the new value is used in subsequent calls to `putData`. The existing contents of `Response.Body.Data` is not changed.

When receipt of a response begins, if this property was initially left empty, then MATLAB sets this property to the chosen charset based on the Content-Type. If you reuse this consumer for a different message, then a new charset can be chosen. If you set this property to a nonempty value, MATLAB never changes it.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`
`Dependent`	`true`

`AllocationLength` — Suggested buffer size

`uint64`

Suggested buffer size, specified as `uint64`. MATLAB sets `AllocationLength` to the anticipated size of buffers of data passed to `putData`. The actual size might be smaller or larger. To improve performance, the consumer can use this value to preallocate space to handle the data.

MATLAB sets this property before calling the `start` method for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`ContentLength` — Expected length of payload

`uint64` | empty

Expected length of the payload, specified as `uint64`. The property normally is the `Value` property of the `matlab.net.http.field.ContentLengthField` in the `Header` property.

If `ContentLength` is empty, then the length is not known. The payload ends when `putData(uint8.empty)` is called.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses that might benefit from knowing the length of the data.

If this `ContentConsumer` is a delegate of a top-level consumer, then the value of `ContentLength` might be different from the `ContentLength` value of the top-level consumer.

Example: `numel(someData)` where `someData` is type `uint8`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`ContentType` — Media type of payload

`matlab.net.http.MediaType` | empty

Media type of payload, specified as a `matlab.net.http.MediaType` object. The property normally is the `Value` property of the `matlab.net.http.field.ContentTypeField` in the `Header` property. If the `ContentType` property is empty, then the `ContentTypeField` is empty or nonexistent.

MATLAB sets this property before calling `initialize` for the convenience of subclasses that might want to examine the `MediaType`. Subclasses can set this property if they determine from the data that it is of a different `MediaType`.

At the end of the transfer, MATLAB copies this value into the `Response.Body.ContentType` property.

Example: `'application/octet-stream'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Header` — Header of payload currently being processed
`matlab.net.http.HeaderField`

Header of the payload currently being processed, specified as a `matlab.net.http.HeaderField` object.

Consumers use this header to determine how to process the payload that is being sent to them. For a top-level consumer, this value is the same as `Response.Header`. For a delegate, the value might be different. For example, in a multipart message processed by a `MultipartConsumer`, it is the header of the part that this delegate is processing. The delegate can still examine `Response.Header` for headers of the original message.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Request` — Completed `RequestMessage` that was sent
`matlab.net.http.RequestMessage`

The completed `RequestMessage` that was sent, specified as a `matlab.net.http.RequestMessage` object. This is the final `RequestMessage` after all redirections, which is the `completedrequest` return value from the `send` method.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Response` — `ResponseMessage` being processed
`matlab.net.http.ResponseMessage`

The `ResponseMessage` being processed, specified as a `matlab.net.http.ResponseMessage` object.

MATLAB sets the `Response` property before calling `initialize`. The value is the `ResponseMessage` after headers have been received but before receiving any payload. At the start of the response message processing (or the start of a part for multipart messages), the `ResponseMesssage.Body` property is a `MessageBody` object with empty `Data` and `Payload` properties. To store received data, consumers can modify the `Response` and `MessageBody.Data` properties during data transfer. Usually, consumers that process and then store data set `Response.Body.Data` to their processed payload, though this is not required. At the completion of the transfer, MATLAB returns this `Response` to the caller of `send`. Consumers should not modify other `Response` properties, such as `Header` or `StatusLine`, as those changes are returned to the caller of `send`.

The `Response.Body.Payload` property is empty during the transfer and consumers should not attempt to modify it. If the `HTTPOptions.SavePayload` property is set, then MATLAB sets `Payload` to the received payload at the end of the transfer of the message or the part (after the call to `putData(uint8.empty)`) or when an exception occurs.

If an exception occurs in the consumer during message processing, then MATLAB throws an `HTTPException` object. The `History` property contains this `Response` value.

If the consumer is a delegate that is processing part of a multipart message, then `Response.Header` contains the header of the whole message, and the `Payload` and `Data` properties of `Response.Body` are cleared before invoking the `ContentConsumer` for each part. At the conclusion of each part, a new `ResponseMessage` is added to the end of the array of `ResponseMessage` objects in the original response's `Body.Data` containing the `Header` from this object and the `Body` from this property. The next delegate sees a fresh `Response` with an empty `MessageBody`, not the previous delegate's `MessageBody`.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true

URI — Destination of request being processed
`matlab.net.URI`

Destination of the request being processed, specified as a `matlab.net.URI` object. This value is the original destination URI as determined by `send`. It is not the URI of a proxy or the final URI after redirections.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

Protected Properties

`AppendFcn` — Function called by `putData` to append additional data

function handle

Function, specified as a function handle, called by the `putData` method to append additional data. The `putData` method in this class calls the `AppendFcn` function to append data it receives in its `data` argument to existing data in the response message. The function must have the signature:

```
AppendFcn(consumer,newdata)
```

where `newdata` is the data to be appended to the array at `consumer.Response.Body.Data`. It is the responsibility of this method to update `consumer.CurrentLength` to reflect the new length of `Data`. If `newdata` is empty, which indicates the end of the stream, then the function should update `Response.Body.Data` to its final value.

The default behavior, if this property is empty, uses an internal function that treats `Data` as an array of arbitrary values supporting the `horzcat` function. It efficiently adds `newdata` by preallocating space, maintaining `CurrentLength` to be the actual length of data stored. At the end of the message, it truncates `Response.Body.Data` to `CurrentLength`.

Subclasses can change this property if `horzcat` is not appropriate for the append process. For example, when a `StringConsumer` builds a scalar string, it would add to the string using the `plus` function instead of `horzcat`.

Subclasses that do not invoke `ContentConsumer.putData` to append data, or which are satisfied with `horzcat` behavior when appending data, can ignore this property.

Example: `@customAppend` where `@customAppend` is defined by the consumer

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`CurrentDelegate` — `ContentConsumer` to which this consumer is delegating
`matlab.net.http.io.ContentConsumer` | `[]`

The `ContentConsumer` to which this consumer is delegating, specified as a `matlab.net.http.io.ContentConsumer` object. The `delegateTo` method of the calling consumer (the delegator) sets the `CurrentDelegate` property. If there is no current delegation, then the value is `[]`.

MATLAB sets `CurrentDelegate` to `[]` before calling `initialize`.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`CurrentLength` — Length of data currently in `Response.Body.Data`
`uint64.empty` (default) | `uint64`

Length of data currently in the `Response.Body.Data` property, specified as `uint64`.

This property is used when `Response.Body.Data` has been preallocated to a size larger than the actual amount of data currently stored, to indicate the length of that stored data. If this property is empty, then it means that all of `Response.Body.Data` contains the stored data or that a `ContentConsumer` subclass is disposing of the data in some way other than storing it in `Response.Body.Data`.

This property is used and set by the `putData` method in this base class when the `AppendFcn` property is empty. It is for the benefit of subclasses that call `putData` and want to examine already-stored data, and/or any implementations of `AppendFcn` that maintain results in `Response.Body.Data`.

Subclasses that use `putData` also can modify this property to reset the position in the buffer where the data is stored. For example, when the default `AppendFcn` function is used, a subclass that processes all of `Response.Body.Data` on each call to `putData` might no longer have a use for the original data, so it can reset the `CurrentLength`

property to 1 so that the next `putData` call overwrites the buffer with new data. There is no need to clear elements in the buffer past the end of the new data.

Subclasses that do not call `putData` can use this property to track their own data, or can leave it unset (empty). MATLAB does not place any constraints on the value that can be set here and does not use it for any purpose other than to determine where the default `AppendFcn` should store the next buffer of data, and where to truncate the data at the end of the message. Set this property to empty before the final call to `putData(uint8.empty)` to prevent truncation of the data.

MATLAB sets this property to empty before each call to `initialize`.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`MyDelegator` — `ContentConsumer` that delegated to this consumer
`matlab.net.http.io.ContentConsumer` | empty

The `ContentConsumer` that delegated to this consumer, specified as a `matlab.net.http.io.ContentConsumer` object. If this consumer is a delegate that was invoked by another consumer, such as a `GenericConsumer` or `MultipartConsumer`, then this is the calling consumer. It is empty in a top-level consumer specified in the call to `send`.

Delegates can use this property to access properties of their delegators, for example, to determine which consumer delegated to them.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

Methods

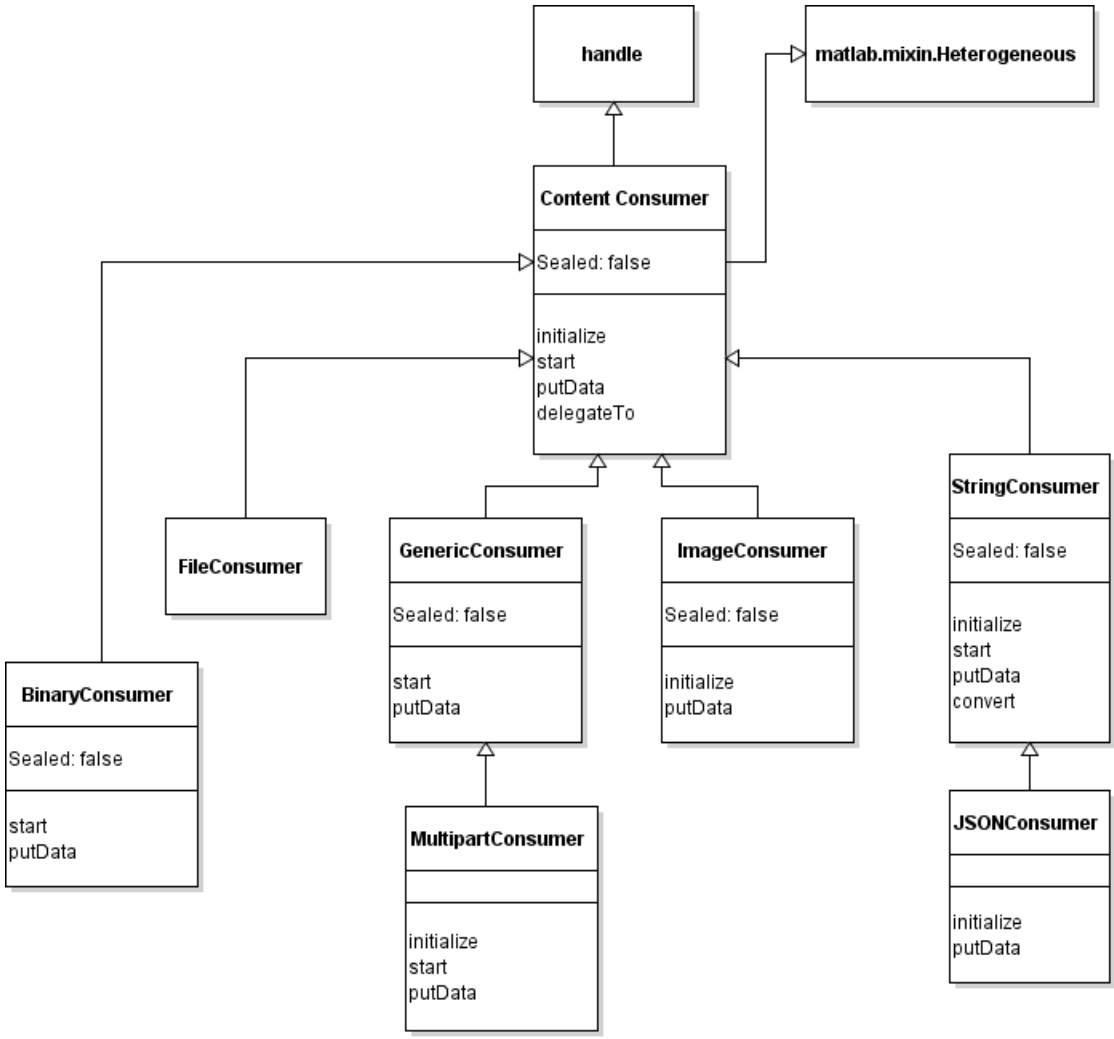
Public Methods

convert	Convert data to string in StringConsumer
initialize	Prepare JSONConsumer for new HTTP message
putData	Save next buffer of JSON data for JSONConsumer

start Start data transfer to StringConsumer

More About

Class Hierarchy



See Also

`ContentConsumer` | `JSONProvider` | `MessageBody` | `StringConsumer` | `jsondecode`

Introduced in R2018a

initialize

Class: `matlab.net.http.io.JSONConsumer`
Package: `matlab.net.http.io`

Prepare JSONConsumer for new HTTP message

Syntax

```
OK = initialize(consumer)
```

Description

`OK = initialize(consumer)` initializes a `JSONConsumer` for a new HTTP message.

MATLAB calls `initialize` to prepare this consumer for receipt of a message. This method tries to determine the charset of the data from the Content-Type in the `Header` property. If it cannot do so, it assumes UTF-8. This method is an overridden method of `initialize`.

`initialize` returns `true` if the consumer can process the data. Otherwise, it returns `false`.

Input Arguments

`consumer` — Content consumer
`matlab.net.http.io.JSONConsumer`

Content consumer, specified as a `matlab.net.http.io.JSONConsumer` object.

Attributes

Access	protected

See Also

`initialize`

Introduced in R2018a

putData

Class: `matlab.net.http.io.JSONConsumer`
Package: `matlab.net.http.io`

Save next buffer of JSON data for JSONConsumer

Syntax

```
[len,stop] = putData(consumer,data)
```

Description

`[len,stop] = putData(consumer,data)` passes data to the superclass to convert the `uint8` buffer to a string, and then uses `jsondecode` to decode it and insert it into `Response.Body.Data`.

This method might not store the decoded JSON data until the entire message has been read.

This method is an overridden method of `putData`. You can override this method to examine or alter the `uint8` data before conversion, or the JSON data after decoding.

Input Arguments

`consumer` — Content consumer
`matlab.net.http.io.JSONConsumer`

Content consumer, specified as a `matlab.net.http.io.JSONConsumer` object.

`data` — Buffer of raw data in `ResponseMessage`
nonempty `uint8` vector | `uint8.empty` | `[]` (empty double)

Buffer of raw data in a `matlab.net.http.ResponseMessage` object, specified as a nonempty `uint8` vector, `uint8.empty`, or `[]`. For more information about these values,

see the “data” on page 1-0 input argument for the `ContentConsumer.putData` method.

Output Arguments

`len` — Length of data processed
double | `[]` (empty double)

Length of data processed, returned as double or empty double. For more information, see the “size” on page 1-0 argument in `ContentConsumer.putData`.

`stop` — Indicate whether to receive further data
`false` (default) | `true`

Indicate whether to receive further data from this message, returned as `true` or `false`. For more information, see the “stop” on page 1-0 argument in `ContentConsumer.putData`.

Attributes

Access	public

To learn about attributes of methods, see Method Attributes.

See Also

`jsondecode` | `putData`

Introduced in R2018a

matlab.net.http.io.MultipartConsumer class

Package: `matlab.net.http.io`
Superclasses: `matlab.net.http.io.GenericConsumer`

Helper for multipart content types in HTTP messages

Description

This consumer processes multipart HTTP response messages. A multipart message is one whose Content-Type header field specifies `"multipart"`, and whose body contains one or more parts. Each part contains its own set of header fields describing the part, the most important of which is a Content-Type field.

The `matlab.net.http.io.MultipartConsumer` class is a `handle` class.

Subclass Authors

If you are writing your own `ContentConsumer`, it will generally work whether it is a top level consumer (specified as the 3rd argument to the `RequestMessage.send` method) or a part of a multipart message (when specified as a "delegate" in the `MultipartConsumer` constructor call). `MultipartConsumer` makes it appear to each delegate as if it was handling the entire response message, while actually assembling the results into an array of `ResponseMessages` stored in the returned `response.Body.Data` property.

The following describes the `MultipartConsumer` behavior:

Each time this `MultipartConsumer` receives a complete part of a message from the server, it parses any headers in the part and then invokes the appropriate delegate consumer appropriate for the Content-Type field in the part. If there is no Content-Type field in the part, it assumes the type is `text/plain`. If there is no delegate able to handle the type, it uses default processing for the part based on the Content-Type, as described for `GenericConsumer`.

`MultipartConsumer` does not invoke a delegate until it receives a complete part. `MultipartConsumer` buffers the data for a part, and at the end of receipt of the part, it copies all the visible properties of `ContentConsumer` from this consumer to the

delegate, clears the delegate's `Response.Body`, sets the delegate's `Header` to the header of the part, and then calls the delegate's `initialize` and `start` methods, followed by one or more calls to the delegate's `putData` method containing the payload of the part, followed by a call to `putData(uint8.empty)` to indicate end-of-data. If the delegate's `initialize` method returns `false` to indicate it does not want to handle the part, the payload of the part is processed using default behavior for the Content-Type of the part, as described for `GenericConsumer`.

If the delegate's `start` method returns `[]` to indicate that there is no maximum desired buffer size, `MultipartConsumer` makes just one call to `putData` that provides the entire payload of the part, followed by the end-of-data call. Otherwise it calls `putData` enough times to supply the entire payload in units of the buffer size.

If the delegate's `putData` method sets the `STOP` return value to `true` to indicate that it does not want any more data, then `MultipartConsumer` closes the connection to end the transfer, as if the message had ended. In this way the delegate controls whether the remainder of the original message should be processed. If `putData` returns a `SIZE` of `[]`, then the message also ends, but with an exception thrown to the caller of `RequestMessage.send`.

If the consumer for a part was specified as a function handle rather than a `ContentConsumer` instance, then the function is called only the first time the consumer is needed, and subsequently the same consumer instance is used for any appropriate parts of the same response message. For parts processed by a function handle, the corresponding `ResponseMessage` in `Response.Body.Data` contains only a header for the part, because the function does not have access to the `ResponseMessage` body.

A delegated consumer can access this consumer and its properties through its `MyDelegator` property, though that is rarely necessary.

Creation

Description

`consumer = MultipartConsumer(types,consumer)` constructs a `MultipartConsumer` to handle the specified `types` using the corresponding `consumer`. You can specify several argument pairs in any order as `types1,consumer1,...,typesN,consumerN`. MATLAB searches the `types` in the

order they appear and uses the first match. If there are no matches among the specified types, MATLAB uses the default set of consumers, depending on the type.

`consumer = MultipartConsumer(puthandle)` constructs a `ContentConsumer` that calls the function specified by `puthandle` for each call to this consumer's `putData` method.

Input Arguments

`types` — Content types

string array | character vector | cell array of character vectors

Content types, specified as a string array, character vector, or cell array of character vectors. `types` specifies content types using the syntax `"type/subtype"`. For more information, see `matlab.net.http.io.GenericConsumer`.

`consumer` — Content consumer

`matlab.net.http.io.ContentConsumer` | function handle

Content consumer, specified as a `matlab.net.http.io.ContentConsumer` object that can handle the specified types, or a handle to a function returning a `ContentConsumer` that can handle those types.

`puthandle` — putData function

function handle

Handle to a `putData` function with the following `ContentConsumer.putData` syntax:

```
[length,stop] = putData(data)
```

where *`data`* is a `uint8` array and *`length`* is the length of that array.

Use this syntax to process all input from the server using a single function, when you know the type of data that the server returns. The function does not have access to the `ResponseMessage` or any information about this consumer.

Properties

Public Properties

`Preamble` — Content before first delimiter

uint8

The part of the multipart message before the first boundary delimiter, if any, specified as `uint8`. The consumer sets `Preamble` before calling the `start` method in a delegate. Once set, the `Preamble` value never changes.

Attributes:

```
GetAccess                               public
SetAccess                               private
```

`Epilogue` — Content after last delimiter

uint8

The part of the multipart message following the last boundary delimiter, if any, specified as `uint8`. The consumer sets `Epilogue` when the message ends, after all calls to delegates. It is not set if a delegate terminates the transfer before the end of the message. You can examine this property after the transfer is complete, for example, when `RequestMessage.send` returns.

Attributes:

```
GetAccess                               public
SetAccess                               private
```

`AllocationLength` — Suggested buffer size

uint64

Suggested buffer size, specified as `uint64`. MATLAB sets `AllocationLength` to the anticipated size of buffers of data passed to `putData`. The actual size might be smaller or larger. To improve performance, the consumer can use this value to preallocate space to handle the data.

MATLAB sets this property before calling the `start` method for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`ContentLength` — Expected length of payload

`uint64` | empty

Expected length of the payload, specified as `uint64`. The property normally is the `Value` property of the `matlab.net.http.field.ContentLengthField` in the `Header` property.

If `ContentLength` is empty, then the length is not known. The payload ends when `putData(uint8.empty)` is called.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses that might benefit from knowing the length of the data.

If this `ContentConsumer` is a delegate of a top-level consumer, then the value of `ContentLength` might be different from the `ContentLength` value of the top-level consumer.

Example: `numel(someData)` where `someData` is type `uint8`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`ContentType` — Media type of payload

`matlab.net.http.MediaType` | empty

Media type of payload, specified as a `matlab.net.http.MediaType` object. The property normally is the `Value` property of the `matlab.net.http.field.ContentTypeField` in the `Header` property. If the `ContentType` property is empty, then the `ContentTypeField` is empty or nonexistent.

MATLAB sets this property before calling `initialize` for the convenience of subclasses that might want to examine the `MediaType`. Subclasses can set this property if they determine from the data that it is of a different `MediaType`.

At the end of the transfer, MATLAB copies this value into the `Response.Body.ContentType` property.

Example: `'application/octet-stream'`

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Header` — Header of payload currently being processed
`matlab.net.http.HeaderField`

Header of the payload currently being processed, specified as a `matlab.net.http.HeaderField` object.

Consumers use this header to determine how to process the payload that is being sent to them. For a top-level consumer, this value is the same as `Response.Header`. For a delegate, the value might be different. For example, in a multipart message processed by a `MultipartConsumer`, it is the header of the part that this delegate is processing. The delegate can still examine `Response.Header` for headers of the original message.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Request` — Completed `RequestMessage` that was sent
`matlab.net.http.RequestMessage`

The completed `RequestMessage` that was sent, specified as a `matlab.net.http.RequestMessage` object. This is the final `RequestMessage` after all redirections, which is the `completedrequest` return value from the `send` method.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Response` — `ResponseMessage` being processed
`matlab.net.http.ResponseMessage`

The `ResponseMessage` being processed, specified as a `matlab.net.http.ResponseMessage` object.

MATLAB sets the `Response` property before calling `initialize`. The value is the `ResponseMessage` after headers have been received but before receiving any payload. At the start of the response message processing (or the start of a part for multipart messages), the `ResponseMesssage.Body` property is a `MessageBody` object with empty `Data` and `Payload` properties. To store received data, consumers can modify the `Response` and `MessageBody.Data` properties during data transfer. Usually, consumers that process and then store data set `Response.Body.Data` to their processed payload, though this is not required. At the completion of the transfer, MATLAB returns this `Response` to the caller of `send`. Consumers should not modify other `Response` properties, such as `Header` or `StatusLine`, as those changes are returned to the caller of `send`.

The `Response.Body.Payload` property is empty during the transfer and consumers should not attempt to modify it. If the `HTTPOptions.SavePayload` property is set, then MATLAB sets `Payload` to the received payload at the end of the transfer of the message or the part (after the call to `putData(uint8.empty)`) or when an exception occurs.

If an exception occurs in the consumer during message processing, then MATLAB throws an `HTTPException` object. The `History` property contains this `Response` value.

If the consumer is a delegate that is processing part of a multipart message, then `Response.Header` contains the header of the whole message, and the `Payload` and `Data` properties of `Response.Body` are cleared before invoking the `ContentConsumer` for each part. At the conclusion of each part, a new `ResponseMessage` is added to the end of the array of `ResponseMessage` objects in the original response's `Body.Data` containing the `Header` from this object and the `Body` from this property. The next delegate sees a fresh `Response` with an empty `MessageBody`, not the previous delegate's `MessageBody`.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true

URI — Destination of request being processed
`matlab.net.URI`

Destination of the request being processed, specified as a `matlab.net.URI` object. This value is the original destination URI as determined by `send`. It is not the URI of a proxy or the final URI after redirections.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Protected Properties

`PutMethod` — Handle to delegate's putData method

function handle | `[]`

`putData` method of the delegate, specified as a function handle, or specified as `[]` if `CurrentDelegate` is set.

Attributes:

```
GetAccess                             protected
SetAccess                             protected
```

`AppendFcn` — Function called by `putData` to append additional data

function handle

Function, specified as a function handle, called by the `putData` method to append additional data. The `putData` method in this class calls the `AppendFcn` function to append data it receives in its `data` argument to existing data in the response message. The function must have the signature:

```
AppendFcn(consumer,newdata)
```

where `newdata` is the data to be appended to the array at `consumer.Response.Body.Data`. It is the responsibility of this method to update `consumer.CurrentLength` to reflect the new length of `Data`. If `newdata` is empty, which indicates the end of the stream, then the function should update `Response.Body.Data` to its final value.

The default behavior, if this property is empty, uses an internal function that treats `Data` as an array of arbitrary values supporting the `horzcat` function. It efficiently adds

newdata by preallocating space, maintaining CurrentLength to be the actual length of data stored. At the end of the message, it truncates Response.Body.Data to CurrentLength.

Subclasses can change this property if horzcat is not appropriate for the append process. For example, when a StringConsumer builds a scalar string, it would add to the string using the plus function instead of horzcat.

Subclasses that do not invoke ContentConsumer.putData to append data, or which are satisfied with horzcat behavior when appending data, can ignore this property.

Example: @customAppend where @customAppend is defined by the consumer

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

CurrentDelegate — ContentConsumer to which this consumer is delegating
matlab.net.http.io.ContentConsumer | []

The ContentConsumer to which this consumer is delegating, specified as a matlab.net.http.io.ContentConsumer object. The delegateTo method of the calling consumer (the delegator) sets the CurrentDelegate property. If there is no current delegation, then the value is [].

MATLAB sets CurrentDelegate to [] before calling initialize.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

CurrentLength — Length of data currently in Response.Body.Data
uint64.empty (default) | uint64

Length of data currently in the Response.Body.Data property, specified as uint64.

This property is used when Response.Body.Data has been preallocated to a size larger than the actual amount of data currently stored, to indicate the length of that stored data. If this property is empty, then it means that all of Response.Body.Data contains the stored data or that a ContentConsumer subclass is disposing of the data in some way other than storing it in Response.Body.Data.

This property is used and set by the `putData` method in this base class when the `AppendFcn` property is empty. It is for the benefit of subclasses that call `putData` and want to examine already-stored data, and/or any implementations of `AppendFcn` that maintain results in `Response.Body.Data`.

Subclasses that use `putData` also can modify this property to reset the position in the buffer where the data is stored. For example, when the default `AppendFcn` function is used, a subclass that processes all of `Response.Body.Data` on each call to `putData` might no longer have a use for the original data, so it can reset the `CurrentLength` property to 1 so that the next `putData` call overwrites the buffer with new data. There is no need to clear elements in the buffer past the end of the new data.

Subclasses that do not call `putData` can use this property to track their own data, or can leave it unset (empty). MATLAB does not place any constraints on the value that can be set here and does not use it for any purpose other than to determine where the default `AppendFcn` should store the next buffer of data, and where to truncate the data at the end of the message. Set this property to empty before the final call to `putData(uint8.empty)` to prevent truncation of the data.

MATLAB sets this property to empty before each call to `initialize`.

Attributes:

GetAccess	protected
SetAccess	protected

`MyDelegator` — `ContentConsumer` that delegated to this consumer
`matlab.net.http.io.ContentConsumer` | empty

The `ContentConsumer` that delegated to this consumer, specified as a `matlab.net.http.io.ContentConsumer` object. If this consumer is a delegate that was invoked by another consumer, such as a `GenericConsumer` or `MultipartConsumer`, then this is the calling consumer. It is empty in a top-level consumer specified in the call to `send`.

Delegates can use this property to access properties of their delegators, for example, to determine which consumer delegated to them.

Attributes:

GetAccess	protected
SetAccess	protected

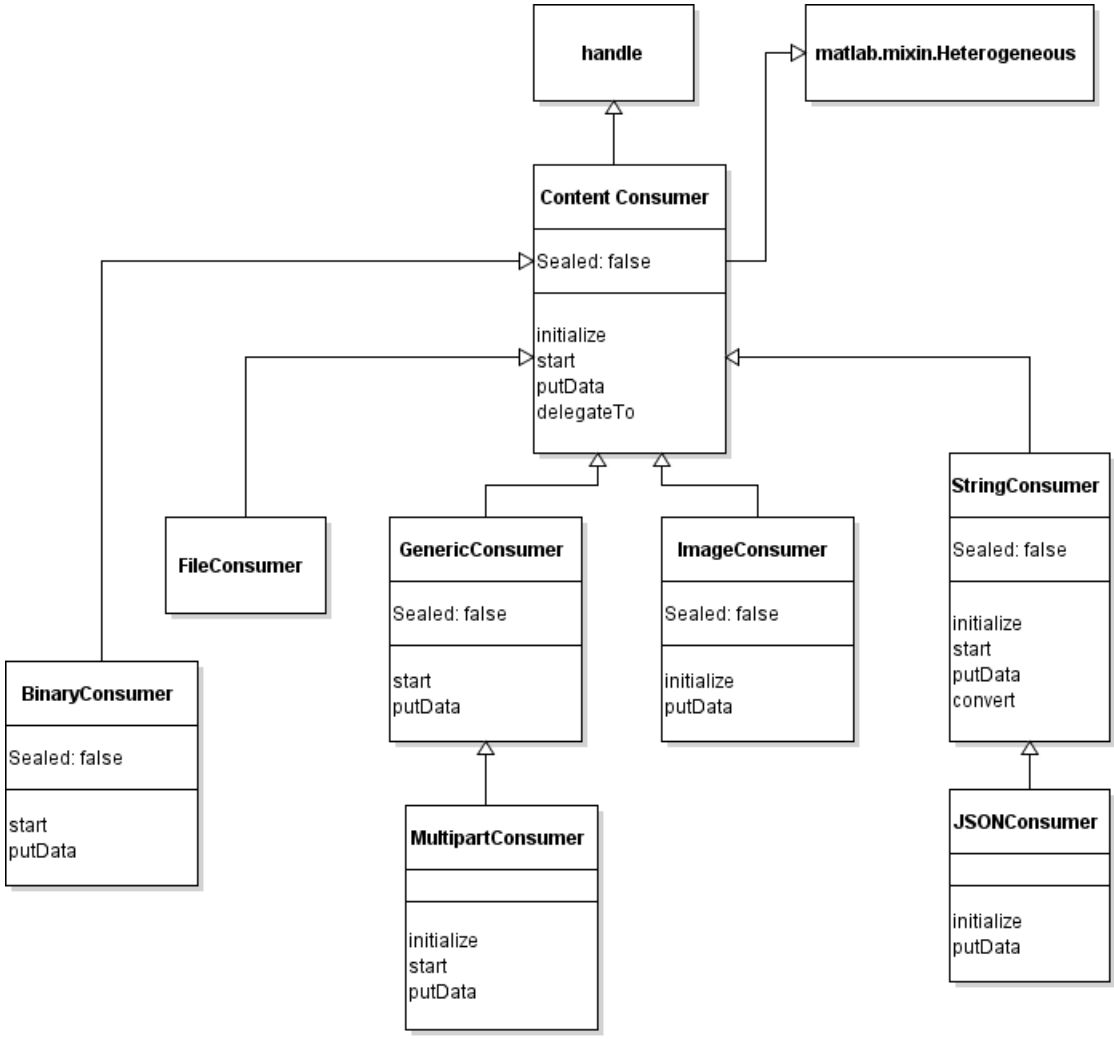
Methods

Public Methods

initialize	Prepare MultipartConsumer for new HTTP message
start	Start data transfer to MultipartConsumer
putData	Process next buffer of data for MultipartConsumer
delegateTo	Delegate to another consumer

More About

Class Hierarchy



See Also

`FileConsumer` | `GenericConsumer` | `ImageConsumer` | `MediaType` | `MessageBody` | `MultipartProvider` | `RequestMessage` | `ResponseMessage`

Introduced in R2018a

initialize

Class: `matlab.net.http.io.MultipartConsumer`
Package: `matlab.net.http.io`

Prepare MultipartConsumer for new HTTP message

Syntax

```
OK = initialize(consumer)
```

Description

`OK = initialize(consumer)` prepares this `MultipartConsumer` for a new message. This method verifies that the `matlab.net.http.field.ContentTypeField` of the message, if present, has a `matlab.net.http.MediaType` object whose `Type` property is `"multipart"`, and that it has a "boundary" parameter indicating the delimiter between parts. `initialize` ignores the subtype.

If the `ContentTypeField` is missing, then this consumer stores the raw data in the `Payload` property of the message.

`initialize` returns `true` if it can process the message. Otherwise, it returns `false`.

This method is an overridden method of `initialize`.

Input Arguments

`consumer` — Content consumer
`matlab.net.http.io.MultipartConsumer`

Content consumer, specified as a `matlab.net.http.io.MultipartConsumer` object.

Attributes

Access	protected

See Also

initialize | matlab.net.http.MediaType | matlab.net.http.field.ContentTypeField

Introduced in R2018a

putData

Class: `matlab.net.http.io.MultipartConsumer`
Package: `matlab.net.http.io`

Process next buffer of data for MultipartConsumer

Syntax

```
[length,stop] = putData(consumer,data)
```

Description

`[length,stop] = putData(consumer,data)` accumulates buffers of `data` until an entire part of a multipart message has been assembled. It then uses the Content-Type field in the part's header to find an appropriate `ContentConsumer` delegate that can handle that type, sets the delegate's `Header` property to the header of the part, and then calls `initialize` and `start` in that delegate. It follows that with one or more `putData` calls, passing in the part's payload, and then calls `putData(uint8.empty)` to indicate the end of the payload.

After the final call to the delegate's `putData`, this method creates a `ResponseMessage` containing the header of the part and a `Body` copied from `Response.Body` in the delegate. (That `Body` might or might not contain data, depending on what the delegate does.) It adds that new `ResponseMessage` to the array of `ResponseMessage` objects in this consumer's `Response.Body.Data` property, which, when the end of the message has been reached, contains one `ResponseMessage` for every part.

This method is an overridden method of `putData`. If you override this method and return `stop=true` before the end of the message (if `data` is not empty) in order to terminate receipt of the message before the normal end of message, you should avoid calling this superclass method on the subsequent `putData(uint8.empty)` call that MATLAB normally makes after you set `stop`. Failure to do so results in an invalid message exception from `MultipartConsumer` due to a premature end of message.

Input Arguments

`consumer` — Content consumer
`matlab.net.http.io.MultipartConsumer`

Content consumer, specified as a `matlab.net.http.io.MultipartConsumer` object.

`data` — Buffer of raw data in `ResponseMessage`
nonempty `uint8` vector | `uint8.empty` | `[]` (empty double)

Buffer of raw data in a `matlab.net.http.ResponseMessage` object, specified as a nonempty `uint8` vector, `uint8.empty`, or `[]`. For more information about these values, see the "data" on page 1-0 input argument for the `ContentConsumer.putData` method.

If `data` is `[]` to indicate that the message has ended and there was a delegate, copies the `Response` property from the delegate to this object's `Response`.

Output Arguments

`length` — Length of data processed
double | `[]` (empty double)

Length of data processed, returned as double or empty double. For more information, see the "size" on page 1-0 argument in `ContentConsumer.putData`.

`stop` — Indicate whether to receive further data
`false` (default) | `true`

Indicate whether to receive further data from this message, returned as `true` or `false`. For more information, see the "stop" on page 1-0 argument in `ContentConsumer.putData`.

Attributes

Access	public

See Also

`matlab.net.http.MessageBody` | `matlab.net.http.ResponseMessage` | `matlab.net.http.field.ContentTypeField` | `putData`

Introduced in R2018a

start

Class: `matlab.net.http.io.MultipartConsumer`
Package: `matlab.net.http.io`

Start data transfer to MultipartConsumer

Syntax

```
bufsize = start(consumer)
```

Description

`bufsize = start(consumer)` prepares `consumer` for receipt of data. The method returns `[]` to indicate that it has no preferred buffer size. This method is an abstract method of `start`.

Input Arguments

`consumer` — Content consumer
`matlab.net.http.io.MultipartConsumer`

Content consumer, specified as a `matlab.net.http.io.MultipartConsumer` object.

Attributes

Access	protected

See Also

`start`

Introduced in R2018a

matlab.net.http.io.StringConsumer class

Package: `matlab.net.http.io`
Superclasses: `matlab.net.http.io.ContentConsumer`

String consumer for HTTP payloads

Description

A `StringConsumer` stores character data in a response message, decoded according to the charset based on the Content-Type, in the response body. You can specify this consumer directly when sending a `RequestMessage` to specify a string conversion for the data with certain parameters.

For an example subclassing this consumer, see the `PricesStreamer` class in “Display Streamed Data in Figure Window”.

The `matlab.net.http.io.StringConsumer` class is a `handle` class.

Creation

Description

`consumer = StringConsumer` constructs a consumer that converts input to a scalar string using the character set specified in the Content-Type of the message.

`consumer = StringConsumer(Name,Value)` constructs a consumer with options specified by one or more `Name,Value` pair arguments. `Name` is the property name `Charset` or `TextType` and `Value` is the corresponding value. You can specify the name-value pair arguments in any order as `Name1,Value1,Name2,Value2`.

If you do not specify a `Charset` property, then this consumer tries to derive the charset from the `ContentType` property, which MATLAB sets based on the Content-Type field in the `Response`. `StringConsumer` knows the charset if `ContentType` has an explicit charset parameter, or if it is one of the types for which MATLAB knows the default charset:

- `"text/*"` - US-ASCII or UTF-8 depending on the subtype.
- `"application/*"` - UTF-8 for subtypes: json, xml, javascript, css, x-www-form-urlencoded; unknown otherwise.

If this consumer cannot determine the charset from the `ContentType` in the message, then this consumer rejects the message and it is not converted. In that case, the `ResponseMessage.Body` contains only a uint8 payload. If you want to convert a message with an unknown charset, then set `Charset` in this consumer before applying it to a message (or, if you are a subclass author, before calling the `initialize` method). A good one to use is UTF-8 because that is a superset of US-ASCII and some other charsets.

Properties

Public Properties

`Charset` — Character set

empty (default) | string

The character set used to convert the data, specified as a string. This value is initially empty. If you leave it empty, then this value is set when a message is received, based on the specified or default charset in the Content-Type field of the message. If you want to force conversion using a different charset, then you can do so by specifying a charset in the `StringConsumer` constructor, or directly setting this property. Subclasses can set this property at any time, including in the middle of a message. If you change this value after calling `putData`, then the new value is used in subsequent calls to `putData`. The existing contents of `Response.Body.Data` is not changed.

When receipt of a response begins, if this property was initially left empty, then MATLAB sets this property to the chosen charset based on the Content-Type. If you reuse this consumer for a different message, then a new charset can be chosen. If you set this property to a nonempty value, MATLAB never changes it.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`
`Dependent`	`true`

`TextType` — Data return type

`"string"` (default) | `"char"`

The type of data to return, specified as `"string"` or `"char"`. The value is from the 'TextType' parameter to the constructor. Default is "string". If you change this value after data is already stored in `Response.Body.Data`, then that data is converted to the new type.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

`AllocationLength` — Suggested buffer size

`uint64`

Suggested buffer size, specified as `uint64`. MATLAB sets `AllocationLength` to the anticipated size of buffers of data passed to `putData`. The actual size might be smaller or larger. To improve performance, the consumer can use this value to preallocate space to handle the data.

MATLAB sets this property before calling the `start` method for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`ContentLength` — Expected length of payload

`uint64` | empty

Expected length of the payload, specified as `uint64`. The property normally is the `Value` property of the `matlab.net.http.field.ContentLengthField` in the `Header` property.

If `ContentLength` is empty, then the length is not known. The payload ends when `putData(uint8.empty)` is called.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses that might benefit from knowing the length of the data.

If this `ContentConsumer` is a delegate of a top-level consumer, then the value of `ContentLength` might be different from the `ContentLength` value of the top-level consumer.

Example: `numel(someData)` where `someData` is type `uint8`

Attributes:

GetAccess	public
SetAccess	public

`ContentType` — Media type of payload
`matlab.net.http.MediaType` | empty

Media type of payload, specified as a `matlab.net.http.MediaType` object. The property normally is the `Value` property of the `matlab.net.http.field.ContentTypeField` in the `Header` property. If the `ContentType` property is empty, then the `ContentTypeField` is empty or nonexistent.

MATLAB sets this property before calling `initialize` for the convenience of subclasses that might want to examine the `MediaType`. Subclasses can set this property if they determine from the data that it is of a different `MediaType`.

At the end of the transfer, MATLAB copies this value into the `Response.Body.ContentType` property.

Example: `'application/octet-stream'`

Attributes:

GetAccess	public
SetAccess	public

`Header` — Header of payload currently being processed
`matlab.net.http.HeaderField`

Header of the payload currently being processed, specified as a `matlab.net.http.HeaderField` object.

Consumers use this header to determine how to process the payload that is being sent to them. For a top-level consumer, this value is the same as `Response.Header`. For a delegate, the value might be different. For example, in a multipart message processed by a `MultipartConsumer`, it is the header of the part that this delegate is processing. The delegate can still examine `Response.Header` for headers of the original message.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Request` — Completed `RequestMessage` that was sent
`matlab.net.http.RequestMessage`

The completed `RequestMessage` that was sent, specified as a `matlab.net.http.RequestMessage` object. This is the final `RequestMessage` after all redirections, which is the `completedrequest` return value from the `send` method.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Response` — `ResponseMessage` being processed
`matlab.net.http.ResponseMessage`

The `ResponseMessage` being processed, specified as a `matlab.net.http.ResponseMessage` object.

MATLAB sets the `Response` property before calling `initialize`. The value is the `ResponseMessage` after headers have been received but before receiving any payload. At the start of the response message processing (or the start of a part for multipart messages), the `ResponseMesssage.Body` property is a `MessageBody` object with empty `Data` and `Payload` properties. To store received data, consumers can modify the `Response` and `MessageBody.Data` properties during data transfer. Usually, consumers that process and then store data set `Response.Body.Data` to their processed payload, though this is not required. At the completion of the transfer, MATLAB returns this `Response` to the caller of `send`. Consumers should not modify other `Response` properties, such as `Header` or `StatusLine`, as those changes are returned to the caller of `send`.

The `Response.Body.Payload` property is empty during the transfer and consumers should not attempt to modify it. If the `HTTPOptions.SavePayload` property is set, then MATLAB sets `Payload` to the received payload at the end of the transfer of the message or the part (after the call to `putData(uint8.empty)`) or when an exception occurs.

If an exception occurs in the consumer during message processing, then MATLAB throws an `HTTPException` object. The `History` property contains this `Response` value.

If the consumer is a delegate that is processing part of a multipart message, then `Response.Header` contains the header of the whole message, and the `Payload` and `Data` properties of `Response.Body` are cleared before invoking the `ContentConsumer` for each part. At the conclusion of each part, a new `ResponseMessage` is added to the end of the array of `ResponseMessage` objects in the original response's `Body.Data` containing the `Header` from this object and the `Body` from this property. The next delegate sees a fresh `Response` with an empty `MessageBody`, not the previous delegate's `MessageBody`.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

`URI` — Destination of request being processed
`matlab.net.URI`

Destination of the request being processed, specified as a `matlab.net.URI` object. This value is the original destination URI as determined by `send`. It is not the URI of a proxy or the final URI after redirections.

MATLAB sets this property before calling `initialize`, for the convenience of subclasses.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Protected Properties

`AppendFcn` — Function called by `putData` to append additional data
function handle

Function, specified as a function handle, called by the `putData` method to append additional data. The `putData` method in this class calls the `AppendFcn` function to append data it receives in its `data` argument to existing data in the response message. The function must have the signature:

```
AppendFcn(consumer,newdata)
```

where `newdata` is the data to be appended to the array at `consumer.Response.Body.Data`. It is the responsibility of this method to update `consumer.CurrentLength` to reflect the new length of `Data`. If `newdata` is empty, which indicates the end of the stream, then the function should update `Response.Body.Data` to its final value.

The default behavior, if this property is empty, uses an internal function that treats `Data` as an array of arbitrary values supporting the `horzcat` function. It efficiently adds `newdata` by preallocating space, maintaining `CurrentLength` to be the actual length of data stored. At the end of the message, it truncates `Response.Body.Data` to `CurrentLength`.

Subclasses can change this property if `horzcat` is not appropriate for the append process. For example, when a `StringConsumer` builds a scalar string, it would add to the string using the `plus` function instead of `horzcat`.

Subclasses that do not invoke `ContentConsumer.putData` to append data, or which are satisfied with `horzcat` behavior when appending data, can ignore this property.

Example: `@customAppend` where `@customAppend` is defined by the consumer

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`CurrentDelegate` — `ContentConsumer` to which this consumer is delegating
`matlab.net.http.io.ContentConsumer` | `[]`

The `ContentConsumer` to which this consumer is delegating, specified as a `matlab.net.http.io.ContentConsumer` object. The `delegateTo` method of the calling consumer (the delegator) sets the `CurrentDelegate` property. If there is no current delegation, then the value is `[]`.

MATLAB sets `CurrentDelegate` to `[]` before calling `initialize`.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`CurrentLength` — Length of data currently in `Response.Body.Data`
`uint64.empty` (default) | `uint64`

Length of data currently in the `Response.Body.Data` property, specified as `uint64`.

This property is used when `Response.Body.Data` has been preallocated to a size larger than the actual amount of data currently stored, to indicate the length of that stored data. If this property is empty, then it means that all of `Response.Body.Data` contains the stored data or that a `ContentConsumer` subclass is disposing of the data in some way other than storing it in `Response.Body.Data`.

This property is used and set by the `putData` method in this base class when the `AppendFcn` property is empty. It is for the benefit of subclasses that call `putData` and want to examine already-stored data, and/or any implementations of `AppendFcn` that maintain results in `Response.Body.Data`.

Subclasses that use `putData` also can modify this property to reset the position in the buffer where the data is stored. For example, when the default `AppendFcn` function is used, a subclass that processes all of `Response.Body.Data` on each call to `putData` might no longer have a use for the original data, so it can reset the `CurrentLength` property to 1 so that the next `putData` call overwrites the buffer with new data. There is no need to clear elements in the buffer past the end of the new data.

Subclasses that do not call `putData` can use this property to track their own data, or can leave it unset (empty). MATLAB does not place any constraints on the value that can be set here and does not use it for any purpose other than to determine where the default `AppendFcn` should store the next buffer of data, and where to truncate the data at the end of the message. Set this property to empty before the final call to `putData(uint8.empty)` to prevent truncation of the data.

MATLAB sets this property to empty before each call to `initialize`.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`MyDelegator` — `ContentConsumer` that delegated to this consumer
`matlab.net.http.io.ContentConsumer` | empty

The `ContentConsumer` that delegated to this consumer, specified as a `matlab.net.http.io.ContentConsumer` object. If this consumer is a delegate that

was invoked by another consumer, such as a `GenericConsumer` or `MultipartConsumer`, then this is the calling consumer. It is empty in a top-level consumer specified in the call to `send`.

Delegates can use this property to access properties of their delegators, for example, to determine which consumer delegated to them.

Attributes:

`GetAccess`	`protected`
`SetAccess`	`protected`

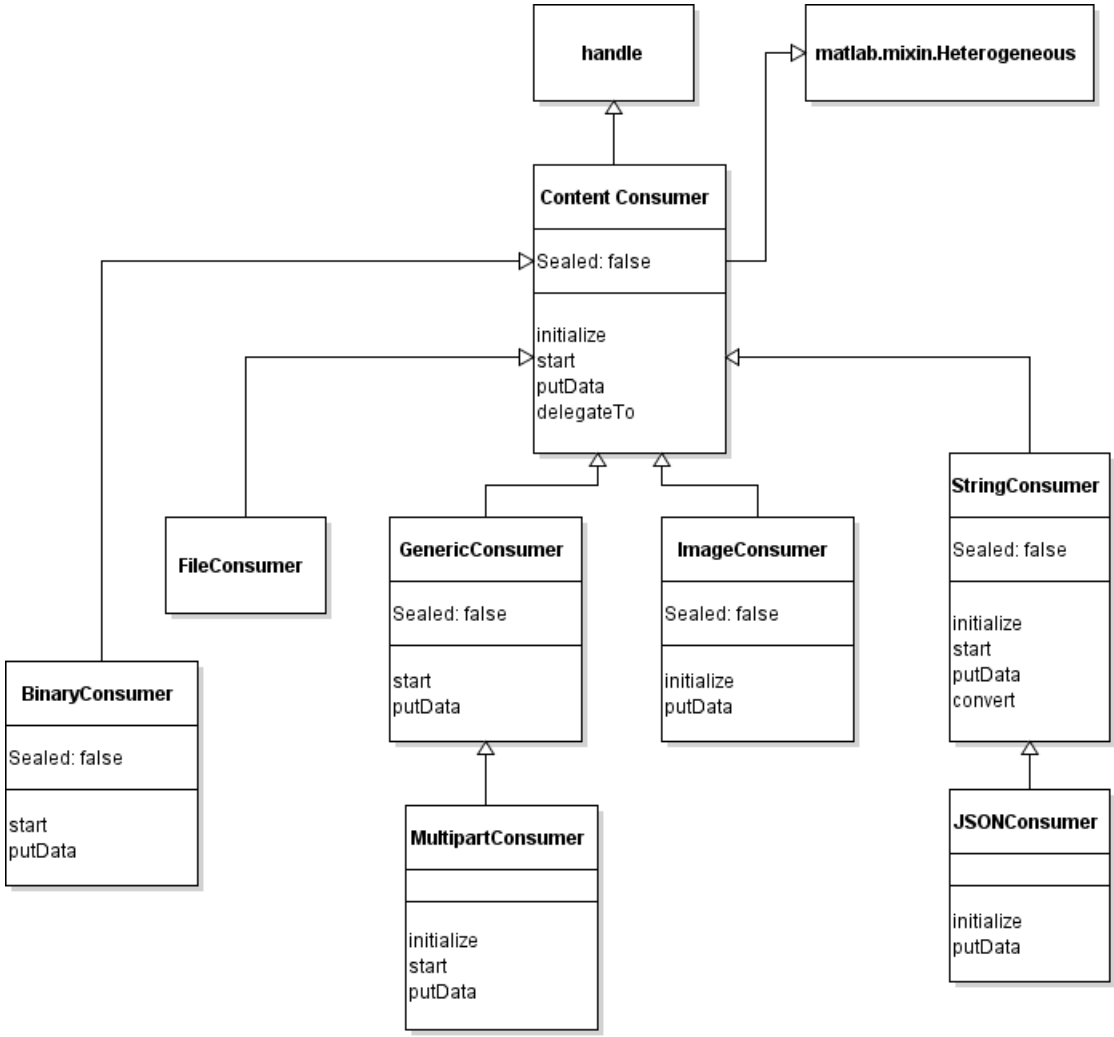
Methods

Public Methods

initialize	Prepare StringConsumer for new HTTP message
start	Start data transfer to StringConsumer
putData	Append next buffer of string data to response
convert	Convert data to string in StringConsumer
delegateTo	Delegate to another consumer

More About

Class Hierarchy



See Also

`ContentConsumer` | `ContentTypeField` | `ResponseMessage` | `StringProvider` | `initialize` | `putData`

Topics

"Display Streamed Data in Figure Window"

Introduced in R2018a

initialize

Class: `matlab.net.http.io.StringConsumer`
Package: `matlab.net.http.io`

Prepare StringConsumer for new HTTP message

Syntax

```
OK = initialize(consumer)
```

Description

`OK = initialize(consumer)` prepares this `StringConsumer` for receipt of a message.

MATLAB calls this method, which is an overridden method of `initialize`.

`initialize` returns `true` if one of the following occurs. Otherwise, it returns `false`.

- `Response.Status` is `OK` and the `Charset` property is not empty.
- ContentType is set to a `MediaType` with a known or default charset.

You can process a message with an empty Charset property or with an unknown `matlab.net.http.MediaType`. To do this, set `Charset` after creating the consumer or write a subclass that overrides this method to set `Charset` before invoking `initialize`.

See the `StringConsumer` constructor for a list of known default charsets.

Input Arguments

`consumer` — Content consumer
`matlab.net.http.io.StringConsumer`

Content consumer, specified as a `matlab.net.http.io.StringConsumer` object.

Attributes

Access	protected

See Also

ContentConsumer.ContentType | ContentConsumer.Response | `initialize` | `matlab.net.http.MediaType`

Introduced in R2018a

putData

Class: `matlab.net.http.io.StringConsumer`
Package: `matlab.net.http.io`

Append next buffer of string data to response

Syntax

```
[len,stop] = putData(consumer,data)
```

Description

`[len,stop] = putData(consumer,data)` converts `data` to a Unicode string using the current value of the Charset property and appends the results to `Response.Body.Data`. During this process the currently converted string is at `Response.Body.Data`. If the TextType property is `'char'`, then only characters up to `CurrenLength` are valid.

If `data` is `[]`, it indicates the end of the message. On return, `Response.Body.Data` contains the entire converted string or character vector.

For multibyte encodings such as UTF-8, it is possible that a given buffer of `data` ends with a partial multibyte character. In that case `Response.Body.Data` might be missing that last character, until the next call to `putData` completes it.

This method is an overridden method of `putData`. If you implement a subclass of this consumer and want to examine the raw bytes before the charset conversion, then override this method, examine `data`, change the `Charset` property if necessary, and then pass `data` to this superclass method for conversion and storage in `Response.Body.Data`. If you change `Charset` after `putData` has already been called to process previous buffers, be aware that a partial multibyte character at the end of the previous buffer that has not yet been converted could be lost. This would not occur if all characters previously received are single-byte (for example, US-ASCII or the ASCII subset of UTF-8).

A more likely scenario is that you want to examine each buffer of data as it arrives after charset conversion. To do so, override this method as follows (this works whether `TextType` is `char` or `string`):

```
function [len, stop] = putData(obj, data)
    oldLength = obj.CurrentLength;
    % send raw bytes to StringConsumer for conversion
    [len, stop] = obj.putData@matlab.net.http.io.StringConsumer(data);
    newData = obj.Response.Body.Data.extractAfter(oldLength);
    % ...process newData...
```

Now `newData` contains the most recently added data, after conversion. Note that the above pattern still stores the resulting string in `Response.Body.Data`.

If your subclass wants to stream its own results into the response after processing the string, use the `convert` method to convert your data based on the `TextType` and `Charset` in this object. In that case, call this `putData` method only at the end of the data, with an empty argument.

Input Arguments

`consumer` — Content consumer
`matlab.net.http.io.StringConsumer`

Content consumer, specified as a `matlab.net.http.io.StringConsumer` object.

`data` — Buffer of raw data in `ResponseMessage`
nonempty `uint8` vector | `uint8.empty` | `[]` (empty double)

Buffer of raw data in a `matlab.net.http.ResponseMessage` object, specified as a nonempty `uint8` vector, `uint8.empty`, or `[]`. For more information about these values, see the “data” on page 1-0 input argument for the `ContentConsumer.putData` method.

Output Arguments

`len` — Length of data processed
double | `[]` (empty double)

Length of data processed, returned as double or empty double. For more information, see the "size" on page 1-0 argument in `ContentConsumer.putData`.

`stop` — Indicate whether to receive further data
`false` (default) | `true`

Indicate whether to receive further data from this message, returned as `true` or `false`. For more information, see the "stop" on page 1-0 argument in `ContentConsumer.putData`.

Attributes

Access	public

Examples

For an example subclassing this method, see the `PricesStreamer` class `putData` method in "Display Streamed Data in Figure Window".

See Also

ContentConsumer.Response | StringConsumer.Charset | StringConsumer.TextType | `convert` | `putData`

Introduced in R2018a

start

Class: `matlab.net.http.io.StringConsumer`
Package: `matlab.net.http.io`

Start data transfer to StringConsumer

Syntax

```
bufsize = start(consumer)
```

Description

`bufsize = start(consumer)` prepares `consumer` for receipt of data. By default, `start` accumulates the converted string or character vector in `Response.Body.Data`. This method returns `[]` to indicate it has no preferred buffer size.

This method is an abstract method of `ContentConsumer`. If you override this method, you should call the `start` method as well.

Input Arguments

`consumer` — Content consumer
`matlab.net.http.io.StringConsumer`

Content consumer, specified as a `matlab.net.http.io.StringConsumer` object.

Attributes

Access	protected

Examples

For an example subclassing this method, see the `PricesStreamer` class `start` method in “Display Streamed Data in Figure Window”.

See Also

`start`

Introduced in R2018a

convert

Class: `matlab.net.http.io.StringConsumer`
Package: `matlab.net.http.io`

Convert data to string in StringConsumer

Syntax

```
str = convert(consumer,data)
```

Description

`str = convert(consumer,data)` converts a buffer of `data` to a string or character vector, `str`, based on the current values of Charset and TextType. This has the same behavior as `putData`, but returns the converted string instead of storing it in `Response.Body.Data`. It does not update CurrentLength.

This is a utility method for the benefit of subclasses that want to interpret the data as a string, and then process the results and store their own data in `Response.Body.Data`. Subclasses that use this method should not call `putData` except to pass in empty `data` at the end of the stream to tell this class that input has ended.

If `data` ends with a partial multibyte character, that partial character is saved internally and not returned until the next call to `convert` that provides the remainder of the bytes.

Input Arguments

`consumer` — Content consumer
`matlab.net.http.io.StringConsumer`

Content consumer, specified as a `matlab.net.http.io.StringConsumer` object.

`data` — Buffer of raw data
nonempty `uint8` vector | `uint8.empty` | `[]` (empty double)

Buffer of raw data in `ResponseMessage`, specified as a nonempty `uint8` vector, `uint8.empty`, or `[]`.

Output Arguments

`str` — Text
string | character vector

Text, returned as a string or character vector.

Attributes

Access	protected

See Also

Charset | TextType | `putData`

Introduced in R2018a

matlab.net.http.io.ContentProvider class

Package: `matlab.net.http.io`
Superclasses: `handle`, `matlab.mixin.Heterogeneous`

ContentProvider for HTTP message payloads

Description

A ContentProvider supplies data for an HTTP `RequestMessage` while the message is being sent. A simple provider converts data from a MATLAB type to a byte stream. More complex providers can stream data to the server, obtaining or generating the data at the same time it is being sent, which avoids the need to have all the data in memory before the start of the message.

Normally, when sending data to a web service (typically in a PUT or POST request), you would create a `RequestMessage` and insert data in the form of a `MessageBody` object in the `RequestMessage.Body` property. When you send that message using `RequestMessage.send`, MATLAB converts that data into a byte stream to be sent to the server, converting it based on the Content-Type of the message and the type of data in `Body.Data`. See `MessageBody.Data` for these conversion rules.

Instead of inserting a `MessageBody` object into the `RequestMessage.Body` property, you can create a `ContentProvider` object and insert that instead. Then, when you send the message, MATLAB calls methods in the `ContentProvider` to obtain buffers of data to send, while the message is being sent.

Whether you insert a `MessageBody` or a `ContentProvider` into the message, the call to `RequestMessage.send` does not return (that is, it is blocked) until the entire message has been sent and a response has been received, or an error has occurred. But with a `ContentProvider`, MATLAB makes periodic callbacks into the provider to obtain buffers of data to send, during the time send is blocked. In these callbacks, your `ContentProvider` can obtain data from any source such as a file, a MATLAB array, a hardware sensor, a MATLAB function, etc. The provider's job is to convert that data to a byte stream, in the form of uint8 buffers, that can be sent to the web.

`ContentProvider` is an abstract class designed for class authors to subclass with their own data generator or converter, or you can use (or subclass) one of the MATLAB

providers that generate the data for you from various sources, without writing a subclass. These providers have options that give you more flexible control over how data is obtained and converted, compared to the automatic conversions that occur when you insert data directly into a `MessageBody`. Use one of the `ContentProvider` subclasses:

- `FileProvider`
- `StringProvider`
- `JSONProvider`
- `ImageProvider`
- `FormProvider`
- `MultipartProvider`
- `MultipartFormProvider`
- `GenericProvider`

Even if you do not need to stream data, using one of these providers can simplify the process of sending certain types of content, as they convert data from an internal form into a `uint8` stream. For example, `FormProvider` lets you send form responses to a server, where you can conveniently express the data as an array of `QueryParameter` objects. `MultipartFormProvider` lets you send multipart form responses, simplifying the creation of responses to multipart forms. To use any `ContentProvider`, you need to understand the type of content that the server expects you to send.

The `matlab.net.http.io.ContentProvider` class is a `handle` class.

Subclass Authors

The simplest possible `ContentProvider` need only implement a `getData` method to provide buffers of data as MATLAB requests them. To use your provider, insert it into in the `Body` property of the `RequestMessage`. In this example, the third argument to the `RequestMessage` constructor, a `MyProvider` object, goes into the `Body`:

```
provider = MyProvider;
req = matlab.net.http.RequestMessage('put', headers, provider);
resp = req.send(uri);
```

Here is an example a `MyProvider` class that reads from a file name passed in as an argument to the constructor and sends it to the web. For good measure, we close the file at the end or when this provider is deleted.

```
classdef MyProvider < matlab.net.http.io.ContentProvider
    properties
        FileID double
    end

    methods
        function obj = MyProvider(name)
            obj.FileID = fopen(name);
        end

        function [data, stop] = getData(obj, length)
            [data, len] = fread(obj.FileID, length, '*uint8');
            stop = len < length;
            if (stop)
                fclose(obj.FileID);
                obj.FileID = [];
            end
        end

        function delete(obj)
            if ~isempty(obj.FileID)
                fclose(obj.FileID);
                obj.FileID = [];
            end
        end
    end
end
```

MATLAB calls a provider's `complete` method when it is forming a new message to send. The purpose is to allow the provider to prepare for a new message and add any required header fields to the message. MATLAB calls a provider's `start` method when it is time to send the data, but before the first call to `getData`.

Properties

Public Properties

`Header` — Header fields of message or part

`matlab.net.http.HeaderField.empty` (default) | `matlab.net.http.HeaderField`

Header fields of the message or part, specified as a vector of one or more `matlab.net.http.HeaderField` objects.

This property is only used by subclass authors. MATLAB sets this property before calling the provider's `complete` method. For non-multipart messages, MATLAB initializes this property to the contents of `Request.Header`, minus any `matlab.net.http.field.GenericFields` or empty-valued fields. The `ContentProvider` uses this property to add header fields that describe the data to be sent, or to add parameters to header fields already in the message. In a delegate for a `MultipartProvider`, MATLAB initializes this property to header fields that the delegating provider intends to insert for the part. Delegates can modify or change these fields.

Upon return from the provider's `complete` method, if this not a multipart message, then MATLAB reads this property and merges its contents into the header of `Request`. Fields in this `Header` with `Names` that do not already appear in `Request.Header` are added to the end of `Request.Header`. If a field in this `Header` has a `Name` that is the same as one in `Request.Header`, and both have nonempty `Values`, then:

If one or both of these has an empty `Value`, then the field is removed from `Request.Header` and it is not added as part of normal message completion.

If this is a delegate of a `MultipartProvider`, then the entire contents of this `Header` is used as the header of the part. Multipart delegates must not assume that `Request.Header` contains any fields pertaining to their own `Header`. A provider can determine whether it is a multipart delegate by checking whether `MyDelegator` is a `MultipartProvider`, though this test is unlikely to be needed.

MATLAB reads this property only on return from calling the provider's `complete` method. Changes to this array are ignored once MATLAB calls `start`.

Class authors should be aware that their subclasses might have added fields to this `Header` (in their `complete` method) before calling `complete` in their superclass. It is best to preserve such fields and not to add fields with the same names. However, adding a parameter to a field is permissible. For example, a superclass can add a charset parameter to an existing Content-Type field that does not already have one.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`ForceChunked` — Indicate whether to force chunked transfer coding
`false` (default) | `true`

Indicate whether to force chunked transfer coding, specified as boolean. This property is of interest only to subclass authors, and is applicable only to providers that are not multipart delegates. Subclasses set `ForceChunked` to control whether contents should be sent using chunked transfer coding. If `false` (default), MATLAB decides whether to send the contents chunked, based on whether it knows the content length at the time the message is ready to be sent:

If `ForceChunked` is `true`, then MATLAB sends the message chunked regardless of whether it knows the content length, unless the known length is smaller than the chunk size. If this property is `true`, then the message must not contain a Content-Length field, because HTTP does not allow a chunked message to have a Content-Length field. However, you can still return a nonzero value in the `expectedContentLength` method if you want MATLAB to verify that you are returning the expected length of data.

When MATLAB chooses to send the message chunked, the size of each chunk is equal to the length of data returned by `getData`.

MATLAB reads this value after calling the `complete` method, before calling `start`. It does not set this field.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Request` — Request message to send
`matlab.net.http.RequestMessage`

Request message to send, specified as a `matlab.net.http.RequestMessage` object.

This property is used only by subclass authors. The `RequestMessage.send` and `RequestMessage.complete` methods set this property to the `RequestMessage` in whose `Body` this provider has been placed, before calling any other methods in this provider, and before adding any additional header fields or validating the message. The provider can examine this message to see what was contained in the original request.

Delegates see the same value for this property as the delegator. `ContentProviders` should be aware that, if they are delegates, they are not necessarily providing the entire body of the request message, so they should not assume that header fields in this Request are pertinent to the data they are providing. Usually, delegates should ignore header fields in this request relevant to the data, such as Content-Type.

If the provider wishes to add any header fields to this message, or to modify existing ones, it should do so in its `complete` method by adding those fields to the `Header` property. The caller of `complete` (`RequestMessage` or a delegating provider) determines what to do with those fields. `RequestMessage.send` and `RequestMessage.complete` always copy these fields to the `Header` of the `RequestMessage`. A delegating provider can copy the fields to its own `Header` property or insert them into the message (as in the case of `MultipartProvider`). For more information, see the ContentProvider.Header property.

This property is read-only.

Attributes:

GetAccess	public
SetAccess	matlab.net.http.RequestMessage

Protected Properties

`CurrentDelegate` — `ContentProvider` to which this provider is delegating
`matlab.net.http.io.ContentProvider` | empty

`ContentProvider` to which this provider is delegating, specified as a `matlab.net.http.io.ContentProvider` object. This property is set in the calling provider (the delegator) by the `delegateTo` method to indicate the current delegated provider. If there is no current delegation, then the value is empty.

The `complete` methods set this property to empty.

Attributes:

GetAccess	protected
SetAccess	protected

`MyDelegator` — `ContentProvider` that delegated to this provider
`matlab.net.http.io.ContentProvider.empty` (default) | `matlab.net.http.io.ContentProvider`

`ContentProvider` that delegated to this provider, specified as a `matlab.net.http.io.ContentProvider` object.

If a `ContentProvider` delegates responsibility for sending all or a portion of the message data to another provider, then this property identifies the delegating provider to

the delegate. For example, a `MultipartProvider` delegates parts of the message to other providers, so it inserts a handle to itself in each delegate. Otherwise, `MyDelegator` is empty. The `delegateTo` method sets this property in the delegate.

Attributes:

`GetAccess`	`protected`
`SetAccess`	`protected`

Methods

Public Methods

string	Show ContentProvider as string
show	Display ContentProvider information

Protected Methods

complete	Complete HTTP header for ContentProvider
preferredBufferSize	Preferred buffer size for ContentProvider
expectedContentLength	Content length of ContentProvider
start	Start data transfer from ContentProvider
restartable	Indicate whether ContentProvider is restartable
reusable	Indicate whether ContentProvider is reusable
delegateTo	Delegate to another provider

Abstract Methods

getData	Next buffer of data to send in HTTP request message from ContentProvider

More About

Restartability and Reusability

A provider can be restartable and/or reusable. Restartable means that the provider is able to resend the same message multiple times, with the same data stream each time MATLAB calls `start`, even if the previous use did not end in a normal completion. This behavior is needed because the server can redirect a message to a different server, which means the data needs to be retransmitted. In that case MATLAB calls `start` without

calling `complete` again. MATLAB calls the `restartable` method to determine whether a provider can be restarted. If false, MATLAB throws an exception if it needs to call `start` on a provider that has already been started, if there was no intervening call to `complete` (which happens only on a new message).

Reusable means that the provider can be reused for a different (or the same) message, each time MATLAB calls its `complete` method. MATLAB calls the `reusable` method to determine whether a provider can be reused. If `false`, then MATLAB throws an exception if it needs to call `complete` on a provider that has already been started. If a provider is reusable, then the assumption is that the next call to `start` should succeed, even if the provider is restartable.

`ContentProvider` returns `false` for both `restartable` and `reusable`, so if you are extending this base class directly with a restartable or reusable provider, you should override one or both of these methods to return `true`. All concrete subclasses of `ContentProvider` in the `matlab.net.http.io` package are both restartable and reusable, so they return true for these methods. If you are extending one of those subclasses with a provider that is not reusable or restartable, override one or both of those methods to return false.

The `MyProvider` class in this example is not restartable or reusable, because the provider closes the file at the end of the message. To make it reusable, the `fopen` call should take place in the `complete` method instead of the constructor, thereby restoring the provider's state back to what it was before it was used for a message.

```
classdef MyProvider < matlab.net.http.io.ContentProvider
   properties
       FileID double
       Name string
   end

   methods
       function obj = MyProvider(name)
           obj.Name = name;
       end

       function [data, stop] = getData(obj, length)
           ...as above...
       end

       function complete(obj, uri)
           obj.FileID = fopen(name);
           obj.complete@matlab.net.http.io.ContentProvider();
```

```
            end

            function tf = reusable(~)
                tf = true;
            end

            function delete(obj)
                ...as above...
            end
        end
    end
```

To make the provider restartable, add `restartable` and `start` methods and issue an `fseek` in the `start` method to "rewind" the file:

```
            function start(obj)
                obj.start@matlab.net.http.io.ContentProvider();
                fseek(obj.FileID, 0, -1);
            end

            function tf = restartable(~)
                tf = true;
            end
```

When you call `complete` or `send` on a `RequestMessage` that contains a `ContentProvider` in its body, MATLAB sets the `Request` property in the provider to the `RequestMessage` in which the provider was placed and the `Header` property to the headers in the `Request`, before adding automatic fields. It then calls the following methods in the provider, in this order:

- `complete` - called on message completion, which usually happens once per message, when you call `RequestMessage.send` or `RequestMessage.complete`. The provider is expected to set its `Header` property to any header fields to be added to the message specific to the provider. If MATLAB calls this method a subsequent time, the provider should assume it is being used for a new message. Most providers need to implement this method to add their headers and then, if they are not a direct subclass of this abstract class, they should call their superclass complete to invoke any additional default behavior. MATLAB does not call `complete` more than once in a provider, unless its reusable method returns true. This abstract class is not reusable by default, but all concrete providers in the `matlab.net.http.io` package are reusable.
- `preferredBufferSize/expectedContentLength` - called from `RequestMessage.send`, sometime after `complete`, before a call to `start`. Most

providers need not implement these methods, as the default behavior is appropriate. However, providers can override this to support the `force` argument.

After return from these methods, MATLAB sends the header of the `RequestMessage` to the server. When it is time to send the body, MATLAB calls these methods.

- `start` - called from `RequestMessage.send`, sometime after calling the previous methods, when MATLAB has determined that the server is ready to receive the body of the request message. If MATLAB calls this a subsequent time, without an intervening complete, the provider should assume it is being asked to resend the body of the same message (with the same headers) once again. MATLAB does not call start more than once since the last call to `complete`, unless the provider's `restartable` method returns `true`. This abstract class is not restartable by default, but all concrete providers in the `matlab.net.http.io` package, are restartable.
- `getData` - called multiple times after the call to `start`, while `RequestMessage.send` is blocked, each time MATLAB determines that the server is ready for a new buffer of data. The method must return a `uint8` vector of data. The provider signals the end of the data by returning a `stop` indicator. All providers must implement this method.

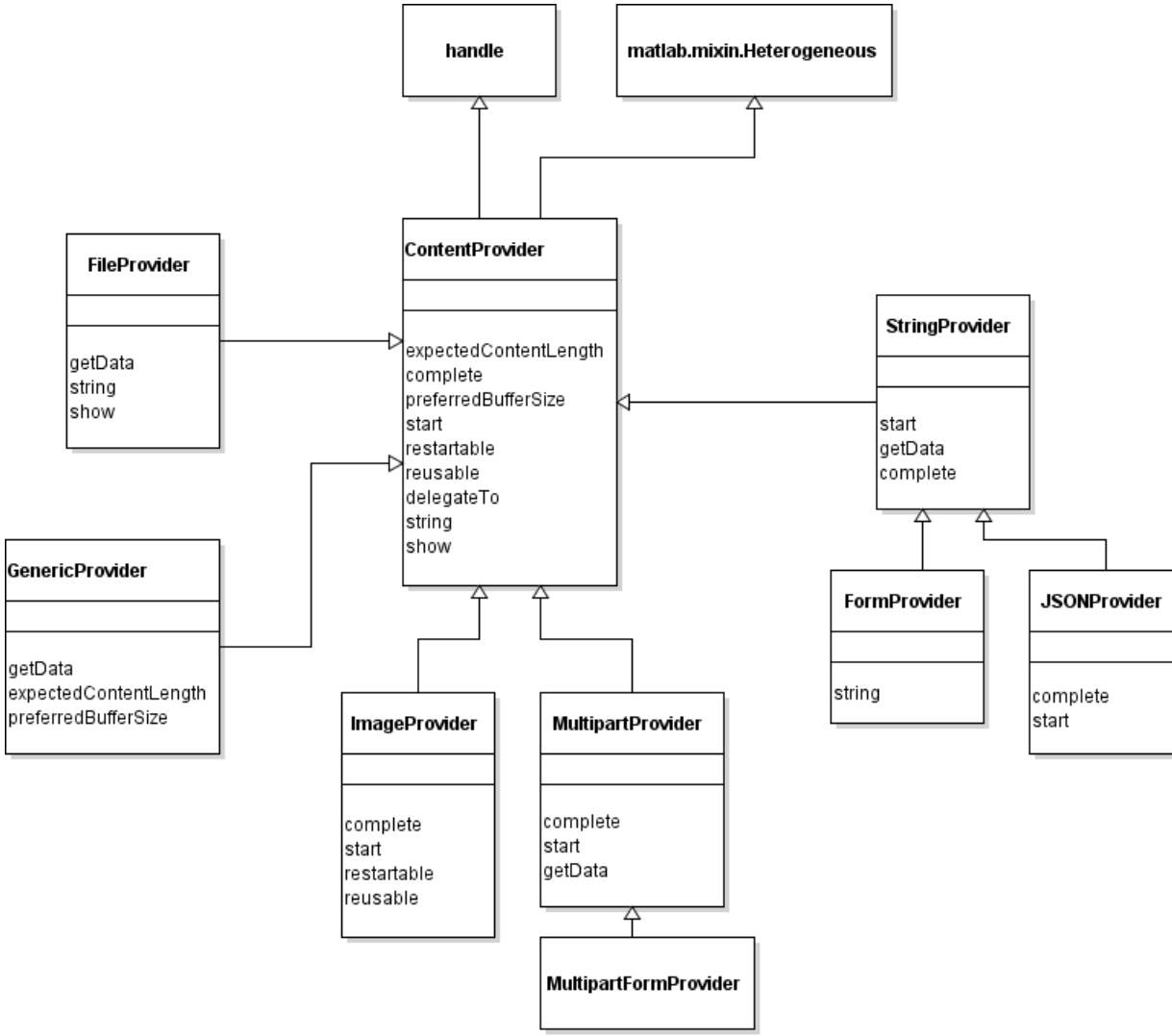
After `getData` returns a `stop` indicator, MATLAB ends the request message and awaits a response from the server.

Delegation

A `ContentProvider` that is inserted into a `RequestMessage.Body` can delegate to one or more other providers to provide all or some of the data for the message. For example, a `MultipartProvider` creates a message with multiple parts, each of which are provided by various other providers specified to the `MultipartProvider` constructor. In this case, `MultipartProvider` is the delegator, and the other providers are the delegates, each one being called in turn to provide its own header fields and its portion of the data.

A provider delegates to another by calling `delegateTo`, which sets `CurrentDelegate` to the delegate and the delegate's `MyDelegator` to the current provider (that is, the delegator), and then calls the delegate's `complete` and `start` methods and returns a `GetDataFcn` function handle. Then the delegator's `getData` method calls the delegate's `GetDataFcn` to obtain the data, possibly altering it before returning it to MATLAB. Providers generally do not have to check whether they are delegates, or who delegated to them.

Class Hierarchy



See Also

`ContentConsumer` | `FileProvider` | `FormProvider` | `ImageProvider` | `JSONProvider` | `MessageBody` | `MultipartFormProvider` | `MultipartProvider` | `QueryParameter` | `RequestMessage` | `StringProvider`

Introduced in R2018a

complete

Class: `matlab.net.http.io.ContentProvider`
Package: `matlab.net.http.io`

Complete HTTP header for ContentProvider

Syntax

```
complete(provider,URI)
```

Description

`complete(provider,URI)` augments the header of the message with header fields required by this provider. The `RequestMessage.send` and `RequestMessage.complete` methods call this method before validating the header or adding any default fields, and before calling other methods in this class except for `expectedContentLength`.

This is where subclasses can add any fields to Header that depend on the content, such as Content-Type. See the description of the `Header` property for more information.

The `RequestMessage` methods do not call this method if the message has already been completed (that is, if `RequestMessage.Completed` is `true`). However, a subsequent change to the message after completion resets the `RequestMessage.Completed` property, allowing those methods to invoke this method again. Therefore, providers should be prepared for more than one call to `complete` before a call to `start`. Once `start` has been called, MATLAB does not reinvoke `complete` in this provider unless `reusable` returns `true` to indicate that this provider can be reused for another message.

A `ContentProvider` that extends another `ContentProvider` should first call its superclass `complete` method to add header fields to `Header` that the superclass needs, and then, on return, modify those fields if desired.

The default behavior of this method does nothing, but throws an exception if this provider has been started and is not reusable. Providers that override this method should always invoke their superclass.

If this provider is not a multipart delegate, and you want to include a Content-Length field in the message (thereby avoiding chunked transfer coding), then you should return a nonempty value in `expectedContentLength` or implement this method to insert a Content-Length field in the `Header`.

This method is not invoked on messages with a `Completed` property set to `true`, which generally means that this method is invoked only once per message, even if this message is resent multiple times. Implementations of this method should, therefore, perform any initialization that needs to be done only once per message. Costly initialization that does not need to be done until the data is ready to be sent should be performed in the `start` method.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.ContentProvider`

Content provider, specified as a `matlab.net.http.io.ContentProvider` object.

`URI` — Uniform resource identifier
`matlab.net.URI`

Uniform resource identifier, specified as a `matlab.net.URI` object.

Attributes

`Access`	`protected`

Examples

Extend Class Provider

The following is a coding pattern for a `ContentProvider` that extends a `SuperclassProvider` and adds the `HeaderField` `myField` .

```
function complete(obj, uri)
            complete@SuperclassProvider(obj, uri);
```

```
field = obj.Header.getFields('My-Field');
if isempty(field)
    myField = HeaderField('My-Field', value);
    obj.Header = obj.Header.addFields(myField);
end
```

See Also

ContentProvider.Header | ContentProvider.Request | `expectedContentLength` | `matlab.net.URI` | `matlab.net.http.RequestMessage` | `start`

Introduced in R2018a

delegateTo

Class: `matlab.net.http.io.ContentProvider`
Package: `matlab.net.http.io`

Delegate to another provider

Syntax

```
[getDataFnc,length] = delegateTo(provider,delegate,URI)
[getDataFnc,length] = delegateTo( ___ ,force)
```

Description

`[getDataFnc,length] = delegateTo(provider,delegate,URI)` sets up a `delegate` `ContentProvider` to provide all or part of the subsequent data in a `RequestMessage`. This method initializes properties in the delegate using properties of this object and supplied parameters, as if a new message was about to be transmitted using that delegate, and invokes the `complete`, `expectedContentLength`, and `start` methods in the `delegate`. It returns a handle to a function, `getDataFnc`, that you invoke to obtain data from the delegate:

```
[data, stop] = getDataFcn(length)
```

where the arguments are as described for `getData`. You can use any value of `length`, but normally you make this call in your `getData` method, passing the same value that was passed to your method.

To delegate to a provider that provides the entire contents of a message, call `delegateTo` in your `start` method. If you use the delegate to obtain part of the message content, then call `delegateTo` at the appropriate time in your `putData` method.

To obtain data from the delegate, always use the returned `getDataFnc`. Do not call the delegate's `getData` directly because the delegate might provide its data through some other means.

`[getDataFnc,length] = delegateTo( ___ ,force)` indicates if the `expectedContentLength` method should return the length of the data.

Input Arguments

`provider` — Content Provider
`matlab.net.http.io.ContentProvider`

Content provider, specified as a `matlab.net.http.io.ContentProvider` object.

`delegate` — Provider for subsequent data
`matlab.net.http.io.ContentProvider`

Provider for subsequent data, specified as a `matlab.net.http.io.ContentProvider` object.

`URI` — Request destination
`matlab.net.URI`

Destination of the request being processed, specified as a `matlab.net.URI` object. The value is the URI provided to the caller's `complete` method.

`force` — Indicate whether to return length of data
`false` (default) | `true`

Indicate whether the `expectedContentLength` method should return the length of data, specified as `true` or `false`. The `delegateTo` method passes this value to `expectedContentLength`.

Output Arguments

`getDataFnc` — Function to get data
function handle

Function to get data from a delegate, returned as a function handle.

`length` — Value returned by `expectedContentLength`
`[ ]` (default) | bytes

Value returned by the `expectedContentLength` method, returned as bytes. For more information, see the `length` output argument for `ContentProvider.expectedContentLength`.

Attributes

Access	protected

See Also

complete | expectedContentLength | getData | matlab.net.URI | start

Introduced in R2018a

expectedContentLength

Class: `matlab.net.http.io.ContentProvider`
Package: `matlab.net.http.io`

Content length of ContentProvider

Syntax

```
length = expectedContentLength(provider)
length = expectedContentLength(provider,force)
```

Description

`length = expectedContentLength(provider)` returns the expected content length in bytes. This method is intended to be overridden by subclasses that want to report their content length to MATLAB. `RequestMessage.send` and `RequestMessage.complete` call this method and use the return value to set the Content-Length header field in the `RequestMessage`. If the message already has a Content-Length field with a value, and `length` is nonempty, then its value must be equal to the value in that Content-Length field. `length` might be 0 to indicate there is no contents, in which case the first call to `getData` should return empty `data` and `stop=true`.

MATLAB calls this method from `RequestMessage.send`, `RequestMessage.complete` and in the delegate by `delegateTo`. MATLAB calls this after `ContentProvider.complete` and before `ContentProvider.start`. If this method is called before calling `complete`, then the return value might be invalid, because a provider cannot necessarily determine the length of its converted data without seeing all the header fields that control the conversion.

If you do not choose to have a Content-Length header field in your message (the message is being sent using chunked transfer coding), then the only reason to override this method and return a nonempty value is as a double-check to insure that your provider returns the expected length of data.

In cases where the length of the data is known (that is, when this method returns a number or the Content-Length field is nonempty), this provider's `getData` method must

return `stop=true` after exactly that number of bytes have been returned. MATLAB always calls `getData` repeatedly, even if `length=0`, until `getData` returns `stop=true`. In cases where the length is not known, if this is a top level provider (not a multipart delegate), then MATLAB uses chunked transfer coding to send the contents and the provider is free to return any length of data, including none, prior to setting `stop=true`.

You should return `[]` if you do not know the length of the data in advance, or if computing the length of the data would be time-consuming. It is harmless (and perfectly normal) to allow any message to use chunked transfer coding, even if you know the length. If this provider is a multipart delegate, a nonempty return value is only used to force an error in case `getData` returns more or fewer bytes, and will not cause a Content-Length header field to appear in the part. See `MultipartProvider` for more information.

`length = expectedContentLength(provider,force)`, if `force` is `true`, requires that you return the length of the data, computing it if necessary, even if you would otherwise return `[]`, unless computing the length is impossible. If returning this number requires a lengthy computation or generation of all the data in the message, then you should cache the data so that you do not have to recompute it in subsequent `getData` calls. The `force` argument is provided for use by subclasses who must know the length of the data in advance. MATLAB never sets this option when calling this method, and if you know that your provider is never used as a subclass that might set this option, then you can ignore the `force` argument.

Callers of this method who get `[]` in response to setting `force` to `true` can either consider it an error, or behave in a way that is compatible with content of unknown length.

Specifying `force` can negate the benefit of streaming (sending data as it is being generated) if it requires all the data to be generated to compute `length`, so this option is best used for special cases, e.g. debugging, or when the length of data is known to be small.

An example of the use of `force` is a hypothetical `CompressProvider` that optionally compresses the output of any other provider, but only if that output is greater than a certain length (because compression is inefficient for short messages). To determine the length, the `CompressProvider` needs to invoke the other provider's `expectedContentLength` with `force` set to `true`. If that other provider is a streaming `JSONProvider`, `expectedContentLength` normally returns `[]`, because determining the length of a JSON string requires processing all of the input data. With `force` set to `true`, the `JSONProvider`'s `expectedContentLength` method processes all of the data

(perhaps caching the output string internally for later use by its `putData` method), and returns that string's length.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.ContentProvider`

Content provider, specified as a `matlab.net.http.io.ContentProvider` object.

`force` — Indicate whether to return length of data
`false` (default) | `true`

Indicate whether to return the length of the data, specified as `true` or `false`.

If `true`, the `expectedContentLength` must return the length of the data, computing it if necessary, even if you would otherwise return `[]`, unless computing the length is impossible. If returning this number requires a lengthy computation or generation of all the data in the message, then you should cache the data so that you do not have to recompute it in subsequent `getData` calls. The `force` argument is provided for use by subclasses who must know the length of the data in advance. MATLAB never sets this option when calling this method, and if you know that your provider is never used as a subclass that might set this option, then you can ignore the `force` argument.

Callers of this method who get `[]` in response to setting `force` to `true` can either consider it an error, or behave in a way that is compatible with content of unknown length.

Specifying `force` can negate the benefit of streaming (sending data as it is being generated) if it requires all the data to be generated to compute `length`, so this option is best used for special cases, for example, debugging, or when the length of data is known to be small.

Output Arguments

`length` — Expected content length
`[]` (default) | bytes

Expected content length, in bytes. If you do not override this method, then `expectedContentLength` returns `[]`. MATLAB determines the content length as follows.

- If this `ContentProvider` is not a multipart delegate (see `MultipartProvider`), and the message has a Content-Length field with a nonempty value (inserted in the original `RequestMessage` or added to the `Header` property by the `complete` method), then that Content-Length field is the length of the contents.
- If there is no Content-Length field (or this provider is a multipart delegate), then the payload (or data in the part) ends when this provider's `getData` method sets the `stop` return value. In that case, the content length need not be specified.

Attributes

`Access`	`protected`

See Also

ContentProvider.Header | ContentProvider.Request | `JSONProvider` | `MultipartProvider` | `complete` | `getData` | `matlab.net.http.RequestMessage`

Introduced in R2018a

getData

Class: `matlab.net.http.io.ContentProvider`
Package: `matlab.net.http.io`

Next buffer of data to send in HTTP request message from ContentProvider

Syntax

```
[data,stop] = getData(provider,length)
```

Description

`[data,stop] = getData(provider,length)` returns a buffer of data to send in an HTTP request message.

MATLAB calls this method multiple times during `RequestMessage.send`, after calling `start`, and sends each buffer of `data` to the server immediately. If the message is chunked (for example, `expectedContentLength` returned empty and there is no Content-Length field in the message), then the size of the chunk is the length of `data`.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.ContentProvider`

Content provider, specified as a `matlab.net.http.io.ContentProvider` object.

`length` — Length of data
double

Length of data that the provider should return for optimum interactive behavior, specified as double.

The value is a suggested length based on the value of `preferredBufferSize`, if specified, and the internal buffer sizes. The provider can, however, return more or fewer

bytes, and if your provider wants to send chunks of specific sizes, it can ignore `length`. MATLAB does not guarantee that any specific value of `length` is specified, but it always is a finite number greater than zero. Returning a large buffer of `data` might cause MATLAB to block for a considerable time while sending the data, during which you cannot interrupt the operation using **Ctrl+C**. This might not be an issue for non-interactive applications, where larger buffers are efficient.

Output Arguments

`data` — Next buffer of data
`uint8` vector | empty

Next buffer of data, returned as a `uint8` vector or empty. If `data` is empty and `stop` is not set, then MATLAB calls this method repeatedly to get more data (after a small delay). To end the message, return `stop=true`. However, you can also throw an exception to abort the message, which is returned to the caller of `RequestMessage.send`.

If the Content-Length header field was included in the message header or returned by `expectedContentLength` (that is, the message is not being sent using chunked transfer coding), then the total number of bytes returned in `data` over multiple calls, ending with `stop=true`, must be equal to that number. If `stop=true` is returned prematurely, or the total amount of `data` returned is greater than that number, then MATLAB throws an exception and closes the connection.

`stop` — Indicate whether to end transmission
`false` (default) | `true`

Indicate whether to end transmission, returned as a logical that the provider must set. If `false`, then MATLAB calls this `getData` again to get more data when it is ready to send the next buffer. If `true`, then this indicates that the provider has no more data to send, beyond what is returned in `data`, and tells MATLAB to end the message. This is the normal way to end the `RequestMessage` and prepare MATLAB to receive a `ResponseMessage`.

Attributes

`Access`	`public`

See Also

`MultipartProvider` | `expectedContentLength` | `matlab.net.http.RequestMessage` | `preferredBufferSize` | `start`

Introduced in R2018a

preferredBufferSize

Class: `matlab.net.http.io.ContentProvider`
Package: `matlab.net.http.io`

Preferred buffer size for ContentProvider

Syntax

```
size = preferredBufferSize(provider)
```

Description

`size = preferredBufferSize(provider)` returns the preferred size of data buffers of this provider. MATLAB should use this value in the `length` parameter to the `getData` method.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.ContentProvider`

Content provider, specified as a `matlab.net.http.io.ContentProvider` object.

Output Arguments

`size` — Preferred buffer size
`[]` (default) | bytes

Preferred buffer size, returned as bytes. By default this returns `[]`, which indicates that this provider does not care what size of buffers are requested and MATLAB should choose a size. Since `getData` can always return fewer or more bytes than this, this value is an optimization to minimize the number of `getData` calls and amount of data copying that might take place.

See Also

`expectedContentLength` | `getData`

Introduced in R2018a

restartable

Class: `matlab.net.http.io.ContentProvider`
Package: `matlab.net.http.io`

Indicate whether ContentProvider is restartable

Syntax

```
TF = restartable(provider)
```

Description

`TF = restartable(provider)` indicates whether `ContentProvider` is restartable. The method returns `true` if the `ContentProvider` can restart transmission of the same data, by accepting a subsequent call to the `start` method without an intervening call to `complete`. For more information on restartability, see the class description of `ContentProvider`. By default, `restartable` returns `false`, but most concrete subclasses return `true`.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.ContentProvider`

Content provider, specified as a `matlab.net.http.io.ContentProvider` object.

Attributes

Access	protected

See Also

`ContentProvider` | `complete` | `reusable` | `start`

Introduced in R2018a

reusable

Class: `matlab.net.http.io.ContentProvider`
Package: `matlab.net.http.io`

Indicate whether ContentProvider is reusable

Syntax

```
TF = reusable(provider)
```

Description

`TF = reusable(provider)` indicates whether `ContentProvider` is reusable. The method returns `true` if the `ContentProvider` can be reused for a new message, by accepting a subsequent call to the `complete` method. For more information on reusability, see the class description of `ContentProvider`. By default `reusable` returns `false`, but most concrete subclasses return `true`.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.ContentProvider`

Content provider, specified as a `matlab.net.http.io.ContentProvider` object.

Attributes

Access	protected

See Also

`ContentProvider` | `complete` | `restartable` | `start`

Introduced in R2018a

show

Class: `matlab.net.http.io.ContentProvider`
Package: `matlab.net.http.io`

Display ContentProvider information

Syntax

```
show(provider)
show(provider,maxlength)
str = show( ___ )
```

Description

`show(provider)` displays information about the provider and possibly its data.

Use this method for debugging. If the `Body` property in a `RequestMessage` contains this provider, then MATLAB calls `show`. In this abstract class, `show` returns the value of the `string` method, but limits the output to `maxlength` characters.

`show(provider,maxlength)` displays up to `maxlength` characters of data. If the data is longer than `maxlength`, then `show` displays a message indicating the total length in characters.

`str = show( ___ )` returns the information as a string.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.ContentProvider`

Content provider, specified as a `matlab.net.http.io.ContentProvider` object.

`maxlength` — Number of characters
double

Number of characters to display, specified as double.

Attributes

Access	public

See Also

string

Introduced in R2018a

start

Class: `matlab.net.http.io.ContentProvider`
Package: `matlab.net.http.io`

Start data transfer from ContentProvider

Syntax

```
start(provider)
```

Description

`start(provider)` is called each time MATLAB is ready to start the transfer of a data stream by calling `getData` one or more times. Each time this is called, the provider is expected to reset so that the next call to `getData` goes back to the beginning of the data stream.

Calling `start` indicates that a connection to the server has been established and transfer of data is about to start. A subsequent call to `start` (without an intervening call to the `complete` method) might indicate that the server requested a redirect to a different server, or requires another try with authentication credentials, and this could occur before, during, or after transmission of the data stream.

If your provider is restartable, reset your provider so that the next call to `getData` returns to the beginning of the data stream, and insure that the `restartable` method returns `true`.

If your provider is not restartable for the same message, but can be reused for a new message, insure `restartable` returns false.

Subclasses that override this method should always call their superclass method first. If the provider was already started and is not restartable, then `start` throws an exception by default.

This method is the best place to implement costly initialization that is not needed until the server is ready to receive data, as opposed to the `complete` method, which must do

initialization necessary to create the message header. If the server cannot be contacted or rejects the message, then MATLAB does not call `start`.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.ContentProvider`

Content provider, specified as a `matlab.net.http.io.ContentProvider` object.

See Also

`complete` | `getData` | `restartable` | `reusable`

Introduced in R2018a

string

Class: `matlab.net.http.io.ContentProvider`
Package: `matlab.net.http.io`

Show ContentProvider as string

Syntax

```
str = string(provider)
```

Description

`str = string(provider)` returns information about the provider as a string.

Use this method for debugging. It is also invoked by calling in `string` in `MessageBody` or `RequestMessage` that contains this provider. In this abstract class, `string` returns the class of the provider. Subclasses can override this method to return the provider data, if any, or other information about the provider.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.ContentProvider`

Content provider, specified as a `matlab.net.http.io.ContentProvider` object.

Output Arguments

`str` — Data in text format
`string`

Data in text format, returned as a string.

Attributes

Access	public

See Also

matlab.net.http.MessageBody | matlab.net.http.RequestMessage

Introduced in R2018a

matlab.net.http.io.FileProvider class

Package: `matlab.net.http.io`
Superclasses: `matlab.net.http.io.ContentProvider`, `matlab.mixin.Copyable`

ContentProvider to send files

Description

Use a `FileProvider` object as a convenient way to send one or more files to a server.

The `matlab.net.http.io.FileProvider` class is a `handle` class.

Creation

Description

`providers = FileProvider(files)` constructs an array of `FileProviders`, one for each file in the `files` array. Each entry sends one file to the server.

To terminate the file transfer before reaching the end of the file, set the `FileSize` property to the number of bytes desired. To decide where to end the transfer based on the file contents while it is being read, write a subclass and override `getData` to examine the data being read and set the `stop` return value to end the transfer.

`providers = FileProvider(files,permission,machineformat,encoding)` specifies options for opening the files.

`providers = FileProvider(fileIds)` constructs `FileProviders` specified by file identifiers. The files are read starting at the current file position indicator to the end of the file. The file identifiers are not closed when the transfer is complete. This method is useful if the file is already open, or when you want to transfer just the trailing part of the file. To do this, open the file, set the file position indicator to the start of the data in the file that you want to transfer, and then pass that file identifier into this constructor. You can also set `FileSize` to limit the total number of bytes or write a subclass to control when to end the transfer.

Input Arguments

`files` — File names

string | string array | character vector | cell array of character vectors

One or more file names, specified as a string, string array, character vector, or cell array of character vectors. For more information about the use of file names, see `fopen`.

Data Types: `char` | `string`

`permission` — File access type

`w+` (default) | `u+` | `u` | `T` | value allowed by `fopen` function

File access type, specified as a string. If `permission` is specified, it must allow write access. The default value is `'w+'`, which opens or creates a file for reading and writing and discards existing contents, if any.

Data Types: `char` | `string`

`machineformat` — Order for reading or writing bytes or bits

any value allowed by `fopen`

Order for reading or writing bytes or bits, specified as any value allowed by the `fopen` function.

Data Types: `char` | `string`

`encoding` — Character encoding

any value allowed by `fopen`

Character encoding, specified as any value allowed by the `fopen` function.

Data Types: `char` | `string`

`fileIds` — File identifiers

double | array of double

One or more file identifiers, specified as a double or an array of double.

Properties

Public Properties

`Filename` — Full path of file
string

Full path of the file, derived from the input argument, specified as a string.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

`FileSize` — Number of bytes to transmit
empty (default) | double

Number of bytes to transmit, specified as double.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Request` — Request message to send
`matlab.net.http.RequestMessage`

Request message to send, specified as a `matlab.net.http.RequestMessage` object.

This property is used only by subclass authors. The `RequestMessage.send` and `RequestMessage.complete` methods set this property to the `RequestMessage` in whose `Body` this provider has been placed, before calling any other methods in this provider, and before adding any additional header fields or validating the message. The provider can examine this message to see what was contained in the original request.

Delegates see the same value for this property as the delegator. `ContentProviders` should be aware that, if they are delegates, they are not necessarily providing the entire body of the request message, so they should not assume that header fields in this Request are pertinent to the data they are providing. Usually, delegates should ignore header fields in this request relevant to the data, such as Content-Type.

If the provider wishes to add any header fields to this message, or to modify existing ones, it should do so in its `complete` method by adding those fields to the `Header` property. The caller of `complete` (`RequestMessage` or a delegating provider) determines what to do with those fields. `RequestMessage.send` and `RequestMessage.complete` always copy these fields to the `Header` of the `RequestMessage`. A delegating provider can copy the fields to its own `Header` property or insert them into the message (as in the case of `MultipartProvider`). For more information, see the ContentProvider.Header property.

This property is read-only.

Attributes:

`GetAccess`	`public`
`SetAccess`	`matlab.net.http.RequestMessage`

`Header` — Header fields of message or part

`matlab.net.http.HeaderField.empty` (default) | `matlab.net.http.HeaderField`

Header fields of the message or part, specified as a vector of one or more `matlab.net.http.HeaderField` objects.

This property is only used by subclass authors. MATLAB sets this property before calling the provider's `complete` method. For non-multipart messages, MATLAB initializes this property to the contents of `Request.Header`, minus any `matlab.net.http.field.GenericFields` or empty-valued fields. The `ContentProvider` uses this property to add header fields that describe the data to be sent, or to add parameters to header fields already in the message. In a delegate for a `MultipartProvider`, MATLAB initializes this property to header fields that the delegating provider intends to insert for the part. Delegates can modify or change these fields.

Upon return from the provider's `complete` method, if this not a multipart message, then MATLAB reads this property and merges its contents into the header of `Request`. Fields in this `Header` with `Names` that do not already appear in `Request.Header` are added to the end of `Request.Header`. If a field in this `Header` has a `Name` that is the same as one in `Request.Header`, and both have nonempty `Values`, then:

If one or both of these has an empty `Value`, then the field is removed from `Request.Header` and it is not added as part of normal message completion.

If this is a delegate of a `MultipartProvider`, then the entire contents of this `Header` is used as the header of the part. Multipart delegates must not assume that `Request.Header` contains any fields pertaining to their own `Header`. A provider can determine whether it is a multipart delegate by checking whether `MyDelegator` is a `MultipartProvider`, though this test is unlikely to be needed.

MATLAB reads this property only on return from calling the provider's `complete` method. Changes to this array are ignored once MATLAB calls `start`.

Class authors should be aware that their subclasses might have added fields to this `Header` (in their `complete` method) before calling `complete` in their superclass. It is best to preserve such fields and not to add fields with the same names. However, adding a parameter to a field is permissible. For example, a superclass can add a charset parameter to an existing Content-Type field that does not already have one.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`ForceChunked` — Indicate whether to force chunked transfer coding
`false` (default) | `true`

Indicate whether to force chunked transfer coding, specified as boolean. This property is of interest only to subclass authors, and is applicable only to providers that are not multipart delegates. Subclasses set `ForceChunked` to control whether contents should be sent using chunked transfer coding. If `false` (default), MATLAB decides whether to send the contents chunked, based on whether it knows the content length at the time the message is ready to be sent:

If `ForceChunked` is `true`, then MATLAB sends the message chunked regardless of whether it knows the content length, unless the known length is smaller than the chunk size. If this property is `true`, then the message must not contain a Content-Length field, because HTTP does not allow a chunked message to have a Content-Length field. However, you can still return a nonzero value in the `expectedContentLength` method if you want MATLAB to verify that you are returning the expected length of data.

When MATLAB chooses to send the message chunked, the size of each chunk is equal to the length of data returned by `getData`.

MATLAB reads this value after calling the `complete` method, before calling `start`. It does not set this field.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

Protected Properties

`CurrentDelegate` — `ContentProvider` to which this provider is delegating

`matlab.net.http.io.ContentProvider` | empty

`ContentProvider` to which this provider is delegating, specified as a `matlab.net.http.io.ContentProvider` object. This property is set in the calling provider (the delegator) by the `delegateTo` method to indicate the current delegated provider. If there is no current delegation, then the value is empty.

The `complete` methods set this property to empty.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`MyDelegator` — `ContentProvider` that delegated to this provider

`matlab.net.http.io.ContentProvider.empty` (default) | `matlab.net.http.io.ContentProvider`

`ContentProvider` that delegated to this provider, specified as a `matlab.net.http.io.ContentProvider` object.

If a `ContentProvider` delegates responsibility for sending all or a portion of the message data to another provider, then this property identifies the delegating provider to the delegate. For example, a `MultipartProvider` delegates parts of the message to other providers, so it inserts a handle to itself in each delegate. Otherwise, `MyDelegator` is empty. The `delegateTo` method sets this property in the delegate.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

Methods

Public Methods

getData Next buffer of data to send in HTTP request message from FileProvider

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions and inherited methods for objects in this class.

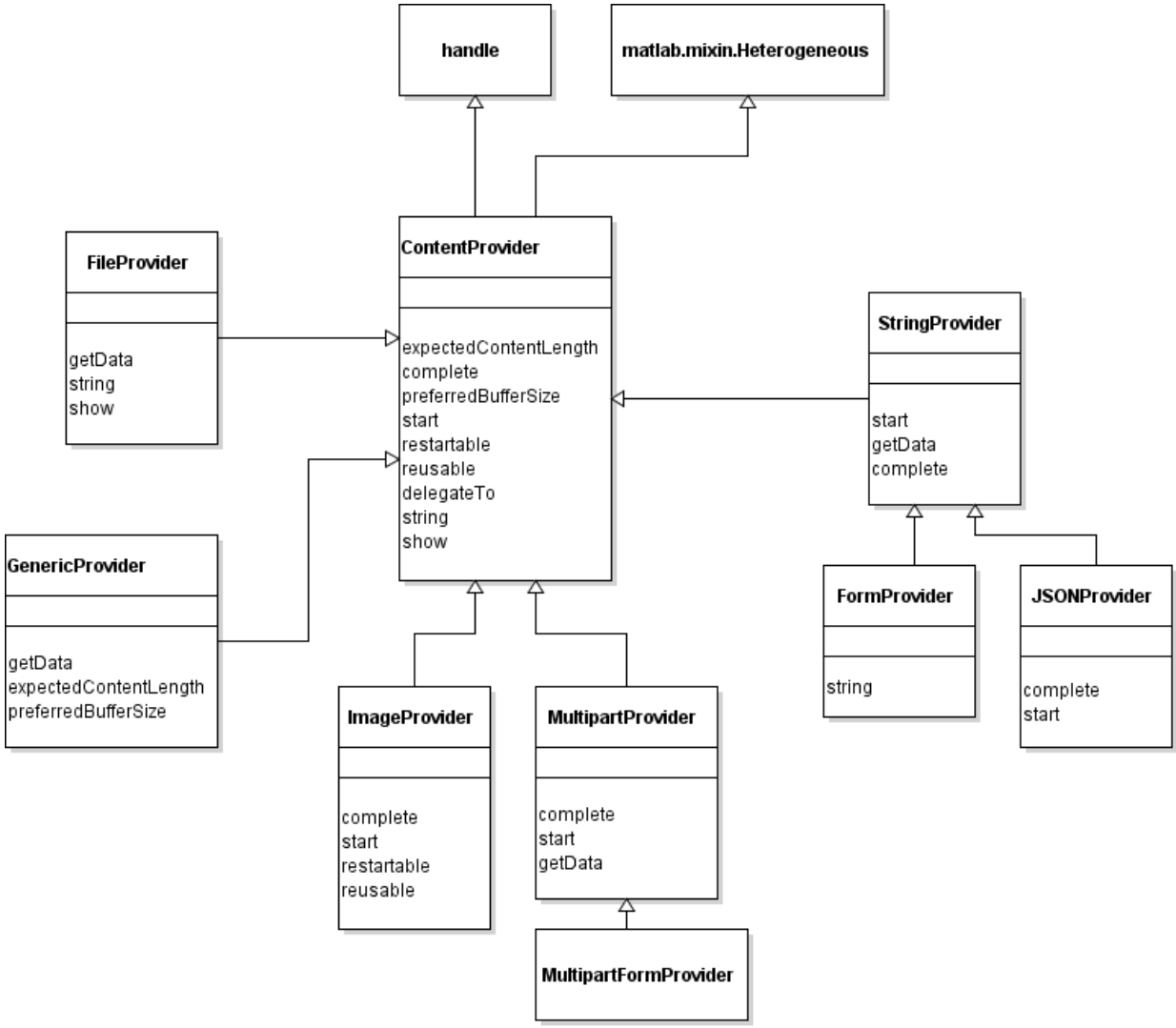
`string`	`FileProvider` information as string
`show`	Display `FileProvider` information

Protected Methods

complete	Complete HTTP header for ContentProvider
preferredBufferSize	Preferred buffer size for ContentProvider
expectedContentLength	Content length of ContentProvider
start	Start data transfer from ContentProvider
restartable	Indicate whether ContentProvider is restartable
reusable	Indicate whether ContentProvider is reusable
delegateTo	Delegate to another provider

More About

Class Hierarchy



See Also

`FileConsumer` | `MessageBody` | `MultipartFormProvider` | `MultipartProvider` | `RequestMessage` | `fopen` | `getData`

Introduced in R2018a

getData

Class: `matlab.net.http.io.FileProvider`
Package: `matlab.net.http.io`

Next buffer of data to send in HTTP request message from FileProvider

Syntax

```
[data,stop] = getData(provider,length)
```

Description

`[data,stop] = getData(provider,length)` returns the next buffer of data from the file. This method is an overridden method of `getData`.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.FileProvider`

Content provider, specified as a `matlab.net.http.io.FileProvider` object.

`length` — Length of data
double

Length of data that the provider should return, specified as double. For more information, see the `length` argument in `ContentProvider.getData`.

Output Arguments

`data` — Next buffer of data
`uint8` vector | empty

Next buffer of data, returned as a `uint8` vector or empty.

`stop` — Indicate whether to end transmission
`false` (default) | `true`

Indicate whether to end transmission, returned as a logical that the provider must set. `getData` sets `stop` to `true` if the end of file has been reached or `FileSize` bytes have been returned, whichever comes first.

Attributes

`Access`	`public`

See Also

`getData`

Introduced in R2018a

string

Class: `matlab.net.http.io.FileProvider`
Package: `matlab.net.http.io`

Show FileProvider as string

Syntax

```
str = string(provider)
```

Description

`str = string(provider)` returns information about this provider in a string. Use the `string` method for debugging. The `show` method displays the same information.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.FileProvider`

Content provider specified as a `matlab.net.http.io.FileProvider` object.

Attributes

Access	public

See Also

`show`

Introduced in R2018a

matlab.net.http.io.FormProvider class

Package: `matlab.net.http.io`
Superclasses: `matlab.net.http.io.StringProvider`

ContentProvider that sends form data

Description

This provider creates data suitable for a request message whose Content-Type is `"application/x-www-form-urlencoded"`, as required by many servers that expect users to fill in HTML forms.

Using this provider in the `Body` of a `RequestMessage` is optional, because you can insert a `QueryParameter` vector directly into the `Body` of a `RequestMessage` to get the same conversion done automatically.

Subclass authors can create a `FormProvider` subclass to create the data dynamically only when the message is ready to be transmitted, or during transmission.

The `matlab.net.http.io.FormProvider` class is a `handle` class.

Creation

Description

`provider = FormProvider(queryparams)` constructs a provider that sends a vector of `QueryParameter` objects specified as `queryparams`. This constructor sets the `Parameters` property to the value of `queryparams`.

`provider = FormProvider(arg1,...,argn)` sends an arbitrary list of arguments to the `matlab.net.QueryParameter` constructor to obtain a `QueryParameter` vector. This syntax a shortcut for:

```
FormProvider(QueryParameter(arg1,...,argn))
```

Input Arguments

`arg1,...,argn` — `QueryParameter` constructor arguments
valid types

One or more input arguments to the `matlab.net.QueryParameter` constructor, specified as valid argument types. If your arguments are already encoded, append a `'literal'` argument to the end of the list.

Properties

Public Properties

`Parameters` — Query parameters
vector or cell array of `matlab.net.QueryParameter`

Query parameters, specified as a vector or cell array of `matlab.net.QueryParameter` objects as passed into or derived from the arguments to the constructor. To see the body of the message, use the `string` method.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

`Header` — Header fields of message or part
`matlab.net.http.HeaderField.empty` (default) | `matlab.net.http.HeaderField`

Header fields of the message or part, specified as a vector of one or more `matlab.net.http.HeaderField` objects.

This property is only used by subclass authors. MATLAB sets this property before calling the provider's `complete` method. For non-multipart messages, MATLAB initializes this property to the contents of `Request.Header`, minus any `matlab.net.http.field.GenericFields` or empty-valued fields. The `ContentProvider` uses this property to add header fields that describe the data to be sent, or to add parameters to header fields already in the message. In a delegate for a `MultipartProvider`, MATLAB initializes this property to header fields that the

delegating provider intends to insert for the part. Delegates can modify or change these fields.

Upon return from the provider's `complete` method, if this not a multipart message, then MATLAB reads this property and merges its contents into the header of `Request`. Fields in this `Header` with `Names` that do not already appear in `Request.Header` are added to the end of `Request.Header`. If a field in this `Header` has a `Name` that is the same as one in `Request.Header`, and both have nonempty `Values`, then:

If one or both of these has an empty `Value`, then the field is removed from `Request.Header` and it is not added as part of normal message completion.

If this is a delegate of a `MultipartProvider`, then the entire contents of this `Header` is used as the header of the part. Multipart delegates must not assume that `Request.Header` contains any fields pertaining to their own `Header`. A provider can determine whether it is a multipart delegate by checking whether `MyDelegator` is a `MultipartProvider`, though this test is unlikely to be needed.

MATLAB reads this property only on return from calling the provider's `complete` method. Changes to this array are ignored once MATLAB calls `start`.

Class authors should be aware that their subclasses might have added fields to this `Header` (in their `complete` method) before calling `complete` in their superclass. It is best to preserve such fields and not to add fields with the same names. However, adding a parameter to a field is permissible. For example, a superclass can add a charset parameter to an existing Content-Type field that does not already have one.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`ForceChunked` — Indicate whether to force chunked transfer coding
`false` (default) | `true`

Indicate whether to force chunked transfer coding, specified as boolean. This property is of interest only to subclass authors, and is applicable only to providers that are not multipart delegates. Subclasses set `ForceChunked` to control whether contents should be sent using chunked transfer coding. If `false` (default), MATLAB decides whether to send the contents chunked, based on whether it knows the content length at the time the message is ready to be sent:

If `ForceChunked` is `true`, then MATLAB sends the message chunked regardless of whether it knows the content length, unless the known length is smaller than the chunk size. If this property is `true`, then the message must not contain a Content-Length field, because HTTP does not allow a chunked message to have a Content-Length field. However, you can still return a nonzero value in the `expectedContentLength` method if you want MATLAB to verify that you are returning the expected length of data.

When MATLAB chooses to send the message chunked, the size of each chunk is equal to the length of data returned by `getData`.

MATLAB reads this value after calling the `complete` method, before calling `start`. It does not set this field.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Request` — Request message to send
`matlab.net.http.RequestMessage`

Request message to send, specified as a `matlab.net.http.RequestMessage` object.

This property is used only by subclass authors. The `RequestMessage.send` and `RequestMessage.complete` methods set this property to the `RequestMessage` in whose `Body` this provider has been placed, before calling any other methods in this provider, and before adding any additional header fields or validating the message. The provider can examine this message to see what was contained in the original request.

Delegates see the same value for this property as the delegator. `ContentProviders` should be aware that, if they are delegates, they are not necessarily providing the entire body of the request message, so they should not assume that header fields in this Request are pertinent to the data they are providing. Usually, delegates should ignore header fields in this request relevant to the data, such as Content-Type.

If the provider wishes to add any header fields to this message, or to modify existing ones, it should do so in its `complete` method by adding those fields to the `Header` property. The caller of `complete` (`RequestMessage` or a delegating provider) determines what to do with those fields. `RequestMessage.send` and `RequestMessage.complete` always copy these fields to the `Header` of the `RequestMessage`. A delegating provider can copy the fields to its own `Header` property or insert them into the message (as in the case of `MultipartProvider`). For more information, see the ContentProvider.Header property.

This property is read-only.

Attributes:

GetAccess	public
SetAccess	matlab.net.http.RequestMessage

Protected Properties

CurrentDelegate — ContentProvider to which this provider is delegating

matlab.net.http.io.ContentProvider | empty

ContentProvider to which this provider is delegating, specified as a matlab.net.http.io.ContentProvider object. This property is set in the calling provider (the delegator) by the delegateTo method to indicate the current delegated provider. If there is no current delegation, then the value is empty.

The complete methods set this property to empty.

Attributes:

GetAccess	protected
SetAccess	protected

MyDelegator — ContentProvider that delegated to this provider

matlab.net.http.io.ContentProvider.empty (default) | matlab.net.http.io.ContentProvider

ContentProvider that delegated to this provider, specified as a matlab.net.http.io.ContentProvider object.

If a ContentProvider delegates responsibility for sending all or a portion of the message data to another provider, then this property identifies the delegating provider to the delegate. For example, a MultipartProvider delegates parts of the message to other providers, so it inserts a handle to itself in each delegate. Otherwise, MyDelegator is empty. The delegateTo method sets this property in the delegate.

Attributes:

GetAccess	protected
SetAccess	protected

Methods

Public Methods

getData Next buffer of data to send in HTTP request message from StringProvider

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions and inherited methods for objects in this class.

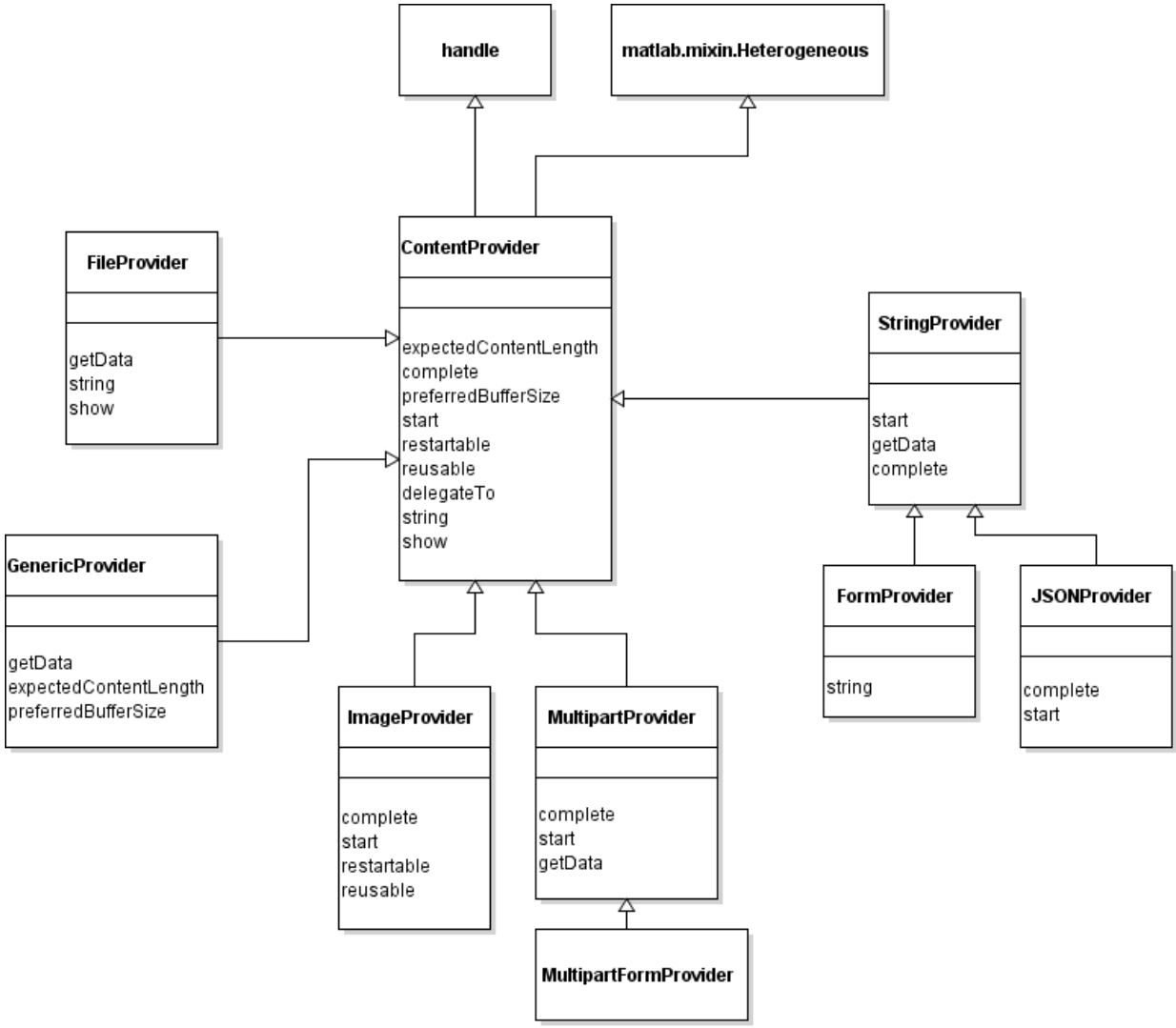
`string`	`FormProvider` information as string
`show`	Display `FormProvider` information

Protected Methods

complete	Complete HTTP header for StringProvider
preferredBufferSize	Preferred buffer size for ContentProvider
expectedContentLength	Content length of ContentProvider
start	Start data transfer from ContentProvider
restartable	Indicate whether ContentProvider is restartable
reusable	Indicate whether ContentProvider is reusable
delegateTo	Delegate to another provider

More About

Class Hierarchy



See Also

`matlab.net.QueryParameter` | `matlab.net.http.io.MultipartFormProvider`

Introduced in R2018a

string

Class: `matlab.net.http.io.FormProvider`
Package: `matlab.net.http.io`

FormProvider Data property

Syntax

```
str = string(provider)
```

Description

`str = string(provider)` returns the contents of the `Data` property as a string. This information is also displayed by the `show` method.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.FormProvider`

Content provider specified as a `matlab.net.http.io.FormProvider` object.

Attributes

Access	public

See Also

`show`

Introduced in R2018a

matlab.net.http.io.GenericProvider class

Package: `matlab.net.http.io`
Superclasses: `matlab.net.http.io.ContentProvider`

Generic ContentProvider for HTTP payloads

Description

Use a `GenericProvider` object to stream custom data generated by a function instead of creating a `ContentProvider` subclass.

The `matlab.net.http.io.GenericProvider` class is a `handle` class.

Creation

Description

`provider = GenericProvider(getDataFcn)` constructs a `ContentProvider` using function `getDataFcn` to obtain buffers of data. This constructor sets the `GetDataFcn` property to `getDataFcn`.

`provider = GenericProvider(getDataFcn,contentLength)` adds a Content-Length field to the header of the message whose value is `contentLength`. Use this syntax as an alternative to adding a Content-Length field to the header directly. In this case, the length of the data returned by `getDataFcn`, before `stop=true` is set, must exactly equal this length.

`provider = GenericProvider(getDataFcn,contentLength,bufferSize)` constructs a `ContentProvider` with additional options. If `contentLength` is not empty, then the constructor sets the length of the data. If `bufferSize` is not empty, then the constructor sets the maximum size of the data buffer.

Input Arguments

`contentLength` — Expected length of payload
uint64 | empty

Expected length of the payload, specified as `uint64`.

`bufferSize` — Buffer size
double

Buffer size, specified as double.

Properties

Public Properties

`GetDataFcn` — Function to get data
function handle

Function to get data, specified as a function handle.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true

`Header` — Header fields of message or part
`matlab.net.http.HeaderField.empty` (default) | `matlab.net.http.HeaderField`

Header fields of the message or part, specified as a vector of one or more `matlab.net.http.HeaderField` objects.

This property is only used by subclass authors. MATLAB sets this property before calling the provider's `complete` method. For non-multipart messages, MATLAB initializes this property to the contents of `Request.Header`, minus any `matlab.net.http.field.GenericFields` or empty-valued fields. The `ContentProvider` uses this property to add header fields that describe the data to be sent, or to add parameters to header fields already in the message. In a delegate for a `MultipartProvider`, MATLAB initializes this property to header fields that the

delegating provider intends to insert for the part. Delegates can modify or change these fields.

Upon return from the provider's `complete` method, if this not a multipart message, then MATLAB reads this property and merges its contents into the header of `Request`. Fields in this `Header` with `Names` that do not already appear in `Request.Header` are added to the end of `Request.Header`. If a field in this `Header` has a `Name` that is the same as one in `Request.Header`, and both have nonempty `Values`, then:

If one or both of these has an empty `Value`, then the field is removed from `Request.Header` and it is not added as part of normal message completion.

If this is a delegate of a `MultipartProvider`, then the entire contents of this `Header` is used as the header of the part. Multipart delegates must not assume that `Request.Header` contains any fields pertaining to their own `Header`. A provider can determine whether it is a multipart delegate by checking whether `MyDelegator` is a `MultipartProvider`, though this test is unlikely to be needed.

MATLAB reads this property only on return from calling the provider's `complete` method. Changes to this array are ignored once MATLAB calls `start`.

Class authors should be aware that their subclasses might have added fields to this `Header` (in their `complete` method) before calling `complete` in their superclass. It is best to preserve such fields and not to add fields with the same names. However, adding a parameter to a field is permissible. For example, a superclass can add a charset parameter to an existing Content-Type field that does not already have one.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`ForceChunked` — Indicate whether to force chunked transfer coding

`false` (default) | `true`

Indicate whether to force chunked transfer coding, specified as boolean. This property is of interest only to subclass authors, and is applicable only to providers that are not multipart delegates. Subclasses set `ForceChunked` to control whether contents should be sent using chunked transfer coding. If `false` (default), MATLAB decides whether to send the contents chunked, based on whether it knows the content length at the time the message is ready to be sent:

If `ForceChunked` is `true`, then MATLAB sends the message chunked regardless of whether it knows the content length, unless the known length is smaller than the chunk size. If this property is `true`, then the message must not contain a Content-Length field, because HTTP does not allow a chunked message to have a Content-Length field. However, you can still return a nonzero value in the `expectedContentLength` method if you want MATLAB to verify that you are returning the expected length of data.

When MATLAB chooses to send the message chunked, the size of each chunk is equal to the length of data returned by `getData`.

MATLAB reads this value after calling the `complete` method, before calling `start`. It does not set this field.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Request` — Request message to send
`matlab.net.http.RequestMessage`

Request message to send, specified as a `matlab.net.http.RequestMessage` object.

This property is used only by subclass authors. The `RequestMessage.send` and `RequestMessage.complete` methods set this property to the `RequestMessage` in whose `Body` this provider has been placed, before calling any other methods in this provider, and before adding any additional header fields or validating the message. The provider can examine this message to see what was contained in the original request.

Delegates see the same value for this property as the delegator. `ContentProviders` should be aware that, if they are delegates, they are not necessarily providing the entire body of the request message, so they should not assume that header fields in this Request are pertinent to the data they are providing. Usually, delegates should ignore header fields in this request relevant to the data, such as Content-Type.

If the provider wishes to add any header fields to this message, or to modify existing ones, it should do so in its `complete` method by adding those fields to the `Header` property. The caller of `complete` (`RequestMessage` or a delegating provider) determines what to do with those fields. `RequestMessage.send` and `RequestMessage.complete` always copy these fields to the `Header` of the `RequestMessage`. A delegating provider can copy the fields to its own `Header` property or insert them into the message (as in the case of `MultipartProvider`). For more information, see the ContentProvider.Header property.

This property is read-only.

Attributes:

GetAccess	public
SetAccess	matlab.net.http.RequestMessage

Protected Properties

CurrentDelegate — ContentProvider to which this provider is delegating

matlab.net.http.io.ContentProvider | empty

ContentProvider to which this provider is delegating, specified as a matlab.net.http.io.ContentProvider object. This property is set in the calling provider (the delegator) by the delegateTo method to indicate the current delegated provider. If there is no current delegation, then the value is empty.

The complete methods set this property to empty.

Attributes:

GetAccess	protected
SetAccess	protected

MyDelegator — ContentProvider that delegated to this provider

matlab.net.http.io.ContentProvider.empty (default) | matlab.net.http.io.ContentProvider

ContentProvider that delegated to this provider, specified as a matlab.net.http.io.ContentProvider object.

If a ContentProvider delegates responsibility for sending all or a portion of the message data to another provider, then this property identifies the delegating provider to the delegate. For example, a MultipartProvider delegates parts of the message to other providers, so it inserts a handle to itself in each delegate. Otherwise, MyDelegator is empty. The delegateTo method sets this property in the delegate.

Attributes:

GetAccess	protected
SetAccess	protected

Methods

Public Methods

getData Next buffer of data to send in HTTP request message from GenericProvider

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions and inherited methods for objects in this class.

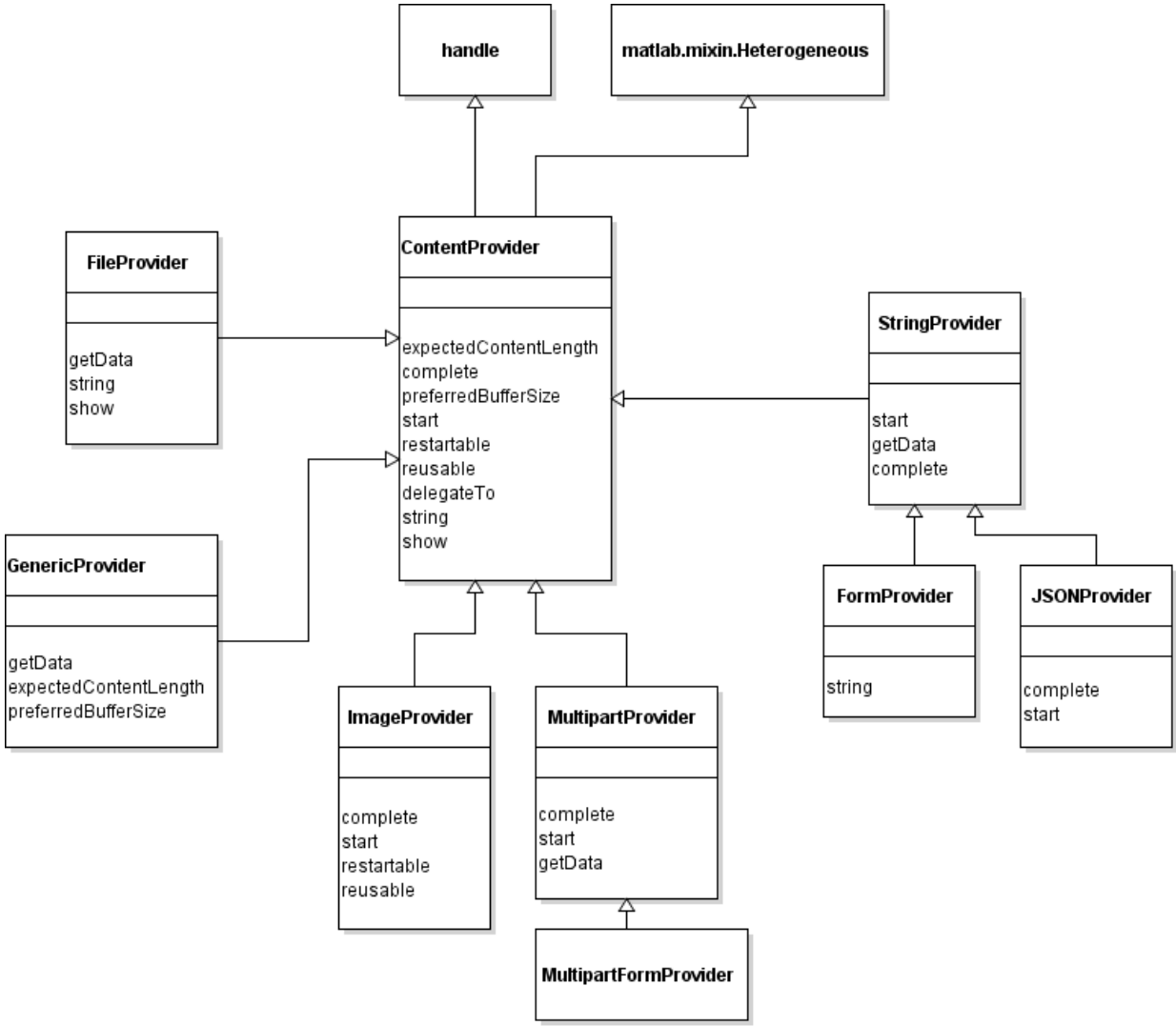
`string`	`GenericProvider` information as string
`show`	Display `GenericProvider` information

Protected Methods

complete	Complete HTTP header for ContentProvider
preferredBufferSize	Preferred buffer size for ContentProvider
expectedContentLength	Content length of ContentProvider
start	Start data transfer from ContentProvider
restartable	Indicate whether ContentProvider is restartable
reusable	Indicate whether ContentProvider is reusable
delegateTo	Delegate to another provider

More About

Class Hierarchy



See Also

`GenericConsumer` | `MessageBody` | `RequestMessage`

Introduced in R2018a

expectedContentLength

Class: `matlab.net.http.io.GenericProvider`
Package: `matlab.net.http.io`

Content length of GenericProvider

Syntax

```
length = expectedContentLength(provider,force)
```

Description

`length = expectedContentLength(provider,force)` returns the value of the `contentLength` argument provided to the constructor. If none was provided, then `expectedContentLength` returns `[]`, which results in chunked transfer coding if the message has no Content-Length field. This method is an overridden method of `expectedContentLength`.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.GenericProvider`

Content provider, specified as a `matlab.net.http.io.GenericProvider` object.

`force` — Indicate whether to return length of data
`false` (default) | `true`

Indicate whether to return the length of the data, specified as `true` or `false`. For more information, see the `force` argument in `ContentProvider.expectedContentLength`.

Output Arguments

`length` — Expected content length
`[]` (default) | bytes

Expected content length, returned as bytes. If you do not override this method, then `expectedContentLength` returns `[]`. For more information, see the length argument in `ContentProvider.expectedContentLength`.

Attributes

Access	protected

See Also

`expectedContentLength`

Introduced in R2018a

getData

Class: `matlab.net.http.io.GenericProvider`
Package: `matlab.net.http.io`

Next buffer of data to send in HTTP request message from GenericProvider

Syntax

```
[data,stop] = getData(provider,length)
```

Description

`[data,stop] = getData(provider,length)` calls the `getDataFcn` function, specified by the argument to the `GenericProvider` constructor. `getData` is an overridden method of `getData`.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.GenericProvider`

Content provider, specified as a `matlab.net.http.io.GenericProvider` object.

`length` — Length of data
double

Length of data that the provider should return, specified as double. For more information, see the `length` argument in `ContentProvider.getData`.

Output Arguments

`data` — Next buffer of data
`uint8` vector | empty

Next buffer of data, returned as a `uint8` vector or empty.

`stop` — Indicate whether to end transmission
`false` (default) | `true`

Indicate whether to end transmission, returned as a logical that the provider must set.

Attributes

Access	public

See Also

`getData`

Introduced in R2018a

preferredBufferSize

Class: `matlab.net.http.io.GenericProvider`
Package: `matlab.net.http.io`

Preferred buffer size for GenericProvider

Syntax

```
size = preferredBufferSize(provider)
```

Description

`size = preferredBufferSize(provider)` returns the value of the `bufferSize` argument provided to the constructor. If none was provided, then `preferredBufferSize` returns `[]`, which uses a default buffer size. This method is an overridden method of `preferredBufferSize`.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.GenericProvider`

Content provider, specified as a `matlab.net.http.io.GenericProvider` object.

Output Arguments

`size` — Preferred buffer size
`[]` (default) | bytes

Preferred buffer size, returned as bytes. For more information, see the `size` argument in `ContentProvider.preferredBufferSize`.

Attributes

Access	protected

See Also

preferredBufferSize

Introduced in R2018a

restartable

Class: `matlab.net.http.io.GenericProvider`
Package: `matlab.net.http.io`

Indicate whether GenericProvider is restartable

Syntax

```
TF = restartable(provider)
```

Description

`TF = restartable(provider)` returns `true` if the `GetDataFcn` argument specified to the `GenericProvider` constructor takes more than one input argument. Otherwise, the method returns `false`.This method is an overridden method of `restartable`.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.GenericProvider`

Content provider, specified as a `matlab.net.http.io.GenericProvider` object.

Attributes

`Access`	`protected`

See Also

`restartable` | `reusable`

Introduced in R2018a

reusable

Class: `matlab.net.http.io.GenericProvider`
Package: `matlab.net.http.io`

Indicate whether GenericProvider is reusable

Syntax

```
TF = reusable(provider)
```

Description

`TF = reusable(provider)` returns `true` if the `GetDataFcn` argument specified to the `GenericProvider` constructor takes more than one input argument. Otherwise, the method returns `false`. This method is an overridden method of `reusable`.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.GenericProvider`

Content provider, specified as a `matlab.net.http.io.GenericProvider` object.

Attributes

Access	protected

See Also

`restartable` | `reusable`

Introduced in R2018a

matlab.net.http.io.ImageProvider class

Package: `matlab.net.http.io`
Superclasses: `matlab.net.http.io.ContentProvider`

ContentProvider to send MATLAB image data

Description

Use an `ImageProvider` object to convert and send MATLAB image data in an HTTP `RequestMessage`. This provider converts the data to one of the standard types, as specified by the Content-Type of the request or properties in this object. This provider also converts an image file to a different format.

By default, if you specify an image Content-Type in the `RequestMessage`, for example `"image/jpeg"` and `RequestMessage.Body` is a `MessageBody` containing your image data, then MATLAB assumes that `MessageBody.Data` is image data and tries to convert it appropriately. To control how your data is converted, use an `ImageProvider` in `RequestMessage.Body`.

If the `RequestMessage` does not contain a Content-Type header field, then this provider adds the appropriate image Content-Type to the header. Otherwise, the header field is not modified and conversion is done as specified in the properties of this object, even if its value is inconsistent with the Content-Type field.

The `matlab.net.http.io.ImageProvider` class is a `handle` class.

Creation

Description

`provider = ImageProvider(data,arg1,...,argN)` creates a provider that converts MATLAB image `data` to the format specified by `arg1,...,argN`, setting the Data property to `data` and the Arguments property to `arg1,...,argN`.

`provider = ImageProvider(filename,arg1,...,argN)` obtains the image data from the file `filename` and sends it in the format specified in `arg1,...,argN` or the Content-Type field, setting the Filename property to `filename`.

While this provider can convert a file in one format to data in another format, it is not designed to send an image file as is. To send a file without changing its type, use a `FileProvider`.

Properties

Public Properties

`Data` — MATLAB data to be converted

MATLAB type

This is the value specified as the Data argument to the constructor, if any, or the data converted from the `filename` argument. Set this value to a string scalar or character vector before sending the message containing this provider.

If a subclass author wants to set this property before MATLAB calls the `start` method, then set the value in the `complete` method. If you change this property, then the change takes effect with the next call to `start`.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Arguments` — Arguments to `imwrite` function

cell array

Arguments to the `imwrite` function, specified as a cell array. This is the value specified to the constructor containing a list of arguments as documented for `imwrite`, but omitting the image data and file name arguments.

Subclass authors can set this property before MATLAB calls `start`, for example, in `complete`. If you change these values, then the changes do not take effect until the next call to `start`.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

`Filename` — File containing image data
string

Name of the file containing the image data, specified as a string. This is the value specified as the `filename` argument to the constructor, if any. It is read-only.

The format of `filename` is derived from the file name extension, which might be different from the `fmt` argument in `arg1,...,argN` or the Content-Type in the message. This constructor syntax allows you to send an image file in one format to a server that expects it in a different format. However, some conversions are incompatible. For example, you cannot convert an RGB file like JPEG to a GIF format.

Attributes:

```
GetAccess                                 public
SetAccess                                 private
```

`Header` — Header fields of message or part
`matlab.net.http.HeaderField.empty` (default) | `matlab.net.http.HeaderField`

Header fields of the message or part, specified as a vector of one or more `matlab.net.http.HeaderField` objects.

This property is only used by subclass authors. MATLAB sets this property before calling the provider's `complete` method. For non-multipart messages, MATLAB initializes this property to the contents of `Request.Header`, minus any `matlab.net.http.field.GenericFields` or empty-valued fields. The `ContentProvider` uses this property to add header fields that describe the data to be sent, or to add parameters to header fields already in the message. In a delegate for a `MultipartProvider`, MATLAB initializes this property to header fields that the delegating provider intends to insert for the part. Delegates can modify or change these fields.

Upon return from the provider's `complete` method, if this not a multipart message, then MATLAB reads this property and merges its contents into the header of `Request`. Fields in this `Header` with `Names` that do not already appear in `Request.Header` are added to

the end of `Request.Header`. If a field in this `Header` has a `Name` that is the same as one in `Request.Header`, and both have nonempty `Values`, then:

If one or both of these has an empty `Value`, then the field is removed from `Request.Header` and it is not added as part of normal message completion.

If this is a delegate of a `MultipartProvider`, then the entire contents of this `Header` is used as the header of the part. Multipart delegates must not assume that `Request.Header` contains any fields pertaining to their own `Header`. A provider can determine whether it is a multipart delegate by checking whether `MyDelegator` is a `MultipartProvider`, though this test is unlikely to be needed.

MATLAB reads this property only on return from calling the provider's `complete` method. Changes to this array are ignored once MATLAB calls `start`.

Class authors should be aware that their subclasses might have added fields to this `Header` (in their `complete` method) before calling `complete` in their superclass. It is best to preserve such fields and not to add fields with the same names. However, adding a parameter to a field is permissible. For example, a superclass can add a charset parameter to an existing Content-Type field that does not already have one.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`ForceChunked` — Indicate whether to force chunked transfer coding

`false` (default) | `true`

Indicate whether to force chunked transfer coding, specified as boolean. This property is of interest only to subclass authors, and is applicable only to providers that are not multipart delegates. Subclasses set `ForceChunked` to control whether contents should be sent using chunked transfer coding. If `false` (default), MATLAB decides whether to send the contents chunked, based on whether it knows the content length at the time the message is ready to be sent:

If `ForceChunked` is `true`, then MATLAB sends the message chunked regardless of whether it knows the content length, unless the known length is smaller than the chunk size. If this property is `true`, then the message must not contain a Content-Length field, because HTTP does not allow a chunked message to have a Content-Length field.

However, you can still return a nonzero value in the `expectedContentLength` method if you want MATLAB to verify that you are returning the expected length of data.

When MATLAB chooses to send the message chunked, the size of each chunk is equal to the length of data returned by `getData`.

MATLAB reads this value after calling the `complete` method, before calling `start`. It does not set this field.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Request` — Request message to send
`matlab.net.http.RequestMessage`

Request message to send, specified as a `matlab.net.http.RequestMessage` object.

This property is used only by subclass authors. The `RequestMessage.send` and `RequestMessage.complete` methods set this property to the `RequestMessage` in whose `Body` this provider has been placed, before calling any other methods in this provider, and before adding any additional header fields or validating the message. The provider can examine this message to see what was contained in the original request.

Delegates see the same value for this property as the delegator. `ContentProviders` should be aware that, if they are delegates, they are not necessarily providing the entire body of the request message, so they should not assume that header fields in this Request are pertinent to the data they are providing. Usually, delegates should ignore header fields in this request relevant to the data, such as Content-Type.

If the provider wishes to add any header fields to this message, or to modify existing ones, it should do so in its `complete` method by adding those fields to the `Header` property. The caller of `complete` (`RequestMessage` or a delegating provider) determines what to do with those fields. `RequestMessage.send` and `RequestMessage.complete` always copy these fields to the `Header` of the `RequestMessage`. A delegating provider can copy the fields to its own `Header` property or insert them into the message (as in the case of `MultipartProvider`). For more information, see the ContentProvider.Header property.

This property is read-only.

Attributes:

GetAccess	public
SetAccess	matlab.net.http.RequestMessage

Protected Properties

CurrentDelegate — ContentProvider to which this provider is delegating
matlab.net.http.io.ContentProvider | empty

ContentProvider to which this provider is delegating, specified as a matlab.net.http.io.ContentProvider object. This property is set in the calling provider (the delegator) by the delegateTo method to indicate the current delegated provider. If there is no current delegation, then the value is empty.

The complete methods set this property to empty.

Attributes:

GetAccess	protected
SetAccess	protected

MyDelegator — ContentProvider that delegated to this provider
matlab.net.http.io.ContentProvider.empty (default) | matlab.net.http.io.ContentProvider

ContentProvider that delegated to this provider, specified as a matlab.net.http.io.ContentProvider object.

If a ContentProvider delegates responsibility for sending all or a portion of the message data to another provider, then this property identifies the delegating provider to the delegate. For example, a MultipartProvider delegates parts of the message to other providers, so it inserts a handle to itself in each delegate. Otherwise, MyDelegator is empty. The delegateTo method sets this property in the delegate.

Attributes:

GetAccess	protected
SetAccess	protected

Methods

Public Methods

getData Next buffer of data to send in HTTP request message from ContentProvider

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions and inherited methods for objects in this class.

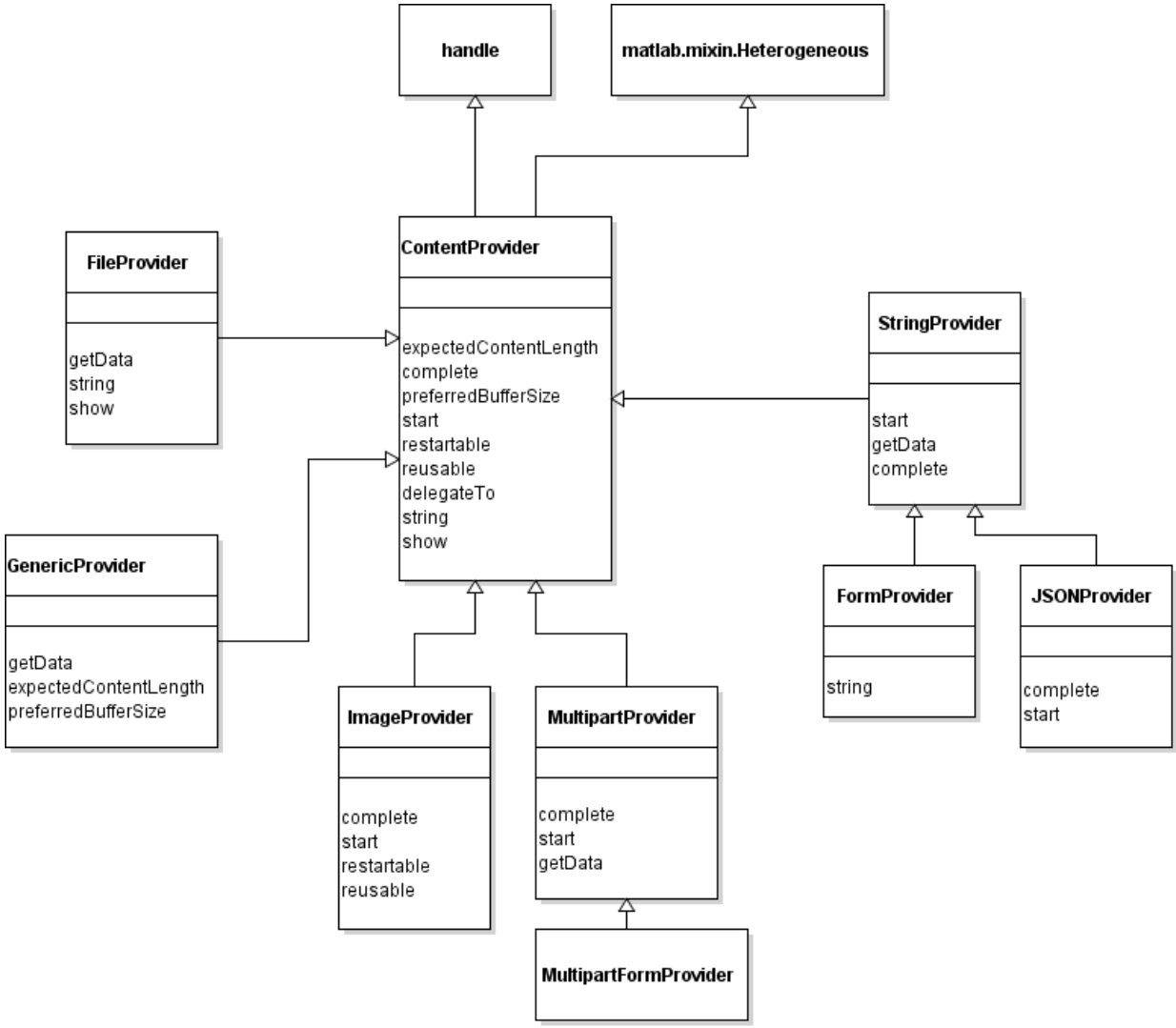
`string`	`ImageProvider` information as string
`show`	Display `ImageProvider` information

Protected Methods

complete	Complete HTTP header for ImageProvider
preferredBufferSize	Preferred buffer size for ContentProvider
expectedContentLength	Content length of ContentProvider
start	Start data transfer from ImageProvider
restartable	Indicate whether ImageProvider is restartable
reusable	Indicate whether ImageProvider is reusable
delegateTo	Delegate to another provider

More About

Class Hierarchy



See Also

`ContentProvider` | `FileProvider` | `ImageConsumer` | `MessageBody` | `RequestMessage` | `imwrite`

Introduced in R2018a

complete

Class: `matlab.net.http.io.ImageProvider`
Package: `matlab.net.http.io`

Complete HTTP header for ImageProvider

Syntax

```
complete(provider,URI)
```

Description

`complete(provider,URI)` completes the header of the message, or, in the case of a multipart message, the part for which this provider is being used. If there is no Content-Type field, then `complete` adds one specifying the `MediaType` derived from the arguments to the constructor. If there is a Content-Type field, then the method sets the `MediaType`. If the `MediaType` cannot be determined from either the Content-Type or the arguments, then the method tries to derive it from the suffix of the file name in the Content-Disposition field. If that algorithm does not work, then `MediaType` is set to `"image/jpeg"`.

This method is an overridden method of `complete`.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.ImageProvider`

Content provider, specified as a `matlab.net.http.io.ImageProvider` object.

`URI` — Uniform resource identifier
`matlab.net.URI`

Uniform resource identifier, specified as a `matlab.net.URI` object.

Attributes

Access	protected

See Also

complete | matlab.net.URI | matlab.net.http.MediaType

Introduced in R2018a

restartable

Class: `matlab.net.http.io.ImageProvider`
Package: `matlab.net.http.io`

Indicate whether ImageProvider is restartable

Syntax

```
TF = restartable(provider)
```

Description

`TF = restartable(provider)` indicates whether this `ImageProvider` is restartable. This method is an overridden method of `restartable`, and it always returns `true`.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.ImageProvider`

Content provider, specified as a `matlab.net.http.io.ImageProvider` object.

Attributes

Access	protected

See Also

`restartable` | `reusable`

Introduced in R2018a

reusable

Class: `matlab.net.http.io.ImageProvider`
Package: `matlab.net.http.io`

Indicate whether ImageProvider is reusable

Syntax

```
TF = reusable(provider)
```

Description

`TF = reusable(provider)` indicates whether this `ImageProvider` is reusable. This method is an overridden method of `reusable`, and it always returns `true`.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.ImageProvider`

Content provider, specified as a `matlab.net.http.io.ImageProvider` object.

Attributes

Access	protected

See Also

`restartable` | `reusable`

Introduced in R2018a

start

Class: `matlab.net.http.io.ImageProvider`
Package: `matlab.net.http.io`

Start data transfer from ImageProvider

Syntax

```
start(provider)
```

Description

`start(provider)` is called by MATLAB to prepare this provider for a new transfer. This method is an overridden method of `start`

Input Arguments

`provider` — Content provider
`matlab.net.http.io.ImageProvider`

Content provider, specified as a `matlab.net.http.io.ImageProvider` object.

Attributes

Access	protected

See Also

`start`

Introduced in R2018a

matlab.net.http.io.JSONProvider class

Package: `matlab.net.http.io`
Superclasses: `matlab.net.http.io.StringProvider`

ContentProvider to send MATLAB data as JSON string

Description

Use a `JSONProvider` object to convert MATLAB data to a JSON string and send it in a `RequestMessage`. Conversion is done using `jsonencode`.

If the `RequestMessage` contains no Content-Type header field, then this provider adds one specifying `"application/json"`. Otherwise, the header field is not modified and conversion is done even if its value is inconsistent with JSON data.

For nonmultipart messages, you do not usually specify this provider explicitly, as the contents of `MessageBody.Data` is automatically converted to JSON if the Content-Type of the message is `"application/json"`. For more information, see `MessageBody.Data`. Specify this provider explicitly to send JSON data for other Content-Types, or to send JSON data as a part in a multipart message.

The `matlab.net.http.io.JSONProvider` class is a `handle` class.

Creation

Description

`provider = JSONProvider(data)` constructs a `ContentProvider` with the `JSONData` property set to `data`.

Properties

Public Properties

`JSONData` — MATLAB data to convert
MATLAB data

MATLAB data to convert. The value, if any, is specified in the constructor.

Subclass authors can set this value any time before MATLAB calls the `start` method. If you change the value, it does not take effect until the next call to `start`.

Attributes:

GetAccess	public
SetAccess	public
Dependent	true

`Header` — Header fields of message or part
`matlab.net.http.HeaderField.empty` (default) | `matlab.net.http.HeaderField`

Header fields of the message or part, specified as a vector of one or more `matlab.net.http.HeaderField` objects.

This property is only used by subclass authors. MATLAB sets this property before calling the provider's `complete` method. For non-multipart messages, MATLAB initializes this property to the contents of `Request.Header`, minus any `matlab.net.http.field.GenericFields` or empty-valued fields. The `ContentProvider` uses this property to add header fields that describe the data to be sent, or to add parameters to header fields already in the message. In a delegate for a `MultipartProvider`, MATLAB initializes this property to header fields that the delegating provider intends to insert for the part. Delegates can modify or change these fields.

Upon return from the provider's `complete` method, if this not a multipart message, then MATLAB reads this property and merges its contents into the header of `Request`. Fields in this `Header` with `Names` that do not already appear in `Request.Header` are added to the end of `Request.Header`. If a field in this `Header` has a `Name` that is the same as one in `Request.Header`, and both have nonempty `Values`, then:

If one or both of these has an empty `Value`, then the field is removed from `Request.Header` and it is not added as part of normal message completion.

If this is a delegate of a `MultipartProvider`, then the entire contents of this `Header` is used as the header of the part. Multipart delegates must not assume that `Request.Header` contains any fields pertaining to their own `Header`. A provider can determine whether it is a multipart delegate by checking whether `MyDelegator` is a `MultipartProvider`, though this test is unlikely to be needed.

MATLAB reads this property only on return from calling the provider's `complete` method. Changes to this array are ignored once MATLAB calls `start`.

Class authors should be aware that their subclasses might have added fields to this `Header` (in their `complete` method) before calling `complete` in their superclass. It is best to preserve such fields and not to add fields with the same names. However, adding a parameter to a field is permissible. For example, a superclass can add a charset parameter to an existing Content-Type field that does not already have one.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`ForceChunked` — Indicate whether to force chunked transfer coding

`false` (default) | `true`

Indicate whether to force chunked transfer coding, specified as boolean. This property is of interest only to subclass authors, and is applicable only to providers that are not multipart delegates. Subclasses set `ForceChunked` to control whether contents should be sent using chunked transfer coding. If `false` (default), MATLAB decides whether to send the contents chunked, based on whether it knows the content length at the time the message is ready to be sent:

If `ForceChunked` is `true`, then MATLAB sends the message chunked regardless of whether it knows the content length, unless the known length is smaller than the chunk size. If this property is `true`, then the message must not contain a Content-Length field, because HTTP does not allow a chunked message to have a Content-Length field. However, you can still return a nonzero value in the `expectedContentLength` method if you want MATLAB to verify that you are returning the expected length of data.

When MATLAB chooses to send the message chunked, the size of each chunk is equal to the length of data returned by `getData`.

MATLAB reads this value after calling the `complete` method, before calling `start`. It does not set this field.

Attributes:

GetAccess	public
SetAccess	public

`Request` — Request message to send
`matlab.net.http.RequestMessage`

Request message to send, specified as a `matlab.net.http.RequestMessage` object.

This property is used only by subclass authors. The `RequestMessage.send` and `RequestMessage.complete` methods set this property to the `RequestMessage` in whose `Body` this provider has been placed, before calling any other methods in this provider, and before adding any additional header fields or validating the message. The provider can examine this message to see what was contained in the original request.

Delegates see the same value for this property as the delegator. `ContentProviders` should be aware that, if they are delegates, they are not necessarily providing the entire body of the request message, so they should not assume that header fields in this Request are pertinent to the data they are providing. Usually, delegates should ignore header fields in this request relevant to the data, such as Content-Type.

If the provider wishes to add any header fields to this message, or to modify existing ones, it should do so in its `complete` method by adding those fields to the `Header` property. The caller of `complete` (`RequestMessage` or a delegating provider) determines what to do with those fields. `RequestMessage.send` and `RequestMessage.complete` always copy these fields to the `Header` of the `RequestMessage`. A delegating provider can copy the fields to its own `Header` property or insert them into the message (as in the case of `MultipartProvider`). For more information, see the ContentProvider.Header property.

This property is read-only.

Attributes:

GetAccess	public
SetAccess	matlab.net.http.RequestMessage

Protected Properties

`CurrentDelegate` — `ContentProvider` to which this provider is delegating
`matlab.net.http.io.ContentProvider` | empty

`ContentProvider` to which this provider is delegating, specified as a `matlab.net.http.io.ContentProvider` object. This property is set in the calling provider (the delegator) by the `delegateTo` method to indicate the current delegated provider. If there is no current delegation, then the value is empty.

The `complete` methods set this property to empty.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`MyDelegator` — `ContentProvider` that delegated to this provider
`matlab.net.http.io.ContentProvider.empty` (default) | `matlab.net.http.io.ContentProvider`

`ContentProvider` that delegated to this provider, specified as a `matlab.net.http.io.ContentProvider` object.

If a `ContentProvider` delegates responsibility for sending all or a portion of the message data to another provider, then this property identifies the delegating provider to the delegate. For example, a `MultipartProvider` delegates parts of the message to other providers, so it inserts a handle to itself in each delegate. Otherwise, `MyDelegator` is empty. The `delegateTo` method sets this property in the delegate.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

Methods

Public Methods

getData Next buffer of data to send in HTTP request message from StringProvider

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions and inherited methods for objects in this class.

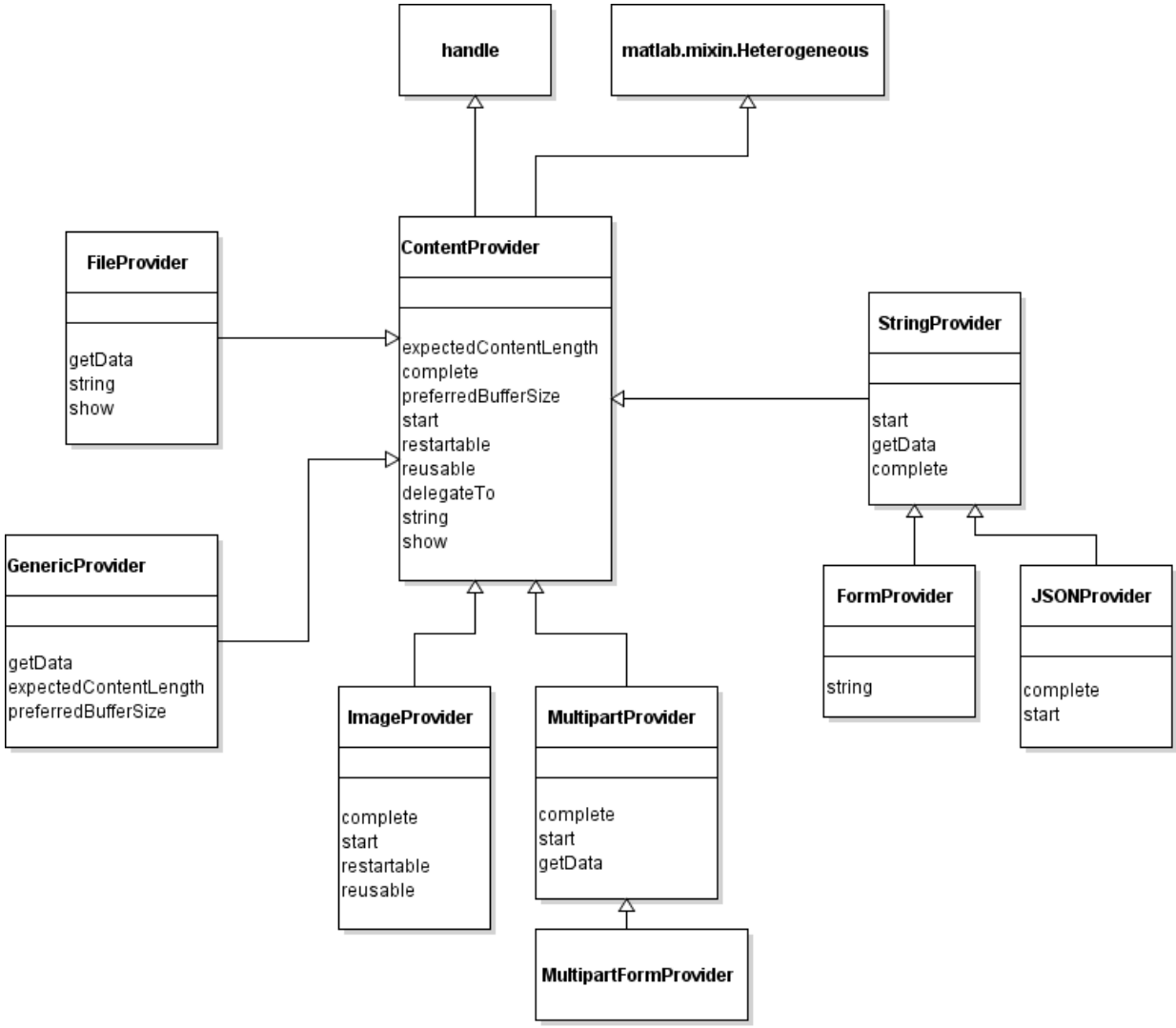
`string`	Provider information as string
`show`	Display provider information

Protected Methods

complete	Complete HTTP header for JSONProvider
preferredBufferSize	Preferred buffer size for ContentProvider
expectedContentLength	Content length of ContentProvider
start	Start data transfer from JSONProvider
restartable	Indicate whether ImageProvider is restartable
reusable	Indicate whether ImageProvider is reusable
delegateTo	Delegate to another provider

More About

Class Hierarchy



See Also

`ContentProvider` | `JSONConsumer` | `MessageBody` | `RequestMessage` | `StringProvider` | `jsonencode`

Introduced in R2018a

complete

Class: `matlab.net.http.io.JSONProvider`
Package: `matlab.net.http.io`

Complete HTTP header for JSONProvider

Syntax

```
complete(provider,URI)
```

Description

`complete(provider,URI)` completes the header of the message, or, in the case of a multipart message, the part for which this provider is being used. If there is no Content-Type field, then `complete` adds one specifying `"application/json"`. If there is already a Content-Type field that does not contain a charset parameter, and this object's `Charset` property is different from the default for that Content-Type, then a charset parameter is added to the header field.

This method is an overridden method of `complete`.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.JSONProvider`

Content provider, specified as a `matlab.net.http.io.JSONProvider` object.

`URI` — Uniform resource identifier
`matlab.net.URI`

Uniform resource identifier, specified as a `matlab.net.URI` object.

Attributes

Access	protected

See Also

complete | matlab.net.URI

Introduced in R2018a

start

Class: `matlab.net.http.io.JSONProvider`
Package: `matlab.net.http.io`

Start data transfer from JSONProvider

Syntax

```
start(provider)
```

Description

`start(provider)` prepares this `JSONProvider` for a new HTTP transfer. MATLAB calls `start`. It is an overridden method of `start`.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.JSONProvider`

Content provider, specified as a `matlab.net.http.io.JSONProvider` object.

Attributes

Access	protected

See Also

`start`

Introduced in R2018a

matlab.net.http.io.MultipartFormProvider class

Package: `matlab.net.http.io`
Superclasses: `matlab.net.http.io.MultipartProvider`

ContentProvider to send multipart/form-data messages

Description

Use this provider to send a multipart form to the server. A multipart form is a message containing a series of parts, where each part has a "control name" and its data. The data can be any of the types allowed for `RequestMessage.Body.Data` or another `ContentProvider`.

Some servers require multiple parts under the same name to be in a nested multipart/mixed part. To send nested parts, wrap the parts in a `MultipartProvider`. For example, to send a message as described at the very end of chapter 17 of the HTML 4.01 specification for form data:

```
fps = FileProvider(["file1.txt","file2.gif"]); % get array of providers
mp = MultipartProvider(fps);
formProvider = MultipartFormProvider("submit-name","Larry","files",mp);
req = RequestMessage('put',[],formProvider);
req.send(uri);
```

The `matlab.net.http.io.MultipartFormProvider` class is a `handle` class.

Creation

Description

`provider = MultipartFormProvider(Name,Part)` creates `"multipart/form-data"` content specified by one or more name-part pair arguments. `Part` is form-data containing a `Name` and its contents. The `Part` arguments can be any of the types supported by `MultipartProvider`, including other `ContentProvider` objects.

If a `Part` is an array, it is equivalent to repeating the `Name,Part` for each element of the array. For example, the statement:

```
MultipartFormProvider("name",FileProvider(["file1" "file2"]))
```

is equivalent to:

```
MultipartFormProvider("name",FileProvider("file1"),"name",FileProvider("file2"));
```

Properties

Public Properties

`Names` — Part names

string

Part names, specified as a string.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Parts` — Message body parts

cell array

Parts of the message body, specified as a cell array of one or more of these values:

`ContentProvider` object	The `MultipartProvider` delegates creation of the part to the specified provider (called the delegate), invoking its complete method to obtain header information about the part and its `getData` method to obtain the data. The delegate's `Header` property is used for the header of the part. Any subclass of `ContentProvider` can be specified here. Normally, the delegate does not specify the content length nor implement the `expectedContentLength` method, since the end of a part is designated by a boundary string rather than a header field. If that method is implemented to return a nonempty value, then the value is used only to enforce the length of the content, not to create a Content-Length field.
`RequestMessage` object	The `MultipartProvider` sends the `Header` and `Body` of the `RequestMessage` as the part. If the Body's `Payload` property is set, then that is used for the raw payload. Otherwise the Body's `Data` property is converted based on its type or the Content-Type field in the `Header`, as described for `MessageBody.Data`. This option is useful if you have data to send and want to take advantage of the default processing of that data that MATLAB normally does when sending a `RequestMessage`. It allows you to specify custom header fields in the request to be used as the part's `Header` and control how the data is converted, without having to write a `ContentProvider` subclass. The `RequestMessage.RequestLine` property is ignored.

`MessageBody` object	The `MessageBody` is processed the same as if it was in a `RequestMessage` that had no Content-Type field. This option is useful if default processing of the data based on its type is sufficient, and you do not need to specify any custom header fields for the part. MATLAB inserts a Content-Type field in the part based on the type of the data. See `MessageBody.Data` for conversion rules.
Array of `ContentProvider`, `RequestMessage`, and/or `MessageBody` objects	This treats each element of the array as a part. Not a cell array.
Handle to `getData` method	This method must have the signature of `ContentProvider.getData`. In this case, the part's Content-Type is set to `"application/octet-stream"`, so this option is useful for sending binary data. When using this option, you cannot specify any custom header fields for the part.
Any other type	If the type does not match any of these types and is not a function handle, then it is treated as if it was present in the `Data` property of a `MessageBody`. See the description for `MessageBody` types.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`
`Dependent`	`true`

`Header` — Header fields of message or part

`matlab.net.http.HeaderField.empty` (default) | `matlab.net.http.HeaderField`

Header fields of the message or part, specified as a vector of one or more `matlab.net.http.HeaderField` objects.

This property is only used by subclass authors. MATLAB sets this property before calling the provider's `complete` method. For non-multipart messages, MATLAB initializes this

property to the contents of `Request.Header`, minus any `matlab.net.http.field.GenericFields` or empty-valued fields. The `ContentProvider` uses this property to add header fields that describe the data to be sent, or to add parameters to header fields already in the message. In a delegate for a `MultipartProvider`, MATLAB initializes this property to header fields that the delegating provider intends to insert for the part. Delegates can modify or change these fields.

Upon return from the provider's `complete` method, if this not a multipart message, then MATLAB reads this property and merges its contents into the header of `Request`. Fields in this `Header` with `Names` that do not already appear in `Request.Header` are added to the end of `Request.Header`. If a field in this `Header` has a `Name` that is the same as one in `Request.Header`, and both have nonempty `Values`, then:

If one or both of these has an empty `Value`, then the field is removed from `Request.Header` and it is not added as part of normal message completion.

If this is a delegate of a `MultipartProvider`, then the entire contents of this `Header` is used as the header of the part. Multipart delegates must not assume that `Request.Header` contains any fields pertaining to their own `Header`. A provider can determine whether it is a multipart delegate by checking whether `MyDelegator` is a `MultipartProvider`, though this test is unlikely to be needed.

MATLAB reads this property only on return from calling the provider's `complete` method. Changes to this array are ignored once MATLAB calls `start`.

Class authors should be aware that their subclasses might have added fields to this `Header` (in their `complete` method) before calling `complete` in their superclass. It is best to preserve such fields and not to add fields with the same names. However, adding a parameter to a field is permissible. For example, a superclass can add a charset parameter to an existing Content-Type field that does not already have one.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`ForceChunked` — Indicate whether to force chunked transfer coding

`false` (default) | `true`

Indicate whether to force chunked transfer coding, specified as boolean. This property is of interest only to subclass authors, and is applicable only to providers that are not

multipart delegates. Subclasses set `ForceChunked` to control whether contents should be sent using chunked transfer coding. If `false` (default), MATLAB decides whether to send the contents chunked, based on whether it knows the content length at the time the message is ready to be sent:

If `ForceChunked` is `true`, then MATLAB sends the message chunked regardless of whether it knows the content length, unless the known length is smaller than the chunk size. If this property is `true`, then the message must not contain a Content-Length field, because HTTP does not allow a chunked message to have a Content-Length field. However, you can still return a nonzero value in the `expectedContentLength` method if you want MATLAB to verify that you are returning the expected length of data.

When MATLAB chooses to send the message chunked, the size of each chunk is equal to the length of data returned by `getData`.

MATLAB reads this value after calling the `complete` method, before calling `start`. It does not set this field.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`Request` — Request message to send
`matlab.net.http.RequestMessage`

Request message to send, specified as a `matlab.net.http.RequestMessage` object.

This property is used only by subclass authors. The `RequestMessage.send` and `RequestMessage.complete` methods set this property to the `RequestMessage` in whose `Body` this provider has been placed, before calling any other methods in this provider, and before adding any additional header fields or validating the message. The provider can examine this message to see what was contained in the original request.

Delegates see the same value for this property as the delegator. `ContentProviders` should be aware that, if they are delegates, they are not necessarily providing the entire body of the request message, so they should not assume that header fields in this Request are pertinent to the data they are providing. Usually, delegates should ignore header fields in this request relevant to the data, such as Content-Type.

If the provider wishes to add any header fields to this message, or to modify existing ones, it should do so in its `complete` method by adding those fields to the `Header` property.

The caller of `complete` (`RequestMessage` or a delegating provider) determines what to do with those fields. `RequestMessage.send` and `RequestMessage.complete` always copy these fields to the `Header` of the `RequestMessage`. A delegating provider can copy the fields to its own `Header` property or insert them into the message (as in the case of `MultipartProvider`). For more information, see the ContentProvider.Header property.

This property is read-only.

Attributes:

`GetAccess`	`public`
`SetAccess`	`matlab.net.http.RequestMessage`

Protected Properties

Subtype — Provider subtype

`"mixed"` (default) | string

Provider subtype, specified as a string. The default value `"mixed"` adds a Content-Type header set to `"multipart/mixed"` to the message, plus appropriate parameters. Subclasses can alter this value in the constructor or the `complete` method. This value appears in the Content-Type after `"multipart/"`.

Attributes:

`GetAccess`	`protected`
`SetAccess`	`protected`

`CurrentDelegate` — `ContentProvider` to which this provider is delegating

`matlab.net.http.io.ContentProvider` | empty

`ContentProvider` to which this provider is delegating, specified as a `matlab.net.http.io.ContentProvider` object. This property is set in the calling provider (the delegator) by the `delegateTo` method to indicate the current delegated provider. If there is no current delegation, then the value is empty.

The `complete` methods set this property to empty.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`MyDelegator` — `ContentProvider` that delegated to this provider
`matlab.net.http.io.ContentProvider.empty` (default) | `matlab.net.http.io.ContentProvider`

`ContentProvider` that delegated to this provider, specified as a `matlab.net.http.io.ContentProvider` object.

If a `ContentProvider` delegates responsibility for sending all or a portion of the message data to another provider, then this property identifies the delegating provider to the delegate. For example, a `MultipartProvider` delegates parts of the message to other providers, so it inserts a handle to itself in each delegate. Otherwise, `MyDelegator` is empty. The `delegateTo` method sets this property in the delegate.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

Methods

Public Methods

getData Next buffer of data to send in multipart HTTP request message

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions and inherited methods for objects in this class.

`string`	Provider information as string
`show`	Display provider information

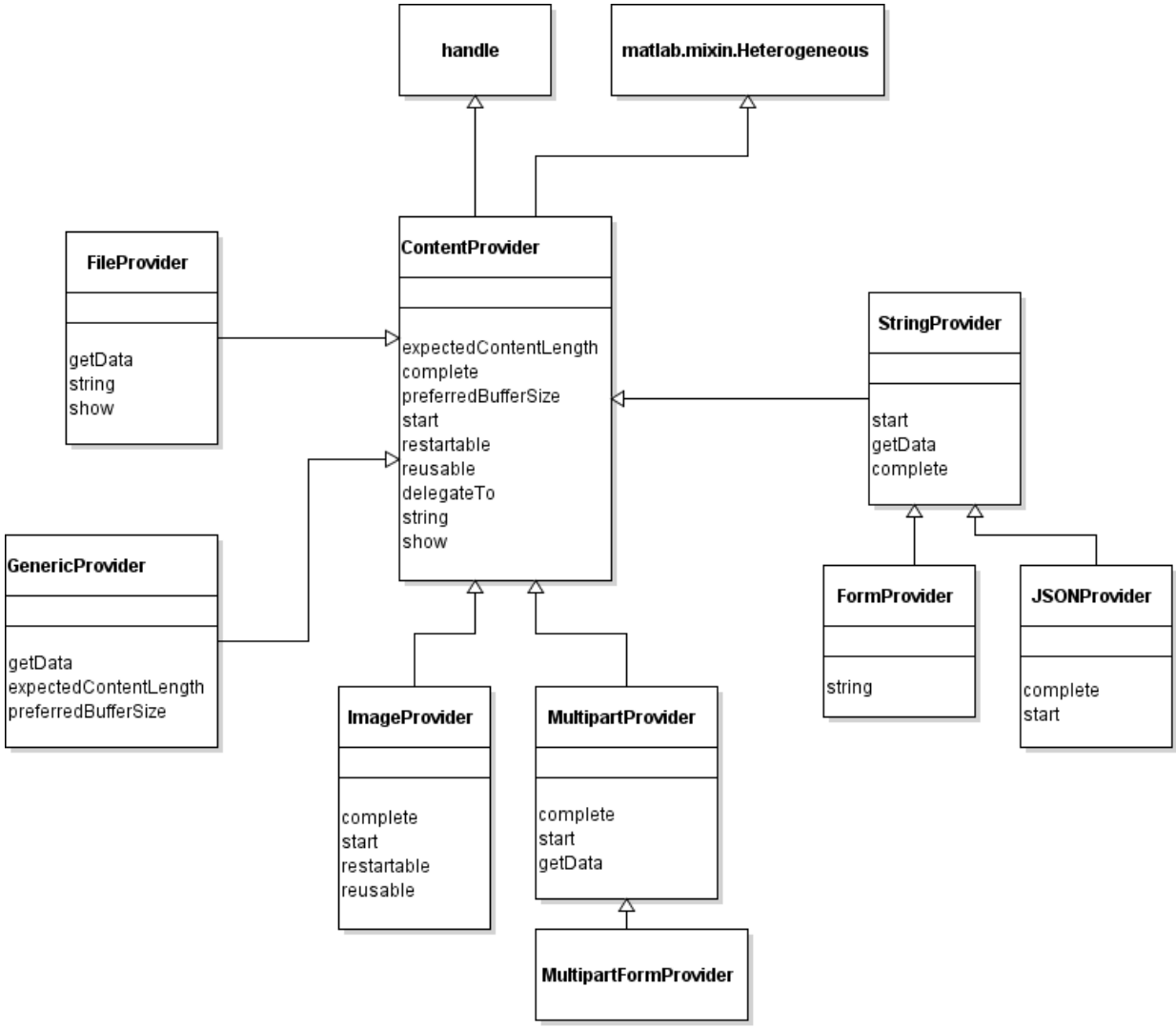
Protected Methods

complete Complete HTTP header for MultipartProvider

preferredBufferSize	Preferred buffer size for ContentProvider
expectedContentLength	Content length of ContentProvider
start	Start data transfer from MultipartProvider
restartable	Indicate whether ImageProvider is restartable
reusable	Indicate whether ImageProvider is reusable
delegateTo	Delegate to another provider

More About

Class Hierarchy



See Also

Introduced in R2018a

matlab.net.http.io.MultipartProvider class

Package: `matlab.net.http.io`
Superclasses: `matlab.net.http.io.ContentProvider`

ContentProvider to send multipart/mixed HTTP messages

Description

This provider helps with the creation of multipart HTTP messages.

The default Content-Type is `"multipart/mixed"`, and the payload of the message contains an arbitrary number of parts, each part containing its own header describing that part. For more information on multipart messages, see RFC 2046, section 5.1.

Use this provider directly only if you know that your server accepts multipart/mixed messages. Usually, servers that accept multipart messages instead require `"multipart/form-data"`, which is implemented by the subclass `MultipartFormProvider`. You can implement other multipart types using subclasses.

The `matlab.net.http.io.MultipartProvider` class is a `handle` class.

Subclass Authors

Each of the parts of the multipart message can be specified as data in any of the formats permitted for `RequestMessage.Body`, or as a `ContentProvider` that creates the data. The `ContentProvider` objects that are used to supply data for the parts are called delegates, while this `MultipartProvider` is the top level provider. In general, any `ContentProvider` is suitable as a delegate. The `MultipartProvider` invokes each delegate in turn as the message is being sent, calling its methods, such as `complete` or `start`, so that the delegate in general need not be aware that it is providing content for a part, rather than for a whole message.

This provider always transmits the `RequestMessage` as chunked, so it does not include a Content-Length header field in the message or in the headers of any of the parts. While `MultipartProvider` calls each delegate's `expectedContentLength` method before sending the part, it only uses the return value (if nonempty) to enforce the length, not to create a Content-Length field. If the delegate does want a Content-Length field to appear

in the part, it must insert such a field explicitly in its `Header` property. None of the `ContentProvider` subclasses provided by MATLAB do this processing.

Creation

Description

`provider = MultipartProvider(part1,...,partN)` constructs a `MultipartProvider` that sends the specified parts, in the specified order, in an HTTP request. By default this provider sets the Content-Type of the message to `"multipart/mixed"`, but subclasses can alter the subtype by setting the `Subtype` property. The constructor sets elements of the `Parts` property cell array to each `part1,...,partN` argument.

Properties

Public Properties

`Parts` — Message body parts
cell array

Parts of the message body, specified as a cell array of one or more of these values:

`ContentProvider` object	The `MultipartProvider` delegates creation of the part to the specified provider (called the delegate), invoking its complete method to obtain header information about the part and its `getData` method to obtain the data. The delegate's `Header` property is used for the header of the part. Any subclass of `ContentProvider` can be specified here. Normally, the delegate does not specify the content length nor implement the `expectedContentLength` method, since the end of a part is designated by a boundary string rather than a header field. If that method is implemented to return a nonempty value, then the value is used only to enforce the length of the content, not to create a Content-Length field.
`RequestMessage` object	The `MultipartProvider` sends the `Header` and `Body` of the `RequestMessage` as the part. If the Body's `Payload` property is set, then that is used for the raw payload. Otherwise the Body's `Data` property is converted based on its type or the Content-Type field in the `Header`, as described for `MessageBody.Data`. This option is useful if you have data to send and want to take advantage of the default processing of that data that MATLAB normally does when sending a `RequestMessage`. It allows you to specify custom header fields in the request to be used as the part's `Header` and control how the data is converted, without having to write a `ContentProvider` subclass. The `RequestMessage.RequestLine` property is ignored.

`MessageBody` object	The `MessageBody` is processed the same as if it was in a `RequestMessage` that had no Content-Type field. This option is useful if default processing of the data based on its type is sufficient, and you do not need to specify any custom header fields for the part. MATLAB inserts a Content-Type field in the part based on the type of the data. See `MessageBody.Data` for conversion rules.
Array of `ContentProvider`, `RequestMessage`, and/or `MessageBody` objects	This treats each element of the array as a part. Not a cell array.
Handle to `getData` method	This method must have the signature of `ContentProvider.getData`. In this case, the part's Content-Type is set to `"application/octet-stream"`, so this option is useful for sending binary data. When using this option, you cannot specify any custom header fields for the part.
Any other type	If the type does not match any of these types and is not a function handle, then it is treated as if it was present in the `Data` property of a `MessageBody`. See the description for `MessageBody` types.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`
`Dependent`	`true`

`Header` — Header fields of message or part

`matlab.net.http.HeaderField.empty` (default) | `matlab.net.http.HeaderField`

Header fields of the message or part, specified as a vector of one or more `matlab.net.http.HeaderField` objects.

This property is only used by subclass authors. MATLAB sets this property before calling the provider's `complete` method. For non-multipart messages, MATLAB initializes this

property to the contents of `Request.Header`, minus any `matlab.net.http.field.GenericFields` or empty-valued fields. The `ContentProvider` uses this property to add header fields that describe the data to be sent, or to add parameters to header fields already in the message. In a delegate for a `MultipartProvider`, MATLAB initializes this property to header fields that the delegating provider intends to insert for the part. Delegates can modify or change these fields.

Upon return from the provider's `complete` method, if this not a multipart message, then MATLAB reads this property and merges its contents into the header of `Request`. Fields in this `Header` with `Names` that do not already appear in `Request.Header` are added to the end of `Request.Header`. If a field in this `Header` has a `Name` that is the same as one in `Request.Header`, and both have nonempty `Values`, then:

If one or both of these has an empty `Value`, then the field is removed from `Request.Header` and it is not added as part of normal message completion.

If this is a delegate of a `MultipartProvider`, then the entire contents of this `Header` is used as the header of the part. Multipart delegates must not assume that `Request.Header` contains any fields pertaining to their own `Header`. A provider can determine whether it is a multipart delegate by checking whether `MyDelegator` is a `MultipartProvider`, though this test is unlikely to be needed.

MATLAB reads this property only on return from calling the provider's `complete` method. Changes to this array are ignored once MATLAB calls `start`.

Class authors should be aware that their subclasses might have added fields to this `Header` (in their `complete` method) before calling `complete` in their superclass. It is best to preserve such fields and not to add fields with the same names. However, adding a parameter to a field is permissible. For example, a superclass can add a charset parameter to an existing Content-Type field that does not already have one.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`ForceChunked` — Indicate whether to force chunked transfer coding

`false` (default) | `true`

Indicate whether to force chunked transfer coding, specified as boolean. This property is of interest only to subclass authors, and is applicable only to providers that are not

multipart delegates. Subclasses set `ForceChunked` to control whether contents should be sent using chunked transfer coding. If `false` (default), MATLAB decides whether to send the contents chunked, based on whether it knows the content length at the time the message is ready to be sent:

If `ForceChunked` is `true`, then MATLAB sends the message chunked regardless of whether it knows the content length, unless the known length is smaller than the chunk size. If this property is `true`, then the message must not contain a Content-Length field, because HTTP does not allow a chunked message to have a Content-Length field. However, you can still return a nonzero value in the `expectedContentLength` method if you want MATLAB to verify that you are returning the expected length of data.

When MATLAB chooses to send the message chunked, the size of each chunk is equal to the length of data returned by `getData`.

MATLAB reads this value after calling the `complete` method, before calling `start`. It does not set this field.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Request` — Request message to send
`matlab.net.http.RequestMessage`

Request message to send, specified as a `matlab.net.http.RequestMessage` object.

This property is used only by subclass authors. The `RequestMessage.send` and `RequestMessage.complete` methods set this property to the `RequestMessage` in whose `Body` this provider has been placed, before calling any other methods in this provider, and before adding any additional header fields or validating the message. The provider can examine this message to see what was contained in the original request.

Delegates see the same value for this property as the delegator. `ContentProviders` should be aware that, if they are delegates, they are not necessarily providing the entire body of the request message, so they should not assume that header fields in this Request are pertinent to the data they are providing. Usually, delegates should ignore header fields in this request relevant to the data, such as Content-Type.

If the provider wishes to add any header fields to this message, or to modify existing ones, it should do so in its `complete` method by adding those fields to the `Header` property.

The caller of `complete` (`RequestMessage` or a delegating provider) determines what to do with those fields. `RequestMessage.send` and `RequestMessage.complete` always copy these fields to the `Header` of the `RequestMessage`. A delegating provider can copy the fields to its own `Header` property or insert them into the message (as in the case of `MultipartProvider`). For more information, see the ContentProvider.Header property.

This property is read-only.

Attributes:

GetAccess	public
SetAccess	matlab.net.http.RequestMessage

Protected Properties

Subtype — Provider subtype

`"mixed"` (default) | string

Provider subtype, specified as a string. The default value `"mixed"` adds a Content-Type header set to `"multipart/mixed"` to the message, plus appropriate parameters. Subclasses can alter this value in the constructor or the `complete` method. This value appears in the Content-Type after `"multipart/"`.

Attributes:

GetAccess	protected
SetAccess	protected

`CurrentDelegate` — `ContentProvider` to which this provider is delegating

`matlab.net.http.io.ContentProvider` | empty

`ContentProvider` to which this provider is delegating, specified as a `matlab.net.http.io.ContentProvider` object. This property is set in the calling provider (the delegator) by the `delegateTo` method to indicate the current delegated provider. If there is no current delegation, then the value is empty.

The `complete` methods set this property to empty.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

`MyDelegator` — `ContentProvider` that delegated to this provider
`matlab.net.http.io.ContentProvider.empty` (default) | `matlab.net.http.io.ContentProvider`

`ContentProvider` that delegated to this provider, specified as a `matlab.net.http.io.ContentProvider` object.

If a `ContentProvider` delegates responsibility for sending all or a portion of the message data to another provider, then this property identifies the delegating provider to the delegate. For example, a `MultipartProvider` delegates parts of the message to other providers, so it inserts a handle to itself in each delegate. Otherwise, `MyDelegator` is empty. The `delegateTo` method sets this property in the delegate.

Attributes:

```
GetAccess                           protected
SetAccess                           protected
```

Methods

Public Methods

getData Next buffer of data to send in multipart HTTP request message

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions and inherited methods for objects in this class.

`string`	Provider information as string
`show`	Display provider information

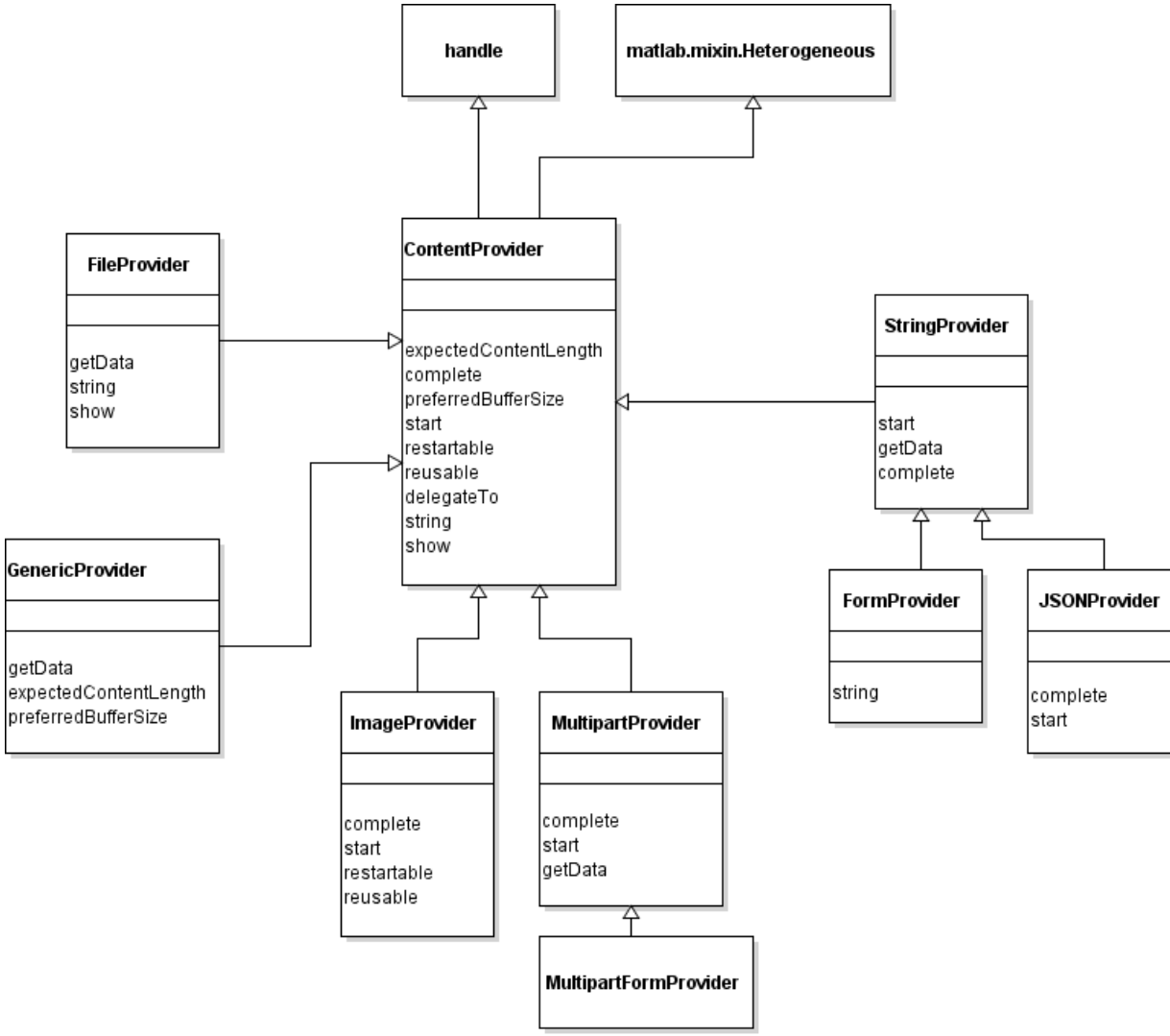
Protected Methods

complete Complete HTTP header for MultipartProvider

preferredBufferSize	Preferred buffer size for ContentProvider
expectedContentLength	Content length of ContentProvider
start	Start data transfer from MultipartProvider
restartable	Indicate whether ImageProvider is restartable
reusable	Indicate whether ImageProvider is reusable
delegateTo	Delegate to another provider

More About

Class Hierarchy



See Also

`ContentProvider` | `MessageBody` | `MultipartConsumer` | `MultipartFormProvider` | `RequestMessage`

Introduced in R2018a

complete

Class: `matlab.net.http.io.MultipartProvider`
Package: `matlab.net.http.io`

Complete HTTP header for MultipartProvider

Syntax

```
complete(provider,URI)
```

Description

`complete(provider,URI)` adds a `"multipart/subtype"` Content-Type field with appropriate parameters to the `RequestMessage`. The subtype is taken from the value of the `Subtype` property, which is, by default, `"mixed"`. If the message already contains a Content-Type field, it is preserved. If the field contains a `"boundary"` parameter, then the value of the parameter becomes the boundary delimiter. If it does not contain such a value and the type is `"multipart"`, then `complete` generates a boundary parameter and adds it to the field. If changed or added, then the new `ContentTypeField` is inserted in this provider's `Header` property.

This method is an overridden method of `complete`. Subclasses that extend `MultipartProvider` can specify their own subtype and other parameters by calling `complete` first and then modifying the ContentTypeField in `Header`.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.MultipartProvider`

Content provider, specified as a `matlab.net.http.io.MultipartProvider` object.

`URI` — Uniform resource identifier
`matlab.net.URI`

Uniform resource identifier, specified as a `matlab.net.URI` object.

Attributes

`Access`	`protected`

See Also

`complete` | `matlab.net.URI` | `matlab.net.http.field.ContentTypeField`

Introduced in R2018a

getData

Class: `matlab.net.http.io.MultipartProvider`
Package: `matlab.net.http.io`

Next buffer of data to send in multipart HTTP request message

Syntax

```
[data,stop] = getData(provider,length)
```

Description

`[data,stop] = getData(provider,length)` returns the next buffer of data. For each part of the multipart message, this method returns in successive buffers of `data`: a boundary delimiter, headers for the part, and the data for the part. It obtains these by invoking methods in the current delegate, including the delegate's `getData` method, and moves on to the next delegate when the current delegate indicates the end of its data by returning `stop=true`.

When the last delegate is done, this method returns the final boundary delimiter and then sets `stop=true` to indicate the end of the message.

This method is an overridden method of `getData`.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.MultipartProvider`

Content provider, specified as a `matlab.net.http.io.MultipartProvider` object.

`length` — Length of data
double

Length of data that the provider should return, specified as double. For more information, see the `length` argument in `ContentProvider.getData`.

Output Arguments

`data` — Next buffer of data
`uint8` vector | empty

Next buffer of data, returned as a `uint8` vector or empty.

`stop` — Indicate whether to end transmission
`false` (default) | `true`

Indicate whether to end transmission, returned as a logical that the provider must set.

Attributes

Access	public

See Also

`getData`

Introduced in R2018a

start

Class: `matlab.net.http.io.MultipartProvider`
Package: `matlab.net.http.io`

Start data transfer from MultipartProvider

Syntax

```
start(provider)
```

Description

`start(provider)` starts the transfer of the data from this provider. This method is an abstract method of `start`. MATLAB calls this method, which resets the provider so that the next call to `getData` starts the first delegate.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.MultipartProvider`

Content provider, specified as a `matlab.net.http.io.MultipartProvider` object.

Attributes

Access	protected

See Also

`delegateTo` | `start`

Introduced in R2018a

matlab.net.http.io.StringProvider class

Package: `matlab.net.http.io`
Superclasses: `matlab.net.http.io.ContentProvider`

ContentProvider to send MATLAB strings

Description

Use a `StringProvider` object to send a MATLAB string or character vector in a `RequestMessage`. By default, if a `RequestMessage.Body.Data` property contains a string or character vector, then it is converted to binary according to the encoding (charset) specified or implied by the Content-Type field in the message, so you would not normally need to use this object to send plain text in cases where MATLAB can determine what encoding to use.

Use this object in a `Request.Body` property to send a string encoded using a charset that might be different from the one that MATLAB would use for the Content-Type in the header. You specify that charset in the `ContentProvider` constructor or by setting the `Charset` property. If the message contains no Content-Type, this provider adds one specifying `"text/plain"` and the specified charset.

The `matlab.net.http.io.StringProvider` class is a `handle` class.

Creation

Description

`provider = StringProvider` constructs a `StringProvider` to send the data in the `Data` property encoded with the value specified in the `Charset` property. Set these properties before sending a message that contains this provider.

`provider = StringProvider(data,charset)` constructs a `StringProvider` to send the specified `data` encoded with the specified `charset`. The constructor sets the `Data` property to the value of `data` and the `Charset` property to the value of `charset`. The `charset` argument is optional.

Properties

Public Properties

`Data` — Data to send

string | character vector

Data to send, specified as a string or character vector. This is the value of Data that was provided to the constructor. You also can set this property directly, after calling the constructor, or in your subclass.

Subclass authors can set this property to new data at any time. The next call to `getData` converts this data, up to the value of the `getData` `length` argument.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

`Charset` — Character set used for encoding

`''` (default) | character vector

Character set used for encoding, specified as a character vector.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
Dependent                                   true
```

`Header` — Header fields of message or part

`matlab.net.http.HeaderField.empty` (default) | `matlab.net.http.HeaderField`

Header fields of the message or part, specified as a vector of one or more `matlab.net.http.HeaderField` objects.

This property is only used by subclass authors. MATLAB sets this property before calling the provider's `complete` method. For non-multipart messages, MATLAB initializes this property to the contents of `Request.Header`, minus any `matlab.net.http.field.GenericFields` or empty-valued fields. The `ContentProvider` uses this property to add header fields that describe the data to be

sent, or to add parameters to header fields already in the message. In a delegate for a `MultipartProvider`, MATLAB initializes this property to header fields that the delegating provider intends to insert for the part. Delegates can modify or change these fields.

Upon return from the provider's `complete` method, if this not a multipart message, then MATLAB reads this property and merges its contents into the header of `Request`. Fields in this `Header` with `Names` that do not already appear in `Request.Header` are added to the end of `Request.Header`. If a field in this `Header` has a `Name` that is the same as one in `Request.Header`, and both have nonempty `Values`, then:

If one or both of these has an empty `Value`, then the field is removed from `Request.Header` and it is not added as part of normal message completion.

If this is a delegate of a `MultipartProvider`, then the entire contents of this `Header` is used as the header of the part. Multipart delegates must not assume that `Request.Header` contains any fields pertaining to their own `Header`. A provider can determine whether it is a multipart delegate by checking whether `MyDelegator` is a `MultipartProvider`, though this test is unlikely to be needed.

MATLAB reads this property only on return from calling the provider's `complete` method. Changes to this array are ignored once MATLAB calls `start`.

Class authors should be aware that their subclasses might have added fields to this `Header` (in their `complete` method) before calling `complete` in their superclass. It is best to preserve such fields and not to add fields with the same names. However, adding a parameter to a field is permissible. For example, a superclass can add a charset parameter to an existing Content-Type field that does not already have one.

Attributes:

`GetAccess`	`public`
`SetAccess`	`public`

`ForceChunked` — Indicate whether to force chunked transfer coding
`false` (default) | `true`

Indicate whether to force chunked transfer coding, specified as boolean. This property is of interest only to subclass authors, and is applicable only to providers that are not multipart delegates. Subclasses set `ForceChunked` to control whether contents should be sent using chunked transfer coding. If `false` (default), MATLAB decides whether to send

the contents chunked, based on whether it knows the content length at the time the message is ready to be sent:

If `ForceChunked` is `true`, then MATLAB sends the message chunked regardless of whether it knows the content length, unless the known length is smaller than the chunk size. If this property is `true`, then the message must not contain a Content-Length field, because HTTP does not allow a chunked message to have a Content-Length field. However, you can still return a nonzero value in the `expectedContentLength` method if you want MATLAB to verify that you are returning the expected length of data.

When MATLAB chooses to send the message chunked, the size of each chunk is equal to the length of data returned by `getData`.

MATLAB reads this value after calling the `complete` method, before calling `start`. It does not set this field.

Attributes:

```
GetAccess                                   public
SetAccess                                   public
```

`Request` — Request message to send
`matlab.net.http.RequestMessage`

Request message to send, specified as a `matlab.net.http.RequestMessage` object.

This property is used only by subclass authors. The `RequestMessage.send` and `RequestMessage.complete` methods set this property to the `RequestMessage` in whose `Body` this provider has been placed, before calling any other methods in this provider, and before adding any additional header fields or validating the message. The provider can examine this message to see what was contained in the original request.

Delegates see the same value for this property as the delegator. `ContentProviders` should be aware that, if they are delegates, they are not necessarily providing the entire body of the request message, so they should not assume that header fields in this Request are pertinent to the data they are providing. Usually, delegates should ignore header fields in this request relevant to the data, such as Content-Type.

If the provider wishes to add any header fields to this message, or to modify existing ones, it should do so in its `complete` method by adding those fields to the `Header` property. The caller of `complete` (`RequestMessage` or a delegating provider) determines what to

do with those fields. `RequestMessage.send` and `RequestMessage.complete` always copy these fields to the `Header` of the `RequestMessage`. A delegating provider can copy the fields to its own `Header` property or insert them into the message (as in the case of `MultipartProvider`). For more information, see the ContentProvider.Header property.

This property is read-only.

Attributes:

GetAccess	public
SetAccess	matlab.net.http.RequestMessage

Protected Properties

`CurrentDelegate` — `ContentProvider` to which this provider is delegating

`matlab.net.http.io.ContentProvider` | empty

`ContentProvider` to which this provider is delegating, specified as a `matlab.net.http.io.ContentProvider` object. This property is set in the calling provider (the delegator) by the `delegateTo` method to indicate the current delegated provider. If there is no current delegation, then the value is empty.

The `complete` methods set this property to empty.

Attributes:

GetAccess	protected
SetAccess	protected

`MyDelegator` — `ContentProvider` that delegated to this provider

`matlab.net.http.io.ContentProvider.empty` (default) | `matlab.net.http.io.ContentProvider`

`ContentProvider` that delegated to this provider, specified as a `matlab.net.http.io.ContentProvider` object.

If a `ContentProvider` delegates responsibility for sending all or a portion of the message data to another provider, then this property identifies the delegating provider to the delegate. For example, a `MultipartProvider` delegates parts of the message to other providers, so it inserts a handle to itself in each delegate. Otherwise, `MyDelegator` is empty. The `delegateTo` method sets this property in the delegate.

Attributes:

GetAccess	protected
SetAccess	protected

Methods

Public Methods

getData Next buffer of data to send in HTTP request message from StringProvider

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions and inherited methods for objects in this class.

`string`	Provider information as string
`show`	Display provider information

Protected Methods

complete	Complete HTTP header for StringProvider
preferredBufferSize	Preferred buffer size for ContentProvider
expectedContentLength	Content length of ContentProvider
start	Start data transfer from StringProvider
restartable	Indicate whether ImageProvider is restartable
reusable	Indicate whether ImageProvider is reusable
delegateTo	Delegate to another provider

Examples

Using `StringProvider`

The following code prepares a message that sends the string `"myText"` using the Content-Type `"text/plain"` to the server using Shift_JIS encoding:

```
ctf = ContentTypeField(MediaType('text/plain','charset','Shift_JIS');
r = RequestMessage('put',ctf,StringProvider('myText'));
```

In this example, the header has no Content-Type field, so `StringProvider` inserts one based on the constructor arguments.

```
r = RequestMessage('put',[],StringProvider('myText','Shift_JIS'));
show(r.complete('www.someurl.com'))
```

```
PUT / HTTP/1.1
      Host: www.someurl.com
      Content-Type: text/plain; charset=Shift_JIS
      User-Agent: MATLAB/9.2.0.512567 (R2017b)
      Connection: close
      Date: Fri, 20 Jun 2017 14:26:42 GMT
```

In this example, the charset specified to the `StringProvider` constructor used to convert the data is different from the charset in the Content-Type field. `StringProvider` does not alter an existing Content-Type field that already specifies a character set, so the server assumes that the data is US-ASCII, not Shift-JIS.

```
ctf = ContentTypeField(MediaType('text/plain','charset','US-ASCII'));
      r = RequestMessage('put',ctf,StringProvider('myText','Shift_JIS'));
```

In this example, MATLAB adds a charset parameter to the Content-Type field that did not specify a charset, because the default for `"application/json"` is UTF-8, which is different from Shift_JIS.

```
ctf = ContentTypeField(MediaType('application/json'));
      r = RequestMessage('put',ctf,StringProvider('myText','Shift_JIS'));
      show(r.complete('www.someurl.com'))
```

```
PUT / HTTP/1.1
      Host: www.someurl.com
      Content-Type: application/json; charset=Shift_JIS
      User-Agent: MATLAB/9.2.0.512567 (R2017b)
      Connection: close
      Date: Fri, 20 Jun 2017 14:26:42 GMT
```

When there is no Content-Type header field and no charset is specified to `StringProvider`, MATLAB uses a heuristic to find the "minimal" encoding that can represent the data, one of which includes the default encoding for the platform. In this example when run on Windows, the Unicode characters in the string are within the Windows-1252 range, but outside the US-ASCII range, so Windows-1252 is used:

```
r = RequestMessage('put',[],StringProvider('€abc'));
      show(r.complete('www.someurl.com'))
```

```
PUT / HTTP/1.1
      Host: www.someurl.com
```

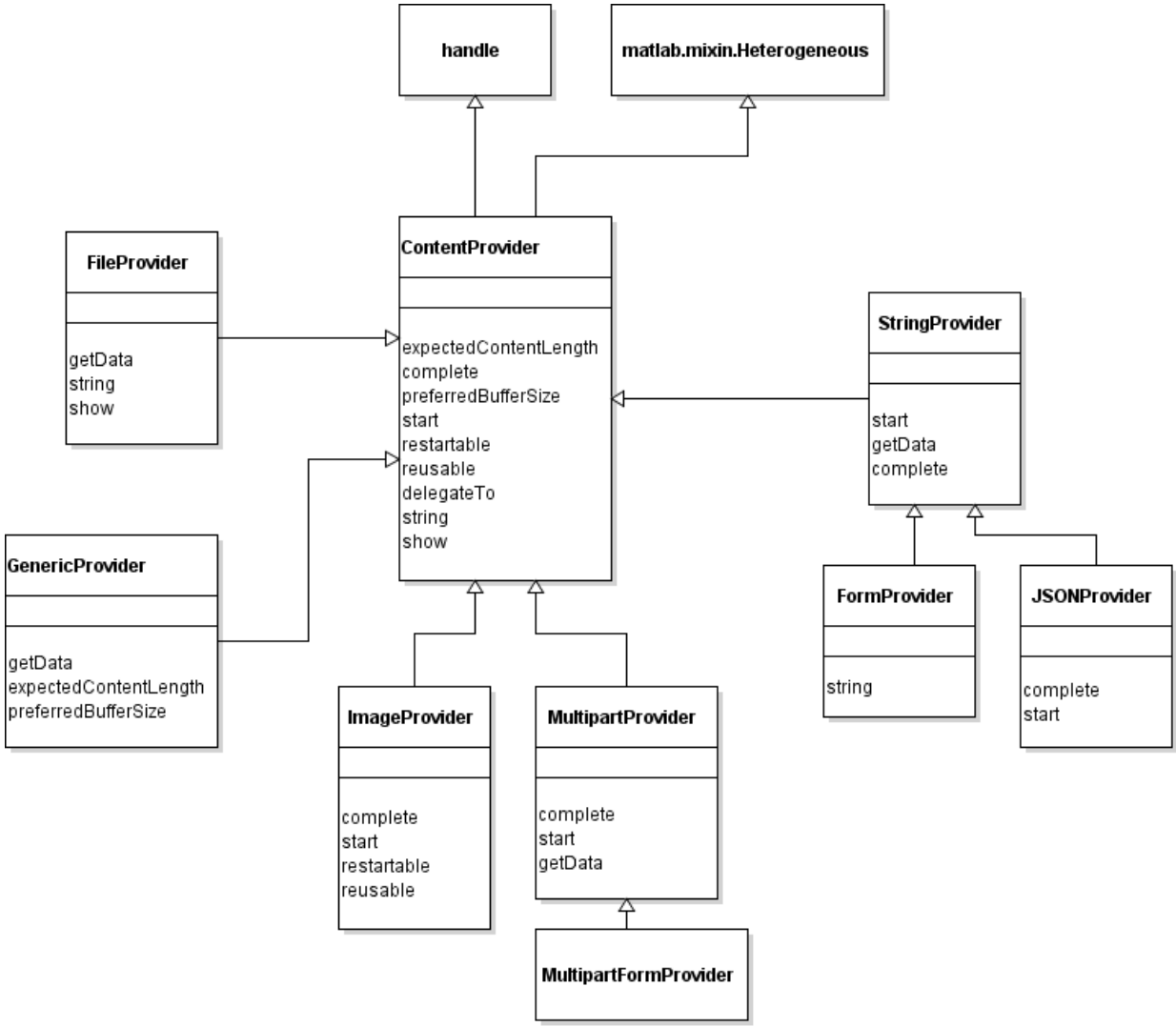
```
Content-Type: text/plain; charset=windows-1252
User-Agent: MATLAB/9.2.0.512567 (R2017b)
Connection: close
Date: Fri, 20 Jun 2017 14:26:42 GMT
```

In this case, the Content-Type field specifies `"application/json"` with no charset, and none is specified to `StringProvider`. Since the default charset for `"application/json"` is UTF-8, `StringProvider` uses that to convert and does not specify the charset explicitly in the Content-Type field.

```
ctf = ContentTypeField(MediaType('application/json'));
      r = RequestMessage('put',ctf,StringProvider('myText')); % uses UTF-8
```

More About

Class Hierarchy



See Also

`StringConsumer` | `getData`

Introduced in R2018a

complete

Class: `matlab.net.http.io.StringProvider`
Package: `matlab.net.http.io`

Complete HTTP header for StringProvider

Syntax

```
complete(provider,URI)
```

Description

`complete(provider,URI)` completes the HTTP message. This method is an overridden method of `complete` called by MATLAB. This method might augment or add a Content-Type header field to the message to specify the charset that this provider is using to convert the data. The conversion to be used depends on the value of the Content-Type field in the `Header` or `Request.Header` property, if present (which might have an explicit or default charset), and the value of the `Charset` property in this object. This provider might add a Content-Type field or charset parameter to the existing Content-Type field, if it does not contain one. To prevent that, subclasses can override this method.

In contrast to some other providers that only replace, not alter, a header already in the `RequestMessage`, this provider might augment an existing Content-Type field in the `Request` property by adding a charset parameter, if necessary.

On return from this method, the `Charset` property is always set to the charset used to encode the data, whether or not that charset is explicit in the Content-Type field. Subclasses can override this method to specify a different `Charset`.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.StringProvider`

Content provider, specified as a `matlab.net.http.io.StringProvider` object.

`URI` — Uniform resource identifier
`matlab.net.URI`

Uniform resource identifier, specified as a `matlab.net.URI` object.

Attributes

`Access`	`protected`

See Also

Charset | ContentProvider.Header | ContentProvider.Request | `complete` | `matlab.net.URI`

Introduced in R2018a

getData

Class: `matlab.net.http.io.StringProvider`
Package: `matlab.net.http.io`

Next buffer of data to send in HTTP request message from StringProvider

Syntax

```
[data,stop] = getData(provider,length)
```

Description

`[data,stop] = getData(provider,length)` returns the next buffer of data.

This method is an overridden method of `getData`. Subclasses that generate their own buffers of data in an overridden `getData` method, but which want to take advantage of code conversion provide by this method, should set `Data` to their buffer of data and call this superclass `getData` method to convert `Data` to the desired charset. In that call, specify a value of `length` at least as large as the number of characters in the buffer, or only part of the `Data` is converted. For example:

```
function [data, stop] = getData(obj, length)
    obj.Data = generateNextBufferOfData(obj);
    if isempty(obj.Data)
        stop = true;
    else
        [data, stop] = getData(obj, strlength(obj.Data);
    end
end
```

Input Arguments

`provider` — Content provider
`matlab.net.http.io.StringProvider`

Content provider, specified as a `matlab.net.http.io.StringProvider` object.

`length` — Length of data
double

Length of data that the provider should return, specified as double. For more information, see the `length` argument in `ContentProvider.getData`.

Output Arguments

`data` — Next buffer of data
`uint8` vector | empty

Next buffer of data, returned as a `uint8` vector or empty. `getData` normally returns at least `length` bytes (up to the length of the `Data` property), as a `uint8` vector, by reading up to `length` characters from `Data`, but, depending on the characters in `Data` and `Charset`, the result might be much longer than `length`.

`stop` — Indicate whether to end transmission
`false` (default) | `true`

Indicate whether to end transmission, returned as a logical that the provider must set.

Attributes

`Access`	`public`

See Also

Charset | Data | `getData`

Introduced in R2018a

start

Class: `matlab.net.http.io.StringProvider`
Package: `matlab.net.http.io`

Start data transfer from StringProvider

Syntax

```
start(provider)
```

Description

`start(provider)` prepares this `StringProvider` for a new transfer. This method is an overridden method of `start` called by MATLAB.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.StringProvider`

Content provider, specified as a `matlab.net.http.io.StringProvider` object.

Attributes

Access	protected

See Also

`start`

Introduced in R2018a

string

Class: `matlab.net.http.io.StringProvider`
Package: `matlab.net.http.io`

StringProvider Data property

Syntax

```
str = string(provider)
```

Description

`str = string(provider)` returns the contents of the StringProvider.Data property as a string. If `Data` is not set, then the method returns an empty string. The `show` method returns the same information.

Input Arguments

`provider` — Content provider
`matlab.net.http.io.StringProvider`

Content provider specified as a `matlab.net.http.io.StringProvider` object.

Attributes

Access	public

See Also

StringProvider.Data | `show`

Introduced in R2018a

hypot

Square root of sum of squares (hypotenuse)

Syntax

```
C = hypot(A,B)
```

Description

`C = hypot(A,B)` returns the result of the following equation, computed to avoid underflow and overflow:

```
C = sqrt(abs(A).^2 + abs(B).^2)
```

Examples

Compute Hypotenuse

Compute the hypotenuse of a right triangle with side lengths of `3` and `4`.

```
C = hypot(3,4)
```

```
C = 5
```

Overflow and Underflow Behavior

Examine the difference between using `hypot` and coding the basic `hypot` equation in M-code.

Create an anonymous function that performs essentially the same basic function as `hypot`.

```
myhypot = @(a,b)sqrt(abs(a).^2+abs(b).^2);
```

`myhypot` does not have the same consideration for underflow and overflow behavior that `hypot` offers.

Find the upper limit at which `myhypot` returns a useful value. You can see that this test function reaches its maximum at about `1e154`, returning an infinite result at that point.

```
myhypot(1e153,1e153)
```

```
ans = 1.4142e+153
```

```
myhypot(1e154,1e154)
```

```
ans = Inf
```

Do the same using the `hypot` function, and observe that `hypot` operates on values up to about `1e308`, which is approximately equal to the value for `realmax` on your computer (the largest representable double-precision floating-point number).

```
hypot(1e308,1e308)
```

```
ans = 1.4142e+308
```

```
hypot(1e309,1e309)
```

```
ans = Inf
```

Input Arguments

`A,B` — Input arrays
scalars | vectors | matrices | multidimensional arrays

Input arrays, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs `A` and `B` must either be the same size or have sizes that are compatible (for example, `A` is an `M`-by-`N` matrix and `B` is a scalar or `1`-by-`N` row vector). For more information, see “Compatible Array Sizes for Basic Operations”.

If neither `A` nor `B` is `Inf`, but one or both inputs is `NaN`, then `hypot` returns `NaN`.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

IEEE Compliance

For real inputs, `hypot` has a few behaviors that differ from those recommended in the IEEE-754 Standard.

	MATLAB	IEEE
`hypot(NaN,Inf)`	`NaN`	`Inf`
`hypot(NaN,-Inf)`	`NaN`	`Inf`
`hypot(Inf,NaN)`	`NaN`	`Inf`
`hypot(-Inf,NaN)`	`NaN`	`Inf`

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`abs` | `norm` | `sqrt`

Introduced before R2006a

i

Imaginary unit

Syntax

```
1i
z = a + bi
z = x + 1i*y
```

Description

`1i` returns the basic imaginary unit. `i` is equivalent to `sqrt(-1)`.

You can use `i` to enter complex numbers. You also can use the character `j` as the imaginary unit. To create a complex number without using `i` and `j`, use the `complex` function.

`z = a + bi` returns a complex numerical constant, `z`.

`z = x + 1i*y` returns a complex array, `z`.

Examples

Complex Scalar

Create a complex scalar and use the character, `i`, without a multiplication sign as a suffix in forming a complex numerical constant.

```
z = 1+2i
```

```
z = 1.0000 + 2.0000i
```

Complex Vector

Create a complex vector from two 4-by-1 vectors of real numbers. `z` is a 4-by-1 complex vector.

```
x = [1:4]';
y = [8:-2:2]';

z = x+1i*y
```

```
z = 4×1 complex

   1.0000 + 8.0000i
   2.0000 + 6.0000i
   3.0000 + 4.0000i
   4.0000 + 2.0000i
```

Complex Exponential

Create a complex scalar representing a complex vector with radius, `r`, and angle from the origin, `theta`.

```
r = 4;
theta = pi/4;

z = r*exp(1i*theta)
```

```
z = 2.8284 + 2.8284i
```

Input Arguments

a — Real component of complex scalar
scalar

Real component of a complex scalar, specified as a scalar.

Data Types: `single` | `double`

b — Imaginary component of complex scalar
scalar

Imaginary component of a complex scalar, specified as a scalar.

If `b` is `double`, you can use the character, `i`, without a multiplication sign as a suffix in forming the complex numerical constant.

Example: `7i`

If `b` is `single`, you must use a multiplication sign when forming the complex numerical constant.

Example: `single(7)*i`

Data Types: `single` | `double`

x — Real component of complex array
scalar | vector | matrix | multidimensional array

Real component of a complex array, specified as a scalar, vector, matrix, or multidimensional array.

The size of `x` must match the size of `y`, unless one is a scalar. If either `x` or `y` is a scalar, MATLAB expands the scalar to match the size of the other input.

`single` can combine with `double`.

Data Types: `single` | `double`

y — Imaginary component of complex array
scalar | vector | matrix | multidimensional array

Imaginary component of a complex array, specified as a scalar, vector, matrix, or multidimensional array.

The size of `x` must match the size of `y`, unless one is a scalar. If either `x` or `y` is a scalar, MATLAB expands the scalar to match the size of the other input.

`single` can combine with `double`.

Data Types: `single` | `double`

Output Arguments

z — Complex array
scalar | vector | matrix | multidimensional array

Complex array, returned as a scalar, vector, matrix, or multidimensional array.

The size of `z` is the same as the input arguments.

`z` is `single` if at least one input argument is `single`. Otherwise, `z` is `double`.

Tips

- For speed and improved robustness in complex arithmetic, use `1i` and `1j` instead of `i` and `j`.
- Since `i` is a function, it can be overridden and used as a variable. However, it is best to avoid using `i` and `j` for variable names if you intend to use them in complex arithmetic.
- Use the `complex` function to create a complex output in the following cases:
  - When the names `i` and `j` might be used for other variables (and do not equal `sqrt(-1)`)
  - When the inputs are not `double` or `single`
  - When the imaginary component is all zeros

See Also

`complex` | `conj` | `imag` | `j` | `real`

Topics

"Create Complex Numbers"

Introduced before R2006a

ichol

Incomplete Cholesky factorization

Syntax

```
L = ichol(A)
L = ichol(A,opts)
```

Description

`L = ichol(A)` performs the incomplete Cholesky factorization of `A` with zero-fill.

`L = ichol(A,opts)` performs the incomplete Cholesky factorization of `A` with options specified by `opts`.

By default, `ichol` references the lower triangle of `A` and produces lower triangular factors.

Input Arguments

`A`

Sparse matrix

`opts`

Structure with up to five fields:

Field Name	Summary	Description
`type`	Type of factorization	Indicates which flavor of incomplete Cholesky to perform. Valid values of this field are `'nofill'` and `'ict'`. The `'nofill'` variant performs incomplete Cholesky with zero-fill (`IC(0)`). The `'ict'` variant performs incomplete Cholesky with threshold dropping (ICT). The default value is `'nofill'`.
`droptol`	Drop tolerance when type is `'ict'`	Nonnegative scalar used as a drop tolerance when performing ICT. Elements which are smaller in magnitude than a local drop tolerance are dropped from the resulting factor except for the diagonal element which is never dropped. The local drop tolerance at step `j` of the factorization is `norm(A(j:end,j),1)*droptol`. `'droptol'` is ignored if `'type'` is `'nofill'`. The default value is `0`.
`michol`	Indicates whether to perform modified incomplete Cholesky	Indicates whether or not modified incomplete Cholesky (MIC) is performed. The field may be `'on'` or `'off'`. When performing MIC, the diagonal is compensated for dropped elements to enforce the relationship `A*e = L*L'*e` where `e = ones(size(A,2),1)`. The default value is `'off'`.

Field Name	Summary	Description
`diagcomp`	Perform compensated incomplete Cholesky with the specified coefficient	Real nonnegative scalar used as a global diagonal shift `alpha` in forming the incomplete Cholesky factor. That is, instead of performing incomplete Cholesky on `A`, the factorization of `A + alpha*diag(diag(A))` is formed. The default value is `0`.
`shape`	Determines which triangle is referenced and returned	Valid values are `'upper'` and `'lower'`. If `'upper'` is specified, only the upper triangle of `A` is referenced and `R` is constructed such that `A` is approximated by `R'*R`. If `'lower'` is specified, only the lower triangle of `A` is referenced and `L` is constructed such that `A` is approximated by `L*L'`. The default value is `'lower'`.

Examples

Incomplete Cholesky Factorization

This example generates an incomplete Cholesky factorization.

Start with a symmetric positive definite matrix, `A`:

```
N = 100;
A = delsq(numgrid('S',N));
```

`A` is the two-dimensional, five-point discrete negative Laplacian on a 100-by-100 square grid with Dirichlet boundary conditions. The size of `A` is `98*98 = 9604` (not 10000 as the borders of the grid are used to impose the Dirichlet conditions).

The no-fill incomplete Cholesky factorization is a factorization which contains only nonzeros in the same position as `A` contains nonzeros. This factorization is extremely

cheap to compute. Although the product `L*L'` is typically very different from `A`, the product `L*L'` will match `A` on its pattern up to round-off.

```
L = ichol(A);
norm(A-L*L','fro')./norm(A,'fro')
```

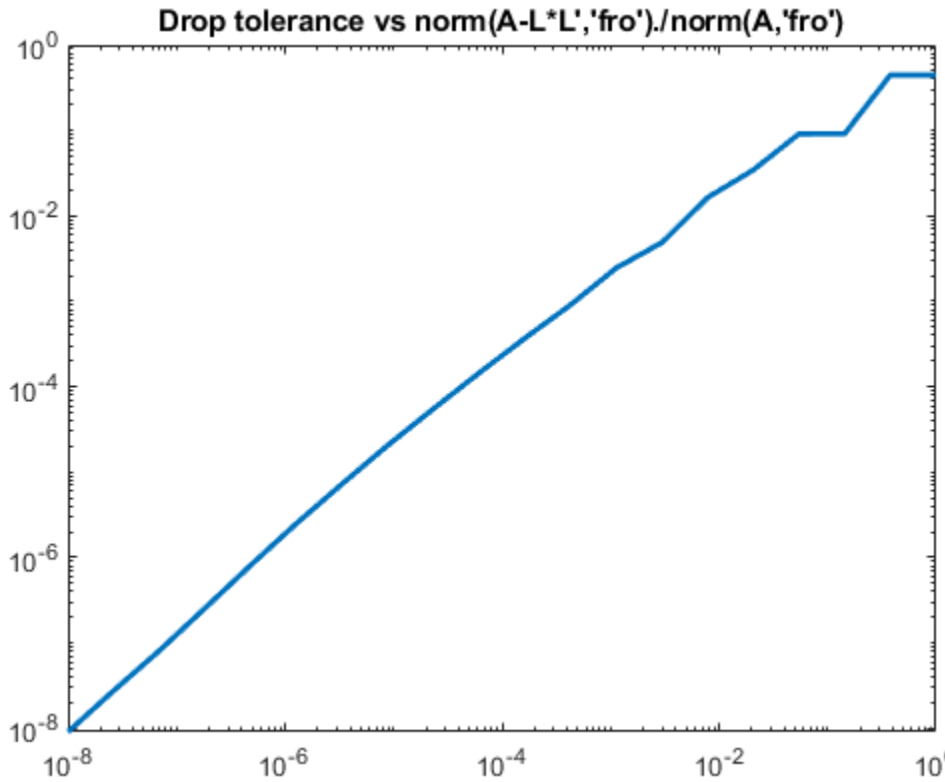
```
ans = 0.0916
```

```
norm(A-(L*L').*spones(A),'fro')./norm(A,'fro')
```

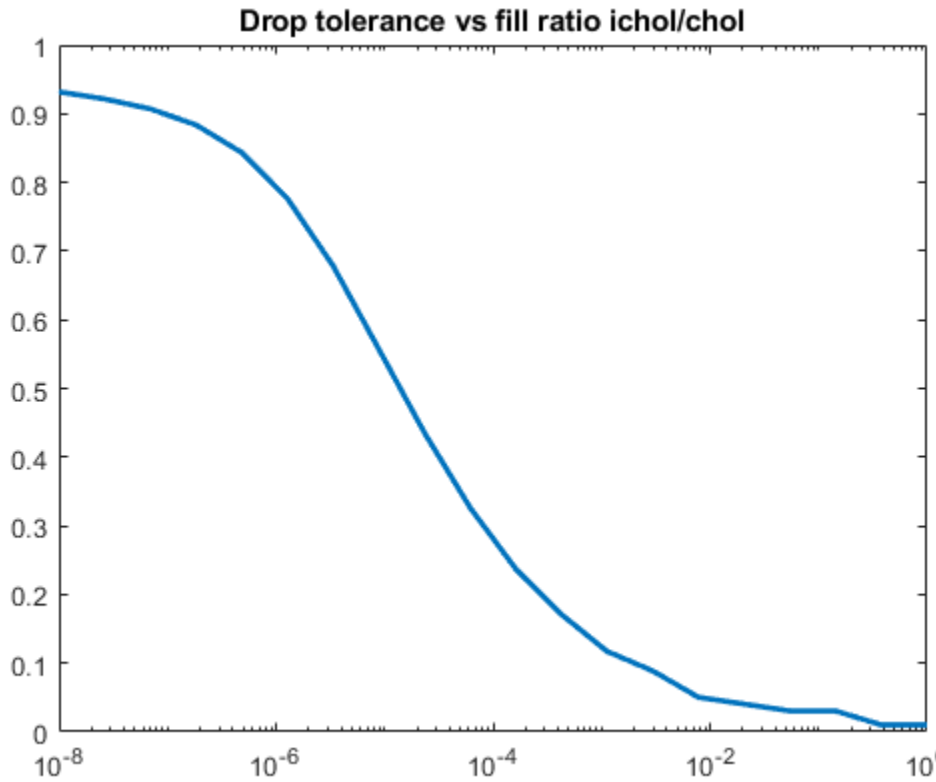
```
ans = 4.9606e-17
```

`ichol` may also be used to generate incomplete Cholesky factorizations with threshold dropping. As the drop tolerance decreases, the factor tends to get more dense and the product `L*L'` tends to be a better approximation of `A`. The following plots show the relative error of the incomplete factorization plotted against the drop tolerance as well as the ratio of the density of the incomplete factors to the density of the complete Cholesky factor.

```
n = size(A,1);
ntols = 20;
droptol = logspace(-8,0,ntols);
nrm = zeros(1,ntols);
nz = zeros(1,ntols);
nzComplete = nnz(chol(A,'lower'));
for k = 1:ntols
    L = ichol(A,struct('type','ict','droptol',droptol(k)));
    nz(k) = nnz(L);
    nrm(k) = norm(A-L*L','fro')./norm(A,'fro');
end
figure
loglog(droptol,nrm,'LineWidth',2)
title('Drop tolerance vs norm(A-L*L'',''fro'')./norm(A,''fro'')')
```




```
figure
semilogx(droptol,nz./nzComplete,'LineWidth',2)
title('Drop tolerance vs fill ratio ichol/chol')
```



The relative error is typically on the same order as the drop tolerance, although this is not guaranteed.

Using ichol as a Preconditioner

This example shows how to use an incomplete Cholesky factorization as a preconditioner to improve convergence.

Create a symmetric positive definite matrix, A.

```
N = 100;
A = delsq(numgrid('S',N));
```

Create an incomplete Cholesky factorization as a preconditioner for `pcg`. Use a constant vector as the right hand side. As a baseline, execute `pcg` without a preconditioner.

```
b = ones(size(A,1),1);
tol = 1e-6;
maxit = 100;
[x0,fl0,rr0,it0,rv0] = pcg(A,b,tol,maxit);
```

Note that `fl0 = 1` indicating that `pcg` did not drive the relative residual to the requested tolerance in the maximum allowed iterations. Try the no-fill incomplete Cholesky factorization as a preconditioner.

```
L1 = ichol(A);
[x1,fl1,rr1,it1,rv1] = pcg(A,b,tol,maxit,L1,L1');
```

`fl1 = 0`, indicating that `pcg` converged to the requested tolerance and did so in 59 iterations (the value of `it1`). Since this matrix is a discretized Laplacian, however, using modified incomplete Cholesky can create a better preconditioner. A modified incomplete Cholesky factorization constructs an approximate factorization that preserves the action of the operator on the constant vector. That is, `norm(A*e-L*(L'*e))` will be approximately zero for `e = ones(size(A,2),1)` even though `norm(A-L*L','fro')/norm(A,'fro')` is not close to zero. It is not necessary to specify type for this syntax since `nofill` is the default, but it is good practice.

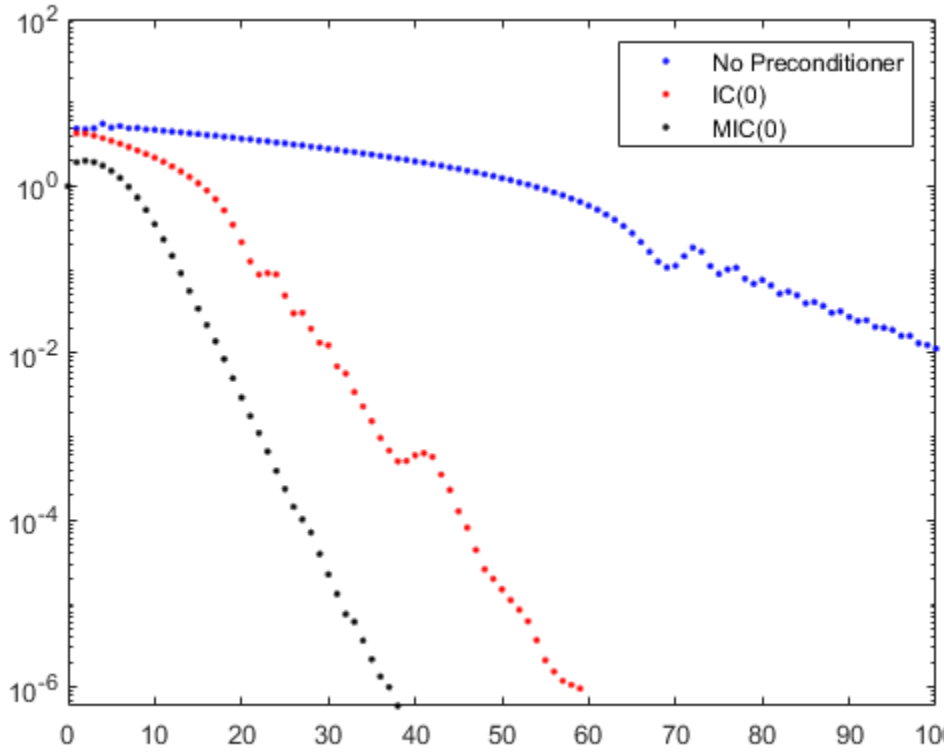
```
opts.type = 'nofill';
opts.michol = 'on';
L2 = ichol(A,opts);
e = ones(size(A,2),1);
norm(A*e-L2*(L2'*e))
```

```
ans = 3.7983e-14
```

```
[x2,fl2,rr2,it2,rv2] = pcg(A,b,tol,maxit,L2,L2');
```

`pcg` converges (`fl2 = 0`) but in only 38 iterations. Plotting all three convergence histories shows the convergence.

```
semilogy(0:maxit,rv0./norm(b),'b.');
hold on
semilogy(0:it1,rv1./norm(b),'r.');
semilogy(0:it2,rv2./norm(b),'k.');
legend('No Preconditioner','IC(0)','MIC(0)');
```

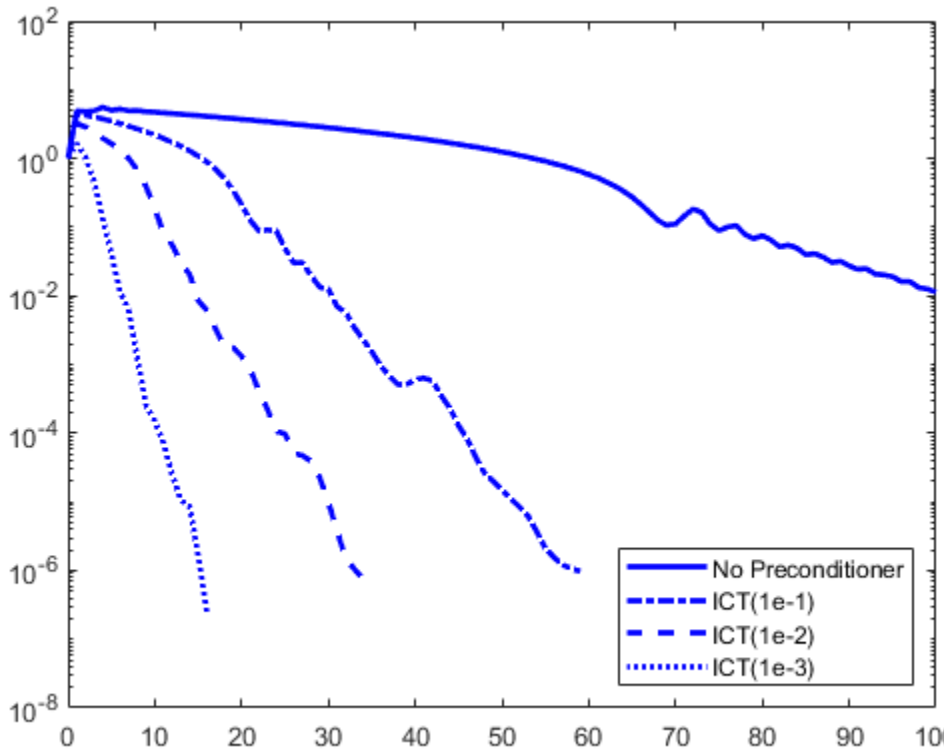


The plot shows that the modified incomplete Cholesky preconditioner creates a much faster convergence.

You can also try incomplete Cholesky factorizations with threshold dropping. The following plot shows convergence of `pcg` with preconditioners constructed with various drop tolerances.

```
L3 = ichol(A, struct('type','ict','droptol',1e-1));
[x3,fl3,rr3,it3,rv3] = pcg(A,b,tol,maxit,L3,L3');
L4 = ichol(A, struct('type','ict','droptol',1e-2));
[x4,fl4,rr4,it4,rv4] = pcg(A,b,tol,maxit,L4,L4');
L5 = ichol(A, struct('type','ict','droptol',1e-3));
[x5,fl5,rr5,it5,rv5] = pcg(A,b,tol,maxit,L5,L5');
```

```
figure
semilogy(0:maxit,rv0./norm(b),'b-','linewidth',2);
hold on
semilogy(0:it3,rv3./norm(b),'b-.','linewidth',2);
semilogy(0:it4,rv4./norm(b),'b--','linewidth',2);
semilogy(0:it5,rv5./norm(b),'b:','linewidth',2);
legend('No Preconditioner','ICT(1e-1)','ICT(1e-2)', ...
   'ICT(1e-3)','Location','SouthEast');
```



Note the incomplete Cholesky preconditioner constructed with drop tolerance `1e-2` is denoted as `ICT(1e-2)`.

As with the zero-fill incomplete Cholesky, the threshold dropping factorization can benefit from modification (i.e. `opts.michol = 'on'`) since the matrix arises from an elliptic

partial differential equation. As with `MIC(0)`, the modified threshold based dropping incomplete Cholesky will preserve the action of the preconditioner on constant vectors, that is `norm(A*e-L*(L'*e))` will be approximately zero.

Using the diagcomp Option

This example illustrates the use of the `diagcomp` option of `ichol`.

Incomplete Cholesky factorizations of positive definite matrices do not always exist. The following code constructs a random symmetric positive definite matrix and attempts to solve a linear system using `pcg`.

```
S = rng('default');
A = sprandsym(1000,1e-2,1e-4,1);
rng(S);
b = full(sum(A,2));
[x0,fl0,rr0,it0,rv0] = pcg(A,b,1e-6,100);
```

Since convergence is not attained, try to construct an incomplete Cholesky preconditioner.

```
L = ichol(A,struct('type','ict','droptol',1e-3));
```

```
Error using ichol
Encountered nonpositive pivot.
```

If `ichol` breaks down as above, you can use the `diagcomp` option to construct a shifted incomplete Cholesky factorization. That is, instead of constructing `L` such that `L*L'` approximates `A`, `ichol` with diagonal compensation constructs `L` such that `L*L'` approximates `M = A + alpha*diag(diag(A))` without explicitly forming `M`. As incomplete factorizations always exist for diagonally dominant matrices, `alpha` can be found to make `M` diagonally dominant.

```
alpha = max(sum(abs(A),2)./diag(A))-2
```

```
alpha = 62.9341
```

```
L1 = ichol(A, struct('type','ict','droptol',1e-3,'diagcomp',alpha));
[x1,fl1,rr1,it1,rv1] = pcg(A,b,1e-6,100,L1,L1');
```

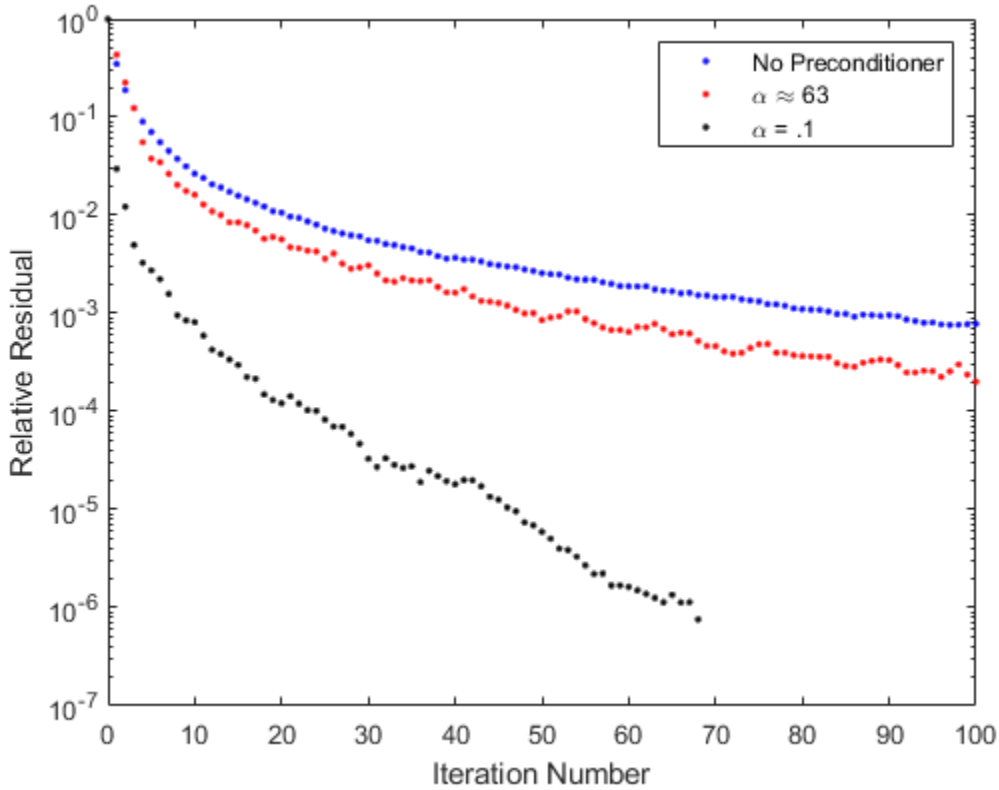
Here, `pcg` still fails to converge to the desired tolerance within the desired number of iterations, but as the plot below shows, convergence is better for `pcg` with this

preconditioner than with no preconditioner. Choosing a smaller `alpha` may help. With some experimentation, we can settle on an appropriate value for `alpha`.

```
alpha = .1;
L2 = ichol(A, struct('type','ict','droptol',1e-3,'diagcomp',alpha));
[x2,fl2,rr2,it2,rv2] = pcg(A,b,1e-6,100,L2,L2');
```

Now, `pcg` converges and a plot can show the convergence histories with each preconditioner.

```
semilogy(0:100,rv0./norm(b),'b.');
hold on;
semilogy(0:100,rv1./norm(b),'r.');
semilogy(0:it2,rv2./norm(b),'k.');
legend('No Preconditioner','\alpha \approx 63','\alpha = .1');
xlabel('Iteration Number');
ylabel('Relative Residual');
```



Tips

- The factor given by this routine may be useful as a preconditioner for a system of linear equations being solved by iterative methods such as `pcg` or `minres`.
- `ichol` works only for sparse square matrices

References

[1] Saad, Yousef. “Preconditioning Techniques.” *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, 1996.

[2] Manteuffel, T.A. “An incomplete factorization technique for positive definite linear systems.” *Math. Comput.* 34, 473–497, 1980.

See Also

`chol` | `ilu` | `minres` | `pcg`

idivide

Integer division with rounding option

Syntax

```
C = idivide(A,B)
C = idivide(A,B,opt)
```

Description

`C = idivide(A,B)` divides each element of `A` by the corresponding element of `B`, rounded to the nearest integers toward zero. `A` and `B` must contain real numbers and at least one of them must belong to an integer class.

- If `A` and `B` are arrays, then they must belong to the same integer class and have sizes that are compatible.
- If `A` or `B` is a scalar double, then the other input must be an integer class, but not `int64` or `uint64`. The `idivide` function then returns `C` as the same integer class.

`C = idivide(A,B,opt)` specifies an alternative rounding option: `'fix'`, `'floor'`, `'ceil'`, or `'round'`. For example, `idivide(A,B,'ceil')` rounds the quotients to the nearest integers toward positive infinity. The default rounding option is `'fix'`.

Examples

Integer Division

Create an integer array `A` and an integer scalar `B`. Divide each element of `A` by `B`, rounded to the nearest integers toward zero.

```
A = int16([-7 -4 7 11]);
B = int16(10);
C = idivide(A,B)
```

```
C = 1x4 int16 row vector

   0   0   0   1
```

Divide Two Integer Arrays

Create two integer arrays, A and B. Divide each element of A by the corresponding element of B, rounded to the nearest integers toward zero.

```
A = int64([-2 3]);
B = int64([3 5]);
C = idivide(A,B)
```

```
C = 1x2 int64 row vector

   0   0
```

Rounding Options for Integer Division

Create a scalar double A and an integer array B. Divide A by each element of B with the default rounding option `'fix'`.

```
A = 2.0;
B = int32([-3 3 4]);
C = idivide(A,B)
```

```
C = 1x3 int32 row vector

   0   0   0
```

Compare the results with other rounding options.

```
C = idivide(A,B,'floor')
```

```
C = 1x3 int32 row vector
```

```
   -1    0    0
```

```
C = idivide(A,B,'ceil')
```

```
C = 1x3 int32 row vector

   0   1   1
```

```
C = idivide(A,B,'round')
```

```
C = 1x3 int32 row vector

   -1    1    1
```

Input Arguments

A — Numerator
scalar | vector | matrix | multidimensional array

Numerator, specified as a scalar, vector, matrix, or multidimensional array. Integer inputs A and B must be either the same size or have sizes that are compatible. For example, A is an M-by-N matrix, and B is a scalar or 1-by-N row vector. For more information, see "Compatible Array Sizes for Basic Operations".

If A is a scalar double, then B must be an integer class, but not `int64` or `uint64`.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

B — Denominator
scalar | vector | matrix | multidimensional array

Denominator, specified as a scalar, vector, matrix, or multidimensional array. Integer inputs A and B must be either the same size or have sizes that are compatible. For example, A is an M-by-N matrix, and B is a scalar or 1-by-N row vector. For more information, see "Compatible Array Sizes for Basic Operations".

If B is a scalar double, then A must be an integer class, but not `int64` or `uint64`.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`opt` — Rounding option
`'fix'` (default) | `'floor'` | `'ceil'` | `'round'`

Rounding option, specified as `'fix'`, `'floor'`, `'ceil'`, or `'round'`.

- `'fix'` rounds to the nearest integers toward zero, which is equivalent to removing any digits after the decimal point.
- `'floor'` rounds to the nearest integers toward negative infinity.
- `'ceil'` rounds to the nearest integer toward positive infinity.
- `'round'` rounds to the nearest integers. If an element has a fractional part of exactly 0.5, then it rounds away from zero to the integer with larger magnitude.

Output Arguments

`C` — Integer solution
scalar | vector | matrix | multidimensional array

Integer solution, returned as a scalar, vector, matrix, or multidimensional array. If either `A` or `B` is an integer data type, then `C` is the same integer data type.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Tips

- The function `idivide(A,B,'round')` is the same as `A./B` and `B.\A` for integer data types. The argument `opt` provides the rounding options for `A./B` and `B.\A` when dividing integers.
- MATLAB does not support complex integer division.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For efficient generated code, MATLAB rules for divide by zero are supported only for the `'round'` option.
- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`ceil` | `fix` | `floor` | `ldivide` | `rdivide` | `round`

Topics

"Integers"
"Floating-Point Numbers"

Introduced in R2006a

if, elseif, else

Execute statements if condition is true

Syntax

```
if expression
    statements
elseif expression
    statements
else
    statements
end
```

Description

`if expression, statements, end` evaluates an expression on page 1-7281, and executes a group of statements when the expression is true. An expression is true when its result is nonempty and contains only nonzero elements (logical or real numeric). Otherwise, the expression is false.

The `elseif` and `else` blocks are optional. The statements execute only if previous expressions in the `if...end` block are false. An `if` block can include multiple `elseif` blocks.

Examples

Use if, elseif, and else for Conditional Assignment

Create a matrix of 1s.

```
nrows = 4;
ncols = 6;
A = ones(nrows,ncols);
```

Loop through the matrix and assign each element a new value. Assign `2` on the main diagonal, `-1` on the adjacent diagonals, and `0` everywhere else.

```
for c = 1:ncols
    for r = 1:nrows

        if r == c
            A(r,c) = 2;
        elseif abs(r-c) == 1
            A(r,c) = -1;
        else
            A(r,c) = 0;
        end

    end
end
A
```

```
A = 4×6

     2    -1     0     0     0     0
    -1     2    -1     0     0     0
     0    -1     2    -1     0     0
     0     0    -1     2    -1     0
```

Compare Arrays

Expressions that include relational operators on arrays, such as `A > 0`, are true only when every element in the result is nonzero.

Test if any results are true using the `any` function.

```
limit = 0.75;
A = rand(10,1)
```

```
A = 10×1

    0.8147
    0.9058
    0.1270
    0.9134
```

```
    0.6324
    0.0975
    0.2785
    0.5469
    0.9575
    0.9649
```

```
if any(A > limit)
    disp('There is at least one value above the limit.')
else
    disp('All values are below the limit.')
end
```

```
There is at least one value above the limit.
```

Test Arrays for Equality

Compare arrays using `isequal` rather than the == operator to test for equality, because == results in an error when the arrays are different sizes.

Create two arrays.

```
A = ones(2,3);
B = rand(3,4,5);
```

If `size(A)` and `size(B)` are the same, concatenate the arrays; otherwise, display a warning and return an empty array.

```
if isequal(size(A),size(B))
   C = [A; B];
else
   disp('A and B are not the same size.')
   C = [];
end
```

```
A and B are not the same size.
```

Compare Character Vectors

Use `strcmp` to compare character vectors. Using == to test for equality results in an error when the character vectors are different sizes.

```
reply = input('Would you like to see an echo? (y/n): ','s');
if strcmp(reply,'y')
  disp(reply)
end
```

Evaluate Multiple Conditions in Expression

Determine if a value falls within a specified range.

```
x = 10;
minVal = 2;
maxVal = 6;

if (x >= minVal) && (x <= maxVal)
    disp('Value within specified range.')
elseif (x > maxVal)
    disp('Value exceeds maximum value.')
else
    disp('Value is below minimum value.')
end
```

```
Value exceeds maximum value.
```

More About

Expression

An expression can include relational operators (such as < or ==) and logical operators (such as &&, ||, or ~). Use the logical operators `and` and `or` to create compound expressions. MATLAB evaluates compound expressions from left to right, adhering to operator precedence rules.

Within the conditional expression of an `if...end` block, logical operators & and | behave as short-circuit operators. This behavior is the same as && and ||, respectively. Since &&

and || consistently short-circuit in conditional expressions and statements, it is good practice to use && and || instead of & and | within the expression. For example,

```
x = 42;
if exist('myfunction.m','file') && (myfunction(x) >= pi)
    disp('Expressions are true')
end
```

The first part of the expression evaluates to false. Therefore, MATLAB does not need to evaluate the second part of the expression, which would result in an undefined function error.

Tips

- You can nest any number of `if` statements. Each `if` statement requires an `end` keyword.
- Avoid adding a space after `else` within the `elseif` keyword (`else if`). The space creates a nested `if` statement that requires its own `end` keyword.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`for` | `return` | `switch` | `while`

Topics

"Operators and Elementary Operations"

Introduced before R2006a

ifft

Inverse fast Fourier transform

Syntax

```
X = ifft(Y)
X = ifft(Y,n)
X = ifft(Y,n,dim)
X = ifft( ___ ,symflag)
```

Description

`X = ifft(Y)` computes the inverse discrete Fourier transform on page 1-7287 of `Y` using a fast Fourier transform algorithm. `X` is the same size as `Y`.

- If `Y` is a vector, then `ifft(Y)` returns the inverse transform of the vector.
- If `Y` is a matrix, then `ifft(Y)` returns the inverse transform of each column of the matrix.
- If `Y` is a multidimensional array, then `ifft(Y)` treats the values along the first dimension whose size does not equal 1 as vectors and returns the inverse transform of each vector.

`X = ifft(Y,n)` returns the `n`-point inverse Fourier transform of `Y` by padding `Y` with trailing zeros to length `n`.

`X = ifft(Y,n,dim)` returns the inverse Fourier transform along the dimension `dim`. For example, if `Y` is a matrix, then `ifft(Y,n,2)` returns the `n`-point inverse transform of each row.

`X = ifft( ___ ,symflag)` specifies the symmetry of `Y`. For example, `ifft(Y,'symmetric')` treats `Y` as conjugate symmetric.

Examples

Inverse Transform of Vector

The Fourier transform and its inverse convert between data sampled in time and space and data sampled in frequency.

Create a vector and compute its Fourier transform.

```
X = [1 2 3 4 5];
Y = fft(X)
```

```
Y = 1×5 complex

  15.0000 + 0.0000i  -2.5000 + 3.4410i  -2.5000 + 0.8123i  -2.5000 - 0.8123i  -2.5000 -
```

Compute the inverse transform of `Y`, which is the same as the original vector `X`.

```
ifft(Y)
```

```
ans = 1×5

     1     2     3     4     5
```

Padded Inverse Transform of Matrix

The `ifft` function allows you to control the size of the transform.

Create a random 3-by-5 matrix and compute the 8-point inverse Fourier transform of each row. Each row of the result has length 8.

```
Y = rand(3,5);
n = 8;
X = ifft(Y,n,2);
size(X)
```

```
ans = 1×2

     3     8
```

Conjugate Symmetric Vector

For nearly conjugate symmetric vectors, you can compute the inverse Fourier transform faster by specifying the `'symmetric'` option, which also ensures that the output is real. Nearly conjugate symmetric data can arise when computations introduce round-off error.

Create a vector Y that is nearly conjugate symmetric and compute its inverse Fourier transform. Then, compute the inverse transform specifying the `'symmetric'` option, which eliminates the nearly 0 imaginary parts.

```
Y = [1 2:4+eps(4) 4:-1:2]
```

```
Y = 1×7

    1.0000    2.0000    3.0000    4.0000    4.0000    3.0000    2.0000
```

```
X = ifft(Y)
```

```
X = 1×7 complex

   2.7143 + 0.0000i  -0.7213 + 0.0000i  -0.0440 - 0.0000i  -0.0919 + 0.0000i  -0.0919 -
```

```
Xsym = ifft(Y,'symmetric')
```

```
Xsym = 1×7

    2.7143   -0.7213   -0.0440   -0.0919   -0.0919   -0.0440   -0.7213
```

Input Arguments

Y — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, a matrix, or a multidimensional array. If Y is of type `single`, then `ifft` natively computes in single precision, and X is also of type `single`. Otherwise, X is returned as type `double`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

Complex Number Support: Yes

n — Inverse transform length
[] (default) | nonnegative integer scalar

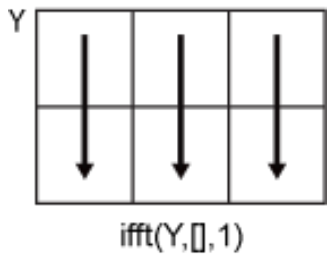
Inverse transform length, specified as `[]` or a nonnegative integer scalar. Padding `Y` with zeros by specifying a transform length larger than the length of `Y` can improve the performance of `ifft`. The length is typically specified as a power of 2 or a product of small prime numbers. If `n` is less than the length of the signal, then `ifft` ignores the remaining signal values past the `n`th entry and returns the truncated result. If `n` is 0, then `ifft` returns an empty matrix.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

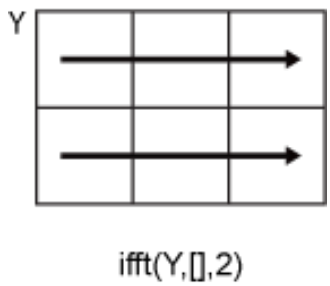
dim — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. By default, `dim` is the first array dimension whose size does not equal 1. For example, consider a matrix `Y`.

- `ifft(Y,[],1)` returns the inverse Fourier transform of each column.



- `ifft(Y,[],2)` returns the inverse Fourier transform of each row.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

`symflag` — Symmetry type
`'nonsymmetric'` (default) | `'symmetric'`

Symmetry type, specified as `'nonsymmetric'` or `'symmetric'`. When `Y` is not exactly conjugate symmetric due to round-off error, `ifft(Y,'symmetric')` treats `Y` as if it were conjugate symmetric. For more information on conjugate symmetry, see "Algorithms" on page 1-7287.

More About

Discrete Fourier Transform of Vector

`Y = fft(X)` and `X = ifft(Y)` implement the Fourier transform and inverse Fourier transform, respectively. For `X` and `Y` of length `n`, these transforms are defined as follows:

$$Y(k) = \sum_{j=1}^{n} X(j)\, W_n^{(j-1)(k-1)}$$

$$X(j) = \frac{1}{n} \sum_{k=1}^{n} Y(k)\, W_n{}^{-(j-1)(k-1)},$$

where

$$W_n = e^{(-2\pi i)/n}$$

is one of n roots of unity.

Algorithms

- The `ifft` function tests whether the vectors in `Y` are conjugate symmetric. A vector `v` is conjugate symmetric when it equals `conj(v([1,end:-1:2]))`. If the vectors in `Y` are conjugate symmetric, then the inverse transform computation is faster and the output is real.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Output is complex.
- Symmetry type `'symmetric'` is not supported.
- For limitations related to variable-size data, see “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).
- For MEX output, MATLAB Coder uses the library that MATLAB uses for FFT algorithms. For standalone C/C++ code, by default, the code generator produces code for FFT algorithms instead of producing FFT library calls. To generate calls to a specific installed FFTW library, provide an FFT library callback class. For more information about an FFT library callback class, see `coder.fftw.StandaloneFFTW3Interface`.
- For simulation of a MATLAB Function block, the simulation software uses the library that MATLAB uses for FFT algorithms. For C/C++ code generation, by default, the code generator produces code for FFT algorithms instead of producing FFT library calls. To generate calls to a specific installed FFTW library, provide an FFT library callback class. For more information about an FFT library callback class, see `coder.fftw.StandaloneFFTW3Interface`.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Unless `symflag` is `'symmetric'`, the output is always complex even if all imaginary parts are zero.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`fft` | `fftw` | `ifft2` | `ifftn` | `ifftshift`

Introduced before R2006a

ifft2

2-D inverse fast Fourier transform

Syntax

```
X = ifft2(Y)
X = ifft2(Y,m,n)
X = ifft2( ___ ,symflag)
```

Description

`X = ifft2(Y)` returns the two-dimensional discrete inverse Fourier transform on page 1-7293 of a matrix using a fast Fourier transform algorithm. If `Y` is a multidimensional array, then `ifft2` takes the 2-D inverse transform of each dimension higher than 2. The output `X` is the same size as `Y`.

`X = ifft2(Y,m,n)` truncates `Y` or pads `Y` with trailing zeros to form an `m`-by-`n` matrix before computing the inverse transform. `X` is also `m`-by-`n`. If `Y` is a multidimensional array, then `ifft2` shapes the first two dimensions of `Y` according to `m` and `n`.

`X = ifft2( ___ ,symflag)` specifies the symmetry of `Y`. For example, `ifft2(Y,'symmetric')` treats `Y` as conjugate symmetric.

Examples

2-D Inverse Transform of Matrix

You can use the `ifft2` function to convert 2-D signals sampled in frequency to signals sampled in time or space. The `ifft2` function also allows you to control the size of the transform.

Create a 3-by-3 matrix and compute its Fourier transform.

```
X = magic(3)
```

```
X = 3×3

     8     1     6
     3     5     7
     4     9     2
```

```
Y = fft2(X)
```

```
Y = 3×3 complex

  45.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i  13.5000 + 7.7942i   0.0000 - 5.1962i
   0.0000 - 0.0000i   0.0000 + 5.1962i  13.5000 - 7.7942i
```

Take the inverse transform of Y, which is the same as the original matrix X, up to round-off error.

```
ifft2(Y)
```

```
ans = 3×3

    8.0000    1.0000    6.0000
    3.0000    5.0000    7.0000
    4.0000    9.0000    2.0000
```

Pad both dimensions of Y with trailing zeros so that the transform has size 8-by-8.

```
Z = ifft2(Y,8,8);
size(Z)
```

```
ans = 1×2

     8     8
```

Conjugate Symmetric Matrix

For nearly conjugate symmetric matrices, you can compute the inverse Fourier transform faster by specifying the `'symmetric'` option, which also ensures that the output is real.

Compute the 2-D inverse Fourier transform of a nearly conjugate symmetric matrix.

```
Y = [3+1e-15*i 5;
     5 3];
X = ifft2(Y,'symmetric')
```

```
X = 2×2

     4     0
     0    -1
```

Input Arguments

Y — Input array
matrix | multidimensional array

Input array, specified as a matrix or a multidimensional array. If `Y` is of type `single`, then `ifft2` natively computes in single precision, and `X` is also of type `single`. Otherwise, `X` is returned as type `double`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`
Complex Number Support: Yes

m — Number of inverse transform rows
positive integer scalar

Number of inverse transform rows, specified as a positive integer scalar.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

n — Number of inverse transform columns
positive integer scalar

Number of inverse transform columns, specified as a positive integer scalar.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

symflag — Symmetry type
`'nonsymmetric'` (default) | `'symmetric'`

Symmetry type, specified as `'nonsymmetric'` or `'symmetric'`. When `Y` is not exactly conjugate symmetric due to round-off error, `ifft2(Y,'symmetric')` treats `Y` as if it were conjugate symmetric. For more information on conjugate symmetry, see "Algorithms" on page 1-7293.

More About

2-D Inverse Fourier Transform

This formula defines the discrete inverse Fourier transform X of an m-by-n matrix Y:

$$X_{p,q} = \frac{1}{m}\sum_{j=1}^{m}\frac{1}{n}\sum_{k=1}^{n}\omega_m^{(j-1)(p-1)}\omega_n^{(k-1)(q-1)}Y_{j,k}$$

ω_m and ω_n are complex roots of unity:

$$\omega_m = e^{2\pi i/m}$$
$$\omega_n = e^{2\pi i/n}$$

i is the imaginary unit. p runs from 1 to m and q runs from 1 to n.

Algorithms

- The `ifft2` function tests whether the vectors in a matrix `Y` are conjugate symmetric in both dimensions. A vector `v` is conjugate symmetric when the ith element satisfies `v(i) = conj(v([1,end:-1:2]))`. If the vectors in `Y` are conjugate symmetric in both dimensions, then the inverse transform computation is faster and the output is real.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Symmetry type `'symmetric'` is not supported.
- For MEX output, MATLAB Coder uses the library that MATLAB uses for FFT algorithms. For standalone C/C++ code, by default, the code generator produces code for FFT algorithms instead of producing FFT library calls. To generate calls to a specific installed FFTW library, provide an FFT library callback class. For more information about an FFT library callback class, see `coder.fftw.StandaloneFFTW3Interface`.
- For simulation of a MATLAB Function block, the simulation software uses the library that MATLAB uses for FFT algorithms. For C/C++ code generation, by default, the code generator produces code for FFT algorithms instead of producing FFT library calls. To generate calls to a specific installed FFTW library, provide an FFT library callback class. For more information about an FFT library callback class, see `coder.fftw.StandaloneFFTW3Interface`.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Unless `symflag` is `'symmetric'`, the output is always complex even if all imaginary parts are zero.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`fft2` | `fftw` | `ifft` | `ifftn` | `ifftshift`

Introduced before R2006a

ifftn

Multidimensional inverse fast Fourier transform

Syntax

```
X = ifftn(Y)
X = ifftn(Y,sz)
X = ifftn( ___ ,symflag)
```

Description

`X = ifftn(Y)` returns the multidimensional discrete inverse Fourier transform on page 1-7298 of an N-D array using a fast Fourier transform algorithm. The N-D inverse transform is equivalent to computing the 1-D inverse transform along each dimension of `Y`. The output `X` is the same size as `Y`.

`X = ifftn(Y,sz)` truncates `Y` or pads `Y` with trailing zeros before taking the inverse transform according to the elements of the vector `sz`. Each element of `sz` defines the length of the corresponding transform dimension. For example, if `Y` is a 5-by-5-by-5 array, then `X = ifftn(Y,[8 8 8])` pads each dimension with zeros, resulting in an 8-by-8-by-8 inverse transform `X`.

`X = ifftn( ___ ,symflag)` specifies the symmetry of `Y`. For example, `ifftn(Y,'symmetric')` treats `Y` as conjugate symmetric.

Examples

3-D Inverse Transform

You can use the `ifftn` function to convert multidimensional data sampled in frequency to data sampled in time or space. The `ifftn` function also allows you to control the size of the transform.

Create a 3-by-3-by-3 array and compute its inverse Fourier transform.

```
Y = rand(3,3,3);
ifftn(Y);
```

Pad the dimensions of Y with trailing zeros so that the transform has size 8-by-8-by-8.

```
X = ifftn(Y,[8 8 8]);
size(X)
```

```
ans = 1×3

     8     8     8
```

Conjugate Symmetric Array

For nearly conjugate symmetric arrays, you can compute the inverse Fourier transform faster by specifying the `'symmetric'` option, which also ensures that the output is real.

Compute the 3-D inverse Fourier transform of a nearly conjugate symmetric array.

```
Y(:,:,1) = [1e-15*i 0; 1 0];
Y(:,:,2) = [0 1; 0 1];
X = ifftn(Y,'symmetric')
```

```
X =
X(:,:,1) =

    0.3750   -0.1250
   -0.1250   -0.1250


X(:,:,2) =

   -0.1250    0.3750
   -0.1250   -0.1250
```

Input Arguments

Y — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, a matrix, or a multidimensional array. If Y is of type `single`, then `ifftn` natively computes in single precision, and X is also of type `single`. Otherwise, X is returned as type `double`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`
Complex Number Support: Yes

`sz` — Lengths of inverse transform dimensions
vector of positive integers

Lengths of inverse transform dimensions, specified as a vector of positive integers.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

`symflag` — Symmetry type
`'nonsymmetric'` (default) | `'symmetric'`

Symmetry type, specified as `'nonsymmetric'` or `'symmetric'`. When Y is not exactly conjugate symmetric due to round-off error, `ifftn(Y,'symmetric')` treats Y as if it were conjugate symmetric. For more information on conjugate symmetry, see "Algorithms" on page 1-7299.

More About

N-D Inverse Fourier Transform

The discrete inverse Fourier transform X of an N-D array Y is defined as

$$X_{p_1, p_2, \ldots, p_N} = \sum_{j_1=1}^{m_1} \frac{1}{m_1} \omega_{m_1}^{p_1 j_1} \sum_{j_2=1}^{m_2} \frac{1}{m_2} \omega_{m_2}^{p_2 j_2} \ldots \sum_{j_N=1}^{m_N} \frac{1}{m_N} \omega_{m_N}^{p_N j_N} Y_{j_1, j_2, \ldots, j_N}.$$

Each dimension has length m_k for $k = 1,2,...,N$, and $\omega_{m_k} = e^{2\pi i/m_k}$ are complex roots of unity where i is the imaginary unit.

Algorithms

- The `ifftn` function tests whether the vectors in an array `Y` are conjugate symmetric in all dimensions. A vector `v` is conjugate symmetric when the ith element satisfies `v(i) = conj(v([1,end:-1:2]))`. If the vectors in `Y` are conjugate symmetric in all dimensions, then the inverse transform computation is faster and the output is real.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Symmetry type `'symmetric'` is not supported.
- The `sz` argument must have a fixed size.
- For MEX output, MATLAB Coder uses the library that MATLAB uses for FFT algorithms. For standalone C/C++ code, by default, the code generator produces code for FFT algorithms instead of producing FFT library calls. To generate calls to a specific installed FFTW library, provide an FFT library callback class. For more information about an FFT library callback class, see `coder.fftw.StandaloneFFTW3Interface`.
- For simulation of a MATLAB Function block, the simulation software uses the library that MATLAB uses for FFT algorithms. For C/C++ code generation, by default, the code generator produces code for FFT algorithms instead of producing FFT library calls. To generate calls to a specific installed FFTW library, provide an FFT library callback class. For more information about an FFT library callback class, see `coder.fftw.StandaloneFFTW3Interface`.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Unless `symflag` is `'symmetric'`, the output is always complex even if all imaginary parts are zero.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`fftn` | `fftw` | `ifft` | `ifft2` | `ifftshift`

Introduced before R2006a

ifftshift

Inverse zero-frequency shift

Syntax

```
X = ifftshift(Y)
X = ifftshift(Y,dim)
```

Description

`X = ifftshift(Y)` rearranges a zero-frequency-shifted Fourier transform `Y` back to the original transform output. In other words, `ifftshift` undoes the result of `fftshift`.

- If `Y` is a vector, then `ifftshift` swaps the left and right halves of `Y`.
- If `Y` is a matrix, then `ifftshift` swaps the first quadrant of `Y` with the third, and the second quadrant with the fourth.
- If `Y` is a multidimensional array, then `ifftshift` swaps half-spaces of `Y` along each dimension.

`X = ifftshift(Y,dim)` operates along the dimension `dim` of `Y`. For example, if `Y` is a matrix whose rows represent multiple 1-D transforms, then `ifftshift(Y,2)` swaps the halves of each row of `Y`.

Examples

Inverse Shift Vector Elements

You can use the `fftshift` and `ifftshift` functions to swap left and right halves of a vector.

Create a vector containing an odd number of elements, and swap the left and right sides of the vector using the `fftshift` function.

```
V = [1 2 3 4 5 6 7];
X = fftshift(V)
```

```
X = 1×7

     5     6     7     1     2     3     4
```

Use the `ifftshift` function to swap the left and right sides of `X`. The result is the same as the original vector `V`.

```
Y = ifftshift(X)
```

```
Y = 1×7

     1     2     3     4     5     6     7
```

Calling the `fftshift` function twice does not necessarily reconstruct the original input.

```
fftshift(fftshift(V))
```

```
ans = 1×7

     2     3     4     5     6     7     1
```

Inverse Shift Matrix Rows

Rearrange the rows of a matrix to shift the nonnegative elements to the left.

```
Y = [-2 -1 0 1 2;
    -10 -5 0 5 10];
X = ifftshift(Y,2)
```

```
X = 2×5

     0     1     2    -2    -1
     0     5    10   -10    -5
```

Input Arguments

Y — Input array
vector | matrix | multidimensional array

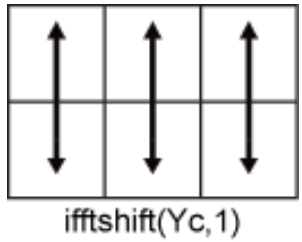
Input array, specified as a vector, a matrix, or a multidimensional array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

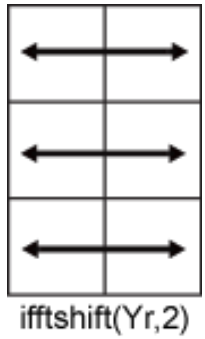
`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then `ifftshift` swaps along all dimensions.

- Consider an input matrix `Yc`. The operation `ifftshift(Yc,1)` swaps halves of each column of `Yc`.



- Consider an input matrix `Yr`. The operation `ifftshift(Yr,2)` swaps halves of each row of `Yr`.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`fft` | `fft2` | `fftn` | `fftshift` | `fftw`

Introduced before R2006a

ilu

Incomplete LU factorization

Syntax

```
ilu(A,setup)
[L,U] = ilu(A,setup)
[L,U,P] = ilu(A,setup)
```

Description

`ilu` produces a unit lower triangular matrix, an upper triangular matrix, and a permutation matrix.

`ilu(A,setup)` computes the incomplete LU factorization of `A`. `setup` is an input structure with up to five setup options. The fields must be named exactly as shown in the table below. You can include any number of these fields in the structure and define them in any order. Any additional fields are ignored.

Field Name	Description
`type`	Type of factorization. Values for `type` include: • `'nofill'`(default)—Performs ILU factorization with `0` level of fill in, known as ILU(0). With `type` set to `'nofill'`, only the `milu` setup option is used; all other fields are ignored. • `'crout'`—Performs the Crout version of ILU factorization, known as ILUC. With `type` set to `'crout'`, only the `droptol` and `milu` setup options are used; all other fields are ignored. • `'ilutp'` —Performs ILU factorization with threshold and pivoting. If `type` is not specified, the ILU factorization with `0` level of fill in is performed. Pivoting is only performed with `type` set to `'ilutp'`.

Field Name	Description

`droptol`	Drop tolerance of the incomplete LU factorization. `droptol` is a non-negative scalar. The default value is `0`, which produces the complete LU factorization.
	
	The nonzero entries of `U` satisfy
	
	`abs(U(i,j)) >= droptol*norm(A(:,j)),`
	
	with the exception of the diagonal entries, which are retained regardless of satisfying the criterion. The entries of `L` are tested against the local drop tolerance before being scaled by the pivot, so for nonzeros in `L`
	
	`abs(L(i,j)) >= droptol*norm(A(:,j))/U(j,j).`

`milu`	Modified incomplete LU factorization. Values for `milu` include: • `'row'`—Produces the row-sum modified incomplete LU factorization. Entries from the newly-formed column of the factors are subtracted from the diagonal of the upper triangular factor, `U`, preserving column sums. That is, `A*e = L*U*e`, where `e` is the vector of ones. • `'col'`—Produces the column-sum modified incomplete LU factorization. Entries from the newly-formed column of the factors are subtracted from the diagonal of the upper triangular factor, `U`, preserving column sums. That is, `e'*A = e'*L*U`. • `'off'` (default)—No modified incomplete LU factorization is produced.
`udiag`	If `udiag` is `1`, any zeros on the diagonal of the upper triangular factor are replaced by the local drop tolerance. The default is `0`.
`thresh`	Pivot threshold between `0` (forces diagonal pivoting) and `1`, the default, which always chooses the maximum magnitude entry in the column to be the pivot.

`ilu(A,setup)` returns `L+U-speye(size(A))`, where `L` is a unit lower triangular matrix and `U` is an upper triangular matrix.

`[L,U] = ilu(A,setup)` returns a unit lower triangular matrix in `L` and an upper triangular matrix in `U`.

`[L,U,P] = ilu(A,setup)` returns a unit lower triangular matrix in `L`, an upper triangular matrix in `U`, and a permutation matrix in `P`.

Limitations

`ilu` works on sparse square matrices only.

Examples

Start with a sparse matrix and compute the LU factorization.

```
A = gallery('neumann', 1600) + speye(1600);
setup.type = 'crout';
setup.milu = 'row';
setup.droptol = 0.1;
[L,U] = ilu(A,setup);
e = ones(size(A,2),1);
norm(A*e-L*U*e)

ans =

  1.4251e-014
```

This shows that `A` and `L*U`, where `L` and `U` are given by the modified Crout `ILU`, have the same row-sum.

Start with a sparse matrix and compute the LU factorization.

```
A = gallery('neumann', 1600) + speye(1600);
setup.type = 'nofill';
nnz(A)
ans =

        7840

nnz(lu(A))
ans =

      126478

nnz(ilu(A,setup))
```

```
ans =

        7840
```

This shows that `A` has `7840` nonzeros, the complete LU factorization has `126478` nonzeros, and the incomplete LU factorization, with `0` level of fill-in, has `7840` nonzeros, the same amount as `A`.

Tips

These incomplete factorizations may be useful as preconditioners for a system of linear equations being solved by iterative methods such as BICG (BiConjugate Gradients), GMRES (Generalized Minimum Residual Method).

References

[1] Saad, Yousef, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, 1996, Chapter 10 - Preconditioning Techniques.

See Also

`bicg` | `gmres` | `ichol`

im2double

Convert image to double precision

Syntax

```
I2 = im2double(I)
I2 = im2double(I,'indexed')
```

Description

`I2 = im2double(I)` converts the image `I` to double precision. `I` can be a grayscale intensity image, a truecolor image, or a binary image. `im2double` rescales the output from integer data types to the range [0, 1].

`I2 = im2double(I,'indexed')` converts the indexed image `I` to double precision. `im2double` adds an offset of 1 to the output from integer data types.

Examples

Convert Image to Double Precision

```
I = imread('peppers.png');
whos I
```

```
  Name        Size                Bytes  Class    Attributes

  I         384x512x3            589824  uint8
```

```
I2 = im2double(I);
whos I2
```

```
  Name        Size                  Bytes  Class     Attributes

  I2        384x512x3            4718592  double
```

Convert Image to Double Precision on GPU

Convert an array to class `double` on the GPU. This example requires the Parallel Computing Toolbox.

```
I1 = gpuArray(reshape(uint8(linspace(1,255,25)),[5 5]));
I2 = im2double(I1);
```

Input Arguments

`I` — Input image
scalar | vector | matrix | multidimensional array

Input image, specified as a numeric scalar, vector, matrix, or multidimensional array.

- If `I` is a grayscale or truecolor (RGB) image, it can be `uint8`, `uint16`, `double`, `logical`, `single`, or `int16`.
- If `I` is an indexed image, it can be `uint8`, `uint16`, `double` or `logical`.
- If `I` is a binary image, it must be `logical`.

If the Parallel Computing Toolbox is installed, then `I` can be a gpuArray and `im2double` converts `I` on a GPU.

Data Types: `single` | `double` | `int16` | `uint8` | `uint16` | `logical`

Output Arguments

`I2` — Converted image
numeric array

Converted image, returned as a numeric array of the same size as the input image `I`.

Data Types: `double`

Tips

- If the data type of the input image `I` is `double`, `single`, or `logical`, then the output pixel values are identical to the input pixel values.

  Note Many MATLABfunctions expect pixel values to be in the range [0, 1] for truecolor images of data type `single` or `double`. The `im2double` function does not rescale the output when the input image has `single` or `double` data type. If your input image is a truecolor image of data type `single` or `double` with pixel values outside this range, then you can use the `rescale` function to scale pixel values to the expected range [0, 1].

- If `I` is a grayscale or truecolor image with data type `uint8`, `uint16` or `int16`, then `im2double` rescales output pixel values to the range [0, 1].
- If `I` is an indexed image with data type `uint8` or `uint16`, then `im2double` adds an offset of `1` to the output pixel values.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`double` | `gpuArray` | `im2int16` | `im2single` | `im2uint16` | `im2uint8`

Topics

“Image Types”

Introduced before R2006a

im2frame

Convert image to movie frame

Syntax

```
F = im2frame(RGB)
F = im2frame(X,map)
F = im2frame(X)
```

Description

`F = im2frame(RGB)` converts the truecolor image, `RGB`, into a movie frame `F`.

`F = im2frame(X,map)` converts the indexed image, `X`, and the associated colormap, `map`, into a movie frame `F`.

`F = im2frame(X)` converts the indexed image, `X`, into a movie frame `F` using the current colormap.

Examples

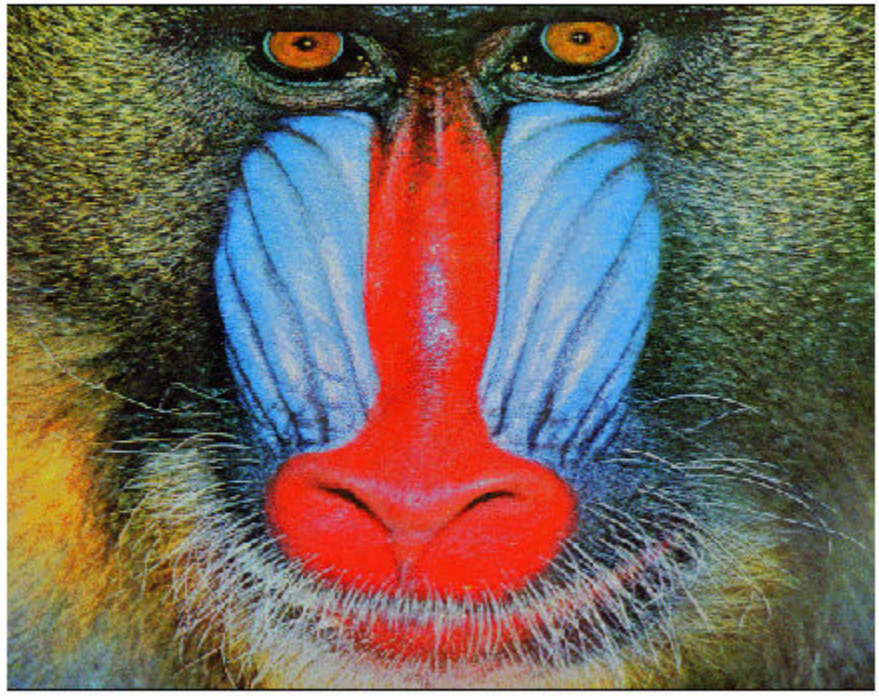
Convert Images to Movie Frames

Load an indexed image of a mandrill's face.

```
load mandrill
```

Display the image `X` using its associated colormap, `map`, which has 220 colors.

```
figure
image(X)
colormap(map)
axis off
```



Make a movie that shows the effect of reducing the number of colors of the image. There will be eight movie frames in total. Preallocate an array of structures to store the movie frames.

```
F(8) = struct('cdata',[],'colormap',[]);
```

In a loop, reduce the number of colors in the indexed image by using the `imapprox` function. Use the `im2frame` function to convert the images into frames of a movie. The first frame is the original image with all 220 colors. The second frame has 128 colors. Each successive frame has half the number of colors. The last frame has the minimum number of colors, 2.

```
for j = 1:8
    q = 2^(9-j);
```

```
    [Y,newmap] = imapprox(X,map,q,'nodither');
    F(j) = im2frame(Y,newmap);
end
```

To play the movie twice with a frame rate of three frames per second, use `movie(F,2,3)`.

Input Arguments

RGB — Truecolor image
m-by-*n*-by-3 numeric array

Truecolor image, specified as an *m*-by-*n*-by-3 numeric array. If you specify an image of data type `double`, then values must be in the range [0, 1].

Data Types: `double` | `uint8`

X — Indexed image
m-by-*n* matrix of integers

Indexed image, specified as an *m*-by-*n* matrix of integers.

Data Types: `double` | `uint8`

`map` — Colormap
c-by-3 numeric matrix

Colormap associated with indexed image `X`, specified as a *c*-by-3 numeric matrix with values in the range [0, 1]. Each row of `map` is a three-element RGB triplet that specifies the red, green, and blue components of a single color of the colormap.

Data Types: `double`

Output Arguments

F — Movie frame
structure

Movie frame, returned as a structure with two fields:

- `cdata` — The image data stored as an array of `uint8` values.
- `colormap` — The colormap. For truecolor (RGB) images, this field is empty.

See Also

`frame2im` | `getframe` | `movie`

Topics

"Image Types"

Introduced before R2006a

im2java

Convert image to Java image

Syntax

```
jimage = im2java(RGB)
jimage = im2java(I)
jimage = im2java(X,map)
```

Description

To work with a MATLAB image in the Java environment, you must convert the image from its MATLAB representation into an instance of the Java image class, `sun.awt.image.ToolkitImage`.

`jimage = im2java(RGB)` converts the truecolor (RGB) image `RGB` to an instance of the Java image class.

`jimage = im2java(I)` converts the grayscale (intensity) image `I` to an instance of the Java image class.

`jimage = im2java(X,map)` converts the indexed image `X`, with colormap `map`, to an instance of the Java image class.

Input Arguments

RGB — RGB image
m-by-*n*-by-3 numeric array

RGB image, specified as an *m*-by-*n*-by-3 numeric array. If you specify an image of data type `double`, then values must be in the range [0, 1].

Data Types: `double` | `uint8` | `uint16`

I — Grayscale image
m-by-*n* numeric matrix

Grayscale image, specified as an *m*-by-*n* numeric matrix. If you specify an image of data type `double`, then values must be in the range [0, 1].

Data Types: `double` | `uint8` | `uint16`

`X` — Indexed image
m-by-*n* matrix of nonnegative integers

Indexed image, specified as an *m*-by-*n* matrix of nonnegative integers.

- If you specify `X` as an array of data type `uint8` or `uint16`, then values must be in the range [0, *c*-1].
- If you specify `X` as an array of data type `double`, then values must be in the range [1, *c*].

Data Types: `double` | `uint8` | `uint16`

`map` — Colormap
c-by-3 matrix

Colormap associated with indexed image `X`, specified as a *c*-by-3 matrix with values in the range [0, 1]. Each row of `map` is a three-element RGB triplet that specifies the red, green, and blue components of a single color of the colormap. The colormap can have a maximum of 256 colors.

Data Types: `double`

Output Arguments

`jimage` — Java image
`sun.awt.image.ToolkitImage`

Java image, returned as a `sun.awt.image.ToolkitImage`.

Tips

- Java requires `uint8` data to create an instance of the Java image class, `sun.awt.image.ToolkitImage`. If the input image is of class `double` or `uint16`, then `im2java` makes an equivalent image of class `uint8`, rescaling or offsetting the

data as necessary, and then converts this `uint8` representation to an instance of the Java image class. For more information, see “8-Bit and 16-Bit Images”.

See Also

`im2java2d`

Introduced before R2006a

imag

Imaginary part of complex number

Syntax

```
Y = imag(Z)
```

Description

`Y = imag(Z)` returns the imaginary part of each element in array `Z`.

Examples

Imaginary Part of Complex Number

Find the imaginary part of the complex number `Z`.

```
Z = 2+3i;
Y = imag(Z)
```

```
Y = 3
```

Imaginary Part of Vector of Complex Values

Find the imaginary part of each element in vector `Z`. The `imag` function acts on `Z` element-wise.

```
Z = [0.5i 1+3i -2.2];
Y = imag(Z)
```

```
Y = 1×3
```

```
    0.5000    3.0000         0
```

Input Arguments

Z — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. `imag` operates element-wise when `Z` is nonscalar.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`abs` | `complex` | `conj` | `i` | `j` | `real`

Introduced before R2006a

image

Display image from array

Syntax

```
image(C)
image(x,y,C)
image('CData',C)
image('XData',x,'YData',y,'CData',C)

image( ___ ,Name,Value)
image(ax, ___ )

im = image( ___ )
```

Description

`image(C)` displays the data in array `C` as an image. Each element of `C` specifies the color for 1 pixel of the image. The resulting image is an `m`-by-`n` grid of pixels where `m` is the number of rows and `n` is the number of columns in `C`. The row and column indices of the elements determine the centers of the corresponding pixels.

`image(x,y,C)` specifies the image location. Use `x` and `y` to specify the locations of the corners corresponding to `C(1,1)` and `C(m,n)`. To specify both corners, set `x` and `y` as two-element vectors. To specify the first corner and let `image` determine the other, set `x` and `y` as scalar values. The image is stretched and oriented as applicable.

`image('CData',C)` adds the image to the current axes without replacing existing plots. This syntax is the low-level version of `image(C)`. For more information, see “High-Level Versus Low-Level Version of Image” on page 1-7339.

`image('XData',x,'YData',y,'CData',C)` specifies the image location. This syntax is the low-level version of `image(x,y,C)`.

`image( ___ ,Name,Value)` specifies image properties using one or more name-value pair arguments. You can specify image properties with any of the input argument combinations in the previous syntaxes.

`image(ax, ___ )` creates the image in the axes specified by `ax` instead of in the current axes (`gca`). The option `ax` can precede any of the input argument combinations in the previous syntaxes.

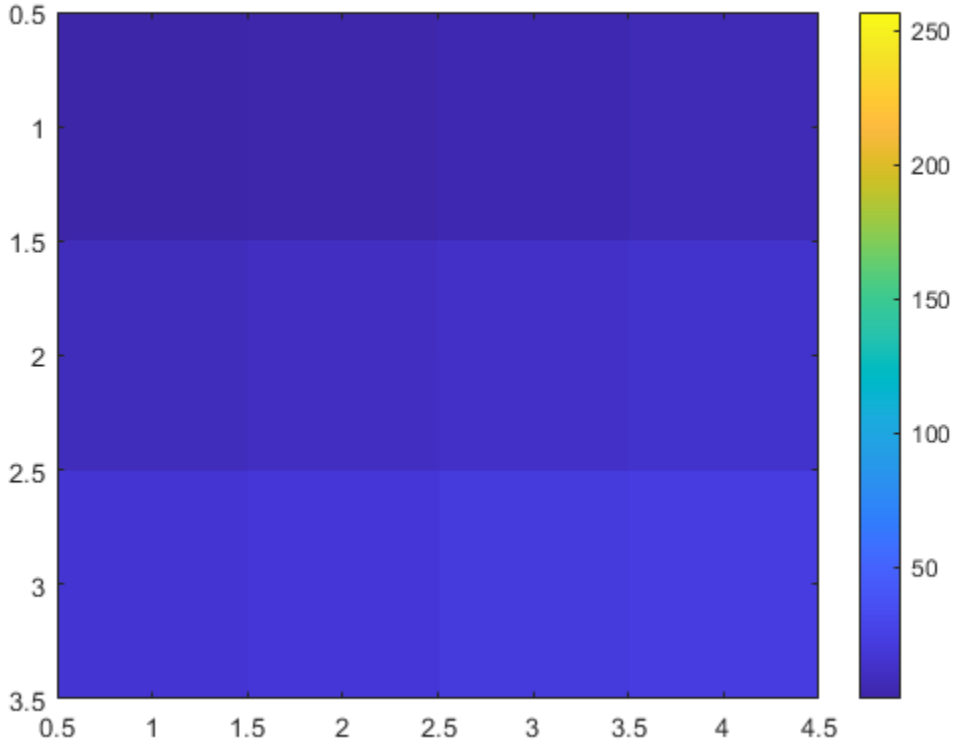
`im = image( ___ )` returns the `Image` object created. Use `im` to set properties of the image after it is created. You can specify this output with any of the input argument combinations in the previous syntaxes. For a list of image properties and descriptions, see Image.

Examples

Display Image of Matrix Data

Create matrix `C`. Display an image of the data in `C`. Add a colorbar to the graph to show the current colormap.

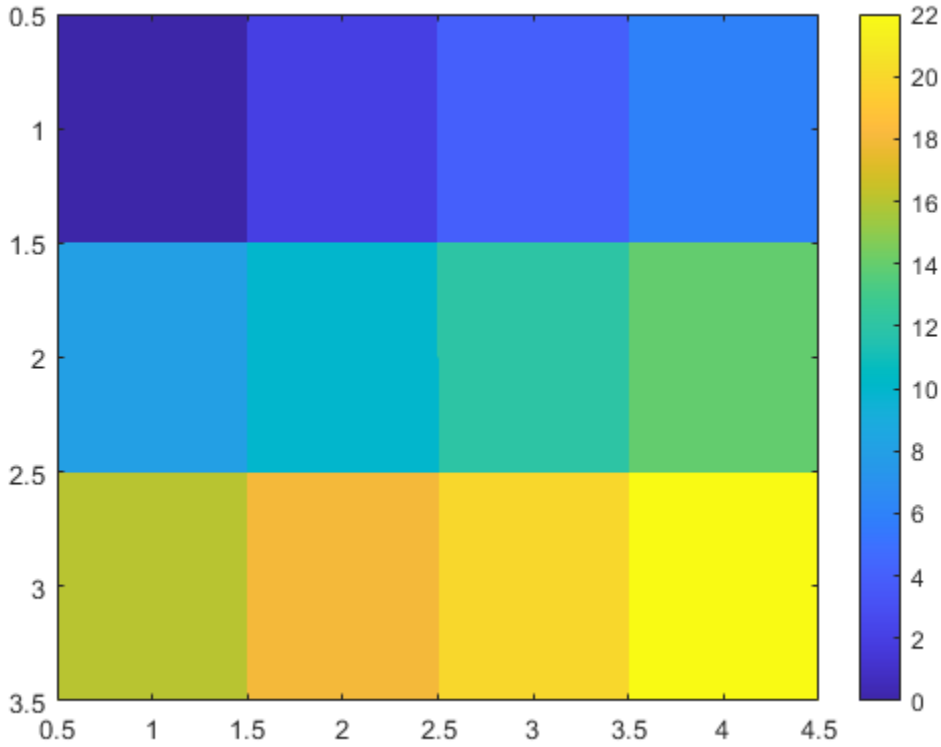
```
C = [0 2 4 6; 8 10 12 14; 16 18 20 22];
image(C)
colorbar
```



By default, the `CDataMapping` property for the image is set to `'direct'` so `image` interprets values in `C` as indices into the colormap. For example, the bottom right pixel corresponding to the last element in `C`, 22, uses the 22nd color of the colormap.

Scale the values to the full range of the current colormap by setting the `CDataMapping` property to `'scaled'` when creating the image.

```
image(C,'CDataMapping','scaled')
colorbar
```

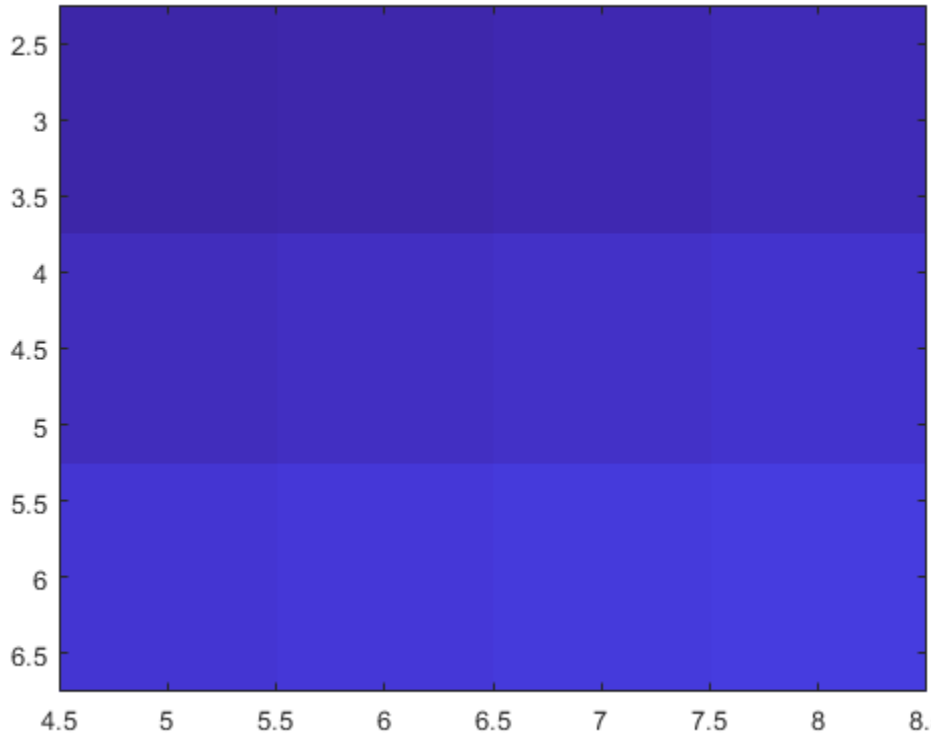


Alternatively, you can use the `imagesc` function to scale the values instead of using `image(C,'CDataMapping','scaled')`. For example, use `imagesc(C)`.

Control Image Placement

Place the image so that it lies between 5 and 8 on the *x*-axis and between 3 and 6 on the *y*-axis.

```
x = [5 8];
y = [3 6];
C = [0 2 4 6; 8 10 12 14; 16 18 20 22];
image(x,y,C)
```



Notice that the pixel corresponding to C(1,1) is centered over the point (5,3). The pixel corresponding to C(3,4) is centered over the point (8,6). `image` positions and orients the rest of the image between those two points.

Display Image of 3-D Array of True Colors

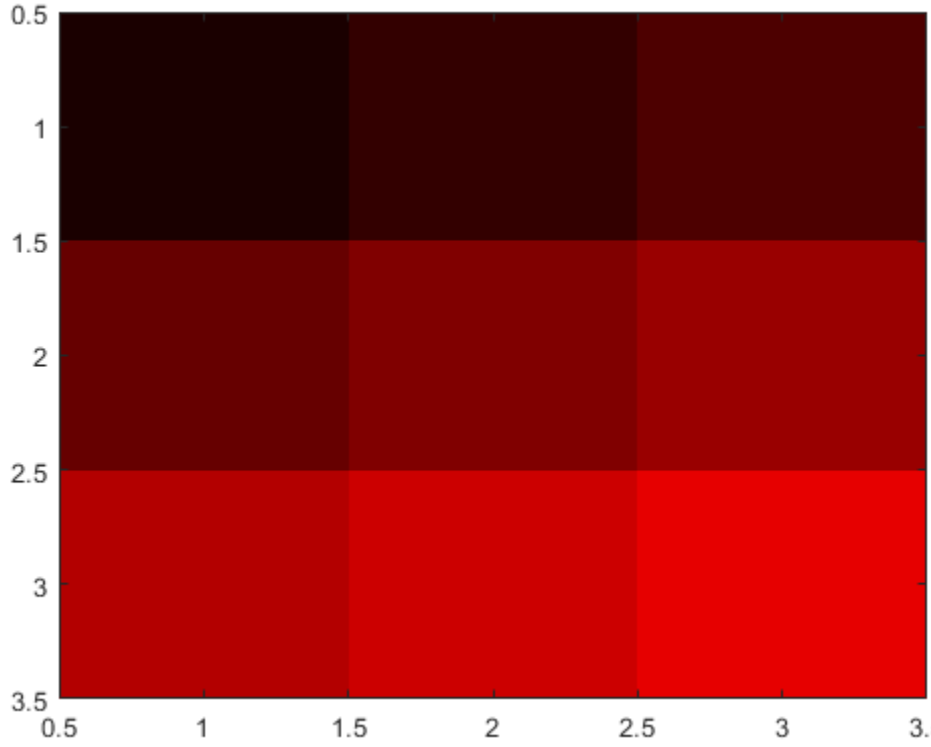
Create `C` as a 3-D array of true colors. Use only red colors by setting the last two pages of the array to zeros.

```
C = zeros(3,3,3);
C(:,:,1) = [.1 .2 .3; .4 .5 .6; .7 .8 .9]
```

```
C =
C(:,:,1) =

    0.1000    0.2000    0.3000
    0.4000    0.5000    0.6000
    0.7000    0.8000    0.9000


C(:,:,2) =

     0     0     0
     0     0     0
     0     0     0


C(:,:,3) =

     0     0     0
     0     0     0
     0     0     0
```

Display an image of the data in `C`.

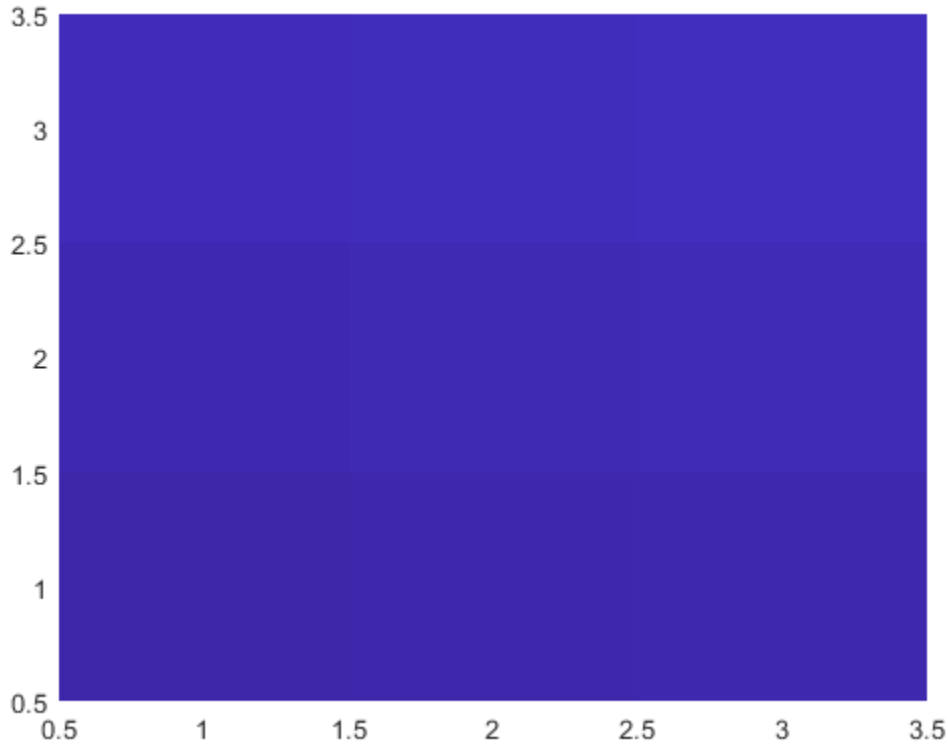
```
image(C)
```



Modify Image After Creation

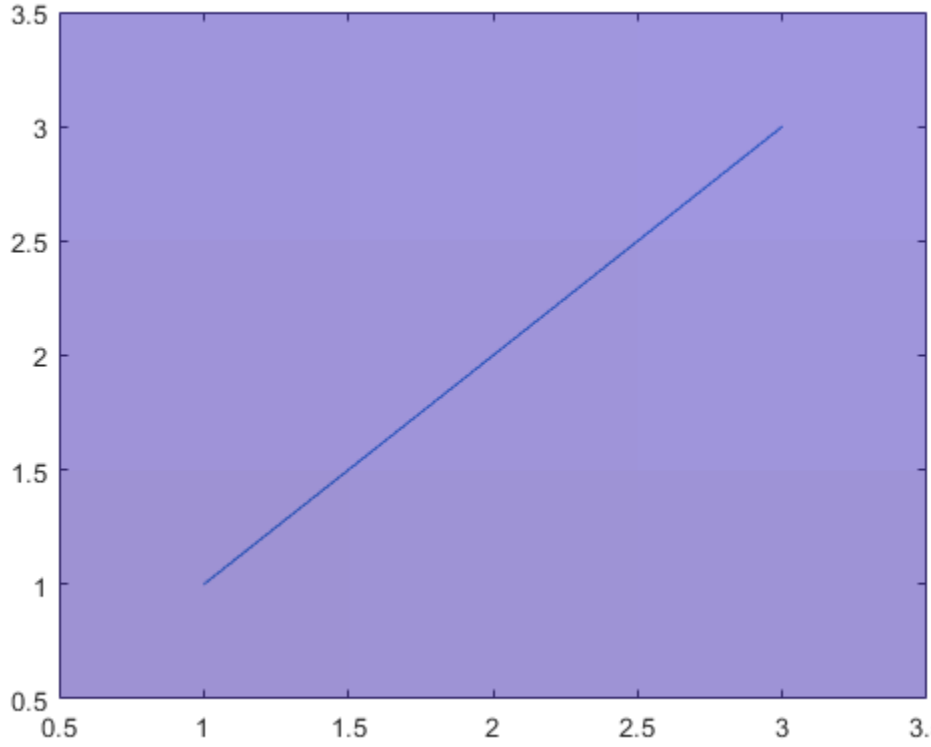
Plot a line, and then create an image on top of the line. Return the image object.

```
plot(1:3)
hold on
C = [1 2 3; 4 5 6; 7 8 9];
im = image(C);
```



Make the image semitransparent so that the line shows through the image.

```
im.AlphaData = 0.5;
```



Read and Display JPEG Image File

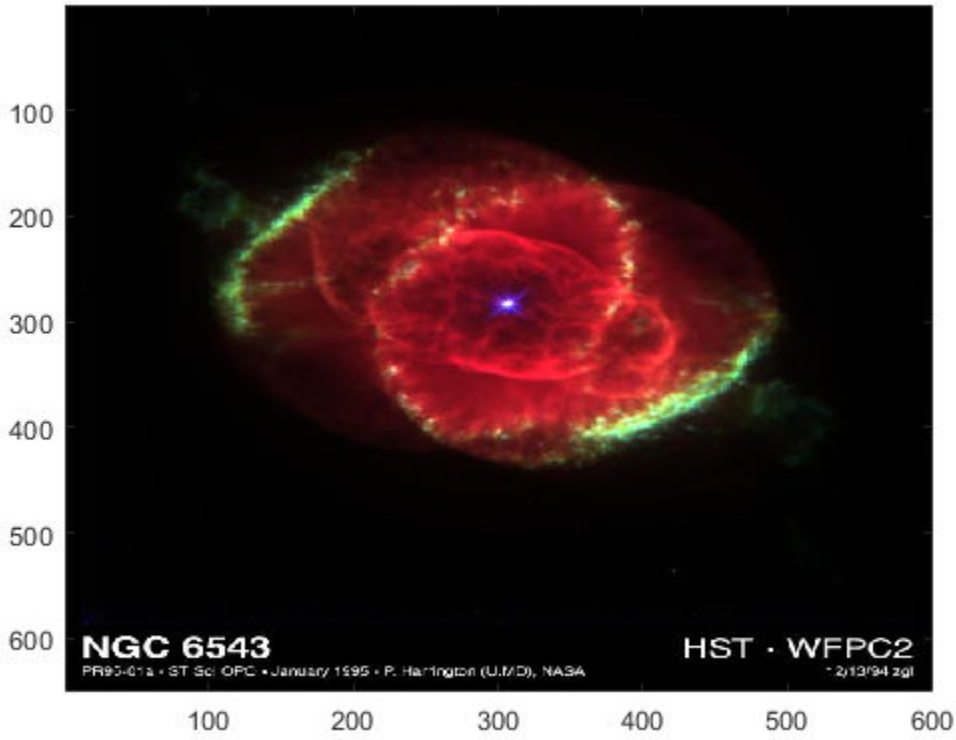
Read a JPEG image file.

```
C = imread('ngc6543a.jpg');
```

`imread` returns a 650-by-600-by-3 array, `C`.

Display the image.

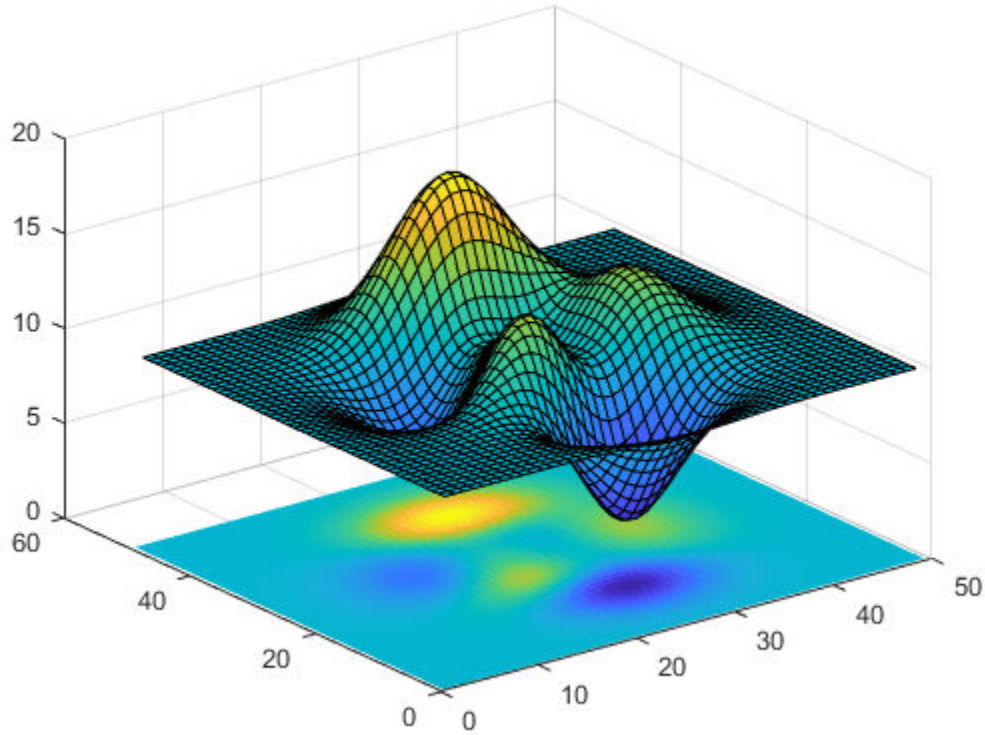
```
image(C)
```



Add Image to Axes in 3-D View

Create a surface plot. Then, add an image under the surface. `image` displays the image in the *xy*-plane.

```
Z = 10 + peaks;
surf(Z)
hold on
image(Z,'CDataMapping','scaled')
```



Input Arguments

C — Image color data

vector or matrix | 3-D array of RGB triplets

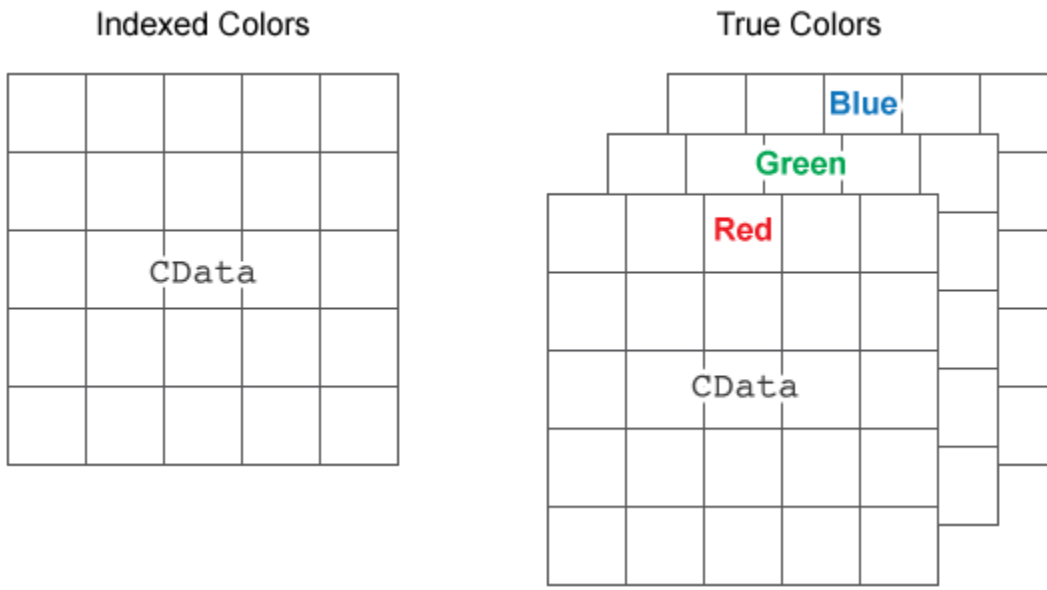
Image color data, specified in one of these forms:

- Vector or matrix — This format defines indexed image data. Each element of `C` defines a color for 1 pixel of the image. For example, `C = [1 2 3; 4 5 6; 7 8 9];`. The elements of `C` map to colors in the colormap of the associated axes. The `CDataMapping` property controls the mapping method.

- 3-D array of RGB triplets — This format defines true color image data using RGB triplet values. Each RGB triplet defines a color for 1 pixel of the image. An RGB triplet is a three-element vector that specifies the intensities of the red, green, and blue components of the color. The first page of the 3-D array contains the red components, the second page contains the green components, and the third page contains the blue components. Since the image uses true colors instead of colormap colors, the `CDataMapping` property has no effect.

  - If `C` is of type `double`, then an RGB triplet value of `[0 0 0]` corresponds to black and `[1 1 1]` corresponds to white.
  - If `C` is an integer type, then the image uses the full range of data to determine the color. For example, if `C` is of type `uint8`, then `[0 0 0]` corresponds to black and `[255 255 255]` corresponds to white. If `CData` is of type `int8`, then `[-128 -128 -128]` corresponds to black and `[127 127 127]` corresponds to white.
  - If `C` is of type `logical`, then `[0 0 0]` corresponds to black and `[1 1 1]` corresponds to white.

This illustration shows the relative dimensions of `C` for the two color models.



The behavior of `NaN` elements is not defined.

To use the low-level version of the `image` function instead, set the `CData` property as a name-value pair. For example, `image('CData',C)`.

Converting Between Data Types

To convert indexed image data from an integer type to type `double`, add 1. For example, if `X8` is indexed image data of type `uint8`, convert it to type `double` using:

```
X64 = double(X8) + 1;
```

To convert indexed image data from type `double` to an integer type, subtract 1 and use `round` to ensure that all the values are integers. For example, if `X64` is indexed image data of type `double`, convert it to `uint8` using:

```
X8 = uint8(round(X64 - 1));
```

To convert true color image data from an integer type to type `double`, rescale the data. For example, if `RGB8` is true color image data of type `uint8`, convert it to `double` using:

```
RGB64 = double(RGB8)/255;
```

To convert true color image data from type `double` to an integer type, rescale the data and use `round` to ensure that all the values are integers. For example, if `RGB64` is image data of type `double`, convert it to `uint8` using:

```
RGB8 = uint8(round(RGB64*255));
```

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

x — Placement along x-axis

`[1 size(C,2)]` (default) | two-element vector | scalar

Placement along the *x*-axis, specified in one of these forms:

- Two-element vector — Use the first element as the location for the center of `C(1,1)` and the second element as the location for the center of `C(m,n)`, where `[m,n] = size(C)`. If `C` is a 3-D array, then `m` and `n` are the first two dimensions. Evenly distribute the centers of the remaining elements of `C` between those two points.

  The width of each pixel is determined by the expression:

  ```
  (x(2)-x(1))/(size(C,2)-1)
  ```

  If `x(1) > x(2)`, then the image is flipped left-right.

- Scalar — Center `C(1,1)` at this location and each following element one unit apart.

To use the low-level version of the `image` function instead, set the `XData` property as a name-value pair. For example, `image('XData',x,'YData',y,'CData',C)`.

You cannot interactively pan or zoom outside the x-axis limits or y-axis limits of an image, unless the limits are already set outside the bounds of the image. If the limits are already outside the bounds, there is no such restriction. If other objects (such as a line) occupy the axes and extend beyond the bounds of the image, you can pan or zoom to the bounds of the other objects, but no further.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

y — Placement along y-axis
`[1 size(C,1)]` (default) | two-element vector | scalar

Placement along *y*-axis, specified in one of these forms:

- Two-element vector — Use the first element as the location for the center of `C(1,1)` and the second element as the location for the center of `C(m,n)`, where `[m,n] = size(C)`. If `C` is a 3-D array, then `m` and `n` are the first two dimensions. Evenly distribute the centers of the remaining elements of `C` between those two points.

  The height of each pixel is determined by the expression:

  ```
  (y(2)-y(1))/(size(C,1)-1)
  ```

  If `y(1) > y(2)`, then the image is flipped up-down.
- Scalar — Center `C(1,1)` at this location and each following element one unit apart.

To use the low-level version of the `image` function instead, set the `YData` property as a name-value pair. For example, `image('XData',x,'YData',y,'CData',C)`.

You cannot interactively pan or zoom outside the x-axis limits or y-axis limits of an image, unless the limits are already set outside the bounds of the image. If the limits are already outside the bounds, there is no such restriction. If other objects (such as a line) occupy the axes and extend beyond the bounds of the image, you can pan or zoom to the bounds of the other objects, but no further.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

ax — Axes object
`Axes` object

`Axes` object. If you do not specify an `Axes` object, then `image` uses the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `image([1 2 3],'AlphaData',0.5)` displays a semitransparent image.

The properties listed here are a subset of image properties. For a complete list, see Image.

`CDataMapping` — Color data mapping method
`'direct'` (default) | `'scaled'`

Color data mapping method, specified as `'direct'` or `'scaled'`. Use this property to control the mapping of color data values in `CData` into the colormap. `CData` must be a vector or a matrix defining indexed colors. This property has no effect if `CData` is a 3-D array defining true colors.

The methods have these effects:

- `'direct'` — Interpret the values as indices into the current colormap. Values with a decimal portion are fixed to the nearest lower integer.
  - If the values are of type `double` or `single`, then values of `1` or less map to the first color in the colormap. Values equal to or greater than the length of the colormap map to the last color in the colormap.
  - If the values are of type `uint8`, `uint16`, `uint32`, `uint64` , `int8`, `int16`, `int32`, or `int64`, then values of `0` or less map to the first color in the colormap. Values equal to or greater than the length of the colormap map to the last color in the colormap (or up to the range limits of the type).
  - If the values are of type `logical`, then values of `0` map to the first color in the colormap and values of `1` map to the second color in the colormap.
- `'scaled'` — Scale the values to range between the minimum and maximum color limits. The `CLim` property of the axes contains the color limits.

`AlphaData` — Transparency data
`1` (default) | scalar | array the same size as `CData`

Transparency data, specified in one of these forms:

- Scalar — Use a consistent transparency across the entire image.
- Array the same size as `CData` — Use a different transparency value for each image element.

The `AlphaDataMapping` property controls how MATLAB interprets the alpha data transparency values.

Example: `0.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`AlphaDataMapping` — Interpretation of `AlphaData` values
`'none'` (default) | `'scaled'` | `'direct'`

Interpretation of `AlphaData` values, specified as one of these values:

- `'none'` — Interpret the values as transparency values. A value of 1 or greater is completely opaque, a value of 0 or less is completely transparent, and a value between 0 and 1 is semitransparent.
- `'scaled'` — Map the values into the figure's alphamap. The minimum and maximum alpha limits of the axes determine the alpha data values that map to the first and last elements in the alphamap, respectively. For example, if the alpha limits are `[3 5]`, then alpha data values less than or equal to `3` map to the first element in the alphamap. Alpha data values greater than or equal to `5` map to the last element in the alphamap. The `ALim` property of the axes contains the alpha limits. The `Alphamap` property of the figure contains the alphamap.
- `'direct'` — Interpret the values as indices into the figure's alphamap. Values with a decimal portion are fixed to the nearest lower integer:
  - If the values are of type `double` or `single`, then values of 1 or less map to the first element in the alphamap. Values equal to or greater than the length of the alphamap map to the last element in the alphamap.
  - If the values are of type integer, then values of 0 or less map to the first element in the alphamap. Values equal to or greater than the length of the alphamap map to the last element in the alphamap (or up to the range limits of the type). The integer types are `uint8`, `uint16`, `uint32`, `uint64` , `int8`, `int16`, `int32`, and `int64`.

- If the values are of type `logical`, then values of 0 map to the first element in the alphamap and values of 1 map to the second element in the alphamap.

Output Arguments

`im` — `Image` object
`Image` object

`Image` object, returned as a scalar. Use `im` to set properties of the image after it is created. For a list, see Image.

More About

High-Level Versus Low-Level Version of Image

The `image` function has two versions, the high-level version and the low-level version. If you use `image` with `'CData'` as an input argument, then you are using the low-level version. Otherwise, you are using the high-level version.

The high-level version of `image` calls `newplot` before plotting and sets these axes properties:

- `Layer` to `'top'`. The image is shown in front of any tick marks or grid lines.
- `YDir` to `'reverse'`. Values along the *y*-axis increase from top to bottom. To decrease the values from top to bottom, set `YDir` to `'normal'`. This setting reverses both the *y*-axis and the image.
- `View` to `[0 90]`.

The low-level version of the `image` function does not call `newplot` and does not set these axes properties.

Tips

- To read image data into MATLAB from graphics files in various standard formats, such as TIFF, use `imread`. To write MATLAB image data to graphics files, use `imwrite`. The

`imread` and `imwrite` functions support a variety of graphics file formats and compression schemes.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions

`colormap` | `imagesc` | `imfinfo` | `imread` | `imshow` | `imwrite`

Properties

Image

Introduced before R2006a

Image Properties

Control image component appearance and behavior

Description

Images are UI components that allow you to display a picture, such as an icon or logo in your app. `Image` properties control the appearance and behavior of an image. Use dot notation to refer to a specific object and property.

```
f = uifigure;
im = uiimage(f);
im.ImageSource = 'peppers.png';
```

Properties

Image

`ImageSource` — Image source or file

`' '` (default) | file path | `m`-by-`n`-by-3 truecolor image array

Image source or file, specified as a file path or an `m`-by-`n`-by-3 truecolor image array. Supported image formats include JPEG, PNG, GIF, SVG, or `m`-by-`n`-by-3 truecolor image array.

For more information on truecolor image arrays, see “Image Types”.

Example: `im = uiimage('ImageSource','peppers.png');`

Example: `im.ImageSource = 'C:\TEMP\ngc6543a.jpg';`

`HorizontalAlignment` — Horizontal alignment of rendered image

`'center'` (default) | `'left'` | `'right'`

Horizontal alignment of the rendered image within the image component area, specified as `'center'`, `'left'`, or `'right'`. The horizontal alignment is relative to the inside borders of the image component. Setting this property has no effect when the `ScaleMethod` property value is set to `'stretch'`.

For example, the table shows rendered images for each `HorizontalAlignment` value, where the `ScaleMethod` property value is set to `'none'` and the `VerticalAlignment` property has the default value, `'center'`. You can implement many other combinations that are not shown here.

Horizontal Alignment Value	Description	Rendered Image
`'center'`	Left and right edges of the image are equally spaced from the inside-left and inside-right borders of the image component, respectively.	
`'left'`	Left edge of the image aligns with the inside-left border of the image component.	
`'right'`	Right edge of the image aligns with the inside-right border of the image component.	

`VerticalAlignment` — Vertical alignment of rendered image
`'center'` (default) | `'top'` | `'bottom'`

Vertical alignment of the rendered image within the image component area, specified as `'center'`, `'left'`, or `'right'`. The vertical alignment is relative to the inside borders of the image component. Setting this property has no effect when the `ScaleMethod` property value is set to `'stretch'`.

For example, the table shows rendered images for each `VerticalAlignment` value, where the `ScaleMethod` property value is set to `'none'` and the `HorizontalAlignment` property has the default value, `'center'`. You can implement many other combinations that are not shown here.

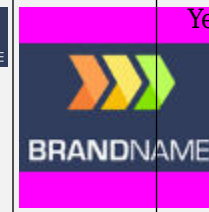
Vertical Alignment Value	Description	Rendered Image
`'center'`	Top and bottom edges of the image are equally spaced from the inside-top and inside-bottom borders of the image component, respectively.	
`'top'`	Top edge of the image aligns with the inside-top border of the image component.	
`'bottom'`	Bottom edge of the image aligns with the inside-bottom border of the image component.	

`ScaleMethod` — Image scaling method

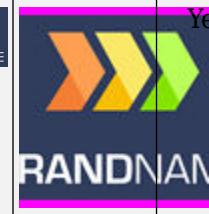
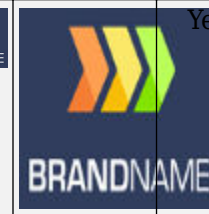
`'fit'` (default) | `'fill'` | `'none'` | `'scaledown'` | `'scaleup'` | `'stretch'`

Image scaling method, specified as one of the values listed in the table. Use this name-value pair argument to specify how you want your image to render within the component area.

The table also demonstrates each scale method with an example image. In the rendered image examples, the `BackgroundColor` property of the image component has been set to `'magenta'`. The scaling behavior of SVG image files may vary based on how the file is defined.

Value	Description	Example		Scales Up	Scales Down	Maintains Aspect Ratio	Clips Image
		Original Image	Rendered Image				
`'fit'`	Scales in any direction to display the image within the component area, and maintains aspect ratio without clipping.			Yes	Yes	Yes	No
`'fill'`	Scales in any direction to fill the component area, maintaining aspect ratio and clipping if necessary.			Yes	Yes	Yes	Yes
`'none'`	Uses the actual size of the image and maintains aspect ratio. If the component area is smaller than the image, the image is clipped.			No	No	Yes	Yes

Value	Description	Example		Scales Up	Scales Down	Maintains Aspect Ratio	Clips Image
		Original Image	Rendered Image				
`'scaledown'`	Scales down and maintains aspect ratio without clipping. If the original image is larger than the component area, the image scales down and renders as if the `ScaleMethod` was set to `'fit'`. If the original image is smaller than the component area, the image does not scale down and renders as if the `ScaleMethod` was set to `'none'`.			No	Yes	Yes	No

Value	Description	Example		Scales Up	Scales Down	Maintains Aspect Ratio	Clips Image
		Original Image	Rendered Image				
`'scaleup'`	Scales up and maintains aspect ratio with clipping. If the original image is smaller than the component area, the image scales up and renders as if the `ScaleMethod` was set to `'fit'`. If the original image is larger than the component area, the image does not scale up and renders as if the `ScaleMethod` was set to `'none'`.			Yes	No	Yes	Yes
`'stretch'`	Scales in any direction to fill the component area, without maintaining the aspect ratio and without clipping.			Yes	Yes	No	No

Color

BackgroundColor — Background Color

`'none'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Interactivity

`Visible` — Image visibility

`'on'` (default) | `'off'`

Image visibility, specified as `'on'` or `'off'`. The `Visible` property determines whether the image is displayed on the screen. If the `Visible` property is set to `'off'`, the entire image component is hidden, but you can still set and access its properties.

To make your app start faster, set the `Visible` property to `'off'` for all components that do not need to appear at startup.

`Enable` — Operational state of image

`'on'` (default) | `'off'`

Operational state of the image, specified as `'on'` or `'off'`.

- `'on'` — The image appears normal, and the app user can click the image, which triggers any `ImageClickedFcn` callback associated with it.

- `'off'` — The image appears dimmed, indicating that the app user cannot click the image, and that associated callbacks do not execute.

`Tooltip` — Tooltip

`' '` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size of image component

`[100 100 100 100]` (default) | `[left bottom width height]`

Location and size of image component relative to the parent, specified as a four element vector of the form `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the image component
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the image component
`width`	Distance between the right and left outer edges of the image component
`height`	Distance between the top and bottom outer edges of the image component

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

All measurements are in pixel units.

`InnerPosition` — Inner location and size of image component
`[100 100 100 100]` (default) | `[left bottom width height]`

Inner location and size of image component, specified as a four element vector of the form `[left bottom width height]`. All measurements are in pixel units. This property value is identical to the `Position` property.

`OuterPosition` — Outer location and size of image component
`[100 100 100 100]` (default) | `[left bottom width height]`

This property is read-only.

Outer location and size of image component, returned as a four element vector of the form `[left bottom width height]`. All measurements are in pixel units. This property value is identical to the `Position` property.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places an image component in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
im = uiimage(g);
im.ImageSource = 'peppers.png';
im.ScaleMethod = 'fill';
im.Layout.Row = 3;
im.Layout.Column = 2;
```

To make the image span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this image spans columns 2 through 3.

```
im.Layout.Column = [2 3];
```

Callbacks

`ImageClickedFcn` — Image clicked callback

' ' (default) | function handle | cell array | character vector

Image clicked callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when the user clicks the image in the app.

This callback function can access specific information about the user's interaction with the image. MATLAB passes this information in an `ImageClickedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object's properties using dot notation. For example, `event.Source` returns the `Image` object that the user is interacting with to trigger the callback. The `ImageClickedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ImageClickedData` object.

Property	Value
`EventName`	`'ImageClicked'`
`Source`	Component executing the callback

For more information about writing callbacks, see “Write Callbacks in App Designer”.

`CreateFcn` — Creation function

' ' (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.

- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object
`'uiimage'`

This property is read-only.

Type of graphics object, returned as `'uiimage'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `image` | `imshow` | `uifigure` | `uiimage`

Topics

“Share Data Within App Designer Apps”
“Write Callbacks in App Designer”

Introduced in R2019a

Image Properties

Image appearance and behavior

Description

`Image` properties control the appearance and behavior of `Image` objects. By changing property values, you can modify certain aspects of the image.

Starting in R2014b, you can use dot notation to query and set properties.

```
im = image(rand(20));
C = im.CData;
im.CDataMapping = 'scaled';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Color and Transparency

`CData` — Image color data

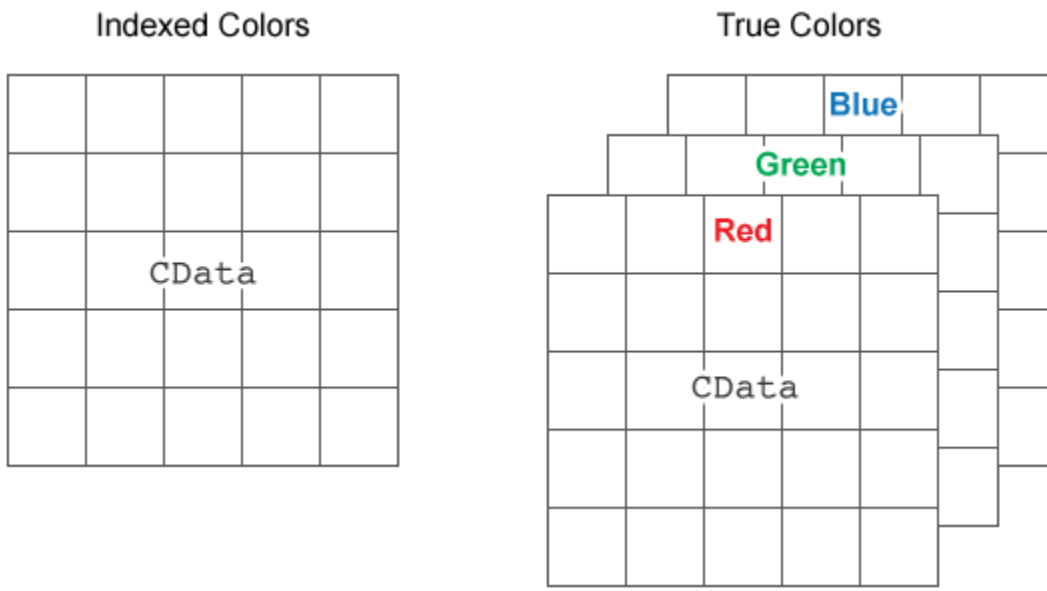
64-by-64 array (default) | vector or matrix | 3-D array of RGB triplets

Image color data, specified in one of these forms:

- Vector or matrix — This format defines indexed image data. Each element defines a color for one pixel of the image. The elements map to colors in the colormap. The `CDataMapping` property controls the mapping method.
- 3-D array of RGB triplets — This format defines true color image data using RGB triplet values. Each RGB triplet defines a color for one pixel of the image. An RGB triplet is a three-element vector that specifies the intensities of the red, green, and blue components of the color. The first page of the 3-D array contains the red components, the second page contains the green components, and the third page contains the blue components. Since the image uses true colors instead of colormap colors, the `CDataMapping` property has no effect.

- If `CData` is of type `double`, then an RGB triplet value of `[0 0 0]` corresponds to black and `[1 1 1]` corresponds to white.
- If `CData` is an integer type, then the image uses the full range of data to determine the color. For example, if `CData` is of type `uint8`, then `[0 0 0]` corresponds to black and `[255 255 255]` corresponds to white. If `CData` is of type `int8`, then `[-128 -128 -128]` corresponds to black and `[127 127 127]` corresponds to white.
- If `CData` is of type `logical`, then `[0 0 0]` corresponds to black and `[1 1 1]` corresponds to white.

This illustration shows the relative dimensions of `CData` for the two color models.



The behavior of `NaN` elements is not defined.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`CDataMapping` — Color data mapping method
`'direct'` (default) | `'scaled'`

Color data mapping method, specified as `'direct'` or `'scaled'`. Use this property to control the mapping of color data values in `CData` into the colormap. `CData` must be a vector or a matrix defining indexed colors. This property has no effect if `CData` is a 3-D array defining true colors.

The methods have these effects:

- `'direct'` — Interpret the values as indices into the current colormap. Values with a decimal portion are fixed to the nearest lower integer.
  - If the values are of type `double` or `single`, then values of `1` or less map to the first color in the colormap. Values equal to or greater than the length of the colormap map to the last color in the colormap.
  - If the values are of type `uint8`, `uint16`, `uint32`, `uint64` , `int8`, `int16`, `int32`, or `int64`, then values of `0` or less map to the first color in the colormap. Values equal to or greater than the length of the colormap map to the last color in the colormap (or up to the range limits of the type).
  - If the values are of type `logical`, then values of `0` map to the first color in the colormap and values of `1` map to the second color in the colormap.
- `'scaled'` — Scale the values to range between the minimum and maximum color limits. The `CLim` property of the axes contains the color limits.

`AlphaData` — Transparency data

`1` (default) | scalar | array the same size as `CData`

Transparency data, specified in one of these forms:

- Scalar — Use a consistent transparency across the entire image.
- Array the same size as `CData` — Use a different transparency value for each image element.

The `AlphaDataMapping` property controls how MATLAB interprets the alpha data transparency values.

Example: `0.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`AlphaDataMapping` — Interpretation of `AlphaData` values

`'none'` (default) | `'scaled'` | `'direct'`

Interpretation of `AlphaData` values, specified as one of these values:

- `'none'` — Interpret the values as transparency values. A value of 1 or greater is completely opaque, a value of 0 or less is completely transparent, and a value between 0 and 1 is semitransparent.
- `'scaled'` — Map the values into the figure's alphamap. The minimum and maximum alpha limits of the axes determine the alpha data values that map to the first and last elements in the alphamap, respectively. For example, if the alpha limits are `[3 5]`, then alpha data values less than or equal to `3` map to the first element in the alphamap. Alpha data values greater than or equal to `5` map to the last element in the alphamap. The `ALim` property of the axes contains the alpha limits. The `Alphamap` property of the figure contains the alphamap.
- `'direct'` — Interpret the values as indices into the figure's alphamap. Values with a decimal portion are fixed to the nearest lower integer:
  - If the values are of type `double` or `single`, then values of 1 or less map to the first element in the alphamap. Values equal to or greater than the length of the alphamap map to the last element in the alphamap.
  - If the values are of type integer, then values of 0 or less map to the first element in the alphamap. Values equal to or greater than the length of the alphamap map to the last element in the alphamap (or up to the range limits of the type). The integer types are `uint8`, `uint16`, `uint32`, `uint64` , `int8`, `int16`, `int32`, and `int64`.
  - If the values are of type `logical`, then values of 0 map to the first element in the alphamap and values of 1 map to the second element in the alphamap.

`Interpolation` — Interpolation method
`'nearest'` (default) | `'bilinear'`

Interpolation method, specified as `'nearest'` or `'bilinear'`. MATLAB uses interpolation to display a scaled or rotated version of the image on your screen. The value you choose does not affect the image array. However, MATLAB saves the interpolated visual representation of the image if you save the figure or export the contents of the axes.

Choose an interpolation method based on your image content and the effect you want to achieve:

- `'nearest'` — This method is best when there are a small number of pixel values that represent distinct categories, or when you want to see individual pixels in a highly zoomed-in view. The value of a pixel located at (x, y) is the value of the pixel that is closest to (x, y) in the original image.

- `'bilinear'` — This method is best in almost all other situations. The value of a pixel located at (x, y) is a weighted average of the surrounding pixels in the original image. To minimize display artifacts, additional smoothing is applied when you shrink the image. If your MATLAB session or figure is using the Painters renderer, then there is no additional smoothing when the image shrinks. To determine the renderer, call the `rendererinfo` function.

Position

`XData` — Placement along x-axis
`[1 size(CData,2)]` (default) | two-element vector | scalar

Placement along the x-axis, specified in one of these forms:

- Two-element vector — Use the first element as the location for the center of `CData(1,1)` and the second element as the location for the center of `CData(m,n)`, where `[m,n] = size(CData)`. Evenly distribute the centers of the remaining `CData` elements between those two points.

  The width of each pixel is determined by the expression:

  ```
  (XData(2)-XData(1))/(size(CData,2)-1)
  ```

  If `XData(1)` > `XData(2)`, then the image is flipped left-right.
- Scalar — Center `CData(1,1)` at this location and each following element one unit apart.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`YData` — Placement along y-axis
`[1 size(CData,1)]` (default) | two-element vector | scalar

Placement along y-axis, specified in one of these forms:

- Two-element vector — Use the first element as the location for the center of `CData(1,1)` and the second element as the location for the center of `CData(m,n)`, where `[m,n] = size(CData)`. Evenly distribute the centers of the remaining `CData` elements between those two points.

  The height of each pixel is determined by the expression:

  ```
  (YData(2)-YData(1))/(size(CData,1)-1)
  ```

If `YData(1) > YData(2)`, then the image is flipped up-down.

- Scalar — Center `CData(1,1)` at this location and each following element one unit apart.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`DataTipTemplate` — Data tip content

`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

This property applies only to images with pinned data tips.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

`Clipping` — Clipping of object to axes limits

`'on'` (default) | `'off'`

Clipping of the object to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the object that are outside the axes limits.
- `'off'` — Display the entire object, even if parts of it appear outside the axes limits. Parts of the object might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the object so that it is larger than the original plot.

The `Clipping` property of the axes that contains the object must be set to `'on'`. Otherwise, this property has no effect. For more information about the clipping behavior, see the `Clipping` property of the axes.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see "Callback Definition".

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you

do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks only when visible. The `Visible` property must be set to `'on'`. The `HitTest` property determines if the `Image` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `Image` object passes the click to the object behind it in the current view of the figure window. The `HitTest` property of the `Image` object has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `Image` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `Image` object that has one of these:
  - `HitTest` property set to `'on'`

- `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

---

Note The `PickableParts` property determines if the `Image` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent

`Axes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `Group`, or `Transform` object.

`Children` — Children

empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the chart object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object
`'image'`

This property is read-only.

Type of graphics object, returned as `'image'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[ ]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`image` | `imagesc`

Topics

“Access Property Values”
“Graphics Object Properties”

Introduced before R2006a

imagesc

Display image with scaled colors

Syntax

```
imagesc(C)
imagesc(x,y,C)
imagesc('CData',C)
imagesc('XData',x,'YData',y,'CData',C)

imagesc( ___ ,Name,Value)
imagesc( ___ ,clims)
imagesc(ax, ___ )

im = imagesc( ___ )
```

Description

`imagesc(C)` displays the data in array `C` as an image that uses the full range of colors in the colormap. Each element of `C` specifies the color for one pixel of the image. The resulting image is an `m`-by-`n` grid of pixels where `m` is the number of rows and `n` is the number of columns in `C`. The row and column indices of the elements determine the centers of the corresponding pixels.

`imagesc(x,y,C)` specifies the image location. Use `x` and `y` to specify the locations of the corners corresponding to `C(1,1)` and `C(m,n)`. To specify both corners, set `x` and `y` as two-element vectors. To specify the first corner and let `imagesc` determine the other, set `x` and `y` as scalar values. The image is stretched and oriented as applicable.

`imagesc('CData',C)` adds the image to the current axes without replacing existing plots. This syntax is the low-level version of `imagesc(C)`. For more information, see “High-Level Versus Low-Level Version” on page 1-7384.

`imagesc('XData',x,'YData',y,'CData',C)` specifies the image location. This syntax is the low-level version of `imagesc(x,y,C)`.

`imagesc( ___ ,Name,Value)` specifies image properties using one or more name-value pair arguments. You can specify name-value pair arguments after any of the input argument combinations in the previous syntaxes. For a list of image properties and descriptions, see Image.

`imagesc( ___ ,clims)` specifies the data values that map to the first and last elements of the colormap. Specify `clims` as a two-element vector of the form `[cmin cmax]`, where values less than or equal to `cmin` map to the first color in the colormap and values greater than or equal to `cmax` map to the last color in the colormap. Specify `clims` after name-value pair arguments.

`imagesc(ax, ___ )` creates the image in the axes specified by `ax` instead of in the current axes (`gca`). Specify the axes as the first input argument.

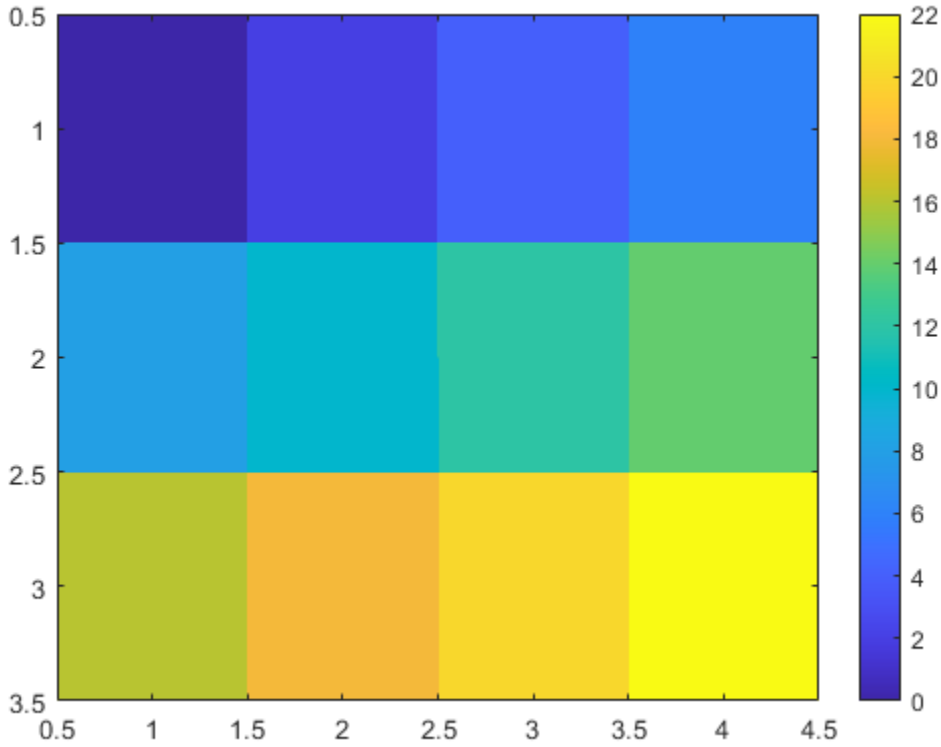
`im = imagesc( ___ )` returns the `Image` object created. Use `im` to set properties of the image after it is created. You can specify this output with any of the input argument combinations in the previous syntaxes.

Examples

Display Image of Matrix Data

Create matrix C. Display an image of the data in C. Add a colorbar to the graph to show the current colormap. By default, `imagesc` scales the color limits so that image uses the full range of the colormap, where the smallest value in `C` maps to the first color in the colormap and the largest value maps to the last color.

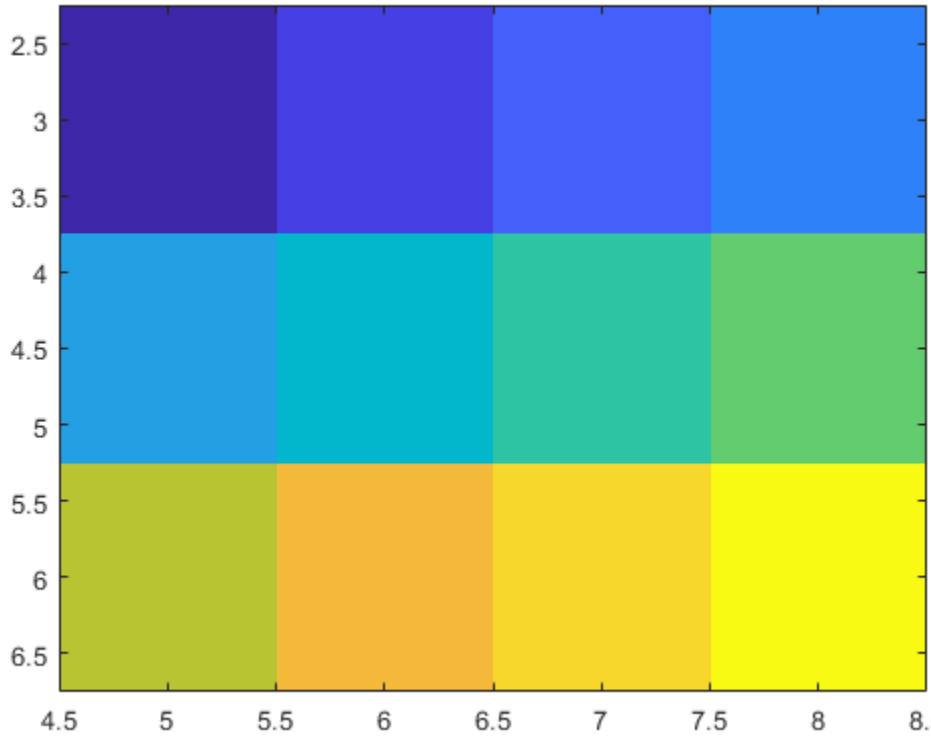
```
C = [0 2 4 6; 8 10 12 14; 16 18 20 22];
imagesc(C)
colorbar
```



Control Image Placement

Place the image so that it lies between 5 and 8 on the *x*-axis and between 3 and 6 on the *y*-axis.

```
x = [5 8];
y = [3 6];
C = [0 2 4 6; 8 10 12 14; 16 18 20 22];
imagesc(x,y,C)
```

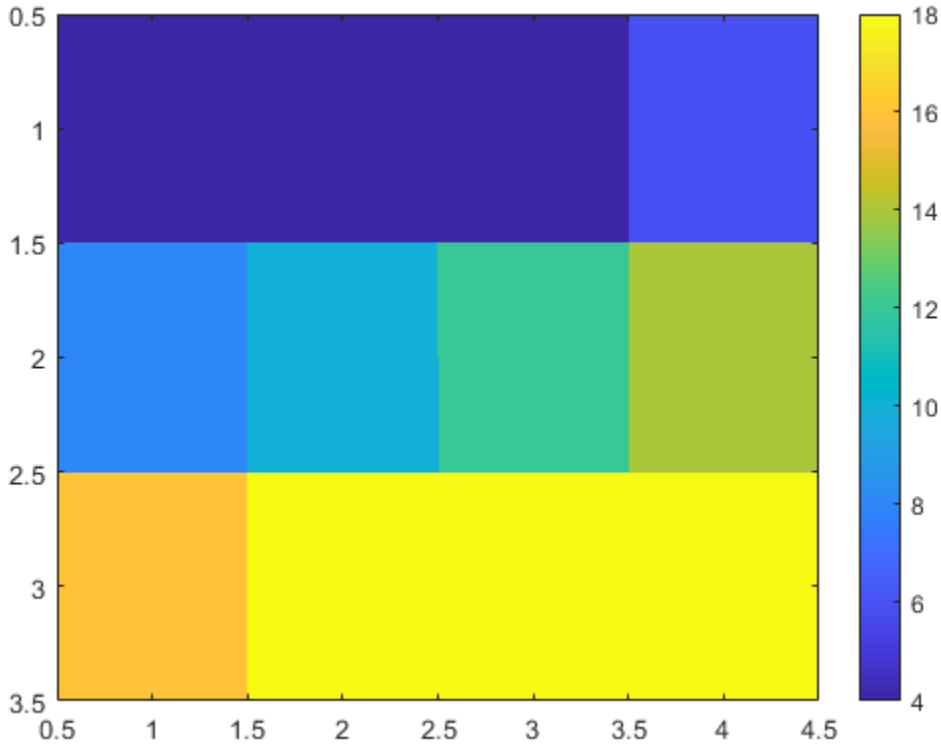


Notice that the pixel corresponding to `C(1,1)` is centered over the point (5,3). The pixel corresponding to `C(3,4)` is centered over the point (8,6). `imagesc` positions and orients the rest of the image between those two points.

Control Scaling of Data Values into Colormap

Create `C` as an array of data values. Create an image of `C` and set the color limits so that values of 4 or less map to the first color in the colormap and values of 18 or more map to the last color in the colormap. Display a colorbar to show how the data values map into the colormap.

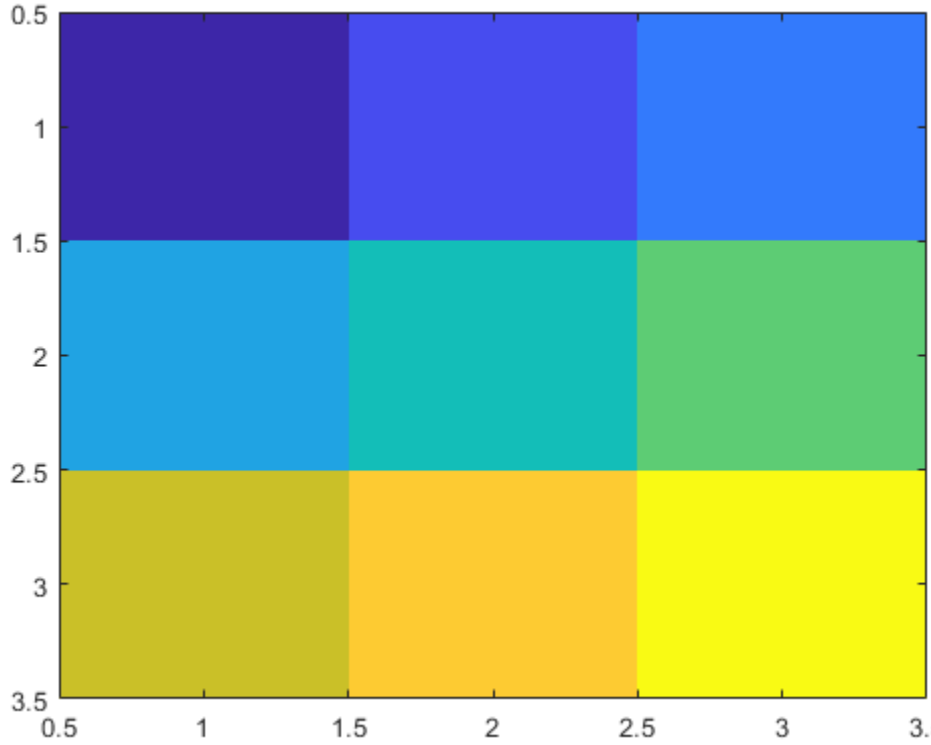
```
C = [0 2 4 6; 8 10 12 14; 16 18 20 22];
clims = [4 18];
imagesc(C,clims)
colorbar
```



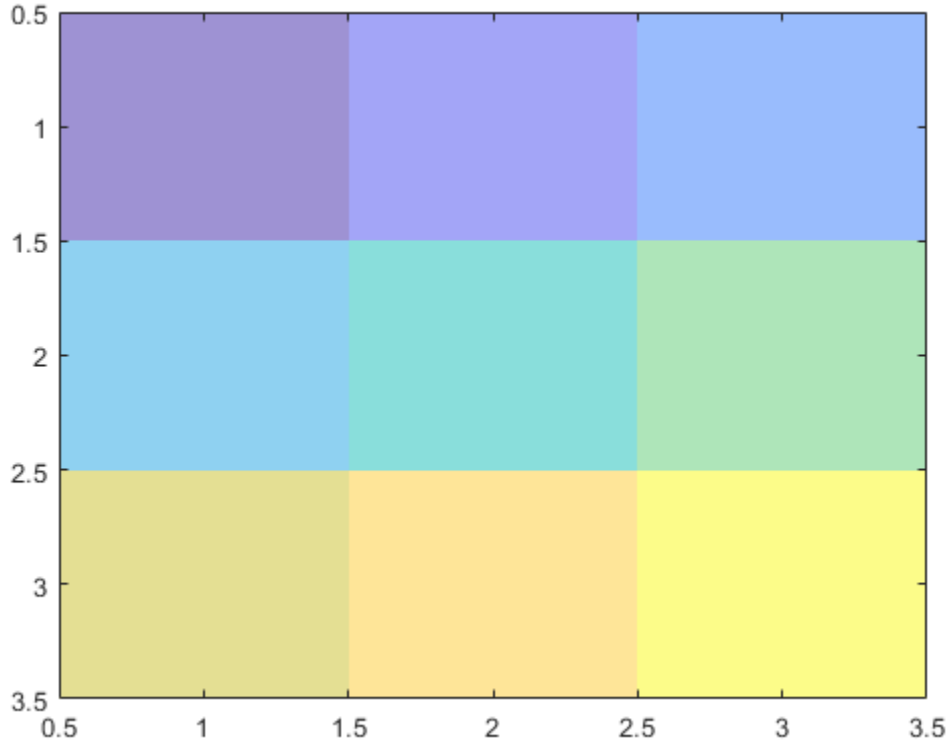
Modify Image After Creation

Create an image and return the image object, `im`. Then, make the image semitransparent by setting the `AlphaData` property of the image object.

```
C = [1 2 3; 4 5 6; 7 8 9];
im = imagesc(C);
```



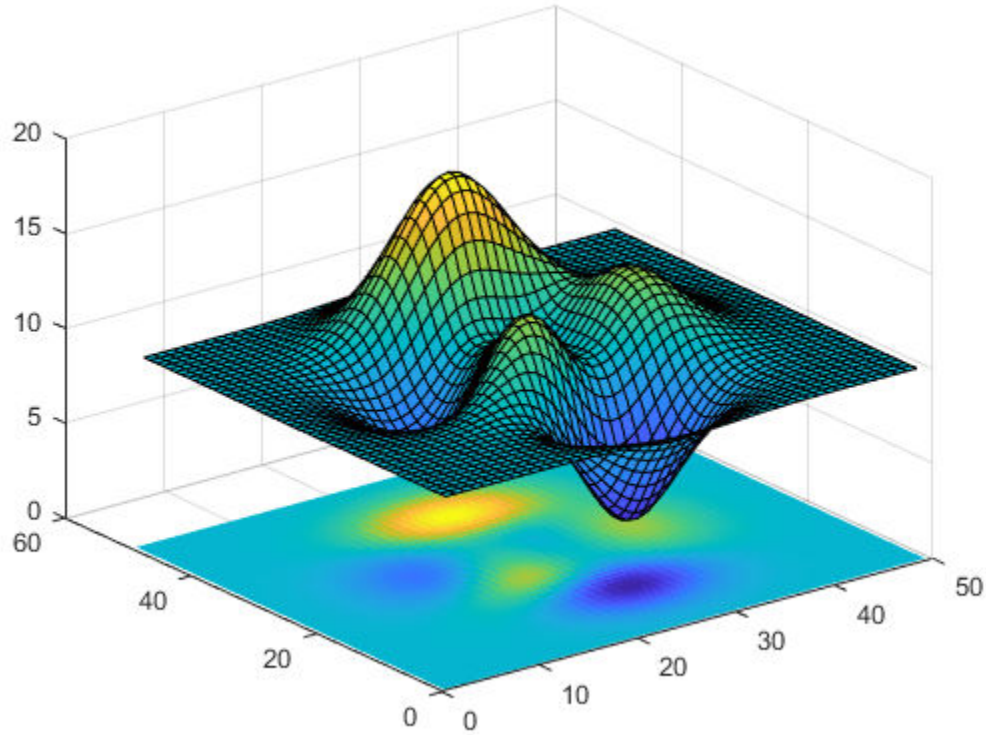

```
im.AlphaData = .5;
```



Add Image to Axes in 3-D View

Create a surface plot. Then, add an image under the surface. `imagesc` displays the image in the xy-plane.

```
Z = 10 + peaks;
surf(Z)
hold on
imagesc(Z)
```



Input Arguments

C — Image color data
vector or matrix

Image color data, specified as a vector or a matrix. Each element of C defines a color for one pixel of the image. The elements of C map to colors in the colormap of the associated axes. The smallest value in C maps to the first color in the colormap and the largest value maps to the last color. The behavior of NaN elements is not defined.

Note If you specify `C` as an `m`-by-`n`-by-3 array, then the `imagesc` function interprets the image as a truecolor (RGB) image. `imagesc` does not rescale pixel values of truecolor images. Use the `rescale` function to scale truecolor pixel values before calling `imagesc`.

To use the low-level version of the `imagesc` function instead, set the `CData` property as a name-value pair. For example, `imagesc('CData',C)`.

Converting Between Data Types

To convert grayscale image data from an integer type to type `double`, add 1. For example, if `X8` is grayscale image data of type `uint8`, convert it to type `double` using:

```
X64 = double(X8) + 1;
```

To convert grayscale image data from type `double` to an integer type, subtract 1 and use `round` to ensure that all the values are integers. For example, if `X64` is grayscale image data of type `double`, convert it to `uint8` using:

```
X8 = uint8(round(X64 - 1));
```

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

x — Placement along x-axis

`[1 size(C,2)]` (default) | two-element vector | scalar

Placement along the *x*-axis, specified in one of these forms:

- Two-element vector — Use the first element as the location for the center of `C(1,1)` and the second element as the location for the center of `C(m,n)`, where `[m,n] = size(C)`. If `C` is a 3-D array, then `m` and `n` are the first two dimensions. Evenly distribute the centers of the remaining elements of `C` between those two points.

  The width of each pixel is determined by the expression:

  ```
  (x(2)-x(1))/(size(C,2)-1)
  ```

  If `x(1) > x(2)`, then the image is flipped left-right.
- Scalar — Center `C(1,1)` at this location and each following element one unit apart.

To use the low-level version of the `imagesc` function instead, set the `XData` property as a name-value pair. For example, `imagesc('XData',x,'YData',y,'CData',C)`.

You cannot interactively pan or zoom outside the x-axis limits or y-axis limits of an image, unless the limits are already set outside the bounds of the image. If the limits are already outside the bounds, there is no such restriction. If other objects (such as a line) occupy the axes and extend beyond the bounds of the image, you can pan or zoom to the bounds of the other objects, but no further.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

y — Placement along y-axis
`[1 size(C,1)]` (default) | two-element vector | scalar

Placement along *y*-axis, specified in one of these forms:

- Two-element vector — Use the first element as the location for the center of `C(1,1)` and the second element as the location for the center of `C(m,n)`, where `[m,n] = size(C)`. If `C` is a 3-D array, then `m` and `n` are the first two dimensions. Evenly distribute the centers of the remaining elements of `C` between those two points.

  The height of each pixel is determined by the expression:

  ```
  (y(2)-y(1))/(size(C,1)-1)
  ```

  If `y(1) > y(2)`, then the image is flipped up-down.
- Scalar — Center `C(1,1)` at this location and each following element one unit apart.

To use the low-level version of the `imagesc` function instead, set the `YData` property as a name-value pair. For example, `imagesc('XData',x,'YData',y,'CData',C)`.

You cannot interactively pan or zoom outside the x-axis limits or y-axis limits of an image, unless the limits are already set outside the bounds of the image. If the limits are already outside the bounds, there is no such restriction. If other objects (such as a line) occupy the axes and extend beyond the bounds of the image, you can pan or zoom to the bounds of the other objects, but no further.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`clims` — Color limits
two-element vector of form `[cmin cmax]`

Color limits, specified as a two-element vector of the form `[cmin cmax]`, where `cmax` is greater than `cmin`. Values in `C` that are less than or equal to `cmin` map to the first color in

the colormap. Values greater than or equal to `cmax` map to the last color in the colormap. Values between `cmin` and `cmax` linearly map to the colormap.

If you specify the color limits, then the `imagesc` function sets the `CLim` property of the axes to the values specified. If you do not specify the color limits, then `imagesc` sets the `CLim` property of the axes to the minimum and maximum values in `C`.

`ax` — `Axes` object
`Axes` object

`Axes` object. If you do not specify an `Axes` object, then `imagesc` uses the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `imagesc([1 2 3],'AlphaData',0.5)` displays a semitransparent image.

The properties listed here are a subset of image properties. For a complete list, see Image.

`AlphaData` — Transparency data
`1` (default) | scalar | array the same size as `CData`

Transparency data, specified in one of these forms:

- Scalar — Use a consistent transparency across the entire image.
- Array the same size as `CData` — Use a different transparency value for each image element.

The `AlphaDataMapping` property controls how MATLAB interprets the alpha data transparency values.

Example: `0.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`AlphaDataMapping` — Interpretation of `AlphaData` values
`'none'` (default) | `'scaled'` | `'direct'`

Interpretation of `AlphaData` values, specified as one of these values:

- `'none'` — Interpret the values as transparency values. A value of 1 or greater is completely opaque, a value of 0 or less is completely transparent, and a value between 0 and 1 is semitransparent.
- `'scaled'` — Map the values into the figure's alphamap. The minimum and maximum alpha limits of the axes determine the alpha data values that map to the first and last elements in the alphamap, respectively. For example, if the alpha limits are `[3 5]`, then alpha data values less than or equal to `3` map to the first element in the alphamap. Alpha data values greater than or equal to `5` map to the last element in the alphamap. The `ALim` property of the axes contains the alpha limits. The `Alphamap` property of the figure contains the alphamap.
- `'direct'` — Interpret the values as indices into the figure's alphamap. Values with a decimal portion are fixed to the nearest lower integer:
  - If the values are of type `double` or `single`, then values of 1 or less map to the first element in the alphamap. Values equal to or greater than the length of the alphamap map to the last element in the alphamap.
  - If the values are of type integer, then values of 0 or less map to the first element in the alphamap. Values equal to or greater than the length of the alphamap map to the last element in the alphamap (or up to the range limits of the type). The integer types are `uint8`, `uint16`, `uint32`, `uint64` , `int8`, `int16`, `int32`, and `int64`.
  - If the values are of type `logical`, then values of 0 map to the first element in the alphamap and values of 1 map to the second element in the alphamap.

Output Arguments

`im` — `Image` object
`Image` object

`Image` object. Use `im` to set properties of the image after it is created. For a list, see Image.

More About

High-Level Versus Low-Level Version

The `imagesc` function has two versions, the high-level version and the low-level version. If you use `imagesc` with `'CData'` as an input argument, then you are using the low-level version. Otherwise, you are using the high-level version.

The high-level version of `imagesc` calls `newplot` before plotting and sets these axes properties:

- `Layer` to `'top'`. The image is shown in front of any tick marks or grid lines.
- `YDir` to `'reverse'`. Values along the *y*-axis increase from top to bottom. To decrease the values from top to bottom, set `YDir` to `'normal'`. This setting reverses both the *y*-axis and the image.
- `View` to `[0 90]`.

The low-level version of the `imagesc` function does not call `newplot` and does not set these axes properties.

For both versions, the `imagesc` function sets:

- The `CData` property of the `Image` object to the values in `C`.
- The `CDataMapping` property of the `Image` object to `'scaled'`.
- The `CLim` property of the `Axes` object to the minimum and maximum values in `C`, unless you specify the `clims` input argument.

Tips

- To read image data into MATLAB from graphics files in various standard formats, such as TIFF, use `imread`. To write MATLAB image data to graphics files, use `imwrite`. The `imread` and `imwrite` functions support various graphics file formats and compression schemes.
- To view or set the color limits of the axes, you can use the `caxis` function.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions

`caxis` | `colorbar` | `colormap` | `image` | `imfinfo` | `imread` | `imshow` | `imwrite`

Properties

Image

Topics

“Image Types”

Introduced before R2006a

imapprox

Approximate indexed image by reducing number of colors

Syntax

```
[Y,newmap] = imapprox(X,map,Q)
[Y,newmap] = imapprox(X,map,tol)
Y = imapprox(X,map,inmap)
___ = imapprox(___,dithering)
```

Description

`[Y,newmap] = imapprox(X,map,Q)` approximates the colors in indexed image `X` and associated colormap `map` by using minimum variance quantization with `Q` quantized colors. `imapprox` returns the indexed image `Y` with colormap `newmap`.

`[Y,newmap] = imapprox(X,map,tol)` approximates the colors in indexed image `X` and associated colormap `map` by using uniform quantization with tolerance `tol`.

`Y = imapprox(X,map,inmap)` approximates the colors in indexed image `X` and associated colormap `map` by using inverse colormap mapping with colormap `inmap`. The inverse colormap algorithm finds the colors in `inmap` that best match the colors in `map`.

`___ = imapprox(___,dithering)` enables or disables dithering.

Examples

Reduce Number of Colors in Indexed Image

Load an indexed image of a mandrill's face. Display image `X` using its associated colormap, `map`, which has 220 colors.

```
figure
load mandrill
```

```
image(X)
colormap(map)
axis off
axis image
```

Reduce the number of colors in the indexed image from 220 to only 16 colors by producing a new image, `Y`, and its associated colormap, `newmap`.

```
figure
[Y,newmap] = imapprox(X,map,16);
image(Y)
colormap(newmap)
axis off
axis image
```

Input Arguments

X — Indexed image with many colors
m-by-*n* matrix of nonnegative integers

Indexed image with many colors, specified as an *m*-by-*n* matrix of nonnegative integers.

Data Types: `single` | `double` | `uint8` | `uint16`

`map` — Colormap with many colors
d-by-3 matrix

Colormap with many colors associated with indexed image `X`, specified as a *d*-by-3 matrix with values in the range [0, 1]. Each row of `map` is a three-element RGB triplet that specifies the red, green, and blue components of a single color of the colormap.

Data Types: `double`

`Q` — Number of quantized colors
positive integer

Number of quantized colors used for minimum variance quantization, specified as a positive integer that is less than or equal to 65,536. The returned colormap `newmap` has `Q` or fewer colors.

`tol` — Tolerance
number in the range [0, 1]

Tolerance used for uniform quantization, specified as a number in the range [0, 1]. The returned colormap `newmap` has `(floor(1/tol)+1)^3` or fewer colors.

`inmap` — Colormap with fewer colors
c-by-3 matrix

Colormap with fewer colors used for inverse colormap mapping, specified as a *c*-by-3 matrix with values in the range [0, 1]. Each row of `inmap` is a three-element RGB triplet that specifies the red, green, and blue components of a single color of the colormap. The colormap has a maximum of 65,536 colors.

Data Types: `double`

`dithering` — Perform dithering
`'dither'` (default) | `'nodither'`

Perform dithering, specified as `'dither'` or `'nodither'`. Dithering increases the color resolution at the expense of spatial resolution. For more information, see `dither`.

If you select `'nodither'`, then `imapprox` does not perform dithering. Instead, the function maps each color in the original image to the closest color in the new colormap.

Output Arguments

`Y` — Indexed image with fewer colors
m-by-*n* matrix of nonnegative integers

Indexed image with fewer colors, returned as an *m*-by-*n* matrix of positive integers. If the length of `newmap` (or `immap`, if specified) is less than or equal to 256, then the output image is of class `uint8`. Otherwise, the output image is of class `double`.

Data Types: `double` | `uint8`

`newmap` — Colormap with fewer colors
c-by-3 matrix

Colormap with fewer colors associated with the output indexed image `Y`, returned as a *c*-by-3 matrix with values in the range [0, 1]. Each row of `newmap` is a three-element RGB triplet that specifies the red, green, and blue components of a single color of the colormap. The colormap has a maximum of 65,536 colors.

Data Types: `double`

Algorithms

`imapprox` uses `rgb2ind` to create a new colormap that uses fewer colors. For more information about quantization and inverse colormap mapping, see the Algorithms on page 1-13467 of `rgb2ind`.

See Also

`cmunique` | `dither` | `rgb2ind`

Topics

"Image Types"
"Reduce the Number of Colors in an Image" (Image Processing Toolbox)

Introduced before R2006a

imfinfo

Information about graphics file

Syntax

```
info = imfinfo(filename)
info = imfinfo(filename,fmt)
```

Description

`info = imfinfo(filename)` returns a structure whose fields contain information about an image in a graphics file, `filename`.

The format of the file is inferred from its contents.

- If `filename` is a TIFF, PGM, PBM, PPM, HDF, ICO, GIF, or CUR file containing more than one image, then `info` is a structure array with one element for each image in the file. For example, `info(3)` would contain information about the third image in the file.

`info = imfinfo(filename,fmt)` additionally looks for a file named `filename.fmt`, if MATLAB cannot find a file named `filename`.

Examples

Return Information About Graphics File

Find information about the example image, `ngc6543a.jpg`.

```
info = imfinfo('ngc6543a.jpg');
```

The `info` structure contains the following information fields: `Filename`, `FileModDate`, `FileSize`, `Format`, `FormatVersion`, `Width`, `Height`, `BitDepth`, `ColorType`, `FormatSignature`, `NumberOfSamples`, `CodingMethod`, `CodingProcess`, and `Comment`.

To display information from the structure, for example `CodingMethod`, type `info.CodingMethod` in the command window.

```
info.CodingMethod
```

```
ans =
'Huffman'
```

To display all the properties in the structure, type `info` in the command window.

Input Arguments

`filename` — Name of graphics file
character vector | string scalar

Name of graphics file, specified as a character vector or string scalar.

Depending on the location of the file, `filename` can take on one of these forms.

Location	Form
Current folder or folder on the MATLAB path	Specify the name of the file in `filename`. **Example:** `'myImage.jpg'`
File in a folder	If the file is not in the current folder or in a folder on the MATLAB path, then specify the full or relative path name. **Example:** `'C:\myFolder\myImage.ext'` **Example:** `'\imgDir\myImage.ext'`
URL	If the file is located by an internet URL, then `filename` must contain the protocol type such as, `http://`. **Example:** `'http://hostname/path_to_file/my_image.jpg'`

<table>
<tr><th>Location</th><th>Form</th></tr>
<tr><td>Remote Location</td><td>If the file is stored at a remote location, then <code>filename</code> must contain the full path of the file specified as a uniform resource locator (URL) of the form:

<code>scheme_name://path_to_file/my_file.ext</code>

Based on your remote location, <code>scheme_name</code> can be one of the values in this table.
<table>
<tr><th>Remote Location</th><th><code>scheme_name</code></th></tr>
<tr><td>Amazon S3</td><td><code>s3</code></td></tr>
<tr><td>Windows Azure Blob Storage</td><td><code>wasb</code>, <code>wasbs</code></td></tr>
<tr><td>HDFS</td><td><code>hdfs</code></td></tr>
</table>
For more information, see “Work with Remote Data”.

Example: <code>'s3://bucketname/path_to_file/my_image.jpg'</code></td></tr>
</table>

Data Types: `char` | `string`

`fmt` — Image format
character vector | string scalar

Image format, specified as a character vector or string scalar. The possible values for `fmt` are contained in the MATLAB file format registry. To view of list of these formats, run the `imformats` command.

Example: `'gif'`

Data Types: `char` | `string`

Output Arguments

`info` — Information about graphics file
structure array

Information about the graphics file, returned as a structure array. The set of fields in `info` depends on the individual file and its format. This table lists the nine fields that always appear, and describes their values.

Field Name	Description	Value
`Filename`	Name of the file or the internet URL specified. If the file is not in the current folder, the character vector contains the full path name of the file.	character vector
`FileModDate`	Date when the file was last modified.	character vector
`FileSize`	Size of the file, in bytes.	integer
`Format`	File format, as specified by *`fmt`*. For formats with more than one possible extension (for example, JPEG and TIFF files), `imfinfo` returns the first variant in the file format registry.	character vector
`FormatVersion`	File format version.	character vector or number
`Width`	Image width, in pixels.	integer
`Height`	Image height, in pixels.	integer
`BitDepth`	Number of bits per pixel.	integer
`ColorType`	Image type. `ColorType` includes, but is not limited to, `'truecolor'` for a truecolor (RGB) image, `'grayscale'` for a grayscale intensity image, or `'indexed'` for an indexed image.	character vector

Additional fields returned by some file formats:

- **JPEG and TIFF only** — If `filename` contains Exchangeable Image File Format (EXIF) tags, then `info` might also contain `'DigitalCamera'` or `'GPSInfo'` (global positioning system information) fields.
- **GIF only** — `imfinfo` returns the value of the `'DelayTime'` field in hundredths of seconds.
- **JPEG2000 only** — The `info` structure contains an m-by-3 cell array, `'ChannelDefinition'`. The first column of `'ChannelDefinition'` reports a channel position as it exists in the file. The second column reports the type of channel, and the third column reports the channel mapping.

See Also

`imformats` | `imread` | `imwrite`

Introduced before R2006a

imformats

Manage image file format registry

Syntax

```
imformats

formatStruct = imformats(fmt)

registry = imformats
registry = imformats(formatStruct)
registry = imformats('add',formatStruct)
registry = imformats('remove',fmt)
registry = imformats('update',fmt,formatStruct)
registry = imformats('factory')
```

Description

`imformats` displays a table of information listing all the values in the MATLAB file format registry. This registry determines which file formats the `imfinfo`, `imread`, and `imwrite` functions support.

`formatStruct = imformats(fmt)` searches the known formats in the MATLAB file format registry for the format associated with the file name extension specified by `fmt`. If found, `formatStruct` is a structure containing the characteristics and function names associated with the format. Otherwise, `formatStruct` is an empty structure.

`registry = imformats` returns a structure array, `registry`, containing all the values in the MATLAB file format registry.

`registry = imformats(formatStruct)` sets the MATLAB file format registry for the current MATLAB session to the values in `formatStruct`. The output structure, `registry`, contains the new registry settings. Use this syntax to replace image file format support.

Incorrect use of `imformats` to specify values in the MATLAB file format registry can result in the inability to load any image files. To return the file format registry to a working state, use `imformats` with the `'factory'` input.

`registry = imformats('add',formatStruct)` adds the values in `formatStruct` to the file format registry. Use this syntax to add image file format support.

`registry = imformats('remove',fmt)` removes the format with the extension specified by `fmt` from the file format registry. Use this syntax to remove image file format support.

`registry = imformats('update',fmt,formatStruct)` changes the format registry values for the format with extension `fmt` to have the values specified by `formatStruct`.

`registry = imformats('factory')` resets the MATLAB file format registry to the default format registry values. This removes any user-specified settings.

Examples

Determine if File Format Exists in Registry

Determine if the file format associated with the `.bmp` file extension is in the image file format registry.

```
formatStruct = imformats('bmp')
```

```
formatStruct = struct with fields:
            ext: {'bmp'}
            isa: @isbmp
           info: @imbmpinfo
           read: @readbmp
          write: @writebmp
          alpha: 0
    description: 'Windows Bitmap'
```

`formatStruct` is a non-empty structure, so the BMP file format is in the registry.

Add, Update, or Remove File Format from Registry

Add a hypothetical file format, ABC, to the image file format registry. Update, and then remove the format.

Create a structure with seven fields, defining values for the new format.

```
formatStruct = struct('ext','abc','isa',@isabc,...
    'info',@abcinfo,'read',@readabc,'write','',...
    'alpha',0,'description','My ABC Format')
```

```
formatStruct = struct with fields:
            ext: 'abc'
            isa: @isabc
           info: @abcinfo
           read: @readabc
          write: ''
          alpha: 0
    description: 'My ABC Format'
```

`formatStruct` is a 1-by-1 structure with seven fields. In this example, the `write` field is empty.

Add the new format to the file format registry.

```
registry = imformats('add',formatStruct);
```

Redefine the format associated with the extension, `abc`, by adding a value for the `write` field. Then, update the registry value for the format.

```
formatStruct2 = struct('ext','abc','isa',@isabc,...
    'info',@abcinfo,'read',@readabc,'write',@writeabc,...
    'alpha',0,'description','My ABC Format');

registry = imformats('update','abc',formatStruct2);
```

Remove the format with the extension, `abc`, from the file format registry.

```
registry = imformats('remove','abc');
```

Input Arguments

`formatStruct` — File format registry values

structure array

File format registry values, specified as a structure array with the following 7 fields.

Field	Description	Value
`ext`	File name extensions that are valid for this format.	Cell array of character vectors or string array
`isa`	Name of the function that determines if a file is of a certain format.	Character vector or string scalar, or function handle
`info`	Name of the function that reads information about a file.	Character vector or string scalar, or function handle
`read`	Name of the function that reads image data in a file.	Character vector or string scalar, or function handle
`write`	Name of the function that writes MATLAB data to a file.	Character vector or string scalar, or function handle
`alpha`	Presence or absence of an alpha channel.	`1` if the format has an alpha channel; otherwise it is `0`.
`description`	Text description of the file format.	Character vector or string scalar

The values for the `isa`, `info`, `read`, and `write` fields must be either functions on the MATLAB search path or function handles.

Data Types: `struct`

`fmt` — File format extension

character vector | string scalar

File format extension, specified as a character vector or string scalar.

Example: `'jpg'`

Data Types: `char`

Output Arguments

`registry` — File format registry
structure array

File format registry, returned as a structure array with the following fields.

Field	Description	Value
`ext`	File name extensions that are valid for this format.	Cell array of character vectors
`isa`	Name of the function that determines if a file is of a certain format.	Character vector or function handle
`info`	Name of the function that reads information about a file.	Character vector or function handle
`read`	Name of the function that reads image data in a file.	Character vector or function handle
`write`	Name of the function that writes MATLAB data to a file.	Character vector or function handle
`alpha`	Presence or absence of an alpha channel.	`1` if the format has an alpha channel; otherwise it is `0`.
`description`	Text description of the file format.	Character vector

Note Use the `imread`, `imwrite`, and `imfinfo` functions to read, write, or get information about an image file when the file format is in the format registry. Do not directly invoke the functions returned in the fields of the `registry` structure array.

Tips

- Changes to the format registry do not persist between MATLAB sessions. To have a format always available when you start MATLAB, add the appropriate `imformats` command to the MATLAB startup file, `startup.m`. For more information, see `startup`.

See Also

`imfinfo` | `imread` | `imwrite` | `path`

Topics

"What Is the MATLAB Search Path?"

Introduced before R2006a

ImplicitFunctionLine Properties

Implicit line chart appearance and behavior

Description

`ImplicitFunctionLine` properties control the appearance and behavior of an `ImplicitFunctionLine` object. By changing property values, you can modify certain aspects of the line chart. You can use dot notation to refer to a particular object and property:

```
fp = fimplicit(@(x,y) x.^2 + y.^2 - 3)
lw = fp.LineWidth
fp.LineWidth = 2
```

Properties

Color and Styling

`Color` — Line color

`[0 0 1]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`ColorMode` — Control how `Color` is set
`'auto'` (default) | `'manual'`

Control how the `Color` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the `Color` property by selecting a color from the `ColorOrder` property of the axes.
- `'manual'` — You control the value of the `Color` property manually, either by setting the value of the `Color` property directly on the object, or by including a color in the `LineSpec` argument when you call a plotting function.

If you change the value of the `Color` property manually, MATLAB changes the value of the `ColorMode` property to `'manual'`.

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineStyleMode` — Control how `LineStyle` is set
`'auto'` (default) | `'manual'`

Control how the `LineStyle` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the `LineStyle` property by selecting a line style from the `LineStyleOrder` property of the axes.
- `'manual'` — You control the value of the `LineStyle` property manually, either by setting the value of the `LineStyle` property directly on the object, or by specifying the `LineSpec` argument when you call a plotting function.

If you change the value of the `LineStyle` property manually, MATLAB changes the value of the `LineStyleMode` property to `'manual'`.

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

Markers

`Marker` — Marker symbol

`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | `'x'` | `...`

Marker symbol, specified as one of the markers listed in this table. By default, the line does not have markers. Specifying a marker symbol adds markers at selected points along the line.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'+'`

Example: `'diamond'`

`MarkerMode` — Control how `Marker` is set

`'auto'` (default) | `'manual'`

Control how the `Marker` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the object's `Marker` property.
- `'manual'` — You control the value of the `Marker` property manually, either by setting the value of the `Marker` property directly on the object, or by including a marker in the `LineSpec` argument when you call a plotting function.

If you change the value of the `Marker` property manually, MATLAB changes the value of the `MarkerMode` property to `'manual'`.

`MarkerSize` — Marker size
6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

`MarkerEdgeColor` — Marker outline color
`'auto'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'auto'` uses the same color as the `Color` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` value uses the same color as the `MarkerEdgeColor` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to

F. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Example: `'#D2F9A7'`

Function

`Function` — Function to plot
function handle | anonymous function | symbolic expression | symbolic function

Function to plot, specified as a function handle, anonymous function, or a symbolic expression or function.

`XRange` — Plotting interval for *x* values
`[−5 5]` (default) | two-element vector of form `[xmin xmax]`

Plotting interval for *x* values, specified as a two-element vector of the form `[xmin xmax]`.

`XRangeMode` — Selection mode for `XRange`
`'auto'` (default) | `'manual'`

Selection mode for `XRange`, specified as one of these values:

- `'auto'` — Use the default value `[-5 5]`. If axes limits are specified, follow the specified limits instead.
- `'manual'` — Use manually specified values. To specify the values, set the `XRange` property.

`YRange` — Plotting interval for *y* values
`[−5 5]` (default) | two-element vector of form `[ymin ymax]`

Plotting interval for *y* values, specified as a two-element vector of the form `[ymin ymax]`.

`YRangeMode` — Selection mode for `YRange`
`'auto'` (default) | `'manual'`

Selection mode for `YRange`, specified as one of these values:

- `'auto'` — Use the default value `[-5 5]`. If the axes limits are specified, follow the specified limits instead.
- `'manual'` — Use manually specified values. To specify the values, set the `YRange` property.

`MeshDensity` — Number of evaluation points
151 (default) | scalar

Number of evaluation points, specified as a scalar.

Data

`XData` — *x* values
vector

This property is read-only.

x values, returned as a vector. `XData`, `YData`, and `ZData` have equal lengths.

`YData` — *y* values
vector

This property is read-only.

y values, returned as a vector. `XData`, `YData`, and `ZData` have equal lengths.

`ZData` — *z* values
vector of zeros

This property is read-only.

z values, returned as a vector of zeros. `XData`, `YData`, and `ZData` have equal lengths.

Legend

`DisplayName` — Text for legend label
autogenerated label (default) | character vector | string

Text for legend label, specified as a custom character vector or string. The default label is autogenerated from the `Function` property and the `texlabel` function. The legend does not appear until you call the `legend` function.

Data Types: `char` | `string`

`Annotation` — Control for including or excluding object from legend
`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).

- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`DataTipTemplate` — Data tip content

`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

`Clipping` — Clipping of object to axes limits

`'on'` (default) | `'off'`

Clipping of the object to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the object that are outside the axes limits.
- `'off'` — Display the entire object, even if parts of it appear outside the axes limits. Parts of the object might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the object so that it is larger than the original plot.

The `Clipping` property of the axes that contains the object must be set to `'on'`. Otherwise, this property has no effect. For more information about the clipping behavior, see the `Clipping` property of the axes.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks when visible. The `Visible` property must be set to `'on'` and you must click a part of the `ImplicitFunctionLine` object that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The `HitTest` property determines if the `ImplicitFunctionLine` object responds to the click or if an ancestor does.
- `'all'` — Capture mouse clicks regardless of visibility. The `Visible` property can be set to `'on'` or `'off'` and you can click a part of the `ImplicitFunctionLine` object that has no color. The `HitTest` property determines if the `ImplicitFunctionLine` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `ImplicitFunctionLine` object passes the click through it to the object below it in the current view of the figure window. The `HitTest` property has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `ImplicitFunctionLine` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `ImplicitFunctionLine` object that has one of these:
  - `HitTest` property set to `'on'`
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

Note The `PickableParts` property determines if the `ImplicitFunctionLine` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent

`Axes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `Group`, or `Transform` object.

`Children` — Children

empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the chart object.

HandleVisibility — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'implicitfunctionline'`

This property is read-only.

Type of graphics object, returned as `'implicitfunctionline'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
[ ] (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`fimplicit`

Topics

“Access Property Values”
“Graphics Object Properties”

Introduced in R2016b

ImplicitFunctionSurface Properties

Implicit surface chart appearance and behavior

Description

`ImplicitFunctionSurface` properties control the appearance and behavior of `ImplicitFunctionSurface` objects. By changing property values, you can modify certain aspects of the surface chart. You can use dot notation to refer to a particular object and property:

```
fs = fimplicit3(@(x,y,z) x.^2 + y.^2 - z.^2)
lw = fs.LineWidth
fs.LineWidth = 2
```

Properties

Faces

`FaceColor` — Face color

`'interp'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Face color, specified as `'interp'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'interp'` interpolates the colors based on the `ZData` values.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`FaceAlpha` — Face transparency

1 (default) | scalar in range `[0 1]`

Face transparency, specified as a scalar in the range `[0,1]`. Use uniform transparency across all of the faces. A value of `1` is fully opaque and `0` is completely transparent. Values between `0` and `1` are semitransparent.

Edges

EdgeColor — Line color

`[0 0 0]` (default) | `'interp'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as `'interp'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default RGB triplet value of `[0 0 0]` corresponds to black. The `'interp'` value colors the edges based on the `ZData` values.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
'-'	Solid line	
'--'	Dashed line	
':'	Dotted line	
'-.'	Dash-dotted line	
'none'	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

Markers

`Marker` — Marker symbol
`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | `'x'` | `...`

Marker symbol, specified as one of the markers listed in this table. By default, the object does not display markers. Specifying a marker symbol adds markers at the intersection points of mesh lines.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'+'`

Example: `'diamond'`

`MarkerSize` — Marker size

6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

`MarkerEdgeColor` — Marker outline color

`'auto'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'auto'` uses the same color as the `EdgeColor` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

Example: `'#D2F9A7'`

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` value uses the same color as the `MarkerEdgeColor` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Example: `'#D2F9A7'`

Function

`Function` — Function to plot
function handle | anonymous function | symbolic expression | symbolic function

Function to plot, specified as a function handle, anonymous function, or a symbolic expression or function.

XRange — Plotting interval for *x* values
`[−5 5]` (default) | two-element vector of form `[xmin xmax]`

Plotting interval for *x* values, specified as a two-element vector of the form `[xmin xmax]`.

XRangeMode — Selection mode for XRange
`'auto'` (default) | `'manual'`

Selection mode for `XRange`, specified as one of these values:

- `'auto'` — Use the default value `[-5 5]`. If axes limits are specified, follow the specified limits instead.
- `'manual'` — Use manually specified values. To specify the values, set the `XRange` property.

YRange — Plotting interval for *y* values
`[−5 5]` (default) | two-element vector of form `[ymin ymax]`

Plotting interval for *y* values, specified as a two-element vector of the form `[ymin ymax]`.

YRangeMode — Selection mode for YRange
`'auto'` (default) | `'manual'`

Selection mode for `YRange`, specified as one of these values:

- `'auto'` — Use the default value `[-5 5]`. If the axes limits are specified, follow the specified limits instead.
- `'manual'` — Use manually specified values. To specify the values, set the `YRange` property.

ZRange — Plotting interval for *z* values
`[−5 5]` (default) | two-element vector of form `[zmin zmax]`

Plotting interval for *z* values, specified as a two-element vector of the form `[zmin zmax]`.

ZRangeMode — Selection mode for ZRange
`'auto'` (default) | `'manual'`

Selection mode for `ZRange`, specified as one of these values:

- `'auto'` — Use the default value `[-5 5]`. If the axes limits are specified, follow the specified limits instead.
- `'manual'` — Use manually specified values. To specify the values, set the `ZRange` property.

`MeshDensity` — Number of evaluation points per direction
35 (default) | scalar

Number of evaluation points per direction, specified as a scalar.

Lighting

`AmbientStrength` — Strength of ambient light
`0.3` (default) | scalar in range `[0,1]`

Strength of ambient light, specified as a scalar value in the range `[0,1]`. Ambient light is a nondirectional light that illuminates the entire scene. There must be at least one visible light object in the axes for the ambient light to be visible.

The `AmbientLightColor` property for the axes sets the color of the ambient light. The color is the same for all objects in the axes.

Example: `0.5`

Data Types: `double`

`DiffuseStrength` — Strength of diffuse light
`0.6` (default) | scalar in range `[0,1]`

Strength of diffuse light, specified as a scalar value in the range `[0,1]`. Diffuse light is the nonspecular reflectance from light objects in the axes.

Example: `0.3`

Data Types: `double`

`SpecularStrength` — Strength of specular reflection
`0.9` (default) | scalar in range `[0,1]`

Strength of specular reflection, specified as a scalar value in the range `[0,1]`. Specular reflections are the bright spots on the surface from light objects in the axes.

Example: `0.3`

Data Types: `double`

`SpecularExponent` — Size of specular spot
10 (default) | scalar greater than or equal to 1

Size of specular spot, specified as a scalar value greater than or equal to 1. Most materials have exponents in the range `[5 20]`.

Example: 7

Data Types: `double`

`SpecularColorReflectance` — Color of specular reflections
1 (default) | scalar in range `[0,1]`

Color of specular reflections, specified as a scalar value in the range `[0,1]`. A value of `1` sets the color using only the color of the light source. A value of `0` sets the color using both the color of the object from which it reflects and the color of the light source. The `Color` property of the light contains the color of the light source. The proportions vary linearly for values in between.

Example: `0.5`

Data Types: `double`

Legend

`DisplayName` — Text for legend label

autogenerated label (default) | character vector | string

Text for legend label, specified as a custom character vector or string. The default label is autogenerated from the `Function` property and the `texlabel` function. The legend does not appear until you call the `legend` function.

Data Types: `char` | `string`

`Annotation` — Control for including or excluding object from legend

`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`DataTipTemplate` — Data tip content
`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

`UIContextMenu` — Context menu
empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state
`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles
`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.

- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.

- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks when visible. The `Visible` property must be set to `'on'` and you must click a part of the `ImplicitFunctionSurface` object that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The `HitTest` property determines if the `ImplicitFunctionSurface` object responds to the click or if an ancestor does.
- `'all'` — Capture mouse clicks regardless of visibility. The `Visible` property can be set to `'on'` or `'off'` and you can click a part of the `ImplicitFunctionSurface`

object that has no color. The `HitTest` property determines if the `ImplicitFunctionSurface` object responds to the click or if an ancestor does.

- `'none'` — Cannot capture mouse clicks. Clicking the `ImplicitFunctionSurface` object passes the click through it to the object below it in the current view of the figure window. The `HitTest` property has no effect.

`HitTest` — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `ImplicitFunctionSurface` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `ImplicitFunctionSurface` object that has one of these:
  - `HitTest` property set to `'on'`
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

---

Note The `PickableParts` property determines if the `ImplicitFunctionSurface` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent

`Axes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `Group`, or `Transform` object.

`Children` — Children
empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the chart object.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object
`'implicitfunctionsurface'`

This property is read-only.

Type of graphics object, returned as `'implicitfunctionsurface'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

`Tag` — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`fimplicit3`

Topics

“Access Property Values”
“Graphics Object Properties”

Introduced in R2016b

import

Add package, class, or functions to current import list

Syntax

```
import PackageName.ClassName
import PackageName.FunctionName
import PackageName.ClassName.staticMethodName
import PackageName.*
```

```
import
L = import
```

Description

`import PackageName.ClassName` adds the class name to the current import list. To refer to a class without specifying the entire package name, use the `import` function in your code.

The import list scope is defined as follows:

- Function or script, including nested and local function — Scope is the function and the function does not share the import list of the parent function. If the import list is needed in a MATLAB function or script and in any local functions, you must call the `import` function for each function.

  The scope of a script is the script body. The imports in a script are available only in the script body and are not available in the scopes which call the script. For example, executing a script containing imports at the command prompt does not make the imports available in the command window.

  The import list of a function is persistent across calls to that function and is cleared only when the function is cleared. For more information, see the `clear` function. Do not call `clear import` within a function or a script.
- Base workspace — Scope is code executed at the command prompt. To clear the base import list, type `clear import` at the MATLAB command prompt.

`import PackageName.FunctionName` adds the specified package-based function. Use this syntax to shorten the name of a specific function in a package without importing every function in the package, which might cause unexpected name conflicts.

`import PackageName.ClassName.staticMethodName` adds the specified static method. Use this syntax to shorten the name of a specific static method.

`import PackageName.*` adds the specified package name. `PackageName` must be followed by `.*`.

Avoid using this syntax, as importing packages brings an unspecified set of names into the local scope, which might conflict with names in the MATLAB workspace. One possible use for this syntax is to import a partial package name. Then when you call a function, you use a shorter package name which does not conflict with simple function names. For example, the `matlab.io.hdf4.sd` package has a `close` function, which can conflict with the MATLAB `close` function.

`import` displays the current import list in the scope.

`L = import` returns the current import list.

Examples

Shorten Calls to Java Class Methods

```
import java.util.Currency java.lang.String
```

Create a `java.lang.String` object. There is no need to type the package name, `java.lang`.

```
s = String('hello')
```

```
s =

hello
```

List the `Currency` class methods, without typing the package name.

```
methods Currency
```

```
Methods for class Currency:
```

```
equals                    getDisplayName            notify
getAvailableCurrencies    getInstance               notifyAll
getClass                  getNumericCode            toString
getCurrencyCode           getSymbol                 wait
getDefaultFractionDigits  hashCode
```

Shorten HDF4 Scientific Data Set Package Name

Use partial package names on your import list to simplify calls to `matlab.io.hdf4.sd` package functions and avoid conflicts with the MATLAB `close` function.

```
import matlab.io.hdf4.*
```

Display the full path to the example file `sd.hdf` on your system using the shortened package name `sd`.

```
sdID = sd.start('sd.hdf');
filename = sd.getFilename(sdID)
```

```
filename =

C:\Program Files\MATLAB\R2015a\toolbox\matlab\imagesci\sd.hdf
```

Call the close function with the `sd` package name.

```
sd.close(sdID)
```

There is no name conflict with the MATLAB `close` function when you import the partial package name.

```
which close
```

```
C:\Program Files\MATLAB\R2015a\toolbox\matlab\graphics\close.p
```

If you use the `matlab.io.hdf4.sd.*` syntax to import the entire package name, when you call `close`, MATLAB always chooses the package function. You cannot use `close` to remove a figure.

Import Single Package Function

Import the `matlab.io.hdf4.sd` package function, `readChunk` in a function, `myfunc`. You can call the function using the simple name `readChunk`, but only within the scope of `myfunc`.

```
function data = myfunc(ID,n,m)
import matlab.io.hdf4.sd.readChunk
data = readChunk(ID,[n m]);
end
```

Import Single Static Method

Import the `meta.class.fromName` static method in a function, `myFunc`. You can call the static method using the simple name `fromName`, but only within the scope of `myFunc`.

```
function metaClsObj = myFunc(ClassName)
    import meta.class.fromName
    metaClsObj = fromName(ClassName);
end
```

Import Package in Both Script and Function

Open the `sd.hdf` example file and access the `temperature` data set.

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
```

Call the `myfunc` function from the previous example to read the data. `myfunc` must have its own `import` statement to use a shorted package name.

```
dataChunk = myfunc(sdsID,0,1);
```

Close the file.

```
sd.endAccess(sdsID)
sd.close(sdID)
```

Display Current Import List for Your System

```
import
```

```
ans =

    'java.util.Currency'
    'java.lang.String'
    'matlab.io.hdf4.*'
    'matlab.io.hdf4.sd.readChunk'
```

Input Arguments

`PackageName` — Name of package
string | character vector

Name of the package, specified as a string or character vector.

Example: `matlab.io.hdf4`

`ClassName` — Name of class
string | character vector

Name of the class, specified as a string or character vector.

Example: `Currency`

`FunctionName` — Name of package function
string | character vector

Name of the package function, specified as a string or character vector.

Example: `readChunk`

`staticMethodName` — Name of static method
string | character vector

Name of the static method, specified as a string or character vector.

Example: `fromName`

Data Types: `char` | `string`

Output Arguments

L — Import list
cell array of character vectors

Import list, returned as a cell array of character vectors.

Limitations

- `import` cannot load a Java JAR package created by the MATLAB Compiler SDK™ product.
- Do not use `import` in conditional statements inside a function. MATLAB preprocesses the `import` statement before evaluating the variables in the conditional statements.

Compatibility Considerations

Change in Rules For Function Precedence Order

Behavior changed in R2019b

Starting in R2019b, MATLAB changes the rules for name resolution, impacting the precedence order of variables, nested functions, local functions, and external functions. These rule are described in “Function Precedence Order”. For information about the changes and tips for updating your code, see “Update Code for R2019b Changes to Function Precedence Order”.

- Identifiers cannot be used for two purposes inside a function
- Identifiers without explicit declarations might not be treated as variables
- Variables cannot be implicitly shared between parent and nested functions
- Change in precedence of compound name resolution
- Anonymous functions can include resolved and unresolved identifiers

The behavior of the `import` function has changed.

- Change in precedence of wildcard-based imports
- Fully qualified import functions cannot have the same name as nested functions

- Fully qualified imports shadow outer scope definitions of the same name
- Error handling when import not found
- Nested functions inherit import statements from parent functions

See Also

`clear` | `importdata` | `load`

Topics

"Use import in MATLAB Functions"
"Package Function and Class Method Name Conflict"
"Function Precedence Order"
"Update Code for R2019b Changes to Function Precedence Order"
"Import Classes"

Introduced before R2006a

importdata

Load data from file

Syntax

```
A = importdata(filename)
A = importdata('-pastespecial')
A = importdata( ___ ,delimiterIn)
A = importdata( ___ ,delimiterIn,headerlinesIn)
[A,delimiterOut,headerlinesOut] = importdata( ___ )
```

Description

`A = importdata(filename)` loads data into array `A`.

`A = importdata('-pastespecial')` loads data from the system clipboard rather than from a file.

`A = importdata( ___ ,delimiterIn)` interprets `delimiterIn` as the column separator in ASCII file, `filename`, or the clipboard data. You can use `delimiterIn` with any of the input arguments in the above syntaxes.

`A = importdata( ___ ,delimiterIn,headerlinesIn)` loads data from ASCII file, `filename`, or the clipboard, reading numeric data starting from line `headerlinesIn+1`.

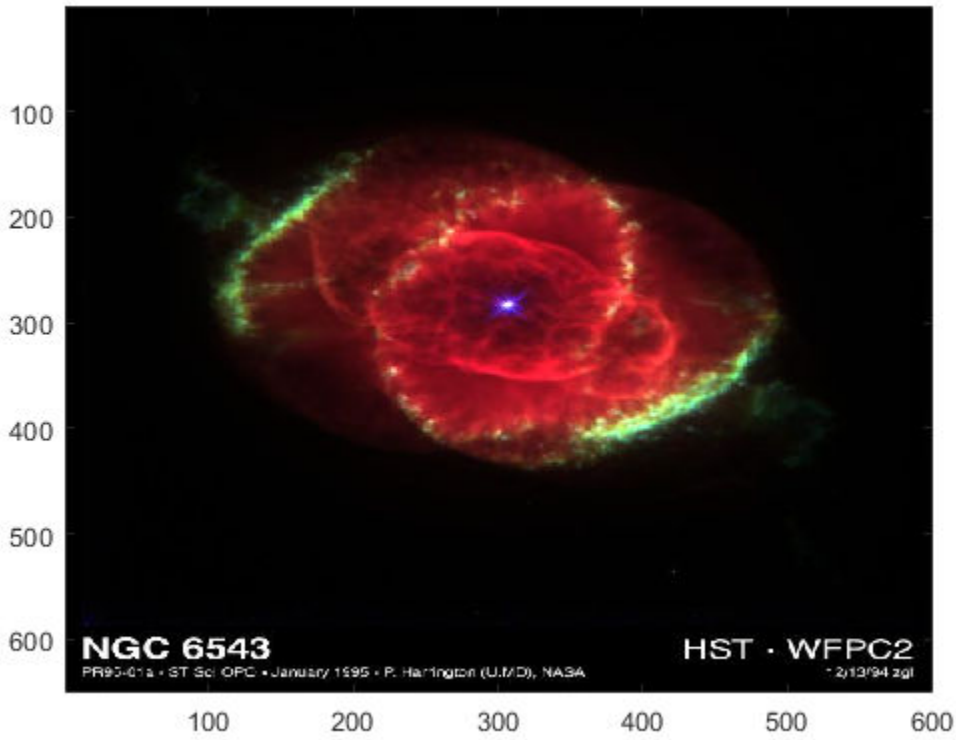
`[A,delimiterOut,headerlinesOut] = importdata( ___ )` additionally returns the detected delimiter character for the input ASCII file in `delimiterOut` and the detected number of header lines in `headerlinesOut`, using any of the input arguments in the previous syntaxes.

Examples

Import and Display an Image

Import and display the sample image, `ngc6543a.jpg`.

```
A = importdata('ngc6543a.jpg');
image(A)
```



The output, `A`, is class `uint8` because the helper function, `imread`, returns empty results for `colormap` and `alpha`.

Import a Text File and Specify Delimiter and Column Header

Using a text editor, create a space-delimited ASCII file with column headers called `myfile01.txt`.

```
Day1  Day2  Day3  Day4  Day5  Day6  Day7
95.01 76.21 61.54 40.57  5.79 20.28  1.53
23.11 45.65 79.19 93.55 35.29 19.87 74.68
60.68  1.85 92.18 91.69 81.32 60.38 44.51
48.60 82.14 73.82 41.03  0.99 27.22 93.18
89.13 44.47 17.63 89.36 13.89 19.88 46.60
```

Import the file, specifying the space delimiter and the single column header.

```
filename = 'myfile01.txt';
delimiterIn = ' ';
headerlinesIn = 1;
A = importdata(filename,delimiterIn,headerlinesIn);
```

View columns 3 and 5.

```
for k = [3, 5]
   disp(A.colheaders{1, k})
   disp(A.data(:, k))
   disp(' ')
end
```

```
Day3
   61.5400
   79.1900
   92.1800
   73.8200
   17.6300


Day5
    5.7900
   35.2900
   81.3200
    0.9900
   13.8900
```

Import a Text File and Return Detected Delimiter

Using a text editor, create a comma-delimited ASCII file called `myfile02.txt`.

```
1,2,3
4,5,6
7,8,9
```

Import the file, and display the output data and detected delimiter character.

```
filename = 'myfile02.txt';
[A,delimiterOut]=importdata(filename)
```

```
A =

     1     2     3
     4     5     6
     7     8     9


delimiterOut =

,
```

Import Data from Clipboard

Copy the following lines to the clipboard. Select the text, right-click, and then select **Copy**.

```
1,2,3
4,5,6
7,8,9
```

Import the clipboard data into MATLAB by typing the following.

```
A = importdata('-pastespecial')
```

```
A =

     1     2     3
```

```
     4     5     6
     7     8     9
```

Input Arguments

`filename` — Name and extension of file to import

character vector | string scalar

Name and extension of the file to import, specified as a character vector or a string scalar. If `importdata` recognizes the file extension, it calls the MATLAB helper function designed to import the associated file format (such as `load` for MAT-files or `xlsread` for spreadsheets). Otherwise, `importdata` interprets the file as a delimited ASCII file.

For ASCII files and spreadsheets, `importdata` expects to find numeric data in a rectangular form (that is, like a matrix). Text headers can appear above or to the left of the numeric data, as follows:

- Column headers or file description text at the top of the file, above the numeric data.
- Row headers to the left of the numeric data.

Example: `'myFile.jpg'`

Data Types: `char` | `string`

`delimiterIn` — Column separator character

character vector | string scalar

Column separator character, specified as a character vector or a string scalar. The default character is interpreted from the file. Use `'\t'` for tab.

Example: `','`

Example: `' '`

Data Types: `char` | `string`

`headerlinesIn` — Number of text header lines in ASCII file

nonnegative scalar integer

Number of text header lines in the ASCII file, specified as a nonnegative scalar integer. If you do not specify `headerlinesIn`, the `importdata` function detects this value in the file.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`A` — Data from the file
matrix | multidimensional array | scalar structure array

Data from the file, returned as a matrix, multidimensional array, or scalar structure array, depending on the characteristics of the file. Based on the file format of the input file, `importdata` calls a helper function to read the data. When the helper function returns more than one nonempty output, `importdata` combines the outputs into a `struct` array.

This table lists the file formats associated with helper functions that can return more than one output, and the possible fields in the structure array, `A`.

File Format	Possible Fields	Class
MAT-files	One field for each variable	Associated with each variable.

ASCII files and Spreadsheets	`data`
	`textdata`
	`colheaders`
	`rowheaders`
		For ASCII files, `data` contains a `double` array. Other fields contain `cell` arrays of character vectors. `textdata` includes row and column headers.
		For spreadsheets, each field contains a `struct`, with one field for each worksheet.

Images	`cdata`
	`colormap`
	`alpha`
		See `imread`.

Audio files	`data` `fs`	See `audioread`.

The MATLAB helper functions for most other supported file formats return one output. For more information about the class of each output, see the functions listed in “Supported File Formats for Import and Export”.

If the ASCII file or spreadsheet contains either column or row headers, but not both, `importdata` returns a `colheaders` or `rowheaders` field in the output structure, where:

- `colheaders` contains only the last line of column header text. `importdata` stores all text in the `textdata` field.
- `rowheaders` is created only when the file or worksheet contains a single column of row headers.

`delimiterOut` — Detected column separator in the input ASCII file
character vector

Detected column separator in the input ASCII file, returned as a character vector.

`headerlinesOut` — Detected number of text header lines in the input ASCII file
integer

Detected number of text header lines in the input ASCII file, returned as an integer.

Tips

- To import ASCII files with nonnumeric characters outside of column or row headers, including columns of character data or formatted dates or times, use `textscan` instead of `importdata`.

See Also

Import Tool | `imread` | `load` | `readcell` | `readmatrix` | `readtable` | `readvars` | `save` | `textscan`

Topics

"Supported File Formats for Import and Export"
"Import Text Files"
"Import Spreadsheets"
"Import or Export a Sequence of Files"

Introduced before R2006a

delimitedTextImportOptions

Import options object for delimited text

Description

A `DelimitedTextImportOptions` object enables you to specify how MATLAB imports tabular data from delimited text files. The object contains properties that control the data import process, including the handling of errors and missing data.

Creation

You can create a `DelimitedTextImportOptions` object using either the `detectImportOptions` function or the `delimitedTextImportOptions` function (described here):

- Use `detectImportOptions` to detect and populate the import properties based on the contents of the delimited text file specified in `filename`.

  ```
  opts = detectImportOptions(filename);
  ```

- Use `delimitedTextImportOptions` to define the import properties based on your import requirements.

Syntax

```
opts = delimitedTextImportOptions
opts = delimitedTextImportOptions('NumVariables',numVars)
opts = delimitedTextImportOptions( ___ ,Name,Value)
```

Description

`opts = delimitedTextImportOptions` creates a `DelimitedTextImportOptions` object with one variable.

`opts = delimitedTextImportOptions('NumVariables',numVars)` creates the object with the number of variables specified in `numVars`.

`opts = delimitedTextImportOptions( ___ ,Name,Value)` specifies additional properties for `DelimitedTextImportOptions` object using one or more name-value pair arguments.

Input Arguments

`numVars` — Number of variables
positive scalar integer

Number of variables, specified as a positive scalar integer.

Properties

Variable Properties

`VariableNames` — Variable names
cell array of character vectors | string array

Variable names, specified as a cell array of character vectors or string array. The `VariableNames` property contains the names to use when importing variables.

If the data contains `N` variables, but no variable names are detected, then the `VariableNames` property contains `{'Var1','Var2',...,'VarN'}`.

To support invalid MATLAB identifiers as variable names, such as varible names containing spaces and non-ASCII characters, set the `PreserveVariableNames` parameter to `true`.

Example: `opts.VariableNames` returns the current (detected) variable names.

Example: `opts.VariableNames(3) = {'Height'}` changes the name of the third variable to `Height`.

Data Types: `char` | `string` | `cell`

`PreserveVariableNames` — Flag to preserve variable names
`false` (default) | `true`

Flag to preserve variable names, specified as the comma-separated pair consisting of `PreserveVariableNames` and either `true`, or `false`.

- `true` — Preserve variable names that are not valid MATLAB identifiers such as variable names that include spaces and non-ASCII characters.
- `false` — Convert invalid variable names (as determined by the `isvarname` function) to valid MATLAB identifiers.

Starting in R2019b, variable names and row names can include any characters, including spaces and non-ASCII characters. Also, they can start with any characters, not just letters. Variable and row names do not have to be valid MATLAB identifiers (as determined by the `isvarname` function). To preserve these variable names and row names, set `PreserveVariableNames` to `true`.

`VariableTypes` — Variable data types
cell array of character vectors | string array

Variable data types, specified as a cell array of character vectors or string array. The `VariableTypes` property designates the data types to use when importing variables. When assigning new values, specify `VariableTypes` as a cell array of valid data type names.

To update the `VariableTypes` property, use the `setvartype` function.

Example: `opts.VariableTypes` returns the current (detected) variable data types.

Example: `opts = setvartype(opts,'Height',{'double'})` changes the data type of the variable `Height` to `double`.

Data Types: `cell` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `categorical` | `datetime`

`SelectedVariableNames` — Subset of variables to import
character vector | string scalar | cell array of character vectors | string array | array of numeric indices

Subset of variables to import, specified as a character vector, string scalar, cell array of character vectors, string array or an array of numeric indices.

`SelectedVariableNames` must be a subset of names contained in the `VariableNames` property. By default, `SelectedVariableNames` contains all the variable names from the `VariableNames` property, which means that all variables are imported.

Use the `SelectedVariableNames` property to import only the variables of interest. Specify a subset of variables using the `SelectedVariableNames` property and use `readtable` to import only that subset.

To support invalid MATLAB identifiers as variable names, such as varible names containing spaces and non-ASCII characters, set the `PreserveVariableNames` parameter to `true`.

Example: `opts.SelectedVariableNames = {'Height','LastName'}` selects only two variables, `Height` and `LastName`, for the import operation.

Example: `opts.SelectedVariableNames = [1 5]` selects only two variables, the first variable and the fifth variable, for the import operation.

Example: `T = readtable(filename,opts)` returns a table containing only the variables specified in the `SelectedVariableNames` property of the `opts` object.

Data Types: `uint16` | `uint32` | `uint64` | `char` | `string` | `cell`

`VariableOptions` — Type specific variable import options
array of variable import options objects

Type specific variable import options, returned as an array of variable import options objects. The array contains an object corresponding to each variable specified in the `VariableNames` property. Each object in the array contains properties that support the importing of data with a specific data type.

Variable options support these data types: numeric, text, `logical`, `datetime`, or `categorical`.

To query the current (or detected) options for a variable, use the `getvaropts` function.

To set and customize options for a variable, use the `setvaropts` function.

Example: `opts.VariableOptions` returns a collection of `VariableImportOptions` objects, one corresponding to each variable in the data.

Example: `getvaropts(opts,'Height')` returns the `VariableImportOptions` object for the `Height` variable.

Example: `opts = setvaropts(opts,'Height','FillValue',0)` sets the `FillValue` property for the variable `Height` to `0`.

Location Properties

`DataLines` — Data location

positive scalar integer | array of positive scalar integers

Data location, specified as a positive scalar integer or a `N-by-2` array of positive scalar integers. Specify `DataLines` using one of these forms.

Specify as	Description
`n`	Specify the *first line* that contains the data. Specifying the value using `n` sets the value of `DataLines` property to `[n inf]`. The importing function reads all rows between `n` and the end-of-file. `n` must be a positive integer greater than zero.
`[n1 n2]`	Specify the *line range* that contains the data. `n1` is the first line that contains the data and the `n2` is the last line that contains the data. Values in the array `[n1 n2]` must be nonzero positive integers and `n2` must be greater than `n1`.
`[n1 n2; n3 n4;...]`	Specify *multiple line ranges* to read with an `N-by-2` array containing `N` different line ranges. A valid array of multiple line ranges must: • Specify line ranges in an increasing order, that is the first line range specified in the array appears in the file before the other line ranges. • Contain only nonoverlapping line ranges. When specifying multiple line ranges, use `Inf` only when specifying the end of the last line range in the array. For example, `[1 3; 5 6; 8 Inf]`.

Example: `opts.DataLines = 5` sets the `DataLines` property to the value `[5 inf]`. Read all rows of data starting from row `5` to the end-of-file.

Example: `opts.DataLines = [2 6]` sets the property to read lines `2` through `6`.

Example: `opts.DataLines = [1 3; 5 6; 8 inf]` sets the property to read rows 1, 2, 3, 5, 6, and all rows between 8, and the end-of-file.

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64`

`RowNamesColumn` — Row names location
`0` (default) | positive scalar integer

Row names location, specified as a positive scalar integer. The `RowNamesColumn` property specifies the location of the column containing the row names.

If `RowNamesColumn` is specified as 0, then do not import the row names. Otherwise, import the row names from the specified column.

Example: `opts.RowNamesColumn = 2;`

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64`

`VariableNamesLine` — Variable names location
`0` (default) | positive scalar integer

Variable names location, specified as a positive scalar integer. The `VariableNamesLine` property specifies the line number where variable names are located.

If `VariableNamesLine` is specified as 0, then do not import the variable names. Otherwise, import the variable names from the specified line.

Example: `opts.VariableNamesLine = 6;`

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64`

`VariableDescriptionsLine` — Variable description location
`0` (default) | positive scalar integer

Variable description location, specified as a positive scalar integer. The `VariableDescriptionsLine` property specifies the line number where variable descriptions are located.

If `VariableDescriptionsLine` is specified as 0, then do not import the variable descriptions. Otherwise, import the variable descriptions from the specified line.

Example: `opts.VariableDescriptionsLine = 7;`

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64`

`VariableUnitsLine` — Variable units location

`0` (default) | positive scalar integer

Variable units location, specified as a positive scalar integer. The `VariableUnitsLine` property specifies the line number where variable units are located.

If `VariableUnitsLine` is specified as 0, then do not import the variable units. Otherwise, import the variable units from the specified line.

Example: `opts.VariableUnitsLine = 8;`

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64`

Delimited Text Properties

`Delimiter` — Field delimiter characters

character vector | string scalar | cell array of character vectors | string array

Field delimiter characters in a delimited text file, specified as a character vector, string scalar, cell array of character vectors, or string array.

Example: `'Delimiter','|'`

Example: `'Delimiter',{';','*'}`

Data Types: `char` | `string` | `cell`

`Whitespace` — Characters to treat as white space

character vector | string scalar

Characters to treat as white space, specified as a character vector or string scalar containing one or more characters.

Example: `'Whitespace',' _'`

Example: `'Whitespace','?!.,'`

`LineEnding` — End-of-line characters

`{'\n','\r','\r\n'}` (default) | character vector | string scalar | cell array of character vectors | string array

End-of-line characters, specified as a character vector, string scalar, cell array of character vectors, or string array.

Example: `'LineEnding','\n'`

Example: `'LineEnding','\r\n'`

Example: `'LineEnding',{'\b',':'}`

Data Types: `char` | `string` | `cell`

`CommentStyle` — Style of comments
character vector | string scalar | cell array of character vectors | string array

Style of comments, specified as a character vector, string scalar, cell array of character vectors, or string array.

For example, to ignore the text following a percent sign on the same line, specify `CommentStyle` as `'%'`.

Example: `'CommentStyle',{'/*'}`

Data Types: `char` | `string` | `cell`

`ConsecutiveDelimitersRule` — Procedure to handle consecutive delimiters
`'split'` | `'join'` | `'error'`

Procedure to handle consecutive delimiters in a delimited text file, specified as one of the values in this table.

Consecutive Delimiters Rule	Behavior
`'split'`	Split the consecutive delimiters into multiple fields.
`'join'`	Join the delimiters into one delimiter.
`'error'`	Return an error and abort the import operation.

Data Types: `char` | `string`

`LeadingDelimitersRule` — Procedure to manage leading delimiters
`'keep'` | `'ignore'` | `'error'`

Procedure to manage leading delimiters in a delimited text file, specified as one of the values in this table.

Leading Delimiters Rule	Behavior
`'keep'`	Keep the delimiter.
`'ignore'`	Ignore the delimiter.

Leading Delimiters Rule	Behavior
`'error'`	Return an error and abort the import operation.

`Encoding` — Character encoding scheme
`'system'` | `'UTF-8'` | `'ISO-8859-1'` | `'windows-1251'` | `'windows-1252'` | ...

Character encoding scheme associated with the file, specified as the comma-separated pair consisting of `'Encoding'` and `'system'` or a standard character encoding scheme name, such as one of the values in this table.

`'Big5'`	`'ISO-8859-1'`	`'windows-847'`
`'Big5-HKSCS'`	`'ISO-8859-2'`	`'windows-949'`
`'CP949'`	`'ISO-8859-3'`	`'windows-1250'`
`'EUC-KR'`	`'ISO-8859-4'`	`'windows-1251'`
`'EUC-JP'`	`'ISO-8859-5'`	`'windows-1252'`
`'EUC-TW'`	`'ISO-8859-6'`	`'windows-1253'`
`'GB18030'`	`'ISO-8859-7'`	`'windows-1254'`
`'GB2312'`	`'ISO-8859-8'`	`'windows-1255'`
`'GBK'`	`'ISO-8859-9'`	`'windows-1256'`
`'IBM866'`	`'ISO-8859-11'`	`'windows-1257'`
`'KOI8-R'`	`'ISO-8859-13'`	`'windows-1258'`
`'KOI8-U'`	`'ISO-8859-15'`	`'US-ASCII'`
	`'Macintosh'`	`'UTF-8'`
	`'Shift_JIS'`	

Example: `'Encoding','system'` uses the system default encoding.

Data Types: `char` | `string`

Replacement Rules

`MissingRule` — Procedure to manage missing data
`'fill'` | `'error'` | `'omitrow'` | `'omitvar'`

Procedure to manage missing data, specified as one of the values in this table.

Missing Rule	Behavior
`'fill'`	Replace missing data with the contents of the `FillValue` property. The `FillValue` property is specified in the `VariableImportOptions` object of the variable being imported. For more information on accessing the `FillValue` property, see `getvaropts`.
`'error'`	Stop importing and display an error message showing the missing record and field.
`'omitrow'`	Omit rows that contain missing data.
`'omitvar'`	Omit variables that contain missing data.

Example: `opts.MissingRule = 'omitrow';`uint32

Data Types: `char` | `string`

`EmptyLineRule` — Procedure to handle empty lines
`'skip'` | `'read'` | `'error'`

Procedure to handle empty lines in the data, specified as `'skip'`, `'read'`, or `'error'`. The importing function interprets white space as empty.

Empty Line Rule	Behavior
`'skip'`	Skip the empty lines.
`'read'`	Import the empty lines. The importing function parses the empty line using the values specified in `VariableWidths`, `VariableOptions`, `MissingRule`, and other relevant properties, such as `Whitespace`.
`'error'`	Display an error message and abort the import operation.

Example: `opts.EmptyLineRule = 'skip';`

Data Types: `char` | `string`

`ImportErrorRule` — Procedure to handle import errors
`'fill'` | `'error'` | `'omitrow'` | `'omitvar'`

Procedure to handle import errors, specified as one of the values in this table.

Import Error Rule	Behavior
`'fill'`	Replace the data where the error occurred with the contents of the `FillValue` property. The `FillValue` property is specified in the `VariableImportOptions` object of the variable being imported. For more information on accessing the `FillValue` property, see `getvaropts`.
`'error'`	Stop importing and display an error message showing the error-causing record and field.
`'omitrow'`	Omit rows where errors occur.
`'omitvar'`	Omit variables where errors occur.

Example: `opts.ImportErrorRule = 'omitvar';`

Data Types: `char` | `string`

`ExtraColumnsRule` — Procedure to handle extra columns
`'addvars'` | `'ignore'` | `'wrap'` | `'error'`

Procedure to handle extra columns in the data, specified as one of the values in this table.

Extra Columns Rule	Behavior
`'addvars'`	To import extra columns, create new variables. If there are `N` extra columns, then import new variables as `'ExtraVar1', 'ExtraVar2',..., 'ExtraVarN'`. NOTE: The extra columns are imported as text with data type`char`.
`'ignore'`	Ignore the extra columns of data.
`'wrap'`	Wrap the extra columns of data to new records. This action does not change the number of variables.
`'error'`	Display an error message and abort the import operation.

Data Types: `char` | `string`

Object Functions

getvaropts	Get variable import options
setvaropts	Set variable import options
setvartype	Set variable data types
preview	Preview eight rows from file using import options

Examples

Define Import Options for Variables in Delimited Text File

Define an import options object to read multiple variables from `patients.dat`.

Based on the contents of your file, define these variable properties: names, types, delimiter character, data starting location, and the extra column rule.

```
varNames = {'LastName','Gender','Age','Location','Height','Weight','Smoker'} ;
varTypes = {'char','categorical','int32','char','double','double','logical'} ;
delimiter = ',';
dataStartLine = 2;
extraColRule = 'ignore';
```

Use the `delimitedTextImportOptions` function and your variable information to initialize the import options object `opts`.

```
opts = delimitedTextImportOptions('VariableNames',varNames,...
                                  'VariableTypes',varTypes,...
                                  'Delimiter',delimiter,...
                                  'DataLines', dataStartLine,...
                                  'ExtraColumnsRule',extraColRule);
```

Use the `preview` function with the import options object to preview the data.

```
preview('patients.dat',opts)
```

```
ans=8×7 table
      LastName      Gender    Age              Location               Height    Weight
    ____________    ______    ___    ______________________________    ______    ______

    {'Smith'   }    Male      38     {'County General Hospital'  }      71        176
    {'Johnson' }    Male      43     {'VA Hospital'              }      69        163
    {'Williams'}    Female    38     {'St. Mary's Medical Center'}      64        131
```

```
    {'Jones'   }    Female    40     {'VA Hospital'              }      67       133
    {'Brown'   }    Female    49     {'County General Hospital'  }      64       119
    {'Davis'   }    Female    46     {'St. Mary's Medical Center'}      68       142
    {'Miller'  }    Female    33     {'VA Hospital'              }      64       142
    {'Wilson'  }    Male      40     {'VA Hospital'              }      68       180
```

Import the data using `readtable`.

```
T = readtable('patients.dat',opts);
whos T
```

```
  Name        Size            Bytes  Class    Attributes

  T         100x7             32332  table
```

Tips

- Introduced in:
  - R2016b — `DelimitedTextImportOptions` object
  - R2018b — `delimitedTextImportOptions` function

See Also

`detectImportOptions` | `readtable`

fixedWidthImportOptions

Import options object for fixed-width text files

Description

A `FixedWidthImportOptions` object enables you to specify how MATLAB imports fixed-width tabular data from text files. The object contains properties that control the data import process, including the handling of errors and missing data.

Creation

You can create a `FixedWidthImportOptions` object using either the `fixedWidthImportOptions` function (described here) or the `detectImportOptions` function:

- Use `fixedWidthImportOptions` to define the import properties based on your import requirements.
- Use `detectImportOptions` to detect and populate the import properties based on the contents of the fixed-width text file specified in `filename`.

  ```
  opts = detectImportOptions(filename)
  ```

Syntax

```
opts = fixedWidthImportOptions
opts = fixedWidthImportOptions('NumVariables',numVars)
opts = fixedWidthImportOptions( ___ ,Name,Value)
```

Description

`opts = fixedWidthImportOptions` creates a `FixedWidthImportOptions` object with one variable.

`opts = fixedWidthImportOptions('NumVariables',numVars)` creates the object with the number of variables specified in `numVars`.

`opts = fixedWidthImportOptions( ___ ,Name,Value)` specifies additional properties for `FixedWidthImportOptions` object using one or more name-value pair arguments.

Input Arguments

`numVars` — Number of variables
positive scalar integer

Number of variables, specified as a positive scalar integer.

Properties

Variable Properties

`VariableNames` — Variable names
cell array of character vectors | string array

Variable names, specified as a cell array of character vectors or string array. The `VariableNames` property contains the names to use when importing variables.

If the data contains `N` variables, but no variable names are detected, then the `VariableNames` property contains `{'Var1','Var2',...,'VarN'}`.

To support invalid MATLAB identifiers as variable names, such as varible names containing spaces and non-ASCII characters, set the `PreserveVariableNames` parameter to `true`.

Example: `opts.VariableNames` returns the current (detected) variable names.

Example: `opts.VariableNames(3) = {'Height'}` changes the name of the third variable to `Height`.

Data Types: `char` | `string` | `cell`

`PreserveVariableNames` — Flag to preserve variable names
`false` (default) | `true`

Flag to preserve variable names, specified as the comma-separated pair consisting of `PreserveVariableNames` and either `true`, or `false`.

- `true` — Preserve variable names that are not valid MATLAB identifiers such as variable names that include spaces and non-ASCII characters.
- `false` — Convert invalid variable names (as determined by the `isvarname` function) to valid MATLAB identifiers.

Starting in R2019b, variable names and row names can include any characters, including spaces and non-ASCII characters. Also, they can start with any characters, not just letters. Variable and row names do not have to be valid MATLAB identifiers (as determined by the `isvarname` function). To preserve these variable names and row names, set `PreserveVariableNames` to `true`.

`VariableWidths` — Field widths of variables
vector of positive integer values

Field widths of variables in a fixed-width text file, specified as a vector of positive integer values. Each positive integer in the vector corresponds to the number of characters in a field that makes up the variable. The `VariableWidths` property contains an entry corresponding to each variable specified in the `VariableNames` property.

`VariableTypes` — Variable data types
cell array of character vectors | string array

Variable data types, specified as a cell array of character vectors or string array. The `VariableTypes` property designates the data types to use when importing variables. When assigning new values, specify `VariableTypes` as a cell array of valid data type names.

To update the `VariableTypes` property, use the `setvartype` function.

Example: `opts.VariableTypes` returns the current (detected) variable data types.

Example: `opts = setvartype(opts,'Height',{'double'})` changes the data type of the variable `Height` to `double`.

Data Types: `cell` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `categorical` | `datetime`

`SelectedVariableNames` — Subset of variables to import
character vector | string scalar | cell array of character vectors | string array | array of numeric indices

Subset of variables to import, specified as a character vector, string scalar, cell array of character vectors, string array or an array of numeric indices.

`SelectedVariableNames` must be a subset of names contained in the `VariableNames` property. By default, `SelectedVariableNames` contains all the variable names from the `VariableNames` property, which means that all variables are imported.

Use the `SelectedVariableNames` property to import only the variables of interest. Specify a subset of variables using the `SelectedVariableNames` property and use `readtable` to import only that subset.

To support invalid MATLAB identifiers as variable names, such as varible names containing spaces and non-ASCII characters, set the `PreserveVariableNames` parameter to `true`.

Example: `opts.SelectedVariableNames = {'Height','LastName'}` selects only two variables, `Height` and `LastName`, for the import operation.

Example: `opts.SelectedVariableNames = [1 5]` selects only two variables, the first variable and the fifth variable, for the import operation.

Example: `T = readtable(filename,opts)` returns a table containing only the variables specified in the `SelectedVariableNames` property of the `opts` object.

Data Types: `uint16` | `uint32` | `uint64` | `char` | `string` | `cell`

`VariableOptions` — Type specific variable import options
array of variable import options objects

Type specific variable import options, returned as an array of variable import options objects. The array contains an object corresponding to each variable specified in the `VariableNames` property. Each object in the array contains properties that support the importing of data with a specific data type.

Variable options support these data types: numeric, text, `logical`, `datetime`, or `categorical`.

To query the current (or detected) options for a variable, use the `getvaropts` function.

To set and customize options for a variable, use the `setvaropts` function.

Example: `opts.VariableOptions` returns a collection of `VariableImportOptions` objects, one corresponding to each variable in the data.

Example: `getvaropts(opts,'Height')` returns the `VariableImportOptions` object for the `Height` variable.

Example: `opts = setvaropts(opts,'Height','FillValue',0)` sets the `FillValue` property for the variable `Height` to `0`.

Location Properties

`DataLines` — Data location

positive scalar integer | array of positive scalar integers

Data location, specified as a positive scalar integer or a `N-by-2` array of positive scalar integers. Specify `DataLines` using one of these forms.

Specify as	Description
`n`	Specify the *first line* that contains the data. Specifying the value using `n` sets the value of `DataLines` property to `[n inf]`. The importing function reads all rows between `n` and the end-of-file. `n` must be a positive integer greater than zero.
`[n1 n2]`	Specify the *line range* that contains the data. `n1` is the first line that contains the data and the `n2` is the last line that contains the data. Values in the array `[n1 n2]` must be nonzero positive integers and `n2` must be greater than `n1`.
`[n1 n2; n3 n4;...]`	Specify *multiple line ranges* to read with an `N-by-2` array containing `N` different line ranges. A valid array of multiple line ranges must: • Specify line ranges in an increasing order, that is the first line range specified in the array appears in the file before the other line ranges. • Contain only nonoverlapping line ranges. When specifying multiple line ranges, use `Inf` only when specifying the end of the last line range in the array. For example, `[1 3; 5 6; 8 Inf]`.

Example: `opts.DataLines = 5` sets the `DataLines` property to the value `[5 inf]`. Read all rows of data starting from row 5 to the end-of-file.

Example: `opts.DataLines = [2 6]` sets the property to read lines 2 through 6.

Example: `opts.DataLines = [1 3; 5 6; 8 inf]` sets the property to read rows 1, 2, 3, 5, 6, and all rows between 8, and the end-of-file.

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64`

`RowNamesColumn` — Row names location
`0` (default) | positive scalar integer

Row names location, specified as a positive scalar integer. The `RowNamesColumn` property specifies the location of the column containing the row names.

If `RowNamesColumn` is specified as 0, then do not import the row names. Otherwise, import the row names from the specified column.

Example: `opts.RowNamesColumn = 2;`

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64`

`VariableNamesLine` — Variable names location
`0` (default) | positive scalar integer

Variable names location, specified as a positive scalar integer. The `VariableNamesLine` property specifies the line number where variable names are located.

If `VariableNamesLine` is specified as 0, then do not import the variable names. Otherwise, import the variable names from the specified line.

Example: `opts.VariableNamesLine = 6;`

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64`

`VariableDescriptionsLine` — Variable description location
`0` (default) | positive scalar integer

Variable description location, specified as a positive scalar integer. The `VariableDescriptionsLine` property specifies the line number where variable descriptions are located.

If `VariableDescriptionsLine` is specified as 0, then do not import the variable descriptions. Otherwise, import the variable descriptions from the specified line.

Example: `opts.VariableDescriptionsLine = 7;`

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64`

`VariableUnitsLine` — Variable units location
`0` (default) | positive scalar integer

Variable units location, specified as a positive scalar integer. The `VariableUnitsLine` property specifies the line number where variable units are located.

If `VariableUnitsLine` is specified as 0, then do not import the variable units. Otherwise, import the variable units from the specified line.

Example: `opts.VariableUnitsLine = 8;`

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64`

Delimited Text Properties

`Whitespace` — Characters to treat as white space
character vector | string scalar

Characters to treat as white space, specified as a character vector or string scalar containing one or more characters.

Example: `'Whitespace',' _'`

Example: `'Whitespace','?!.,'`

`LineEnding` — End-of-line characters
`{'\n','\r','\r\n'}` (default) | character vector | string scalar | cell array of character vectors | string array

End-of-line characters, specified as a character vector, string scalar, cell array of character vectors, or string array.

Example: `'LineEnding','\n'`

Example: `'LineEnding','\r\n'`

Example: `'LineEnding',{'\b',':'}`

Data Types: `char` | `string` | `cell`

`CommentStyle` — Style of comments
character vector | string scalar | cell array of character vectors | string array

Style of comments, specified as a character vector, string scalar, cell array of character vectors, or string array.

For example, to ignore the text following a percent sign on the same line, specify `CommentStyle` as `'%'`.

Example: `'CommentStyle',{'/*'}`

Data Types: `char` | `string` | `cell`

`Encoding` — Character encoding scheme
`'system'` | `'UTF-8'` | `'ISO-8859-1'` | `'windows-1251'` | `'windows-1252'` | ...

Character encoding scheme associated with the file, specified as the comma-separated pair consisting of `'Encoding'` and `'system'` or a standard character encoding scheme name, such as one of the values in this table.

'Big5'	'ISO-8859-1'	'windows-847'
'Big5-HKSCS'	'ISO-8859-2'	'windows-949'
'CP949'	'ISO-8859-3'	'windows-1250'
'EUC-KR'	'ISO-8859-4'	'windows-1251'
'EUC-JP'	'ISO-8859-5'	'windows-1252'
'EUC-TW'	'ISO-8859-6'	'windows-1253'
'GB18030'	'ISO-8859-7'	'windows-1254'
'GB2312'	'ISO-8859-8'	'windows-1255'
'GBK'	'ISO-8859-9'	'windows-1256'
'IBM866'	'ISO-8859-11'	'windows-1257'
'KOI8-R'	'ISO-8859-13'	'windows-1258'
'KOI8-U'	'ISO-8859-15'	'US-ASCII'
	'Macintosh'	'UTF-8'
	'Shift_JIS'	

Example: `'Encoding','system'` uses the system default encoding.

Data Types: `char` | `string`

Replacement Rules

`PartialFieldRule` — Procedure to handle partial fields

`'keep'` | `'fill'` | `'omitrow'` | `'omitvar'` | `'wrap'` | `'error'`

Procedure to handle partial fields in the data, specified as one of the values in this table.

Partial Field Rule	Behavior
`'keep'`	Keep the partial field data and convert the text to the appropriate data type. In some cases, when the importing function is unable to interpret the partial data, a conversion error might occur.
`'fill'`	Replace missing data with the contents of the `FillValue` property. The `FillValue` property is specified in the `VariableImportOptions` object of the variable being imported. For more information on accessing the `FillValue` property, see `getvaropts`.
`'omitrow'`	Omit rows that contain partial data.
`'omitvar'`	Omit variables that contain partial data.
`'wrap'`	Begin reading the next line of characters.
`'error'`	Display an error message and abort the import operation.

Example: `opts.PartialFieldRule = 'keep';`

Data Types: `char` | `string`

`EmptyLineRule` — Procedure to handle empty lines

`'skip'` | `'read'` | `'error'`

Procedure to handle empty lines in the data, specified as `'skip'`, `'read'`, or `'error'`. The importing function interprets white space as empty.

Empty Line Rule	Behavior
`'skip'`	Skip the empty lines.

Empty Line Rule	Behavior
`'read'`	Import the empty lines. The importing function parses the empty line using the values specified in `VariableWidths`, `VariableOptions`, `MissingRule`, and other relevant properties, such as `Whitespace`.
`'error'`	Display an error message and abort the import operation.

Example: `opts.EmptyLineRule = 'skip';`

Data Types: `char` | `string`

`MissingRule` — Procedure to manage missing data
`'fill'` | `'error'` | `'omitrow'` | `'omitvar'`

Procedure to manage missing data, specified as one of the values in this table.

Missing Rule	Behavior
`'fill'`	Replace missing data with the contents of the `FillValue` property. The `FillValue` property is specified in the `VariableImportOptions` object of the variable being imported. For more information on accessing the `FillValue` property, see `getvaropts`.
`'error'`	Stop importing and display an error message showing the missing record and field.
`'omitrow'`	Omit rows that contain missing data.
`'omitvar'`	Omit variables that contain missing data.

Example: `opts.MissingRule = 'omitrow';`uint32

Data Types: `char` | `string`

`ImportErrorRule` — Procedure to handle import errors
`'fill'` | `'error'` | `'omitrow'` | `'omitvar'`

Procedure to handle import errors, specified as one of the values in this table.

Import Error Rule	Behavior
`'fill'`	Replace the data where the error occurred with the contents of the `FillValue` property. The `FillValue` property is specified in the `VariableImportOptions` object of the variable being imported. For more information on accessing the `FillValue` property, see `getvaropts`.
`'error'`	Stop importing and display an error message showing the error-causing record and field.
`'omitrow'`	Omit rows where errors occur.
`'omitvar'`	Omit variables where errors occur.

Example: `opts.ImportErrorRule = 'omitvar';`

Data Types: `char` | `string`

`ExtraColumnsRule` — Procedure to handle extra columns
`'addvars'` | `'ignore'` | `'wrap'` | `'error'`

Procedure to handle extra columns in the data, specified as one of the values in this table.

Extra Columns Rule	Behavior
`'addvars'`	To import extra columns, create new variables. If there are `N` extra columns, then import new variables as `'ExtraVar1', 'ExtraVar2',..., 'ExtraVarN'`. NOTE: The extra columns are imported as text with data type`char`.
`'ignore'`	Ignore the extra columns of data.
`'wrap'`	Wrap the extra columns of data to new records. This action does not change the number of variables.
`'error'`	Display an error message and abort the import operation.

Data Types: `char` | `string`

Object Functions

getvaropts	Get variable import options
setvaropts	Set variable import options
setvartype	Set variable data types
preview	Preview eight rows from file using import options

Examples

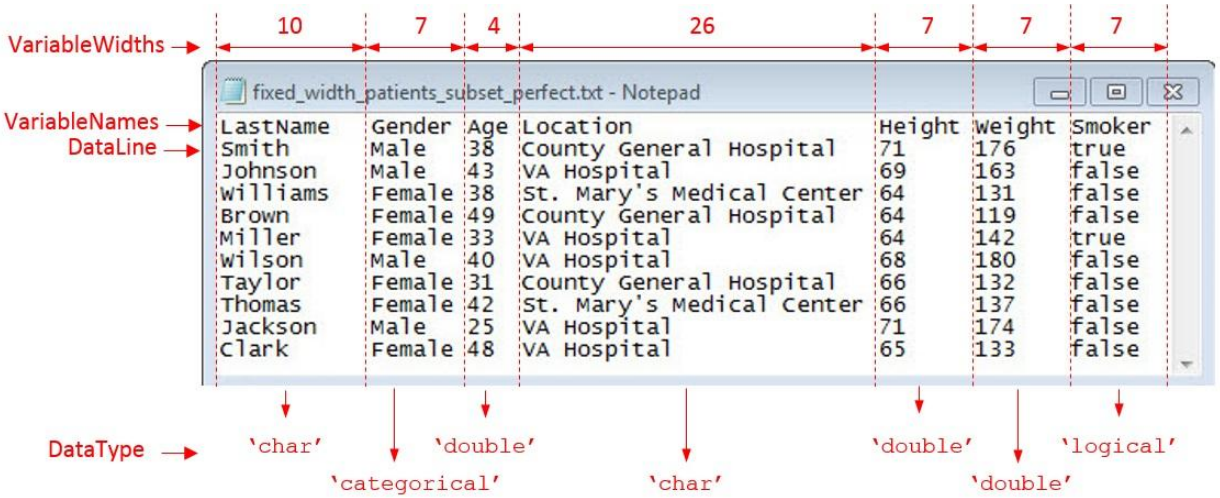
Define Import Options Object to Read Fixed-Width Text File

Examine a fixed-width formatted text file, initialize an import options object, and use the object to import the table from the text file.

Load and Preview Fixed-Width Text File

Load the file `fixed_width_patients_subset_perfect.txt` and preview its contents in a text editor. The screen shot shows that the file contains fixed-width formatted data.

```
filename = fullfile(matlabroot,'examples','matlab','fixed_width_patients_subset_perfect
```



Examine and Extract Properties of Fixed-Width File

The fixed-width text file has tabular data organized by starting location, number of variables, variable names, and variable widths. Capture these properties and the desired data type for the variables.

```
DataStartLine = 2;
NumVariables = 7;
VariableNames  = {'LastName','Gender','Age','Location','Height',...
                                                 'Weight','Smoker'};
VariableWidths = [ 10,     7,     4,    26,     7, ...
                                                    7,     7 ] ;
DataType       = {'char','categorical','double','char','double',...
                                                'double','logical'};
```

Initialize and Configure `FixedWidthImportOptions` Object

Initialize a `FixedWidthImportOptions` object and configure its properties to match the properties of the data in `fixed_width_patients_subset_perfect.txt`.

```
opts = fixedWidthImportOptions('NumVariables',NumVariables,...
                               'DataLines',DataStartLine,...
                               'VariableNames',VariableNames,...
                               'VariableWidths',VariableWidths,...
                               'VariableTypes',DataType);
```

Import Table

Use `readtable` with the `FixedWidthImportOptions` object to import the table.

```
T = readtable(filename,opts)
```

```
T=10×7 table
     LastName     Gender    Age             Location              Height    Weight    S
    __________    ______    ___    ____________________________    ______    ______    _

    'Smith'       Male      38     'County General Hospital'        71       176      t
    'Johnson'     Male      43     'VA Hospital'                    69       163      t
    'Williams'    Female    38     'St. Mary's Medical Center'      64       131      t
    'Brown'       Female    49     'County General Hospital'        64       119      t
    'Miller'      Female    33     'VA Hospital'                    64       142      t
    'Wilson'      Male      40     'VA Hospital'                    68       180      t
    'Taylor'      Female    31     'County General Hospital'        66       132      t
    'Thomas'      Female    42     'St. Mary's Medical Center'      66       137      t
    'Jackson'     Male      25     'VA Hospital'                    71       174      t
```

```
    'Clark'       Female    48     'VA Hospital'                    65       133      
```

Import Messy Data from Fixed-Width Formatted Text File

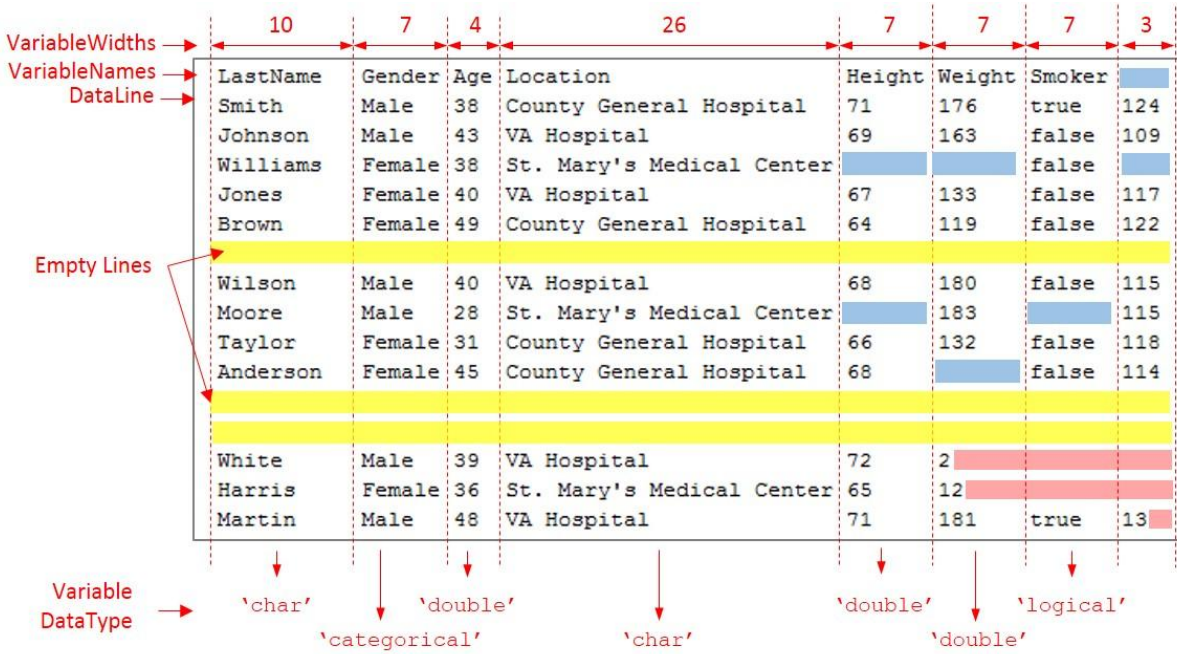
Define an import options object to import messy data from a fixed-width formatted text file. Configure the object to handle the messy data and use it to import the table.

Load and Preview Fixed-Width Text File

Load the file `fixed_width_patients_subset_messy.txt` and preview its contents in a text editor. A screen shot is shown below. The screen shot shows that the file contains:

- Empty lines – Lines 7, 12, and 13
- An extra column – Column 8
- Missing data – Lines 1, 4, 9 and 11
- Partial fields – Last 3 rows

```
filename = fullfile(matlabroot,'examples','matlab','fixed_width_patients_subset_messy.t
```



Examine and Capture Properties of Fixed-Width File

The fixed-width text file has tabular data organized by the staring location, number of variables, variable names, and variable widths. Capture these properties and the data type you want to use for the variables.

```
DataStartLine = 2;
NumVariables = 7;
VariableNames  = {'LastName','Gender','Age','Location','Height',...
                                                 'Weight','Smoker'};
VariableWidths = [ 10,     7,     4,    26,     7, ...
                                                     7,     7 ] ;
DataType       = {'char','categorical','double','char','double',...
                                                 'double','logical'};
```

Initialize `FixedWidthImportOptions` Object and Set Up Variable Properties

Initialize a `FixedWidthImportOptions` object and configure its properties to match the properties of the data.

```
opts = fixedWidthImportOptions('NumVariables',NumVariables,...
                               'DataLines',DataStartLine,...
                               'VariableNames',VariableNames,...
                               'VariableWidths',VariableWidths,...
                               'VariableTypes',DataType);
```

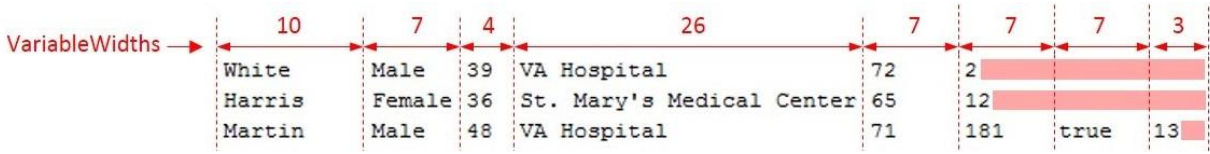
Set Up EmptyLinesRule, Missing Rule, and ExtraColumnsRule

Read the empty lines in the data by setting the `EmptyLineRule` to `'read'`. Next, fill the missing instances with predefined values by setting the `MissingRule` to `'fill'`. Finally, to ignore the extra column during the import, set the `ExtraColumnsRule` to `'ignore'`. For more information on the properties and their values, see documentation for `FixedWidthImportOptions`.

```
opts.EmptyLineRule = 'read';
opts.MissingRule = 'fill';
opts.ExtraColumnsRule ='ignore';
```

Set Up PartialFieldRule

Partial fields occur when the importing function reaches the end-of-line character before the full variable width is traversed. For example, in this preview, the last three rows from the file `fixed_width_patients_subset_messy.txt`. Here, in the last row of the last column, the end-of-line character appears after two places from the start of the field, before the full variable-width of three is reached.



This occurrence of a partial field sometimes can indicate an error. Therefore, use the `PartialFieldRule` to decide how to handle this data. To keep the partial field data and convert it to the appropriate data type, set the `PartialFieldRule` to `'keep'`. For more information on the `PartialFieldRule`, see documentation for `FixedWidthImportOptions`.

```
opts.PartialFieldRule = 'keep';
```

Import Table

Import the table by using `readtable` function and the `FixedWidthImportOptions` object and preview the data.

```
T = readtable(filename,opts)
```

```
T=15×7 table
     LastName       Gender       Age             Location              Height    Weight
    __________    ___________    ___    ____________________________    ______    ______

    'Smith'       Male            38    'County General Hospital'        71       176
    'Johnson'     Male            43    'VA Hospital'                    69       163
    'Williams'    Female          38    'St. Mary's Medical Center'     NaN       NaN
    'Jones'       Female          40    'VA Hospital'                    67       133
    'Brown'       Female          49    'County General Hospital'        64       119
    ''            <undefined>    NaN    ''                              NaN       NaN
    'Wilson'      Male            40    'VA Hospital'                    68       180
    'Moore'       Male            28    'St. Mary's Medical Center'     NaN       183
    'Taylor'      Female          31    'County General Hospital'        66       132
    'Anderson'    Female          45    'County General Hospital'        68       NaN
    ''            <undefined>    NaN    ''                              NaN       NaN
    ''            <undefined>    NaN    ''                              NaN       NaN
    'White'       Male            39    'VA Hospital'                    72         2
    'Harris'      Female          36    'St. Mary's Medical Center'      65        12
    'Martin'      Male            48    'VA Hospital'                    71       181
```

Tips

- Introduced in:
  - R2017a — `FixedWidthImportOptions` object
  - R2018b — `fixedWidthImportOptions` function

See Also

`detectImportOptions` | `readtable`

getvaropts

Package: `matlab.io.text`

Get variable import options

Syntax

```
varOpts = getvaropts(opts,selection)
```

Description

`varOpts = getvaropts(opts,selection)` returns the variable import options for the variables specified in `selection`.

Examples

Get Import Options for Variable

Access the variable import options object that controls the importing of a variable using the `getvaropts` function.

Create import options from a file. The options object is a collection of individual variable import options.

```
opts = detectImportOptions('patients.xls')
```

```
opts 
  SpreadsheetImportOptions with properties:

   Sheet Properties:
                        Sheet: ''

   Replacement Properties:
                   MissingRule: 'fill'
               ImportErrorRule: 'fill'
```

```
   Variable Import Properties: Set types by name using setvartype
                VariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
                VariableTypes: {'char', 'char', 'double' ... and 7 more}
        SelectedVariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
              VariableOptions: Show all 10 VariableOptions
    Access VariableOptions sub-properties using setvaropts/getvaropts
        PreserveVariableNames: false

   Range Properties:
                    DataRange: 'A2' (Start Cell)
           VariableNamesRange: 'A1'
                RowNamesRange: ''
           VariableUnitsRange: ''
    VariableDescriptionsRange: ''
    To display a preview of the table, use preview
```

Get the variable import options object for the specified variable.

```
varOpts = getvaropts(opts, 'Systolic')
```

```
varOpts 
  NumericVariableImportOptions with properties:

   Variable Properties:
                  Name: 'Systolic'
                  Type: 'double'
             FillValue: NaN
        TreatAsMissing: {}
             QuoteRule: 'remove'
              Prefixes: {}
              Suffixes: {}
        EmptyFieldRule: 'missing'

   Numeric Options:
     ExponentCharacter: 'eEdD'
      DecimalSeparator: '.'
    ThousandsSeparator: ''
        TrimNonNumeric: 0
```

To modify the variable import options, see the `setvaropts` and `setvartype` function reference pages.

Input Arguments

opts — File import options
SpreadsheetImportOptions | DelimitedTextImportOptions | FixedWidthImportOptions

File import options, specified as a SpreadsheetImportOptions, DelimitedTextImportOptions, or a FixedWidthImportOptions object created by the detectImportOptions function. The opts object contains properties that control the data import process, such as variable properties, data location properties, replacement rules, and others.

selection — Selected variables
character vector | string scalar | cell array of character vector | string array | array of indices | logical array

Selected variables, specified as a character vector, string scalar, cell array of character vectors, string array, array of numeric indices, or a logical array.

Variable names (or indices) must be a subset of the names contained in the VariableNames property of the opts object.

Example: 'Height'

Example: {'Height','LastName'}

Example: [5 9]

Data Types: char | string | cell | uint64

Output Arguments

varOpts — Type dependent options for selected variables
array of variable import options objects

Type dependent options for selected variables, returned as an array of variable import options objects. The array contains an object corresponding to each variable specified in the selection argument. Depending on the data types of the variables, each object in the array is one of these types: numeric, text, logical, datetime, and categorical.

You can modify properties of the individual objects.

- To modify the `Type` property, use the `setvartype` function.
- To modify other properties, use the `setvaropts` function.

See Also

`detectImportOptions` | `preview` | `setvaropts` | `setvartype`

Introduced in R2016b

preview

Package: `matlab.io.text`

Preview eight rows from file using import options

Syntax

```
T = preview(filename,opts)
```

Description

`T = preview(filename,opts)` returns a table containing the first eight rows of data in `filename`, using the import options `opts`.

- The value of the `SelectedVariableNames` property of the import options object `opts` selects the variables that appear in the table `T`.
- The preview function returns a subset of the data that the `readtable` function returns with the import options object `opts`.
- Usually, the output of the `preview` function contains eight rows of data. However, in some instances the number of rows differs depending on property values defined in the import options object `opts`. For more details see, Tips.

Examples

Preview Data Before Importing Full Table

First, create an import options object. Next, use the import options to get the first eight rows of the table, and then generate a preview showing only the variables of interest.

Create an import options object for a file.

```
filename = 'outages.csv';
opts = detectImportOptions(filename);
```

Preview the table.

```
preview(filename,opts)
```

```
ans=8×6 table
      Region            OutageTime          Loss     Customers       RestorationTime
    ___________    ____________________    ______    __________    ___________________

    'SouthWest'    01-Feb-2002 12:18:00    458.98    1.8202e+06    07-Feb-2002 16:50:00
    'SouthEast'    23-Jan-2003 00:49:00    530.14    2.1204e+05                     NaT
    'SouthEast'    07-Feb-2003 21:15:00     289.4    1.4294e+05    17-Feb-2003 08:14:00
    'West'         06-Apr-2004 05:44:00    434.81    3.4037e+05    06-Apr-2004 06:10:00
    'MidWest'      16-Mar-2002 06:18:00    186.44    2.1275e+05    18-Mar-2002 23:23:00
    'West'         18-Jun-2003 02:49:00         0             0    18-Jun-2003 10:54:00
    'West'         20-Jun-2004 14:39:00    231.29           NaN    20-Jun-2004 19:16:00
    'West'         06-Jun-2002 19:28:00    311.86           NaN    07-Jun-2002 00:51:00
```

Select which variables to include in the preview and read operations.

```
opts.SelectedVariableNames = {'Region','OutageTime'};
```

Preview the first eight rows and selected variables.

```
preview(filename,opts)
```

```
ans=8×2 table
      Region            OutageTime
    ___________    ____________________

    'SouthWest'    01-Feb-2002 12:18:00
    'SouthEast'    23-Jan-2003 00:49:00
    'SouthEast'    07-Feb-2003 21:15:00
    'West'         06-Apr-2004 05:44:00
    'MidWest'      16-Mar-2002 06:18:00
    'West'         18-Jun-2003 02:49:00
    'West'         20-Jun-2004 14:39:00
    'West'         06-Jun-2002 19:28:00
```

Input Arguments

`filename` — Name of file to read
character vector | string scalar

Name of the file to read, specified as a character vector or a string scalar.

Example: `'myFile.xlsx'`

Data Types: `char` | `string`

`opts` — File import options
`SpreadsheetImportOptions` | `DelimitedtextImportOptions` | `FixedWidthImportOptions`

File import options, specified as an `SpreadsheetImportOptions`, `DelimitedTextImportOptions` or `FixedWidthImportOptions` object created by the `detectImportOptions` function. The `opts` object contains properties that control the data import process. For more information on properties for individual objects, see the appropriate object page.

Type of Files	Output
Spreadsheet files	`SpreadsheetImportOptions` object
Text files	`DelimitedTextImportOptions` object
Fixed-width text files	`FixedWidthImportOptions` object

Algorithms

- The `preview` function returns fewer than eight rows in these cases:
  - When the file contains fewer than eight rows of data.
  - When the `DataLines` or the `DataRange` properties of the `opts` object specify less than eight rows.
  - When a combination of the properties of the `opts` object result in less than eight rows of data. For example, `preview` returns less than eight rows when the value of the `EmptyLineRule`, `ImportErrorRule` or the `MissingRule` of the `opts` object causes the importing function to omit data.

- When the `DataLines` or the `DataRange` property of the `opts` object specifies multiple row ranges and the first row range spans fewer than eight rows, then `preview` returns the rows from the first row range followed by rows from successive row ranges until it reaches a total of eight rows.

See Also

`detectImportOptions` | `getvaropts` | `readtable` | `setvaropts` | `setvartype`

Introduced in R2018a

setvaropts

Package: `matlab.io.text`

Set variable import options

Syntax

```
opts = setvaropts(opts,Name,Value)
opts = setvaropts(opts,selection,Name,Value)
```

Description

`opts = setvaropts(opts,Name,Value)` updates all the variables in the `opts` object based on the specifications in the `Name,Value` arguments and returns the `opts` object.

`opts = setvaropts(opts,selection,Name,Value)` updates and returns `opts` for the variables specified in the `selection` argument, based on the specifications in the `Name,Value` arguments.

Examples

Set Options for Selected Variables

Create an import options object, set the options for selected variables, and import the data using the tailored options and the `readtable` function.

Create an options object for the spreadsheet `patients.xls`.

```
opts = detectImportOptions('patients.xls');
```

Set the `FillValue` property for the `Smoker`, `Diastolic`, and `Systolic` variables.

```
opts = setvaropts(opts,'Smoker','FillValue',false);
opts = setvaropts(opts,{'Diastolic','Systolic'},'FillValue',0);
```

Select the variables you want to import.

```
opts.SelectedVariableNames = {'Smoker','Diastolic','Systolic'};
```

Import the variables and display a summary.

```
T = readtable('patients.xls',opts);
summary(T)
```

```
Variables:

    Smoker: 100x1 logical

        Values:

            True        34
            False       66

    Diastolic: 100x1 double

        Values:

            Min            68
            Median       81.5
            Max            99

    Systolic: 100x1 double

        Values:

            Min          109
            Median       122
            Max          138
```

Set Options for Numeric Data Containing Missing or Incomplete Records

Importing data that has missing or incomplete fields requires recognizing the missing instances and deciding how the missing instances will be imported. Use `importOptions` to capture both these decisions and fetch the data using `readtable`.

Create an import options object for the file, update properties that control the import of missing data, and then use `readtable` to import the data. Note that the dataset

`airlinesmall.csv` has two numeric variables `ArrDelay` and `DepDelay`, that contain missing data indicated by `NA`.

	O	P	Q	R	S
1	ArrDelay	DepDelay	Origin	Dest	Distance
167	-1	0	MEM	JAN	189
168	102	105	MEM	CVG	403
169	-11	0	ABE	DTW	424
170	NA	NA	MSP	BUF	734
171	-1	0	MEM	BNA	200
172	0	0	STL	DTW	440
173	1	0	VPS	MEM	369
174	1	0	MSP	IAD	908
175	14	0	CLE	DTW	95
176	-14	0	JAN	MEM	189
177	5	0	SEA	PDX	129
178	25	38	MSP	ATW	236
179	0	-5	DTW	CAK	133
180	NA	NA	MIA	MCO	193
181	6	0	DCA	LGA	214

Create an import options object from the file.

```
opts = detectImportOptions('airlinesmall.csv');
```

Use the `TreatAsMissing` property to specify the characters in the data that are place holders for missing instances. In this example, the two numeric variables `ArrDelay` and `DepDelay` contain missing fields that contain the text `NA`.

```
opts = setvaropts(opts,{'ArrDelay','DepDelay'},'TreatAsMissing','NA');
```

Specify the action for the importing function to take when importing missing instances. See `ImportOptions` properties page for more options.

```
opts.MissingRule = 'fill';
```

Specify the value to use when the importing function finds a missing instance. Here the missing instances in variables `ArrDelay` and `DepDelay` are replaced by `0`.

```
opts = setvaropts(opts,{'ArrDelay','DepDelay'},'FillValue',0);
```

Select the variables you want to work with and import them using `readtable`.

```
opts.SelectedVariableNames = {'ArrDelay','DepDelay'};
T = readtable('airlinesmall.csv',opts);
```

Examine the values in `ArrDelay` and `DepDelay`. Verify that the importing function replaced the missing values denoted by `NA`.

```
T(166:180,:)
```

```
ans=15×2 table
    ArrDelay    DepDelay
    ________    ________

       -1           0
      102         105
      -11           0
        0           0
       -1           0
        0           0
        1           0
        1           0
       14           0
      -14           0
        5           0
       25          38
        0          -5
        0           0
        6           0
```

Set Options When Importing Text Data

Use the `setvaropts` function to update properties that control the import of text data. First, get the import options object for the file. Next, examine and update the options for the text variables. Finally, import the variables using the `readtable` function.

Preview the data in `patients.xls`. Notice the text data in the column `LastName`. Only a preview of the first 10 rows is shown here.

	A	B	C	D
1	LastName	Gender	Age	Location
2	Smith	Male	38	County General Hospital
3	Johnson	Male	43	VA Hospital
4	Williams	Female	38	St. Mary's Medical Center
5	Jones	Female	40	VA Hospital
6	Brown	Female	49	County General Hospital
7	Davis	Female	46	St. Mary's Medical Center
8	Miller	Female	33	VA Hospital
9	Wilson	Male	40	VA Hospital
10	Moore	Male	28	St. Mary's Medical Center

Get the import options object.

```
opts = detectImportOptions('patients.xls');
```

Get and examine the `VariableImportOptions` for variable `LastName`.

```
getvaropts(opts,'LastName')
```

```
ans 
  TextVariableImportOptions with properties:

   Variable Properties:
              Name: 'LastName'
              Type: 'char'
          FillValue: ''
    TreatAsMissing: {}
          QuoteRule: 'remove'
           Prefixes: {}
           Suffixes: {}
    EmptyFieldRule: 'missing'

   String Options:
    WhitespaceRule: 'trim'
```

Set the data type of the variable to `string`.

```
opts = setvartype(opts,'LastName','string');
```

Set the `FillValue` property of the variable to replace missing values with `'NoName'`.

```
opts = setvaropts(opts,'LastName','FillValue','NoName');
```

Select, read, and display a preview of the first 10 rows of the variable.

```
opts.SelectedVariableNames = 'LastName';
T = readtable('patients.xls',opts);
T.LastName(1:10)
```

```
ans = 10x1 string array
    "Smith"
    "Johnson"
    "Williams"
    "Jones"
    "Brown"
    "Davis"
    "Miller"
    "Wilson"
    "Moore"
    "Taylor"
```

Set Options When Importing Logical Data

Use the `setvaropts` function to update properties that control the import of `logical` data. First, get the import options object for the file. Next, examine and update the options for the logical variables. Finally, import the variables using the `readtable` function.

Preview the data in `airlinesmall_subset.xlsx`. Notice the logical data in the column `Cancelled`. Only a preview of rows 30 to 40 is shown here.

	R	S	T	U	V	
1	Dest	Distance	TaxiIn	TaxiOut	Cancelled	Ca
30	LAX	373	7	11	0	
31	LGA	214	4	11	0	
32	LGA	733	5	10	0	
33	MAF	429	0	0	1	
34	MCI	1499	6	13	0	
35	MCO	758	9	9	0	
36	MDW	307	8	15	0	
37	MEM	200	3	9	0	
38	MFE	316	3	10	0	
39	MIA	1090	7	85	0	
40	MLB	443	5	14	0	
41	MSP	680	3	10	0	

Get the import options object.

```
opts = detectImportOptions('airlinesmall_subset.xlsx');
```

Get and examine the `VariableImportOptions` for variable `Cancelled`.

```
getvaropts(opts,'Cancelled')
```

```
ans = 
  NumericVariableImportOptions with properties:

   Variable Properties:
                   Name: 'Cancelled'
                   Type: 'double'
              FillValue: NaN
        TreatAsMissing: {}
              QuoteRule: 'remove'
               Prefixes: {}
               Suffixes: {}
        EmptyFieldRule: 'missing'

   Numeric Options:
     ExponentCharacter: 'eEdD'
```

```
      DecimalSeparator: '.'
    ThousandsSeparator: ''
        TrimNonNumeric: 0
```

Set the data type of the variable to `logical`.

```
opts = setvartype(opts,'Cancelled','logical');
```

Set the `FillValue` property of the variable to replace missing values with `true`.

```
opts = setvaropts(opts,'Cancelled','FillValue',true);
```

Select, read, and display a summary of the variable.

```
opts.SelectedVariableNames = 'Cancelled';
T = readtable('airlinesmall_subset.xlsx',opts);
summary(T)
```

```
Variables:

    Cancelled: 1338x1 logical

        Values:

            True          29
            False       1309
```

Importing Date and Time Data

Use `DatetimeVariableImportOptions` properties to control the import of `datetime` data. First, get the `ImportOptions` object for the file. Next, examine and update the `VariableImportOptions` for the datetime variables. Finally, import the variables using `readtable`.

Preview of data in `outages.csv`. Notice the date and time data in the columns `OutageTime` and `RestorationTime`. Only the first 10 rows are shown here.

	A	B	C	D	E	F
1	Region	OutageTime	Loss	Customers	RestorationTime	Cause
2	SouthWest	2/1/2002 12:18	458.977222	1820159.482	2/7/2002 16:50	winter storm
3	SouthEast	1/23/2003 0:49	530.13995	212035.3001		winter storm
4	SouthEast	2/7/2003 21:15	289.403549	142938.6282	2/17/2003 8:14	winter storm
5	West	4/6/2004 5:44	434.805352	340371.0338	4/6/2004 6:10	equipment fault
6	MidWest	3/16/2002 6:18	186.436779	212754.055	3/18/2002 23:23	severe storm
7	West	6/18/2003 2:49	0	0	6/18/2003 10:54	attack
8	West	6/20/2004 14:39	231.294723		6/20/2004 19:16	equipment fault
9	West	6/6/2002 19:28	311.860732		6/7/2002 0:51	equipment fault
10	NorthEast	7/16/2003 16:23	239.932847	49434.48932	7/17/2003 1:12	fire

Get the import options object.

```
opts = detectImportOptions('outages.csv');
```

Get and examine the `VariableImportOptions` for datetime variables `OutageTime` and `RestorationTime`.

```
varOpts = getvaropts(opts,{'OutageTime','RestorationTime'})
```

```
varOpts 
  1x2 DatetimeVariableImportOptions array with properties:

    Name
    Type
    FillValue
    TreatAsMissing
    QuoteRule
    Prefixes
    Suffixes
    EmptyFieldRule
    DatetimeFormat
    DatetimeLocale
    TimeZone
    InputFormat
```

Set the `FillValue` property of the variables to replace missing values with current date and time.

```
opts = setvaropts(opts,{'OutageTime','RestorationTime'},...
                                          'FillValue','now');
```

Select, read, and preview the two variables. Notice the missing value in the second row of `RestorationTime` has been filled with current date and time.

```
opts.SelectedVariableNames = {'OutageTime','RestorationTime'};
T = readtable('outages.csv',opts);
T(1:10,:)
```

```
ans=10×2 table
         OutageTime           RestorationTime
    ____________________    ____________________

    01-Feb-2002 12:18:00    07-Feb-2002 16:50:00
    23-Jan-2003 00:49:00    27-Aug-2019 00:15:35
    07-Feb-2003 21:15:00    17-Feb-2003 08:14:00
    06-Apr-2004 05:44:00    06-Apr-2004 06:10:00
    16-Mar-2002 06:18:00    18-Mar-2002 23:23:00
    18-Jun-2003 02:49:00    18-Jun-2003 10:54:00
    20-Jun-2004 14:39:00    20-Jun-2004 19:16:00
    06-Jun-2002 19:28:00    07-Jun-2002 00:51:00
    16-Jul-2003 16:23:00    17-Jul-2003 01:12:00
    27-Sep-2004 11:09:00    27-Sep-2004 16:37:00
```

Set Options When Importing Categorical Data

Use the `setvaropts` function to update properties that control the import of `categorical` data. First, get the import options object for the file. Next, examine and update the options for the categorical variables. Finally, import the variables using the `readtable` function.

Preview the data in `outages.csv`. Notice the categorical data in the columns `Region` and `Cause`. This table shows only the first 10 rows.

	A	B	C	D	E	F
1	Region	OutageTime	Loss	Customers	RestorationTime	Cause
2	SouthWest	2/1/2002 12:18	458.977222	1820159.482	2/7/2002 16:50	winter storm
3	SouthEast	1/23/2003 0:49	530.13995	212035.3001		winter storm
4	SouthEast	2/7/2003 21:15	289.403549	142938.6282	2/17/2003 8:14	winter storm
5	West	4/6/2004 5:44	434.805352	340371.0338	4/6/2004 6:10	equipment fault
6	MidWest	3/16/2002 6:18	186.436779	212754.055	3/18/2002 23:23	severe storm
7	West	6/18/2003 2:49	0	0	6/18/2003 10:54	attack
8	West	6/20/2004 14:39	231.294723		6/20/2004 19:16	equipment fault
9	West	6/6/2002 19:28	311.860732		6/7/2002 0:51	equipment fault
10	NorthEast	7/16/2003 16:23	239.932847	49434.48932	7/17/2003 1:12	fire

Get the import options object.

```
opts = detectImportOptions('outages.csv');
```

Get and examine the options for variables `Region` and `Cause`.

```
getvaropts(opts,{'Region','Cause'})
```

```
ans 
  1x2 TextVariableImportOptions array with properties:

    Name
    Type
    FillValue
    TreatAsMissing
    QuoteRule
    Prefixes
    Suffixes
    EmptyFieldRule
    WhitespaceRule
```

Set the data type of the variables to `categorical`.

```
opts = setvartype(opts,{'Region','Cause'},'categorical');
```

Set the `FillValue` property of the variables to replace missing values with category name `'Miscellaneous'`. Set `TreatAsMissing` property to `'unknown'`.

```
opts = setvaropts(opts,{'Region','Cause'},...
                                    'FillValue','Miscellaneous',...
                                    'TreatAsMissing','unknown');
```

Select, read, and display a summary of the two variables.

```
opts.SelectedVariableNames = {'Region','Cause'};
T = readtable('outages.csv',opts);
summary(T)
```

```
Variables:

    Region: 1468x1 categorical

        Values:

            MidWest        142
            NorthEast      557
            SouthEast      389
            SouthWest       26
            West           354

    Cause: 1468x1 categorical

        Values:

            Miscellaneous          24
            attack                294
            earthquake              2
            energy emergency      188
            equipment fault       156
            fire                   25
            severe storm          338
            thunder storm         201
            wind                   95
            winter storm          145
```

Remove Prefix or Suffix Characters From Variables

Import tabular data containing variables that have unwanted prefix and suffix characters. First, create an import options object and preview the data. Then, select the variables of interest and set their variable types and properties to remove the unwanted characters. Finally, import the data of interest.

Create import options for the file and preview the table.

```
filename = 'pref_suff_trim.csv';
opts = detectImportOptions(filename);
preview(filename,opts)
```

```
ans=8×5 table
             Time                    DayOfWeek          Power        Total_Fees    Tempera
    ______________________    ______________    ____________    __________    _______

    'Timestamp:1/1/06 0:00'    '&amp; Sun %20'    '54.5448 MW'    '$1.23'       '-7.222
    'Timestamp:1/2/06 1:00'    '&amp; Thu %20'    '.3898 MW'      '$300.00'     '-7.305
    'Timestamp:1/3/06 2:00'    '&amp; Sun %20'    '51.6344 MW'    'Â£2.50'      '-7.852
    'Timestamp:1/4/06 3:00'    '&amp; Sun %20'    '51.5597 MW'    '$0.00'       '-8.177
    'Timestamp:1/5/06 4:00'    '&amp; Wed %20'    '51.7148 MW'    'Â¥4.00'      '-8.934
    'Timestamp:1/6/06 5:00'    '&amp; Sun %20'    '52.6898 MW'    '$0.00'       '-8.755
    'Timestamp:1/7/06 6:00'    '&amp; Mon %20'    '55.341 MW'     '$50.70'      '-8.041
    'Timestamp:1/8/06 7:00'    '&amp; Sat %20'    '57.9512 MW'    '$0.00'       '-8.202
```

Select variables of interest, specify their types, and examine their variable import options values.

```
opts.SelectedVariableNames = {'Time','Total_Fees','Temperature'};
opts = setvartype(opts,'Time','datetime');
opts = setvaropts(opts,'Time','InputFormat','MM/dd/uu HH:mm'); % Specify datetime forma
opts = setvartype(opts,{'Total_Fees','Temperature'},'double');
getvaropts(opts,{'Time','Total_Fees','Temperature'})
```

```
ans =
    1x3 VariableImportOptions array with properties:

   Variable Options:
                         (1) |          (2) |           (3)
             Name:     'Time' | 'Total_Fees' | 'Temperature'
             Type: 'datetime' |     'double' |      'double'
        FillValue:      [NaT] |        [NaN] |         [NaN]
  TreatAsMissing:         {} |           {} |            {}
       QuoteRule:   'remove' |     'remove' |      'remove'
        Prefixes:         {} |           {} |            {}
        Suffixes:         {} |           {} |            {}

    To access sub-properties of each variable, use getvaropts
```

Set the `Prefixes`, `Suffixes`, and `TrimNonNumeric` properties of variable import options to remove `'Timestamp:'` from the variable `Time`, remove suffix `'C'` from variable `Temperature`, and remove all nonnumeric characters from variable `Total_Fees`. Preview the table with the new import options.

```
opts = setvaropts(opts,'Time','Prefixes','Timestamp:');
opts = setvaropts(opts,'Temperature','Suffixes','C');
opts = setvaropts(opts,'Total_Fees','TrimNonNumeric',true);
preview(filename,opts)
```

```
ans=8×3 table
         Time         Total_Fees    Temperature
    _____________    __________    ___________

    01/01/06 00:00       1.23         -7.2222
    01/02/06 01:00        300         -7.3056
    01/03/06 02:00        2.5         -7.8528
    01/04/06 03:00          0         -8.1778
    01/05/06 04:00          4         -8.9343
    01/06/06 05:00          0         -8.7556
    01/07/06 06:00       50.7         -8.0417
    01/08/06 07:00          0         -8.2028
```

Import the data using `readtable`.

```
T = readtable(filename,opts);
```

Manage Import of Empty Fields

Create an import options object for a file containing empty fields. Use the `EmptyFieldRule` parameter to manage the import of empty fields in your data. First, preview the data, and then set the `EmptyFieldRule` parameter for a specific variable. Finally, set `EmptyFieldRule` for all the variables and import the data.

Create an import options object for a file containing empty fields. Get the first eight rows of the table using the `preview` function. The default value for `EmptyFieldRule` is `'missing'`. Therefore, the importing function treats empty fields as missing and replaces them with the `FillValue` value for that variable. Using `VariableOptions` for the third variable, preview the data. Here, the `preview` function imports the empty fields in the third variable as `NaN`s.

```
filename = 'DataWithEmptyFields.csv';
opts = detectImportOptions(filename);
opts.VariableOptions(3) % Display the Variable Options for the 3rd Variable
```

```
ans = 
  NumericVariableImportOptions with properties:

   Variable Properties:
                   Name: 'Double'
                   Type: 'double'
              FillValue: NaN
          TreatAsMissing: {}
               QuoteRule: 'remove'
                Prefixes: {}
                Suffixes: {}
          EmptyFieldRule: 'missing'

   Numeric Options:
     ExponentCharacter: 'eEdD'
      DecimalSeparator: '.'
    ThousandsSeparator: ''
        TrimNonNumeric: 0
```

```
preview(filename,opts)
```

```
ans=8×7 table
       Text        Categorical    Double          Datetime           Logical      Durati
    __________    ___________    ______    ____________________    __________    ______

    {'abc'   }    {'a'     }        1      14-Jan-0018 00:00:00    {'TRUE'  }    00:00
    {0x0 char}    {'b'     }        2      21-Jan-0018 00:00:00    {'FALSE' }    09:00
    {'ghi'   }    {0x0 char}        3      31-Jan-0018 00:00:00    {'TRUE'  }    02:00
    {'jkl'   }    {'a'     }      NaN      23-Feb-2018 00:00:00    {'FALSE' }    03:00
    {'mno'   }    {'a'     }        4                       NaT    {'FALSE' }    04:00
    {'pqr'   }    {'b'     }        5      23-Jan-0018 00:00:00    {0x0 char}    05:00
    {'stu'   }    {'b'     }        5      23-Mar-0018 00:00:00    {'FALSE' }         N
    {0x0 char}    {'a'     }        6      24-Mar-2018 00:00:00    {'TRUE'  }    07:00
```

Set `EmptyFieldRule` for the second variable in the table. First, select the variable, and then set `EmptyFieldRule` to `'auto'`. Here, the `readtable` function imports empty fields of the `categorical` variable as `<undefined>`.

```
opts.SelectedVariableNames = 'Categorical';
opts = setvartype(opts,'Categorical','categorical');
opts = setvaropts(opts,'Categorical','EmptyFieldRule','auto');
T = readtable(filename,opts)
```

```
T=11×1 table
    Categorical
    ___________

    a
    b
    <undefined>
    a
    a
    b
    b
    a
    a
    <undefined>
    <undefined>
```

Next, set the `EmptyFieldRule` parameter for all the variables in the table. First, update the data types of the variables appropriately. For this example, set the data type of the fifth and seventh variables to `logical` and `string`, respectively. Then, set `EmptyFieldRule` for all the variables to `'auto'`. The importing function imports the empty fields based on data type of the variable. Here, the `readtable` function imports empty fields of the `logical` variable as `0` and empty fields of the `categorical` variable as `<undefined>`.

```
VariableNames = opts.VariableNames;
opts.SelectedVariableNames =  VariableNames; % select all variables
opts = setvartype(opts,{'Logical','String'},{'logical','string'});
opts = setvaropts(opts,VariableNames,'EmptyFieldRule','auto');
T = readtable(filename,opts)
```

```
T=11×7 table
       Text       Categorical    Double          Datetime          Logical    Duration
    __________    ___________    ______    ____________________    _______    ________

    {'abc'   }    a                 1      14-Jan-0018 00:00:00     true      00:00:01
    {0x0 char}    b                 2      21-Jan-0018 00:00:00     false     09:00:01
    {'ghi'   }    <undefined>       3      31-Jan-0018 00:00:00     true      02:00:01
    {'jkl'   }    a               NaN      23-Feb-2018 00:00:00     false     03:00:01
```

```
    {'mno'   }    a                 4                       NaT     false     04:00:01
    {'pqr'   }    b                 5      23-Jan-0018 00:00:00     false     05:00:01
    {'stu'   }    b                 5      23-Mar-0018 00:00:00     false          NaN
    {0x0 char}    a                 6      24-Mar-2018 00:00:00     true      07:00:01
    {0x0 char}    a                 7      25-Mar-2018 00:00:00     true      08:00:01
    {'xyz'   }    <undefined>     NaN                       NaT     true      06:00:01
    {0x0 char}    <undefined>     NaN                       NaT     false          NaN
```

In addition to `'missing'` and `'auto'`, you can also set the `EmptyFieldRule` parameter to `'error'`. When you set it to `'error'`, the `readtable` function imports empty fields by following the procedure specified in the `ImportErrorRule` parameter.

Input Arguments

`opts` — File import options
`SpreadsheetImportOptions` | `DelimitedTextImportOptions` | `FixedWidthImportOptions`

File import options, specified as a `SpreadsheetImportOptions`, `DelimitedTextImportOptions`, or a `FixedWidthImportOptions` object created by the `detectImportOptions` function. The `opts` object contains properties that control the data import process, such as variable properties, data location properties, replacement rules, and others.

`selection` — Selected variables
character vector | string scalar | cell array of character vector | string array | array of indices | logical array

Selected variables, specified as a character vector, string scalar, cell array of character vectors, string array, array of numeric indices, or a logical array.

Variable names (or indices) must be a subset of the names contained in the `VariableNames` property of the `opts` object.

Example: `'Height'`

Example: `{'Height','LastName'}`

Example: `[5 9]`

Data Types: `char` | `string` | `cell` | `uint64` | `logical`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `opts = setvaropts(opts,'Weight','FillValue',0)` sets the `FillValue` for the variable `Weight` to `0`.

Common Options for All Variable Types

`FillValue` — Replacement value for missing data

character vector | string scalar | scalar numeric value | `true` | `false`

Replacement value for missing data, depending on the type of the variable, specified as a character vector, string scalar, scalar numeric, or a logical value `true` or `false`.

Type of Variable	Description
Text	Replacement text specified as a character vector or string scalar. **Example:** `'not applicable'`
Numeric	Scalar numeric to replace missing instances The importing function converts the input for `FillValue` to the data type specified by the `Type` property of the variable. For example, if `Type` property value is `uint8`, then the importing function also converts the value of the `FillValue` property to `uint8`. **Example:** `0`
Logical	`true` or `false`. **Example:** `false`

Type of Variable	Description
Datetime	Character vector or string scalar, or a scalar value representing date and time data. For more information on valid `datetime` inputs, see the `datetime` function page. **Example:** `'now'` sets the missing datetime instances to the current date and time. **Example:** `[1998 12 1]` sets the missing datetime instances to the date December 1st, 1998.
Duration	Character vector or string scalar, or a scalar value representing duration data. For more information on valid `duration` inputs, see the `duration` function page. **Example:** `'12:30:16'` sets the missing duration instances to the duration of 12 hours, 30 minutes, and 16 seconds.
Categorical	Character vector or string scalar containing the name to use for the replacement category. **Example:** `'Miscellaneous'` assigns the category name Miscellaneous to missing instances in the categorical data.

To direct the import of data that is missing, unconvertible, or that causes errors, use these four properties together: `FillValue`, `TreatAsMissing`, `MissingRule`, and `ErrorRule`. The importing function uses the value specified in the `FillValue` property when:

- Data is unconvertible or matches a value in `TreatAsMissing`.
- `MissingRule` or the `ErrorRule` is set to `fill`.

`TreatAsMissing` — Text to interpret as missing data
character vector | string scalar | cell array of character vectors | string array

Text to interpret as missing data, specified as a character vector, string scalar, cell array of character vectors, or string array.

When the importing function finds missing instances, it uses the specification in the `MissingRule` property to determine the appropriate action.

Example: `'TreatAsMissing',{'NA','TBD'}` instructs the importing function to treat any occurrence of `NA` or `TBD` as a missing fields.

Data Types: `char` | `string` | `cell`

`QuoteRule` — Procedure to manage double quotation marks
`'remove'` | `'keep'` | `'error'`

Procedure to manage double quotation marks in the data, specified as one of the values in this table.

Quote Rule	Process
`'remove'`	If double quotes (`"`) surround characters, then the importing function removes both the opening double quote and the next occurring double quote, which would be interpreted as the closing double quote. **Example:** `"500"` is imported as `500`. If two sets of double quotes (`""`) surround characters, then the importing function removes the first two occurrences. **Example:** `""abc""` is imported as `abc""`. If a pair of opening and closing quotes surrounding characters is followed by a single lone unpaired double quotes, then the importing function ignores the lone unpaired double quote. **Example:** `"abc""` is imported as `abc"`.
`'keep'`	Retain all quotation marks.
`'error'`	Report an error when converting data which begins with a double quotation mark (`"`). Use this setting if the field should never be quoted.

Example: If the `'QuoteRule'` is set to `'remove'`, then the importing function imports `"abc"def"` as `abcdef"`.

Data Types: `char` | `string`

`EmptyFieldRule` — Procedure to manage empty fields
`'missing'` (default) | `'error'` | `'auto'`

Procedure to manage empty fields in the data, specified as one of these values:

- `'missing'` — Use the procedure specified in the `MissingRule` property
- `'error'` — Use the procedure specified in the `ImportErrorRule` property
- `'auto'` — Use the empty value based on the data type of the variable

Data Type	Empty Value
`text`	zero length `char` or `string`
`numeric`	`NaN` for floating-point numbers `0` for integers
`duration`	`NaN`
`datetime`	`NaT`
`logical`	`false`
`categorical`	`<undefined>`

Example: `opts = setvaropts(opts,'RestorationTime','EmptyFieldRule','auto');`

Data Types: `char` | `string`

`Prefixes` — Text to remove from prefix position
character vector | cell array of character vectors | string scalar | string array

Text to remove from the prefix position of a variable value, specified as a character vector, cell array of character vectors, string scalar, or string array.

Example: `opts = setvaropts(opts,'Var1','Prefixes','$')` sets the `Prefixes` option for the variable `Var1`. If `Var1` contains a value of `'$500'`, then `readtable` reads it as `'500'`.

Data Types: `char` | `string` | `cell`

`Suffixes` — Text to remove from suffix position

character vector | cell array of character vectors | string scalar | string array

Text to remove from the suffix position of a variable value, specified as a character vector, cell array of character vectors, string scalar, or string array.

Example: `opts = setvaropts(opts,'Var1','Suffixes','/-')` sets the `Suffixes` option for the variable `Var1`. If `Var1` contains a value of `'$500/-'` , then `readtable` reads it as `'$500'`.

Data Types: `char` | `string` | `cell`

Text Variable Options

`WhiteSpaceRule` — Procedure to manage leading and trailing white spaces

`'trim'` (default) | `'trimleading'` | `'trimtrailing'` | `'preserve'`

Procedure to manage leading and trailing white spaces when importing text data, specified as one of the values in the table.

White Space Rule	Process
`'trim'`	Remove any leading or trailing white spaces from the text. Interior white space is unaffected. **Example:** `' World Time '` is imported as `'World Time'`
`'trimleading'`	Remove only the leading white spaces. **Example:** `' World Time '` is imported as `'World Time '`
`'trimtrailing'`	Remove only the trailing white spaces. **Example:** `' World Time '` is imported as `'World Time'`
`'preserve'`	Preserve white spaces. **Example:** `' World Time '` is imported as `' World Time '`

Data Types: `char` | `string`

Numeric Variable Options

`ExponentCharacter` — Characters indicating exponent
character vector | string scalar

Characters indicating the exponent, specified as a character vector or string scalar. The importing function uses the `ExponentCharacter` property to recognize the characters indicating the exponent for a number expressed in the scientific notation.

Example: If `varOpts.ExponentCharacter = 'a'`, then the importing function imports the text `"1.2a3"` as the number `1200`.

Data Types: `char` | `string`

`DecimalSeparator` — Characters indicating decimal separator
character vector | string scalar

Characters indicating the decimal separator in numeric variables, specified as a character vector or string scalar. The importing function uses the characters specified in the `DecimalSeparator` name-value pair to distinguish the integer part of a number from the decimal part.

When converting to integer data types, numbers with a decimal part are rounded to the nearest integer.

Example: If name-value pair is specified as `'DecimalSeparator',','`, then the importing function imports the text `"3,14159"` as the number `3.14159`.

Data Types: `char` | `string`

`ThousandsSeparator` — Characters that indicate thousands grouping
character vector | string scalar

Characters that indicate the thousands grouping in numeric variables, specified as a character vector or string scalar. The thousands grouping characters act as visual separators, grouping the number at every three place values. The importing function uses the characters specified in the `ThousandsSeparator` name-value pair to interpret the numbers being imported.

Example: If name-value pair is specified as `'ThousandsSeparator',','`, then the importing function imports the text `"1,234,000"` as `1234000`.

Data Types: `char` | `string`

`TrimNonNumeric` — Remove nonnumeric characters
`false` (default) | `true`

Remove nonnumeric characters from a numeric variable, specified as a logical `true` or `false`.

Example: If name-value pair is specified as `'TrimNonNumeric',true`, then the importing function reads `'$500/-'` as `500`.

Data Types: `logical`

Logical Variable Options

`TrueSymbols` — Text to treat as logical value `true`
character vector | string scalar | cell array of character vectors | string array

Text to treat as the logical value `true`, specified as a character vector, string scalar, cell array of character vectors, or a string array.

Example: If `varOpts.TrueSymbols = {'t','TRUE'}`, then the importing function imports any fields containing `t` or `TRUE` as the logical value `true`.

Data Types: `char` | `string` | `cell`

`FalseSymbols` — Text to treat as logical value `false`
character vector | string scalar | cell array of character vectors | string array

Text to treat as the logical value `false`, specified as a character vector, string scalar, cell array of character vectors, or a string array.

Example: If `varOpts.FalseSymbols = {'f','FALSE'}`, then the importing function imports any fields containing `f` or `FALSE` as the logical value `false`.

Data Types: `char` | `string` | `cell`

`CaseSensitive` — Indicator to match case
`true` | `false`

Indicator to match case, specified as a logical value `true` or `false`.

To interpret the input data as missing, `true`, or `false`, the importing function matches the data to values specified in `TreatAsMissing`, `TrueSymbols`, and `FalseSymbols`.

Datetime Variable Options

DatetimeFormat — Display format

'default' | 'defaultdate' | 'preserveinput' | custom format | character vector | string scalar

Display format, specified as a character vector or string scalar. The DatetimeFormat property controls the display format of dates and times in the output. Specify DatetimeFormat as one of these values.

Value of DatetimeFormat	Description
'default'	Use the default display format.
'defaultdate'	Use the default display format for datetime values created without time components.
'preserveinput'	Use the format specified by the input format, InputFormat.
Custom formats	Use the letters A-Z and a-z to construct a custom value for DatetimeFormat. These letters correspond to the Unicode Locale Data Markup Language (LDML) standard for dates. You can include non-ASCII or nonletter characters such as a hyphen, space, or colon to separate the fields. To include the letters A-Z and a-z as literal characters in the format, enclose them with single quotes.

The factory default format depends on your system locale. To change the default display format, see “Default datetime Format”.

Data Types: char | string

InputFormat — Format of input text

character vector | string scalar

Format of the input text representing dates and times, specified as a character vector or string scalar that contains letter identifiers.

This table shows several common input formats and examples of the formatted input for the date, Saturday, April 19, 2014 at 9:41:06 PM in New York City.

Value of InputFormat	Example
'yyyy-MM-dd'	2014-04-19
'dd/MM/yyyy'	19/04/2014
'dd.MM.yyyy'	19.04.2014
'yyyy 年 MM 月 dd 日'	2014 年 04 月 19 日
'MMMM d, yyyy'	April 19, 2014
'eeee, MMMM d, yyyy h:mm a'	Saturday, April 19, 2014 9:41 PM
'MMMM d, yyyy HH:mm:ss Z'	April 19, 2014 21:41:06 -0400
'yyyy-MM-dd''T''HH:mmXXX'	2014-04-19T21:41-04:00

For a complete list of valid letter identifiers, see the `Format` property for datetime arrays.

Example: `'InputFormat','eeee, MMMM d, yyyy HH:mm:ss'`

Data Types: `char` | `string`

`DatetimeLocale` — Locale to interpret dates
character vector | string scalar

Locale to interpret dates, specified as a character vector or string scalar. The `DatetimeLocale` value determines how the importing function interprets text that represents dates and times.

Set the `DatetimeLocale` to one of these values:

- `'system'` — Specify your system locale.
- Character vector or string scalar — Use the form *xx_YY*, where *xx* is a lowercase ISO 639-1 two-letter code that specifies a language, and *YY* is an uppercase ISO 3166-1 alpha-2 code that specifies a country.

This table lists some common values for the locale.

Locale	Language	Country
'de_DE'	German	Germany
'en_GB'	English	United Kingdom
'en_US'	English	United States

Locale	Language	Country
'es_ES'	Spanish	Spain
'fr_FR'	French	France
'it_IT'	Italian	Italy
'ja_JP'	Japanese	Japan
'ko_KR'	Korean	Korea
'nl_NL'	Dutch	Netherlands
'zh_CN'	Chinese (simplified)	China

Example: `varOpts.DatetimeLocale = 'de_DE'` sets the date time locale to German.

Note The `Locale` value determines how input values are interpreted. The display format and language is specified by the **Locale** option in the **Datetime format** section of the Preferences panel. To change the default datetime locale, see "Set Command Window Preferences".

Data Types: `char` | `string`

`TimeZone` — Time zone
`''` (default) | character vector | string scalar

Time zone, specified as a character vector or string scalar. The value of `TimeZone` specifies the time zone that the importing function uses to interpret the input data. `TimeZone` also specifies the time zone of the output array. If the input data are character vectors that include a time zone, then the importing function converts all values to the specified time zone.

`TimeZone` use any of these values:

- `''`, to create "unzoned" datetime values that do not belong to a specific time zone.
- A time zone region from the IANA Time Zone Database; for example, `'America/Los_Angeles'`. The name of a time zone region accounts for the current and historical rules for standard and daylight offsets from UTC that are observed in a geographic region.
- An ISO 8601 character vector of the form `+HH:mm` or `-HH:mm`. For example, `'+01:00'`, specifies a time zone that is a fixed offset from UTC.

- `'UTC'`, to create datetime values in Universal Coordinated Time.
- `'UTCLeapSeconds'`, to create datetime values in Universal Coordinated Time that account for leap seconds.

This table lists some common names of time zone regions from the IANA Time Zone Database.

Value of `TimeZone`	UTC Offset	UTC DST Offset
`'Africa/Johannesburg'`	+02:00	+02:00
`'America/Chicago'`	−06:00	−05:00
`'America/Denver'`	−07:00	−06:00
`'America/Los_Angeles'`	−08:00	−07:00
`'America/New_York'`	−05:00	−04:00
`'America/Sao_Paulo'`	−03:00	−02:00
`'Asia/Hong_Kong'`	+08:00	+08:00
`'Asia/Kolkata'`	+05:30	+05:30
`'Asia/Tokyo'`	+09:00	+09:00
`'Australia/Sydney'`	+10:00	+11:00
`'Europe/London'`	+00:00	+01:00
`'Europe/Zurich'`	+01:00	+02:00

Datetime arrays with no specified `TimeZone` cannot be compared or combined with arrays that have their `TimeZone` property set to a specific time zone.

Data Types: `char` | `string`

Categorical Variable Options

`Categories` — Expected categories

cell array of character vectors | string array

Expected categories, specified as a cell array of character vectors or string array containing a list of category names.

Names in the input fields must match one of the names specified in the `Categories` property to avoid a conversion error.

Example: `varOpts.Categories = {'BareLand','Forest','Water','Roads','Buildings'};`

`Protected` — Category protection indicator
`true` | `false`

Category protection indicator, specified as a logical `true` or `false`. The categories of ordinal categorical arrays are always protected. If the `Ordinal` property is set to `true`, then the default value for `Protected` is `true`. Otherwise, the value for `Protected` is `false`. For more information on categorical arrays, see the `categorical` function reference page.

`Ordinal` — Mathematical ordering indicator
`false` (default) | `true` | `0` | `1`

Mathematical ordering indicator, specified as either `false`, `true`, `0`, or `1`. For more information on categorical arrays, see the `categorical` function reference page.

Duration Variable Options

`InputFormat` — Format of input text
character vector | string scalar

Format of the input text representing duration data, specified as a character vector or string array.

Specify `InputFormat` as any of the following formats, where `dd`, `hh`, `mm`, and `ss` represent days, hours, minutes, and seconds, respectively:

- `'dd:hh:mm:ss'`
- `'hh:mm:ss'`
- `'mm:ss'`
- `'hh:mm'`
- Any of the first three formats, with up to nine `S` characters to indicate fractional second digits, such as `'hh:mm:ss.SSSS'`

Example: `varOpts.InputFormat = 'dd:hh:mm:ss'`

Data Types: `char` | `string`

`DurationFormat` — Display format for duration data
character vector | string scalar

Display format for duration data, specified as a character vector or string scalar.

For numbers with time units, specify one of these values:

- `'y'` — Fixed-length years, where 1 year equals 365.2425 days
- `'d'` — Fixed-length days, where 1 day equals 24 hours
- `'h'` — Hours
- `'m'` — Minutes
- `'s'` — Seconds

For the digital timer, specify one of these formats:

- `'dd:hh:mm:ss'`
- `'hh:mm:ss'`
- `'mm:ss'`
- `'hh:mm'`
- Any of the first three formats, with up to nine `S` characters to indicate fractional second digits, such as `'hh:mm:ss.SSSS'`

Example: `DurationFormat` can be either a single number with time units (such as `0.5 yr`) or a digital timer (such as `10:30:15` for 10 hours, 30 minutes, and 15 seconds).

Data Types: `char` | `string`

`DecimalSeparator` — Characters separating fractional seconds
`.` (default) | character vector | string scalar

Characters separating fractional seconds, specified as a character vector or string scalar. The importing function uses the `DecimalSeparator` property to distinguish the integer part of the duration value from the fractional seconds part.

Data Types: `char` | `string`

`FieldSeparator` — Characters indicating field separation
`:` (default) | character vector | string scalar

Characters indicating field separation in the duration data, specified as a character vector or string scalar.

Data Types: `char` | `string`

See Also

`detectImportOptions` | `getvaropts` | `preview` | `setvartype`

Introduced in R2016b

setvartype

Package: `matlab.io.text`

Set variable data types

Syntax

```
opts = setvartype(opts,type)
opts = setvartype(opts,selection,type)
```

Description

`opts = setvartype(opts,type)` updates all the variables in the `opts` object based on the specified type. `type` must be a character vector or a cell array of character vector of valid data type names.

`opts = setvartype(opts,selection,type)` updates data type for only the variables specified in the `selection` argument.

Examples

Set Data Types for Multiple Variables

Use `detectImportOptions` to create import options, set multiple variable data types, and then read the data using `readtable`.

Create an options object.

```
opts = detectImportOptions('patients.xls');
```

Examine the current (detected) data types of the variables.

```
disp([opts.VariableNames' opts.VariableTypes'])
```

```
    'LastName'                    'char'
    'Gender'                      'char'
    'Age'                         'double'
    'Location'                    'char'
    'Height'                      'double'
    'Weight'                      'double'
    'Smoker'                      'logical'
    'Systolic'                    'double'
    'Diastolic'                   'double'
    'SelfAssessedHealthStatus'    'char'
```

Change the data type of multiple variables depending on your import needs.

```
opts = setvartype(opts,{'LastName','Gender','Location',...
                    'Smoker','SelfAssessedHealthStatus'},'string');
opts = setvartype(opts,{'Age','Height','Weight',...
                                 'Systolic','Diastolic'},'single');
```

Examine the updated data types of the variables.

```
disp([opts.VariableNames' opts.VariableTypes'])
```

```
    'LastName'                    'string'
    'Gender'                      'string'
    'Age'                         'single'
    'Location'                    'string'
    'Height'                      'single'
    'Weight'                      'single'
    'Smoker'                      'string'
    'Systolic'                    'single'
    'Diastolic'                   'single'
    'SelfAssessedHealthStatus'    'string'
```

Import the variables with their updated types using `readtable`.

```
T = readtable('patients.xls',opts);
```

Input Arguments

opts — File import options
`SpreadsheetImportOptions` | `DelimitedTextImportOptions` | `FixedWidthImportOptions`

File import options, specified as a `SpreadsheetImportOptions`, `DelimitedTextImportOptions`, or a `FixedWidthImportOptions` object created by the `detectImportOptions` function. The `opts` object contains properties that control the data import process, such as variable properties, data location properties, replacement rules, and others.

`selection` — Selected variables

character vector | string scalar | cell array of character vector | string array | array of indices | logical array

Selected variables, specified as a character vector, string scalar, cell array of character vectors, string array, array of numeric indices, or a logical array.

Variable names (or indices) must be a subset of the names contained in the `VariableNames` property of the `opts` object.

Example: `'Height'`

Example: `{'Height','LastName'}`

Example: `[5 9]`

Data Types: `char` | `string` | `cell` | `uint64` | `logical`

`type` — Variable data type

character vector | string scalar | cell array of character vectors | string array

Variable data type, specified as a character vector, string scalar, cell array of character vectors, or string array containing valid data type names. The `type` argument designates the data types to use when importing the variable. Import the variables using one of the data types listed here.

Data	MATLAB Data Type
Text	`'char'` or `'string'`

Data	MATLAB Data Type
Numeric	`'single'`, `'double'`, `'int8'`, `'int16'`, `'int32'`, `'int64'`, `'uint8'`, `'uint16'`, `'uint32'`, or `'uint64'` Undefined floating-point numbers `NaN`, `-Inf`, `+Inf` are only valid for `single` and `double` data types. Therefore, when you change the type of floating-point data to an integer, the importing function converts the undefined floating-point numbers. For example, when converting to the `'uint8'` data type: • `NaN` is converted to `0`. • `-Inf` is converted to `intmin('int8')`. • `+Inf` is converted to `intmax('int8')`. The same conversion process applies to all the integer data types: `int8`, `int16`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, or `uint64`.
Logical	`'logical'`
Date and time	`'datetime'`
Duration	`'duration'`
Categorical	`'categorical'`

Example: `opts = setvartype(opts,'Height','double')` changes the data type of the variable `Height` to `double`.

Example: `opts = setvartype(opts,{'Weight','LastName'},{'single','string'})` changes the data type of the variable `Weight` to `single` and variable `LastName` to `string`.

Data Types: `char` | `string` | `cell`

See Also

`detectImportOptions` | `getvaropts` | `preview` | `setvaropts`

Introduced in R2016b

spreadsheetImportOptions

Import options object for Spreadsheets

Description

A `SpreadsheetImportOptions` object enables you to specify how MATLAB imports tabular data from spreadsheet files. The object contains properties that control the data import process, including the handling of errors and missing data.

Creation

You can create a `SpreadsheetImportOptions` object by using either the `detectImportOptions` function or the `spreadsheetImportOptions` function (described here):

- Use `detectImportOptions` to detect and populate the import properties based on the contents of the spreadsheet specified in `filename`.

  ```
  opts = detectImportOptions(filename);
  ```

- Use `spreadsheetImportOptions` to define the import properties based on your import requirements.

Syntax

```
opts = spreadsheetImportOptions
opts = spreadsheetImportOptions('NumVariables',numVars)
opts = spreadsheetImportOptions( ___ ,Name,Value)
```

Description

`opts = spreadsheetImportOptions` creates a `SpreadsheetImportOptions` object with one variable.

`opts = spreadsheetImportOptions('NumVariables',numVars)` creates the object with the number of variables specified in `numVars`.

`opts = spreadsheetImportOptions( ___ ,Name,Value)` specifies additional properties for `SpreadsheetImportOptions` object using one or more name-value pair arguments.

Input Arguments

`numVars` — Number of variables
positive scalar integer

Number of variables, specified as a positive scalar integer.

Properties

Variable Properties

`VariableNames` — Variable names
cell array of character vectors | string array

Variable names, specified as a cell array of character vectors or string array. The `VariableNames` property contains the names to use when importing variables.

If the data contains `N` variables, but no variable names are detected, then the `VariableNames` property contains `{'Var1','Var2',...,'VarN'}`.

To support invalid MATLAB identifiers as variable names, such as varible names containing spaces and non-ASCII characters, set the `PreserveVariableNames` parameter to `true`.

Example: `opts.VariableNames` returns the current (detected) variable names.

Example: `opts.VariableNames(3) = {'Height'}` changes the name of the third variable to `Height`.

Data Types: `char` | `string` | `cell`

`PreserveVariableNames` — Flag to preserve variable names
`false` (default) | `true`

Flag to preserve variable names, specified as the comma-separated pair consisting of `PreserveVariableNames` and either `true`, or `false`.

- `true` — Preserve variable names that are not valid MATLAB identifiers such as variable names that include spaces and non-ASCII characters.
- `false` — Convert invalid variable names (as determined by the `isvarname` function) to valid MATLAB identifiers.

Starting in R2019b, variable names and row names can include any characters, including spaces and non-ASCII characters. Also, they can start with any characters, not just letters. Variable and row names do not have to be valid MATLAB identifiers (as determined by the `isvarname` function). To preserve these variable names and row names, set `PreserveVariableNames` to `true`.

`VariableTypes` — Variable data types
cell array of character vectors | string array

Variable data types, specified as a cell array of character vectors or string array. The `VariableTypes` property designates the data types to use when importing variables. When assigning new values, specify `VariableTypes` as a cell array of valid data type names.

To update the `VariableTypes` property, use the `setvartype` function.

Example: `opts.VariableTypes` returns the current (detected) variable data types.

Example: `opts = setvartype(opts,'Height',{'double'})` changes the data type of the variable `Height` to `double`.

Data Types: `cell` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `categorical` | `datetime`

`SelectedVariableNames` — Subset of variables to import
character vector | string scalar | cell array of character vectors | string array | array of numeric indices

Subset of variables to import, specified as a character vector, string scalar, cell array of character vectors, string array or an array of numeric indices.

`SelectedVariableNames` must be a subset of names contained in the `VariableNames` property. By default, `SelectedVariableNames` contains all the variable names from the `VariableNames` property, which means that all variables are imported.

Use the `SelectedVariableNames` property to import only the variables of interest. Specify a subset of variables using the `SelectedVariableNames` property and use `readtable` to import only that subset.

To support invalid MATLAB identifiers as variable names, such as varible names containing spaces and non-ASCII characters, set the `PreserveVariableNames` parameter to `true`.

Example: `opts.SelectedVariableNames = {'Height','LastName'}` selects only two variables, `Height` and `LastName`, for the import operation.

Example: `opts.SelectedVariableNames = [1 5]` selects only two variables, the first variable and the fifth variable, for the import operation.

Example: `T = readtable(filename,opts)` returns a table containing only the variables specified in the `SelectedVariableNames` property of the `opts` object.

Data Types: `uint16` | `uint32` | `uint64` | `char` | `string` | `cell`

`VariableOptions` — Type specific variable import options
array of variable import options objects

Type specific variable import options, returned as an array of variable import options objects. The array contains an object corresponding to each variable specified in the `VariableNames` property. Each object in the array contains properties that support the importing of data with a specific data type.

Variable options support these data types: numeric, text, `logical`, `datetime`, or `categorical`.

To query the current (or detected) options for a variable, use the `getvaropts` function.

To set and customize options for a variable, use the `setvaropts` function.

Example: `opts.VariableOptions` returns a collection of `VariableImportOptions` objects, one corresponding to each variable in the data.

Example: `getvaropts(opts,'Height')` returns the `VariableImportOptions` object for the `Height` variable.

Example: `opts = setvaropts(opts,'Height','FillValue',0)` sets the `FillValue` property for the variable `Height` to `0`.

Location Properties

Sheet — Sheet to read from

`''` empty character array (default) | character vector | string scalar | positive scalar integer

Sheet to read from, specified as an empty character array, a character vector or string scalar containing the sheet name, or a positive scalar integer denoting the sheet index. Based on the value specified for the `Sheet` property, the import function behaves as described in the table.

Specification	Behavior
`''` (default)	Import data from the first sheet.
Name	Import data from the matching sheet name, regardless of order of sheets in the spreadsheet file.
Integer	Import data from sheet in the position denoted by the integer, regardless of the sheet names in the spreadsheet file.

Data Types: `char` | `string` | `single` | `double`

DataRange — Location of data

character vector | string scalar | positive scalar integer | array of positive scalar integers | cell array of character vector | string array

Location of data to be imported, specified as a character vector, string scalar, cell array of character vectors, string array, positive scalar integer or an `N`-by-`2` array of positive scalar integers. Specify `DataRange` using one of these forms.

Specified by	Behavior
`'Cell'` or *n* Starting Cell or Starting Row	Specify the starting cell for the data, using Excel `A1` notation. For example, `A5` is the identifier for the cell at the intersection of column `A` and row `5`. Using the starting cell, the importing function automatically detects the extent of the data, by beginning the import at the start cell and ending at the last empty row or footer range. Alternatively, specify the first row containing the data using the positive scalar row index. Using the specified row index, the importing function automatically detects the extent of the data by reading from the specified first row to the end of the data or the footer range. **Example:** `'A5'` or `5`
`'Corner1:Corner2'` Rectangular Range	Specify the exact range to read using the rectangular range form, where *`Corner1`* and *`Corner2`* are two opposing corners that define the region to read. The importing function only reads the data contained in the specified range. Any empty fields within the specified range are imported as missing cells. The number of columns must match the number specified in the `NumVariables` property. **Example:** `'A5:K50'`

Specified by	Behavior
'*Row1:Row2*' or '*Column1:Column2*' Row Range or Column Range	Specify the range by identifying the beginning and ending rows using Excel row numbers. Using the specified row range, the importing function automatically detects the column extent by reading from the first nonempty column to the end of the data, and creates one variable per column. **Example:** `'5:500'` Alternatively, specify the range by identifying the beginning and ending columns using Excel column letters or numbers. Using the specified column range, the import function automatically detects the row extent by reading from the first nonempty row to the end of the data or the footer range. The number of columns in the specified range must match the number specified in the `NumVariables` property. **Example:** `'A:K'`
[n1 n2; n3 n4;...] Multiple Row Ranges	Specify *multiple row ranges* to read with an `N`-by-`2` array containing `N` different row ranges. A valid array of multiple row ranges must: • Specify line ranges in an increasing order, that is the first row range specified in the array appears in the file before the other row ranges. • Contain only non-overlapping row ranges. Use of `Inf` is only supported to indicate the last range in the numeric array specifying multiple row ranges. For example, `[1 3; 5 6; 8 Inf]`. **Example:** `[1 3; 5 6; 8 Inf]`

Specified by	Behavior
`''` Unspecified or Empty	Do not fetch any data. **Example:** `''`

Data Types: `char` | `string` | `cell` | `single` | `double`

`RowNamesRange` — Location of row names
character vector | string scalar | positive scalar integer | `''` empty character array

Location of row names, specified as a character vector, string scalar, positive scalar integer, or an empty character array. Specify `RowNamesRange` as one of the values in this table.

Specified by	Behavior
`'Cell'`	Specify the starting cell for the data, using Excel `A1` notation. For example, `A5` is the identifier for the cell at the intersection of column `A` and row `5`. The importing function identifies a name for each variable in the data. **Example:** `'A5'`
`'Corner1:Corner2'` Rectangular Range	Specify the exact range to read using the rectangular range form, where *`Corner1`* and *`Corner2`* are two opposing corners that define the region to read. The number of rows contained in `RowNamesRange` must match the number of data rows, and the range indicated by `RowNamesRange` must span only one column. **Example:** `'A5:A50'`
`'Row1:Row2'` Row Range	Specify range by identifying the beginning and ending rows using Excel row numbers. Row names must be in a single column. **Example:** `'5:50'`

Specified by	Behavior
n Number Index	Specify the column containing the row names using a positive scalar column index. **Example:** `5`
`''` Unspecified or Empty	Indicate that there are no row names. **Example:** `''`

Data Types: `char` | `single` | `double`

`VariableNamesRange` — Location of variable names
character vector | string scalar | positive scalar integer | `''` empty character array

Location of variable names, specified as a character vector, string scalar, positive scalar integer, or an empty character array. Specify `VariableNamesRange` as one of the values in this table.

Specified by	Behavior
`'`*`Cell`*`'`	Specify the starting cell for the data, using Excel `A1` notation. For example, `A5` is the identifier for the cell at the intersection of column `A` and row `5`. The importing function reads a name for each variable in the data. **Example:** `'A5'`
`'`*`Corner1:Corner2`*`'` Rectangular Range	Specify the exact range to read using the rectangular range form, where *`Corner1`* and *`Corner2`* are two opposing corners that define the region to read. The number of columns must match the number specified in the `NumVariables` property, and the range must span only one row. **Example:** `'A5:K5'`

Specified by	Behavior
'*Row1:Row2*' Row Range	Specify range by identifying the beginning and ending rows using Excel row numbers. Must be a single row. **Example:** `'5:5'`
n Number Index	Specify the row containing the variable names using a positive scalar row index. **Example:** 5
`''` Unspecified or Empty	Indicate that there are no variable names. **Example:** `''`

Data Types: `char` | `single` | `double`

`VariableDescriptionsRange` — Location of variable descriptions
character vector | string scalar | `''` empty character array

Location of variable descriptions, specified as a character vector, string scalar, positive scalar integer, or an empty character array. Specify `VariableDescriptionRange` as one of the values in this table.

Specified by	Behavior
'*Cell*'	Specify the starting cell for the data, using Excel `A1` notation. For example, `A5` is the identifier for the cell at the intersection of column `A` and row `5`. The importing function reads a description for each variable in the data. **Example:** `'A5'`

Specified by	Behavior
`'Corner1:Corner2'` Rectangular Range	Specify the exact range to read using the rectangular range form, where *`Corner1`* and *`Corner2`* are two opposing corners that define the region to read. The number of columns must match the number specified in the `NumVariables` property, and the range must span only one row. **Example:** `'A5:K5'`
`'Row1:Row2'` Row Range	Specify range by identifying the beginning and ending rows using Excel row numbers. Must be a single row. **Example:** `'5:5'`
n Number Index	Specify the row containing the descriptions using a positive scalar row index. **Example:** 5
`''` Unspecified or Empty	Indicate that there are no variable descriptions. **Example:** `''`

Data Types: `char` | `string` | `single` | `double`

`VariableUnitsRange` — Location of variable units
character vector | string scalar | positive scalar integer | `''` empty character array

Location of variable units, specified as a character vector, string scalar, positive scalar integer, or an empty character array. Specify `VariableUnitsRange` as one of the values in this table.

Specified by	Behavior
`'Cell'`	Specify the starting cell for the data, using Excel `A1` notation. For example, `A5` is the identifier for the cell at the intersection of column `A` and row `5`. The importing function reads a unit for each variable in the data. **Example:** `'A5'`
`'Corner1:Corner2'` Rectangular Range	Specify the exact range to read using the rectangular range form, where *`Corner1`* and *`Corner2`* are two opposing corners that define the region to read. The number of columns must match the number specified in the `NumVariables` property, and the range must span only one row. **Example:** `'A5:K5'`
`'Row1:Row2'` Row Range	Specify range by identifying the beginning and ending rows using Excel row numbers. Must be a single row. **Example:** `'5:5'`
n Number Index	Specify the row containing the data units using a positive scalar row index. **Example:** 5
`''` Unspecified or Empty	Indicate that there are no variable units. **Example:** `''`

Data Types: `char` | `string` | `single` | `double`

Replacement Rules

`MissingRule` — Procedure to manage missing data

`'fill'` | `'error'` | `'omitrow'` | `'omitvar'`

Procedure to manage missing data, specified as one of the values in this table.

Missing Rule	Behavior
`'fill'`	Replace missing data with the contents of the `FillValue` property. The `FillValue` property is specified in the `VariableImportOptions` object of the variable being imported. For more information on accessing the `FillValue` property, see `getvaropts`.
`'error'`	Stop importing and display an error message showing the missing record and field.
`'omitrow'`	Omit rows that contain missing data.
`'omitvar'`	Omit variables that contain missing data.

Example: `opts.MissingRule = 'omitrow';`uint32

Data Types: `char` | `string`

`ImportErrorRule` — Procedure to handle import errors
`'fill'` | `'error'` | `'omitrow'` | `'omitvar'`

Procedure to handle import errors, specified as one of the values in this table.

Import Error Rule	Behavior
`'fill'`	Replace the data where the error occurred with the contents of the `FillValue` property. The `FillValue` property is specified in the `VariableImportOptions` object of the variable being imported. For more information on accessing the `FillValue` property, see `getvaropts`.
`'error'`	Stop importing and display an error message showing the error-causing record and field.
`'omitrow'`	Omit rows where errors occur.
`'omitvar'`	Omit variables where errors occur.

Example: `opts.ImportErrorRule = 'omitvar';`

Data Types: `char` | `string`

Object Functions

getvaropts	Get variable import options
setvaropts	Set variable import options
setvartype	Set variable data types
preview	Preview eight rows from file using import options

Examples

Define Import Options for Variable in Spreadsheet File

Create an import options object to read one variable from the file `patients.xls`.

Initialize a `SpreadsheetImportOptions` object, specify the variable name, variable type, and the data starting cell. Then, use the `preview` function with the import options object to preview the data.

```
opts = spreadsheetImportOptions;
opts.VariableNames = 'LastName';
opts.VariableTypes = 'categorical';
opts.DataRange = 'A2';
preview('patients.xls',opts)
```

```
ans=8×1 table
    LastName
    ________

    Smith
    Johnson
    Williams
    Jones
    Brown
    Davis
    Miller
    Wilson
```

Import the variable from the file.

```
oneVar = readtable('patients.xls',opts);
whos oneVar
```

```
  Name          Size            Bytes  Class    Attributes

  oneVar      100x1             13566  table
```

Define Import Options for Multiple Variables in Spreadsheet File

Create an import options object to read multiple variables from `patients.xls`.

Based on the contents of your file, define the number of variables, variable names, variable types, and the data starting location.

```
numVars = 7;
varNames = {'LastName','Gender','Age','Location','Height','Weight','Smoker'} ;
varTypes = {'char','categorical','int32','char','double','double','logical'} ;
dataStartLoc = 'A2';
```

Initialize the import options object `opts` using the `spreadsheetImportOptions` function and your variable information.

```
opts = spreadsheetImportOptions('NumVariables',numVars,...
                                'VariableNames',varNames,...
                                'VariableTypes',varTypes,...
                                'DataRange', dataStartLoc);
```

Preview the data using the `preview` function with the import options.

```
preview('patients.xls',opts)
```

```
ans=8×7 table
      LastName      Gender    Age              Location               Height    Weight
    ____________    ______    ___    ______________________________    ______    ______

    {'Smith'   }    Male      38     {'County General Hospital'  }      71       176
    {'Johnson' }    Male      43     {'VA Hospital'              }      69       163
    {'Williams'}    Female    38     {'St. Mary's Medical Center'}      64       131
    {'Jones'   }    Female    40     {'VA Hospital'              }      67       133
    {'Brown'   }    Female    49     {'County General Hospital'  }      64       119
    {'Davis'   }    Female    46     {'St. Mary's Medical Center'}      68       142
    {'Miller'  }    Female    33     {'VA Hospital'              }      64       142
    {'Wilson'  }    Male      40     {'VA Hospital'              }      68       180
```

Import the data using `readtable`.

```
T = readtable('patients.xls',opts);
whos T
```

```
  Name        Size            Bytes  Class    Attributes

  T         100x7             32332  table
```

Tips

- Introduced in:
  - R2016b — `SpreadsheetImportOptions` object
  - R2018b — `spreadsheetImportOptions` function

See Also

`detectImportOptions` | `readtable`

Import Tool

Import data from file

Description

The **Import Tool** lets you preview and import data from spreadsheet files, delimited text files, and fixed-width text files. You can interactively select the data to import and reuse the script or function that the tool generates to import other similar files.

Open the Import Tool App

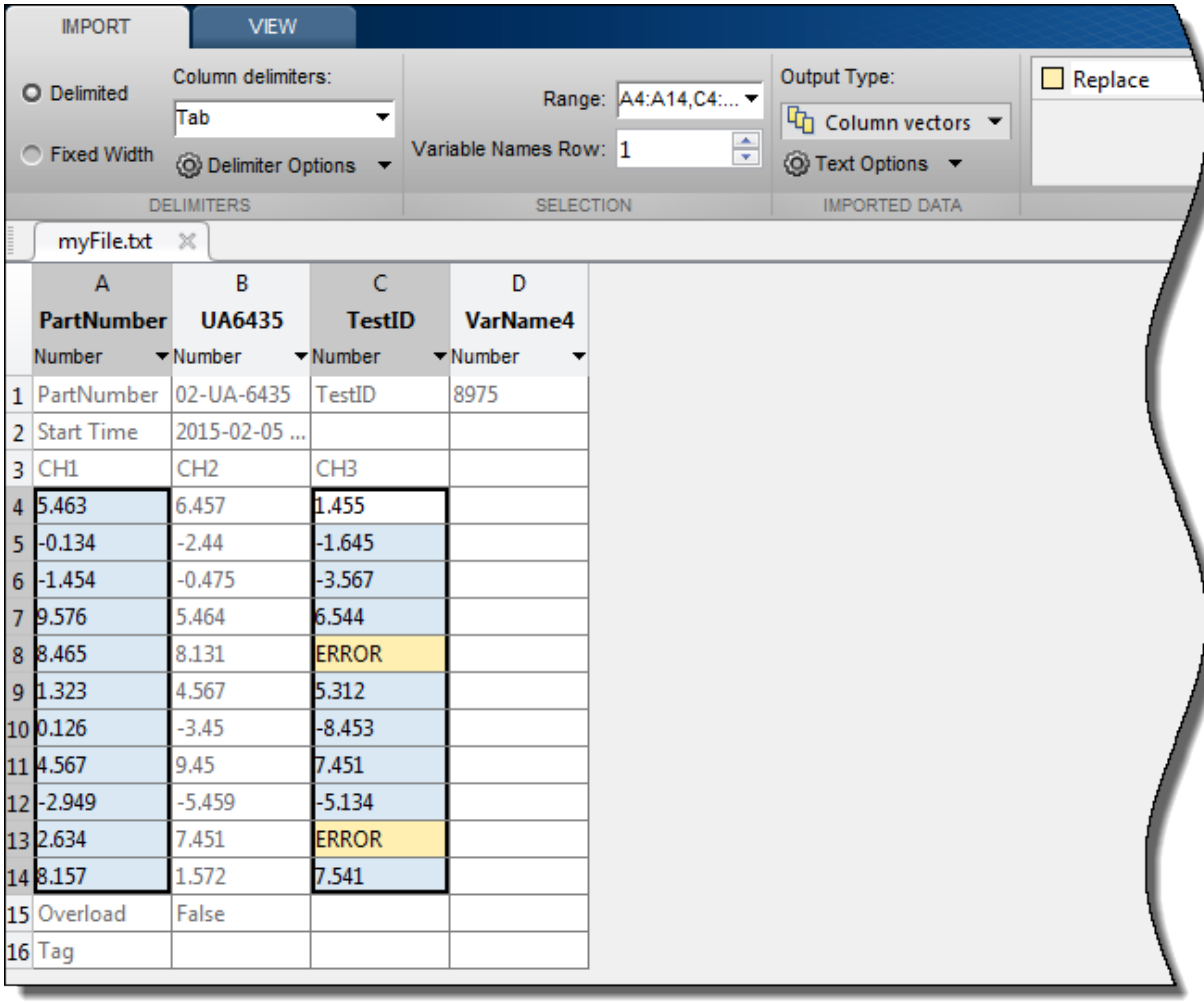
- MATLAB Toolstrip: On the **Home** tab, in the **Variable** section, click **Import Data**.
- MATLAB command prompt: Enter `uiimport(filename)`, where *`filename`* is a character vector specifying the name of a text or spreadsheet file.

Examples

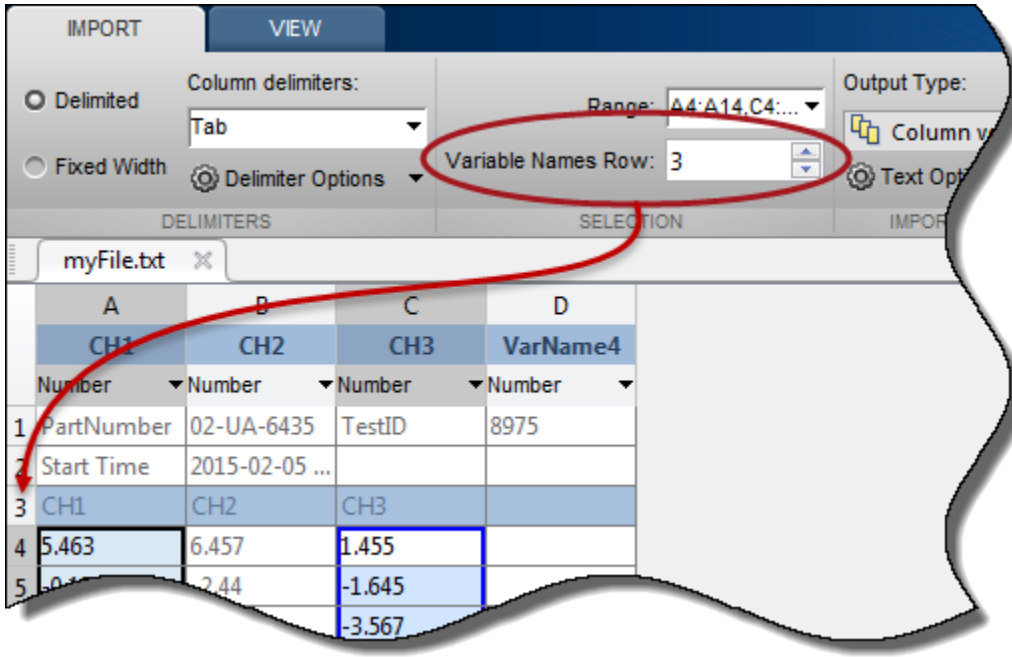
Select Range to Import

Import ranges of data from a tab-delimited text file and replace nonnumeric values in the file.

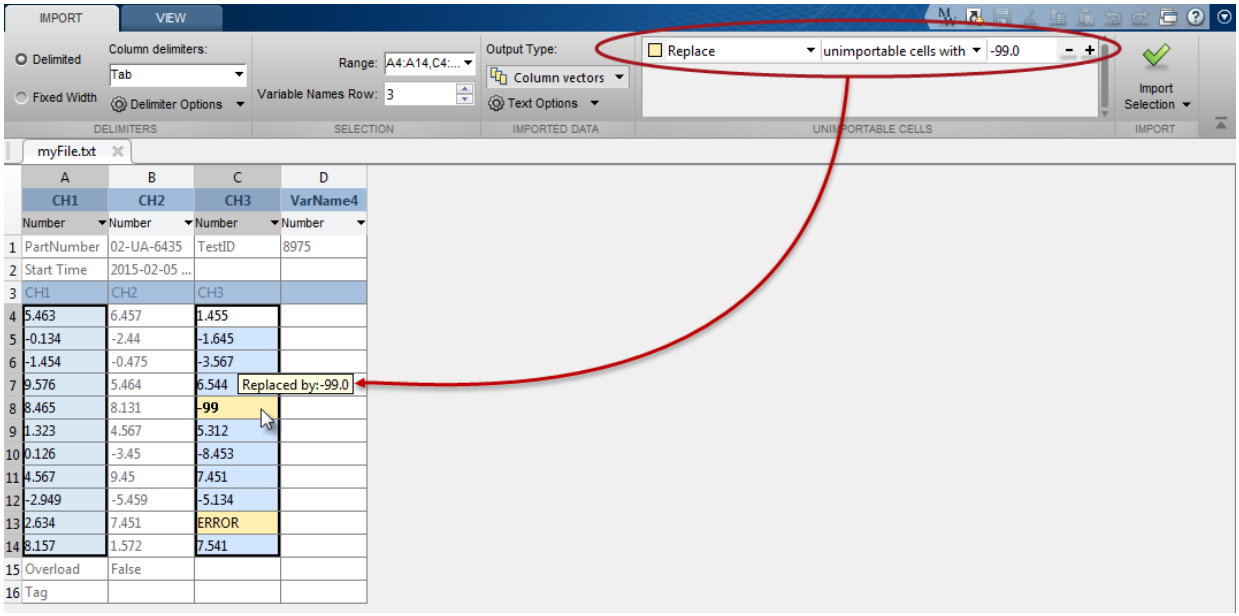
Drag the cursor to select the cells to import. Press **Ctrl** to select noncontiguous ranges.



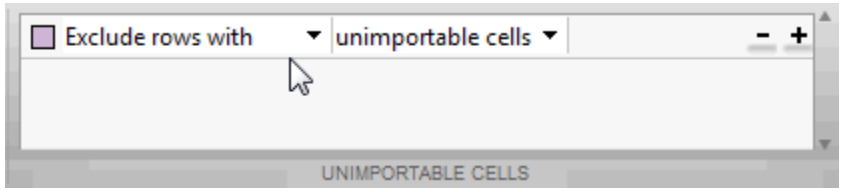
Modify the **Variable Names Row** field to indicate the row you want the Import Tool to use for the variable names.



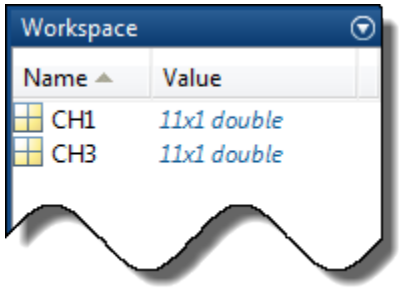
By default, `NaN` replaces unimportable values in numeric cells. However, you can change this value to any other numeric value. Specify how to treat unimportable values on the **Import** tab, in the **Unimportable Cells** section.



Alternatively, exclude any row that contains an unimportable cell from being imported.



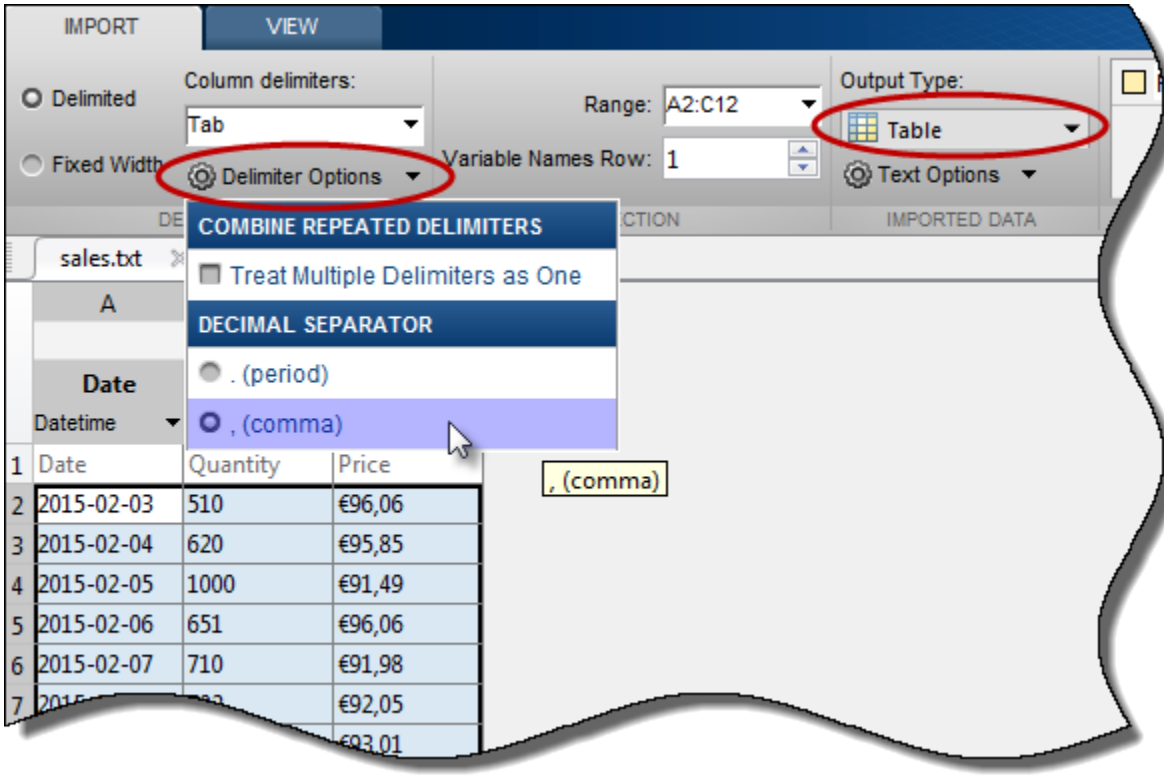
Click **Import Selection** . The new column vectors appear in your workspace.



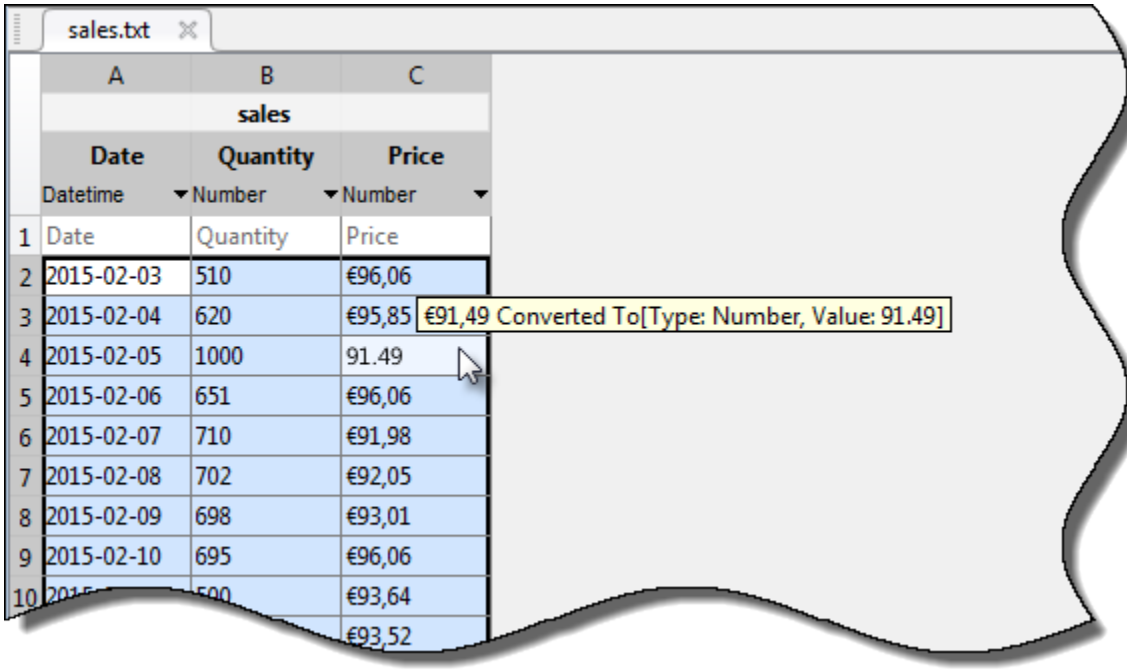
Specify Decimal Separator and Date Format

Import dates and numbers from a text file into a table. Specify the decimal separator for numeric values in the file and a custom format for dates.

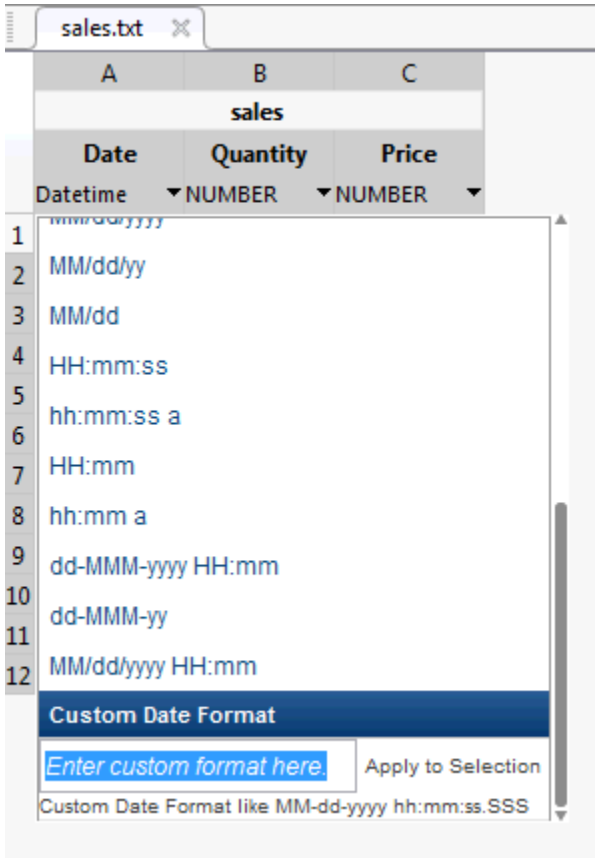
In the **Imported Data** section, select **Table** to import the file data into a `table` variable. In this example, the third column of data contains monetary amounts where the decimal separator is a comma. In the **Delimiters** section, select **Delimiter Options**. Then, under **Decimal Separator**, select **, (comma)**.



The Import Tool appropriately converts the monetary values.



Specify that a column contains date or time data by changing the data type for the column to **DATE/TIME**. Click **more date formats...**. Scroll down the list to enter a custom date format.



Click **Import Selection** to import the table.

- “Read Text File Data Using Import Tool”
- “Read Spreadsheet Data Using Import Tool”
- “Supported File Formats for Import and Export”

Programmatic Use

`uiimport` opens a dialog to interactively load data from a file or the clipboard. MATLAB displays a preview of the data in the file.

`uiimport(filename)` opens the file specified in `filename` using either **Import Tool** or **Import Wizard** depending on the file type. For spreadsheet and text files, `uiimport` opens the file using **Import Tool**. For all other file types, such as image, audio, or MAT-files, `uiimport` opens the file using **Import Wizard**.

`uiimport('-file')` presents the file selection dialog first.

`uiimport('-pastespecial')` presents the clipboard contents first.

`S = uiimport( ___ )` opens the file using **Import Wizard** and stores the resulting variables as fields in the struct `S`.

See Also

Functions

`clipboard` | `load` | `readcell` | `readmatrix` | `readtable` | `readvars` | `textscan`

Topics

"Read Text File Data Using Import Tool"
"Read Spreadsheet Data Using Import Tool"
"Supported File Formats for Import and Export"

imread

Read image from graphics file

Syntax

```
A = imread(filename)
A = imread(filename,fmt)
A = imread( ___ ,idx)
A = imread( ___ ,Name,Value)
[A,map] = imread( ___ )
[A,map,transparency] = imread( ___ )
```

Description

`A = imread(filename)` reads the image from the file specified by `filename`, inferring the format of the file from its contents. If `filename` is a multi-image file, then `imread` reads the first image in the file.

`A = imread(filename,fmt)` additionally specifies the format of the file with the standard file extension indicated by `fmt`. If `imread` cannot find a file with the name specified by `filename`, it looks for a file named *`filename.fmt`*.

`A = imread( ___ ,idx)` reads the specified image or images from a multi-image file. This syntax applies only to GIF, PGM, PBM, PPM, CUR, ICO, TIF, and HDF4 files. You must specify a `filename` input, and you can optionally specify `fmt`.

`A = imread( ___ ,Name,Value)` specifies format-specific options using one or more name-value pair arguments, in addition to any of the input arguments in the previous syntaxes.

`[A,map] = imread( ___ )` reads the indexed image in `filename` into `A` and reads its associated colormap into `map`. Colormap values in the image file are automatically rescaled into the range `[0,1]`.

`[A,map,transparency] = imread( ___ )` additionally returns the image transparency. This syntax applies only to PNG, CUR, and ICO files. For PNG files,

`transparency` is the alpha channel, if one is present. For CUR and ICO files, it is the AND (opacity) mask.

Examples

Read and Display Image

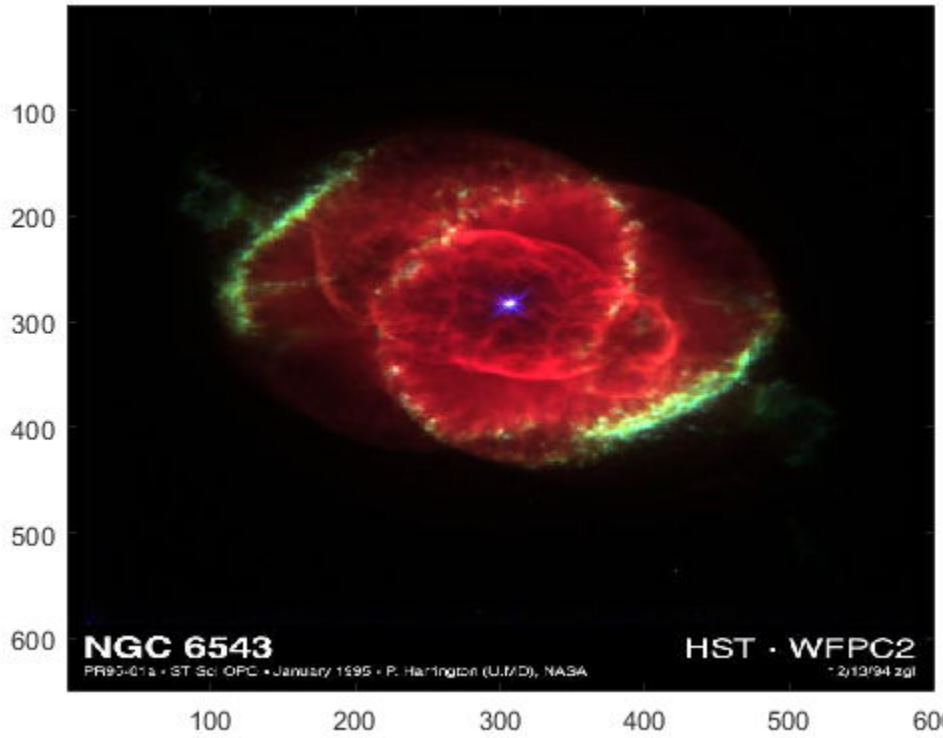
Read a sample image.

```
A = imread('ngc6543a.jpg');
```

`imread` returns a 650-by-600-by-3 array, `A`.

Display the image.

```
image(A)
```



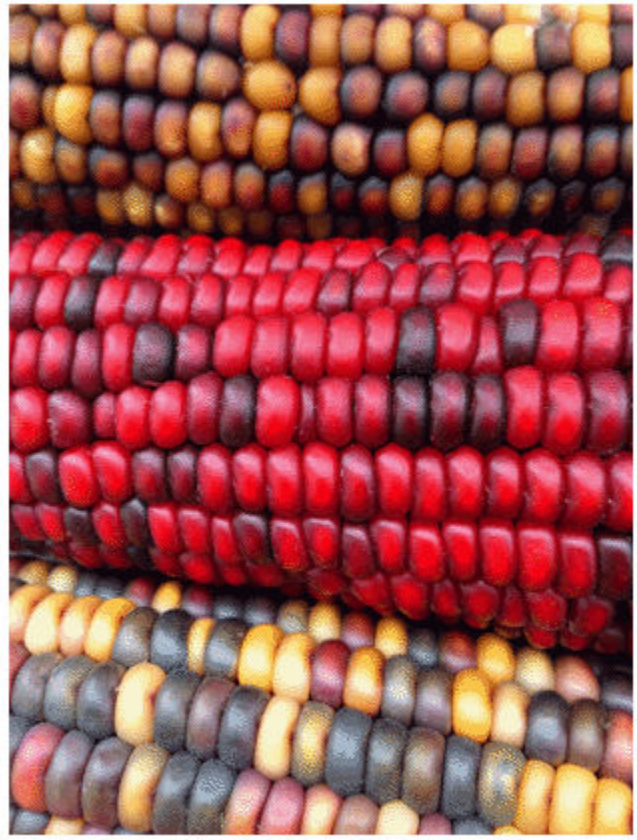
Convert Indexed Image to RGB

Read the first image in the sample indexed image file, `corn.tif`.

```
[X,cmap] = imread('corn.tif');
```

The indexed image `X` is a 415-by-312 array of type `uint8`. The colormap `cmap` is a 256-by-3 matrix of type `double`, therefore there are 256 colors in the indexed image. Display the image.

```
imshow(X,cmap)
```



Convert the indexed image to an RGB image. The result is a 415-by-312-by-3 array of type `double`.

```
RGB = ind2rgb(X,cmap);
```

Check that values of the RGB image are in the range [0, 1].

```
disp(['Range of RGB image is [',num2str(min(RGB(:))),', ',num2str(max(RGB(:))),'].'])
```

```
Range of RGB image is [0.0078431, 0.97647].
```

Read Specific Image in Multipage TIFF File

Read the third image in the sample file, `corn.tif`.

```
[X,map] = imread('corn.tif',3);
```

Return Alpha Channel of PNG Image

Return the alpha channel of the sample image, `peppers.png`.

```
[X,map,alpha] = imread('peppers.png');
whos alpha
```

```
  Name       Size            Bytes  Class     Attributes

  alpha      0x0                 0  double
```

No alpha channel is present, so `alpha` is empty.

Read Specified Region of TIFF Image

Read a specific region of pixels of the sample image, `corn.tif`.

Specify the `'PixelRegion'` parameter with a cell array of vectors indicating the boundaries of the region to read. The first vector specifies the range of rows to read, and the second vector specifies the range of columns to read.

```
A = imread('corn.tif','PixelRegion',{[1,2],[2,5]});
```

`imread` reads the image data in rows 1-2 and columns 2-5 from `corn.tif` and returns the 2-by-4 array, `A`.

Input Arguments

`filename` — Name of graphics file
character vector | string scalar

Name of graphics file, specified as a character vector or string scalar.

Depending on the location of your file, `filename` can take on one of these forms.

Location	Form
Current folder or folder on the MATLAB path	Specify the name of the file in `filename`. **Example:** `'myImage.jpg'`
File in a folder	If the file is not in the current folder or in a folder on the MATLAB path, then specify the full or relative path name. **Example:** `'C:\myFolder\myImage.ext'` **Example:** `'\imgDir\myImage.ext'`
URL	If the file is located by an internet URL, then `filename` must contain the protocol type such as, `http://`. **Example:** `'http://hostname/path_to_file/my_image.jpg'`

<table>
<tr><th>Location</th><th>Form</th></tr>
<tr><td>Remote Location</td><td>If the file is stored at a remote location, then <code>filename</code> must contain the full path of the file specified as a uniform resource locator (URL) of the form:

<code>scheme_name://path_to_file/my_file.ext</code>

Based on your remote location, <code>scheme_name</code> can be one of the values in this table.
<table>
<tr><th>Remote Location</th><th>scheme_name</th></tr>
<tr><td>Amazon S3</td><td>s3</td></tr>
<tr><td>Windows Azure Blob Storage</td><td>wasb, wasbs</td></tr>
<tr><td>HDFS</td><td>hdfs</td></tr>
</table>
For more information, see “Work with Remote Data”.

Example: <code>'s3://bucketname/path_to_file/my_image.jpg'</code></td></tr>
</table>

For information on the bit depths, compression schemes, and color spaces supported for each file type, see “Algorithms” on page 1-7564.

Data Types: `char` | `string`

`fmt` — Image format

character vector | string scalar

Image format, specified as a character vector or string scalar indicating the standard file extension. Call `imformats` to see a list of supported formats and their file extensions.

Example: `'png'`

Data Types: `char` | `string`

`idx` — Image to read

integer scalar | vector of integers

Image to read, specified as an integer scalar or, for GIF files, a vector of integers. For example, if `idx` is 3, then `imread` returns the third image in the file. For a GIF file, if `idx` is `1:5`, then `imread` returns only the first five frames. The `idx` argument is supported only for multi-image GIF, CUR, ICO, and HDF4 files.

When reading multiple frames from the same GIF file, specify `idx` as a vector of frames or use the `'Frames','all'` name-value pair argument. Because of the way that GIF files are structured, these syntaxes provide faster performance compared to calling `imread` in a loop.

For HDF4 files, `idx` corresponds to the reference number of the image to read. Reference numbers do not necessarily correspond to the order of the images in the file. You can use `imfinfo` to match image order with reference number.

Example: `3`

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Index',5` reads the fifth image of a TIFF file.

GIF Files

`Frames` — Frame to read
1 (default) | positive integer | vector of integers | `'all'`

Frames to read, specified as the comma-separated pair consisting of `'Frames'` and a positive integer, a vector of integers, or `'all'`. For example, if you specify the value 3, `imread` reads the third frame in the file. If you specify `'all'`, then `imread` reads all frames and returns them in the order in which they appear in the file.

Example: `'frames',5`

JPEG 2000 Files

`PixelRegion` — Subimage to read
cell array in the form `{rows,cols}`

Subimage to read, specified as the comma-separated pair consisting of `'PixelRegion'` and a cell array of the form `{rows,cols}`. The `rows` input specifies the range of rows to read. The `cols` input specifies the range of columns to read. Both `rows` and `cols` must be two-element vectors containing 1-based indices. For example, `'PixelRegion',{[1`

2],[3 4]} reads the subimage bounded by rows 1 and 2 and columns 3 and 4 in the image data. If the 'ReductionLevel' value is greater than 0, then rows and cols are coordinates of the subimage.

Example: 'PixelRegion',{[1 100],[4 500]}

ReductionLevel — Reduction of image resolution
0 (default) | nonnegative integer

Reduction of the image resolution, specified as the comma-separated pair consisting of 'ReductionLevel' and a nonnegative integer. For reduction level *L*, the image resolution is reduced by a factor of 2^*L*. The reduction level is limited by the total number of decomposition levels as specified by the 'WaveletDecompositionLevels' field in the output of the imfinfo function.

Example: 'ReductionLevel',5

Data Types: single | double

V79Compatible — Compatibility with MATLAB 7.9 (R2009b) and earlier
false (default) | true

Compatibility with MATLAB 7.9 (R2009b) and earlier, specified as the comma-separated pair consisting of 'V79Compatible' and either true or false. If you specify true, then the returned grayscale or RGB image is consistent with previous versions of imread (MATLAB 7.9 (R2009b) and earlier).

Example: 'V79Compatible',true

Data Types: logical

PNG Files

BackgroundColor — Background color
'none' | integer | 3-element vector of integers

Background color, specified as 'none', an integer, or a three-element vector of integers. If BackgroundColor is 'none', then imread does not perform any compositing. Otherwise, imread blends transparent pixels with the background color.

- If the input image is indexed, then the value of BackgroundColor must be an integer in the range [1,P], where P is the colormap length.
- If the input image is grayscale, then the value of BackgroundColor must be an integer in the range [0,1].

- If the input image is RGB, then the value of `BackgroundColor` must be a three-element vector with values in the range `[0,1]`.

The default value for `BackgroundColor` depends on the presence of the `transparency` output argument and the image type:

- If you request the `transparency` output argument, then the default value of `BackgroundColor` is `'none'`.
- If you do not request the `transparency` output and the PNG file contains a background color chunk, then that color is the default value for `BackgroundColor`.
- If you do not request the `transparency` output and the file does not contain a background color chunk, then the default value for `BackgroundColor` is `1` for indexed images, `0` for grayscale images, and `[0 0 0]` for truecolor (RGB) images.

TIFF Files

`Index` — Image to read

1 (default) | positive integer

Image to read, specified as the comma-separated pair consisting of `'Index'` and a positive integer. For example, if the value of `Index` is 3, then `imread` reads the third image in the file.

Data Types: `single` | `double`

`Info` — Information about image

structure array

Information about the image, specified as the comma-separated pair consisting of `'Info'` and a structure array returned by the `imfinfo` function. Use the `Info` name-value pair argument to help `imread` locate the images in a multi-image TIFF file more quickly.

Data Types: `struct`

`PixelRegion` — Region boundary

cell array

Region boundary, specified as the comma-separated pair consisting of `'PixelRegion'` and a cell array of the form `{rows,cols}`. The `rows` input specifies the range of rows to read. The `cols` input specifies the range of columns to read. `rows` and `cols` must be either two-element or three-element vectors of 1-based indices. A two-element vector specifies the first and last rows or columns to read. For example, `'PixelRegion',{[1`

`2],[3 4]}` reads the region bounded by rows 1 and 2 and columns 3 and 4 in the image data.

A three-element vector must be in the form `[start increment stop]`, where `start` is the first row or column to read, `increment` is an incremental value, and `stop` is the last row or column to read. This syntax allows image downsampling. For example, `'PixelRegion',{[1 2 10],[4 3 12]}` reads the region bounded by rows 1 and 10 and columns 4 and 12, and samples data from every 2 pixels in the vertical direction, and every 3 pixels in the horizontal direction.

Example: `'PixelRegion',{[1 100],[4 500]}`

Data Types: `cell`

Output Arguments

`A` — Image data
array

Image data, returned as an array.

- If the file contains a grayscale image, then A is an `m`-by-`n` array.
- If the file contains an indexed image, then A is an `m`-by-`n` array of index values corresponding to the color at that index in `map`.
- If the file contains a truecolor image, then A is an `m`-by-`n`-by-3 array.
- If the file is a TIFF file containing color images that use the CMYK color space, then A is an `m`-by-`n`-by-4 array.

The class of `A` depends on the image format and the bit depth of the image data. For more information, see "Algorithms" on page 1-7564

`map` — Colormap
`m`-by-3 matrix

Colormap associated with the indexed image data in `A`, returned as an `m`-by-3 matrix of class `double`.

`transparency` — Transparency information
matrix

Transparency information, returned as a matrix. For PNG files, `transparency` is the alpha channel, if present. If no alpha channel is present, or if you specify the `'BackgroundColor'` name-value pair argument, then `transparency` is empty. For CUR and ICO files, `transparency` is the AND mask. For cursor files, this mask sometimes contains the only useful data.

More About

Bit Depth

Bit depth is the number of bits used to represent each image pixel.

Bit depth is calculated by multiplying the bits-per-sample with the samples-per-pixel. Thus, a format that uses 8 bits for each color component (or sample) and three samples per pixel has a bit depth of 24. Sometimes the sample size associated with a bit depth can be ambiguous. For example, does a 48-bit bit depth represent six 8-bit samples, four 12-bit samples, or three 16-bit samples? See "Algorithms" on page 1-7564 for sample size information to avoid this ambiguity.

Algorithms

For most image file formats, `imread` uses 8 or fewer bits per color plane to store image pixels. This table lists the class of the returned image array, `A`, for the bit depths used by the file formats.

Bit Depth in File	Class of Array Returned by `imread`
1 bit per pixel	`logical`
2 to 8 bits per color plane	`uint8`
9 to 16 bits per pixel	`uint16` (`BMP`, `JPEG`, `PNG`, and `TIFF`) For the 16-bit BMP packed format (5-6-5), MATLAB returns `uint8`

The following sections provide information about the support for specific formats, listed in alphabetical order by format name.

“BMP — Windows Bitmap” on page 1-7565	“JPEG — Joint Photographic Experts Group” on page 1-7567	“PNG — Portable Network Graphics” on page 1-7568
“CUR — Cursor File” on page 1-7565	“JPEG 2000 — Joint Photographic Experts Group 2000” on page 1-7567	“PPM — Portable Pixmap” on page 1-7569
“GIF — Graphics Interchange Format” on page 1-7566	“PBM — Portable Bitmap” on page 1-7567	“RAS — Sun Raster” on page 1-7569
“HDF4 — Hierarchical Data Format” on page 1-7566	“PCX — Windows Paintbrush” on page 1-7568	“TIFF — Tagged Image File Format” on page 1-7569
“ICO — Icon File” on page 1-7566	“PGM — Portable Graymap” on page 1-7568	“XWD — X Window Dump” on page 1-7570

BMP — Windows Bitmap

This table lists supported bit depths and the data type of the output image data array.

Supported Bit Depths	No Compression	RLE Compression	Output Class	Notes
1 bit	✓	–	`logical`	
4 bit	✓	✓	`uint8`	
8 bit	✓	✓	`uint8`	
16 bit	✓	–	`uint8`	1 sample/pixel
24 bit	✓	–	`uint8`	3 samples/pixel
32 bit	✓	–	`uint8`	3 samples/pixel (1 byte padding)

CUR — Cursor File

This table lists supported bit depths and the data type of the output image data array.

Supported Bit Depths	No Compression	Compression	Output Class
1 bit	✓	–	`logical`
4 bit	✓	–	`uint8`
8 bit	✓	–	`uint8`

Note By default, Microsoft Windows cursors are 32-by-32 pixels. Since MATLAB pointers must be 16-by-16, you might need to scale your image. If you have Image Processing Toolbox™, you can use the `imresize` function.

GIF — Graphics Interchange Format

This table lists supported bit depths and the data type of the output image data array.

Supported Bit Depths	No Compression	Compression	Output Class
1 bit	✓	–	`logical`
2 bit to 8 bit	✓	–	`uint8`

HDF4 — Hierarchical Data Format

This table lists supported bit depths and the data type of the output image data array.

Supported Bit Depths	Raster Image with colormap	Raster image without colormap	Output Class	Notes
8 bit	✓	✓	`uint8`	
24 bit	–	✓	`uint8`	3 samples/pixel

ICO — Icon File

See “CUR — Cursor File” on page 1-7565

JPEG — Joint Photographic Experts Group

`imread` reads any baseline JPEG image, as well as JPEG images with some commonly used extensions. For information on JPEG 2000 file support, see JPEG 2000 on page 1-7567.

Supported Bits per Sample	Lossy Compression	Lossless Compression	Output Class	Notes
8 bit	✓	✓	`uint8`	Grayscale or RGB
12 bit	✓	✓	`uint16`	Grayscale or RGB
16 bit	–	✓	`uint16`	Grayscale

JPEG 2000 — Joint Photographic Experts Group 2000

For information about JPEG files, see JPEG on page 1-7567.

Note Indexed JPEG 2000 images are not supported. Only JP2 compatible color spaces are supported for JP2/JPX files. By default, all image channels are returned in the order they are stored in the file.

Supported Bits per Sample	Lossy Compression	Lossless Compression	Output Class	Notes
1 bit	✓	✓	`logical`	Grayscale only
2 bit to 8 bit	✓	✓	`uint8` or `int8`	Grayscale or RGB
9 bit to 16 bit	✓	✓	`uint16` or `int16`	Grayscale or RGB

PBM — Portable Bitmap

This table lists supported bit depths and the data type of the output image data array.

Supported Bit Depths	Raw Binary	ASCII (Plain) Encoded	Output Class
1 bit	✓	✓	`logical`

PCX — Windows Paintbrush

This table lists supported bit depths and the data type of the output image data array.

Supported Bit Depths	Output Class	Notes
1 bit	`logical`	Grayscale only
8 bit	`uint8`	Grayscale or indexed
24 bit	`uint8`	RGB Three 8-bit samples/pixel

PGM — Portable Graymap

This table lists supported bit depths and the data type of the output image data array.

Supported Bit Depths	Raw Binary	ASCII (Plain) Encoded	Output Class	Notes
8 bit	✓	–	`uint8`	
16 bit	✓	–	`uint16`	
Arbitrary	–	✓	1-bit to 8-bit: `uint8` 9-bit to 16-bit: `uint16`	Values are scaled

PNG — Portable Network Graphics

This table lists supported bit depths and the data type of the output image data array.

Supported Bit Depths	Output Class	Notes
1 bit	`logical`	Grayscale
2 bit	`uint8`	Grayscale

Supported Bit Depths	Output Class	Notes
4 bit	`uint8`	Grayscale
8 bit	`uint8`	Grayscale or Indexed
16 bit	`uint16`	Grayscale or Indexed
24 bit	`uint8`	RGB Three 8-bit samples/pixel.
48 bit	`uint16`	RGB Three 16-bit samples/pixel.

PPM — Portable Pixmap

This table lists supported bit depths and the data type of the output image data array.

Supported Bit Depths	Raw Binary	ASCII (Plain) Encoded	Output Class
Up to 16 bit	✓	–	`uint8`
Arbitrary	–	✓	

RAS — Sun Raster

This table lists supported bit depths and the data type of the output image data array.

Supported Bit Depths	Output Class	Notes
1 bit	`logical`	Bitmap
8 bit	`uint8`	Indexed
24 bit	`uint8`	RGB Three 8-bit samples/pixel
32 bit	`uint8`	RGB with Alpha Four 8-bit samples/pixel

TIFF — Tagged Image File Format

`imread` reads most images supported by the TIFF specification or LibTIFF. The `imread` function supports these TIFF capabilities:

- Any number of samples per pixel
- CCITT group 3 and 4 FAX, Packbits, JPEG, LZW, Deflate, ThunderScan compression, and uncompressed images
- Logical, grayscale, indexed color, truecolor and hyperspectral images
- RGB, CMYK, CIELAB, ICCLAB color spaces. If the color image uses the CMYK color space, `A` is an `m`-by-`n`-by-4 array. To determine which color space is used, use `imfinfo` to get information about the graphics file and look at the value of the `PhotometricInterpretation` field. If a file contains CIELAB color data, `imread` converts it to ICCLAB before bringing it into the MATLAB workspace. This conversion is necessary because 8-bit or 16-bit TIFF CIELAB-encoded values use a mixture of signed and unsigned data types that cannot be represented as a single MATLAB array.
- Data organized into tiles or scanlines

`imread` reads and converts TIFF images as follows:

- YCbCr images are converted into the RGB colorspace.
- All grayscale images are read as if black = `0`, white = largest value.
- 1-bit images are returned as class `logical`.
- 16-bit floating-point images are returned as class `single`.
- CIELab images are converted into ICCLab colorspace.

For copyright information, open the `libtiffcopyright.txt` file.

XWD — X Window Dump

This table lists the supported bit depths, compression, and output classes for XWD files.

Supported Bit Depths	ZPixmaps	XYBitmaps	XYPixmaps	Output Class
1 bit	✓	–	✓	`logical`
8 bit	✓	–	–	`uint8`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supports reading of 8-bit JPEG images only. The input argument `filename` must be a valid absolute path or relative path.
- This function generates code that uses a precompiled, platform-specific shared library (Image Processing Toolbox).
- In a MATLAB Function block, the input argument `filename` must be a compile-time constant.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Supports reading of 8-bit JPEG images only. The input argument `filename` must be a valid absolute path or relative path.
- This function generates code that uses a precompiled, platform-specific shared library (Image Processing Toolbox).
- In a MATLAB Function block, the input argument `filename` must be a compile-time constant.

See Also

`fread` | `image` | `imfinfo` | `imformats` | `imwrite` | `ind2rgb`

Topics

"Importing Images"

Introduced before R2006a

imresize

Resize image

Syntax

```
B = imresize(A,scale)
B = imresize(A,[numrows numcols])
[Y,newmap] = imresize(X,map, ___ )
___ = imresize( ___ ,method)
___ = imresize( ___ ,Name,Value)
```

Description

`B = imresize(A,scale)` returns image `B` that is `scale` times the size of `A`. The input image `A` can be a grayscale, RGB, or binary image. If `A` has more than two dimensions, `imresize` only resizes the first two dimensions. If `scale` is in the range [0, 1], `B` is smaller than `A`. If `scale` is greater than 1, `B` is larger than `A`. By default, `imresize` uses bicubic interpolation.

`B = imresize(A,[numrows numcols])` returns image `B` that has the number of rows and columns specified by the two-element vector `[numrows numcols]`.

`[Y,newmap] = imresize(X,map, ___ )` resizes the indexed image `X` where `map` is the colormap associated with the image. By default, `imresize` returns a new, optimized colormap (`newmap`) with the resized image. To return a colormap that is the same as the original colormap, use the `'Colormap'` parameter.

`___ = imresize( ___ ,method)` specifies the interpolation method used.

`___ = imresize( ___ ,Name,Value)` returns the resized image where `Name,Value` pairs control various aspects of the resizing operation.

Examples

Shrink Image By Factor of Two Using Default Interpolation Method

Load image into the workspace.

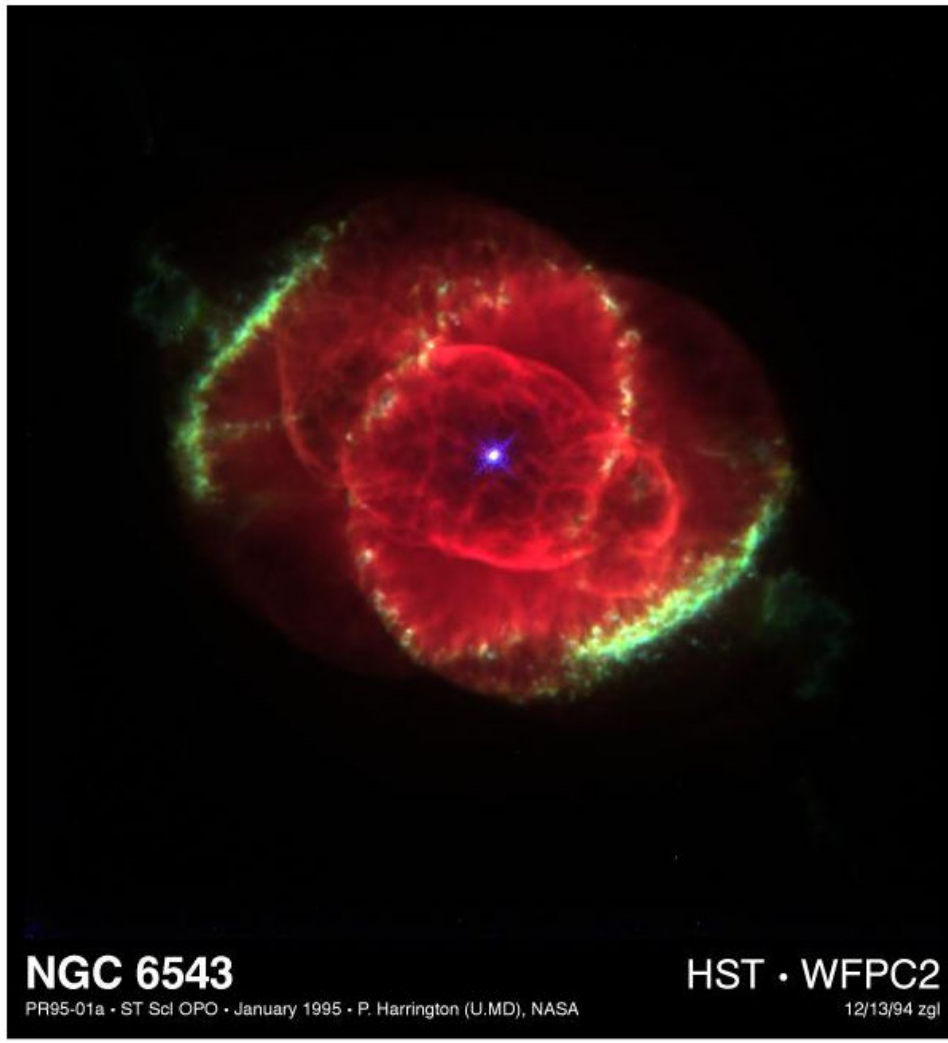
```
I = imread('ngc6543a.jpg');
```

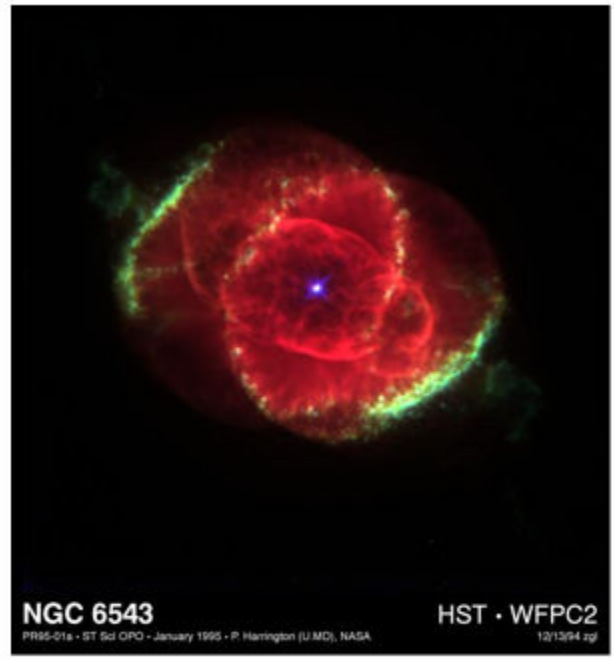
Shrink the image by a factor of two.

```
J = imresize(I, 0.5);
```

Display the original image and the resized image.

```
figure, imshow(I), figure, imshow(J)
```


NGC 6543
HST • WFPC2
PR95-01a • ST ScI OPO • January 1995 • P. Harrington (U.MD), NASA
12/13/94 zgl



Shrink By Factor of Two Using Nearest-Neighbor Interpolation

Load an image into the workspace.

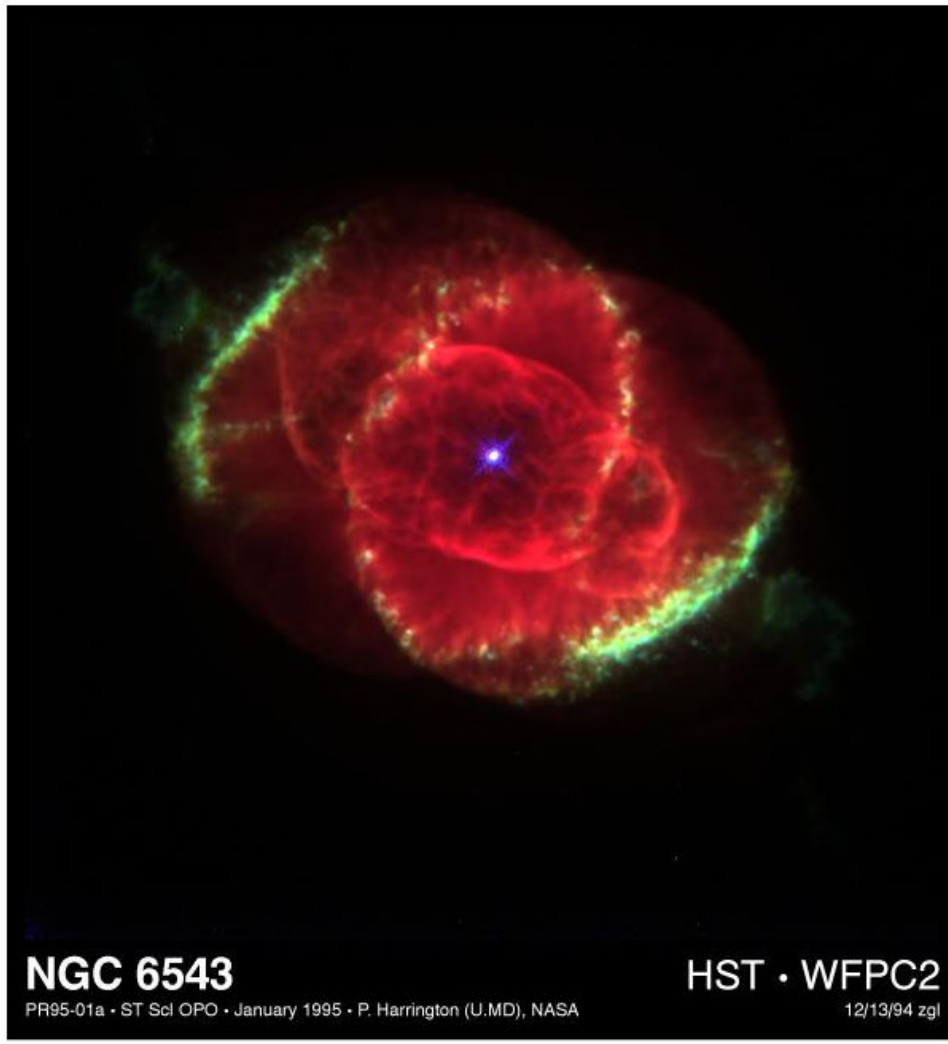
```
I = imread('ngc6543a.jpg');
```

Shrink by factor of two using nearest-neighbor interpolation. This is the fastest method, but it has the lowest quality.

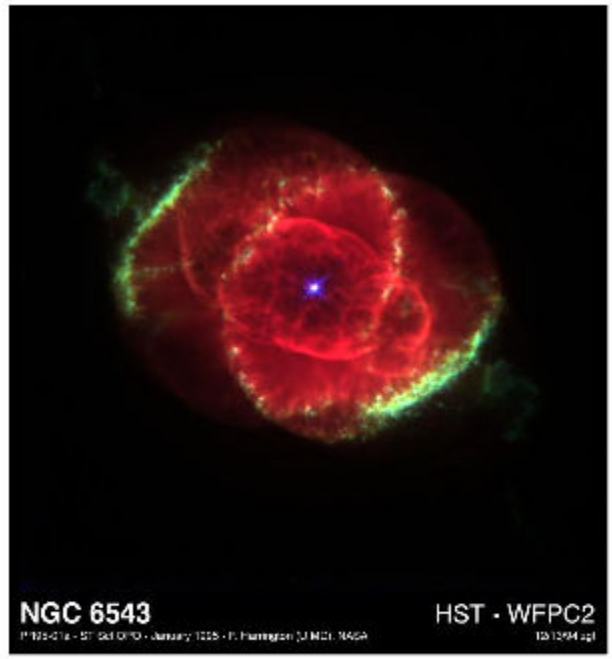
```
J = imresize(I, 0.5, 'nearest');
```

Display the original image and the resized image.

```
figure
imshow(I)
```



```
figure
imshow(J)
```



Resize RGB Image

Read an RGB image into the workspace.

```
RGB = imread('peppers.png');
```

Resize the RGB image to have 64 rows. `imresize` calculates the number of columns automatically.

```
RGB2 = imresize(RGB, [64 NaN]);
```

Display the original image and the resized image.

```
figure
imshow(RGB)
```

```
figure
imshow(RGB2)
```

Resize Indexed Image

Read an indexed image into the workspace.

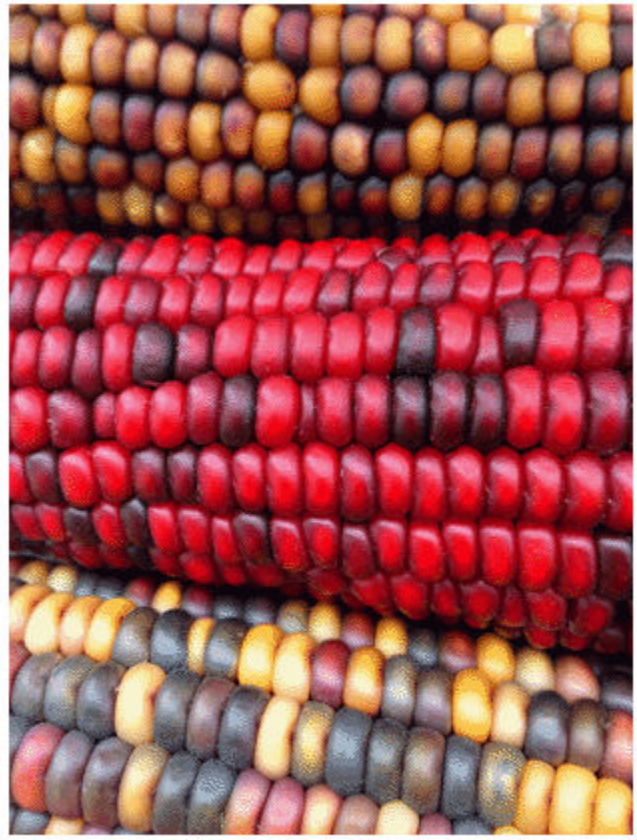
```
[X, map] = imread('corn.tif');
```

Shrink the indexed image by a factor of two.

```
[Y, newmap] = imresize(X, map, 0.5);
```

Display the original image and the resized image.

```
figure
imshow(X,map)
```



```
figure
imshow(Y, newmap)
```

Input Arguments

A — Image to be resized
real, nonsparse numeric or logical array

Image to be resized, specified as a real, nonsparse numeric array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

`scale` — Resize factor
real, numeric scalar

Resize factor, specified as a real, numeric scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`[numrows numcols]` — Row and column dimensions of output image
two-element numeric vector of positive values'

Row and column dimensions of output image, specified as a two-element numeric vector of positive values. Either `numrows` or `numcols` can be `NaN`, in which case `imresize` computes the number of rows or columns automatically to preserve the image aspect ratio.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`X` — Indexed image to be resized
real, nonsparse numeric array

Indexed image to be resized, specified as a real, nonsparse numeric array.

Example: `[X2, newmap] = imresize(X,map,0.75);`

Data Types: `double` | `uint8` | `uint16`

`map` — Colormap associated with indexed image
m-by-3 numeric array

Colormap associated with indexed image, *m*-by-3 numeric array.

Data Types: `double`

`method` — Interpolation method
`'bicubic'` (default) | character vector | two-element cell array

Interpolation method, specified as a character vector or two-element cell array.

When `method` is a character vector, it identifies a particular method or named interpolation kernel, listed in the following table.

Method	Description
`'nearest'`	Nearest-neighbor interpolation; the output pixel is assigned the value of the pixel that the point falls within. No other pixels are considered.
`'bilinear'`	Bilinear interpolation; the output pixel value is a weighted average of pixels in the nearest 2-by-2 neighborhood

Method	Description
`'bicubic'`	Bicubic interpolation; the output pixel value is a weighted average of pixels in the nearest 4-by-4 neighborhood **Note** Bicubic interpolation can produce pixel values outside the original range.
Interpolation Kernel	**Description**
`'box'`	Box-shaped kernel
`'triangle'`	Triangular kernel (equivalent to `'bilinear'`)
`'cubic'`	Cubic kernel (equivalent to `'bicubic'`)
`'lanczos2'`	Lanczos-2 kernel
`'lanczos3'`	Lanczos-3 kernel

When `method` is a two-element cell array, it defines a custom interpolation kernel. The cell array has the form $\{f,w\}$, where f is a function handle for a custom interpolation kernel and w is the width of the custom kernel. $f(x)$ must be zero outside the interval $-w/2 <= x < w/2$. The function handle f can be called with a scalar or a vector input. For user-specified interpolation kernels, the output image can have some values slightly outside the range of pixel values in the input image.

Data Types: `char` | `cell`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `I2 = imresize(I,0.5,'Antialiasing',false);`

`Antialiasing` — Perform antialiasing when shrinking an image
`true` | `false`

Perform antialiasing when shrinking an image, specified as the comma-separated pair consisting of `'Antialiasing'` and the logical Boolean value `true` or `false`. The default value depends on the interpolation method. If the method is nearest-neighbor

('nearest'), the default is false. For all other interpolation methods, the default is true.

Data Types: logical

Colormap — Return optimized colormap
'optimized' (default) | 'original'

Return optimized colormap, specified as the comma-separated pair consisting of 'Colormap' and the character vector 'optimized' or 'original'. (Indexed images only). If set to 'original', the output colormap (newmap) is the same as the input colormap (map). If set to 'optimized', imresize returns a new optimized colormap.

Data Types: char

Dither — Perform color dithering
true (default) | false

Perform color dithering, specified as the comma-separated pair consisting of 'Dither' and the logical Boolean value true or false. (Indexed images only).

In dithering, you apply a form of noise to the image to randomize quantization error and prevent large-scale patterns.

Data Types: logical

Method — Interpolation method
'bicubic' (default) | character vector | cell array

Interpolation method, specified as the comma-separated pair consisting of 'Method' and a character vector or two-element cell array. For details, see method.

Data Types: char | cell

OutputSize — Size of output image
two-element numeric vector

Size of the output image, specified as the comma-separated pair consisting of 'OutputSize' and a two-element vector of the form [numrows numcols].

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Scale — Resize scale factor
positive numeric scalar | two-element vector of positive values

Resize scale factor, specified as the comma-separated pair consisting of `'Scale'` and a positive numeric scalar or two-element vector of positive values.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

B — Resized image
real, nonsparse numeric array

Resized image, returned as a real, nonsparse numeric array, the same class as the input image.

Y — Resized indexed image
real, nonsparse numeric array

Resized indexed image, returned as a real, nonsparse numeric array, the same class as the input image.

`newmap` — Optimized colormap
m-by-3 numeric array

Optimized colormap, returned as an *m*-by-3 numeric array.

Tips

- If the size of the output image is not an integer, `imresize` does not use the scale specified. `imresize` uses `ceil` when calculating the output image size.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Syntaxes that support indexed images are not supported, including the named parameters `'Colormap'` and `'Dither'`.
- Custom interpolation kernels are not supported.
- All parameter-value pairs must be compile-time constants.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- `gpuArray` input must be of type `single` or `double`.
- Only bicubic interpolation is supported on GPU and the function always performs antialiasing.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`gpuArray` | `imresize3` | `imrotate` | `imtransform` | `interp2` | `tformarray`

Introduced before R2006a

imshow

Display image

Syntax

```
imshow(I)
imshow(I,[low high])
imshow(I,[])
imshow(RGB)
imshow(BW)
imshow(X,map)
imshow(filename)
imshow(___,Name,Value)

himage = imshow( ___ )
```

Description

`imshow(I)` displays the grayscale image `I` in a figure. `imshow` uses the default display range for the image data type and optimizes figure, axes, and image object properties for image display.

`imshow(I,[low high])` displays the grayscale image `I`, specifying the display range as a two-element vector, `[low high]`. For more information, see the `DisplayRange` parameter.

`imshow(I,[])` displays the grayscale image `I`, scaling the display based on the range of pixel values in `I`. `imshow` uses `[min(I(:)) max(I(:))]` as the display range. `imshow` displays the minimum value in `I` as black and the maximum value as white. For more information, see the `DisplayRange` parameter.

`imshow(RGB)` displays the truecolor image `RGB` in a figure.

`imshow(BW)` displays the binary image `BW` in a figure. For binary images, `imshow` displays pixels with the value `0` (zero) as black and `1` as white.

`imshow(X,map)` displays the indexed image `X` with the colormap `map`. A colormap matrix can have any number of rows, but it must have exactly 3 columns. Each row is interpreted as a color, with the first element specifying the intensity of red, the second green, and the third blue. Color intensity can be specified on the interval [0, 1].

`imshow(filename)` displays the image stored in the graphics file specified by `filename`.

`imshow(___,Name,Value)` displays an image, using name-value pairs to control aspects of the operation.

`himage = imshow( ___ )` returns the image object created by `imshow`.

Examples

Display Grayscale, RGB, Indexed, or Binary Image

Display a grayscale, RGB (truecolor), indexed or binary image using `imshow`. MATLAB® includes a TIF file, named `corn.tif`, that contains three images: a grayscale image, an indexed image, and a truecolor (RGB) image. This example creates a binary image from the grayscale image.

Display a Grayscale Image

Read the grayscale image from the `corn.tif` file into the MATLAB workspace. The grayscale version of the image is the third image in the file.

```
corn_gray = imread('corn.tif',3);
```

Display the grayscale image using `imshow`.

```
imshow(corn_gray)
```

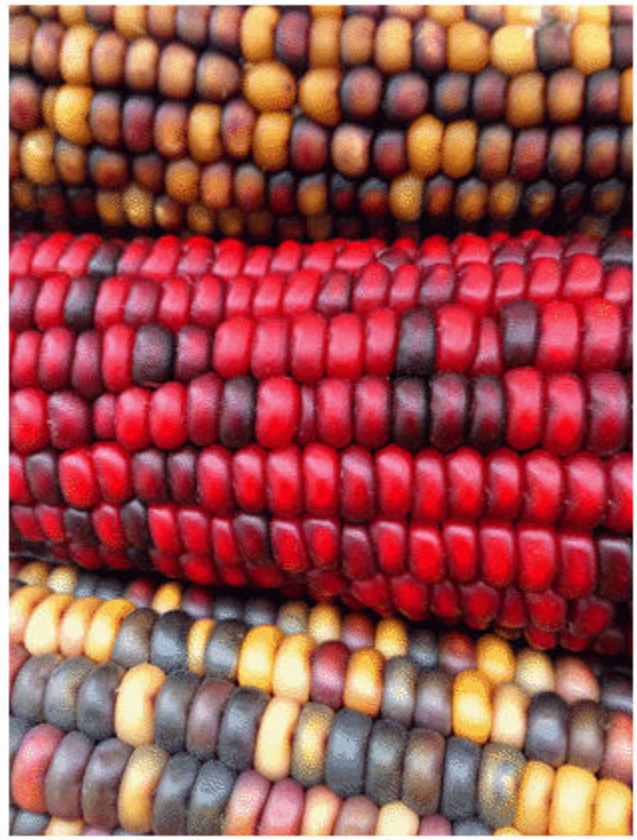
Display an Indexed Image

Read the indexed image from the `corn.tif` file into the MATLAB workspace. The indexed version of the image is the first image in the file.

```
[corn_indexed,map] = imread('corn.tif',1);
```

Display the indexed image using `imshow`.

```
imshow(corn_indexed,map)
```



Display an RGB Image

Read the RGB image from the `corn.tif` file into the MATLAB workspace. The RGB version of the image is the second image in the file.

```
[corn_rgb] = imread('corn.tif',2);
```

Display the RGB image using `imshow`.

```
imshow(corn_rgb)
```

Display a Binary Image

Read the grayscale image from the `corn.tif` file into the MATLAB workspace and use thresholding to convert it into a binary image. The grayscale version of the image is the third image in the file.

```
[corn_gray] = imread('corn.tif',3);
```

Determine the mean value of pixels in the grayscale image.

```
meanIntensity = mean(corn_gray(:));
```

Create a binary image by thresholding, using the mean intensity value as the threshold.

```
corn_binary = corn_gray > meanIntensity;
```

Display the binary image using `imshow`.

```
imshow(corn_binary)
```

Display Image from File

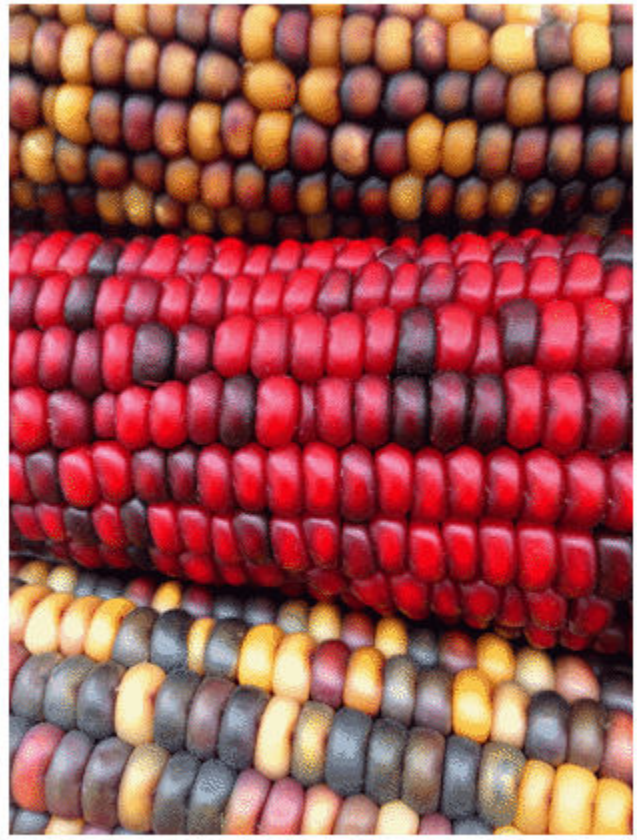
Display an image stored in a file.

```
imshow('peppers.png');
```

Change Colormap of Displayed Image

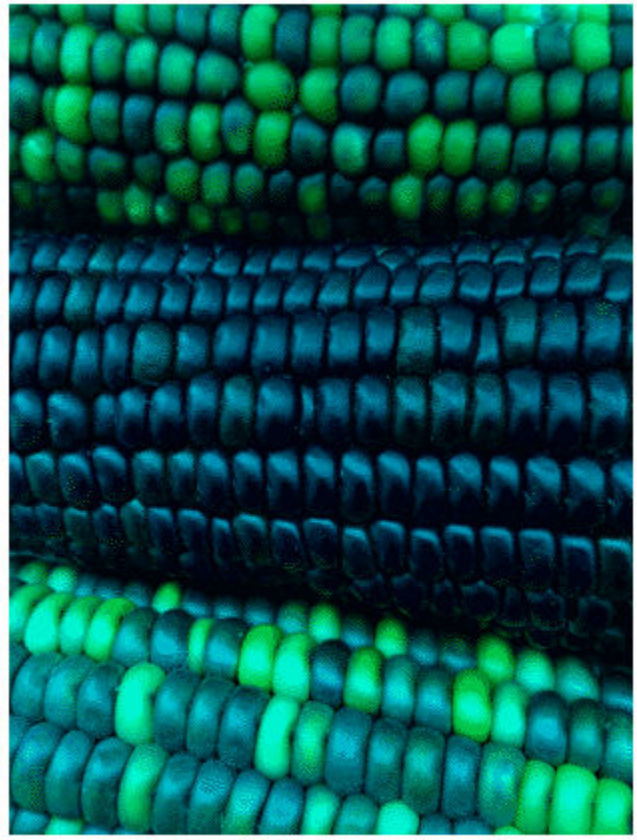
Read a sample indexed image, `corn.tif`, into the workspace, and then display it.

```
[X,map] = imread('corn.tif');
imshow(X,map)
```



Change the colormap for the image using the `colormap` function and specifying the target axes as the first input argument. Use the original colormap without the red component.

```
newmap = map;
newmap(:,1) = 0;
colormap(gca,newmap)
```



Input Arguments

`I` — Input grayscale image
matrix

Input grayscale image, specified as a matrix. A grayscale image can be any numeric data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`RGB` — Input truecolor image
m-by-*n*-by-3 array

Input truecolor image, specified as an *m*-by-*n*-by-3 array.

If you specify a truecolor image of data type `single` or `double`, then values should be in the range [0, 1]. If pixel values are outside this range, then you can use the `rescale` function to scale pixel values to the range [0, 1]. The `'DisplayRange'` argument has no effect when the input image is truecolor.

Data Types: `single` | `double` | `uint8` | `uint16`

`BW` — Input binary image
matrix

Input binary image, specified as a matrix.

Data Types: `logical`

`X` — Indexed image
2-D array of real numeric values

Indexed image, specified as a 2-D array of real numeric values. The values in `X` are indices into the colormap specified by `map`.

Data Types: `single` | `double` | `uint8` | `logical`

`map` — Colormap
c-by-3 array

Colormap, specified as an *c*-by-3 array of type `single` or `double` in the range `[0 1]`, or a *c*-by-3 array of type `uint8`. Each row specifies an RGB color value.

Data Types: `single` | `double` | `uint8`

`filename` — File name
character vector

File name, specified as a character vector. The image must be readable by `imread`. The `imshow` function displays the image, but does not store the image data in the MATLAB workspace. If the file contains multiple images, `imshow` displays the first image in the file.

Example: `imshow('peppers.png')`

Data Types: `char`

`[low high]` — Grayscale image display range
two-element vector

Grayscale image display range, specified as a two-element vector. For more information, see the `'DisplayRange'` name-value pair argument.

Example: `[50 250]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `imshow('board.tif','Border','tight')`

`Border` — Figure window border space
`'loose'` (default) | `'tight'`

Figure window border space, specified as the comma-separated pair consisting of `'Border'` and either `'tight'` or `'loose'`. When set to `'loose'`, the figure window includes space around the image in the figure. When set to `'tight'`, the figure window does not include any space around the image in the figure.

If the image is very small or if the figure contains other objects besides an image and its axes, `imshow` might use a border regardless of how this parameter is set.

Example: `imshow('board.tif','Border','tight')`

Data Types: `char`

`Colormap` — Colormap
c-by-3 matrix

Colormap, specified as the comma-separated pair consisting of `'Colormap'` and a *c*-by-3 matrix. `imshow` uses this to set the colormap for the axes. Use this parameter to view grayscale images in false color. If you specify an empty colormap (`[]`), then `imshow` ignores this parameter.

Note Starting in R2016b, `imshow` changes the colormap for the axes that contains the image instead of the figure.

Example: `newmap = copper; imshow('board.tif','Colormap',newmap)`

Data Types: `double`

`DisplayRange` — Grayscale image display range
two-element vector | `[]`

Display range of a grayscale image, specified as a two-element vector of the form `[low high]`. The `imshow` function displays the value `low` (and any value less than `low`) as black, and it displays the value `high` (and any value greater than `high`) as white. Values between `low` and `high` are displayed as intermediate shades of gray, using the default number of gray levels.

If you specify an empty matrix (`[]`), then `imshow` uses a display range of `[min(I(:)) max(I(:))]`. In other words, the minimum value in `I` is black, and the maximum value is white.

If you do not specify a display range, then `imshow` selects a default display range based on the image data type.

- If `I` is an integer type, then `DisplayRange` defaults to the minimum and maximum representable values for that integer class. For example, the default display range for `uint16` arrays is [0, 65535].
- If `I` is data type `single` or `double`, then the default display range is [0, 1].

Note Including the parameter name is optional, except when the image is specified by a file name. The syntax `imshow(I,[low high])` is equivalent to `imshow(I,'DisplayRange',[low high])`. If you call `imshow` with a file name, then you must specify the `'DisplayRange'` parameter.

Example: `h = imshow(I,'DisplayRange',[0 80]);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`InitialMagnification` — Initial magnification of image display
`100` (default) | numeric scalar | `'fit'`

Initial magnification of image display, specified as the comma-separated pair consisting of `'InitialMagnification'` and a numeric scalar or `'fit'`. If set to `100`, then `imshow` displays the image at 100% magnification (one screen pixel for each image pixel). If set to `'fit'`, then `imshow` scales the entire image to fit in the window.

Initially, `imshow` attempts to display the entire image at the specified magnification. If the magnification value is so large that the image is too big to display on the screen, `imshow` displays the image at the largest magnification that fits on the screen.

If the image is displayed in a figure with its `'WindowStyle'` property set to `'docked'`, then `imshow` displays the image at the largest magnification that fits in the figure.

Note: If you specify the axes position (using `subplot` or `axes`), `imshow` ignores any initial magnification you might have specified and defaults to the `'fit'` behavior.

When you use `imshow` with the `'Reduce'` parameter, the initial magnification must be `'fit'`.

In MATLAB Online, `'InitialMagnification'` is set to `'fit'` and cannot be changed.

Example: `h = imshow(I,'InitialMagnification','fit');`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char`

`Interpolation` — Interpolation technique
`'nearest'` (default) | `'bilinear'`

Interpolation technique used when scaling an image, specified as the comma-separated pair consisting of `'Interpolation'` and one of the following values.

Value	Description
`'nearest'`	Nearest neighbor interpolation (default)
`'bilinear'`	Bilinear interpolation

`Parent` — Parent axes of image object
`Axes` object

Parent axes of image object, specified as the comma-separated pair consisting of `'Parent'` and an `Axes` object. Use the `'Parent'` name-value argument to build a UI that gives you control of the `Figure` and `Axes` properties. Do not specify a `UIAxes` as the parent axes.

`Reduce` — Indicator for subsampling
`true` | `false` | `1` | `0`

Indicator for subsampling image, specified as the comma-separated pair consisting of `'Reduce'` and either `true`, `false`, `1`, or `0`. This argument is valid only when you use it with the name of a TIFF file. Use the `Reduce` argument to display overviews of very large images.

Data Types: `logical`

`XData` — X-axis limits of nondefault coordinate system
two-element vector

X-axis limits of nondefault coordinate system, specified as the comma-separated pair consisting of `'XData'` and a two-element vector. This argument establishes a nondefault spatial coordinate system by specifying the image `XData`. The value can have more than two elements, but `imshow` uses only the first and last elements.

Example: `'XData',[100 200]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`YData` — Y-axis limits of nondefault coordinate system
two-element vector

Y-axis limits of nondefault coordinate system, specified as the comma-separated pair consisting of `'YData'` and a two-element vector. The value can have more than two elements, but `imshow` uses only the first and last elements.

Example: `'YData',[100 200]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

himage — Image created by imshow
image object

Image created by `imshow`, specified as an image object.

Tips

- To change the colormap after you create the image, use the `colormap` command. Specify the axes that contains the image as the first input argument and the colormap you want as the second input argument. For an example, see “Change Colormap of Displayed Image” on page 1-7594.
- You can display multiple images with different colormaps in the same figure using `imshow` with the `subplot` function.
- If you have Image Processing Toolbox, then you can use the Image Viewer app as an integrated environment for displaying images and performing common image processing tasks.
- If you have Image Processing Toolbox, then you can use the `iptsetpref` function to set toolbox preferences that modify the behavior of `imshow`.
- The `imshow` function is not supported when you start MATLAB with the `-nojvm` option.
- Do not

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`image` | `imagesc` | `imfinfo` | `imread` | `imwrite` | `iptsetpref`

Topics

“Image Types”

Introduced before R2006a

imtile

Combine multiple image frames into one rectangular tiled image

Syntax

```
out = imtile(filenames)
out = imtile(I)
out = imtile(images)
out = imtile(imds)
out = imtile(X,map)
out = imtile( ___ ,Name,Value)
```

Description

`out = imtile(filenames)` returns a tiled image containing the images specified in `filenames`. `filenames` is an *n*-by-1 or 1-by-*n* string array, character vector, or cell array of character vectors. If the files are not in the current folder or in a folder on the MATLAB path, specify the full path name. See the `imread` command for more information.

By default, `imtile` arranges the images so that they roughly form a square, but you can change that using optional parameters. The images can have different sizes and types.

- If you specify an indexed image, then `imtile` converts it to RGB using the colormap present in the file.
- If there is a data type mismatch between images, then `imtile` rescales all images to be `double` using the `im2double` function.

`out = imtile(I)` returns a tiled image containing all the frames of the multiframe image array `I`. A multiframe image array can be a sequence of binary, grayscale, or truecolor images.

`out = imtile(images)` returns a tiled image containing the images specified in the cell array `images`. `imtile` displays empty cell array elements as a blank tile.

`out = imtile(imds)` returns a tiled image containing the images specified in the ImageDatastore object `imds`. For information about image datastores, see `ImageDatastore`.

`out = imtile(X,map)` treats all grayscale images in `X` as indexed images and applies the specified colormap `map` to all frames. `X` can be an array of grayscale images (*m*-by-*n*-by-1-by-*k*), a string array of file names, or a cell array of character vectors. If `X` represents file names, `map` overrides any internal colormap present in the image files.

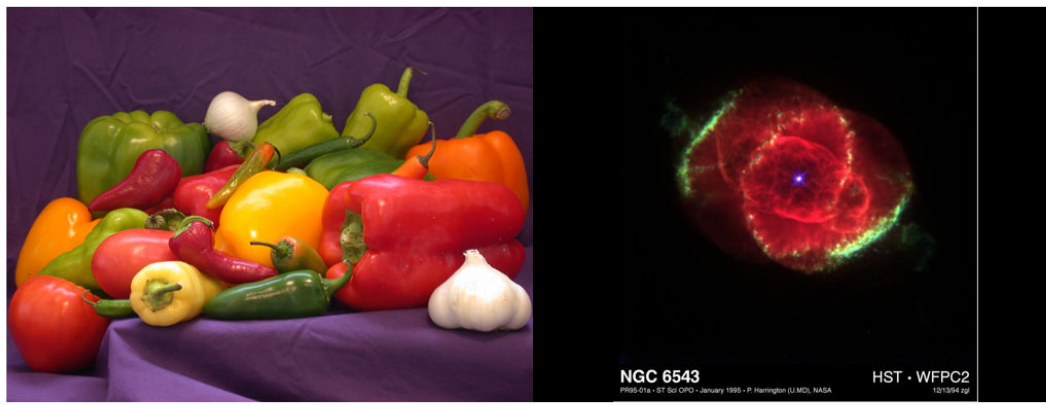
`out = imtile( ___ ,Name,Value)` returns a customized tiled image, depending on the values of the optional parameter name-value pairs. You can abbreviate parameter names, and case does not matter.

Examples

Create Tiled Image from Files

Read multiple images from files into the workspace and create a tiled image containing the images. Display the tiled image.

```
out = imtile({'peppers.png', 'ngc6543a.jpg'});
imshow(out);
```

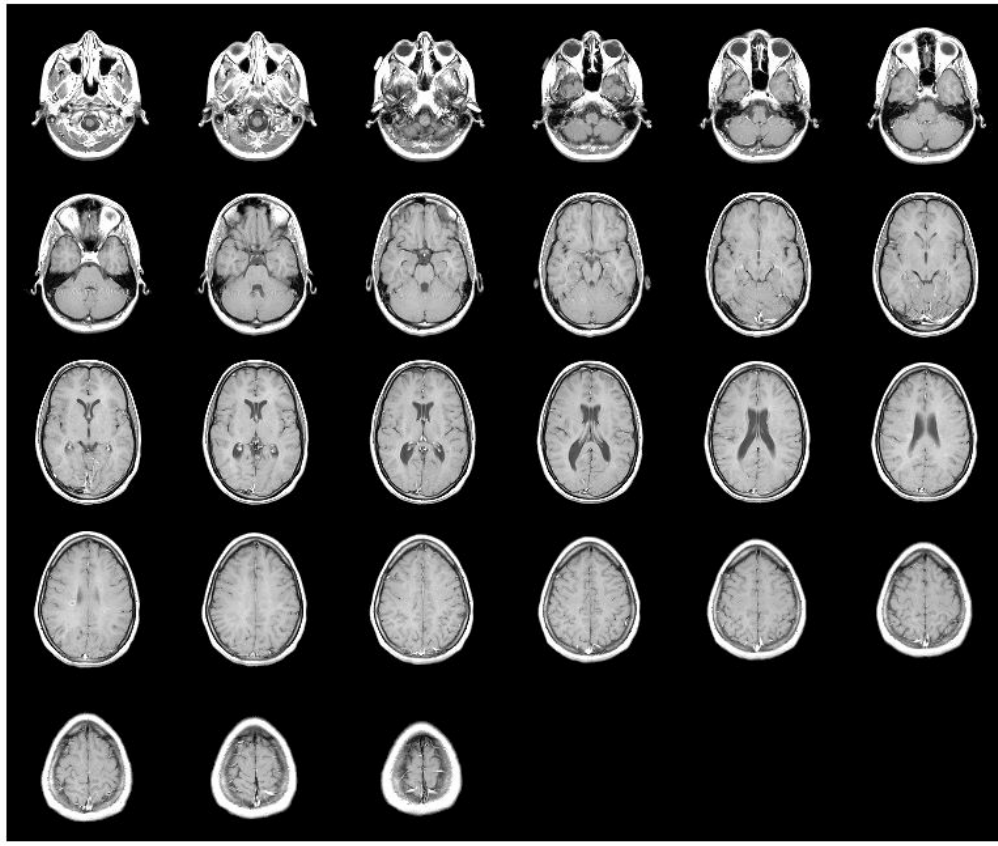


Customize Number of Images in Tiled Image

Using a data set containing multiple images, tile the images in a grid.

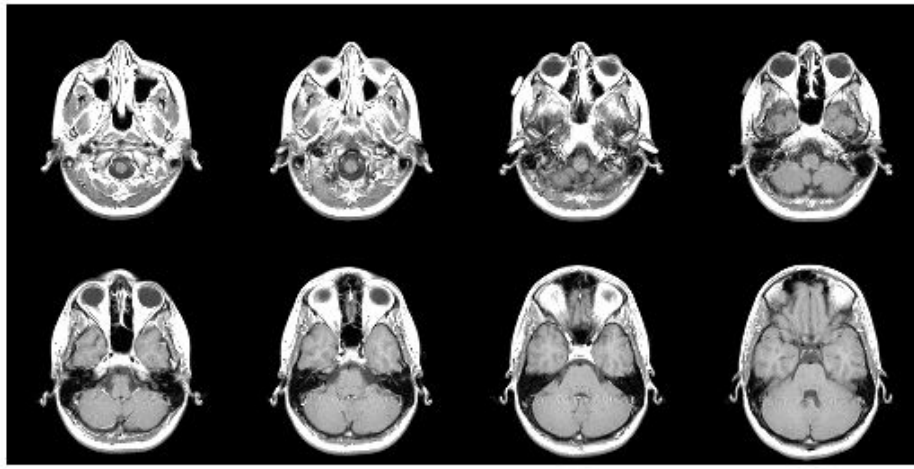
Load the MRI data set.

```
load mri
out = imtile(D, map);
imshow(out);
```



Create a tiled image containing only the first eight images in the data set. Use the `'GridSize'` parameter to arrange the images in a 2-by-4 grid.

```
out = imtile(D, map, 'Frames', 1:8, 'GridSize', [2 4]);
figure;
imshow(out);
```



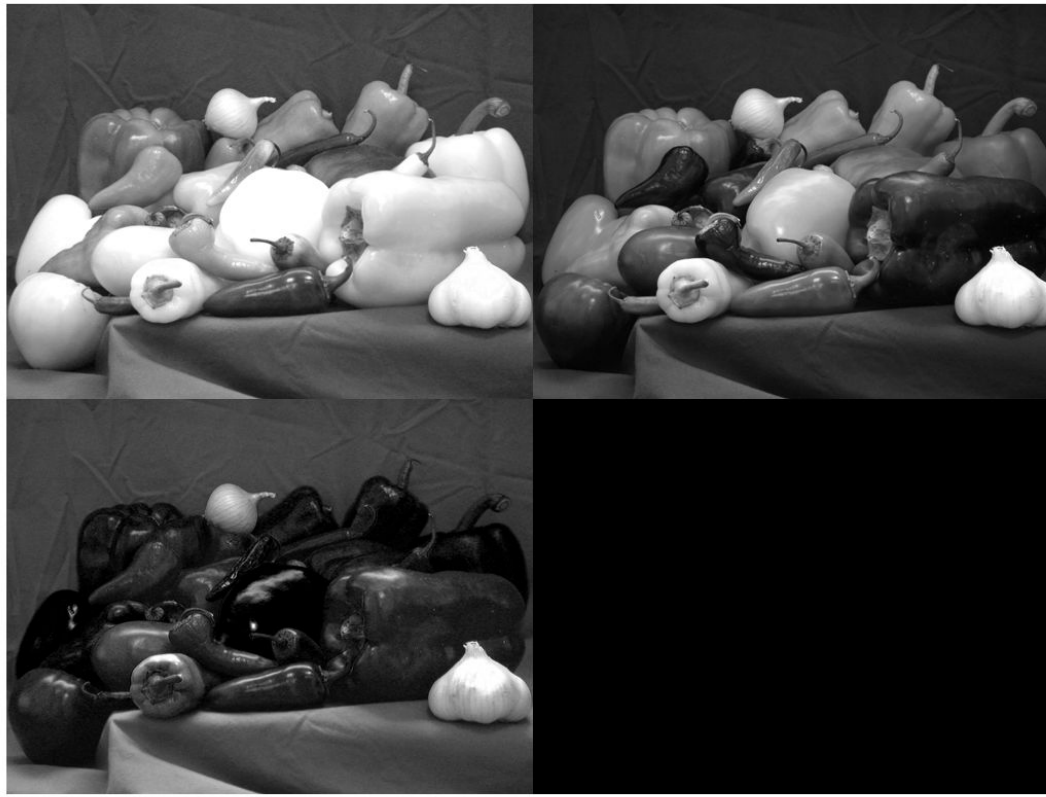
Inspect Color Planes of RGB Image

Read an RGB image into the workspace.

```
imRGB = imread('peppers.png');
```

Create a tiled image containing each of the three planes of the RGB image. Display the tiled image.

```
out = imtile(imRGB);
imshow(out)
```



Create Tiled Image from Image Datastore

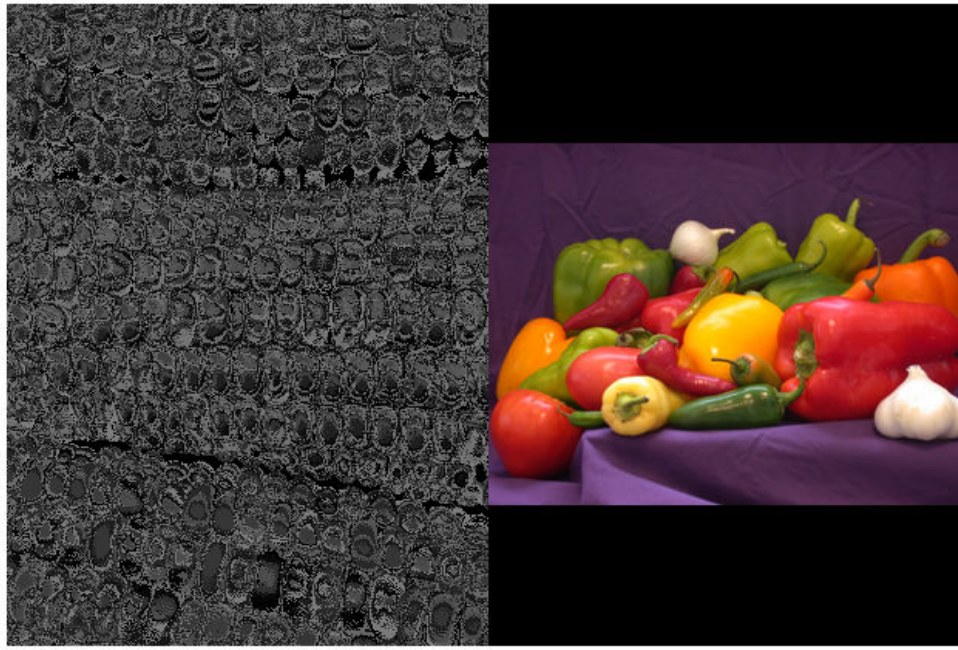
From an image datastore, create and customize a tiled image.

Create an image datastore containing all the files with the file extension `'tif'` or `'png'` in the specified folder.

```
fileFolder = fullfile(matlabroot,'toolbox','matlab','imagesci');
imds = imageDatastore(fileFolder,'FileExtensions',{'.tif','.png'});
```

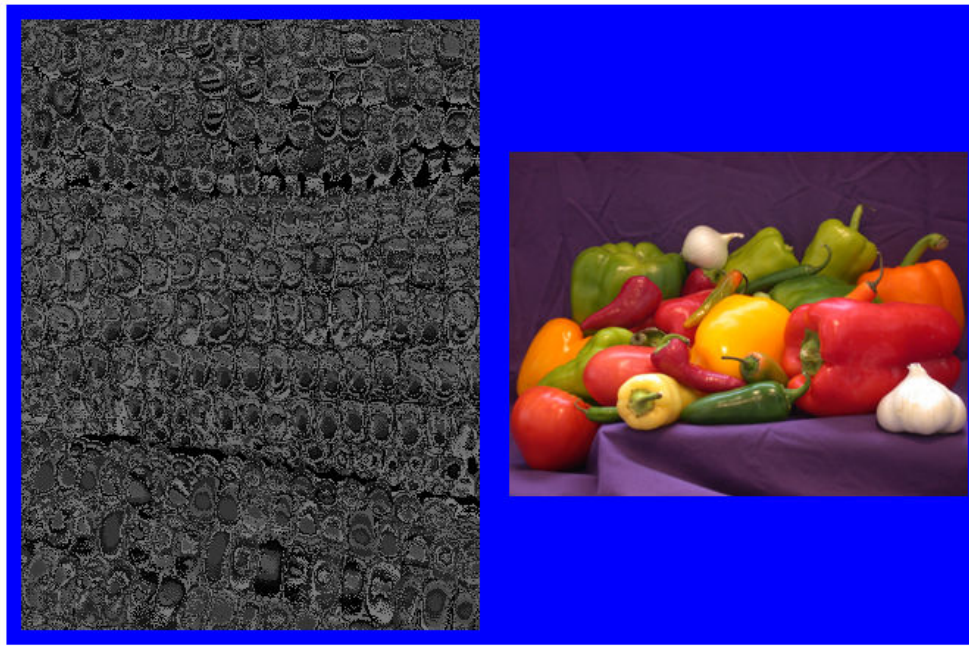
Create a tiled image containing the images in the datastore.

```
out1 = imtile(imds);
imshow(out1);
```



Use the `'BorderSize'` and `'BackgroundColor'` parameters to add a blue border to the tiled image.

```
out2 = imtile(imds, 'BorderSize', 10, 'BackgroundColor', 'b');
figure;
imshow(out2);
```



Input Arguments

`filenames` — Names of files containing images
n-by-1 or 1-by-*n* string array | character vector | cell array of character vectors

Names of files containing images, specified as an *n*-by-1 or 1-by-*n* string array, character vector, or cell array of character vectors.

`I` — Multiframe image
sequence of binary, grayscale, or truecolor images

Multiframe image, specified as a sequence of binary, grayscale, or truecolor images. `I` can be an *m*-by-*n*-by-*k* or an *m*-by-*n*-by-1-by-*k* array, or a sequence of binary, grayscale, or

truecolor images. A binary or grayscale image sequence must be an *m*-by-*n*-by-*k* or an *m*-by-*n*-by-1-by-*k* array. A truecolor image sequence must be an *m*-by-*n*-by-3-by-*k* array.

`images` — List of images
n-by-1 or 1-by-n cell array

List of images, specified as an *n*-by-1 or 1-by-*n* cell array of numeric matrices. The cell array can contain numeric matrices of size *m*-by-*n* or *m*-by-*n*-by-3.

`imds` — Image data store
ImageDatastore object

Image data store, specified as an ImageDatastore object.

`X,map` — Indexed image and associated colormap
2-D numeric array

Indexed image and associated colormap, specified as a 2-D numeric array (image) and an *n*-by-3 numeric array of class `double` (colormap). *n* is the number of colors in the colormap.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `tiled_image = imtile({'peppers.png', 'ngc6543a.jpg'},'BackgroundColor','g');`

`BackgroundColor` — Color of background
`'black'` (default) | MATLABColorSpec

Color of the background, specified as the comma-separated pair consisting of `'BackgroundColor'` and MATLAB `ColorSpec (Color Specification)`. The `imtile` function fills all blank spaces with this color, including the space specified by `BorderSize`. If you specify a background color, the `imtile` function renders the output as an RGB image.

Example: `'green'`

Example: `'g'`

Example: `[0 1 0]`

`BorderSize` — Padding around each thumbnail image
`[0 0]` (default) | numeric scalar or 1-by-2 vector

Padding around each thumbnail image, specified as the comma-separated pair consisting of `'BorderSize'` and a numeric scalar or 1-by-2 vector of the form `[brows bcols]`. `imtile` pads the borders with the background color.

`Frames` — Frames to include
total number of images (default) | numeric array | logical mask

Frames to include, specified as the comma-separated pair consisting of `'Frames'` and a numeric array or a logical mask. The `imtile` function interprets the values as indices into the image array or cell array. The following examples create a tiled image containing the first three image frames.

Example: `out = imtile(I,'Frames',1:3);`

Example: `out = imtile(I,'Frames',[true true true]);`

`GridSize` — Number of rows and columns of thumbnails
grid of images form a square (default) | two-element vector

Number of rows and columns of thumbnails in tiled image, specified as the comma-separated pair consisting of `'GridSize'` and a two-element vector of the form `[nrows ncols]`. `nrows` specifies the number of rows in the grid and `ncols` specifies the number of columns in the grid. Use `NaN`s or `Inf`s to have `imtile` calculate the size in a particular dimension in a way that includes all the images.

- If `'GridSize'` is `[2 NaN]`, then `imtile` creates a tiled image with two rows and the number of columns necessary to include all the images.
- If both the elements are `NaN` or `Inf`, then `imtile` calculates the grid size to form a square. `imtile` returns the images horizontally across columns.
- If there is a mismatch between `GridSize` and number of images (frames), `imtile` creates the tiled image based on `GridSize`.

`ThumbnailSize` — Size of thumbnails
full size of first image (default) | two-element vector

Size of thumbnails, specified as the comma-separated pair consisting of `'ThumbnailSize'` and a two-element vector of the form `[trows tcols]`, in pixels. The

`imtile` function preserves the aspect ratio of the original image by zero-padding the boundary.

- If you specify a `NaN` or `Inf`, then the `imtile` function calculates the corresponding value automatically to preserve the aspect ratio of the first image.
- If you specify an empty array (`[]`), then the `imtile` function uses the full size of the first image as the thumbnail size.

Output Arguments

`out` — Tiled output image
numeric array

Tiled output image, returned as a numeric array.

See Also

`imread` | `imshow`

Introduced in R2018b

imwrite

Write image to graphics file

Syntax

```
imwrite(A,filename)
imwrite(A,map,filename)
```

```
imwrite( ___ ,fmt)
```

```
imwrite( ___ ,Name,Value)
```

Description

`imwrite(A,filename)` writes image data `A` to the file specified by `filename`, inferring the file format from the extension. `imwrite` creates the new file in your current folder. The bit depth of the output image depends on the data type of `A` and the file format. For most formats:

- If `A` is of data type `uint8`, then `imwrite` outputs 8-bit values.
- If `A` is of data type `uint16` and the output file format supports 16-bit data (JPEG, PNG, and TIFF), then `imwrite` outputs 16-bit values. If the output file format does not support 16-bit data, then `imwrite` returns an error.
- If `A` is a grayscale or RGB color image of data type `double` or `single`, then `imwrite` assumes that the dynamic range is [0,1] and automatically scales the data by 255 before writing it to the file as 8-bit values. If the data in `A` is `single`, convert `A` to `double` before writing to a GIF or TIFF file.
- If `A` is of data type `logical`, then `imwrite` assumes that the data is a binary image and writes it to the file with a bit depth of 1, if the format allows it. BMP, PNG, or TIFF formats accept binary images as input arrays.

If `A` contains indexed image data, you should additionally specify the `map` input argument.

`imwrite(A,map,filename)` writes the indexed image in `A` and its associated colormap, `map`, to the file specified by `filename`.

- If A is an indexed image of data type double or single, then imwrite converts the indices to zero-based indices by subtracting 1 from each element, and then writes the data as uint8. If the data in A is single, convert A to double before writing to a GIF or TIFF file.

imwrite(___ ,fmt) writes the image in the format specified by fmt, regardless of the file extension in filename. You can specify fmt after the input arguments in any of the previous syntaxes.

imwrite(___ ,Name,Value) specifies additional parameters for output GIF, HDF, JPEG, PBM, PGM, PNG, PPM, and TIFF files, using one or more name-value pair arguments. You can specify Name,Value after the input arguments in any of the previous syntaxes.

Examples

Write Grayscale Image to PNG

Write a 50-by-50 array of grayscale values to a PNG file in the current folder.

```
A = rand(50);
imwrite(A,'myGray.png')
```

Write Indexed Image Data to PNG

Write an indexed image array and its associated colormap to a PNG file.

Load sample image data from the file, clown.mat.

```
load clown.mat
```

The image array X and its associated colormap, map, are loaded into the MATLAB workspace.

Write the data to a new PNG file.

```
imwrite(X,map,'myclown.png')
```

imwrite creates the file, myclown.png, in your current folder.

View the new file by opening it outside of MATLAB.

Write Indexed Image with MATLAB Colormap

Write image data to a new PNG file with the built-in MATLAB colormap, `copper`.

Load sample image data from the file `clown.mat`.

```
load clown.mat
```

The image array `X` and its associated colormap, `map`, are loaded into the MATLAB workspace. `map` is a matrix of 81 RGB vectors.

Define a copper-tone colormap with 81 RGB vectors. Then, write the image data to a PNG file using the new colormap.

```
newmap = copper(81);
imwrite(X,newmap,'copperclown.png');
```

`imwrite` creates the file, `copperclown.png`, in your current folder.

View the new file by opening it outside of MATLAB.

Write Truecolor Image to JPEG

Create and write truecolor image data to a JPEG file.

Create a 49-by-49-by-3 array of random RGB values.

```
A = rand(49,49);
A(:,:,2) = rand(49,49);
A(:,:,3) = rand(49,49);
```

Write the image data to a JPEG file, specifying the output format using `'jpg'`. Add a comment to the file using the `'Comment'` name-value pair argument.

```
imwrite(A,'newImage.jpg','jpg','Comment','My JPEG file')
```

View information about the new file.

```
imfinfo('newImage.jpg')
```

```
ans =

           Filename: 'S:\newImage.jpg'
```

```
        FileModDate: '25-Jan-2013 16:18:41'
           FileSize: 2339
             Format: 'jpg'
      FormatVersion: ''
              Width: 49
             Height: 49
           BitDepth: 24
          ColorType: 'truecolor'
    FormatSignature: ''
    NumberOfSamples: 3
       CodingMethod: 'Huffman'
      CodingProcess: 'Sequential'
            Comment: {'My JPEG file'}
```

Write Multiple Images to TIFF File

Write multiple images to a single multipage TIFF file.

Create two sets of random image data, `im1` and `im2`.

```
im1 = rand(50,40,3);
im2 = rand(50,50,3);
```

Write the first image to a new TIFF file. Then, append the second image to the same file.

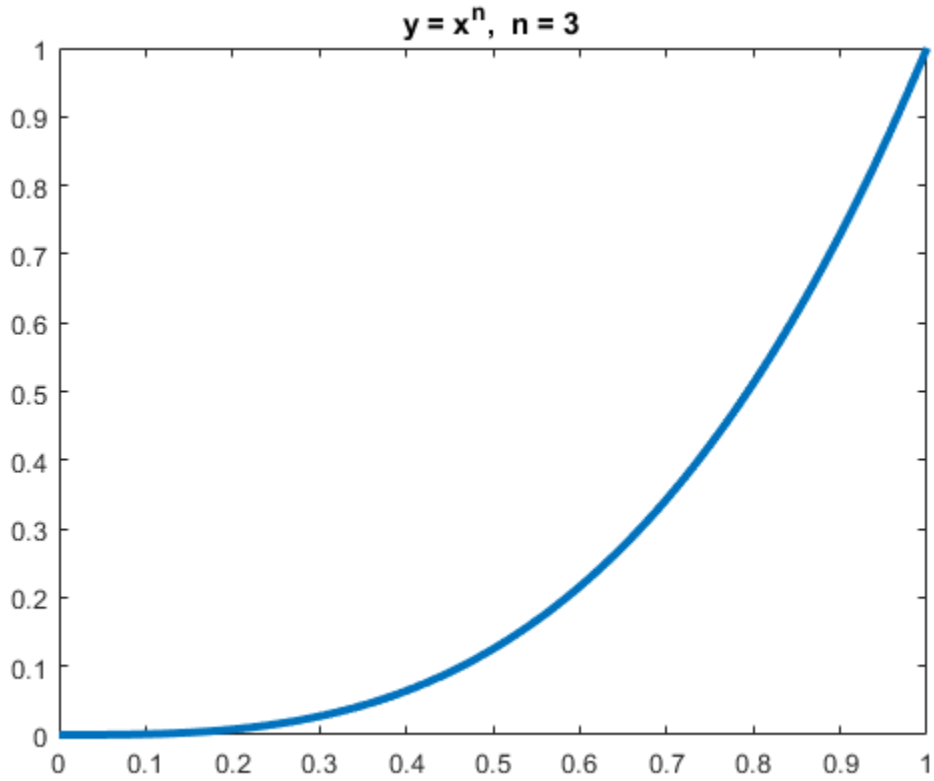
```
imwrite(im1,'myMultipageFile.tif')
imwrite(im2,'myMultipageFile.tif','WriteMode','append')
```

Write Animated GIF

Draw a series of plots, capture them as images, and write them into one animated GIF file.

Plot $y = x^n$ for $n = 3$.

```
x = 0:0.01:1;
n = 3;
y = x.^n;
plot(x,y,'LineWidth',3)
title(['y = x^n,  n = ' num2str(n) ])
```



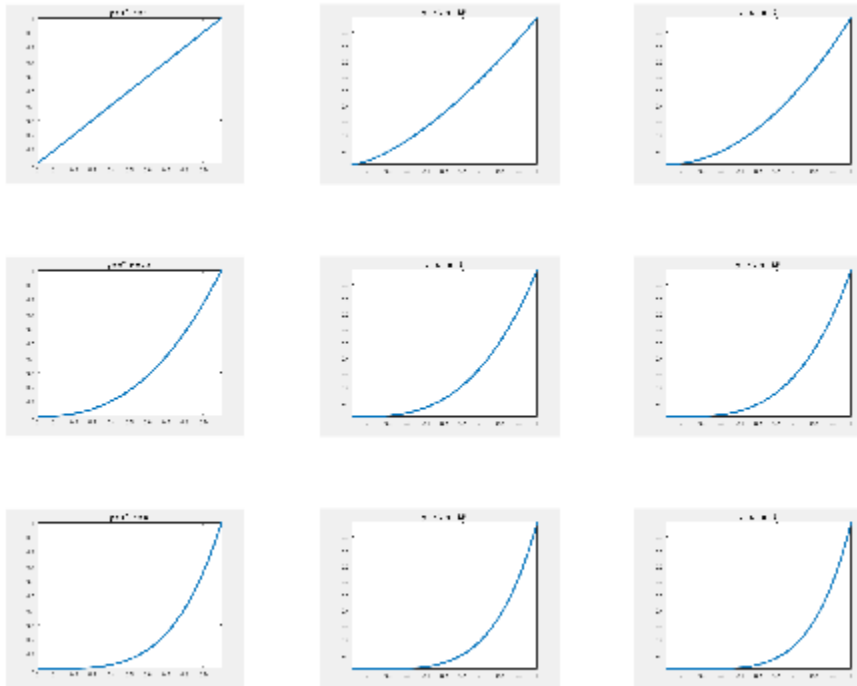
Capture a series of plots for increasing values of n.

```
n = 1:0.5:5;
nImages = length(n);

fig = figure;
for idx = 1:nImages
    y = x.^n(idx);
    plot(x,y,'LineWidth',3)
    title(['y = x^n,  n = ' num2str( n(idx)) ])
    drawnow
    frame = getframe(fig);
    im{idx} = frame2im(frame);
```

```
end
close;
```

Display the series of images in one figure.

```
figure;
for idx = 1:nImages
    subplot(3,3,idx)
    imshow(im{idx});
end
```



Save the nine images into a GIF file. Because three-dimensional data is not supported for GIF files, call `rgb2ind` to convert the RGB data in the image to an indexed image `A` with a

colormap `map`. To append multiple images to the first image, call `imwrite` with the name-value pair argument `'WriteMode','append'`.

```
filename = 'testAnimated.gif'; % Specify the output file name
for idx = 1:nImages
    [A,map] = rgb2ind(im{idx},256);
    if idx == 1
        imwrite(A,map,filename,'gif','LoopCount',Inf,'DelayTime',1);
    else
        imwrite(A,map,filename,'gif','WriteMode','append','DelayTime',1);
    end
end
```

`imwrite` writes the GIF file to your current folder. Name-value pair `'LoopCount',Inf` causes the animation to continuously loop. `'DelayTime',1` specifies a 1-second delay between the display of each image in the animation.

Input Arguments

`A` — Image data
matrix

Image data, specified as a full (nonsparse) matrix.

- For grayscale images, `A` can be `m`-by-`n`.
- For indexed images, `A` can be `m`-by-`n`. Specify the associated colormap in the `map` input argument.
- For truecolor images, `A` must be `m`-by-`n`-by-3. `imwrite` does not support writing RGB images to GIF files.

For TIFF files, `A` can be an `m`-by-`n`-by-4 array containing color data that uses the CMYK color space.

For multiframe GIF files, `A` can be an `m`-by-`n`-by-1-by-`p` array containing grayscale or indexed images, where `p` is the number of frames to write. RGB images are not supported in this case.

Data Types: `double` | `single` | `uint8` | `uint16` | `logical`

`filename` — Name of output file
character vector | string scalar

Name of the output file, specified as a character vector or string scalar.

Depending on the location you are writing to, `filename` can take on one of these forms.

Location	Form
Current folder	To write to the current folder, specify the name of the file in `filename`. `filename` must include the file extension. For a list of the image types that `imwrite` can write, see the description for the `fmt` input argument. **Example:** `'myImage.jpg'`
Other folders	To write to a folder different from the current folder, specify the full or relative path name in `filename`. **Example:** `'C:\myFolder\myImage.ext'` **Example:** `'\imgDir\myImage.ext'`
Remote Location	To write to a remote location, `filename` must contain the full path of the file specified as a uniform resource locator (URL) of the form: *`scheme_name`*`://path_to_file/my_file.ext` Based on your remote location, *`scheme_name`* can be one of the values in this table. **Remote Location** / **`scheme_name`**: Amazon S3 — `s3` Windows Azure Blob Storage — `wasb`, `wasbs` HDFS — `hdfs` For more information, see “Work with Remote Data”. **Example:** `'s3://bucketname/path_to_file/my_image.jpg'`

Data Types: `char` | `string`

map — Colormap of indexed image
m-by-3 array

Colormap associated with indexed image data in A, specified as an m-by-3 array. map must be a valid MATLAB colormap. See colormap for a list of built-in MATLAB colormaps. Most image file formats do not support colormaps with more than 256 entries.

Example: [0,0,0;0.5,0.5,0.5;1,1,1]

Example: jet(60)

Data Types: double

fmt — Format of output file
'bmp' | 'gif' | 'hdf' | 'jpg' | 'jp2' | ...

Format of the output file, specified as one of the formats in this table.

This table also summarizes the types of images that imwrite can write. The MATLAB file format registry determines which file formats are supported. See imformats for more information about this registry.

For certain formats, imwrite can accept additional name-value pair arguments. To view these arguments, click the linked format names below.

Value of fmt	Format of Output File	Description
'bmp'	Windows Bitmap (BMP)	1-bit, 8-bit, and 24-bit uncompressed images
'gif'	“GIF — Graphics Interchange Format” on page 1-0	8-bit images
'hdf'	“HDF4 — Hierarchical Data Format” on page 1-0	8-bit raster image data sets with or without associated colormap, 24-bit raster image data sets

Value of `fmt`	Format of Output File	Description
`'jpg'` or `'jpeg'`	“JPEG — Joint Photographic Experts Group” on page 1-0	8-bit, 12-bit, and 16-bit Baseline JPEG images **Note** `imwrite` converts indexed images to RGB before writing data to JPEG files, because the JPEG format does not support indexed images.
`'jp2'` or `'jpx'`	“JPEG 2000—Joint Photographic Experts Group 2000” on page 1-0	1-bit, 8-bit, and 16-bit JPEG 2000 images
`'pbm'`	Portable Bitmap (PBM) on page 1-0	Any 1-bit PBM image, ASCII (plain) or raw (binary) encoding
`'pcx'`	Windows Paintbrush (PCX)	8-bit images
`'pgm'`	Portable Graymap (PGM) on page 1-0	Any standard PGM image; ASCII (plain) encoded with arbitrary color depth; raw (binary) encoded with up to 16 bits per gray value
`'png'`	“PNG — Portable Network Graphics” on page 1-0	1-bit, 2-bit, 4-bit, 8-bit, and 16-bit grayscale images; 8-bit and 16-bit grayscale images with alpha channels; 1-bit, 2-bit, 4-bit, and 8-bit indexed images; 24-bit and 48-bit truecolor images; 24-bit and 48-bit truecolor images with alpha channels **Note** The `imwrite` function does not support writing of indexed PNG files that have insufficient colormap entries.
`'pnm'`	Portable Anymap (PNM) on page 1-0	Any of the PPM/PGM/PBM formats, chosen automatically

Value of `fmt`	Format of Output File	Description
`'ppm'`	Portable Pixmap (PPM) on page 1-0	Any standard PPM image: ASCII (plain) encoded with arbitrary color depth or raw (binary) encoded with up to 16 bits per color component
`'ras'`	Sun™ Raster (RAS) on page 1-0	Any RAS image, including 1-bit bitmap, 8-bit indexed, 24-bit truecolor, and 32-bit truecolor with alpha
`'tif'` or `'tiff'`	Tagged Image File Format (TIFF) on page 1-0	Baseline TIFF images, including: • 1-bit, 8-bit, 16-bit, 24-bit, and 48-bit uncompressed images and images with packbits, LZW, or Deflate compression • 1-bit images with CCITT 1D, Group 3, and Group 4 compression • CIELAB, ICCLAB, and CMYK images
`'xwd'`	X Windows Dump (XWD)	8-bit ZPixmaps

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `imwrite(A,'myFile.png','BitDepth',8)` writes the data in `A` using 8 bits to represent each pixel.

GIF — Graphics Interchange Format

`BackgroundColor` — Color to use as background color

scalar integer

Color to use as background color for the indexed image, specified as the comma-separated pair consisting of `'BackgroundColor'` and a scalar integer corresponding to the colormap index.

The background color is used for some disposal methods in animated GIFs.

- If image data `A` is `uint8` or `logical`, then the colormap index is zero-based.
- If image data `A` is `double`, then the colormap index is one-based.

The default background color corresponds to the first color in the colormap.

Example: `'BackgroundColor',15`

`Comment` — Comment to add to image
character vector | string scalar | cell array of character vectors | string array

Comment to add to the image, specified as the comma-separated pair consisting of `'Comment'` and a character vector, string scalar, a 1-by-`n` cell array of character vectors, or a string array. For a cell array of character vectors, `imwrite` adds a carriage return after each character vector.

Example: `'Comment',{'Sample #314','January 5, 2013'}`

Data Types: `char` | `cell` | `string`

`DelayTime` — Delay before displaying next image
0.5 (default) | scalar value in the range [0,655]

Delay before displaying next image, in seconds, specified as the comma-separated pair consisting of `'DelayTime'` and a scalar value in the range [0,655]. A value of `0` displays images as fast as your hardware allows.

Example: `'DelayTime',60`

`DisposalMethod` — Disposal method of animated GIF
`'doNotSpecify'` (default) | `'leaveInPlace'` | `'restoreBG'` | `'restorePrevious'`

Disposal method of an animated GIF, specified as the comma-separated pair consisting of `'DisposalMethod'` and one of the methods in this table.

Value of `DisposalMethod`	Result
`'doNotSpecify'` (default)	Replace one full-size, nontransparent frame with another.
`'leaveInPlace'`	Any pixels not covered up by the next frame continue to display.
`'restoreBG'`	The background color or background tile shows through transparent pixels.

Value of DisposalMethod	Result
'restorePrevious'	Restore to the state of a previous, undisposed frame.

Example: 'DisposalMethod','restoreBG'

Location — Offset of screen relative to image
[0,0] (default) | two-element vector

Offset of the screen relative to the image, measured from the top left corner of each, specified as the comma-separated pair consisting of 'Location' and a two-element vector. The first vector element specifies the offset from the top, and the second element specifies the offset from the left, in pixels.

Example: 'Location',[10,15]

Data Types: double

LoopCount — Number of times to repeat animation
Inf (default) | integer in the range [0,65535]

Number of times to repeat the animation, specified as the comma-separated pair consisting of 'LoopCount' and either an integer in the range [0,65535], or the value Inf. If you specify 0, the animation plays once. If you specify the value 1, the animation plays twice, and so on. A LoopCount value of Inf causes the animation to continuously loop.

To enable animation within Microsoft PowerPoint®, specify a value for 'LoopCount' within the range [1,65535]. Some Microsoft applications interpret the value 0 to mean do not loop at all.

Example: 'LoopCount',3

ScreenSize — Height and width of frame
height and width of input image (default) | two-element vector

Height and width of the frame, specified as the comma-separated pair consisting of 'ScreenSize' and a two-element vector. When you use the ScreenSize argument with 'Location', it provides a way to write frames to the image that are smaller than the whole frame. 'DisposalMethod' determines the fill value for pixels outside the frame.

Example: 'ScreenSize',[1000 1060]

Data Types: double

`TransparentColor` — Color to use as transparent color

scalar integer

Color to use as transparent color for the image, specified as the comma-separated pair consisting of `'TransparentColor'` and a scalar integer corresponding to the colormap index.

- If image data `A` is `uint8` or `logical`, then indexing begins at `0`.
- If image data `A` is `double`, then indexing begins at `1`.

Example: `'TransparentColor',20`

`WriteMode` — Writing mode

`'overwrite'` (default) | `'append'`

Writing mode, specified as the comma-separated pair consisting of `'WriteMode'` and either `'overwrite'` or `'append'`. In `overwrite` mode, `imwrite` overwrites an existing file,`filename`. In `append` mode, `imwrite` adds a single frame to the existing file.

Example: `'WriteMode','append'`

HDF4 — Hierarchical Data Format

`Compression` — Compression scheme

`'none'` (default) | `'jpeg'` | `'rle'`

Compression scheme, specified as the comma-separated pair consisting of `'Compression'` and one of the options in this table.

Value of `Compression`	Result
`'none'` (default)	No compression
`'jpeg'`	JPEG compression. Valid only for grayscale and RGB images.
`'rle'`	Run-length encoding. Valid only for grayscale and indexed images.

Example: `'Compression','jpeg'`

`Quality` — Quality of JPEG-compressed file

75 (default) | scalar in the range [0,100]

Quality of the JPEG-compressed file, specified as the comma-separated pair consisting of `'Quality'` and a scalar in the range [0,100], where 0 is lower quality and higher compression, and 100 is higher quality and lower compression. This parameter applies only if `'Compression'` is `'jpeg'`.

Example: `'Quality',25`

`WriteMode` — Writing mode
`'overwrite'` (default) | `'append'`

Writing mode, specified as the comma-separated pair consisting of `'WriteMode'` and either `'overwrite'` or `'append'`. In `overwrite` mode, `imwrite` overwrites an existing file,`filename`. In `append` mode, `imwrite` adds a single frame to the existing file.

Example: `'WriteMode','append'`

JPEG — Joint Photographic Experts Group

`BitDepth` — Number of bits per pixel
8 (default) | scalar

Number of bits per pixel, specified as the comma-separated pair consisting of `'BitDepth'` and a scalar.

- For grayscale images, the `BitDepth` value can be 8, 12, or 16. The default value is 8. For 16-bit images, the `'Mode'` name-value pair argument must be `'lossless'`.
- For color images, the `BitDepth` value is the number of bits per plane, and can be 8 or 12. The default is 8 bits per plane.

Example: `'BitDepth',12`

`Comment` — Comment to add to image
character vector | character array | n-by-1 cell array of character vectors

Comment to add to the image, specified as the comma-separated pair consisting of `'Comment'` and a character vector, a string scalar, a character array, an n-by-1 cell array of character vectors, or a string array. `imwrite` writes each row of input as a comment in the JPEG file.

Example: `'Comment',{'First line';'second line';'third line'}`

Data Types: `char` | `string` | `cell`

`Mode` — Type of compression
`'lossy'` (default) | `'lossless'`

Type of compression, specified as the comma-separated pair consisting of `'Mode'` and one of these options:

- `'lossy'`
- `'lossless'`

Example: `'Mode','lossless'`

`Quality` — Quality of output file
75 (default) | scalar in the range [0,100]

Quality of the output file, specified as the comma-separated pair consisting of `'Quality'` and a scalar in the range [0,100], where 0 is lower quality and higher compression, and 100 is higher quality and lower compression. A `Quality` value of 100 does not write a lossless JPEG image. Instead, use the `'Mode','lossless'` name-value pair argument.

Example: `'Quality',100`

JPEG 2000— Joint Photographic Experts Group 2000

`Comment` — Comment to add to image
character vector | string scalar | character array | n-by-1 cell array of character vectors | string array

Comment to add to the image, specified as the comma-separated pair consisting of `'Comment'` and a character vector, a character array, string scalar, a cell array of character vectors, or string array. `imwrite` writes each row of input as a comment in the JPEG 2000 file.

Example: `'Comment',{'First line';'second line';'third line'}`

Example: `'Comment',{'First line','second line','third line'}`

Data Types: `cell` | `char` | `string`

`CompressionRatio` — Target compression ratio
1 (default) | scalar

Target compression ratio, specified as the comma-separated pair consisting of `'CompressionRatio'` and a real scalar greater than or equal to 1. The compression ratio is the ratio of the input image size to the output compressed size. For example, a

value of 2.0 implies that the output image size is half of the input image size or less. A higher value implies a smaller file size and reduced image quality. The compression ratio does not take into account the header size.

Specifying `CompressionRatio` is valid only when `'Mode'` is `'lossy'`.

Example: `'CompressionRatio',3`

`Mode` — Type of compression
`'lossy'` (default) | `'lossless'`

Type of compression, specified as the comma-separated pair consisting of `'Mode'` and one of these options:

- `'lossy'`
- `'lossless'`

Example: `'Mode','lossless'`

`ProgressionOrder` — Order of packets in code stream
`'LRCP'` (default) | `'RLCP'` | `'RPCL'` | `'PCRL'` | `'CPRL'`

Order of packets in the code stream, specified as the comma-separated pair consisting of `'ProgressionOrder'` and one of these options:

- `'LRCP'`
- `'RLCP'`
- `'RPCL'`
- `'PCRL'`
- `'CPRL'`

The characters represent the following: `L` = layer, `R` = resolution, `C` = component and `P` = position.

Example: `'ProgressionOrder','RLCP'`

`QualityLayers` — Number of quality layers
1 (default) | integer in the range [1,20]

Number of quality layers, specified as the comma-separated pair consisting of `'QualityLayers'` and an integer in the range [1,20].

Example: `'QualityLayers',8`

`ReductionLevels` — Number of reduction levels
4 (default) | integer in the range [1,8]

Number of reduction levels, or wavelet decomposition levels, specified as the comma-separated pair consisting of `'ReductionLevels'` and an integer in the range [1,8].

Example: `'ReductionLevels',6`

`TileSize` — Tile height and width
image size (default) | two-element vector

Tile height and width, specified as the comma-separated pair consisting of `'TileSize'` and a two-element vector. The minimum size you can specify is `[128 128]`.

Example: `'TileSize',[130 130]`

PBM-, PGM-, and PPM — Portable Bitmap, Graymap, Pixmap

`Encoding` — Encoding
`'rawbits'` (default) | `'ASCII'`

Encoding, specified as the comma-separated pair consisting of `'Encoding'` and either `'rawbits'` for binary encoding, or `'ASCII'` for plain encoding.

Example: `'Encoding','ASCII'`

`MaxValue` — Maximum gray or color value
scalar

Maximum gray or color value, specified as the comma-separated pair consisting of `'MaxValue'` and a scalar.

Available only for PGM and PPM files. For PBM files, this value is always 1.

If the image array is `uint16`, then the default value for `MaxValue` is `65535`. Otherwise, the default value is `255`.

Example: `'MaxValue',510`

PNG — Portable Network Graphics

`Alpha` — Transparency of each pixel
matrix of values in the range [0,1]

Transparency of each pixel, specified as the comma-separated pair consisting of `'Alpha'` and a matrix of values in the range [0,1]. The row and column dimensions of the `Alpha` matrix must be the same as those of the image data array. You can specify `Alpha` only for grayscale (`m`-by-`n`) and truecolor (`m`-by-`n`-by-3) image data.

Note You cannot specify both `'Alpha'` and `'Transparency'` at the same time.

Data Types: `double` | `uint8` | `uint16`

`Author` — Author information
character vector | string scalar

Author information, specified as the comma-separated pair consisting of `'Author'` and a character vector or string scalar.

Example: `"Author','Ann Smith'`

Data Types: `char`

`Background` — Background color when compositing transparent pixels
scalar in the range `[0,1]` | integer in the range `[1,P]` | 3-element vector in the range `[0,1]`

Background color when compositing transparent pixels, specified as the comma-separated pair consisting of `'Background'` and a value dependent on the image data, as follows.

Image Type	Form of `Background` Value
Grayscale images	Scalar in the range `[0,1]`.
Indexed images	Integer in the range `[1,P]`, where `P` is the colormap length. For example, `'Background',50` sets the background color to the color specified by the 50th index in the colormap.
Truecolor images	Three-element vector of RGB intensities in the range `[0,1]`. For example, `'Background',[0 1 1]` sets the background color to cyan.

Data Types: `double`

`BitDepth` — Number of bits per pixel
scalar

Number of bits per pixel, specified as the comma-separated pair consisting of `'BitDepth'` and a scalar. Depending on the output image, the scalar can be one of the following values.

Image Type	Allowed Values for `BitDepth`
Grayscale images	1, 2, 4, 8, or 16
Grayscale images with an alpha channel	8 or 16
Indexed images	1, 2, 4, or 8
Truecolor images	8 or 16

- If the image is of class `double` or `uint8`, then the default bit depth is 8 bits per pixel.
- If the image is `uint16`, then the default is 16 bits per pixel.
- If the image is `logical`, then the default is 1 bit per pixel.

Example: `'BitDepth',4`

`Chromaticities` — Reference white point and primary chromaticities
8-element vector

Reference white point and primary chromaticities, specified as the comma-separated pair consisting of `'Chromaticities'` and an 8-element vector, `[wx wy rx ry gx gy bx by]`. The elements `wx` and `wy` are the chromaticity coordinates of the white point, and the elements `rx`, `ry`, `gx`, `gy`, `bx`, and `by` are the chromaticity coordinates of the three primary colors.

If you specify `Chromaticities`, you should also specify the `Gamma` name-value pair argument.

Example: `'Chromaticities', [0.312,0.329,0.002,0.002,0.001,0.001,0.115,0.312]`

Data Types: `double`

`Comment` — Comment to add to image
character vector | string scalar

Comment to add to the image, specified as the comma-separated pair consisting of `'Comment'` and a character vector or string scalar.

`Copyright` — Copyright notice
character vector | string scalar

Copyright notice, specified as the comma-separated pair consisting of `'Copyright'` and a character vector or string scalar.

`CreationTime` — Time of original image creation
character vector | string scalar

Time of original image creation, specified as a character vector or string scalar.

`Description` — Description of image
character vector | string scalar

Description of the image, specified as the comma-separated pair consisting of `'Description'` and a character vector or string scalar.

`Disclaimer` — Legal disclaimer
character vector | string scalar

Legal disclaimer, specified as the comma-separated pair consisting of `'Disclaimer'` and a character vector or string scalar.

`Gamma` — File gamma
scalar

File gamma, specified as the comma-separated pair consisting of `'Gamma'` and a scalar.

Example: `'Gamma',2.2`

`ImageModTime` — Time of last image modification
serial date number | character vector or string scalar containing a date

Time of the last image modification, specified as the comma-separated pair consisting of `'ImageModTime'` and a MATLAB serial date number or a character vector or string scalar of a date that can be converted to a date vector using the `datevec` function. Values should be in Coordinated Universal Time (UTC).

The default `ImageModTime` value is the time when you call `imwrite`.

Example: `'ImageModTime','17-Jan-2013 11:23:10'`

Data Types: `double` | `char` | `string`

`InterlaceType` — Interlacing scheme
`'none'` (default) | `'adam7'`

Interlacing scheme, specified as the comma-separated pair consisting of `'InterlaceType'` and either `'none'` for no interlacing, or `'adam7'` to use the Adam7 algorithm.

Example: `'InterlaceType','adam7'`

`ResolutionUnit` — Unit for image resolution
`'unknown'` (default) | `'meter'`

Unit for image resolution, specified as the comma-separated pair consisting of `'ResolutionUnit'` and either `'unknown'` or `'meter'`. If you specify `ResolutionUnit`, you must include at least one of the `XResolution` and `YResolution` name-value pair arguments. When the value of `ResolutionUnit` is `'meter'`, the `XResolution` and `YResolution` values are interpreted in pixels per meter.

Example: `'ResolutionUnit','meter','XResolution',1000`

`SignificantBits` — Number of bits to regard as significant
`[]` (default) | scalar | vector

Number of bits in the data array to regard as significant, specified as the comma-separated pair consisting of `'SignificantBits'` and a scalar or a vector in the range [1,`BitDepth`]. Depending on the output image type, the value must be in the following form.

Image Type	Form of `SignificantBits` Value
Grayscale image without an alpha channel	Scalar
Grayscale image with an alpha channel	2-element vector
Indexed image	3-element vector
Truecolor image without an alpha channel	3-element vector
Truecolor image with an alpha channel	4-element vector

Example: `'SignificantBits',[2,3]`

`Software` — Software used to create the image
character vector | string scalar

Software used to create the image, specified as the comma-separated pair consisting of `'Software'` and a character vector or string scalar.

`Source` — Device used to create the image
character vector | string scalar

Device used to create the image, specified as the comma-separated pair consisting of `'Source'` and a character vector or string scalar.

`Transparency` — Pixels to consider transparent
`[]` (default) | scalar in the range [0,1] | vector

Pixels to consider transparent when no alpha channel is used, specified as the comma-separated pair consisting of `'Transparency'` and a scalar or a vector. Depending on the output image, the value must be in the following form.

Image Type	Form of `Transparency` Value
Grayscale images	Scalar in the range [0,1], indicating the grayscale color to be considered transparent.
Indexed images	`Q`-element vector of values in the range [0,1], where `Q` is no larger than the colormap length and each value indicates the transparency associated with the corresponding colormap entry. In most cases, `Q = 1`.
Truecolor images	3-element vector of RGB intensities in the range [0,1], indicating the truecolor color to consider transparent.

Note You cannot specify both `'Transparency'` and `'Alpha'` at the same time.

Example: `'Transparency',[1 1 1]`

Data Types: `double`

`Warning` — Warning of nature of content
character vector | string scalar

Warning of nature of content, specified as the comma-separated pair consisting of `'Warning'` and a character vector or string scalar.

`XResolution` — Image resolution in horizontal direction
scalar

Image resolution in the horizontal direction, in pixels/unit, specified as the comma-separated pair consisting of `'XResolution'` and a scalar. Define the unit by specifying the `ResolutionUnit` name-value pair argument.

If you do not also specify `YResolution`, then the `XResolution` value applies to both the horizontal and vertical directions.

Example: `'XResolution',900`

`YResolution` — Image resolution in vertical direction
scalar

Image resolution in the vertical direction, in pixels/unit, specified as the comma-separated pair consisting of `'XResolution'` and a scalar. Define the unit by specifying the `ResolutionUnit` name-value pair argument.

If you do not also specify `XResolution`, then the `YResolution` value applies to both the horizontal and vertical directions.

Example: `'YResolution',900`

In addition to the listed name-value pair arguments for PNG, you can use any parameter name that satisfies the PNG specification for keywords. That is, the name uses only printable characters, contains 80 or fewer characters, and does not contain leading or trailing spaces. The value corresponding to these user-specified names must be a character vector or string scalar that contains no control characters other than linefeed.

RAS — Sun Raster Graphic

`Alpha` — Transparency of each pixel
`[ ]` (default) | matrix

Transparency of each pixel, specified as the comma-separated pair consisting of `'Alpha'` and a matrix with row and column dimensions the same as those of the image data array.

Valid only for truecolor (`m`-by-`n`-by-3) image data.

Data Types: `double` | `single` | `uint8` | `uint16`

Type — Image type
`'standard'` (default) | `'rgb'` | `'rle'`

Image type, specified as the comma-separated pair consisting of `'Type'` and one of the options in this table.

Value of Type	Description
`'standard'` (default)	Uncompressed, B-G-R color order for truecolor images
`'rgb'`	Uncompressed, R-G-B color order for truecolor images
`'rle`	Run-length encoding of 1-bit and 8-bit images

Example: `'Type','rgb'`

TIFF — Tagged Image File Format

`ColorSpace` — Color space representing color data
`'rgb'` (default) | `'cielab'` | `'icclab'`

Color space representing the color data, specified as the comma-separated pair consisting of `'ColorSpace'` and one of these options:

- `'rgb'`
- `'cielab'`
- `'icclab'`

Valid only when the image data array, `A`, is truecolor (`m`-by-`n`-by-3). To use the CMYK color space in a TIFF file, do not use the `'ColorSpace'` name-value pair argument. Instead, specify an `m`-by-`n`-by-4 image data array.

`imwrite` can write color image data that uses the *L*a*b** color space to TIFF files. The 1976 CIE *L*a*b** specification defines numeric values that represent luminance (*L**) and chrominance (*a** and *b**) information. To store *L*a*b** color data in a TIFF file, the values must be encoded to fit into either 8-bit or 16-bit storage. `imwrite` can store *L*a*b** color data in a TIFF file using the following encodings:

- CIELAB encodings — 8-bit and 16-bit encodings defined by the TIFF specification

- ICCLAB encodings — 8-bit and 16-bit encodings defined by the International Color Consortium

The output class and encoding used by `imwrite` depends on the class of the input image data array and the `ColorSpace` value, as shown in the following table. (The 8-bit and 16-bit CIELAB encodings cannot be input arrays because they use a mixture of signed and unsigned values and cannot be represented as a single MATLAB array.)

Input Class and Encoding	Value of `ColorSpace`	Output Class and Encoding
8-bit ICCLAB Values are integers in the range [0 255]. *L** values are multiplied by `255/100`. `128` is added to both the *a** and *b** values.	`'icclab'`	8-bit ICCLAB
	`'cielab'`	8-bit CIELAB
16-bit ICCLAB Values are integers in the range [0, 65280]. *L** values are multiplied by `65280/100`. `32768` is added to both the *a** and *b** values, which are represented as integers in the range [0,65535].	`'icclab'`	16-bit ICCLAB
	`'cielab'`	16-bit CIELAB
Double-precision 1976 CIE *L*a*b** values *L** is in the dynamic range [0, 100]. *a** and *b** can take any value. Setting *a** and *b** to 0 (zero) produces a neutral color (gray).	`'icclab'`	8-bit ICCLAB
	`'cielab'`	8-bit CIELAB

Example: `'ColorSpace','cielab'`

`Compression` — Compression scheme
`'packbits'` | `'none'` | `'lzw'` | `'deflate'` | `'jpeg'` | `'ccitt'` | `'fax3'` | `'fax4'`

Compression scheme, specified as the comma-separated pair consisting of `'Compression'` and one of these options:

- `'packbits'` (default for nonbinary images)
- `'none'`
- `'lzw'`
- `'deflate'`
- `'jpeg'`
- `'ccitt'` (binary images only, and the default for such images)
- `'fax3'` (binary images only)
- `'fax4'` (binary images only)

`'jpeg'` is a lossy compression scheme; other compression modes are lossless. Also, if you specify `'jpeg'` compression, you must specify the `'RowsPerStrip'` parameter and the value must be a multiple of 8.

Example: `'Compression','none'`

`Description` — Image description
character vector | string scalar

Image description, specified by the comma-separated pair consisting of `'Description'` and a character vector or string scalar. This is the text that `imfinfo` returns in the `ImageDescription` field for the output image.

Example: `'Description','Sample 2A301'`

`Resolution` — X- and Y-resolution
72 (default) | scalar | two-element vector

X- and Y-resolution, specified as the comma-separated pair consisting of `'Resolution'` and a scalar indicating both resolution, or a two-element vector containing the X-Resolution and Y-Resolution.

Example: `'Resolution',80`

Example: `'Resolution',[320,72]`

Data Types: `double`

`RowsPerStrip` — Number of rows to include in each strip
scalar

Number of rows to include in each strip, specified as the comma-separated pair consisting of `'RowsPerStrip'` and a scalar. The default value is such that each strip is about 8 kilobytes.

You must specify `RowsPerStrip` if you specify `'jpeg'` compression. The value must be a multiple of 8.

Example: `'RowsPerStrip',16`

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`WriteMode` — Writing mode
`'overwrite'` (default) | `'append'`

Writing mode, specified as the comma-separated pair consisting of `'WriteMode'` and either `'overwrite'` or `'append'`. In `overwrite` mode, `imwrite` overwrites an existing file. In `append` mode, `imwrite` adds a page to the existing file.

Example: `'WriteMode','append'`

Tips

- For copyright information, see the `libtiffcopyright.txt` file.

See Also

`Tiff` | `fwrite` | `getframe` | `imfinfo` | `imformats` | `imread`

Introduced before R2006a

incenters

Class: `TriRep`

(Not recommended) Incenters of specified simplices

Note `incenters(TriRep)` is not recommended. Use `incenter(triangulation)` instead.

`TriRep` is not recommended. Use `triangulation` instead.

Syntax

```
IC = incenters(TR,SI)
[IC RIC] = incenters(TR, SI)
```

Description

`IC = incenters(TR,SI)` returns the coordinates of the incenter of each specified simplex `SI`.

`[IC RIC] = incenters(TR, SI)` returns the incenters and the corresponding radius of the inscribed circle/sphere.

Input Arguments

`TR`	Triangulation representation.
`SI`	Column vector of simplex indices that index into the triangulation matrix `TR.Triangulation`. If `SI` is not specified the incenter information for the entire triangulation is returned, where the incenter associated with simplex `i` is the `i`'th row of `IC`.

Output Arguments

`IC`	`m`-by-`n` matrix, where `m = length(SI)`, the number of specified simplices, and `n` is the dimension of the space where the triangulation resides. Each row `IC(i,:)` represents the coordinates of the incenter of simplex `SI(i)`.
`RIC`	Vector of length `length(SI)`, the number of specified simplices.

Examples

Example 1

Load a 3-D triangulation:

```
load tetmesh
```

Use `TriRep` to compute the incenters of the first five tetrahedra.

```
trep = TriRep(tet, X)
ic = incenters(trep, [1:5]')
```

Example 2

Query a 2-D triangulation created with `DelaunayTri`.

```
x = [0 1 1 0 0.5]';
y = [0 0 1 1 0.5]';
dt = DelaunayTri(x,y);
```

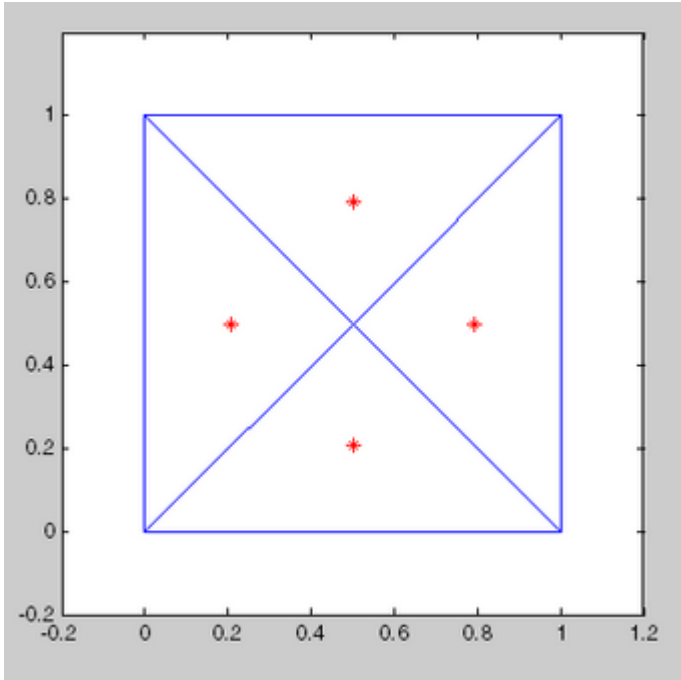
Compute incenters of the triangles:

```
ic = incenters(dt);
```

Plot the triangles and incenters:

```
triplot(dt);
axis equal;
axis([-0.2 1.2 -0.2 1.2]);
hold on;
```

```
plot(ic(:,1),ic(:,2),'*r');
hold off;
```



More About

Simplex

A simplex is a triangle/tetrahedron or higher-dimensional equivalent.

See Also

`circumcenter` | `delaunayTriangulation` | `triangulation`

inOutStatus

Class: `DelaunayTri`

(Not recommended) Status of triangles in 2-D constrained Delaunay triangulation

---

Note `inOutStatus(DelaunayTri)` is not recommended. Use `isInterior(delaunayTriangulation)` instead.

`DelaunayTri` is not recommended. Use `delaunayTriangulation` instead.

Syntax

```
IN = inOutStatus(DT)
```

Description

`IN = inOutStatus(DT)` returns the in/out status of the triangles in a 2-D constrained Delaunay triangulation of a geometric domain. Given a Delaunay triangulation that has a set of constrained edges that define a bounded geometric domain. The `i`'th triangle in the triangulation is classified as inside the domain if `IN(i) = 1` and outside otherwise.

Note `inOutStatus` is only relevant for 2-D constrained Delaunay triangulations where the imposed edge constraints bound a closed geometric domain.

Input Arguments

`DT`	Delaunay triangulation.

Output Arguments

`IN`	Logical array of length equal to the number of triangles in the triangulation. The constrained edges in the triangulation define the boundaries of a valid geometric domain.

Examples

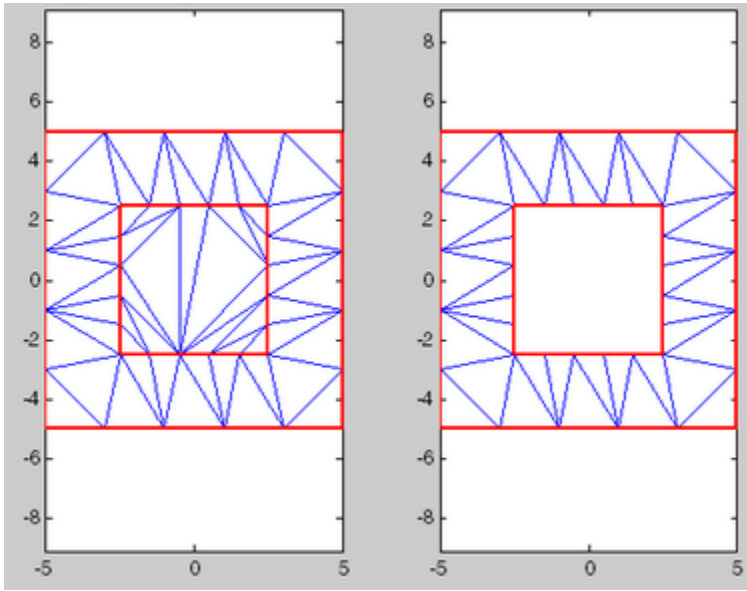
Create a geometric domain that consists of a square with a square hole:

```
outerprofile = [-5 -5; -3 -5; -1 -5; 1 -5; 3 -5; ...
 5 -5; 5 -3; 5 -1; 5 1; 5 3;...
 5 5; 3 5; 1 5; -1 5; -3 5; ...
 -5 5; -5 3; -5 1; -5 -1; -5 -3; ];
innerprofile = outerprofile.*0.5;
profile = [outerprofile; innerprofile];
outercons = [(1:19)' (2:20)'; 20 1;];
innercons = [(21:39)' (22:40)'; 40 21];
edgeconstraints = [outercons; innercons];
```

Create a constrained Delaunay triangulation of the domain:

```
dt = DelaunayTri(profile, edgeconstraints)
subplot(1,2,1);
triplot(dt);
hold on;
plot(dt.X(outercons',1), dt.X(outercons',2), ...
     '-r', 'LineWidth', 2);
plot(dt.X(innercons',1), dt.X(innercons',2), ...
     '-r', 'LineWidth', 2);
axis equal;
% Plot showing interior and exterior
% triangles with respect to the domain.
hold off;
subplot(1,2,2);
inside = inOutStatus(dt);
triplot(dt(inside, :), dt.X(:,1), dt.X(:,2));
hold on;
plot(dt.X(outercons',1), dt.X(outercons',2), ...
     '-r', 'LineWidth', 2);
plot(dt.X(innercons',1), dt.X(innercons',2), ...
     '-r', 'LineWidth', 2);
```

```
axis equal;
% Plot showing interior triangles only
hold off;
```



See Also

delaunayTriangulation | isInterior | triangulation

ind2rgb

Convert indexed image to RGB image

Syntax

```
RGB = ind2rgb(X,map)
```

Description

`RGB = ind2rgb(X,map)` converts the indexed image `X` and corresponding colormap `map` to RGB (truecolor) format.

Examples

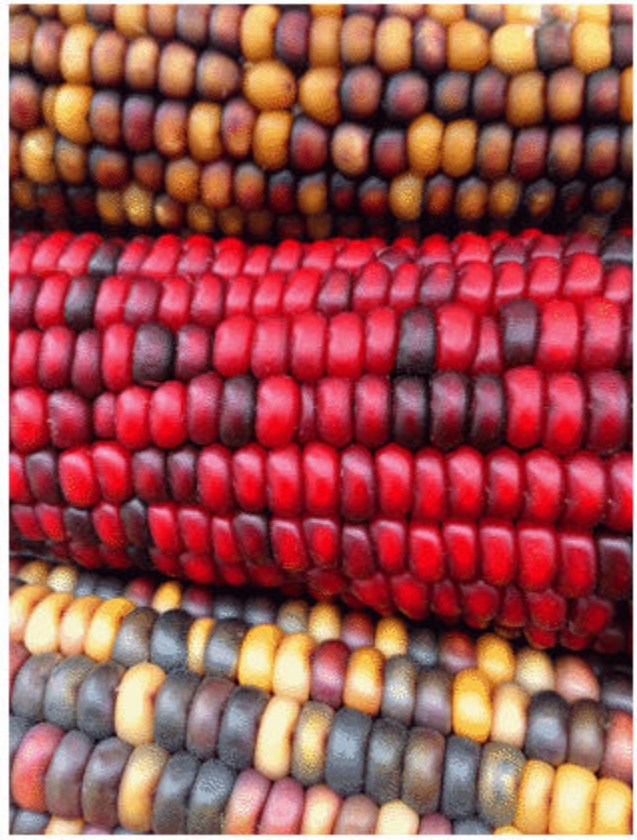
Convert Indexed Image to RGB

Read the first image in the sample indexed image file, `corn.tif`.

```
[X,cmap] = imread('corn.tif');
```

The indexed image `X` is a 415-by-312 array of type `uint8`. The colormap `cmap` is a 256-by-3 matrix of type `double`, therefore there are 256 colors in the indexed image. Display the image.

```
imshow(X,cmap)
```



Convert the indexed image to an RGB image. The result is a 415-by-312-by-3 array of type `double`.

```
RGB = ind2rgb(X,cmap);
```

Check that values of the RGB image are in the range [0, 1].

```
disp(['Range of RGB image is [',num2str(min(RGB(:))),', ',num2str(max(RGB(:))),'].'])
```

```
Range of RGB image is [0.0078431, 0.97647].
```

Input Arguments

X — Indexed image
m-by-*n* matrix of integers

Indexed image, specified as an *m*-by-*n* matrix of integers.

- If you specify X as an array of integer data type, then the value 0 corresponds to the first color in the colormap `map`. For a colormap containing *c* colors, values of image X are clipped to the range [0, *c*-1].
- If you specify X as an array of data type `single` or `double`, then the value 1 corresponds to the first color in the colormap. For a colormap containing *c* colors, values of image X are clipped to the range [1, *c*].

Data Types: `single` | `double` | `uint8` | `uint16`

`map` — Colormap
c-by-3 matrix

Colormap associated with indexed image X, specified as a *c*-by-3 matrix with values in the range [0, 1]. Each row of `map` is a three-element RGB triplet that specifies the red, green, and blue components of a single color of the colormap.

Data Types: `double`

Output Arguments

RGB — RGB image
m-by-*n*-by-3 numeric array

RGB image, returned as an *m*-by-*n*-by-3 numeric array with values in the range [0, 1].

Data Types: `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`image` | `imread` | `rgb2ind`

Topics

"Image Types"

Introduced before R2006a

ind2sub

Convert linear indices to subscripts

Syntax

```
[row,col] = ind2sub(sz,ind)
[I1,I2,...,In] = ind2sub(sz,ind)
```

Description

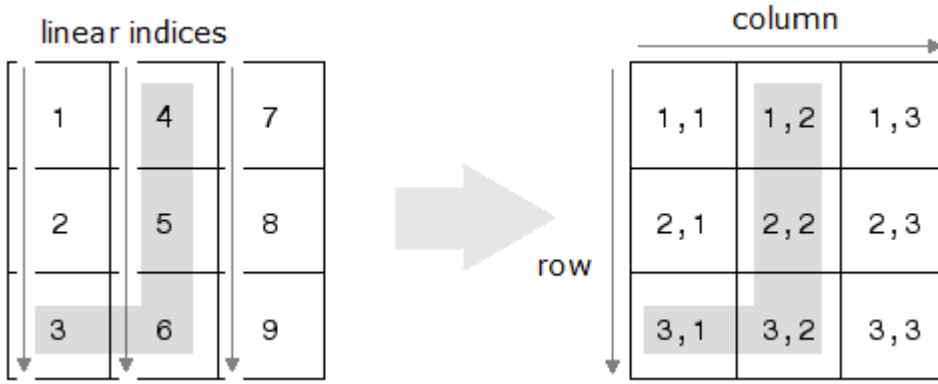
`[row,col] = ind2sub(sz,ind)` returns the arrays `row` and `col` containing the equivalent row and column subscripts corresponding to the linear indices `ind` for a matrix of size `sz`. Here `sz` is a vector with two elements, where `sz(1)` specifies the number of rows and `sz(2)` specifies the number of columns.

`[I1,I2,...,In] = ind2sub(sz,ind)` returns `n` arrays `I1,I2,...,In` containing the equivalent multidimensional subscripts corresponding to the linear indices `ind` for a multidimensional array of size `sz`. Here `sz` is a vector with `n` elements that specifies the size of each array dimension.

Examples

Convert Linear Indices to Subscripts for Matrix

Convert the linear indices `[3 4 5 6]` to row and column subscripts in a 3-by-3 matrix. The mapping from linear indices to subscripts (indexing by position) is illustrated in the following.



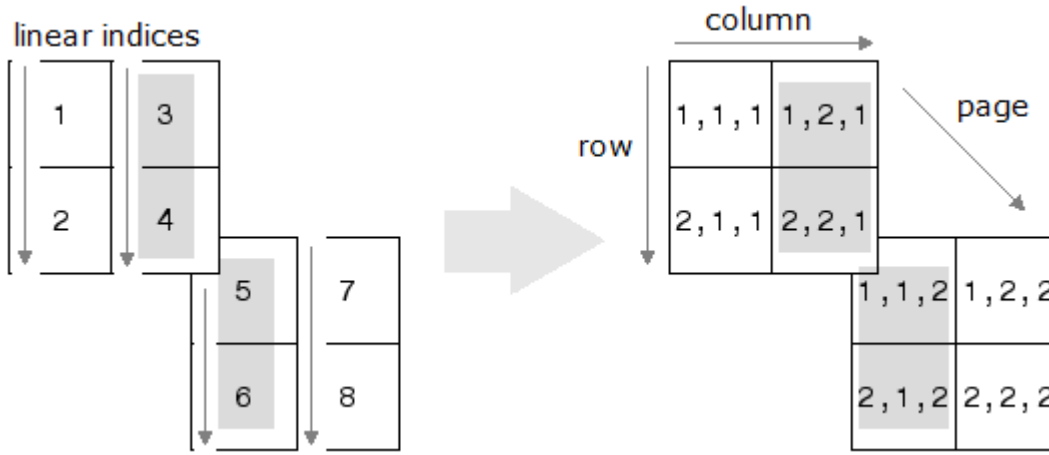
Create input vectors and perform the conversion.

```
ind = [3 4 5 6];
sz = [3 3];
[row,col] = ind2sub(sz,ind)
```

```
row = 1×4

     3     1     2     3


col = 1×4

     1     2     2     2
```

Convert Linear Indices to Subscripts for Three-Dimensional Array

Convert the linear indices `[3 4 5 6]` to subscripts in a 2-by-2-by-2 array. The mapping from linear indices to subscripts (indexing by position) for a 2-by-2-by-2 array can be illustrated as in the following.



Create input vectors and perform the conversion.

```
ind = [3 4 5 6];
sz = [2 2 2];
[I1,I2,I3] = ind2sub(sz,ind)
```

```
I1 = 1×4

     1     2     1     2


I2 = 1×4

     2     2     1     1


I3 = 1×4

     1     1     2     2
```

Access Element of 3-D Array

Convert a linear index of a 3-D array to a subscript index.

Create an array, and find the subscript index corresponding to the 14 th element of the array.

```
A = rand(3,4,2);
[row,col,page] = ind2sub(size(A),14)
```

```
row = 2

col = 1

page = 2
```

Check that both index versions refer to the same element of the array.

```
A(14)
```

```
ans = 0.4854
```

```
A(row,col,page)
```

```
ans = 0.4854
```

Use `ind2sub` with Fewer Output Arguments

When using `ind2sub` for an N-dimensional array, you would typically supply N output arguments for each dimension of the matrix. This example shows the different results when you return fewer output arguments for a 3-dimensional array.

Create the input arguments needed to convert the linear indices 1 through 8 for a 3-dimensional array with size 2-by-2-by-2.

```
ind = 1:8;
sz = [2 2 2];
```

Specify three output arguments when using `ind2sub` to return the row, column, and page subscripts for the 2-by-2-by-2 array.

```
[row,col,page] = ind2sub(sz,ind)
```

```
row = 1×8

     1     2     1     2     1     2     1     2
```

```
col = 1×8

     1     1     2     2     1     1     2     2


page = 1×8

     1     1     1     1     2     2     2     2
```

If you specify only two output arguments, `ind2sub` ignores the third dimension of the array and returns subscripts for a 2-dimensional array with size 2-by-4 instead.

```
[row,col] = ind2sub(sz,ind)
```

```
row = 1×8

     1     2     1     2     1     2     1     2


col = 1×8

     1     1     2     2     3     3     4     4
```

If you specify only one output argument, `ind2sub` ignores the second and third dimensions of the array and returns subscripts for a 1-dimensional array with size 1-by-8 instead.

```
row = ind2sub(sz,ind)
```

```
row = 1×8

     1     2     3     4     5     6     7     8
```

Input Arguments

`sz` — Size of array
vector of positive integers

Size of array, specified as a vector of positive integers. Each element of this vector indicates the size of the corresponding dimension. For example, `[2 3 4]` defines a 2-by-3-by-4 array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ind` — Linear indices
scalar | vector | matrix | multidimensional array

Linear indices, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`row` — Row subscripts
scalar | vector | matrix | multidimensional array

Row subscripts, returned as a scalar, vector, matrix, or multidimensional array. The size of `row` is the same as the size of the input `ind`.

Data Types: `double`

`col` — Column subscripts
scalar | vector | matrix | multidimensional array

Column subscripts, returned as a scalar, vector, matrix, or multidimensional array. The size of `col` is the same as the size of the input `ind`.

Data Types: `double`

`I1,I2,...,In` — Multidimensional subscripts
scalar | vector | matrix | multidimensional array

Multidimensional subscripts, returned as a scalar, vector, matrix, or multidimensional array. The size of each array `I1,I2,…,In` is the same as the size of the input `ind`.

Data Types: `double`

Tips

- To get the linear indices of matrix elements that satisfy a specific condition for matrix `A`, you can use the `find` function with one output argument. To get the subscript indices, use the `find` function with two output arguments. For example, `[row,col] = ind2sub(size(A),find(A>5))` gives the same result as `[row,col] = find(A>5)`.

Algorithms

For an array `A`, if `[I1,…,In] = ind2sub(size(A),ind)`, then `A(I1(k),…,In(k)) = A(ind(k))` for all `k`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The first argument must be a valid size vector. Size vectors for arrays with more than `intmax` elements are not supported.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`find` | `size` | `sub2ind`

Topics

“Array Indexing”
“Multidimensional Arrays”

Introduced before R2006a

Inf

Create array of all `Inf` values

Syntax

```
X = Inf
X = Inf(n)
X = Inf(sz1,...,szN)
X = Inf(sz)
```

```
X = Inf( ___ ,typename)
X = Inf( ___ ,'like',p)
```

Description

`X = Inf` returns the scalar representation of positive infinity. Operations return `Inf` when their result is too large to represent as a floating point number, such as `1/0` or `log(0)`.

For double-precision, `Inf` represents numbers larger than `realmax`. For single-precision, `Inf` represents numbers larger than `realmax('single')`.

`X = Inf(n)` returns an `n`-by-`n` matrix of `Inf` values.

`X = Inf(sz1,...,szN)` returns an `sz1`-by-...-by-`szN` array of `Inf` values, where `sz1,...,szN` indicate the size of each dimension. For example, `Inf(3,4)` returns a 3-by-4 matrix.

`X = Inf(sz)` returns an array of `Inf` values, where the size vector `sz` defines `size(X)`. For example, `Inf([3 4])` returns a 3-by-4 matrix.

`X = Inf( ___ ,typename)` returns an array of `Inf` values of data type `typename`, which can be either `'single'` or `'double'`.

`X = Inf( ___ ,'like',p)` returns an array of `Inf` values of the same data type, sparsity, and complexity (real or complex) as `p`. You can specify `typename` or `'like'` but not both.

Examples

Matrix of `Inf` Values

Create a 3-by-3 matrix of `Inf` values.

```
X = Inf(3)
```

```
X = 3×3

   Inf   Inf   Inf
   Inf   Inf   Inf
   Inf   Inf   Inf
```

3-D Array of `Inf` Values

Create a 2-by-3-by-4 array of `Inf` values and display its size.

```
X = Inf(2,3,4);
size(X)
```

```
ans = 1×3

     2     3     4
```

Clone Size from Existing Array

Create an array of `Inf` values that is the same size as an existing array.

```
A = [1 4; 2 5; 3 6];
sz = size(A);
X = Inf(sz)
```

```
X = 3×2

   Inf   Inf
```

```
   Inf   Inf
   Inf   Inf
```

It is a common pattern to combine the previous two lines of code into a single line.

```
X = Inf(size(A));
```

Specify Data Type of `Inf` Values

Create a 1-by-3 vector of `Inf` values whose elements are of type `single`.

```
X = Inf(1,3,'single')
```

```
X = 1x3 single row vector

   Inf   Inf   Inf
```

You can also specify the output type based on the type of another variable. Create a variable `p` of type `single`. Then, create a vector of `Inf` values with the same size and type as `p`.

```
p = single([1 2 3]);
X = Inf(size(p),'like',p)
```

```
X = 1x3 single row vector

   Inf   Inf   Inf
```

Input Arguments

n — Size of square matrix
integer

Size of square matrix, specified as an integer.

- If `n` is 0, then `X` is an empty matrix.

- If n is negative, then it is treated as 0.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

sz1,...,szN — Size of each dimension in a list
integers

Size of each dimension in a list, specified as separate integer arguments.

- If the size of any dimension is 0, then X is an empty array.
- If the size of any dimension is negative, then it is treated as 0.
- Beyond the second dimension, Inf ignores trailing dimensions of length 1. For example, Inf(3,1,1) creates a 3-by-1 vector of Inf values.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

sz — Size of each dimension in a vector
row vector of integers

Size of each dimension in a vector, specified as a row vector of integers.

- If the size of any dimension is 0, then X is an empty array.
- If the size of any dimension is negative, then it is treated as 0.
- Beyond the second dimension, Inf ignores trailing dimensions of length 1. For example, Inf([3 1]) creates a 3-by-1 vector of Inf values.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

typename — Data type
'double' (default) | single'

Data type to create, specified as 'double' or 'single'.

p — Prototype of array
array

Prototype of array to create, specified as an array.

Data Types: double | single

Complex Number Support: Yes

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Dimensions must be real, nonnegative, integers.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `Inf` in the Parallel Computing Toolbox documentation.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `Inf` in the Parallel Computing Toolbox documentation.

See Also

`NaN` | `isfinite` | `isinf` | `realmax`

Introduced before R2006a

info

Information about contacting MathWorks

Note `info` will be removed in a future release.

Syntax

```
info
```

Description

`info` displays in the Command Window, information about contacting MathWorks.

See Also

`help` | `version`

Introduced before R2006a

inline

Construct inline object

Note `inline` will be removed in a future release. Use “Anonymous Functions” instead.

Syntax

```
inline(expr)
inline(expr,arg1,arg2,...)
inline(expr,n)
```

Description

`inline(expr)` constructs an inline function object from the MATLAB expression contained in `expr`. The input argument to the inline function is automatically determined by searching `expr` for an isolated lower case alphabetic character, other than `i` or `j`, that is not part of a word formed from several alphabetic characters. If no such character exists, `x` is used. If the character is not unique, the one closest to `x` is used. If two characters are found, the one later in the alphabet is chosen.

`inline(expr,arg1,arg2,...)` constructs an inline function whose input arguments are specified by `arg1`, `arg2`,`...`. Multicharacter symbol names may be used.

`inline(expr,n)` where `n` is a scalar, constructs an inline function whose input arguments are `x`, `P1`, `P2`, ...

Examples

Example 1

This example creates a simple inline function to square a number.

```
g = inline('t^2')
g =
```

```
     Inline function:
     g(t) = t^2
```

You can convert the result to a character vector using the `char` function.

```
char(g)

ans =

t^2
```

Example 2

This example creates an inline function to represent the formula $f = 3\sin(2x^2)$. The resulting inline function can be evaluated with the `argnames` and `formula` functions.

```
f = inline('3*sin(2*x.^2)')

f =
     Inline function:
     f(x) = 3*sin(2*x.^2)

argnames(f)

ans =
    'x'

formula(f)
ans =

3*sin(2*x.^2)
```

Example 3

This call to `inline` defines the function `f` to be dependent on two variables, `alpha` and `x`:

```
f = inline('sin(alpha*x)')

f =
     Inline function:
     f(alpha,x) = sin(alpha*x)
```

If `inline` does not return the desired function variables or if the function variables are in the wrong order, you can specify the desired variables explicitly with the `inline` argument list.

```
g = inline('sin(alpha*x)','x','alpha')

g =

     Inline function:
     g(x,alpha) = sin(alpha*x)
```

Tips

Three commands related to `inline` allow you to examine an inline function object and determine how it was created.

`char(fun)` converts the inline function into a character array. This is identical to `formula(fun)`.

`argnames(fun)` returns the names of the input arguments of the inline object `fun` as a cell array of character vectors.

`formula(fun)` returns the formula for the inline object `fun`.

A fourth command `vectorize(fun)` inserts a `.` before any `^`, `*` or `/'` in the formula for `fun`. The result is a vectorized version of the inline function.

Introduced before R2006a

inmem

Names of functions, MEX-files, classes in memory

Syntax

```
M = inmem
[M,X] = inmem
[M,X,C] = inmem
[...] = inmem('-completenames')
```

Description

`M = inmem` returns a cell array of character vectors containing the names of the functions that are currently loaded.

`[M,X] = inmem` returns an additional cell array `X` containing the names of the MEX-files that are currently loaded.

`[M,X,C] = inmem` also returns a cell array `C` containing the names of the classes that are currently loaded.

`[...] = inmem('-completenames')` returns not only the names of the currently loaded function and MEX-files, but the path and filename extension for each as well. No additional information is returned for loaded classes.

Examples

Functions in Memory

List the functions that remain in memory after calling the `magic` function.

```
clear all
magic(10);
```

```
M = inmem
```

```
M 
    'workspacefunc'
    'magic'
```

The function list includes `magic` and additional functions that are in memory in your current session.

MEX-Files in Memory

Call a sample MEX-function named `arrayProduct`, and then verify that the MEX-function is in memory. You must have a supported C compiler installed on your system to run this example.

```
clear all
sampleFolder = fullfile(matlabroot,'extern','examples','mex');
addpath(sampleFolder)
mex arrayProduct.c

s = 5;
A = [1.5, 2, 9];
B = arrayProduct(s,A);

[M,X] = inmem('-completenames');
X
```

```
X 
    'matlabroot\extern\examples\mex\arrayProduct.mexw64'
```

Tips

If `inmem` is called with any argument other than `'-completenames'`, it behaves as if it were called with no argument.

See Also

`clear`

Introduced before R2006a

inner2outer

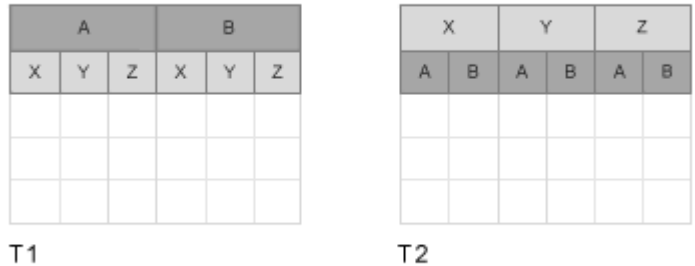
Invert nested table-in-table hierarchy in tables or timetables

Syntax

```
T2 = inner2outer(T1)
```

Description

`T2 = inner2outer(T1)` finds the variables in `T1` that are themselves tables or timetables. It returns `T2`, a table or timetable that also contains nested tables or timetables as variables. The names of the variables in `T2` are taken from the names of the variables inside the nested tables or timetables of `T1`. Then, `inner2outer` regroups variables in the nested tables or timetables of `T2` appropriately, as shown in the diagram. If `T1` has variables that are not tables or timetables, then those variables are unaltered in `T2`.



Examples

Invert Nested Tables

Load and display a timetable, `T1`, that has nested tables containing stock information. The nested tables `AAPL` and `MSFT` are the variables of `T1`. Each nested table has the stock prices at the open and close of trading, and the volume, for a different company.

```
load nestedTables
T1
```

```
T1 =

  3x2 timetable

      Dates                  AAPL                          MSFT
                   Open     Close     Volume     Open     Close     Volume
    ___________    ____________________________    ____________________________

    01-Jan-2017    64.539    71.704    107.17    66.429     91.77      78.7
    01-Feb-2017    101.53    87.619    57.909    72.984    84.629    57.959
    01-Mar-2017    60.381    76.464    72.067    78.127    76.492    82.883
```

To group the `Open`, `Close`, and `Volume` variables together in nested tables of their own, use the `inner2outer` function.

```
T2 = inner2outer(T1)
```

```
T2 =

  3x3 timetable

      Dates             Open                Close                Volume
                   AAPL      MSFT      AAPL      MSFT      AAPL      MSFT
    ___________    _________________    _________________    _________________

    01-Jan-2017    64.539    66.429    71.704     91.77    107.17      78.7
    01-Feb-2017    101.53    72.984    87.619    84.629    57.909    57.959
    01-Mar-2017    60.381    78.127    76.464    76.492    72.067    82.883
```

Some calculations are more convenient with data from each stock grouped in the nested tables of `T2`. For example, you can calculate the normalized volume for all stocks using `T2.Volume`.

Use the `Variables` property of `T2` to convert `T2.Volume` into a matrix. Then subtract the mean of `T2.Volume` from `T2.Volume` and return the result as a matrix.

```
normVolume = T2.Volume.Variables - mean(T2.Volume.Variables)
```

```
normVolume =

   28.1213    5.5193
  -21.1397  -15.2217
   -6.9817    9.7023
```

You also can use table functions on the nested tables. Calculate the mean closing price of all stocks using the `varfun` function, returning the means in a table.

```
meanClose = varfun(@mean,T2.Close)
```

```
meanClose =

  1x2 table

    mean_AAPL    mean_MSFT
    _________    _________

     78.596       84.297
```

Input Arguments

T1 — Input table
table | timetable

Input table, specified as a table or timetable.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

See Also

`mergevars` | `rows2vars` | `splitvars`

Introduced in R2018a

innerjoin

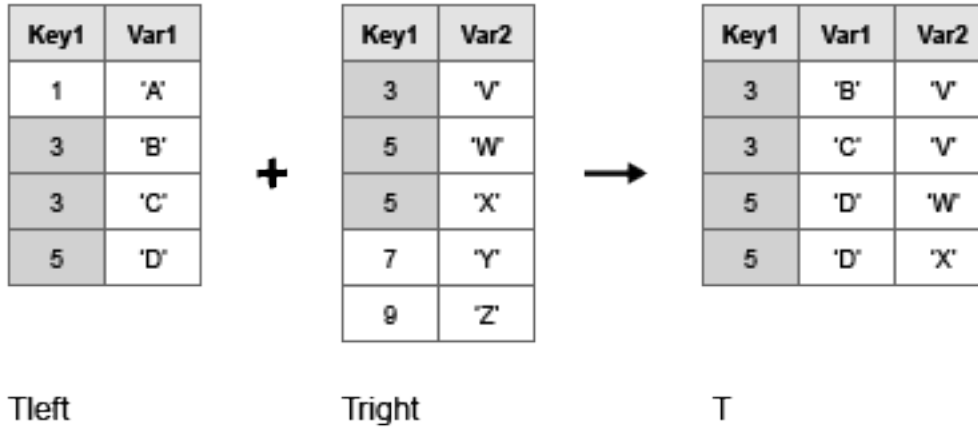
Inner join between two tables or timetables

Syntax

```
T = innerjoin(Tleft,Tright)
T = innerjoin(Tleft,Tright,Name,Value)
[T,ileft,iright] = innerjoin( ___ )
```

Description

`T = innerjoin(Tleft,Tright)` creates the table or timetable, `T`, as the *inner join* of `Tleft` and `Tright` using key variables on page 1-7689. All variables with the same names in both inputs are key variables. An inner join combines rows where the key variables have matching values. For example, if `Tleft` has variables named `Key1` and `Var1`, and `Tright` has variables `Key1` and `Var2`, then `T=innerjoin(Tleft,Tright)` uses `Key1` as a key variable.



The matching values of the key variables in the left and right tables do not have to be in the same order. Inner joins can perform one-to-many and many-to-one matches between the key variables of the two tables. That is, a value that occurs once in a key variable of the left table can have multiple matches in the right table. Similarly, a value that occurs once in a key variable of the right table can have multiple matches in the left table.

You can perform inner joins only on certain combinations of tables and timetables.

- If `Tleft` is a table, then `Tright` must be a table. `innerjoin` returns `T` as a table.
- If `Tleft` is a timetable, then `Tright` can be either a table or a timetable. `innerjoin` returns `T` as a timetable for either combination of inputs.

The vectors of row labels of `Tleft` and `Tright` can be key variables. Row labels are the row names of a table, or the row times of a timetable.

`T = innerjoin(Tleft,Tright,Name,Value)` performs the inner-join operation with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify the variables to use as key variables.

`[T,ileft,iright] = innerjoin( ___ )` also returns index vectors, `ileft` and `iright` indicating the correspondence between rows in `T` and rows in `Tleft` and `Tright`

respectively. You can use this syntax with any of the input arguments in the previous syntaxes.

Examples

Inner-Join Operation of Tables with One Variable in Common

Create a table, `Tleft`.

```
Tleft = table([5;12;23;2;6],...
        {'cereal';'pizza';'salmon';'cookies';'pizza'},...
        'VariableNames',{'Age','FavoriteFood'})
```

```
Tleft=5×2 table
    Age    FavoriteFood
    ___    ____________

     5     {'cereal' }
    12     {'pizza'  }
    23     {'salmon' }
     2     {'cookies'}
     6     {'pizza'  }
```

Create a table, `Tright`, with one variable in common with `Tleft`.

```
Tright = table({'cereal';'cookies';'pizza';'salmon';'cake'},...
         [110;160;140;367;243],...
         {'A-';'D';'B';'B';'C-'},...
         'VariableNames',{'FavoriteFood','Calories','NutritionGrade'})
```

```
Tright=5×3 table
    FavoriteFood    Calories    NutritionGrade
    ____________    ________    ______________

    {'cereal' }       110           {'A-'}
    {'cookies'}       160           {'D' }
    {'pizza'  }       140           {'B' }
    {'salmon' }       367           {'B' }
    {'cake'   }       243           {'C-'}
```

Use the `innerjoin` function to create a new table, `T`, with data from tables `Tleft` and `Tright`.

```
T = innerjoin(Tleft,Tright)
```

```
T=5×4 table
    Age    FavoriteFood    Calories    NutritionGrade
    ___    ____________    ________    ______________

     5     {'cereal' }       110           {'A-'}
     2     {'cookies'}       160           {'D' }
    12     {'pizza'  }       140           {'B' }
     6     {'pizza'  }       140           {'B' }
    23     {'salmon' }       367           {'B' }
```

Table `T` is sorted by the key variable, `FavoriteFood`.

Inner-Join Operation of Tables and Indices to Values

Create a table, `Tleft`.

```
Tleft = table({'a' 'b' 'c' 'e' 'h'}',[1 2 3 11 17]',...
        'VariableNames',{'Key1' 'Var1'})
```

```
Tleft=5×2 table
    Key1     Var1
    _____    ____

    {'a'}      1
    {'b'}      2
    {'c'}      3
    {'e'}     11
    {'h'}     17
```

Create a table, `Tright`, with common values in the variable `Key1` between tables `Tleft` and `Tright`, but also containing rows with values of `Key1` not present in `Tleft`.

```
Tright = table({'a' 'b' 'd' 'e'}',[4 5 6 7]',...
         'VariableNames',{'Key1' 'Var2'})
```

```
Tright=4×2 table
    Key1     Var2
    _____    ____

    {'a'}     4
    {'b'}     5
    {'d'}     6
    {'e'}     7
```

Use the `innerjoin` function to create a new table, `T`, with data from tables `Tleft` and `Tright`. Retain only rows whose values in the variable `Key1` match.

Also, return index vectors, `ileft` and `iright` indicating the correspondence between rows in `T` and rows in `Tleft` and `Tright` respectively.

```
[T,ileft,iright] = innerjoin(Tleft,Tright)
```

```
T=3×3 table
    Key1     Var1    Var2
    _____    ____    ____

    {'a'}      1       4
    {'b'}      2       5
    {'e'}     11       7


ileft = 3×1

     1
     2
     4


iright = 3×1

     1
     2
     4
```

Table `T` is sorted by the values in the key variable, `Key1`, and contains the horizontal concatenation of `Tleft(ileft,:)` and `Tright(iright,'Var2')`.

Inner-Join Operation of Tables Using Left and Right Keys

Create a table, `Tleft`.

```
Tleft = table([10;4;2;3;7],[5;4;9;6;1],[10;3;8;8;4])
```

```
Tleft=5×3 table
    Var1    Var2    Var3
    ____    ____    ____

     10      5       10
      4      4        3
      2      9        8
      3      6        8
      7      1        4
```

Create a table, `Tright`, with common values in the second variable as the first variable of table `Tleft`.

```
Tright = table([6;1;1;6;8],[2;3;4;5;6])
```

```
Tright=5×2 table
    Var1    Var2
    ____    ____

     6       2
     1       3
     1       4
     6       5
     8       6
```

Use the `innerjoin` function to create a new table, `T`, with data from tables `Tleft` and `Tright`. Use the first variable of `Tleft` and the second variable of `Tright` as key variables.

```
[T,ileft,iright] = innerjoin(Tleft,Tright,'LeftKeys',1,'RightKeys',2)
```

```
T=3×4 table
    Var1_Tleft    Var2    Var3    Var1_Tright
    __________    ____    ____    ___________

        2          9       8           6
        3          6       8           1
```

```
        4          4       3           1


ileft = 3×1

     3
     4
     2


iright = 3×1

     1
     2
     3
```

Table `T` retains only the rows that match between `Tleft` and `Tright` with respect to the key variables.

Table `T` contains the horizontal concatenation of `Tleft(ileft,:)` and `Tright(iright,'Var1')`.

Inner-Join Operation of Timetables

Create two timetables, `Tleft` and `Tright`. They have some row times in common, but each also includes row times that are not in the other timetable.

```
Tleft = timetable(seconds([1;2;4;6]),[1 2 3 11]')
```

```
Tleft=4×2 timetable
    Time     Var1
    _____    ____

    1 sec      1
    2 sec      2
    4 sec      3
    6 sec     11


Tright = timetable(seconds([2;4;6;7]),[4 5 6 7]')
```

```
Tright=4×2 timetable
    Time     Var1
    _____    ____

    2 sec     4
    4 sec     5
    6 sec     6
    7 sec     7
```

Combine `Tleft` and `Tright` with an inner join. `T` matches up the rows with common row times. `T` does not contain any other rows from either timetable.

```
T = innerjoin(Tleft,Tright)
```

```
T=3×3 timetable
    Time     Var1_Tleft    Var1_Tright
    _____    __________    ___________

    2 sec         2             4
    4 sec         3             5
    6 sec        11             6
```

Input Arguments

`Tleft` — Left table
table | timetable

Left table, specified as a table or a timetable.

`Tright` — Right table
table | timetable

Right table, specified as a table or a timetable.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Keys',2` uses the second variable in `Tleft` and the second variable in `Tright` as key variables.

Keys — Variables to use as keys
positive integer | vector of positive integers | character vector | cell array of character vectors | string array | logical vector

Variables to use as keys, specified as the comma-separated pair consisting of `'Keys'` and a positive integer, vector of positive integers, character vector, cell array of character vectors, string array, or logical vector.

You cannot use the `'Keys'` name-value pair argument with the `'LeftKeys'` and `'RightKeys'` name-value pair arguments.

A vector of row labels can be a key, alone or in combination with other key variables. For more information, see the "Tips" on page 1-7689 section.

Example: `'Keys',[1 3]` uses the first and third variables in `Tleft` and `Tright` as a key variables.

Example: `'Keys',{'X','Y'}` uses the variables named `X` and `Y` in `Tleft` and `Tright` as key variables.

Example: `'Keys','Row'` uses the vectors of row names of `Tleft` and `Tright` as key variables, if both `Tleft` and `Tright` are tables with row names.

`LeftKeys` — Variables to use as keys in `Tleft`
positive integer | vector of positive integers | character vector | cell array of character vectors | string array | logical vector

Variables to use as keys in `Tleft`, specified as the comma-separated pair consisting of `'LeftKeys'` and a positive integer, vector of positive integers, character vector, cell array of character vectors, string array, or logical vector.

You must use the `'LeftKeys'` name-value pair argument in conjunction with the `'RightKeys'` name-value pair argument. `'LeftKeys'` and `'RightKeys'` both must specify the same number of key variables. `innerjoin` pairs key values based on their order.

A vector of row labels can be a key, alone or in combination with other key variables. For more information, see the "Tips" on page 1-7689 section.

Example: `'LeftKeys',1` uses only the first variable in `Tleft` as a key variable.

`RightKeys` — Variables to use as keys in `Tright`

positive integer | vector of positive integers | character vector | cell array of character vectors | string array | logical vector

Variables to use as keys in `Tright`, specified as the comma-separated pair consisting of `'RightKeys'` and a positive integer, vector of positive integers, character vector, cell array of character vectors, string array, or logical vector.

You must use the `'RightKeys'` name-value pair argument in conjunction with the `'LeftKeys'` name-value pair argument. `'LeftKeys'` and `'RightKeys'` both must specify the same number of key variables. `innerjoin` pairs key values based on their order.

A vector of row labels can be a key, alone or in combination with other key variables. For more information, see the “Tips” on page 1-7689 section.

Example: `'RightKeys',3` uses only the third variable in `Tright` as a key variable.

`LeftVariables` — Variables from `Tleft` to include in `T`

positive integer | vector of positive integers | character vector | cell array of character vectors | string array | logical vector

Variables from `Tleft` to include in `T`, specified as the comma-separated pair consisting of `'LeftVariables'` and a positive integer, vector of positive integers, character vector, cell array of character vectors, string array, or logical vector.

You can use `'LeftVariables'` to include or exclude key variables, as well as nonkey variables from the output, `T`.

By default, `innerjoin` includes all variables from `Tleft`.

`RightVariables` — Variables from `Tright` to include in `T`

positive integer | vector of positive integers | character vector | cell array of character vectors | string array | logical vector

Variables from `Tright` to include in `T`, specified as the comma-separated pair consisting of `'RightVariables'` and a positive integer, vector of positive integers, character vector, cell array of character vectors, string array, or logical vector.

You can use `'RightVariables'` to include or exclude key variables, as well as nonkey variables from the output, `T`.

By default, `innerjoin` includes all the variables from `Tright` except the key variables.

Output Arguments

`T` — Inner join from `Tleft` and `Tright`
table | timetable

Inner join from `Tleft` and `Tright`, returned as a table or a timetable. The output table or timetable, `T`, contains one row for each pair of rows in `Tleft` and `Tright` that share the same combination of values in the key variables. If `Tleft` and `Tright` contain variables with the same name, `innerjoin` adds a unique suffix to the corresponding variable names in `T`.

In general, if there are `m` rows in `Tleft` and `n` rows in `Tright` that all contain the same combination of values in the key variables, then `T` contains `m*n` rows for that combination.

`T` is sorted by the values in the key variables and contains the horizontal concatenation of `Tleft(ileft,LeftVars)` and `Tright(iright,RightVars)`. By default, `LeftVars` consists of all the variables of `Tleft`, and `RightVars` consists of all the nonkey variables from `Tright`. Otherwise, `LefttVars` consists of the variables specified by the `'LeftVariables'` name-value pair argument, and `RightVars` is the variables specified by the `'RightVariables'` name-value pair argument.

You can store additional metadata such as descriptions, variable units, variable names, and row names in `T`. For more information, see the Properties sections of `table` or `timetable`.

`ileft` — Index to `Tleft`
column vector

Index to `Tleft`, returned as a column vector. Each element of `ileft` identifies the row in `Tleft` that corresponds to that row in the output table or timetable, `T`.

`iright` — Index to `Tright`
column vector

Index to `Tright`, returned as a column vector. Each element of `iright` identifies the row in `Tright` that corresponds to that row in the output table or timetable, `T`.

More About

Key Variable

Variable used to match and combine data between the input tables, `Tleft` and `Tright`.

Tips

The vector of row labels from an input table or timetable can be a key, alone or in combination with other key variables. Row labels are the row names of a table or the row times of a timetable. To use this vector as a key, specify it as `'Row'` (for the row names of a table), as the name of a timetable vector of row times, or as the value of *T*`.Properties.DimensionNames{1}`, where *T* is the table or timetable.

In general, `innerjoin` copies row labels from the input table `Tleft` to the output table `T`.

- If `Tleft` has no row labels, then `T` has no row labels.
- If `Tleft` has row labels, then `innerjoin` copies row labels from `Tleft` to create row labels in `T`.
  - However, if both `Tleft` and `Tright` are tables, but you do not specify either input table's row names as a key, then `innerjoin` does not create row names in `T`.

You cannot perform an inner join using the row labels of `Tleft` as the left key and a variable of `Tright` as the right key. To perform the inner join, convert the row labels of `Tleft` to a table variable and use the new table variable as a key.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see "Tall Arrays".

See Also

Join Tables | `join` | `outerjoin`

Introduced in R2013b

inpolygon

Points located inside or on edge of polygonal region

Syntax

```
in = inpolygon(xq,yq,xv,yv)
[in,on] = inpolygon(xq,yq,xv,yv)
```

Description

`in = inpolygon(xq,yq,xv,yv)` returns `in` indicating if the query points specified by `xq` and `yq` are inside or on the edge of the polygon area defined by `xv` and `yv`.

`[in,on] = inpolygon(xq,yq,xv,yv)` also returns `on` indicating if the query points are on the edge of the polygon area.

Examples

Points Inside Convex Polygon

Define a pentagon and a set of points. Then, determine which points lie inside (or on the edge) of the pentagon.

Define the x and y coordinates of polygon vertices to create a pentagon.

```
L = linspace(0,2*pi,6);
xv = cos(L)';
yv = sin(L)';
```

Define x and y coordinates of 250 random query points. Initialize the random-number generator to make the output of `randn` repeatable.

```
rng default
xq = randn(250,1);
yq = randn(250,1);
```

Determine whether each point lies inside or on the edge of the polygon area. Also determine whether any of the points lie on the edge of the polygon area.

```
[in,on] = inpolygon(xq,yq,xv,yv);
```

Determine the number of points lying inside or on the edge of the polygon area.

```
numel(xq(in))
```

```
ans = 80
```

Determine the number of points lying on the edge of the polygon area.

```
numel(xq(on))
```

```
ans = 0
```

Since there are no points lying on the edge of the polygon area, all 80 points identified by `xq(in)`, `yq(in)` are strictly inside the polygon area.

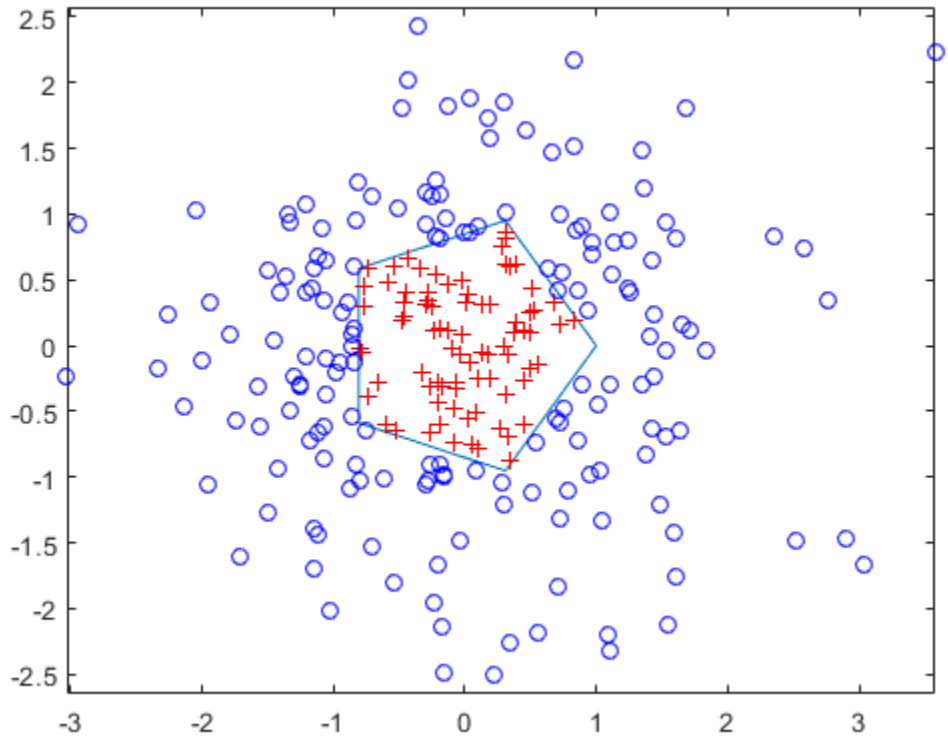
Determine the number of points lying outside the polygon area (not inside or on the edge).

```
numel(xq(~in))
```

```
ans = 170
```

Plot the polygon and the query points. Display the points inside the polygon with a red plus. Display the points outside the polygon with a blue circle.

```
figure

plot(xv,yv) % polygon
axis equal

hold on
plot(xq(in),yq(in),'r+') % points inside
plot(xq(~in),yq(~in),'bo') % points outside
hold off
```



Points Inside Multiply Connected Polygon

Find the points inside a square with a square hole.

Define a square region with a square hole. Specify the vertices of the outer loop in a counterclockwise direction, and specify the vertices for the inner loop in a clockwise direction. Use `NaN` to separate the coordinates for the outer and inner loops.

```
xv = [1 4 4 1 1 NaN 2 2 3 3 2];
yv = [1 1 4 4 1 NaN 2 3 3 2 2];
```

Define x and y coordinates of 500 random points. Initialize the random-number generator to make the output of `randn` repeatable.

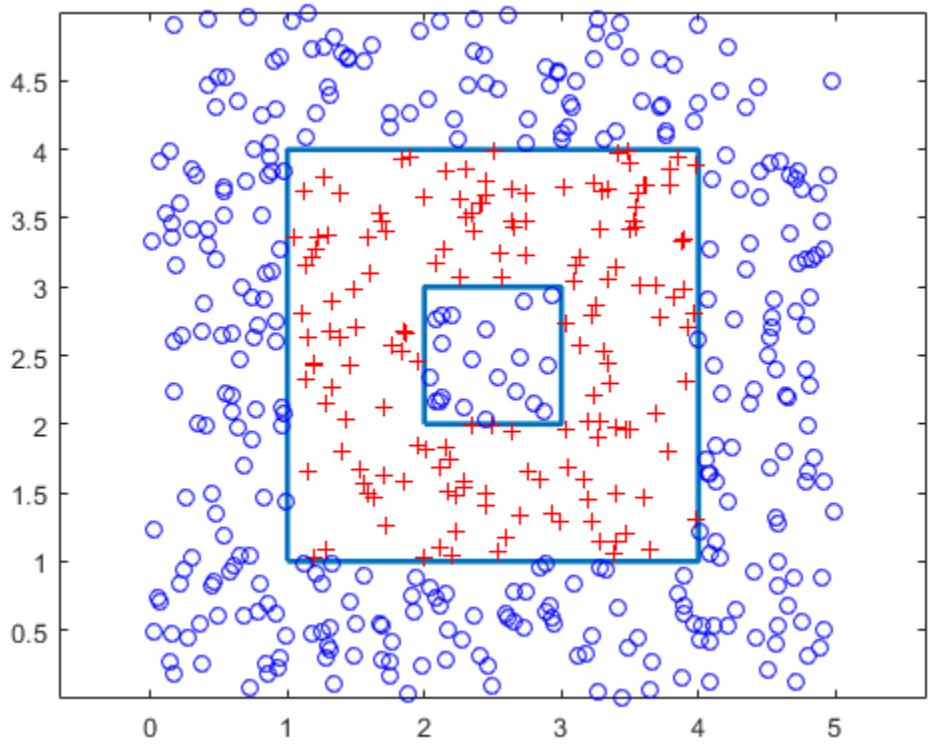
```
rng default
xq = rand(500,1)*5;
yq = rand(500,1)*5;
```

Determine whether each point lies inside or on the edge of the polygon area.

```
in = inpolygon(xq,yq,xv,yv);
```

Plot the polygon and the query points. Display the points inside the polygon with a red plus. Display the points outside the polygon with a blue circle.

```
figure

plot(xv,yv,'LineWidth',2) % polygon
axis equal

hold on
plot(xq(in),yq(in),'r+') % points inside
plot(xq(~in),yq(~in),'bo') % points outside
hold off
```



Query points in the square hole are outside the polygon.

Points Inside Self-Intersecting Polygon

Define the x and y coordinates for a pentagram.

```
xv = [0.5;0.2;1.0;0;0.8;0.5];
yv = [1.0;0.1;0.7;0.7;0.1;1];
```

Define the x and y coordinates of 12 query points.

```
xq = [0.1;0.5;0.9;0.2;0.4;0.5;0.5;0.9;0.6;0.8;0.7;0.2];
yq = [0.4;0.6;0.9;0.7;0.3;0.8;0.2;0.4;0.4;0.6;0.2;0.6];
```

Determine whether each point lies inside or on the edge of the polygon area. Also determine whether any of the points lie on the edge of the polygon area.

```
[in,on] = inpolygon(xq,yq,xv,yv);
```

Determine the number of points lying inside or on the edge of the polygon area.

```
numel(xq(in))
```

```
ans = 8
```

Determine the number of points lying on the edge of the polygon area.

```
numel(xq(on))
```

```
ans = 2
```

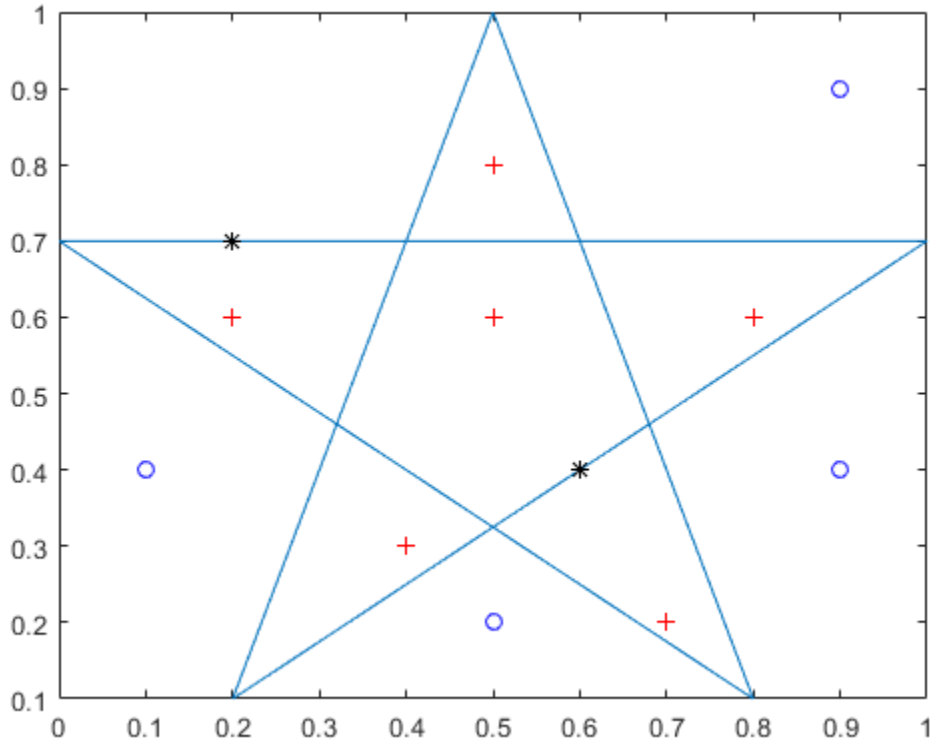
Determine the number of points lying outside the polygon area (not inside or on the edge).

```
numel(xq(~in))
```

```
ans = 4
```

Plot the polygon and the points. Display the points strictly inside the polygon with a red plus. Display the points on the edge with a black asterisk. Display the points outside the polygon with a blue circle.

```
figure

plot(xv,yv) % polygon

hold on
plot(xq(in&~on),yq(in&~on),'r+') % points strictly inside
plot(xq(on),yq(on),'k*') % points on edge
plot(xq(~in),yq(~in),'bo') % points outside
hold off
```



Six points lie inside the polygon. Two points lie on the edge of the polygon. Four points lie outside the polygon.

Input Arguments

xq — x-coordinates of query points
scalar | vector | matrix | multidimensional array

x-coordinates of query points, specified as a scalar, vector, matrix, or multidimensional array.

The size of `xq` must match the size of `yq`.

Data Types: `double` | `single`

yq — y-coordinates of query points
scalar | vector | matrix | multidimensional array

y-coordinates of query points, specified as a scalar, vector, matrix, or multidimensional array.

The size of `yq` must match the size of `xq`.

Data Types: `double` | `single`

xv — x-coordinates of polygon vertices
vector

x-coordinates of polygon vertices, specified as a vector.

The size of `xv` must match the size of `yv`.

To specify vertices of multiply connected or disjoint polygons, separate the coordinates for distinct loops with `NaN`. Additionally for multiply connected polygons, you must orient the vertices for external and internal loops in opposite directions.

The polygon cannot be self-intersecting and multiply connected due to the ambiguity associated with self-intersections and loop orientations.

Data Types: `double` | `single`

yv — y-coordinates of polygon vertices
vector

y-coordinates of polygon vertices, specified as a vector.

The size of `yv` must match the size of `xv`.

To specify vertices of multiply connected or disjoint polygons, separate the coordinates for distinct loops with `NaN`. Additionally for multiply connected polygons, you must orient the vertices for external and internal loops in opposite directions.

The polygon cannot be self-intersecting and multiply connected due to the ambiguity associated with self-intersections and loop orientations.

Data Types: `double` | `single`

Output Arguments

`in` — Indicator for points inside or on edge of polygon area

logical array

Indicator for the points inside or on the edge of the polygon area, returned as a logical array. `in` is the same size as `xq` and `yq`.

- A logical `1` (`true`) indicates that the corresponding query point is inside the polygonal region or on the edge of the polygon boundary.
- A logical `0` (`false`) indicates that the corresponding query point is outside the polygonal region.

Therefore, you can use `in` to index into `xq` and `yq` to identify query points of interest.

`xq(in)`, `yq(in)`	Query points inside or on the edge of the polygon area
`xq(~in)`, `yq(~in)`	Query points outside the polygonal region

`on` — Indicator for points on edge of polygon area

logical array

Indicator for the points on the edge of the polygon area, returned as a logical array. `on` is the same size as `xq` and `yq`.

- A logical `1` (`true`) indicates that the corresponding query point is on the polygon boundary.
- A logical `0` (`false`) indicates that the corresponding query point is inside or outside the polygon boundary.

Therefore, you can use `on` and `in` to index into `xq` and `yq` identify query points of interest.

`xq(on)`, `yq(on)`	Query points on the polygon boundary
`xq(~on)`, `yq(~on)`	Query points inside or outside the polygon boundary
`xq(in&~on)`, `yq(in&~on)`	Query points strictly inside the polygonal region

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supports single-precision and double-precision inputs, but uses double-precision arithmetic even if all inputs are single-precision.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`delaunay`

Introduced before R2006a

input

Request user input

Syntax

```
x = input(prompt)
str = input(prompt,'s')
```

Description

`x = input(prompt)` displays the text in `prompt` and waits for the user to input a value and press the **Return** key. The user can enter expressions, like `pi/4` or `rand(3)`, and can use variables in the workspace.

- If the user presses the **Return** key without entering anything, then `input` returns an empty matrix.
- If the user enters an invalid expression at the prompt, then MATLAB displays the relevant error message, and then redisplays the prompt.

`str = input(prompt,'s')` returns the entered text, without evaluating the input as an expression.

Examples

Request Numeric Input or Expression

Request a numeric input, and then multiply the input by 10.

```
prompt = 'What is the original value? ';
x = input(prompt)
y = x*10
```

At the prompt, enter a numeric value or array, such as 42.

```
x =
    42

y =
   420
```

The `input` function also accepts expressions. For example, rerun the code.

```
prompt = 'What is the original value? ';
x = input(prompt)
y = x*10
```

At the prompt, enter `magic(3)`.

```
x =
     8     1     6
     3     5     7
     4     9     2

y =
    80    10    60
    30    50    70
    40    90    20
```

Request Unprocessed Text Input

Request a simple text response that requires no evaluation.

```
prompt = 'Do you want more? Y/N [Y]: ';
str = input(prompt,'s');
if isempty(str)
    str = 'Y';
end
```

The `input` function returns the text exactly as typed. If the input is empty, this code assigns a default value, `'Y'`, to `str`.

Input Arguments

`prompt` — Text displayed to user
character vector

Text displayed to the user, specified as a character vector.

To create a prompt that spans several lines, use `'\n'` to indicate each new line. To include a backslash (`'\'`) in the prompt, use `'\\'`.

Output Arguments

x — Result calculated from input
array

Result calculated from input, returned as an array. The type and dimensions of the array depend upon the response to the prompt.

`str` — Exact text of input
character vector

Exact text of the input, returned as a character vector.

Algorithms

The Workspace browser does not refresh while `input` is waiting for a response from the user. Therefore, if you run `input` within a script, the Workspace browser does not display changes made to variables in the workspace until the script finishes running.

See Also

`ginput` | `inputdlg` | `keyboard` | `menu` | `uicontrol`

Introduced before R2006a

inputdlg

Create dialog box to gather user input

Syntax

```
answer = inputdlg(prompt)
answer = inputdlg(prompt,dlgtitle)
answer = inputdlg(prompt,dlgtitle,dims)
answer = inputdlg(prompt,dlgtitle,dims,definput)
answer = inputdlg(prompt,dlgtitle,dims,definput,opts)
```

Description

`answer = inputdlg(prompt)` creates a modal on page 1-7712 dialog box containing one or more text edit fields and returns the values entered by the user. The return values are elements of a cell array of character vectors. The first element of the cell array corresponds to the response in the edit field at the top of the dialog box. The second element corresponds to the next edit field response, and so on.

`answer = inputdlg(prompt,dlgtitle)` specifies a title for the dialog box.

`answer = inputdlg(prompt,dlgtitle,dims)` specifies the height of each edit field when `dims` is a scalar value. When `dims` is an array, the first value in each array element sets the edit field height. The second value in each array element sets the edit field width.

`answer = inputdlg(prompt,dlgtitle,dims,definput)` specifies the default value for each edit field. The `definput` input argument must contain the same number of elements as `prompt`.

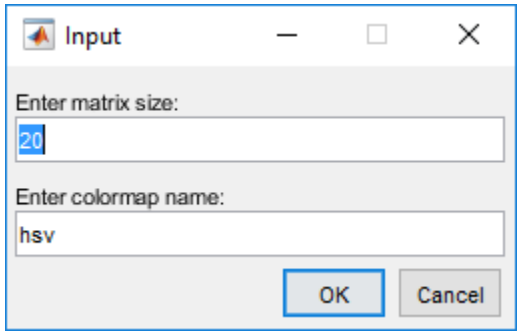
`answer = inputdlg(prompt,dlgtitle,dims,definput,opts)` specifies that the dialog box is resizeable in the horizontal direction when `opts` is set to `'on'`. When `opts` is a structure, it specifies whether the dialog box is resizeable in the horizontal direction, whether it is modal, and whether the `prompt` text is interpreted.

Examples

Dialog Box to Get User Input

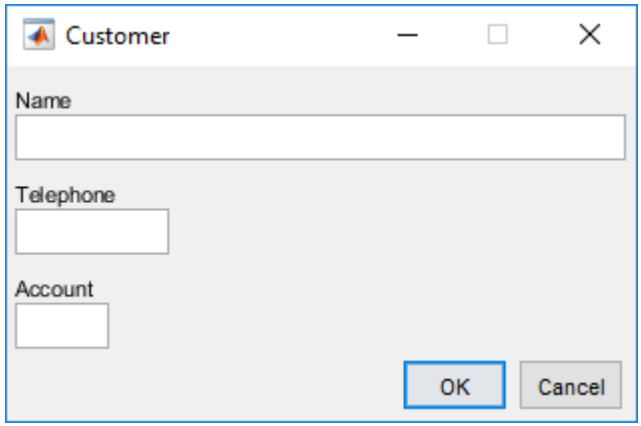
Create a dialog box that contains two text edit fields to get integer and colormap name inputs from users.

```
prompt = {'Enter matrix size:','Enter colormap name:'};
dlgtitle = 'Input';
dims = [1 35];
definput = {'20','hsv'};
answer = inputdlg(prompt,dlgtitle,dims,definput)
```



Text Edit Fields of Different Widths

```
x = inputdlg({'Name','Telephone','Account'},...
               'Customer', [1 50; 1 12; 1 7]);
```

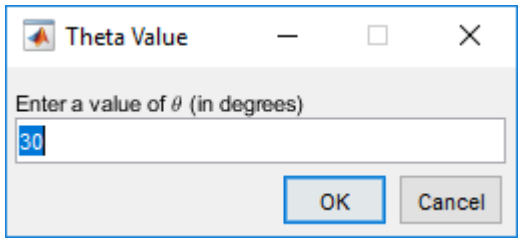


Interpret `prompt` Value with TeX Interpreter

Create a dialog box, specifying a value in the options structure.

Use the options structure to specify TeX to be the interpreter.

```
prompt = {'Enter a value of \theta (in degrees)'};
dlgtitle = 'Theta Value';
definput = {'30'};
opts.Interpreter = 'tex';
answer = inputdlg(prompt,dlgtitle,[1 40],definput,opts);
```

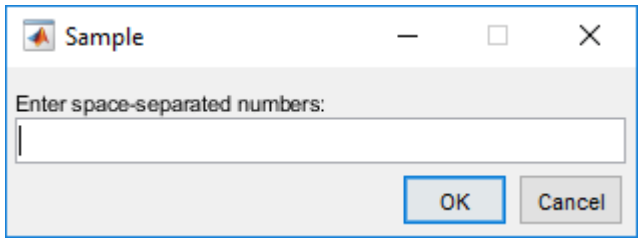


Convert Input to Numeric Values

Users can enter scalar or vector values into `inputdlg` text edit fields. MATLAB stores the input as a cell array of character vectors. Convert a member of the input cell array to a number, using `str2num`.

Create an input dialog box that asks users to enter numerical data.

```
answer = inputdlg('Enter space-separated numbers:',...
             'Sample', [1 50])
```



Suppose that the user enters `1 2 3 4 -5 6+7`, and then clicks **OK**. MATLAB stores the answer as a cell array of character vectors – `{'1 2 3 4 -5 6+7'}`

Use `str2num` to convert the cell array to numerals.

```
user_val = str2num(answer{1})
```

```
user_val =

     1     2     3     4    -5    13
```

Input Arguments

`prompt` — Text edit field labels
`'Input:'` (default) | character vector | cell array of character vectors | string array

Text edit field labels, specified as a character vector, cell array of character vectors, or string array. For cell arrays and string arrays, each element specifies an edit field label. Both types of arrays specify a top to bottom sequence of edit fields in the dialog box.

Example: `{'First Name','Last Name'}`

`dlgtitle` — Dialog box title

`''` (default) | character vector | string scalar

Dialog box title, specified as a character vector or string scalar.

Example: `'Data'`

`dims` — Text edit field height and width

1 (default) | scalar | column vector | row vector | m x 2 array

Text edit field height and width (dimensions), specified as one of these values:

- If `dims` is a scalar, then it specifies the height for all edit fields. Height is the distance between the baselines of two lines of text. The width for all edit fields is the maximum that the dialog box allows.
- If `dims` is a column vector or row vector, then each element specifies the height of each corresponding edit field from the top to the bottom of the dialog box. The width for all edit fields is the maximum that the dialog box allows.
- If `dims` is an array, then it must be size m-by-2, where m is the number of prompts in the dialog box. Each row refers to the edit field for the corresponding prompt. The first column specifies the height of the edit field in lines of text. The second column specifies the width of that edit field in character units. The width of a character unit equals the width of the letter `x` when using the system font.

Note With one exception, the height and width of the edit fields do not limit the amount of text that users can enter. When the edit field height is 1, users cannot enter multiple lines of text.

Example: 2

Example: `[1;2]`

Example: `[1 50; 2 10]`

`definput` — Default input value or values

`{''}` (default) | cell array of character vectors | string array

Default input value or values, specified as a cell array of character vectors or string array.

Example: `{'Color','1'}`

opts — Dialog box settings
`'on'` | structure

Dialog box settings specified as `'on'` or a structure. When `opts` is set to `'on'`, then users can resize the dialog box in the horizontal direction. When `opts` is a structure, the structure fields specify the options in the following table.

Field	Values
`Resize`	`'off'` (default) or `'on'`. If set to `'off'`, then users cannot resize the dialog box. If `'on'`, then users can resize the window horizontally.
`WindowStyle`	`'modal'` (default) or `'normal'`. If set to `'modal'`, then users cannot interact with other windows before responding.
`Interpreter`	`'none'` (default) or `'tex'`. If set to `'tex'`, then the prompt is rendered using TeX. The dialog box title is not affected. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the `prompt` text.

Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they modify only the next character or the characters within the curly braces. When you set the `Interpreter` property to `'tex'`, the supported modifiers are as follows.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{`*`specifier`*`}`	Font name — Replace *`specifier`* with the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`

Modifier	Description	Example
`\fontsize{`*`specifier`*`}`	Font size —Replace *`specifier`* with a numeric scalar value in point units.	`'\fontsize{15} text'`
`\color{`*`specifier`*`}`	Font color — Replace *`specifier`* with one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Replace *`specifier`* with a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	β	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ϵ	`\Delta`	Δ	`\spadesuit`	♠
`\zeta`	ζ	`\Theta`	Θ	`\leftrightarrow`	↔
`\eta`	η	`\Lambda`	Λ	`\leftarrow`	←
`\theta`	θ	`\Xi`	Ξ	`\Leftarrow`	⇐
`\vartheta`	ϑ	`\Pi`	Π	`\uparrow`	↑

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	μ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

Example: `'on'`

Example: `opts.Resize = 'on'`

Example: `opts.WindowStyle = 'normal'`

Example: `opts.Interpreter = 'tex'`

Output Arguments

`answer` — Output
empty cell array | cell array of character vectors

Returns a cell array of character vectors containing one input per edit field, starting from the top of the dialog box. Use the `str2num` function to convert space-delimited and comma-delimited values into row vectors, and semicolon-delimited values into column vectors. For an example, see Convert Input to Numeric Values on page 1-7706.

If the user clicks the **Cancel** button to close the dialog box, then `answer` is an empty cell array, `{ }`.

If the user presses the keyboard **Return** key to close the dialog box, then `answer` is the value of `definput`. If `definput` is undefined, then `answer` is an empty cell array, `{ }`.

More About

Modal Dialog Box

A modal dialog box prevents a user from interacting with other MATLAB windows before responding to the modal dialog box.

Tips

- MATLAB program execution continues even when a modal input dialog box is active. To block program execution until the user responds, use the `uiwait` function.

Users can enter scalar or vector values into Input dialog boxes. Use `str2num` to convert space-delimited and comma-delimited values into row vectors and to convert semicolon-delimited values into column vectors. For example, if `answer{1}` contains `'1 2 3;4 -5 6+7i'`, the conversion produces:

```
input = str2num(answer{1})
input =
   1.0000     2.0000     3.0000
   4.0000    -5.0000     6.0000 + 7.0000i
```

See Also

listdlg | questdlg

Introduced before R2006a

inputname

Variable name of function input

Syntax

```
s = inputname(argNumber)
```

Description

`s = inputname(argNumber)` returns the workspace variable name, `s`, corresponding to the argument number `argNumber`.

You cannot call `inputname` from the MATLAB command prompt or in a script you call from the command prompt.

Examples

Display Variable Name of Function Input

Create the following function in a file, `myfun.m`, in your current working folder.

```
function myfun(a,b)
s = inputname(1);
disp(['First calling variable is ''' s '''.'])
end
```

Call the function at the command prompt using the variables `x` and `y`.

```
x = 5;
y = 3;
myfun(x,y)
```

```
First calling variable is 'x'.
```

Call the function using values instead of variables. The `inputname` function returns an empty `char` array because its input does not have a name.

```
myfun(5,3)
```

```
First calling variable is ''.
```

Display All Calling Variable Names

Create the following function in a file, `myfun2.m`, in your current working folder.

```
function myfun2(a,b,c)
for m = 1:nargin
    disp(['Calling variable ' num2str(m) ' is ''' inputname(m) '''.'])
end
```

Call the function at the command prompt.

```
x = {'hello','goodbye'};
y = struct('a',42,'b',78);
z = rand(4);
```

```
myfun2(x,y,z)
```

```
Calling variable 1 is 'x'.
Calling variable 2 is 'y'.
Calling variable 3 is 'z'.
```

Call the function using a field of `y`. Because the input argument contains dot indexing, the `inputname` function returns an empty `char` array for the second variable name and all subsequent variable names.

```
myfun2(x,y.a,z)
```

```
Calling variable 1 is 'x'.
Calling variable 2 is ''.
Calling variable 3 is ''.
```

Call the function using the second cell of `x`. Because the input argument contains cell indexing, the `inputname` function returns an empty `char` array for the first variable name and all subsequent variable names.

```
myfun2(x{2},y,z)
```

```
Calling variable 1 is ''.
Calling variable 2 is ''.
Calling variable 3 is ''.
```

Input Arguments

`argNumber` — Number of function input argument
scalar, real, positive integer value

Number of function input argument, specified as a scalar, real, positive integer value. If `argNumber` exceeds the number of input arguments passed into the function, MATLAB throws an error.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`s` — Workspace variable name
character vector

Workspace variable name, returned as a character vector. If the input argument has no name, the `inputname` function returns an empty character array (`''`). For example, an input argument has no name if it is a number, an expression, or an indexing expression instead of a variable.

If an input argument to the function that calls `inputname` contains cell or dot indexing, `inputname` returns an empty character array for that variable and subsequent variables. Extracting elements from a cell array or a field from a structure yields a comma-separated list. A comma-separated list causes the location of any input that follows to be dynamic. Therefore, `inputname` returns `''` for the argument that produced the list and the arguments that follow.

Tips

- `inputname` is a convenient way to communicate the name of an input variable to a function. For example, a function checks the data types of inputs and, if it finds an incorrect type, displays the name of the variable from your workspace.
- Avoid using `inputname` in the critical path of code or to obtain variable names to be used with commands such as `eval`, `evalin`, and `assignin`. This use of `inputname` can lead to code that is difficult to maintain.
- `inputname` returns an error if it is called inside an overloaded `subsref`, `subsasgn`, `subsindex`, `numel`, or property set or get method.
- If the function that calls `inputname` is not called from a MATLAB code file, `inputname` walks up the stack until it finds a call from MATLAB code and returns the names it finds there. For example, this behavior occurs if `inputname` is called from a built-in function or a MEX function.

  Consider the following code in which the built-in `arrayfun` function calls `inputname` via a function handle.

  ```
  fn=@(x) inputname(x);
  a=1:4;
  arrayfun(fn,a,'uniformoutput',false)
  ```

  ```
  ans =

      'fn'    'a'    ''    ''
  ```

  The `inputname` function walks up the stack until it finds a call from MATLAB code. In this case, the MATLAB code is the base workspace, and `inputname` returns variable names from the base workspace

See Also

`nargin` | `narginchk` | `nargout`

Introduced before R2006a

inputParser

Input parser for functions

Description

The `inputParser` object enables you to manage inputs to a function by creating an input parser scheme. To check the input, you can define validation functions for required arguments, optional arguments, and name-value pair arguments. Optionally, you can set properties to adjust the parsing behavior, such as handling case sensitivity, structure array inputs, and inputs that are not in the input parser scheme.

After defining your input parser scheme, call the `parse` function. The `inputParser` stores information about inputs.

Input Names and Values	Where Stored
Matching input parser scheme	`Results` property
Not passed to function and, therefore, assigned default values	`UsingDefaults` property
No matching input parser scheme	`Unmatched` property

Creation

Syntax

```
p = inputParser
```

Description

`p = inputParser` creates an input parser object with default property values.

Properties

`CaseSensitive` — Indicator to match case

`false` (default) | `true`

Indicator to match case when checking argument names, specified as `false` or `true` (or 0 or 1). By default, argument name matches are not case sensitive. For example, `'a'` matches `'A'`. For case-sensitive matches, set `CaseSensitive` to `true` (or 1).

This property value is stored as a logical value.

`FunctionName` — Name of function for error message

empty character vector, `''`. (default) | character vector | string scalar

Name of the function to display in error messages, specified as a character vector or string scalar. By default, `FunctionName` is an empty character vector (`''`). Typically, you set `FunctionName` to the name of the function you are validating. Then, if the `parse` function encounters invalid input arguments, it reports the error using the function name.

This property value is stored as a character vector.

Data Types: `char` | `string`

`KeepUnmatched` — Matching indicator

`false` (default) | `true`

Matching indicator to throw error when an input is not found in the input parser scheme, specified as `false` or `true` (or 0 or 1). By default, the `parse` function throws an error if an input argument name does not match one defined in the input parser scheme. To suppress the error and store the input argument name and value, set `KeepUnmatched` to `true` (or 1). The `inputParser` stores unmatched input argument names and values in the `Unmatched` property.

This property value is stored as a logical value.

`PartialMatching` — Partial matching indicator

`true` (default) | `false`

Partial matching indicator for accepting partially matched input names as valid, specified as `true` or `false` (or 1 or 0). By default, input parameter names that are leading substrings of parameter names in the input parser scheme are valid and the input value is matched to that parameter. If there are multiple possible matches to the input parameter,

MATLAB throws an error. To require input parameter names to match a name in the input parser scheme exactly, respecting the `CaseSensitive` property, set `PartialMatching` to `false` (or 0).

Partial matching is supported only by arguments that you add to the input parser scheme using the `addParameter` function.

- If the value of the `StructExpand` property is `true` (or 1), then `inputParser` does not support partial matching for structure field names that correspond to input parameter names.
- If `PartialMatching` and `KeepUnmatched` are both `true` (or 1), then MATLAB does not throw an error. Instead, it stores the ambiguous parameter name in the `Unmatched` property.

This property value is stored as a logical value.

`StructExpand` — Structure indicator
`true` (default) | `false`

Structure indicator that interprets a structure as a single input or as a set of parameter name-value pairs, specified as `true` or `false` (or 1 or 0). By default, the `inputParser` expands structures into separate inputs, where each field name corresponds to an input parameter name. To consider structures as a single input argument, specify `StructExpand` as `false` (or 0).

This property value is stored as a logical value.

`Parameters` — Argument names
cell array of character vectors

This property is read-only.

Argument names defined in the input parser scheme, stored as a cell array of character vectors. Each function that adds an input argument to the scheme updates the `Parameters` property. These functions include `addRequired`, `addOptional`, and `addParameter`.

Data Types: `cell`

`Results` — Results
structure

This property is read-only.

Results specified as the names of valid input arguments and the corresponding values, stored as a structure. A valid input argument is one with a name that matches an argument defined in the input parser scheme. Each field of the `Results` structure corresponds to the name of an argument in the input parser scheme. The `parse` function populates the `Results` property.

Data Types: `struct`

`Unmatched` — Unmatched input
structure

This property is read-only.

Unmatched input names and values of inputs that do not match input parser scheme, stored as a structure. If the `KeepUnmatched` property is set to `false` (or 0), which is the default, or if all inputs match the input parser scheme, then `Unmatched` is a 1-by-1 structure with no fields. Otherwise, each field of the `Unmatched` structure corresponds to the name of an input argument that does not match the arguments defined in the input parser scheme.

The `parse` function populates the `Unmatched` property.

Data Types: `struct`

`UsingDefaults` — Inputs not passed explicitly to function
cell array of character vectors

This property is read-only.

Inputs not passed explicitly to the function, stored as a cell array of character vectors. These input arguments are assigned default values in the `Results` property. The `parse` function populates the `UsingDefaults` property.

Data Types: `cell`

Object Functions

addOptional	Add optional, positional argument into input parser scheme
addParameter	Add optional name-value pair argument into input parser scheme
addRequired	Add required, positional argument into input parser scheme
parse	Parse function inputs

addParamValue (Not recommended) Add optional name-value pair argument into input parser scheme

You can define your input parser scheme by calling the `addRequired`, `addOptional`, and `addParameter` functions in any order. However, when you call the function that uses the input parser, arguments are passed in this order:

1. Required arguments
2. Any optional, positional arguments
3. Any name-value pairs

Examples

Input Validation

Check the validity of required and optional arguments.

Create a function in the file `findArea.m`. The `findArea` function requires the `width` input argument and accepts a variable number of additional inputs. The input parser scheme specifies these argument conditions:

- `width` (required argument). Since required arguments are positional, `width` must be the first argument to the `findArea` function. The input parser checks that `width` is positive, scalar, and numeric.
- `height` (optional argument). Since optional arguments are positional, if `height` is an argument to the `findArea` function, then it must be the second argument. The input parser checks that `height` is positive, scalar, and numeric.
- `'units'` and its associated value (name-value pair). Name-value pairs are optional. When you call the `findArea` function, specify name-value pairs in any order after positional arguments. The input parser checks that the value for `'units'` is a string.
- `'shape'` and its associated value (another name-value pair). The input parser checks that the value for `'shape'` is contained in the `expectedShapes` array.

```
function a = findArea(width,varargin)
   defaultHeight = 1;
   defaultUnits = 'inches';
   defaultShape = 'rectangle';
   expectedShapes = {'square','rectangle','parallelogram'};
```

```
    p = inputParser;
    validScalarPosNum = @(x) isnumeric(x) && isscalar(x) && (x > 0);
    addRequired(p,'width',validScalarPosNum);
    addOptional(p,'height',defaultHeight,validScalarPosNum);
    addParameter(p,'units',defaultUnits,@isstring);
    addParameter(p,'shape',defaultShape,...
                  @(x) any(validatestring(x,expectedShapes)));
    parse(p,width,varargin{:});

    a = p.Results.width*p.Results.height;
end
```

Call the `findArea` function several times. The input parser does not throw an error for any of these function calls.

```
a = findArea(7);
a = findArea(7,3);
a = findArea(13,'shape','square');
a = findArea(13,'units',"miles",'shape','square');
```

Call the function with arguments that do not match the input parser scheme. Specify a nonnumeric value for the `width` input:

```
a = findArea('text')
```

```
Error using findArea (line 14)
The value of 'width' is invalid. It must satisfy the function: @(x)isnumeric(x)&&issca
```

Specify an unsupported value for `'shape'`.

```
a = findArea(4,12,'shape','circle')
```

```
Error using findArea (line 14)
The value of 'shape' is invalid. Expected input to match one of these values:

'square', 'rectangle', 'parallelogram'

The input, 'circle', did not match any of the valid values.
```

Extra Parameter Value Inputs

Store parameter name and value inputs that are not in the input scheme instead of throwing an error.

```
default = 0;
value = 1;
```

```
p = inputParser;
p.KeepUnmatched = true;
addOptional(p,'expectedInputName',default)
parse(p,'extraInput',value);
```

View the unmatched parameter name and value:

```
p.Unmatched
```

```
ans = struct with fields:
    extraInput: 1
```

Case Sensitivity

Enforce case sensitivity when checking function inputs.

```
p = inputParser;
p.CaseSensitive = true;
defaultValue = 0;
addParameter(p,'InputName',defaultValue)

parse(p,'inputname',10)
```

```
'inputname' is not a recognized parameter. For a list of valid name-value pair arguments, see the documentat
```

Structure Array Inputs

Expand a structure argument into name-value pairs.

```
s.input1 = 10;
s.input2 = 20;
default = 0;

p = inputParser;
addParameter(p,'input1',default)
addParameter(p,'input2',default)
parse(p,s)

p.Results
```

```
ans = struct with fields:
    input1: 10
```

```
    input2: 20
```

Accept a structure as a single argument by setting the `StructExpand` property to `false`.

```
s2.first = 1;
s2.random = rand(3,4,2);
s2.mytext = 'some text';

p = inputParser;
p.StructExpand = false;
addRequired(p,'structInput')
parse(p,s2)

results = p.Results
```

```
results = struct with fields:
    structInput: [1x1 struct]
```

```
fieldList = fieldnames(p.Results.structInput)
```

```
fieldList = 3x1 cell array
    {'first' }
    {'random'}
    {'mytext'}
```

Parse Inputs Using `validateattributes`

Create a function that parses information about people and, if parsing passes, adds the information to a cell array.

Create function `addPerson` and include an input parser scheme that uses the `validateattributes` function. The `addPerson` function accepts a list of people, modifies the list if necessary, and then returns the list. Use a persistent `inputParser` object to avoid constructing of a new object with every function call.

```
function mlist = addPerson(mlist,varargin)
    persistent p
    if isempty(p)
        p = inputParser;
        p.FunctionName = 'addPerson';
        addRequired(p,'name',@(x)validateattributes(x,{'char'},...
            {'nonempty'}))
```

```
        addRequired(p,'id',@(x)validateattributes(x,{'numeric'},...
            {'nonempty','integer','positive'}))
        addOptional(p,'birthyear',9999,@(x)validateattributes(x,...
            {'numeric'},{'nonempty'}))
        addParameter(p,'nickname','-',@(x)validateattributes(x,...
            {'char'},{'nonempty'}))
        addParameter(p,'favColor','-',@(x)validateattributes(x,...
            {'char'},{'nonempty'}))
    end

    parse(p,varargin{:})

    if isempty(mlist)
        mlist = fieldnames(p.Results)';
    end
    mlist = [mlist; struct2cell(p.Results)'];
end
```

Create an empty list, and add a person to it.

```
pList = {};
pList = addPerson(pList,78,'Joe');
```

```
Error using addPerson
The value of 'name' is invalid. Expected input to be one of these types:

char

Instead its type was double.

Error in addPerson (line 19)
parse(p,varargin{:})
```

The parsing fails because the function receives arguments in the incorrect order and tries to assign `name` a value of `78`. This entry is not added to `pList`.

Add several more people to the list.

```
pList = addPerson(pList,'Joe',78);
pList = addPerson(pList,'Mary',3,1942,'favColor','red');
pList = addPerson(pList,'James',182,1970,'nickname','Jimmy')
```

```
pList =

  4×5 cell array

    'birthyear'    'favColor'    'id'     'name'     'nickname'
    [     9999]    '-'           [ 78]    'Joe'      '-'
    [     1942]    'red'         [  3]    'Mary'     '-'
    [     1970]    '-'           [182]    'James'    'Jimmy'
```

Tips

- Arguments added to the input parser scheme with the `addOptional` function are positional. Therefore, add them to the input parser scheme in the same order they are passed into the function.

- Use `addOptional` to add an individual argument into the input parser scheme. If you want to parse an optional name-value pair, then use the `addParameter` function.

See Also

`arguments` | `validateattributes` | `validatestring` | `varargin`

Topics

"Input Parser Validation Functions"
"Function Argument Validation"

Introduced in R2007a

addOptional

Add optional, positional argument into input parser scheme

Syntax

```
addOptional(p,argName,defaultVal)
addOptional(p,argName,defaultVal,validationFcn)
```

Description

`addOptional(p,argName,defaultVal)` adds an optional, positional input argument, `argName`, into the input parser scheme `p`. When the inputs to a function do not include a value for this optional input, the input parser assigns it the value `defaultVal`.

`addOptional(p,argName,defaultVal,validationFcn)` specifies a validation function for the input argument.

Examples

Add Optional Input

Create an `inputParser` object and add an optional input to the input parser scheme. Name the argument `myinput`, and assign it a default value of 13.

```
p = inputParser;
argName = 'myInput';
defaultVal = 13;
addOptional(p,argName,defaultVal)
```

Call the parse function with no inputs, and display the results.

```
parse(p)
p.Results
```

```
ans = struct with fields:
    myInput: 13
```

Call the `parse` function with an input value of 42, and display the results.

```
parse(p,42)
p.Results
```

```
ans = struct with fields:
    myInput: 42
```

Validate Optional Input

Validate that an optional input named `num`, with a default value of 1, is a numeric scalar greater than zero.

Create an input parser scheme. For the validation function, `@(x)` creates a handle to an anonymous function that accepts one input.

```
p = inputParser;
argName = 'num';
defaultVal = 1;
validationFcn = @(x) isnumeric(x) && isscalar(x) && (x > 0);
addOptional(p,argName,defaultVal,validationFcn)
```

Parse an invalid input argument, such as `-1`.

```
parse(p,-1)
```

```
The value of 'num' is invalid. It must satisfy the function: @(x)isnumeric(x)&&isscalar(x)&&(x>0).
```

Validate Optional Input with `validateattributes`

Define a validation function using `validateattributes`. Validate that an argument is numeric, positive, and even.

```
validationFcn = @(x) validateattributes(x,{'numeric'},...
    {'even','positive'});
```

Create an input parser scheme that includes an optional `evenPosNum` argument with a default value of 1. Validate the input argument with `validationFcn`.

```
p = inputParser;
argName = 'evenPosNum';
defaultVal = 1;
addOptional(p,argName,defaultVal,validationFcn)
```

Parse an input string. Parse fails.

```
parse(p,"hello")
```

```
The value of 'evenPosNum' is invalid. Expected input to be one of these types:

numeric

Instead its type was string.
```

Parse an odd number. Parse fails.

```
 parse(p,13)
```

```
The value of 'evenPosNum' is invalid. Expected input to be even.
```

Parse an even, positive number. Parse passes.

```
parse(p,42)
```

Input Arguments

p — Input parser scheme
`inputParser` object

Input parser scheme, specified as an `inputParser` object.

`argName` — Name of input argument
character vector | string scalar

Name of the input argument, specified as a character vector or string scalar.

Example: `'firstName'`

Example: `'address'`

Data Types: `char` | `string`

`defaultVal` — Default value for input
value

Default value for the input, specified as any data type. If `argName` is not an input to the function, when the `parse` function parses the inputs, then it assigns `argName` the value `defaultVal`.

`validationFcn` — Function to validate argument
function handle

Function to validate an argument, specified as a function handle.

The function handle must be associated with a function that returns `true` or `false`, or passes a test, or throws an error. Both types of functions must accept a single input argument.

Example: `@(s)isstring(s)`

Example: `@(x)isnumeric(x)&&isscalar(x)`

Example: `@(n)validateattributes(n,{'numeric'},{'nonnegative'})`

Data Types: `function_handle`

Tips

- Arguments added to the input parser scheme with the `addOptional` function are positional. Therefore, add them to the input parser scheme in the same order they are passed into the function.
- For optional string arguments, specify a validation function. Without a validation function, the input parser interprets a string argument as an invalid parameter name and throws an error.
- Use `addOptional` to add an individual argument into the input parser scheme. If you want to parse an optional name-value pair, then use the `addParameter` function.

See Also

`addParameter` | `addRequired` | `inputParser` | `validateattributes`

Topics

"Input Parser Validation Functions"
"Anonymous Functions"

Introduced in R2007a

addParameter

Add optional name-value pair argument into input parser scheme

Syntax

```
addParameter(p,paramName,defaultVal)
addParameter(p,paramName,defaultVal,validationFcn)
addParameter( ___ ,'PartialMatchPriority',matchPriorityValue)
```

Description

`addParameter(p,paramName,defaultVal)` adds the parameter name of an optional name-value pair argument into the input parser scheme. When the inputs to a function do not include this optional name-value pair, the input parser assigns `paramName` the value `defaultVal`.

Unlike positional inputs added with the `addRequired` and `addOptional` functions, each parameter added with `addParameter` corresponds to two input arguments: one for the name and one for the value.

`addParameter(p,paramName,defaultVal,validationFcn)` specifies a validation function for the input argument.

`addParameter( ___ ,'PartialMatchPriority',matchPriorityValue)` specifies the priority for the partial matching of conflicting parameter names. The input parser scheme selects lower priority values over higher ones. Use this option with any of the input argument combinations in the previous syntaxes.

Examples

Add Parameter Value Input

Create an `inputParser` object and add a name-value pair into the input scheme.

```
p = inputParser;
paramName = 'myParam';
defaultVal = 0;
addParameter(p,paramName,defaultVal)
```

Pass both the parameter name and value to the `parse` method, and display the results.

```
parse(p,'myParam',100);
p.Results
```

```
ans = struct with fields:
    myParam: 100
```

Validate Parameter Value

Validate that the value corresponding to `myParam`, with a default value of 1, is a numeric scalar greater than zero.

Create an input parser scheme. For the validation function, `@(x)` creates a handle to an anonymous function that accepts one input.

```
p = inputParser;
paramName = 'myParam';
defaultVal = 1;
errorMsg = 'Value must be positive, scalar, and numeric.';
validationFcn = @(x) assert(isnumeric(x) && isscalar(x) ...
    && (x > 0),errorMsg);
addParameter(p,paramName,defaultVal,validationFcn)
```

Parse an invalid input argument, such as `-1`.

```
parse(p,'myparam',-1)
```

```
The value of 'myparam' is invalid. Value must be positive, scalar, and numeric.
```

Validate Parameter Value Input Using `validateattributes`

Define a validation function using `validateattributes`. Validate that an argument is a nonempty character vector.

```
validationFcn = @(x) validateattributes(x,{'char'},{'nonempty'});
```

Create an input parser scheme that includes an optional name-value pair argument, with a parameter name `myName` and a default value of `'John Doe'`. Validate the input argument with `validationFcn`.

```
p = inputParser;
paramName = 'myName';
defaultVal = 'John Doe';
addParameter(p,paramName,defaultVal,validationFcn)
```

Define `myName` as a number. The parse fails.

```
parse(p,'myName',1138)
```

```
The value of 'myName' is invalid. Expected input to be one of these types:

char

Instead its type was double.
```

Parse a character vector. The parse passes.

```
parse(p,'myName','George')
```

Input Arguments

p — Input parser scheme
`inputParser` object

Input parser scheme, specified as an `inputParser` object.

`paramName` — Name of input parameter
character vector | string scalar

Name of the input parameter, specified as a character vector or string scalar.

Example: `"firstName"`

Example: `'address'`

Data Types: `char` | `string`

`defaultVal` — Default value for input
value

Default value for the input, specified as any data type. If `argName` is not an input to the function, when the `parse` function parses the inputs, then it assigns `argName` the value `defaultVal`.

`validationFcn` — Function to validate argument
function handle

Function to validate an argument, specified as a function handle.

The function handle must be associated with a function that returns `true` or `false`, or passes a test, or throws an error. Both types of functions must accept a single input argument.

Example: `@(s)isstring(s)`

Example: `@(x)isnumeric(x)&&isscalar(x)`

Example: `@(n)validateattributes(n,{'numeric'},{'nonnegative'})`

Data Types: `function_handle`

`matchPriorityValue` — Priority for partial matching of conflicting parameter names
1 (default) | positive integer

Priority for partial matching of conflicting parameter names, specified as a positive integer. The input parser scheme selects lower priority values over higher ones. If partial parameter names are ambiguous and have the same priority, then `parse` throws an error. If the names are ambiguous, but have different priority values, then `parse` issues a warning that indicates the matched name.

Tips

- Parameter name-value pairs are optional inputs. When calling the function, name-value pairs can appear in any order after positional arguments. They take the general form `Name1,Value1,...,NameN,ValueN`.

See Also

`addOptional` | `addRequired` | `inputParser` | `validateattributes`

Topics

"Input Parser Validation Functions"
"Anonymous Functions"

Introduced in R2013b

addParamValue

(Not recommended) Add optional name-value pair argument into input parser scheme

Note `addParamValue` is not recommended. Use `addParameter` instead.

Syntax

```
addParamValue(p,paramName,defaultVal)
addParamValue(p,paramName,defaultVal,validationFcn)
```

Description

`addParamValue(p,paramName,defaultVal)` adds the parameter name of an optional name-value pair argument into the input parser scheme. When the inputs to a function do not include this optional name-value pair, the input parser assigns `paramName` the value `defaultVal`.

Unlike positional inputs added with the `addRequired` and `addOptional` functions, each parameter added with `addParamValue` corresponds to two input arguments: one for the name and one for the value.

`addParamValue(p,paramName,defaultVal,validationFcn)` specifies a validation function for the input argument.

Examples

Validate Parameter Value

Validate that the value corresponding to `myParam`, with a default value of 1, is a numeric scalar greater than zero.

Create an input parser scheme. For the validation function, `@(x)` creates a handle to an anonymous function that accepts one input.

```
p = inputParser;
paramName = 'myParam';
defaultVal = 1;
errorMsg = 'Value must be positive, scalar, and numeric.';
validationFcn = @(x) assert(isnumeric(x) && isscalar(x) ...
    && (x > 0),errorMsg);
addParamValue(p,paramName,defaultVal,validationFcn)
```

Parse an invalid input argument, such as `-1`.

```
parse(p,'myparam',-1)
```

```
The value of 'myparam' is invalid. Value must be positive, scalar, and numeric.
```

Input Arguments

p — Input parser scheme
`inputParser` object

Input parser scheme, specified as an `inputParser` object.

`paramName` — Name of input parameter
character vector | string scalar

Name of the input parameter, specified as a character vector or string scalar.

Example: `"firstName"`

Example: `'address'`

Data Types: `char` | `string`

`defaultVal` — Default value for input
value

Default value for the input, specified as any data type. If `argName` is not an input to the function, when the `parse` function parses the inputs, then it assigns `argName` the value `defaultVal`.

`validationFcn` — Function to validate argument
function handle

Function to validate an argument, specified as a function handle.

The function handle must be associated with a function that returns `true` or `false`, or passes a test, or throws an error. Both types of functions must accept a single input argument.

Example: `@(s)isstring(s)`

Example: `@(x)isnumeric(x)&&isscalar(x)`

Example: `@(n)validateattributes(n,{'numeric'},{'nonnegative'})`

Data Types: `function_handle`

Tips

- Parameter name-value pairs are optional inputs. When calling the function, name-value pairs can appear in any order after positional arguments. They take the general form `Name1,Value1,...,NameN,ValueN`.

See Also

`addParameter` | `inputParser`

Topics

"Input Parser Validation Functions"
"Anonymous Functions"

Introduced in R2007a

addRequired

Add required, positional argument into input parser scheme

Syntax

```
addRequired(p,argName)
addRequired(p,argName,validationFcn)
```

Description

`addRequired(p,argName)` adds a required, positional input argument `argName` into the input parser scheme `p`.

`addRequired(p,argName,validationFcn)` specifies a validation function for the input argument.

Examples

Add Required Input

Create an `inputParser` object and add a required input named `myinput` to the input scheme.

```
p = inputParser;
argName = 'myinput';
addRequired(p,argName)
```

Call the `parse` function with the input value 7, and display the results.

```
parse(p,7)
p.Results
```

```
ans = struct with fields:
    myinput: 7
```

Validate Required Input Is Nonnegative

Create an input parser scheme that checks that a required input is a nonnegative, numeric scalar. The syntax `@(x)` creates a handle to an anonymous function with one input.

```
p = inputParser;
argName = 'num';
validationFcn = @(x) (x > 0) && isnumeric(x) && isscalar(x);
addRequired(p,argName,validationFcn)
```

Parse an invalid input, such as `-1`:

```
parse(p,-1)
```

```
The value of 'num' is invalid. It must satisfy the function: @(x)(x>0)&&isnumeric(x)&&isscalar(x).
```

Validate Required Input Using `validateattributes`

Create an `inputParser` object and define a validation function using `validateattributes`. The validation function tests that a required input is numeric, positive, and even.

```
p = inputParser;
argName = 'evenPosNum';
validationFcn = @(x) validateattributes(x,{'numeric'},...
    {'even','positive'});
addRequired(p,argName,validationFcn)
```

Parse an input character vector. Parse fails because the input is invalid.

```
parse(p,'hello')
```

```
The value of 'evenPosNum' is invalid. Expected input to be one of these types:

double, single, uint8, uint16, uint32, uint64, int8, int16, int32, int64

Instead its type was char.
```

Parse an odd number. Parse fails because the input is invalid.

```
 parse(p,13)
```

```
The value of 'evenPosNum' is invalid. Expected input to be even.
```

Parse an even, positive number. Parse passes.

```
parse(p,42)
```

Input Arguments

p — Input parser scheme
`inputParser` object

Input parser scheme, specified as an `inputParser` object.

`argName` — Name of input argument
character vector | string scalar

Name of the input argument, specified as a character vector or string scalar.

Example: `'firstName'`

Example: `'address'`

Data Types: `char` | `string`

`validationFcn` — Function to validate argument
function handle

Function to validate an argument, specified as a function handle.

The function handle must be associated with a function that returns `true` or `false`, or passes a test, or throws an error. Both types of functions must accept a single input argument.

Example: `@(s)isstring(s)`

Example: `@(x)isnumeric(x)&&isscalar(x)`

Example: `@(n)validateattributes(n,{'numeric'},{'nonnegative'})`

Data Types: `function_handle`

See Also

`addOptional` | `addParameter` | `addRequired` | `inputParser` | `validateattributes`

Topics

"Input Parser Validation Functions"
"Anonymous Functions"

Introduced in R2007a

parse

Parse function inputs

Syntax

```
parse(p,argList)
```

Description

`parse(p,argList)` parses and validates the inputs in `arglist`.

Examples

Validate Required Input Is Nonnegative

Create an input parser scheme that checks that a required input is a nonnegative, numeric scalar. The syntax `@(x)` creates a handle to an anonymous function with one input.

```
p = inputParser;
argName = 'num';
validationFcn = @(x) (x > 0) && isnumeric(x) && isscalar(x);
addRequired(p,argName,validationFcn)
```

Parse an invalid input, such as `-1`:

```
parse(p,-1)
```

```
The value of 'num' is invalid. It must satisfy the function: @(x)(x>0)&&isnumeric(x)&&isscalar(x).
```

Input Parsing

Parse and validate required and optional function inputs.

Create a function in the file `findArea.m`. The `findArea` function requires the `width` input argument and accepts a variable number of additional inputs. The input parser scheme specifies these argument conditions:

- `width` (required argument). Since required arguments are positional, `width` must be the first argument to the `findArea` function. The input parser checks that `width` is positive, scalar, and numeric.
- `height` (optional argument). Since optional arguments are positional, if `height` is an argument to the `findArea` function, then it must be the second argument. The input parser checks that `height` is positive, scalar, and numeric.
- `'units'` and its associated value (name-value pair). Name-value pairs are optional. When you call the `findArea` function, specify name-value pairs in any order after positional arguments. The input parser checks that the value for `'units'` is a string.
- `'shape'` and its associated value (another name-value pair). The input parser checks that the value for `'shape'` is contained in the `expectedShapes` array.

```
function a = findArea(width,varargin)
   defaultHeight = 1;
   defaultUnits = 'inches';
   defaultShape = 'rectangle';
   expectedShapes = {'square','rectangle','parallelogram'};

   p = inputParser;
   validScalarPosNum = @(x) isnumeric(x) && isscalar(x) && (x > 0);
   addRequired(p,'width',validScalarPosNum);
   addOptional(p,'height',defaultHeight,validScalarPosNum);
   addParameter(p,'units',defaultUnits,@isstring);
   addParameter(p,'shape',defaultShape,...
                 @(x) any(validatestring(x,expectedShapes)));
   parse(p,width,varargin{:});

   a = p.Results.width*p.Results.height;
end
```

Call the `findArea` function several times. The input parser does not throw an error for any of these function calls.

```
a = findArea(7);
a = findArea(7,3);
a = findArea(13,'shape','square');
a = findArea(13,'units',"miles",'shape','square');
```

Call the function with arguments that do not match the input parser scheme. Specify a nonnumeric value for the `width` input:

```
a = findArea('text')
```

```
Error using findArea (line 14)
The value of 'width' is invalid. It must satisfy the function: @(x)isnumeric(x)&&issca
```

Specify an unsupported value for `'shape'`.

```
a = findArea(4,12,'shape','circle')
```

```
Error using findArea (line 14)
The value of 'shape' is invalid. Expected input to match one of these values:

'square', 'rectangle', 'parallelogram'

The input, 'circle', did not match any of the valid values.
```

Input Arguments

p — Input parser scheme
`inputParser` object

Input parser scheme, specified as an `inputParser` object.

`argList` — Inputs to parse and validate
comma-separated list

Inputs to parse and validate, specified as a comma-separated list. The elements of `argList` can be any data type. The input parser determines argument validity using the validation function you specified when you added arguments to the input parser scheme.

Example: `'textA',13,mtxB`

Example: `varargin{:}`

See Also

`addOptional` | `addParameter` | `addRequired` | `inputParser`

Introduced in R2007a

insertAfter

Insert strings after specified substrings

Syntax

```
newStr = insertAfter(str,startStr,newText)
newStr = insertAfter(str,startPos,newText)
```

Description

`newStr = insertAfter(str,startStr,newText)` inserts `newText` into `str` after the substring specified by `startStr` and returns the result as `newStr`. If `startStr` occurs multiple times in `str`, then `insertAfter` inserts text after every occurrence of `startStr`.

If `str` is a string array or a cell array of character vectors, then `insertAfter` inserts `newText` into each element of `str`. The output argument `newStr` has the same data type as `str`.

`newStr = insertAfter(str,startPos,newText)` inserts the text specified by `newText` into `str` after the position specified by `startPos`.

Examples

Insert Text After Substring

Create string arrays and insert text after substrings.

Starting in R2017a, you can create strings using double quotes.

```
str = "The quick fox"
```

```
str =
"The quick fox"
```

Insert text after the substring `"quick"`.

```
newStr = insertAfter(str,"quick"," brown")
```

```
newStr =
"The quick brown fox"
```

Insert substrings into each element of a string array. When you specify different substrings as positions, they must be contained in a string array or a cell array that is the same size as `str`.

```
str = ["The quick fox jumps";"over the dog"]
```

```
str = 2x1 string array
    "The quick fox jumps"
    "over the dog"
```

```
newStr = insertAfter(str,["quick";"the"],[" brown";" lazy"])
```

```
newStr = 2x1 string array
    "The quick brown fox jumps"
    "over the lazy dog"
```

Insert Substring After Position

Create string arrays and specify positions to insert substrings.

Starting in R2017a, you can create strings using double quotes.

```
str = "James Maxwell"
```

```
str =
"James Maxwell"
```

Insert a substring after the fifth character.

```
newStr = insertAfter(str,5," Clerk")
```

```
newStr =
"James Clerk Maxwell"
```

Insert substrings into each element of a string array. When you specify different positions with numeric arrays, they must be the same size as the input string array.

```
str = ["James Maxwell";"Carl Gauss"]
```

```
str = 2x1 string array
    "James Maxwell"
    "Carl Gauss"
```

```
newStr = insertAfter(str,[5;4],[" Clerk";" Friedrich"])
```

```
newStr = 2x1 string array
    "James Clerk Maxwell"
    "Carl Friedrich Gauss"
```

Insert Text After Position in Character Vector

Create a character vector and insert text after a specified position.

```
chr = 'mushrooms and onions'
```

```
chr =
'mushrooms and onions'
```

Insert text after the ninth position.

```
newChr = insertAfter(chr,9,', peppers,')
```

```
newChr =
'mushrooms, peppers, and onions'
```

Insert text after a substring.

```
newChr = insertAfter(chr,'mushrooms',', peppers,')
```

```
newChr =
'mushrooms, peppers, and onions'
```

Input Arguments

`str` — Input text

string array | character vector | cell array of character vectors

Input text, specified as a string array, a character vector, or a cell array of character vectors.

Data Types: `string` | `char` | `cell`

`startStr` — Substring to insert text after

string array | character vector | cell array of character vectors

Substring to insert text after, specified as a string array, a character vector, or a cell array of character vectors.

If `str` is a string array or a cell array of character vectors, then `startStr` can be a character vector, a string scalar, or a string array or a cell array of the same size as `str`.

Data Types: `string` | `char` | `cell`

`startPos` — Start position to insert text after

numeric array

Start position to insert text after, specified as a numeric array.

If `str` is a string array or a cell array of character vectors, then `startPos` can be a numeric scalar or a numeric array of the same size as `str`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`newText` — Text to insert

string array | character vector | cell array of character vectors

Text to insert, specified as a string array, a character vector, or a cell array of character vectors.

If `str` is a string array or a cell array of character vectors, then `newText` can be a character vector, a string scalar, or a string array or a cell array of the same size as `str`.

Data Types: `string` | `char` | `cell`

Output Arguments

`newStr` — Output text
string array | character vector | cell array of character vectors

Output text, returned as a string array, a character vector, or a cell array of character vectors. `str` and `newStr` have the same data type.

Data Types: `string` | `char` | `cell`

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `str`, `startStr`, and `newText` must be a string scalar, a character vector, or a cell array containing not more than one character vector.

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`erase` | `eraseBetween` | `extractAfter` | `extractBefore` | `insertBefore` | `join` | `plus` | `replace` | `split`

Topics

"Create String Arrays"
"Search and Replace Text"
"Test for Empty Strings and Missing Values"

Introduced in R2016b

insertBefore

Insert strings before specified substrings

Syntax

```
newStr = insertBefore(str,endStr,newText)
newStr = insertBefore(str,endPos,newText)
```

Description

`newStr = insertBefore(str,endStr,newText)` inserts `newText` into `str` before the substring specified by `endStr` and returns the result as `newStr`. If `endStr` occurs multiple times in `str`, then `insertBefore` inserts text before every occurrence of `endStr`.

If `str` is a string array or a cell array of character vectors, then `insertBefore` inserts `newText` into each element of `str`. The output argument `newStr` has the same data type as `str`.

`newStr = insertBefore(str,endPos,newText)` inserts the text specified by `newText` into `str` before the position specified by `endPos`.

Examples

Insert Text Before Substring

Create string arrays and insert text before substrings.

Starting in R2017a, you can create strings using double quotes.

```
str = "bread cheese wine"
```

```
str =
"bread cheese wine"
```

Insert a comma before each space character in the string. The `insertBefore` function inserts text before each matching substring.

```
newStr = insertBefore(str," ",",")
```

```
newStr =
"bread, cheese, wine"
```

Insert substrings into each element of a string array. When you specify different substrings as positions, they must be contained in a string array or a cell array that is the same size as `str`.

```
str = ["The quick fox jumps";"over the dog"]
```

```
str = 2x1 string array
    "The quick fox jumps"
    "over the dog"
```

```
newStr = insertBefore(str,[" fox";" dog"],[" brown";" lazy"])
```

```
newStr = 2x1 string array
    "The quick brown fox jumps"
    "over the lazy dog"
```

Insert Substring Before Position

Create string arrays and specify positions to insert substrings.

Starting in R2017a, you can create strings using double quotes.

```
str = "James Maxwell"
```

```
str =
"James Maxwell"
```

Insert a substring before the seventh character.

```
newStr = insertBefore(str,7,"Clerk ")
```

```
newStr =
"James Clerk Maxwell"
```

Insert substrings into each element of a string array. When you specify different positions with numeric arrays, they must be the same size as the input string array.

```
str = ["James Maxwell";"Carl Gauss"]
```

```
str = 2x1 string array
    "James Maxwell"
    "Carl Gauss"
```

```
newStr = insertBefore(str,[7;6],["Clerk ";"Friedrich "])
```

```
newStr = 2x1 string array
    "James Clerk Maxwell"
    "Carl Friedrich Gauss"
```

Insert Text Before Position in Character Vector

Create a character vector and insert text before a specified position.

```
chr = 'mushrooms and onions'
```

```
chr =
'mushrooms and onions'
```

Insert text before the tenth position.

```
newChr = insertBefore(chr,10,', peppers,')
```

```
newChr =
'mushrooms, peppers, and onions'
```

Insert text before a substring.

```
newChr = insertBefore(chr,' and',', peppers,')
```

```
newChr =
'mushrooms, peppers, and onions'
```

Input Arguments

`str` — Input text

string array | character vector | cell array of character vectors

Input text, specified as a string array, a character vector, or a cell array of character vectors.

Data Types: `string` | `char` | `cell`

`endStr` — Substring to insert text before

string array | character vector | cell array of character vectors

Substring to insert text before, specified as a string array, a character vector, or a cell array of character vectors.

If `str` is a string array or a cell array of character vectors, then `endStr` can be a character vector, a string scalar, or a string array or a cell array of the same size as `str`.

Data Types: `string` | `char` | `cell`

`endPos` — End position to insert text before

numeric array

End position to insert text before, specified as a numeric array.

If `str` is a string array or a cell array of character vectors, then `endPos` can be a numeric scalar or a numeric array of the same size as `str`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`newText` — Text to insert

string array | character vector | cell array of character vectors

Text to insert, specified as a string array, a character vector, or a cell array of character vectors.

If `str` is a string array or a cell array of character vectors, then `newText` can be a character vector, a string scalar, or a string array or a cell array of the same size as `str`.

Data Types: `string` | `char` | `cell`

Output Arguments

`newStr` — Output text
string array | character vector | cell array of character vectors

Output text, returned as a string array, a character vector, or a cell array of character vectors. `str` and `newStr` have the same data type.

Data Types: `string` | `char` | `cell`

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `str`, `endStr`, and `newText` must be a string scalar, a character vector, or a cell array containing not more than one character vector.

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`erase` | `eraseBetween` | `extractAfter` | `extractBefore` | `insertAfter` | `join` | `split`

Topics

"Create String Arrays"
"Search and Replace Text"
"Test for Empty Strings and Missing Values"

Introduced in R2016b

Property Inspector

Open property inspector

Description

The Property Inspector enables you to interactively modify objects, such as graphics objects. When you select an object, the Property Inspector shows a list of object properties. You can change property values by editing the fields that appear next to each property name.

Open the Property Inspector

- Figure toolbar: Click the Property Inspector icon .
- MATLAB command prompt: Enter `inspect`.

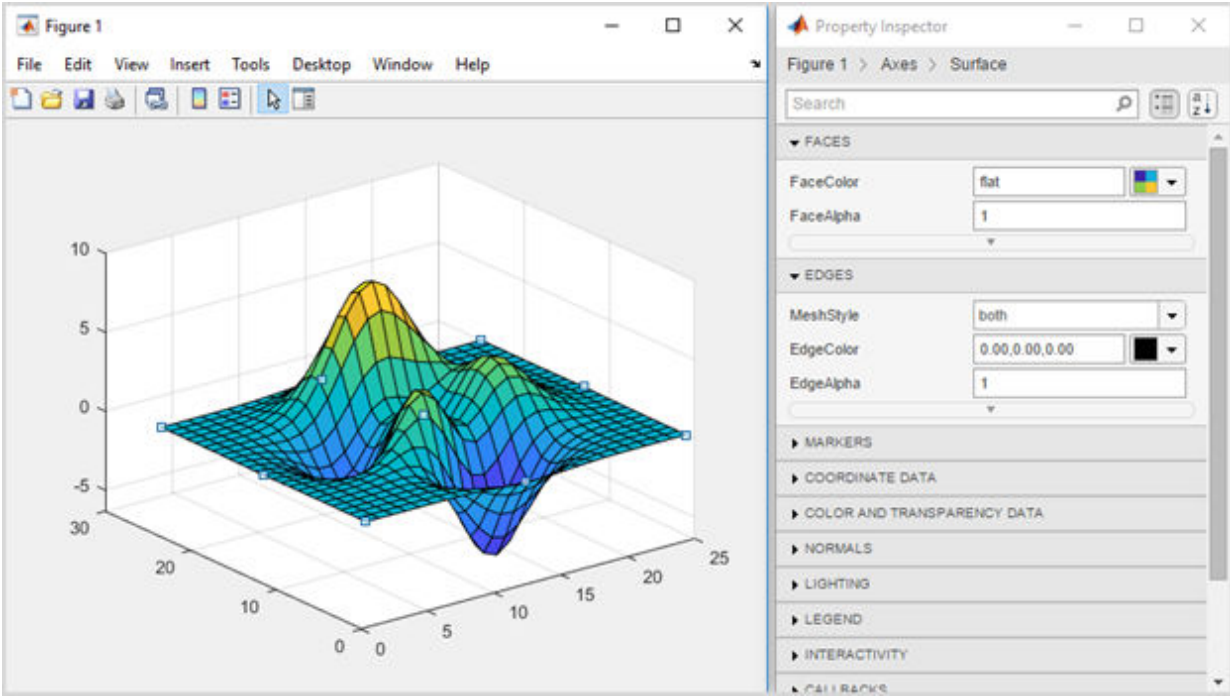
Examples

Graphics Objects

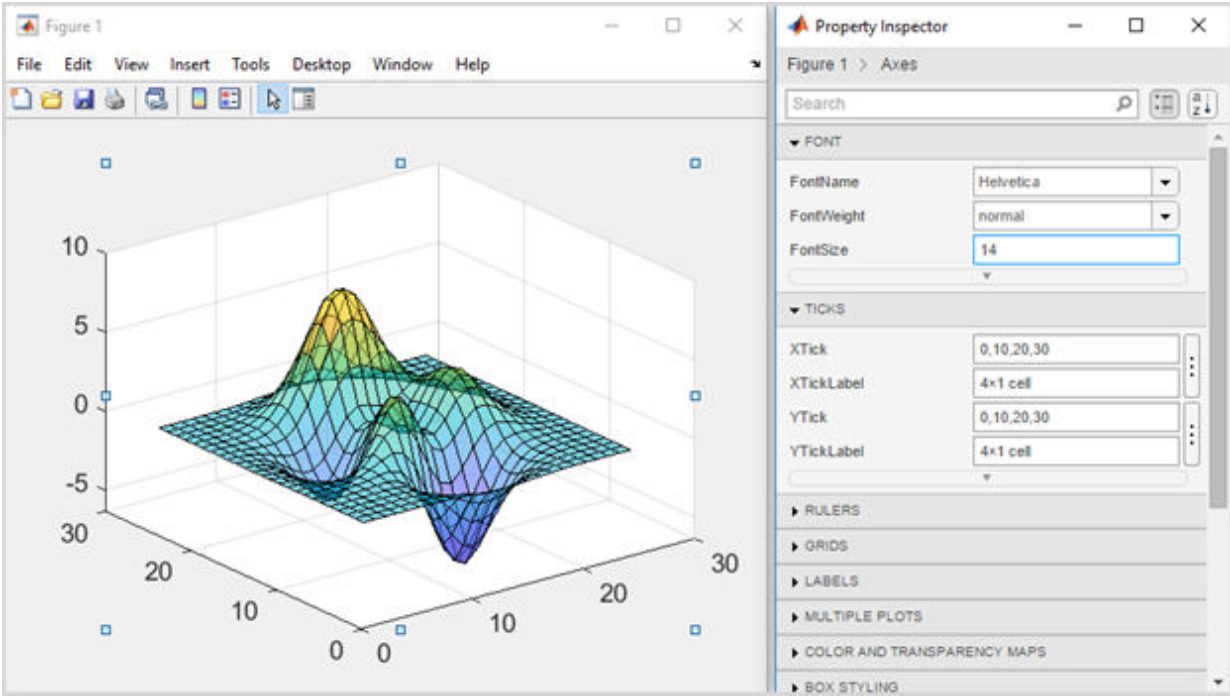
Use the Property Inspector to change the transparency of a surface plot and to change the font size of the axes text.

Create a surface plot. Open the Property Inspector by entering `inspect` at the command prompt. Then click the surface. The Property Inspector shows surface properties.

```
surf(peaks(25))
inspect
```



Change the `FaceAlpha` value from `1` to `0.5`. The surface plot updates so that it is semi-transparent. Then view axes properties by clicking the axes. Change the `FontSize` value to 14 and change the `FontWeight` value to `bold`.



Programmatic Use

`inspect` opens the Property Inspector for the current figure, if one exists. To view properties for a specific object, select the object. If no figures exist, then this command opens an empty Property Inspector.

`inspect(obj)` displays the properties for the specified object. You can specify `obj` as a single object or as a vector containing multiple objects, such as `[obj1 obj2]`. If you specify multiple objects, then the Property Inspector displays the properties that the objects have in common.

`inspect -close` closes the Property Inspector.

See Also

Figure Properties | `figure`

Topics

"Add Title and Axis Labels to Chart"
"Access Property Values"

Introduced before R2006a

instrcallback

(Not recommended) Event information when event occurs

Note This `serial` object function is not recommended. Use `serialport` object functions instead. See “Compatibility Considerations”.

Syntax

`instrcallback(obj,event)`

Description

`instrcallback(obj,event)` displays a message that contains the event type, `event`, the time the event occurred, and the name of the serial port object, `obj`, that caused the event to occur.

For error events, the error message is also displayed. For pin status events, the pin that changed value and its value are also displayed.

Examples

The following example creates the serial port object, `s`, on a Windows platform. It configures `s` to execute `instrcallback` when an output-empty event occurs. The event occurs after the `*IDN?` command is written to the instrument.

```
s = serial('COM1');
set(s,'OutputEmptyFcn',@instrcallback)
fopen(s)
fprintf(s,'*IDN?','async')
```

```
OutputEmpty event occurred at 08:37:49 for the object:
Serial-COM1.
```

Read the identification information from the input buffer and end the serial port session.

```
idn = fscanf(s);
fclose(s)
delete(s)
clear s
```

Tips

Use `instrcallback` as a template to create callback functions that suit your specific application needs.

Compatibility Considerations

`serial` object interface is not recommended

Not recommended starting in R2019b

Use of this function with a `serial` object is not recommended. To access a serial port device, use a `serialport` object with its functions and properties instead.

See “Transition Your Code to serialport Interface” for more information about using the recommended functionality.

Introduced before R2006a

instrfind

(Not recommended) Read serial port objects from memory to MATLAB workspace

Note This `serial` object function is not recommended. Use `serialport` object functions instead. See “Compatibility Considerations”.

Syntax

```
out = instrfind
out = instrfind('PropertyName',PropertyValue,...)
out = instrfind(S)
out = instrfind(obj,'PropertyName',PropertyValue,...)
```

Description

`out = instrfind` returns all valid serial port objects as an array to `out`.

`out = instrfind('PropertyName',PropertyValue,...)` returns an array of serial port objects whose property names and property values match those specified.

`out = instrfind(S)` returns an array of serial port objects whose property names and property values match those defined in the structure `S`. The field names of `S` are the property names, while the field values are the associated property values.

`out = instrfind(obj,'PropertyName',PropertyValue,...)` restricts the search for matching property name/property value pairs to the serial port objects listed in `obj`.

Examples

Suppose you create the following two serial port objects on a Windows platform.

```
s1 = serial('COM1');
s2 = serial('COM2');
```

```
set(s2,'BaudRate',4800)
fopen([s1 s2])
```

You can use `instrfind` to return serial port objects based on property values.

```
out1 = instrfind('Port','COM1');
out2 = instrfind({'Port','BaudRate'},{'COM2',4800});
```

You can also use `instrfind` to return cleared serial port objects to the MATLAB workspace.

```
clear s1 s2
newobjs = instrfind
```

```
   Instrument Object Array
   Index:   Type:          Status:     Name:
   1        serial         open        Serial-COM1
   2        serial         open        Serial-COM2
```

To close both `s1` and `s2`

```
fclose(newobjs)
```

Tips

Refer to “Displaying Property Names and Property Values” for a list of serial port object properties that you can use with `instrfind`.

You must specify property values using the same format as the `get` function returns. For example, if `get` returns the `Name` property value as `MyObject`, `instrfind` will not find an object with a `Name` property value of `myobject`. However, this is not the case for properties that have a finite set of string values. For example, `instrfind` will find an object with a `Parity` property value of `Even` or `even`.

You can use property name/property value string pairs, structures, and cell array pairs in the same call to `instrfind`.

Compatibility Considerations

`serial` object interface is not recommended

Not recommended starting in R2019b

Use of this function with a `serial` object is not recommended. To access a serial port device, use a `serialport` object with its functions and properties instead.

See “Transition Your Code to serialport Interface” for more information about using the recommended functionality.

See Also

Functions

Introduced before R2006a

instrfindall

(Not recommended) Find visible and hidden serial port objects

Note This `serial` object function is not recommended. Use `serialport` object functions instead. See “Compatibility Considerations”.

Syntax

```
out = instrfindall
out = instrfindall('P1',V1,...)
out = instrfindall(s)
out = instrfindall(objs,'P1',V1,...)
```

Description

`out = instrfindall` finds all serial port objects, regardless of the value of the object’s `ObjectVisibility` property. The object or objects are returned to `out`.

`out = instrfindall('P1',V1,...)` returns an array, `out`, of serial port objects whose property names and corresponding property values match those specified as arguments.

`out = instrfindall(s)` returns an array, `out`, of serial port objects whose property names and corresponding property values match those specified in the structure `s`, where the field names correspond to property names and the field values correspond to the current value of the respective property.

`out = instrfindall(objs,'P1',V1,...)` restricts the search for objects with matching property name/value pairs to the serial port objects listed in `objs`.

Note that you can use string property name/property value pairs, structures, and cell array property name/property value pairs in the same call to `instrfindall`.

Examples

Suppose you create the following serial port objects on a Windows platform:

```
s1 = serial('COM1');
s2 = serial('COM2');
set(s2,'ObjectVisibility','off')
```

Because object `s2` has its `ObjectVisibility` set to `'off'`, it is not visible to commands like `instrfind`:

```
instrfind
```

```
   Serial Port Object : Serial-COM1
```

However, `instrfindall` finds all objects regardless of the value of `ObjectVisibility`:

```
instrfindall
```

```
    Instrument Object Array
    Index:   Type:          Status:     Name:
    1        serial         closed      Serial-COM1
    2        serial         closed      Serial-COM2
```

The following statements use `instrfindall` to return objects with specific property settings, which are passed as cell arrays:

```
props = {'PrimaryAddress','SecondaryAddress};
vals = {2,0};
obj = instrfindall(props,vals);
```

You can use `instrfindall` as an argument when you want to apply the command to all objects, visible and invisible. For example, the following statement makes all objects visible:

```
set(instrfindall,'ObjectVisibility','on')
```

Tips

`instrfindall` differs from `instrfind` in that it finds objects whose `ObjectVisibility` property is set to `off`.

Property values are case sensitive. You must specify property values using the same format as that returned by the `get` function. For example, if `get` returns the `Name` property value as `'MyObject'`, `instrfindall` will not find an object with a `Name` property value of `'myobject'`. However, this is not the case for properties that have a finite set of string values. For example, `instrfindall` will find an object with a `Parity` property value of `'Even'` or `'even'`.

Compatibility Considerations

`serial` object interface is not recommended

Not recommended starting in R2019b

Use of this function with a `serial` object is not recommended. To access a serial port device, use a `serialport` object with its functions and properties instead.

See “Transition Your Code to serialport Interface” for more information about using the recommended functionality.

See Also

Functions

`get` | `instrfind`

int2str

Convert integers to characters

Syntax

```
chr = int2str(N)
```

Description

`chr = int2str(N)` treats `N` as a matrix of integers and converts it to a character array that represents the integers. If `N` contains floating-point values, `int2str` rounds them before conversion.

Examples

Convert Integers

Convert an integer.

```
chr = int2str(256)
```

```
chr =
'256'
```

Round off a floating-point value and convert it.

```
chr = int2str(3.14159)
```

```
chr =
'3'
```

Convert a numeric matrix.

```
chr = int2str([5 10 20;100 200 400])
```

```
chr = 2x13 char array
    '  5   10   20'
    '100  200  400'
```

Input Arguments

N — Input array
numeric matrix

Input array, specified as a numeric matrix.

Tips

- `int2str` returns character arrays only. Starting in R2016b, you can convert numeric arrays to string arrays using the `string` function.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Input array must be scalar at compile time.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`cast` | `mat2str` | `num2str` | `sprintf` | `str2double` | `str2num` | `string`

Topics

“Convert from Numeric Values to Character Array”
“Convert from Character Arrays to Numeric Values”

Introduced before R2006a

int8

8-bit signed integer arrays

Description

Variables in MATLAB of data type (class) `int8` are stored as 1-byte (8-bit) signed integers. For example:

```
y = int8(10);
whos y
```

```
  Name      Size            Bytes  Class    Attributes

  y         1x1                 1  int8
```

For more information on integer types, see “Integers”.

Creation

Some array creation functions allow you to specify the data type. For instance, `zeros(100,'int8')` creates a 100-by-100 matrix of zeros of type `int8`.

If you have an array of a different type, such as `double` or `single`, then you can convert that array to an array of type `int8` by using the `int8` function.

Syntax

```
Y = int8(X)
```

Description

`Y = int8(X)` converts the values in `X` to type `int8`. Values outside the range [-2^7,2^7-1] map to the nearest endpoint.

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `double` | `single` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`

Examples

Convert to 8-Bit Signed Integer Variable

Convert a double-precision variable to an 8-bit signed integer.

```
x = 100;
xtype = class(x)
```

```
xtype =
'double'
```

```
y = int8(x)
```

```
y = int8
    100
```

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cast` | `int16` | `int32` | `int64` | `typecast` | `uint8`

Topics

“Integers”
“Identifying Numeric Classes”
“Hexadecimal and Binary Values”

Introduced before R2006a

int16

16-bit signed integer arrays

Description

Variables in MATLAB of data type (class) `int16` are stored as 2-byte (16-bit) signed integers. For example:

```
y = int16(10);
whos y
```

```
  Name      Size            Bytes  Class    Attributes

  y         1x1                 2  int16
```

For more information on integer types, see “Integers”.

Creation

Some array creation functions allow you to specify the data type. For instance, `zeros(100,'int16')` creates a 100-by-100 matrix of zeros of type `int16`.

If you have an array of a different type, such as `double` or `single`, then you can convert that array to an array of type `int16` by using the `int16` function.

Syntax

```
Y = int16(X)
```

Description

`Y = int16(X)` converts the values in `X` to type `int16`. Values outside the range $[-2^{15},2^{15}-1]$ map to the nearest endpoint.

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `double` | `single` | `int8` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`

Examples

Convert to 16-Bit Signed Integer Variable

Convert a double-precision variable to a 16-bit signed integer.

```
x = 100;
xtype = class(x)
```

```
xtype =
'double'
```

```
y = int16(x)
```

```
y = int16
    100
```

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cast` | `int32` | `int64` | `int8` | `typecast` | `uint16`

Topics

“Integers”
“Identifying Numeric Classes”
“Hexadecimal and Binary Values”

Introduced before R2006a

int32

32-bit signed integer arrays

Description

Variables in MATLAB of data type (class) `int32` are stored as 4-byte (32-bit) signed integers. For example:

```
y = int32(10);
whos y
```

```
  Name      Size            Bytes  Class    Attributes

  y         1x1                 4  int32
```

For more information on integer types, see “Integers”.

Creation

Some array creation functions allow you to specify the data type. For instance, `zeros(100,'int32')` creates a 100-by-100 matrix of zeros of type `int32`.

If you have an array of a different type, such as `double` or `single`, then you can convert that array to an array of type `int32` by using the `int32` function.

Syntax

```
Y = int32(X)
```

Description

`Y = int32(X)` converts the values in `X` to type `int32`. Values outside the range $[-2^{31},2^{31}-1]$ map to the nearest endpoint.

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `double` | `single` | `int8` | `int16` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`

Examples

Convert to 32-Bit Signed Integer Variable

Convert a double-precision variable to a 32-bit signed integer.

```
x = 100;
xtype = class(x)
```

```
xtype =
'double'
```

```
y = int32(x)
```

```
y = int32
    100
```

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cast` | `int16` | `int64` | `int8` | `typecast` | `uint32`

Topics

“Integers”
“Identifying Numeric Classes”
“Hexadecimal and Binary Values”

Introduced before R2006a

int64

64-bit signed integer arrays

Description

Variables in MATLAB of data type (class) `int64` are stored as 8-byte (64-bit) signed integers. For example:

```
y = int64(10);
whos y
```

```
  Name      Size            Bytes  Class    Attributes

  y         1x1                 8  int64
```

For more information on integer types, see “Integers”.

Creation

Some array creation functions allow you to specify the data type. For instance, `zeros(100,'int64')` creates a 100-by-100 matrix of zeros of type `int64`.

If you have an array of a different type, such as `double` or `single`, then you can convert that array to an array of type `int64` by using the `int64` function.

Syntax

```
Y = int64(X)
```

Description

`Y = int64(X)` converts the values in `X` to type `int64`. Values outside the range $[-2^{63},2^{63}-1]$ map to the nearest endpoint.

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`

Examples

Convert to 64-Bit Signed Integer Variable

Convert a double-precision variable to a 64-bit signed integer.

```
x = 100;
xtype = class(x)
```

```
xtype =
'double'
```

```
y = int64(x)
```

```
y = int64
    100
```

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cast` | `int16` | `int32` | `int8` | `typecast` | `uint64`

Topics

“Integers”
“Identifying Numeric Classes”
“Hexadecimal and Binary Values”

Introduced before R2006a

integral

Numerical integration

Syntax

```
q = integral(fun,xmin,xmax)
q = integral(fun,xmin,xmax,Name,Value)
```

Description

`q = integral(fun,xmin,xmax)` numerically integrates function `fun` from `xmin` to `xmax` using global adaptive quadrature and default error tolerances.

`q = integral(fun,xmin,xmax,Name,Value)` specifies additional options with one or more `Name,Value` pair arguments. For example, specify `'WayPoints'` followed by a vector of real or complex numbers to indicate specific points for the integrator to use.

Examples

Improper Integral

Create the function $f(x) = e^{-x^2}(\ln x)^2$.

```
fun = @(x) exp(-x.^2).*log(x).^2;
```

Evaluate the integral from `x=0` to `x=Inf`.

```
q = integral(fun,0,Inf)
```

```
q = 1.9475
```

Parameterized Function

Create the function $f(x) = 1/(x^3 - 2x - c)$ with one parameter, c.

```
fun = @(x,c) 1./(x.^3-2*x-c);
```

Evaluate the integral from `x=0` to `x=2` at `c=5`.

```
q = integral(@(x)fun(x,5),0,2)
```

```
q = -0.4605
```

Singularity at Lower Limit

Create the function $f(x) = \ln(x)$.

```
fun = @(x)log(x);
```

Evaluate the integral from `x=0` to `x=1` with the default error tolerances.

```
format long
q1 = integral(fun,0,1)
```

```
q1 =
  -1.000000010959678
```

Evaluate the integral again, this time with 12 decimal places of accuracy. Set `RelTol` to zero so that `integral` only attempts to satisfy the absolute error tolerance.

```
q2 = integral(fun,0,1,'RelTol',0,'AbsTol',1e-12)
```

```
q2 =
  -1.000000000000010
```

Complex Contour Integration Using Waypoints

Create the function $f(z) = 1/(2z - 1)$.

```
fun = @(z) 1./(2*z-1);
```

Integrate in the complex plane over the triangular path from `0` to `1+1i` to `1-1i` to `0` by specifying waypoints.

```
q = integral(fun,0,0,'Waypoints',[1+1i,1-1i])
```

```
q = 0.0000 - 3.1416i
```

Vector-Valued Function

Create the vector-valued function $f(x) = [\sin x,\ \sin 2x,\ \sin 3x,\ \sin 4x,\ \sin 5x]$ and integrate from `x=0` to `x=1`. Specify `'ArrayValued',true` to evaluate the integral of an array-valued or vector-valued function.

```
fun = @(x)sin((1:5)*x);
q = integral(fun,0,1,'ArrayValued',true)
```

```
q = 1×5

    0.4597    0.7081    0.6633    0.4134    0.1433
```

Improper Integral of Oscillatory Function

Create the function $f(x) = x^5 e^{-x} \sin x$.

```
fun = @(x)x.^5.*exp(-x).*sin(x);
```

Evaluate the integral from `x=0` to `x=Inf` , adjusting the absolute and relative tolerances.

```
format long
q = integral(fun,0,Inf,'RelTol',1e-8,'AbsTol',1e-13)
```

```
q =
 -14.999999999998364
```

Input Arguments

`fun` — Integrand
function handle

Integrand, specified as a function handle, which defines the function to be integrated from `xmin` to `xmax`.

For scalar-valued problems, the function `y = fun(x)` must accept a vector argument, `x`, and return a vector result, `y`. This generally means that `fun` must use array operators instead of matrix operators. For example, use `.*` (`times`) rather than `*` (`mtimes`). If you set the `'ArrayValued'` option to `true`, then `fun` must accept a scalar and return an array of fixed size.

`xmin` — Lower limit of *x*
real number | complex number

Lower limit of *x*, specified as a real (finite or infinite) scalar value or a complex (finite) scalar value. If either `xmin` or `xmax` are complex, then `integral` approximates the path integral from `xmin` to `xmax` over a straight line path.

Data Types: `double` | `single`
Complex Number Support: Yes

`xmax` — Upper limit of *x*
real number | complex number

Upper limit of *x*, specified as a real number (finite or infinite) or a complex number (finite). If either `xmin` or `xmax` are complex, `integral` approximates the path integral from `xmin` to `xmax` over a straight line path.

Data Types: `double` | `single`
Complex Number Support: Yes

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `integral(fun,a,b,'AbsTol',1e-12)` sets the absolute error tolerance to approximately 12 decimal places of accuracy.

`AbsTol` — Absolute error tolerance
`1e-10` (default) | nonnegative real number

Absolute error tolerance, specified as the comma-separated pair consisting of `'AbsTol'` and a nonnegative real number. `integral` uses the absolute error tolerance to limit an estimate of the absolute error, $|q - Q|$, where q is the computed value of the integral and Q is the (unknown) exact value. `integral` might provide more decimal places of precision if you decrease the absolute error tolerance.

Note `AbsTol` and `RelTol` work together. `integral` might satisfy the absolute error tolerance or the relative error tolerance, but not necessarily both. For more information on using these tolerances, see the "Tips" on page 1-7793 section.

Example: `integral(fun,a,b,'AbsTol',1e-12)` sets the absolute error tolerance to approximately 12 decimal places of accuracy.

Data Types: `single` | `double`

`RelTol` — Relative error tolerance
`1e-6` (default) | nonnegative real number

Relative error tolerance, specified as the comma-separated pair consisting of `'RelTol'` and a nonnegative real number. `integral` uses the relative error tolerance to limit an estimate of the relative error, $|q - Q|/|Q|$, where q is the computed value of the integral and Q is the (unknown) exact value. `integral` might provide more significant digits of precision if you decrease the relative error tolerance.

Note `RelTol` and `AbsTol` work together. `integral` might satisfy the relative error tolerance or the absolute error tolerance, but not necessarily both. For more information on using these tolerances, see the Tips on page 1-7793 section.

Example: `integral(fun,a,b,'RelTol',1e-9)` sets the relative error tolerance to approximately 9 significant digits.

Data Types: `single` | `double`

`ArrayValued` — Array-valued function flag
`false` or `0` (default) | `true` or `1`

Array-valued function flag, specified as the comma-separated pair consisting of `'ArrayValued'` and a numeric or logical `1` (`true`) or `0` (`false`). Set this flag to `true` or `1` to indicate that `fun` is a function that accepts a scalar input and returns a vector, matrix, or N-D array output.

The default value of `false` indicates that `fun` is a function that accepts a vector input and returns a vector output.

Example: `integral(fun,a,b,'ArrayValued',true)` indicates that the integrand is an array-valued function.

`Waypoints` — Integration waypoints
vector

Integration waypoints, specified as the comma-separated pair consisting of `'Waypoints'` and a vector of real or complex numbers. Use waypoints to indicate points in the integration interval that you would like the integrator to use in the initial mesh:

- Add more evaluation points near interesting features of the function, such as a local extrema.
- Integrate efficiently across discontinuities of the integrand by specifying the locations of the discontinuities.
- Perform complex contour integrations by specifying complex numbers as waypoints. If `xmin`, `xmax`, or any entry of the waypoints vector is complex, then the integration is performed over a sequence of straight line paths in the complex plane. In this case, all of the integration limits and waypoints must be finite.

Do not use waypoints to specify singularities. Instead, split the interval and add the results of separate integrations with the singularities at the endpoints.

Example: `integral(fun,a,b,'Waypoints',[1+1i,1-1i])` specifies two complex waypoints along the interval of integration.

Data Types: `single` | `double`
Complex Number Support: Yes

Tips

- The `integral` function attempts to satisfy:

  ```
  abs(q - Q) <= max(AbsTol,RelTol*abs(q))
  ```

  where `q` is the computed value of the integral and `Q` is the (unknown) exact value. The absolute and relative tolerances provide a way of trading off accuracy and computation time. Usually, the relative tolerance determines the accuracy of the integration. However if `abs(q)` is sufficiently small, the absolute tolerance determines the accuracy of the integration. You should generally specify both absolute and relative tolerances together.
- If you are specifying single-precision limits of integration, or if `fun` returns single-precision results, you might need to specify larger absolute and relative error tolerances.

References

[1] L.F. Shampine "*Vectorized Adaptive Quadrature in MATLAB*," *Journal of Computational and Applied Mathematics*, 211, 2008, pp.131–140.

See Also

`integral2` | `integral3` | `trapz`

Topics

"Integration of Numeric Data"
"Integration to Find Arc Length"
"Complex Line Integrals"
"Create Function Handle"
"Parameterizing Functions"

Introduced in R2012a

integral2

Numerically evaluate double integral

Syntax

```
q = integral2(fun,xmin,xmax,ymin,ymax)
q = integral2(fun,xmin,xmax,ymin,ymax,Name,Value)
```

Description

`q = integral2(fun,xmin,xmax,ymin,ymax)` approximates the integral of the function `z = fun(x,y)` over the planar region `xmin` ≤ `x` ≤ `xmax` and `ymin(x)` ≤ `y` ≤ `ymax(x)`.

`q = integral2(fun,xmin,xmax,ymin,ymax,Name,Value)` specifies additional options with one or more `Name,Value` pair arguments.

Examples

Integrate Triangular Region with Singularity at the Boundary

Consider the function

$$f(x,y) = \frac{1}{(\sqrt{x+y})(1+x+y)^2}.$$

This function is undefined when x and y are zero. `integral2` performs best when singularities are on the integration boundary.

Create the anonymous function.

```
fun = @(x,y) 1./( sqrt(x + y) .* (1 + x + y).^2 )
```

```
fun = function_handle with value:
    @(x,y)1./(sqrt(x+y).*(1+x+y).^2)
```

Integrate over the triangular region bounded by $0 \le x \le 1$ and $0 \le y \le 1 - x$.

```
ymax = @(x) 1 - x;
q = integral2(fun,0,1,0,ymax)
```

```
q = 0.2854
```

Evaluate Double Integral in Polar Coordinates

Define the function

$$f(\theta, r) = \frac{r}{\sqrt{r\cos\theta + r\sin\theta}(1 + r\cos\theta + r\sin\theta)^2}$$

```
fun = @(x,y) 1./( sqrt(x + y) .* (1 + x + y).^2 );
polarfun = @(theta,r) fun(r.*cos(theta),r.*sin(theta)).*r;
```

Define a function for the upper limit of r.

```
rmax = @(theta) 1./(sin(theta) + cos(theta));
```

Integrate over the region bounded by $0 \le \theta \le \pi/2$ and $0 \le r \le r_{max}$.

```
q = integral2(polarfun,0,pi/2,0,rmax)
```

```
q = 0.2854
```

Evaluate Double Integral of Parameterized Function with Specific Method and Error Tolerance

Create the anonymous parameterized function $f(x, y) = ax^2 + by^2$ with parameters $a = 3$ and $b = 5$.

```
a = 3;
b = 5;
fun = @(x,y) a*x.^2 + b*y.^2;
```

Evaluate the integral over the region $0 \le x \le 5$ and $-5 \le y \le 0$. Specify the `'iterated'` method and approximately 10 significant digits of accuracy.

```
format long
q = integral2(fun,0,5,-5,0,'Method','iterated',...
'AbsTol',0,'RelTol',1e-10)
```

```
q =
     1.666666666666666e+03
```

Input Arguments

`fun` — Integrand
function handle

Integrand, specified as a function handle, defines the function to be integrated over the planar region `xmin` ≤ `x` ≤ `xmax` and `ymin`(`x`) ≤ `y` ≤ `ymax`(`x`). The function `fun` must accept two arrays of the same size and return an array of corresponding values. It must perform element-wise operations.

Data Types: `function_handle`

`xmin` — Lower limit of *x*
real number

Lower limit of *x*, specified as a real scalar value that is either finite or infinite.

Data Types: `double` | `single`

`xmax` — Upper limit of *x*
real number

Upper limit of *x*, specified as a real scalar value that is either finite or infinite.

Data Types: `double` | `single`

`ymin` — Lower limit of *y*
real number | function handle

Lower limit of *y*, specified as a real scalar value that is either finite or infinite. You can specify `ymin` to be a function handle (a function of *x*) when integrating over a nonrectangular region.

Data Types: `double` | `function_handle` | `single`

`ymax` — Upper limit of *y*
real number | function handle

Upper limit of *y*, specified as a real scalar value that is either finite or infinite. You also can specify `ymax` to be a function handle (a function of *x*) when integrating over a nonrectangular region.

Data Types: `double` | `function_handle` | `single`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'AbsTol',1e-12` sets the absolute error tolerance to approximately 12 decimal places of accuracy.

`AbsTol` — Absolute error tolerance
nonnegative real number

Absolute error tolerance, specified as the comma-separated pair consisting of `'AbsTol'` and a nonnegative real number. `integral2` uses the absolute error tolerance to limit an estimate of the absolute error, $|q - Q|$, where q is the computed value of the integral and Q is the (unknown) exact value. `integral2` might provide more decimal places of precision if you decrease the absolute error tolerance. The default value is `1e-10`.

Note `AbsTol` and `RelTol` work together. `integral2` might satisfy the absolute error tolerance or the relative error tolerance, but not necessarily both. For more information on using these tolerances, see the “Tips” on page 1-7799 section.

Example: `'AbsTol',1e-12` sets the absolute error tolerance to approximately 12 decimal places of accuracy.

Data Types: `double` | `single`

`RelTol` — Relative error tolerance
nonnegative real number

Relative error tolerance, specified as the comma-separated pair consisting of `'RelTol'` and a nonnegative real number. `integral2` uses the relative error tolerance to limit an estimate of the relative error, $|q - Q|/|Q|$, where q is the computed value of the integral and Q is the (unknown) exact value. `integral2` might provide more significant digits of precision if you decrease the relative error tolerance. The default value is `1e-6`.

Note `RelTol` and `AbsTol` work together. `integral2` might satisfy the relative error tolerance or the absolute error tolerance, but not necessarily both. For more information on using these tolerances, see the "Tips" on page 1-7799 section.

Example: `'RelTol',1e-9` sets the relative error tolerance to approximately 9 significant digits.

Data Types: `double` | `single`

`Method` — Integration method
`'auto'` (default) | `'tiled'` | `'iterated'`

Integration method, specified as the comma-separated pair consisting of `'Method'` and one of the methods described below.

Integration Method	Description
`'auto'`	For most cases, `integral2` uses the `'tiled'` method. It uses the `'iterated'` method when any of the integration limits are infinite. This is the default method.
`'tiled'`	`integral2` transforms the region of integration to a rectangular shape and subdivides it into smaller rectangular regions as needed. The integration limits must be finite.
`'iterated'`	`integral2` calls `integral` to perform an iterated integral. The outer integral is evaluated over `xmin` ≤ `x` ≤ `xmax`. The inner integral is evaluated over `ymin(x)` ≤ `y` ≤ `ymax(x)`. The integration limits can be infinite.

Example: `'Method','tiled'` specifies the tiled integration method.

Data Types: `char` | `string`

Tips

- The `integral2` function attempts to satisfy:

  ```
  abs(q - Q) <= max(AbsTol,RelTol*abs(q))
  ```

  where `q` is the computed value of the integral and `Q` is the (unknown) exact value. The absolute and relative tolerances provide a way of trading off accuracy and computation time. Usually, the relative tolerance determines the accuracy of the integration. However if `abs(q)` is sufficiently small, the absolute tolerance determines the accuracy of the integration. You should generally specify both absolute and relative tolerances together.
- The `'iterated'` method can be more effective when your function has discontinuities within the integration region. However, the best performance and accuracy occurs when you split the integral at the points of discontinuity and sum the results of multiple integrations.
- When integrating over nonrectangular regions, the best performance and accuracy occurs when `ymin`, `ymax`, (or both) are function handles. Avoid setting integrand function values to zero to integrate over a nonrectangular region. If you must do this, specify `'iterated'` method.
- Use the `'iterated'` method when `ymin`, `ymax`, (or both) are unbounded functions.
- When paramaterizing anonymous functions, be aware that parameter values persist for the life of the function handle. For example, the function `fun = @(x,y) x + y + a` uses the value of `a` at the time `fun` was created. If you later decide to change the value of `a`, you must redefine the anonymous function with the new value.
- If you are specifying single-precision limits of integration, or if `fun` returns single-precision results, you might need to specify larger absolute and relative error tolerances.

References

[1] L.F. Shampine *"Vectorized Adaptive Quadrature in MATLAB," Journal of Computational and Applied Mathematics*, 211, 2008, pp.131–140.

[2] L.F. Shampine, "*MATLAB Program for Quadrature in 2D.*" *Applied Mathematics and Computation.* Vol. 202, Issue 1, 2008, pp. 266–274.

See Also

`integral` | `integral3` | `trapz`

Topics

"Singularity on Interior of Integration Domain"
"Parameterizing Functions"
"Create Function Handle"

Introduced in R2012a

integral3

Numerically evaluate triple integral

Syntax

```
q = integral3(fun,xmin,xmax,ymin,ymax,zmin,zmax)
q = integral3(fun,xmin,xmax,ymin,ymax,zmin,zmax,Name,Value)
```

Description

`q = integral3(fun,xmin,xmax,ymin,ymax,zmin,zmax)` approximates the integral of the function `z = fun(x,y,z)` over the region `xmin` ≤ `x` ≤ `xmax`, `ymin(x)` ≤ `y` ≤ `ymax(x)` and `zmin(x,y)` ≤ `z` ≤ `zmax(x,y)`.

`q = integral3(fun,xmin,xmax,ymin,ymax,zmin,zmax,Name,Value)` specifies additional options with one or more `Name,Value` pair arguments.

Examples

Triple Integral with Finite Limits

Define the anonymous function $f(x, y, z) = y\sin x + z\cos x$.

```
fun = @(x,y,z) y.*sin(x)+z.*cos(x)
```

```
fun = function_handle with value:
    @(x,y,z)y.*sin(x)+z.*cos(x)
```

Integrate over the region $0 \le x \le \pi$, $0 \le y \le 1$, and $-1 \le z \le 1$.

```
q = integral3(fun,0,pi,0,1,-1,1)
```

```
q = 2.0000
```

Integral Over the Unit Sphere in Cartesian Coordinates

Define the anonymous function $f(x, y, z) = x\cos y + x^2\cos z$.

```
fun = @(x,y,z) x.*cos(y) + x.^2.*cos(z)
```

```
fun = function_handle with value:
    @(x,y,z)x.*cos(y)+x.^2.*cos(z)
```

Define the limits of integration.

```
xmin = -1;
xmax = 1;
ymin = @(x)-sqrt(1 - x.^2);
ymax = @(x) sqrt(1 - x.^2);
zmin = @(x,y)-sqrt(1 - x.^2 - y.^2);
zmax = @(x,y) sqrt(1 - x.^2 - y.^2);
```

Evaluate the definite integral with the `'tiled'` method.

```
q = integral3(fun,xmin,xmax,ymin,ymax,zmin,zmax,'Method','tiled')
```

```
q = 0.7796
```

Evaluate Improper Triple Integral of Parameterized Function

Define the anonymous parameterized function $f(x, y, z) = 10/(x^2 + y^2 + z^2 + a)$.

```
a = 2;
f = @(x,y,z) 10./(x.^2 + y.^2 + z.^2 + a);
```

Evaluate the triple integral over the region $-\infty \le x \le 0$, $-100 \le y \le 0$, and $-100 \le z \le 0$.

```
format long
q1 = integral3(f,-Inf,0,-100,0,-100,0)
```

```
q1 =
     2.734244598320928e+03
```

Evaluate the integral again and specify accuracy to approximately 9 significant digits.

```
q2 = integral3(f,-Inf,0,-100,0,-100,0,'AbsTol', 0,'RelTol',1e-9)
```

```
q2 =
     2.734244599944285e+03
```

4-D Integral of Sphere

Use nested calls to `integral3` and `integral` to calculate the volume of a 4-D sphere.

The volume of a 4-D sphere of radius r is

$$V_4(r) = \int_0^{2\pi}\int_0^{\pi}\int_0^{\pi}\int_0^{r} r^3 \sin^2(\theta)\sin(\phi)\, dr\, d\theta\, d\phi\, d\xi\,.$$

The `integral` quadrature functions in MATLAB® directly support 1-D, 2-D, and 3-D integrations. However, to solve 4-D and higher order integrals, you need to nest calls to the solvers.

Create a function handle $f(r,\theta,\phi,\xi)$ for the integrand using element-wise operators (`.^` and `.*`).

```
f = @(r,theta,phi,xi) r.^3 .* sin(theta).^2 .* sin(phi);
```

Next, create a function handle that calculates three of the integrals using `integral3`.

```
Q = @(r) integral3(@(theta,phi,xi) f(r,theta,phi,xi),0,pi,0,pi,0,2*pi);
```

Finally, use `Q` as the integrand in a call to `integral`. Solving this integral requires choosing a value for the radius r, so use $r = 2$.

```
I = integral(Q,0,2,'ArrayValued',true)
```

```
I = 78.9568
```

The exact answer is $\frac{\pi^2 r^4}{2\,\Gamma(2)}$.

```
I_exact = pi^2*2^4/(2*gamma(2))
```

```
I_exact = 78.9568
```

Input Arguments

`fun` — Integrand
function handle

Integrand, specified as a function handle, defines the function to be integrated over the region `xmin` ≤ `x` ≤ `xmax`, `ymin`(`x`) ≤ `y` ≤ `ymax`(`x`), and `zmin`(`x,y`) ≤ `z` ≤ `zmax`(`x,y`). The function `fun` must accept three arrays of the same size and return an array of corresponding values. It must perform element-wise operations.

Data Types: `function_handle`

`xmin` — Lower limit of *x*
real number

Lower limit of *x*, specified as a real scalar value that is either finite or infinite.

Data Types: `double` | `single`

`xmax` — Upper limit of *x*
real number

Upper limit of *x*, specified as a real scalar value that is either finite or infinite.

Data Types: `double` | `single`

`ymin` — Lower limit of *y*
real number | function handle

Lower limit of *y*, specified as a real scalar value that is either finite or infinite. You also can specify `ymin` to be a function handle (a function of *x*) when integrating over a nonrectangular region.

Data Types: `double` | `function_handle` | `single`

`ymax` — Upper limit of *y*
real number | function handle

Upper limit of *y*, specified as a real scalar value that is either finite or infinite. You also can specify `ymax` to be a function handle (a function of *x*) when integrating over a nonrectangular region.

Data Types: `double` | `function_handle` | `single`

`zmin` — Lower limit of *z*
real number | function handle

Lower limit of *z*, specified as a real scalar value that is either finite or infinite. You also can specify `zmin` to be a function handle (a function of *x*,*y*) when integrating over a nonrectangular region.

Data Types: `double` | `function_handle` | `single`

`zmax` — Upper limit of *z*
real number | function handle

Upper limit of *z*, specified as a real scalar value that is either finite or infinite. You also can specify `zmax` to be a function handle (a function of *x*,*y*) when integrating over a nonrectangular region.

Data Types: `double` | `function_handle` | `single`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'AbsTol',1e-12` sets the absolute error tolerance to approximately 12 decimal places of accuracy.

`AbsTol` — Absolute error tolerance
nonnegative real number

Absolute error tolerance, specified as the comma-separated pair consisting of `'AbsTol'` and a nonnegative real number. `integral3` uses the absolute error tolerance to limit an estimate of the absolute error, $|q - Q|$, where q is the computed value of the integral and Q is the (unknown) exact value. `integral3` might provide more decimal places of precision if you decrease the absolute error tolerance. The default value is `1e-10`.

Note `AbsTol` and `RelTol` work together. `integral3` might satisfy the absolute error tolerance or the relative error tolerance, but not necessarily both. For more information on using these tolerances, see the “Tips” on page 1-7807 section.

Example: `'AbsTol',1e-12` sets the absolute error tolerance to approximately 12 decimal places of accuracy.

Data Types: `double` | `single`

`RelTol` — Relative error tolerance
nonnegative real number

Relative error tolerance, specified as the comma-separated pair consisting of `'RelTol'` and a nonnegative real number. `integral3` uses the relative error tolerance to limit an estimate of the relative error, $|q - Q|/|Q|$, where q is the computed value of the integral and Q is the (unknown) exact value. `integral3` might provide more significant digits of precision if you decrease the relative error tolerance. The default value is `1e-6`.

Note `RelTol` and `AbsTol` work together. `integral3` might satisfy the relative error tolerance or the absolute error tolerance, but not necessarily both. For more information on using these tolerances, see the "Tips" on page 1-7807 section.

Example: `'RelTol',1e-9` sets the relative error tolerance to approximately 9 significant digits.

Data Types: `double` | `single`

`Method` — Integration method
`'auto'` (default) | `'tiled'` | `'iterated'`

Integration method, specified as the comma-separated pair consisting of `'Method'` and one of the methods described below.

Integration Method	Description
`'auto'`	For most cases, `integral3` uses the `'tiled'` method. It uses the `'iterated'` method when any of the integration limits are infinite. This is the default method.
`'tiled'`	`integral3` calls `integral` to integrate over `xmin` ≤ `x` ≤ `xmax`. It calls `integral2` with the `'tiled'` method to evaluate the double integral over `ymin(x)` ≤ `y` ≤ `ymax(x)` and `zmin(x,y)` ≤ `z` ≤ `zmax(x,y)`.

Integration Method	Description
`'iterated'`	`integral3` calls `integral` to integrate over `xmin` ≤ `x` ≤ `xmax`. It calls `integral2` with the `'iterated'` method to evaluate the double integral over `ymin(x)` ≤ `y` ≤ `ymax(x)` and `zmin(x,y)` ≤ `z` ≤ `zmax(x,y)`. The integration limits can be infinite.

Example: `'Method','tiled'` specifies the tiled integration method.

Data Types: `char` | `string`

Tips

- The `integral3` function attempts to satisfy:

  ```
  abs(q - Q) <= max(AbsTol,RelTol*abs(q))
  ```

  where `q` is the computed value of the integral and `Q` is the (unknown) exact value. The absolute and relative tolerances provide a way of trading off accuracy and computation time. Usually, the relative tolerance determines the accuracy of the integration. However if `abs(q)` is sufficiently small, the absolute tolerance determines the accuracy of the integration. You should generally specify both absolute and relative tolerances together.
- The `'iterated'` method can be more effective when your function has discontinuities within the integration region. However, the best performance and accuracy occurs when you split the integral at the points of discontinuity and sum the results of multiple integrations.
- When integrating over nonrectangular regions, the best performance and accuracy occurs when any or all of the limits: `ymin`, `ymax`, `zmin`, `zmax` are function handles. Avoid setting integrand function values to zero to integrate over a nonrectangular region. If you must do this, specify `'iterated'` method.
- Use the `'iterated'` method when any or all of the limits: `ymin(x)`, `ymax(x)`, `zmin(x,y)`, `zmax(x,y)` are unbounded functions.
- When paramaterizing anonymous functions, be aware that parameter values persist for the life of the function handle. For example, the function `fun = @(x,y,z) x + y + z + a` uses the value of `a` at the time `fun` was created. If you later decide to change the value of `a`, you must redefine the anonymous function with the new value.

- If you are specifying single-precision limits of integration, or if `fun` returns single-precision results, you may need to specify larger absolute and relative error tolerances.
- To solve 4-D and higher order integrals, you can nest calls to `integral`, `integral2`, and `integral3`. Another option is to use the `integralN` function on the MATLAB File Exchange, which solves integrals of orders 4 - 6.

References

[1] L.F. Shampine "*Vectorized Adaptive Quadrature in MATLAB*," *Journal of Computational and Applied Mathematics*, 211, 2008, pp.131–140.

[2] L.F. Shampine, "*MATLAB Program for Quadrature in 2D.*" *Applied Mathematics and Computation.* Vol. 202, Issue 1, 2008, pp. 266–274.

See Also

`integral` | `integral2` | `trapz`

Topics

"Parameterizing Functions"
"Create Function Handle"

Introduced in R2012a

interp1

1-D data interpolation (table lookup)

Syntax

```
vq = interp1(x,v,xq)
vq = interp1(x,v,xq,method)
vq = interp1(x,v,xq,method,extrapolation)

vq = interp1(v,xq)
vq = interp1(v,xq,method)
vq = interp1(v,xq,method,extrapolation)

pp = interp1(x,v,method,'pp')
```

Description

`vq = interp1(x,v,xq)` returns interpolated values of a 1-D function at specific query points using linear interpolation. Vector `x` contains the sample points, and `v` contains the corresponding values, $v(x)$. Vector `xq` contains the coordinates of the query points.

If you have multiple sets of data that are sampled at the same point coordinates, then you can pass `v` as an array. Each column of array `v` contains a different set of 1-D sample values.

`vq = interp1(x,v,xq,method)` specifies an alternative interpolation method: `'linear'`, `'nearest'`, `'next'`, `'previous'`, `'pchip'`, `'cubic'`, `'v5cubic'`, `'makima'`, or `'spline'`. The default method is `'linear'`.

`vq = interp1(x,v,xq,method,extrapolation)` specifies a strategy for evaluating points that lie outside the domain of `x`. Set `extrapolation` to `'extrap'` when you want to use the `method` algorithm for extrapolation. Alternatively, you can specify a scalar value, in which case, `interp1` returns that value for all points outside the domain of `x`.

`vq = interp1(v,xq)` returns interpolated values and assumes a default set of sample point coordinates. The default points are the sequence of numbers from `1` to `n`, where `n` depends on the shape of `v`:

- When v is a vector, the default points are `1:length(v)`.
- When v is an array, the default points are `1:size(v,1)`.

Use this syntax when you are not concerned about the absolute distances between points.

`vq = interp1(v,xq,method)` specifies any of the alternative interpolation methods and uses the default sample points.

`vq = interp1(v,xq,method,extrapolation)` specifies an extrapolation strategy and uses the default sample points.

`pp = interp1(x,v,method,'pp')` returns the piecewise polynomial form of $v(x)$ using the `method` algorithm.

Note This syntax is not recommended. Use `griddedInterpolant` instead.

Examples

Interpolation of Coarsely Sampled Sine Function

Define the sample points, `x`, and corresponding sample values, `v`.

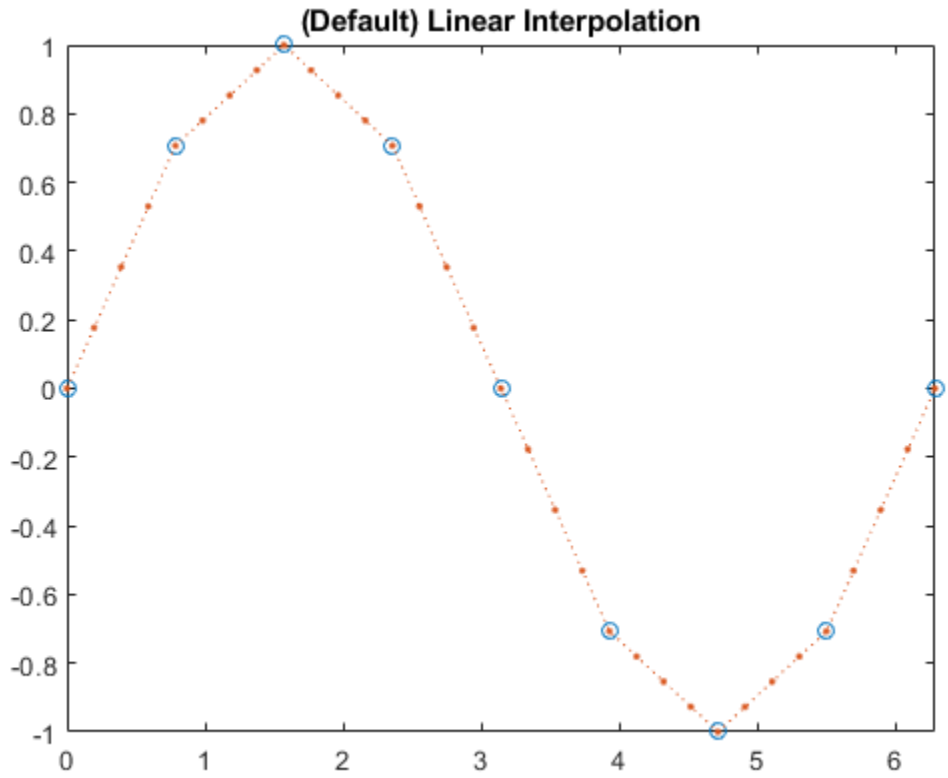
```
x = 0:pi/4:2*pi;
v = sin(x);
```

Define the query points to be a finer sampling over the range of `x`.

```
xq = 0:pi/16:2*pi;
```

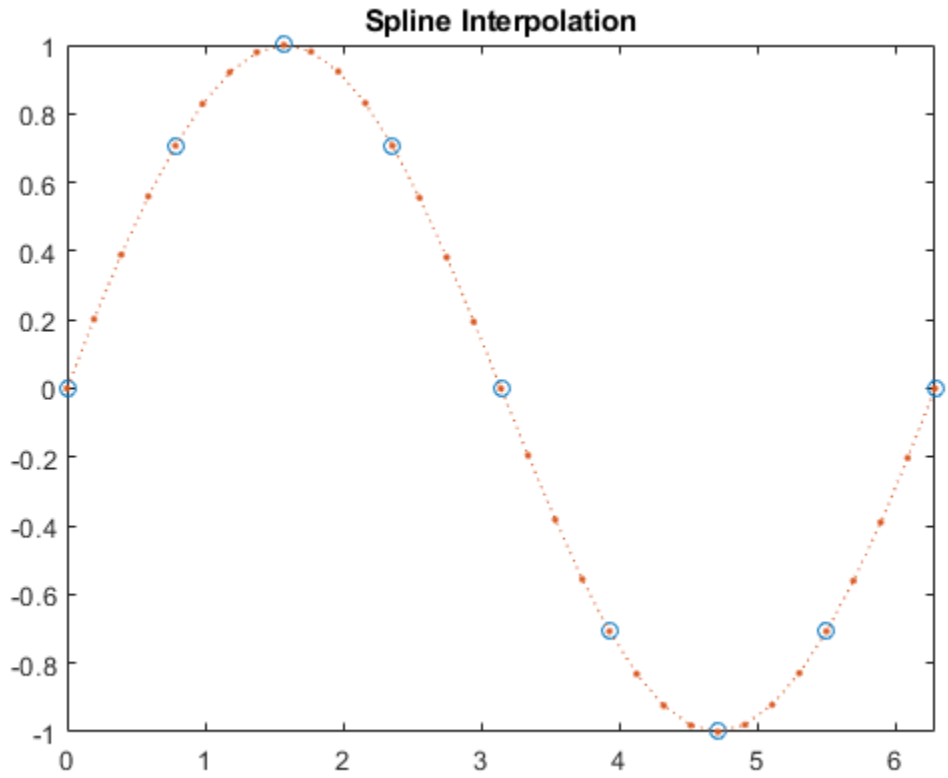
Interpolate the function at the query points and plot the result.

```
figure
vq1 = interp1(x,v,xq);
plot(x,v,'o',xq,vq1,':.');
xlim([0 2*pi]);
title('(Default) Linear Interpolation');
```



Now evaluate `v` at the same points using the `'spline'` method.

```
figure
vq2 = interp1(x,v,xq,'spline');
plot(x,v,'o',xq,vq2,':.');
xlim([0 2*pi]);
title('Spline Interpolation');
```



Interpolation Without Specifying Points

Define a set of function values.

```
v = [0  1.41  2  1.41  0  -1.41  -2  -1.41 0];
```

Define a set of query points that fall between the default points, `1:9`. In this case, the default points are `1:9` because `v` contains `9` values.

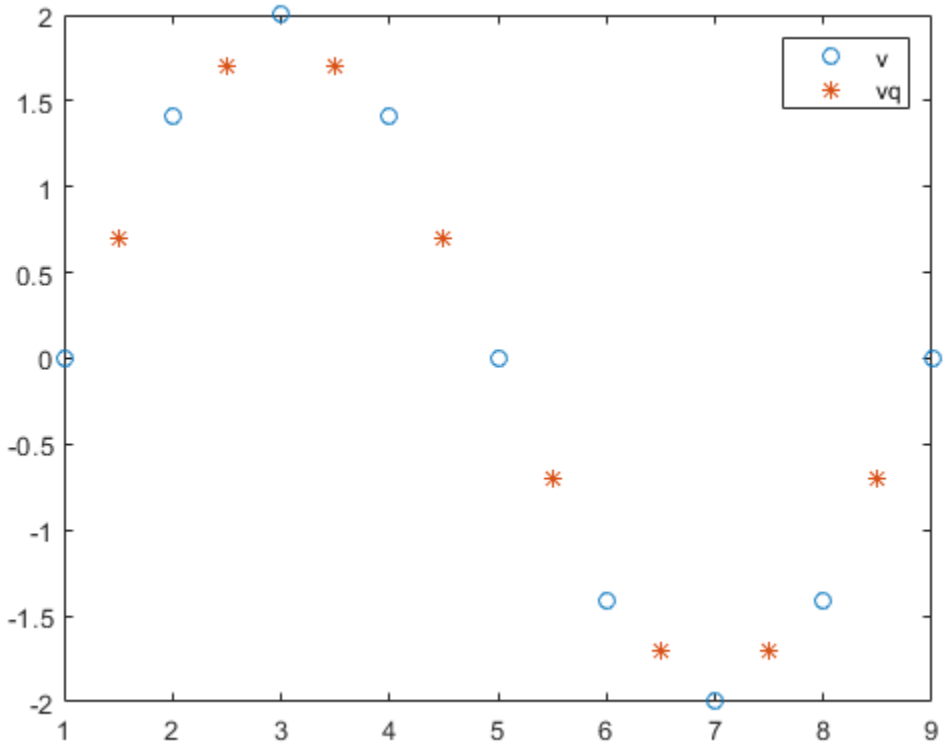
```
xq = 1.5:8.5;
```

Evaluate `v` at `xq`.

```
vq = interp1(v,xq);
```

Plot the result.

```
figure
plot((1:9),v,'o',xq,vq,'*');
legend('v','vq');
```



Interpolation of Complex Values

Define a set of sample points.

```
x = 1:10;
```

Define the values of the function, $v(x) = 5x + x^2 i$, at the sample points.

```
v = (5*x)+(x.^2*1i);
```

Define the query points to be a finer sampling over the range of `x`.

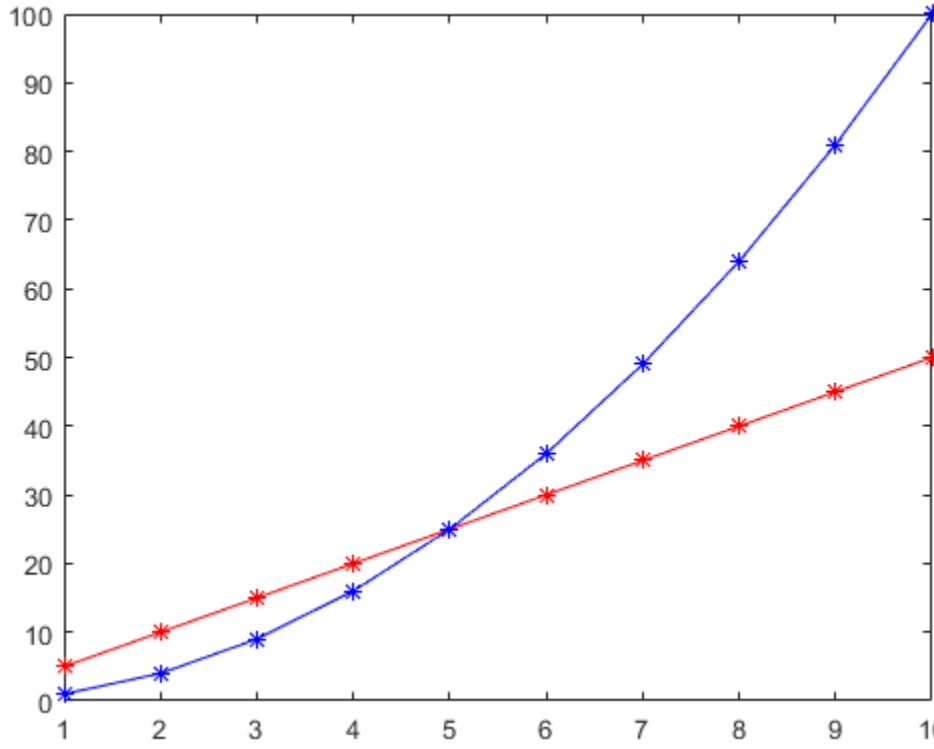
```
xq = 1:0.25:10;
```

Interpolate `v` at the query points.

```
vq = interp1(x,v,xq);
```

Plot the real part of the result in red and the imaginary part in blue.

```
figure
plot(x,real(v),'*r',xq,real(vq),'-r');
hold on
plot(x,imag(v),'*b',xq,imag(vq),'-b');
```



Interpolation of Dates and Times

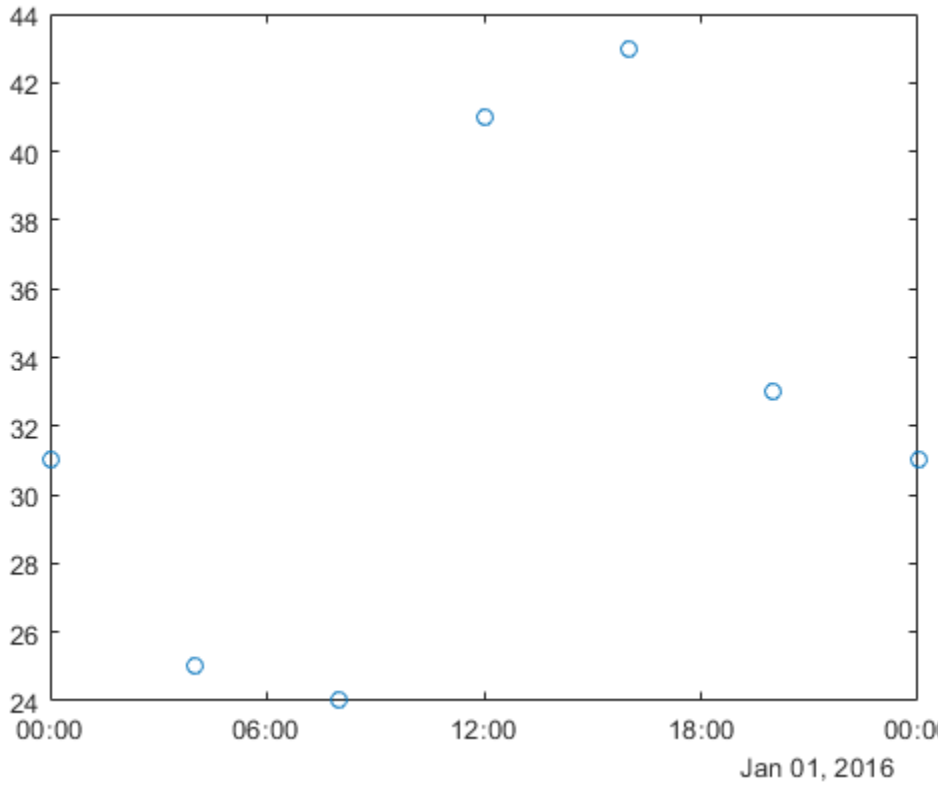
Interpolate time-stamped data points.

Consider a data set containing temperature readings that are measured every four hours. Create a table with one day's worth of data and plot the data.

```
x = (datetime(2016,1,1):hours(4):datetime(2016,1,2))';
x.Format = 'MMM dd, HH:mm';
T = [31 25 24 41 43 33 31]';
WeatherData = table(x,T,'VariableNames',{'Time','Temperature'})
```

```
WeatherData=7×2 table
        Time          Temperature
    _____________     ___________

    Jan 01, 00:00        31
    Jan 01, 04:00        25
    Jan 01, 08:00        24
    Jan 01, 12:00        41
    Jan 01, 16:00        43
    Jan 01, 20:00        33
    Jan 02, 00:00        31
```

```
plot(WeatherData.Time, WeatherData.Temperature, 'o')
```

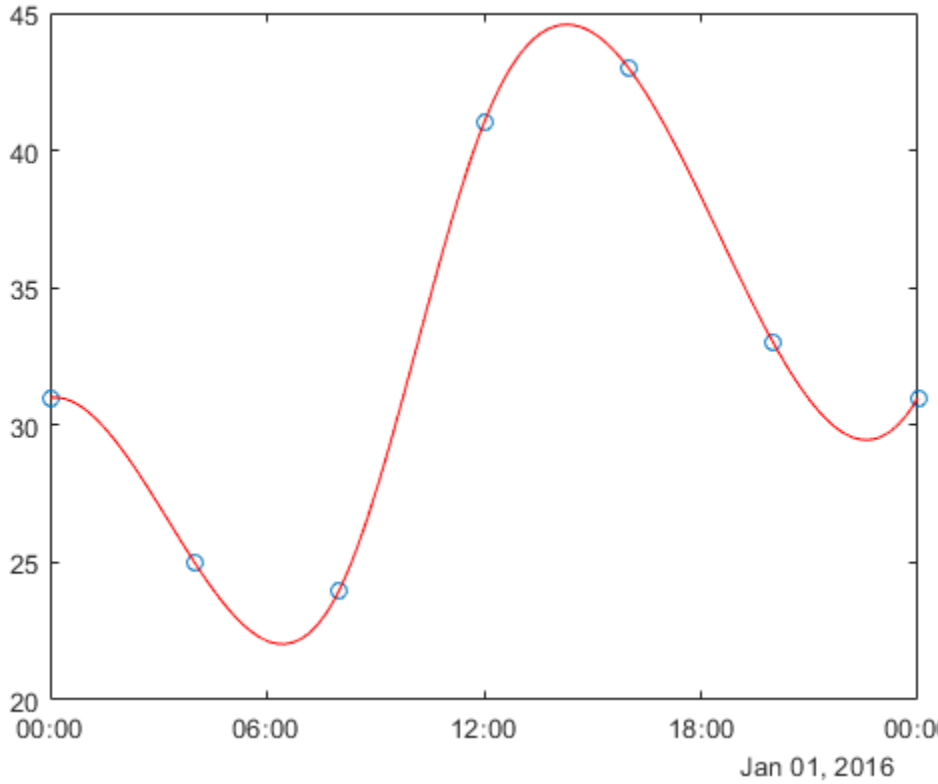


Interpolate the data set to predict the temperature reading during each minute of the day. Since the data is periodic, use the `'spline'` interpolation method.

```
xq = (datetime(2016,1,1):minutes(1):datetime(2016,1,2))';
V = interp1(WeatherData.Time, WeatherData.Temperature, xq, 'spline');
```

Plot the interpolated points.

```
hold on
plot(xq,V,'r')
```



Extrapolation Using Two Different Methods

Define the sample points, x, and corresponding sample values, v.

```
x = [1 2 3 4 5];
v = [12 16 31 10 6];
```

Specify the query points, xq, that extend beyond the domain of x.

```
xq = [0 0.5 1.5 5.5 6];
```

Evaluate v at xq using the 'pchip' method.

```
vq1 = interp1(x,v,xq,'pchip')
```

```
vq1 = 1×5

   19.3684   13.6316   13.2105    7.4800   12.5600
```

Next, evaluate v at xq using the 'linear' method.

```
vq2 = interp1(x,v,xq,'linear')
```

```
vq2 = 1×5

   NaN   NaN    14   NaN   NaN
```

Now, use the 'linear' method with the 'extrap' option.

```
vq3 = interp1(x,v,xq,'linear','extrap')
```

```
vq3 = 1×5

     8    10    14     4     2
```

'pchip' extrapolates by default, but 'linear' does not.

Designate Constant Value for All Queries Outside the Domain of x

Define the sample points, x, and corresponding sample values, v.

```
x = [-3 -2 -1 0 1 2 3];
v = 3*x.^2;
```

Specify the query points, `xq`, that extend beyond the domain of `x`.

```
xq = [-4 -2.5 -0.5 0.5 2.5 4];
```

Now evaluate `v` at `xq` using the `'pchip'` method and assign any values outside the domain of `x` to the value, `27`.

```
vq = interp1(x,v,xq,'pchip',27)
```

```
vq = 1×6

   27.0000   18.6563    0.9375    0.9375   18.6563   27.0000
```

Interpolate Multiple Sets of Data in One Pass

Define the sample points.

```
x = (-5:5)';
```

Sample three different parabolic functions at the points defined in `x`.

```
v1 = x.^2;
v2 = 2*x.^2 + 2;
v3 = 3*x.^2 + 4;
```

Create matrix `v`, whose columns are the vectors, `v1`, `v2`, and `v3`.

```
v = [v1 v2 v3];
```

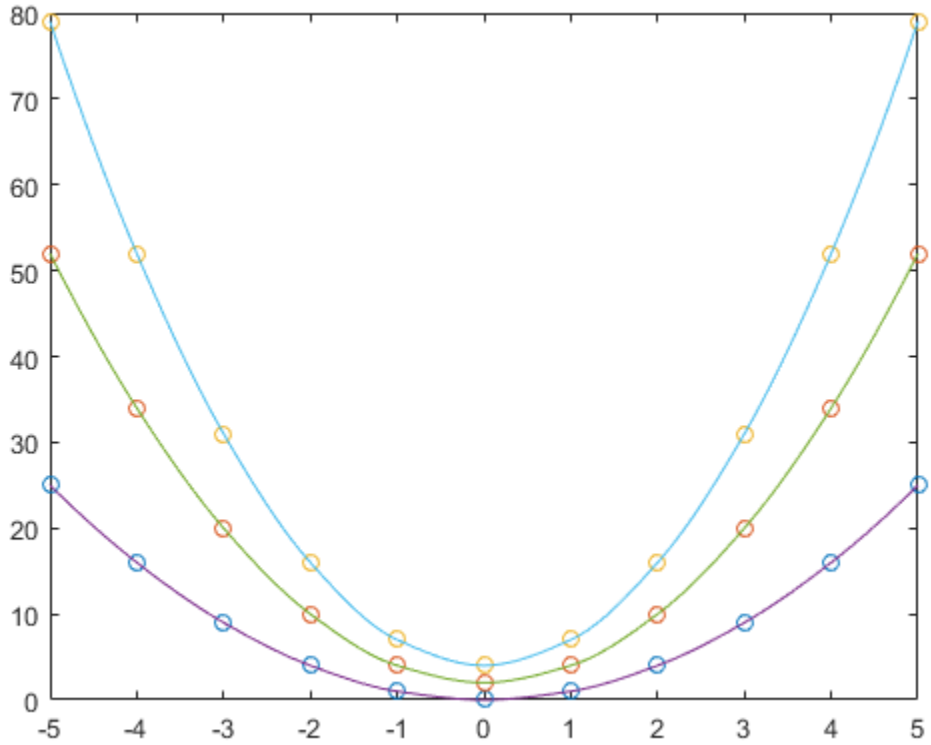
Define a set of query points, `xq`, to be a finer sampling over the range of `x`.

```
xq = -5:0.1:5;
```

Evaluate all three functions at `xq` and plot the results.

```
vq = interp1(x,v,xq,'pchip');
figure
plot(x,v,'o',xq,vq);
```

```
h = gca;
h.XTick = -5:5;
```



The circles in the plot represent `v`, and the solid lines represent `vq`.

Input Arguments

x — Sample points
vector

Sample points, specified as a row or column vector of real numbers. The values in `x` must be distinct. The length of `x` must conform to one of the following requirements:

- If `v` is a vector, then `length(x)` must equal `length(v)`.
- If `v` is an array, then `length(x)` must equal `size(v,1)`.

Example: `[1 2 3 4 5 6 7 8 9 10]`

Example: `1:10`

Example: `[3 7 11 15 19 23 27 31]'`

Data Types: `single` | `double` | `duration` | `datetime`

`v` — Sample values
vector | matrix | array

Sample values, specified as a vector, matrix, or array of real or complex numbers. If `v` is a matrix or an array, then each column contains a separate set of 1-D values.

If `v` contains complex numbers, then `interp1` interpolates the real and imaginary parts separately.

Example: `rand(1,10)`

Example: `rand(10,1)`

Example: `rand(10,3)`

Data Types: `single` | `double` | `duration` | `datetime`
Complex Number Support: Yes

`xq` — Query points
scalar | vector | matrix | array

Query points, specified as a scalar, vector, matrix, or array of real numbers.

Example: `5`

Example: `1:0.05:10`

Example: `(1:0.05:10)'`

Example: `[0 1 2 7.5 10]`

Data Types: `single` | `double` | `duration` | `datetime`

`method` — Interpolation method
`'linear'` (default) | `'nearest'` | `'next'` | `'previous'` | `'pchip'` | `'cubic'` | `'v5cubic'` | `'makima'` | `'spline'`

Interpolation method, specified as one of the options in this table.

Method	Description	Continuity	Comments
`'linear'`	Linear interpolation. The interpolated value at a query point is based on linear interpolation of the values at neighboring grid points in each respective dimension. This is the default interpolation method.	C^0	• Requires at least 2 points • Requires more memory and computation time than nearest neighbor
`'nearest'`	Nearest neighbor interpolation. The interpolated value at a query point is the value at the nearest sample grid point.	Discontinuous	• Requires at least 2 points • Modest memory requirements • Fastest computation time
`'next'`	Next neighbor interpolation. The interpolated value at a query point is the value at the next sample grid point.	Discontinuous	• Requires at least 2 points • Same memory requirements and computation time as `'nearest'`
`'previous'`	Previous neighbor interpolation. The interpolated value at a query point is the value at the previous sample grid point.	Discontinuous	• Requires at least 2 points • Same memory requirements and computation time as `'nearest'`
`'pchip'`	Shape-preserving piecewise cubic interpolation. The interpolated value at a query point is based on a shape-preserving piecewise cubic interpolation of the values at neighboring grid points.	C^1	• Requires at least 4 points • Requires more memory and computation time than `'linear'`

Method	Description	Continuity	Comments
`'cubic'` **Note** The behavior of `interp1(...,'cubic')` will change in a future release. In a future release, this method will perform cubic convolution.	Same as `'pchip'`.	C^1	This method currently returns the same result as `'pchip'`
`'v5cubic'`	Cubic convolution used in MATLAB 5.	C^1	Points must be uniformly spaced. `'cubic'` will replace `'v5cubic'` in a future release
`'makima'`	Modified Akima cubic Hermite interpolation. The interpolated value at a query point is based on a piecewise function of polynomials with degree at most three. The Akima formula is modified to avoid overshoots.	C^1	• Requires at least 2 points • Produces fewer undulations than `'spline'`, but does not flatten as aggressively as `'pchip'` • Computation is more expensive than `'pchip'`, but typically less than `'spline'` • Memory requirements are similar to those of `'spline'`

Method	Description	Continuity	Comments
`'spline'`	Spline interpolation using not-a-knot end conditions. The interpolated value at a query point is based on a cubic interpolation of the values at neighboring grid points in each respective dimension.	C^2	• Requires at least 4 points • Requires more memory and computation time than `'pchip'`

`extrapolation` — Extrapolation strategy
`'extrap'` | scalar value

Extrapolation strategy, specified as `'extrap'` or a real scalar value.

- Specify `'extrap'` when you want `interp1` to evaluate points outside the domain using the same method it uses for interpolation.
- Specify a scalar value when you want `interp1` to return a specific constant value for points outside the domain.

The default behavior depends on the input arguments:

- If you specify the `'pchip'`, `'spline'`, or `'makima'` interpolation methods, then the default behavior is `'extrap'`.
- All other interpolation methods return `NaN` by default for query points outside the domain.

Example: `'extrap'`

Example: 5

Data Types: `char` | `string` | `single` | `double`

Output Arguments

`vq` — Interpolated values
scalar | vector | matrix | array

Interpolated values, returned as a scalar, vector, matrix, or array. The size of `vq` depends on the shape of `v` and `xq`.

Shape of v	Shape of xq	Size of Vq	Example
Vector	Vector	`size(xq)`	If `size(v) = [1 100]` and `size(xq) = [1 500]`, then `size(vq) = [1 500]`.
Vector	Matrix or N-D Array	`size(xq)`	If `size(v) = [1 100]` and `size(xq) = [50 30]`, then `size(vq) = [50 30]`.
Matrix or N-D Array	Vector	`[length(xq) size(v,2),...,size(v,n)]`	If `size(v) = [100 3]` and `size(xq) = [1 500]`, then `size(vq) = [500 3]`.
Matrix or N-D Array	Matrix or N-D Array	`[size(xq,1),...,size(xq,n),... size(v,2),...,size(v,m)]`	If `size(v) = [4 5 6]` and `size(xq) = [2 3 7]`, then `size(vq) = [2 3 7 5 6]`.

pp — Piecewise polynomial
structure

Piecewise polynomial, returned as a structure that you can pass to the `ppval` function for evaluation.

More About

Akima and Spline Interpolation

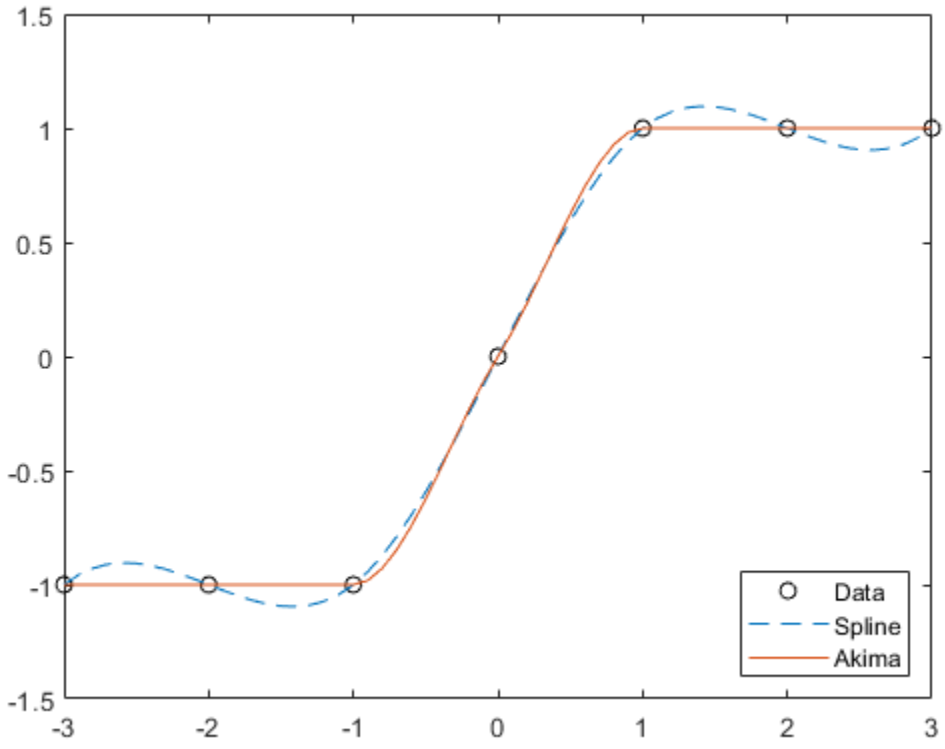
The **Akima algorithm** for one-dimensional interpolation, described in [1] on page 1-7827 and [2] on page 1-7827, performs cubic interpolation to produce piecewise polynomials with continuous first-order derivatives (C1). The algorithm preserves the slope and avoids undulations in flat regions. A flat region occurs whenever there are three or more consecutive collinear points, which the algorithm connects with a straight line. To ensure that the region between two data points is flat, insert an additional data point between those two points.

When two flat regions with different slopes meet, the modification made to the original Akima algorithm gives more weight to the side where the slope is closer to zero. This

modification gives priority to the side that is closer to horizontal, which is more intuitive and avoids the overshoot. (The original Akima algorithm gives equal weights to the points on both sides, thus evenly dividing the undulation.)

The **spline algorithm**, on the other hand, performs cubic interpolation to produce piecewise polynomials with continuous second-order derivatives (C2). The result is comparable to a regular polynomial interpolation, but is less susceptible to heavy oscillation between data points for high degrees. Still, this method can be susceptible to overshoots and oscillations between data points.

Compared to the spline algorithm, the Akima algorithm produces fewer undulations and is better suited to deal with quick changes between flat regions. This difference is illustrated below using test data that connects multiple flat regions.



References

[1] Akima, Hiroshi. "A new method of interpolation and smooth curve fitting based on local procedures." *Journal of the ACM (JACM)* , 17.4, 1970, pp. 589-602.

[2] Akima, Hiroshi. "A method of bivariate interpolation and smooth surface fitting based on local procedures." *Communications of the ACM* , 17.1, 1974, pp. 18-20.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support the `'cubic'` or `'makima'` interpolation methods.
- The input argument `x` (sample points) must be strictly increasing or strictly decreasing. Indices are not reordered.
- If the input argument `v` (sample values) is a variable-length vector (1-by-`:` or `:`-by-1), then the shape of the output `vq` matches the shape in MATLAB.

  If the input argument `v` is variable-size, is not a variable-length vector, and becomes a row vector at run time, then an error occurs.
- If the input argument `xq` (query points) is variable-size, is not a variable-length vector, and becomes a row or column vector at run time, then an error occurs.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- `x` must be a finite, increasing vector without repeating elements. `x` must be double or single.
- `v` must be an array of type double or single. The size of the first nonsingleton dimension of `v` must be the same as the length of `x`. `V` can be complex.
- `xq` must be an array of type double or single.
- `method` must be `'linear'`, `'nearest'`, `'v5cubic'`, or `'spline'`.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`griddedInterpolant` | `interp2` | `interp3` | `interpn`

Introduced before R2006a

interp1q

Quick 1-D linear interpolation

Note `interp1q` is not recommended. Use `interp1` instead.

Syntax

```
yi = interp1q(x,Y,xi)
```

Description

`yi = interp1q(x,Y,xi)` returns the value of the 1-D function `Y` at the points of column vector `xi` using linear interpolation. The vector `x` specifies the coordinates of the underlying interval. The length of output `yi` is equal to the length of `xi`.

For `interp1q` to work properly,

- `x` must be a monotonically increasing column vector.
- `Y` must be a column vector or matrix with `length(x)` rows.
- `xi` must be a column vector

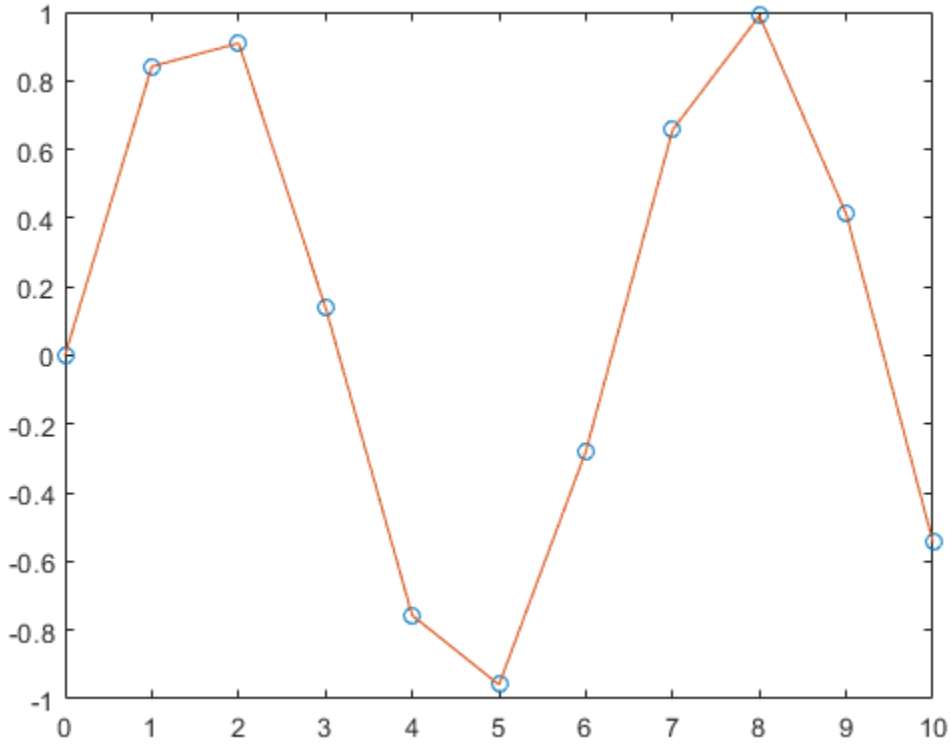
`interp1q` returns NaN for any values of `xi` that lie outside the coordinates in `x`. If `Y` is a matrix, then the interpolation is performed for each column of `Y`, in which case `yi` is `length(xi)-by-size(Y,2)`.

Examples

Linear Interpolation Using `interp1q`

Generate a coarse sine curve and interpolate over a finer abscissa.

```
x = (0:10)';
y = sin(x);
xi = (0:.25:10)';
yi = interp1q(x,y,xi);
plot(x,y,'o',xi,yi)
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Results might not match MATLAB when some `Y` values are `Inf` or `NaN`.

See Also

`interp1` | `interp2` | `interp3` | `interpn`

interp2

Interpolation for 2-D gridded data in meshgrid format

Syntax

```
Vq = interp2(X,Y,V,Xq,Yq)
Vq = interp2(V,Xq,Yq)
Vq = interp2(V)
Vq = interp2(V,k)

Vq = interp2( ___ ,method)
Vq = interp2( ___ ,method,extrapval)
```

Description

`Vq = interp2(X,Y,V,Xq,Yq)` returns interpolated values of a function of two variables at specific query points using linear interpolation. The results always pass through the original sampling of the function. `X` and `Y` contain the coordinates of the sample points. `V` contains the corresponding function values at each sample point. `Xq` and `Yq` contain the coordinates of the query points.

`Vq = interp2(V,Xq,Yq)` assumes a default grid of sample points. The default grid points cover the rectangular region, `X=1:n` and `Y=1:m`, where `[m,n] = size(V)`. Use this syntax when you want to conserve memory and are not concerned about the absolute distances between points.

`Vq = interp2(V)` returns the interpolated values on a refined grid formed by dividing the interval between sample values once in each dimension.

`Vq = interp2(V,k)` returns the interpolated values on a refined grid formed by repeatedly halving the intervals `k` times in each dimension. This results in `2^k-1` interpolated points between sample values.

`Vq = interp2( ___ ,method)` specifies an alternative interpolation method: `'linear'`, `'nearest'`, `'cubic'`, `'makima'`, or `'spline'`. The default method is `'linear'`.

`Vq = interp2( ___ ,method,extrapval)` also specifies `extrapval`, a scalar value that is assigned to all queries that lie outside the domain of the sample points.

If you omit the `extrapval` argument for queries outside the domain of the sample points, then based on the `method` argument `interp2` returns one of the following:

- Extrapolated values for the `'spline'` and `'makima'` methods
- `NaN` values for other interpolation methods

Examples

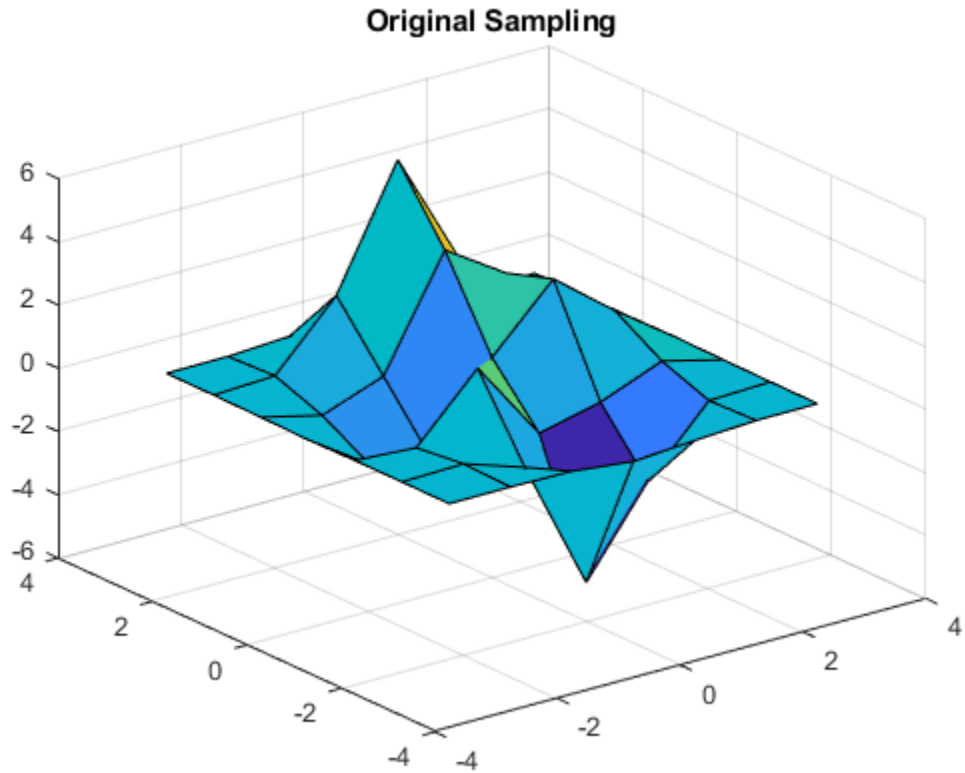
Interpolate Over a Grid Using Default Method

Coarsely sample the `peaks` function.

```
[X,Y] = meshgrid(-3:3);
V = peaks(X,Y);
```

Plot the coarse sampling.

```
figure
surf(X,Y,V)
title('Original Sampling');
```



Create the query grid with spacing of 0.25.

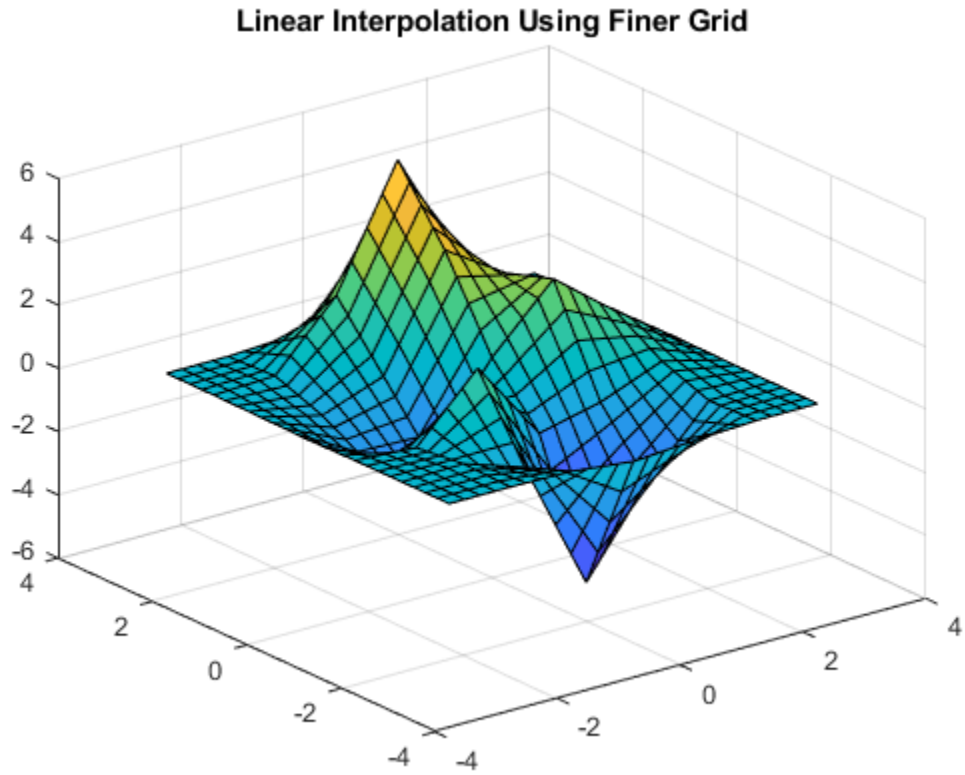
```
[Xq,Yq] = meshgrid(-3:0.25:3);
```

Interpolate at the query points.

```
Vq = interp2(X,Y,V,Xq,Yq);
```

Plot the result.

```
figure
surf(Xq,Yq,Vq);
title('Linear Interpolation Using Finer Grid');
```



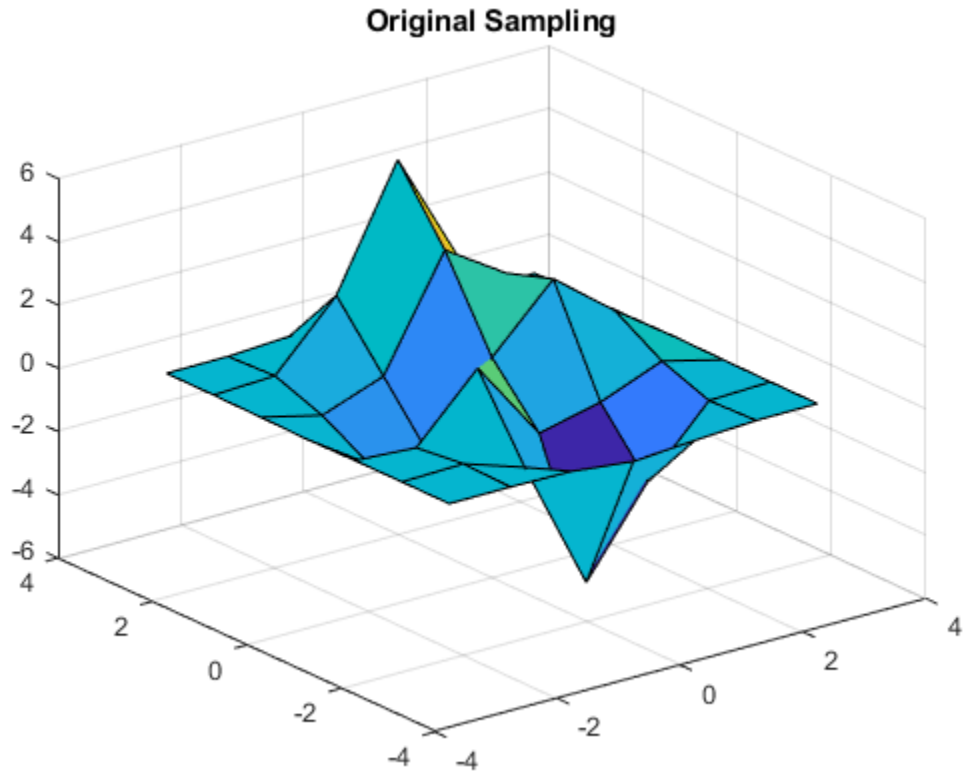
Interpolate Over a Grid Using Cubic Method

Coarsely sample the peaks function.

```
[X,Y] = meshgrid(-3:3);
V = peaks(7);
```

Plot the coarse sampling.

```
figure
surf(X,Y,V)
title('Original Sampling');
```



Create the query grid with spacing of 0.25.

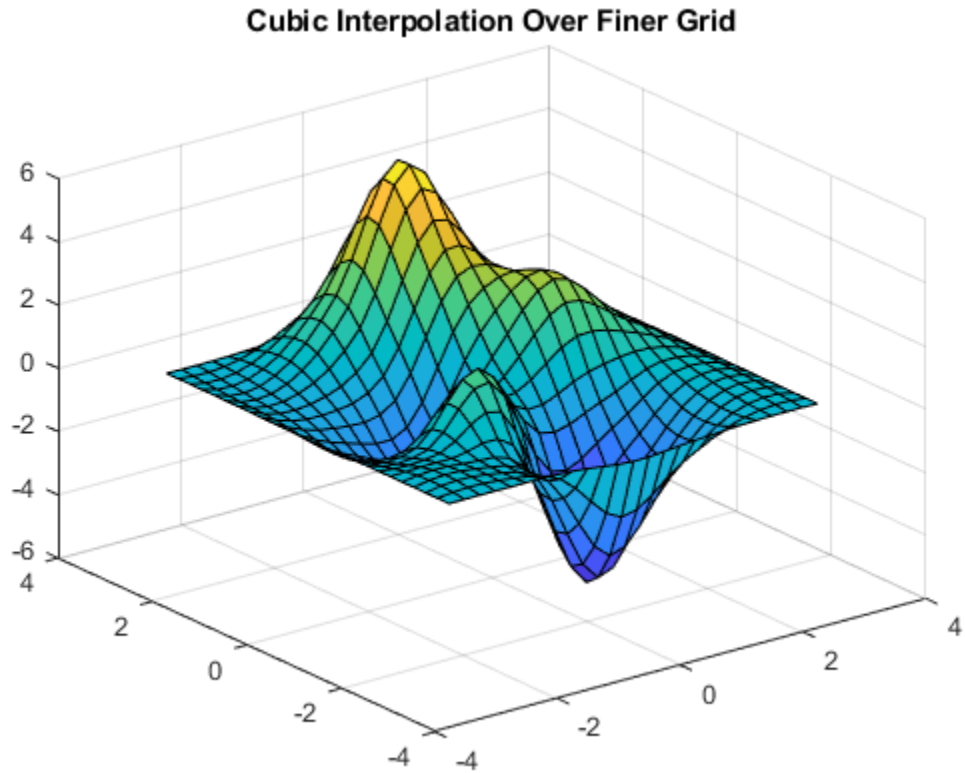
```
[Xq,Yq] = meshgrid(-3:0.25:3);
```

Interpolate at the query points, and specify cubic interpolation.

```
Vq = interp2(X,Y,V,Xq,Yq,'cubic');
```

Plot the result.

```
figure
surf(Xq,Yq,Vq);
title('Cubic Interpolation Over Finer Grid');
```



Refine Grayscale Image

Load some image data into the workspace.

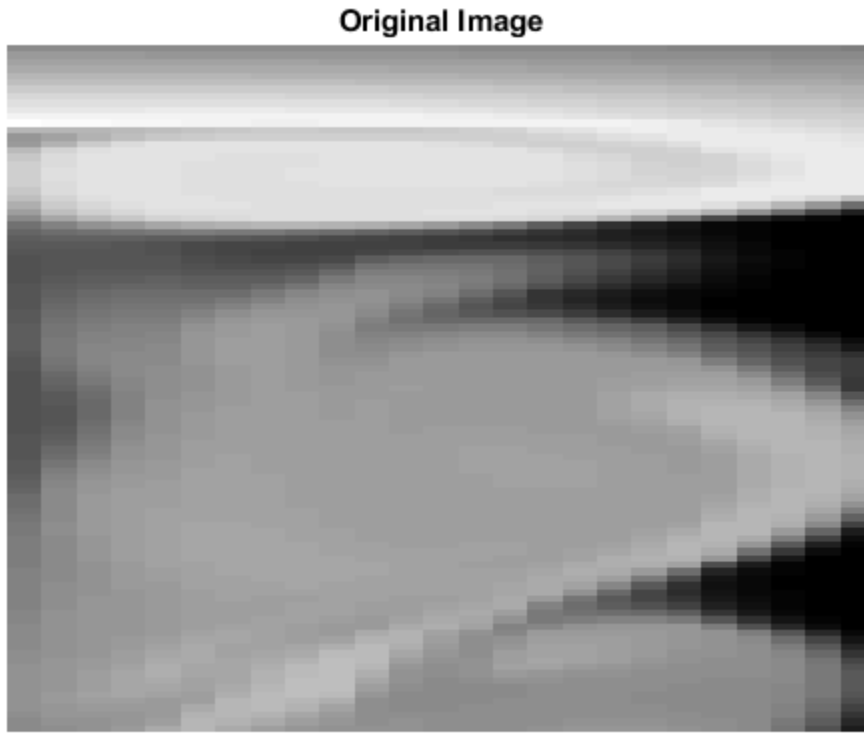
```
load flujet.mat
colormap gray
```

Isolate a small region of the image and cast it to single-precision.

```
V = single(X(200:300,1:25));
```

Display the image region.

```
imagesc(V);
axis off
title('Original Image')
```

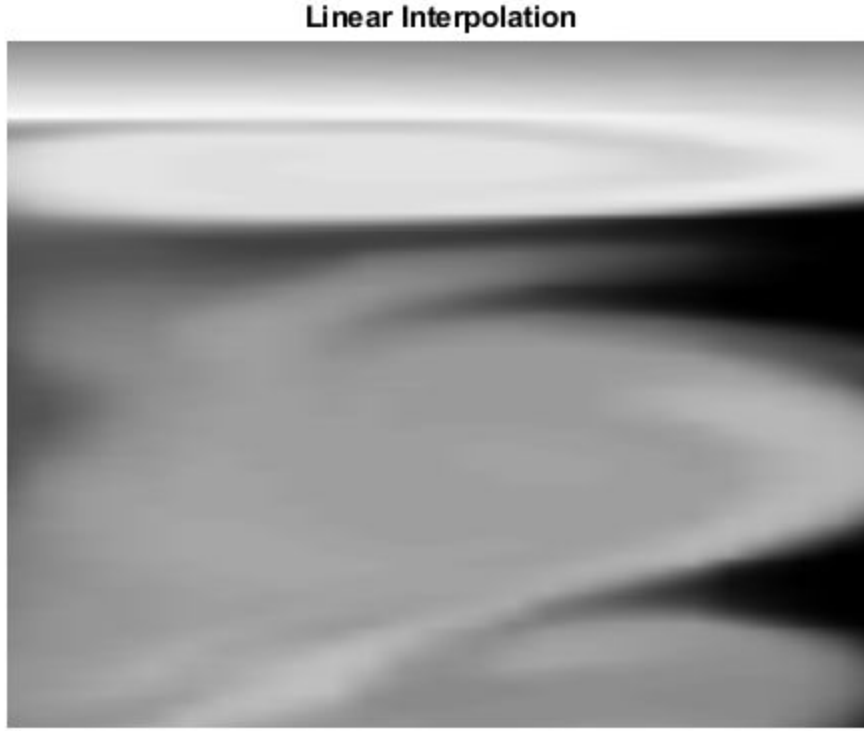


Insert interpolated values by repeatedly dividing the intervals between points of the refined grid five times in each dimension.

```
Vq = interp2(V,5);
```

Display the result.

```
imagesc(Vq);
axis off
title('Linear Interpolation')
```



Evaluate Outside the Domain of X and Y

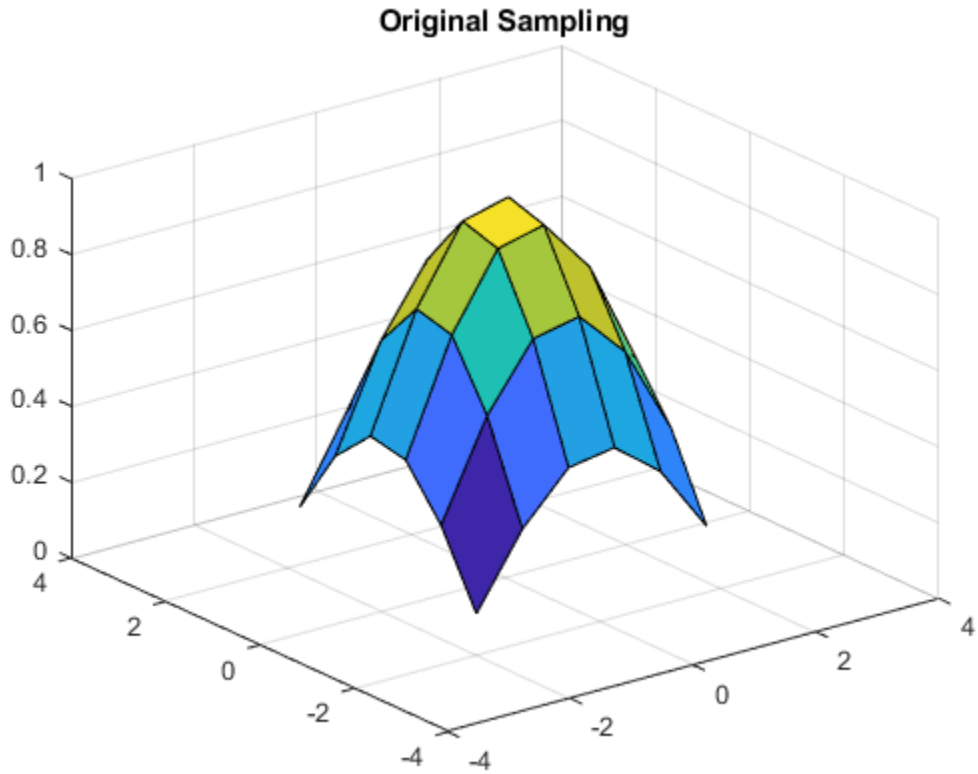
Coarsely sample a function over the range, `[-2, 2]` in both dimensions.

```
[X,Y] = meshgrid(-2:0.75:2);
R = sqrt(X.^2 + Y.^2)+ eps;
V = sin(R)./(R);
```

Plot the coarse sampling.

```
figure
surf(X,Y,V)
```

```
xlim([-4 4])
ylim([-4 4])
title('Original Sampling')
```



Create the query grid that extends beyond the domain of X and Y.

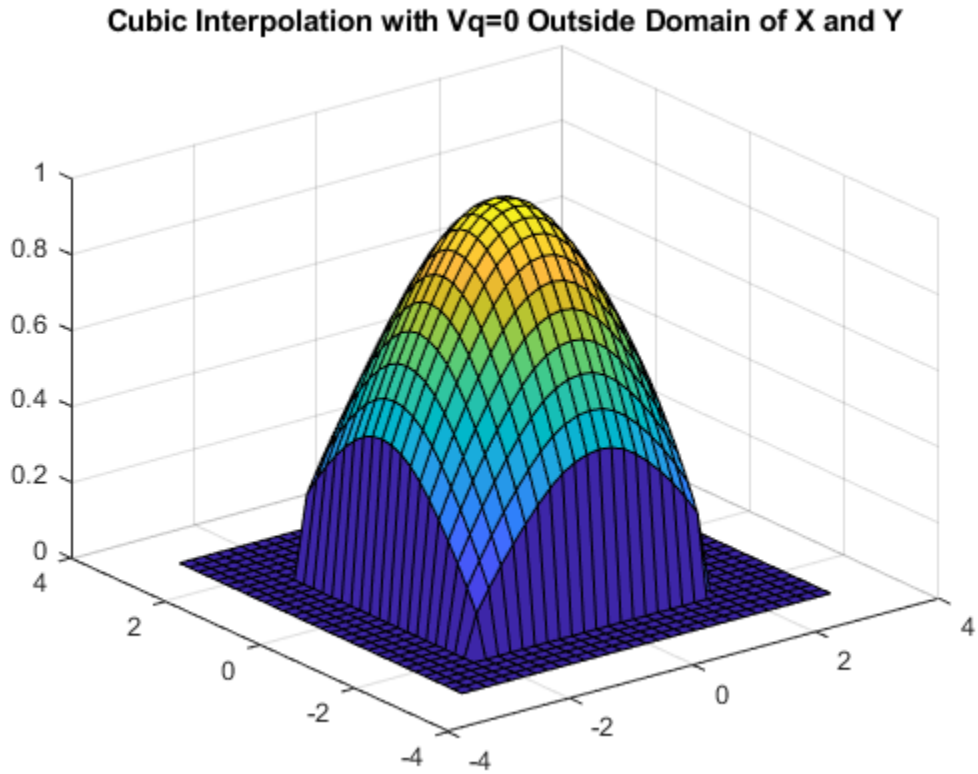
```
[Xq,Yq] = meshgrid(-3:0.2:3);
```

Perform cubic interpolation within the domain of X and Y, and assign all queries that fall outside to zero.

```
Vq = interp2(X,Y,V,Xq,Yq,'cubic',0);
```

Plot the result.

```
figure
surf(Xq,Yq,Vq)
title('Cubic Interpolation with Vq=0 Outside Domain of X and Y');
```



Input Arguments

X,Y — Sample grid points
matrices | vectors

Sample grid points, specified as real matrices or vectors. The sample grid points must be unique.

- If X and Y are matrices, then they contain the coordinates of a full grid (in meshgrid format) on page 1-7847. Use the `meshgrid` function to create the X and Y matrices together. Both matrices must be the same size.
- If X and Y are vectors, then they are treated as a grid vectors on page 1-7849. The values in both vectors must be strictly monotonic on page 1-7847 and increasing.

Note In a future release, `interp2` will not accept mixed combinations of row and column vectors for the sample and query grids. Instead, you must construct the full grid using `meshgrid`. Alternatively, if you have a large data set, you can use `griddedInterpolant` instead of `interp2`.

Example: `[X,Y] = meshgrid(1:30,-10:10)`

Data Types: `single` | `double`

`V` — Sample values
matrix

Sample values, specified as a real or complex matrix. The size requirements for `V` depend on the size of `X` and `Y`:

- If `X` and `Y` are matrices representing a full grid (in `meshgrid` format), then `V` must be the same size as `X` and `Y`.
- If `X` and `Y` are grid vectors, then `V` must be a matrix containing `length(Y)` rows and `length(X)` columns.

If `V` contains complex numbers, then `interp2` interpolates the real and imaginary parts separately.

Example: `rand(10,10)`

Data Types: `single` | `double`
Complex Number Support: Yes

`Xq,Yq` — Query points
scalars | vectors | matrices | arrays

Query points, specified as a real scalars, vectors, matrices, or arrays.

- If `Xq` and `Yq` are scalars, then they are the coordinates of a single query point.

- If `Xq` and `Yq` are vectors of different orientations, then `Xq` and `Yq` are treated as grid vectors.
- If `Xq` and `Yq` are vectors of the same size and orientation, then `Xq` and `Yq` are treated as scattered points on page 1-7850 in 2-D space.
- If `Xq` and `Yq` are matrices, then they represent either a full grid of query points (in `meshgrid` format) or scattered points.
- If `Xq` and `Yq` are N-D arrays, then they represent scattered points in 2-D space.

Note In a future release, `interp2` will not accept mixed combinations of row and column vectors for the sample and query grids. Instead, you must construct the full grid using `meshgrid`. Alternatively, if you have a large data set, you can use `griddedInterpolant` instead of `interp2`.

Example: `[Xq,Yq] = meshgrid((1:0.1:10),(-5:0.1:0))`

Data Types: `single` | `double`

k — Refinement factor

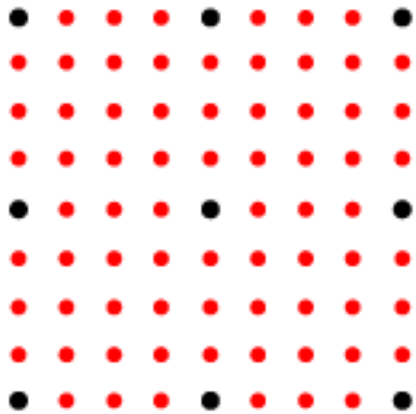
1 (default) | real, nonnegative, integer scalar

Refinement factor, specified as a real, nonnegative, integer scalar. This value specifies the number of times to repeatedly divide the intervals of the refined grid in each dimension. This results in `2^k-1` interpolated points between sample values.

If `k` is `0`, then `Vq` is the same as `V`.

`interp2(V,1)` is the same as `interp2(V)`.

The following illustration shows the placement of interpolated values (in red) among nine sample values (in black) for `k=2`.



Example: `interp2(V,2)`

Data Types: `single` | `double`

`method` — Interpolation method
`'linear'` (default) | `'nearest'` | `'cubic'` | `'spline'` | `'makima'`

Interpolation method, specified as one of the options in this table.

Method	Description	Continuity	Comments
`'linear'`	The interpolated value at a query point is based on linear interpolation of the values at neighboring grid points in each respective dimension. This is the default interpolation method.	C^0	• Requires at least two grid points in each dimension • Requires more memory than `'nearest'`
`'nearest'`	The interpolated value at a query point is the value at the nearest sample grid point.	Discontinuous	• Requires two grid points in each dimension. • Fastest computation with modest memory requirements

Method	Description	Continuity	Comments
`'cubic'`	The interpolated value at a query point is based on a cubic interpolation of the values at neighboring grid points in each respective dimension. The interpolation is based on a cubic convolution.	C^1	• Grid must have uniform spacing in each dimension, but the spacing does not have to be the same for all dimensions • Requires at least four points in each dimension • Requires more memory and computation time than `'linear'`
`'makima'`	Modified Akima cubic Hermite interpolation. The interpolated value at a query point is based on a piecewise function of polynomials with degree at most three evaluated using the values of neighboring grid points in each respective dimension. The Akima formula is modified to avoid overshoots.	C^1	• Requires at least 2 points in each dimension • Produces fewer undulations than `'spline'` • Computation time is typically less than `'spline'`, but the memory requirements are similar
`'spline'`	The interpolated value at a query point is based on a cubic interpolation of the values at neighboring grid points in each respective dimension. The interpolation is based on a cubic spline using not-a-knot end conditions.	C^2	• Requires four points in each dimension • Requires more memory and computation time than `'cubic'`

`extrapval` — Function value outside domain of X and Y
scalar

Function value outside domain of X and Y, specified as a real or complex scalar. `interp2` returns this constant value for all points outside the domain of X and Y.

Example: `5`

Example: `5+1i`

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

`Vq` — Interpolated values
scalar | vector | matrix

Interpolated values, returned as a real or complex scalar, vector, or matrix. The size and shape of `Vq` depends on the syntax you use and, in some cases, the size and value of the input arguments.

Syntaxes	Special Conditions	Size of Vq	Example

`interp2(X,Y,V,Xq,Yq)`
`interp2(V,Xq,Yq)`
and variations of these syntaxes that include `method` or `extrapval`
	`Xq`, `Yq` are scalars	Scalar	`size(Vq) = [1 1]` when you pass `Xq` and `Yq` as scalars.

Same as above	`Xq`, `Yq` are vectors of the same size and orientation	Vector of same size and orientation as `Xq` and `Yq`	If `size(Xq) = [100 1]` and `size(Yq) = [100 1]`, then `size(Vq) = [100 1]`.
Same as above	`Xq`, `Yq` are vectors of mixed orientation	Matrix in which the number of rows is `length(Yq)`, and the number of columns is `length(Xq)`	If `size(Xq) = [1 100]` and `size(Yq) = [50 1]`, then `size(Vq) = [50 100]`.
Same as above	`Xq`, `Yq` are matrices or arrays of the same size	Matrix or array of the same size as `Xq` and `Yq`	If `size(Xq) = [50 25]` and `size(Yq) = [50 25]`, then `size(Vq) = [50 25]`.

Syntaxes	Special Conditions	Size of Vq	Example
`interp2(V,k)` and variations of this syntax that include `method` or `extrapval`	None	Matrix in which the number of rows is: `2^k * (size(V,1)-1)+1`, and the number of columns is: `2^k * (size(V,2)-1)+1`	If `size(V) = [10 20]` and `k = 2`, then `size(Vq) = [37 77]`.

More About

Strictly Monotonic

A set of values that are always increasing or decreasing, without reversals. For example, the sequence, `a = [2 4 6 8]` is strictly monotonic and increasing. The sequence, `b = [2 4 4 6 8]` is not strictly monotonic because there is no change in value between `b(2)` and `b(3)`. The sequence, `c = [2 4 6 8 6]` contains a reversal between `c(4)` and `c(5)`, so it is not monotonic at all.

Full Grid (in meshgrid Format)

For `interp2`, the full grid is a pair of matrices whose elements represent a grid of points over a rectangular region. One matrix contains the *x*-coordinates, and the other matrix contains the *y*-coordinates. The values in the *x*-matrix are strictly monotonic on page 1-7847 and increasing along the rows. The values along its columns are constant. The values in the *y*-matrix are strictly monotonic and increasing along the columns. The values along its rows are constant. Use the `meshgrid` function to create a full grid that you can pass to `interp2`.

For example, the following code creates a full grid for the region, $-1 \le x \le 3$ and $1 \le y \le 4$:

```
[X,Y] = meshgrid(-1:3,(1:4))
```

```
X =

    -1     0     1     2     3
    -1     0     1     2     3
```

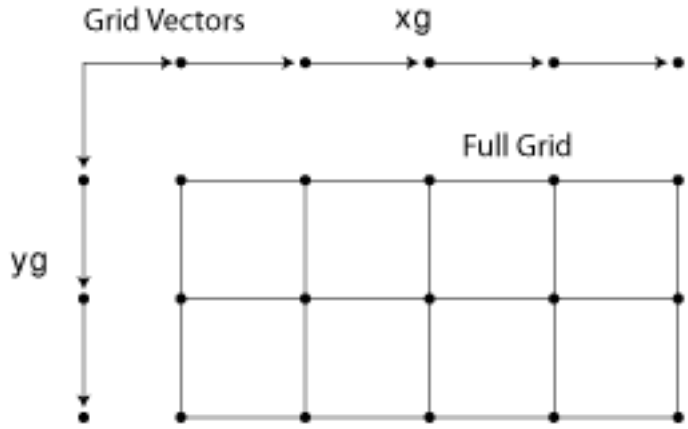
```
    -1     0     1     2     3
    -1     0     1     2     3

Y =

     1     1     1     1     1
     2     2     2     2     2
     3     3     3     3     3
     4     4     4     4     4
```

Grid vectors are a more compact format to represent a grid than the full grid. The relation between the two formats and the matrix of sample values `V` is

```
[X_grid, Y_grid] = meshgrid(xg,yg)
```

Grid Vectors

For `interp2`, grid vectors consist of a pair of vectors that define the *x*- and *y*-coordinates in a grid. The row vector defines *x*-coordinates, and the column vector defines *y*-coordinates.



For example, the following code creates the grid vectors that specify the region, $-1 \le x \le 3$ and $1 \le y \le 4$:

```
x = -1:3;
y = (1:4)';
```

Scattered Points

For `interp2`, scattered points consist of a pair of arrays that define a collection of points scattered in 2-D space. One array contains the *x*-coordinates, and the other contains the *y*-coordinates.

For example, the following code specifies the points, (2,7), (5,3), (4,1), and (10,9):

```
x = [2 5; 4 10];
y = [7 3; 1 9];
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `Xq` and `Yq` must be the same size. Use `meshgrid` to evaluate on a grid.
- For best results, provide `X` and `Y` as vectors.
- Code generation does not support the `'makima'` interpolation method.
- For the `'cubic'` interpolation method, if the grid does not have uniform spacing, an error results. In this case, use the `'spline'` interpolation method.
- For best results when you use the `'spline'` interpolation method:
  - Use `meshgrid` to create the inputs `Xq` and `Yq`.
  - Use a small number of interpolation points relative to the dimensions of `V`. Interpolating over a large set of scattered points can be inefficient.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- `V` must be a double or single 2-D array. `V` can be real or complex. `V` cannot be a vector.
- `X` and `Y` must:
  - Have the same type (double or single).
  - Be finite vectors or 2-D arrays with increasing and nonrepeating elements in corresponding dimensions.
  - Align with cartesian axes when `X` and `Y` are nonvector 2-D arrays (as if they were produced by `meshgrid`).
  - Have dimensions consistent with `V`.

- `Xq` and `Yq` must be vectors or arrays of the same type (double or single). If `Xq` and `Yq` are arrays, then they must have the same size. If they are vectors with different lengths, then they must have different orientations.
- `method` must be `'linear'`, `'nearest'`, or `'cubic'`.
- The extrapolation for the out-of-boundary input is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`griddata` | `griddedInterpolant` | `interp1` | `interp3` | `interpn` | `meshgrid` | `scatteredInterpolant`

Introduced before R2006a

interp3

Interpolation for 3-D gridded data in meshgrid format

Syntax

```
Vq = interp3(X,Y,Z,V,Xq,Yq,Zq)
Vq = interp3(V,Xq,Yq,Zq)
Vq = interp3(V)
Vq = interp3(V,k)

Vq = interp3( ___ ,method)
Vq = interp3( ___ ,method,extrapval)
```

Description

`Vq = interp3(X,Y,Z,V,Xq,Yq,Zq)` returns interpolated values of a function of three variables at specific query points using linear interpolation. The results always pass through the original sampling of the function. `X`, `Y`, and `Z` contain the coordinates of the sample points. `V` contains the corresponding function values at each sample point. `Xq`, `Yq`, and `Zq` contain the coordinates of the query points.

`Vq = interp3(V,Xq,Yq,Zq)` assumes a default grid of sample points. The default grid points cover the region, `X=1:n`, `Y=1:m`, `Z=1:p`, where `[m,n,p] = size(V)`. Use this syntax when you want to conserve memory and are not concerned about the absolute distances between points.

`Vq = interp3(V)` returns the interpolated values on a refined grid formed by dividing the interval between sample values once in each dimension.

`Vq = interp3(V,k)` returns the interpolated values on a refined grid formed by repeatedly halving the intervals `k` times in each dimension. This results in `2^k-1` interpolated points between sample values.

`Vq = interp3( ___ ,method)` specifies an alternative interpolation method: `'linear'`, `'nearest'`, `'cubic'`, `'makima'`, or `'spline'`. The default method is `'linear'`.

`Vq = interp3( ___ ,method,extrapval)` also specifies `extrapval`, a scalar value that is assigned to all queries that lie outside the domain of the sample points.

If you omit the `extrapval` argument for queries outside the domain of the sample points, then based on the `method` argument `interp3` returns one of the following:

- The extrapolated values for the `'spline'` and `'makima'` methods
- `NaN` values for other interpolation methods

Examples

Interpolate Using Default Method

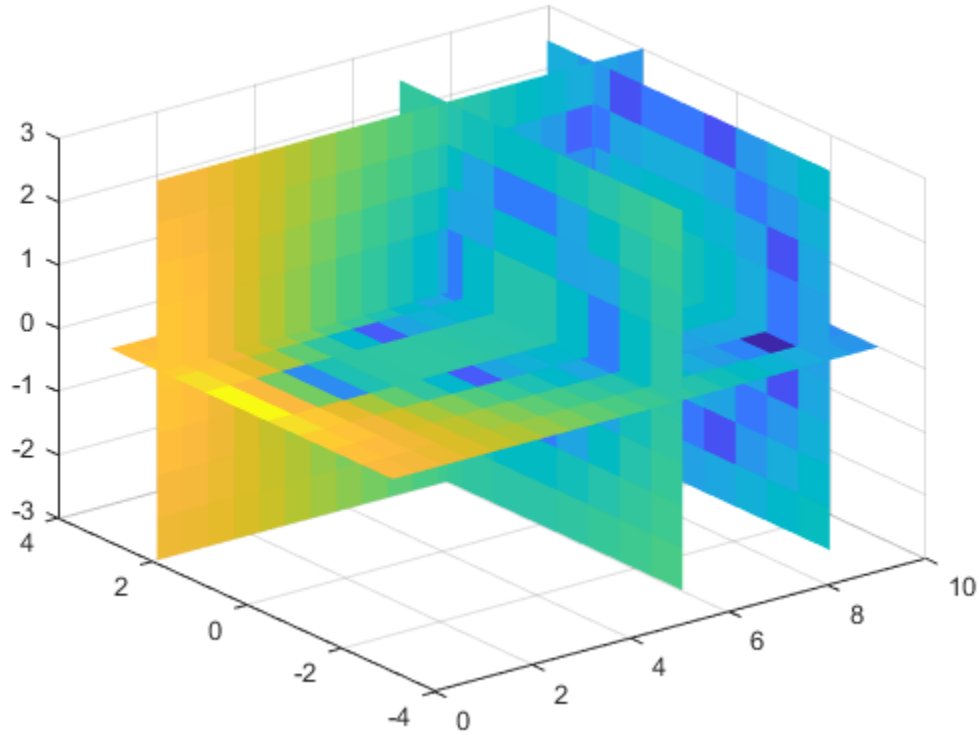
Load the points and values of the flow function, sampled at 10 points in each dimension.

```
[X,Y,Z,V] = flow(10);
```

The `flow` function returns the grid in the arrays, `X`, `Y`, `Z`. The grid covers the region, $0.1 \le X \le 10$, $-3 \le Y \le 3$, $-3 \le Z \le 3$, and the spacing is $\Delta X = 0.5$, $\Delta Y = 0.7$, and $\Delta Z = 0.7$.

Now, plot slices through the volume of the sample at: `X=6`, `X=9`, `Y=2`, and `Z=0`.

```
figure
slice(X,Y,Z,V,[6 9],2,0);
shading flat
```

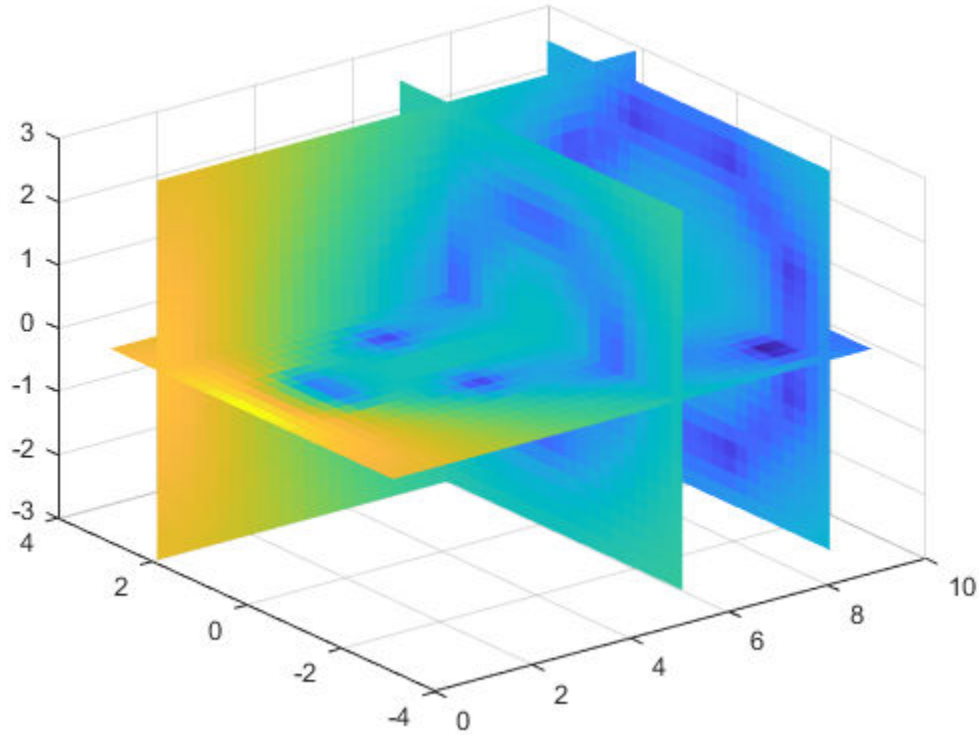


Create a query grid with spacing of 0.25.

```
[Xq,Yq,Zq] = meshgrid(.1:.25:10,-3:.25:3,-3:.25:3);
```

Interpolate at the points in the query grid and plot the results using the same slice planes.

```
Vq = interp3(X,Y,Z,V,Xq,Yq,Zq);
figure
slice(Xq,Yq,Zq,Vq,[6 9],2,0);
shading flat
```



Interpolate Using Cubic Method

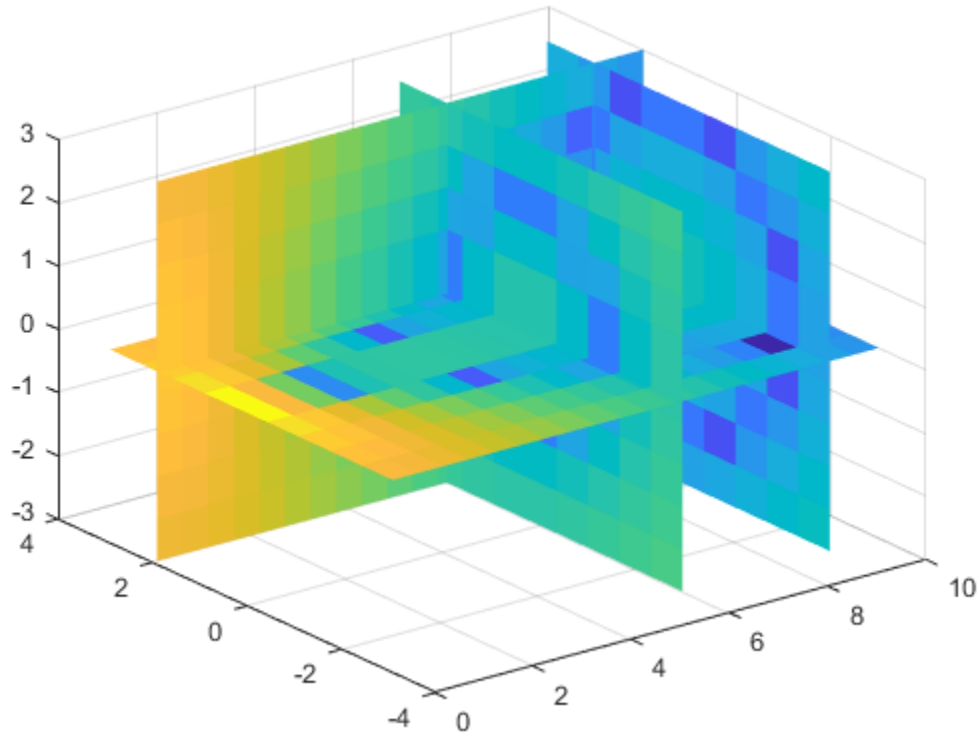
Load the points and values of the flow function, sampled at 10 points in each dimension.

```
[X,Y,Z,V] = flow(10);
```

The `flow` function returns the grid in the arrays, `X`, `Y`, `Z`. The grid covers the region, $0.1 \le X \le 10$, $-3 \le Y \le 3$, $-3 \le Z \le 3$, and the spacing is $\Delta X = 0.5$, $\Delta Y = 0.7$, and $\Delta Z = 0.7$.

Plot slices through the volume of the sample at: `X=6`, `X=9`, `Y=2`, and `Z =0`.

```
figure
slice(X,Y,Z,V,[6 9],2,0);
shading flat
```

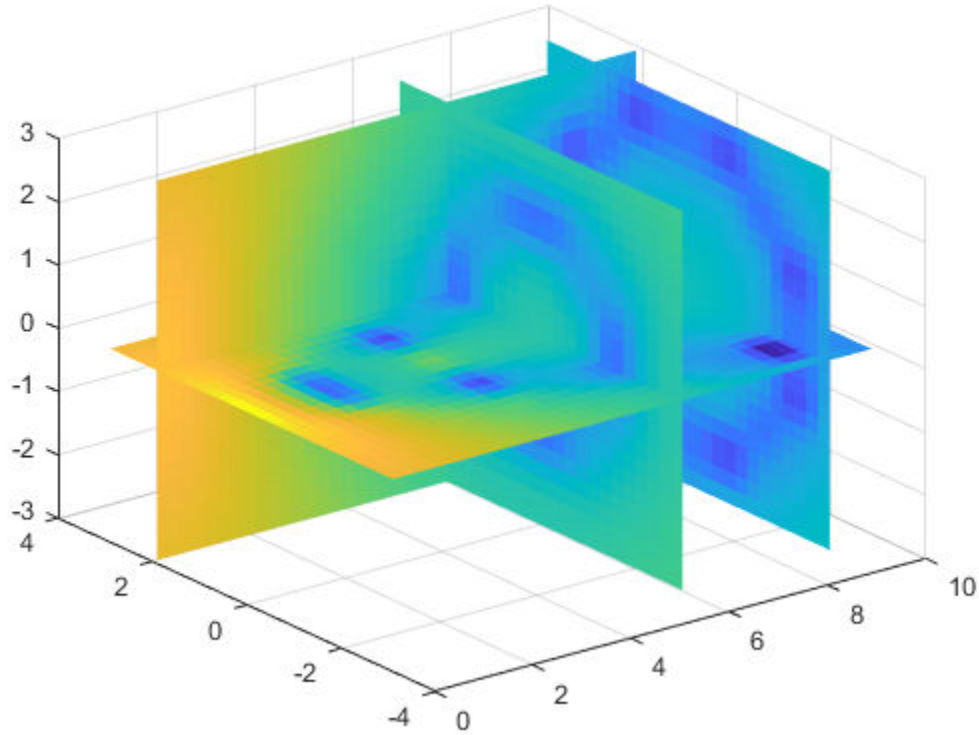


Create a query grid with spacing of 0.25.

```
[Xq,Yq,Zq] = meshgrid(.1:.25:10,-3:.25:3,-3:.25:3);
```

Interpolate at the points in the query grid using the `'cubic'` interpolation method. Then plot the results.

```
Vq = interp3(X,Y,Z,V,Xq,Yq,Zq,'cubic');
figure
slice(Xq,Yq,Zq,Vq,[6 9],2,0);
shading flat
```



Evaluate Outside the Domain of X, Y, and Z

Create the grid vectors, `x`, `y`, and `z`. These vectors define the points associated with values in `V`.

```
x = 1:100;
y = (1:50)';
z = 1:30;
```

Define the sample values to be a 50-by-100-by-30 random number array, `V`. Use the `gallery` function to create the array.

```
V = gallery('uniformdata',50,100,30,0);
```

Evaluate V at three points outside the domain of x, y, and z. Specify `extrapval = -1`.

```
xq = [0 0 0];
yq = [0 0 51];
zq = [0 101 102];
vq = interp3(x,y,z,V,xq,yq,zq,'linear',-1)
```

```
vq = 1×3

    -1    -1    -1
```

All three points evaluate to `-1` because they are outside the domain of x, y, and z.

Input Arguments

X,Y,Z — Sample grid points
arrays | vectors

Sample grid points, specified as real arrays or vectors. The sample grid points must be unique.

- If X, Y, and Z are arrays, then they contain the coordinates of a full grid (in meshgrid format) on page 1-7865. Use the `meshgrid` function to create the X, Y, and Z arrays together. These arrays must be the same size.
- If X, Y, and Z are vectors, then they are treated as a grid vectors on page 1-7865. The values in these vectors must be strictly monotonic on page 1-7865 and increasing.

Note In a future release, `interp3` will not accept mixed combinations of row and column vectors for the sample and query grids. Instead, you must construct the full grid using `meshgrid`. Alternatively, if you have a large data set, you can use `griddedInterpolant` instead of `interp3`.

Example: `[X,Y,Z] = meshgrid(1:30,-10:10,1:5)`

Data Types: `single` | `double`

V — Sample values
array

Sample values, specified as a real or complex array. The size requirements for V depend on the size of X, Y, and Z:

- If X, Y, and Z are arrays representing a full grid (in meshgrid format), then the size of V matches the size of X, Y, or Z .
- If X, Y, and Z are grid vectors, then size(V) = [length(Y) length(X) length(Z)].

If V contains complex numbers, then interp3 interpolates the real and imaginary parts separately.

Example: rand(10,10,10)

Data Types: single | double
Complex Number Support: Yes

Xq,Yq,Zq — Query points
scalars | vectors | arrays

Query points, specified as a real scalars, vectors, or arrays.

- If Xq, Yq, and Zq are scalars, then they are the coordinates of a single query point in R^3.
- If Xq, Yq, and Zq are vectors of different orientations, then Xq, Yq, and Zq are treated as grid vectors in R^3.
- If Xq, Yq, and Zq are vectors of the same size and orientation, then Xq, Yq, and Zq are treated as scattered points on page 1-7866 in R^3.
- If Xq, Yq, and Zq are arrays of the same size, then they represent either a full grid of query points (in meshgrid format) or scattered points in R^3.

Note In a future release, interp3 will not accept mixed combinations of row and column vectors for the sample and query grids. Instead, you must construct the full grid using meshgrid. Alternatively, if you have a large data set, you can use griddedInterpolant instead of interp3.

Example: [Xq,Yq,Zq] = meshgrid((1:0.1:10),(-5:0.1:0),3:5)

Data Types: `single` | `double`

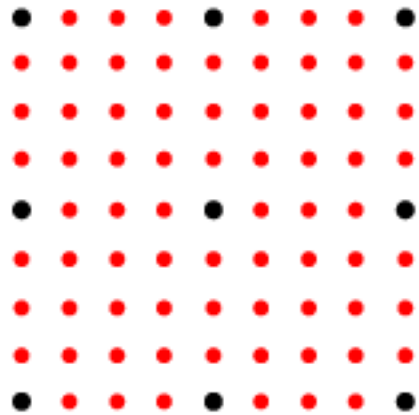
`k` — Refinement factor
1 (default) | real, nonnegative, integer scalar

Refinement factor, specified as a real, nonnegative, integer scalar. This value specifies the number of times to repeatedly divide the intervals of the refined grid in each dimension. This results in `2^k-1` interpolated points between sample values.

If `k` is `0`, then `Vq` is the same as `V`.

`interp3(V,1)` is the same as `interp3(V)`.

The following illustration depicts `k=2` in one plane of R^3. There are 72 interpolated values in red and 9 sample values in black.



Example: `interp3(V,2)`

Data Types: `single` | `double`

`method` — Interpolation method
`'linear'` (default) | `'nearest'` | `'cubic'` | `'spline'` | `'makima'`

Interpolation method, specified as one of the options in this table.

Method	Description	Continuity	Comments
`'linear'`	The interpolated value at a query point is based on linear interpolation of the values at neighboring grid points in each respective dimension. This is the default interpolation method.	C^0	• Requires at least two grid points in each dimension • Requires more memory than `'nearest'`
`'nearest'`	The interpolated value at a query point is the value at the nearest sample grid point.	Discontinuous	• Requires two grid points in each dimension • Fastest computation with modest memory requirements
`'cubic'`	The interpolated value at a query point is based on a cubic interpolation of the values at neighboring grid points in each respective dimension. The interpolation is based on a cubic convolution.	C^1	• Grid must have uniform spacing in each dimension, but the spacing does not have to be the same for all dimensions • Requires at least four points in each dimension • Requires more memory and computation time than `'linear'`
`'makima'`	Modified Akima cubic Hermite interpolation. The interpolated value at a query point is based on a piecewise function of polynomials with degree at most three evaluated using the values of neighboring grid points in each respective dimension. The Akima formula is modified to avoid overshoots.	C^1	• Requires at least 2 points in each dimension • Produces fewer undulations than `'spline'` • Computation time is typically less than `'spline'`, but the memory requirements are similar

Method	Description	Continuity	Comments
`'spline'`	The interpolated value at a query point is based on a cubic interpolation of the values at neighboring grid points in each respective dimension. The interpolation is based on a cubic spline using not-a-knot end conditions.	C^2	• Requires four points in each dimension • Requires more memory and computation time than `'cubic'`

`extrapval` — Function value outside domain of X, Y, and Z
scalar

Function value outside domain of `X`, `Y`, and `Z`, specified as a real or complex scalar. `interp3` returns this constant value for all points outside the domain of `X`, `Y`, and `Z`.

Example: `5`

Example: `5+1i`

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

`Vq` — Interpolated values
scalar | vector | array

Interpolated values, returned as a real or complex scalar, vector, or array. The size and shape of `Vq` depends on the syntax you use and, in some cases, the size and value of the input arguments.

Syntaxes	Special Conditions	Size of Vq	Example

`interp3(X,Y,Z,V,Xq,Yq,Zq)`
`interp3(V,Xq,Yq,Zq)`
and variations of these syntaxes that include `method` or `extrapval`
	`Xq`, `Yq`, and `Zq` are scalars.	Scalar	`size(Vq) = [1 1]` when you pass `Xq`, `Yq`, and `Zq` as scalars.

Same as above	`Xq`, `Yq`, and `Zq` are vectors of the same size and orientation.	Vector of same size and orientation as `Xq`, `Yq`, and `Zq`	If `size(Xq) = [100 1]`, and `size(Yq) = [100 1]`, and `size(Zq) = [100 1]`, then `size(Vq) = [100 1]`.
Same as above	`Xq`, `Yq`, and `Zq` are vectors of mixed orientation.	`size(Vq) = [length(Y) length(X) length(Z)]`	If `size(Xq) = [1 100]`, and `size(Yq) = [50 1]`, and `size(Zq) = [1 5]`, then `size(Vq) = [50 100 5]`.
Same as above	`Xq`, `Yq`, and `Zq` are arrays of the same size.	Array of the same size as `Xq`, `Yq`, and `Zq`	If `size(Xq) = [50 25]`, and `size(Yq) = [50 25]`, and `size(Zq) = [50 25]`, then `size(Vq) = [50 25]`.
`interp3(V,k)` and variations of this syntax that include `method` or `extrapval`	None	Array in which the length of the `i`th dimension is `2^k * (size(V,i)-1)+1`	If `size(V) = [10 12 5]`, and `k = 3`, then `size(Vq) = [73 89 33]`.

More About

Strictly Monotonic

A set of values that are always increasing or decreasing, without reversals. For example, the sequence, `a = [2 4 6 8]` is strictly monotonic and increasing. The sequence, `b = [2 4 4 6 8]` is not strictly monotonic because there is no change in value between `b(2)` and `b(3)`. The sequence, `c = [2 4 6 8 6]` contains a reversal between `c(4)` and `c(5)`, so it is not monotonic at all.

Full Grid (in meshgrid Format)

For `interp3`, a full grid consists of three arrays whose elements represent a grid of points that define a region in R^3. The first array contains the *x*-coordinates, the second array contains the *y*-coordinates, and the third array contains the *z*-coordinates. The values in each array vary along a single dimension and are constant along the other dimensions.

The values in the *x*-array are strictly monotonic on page 1-7865, increasing, and vary along the second dimension. The values in the *y*-array are strictly monotonic, increasing, and vary along the first dimension. The values in the *z*-array are strictly monotonic, increasing, and vary along the third dimension. Use the `meshgrid` function to create a full grid that you can pass to `interp3`.

Grid Vectors

For `interp3`, grid vectors consist of three vectors of mixed-orientation that define the points on a grid in R^3.

For example, the following code creates the grid vectors for the region, $1 \le x \le 3$, $4 \le y \le 5$, and $6 \le z \le 8$:

```
x = 1:3;
y = (4:5)';
z = 6:8;
```

Scattered Points

For `interp3`, scattered points consist of three arrays or vectors, `Xq`, `Yq`, and `Zq`, that define a collection of points scattered in R^3. The ith array contains the coordinates in the ith dimension.

For example, the following code specifies the points, (1, 19, 10), (6, 40, 1), (15, 33, 22), and (0, 61, 13).

```
Xq = [1 6; 15 0];
Yq = [19 40; 33 61];
Zq = [10 1; 22 13];
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `Xq`, `Yq`, and `Zq` must be the same size. Use `meshgrid` to evaluate on a grid.
- For best results, provide `X`, `Y`, and `Z` as vectors.
- Code generation does not support the `'makima'` interpolation method.
- For the `'cubic'` interpolation method, if the grid does not have uniform spacing, an error results. In this case, use the `'spline'` interpolation method.
- For best results when you use the `'spline'` interpolation method:
  - Use `meshgrid` to create the inputs `Xq`, `Yq`, and `Zq`.
  - Use a small number of interpolation points relative to the dimensions of `V`. Interpolating over a large set of scattered points can be inefficient.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- `V` must be a double or single 3-D array. `V` can be real or complex.
- `X`, `Y`, and `Z` must:
  - Have the same type (double or single).
  - Be finite vectors or 3-D arrays with increasing and nonrepeating elements in corresponding dimensions.
  - Align with cartesian axes when `X`,`Y`, and `Z` are 3-D arrays (as if they were produced by `meshgrid`).
  - Have dimensions consistent with `V`.
- `Xq`, `Yq`, and `Zq` must be vectors or arrays of the same type (double or single). If `Xq`, `Yq`, and `Zq` are arrays, then they must have the same size. If they are vectors with different lengths, then one of them must have a different orientation.
- `method` must be `'linear'` or`'nearest'`.
- The extrapolation for the out-of-boundary input is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`interp1` | `interp2` | `interpn` | `meshgrid`

Introduced before R2006a

interpft

1-D interpolation (FFT method)

Syntax

```
y = interpft(X,n)
y = interpft(X,n,dim)
```

Description

`y = interpft(X,n)` interpolates the Fourier transform of the function values in `X` to produce `n` equally spaced points. `interpft` operates on the first dimension whose size does not equal 1.

`y = interpft(X,n,dim)` operates along dimension `dim`. For example, if `X` is a matrix, then `interpft(X,n,2)` operates on the rows of `X`.

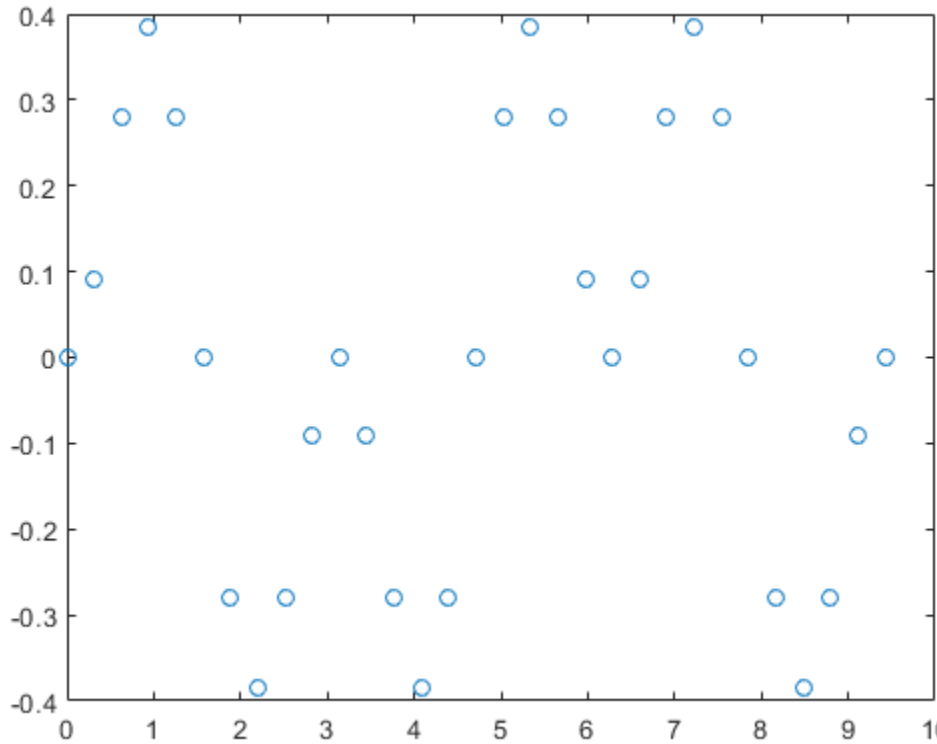
Examples

Fourier Interpolation

Interpolate 1-D data using the FFT method and visualize the result.

Generate some sample points in the interval $[0, 3\pi]$ for the function $f(x) = \sin^2(x)\cos(x)$. Use a spacing interval `dx` to ensure the data is evenly spaced. Plot the sample points.

```
dx = 3*pi/30;
x = 0:dx:3*pi;
f = sin(x).^2 .* cos(x);
plot(x,f,'o')
```



Use FFT interpolation to find the function value at 200 query points.

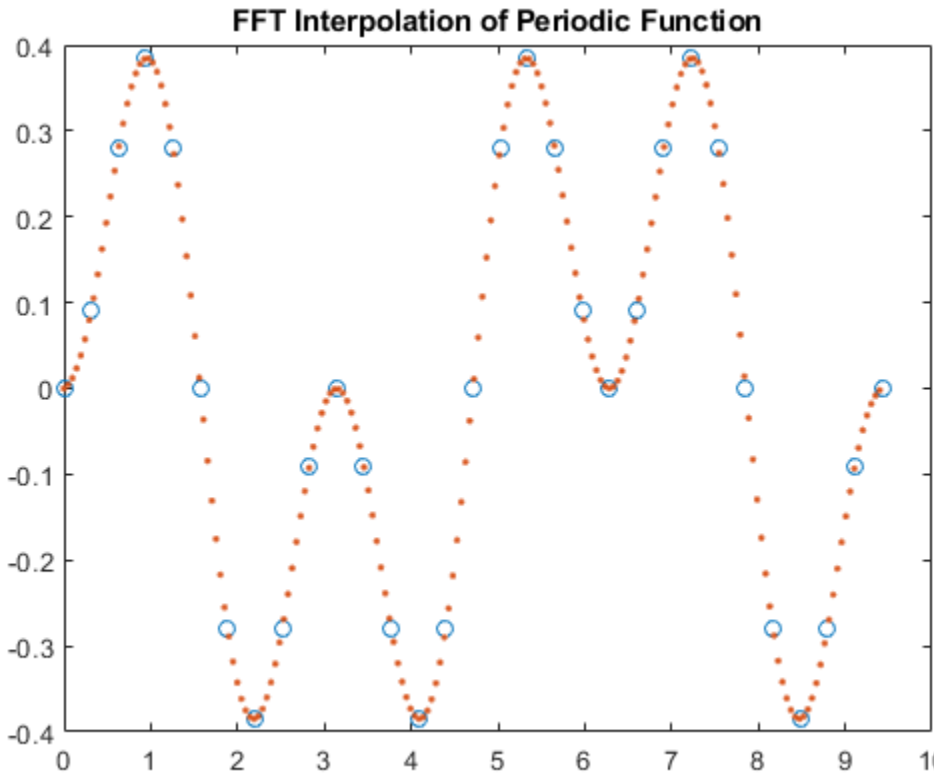
```
N = 200;
y = interpft(f,N);
```

Calculate the spacing of the interpolated data from the spacing of the sample points with `dy = dx*length(x)/N`, where `N` is the number of interpolation points. Truncate the data in `y` to match the sampling density of `x2`.

```
dy = dx*length(x)/N;
x2 = 0:dy:3*pi;
y = y(1:length(x2));
```

Plot the results.

```
hold on
plot(x2,y,'.')
title('FFT Interpolation of Periodic Function')
```



Interpolate Rows of Data

Generate three separate data sets of normally distributed random numbers. Assume the data is sampled at the positive integers, `1:N`. Store the data sets as rows in a matrix.

```
A = randn(3,20);
x = 1:20;
```

Interpolate the rows of the matrix at 500 query points each. Specify `dim = 2` so that `interpft` works on the rows of `A`.

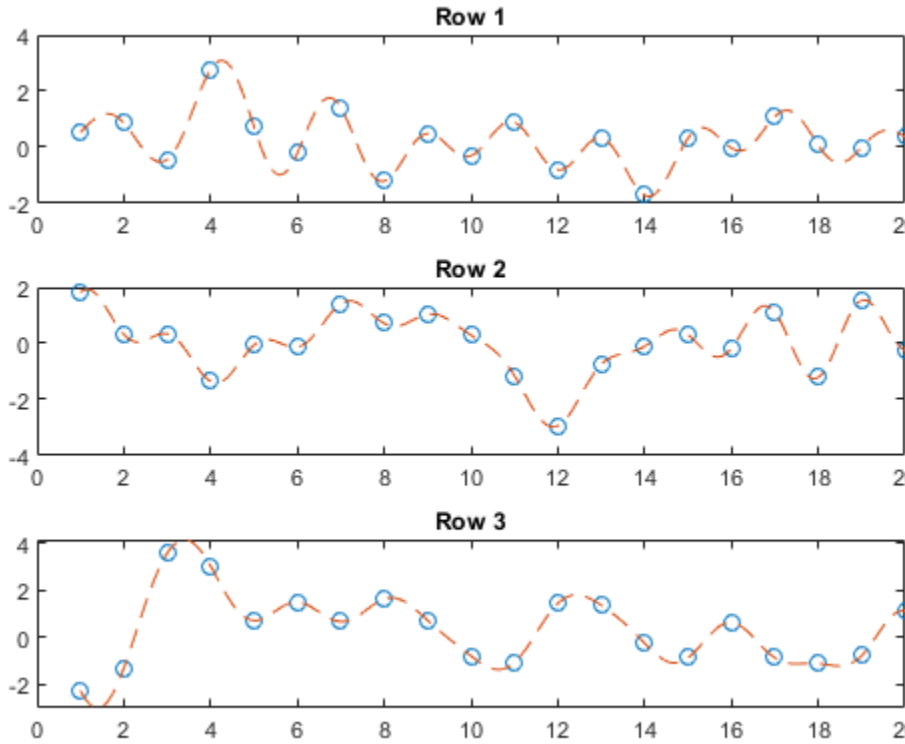
```
N = 500;
y = interpft(A,N,2);
```

Calculate the spacing interval of the interpolated data `dy`. Truncate the data in `y` to match the sampling density of `x2`.

```
dy = length(x)/N;
x2 = 1:dy:20;
y = y(:,1:length(x2));
```

Plot the results.

```
subplot(3,1,1)
plot(x,A(1,:)','o');
hold on
plot(x2,y(1,:)','--')
title('Row 1')

subplot(3,1,2)
plot(x,A(2,:)','o');
hold on
plot(x2,y(2,:)','--')
title('Row 2')

subplot(3,1,3)
plot(x,A(3,:)','o');
hold on
plot(x2,y(3,:)','--')
title('Row 3')
```



Input Arguments

X — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array. The data in X is assumed to be sampled at an evenly spaced interval of the independent variable. `interpft` works best with periodic data.

Data Types: `single` | `double`
Complex Number Support: Yes

n — Number of points
positive integer scalar

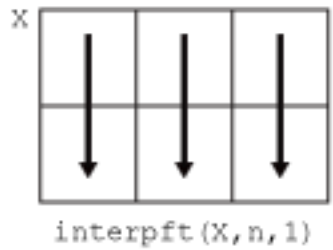
Number of points, specified as a positive integer scalar.

Data Types: `single` | `double`

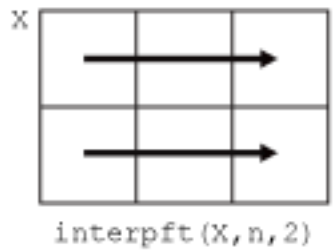
dim — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

- `interpft(X,n,1)` interpolates the columns of X.



- `interpft(X,n,2)` interpolates the rows of X.



Example: `interpft(X,n,3)`

Output Arguments

y — Interpolated points
vector | matrix | multidimensional array

Interpolated points, returned as a vector, matrix, or multidimensional array. If `length(X,dim) = m`, and `X` has a sampling interval of `dx`, then the new sampling interval for `y` is `dy = dx*m/n`, where `n > m`.

If `dim` is specified, then `interpft` pads or truncates `X` to length `n` in dimension `dim`, so that `size(y,dim) = n`.

Algorithms

The `interpft` function uses the FFT method. The original vector `x` is transformed to the Fourier domain using `fft`, and then it is transformed back with more points.

See Also

`fft` | `interp1`

Introduced before R2006a

interpn

Interpolation for 1-D, 2-D, 3-D, and N-D gridded data in ndgrid format

Syntax

```
Vq = interpn(X1,X2,...,Xn,V,Xq1,Xq2,...,Xqn)
Vq = interpn(V,Xq1,Xq2,...,Xqn)
Vq = interpn(V)
Vq = interpn(V,k)
```

```
Vq = interpn( ___ ,method)
Vq = interpn( ___ ,method,extrapval)
```

Description

`Vq = interpn(X1,X2,...,Xn,V,Xq1,Xq2,...,Xqn)` returns interpolated values of a function of *n* variables at specific query points using linear interpolation. The results always pass through the original sampling of the function. `X1,X2,...,Xn` contain the coordinates of the sample points. `V` contains the corresponding function values at each sample point. `Xq1,Xq2,...,Xqn` contain the coordinates of the query points.

`Vq = interpn(V,Xq1,Xq2,...,Xqn)` assumes a default grid of sample points. The default grid consists of the points, 1,2,3,...n_i in each dimension. The value of n_i is the length of the ith dimension in `V`. Use this syntax when you want to conserve memory and are not concerned about the absolute distances between points.

`Vq = interpn(V)` returns the interpolated values on a refined grid formed by dividing the interval between sample values once in each dimension.

`Vq = interpn(V,k)` returns the interpolated values on a refined grid formed by repeatedly halving the intervals `k` times in each dimension. This results in `2^k-1` interpolated points between sample values.

`Vq = interpn( ___ ,method)` specifies an alternative interpolation method: `'linear'`, `'nearest'`, `'pchip'`,`'cubic'`, `'makima'`, or `'spline'`. The default method is `'linear'`.

`Vq = interpn( ___ ,method,extrapval)` also specifies `extrapval`, a scalar value that is assigned to all queries that lie outside the domain of the sample points.

If you omit the `extrapval` argument for queries outside the domain of the sample points, then based on the `method` argument `interpn` returns one of the following:

- The extrapolated values for the `'spline'` and `'makima'` methods
- `NaN` values for other interpolation methods

Examples

1-D Interpolation

Define the sample points and values.

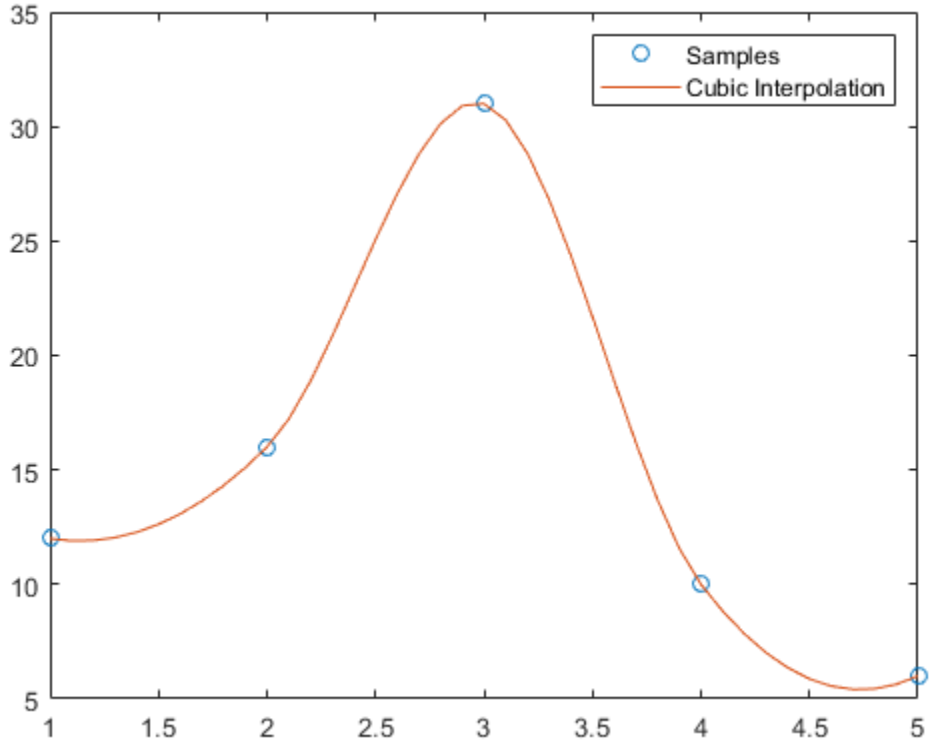
```
x = [1 2 3 4 5];
v = [12 16 31 10 6];
```

Define the query points, `xq`, and interpolate.

```
xq = (1:0.1:5);
vq = interpn(x,v,xq,'cubic');
```

Plot the result.

```
figure
plot(x,v,'o',xq,vq,'-');
legend('Samples','Cubic Interpolation');
```



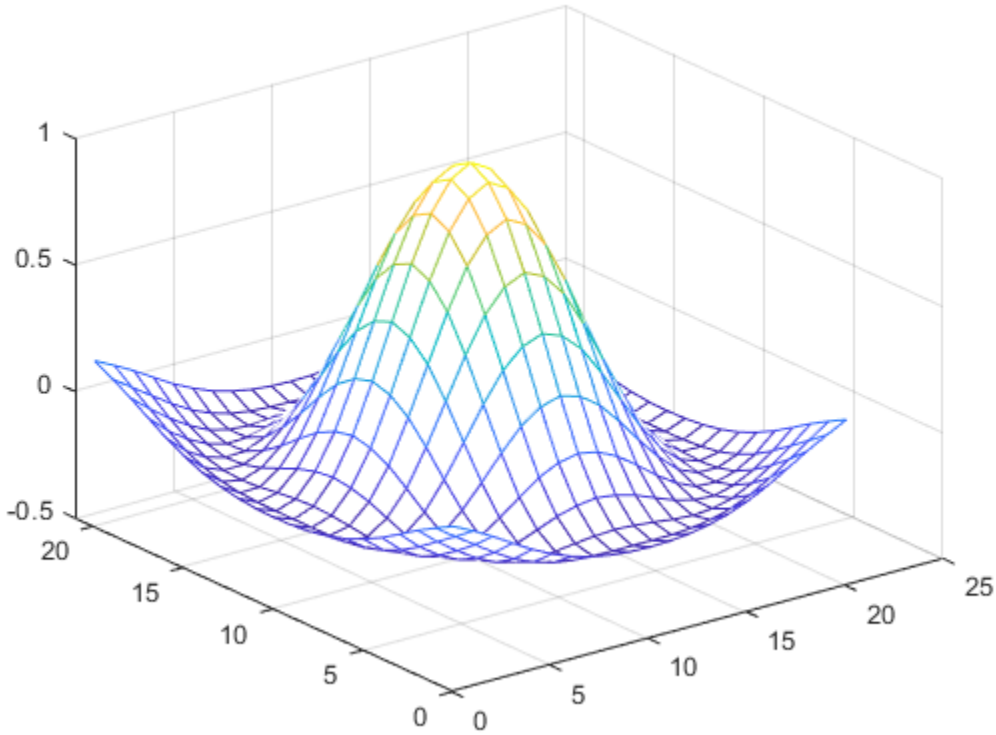
2-D Interpolation

Create a set of grid points and corresponding sample values.

```
[X1,X2] = ndgrid((-5:1:5));
R = sqrt(X1.^2 + X2.^2)+ eps;
V = sin(R)./(R);
```

Interpolate over a finer grid using `ntimes=1`.

```
Vq = interpn(V,'cubic');
mesh(Vq);
```



Evaluate Outside Domain of 3-D Function

Create the grid vectors, `x1`, `x2`, and `x3`. These vectors define the points associated with the values in `V`.

```
x1 = 1:100;
x2 = (1:50)';
x3 = 1:30;
```

Define the sample values to be a 100-by-50-by-30 random number array, `V`. Use the `gallery` function to create the array.

```
V = gallery('uniformdata',100,50,30,0);
```

Evaluate V at three points outside the domain of x1, x2, and x3. Specify extrapval = -1.

```
xq1 = [0 0 0];
xq2 = [0 0 51];
xq3 = [0 101 102];
vq = interpn(x1,x2,x3,V,xq1,xq2,xq3,'linear',-1)
```

```
vq = 1×3

    -1    -1    -1
```

All three points evaluate to -1 because they are outside the domain of x1, x2, and x3.

4-D Interpolation

Define an anonymous function that represents $f = te^{-x^2-y^2-z^2}$.

```
f = @(x,y,z,t) t.*exp(-x.^2 - y.^2 - z.^2);
```

Create a grid of points in R^4. Then, pass the points through the function to create the sample values, V.

```
[x,y,z,t] = ndgrid(-1:0.2:1,-1:0.2:1,-1:0.2:1,0:2:10);
V = f(x,y,z,t);
```

Now, create the query grid.

```
[xq,yq,zq,tq] = ...
ndgrid(-1:0.05:1,-1:0.08:1,-1:0.05:1,0:0.5:10);
```

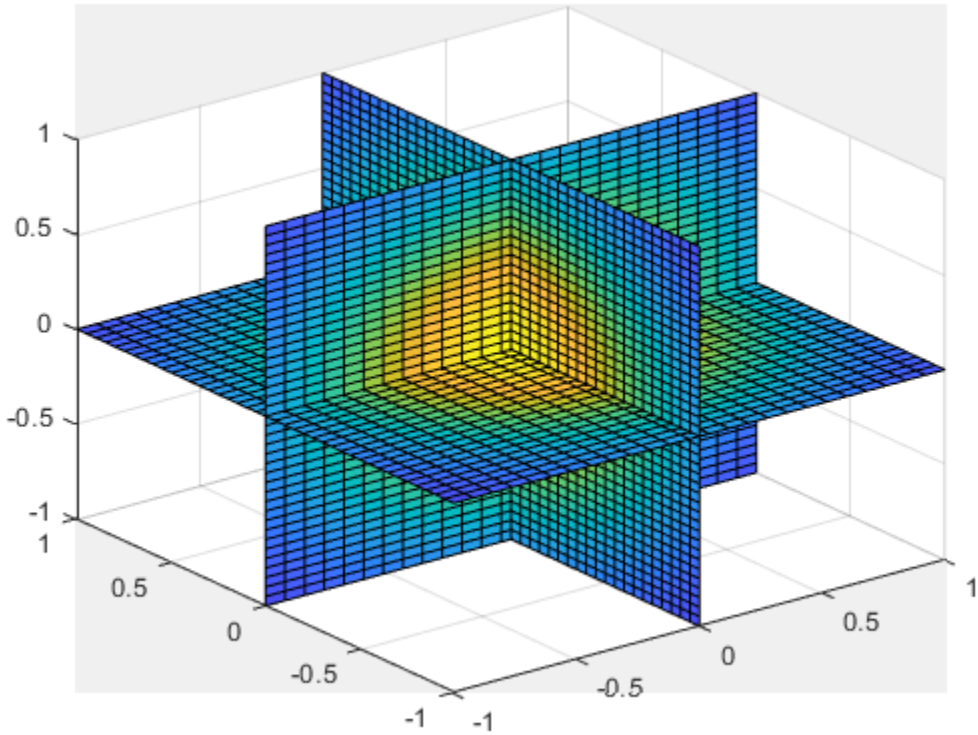
Interpolate V at the query points.

```
Vq = interpn(x,y,z,t,V,xq,yq,zq,tq);
```

Create a movie to show the results.

```
figure('renderer','zbuffer');
nframes = size(tq, 4);
```

```
for j = 1:nframes
   slice(yq(:,:,:,j),xq(:,:,:,j),zq(:,:,:,j),...
         Vq(:,:,:,j),0,0,0);
   caxis([0 10]);
   M(j) = getframe;
end
movie(M);
```



Input Arguments

X1,X2,...,Xn — Sample grid points
arrays | vectors

Sample grid points, specified as real arrays or vectors. The sample grid points must be unique.

- If `X1,X2,...,Xn` are arrays, then they contain the coordinates of a full grid (in ndgrid format) on page 1-7887. Use the `ndgrid` function to create the `X1,X2,...,Xn` arrays together. These arrays must be the same size.
- If `X1,X2,...,Xn` are vectors, then they are treated as grid vectors on page 1-7887. The values in these vectors must be strictly monotonic on page 1-7886 and increasing.

Note In a future release, `interpn` will not accept mixed combinations of row and column vectors for the sample and query grids. Instead, you must construct the full grid using `ndgrid`. Alternatively, if you have a large data set, you can use `griddedInterpolant` instead of `interpn`.

Example: `[X1,X2,X3,X4] = ndgrid(1:30,-10:10,1:5,10:13)`

Data Types: `single` | `double`

`V` — Sample values
array

Sample values, specified as a real or complex array. The size requirements for `V` depend on the size of `X1,X2,...,Xn`:

- If `X1,X2,...,Xn` are arrays representing a full grid (in `ndgrid` format), then the size of `V` matches the size of any array, `X1,X2,...,Xn`.
- If `X1,X2,...,Xn` are grid vectors, then `V` is an array whose `i`th dimension is the same length as grid vector `Xi`, where `i= 1,2,...n`.

If `V` contains complex numbers, then `interpn` interpolates the real and imaginary parts separately.

Example: `rand(10,5,3,2)`

Data Types: `single` | `double`
Complex Number Support: Yes

`Xq1,Xq2,...,Xqn` — Query points
scalars | vectors | arrays

Query points, specified as a real scalars, vectors, or arrays.

- If `Xq1,Xq2,...,Xqn` are scalars, then they are the coordinates of a single query point in R^n.
- If `Xq1,Xq2,...,Xqn` are vectors of different orientations, then `Xq1,Xq2,...,Xqn` are treated as grid vectors in R^n.
- If `Xq1,Xq2,...,Xqn` are vectors of the same size and orientation, then `Xq1,Xq2,...,Xqn` are treated as scattered points in R^n.
- If `Xq1,Xq2,...,Xqn` are arrays of the same size, then they represent either a full grid of query points (in `ndgrid` format) or scattered points in R^n.

Note In a future release, `interpn` will not accept mixed combinations of row and column vectors for the sample and query grids. Instead, you must construct the full grid using `ndgrid`. Alternatively, if you have a large data set, you can use `griddedInterpolant` instead of `interpn`.

Example: `[X1,X2,X3,X4] = ndgrid(1:10,1:5,7:9,10:11)`

Data Types: `single` | `double`

k — Refinement factor

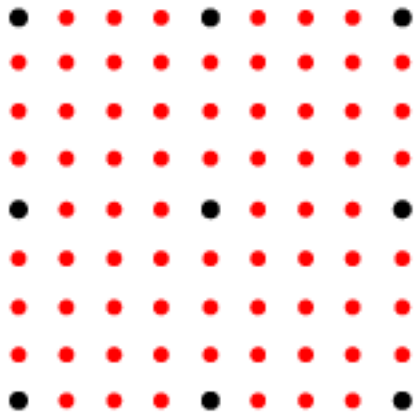
1 (default) | real, nonnegative, integer scalar

Refinement factor, specified as a real, nonnegative, integer scalar. This value specifies the number of times to repeatedly divide the intervals of the refined grid in each dimension. This results in `2^k-1` interpolated points between sample values.

If `k` is `0`, then `Vq` is the same as `V`.

`interpn(V,1)` is the same as `interpn(V)`.

The following illustration depicts `k=2` in R^2. There are 72 interpolated values in red and 9 sample values in black.



Example: `interpn(V,2)`

Data Types: `single` | `double`

`method` — Interpolation method
`'linear'` (default) | `'nearest'` | `'pchip'` | `'cubic'` | `'spline'` | `'makima'`

Interpolation method, specified as one of the options in this table.

Method	Description	Continuity	Comments
`'linear'`	The interpolated value at a query point is based on linear interpolation of the values at neighboring grid points in each respective dimension. This is the default interpolation method.	C^0	• Requires at least two grid points in each dimension • Requires more memory than `'nearest'`
`'nearest'`	The interpolated value at a query point is the value at the nearest sample grid point.	Discontinuous	• Requires two grid points in each dimension. • Fastest computation with modest memory requirements

Method	Description	Continuity	Comments
`'pchip'`	Shape-preserving piecewise cubic interpolation (for 1-D only). The interpolated value at a query point is based on a shape-preserving piecewise cubic interpolation of the values at neighboring grid points.	C^1	• Requires at least four points • Requires more memory and computation time than `'linear'`
`'cubic'`	The interpolated value at a query point is based on a cubic interpolation of the values at neighboring grid points in each respective dimension. The interpolation is based on a cubic convolution.	C^1	• Grid must have uniform spacing in each dimension, but the spacing does not have to be the same for all dimensions • Requires at least four points in each dimension • Requires more memory and computation time than `'linear'`
`'makima'`	Modified Akima cubic Hermite interpolation. The interpolated value at a query point is based on a piecewise function of polynomials with degree at most three evaluated using the values of neighboring grid points in each respective dimension. The Akima formula is modified to avoid overshoots.	C^1	• Requires at least 2 points in each dimension • Produces fewer undulations than `'spline'` • Computation time is typically less than `'spline'`, but the memory requirements are similar
`'spline'`	The interpolated value at a query point is based on a cubic interpolation of the values at neighboring grid points in each respective dimension. The interpolation is based on a cubic spline using not-a-knot end conditions.	C^2	• Requires four points in each dimension • Requires more memory and computation time than `'cubic'`

`extrapval` — Function value outside domain of `X1,X2,...,Xn`
scalar

Function value outside domain of `X1,X2,...,Xn`, specified as a real or complex scalar. `interpn` returns this constant value for all points outside the domain of `X1,X2,...,Xn`.

Example: `5`

Example: `5+1i`

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

`Vq` — Interpolated values
scalar | vector | array

Interpolated values, returned as a real or complex scalar, vector, or array. The size and shape of `Vq` depends on the syntax you use and, in some cases, the size and value of the input arguments.

Syntaxes	Special Conditions	Size of Vq	Example

`interpn(X1,...,Xn,V,Xq1,...,Xqn)`
`interpn(V,Xq1,...,Xqn)`
and variations of these syntaxes that include `method` or `extrapval`
	`Xq1,...,Xqn` are scalars	Scalar	`size(Vq) = [1 1]` when you pass `Xq1,...,Xqn` as scalars.

Same as above	`Xq1,...,Xqn` are vectors of the same size and orientation	Vector of same size and orientation as `Xq1,...,Xqn`	In 3-D, if `size(Xq1) = [100 1]`, and `size(Xq2) = [100 1]`, and `size(Xq3) = [100 1]`, then `size(Vq) = [100 1]`.

Syntaxes	Special Conditions	Size of Vq	Example
Same as above	`Xq1,...,Xqn` are vectors of mixed orientation	`size(Vq) = [length(Xq1),...,length(Xqn)]`	In 3-D, if `size(Xq1) = [1 100]`, and `size(Xq2) = [50 1]`, and `size(Xq3) = [1 5]`, then `size(Vq) = [100 50 5]`.
Same as above	`Xq1,...,Xqn` are arrays of the same size	Array of the same size as `Xq1,...,Xqn`	In 3-D, if `size(Xq1) = [50 25]`, and `size(Xq2) = [50 25]`, and `size(Xq3) = [50 25]`, then `size(Vq) = [50 25]`.
`interpn(V,k)` and variations of this syntax that include `method` or `extrapval`	None	Array in which the length of the `i`th dimension is `2^k * (size(V,i)-1)+1`,	In 3-D, if `size(V) = [10 12 5]`, and `k = 3`, then `size(Vq) = [73 89 33]`.

More About

Strictly Monotonic

A set of values that are always increasing or decreasing, without reversals. For example, the sequence, `a = [2 4 6 8]` is strictly monotonic and increasing. The sequence, `b = [2 4 4 6 8]` is not strictly monotonic because there is no change in value between `b(2)` and `b(3)`. The sequence, `c = [2 4 6 8 6]` contains a reversal between `c(4)` and `c(5)`, so it is not monotonic at all.

Full Grid (in ndgrid Format)

For `interpn`, the full grid consists of *n* arrays, `X1,X2,...,Xn`, whose elements represent a grid of points in R^n. The ith array, X_i, contains strictly monotonic on page 1-7886, increasing values that vary most rapidly along the ith dimension.

Use the `ndgrid` function to create a full grid that you can pass to `interpn`. For example, the following code creates a full grid in R^2 for the region, $1 \le X1 \le 3$, $1 \le X2 \le 4$.

```
[X1,X2] = ndgrid(-1:3,(1:4))
```

```
X1 =

    -1    -1    -1    -1
     0     0     0     0
     1     1     1     1
     2     2     2     2
     3     3     3     3


X2 =

     1     2     3     4
     1     2     3     4
     1     2     3     4
     1     2     3     4
     1     2     3     4
```

Grid Vectors

For `interpn`, grid vectors consist of *n* vectors of mixed-orientation that define the points of a grid in R^n.

For example, the following code creates the grid vectors in R^3 for the region, $1 \le x1 \le 3$, $4 \le x2 \le 5$, and $6 \le x3 \le 8$:

```
x1 = 1:3;
x2 = (4:5)';
x3 = 6:8;
```

Scattered Points

For `interpn`, scattered points consist of *n* arrays or vectors, `Xq1,Xq2,...,Xqn`, that define a collection of points scattered in R^n. The `i`th array, `Xi`, contains the coordinates in the `i`th dimension.

For example, the following code specifies the points, (1, 19, 10), (6, 40, 1), (15, 33, 22), and (0, 61, 13) in R^3.

```
Xq1 = [1 6; 15 0];
Xq2 = [19 40; 33 61];
Xq3 = [10 1; 22 13];
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For best results, provide `X1,X2,...,Xn` as vectors.
- Code generation does not support the `'makima'` interpolation method.
- Code generation does not support the `'cubic'` or `'spline'` interpolation methods for 2-D and higher interpolation.
- The interpolation method must be a constant character vector.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- A maximum of five dimensions is supported.
- `V` must be a double or single N-D array. `V` can be real or complex.

- `X1,X2,...,Xn`, `Y` must:
  - Have the same type (double or single).
  - Be finite vectors or N-D arrays with increasing and nonrepeating elements in corresponding dimensions.
  - Align with cartesian axes when `X1,X2,...,Xn` are N-D arrays (as if they were produced by `ndgrid`).
  - Have dimensions consistent with `V`.
- `X1,X2,...,Xn` must be vectors or arrays of the same type (double or single). If `X1,X2,...,Xn` are arrays, then they must have the same size. If they are vectors with different lengths, then one of them must have a different orientation.
- `method` must be `'linear'` or`'nearest'`.
- The extrapolation for the out-of-boundary input is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`interp1` | `interp2` | `interp3` | `ndgrid`

Introduced before R2006a

interpstreamspeed

Interpolate stream-line vertices from flow speed

Syntax

```
interpstreamspeed(X,Y,Z,U,V,W,vertices)
interpstreamspeed(U,V,W,vertices)
interpstreamspeed(X,Y,Z,speed,vertices)
interpstreamspeed(speed,vertices)
interpstreamspeed(X,Y,U,V,vertices)
interpstreamspeed(U,V,vertices)
interpstreamspeed(X,Y,speed,vertices)
interpstreamspeed(speed,vertices)
interpstreamspeed(...,sf)
vertsout = interpstreamspeed(...)
```

Description

`interpstreamspeed(X,Y,Z,U,V,W,vertices)` interpolates streamline vertices based on the magnitude of the vector data `U`, `V`, `W`.

The arrays `X`, `Y`, and `Z`, which define the coordinates for `U`, `V`, and `W`, must be monotonic, but do not need to be uniformly spaced. `X`, `Y`, and `Z` must have the same number of elements, as if produced by `meshgrid`.

`interpstreamspeed(U,V,W,vertices)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p] = size(U)`.

`interpstreamspeed(X,Y,Z,speed,vertices)` uses the 3-D array `speed` for the speed of the vector field.

`interpstreamspeed(speed,vertices)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p]=size(speed)`.

`interpstreamspeed(X,Y,U,V,vertices)` interpolates streamline vertices based on the magnitude of the vector data `U`, `V`.

The arrays `X` and `Y`, which define the coordinates for `U` and `V`, must be monotonic, but do not need to be uniformly spaced. `X` and `Y` must have the same number of elements, as if produced by `meshgrid`.

`interpstreamspeed(U,V,vertices)` assumes `X` and `Y` are determined by the expression

```
[X Y] = meshgrid(1:n,1:m)
```

where `[M N]=size(U)`.

`interpstreamspeed(X,Y,speed,vertices)` uses the 2-D array `speed` for the speed of the vector field.

`interpstreamspeed(speed,vertices)` assumes `X` and `Y` are determined by the expression

```
[X Y] = meshgrid(1:n,1:m)
```

where `[M,N]= size(speed)`.

`interpstreamspeed(...,sf)` uses `sf` to scale the magnitude of the vector data and therefore controls the number of interpolated vertices. For example, if `sf` is 3, then `interpstreamspeed` creates only one-third of the vertices.

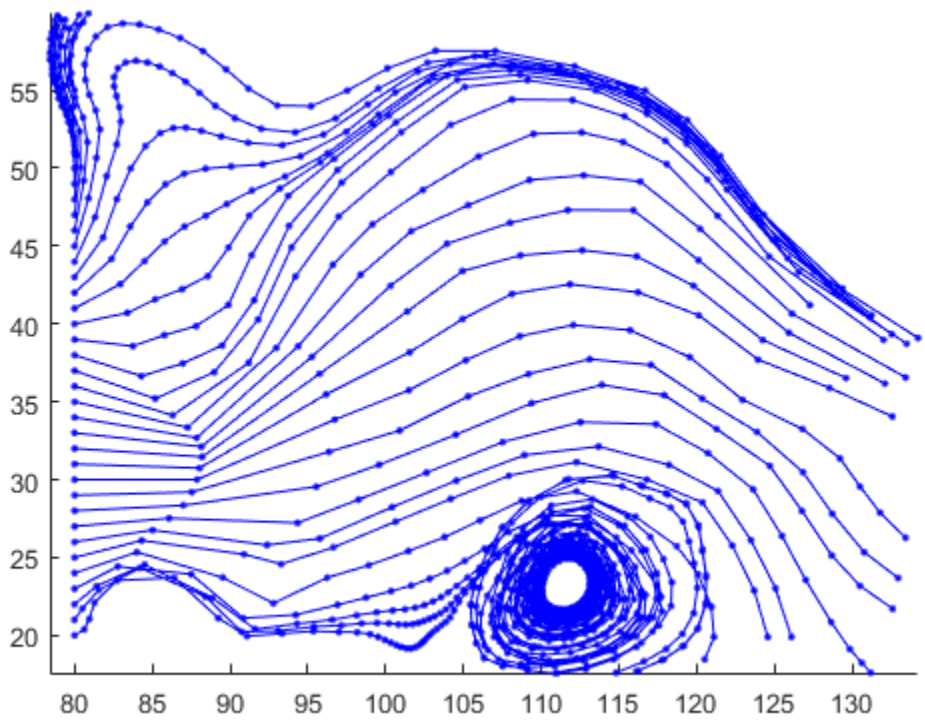
`vertsout = interpstreamspeed(...)` returns a cell array of vertex arrays.

Examples

Draw Streamlines Using Vertex Data

Draw streamlines using the vertices returned by `interpstreamspeed`. Dot markers indicate the location of each vertex. Streamlines with widely spaced vertices indicate faster flow. Those with closely spaced vertices indicate slower flow.

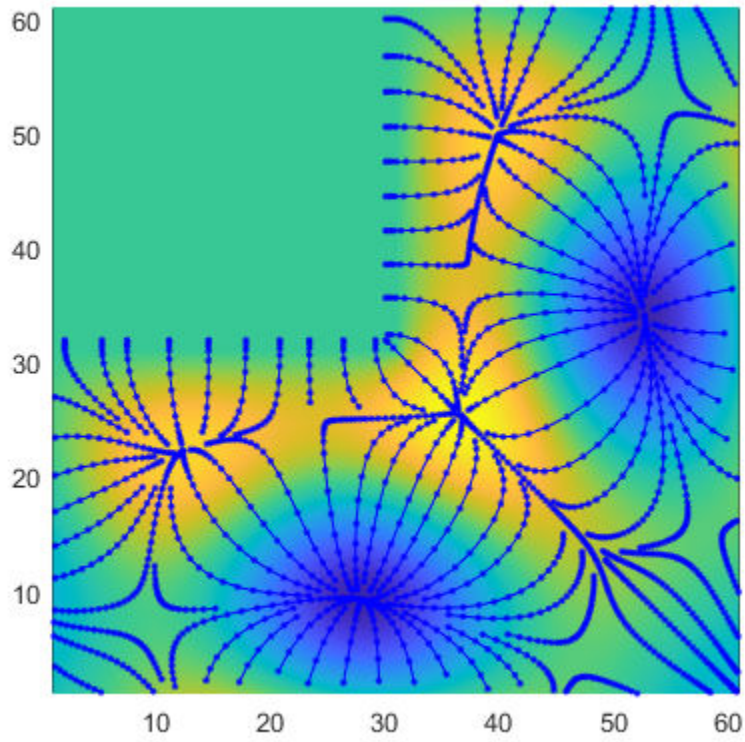
```
load wind
[sx,sy,sz] = meshgrid(80,20:1:55,5);
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
iverts = interpstreamspeed(x,y,z,u,v,w,verts,0.2);
sl = streamline(iverts);
set(sl,'Marker','.');
axis tight;
view(2);
daspect([1 1 1]);
```



Vertex Spacing That Follows Streamline Gradient

Plot streamlines to show vertex spacing as it changes with the gradient along the streamline.

```
z = membrane(6,30);
[u,v] = gradient(z);
pcolor(z);
hold on
[verts,averts] = streamslice(u,v);
iverts = interpstreamspeed(u,v,verts,15);
sl = streamline(iverts);
set(sl,'Marker','.');
shading interp
axis tight
view(2);
daspect([1,1,1]);
hold off
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`stream2` | `stream3` | `streamline` | `streamparticles` | `streamslice`

Introduced before R2006a

intersect

Set intersection of two arrays

Syntax

```
C = intersect(A,B)
C = intersect(A,B,setOrder)
C = intersect(A,B, ___ ,'rows')
C = intersect(A,B,'rows', ___ )
[C,ia,ib] = intersect( ___ )

[C,ia,ib] = intersect(A,B,'legacy')
[C,ia,ib] = intersect(A,B,'rows','legacy')
```

Description

`C = intersect(A,B)` returns the data common to both `A` and `B`, with no repetitions. `C` is in sorted order.

- If `A` and `B` are tables or timetables, then `intersect` returns the set of rows common to both tables. For timetables, `intersect` takes row times into account to determine equality, and sorts the output timetable `C` by row times.

`C = intersect(A,B,setOrder)` returns `C` in a specific order. `setOrder` can be `'sorted'` or `'stable'`.

`C = intersect(A,B, ___ ,'rows')` and `C = intersect(A,B,'rows', ___ )` treat each row of `A` and each row of `B` as single entities and return the rows common to both `A` and `B`, with no repetitions. You must specify `A` and `B` and optionally can specify `setOrder`.

The `'rows'` option does not support cell arrays, unless one of the inputs is either a categorical array or a datetime array.

`[C,ia,ib] = intersect( ___ )` also returns index vectors `ia` and `ib` using any of the previous syntaxes.

- Generally, `C = A(ia)` and `C = B(ib)`.
- If the `'rows'` option is specified, then `C = A(ia,:)` and `C = B(ib,:)`.
- If `A` and `B` are tables or timetables, then `C = A(ia,:)` and `C = B(ib,:)`.

`[C,ia,ib] = intersect(A,B,'legacy')` and `[C,ia,ib] = intersect(A,B,'rows','legacy')` preserve the behavior of the `intersect` function from R2012b and prior releases.

The `'legacy'` option does not support categorical arrays, datetime arrays, duration arrays, tables, or timetables.

Examples

Intersection of Two Vectors

Create two vectors that have some values in common.

```
A = [7 1 7 7 4];
B = [7 0 4 4 0];
```

Find the values common to both `A` and `B`.

```
C = intersect(A,B)
```

```
C = 1×2

     4     7
```

Intersection of Two Tables

Create two tables with rows in common.

```
A = table([1:5]',categorical({'A';'B';'C';'D';'E'}),logical([0;1;0;1;0]))
```

```
A=5×3 table
    Var1    Var2    Var3
    ____    ____    _____
```

```
     1       A      false
     2       B      true
     3       C      false
     4       D      true
     5       E      false
```

```
B = table([1:2:10]',categorical({'A';'C';'E';'G';'I'}),logical(zeros(5,1)))
```

```
B=5×3 table
    Var1    Var2    Var3
    ____    ____    _____

     1       A      false
     3       C      false
     5       E      false
     7       G      false
     9       I      false
```

Find the rows common to both `A` and `B`.

```
C = intersect(A,B)
```

```
C=3×3 table
    Var1    Var2    Var3
    ____    ____    _____

     1       A      false
     3       C      false
     5       E      false
```

Intersection of Two Vectors and Their Indices

Create two vectors with values in common.

```
A = [7 1 7 7 4];
B = [7 0 4 4 0];
```

Find the values common to both `A` and `B`, as well as the index vectors `ia` and `ib`, such that `C = A(ia)` and `C = B(ib)`.

```
[C,ia,ib] = intersect(A,B)
```

```
C = 1×2

     4     7


ia = 2×1

     5
     1


ib = 2×1

     3
     1
```

Intersection of Two Tables and Their Indices

Create a table, A, of gender, age, and height for five people.

```
A = table(categorical({'M';'M';'F';'M';'F'}),...
[27;52;31;46;35],[74;68;64;61;64],...
'VariableNames',{'Gender' 'Age' 'Height'},...
'RowNames',{'Ted' 'Fred' 'Betty' 'Bob' 'Judy'})
```

```
A=5×3 table
             Gender    Age    Height
             ______    ___    ______

    Ted        M       27       74
    Fred       M       52       68
    Betty      F       31       64
    Bob        M       46       61
    Judy       F       35       64
```

Create a table, B, with rows in common with A.

```
B = table(categorical({'F';'M';'F';'F'}),...
[31;47;35;23],[64;68;62;58],...
```

```
'VariableNames',{'Gender' 'Age' 'Height'},...
'RowNames',{'Meg' 'Joe' 'Beth' 'Amy'})
```

```
B=4×3 table
            Gender    Age    Height
            ______    ___    ______

    Meg       F       31       64
    Joe       M       47       68
    Beth      F       35       62
    Amy       F       23       58
```

Find the rows common to both `A` and `B`, as well as the index vectors `ia` and `ib`, such that `C = A(ia,:)` and `C = B(ib,:)`.

```
[C,ia,ib] = intersect(A,B)
```

```
C=1×3 table
             Gender    Age    Height
             ______    ___    ______

    Betty      F       31       64


ia = 3

ib = 1
```

Two rows that have the same values, but different names, are considered equal. Therefore, we discover that Betty, `A(3,:)`, and Meg, `B(1,:)` have the same gender, age, and height.

Intersection of Rows in Two Matrices

Create two matrices with rows in common.

```
A = [2 2 2; 0 0 1; 1 2 3; 1 1 1];
B = [1 2 3; 2 2 2; 2 2 0];
```

Find the rows common to both `A` and `B` as well as the index vectors `ia` and `ib`, such that `C = A(ia,:)` and `C = B(ib,:)`.

```
[C,ia,ib] = intersect(A,B,'rows')
```

```
C = 2×3

     1     2     3
     2     2     2


ia = 2×1

     3
     1


ib = 2×1

     1
     2
```

A and B do not need to have the same number of rows, but they must have the same number of columns.

Intersection with Specified Output Order

Use the `setOrder` argument to specify the ordering of the values in C.

Specify `'stable'` if you want the values in C to have the same order as in A.

```
A = [7 1 7 7 4]; B = [7 0 4 4 0];
[C,ia,ib] = intersect(A,B,'stable')
```

```
C = 1×2

     7     4


ia = 2×1

     1
     5
```

```
ib = 2×1

     1
     3
```

Alternatively, you can specify `'sorted'` order.

```
[C,ia,ib] = intersect(A,B,'sorted')
```

```
C = 1×2

     4     7


ia = 2×1

     5
     1


ib = 2×1

     3
     1
```

Intersection of Vectors Containing NaNs

Create two vectors containing `NaN`.

```
A = [5 NaN NaN];
B = [5 NaN NaN];
```

Find the values common to both `A` and `B`.

```
C = intersect(A,B)
```

```
C = 5
```

`intersect` treats `NaN` values as distinct.

Cell Array of Character Vectors with Trailing White Space

Create a cell array of character vectors, A.

```
A = {'dog','cat','fish','horse'};
```

Create a cell array of character vectors, B, where some of the vectors have trailing white space.

```
B = {'dog ','cat','fish ','horse'};
```

Find the character vectors common to both A and B.

```
[C,ia,ib] = intersect(A,B)
```

```
C = 1x2 cell array
    {'cat'}    {'horse'}


ia = 2×1

     2
     4


ib = 2×1

     2
     4
```

`intersect` treats trailing white space in cell arrays of character vectors as distinct characters.

Intersection of Arrays of Different Classes and Shapes

Create a column vector character array.

```
A = ['A';'B';'C'], class(A)
```

```
A = 3x1 char array
    'A'
    'B'
```

```
    'C'


ans =
'char'
```

Create a 2-by-3 matrix containing elements of numeric type `double`.

```
B = [65 66 67;68 69 70], class(B)
```

```
B = 2×3

    65    66    67
    68    69    70


ans =
'double'
```

Find the values common to both `A` and `B`.

```
[C,ia,ib] = intersect(A,B)
```

```
C = 3x1 char array
    'A'
    'B'
    'C'


ia = 3×1

     1
     2
     3


ib = 3×1

     1
     3
     5
```

`intersect` interprets `B` as a character array and returns a character array, `C`.

```
class(C)
```

```
ans =
'char'
```

Intersection of Char and Cell Array of Character Vectors

Create a character vector containing animal names that have three letters.

```
A = ['dog';'cat';'fox';'pig'];
class(A)
```

```
ans =
'char'
```

Create a cell array of character vectors containing animal names of varying lengths.

```
B = {'cat','dog','fish','horse'};
class(B)
```

```
ans =
'cell'
```

Find the character vectors common to both A and B.

```
C = intersect(A,B)
```

```
C = 2x1 cell array
    {'cat'}
    {'dog'}
```

The result, C, is a cell array of character vectors.

```
class(C)
```

```
ans =
'cell'
```

Preserve Legacy Behavior of intersect

Use the `'legacy'` flag to preserve the behavior of `intersect` from R2012b and prior releases in your code.

Find the intersection of A and B with the current behavior.

```
A = [7 1 7 7 4];
B = [7 0 4 4 0];
[C1,ia1,ib1] = intersect(A,B)
```

```
C1 = 1×2

     4     7


ia1 = 2×1

     5
     1


ib1 = 2×1

     3
     1
```

Find the unique elements of A and preserve the legacy behavior.

```
[C2,ia2,ib2] = intersect(A,B,'legacy')
```

```
C2 = 1×2

     4     7


ia2 = 1×2

     5     4


ib2 = 1×2

     4     1
```

Input Arguments

A,B — Input arrays

numeric arrays | logical arrays | character arrays | string arrays | categorical arrays | datetime arrays | duration arrays | cell arrays of character vectors | tables | timetables

Input arrays, specified as numeric arrays, logical arrays, character arrays, string arrays, categorical arrays, datetime arrays, duration arrays, cell arrays of character vectors, tables, or timetables. If you specify the `'rows'` option, `A` and `B` must have the same number of columns.

`A` and `B` must be of the same class with the following exceptions:

- `logical`, `char`, and all numeric classes can combine with `double` arrays.
- Cell arrays of character vectors can combine with character arrays or string arrays.
- Categorical arrays can combine with character arrays, cell arrays of character vectors, or string arrays.
- Datetime arrays can combine with cell arrays of date character vectors or single date character vectors.

There are additional requirements for `A` and `B` based on data type:

- If `A` and `B` are both ordinal categorical arrays, they must have the same sets of categories, including their order. If neither `A` nor `B` are ordinal, they need not have the same sets of categories, and the comparison is performed using the category names. In this case, the categories of `C` are the sorted union of the categories from `A` and `B`.
- If `A` and `B` are tables or timetables, they must have the same variable names (except for order). For tables, row names are ignored, so that two rows that have the same values, but different names, are considered equal. For timetables, row times are taken into account, so that two rows that have the same values, but different times, are not considered equal.
- If `A` and `B` are datetime arrays, they must be consistent with each other in whether they specify a time zone.

`A` and `B` also can be objects with the following class methods:

- `sort` (or `sortrows` for the `'rows'` option)
- `ne`

The object class methods must be consistent with each other. These objects include heterogeneous arrays derived from the same root class. For example, A and B can be arrays of handles to graphics objects.

setOrder — Order flag
'sorted' (default) | 'stable'

Order flag, specified as 'sorted' or 'stable', indicates the order of the values (or rows) in C.

Flag	Description
'sorted'	The values (or rows) in C return in sorted order as returned by sort. **Example** C = intersect([7 0 1 5],[0 2 7 5],'sorted') C = 0 5 7
'stable'	The values (or rows) in C return in the same order as they appear in A. **Example** C = intersect([7 0 1 5],[0 2 7 5],'stable') C = 7 0 5

Data Types: char | string

Output Arguments

C — Data common to A and B
vector | matrix | table | timetable

Data common to A and B, returned as a vector, matrix, or table. If the inputs A and B are tables or timetables, then the order of the variables in C is the same as the order of the variables in A.

The following describes the shape of C when the inputs are vectors or matrices and when the `'legacy'` flag is not specified:

- If the `'rows'` flag is not specified, then C is a column vector unless both A and B are row vectors, in which case C is a row vector.
- If the `'rows'` flag is specified, then C is a matrix containing the rows in common from A and B.

The class of the inputs A and B determines the class of C:

- If the class of A and B are the same, then C is the same class.
- If you combine a `char` or nondouble numeric class with `double`, then C is the same class as the nondouble input.
- If you combine a `logical` class with `double`, then C is `double`.
- If you combine a cell array of character vectors with `char`, then C is a cell array of character vectors.
- If you combine a categorical array with a character vector, cell array of character vectors, or string, then C is a categorical array.
- If you combine a datetime array with a cell array of date character vectors or single date character vector, then C is a datetime array.
- If you combine a string array with a character vector or cell array of character vectors, then C is a string array.

`ia` — Index to A

column vector

Index to A, returned as a column vector when the `'legacy'` flag is not specified. `ia` identifies the values (or rows) in A that are common to B. If there is a repeated value (or row) in A, then `ia` contains the index to the first occurrence of the value (or row).

`ib` — Index to B

column vector

Index to B, returned as a column vector when the `'legacy'` flag is not specified. `ib` identifies the values (or rows) in B that are common to A. If there is a repeated value (or row) in B, then `ib` contains the index to the first occurrence of the value (or row).

Tips

- To find the intersection with respect to a subset of variables from a table or timetable, you can use column subscripting. For example, you can use `intersect(A(:,vars),B(:,vars))`, where *`vars`* is a positive integer, a vector of positive integers, a variable name, a cell array of variable names, or a logical vector. Alternatively, you can use `vartype` to create a subscript that selects variables of a specified type.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- The `'legacy'` flag is not supported.
- The `'stable'` flag is not supported when both inputs are tall arrays.
- Inputs of type `'char'` are not supported when both inputs are tall arrays.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support cell arrays for the first or second arguments.
- When you do not specify the `'rows'` option:
  - Inputs `A` and `B` must be vectors. If you specify the `'legacy'` option, inputs `A` and `B` must be row vectors.
  - The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1.

  - The input `[]` is not supported. Use a 1-by-0 or 0-by-1 input, for example, `zeros(1,0)`, to represent the empty set.
  - If you specify the `'legacy'` option, then empty outputs are row vectors, 1-by-0. They are never 0-by-0.
- When you specify both the `'legacy'` option and the `'rows'` option, the outputs `ia` and `ib` are column vectors. If these outputs are empty, they are 0-by-1. They are never 0-by-0, even if the output `C` is 0-by-0.
- When the `setOrder` is `'sorted'` or when you specify the `'legacy'` option, the inputs must already be sorted in ascending order. The first output, `C`, is sorted in ascending order.
- Complex inputs must be `single` or `double`.
- When one input is complex and the other input is real, do one of the following:
  - Set `setOrder` to `'stable'`.
  - Sort the real input in complex ascending order (by absolute value). Suppose the real input is `x`. Use `sort(complex(x))`or `sortrows(complex(x))`.
- See “Code Generation for Complex Data with Zero-Valued Imaginary Parts” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `'legacy'` flag is not supported.
- 64-bit integers are not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`ismember` | `issorted` | `setdiff` | `setxor` | `sort` | `union` | `unique`

Topics

"Combine Categorical Arrays"

Introduced before R2006a

intmax

Largest value of specific integer type

Syntax

```
v = intmax
v = intmax(type)
```

Description

`v = intmax` returns the largest value of the 32-bit signed integer type.

`v = intmax(type)` returns the largest value of the specified integer type. When you convert a value that is larger than `intmax(type)` into the integer type `type`, the value becomes `intmax(type)`.

Examples

Largest Value of 32-bit Signed Integer

Return the largest value of the 32-bit signed integer type.

```
v = intmax
```

```
v = int32
    2147483647
```

Largest Value of 64-bit Unsigned Integer

Return the largest value of the 64-bit unsigned integer type.

```
v = intmax('uint64')
```

```
v = uint64
    18446744073709551615
```

Check the class of `v`.

```
class(v)
```

```
ans =
'uint64'
```

Convert Value Larger Than `intmax`

Return the largest value of the 8-bit signed integer type, which is 127.

```
v = intmax('int8')
```

```
v = int8
    127
```

Convert a value larger than 127 to 8-bit signed integer.

```
v1 = int8(198)
```

```
v1 = int8
    127
```

The converted value becomes 127.

Check that `v1` is the same as `v`.

```
isequal(v,v1)
```

```
ans = logical
   1
```

Input Arguments

type — Integer type
`'int32'` (default) | character vector | string

Integer type, specified as a character vector or string. This table shows the valid options for `type`.

Integer Type	Description
`'int8'`	8-bit signed integer
`'int16'`	16-bit signed integer
`'int32'`	32-bit signed integer
`'int64'`	64-bit signed integer
`'uint8'`	8-bit unsigned integer
`'uint16'`	16-bit unsigned integer
`'uint32'`	32-bit unsigned integer
`'uint64'`	64-bit unsigned integer

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`class` | `intmin` | `isa` | `realmax` | `realmin`

Introduced before R2006a

intmin

Smallest value of specified integer type

Syntax

```
v = intmin
v = intmin('classname')
```

Description

`v = intmin` is the smallest value that can be represented in the MATLAB software with a 32-bit integer. Any value smaller than the value returned by `intmin` saturates to the `intmin` value when cast to a 32-bit integer.

`v = intmin('classname')` is the smallest positive value in the integer class `classname`. Valid values for the string `classname` are

`'int8'`	`'int16'`	`'int32'`	`'int64'`
`'uint8'`	`'uint16'`	`'uint32'`	`'uint64'`

`intmin('int32')` is the same as `intmin` with no arguments.

Examples

Find the minimum value for a 64-bit signed integer:

```
v = intmin('int64')
v =

 -9223372036854775808
```

Convert this value to a 32-bit signed integer:

```
x = int32(v)
x =
```

```
 -2147483648
```

Compare the result with the default value returned by `intmin`:

```
isequal(x, intmin)
ans =

     1
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`class` | `int8` | `intmax` | `isa` | `realmax` | `realmin` | `uint8`

Introduced before R2006a

inv

Matrix inverse

Syntax

```
Y = inv(X)
```

Description

`Y = inv(X)` computes the inverse on page 1-7921 of square matrix `X`.

- `X^(-1)` is equivalent to `inv(X)`.
- `x = A\b` is computed differently than `x = inv(A)*b` and is recommended for solving systems of linear equations.

Examples

Inverse Matrix

Compute the inverse of a 3-by-3 matrix.

```
X = [1 0 2; -1 5 0; 0 3 -9]
```

```
X = 3×3

     1     0     2
    -1     5     0
     0     3    -9
```

```
Y = inv(X)
```

```
Y = 3×3

    0.8824   -0.1176    0.1961
```

```
    0.1765    0.1765    0.0392
    0.0588    0.0588   -0.0980
```

Check the results. Ideally, `Y*X` produces the identity matrix. Since `inv` performs the matrix inversion using floating-point computations, in practice `Y*X` is close to, but not exactly equal to, the identity matrix `eye(size(X))`.

```
Y*X
```

```
ans = 3×3

    1.0000    0.0000   -0.0000
         0    1.0000   -0.0000
         0   -0.0000    1.0000
```

Solve Linear System

Examine why solving a linear system by inverting the matrix using `inv(A)*b` is inferior to solving it directly using the backslash operator, `x = A\b`.

Create a random matrix `A` of order 500 that is constructed so that its condition number, `cond(A)`, is `1e10`, and its norm, `norm(A)`, is `1`. The exact solution `x` is a random vector of length 500, and the right side is `b = A*x`. Thus the system of linear equations is badly conditioned, but consistent.

```
n = 500;
Q = orth(randn(n,n));
d = logspace(0,-10,n);
A = Q*diag(d)*Q';
x = randn(n,1);
b = A*x;
```

Solve the linear system `A*x = b` by inverting the coefficient matrix `A`. Use `tic` and `toc` to get timing information.

```
tic
y = inv(A)*b;
t = toc
```

```
t = 0.0270
```

Find the absolute and residual error of the calculation.

```
err_inv = norm(y-x)
```

```
err_inv = 4.8285e-06
```

```
res_inv = norm(A*y-b)
```

```
res_inv = 4.3456e-07
```

Now, solve the same linear system using the backslash operator `\`.

```
tic
z = A\b;
t1 = toc
```

```
t1 = 0.0165
```

```
err_bs = norm(z-x)
```

```
err_bs = 4.2092e-06
```

```
res_bs = norm(A*z-b)
```

```
res_bs = 3.4458e-15
```

The backslash calculation is quicker and has less residual error by several orders of magnitude. The fact that `err_inv` and `err_bs` are both on the order of `1e-6` simply reflects the condition number of the matrix.

The behavior of this example is typical. Using `A\b` instead of `inv(A)*b` is two to three times faster, and produces residuals on the order of machine accuracy relative to the magnitude of the data.

Input Arguments

X — Input matrix
square matrix

Input matrix, specified as a square matrix. If `X` is badly scaled or nearly singular, then the `inv` calculation loses numerical accuracy. Use `rcond` or `cond` to check the condition number of the matrix.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Matrix Inverse

A matrix `X` is invertible if there exists a matrix `Y` of the same size such that $XY = YX = I_n$, where I_n is the `n`-by-`n` identity matrix. The matrix `Y` is called the inverse of `X`.

A matrix that has no inverse is singular. A square matrix is singular only when its determinant is exactly zero.

Tips

- It is seldom necessary to form the explicit inverse of a matrix. A frequent misuse of `inv` arises when solving the system of linear equations $Ax = b$. One way to solve the equation is with `x = inv(A)*b`. A better way, from the standpoint of both execution time and numerical accuracy, is to use the matrix backslash operator `x = A\b`. This produces the solution using Gaussian elimination, without explicitly forming the inverse. See `mldivide` for further information.

Algorithms

`inv` performs an LU decomposition of the input matrix (or an LDL decomposition if the input matrix is Hermitian). It then uses the results to form a linear system whose solution is the matrix inverse `inv(X)`. For sparse inputs, `inv(X)` creates a sparse identity matrix and uses backslash, `X\speye(size(X))`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Singular matrix inputs can produce nonfinite values that differ from MATLAB results.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

- `X` must be nonsparse.
- The MATLAB `inv` function prints a warning if `X` is badly scaled or nearly singular. The `gpuArray` `inv` is unable to check for this condition. Take action to avoid this condition.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- `X` must be nonsparse.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`det` | `lu` | `mldivide` | `rref`

Introduced before R2006a

invhilb

Inverse of Hilbert matrix

Syntax

```
H = invhilb(n)
H = invhilb(n,classname)
```

Description

`H = invhilb(n)` generates the exact inverse of the exact Hilbert matrix for `n` less than about 15. For larger `n`, the `invhilb` function generates an approximation to the inverse Hilbert matrix.

`H = invhilb(n,classname)` returns a matrix of class `classname`, which can be either `'single'` or `'double'`.

Examples

Inverse Hilbert Matrix

Compute the fourth-order inverse Hilbert matrix.

```
invhilb(4)
```

```
ans = 4×4

          16        -120         240        -140
        -120        1200       -2700        1680
         240       -2700        6480       -4200
        -140        1680       -4200        2800
```

Input Arguments

n — Matrix order
scalar, nonnegative integer

Matrix order, specified as a scalar, nonnegative integer.

Example: `invhilb(10)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`classname` — Matrix class
`'double'` (default) | `'single'`

Matrix class, specified as either `'double'` or `'single'`.

Example: `invhilb(10,'single')`

Data Types: `char`

Limitations

The exact inverse of the exact Hilbert matrix is a matrix whose elements are large integers. As long as the order of the matrix `n` is less than 15, these integers can be represented as floating-point numbers without roundoff error.

Comparing `invhilb(n)` with `inv(hilb(n))` involves the effects of two or three sets of roundoff errors:

- Errors caused by representing `hilb(n)`
- Errors in the matrix inversion process
- Errors, if any, in representing `invhilb(n)`

The first of these roundoff errors involves representing fractions like 1/3 and 1/5 in floating-point representation and is the most significant.

References

[1] Forsythe, G. E. and C. B. Moler. *Computer Solution of Linear Algebraic Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1967.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`hilb`

Introduced before R2006a

invoke

Invoke method on COM object or interface, or display methods

Syntax

```
invoke(c)
S = invoke(c,methodName)
S = invoke(c,methodName,arg1,...,argN)
```

Description

`invoke(c)` displays methods supported by object or interface `c`, along with the prototypes for these methods. If the output is empty, then either there are no properties or methods in the object, or MATLAB cannot read the object type library. For information about the object type library, see your COM vendor documentation.

`S = invoke(c,methodName)` invokes the method specified by `methodName` and returns an output value, if any, in a structure array. The method determines the data type of the return value. For a description of how MATLAB converts COM types, see “Handle COM Data in MATLAB”.

If the method returns a COM interface, then the `invoke` function returns a new MATLAB COM object that represents the interface.

`S = invoke(c,methodName,arg1,...,argN)` invokes `methodName` with input arguments `arg1,...,argN`.

Input Arguments

c — COM object
function handle

COM object or interface, specified as a function handle.

`methodName` — Object method name
string | character vector

Object method name, specified as a string or a character vector.

`arg1,...,argN` — Method input arguments
any type

Method input arguments, 1 through N (if any), required by `methodName`, specified by any type. The method argument list specifies the argument type.

See Also

`ismethod` | `methods`

Topics

"Handle COM Data in MATLAB"

Introduced before R2006a

ipermute

Inverse permute array dimensions

Syntax

```
A = ipermute(B,dimorder)
```

Description

`A = ipermute(B,dimorder)` rearranges the dimensions of an array `B` in the order specified by the vector `dimorder` such that `B = permute(A,dimorder)`.

Examples

3-D Array

Create a 4-by-3-by-2 array `B`, and compute its inverse permutation according to the dimension order `[3 1 2]`.

```
rng default
B = rand(4,3,2)
```

```
B =
B(:,:,1) =

    0.8147    0.6324    0.9575
    0.9058    0.0975    0.9649
    0.1270    0.2785    0.1576
    0.9134    0.5469    0.9706


B(:,:,2) =

    0.9572    0.4218    0.6557
    0.4854    0.9157    0.0357
```

```
    0.8003    0.7922    0.8491
    0.1419    0.9595    0.9340
```

```
A = ipermute(B,[3 1 2])
```

```
A =
A(:,:,1) =

    0.8147    0.9572
    0.6324    0.4218
    0.9575    0.6557


A(:,:,2) =

    0.9058    0.4854
    0.0975    0.9157
    0.9649    0.0357


A(:,:,3) =

    0.1270    0.8003
    0.2785    0.7922
    0.1576    0.8491


A(:,:,4) =

    0.9134    0.1419
    0.5469    0.9595
    0.9706    0.9340
```

The inverse permutation A is the array such that, when you permute it using the same dimension order, the result is equal to the original array B.

```
C = permute(A,[3 1 2])
```

```
C =
C(:,:,1) =

    0.8147    0.6324    0.9575
    0.9058    0.0975    0.9649
```

```
    0.1270    0.2785    0.1576
    0.9134    0.5469    0.9706


C(:,:,2) =

    0.9572    0.4218    0.6557
    0.4854    0.9157    0.0357
    0.8003    0.7922    0.8491
    0.1419    0.9595    0.9340
```

Input Arguments

B — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

`dimorder` — Dimension order
row vector

Dimension order, specified as a row vector with unique, positive integer elements representing the dimensions of the input array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitation:

Permuting the tall dimension (dimension one) is not supported.

For more information, see “Tall Arrays for Out-of-Memory Data”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Does not support cell arrays for the first argument.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`permute` | `reshape` | `shiftdim` | `transpose`

Introduced before R2006a

is*

Detect state

Description

These functions detect the state of MATLAB entities:

`isa`	Detect object of given MATLAB class or Java class
`isappdata`	Determine if object has specific application-defined data
`isbanded`	Determine if matrix is within specific bandwidth
`isbetween`	Array elements occurring within date and time interval
`iscalendarduration`	Determine if input is duration array
`iscategorical`	Determine whether input is categorical array
`iscategory`	Test for categorical array categories
`iscell`	Determine if input is cell array
`iscellstr`	Determine if input is cell array of character vectors
`ischar`	Determine if input is character array
`iscolumn`	Determine whether input is column vector
`iscom`	Determine if input is Component Object Model (COM) object
`isdatetime`	Determine if input is datetime array
`isdiag`	Determine if matrix is diagonal
`isdst`	Datetime values occurring during daylight saving time
`isduration`	Determine if input is duration array
`isempty`	Determine if input is empty array
`isenum`	Determine if variable is enumeration
`isequal`	Determine if arrays are numerically equal
`isequaln`	Determine if arrays are numerically equal, treating `NaN`s as equal

isevent	Determine if input is Component Object Model (COM) object event
isfield	Determine if input is MATLAB structure array field
isfile	Determine if input is a file
isfinite	Detect finite elements of array
isfloat	Determine if input is floating-point array
isfolder	Determine if input is folder
ishandle	Detect valid graphics object handles
ishermitian	Determine if matrix is Hermitian or skew-Hermitian
ishold	Determine if graphics hold state is on
isinf	Detect infinite elements of array
isinteger	Determine if input is integer array
isinterface	Determine if input is Component Object Model (COM) interface
isjava	Determine if input is Java object
iskeyword	Determine if input is MATLAB keyword
isletter	Detect elements that are alphabetic letters
islogical	Determine if input is logical array
ismac	Determine if running MATLAB for Macintosh OS X platform
ismatrix	Determine whether input is matrix
ismember	Detect members of specific set
ismethod	Determine if input is object method
ismissing	Find table elements with missing values
isnan	Detect elements of array that are not a number (NaN)
isnat	Determine NaT (Not-a-Time) elements
isnumeric	Determine if input is numeric array
isobject	Determine if input is MATLAB object
isordinal	Determine whether input is ordinal categorical array
ispc	Determine if running MATLAB for PC (Windows) platform

isprime	Detect prime elements of array
isprop	Determine if input is object property
isprotected	Determine whether categories of categorical array are protected
isreal	Determine if all array elements are real numbers
isregular	Determine whether times in timetable are regular
isrow	Determine whether input is row vector
isscalar	Determine if input is scalar
issorted	Determine if set elements are in sorted order
issortedrows	Determine if matrix or table rows are sorted
isspace	Detect space characters in array
issparse	Determine if input is sparse array
isstring	Determine if input is string array
isStringScalar	Determine whether input is string array with one element
isstrprop	Determine if string is of specified category
isstruct	Determine if input is MATLAB structure array
isstudent	Determine if Student Version of MATLAB
issymmetric	Determine if matrix is symmetric or skew-symmetric
istable	Determine whether input is table
istall	Determine if input is tall array
istimetable	Determine whether input is timetable
istril	Determine if matrix is lower triangular
istriu	Determine if matrix is upper triangular
isundefined	Find undefined elements in categorical array
isunix	Determine if running MATLAB for UNIX[a] platform.
isvarname	Determine if input is valid variable name
isvector	Determine if input is vector
isweekend	Datetime values occurring during weekend

a. UNIX is a registered trademark of The Open Group in the United States and other countries.

See Also

`exist` | `isa`

isa

Determine if input has specified data type

Syntax

```
tf = isa(A,dataType)
tf = isa(A,typeCategory)
```

Description

`tf = isa(A,dataType)` returns `1` (`true`) if `A` has the data type specified by `dataType`. Otherwise, it returns `0` (`false`). The input argument `A` can have any data type.

If `A` is an object, then `isa` returns `1` if `dataType` is either the class of `A` or a superclass of `A`.

`tf = isa(A,typeCategory)` returns `1` (`true`) if the data type of `A` belongs to the category specified by `typeCategory`. Otherwise, it returns `0` (`false`).

If `A` is an object, then `isa` returns `1` if the class of `A`, or any superclass of `A`, belongs to the specified category.

Examples

Determine If Variables Have Specified Data Types

Create a numeric variable and determine if its data type is `double`.

```
A = 3.1416;
tf = isa(A,'double')
```

```
tf = logical
   1
```

Create an array of 32-bit integers and determine if its data type is `int32`.

```
A = int32([0 2 4 6 8])
```

```
A = 1x5 int32 row vector

   0   2   4   6   8
```

```
tf = isa(A,'int32')
```

```
tf = logical
   1
```

Determine if the data type of `A` is `char`.

```
tf = isa(A,'char')
```

```
tf = logical
   0
```

Determine If Variable Type Belongs To Specified Categories

Create an array whose data type is `uint8`. Determine if the array has a data type that belongs to the `integer` category.

```
A = uint8([0 2 4 6 8])
```

```
A = 1x5 uint8 row vector

   0   2   4   6   8
```

```
tf = isa(A,'integer')
```

```
tf = logical
   1
```

Determine if the data type of `A` belongs to the `float` category.

```
tf = isa(A,'float')
```

```
tf = logical
   0
```

Input Arguments

A — Input array
array

Input array.

dataType — Data type
character vector | string scalar

Data type, specified as a character vector or string scalar. `dataType` can be the name of:

- Any fundamental data type or MATLAB class
- A Java or .NET class

The table shows the names of many commonly used MATLAB data types.

`'single'`	Single-precision number
`'double'`	Double-precision number
`'int8'`	Signed 8-bit integer
`'int16'`	Signed 16-bit integer
`'int32'`	Signed 32-bit integer
`'int64'`	Signed 64-bit integer
`'uint8'`	Unsigned 8-bit integer
`'uint16'`	Unsigned 16-bit integer
`'uint32'`	Unsigned 32-bit integer
`'uint64'`	Unsigned 64-bit integer
`'logical'`	Logical `1` (`true`) or `0` (`false`)
`'char'`	Character
`'string'`	String array

`'struct'`	Structure array
`'cell'`	Cell array
`'table'`	Table
`'function_handle'`	Function handle

`typeCategory` — Data type category
`'numeric'` | `'float'` | `'integer'`

Data type category, specified as `'numeric'`, `'float'`, or `'integer'`. These values represent categories of numeric types, as shown in the table.

`'numeric'`	Integer or floating-point array, having one of these data types: • `double` • `single` • `int8` • `int16` • `int32` • `int64` • `uint8` • `uint16` • `uint32` • `uint64`
`'float'`	Single- or double-precision floating-point array, having either of these data types: • `double` • `single`

`'integer'`	Signed or unsigned integer array, having one of these data types: • `int8` • `int16` • `int32` • `int64` • `uint8` • `uint16` • `uint32` • `uint64`

Tips

- To test whether the input array is sparse, use the `issparse` function.
- To test whether the input array has any imaginary or complex elements, use `~isreal(A)`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`class` | `exist` | `is*` | `isenum` | `isfloat` | `isinteger` | `isnumeric`

Topics

"Fundamental MATLAB Classes"
"Valid Combinations of Unlike Classes"

Introduced before R2006a

isappdata

True if application-defined data exists

Syntax

```
tf = isappdata(h,name)
```

Description

`tf = isappdata(h,name)` returns `logical(1)` if application-defined data exists and these conditions are met:

- The application data has the specified `name` value.
- The application data is associated with the UI component, `h`.

Otherwise, `isappdata` returns `logical(0)`

See Also

`getappdata` | `rmappdata` | `setappdata`

Introduced before R2006a

isaUnderlying

Determine if tall array data is of specified class

Syntax

```
tf = isaUnderlying(X,cl)
```

Description

`tf = isaUnderlying(X,cl)` returns a tall array containing logical `1` (`true`) if the underlying data in tall array `X` is of class `cl`. Otherwise, the tall array contains logical `0` (`false`).

Examples

Test Underlying Class of Tall Arrays

All tall tables and arrays belong to the `tall` class. However, the *underlying* data type of a tall array can vary.

Create a datastore for the `airlinesmall.csv` data set. Select a subset of the variables to work with, and treat `'NA'` values as missing data so that `datastore` replaces them with `NaN` values. Convert the datastore into a tall table.

```
varnames = {'Year', 'UniqueCarrier'};
ds = datastore('airlinesmall.csv','TreatAsMissing','NA',...
    'SelectedVariableNames',varnames);
tt = tall(ds)
```

```
tt =

  Mx2 tall table

    Year    UniqueCarrier
```

```
____    ____________

1987       {'PS'}
1987       {'PS'}
1987       {'PS'}
1987       {'PS'}
1987       {'PS'}
1987       {'PS'}
1987       {'PS'}
1987       {'PS'}
 :            :
 :            :
```

Test whether the underlying data type of the first table variable `Year` is `single`.

```
tf = isaUnderlying(tt.Year,'single')
```

```
tf =

  tall logical

   0
```

Determine the actual underlying data type of `Year`.

```
udt = classUnderlying(tt.Year)
```

```
udt =

  1x6 tall char array

    'double'
```

Input Arguments

X — Input array
tall array

Input array, specified as a tall array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `table` | `cell` | `categorical` | `datetime` | `duration` | `calendarDuration`

`cl` — Underlying class
character vector

Underlying class, specified as a character vector specifying any valid MATLAB class name.

Example: `tf = isaUnderlying(X,'double')`

Tips

- Use `classUnderlying` to determine the underlying data type of a tall array.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`classUnderlying` | `istall` | `tall`

Topics

“Tall Arrays for Out-of-Memory Data”

Introduced in R2016b

isbanded

Determine if matrix is within specific bandwidth

Syntax

```
tf = isbanded(A,lower,upper)
```

Description

`tf = isbanded(A,lower,upper)` returns logical `1` (`true`) if matrix `A` is within the specified lower bandwidth, `lower`, and upper bandwidth, `upper`; otherwise, it returns logical `0` (`false`).

Examples

Test Square Matrix

Create a 5-by-5 square matrix with nonzero diagonals above and below the main diagonal.

```
A = [2 3 0 0 0 ; 1 -2 -3 0 0; 0 -1 2 3 0 ; 0 0 1 -2 -3; 0 0 0 -1 2]
```

```
A = 5×5

     2     3     0     0     0
     1    -2    -3     0     0
     0    -1     2     3     0
     0     0     1    -2    -3
     0     0     0    -1     2
```

Specify both bandwidths, `lower` and `upper`, as `1` to test if `A` is tridiagonal.

```
isbanded(A,1,1)
```

```
ans = logical
   1
```

The result is logical `1` (`true`).

Test if `A` has nonzero elements below the main diagonal by specifying `lower` as `0`.

```
isbanded(A,0,1)
```

```
ans = logical
   0
```

The result is logical `0` (`false`) because `A` has nonzero elements below the main diagonal.

Test Nonsquare Matrix

Create a 3-by-5 matrix.

```
A = [1 0 0 0 0; 2 1 0 0 0; 3 2 1 0 0]
```

```
A = 3×5

     1     0     0     0     0
     2     1     0     0     0
     3     2     1     0     0
```

Test if `A` has nonzero elements above the main diagonal.

```
isbanded(A,2,0)
```

```
ans = logical
   1
```

The result is logical `1` (`true`) because the elements above the main diagonal are all zero.

Test Sparse Block Matrix

Create a 100-by-100 sparse block matrix.

```
B = kron(speye(25),ones(4));
```

Test if B has a lower and upper bandwidth of 1.

```
isbanded(B,1,1)
```

```
ans = logical
   0
```

The result is logical 0 (false) because the nonzero blocks centered on the main diagonal are larger than 2-by-2.

Test if B has a lower and upper bandwidth of 3.

```
isbanded(B,3,3)
```

```
ans = logical
   1
```

The result is logical 1 (true). The matrix, B, has an upper and lower bandwidth of 3 since the nonzero diagonal blocks are 4-by-4.

Input Arguments

A — Input array
numeric array

Input array, specified as a numeric array. isbanded returns logical 0 (false) if A has more than two dimensions.

Data Types: single | double
Complex Number Support: Yes

lower — Lower bandwidth
nonnegative integer scalar

Lower bandwidth, specified as a nonnegative integer scalar. The lower bandwidth is the number of nonzero diagonals below the main diagonal. `isbanded` returns logical `0` (`false`) if there are nonzero elements below the boundary diagonal, `diag(A,-lower)`.

`upper` — Upper bandwidth
nonnegative integer scalar

Upper bandwidth, specified as a nonnegative integer scalar. The upper bandwidth is the number of nonzero diagonals above the main diagonal. `isbanded` returns logical `0` (`false`) if there are nonzero elements above the boundary diagonal, `diag(A,upper)`.

Tips

- Use the `bandwidth` function to find the upper and lower bandwidths of a given matrix.
- Use `isbanded` to test for several different matrix structures by specifying appropriate upper and lower bandwidths. The table below lists some common tests.

Lower Bandwidth	Upper Bandwidth	Function Call	Tests for
`0`	`0`	`isbanded(A,0,0)`	Diagonal matrix
`1`	`1`	`isbanded(A,1,1)`	Tridiagonal matrix
`0`	`size(A,2)`	`isbanded(A,0,size(A,2))`	Upper triangular matrix
`size(A,1)`	`0`	`isbanded(A,size(A,1),0)`	Lower triangular matrix
`1`	`size(A,2)`	`isbanded(A,1,size(A,2))`	Upper Hessenberg matrix
`size(A,1)`	`1`	`isbanded(A,size(A,1),1)`	Lower Hessenberg matrix

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bandwidth` | `diag` | `isdiag` | `istril` | `istriu`

Introduced in R2014a

isbetween

Determine elements within date and time interval

Syntax

```
tf = isbetween(t,tlower,tupper)
```

Description

`tf = isbetween(t,tlower,tupper)` returns an array the same size as `t` containing logical 1 (true) where the corresponding element of `t` is a datetime or duration that lies within the closed interval specified by the corresponding elements of `tlower` and `tupper`. The output `tf` indicates which elements of `t` satisfy:

```
tlower <= t <= tupper
```

Examples

Determine if Dates Occur Within Interval

Define a lower bound and an upper bound for dates.

```
tlower = datetime(2014,05,16)
```

```
tlower = datetime
   16-May-2014
```

```
tupper = '23-May-2014'
```

```
tupper =
'23-May-2014'
```

`tlower` and `tupper` can be `datetime` arrays or character vectors. Here, `tlower` is a `datetime` array and `tupper` is a single character vector.

Create an array of datetime values and determine if each datetime lies within the interval bounded by `tlower` and `tupper`.

```
t = tlower + caldays(2:2:10)
```

```
t = 1x5 datetime array
   18-May-2014   20-May-2014   22-May-2014   24-May-2014   26-May-2014
```

```
tf = isbetween(t,tlower,tupper)
```

```
tf = 1x5 logical array

   1   1   1   0   0
```

Input Arguments

`t` — Input date and time
`datetime` array | `duration` array | character vector | cell array of character vectors | string array

Input date and time, specified as a `datetime` array, `duration` array, character vector, cell array of character vectors, or string array. Character vectors and strings must be formatted to represent dates and times.

`tlower` — Lower bound of date and time interval
`datetime` array | `duration` array | character vector | cell array of character vectors | string array

Lower bound of date and time interval, specified as a `datetime` array, `duration` array, character vector, cell array of character vectors, or string array. Character vectors and strings must be formatted to represent dates and times.

`tupper` — Upper bound of date and time interval
`datetime` array | `duration` array | character vector | cell array of character vectors | string array

Upper bound of date and time interval, specified as a `datetime` array, `duration` array, character vector, cell array of character vectors, or string array. Character vectors and strings must be formatted to represent dates and times.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitation:

Tall character vector inputs are not supported.

For more information, see “Tall Arrays for Out-of-Memory Data”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`ge` | `gt` | `ismember` | `le` | `lt`

Introduced in R2014b

iscalendarduration

Determine if input is calendar duration array

Syntax

```
tf = iscalendarduration(t)
```

Description

`tf = iscalendarduration(t)` returns logical `1` (`true`) if `t` is a `calendarDuration` array. Otherwise, it returns logical `0` (`false`).

Examples

Determine if Array Contains Calendar Duration Values

Determine if the output of an arithmetic calculation is a `calendarDuration` array.

Add two `calendarDuration` arrays.

```
d1 = calyears(1:4);
d2 = caldays(1:4);
d = d1 + d2
```

```
d = 1x4 calendarDuration array
   1y 1d   2y 2d   3y 3d   4y 4d
```

Determine if the output is a `calendarDuration` array.

```
tf = iscalendarduration(d)
```

```
tf = logical
   1
```

Input Arguments

`t` — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. `t` can be any data type.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also
`calendarDuration` | `isdatetime` | `isduration`

Introduced in R2014b

iscategorical

Determine whether input is categorical array

Syntax

```
tf = iscategorical(A)
```

Description

`tf = iscategorical(A)` returns logical `1` (`true`) if `A` is a categorical array. Otherwise, `iscategorical` returns logical `0` (`false`).

Examples

Determine Whether Workspace Variable Is Categorical Array

Create a workspace variable, `A`.

```
A = categorical({'red' 'green' 'violet'; 'orange' 'red' 'yellow'})
```

```
A = 2x3 categorical array
     red         green      violet
     orange      red        yellow
```

Verify that the workspace variable, `A`, is a categorical array.

```
tf = iscategorical(A)
```

```
tf = logical
   1
```

`A` is a 2-by-3 categorical array.

Input Arguments

A — Input variable
workspace variable

Input variable, specified as a workspace variable. A can be any data type.

Extended Capabilities

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`categorical` | `iscell` | `islogical` | `isnumeric` | `isobject` | `isstruct` | `istable`

Introduced in R2013b

iscategory

Test for categorical array categories

Syntax

```
tf = iscategory(A,catnames)
```

Description

`tf = iscategory(A,catnames)` returns an array containing logical `1` (`true`) where the data in `catnames` is a category of `A`. Otherwise, `iscategory` returns logical `0` (`false`).

`tf` is the same size as `catnames`.

Examples

Test for Categories

Create an ordinal categorical array, `A`.

```
A = categorical({'shirt' 'pants'; 'pants' 'hat'; 'shirt' 'pants'})
```

```
A = 3x2 categorical array
     shirt      pants
     pants      hat
     shirt      pants
```

`A` is a 3-by-2 categorical array.

Test if the articles of clothing, `shirt`, `pants`, `socks`, and `shoes`, are categories of `A`.

```
catnames = {'shirt' 'pants' 'socks' 'shoes'};
tf = iscategory(A,catnames)
```

```
tf = 1x4 logical array

   1   1   0   0
```

`shirt` and `pants` are categories of `A`, but `socks` and `shoes` are not.

`iscategory` does not tell us anything about the category, `hat`, which we did not include in `catnames`.

Test for Category with No Corresponding Data

Create a categorical array, `A`.

```
A = categorical({'plane' 'car' 'train' 'car' 'plane'},...
    {'boat' 'car' 'plane' 'train'})
```

```
A = 1x5 categorical array
     plane      car      train      car      plane
```

`A` is a 1-by-5 categorical array.

Test to see if `boat` is a category in `A`.

```
tf = iscategory(A,'boat')
```

```
tf = logical
   1
```

`iscategory` returns true, even though `A` does not contain any values from the category `boat`.

Input Arguments

`A` — Categorical array
vector | matrix | multidimensional array

Categorical array, specified as a vector, matrix, or multidimensional array.

`catnames` — Category names
character vector | cell array of character vectors | string array | categorical array

Category names, specified as a character vector, cell array of character vectors, string array, or categorical array.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

This function fully supports code generation. For more information, see “Code Generation for Categorical Arrays” (MATLAB Coder).

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also
`addcats` | `categorical` | `categories` | `ismember` | `mergecats` | `removecats` | `renamecats` | `reordercats` | `setcats` | `unique`

Introduced in R2013b

iscell

Determine if input is cell array

Syntax

```
tf = iscell(A)
```

Description

`tf = iscell(A)` returns logical `1` (`true`) if `A` is a cell array and logical `0` (`false`) otherwise.

Examples

```
A{1,1} = [1 4 3; 0 5 8; 7 2 9];
A{1,2} = 'Anne Smith';
A{2,1} = 3+7i;
A{2,2} = -pi:pi/10:pi;

iscell(A)

ans =

     1
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cell` | `is*` | `isa` | `iscellstr` | `islogical` | `isnumeric` | `isobject` | `isstruct` | `istable`

Introduced before R2006a

iscellstr

Determine if input is cell array of character vectors

Syntax

```
tf = iscellstr(A)
```

Description

`tf = iscellstr(A)` returns logical `1` (`true`) if `A` is a cell array of character vectors (or an empty cell array), and logical `0` (`false`) otherwise. A cell array of character vectors is a cell array where every cell contains a character vector.

Examples

Determine if Array is Cell Array of Character Vectors

Create different arrays, and then determine if they are cell arrays of character vectors.

Test a cell array of character vectors.

```
C1 = {'Smith','Chung','Morales'; ...
      'Sanchez','Peterson','Adams'}
```

```
C1 = 2x3 cell array
    {'Smith'  }    {'Chung'   }    {'Morales'}
    {'Sanchez'}    {'Peterson'}    {'Adams'  }
```

```
tf = iscellstr(C1)
```

```
tf = logical
   1
```

Every cell of `C1` contains a character vector, so `iscellstr` returns `1`.

Convert `C1` to a string array and test it.

```
str = string(C1)
```

```
str = 2x3 string array
    "Smith"      "Chung"       "Morales"
    "Sanchez"    "Peterson"    "Adams"
```

```
tf = iscellstr(str)
```

```
tf = logical
   0
```

`str` is a string array, not a cell array, so `iscellstr` returns `0`.

Test a cell array that contains elements of different data types.

```
X = rand(1,3);
C2 = {'red','blue',X}
```

```
C2=1×3 cell
    {'red'}    {'blue'}    {1x3 double}
```

```
tf = iscellstr(C2)
```

```
tf = logical
   0
```

`C2` has a cell that does not contain a character vector, so `iscellstr` returns `0`.

Input Arguments

`A` — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. `A` can be any data type.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cellstr` | `char` | `is*` | `isa` | `iscategorical` | `iscell` | `ischar` | `isstring` | `isstrprop` | `isstruct` | `istable` | `string`

Topics

“Cell Arrays of Character Vectors”
“Create String Arrays”

Introduced before R2006a

ischar

Determine if input is character array

Syntax

```
tf = ischar(A)
```

Description

`tf = ischar(A)` returns logical `1` (`true`) if `A` is a character array and logical `0` (`false`) otherwise.

Examples

Determine if Array is Character Array

Create different arrays, and then determine if they are character arrays.

Test a character vector.

```
chr = 'Mary Jones'
```

```
chr =
'Mary Jones'
```

```
tf = ischar(chr)
```

```
tf = logical
   1
```

Test a numeric array.

```
X = rand(1,3)
```

```
X = 1×3

    0.8147    0.9058    0.1270
```

```
tf = ischar(X)
```

```
tf = logical
   0
```

Test a string array. Starting in R2017a, you can create strings using double quotes.

```
str = ["Smith","Chung","Morales"; ...
       "Sanchez","Peterson","Adams"]
```

```
str = 2x3 string array
    "Smith"      "Chung"       "Morales"
    "Sanchez"    "Peterson"    "Adams"
```

```
tf = ischar(str)
```

```
tf = logical
   0
```

`str` is a string array, not a character array, so `ischar` returns `0`.

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. `A` can be any data type.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cellstr` | `char` | `is*` | `isa` | `iscellstr` | `isletter` | `isnumeric` | `isspace` | `isstring` | `isstrprop` | `string`

Topics

“Text in String and Character Arrays”

Introduced before R2006a

iscolumn

Determine whether input is column vector

Syntax

```
iscolumn(V)
```

Description

`iscolumn(V)` returns logical 1 (`true`) if `size(V)` returns `[n 1]` with a nonnegative integer value `n`, and logical 0 (`false`) otherwise.

Examples

Determine Column Vector

Determine if a vector is a column or not.

```
V = rand(1,5);
iscolumn(V)
```

```
ans = logical
   0
```

```
Vt = V';
iscolumn(Vt)
```

```
ans = logical
   1
```

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`ismatrix` | `isrow` | `isscalar` | `isvector`

Introduced in R2010b

iscom

Determine whether input is COM object

Syntax

```
tf = iscom(c)
```

Description

`tf = iscom(c)` returns logical `1` (`true`) if `c` is a COM object. Otherwise, it returns logical `0` (`false`).

Examples

Test for COM Objects

Test an instance of a Microsoft Excel application. MATLAB displays `true`, indicating that the Excel application is a COM object.

```
app = actxserver('Excel.Application');
iscom(app)
```

Create a workbooks object and test. MATLAB displays `false`, indicating that a workbook is not a COM object.

```
w = get(app,'workbooks');
iscom(w)
```

Input Arguments

c — COM object
function handle

COM object, specified as a function handle.

Limitations

- COM functions are available on Microsoft Windows systems only.

See Also

Topics

"MATLAB COM Integration"

Introduced before R2006a

isdatetime

Determine if input is datetime array

Syntax

```
tf = isdatetime(t)
```

Description

`tf = isdatetime(t)` returns logical `1` (`true`) if `t` is a datetime array. Otherwise, it returns logical `0` (`false`).

Examples

Determine if Array Contains Datetime Values

Define an array.

```
A = [datetime('now');datetime('tomorrow');datetime(2016,1,15)]
```

```
A = 3x1 datetime array
   26-Aug-2019 23:42:54
   27-Aug-2019 00:00:00
   15-Jan-2016 00:00:00
```

Determine if the array is a `datetime` array.

```
tf = isdatetime(A)
```

```
tf = logical
   1
```

Input Arguments

`t` — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. `t` can be any data type.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also
`datetime` | `iscalendarduration` | `isduration`

Introduced in R2014b

isdiag

Determine if matrix is diagonal

Syntax

```
tf = isdiag(A)
```

Description

`tf = isdiag(A)` returns logical `1` (`true`) if `A` is a diagonal matrix on page 1-7978; otherwise, it returns logical `0` (`false`).

Examples

Test Diagonal Matrix

Create a 4-by-4 identity matrix.

```
I = eye(4)
```

```
I = 4×4

     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
```

Test to see if the matrix is diagonal.

```
isdiag(I)
```

```
ans = logical
   1
```

The result is logical `1` (`true`) because all of the nonzero elements in `I` are on the main diagonal.

Test Banded Matrix

Create a matrix with nonzero elements on the main and first diagonals.

```
A = 3*eye(4) + diag([2 2 2],1)
```

```
A = 4×4

     3     2     0     0
     0     3     2     0
     0     0     3     2
     0     0     0     3
```

Test to see if the matrix is diagonal.

```
isdiag(A)
```

```
ans = logical
   0
```

The matrix is not diagonal since there are nonzero elements above the main diagonal.

Create a new matrix, `B`, from the main diagonal elements of `A`.

```
B = diag(diag(A));
```

Test to see if `B` is a diagonal matrix.

```
isdiag(B)
```

```
ans = logical
   1
```

The result is logical `1` (`true`) because there are no nonzero elements above or below the main diagonal of `B`.

Input Arguments

A — Input array
numeric array

Input array, specified as a numeric array. `isdiag` returns logical `0` (`false`) if `A` has more than two dimensions.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Diagonal Matrix

A matrix is diagonal if all elements above and below the main diagonal are zero. Any number of the elements on the main diagonal can also be zero.

For example, the 4-by-4 identity matrix,

$$I_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

is a diagonal matrix. Diagonal matrices are typically, but not always, square.

Tips

- Use the `diag` function to produce diagonal matrices for which `isdiag` returns logical `1` (`true`).
- The functions `isdiag`, `istriu`, and `istril` are special cases of the function `isbanded`, which can perform all of the same tests with suitably defined upper and lower bandwidths. For example, `isdiag(A) == isbanded(A,0,0)`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bandwidth` | `diag` | `isbanded` | `istril` | `istriu` | `tril` | `triu`

Introduced in R2014a

isdir

(Not recommended) Determine if input is folder on search path

Note `isdir` is not recommended. Use `isfolder` instead. `isdir` searches for folders on the search path, which can lead to unexpected results. `isfolder` searches for folders only on the specified path or in the current folder.

Syntax

```
result = isdir(folderName)
```

Description

`result = isdir(folderName)` returns 1 if `folderName` is a folder on the search path. Otherwise, `isdir` returns 0.

Examples

Is Input a Folder

Check if the input `myfile1.txt` is a folder. A result of 0 indicates that `myfile1.txt` is not a folder.

```
result = isdir('myfile1.txt')
```

```
result = logical
   0
```

Create the folder `myfolder`, and then check if `myfolder` is a folder. A result of 1 indicates that `myfolder` is a folder.

```
mkdir myfolder;
result = isdir('myfolder')
```

```
result = logical
   1
```

List Only Folders in Current Folder

Get the Current Folder listing and filter out only the folders.

Create two folders, `myfolder1` and `myfolder2`.

```
mkdir myfolder1
mkdir myfolder2
```

Get the current folder listing.

```
listing = dir;
```

Filter out only the folders.

```
onlyFolders = listing([listing.isdir]);
onlyFolders.name
```

```
ans =
'.'

ans =
'..'

ans =
'myfolder1'

ans =
'myfolder2'
```

Input Arguments

`folderName` — Folder name
character vector

Folder name, specified as a character vector. You can specify `folderName` as an absolute or relative path.

See Also

`dir` | `is*` | `isfile` | `isfolder`

Introduced before R2006a

isdst

Determine daylight saving time elements

Syntax

```
tf = isdst(t)
```

Description

`tf = isdst(t)` returns an array the same size as `t` containing logical 1 (`true`) where the corresponding element of `t` is a datetime that occurs during Daylight Saving Time, and logical 0 (`false`) otherwise. `isdst` returns `false` for all elements when the `TimeZone` property of `t` is empty (`''`).

Examples

Determine If Datetime Occurs During Daylight Saving Time

```
t = datetime(2014,3,7:11,'TimeZone','America/New_York')
```

```
t = 1x5 datetime array
   07-Mar-2014   08-Mar-2014   09-Mar-2014   10-Mar-2014   11-Mar-2014
```

```
tf = isdst(t)
```

```
tf = 1x5 logical array

   0   0   0   1   1
```

March 10 and March 11, 2014 in the `America/New_York` time zone occur during daylight saving time.

Input Arguments

t — Input date and time
`datetime` array

Input date and time, specified as a `datetime` array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`datetime` | `isweekend` | `tzoffset`

Introduced in R2014b

isduration

Determine if input is duration array

Syntax

```
tf = isduration(t)
```

Description

`tf = isduration(t)` returns logical `1` (`true`) if `t` is a `duration` array. Otherwise, it returns logical `0` (`false`).

Examples

Determine if Array Contains Duration Values

Determine if the output of an arithmetic calculation is a `duration` array.

Subtract a `datetime` array from another.

```
t1 = datetime(2014,03,16:17);
t2 = datetime(2014,03,20);
dt = t2 - t1
```

```
dt = 1x2 duration array
   96:00:00   72:00:00
```

Determine if the output is a `duration` array.

```
tf = isduration(dt)
```

```
tf = logical
   1
```

Input Arguments

`t` — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. `t` can be any data type.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also
`duration` | `iscalendarduration` | `isdatetime`

Introduced in R2014b

isEdge

Class: `TriRep`

(Not recommended) Test if vertices are joined by edge

Note `isEdge(TriRep)` is not recommended. Use `isConnected(triangulation)` instead.

`TriRep` is not recommended. Use `triangulation` instead.

Syntax

```
TF = isEdge(TR, V1, V2)
TF = isEdge(TR, EDGE)
```

Description

`TF = isEdge(TR, V1, V2)` returns an array of 1/0 (true/false) flags, where each entry `TF(i)` is true if `V1(i)`, `V2(i)` is an edge in the triangulation. `V1`, `V2` are column vectors representing the indices of the vertices in the mesh, that is, indices into the vertex coordinate arrays.

`TF = isEdge(TR, EDGE)` specifies the edge start and end indices in matrix format.

Input Arguments

`TR`	Triangulation representation.
`V1, V2`	Column vectors of mesh vertices.
`EDGE`	Matrix of size `n`-by-2 where `n` is the number of query edges.

Output Arguments

TF	Array of 1/0 (true/false) flags, where each entry `TF(i)` is true if `V1(i)`, `V2(i)` is an edge in the triangulation.

Examples

Example 1

Load a 2-D triangulation and use `TriRep` to query the presence of an edge between pairs of points.

```
load trimesh2d
trep = TriRep(tri, x,y);
```

Test if vertices 3 and 117 are connected by an edge

```
isEdge(trep, 3, 117)
```

Test if vertices 3 and 164 are connected by an edge

```
isEdge(trep, 3, 164)
```

Example 2

Direct query of a 3-D Delaunay triangulation created using `DelaunayTri`.

```
X = rand(10,3)
dt = DelaunayTri(X)
```

Test if vertices 2 and 7 are connected by an edge

```
isEdge(dt, 2, 7);
```

See Also

`delaunayTriangulation` | `triangulation`

ischange

Find abrupt changes in data

Syntax

```
TF = ischange(A)
TF = ischange(A,method)
TF = ischange( ___ ,dim)
TF = ischange( ___ ,Name,Value)
[TF,S1] = ischange( ___ )
[TF,S1,S2] = ischange( ___ )
```

Description

`TF = ischange(A)` returns a logical array whose elements are 1 (`true`) when there is an abrupt change on page 1-7997 in the mean of the corresponding elements of `A`.

`TF = ischange(A,method)` specifies how to define a change point in the data. For example, `ischange(A,'variance')` finds abrupt changes in the variance of the elements of `A`.

`TF = ischange( ___ ,dim)` specifies the dimension of `A` to operate along for either of the previous syntaxes. For example, `ischange(A,2)` computes change points for each row of a matrix `A`.

`TF = ischange( ___ ,Name,Value)` specifies additional parameters for finding change points using one or more name-value pair arguments. For example, `ischange(A,'MaxNumChanges',m)` detects no more than `m` change points.

`[TF,S1] = ischange( ___ )` also returns information about the line segments in between change points. For example, `[TF,S1] = ischange(A)` returns a vector `S1` containing the mean of data between change points of a vector `A`.

`[TF,S1,S2] = ischange( ___ )` returns additional information about the line segments in between change points. For example, `[TF,S1,S2] = ischange(A)` returns

a vector S1 that contains the mean for each segment, as well as a vector S2 that contains the variance for each segment of a vector A.

Examples

Change in Mean

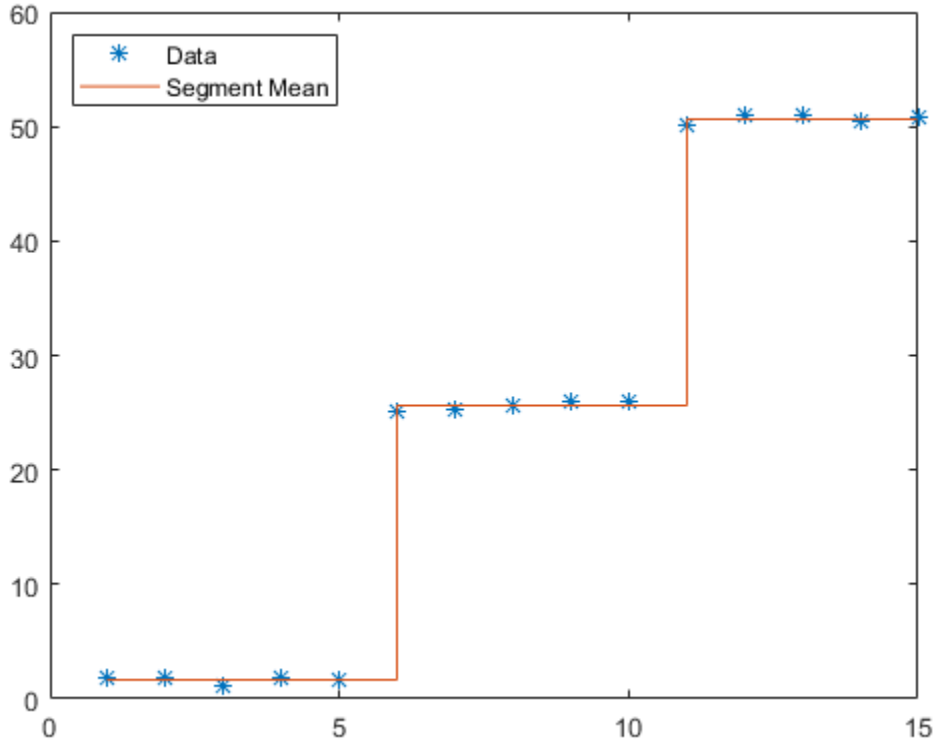
Create a vector of noisy data, and compute the abrupt changes in the mean of the data.

```
A = [ones(1,5) 25*ones(1,5) 50*ones(1,5)] + rand(1,15);
TF = ischange(A)
```

```
TF = 1x15 logical array

   0   0   0   0   0   1   0   0   0   0   1   0   0   0   0
```

To compute the mean of the data in between change points, specify a second output argument.

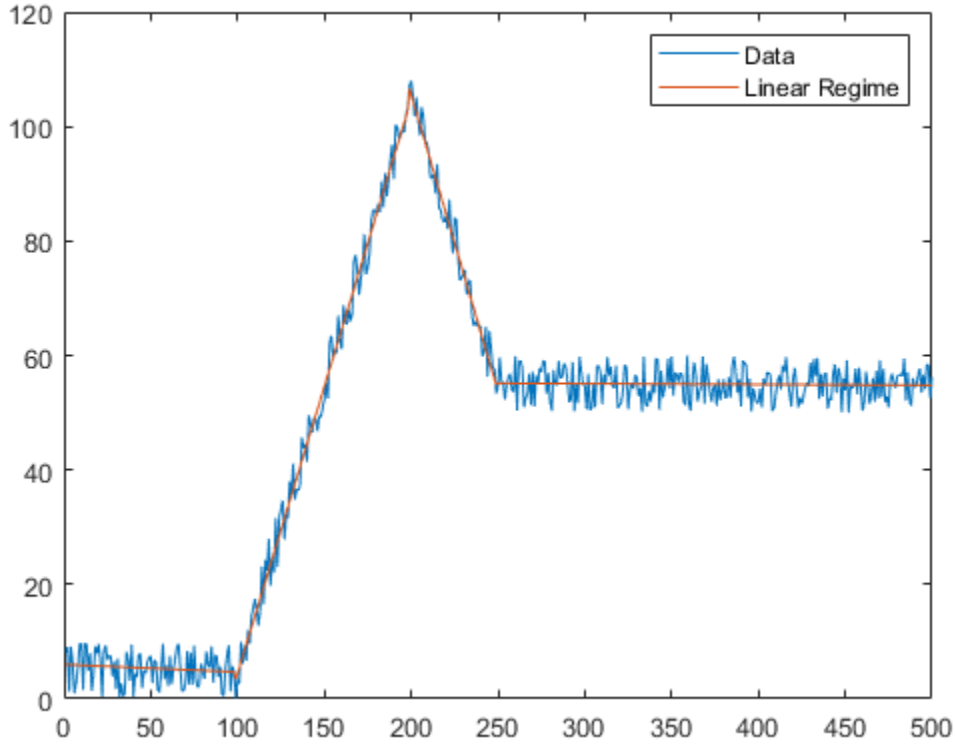
```
[TF,S1] = ischange(A);
plot(A,'*')
hold on
stairs(S1)
legend('Data','Segment Mean','Location','NW')
```



Change in Linear Regime

Create a vector of noisy data, and compute abrupt changes in the slope and intercept of the data. Setting a large detection threshold reduces the number of change points detected due to noise.

```
A = [zeros(1,100) 1:100 99:-1:50  50*ones(1,250)] + 10*rand(1,500);
[TF,S1,S2] = ischange(A,'linear','Threshold',200);
segline = S1.*(1:500) + S2;
plot(1:500,A,1:500,segline)
legend('Data','Linear Regime')
```



As an alternative to providing a threshold value, you also can specify the maximum number of change points to detect.

```
[TF,S1,S2] = ischange(A,'linear','MaxNumChanges',3);
```

Matrix Data

Compute abrupt changes in the mean for each row of a matrix.

```
A = diag(25*ones(5,1)) + rand(5,5)
```

```
A = 5×5

   25.8147    0.0975    0.1576    0.1419    0.6557
    0.9058   25.2785    0.9706    0.4218    0.0357
    0.1270    0.5469   25.9572    0.9157    0.8491
    0.9134    0.9575    0.4854   25.7922    0.9340
    0.6324    0.9649    0.8003    0.9595   25.6787
```

```
TF = ischange(A,2)
```

```
TF = 5x5 logical array

   0   1   0   0   0
   0   1   1   0   0
   0   0   1   1   0
   0   0   0   1   1
   0   0   0   0   1
```

Input Arguments

A — Input data
vector | matrix | multidimensional array | table | timetable

Input data, specified as a vector, matrix, multidimensional array, table, or timetable.

Data Types: `single` | `double` | `table` | `timetable`

`method` — Change detection method
`'mean'` (default) | `'variance'` | `'linear'`

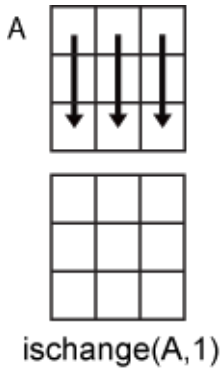
Change detection method, specified as one of the following:

- `'mean'` — Find abrupt changes in the mean of the data.
- `'variance'` — Find abrupt changes in the variance of the data.
- `'linear'` — Find abrupt changes in the slope and intercept of the data.

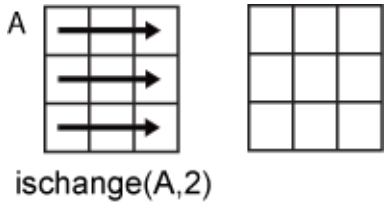
`dim` — Operating dimension
positive integer scalar

Operating dimension, specified as a positive integer scalar. By default, `ischange` operates along the first dimension whose size does not equal 1.

For example, if `A` is a matrix, then `ischange(A,1)` operates along the rows of `A`, computing change points for each column.



`ischange(A,2)` operates along the columns of `A`, computing change points for each row.



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `TF = ischange(A,'MaxNumChanges',5)`

`Threshold` — Change point threshold
1 (default) | nonnegative scalar

Change point threshold, specified as the comma-separated pair consisting of `'Threshold'` and a nonnegative scalar. Increasing the threshold greater than 1 produces fewer change points.

The threshold value defines the number of detected change points and cannot be specified when `'MaxNumChanges'` is specified.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`MaxNumChanges` — Maximum number of change points
positive integer scalar

Maximum number of change points to detect, specified as the comma-separated pair consisting of `'MaxNumChanges'` and a positive integer scalar. `ischange` uses an automatic threshold that computes no more than the specified value of change points, thus `'Threshold'` cannot be specified when `'MaxNumChanges'` is specified.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`DataVariables` — Table variables
variable name | cell array of variable names | numeric vector | logical vector | function handle

Table variables, specified as the comma-separated pair consisting of `'DataVariables'` and a variable name, a cell array of variable names, a numeric vector, a logical vector, or a function handle. The `'DataVariables'` value indicates which columns of an input table or timetable to operate on, and can be one of the following:

- A character vector specifying a single table variable name
- A cell array of character vectors where each element is a table variable name
- A vector of table variable indices
- A logical vector whose elements each correspond to a table variable, where `true` includes the corresponding variable and `false` excludes it
- A function handle that takes the table as input and returns a logical scalar

The specified table variables must be of type `double` or `single`.

Example: `'Age'`

Example: `{'Height','Weight'}`

Example: `@isnumeric`

Data Types: `char` | `cell` | `logical` | `double` | `single` | `function_handle`

`SamplePoints` — Sample points
vector

Sample points, specified as the comma-separated pair consisting of `'SamplePoints'` and a vector. The sample points represent the location of the data in `A`. Sample points do not need to be uniformly sampled, but must be sorted with unique elements. By default, the sample points vector is `[1 2 3 ...]`.

`ischange` does not support this name-value pair when the input data is a timetable.

Data Types: `double` | `single` | `datetime` | `duration`

Output Arguments

`TF` — Change point indicator
vector | matrix | multidimensional array

Change point indicator, returned as a vector, matrix, or multidimensional array. `TF` is the same size as `A`.

Data Types: `logical`

`S1` — Mean or slope
vector | matrix | multidimensional array | table | timetable

Mean or slope of data between change points, returned as a vector, matrix, multidimensional array, table, or timetable.

- If the change point detection method is `'mean'` or `'variance'`, then `S1` contains the mean for each segment.
- If the method is `'linear'`, then `S1` contains the slope for each segment.

`s1` has the same type is the input data.

Data Types: `double` | `single` | `table` | `timetable`

`S2` — Variance or intercept
vector | matrix | multidimensional array | table | timetable

Variance or intercept of data between change points, returned as a vector, matrix, multidimensional array, table, or timetable.

- If the change point detection method is `'mean'` or `'variance'`, then `S2` contains the variance for each segment.
- If the method is `'linear'`, then `S2` contains the intercept for each segment.

`s2` has the same type is the input data.

Data Types: `double` | `single` | `table` | `timetable`

More About

Change Points

A vector of data A contains a change point if it can be split into two segments A_1 and A_2 such that

$$C(A_1)+C(A_2)+\tau<C(A).$$

τ is the threshold value specified by the `'Threshold'` parameter, and C represents a cost function.

For example, the cost function for detecting abrupt changes in the mean is $C(x)=N\ \mathrm{var}(x)$, where N is the number of elements in a vector x. The cost function measures how well a segment is approximated by its mean.

`ischange` iteratively minimizes the sum of the cost functions to determine the number of change points k and their locations such that

$$C(A_1)+C(A_2)+...+C(A_k)+k\tau<C(A).$$

References

[1] Killick R., P. Fearnhead, and I.A. Eckley. "Optimal detection of changepoints with a linear computational cost." *Journal of the American Statistical Association*. Vol. 107, Number 500, 2012, pp.1590-1598.

See Also

Find Change Points | `islocalmax` | `islocalmin` | `isoutlier`

Introduced in R2017b

isempty

Determine whether array is empty

Syntax

```
TF = isempty(A)
```

Description

`TF = isempty(A)` returns logical `1` (`true`) if `A` is empty, and logical `0` (`false`) otherwise. An empty array, table, or timetable has at least one dimension with length 0, such as 0-by-0 or 0-by-5.

Examples

Empty Multidimensional Array

Create a 3-D array with one dimension length equal to zero, and determine if it is empty.

```
A = zeros(0,2,2);
TF = isempty(A)
```

```
TF = logical
   1
```

Empty Arrays and Missing Values

Compare empty arrays to arrays containing missing values.

In MATLAB®, an empty array has at least one dimension length equal to zero. An array containing missing values, such as `NaN` or `<undefined>`, is not necessarily empty.

Create a categorical vector with missing values.

```
cat1 = categorical([missing missing])
```

```
cat1 = 1x2 categorical array
     <undefined>      <undefined>
```

Since `cat1` does not have a dimension of length zero, it is not empty.

```
TF1 = isempty(cat1)
```

```
TF1 = logical
   0
```

Create a 0-by-0 categorical array and test if it is empty.

```
cat2 = categorical([]);
TF2 = isempty(cat2)
```

```
TF2 = logical
   1
```

Empty String Arrays

Compare empty string arrays and strings with no characters.

Create a string vector whose elements are strings with no characters. `str1` is nonempty since none of its dimensions have length zero.

```
str1 = strings(1,3)
```

```
str1 = 1x3 string array
    ""    ""    ""
```

```
TF1 = isempty(str1)
```

```
TF1 = logical
   0
```

Create a 0-by-3 string array and test if it is empty.

```
str2 = strings(0,3);
TF2 = isempty(str2)
```

```
TF2 = logical
   1
```

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array | table | timetable

Input array or table, specified as a scalar, vector, matrix, multidimensional array, table, or timetable.

Tips

- To determine whether a string array has empty strings (string elements with zero characters), use the == operator. For example, if `str` is a string containing zero characters, then `str == ""` returns logical `1` (`true`). For more information on testing empty strings, see “Test for Empty Strings and Missing Values”. For information on string comparison, see “Compare Text”.
- To test for missing values in an array, use the `ismissing` function.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`eq` | `ismissing`

Introduced before R2006a

isempty

Determine if `tscollection` is empty

Syntax

```
TF = isempty(tsc)
```

Description

`TF = isempty(tsc)` returns a logical 1 (`true`) when `tsc` does not contain any `timeseries` members or a time vector, and 0 (`false`) otherwise.

Examples

Empty tscollection

Create a `tscollection` object and determine if it is empty.

```
ts = timeseries();
tsc = tscollection(ts);
TF = isempty(tsc)
```

```
TF = logical
   1
```

Input Arguments

tsc — Input tscollection
scalar

Input `tscollection`, specified as a scalar.

See Also

`timeseries` | `tscollection`

Introduced before R2006a

isequal

Determine array equality

Syntax

```
tf = isequal(A,B)
tf = isequal(A1,A2,...,An)
```

Description

`tf = isequal(A,B)` returns logical `1` (`true`) if `A` and `B` are equivalent; otherwise, it returns logical `0` (`false`). See the Input Arguments on page 1-8008 section for a definition of equivalence for each data type. `NaN` (Not a Number), `NaT` (Not a Time), undefined categorical elements, and `<missing>` values are considered to be *unequal* to other elements, as well as themselves.

`tf = isequal(A1,A2,...,An)` returns logical `1` (`true`) if all the inputs are equivalent.

Examples

Compare Two Numeric Matrices

Create two numeric matrices and compare them for equality.

```
A = zeros(3,3)+1e-20;
B = zeros(3,3);
tf = isequal(A,B)
```

```
tf = logical
   0
```

The function returns logical `0` (`false`) because the matrices differ by a very small amount and are not *exactly* equal.

Compare Two Structures

Create two structures and specify the fields in a different order.

```
A = struct('field1',0.005,'field2',2500);
B = struct('field2',2500,'field1',0.005);
```

Compare the structures for equality.

```
tf = isequal(A,B)
```

```
tf = logical
   1
```

Even though the ordering of the fields in each structure is different, `isequal` treats them as the same because the values are equal.

Comparing Numeric Values with Special Nonnumeric Values

Compare the logical value `true` to the double integer `1`.

```
isequal(true,1)
```

```
ans = logical
   1
```

Notice that `isequal` does not consider data type when it tests for equality.

Similarly, compare `'A'` to the ASCII-equivalent integer, `65`.

```
isequal('A',65)
```

```
ans = logical
   1
```

The result is logical `1` (`true`) since `double('A')` equals `65`.

Compare Vectors Containing NaN Values

Create three vectors containing NaN values.

```
A1 = [1 NaN NaN];
A2 = [1 NaN NaN];
A3 = [1 NaN NaN];
```

Compare the vectors for equality.

```
tf = isequal(A1,A2,A3)
```

```
tf = logical
   0
```

The result is logical `0` (`false`) because `isequal` does not treat NaN values as equal to each other.

Compare Two Datetime Values

Determine if midnight on January 13, 2013 in Anchorage, Alaska is equal to 11 AM on the same date in Cairo.

```
t1 = datetime(2013,1,13,0,0,0,'TimeZone','America/Anchorage');
t2 = datetime(2013,1,13,11,0,0,'TimeZone','Africa/Cairo');
tf = isequal(t1,t2)
```

```
tf = logical
   1
```

Add 8 months to the date, and compare the datetime values for equality.

```
t1 = datetime(2013,9,13,0,0,0,'TimeZone','America/Anchorage');
t2 = datetime(2013,9,13,11,0,0,'TimeZone','Africa/Cairo');
tf = isequal(t1,t2)
```

```
tf = logical
   0
```

The datetime values are no longer equal since Cairo does not observe daylight saving time.

Compare Character Vector to String Scalar

Even though the sizes and data types are different, `isequal` returns logical `1` (`true`) when comparing a character vector and string scalar that contain the same sequence of characters.

```
isequal("foo",'foo')
```

```
ans = logical
   1
```

Input Arguments

`A,B` — Inputs to be compared (as separate arguments)
arrays

Inputs to be compared, specified as arrays.

In some cases, the types of the inputs do not have to match:

- Numeric inputs are equivalent if they are the same size and their contents are of equal value. The test compares both real and imaginary parts of numeric arrays.
- Tables, timetables, structures, and cell arrays are equivalent only when all elements and properties are equal.
- String scalars and character vectors containing the same sequence of characters are equivalent.

Some data type comparisons have special considerations involving metadata. If the inputs are *all*:

- Structures — Fields need not be in the same order as long as the contents are equal.
- Ordinal categorical arrays — Must have the same sets of categories, including their order.

- Categorical arrays that are not ordinal — Can have different sets of categories, and `isequal` compares the category names of each pair of elements.
- Datetime arrays — `isequal` ignores display format when it compares points in time. If the arrays are all associated with time zones, then `isequal` compares the instants in time rather than the clockface times (for example, `01-May-2018 09:00:00 EDT` is the same instant as `01-May-2018 06:00:00 PDT`, so `isequal` returns `true` even though the clockface times of 9:00 and 6:00 differ).
- Objects — `isequal` returns logical `1` (`true`) for objects of the same class with equal property values.

`A1,A2,...,An` — Series of inputs to be compared (as separate arguments)
arrays

Series of inputs to be compared, specified as arrays.

In some cases, the types of the inputs do not have to match:

- Numeric inputs are equivalent if they are the same size and their contents are of equal value. The test compares both real and imaginary parts of numeric arrays.
- Tables, timetables, structures, and cell arrays are equivalent only when all elements and properties are equal.
- String scalars and character vectors containing the same sequence of characters are equivalent.

Some data type comparisons have special considerations involving metadata. If the inputs are *all*:

- Structures — Fields need not be in the same order as long as the contents are equal.
- Ordinal categorical arrays — Must have the same sets of categories, including their order.
- Categorical arrays that are not ordinal — Can have different sets of categories, and `isequal` compares the category names of each pair of elements.
- Datetime arrays — `isequal` ignores display format when it compares points in time. If the arrays are all associated with time zones, then `isequal` compares the instants in time rather than the clockface times (for example, `01-May-2018 09:00:00 EDT` is the same instant as `01-May-2018 06:00:00 PDT`, so `isequal` returns `true` even though the clockface times of 9:00 and 6:00 differ).
- Objects — `isequal` returns logical `1` (`true`) for objects of the same class with equal property values.

Tips

- Use `isequaln` if you want to test for equality and treat `NaN`, `NaT`, or `<missing>` values as equal.
- The equality of two function handles depends on how they are constructed. For more information, see “Compare Function Handles”.
- `isequal` returns logical `0` (`false`) for two objects with dynamic properties, even if the properties have the same names and values.
- `isequal` compares only stored (non-dependent) properties when testing two objects for equality.
- When comparing two handle objects, use `==` to test whether objects have the same handle. Use `isequal` to determine if two objects with different handles have equal property values.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`eq` | `is*` | `isa` | `isequaln` | `strcmp`

Topics

“Determining Equality of Objects”

Introduced before R2006a

isenum

Determine if variable is enumeration

Syntax

```
tf = isenum(e)
```

Description

`tf = isenum(e)` returns logical 1 (`true`) if `e` is an enumeration. Otherwise, it returns logical 0 (`false`). Empty enumeration objects return `true`.

If `e` is a heterogeneous array, `isenum` always returns `false`.

Examples

Test for Enumeration

Determine if a variable is an enumeration.

The `PPM` class defines enumerations for three levels:

```
classdef PPM < int32
   enumeration
      High (1000)
      Medium (100)
      Low (10)
   end
end
```

Create a variable representing a level. Use `isenum` to determine if the variable is an enumeration:

```
currentLevel = PPM.High;
isenum(currentLevel)
```

```
ans =

     1
```

Input Arguments

e — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as any MATLAB variable.

More About

Enumerations

Enumerations are a fixed set of names representing a single type of value. For more information, see “Named Values”.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`islogical` | `isnumeric` | `isobject` | `isstruct`

Topics
“Enumerations”
“Define Enumeration Classes”

Introduced in R2015a

isequaln

Determine array equality, treating NaN values as equal

Syntax

```
tf = isequaln(A,B)
tf = isequaln(A1,A2,...,An)
```

Description

`tf = isequaln(A,B)` returns logical `1` (`true`) if `A` and `B` are equivalent; otherwise, it returns logical `0` (`false`). See the Input Arguments on page 1-8016 section for a definition of equivalence for each data type. `NaN` (Not a Number), `NaT` (Not a Time), undefined categorical elements, and `<missing>` values are considered to be *equal* to other such values.

`tf = isequaln(A1,A2,...,An)` returns logical `1` (`true`) if all the inputs are equivalent.

Examples

Compare Two Numeric Matrices

Create two numeric matrices and compare them for equality.

```
A = zeros(3,3)+1e-20;
B = zeros(3,3);
tf = isequaln(A,B)
```

```
tf = logical
   0
```

The function returns logical `0` (`false`) because the matrices differ by a very small amount and are not *exactly* equal.

Compare Two Structures

Create two structures and specify the fields in a different order.

```
A = struct('field1',0.005,'field2',2500);
B = struct('field2',2500,'field1',0.005);
```

Compare the structures for equality.

```
tf = isequaln(A,B)
```

```
tf = logical
   1
```

Even though the ordering of the fields in each structure is different, `isequaln` treats them as the same because the values are equal.

Comparing Numeric Values with Special Nonnumeric Values

Compare the logical value `true` to the double integer `1`.

```
isequaln(true,1)
```

```
ans = logical
   1
```

Notice that `isequaln` does not consider data type when it tests for equality.

Similarly, compare `'A'` to the ASCII-equivalent integer, `65`.

```
isequaln('A',65)
```

```
ans = logical
   1
```

The result is logical `1` (`true`) since `double('A')` equals `65`.

Compare Vectors Containing NaN Values

Create three vectors containing NaN values.

```
A1 = [1 NaN NaN];
A2 = [1 NaN NaN];
A3 = [1 NaN NaN];
```

Compare the vectors for equality.

```
tf = isequaln(A1,A2,A3)
```

```
tf = logical
   1
```

The result is logical `1` (`true`) because `isequaln` treats the NaN values as equal to each other.

Compare Character Vector to String Scalar

Even though the sizes and data types are different, `isequaln` returns logical `1` (`true`) when comparing a character vector and string scalar that contain the same sequence of characters.

```
isequaln("foo",'foo')
```

```
ans = logical
   1
```

Input Arguments

`A,B` — Inputs to be compared (as separate arguments)
arrays

Inputs to be compared, specified as arrays.

In some cases, the types of the inputs do not have to match:

- Numeric inputs are equivalent if they are the same size and their contents are of equal value. The test compares both real and imaginary parts of numeric arrays.
- Tables, timetables, structures, and cell arrays are equivalent only when all elements and properties are equal.
- String scalars and character vectors containing the same sequence of characters are equivalent.

Some data type comparisons have special considerations involving metadata. If the inputs are *all*:

- Structures — Fields need not be in the same order as long as the contents are equal.
- Ordinal categorical arrays — Must have the same sets of categories, including their order.
- Categorical arrays that are not ordinal — Can have different sets of categories, and `isequaln` compares the category names of each pair of elements.
- Datetime arrays — `isequaln` ignores display format when it compares points in time. If the arrays are all associated with time zones, then `isequaln` compares the instants in time rather than the clockface times (for example, `01-May-2018 09:00:00 EDT` is the same instant as `01-May-2018 06:00:00 PDT`, so `isequaln` returns `true` even though the clockface times of 9:00 and 6:00 differ).
- Objects — `isequaln` returns logical `1` (`true`) for objects of the same class with equal property values.

`A1,A2,...,An` — Series of inputs to be compared (as separate arguments)
arrays

Series of inputs to be compared, specified as arrays.

In some cases, the types of the inputs do not have to match:

- Numeric inputs are equivalent if they are the same size and their contents are of equal value. The test compares both real and imaginary parts of numeric arrays.
- Tables, timetables, structures, and cell arrays are equivalent only when all elements and properties are equal.
- String scalars and character vectors containing the same sequence of characters are equivalent.

Some data type comparisons have special considerations involving metadata. If the inputs are *all*:

- Structures — Fields need not be in the same order as long as the contents are equal.
- Ordinal categorical arrays — Must have the same sets of categories, including their order.
- Categorical arrays that are not ordinal — Can have different sets of categories, and `isequaln` compares the category names of each pair of elements.
- Datetime arrays — `isequaln` ignores display format when it compares points in time. If the arrays are all associated with time zones, then `isequaln` compares the instants in time rather than the clockface times (for example, `01-May-2018 09:00:00 EDT` is the same instant as `01-May-2018 06:00:00 PDT`, so `isequaln` returns `true` even though the clockface times of 9:00 and 6:00 differ).
- Objects — `isequaln` returns logical `1` (`true`) for objects of the same class with equal property values.

Tips

- Use `isequal` if you want to test for equality and treat `NaN`, `NaT`, or `<missing>` values as unequal.
- The equality of two function handles depends on how they are constructed. For more information, see “Compare Function Handles”.
- `isequaln` returns logical `0` (`false`) for two objects with dynamic properties, even if the properties have the same names and values.
- `isequaln` compares only stored (non-dependent) properties when testing two objects for equality.
- When comparing two handle objects, use `==` to test whether objects have the same handle. Use `isequaln` to determine if two objects with different handles have equal property values.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`eq` | `is*` | `isa` | `isequal` | `strcmp`

Topics

“Determining Equality of Objects”

Introduced in R2012a

isequalwithequalnans

Test arrays for equality, treating NaNs as equal

Note `isequalwithequalnans` is not recommended. Use `isequaln` instead.

Syntax

```
tf = isequalwithequalnans(A, B, ...)
```

Description

`tf = isequalwithequalnans(A, B, ...)` returns logical `1` (`true`) if the input arrays are the same type and size and hold the same contents, and logical `0` (`false`) otherwise. `NaN` (Not a Number) values are considered to be equal to each other. Numeric data types and structure field order do not have to match.

Examples

Arrays containing `NaN`s are handled differently by `isequal` and `isequalwithequalnans`. `isequal` does not consider `NaN`s to be equal, while `isequalwithequalnans` does.

```
A = [32 8 -29 NaN 0 5.7];
B = A;
isequal(A, B)
ans =
     0

isequalwithequalnans(A, B)
ans =
     1
```

The position of NaN elements in the array does matter. If they are not in the same position in the arrays being compared, then isequalwithequalnans returns zero.

```
A = [2 4 6 NaN 8];   B = [2 4 NaN 6 8];
isequalwithequalnans(A, B)
ans =
     0
```

Tips

isequalwithequalnans is the same as isequal, except isequalwithequalnans considers NaN (Not a Number) values to be equal, and isequal does not.

isequalwithequalnans recursively compares the contents of cell arrays and structures. If all the elements of a cell array or structure are numerically equal, isequalwithequalnans returns logical 1.

See Also

is* | isa | isequal | isequaln | strcmp

Introduced before R2006a

isevent

Determine whether input is COM object event

Syntax

```
tf = isevent(c,eventname)
```

Description

`tf = isevent(c,eventname)` returns logical `1` (`true`) if `eventname` is an event recognized by the COM object. Otherwise, it returns logical `0` (`false`).

Input Arguments

c — COM object
function handle

COM object, specified as a function handle.

eventname — Event name
string | character vector

Event name, specified as a string or a character vector. The `eventname` argument is not case-sensitive.

Limitations

- COM functions are available on Microsoft Windows systems only.

See Also

`eventlisteners` | `events (COM)` | `registerevent`

Topics

"Events"

Introduced before R2006a

isfield

Determine if input is structure array field

Syntax

```
TF = isfield(S,field)
```

Description

`TF = isfield(S,field)` returns `1` if `field` is the name of a field of the structure array `S`. Otherwise, it returns `0`.

If `field` is an array that contains multiple names, then `TF` is a logical array that has the same size.

Examples

Determine If Input Names Are Field Names

Create a structure.

```
S.x = linspace(0,2*pi);
S.y = sin(S.x);
S.title = 'y = sin(x)'
```

```
S = struct with fields:
        x: [1x100 double]
        y: [1x100 double]
    title: 'y = sin(x)'
```

Determine if `'title'` is the name of a field of `S`.

```
TF = isfield(S,'title')
```

```
TF = logical
   1
```

You can test multiple names in one call to the `isfield` function by specifying them in a cell array or string array.

```
field = {'x','y','z','title','error'};
TF = isfield(S,field)
```

```
TF = 1x5 logical array

   1   1   0   1   0
```

Input Arguments

S — Structure array
structure array

Structure array.

`field` — Potential field names
character vector | cell array of character vectors | string array

Potential field names, specified as a character vector, cell array of character vectors, or string array.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Cell arrays are not supported for the second argument.

See Also

`fieldnames` | `getfield` | `is*` | `isa` | `isstruct` | `orderfields` | `rmfield` | `setfield` | `struct`

Topics

"Access Data in Structure Array"
"Generate Field Names from Variables"

Introduced before R2006a

isfile

Determine if input is file

Syntax

```
result = isfile(fileName)
```

Description

`result = isfile(fileName)` returns 1 if `fileName` is a file located on the specified path or in the current folder. Otherwise, `isfile` returns 0.

Examples

Is Input a File

Check if the input `myfile1.txt` is a file. A result of 1 indicates that `myfile1.txt` is a file.

```
result = isfile('myfile1.txt')
```

```
result = logical
   1
```

Create the folder `myfolder`, then check if `myfolder` is a file. A result of 0 indicates that `myfolder` is not a file.

```
mkdir myfolder;
result = isfile('myfolder')
```

```
result = logical
   0
```

Check if the inputs `myfile1.txt` and `myfolder` are files. A result of `[1 0]` indicates that `myfile1.txt` is a file and `myfolder` is not a file.

```
result = isfile(["myfile1.txt", "myfolder"])
```

```
result = 1×2 logical array
   1   0
```

Input Arguments

`fileName` — File name

string array | character vector | cell array of character vector

File name, specified as a string array, character vector, or cell array of character vectors. For a local file, `fileName` can include a relative path, but the relative path must be in the current folder. Otherwise, `fileName` must include a full path. If the file is stored at a remote location, then `fileName` must contain a full path specified as a uniform resource locator (URL). For more information, see “Work with Remote Data”.

See Also

`dir` | `is*` | `isfolder`

Topics

“Work with Remote Data”

Introduced in R2017b

isfinite

Determine which array elements are finite

Syntax

```
TF = isfinite(A)
```

Description

`TF = isfinite(A)` returns a logical array containing `1` (`true`) where the elements of the array `A` are finite, and `0` (`false`) where they are infinite or `NaN`. If `A` contains complex numbers, `isfinite(A)` contains `1` for elements with finite real and imaginary parts, and `0` for elements where either part is infinite or `NaN`.

Examples

Determine Finite Real Elements

Create a row vector and determine the finite real elements.

```
A = 1./[-2 -1 0 1 2]
```

```
A = 1×5

   -0.5000   -1.0000       Inf    1.0000    0.5000
```

```
TF = isfinite(A)
```

```
TF = 1x5 logical array

   1   1   0   1   1
```

Determine Finite Complex Elements

Create an array of complex numbers. Determine whether the complex numbers contain finite values.

```
A = [2+1i 3+1i/0 0/0-2i]
```

```
A = 1×3 complex

   2.0000 + 1.0000i   3.0000 +    Infi      NaN - 2.0000i
```

```
TF = isfinite(A)
```

```
TF = 1x3 logical array

   1   0   0
```

The second element of `A` has an infinite imaginary part. The third element has a `NaN` real part. For both of these elements, `isfinite` returns 0.

Input Arguments

`A` — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Tips

If `x` is a real scalar, exactly one of `isfinite(x)`, `isinf(x)`, and `isnan(x)` returns logical `1` (`true`).

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`is*` | `isinf` | `isnan`

Introduced before R2006a

isfloat

Determine if input is floating-point array

Syntax

```
tf = isfloat(A)
```

Description

`tf = isfloat(A)` returns `true` if `A` is a floating-point array and `false` otherwise. The floating-point types are `single` and `double`, and subclasses of `single` and `double`.

Examples

These examples show the values `isfloat` returns when passed specific types:

```
% pi returns a floating-point value
isfloat(pi)
ans =
     1
% Complex numbers are floating-point values
isfloat(3+7i)
ans =
     1
% Single-precision numbers are floating-point values
isfloat(realmax('single'))
ans =
     1
% isfloat returns a logical value
isfloat(isfloat(pi))
ans =
     0
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`double` | `isa` | `isinteger` | `isnumeric` | `single`

Introduced before R2006a

isfolder

Determine if input is folder

Syntax

```
result = isfolder(folderName)
```

Description

`result = isfolder(folderName)` returns 1 if `folderName` is a folder located on the specified path or in the current folder. Otherwise, `isfolder` returns 0.

Examples

Is Input a Folder

Create the folder `myfolder`, then check if `myfolder` is a folder. A result of 1 indicates that `myfolder` is a folder.

```
mkdir myfolder;
result = isfolder('myfolder')
```

```
result = logical
   1
```

Check if the inputs `myfile1.txt` and `myfolder` are folders. A result of `[0 1]` indicates that `myfile1.txt` is not a folder and `myfolder` is a folder.

```
result = isfolder(["myfile1.txt", "myfolder"])
```

```
result = 1×2 logical array
   0   1
```

Input Arguments

`folderName` — Folder name
string array | character vector | cell array of character vector

Folder name, specified as a string array, character vector, or cell array of character vectors. For a local file, `folderName` can include a relative path, but the relative path must be in the current folder. Otherwise, `folderName` must include a full path. If the folder is at a remote location, then `folderName` must contain a full path specified as a uniform resource locator (URL). For more information, see “Work with Remote Data”.

See Also

`dir` | `is*` | `isfile`

Topics

“Work with Remote Data”

Introduced in R2017b

isgraphics

True for valid graphics object handles

Syntax

```
tf = isgraphics(H)
tf = isgraphics(H,type)
```

Description

`tf = isgraphics(H)` returns `true` for elements of `H` that are valid graphics objects and `false` where elements are not valid graphics objects or are graphics objects that have been deleted.

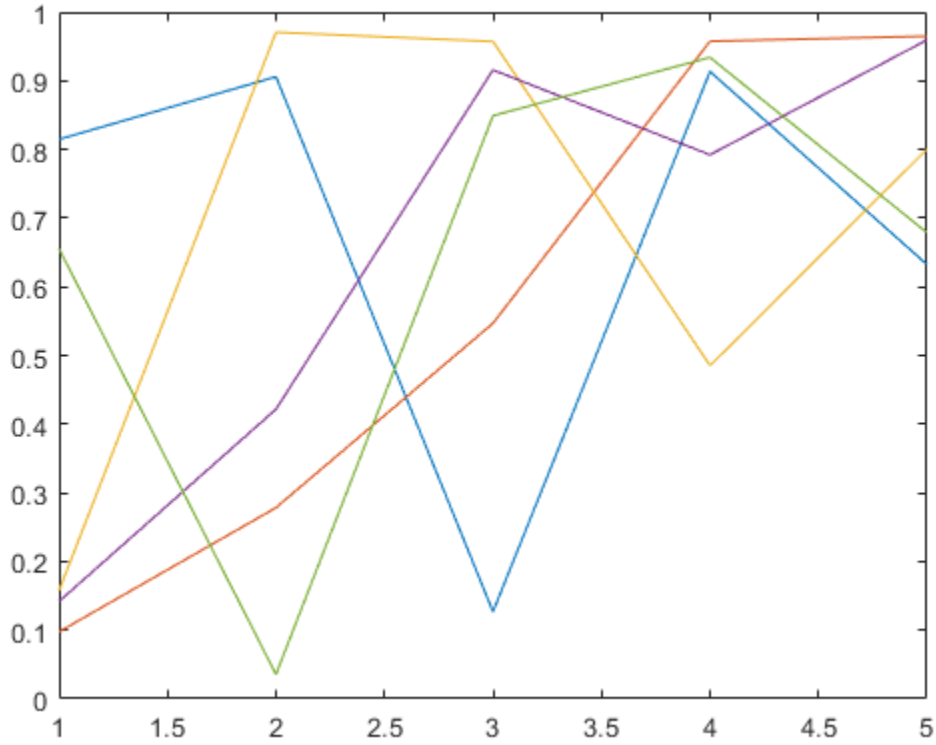
`tf = isgraphics(H,type)` returns `true` for elements of `H` that are valid graphics objects of the type specified by the `type` argument. An object type is the value contained in the object's `Type` property.

Examples

Test for Valid Handles

Create a plot and return the handle array. Test array for valid handles.

```
H = plot(rand(5));
```



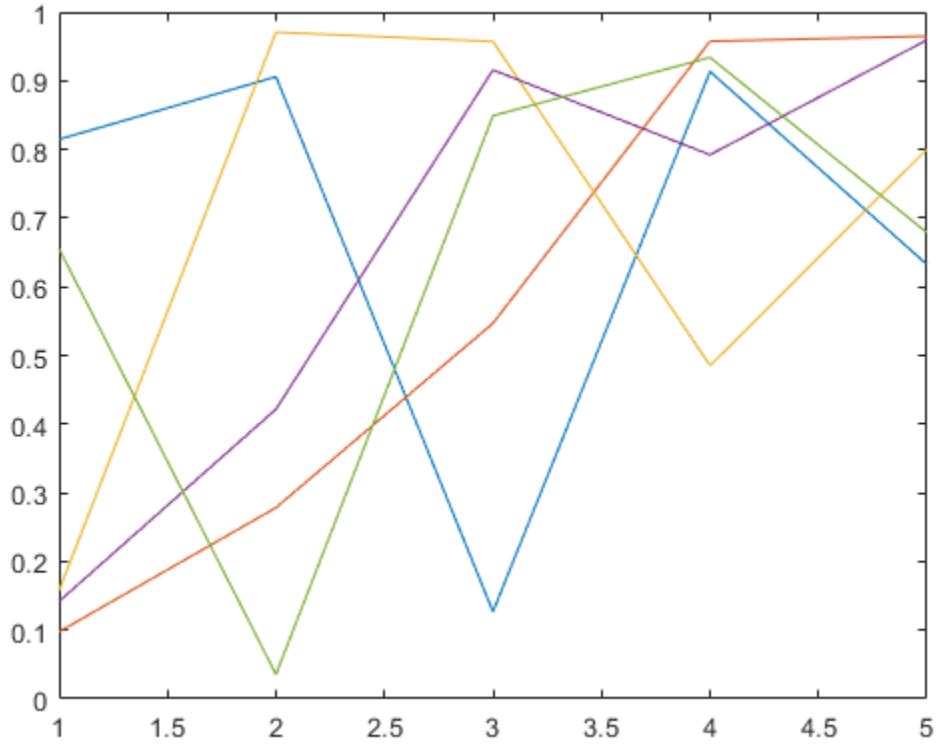

```
isgraphics(H)
```

```
ans = 5x1 logical array

   1
   1
   1
   1
   1
```

Test for Handle Types

Create a plot and return the handle array. Concatenate with other graphics objects and test for handles that are of type `Line`.

```
H = plot(rand(5));
```




```
a = [H;gca;gcf];
isgraphics(a,'line')
```

```
ans = 7x1 logical array

   1
   1
   1
```

```
1
1
0
0
```

Input Arguments

H — Input variable or expression
graphics object array

Input variable or expression that evaluates to graphics object handles.

type — Object type
character vector

Object type, specified as a character vector. An object's type is contained in its `Type` property.

Output Arguments

tf — result
logical array | empty `GraphicsPlaceholder` array

Result, returned as a logical array or an empty `GraphicsPlaceholder` array. If the result is a logical array, `1` indicates elements of the input array that are valid, and `0` indicates elements that are invalid. The result is an empty `GraphicsPlaceholder` array when the input array is empty.

See Also

Functions
`isa` | `ishghandle`

Topics
"Graphics Object Handles"

Introduced in R2014b

ishandle

Test for valid graphics or Java object handle

Syntax

```
ishandle(H)
```

Description

`ishandle(H)` returns an array whose elements are `1` where the elements of `H` are graphics or Java object handles, and `0` where they are not.

Note Use the most specific function for your application instead of `ishandle`, as described in the following sections.

MATLAB Handle Objects

Use the `isa` function to determine the class of MATLAB objects.

Use the `handle` class `isvalid` method to determine the validity of handle objects. See "Test Handle Validity" for information on testing for MATLAB handle objects.

Graphics Object Handles

Use `isgraphics` for graphics objects.

Java Object Handles

Use `isjava` for Java objects.

See Also

`isa` | `isgraphics` | `isjava`

Introduced before R2006a

ishermitian

Determine if matrix is Hermitian or skew-Hermitian

Syntax

```
tf = ishermitian(A)
tf = ishermitian(A,skewOption)
```

Description

`tf = ishermitian(A)` returns logical `1` (`true`) if square matrix `A` is Hermitian on page 1-8045; otherwise, it returns logical `0` (`false`).

`tf = ishermitian(A,skewOption)` specifies the type of the test. Specify `skewOption` as `'skew'` to determine if `A` is skew-Hermitian on page 1-8046.

Examples

Test if Symmetric Matrix Is Hermitian

Create a 3-by-3 matrix.

```
A = [1 0 1i; 0 1 0; 1i 0 1]
```

```
A = 3×3 complex

   1.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 1.0000i
   0.0000 + 0.0000i   1.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 1.0000i   0.0000 + 0.0000i   1.0000 + 0.0000i
```

The matrix is symmetric with respect to its real-valued diagonal.

Test whether the matrix is Hermitian.

```
tf = ishermitian(A)
```

```
tf = logical
   0
```

The result is logical `0` (`false`) because `A` is not Hermitian. In this case, `A` is equal to its transpose, `A.'`, but not its complex conjugate transpose, `A'`.

Change the element in `A(3,1)` to be `-1i`.

```
A(3,1) = -1i;
```

Determine if the modified matrix is Hermitian.

```
tf = ishermitian(A)
```

```
tf = logical
   1
```

The matrix, `A`, is now Hermitian because it is equal to its complex conjugate transpose, `A'`.

Test if Matrix Is Skew-Hermitian

Create a 3-by-3 matrix.

```
A = [-1i -1 1-i;1 -1i -1;-1-i 1 -1i]
```

```
A = 3×3 complex

   0.0000 - 1.0000i  -1.0000 + 0.0000i   1.0000 - 1.0000i
   1.0000 + 0.0000i   0.0000 - 1.0000i  -1.0000 + 0.0000i
  -1.0000 - 1.0000i   1.0000 + 0.0000i   0.0000 - 1.0000i
```

The matrix has pure imaginary numbers on the main diagonal.

Specify `skewOption` as `'skew'` to determine whether the matrix is skew-Hermitian.

```
tf = ishermitian(A,'skew')
```

```
tf = logical
   1
```

The matrix, `A`, is skew-Hermitian since it is equal to the negation of its complex conjugate transpose, `-A'`.

Input Arguments

`A` — Input matrix
numeric matrix

Input matrix, specified as a numeric matrix. If `A` is not square, then `ishermitian` returns logical `0` (`false`).

Data Types: `single` | `double` | `logical`
Complex Number Support: Yes

`skewOption` — Test type
`'nonskew'` (default) | `'skew'`

Test type, specified as `'nonskew'` or `'skew'`. Specify `'skew'` to test whether `A` is skew-Hermitian on page 1-8046.

More About

Hermitian Matrix

- A square matrix, `A`, is Hermitian if it is equal to its complex conjugate transpose, `A = A'`.

  In terms of the matrix elements, this means that

  $$a_{i,\ j} = \bar{a}_{j,\ i}\ .$$

- The entries on the diagonal of a Hermitian matrix are always real. Since real matrices are unaffected by complex conjugation, a real matrix that is symmetric is also Hermitian. For example, the matrix

$$A = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

is both symmetric and Hermitian.

- The eigenvalues of a Hermitian matrix are real.

Skew-Hermitian Matrix

- A square matrix, `A`, is skew-Hermitian if it is equal to the negation of its complex conjugate transpose, `A = -A'`.

  In terms of the matrix elements, this means that

  $$a_{i,\ j} = -\bar{a}_{j,\ i}\ .$$

- The entries on the diagonal of a skew-Hermitian matrix are always pure imaginary or zero. Since real matrices are unaffected by complex conjugation, a real matrix that is skew-symmetric is also skew-Hermitian. For example, the matrix

  $$A = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

  is both skew-Hermitian and skew-symmetric.

- The eigenvalues of a skew-Hermitian matrix are purely imaginary or zero.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`ctranspose` | `eig` | `isreal` | `issymmetric` | `transpose`

Introduced in R2014a

ishghandle

True for graphics object handles

Syntax

```
ishghandle(h)
```

Description

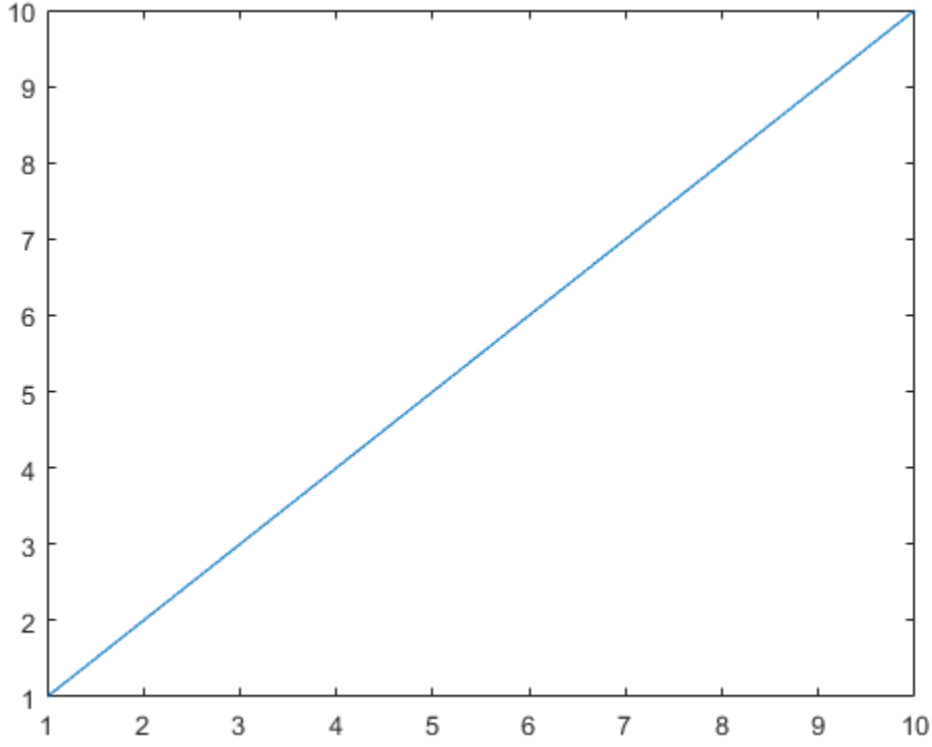
`ishghandle(h)` returns an array that contains 1's where the elements of `h` are handles to existing graphics objects and 0's where they are not. Differs from `ishandle` in that Simulink objects handles return false.

Examples

Test for Valid Graphics Object Handle

Create a plot. Test if the output `p` is a valid graphics object handle.

```
x = 1:10;
y = 1:10;
p = plot(x,y);
```




```
ishghandle(p)
```

```
ans = logical
   1
```

See Also

gca | isa | isgraphics

Topics

"Graphics Object Handles"

ishold

Current hold state

Syntax

```
tf = ishold
tf = ishold(ax)
```

Description

`tf = ishold` returns the hold state of the current axes. The return state is `1` if `hold` is `on`, and `0` if it is `off`. When `hold` is `on`, the current plot and most axes properties are held so that subsequent graphing commands add to the existing graph. If there is no figure, `ishold` returns `0` and MATLAB creates a new figure.

`tf = ishold(ax)` returns the hold state of the specified `Axes` object or `GeographicAxes` object.

A state of `hold on` implies that both figure and axes `NextPlot` properties are set to `'add'`.

See Also

`hold` | `newplot`

Topics

"Control Graph Display"

Introduced before R2006a

isinf

Determine which array elements are infinite

Syntax

```
TF = isinf(A)
```

Description

`TF = isinf(A)` returns a logical array containing `1` (`true`) where the elements of the array `A` are `Inf` or `-Inf`, and `0` (`false`) where they are not. If `A` contains complex numbers, `isinf(A)` contains `1` for elements with infinite real or imaginary part, and `0` for elements where both real and imaginary parts are finite or `NaN`.

Examples

Determine Infinite Real Elements

Create a row vector and determine the infinite elements.

```
A = 1./[-2 -1 0 1 2]
```

```
A = 1×5

   -0.5000   -1.0000       Inf    1.0000    0.5000
```

```
TF = isinf(A)
```

```
TF = 1x5 logical array

   0   0   1   0   0
```

Determine Infinite Complex Elements

Create an array of complex numbers. Determine whether the complex numbers contain infinite values.

```
A = [2 + 1i, 3 + 1i/0, 0/0 - 2i]
```

```
A = 1×3 complex

   2.0000 + 1.0000i   3.0000 +    Infi      NaN - 2.0000i
```

```
TF = isinf(A)
```

```
TF = 1x3 logical array

   0   1   0
```

The second element of `A` has an infinite imaginary part. For this element, `isinf` returns 1.

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Tips

- If `x` is a real scalar, exactly one of `isfinite(x)`, `isinf(x)`, and `isnan(x)` returns logical `1` (`true`).
- For a complex scalar `z`, `isinf(z)` and `isnan(z)` can both return logical 1. For example, `isinf(complex(Inf,NaN))` and `isnan(complex(Inf,NaN))` both return logical 1.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`is*` | `isfinite` | `isnan`

Introduced before R2006a

isinteger

Determine whether input is integer array

Syntax

```
TF = isinteger(A)
```

Description

`TF = isinteger(A)` returns logical `1` (`true`) if `A` is an array of integer type. Otherwise, it returns logical `0` (`false`).

Integer types in MATLAB include: `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, and `uint64`. For more information, see “Integer Classes”.

Examples

Determine Whether Real Number Is Integer Type

Determine if a real number is an integer type.

```
TF = isinteger(2)
```

```
TF = logical
   0
```

MATLAB® stores a real number as a `double` type by default.

Convert the number to a signed 8-bit integer type using the `int8` function. Check if it is an integer type.

```
TF = isinteger(int8(2))
```

```
TF = logical
   1
```

Determine Whether Complex Number Is Integer Type

Determine if a complex number is an integer type.

```
A = 3.5 - 2.5i
```

```
A = 3.5000 - 2.5000i
```

```
TF = isinteger(A)
```

```
TF = logical
   0
```

MATLAB stores a complex number as a `double` type by default.

Convert the complex number into a signed 32-bit integer type using the `int32` function. Check if it is an integer type.

```
B = int32(A)
```

```
B = int32
    4 - 3i
```

```
TF = isinteger(B)
```

```
TF = logical
   1
```

When a number with decimal digits is converted to an integer type, MATLAB rounds it to the nearest integer.

Determine Whether Arrays Are Integer Type

Determine if an array containing integer numbers is an integer type.

Create an array using the `int8` function. Check if it is an integer type.

```
A = [int8(1:5)]
```

```
A = 1x5 int8 row vector

   1   2   3   4   5
```

```
TF = isinteger(A)
```

```
TF = logical
   1
```

Now, create a cell array that contains the array `A` and other integer numbers. Use `class` to identify the type of the cell array. Check if it is an integer type.

```
B = {A int8(-4); int8(2) int8(1)}
```

```
B=2×2 cell
    {1x5 int8}    {[-4]}
    {[     2]}    {[ 1]}
```

```
type = class(B)
```

```
type =
'cell'
```

```
TF = isinteger(B)
```

```
TF = logical
   0
```

The cell array is not an integer type since it is a cell type.

Input Arguments

A — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. MATLAB has the following integer types.

Integer Type	Description
`int8`	8-bit signed integer
`int16`	16-bit signed integer
`int64`	64-bit signed integer
`int32`	32-bit signed integer
`uint8`	8-bit unsigned integer
`uint16`	16-bit unsigned integer
`uint32`	32-bit unsigned integer
`uint64`	64-bit unsigned integer

Tips

- For a floating-point number of a `single` or `double` type, you can check if it is also an integer by using the `round` function (within the floating-point relative accuracy `eps`). If the rounded value of the number is equal to the original value before rounding, then the number is an integer. For example, `2 == round(2)` returns logical `1` (`true`) since `2` is an integer.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`class` | `is*` | `isa` | `isfloat` | `isnumeric`

Topics

“Integers”
“Floating-Point Numbers”

Introduced before R2006a

isinterface

Determine whether input is COM interface

Syntax

```
tf = isinterface(c)
```

Description

`tf = isinterface(c)` returns logical `1` (`true`) if `c` is a COM interface. Otherwise, it returns logical `0` (`false`).

Examples

Test for Interface Objects

Test an instance of a Microsoft Excel application. MATLAB displays `false`, indicating the application is not an interface.

```
app = actxserver('Excel.Application');
isinterface(app)
```

Test a `workbooks` object. MATLAB displays `true`, indicating a workbook is an interface.

```
wbk = get(app,'workbooks');
isinterface(wbk)
```

Input Arguments

c — COM object
function handle

COM object, specified as a function handle.

Limitations

- COM functions are available on Microsoft Windows systems only.

See Also

`iscom`

Topics

"Interfaces"

Introduced before R2006a

isjava

Determine if input is Java object

Syntax

```
tf = isjava(A)
```

Description

`tf = isjava(A)` returns logical `1` (`true`) if object `A` is a Java object. Otherwise, it returns logical `0` (`false`).

Examples

Test If java.util.Date Is Java Object

Create an instance of the Java® Date class.

```
myDate = java.util.Date;
isjava(myDate)
```

```
ans = logical
   1
```

`myDate` is a Java object.

However, `myDate` is not a MATLAB® object.

```
isobject(myDate)
```

```
ans = logical
   0
```

Input Arguments

A — Input variable
any data type

Input variable, specified as any data type.

See Also

`is*` | `isa` | `isobject` | `javaArray` | `javaMethod` | `javaObject`

Introduced before R2006a

isKey

Package: `containers`

Determine if `Map` object contains key

Syntax

```
TF = isKey(M,keySet)
```

Description

`TF = isKey(M,keySet)` returns `1` (`true`) if `M` contains the specified key, and returns `0` (`false`) otherwise.

If `keySet` is an array that specifies multiple keys, then `TF` is a logical array of the same size.

Examples

Find Keys in Map

Create a `Map` object containing several key-value pairs.

```
months = {'Jan','Feb','Mar','Apr'};
rainfall = [327.2 368.2 197.6 178.4];
M = containers.Map(months,rainfall)
```

```
M = 
  Map with properties:

        Count: 4
      KeyType: char
    ValueType: double
```

Determine whether `m` has `'Feb'` as a key.

```
TF = isKey(M,'Feb')
```

```
TF = logical
   1
```

Search for multiple keys.

```
keySet = {'Mar','Apr','May','Jun'};
TF = isKey(M,keySet)
```

```
TF = 1x4 logical array

   1   1   0   0
```

Input Arguments

`M` — Input `Map` object
`Map` object

Input `Map` object.

`keySet` — Keys to search for
numeric scalar | character vector | string scalar | cell array

Keys to search for in the `Map` object, specified as a numeric scalar, character vector, string scalar, or cell array. To search for multiple keys, specify `keySet` as a cell array—even when you specify keys as numeric scalars or strings.

See Also

`containers.Map` | `keys` | `remove` | `values`

Topics

"Overview of Map Data Structure"
"Examine Contents of Map"
"Read and Write Using Key Index"

Introduced in R2008b

iskeyword

Determine whether input is MATLAB keyword

Syntax

```
tf = iskeyword(txt)
iskeyword
```

Description

`tf = iskeyword(txt)` returns logical `1` (`true`) if the character vector or string scalar `txt` is a keyword in the MATLAB language and logical `0` (`false`) otherwise. MATLAB keywords cannot be used as variable names.

`iskeyword` returns a list of all MATLAB keywords.

Examples

To test if the word `while` is a MATLAB keyword,

```
iskeyword('while')
ans =
    1
```

To obtain a list of all MATLAB keywords,

```
iskeyword
    'break'
    'case'
    'catch'
    'classdef'
    'continue'
    'else'
    'elseif'
    'end'
    'for'
```

```
'function'
'global'
'if'
'otherwise'
'parfor'
'persistent'
'return'
'spmd'
'switch'
'try'
'while'
```

See Also

`is*` | `isvarname` | `matlab.lang.makeUniqueStrings` | `matlab.lang.makeValidName`

Introduced before R2006a

isletter

Determine which characters are letters

Syntax

```
TF = isletter(A)
```

Description

`TF = isletter(A)` returns a logical array `TF`. If `A` is a character array or string scalar, then the elements of `TF` are logical `1` (`true`) where the corresponding characters in `A` are letters, and logical `0` (`false`) elsewhere.

If `A` is not a character array or string scalar, then `isletter` returns logical `0` (`false`).

Examples

Determine Which Character Array Elements Are Letters

Create different arrays, and then determine which array elements are letters.

```
chr = '123 Main St.'
```

```
chr =
'123 Main St.'
```

```
TF = isletter(chr)
```

```
TF = 1x12 logical array

   0   0   0   0   1   1   1   1   0   1   1   0
```

Starting in R2016b, you can convert character arrays to strings using the `string` function. Convert `chr` to a string, and test it. The input argument is not a character array, so `isletter` returns `0`.

```
str = string(chr)
```

```
str =
"123 Main St."
```

```
TF = isletter(str)
```

```
TF = 1x12 logical array

   0   0   0   0   1   1   1   1   0   1   1   0
```

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. `A` can be any data type.

Tips

To find letters within elements of a nonscalar string array, use the `isstrprop` function.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Input values from the `char` class must be in the range 0-127.

See Also

`char` | `is*` | `isa` | `iscellstr` | `ischar` | `isnumeric` | `isspace` | `isspace` | `isstring` | `isstrprop` | `string`

Topics

"Text in String and Character Arrays"
"Search and Replace Text"

Introduced before R2006a

islocalmax

Find local maxima

Syntax

```
TF = islocalmax(A)
TF = islocalmax(A,dim)
TF = islocalmax( ___ ,Name,Value)
[TF,P] = islocalmax( ___ )
```

Description

`TF = islocalmax(A)` returns a logical array whose elements are 1 (`true`) when a local maximum is detected in the corresponding element of an array, table, or timetable.

`TF = islocalmax(A,dim)` specifies the dimension of `A` to operate along. For example, `islocalmax(A,2)` finds local maxima of each row of a matrix `A`.

`TF = islocalmax( ___ ,Name,Value)` specifies additional parameters for finding local maxima using one or more name-value pair arguments. For example, `islocalmax(A,'SamplePoints',t)` finds local maxima of `A` with respect to the time stamps contained in the time vector `t`.

`[TF,P] = islocalmax( ___ )` also returns the prominence on page 1-8084 corresponding to each element of `A` for any of the previous syntaxes.

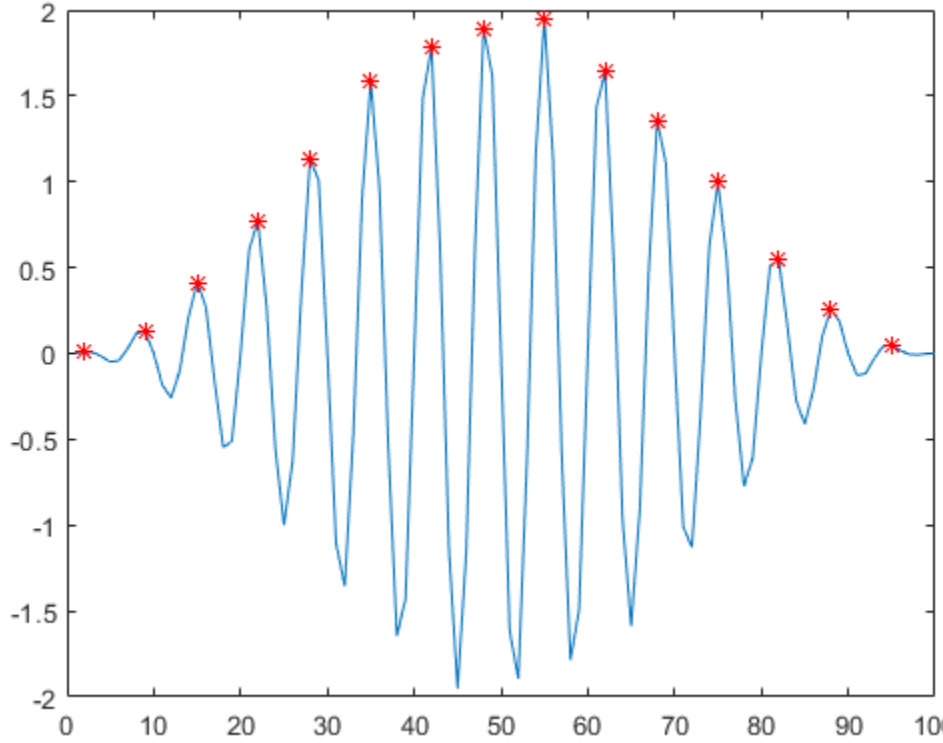
Examples

Local Maxima in Vector

Compute and plot the local maxima of a vector of data.

```
x = 1:100;
A = (1-cos(2*pi*0.01*x)).*sin(2*pi*0.15*x);
```

```
TF = islocalmax(A);
plot(x,A,x(TF),A(TF),'r*')
```



Maxima in Matrix Rows

Create a matrix of data, and compute the local maxima for each row.

```
A = 25*diag(ones(5,1)) + rand(5,5);
TF = islocalmax(A,2)
```

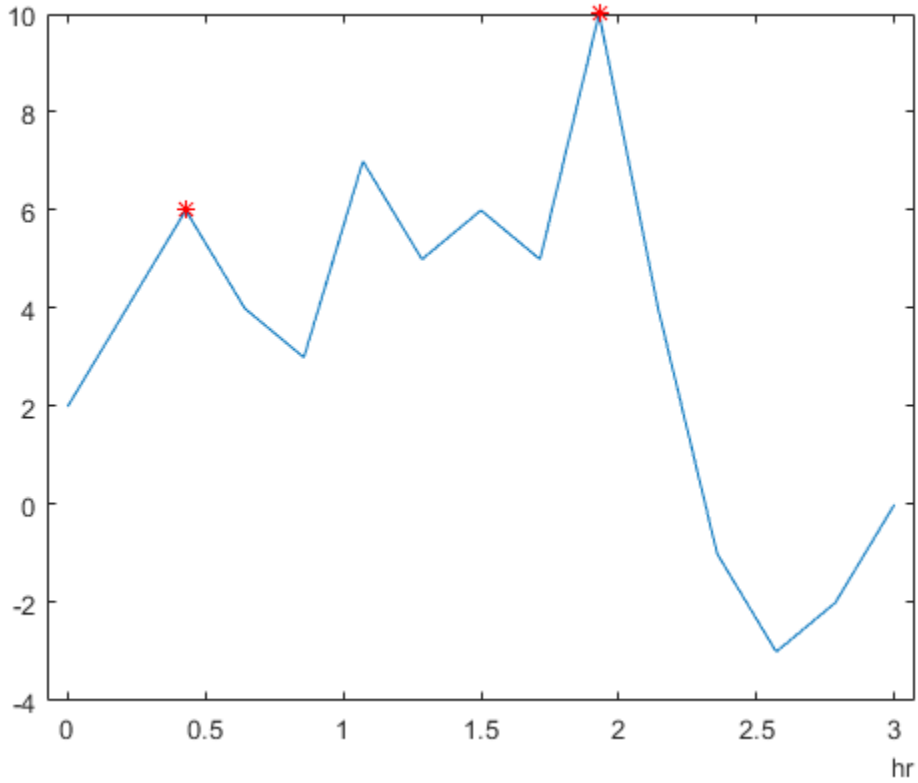
```
TF = 5x5 logical array
```

```
     0     0     1     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     1     0     1     0
     0     1     0     0     0
```

Separated Maxima

Compute the local maxima of a vector of data relative to the time stamps in the vector `t`. Use the `'MinSeparation'` parameter to compute maxima that are at least 45 minutes apart.

```
t = hours(linspace(0,3,15));
A = [2 4 6 4 3 7 5 6 5 10 4 -1 -3 -2 0];
TF = islocalmax(A,'MinSeparation',minutes(45),'SamplePoints',t);
plot(t,A,t(TF),A(TF),'r*')
```

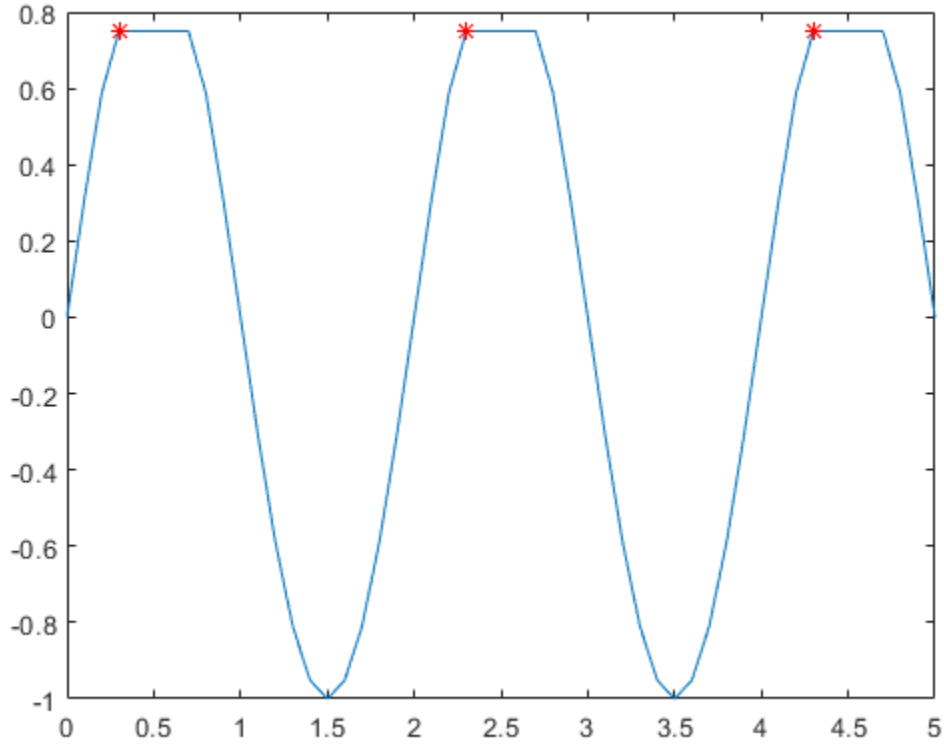


Flat Maxima Regions

Specify a method for indicating consecutive maxima values.

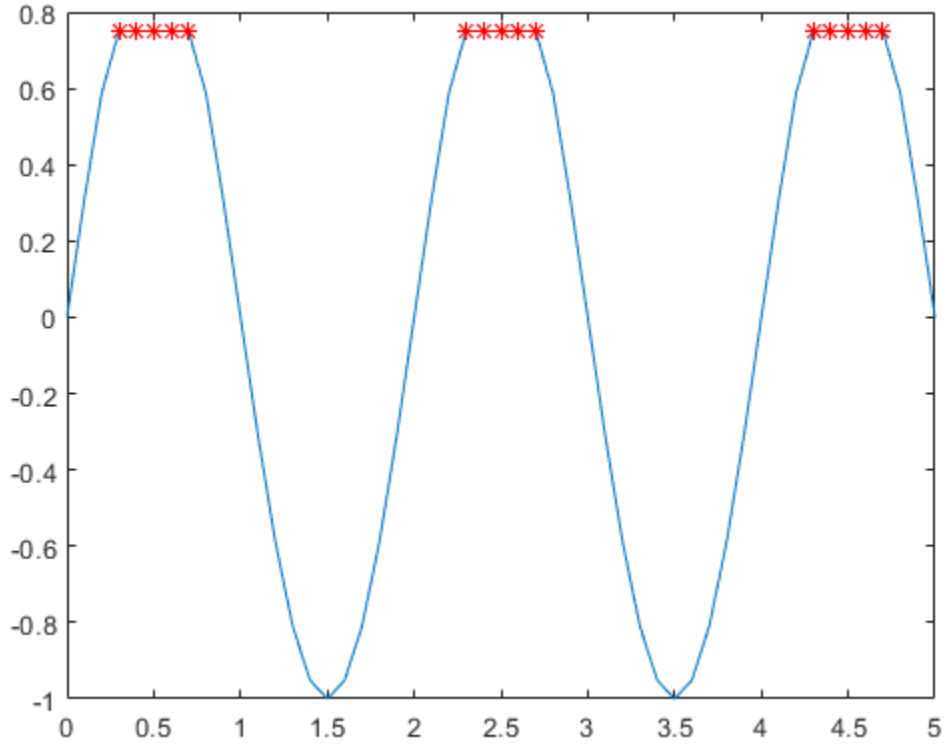
Compute the local maxima of data that contains consecutive maxima values. Indicate the maximum of each flat region based on the first occurrence of that value.

```
x = 0:0.1:5;
A = min(0.75, sin(pi*x));
TF1 = islocalmax(A, 'FlatSelection', 'first');
plot(x,A,x(TF1),A(TF1),'r*')
```



Indicate the maximum of each flat region with all occurrences of that value.

```
TF2 = islocalmax(A, 'FlatSelection', 'all');
plot(x,A,x(TF2),A(TF2),'r*')
```



Prominent Maxima

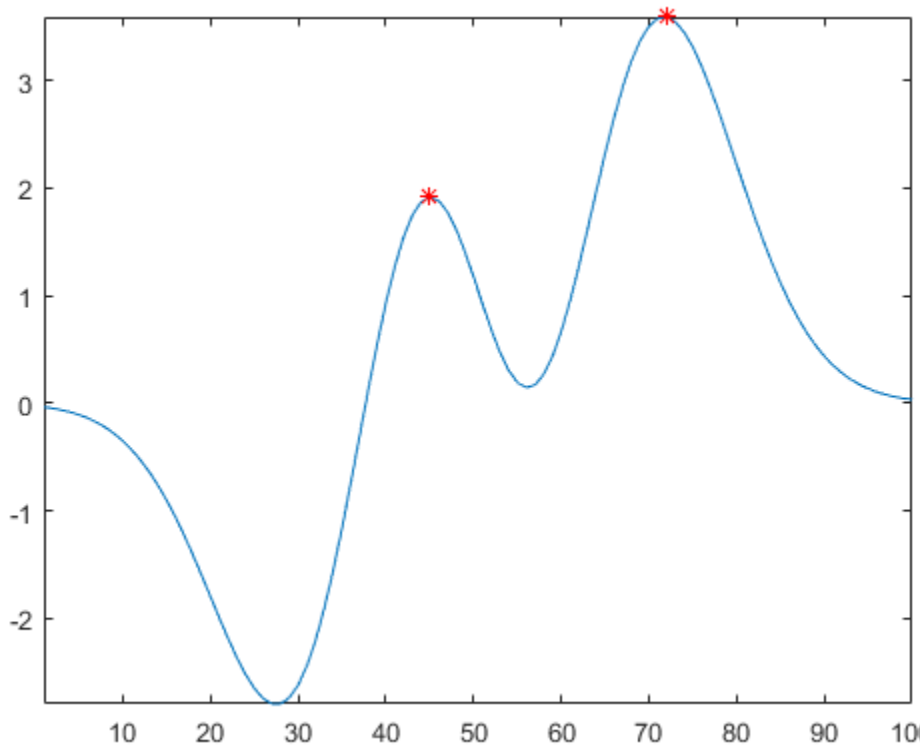
Select maxima based on their prominence.

Compute the local maxima of a vector of data and their prominence, and then plot them with the data.

```
x = 1:100;
A = peaks(100);
A = A(50,:);
[TF1,P] = islocalmax(A);
P(TF1)
```

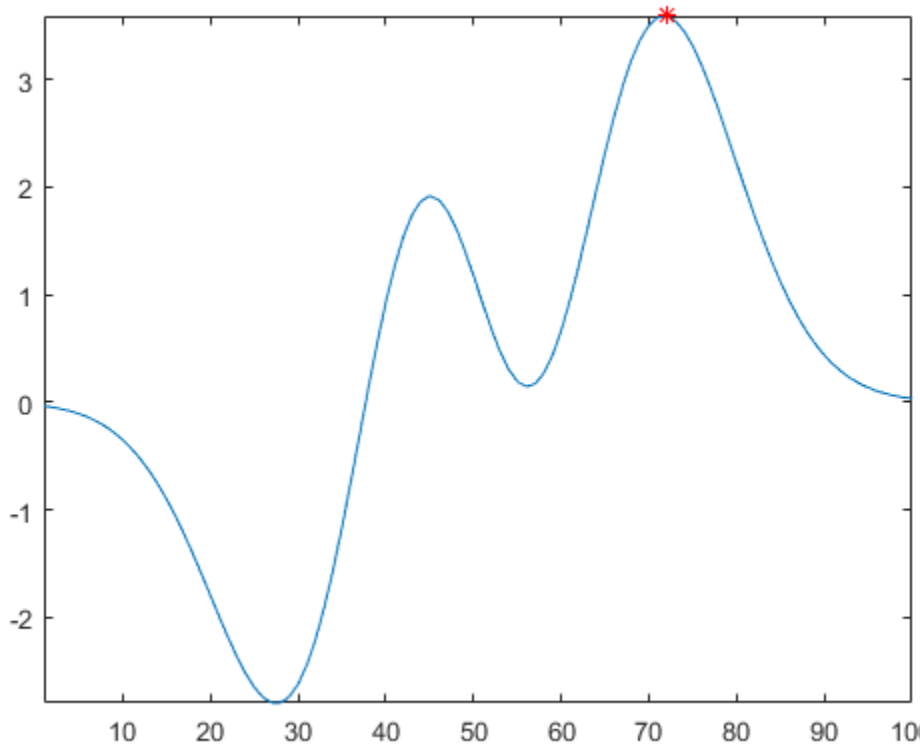
```
ans = 1×2

    1.7703    3.5548
```

```
plot(x,A,x(TF1),A(TF1),'r*')
axis tight
```



Compute only the most prominent maximum in the data by specifying a minimum prominence requirement.

```
TF2 = islocalmax(A,'MinProminence',2);
plot(x,A,x(TF2),A(TF2),'r*')
axis tight
```



Input Arguments

A — Input data
vector | matrix | multidimensional array | table | timetable

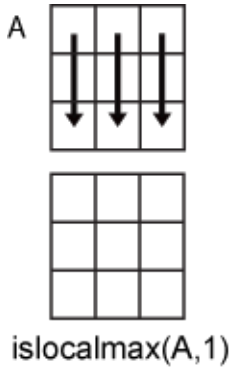
Input data, specified as a vector, matrix, multidimensional array, table, or timetable.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `table` | `timetable`

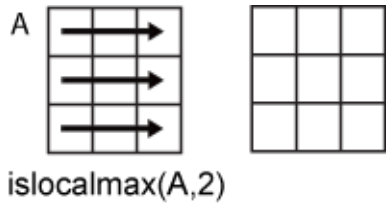
`dim` — Operating dimension
positive integer scalar

Operating dimension, specified as a positive integer scalar. By default, `islocalmax` operates along the first dimension whose size does not equal 1.

For example, if `A` is a matrix, then `islocalmax(A,1)` operates along the rows of `A`, computing local maxima for each column.



`islocalmax(A,2)` operates along the columns of `A`, computing local maxima for each row.



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `TF = islocalmax(A,'MinProminence',2)`

`MinProminence` — Minimum prominence
0 (default) | nonnegative scalar

Minimum prominence, specified as the comma-separated pair consisting of `'MinProminence'` and a nonnegative scalar. `islocalmax` returns only local maxima whose prominence is at least the value specified.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ProminenceWindow` — Prominence window
positive integer scalar | two-element vector of positive integers | positive duration scalar | two-element vector of positive durations

Prominence window, specified as the comma-separated pair consisting of `'ProminenceWindow'` and a positive integer scalar, a two-element vector of positive integers, a positive duration scalar, or a two-element vector of positive durations. The value defines a window of neighboring points for which to compute the prominence for each local maximum.

When the window value is a positive integer scalar *k*, then the window is centered about each local maximum and contains *k-1* neighboring elements. If *k* is even, then the window is centered about the current and previous elements. If a local maximum is within a flat region, then `islocalmax` treats the entire flat region as the center point of the window.

When the value is a two-element vector `[b f]` of positive integers, then the window contains the local maximum, `b` elements backward, and `f` elements forward. If a local maximum is within a flat region, then the window starts `b` elements before the first point of the region and ends `f` elements after the last point of the region.

When the input data is a timetable or when `'SamplePoints'` is specified as a `datetime` or `duration` vector, the window value must be of type `duration`, and the window is computed relative to the sample points.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`FlatSelection` — Flat region indicator
`'center'` (default) | `'first'` | `'last'` | `'all'`

Flat region indicator for when a local maximum value is repeated consecutively, specified as the comma-separated pair consisting of `'FlatSelection'` and one of the following:

- `'center'` — Indicate only the center element of a flat region as the local maximum. The element of `TF` corresponding to the center of the flat is 1, and is 0 for the remaining flat elements.
- `'first'` — Indicate only the first element of a flat region as the local maximum. The element of `TF` corresponding to the start of the flat is 1, and is 0 for the remaining flat elements.
- `'last'` — Indicate only the last element of a flat region as the local maximum. The element of `TF` corresponding to the end of the flat is 1, and is 0 for the remaining flat elements.
- `'all'` — Indicate all the elements of a flat region as the local maxima. The elements of `TF` corresponding to all parts of the flat are 1.

When using the `'MinSeparation'` or `'MaxNumExtrema'` name-value pairs, flat region points are jointly considered a single maximum point.

`MinSeparation` — Minimum separation
0 (default) | nonnegative scalar

Minimum separation between local maxima, specified as the comma-separated pair consisting of `'MinSeparation'` and a nonnegative scalar. The separation value is defined in the same units as the sample points vector, which is `[1 2 3 ...]` by default. When the separation value is greater than 0, `islocalmax` selects the largest local maximum and ignores all other local maxima within the specified separation. This process is repeated until there are no more local maxima detected.

When the sample points vector has type `datetime`, the separation value must have type `duration`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`MaxNumExtrema` — Maximum number of maxima
positive integer scalar

Maximum number of maxima detected, specified as the comma-separated pair consisting of `'MaxNumExtrema'` and a positive integer scalar. `islocalmax` finds no more than the specified number of most prominent maxima, which is the length of the operating dimension by default.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`SamplePoints` — Sample points
vector

Sample points, specified as the comma-separated pair consisting of `'SamplePoints'` and a vector. The sample points represent the location of the data in `A`. Sample points do not need to be uniformly sampled, but must be sorted with unique elements. By default, the sample points vector is `[1 2 3 ...]`.

`islocalmax` does not support this name-value pair when the input data is a timetable.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `datetime` | `duration`

`DataVariables` — Table variables
variable name | cell array of variable names | numeric vector | logical vector | function handle

Table variables, specified as the comma-separated pair consisting of `'DataVariables'` and a variable name, a cell array of variable names, a numeric vector, a logical vector, or a function handle. The `'DataVariables'` value indicates which columns of an input table or timetable to operate on. This value can be one of the following:

- A character vector specifying a single table variable name
- A cell array of character vectors where each element is a table variable name
- A vector of table variable indices
- A logical vector whose elements each correspond to a table variable, where `true` includes the corresponding variable and `false` excludes it
- A function handle that takes a table variable as input and returns a logical scalar

The specified table variables must have numeric or `logical` type.

Example: `'Age'`

Example: `{'Height','Weight'}`

Example: `@isnumeric`

Data Types: `char` | `cell` | `double` | `single` | `logical` | `function_handle`

Output Arguments

TF — Local maxima indicator
vector | matrix | multidimensional array

Local maxima indicator, returned as a vector, matrix, or multidimensional array. TF is the same size as A.

Data Types: `logical`

P — Prominence
vector | matrix | multidimensional array

Prominence on page 1-8084, returned as a vector, matrix, or multidimensional array. P is the same size as A.

If the input data has a signed or unsigned integer type, then P is an unsigned integer.

More About

Prominence of Local Maximum

The prominence of a local maximum (or peak) measures how the peak stands out with respect to its height and location relative to other peaks.

To measure the prominence of a peak, first extend a horizontal line from the peak to the left and to the right of the peak. Find where the line intersects the data on the left and on the right, which will either be another peak or the end of the data. Mark these locations as the outer endpoints of the left and right intervals. Next, find the lowest valley in both intervals. Take the larger of these two valleys, and measure the vertical distance from that valley to the peak. This distance is the prominence.

For a vector x, the largest prominence is at most `max(x)-min(x)`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- Tall timetables are not supported.
- You must specify a value for the `'ProminenceWindow'` name-value pair.
- The `'MaxNumExtrema'`, `'MinSeparation'`, and `'SamplePoints'` name-value pairs are not supported.
- The value of `'DataVariables'` cannot be a function handle.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- You must enable support for variable-size arrays.
- The `'ProminenceWindow'` name-value pair is not supported.
- Tables are not supported.

See Also

Find Local Extrema | `ischange` | `islocalmin` | `isoutlier` | `max`

Topics

Clean Messy Data and Locate Extrema Using Live Editor Tasks

Introduced in R2017b

islocalmin

Find local minima

Syntax

```
TF = islocalmin(A)
TF = islocalmin(A,dim)
TF = islocalmin( ___ ,Name,Value)
[TF,P] = islocalmin( ___ )
```

Description

`TF = islocalmin(A)` returns a logical array whose elements are 1 (`true`) when a local minimum is detected in the corresponding element of an array, table, or timetable.

`TF = islocalmin(A,dim)` specifies the dimension of `A` to operate along. For example, `islocalmin(A,2)` finds local minima of each row of a matrix `A`.

`TF = islocalmin( ___ ,Name,Value)` specifies additional parameters for finding local minima using one or more name-value pair arguments. For example, `islocalmin(A,'SamplePoints',t)` finds local minima of `A` with respect to the time stamps contained in the time vector `t`.

`[TF,P] = islocalmin( ___ )` also returns the prominence on page 1-8098 corresponding to each element of `A` for any of the previous syntaxes.

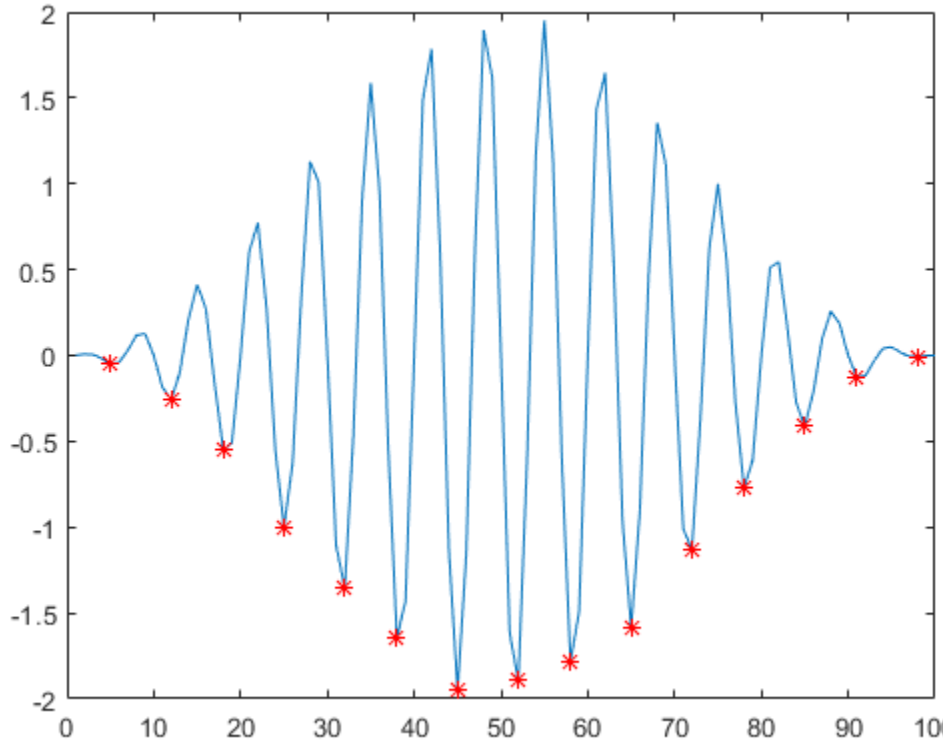
Examples

Local Minima in Vector

Compute and plot the local minima of a vector of data.

```
x = 1:100;
A = (1-cos(2*pi*0.01*x)).*sin(2*pi*0.15*x);
```

```
TF = islocalmin(A);
plot(x,A,x(TF),A(TF),'r*')
```



Minima in Matrix Rows

Create a matrix of data, and compute the local minima for each row.

```
A = -25*diag(ones(5,1)) + rand(5,5);
TF = islocalmin(A,2)
```

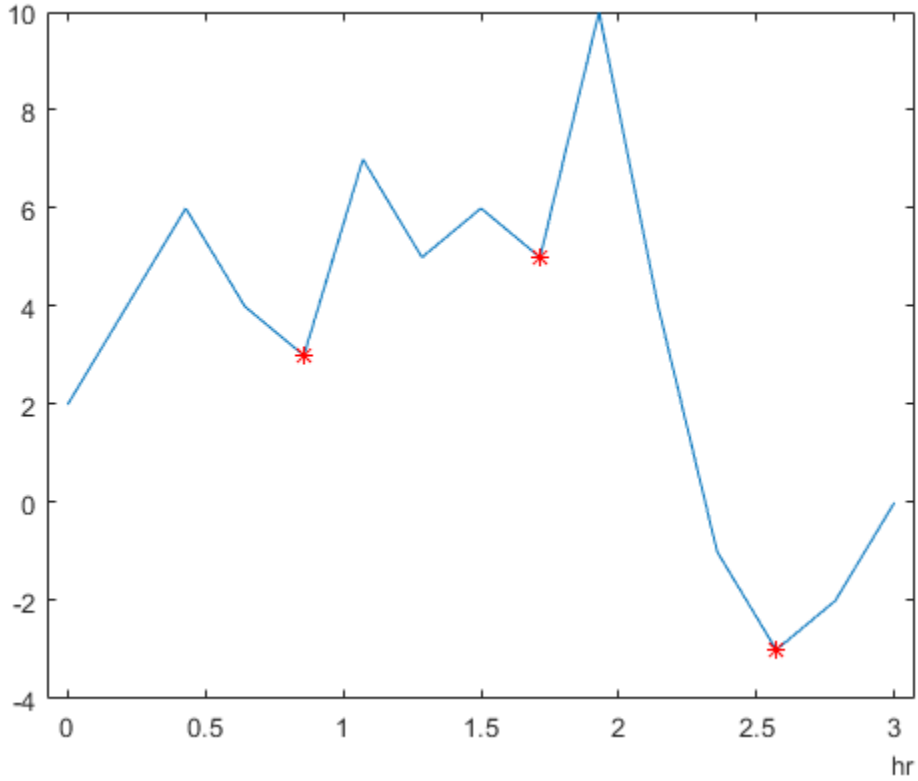
```
TF = 5x5 logical array
```

```
     0     0     0     1     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     1     0     0
```

Separated Minima

Compute the local minima of a vector of data relative to the time stamps in the vector `t`. Use the `'MinSeparation'` parameter to compute minima that are at least 45 minutes apart.

```
t = hours(linspace(0,3,15));
A = [2 4 6 4 3 7 5 6 5 10 4 -1 -3 -2 0];
TF = islocalmin(A,'MinSeparation',minutes(45),'SamplePoints',t);
plot(t,A,t(TF),A(TF),'r*')
```

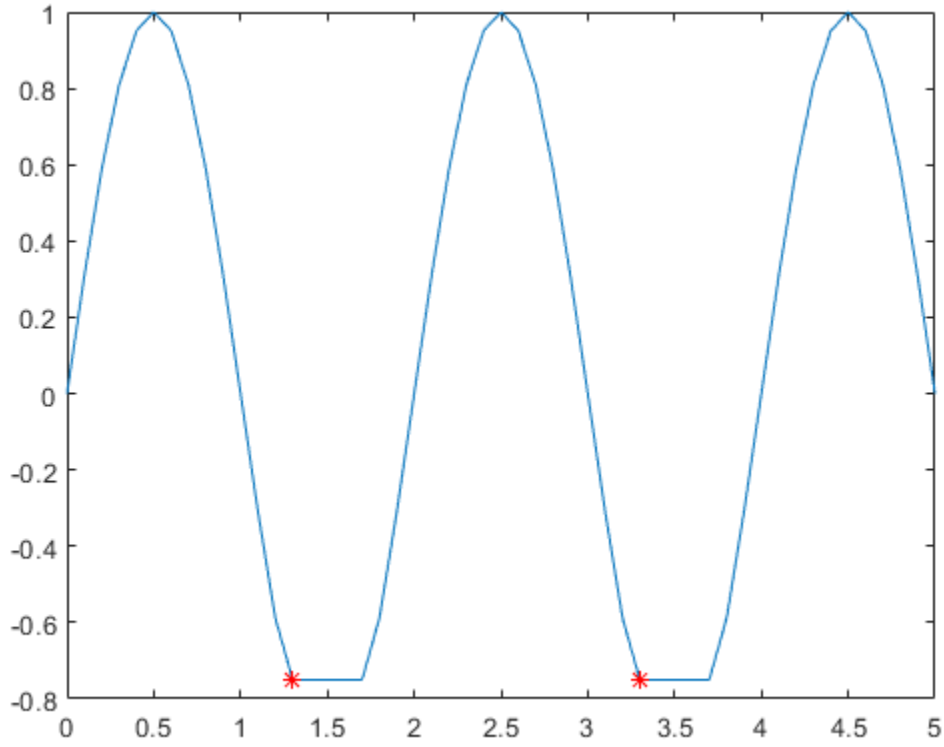


Flat Minima Regions

Specify a method for indicating consecutive minima values.

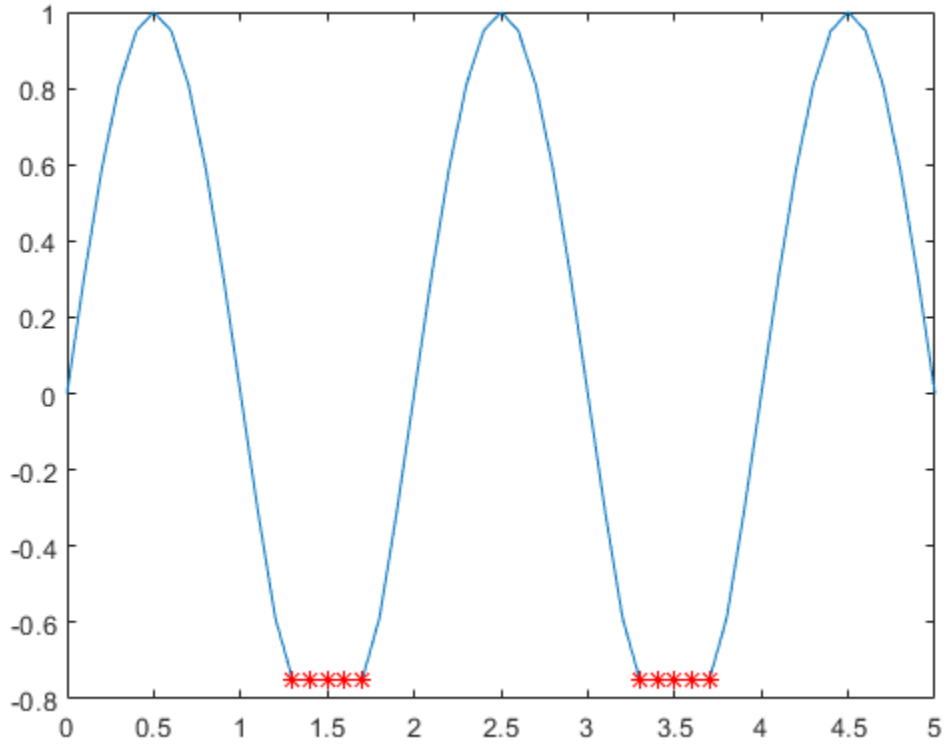
Compute the local minima of data that contains consecutive minima values. Indicate the minimum of each flat region based on the first occurrence of that value.

```
x = 0:0.1:5;
A = max(-0.75, sin(pi*x));
TF1 = islocalmin(A, 'FlatSelection', 'first');
plot(x,A,x(TF1),A(TF1),'r*')
```



Indicate the minimum of each flat region with all occurrences of that value.

```
TF2 = islocalmin(A, 'FlatSelection', 'all');
plot(x,A,x(TF2),A(TF2),'r*')
```



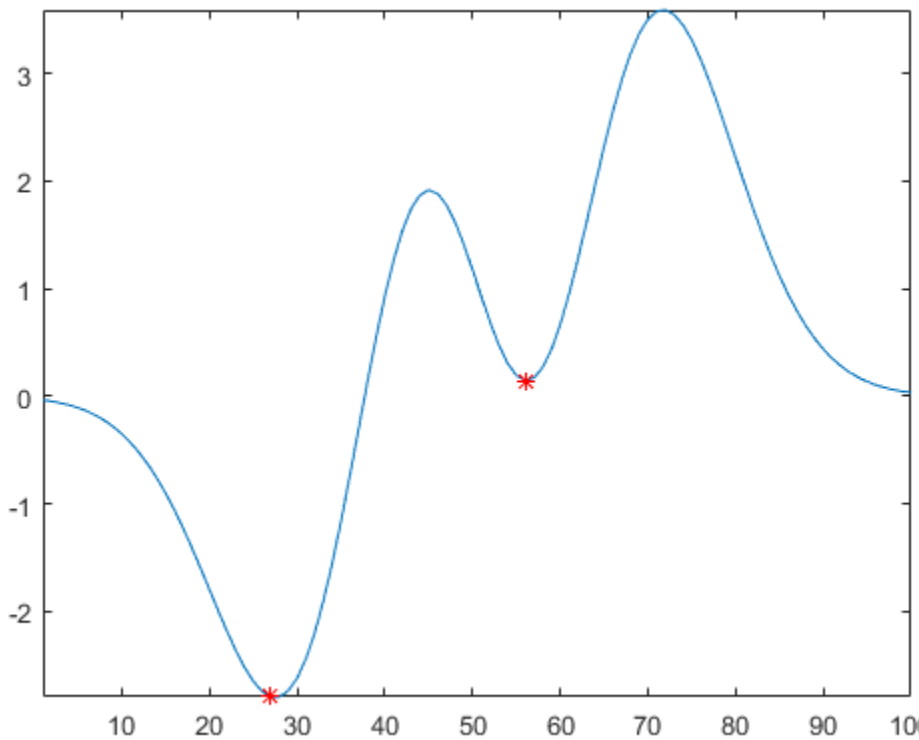
Prominent Minima

Compute the local minima of a vector of data and their prominence, and then plot them with the data.

```
x = 1:100;
A = peaks(100);
A = A(50,:);
[TF1,P] = islocalmin(A);
P(TF1)
```

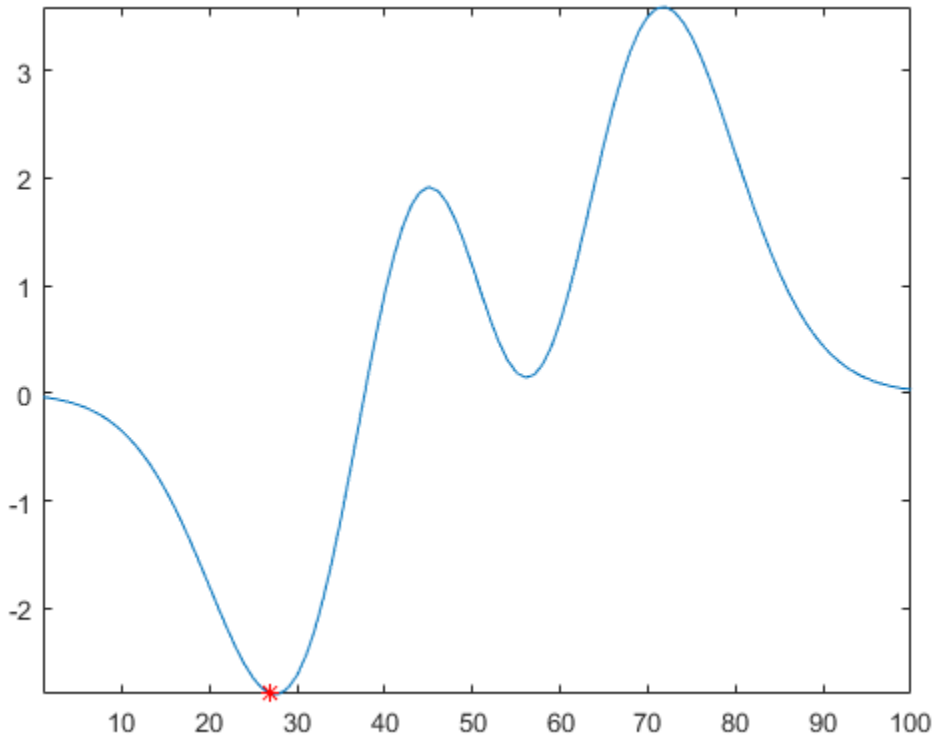
```
ans = 1×2

    2.7585    1.7703
```

```
plot(x,A,x(TF1),A(TF1),'r*')
axis tight
```



Compute the most prominent minimum in the data by specifying a minimum prominence requirement.

```
TF2 = islocalmin(A,'MinProminence',2);
plot(x,A,x(TF2),A(TF2),'r*')
axis tight
```



Input Arguments

A — Input data
vector | matrix | multidimensional array | table | timetable

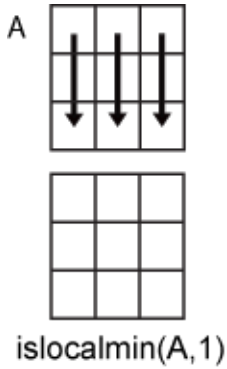
Input data, specified as a vector, matrix, multidimensional array, table, or timetable.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `table` | `timetable`

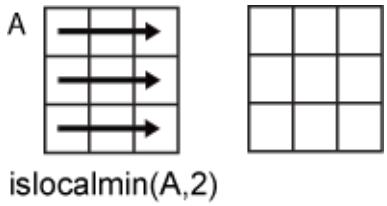
`dim` — Operating dimension
positive integer scalar

Operating dimension, specified as a positive integer scalar. By default, `islocalmin` operates along the first dimension whose size does not equal 1.

For example, if `A` is a matrix, then `islocalmin(A,1)` operates along the rows of `A`, computing local minima for each column.



`islocalmin(A,2)` operates along the columns of `A`, computing local minima for each row.



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `TF = islocalmin(A,'MinProminence',2)`

`MinProminence` — Minimum prominence
0 (default) | nonnegative scalar

Minimum prominence, specified as the comma-separated pair consisting of `'MinProminence'` and a nonnegative scalar. `islocalmin` returns only local minima whose prominence is at least the value specified.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ProminenceWindow` — Prominence window
positive integer scalar | two-element vector of positive integers | positive duration scalar | two-element vector of positive durations

Prominence window, specified as the comma-separated pair consisting of `'ProminenceWindow'` and a positive integer scalar, a two-element vector of positive integers, a positive duration scalar, or a two-element vector of positive durations. The value defines a window of neighboring points for which to compute the prominence for each local minimum.

When the window value is a positive integer scalar *k*, then the window is centered about each local minimum and contains *k-1* neighboring elements. If *k* is even, then the window is centered about the current and previous elements. If a local minimum is within a flat region, then `islocalmin` treats the entire flat region as the center point of the window.

When the value is a two-element vector `[b f]` of positive integers, then the window contains the local minimum, `b` elements backward, and `f` elements forward. If a local minimum is within a flat region, then the window starts `b` elements before the first point of the region and ends `f` elements after the last point of the region.

When the input data is a timetable or when `'SamplePoints'` is specified as a `datetime` or `duration` vector, the window value must be of type `duration`, and the window is computed relative to the sample points.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`FlatSelection` — Flat region indicator
`'center'` (default) | `'first'` | `'last'` | `'all'`

Flat region indicator for when a local minimum value is repeated consecutively, specified as the comma-separated pair consisting of `'FlatSelection'` and one of the following:

- `'center'` — Indicate only the center element of a flat region as the local minimum. The element of `TF` corresponding to the center of the flat is 1, and is 0 for the remaining flat elements.
- `'first'` — Indicate only the first element of a flat region as the local minimum. The element of `TF` corresponding to the start of the flat is 1, and is 0 for the remaining flat elements.
- `'last'` — Indicate only the last element of a flat region as the local minimum. The element of `TF` corresponding to the end of the flat is 1, and is 0 for the remaining flat elements.
- `'all'` — Indicate all the elements of a flat region as the local minima. The elements of `TF` corresponding to all parts of the flat are 1.

When using the `'MinSeparation'` or `'MaxNumExtrema'` name-value pairs, flat region points are jointly considered a single minimum point.

`MinSeparation` — Minimum separation

0 (default) | nonnegative scalar

Minimum separation between local minima, specified as the comma-separated pair consisting of `'MinSeparation'` and a nonnegative scalar. The separation value is defined in the same units as the sample points vector, which is `[1 2 3 ...]` by default. When the separation value is greater than 0, `islocalmin` selects the smallest local minimum and ignores all other local minima within the specified separation. This process is repeated until there are no more local minima detected.

When the sample points vector has type `datetime`, the separation value must have type `duration`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`MaxNumExtrema` — Maximum number of minima

positive integer scalar

Maximum number of minima detected, specified as the comma-separated pair consisting of `'MaxNumExtrema'` and a positive integer scalar. `islocalmin` finds no more than the specified number of most prominent minima, which is the length of the operating dimension by default.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`SamplePoints` — Sample points
vector

Sample points, specified as the comma-separated pair consisting of `'SamplePoints'` and a vector. The sample points represent the location of the data in `A`. Sample points do not need to be uniformly sampled, but must be sorted with unique elements. By default, the sample points vector is `[1 2 3 ...]`.

`islocalmin` does not support this name-value pair when the input data is a timetable.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `datetime` | `duration`

`DataVariables` — Table variables
variable name | cell array of variable names | numeric vector | logical vector | function handle

Table variables, specified as the comma-separated pair consisting of `'DataVariables'` and a variable name, a cell array of variable names, a numeric vector, a logical vector, or a function handle. The `'DataVariables'` value indicates which columns of an input table or timetable to operate on. This value can be one of the following:

- A character vector specifying a single table variable name
- A cell array of character vectors where each element is a table variable name
- A vector of table variable indices
- A logical vector whose elements each correspond to a table variable, where `true` includes the corresponding variable and `false` excludes it
- A function handle that takes a table variable as input and returns a logical scalar

The specified table variables must have numeric or `logical` type.

Example: `'Age'`

Example: `{'Height','Weight'}`

Example: `@isnumeric`

Data Types: `char` | `cell` | `double` | `single` | `logical` | `function_handle`

Output Arguments

TF — Local minima indicator
vector | matrix | multidimensional array

Local minima indicator, returned as a vector, matrix, or multidimensional array. TF is the same size as A.

Data Types: `logical`

P — Prominence
vector | matrix | multidimensional array

Prominence on page 1-8098, returned as a vector, matrix, or multidimensional array. P is the same size as A.

If the input data has a signed or unsigned integer type, then P is an unsigned integer.

More About

Prominence of Local Minimum

The prominence of a local minimum (or valley) measures how the valley stands out with respect to its depth and location relative to other valleys.

To measure the prominence of a valley, first extend a horizontal line from the valley to the left and to the right of the valley. Find where the line intersects the data on the left and on the right, which will either be another valley or the end of the data. Mark these locations as the outer endpoints of the left and right intervals. Next, find the highest peak in both the left and right intervals. Take the smaller of these two peaks, and measure the vertical distance from that peak to the valley. This distance is the prominence.

For a vector x, the largest prominence is at most `max(x)-min(x)`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- Tall timetables are not supported.
- You must specify a value for the `'ProminenceWindow'` name-value pair.
- The `'MaxNumExtrema'`, `'MinSeparation'`, and `'SamplePoints'` name-value pairs are not supported.
- The value of `'DataVariables'` cannot be a function handle.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- You must enable support for variable-size arrays.
- The `'ProminenceWindow'` name-value pair is not supported.
- Tables are not supported.

See Also

Find Local Extrema | `ischange` | `islocalmax` | `isoutlier` | `min`

Topics

Clean Messy Data and Locate Extrema Using Live Editor Tasks

Introduced in R2017b

islogical

Determine if input is logical array

Syntax

```
tf = islogical(A)
```

Description

`tf = islogical(A)` returns `true` if `A` is a logical array and `false` otherwise. `islogical` also returns `true` if `A` is an instance of a class that is derived from the `logical` class.

Examples

These examples show the values `islogical` returns when passed specific types:

```
% Relational operators return logical values
islogical(5<7)
ans =
     1

% true and false return logical values
islogical(true) & islogical(false)
ans =
     1

% Constants are double-precision by default
islogical(1)
ans =
     0

% logical creates logical values
islogical(logical(1))
ans =
     1
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`is*` | `isa` | `logical`

Introduced before R2006a

ismac

Determine if version is for macOS platform

Syntax

```
tf = ismac
```

Description

`tf = ismac` returns logical `1` (`true`) if the version of MATLAB software is for the Apple macOS platform. Otherwise, it returns logical `0` (`false`).

Examples

Execute MATLAB Command Based on Platform

```
if ismac
    % Code to run on Mac platform
elseif isunix
    % Code to run on Linux platform
elseif ispc
    % Code to run on Windows platform
else
    disp('Platform not supported')
end
```

Tips

- The `isunix` function also determines if version is for macOS platforms.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Returns true or false based on the MATLAB version used for code generation.
- Use only when the code generation target is S-function (Simulation) or MEX-function.

See Also

`computer` | `is*` | `ispc` | `isstudent` | `isunix`

Introduced in R2007a

ismatrix

Determine whether input is matrix

Syntax

```
TF = ismatrix(A)
```

Description

`TF = ismatrix(A)` returns logical `1` (`true`) if `A` is a matrix. Otherwise, it returns logical `0` (`false`). A matrix is a two-dimensional array that has a size of *m*-by-*n*, where *m* and *n* are nonnegative integers.

Examples

Arrays of Different Sizes

Determine whether arrays of different sizes are matrices.

Create an array of size 1-by-3. Determine whether it is a matrix.

```
A1 = zeros(1,3);
TF = ismatrix(A1)
```

```
TF = logical
   1
```

Create an empty array of size 0-by-3. Determine whether it is a matrix. A 2-D empty array is a matrix.

```
A2 = zeros(0,3);
TF = ismatrix(A2)
```

```
TF = logical
   1
```

Create an array of size 1-by-3-by-2. Determine whether it is a matrix. A 3-D array is not a matrix.

```
A3 = zeros(1,3,2);
TF = ismatrix(A3)
```

```
TF = logical
   0
```

Determine Matrix from Three-Dimensional Array

Create a 3-D array and determine whether the array elements are a matrix.

First define a 2-D array of size 2-by-3. Determine whether it is a matrix.

```
A = [0.1 0.2 0.5; 0.3 0.6 0.4]
```

```
A = 2×3

    0.1000    0.2000    0.5000
    0.3000    0.6000    0.4000
```

```
TF = ismatrix(A)
```

```
TF = logical
   1
```

To create a 3-D array, add a third dimension to the array `A`. Assign another 2-by-3 matrix to the third dimension of `A` with index value 2.

```
A(:,:,2) = ones(2,3)
```

```
A =
A(:,:,1) =

    0.1000    0.2000    0.5000
```

```
    0.3000    0.6000    0.4000


A(:,:,2) =

     1     1     1
     1     1     1
```

Check whether the 3-D array of size 2-by-3-by-2 is a matrix.

```
TF = ismatrix(A)
```

```
TF = logical
   0
```

Now determine whether the array elements of `A` are a matrix. Check whether the second page of the 3-D array is a matrix. The syntax `A(:,:,2)` uses a colon in the first and second dimensions to access all rows and all columns.

```
TF = ismatrix(A(:,:,2))
```

```
TF = logical
   1
```

Check whether the second row of the 3-D array is a matrix. The syntax `A(2,:,:)` uses a colon in the second and third dimensions to include all columns and all pages.

```
TF = ismatrix(A(2,:,:))
```

```
TF = logical
   0
```

`A(:,:,2)` is a matrix since it is a multidimensional array of size 2-by-3. However, `A(2,:,:)` is not a matrix since it is a multidimensional array of size 1-by-3-by-2.

Determine Matrix from Character Vector and String

Create an array of characters. Determine whether it is a matrix.

```
A = 'Hello, World!';
TF = ismatrix(A)
```

```
TF = logical
   1
```

Check the dimension of A using `size`. A is a matrix of size 1-by-13.

```
size(A)
```

```
ans = 1×2

     1    13
```

Now create a string scalar by enclosing a piece of text in double quotes.

```
A = "Hello, World!";
```

Check whether the scalar A of size 1-by-1 is also a matrix.

```
TF = ismatrix(A)
```

```
TF = logical
   1
```

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`iscolumn` | `isrow` | `isscalar` | `isvector` | `size`

Introduced in R2010b

ismember

Array elements that are members of set array

Syntax

```
Lia = ismember(A,B)
Lia = ismember(A,B,'rows')
[Lia,Locb] = ismember( ___ )

[Lia,Locb] = ismember( ___ ,'legacy')
```

Description

`Lia = ismember(A,B)` returns an array containing logical `1` (`true`) where the data in `A` is found in `B`. Elsewhere, the array contains logical `0` (`false`).

- If `A` and `B` are tables or timetables, then `ismember` returns a logical value for each row. For timetables, `ismember` takes row times into account to determine equality. The output, `Lia`, is a column vector.

`Lia = ismember(A,B,'rows')` treats each row of `A` and each row of `B` as single entities and returns a column vector containing logical `1` (`true`) where the rows of `A` are also rows of `B`. Elsewhere, the array contains logical `0` (`false`).

The `'rows'` option does not support cell arrays, unless one of the inputs is either a categorical array or a datetime array.

`[Lia,Locb] = ismember( ___ )` also returns an array, `Locb`, using any of the previous syntaxes.

- Generally, `Locb` contains the lowest index in `B` for each value in `A` that is a member of `B`. Values of `0` indicate where `A` is not a member of `B`.
- If the `'rows'` option is specified, then `Locb` contains the lowest index in `B` for each row in `A` that is also a row in `B`. Values of `0` indicate where `A` is not a row of `B`.
- If `A` and `B` are tables or timetables, then `Locb` contains the lowest index in `B` for each row in `A` that is also a row in `B`. Values of `0` indicate where `A` is not a row of `B`.

`[Lia,Locb] = ismember( ___ ,'legacy')` preserves the behavior of the `ismember` function from R2012b and prior releases using any of the input arguments in previous syntaxes.

The `'legacy'` option does not support categorical arrays, datetime arrays, duration arrays, tables, or timetables.

Examples

Values That Are Members of Set

Create two vectors with values in common.

```
A = [5 3 4 2];
B = [2 4 4 4 6 8];
```

Determine which elements of `A` are also in `B`.

```
Lia = ismember(A,B)
```

```
Lia = 1x4 logical array

   0   0   1   1
```

`A(3)` and `A(4)` are found in `B`.

Table Rows Found in Another Table

Create two tables with rows in common.

```
A = table([1:5]',['A';'B';'C';'D';'E'],logical([0;1;0;1;0]))
```

```
A=5×3 table
    Var1    Var2    Var3
    ____    ____    _____

     1       A      false
     2       B      true
```

```
     3       C      false
     4       D      true
     5       E      false
```

```
B = table([1:2:10]',['A';'C';'E';'G';'I'],logical(zeros(5,1)))
```

```
B=5×3 table
    Var1    Var2    Var3
    ____    ____    _____

     1       A      false
     3       C      false
     5       E      false
     7       G      false
     9       I      false
```

Determine which rows of A are also in B.

```
Lia = ismember(A,B)
```

```
Lia = 5x1 logical array

   1
   0
   1
   0
   1
```

A(1,:), A(3,:), and A(5,:) are found in B.

Members of Set and Indices to Values

Create two vectors with values in common.

```
A = [5 3 4 2];
B = [2 4 4 4 6 8];
```

Determine which elements of A are also in B as well as their corresponding locations in B.

```
[Lia,Locb] = ismember(A,B)
```

```
Lia = 1x4 logical array

   0   0   1   1


Locb = 1×4

     0     0     2     1
```

The lowest index to `A(3)` is `B(2)`, and `A(4)` is found in `B(1)`.

Set Members in Presence of Numerical Error

Create a vector `x`. Obtain a second vector `y` by transforming and untransforming `x`. This transformation introduces round-off differences in `y`.

```
x = (1:6)'*pi;
y = 10.^log10(x);
```

Verify that `x` and `y` are not identical by taking the difference.

```
x-y
```

```
ans = 6×1
10^-14 ×

    0.0444
         0
         0
         0
         0
   -0.3553
```

Use `ismember` to find the elements of `x` that are in `y`. The `ismember` function performs exact comparisons and determines that some of the matrix elements in `x` are not members of `y`.

```
lia = ismember(x,y)
```

```
lia = 6x1 logical array
```

```
   0
   1
   1
   1
   1
   0
```

Use `ismembertol` to perform the comparison using a small tolerance. `ismembertol` treats elements that are within tolerance as equal and determines that all of the elements in `x` are members of `y`.

```
LIA = ismembertol(x,y)
```

```
LIA = 6x1 logical array

   1
   1
   1
   1
   1
   1
```

Rows of Another Table and Their Location

Create a table, `A`, of gender, age, and height for five people.

```
A = table(['M';'M';'F';'M';'F'],[27;52;31;46;35],[74;68;64;61;64],...
'VariableNames',{'Gender' 'Age' 'Height'},...
'RowNames',{'Ted' 'Fred' 'Betty' 'Bob' 'Judy'})
```

```
A=5×3 table
             Gender    Age    Height
             ______    ___    ______

    Ted        M       27       74
    Fred       M       52       68
    Betty      F       31       64
    Bob        M       46       61
    Judy       F       35       64
```

Create another table, B, with rows in common with A.

```
B = table(['M';'F';'F';'F'],[47;31;35;23],[68;64;62;58],...
'VariableNames',{'Gender' 'Age' 'Height'},...
'RowNames',{'Joe' 'Meg' 'Beth' 'Amy'})
```

```
B=4×3 table
            Gender    Age    Height
            ______    ___    ______

    Joe       M       47       68
    Meg       F       31       64
    Beth      F       35       62
    Amy       F       23       58
```

Determine which rows of A are also in B, as well as their corresponding locations in B.

```
[Lia,Locb] = ismember(A,B)
```

```
Lia = 5x1 logical array

   0
   0
   1
   0
   0


Locb = 5×1

     0
     0
     2
     0
     0
```

Two rows that have the same values, but different names, are considered equal. The same data for `Betty` is found in `B(2,:)`, which corresponds to `Meg`.

Rows That Belong to a Set

Create two matrices with a row in common.

```
A = [1 3 5 6; 2 4 6 8];
B = [2 4 6 8; 1 3 5 7; 2 4 6 8];
```

Determine which rows of A are also in B as well as their corresponding locations in B.

```
[Lia, Locb] = ismember(A,B, 'rows')
```

```
Lia = 2x1 logical array

   0
   1


Locb = 2×1

     0
     1
```

The lowest index to `A(2,:)` is `B(1,:)`.

Members of Set Containing NaN Values

Create two vectors containing NaN.

```
A = [5 NaN NaN];
B = [5 NaN NaN];
```

Determine which elements of A are also in B, as well as their corresponding locations in B.

```
[Lia,Locb] = ismember(A,B)
```

```
Lia = 1x3 logical array

   1   0   0


Locb = 1×3
```

```
     1     0     0
```

`ismember` treats `NaN` values as distinct.

Cell Array of Character Vectors with Trailing White Space

Create a cell array of character vectors, `A`.

```
A = {'dog','cat','fish','horse'};
```

Create a cell array of character vectors, `B`, where some of the vectors have trailing white space.

```
B = {'dog ','cat','fish ','horse'};
```

Determine which character vectors of `A` are also in `B`.

```
[Lia,Locb] = ismember(A,B)
```

```
Lia = 1x4 logical array

   0   1   0   1


Locb = 1×4

     0     2     0     4
```

`ismember` treats trailing white space in cell arrays of character vectors as distinct characters.

Members of Char and Cell Array of Character Vectors

Create a character vector, `A`, and a cell array of character vectors, `B`.

```
A = ['cat';'dog';'fox';'pig'];
B = {'dog','cat','fish','horse'};
```

Determine which character vectors of A are also in B.

```
[Lia,Locb] = ismember(A,B)
```

```
Lia = 4x1 logical array

   1
   1
   0
   0


Locb = 4×1

     2
     1
     0
     0
```

Preserve Legacy Behavior of ismember

Use the `'legacy'` flag to preserve the behavior of `ismember` from R2012b and prior releases in your code.

Find the members of B with the current behavior.

```
A = [5 3 4 2];
B = [2 4 4 4 6 8];
[Lia1,Locb1] = ismember(A,B)
```

```
Lia1 = 1x4 logical array

   0   0   1   1


Locb1 = 1×4

     0     0     2     1
```

Find the members of B, and preserve the legacy behavior.

```
[Lia2,Locb2] = ismember(A,B,'legacy')
```

```
Lia2 = 1x4 logical array

   0   0   1   1


Locb2 = 1×4

     0     0     4     1
```

Input Arguments

A — Query array
numeric arrays | logical arrays | character arrays | string arrays | categorical arrays | datetime arrays | duration arrays | cell arrays of character vectors | tables | timetables

Query array, specified as a numeric array, logical array, character array, string array, categorical array, datetime array, duration array, cell array of character vectors, table, or timetable. If you specify the `'rows'` option, A and B must have the same number of columns.

A must belong to the same class as B with the following exceptions:

- `logical`, `char`, and all numeric classes can combine with `double` arrays.
- Cell arrays of character vectors can combine with character arrays or string arrays.
- Categorical arrays can combine with character arrays, cell arrays of character vectors, or string arrays.
- Datetime arrays can combine with cell arrays of date character vectors or single date character vectors.

There are additional requirements for A and B based on data type:

- If A and B are both ordinal categorical arrays, they must have the same sets of categories, including their order. If neither A nor B are ordinal, they need not have the same sets of categories, and the comparison is performed using the category names.
- If A is a table or timetable, it must have the same variable names as B (except for order). For tables, row names are ignored, so that two rows that have the same values,

but different names, are considered equal. For timetables, row times are taken into account, so that two rows that have the same values, but different times, are not considered equal.

- If A and B are datetime arrays, they must be consistent with each other in whether they specify a time zone.

For textual inputs, `ismember` generally does not ignore trailing spaces in character vectors, cell arrays of character vectors, and string arrays. However, there are a few cases when `ismember` does ignore trailing spaces:

- If A is a character array and B is a cell array of character vectors, then `ismember` ignores trailing spaces in the character array.
- When the `'rows'` option is specified, `ismember` ignores trailing spaces in character vectors and character arrays.

A also can be an object with the following class methods:

- `sort` (or `sortrows` for the `'rows'` option)
- `eq`
- `ne`

The object class methods must be consistent with each other. These objects include heterogeneous arrays derived from the same root class. For example, A can be an array of handles to graphics objects.

B — Set array

numeric arrays | logical arrays | character arrays | string arrays | categorical arrays | datetime arrays | duration arrays | cell arrays of character vectors | tables | timetables

Set array, specified as a numeric array, logical array, character array, string array, categorical array, datetime array, duration array, cell array of character vectors, table, or timetable. If you specify the `'rows'` option, A and B must have the same number of columns.

B must belong to the same class as A with the following exceptions:

- `logical`, `char`, and all numeric classes can combine with `double` arrays.
- Cell arrays of character vectors can combine with character arrays or string arrays.
- Categorical arrays can combine with character arrays, cell arrays of character vectors, or string arrays.

- Datetime arrays can combine with cell arrays of date character vectors or single date character vectors.

There are additional requirements for A and B based on data type:

- If A and B are both ordinal categorical arrays, they must have the same sets of categories, including their order. If neither A nor B are ordinal, they need not have the same sets of categories, and the comparison is performed using the category names.
- If B is a table or timetable, it must have the same variable names as A (except for order). For tables, row names are ignored, so that two rows that have the same values, but different names, are considered equal. For timetables, row times are taken into account, so that two rows that have the same values, but different times, are not considered equal.
- If A and B are datetime arrays, they must be consistent with each other in whether they specify a time zone.

For textual inputs, `ismember` generally does not ignore trailing spaces in character vectors, cell arrays of character vectors, and string arrays. However, there are a few cases when `ismember` does ignore trailing spaces:

- If A is a character array and B is a cell array of character vectors, then `ismember` ignores trailing spaces in the character array.
- When the `'rows'` option is specified, `ismember` ignores trailing spaces in character vectors and character arrays.

B also can be an object with the following class methods:

- `sort` (or `sortrows` for the `'rows'` option)
- `eq`
- `ne`

The object class methods must be consistent with each other. These objects include heterogeneous arrays derived from the same root class. For example, B can be an array of handles to graphics objects.

Output Arguments

`Lia` — Logical index to `A`
vector | matrix | N-D array

Logical index to `A`, returned as a vector, matrix or N-D array containing logical `1` (`true`) wherever the values (or rows) in `A` are members of `B`. Elsewhere, it contains logical `0` (`false`).

`Lia` is an array of the same size as `A`, unless you specify the `'rows'` flag.

If the `'rows'` flag is specified or if `A` is a table or timetable, `Lia` is a column vector with the same number of rows as `A`.

`Locb` — Locations in `B`
vector | matrix | N-D array

Locations in `B`, returned as a vector, matrix, or N-D array. If the `'legacy'` flag is not specified, `Locb` contains the lowest indices to the values (or rows) in `B` that are found in `A`. Values of `0` indicate where `A` is not a member of `B`.

`Locb` is an array of the same size as `A` unless you specify the `'rows'` flag.

If the `'rows'` flag is specified or if `A` is a table or timetable, `Locb` is a column vector with the same number of rows as `A`.

Tips

- Use `ismembertol` to perform comparisons between floating-point numbers using a tolerance.
- To find the rows from table or timetable `A` that are found in `B` with respect to a subset of variables, you can use column subscripting. For example, you can use `ismember(A(:,`*`vars`*`),B(:,`*`vars`*`))`, where *`vars`* is a positive integer, a vector of positive integers, a variable name, a cell array of variable names, or a logical vector. Alternatively, you can use `vartype` to create a subscript that selects variables of a specified type.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

Only one of A or B can be a tall array.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support cell arrays for the first or second arguments.
- Complex inputs must be `single` or `double`.
- See “Code Generation for Complex Data with Zero-Valued Imaginary Parts” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `'legacy'` flag is not supported.
- 64-bit integers are not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`intersect` | `ismembertol` | `issorted` | `setdiff` | `setxor` | `sort` | `union` | `unique`

Introduced before R2006a

ismembertol

Members of set within tolerance

Syntax

```
LIA = ismembertol(A,B,tol)
LIA = ismembertol(A,B)
[LIA,LocB] = ismembertol( ___ )
[ ___ ] = ismembertol( ___ ,Name,Value)
```

Description

`LIA = ismembertol(A,B,tol)` returns an array containing logical `1` (`true`) where the elements of `A` are within tolerance of the elements in `B`. Otherwise, the array contains logical `0` (`false`). Two values, `u` and `v`, are within tolerance if

```
abs(u-v) <= tol*max(abs([A(:);B(:)]))
```

That is, `ismembertol` scales the `tol` input based on the magnitude of the data.

`ismembertol` is similar to `ismember`. Whereas `ismember` performs exact comparisons, `ismembertol` performs comparisons using a tolerance.

`LIA = ismembertol(A,B)` uses a default tolerance of `1e-6` for single-precision inputs and `1e-12` for double-precision inputs.

`[LIA,LocB] = ismembertol( ___ )` also returns an array, `LocB`, that contains the index location in `B` for each element in `A` that is a member of `B`. You can use any of the input arguments in previous syntaxes.

`[ ___ ] = ismembertol( ___ ,Name,Value)` uses additional options specified by one or more Name-Value pair arguments using any of the input or output argument combinations in previous syntaxes. For example, `ismembertol(A,B,'ByRows',true)` compares the rows of `A` and `B` and returns a logical column vector.

Examples

Set Members in Presence of Numerical Error

Create a vector `x`. Obtain a second vector `y` by transforming and untransforming `x`. This transformation introduces round-off differences in `y`.

```
x = (1:6)'*pi;
y = 10.^log10(x);
```

Verify that `x` and `y` are not identical by taking the difference.

```
x-y
```

```
ans = 6×1
10^-14 ×

    0.0444
         0
         0
         0
         0
   -0.3553
```

Use `ismember` to find the elements of `x` that are in `y`. The `ismember` function performs exact comparisons and determines that some of the matrix elements in `x` are not members of `y`.

```
lia = ismember(x,y)
```

```
lia = 6x1 logical array

   0
   1
   1
   1
   1
   0
```

Use `ismembertol` to perform the comparison using a small tolerance. `ismembertol` treats elements that are within tolerance as equal and determines that all of the elements in `x` are members of `y`.

```
LIA = ismembertol(x,y)
```

```
LIA = 6x1 logical array

   1
   1
   1
   1
   1
   1
```

Determine Set Members by Rows

By default, `ismembertol` looks for *elements* that are within tolerance, but it also can find *rows* of a matrix that are within tolerance.

Create a numeric matrix, `A`. Obtain a second matrix, `B`, by transforming and untransforming `A`. This transformation introduces round-off differences to `B`.

```
A = [0.05 0.11 0.18; 0.18 0.21 0.29; 0.34 0.36 0.41; ...
    0.46 0.52 0.76; 0.82 0.91 1.00];
B = log10(10.^A);
```

Use `ismember` to find the rows of `A` that are in `B`. `ismember` performs exact comparisons and thus determines that most of the rows in `A` are not members of `B`, even though some of the rows differ by only a small amount.

```
lia = ismember(A,B,'rows')
```

```
lia = 5x1 logical array

   0
   0
   0
   0
   1
```

Use `ismembertol` to perform the row comparison using a small tolerance. `ismembertol` treats rows that are within tolerance as equal and thus determines that all of the rows in A are members of B.

```
LIA = ismembertol(A,B,'ByRows',true)
```

```
LIA = 5x1 logical array

   1
   1
   1
   1
   1
```

Average Similar Values in Vectors

Create two vectors of random numbers and determine which values in A are also members of B, using a tolerance. Specify `OutputAllIndices` as `true` to return all of the indices for the elements in B that are within tolerance of the corresponding elements in A.

```
rng(5)
A = rand(1,15);
B = rand(1,5);
[LIA,LocAllB] = ismembertol(A,B,0.2,'OutputAllIndices',true)
```

```
LIA = 1x15 logical array

   1   0   1   0   1   1   1   1   1   1   0   1   1   1   0


LocAllB=1×15 cell
  Columns 1 through 5

    {2x1 double}    {[0]}    {2x1 double}    {[0]}    {3x1 double}

  Columns 6 through 10

    {2x1 double}    {[4]}    {3x1 double}    {3x1 double}    {2x1 double}

  Columns 11 through 15
```

```
    {[0]}    {2x1 double}    {4x1 double}    {2x1 double}    {[0]}
```

Find the average value of the elements in B that are within tolerance of the value A(13). The cell LocAllB{13} contains all the indices for elements in B that are within tolerance of A(13).

```
A(13)
```

```
ans = 0.4413
```

```
allB = B(LocAllB{13})
```

```
allB = 1×4

    0.2741    0.4142    0.2961    0.5798
```

```
aveB = mean(allB)
```

```
aveB = 0.3911
```

Specify Absolute Tolerance

By default, ismembertol uses a tolerance test of the form abs(u-v) <= tol*DS, where DS automatically scales based on the magnitude of the input data. You can specify a different DS value to use with the DataScale option. However, absolute tolerances (where DS is a scalar) do not scale based on the magnitude of the input data.

First, compare two small values that are a distance eps apart. Specify tol and DS to make the within tolerance equation abs(u-v) <= 10^-6.

```
x = 0.1;
ismembertol(x, exp(log(x)), 10^-6, 'DataScale', 1)
```

```
ans = logical
   1
```

Next, increase the magnitude of the values. The round-off error in the calculation exp(log(x)) is proportional to the magnitude of the values, specifically to eps(x). Even though the two large values are a distance eps from one another, eps(x) is now much larger. Therefore, 10^-6 is no longer a suitable tolerance.

```
x = 10^10;
ismembertol(x, exp(log(x)), 10^-6, 'DataScale', 1)
```

```
ans = logical
   0
```

Correct this issue by using the default (scaled) value of DS.

```
Y = [0.1 10^10];
ismembertol(Y, exp(log(Y)))
```

```
ans = 1x2 logical array

   1   1
```

Specify DataScale by Column

Create a set of random 2-D points, and then use `ismembertol` to group the points into vertical bands that have a similar (within-tolerance) x-coordinate to a small set of query points, B. Use these options with `ismembertol`:

- Specify ByRows as `true`, since the point coordinates are in the rows of A and B.
- Specify `OutputAllIndices` as `true` to return the indices for all points in A that have an x-coordinate within tolerance of the query points in B.
- Specify DataScale as `[1 Inf]` to use an absolute tolerance for the x-coordinate, while ignoring the y-coordinate.

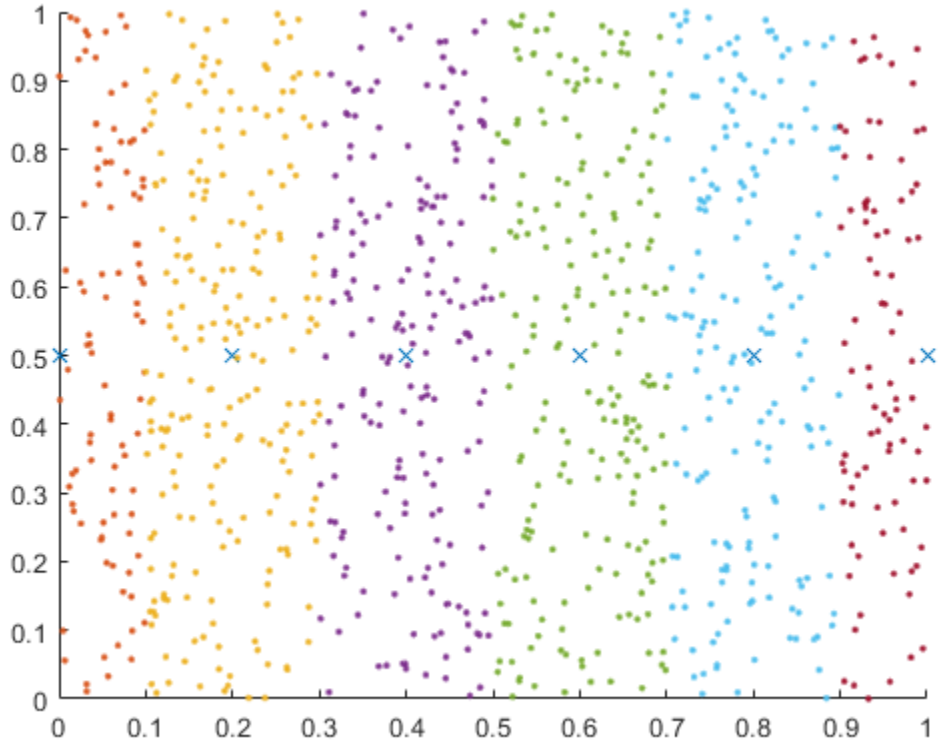
```
A = rand(1000,2);
B = [(0:.2:1)',0.5*ones(6,1)];
[LIA,LocAllB] = ismembertol(B, A, 0.1, 'ByRows', true, ...
    'OutputAllIndices', true, 'DataScale', [1,Inf])
```

```
LIA = 6x1 logical array

   1
   1
   1
   1
   1
```

```
   1


LocAllB=6×1 cell
    { 94x1 double}
    {223x1 double}
    {195x1 double}
    {212x1 double}
    {187x1 double}
    { 89x1 double}
```

Plot the points in A that are within tolerance of each query point in B.

```
hold on
plot(B(:,1),B(:,2),'x')
for k = 1:length(LocAllB)
    plot(A(LocAllB{k},1), A(LocAllB{k},2),'.')
end
```



Input Arguments

A — Query array
scalar | vector | matrix | multidimensional array

Query array, specified as a scalar, vector, matrix, or multidimensional array. Inputs A and B must be full.

If you specify the ByRows option, then A and B must have the same number of columns.

Data Types: `single` | `double`

B — Query array
scalar | vector | matrix | multidimensional array

Query array, specified as a scalar, vector, matrix, or multidimensional array. Inputs A and B must be full.

If you specify the ByRows option, then A and B must have the same number of columns.

Data Types: single | double

tol — Comparison tolerance
positive real scalar

Comparison tolerance, specified as a positive real scalar. ismembertol scales the tol input using the maximum absolute values in the input arrays A and B. Then ismembertol uses the resulting scaled comparison tolerance to determine which elements in A are also a member of B. If two elements are within tolerance of each other, then ismembertol considers them to be equal.

Two values, u and v, are within tolerance if abs(u-v) <= tol*max(abs([A(:);B(:)])).

To specify an absolute tolerance, specify both tol and the 'DataScale' Name-Value pair.

Example: tol = 0.05

Example: tol = 1e-8

Example: tol = eps

Data Types: single | double

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: LIA = ismembertol(A,B,'ByRows',true)

OutputAllIndices — Output index type
false (default) | true | 0 | 1

Output index type, specified as the comma-separated pair consisting of `'OutputAllIndices'` and either `false` (default), `true`, `0`, or `1`. `ismembertol` interprets numeric `0` as `false` and numeric `1` as `true`.

When `OutputAllIndices` is `true`, the `ismembertol` function returns the second output, `LocB`, as a cell array. The cell array contains the indices for *all* elements in `B` that are within tolerance of the corresponding value in `A`. That is, each cell in `LocB` corresponds to a value in `A`, and the values in each cell correspond to locations in `B`.

Example: `[LIA,LocAllB] = ismembertol(A,B,tol,'OutputAllIndices',true)`

`ByRows` — Row comparison toggle
`false` (default) | `true` | `0` | `1`

Row comparison toggle, specified as the comma-separated pair consisting of `'ByRows'` and either `false` (default), `true`, `0`, or `1`. `ismembertol` interprets numeric `0` as `false` and numeric `1` as `true`. Use this option to find rows in `A` and `B` that are within tolerance.

When `ByRows` is `true`:

- `ismembertol` compares the rows of `A` and `B` by considering each column separately. Thus, `A` and `B` must be 2-D arrays with the same number of columns.
- If the corresponding row in `A` is within tolerance of a row in `B`, then `LIA` contains logical `1` (`true`). Otherwise, it contains logical `0` (`false`).

Two rows, `u` and `v`, are within tolerance if `all(abs(u-v) <= tol*max(abs([A;B])))`.

Example: `LIA = ismembertol(A,B,tol,'ByRows',true)`

`DataScale` — Scale of data
scalar | vector

Scale of data, specified as the comma-separated pair consisting of `'DataScale'` and either a scalar or vector. Specify `DataScale` as a numeric scalar, `DS`, to change the tolerance test to be, `abs(u-v) <= tol*DS`.

When used together with the `ByRows` option, the `DataScale` value also can be a vector. In this case, each element of the vector specifies `DS` for a corresponding column in `A`. If a value in the `DataScale` vector is `Inf`, then `ismembertol` ignores the corresponding column in `A`.

Example: `LIA = ismembertol(A,B,'DataScale',1)`

Example: `[LIA,LocB] = ismembertol(A,B,'ByRows',true,'DataScale', [eps(1) eps(10) eps(100)])`

Data Types: `single` | `double`

Output Arguments

`LIA` — Logical index to `A`
vector | matrix

Logical index to `A`, returned as a vector or matrix containing logical `1` (`true`) wherever the elements (or rows) in `A` are members of `B` (within tolerance). Elsewhere, `LIA` contains logical `0` (`false`).

`LIA` is the same size as `A`, unless you specify the `ByRows` option. In that case, `LIA` is a column vector with the same number of rows as `A`.

`LocB` — Locations in `B`
vector | matrix | cell array

Locations in `B`, returned as a vector, matrix, or cell array. `LocB` contains the indices to the elements (or rows) in `B` that are found in `A` (within tolerance). `LocB` contains `0` wherever an element in `A` is not a member of `B`.

If `OutputAllIndices` is `true`, then `ismembertol` returns `LocB` as a cell array. The cell array contains the indices for *all* elements in `B` that are within tolerance of the corresponding value in `A`. That is, each cell in `LocB` corresponds to a value in `A`, and the values in each cell correspond to locations in `B`.

`LocB` is the same size as `A`, unless you specify the `ByRows` option. In that case, `LocB` is a column vector with the same number of rows as `A`.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `'ByRows'` and `'OutputAllIndices'` arguments are not supported.
- 64-bit integers are not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`eps` | `ismember` | `unique` | `uniquetol`

Topics

“Group Scattered Data Using a Tolerance”

Introduced in R2015a

ismethod

Determine if method of object

Syntax

```
tf = ismethod(obj,methodName)
```

Description

`tf = ismethod(obj,methodName)` returns logical `1` (`true`) if the specified `methodName` is a nonhidden, public method of object `obj`. Otherwise, `ismethod` returns logical `0` (`false`). The `methodName` argument is the name of the method specified as a character vector or a string scalar.

`ismethod` does not accept class names specified as character vectors or strings as the first argument.

Use the `methods` function to list the methods of abstract classes.

Examples

Determine if objects support equality testing:

```
if ismethod(obj1,'eq') && ismethod(obj2,'eq')
   tf = obj1 == obj2;
end
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`class` | `isobject` | `isprop` | `methods`

Topics

"Methods"
"Method Attributes"

Introduced before R2006a

cool

Cool colormap array

Syntax

```
c = cool
c = cool(m)
```

Description

`c = cool` returns the cool colormap as a three-column array with the same number of rows as the colormap for the current figure. If no figure exists, then the number of rows is equal to the default length of 256. Each row in the array contains the red, green, and blue intensities for a specific color. The intensities are in the range [0,1], and the color scheme looks like this image.

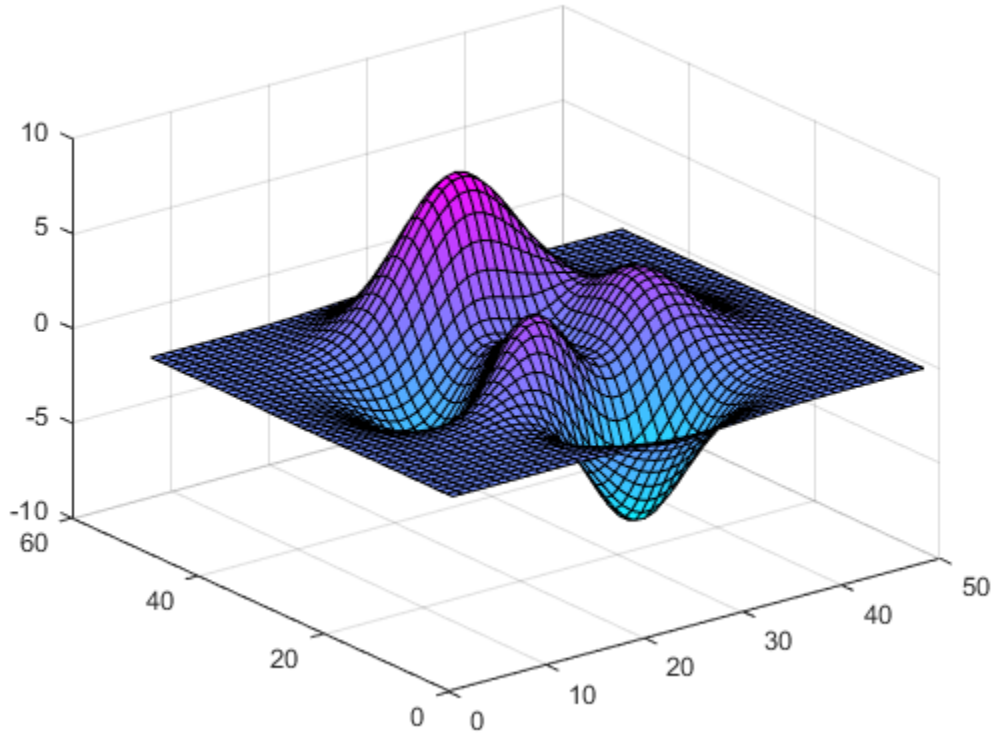
`c = cool(m)` returns the colormap with `m` colors.

Examples

Reverse the Colormap

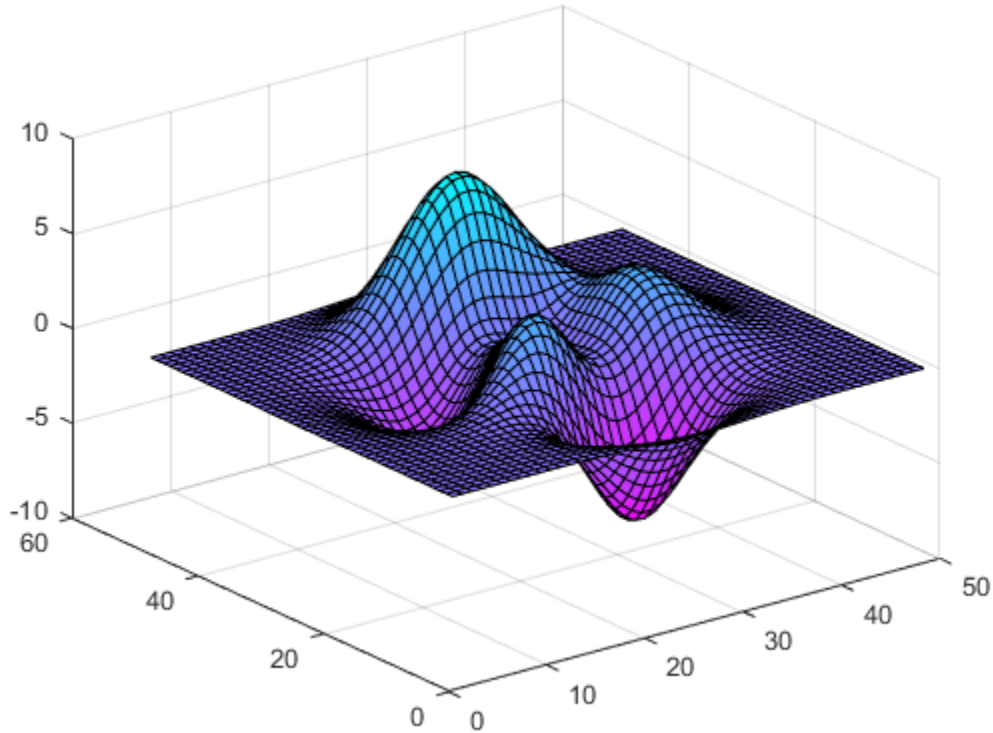
Plot a surface and assign the cool colormap.

```
surf(peaks);
colormap('cool');
```



Get the cool colormap array and reverse the order. Then apply the modified colormap to the surface.

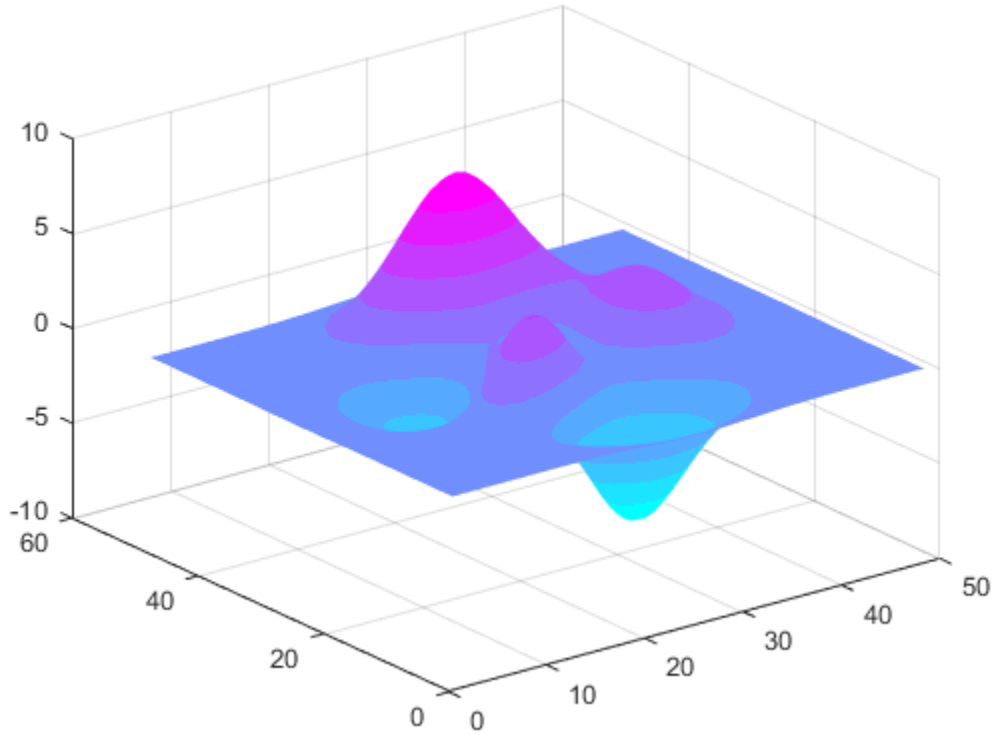
```
c = cool;
c = flipud(c);
colormap(c);
```



Downsample the Cool Colormap

Get a downsampled version of the cool colormap containing only ten colors. Then display the contours of the peaks function by applying the colormap and interpolated shading.

```
c = cool(10);
surf(peaks);
colormap(c);
shading interp;
```



Input Arguments

m — Number of colors

256 (default) | scalar integer value

Number of colors, specified as a scalar integer value. The default value of `m` is equal to the length of the colormap for the current figure. If no figure exists, the default value is 256.

Data Types: `single` | `double`

Compatibility Considerations

Colormaps Have 256 Colors by Default

Behavior changed in R2019b

Starting in R2019b, colormaps have 256 colors by default.

In R2019a and previous releases, the default size is 64. If you have code that depends on a colormap having 64 colors, specify the number of colors when you set the colormap for the figure, axes, or chart. For example, `colormap(cool(64))` sets the figure's colormap to the 64-color `cool` colormap.

Alternatively, you can change the default colormap for all figures within your MATLAB session:

```
set(groot,'defaultFigureColormap',cool(64))
```

See Also

`colormap`

Topics

"Change Color Scheme Using a Colormap"

Introduced before R2006a

ismissing

Find missing values

Syntax

```
TF = ismissing(A)
TF = ismissing(A,indicator)
```

Description

`TF = ismissing(A)` returns a logical array that indicates which elements of an array or table contain missing values. The size of `TF` is the same as the size of `A`.

Standard missing values depend on the data type:

- `NaN` for `double`, `single`, `duration`, and `calendarDuration`
- `NaT` for `datetime`
- `<missing>` for `string`
- `<undefined>` for `categorical`
- `' '` for `char`
- `{''}` for `cell` of character vectors

`TF = ismissing(A,indicator)` treats the values in `indicator` as missing value indicators, ignoring all default indicators listed in the previous syntax. `indicator` can be a single indicator or multiple indicators. For example, if `A` is an array of type `double`, then `ismissing(A,[0,-99])` treats 0 and -99 as missing `double` values instead of `NaN`.

Examples

NaN Values in Vector

Create a row vector `A` that contains `NaN` values, and identify their location in `A`.

```
A = [3 NaN 5 6 7 NaN NaN 9];
TF = ismissing(A)
```

```
TF = 1x8 logical array

   0   1   0   0   0   1   1   0
```

Missing Values in Table with Various Data Types

Create a table with variables of different data types and find the elements with missing values.

```
dblVar = [NaN;3;5;7;9;11;13];
singleVar = single([1;NaN;5;7;9;11;13]);
cellstrVar = {'one';'three';'';'seven';'nine';'eleven';'thirteen'};
charVar = ['A';'C';'E';' ';'I';'J';'L'];
categoryVar = categorical({'red';'yellow';'blue';'violet';'';'ultraviolet';'orange'});
dateVar = [datetime(2015,1:2:10,15) NaT datetime(2015,11,15)]';
stringVar = ["a";"b";"c";"d";"e";"f";missing];

A = table(dblVar,singleVar,cellstrVar,charVar,categoryVar,dateVar,stringVar)
```

```
A=7×7 table
    dblVar    singleVar     cellstrVar     charVar    categoryVar      dateVar      str
    ______    _________    ___________    _______    ___________    ___________    ___

     NaN            1       {'one'     }       A       red            15-Jan-2015    "a"
       3          NaN       {'three'   }       C       yellow         15-Mar-2015    "b"
       5            5       {0x0 char  }       E       blue           15-May-2015    "c"
       7            7       {'seven'   }               violet         15-Jul-2015    "d"
       9            9       {'nine'    }       I       <undefined>    15-Sep-2015    "e"
      11           11       {'eleven'  }       J       ultraviolet            NaT    "f"
      13           13       {'thirteen'}       L       orange         15-Nov-2015    <m
```

`ismissing` returns 1 where the corresponding element in `A` has a missing value.

```
TF = ismissing(A)
```

```
TF = 7x7 logical array

   1   0   0   0   0   0   0
```

```
   0   1   0   0   0   0   0
   0   0   1   0   0   0   0
   0   0   0   1   0   0   0
   0   0   0   0   1   0   0
   0   0   0   0   0   1   0
   0   0   0   0   0   0   1
```

The size of TF is the same as the size of A.

Specify Indicators for Missing Values in Table

Create a table where 'NA', '', -99, NaN, and Inf represent missing values. Then, find the elements with missing values.

```
dblVar = [NaN;3;Inf;7;9];
int8Var = int8([1;3;5;7;-99]);
cellstrVar = {'one';'three';'';'NA';'nine'};
charVar = ['A';'C';'E';' ';'I'];

A = table(dblVar,int8Var,cellstrVar,charVar)
```

```
A=5×4 table
    dblVar    int8Var    cellstrVar    charVar
    ______    _______    __________    _______

     NaN          1      {'one'   }       A
       3          3      {'three' }       C
     Inf          5      {0x0 char}       E
       7          7      {'NA'    }
       9        -99      {'nine'  }       I
```

ismissing returns 1 where the corresponding element in A has a missing value.

```
id = {'NA' '' -99 NaN Inf};
TF = ismissing(A,id)
```

```
TF = 5x4 logical array

   1   0   0   0
   0   0   0   0
```

```
1   0   1   0
0   0   1   1
0   1   0   0
```

`ismissing` ignores trailing white space in character arrays. Therefore, since the empty character vector, `''`, is specified as a missing value indicator, `ismissing` identifies the empty character vector in `A.cellstrVar` and also the blank space in `A.charVar` as missing values.

Input Arguments

`A` — Input data
vector | matrix | multidimensional array | table | timetable

Input data, specified as a vector, matrix, multidimensional array, table, or timetable.

If `A` is a timetable, then `ismissing` operates on the table data only and ignores `NaT` or `NaN` values in the vector of row times.

When the input argument is a cell array, it must be a cell array of character vectors. If the input is a table or timetable with a variable of type `cell`, then `ismissing` only detects missing elements when the variable is a cell array of character vectors.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `cell` | `table` | `timetable` | `categorical` | `datetime` | `duration` | `calendarDuration`

`indicator` — Missing value indicator
scalar | vector | cell array

Missing value indicators, specified as a scalar, vector, or cell array. If `A` is an array, then `indicator` must be a vector. If `A` is a table or timetable, then `indicator` can also be a cell array with entries of multiple data types.

The entries of `indicator` indicate the values that `ismissing` treats as missing. Specifying `indicator` overrides all default standard missing indicators. If you want to add indicators while maintaining the list of standard indicators, then you must include all default indicators as elements of `indicator`. For example, if `A` is a table with `categorical` and numeric values, use `ismissing(A,{-99,'<undefined>'})` to

indicate `-99` as a missing numeric value, but preserve `<undefined>` as a missing `categorical` value.

You can also use the `missing` value as an indicator for any missing data represented as `NaN`, `NaT`, `missing`, or `<undefined>`. If your input is a table, then `missing` is also an indicator for missing character vectors (`' '`) and missing cell arrays of character vectors (`{''}`).

Indicator data types match data types in the entries of `A`. The following are additional data type matches between the indicator and elements of `A`:

- `double` indicators match `double`, `single`, integer, and `logical` entries of `A`.
- `string` and `char` indicators, and indicators that are cell arrays of character vectors, match `string` entries of `A`.
- `string` and `char` indicators match `categorical` entries of `A`.

Example: `TF = ismissing(A,0)` recognizes only `0` as a missing value.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `cell` | `datetime` | `duration`

Tips

- Since integer variables cannot store `NaN`, use a special integer value (otherwise unused) to indicate missing integer data, such as `-99`.
- For more information on finding missing strings, see “Test for Empty Strings and Missing Values”.

Algorithms

`ismissing` handles leading and trailing white space differently for indicators that are cell arrays of character vectors, character arrays, or categorical arrays.

- For cell arrays of character vectors, `ismissing` does not ignore indicator white space. All character vectors must match exactly.
- For character arrays in table variables, `ismissing` ignores trailing white space in the indicator.

- For categorical arrays, `ismissing` ignores leading and trailing white space in the indicator.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Tables are not supported.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Clean Missing Data | `all` | `any` | `fillmissing` | `isempty` | `isnan` | `isoutlier` | `isundefined` | `missing` | `rmmissing` | `standardizeMissing`

Topics

Clean Messy Data and Locate Extrema Using Live Editor Tasks
“Missing Data in MATLAB”

Introduced in R2013b

isnan

Determine which array elements are NaN

Syntax

```
TF = isnan(A)
```

Description

`TF = isnan(A)` returns a logical array containing `1` (`true`) where the elements of `A` are `NaN`, and `0` (`false`) where they are not. If `A` contains complex numbers, `isnan(A)` contains `1` for elements with either real or imaginary part is `NaN`, and `0` for elements where both real and imaginary parts are not `NaN`.

Examples

Determine NaN Elements in Real Numbers

Create a row vector and determine which elements are `NaN`.

```
A = 0./[-2 -1 0 1 2]
```

```
A = 1×5

     0     0   NaN     0     0
```

```
TF = isnan(A)
```

```
TF = 1x5 logical array

   0   0   1   0   0
```

Determine NaN Elements in Complex Numbers

Create an array of complex numbers. Determine whether the complex numbers contain NaN.

```
A = [2 + 1i, 1/0 + 3i, 1/2 - 1i*NaN]
```

```
A = 1×3 complex

   2.0000 + 1.0000i      Inf + 3.0000i      NaN +    NaNi
```

```
TF = isnan(A)
```

```
TF = 1x3 logical array

   0   0   1
```

Replace NaN Elements

Create an array and find the elements with NaN values.

```
A = [1,3,5,7,NaN,10,NaN,4,6,8]
```

```
A = 1×10

     1     3     5     7   NaN    10   NaN     4     6     8
```

```
TF = isnan(A)
```

```
TF = 1x10 logical array

   0   0   0   0   1   0   1   0   0   0
```

Index into A with TF to access the elements of A that are NaN. Replace the NaN values with 0.

```
A(TF) = 0
```

```
A = 1×10
```

```
     1     3     5     7     0    10     0     4     6     8
```

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Tips

- If `x` is a real scalar, exactly one of `isfinite(x)`, `isinf(x)`, and `isnan(x)` returns logical `1` (`true`).
- For a complex scalar `z`, `isinf(z)` and `isnan(z)` can both return logical 1. For example, `isinf(complex(Inf,NaN))` and `isnan(complex(Inf,NaN))` both return logical 1.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`NaN` | `is*` | `isfinite` | `isinf` | `ismissing`

Topics

“Missing Data in MATLAB”

Introduced before R2006a

isnat

Determine NaT (Not-a-Time) elements

Syntax

```
tf = isnat(A)
```

Description

`tf = isnat(A)` returns an array the same size as the datetime array, `A`, containing logical 1 (`true`) where the elements of `A` are Not-a-Time (`NaT`s) and logical 0 (`false`) where they are not. `NaT` represents a datetime that is undefined.

Examples

Determine If Array Elements are NaT

Create a datetime array from numeric values containing `NaN`.

```
d = datetime(2014,[1 2 NaN 4],1)
```

```
d = 1x4 datetime array
   01-Jan-2014   01-Feb-2014   NaT           01-Apr-2014
```

Determine if any elements of `d` are `NaT` (Not-a-Time).

```
isnat(d)
```

```
ans = 1x4 logical array

   0   0   1   0
```

Input Arguments

A — Input array
datetime array

Input array, specified as a datetime array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`isfinite` | `isinf`

Introduced in R2014b

isnumeric

Determine whether input is numeric array

Syntax

```
TF = isnumeric(A)
```

Description

`TF = isnumeric(A)` returns logical `1` (`true`) if `A` is an array of numeric data type. Otherwise, it returns logical `0` (`false`).

Numeric types in MATLAB include: `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, `single`, and `double`. For more information, see “Integer Classes” and “Floating-Point Numbers”.

Examples

Determine Whether Number Is Numeric Type

Determine if a real number is a numeric type.

```
TF = isnumeric(2)
```

```
TF = logical
   1
```

MATLAB® stores numeric data as a double-precision format by default, which is a numeric type.

Now create a signed 16-bit integer type using `int16`. Check if it is a numeric type.

```
TF = isnumeric(int16(-256))
```

```
TF = logical
   1
```

Determine Whether Infinity and NaN Are Numeric Type

Determine if an array containing infinity or not-a-number values is a numeric type.

Create several calculations that return `Inf` and assign the results to an array. Check if the array is a numeric type.

```
A = [1/0 log(0) 1e1000]
```

```
A = 1×3

   Inf  -Inf   Inf
```

```
TF = isnumeric(A)
```

```
TF = logical
   1
```

MATLAB represents infinity by the special value `Inf` as a `double` type.

Create several calculations that return `NaN` and assign the results to an array. Check if the array is a numeric type.

```
A = [0/0 -Inf/Inf]
```

```
A = 1×2

   NaN   NaN
```

```
TF = isnumeric(A)
```

```
TF = logical
   1
```

MATLAB represents not-a-number by the special value `NaN`, as a `double` type.

Determine Whether Arrays Are Numeric Type

Determine if an array containing floating-point numbers is a numeric type.

```
A = [-3.5e2 2.5; single(3) pi]
```

```
A = 2x2 single matrix

 -350.0000    2.5000
    3.0000    3.1416
```

```
TF = isnumeric(A)
```

```
TF = logical
   1
```

Now create a cell array that contains the array `A` and other numbers. Use `class` to identify the type of the cell array. Check if it is a numeric type.

```
B = {A -4; 2 1}
```

```
B=2×2 cell
    {2x2 single}    {[-4]}
    {[       2]}    {[ 1]}
```

```
type = class(B)
```

```
type =
'cell'
```

```
TF = isnumeric(B)
```

```
TF = logical
   0
```

The cell array is not a numeric type since it is a cell type.

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. MATLAB has the following numeric types.

Integer Type	Description
`single`	single-precision floating-point
`double`	double-precision floating-point
`int8`	8-bit signed integer
`int16`	16-bit signed integer
`int64`	64-bit signed integer
`int32`	32-bit signed integer
`uint8`	8-bit unsigned integer
`uint16`	16-bit unsigned integer
`uint32`	32-bit unsigned integer
`uint64`	64-bit unsigned integer

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`class` | `is*` | `isa` | `isfloat` | `isinteger`

Topics

“Integers”
“Floating-Point Numbers”

Introduced before R2006a

isobject

Determine if input is MATLAB object

Syntax

```
tf = isobject(A)
```

Description

`tf = isobject(A)` returns `true` if `A` is an object of a MATLAB class. Otherwise, it returns `false`.

Instances of MATLAB numeric, `logical`, `char`, `cell`, `struct`, and function handle classes return `false`. Use `isa` to test for any of these types.

Examples

Define the following MATLAB class:

```
classdef button < handle
   properties
      UiHandle
   end
   methods
      function obj = button(pos)
         obj.UiHandle = uicontrol('Position',pos,...
            'Style','pushbutton');
      end
   end
end
```

Test for MATLAB objects.

```
h = button([20 20 60 60]);
isobject(h)
```

```
ans =

  logical

   1
```

```
isobject(h.UiHandle)
```

```
ans =

  logical

   1
```

Create an object that is a MATLAB numeric type:

```
a = pi;
isobject(a)
```

```
  logical
     0
```

```
isa(a,'double')
```

```
ans =

  logical

   1
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`class` | `is*` | `isa`

Topics

"Class Components"

Introduced before R2006a

isocaps

Compute isosurface end-cap geometry

Syntax

```
fvc = isocaps(X,Y,Z,V,isovalue)
fvc = isocaps(V,isovalue)
fvc = isocaps(...,'enclose')
fvc = isocaps(...,'whichplane')
[f,v,c] = isocaps(...)
isocaps(...)
```

Description

`fvc = isocaps(X,Y,Z,V,isovalue)` computes isosurface end-cap geometry for the volume data `V` at isosurface value `isovalue`. The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`.

The struct `fvc` contains the face, vertex, and color data for the end-caps and can be passed directly to the `patch` command.

`fvc = isocaps(V,isovalue)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)` where `[m,n,p] = size(V)`.

`fvc = isocaps(...,'enclose')` specifies whether the end-caps enclose data values above or below the value specified in `isovalue`. The `'enclose'` option can be either `'above'` (default) or `'below'`.

`fvc = isocaps(...,'whichplane')` specifies on which planes to draw the end-caps. Possible values for `'whichplane'` are `'all'` (default), `'xmin'`, `'xmax'`, `'ymin'`, `'ymax'`, `'zmin'`, or `'zmax'`.

`[f,v,c] = isocaps(...)` returns the face, vertex, and color data for the end-caps in three arrays instead of the struct `fvc`.

`isocaps(...)` without output arguments draws a patch with the computed faces, vertices, and colors.

Examples

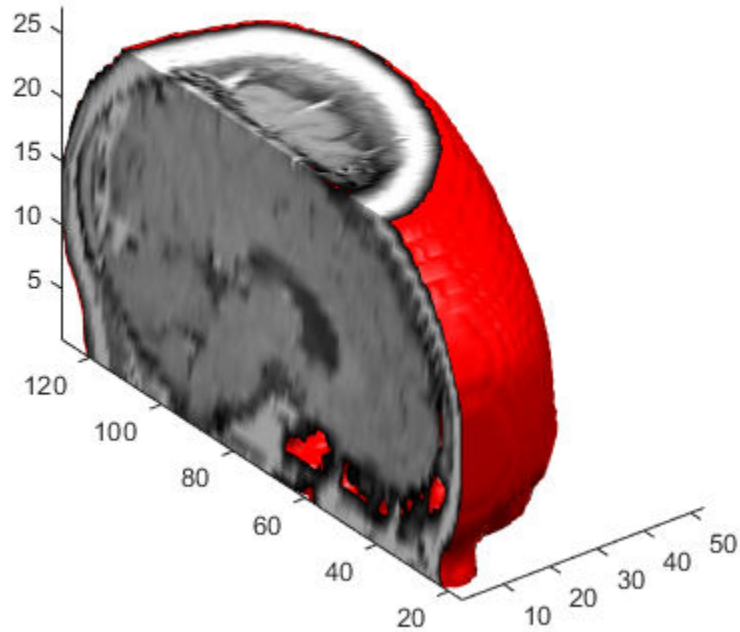
Compute Isosurface End-Cap Geometry

This example uses a data set that is a collection of MRI slices of a human skull. It illustrates the use of isocaps to draw the end-caps on this cutaway volume.

The red isosurface shows the outline of the volume (skull) and the end-caps show what is inside of the volume.

The patch created from the end-cap data (`p2`) uses interpolated face coloring, which means the gray colormap and the light sources determine how it is colored. The isosurface patch (`p1`) used a flat red face color, which is affected by the lights, but does not use the colormap.

```
load mri
D = squeeze(D);
D(:,1:60,:) = [];
p1 = patch(isosurface(D, 5),'FaceColor','red',...
    'EdgeColor','none');
p2 = patch(isocaps(D, 5),'FaceColor','interp',...
    'EdgeColor','none');
view(3)
axis tight
daspect([1,1,.4])
colormap(gray(100))
camlight left
camlight
lighting gouraud
isonormals(D,p1)
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`isonormals` | `isosurface` | `reducepatch` | `reducevolume` | `smooth3` | `subvolume`

Topics

“Isocaps Add Context to Visualizations”

Introduced before R2006a

isocolors

Calculate isosurface and patch colors

Syntax

```
nc = isocolors(X,Y,Z,C,vertices)
nc = isocolors(X,Y,Z,R,G,B,vertices)
nc = isocolors(C,vertices)
nc = isocolors(R,G,B,vertices)
nc = isocolors(...,PatchHandle)
isocolors(...,PatchHandle)
```

Description

`nc = isocolors(X,Y,Z,C,vertices)` computes the colors of isosurface (patch object) vertices (`vertices`) using color values `C`. Arrays `X`, `Y`, `Z` define the coordinates for the color data in `C` and must be monotonic vectors that represent a Cartesian, axis-aligned grid (as if produced by `meshgrid`). The colors are returned in `nc`. `C` must be 3-D (index colors).

`nc = isocolors(X,Y,Z,R,G,B,vertices)` uses `R`, `G`, `B` as the red, green, and blue color arrays (true colors).

`nc = isocolors(C,vertices)`, and `nc = isocolors(R,G,B,vertices)` assume `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p] = size(C)`.

`nc = isocolors(...,PatchHandle)` uses the vertices from the patch identified by `PatchHandle`.

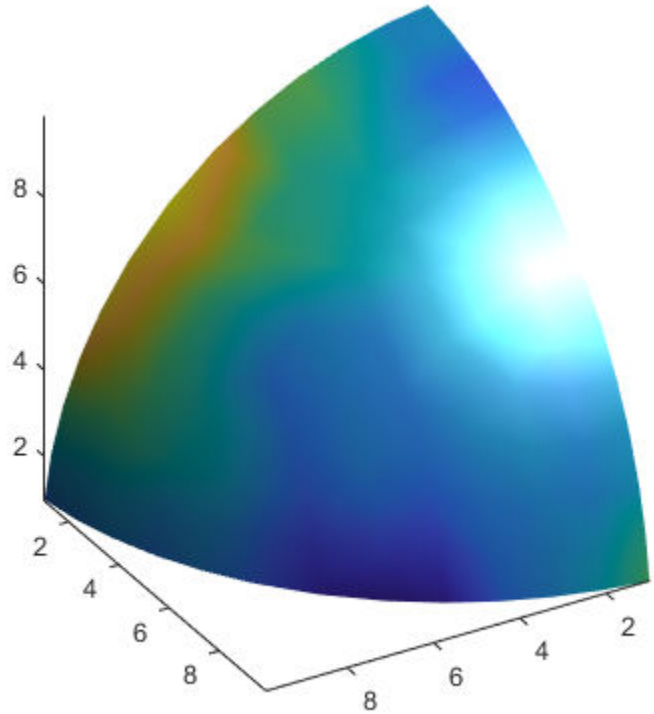
`isocolors(...,PatchHandle)` sets the `FaceVertexCData` property of the patch specified by `PatchHandle` to the computed colors.

Examples

Indexed Color Data

This example displays an isosurface and colors it with random data using indexed color.

```
[x,y,z] = meshgrid(1:20,1:20,1:20);
data = sqrt(x.^2 + y.^2 + z.^2);
cdata = smooth3(rand(size(data)),'box',7);
p = patch(isosurface(x,y,z,data,10));
isonormals(x,y,z,data,p)
isocolors(x,y,z,cdata,p)
p.FaceColor = 'interp';
p.EdgeColor = 'none';
view(150,30)
daspect([1 1 1])
axis tight
camlight
lighting gouraud
```

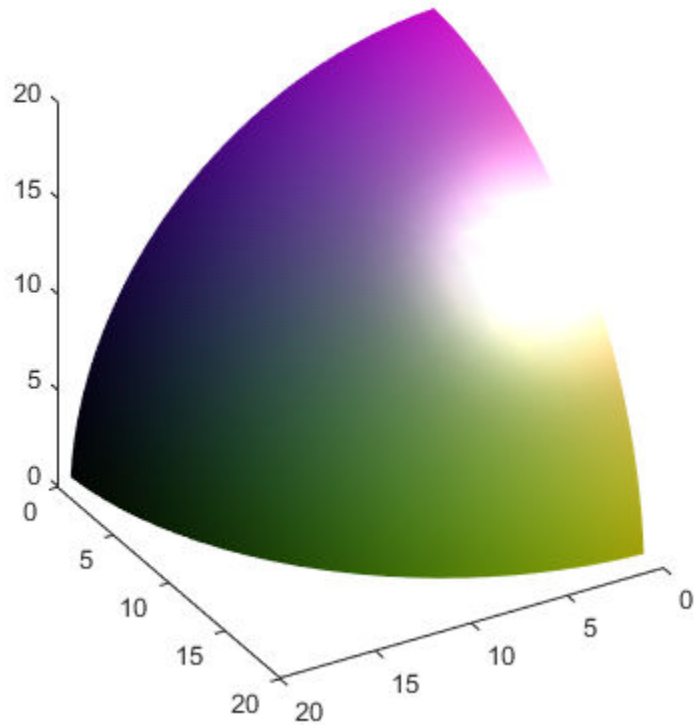


True Color Data

This example displays an isosurface and colors it with true color (RGB) data.

```
[x,y,z] = meshgrid(1:20,1:20,1:20);
data = sqrt(x.^2 + y.^2 + z.^2);
p = patch(isosurface(x,y,z,data,20));
isonormals(x,y,z,data,p)
[r,g,b] = meshgrid(20:-1:1,1:20,1:20);
isocolors(x,y,z,r/20,g/20,b/20,p)
p.FaceColor = 'interp';
```

```
p.EdgeColor = 'none';
view(150,30)
daspect([1 1 1])
camlight
lighting gouraud
```

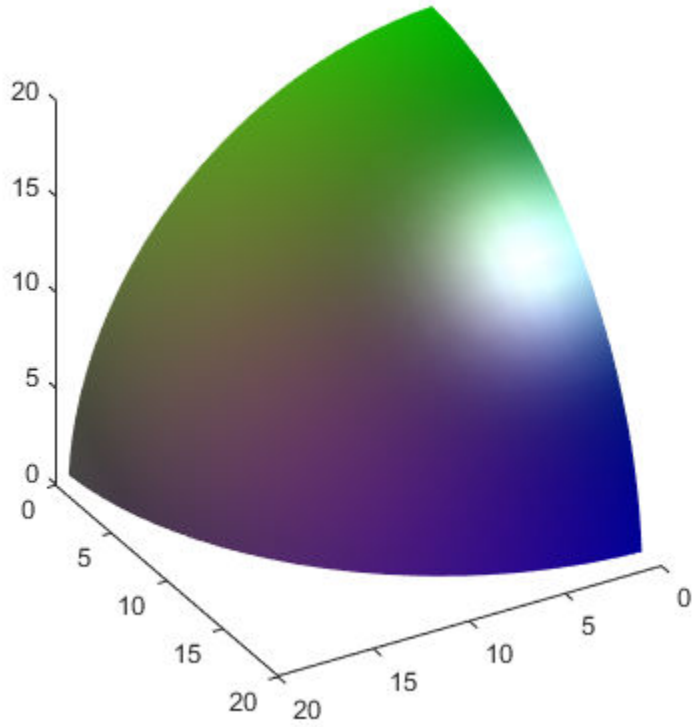


Modified True Color Data

This example uses `isocolors` to calculate the true color data using the isosurface's (patch object's) vertices, but then returns the color data in a variable (`c`) in order to

modify the values. It then explicitly sets the isosurface's FaceVertexCData to the new data (`1-c`).

```
[x,y,z] = meshgrid(1:20,1:20,1:20);
data = sqrt(x.^2 + y.^2 + z.^2);
p = patch(isosurface(data,20));
isonormals(data,p)
[r,g,b] = meshgrid(20:-1:1,1:20,1:20);
c = isocolors(r/20,g/20,b/20,p);
p.FaceVertexCData = 1-c;
p.FaceColor = 'interp';
p.EdgeColor = 'none';
view(150,30)
daspect([1 1 1])
camlight
lighting gouraud
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`isocaps` | `isonormals` | `isosurface` | `reducepatch` | `reducevolume` | `smooth3` | `subvolume`

Introduced before R2006a

isonormals

Compute normals of isosurface vertices

Syntax

```
n = isonormals(X,Y,Z,V,vertices)
n = isonormals(V,vertices)
n = isonormals(V,p)
n = isonormals(X,Y,Z,V,p)
n = isonormals(...,'negate')
isonormals(V,p)
isonormals(X,Y,Z,V,p)
```

Description

`n = isonormals(X,Y,Z,V,vertices)` computes the normals of the isosurface vertices from the vertex list, `vertices`, using the gradient of the data `V`. The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`. The computed normals are returned in `n`.

`n = isonormals(V,vertices)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)` where `[m,n,p] = size(V)`.

`n = isonormals(V,p)` and `n = isonormals(X,Y,Z,V,p)` compute normals from the vertices of the patch identified by the handle `p`.

`n = isonormals(...,'negate')` negates (reverses the direction of) the normals.

`isonormals(V,p)` and `isonormals(X,Y,Z,V,p)` set the `VertexNormals` property of the patch identified by the handle `p` to the computed normals rather than returning the values.

Examples

Isosurface Using Different Types of Surface Normals

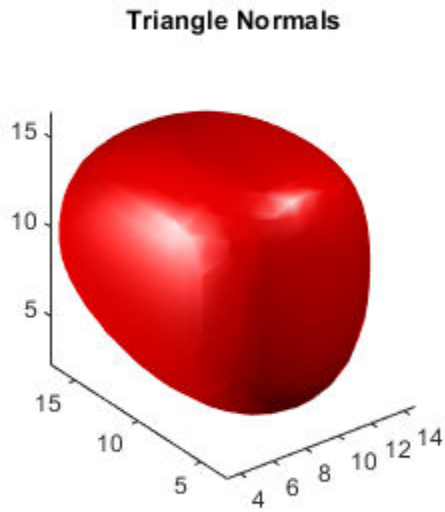
Compare the effect of different surface normals on the visual appearance of lit isosurfaces. In one case, the triangles used to draw the isosurface define the normals. In the other, the isonormals function uses the volume data to calculate the vertex normals based on the gradient of the data points. The latter approach generally produces a smoother-appearing isosurface.

Define a 3-D array of volume data.

```
data = cat(3,[0 .2 0; 0 .3 0; 0 0 0],...
    [.1 .2 0; 0 1 0; .2 .7 0],...
    [0 .4 .2; .2 .4 0;.1 .1 0]);
data = interp3(data,3,'cubic');
```

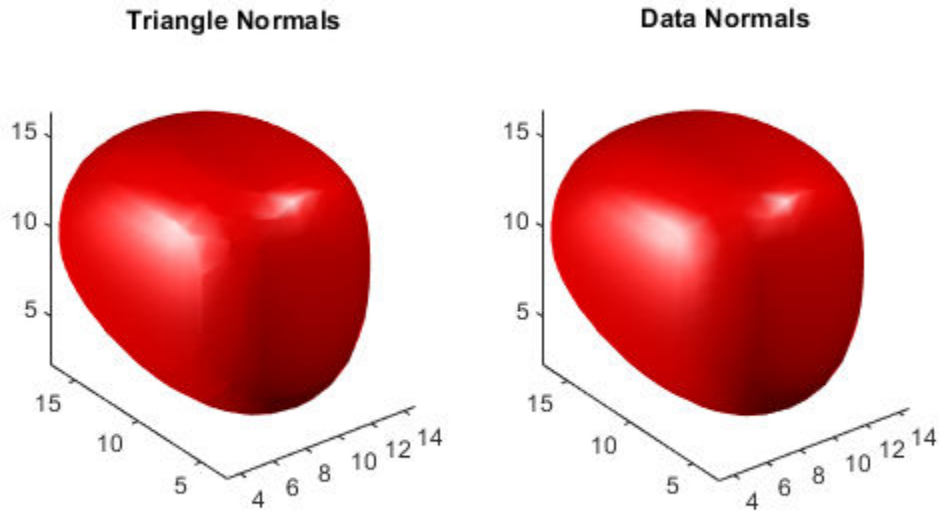
Draw an isosurface from the volume data and add lights. This isosurface uses triangle normals.

```
figure
subplot(1,2,1)
fv = isosurface(data,.5);
p1 = patch(fv,'FaceColor','red','EdgeColor','none');
view(3)
daspect([1,1,1])
axis tight
camlight
camlight(-80,-10)
lighting gouraud
title('Triangle Normals')
```



Draw the same lit isosurface using normals calculated from the volume data.

```
subplot(1,2,2)
fv = isosurface(data,.5);
p2 = patch(fv,'FaceColor','red','EdgeColor','none');
isonormals(data,p2)
view(3)
daspect([1 1 1])
axis tight
camlight
camlight(-80,-10)
lighting gouraud
title('Data Normals')
```



These isosurfaces illustrate the difference between triangle and data normals.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`interp3` | `isocaps` | `isosurface` | `reducepatch` | `reducevolume` | `smooth3` | `subvolume`

Introduced before R2006a

isordinal

Determine whether input is ordinal categorical array

Syntax

```
tf = isordinal(A)
```

Description

`tf = isordinal(A)` returns logical `1` (`true`) if `A` is an ordinal categorical array. Otherwise, `isordinal` returns logical `0` (`false`).

If a categorical array is ordinal, you can use relational operations for inequality comparisons, such as greater and less than, in addition to tests for equality.

Examples

Determine Whether Categorical Array Is Ordinal

Create a categorical array containing the sizes of 10 objects. Use the names `small`, `medium`, and `large` for the values `'S'`, `'M'`, and `'L'`.

```
A = categorical({'M';'L';'S';'S';'M';'L';'M';'L';'M';'S'},...
    {'S','M','L'},{'small','medium','large'})
```

```
A = 10x1 categorical array
     medium
     large
     small
     small
     medium
     large
     medium
     large
     medium
```

```
     small
```

Determine if the categories of A have a mathematical ordering.

```
isordinal(A)
```

```
ans = logical
   0
```

A is not ordinal. You must use the `'Ordinal',true` name-value pair argument in the function `categorical` to create an ordinal categorical array.

Input Arguments

A — Categorical array
vector | matrix | multidimensional array

Categorical array, specified as a vector, matrix, or multidimensional array.

Tips

- To convert a categorical array, A, from nonordinal to ordinal, use `A = categorical(A,'Ordinal',true)`.
- To convert a categorical array, A, from ordinal to nonordinal, use `A = categorical(A,'Ordinal',false)`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This function fully supports code generation. For more information, see “Code Generation for Categorical Arrays” (MATLAB Coder).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`categorical` | `categories`

Topics

“Ordinal Categorical Arrays”

Introduced in R2013b

isosurface

Extract isosurface data from volume data

Syntax

```
fv = isosurface(X,Y,Z,V,isovalue)
fv = isosurface(V,isovalue)
fvc = isosurface(...,colors)
fv = isosurface(...,'noshare')
fv = isosurface(...,'verbose')
[f,v] = isosurface(...)
[f,v,c] = isosurface(...)
isosurface(...)
```

Description

`fv = isosurface(X,Y,Z,V,isovalue)` computes isosurface data from the volume data `V` at the isosurface value specified in `isovalue`. That is, the isosurface connects points that have the specified value much the way contour lines connect points of equal elevation.

The arrays `X`, `Y`, and `Z` represent a Cartesian, axis-aligned grid. `V` contains the corresponding values at these grid points. The coordinate arrays (`X`, `Y`, and `Z`) must be monotonic and conform to the format produced by `meshgrid`. `V` must be a 3D volume array of the same size as `X`, `Y`, and `Z`.

The `struct` `fv` contains the faces and vertices of the isosurface, which you can pass directly to the `patch` command.

`fv = isosurface(V,isovalue)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)` where `[m,n,p] = size(V)`.

`fvc = isosurface(...,colors)` interpolates the array `colors` onto the scalar field and returns the interpolated values in the `facevertexcdata` field of the `fvc` structure. The size of the `colors` array must be the same as `V`. The `colors` argument enables you to control the color mapping of the isosurface with data different from that used to

calculate the isosurface (e.g., temperature data superimposed on a wind current isosurface).

`fv = isosurface(...,'noshare')` does not create shared vertices. This is faster, but produces a larger set of vertices.

`fv = isosurface(...,'verbose')` prints progress messages to the command window as the computation progresses.

`[f,v] = isosurface(...)` or `[f,v,c] = isosurface(...)` returns the faces and vertices (and `faceVertexcCData`) in separate arrays instead of a struct.

`isosurface(...)` with no output arguments, creates a patch in the current axes with the computed faces and vertices. If no current axes exists, a new axes is created with a 3-D view.

Special Case Behavior — isosurface Called with No Output Arguments

If there is no current axes and you call `isosurface` without assigning output arguments, MATLAB creates a new axes, sets it to a 3-D view, and adds lighting to the isosurface graph.

Examples

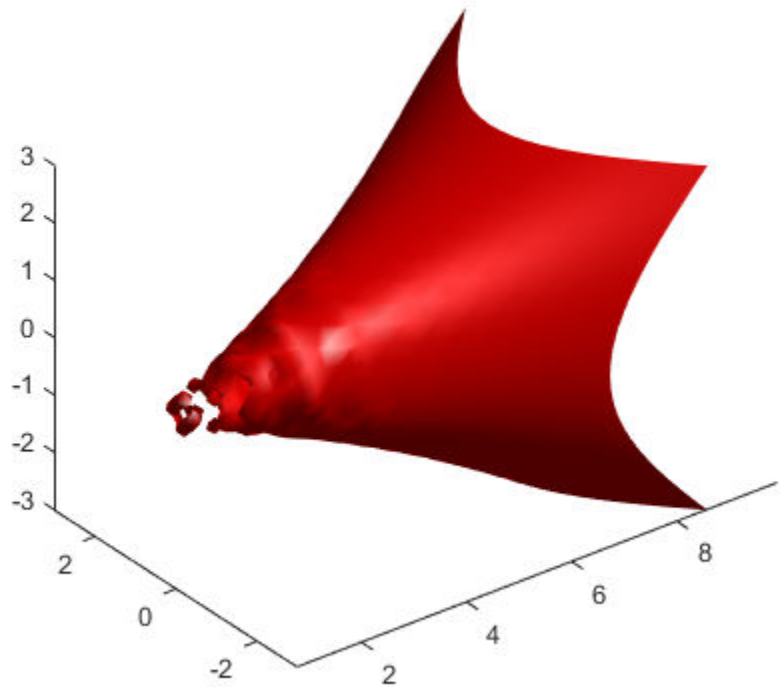
Draw Isosurface with Lighting

Load the flow data set, which represents the speed profile of a submerged jet within an infinite tank. Draw the isosurface at the data value of `-3` and prepare the isosurface for lighting by:

- Recalculating the isosurface normals based on the volume data.
- Setting the face and edge color.
- Specifying the view.
- Adding lights.

```
[x,y,z,v] = flow;
p = patch(isosurface(x,y,z,v,-3));
```

```
isonormals(x,y,z,v,p)
p.FaceColor = 'red';
p.EdgeColor = 'none';
daspect([1 1 1])
view(3);
axis tight
camlight
lighting gouraud
```

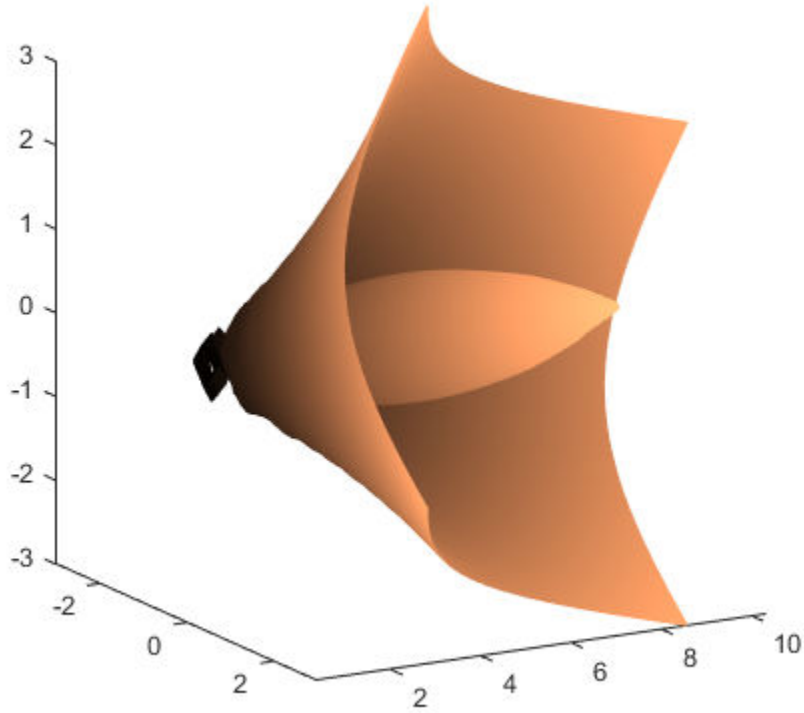


Set Isosurface Colors

Visualize the flow data, but color-code the surface to indicate magnitude along the *x*-axis. Use a sixth argument to `isosurface`, which provides a means to overlay another data

set by coloring the resulting isosurface. The `colors` variable is a vector containing a scalar value for each vertex in the isosurface, to be portrayed with the current color map. In this case, it is one of the variables that define the surface, but it could be entirely independent. You can apply a different color scheme by changing the current figure color map.

```
[x,y,z,v] = flow;
[faces,verts,colors] = isosurface(x,y,z,v,-3,x);
patch('Vertices',verts,'Faces',faces,'FaceVertexCData',colors,...
    'FaceColor','interp','EdgeColor','interp')
view(30,-15)
axis vis3d
colormap copper
```



Tips

You can pass the `fv` structure created by `isosurface` directly to the `patch` command, but you cannot pass the individual faces and vertices arrays (`f`, `v`) to `patch` without specifying property names. For example,

```
patch(isosurface(X,Y,Z,V,isovalue))
```

or

```
[f,v] = isosurface(X,Y,Z,V,isovalue);
patch('Faces',f,'Vertices',v)
```

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`isonormals` | `shrinkfaces` | `smooth3` | `subvolume`

Topics

"Connecting Equal Values with Isosurfaces"

Introduced before R2006a

isoutlier

Find outliers in data

Syntax

```
TF = isoutlier(A)
TF = isoutlier(A,method)
TF = isoutlier(A,'percentiles',threshold)
TF = isoutlier(A,movmethod,window)
TF = isoutlier( ___ ,dim)
TF = isoutlier( ___ ,Name,Value)
[TF,L,U,C] = isoutlier( ___ )
```

Description

`TF = isoutlier(A)` returns a logical array whose elements are `true` when an outlier is detected in the corresponding element of `A`. By default, an outlier is a value that is more than three scaled median absolute deviations (MAD) on page 1-8200 away from the median. If `A` is a matrix or table, then `isoutlier` operates on each column separately. If `A` is a multidimensional array, then `isoutlier` operates along the first dimension whose size does not equal 1.

`TF = isoutlier(A,method)` specifies a method for detecting outliers. For example, `isoutlier(A,'mean')` returns `true` for all elements more than three standard deviations from the mean.

`TF = isoutlier(A,'percentiles',threshold)` defines outliers as points outside of the percentiles specified in `threshold`. The `threshold` argument is a two-element row vector containing the lower and upper percentile thresholds, such as `[10 90]`.

`TF = isoutlier(A,movmethod,window)` specifies a moving method for detecting local outliers according to a window length defined by `window`. For example, `isoutlier(A,'movmedian',5)` returns `true` for all elements more than three local scaled MAD from the local median within a sliding window containing five elements.

`TF = isoutlier( ___ ,dim)` operates along dimension `dim` of `A` for any of the previous syntaxes. For example, `isoutlier(A,2)` operates on each row of a matrix `A`.

`TF = isoutlier( ___ ,Name,Value)` specifies additional parameters for detecting outliers using one or more name-value pair arguments. For example, `isoutlier(A,'SamplePoints',t)` detects outliers in `A` relative to the corresponding elements of a time vector `t`.

`[TF,L,U,C] = isoutlier( ___ )` also returns the lower and upper thresholds and the center value used by the outlier detection method.

Examples

Detect Outliers in Vector

Find the outliers in a vector of data. A logical 1 in the output indicates the location of an outlier.

```
A = [57 59 60 100 59 58 57 58 300 61 62 60 62 58 57];
TF = isoutlier(A)
```

```
TF = 1x15 logical array

   0   0   0   1   0   0   0   0   1   0   0   0   0   0   0
```

Detect Outliers using Mean

Define outliers as points more than three standard deviations from the mean, and find the locations of outliers in a vector.

```
A = [57 59 60 100 59 58 57 58 300 61 62 60 62 58 57];
TF = isoutlier(A,'mean')
```

```
TF = 1x15 logical array

   0   0   0   0   0   0   0   0   1   0   0   0   0   0   0
```

Detect Outliers with Sliding Window

Create a vector of data containing a local outlier.

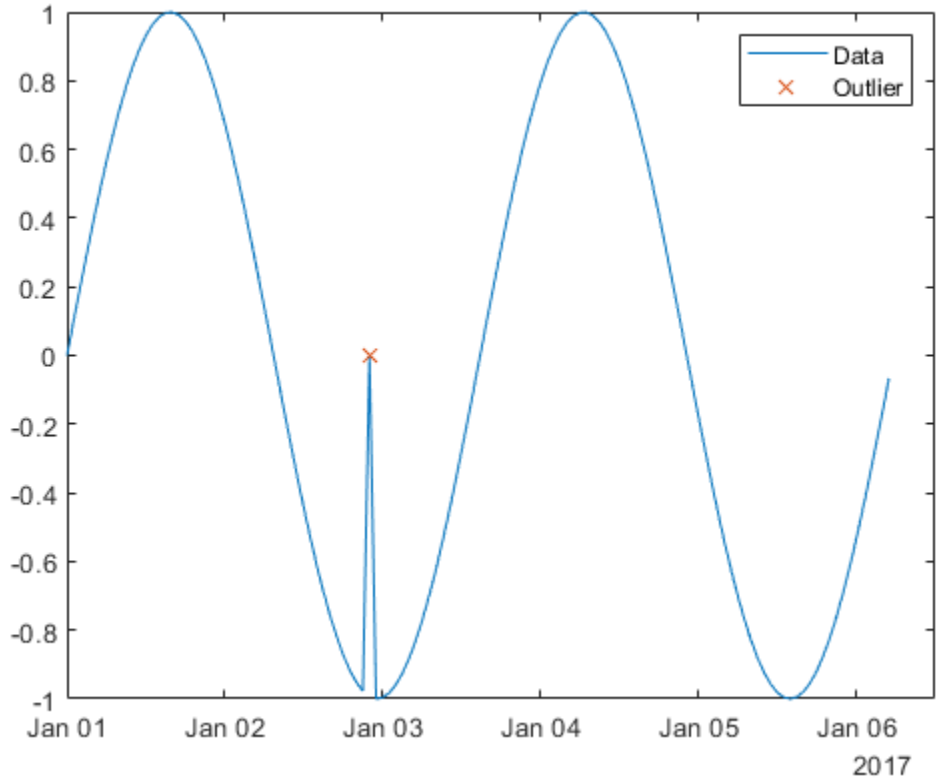
```
x = -2*pi:0.1:2*pi;
A = sin(x);
A(47) = 0;
```

Create a time vector that corresponds to the data in A.

```
t = datetime(2017,1,1,0,0,0) + hours(0:length(x)-1);
```

Define outliers as points more than three local scaled MAD away from the local median within a sliding window. Find the locations of the outliers in A relative to the points in t with a window size of 5 hours. Plot the data and detected outliers.

```
TF = isoutlier(A,'movmedian',hours(5),'SamplePoints',t);
plot(t,A,t(TF),A(TF),'x')
legend('Data','Outlier')
```



Matrix of Data

Find outliers for each row of a matrix.

Create a matrix of data containing outliers along the diagonal.

```
A = magic(5) + diag(200*ones(1,5))
```

```
A = 5×5

   217    24     1     8    15
    23   205     7    14    16
```

```
     4     6   213    20    22
    10    12    19   221     3
    11    18    25     2   209
```

Find the locations of outliers based on the data in each row.

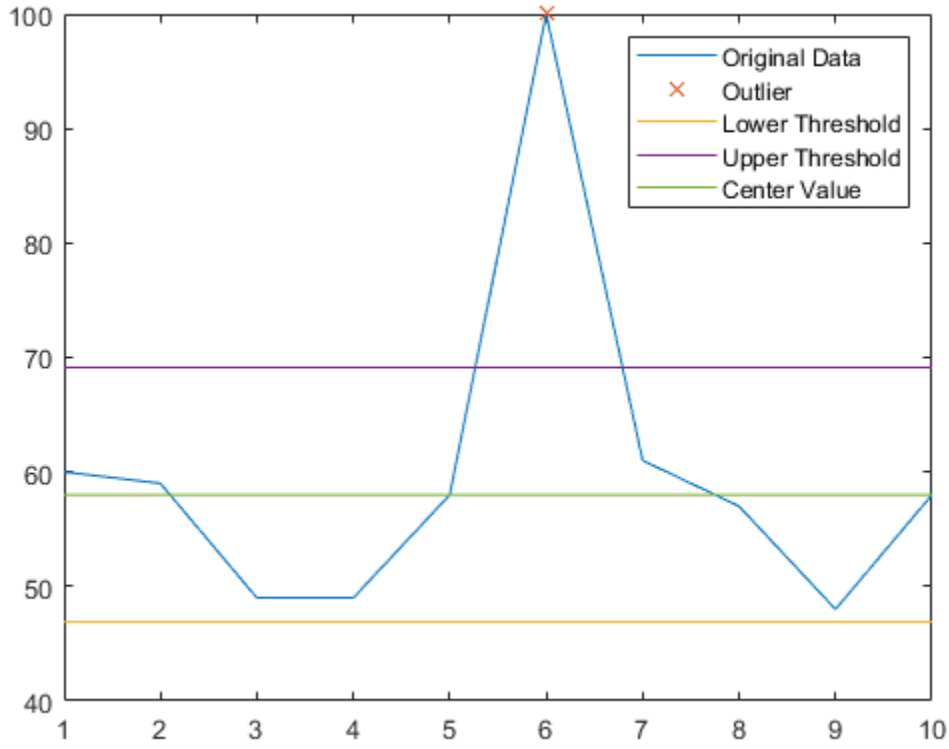
```
TF = isoutlier(A,2)
```

```
TF = 5x5 logical array

   1   0   0   0   0
   0   1   0   0   0
   0   0   1   0   0
   0   0   0   1   0
   0   0   0   0   1
```

Compute Outlier Thresholds

Create a vector of data containing an outlier. Find and plot the location of the outlier, and the thresholds and center value determined by the outlier method. The center value is the median of the data, and the upper and lower thresholds are three scaled MAD above and below the median.

```
x = 1:10;
A = [60 59 49 49 58 100 61 57 48 58];
[TF,L,U,C] = isoutlier(A);
plot(x,A,x(TF),A(TF),'x',x,L*ones(1,10),x,U*ones(1,10),x,C*ones(1,10))
legend('Original Data','Outlier','Lower Threshold','Upper Threshold','Center Value')
```



Input Arguments

A — Input data
vector | matrix | multidimensional array | table | timetable

Input data, specified as a vector, matrix, multidimensional array, table, or timetable.

If `A` is a table, then its variables must be of type `double` or `single`, or you can use the `'DataVariables'` name-value pair to list `double` or `single` variables explicitly. Specifying variables is useful when you are working with a table that contains variables with data types other than `double` or `single`.

If A is a timetable, then `isoutlier` operates only on the table elements. Row times must be unique and listed in ascending order.

Data Types: `double` | `single` | `table` | `timetable`

`method` — Method for detecting outliers
`'median'` (default) | `'mean'` | `'quartiles'` | `'grubbs'` | `'gesd'`

Method for detecting outliers, specified as one of the following:

Method	Description
`'median'`	Returns `true` for elements more than three scaled MAD from the median. The scaled MAD is defined as `c*median(abs(A-median(A)))`, where `c=-1/(sqrt(2)*erfcinv(3/2))`.
`'mean'`	Returns `true` for elements more than three standard deviations from the mean. This method is faster but less robust than `'median'`.
`'quartiles'`	Returns `true` for elements more than 1.5 interquartile ranges above the upper quartile or below the lower quartile. This method is useful when the data in A is not normally distributed.
`'grubbs'`	Applies Grubbs's test for outliers, which removes one outlier per iteration based on hypothesis testing. This method assumes that the data in A is normally distributed.
`'gesd'`	Applies the generalized extreme Studentized deviate test for outliers. This iterative method is similar to `'grubbs'`, but can perform better when there are multiple outliers masking each other.

`threshold` — Percentile thresholds
two-element row vector

Percentile thresholds, specified as a two-element row vector whose elements are in the interval [0,100]. The first element indicates the lower percentile threshold and the second element indicates the upper percentile threshold. For example, a threshold of `[10 90]` defines outliers as points below the 10th percentile and above the 90th percentile. The first element of `threshold` must be less than the second element.

`movmethod` — Moving method
`'movmedian'` | `'movmean'`

Moving method for detecting outliers, specified as one of the following:

Method	Description
`'movmedian'`	Returns `true` for elements more than three local scaled MAD from the local median over a window length specified by `window`.
`'movmean'`	Returns `true` for elements more than three local standard deviations from the local mean over a window length specified by `window`.

`window` — Window length
positive integer scalar | two-element vector of positive integers | positive duration scalar | two-element vector of positive durations

Window length, specified as a positive integer scalar, a two-element vector of positive integers, a positive duration scalar, or a two-element vector of positive durations.

When `window` is a positive integer scalar, the window is centered about the current element and contains `window-1` neighboring elements. If `window` is even, then the window is centered about the current and previous elements.

When `window` is a two-element vector of positive integers `[b f]`, the window contains the current element, `b` elements backward, and `f` elements forward.

When `A` is a timetable or `'SamplePoints'` is specified as a `datetime` or `duration` vector, then `window` must be of type `duration`, and the windows are computed relative to the sample points.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

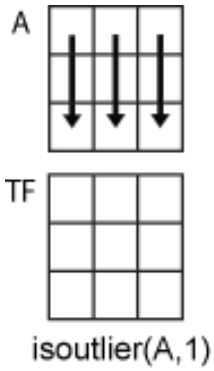
`dim` — Dimension to operate along

positive integer scalar

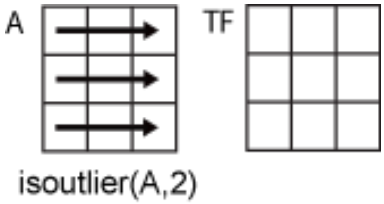
Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a matrix `A`.

`isoutlier(A,1)` detects outliers based on the data in each column of `A`.



`isoutlier(A,2)` detects outliers based on the data in each row of `A`.



When `A` is a table or timetable, `dim` is not supported. `isoutlier` operates along each table or timetable variable separately.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `isoutlier(A,'mean','ThresholdFactor',4)`

`ThresholdFactor` — Detection threshold factor
nonnegative scalar

Detection threshold factor, specified as the comma-separated pair consisting of `'ThresholdFactor'` and a nonnegative scalar.

For methods `'median'` and `'movmedian'`, the detection threshold factor replaces the number of scaled MAD, which is 3 by default.

For methods `'mean'` and `'movmean'`, the detection threshold factor replaces the number of standard deviations from the mean, which is 3 by default.

For methods `'grubbs'` and `'gesd'`, the detection threshold factor is a scalar ranging from 0 to 1. Values close to 0 result in a smaller number of outliers and values close to 1 result in a larger number of outliers. The default detection threshold factor is 0.5.

For the `'quartiles'` method, the detection threshold factor replaces the number of interquartile ranges, which is 1.5 by default.

This name-value pair is not supported when the specified method is `'percentiles'`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`SamplePoints` — Sample points
vector

Sample points, specified as the comma-separated pair consisting of `'SamplePoints'` and a vector. The sample points represent the location of the data in `A`. Sample points do not need to be uniformly sampled. By default, the sample points vector is `[1 2 3 ...]`.

Moving windows are defined relative to the sample points, which must be sorted and contain unique elements. For example, if `t` is a vector of times corresponding to the input data, then `isoutlier(rand(1,10),'movmean',3,'SamplePoints',t)` has a window that represents the time interval between `t(i)-1.5` and `t(i)+1.5`.

When the sample points vector has data type `datetime` or `duration`, then the moving window length must have type `duration`.

Data Types: `double` | `single` | `datetime` | `duration`

`DataVariables` — Table variables
variable name | cell array of variable names | numeric vector | logical vector | function handle

Table variables, specified as the comma-separated pair consisting of `'DataVariables'` and a variable name, a cell array of variable names, a numeric vector, a logical vector, or a function handle. The `'DataVariables'` value indicates which columns of the input table to detect outliers in, and can be one of the following:

- A character vector specifying a single table variable name
- A cell array of character vectors where each element is a table variable name
- A vector of table variable indices
- A logical vector whose elements each correspond to a table variable, where `true` includes the corresponding variable and `false` excludes it
- A function handle that takes the table as input and returns a logical scalar

The data type associated with the indicated variable must be `double` or `single`.

Example: `'Age'`

Example: `{'Height','Weight'}`

Example: `@isnumeric`

Data Types: `char` | `cell` | `double` | `single` | `logical` | `function_handle`

`MaxNumOutliers` — Maximum outlier count
positive integer

Maximum outlier count, for the `'gesd'` method only, specified as the comma-separated pair consisting of `'MaxNumOutliers'` and a positive integer. The `'MaxNumOutliers'` value specifies the maximum number of outliers returned by the `'gesd'` method. For example, `isoutlier(A,'gesd','MaxNumOutliers',5)` returns no more than five outliers.

The default value for `'MaxNumOutliers'` is the integer nearest to 10 percent of the number of elements in `A`. Setting a larger value for the maximum number of outliers can ensure that all outliers are detected, but at the cost of reduced computational efficiency.

The `'gesd'` method assumes the non-outlier input data is sampled from an approximate normal distribution. When the data is not sampled in this way, the number of returned outliers might exceed the `'MaxNumOutliers'` value.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`TF` — Outlier indicator
vector | matrix | multidimensional array

Outlier indicator, returned as a vector, matrix, or multidimensional array. An element of `TF` is `true` when the corresponding element of `A` is an outlier and `false` otherwise. `TF` is the same size as `A`.

Data Types: `logical`

`L` — Lower threshold
scalar | vector | matrix | multidimensional array | table | timetable

Lower threshold used by the outlier detection method, returned as a scalar, vector, matrix, multidimensional array, table, or timetable. For example, the lower value of the default outlier detection method is three scaled MAD below the median of the input data. `L` has the same size as `A` in all dimensions except for the operating dimension where the length is 1.

Data Types: `double` | `single` | `table` | `timetable`

`U` — Upper threshold
scalar | vector | matrix | multidimensional array | table | timetable

Upper threshold used by the outlier detection method, returned as a scalar, vector, matrix, multidimensional array, table, or timetable. For example, the upper value of the default outlier detection method is three scaled MAD above the median of the input data. `U` has the same size as `A` in all dimensions except for the operating dimension where the length is 1.

Data Types: `double` | `single` | `table` | `timetable`

`C` — Center value
scalar | vector | matrix | multidimensional array | table | timetable

Center value used by the outlier detection method, returned as a scalar, vector, matrix, multidimensional array, table, or timetable. For example, the center value of the default outlier detection method is the median of the input data. `C` has the same size as `A` in all dimensions except for the operating dimension where the length is 1.

Data Types: `double` | `single` | `table` | `timetable`

More About

Median Absolute Deviation

For a random variable vector A made up of N scalar observations, the median absolute deviation (MAD) is defined as

$$\text{MAD} = \text{median}(|A_i - \text{median}(A)|)$$

for $i = 1,2,...,N$.

The scaled MAD is defined as `c*median(abs(A-median(A)))` where `c=-1/(sqrt(2)*erfcinv(3/2))`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- The `'percentiles'`, `'grubbs'`, and `'gesd'` methods are not supported.
- The `'movmedian'` and `'movmean'` methods do not support tall timetables.
- The `'SamplePoints'` and `'MaxNumOutliers'` name-value pairs are not supported.
- The value of `'DataVariables'` cannot be a function handle.
- Computation of `isoutlier(A)`, `isoutlier(A,'median',...)`, or `isoutlier(A,'quartiles',...)` along the first dimension is only supported for tall column vectors `A`.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The `'movmean'` and `'movmedian'` methods do not support the `'SamplePoints'` name-value pair argument.
- String and character array inputs must be constant.
- Tables are not supported.

See Also

Clean Outlier Data | `filloutliers` | `ischange` | `islocalmax` | `islocalmin` | `ismissing` | `rmoutliers`

Topics

Clean Messy Data and Locate Extrema Using Live Editor Tasks
“Data Smoothing and Outlier Detection”

Introduced in R2017a

ispc

Determine if version is for Windows (PC) platform

Syntax

```
tf = ispc
```

Description

`tf = ispc` returns logical `1` (`true`) if the version of MATLAB software is for the Microsoft Windows platform. Otherwise, it returns logical `0` (`false`).

Examples

Execute MATLAB Command Based on Platform

```
if ismac
    % Code to run on Mac platform
elseif isunix
    % Code to run on Linux platform
elseif ispc
    % Code to run on Windows platform
else
    disp('Platform not supported')
end
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Returns true or false based on the MATLAB version you use for code generation.
- Use only when the code generation target is S-function (Simulation) or MEX-function.

See Also

`computer` | `is*` | `ismac` | `isstudent` | `isunix`

Introduced before R2006a

ispref

Determine if custom preference exists

Syntax

```
ispref(group,pref)
ispref(group)
```

Description

`ispref(group,pref)` returns `1` if the specified preference in the specified group exists, and `0` otherwise. If `pref` specifies multiple preferences, `ispref` returns a logical array containing a `1` or `0` for each specified preference.

`ispref(group)` returns `1` if the specified group exists, and `0` otherwise.

Examples

Check If Preference Exists

Add a preference called `version` to the `mytoolbox` group of preferences and then check if the preference exists.

```
addpref('mytoolbox','version','1.0')
ispref('mytoolbox','version')
```

```
ans =
  logical
   1
```

Check If Multiple Preferences Exist

Add two preferences to the `mytoolbox` group of preferences and then check if the preferences exist.

```
addpref('mytoolbox',{'modifieddate','docpath'},{'1/9/2019','C:\mytoolbox\documentation'})
ispref('mytoolbox',{'modifieddate','docpath','hasdoc'})
```

```
ans =
  1×3 logical array
   1   1   0
```

MATLAB returns a logical array containing a `1` for the `modifieddate` preference and a `1` for the `docpath` preference. The `hasdoc` preference does not exist and therefore the last value of the logical array is `0`.

Input Arguments

`group` — Custom preference group name

character vector | string scalar

Custom preference group name, specified as a character vector or a string scalar.

Example: `'mytoolbox'`

Data Types: `char` | `string`

`pref` — Custom preference name

character vector | cell array of character vectors | string array

Custom preference name, specified as a character vector, a cell array of character vectors, or a string array.

Example: `'version'`

Example: `{'version','modifieddate','docpath'}`

Data Types: `char` | `string`

See Also

`addpref` | `getpref` | `rmpref` | `setpref` | `uigetpref` | `uisetpref`

Topics

"Preferences"

Introduced before R2006a

isprime

Determine which array elements are prime

Syntax

```
TF = isprime(X)
```

Description

`TF = isprime(X)` returns a logical array the same size as `X`. The value at `TF(i)` is `true` when `X(i)` is a prime number. Otherwise, the value is `false`.

Examples

Determine if Double Integer Values Are Prime

```
tf = isprime([2 3 0 6 10])
```

```
tf = 1x5 logical array

   1   1   0   0   0
```

`2` and `3` are prime, but `0`, `6`, and `10` are not.

Determine if Unsigned Integer Values Are Prime

```
x = uint16([333 71 99]);
tf = isprime(x)
```

```
tf = 1x3 logical array
```

```
0   1   0
```

71 is prime, but 333 and 99 are not.

Input Arguments

X — Input values
scalar, vector, or array of real, nonnegative integer values

Input values, specified as a scalar, vector, or array of real, nonnegative integer values.

Example: `17`

Example: `[1 2 3 4]`

Example: `int16([127 255 4095])`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The maximum double precision input is `2^33`.
- The maximum single precision input is `2^25`.
- The input `X` cannot have type `int64` or `uint64`.

See Also

`is*` | `primes`

Introduced before R2006a

isprop

True if property exists

Syntax

```
tf = isprop(obj,PropertyName)
```

Description

`tf = isprop(obj,PropertyName)` returns `true` if the specified `PropertyName` is a property of object `obj`. Otherwise, `isprop` returns logical `false`. Specify `PropertyName` as a character vector or a string scalar.

If `obj` is an array, `isprop` returns a logical array the same size as `obj`. Each true element of `tf` corresponds to an element of `obj` that has the property, `PropertyName`.

Note If `obj` is an empty object or an array of empty objects, `isprop` returns an empty logical array, even if `PropertyName` is a property of `obj`.

While `isprop` returns `true` if the class of an object defines a property of that name, classes can control access to property values by defining property attributes. Property access can be defined as:

- Readable and writable
- Read only
- Write only
- Accessible only to certain class methods

Therefore, `isprop` might indicate that a property exists, but you might not be able to access that property. For more information, see “Get Information About Properties”.

Examples

This example uses `isprop` to determine if `XDataSource` is a property of object `h` before attempting to set the property value:

```
h = plot(1:10);
if isprop(h,'XDataSource')
   set(h,'XDataSource','x')
else
   error(['XDataSource not a property of class ',class(h)])
end
```

Since `XDataSource` is a property of `h`, its value is set to `'x'`.

See Also

`ismethod` | `properties`

Topics

"Properties"

Introduced before R2006a

isprotected

Determine whether categories of categorical array are protected

Syntax

```
tf = isprotected(A)
```

Description

`tf = isprotected(A)` returns logical `1` (`true`) if the categories of `A` are protected. Otherwise, `isprotected` returns logical `0` (`false`).

- `true` — When you assign new values to `B`, the values must belong to one of the existing categories. Therefore, you only can combine arrays that have the same categories. To add new categories to `B`, you must use the `addcats` function.
- `false` — When you assign new values to `B`, the categories update automatically. Therefore, you can combine (nonordinal) categorical arrays that have different categories. The categories can update to include the categories from both arrays.

Examples

Determine Whether Categories Are Protected

Create a categorical array containing the sizes of 10 objects. Use the names `small`, `medium`, and `large` for the values `'S'`, `'M'`, and `'L'`.

```
valueset = {'S','M','L'};
catnames = {'small','medium','large'};

A = categorical({'M';'L';'S';'S';'M';'L';'M';'L';'M';'S'},...
    valueset,catnames,'Ordinal',true)
```

```
A = 10x1 categorical array
     medium
```

```
     large
     small
     small
     medium
     large
     medium
     large
     medium
     small
```

A is a 10-by-1 categorical array.

Display the categories of A.

```
categories(A)
```

```
ans = 3x1 cell array
    {'small' }
    {'medium'}
    {'large' }
```

Determine whether the categories of A are protected.

```
tf = isprotected(A)
```

```
tf = logical
   1
```

Since A is an ordinal categorical array, the categories are protected. If you try to add a new value that does not belong to one of the existing categories, for example `A(11) = 'xlarge'`, then an error is returned.

First, use `addcats` to add a new category for `xlarge`.

```
A = addcats(A,'xlarge','After','large');
```

Since A is protected, you can now add a value for `xlarge` since it has an existing category.

```
A(11) = 'xlarge'
```

```
A = 11x1 categorical array
     medium
```

```
large
small
small
medium
large
medium
large
medium
small
xlarge
```

A is now a 11-by-1 categorical array with four categories, such that `small < medium < large < xlarge`.

Input Arguments

A — Categorical array
vector | matrix | multidimensional array

Categorical array, specified as a vector, matrix, or multidimensional array.

The categories of ordinal categorical arrays are always protected.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This function fully supports code generation. For more information, see “Code Generation for Categorical Arrays” (MATLAB Coder).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`categorical` | `categories`

Topics

“Work with Protected Categorical Arrays”

Introduced in R2013b

isreal

Determine whether array is real

Syntax

```
tf = isreal(A)
```

Description

`tf = isreal(A)` returns logical `1` (`true`) when a numeric array `A` does not have an imaginary part, and logical `0` (`false`) otherwise.

Examples

Determine Whether Matrix Contains All Real Values

Define a 3-by-4 matrix, `A`.

```
A = [7 3+4i 2 5i;...
     2i 1+3i 12 345;...
     52 108 78 3];
```

Determine whether the array is real.

```
tf = isreal(A)
```

```
tf = logical
   0
```

Since `A` contains complex elements, `isreal` returns false.

Define Complex Number with Zero-Valued Imaginary Part

Use the `complex` function to create a scalar, `A`, with zero-valued imaginary part.

```
A = complex(12)
```

```
A = 12.0000 + 0.0000i
```

Determine whether `A` is real.

```
tf = isreal(A)
```

```
tf = logical
   0
```

`A` is not real because it has an imaginary part, even though the value of the imaginary part is `0`.

Determine whether `A` contains any elements with zero-valued imaginary part.

```
~any(imag(A))
```

```
ans = logical
   1
```

`A` contains elements with zero-valued imaginary part.

Computation Resulting in Zero-Valued Imaginary Part

Define two complex scalars, `x` and `y`.

```
x=3+4i;
y=5-4i;
```

Determine whether the addition of two complex scalars, `x` and `y`, is real.

```
A = x+y
```

```
A = 8
```

MATLAB® drops the zero imaginary part.

```
isreal(A)
```

```
ans = logical
   1
```

A is real since it does not have an imaginary part.

Find Real Elements in Cell Array

Create a cell array.

```
C{1,1} = pi;                    % double
C{2,1} = 'John Doe';            % char array
C{3,1} = 2 + 4i;                % complex double
C{4,1} = ispc;                  % logical
C{5,1} = magic(3);              % double array
C{6,1} = complex(5,0)           % complex double
```

```
C=6×1 cell
    {[          3.1416]}
    {'John Doe'        }
    {[2.0000 + 4.0000i]}
    {[               1]}
    {3x3 double        }
    {[5.0000 + 0.0000i]}
```

C is a 1-by-6 cell array.

Loop over the elements of a cell array to distinguish between real and complex elements.

```
for k = 1:6
x(k,1) = isreal(C{k,1});
end
```

```
x
```

```
x = 6x1 logical array

   1
   1
   0
```

```
1
1
0
```

All but `C{3,1}` and `C{6,1}` are real arrays.

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

- For numeric data types, if `A` does not have an imaginary part, `isreal` returns `true`; if `A` does have an imaginary part `isreal` returns `false`.
- For `duration`, `calendarDuration`, `logical`, and `char` data types, `isreal` always returns `true`.
- For `string`, `table`, `cell`, `struct`, `datetime`, `function_handle`, and `object` data types, `isreal` always returns `false`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `datetime` | `duration` | `calendarDuration` | `function_handle`
Complex Number Support: Yes

Tips

- To check whether each element of an array `A` is real, use `A == real(A)`.
- `isreal(complex(A))` always returns `false`, even when the imaginary part is all zeros.
- `~isreal(x)` detects arrays that have an imaginary part, even if it is all zeros.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`complex` | `isa` | `isfinite` | `isinf` | `isnan` | `isnumeric` | `isprime`

Topics

“Create Complex Numbers”

Introduced before R2006a

isregular

Determine whether times in timetable are regular

Syntax

```
TF = isregular(TT)
TF = isregular(TT,unit)
[TF,dt] = isregular( ___ )
```

Description

`TF = isregular(TT)` returns `1` (`true`) if the row times in the timetable `TT` are regular. Otherwise, it returns `0` (`false`). The row times are regular if they increase or decrease monotonically by a fixed time step. For example, if consecutive row times always differ by one second, then the times are regular.

`TF = isregular(TT,unit)` returns `1` (`true`) if the row times are regular with respect to the calendar duration unit specified by `unit`. For example, if the row times are datetime values whose year and month components are regular to the month, and `unit` is `'month'`, then `isregular` returns `1`.

`[TF,dt] = isregular( ___ )` also returns `dt`, the fixed time step between row times. If `TT` is regular, then `dt` is either a duration or a calendar duration. If `TT` is not regular, then `dt` is a `NaN` value.

Examples

Determine if Timetable is Regular

Create a timetable using a monthly time vector. Determine whether it is regular with respect to time, and then with respect to months.

Create a timetable whose row times are the first five months of the year `2016`. Add the monthly price of a stock as a table variable.

```
StockPrice = [109.0;107.82;113.17;128.01;116];
M = timetable(datetime(2016,1:5,3)',StockPrice)
```

```
M=5×2 timetable
       Time        StockPrice
    ___________    __________

    03-Jan-2016         109
    03-Feb-2016      107.82
    03-Mar-2016      113.17
    03-Apr-2016      128.01
    03-May-2016         116
```

Determine whether `M` is a regular timetable.

```
TF = isregular(M)
```

```
TF = logical
   0
```

`M` is not regular because the first five months have different numbers of days. You can use the `diff` function to calculate the differences in the time steps between consecutive times in `M`. The differences are durations, formatted to display the time steps as hours, minutes, and seconds.

```
D = diff(M.Time)
```

```
D = 4x1 duration array
   744:00:00
   696:00:00
   744:00:00
   720:00:00
```

Determine whether `M` is regular with respect to months, by specifying `'month'` as the unit of measure.

```
TF = isregular(M,'months')
```

```
TF = logical
   1
```

Determine Time Step of Row Times

Create a timetable. Determine if it is regular, and return the size of the time step if it is.

```
Time = [minutes(0):minutes(15):minutes(60)]';
Pulse = [72 75 80 73 69]';
TT = timetable(Time,Pulse)
```

```
TT=5×2 timetable
     Time     Pulse
    ______    _____

    0 min      72
    15 min     75
    30 min     80
    45 min     73
    60 min     69
```

```
[TF,dt] = isregular(TT)
```

```
TF = logical
   1


dt = duration
   15 min
```

TT is a regular timetable.

Input Arguments

TT — Input timetable
timetable

Input timetable.

unit — Calendar duration unit
character vector | string scalar

Calendar duration unit, specified as a character vector or string scalar. `isregular` determines if the row times of `TT` are regular to the calendar unit specified by `unit`. The table lists the calendar duration units you can specify.

Time Unit	Description
`'years'`	Regular to the year
`'quarters'`	Regular to the quarter
`'months'`	Regular to the month
`'weeks'`	Regular to the week
`'days'`	Regular to the day
`'time'` (default)	Regular with respect to time

Output Arguments

TF — Regularity of row times
`1` | `0`

Regularity of row times, returned as a logical `1` if the row times are regular, and a logical `0` if they are not.

`dt` — Time step between row times
duration | calendar duration

Time step between row times, returned as a duration or a calendar duration. If the timetable is not regular, then `dt` is a `NaN` value.

Tips

- In certain cases, you can create a timetable while specifying a regular time step between row times, and yet the resulting timetable is irregular. This result occurs when you specify the time step using a calendar unit of time and there is a row time that introduces an irregular step. For example, if you create a timetable with a time step of one calendar month, starting on January 31, 2019, then it is irregular with respect to months.

```
stime = datetime(2019,1,31);
tstep = calmonths(1);
```

```
TT = timetable('Size',[3 1],'VariableTypes',{'double'},...
               'TimeStep',tstep,'StartTime',stime);
tf = isregular(TT,'month')
```

```
tf =

  logical

   0
```

In addition, there are other cases where irregularities are due to shifts from Daylight Saving Time (DST) or to row times that are leap seconds. This table specifies the row time values and time steps that can produce irregular timetables unexpectedly.

Row Time Value	Time Step
Start time specified as the 29th, 30th, or 31st day of the month	Number of calendar months or quarters
Start time specified as February 29	Number of calendar years
Any row time occurring between 1:00 a.m. and 2:00 a.m. on a day shifting from DST to standard time (when row times are specified as datetime values whose time zone observes DST)	Number of calendar days or months
Any row time that is a leap second (when row times are specified as datetime values whose time zone is the `UTCLeapSeconds` time zone)	Time step specified in any calendar unit (days, weeks, months, quarters, or years)

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

See Also

`caldiff` | `calendarDuration` | `datetime` | `diff` | `duration` | `issorted` | `retime` | `sortrows` | `split` | `synchronize`

Topics

"Create Timetables"
"Clean Timetable with Missing, Duplicate, or Nonuniform Times"
"Select Timetable Data by Row Time and Variable Type"
"Access Data in Tables"
"Tables"
"Represent Dates and Times in MATLAB"

Introduced in R2016b

isrow

Determine whether input is row vector

Syntax

```
isrow(V)
```

Description

`isrow(V)` returns logical 1 (`true`) if `size`(`V`) returns `[1 n]` with a nonnegative integer value `n`, and logical 0 (`false`) otherwise.

Examples

Determine Row Vector

Determine if a vector is a row or not.

```
V = rand(5,1);
isrow(V)
```

```
ans = logical
   0
```

```
Vt = V';
isrow(Vt)
```

```
ans = logical
   1
```

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`iscolumn` | `ismatrix` | `isscalar` | `isvector`

Introduced in R2010b

isscalar

Determine whether input is scalar

Syntax

```
TF = isscalar(A)
```

Description

`TF = isscalar(A)` returns logical `1` (`true`) if `A` is a scalar. Otherwise, it returns logical `0` (`false`). A scalar is a two-dimensional array that has a size of 1-by-1.

Examples

Determine Scalar from Matrix

Create a 2-by-2 matrix. Determine whether it is a scalar.

```
A = [1 2; 3 4];
TF = isscalar(A)
```

```
TF = logical
   0
```

Check whether the element at the first row and second column of the matrix is a scalar.

```
TF = isscalar(A(1,2))
```

```
TF = logical
   1
```

String Scalar and Character Vector

Create a string scalar by enclosing a piece of text in double quotes. Determine whether it is a scalar.

```
A = "Hello, World!";
TF = isscalar(A)
```

```
TF = logical
   1
```

Now create a character vector by enclosing a piece of text in single quotes. Determine whether it is a scalar.

```
B = 'Hello, World!';
TF = isscalar(B)
```

```
TF = logical
   0
```

Check the dimension of `B` using `size`. `B` is not a scalar since it has a size of 1-by-13.

```
sz = size(B)
```

```
sz = 1×2

     1    13
```

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`isStringScalar` | `iscolumn` | `ismatrix` | `isrow` | `isvector` | `size`

Introduced before R2006a

issorted

Determine if array is sorted

Syntax

```
TF = issorted(A)
TF = issorted(A,dim)
TF = issorted( ___ ,direction)
TF = issorted( ___ ,Name,Value)

TF = issorted(A,'rows')
```

Description

`TF = issorted(A)` returns the logical scalar 1 (`true`) when the elements of `A` are listed in ascending order and 0 (`false`) otherwise.

- If `A` is a vector, then `issorted` returns 1 when the vector elements are in ascending order.
- If `A` is a matrix, then `issorted` returns 1 when each column of `A` is in ascending order.
- If `A` is a multidimensional array, then `issorted` returns 1 when `A` is in ascending order along the first dimension whose size does not equal 1.
- If `A` is a timetable, then `issorted` returns 1 when its row time vector is in ascending order. To check the ordering of row times or variables of a timetable with additional options, use the `issortedrows` function.

`TF = issorted(A,dim)` returns 1 when `A` is sorted along dimension `dim`. For example, if `A` is a matrix, then `issorted(A,2)` returns 1 when each row of `A` is in ascending order.

`TF = issorted( ___ ,direction)` returns 1 when `A` is sorted in the order specified by `direction` for any of the previous syntaxes. For example, `issorted(A,'monotonic')` returns 1 if the elements of `A` are ascending or descending.

`TF = issorted( ___ ,Name,Value)` specifies additional parameters for checking sort order. For example, `issorted(A,'ComparisonMethod','abs')` checks if `A` is sorted by magnitude.

`TF = issorted(A,'rows')` returns 1 when the elements of the first column of a matrix are sorted. If the first column contains repeated elements, then `issorted` looks at the ordering of the second column to determine `TF`. In general, `issorted` looks to the column immediately to the right to determine `TF` when the current and previous columns have repeated elements.

- If `A` is a timetable, then `issortedrows` checks if the row time vector is in ascending order.
- This syntax is not supported for a matrix of character vectors.

Note This syntax is not recommended. Use `issortedrows` instead.

Examples

Sorted Vector

Create a vector and check if it is sorted in ascending order.

```
A = [5 12 33 39 78 90 95 107];
issorted(A)
```

```
ans = logical
   1
```

Sorted Matrix Rows

Create a 5-by-5 matrix and check if each row is sorted in descending order.

```
A = magic(5)
```

```
A = 5×5
```

```
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

```
issorted(A,2,'descend')
```

```
ans = logical
   0
```

Sort each row of A in descending order using the `sort` function, and check that the result has descending rows.

```
B = sort(A,2,'descend')
```

```
B = 5×5

    24    17    15     8     1
    23    16    14     7     5
    22    20    13     6     4
    21    19    12    10     3
    25    18    11     9     2
```

```
issorted(B,2,'descend')
```

```
ans = logical
   1
```

2-D Array of Strings

Create a 2-D array of strings and determine if each column is sorted.

```
str = string({'Horse','Chicken';'cow','Goat'})
```

```
str = 2x2 string array
    "Horse"    "Chicken"
    "cow"      "Goat"
```

```
issorted(str)
```

```
ans = logical
   1
```

Determine if the rows are sorted from left to right.

```
issorted(str,2)
```

```
ans = logical
   0
```

Determine if each row is sorted in descending order from left to right.

```
issorted(str,2,'descend')
```

```
ans = logical
   1
```

Complex Vector with NaN

Create a vector containing complex numbers and NaN values.

```
A = [NaN NaN 1+i 1+2i 2+2i 3+i];
```

Check that the NaN elements are placed first within the vector, and that the remaining elements are sorted by real part.

```
issorted(A,'MissingPlacement','first','ComparisonMethod','real')
```

```
ans = logical
   1
```

Since the third and fourth elements of A have equal real part, `issorted` checks if the imaginary part of these elements are also sorted.

```
imag(A(3))
```

```
ans = 1
```

```
imag(A(4))
```

```
ans = 2
```

Input Arguments

A — Input array

vector | matrix | multidimensional array | cell array of character vectors | timetable

Input array, specified as a vector, matrix, multidimensional array, cell array of character vectors, or timetable.

- If A contains missing values, such as NaN, NaT, <undefined>, and missing, then by default, issorted requires that they are placed at the end to return 1.
- If A is complex, then by default, issorted determines sort order by the magnitude of the elements. If there are consecutive elements with equal magnitude, then issorted also checks the phase angle in the interval (-π, π] to break ties.
- If A is a cell array of character vectors or a string array, then issorted determines sort order using the code order for the UTF-16 character encoding scheme. The sort is case-sensitive. For more information on sorted character and string arrays, see “Sort Order for Character and String Arrays” on page 1-8240.

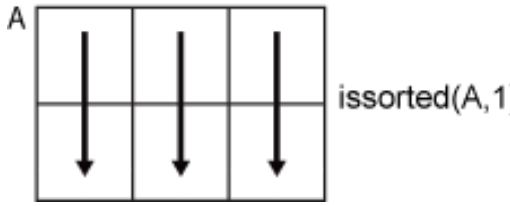
Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char | string | cell | categorical | datetime | duration | timetable
Complex Number Support: Yes

dim — Dimension to operate along

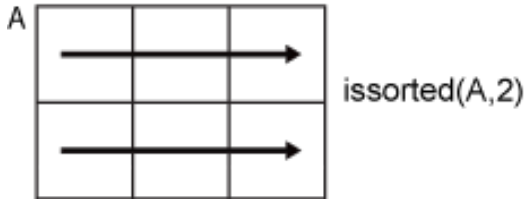
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a matrix A. issorted(A,1) checks if the data in each column of A is sorted.



`issorted(A,2)` checks if the data in each row of `A` is sorted.



`dim` is not supported for timetable input.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`direction` — Sorting direction
`'ascend'` (default) | `'descend'` | `'monotonic'` | `'strictascend'` | `'strictdescend'` | `'strictmonotonic'`

Sorting direction, specified as one of the following:

- `'ascend'` — Checks if data is in ascending order. Data can contain consecutive repeated elements.
- `'descend'` — Checks if data is in descending order. Data can contain consecutive repeated elements.
- `'monotonic'` — Checks if data is in descending or ascending order. Data can contain consecutive repeated elements.
- `'strictascend'` — Checks if data is in strictly ascending order. Data cannot contain duplicate or missing elements.
- `'strictdescend'` — Checks if data is in strictly descending order. Data cannot contain duplicate or missing elements.
- `'strictmonotonic'` — Checks if data is in strictly descending or strictly ascending order. Data cannot contain duplicate or missing elements.

`direction` is not supported for timetable input. Use `issortedrows` instead.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `issorted(A,'MissingPlacement','last')`

`MissingPlacement` — Placement of missing values
`'auto'` (default) | `'first'` | `'last'`

Placement of missing values (`NaN`, `NaT`, `<undefined>`, and `missing`) specified as the comma-separated pair consisting of `'MissingPlacement'` and one of the following:

- `'auto'` — Missing elements are required to be placed last for ascending order and first for descending order to return 1.
- `'first'` — Missing elements are required to be placed first to return 1.
- `'last'` — Missing elements are required to be placed last to return 1.

This name-value pair is not supported for timetable input. Use `issortedrows` instead.

`ComparisonMethod` — Element comparison method
`'auto'` (default) | `'real'` | `'abs'`

Element comparison method, specified as the comma-separated pair consisting of `'ComparisonMethod'` and one of the following:

- `'auto'` — Check if `A` is sorted by `real(A)` when `A` is real, and check if `A` is sorted by `abs(A)` when `A` is complex.
- `'real'` — Check if `A` is sorted by `real(A)` when `A` is real or complex. If `A` has elements with consecutive equal real parts, then check `imag(A)` to break ties.
- `'abs'` — Check if `A` is sorted by `abs(A)` when `A` is real or complex. If `A` has elements with consecutive equal magnitude, then check `angle(A)` in the interval (-π,π] to break ties.

More About

Sort Order for Character and String Arrays

MATLAB stores characters as Unicode using the UTF-16 character encoding scheme. Character and string arrays are sorted according to the UTF-16 code point order. For the characters that are also the ASCII characters, this order means that uppercase letters come before lowercase letters. Digits and some punctuation also come before letters.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The first input argument must not be a cell array.
- If `A` is complex with all zero imaginary parts, then MATLAB might convert `A` to `real(A)` before calling `issorted(A)`. In this case, MATLAB checks that `A` is sorted by `real(A)`, but the generated code checks that `A` is sorted by `abs(A)`. To make the generated code match MATLAB, use `issorted(real(A))` or `issorted(A,'ComparisonMethod','real')`. See “Code Generation for Complex Data with Zero-Valued Imaginary Parts” (MATLAB Coder).
- If you supply `dim`, then it must be constant.
- For limitations related to variable-size inputs, see “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Only the single input syntax `TF = issorted(A)` is supported. The input array `A` must be a vector.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`issortedrows` | `sort` | `sortrows`

Introduced before R2006a

issortedrows

Determine if matrix or table rows are sorted

Syntax

```
TF = issortedrows(A)
TF = issortedrows(A,column)
TF = issortedrows( ___ ,direction)
TF = issortedrows( ___ ,Name,Value)

TF = issortedrows(tblA)

TF = issortedrows(tblA,'RowNames')
TF = issortedrows(tblA,rowDimName)
TF = issortedrows(tblA,vars)
TF = issortedrows( ___ ,direction)
TF = issortedrows( ___ ,Name,Value)
```

Description

`TF = issortedrows(A)` returns the logical scalar 1 (`true`) when the elements of the first column of a matrix `A` are listed in ascending order. Otherwise, `issortedrows` returns 0 (`false`). When the first column has consecutive repeated elements, `issortedrows` determines whether the next column is in ascending order, and repeats this behavior for succeeding equal values.

`TF = issortedrows(A,column)` returns 1 when `A` is sorted based on the columns specified in the vector `column`. For example, `issortedrows(A,[4 6])` first checks if the fourth column of `A` is in ascending order, then checks if the sixth column is in ascending order to break ties.

`TF = issortedrows( ___ ,direction)` returns 1 when the first column of `A` is in the order specified by `direction` for any of the previous syntaxes. For example, `issortedrows(A,'monotonic')` checks if the first column of `A` is in ascending or descending order. `direction` can also be a cell array of character vectors representing multiple directions for each column being checked. For example, `issortedrows(A,[2`

`3],{'ascend' 'descend'})` checks if the second column of `A` is in ascending order, then checks if the third column is in descending order to break ties.

`TF = issortedrows( ___ ,Name,Value)` specifies additional parameters for checking sort order. For example, `issortedrows(A,'ComparisonMethod','abs')` checks if the elements in the first column of `A` are sorted by magnitude.

`TF = issortedrows(tblA)` checks if the rows of a table are in ascending order based on the elements in the first variable. If elements in the first variable are repeated, then `issortedrows` checks the elements in the second variable, and so on.

If `tblA` is a timetable, then `issortedrows` checks if the rows of `tblA` are in ascending order based on its row times. Row times of a timetable label the rows along the first dimension of the timetable.

`TF = issortedrows(tblA,'RowNames')` checks if the rows of a table are in ascending order based on its row names. Row names of a table label the rows along the first dimension of the table.

This syntax is not supported when `tblA` is a timetable.

`TF = issortedrows(tblA,rowDimName)` checks if the rows of a table are sorted by row labels `rowDimName` along the first dimension.

- If `tblA` is a table, then the labels are row names.
- If `tblA` is a timetable, then the labels are row times.

`TF = issortedrows(tblA,vars)` checks if the rows of a table are in ascending order based on the elements in variables `vars`. For example, if `Age` and `Weight` are variables of `tblA`, then `issortedrows(tblA,{'Age','Weight'})` checks if the rows are in ascending order by age, then by weight to break ties.

- If `tblA` is a table with row names, then `vars` can include the row names.
- If `tblA` is a timetable, then `vars` can include the row times.

`TF = issortedrows( ___ ,direction)` checks if a table is sorted in the order specified by `direction` for any of the previous table syntaxes. `direction` can be a single sort order such as `'descend'` or `'monotonic'`, which is applied to each specified variable, row name, or row time. `direction` can also be a cell array whose elements contain different sort orders for each specified variable, row name, or row time that `issortedrows` operates on.

`TF = issortedrows( ___ ,Name,Value)` specifies additional parameters for sorting tables. For example, `issortedrows(tblA,'Var1','MissingPlacement','first')` checks that missing elements in `Var1`, such as `NaN` or `NaT`, are placed at the beginning of the table.

Examples

Matrix Rows

Create a matrix and determine if its rows are in ascending order based on the values in the first column. Since the first column has a repeated element, `sortrows` looks to the second column to determine whether the matrix rows are sorted.

```
A = [1 2 9; 1 5 8; 4 0 7]
```

```
A = 3×3

     1     2     9
     1     5     8
     4     0     7
```

```
TF = issortedrows(A)
```

```
TF = logical
   1
```

Determine if the rows of `A` are in ascending order based on the values in the third column.

```
TF = issortedrows(A,3)
```

```
TF = logical
   0
```

Determine if the rows of `A` are in descending order based on the values in the third column.

```
TF = issortedrows(A,3,'descend')
```

```
TF = logical
   1
```

Complex Matrix

Create a matrix containing complex numbers, and determine if its rows are in ascending order based on the real parts of the elements in the first column. Since the elements in the first column have equal real parts, `issortedrows` then checks the imaginary parts to break the tie.

```
A = [1+i 2i; 1+2i 3+4i]
```

```
A = 2×2 complex

   1.0000 + 1.0000i   0.0000 + 2.0000i
   1.0000 + 2.0000i   3.0000 + 4.0000i
```

```
TF = issortedrows(A,'ComparisonMethod','real')
```

```
TF = logical
   1
```

Table Rows

For a table that describes patient information for five people, determine how the rows of the table are sorted.

Create a table with four variables, and determine if the rows of the table are in ascending order based on age. Since the age variable contains a repeated element, `issortedrows` then checks the next column (`Height`) to break the tie.

```
LastName = {'Sweet';'Jacobson';'Wang';'Joiner';'Berger'};
Age = [38;38;40;43;49];
Height = [69;71;64;67;64];
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];
```

```
tblA = table(Age,Height,Weight,BloodPressure,'RowNames',LastName)
```

```
tblA=5×4 table
                Age    Height    Weight    BloodPressure
                ___    ______    ______    _____________

    Sweet       38       69       176       124     93
    Jacobson    38       71       163       109     77
    Wang        40       64       131       125     83
    Joiner      43       67       133       117     75
    Berger      49       64       119       122     80
```

```
TF = issortedrows(tblA)
```

```
TF = logical
   1
```

Check if the table rows are sorted by last name, which are the row names for `tblA`.

```
TF = issortedrows(tblA,'RowNames')
```

```
TF = logical
   0
```

Check if the table rows are in ascending order by age, then in descending order by weight.

```
TF = issortedrows(tblA,{'Age','Weight'},{'ascend','descend'})
```

```
TF = logical
   1
```

Timetable with Missing Times

Create a timetable, and check that the rows of the timetable are in ascending order based on the row times. Also check that missing elements are placed last.

```
Time = [seconds(1:3) NaN NaN]';
TT = timetable(Time,[98;97.5;97.9;98.1;99.9],[120;111;119;117;112],...
               'VariableNames',{'Temperature','Distance'})
```

```
TT=5×3 timetable
     Time      Temperature    Distance
    _______    ___________    ________

    1 sec           98          120
    2 sec         97.5          111
    3 sec         97.9          119
    NaN sec       98.1          117
    NaN sec       99.9          112
```

```
TF = issortedrows(TT,'Time','MissingPlacement','last')
```

```
TF = logical
   1
```

Input Arguments

A — Input array
column vector | matrix

Input array, specified as a column vector or matrix.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `categorical` | `datetime` | `duration`
Complex Number Support: Yes

`column` — Column sorting vector
nonzero integer scalar | vector of nonzero integers

Column sorting vector, specified as a nonzero integer scalar or a vector of nonzero integers. Each specified integer value indicates a column to check for sort order. Negative integers indicate that the sort order is descending.

`direction` — Sorting direction
character vector | cell array of character vectors

Sorting direction, specified as one of the following:

- `'ascend'` (default) — Checks if data is in ascending order. Data can contain consecutive repeated elements.
- `'descend'` — Checks if data is in descending order. Data can contain consecutive repeated elements.
- `'monotonic'` — Checks if data is in descending or ascending order. Data can contain consecutive repeated elements.
- `'strictascend'` — Checks if data is in strictly ascending order. Data cannot contain duplicate or missing elements.
- `'strictdescend'` — Checks if data is in strictly descending order. Data cannot contain duplicate or missing elements.
- `'strictmonotonic'` — Checks if data is in strictly descending or strictly ascending order. Data cannot contain duplicate or missing elements.

`direction` can also be a cell array containing a list of these character vectors, where each element in the list corresponds to a column of `A`. For example, `issortedrows(A,[2 4],{'ascend' 'descend'})` first checks if the rows of `A` are in ascending order based on the second column. Then, to break ties, `issortedrows` checks if the rows are in descending order based on the fourth column.

If `column` is specified, then the number of elements in the cell array must match the length of `column`. When `column` is not specified, the cell array must contain an element for every column of `A`, or a single element that is applied to all columns.

Data Types: `char` | `cell`

`tblA` — Input table
table | timetable

Input table, specified as a table or a timetable. Each variable in `tblA` must be a valid input to `sort` or `sortrows`.

Data Types: `table` | `timetable`

`rowDimName` — Name of first dimension of input table or timetable
character vector

Name of the first dimension of the input table or timetable, specified as a character vector.

- If `tblA` is a table with row names, then `rowDimName` is the name of the first dimension of the table. By default, the name of the first dimension is `'Row'`. Dimension names are a property of tables. You can access the dimension names of `tblA` using `tblA.Properties.DimensionNames`.
- If `tblA` is a timetable, then `rowDimName` is the name of the vector of row times. You can specify its name when you create a timetable, such as `Time` or `Date`. You can also access the dimension names using `tblA.Properties.DimensionNames`.

Example: If a table `T` has row names, and you changed the name of the first dimension using `T.Properties.DimensionName{1} = 'Name'`, then `issortedrows(T,'Name')` checks if the table is sorted by row names.

Example: If a timetable `TT` has a time vector named `Date`, then `issortedrows(TT,'Date')` checks if the timetable is sorted by the dates and times that `Date` contains.

Data Types: `char`

`vars` — Sorting variables

scalar integer | vector of integers | variable name | cell array of variable names | logical vector

Sorting variables, specified as a scalar integer, a vector of integers, a variable name, a cell array of variable names, or a logical vector. `vars` indicates the table variables to sort by.

If an element of `vars` is a positive integer, then `issortedrows` checks if the rows in the corresponding variable in `tblA` are in ascending order. If an element of `vars` is a negative integer, then `issortedrows` checks if the rows in the corresponding variable in `tblA` are in descending order.

Example: `issortedrows(tblA,{'Height','Weight'})` checks if the rows of `tblA` are in ascending order, first by the variable `Height`, then by the variable `Weight` to break ties.

Example: `issortedrows(tblA,[1 4])` first checks if the table rows are in ascending order based on the first variable, then breaks ties by checking if the rows are in ascending order based on the fourth variable.

Example: `issortedrows(TT,{'Time','X'})` checks if the row times of a timetable are in ascending order, then breaks ties by checking if the rows are in ascending order based on the table variable `'X'`.

Data Types: `double` | `single` | `char` | `cell` | `logical`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `issortedrows(A,'MissingPlacement','last')`

`MissingPlacement` — Placement of missing values
`'auto'` (default) | `'first'` | `'last'`

Placement of missing values (`NaN`, `NaT`, `<undefined>`, and `missing`) specified as the comma-separated pair consisting of `'MissingPlacement'` and one of the following:

- `'auto'` — Missing elements are required to be placed last for ascending order and first for descending order to return 1.
- `'first'` — Missing elements are required to be placed first to return 1.
- `'last'` — Missing elements are required to be placed last to return 1.

`ComparisonMethod` — Element comparison method
`'auto'` (default) | `'real'` | `'abs'`

Element comparison method, specified as the comma-separated pair consisting of `'ComparisonMethod'` and one of the following:

- `'auto'` — Check if the rows of `A` are sorted by `real(A)` when `A` is real, and check if the rows of `A` are sorted by `abs(A)` when `A` is complex.
- `'real'` — Check if the rows of `A` are sorted by `real(A)` when `A` is real or complex. If a column has elements with consecutive equal real parts, then check `imag(A)` to break ties.
- `'abs'` — Check if the rows of `A` are sorted by `abs(A)` when `A` is real or complex. If a column has elements with consecutive equal magnitude, then check `angle(A)` in the interval (-π,π] to break ties.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If `A` is complex with all zero imaginary parts, then MATLAB might convert `A` to `real(A)` before calling `issortedrows(A)`. In this case, MATLAB checks that the rows of `A` are sorted by `real(A)`, but the generated code checks that the rows of `A` are sorted by `abs(A)`. To make the generated code match MATLAB, use `issortedrows(real(A))` or `issortedrows(A,'ComparisonMethod','real')`. See “Code Generation for Complex Data with Zero-Valued Imaginary Parts” (MATLAB Coder).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`issorted` | `sort` | `sortrows`

Introduced in R2017a

isspace

Determine which characters are space characters

Syntax

```
TF = isspace(A)
```

Description

`TF = isspace(A)` returns a logical array `TF`. If `A` is a character array or string scalar, then the elements of `TF` are logical `1` (`true`) where corresponding characters in `A` are space characters, and logical `0` (`false`) elsewhere. `isspace` recognizes all Unicode whitespace characters.

If `A` is not a character array or string scalar, then `isspace` returns logical `0` (`false`).

Examples

Determine Which Character Array Elements Are Spaces

Create a character array, and then determine which elements are space characters.

```
chr = '123 Main St.'
```

```
chr =
'123 Main St.'
```

```
TF = isspace(chr)
```

```
TF = 1x12 logical array

   0   0   0   1   0   0   0   0   1   0   0   0
```

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. A can be any data type.

Tips

- To find space characters within elements of a nonscalar string array, use the `isstrprop` function.
- To find all characters for which `isspace` returns logical `1`, use the code below. Then look up the returned decimal codes in a Unicode reference, such as the List of Unicode characters.

```
find(isspace(char(1):char(intmax('uint16'))))
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Input values from the `char` class must be in the range 0–127.

See Also

`blanks` | `char` | `deblank` | `is*` | `isa` | `ischar` | `isletter` | `isnumeric` | `isstring` | `isstrprop` | `newline` | `regexp` | `string` | `strip` | `strtrim`

Topics

"Text in String and Character Arrays"

"Search and Replace Text"

Introduced before R2006a

issparse

Determine whether input is sparse

Syntax

```
TF = issparse(S)
```

Description

`TF = issparse(S)` returns logical 1 (`true`) if the storage class of `S` is sparse and logical 0 (`false`) otherwise.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`full` | `is*` | `sparse`

Introduced before R2006a

isstr

(Not recommended) Determine whether input is character array

Note `isstr` is not recommended. Use `ischar` instead.

See Also

`is*` | `isa` | `ischar`

Introduced before R2006a

isstring

Determine if input is string array

Syntax

```
tf = isstring(A)
```

Description

`tf = isstring(A)` returns `1` (`true`) if `A` is a string array. Otherwise, it returns `0` (`false`).

Examples

Determine if Array Contains String Values

Create different arrays, and then determine if they are string arrays.

Test a character vector.

```
chr = 'Mary Jones'
```

```
chr =
'Mary Jones'
```

```
tf = isstring(chr)
```

```
tf = logical
   0
```

Character vectors are not strings, so `isstring` returns `0`.

Test a string array. Starting in R2017a, you can create strings using double quotes.

```
str = ["Smith","Chung","Morales"; ...
       "Sanchez","Peterson","Adams"]
```

```
str = 2x3 string array
    "Smith"      "Chung"       "Morales"
    "Sanchez"    "Peterson"    "Adams"
```

```
tf = isstring(str)
```

```
tf = logical
   1
```

`str` is a string array, so `isstring` returns `1`.

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. `A` can be any data type.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cellstr` | `char` | `is*` | `isStringScalar` | `isa` | `iscategorical` | `iscell` | `iscellstr` | `ischar` | `isstrprop` | `isstruct` | `istable` | `string`

Topics

"Text in String and Character Arrays"
"Create String Arrays"

Introduced in R2016b

isStringScalar

Determine if input is string array with one element

Syntax

```
tf = isStringScalar(A)
```

Description

`tf = isStringScalar(A)` returns `1` (`true`) if `A` is a string array and `A` has only one element. Otherwise, it returns `0` (`false`).

Examples

Determine If Array Is String Scalar

Create different arrays, and then determine if they are string scalars.

Test a string scalar.

```
str = "Mercury"
```

```
str =
"Mercury"
```

```
tf = isStringScalar(str)
```

```
tf = logical
   1
```

Test a string array with multiple elements. Since `str` contains more than one element, `isStringScalar` returns `0`.

```
str = ["Mercury","Gemini","Apollo"]
```

```
str = 1x3 string array
    "Mercury"    "Gemini"    "Apollo"
```

```
tf = isStringScalar(str)
```

```
tf = logical
   0
```

Test a character vector. Since character vectors are not strings, `isStringScalar` returns `0`.

```
chr = 'Pluto'
```

```
chr =
'Pluto'
```

```
tf = isStringScalar(chr)
```

```
tf = logical
   0
```

Input Arguments

A — Input array
array of any size and data type

Input array, specified as an array of any size and data type.

See Also

`cellstr` | `char` | `is*` | `isa` | `iscellstr` | `ischar` | `ismissing` | `isstring` | `isstrprop` | `string`

Topics

"Text in String and Character Arrays"
"Create String Arrays"

Introduced in R2017b

isstrprop

Determine which characters in input strings are of specified category

Syntax

```
TF = isstrprop(str,category)
TF = isstrprop(str,category,'ForceCellOutput',tf)
```

Description

`TF = isstrprop(str,category)` determines if characters in the input text are of the specified category, such as letters, numbers, or whitespace. For example, `isstrprop('ABC123','alpha')` returns a 1-by-6 logical array, `[1 1 1 0 0 0]`, indicating that the first three characters are letters.

- If `str` is a character array, string scalar, or numeric array, then `isstrprop` returns a logical array.
- If `str` is a cell array of character vectors or a string array, then `isstrprop` returns a cell array of logical vectors.

`TF = isstrprop(str,category,'ForceCellOutput',tf)`, where `tf` is `1` (`true`), returns `TF` as a cell array even when `str` is a character array, string scalar, or numeric array. The default for `tf` is `0` (`false`).

Examples

Determine Which Characters Are Letters

Create a character vector and determine which characters are letters.

```
chr = '123 Maple Street'
```

```
chr =
'123 Maple Street'
```

```
TF = isstrprop(chr,'alpha')
```

```
TF = 1x16 logical array

   0   0   0   0   1   1   1   1   1   0   1   1   1   1   1   1
```

Find indices for the letters in `chr` using `TF`.

```
idx = find(TF)
```

```
idx = 1×11

     5     6     7     8     9    11    12    13    14    15    16
```

```
chr(idx)
```

```
ans =
'MapleStreet'
```

Determine Types of Characters in String Arrays

Create string arrays. Then determine which characters belong to various categories using the `isstrprop` function.

Create a string scalar and determine which of its characters are numeric digits. Starting in R2017a, you can create strings using double quotes.

```
str = "123 Maple Street"
```

```
str =
"123 Maple Street"
```

```
TF = isstrprop(str,'digit')
```

```
TF = 1x16 logical array

   1   1   1   0   0   0   0   0   0   0   0   0   0   0   0   0
```

Create a nonscalar string array. Determine which characters in each string are whitespace characters. `isstrprop` returns a cell array in which each cell contains results for a string in `str`.

```
str = ["123 Maple St.";"456 Oak St."]
```

```
str = 2x1 string array
    "123 Maple St."
    "456 Oak St."
```

```
TF = isstrprop(str,'wspace')
```

```
TF=2×1 cell
    {1x13 logical}
    {1x11 logical}
```

To display the results for the second string, `str(2)`, index into `TF{2}`.

```
TF{2}
```

```
ans = 1x11 logical array

   0   0   0   1   0   0   0   1   0   0   0
```

Determine Types of Characters in Cell Array

Create a cell array of character vectors. Determine which characters are whitespace characters.

```
C = {'123 Maple St.';'456 Oak St.'}
```

```
C = 2x1 cell array
    {'123 Maple St.'}
    {'456 Oak St.'  }
```

```
TF = isstrprop(C,'wspace')
```

```
TF=2×1 cell
    {1x13 logical}
```

```
    {1x11 logical}
```

Return Logical Vector in Cell Array

Find the punctuation characters in a character vector. `isstrprop` returns a logical vector indicating which characters belong to that category. Force `isstrprop` to return the logical vector in a cell array.

```
chr = 'A horse! A horse! My kingdom for a horse!'
```

```
chr =
'A horse! A horse! My kingdom for a horse!'
```

```
TF = isstrprop(chr,'punct','ForceCellOutput',true)
```

```
TF = 1x1 cell array
    {1x41 logical}
```

Find indices for the punctuation marks in `chr` using `TF{1}`.

```
find(TF{1})
```

```
ans = 1×3

     8    17    41
```

Determine Character Types in Numeric Array

Create a numeric array. Determine which numbers correspond to character codes for letters.

```
X = [77 65 84 76 65 66]
```

```
X = 1×6

    77    65    84    76    65    66
```

```
TF = isstrprop(X,'alpha')
```

```
TF = 1x6 logical array

   1   1   1   1   1   1
```

`isstrprop` identifies all the numbers as character codes for letters. Convert the numbers to their corresponding characters with the `char` function.

```
c = char(X)
```

```
c =
'MATLAB'
```

Input Arguments

`str` — Input array
string array | character array | cell array of character vectors | numeric array

Input array, specified as a string array, character array, cell array of character vectors, or numeric array.

If `str` is a numeric array, then `isstrprop` treats the numbers as Unicode character codes. If the numbers are double- or single-precision floating-point numbers, then `isstrprop` rounds them to the nearest integer values before interpreting them as character codes.

Data Types: `string` | `char` | `cell` | `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`category` — Character category
`'alpha'` | `'alphanum'` | `'cntrl'` | `'digit'` | `'graphic'` | ...

Character category, specified as a character vector or string scalar. `isstrprop` classifies the characters in `str` according to categories defined by the Unicode standard.

Category	Description
`alpha`	Letters.
`alphanum`	Letters or numeric digits.

Category	Description
`cntrl`	Control characters (for example, `char(0:20)`).
`digit`	Numeric digits.
`graphic`	Graphic characters. `isstrprop` treats all Unicode characters as graphic characters, except for the following: • Unassigned characters • Whitespace characters • The line separator • The paragraph separator • Control characters • Private user-defined characters • Surrogate characters
`lower`	Lowercase letters.
`print`	Graphic characters, plus `char(32)`.
`punct`	Punctuation characters.
`wspace`	Whitespace characters. This range includes the ANSI® C definition of whitespace, `{' ','\t','\n','\r','\v','\f'}`, in addition to some other Unicode characters.
`upper`	Uppercase letters.
`xdigit`	Valid hexadecimal digits.

`tf` — True or false
`0` (default) | `1`

True or false, specified as `1` or `0`.

Output Arguments

TF — True or false
logical array | cell array of logical vectors

True or false, returned as a logical array or cell array of logical vectors.

- If `str` is a character vector, string scalar, or numeric array, then `TF` is a logical array indicating which characters belong to the specified category.
- If `str` is a cell array of character vectors or a string array, then `TF` is a cell array. For each element of `str`, the corresponding cell of `TF` contains a logical vector indicating which characters in that element belong to the specified category.

Tips

Whitespace characters for which the `wspace` option returns `true` include tab, line feed, vertical tab, form feed, carriage return, and space, in addition to some other Unicode characters. To see all characters for which the `wspace` option returns `true`, enter the following command, and then look up the returned decimal codes in a Unicode reference:

```
find(isstrprop(char(1):char(intmax('uint16')),'wspace'))
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Input array must be specified as a character array or an array of integers.
- Input values must be in the range 0–127.

See Also

`char` | `find` | `is*` | `isa` | `iscellstr` | `ischar` | `isletter` | `isnumeric` | `isspace` | `isstring` | `isvarname` | `regexp` | `string`

Topics

"Search and Replace Text"
"Compare Text"

Introduced before R2006a

isstruct

Determine if input is structure array

Syntax

```
tf = isstruct(A)
```

Description

`tf = isstruct(A)` returns logical `1` (`true`) if `A` is a MATLAB structure and logical `0` (`false`) otherwise.

Examples

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];

isstruct(patient)

ans =

     1
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`is*` | `isa` | `iscell` | `ischar` | `isfield` | `islogical` | `isnumeric` | `isobject` | `istable` | `struct`

Topics

dynamic field names

Introduced before R2006a

isstudent

Determine if version is Student Version

Syntax

```
tf = isstudent
```

Description

`tf = isstudent` returns logical `1` (`true`) if the version of MATLAB is the Student Version. Otherwise, it returns logical `0` (`false`) for commercial versions.

See Also

`is*` | `license` | `ver` | `version`

Introduced before R2006a

issymmetric

Determine if matrix is symmetric or skew-symmetric

Syntax

```
tf = issymmetric(A)
tf = issymmetric(A,skewOption)
```

Description

`tf = issymmetric(A)` returns logical `1` (`true`) if square matrix `A` is symmetric on page 1-8277; otherwise, it returns logical `0` (`false`).

`tf = issymmetric(A,skewOption)` specifies the type of the test. Specify `skewOption` as `'skew'` to determine if `A` is skew-symmetric on page 1-8278.

Examples

Test if Hermitian Matrix Is Symmetric

Create a 3-by-3 matrix.

```
A = [1 0 1i; 0 1 0;-1i 0 1]
```

```
A = 3×3 complex

   1.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 1.0000i
   0.0000 + 0.0000i   1.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 - 1.0000i   0.0000 + 0.0000i   1.0000 + 0.0000i
```

The matrix is Hermitian and has a real-valued diagonal.

Test whether the matrix is symmetric.

```
tf = issymmetric(A)
```

```
tf = logical
   0
```

The result is logical `0` (`false`) because `A` is not symmetric. In this case, `A` is equal to its complex conjugate transpose, `A'`, but not its nonconjugate transpose, `A.'`.

Change the element in `A(3,1)` to be `1i`.

```
A(3,1) = 1i;
```

Determine whether the modified matrix is symmetric.

```
tf = issymmetric(A)
```

```
tf = logical
   1
```

The matrix, `A`, is now symmetric because it is equal to its nonconjugate transpose, `A.'`.

Test if Matrix Is Skew-Symmetric

Create a 4-by-4 matrix.

```
A = [0 1 -2 5; -1 0 3 -4; 2 -3 0 6; -5 4 -6 0]
```

```
A = 4×4

     0     1    -2     5
    -1     0     3    -4
     2    -3     0     6
    -5     4    -6     0
```

The matrix is real and has a diagonal of zeros.

Specify `skewOption` as `'skew'` to determine whether the matrix is skew-symmetric.

```
tf = issymmetric(A,'skew')
```

```
tf = logical
   1
```

The matrix, `A`, is skew-symmetric since it is equal to the negation of its nonconjugate transpose, `-A.'`.

Input Arguments

`A` — Input matrix
numeric matrix

Input matrix, specified as a numeric matrix. If `A` is not square, then `issymmetric` returns logical `0` (`false`).

Data Types: `single` | `double` | `logical`
Complex Number Support: Yes

`skewOption` — Test type
`'nonskew'` (default) | `'skew'`

Test type, specified as `'nonskew'` or `'skew'`. Specify `'skew'` to test whether `A` is skew-symmetric on page 1-8278.

More About

Symmetric Matrix

- A square matrix, `A`, is symmetric if it is equal to its nonconjugate transpose, `A = A.'`.

  In terms of the matrix elements, this means that

  $$a_{i,\ j} = a_{j,\ i}\ .$$

- Since real matrices are unaffected by complex conjugation, a real matrix that is symmetric is also Hermitian. For example, the matrix

  $$A = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

is both symmetric and Hermitian.

Skew-Symmetric Matrix

- A square matrix, `A`, is skew-symmetric if it is equal to the negation of its nonconjugate transpose, `A = -A.'`.

  In terms of the matrix elements, this means that

  $$a_{i,\ j} = -a_{j,\ i}\ .$$

- Since real matrices are unaffected by complex conjugation, a real matrix that is skew-symmetric is also skew-Hermitian. For example, the matrix

  $$A = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

  is both skew-symmetric and skew-Hermitian.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`ctranspose` | `ishermitian` | `isreal` | `transpose`

Introduced in R2014a

isTiled

Determine if image is tiled

Syntax

```
tf = isTiled(t)
```

Description

`tf = isTiled(t)` returns `true` if the image has a tiled layout. Otherwise the function returns `false`.

Examples

Determine if Image Has Tiled Organization

Create a `Tiff` object for a TIFF file and check if the image in the file has a tiled organization.

```
t = Tiff('example.tif','r');
tf = isTiled(t)
```

```
tf = logical
   1
```

The image has a tiled organization.

Close the `Tiff` object.

```
close(t)
```

Input Arguments

t — Tiff object
`Tiff` object

`Tiff` object representing a TIFF file. Use the `Tiff` function to create the object.

Algorithms

References

This function corresponds to the `TIFFIsTiled` function in the LibTIFF C API. To use this function, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff`

Topics

"Importing Images"
"Exporting to Images"

Introduced in R2009b

istable

Determine whether input is table

Syntax

```
TF = istable(T)
```

Description

`TF = istable(T)` returns logical `1` (`true`) if `T` is a table, and logical `0` (`false`) otherwise.

Examples

Determine if Workspace Variable Is Table

Create a workspace variable, `T`.

```
T = table(categorical({'M';'F';'M'}),[45 45;41 32;40 34],...
    {'NY';'CA';'MA'},[true;false;false])
```

```
T=3×4 table
    Var1      Var2       Var3     Var4
    ____    ________    ______    _____

     M      45    45    {'NY'}    true
     F      41    32    {'CA'}    false
     M      40    34    {'MA'}    false
```

Verify that the workspace variable, `T`, is a table.

```
istable(T)
```

```
ans = logical
   1
```

T is a table.

Determine if Subset of Table Is Table

Create a table, T.

```
T = table(categorical({'M';'F';'M'}),[45 45;41 32;40 34],...
    {'NY';'CA';'MA'},[true;false;false])
```

```
T=3×4 table
    Var1      Var2       Var3     Var4
    ____    ________    ______    _____

     M      45    45    {'NY'}    true
     F      41    32    {'CA'}    false
     M      40    34    {'MA'}    false
```

Determine if the subset of table T that contains only the second and fourth variables is a table.

```
istable(T{:,[2 4]})
```

```
ans = logical
   0
```

Conversely, accessing data with curly braces, `T{:,[2 4]}`, returns a matrix and not a table.

Input Arguments

T — Input variable
workspace variable

Input variable, specified as a workspace variable. T can be any data type.

Extended Capabilities

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`iscell` | `islogical` | `isnumeric` | `isobject` | `isstruct` | `table`

Introduced in R2013b

istall

Determine if input is tall array

Syntax

```
tf = istall(X)
```

Description

`tf = istall(X)` returns logical `1` (`true`) if `X` is a tall array. Otherwise, it returns logical `0` (`false`). The output, `tf`, is an in-memory logical scalar.

Examples

Determine if Array is Tall

Create an array and determine if it is a tall array.

```
X = tall(randn(1000,3))
```

```
X =

  1,000x3 tall double matrix

    0.5377    0.6737    0.2962
    1.8339   -0.6691    1.2008
   -2.2588   -0.4003    1.0902
    0.8622   -0.6718   -0.3587
    0.3188    0.5756   -0.1299
   -1.3077   -0.7781    0.7337
   -0.4336   -1.0636    0.1203
    0.3426    0.5530    1.1363
      :         :         :
      :         :         :
```

```
tf = istall(X)
```

```
tf = logical
   1
```

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. X can be any data type.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

See Also
`classUnderlying` | `isaUnderlying` | `tall`

Topics
"Tall Arrays for Out-of-Memory Data"

Introduced in R2016b

istimetable

Determine if input is timetable

Syntax

```
TF = istimetable(A)
```

Description

`TF = istimetable(A)` returns logical `1` (`true`) if `A` is a timetable, and logical `0` (`false`) otherwise.

Examples

Determine If Workspace Variable is Timetable

Create a timetable and verify that it is a timetable.

```
Date = datetime({'2015-12-18';'2015-12-19';'2015-12-20'});
Temp = [37.3;39.1;42.3];
Pressure = [29.4;29.6;30.0];
Precip = [0.1;0.9;0.0];
TT = timetable(Date,Temp,Pressure,Precip)
```

```
TT=3×4 timetable
       Date        Temp    Pressure    Precip
    ___________    ____    ________    ______

    18-Dec-2015    37.3      29.4       0.1
    19-Dec-2015    39.1      29.6       0.9
    20-Dec-2015    42.3        30         0
```

```
TF = istimetable(TT)
```

```
TF = logical
   1
```

Convert TT to a table, T. The `istimetable` function returns `0` because T is not a timetable.

```
T = timetable2table(TT);
TF = istimetable(T)
```

```
TF = logical
   0
```

Input Arguments

A — Input variable
workspace variable

Input variable, specified as a workspace variable. A can be any data type.

Extended Capabilities

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`iscell` | `islogical` | `isnumeric` | `isobject` | `isstruct` | `istable` | `timetable`

Topics

“Create Timetables”

"Tables"
"Represent Dates and Times in MATLAB"

Introduced in R2016b

istril

Determine if matrix is lower triangular

Syntax

```
tf = istril(A)
```

Description

`tf = istril(A)` returns logical `1` (`true`) if `A` is a lower triangular matrix on page 1-8291; otherwise, it returns logical `0` (`false`).

Examples

Test Lower Triangular Matrix

Create a 5-by-5 matrix.

```
D = tril(magic(5))
```

```
D = 5×5

    17     0     0     0     0
    23     5     0     0     0
     4     6    13     0     0
    10    12    19    21     0
    11    18    25     2     9
```

Test `D` to see if it is lower triangular.

```
istril(D)
```

```
ans = logical
   1
```

The result is logical `1` (`true`) because all elements above the main diagonal are zero.

Test Matrix of Zeros

Create a 5-by-5 matrix of zeros.

```
Z = zeros(5);
```

Test `Z` to see if it is lower triangular.

```
istril(Z)
```

```
ans = logical
   1
```

The result is logical `1` (`true`) because a lower triangular matrix can have any number of zeros on its main diagonal.

Input Arguments

`A` — Input array
numeric array

Input array, specified as a numeric array. `istril` returns logical `0` (`false`) if `A` has more than two dimensions.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Lower Triangular Matrix

A matrix is lower triangular if all elements above the main diagonal are zero. Any number of the elements on the main diagonal can also be zero.

For example, the matrix

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ -2 & -2 & 1 & 0 \\ -3 & -3 & -3 & 1 \end{pmatrix}$$

is lower triangular. A diagonal matrix is both upper and lower triangular.

Tips

- Use the `tril` function to produce lower triangular matrices for which `istril` returns logical `1` (`true`).
- The functions `isdiag`, `istriu`, and `istril` are special cases of the function `isbanded`, which can perform all of the same tests with suitably defined upper and lower bandwidths. For example, `istril(A) == isbanded(A,size(A,1),0)`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bandwidth` | `diag` | `isbanded` | `isdiag` | `istriu` | `tril` | `triu`

Introduced in R2014a

istriu

Determine if matrix is upper triangular

Syntax

```
tf = istriu(A)
```

Description

`tf = istriu(A)` returns logical `1` (`true`) if `A` is an upper triangular matrix on page 1-8295; otherwise, it returns logical `0` (`false`).

Examples

Test Upper Triangular Matrix

Create a 5-by-5 matrix.

```
A = triu(magic(5))
```

```
A = 5×5

    17    24     1     8    15
     0     5     7    14    16
     0     0    13    20    22
     0     0     0    21     3
     0     0     0     0     9
```

Test `A` to see if it is upper triangular.

```
istriu(A)
```

```
ans = logical
   1
```

The result is logical `1` (`true`) because all elements below the main diagonal are zero.

Test Matrix of Zeros

Create a 5-by-5 matrix of zeros.

```
Z = zeros(5);
```

Test `Z` to see if it is upper triangular.

```
istriu(Z)
```

```
ans = logical
   1
```

The result is logical `1` (`true`) because an upper triangular matrix can have any number of zeros on the main diagonal.

Input Arguments

`A` — Input array
numeric array

Input array, specified as a numeric array. `istriu` returns logical `0` (`false`) if `A` has more than two dimensions.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Upper Triangular Matrix

A matrix is upper triangular if all elements below the main diagonal are zero. Any number of the elements on the main diagonal can also be zero.

For example, the matrix

$$A = \begin{pmatrix} 1 & -1 & -1 & -1 \\ 0 & 1 & -2 & -2 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

is upper triangular. A diagonal matrix is both upper and lower triangular.

Tips

- Use the `triu` function to produce upper triangular matrices for which `istriu` returns logical `1` (`true`).
- The functions `isdiag`, `istriu`, and `istril` are special cases of the function `isbanded`, which can perform all of the same tests with suitably defined upper and lower bandwidths. For example, `istriu(A) == isbanded(A,0,size(A,2))`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bandwidth` | `diag` | `isbanded` | `isdiag` | `istril` | `tril` | `triu`

Introduced in R2014a

isundefined

Find undefined elements in categorical array

Syntax

```
TF = isundefined(A)
```

Description

`TF = isundefined(A)` returns a logical array, `TF`, that indicates which elements in the categorical array, `A`, contain undefined values. `isundefined` returns logical `1` (`true`) for undefined elements; otherwise it returns logical `0` (`false`). The size of `TF` is the same as the size of `A`.

Any elements in `A` without a corresponding category are undefined. Undefined values are similar to `NaN` in numeric arrays.

Examples

Categorical Array with Undefined Values

Create a categorical array, `A`, from numeric values where `1`, `2`, and `3` represent red, green, and blue respectively.

```
A = categorical([4 1; 2 3; 2 1; 3 4; 1 1],1:3,{'red','green','blue'})
```

```
A = 5x2 categorical array
     <undefined>      red
     green            blue
     green            red
     blue             <undefined>
     red              red
```

A is a 5-by-2 categorical array with three categories: red, green, and blue. Array elements corresponding to the numeric value 4 in the input array to the categorical function do not have a corresponding category. Therefore, they are undefined in the output categorical array, A.

Find undefined elements in A.

```
TF = isundefined(A)
```

```
TF = 5x2 logical array

   1   0
   0   0
   0   0
   0   1
   0   0
```

A(1,1) and A(4,2) are undefined.

Input Arguments

A — Categorical array
vector | matrix | multidimensional array

Categorical array, specified as a vector, matrix, or multidimensional array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This function fully supports code generation. For more information, see “Code Generation for Categorical Arrays” (MATLAB Coder).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`exist` | `ismember`

Introduced in R2013b

isunix

Determine if version is for Linux or Mac platforms

Syntax

```
tf = isunix
```

Description

`tf = isunix` returns logical `1` (`true`) if the version of MATLAB is for Linux or Apple macOS platforms. Otherwise, it returns logical `0` (`false`).

Examples

Execute MATLAB Command Based on Platform

```
if ismac
    % Code to run on Mac platform
elseif isunix
    % Code to run on Linux platform
elseif ispc
    % Code to run on Windows platform
else
    disp('Platform not supported')
end
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Returns true or false based on the MATLAB version used for code generation.
- Use only when the code generation target is S-function (Simulation) or MEX-function.

See Also

`computer` | `is*` | `ismac` | `ispc` | `isstudent`

Introduced before R2006a

isvarname

Determine if input is valid variable name

Syntax

```
tf = isvarname(s)
isvarname s
```

Description

`tf = isvarname(s)` determines if input `s` is a valid variable name. If `s` is a valid MATLAB variable name the `isvarname` function returns logical `1` (`true`). Otherwise it returns logical `0` (`false`).

A valid variable name begins with a letter and contains not more than `namelengthmax` characters. Valid variable names can include letters, digits, and underscores. MATLAB keywords are not valid variable names. To determine if the input is a MATLAB keyword, use the `iskeyword` function.

`isvarname s` is the command form of the syntax. The command form requires fewer special characters. You do not need parentheses or single quotes around the input.

Examples

Determine Variable Name Validity

Show that input that starts with a number is not a valid variable name.

```
s = '8th_column';
isvarname(s)
```

```
ans = logical
   0
```

Change the input to a valid variable name.

```
s = 'column_8';
isvarname(s)
```

```
ans = logical
   1
```

Use the command form to determine if the input is a valid variable name.

```
isvarname column_8
```

```
ans = logical
   1
```

Input Arguments

s — Potential variable name
character vector | string

Potential variable name, specified as a character vector, or string.

Example: `'myVar'`

Example: `"myVar2"`

See Also

`is*` | `iskeyword` | `matlab.lang.makeUniqueStrings` | `matlab.lang.makeValidName` | `namelengthmax`

Topics

"Variable Names"

Introduced before R2006a

isvector

Determine whether input is vector

Syntax

```
TF = isvector(A)
```

Description

`TF = isvector(A)` returns logical `1` (`true`) if `A` is a vector. Otherwise, it returns logical `0` (`false`). A vector is a two-dimensional array that has a size of 1-by-N or N-by-1, where N is a nonnegative integer.

Examples

Determine Vector from Matrix

Create a 2-by-2 matrix. Determine whether it is a vector.

```
A = [1 2; 3 4];
TF = isvector(A)
```

```
TF = logical
   0
```

Check whether the first column of the matrix is a vector.

```
TF = isvector(A(:,1))
```

```
TF = logical
   1
```

Check whether the first row of the matrix is a vector.

```
TF = isvector(A(1,:))
```

```
TF = logical
   1
```

Determine Vector from Scalar

Create a scalar, which is a 1-by-1 array.

```
A = 5;
```

Determine whether the scalar `A` is also a vector.

```
TF = isvector(A)
```

```
TF = logical
   1
```

Character Vector and String Scalar

Create an array of characters. Determine whether it is a vector.

```
A = 'Hello, World!';
TF = isvector(A)
```

```
TF = logical
   1
```

Check the dimension of `A` using `size`. `A` is a 1-by-13 character vector.

```
sz = size(A)
```

```
sz = 1×2

     1    13
```

Now create a string scalar by enclosing a piece of text in double quotes.

```
A = "Hello, World!";
```

Check whether the scalar A is also a vector.

```
TF = isvector(A)
```

```
TF = logical
   1
```

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Algorithms

- If the input array A has more than two dimensions, then `isvector(A)` always returns logical `0` (`false`). For example, an array of size 1-by-1-by-N is not a vector.
- `isvector(A)` function does not have any special behavior for dimension lengths equal to 0. For example, `isvector(A)` returns logical `1` (`true`) if the size of A is 0-by-1. But, `isvector(A)` returns logical `0` (`false`) if the size of A is 0-by-3.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`iscolumn` | `ismatrix` | `isrow` | `isscalar` | `size`

Introduced before R2006a

isweekend

Determine weekend elements

Syntax

```
tf = isweekend(t)
```

Description

`tf = isweekend(t)` returns an array the same size as `t` containing logical 1 (`true`) where the corresponding element of `t` is a datetime that occurs on a weekend day, and logical 0 (`false`) otherwise. Weekend days are Saturday and Sunday.

Examples

Determine If Date Occurs During Weekend

```
t = datetime(2014,5,2:5,'Format','eee dd-MMM-yyyy')
```

```
t = 1x4 datetime array
   Fri 02-May-2014   Sat 03-May-2014   Sun 04-May-2014   Mon 05-May-2014
```

```
tf = isweekend(t)
```

```
tf = 1x4 logical array

   0   1   1   0
```

May 3 and May 4, 2014 are days that fall on a weekend.

Input Arguments

t — Input date and time
`datetime` array

Input date and time, specified as a `datetime` array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`day` | `isdst`

Introduced in R2014b

j

Imaginary unit

Syntax

```
1j
z = a + bj
z = x + 1j*y
```

Description

`1j` returns the basic imaginary unit. `j` is equivalent to `sqrt(-1)`.

You can use `j` to enter complex numbers. You also can use the character `i` as the imaginary unit. To create a complex number without using `i` and `j`, use the `complex` function.

`z = a + bj` returns a complex numerical constant, `z`.

`z = x + 1j*y` returns a complex array, `z`.

Examples

Complex Scalar

Create a complex scalar and use the character, `j`, without a multiplication sign as a suffix in forming a complex numerical constant.

```
z = 1+2j
```

```
z = 1.0000 + 2.0000i
```

Complex Vector

Create a complex vector from two 4-by-1 vectors of real numbers. `z` is a 4-by-1 complex vector.

```
x = [1:4]';
y = [8:-2:2]';

z = x+1j*y
```

```
z = 4×1 complex

   1.0000 + 8.0000i
   2.0000 + 6.0000i
   3.0000 + 4.0000i
   4.0000 + 2.0000i
```

Complex Exponential

Create a complex scalar representing a complex vector with radius, `r`, and angle from the origin, `theta`.

```
r = 4;
theta = pi/4;

z = r*exp(1j*theta)
```

```
z = 2.8284 + 2.8284i
```

Input Arguments

a — Real component of complex scalar
scalar

Real component of a complex scalar, specified as a scalar.

Data Types: `single` | `double`

b — Imaginary component of complex scalar
scalar

Imaginary component of a complex scalar, specified as a scalar.

If `b` is `double`, you can use the character, `j`, without a multiplication sign as a suffix in forming the complex numerical constant.

Example: `7j`

If `b` is `single`, you must use a multiplication sign when forming the complex numerical constant.

Example: `single(7)*j`

Data Types: `single` | `double`

x — Real component of complex array
scalar | vector | matrix | multidimensional array

Real component of a complex array, specified as a scalar, vector, matrix, or multidimensional array.

The size of `x` must match the size of `y`, unless one is a scalar. If either `x` or `y` is a scalar, MATLAB expands the scalar to match the size of the other input.

`single` can combine with `double`.

Data Types: `single` | `double`

y — Imaginary component of complex array
scalar | vector | matrix | multidimensional array

Imaginary component of a complex array, specified as a scalar, vector, matrix, or multidimensional array.

The size of `x` must match the size of `y`, unless one is a scalar. If either `x` or `y` is a scalar, MATLAB expands the scalar to match the size of the other input.

`single` can combine with `double`.

Data Types: `single` | `double`

Output Arguments

z — Complex array
scalar | vector | matrix | multidimensional array

Complex array, returned as a scalar, vector, matrix, or multidimensional array.

The size of `z` is the same as the input arguments.

`z` is `single` if at least one input argument is `single`. Otherwise, `z` is `double`.

Tips

- For speed and improved robustness in complex arithmetic, use `1i` and `1j` instead of `i` and `j`.
- Since `j` is a function, it can be overridden and used as a variable. However, it is best to avoid using `i` and `j` for variable names if you intend to use them in complex arithmetic.
- Use the `complex` function to create a complex output in the following cases:
  - When the names `i` and `j` might be used for other variables (and do not equal `sqrt(-1)`)
  - When the inputs are not `double` or `single`
  - When the imaginary component is all zeros

See Also

`complex` | `conj` | `i` | `imag` | `real`

Topics

"Create Complex Numbers"

Introduced before R2006a

jet

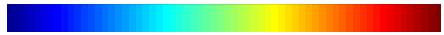
Jet colormap array

Syntax

```
c = jet
c = jet(m)
```

Description

`c = jet` returns the jet colormap as a three-column array with the same number of rows as the colormap for the current figure. If no figure exists, then the number of rows is equal to the default length of 256. Each row in the array contains the red, green, and blue intensities for a specific color. The intensities are in the range [0,1], and the color scheme looks like this image.



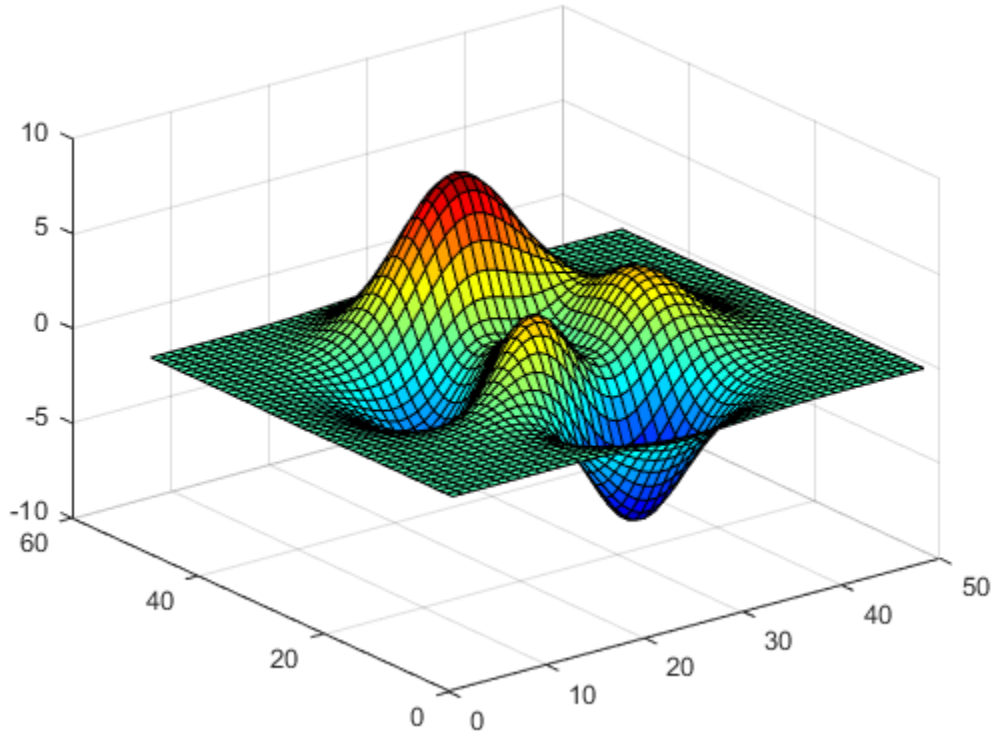
`c = jet(m)` returns the colormap with `m` colors.

Examples

Reverse the Colormap

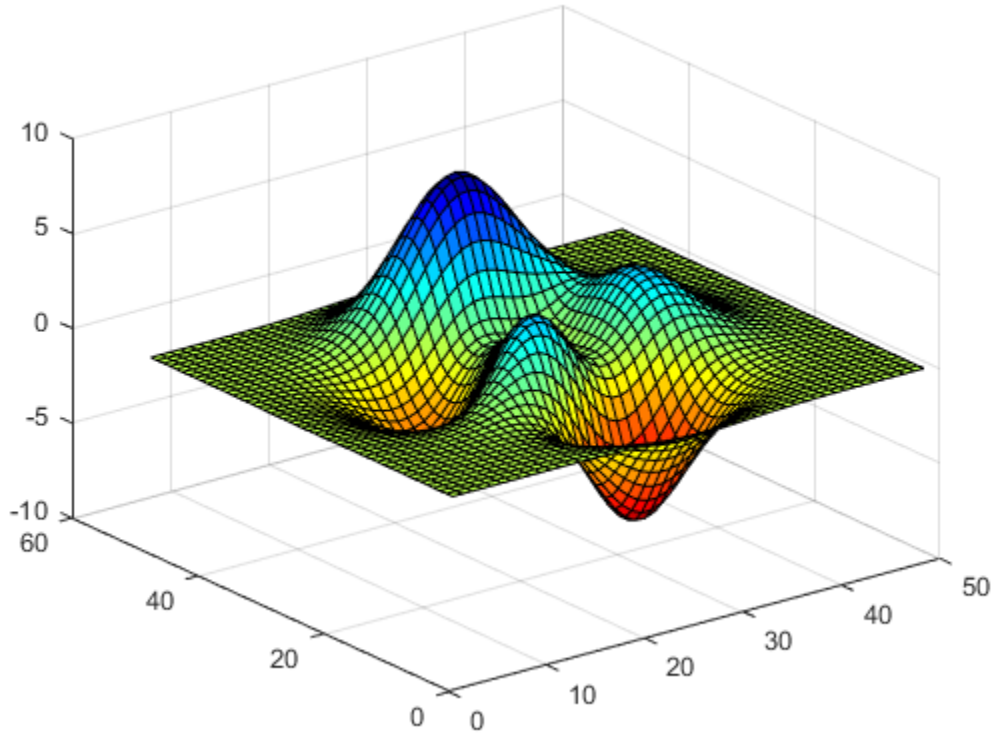
Plot a surface and assign the jet colormap.

```
surf(peaks);
colormap('jet');
```



Get the jet colormap array and reverse the order. Then apply the modified colormap to the surface.

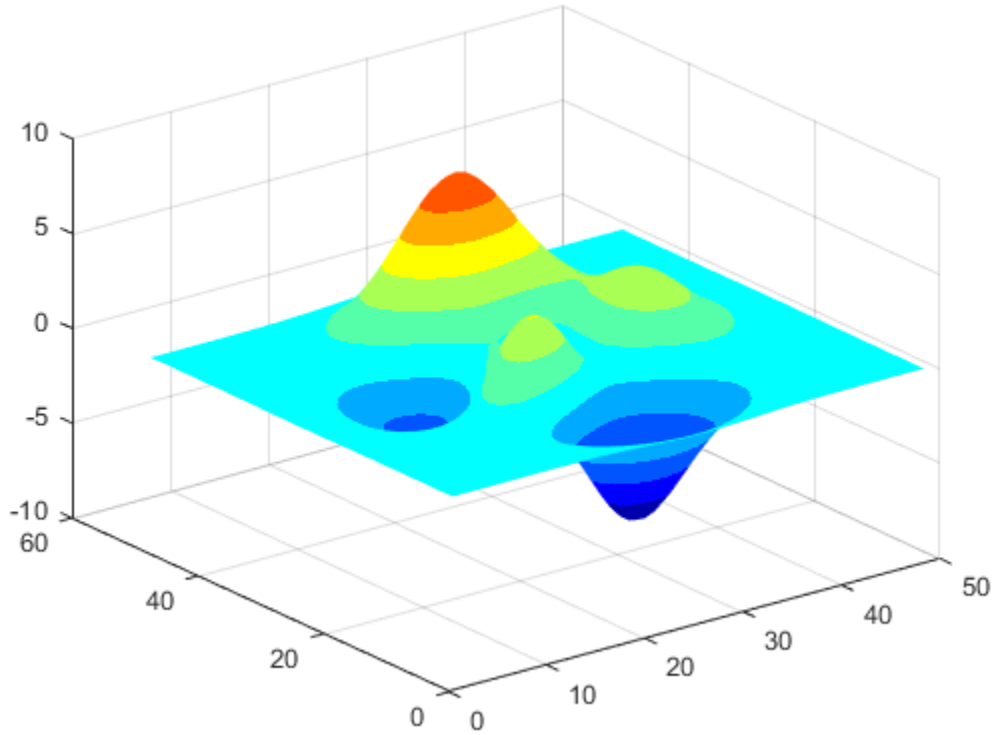
```
c = jet;
c = flipud(c);
colormap(c);
```



Downsample the Jet Colormap

Get a downsampled version of the jet colormap containing only ten colors. Then display the contours of the peaks function by applying the colormap and interpolated shading.

```
c = jet(10);
surf(peaks);
colormap(c);
shading interp;
```



Input Arguments

m — Number of colors
256 (default) | scalar integer value

Number of colors, specified as a scalar integer value. The default value of `m` is equal to the length of the colormap for the current figure. If no figure exists, the default value is 256.

Data Types: `single` | `double`

Compatibility Considerations

Colormaps Have 256 Colors by Default

Behavior changed in R2019b

Starting in R2019b, colormaps have 256 colors by default.

In R2019a and previous releases, the default size is 64. If you have code that depends on a colormap having 64 colors, specify the number of colors when you set the colormap for the figure, axes, or chart. For example, `colormap(jet(64))` sets the figure's colormap to the 64-color `jet` colormap.

Alternatively, you can change the default colormap for all figures within your MATLAB session:

```
set(groot,'defaultFigureColormap',jet(64))
```

See Also

`colormap`

Topics

"Change Color Scheme Using a Colormap"

Introduced before R2006a

jsondecode

Decode JSON-formatted text

Syntax

```
value = jsondecode(txt)
```

Description

`value = jsondecode(txt)` parses JSON text.

Examples

Decode JSON Text

Display the JSON-formatted string `["one", "two", "three"]`.

```
jsondecode('["one", "two", "three"]')
```

```
ans = 3x1 cell array
    {'one'  }
    {'two'  }
    {'three'}
```

Input Arguments

`txt` — JSON-formatted text
character vector

JSON-formatted text, specified as a character vector. Invalid names in the JSON text are made valid with `matlab.lang.makeValidName`.

Example: `'{"IDs":[116,943,234,38793]}'`

Data Types: `char`

Output Arguments

`value` — MATLAB data
any MATLAB data type

MATLAB data returned as decoded JSON-formatted text. `value` depends on the data encoded in `txt`.

Limitations

- If you decode, then encode a value, MATLAB does not guarantee that the result is identical to the original string. In particular, field names in JSON objects that are not valid MATLAB identifiers might be altered by the `makeValidName` function.

Algorithms

JSON supports fewer data types than MATLAB. `jsondecode` converts JSON data types to the MATLAB data types in this table. `jsondecode` converts JSON object field names to MATLAB structure field names.

JSON Data Type	MATLAB Data Type
null, in numeric arrays	NaN
null, in nonnumeric arrays	Empty `double []`
Boolean	Scalar `logical`
Number	Scalar `double`
String	Character vector
Object (In JSON, *object* means an unordered set of name-value pairs.)	Scalar structure (Names are made valid using `matlab.lang.makeValidName`.)
Array, when elements are of different data types	Cell array

JSON Data Type	MATLAB Data Type
Array of boolean	Array of `logical`
Array of numbers	Array of `double`
Array of strings	Cell array of character vectors
Array of objects — Same field names	Structure array
Array of objects — Different field names	Cell array of scalar structures

See Also

`fileread` | `jsonencode` | `matlab.lang.makeValidName` | `webread`

Introduced in R2016b

jsonencode

Create JSON-formatted text from structured MATLAB data

Syntax

```
txt = jsonencode(data)
txt = jsonencode(data,'ConvertInfAndNaN',TF)
```

Description

`txt = jsonencode(data)` encodes `data` and returns a character vector in JSON format.

`txt = jsonencode(data,'ConvertInfAndNaN',TF)` customizes the encoding of special floating point values NaN, Inf, -Inf.

Examples

Convert Cell Array of Text to JSON

```
value = {'one'; 'two'; 'three'};
jsonencode(value)
```

```
ans =
'["one","two","three"]'
```

Input Arguments

data — MATLAB data
any supported MATLAB data type

MATLAB data, specified as any supported MATLAB data type. For more information, see “Limitations” on page 1-8324.

Example: `s.IDs = [116, 943, 234, 38793]`

TF — Custom encoding
`true` (default) | `false`

Customize the encoding of special floating point values NaN, Inf, and -Inf, specified as `true` or `false`. A `true` value encodes floating point values as `null`. A `false` value encodes the values as literal `NaN`, `Infinity`, or `-Infinity`.

Example: `jsonencode(-Inf,'ConvertInfAndNaN',false)`

Limitations

- `jsonencode` does not support complex numbers, sparse arrays, or MATLAB enumerations. Objects must have public properties encoded as name-value pairs with `get` methods defined on the object properties.
- `jsonencode` does not support recursive structures such as graphics objects that contain references to parent and child objects.
- If you encode, then decode a value, MATLAB does not guarantee that the data type is preserved. JSON supports fewer data types than MATLAB, which results in loss of type information. For example, JSON data does not distinguish between `double` and `int32`. If you encode an `int32` value and then call `jsondecode`, the decoded value is type `double`.
- MATLAB does not guarantee that the shape of an array is preserved. For example, a 1-by-N numeric vector is encoded as an array. If you call `jsondecode`, then MATLAB decodes the array as an N-by-1 vector.

Tips

- To preserve the newline escape character `\n`, use the `newline` function.

  ```
  jsonencode(['one' newline 'two'])
  ```

  ```
  ans = '"one\ntwo"'
  ```
- To preserve other `\` escape characters, consider calling `sprintf` on the input. Test your input to see if `sprintf` creates the desired result.

```
jsonencode(sprintf('AB\tCD'))
```

```
ans = '"AB\tCD"'
```

- If the input contains a double quote character `"`, then the function inserts the `\` escape character.

```
jsonencode('one"two')
```

```
ans = '"one\"two"'
```

Algorithms

JSON supports fewer data types than MATLAB. `jsonencode` converts MATLAB data types to the JSON data types listed here.

MATLAB Data Type	JSON Data Type	Example	Output
array, empty	Array, empty	`jsonencode([])` `jsonencode(string.empty)`	`'[]'`
logical scalar	Boolean	`jsonencode(true)`	`'true'`
logical vector	Array of boolean	`jsonencode([true,false,false])`	`'[true,false,false]'`
logical array	Nested array of boolean	`jsonencode(logical([0,1,0;1,1,0]))`	`'[[false,true,false],[true,true,`
character vector	String	`jsonencode('This is a char.')`	`'"This is a char."'`
character array	Array of strings	`jsonencode(['AC';'EG'])`	`'["AC","EG"]'`
string scalar	String	`jsonencode("This is a string.")`	`'"This is a string."'`
string vector	Array of strings	`jsonencode(["AC";"EG"])`	`'["AC","EG"]'`

MATLAB Data Type	JSON Data Type	Example	Output
string array	Nested array of strings	`jsonencode(["AC","EG";"BD","FH"])`	`'[["AC","EG"],["BD","FH"]]'`
empty character vector	String	`jsonencode('')`	`'""'`
`<missing>`	null	`jsonencode(string(nan))`	`'null'`
numeric scalar	Number	`jsonencode(2.5)`	`'2.5'`
numeric vector	Array of numbers	`jsonencode(1:3)`	`'[1,2,3]'`
numeric array	Nested array of numbers	`jsonencode(eye(2))`	`'[[1,0],[0,1]]'`
complex numbers	Not supported		

table	Array of objects	`Name = {'Jones';'Brown'};`
		`Age = [40;49];`
		`jsonencode(table(Name,Age))`
			`'[{"Name":"Jones","Age":40},{"Name":"B`

cell scalar	Array of 1 element	`jsonencode({5})`	`'[5]'`
cell vector	Array	`jsonencode({'a',true,[2;3]})`	`'["a",true,[2,3]]'`
cell array	Array flattened to a single dimension	`jsonencode({1 2;3 4})`	`'[1,3,2,4]'`

MATLAB Data Type	JSON Data Type	Example	Output
structure scalar object scalar	Object Object (Public properties encoded as `name-value` pairs.)	`jsonencode(struct('a','value'))`	`'{"a":"value"}'`
structure vector object vector	Array of objects	`jsonencode(struct('a',{true,false}))`	`'[{"a":true},{"a":false}]'`
structure array object array	Nested array of objects		
datetime scalar	String (`string` method used to convert date and time to string format.)	`jsonencode(datetime('tomorrow'))`	`'"04-Nov-2016"'`

datetime vector	Array of strings	`DT = datetime({'8 April 2015','9 May 2015'},...`
		`'InputFormat','d MMMM yyyy');`
		`jsonencode(DT)`
			`'["08-Apr-2015","09-May-2015"]'`

datetime array	Nested array of strings	`DT = datetime(...`
		`[{'April 2015','May 2015'},{'June 2015','July 2015'}],...`
		`'InputFormat','MMMM yyyy');`
		`jsonencode(DT)`
			`ans = '[["01-Apr-2015","01-May-2015"],`
			`["01-Jun-2015","01-Jul-2015"]]'`

MATLAB Data Type	JSON Data Type	Example	Output
categorical scalar	String (`string` method used to create string format.)	`jsonencode(categorical({'r'}))`	`ans = '"r"'`
categorical vector	Array of strings	`jsonencode(categorical({'r';'g';'b'}))`	`'["r","g","b"]'`
categorical array	Nested array of strings	`jsonencode(categorical( ... {'r' 'b' 'g'; ... 'g' 'r' 'b'; ... 'b' 'r' 'g'}))`	`'[["r","b","g"],["g","r","b"],["b","r"`
containers.Map	Object	`jsonencode(containers.Map( ... {'Jan','Feb','Mar'}, ... [327,368,197]))`	`'{"Feb":368,"Jan":327,"Mar":197}'`
NaN Inf	null	`jsonencode([1,2,NaN,3,Inf])`	`'[1,2,null,3,null]'`

To pass a scalar MATLAB object as a scalar JSON array (enclosed in `[]` characters), convert the object using the cell array construction operator `{}`. For example, the following code converts the value of the `features` field into a scalar JSON array.

```
S = struct("features", struct("type", "Feature", "geometry",...
    struct("type", "point", "coordinates", [-105, 40])));
S.features = {S.features};
s = jsonencode(S)
```

```
s = '{"features":[{"type":"Feature","geometry":{"type":"point","coordinates":[-105,40]}}]}'
```

Compatibility Considerations

`jsonencode` Function: Encode NaN and Inf as null

Behavior changed in R2017b

The `jsonencode` function encodes MATLAB NaN and Inf values in JSON as null.

In R2017a and earlier, the function encodes NaN as `'NaN'` and Inf as `'Inf'`. To continue encoding NaN as `'NaN'` or Inf as `'Inf'`, call `jsonencode` with the `'ConvertInfAndNaN'` option set to `false`. For example,

```
jsonencode(NaN,'ConvertInfAndNaN',false)
```

```
ans =

    'NaN'
```

See Also

`jsondecode` | `webwrite`

Introduced in R2016b

javaaddpath

Add entries to dynamic Java class path

Syntax

```
javaaddpath(dpath)
javaaddpath(dpath,'-end')
```

Description

`javaaddpath(dpath)` adds one or more folders or Java Archive (JAR) files to the beginning of the current dynamic class path. Use the dynamic path when developing and debugging your own Java classes.

The `javaaddpath` command clears the definitions of all Java classes defined by files on the dynamic class path.

`javaaddpath(dpath,'-end')` adds files or folders to the end of the path.

Examples

Add Folder to Dynamic Class Path

Display the current dynamic path. The output reflects your configuration.

```
javaclasspath('-dynamic')
```

```
        DYNAMIC JAVA PATH

    <empty>
```

Add the current folder.

```
javaaddpath(pwd)
```

Display the updated dynamic path. The output reflects your current folder.

```
p = javaclasspath
```

```
p = 
    'c:\work\Java'
```

Append URL to Dynamic Class Path

```
javaaddpath('http://www.example.com','-end')
p = javaclasspath
```

```
p = 
    'c:\work\Java'
    'http://www.example.com'
```

Input Arguments

dpath — Folder or JAR file
string | array of strings | character vector | cell array of character vectors

Folder or JAR file, specified as a string, an array of strings, a character vector, or a cell array of character vectors to add to the dynamic path. When you add a folder to the path, MATLAB includes all files in that folder as part of the path.

Data Types: `char` | `cell`

Limitations

- MATLAB does not support JAR files generated by the MATLAB Compiler SDK product.

Tips

- Use the dynamic path while you develop your own Java classes. After you develop and debug a Java class, add the class to the static path. For more information, see `javaclasspath`.
- To add folders to the static path, which MATLAB loads at startup, create a `javaclasspath.txt` file, as described in “Static Path”.

- If `javaaddpath` displays the message `Not clearing Java`, the dynamic path is updated, but you might need to type `clear java` so that MATLAB detects the new Java classes.

See Also

`clear` | `javaclasspath` | `javarmpath`

Topics

"Java Class Path"

Introduced before R2006a

javaArray

Construct Java array object

Syntax

```
ObjArr = javaArray(PackageName.ClassName,x1,...,xN)
```

Description

`ObjArr = javaArray(PackageName.ClassName,x1,...,xN)` constructs an empty Java array object on page 1-8335 for objects of the specified `PackageName.ClassName` class. The array created by `javaArray` is equivalent to the array created by the following Java code.

```
A = new PackageName.ClassName[x1]...[xN];
```

Examples

Create 4-by-5 Java Array

Create 4-by-5 array of `java.lang.Double` type.

```
x1 = 4;
x2 = 5;
dblArray = javaArray('java.lang.Double',x1,x2);
```

Fill in values.

```
for m = 1:x1
    for n = 1:x2
        dblArray(m,n) = java.lang.Double((m*10) + n);
    end
end
```

Display results.

```
dblArray
```

```
dblArray =

  java.lang.Double[][]:

    [11]    [12]    [13]    [14]    [15]
    [21]    [22]    [23]    [24]    [25]
    [31]    [32]    [33]    [34]    [35]
    [41]    [42]    [43]    [44]    [45]
```

Input Arguments

`PackageName.ClassName` — Java class name
string | character vector

Name of the Java class, including the package name, specified as a string or character vector.

Data Types: `char`

`x1,...,xN` — Dimensions of array
integer

Dimensions of the array, specified as an integer. If any argument is zero, `javaArray` creates a zero-length Java array with the specified number of dimensions. A zero-length Java array is not the same as an empty MATLAB array, which is converted to a Java `null` when passed to a Java method.

Data Types: `double`

Output Arguments

`ObjArr` — Java array
Java array

Java array with dimensions `x1,...,xN`.

More About

Java Array Object

A Java array object is an object with Java dimensionality. For more information, see “How MATLAB Represents Java Arrays”.

Tips

- To create an array of primitive Java types, create an array of the equivalent MATLAB type. For more information, see “MATLAB Type to Java Type Mapping”.

See Also

`class` | `isjava` | `javaMethod` | `javaObject` | `methodsview`

Topics

“How MATLAB Represents Java Arrays”

Introduced before R2006a

javachk

Error message based on Java feature support

Syntax

```
MSG = javachk(feature)
javachk(feature,component)
```

Description

`MSG = javachk(feature)` returns a generic error message if the specified Java feature is not available in the current MATLAB session.

`javachk(feature,component)` also names the specified `component` in the error message.

Examples

Generate Error If Java Not Available

To test if Java is available before calling a Java command, use these statements. The `isempty` function is true when there is no error.

```
if isempty(javachk('jvm'))
   scalar = java.lang.Double(5);
end
```

Test Java Availability in User-Defined Script

To provide context in the `javachk` error message, add the optional text parameter to the `javachk` function.

Create a script, `myFile.m`, with the following statements. The script checks if Java is available before calling Java to create a frame.

```
error(javachk('awt','myFile'))
myFrame = java.awt.Frame;
myFrame.setVisible(1)
```

Start MATLAB from the Linux system prompt.

```
matlab -nojvm
```

Run the script.

```
myFile
```

If Java is not available, then MATLAB displays this error, identifying the source as the `myFile` script.

```
Error using myFile (line 1)
myFile is not supported because:
Java is not currently available.
```

Input Arguments

`feature` — Java feature
`'awt'` | `'desktop'` | `'jvm'` | `'swing'`

Java feature, specified as one of these values.

`'awt'`	UI components in the Java Abstract Window Toolkit (AWT) are available.
`'desktop'`	MATLAB interactive desktop is running.
`'jvm'`	Java Virtual Machine software (JVM™) is running.
`'swing'`	Swing components (Java lightweight UI components in the Java Foundation Classes) are available.

`component` — Identifier
string | character vector

Identifier, specified as a string or a character vector, to display in the error message.

Data Types: `string` | `char`

Output Arguments

`MSG` — Error message
structure

Error message, returned as a structure with the following fields. If the feature is available, `javachk` returns an error structure with empty fields.

`message` — Message
character vector | empty

Message, specified as a character vector.

`identifier` — Identifier
character vector | empty

Identifier, specified as a character vector.

See Also

`error` | `usejava`

Introduced before R2006a

javaclasspath

Return Java class path or specify dynamic path

Syntax

```
javaclasspath
javaclasspath('-dynamic')
javaclasspath('-static')

dpath = javaclasspath
spath = javaclasspath('-static')
jpath = javaclasspath('-all')

javaclasspath(dpath)
javaclasspath(dpath1,dpath2)

javaclasspath(statusmsg)
```

Description

`javaclasspath` displays the static and dynamic segments of the Java class path.

`javaclasspath('-dynamic')` displays the dynamic path.

`javaclasspath('-static')` displays the static path.

`dpath = javaclasspath` returns the dynamic path, `dpath`.

`spath = javaclasspath('-static')` returns the static path, `spath`.

`jpath = javaclasspath('-all')` returns the entire path, `jpath`. The returned cell array contains first the static segment of the path, and then the dynamic segment.

`javaclasspath(dpath)` changes the dynamic path to `dpath`. Use this syntax to reload Java classes that you are actively developing and debugging.

`javaclasspath(dpath1,dpath2)` changes the dynamic path to the concatenation of paths `dpath1,dpath2`.

`javaclasspath(statusmsg)` enables or disables the display of status messages.

Examples

Modify Path Using Cell Array

Use a cell array to add folders to an existing Java class path.

Create a cell array with two path values.

```
dpath = {'http://domain.com','http://some.domain.com/jarfile.jar'};
```

Set the message flag to display the class-loading messages.

```
javaclasspath('-v1')
```

Add the paths in `dpath` to the class path.

```
javaclasspath(dpath)
```

```
Loading following class path(s) from local file system:
* http://domain.com
* http://some.domain.com/jarfile.jar
```

Display the updated dynamic path. MATLAB adds folders from `dpath` to the existing path. The output depends on your system configuration.

```
javaclasspath('-dynamic')
```

```
        DYNAMIC JAVA PATH

    http://domain.com
    http://some.domain.com/jarfile.jar
```

Capture Contents of Dynamic Path

Create a cell array, `p`, with the entries of the dynamic path. If there are no entries on the dynamic path, then MATLAB creates an empty cell array.

```
javaclasspath('-v0') % Suppress display of class-loading messages
p = javaclasspath
```

```
p =

     {}
```

Input Arguments

`dpath` — Path entries
string | array of strings | character vector | cell array of character vectors

Path entries, specified as a string, an array of strings, a character vector, or a cell array of character vectors for the dynamic path. MATLAB converts relative paths to absolute paths.

Example: `javaclasspath('http://domain.com')`

Data Types: `char` | `cell`

`dpath1,dpath2` — Path entries
string | array of strings | character vector | cell array of character vectors

Path entries, specified as a string, an array of strings, a character vector, or a cell array of character vectors concatenated for the dynamic path.

Data Types: `char` | `cell`

`statusmsg` — Message flag
`'-v0'` (default) | `'-v1'`

Message flag, specified as one of these values.

`'-v0'`	Do not display status messages when loading the Java path from the file system.
`'-v1'`	Display status messages.

The `statusmsg` argument controls status message display from the `javaclasspath`, `javaaddpath`, and `javarmpath` functions.

Output Arguments

`dpath` — Dynamic path entries
array of strings | cell array of character vectors

Dynamic path entries for the current path, returned as an array of strings or a cell array of character vectors. If there are no path entries, then `dpath` is an empty cell array.

`spath` — Static path entries
array of strings | cell array of character vectors

Static path entries for the current path, returned as an array of strings or a cell array of character vectors. If there are no path entries, then `spath` is an empty cell array.

`jpath` — All path entries
array of strings | cell array of character vectors

All path entries, returned as an array of strings or a cell array of character vectors. If there are no path entries, then `jpath` is an empty cell array.

More About

Static Path

The static path is a segment of the Java path that is loaded at the start of each MATLAB session from the MATLAB built-in Java path and the file `javaclasspath.txt`.

MATLAB searches the static path before the dynamic path.

The static Java path offers better Java class-loading performance than the dynamic Java path. To modify the static Java path, edit the file `javaclasspath.txt` and restart MATLAB. For more information, see “Static Path”.

Dynamic Path

The dynamic path is a segment of the Java class path.

MATLAB provides the dynamic path as a convenience for when you develop your own Java classes. You can load the dynamic path any time during a MATLAB session using the `javaclasspath` function. Although the dynamic path offers flexibility in changing the path, Java classes on the dynamic path might load more slowly than classes on the static path. Also, classes on the dynamic path might not behave the same way as classes on the static path. If your class does not behave as expected, then use the static path. After you develop and debug a Java class, add the class to the static path.

- To define the dynamic path, use `javaclasspath`.
- To modify the path. use `javaaddpath` and `javarmpath`.
- To refresh the Java class definitions for all classes on the dynamic path without restarting MATLAB, use `clear java`.

For more information, see “Dynamic Path”.

Tips

- Do not put Java classes on the static path that have dependencies on classes on the dynamic path. Such dependencies produce run-time errors.
- To clear the definitions of all Java classes defined by files on the dynamic class path, call the `clear` command.
- Adding an entry to the dynamic path when it is already specified on the static path produces a warning.

See Also

`clear` | `javaaddpath` | `javarmpath`

Topics

“Java Class Path”

Introduced before R2006a

matlab.exception.JavaException class

Package: `matlab.exception`

Capture error information for Java exception

Description

Process information from a `matlab.exception.JavaException` object to handle Java errors thrown by Java methods called from MATLAB. This class is derived from `MException`.

Creation

You do not construct a `matlab.exception.JavaException` object explicitly. MATLAB automatically constructs a `JavaException` object whenever Java throws an exception. The `JavaException` object wraps the original Java exception.

Properties

`ExceptionObject` — Java exception object
`java.lang.Throwable`

Java exception object that caused the error, specified as a `java.lang.Throwable` object.

Examples

Display Error Information

Add a `matlab.exception.JavaException` object to a `try-catch` statement.

```
try
    java.lang.Class.forName('myfunction');
```

```
catch e
    e.message
    if(isa(e,'matlab.exception.JavaException'))
        ex = e.ExceptionObject;
        assert(isjava(ex));
        ex.printStackTrace;
    end
end
```

See Also

Topics

"Capture Information About Exceptions"
"Throw an Exception"

Introduced in R2012b

javaMethod

Call Java method

Syntax

```
javaMethod(MethodName,JavaObj,x1,...,xN)
javaMethod(StaticMethodName,ClassName,x1,...,xN)
```

Description

`javaMethod(MethodName,JavaObj,x1,...,xN)` calls the method in the class of the Java object array with the signature matching the arguments `x1,...,xN`. Use `javaMethod` to call methods having names that exceed the maximum length of a MATLAB identifier. This approach is the only way you can call such a method in MATLAB. To obtain the maximum identifier length, call the `namelengthmax` function.

In general, use MATLAB syntax to call methods on Java objects.

```
method(object,arg1,...,argn)
```

Alternatively, use Java syntax.

```
object.method(arg1,...,argn)
```

`javaMethod(StaticMethodName,ClassName,x1,...,xN)` calls the static method in class `ClassName`.

In general, use MATLAB syntax to call static methods on Java objects.

```
class.method(arg1,...,argn)
```

Examples

Call Method on Java Object

Create a `java.util.Date` object `myDate` and change the month to April. From the Java documentation, "A month is represented by an integer from 0 to 11; 0 is January, 1 is February, and so forth." Therefore, the numeric value for April is 3.

```
myDate = java.util.Date;
javaMethod('setMonth',myDate,3)
```

Call Static Method

Call `java.lang.Double` static method `isNaN` to test variable `num`. Since `num` contains a number, no message is displayed.

```
num = 2.2;
if javaMethod('isNaN','java.lang.Double',num)
    disp('This is not a number')
end
```

Call Method Specified at Runtime

Search for a text pattern in a string using variables for the pattern and for the search method. These variables could be set at runtime from user input.

Choose method, `startsWith`, and identify pattern, `str`.

```
fnc = 'startsWith';
str = java.lang.String('Four score');
```

Identify text to search.

```
gAddress = java.lang.String('Four score and seven years ago');
```

Search `gAddress` for the pattern.

```
javaMethod(fnc,gAddress,str)
```

```
ans = logical
   1
```

`gAddress` starts with the words `Four score`.

Call Java Inner Class

Call the constructor of or a static method in an inner class. In the `javaMethod` and `javaObject` functions, specify the class name, using the `$` character, as `OuterClass$InnerClass`.

For example, suppose class `com.ams.MyClass` contains class `MyInnerClass` with static method `methodname`. In Java, the calling syntax is:

```
out = com.ams.MyClass.MyInnerClass.methodname(arg);
```

In MATLAB, type:

```
out = javaMethod('methodname','com.ams.MyClass$MyInnerClass',arg)
```

Input Arguments

`MethodName` — Nonstatic Java method
string | character vector

Nonstatic Java method name, specified as a string or character vector.

Example: `javaMethod('DataDefinitionAndDataManipulationTransactions',T)`

Data Types: `char` | `string`

`JavaObj` — Array
Java object

Array, specified as a Java object of the class containing the method.

`x1,...,xN` — Java method input arguments
any type

Java method input arguments, 1 through N (if any), required by `MethodName` or `StaticMethodName`, specified by any type. The method argument list specifies the argument type.

`StaticMethodName` — Static Java method
string | character vector

Static Java method name, specified as a string or character vector.

Example: `java.lang.Double.isNaN(2.2)`

Data Types: `char` | `string`

`ClassName` — Java class
string | character vector

Java class name, specified as a string or character vector, containing `StaticMethodName`.

Data Types: `char` | `string`

Tips

- Use `javaMethod` to specify the method name as a variable to be invoked at runtime. When calling a static method, you also can use a variable in place of the class name argument. For example, see “Call Method Specified at Runtime” on page 1-8347.

See Also

`import` | `isjava` | `javaArray` | `javaMethodEDT` | `javaObject` | `methods`

Introduced before R2006a

javaMethodEDT

Call Java method from Event Dispatch Thread (EDT)

Syntax

```
javaMethodEDT(MethodName,JavaObj,x1,...,xN)
javaMethodEDT(StaticMethodName,ClassName,x1,...,xN)
```

Description

`javaMethodEDT(MethodName,JavaObj,x1,...,xN)` calls the method in the class of the Java object array with the signature matching the arguments `x1,...,xN`. MATLAB calls the method from the Event Dispatch Thread (EDT on page 1-8351).

`javaMethodEDT(StaticMethodName,ClassName,x1,...,xN)` calls the static method in class `ClassName`.

Examples

Call Method from EDT

Create a `java.util.Vector` object `v` and add a `string` element.

```
v = java.util.Vector;
javaMethodEDT('add',v,'string');
```

Input Arguments

`MethodName` — Nonstatic Java method
string | character vector

Nonstatic Java method name, specified as a string or character vector.

Example: `javaMethod('DataDefinitionAndDataManipulationTransactions',T)`

Data Types: `char` | `string`

`JavaObj` — Array
Java object

Array, specified as a Java object of the class containing the method.

`x1,...,xN` — Java method input arguments
any type

Java method input arguments, 1 through N (if any), required by `MethodName` or `StaticMethodName`, specified by any type. The method argument list specifies the argument type.

`StaticMethodName` — Static Java method
string | character vector

Static Java method name, specified as a string or character vector.

Example: `java.lang.Double.isNaN(2.2)`

Data Types: `char` | `string`

`ClassName` — Java class
string | character vector

Java class name, specified as a string or character vector, containing `StaticMethodName`.

Data Types: `char` | `string`

More About

EDT

The EDT is the Event Dispatch Thread, used in Java.

See Also

`import` | `isjava` | `javaMethod` | `javaObjectEDT` | `methods`

Introduced in R2009a

javaObject

Call Java constructor

Syntax

```
JavaObj = javaObject(ClassName,x1,...,xN)
```

Description

`JavaObj = javaObject(ClassName,x1,...,xN)` returns Java object array, `JavaObj`, created by the Java constructor for the class with the argument list matching `x1,...,xN`.

Examples

Create Java Object

Create a Java® object `strObj` of class `java.lang.String`.

```
strObj = javaObject('java.lang.String','hello');
```

Input Arguments

`ClassName` — Java class
string | character vector

Java class name, specified as a string or character vector.

Data Types: `char`

`x1,...,xN` — Java constructor input arguments
any type

Java constructor input arguments, 1 through N (if any), required by `ClassName`, specified as any type. The class constructor argument list specifies the argument type.

See Also

`import` | `javaArray` | `javaMethod` | `javaObjectEDT` | `methods`

Topics

"Call Java Method"

Introduced before R2006a

javaObjectEDT

Call Java constructor on Event Dispatch Thread (EDT)

Syntax

```
JavaObj = javaObjectEDT(ClassName,x1,...,xN)
```

Description

`JavaObj = javaObjectEDT(ClassName,x1,...,xN)` returns Java object array, `JavaObj`, created from the EDT on page 1-8356 by the Java constructor for the class with the signature matching the arguments `x1,...,xN`.

Examples

Construct Java Object Array from EDT

```
f = javaObjectEDT('javax.swing.JFrame','New Title');
```

Call Method on Java Object

Create a `JOptionPane` on the EDT.

```
optPane = javaObjectEDT('javax.swing.JOptionPane');
```

Call the `createDialog` method on the EDT.

```
dlg = optPane.createDialog([],'Sample Dialog');
```

Input Arguments

`ClassName` — Java class
string | character vector

Java class name, specified as a string or character vector.

Data Types: `char`

`x1,...,xN` — Java constructor input arguments
any type

Java constructor input arguments, 1 through N (if any), required by `ClassName`, specified as any type. The class constructor argument list specifies the argument type.

More About

EDT

The EDT is the Event Dispatch Thread, used in Java.

Tips

- MATLAB calls methods on `JavaObj` from the EDT.
- Static methods on the specified class or Java object run on the MATLAB thread unless called using the `javaMethodEDT` function.

See Also

`import` | `javaMethodEDT` | `javaObject` | `methods`

Introduced in R2009a

javarmpath

Remove entries from dynamic Java class path

Syntax

```
javarmpath(dpath1,...,dpathN)
```

Description

`javarmpath(dpath1,...,dpathN)` removes one or more files or folders from the current dynamic class path.

Examples

Remove Folder from Dynamic Path

Preserve the state of the dynamic path on your system by first adding folders to the path, and then removing one of these folders.

Create a variable that points to the MATLAB examples folder. The path reflects the folder to your MATLAB installation.

```
expath = fullfile(matlabroot,'extern','examples')
```

```
expath =

C:\Program Files\MATLAB\R2012b\extern\examples
```

Add two folders to the path. The output displays these new folders on your existing path.

```
javaclasspath({...
   expath,...
   'http://www.example.com'})
javaclasspath('-dynamic')
```

```
        DYNAMIC JAVA PATH

    C:\Program Files\MATLAB\R2012b\extern\examples
    http://www.example.com
```

Remove one folder. The path no longer contains the `extern\examples` folder.

```
javarmpath(expath)
javaclasspath('-dynamic')
```

```
        DYNAMIC JAVA PATH

    http://www.example.com
```

Input Arguments

`dpath1,...,dpathN` — Folders or JAR files
string | character vector

Folders or JAR files, specified as strings or character vectors, to remove from path.

Data Types: `char`

Tips

- `javarmpath` clears the definitions of all Java classes defined by files on the dynamic class path.

See Also

Functions
`clear` | `javaaddpath` | `javaclasspath`

Topics
"Java Class Path"

Introduced before R2006a

join

Combine strings

Syntax

```
newStr = join(str)
newStr = join(str,delimiter)
newStr = join(str,dim)
newStr = join(str,delimiter,dim)
```

Description

`newStr = join(str)` combines the text in `str` by joining consecutive elements of the input array, placing a space character between them. `str` can be a string array or a cell array of character vectors. `newStr` has the same data type as `str`.

- If `str` is a `1`-by-`N` or an `N`-by-`1` string array or cell array, then `newStr` is a string scalar or a cell array that contains one character vector.
- If `str` is an `M`-by-`N` string array or cell array, then `newStr` is an `M`-by-`1` string array or cell array.

For a string or cell array of any size, `join` concatenates elements along the last dimension of `str` with a size that does not equal `1`.

`newStr = join(str,delimiter)` combines the text in `str` and places the elements of `delimiter` between the elements of `str` instead of a space character.

If the `delimiter` argument is an array of different delimiters, and `str` has `N` elements along the dimension that is joined, then `delimiter` must have `N-1` elements along the same dimension. The other dimensions of `delimiter` must have either a size of `1` or the same size as the corresponding dimension of `str`.

`newStr = join(str,dim)` combines the elements in `str` along the dimension `dim`.

`newStr = join(str,delimiter,dim)` combines the elements in `str` along the dimension `dim` and places the elements of `delimiter` between the elements of `str`.

Examples

Combine Strings in String Array

Create a string array. Starting in R2017a, you can create strings using double quotes.

```
str = ["Carlos","Sada";
       "Ella","Olsen";
       "Diana","Lee"]
```

```
str = 3x2 string array
    "Carlos"    "Sada"
    "Ella"      "Olsen"
    "Diana"     "Lee"
```

Combine the strings using the `join` function. `join` concatenates the strings from `str` and places a space character between them. `join` concatenates along the second dimension, because it is the last dimension with a size that does not equal `1`.

```
newStr = join(str)
```

```
newStr = 3x1 string array
    "Carlos Sada"
    "Ella Olsen"
    "Diana Lee"
```

Combine Elements of String Array with Delimiters

Combine elements in a string array. Instead of spaces, insert different pieces of text between the strings in `str`.

Create a string array. Starting in R2017a, you can create strings using double quotes.

```
str = ["x","y","z";
       "a","b","c"]
```

```
str = 2x3 string array
    "x"    "y"    "z"
```

```
    "a"    "b"    "c"
```

Concatenate the strings with dashes between them.

```
newStr = join(str,"-")
```

```
newStr = 2x1 string array
    "x-y-z"
    "a-b-c"
```

Concatenate the strings with symbols that make the output strings represent equations. The `delimiters` argument must be a 2-by-2 array because `str` is a 2-by-3 array.

```
delimiters = [" + "," = ";
              " - "," = "];
newStr = join(str,delimiters)
```

```
newStr = 2x1 string array
    "x + y = z"
    "a - b = c"
```

Combine String Array Along Specified Dimension

Create a string array. Starting in R2017a, you can create strings using double quotes.

```
str = ["Carlos","Sada";
       "Ella","Olsen";
       "Diana","Lee"]
```

```
str = 3x2 string array
    "Carlos"    "Sada"
    "Ella"      "Olsen"
    "Diana"     "Lee"
```

Combine the strings in `str` along the first dimension. By default, the `join` function combines strings along the last dimension with a size that does not equal `1`. To combine the strings along the first dimension, specify it as an additional input argument.

```
newStr = join(str,1)
```

```
newStr = 1x2 string array
    "Carlos Ella Diana"    "Sada Olsen Lee"
```

Input Arguments

`str` — Input text
string array | cell array of character vectors

Input text, specified as a string array or a cell array of character vectors.

`delimiter` — Delimiting characters for joining strings
`' '` (default) | character vector | cell array of character vectors | string array

Delimiting characters for joining strings, specified as a character vector, a cell array of character vectors, or a string array. `join` forms the output string array by combining string elements with delimiters between them.

`join` inserts all characters in `delimiter` as literal text, including escaped character sequences.

`dim` — Dimension along which to join strings
positive integer

Dimension along which to join strings, specified as a positive integer. If `dim` is not specified, then the default is the last dimension with a size that does not equal `1`.

Output Arguments

`newStr` — Output text
string array | cell array of character vectors

Output text, returned as a string array or a cell array of character vectors. `newStr` has the same data type as the input text and has a size of `1` along the dimension being joined.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`compose` | `newline` | `plus` | `regexp` | `split` | `splitlines` | `sprintf` | `string`

Introduced in R2016b

Join Tables

Combine two tables using key variables in the Live Editor

Description

The **Join Tables** task lets you interactively combine two tables by performing joins or by concatenating the tables horizontally or vertically. The task automatically generates MATLAB code for your live script.

Using this task, you can:

- Perform joins, including inner and outer joins, on tables by specifying *key variables*—table variables whose matching values, or *key values*, determine which rows are merged into the output table.
- Horizontally or vertically concatenate tables with the same number of rows or variables, respectively.

Open the Task

To add the **Join Tables** task to a live script in the MATLAB Editor:

- On the **Live Editor** tab, select **Task > Join Tables**.
- In a code block in the script, type a relevant keyword, such as `join` or `tables`. Select `Join Tables` from the suggested command completions.

Parameters

`Left table` — Name of table or timetable

drop-down list item

Specify the name from a list of all the nonempty tables and timetables that are in the workspace.

`Right table` — Name of table or timetable

drop-down list item

Specify the name from a list of all the nonempty tables and timetables that are in the workspace.

`Merging variable` — Name of table variable with key values
drop-down list item

Specify the name of a variable from a list of variables in the left or right table.

- The drop-down list for the left table contains a full list of its variables.
- The drop-down list for the right table contains a subset of variables. The list of merging variables from the right table is constrained by your choice of merging variables from the left table.

When you specify a merging, or *key*, variable, its values determine which rows are merged from the left and right tables. To specify multiple sets of merging variables, use the **+** button.

`Combine merging variable` — Check box for combining left and right merging variables
`off` (default) | `on`

Combine corresponding merging variables when performing outer joins. By default, outer joins copy key variables from the left and right tables to their own variables in the output table. To combine corresponding key variables in the left and right tables into one variable in the output, select this check box.

Examples

Join Tables Using Live Editor Task

This example shows how to use the **Join Tables** Live Editor task to perform an inner join and an outer join on two tables.

First, load two tables. The `orders` table has order IDs, customer names, and the order date for a number of shipments.

```
load orders
```

```
orders = 3×3 table
```

	OrderID	CustomerID	OrderDate
1	5120	'Sanchez'	23-Apr-2019
2	1037	'Li'	18-Apr-2019
3	8937	'Johnson'	16-Apr-2019

Then load the `items` table, which contains products that customers ordered, along with the price and status of the shipment for that item. Each row of this table has an order ID, just like `orders`. Because a customer can order multiple items, several rows of `items` can refer to one order from `orders`.

```
load items
```

```
items = 5×5 table
```

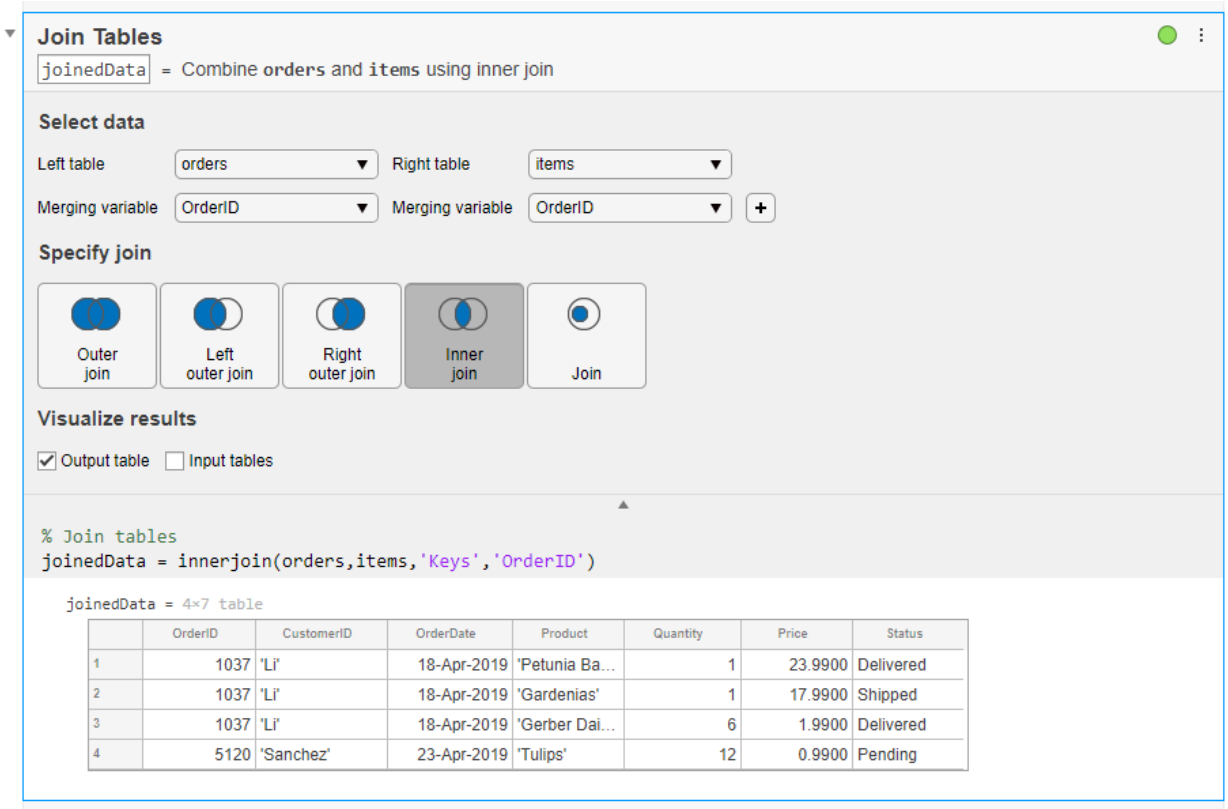
	OrderID	Product	Quantity	Price	Status
1	6005	'Dozen Ros...	1	39.9900	Shipped
2	1037	'Petunia Ba...	1	23.9900	Delivered
3	5120	'Tulips'	12	0.9900	Pending
4	1037	'Gardenias'	1	17.9900	Shipped
5	1037	'Gerber Dai...	6	1.9900	Delivered

Open the **Join Tables** task. To open the task, type the keyword `join` in a code block and click `Join Tables` when it appears in the menu.

Use the task to perform an *inner join* of `orders` and `items`. When the task opens:

1. Select `orders` and `items` as the left and right tables, respectively.
2. Select `OrderID` as the merging variable for both tables.
3. Click the **Inner join** button.

4 To see the code that this task generates, click the small arrow under the **Visualize results** section. In this example, the generated code is a call to the `innerjoin` function.



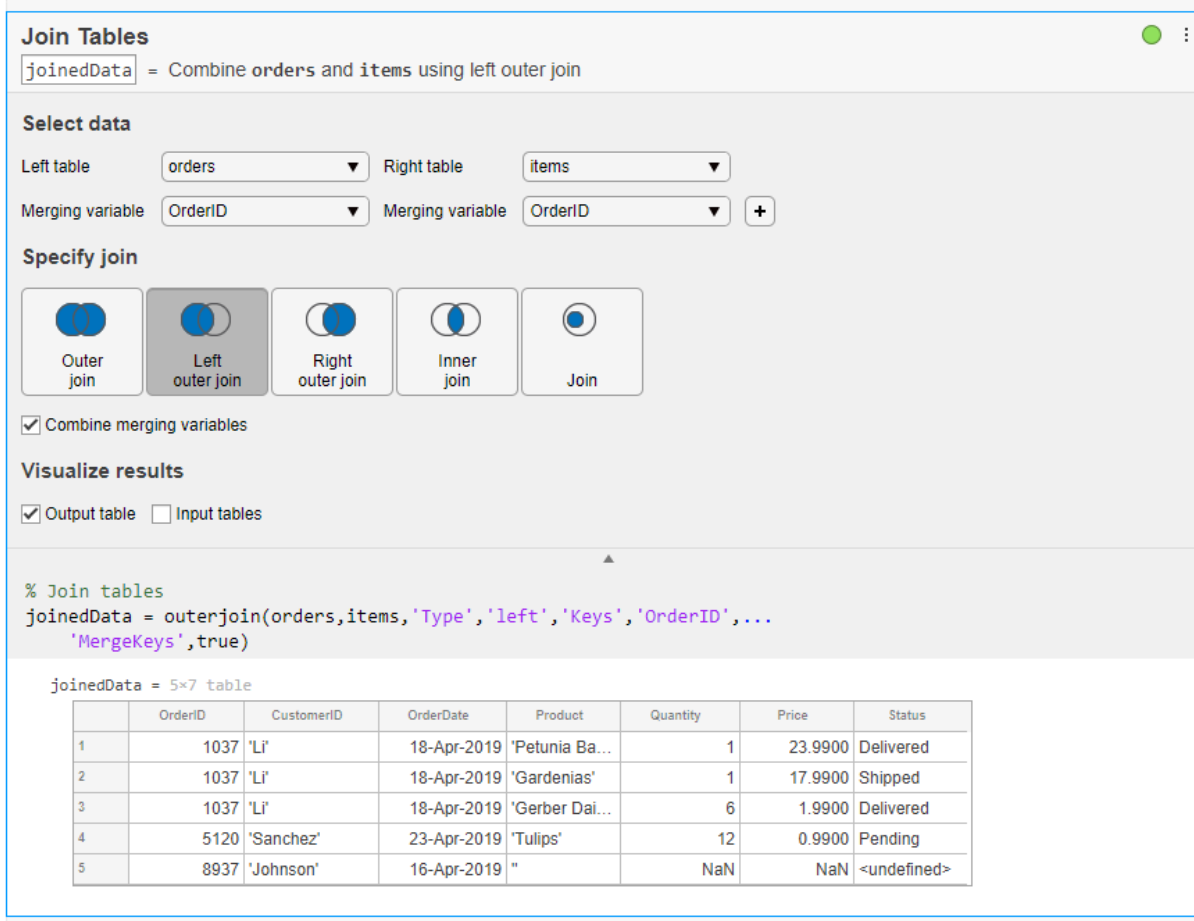
	OrderID	CustomerID	OrderDate	Product	Quantity	Price	Status
1	1037	'Li'	18-Apr-2019	'Petunia Ba...	1	23.9900	Delivered
2	1037	'Li'	18-Apr-2019	'Gardenias'	1	17.9900	Shipped
3	1037	'Li'	18-Apr-2019	'Gerber Dai...	6	1.9900	Delivered
4	5120	'Sanchez'	23-Apr-2019	'Tulips'	12	0.9900	Pending

When you perform an inner join, the output table includes only those key values that appear in *both* the left and right tables.

- Only the values `1037` and `5120` occur in both input tables. Therefore, the output table contains data for only those two orders.
- Value `1037` occurs multiple times in `items`, once for each item, but only once in `orders`. Therefore, the `CustomerID` and `OrderDate` values corresponding to `1037` are copied as many times as needed in the output.

Next, use the task to perform a *left outer join*. Outer joins can include key values that appear in only one input table. For example, a left outer join includes all key values from the left table, even when the right table has no corresponding matches. If the right table has key values that do not have matches in the left table, then those key values are not included.

1. Click the **Left outer join** button.
2. Select the **Combine merging variables** check box. By default, outer joins copy the key variables from the left and right tables into separate variables in the output table. Merge the key variables so one key variable is in the output.
3. To see the generated call to the `outerjoin` function, click the small arrow under **Visualize results**.



	OrderID	CustomerID	OrderDate	Product	Quantity	Price	Status
1	1037	'Li'	18-Apr-2019	'Petunia Ba...	1	23.9900	Delivered
2	1037	'Li'	18-Apr-2019	'Gardenias'	1	17.9900	Shipped
3	1037	'Li'	18-Apr-2019	'Gerber Dai...	6	1.9900	Delivered
4	5120	'Sanchez'	23-Apr-2019	'Tulips'	12	0.9900	Pending
5	8937	'Johnson'	16-Apr-2019	''	NaN	NaN	<undefined>

The output table now includes data for order `8937`. However, because the `items` table for order `8937` had no items, the rest of the row is filled in with empty values (such as `''`, `NaN`, or `<undefined>`). Outer joins fill table elements with empty values when the left or right tables do not have data associated with a key value.

- Full outer joins include all key values and data from both tables.
- Left outer joins include all key values and data from the left table, but only matching key values and associated data from the right.

- Right outer joins include all key values and data from the right table, but only matching key values and associated data from the left.

- "Add Interactive Tasks to a Live Script"

See Also

`innerjoin` | `join` | `outerjoin`

Topics

"Add Interactive Tasks to a Live Script"

Introduced in R2019b

juliandate

Convert MATLAB datetime to Julian date

Syntax

```
d = juliandate(t)
d = juliandate(t,dateType)
```

Description

`d = juliandate(t)` returns the “Julian dates” on page 1-8373 equivalent to the `datetime` values in `t`.

- If the time zone of `t` is not specified, then `juliandate` treats the times in `t` as UTC times. This interpretation might differ from your treatment of “unzoned” `datetime` arrays in other contexts. For example, you might think of `datetime('now')` as returning your local time. However, `juliandate` interprets it as a UTC time.
- If the time zone of `t` is specified, then `juliandate` uses the offset for the time zone to compute Julian dates with respect to UTC.
- `juliandate` ignores leap seconds unless the time zone of `t` is `UTCLeapSeconds`.

The best practice is to specify the time zone of `t` before calling `juliandate`.

`d = juliandate(t,dateType)` returns the type of Julian dates specified by `dateType`. For example, you can convert datetime values to modified Julian dates.

Examples

Convert Datetime Array to Julian Dates

Create `datetime` values and convert them to the equivalent Julian dates. Show the differences in Julian dates between zoned and unzoned `datetime` values. The best practice is to specify a time zone for a `datetime` array before calling `juliandate`.

Create a `datetime` array and specify its time zone.

```
t1 = datetime('2016-07-29 10:05:24') + calmonths(1:3);
t1.TimeZone = 'America/New_York'
```

```
t1 = 1x3 datetime array
   29-Aug-2016 10:05:24   29-Sep-2016 10:05:24   29-Oct-2016 10:05:24
```

Convert `t1` to the equivalent Julian dates. `juliandate` accounts for the time zone offset when it computes Julian dates.

```
format longG
jd1 = juliandate(t1)
```

```
jd1 = 1×3

          2457630.08708333          2457661.08708333          2457691.08708333
```

Create a `datetime` array with the same values as `t1`, but with no time zone. Convert it to the equivalent Julian dates. `juliandate` treats the times in `t2` as UTC times, with no time zone offset.

```
t2 = datetime('2016-07-29 10:05:24') + calmonths(1:3);
jd2 = juliandate(t2)
```

```
jd2 = 1×3

          2457629.92041667          2457660.92041667          2457690.92041667
```

Show the differences between `jd2` and `jd1`. The differences are equal to the time offset, in days, between UTC and the time zone of `jd1`.

```
jd2 - jd1
```

```
ans = 1×3

        -0.166666666511446        -0.166666666511446        -0.166666666511446
```

Input Arguments

`t` — Input date and time
`datetime` array

Input date and time, specified as a `datetime` array.

`dateType` — Type of Julian date values
`'juliandate'` (default) | `'modifiedjuliandate'`

Type of Julian date values, specified as either `'juliandate'` or `'modifiedjuliandate'`.

- If `dateType` is `'juliandate'`, then `juliandate` converts the datetime values in `t` to the equivalent Julian dates. A Julian date is the number of days and fractional days since noon on November 24, 4714 BCE in the proleptic Gregorian calendar, or January 1, 4713 BCE in the proleptic Julian calendar.
- If `dateType` is `'modifiedjuliandate'`, then `juliandate` converts the datetime values in `t` to the equivalent modified Julian dates. A modified Julian date is the number of days and fractional days since November 17, 1858 00:00:00.

More About

Julian dates

A Julian date is the number of days and fractional days since noon on November 24, 4714 BCE in the proleptic Gregorian calendar, or January 1, 4713 BCE in the proleptic Julian calendar.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`datenum` | `datetime` | `exceltime` | `posixtime` | `yyyymmdd`

Topics

“Convert Date and Time to Julian Date or POSIX Time”

Introduced in R2014b

keyboard

Give control to keyboard

Syntax

```
keyboard
```

Description

`keyboard` pauses execution of a running program and gives control to the keyboard. Place the `keyboard` function in a program at the location where you want MATLAB to pause. When the program pauses, the prompt in the Command Window changes to K>>, indicating that MATLAB is in debug mode. You then can view or change the values of variables to see if the new values produce expected results.

The `keyboard` function is useful for debugging your functions.

Examples

Modify Variables While Debugging

Use the `keyboard` command to pause execution of a program and modify a variable before continuing.

Create a file, `buggy.m`, containing these statements.

```
function z = buggy(x)
n = length(x);
keyboard
z = (1:n)./x;
```

Run `buggy.m`. MATLAB pauses at line 3, where the `keyboard` command is located.

```
buggy(5)
```

Multiply the variable `x` by 2 and continue running the program. MATLAB executes the rest of the program using the new value of `x`.

```
x = x * 2
dbcont
```

Tips

- To terminate debug mode and continue execution, use the `dbcont` command.
- To terminate debug mode and exit the file without completing execution, use the `dbquit` command.

See Also

`dbcont` | `dbquit` | `dbstop` | `input`

Introduced before R2006a

matlab.keyboard Settings

Keyboard settings including suggestions and completions settings

You can customize MATLAB using the `matlab.keyboard` settings. Access `matlab.keyboard` settings using the root `SettingsGroup` object returned by the `settings` function. For example, set the temporary value for whether to show suggestions and completions automatically in the Live Editor to `0` (where the default is `1`). When set to `0`, suggestions and completions show only after you press the **Tab** key.

```
s = settings;
s.matlab.keyboard.suggestions.ShowAutomatically.TemporaryValue = 0
```

For more information about settings, see “Access and Modify Settings”.

Settings

matlab.keyboard.suggestions

`ShowAutomatically` — Show suggestions and completions automatically

`1` (default) | `0`

Show suggestions and completions automatically in the Live Editor, specified as `1` or `0`.

Set to `1` to show suggestions and completions automatically in the Live Editor. Set to `0` to show suggestions and completions only after you press the **Tab** key.

Example:
`s.matlab.keyboard.suggestions.ShowAutomatically.TemporaryValue = 0`

See Also

`settings`

Topics

“Access and Modify Settings”
“Set Keyboard Preferences”

Introduced in R2018a

keys

Package: `containers`

Return keys of `Map` object

Syntax

```
keySet = keys(M)
```

Description

`keySet = keys(M)` returns a cell array containing all the keys in the input `Map` object.

Examples

Keys of Map

Create a `Map` object.

```
months = {'Jan','Feb','Mar','Apr'};
rainfall = [327.2 368.2 197.6 178.4];
M = containers.Map(months,rainfall)
```

```
M = 
  Map with properties:

        Count: 4
      KeyType: char
    ValueType: double
```

Return a cell array containing its keys.

```
keySet = keys(M)
```

```
keySet = 1x4 cell array
    {'Apr'}    {'Feb'}    {'Jan'}    {'Mar'}
```

Input Arguments

M — Input Map object
Map object

Input Map object.

See Also

containers.Map | isKey | remove | values

Topics

"Overview of Map Data Structure"
"Examine Contents of Map"
"Read and Write Using Key Index"
"Modify Keys and Values in Map"

Introduced in R2008b

kron

Kronecker tensor product

Syntax

```
K = kron(A,B)
```

Description

`K = kron(A,B)` returns the Kronecker tensor product on page 1-8384 of matrices `A` and `B`. If `A` is an `m`-by-`n` matrix and `B` is a `p`-by-`q` matrix, then `kron(A,B)` is an `m*p`-by-`n*q` matrix formed by taking all possible products between the elements of `A` and the matrix `B`.

Examples

Block Diagonal Matrix

Create a block diagonal matrix.

Create a 4-by-4 identity matrix and a 2-by-2 matrix that you want to be repeated along the diagonal.

```
A = eye(4);
B = [1 -1;-1 1];
```

Use `kron` to find the Kronecker tensor product.

```
K = kron(A,B)
```

```
K = 8×8

     1    -1     0     0     0     0     0     0
    -1     1     0     0     0     0     0     0
     0     0     1    -1     0     0     0     0
     0     0    -1     1     0     0     0     0
```

```
     0     0     0     0     1    -1     0     0
     0     0     0     0    -1     1     0     0
     0     0     0     0     0     0     1    -1
     0     0     0     0     0     0    -1     1
```

The result is an 8-by-8 block diagonal matrix.

Repeat Matrix Elements

Expand the size of a matrix by repeating elements.

Create a 2-by-2 matrix of ones and a 2-by-3 matrix whose elements you want to repeat.

```
A = [1 2 3; 4 5 6];
B = ones(2);
```

Calculate the Kronecker tensor product using `kron`.

```
K = kron(A,B)
```

```
K = 4×6

     1     1     2     2     3     3
     1     1     2     2     3     3
     4     4     5     5     6     6
     4     4     5     5     6     6
```

The result is a 4-by-6 block matrix.

Sparse Laplacian Operator Matrix

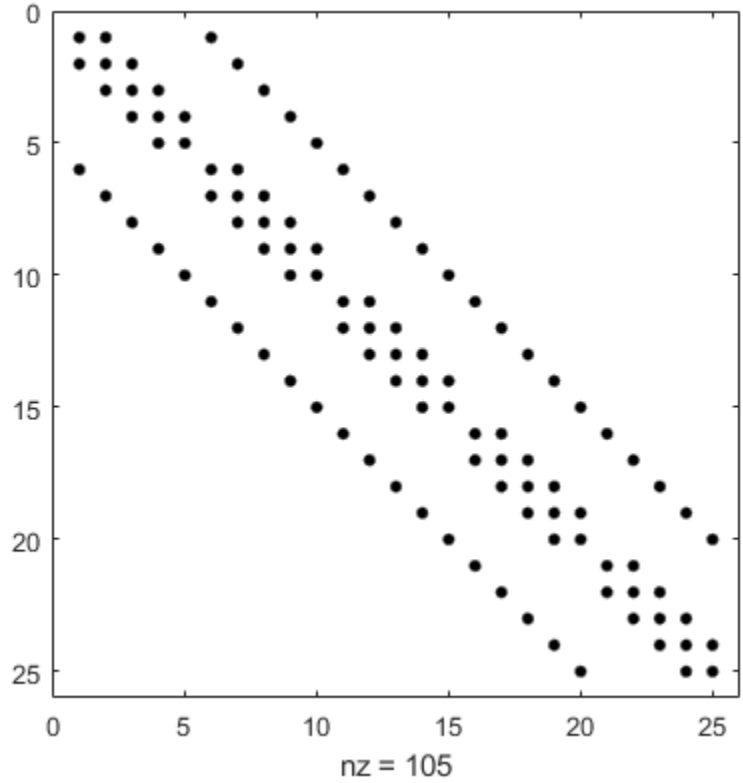
This example visualizes a sparse Laplacian operator matrix.

The matrix representation of the discrete Laplacian operator on a two-dimensional, `n`-by-`n` grid is a `n*n`-by- `n*n` sparse matrix. There are at most five nonzero elements in each row or column. You can generate the matrix as the Kronecker product of one-dimensional difference operators. In this example `n = 5`.

```
n = 5;
I = speye(n,n);
E = sparse(2:n,1:n-1,1,n,n);
D = E+E'-2*I;
A = kron(D,I)+kron(I,D);
```

Visualize the sparsity pattern with `spy`.

```
spy(A,'k')
```



Input Arguments

A,B — Input matrices
scalars | vectors | matrices

Input matrices, specified as scalars, vectors, or matrices. If either A or B is sparse, then kron multiplies only nonzero elements and the result is also sparse.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical
Complex Number Support: Yes

More About

Kronecker Tensor Product

If A is an m-by-n matrix and B is a p-by-q matrix, then the Kronecker tensor product of A and B is a large matrix formed by multiplying B by each element of A

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \cdots & a_{1n}B \\ a_{21}B & a_{22}B & \cdots & a_{2n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}B & a_{m2}B & \cdots & a_{mn}B \end{bmatrix}.$$

For example, two simple 2-by-2 matrices produce

$$A = \begin{bmatrix} 1 & -2 \\ -1 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 4 & -3 \\ 2 & 3 \end{bmatrix}$$

$$A \otimes B = \begin{bmatrix} 1 \cdot 4 & 1 \cdot -3 & -2 \cdot 4 & -2 \cdot -3 \\ 1 \cdot 2 & 1 \cdot 3 & -2 \cdot 2 & -2 \cdot 3 \\ -1 \cdot 4 & -1 \cdot -3 & 0 \cdot 4 & 0 \cdot -3 \\ -1 \cdot 2 & -1 \cdot 3 & 0 \cdot 2 & 0 \cdot 3 \end{bmatrix} = \begin{bmatrix} 4 & -3 & -8 & 6 \\ 2 & 3 & -4 & -6 \\ -4 & 3 & 0 & 0 \\ -2 & -3 & 0 & 0 \end{bmatrix}.$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`cross` | `dot` | `hankel` | `toeplitz`

Introduced before R2006a

KeyValueStore

Store key-value pairs for use with mapreduce

Description

The `mapreduce` function automatically creates a `KeyValueStore` object during execution and uses it to store key-value pairs added by the map and reduce functions. Although you never need to explicitly create a `KeyValueStore` object to use `mapreduce`, you do need to use the `add` and `addmulti` object functions to interact with this object in the map and reduce functions.

Creation

The `mapreduce` function automatically creates `KeyValueStore` objects during execution.

Object Functions

add	Add single key-value pair to KeyValueStore
addmulti	Add multiple key-value pairs to KeyValueStore

Examples

Add Key-Value Pair to KeyValueStore in Map Function

The following map function uses the `add` function to add key-value pairs one at a time to an intermediate `KeyValueStore` object (named `intermKVStore`).

```
function MeanDistMapFun(data, info, intermKVStore)
    distances = data.Distance(~isnan(data.Distance));
    sumLenKey = 'sumAndLength';
    sumLenValue = [sum(distances), length(distances)];
```

```
    add(intermKVStore, sumLenKey, sumLenValue);
end
```

Add Multiple Key-Value Pairs to KeyValueStore in Map Function

The following map function uses `addmulti` to add several key-value pairs to an intermediate `KeyValueStore` object (named `intermKVStore`). Note that this map function collects multiple keys in the `intermKeys` variable, and multiple values in the `intermVals` variable. This prepares a single call to `addmulti` to add all of the key-value pairs at once. It is a best practice to use a single call to `addmulti` rather than using `add` in a loop.

```
function meanArrivalDelayByDayMapper(data, ~, intermKVStore)
% Mapper function for the MeanByGroupMapReduceExample.

% Copyright 2014 The MathWorks, Inc.

% Data is an n-by-2 table: first column is the DayOfWeek and the second
% is the ArrDelay. Remove missing values first.
delays = data.ArrDelay;
day = data.DayOfWeek;
notNaN =~isnan(delays);
day = day(notNaN);
delays = delays(notNaN);

% find the unique days in this chunk
[intermKeys,~,idx] = unique(day, 'stable');

% group delays by idx and apply @grpstatsfun function to each group
intermVals = accumarray(idx,delays,size(intermKeys),@countsum);
addmulti(intermKVStore,intermKeys,intermVals);

function out = countsum(x)
n = length(x); % count
s = sum(x); % mean
out = {[n, s]};
```

See Also

`mapreduce`

Topics

"Getting Started with MapReduce"

Introduced in R2014b

lag

Time-shift data in timetable

Syntax

```
TT2 = lag(TT1)
TT2 = lag(TT1,n)
TT2 = lag(TT1,dt)
```

Description

`TT2 = lag(TT1)` shifts the data in each variable in `TT1` forward in time by one time step. A shift forward in time is a *lag*. The row times of `TT1` must be regular.

The timetables `TT1` and `TT2` have the same row times. When you shift the data in each variable forward, `lag` fills the first row of `TT2` with missing data indicators.

`TT2 = lag(TT1,n)` shifts data by `n` time steps. `n` must be an integer. If `n` is positive, then `lag` shifts the data forward in time (a *lag*). If `n` is negative, then `lag` shifts the data backward in time (a *lead*).

`TT2 = lag(TT1,dt)` shifts data by `dt`, a time interval. `dt` is either a duration or a calendar duration. `dt` must be a multiple of the regular time step of `TT1`.

Examples

Shift Timetable Data

Create a timetable with temperature data as the variable. Shift the data forward in time by one time step and compare it to the original timetable.

```
Time = datetime('2015-12-18 12:00:00') + days(1:5)';
Temp = [43 42.6 42.3 39.1 37.3]';
TT = timetable(Time,Temp)
```

```
TT=5×2 timetable
            Time            Temp
    ____________________    ____

    19-Dec-2015 12:00:00      43
    20-Dec-2015 12:00:00    42.6
    21-Dec-2015 12:00:00    42.3
    22-Dec-2015 12:00:00    39.1
    23-Dec-2015 12:00:00    37.3
```

```
TT2 = lag(TT)
```

```
TT2=5×2 timetable
            Time            Temp
    ____________________    ____

    19-Dec-2015 12:00:00     NaN
    20-Dec-2015 12:00:00      43
    21-Dec-2015 12:00:00    42.6
    22-Dec-2015 12:00:00    42.3
    23-Dec-2015 12:00:00    39.1
```

Synchronize the two timetables for comparison. Since the timetables already have the same row times, `synchronize` renames the variables and horizontally concatenates them.

```
synchronize(TT,TT2)
```

```
ans=5×3 timetable
            Time            Temp_TT    Temp_TT2
    ____________________    _______    ________

    19-Dec-2015 12:00:00       43         NaN
    20-Dec-2015 12:00:00     42.6          43
    21-Dec-2015 12:00:00     42.3        42.6
    22-Dec-2015 12:00:00     39.1        42.3
    23-Dec-2015 12:00:00     37.3        39.1
```

Shift Timetable Data by Multiple Time Steps

Create a timetable with temperature data as the variable. Shift the data in time by multiple time steps.

```
Time = datetime('2015-12-18 12:00:00') + days(1:5)';
Temp = [43 42.6 42.3 39.1 37.3]';
TT = timetable(Time,Temp)
```

```
TT=5×2 timetable
            Time            Temp
    ____________________    ____

    19-Dec-2015 12:00:00      43
    20-Dec-2015 12:00:00    42.6
    21-Dec-2015 12:00:00    42.3
    22-Dec-2015 12:00:00    39.1
    23-Dec-2015 12:00:00    37.3
```

Shift the data forward two time steps.

```
TT2 = lag(TT,2)
```

```
TT2=5×2 timetable
            Time            Temp
    ____________________    ____

    19-Dec-2015 12:00:00     NaN
    20-Dec-2015 12:00:00     NaN
    21-Dec-2015 12:00:00      43
    22-Dec-2015 12:00:00    42.6
    23-Dec-2015 12:00:00    42.3
```

Shift the data backward by three time steps.

```
TT2 = lag(TT,-3)
```

```
TT2=5×2 timetable
            Time            Temp
    ____________________    ____

    19-Dec-2015 12:00:00    39.1
    20-Dec-2015 12:00:00    37.3
```

```
    21-Dec-2015 12:00:00     NaN
    22-Dec-2015 12:00:00     NaN
    23-Dec-2015 12:00:00     NaN
```

Shift Timetable Data by Time Interval

Create a timetable with temperature data as the variable. Shift the data in time by two calendar months.

```
Time = datetime('2015-12-01 12:00:00') + calmonths(1:5)';
Temp = [43 37 35 39 45]';
TT = timetable(Time,Temp)
```

```
TT=5×2 timetable
            Time            Temp
    ____________________    ____

    01-Jan-2016 12:00:00     43
    01-Feb-2016 12:00:00     37
    01-Mar-2016 12:00:00     35
    01-Apr-2016 12:00:00     39
    01-May-2016 12:00:00     45
```

```
TT2 = lag(TT,calmonths(2))
```

```
TT2=5×2 timetable
            Time            Temp
    ____________________    ____

    01-Jan-2016 12:00:00    NaN
    01-Feb-2016 12:00:00    NaN
    01-Mar-2016 12:00:00     43
    01-Apr-2016 12:00:00     37
    01-May-2016 12:00:00     35
```

Input Arguments

TT1 — Input timetable
timetable

Input timetable.

n — Number of time steps to shift data in regular timetable
integer

Number of time steps to shift the data in a regular timetable, specified as an integer.

dt — Time interval to shift data in regular timetable
duration | calendar duration

Time interval to shift the data in a regular timetable, specified as a duration or calendar duration.

See Also

`horzcat` | `innerjoin` | `outerjoin` | `retime` | `synchronize` | `vertcat`

Topics

"Create Timetables"
"Clean Timetable with Missing, Duplicate, or Nonuniform Times"
"Select Timetable Data by Row Time and Variable Type"

Introduced in R2016b

lastDirectory

Determine if current IFD is last in file

Syntax

```
tf = lastDirectory(t)
```

Description

`tf = lastDirectory(t)` returns `true` if the current image file directory (IFD) is the last IFD in the TIFF file. Otherwise, the function returns `false`. If the file contains only one image, then the current IFD is the last IFD.

Examples

Determine If Current Directory Is Last Directory

Create a `Tiff` object for a file and determine if the current directory is the last directory in the file. If the current directory is the last directory, then the `lastDirectory` function returns `1`. Otherwise, the function returns `0`.

```
t = Tiff('example.tif','r');
lastDirectory(t)
```

```
ans = logical
   0
```

Set the current directory to directory number `3` and check if it is the last directory in the file.

```
setDirectory(t,3);
lastDirectory(t)
```

```
ans = logical
   1
```

Close the `Tiff` object.

```
close(t);
```

Input Arguments

`t` — Tiff object
`Tiff` object

`Tiff` object representing a TIFF file. Use the `Tiff` function to create the object.

Algorithms

References

This function corresponds to the `TIFFLastDirectory` function in the LibTIFF C API. To use this function, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff` | `setDirectory`

Topics

"Importing Images"
"Exporting to Images"

Introduced in R2009b

Line Properties

Chart line appearance and behavior

Description

`Line` properties control the appearance and behavior of a `Line` object. By changing property values, you can modify certain aspects of the line chart.

Starting in R2014b, you can use dot notation to query and set properties.

```
p = plot(1:10);
c = p.Color;
p.Color = 'red';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Line

`Color` — Line color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`ColorMode` — Control how `Color` is set

`'auto'` (default) | `'manual'`

Control how the `Color` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the `Color` property by selecting a color from the `ColorOrder` property of the axes.
- `'manual'` — You control the value of the `Color` property manually, either by setting the value of the `Color` property directly on the object, or by including a color in the `LineSpec` argument when you call a plotting function.

If you change the value of the `Color` property manually, MATLAB changes the value of the `ColorMode` property to `'manual'`.

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineStyleMode` — Control how `LineStyle` is set
`'auto'` (default) | `'manual'`

Control how the `LineStyle` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the `LineStyle` property by selecting a line style from the `LineStyleOrder` property of the axes.
- `'manual'` — You control the value of the `LineStyle` property manually, either by setting the value of the `LineStyle` property directly on the object, or by specifying the `LineSpec` argument when you call a plotting function.

If you change the value of the `LineStyle` property manually, MATLAB changes the value of the `LineStyleMode` property to `'manual'`.

`LineWidth` — Line width

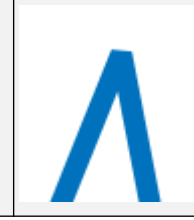
`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

`LineJoin` — Style of line corners

`'round'` (default) | `'miter'` | `'chamfer'`

Style of line corners, specified as `'round'`, `'miter'`, or `'chamfer'`. This table illustrates the appearance of the different values.

`'round'`	`'miter'`	`'chamfer'`
		

The appearance of the `'round'` option might look different if the `Renderer` property of the figure is set to `'opengl'` instead of `'painters'`.

`AlignVertexCenters` — Sharp vertical and horizontal lines

`'off'` (default) | `'on'`

Sharp vertical and horizontal lines, specified as `'off'` or `'on'`.

If the associated figure has a `GraphicsSmoothing` property set to `'on'` and a `Renderer` property set to `'opengl'`, then the figure applies a smoothing technique to plots. In some cases, this smoothing technique can cause vertical and horizontal lines to appear uneven in thickness or color. Use the `AlignVertexCenters` property to eliminate the uneven appearance.

- `'off'` — Do not sharpen vertical or horizontal lines. The lines might appear uneven in thickness or color.
- `'on'` — Sharpen vertical and horizontal lines to eliminate an uneven appearance.

Note You must have a graphics card that supports this feature. To see if the feature is supported, call the `rendererinfo` function. If it is supported, `rendererinfo` returns value of `1` for `info.Details.SupportsAlignVertexCenters`.

Markers

`Marker` — Marker symbol

`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | ...

Marker symbol, specified as one of the values listed in this table. By default, the object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerMode` — Control how `Marker` is set

`'auto'` (default) | `'manual'`

Control how the `Marker` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the object's `Marker` property.

- `'manual'` — You control the value of the `Marker` property manually, either by setting the value of the `Marker` property directly on the object, or by including a marker in the `LineSpec` argument when you call a plotting function.

If you change the value of the `Marker` property manually, MATLAB changes the value of the `MarkerMode` property to `'manual'`.

`MarkerIndices` — Indices of data points at which to display markers
`1:length(YData)` (default) | vector of positive integers | scalar positive integer

Indices of data points at which to display markers, specified as a vector of positive integers. If you do not specify the indices, then MATLAB displays a marker at every data point.

Note To see the markers, you must also specify a marker symbol.

Example: `plot(x,y,'-o','MarkerIndices',[1 5 10])` displays a circle marker at the first, fifth, and tenth data points.

Example: `plot(x,y,'-x','MarkerIndices',1:3:length(y))` displays a cross marker every three data points.

Example: `plot(x,y,'Marker','square','MarkerIndices',5)` displays one square marker at the fifth data point.

`MarkerSize` — Marker size
`6` (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

`MarkerEdgeColor` — Marker outline color
`'auto'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'auto'` uses the same color as the `Color` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` option uses the same color as the `Color` property of the parent axes. If you specify `'auto'` and the axes plot box is invisible, the marker fill color is the color of the figure.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Cartesian Coordinate Data

XData — x values
vector

x values, specified as a vector.

- For 2-D line plots, if you do not specify the *x* values, then MATLAB uses the indices of `YData` as the *x* values for the plot. `XData` and `YData` must have equal lengths.
- For 3-D line plots, if you do not specify the *x* values, then MATLAB uses the indices of `ZData` as the *x* values for the plot. `XData`, `YData`, and `ZData` must have equal lengths.

Example: `[1:10]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

XDataMode — Selection mode for XData
`'auto'` (default) | `'manual'`

Selection mode for `XData`, specified as one of these values:

- `'auto'` — Use the indices of the values in `YData` (or `ZData` for 3-D plots).
- `'manual'` — Use manually specified values. To specify the values, set the `XData` property or specify the input argument `X` to the plotting function.

XDataSource — Variable linked to XData
`''` (default) | character vector | string

Variable linked to `XData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `XData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `XData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'x'`

`YData` — y values
vector

y values, specified as a vector. For 2-D line plots, `XData` and `YData` must have equal lengths. For 3-D line plots, `XData`, `YData`, and `ZData` must have equal lengths.

Example: `[1:10]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`YDataSource` — Variable linked to `YData`
`''` (default) | character vector | string

Variable linked to `YData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `YData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `YData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'y'`

`ZData` — z values
vector

z values for the 3-D line plot, specified as a vector. `XData`, `YData`, and `ZData` must have equal lengths.

Example: `[1:10]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`ZDataSource` — Variable linked to `ZData`

`''` (default) | character vector | string

Variable linked to `ZData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `ZData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `ZData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'z'`

Polar Coordinate Data

`ThetaData` — Angle values

vector

Angle values, specified as a vector. `ThetaData` and `RData` must be vectors of equal length.

This property applies only to lines in a polar axes.

`ThetaDataMode` — Selection mode for `ThetaData`

`'auto'` (default) | `'manual'`

Selection mode for the `ThetaData` property value, specified as one of these values:

- `'auto'` — Automatically select the value.

- `'manual'` — Use the specified values. To specify the value, set the `ThetaData` property or specify the input argument `theta` to the plotting function.

This property applies only to lines in a polar axes.

`ThetaDataSource` — Variable linked to `ThetaData`
`''` (default) | character vector containing MATLAB workspace variable name

Variable linked to `ThetaData`, specified as a character vector containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `RData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `ThetaData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

This property applies only to lines in a polar axes.

`RData` — Radius values
vector

Radius values, specified as a vector. `ThetaData` and `RData` must be vectors of equal length.

This property applies only to lines in a polar axes.

`RDataSource` — Variable linked to `RData`
`''` (default) | character vector containing MATLAB workspace variable name

Variable linked to `RData`, specified as a character vector containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `RData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `RData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

This property applies only to lines in a polar axes.

Geographic Coordinate Data

`LatitudeData` — Latitude values
vector

Latitude values, specified as a vector. `LatitudeData` and `LongitudeData` must be vectors of equal length.

This property applies only to lines in a geographic axes.

`LatitudeDataSource` — Variable linked to `LatitudeData`
`''` (default) | character vector containing MATLAB workspace variable name

Variable linked to `LatitudeData`, specified as a character vector containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate `LatitudeData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `LatitudeData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

This property applies only to lines in a geographic axes.

`LongitudeData` — Longitude values
vector

Longitude values, specified as a vector. `LongitudeData` and `LatitudeData` must be vectors of equal length.

This property applies only to lines in a geographic axes.

`LongitudeDataSource` — Variable linked to `LongitudeData`

`''` (default) | character vector containing MATLAB workspace variable name

Variable linked to `LongitudeData`, specified as a character vector containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate `LongitudeData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `LongitudeData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

This property applies only to lines in a geographic axes.

Legend

`DisplayName` — Legend label

`''` (default) | character vector | string scalar

Legend label, specified as a character vector or string scalar. The legend does not display until you call the `legend` command. If you do not specify the text, then `legend` sets the label using the form `'dataN'`.

`Annotation` — Control for including or excluding object from legend

`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`DataTipTemplate` — Data tip content

`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

`Clipping` — Clipping of object to axes limits

`'on'` (default) | `'off'`

Clipping of the object to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the object that are outside the axes limits.
- `'off'` — Display the entire object, even if parts of it appear outside the axes limits. Parts of the object might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the object so that it is larger than the original plot.

The `Clipping` property of the axes that contains the object must be set to `'on'`. Otherwise, this property has no effect. For more information about the clipping behavior, see the `Clipping` property of the axes.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see "Callback Definition".

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks when visible. The `Visible` property must be set to `'on'` and you must click a part of the `Line` object that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The `HitTest` property determines if the `Line` object responds to the click or if an ancestor does.
- `'all'` — Capture mouse clicks regardless of visibility. The `Visible` property can be set to `'on'` or `'off'` and you can click a part of the `Line` object that has no color. The `HitTest` property determines if the `Line` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `Line` object passes the click through it to the object below it in the current view of the figure window. The `HitTest` property has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `Line` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `Line` object that has one of these:

- `HitTest` property set to `'on'`
- `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

Note The `PickableParts` property determines if the `Line` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent

`Axes` object | `PolarAxes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `PolarAxes`, `Group`, or `Transform` object.

`Children` — Children

empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the chart object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'line'`

This property is read-only.

Type of graphics object, returned as `'line'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`loglog` | `plot` | `plot3` | `polarplot` | `semilogx` | `semilogy`

Topics

“Access Property Values”
“Graphics Object Properties”

Introduced before R2006a

FunctionLine Properties

Line chart appearance and behavior

Description

`FunctionLine` properties control the appearance and behavior of a `FunctionLine` object. By changing property values, you can modify certain aspects of the line chart. You can use dot notation to refer to a particular object and property:

```
fp = fplot(@(x) sin(x))
ls = fp.LineStyle
fp.LineStyle = ':'
```

Properties

Line

`Color` — Line color

`[0 0 1]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`ColorMode` — Control how `Color` is set
`'auto'` (default) | `'manual'`

Control how the `Color` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the `Color` property by selecting a color from the `ColorOrder` property of the axes.
- `'manual'` — You control the value of the `Color` property manually, either by setting the value of the `Color` property directly on the object, or by including a color in the `LineSpec` argument when you call a plotting function.

If you change the value of the `Color` property manually, MATLAB changes the value of the `ColorMode` property to `'manual'`.

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineStyleMode` — Control how `LineStyle` is set
`'auto'` (default) | `'manual'`

Control how the `LineStyle` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the `LineStyle` property by selecting a line style from the `LineStyleOrder` property of the axes.
- `'manual'` — You control the value of the `LineStyle` property manually, either by setting the value of the `LineStyle` property directly on the object, or by specifying the `LineSpec` argument when you call a plotting function.

If you change the value of the `LineStyle` property manually, MATLAB changes the value of the `LineStyleMode` property to `'manual'`.

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

Markers

`Marker` — Marker symbol

`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | `'x'` | `...`

Marker symbol, specified as one of the markers listed in this table. By default, the line does not have markers. Specifying a marker symbol adds markers at selected points along the line.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'+'`

Example: `'diamond'`

`MarkerMode` — Control how `Marker` is set

`'auto'` (default) | `'manual'`

Control how the `Marker` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the object's `Marker` property.
- `'manual'` — You control the value of the `Marker` property manually, either by setting the value of the `Marker` property directly on the object, or by including a marker in the `LineSpec` argument when you call a plotting function.

If you change the value of the `Marker` property manually, MATLAB changes the value of the `MarkerMode` property to `'manual'`.

`MarkerSize` — Marker size
6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

`MarkerEdgeColor` — Marker outline color
`'auto'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'auto'` uses the same color as the `Color` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` value uses the same color as the `MarkerEdgeColor` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to

F. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Example: `'#D2F9A7'`

Function

`Function` — Function to plot
function handle | anonymous function | symbolic expression | symbolic function

Function to plot, specified as a function handle, anonymous function, or a symbolic expression or function.

`MeshDensity` — Number of evaluation points
23 (default) | number

Number of evaluation points, specified as a number. The default is `23`. Because `FunctionLine` uses adaptive evaluation, the actual number of evaluation points is greater.

`ShowPoles` — Display asymptotes at poles
`'on'` (default) | `'off'`

Display asymptotes at poles, specified as `'on'` (default) or `'off'`. The asymptotes display as gray, dashed vertical lines. `fplot` displays asymptotes only with the `fplot(f)` syntax or variants, and not with the `fplot(xt,yt)` syntax.

`XRange` — Plotting interval for *x* values
`[-5 5]` (default) | two-element vector of form `[xmin xmax]`

Plotting interval for *x* values, specified as a two-element vector of the form `[xmin xmax]`.

`XRangeMode` — Selection mode for `XRange`
`'auto'` (default) | `'manual'`

Selection mode for `XRange`, specified as one of these values:

- `'auto'` — Use the default value `[-5 5]`. If axes limits are specified, follow the specified limits instead.
- `'manual'` — Use manually specified values. To specify the values, set the `XRange` property.

Data

`XData` — *x* values
vector

This property is read-only.

x values, specified as a vector. `XData`, `YData`, and `ZData` have equal lengths.

`YData` — *y* values
vector

This property is read-only.

y values, specified as a vector. `XData`, `YData`, and `ZData` have equal lengths.

`ZData` — *z* values
vector of zeros

This property is read-only.

z values, returned as a vector of zeros. `XData`, `YData`, and `ZData` have equal lengths.

Legend

`DisplayName` — Text for legend label
autogenerated label (default) | character vector | string

Text for legend label, specified as a custom character vector or string. The default label is autogenerated from the `Function` property and the `texlabel` function. The legend does not appear until you call the `legend` function.

Data Types: `char` | `string`

`Annotation` — Control for including or excluding object from legend
`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`DataTipTemplate` — Data tip content

`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

`Clipping` — Clipping of object to axes limits

`'on'` (default) | `'off'`

Clipping of the object to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the object that are outside the axes limits.
- `'off'` — Display the entire object, even if parts of it appear outside the axes limits. Parts of the object might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the object so that it is larger than the original plot.

The `Clipping` property of the axes that contains the object must be set to `'on'`. Otherwise, this property has no effect. For more information about the clipping behavior, see the `Clipping` property of the axes.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see "Callback Definition".

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks when visible. The `Visible` property must be set to `'on'` and you must click a part of the `FunctionLine` object that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The `HitTest` property determines if the `FunctionLine` object responds to the click or if an ancestor does.
- `'all'` — Capture mouse clicks regardless of visibility. The `Visible` property can be set to `'on'` or `'off'` and you can click a part of the `FunctionLine` object that has no color. The `HitTest` property determines if the `FunctionLine` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `FunctionLine` object passes the click through it to the object below it in the current view of the figure window. The `HitTest` property has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `FunctionLine` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `FunctionLine` object that has one of these:

- `HitTest` property set to `'on'`
- `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

Note The `PickableParts` property determines if the `FunctionLine` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent

`Axes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `Group`, or `Transform` object.

`Children` — Children

empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the chart object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'functionline'`

This property is read-only.

Type of graphics object, returned as `'functionline'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`fplot`

Topics

“Access Property Values”
“Graphics Object Properties”

Introduced in R2016a

FunctionContour Properties

Contour chart appearance and behavior

Description

`FunctionContour` properties control the appearance and behavior of `FunctionContour` objects. By changing property values, you can modify certain aspects of the contour chart.

Properties

Levels

`LevelList` — Contour levels
vector of z values

Contour levels, specified as a vector of z values. By default, the `fcontour` function chooses values that span the range of values in the `ZData` property.

Setting this property sets the associated mode property to manual.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`LevelListMode` — Selection mode for `LevelList`
`'auto'` (default) | `'manual'`

Selection mode for the `LevelList`, specified as one of these values:

- `'auto'` — Determine the values based on the `ZData` values.
- `'manual'` — Use manually specified values. To specify the values, set the `LevelList` property. When the mode is `'manual'`, the `LevelList` values do not change if you change the `Function` property or the limits.

`LevelStep` — Spacing between contour lines
`0` (default) | scalar numeric value

Spacing between contour lines, specified as a scalar numeric value. For example, specify a value of 2 to draw contour lines at increments of 2. By default, `LevelStep` is determined by using the `ZData` values.

Setting this property sets the associated mode property to `'manual'`.

Example: `3.4`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`LevelStepMode` — Selection mode for `LevelStep`

`'auto'` (default) | `'manual'`

Selection mode for the `LevelStep`, specified as one of these values:

- `'auto'` — Determine the value based on the `ZData` values.
- `'manual'` — Use a manually specified value. To specify the value, set the `LevelStep` property. When the mode is `'manual'`, the value of `LevelStepMode` does not change when the `Function` property or the limits change.

Color and Styling

`Fill` — Fill between contour lines

`'off'` (default) | `'on'`

Fill between contour lines, specified as one of these values:

- `'off'` — Do not fill the spaces between contour lines with a color.
- `'on'` — Fill the spaces between contour lines with color.

`LineColor` — Color of contour lines

`'flat'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of contour lines, specified as `'flat'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. To use a different color for each contour line, specify `'flat'`. The color is determined by the contour value of the line, the colormap, and the scaling of data values into the colormap. For more information on color scaling, see `caxis`.

To use the same color for all the contour lines, specify an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width

`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

Function

`Function` — Function to plot

function handle | anonymous function | symbolic expression | symbolic function

Function to plot, specified as a function handle, anonymous function, or a symbolic expression or function.

XRange — Plotting interval for *x* values

`[-5 5]` (default) | two-element vector of form `[xmin xmax]`

Plotting interval for *x* values, specified as a two-element vector of the form `[xmin xmax]`.

XRangeMode — Selection mode for XRange

`'auto'` (default) | `'manual'`

Selection mode for `XRange`, specified as one of these values:

- `'auto'` — Use the default value `[-5 5]`. If axes limits are specified, follow the specified limits instead.
- `'manual'` — Use manually specified values. To specify the values, set the `XRange` property.

YRange — Plotting interval for *y* values

`[−5 5]` (default) | two-element vector of form `[ymin ymax]`

Plotting interval for *y* values, specified as a two-element vector of the form `[ymin ymax]`.

YRangeMode — Selection mode for YRange

`'auto'` (default) | `'manual'`

Selection mode for `YRange`, specified as one of these values:

- `'auto'` — Use the default value `[-5 5]`. If the axes limits are specified, follow the specified limits instead.
- `'manual'` — Use manually specified values. To specify the values, set the `YRange` property.

MeshDensity — Number of evaluation points per direction

71 (default) | number

Number of evaluation points per direction, specified as a number. The default is `71`. Because `fcontour` uses adaptive evaluation, the actual number of evaluation points is greater.

Example: `30`

Data

ContourMatrix — Contour line definitions

`[]` (default) | two-row matrix

This property is read-only.

Contour line definitions, returned as a two-row matrix. Each contour line in the plot has an associated definition. If the plot has a total of N contour lines, then the contour matrix consists of N definitions:

```
C = [C(1) C(2)...C(k)...C(N)]
```

Each contour line definition follows this pattern:

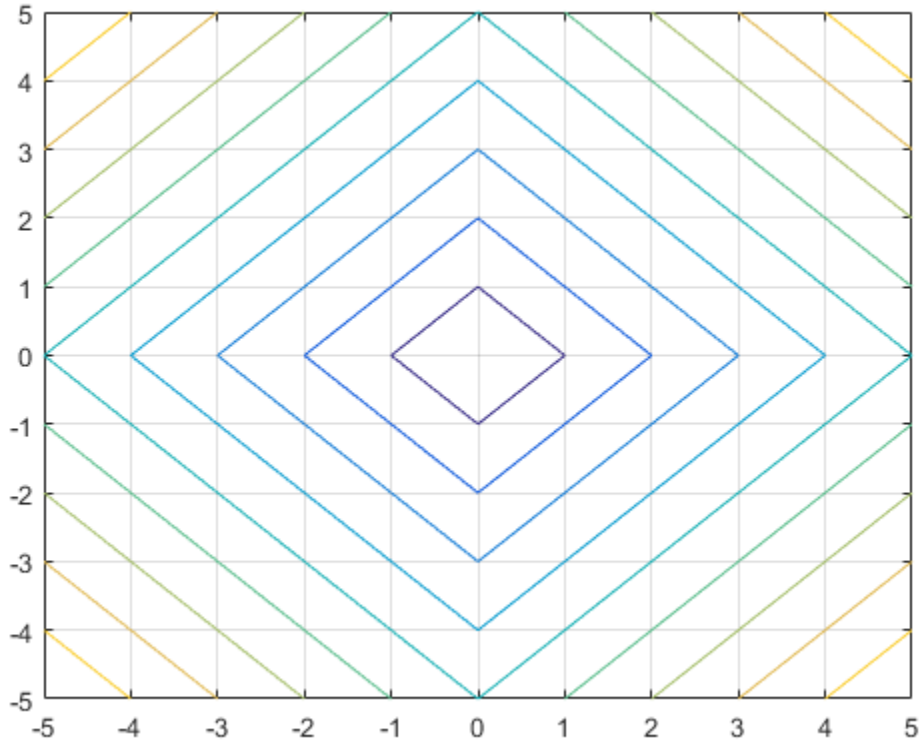
```
C(k) = [level   x(1) x(2)...
        numxy   y(1) y(2)... ]
```

The first entry, `level`, indicates the contour level where the contour line is drawn. Beneath the contour level is the number of (*x*,*y*) vertices that define the contour line. The remaining columns contain the data for each of the vertices. If the first and last vertices are the same, then the contour line is a closed loop. If a particular contour level has multiple contour lines in the graph, then the matrix contains a separate definition for each line.

Example

Create a contour plot of `x^2 + y^2`. For demonstration purposes, use the lowest `MeshDensity` that results in a plot, which is `3`.

```
h = fcontour(@(x,y) x.^2+y.^2, 'MeshDensity', 3);
grid on
```



Access the contour matrix using the `ContourMatrix` property of the contour object (`h.ContourMatrix`). The contour matrix contains definitions for each of the seven contour lines. The circles in this matrix show the beginnings of the contour line definitions.

```
>> h.ContourMatrix
ans =
  Columns 1 through 13
    5     1     0    -1     0     1    10     2     0    -2     0     2    15
    5     0    -1     0     1     0     5     0    -2     0     2     0     5
  Columns 14 through 26
    3     0    -3     0     3    20     4     0    -4     0     4    25     5
    0    -3     0     3     0     5     0    -4     0     4     0     5     0
  Columns 27 through 39
    0    -5     0     5    30    -1    -5    30     5     1    30    -5    -1
   -5     0     5     0     2    -5    -1     2    -1    -5     2     1     5
  Columns 40 through 52
   30     1     5    35    -2    -5    35     5     2    35    -5    -2    35
    2     5     1     2    -5    -2     2    -2    -5     2     2     5     2
  Columns 53 through 65
    2     5    40    -3    -5    40     5     3    40    -5    -3    40     3
    5     2     2    -5    -3     2    -3    -5     2     3     5     2     5
  Columns 66 through 78
    5    45    -4    -5    45     5     4    45    -5    -4    45     4     5
    3     2    -5    -4     2    -4    -5     2     4     5     2     5     4
  Columns 79 through 90
   50    -5    -5    50     5     5    50    -5    -5    50     5     5
    2    -5    -5     2    -5    -5     2     5     5     2     5     5
```

The first definition in the matrix indicates that there is a contour line drawn at the `5` level , consisting of the five vertices `(1,0)`, `(0,-1)`, `(-1,0)`, `(0,1)`, and `(1,0)`. Since the first and last vertices are the same, the contour line is a closed loop. The last definition indicates that there is a point at the `50` level because the line starts and ends at the same point with no intervening points.

`XData` — x values
matrix

This property is read-only.

x values specified as a matrix. `XData` is at least a 2-by-2 matrix. `size(XData)`, `size(YData)`, and `size(ZData)` are equal.

`YData` — y values
matrix

This property is read-only.

y values, specified as a matrix. `YData` is at least a 2-by-2 matrix. `size(XData)`, `size(YData)`, and `size(ZData)` are equal.

`ZData` — Data that defines surface to contour
matrix

This property is read-only.

Data that defines the surface to contour, specified as a matrix. `ZData` is at least a 2-by-2 matrix. `size(XData)`, `size(YData)`, and `size(ZData)` are equal.

Legend

`DisplayName` — Text for legend label
autogenerated label (default) | character vector | string

Text for legend label, specified as a custom character vector or string. The default label is autogenerated from the `Function` property and the `texlabel` function. The legend does not appear until you call the `legend` function.

Data Types: `char` | `string`

`Annotation` — Control for including or excluding object from legend
`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do

appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`DataTipTemplate` — Data tip content

`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.

- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

PickableParts — Ability to capture mouse clicks

`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks only when visible. The `Visible` property must be set to `'on'`. The `HitTest` property determines if the `FunctionContour` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `FunctionContour` object passes the click to the object behind it in the current view of the figure window. The `HitTest` property of the `FunctionContour` object has no effect.

HitTest — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `FunctionContour` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `FunctionContour` object that has one of these:
  - `HitTest` property set to `'on'`
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

---

Note The `PickableParts` property determines if the `FunctionContour` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

BeingDeleted — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

Parent — Parent

Axes object | Group object | Transform object

Parent, specified as an Axes, Group, or Transform object.

Children — Children

empty GraphicsPlaceholder array | DataTip object array

Children, returned as an empty GraphicsPlaceholder array or a DataTip object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the Children property. To add a child to this list, set the Parent property of the DataTip object to the chart object.

HandleVisibility — Visibility of object handle

'on' (default) | 'off' | 'callback'

Visibility of the object handle in the Children property of the parent, specified as one of these values:

- 'on' — Object handle is always visible.
- 'off' — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the HandleVisibility to 'off' to temporarily hide the handle during the execution of that function.
- 'callback' — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the Children property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the get, findobj, gca, gcf, gco, newplot, cla, clf, and close functions.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to 'on' to list all object handles regardless of their HandleVisibility property setting.

Identifiers

Type — Type of graphics object
`'functioncontour'`

This property is read-only.

Type of graphics object, returned as `'functioncontour'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also
`fcontour`

Topics
“Access Property Values”
“Graphics Object Properties”

Introduced in R2016a

FunctionSurface Properties

Surface chart appearance and behavior

Description

`FunctionSurface` properties control the appearance and behavior of `FunctionSurface` objects. By changing property values, you can modify certain aspects of the surface chart. You can use dot notation to refer to a particular object and property:

```
fs = fsurf(@(x,y) sin(x)+cos(y))
alp = fs.FaceAlpha
fs.FaceAlpha = 0.6
```

Properties

Faces

`FaceColor` — Face color

`'interp'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Face color, specified as `'interp'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'interp'` interpolates the colors based on the `ZData` values.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`FaceAlpha` — Face transparency

1 (default) | scalar in range `[0 1]`

Face transparency, specified as a scalar in the range `[0,1]`. Use uniform transparency across all of the faces. A value of `1` is fully opaque and `0` is completely transparent. Values between `0` and `1` are semitransparent.

Edges

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width

`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

`EdgeColor` — Line color

`[0 0 0]` (default) | `'interp'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as `'interp'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default RGB triplet value of `[0 0 0]` corresponds to black. The `'interp'` value colors the edges based on the `ZData` values.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Markers

`Marker` — Marker symbol

`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | `'x'` | `...`

Marker symbol, specified as one of the markers listed in this table. By default, the object does not display markers. Specifying a marker symbol adds markers at the intersection points of mesh lines.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'+'`

Example: `'diamond'`

`MarkerSize` — Marker size
6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

`MarkerEdgeColor` — Marker outline color
`'auto'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'auto'` uses the same color as the `EdgeColor` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: [0.5 0.5 0.5]

Example: 'blue'

Example: '#D2F9A7'

MarkerFaceColor — Marker fill color

'none' (default) | 'auto' | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...

Marker fill color, specified as 'auto', an RGB triplet, a hexadecimal color code, a color name, or a short name. The 'auto' value uses the same color as the MarkerEdgeColor property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Example: `'#D2F9A7'`

Function

`Function` — Function to plot

function handle | anonymous function | symbolic expression | symbolic function

Function to plot, specified as a function handle, anonymous function, or a symbolic expression or function.

XRange — Plotting interval for *x* values

`[-5 5]` (default) | two-element vector of form `[xmin xmax]`

Plotting interval for *x* values, specified as a two-element vector of the form `[xmin xmax]`.

XRangeMode — Selection mode for XRange

`'auto'` (default) | `'manual'`

Selection mode for `XRange`, specified as one of these values:

- `'auto'` — Use the default value `[-5 5]`. If axes limits are specified, follow the specified limits instead.
- `'manual'` — Use manually specified values. To specify the values, set the `XRange` property.

YRange — Plotting interval for *y* values
`[−5 5]` (default) | two-element vector of form `[ymin ymax]`

Plotting interval for *y* values, specified as a two-element vector of the form `[ymin ymax]`.

YRangeMode — Selection mode for YRange
`'auto'` (default) | `'manual'`

Selection mode for `YRange`, specified as one of these values:

- `'auto'` — Use the default value `[-5 5]`. If the axes limits are specified, follow the specified limits instead.
- `'manual'` — Use manually specified values. To specify the values, set the `YRange` property.

`MeshDensity` — Number of evaluation points per direction
35 (default) | number

Number of evaluation points per direction, specified as a number. The default is `35`. Because `FunctionSurface` objects use adaptive evaluation, the actual number of evaluation points is greater.

Example: `100`

`ShowContours` — Display contour plot under plot
`'off'` (default) | `'on'`

Display contour plot under plot, specified as `'off'` (default) or `'on'`.

Data

`XData` — x values
matrix

This property is read-only.

x values specified as a matrix. `XData` is at least a 2-by-2 matrix. `size(XData)`, `size(YData)`, and `size(ZData)` are equal.

`YData` — y values
matrix

This property is read-only.

y values, specified as a matrix. `YData` is at least a 2-by-2 matrix. `size(XData)`, `size(YData)`, and `size(ZData)` are equal.

`ZData` — Data that defines surface to contour
matrix

This property is read-only.

Data that defines the surface to contour, specified as a matrix. `ZData` is at least a 2-by-2 matrix. `size(XData)`, `size(YData)`, and `size(ZData)` are equal.

Ambient Lighting

`AmbientStrength` — Strength of ambient light
`0.3` (default) | scalar in range `[0,1]`

Strength of ambient light, specified as a scalar value in the range `[0,1]`. Ambient light is a nondirectional light that illuminates the entire scene. There must be at least one visible light object in the axes for the ambient light to be visible.

The `AmbientLightColor` property for the axes sets the color of the ambient light. The color is the same for all objects in the axes.

Example: `0.5`

Data Types: `double`

`DiffuseStrength` — Strength of diffuse light
`0.6` (default) | scalar in range `[0,1]`

Strength of diffuse light, specified as a scalar value in the range `[0,1]`. Diffuse light is the nonspecular reflectance from light objects in the axes.

Example: `0.3`

Data Types: `double`

`SpecularColorReflectance` — Color of specular reflections
1 (default) | scalar in range `[0,1]`

Color of specular reflections, specified as a scalar value in the range `[0,1]`. A value of `1` sets the color using only the color of the light source. A value of `0` sets the color using both the color of the object from which it reflects and the color of the light source. The `Color` property of the light contains the color of the light source. The proportions vary linearly for values in between.

Example: `0.5`

Data Types: `double`

`SpecularExponent` — Size of specular spot
10 (default) | scalar greater than or equal to 1

Size of specular spot, specified as a scalar value greater than or equal to 1. Most materials have exponents in the range `[5 20]`.

Example: `7`

Data Types: `double`

`SpecularStrength` — Strength of specular reflection
`0.9` (default) | scalar in range `[0,1]`

Strength of specular reflection, specified as a scalar value in the range `[0,1]`. Specular reflections are the bright spots on the surface from light objects in the axes.

Example: `0.3`

Data Types: `double`

Legend

`DisplayName` — Text for legend label
autogenerated label (default) | character vector | string

Text for legend label, specified as a custom character vector or string. The default label is autogenerated from the `Function` property and the `texlabel` function. The legend does not appear until you call the `legend` function.

Data Types: `char` | `string`

`Annotation` — Control for including or excluding object from legend
`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`DataTipTemplate` — Data tip content

`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

UIContextMenu — Context menu

empty GraphicsPlaceholder array (default) | ContextMenu object

Context menu, specified as a ContextMenu object. Use this property to display a context menu when you right-click the object. Create the context menu using the uicontextmenu function.

Note If the PickableParts property is set to 'none' or if the HitTest property is set to 'off', then the context menu does not appear.

Selected — Selection state

'off' (default) | 'on'

Selection state, specified as one of these values:

- 'on' — Selected. If you click the object when in plot edit mode, then MATLAB sets its Selected property to 'on'. If the SelectionHighlight property also is set to 'on', then MATLAB displays selection handles around the object.
- 'off' — Not selected.

SelectionHighlight — Display of selection handles

'on' (default) | 'off'

Display of selection handles when selected, specified as one of these values:

- 'on' — Display selection handles when the Selected property is set to 'on'.
- 'off' — Never display selection handles, even when the Selected property is set to 'on'.

Callbacks

ButtonDownFcn — Mouse-click callback

'' (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks when visible. The `Visible` property must be set to `'on'` and you must click a part of the `FunctionSurface` object that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The `HitTest` property determines if the `FunctionSurface` object responds to the click or if an ancestor does.
- `'all'` — Capture mouse clicks regardless of visibility. The `Visible` property can be set to `'on'` or `'off'` and you can click a part of the `FunctionSurface` object that has no color. The `HitTest` property determines if the `FunctionSurface` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `FunctionSurface` object passes the click through it to the object below it in the current view of the figure window. The `HitTest` property has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `FunctionSurface` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `FunctionSurface` object that has one of these:
  - `HitTest` property set to `'on'`

- `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

---

Note The `PickableParts` property determines if the `FunctionSurface` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent

`Axes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `Group`, or `Transform` object.

`Children` — Children

empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the chart object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'functionsurface'`

This property is read-only.

Type of graphics object, returned as `'functionsurface'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`fmesh` | `fsurf`

Topics

“Access Property Values”
“Graphics Object Properties”

Introduced in R2016a

lasterr

Last error message

Note `lasterr` will be removed in a future version. You can obtain information about any error that has been generated by catching an `MException`. See “Capture Information About Exceptions” in the Programming Fundamentals documentation.

Syntax

```
msgstr = lasterr
[msgstr, msgid] = lasterr
lasterr('new_msgstr')
lasterr('new_msgstr', 'new_msgid')
[msgstr, msgid] = lasterr('new_msgstr', 'new_msgid')
```

Description

`msgstr = lasterr` returns the last error message generated by the MATLAB software.

`[msgstr, msgid] = lasterr` returns the last error in `msgstr` and its message identifier in `msgid`. If the error was not defined with an identifier, `lasterr` returns an empty character vector for `msgid`. See “Message Identifiers” in the MATLAB Programming Fundamentals documentation for more information on the `msgid` argument and how to use it.

`lasterr('new_msgstr')` sets the last error message to a new character vector, `new_msgstr`, so that subsequent invocations of `lasterr` return the new error message. You can also set the last error to an empty character vector with `lasterr('')`.

`lasterr('new_msgstr', 'new_msgid')` sets the last error message and its identifier to `new_msgstr` and `new_msgid`, respectively. Subsequent invocations of `lasterr` return the new error message and message identifier.

`[msgstr, msgid] = lasterr('new_msgstr', 'new_msgid')` returns the last error message and its identifier, also changing these values so that subsequent invocations of `lasterr` return the message and identifier specified by `new_msgstr` and `new_msgid` respectively.

Examples

Example 1

Here is a function that examines the `lasterr` character vector and displays its own message based on the error that last occurred. This example deals with two cases, each of which is an error that can result from a matrix multiply:

```
function matrix_multiply(A, B)
try
   A * B
catch
   errmsg = lasterr;
   if(strfind(errmsg, 'Inner matrix dimensions'))
      disp('** Wrong dimensions for matrix multiply')
   else
      if(strfind(errmsg, 'not defined for variables of class'))
         disp('** Both arguments must be double matrices')
      end
   end
end
```

If you call this function with matrices that are incompatible for matrix multiplication (e.g., the column dimension of `A` is not equal to the row dimension of `B`), MATLAB catches the error and uses `lasterr` to determine its source:

```
A = [1  2  3; 6  7  2; 0  -1  5];
B = [9  5  6; 0  4  9];

matrix_multiply(A, B)
** Wrong dimensions for matrix multiply
```

Example 2

Specify a message identifier and error message with `error`:

```
error('MyToolbox:angleTooLarge', ...
      'The angle specified must be less than 90 degrees.');
```

In your error handling code, use `lasterr` to determine the message identifier and error message for the failing operation:

```
[errmsg, msgid] = lasterr
errmsg =
   The angle specified must be less than 90 degrees.
msgid =
   MyToolbox:angleTooLarge
```

See Also

`error` | `lasterror` | `lastwarn` | `rethrow` | `warning`

Introduced before R2006a

lasterror

Last error message and related information

Note `lasterror` will be removed in a future version. You can obtain information about any error that has been generated by catching an `MException`. See “Capture Information About Exceptions” in the Programming Fundamentals documentation.

Syntax

```
s = lasterror
s = lasterror(err)
s = lasterror('reset')
```

Description

`s = lasterror` returns a structure `s` containing information about the most recent error issued by the MATLAB software. The return structure contains the following fields:

Fieldname	Description
`message`	Character array containing the text of the error message.
`identifier`	Character array containing the message identifier of the error message. If the last error issued by MATLAB had no message identifier, then the `identifier` field is an empty character array.
`stack`	Structure providing information on the location of the error. The structure has fields `file`, `name`, and `line`, and is the same as the structure returned by the `dbstack` function. If `lasterror` returns no stack information, `stack` is a 0-by-1 structure having the same three fields.

Note The `lasterror` return structure might contain additional fields in future versions of MATLAB.

The fields of the structure returned in `stack` are

Fieldname	Description
`file`	Name of the file in which the function generating the error appears. This field is the empty character vector if there is no file.
`name`	Name of the function in which the error occurred. If this is the primary function in the file, and the function name differs from the file name, `name` is set to the file name.
`line`	Line number of the file at which the error occurred.

See “Message Identifiers” in the MATLAB Programming Fundamentals documentation for more information on the syntax and usage of message identifiers.

`s = lasterror(err)` sets the last error information to the error message and identifier specified in the structure `err`. Subsequent invocations of `lasterror` return this new error information. The optional return structure `s` contains information on the previous error.

`s = lasterror('reset')` sets the last error information to the default state. In this state, the `message` and `identifier` fields of the return structure are empty character vectors, and the `stack` field is a 0-by-1 structure.

Examples

Example 1

Save the following MATLAB code in a file called `average.m`:

```
function y = average(x)
% AVERAGE Mean of vector elements.
% AVERAGE(X), where X is a vector, is the mean of vector elements.
% Nonvector input results in an error.
check_inputs(x)
y = sum(x)/length(x);      % The actual computation

function check_inputs(x)
[m,n] = size(x);
if (~((m == 1) || (n == 1)) || (m == 1 && n == 1))
    error('AVG:NotAVector', 'Input must be a vector.')
end
```

Now run the function. Because this function requires vector input, passing a scalar value to it forces an error. The error occurs in subroutine `check_inputs`:

```
average(200)
Error using average>check_inputs (line 11)
Input must be a vector.

Error in average (line 5)
check_inputs(x)
```

Get the three fields from `lasterror`:

```
err = lasterror
err =
       message: [1x61 char]
    identifier: 'AVG:NotAVector'
         stack: [2x1 struct]
```

Display the text of the error message:

```
msg = err.message
msg =
    Error using average>check_inputs (line 11)
    Input must be a vector.
```

Display the fields containing the `stack` information. `err.stack` is a 2-by-1 structure because it provides information on the failing subroutine `check_inputs` and also the outer, primary function `average`:

```
st1 = err.stack(1,1)
st1 =
    file: 'd:\matlab_test\average.m'
    name: 'check_inputs'
    line: 11

st2 = err.stack(2,1)
st2 =
    file: 'd:\matlab_test\average.m'
    name: 'average'
    line: 5
```

Note As a rule, the name of your primary function should be the same as the name of the file that contains that function. If these names differ, MATLAB uses the file name in the `name` field of the `stack` structure.

Example 2

`lasterror` is often used in conjunction with the `rethrow` function in `try, catch` statements. For example,

```
try
   do_something
catch
   do_cleanup
   rethrow(lasterror)
end
```

Tips

MathWorks is gradually transitioning MATLAB error handling to an object-oriented scheme that is based on the `MException` class. Although support for `lasterror` is expected to continue, using the static `MException.last` method of `MException` is preferable.

Warning `lasterror` and `MException.last` are not guaranteed to always return identical results. For example, `MException.last` updates its error status only on uncaught errors, where `lasterror` can update its error status on any error, whether it is caught or not.

See Also

`MException` | `MException.last` | `assert` | `dbstack` | `error` | `lastwarn` | `rethrow` | `try, catch`

Introduced before R2006a

lastwarn

Last warning message

Syntax

```
msg = lastwarn
[msg,msgID] = lastwarn
[ ___ ] = lastwarn(newMsg,newID)
```

Description

`msg = lastwarn` returns the last warning message generated by MATLAB, regardless of the display state of the warning.

`[msg,msgID] = lastwarn` also returns the message identifier associated with `msg`.

`[ ___ ] = lastwarn(newMsg,newID)` sets the last warning message and the last warning identifier. Subsequent calls to the `lastwarn` function return the new warning message and, if requested, the new message identifier. You can use this syntax with any of the output arguments of the previous syntaxes.

Examples

Display Last Warning Message

Display a warning message.

```
warning('Message 1.')
```

```
Warning: Message 1.
```

Call the `lastwarn` function to display the last warning message.

```
msg = lastwarn
```

```
msg =

    'Message 1.'
```

Save the current warning settings, and then disable all warnings.

```
origState = warning;
warning('off')
```

Call the `warning` function with a different message. The warning is not displayed.

```
warning('Message 2.')
```

Call the `lastwarn` function. MATLAB returns the last warning message, even though it was not displayed.

```
msg = lastwarn
```

```
msg =

    'Message 2.'
```

Restore the saved warning state.

```
warning(origState)
```

Obtain Identifier from Warning

The warning message identifier is often used to suppress warnings. To find the identifier, use the `lastwarn` function.

Generate a singular matrix warning.

```
A = eye(2);
B = [3 6; 4 8];
C = B\A;
```

```
Warning: Matrix is singular to working precision.
```

Find the identifier of the warning.

```
[msg,msgID] = lastwarn;
```

```
msg =
```

```
    'Matrix is singular to working precision.'


msgID =

    'MATLAB:singularMatrix'
```

Save the current warning state, and disable the specific warning. Perform the original calculation again. This time the warning does not display.

```
warnStruct = warning('off',msgId);
C = B\A;
```

Restore the previous warning state.

```
warning(warnStruct);
```

Input Arguments

newMsg — New message for last warning
character vector | string scalar

New message for last warning, specified as a character vector or string scalar. Subsequent calls to the `lastwarn` function return the new warning message.

You can set the new message to an empty character vector (`''`) or string scalar (`""`).

Example: `'Warning message to display.'`

newID — New identifier for last warning
character vector | string scalar

Identifier for the last warning, specified as a character vector or string scalar. Use the message identifier to help identify the source of the error or to control a selected subset of the errors in your program.

The message identifier includes a `component` and `mnemonic`. The identifier must always contain a colon and follows this simple format: `component:mnemonic`. The `component` and `mnemonic` fields must each begin with a letter. The remaining characters can be alphanumerics (A–Z, a–z, 0–9) and underscores. No whitespace characters can appear anywhere in `newID`. For more information, see “Message Identifiers”.

Example: `'MATLAB:singularMatrix'`

Example: `'MyProject:myFunction:notEnoughInputs'`

Output Arguments

`msg` — Last warning message
character vector

Last warning message generated by MATLAB, regardless of the display state of the warning, returned as a character vector.

`msgID` — Last warning identifier
character vector

Last warning identifier, returned as a character vector. If the warning was not defined with an identifier, `lastwarn` returns an empty character vector for `msgID`.

See Also

`error` | `warning`

Topics

"Suppress Warnings"
"Restore Warnings"

Introduced before R2006a

lcm

Least common multiple

Syntax

```
L = lcm(A,B)
```

Description

`L = lcm(A,B)` returns the least common multiples of the elements of `A` and `B`.

Examples

Least Common Multiples of Double Array and a Scalar

```
A = [5 17; 10 60];
B = 45;
L = lcm(A,B)
```

```
L = 2×2

    45   765
    90   180
```

Least Common Multiples of Unsigned Integers

```
A = uint16([255 511 15]);
B = uint16([15 127 1023]);
L = lcm(A,B)
```

```
L = 1x3 uint16 row vector
```

```
255    64897    5115
```

Input Arguments

A,B — Input values
scalars, vectors, or arrays of real, positive integer values

Input values, specified as scalars, vectors, or arrays of real, positive integer values. A and B can be any numeric type, and they can be of different types within certain limitations:

- If A or B is of type `single`, then the other can be of type `single` or `double`.
- If A or B belongs to an integer class, then the other must belong to the same class or it must be a `double` scalar value.

A and B must be the same size or one must be a scalar.

Example: `[20 3 13],[10 6 7]`

Example: `int16([100 30 200]),int16([20 15 9])`

Example: `int16([100 30 200]),20`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

L — Least common multiple
real, positive integer values

Least common multiple, returned as an array of real positive integer values. L is the same size as A and B, and it has the same type as A and B. If A and B are of different types, then L is returned as the nondouble type.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`gcd`

Introduced before R2006a

ldl

Block LDL' factorization for Hermitian indefinite matrices

Syntax

```
L = ldl(A)
[L,D] = ldl(A)
[L,D,P] = ldl(A)
[L,D,p] = ldl(A,'vector')
[U,D,P] = ldl(A,'upper')
[U,D,p] = ldl(A,'upper','vector')
[L,D,P,S] = ldl(A)
[L,D,P,S] = LDL(A,THRESH)
[U,D,p,S] = LDL(A,THRESH,'upper','vector')
```

Description

`L = ldl(A)` returns only the permuted lower triangular matrix `L` as in the two-output form. The permutation information is lost, as is the block diagonal factor `D`. By default, `ldl` references only the diagonal and lower triangle of `A`, and assumes that the upper triangle is the complex conjugate transpose of the lower triangle. Therefore `[L,D,P] = ldl(TRIL(A))` and `[L,D,P] = ldl(A)`both return the exact same factors. Note, this syntax is not valid for sparse `A`.

`[L,D] = ldl(A)` stores a block diagonal matrix `D` and a permuted lower triangular matrix in `L` such that `A = L*D*L'`. The block diagonal matrix `D` has 1-by-1 and 2-by-2 blocks on its diagonal. Note, this syntax is not valid for sparse `A`.

`[L,D,P] = ldl(A)` returns unit lower triangular matrix `L`, block diagonal `D`, and permutation matrix `P` such that `P'*A*P = L*D*L'`. This is equivalent to `[L,D,P] = ldl(A,'matrix')`.

`[L,D,p] = ldl(A,'vector')` returns the permutation information as a vector, `p`, instead of a matrix. The `p` output is a row vector such that `A(p,p) = L*D*L'`.

`[U,D,P] = ldl(A,'upper')` references only the diagonal and upper triangle of `A` and assumes that the lower triangle is the complex conjugate transpose of the upper triangle.

This syntax returns a unit upper triangular matrix `U` such that `P'*A*P = U'*D*U` (assuming that `A` is Hermitian, and not just upper triangular). Similarly, `[L,D,P] = ldl(A,'lower')` gives the default behavior.

`[U,D,p] = ldl(A,'upper','vector')` returns the permutation information as a vector, `p`, as does `[L,D,p] = ldl(A,'lower','vector')`. `A` must be a full matrix.

`[L,D,P,S] = ldl(A)` returns unit lower triangular matrix `L`, block diagonal `D`, permutation matrix `P`, and scaling matrix `S` such that `P'*S*A*S*P = L*D*L'`. This syntax is only available for real sparse matrices, and only the lower triangle of `A` is referenced. `ldl` uses MA57 for sparse real symmetric `A`.

`[L,D,P,S] = LDL(A,THRESH)` uses `THRESH` as the pivot tolerance in MA57. `THRESH` must be a double scalar lying in the interval `[0, 0.5]`. The default value for `THRESH` is `0.01`. Using smaller values of `THRESH` may give faster factorization times and fewer entries, but may also result in a less stable factorization. This syntax is available only for real sparse matrices.

`[U,D,p,S] = LDL(A,THRESH,'upper','vector')` sets the pivot tolerance and returns upper triangular `U` and permutation vector `p` as described above.

Examples

These examples illustrate the use of the various forms of the `ldl` function, including the one-, two-, and three-output form, and the use of the `vector` and `upper` options. The topics covered are:

- “Example 1 — Two-Output Form of ldl” on page 1-8490
- “Example 2 — Three Output Form of ldl” on page 1-8490
- “Example 3 — The Structure of D” on page 1-8491
- “Example 4 — Using the 'vector' Option” on page 1-8491
- “Example 5 — Using the 'upper' Option” on page 1-8491
- “Example 6 — linsolve and the Hermitian indefinite solver” on page 1-8492

Before running any of these examples, you will need to generate the following positive definite and indefinite Hermitian matrices:

```
A = full(delsq(numgrid('L', 10)));
B = gallery('uniformdata',10,0);
M = [eye(10) B; B' zeros(10)];
```

The structure of M here is very common in optimization and fluid-flow problems, and M is in fact indefinite. Note that the positive definite matrix A must be full, as ldl does not accept sparse arguments.

Example 1 — Two-Output Form of ldl

The two-output form of ldl returns L and D such that A-(L*D*L') is small, L is permuted unit lower triangular, and D is a block 2-by-2 diagonal. Note also that, because A is positive definite, the diagonal of D is all positive:

```
[LA,DA] = ldl(A);
fprintf(1, ...
'The factorization error ||A - LA*DA*LA''|| is %g\n', ...
norm(A - LA*DA*LA'));
neginds = find(diag(DA) < 0)
```

Given a b, solve Ax=b using LA, DA:

```
bA = sum(A,2);
x = LA'\(DA\(LA\bA));
fprintf(...
'The absolute error norm ||x - ones(size(bA))|| is %g\n', ...
norm(x - ones(size(bA))));
```

Example 2 — Three Output Form of ldl

The three output form returns the permutation matrix as well, so that L is in fact unit lower triangular:

```
[Lm, Dm, Pm] = ldl(M);
fprintf(1, ...
'The error norm ||Pm''*M*Pm - Lm*Dm*Lm''|| is %g\n', ...
norm(Pm'*M*Pm - Lm*Dm*Lm'));
fprintf(1, ...
'The difference between Lm and tril(Lm) is %g\n', ...
norm(Lm - tril(Lm)));
```

Given b, solve Mx=b using Lm, Dm, and Pm:

```
bM = sum(M,2);
x = Pm*(Lm'\(Dm\(Lm\(Pm'*bM))));
fprintf(...
'The absolute error norm ||x - ones(size(b))|| is %g\n', ...
norm(x - ones(size(bM))));
```

Example 3 — The Structure of D

D is a block diagonal matrix with 1-by-1 blocks and 2-by-2 blocks. That makes it a special case of a tridiagonal matrix. When the input matrix is positive definite, D is almost always diagonal (depending on how definite the matrix is). When the matrix is indefinite however, D may be diagonal or it may express the block structure. For example, with A as above, DA is diagonal. But if you shift A just a bit, you end up with an indefinite matrix, and then you can compute a D that has the block structure.

```
figure; spy(DA); title('Structure of D from ldl(A)');
[Las, Das] = ldl(A - 4*eye(size(A)));
figure; spy(Das);
title('Structure of D from ldl(A - 4*eye(size(A)))');
```

Example 4 — Using the 'vector' Option

Like the `lu` function, `ldl` accepts an argument that determines whether the function returns a permutation vector or permutation matrix. `ldl` returns the latter by default. When you select `'vector'`, the function executes faster and uses less memory. For this reason, specifying the `'vector'` option is recommended. Another thing to note is that indexing is typically faster than multiplying for this kind of operation:

```
[Lm, Dm, pm] = ldl(M, 'vector');
fprintf(1, 'The error norm ||M(pm,pm) - Lm*Dm*Lm''|| is %g\n', ...
  norm(M(pm,pm) - Lm*Dm*Lm'));

% Solve a system with this kind of factorization.
clear x;
x(pm,:) = Lm'\(Dm\(Lm\(bM(pm,:))));
fprintf('The absolute error norm ||x - ones(size(b))|| is %g\n', ...
  norm(x - ones(size(bM))));
```

Example 5 — Using the 'upper' Option

Like the `chol` function, `ldl` accepts an argument that determines which triangle of the input matrix is referenced, and also whether `ldl` returns a lower (L) or upper (L') triangular factor. For dense matrices, there are no real savings with using the upper triangular version instead of the lower triangular version:

```
Ml = tril(M);
[Lml, Dml, Pml] = ldl(Ml, 'lower'); % 'lower' is default behavior.
fprintf(1, ...
'The difference between Lml and Lm is %g\n', norm(Lml - Lm));
[Umu, Dmu, pmu] = ldl(triu(M), 'upper', 'vector');
fprintf(1, ...
'The difference between Umu and Lm'' is %g\n', norm(Umu - Lm'));

% Solve a system using this factorization.
clear x;
x(pm,:) = Umu\(Dmu\(Umu'\(bM(pmu,:))));
fprintf(...
'The absolute error norm ||x - ones(size(b))|| is %g\n', ...
norm(x - ones(size(bM))));
```

When specifying both the `'upper'` and `'vector'` options, `'upper'` must precede `'vector'` in the argument list.

Example 6 — linsolve and the Hermitian indefinite solver

When using the `linsolve` function, you may experience better performance by exploiting the knowledge that a system has a symmetric matrix. The matrices used in the examples above are a bit small to see this so, for this example, generate a larger matrix. The matrix here is symmetric positive definite, and below we will see that with each bit of knowledge about the matrix, there is a corresponding speedup. That is, the symmetric solver is faster than the general solver while the symmetric positive definite solver is faster than the symmetric solver:

```
Abig = full(delsq(numgrid('L', 30)));
bbig = sum(Abig, 2);
LSopts.POSDEF = false;
LSopts.SYM = false;
tic; linsolve(Abig, bbig, LSopts); toc;
LSopts.SYM = true;
tic; linsolve(Abig, bbig, LSopts); toc;
LSopts.POSDEF = true;
tic; linsolve(Abig, bbig, LSopts); toc;
```

Algorithms

`ldl` uses the MA57 routines in the Harwell Subroutine Library (HSL) for real sparse matrices.

References

[1] Ashcraft, C., R.G. Grimes, and J.G. Lewis. “Accurate Symmetric Indefinite Linear Equations Solvers.” *SIAM J. Matrix Anal. Appl.* Vol. 20. Number 2, 1998, pp. 513–561.

[2] Duff, I. S. "MA57 — A new code for the solution of sparse symmetric definite and indefinite systems." Technical Report RAL-TR-2002-024, Rutherford Appleton Laboratory, 2002.

See Also

`chol` | `lu` | `qr`

ldivide, .\

Left array division

Syntax

```
x = B.\A
x = ldivide(B,A)
```

Description

`x = B.\A` divides each element of `A` by the corresponding element of `B`. The sizes of `A` and `B` must be the same or be compatible.

If the sizes of `A` and `B` are compatible, then the two arrays implicitly expand to match each other. For example, if one of `A` or `B` is a scalar, then the scalar is combined with each element of the other array. Also, vectors with different orientations (one row vector and one column vector) implicitly expand to form a matrix.

`x = ldivide(B,A)` is an alternative way to divide `A` by `B`, but is rarely used. It enables operator overloading for classes.

Examples

Divide Two Numeric Arrays

```
A = ones(2,3);
B = [1 2 3; 4 5 6];
x = B.\A
```

```
x = 2×3

    1.0000    0.5000    0.3333
    0.2500    0.2000    0.1667
```

Divide a Scalar by a Numeric Array

```
C = 2;
D = [1 2 3; 4 5 6];
x = D.\C
```

```
x = 2×3

    2.0000    1.0000    0.6667
    0.5000    0.4000    0.3333
```

Divide Row and Column Vectors

Create a 1-by-2 row vector and 3-by-1 column vector and divide them.

```
a = 1:2;
b = (1:3)';
b .\ a
```

```
ans = 3×2

    1.0000    2.0000
    0.5000    1.0000
    0.3333    0.6667
```

The result is a 3-by-2 matrix, where each (i,j) element in the matrix is equal to `b(i) .\ a(j)`:

$$a = [a_1\, a_2], \qquad b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}, \qquad b\ .\backslash\, a = \begin{bmatrix} b_1\ .\backslash\, a_1 & b_1\ .\backslash\, a_2 \\ b_2\ .\backslash\, a_1 & b_2\ .\backslash\, a_2 \\ b_3\ .\backslash\, a_1 & b_3\ .\backslash\, a_2 \end{bmatrix}.$$

Input Arguments

A, B — Operands
scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Numeric inputs A and B must either be the same size or have sizes that are compatible (for example, A is an M-by-N matrix and B is a scalar or 1-by-N row vector). For more information, see “Compatible Array Sizes for Basic Operations”.

- If A or B is an integer data type, then the other input must be the same integer type or be a scalar double. Operands with an integer data type cannot be complex.
- If A and B are duration arrays, then they must be the same size unless one is a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `duration` | `char`

Tips

- The element-wise operators `./` and `.\` are related to each other by the equation `A./B = B.\A`.
- When dividing integers, use `idivide` for more rounding options.
- MATLAB does not support complex integer division.

Compatibility Considerations

Implicit expansion change affects arguments for operators

Behavior changed in R2016b

Starting in R2016b with the addition of implicit expansion, some combinations of arguments for basic operations that previously returned errors now produce results. For example, you previously could not add a row and a column vector, but those operands are now valid for addition. In other words, an expression like `[1 2] + [1; 2]` previously returned a size mismatch error, but now it executes.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a `try/catch` block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see "Compatible Array Sizes for Basic Operations".

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see "Tall Arrays".

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- 64-bit integers are not supported.

For more information, see "Run MATLAB Functions on a GPU" (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`idivide` | `mldivide` | `mrdivide` | `rdivide`

Topics

“Array vs. Matrix Operations”
“Operator Precedence”
“MATLAB Operators and Special Characters”

Introduced before R2006a

le, <=

Determine less than or equal to

Syntax

```
A <= B
le(A,B)
```

Description

`A <= B` returns a logical array with elements set to logical 1 (`true`) where `A` is less than or equal to `B`; otherwise, the element is logical `0` (`false`). The test compares only the real part of numeric arrays. `le` returns logical `0` (`false`) where `A` or `B` have NaN or undefined categorical elements.

`le(A,B)` is an alternate way to execute `A <= B`, but is rarely used. It enables operator overloading for classes.

Examples

Test Vector Elements

Find which vector elements are less than or equal to a given value.

Create a numeric vector.

```
A = [1 12 18 7 9 11 2 15];
```

Test the vector for elements that are less than or equal to `12`.

```
A <= 12
```

```
ans = 1x8 logical array
```

```
   1   1   0   1   1   1   1   0
```

The result is a vector with values of logical `1` (`true`) where the elements of `A` satisfy the expression.

Use the vector of logical values as an index to view the values in `A` that are less than or equal to `12`.

```
A(A <= 12)
```

```
ans = 1×6

     1    12     7     9    11     2
```

The result is a subset of the elements in `A`.

Replace Elements of Matrix

Create a matrix.

```
A = magic(4)
```

```
A = 4×4

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Replace all values less than or equal to `9` with the value `10`.

```
A(A <= 9) = 10
```

```
A = 4×4

    16    10    10    13
    10    11    10    10
    10    10    10    12
    10    14    15    10
```

The result is a new matrix whose smallest element is `10`.

Compare Values in Categorical Array

Create an ordinal categorical array.

```
A = categorical({'large' 'medium' 'small'; 'medium' ...
'small' 'large'},{'small' 'medium' 'large'},'Ordinal',1)
```

```
A = 2x3 categorical array
     large       medium      small
     medium      small       large
```

The array has three categories: `'small'`, `'medium'`, and `'large'`.

Find all values less than or equal to the category `'medium'`.

```
A <= 'medium'
```

```
ans = 2x3 logical array

   0   1   1
   1   1   0
```

A value of logical `1` (`true`) indicates a value less than or equal to the category `'medium'`.

Compare the rows of `A`.

```
A(1,:) <= A(2,:)
```

```
ans = 1x3 logical array

   0   0   1
```

The function returns logical `1` (`true`) where the first row has a category value less than or equal to the second row.

Test Complex Numbers

Create a vector of complex numbers.

```
A = [1+i 2-2i 1+3i 1-2i 5-i];
```

Find the values that are less than or equal to 3.

```
A(A <= 3)
```

```
ans = 1×4 complex

   1.0000 + 1.0000i   2.0000 - 2.0000i   1.0000 + 3.0000i   1.0000 - 2.0000i
```

`le` compares only the real part of the elements in `A`.

Use `abs` to find which elements are within a radius of 3 from the origin.

```
A(abs(A) <= 3)
```

```
ans = 1×3 complex

   1.0000 + 1.0000i   2.0000 - 2.0000i   1.0000 - 2.0000i
```

The result has one less element. The element `1.0000 + 3.0000i` is not within a radius of 3 from the origin.

Test Duration Values

Create a `duration` array.

```
d = hours(21:25) + minutes(75)
```

```
d = 1x5 duration array
   22.25 hr   23.25 hr   24.25 hr   25.25 hr   26.25 hr
```

Test the array for elements that are less than or equal to one standard day.

```
d <= 1
```

```
ans = 1x5 logical array

   1   1   0   0   0
```

Input Arguments

A, B — Operands
scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Numeric or string inputs A and B must either be the same size or have sizes that are compatible (for example, A is an M-by-N matrix and B is a scalar or 1-by-N row vector). For more information, see “Compatible Array Sizes for Basic Operations”.

You can compare numeric inputs of any type, and the comparison does not suffer loss of precision due to type conversion.

If A and B are categorical, datetime, or duration arrays, then they must be the same size unless one is a scalar.

- If one input is an ordinal categorical array, the other input can be an ordinal categorical array, a cell array of character vectors, or a single character vector. A single character vector expands into a cell array of character vectors of the same size as the other input. If both inputs are ordinal categorical arrays, they must have the same sets of categories, including their order. See “Compare Categorical Array Elements” for more details.
- If one input is a datetime array, the other input can be a datetime array, a character vector, or a cell array of character vectors.
- If one input is a duration array, the other input can be a duration array or a numeric array. The operator treats each numeric value as a number of standard 24-hour days.
- If one input is a string array, the other input can be a string array, a character vector, or a cell array of character vectors. The corresponding elements of A and B are compared lexicographically.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `categorical` | `datetime` | `duration`
Complex Number Support: Yes

Tips

- Some floating-point numbers cannot be represented exactly in binary form. This leads to small differences in results that the <= operator reflects. For more information, see "Avoiding Common Problems with Floating-Point Arithmetic".

Compatibility Considerations

Implicit expansion change affects arguments for operators

Behavior changed in R2016b

Starting in R2016b with the addition of implicit expansion, some combinations of arguments for basic operations that previously returned errors now produce results. For example, you previously could not add a row and a column vector, but those operands are now valid for addition. In other words, an expression like `[1 2] + [1; 2]` previously returned a size mismatch error, but now it executes.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a `try/catch` block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see "Compatible Array Sizes for Basic Operations".

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see "Tall Arrays".

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`eq` | `ge` | `gt` | `lt` | `ne`

Topics

“Array Comparison with Relational Operators”
“Ordinal Categorical Arrays”
“MATLAB Operators and Special Characters”

Introduced before R2006a

legend

Add legend to axes

Syntax

```
legend
legend(label1,...,labelN)
legend(labels)
legend(subset, ___ )
legend(target, ___ )

legend( ___ ,'Location',lcn)
legend( ___ ,'Orientation',ornt)
legend( ___ ,Name,Value)
legend(bkgd)
lgd = legend( ___ )

legend(vsbl)
legend('off')
```

Description

`legend` creates a legend with descriptive labels for each plotted data series. For the labels, the legend uses the text from the `DisplayName` properties of the data series. If the `DisplayName` property is empty, then the legend uses a label of the form `'dataN'`. The legend automatically updates when you add or delete data series from the axes. This command creates a legend for the current axes or chart returned by `gca`. If the current axes are empty, then the legend is empty. If axes do not exist, then this command creates them.

`legend(label1,...,labelN)` sets the legend labels. Specify the labels as a list of character vectors or strings, such as `legend('Jan','Feb','Mar')`.

`legend(labels)` sets the labels using a cell array of character vectors, a string array, or a character matrix, such as `legend({'Jan','Feb','Mar'})`.

`legend(subset, ___ )` only includes items in the legend for the data series listed in `subset`. Specify `subset` as a vector of graphics objects. You can specify `subset` before specifying the labels or with no other input arguments.

`legend(target, ___ )` uses the axes or chart specified by `target` instead of the current axes or chart. Specify the target as the first input argument.

`legend( ___ ,'Location',lcn)` sets the legend location. For example, `'Location','northeast'` positions the legend in the upper right corner of the axes. Specify the location after other input arguments.

`legend( ___ ,'Orientation',ornt)`, where `ornt` is `'horizontal'`, displays the legend items side-by-side. The default for `ornt` is `'vertical'`, which stacks the items vertically.

`legend( ___ ,Name,Value)` sets legend properties using one or more name-value pair arguments. When setting properties, you must specify the labels using a cell array, such as `legend({'A','B'},'FontSize',12)`. If you do not want to specify labels, then include an empty cell array, such as `legend({},'FontSize',12)`.

`legend(bkgd)`, where `bkgd` is `'boxoff'`, removes the legend background and outline. The default for `bkgd` is `'boxon'`, which displays the legend background and outline.

`lgd = legend( ___ )` returns the `Legend` object. Use `lgd` to query and set properties of the legend after it is created. For a list of properties, see Legend.

`legend(vsbl)` controls the visibility of the legend, where `vsbl` is `'hide'`, `'show'`, or `'toggle'`.

`legend('off')` deletes the legend.

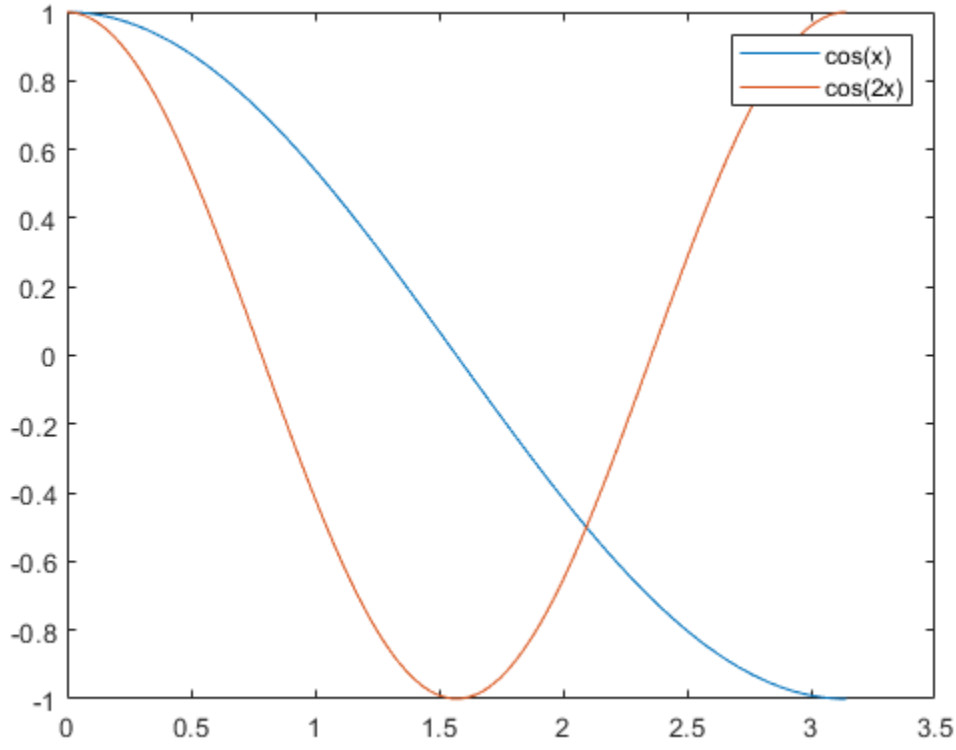
Examples

Add Legend to Current Axes

Plot two lines and add a legend to the current axes. Specify the legend labels as input arguments to the `legend` function.

```
x = linspace(0,pi);
y1 = cos(x);
```

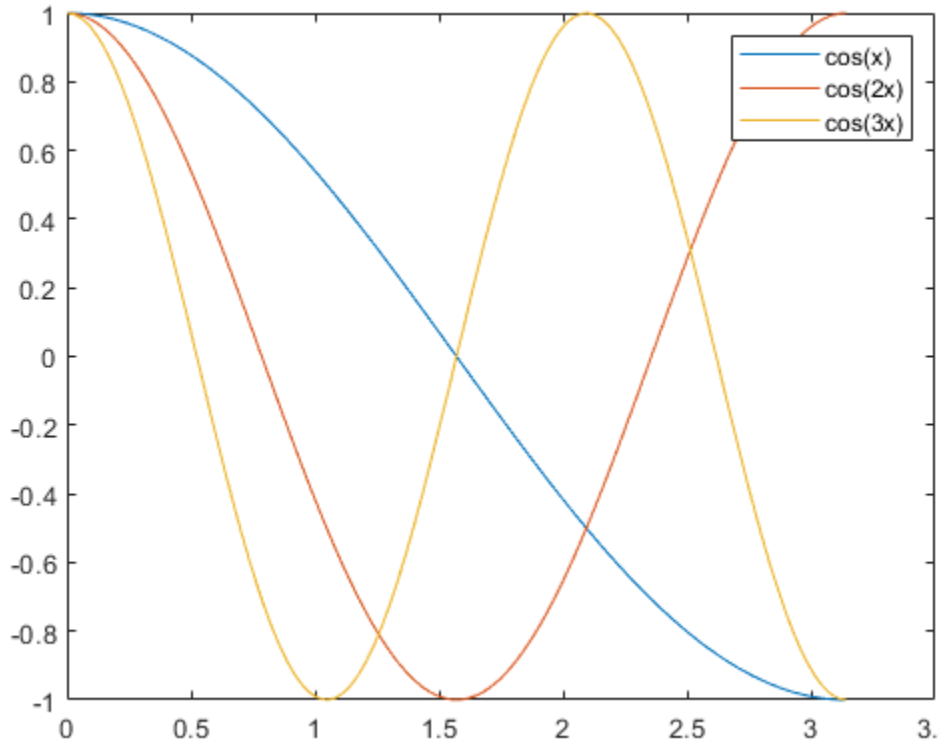
```
plot(x,y1)

hold on
y2 = cos(2*x);
plot(x,y2)

legend('cos(x)','cos(2x)')
```



If you add or delete a data series from the axes, the legend updates accordingly. Control the label for the new data series by setting the `DisplayName` property as a name-value pair during creation. If you do not specify a label, then the legend uses a label of the form `'dataN'`.

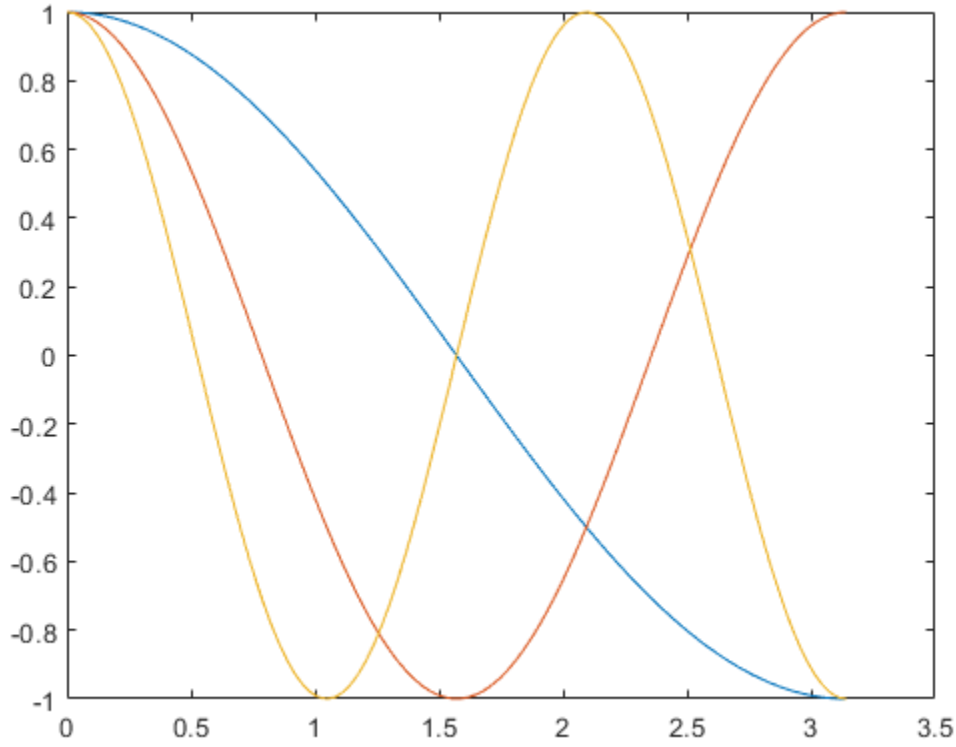
Note: If you do not want the legend to automatically update when data series are added to or removed from the axes, then set the `AutoUpdate` property of the legend to `'off'`.

```
y3 = cos(3*x);
plot(x,y3,'DisplayName','cos(3x)')
hold off
```



Delete the legend.

```
legend('off')
```

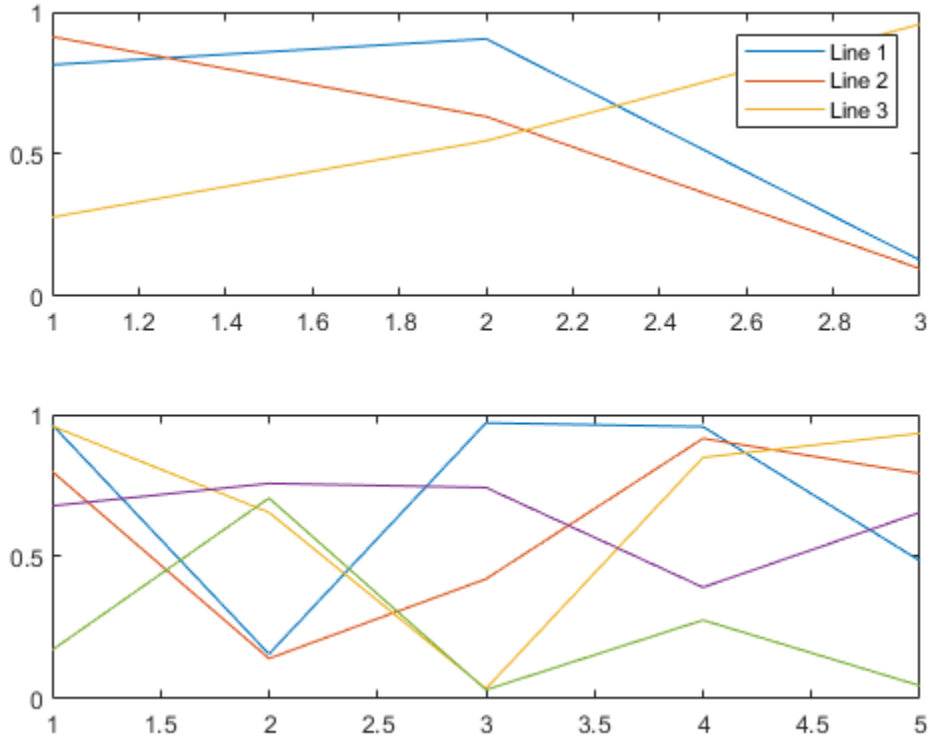


Add Legend to Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Plot random data in each axes. Add a legend to the upper plot by specifying `ax1` as the first input argument to `legend`.

```
tiledlayout(2,1)
y1 = rand(3);
ax1 = nexttile;
```

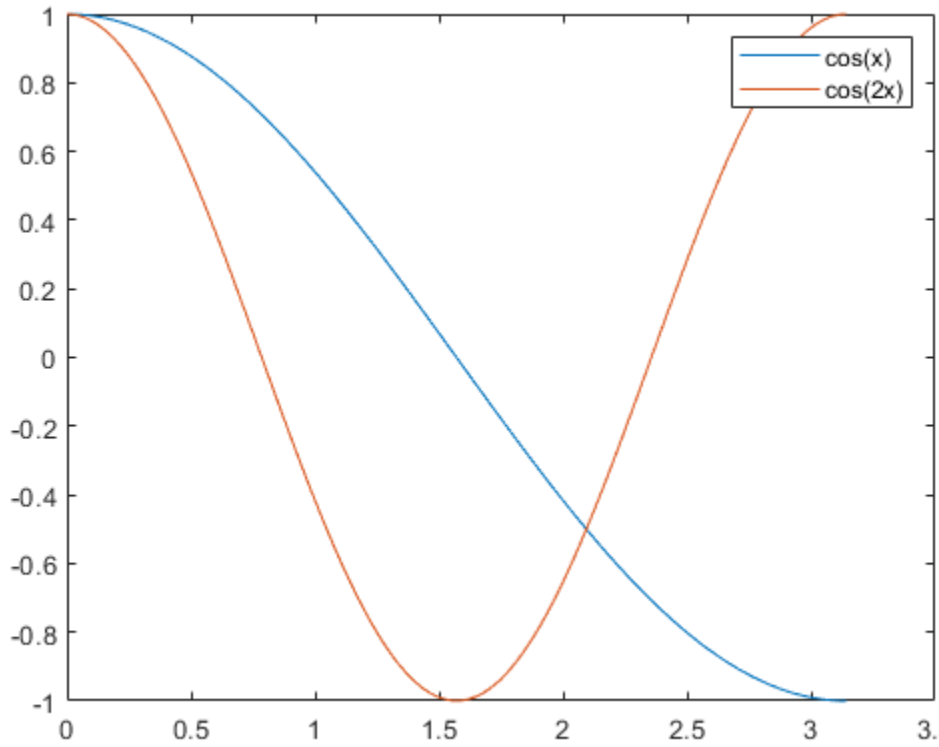
```
plot(y1)

y2 = rand(5);
ax2 = nexttile;
plot(y2)

legend(ax1,{'Line 1','Line 2','Line 3'})
```



Specify Legend Labels During Plotting Commands

Plot two lines. Specify the legend labels during the plotting commands by setting the `DisplayName` property to the desired text. Then, add a legend.

```
x = linspace(0,pi);
y1 = cos(x);
plot(x,y1,'DisplayName','cos(x)')

hold on
y2 = cos(2*x);
plot(x,y2,'DisplayName','cos(2x)')
hold off

legend
```

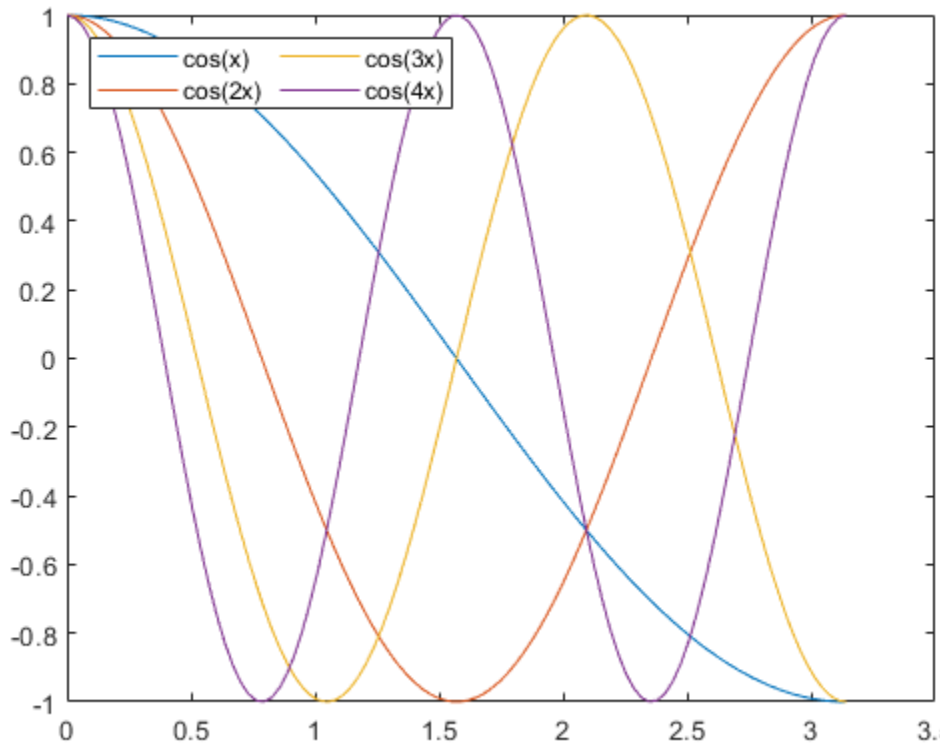


Legend Location and Number of Columns

Plot four lines. Create a legend in the northwest area of the axes. Specify the number of legend columns using the `NumColumns` property.

```
x = linspace(0,pi);
y1 = cos(x);
plot(x,y1)

hold on
y2 = cos(2*x);
```

```
plot(x,y2)

y3 = cos(3*x);
plot(x,y3)

y4 = cos(4*x);
plot(x,y4)
hold off

legend({'cos(x)','cos(2x)','cos(3x)','cos(4x)'},'Location','northwest','NumColumns',2)
```



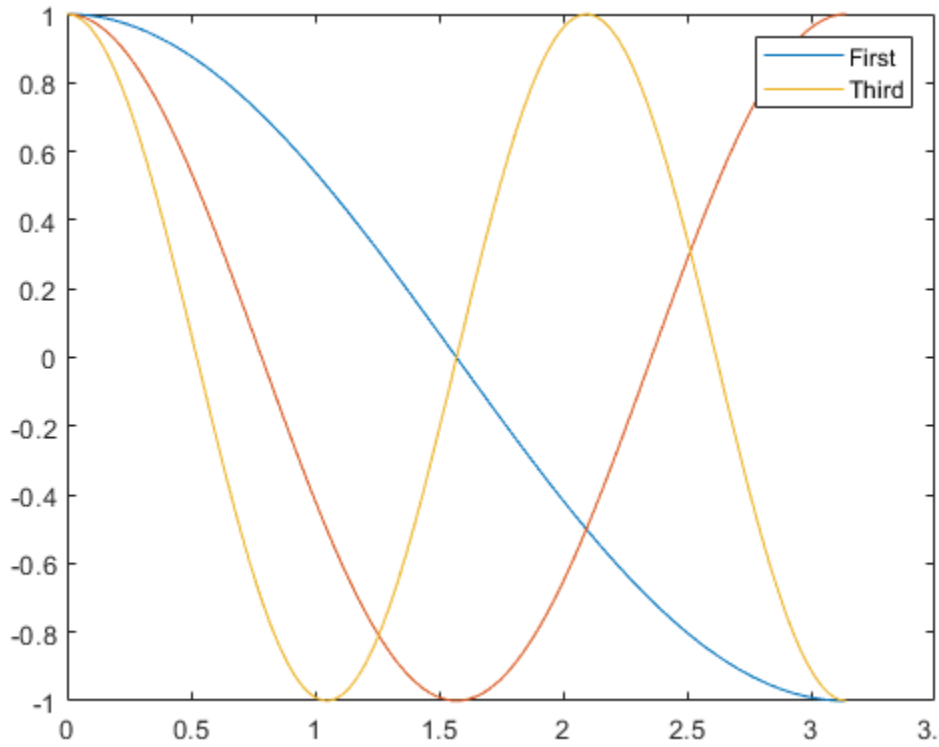
By default, the legend orders the items from top to bottom along each column. To order the items from left to right along each row instead, set the `Orientation` property to `'horizontal'`.

Included Subset of Graphics Objects in Legend

If you do not want to include all of the plotted graphics objects in the legend, then you can specify the graphics objects that you want to include.

Plot three lines and return the `Line` objects created. Create a legend that includes only two of the lines. Specify the first input argument as a vector of the `Line` objects to include.

```
x = linspace(0,pi);
y1 = cos(x);
p1 = plot(x,y1);

hold on
y2 = cos(2*x);
p2 = plot(x,y2);

y3 = cos(3*x);
p3 = plot(x,y3);
hold off

legend([p1 p3],{'First','Third'})
```

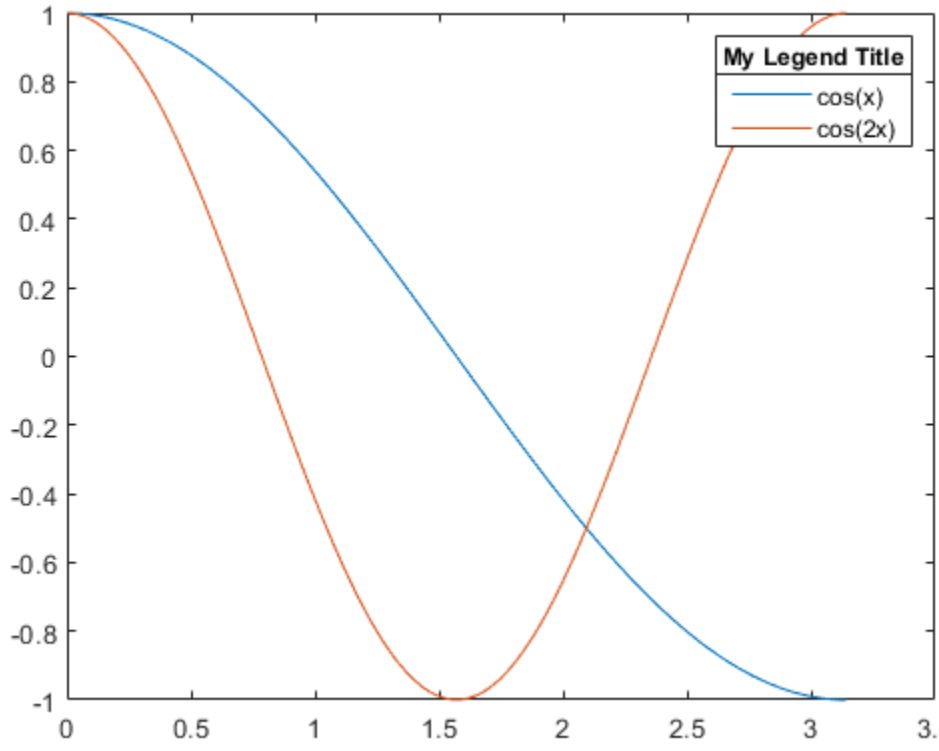


Add Title to Legend

Plot two lines and create a legend. Then, add a title to the legend.

```
x = linspace(0,pi);
y1 = cos(x);
plot(x,y1)

hold on
y2 = cos(2*x);
plot(x,y2)
```

```
hold off

lgd = legend('cos(x)','cos(2x)');
title(lgd,'My Legend Title')
```

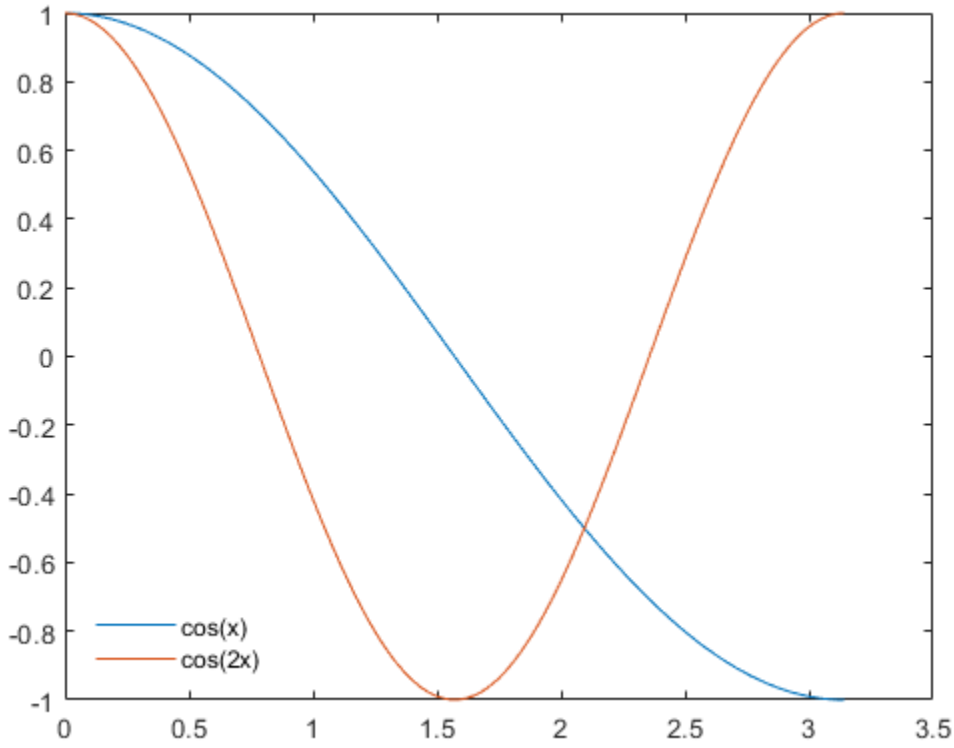


Remove Legend Background

Plot two lines and create a legend in the lower left corner of the axes. Then, remove the legend background and outline.

```
x = linspace(0,pi);
y1 = cos(x);
```

```
plot(x,y1)

hold on
y2 = cos(2*x);
plot(x,y2)
hold off

legend({'cos(x)','cos(2x)'},'Location','southwest')
legend('boxoff')
```

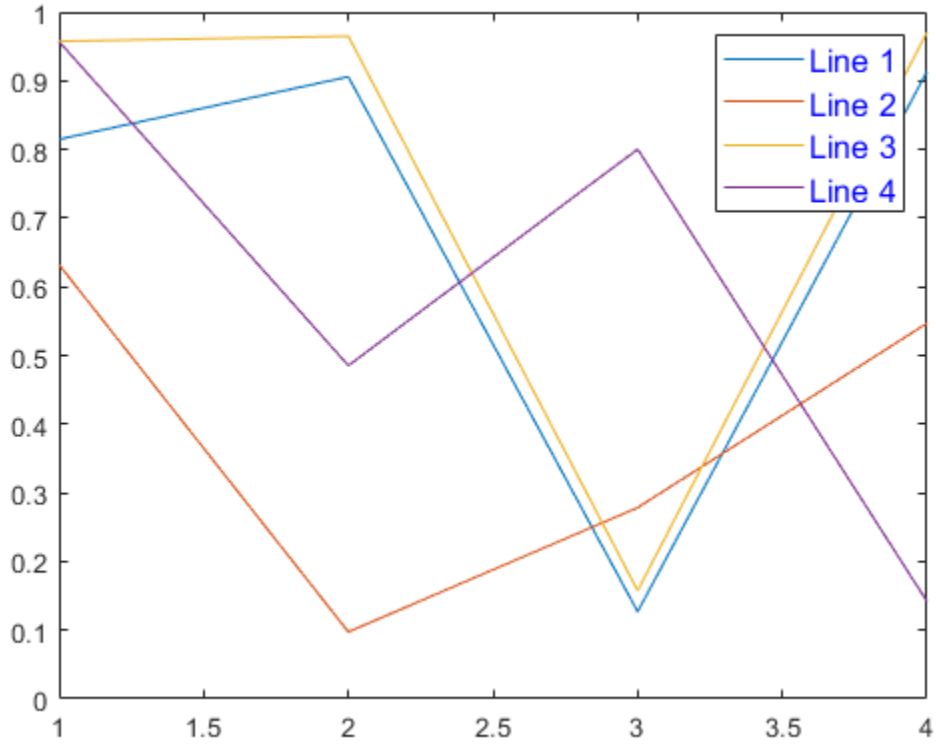


Modify Legend Appearance

Modify the legend appearance by setting `Legend` properties. You can set properties when you create the legend using name-value pairs in the `legend` command. You also can set properties after you create the legend using the `Legend` object.

Plot four lines of random data. Create legend and assign the `Legend` object to the variable `lgd`. Set the `FontSize` and `TextColor` properties using name-value pairs. When you specify name-value pair arguments, you must specify the legend labels using a cell array.

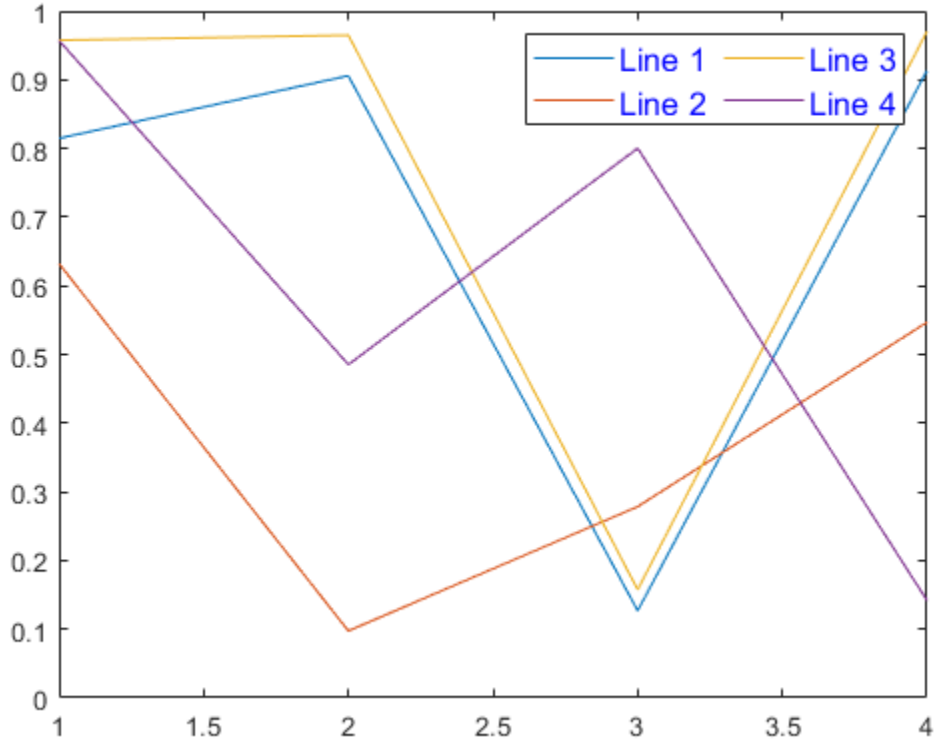
```
rdm = rand(4);
plot(rdm)

lgd = legend({'Line 1','Line 2','Line 3','Line 4'},'FontSize',12,'TextColor','blue')
```




```
lgd =
  Legend (Line 1, Line 2, Line 3, Line 4) with properties:

         String: {'Line 1'  'Line 2'  'Line 3'  'Line 4'}
       Location: 'northeast'
    Orientation: 'vertical'
       FontSize: 12
       Position: [0.7167 0.6877 0.1696 0.2131]
          Units: 'normalized'

  Show all properties
```

Modify the legend after it is created by referring to `lgd`. Set the `NumColumns` property using the object dot property name notation.

```
lgd.NumColumns = 2;
```



Input Arguments

`label1,...,labelN` — Labels (as separate arguments)
character vectors | strings

Labels, specified as separate arguments of character vectors or strings. To include special characters or Greek letters in the labels, use TeX markup. For a table of options, see the `Interpreter` property.

Example: `legend('Sin Function','Cos Function')`

Example: `legend("Sin Function","Cos Function")`

Example: `legend('\gamma','\sigma')`

To specify labels that are keywords, such as `'Location'` or `'off'`, use a cell array of character vectors, a string array, or a character array.

`labels` — Labels (as an array)
cell array of character vectors | string array | categorical array

Labels, specified as a cell array of character vectors, string array, or categorical array. To include special characters or Greek letters in the labels, use TeX markup. For a table of options, see the `Interpreter` property.

Example: `legend({'Sin Function','Cos Function'})`

Example: `legend(["Sin Function","Cos Function"])`

Example: `legend({'\gamma','\sigma'})`

Example: `legend(categorical({'Alabama','New York'}))`

`subset` — Data series to include in legend
vector of graphics objects

Data series to include in the legend, specified as a vector of graphics objects.

`target` — Target for legend
`Axes` object | `PolarAxes` object | `GeographicAxes` object | graphics object

Target for legend, specified as an `Axes` object, a `PolarAxes` object, a `GeographicAxes` object, or a graphics object with a `LegendVisible` property, such as a `GeographicBubbleChart` object. If you do not specify the target, then the `legend` function uses the axes or chart returned by the `gca` command.

Some charts do not support modifying the legend appearance, such as the location, or returning the `Legend` object as an output argument..

`lcn` — Legend location
`'north'` | `'south'` | `'east'` | `'west'` | `'northeast'` | ...

Legend location with respect to the axes, specified as one of the location values listed in this table.

Value	Description
`'north'`	Inside top of axes
`'south'`	Inside bottom of axes
`'east'`	Inside right of axes
`'west'`	Inside left of axes
`'northeast'`	Inside top-right of axes (default for 2-D axes)
`'northwest'`	Inside top-left of axes
`'southeast'`	Inside bottom-right of axes
`'southwest'`	Inside bottom-left of axes
`'northoutside'`	Above the axes
`'southoutside'`	Below the axes
`'eastoutside'`	To the right of the axes
`'westoutside'`	To the left of the axes
`'northeastoutside'`	Outside top-right corner of the axes (default for 3-D axes)
`'northwestoutside'`	Outside top-left corner of the axes
`'southeastoutside'`	Outside bottom-right corner of the axes
`'southwestoutside'`	Outside bottom-left corner of the axes
`'best'`	Inside axes where least conflict occurs with the plot data at the time that you create the legend. If the plot data changes, you might need to reset the location to `'best'`.
`'bestoutside'`	Outside top-right corner of the axes (when the legend has a vertical orientation) or below the axes (when the legend has a horizontal orientation)

Value	Description
`'none'`	Determined by `Position` property. Use the `Position` property to display the legend in a custom location.

Example: `legend('Location','northeastoutside')`

`ornt` — Orientation
`'vertical'` (default) | `'horizontal'`

Orientation, specified as one of these values:

- `'vertical'` — Stack the legend items vertically.
- `'horizontal'` — List the legend items side-by-side.

Example: `legend('Orientation','horizontal')`

bkgd — Legend box display
`'boxon'` (default) | `'boxoff'`

Legend box display, specified as one of these values:

- `'boxon'` — Display the legend background and outline.
- `'boxoff'` — Do not display the legend background and outline.

Example: `legend('boxoff')`

`vsbl` — Legend visibility
`'hide'` | `'show'` | `'toggle'`

Legend visibility, specified as one of these values:

- `'hide'` — Hide the legend.
- `'Show'` — Show the legend or create a legend if one does not exist.
- `'toggle'` — Toggle the legend visibility.

Example: `legend('hide')`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Note The properties listed here are only a subset. For a complete list, see Legend.

Example: `legend({'A','B'},'TextColor','blue','FontSize',12)` creates a legend with blue, 12-point font.

`TextColor` — Text color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Text color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default color is black with a value of `[0 0 0]`.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `[0 0 1]`

Example: `'blue'`

Example: `'#0000FF'`

`FontSize` — Font size
scalar value greater than zero

Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale.

If you change the axes font size, then MATLAB automatically sets the font size of the colorbar to 90% of the axes font size. If you manually set the font size of the colorbar, then changing the axes font size does not affect the colorbar font.

`NumColumns` — Number of columns
1 (default) | positive integer

Number of columns, specified as a positive integer. If there are not enough legend items to fill the specified number of columns, then the number of columns that appear might be fewer.

Use the `Orientation` property to control whether the legend items appear in order along each column or along each row.

Example: `lgd.NumColumns = 3`

Output Arguments

`lgd` — Legend object
`Legend` object

`Legend` object. Use `lgd` to view or modify properties of the legend after it is created.

```
plot(rand(3))
lgd = legend('line1','line2','line3');
lgd.FontSize = 12;
lgd.FontWeight = 'bold';
```

Tips

- To label more than 20 objects in the legend, specify a label for each object. Otherwise, `legend` depicts only the first 20 objects in the graph.

Algorithms

- Recalling the `legend` function does not reset legend properties, such as the location or orientation. If a legend exists, then the `legend` function updates the existing legend. An `Axes` object can have only one legend.
- The legend reflects the visibility of graphics objects in the axes. Graphics objects that have a `Visible` property set to `'off'` appear as grayed out items in the legend.

Compatibility Considerations

`legend` interprets arguments as property names when property exists

Behavior changed in R2018b

Starting in R2018b, if you pass an argument to the `legend` function that matches the name of a legend property, the function interprets the argument as the name of a name-value pair. In previous releases, the `legend` function recognized name-value pairs only when the first argument was a cell array.

As a result of this change, in most cases, it is unnecessary to specify the first argument as a cell array when using name-value pairs. However, if you want a label in your legend that matches the name of a legend property, such as `Position` or `NumColumns`, then you *must* specify all the labels in a cell array. Otherwise, the `legend` function interprets the argument as a name-value pair instead of a label.

Description	Recommended Code
If you want a label in your legend that matches the name of a legend property, such as `'NumColumns'`, then specify all the labels in a cell array. If you specify `'NumColumns'` outside of a cell array, the `legend` function interprets it as a name-value pair.	`legend({'Label1','NumColumns','Label3','Label4'}`
If none of your labels match the name of a legend property, then you do not need to use a cell array around the labels.	`legend('Label1','Label2','Label2')`

`legend` creates axes if they do not exist

Behavior changed in R2017b

Starting in R2017b, if axes do not exist, then the `legend` function creates them.

Legends automatically update when you add or remove data

Behavior changed in R2017a

Starting in R2017a, the legend automatically updates when you add or remove data series from the axes. If you do not want the legend to automatically update, set the `AutoUpdate` property of the legend to `'off'`.

```
lgd = legend;
lgd.AutoUpdate = 'off';
```

Returning multiple outputs is not recommended

Not recommended starting in R2014b

Returning multiple outputs using this syntax is no longer recommended:

```
[lgd,icons,plots,txt] = legend(__)
```

This syntax creates a legend that does not support some functionality introduced in R2014b or later. For example, you cannot add a title to the legend or specify the number of legend columns. Also, the legend does not automatically update when you add or remove data series from the axes.

Instead, return the `Legend` object and set Legend.

```
lgd = legend(__)
```

See Also

Functions

`hold` | `plot` | `text` | `title` | `xlabel` | `ylabel`

Properties

Legend

Topics

"Add Legend to Graph"

Introduced before R2006a

Legend Properties

Legend appearance and behavior

Description

`Legend` properties control the appearance and behavior of a `Legend` object. By changing property values, you can modify certain aspects of the legend. Use dot notation to refer to a particular object and property:

```
plot(rand(3))
lgd = legend('a','b','c');
c = lgd.TextColor;
lgd.TextColor = 'red';
```

Properties

Position and Layout

`Location` — Location with respect to axes

`'north'` | `'south'` | `'east'` | `'west'` | `'northeast'` | ...

Location with respect to the axes, specified as one of the location values listed in this table.

Value	Description
`'north'`	Inside top of axes
`'south'`	Inside bottom of axes
`'east'`	Inside right of axes
`'west'`	Inside left of axes
`'northeast'`	Inside top-right of axes (default for 2-D axes)
`'northwest'`	Inside top-left of axes
`'southeast'`	Inside bottom-right of axes

Value	Description
`'southwest'`	Inside bottom-left of axes
`'northoutside'`	Above the axes
`'southoutside'`	Below the axes
`'eastoutside'`	To the right of the axes
`'westoutside'`	To the left of the axes
`'northeastoutside'`	Outside top-right corner of the axes (default for 3-D axes)
`'northwestoutside'`	Outside top-left corner of the axes
`'southeastoutside'`	Outside bottom-right corner of the axes
`'southwestoutside'`	Outside bottom-left corner of the axes
`'best'`	Inside axes where least conflict with data in plot
`'bestoutside'`	Outside top-right corner of the axes (when the legend has a vertical orientation) or below the axes (when the legend has a horizontal orientation)
`'none'`	Determined by `Position` property. Use the `Position` property to specify a custom location.

Example: `legend('Location','northeastoutside')`

`Orientation` — Orientation
`'vertical'` (default) | `'horizontal'`

Orientation, specified as one of these values:

- `'vertical'` — Stack the legend items vertically. If the legend has multiple columns, layout the items from top to bottom along each column.
- `'horizontal'` — List the legend items side-by-side. If the legend has multiple columns, layout the items from left to right along each row.

Example: `legend('Orientation','horizontal')`

`NumColumns` — Number of columns
1 (default) | positive integer

Number of columns, specified as a positive integer. If there are not enough legend items to fill the specified number of columns, then the number of columns that appear might be fewer.

Use the `Orientation` property to control whether the legend items appear in order along each column or along each row.

Example: `lgd.NumColumns = 3`

`NumColumnsMode` — Selection mode for `NumColumns`
`'auto'` (default) | `'manual'`

Selection mode for the `NumColumns` value, specified as one of these values:

- `'auto'` — Automatically select the value.
- `'manual'` — Use the manually specified value. To specify the value, set the `NumColumns` property.

`Position` — Custom location and size
four-element vector

Custom location and size, specified as a four-element vector of the form `[left bottom width height]`. The first two values, `left` and `bottom`, specify the distance from the lower left corner of the figure to the lower left corner of the legend. The last two values, `width` and `height`, specify the legend dimensions. The `Units` property determines the position units.

If you specify the `Position` property, then MATLAB automatically changes the `Location` property to `'none'`.

Example: `legend({'A','B'},'Position',[0.2 0.6 0.1 0.2])`

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`Units` — Position units
`'normalized'` (default) | `'inches'` | `'centimeters'` | `'characters'` | `'points'` | `'pixels'`

Position units, specified as one of the values in this table.

Units	Description
`'normalized'` (default)	Normalized with respect to the container, which is usually the figure. The lower-left corner of the figure maps to `(0,0)` and the upper-right corner maps to `(1,1)`. Resizing the figure updates the values of the `Position` vector.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the default system font character size. • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.
`'points'`	Points. One point equals 1/72 inch.
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems. • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. • On Linux systems, the size of a pixel is determined by your system resolution.

All units are measured from the lower-left corner of the container window.

This property affects the `Position` property. If you change the units, then it is good practice to return it to its default value after completing your computation to prevent affecting other functions that assume `Units` is the default value.

If you specify the `Position` and `Units` properties as `Name,Value` pairs when creating the object, then the order of specification matters. If you want to define the position with particular units, then you must set the `Units` property before the `Position` property.

Labels

`AutoUpdate` — Automatic update of legend items
`'on'` (default) | `'off'`

Automatic update of legend items to reflect the current state of the axes, specified as one of these values:

- `'on'` — Automatically add legend items for new graphics objects added to the axes. Remove legend items for graphics objects deleted from the axes.
- `'off'` — Do not automatically add or delete legend items.

Example: `legend({'A','B'},'AutoUpdate','off')`

`String` — Text for legend labels
cell array of character vectors | string array | categorical array

Text for legend labels, specified as a cell array of character vectors, string array, or categorical array. To include special characters or Greek letters in the labels, use TeX markup. For a table of options, see the `Interpreter` property.

`Title` — Legend title
legend text object

Legend title, returned as a legend text object. To add a legend title, set the `String` property of the legend text object. To change the title appearance, such as the font style or color, set legend text properties. For a list, see Legend Text.

```
plot(rand(3));
lgd = legend('line 1','line 2','line 3');
lgd.Title.String = 'My Legend Title';
lgd.Title.FontSize = 12;
```

Alternatively, use the `title` function to add a title and control the appearance.

```
plot(rand(3));
lgd = legend('line 1','line 2','line 3');
title(lgd,'My Legend Title','FontSize',12)
```

`Interpreter` — Interpretation of text characters
`'tex'` (default) | `'latex'` | `'none'`

Interpretation of text characters, specified as one of these values:

- `'tex'` — Interpret characters using a subset of TeX markup.
- `'latex'` — Interpret characters using LaTeX markup.
- `'none'` — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text.

Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they modify only the next character or the characters within the curly braces. When you set the `Interpreter` property to `'tex'`, the supported modifiers are as follows.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{`*`specifier`*`}`	Font name — Replace *`specifier`* with the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{`*`specifier`*`}`	Font size —Replace *`specifier`* with a numeric scalar value in point units.	`'\fontsize{15} text'`

Modifier	Description	Example
`\color{specifier}`	Font color — Replace *specifier* with one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Replace *specifier* with a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	β	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ϵ	`\Delta`	Δ	`\spadesuit`	♠
`\zeta`	ζ	`\Theta`	Θ	`\leftrightarrow`	↔
`\eta`	η	`\Lambda`	Λ	`\leftarrow`	←
`\theta`	θ	`\Xi`	Ξ	`\Leftarrow`	⇐
`\vartheta`	ϑ	`\Pi`	Π	`\uparrow`	↑
`\iota`	ι	`\Sigma`	Σ	`\rightarrow`	→

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	μ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. Use dollar symbols around the text, for example, use `'$\int_1^{20} x^2 dx$'` for inline mode or `'$$ \int_1^{20} x^2 dx$$'` for display mode.

The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup.

The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters. For multiline text, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project website at https://www.latex-project.org/.

Font

`FontName` — Font name

supported font name | `'FixedWidth'`

Font name, specified as a supported font name or `'FixedWidth'`. To display and print text properly, you must choose a font that your system supports. The default font depends on your operating system and locale.

To use a fixed-width font that looks good in any locale, use `'FixedWidth'`. The fixed-width font relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`FontSize` — Font size

scalar value greater than zero

Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale.

If you change the axes font size, then MATLAB automatically sets the font size of the colorbar to 90% of the axes font size. If you manually set the font size of the colorbar, then changing the axes font size does not affect the colorbar font.

`FontWeight` — Character thickness

`'normal'` (default) | `'bold'`

Character thickness, specified as `'normal'` or `'bold'`.

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.

FontAngle — Character slant

`'normal'` (default) | `'italic'`

Character slant, specified as `'normal'` or `'italic'`.

Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.

Color and Styling

TextColor — Text color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Text color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default color is black with a value of `[0 0 0]`.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `[0 0 1]`

Example: `'blue'`

Example: `'#0000FF'`

`Color` — Background color

`[1 1 1]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[1 1 1]` corresponds to white.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `legend({'A','B'},'Color','y')`

Example: `legend({'A','B'},'Color',[0.8 0.8 1])`

Example: `legend({'A','B'},'Color','#D9A2E9')`

`EdgeColor` — Box outline color

`[0.15 0.15 0.15]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Box outline color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[0.15 0.15 0.15]` corresponds to dark gray.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `legend({'A','B'},'EdgeColor',[0 1 0])`

Box — Display of box outline
`'on'` (default) | `'off'`

Display of box outline, specified as one of these values:

- `'on'` — Display the box around the legend.
- `'off'` — Do not display the box around the legend.

Example: `legend({'A','B'},'Box','off')`

`LineWidth` — Width of box outline
`0.5` (default) | positive value

Width of box outline, specified as a positive value in point units. One point equals 1/72 inch.

Example: `1.5`

Interactivity

`Visible` — State of visibility
`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`UIContextMenu` — Context menu
uicontextmenu object (default)

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

Callbacks

`ItemHitFcn` — Callback that executes when you click legend items

`@defaultItemHitCallback` (default) | function handle | cell array | character vector

Callback that executes when you click legend items, specified as one of these values:

- Function handle. For example, `@myCallback`.
- Cell array containing a function handle and additional arguments. For example, `{@myCallback,arg3}`.
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended).

If you specify this property using a function handle, then MATLAB passes the `Legend` object and an event data structure as the first and second input arguments to the function. This table describes the fields in the event data structure.

Event Data Structure Fields

Field	Description
`Peer`	Chart object associated with the clicked legend item.
`Region`	Region of legend item clicked, returned as either `'icon'` or `'label'`.
`SelectionType`	Type of click, returned as one of these values: • `'normal'` — Single-click left mouse button • `'extend'` — **Shift** + single-click left mouse button • `'open'` — Double-click any mouse button • `'alt'` — Single-click right mouse button, both mouse buttons (Windows and Mac), or middle mouse button (Mac and Linux). If the `UIContextMenu` property contains a valid context menu (which is the default), then this type of click opens the context menu instead of triggering the `ItemHitFcn` callback.
`Source`	`Legend` object.
`EventName`	Event name, `'ItemHit'`.

Note If you set the `ButtonDownFcn` property, then the `ItemHitFcn` property is disabled.

Example

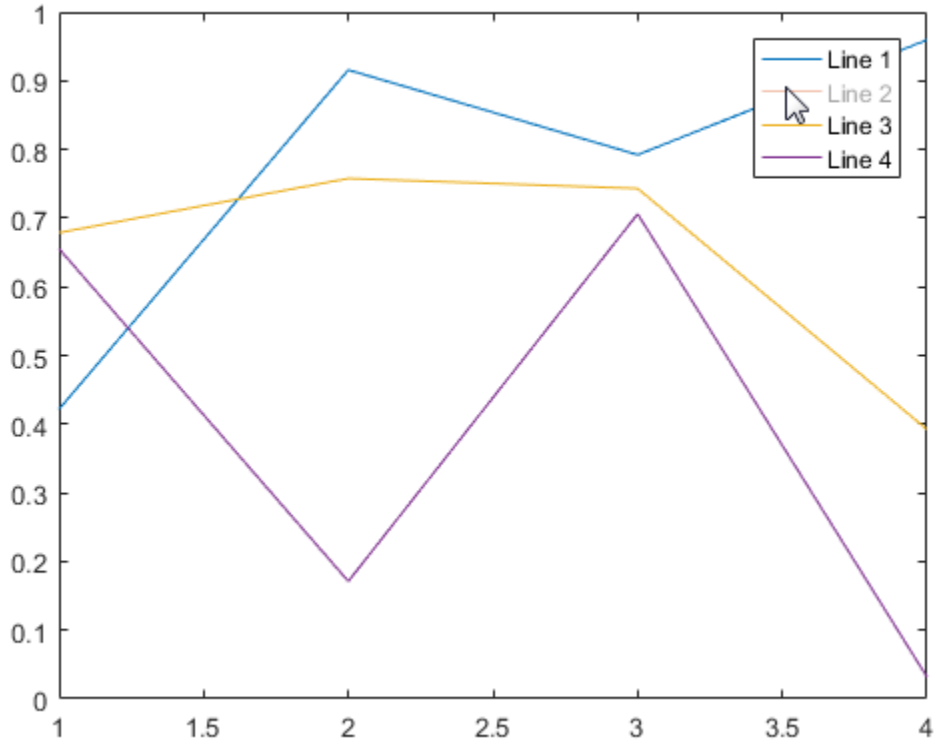
You can create interactive legends so that when you click an item in the legend, the associated chart updates in some way. For example, you can toggle the visibility of the chart or change its line width. Set the `ItemHitFcn` property of the legend to a callback function that controls how the charts change. This example shows how to toggle the visibility of a chart when you click the chart icon or label in a legend. It creates a callback function that changes the `Visible` property of the chart to either `'on'` or `'off'`.

Copy the following code to a new function file and save it as `hitcallback_ex1.m` either in the current folder or in a folder on the MATLAB search path. The two input arguments, `src` and `evnt`, are the legend object and an event data structure. MATLAB automatically passes these inputs to the callback function when you click an item in the legend. Use the `Peer` field of the event data structure to access properties of the chart object associated with the clicked legend item.

```
function hitcallback_ex1(src,evnt)

if strcmp(evnt.Peer.Visible,'on')
    evnt.Peer.Visible = 'off';
else
    evnt.Peer.Visible = 'on';
end

end
```

Then, plot four lines, create a legend, and assign the legend object to a variable. Set the `ItemHitFcn` property of the legend object to the callback function. Click items in the legend to show or hide the associated chart. The legend label changes to gray when you hide a chart.

```
plot(rand(4));
l = legend('Line 1','Line 2','Line 3','Line 4');
l.ItemHitFcn = @hitcallback_ex1;
```



`ButtonDownFcn` — Mouse-click callback

`@bdowncb` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'off'` (default) | `'on'`

Callback interruption, specified as `'off'` or `'on'`. The `Interruptible` property determines if a running callback can be interrupted.

Consider these callback states where:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. The `Interruptible` property has two possible values:

- `'off'` — The running callback cannot be interrupted. MATLAB finishes executing the running callback without any interruptions. The `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue.

- `'on'` — The running callback can be interrupted. Interruption occurs at the next point where MATLAB processes the queue. For example, when you have a command such as `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks when visible. The `Visible` property must be set to `'on'` and you must click a part of the `Legend` object that has a defined color. You cannot click a part that has an associated color property set to `'none'`. The `HitTest` property determines if the `Legend` object responds to the click or if an ancestor does.

- `'none'` — Cannot capture mouse clicks. Clicking the `Legend` object passes the click to the object below it in the current view of the figure window. The `HitTest` property of the `Legend` object has no effect.

`HitTest` — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `Legend` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `Legend` object that has one of these:
  - `HitTest` property set to `'on'`
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

---

Note The `PickableParts` property determines if the `Legend` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container

`Figure` object | `Panel` object | `Tab` object | `TiledChartLayout` object

Parent container, specified as a `Figure` object, `Panel` object, `Tab` object, or a `TiledChartLayout` object.

The `Legend` object must have the same parent as the associated axes. If you change the parent of the associated axes, then the `Legend` object automatically updates to use the same parent.

`Children` — Children
empty `GraphicsPlaceholder` array

The object has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object
`'legend'` (default)

This property is read-only.

Type of graphics object, returned as `'legend'`. Use this property to find all objects of a given type within a plotting hierarchy.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also
`legend`

Topics
“Access Property Values”
“Graphics Object Properties”

Introduced in R2014b

Text Properties

Legend text appearance and behavior

Description

Legend text properties control the appearance and behavior of the legend title. By changing property values, you can modify certain aspects of the title. Use dot notation to refer to a particular object and property:

```
l = legend('show');
l.Title.String = 'My Title';
l.Title.Color = 'red';
c = l.Title.Color
```

Properties

Text

`String` — Text to display as title
character array | string array | cell array | categorical array | numeric value

Text to display as title, specified as a character array, string array, cell array, categorical array, or numeric value.

Example: `'my title'`

Example: `string('my title')`

Example: `{'first line','second line'}`

Example: `123`

To include numeric variables with text in a title, use the `num2str` function. For example:

```
x = 42;
str = ['The value is ',num2str(x)];
```

To include special characters, such as superscripts, subscripts, Greek letters, or mathematical symbols, use TeX markup. For a list of supported markup, see the `Interpreter` property.

To create multiline titles:

- Use a string array, where each element contains a line of text, such as `string({'line one','line two'})`.
- Use a cell array where each cell contains a line of text, such as `{'first line','second line'}`.
- Use a character array where each row contains a line of text, such as `['abc'; 'ab']`. If you use this technique, each row must contain the same number of characters.
- Use `sprintf` to create text with a new line character, such as `sprintf('first line \n second line')`.

Numeric titles are converted to text using `sprintf('%g',value)`. For example, `12345678` displays as `1.23457e+07`.

Note

- The words `default`, `factory`, and `remove` are reserved words that will not appear in a title when quoted as a normal character vector. To display any of these words individually, precede them with a backslash, such as `'\default'` or `'\remove'`.
- If you specify this property as a categorical array, MATLAB uses the values in the array, not the categories.

`Interpreter` — Interpretation of text characters

`'tex'` (default) | `'latex'` | `'none'`

Interpretation of text characters, specified as one of these values:

- `'tex'` — Interpret characters using a subset of TeX markup.
- `'latex'` — Interpret characters using LaTeX markup.
- `'none'` — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text.

Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they modify only the next character or the characters within the curly braces. When you set the `Interpreter` property to `'tex'`, the supported modifiers are as follows.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{`*`specifier`*`}`	Font name — Replace *`specifier`* with the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{`*`specifier`*`}`	Font size —Replace *`specifier`* with a numeric scalar value in point units.	`'\fontsize{15} text'`
`\color{`*`specifier`*`}`	Font color — Replace *`specifier`* with one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`

Modifier	Description	Example
`\color[rgb] {specifier}`	Custom font color — Replace *specifier* with a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	β	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ϵ	`\Delta`	Δ	`\spadesuit`	♠
`\zeta`	ζ	`\Theta`	Θ	`\leftrightarrow`	↔
`\eta`	η	`\Lambda`	Λ	`\leftarrow`	←
`\theta`	θ	`\Xi`	Ξ	`\Leftarrow`	⇐
`\vartheta`	ϑ	`\Pi`	Π	`\uparrow`	↑
`\iota`	ι	`\Sigma`	Σ	`\rightarrow`	→
`\kappa`	κ	`\Upsilon`	ϒ	`\Rightarrow`	⇒
`\lambda`	λ	`\Phi`	Φ	`\downarrow`	↓
`\mu`	µ	`\Psi`	Ψ	`\circ`	º
`\nu`	ν	`\Omega`	Ω	`\pm`	±
`\xi`	ξ	`\forall`	∀	`\geq`	≥
`\pi`	π	`\exists`	∃	`\propto`	∝

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. Use dollar symbols around the text, for example, use `'$\int_1^{20} x^2 dx$'` for inline mode or `'$$ \int_1^{20} x^2 dx$$'` for display mode.

The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup.

The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters. For multiline text, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project website at https://www.latex-project.org/.

`InterpreterMode` — Selection mode for `Interpreter`

`'auto'` (default) | `'manual'`

Selection mode for the `Interpreter` property, specified as one of these values:

- `'auto'` — Use the same value as the `Interpreter` property for the associated `Legend` object.
- `'manual'` — Use a value of `Interpreter` that you specify. To specify the value, set the `Interpreter` property. When you set the `Interpreter` property, the `InterpreterMode` property changes to `'manual'`.

Font Style and Appearance

`Color` — Text color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Text color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: 'blue'

Example: [0 0 1]

Example: '#0000FF'

ColorMode — Selection mode for Color
'auto' (default) | 'manual'

Selection mode for the Color property, specified as one of these values:

- 'auto' — Use the same value as the TextColor property for the associated Legend object.
- 'manual' — Use a value of Color that you specify. To specify the value, set the Color property. When you set the Color property, the ColorMode property changes to 'manual'.

FontAngle — Character slant
'normal' (default) | 'italic'

Character slant, specified as 'normal' or 'italic'.

Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.

FontAngleMode — Selection mode for FontAngle
'auto' (default) | 'manual'

Selection mode for the FontAngle property, specified as one of these values:

- 'auto' — Use the same value as the FontAngle property for the associated Legend object.
- 'manual' — Use a value of FontAngle that you specify. To specify the value, set the FontAngle property. When you set the FontAngle property, the FontAngleMode property changes to 'manual'.

FontName — Font name
supported font name | 'FixedWidth'

Font name, specified as a supported font name or 'FixedWidth'. To display and print text properly, you must choose a font that your system supports. The default font depends on your operating system and locale.

To use a fixed-width font that looks good in any locale, use 'FixedWidth'. The fixed-width font relies on the root FixedWidthFontName property. Setting the root FixedWidthFontName property causes an immediate update of the display to use the new font.

FontNameMode — Selection mode for FontName
'auto' (default) | 'manual'

Selection mode for the FontName property, specified as one of these values:

- 'auto' — Use the same value as the FontName property for the associated Legend object.
- 'manual' — Use a value of FontName that you specify. To specify the value, set the FontName property. When you set the FontName property, the FontNameMode property changes to 'manual'.

FontSize — Font size
scalar value greater than zero

Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale.

If you change the axes font size, then MATLAB automatically sets the font size of the colorbar to 90% of the axes font size. If you manually set the font size of the colorbar, then changing the axes font size does not affect the colorbar font.

FontSizeMode — Selection mode for FontSize
`'auto'` (default) | `'manual'`

Selection mode for the `FontSize` property, specified as one of these values:

- `'auto'` — Use the same value as the `FontSize` property for the associated `Legend` object.
- `'manual'` — Use a value of `FontSize` that you specify. To specify the value, set the `FontSize` property. When you set the `FontSize` property, the `FontSizeMode` property changes to `'manual'`.

FontWeight — Character thickness
`'normal'` (default) | `'bold'`

Character thickness, specified as `'normal'` or `'bold'`.

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.

FontWeightMode — Selection mode for FontWeight
`'auto'` (default) | `'manual'`

Selection mode for the `FontWeight` property, specified as one of these values:

- `'auto'` — Use the same value as the `FontWeight` property for the associated `Legend` object.
- `'manual'` — Use a value of `FontWeight` that you specify. To specify the value, set the `FontWeight` property. When you set the `FontWeight` property, the `FontWeightMode` property changes to `'manual'`.

`Visible` — State of visibility
`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

See Also
`legend` | `title`

Introduced in R2016a

legendre

Associated Legendre functions

Syntax

```
P = legendre(n,X)
P = legendre(n,X,normalization)
```

Description

`P = legendre(n,X)` computes the associated Legendre functions on page 1-8570 of degree `n` and order `m = 0, 1, ..., n` evaluated for each element in `X`.

`P = legendre(n,X,normalization)` computes normalized versions of the associated Legendre functions. `normalization` can be `'unnorm'` (default), `'sch'`, or `'norm'`.

Examples

Associated Legendre Function Values of Vector

Use the `legendre` function to operate on a vector and then examine the format of the output.

Calculate the second-degree Legendre function values of a vector.

```
deg = 2;
x = 0:0.1:0.2;
P = legendre(deg,x)
```

```
P = 3×3

   -0.5000   -0.4850   -0.4400
         0   -0.2985   -0.5879
    3.0000    2.9700    2.8800
```

The format of the output is such that:

- Each row contains the function value for different values of m (the order of the associated Legendre function)
- Each column contains the function value for a different value of x

$$\begin{array}{lccc} & x = 0 & x = 0.1 & x = 0.2 \\ m = 0 & P_2^0(0) & P_2^0(0.1) & P_2^0(0.2) \\ m = 1 & P_2^1(0) & P_2^1(0.1) & P_2^1(0.2) \\ m = 2 & P_2^2(0) & P_2^2(0.1) & P_2^2(0.2) \end{array}$$

The equation for the second-degree associated Legendre function P_2^m is

$$P_2^m(x) = (-1)^m\left(1 - x^2\right)^{m/2}\frac{d^m}{dx^m}\left[\frac{1}{2}\left(3x^2 - 1\right)\right].$$

Therefore, the value of $P_2^0(0)$ is

$$P_2^0(0) = \left[\frac{1}{2}\left(3x^2 - 1\right)\right]\Big|_{x = 0} = -\frac{1}{2}.$$

This result agrees with `P(1,1) = -0.5000`.

Compare Legendre Normalizations

Calculate the associated Legendre function values with several normalizations.

Calculate the first-degree, unnormalized Legendre function values P_1^m. The first row of values corresponds to $m = 0$, and the second row to $m = 1$.

```
x = 0:0.2:1;
n = 1;
P_unnorm = legendre(n,x)
```

```
P_unnorm = 2×6
```

```
         0    0.2000    0.4000    0.6000    0.8000    1.0000
   -1.0000   -0.9798   -0.9165   -0.8000   -0.6000         0
```

Next, compute the Schmidt seminormalized function values. Compared to the unnormalized values, the Schmidt form differs when $m > 0$ by the scaling

$$(-1)^m\sqrt{\frac{2(n-m)!}{(n+m)!}}.$$

For the first row, the two normalizations are the same, since $m = 0$. For the second row, the scaling constant multiplying each value is -1.

```
P_sch = legendre(n,x,'sch')
```

```
P_sch = 2×6

         0    0.2000    0.4000    0.6000    0.8000    1.0000
    1.0000    0.9798    0.9165    0.8000    0.6000         0
```

```
C1 = (-1) * sqrt(2*factorial(0)/factorial(2))
```

```
C1 = -1
```

Lastly, compute the fully normalized function values. Compared to the unnormalized values, the fully normalized form differs by the scaling factor

$$(-1)^m\sqrt{\frac{\left(n+\frac{1}{2}\right)(n-m)!}{(n+m)!}}.$$

This scaling factor applies for all values of m, so the first and second rows have different scaling factors.

```
P_norm = legendre(n,x,'norm')
```

```
P_norm = 2×6

         0    0.2449    0.4899    0.7348    0.9798    1.2247
    0.8660    0.8485    0.7937    0.6928    0.5196         0
```

```
Cm0 = sqrt((3/2))
```

```
Cm0 = 1.2247
```

```
Cm1 = (-1) * sqrt((3/2)/2)
```

```
Cm1 = -0.8660
```

Calculate Spherical Harmonics

Spherical harmonics arise in the solution to Laplace's equation and are used to represent functions defined on the surface of a sphere. Use `legendre` to compute and visualize the spherical harmonic for Y_3^2.

The equation for spherical harmonics includes a term for the Legendre function, as well as a complex exponential:

$$Y_l^m(\theta,\phi) = \sqrt{\frac{(2l+1)(l-m)!}{4\pi(l+m)!}}P_l^m(\cos\theta)e^{im\phi}, \qquad -l \le m \le l\,.$$

First, create a grid of values to represent all combinations of $-\frac{\pi}{2} \le \theta \le \frac{\pi}{2}$ (altitude angle) and $0 \le \phi \le 2\pi$ (azimuthal angle).

```
dx = pi/60;
alt = -pi/2:dx:pi/2;
az = 0:dx:2*pi;
[phi,theta] = meshgrid(az,alt);
```

Calculate $P_l^m(\cos\theta)$ on the grid for $l = 3$.

```
l = 3;
Plm = legendre(l,cos(theta));
```

Since `legendre` computes the answer for all values of m, `Plm` contains some extra function values. Extract the values for $m = 2$ and discard the rest. Use the `reshape` function to orient the results as a matrix with the same size as `phi` and `theta`.

```
m = 2;
P32 = reshape(Plm(m+1,:,:), size(phi));
```

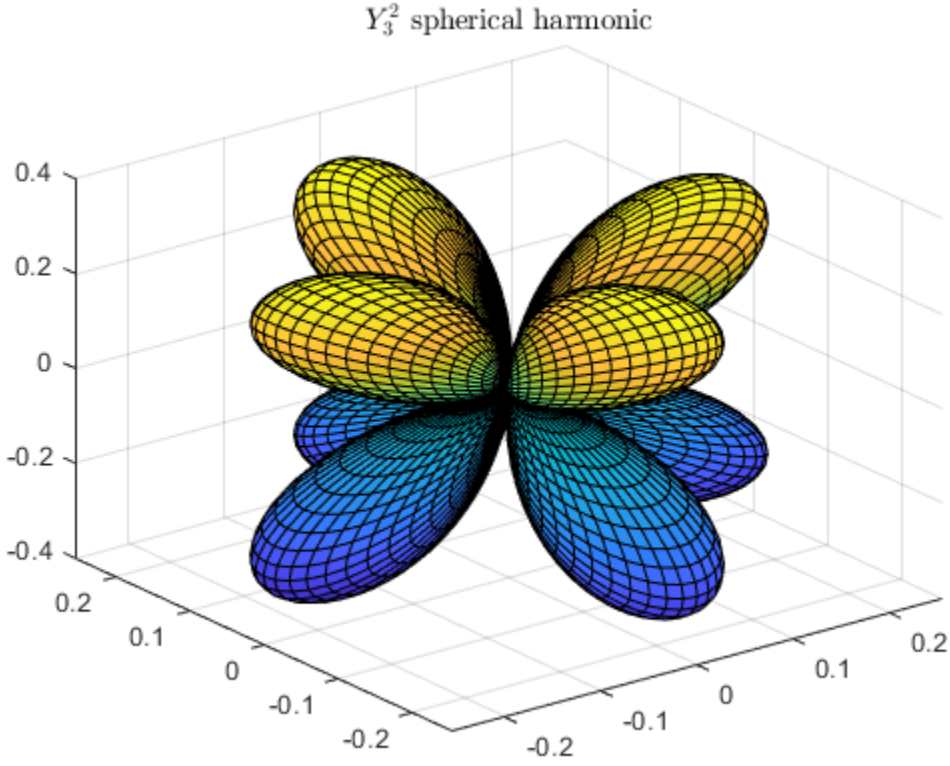
Calculate the spherical harmonic values for Y_3^2.

```
a = (2*l+1)*factorial(l-m);
b = 4*pi*factorial(l+m);
```

```
C = sqrt(a/b);
Y32 = C .* P32 .* exp(1i*m*phi);
```

Convert to Cartesian coordinates and plot the spherical harmonic for Y_3^2 using only the real values.

```
[Xm,Ym,Zm] = sph2cart(phi, theta, real(Y32));
surf(Xm,Ym,Zm)
title('$Y_3^2$ spherical harmonic','interpreter','latex')
```



Input Arguments

n — Degree of Legendre function
positive integer

Degree of Legendre function, specified as a positive integer. For a specified degree, `legendre` computes $P_n^m(x)$ for all orders m from $m = 0$ to $m = n$.

Example: `legendre(2,X)`

X — Input values
scalar | vector | matrix | multidimensional array

Input values, specified as a scalar, vector, matrix, or multidimensional array of real values in the range `[-1,1]`. For example, with spherical harmonics it is common to use `X = cos(theta)` as the input values to compute $P_n^m(\cos\theta)$.

Example: `legendre(2,cos(theta))`

Data Types: `single` | `double`

normalization — Normalization type
`'unnorm'` (default) | `'sch'` | `'norm'`

Normalization type, specified as one of these values.

Value	Result
`'unnorm'`	“Associated Legendre Functions” on page 1-8570
`'sch'`	“Schmidt Seminormalized Associated Legendre Functions” on page 1-8571
`'norm'`	“Fully Normalized Associated Legendre Functions” on page 1-8571

Example: `legendre(n,X,'sch')`

Output Arguments

P — Associated Legendre function values
scalar | vector | matrix | multidimensional array

Associated Legendre function values, returned as a scalar, vector, matrix, or multidimensional array. The normalization of `P` depends on the value of `normalization`.

The size of `P` depends on the size of `X`:

- If `X` is a vector, then `P` is a matrix of size `(n+1)`-by-`length(X)`. The `P(m+1,i)` entry is the associated Legendre function of degree `n` and order `m` evaluated at `X(i)`.
- In general, `P` has one more dimension than `X` and each element `P(m+1,i,j,k,...)` contains the associated Legendre function of degree `n` and order `m` evaluated at `X(i,j,k,...)`.

Limitations

The values of the unnormalized associated Legendre function overflow the range of double-precision numbers for `n > 150` and the range of single-precision numbers for `n > 28`. This overflow results in `Inf` and `NaN` values. For orders larger than these thresholds, consider using the `'sch'` or `'norm'` normalizations instead.

More About

Associated Legendre Functions

The associated Legendre functions $y = P_n^m(x)$ are solutions to the general Legendre differential equation

$$\left(1 - x^2\right)\frac{d^2y}{dx^2} - 2x\frac{dy}{dx} + \left[n(n+1) - \frac{m^2}{1 - x^2}\right]y = 0 \ .$$

n is the integer degree and m is the integer order of the associated Legendre function, such that $0 \le m \le n$.

The associated Legendre functions $P_n^m(x)$ are the most general solutions to this equation given by

$$P_n^m(x) = (-1)^m \left(1 - x^2\right)^{m/2} \frac{d^m}{dx^m} P_n(x) \ .$$

They are defined in terms of derivatives of the Legendre polynomials $P_n(x)$, which are a subset of the solutions given by

$$P_n(x) = \frac{1}{2^n n!} \ \frac{d^n}{dx^n} \left(x^2 - 1\right)^n \ .$$

The first few Legendre polynomials are

Value of n	$P_n(x)$
0	$P_0(x) = 1$
1	$P_1(x) = x$
2	$P_2(x) = \frac{1}{2}\left(3x^2 - 1\right)$

Schmidt Seminormalized Associated Legendre Functions

The Schmidt seminormalized associated Legendre functions are related to the unnormalized associated Legendre functions $P_n^m(x)$ by

$$P_n(x) \text{ for } m = 0,$$

$$S_n^m(x) = (-1)^m \sqrt{\frac{2(n-m)!}{(n+m)!}} \, P_n^m(x) \text{ for } m > 0.$$

Fully Normalized Associated Legendre Functions

The fully normalized associated Legendre functions are normalized such that

$$\int_{-1}^{1}\left[N_n^m(x)\right]^2 dx = 1 \ .$$

The normalized functions are related to the unnormalized associated Legendre functions $P_n^m(x)$ by

$$N_n^m(x) = (-1)^m\sqrt{\frac{\left(n+\frac{1}{2}\right)(n-m)!}{(n+m)!}}P_n^m(x) \ .$$

Algorithms

`legendre` uses a three-term backward recursion relationship in `m`. This recursion is on a version of the Schmidt seminormalized associated Legendre functions $Q_n^m(x)$, which are complex spherical harmonics. These functions are related to the standard Abramowitz and Stegun [1] functions $P_n^m(x)$ by

$$P_n^m(x) = \sqrt{\frac{(n+m)!}{(n-m)!}}\ Q_n^m(x) \ .$$

They are related to the Schmidt form on page 1-8571 by

$$m = 0: \quad S_n^m(x) = Q_n^0(x)$$

$$m > 0: \quad S_n^m(x) = (-1)^m\sqrt{2}\ Q_n^m(x) \ .$$

References

[1] Abramowitz, M. and I. A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, 1965, Ch.8.

[2] Jacobs, J. A., *Geomagnetism*, Academic Press, 1987, Ch.4.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`besselj` | `bessely` | `beta` | `factorial` | `gamma`

Introduced before R2006a

length

Length of largest array dimension

Syntax

```
L = length(X)
```

Description

`L = length(X)` returns the length of the largest array dimension in `X`. For vectors, the length is simply the number of elements. For arrays with more dimensions, the length is `max(size(X))`. The length of an empty array is zero.

Examples

Number of Vector Elements

Find the length of a uniformly spaced vector in the interval `[5,10]`.

```
v = 5:10
```

```
v = 1×6

     5     6     7     8     9    10
```

```
L = length(v)
```

```
L = 6
```

Length of Rectangular Matrix

Find the length of a 3-by-7 matrix of zeros.

```
X = zeros(3,7);
L = length(X)
```

```
L = 7
```

String Array

Create a string array and compute its length, which is the number of elements in each row.

```
X = ["a" "b" "c"; "d" "e" "f"]
```

```
X = 2x3 string array
    "a"    "b"    "c"
    "d"    "e"    "f"
```

```
L = length(X)
```

```
L = 3
```

Length of Structure Fields

Create a structure with fields for `Day` and `Month`. Use the `structfun` function to apply `length` to each field.

```
S = struct('Day',[1 13 14 26],'Month',{{'Jan','Feb', 'Mar'}})
```

```
S = struct with fields:
      Day: [1 13 14 26]
    Month: {'Jan'  'Feb'  'Mar'}
```

```
L = structfun(@(field) length(field),S)
```

```
L = 2×1

     4
     3
```

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.
Complex Number Support: Yes

Tips

- To find the number of characters in a string or character vector, use the `strlength` function.
- `length` does not operate on tables. To examine the dimensions of a table, use the `height`, `width`, or `size` functions.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`ndims` | `numel` | `size` | `strlength`

Introduced before R2006a

length

Package: `containers`

Number of key-value pairs in `Map` object

Syntax

```
L = length(M)
```

Description

`L = length(M)` returns the number of key-value pairs in the input `Map` object. `length(M)` is equivalent to `size(M,1)` and to the property `M.Count`.

Examples

Length of Map

Create a `Map` object.

```
months = {'Jan','Feb','Mar','Apr'};
rainfall = [327.2 368.2 197.6 178.4];
M = containers.Map(months,rainfall)
```

```
M = 
  Map with properties:

        Count: 4
      KeyType: char
    ValueType: double
```

Return the number of key-value pairs that it contains.

```
L = length(M)
```

```
L = 4
```

Input Arguments

M — Input Map object
Map object

Input Map object.

See Also

containers.Map | isKey | keys | size | values

Topics

"Overview of Map Data Structure"
"Create Map Object"
"Examine Contents of Map"

Introduced in R2008b

length

Length of `tscollection` time vector

Syntax

```
tsclength = length(tsc)
```

Description

`tsclength = length(tsc)` returns the number of elements in the time vector of a `tscollection` object.

Examples

Length of `tscollection`

Create a `tscollection` object and display the length of its time vector.

```
ts = timeseries([1.1 2.9 3.7 4.0 3.0]',1:5);
tsc = tscollection(ts);
tsclength = length(tsc)
```

```
tsclength = 5
```

Input Arguments

`tsc` — Input `tscollection`
scalar

Input `tscollection`, specified as a scalar.

See Also

`size` | `timeseries` | `tscollection`

Introduced before R2006a

libfunctions

Return information on functions in shared C library

Syntax

```
libfunctions libname
```

```
m = libfunctions(libname)
m = libfunctions(libname,'-full')
```

Description

`libfunctions libname` displays names of functions defined in C library `libname`. If you called `loadlibrary` using the `alias` option, then you must use the alias name for the `libname` argument.

`m = libfunctions(libname)` returns names of functions in cell array `m`.

`m = libfunctions(libname,'-full')` returns function signatures.

Examples

Display Functions in `shrlibsample` Library

Add path to `examples` folder.

```
addpath(fullfile(matlabroot,'extern','examples','shrlib'))
```

Display functions in library.

```
if not(libisloaded('shrlibsample'))
    loadlibrary('shrlibsample')
end
libfunctions('shrlibsample')
```

```
Functions in library shrlibsample:

addDoubleRef              exportedDoubleValue       printExportedDoubleValue
addMixedTypes             getListOfStrings          readEnum
addStructByRef            multDoubleArray           stringToUpper
addStructFields           multDoubleRef
allocateStruct            multiplyShort
deallocateStruct          print2darray
```

Clean up.

```
unloadlibrary shrlibsample
```

Input Arguments

`libname` — Name of shared library
character vector

Name of shared library, specified as a character vector. Do not include the path or file extension in `libname`.

If you call `loadlibrary` using the `alias` option, then you must use the alias name for the `libname` argument.

Data Types: `char`

Output Arguments

`m` — Function names
cell array

Functions names, returned as a cell array.

Limitations

- Use with libraries that are loaded using the `loadlibrary` function.

See Also

`calllib` | `libfunctionsview` | `loadlibrary`

Introduced before R2006a

libfunctionsview

Display shared C library function signatures in window

Syntax

```
libfunctionsview libname
```

Description

`libfunctionsview libname` displays information about functions in C library `libname` in a new window.

Input Arguments

`libname` — Name of shared library
character vector

Name of shared library, specified as a character vector. Do not include the path or file extension in `libname`.

If you call `loadlibrary` using the `alias` option, then you must use the alias name for the `libname` argument.

Data Types: `char`

Limitations

- Use with libraries that are loaded using the `loadlibrary` function.

Examples

Display Function Signatures for Library shrlibsample

```
if not(libisloaded('shrlibsample'))
    addpath(fullfile(matlabroot,'extern','examples','shrlib'))
    loadlibrary('shrlibsample')
end
libfunctionsview shrlibsample
```

MATLAB creates a new window displaying function signatures.

When finished, unload the library.

```
unloadlibrary shrlibsample
```

See Also

`calllib` | `libfunctions`

Introduced before R2006a

libisloaded

Determine if shared C library is loaded

Syntax

```
tf = libisloaded(libname)
```

Description

`tf = libisloaded(libname)` returns logical `1` (`true`) if the shared C library `libname` is loaded. Otherwise, it returns logical `0` (`false`).

Examples

Load and Unload shrlibsample Library

Add the `shrlib` examples folder to the path.

```
addpath(fullfile(matlabroot,'extern','examples','shrlib'))
```

Load the library, if it is not already loaded.

```
if ~libisloaded('shrlibsample')
   loadlibrary('shrlibsample')
end
```

When finished, unload the library to free memory.

```
unloadlibrary shrlibsample
```

Input Arguments

`libname` — Name of shared library
character vector

Name of shared library, specified as a character vector. Do not include the path or file extension in `libname`.

If you call `loadlibrary` using the `alias` option, then you must use the alias name for the `libname` argument.

Data Types: `char`

See Also

`loadlibrary` | `unloadlibrary`

Introduced before R2006a

libpointer

Pointer object for use with shared C library

Syntax

```
p = libpointer
p = libpointer(DataType)
p = libpointer(DataType,Value)
```

Description

`p = libpointer` creates `NULL` pointer `p` of type `voidPtr`.

`p = libpointer(DataType)` creates `NULL` pointer of specified `DataType`.

`p = libpointer(DataType,Value)` creates pointer initialized to a copy of `Value`.

Examples

Create NULL Pointer for string Argument

```
p = libpointer('string');
```

Input Arguments

`DataType` — Type of pointer
character vector

Type of pointer, specified as a character vector, of any MATLAB numeric type, structure defined in the library, or enumeration defined in the library. For a list of valid MATLAB numeric types, refer to these tables in “C and MATLAB Equivalent Types”.

- MATLAB Primitive Types
- MATLAB Extended Types

Example: `'int16Ptr'`

Data Types: `char`

`Value` — Value for pointer object
any valid value

Value, specified as any valid value for given type.

Limitations

- Use with libraries that are loaded using the `loadlibrary` function.

Tips

- This is an advanced feature for experienced C programmers. MATLAB automatically converts data passed to and from external library functions to the data type expected by the external function. Use a `lib.pointer` object instead of automatic conversion in the following situations.
  - You want to modify the data in the input arguments.
  - You are passing large amounts of data, and you want to control when MATLAB makes copies of the data.
  - The library stores and uses the pointer for a period of time so you want the MATLAB function to control the lifetime of the `lib.pointer` object.

See Also

`calllib` | `lib.pointer` | `libstruct`

Topics

"Pass Pointers Examples"
"Pointer Arguments"

Introduced before R2006a

lib.pointer class

Package: `lib`

Pointer object compatible with C pointer

Description

MATLAB automatically converts arguments passed by reference to a function in an external library. A passed-by-reference argument in the function signature has type names ending with `Ptr` or `PtrPtr`. Use a pointer object instead of automatic conversion in the following situations.

- The function modifies data in an input argument.
- You are passing large amounts of data, and you want to control when MATLAB makes copies of the data.
- The library stores and uses the pointer, so you want the MATLAB function to control the lifetime of the `lib.pointer` object.

Creation

To create a `lib.pointer` object, use the MATLAB `libpointer` function.

A library function can return a `lib.pointer` object. Use the `setdatatype` method to convert the argument manually to use in MATLAB.

Properties

`DataType` — Type of pointer

character vector

Type of pointer, specified as a character vector, of any MATLAB numeric type, structure defined in the library, or enumeration defined in the library. For a list of valid MATLAB numeric types, refer to these tables in “C and MATLAB Equivalent Types”.

Example: `'int16Ptr'`

Attributes:

SetAccess	private
GetAccess	protected

`Value` — Value
any valid value

Value, specified as any valid value for given type.

Attributes:

SetAccess	private
GetAccess	protected

Methods

Public Methods

disp	Display lib.pointer type
isNull	Points to NULL pointer
plus	+ (plus) operator for pointer arithmetic
reshape	Reshape lib.pointer array
setdatatype	Initialize type and size of lib.pointer object

Examples

Create Pointer

Create a pointer `pv` of type `int16` initialized to `485`.

```
pv = libpointer('int16Ptr',485);
```

Display the properties of `pv`.

```
get(pv)
```

```
       Value: 485
    DataType: 'int16Ptr'
```

See Also

`libpointer`

Topics

"Pointer Arguments"

Introduced before R2006a

disp

Class: `lib.pointer`
Package: `lib`

Display lib.pointer type

Syntax

```
disp(h)
```

Description

`disp(h)` displays type for `lib.pointer` object, `h`.

Input Arguments

`h` — `lib.pointer` object.
handle

`lib.pointer` object, specified as a handle.

Examples

Display lib.pointer Type

Create a `double` pointer set to `15`.

```
xp = libpointer('doublePtr',15);
```

Display pointer data type.

```
disp(xp)
```

`libpointer`

See Also

isNull

Class: `lib.pointer`
Package: `lib`

Points to NULL pointer

Syntax

```
tf = isNull(h)
```

Description

`tf = isNull(h)` returns true if `h` is a `lib.pointer` object.

Input Arguments

`h` — `lib.pointer` object.
handle

`lib.pointer` object, specified as a handle.

Examples

Create Null lib.pointer Object

```
nullp = libpointer('doublePtr',[]);
isNull(nullp)
```

```
ans = logical
   1
```

See Also

plus

Class: `lib.pointer`
Package: `lib`

+ (plus) operator for pointer arithmetic

Syntax

```
hout = plus(h,offset)
hout = h + offset
```

Description

`hout = plus(h,offset)` returns pointer `hout`. Pointer `hout` is valid only as long as the original pointer, `h`, exists.

`hout = h + offset` is an alternative syntax.

Input Arguments

`h` — `lib.pointer` object.
handle

`lib.pointer` object, specified as a handle.

`offset` — Scalar increment
`uint64`

Scalar increment, specified as `uint64`, from `h`.

See Also

Topics

"Iterate Through an Array"

reshape

Class: `lib.pointer`
Package: `lib`

Reshape lib.pointer array

Syntax

```
reshape(h,xdim,ydim)
```

Description

`reshape(h,xdim,ydim)` creates an `xdim`-by-`ydim` matrix from `lib.pointer` object `h`.

Input Arguments

`h` — `lib.pointer` object.
handle

`lib.pointer` object, specified as a handle.

`xdim` — Size of x dimension
double

Size of x dimension, specified as `double`.

`ydim` — Size of y dimension
double

Size of y dimension, specified as `double`.

See Also

setdatatype

Class: `lib.pointer`
Package: `lib`

Initialize type and size of lib.pointer object

Syntax

```
setdatatype(h,type,sizeD1,...,sizeDn)
```

Description

`setdatatype(h,type,sizeD1,...,sizeDn)` sets data type to `lib.pointer` `h`.

Input Arguments

`h` — `lib.pointer` object.
handle

`lib.pointer` object, specified as a handle.

`type` — Data type
character vector

Data type, specified as a character vector.

`sizeD1,...,sizeDn` — Size of each dimension
double

Size of each dimension, specified as `double`.

Examples

Set Size and Type of lib.pointer Output Variable

Load the `shrlibsample` library.

```
if ~libisloaded('shrlibsample')
  addpath(fullfile(matlabroot,'extern','examples','shrlib'))
  loadlibrary('shrlibsample')
end
```

The `multDoubleRef` function takes a scalar value specified as `doubleptr`. Create variable `xp` as a `lib.pointer` object, and call the function.

```
xp = libpointer('doublePtr',99);
[xobj,xval] = calllib('shrlibsample','multDoubleRef',xp);
```

To use the variable `xobj`, set its size and data type.

```
setdatatype(xobj,'doublePtr',1,1)
xobj.Value
```

```
ans = 495
```

See Also

Topics

"Pass Arguments to Shared C Library Functions"

libstruct

Convert MATLAB structure to C-style structure for use with shared C library

Syntax

```
S = libstruct(structtype)
S = libstruct(structtype,mlstruct)
```

Description

`S = libstruct(structtype)` creates `NULL` pointer to MATLAB `libstruct` object `S`.

`S = libstruct(structtype,mlstruct)` creates pointer initialized to `mlstruct`.

Examples

Call Function with `c_struct` Input Argument

Call the `addStructFields` function by creating a variable of type `c_struct`.

Load the `shrlibsample` library in the *`matlabroot`*`\extern\examples\shrlib` folder, which contains the `c_struct` type.

```
if ~libisloaded('shrlibsample')
   addpath(fullfile(matlabroot,'extern','examples','shrlib'))
   loadlibrary('shrlibsample')
end
```

Display function signatures for `shrlibsample` and search the list for the `addStructFields` entry.

```
libfunctionsview shrlibsample
```

```
double addStructFields(c_struct)
```

The input argument is a pointer to a `c_struct` data type.

Create a MATLAB structure, `sm`.

```
sm.p1 = 476;   sm.p2 = -299;   sm.p3 = 1000;
```

Construct a `libstruct` object `sc` from the `c_struct` type.

```
sc = libstruct('c_struct',sm)
```

The fields of `sc` contain the values of the MATLAB structure, `sm`.

Call the `addStructFields` function.

```
calllib('shrlibsample','addStructFields',sc)
```

```
ans =
    1177
```

To clean up, first clear the `libstruct` object, and then unload the library.

```
clear sc
unloadlibrary shrlibsample
```

Input Arguments

`structtype` — C structure
structure

C structure defined in shared library.

`mlstruct` — MATLAB structure
structure

MATLAB structure used to initialize the fields in S.

Data Types: `struct`

Output Arguments

S — Pointer
MATLAB `libstruct` object

Pointer, returned as MATLAB `libstruct` object.

Limitations

- Use with libraries that are loaded using the `loadlibrary` function.
- You can only use the `libstruct` function on scalar structures.
- When converting a MATLAB structure to a `libstruct` object, the structure must adhere to the requirements listed in “Structure Argument Requirements”.

Tips

- If a function in the shared library has a structure argument, use `libstruct` to create the argument. The `libstruct` function creates a C-style structure that you pass to functions in the library. You handle this structure in MATLAB as you would a true MATLAB structure.

See Also

`libfunctionsview` | `loadlibrary`

Topics

“Shared Library shrlibsample”
“Structure Arguments”

Introduced before R2006a

license

Get license number or perform licensing task

Syntax

```
license
license('inuse')

S = license('inuse')
S = license('inuse',feature)

status = license('test',feature)
license('test',feature,toggle)

[status,errmsg] = license('checkout',feature)
```

Description

`license` returns the license number for this MATLAB product. The return value also can be `'demo'` for a demonstration version of MATLAB, `'student'`, for a student version of MATLAB, or `'unknown'`, if the license number cannot be determined.

`license('inuse')` displays a list of licenses checked out in the current MATLAB session. The product list is alphabetical by license feature name. These names are the same as the valid values for the `feature` input.

`S = license('inuse')` returns an array of structures indicating checked-out licenses and the user name of each person who has a license checked out.

`S = license('inuse',feature)` checks if the product specified by `feature` is checked out in the current MATLAB session. If the product is checked out, then `license` returns the product name and the user name of the person who has it checked out. Otherwise, the fields of `S` are empty.

`status = license('test',feature)` tests if a license exists for the product specified by `feature`.

`license('test',feature,toggle)` enables or disables testing of the product specified by `feature`, depending on the value of `toggle`.

`[status,errmsg] = license('checkout',feature)` checks out a license for the specified product. If you specify the optional second output argument, `errmsg`, then `license` returns the text of any error message encountered if the checkout is unsuccessful.

Examples

Display Licenses in Use

Display a list of licenses currently being used. `license` displays a list of products in alphabetical order by the license feature name for your configuration.

```
license('inuse')
```

```
image_toolbox
map_toolbox
matlab
```

Get Licenses in Use and User Names

Get a list of licenses in use with information about each user. `license` returns a structure array.

```
S = license('inuse');
```

View the first element of `S`.

```
S(1)
```

```
ans = 

    feature: 'image_toolbox'
       user: 'juser'
```

Determine If License Is in Use

Determine if the license for MATLAB is in use. If the license is in use, then `S` is a structure array with nonempty fields.

```
S = license('inuse','MATLAB')
```

```
S =

    feature: 'matlab'
       user: 'jsmith'
```

Determine If License Exists

Determine if a license exists for Mapping Toolbox. If a license exists, then `license` returns `1`.

```
status = license('test','MAP_Toolbox')
```

```
status =

    1
```

Check Out License

Check out a license for Control System Toolbox™. The `status` output is `1` and the `errmsg` output is empty if the checkout is successful.

```
[status,errmsg] = license('checkout','Control_Toolbox')
```

```
status =

     1


errmsg =
```

```
''
```

Input Arguments

`feature` — License feature name
character vector

License feature name, specified as a character vector. Values of `feature` are not case-sensitive. The `INCREMENT` lines in a license file indicate the valid features. To locate your license file, see Where are the license files for MATLAB located?

`toggle` — Ability to test product license
`'enable'` | `'disable'`

Ability to test the existence of a product license, specified as either `'enable'` or `'disable'`.

- If `toggle` is `'enable'`, then the syntax, `license('test',feature)` returns `1` when the product license exists and `0` when the product license does not exist.
- If `toggle` is `'disable'`, then the syntax, `license('test',feature)` always returns `0` (product license does not exist) for the specified product.

Note Disabling a test for a particular product can affect other tests for the existence of the license, not just tests performed by the `license` command.

Output Arguments

S — Checked out products
array of structures

Checked out products, returned as an array of structures, where each structure represents a checked-out license. The structures contain two fields:

- `feature` — license feature name
- `user` — user name of the person who has the license checked out

If the fields are empty, then the product is not currently checked out.

`status` — Test or checkout status
`1` | `0`

Test or checkout status, returned as `1` or `0`.

- When testing for the existence of a license, `1` indicates that the license exists, and `0` indicates that the license does not exist.

  The existence of a license does not necessarily mean that the license can be checked out or that the product is installed. `status` is `1` even if the license has expired or if a system administrator has excluded you from using the product.
- When checking out a license, `1` indicates that the checkout is successful, and `0` indicates that the `license` function could not check out a license.

`errmsg` — Error message
character vector

Error message for unsuccessful license checkout, returned as a character vector. If the checkout is successful, then `errmsg` is empty.

See Also
`isstudent`

Topics
"Manage Your Licenses"

Introduced before R2006a

light

Create light

Syntax

```
light('PropertyName',propertyvalue,...)
light(ax,...)
handle = light(...)
```

Properties

For a list of properties, see Light.

Description

`light` creates a light in the current axes. Lights affect only patch and surface objects.

`light('PropertyName',propertyvalue,...)` creates a `Light` object using the specified values for the named properties. For a description of the properties, see Light. The MATLAB software parents the light to the current axes unless you specify another axes with the `Parent` property.

`light(ax,...)` creates the light in the axes specified by `ax` instead of in the current axes (`gca`). The option `ax` can precede any of the input argument combinations in the previous syntaxes.

`handle = light(...)` returns the `Light` object created.

Examples

Light the `peaks` surface plot with a local light source oriented along the direction defined by the vector `[-1 0 0]`, that is, looking from `0` along the positive x-axis.

```
surf(peaks,'FaceLighting','gouraud','FaceColor','interp',...
      'AmbientStrength',0.5)
light('Position',[-1 0 0],'Style','local')
```

Tutorials

For more information about lighting, see “Lighting Overview”.

Tips

You cannot see a `Light` object *per se*, but you can see the effects of the light source on patch and surface objects. You can also specify an axes-wide ambient light color that illuminates these objects. However, ambient light is visible only when at least one `Light` object is present and visible in the axes.

You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see `set` and `get` for examples of how to specify these data types).

See also the Patch and Primitive Surface `AmbientStrength`, `DiffuseStrength`, `SpecularStrength`, `SpecularExponent`, `SpecularColorReflectance`, and `VertexNormals` properties. Also see the `lighting` and `material` commands.

See Also

`lighting` | `material` | `patch` | `surface`

Introduced before R2006a

Light Properties

Light appearance and behavior

Description

`Light` properties control the appearance and behavior of `Light` objects. By changing property values, you can modify certain aspects of the light.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = light;
c = h.Color;
h.Style = 'local';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Color and Styling

`Color` — Color of light

`[1 1 1]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of light, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default RGB triplet of `[1 1 1]` corresponds to white.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'green'`

`Style` — Type of light source
`'infinite'` (default) | `'local'`

Type of light source, specified as one of these values:

- `'infinite'` — Place the light at infinity. Use the `Position` property to specify the direction from which the light shines in parallel rays.

- `'local'` — Place the light at the location specified by the `Position` property. The light is a point source that radiates from the location in all directions.

`Position` — Location of light source

`[1 0 1]` (default) | three-element vector of the form `[x y z]`

Location of light source, specified as a three-element vector of the form `[x y z]`. Define the vector elements in data units from the axes origin to the (*x*, *y*, *z*) coordinate. The actual location of the light depends on the value of the `Style` property.

Example: `[-40 -4 140]`

Position

`Position` — Location of light source

`[1 0 1]` (default) | three-element vector of the form `[x y z]`

Location of light source, specified as a three-element vector of the form `[x y z]`. Define the vector elements in data units from the axes origin to the (*x*, *y*, *z*) coordinate. The actual location of the light depends on the value of the `Style` property.

Example: `[-40 -4 140]`

`Visible` — Visibility of light from light source

`'on'` (default) | `'off'`

Visibility of light from light source, specified as `'on'` or `'off'`.

Callbacks

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent
`Axes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `Group`, or `Transform` object.

`Children` — Children
empty `GraphicsPlaceholder` array

The object has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object
`'light'`

This property is read-only.

Type of graphics object, returned as `'light'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

Unused Properties

`ButtonDownFcn` — (unused) Mouse-click callback
`''` (default) | function handle | cell array | character vector

`Light` objects do not use this property.

`UIContextMenu` — (unused) Context menu
uicontextmenu object

`Light` objects do not use this property.

`Selected` — (unused) Selection state
`'off'` (default) | `'on'`

`Light` objects do not use this property.

`SelectionHighlight` — (unused) Display of selection handles when selected
`'on'` (default) | `'off'`

`Light` objects do not use this property.

`PickableParts` — (unused) Ability to capture mouse clicks
`'visible'` (default) | `'none'`

`Light` objects do not use this property.

`HitTest` — (unused) Response to captured mouse clicks
`'on'` (default) | `'off'`

Light objects do not use this property.

`Interruptible` — (unused) Callback interruption
`'on'` (default) | `'off'`

`Light` objects do not use this property.

`BusyAction` — (unused) Callback queuing
`'queue'` (default) | `'cancel'`

`Light` objects do not use this property.

See Also

`light`

Topics

"Access Property Values"
"Graphics Object Properties"

Introduced before R2006a

lightangle

Create or position light object in spherical coordinates

Syntax

```
lightangle(az,el)
lightangle(ax,az,el)
lgt = lightangle(...)
lightangle(lgt,az,el)
[az,el] = lightangle(lgt)
```

Description

`lightangle(az,el)` creates a light at the position specified by azimuth and elevation. `az` is the azimuthal (horizontal) rotation and `el` is the vertical elevation (both in degrees). The interpretation of azimuth and elevation is the same as that of the `view` command.

`lightangle(ax,az,el)` creates a light in the axes specified by `ax` instead of the current axes.

`lgt = lightangle(...)` creates a light and returns the light object as `lgt`. You can specify an output argument with any of the previous input argument combinations.

`lightangle(lgt,az,el)` sets the position of the light specified by `lgt`.

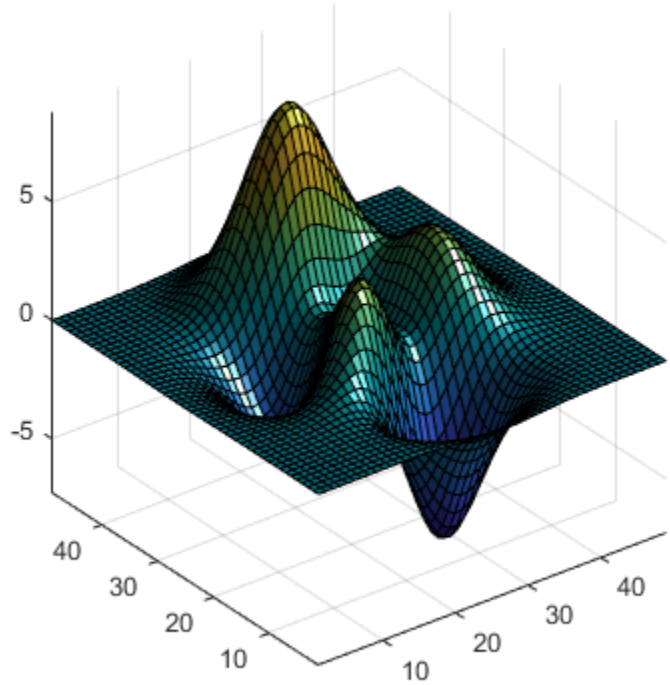
`[az,el] = lightangle(lgt)` returns the azimuth and elevation of the light specified by `lgt`.

Examples

Move Light Position

Create a surface. Add a light and move the position of the light.

```
surf(peaks)
axis vis3d
h = light;
for az = -50:10:50
   lightangle(h,az,30)
   pause(.1)
end
```



Tips

By default, when a light is created, its style is `infinite`. If the light handle passed in to `lightangle` refers to a local light, the distance between the light and the camera target is preserved as the position is changed.

See Also

`camlight` | `light` | `view`

Topics

"Lighting Overview"

Introduced before R2006a

lighting

Specify lighting algorithm

Syntax

```
lighting flat
lighting gouraud
lighting none
lighting(ax,...)
```

Description

`lighting` selects the algorithm used to calculate the effects of `light` objects on all `surface` and `patch` objects in the current axes. In order for the `lighting` command to have any effects, however, you must create a lighting object by using the `light` function.

`lighting flat` produces uniform lighting across each of the faces of the object. Select this method to view faceted objects.

`lighting gouraud` calculates the vertex normals and interpolates linearly across the faces. Select this method to view curved surfaces.

`lighting none` turns off lighting.

`lighting(ax,...)` uses the axes specified by `ax` instead of the current axes.

Tips

The `surf`, `mesh`, `pcolor`, `fill`, `fill3`, `surface`, and `patch` functions create graphics objects that are affected by light sources. The `lighting` command sets the `FaceLighting` and `EdgeLighting` properties of surfaces and patches appropriately for the graphics object.

See Also

`fill` | `fill3` | `light` | `material` | `mesh` | `patch` | `pcolor` | `shading` | `surface`

Topics

"Lighting Overview"

Introduced before R2006a

lin2mu

Convert linear audio signal to mu-law

Syntax

```
mu = lin2mu(y)
```

Description

`mu = lin2mu(y)` converts linear audio signal amplitudes in the range `-1 ≤ Y ≤ 1` to mu-law encoded “flints” in the range `0 ≤ u ≤ 255`.

See Also

`mu2lin`

Introduced before R2006a

line

Create primitive line

Syntax

```
line(x,y)
line(x,y,z)
line

line( ___ ,Name,Value)
line(ax, ___ )

pl = line( ___ )
```

Description

`line(x,y)` plots a line in the current axes using the data in vectors `x` and `y`. If either `x` or `y`, or both are matrices, then `line` draws multiple lines. Unlike the `plot` function, `line` adds the line to the current axes without deleting other graphics objects or resetting axes properties.

`line(x,y,z)` plots a line in three-dimensional coordinates.

`line` draws a line from the point `(0,0)` to `(1,1)` with the default property settings.

`line( ___ ,Name,Value)` modifies the appearance of the line using one or more name-value argument pairs. For example, `'LineWidth',3` sets the line width to `3` points. Specify name-value pairs after all other input arguments. If you specify the data using name-value pairs, for example `line('XData','x','YData','y')`, then you must specify vector data.

`line(ax, ___ )` creates the line in the Cartesian, polar, or geographic axes specified by `ax` instead of in the current axes (`gca`). Specify `ax` as the first input argument.

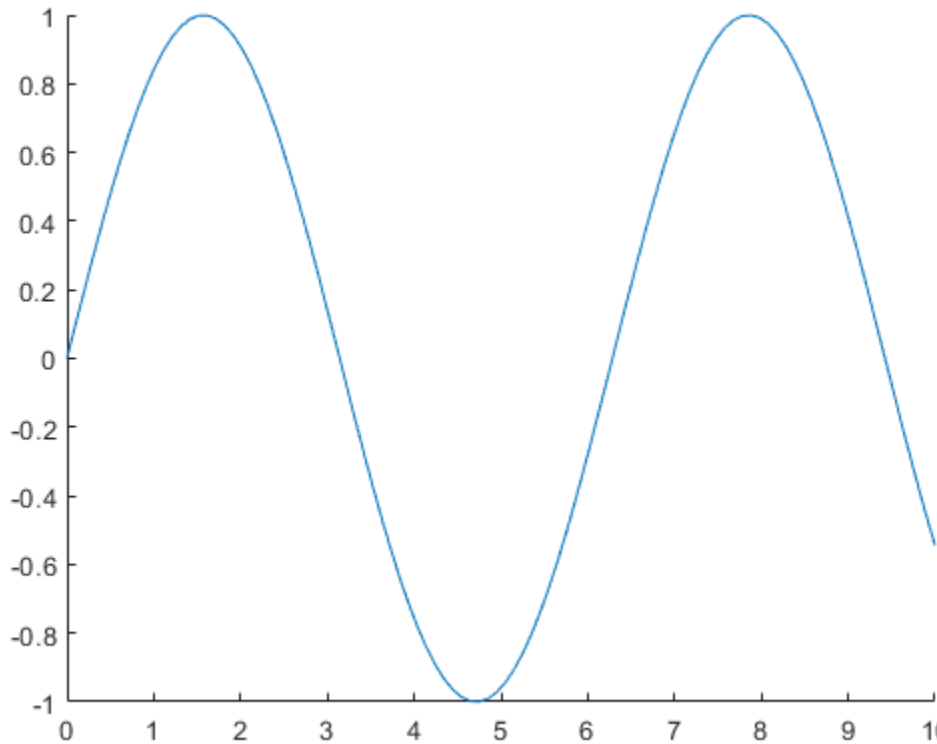
`pl = line( ___ )` returns all primitive `Line` objects created. Use `pl` to modify properties of a specific `Line` object after it is created. For a list, see Primitive Line.

Examples

Plot Line Using Vector Data

Create x and y as vectors. Then plot y versus x.

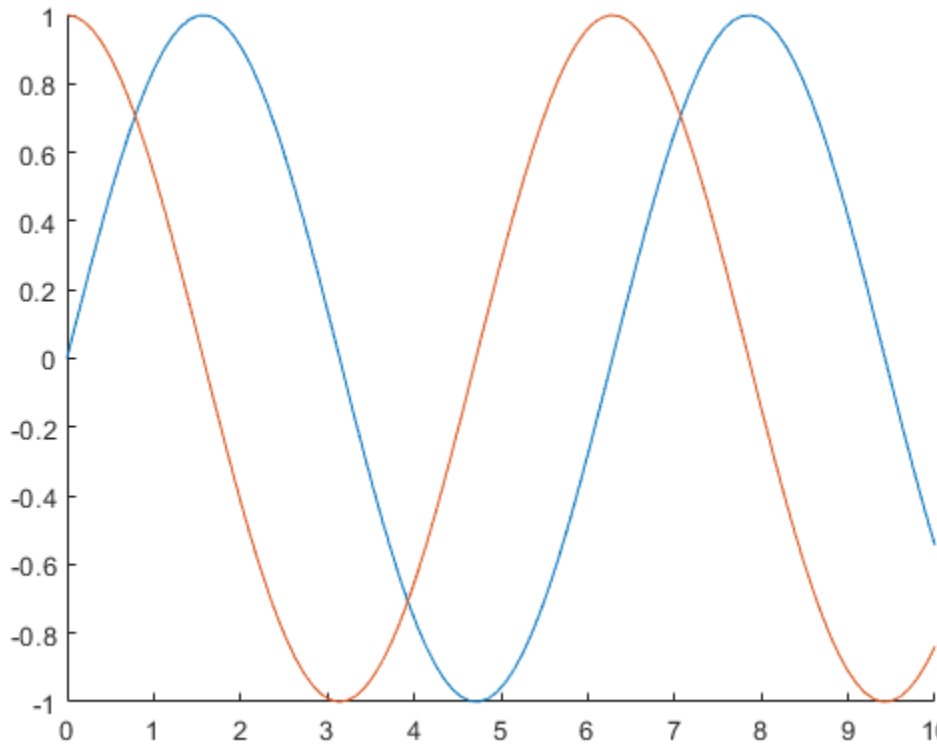
```
x = linspace(0,10);
y = sin(x);
line(x,y)
```



Plot Multiple Lines Using Matrix Data

Plot two lines by specifying `x` and `y` as matrices. Use `line` to plot columns of `y` versus columns of `x` as separate lines.

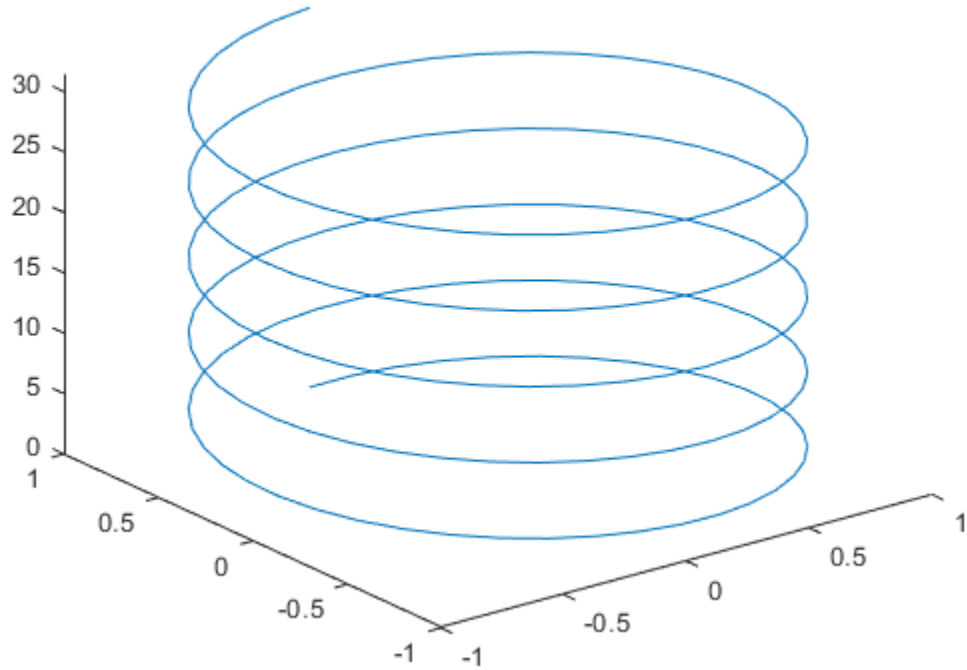
```
x = linspace(0,10)';
y = [sin(x) cos(x)];
line(x,y)
```



Plot Line with 3-D Coordinates

Plot a line in 3-D coordinates by specifying *x*, *y*, and *z* values. Change the axes to a 3-D view using `view(3)`.

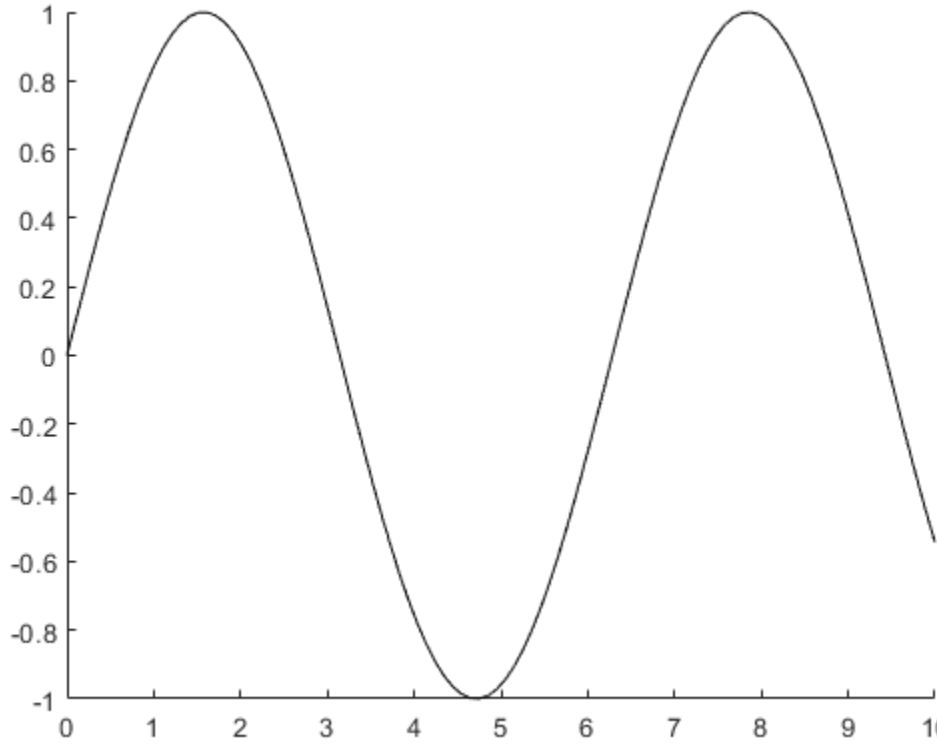
```
t = linspace(0,10*pi,200);
x = sin(t);
y = cos(t);
z = t;
line(x,y,z)
view(3)
```



Plot Line Using Low-Level Syntax

Create `x` and `y` as vectors. Then plot `y` versus `x` using the low-level version of the `line` function.

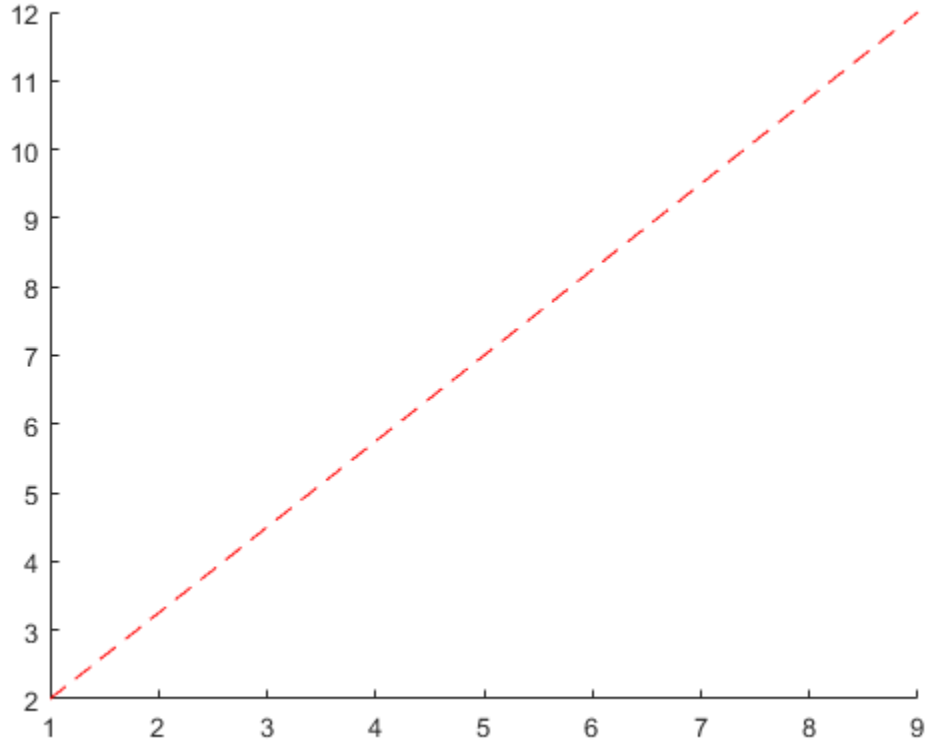
```
x = linspace(0,10);
y = sin(x);
line('XData',x,'YData',y)
```



Specify Line Properties

Draw a red, dashed line between the points (1,2) and (9,12). Set the `Color` and `LineStyle` properties as name-value pairs.

```
x = [1 9];
y = [2 12];
line(x,y,'Color','red','LineStyle','--')
```

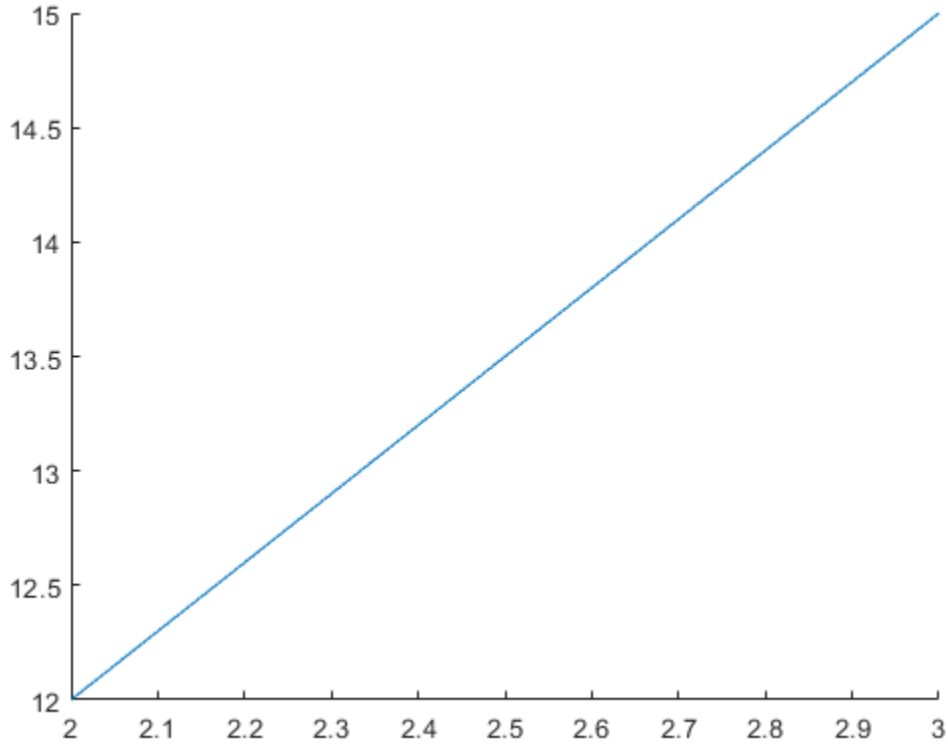


Change Line Properties After Creation

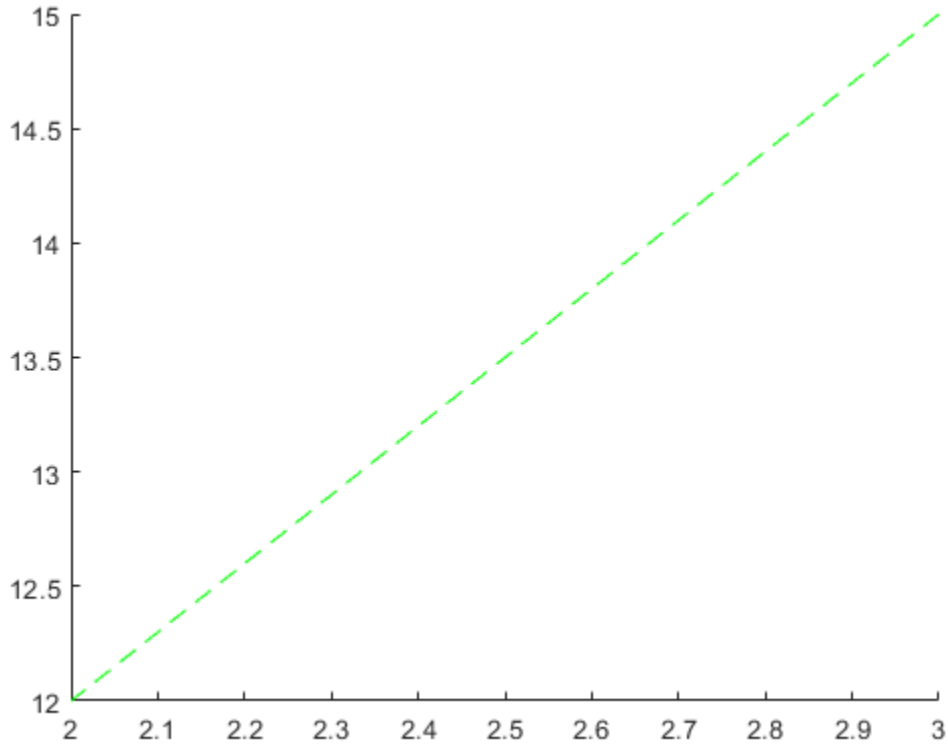
First, draw a line from the point (3,15) to (2,12) and return the `Line` object. Then change the line to a green, dashed line.

Note: Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead.

```
x = [3 2];
y = [15 12];
pl = line(x,y);
```




```
pl.Color = 'green';
pl.LineStyle = '--';
```



Input Arguments

x — First coordinate
vector | matrix

First coordinate, specified as a vector or a matrix. Matrix inputs are supported for Cartesian axes only.

The interpretation of the first coordinate depends on the type of axes. For Cartesian axes, the first coordinate is *x*-axis position in data units.

- If `x` and `y` are both vectors with the same length, then `line` plots a single line.
- If `x` and `y` are matrices with the same size, then `line` plots multiple lines. The function plots columns of `y` versus `x`.
- If one of `x` or `y` is a vector and the other is a matrix, then `line` plots multiple lines. The length of the vector must equal one of the matrix dimensions:
  - If the vector length equals the number of matrix rows, then `line` plots each matrix column versus the vector.
  - If the vector length equals the number of matrix columns, then `line` plots each matrix row versus the vector.
  - If the matrix is square, then `line` plots each column versus the vector.

For polar axes, the first coordinate is the polar angle θ in radians. For geographic axes, the first coordinate is latitude in degrees. To plot lines in these types of axes, `x` and `y` must be the same size.

Example: `x = linspace(0,10,25)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

y — Second coordinate
vector | matrix

Second coordinate, specified as a vector or a matrix. Matrix inputs are supported for Cartesian axes only.

The interpretation of the second coordinate depends on the type of axes. For Cartesian axes, the second coordinate is *y*-axis position in data units.

- If `x` and `y` are both vectors with the same length, then `line` plots a single line.
- If `x` and `y` are matrices with the same size, then `line` plots multiple lines. The function plots columns of `y` versus `x`.
- If one of `x` or `y` is a vector and the other is a matrix, then `line` plots multiple lines. The length of the vector must equal one of the matrix dimensions:
  - If the vector length equals the number of matrix rows, then `line` plots each matrix column versus the vector.
  - If the vector length equals the number of matrix columns, then `line` plots each matrix row versus the vector.

- If the matrix is square, then `line` plots each column versus the vector.

For polar axes, the second coordinate is the radius in data units. For geographic axes, the second coordinate is longitude in degrees. To plot lines in these types of axes, `x` and `y` must be the same size.

Example: `y = sin(x)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

z — Third coordinate
vector | matrix

Third coordinate, specified as a vector or a matrix. Matrix inputs are supported for Cartesian axes only.

The interpretation of the third coordinate depends on the type of axes. For Cartesian axes, the third coordinate is *z*-axis position in data units.

- If `x`, `y`, and `z` are all vectors with the same length, then `line` plots a single 3-D line.
- If `x`, `y`, and `z` are all matrices with the same size, then `line` plots multiple 3-D lines using the matrix columns.
- If one or two of `x`, `y`, and `z` is a vector and the others are matrices of the same size, then `line` plots multiple 3-D lines. The length of the vector must equal one of the matrix dimensions.

For polar and geographic axes, the third coordinate affects the layering of 2-D lines on the axes. To use the third coordinate in these types of axes, `x`, `y`, and `z` must be the same size.

Example: `z = sin(x) + cos(y)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

ax — Target axes
`Axes` object | `PolarAxes` object | `GeographicAxes` object

Target axes, specified as an `Axes` object, a `PolarAxes` object, or a `GeographicAxes` object. If you do not specify the axes, then the `line` function plots in the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The properties listed here are only a subset. For a full list, see Primitive Line.

Example: `line(x,y,'Color','red','LineWidth',3)` creates a red line that is 3 points wide.

`Color` — Line color

`[0 0.4470 0.7410]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `line(x,y,'Color','blue')`

Example: `line(x,y,'Color',[0.5 0.5 1])`

Example: `line(x,y,'Color','#D9A2E9')`

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	

Line Style	Description	Resulting Line
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

`Marker` — Marker symbol
`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | ...

Marker symbol, specified as one of the values listed in this table. By default, the object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)

Value	Description
`'none'`	No markers

`MarkerSize` — Marker size
6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

Output Arguments

`pl` — Primitive line object
primitive line object

Primitive line object. Use `pl` to query or modify properties of the line after it is created. For a list, see Primitive Line.

Tips

- Unlike the `plot` function, the `line` function does not call `newplot` before plotting and does not respect the value of the `NextPlot` property for the figure or axes. It simply adds the line to the current axes without deleting other graphics objects or resetting axes properties. However, some axes properties, such as the axis limits, can update to accommodate the line.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions
`annotation` | `loglog` | `plot` | `plot3` | `xline` | `yline`

Properties
Primitive Line

Topics

“Plot Dates and Durations”
“Plot Categorical Data”

Introduced before R2006a

lines

Lines colormap array

Syntax

```
c = lines
c = lines(m)
```

Description

`c = lines` returns the lines colormap as a three-column array with the same number of rows as the colormap for the current figure. If no figure exists, then the array contains 256 rows. Each row in the array contains the red, green, and blue intensities for a specific color. The intensities are in the range [0,1], and the color scheme matches the default `ColorOrder` property of the `Axes`. Thus, you can use this colormap to make your chart colors match the output of the `plot` function.

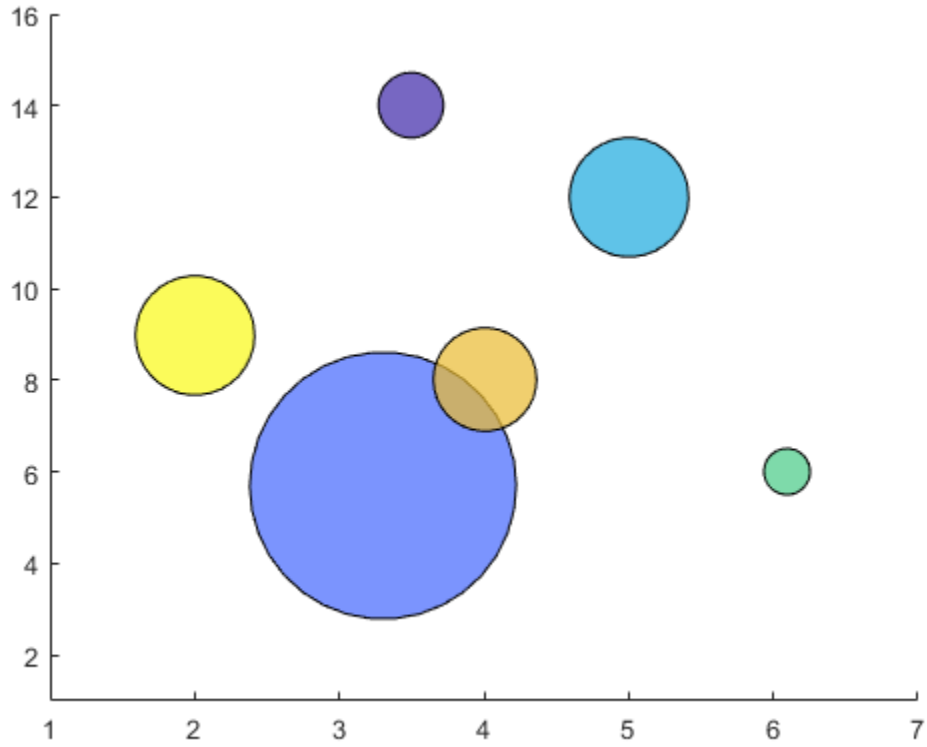
`c = lines(m)` returns the colormap with `m` colors.

Examples

Downsample the Lines Colormap

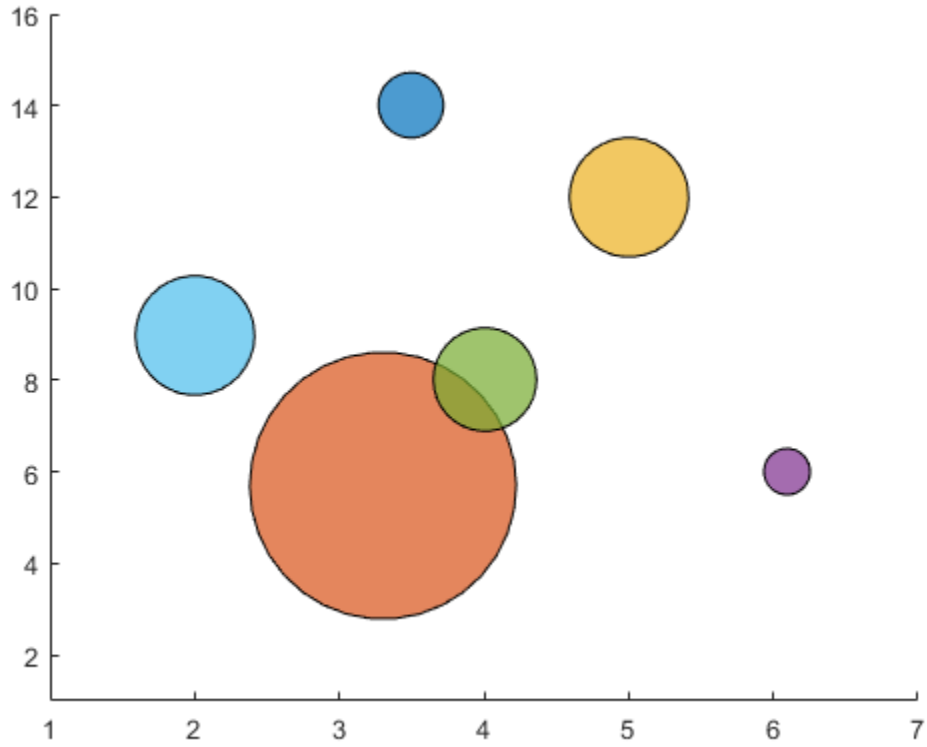
Create a scatter plot using the default colors.

```
x = [3.5 3.3 5 6.1 4 2];
y = [14 5.7 12 6 8 9];
sz = 100*[6 100 20 3 15 20];
c = [1 2 3 4 5 6];
scatter(x,y,sz,c,'filled','MarkerEdgeColor','k','MarkerFaceAlpha',.7);
xlim([1 7]);
ylim([1 16]);
```



Get the lines colormap array with six entries. Then replace the colormap in the scatter plot.

```
c = lines(6);
colormap(c);
```



Input Arguments

`m` — Number of colors
`256` (default) | scalar integer value

Number of colors, specified as a scalar integer value. The default value of `m` is equal to the length of the colormap for the current figure. If no figure exists, the default value is 256.

Data Types: `single` | `double`

Compatibility Considerations

Colormaps Have 256 Colors by Default

Behavior changed in R2019b

Starting in R2019b, colormaps have 256 colors by default.

In R2019a and previous releases, the default size is 64. If you have code that depends on a colormap having 64 colors, specify the number of colors when you set the colormap for the figure, axes, or chart. For example, `colormap(lines(64))` sets the figure's colormap to the 64-color `lines` colormap.

Alternatively, you can change the default colormap for all figures within your MATLAB session:

```
set(groot,'defaultFigureColormap',lines(64))
```

See Also

`colormap`

Topics

"Change Color Scheme Using a Colormap"

Introduced before R2006a

LineSpec (Line Specification)

Line specification

Description

Plotting functions accept line specifications as arguments and modify the graph generated accordingly. You can specify these three components:

- Line style
- Marker symbol
- Color

For example:

```
plot(x,y,'-.or')
```

plots `y` versus `x` using a dash-dot line (`-.`), places circular markers (`o`) at the data points, and colors both line and marker red (`r`). Specify the components (in any order) as a character vector after the data arguments. Note that linespecs are not name-value pairs.

Plotting Data Points with No Line

If you specify a marker, but not a line style, only the markers are plotted. For example:

```
plot(x,y,'d')
```

Line Style Specifiers

You indicate the line styles, marker types, and colors you want to display, detailed in the following tables:

Specifier	LineStyle
`'-'`	Solid line (default)
`'--'`	Dashed line

Specifier	LineStyle
':'	Dotted line
'-.'	Dash-dot line

Marker Specifiers

Specifier	Marker Type
'+'	Plus sign
'o'	Circle
'*'	Asterisk
'.'	Point
'x'	Cross
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'	Six-pointed star (hexagram)

Color Specifiers

Specifier	Color
r	Red
g	Green
b	Blue
c	Cyan
m	Magenta
y	Yellow

Specifier	Color
k	Black
w	White

Related Properties

This page also describes how to specify the properties of lines used for plotting. MATLAB graphics give you control over these visual characteristics:

- `LineWidth` — Specifies the width (in points) of the line.
- `MarkerEdgeColor` — Specifies the color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).
- `MarkerFaceColor` — Specifies the color of the face of filled markers.
- `MarkerSize` — Specifies the size of the marker in points (must be greater than 0).

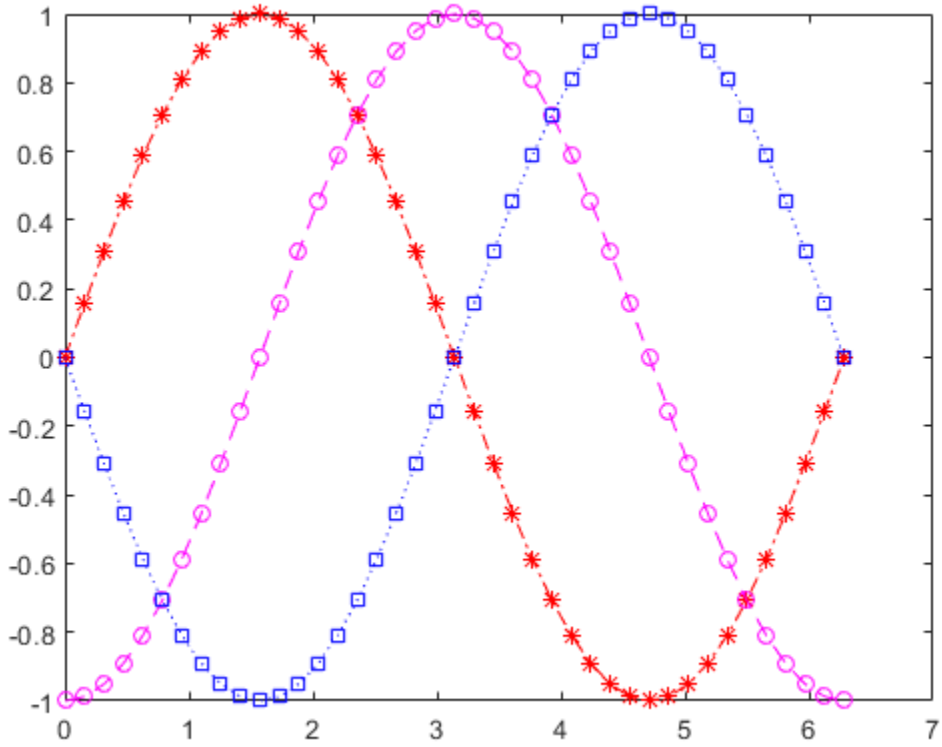
In addition, you can specify the `LineStyle`, `Color`, and `Marker` properties instead of using a line specification character vector. This is useful if you want to specify a color that is not in the list by using RGB triplet values. See Chart Line for details on these properties.

Examples

Modify Line Appearance

Plot the sine function over three different ranges using different line styles, colors, and markers.

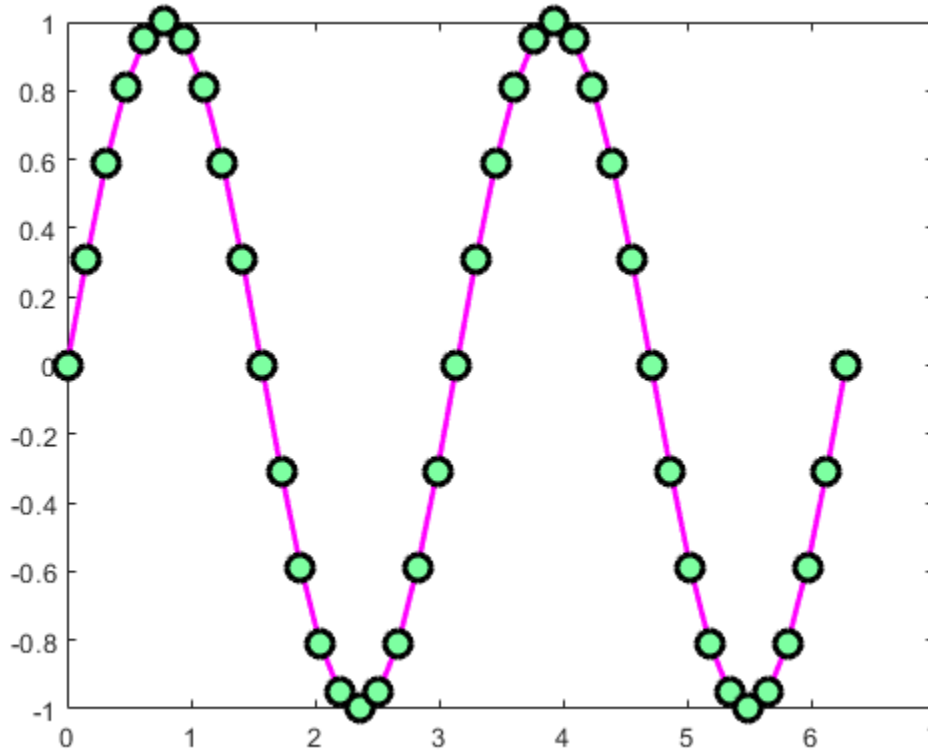
```
figure
t = 0:pi/20:2*pi;
plot(t,sin(t),'-.r*')
hold on
plot(t,sin(t-pi/2),'--mo')
plot(t,sin(t-pi),':bs')
hold off
```



Set Line Properties

Create a plot illustrating how to set line properties.

```
t = 0:pi/20:2*pi;
figure
plot(t,sin(2*t),'-mo',...
    'LineWidth',2,...
    'MarkerEdgeColor','k',...
    'MarkerFaceColor',[.49 1 .63],...
    'MarkerSize',10)
```



See Also

axes | line | patch | plot | set | surface

linkaxes

Synchronize limits of multiple axes

Syntax

```
linkaxes(ax)
linkaxes(ax,dimension)
```

Description

`linkaxes(ax)` synchronizes the limits of the specified vector of axes. Synchronizing limits allows you to zoom or pan in one plot or figure and display the same range of data in another plot or figure. When you first call `linkaxes`, the function chooses new limits that incorporate the current limits of all the specified axes.

`linkaxes(ax,dimension)` synchronizes the axes limits for the specified axis dimension. For example, `linkaxes(ax,'x')` synchronizes the limits for the *x*-axis only.

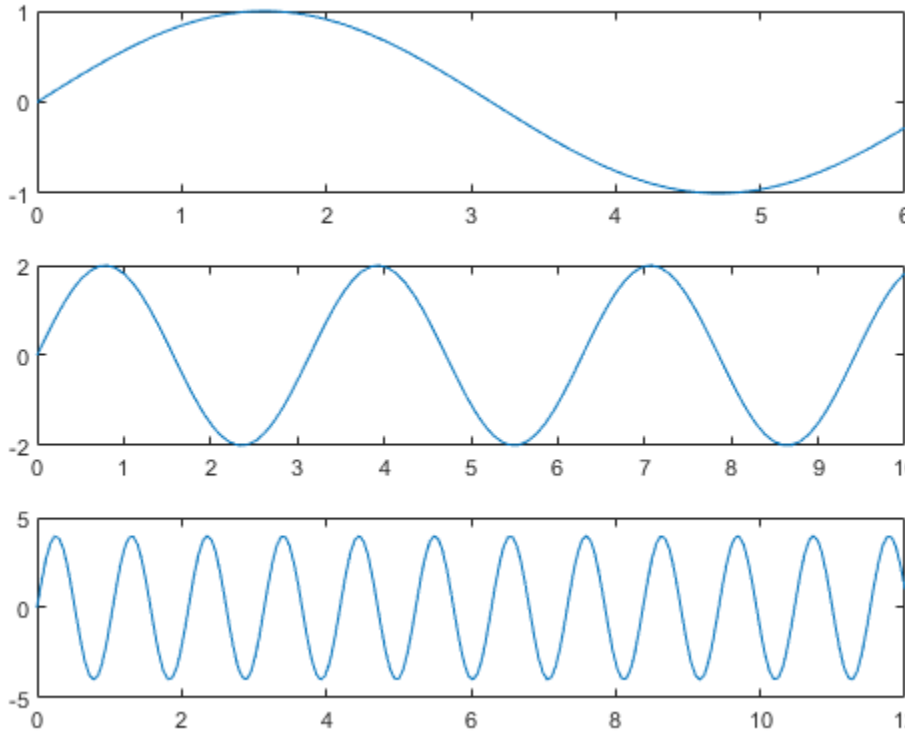
Examples

Synchronize and Change Axis Limits

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 3-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1`, `ax2`, and `ax3`. Then plot into each axes.

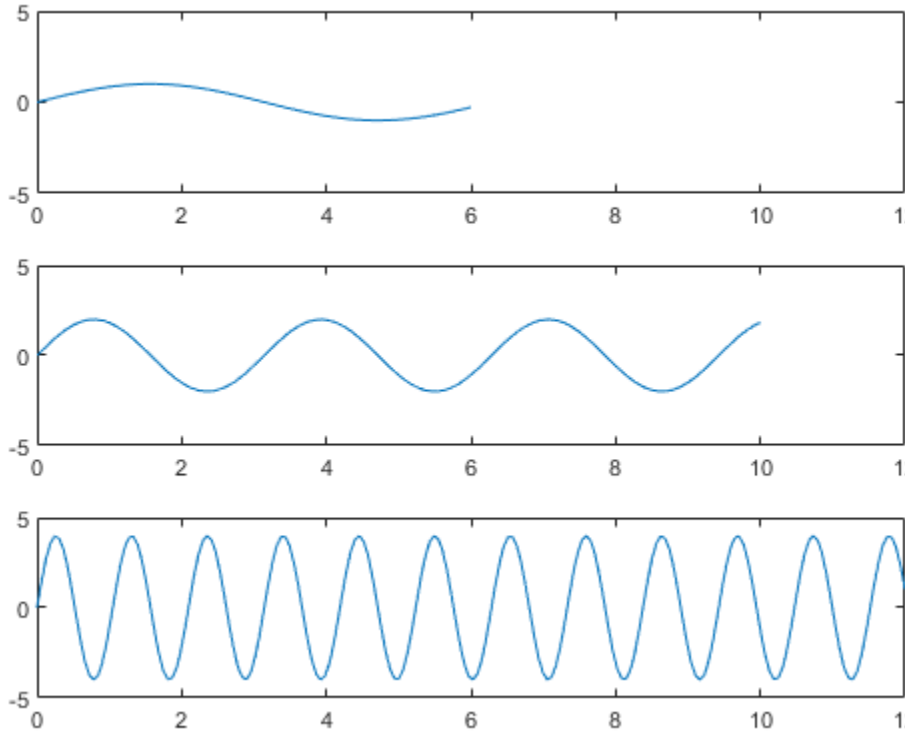
```
tiledlayout(3,1)

% First plot
ax1 = nexttile;
x1 = linspace(0,6);
y1 = sin(x1);
plot(x1,y1)
```

```
% Second plot
ax2 = nexttile;
x2 = linspace(0,10);
y2 = 2*sin(2*x2);
plot(x2,y2)

% Third plot
ax3 = nexttile;
x3 = linspace(0,12,200);
y3 = 4*sin(6*x3);
plot(x3,y3)
```



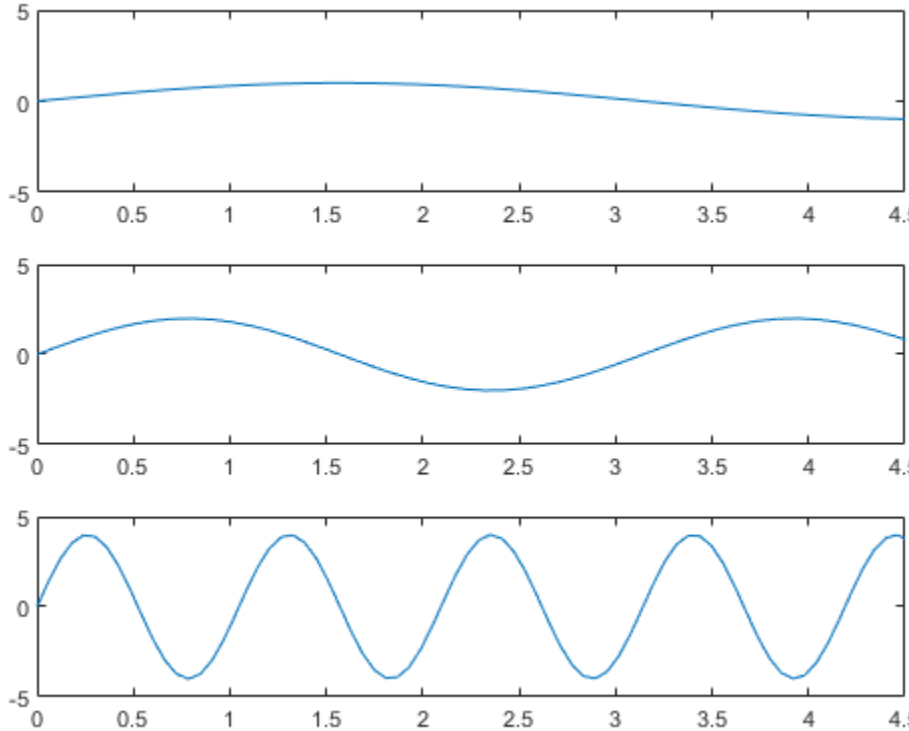
Synchronize the *x*-axis and *y*-axis limits of each plot. Note that the new axes limits incorporate the old limits.

```
linkaxes([ax1 ax2 ax3],'xy')
```



Set the *x*-axis limits for the first plot. All of axes are linked, so the *x*-axis limits in the second and third plots also change.

```
ax1.XLim = [0 4.5];
```



Panning or zooming into one of the plots displays the same range of data in the other two plots.

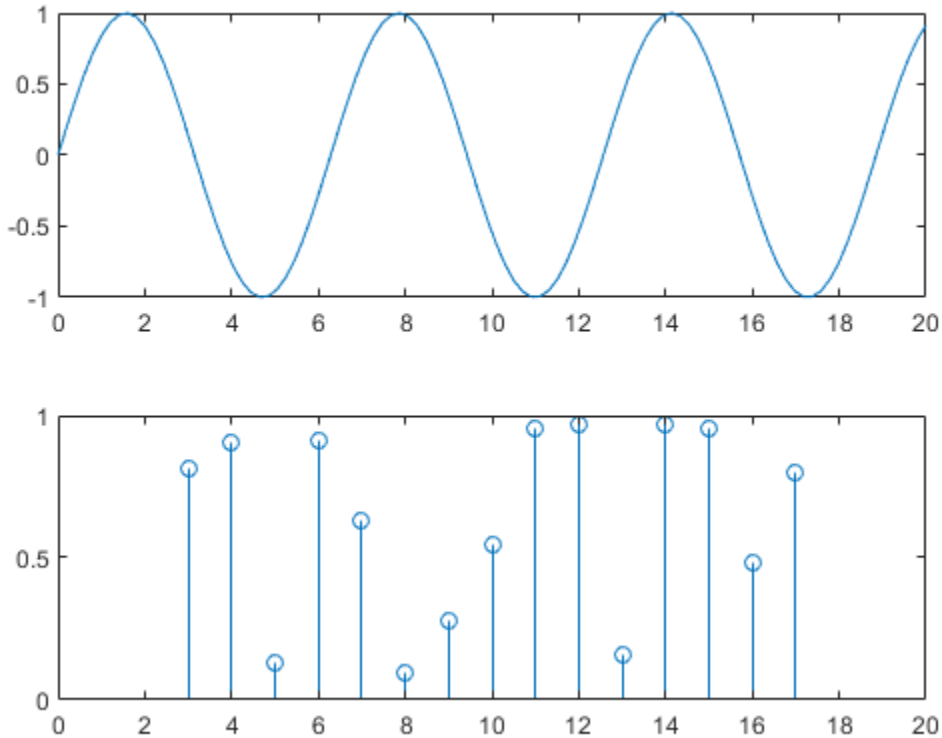
To remove the linking, use `linkaxes([ax1 ax2 ax3],'off')`.

Synchronize *x*-Axis Limits and Display Shared Axis Labels

Create two plots in a 2-by-1 tiled chart layout. Synchronize the *x*-axis limits by calling the `linkaxes` function.

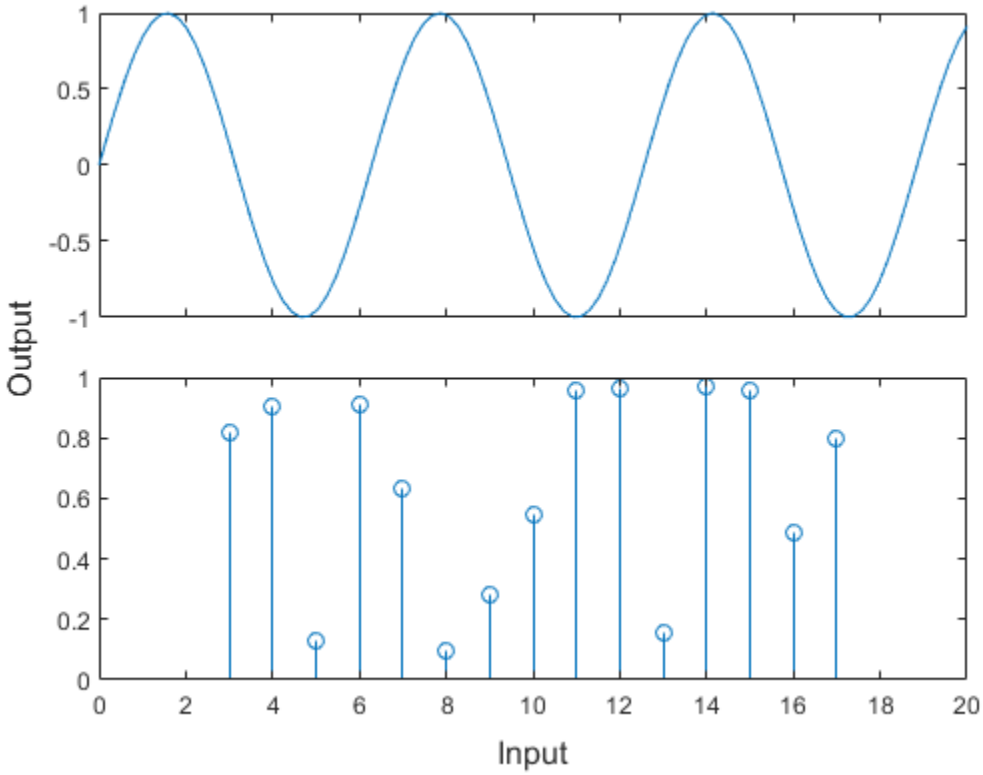
```
x1 = linspace(0,20,100);
y1 = sin(x1);
```

```
x2 = 3:17;
y2 = rand(1,15);

% Create plots.
t = tiledlayout(2,1);
ax1 = nexttile;
plot(ax1,x1,y1)
ax2 = nexttile;
stem(ax2,x2,y2)

% Link the axes. Add title and labels.
linkaxes([ax1,ax2],'x');
```



Create shared *x*- and *y*-axis labels by passing `t` to the `xlabel` and `ylabel` functions. Remove the *x*-axis tick labels from the top plot by calling the `xticklabels` function. Then minimize the space between the plots by setting the `TileSpacing` property of `t` to `'compact'`.

```
xlabel(t,'Input')
ylabel(t,'Output')
xticklabels(ax1,{})
t.TileSpacing = 'compact';
```



Input Arguments

ax — Target axes
vector of `Axes` objects

Target axes, specified as a vector of `Axes` objects. The `linkaxes` function supports 2-D Cartesian axes only.

You can link any number of `Axes` objects. For example, `linkaxes([ax1 ax2 ax3])` links `ax1`, `ax2`, and `ax3`. Separate calls to `linkaxes([ax1 ax2])` and `linkaxes([ax2 ax3])` cancels the link between `ax1` and `ax2`.

`dimension` — Limits to synchronize
`'xy'` (default) | `'x'` | `'y'` | `'off`

Limits to synchronize, specified as one of these values:

- `'xy'` — Synchronize both the *x*-axis and *y*-axis limits.
- `'x'` — Synchronize only the *x*-axis limits.
- `'y'` — Synchronize only the *y*-axis limits.
- `'off'` — Turn off synchronization.

See Also

`linkdata` | `linkprop` | `nexttile` | `tiledlayout`

Introduced before R2006a

linkdata

Automatically update charted data

Description

Use data linking to synchronize charts and their workspace variables. Linked charts automatically update with changes to workspace variables, and workspace variables automatically update with chart modifications made via the brush tool.

Creation

Syntax

```
linkdata on
linkdata off
linkdata
linkdata(fig, ___ )

l = linkdata
l = linkdata(fig)
```

Description

`linkdata on` turns on data linking mode for the current axes.

`linkdata off` turns off data linking mode.

`linkdata` toggles the data linking mode between `'on'` and `'off'`.

`linkdata(fig, ___ )` specifies the data linking mode for the specified figure. Use single or double quotation marks around the options `'on'` and `'off'`. For example, `linkdata(fig,'on')`.

`l = linkdata` creates a `LinkData` object. This syntax is useful for querying the data linking mode.

`l = linkdata(fig)` creates a `LinkData` object for the specified figure.

Input Arguments

`fig` — Target figure
`Figure` object

Target figure, specified as a `Figure` object. If you do not specify the figure, then data linking is enabled or disabled for the current figure.

Properties

`Enable` — Data linking mode
`'off'` (default) | `'on'`

This property is read-only.

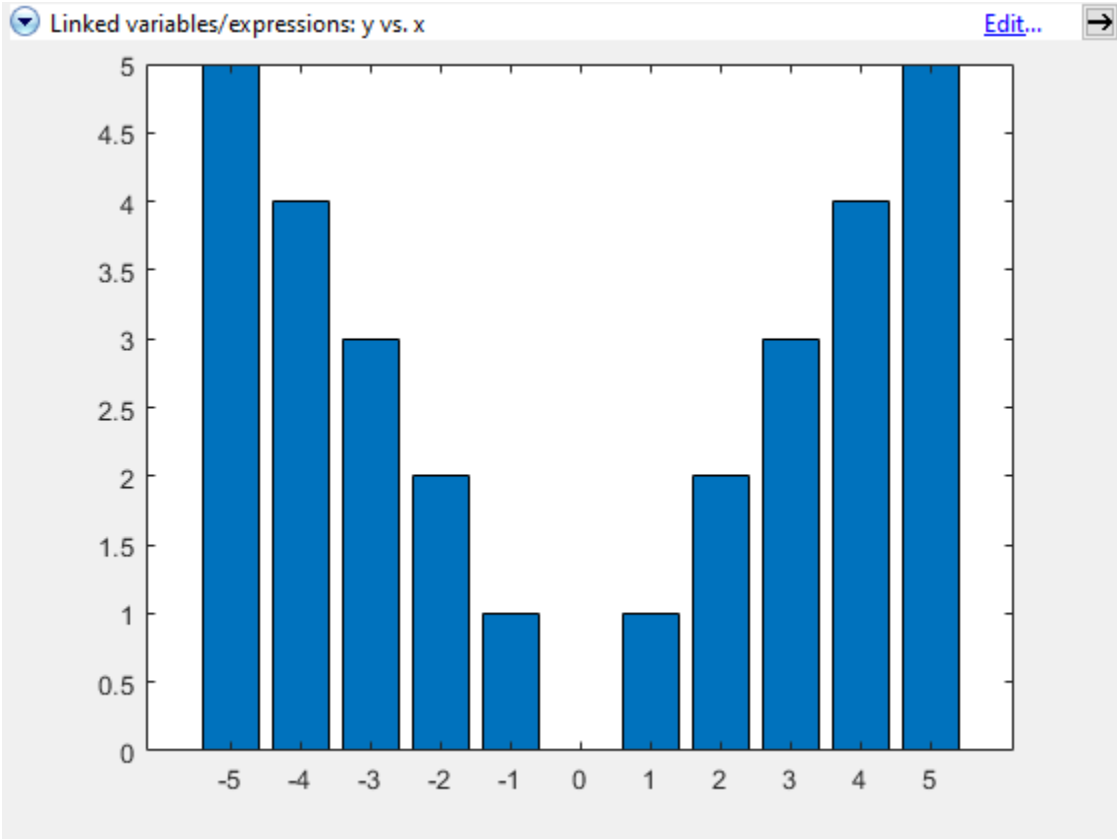
Data linking mode, specified as `'off'` or `'on'`.

Examples

Update Chart When Workspace Variables Change

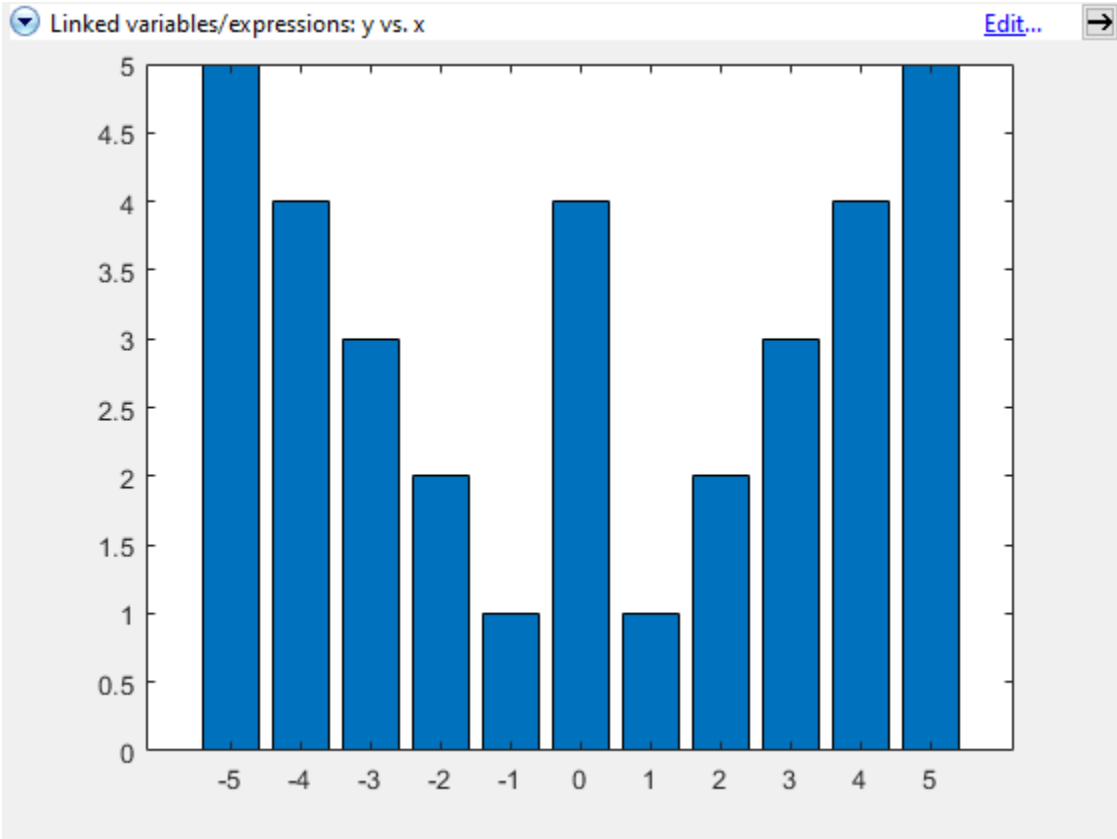
Create a bar chart and enable data linking. Data linking synchronizes the bar chart with the `x` and `y` variables in the workspace.

```
x = linspace(-5,5,11);
y = abs(x);
bar(x,y)
linkdata on
```



Modify one of the values in `y`. Note that the bar chart immediately updates to reflect the change.

```
y(6) = 4;
```



Update Workspace Data When Chart Changes

Interactively delete workspace values by selecting them in a chart, using `linkdata` and the brush tool.

Create some data to plot. Note that `x` and `y` contain 21 values.

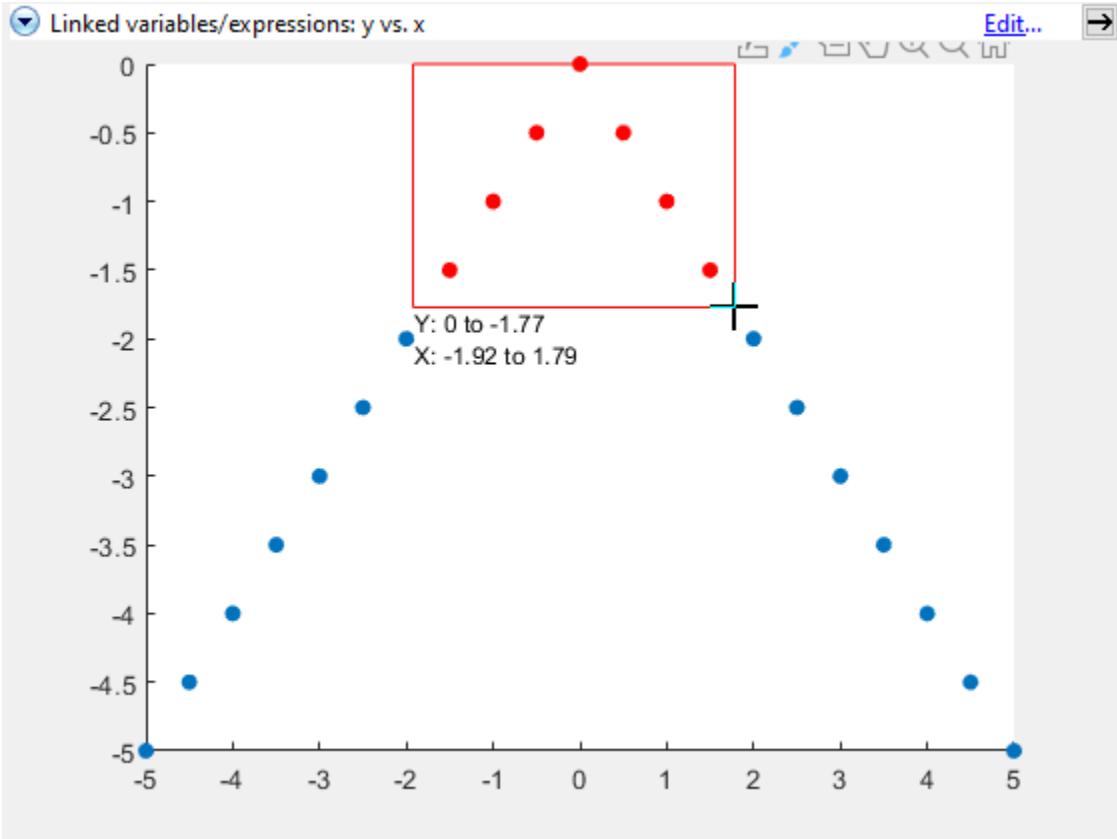
```
x = linspace(-5,5,21);
y = -abs(x);

lengthX = length(x)
lengthY = length(y)
```

```
lengthX =

    21


lengthY =

    21
```

Plot the data, then enable both data linking and brushing.

```
scatter(x,y,'filled')
linkdata on
brush on
```

Highlight the data values you want to delete. To remove the brushed data from the chart, right-click on a brushed value and select **Remove** from the context menu.



The chart is linked to its data sources, so deleting values from the chart also deletes them from the data sources. Note that x and y now contain only 14 data values.

```
lengthX = length(x)
lengthY = length(y)
```

```
lengthX =

    14


lengthY =
```

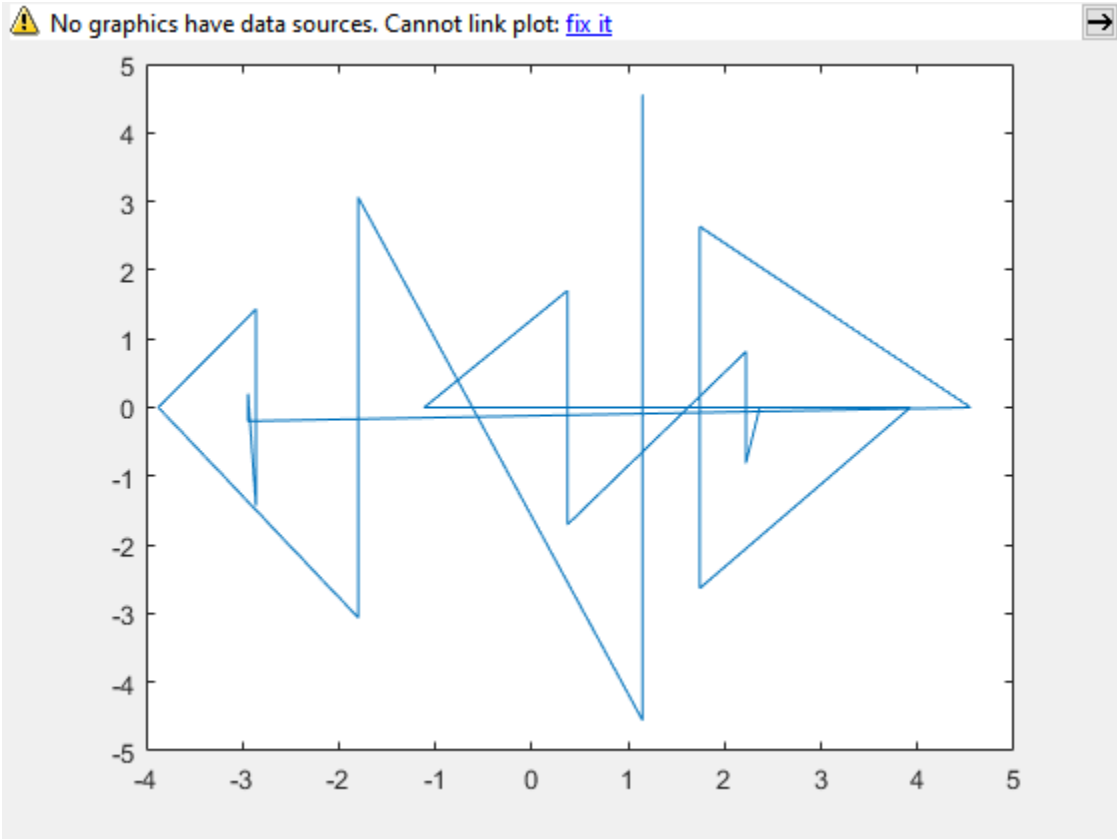
14

Link Complex Data

Use data linking with complex numbers. When you plot complex data, the real and imaginary parts of the data do not correspond to different workspace variables. As a result, the `linkdata` function cannot identify data sources for the *x*- and *y*-axes. To link complex data, you must manually specify the real part of the complex data as the *x*-component and the imaginary part as the *y*-component.

First, create a matrix of complex data values and plot them. Then, enable data linking.

```
z = eig(randn(20,20));
plot(z)
linkdata on
```



The information bar displays a warning because `linkdata` cannot identify the real and imaginary parts of the charted complex data. To manually specify the real and imaginary parts of the data, first click the **fix it** link. Then type `real(z)` and `imag(z)` in the dialog boxes under `XDataSource` and `YDataSource`.

Now the chart is linked to its data sources. If you change `z`, then the plot will update to reflect the new data.

To avoid using the dialog box, you can specify the data sources when first plotting the data values.

```
plot(z,'XDataSource','real(z)','YDataSource','imag(z)')
linkdata on
```

Tips

- Figures with linked charts have an information bar at the top of the window. If `linkdata` cannot unambiguously identify data sources for a chart, then the information bar will prompt you to manually identify data sources using the **fix it** link.
- The `linkdata` function updates linked data sources and charts around twice a second. To smoothly animate changes in data values, create loops that execute two times per second or less. For more information, see the `pause` function.
- Once a chart is synchronized with its workspace variables, programmatic changes to the data properties of the chart (for example, `XData`) do not update the workspace variables. Workspace variables only update when you modify plotted data interactively, using the brush tool.

Algorithms

Data linking connects a chart and its workspace variables using the chart's data source properties (for example, `XDataSource`). When you turn on data linking for a figure, MATLAB compares variables in the workspace to plotted data in the figure. When it finds a match, MATLAB assigns that variable to the appropriate data source property of the chart.

See Also

Functions

`brush` | `linkaxes` | `linkdata` | `linkprop` | `refreshdata`

Topics

“Automatically Refresh Plot After Changing Data”

Introduced in R2008a

linkprop

Keep same value for corresponding properties of graphics objects

Syntax

```
hlink = linkprop(obj_handles,'PropertyName')
```

```
hlink = linkprop(obj_handles,{'PropertyName1','PropertyName2',...})
```

Description

Use `linkprop` to maintain the same values for the corresponding properties of different graphics objects.

Note Use `linkprop` only with graphics objects.

`hlink = linkprop(obj_handles,'`*`PropertyName`*`')` maintains the same value for the property *`PropertyName`* on all objects whose handles appear in `obj_handles`. `linkprop` returns the link object in `hlink`. See "About Link Objects" on page 1-8669 for more information.

`hlink = linkprop(obj_handles,{'`*`PropertyName1`*`','`*`PropertyName2`*`',...})` maintains the same respective values for all properties passed as a cell array on all objects whose handles appear in `obj_handles`.

MATLAB updates the linked properties of all linked objects immediately when `linkprop` is called. The first object in the list `obj_handles` determines the property values for the other objects.

A set of graphics objects can have only one link object connecting their properties at any given time. Calling `linkprop` creates a new link object. This new link object replaces any existing link object that is associated with the objects specified in `obj_handles`. However, you can manage which properties and which objects are linked by calling methods on that object:

- To add an object to the list of linked objects, use the `addtarget` method.
- To link new properties of currently-linked objects, use the `addprop` method.
- To stop linking an object, use the `removetarget` method.
- To stop properties from linking, use the `removeprop` method.

About Link Objects

The link object that `linkprop` returns stores the mechanism that links the properties of different graphics objects. Therefore, the link object must exist within the context where you want property linking to occur (such as in the base workspace if users are to interact with the objects from the command line or figure tools).

The following list describes ways to maintain a reference to the link object.

- Return the link object as an output argument from a function and keep it in the base workspace while interacting with the linked objects.
- Make the `hlink` variable global.
- Store the `hlink` variable in an object's `UserData` property or in application data.

Updating a Link Object

If you want to change either the graphics objects or the properties that are linked, you need to use the link object methods designed for that purpose. These methods are functions that operate only on link objects. To use them, you must first create a link object using `linkprop`.

Method	Purpose
`addtarget`	Add specified graphics object to the link object's targets.
`removetarget`	Remove specified graphics object from the link object's targets.
`addprop`	Add specified property to the linked properties.
`removeprop`	Remove specified property from the linked properties.

Method Syntax

```
addtarget(hlink,obj_handles)
removetarget(hlink,obj_handles)
addprop(hlink,'PropertyName')
removeprop(hlink,'PropertyName')
```

Method Arguments

- `hlink` — Link object returned by `linkprop`
- `obj_handles` — One or more graphic object handles
- *`PropertyName`* — Name of a property common to all target objects

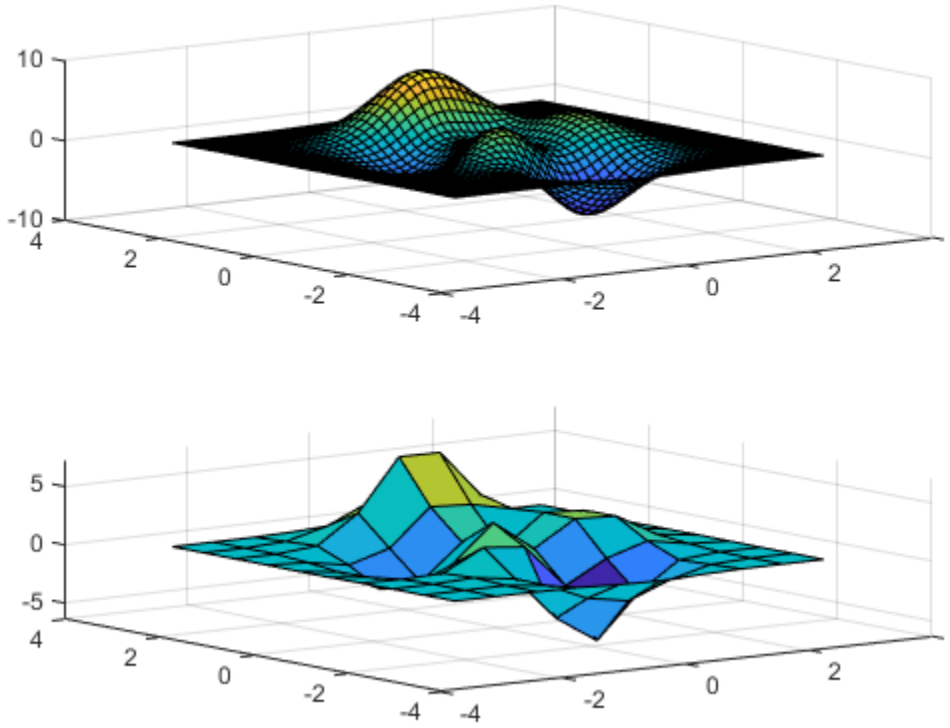
Examples

Link Axes So They Rotate Simultaneously

Link properties of two axes so that rotating one axes automatically rotates the other.

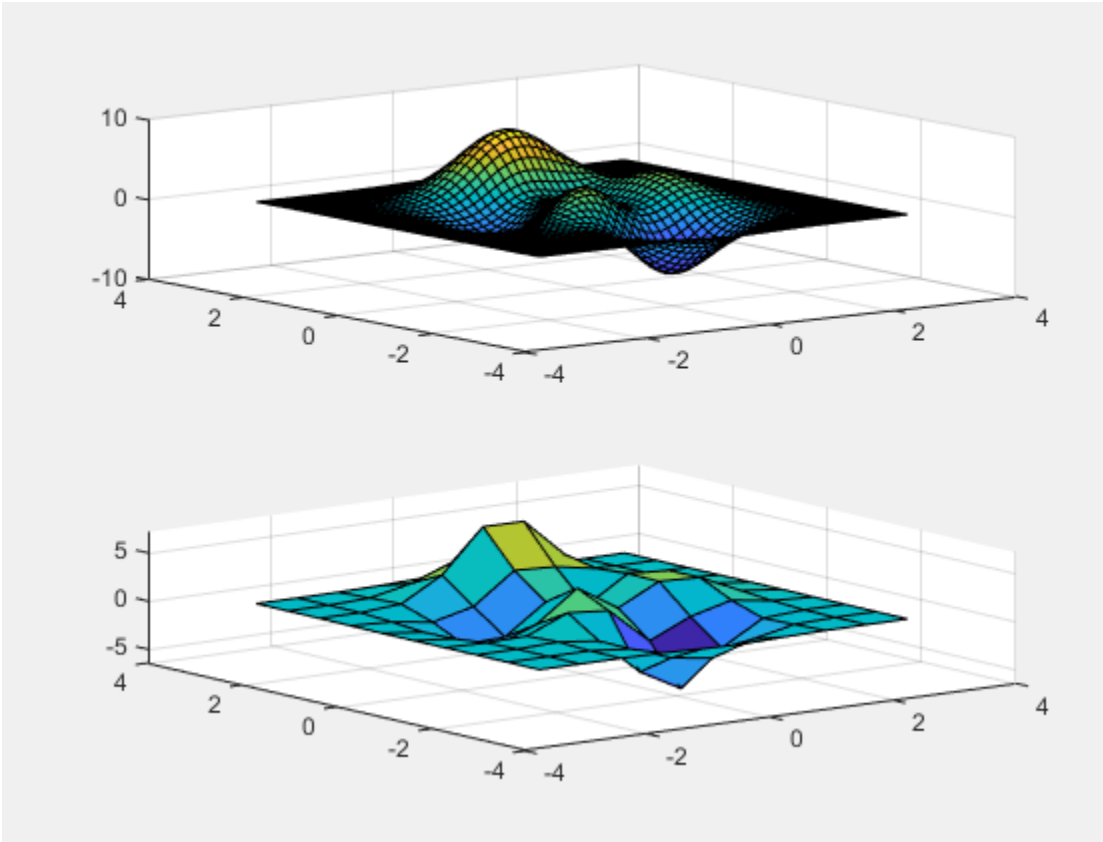
Create a figure with two axes and store the axes handles. Add plots to both axes.

```
figure
ax1 = subplot(2,1,1);
[X1,Y1,Z1] = peaks;
surf(X1,Y1,Z1)

ax2 = subplot(2,1,2);
[X2,Y2,Z2] = peaks(10);
surf(X2,Y2,Z2)
```



Link the `CameraPosition` and `CameraUpVector` properties of the axes and return the link object handle. Then, enable interactive rotation and use the mouse to rotate either axes. Rotating one axes automatically rotates the other in the same manner.

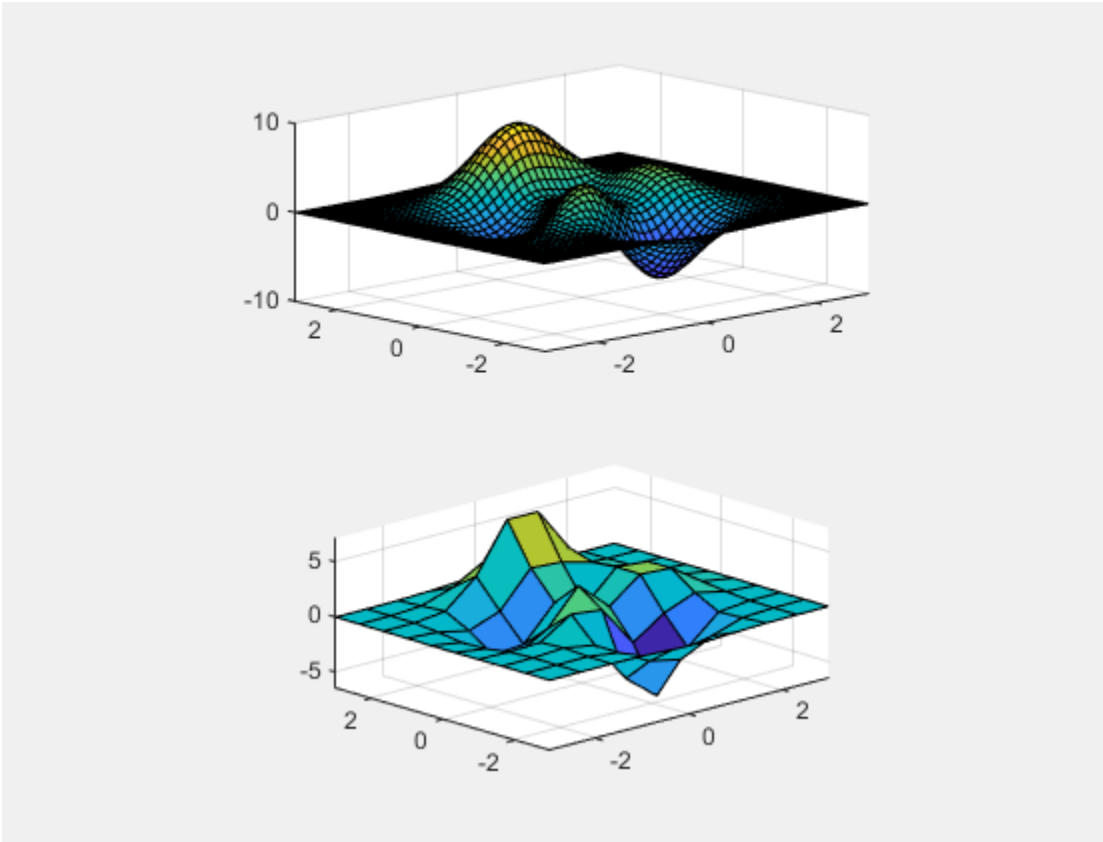
```
hlink = linkprop([ax1,ax2],{'CameraPosition','CameraUpVector'});
rotate3d on
```



To disable interactive rotation, use `rotate3d off`.

Link an additional property by passing the link object handle and the property name to `addprop`.

```
addprop(hlink,'PlotBoxAspectRatio')
```



See Also

`getappdata` | `ishghandle` | `linkaxes` | `linkdata` | `setappdata`

Introduced before R2006a

linsolve

Solve linear system of equations

Syntax

```
X = linsolve(A,B)
X = linsolve(A,B,opts)
[X,r] = linsolve( ___ )
```

Description

`X = linsolve(A,B)` solves the linear system $AX = B$ using one of these methods:

- When `A` is *square*, `linsolve` uses LU factorization with partial pivoting.
- For all other cases, `linsolve` uses QR factorization with column pivoting.

`linsolve` warns if `A` is ill conditioned (for square matrices) or rank deficient (for rectangular matrices).

`X = linsolve(A,B,opts)` uses an appropriate solver as determined by the options structure `opts`. The fields in `opts` are logical values describing properties of the matrix `A`. For example, if `A` is an upper triangular matrix, you can set `opts.UT = true` to make `linsolve` use a solver designed for upper triangular matrices. `linsolve` does not test to verify that `A` has the properties specified in `opts`.

`[X,r] = linsolve( ___ )` also returns `r`, which is the reciprocal of the condition number of `A` (for square matrices) or the rank of `A` (for rectangular matrices). You can use any of the input argument combinations in previous syntaxes. With this syntax, `linsolve` does not warn if `A` is ill conditioned or rank deficient.

Examples

Solve Linear System

Solve a linear system with both `mldivide` and `linsolve` to compare performance.

`mldivide` is the recommended way to solve most linear systems of equations in MATLAB ®. However, the function performs several checks on the input matrix to determine whether it has any special properties. If you know about the properties of the coefficient matrix ahead of time, then you can use `linsolve` to avoid time-consuming checks for large matrices.

Create a 10000-by-10000 magic square matrix and extract the lower triangular portion. Set the `LT` field of the `opts` structure to `true` to indicate that `A` is a lower triangular matrix.

```
A = tril(magic(1e4));
opts.LT = true;
```

Create a vector of ones for the right-hand side of the linear equation Ax = *b*. The number of rows in `A` and `b` must be equal.

```
b = ones(size(A,2),1);
```

Solve the linear system Ax = *b* using `mldivide` and time the calculation.

```
tic
x1 = A\b;
t1 = toc
```

```
t1 = 0.2250
```

Now, solve the system again using `linsolve`. Specify the options structure so that `linsolve` can select an appropriate solver for a lower triangular matrix.

```
tic
x2 = linsolve(A,b,opts);
t2 = toc
```

```
t2 = 0.0658
```

Compare the execution times to see how much faster `linsolve` is. As with any timing comparison, the results can vary between different computers and releases of MATLAB.

```
speedup = t1/t2
```

```
speedup = 3.4189
```

Suppress Matrix Condition Warnings

Solve a linear system using `linsolve` with two outputs to suppress matrix conditioning warnings.

Create a 20-by-20 Hilbert test matrix. This matrix is nearly singular, with the largest singular value being about `2e18` larger than the smallest.

```
A = hilb(20);
```

Solve a linear system involving `A` with `linsolve`. Since `A` is nearly singular, `linsolve` returns a warning.

```
b = ones(20,1);
x = linsolve(A,b);
```

```
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND
```

Now, solve the same linear system, but specify two outputs to `linsolve`. MATLAB® suppresses the warning, and the second output `r` contains the reciprocal condition number of `A`. You can use this syntax to handle ill-conditioned matrices with special cases in your code, without the code producing a warning.

```
[x,r] = linsolve(A,b)
```

```
x = 20×1
10^9 ×

    0.0000
   -0.0000
    0.0001
   -0.0014
    0.0126
   -0.0613
    0.1555
   -0.1083
   -0.4672
    1.3358
       ⋮


r = 1.3514e-19
```

Input Arguments

A — Coefficient matrix
matrix

Coefficient matrix. A appears in the system of linear equations on the left as $AX = B$. The number of rows in A must equal the number of rows in B.

A cannot be sparse. To solve a linear system involving a sparse matrix, use `mldivide` or `decomposition` instead.

Data Types: `single` | `double`
Complex Number Support: Yes

B — Input array
vector | matrix

Input array, specified as a vector or matrix. B appears in the system of linear equations on the right as $AX = B$. If B is a matrix, then each column in the matrix represents a different vector for the right-hand side.

The number of rows in A must equal the number of rows in B.

Data Types: `single` | `double`
Complex Number Support: Yes

opts — Coefficient matrix properties
structure

Coefficient matrix properties, specified as a structure. Use this structure to specify properties of A that `linsolve` uses to select an appropriate solver for the linear system. The fields in the structure contain `true`/`false` values to indicate whether A has each property. By default all fields in the structure are assumed to be `false`. This table lists the possible fields in `opts` and their corresponding matrix properties.

Field	Matrix Property
LT	Lower triangular (nonzero values appearing only on or below the main diagonal)

Field	Matrix Property
UT	Upper triangular (nonzero values appearing only on or above the main diagonal)
UHESS	Upper Hessenberg (all zero values below the first subdiagonal)
SYM	Real symmetric or complex Hermitian (matrix equal to its transpose)
POSDEF	Positive definite (all positive eigenvalues)
RECT	Rectangular matrix (different number of rows and columns)
TRANSA	Conjugate transpose — Specifies whether the function solves `A*X = B` (when `opts.TRANSA = false`) or the transposed problem `A'*X = B` (when `opts.TRANSA = true`)

Example: `opts.UT = true` specifies that `A` is upper triangular.

Example: `opts.SYM = true, opts.POSDEF = true` sets two fields to specify that `A` is symmetric and positive definite.

Valid Combinations

The rows of this table list all combinations of field values in `opts` that are valid for `linsolve`. Empty cells are the default value of `false`, and a `true/false` entry indicates that `linsolve` accepts either value.

	LT	UT	UHESS	SYM	POSDEF	RECT	TRANSA
A is lower triangular	true					true/false	true/false
A is upper triangular		true				true/false	true/false
A is upper Hessenberg			true				true/false
A is symmetric				true	true/false		true/false

	LT	UT	UHESS	SYM	POSDEF	RECT	TRANSA
A is rectangular						true/false	true/false

Notes on Usage

- If A has the properties in opts, then linsolve is faster compared to mldivide, because linsolve invokes the appropriate solver immediately and does not perform any tests to verify that A has the specified properties.
- If A does *not* have the properties that you specify in opts, then linsolve returns incorrect results and does not return an error message. Therefore, if you are unsure whether A has the specified properties, use mldivide or decomposition instead.

Data Types: struct

Output Arguments

X — Linear system solution
vector | matrix

Linear system solution, returned as a vector or matrix that satisfies $AX = B$ (or $A^{T}X = B$ if opts.TRANSA = true). The size of X depends on whether opts.TRANSA = true:

- If A is m-by-n and B is m-by-k, then X is n-by-k and is the solution to $AX = B$.
- If opts.TRANSA = true, then A is m-by-n and B is n-by-k. In this case, X is m-by-k and is the solution to $A^{T}X = B$.

r — Reciprocal condition number or rank
scalar

Reciprocal condition number or rank, returned as a scalar.

- If A is a square matrix, then r is the reciprocal condition number of A.
- If A is a rectangular matrix, then r is the rank of A.
- If opts is specified, then r is the reciprocal of the condition number of A unless RECT is true and both LT and UT are false, in which case, r gives the rank of A.

Tips

- The speed benefit of `linsolve` can vary depending on the matrix structure and the relative optimization of the underlying algorithms. In some cases (such as with small matrices) there might not be any speed-up compared to `mldivide`. The speed benefit with `linsolve` arises by avoiding costly checks on the properties of large matrices, or by choosing an algorithm that is better suited to the input than the choice that `mldivide` makes.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The `opts` structure must be a constant scalar. Code generation does not support arrays of options structures.
- Code generation only optimizes these cases:
  - `UT`
  - `LT`
  - `UHESS = true` (the `TRANSA` can be either `true` or `false`)
  - `SYM = true` and `POSDEF = true`

  Other options are equivalent to using `mldivide`.
- Code generation does not support sparse matrix inputs for this function.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- The `opts` structure must be a constant scalar. Code generation does not support arrays of options structures.

- Code generation only optimizes these cases:
  - `UT`
  - `LT`
  - `UHESS = true` (the `TRANSA` can be either `true` or `false`)
  - `SYM = true` and `POSDEF = true`

  Other options are equivalent to using `mldivide`.
- Code generation does not support sparse matrix inputs for this function.

See Also

`decomposition` | `lsqminnorm` | `mldivide`

Topics

"Systems of Linear Equations"

Introduced before R2006a

linspace

Generate linearly spaced vector

Syntax

```
y = linspace(x1,x2)
y = linspace(x1,x2,n)
```

Description

`y = linspace(x1,x2)` returns a row vector of 100 evenly spaced points between `x1` and `x2`.

`y = linspace(x1,x2,n)` generates `n` points. The spacing between the points is `(x2-x1)/(n-1)`.

`linspace` is similar to the colon operator, "`:`", but gives direct control over the number of points and always includes the endpoints. "`lin`" in the name "`linspace`" refers to generating linearly spaced values as opposed to the sibling function `logspace`, which generates logarithmically spaced values.

Examples

Vector of Evenly Spaced Numbers

Create a vector of 100 evenly spaced points in the interval `[-5,5]`.

```
y = linspace(-5,5);
```

Vector with Specified Number of Values

Create a vector of 7 evenly spaced points in the interval `[-5,5]`.

```
y1 = linspace(-5,5,7)
```

```
y1 = 1×7

   -5.0000   -3.3333   -1.6667         0    1.6667    3.3333    5.0000
```

Vector of Evenly Spaced Complex Numbers

Create a vector of complex numbers with 8 evenly spaced points between `1+2i` and `10+10i`.

```
y = linspace(1+2i,10+10i,8)
```

```
y = 1×8 complex

   1.0000 + 2.0000i   2.2857 + 3.1429i   3.5714 + 4.2857i   4.8571 + 5.4286i   6.1429 -
```

Input Arguments

`x1,x2` — Point interval
pair of numeric scalars

Point interval, specified as a pair of numeric scalars. `x1` and `x2` define the interval over which `linspace` generates points. `x1` and `x2` can be real or complex, and `x2` can be either larger or smaller than `x1`. If `x2` is smaller than `x1`, then the vector contains descending values.

Data Types: `single` | `double` | `datetime` | `duration`
Complex Number Support: Yes

`n` — Number of points
100 (default) | real numeric scalar

Number of points, specified as a real numeric scalar.

- If `n` is `1`, `linspace` returns `x2`.

- If `n` is zero or negative, `linspace` returns an empty 1-by-0 matrix.
- If `n` is not an integer, `linspace` rounds down and returns `floor(n)` points.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- Use `distributed.linspace` to call the distributed version of `linspace`.
- `x1` and `x2` must be `single` or `double` scalars.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`colon` | `logspace`

Introduced before R2006a

RandStream.list

Random number generator algorithms

Class

`RandStream`

Syntax

`RandStream.list`

Description

`RandStream.list` lists all the generator algorithms that can be used when creating a random number stream with `RandStream` or `RandStream.create`. The available generator algorithms and their properties are given in the following table.

Keyword	Generator	Multiple Stream and Substream Support	Approximate Period In Full Precision
mt19937ar	Mersenne twister (used by default stream at MATLAB startup)	No	2^{19937}-1
dsfmt19937	SIMD-oriented fast Mersenne twister	No	2^{19937}-1
mcg16807	Multiplicative congruential generator	No	2^{31}-2
mlfg6331_64	Multiplicative lagged Fibonacci generator	Yes	2^{124} (2^{51} streams of length 2^{72})

Keyword	Generator	Multiple Stream and Substream Support	Approximate Period In Full Precision
mrg32k3a	Combined multiple recursive generator	Yes	2^{191} (2^{63} streams of length 2^{127})
philox4x32_10	Philox 4x32 generator with 10 rounds	Yes	2^{193} (2^{64} streams of length 2^{129})
threefry4x64_20	Threefry 4x64 generator with 20 rounds	Yes	2^{514} (2^{256} streams of length 2^{258})
shr3cong	Shift-register generator summed with linear congruential generator	No	2^{64}
swb2712	Modified subtract with borrow generator	No	2^{1492}

See “Choosing a Random Number Generator” for details about these generator algorithms. See http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html for a full description of the Mersenne twister algorithm.

See Also

Topics

“Creating and Controlling a Random Number Stream”

listdlg

Create list selection dialog box

Syntax

```
[indx,tf] = listdlg('ListString',list)
[indx,tf] = listdlg('ListString',list,Name,Value)
```

Description

`[indx,tf] = listdlg('ListString',list)` creates a modal on page 1-16280 dialog box that allows the user to select one or more items from the specified list.

The `list` value is the list of items to present in the dialog box.

The function returns two output arguments, `indx` and `tf` containing information about which items the user selected.

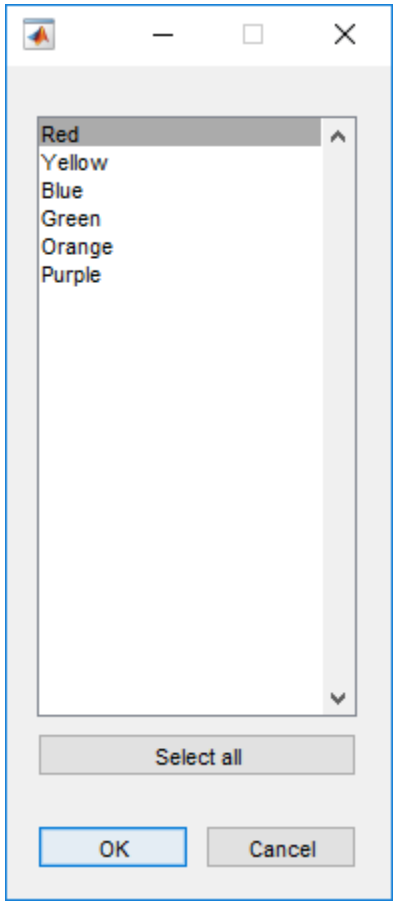
The dialog box includes **Select all**, **Cancel**, and **OK** buttons. You can limit selection to a single item by using the name-value pair, `'SelectionMode','single'`.

`[indx,tf] = listdlg('ListString',list,Name,Value)` specifies additional options using one or more name-value pair arguments. For example, `'PromptString','Select a Color'` presents `Select a Color` above the list.

Examples

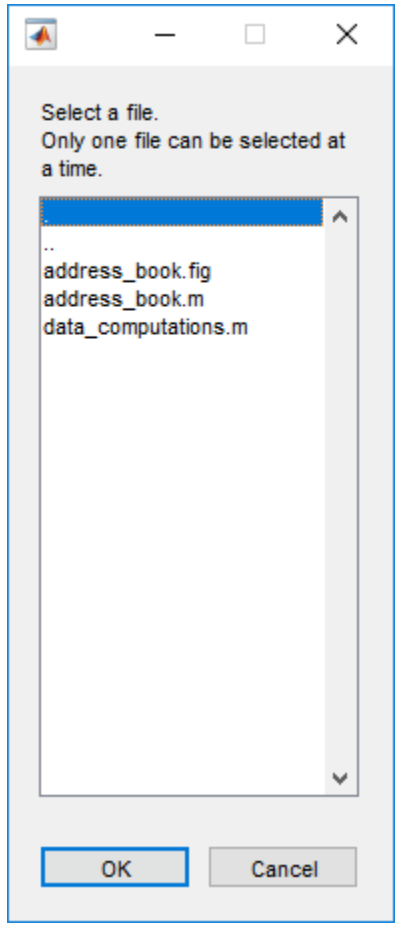
Present List of Colors for Multiple Selection

```
list = {'Red','Yellow','Blue',...
'Green','Orange','Purple'};
[indx,tf] = listdlg('ListString',list);
```



Present Current Folder Files for Single Selection

```
d = dir;
fn = {d.name};
[indx,tf] = listdlg('PromptString',{'Select a file.',...
    'Only one file can be selected at a time.',''},...
    'SelectionMode','single','ListString',fn);
```



Input Arguments

`list` — list of items

character vector | cell array of character vectors | string array

List of items to present in the dialog box, specified as a character vector, cell array of character vectors, or string array. For cell arrays and string arrays, each element typically corresponds to a separate list item. If you insert newline characters using `sprintf`, it

results in more list items. For example, the following code results in four list items, even though there are only three cell array elements.

```
f = listdlg('ListString', ...
            {'John Smith' ...
             sprintf('Cecelia\nPayne-Gaposchkin') ...
             'Gina Peters'});
```

Example: `{'Ellen','Varun','Haruko','Roger'}`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'SelectionMode','single','InitialValue',4` specifies that the user can select one item from the list and that when the dialog box opens, the fourth item in the list is selected.

`PromptString` — List box prompt

[ ] (default) | character vector | cell array of character vectors | string array

List box prompt, specified as a character vector, cell array of character vectors, or string array. The prompt appears above the list box.

If you specify the prompt as a character vector that is longer than the width of the dialog box, the prompt clips. To create a multiline list box prompt, specify the prompt as a cell array or string array. Line breaks occur between each array element. Long elements wrap to fit the dialog box.

Example: `'PromptString','Select a catalog number:'`

`SelectionMode` — List selection mode

`'multiple'` (default) | `'single'`

List selection mode specified as the comma-separated pair consisting of `'SelectionMode'` and either `'multiple'` or `'single'`.

- If the selection mode is set to `'multiple'`, then users can select multiple list items and the **Select all** button displays in the dialog box.

- If the selection mode is set to `'single'`, then users can select one list item only and the **Select all** button does *not* display in the dialog box.

Example: `'SelectionMode','single'`

`ListSize` — list box size
[160 300] (default) | [width height]

List box size in pixels, specified as the comma-separated pair consisting of `'ListSize'` and a two-element vector, `[width height]`.

Example: `'ListSize',[150,250]`

`InitialValue` — Selected list box items
1 (default) | vector of indices

Selected list box items, specified as a scalar index value when `'SelectionMode'` is set to `'single'` and specified as a vector of indices when `'SelectionMode'` is set to `'multiple'`. The indices indicate which rows in the list box are selected when the dialog box opens. For example:

- If `'InitialValue'` is set to `3`, then the third item from the top of the list is selected when the dialog box opens.
- If `'InitialValue'` is set to `[3 4]`, then the third and fourth items from the top of the list are selected when the dialog box opens.

Example: `'InitialValue',5`

Example: `'InitialValue',[2 5]`

`Name` — Dialog box title
[ ] (default) | character vector | string scalar

Dialog box title, specified as a character vector or string scalar.

Example: `'Name','File Selection'`

`OKString` — OK button label
'OK' (default) | character vector | string scalar

OK button label, specified as a character vector or string scalar.

Example: `'OKString','Apply'`

`CancelString` — Cancel button label
'Cancel' (default) | character vector | string scalar

Cancel button label, specified as a character vector or string scalar.

Example: `'CancelString','No Selection'`

Output Arguments

`indx` — Index to selected rows
array of indices

Index to selected rows, returned as an array of indices. The row indices correspond to selections the user made from the list. If the user clicks **Cancel**, presses **Esc**, or clicks the close button in the dialog box title bar, then the `indx` value is returned as an empty array.

`tf` — Selection logical
1 | 0

Selection logical returned as `1` or `0`.

The selection logical value indicates whether the user made a selection. If the user clicks **OK**, double-clicks a list item, or presses **Return**, then the `tf` return value is 1.

If the user clicks **Cancel**, presses **Esc**, or clicks the close button (X) in the dialog box title bar, then the `tf` return value is `0`.

More About

Modal Dialog Box

A modal dialog box prevents the user from interacting with other MATLAB windows before responding. For more information, see `WindowStyle` in Figure.

See Also

`inputdlg` | `questdlg`

Introduced before R2006a

listfonts

List available system fonts

Syntax

```
d = listfonts
d = listfonts(obj)
```

Description

`d = listfonts` returns an alphabetically sorted list of available system fonts.

`d = listfonts(obj)` returns available system fonts and includes the font name of the specified graphics object, if the object has `FontName` as a property.

Examples

List Available System Fonts

Call the `listfonts` function to return the list of available system fonts, with output similar in format to the one shown below.

```
d = listfonts
```

```
d = 
    {'Agency FB'                      }
    {'Algerian'                       }
    {'Arial'                          }
    ...
    {'Yu Gothic UI Semilight'         }
    {'ZapfChancery'                   }
    {'ZapfDingbats'                   }
```

Include Object Font Name in Returned System Fonts

Create a `UIControl` object with a font called `'MyFont'`. When you call `listfonts` with the object as input, the sorted list includes the object font and the system fonts.

```
c = uicontrol('Style','text','String','My Text','FontName','MyFont');
d = listfonts(c)
```

```
 d =
    {'Agency FB'                      }
    {'Algerian'                       }
    {'Arial'                          }
    ...
    {'MyFont'                         }
    ...
    {'Yu Gothic UI Semilight'         }
    {'ZapfChancery'                   }
    {'ZapfDingbats'                   }
```

Input Arguments

`obj` — Graphics object
graphics object

Graphics object, such as an `Axes`, `Annotation`, `Illustration`, or UI object.

Tips

- Some system fonts can't be rendered in MATLAB. To preview fonts that MATLAB can render in figure windows, use the `uisetfont` function. Alternatively, in the **Environment** section of the **Home** tab, select **Preferences > Fonts > Custom** to preview the available fonts that MATLAB can render.
- To determine an object's font name only, use dot notation to query the value of its `FontName` property.

See Also

`uisetfont`

Introduced in R2007a

load

Load variables from file into workspace

Syntax

```
load(filename)
load(filename,variables)
load(filename,'-ascii')
load(filename,'-mat')
load(filename,'-mat',variables)
```

```
S = load( ___ )
```

```
load filename
```

Description

`load(filename)` loads data from `filename`.

- If `filename` is a MAT-file, then `load(filename)` loads variables in the MAT-File into the MATLAB workspace.
- If `filename` is an ASCII file, then `load(filename)` creates a double-precision array containing data from the file.

`load(filename,variables)` loads the specified variables from the MAT-file, `filename`.

`load(filename,'-ascii')` treats `filename` as an ASCII file, regardless of the file extension.

`load(filename,'-mat')` treats `filename` as a MAT-file, regardless of the file extension.

`load(filename,'-mat',variables)` loads the specified variables from `filename`.

`S = load( ___ )` loads data into `S`, using any of the input arguments in the previous syntax group.

- If `filename` is a MAT-file, then `S` is a structure array.
- If `filename` is an ASCII file, then `S` is a double-precision array containing data from the file.

`load filename` is the command form of the syntax. Command form requires fewer special characters. You do not need to type parentheses or enclose input in single or double quotes. Separate inputs with spaces instead of commas.

For example, to load a file named `durer.mat`, these statements are equivalent:

```
load durer.mat      % command form
load('durer.mat')   % function form
```

You can include any of the inputs described in previous syntaxes. For example, to load the variable named `X`:

```
load durer.mat X       % command form
load('durer.mat','X')  % function form
```

Do not use command form when any of the inputs, such as `filename`, are variables or strings.

Examples

Load All Variables from MAT-File

Load all variables from the example MAT-file, `gong.mat`. Check the contents of the workspace before and after the load operation.

```
disp('Contents of workspace before loading file:')
whos

disp('Contents of gong.mat:')
whos('-file','gong.mat')

load('gong.mat')
disp('Contents of workspace after loading file:')
whos
```

You also can use command syntax to load the variables. Clear the previously loaded variables and repeat the `load` operation.

```
clear y Fs

load gong.mat
```

Load Specific Variable From MAT-File

Load only variable `y` from example file `handel.mat`. If the workspace already contains variable `y`, the `load` operation overwrites it with data from the file.

```
load('handel.mat','y')
```

You also can use command syntax to load the variable, `y`.

```
load handel.mat y
```

Use Regular Expressions to Load Specific Variables

View the contents of the example file, `accidents.mat`.

```
whos -file accidents.mat
```

```
 Name              Size            Bytes  Class     Attributes

  datasources       3x1              2724  cell
  hwycols           1x1                 8  double
  hwydata          51x17             6936  double
  hwyheaders        1x17             2758  cell
  hwyidx           51x1               408  double
  hwyrows           1x1                 8  double
  statelabel       51x1              6596  cell
  ushwydata         1x17              136  double
  uslabel           1x1               138  cell
```

Use function syntax to load all variables with names not beginning with `'hwy'`, from the file.

```
load('accidents.mat', '-regexp', '^(?!hwy)...')
```

Alternatively, use command syntax to load the same variables.

```
load accidents.mat -regexp '^(?!hwy)...'
```

Load List of Variables into Structure Array

The file, `durer.mat`, contains variables `X`, `caption`, and `map`. Create a cell array of variable names to load.

```
filename = 'durer.mat';
myVars = {'X','caption'};
S = load(filename,myVars{:})
```

```
S = struct with fields:
          X: [648x509 double]
    caption: [2x28 char]
```

Only the variables `X` and `caption` are loaded into the structure array, `S`.

Load ASCII File

Create an ASCII file from several 4-column matrices, and load the data back into a double-precision array.

```
a = magic(4);
b = ones(2, 4) * -5.7;
c = [8 6 4 2];
save -ascii mydata.dat a b c
clear a b c

load mydata.dat -ascii
```

`load` creates an array of type `double` named `mydata`.

View information about `mydata`.

```
whos mydata

  Name        Size            Bytes  Class     Attributes

  mydata      7x4               224  double
```

Input Arguments

`filename` — Name of file

`matlab.mat` (default) | character vector | string scalar

Name of file, specified as a character vector or string scalar. If you do not specify `filename`, the `load` function searches for a file named `matlab.mat`.

`filename` can include a file extension and a full or partial path. If `filename` has no extension (that is, no text after a period), `load` looks for a file named `filename.mat`. If `filename` has an extension other than `.mat`, the `load` function treats the file as ASCII data.

When using the command form of `load`, it is unnecessary to enclose the input in single quotes. However, if `filename` contains a space, you must enclose the argument in single quotes. For example, `load 'filename withspace.mat'`.

Note Do not use command form when `filename` is a string.

ASCII files must contain a rectangular table of numbers, with an equal number of elements in each row. The file delimiter (the character between elements in each row) can be a blank, comma, semicolon, or tab character. The file can contain MATLAB comments (lines that begin with a percent sign, `%`).

Example: `'myFile.mat'`

`variables` — Names of variables to load

character vector | string scalar

Names of variables to load, specified as one or more character vector or string scalar. When using the command form of `load`, you do not need to enclose the input in single quotes.

Note Do not use command form when `variables` is a string.

`variables` can be in one of the following forms.

Form of variables Input	Variables to Load
var1,...,varN	Load the listed variables, specified as individual character vectors or strings. Use the `'*'` wildcard to match patterns. For example, `load('filename.mat','A*')` or `load filename.mat A*` loads all variables in the file whose names start with `A`.
`'-regexp'`,*expr1,...,exprN*	Load only the variables or fields whose names match the regular expressions, specified as character vectors or strings. For example, `load('filename.mat','-regexp','^Mon','^Tues')` or `load filename.mat -regexp ^Mon ^Tues` loads only the variables in the file whose names begin with `Mon` or `Tues`.

Output Arguments

S — Loaded variables or data
structure array | `m`-by-`n` array

Loaded variables, returned as a structure array, if `filename` is a MAT-File.

Loaded data, returned as an `m`-by-`n` array of type `double`, if `filename` is an ASCII file. `m` is equal to the number of lines in the file, and `n` is equal to the number of values on a line.

Algorithms

If you do not specify an output for the `load` function, MATLAB creates a variable named after the loaded file (minus any file extension). For example, the command

```
load mydata.dat
```

reads data into a variable called `mydata`.

To create the variable name, `load` precedes any leading underscores or digits in `filename` with an `X` and replaces any other nonalphabetic characters with underscores. For example, the command

```
load 10-May-data.dat
```

creates a variable called `X10_May_data`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Use `load` only when generating MEX functions or code for Simulink simulation. To load compile-time constants, use `coder.load`.
- Does not support use of the function without assignment to a structure or array. For example, use `S = load(filename)`, not `load(filename)`.
- The output `S` must be the name of a structure or array without any subscripting. For example, `S(i) = load('myFile.mat')` is not supported.
- Arguments to `load` must be compile-time constant character vectors.
- If the MAT-file contains unsupported constructs, use `S = load(filename,variables)` to load only the supported constructs.
- In a function intended for code generation, the use of `save` or the use of `coder.extrinsic` with `save` is not supported. To save the workspace data to a MAT-file, use the `save` function prior to generating code.
- When using the `load` function to load variables whose size can change at run time, you must explicitly declare the variables as variable-size data by using `coder.varsize`.

See Also

`clear` | `importdata` | `matfile` | `regexp` | `save` | `uiimport` | `whos`

Topics

"Supported File Formats for Import and Export"
"Save and Load Workspace Variables"

“Import Text Files”
“Unexpected Results When Loading Variables Within a Function”
“Import or Export a Sequence of Files”
“Command vs. Function Syntax”

Introduced before R2006a

matlab.settings.loadSettingsCompatibilityResults

Package: `matlabsettings`

Get results of upgrading personal settings of toolbox for specific version

Syntax

```
results = matlab.settings.loadSettingsCompatibilityResults(
toolboxName,version)
```

Description

`results = matlab.settings.loadSettingsCompatibilityResults(toolboxName,version)` gets the results of upgrading the personal settings for the specified toolbox and version and returns them as a release compatibility results object. This function is meant for debugging purposes only and should not be included in shipping toolbox code.

After running `matlab.settings.loadSettingsCompatibilityResults`, delete the log of results before running the function again. Deleting the log ensures the correct upgrade results are always loaded. The log is located in the preferences folder, in the *`toolboxname`* folder.

Examples

Get Results of Upgrading Personal Settings

Create and then upgrade a toolbox factory tree and then test that the upgrade completes successfully.

Create the function `createMyToolboxFactoryTree` that creates the factory settings tree for the toolbox `mytoolbox`.

```
function myToolboxFactoryTree = createMyToolboxFactoryTree()
    myToolboxFactoryTree = matlab.settings.FactoryGroup.createToolboxGroup('mytoolbox', ...
        'Hidden',false);

    toolboxFontGroup = addGroup(myToolboxFactoryTree,'font','Hidden',false)
    addSetting(toolboxFontGroup,'MyFontSize','FactoryValue',11,'Hidden',false, ...
        'ValidationFcn',@matlab.settings.mustBeNumericScalar)
    addSetting(toolboxFontGroup,'MyFontColor','FactoryValue','Black', ...
        'Hidden',false,'ValidationFcn',@matlab.settings.mustBeStringScalar);
end
```

Create the function `createMyToolboxSettingsFileUpgraders` with an empty settings file upgrader object.

```
function upgraders = createMyToolboxSettingsFileUpgraders()
    upgraders = matlab.settings.SettingsFileUpgrader.empty;
end
```

Create the `settingsInfo.json` file for the toolbox. Specify `mytoolbox` as the root settings group name, `createMyToolboxFactoryTree` as the settings tree creation function, and `createMyToolboxSettingsFileUpgraders` as the settings tree upgrade function. Place `settingsInfo.json` in the toolbox `resources` folder.

```
{
"ToolboxGroupName" : "mytoolbox",
"Hidden" : "false",
"CreateTreeFcn" : "createMyToolboxFactoryTree",
"CreateUpgradersFcn" : "createMyToolboxSettingsFileUpgraders"
}
```

Add the folder that contains the settings tree creation function and the toolbox resources folder to the MATLAB path. Then, load the factory settings tree for `mytoolbox`.

```
matlab.settings.reloadFactoryFile('mytoolbox');
```

Use the `settings` function to access the root of the settings tree and set the personal value for the `MyFontSize` setting.

```
s = settings;
s.mytoolbox.font.MyFontSize.PersonalValue = 15;
```

Change the settings names in `createMyToolboxFactoryTree` to be `FontSize` and `FontColor`.

```
function myToolboxFactoryTree = createMyToolboxFactoryTree()
    myToolboxFactoryTree = matlab.settings.FactoryGroup.createToolboxGroup('mytoolbox', ...
        'Hidden',false);

    toolboxFontGroup = addGroup(myToolboxFactoryTree,'font','Hidden',false)
    addSetting(toolboxFontGroup,'FontSize','FactoryValue',11,'Hidden',false, ...
        'ValidationFcn',@matlab.settings.mustBeNumericScalar)
```

```
    addSetting(toolboxFontGroup,'FontColor','FactoryValue','Black', ...
        'Hidden',false,'ValidationFcn',@matlab.settings.mustBeStringScalar);
end
```

Record the rename of the two settings in the `createMyToolboxSettingsFileUpgraders` function as changes to the settings tree for version 2 of `mytoolbox`.

```
function upgraders = createMyToolboxSettingsFileUpgraders()
    upgraders = matlab.settings.SettingsFileUpgrader('Version2');
    upgraders.move('mytoolbox.font.MyFontSize','mytoolbox.font.FontSize');
    upgraders.move('mytoolbox.font.MyFontColor','mytoolbox.font.FontColor');
end
```

Reload the factory settings tree for `mytoolbox`.

```
matlab.settings.reloadFactoryFile('mytoolbox');
```

Use the `settings` function to access the root of the settings tree and verify that the personal value for the `FontSize` setting was correctly moved from the `MyFontSize` setting.

```
s = settings;
s.mytoolbox.font.FontSize
```

```
ans = 
  Setting 'mytoolbox.font.FontSize' with properties:
       ActiveValue: 15
     TemporaryValue: <no value>
      PersonalValue: 15
       FactoryValue: 11
```

Get the upgrade results for version 2 of `mytoolbox`.

```
matlab.settings.loadSettingsCompatibilityResults('mytoolbox','Version2')
```

```
ans = 
  ReleaseCompatibilityResults with properties:
              VersionLabel: "Version2"
    PreValidationExceptions: [0×0 matlab.settings.ReleaseCompatibilityException]
                    Results: [1×1 matlab.settings.VersionResults]
```

Input Arguments

`toolboxName` — Name of toolbox

character vector | string

Name of toolbox to get the upgrade results for, specified as a character vector or string.

Example: `'mytoolbox'`

`version` — Toolbox version
character vector | string

Toolbox version to get the upgrade results for, specified as a character vector or string.

Example: `'version2'`

See Also

`ReleaseCompatibilityResults` | `SettingsFileUpgrader` | `matlab.settings.reloadFactoryFile`

Topics

"Create Factory Settings for Toolboxes"
"Access and Modify Settings"

Introduced in R2019b

loadlibrary

Load C shared library into MATLAB

Syntax

```
loadlibrary(libname,hfile)
loadlibrary(libname)
loadlibrary(libname,hfile,Name,Value)

loadlibrary(libname,@protofile)

[notfound,warnings] = loadlibrary( ___ )
```

Description

`loadlibrary(libname,hfile)` loads functions from C shared library `libname` defined in header file `hfile` into MATLAB. The `loadlibrary` function only supports calling functions that are callable from C and header files that can be parsed by a C compiler. Functions written in C++ must be declared as `extern "C"`.

`loadlibrary(libname)` loads the library if the name of the header file is the same as the name of the library file.

`loadlibrary(libname,hfile,Name,Value)` loads the library with one or more `Name,Value` arguments.

`loadlibrary(libname,@protofile)` uses a prototype file on page 1-8716, `protofile`, in place of a header file.

`[notfound,warnings] = loadlibrary( ___ )` returns warning information, and can include any of the input arguments in previous syntaxes.

Examples

Display Functions in `shrlibsample` Library

Add path to `examples` folder.

```
addpath(fullfile(matlabroot,'extern','examples','shrlib'))
```

Display functions in library.

```
if not(libisloaded('shrlibsample'))
    loadlibrary('shrlibsample')
end
libfunctions('shrlibsample')
```

```
Functions in library shrlibsample:

addDoubleRef              exportedDoubleValue       printExportedDoubleValue
addMixedTypes             getListOfStrings          readEnum
addStructByRef            multDoubleArray           stringToUpper
addStructFields           multDoubleRef
allocateStruct            multiplyShort
deallocateStruct          print2darray
```

Clean up.

```
unloadlibrary shrlibsample
```

Load Library Using Multiple Header Files

Suppose that you have a library, `mylib`, with the header file, `mylib.h`. The header file contains the statement, `#include header2.h`. To use functions defined in `header2.h`, call `loadlibrary` with the `addheader` option.

```
loadlibrary('mylib','mylib.h','addheader','header2')
```

Call `stringToUpper` Function Using Alias Name

```
if libisloaded('shrlibsample')
    unloadlibrary('shrlibsample')
else
    addpath(fullfile(matlabroot,'extern','examples','shrlib'))
end
```

Create an alias name `lib` for library `shrlibsample`.

```
loadlibrary('shrlibsample','shrlibsample.h','alias','lib')
```

Call function `stringToUpper` using the alias name.

```
str = 'This was a Mixed Case string';
calllib('lib','stringToUpper',str)
```

```
ans =
'THIS WAS A MIXED CASE STRING'
```

Clean up.

```
unloadlibrary lib
```

Search Alternative Paths for Header Files

Add path to folder containing `shrlibsample` and its header file, `shrlibsample.h`.

```
addpath(fullfile(matlabroot,'extern','examples','shrlib'))
```

The `shrlibsample.h` header file includes the header file, `shrhelp.h`. If `shrhelp.h` is in a different folder, for example, `c:\work`, use the `'includepath'` option to tell MATLAB where to find the file.

```
loadlibrary('shrlibsample','shrlibsample.h','includepath','c:\work')
```

Cleanup.

```
unloadlibrary shrlibsample
```

Create Alias Name for `shrlibsample` Function

This example shows how to replace the `addMixedTypes` function name in the MATLAB `shrlibsample` library with an alias name, `addTypes`. To define the alias name, create a prototype file then load the library using the prototype file as the header file.

Use a folder for which you have write-access.

```
cd('c:\work')
```

Create a prototype file, `mxproto.m`.

```
hfile = fullfile(matlabroot,'extern','examples','shrlib','shrlibsample.h');
[notfound,warnings] = loadlibrary('shrlibsample',hfile,'mfilename','mxproto')
```

MATLAB creates the prototype file in the current folder. Ignore the warning messages.

Add the alias name to the prototype file. Open the file in MATLAB Editor.

```
edit mxproto.m
```

Search for the function `addMixedTypes`.

The following statement assigns the alias `addTypes`.

```
fcns.alias{fcnNum}='addTypes';
```

Add the statement to the line before the statement to increment `fcnNum`. The new function prototype, with the new statement shown in bold, looks like the following:

```
%  double addMixedTypes ( short , int , double );
fcns.thunkname{fcnNum}='doubleint16int32doubleThunk';
fcns.name{fcnNum}='addMixedTypes';
fcns.calltype{fcnNum}='Thunk';
fcns.LHS{fcnNum}='double';
fcns.RHS{fcnNum}={'int16', 'int32', 'double'};
fcns.alias{fcnNum}='addTypes'; % Alias defined
fcnNum=fcnNum+1; % Increment fcnNum
```

Reload `shrlibsample` using the prototype file.

```
unloadlibrary shrlibsample
loadlibrary('shrlibsample',@mxproto)
```

Call the function by its alias name.

```
calllib('shrlibsample','addTypes',int16(127),int32(33000),pi)
```

```
ans = 3.3130e+04
```

Cleanup.

```
unloadlibrary shrlibsample
```

Input Arguments

`libname` — Name of shared library

character vector

Name of shared library, specified as a character vector. The name is case-sensitive and must match the file on your system.

On Microsoft Windows systems, `libname` refers to the name of a shared library (`.dll`) file. On Linux systems, it refers to the name of a shared object (`.so`) file. On Apple Mac systems, it refers to a dynamic shared library (`.dylib`). If you do not include a file extension with the `libname` argument, `loadlibrary` attempts to find the library with either the appropriate platform MEX-file extension or the appropriate platform library extension. For a list of MEX-file extensions, use `mexext`.

MATLAB extracts the name portion of `libname` to identify the library in other shared library functions. For example, when you call the `calllib` function, do not include the path or file extension in the library argument name.

Data Types: `char`

`hfile` — Name of C header file

character vector

Name of C header file, specified as a character vector. The name is case-sensitive and must match the file on your system. If you do not include a file extension in the file name, `loadlibrary` uses `.h` for the extension.

Data Types: `char`

`protofile` — Name of prototype file

character vector

Name of prototype on page 1-8716 file, specified as a character vector. The name is case-sensitive and must match the file on your system. `@protofile` specifies a function handle to the prototype file. When using a prototype file, the only valid `Name,Value` pair argument is **`alias`**.

Data Types: `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `loadlibrary('mylib','mylib.h','addheader','header2')`

`addheader` — Header file

character vector

Header file, specified as the comma-separated pair consisting of '`addheader`' and a character vector. Specify the file name without a file extension.

Each file specified by `addheader` must have a corresponding `#include` statement in the base header file. To load only the functions defined in the header file that you want to use in MATLAB, use `addheader`.

MATLAB does not verify the existence of header files and ignores any that are not needed.

`alias` — Alternative name for library

character vector

Alternative name for library, specified as the comma-separated pair consisting of '`alias`' and a character vector. Associates the specified name with the library. All subsequent calls to MATLAB functions that reference this library must use this alias until the library is unloaded.

`includepath` — More search paths for subordinate header files

character vector

More search paths for subordinate header files—header files within header files, specified as the comma-separated pair consisting of '`includepath`' and a character vector.

`mfilename` — Prototype file

character vector

Prototype file, specified as the comma-separated pair consisting of '`mfilename`' and a character vector. Generates a prototype on page 1-8716 file in the current folder. The prototype file name must be different from the library name. Use this file in place of a header file when loading the library.

`thunkfilename` — Thunk file
character vector

Thunk file, specified as the comma-separated pair consisting of '`thunkfilename`' and a character vector. Overrides the default thunk file on page 1-8716 name.

Output Arguments

`notfound` — Names of functions
cell array

Names of functions found in header files but missing from the library, returned as cell array.

Data Types: `cell`

`warnings` — Warnings
character array

Warnings produced while processing the header file, returned as character array.

Limitations

- You must have a supported C compiler and Perl must be available.
- Do not call `loadlibrary` if the library is already in memory. To test this condition, call `libisloaded`.
- `loadlibrary` does not support libraries generated by the MATLAB Compiler SDK product.
- The MATLAB Shared Library interface does not support library functions with function pointer inputs.
- For more information, see “Limitations to Shared Library Support”.

More About

Prototype File

A prototype file is a file of MATLAB commands which you can modify and use in place of a header file.

Thunk File

A thunk file is a compatibility layer to a 64-bit library generated by MATLAB. The name of the thunk file is `BASENAME_thunk_COMPUTER.c` where *`BASENAME`* is either the name of the shared library or, if specified, the `mfilename` prototype name. *`COMPUTER`* is the text returned by the `computer` function.

MATLAB compiles this file and creates the file `BASENAME_thunk_COMPUTER.LIBEXT`, where *`LIBEXT`* is the platform-dependent default shared library extension, for example, `dll` on Windows.

Tips

- If you have more than one library file of the same name, load the first using the library file name. Then load the additional libraries using the **`alias`** option.
- Use the **`alias`** option as an alternate name for a library. To load an alternate header file, use the `@protofile` argument.

See Also

`calllib` | `computer` | `libfunctions` | `libisloaded` | `mex` | `mexext` | `unloadlibrary`

Topics

"When to Use Prototype Files"

External Websites

Supported and Compatible Compilers

Introduced before R2006a

loadobj

Customize load process for objects

Syntax

```
b = loadobj(a)
```

Description

`b = loadobj(a)` is called by the `load` function if the class of `a` defines a `loadobj` method. `load` returns `b` as the value loaded from a MAT-file.

Define a `loadobj` method when objects of the class require special processing when loaded from MAT-files. If you define a `saveobj` method, then define a `loadobj` method to restore the object to the desired state.

Define `loadobj` as a static method so it can accept as an argument whatever object or structure that you saved in the MAT-file.

When loading a subclass object, `load` calls only the subclass `loadobj` method. If a superclass defines a `loadobj` method, the subclass inherits this method. However, it is possible that the inherited method does not perform the necessary operations to load the subclass object. Consider overriding superclass `loadobj` methods.

If any superclass in a class hierarchy defines a `loadobj` method, then the subclass `loadobj` method must ensure that the subclass and superclass objects load properly. Ensure proper loading by calling the superclass `loadobj` (or other methods) from the subclass `loadobj` method.

Input Arguments

a

The input argument, `a`, can be:

- The object as loaded from the MAT-file.
- A structure created by `load` (if `load` cannot resolve the object).
- A structure returned by the `saveobj` method that was saved instead of the object.

Implement your `loadobj` method to work with scalar objects or structures. When you have saved an object array, `load` calls `loadobj` on each element of the saved array.

Output Arguments

b — Object passed to `load` function
MATLAB object

Object passed to `load` function by MATLAB.The value returned by a class `loadobj` method is typically an object of the class being loaded. However, the `loadobj` method can return an object of a different class or an updated object that matches a new class definition.

See Also

`load` | `save` | `saveobj`

Topics

"Save and Load Process for Objects"
"Object Save and Load"

Introduced before R2006a

matlab.project.loadProject

Package: `matlab.project`

Load project

Syntax

```
proj = matlab.project.loadProject(projectPath)
```

Description

`proj = matlab.project.loadProject(projectPath)` loads the project specified by the file or folder `projectPath`. If any projects are currently open, MATLAB closes them before loading the specified project.

Examples

Load a Project

Load a project from a folder called `"C:/projects/project1/"`.

```
proj = matlab.project.loadProject("C:/projects/project1/")
```

Input Arguments

`projectPath` — Full path to project file or folder
character vector | string scalar

Full path to project `.prj` file or project root folder, specified as a character vector or string scalar.

Example: `"C:/projects/project1/myProject.prj"`

Example: `"C:/projects/project1/"`

Output Arguments

`proj` — Project
`matlab.project.Project` object

Project, returned as a `matlab.project.Project` object. Use the `matlab.project.Project` object to programmatically manipulate the currently open project.

See Also

`currentProject` | `matlab.project.rootProject` | `openProject`

Topics

"Create and Edit Projects Programmatically"

Introduced in R2019a

localfunctions

Function handles to all local functions in MATLAB file

Syntax

```
fcns = localfunctions
```

Description

`fcns = localfunctions` returns a cell array of function handles, `fcns`, to all local functions in the current file.

You cannot define local functions in the context of the command line or anonymous functions, so when you call `localfunctions` from these contexts, you get an empty cell array. Within the cell array, `localfunctions` returns the function handles in an undefined order.

Examples

Create Handles to Local Functions in Function File

Create the following function in a file, `computeEllipseVals.m`, in your working folder. The function returns a cell array with handles to all the local functions.

```
function fh = computeEllipseVals
fh = localfunctions;
end

function f = computeFocus(a,b)
f = sqrt(a^2-b^2);
end

function e = computeEccentricity(a,b)
```

```
f = computeFocus(a,b);
e = f/a;
end

function ae = computeArea(a,b)
ae = pi*a*b;
end
```

At the command prompt, invoke the function to get a cell array of handles to the local functions.

```
fh = computeEllipseVals
```

```
fh =

  3x1 cell array

    {       @computeFocus}
    {@computeEccentricity}
    {        @computeArea}
```

Call a local function using its handle to compute the area of an ellipse. The `computeArea` function handle is the third element in the cell array.

```
fh{3}(3,1)
```

```
ans =

    9.4248
```

Create Handles to Local Functions in Script File

As of R2016b, you can include local functions in scripts. Therefore, you can use the `localfunctions` function to create function handles that you can invoke in the script or at the command prompt.

Create the following script in a file, `mystats.m`, in your working folder. The script creates a cell array with handles to all the local functions.

```
x = [1 3 5 7 9 10 8 6 4 2 0 -2];

avg = mymean(x)

fh = localfunctions;
med = fh{2}(x) % equivalent to med = mymedian(x,n)

function a = mymean(v)
n = length(v);
a = sum(v)/n;
end

function m = mymedian(v)
n = length(v);
w = sort(v);
if rem(n,2) == 1
    m = w((n + 1)/2);
else
    m = (w(n/2) + w(n/2 + 1))/2;
end
end
```

Run the script. MATLAB® computes the average by directly invoking the mymean local function and the median by invoking `mymedian` local function through a function handle.

```
mystats
```

```
avg =

    4.4167


med =

    4.5000
```

At the command prompt, call the `mymean` local function using its handle. Variables from the script are accessible at the command prompt. The `mymean` function handle is the first element in the cell array.

```
x2 = [1 1 2 6 24 120 720 5040];
fh
avg2 = fh{2}(x2)
```

```
fh =

  2x1 cell array

    {  @mymean}
    {@mymedian}


avg2 =

    15
```

See Also

functiontests

Topics

"Local Functions"

Introduced in R2013b

log

Natural logarithm

Syntax

```
Y = log(X)
```

Description

`Y = log(X)` returns the natural logarithm ln(x) of each element in array `X`.

The `log` function's domain includes negative and complex numbers, which can lead to unexpected results if used unintentionally. For negative and complex numbers `z = u + i*w`, the complex logarithm `log(z)` returns

```
log(abs(z)) + 1i*angle(z)
```

If you want negative and complex numbers to return error messages rather than return complex results, use `reallog` instead.

Examples

Natural Logarithm of Negative Number

Show that the natural logarithm of -1 is $i\pi$.

```
log(-1)
```

```
ans = 0.0000 + 3.1416i
```

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Logarithm values
scalar | vector | matrix | multidimensional array

Logarithm values, returned as a scalar, vector, matrix, or multidimensional array.

For positive real values of `X` in the interval (`0`, `Inf`), `Y` is in the interval (`-Inf`,`Inf`). For complex and negative real values of `X`, `Y` is complex. The data type of `Y` is the same as that of `X`.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- When the input value `x` is real, but the output should be complex, simulation ends with an error. To produce the complex result, make the input value complex by passing in `complex(x)`.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- When the input value `x` is real, but the output should be complex, simulation ends with an error. To produce the complex result, make the input value complex by passing in `complex(x)`.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- If the output of the function running on the GPU can be complex, then you must explicitly specify its input arguments as complex. For more information, see “Work with Complex Numbers on a GPU” (Parallel Computing Toolbox).

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`exp` | `log10` | `log1p` | `log2` | `loglog` | `logm` | `reallog` | `semilogx` | `semilogy`

Introduced before R2006a

log10

Common logarithm (base 10)

Syntax

```
Y = log10(X)
```

Description

`Y = log10(X)` returns the common logarithm of each element in array `X`. The function accepts both real and complex inputs. For real values of `X` in the interval (0, `Inf`), `log10` returns real values in the interval (`-Inf` ,`Inf`). For complex and negative real values of `X`, the `log10` function returns complex values.

Examples

Calculate Scalar Common Logarithm Values

Examine several values of the base 10 logarithm function.

Calculate the common logarithm of 1.

```
log10(1)
```

```
ans = 0
```

The result is `0`, so this is the x-intercept of the `log10` function.

Calculate the common logarithm of 10.

```
log10(10)
```

```
ans = 1
```

The result is `1` since $10^1 = 10$.

Calculate the common logarithm of 100.

```
log10(100)
```

```
ans = 2
```

The result is 2 since $10^2 = 100$.

Calculate the common logarithm of 0.

```
log10(0)
```

```
ans = -Inf
```

The result is `-Inf` since $10^{-\infty} = 0$.

Real-Valued Common Logarithm

Create a vector of numbers in the interval `[0.5 5]`.

```
X = (0.5:0.5:5)';
```

Calculate the common logarithm of X.

```
Y = log10(X)
```

```
Y = 10×1

   -0.3010
         0
    0.1761
    0.3010
    0.3979
    0.4771
    0.5441
    0.6021
    0.6532
    0.6990
```

Complex-Valued Common Logarithm

Create two Cartesian grids for X and Y.

```
[X,Y] = meshgrid(0:0.5:1.5,-2:0.5:2);
```

Calculate the complex base 10 logarithm $log_{10}(X + iY)$ on the grid. Use `1i` for improved speed and robustness with complex arithmetic.

```
Z = log10(X + 1i*Y)
```

```
Z = 9×4 complex

   0.3010 - 0.6822i   0.3142 - 0.5758i   0.3495 - 0.4808i   0.3979 - 0.4027i
   0.1761 - 0.6822i   0.1990 - 0.5425i   0.2559 - 0.4268i   0.3266 - 0.3411i
   0.0000 - 0.6822i   0.0485 - 0.4808i   0.1505 - 0.3411i   0.2559 - 0.2554i
  -0.3010 - 0.6822i  -0.1505 - 0.3411i   0.0485 - 0.2014i   0.1990 - 0.1397i
     -Inf + 0.0000i  -0.3010 + 0.0000i   0.0000 + 0.0000i   0.1761 + 0.0000i
  -0.3010 + 0.6822i  -0.1505 + 0.3411i   0.0485 + 0.2014i   0.1990 + 0.1397i
   0.0000 + 0.6822i   0.0485 + 0.4808i   0.1505 + 0.3411i   0.2559 + 0.2554i
   0.1761 + 0.6822i   0.1990 + 0.5425i   0.2559 + 0.4268i   0.3266 + 0.3411i
   0.3010 + 0.6822i   0.3142 + 0.5758i   0.3495 + 0.4808i   0.3979 + 0.4027i
```

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- If the output of the function running on the GPU can be complex, then you must explicitly specify its input arguments as complex. For more information, see “Work with Complex Numbers on a GPU” (Parallel Computing Toolbox).

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`exp` | `log` | `log1p` | `log2` | `loglog` | `logm` | `reallog` | `semilogx` | `semilogy`

Introduced before R2006a

log1p

Compute log(1+x) accurately for small values of x

Syntax

```
y = log1p(x)
```

Description

`y = log1p(x)` computes `log(1+x)`, compensating for the roundoff in `1+x`. `log1p(x)` is more accurate than `log(1+x)` for small values of x. For small `x`, `log1p(x)` is approximately `x`, whereas `log(1+x)` can be zero.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- If the output of the function running on the GPU can be complex, then you must explicitly specify its input arguments as complex. For more information, see “Work with Complex Numbers on a GPU” (Parallel Computing Toolbox).

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`expm1` | `log`

Introduced before R2006a

log2

Base 2 logarithm and floating-point number dissection

Syntax

```
Y = log2(X)
[F,E] = log2(X)
```

Description

`Y = log2(X)` computes the base 2 logarithm of the elements of `X` such that $2^Y = X$.

`[F,E] = log2(X)` returns arrays `F` and `E` such that $X = F \cdot 2^E$. The values in `F` are typically in the range `0.5 <= abs(F) < 1`.

Examples

Base 2 Logarithm Values

```
X = [0 1 2 10 Inf NaN];
Y = log2(X)
```

```
Y = 1×6

      -Inf         0    1.0000    3.3219       Inf       NaN
```

Floating-Point Number Dissection

Dissect several numbers into the exponent and mantissa. These operations all follow standard IEEE® arithmetic.

Create a vector X that contains several test values. Calculate the exponent and mantissa for each number.

```
X = [1 pi -3 eps realmax realmin];
format rat
[F,E] = log2(X)
```

```
F =
  Columns 1 through 5

       1/2          355/452         -3/4            1/2            1

  Column 6

       1/2


E =
  Columns 1 through 5

       1              2              2            -51           1024

  Column 6

   -1021
```

Collect the results in a table. Convert the numbers into character vectors for display purposes.

```
x = {'1','pi','-3','eps','realmax','realmin'}';
f = strtrim(cellstr(rats(F')));
T = table(x,f,E','VariableNames',{'Value','Mantissa','Exponent'})
```

```
T=6×3 table
       Value        Mantissa      Exponent
    ___________    ___________    ________

    {'1'      }    {'1/2'    }         1
    {'pi'     }    {'355/452'}         2
    {'-3'     }    {'-3/4'   }         2
    {'eps'    }    {'1/2'    }       -51
    {'realmax'}    {'1'      }      1024
    {'realmin'}    {'1/2'    }     -1021
```

The results indicate that, for the first row, $1 = \frac{1}{2}\left(2^{1}\right)$. Similarly, for the fourth row, $\text{eps} = \frac{1}{2}\left(2^{-51}\right)$.

Input Arguments

X — Input matrix
scalar | vector | matrix | multidimensional array

Input matrix, specified as a scalar, vector, matrix, or multidimensional array.

For floating-point number dissection `[F,E] = log2(X)`, any zeros in `X` produce `F = 0` and `E = 0`. Input values of `Inf`, `-Inf`, or `NaN` are returned unchanged in `F` with a corresponding exponent of `E = 0`.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Base 2 logarithm values
scalar | vector | matrix | multidimensional array

Base 2 logarithm values, returned as a scalar, vector, matrix, or multidimensional array of the same size as `X`.

F — Mantissa values
scalar | vector | matrix | multidimensional array

Mantissa values, returned as a scalar, vector, matrix, or multidimensional array of the same size as `X`. The values in `F` and `E` satisfy `X = F.*2.^E`.

E — Exponent values
scalar | vector | matrix | multidimensional array

Exponent values, returned as a scalar, vector, matrix, or multidimensional array of the same size as `X`. The values in `F` and `E` satisfy `X = F.*2.^E`.

Tips

- This function corresponds to the ANSI C function `frexp()` and the IEEE floating-point standard function `logb()`. Any zeros in `X` produce `F = 0` and `E = 0`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The syntax `[F,E] = log2(X)` is not supported.
- If the output of the function running on the GPU can be complex, then you must explicitly specify its input arguments as complex. For more information, see “Work with Complex Numbers on a GPU” (Parallel Computing Toolbox).

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`log` | `log10` | `pow2`

Introduced before R2006a

logical

Convert numeric values to logicals

Syntax

```
L = logical(A)
```

Description

`L = logical(A)` converts `A` into an array of logical values. Any nonzero element of `A` is converted to logical `1` (`true`) and zeros are converted to logical `0` (`false`). Complex values and NaNs cannot be converted to logical values and result in a conversion error.

Examples

Pick Odd Elements from Numeric Matrix

Pick out the odd-numbered elements of a numeric matrix.

Create a numeric matrix.

```
A = [1 -3 2;5 4 7;-8 1 3];
```

Find the modulus, `mod(A,2)`, and convert it to a logical array for indexing.

```
L = logical(mod(A,2))
```

```
L = 3x3 logical array

   1   1   0
   1   0   1
   0   1   1
```

The array has logical `1` (`true`) values where `A` is odd.

Use L as a logical index to pick out the odd elements of A.

```
A(L)
```

```
ans = 6×1

     1
     5
    -3
     1
     7
     3
```

The result is a vector containing all odd elements of A.

Use the logical NOT operator, ~, on L to find the even elements of A.

```
A(~L)
```

```
ans = 3×1

    -8
     4
     2
```

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`

Tips

- Most arithmetic operations involving logical arrays return double values. For example, adding zero to a logical array returns a double array.

- Logical arrays also are created by the relational operators (==,<,>,~, etc.) and functions like `any`, `all`, `isnan`, `isinf`, and `isfinite`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`false` | `islogical` | `true`

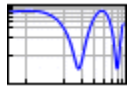
Topics

"Array Indexing"
"Find Array Elements That Meet a Condition"

Introduced before R2006a

loglog

Log-log scale plot



Syntax

```
loglog(Y)
loglog(X1,Y1,...)
loglog(X1,Y1,LineSpec,...)
loglog(...,'PropertyName',PropertyValue,...)
loglog(ax,...)
h = loglog(...)
```

Description

`loglog(Y)` plots the columns of `Y` versus their index if `Y` contains real numbers. If `Y` contains complex numbers, `loglog(Y)` and `loglog(real(Y),imag(Y))` are equivalent. `loglog` ignores the imaginary component in all other uses of this function.

`loglog(X1,Y1,...)` plots all `Yn` versus `Xn` pairs. If only one of `Xn` or `Yn` is a matrix, `loglog` plots the vector argument versus the rows or columns of the matrix, along the dimension of the matrix whose length matches the length of the vector. If the matrix is square, its columns plot against the vector if their lengths match.

`loglog(X1,Y1,LineSpec,...)` plots all lines defined by the `Xn,Yn,LineSpec` triples, where `LineSpec` determines line type, marker symbol, and color of the plotted lines. You can mix `Xn,Yn,LineSpec` triples with `Xn,Yn` pairs, for example,

```
loglog(X1,Y1,X2,Y2,LineSpec,X3,Y3)
```

`loglog(...,'PropertyName',PropertyValue,...)` sets line property values for all the charting lines created. For a list of properties, see Chart Line.

`loglog(ax,...)` creates the line in the axes specified by `ax` instead of in the current axes (`gca`). The option `ax` can precede any of the input argument combinations in the previous syntaxes.

`h = loglog(...)` returns a column vector of chart line objects.

If you do not specify a color when plotting more than one line, `loglog` automatically cycles through the colors and line styles in the order specified by the current axes.

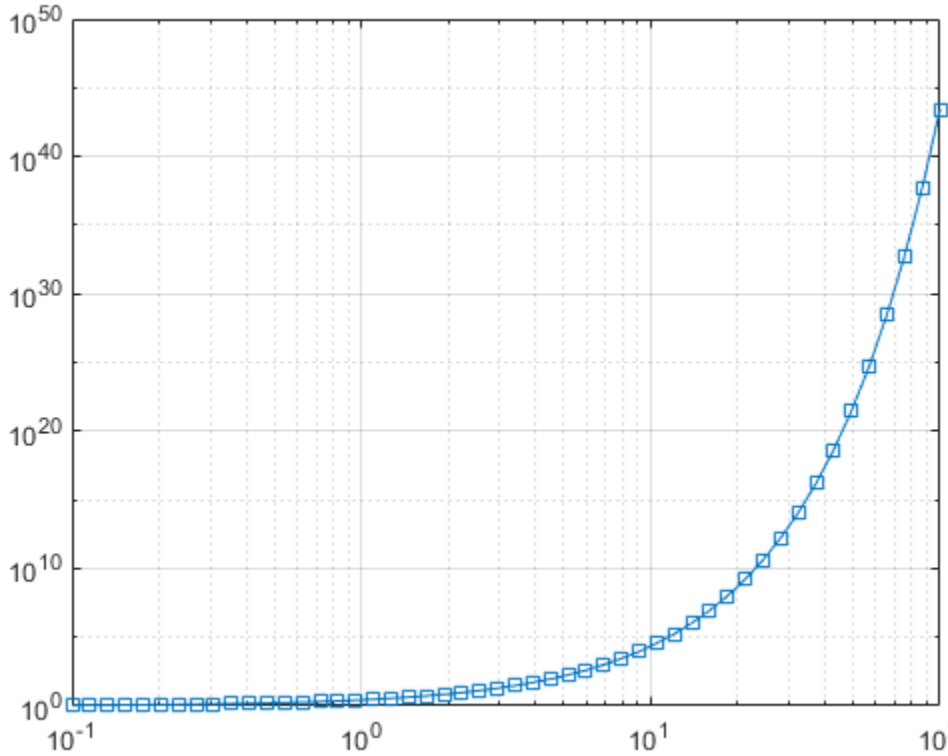
If you attempt to add a `loglog`, `semilogx`, or `semilogy` plot to a linear axis mode graph with `hold on`, the axis mode remains as it is and the new data plots as linear.

Examples

Logarithmic Scale for Both Axes

Create a plot using a logarithmic scale for both the x-axis and the y-axis. Use a line with square markers. Display the grid.

```
x = logspace(-1,2);
y = exp(x);
loglog(x,y,'-s')
grid on
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions
`LineSpec` | `plot` | `semilogx` | `semilogy`

Properties
Chart Line

Introduced before R2006a

logm

Matrix logarithm

Syntax

```
L = logm(A)
[L,exitflag] = logm(A)
```

Description

`L = logm(A)` is the principal matrix logarithm of `A`, the inverse of `expm(A)`. The output, `L`, is the unique logarithm for which every eigenvalue has imaginary part lying strictly between $-\pi$ and π. If `A` is singular or has any eigenvalues on the negative real axis, then the principal logarithm is undefined. In this case, `logm` computes a nonprincipal logarithm and returns a warning message.

`[L,exitflag] = logm(A)` returns a scalar `exitflag` that describes the exit condition of `logm`:

- If `exitflag = 0`, the algorithm was successfully completed.
- If `exitflag = 1`, too many matrix square roots had to be computed. However, the computed value of `L` might still be accurate.

Examples

Compare Matrix Logarithms

Calculate the matrix exponential of a matrix, `A`.

```
A = [1 1 0; 0 0 2; 0 0 -1];
Y = expm(A)
```

```
Y = 3×3
```

```
    2.7183    1.7183    1.0862
         0    1.0000    1.2642
         0         0    0.3679
```

Calculate the matrix logarithm of `Y` to reproduce the original matrix, `A`.

```
P = logm(Y)
```

```
P = 3×3

    1.0000    1.0000    0.0000
         0         0    2.0000
         0         0   -1.0000
```

`log(A)` involves taking the logarithm of zero, so it produces inferior results.

```
Q = log(A)
```

```
Q = 3×3 complex

   0.0000 + 0.0000i   0.0000 + 0.0000i     -Inf + 0.0000i
     -Inf + 0.0000i     -Inf + 0.0000i   0.6931 + 0.0000i
     -Inf + 0.0000i     -Inf + 0.0000i   0.0000 + 3.1416i
```

Input Arguments

A — Input matrix
square matrix

Input matrix, specified as a square matrix.

Data Types: `single` | `double`
Complex Number Support: Yes

Tips

- If `A` is real symmetric or complex Hermitian, then so is `logm(A)`.

- Some matrices, like `A = [0 1; 0 0]`, do not have any logarithms, real or complex, so `logm` cannot be expected to produce one.

Algorithms

The algorithm `logm` uses is described in [1] and [2].

References

[1] Al-Mohy, A. H. and Nicholas J. Higham, "Improved inverse scaling and squaring algorithms for the matrix logarithm," *SIAM J. Sci. Comput.*, 34(4), pp. C153–C169, 2012

[2] Al-Mohy, A. H., Higham, Nicholas J. and Samuel D. Relton, "Computing the Frechet derivative of the matrix logarithm and estimating the condition number," *SIAM J. Sci. Comput.*,, 35(4), pp. C394–C410, 2013

See Also

`expm` | `funm` | `sqrtm`

Introduced before R2006a

logspace

Generate logarithmically spaced vector

Syntax

```
y = logspace(a,b)
y = logspace(a,b,n)
y = logspace(a,pi)
y = logspace(a,pi,n)
```

Description

`y = logspace(a,b)` generates a row vector `y` of 50 logarithmically spaced points between decades `10^a` and `10^b`. The `logspace` function is especially useful for creating frequency vectors. The function is the logarithmic equivalent of `linspace` and the '`:`' operator.

`y = logspace(a,b,n)` generates `n` points between decades `10^a` and `10^b`.

`y = logspace(a,pi)` generates 50 points between `10^a` and `pi`, which is useful in digital signal processing for creating logarithmically spaced frequencies in the interval `[10^a,pi]`.

`y = logspace(a,pi,n)` generates `n` points between `10^a` and `pi`.

Examples

Vector of Logarithmically Spaced Numbers

Create a vector of 50 logarithmically spaced points in the interval `[10^1,10^5]`.

```
y = logspace(1,5);
```

Vector with Specified Number of Values

Create a vector of 7 logarithmically spaced points in the interval `[10^1,10^5]`.

```
y1 = logspace(1,5,7)
```

```
y1 = 1×7
10^5 ×

    0.0001    0.0005    0.0022    0.0100    0.0464    0.2154    1.0000
```

Vector of Logarithmically Spaced Complex Numbers

Create a vector of complex numbers with 8 logarithmically spaced points between `10^(1+2i)` and `10^(5+5i)`.

```
y = logspace(1+2i,5+5i,8)
```

```
y = 1×8 complex
10^4 ×

  -0.0001 - 0.0010i   0.0029 - 0.0024i   0.0133 + 0.0040i   0.0147 + 0.0497i  -0.1242 -
```

Input Arguments

a — First bound
numeric scalar

First bound, specified as a numeric scalar. The `a` argument defines a bound of the interval over which `logspace` generates points. `a` can be real or complex, and the other bound, `b`, can be either larger or smaller than `a`. If `b` is smaller than `a`, then the vector contains descending values.

Data Types: `single` | `double`
Complex Number Support: Yes

b — Second bound
numeric scalar

Second bound, specified as a numeric scalar. The `b` argument defines a bound of the interval over which `logspace` generates points. `b` can be real or complex, and `b` can be either larger or smaller than the other bound, `a`. If `b` is smaller than `a`, then the vector contains descending values.

Data Types: `single` | `double`
Complex Number Support: Yes

n — Number of points
50 (default) | real numeric scalar

Number of points, specified as a real numeric scalar.

- If `n` is `1`, `logspace` returns `10^b`.
- If `n` is zero or negative, `logspace` returns an empty row vector.
- If `n` is not an integer, `logspace` rounds `n` down and returns `floor(n)` points.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- Use `distributed.logspace` to call the distributed version of `logspace`.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`colon` | `linspace`

Introduced before R2006a

lookfor

Search for keyword in all help entries

Syntax

```
lookfor keyword
lookfor keyword -all
```

Description

`lookfor keyword` searches for the specified keyword in the first comment line (the H1 line) of the help text in all MATLAB program files found on the search path. For all files in which a match occurs, `lookfor` displays the H1 line.

`lookfor` is useful if you are searching for a function and you do not know its name. To find a function whose name you already know, use the `what` and `which` functions, which run much faster.

`lookfor keyword -all` searches the entire first comment block of a MATLAB program file.

Examples

Search for Text in H1 Help

Search for the word `literal` in the H1 line of the help text for all MATLAB program files.

```
lookfor inverse
```

```
invhilb                       - Inverse Hilbert matrix.
ipermute                      - Inverse permute array dimensions.
dramadah                      - Matrix of zeros and ones with large determinant or inverse.
invhess                       - Inverse of an upper Hessenberg matrix.
betaincinv                    - Inverse incomplete beta function.
...
```

Search for Text in All Help

Search for the word `literal` in all of the help text for all MATLAB program files.

```
lookfor inverse -all
```

```
bsxfun:                 @atan2            Four-quadrant inverse tangent; result in radians
bsxfun:                 @atan2d            Four-quadrant inverse tangent; result in dgrees

gallery:     dramadah    Matrix of ones and zeroes whose inverse has large integer entries.
gallery:     invhess     Inverse of an upper Hessenberg matrix.

hilb:     calculates the exact inverse.

invhilb: Inverse Hilbert matrix.
invhilb:     IH = INVHILB(N) is the inverse of the N-by-N matrix with elements

ipermute: Inverse permute array dimensions.
ipermute:     A = IPERMUTE(B,ORDER) is the inverse of permute.  IPERMUTE rearranges
...
```

Input Arguments

`keyword` — Keyword to search for

character vector | string scalar

Keyword to search for, specified as a character vector or string scalar.

Alternative Functionality

As an alternative to the `lookfor` command, use the find features in the Current Folder browser. For example, you can look for all occurrences of a specified word in all the MATLAB program files in the current folder and its subfolders. For more information, see “Find Files and Folders”.

To search the documentation, including third-party and custom documentation, use the `docsearch` function.

See Also

`doc` | `docsearch` | `help` | `strfind` | `what` | `which`

Topics

"Search Syntax and Tips"
"Find Files and Folders"

Introduced before R2006a

lower

Convert strings to lowercase

Syntax

```
newStr = lower(str)
```

Description

`newStr = lower(str)` converts all uppercase characters in `str` to the corresponding lowercase characters and leaves all other characters unchanged.

Examples

Convert Character Vector to Lowercase

```
lower('Hello, World.')
```

```
ans =
'hello, world.'
```

Convert String Array to Lowercase

Starting in R2017a, you can create string arrays using double quotes.

Convert a string array to contain lowercase characters.

```
str = ["The SOONER,";"the BETTER."]
```

```
str = 2x1 string array
    "The SOONER,"
    "the BETTER."
```

```
newStr = lower(str)
```

```
newStr = 2x1 string array
    "the sooner,"
    "the better."
```

Input Arguments

str — Input array
string array | character array | cell array of character vectors

Input array, specified as a string array, character array, or cell array of character vectors.

Tips

For character arrays, the `lower` function supports only the character sets:

- PC: Windows Latin-1
- Other: ISO® Latin-1 (ISO 8859-1)

For string arrays, the `lower` function supports Unicode.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Input array must be a string scalar or a character array.
- Input values must be in the range 0–127.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`char` | `isstrprop` | `reverse` | `string` | `upper`

Topics

“Text in String and Character Arrays”
“Create String Arrays”
“Analyze Text Data with String Arrays”
“Search and Replace Text”

Introduced before R2006a

ls

List folder contents

Syntax

```
ls
ls name
list = ls( ___ )
```

Description

`ls` lists the contents of the current folder.

`ls name` lists the files and folders in the current folder that match the specified name.

`list = ls( ___ )` returns the names of all the files and folders in the current folder that match the specified `name`. You can specify `list` with any of the arguments in the previous syntaxes.

Examples

List Files and Folders

List all the files and folders with names that contain `my`.

```
ls *my*
```

```
my_class.m     my_function.m  my_script.m    my_text.txt
```

List all the files and folders with a `.m` extension.

```
ls *.m
```

```
my_class.m       my_function.m    my_script.m      test_function.m
```

Save List of Files and Folders

Save a list of the names of all the files and folders in the current folder to the variable `MyList`.

```
MyList = ls;
```

Input Arguments

`name` — Name of file or folder
character vector | string scalar

Name of file or folder, specified as a character vector or string scalar. Use the `'*'` wildcard to match patterns. For example, `ls *.m` lists all files and folders with a `.m` extension, and `ls m*` lists all files and folders with names that begin with the letter `m`.

Data Types: `char` | `string`

Output Arguments

`list` — List of files and folders
character array

List of files and folders, specified as a character array in one of these formats.

- **UNIX platforms** – `list` is a character vector of names separated by **tab** and **space** characters.
- **Microsoft Windows platforms** – `list` is an `m`-by-`n` character array of names. `m` is the number of names and `n` is the number of characters in the longest name. MATLAB pads names shorter than `n` characters with space characters.

Tips

- To further modify the results of the `ls` command on UNIX platforms, you can add any flags that the operating system supports. For example, `ls -c` displays files by

timestamp and `ls -d` displays only directories. For more information, see the UNIX `ls` documentation.

Alternatives

Use the `dir` command to return file attributes for each file and folder in the output argument.

You can also view files and folders in the Current Folder browser by issuing the `filebrowser` command.

See Also

`dir` | `pwd`

Introduced before R2006a

lscov

Least-squares solution in presence of known covariance

Syntax

```
x = lscov(A,B)
x = lscov(A,B,w)
x = lscov(A,B,V)
x = lscov(A,B,V,alg)
[x,stdx] = lscov(...)
[x,stdx,mse] = lscov(...)
[x,stdx,mse,S] = lscov(...)
```

Description

`x = lscov(A,B)` returns the ordinary least squares solution to the linear system of equations `A*x = B`, i.e., `x` is the n-by-1 vector that minimizes the sum of squared errors `(B - A*x)'*(B - A*x)`, where `A` is m-by-n, and `B` is m-by-1. `B` can also be an m-by-k matrix, and `lscov` returns one solution for each column of `B`. When `rank(A) < n`, `lscov` sets the maximum possible number of elements of `x` to zero to obtain a "basic solution".

`x = lscov(A,B,w)`, where `w` is a vector length m of real positive weights, returns the weighted least squares solution to the linear system `A*x = B`, that is, `x` minimizes `(B - A*x)'*diag(w)*(B - A*x)`. `w` typically contains either counts or inverse variances.

`x = lscov(A,B,V)`, where `V` is an m-by-m real symmetric positive definite matrix, returns the generalized least squares solution to the linear system `A*x = B` with covariance matrix proportional to `V`, that is, `x` minimizes `(B - A*x)'*inv(V)*(B - A*x)`.

More generally, `V` can be positive semidefinite, and `lscov` returns `x` that minimizes `e'*e`, subject to `A*x + T*e = B`, where the minimization is over `x` and `e`, and `T*T' = V`. When `V` is semidefinite, this problem has a solution only if `B` is consistent with `A` and `V` (that is, `B` is in the column space of `[A T]`), otherwise `lscov` returns an error.

By default, `lscov` computes the Cholesky decomposition of `V` and, in effect, inverts that factor to transform the problem into ordinary least squares. However, if `lscov` determines that `V` is semidefinite, it uses an orthogonal decomposition algorithm that avoids inverting `V`.

`x = lscov(A,B,V,alg)` specifies the algorithm used to compute `x` when `V` is a matrix. `alg` can have the following values:

- `'chol'` uses the Cholesky decomposition of `V`.
- `'orth'` uses orthogonal decompositions, and is more appropriate when `V` is ill-conditioned or singular, but is computationally more expensive.

`[x,stdx] = lscov(...)` returns the estimated standard errors of `x`. When `A` is rank deficient, `stdx` contains zeros in the elements corresponding to the necessarily zero elements of `x`.

`[x,stdx,mse] = lscov(...)` returns the mean squared error. If `B` is assumed to have covariance matrix σ^2`V` (or (σ^2)×`diag`(1./`W`)), then `mse` is an estimate of σ^2.

`[x,stdx,mse,S] = lscov(...)` returns the estimated covariance matrix of `x`. When `A` is rank deficient, `S` contains zeros in the rows and columns corresponding to the necessarily zero elements of `x`. `lscov` cannot return `S` if it is called with multiple right-hand sides, that is, if `size(B,2) > 1`.

The standard formulas for these quantities, when `A` and `V` are full rank, are

- `x = inv(A'*inv(V)*A)*A'*inv(V)*B`
- `mse = B'*(inv(V) - inv(V)*A*inv(A'*inv(V)*A)*A'*inv(V))*B./(m-n)`
- `S = inv(A'*inv(V)*A)*mse`
- `stdx = sqrt(diag(S))`

However, `lscov` uses methods that are faster and more stable, and are applicable to rank deficient cases.

`lscov` assumes that the covariance matrix of `B` is known only up to a scale factor. `mse` is an estimate of that unknown scale factor, and `lscov` scales the outputs `S` and `stdx` appropriately. However, if `V` is known to be exactly the covariance matrix of `B`, then that scaling is unnecessary. To get the appropriate estimates in this case, you should rescale `S` and `stdx` by `1/mse` and `sqrt(1/mse)`, respectively.

Examples

Example 1 — Computing Ordinary Least Squares

The MATLAB backslash operator (\) enables you to perform linear regression by computing ordinary least-squares (OLS) estimates of the regression coefficients. You can also use `lscov` to compute the same OLS estimates. By using `lscov`, you can also compute estimates of the standard errors for those coefficients, and an estimate of the standard deviation of the regression error term:

```
x1 = [.2 .5 .6 .8 1.0 1.1]';
x2 = [.1 .3 .4 .9 1.1 1.4]';
X = [ones(size(x1)) x1 x2];
y = [.17 .26 .28 .23 .27 .34]';

a = X\y
a =
    0.1203
    0.3284
   -0.1312

[b,se_b,mse] = lscov(X,y)
b =
    0.1203
    0.3284
   -0.1312
se_b =
    0.0643
    0.2267
    0.1488
mse =
    0.0015
```

Example 2 — Computing Weighted Least Squares

Use `lscov` to compute a weighted least-squares (WLS) fit by providing a vector of relative observation weights. For example, you might want to downweight the influence of an unreliable observation on the fit:

```
w = [1 1 1 1 1 .1]';

[bw,sew_b,msew] = lscov(X,y,w)
```

```
bw =
    0.1046
    0.4614
   -0.2621
sew_b =
    0.0309
    0.1152
    0.0814
msew =
  3.4741e-004
```

Example 3 — Computing General Least Squares

Use `lscov` to compute a general least-squares (GLS) fit by providing an observation covariance matrix. For example, your data may not be independent:

```
V = .2*ones(length(x1)) + .8*diag(ones(size(x1)));

[bg,sew_b,mseg] = lscov(X,y,V)
bg =
    0.1203
    0.3284
   -0.1312
sew_b =
    0.0672
    0.2267
    0.1488
mseg =
    0.0019
```

Example 4 — Estimating the Coefficient Covariance Matrix

Compute an estimate of the coefficient covariance matrix for either OLS, WLS, or GLS fits. The coefficient standard errors are equal to the square roots of the values on the diagonal of this covariance matrix:

```
[b,se_b,mse,S] = lscov(X,y);

S
S =
    0.0041   -0.0130    0.0075
   -0.0130    0.0514   -0.0328
    0.0075   -0.0328    0.0221
```

```
[se_b sqrt(diag(S))]
ans =
    0.0643    0.0643
    0.2267    0.2267
    0.1488    0.1488
```

Algorithms

The vector `x` minimizes the quantity `(A*x-B)'*inv(V)*(A*x-B)`. The classical linear algebra solution to this problem is

```
x = inv(A'*inv(V)*A)*A'*inv(V)*B
```

but the `lscov` function instead computes the QR decomposition of `A` and then modifies `Q` by `V`.

References

[1] Strang, G., *Introduction to Applied Mathematics*, Wellesley-Cambridge, 1986, p. 398.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If A is rank deficient or V is a matrix and V is rank deficient, then you must enable support for variable-size arrays.
- Code generation does not support sparse matrix inputs for this function.

See Also

`lsqnonneg` | `mldivide` | `mrdivide` | `qr`

Introduced before R2006a

lsqminnorm

Minimum norm least-squares solution to linear equation

Syntax

```
X = lsqminnorm(A,B)
X = lsqminnorm(A,B,tol)
X = lsqminnorm( ___ ,rankWarn)
```

Description

`X = lsqminnorm(A,B)` returns an array `X` that solves the linear equation $AX = B$ and minimizes the value of `norm(A*X-B)`. If several solutions exist to this problem, then `lsqminnorm` returns the solution that minimizes `norm(X)`.

`X = lsqminnorm(A,B,tol)` additionally specifies the tolerance that `lsqminnorm` uses to determine the rank of `A`.

`X = lsqminnorm( ___ ,rankWarn)` specifies an optional flag to display a warning if `A` has low rank. You can use any of the input argument combinations in previous syntaxes. `rankWarn` can be `'nowarn'` (default) or `'warn'`.

Examples

Solve Linear System with Infinite Solutions

Solve a linear system that has infinitely many solutions with backslash (\) and `lsqminnorm`. Compare the results using the 2-norms of the solutions.

When infinite solutions exist to $\mathrm{Ax} = b$, each of them minimizes $\|\mathrm{Ax} - b\|$. The backslash command (\) computes one such solution, but this solution typically does not minimize $\|x\|$. The solution computed by `lsqminnorm` minimizes not only `norm(A*x-b)`, but also `norm(x)`.

Consider a simple linear system with one equation and two unknowns, $2x_1 + 3x_2 = 8$. This system is *underdetermined* since there are fewer equations than unknowns. Solve the equation using both backslash and `lsqminnorm`.

```
A = [2 3];
b = 8;
x_a = A\b
```

```
x_a = 2×1

         0
    2.6667
```

```
x_b = lsqminnorm(A,b)
```

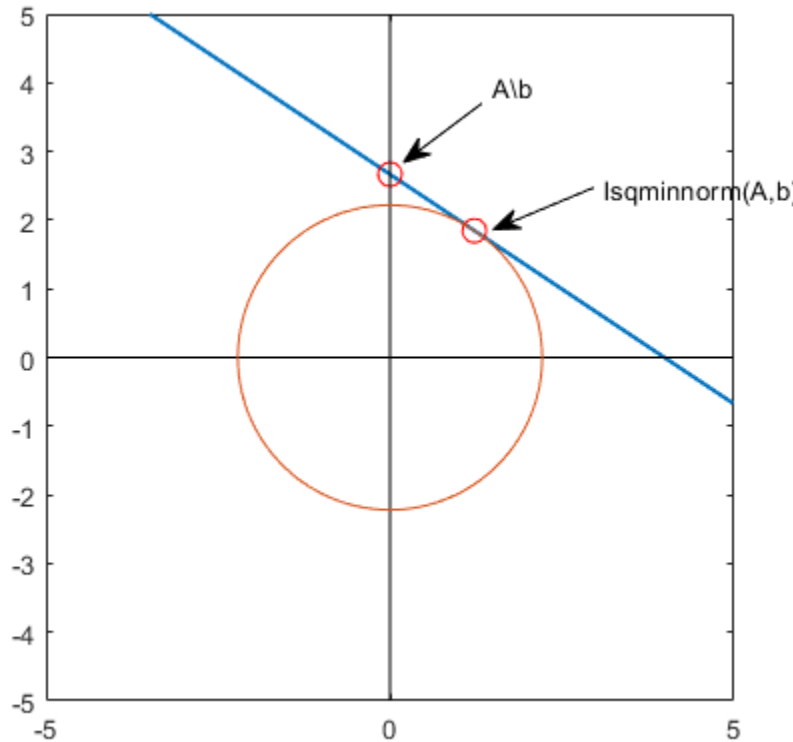
```
x_b = 2×1

    1.2308
    1.8462
```

The two methods obtain different solutions because backslash only aims to minimize `norm(A*x-b)`, whereas `lsqminnorm` also aims to minimize `norm(x)`. Calculate these norms and put the results in a table for easy comparison.

```
s1 = {'Backslash'; 'lsqminnorm'};
s2 = {'norm_Ax_minus_b','norm_x'};
T = table([norm(A*x_a-b); norm(A*x_b-b)],[norm(x_a); norm(x_b)],'RowNames',s1,'Variable
```

```
T=2×2 table
                   norm_Ax_minus_b    norm_x
                   _______________    ______

    Backslash                    0    2.6667
    lsqminnorm       8.8818e-16       2.2188
```

This figure illustrates the situation and shows which solutions each of the methods return. The blue line represents the infinite number of solutions to the equation $x_2 = -\frac{2}{3}x_1 + \frac{8}{3}$. The orange circle represents the minimum distance from the origin to the line of solutions, and the solution returned by `lsqminnorm` lies exactly at the tangent point between the line and circle, indicating it is the solution that is closest to the origin.



Specify Tolerance to Reduce Impact of Noisy Data

Show how specifying a tolerance for the rank computation in `lsqminnorm` can help define the scale of the problem so that random noise does not corrupt the solution.

Create a low-rank matrix of rank 5 and a right-hand side vector `b`.

```
rng default % for reproducibility
U = randn(200,5);
V = randn(100,5);
A = U*V';
b = U*randn(5,1) + 1e-4*randn(200,1);
```

Solve the linear system Ax = *b* using `lsqminnorm`. Compute the norms of `A*x-b` and `x` to check the quality of the solution.

```
x = lsqminnorm(A,b);
norm(A*x-b)
```

```
ans = 0.0014
```

```
norm(x)
```

```
ans = 0.1741
```

Now add a small amount of noise to the matrix `A` and solve the linear system again. The noise affects the solution vector `x` of the linear system disproportionately.

```
Anoise = A + 1e-12*randn(200,100);
xnoise = lsqminnorm(Anoise,b);
norm(Anoise*xnoise - b)
```

```
ans = 0.0010
```

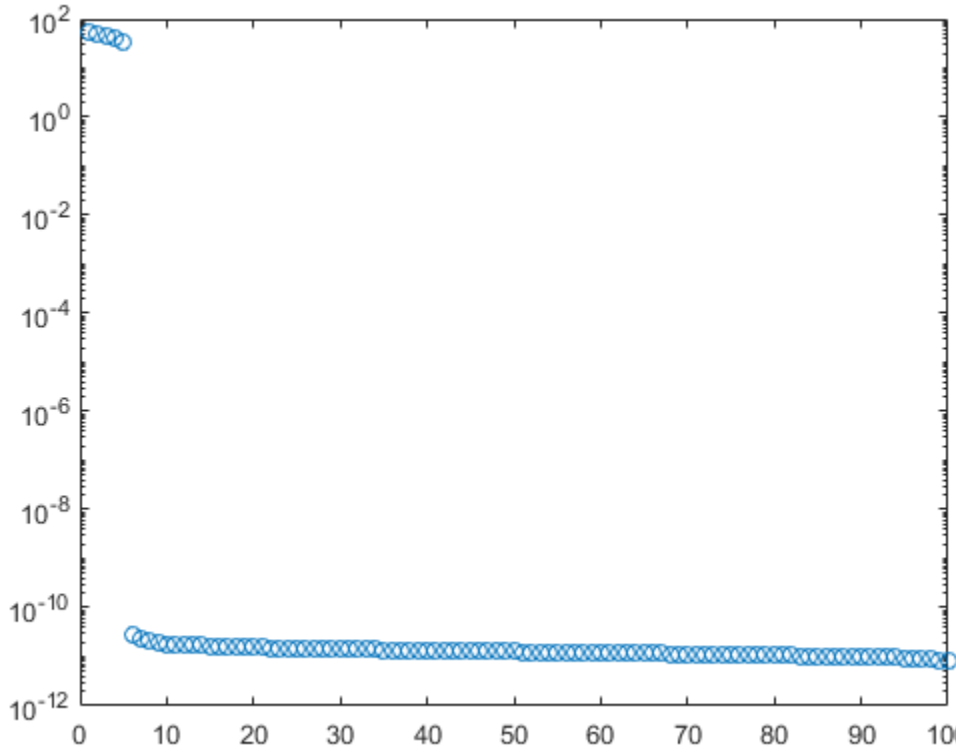
```
norm(xnoise)
```

```
ans = 1.1216e+08
```

The reason for the big difference in the solutions is that the noise affects the low-rank approximation of `A`. In other words, `lsqminnorm` is treating small values on the diagonal of the `R` matrix in the QR decomposition of `A` as being more important than they are. Ideally, these small values on the diagonal of `R` should be treated as zeros.

Plot the diagonal elements of the `R` matrix in the QR decomposition of `Anoise`. A large number of the diagonal elements are on the order of 1e-10.

```
[Q,R,p] = qr(Anoise,0);
semilogy(abs(diag(R)),'o')
```



The solution to this issue is to increase the tolerance used by `lsqminnorm` so that a low-rank approximation of `Anoise` with error less than 1e-8 is used in the calculation. This makes the result much less susceptible to the noise. The solution using a tolerance is very close to the original solution `x`.

```
xnoise = lsqminnorm(Anoise, b, 1e-8);
norm(Anoise*xnoise - b)
```

```
ans = 0.0014
```

```
norm(xnoise)
```

```
ans = 0.1741
```

```
norm(x - xnoise)
```

```
ans = 1.0804e-14
```

Toggle Warnings for Low-Rank Matrices

Solve a linear system involving a low-rank coefficient matrix with warnings turned on.

Create a 3-by-3 matrix that is of rank 2. In this matrix, you can obtain the third column by adding together the first two columns.

```
A = [1 2 3; 4 5 9; 6 7 13]
```

```
A = 3×3

     1     2     3
     4     5     9
     6     7    13
```

Find the minimum norm least-squares solution to the problem Ax = *b*, where *b* is equal to the second column in *A*. Specify the `'warn'` flag for `lsqminnorm` to display a warning if it detects that A is of low rank.

```
b = A(:,2);
x = lsqminnorm(A,b,'warn')
```

```
Warning: Rank deficient, rank = 2, tol =  1.072041e-14.

x = 3×1

   -0.3333
    0.6667
    0.3333
```

Input Arguments

A — Coefficient matrix
matrix

Coefficient matrix. The coefficient matrix appears in the system of linear equations on the left as $Ax = B$. The coefficient matrix can be full or sparse.

Data Types: `single` | `double`
Complex Number Support: Yes

`B` — Input array
vector | matrix

Input array, specified as a vector or matrix. `B` appears in the system of linear equations on the right as $Ax = B$. If `B` is a matrix, then each column in the matrix represents a different vector for the right-hand side.

Data Types: `single` | `double`
Complex Number Support: Yes

`tol` — Rank tolerance
nonnegative scalar

Rank tolerance, specified as a nonnegative scalar. Specifying the tolerance can help prevent the solution from being susceptible to random noise in the coefficient matrix. By default, `lsqminnorm` computes `tol` based on the QR decomposition of `A`.

`lsqminnorm` computes the rank of `A` as the number of diagonal elements in the `R` matrix of the QR decomposition `[Q,R,p] = qr(A,0)` with absolute value larger than `tol`. If the rank of `A` is `k`, then the function forms a low-rank approximation of `A` by multiplying the first `k` columns of `Q` by the first `k` rows of `R`. Changing the tolerance affects this low-rank approximation of `A`.

Example: `X = lsqminnorm(A,B,1e-2)`

Data Types: `double`

`rankWarn` — Warning toggle for low-rank matrices
`'nowarn'` (default) | `'warn'`

Warning toggle for low-rank matrices, specified as either `'nowarn'` or `'warn'`. Specify `'warn'` to indicate that `lsqminnorm` should produce warnings if the coefficient matrix `A` is rank deficient.

Example: `X = lsqminnorm(A,B,'warn')`

Tips

- The minimum-norm solution computed by `lsqminnorm` is of particular interest when several solutions exist. The equation $Ax = b$ has many solutions whenever `A` is underdetermined (fewer rows than columns) or of low rank.
- `lsqminnorm(A,B,tol)` is typically more efficient than `pinv(A,tol)*B` for computing minimum norm least-squares solutions to linear systems. `lsqminnorm` uses the complete orthogonal decomposition (COD) to find a low-rank approximation of `A`, while `pinv` uses the singular value decomposition (SVD). Therefore, the results of `pinv` and `lsqminnorm` do not match exactly.
- For sparse matrices, `lsqminnorm` uses a different algorithm than for dense matrices, and therefore can produce different results.

See Also

`decomposition` | `mldivide` | `pinv`

Topics

"Systems of Linear Equations"

Introduced in R2017b

lsqnonneg

Solve nonnegative linear least-squares problem

Syntax

```
x = lsqnonneg(C,d)
x = lsqnonneg(C,d,options)
x = lsqnonneg(problem)
[x,resnorm,residual] = lsqnonneg( ___ )
[x,resnorm,residual,exitflag,output] = lsqnonneg( ___ )
[x,resnorm,residual,exitflag,output,lambda] = lsqnonneg( ___ )
```

Description

Solve nonnegative least-squares curve fitting problems of the form

$$\min_x \|C \cdot x - d\|_2^2, \text{ where } x \geq 0.$$

`x = lsqnonneg(C,d)` returns the vector `x` that minimizes `norm(C*x-d)` subject to `x ≥ 0`. Arguments `C` and `d` must be real.

`x = lsqnonneg(C,d,options)` minimizes with the optimization options specified in the structure `options`. Use `optimset` to set these options.

`x = lsqnonneg(problem)` finds the minimum for `problem`, where `problem` is a structure.

`[x,resnorm,residual] = lsqnonneg( ___ )`, for any previous syntax, additionally returns the value of the squared 2-norm of the residual, `norm(C*x-d)^2`, and returns the residual `d-C*x`.

`[x,resnorm,residual,exitflag,output] = lsqnonneg( ___ )` additionally returns a value `exitflag` that describes the exit condition of `lsqnonneg`, and a structure `output` with information about the optimization process.

`[x,resnorm,residual,exitflag,output,lambda] = lsqnonneg( ___ )` additionally returns the Lagrange multiplier vector `lambda`.

Examples

Nonnegative Linear Least Squares

Compute a nonnegative solution to a linear least-squares problem, and compare the result to the solution of an unconstrained problem.

Prepare a `C` matrix and `d` vector for the problem $\min||Cx - d||$.

```
C = [0.0372    0.2869
     0.6861    0.7071
     0.6233    0.6245
     0.6344    0.6170];

d = [0.8587
     0.1781
     0.0747
     0.8405];
```

Compute the constrained and unconstrained solutions.

```
x = lsqnonneg(C,d)
```

```
x = 2×1

         0
    0.6929
```

```
xunc = C\d
```

```
xunc = 2×1

   -2.5627
    3.1108
```

All entries in `x` are nonnegative, but some entries in `xunc` are negative.

Compute the norms of the residuals for the two solutions.

```
constrained_norm = norm(C*x - d)
```

```
constrained_norm = 0.9118
```

```
unconstrained_norm = norm(C*xunc - d)
```

```
unconstrained_norm = 0.6674
```

The unconstrained solution has a smaller residual norm because constraints can only increase a residual norm.

Nonnegative Least Squares with Nondefault Options

Set the `Display` option to `'final'` to see output when `lsqnonneg` finishes.

Create the options.

```
options = optimset('Display','final');
```

Prepare a `C` matrix and `d` vector for the problem $\min||Cx - d||$.

```
C = [0.0372    0.2869
     0.6861    0.7071
     0.6233    0.6245
     0.6344    0.6170];

d = [0.8587
     0.1781
     0.0747
     0.8405];
```

Call `lsqnonneg` with the options structure.

```
x = lsqnonneg(C,d,options);
```

```
Optimization terminated.
```

Obtain Residuals from Nonnegative Least Squares

Call `lsqnonneg` with outputs to obtain the solution, residual norm, and residual vector.

Prepare a C matrix and d vector for the problem $\min||Cx - d||$.

```
C = [0.0372    0.2869
     0.6861    0.7071
     0.6233    0.6245
     0.6344    0.6170];

d = [0.8587
     0.1781
     0.0747
     0.8405];
```

Obtain the solution and residual information.

```
 [x,resnorm,residual] = lsqnonneg(C,d)
```

```
x = 2×1

         0
    0.6929


resnorm = 0.8315

residual = 4×1

    0.6599
   -0.3119
   -0.3580
    0.4130
```

Verify that the returned residual norm is the square of the norm of the returned residual vector.

```
 norm(residual)^2
```

```
ans = 0.8315
```

Inspect the Result of Nonnegative Least Squares

Request all output arguments to examine the solution and solution process after `lsqnonneg` finishes.

Prepare a C matrix and d vector for the problem $\min||Cx - d||$.

```
C = [0.0372    0.2869
     0.6861    0.7071
     0.6233    0.6245
     0.6344    0.6170];

d = [0.8587
     0.1781
     0.0747
     0.8405];
```

Solve the problem, requesting all output arguments.

```
[x,resnorm,residual,exitflag,output,lambda] = lsqnonneg(C,d)
```

```
x = 2×1

         0
    0.6929


resnorm = 0.8315

residual = 4×1

    0.6599
   -0.3119
   -0.3580
    0.4130


exitflag = 1

output = struct with fields:
    iterations: 1
     algorithm: 'active-set'
       message: 'Optimization terminated.'


lambda = 2×1

   -0.1506
   -0.0000
```

`exitflag` is `1`, indicating a correct solution.

`x(1) = 0`, and the corresponding `lambda(1)` ≠ `0`, showing the correct duality. Similarly, `x(2) > 0`, and the corresponding `lambda(2) = 0`.

Input Arguments

`C` — Linear multiplier
real matrix

Linear multiplier, specified as a real matrix. Represents the variable C in the problem

$$\min_x \|C \cdot x - d\|_2^2, \text{ where } x \geq 0.$$

For compatibility, the number of rows of `C` must equal the length of `d`.

Example: `C = [1,2;3,-1;-4,4]`

Data Types: `double`

`d` — Additive term
real vector

Additive term, specified as a real vector. Represents the variable d in the problem

$$\min_x \|C \cdot x - d\|_2^2, \text{ where } x \geq 0.$$

For compatibility, the length of `d` must equal the number of rows of `C`.

Example: `d = [1;-6;5]`

Data Types: `double`

`options` — Optimization options
structure such as `optimset` returns

Optimization options, specified as a structure such as `optimset` returns. You can use `optimset` to set or change the values of these fields in the options structure. See “Set Options” for detailed information.

`Display`	Level of display: • `'notify'` (default) displays output only if the function does not converge. • `'off'` or `'none'` displays no output. • `'final'` displays just the final output.
`TolX`	Termination tolerance on `x`, a positive scalar. The default is `10*eps*norm(C,1)*length(C)`. See “Tolerances and Stopping Criteria”.

Example: `options = optimset('Display','final')`

Data Types: `struct`

`problem` — Problem structure
structure

Problem structure, specified as a structure with the following fields.

Field Name	Entry
`C`	Real matrix
`d`	Real vector
`solver`	`'lsqnonneg'`
`options`	Options structure such as returned by `optimset`

The simplest way to obtain a `problem` structure is to export the problem from the Optimization app.

Data Types: `struct`

Output Arguments

`x` — Solution
real vector

Solution, returned as a real vector. The length of `x` is the same as the length of `d`.

`resnorm` — Squared residual norm
nonnegative scalar

Squared residual norm, returned as a nonnegative scalar. Equal to `norm(C*x-d)^2`.

`residual` — Residual
real vector

Residual, returned as a real vector. The residual is `d - C*x`.

`exitflag` — Reason `lsqnonneg` stopped
integer

Reason `lsqnonneg` stopped, returned as an integer.

`1`	Function converged to a solution `x`.
`0`	Number of iterations exceeded `options.MaxIter`.

`output` — Information about the optimization process
structure

Information about the optimization process, returned as a structure with fields:

`iterations`	Number of iterations taken
`algorithm`	`'active-set'`
`message`	Exit message

`lambda` — Lagrange multipliers
real vector

Lagrange multipliers, returned as a real vector. The entries satisfy the complementarity condition `x'*lambda = 0`. This means `lambda(i) < 0` when `x(i)` is approximately `0`, and `lambda(i)` is approximately `0` when `x(i) > 0`.

Algorithms

`lsqnonneg` uses the algorithm described in [1]. The algorithm starts with a set of possible basis vectors and computes the associated dual vector `lambda`. It then selects the basis vector corresponding to the maximum value in `lambda` to swap it out of the basis in exchange for another possible candidate. This continues until `lambda ≤ 0`.

References

[1] Lawson, C. L. and R. J. Hanson. *Solving Least-Squares Problems*. Upper Saddle River, NJ: Prentice Hall. 1974. Chapter 23, p. 161.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For C/C++ code generation:

- You must enable support for variable-size arrays.
- The exit message in the output structure is not translated.
- Code generation does not support sparse matrix inputs for this function.

See Also

`mldivide` | `optimset`

Introduced before R2006a

lsqr

LSQR method

Syntax

```
x = lsqr(A,b)
lsqr(A,b,tol)
lsqr(A,b,tol,maxit)
lsqr(A,b,tol,maxit,M)
lsqr(A,b,tol,maxit,M1,M2)
lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag] = lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag,relres] = lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag,relres,iter] = lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag,relres,iter,resvec] = lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag,relres,iter,resvec,lsvec] = lsqr(A,b,tol,maxit,M1,M2,x0)
```

Description

`x = lsqr(A,b)` attempts to solve the system of linear equations `A*x=b` for `x` if `A` is consistent, otherwise it attempts to solve the least squares solution `x` that minimizes `norm(b-A*x)`. The `m`-by-`n` coefficient matrix `A` need not be square but it should be large and sparse. The column vector `b` must have length `m`. You can specify `A` as a function handle, `afun`, such that `afun(x,'notransp')` returns `A*x` and `afun(x,'transp')` returns `A'*x`.

“Parameterizing Functions” explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `lsqr` converges, a message to that effect is displayed. If `lsqr` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`lsqr(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `lsqr` uses the default, `1e-6`.

`lsqr(A,b,tol,maxit)` specifies the maximum number of iterations.

`lsqr(A,b,tol,maxit,M)` and `lsqr(A,b,tol,maxit,M1,M2)` use `n`-by-`n` preconditioner `M` or `M = M1*M2` and effectively solve the system `A*inv(M)*y = b` for `y`, where `y = M*x`. If `M` is `[]` then `lsqr` applies no preconditioner. `M` can be a function `mfun` such that `mfun(x,'notransp')` returns `M\x` and `mfun(x,'transp')` returns `M'\x`.

`lsqr(A,b,tol,maxit,M1,M2,x0)` specifies the `n`-by-`1` initial guess. If `x0` is `[]`, then `lsqr` uses the default, an all zero vector.

`[x,flag] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns a convergence flag.

Flag	Convergence
`0`	`lsqr` converged to the desired tolerance `tol` within `maxit` iterations.
`1`	`lsqr` iterated `maxit` times but did not converge.
`2`	Preconditioner `M` was ill-conditioned.
`3`	`lsqr` stagnated. (Two consecutive iterates were the same.)
`4`	One of the scalar quantities calculated during `lsqr` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if you specify the `flag` output.

`[x,flag,relres] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns an estimate of the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns a vector of the residual norm estimates at each iteration, including `norm(b-A*x0)`.

`[x,flag,relres,iter,resvec,lsvec] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns a vector of estimates of the scaled normal equations residual at each iteration: `norm((A*inv(M))'*(B-A*X))/norm(A*inv(M),'fro')`. Note that the estimate of `norm(A*inv(M),'fro')` changes, and hopefully improves, at each iteration.

Examples

Example 1

```
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);

x = lsqr(A,b,tol,maxit,M1,M2);
```

displays the following message:

```
lsqr converged at iteration 11 to a solution with relative
residual 3.5e-009
```

Example 2

This example replaces the matrix `A` in Example 1 with a handle to a matrix-vector product function `afun`. The example is contained in a function `run_lsqr` that

- Calls `lsqr` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_lsqr` are available to `afun`.

The following shows the code for `run_lsqr`:

```
function x1 = run_lsqr
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x1 = lsqr(@afun,b,tol,maxit,M1,M2);
```

```
    function y = afun(x,transp_flag)
       if strcmp(transp_flag,'transp')      % y = A'*x
          y = 4 * x;
          y(1:n-1) = y(1:n-1) - 2 * x(2:n);
          y(2:n) = y(2:n) - x(1:n-1);
       elseif strcmp(transp_flag,'notransp') % y = A*x
          y = 4 * x;
          y(2:n) = y(2:n) - 2 * x(1:n-1);
          y(1:n-1) = y(1:n-1) - x(2:n);
       end
    end
end
```

When you enter

```
x1=run_lsqr;
```

MATLAB software displays the message

```
lsqr converged at iteration 11 to a solution with relative
residual 3.5e-009
```

References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Paige, C. C. and M. A. Saunders, "LSQR: An Algorithm for Sparse Linear Equations And Sparse Least Squares," *ACM Trans. Math. Soft.*, Vol.8, 1982, pp. 43-71.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- `b` must be a nonsparse column vector.
- Output of least squares estimates as a sixth return value is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- If `M1` is a function, then it is applied independently to each row.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bicg` | `bicgstab` | `cgs` | `gmres` | `minres` | `norm` | `pcg` | `qmr` | `symmlq`

Topics

“Create Function Handle”

Introduced before R2006a

lt, <

Determine less than

Syntax

```
A < B
lt(A,B)
```

Description

`A < B` returns an array with elements set to logical 1 (`true`) where `A` is less than `B`; otherwise, the element is logical `0` (`false`). The test compares only the real part of numeric arrays. `lt` returns logical `0` (`false`) where `A` or `B` have NaN or undefined categorical elements.

`lt(A,B)` is an alternate way to execute `A < B`, but is rarely used. It enables operator overloading for classes.

Examples

Test Vector Elements

Determine if vector elements are less than a given value.

Create a numeric vector.

```
A = [1 12 18 7 9 11 2 15];
```

Test the vector for elements that are less than 12.

```
A < 12
```

```
ans = 1x8 logical array
```

```
   1   0   0   1   1   1   1   0
```

The result is a vector with values of logical `1` (`true`) where the elements of `A` satisfy the expression.

Use the vector of logical values as an index to view the values in `A` that are less than `12`.

```
A(A < 12)
```

```
ans = 1×5

     1     7     9    11     2
```

The result is a subset of the elements in `A`.

Replace Elements of Matrix

Create a matrix.

```
A = magic(4)
```

```
A = 4×4

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Replace all values less than `9` with the value `10`.

```
A(A < 9) = 10
```

```
A = 4×4

    16    10    10    13
    10    11    10    10
     9    10    10    12
    10    14    15    10
```

The result is a new matrix whose smallest element is 9.

Compare Values in Categorical Array

Create an ordinal categorical array.

```
A = categorical({'large' 'medium' 'small'; 'medium' ...
'small' 'large'},{'small' 'medium' 'large'},'Ordinal',1)
```

```
A = 2x3 categorical array
     large       medium      small
     medium      small       large
```

The array has three categories: `'small'`, `'medium'`, and `'large'`.

Find all values less than the category `'medium'`.

```
A < 'medium'
```

```
ans = 2x3 logical array

   0   0   1
   0   1   0
```

A value of logical `1` (`true`) indicates a value less than the category `'medium'`.

Compare the rows of `A`.

```
A(1,:) < A(2,:)
```

```
ans = 1x3 logical array

   0   0   1
```

The function returns logical `1` (`true`) where the first row has a category value less than the second row.

Test Complex Numbers

Create a vector of complex numbers.

```
A = [1+i 2-2i 1+3i 1-2i 5-i];
```

Find the values that are less than 3.

```
A(A < 3)
```

```
ans = 1×4 complex

   1.0000 + 1.0000i   2.0000 - 2.0000i   1.0000 + 3.0000i   1.0000 - 2.0000i
```

`lt` compares only the real part of the elements in `A`.

Use `abs` to find which elements are within a radius of 3 from the origin.

```
A(abs(A) < 3)
```

```
ans = 1×3 complex

   1.0000 + 1.0000i   2.0000 - 2.0000i   1.0000 - 2.0000i
```

The result has one less element. The element `1.0000 + 3.0000i` is not within a radius of 3 from the origin.

Compare Dates

Create a vector of dates.

```
A = datetime([2014,05,01;2014,05,31])
```

```
A = 2x1 datetime array
   01-May-2014
   31-May-2014
```

Find the dates that occur before May 10, 2014.

```
A(A < '2014-05-10')
```

```
ans = datetime
   01-May-2014
```

Input Arguments

A, B — Operands
scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Numeric or string inputs A and B must either be the same size or have sizes that are compatible (for example, A is an M-by-N matrix and B is a scalar or 1-by-N row vector). For more information, see “Compatible Array Sizes for Basic Operations”.

You can compare numeric inputs of any type, and the comparison does not suffer loss of precision due to type conversion.

If A and B are categorical, datetime, or duration arrays, then they must be the same size unless one is a scalar.

- If one input is an ordinal categorical array, the other input can be an ordinal categorical array, a cell array of character vectors, or a single character vector. A single character vector expands into a cell array of character vectors of the same size as the other input. If both inputs are ordinal categorical arrays, they must have the same sets of categories, including their order. See “Compare Categorical Array Elements” for more details.
- If one input is a datetime array, the other input can be a datetime array, a character vector, or a cell array of character vectors.
- If one input is a duration array, the other input can be a duration array or a numeric array. The operator treats each numeric value as a number of standard 24-hour days.
- If one input is a string array, the other input can be a string array, a character vector, or a cell array of character vectors. The corresponding elements of A and B are compared lexicographically.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char | string | categorical | datetime | duration
Complex Number Support: Yes

Compatibility Considerations

Implicit expansion change affects arguments for operators

Behavior changed in R2016b

Starting in R2016b with the addition of implicit expansion, some combinations of arguments for basic operations that previously returned errors now produce results. For example, you previously could not add a row and a column vector, but those operands are now valid for addition. In other words, an expression like `[1 2] + [1; 2]` previously returned a size mismatch error, but now it executes.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a `try/catch` block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see "Compatible Array Sizes for Basic Operations".

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see "Tall Arrays".

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`eq` | `ge` | `gt` | `le` | `ne`

Topics

“Array Comparison with Relational Operators”
“Ordinal Categorical Arrays”
“MATLAB Operators and Special Characters”

Introduced before R2006a

lu

LU matrix factorization

Syntax

```
[L,U] = lu(A)
[L,U,P] = lu(A)
[L,U,P] = lu(A,outputForm)

[L,U,P,Q] = lu(S)
[L,U,P,Q,D] = lu(S)
[ ___ ] = lu(S,thresh)
[ ___ ] = lu( ___ ,outputForm)
```

Description

`[L,U] = lu(A)` factorizes the full or sparse matrix `A` into an upper triangular matrix `U` and a permuted lower triangular matrix `L` such that `A = L*U`.

`[L,U,P] = lu(A)` also returns a permutation matrix `P` such that `A = P'*L*U`. With this syntax, `L` is unit lower triangular and `U` is upper triangular.

`[L,U,P] = lu(A,outputForm)` returns `P` in the form specified by `outputForm`. Specify `outputForm` as `'vector'` to return `P` as a permutation vector such that `A(P,:) = L*U`.

`[L,U,P,Q] = lu(S)` factorizes sparse matrix `S` into a unit lower triangular matrix `L`, an upper triangular matrix `U`, a row permutation matrix `P`, and a column permutation matrix `Q`, such that `P*S*Q = L*U`.

`[L,U,P,Q,D] = lu(S)` also returns a diagonal scaling matrix `D` such that `P*(D\S)*Q = L*U`. Typically, the row-scaling leads to a sparser and more stable factorization.

`[ ___ ] = lu(S,thresh)` specifies thresholds for the pivoting strategy employed by `lu` using any of the previous output argument combinations. Depending on the number of

output arguments specified, the default value and requirements for the `thresh` input are different. See the `thresh` argument description for details.

`[ ___ ] = lu( ___ ,outputForm)` returns `P` and `Q` in the form specified by `outputForm`. Specify `outputForm` as `'vector'` to return `P` and `Q` as permutation vectors. You can use any of the input argument combinations in previous syntaxes.

Examples

LU Factorization of Matrix

Compute the LU factorization of a matrix and examine the resulting factors. LU factorization is a way of decomposing a matrix *A* into an upper triangular matrix *U*, a lower triangular matrix *L*, and a permutation matrix *P* such that PA = LU. These matrices describe the steps needed to perform Gaussian elimination on the matrix until it is in reduced row echelon form. The *L* matrix contains all of the multipliers, and the permutation matrix *P* accounts for row interchanges.

Create a 3-by-3 matrix and calculate the LU factors.

```
A = [10 -7 0
     -3  2 6
      5 -1 5];

[L,U] = lu(A)
```

```
L = 3×3

    1.0000         0         0
   -0.3000   -0.0400    1.0000
    0.5000    1.0000         0


U = 3×3

   10.0000   -7.0000         0
         0    2.5000    5.0000
         0         0    6.2000
```

Multiply the factors to recreate A. With the two-input syntax, `lu` incorporates the permutation matrix P directly into the L factor, such that the L being returned is really `P'*L` and thus `A = L*U`.

```
L*U
```

```
ans = 3×3

   10.0000   -7.0000         0
   -3.0000    2.0000    6.0000
    5.0000   -1.0000    5.0000
```

You can specify three outputs to separate the permutation matrix from the multipliers in L.

```
[L,U,P] = lu(A)
```

```
L = 3×3

    1.0000         0         0
    0.5000    1.0000         0
   -0.3000   -0.0400    1.0000


U = 3×3

   10.0000   -7.0000         0
         0    2.5000    5.0000
         0         0    6.2000


P = 3×3

     1     0     0
     0     0     1
     0     1     0
```

```
P'*L*U
```

```
ans = 3×3

   10.0000   -7.0000         0
   -3.0000    2.0000    6.0000
```

```
    5.0000   -1.0000    5.0000
```

Solve Linear System with LU Factorization

Solve a linear system by performing an LU factorization and using the factors to simplify the problem. Compare the results with other approaches using the backslash operator and `decomposition` object.

Create a 5-by-5 magic square matrix and solve the linear system Ax = *b* with all of the elements of `b` equal to 65, the magic sum. Since 65 is the magic sum for this matrix (all of the rows and columns add to 65), the expected solution for `x` is a vector of 1s.

```
A = magic(5);
b = 65*ones(5,1);
x = A\b
```

```
x = 5×1

    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
```

For generic square matrices, the backslash operator computes the solution of the linear system using LU decomposition. LU decomposition expresses `A` as the product of triangular matrices, and linear systems involving triangular matrices are easily solved using substitution formulas.

To recreate the answer computed by backslash, compute the LU decomposition of `A`. Then, use the factors to solve two triangular linear systems:

```
y = L\(P*b);
x = U\y;
```

This approach of precomputing the matrix factors prior to solving the linear system can improve performance when many linear systems will be solved, since the factorization occurs only once and does not need to be repeated.

```
[L,U,P] = lu(A)
```

```
L = 5×5

    1.0000         0         0         0         0
    0.7391    1.0000         0         0         0
    0.4783    0.7687    1.0000         0         0
    0.1739    0.2527    0.5164    1.0000         0
    0.4348    0.4839    0.7231    0.9231    1.0000


U = 5×5

   23.0000    5.0000    7.0000   14.0000   16.0000
         0   20.3043   -4.1739   -2.3478    3.1739
         0         0   24.8608   -2.8908   -1.0921
         0         0         0   19.6512   18.9793
         0         0         0         0  -22.2222


P = 5×5

     0     1     0     0     0
     1     0     0     0     0
     0     0     0     0     1
     0     0     1     0     0
     0     0     0     1     0
```

```
y = L\(P*b);
x = U\y
```

```
x = 5×1

    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
```

The `decomposition` object also is useful to solve linear systems using specialized factorizations, since you get many of the performance benefits of precomputing the matrix factors but you do not need to know how to use the factors. Use the decomposition object with the `'lu'` type to recreate the same results.

```
dA = decomposition(A,'lu');
x = dA\b
```

```
x = 5×1

    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
```

LU Factorization of Sparse Matrix

Compute the LU factorization of a sparse matrix and verify the identity `L*U = P*S*Q`.

Create a 60-by-60 sparse adjacency matrix of the connectivity graph of the Buckminster-Fuller geodesic dome.

```
S = bucky;
```

Compute the LU factorization of `S` using the sparse matrix syntax with four outputs to return the row and column permutation matrices.

```
[L,U,P,Q] = lu(S);
```

Permute the rows and columns of `S` with `P*S*Q` and compare the result with multiplying the triangular factors `L*U`. The 1-norm of their difference is within roundoff error, indicating that `L*U = P*S*Q`.

```
e = P*S*Q - L*U;
norm(e,1)
```

```
ans = 2.4425e-15
```

Save Memory with Permutation Vectors

Compute the LU factorization of a matrix. Save memory by returning the row permutations as a vector instead of a matrix.

Create a 1000-by-1000 random matrix.

```
A = rand(1000);
```

Compute the LU factorization with the permutation information stored as a matrix P. Compare the result with the permutation information stored as a vector p. The larger the matrix, the more memory efficient it is to use a permutation vector.

```
[L1,U1,P] = lu(A);
[L2,U2,p] = lu(A,'vector');
whos P p
```

```
  Name         Size                Bytes  Class     Attributes

  P         1000x1000            8000000  double
  p            1x1000               8000  double
```

Using a permutation vector also saves on execution time in subsequent operations. For instance, you can use the previous LU factorizations to solve a linear system Ax = *b*. Although the solutions obtained from the permutation vector and permutation matrix are equivalent (up to roundoff), the solution using the permutation vector typically requires a little less time.

Reduce Fill-in of Sparse Matrix Factorization

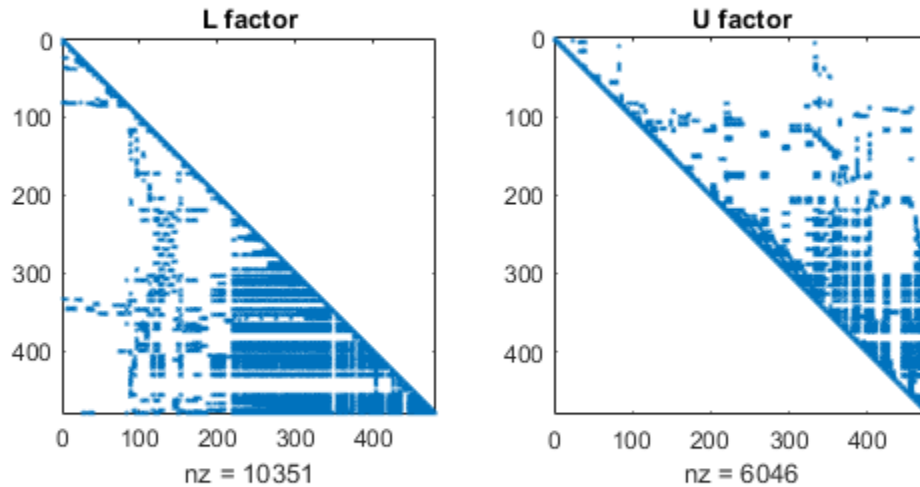
Compare the results of computing the LU factorization of a sparse matrix with and without column permutations.

Load the `west0479` matrix, which is a real-valued 479-by-479 sparse matrix.

```
load west0479
A = west0479;
```

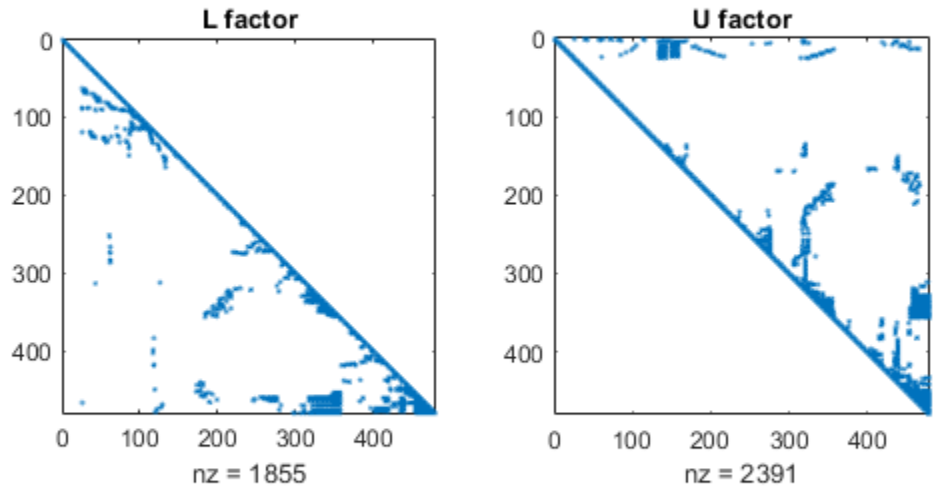
Calculate the LU factorization of A by calling `lu` with three outputs. Generate spy plots of the L and U factors.

```
[L,U,P] = lu(A);
subplot(1,2,1)
spy(L)
title('L factor')
subplot(1,2,2)
spy(U)
title('U factor')
```



Now, calculate the LU factorization of A using `lu` with four outputs, which permutes the columns of A to reduce the number of nonzeros in the factors. The resulting factors are much sparser than if column permutations are not used.

```
[L,U,P,Q] = lu(A);
subplot(1,2,1)
spy(L)
title('L factor')
subplot(1,2,2)
spy(U)
title('U factor')
```



Input Arguments

A — Input matrix
matrix

Input matrix. A can be full or sparse as well as square or rectangular in size.

Data Types: `single` | `double`
Complex Number Support: Yes

S — Sparse input matrix
sparse matrix

Sparse input matrix. `S` can be square or rectangular in size.

Data Types: `double`
Complex Number Support: Yes

`thresh` — Pivoting thresholds for sparse matrices

scalar | two-element vector

Pivoting thresholds for sparse matrices, specified as a scalar or two-element vector. Valid values are in the interval `[0 1]`. The way you specify `thresh` depends on how many outputs are specified in the call to `lu`:

- For **three outputs or less**, `thresh` must be a scalar, and the default value is `1.0`.
- For **four outputs or more**, `thresh` can be a scalar or a two element vector. The default value is `[0.1 0.001]`. If you specify `thresh` as a scalar, then that only replaces the first value in the vector.

At a high level, this input enables you to make trade-offs between accuracy and total execution time. Smaller values of `thresh` tend to lead to sparser LU factors, but the solution can become inaccurate. Larger values can lead to a more accurate solution (but not always), and usually an increase in the total work and memory usage.

`lu` selects a pivoting strategy based first on the number of output arguments and second on the properties of the matrix being factorized. In all cases, setting the threshold value(s) to `1.0` results in partial pivoting, while setting them to `0` causes the pivots to be chosen only based on the sparsity of the resulting matrix. All values of `L` have an absolute value of `1/min(thresh)` or less.

- **Three or fewer output arguments** — The algorithm selects the diagonal pivot if it satisfies the equation

  ```
  A(j,j) >= thresh * max(abs(A(j:m,j)))
  ```

  Otherwise, it selects the row that contains the element of largest absolute value.
- **Symmetric Pivoting Strategy** — If `S` is a square sparse matrix with a mostly symmetric structure and mostly nonzero diagonal, then `lu` uses a symmetric pivoting strategy. For this strategy, the algorithm selects the diagonal pivot `j` if it satisfies the inequality:

  ```
  A(i,j) >= thresh(2) * max(abs(A(j:m,j)))
  ```

  If the diagonal entry fails this test, then `lu` selects the sparsest row `i` satisfying the inequality:

```
A(i,j) >= thresh(1) * max(abs(A(j:m,j)))
```

- **Nonsymmetric Pivoting Strategy** — If `S` does not satisfy the requirements for the symmetric pivoting strategy, then `lu` uses a nonsymmetric strategy. In this case, `lu` selects the sparsest row `i` satisfying the inequality:

  ```
  A(i,j) >= thresh(1) * max(abs(A(j:m,j)))
  ```

  A value of `1.0` for `thresh(1)` results in conventional partial pivoting. Entries in `L` have an absolute value of `1/thresh(1)` or less. The second element of the `thresh` input vector is not used with the nonsymmetric strategy.

Note In some rare cases, an incorrect factorization results in `P*S*Q ≠ L*U`. If this occurs, increase `thresh` to a maximum of `1.0` (regular partial pivoting), and try again.

`outputForm` — Shape of permutation outputs

`'matrix'` (default) | `'vector'`

Shape of permutation outputs, specified as `'matrix'` or `'vector'`. This flag controls whether `lu` returns the row permutations `P` and column permutations `Q` as permutation matrices or permutation vectors.

As **matrices**, the outputs `P` and `Q` satisfy these identities:

- Three outputs — `P` satisfies `P*A = L*U`.
- Four outputs — `P` and `Q` satisfy `P*S*Q = L*U`.
- Five outputs — `P`, `Q`, and `D` satisfy `P*(D\S)*Q = L*U`.

As **vectors**, the outputs `P` and `Q` satisfy these identities:

- Three outputs — `P` satisfies `A(P,:) = L*U`
- Four outputs —`P` and `Q` satisfy `S(P,Q) = L*U`
- Five outputs — `P`, `Q`, and `D` satisfy `D(:,P)\S(:,Q) = L*U`.

Example: `[L,U,P] = lu(A,'vector')`

Output Arguments

L — Lower triangular factor
matrix

Lower triangular factor, returned as a matrix. The form of L depends on whether the row permutations P are returned in a separate output:

- If the third output P is specified, then L is returned as a unit lower triangular matrix (that is, a lower triangular matrix with 1s on the main diagonal).
- If the third output P is not specified, then L is returned as a row-permutation of a unit lower triangular matrix. Specifically, it is the product P'*L of the outputs P and L returned in the three output case.

U — Upper triangular factor
matrix

Upper triangular factor, returned as an upper triangular matrix.

P — Row permutation
vector | matrix

Row permutation, returned as a permutation matrix or, if the `'vector'` option is specified, as a permutation vector. Use this output to improve the numerical stability of the calculation.

See `outputForm` for a description of the identities that this output satisfies.

Q — Column permutation
vector | matrix

Column permutation, returned as a permutation matrix or, if the `'vector'` option is specified, as a permutation vector. Use this output to reduce the fill-in (number of nonzeros) in the factors of a sparse matrix.

See `outputForm` for a description of the identities that this output satisfies.

D — Row scaling
diagonal matrix

Row scaling, returned as a diagonal matrix. `D` is used to scale the values in `S` such that `P*(D\S)*Q = L*U`. Typically, but not always, the row scaling leads to a sparser and more stable factorization.

Algorithms

The LU factorization is computed using a variant of Gaussian elimination. Computing an accurate solution is dependent upon the value of the condition number of the original matrix `cond(A)`. If the matrix has a large condition number (it is nearly singular), then the computed factorization might not be accurate.

The LU factorization is a key step in obtaining the inverse with `inv` and the determinant with `det`. It is also the basis for the linear equation solution or matrix division obtained with the operators `\` and `/`. This necessarily means that the numerical limitations of `lu` are also present in these dependent functions.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- `A` must be nonsparse.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- `A` must be nonsparse.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cond` | `decomposition` | `det` | `ilu` | `inv` | `mldivide` | `qr` | `rref`

Topics

“Factorizations”

Introduced before R2006a

Line Properties

Primitive line appearance and behavior

Description

`Line` properties control the appearance and behavior of a `Line` object. By changing property values, you can modify certain aspects of the line chart.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = line;
s = h.LineStyle;
h.LineStyle = ':';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Line

`Color` — Line color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

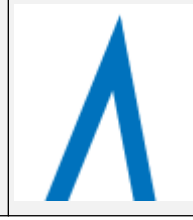
Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

`LineJoin` — Style of line corners
`'round'` (default) | `'miter'` | `'chamfer'`

Style of line corners, specified as `'round'`, `'miter'`, or `'chamfer'`. This table illustrates the appearance of the different values.

`'round'`	`'miter'`	`'chamfer'`
		

The appearance of the `'round'` option might look different if the `Renderer` property of the figure is set to `'opengl'` instead of `'painters'`.

`AlignVertexCenters` — Sharp vertical and horizontal lines
`'off'` (default) | `'on'`

Sharp vertical and horizontal lines, specified as `'off'` or `'on'`.

If the associated figure has a `GraphicsSmoothing` property set to `'on'` and a `Renderer` property set to `'opengl'`, then the figure applies a smoothing technique to

plots. In some cases, this smoothing technique can cause vertical and horizontal lines to appear uneven in thickness or color. Use the `AlignVertexCenters` property to eliminate the uneven appearance.

- `'off'` — Do not sharpen vertical or horizontal lines. The lines might appear uneven in thickness or color.
- `'on'` — Sharpen vertical and horizontal lines to eliminate an uneven appearance.

Note You must have a graphics card that supports this feature. To see if the feature is supported, call the `rendererinfo` function. If it is supported, `rendererinfo` returns value of `1` for `info.Details.SupportsAlignVertexCenters`.

Markers

`Marker` — Marker symbol

`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | ...

Marker symbol, specified as one of the values listed in this table. By default, the object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)

Value	Description
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerIndices` — Indices of data points at which to display markers
`1:length(YData)` (default) | vector of positive integers | scalar positive integer

Indices of data points at which to display markers, specified as a vector of positive integers. If you do not specify the indices, then MATLAB displays a marker at every data point.

Note To see the markers, you must also specify a marker symbol.

Example: `plot(x,y,'-o','MarkerIndices',[1 5 10])` displays a circle marker at the first, fifth, and tenth data points.

Example: `plot(x,y,'-x','MarkerIndices',1:3:length(y))` displays a cross marker every three data points.

Example: `plot(x,y,'Marker','square','MarkerIndices',5)` displays one square marker at the fifth data point.

`MarkerSize` — Marker size
`6` (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

`MarkerEdgeColor` — Marker outline color
`'auto'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'auto'` uses the same color as the `Color` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` option uses the same color as the `Color` property of the parent axes. If you specify `'auto'` and the axes plot box is invisible, the marker fill color is the color of the figure.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Data

XData — x values
[0 1] (default) | vector

x values, specified as a vector. XData and YData must have equal lengths.

Example: 1:10

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | categorical | datetime | duration

YData — y values
[0 1] (default) | vector

y values, specified as a vector. XData and YData must have equal lengths.

Example: 1:10

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | categorical | datetime | duration

ZData — z values
empty matrix (default) | vector

z values, specified as a vector. ZData must have the same length as XData and YData.

Example: 1:10

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | categorical | datetime | duration

RData — Radius values
vector

Radius values, specified as a vector. `ThetaData` and `RData` must be vectors of equal length.

This property applies only to lines in a polar axes.

`ThetaData` — Angle values
vector

Angle values, specified as a vector. `ThetaData` and `RData` must be vectors of equal length.

This property applies only to lines in a polar axes.

Legend

`DisplayName` — Legend label
`''` (default) | character vector | string scalar

Legend label, specified as a character vector or string scalar. The legend does not display until you call the `legend` command. If you do not specify the text, then `legend` sets the label using the form `'dataN'`.

`Annotation` — Control for including or excluding object from legend
`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do

appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`DataTipTemplate` — Data tip content

`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

This property applies only to lines with pinned data tips.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state
`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles
`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

`Clipping` — Clipping of object to axes limits
`'on'` (default) | `'off'`

Clipping of the object to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the object that are outside the axes limits.
- `'off'` — Display the entire object, even if parts of it appear outside the axes limits. Parts of the object might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the object so that it is larger than the original plot.

The `Clipping` property of the axes that contains the object must be set to `'on'`. Otherwise, this property has no effect. For more information about the clipping behavior, see the `Clipping` property of the axes.

Callbacks

`ButtonDownFcn` — Mouse-click callback
`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see "Callback Definition".

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks when visible. The `Visible` property must be set to `'on'` and you must click a part of the `Line` object that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The `HitTest` property determines if the `Line` object responds to the click or if an ancestor does.
- `'all'` — Capture mouse clicks regardless of visibility. The `Visible` property can be set to `'on'` or `'off'` and you can click a part of the `Line` object that has no color. The `HitTest` property determines if the `Line` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `Line` object passes the click through it to the object below it in the current view of the figure window. The `HitTest` property has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `Line` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `Line` object that has one of these:

- `HitTest` property set to `'on'`
- `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

Note The `PickableParts` property determines if the `Line` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent

`Axes` object | `PolarAxes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `PolarAxes`, `Group`, or `Transform` object.

`Children` — Children

empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the chart object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'line'`

This property is read-only.

Type of graphics object, returned as `'line'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`line`

Topics

“Access Property Values”
“Graphics Object Properties”

Introduced before R2006a

magic

Magic square

Syntax

```
M = magic(n)
```

Description

`M = magic(n)` returns an `n`-by-`n` matrix constructed from the integers `1` through n^2 with equal row and column sums. The order `n` must be a scalar greater than or equal to `3`.

Examples

Third-Order Magic Square

Compute the third-order magic square `M`.

```
M = magic(3)
```

```
M = 3×3

     8     1     6
     3     5     7
     4     9     2
```

The sum of the elements in each column and the sum of the elements in each row are the same.

```
sum(M)
```

```
ans = 1×3

    15    15    15
```

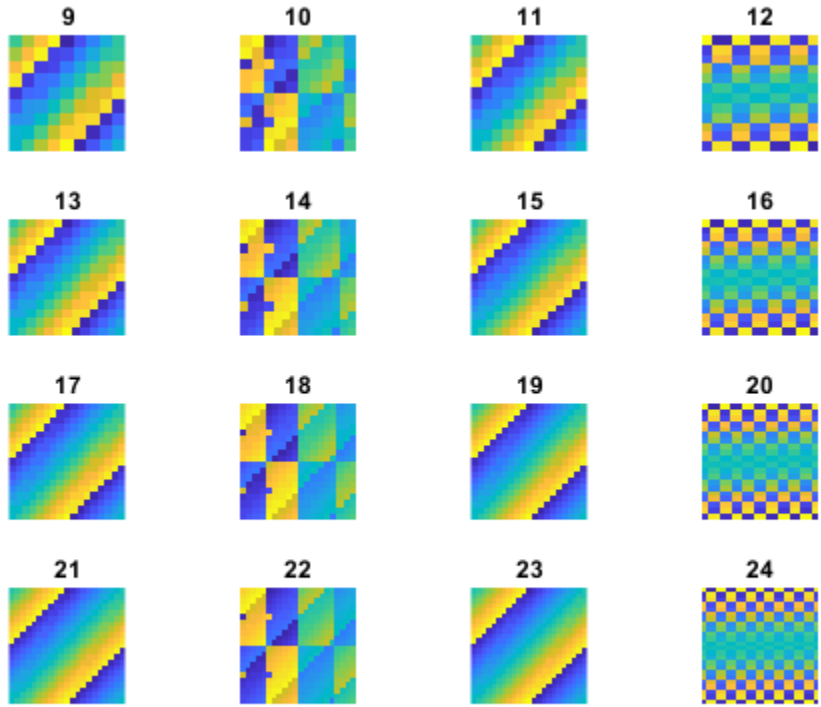
```
sum(M,2)
```

```
ans = 3×1

    15
    15
    15
```

Magic Square Visualization

Visually examine the patterns in magic square matrices with orders between 9 and 24 using `imagesc`. The patterns show that `magic` uses three different algorithms, depending on whether the value of `mod(n,4)` is 0, 2, or odd.

```
for n = 1:16
    subplot(4,4,n)
    ord = n+8;
    m = magic(ord);
    imagesc(m)
    title(num2str(ord))
    axis equal
    axis off
end
```



Input Arguments

n — Matrix order
scalar integer

Matrix order, specified as a scalar integer greater than or equal to 3. If `n` is complex, not an integer, or not scalar, then `magic` converts it into a usable integer with `floor(real(double(n(1))))`.

If you supply `n` less than `3`, then `magic` returns either a nonmagic square, or the degenerate magic squares `1` and `[]`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

See Also

`ones` | `rand`

Introduced before R2006a

makehgtform

Create 4-by-4 transform matrix

Syntax

```
M = makehgtform
M = makehgtform('translate',[tx ty tz])
M = makehgtform('scale',s)
M = makehgtform('scale',[sx,sy,sz])
M = makehgtform('xrotate',t)
M = makehgtform('yrotate',t)
M = makehgtform('zrotate',t)
M = makehgtform('axisrotate',[ax,ay,az],t)
```

Description

Use `makehgtform` to create transform matrices for translation, scaling, and rotation of graphics objects. Apply the transform to graphics objects by assigning the transform to the `Matrix` property of a parent transform object.

`M = makehgtform` returns an identity transform.

`M = makehgtform('translate',[tx ty tz])` or M = makehgtform('translate',tx,ty,tz) returns a transform that translates along the *x*-axis by `tx`, along the *y*-axis by `ty`, and along the *z*-axis by `tz`.

`M = makehgtform('scale',s)` returns a transform that scales uniformly along the *x*-, *y*-, and *z*-axes.

`M = makehgtform('scale',[sx,sy,sz])` returns a transform that scales along the *x*-axis by `sx`, along the *y*-axis by `sy`, and along the *z*-axis by `sz`.

`M = makehgtform('xrotate',t)` returns a transform that rotates around the *x*-axis by `t` radians.

`M = makehgtform('yrotate',t)` returns a transform that rotates around the *y*-axis by `t` radians.

`M = makehgtform('zrotate',t)` returns a transform that rotates around the *z*-axis by `t` radians.

`M = makehgtform('axisrotate',[ax,ay,az],t)` Rotate around axis `[ax ay az]` by `t` radians.

Note that you can specify multiple operations in one call to `makehgtform` and the MATLAB software returns a transform matrix that is the result of concatenating all specified operations. For example,

```
m = makehgtform('xrotate',pi/2,'yrotate',pi/2);
```

is the same as

```
mx = makehgtform('xrotate',pi/2);
my = makehgtform('yrotate',pi/2);
m = mx*my;
```

See Also

`hggroup` | `hgtransform`

Topics

"Create Object Groups"

Introduced before R2006a

makima

Modified Akima piecewise cubic Hermite interpolation

Syntax

```
yq = makima(x,y,xq)
pp = makima(x,y)
```

Description

`yq = makima(x,y,xq)` performs “Modified Akima Interpolation” on page 1-8846 using the values `y` at sample points `x` to find interpolated values `yq` at the query points `xq`.

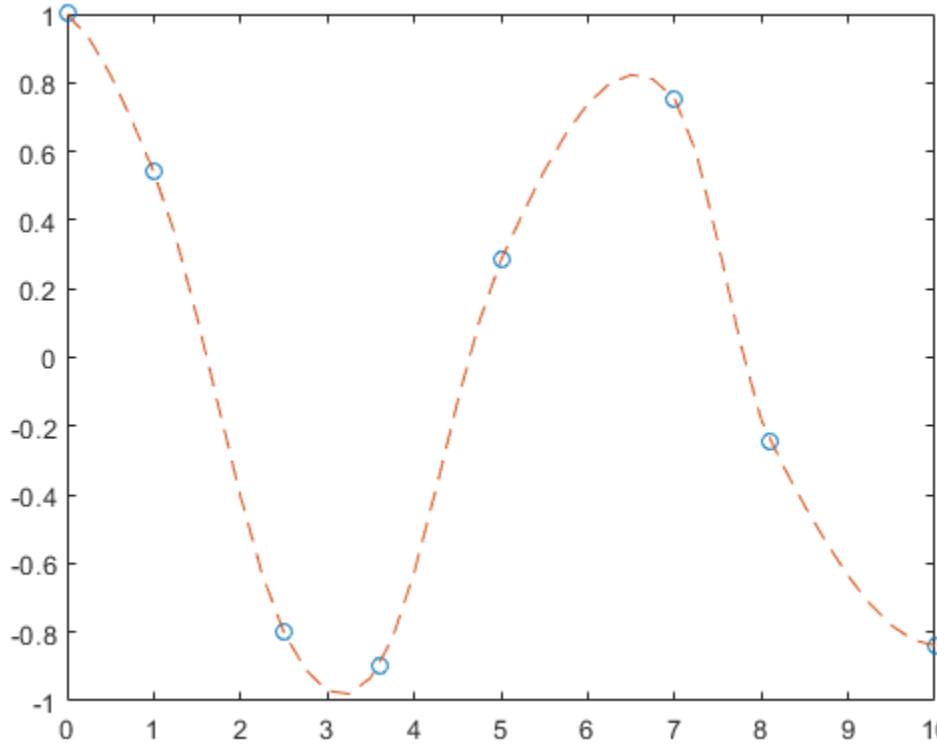
`pp = makima(x,y)` returns a piecewise polynomial structure for use with `ppval` and the spline utility `unmkpp`.

Examples

Akima Interpolation of Cosine Data

Use `makima` to interpolate a cosine curve over unevenly spaced sample points.

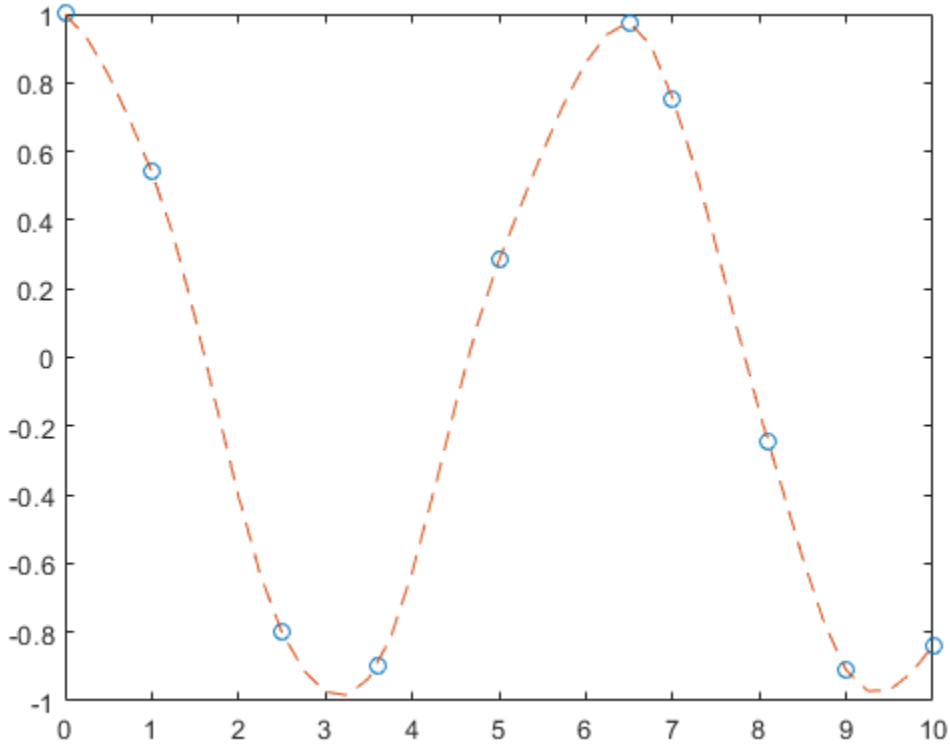
```
x = [0 1 2.5 3.6 5 7 8.1 10];
y = cos(x);
xq = 0:.25:10;
yq = makima(x,y,xq);
plot(x,y,'o',xq,yq,'--')
```



With oscillatory functions, the Akima algorithm flattens the curve near local extrema. To compensate for this flattening, you can add more sample points near the local extrema.

Add sample points near $x = 6.5$ and $x = 9$ and replot the interpolation.

```
x = [0 1 2.5 3.6 5 6.5 7 8.1 9 10];
y = cos(x);
xq = 0:.25:10;
yq = makima(x,y,xq);
plot(x,y,'o',xq,yq,'--')
```

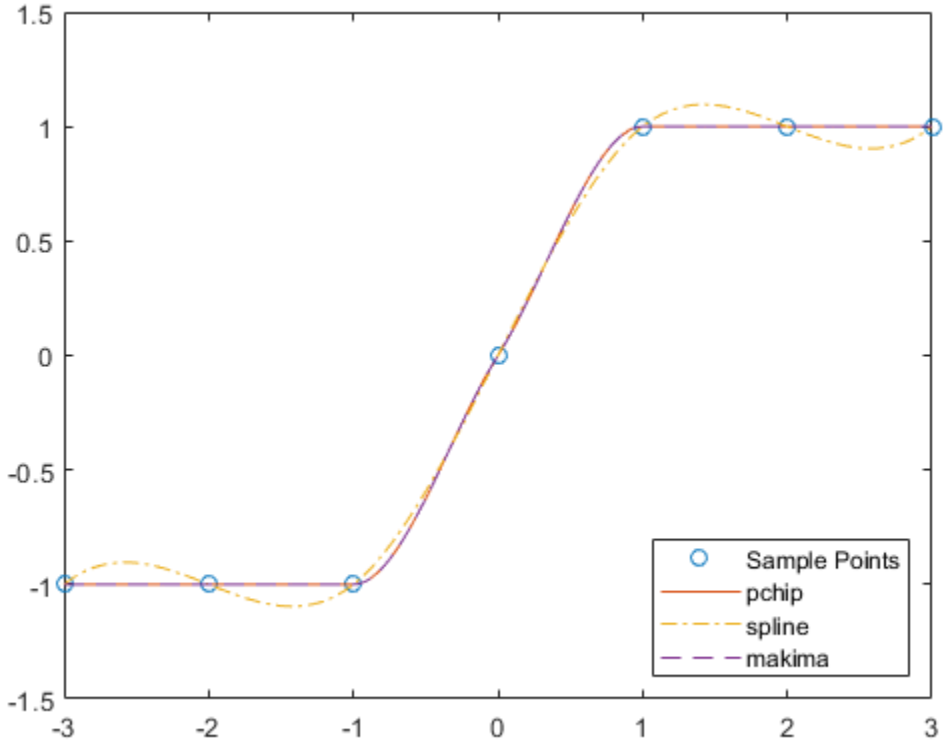


Data Interpolation with `spline`, `pchip`, and `makima`

Compare the interpolation results produced by `spline`, `pchip`, and `makima` for two different data sets. These functions all perform different forms of piecewise cubic Hermite interpolation. Each function differs in how it computes the slopes of the interpolant, leading to different behaviors when the underlying data has flat areas or undulations.

Compare the interpolation results on sample data that connects flat regions. Create vectors of `x` values, function values at those points `y`, and query points `xq`. Compute interpolations at the query points using `spline`, `pchip`, and `makima`. Plot the interpolated function values at the query points for comparison.

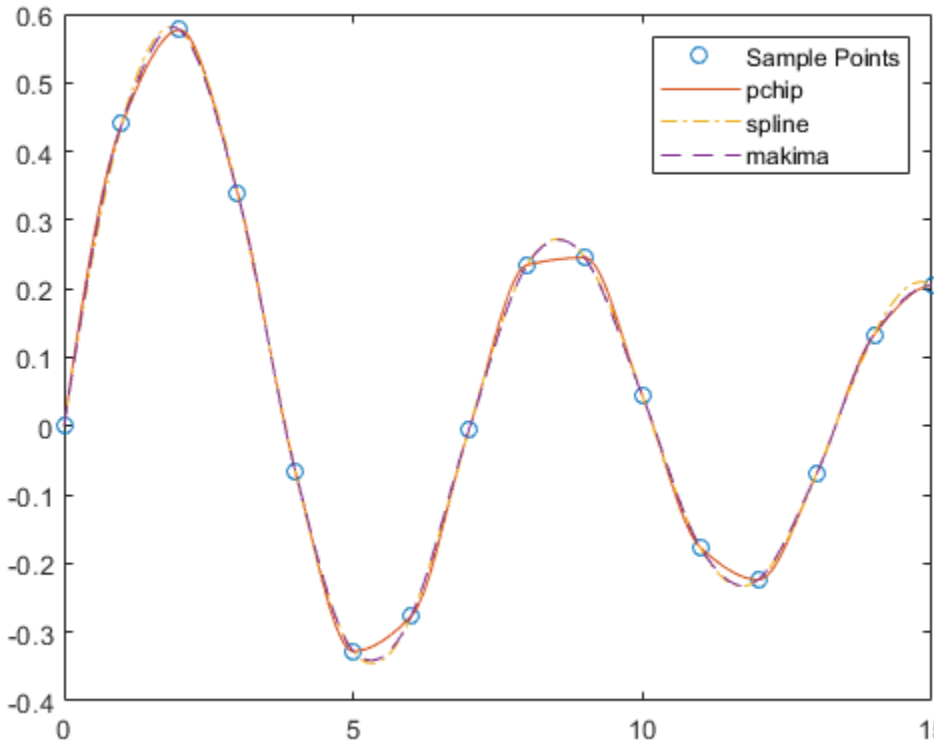
```
x = -3:3;
y = [-1 -1 -1 0 1 1 1];
xq1 = -3:.01:3;
p = pchip(x,y,xq1);
s = spline(x,y,xq1);
m = makima(x,y,xq1);
plot(x,y,'o',xq1,p,'-',xq1,s,'-.',xq1,m,'--')
legend('Sample Points','pchip','spline','makima','Location','SouthEast')
```



In this case, `pchip` and `makima` have similar behavior in that they avoid overshoots and can accurately connect the flat regions.

Perform a second comparison using an oscillatory sample function.

```
x = 0:15;
y = besselj(1,x);
xq2 = 0:0.01:15;
p = pchip(x,y,xq2);
s = spline(x,y,xq2);
m = makima(x,y,xq2);
plot(x,y,'o',xq2,p,'-',xq2,s,'-.',xq2,m,'--')
legend('Sample Points','pchip','spline','makima')
```



When the underlying function is oscillatory, `spline` and `makima` capture the movement between points better than `pchip`, which is aggressively flattened near local extrema.

Akima Interpolation with Piecewise Polynomial Structure

Create vectors for the sample points `x` and values at those points `y`. Use `makima` to construct a piecewise polynomial structure for the data.

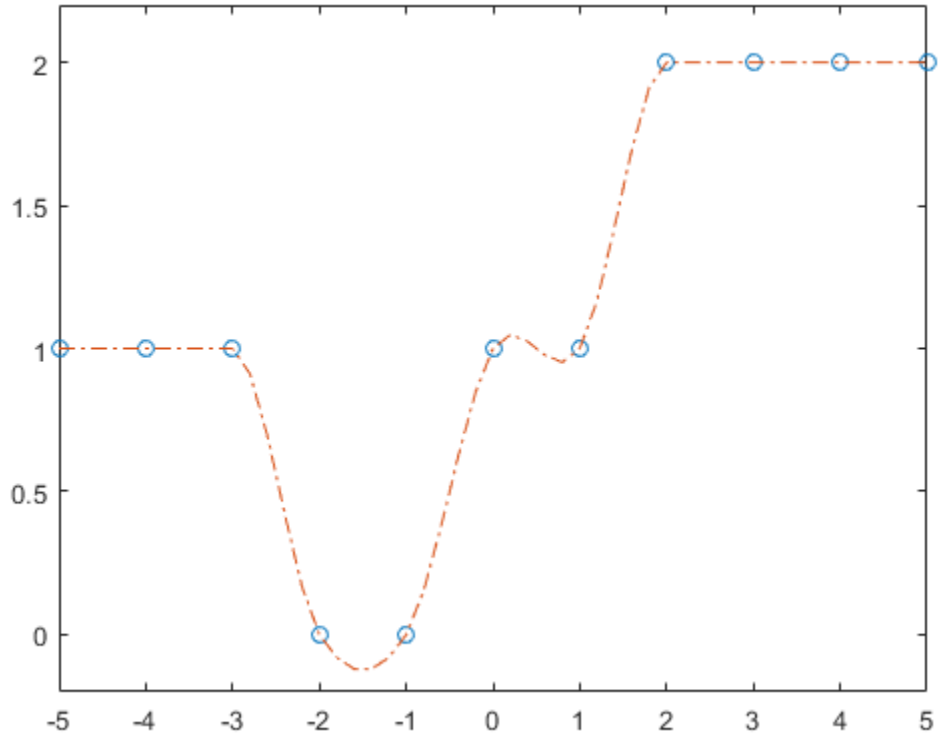
```
x = -5:5;
y = [1 1 1 0 0 1 1 2 2 2 2];
pp = makima(x,y)
```

```
pp = struct with fields:
      form: 'pp'
    breaks: [-5 -4 -3 -2 -1 0 1 2 3 4 5]
     coefs: [10x4 double]
    pieces: 10
     order: 4
       dim: 1
```

The structure contains the information for 10 polynomials of order 4 that span the data. `pp.coefs(i,:)` contains the coefficients for the polynomial that is valid in the region defined by the breakpoints `[breaks(i) breaks(i+1)]`.

Use the structure with `ppval` to evaluate the interpolation at several query points, and then plot the results. In regions with three or more constant points, the Akima algorithm connects the points with a straight line.

```
xq = -5:0.2:5;
m = ppval(pp,xq);
plot(x,y,'o',xq,m,'-.')
ylim([-0.2 2.2])
```



Input Arguments

x — Sample points
vector

Sample points, specified as a vector. The vector x specifies the points at which the data y is given. The elements of x must be unique.

Data Types: `single` | `double`

y — Function values at sample points
vector | matrix | array

Function values at sample points, specified as a numeric vector, matrix, or array. `x` and `y` must have the same length.

If `y` is a matrix or array, then the values in the last dimension, `y(:,...,:,j)`, are taken as the values to match with `x`. In that case, the last dimension of `y` must be the same length as `x`.

Data Types: `single` | `double`

xq — Query points
scalar | vector | matrix | array

Query points, specified as a scalar, vector, matrix, or array. The points specified in `xq` are the *x*-coordinates for the interpolated function values `yq` computed by `makima`.

Data Types: `single` | `double`

Output Arguments

yq — Interpolated values at query points
scalar | vector | matrix | array

Interpolated values at query points, returned as a scalar, vector, matrix, or array. The size of `yq` is related to the sizes of `y` and `xq`:

- If `y` is a vector, then `yq` has the same size as `xq`.
- If `y` is an array of size `Ny = size(y)`, then these conditions apply:
  - If `xq` is a scalar or vector, then `size(yq)` returns `[Ny(1:end-1) length(xq)]`.
  - If `xq` is an array, then `size(yq)` returns `[Ny(1:end-1) size(xq)]`.

pp — Piecewise polynomial
structure

Piecewise polynomial, returned as a structure. Use this structure with the `ppval` function to evaluate the interpolating polynomials at one or more query points. The structure has these fields.

Field	Description
`form`	`'pp'` for *piecewise polynomial*

Field	Description
`breaks`	Vector of length `L+1` with strictly increasing elements that represent the start and end of each of `L` intervals
`coefs`	`L`-by-`k` matrix with each row `coefs(i,:)` containing the local coefficients of an order `k` polynomial on the `i`th interval, `[breaks(i),breaks(i+1)]`
`pieces`	Number of pieces, `L`
`order`	Order of polynomials
`dim`	Dimensionality of target

Because the polynomial coefficients in `coefs` are local coefficients for each interval, you must subtract the lower endpoint of the corresponding knot interval to use the coefficients in a conventional polynomial equation. In other words, for the coefficients `[a,b,c,d]` on the interval `[x1,x2]`, the corresponding polynomial is

$$f(x) = a(x - x_1)^3 + b(x - x_1)^2 + c(x - x_1) + d \ .$$

More About

Modified Akima Interpolation

The **Akima algorithm** for one-dimensional interpolation, described in [1] on page 1-8847 and [2] on page 1-8847, performs cubic interpolation to produce piecewise polynomials with continuous first-order derivatives (C1). The algorithm avoids excessive local undulations.

If $\delta_i = \frac{v_{i+1} - v_i}{x_{i+1} - x_i}$ is the slope on interval $[x_i \ x_{i+1})$, then the value of the derivative d_i at the sample point x_i is a weighted average of nearby slopes:

$$d_i = \frac{w_1}{w_1 + w_2}\delta_{i-1} + \frac{w_2}{w_1 + w_2}\delta_i \, .$$

In Akima's original formula, the weights are:

$$w_1 = |\delta_{i+1} - \delta_i|,$$
$$w_2 = |\delta_{i-1} - \delta_{i-2}|.$$

The original Akima algorithm gives equal weight to the points on both sides, evenly dividing an undulation.

When two flat regions with different slopes meet, the modification made to the original Akima algorithm gives more weight to the side where the slope is closer to zero. This modification gives priority to the side that is closer to horizontal, which is more intuitive and avoids overshoot. In particular, whenever there are three or more consecutive collinear points, the algorithm connects them with a straight line and thus avoids an overshoot.

The weights used in the modified Akima algorithm are:

$$w_1 = |\delta_{i+1} - \delta_i| + \frac{|\delta_{i+1} + \delta_i|}{2},$$
$$w_2 = |\delta_{i-1} - \delta_{i-2}| + \frac{|\delta_{i-1} + \delta_{i-2}|}{2}.$$

Compared to the `spline` algorithm, the Akima algorithm produces fewer undulations and is better suited to deal with quick changes between flat regions. Compared to the `pchip` algorithm, the Akima algorithm is not as aggressively flattened and is therefore still able to deal with oscillatory data.

References

[1] Akima, Hiroshi. "A new method of interpolation and smooth curve fitting based on local procedures." *Journal of the ACM (JACM)* , 17.4, 1970, pp. 589–602.

[2] Akima, Hiroshi. "A method of bivariate interpolation and smooth surface fitting based on local procedures." *Communications of the ACM* , 17.1, 1974, pp. 18–20.

See Also

`interp1` | `pchip` | `ppval` | `spline`

External Websites

Cleve's Corner: Makima Piecewise Cubic Interpolation

Introduced in R2019b

mapreduce

Programming technique for analyzing data sets that do not fit in memory

Syntax

```
outds = mapreduce(ds,mapfun,reducefun)
outds = mapreduce(ds,mapfun,reducefun,mr)
outds = mapreduce( ___ ,Name,Value)
```

Description

`outds = mapreduce(ds,mapfun,reducefun)` applies map function `mapfun` to input datastore `ds`, and then passes the values associated with each unique key to reduce function `reducefun`. The output datastore is a `KeyValueDatastore` object that points to `.mat` files in the current folder.

`outds = mapreduce(ds,mapfun,reducefun,mr)` optionally specifies the run-time configuration settings for `mapreduce`. The `mr` input is the result of a call to the `mapreducer` function. Typically, this argument is used with Parallel Computing Toolbox, MATLAB Parallel Server, or MATLAB Compiler. For more information, see “Speed Up and Deploy MapReduce Using Other Products”.

`outds = mapreduce( ___ ,Name,Value)` specifies additional options with one or more `Name,Value` pair arguments using any of the previous syntaxes. For example, you can specify `'OutputFolder'` followed by a character vector specifying a path to the output folder.

Examples

Count Flights by Airline

Use `mapreduce` to count the number of flights made by each unique airline carrier in a data set.

Create a datastore using the `airlinesmall.csv` data set. This 12-megabyte data set contains 29 columns of flight information for several airline carriers, including arrival and departure times. In this example, select `UniqueCarrier` (airline name) as the variable of interest. Specify the `'TreatAsMissing'` name-value pair so that the datastore treats `'NA'` values as missing and replaces them with `NaN` values.

```
ds = tabularTextDatastore('airlinesmall.csv', 'TreatAsMissing', 'NA');
ds.SelectedVariableNames = 'UniqueCarrier';
ds.SelectedFormats = '%C';
```

Preview the data.

```
preview(ds)
```

```
ans=8×1 table
    UniqueCarrier
    _____________

         PS
         PS
         PS
         PS
         PS
         PS
         PS
         PS
```

Run `mapreduce` on the data. The map and reduce functions count the number of instances of each airline carrier name in each block of data, then combine those intermediate counts into a final count. This method leverages the intermediate sorting by unique key performed by `mapreduce`. The functions `countMapper` and `countReducer` are included at the end of this script.

```
outds = mapreduce(ds, @countMapper, @countReducer);
```

```
********************************
*      MAPREDUCE PROGRESS      *
********************************
Map   0% Reduce   0%
Map  16% Reduce   0%
Map  32% Reduce   0%
Map  48% Reduce   0%
Map  65% Reduce   0%
```

```
Map  81% Reduce   0%
Map  97% Reduce   0%
Map 100% Reduce   0%
Map 100% Reduce  10%
Map 100% Reduce  21%
Map 100% Reduce  31%
Map 100% Reduce  41%
Map 100% Reduce  52%
Map 100% Reduce  62%
Map 100% Reduce  72%
Map 100% Reduce  83%
Map 100% Reduce  93%
Map 100% Reduce 100%
```

```
readall(outds)
```

```
ans=29×2 table
       Key           Value
    __________    _________

    {'AA'    }    {[14930]}
    {'AS'    }    {[ 2910]}
    {'CO'    }    {[ 8138]}
    {'DL'    }    {[16578]}
    {'EA'    }    {[  920]}
    {'HP'    }    {[ 3660]}
    {'ML (1)'}    {[   69]}
    {'NW'    }    {[10349]}
    {'PA (1)'}    {[  318]}
    {'PI'    }    {[  871]}
    {'PS'    }    {[   83]}
    {'TW'    }    {[ 3805]}
    {'UA'    }    {[13286]}
    {'US'    }    {[13997]}
    {'WN'    }    {[15931]}
    {'AQ'    }    {[  154]}
      ⋮
```

The map function `countMapper` leverages the fact that the data is categorical. The `countcats` and `categories` functions are used on each block of the input data to generate key/value pairs of the airline name and associated count.

```
function countMapper(data, info, intermKV)
% Counts unique airline carrier names in each block.
```

```
a = data.UniqueCarrier;
c = num2cell(countcats(a));
keys = categories(a);
addmulti(intermKV, keys, c)
end
```

The reduce function `countReducer` reads in the intermediate data produced by the map function and adds together all of the counts to produce a single final count for each airline carrier.

```
function countReducer(key, intermValIter, outKV)
% Combines counts from all blocks to produce final counts.
count = 0;
while hasnext(intermValIter)
    data = getnext(intermValIter);
    count = count + data;
end
add(outKV, key, count)
end
```

Input Arguments

ds — Input datastore
datastore object

Input datastore, specified as a datastore object. Use the `datastore` function to create a datastore object from your data set.

`mapreduce` works only with datastores that are deterministic. That is, if you use `read` on the datastore, reset the datastore with `reset`, and then read the datastore again, then the data returned must be the same in both cases. `mapreduce` calculations involving a datastore that is not deterministic can produce unpredictable results. See “Select Datastore for File Format or Application” for more information.

mapfun — Function handle to map function
function handle

Function handle to map function. `mapfun` receives blocks from input datastore `ds`, and then uses the `add` and `addmulti` functions to add key-value pairs to an intermediate `KeyValueStore` object. The number of calls to the map function by `mapreduce` is equal

to the number of blocks in the `datastore` (the number of blocks is determined by the `ReadSize` property of the datastore).

The inputs to the map function are `data`, `info`, and `intermKVStore`, which `mapreduce` automatically creates and passes to the map function:

- The `data` and `info` inputs are the result of a call to the `read` function of `datastore`, which `mapreduce` executes automatically before each call to the map function.
- `intermKVStore` is the name of the intermediate `KeyValueStore` object to which the map function needs to add key-value pairs. If none of the calls to the map function add key-value pairs to `intermKVStore`, then `mapreduce` does not call the reduce function and the output datastore is empty.

An example of a template for the map function is

```
function myMapper(data, info, intermKVStore)
%do a calculation with the data block
add(intermKVStore, key, value)
end
```

Example: `@myMapper`

Data Types: `function_handle`

`reducefun` — Function handle to reduce function
function handle

Function handle to reduce function. `mapreduce` calls `reducefun` once for each unique key added to the intermediate `KeyValueStore` by the map function. In each call, `mapreduce` passes the values associated with the active key to `reducefun` as a `ValueIterator` object. The `reducefun` function loops through the values for each key using the `hasnext` and `getnext` functions. Then, after performing some calculation(s), it writes key-value pairs to the final output.

The inputs to the reduce function are `intermKey`, `intermValIter`, and `outKVStore`, which `mapreduce` automatically creates and passes to the reduce function:

- `intermKey` is the active key from the intermediate `KeyValueStore` object. Each call to the reduce function by `mapreduce` specifies a new unique key from the keys in the intermediate `KeyValueStore` object.
- `intermValIter` is the `ValueIterator` associated with the active key, `intermKey`. This `ValueIterator` object contains all of the values associated with the active key. Scroll through the values using the `hasnext` and `getnext` functions.

- `outKVStore` is the name for the final `KeyValueStore` object to which the reduce function needs to add key-value pairs. `mapreduce` takes the output key-value pairs from `outKVStore` and returns them in the output datastore, `outds`, which is a `KeyValueDatastore` object by default. If none of the calls to the reduce function add final key-value pairs to `outKVStore`, then the output datastore is empty.

An example of a template for the reduce function is

```
function myReducer(intermKey, intermValIter, outKVStore)
while hasnext(intermValIter)
    X = getnext(intermValIter);
    %do a calculation with the current value, X
end
add(outKVStore, key, value)
end
```

Example: `@myReducer`

Data Types: `function_handle`

`mr` — Execution environment
MapReducer object

Execution environment, specified as a MapReducer object. `mr` is the result of a call to the `mapreducer` function. The default `mr` argument is a call to `gcmr`, which uses the default global execution environment for `mapreduce` (in MATLAB the default is `mapreducer(0)`, which returns a `SerialMapReducer` object).

Note This setting specifies the execution environment for `mapreduce` and is not necessary to run `mapreduce` on your local computer. For more information, see “Speed Up and Deploy MapReduce Using Other Products”.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `outds = mapreduce(ds, @mapfun, @reducefun, 'Display', 'off', 'OutputFolder', 'C:\Users\username\Desktop')`

`OutputType` — Type of datastore output
`'Binary'` (default) | `'TabularText'`

Type of datastore output, specified as `'Binary'` or `'TabularText'`. The default setting of `'Binary'` returns a `KeyValueDatastore` output datastore that points to binary (`.mat` or `.seq`) files in the output folder. The `'TabularText'` option returns a `tabularTextDatastore` output datastore that points to `.txt` files in the output folder.

The table provides the details for each of the output types.

`'OutputType'`	Type of datastore output	Datastore points to files of type	Values that the Reduce function can add	Keys that the Reduce function can add	Details
`'Binary'` (default)	`KeyValueDatastore`	`.mat` (or `.seq` when running against Hadoop).	Any valid MATLAB object.	Character vectors, strings, or numeric scalars that are not `NaN`, complex, logical, or sparse.	N/A
`'TabularText'`	`TabularTextDatastore`	`.txt`	Character vectors, strings, or numeric scalars that are not `NaN`, complex, logical, or sparse.	Character vectors, strings, or numeric scalars that are not `NaN`, complex, logical, or sparse.	• File is UTF-8 encoded. • Keys and values are tab (`\t`) separated. • The row delimiter is `\r\n` on Windows, and `\n` on Linux and Mac.

Data Types: `char` | `string`

`OutputFolder` — Destination folder of `mapreduce` output
`pwd` (default) | file path

Destination folder for `mapreduce` output, specified as a file path. The default output folder is the current folder, `pwd`. You can specify a different path with a fully qualified path or with a path relative to the current folder.

Example: `mapreduce(..., 'OutputFolder', 'MyOutputFolder\Results')` specifies a file path relative to the current folder for the output.

Data Types: `char` | `string`

`Display` — Toggle for command line progress output
`'on'` (default) | `'off'`

Toggle for command line progress output, specified as `'on'` or `'off'`. The default is `'on'`, so that `mapreduce` displays progress information in the command window during the map and reduce phases of execution.

Data Types: `char` | `string`

Output Arguments

`outds` — Output datastore
`KeyValueDatastore` (default) | `TabularTextDatastore`

Output datastore, returned as a `KeyValueDatastore` or `TabularTextDatastore` object. By default, `outds` is a `KeyValueDatastore` object that points to `.mat` files in the current folder. Use the `Name,Value` pair arguments for `'OutputType'` and `'OutputFolder'` to return a `tabularTextDatastore` object or change the location of the output files, respectively.

`mapreduce` does not sort the key-value pairs in `outds`. Their order may differ when using other products with `mapreduce`.

To view the contents of `outds`, use the `preview`, `read`, or `readall` functions of `datastore`.

Tips

- Debugging your `mapreduce` algorithms to examine how key-value pairs move through the different phases is always useful. To examine the movement of data, set breakpoints in your map and reduce functions. The breakpoints stop execution of `mapreduce`, allowing you to examine the current status of relevant variables, like the `KeyValueStore` or `ValueIterator`. For more information, see “Debug MapReduce Algorithms”.
- Some recommendations to optimize `mapreduce` performance on any platform are:
  - Minimize the number of calls to the map function. The easiest approach is to increase the value of the `ReadSize` property of the input datastore. The result is that `mapreduce` passes larger blocks of data to the map function, and the datastore depletes with fewer reads.
  - Decrease the amount of intermediate data sent between map and reduce functions. One approach is to use `unique` inside a map function to combine similar keys. See “Compute Mean by Group Using MapReduce” for an example of this technique.

See Also

`KeyValueStore` | `ValueIterator` | `datastore` | `gcmr` | `mapreducer` | `tall`

Topics

“Compute Mean Value with MapReduce”
“Compute Summary Statistics by Group Using MapReduce”
“Using MapReduce to Fit a Logistic Regression Model”
“Getting Started with MapReduce”
“Speed Up and Deploy MapReduce Using Other Products”
“Build Effective Algorithms with MapReduce”

Introduced in R2014b

mapreducer

Define execution environment for mapreduce or tall arrays

Syntax

```
mapreducer
mapreducer(0)
mapreducer(mr)
mr = mapreducer( ___ )
mr = mapreducer( ___ ,'ObjectVisibility','Off')
```

Description

`mapreducer`, with no arguments, sets the global execution environment to be the default: a parallel pool if you have Parallel Computing Toolbox available, or else the local MATLAB session. `mapreducer` is a configuration function that changes how MATLAB executes `mapreduce` algorithms and tall array calculations. Use this function to set, change, or store the execution environment to leverage Parallel Computing Toolbox, MATLAB Parallel Server, or MATLAB Compiler. If you have Parallel Computing Toolbox installed, then when you use the `tall` or `mapreduce` functions MATLAB automatically starts a parallel pool of workers (unless you have changed the default preferences).

Note If no toolboxes are available, then `mapreduce` algorithms and tall array calculations automatically run using the local MATLAB session, and it is unnecessary to specify configuration settings using `mapreducer` to use these features. However, if you have Parallel Computing Toolbox, MATLAB Parallel Server, or MATLAB Compiler, then additional `mapreducer` configuration options are available for running in parallel or deployed environments.

For more information, see `mapreducer` in the Parallel Computing Toolbox documentation, or `mapreducer` in the MATLAB Compiler documentation.

`mapreducer(0)` sets the global execution environment to be the local MATLAB session.

`mapreducer(mr)` sets the global execution environment using a previously created MapReducer object, `mr`.

`mr = mapreducer( ___ )` also returns a MapReducer object using any of the previous syntaxes. You can use `mr` as a fourth input argument to `mapreduce` when you want to explicitly specify the execution environment.

`mr = mapreducer( ___ ,'ObjectVisibility','Off')` toggles the visibility of MapReducer object `mr`. Use this syntax to create new MapReducer objects without affecting the global execution environment.

Examples

Switch Between Execution Environments

The command

```
mapreducer
```

automatically starts a parallel pool if Parallel Computing Toolbox is available. Otherwise, the execution environment is set to be the local MATLAB session.

You can force MATLAB to use the local session with the command

```
mapreducer(0)
```

To query the current global execution environment, use the command

```
gcmr
```

Output Arguments

`mr` — Execution environment
object

Execution environment, returned as a MapReducer object.

If the `ObjectVisibility` property of `mr` is set to `'On'` (the default), then `mr` defines the execution environment for all `mapreduce` algorithms and tall array calculations. You can

optionally pass `mr` to the `mapreduce` function to explicitly specify the execution environment, even if its `ObjectVisibility` property is set to `'Off'`.

Tips

- When working with tall arrays, use `mapreducer` to set the execution environment prior to creating the tall array with `tall(ds)`. Tall arrays are bound to the current global execution environment when they are constructed. If the global execution environment is subsequently changed, then the tall array becomes invalid and must be reconstructed.

See Also

`gcmr` | `mapreduce` | `tall`

Topics

"Speed Up and Deploy MapReduce Using Other Products"
"Extend Tall Arrays with Other Products"

Introduced in R2014b

containers.Map

Object that maps values to unique keys

Description

A `Map` object is a data structure that allows you to retrieve values using a corresponding key. Keys can be real numbers or character vectors. As a result, they provide more flexibility for data access than array indices, which must be positive integers. Values can be scalar or nonscalar arrays.

Creation

Syntax

```
M = containers.Map(keySet,valueSet)
M = containers.Map(keySet,valueSet,'UniformValues',isUniform)
M = containers.Map('KeyType',kType,'ValueType',vType)
M = containers.Map
```

Description

`M = containers.Map(keySet,valueSet)` creates a `Map` object that contains keys from `keySet`, each mapped to a corresponding value from `valueSet`. The input arguments `keySet` and `valueSet` must have the same number of elements, with `keySet` having elements that are unique.

`M = containers.Map(keySet,valueSet,'UniformValues',isUniform)`, where `isUniform` is `false`, specifies that the values in `valueSet` do not need to be uniform. The default value of `isUniform` is `true`. The values in `valueSet` are uniform when they are all scalars that have the same data type, or when they are all character vectors.

`M = containers.Map('KeyType',kType,'ValueType',vType)` creates an empty `Map` object and specifies the data types of the keys and values you can add to it later. You

can switch the order of the `'KeyType'` and `'ValueType'` name-value pair arguments, but both name-value pairs are required.

`M = containers.Map` creates an empty `Map` object.

Input Arguments

`keySet` — Keys

numeric array | cell array of character vectors | string array

Keys, specified as a numeric array, cell array of character vectors, or string array.

If you specify keys using a string array, then the `containers.Map` function converts the keys and stores them as character vectors. Because of this conversion, the `KeyType` property of the output `Map` object is set to `'char'`.

`valueSet` — Values

array

Values, specified as an array.

`isUniform` — Indicator of uniform values

`true` (default) | `false`

Indicator of uniform values in `valueSet`, specified as `true` (`1`) or `false` (`0`).

`kType` — Data type of keys

`'char'` (default) | `'double'` | `'single'` | `'int32'` | `'uint32'` | `'int64'` | `'uint64'`

Data type of the keys to be added to an empty `Map` object, specified as a character vector. You can specify `kType` as any of the data types in the table, so that keys are either character vectors or numeric scalars.

kType	Data Type and Size of Key
`'char'` (default)	Character vector
`'double'`	Double scalar
`'single'`	Single scalar
`'int32'`	32-bit signed integer scalar
`'uint32'`	32-bit unsigned integer scalar

kType	Data Type and Size of Key
`'int64'`	64-bit signed integer scalar
`'uint64'`	64-bit unsigned integer scalar

vType — Data type of values
`'any'` (default) | `'char'` | `'logical'` | `'double'` | `'single'` | `'int8'` | `'uint8'` | ...

Data type of the values to be added to an empty `Map` object, specified as a character vector. You can specify `vType` as any of the data types in the table.

vType	Data Type and Size of Value
`'any'` (default)	Array that has any data type
`'char'`	Character vector
`'logical'`	Logical scalar
`'double'`	Double scalar
`'single'`	Single scalar
`'int8'`	8-bit signed integer scalar
`'uint8'`	8-bit unsigned integer scalar
`'int16'`	16-bit signed integer scalar
`'uint16'`	16-bit unsigned integer scalar
`'int32'`	32-bit signed integer scalar
`'uint32'`	32-bit unsigned integer scalar
`'int64'`	64-bit signed integer scalar
`'uint64'`	64-bit unsigned integer scalar

Properties

`Count` — Number of key-value pairs
numeric scalar

This property is read-only.

Number of key-value pairs in the `Map` object, specified as a numeric scalar.

KeyType — Data type of keys
`'char'` (default) | character vector

This property is read-only.

Data type of the keys, specified as a character vector.

`ValueType` — Data type of values
`'any'` (default) | character vector

This property is read-only.

Data type of the values, specified as a character vector.

Object Functions

isKey	Determine if Map object contains key
keys	Return keys of Map object
length	Number of key-value pairs in Map object
remove	Delete key-value pairs from Map object
size	Size of Map object
values	Return values of Map object

Examples

Create Map

Create a `Map` object that contains rainfall data for several months. The map contains the four values in `valueSet`, and the keys are the four month names in `keySet`.

```
keySet = {'Jan','Feb','Mar','Apr'};
valueSet = [327.2 368.2 197.6 178.4];
M = containers.Map(keySet,valueSet)
```

```
M = 
  Map with properties:

        Count: 4
      KeyType: char
```

```
    ValueType: double
```

Display the rainfall for March. You can retrieve the value for March by using `'Mar'` as the key.

```
M('Mar')
```

```
ans = 197.6000
```

Display the number of values in the map. You can access the `Count` property using dot notation.

```
M.Count
```

```
ans = uint64
    4
```

Specify Numbers as Keys

Create a `Map` object with identification numbers as keys and employee names as values.

```
ids = [437 1089 2362];
names = {'Lee, N.','Jones, R.','Sanchez, C.'};
M = containers.Map(ids,names)
```

```
M = 
  Map with properties:

        Count: 3
      KeyType: double
    ValueType: char
```

Retrieve a name using an identification number as a key.

```
M(437)
```

```
ans = 
'Lee, N.'
```

Store Values That Are Not Uniform

Create a `Map` object that contains test results for patients. For any patient, the results might be in either a numeric array or in a file. You can store numeric arrays and file names as values in the same map. To store values that do not have the same data type in the same map, specify `'UniformValues',false`.

```
keySet = {'Li','Jones','Sanchez'};
testLi = [5.8 7.35];
testJones = [27 3.92 6.4 8.21];
testSanchez = 'C:\Tests\Sanchez.dat';

valueSet = {testLi,testJones,testSanchez};
M = containers.Map(keySet,valueSet,'UniformValues',false)
```

```
M = 
  Map with properties:

        Count: 3
      KeyType: char
    ValueType: any
```

Display the numeric array associated with Li.

```
M('Li')
```

```
ans = 1×2

    5.8000    7.3500
```

Display the file name associated with Sanchez. If the file contains numeric values, you could then call a function to read those values into an array.

```
M('Sanchez')
```

```
ans = 
'C:\Tests\Sanchez.dat'
```

Specify Types for Empty Map

Create an empty `Map` object. Specify the data types for key-value pairs added later.

```
M = containers.Map('KeyType','char','ValueType','double')
```

```
M = 

  Map with properties:

        Count: 0
      KeyType: char
    ValueType: double
```

Add key-value pairs to the map.

```
M('Jan') = 327.2;
M('Feb') = 368.2;
M
```

```
M = 
  Map with properties:

        Count: 2
      KeyType: char
    ValueType: double
```

Display the keys and values that the map now contains.

```
keys(M)
```

```
ans = 1x2 cell array
    {'Feb'}    {'Jan'}
```

```
values(M)
```

```
ans=1×2 cell
    {[368.2000]}    {[327.2000]}
```

See Also

cell | isKey | keys | struct | table | values

Topics

"Overview of Map Data Structure"

“Description of Map Class”
“Create Map Object”
“Examine Contents of Map”
“Map to Different Value Types”

Introduced in R2008b

mat2cell

Convert array to cell array whose cells contain subarrays

Syntax

```
C = mat2cell(A,dim1Dist,...,dimNDist)
C = mat2cell(A,rowDist)
```

Description

`C = mat2cell(A,dim1Dist,...,dimNDist)` divides array `A` into smaller arrays and returns them in cell array `C`. The vectors `dim1Dist,...dimNDist` specify how to divide the rows, the columns, and (when applicable) the higher dimensions of `A`. The smaller arrays in `C` can have different sizes. `A` can have any data type.

`C = mat2cell(A,rowDist)` divides array `A` into an `n`-by-1 cell array `C`, where `n` equals the number of elements in `rowDist`.

Examples

Divide Array and Return Subarrays in Cell Array

Create a 5-by-4 numeric array.

```
A = reshape(1:20,5,4)'
```

```
A = 4×5

     1     2     3     4     5
     6     7     8     9    10
    11    12    13    14    15
    16    17    18    19    20
```

Divide `A` into two 2-by-3 and two 2-by-2 subarrays. Return the subarrays in a cell array.

```
C = mat2cell(A,[2 2],[3 2])
```

```
C=2×2 cell
    {2x3 double}    {2x2 double}
    {2x3 double}    {2x2 double}
```

Display the subarrays in `C` using the `celldisp` function.

```
celldisp(C)
```

```
C{1,1} =

     1     2     3
     6     7     8



C{2,1} =

    11    12    13
    16    17    18



C{1,2} =

     4     5
     9    10



C{2,2} =

    14    15
    19    20
```

Divide Array by Rows

Create an array.

```
A = reshape(1:20,5,4)'
```

```
A = 4×5

     1     2     3     4     5
     6     7     8     9    10
    11    12    13    14    15
    16    17    18    19    20
```

Divide the rows of `A` so that the cell array contains two subarrays. Since the first element of `rowDist` is `1`, the first cell of `C` contains the first row of `A`. The second element of `rowDist` is `3`, so the next cell of `C` contains the next three rows of `A`. The sum of the elements of `rowDist` equals the number of rows of `A`.

```
rowDist = [1 3];
C = mat2cell(A,rowDist)
```

```
C=2×1 cell
    {1x5 double}
    {3x5 double}
```

Display the subarrays.

```
celldisp(C)
```

```
C{1} =

     1     2     3     4     5



C{2} =

     6     7     8     9    10
    11    12    13    14    15
    16    17    18    19    20
```

Input Arguments

A — Input array
array

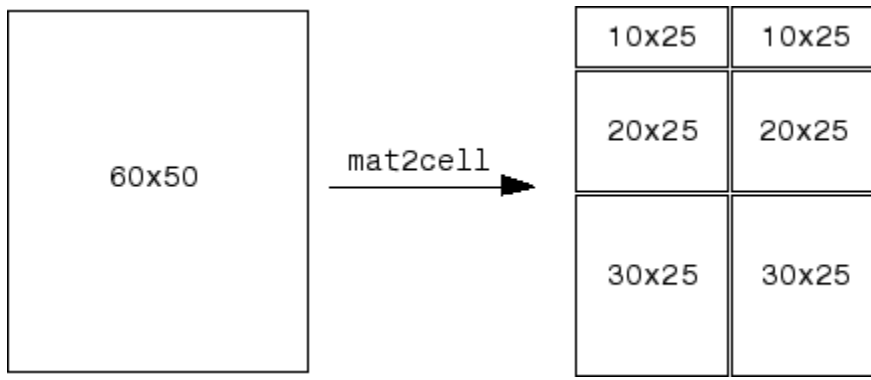
Input array.

dim1Dist,...,dimNDist — Distributions of input array elements
numeric vectors

Vectors describing the distributions of input array elements along each dimension, specified as numeric vectors.

For example, if A is a 60-by-50 array, then you can specify this argument as `[10 20 30], [25 25]` to divide A as shown in the code and figure. C is a cell array that contains the six subarrays split out of A.

```
C = mat2cell(A,[10 20 30],[25 25])
```



For the Kth dimension of A, specify the elements of the corresponding vector `dimKDist` so that `sum(dimKDist)` equals the size of the Kth dimension.

If the Kth dimension of A has a size of zero, then specify the corresponding vector `dimKDist` as the empty array, `[]`, as shown in the code.

```
A = rand(3,0,4);
C = mat2cell(A,[1 2],[],[2 1 1]);
```

rowDist — Distribution by rows
numeric vector

Vector describing the distribution by rows of the input array, specified as a numeric vector. When you do not specify how to divide `A` along any other dimension, the `mat2cell` function returns an `n`-by-1 cell array `C`, where `n` equals the number of elements in `rowDist`.

Each element of `rowDist` specifies the number of rows in the subarray that is in the corresponding cell of `C`. The sum of the elements of `rowDist` must equal the number of rows of `A`.

See Also

`cell2mat` | `num2cell`

Topics

"What Is a Cell Array?"
"Access Data in Cell Array"

Introduced before R2006a

mat2str

Convert matrix to characters

Syntax

```
chr = mat2str(X)
chr = mat2str(X,n)
chr = mat2str( ___ ,'class')
```

Description

`chr = mat2str(X)` converts the numeric matrix `X` into a character vector that represents the matrix, with up to 15 digits of precision.

You can use `chr` as input to the `eval` function. For example, `A = eval(chr)` reproduces the values from the original matrix to the precision specified in `chr`.

`chr = mat2str(X,n)` converts `X` using `n` digits of precision.

`chr = mat2str( ___ ,'class')` includes the name of the class, or data type, of `X` in `chr`. You can use this syntax with any of the arguments from the previous syntaxes.

If you use this syntax to produce `chr`, then `A = eval(chr)` also reproduces the data type of the original matrix.

Examples

Convert Matrix

Convert a numeric matrix to a character vector.

```
chr = mat2str([3.85 2.91; 7.74 8.99])
```

```
chr =
'[3.85 2.91;7.74 8.99]'
```

You can convert `chr` back to a numeric matrix using the `eval` function.

```
A = eval(chr)
```

```
A = 2×2

    3.8500    2.9100
    7.7400    8.9900
```

Specify Precision

Convert a numeric matrix to a character vector, to three digits of precision.

```
chr = mat2str([3.1416 2.7183],3)
```

```
chr =
'[3.14 2.72]'
```

Convert Integers

Create an array of integers and convert it to a character vector. By default, the output of `mat2str` represents an array of doubles. To represent a different numeric type in the output, use the `'class'` input argument.

Create a vector of 16-bit unsigned integers.

```
X = uint16([256 512])
```

```
X = 1x2 uint16 row vector

   256   512
```

Convert `X` to a character vector, including the data type of `X`.

```
chr = mat2str(X,'class')
```

```
chr =
'uint16([256 512])'
```

Convert `chr` back to an array of integers. `A` has the same values and data type as `X`.

```
A = eval(chr)
```

```
A = 1x2 uint16 row vector

   256   512
```

Input Arguments

X — Input array
numeric matrix

Input array, specified as a numeric matrix.

n — Digits of precision
positive integer

Digits of precision, specified as a positive integer.

Tips

- `mat2str` returns character arrays only. Starting in R2016b, you can convert numeric arrays to string arrays using the `string` function.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`cast` | `int2str` | `num2str` | `sprintf` | `str2double` | `str2num` | `string`

Topics

“Convert from Numeric Values to Character Array”
“Convert from Character Arrays to Numeric Values”

Introduced before R2006a

matches

Determine if pattern matches strings

Syntax

```
TF = matches(str,pattern)
TF = matches(str,pattern,'IgnoreCase',true)
```

Description

`TF = matches(str,pattern)` returns `1` (`true`) if the specified pattern matches `str`, and returns `0` (`false`) otherwise.

If `pattern` is an array containing multiple patterns, then `matches` returns `1` if it finds that any element of `pattern` matches `str`.

`TF = matches(str,pattern,'IgnoreCase',true)` ignores case when determining if `pattern` matches `str`.

Examples

Match Pattern

Create a string array.

```
str = ["Mercury","Venus","Earth","Mars"]
```

```
str = 1x4 string array
    "Mercury"    "Venus"    "Earth"    "Mars"
```

Find the strings that match `"Earth"`. Return a logical array where the position of each element equal to `1` corresponds to the position of a matching string in `str`.

```
TF = matches(str,"Earth")
```

```
TF = 1x4 logical array

   0   0   1   0
```

Display the match by indexing back into `str` using `TF`.

```
str(TF)
```

```
ans =
"Earth"
```

Match Multiple Patterns

Create a string array.

```
str = ["Mercury","Venus","Earth","Mars"]
```

```
str = 1x4 string array
    "Mercury"    "Venus"    "Earth"    "Mars"
```

Find elements of `str` that match either `"Venus"` or `"Earth"`.

```
TF = matches(str,["Venus","Earth"])
```

```
TF = 1x4 logical array

   0   1   1   0
```

Display the matches by indexing into `str` using `TF`.

```
str(TF)
```

```
ans = 1x2 string array
    "Venus"    "Earth"
```

Ignore Case

Create a string array.

```
str = ["Mercury","Venus","Earth","Mars"]
```

```
str = 1x4 string array
    "Mercury"    "Venus"    "Earth"    "Mars"
```

Find the element of `str` that matches `"earth"`, ignoring case.

```
TF = matches(str,"earth","IgnoreCase",true)
```

```
TF = 1x4 logical array

   0   0   1   0
```

Display the matching string.

```
str(TF)
```

```
ans =
"Earth"
```

Input Arguments

`str` — Input text
string array | character vector | cell array of character vectors

Input text, specified as a string array, a character vector, or a cell array of character vectors.

`pattern` — Search pattern
string array | character vector | cell array of character vectors

Search pattern, specified as a string array, a character vector, or a cell array of character vectors.

See Also

`contains` | `endsWith` | `find` | `ismember` | `regexp` | `startsWith` | `strcmp` | `strfind`

Topics

"Create String Arrays"
"Search and Replace Text"
"Compare Text"
"Test for Empty Strings and Missing Values"

Introduced in R2019b

matchpairs

Solve linear assignment problem

Syntax

```
M = matchpairs(Cost,costUnmatched)
[M,uR,uC] = matchpairs(Cost,costUnmatched)
[ ___ ] = matchpairs(Cost,costUnmatched,goal)
```

Description

`M = matchpairs(Cost,costUnmatched)` solves the linear assignment problem on page 1-8891 for the rows and columns of the matrix `Cost`. Each row is assigned to a column in such a way that the total cost is minimized. `costUnmatched` specifies the cost per row of not assigning each row, and also the cost per column of not having a row assigned to each column.

`[M,uR,uC] = matchpairs(Cost,costUnmatched)` additionally returns indices for unmatched rows in `uR` and indices for unmatched columns in `uC`.

`[ ___ ] = matchpairs(Cost,costUnmatched,goal)` specifies the goal of the optimization using any of the output argument combinations in previous syntaxes. `goal` can be `'min'` or `'max'` to produce matches that either minimize or maximize the total cost.

Examples

Assign Flights with Minimal Cost

Assign salespeople to flights such that the total cost of transportation is minimized.

A company has four salespeople who need to travel to key cities around the country. The company must book their flights, and wants to spend as little money as possible. These

salespeople are based in different parts of the country, so the cost for them to fly to each city varies.

This table shows the cost for each salesperson to fly to each key city.

	Dallas	Chicago	New York City	St. Louis
Fred	$ 600	$ 670	$ 960	$ 560
Beth	$ 900	$ 280	$ 970	$ 540
Sue	$ 310	$ 350	$ 950	$ 820
Greg	$ 325	$ 290	$ 600	$ 540

Each city represents a sales opportunity. If a city is missed, then the company loses out on an average revenue gain of $2,000.

Create a cost matrix to represent the cost of each salesperson flying to each city.

```
C = [600 670 960 560
     900 280 970 540
     310 350 950 820
     325 290 600 540];
```

Use `matchpairs` to assign the salespeople to the cities with minimal cost. Specify the cost of unassignment as 1000, since the cost of unassignment is counted twice if a row and a column remain unmatched.

```
M = matchpairs(C,1000)
```

```
M = 4×2

     3     1
     2     2
     4     3
     1     4
```

`matchpairs` calculates the least expensive way to get a salesperson to each city.

	Dallas	Chicago	New York City	St. Louis
Fred	$ 600	$ 670	$ 960	**$560**
Beth	$ 900	**$280**	$ 970	$ 540
Sue	**$310**	$ 350	$ 950	$ 820
Greg	$ 325	$ 290	**$600**	$ 540

Unequal Numbers of Rows and Columns

Match rows to columns when you have many more columns than rows in the cost matrix.

Create a 3-by-8 cost matrix. Since you have only three rows, `matchpairs` can produce at most three matches with the eight columns.

```
rng default % for reproducibility
C = randi([10 100], 3, 8)
```

```
C = 3×8

    84    93    35    97    97    22    82    13
    92    67    59    24    54    48    97    87
    21    18    97    98    82    93    69    94
```

Use `matchpairs` to match the rows and columns of the cost matrix. To get the maximum number of matches, use a large cost of unassignment (relative to the magnitude of the entries in the cost matrix). Specify three outputs to return the indices of unmatched rows and columns.

```
[M,uR,uC] = matchpairs(C,1e4)
```

```
M = 3×2

     3     2
     2     4
     1     8


uR =

  0x1 empty double column vector
```

```
uC = 5×1

     1
     3
     5
     6
     7
```

Five of the columns in `C` are not matched with any rows.

Assign Taxis to Maximize Profit

Assign taxis to routes such that the profit is maximized.

A taxi company has several ride requests from across the city. The company wants to dispatch its limited number of taxis in a way that makes the most money.

This table shows the estimated taxi fare for each of five ride requests. Only three of the five ride requests can be filled.

	Ride 1	Ride 2	Ride 3	Ride 4	Ride 5
Cab A	$ 5.70	$ 6.30	$ 3.10	$ 4.80	$ 3.50
Cab B	$ 5.80	$ 6.40	$ 3.30	$ 4.70	$ 3.20
Cab C	$ 5.70	$ 6.30	$ 3.20	$ 4.90	$ 3.40

Create a profits matrix to represent the profits of each taxi ride.

```
P = [5.7 6.3 3.1 4.8 3.5
     5.8 6.4 3.3 4.7 3.2
     5.7 6.3 3.2 4.9 3.4];
```

Use `matchpairs` to match the taxis to the most profitable rides. Specify three outputs to return any unmatched rows and columns, and the `'max'` option to maximize the profits. Specify the cost of unassignment as zero, since the company makes no money from unfilled taxis or ride requests.

```
costUnmatched = 0;
[M,uR,uC] = matchpairs(P,costUnmatched,'max')
```

```
M = 3×2

     1     1
     2     2
     3     4


uR =

  0x1 empty double column vector

uC = 2×1

     3
     5
```

`matchpairs` calculates the most profitable rides to fill. The solution leaves ride requests 3 and 5 unfilled.

	Ride 1	Ride 2	Ride 3	Ride 4	Ride 5
Cab A	**$ 5.70**	$ 6.30	$ 3.10	$ 4.80	$ 3.50
Cab B	$ 5.80	**$ 6.40**	$ 3.30	$ 4.70	$ 3.20
Cab C	$ 5.70	$ 6.30	$ 3.20	**$ 4.90**	$ 3.40

Calculate the total profits for the calculated solution. Since `costUnmatched` is zero, you only need to add together the profits from each match.

```
TotalProfits = sum(P(sub2ind(size(P), M(:,1), M(:,2))))
```

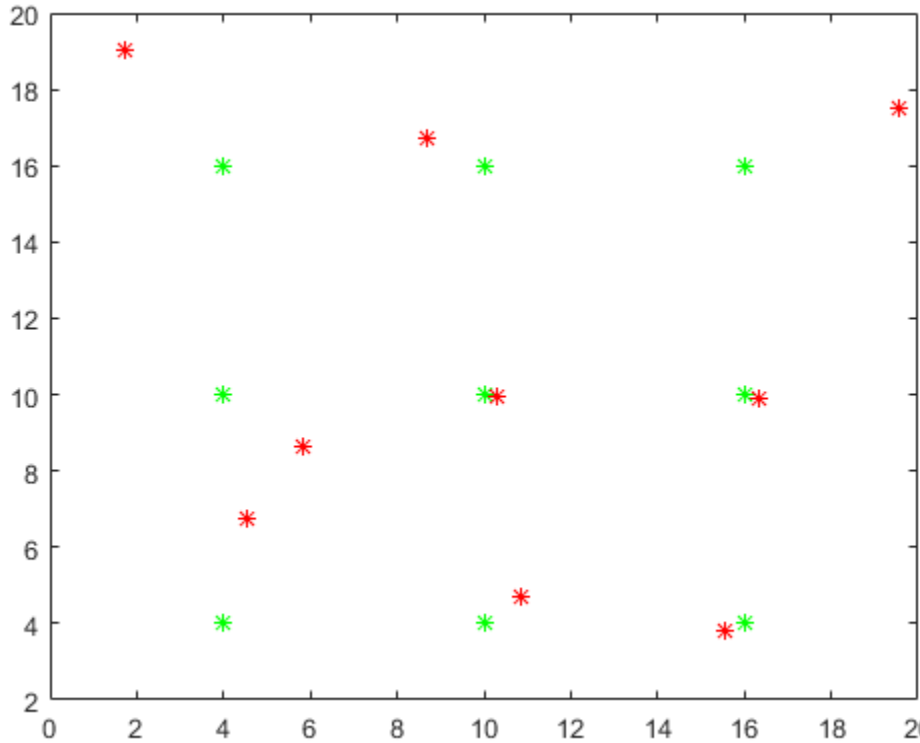
```
TotalProfits = 17
```

Track Point Positions Over Time

Use `matchpairs` to track the movement of several points by minimizing the total changes in distance.

Plot a grid of points at time $t = 0$ in green. At time $t = 1$, some of the points move a small amount in a random direction.

```
[x,y] = meshgrid(4:6:16);
x0 = x(:)';
y0 = y(:)';
plot(x0,y0,'g*')
hold on
rng default % for reproducibility
x1 = x0 + randn(size(x0));
y1 = y0 + randn(size(y0));
plot(x1,y1,'r*')
```



Use `matchpairs` to match the points at $t = 0$ with the points at $t = 1$. To do this, first calculate a cost matrix where `C(i,j)` is the Euclidean distance from point `i` to point `j`.

```
C = zeros(size(x).^2);
for k = 1:length(y1)
  C(k,:) = vecnorm([x1(k)-x0; y1(k)-y0],2,1)';
end
C
```

```
C = 9×9

    2.8211    3.2750    9.2462    6.1243    6.3461   10.7257   11.7922   11.9089   14.7
    4.9987    2.2771    7.5752    6.2434    4.3794    8.4485   11.1792   10.2553   12.5
   15.2037    9.3130    3.7833   17.1539   12.2408    8.7988   20.7211   16.8803   14.5
    6.9004    8.6551   13.1987    1.1267    5.3446   11.3075    5.1888    7.3633   12.3
    8.6703    6.3191    8.7571    5.9455    0.3249    6.0714    8.2173    5.6816    8.3
   13.5530    8.1918    4.7464   12.7818    6.8409    1.4903   14.6652    9.9242    7.3
   11.5682   13.1257   16.8150    5.5702    8.3359   13.4144    0.4796    6.2201   12.2
   13.6699   12.3432   13.7784    8.6461    6.3438    8.8167    5.8858    0.3644    6.1
   20.6072   17.2853   15.6495   16.5444   12.1590    9.6935   13.9562    8.3006    3.8
```

Next, use `matchpairs` to match the rows and columns in the cost matrix. Specify the cost of unassignment as 1. With such a low cost of unassignment relative to the entries in the cost matrix, it is likely `matchpairs` will leave some points unmatched.

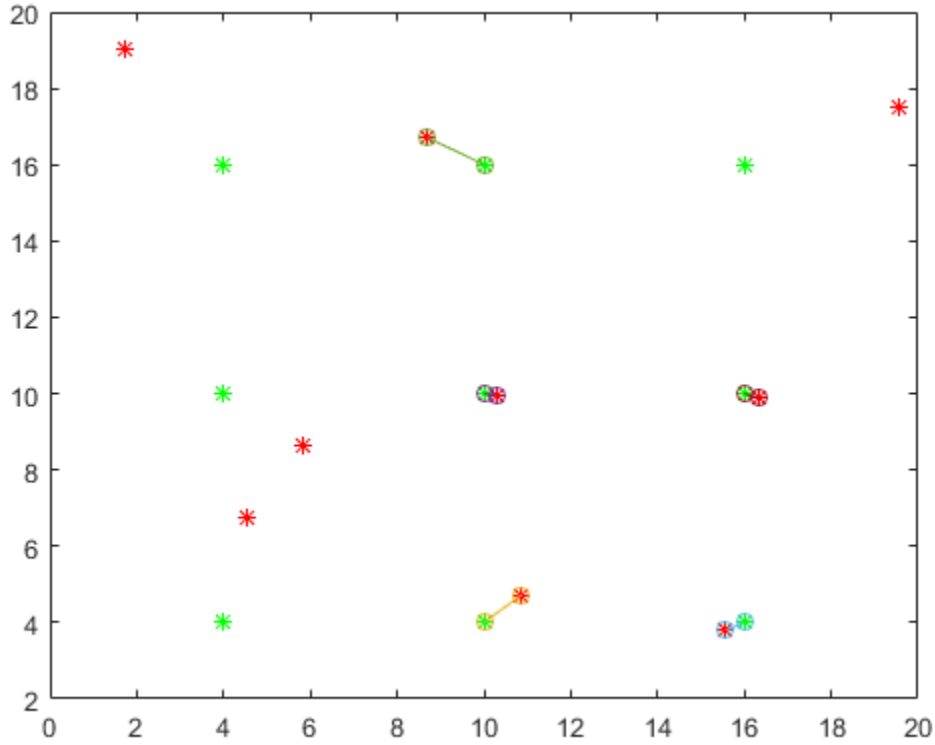
```
M = matchpairs(C,1)
```

```
M = 5×2

     4     4
     5     5
     6     6
     7     7
     8     8
```

The values `M(:,2)` correspond to the original points (x_0, y_0), while the values `M(:,1)` correspond to the moved points (x_1, y_1).

Plot the matched pairs of points. The points that moved farther than `2*costUnmatched` away from the original point remain unmatched.

```
xc = [x0(M(:,2)); x1(M(:,1))];
yc = [y0(M(:,2)); y1(M(:,1))];
plot(xc,yc,'-o')
```



Input Arguments

`Cost` — Cost matrix
matrix

Cost matrix. Each entry `Cost(i,j)` specifies the cost of assigning row `i` to column `j`.

Data Types: `single` | `double`

`costUnmatched` — Cost of not matching
scalar

Cost of not matching, specified as a scalar. `matchpairs` compares the value of `2*costUnmatched` to the entries in `Cost` to determine whether it is more beneficial for a row or column to remain unmatched. Use this parameter to make matches more or less likely in the algorithm. For more information, see linear assignment problem on page 1-8891.

Example: `M = matchpairs(C,10)` specifies a cost of 10 for not matching a row or column of `C`.

Data Types: `single` | `double`

`goal` — Optimization goal
`'min'` (default) | `'max'`

Optimization goal, specified as either `'min'` or `'max'`. The optimization goal specifies whether the total cost should be minimized or maximized.

Example: `M = matchpairs(Cost,costUnmatched,'max')` specifies that the rows and columns of `Cost` should be matched together to maximize the total cost.

Output Arguments

`M` — Matches
matrix

Matches, returned as a matrix. `M` is a `p`-by-2 matrix, where `M(i,1)` and `M(i,2)` are the row and column indices of a matched pair in the cost matrix. The rows of `M` are sorted with the second column in ascending order.

- Each row and column can be matched a single time only, so each `M(i,1)` value and each `M(i,2)` value is unique.
- `M` contains `p` matches, and `p` is less than or equal to the maximum number of matches `min(size(Cost))`.
- The cost of the matches in `M` is `sum([Cost(M(1,1),M(1,2)), Cost(M(2,1),M(2,2)), ..., Cost(M(p,1),M(p,2))])`.

`uR` — Unassigned rows
column vector

Unassigned rows, returned as a column vector of indices. The entries in `uR` indicate which rows in `Cost` are unassigned. Each entry in `uR` and `uC` contributes to the total cost of the solution according to `costUnassigned`.

`uC` — Unassigned columns
column vector

Unassigned columns, returned as a column vector of indices. The entries in `uC` indicate which columns in `Cost` are unassigned. Each entry in `uR` and `uC` contributes to the total cost of the solution according to `costUnassigned`.

More About

Linear Assignment Problem

The *linear assignment problem* is a way of assigning rows to columns such that each row is assigned to a column and the total cost of the assignments is minimized (or maximized). The cost of assigning each row to each column is captured in a *cost matrix*. The entry `Cost(i,j)` is the cost of assigning row `i` to column `j`.

The *cost of unassignment* assigns a cost to any row or column that is not matched. This practice allows for minimum-cost solutions that do not assign all rows or columns. If a row and column are not matched, this increases the total cost by `2*costUnmatched`.

The total cost of a solution `M` is the sum of the cost of all matched pairs added to the cost of all unmatched pairs:

$$TC = \sum_{i=1}^{p} \text{Cost}(M(i,1), M(i,2)) + \text{costUnmatched} \cdot (m + n - 2p)$$

In code the total cost is

```
CostAssigned = sum(Cost(sub2ind(size(Cost), M(:,1), M(:,2))));
CostUnassigned = costUnmatched*(sum(size(Cost))-2*size(M,1));
TotalCost = CostAssigned + CostUnassigned;
```

- `Cost` is an `m`-by-`n` matrix.
- `M` is a `p`-by-2 matrix, where `M(i,1)` and `M(i,2)` are the row and column of a matched pair.

- `(m+n-2*p)` is the total number of *unmatched* rows and columns.

References

[1] Duff, I.S. and J. Koster. "On Algorithms For Permuting Large Entries to the Diagonal of a Sparse Matrix." *SIAM J. Matrix Anal. & Appl.* 22(4), 2001. pp 973–996.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

See Also

`dmperm` | `equilibrate` | `sprank`

Introduced in R2019a

material

Control reflectance properties of surfaces and patches

Syntax

```
material shiny
material dull
material metal
material([ka kd ks])
material([ka kd ks n])
material([ka kd ks n sc])
material default
material(obj,...)
```

Description

`material` sets the lighting characteristics of `surface` and `patch` objects.

`material shiny` sets the reflectance properties so that the object has a high specular reflectance relative to the diffuse and ambient light, and the color of the specular light depends only on the color of the light source.

`material dull` sets the reflectance properties so that the object reflects more diffuse light and has no specular highlights, but the color of the reflected light depends only on the light source.

`material metal` sets the reflectance properties so that the object has a very high specular reflectance, very low ambient and diffuse reflectance, and the color of the reflected light depends on both the color of the light source and the color of the object.

`material([ka kd ks])` sets the ambient/diffuse/specular strength of the objects.

`material([ka kd ks n])` sets the ambient/diffuse/specular strength and specular exponent of the objects.

`material([ka kd ks n sc])` sets the ambient/diffuse/specular strength, specular exponent, and specular color reflectance of the objects.

`material default` sets the ambient/diffuse/specular strength, specular exponent, and specular color reflectance of the objects to their defaults.

`material(obj,...)` operates on the surface and patch objects specified in vector `obj`. If `obj` includes a graphics object that is not a surface or patch object, then `material` operates on any surface or patch objects that are descendants of the specified object.

Tips

The `material` command sets the `AmbientStrength`, `DiffuseStrength`, `SpecularStrength`, `SpecularExponent`, and `SpecularColorReflectance` properties of all `surface` and `patch` objects in the axes. There must be visible `light` objects in the `axes` for lighting to be enabled. Look at the `materal.m` file to see the actual values set (enter the command `type material`).

See Also

Topics

"Lighting Overview"

Introduced before R2006a

matlab.addons.disableAddon

Disable installed add-on

Syntax

```
matlab.addons.disableAddon(name)
matlab.addons.disableAddon(identifier)
matlab.addons.disableAddon( ___ ,version)
```

Description

`matlab.addons.disableAddon(name)` disables the installed add-on with the specified name. If multiple versions of an add-on are installed, `matlab.addons.disableAddon` disables the currently enabled version.

You can disable and enable most add-ons from the MATLAB user community. Disabling and enabling MathWorks products and support packages is not supported.

`matlab.addons.disableAddon(identifier)` disables the installed add-on with the specified identifier.

`matlab.addons.disableAddon( ___ ,version)` also specifies the version of the add-on to disable. Use this syntax with any of the arguments in previous syntaxes.

Examples

Disable Add-On from Installed List

Suppose that you have an add-on called `Random File Name Creator` installed on your system. Get the list of installed add-ons and disable the `Random File Name Creator` add-on.

Get the list of installed add-ons.

```
addons = matlab.addons.installedAddons
```

```
addons = 

  1×4 table 

                Name                    Version     Enabled                   Identifier                   
    ________________________________    _________    _______    ____________________________________________
    "Random File Name Creator"          "1.0"         true      "75442144-f751-4011-bm0e-32b6fb2f1433"
```

Disable `Random File Name Creator` and confirm that it is disabled.

```
matlab.addons.disableAddon('Random File Name Creator')
matlab.addons.isAddonEnabled('Random File Name Creator')
```

```
ans = 

  logical 

   0 
```

Input Arguments

`name` — Name of add-on
character vector | string

Name of the add-on that you want to disable, specified as a character vector or string. To determine the name of an add-on, use the `matlab.addons.installedAddons` function.

`identifier` — Unique identifier of add-on
character vector | string

Unique identifier of the add-on that you want to disable, specified as a character vector or string. To determine the unique identifier of an add-on, use the `matlab.addons.installedAddons` function.

`version` — Version of add-on
character vector | string

Version of the add-on that you want to disable, specified as a character vector or string. To determine the version of an add-on, use the `matlab.addons.installedAddons` function.

See Also

`matlab.addons.enableAddon` | `matlab.addons.installedAddons` | `matlab.addons.isAddonEnabled`

Topics

"Get and Manage Add-Ons"

Introduced in R2017b

matlab.addons.enableAddon

Enable installed add-on

Syntax

```
matlab.addons.enableAddon(name)
matlab.addons.enableAddon(identifier)
matlab.addons.enableAddon( ___ ,version)
```

Description

`matlab.addons.enableAddon(name)` enables the add-on with the specified name. You can disable and enable most add-ons from the MATLAB user community. Disabling and enabling MathWorks products and support packages is not supported.

If multiple versions of an add-on are installed, you must specify a `version`.

`matlab.addons.enableAddon(identifier)` enables the add-on with the specified identifier.

`matlab.addons.enableAddon( ___ ,version)` also specifies the version of the add-on to enable. Use this syntax with any of the arguments in previous syntaxes.

Examples

Enable Add-On from Installed List

Suppose that you have an add-on called `Random File Name Creator` installed on your system. Get the list of installed add-ons and enable the `Random File Name Creator` add-on.

Get the list of installed add-ons.

```
addons = matlab.addons.installedAddons
```

```
addons =

  1×4 table

            Name                    Version     Enabled                  Identifier
    ________________________________    ________    _______    ____________________________________

    "Random File Name Creator"     "1.0"         false      "75442144-f751-4011-bm0e-32b6fb2f1433"
```

Check to see whether the `Random File Name Creator` add-on is disabled. MATLAB confirms that the add-on is disabled.

```
matlab.addons.isAddonEnabled('Random File Name Creator')
```

```
ans =

  logical

   0
```

Enable the `Random File Name Creator` add-on and confirm that it is enabled.

```
matlab.addons.enableAddon('Random File Name Creator')
matlab.addons.isAddonEnabled('Random File Name Creator')
```

```
ans =

  logical

   1
```

Switch to Previous Version of Add-On

Suppose that you have versions `2.0` and `4.0` of an add-on called `My Toolbox` installed on your system. Version `4.0` is enabled. Switch to using version `2.0` of the add-on by enabling version `2.0`.

Get the list of installed add-ons.

```
addons = matlab.addons.installedAddons
```

```
addons =

  2×4 table

            Name                    Version     Enabled                  Identifier
    ________________________________    ________    _______    ____________________________________
```

```
    "My Toolbox_v4.0"              "4.0"         true      "6de8682e-9c3c-407e-bad7-aa103d738d08"
    "My Toolbox"                   "2.0"         false     "6de8682e-9c3c-407e-bad7-aa103d738d08"
```

Enable version `2.0` of `My Toolbox` by specifying the second item in the returned list of identifiers. MATLAB enables version `2.0` and disables version `4.0` of `My Toolbox`.

```
matlab.addons.enableAddon(addons.Identifier(2),'2.0')
addons = matlab.addons.installedAddons
```

```
addons =

  2×4 table

            Name                    Version     Enabled                  Identifier
    ________________________________    _________    _______    ____________________________________

    "My Toolbox_v4.0"               "4.0"         false     "6de8682e-9c3c-407e-bad7-aa103d738d08"
    "My Toolbox"                    "2.0"         true      "6de8682e-9c3c-407e-bad7-aa103d738d08"
```

Input Arguments

`name` — Name of add-on

character vector | string

Name of the add-on that you want to enable, specified as a character vector or string. To determine the name of an add-on, use the `matlab.addons.installedAddons` function.

`identifier` — Unique identifier of add-on

character vector | string

Unique identifier of the add-on that you want to enable, specified as a character vector or string. To determine the unique identifier of an add-on, use the `matlab.addons.installedAddons` function.

`version` — Version of add-on

character vector | string

Version of the add-on that you want to disable, specified as a character vector or string. To determine the version of an add-on, use the `matlab.addons.installedAddons` function.

See Also

`matlab.addons.disableAddon` | `matlab.addons.installedAddons` | `matlab.addons.isAddonEnabled`

Topics

"Get and Manage Add-Ons"

Introduced in R2017b

matlab.addons.install

Install add-on

Syntax

```
newAddon = matlab.addons.install(fileName)
newAddon = matlab.addons.install(fileName,agreeToLicense)
newAddon = matlab.addons.install(fileName,installOption)
newAddon = matlab.addons.install(fileName,agreeToLicense,
installOption)
```

Description

`newAddon = matlab.addons.install(fileName)` installs the add-on specified by `fileName`. If a version of the add-on is already installed, MATLAB overwrites the previous version.

`matlab.addons.install` only supports installing toolboxes.

`newAddon = matlab.addons.install(fileName,agreeToLicense)` accepts the license agreement before installing the add-on if `agreeToLicense` is `true`.

`newAddon = matlab.addons.install(fileName,installOption)` installs the add-on without overwriting previously installed versions if `installOption` is `'add'`. Otherwise, if a version of the add-on is already installed, MATLAB overwrites the previous version.

`newAddon = matlab.addons.install(fileName,agreeToLicense, installOption)` installs the add-on using the specified license and install options.

Examples

Accept License Agreement and Install Toolbox

Assume that you have the `My toolbox.mltbx` toolbox file in your `C:\Downloads\` folder, and that the toolbox contains a license agreement. Install the toolbox and prevent MATLAB from opening the license agreement dialog box by indicating you accept the license.

```
toolboxFile = 'C:\Downloads\My toolbox.mltbx';
agreeToLicense = true;
matlab.addons.toolbox.install(toolboxFile,agreeToLicense)
```

Verify that the toolbox is installed.

```
addons = matlab.addons.installedAddons
```

```
addons =

  1×4 table

            Name                   Version     Enabled                   Identifier
    ___________________________    ________    _______    ____________________________________

    "My Toolbox"                   "2.0"        true      "6de8682e-9c3c-407e-bad7-aa103d738d08"
```

Install Toolbox Without Overwriting Previous Versions

Suppose that you have version `2.0` of a toolbox called `My Toolbox` installed on your system. Install version `4.0` of the toolbox without overwriting version `2.0`.

Assume that you have the `'My toolbox_v4.0.mltbx'` toolbox file in your current working folder. Install the toolbox without overwriting the existing installed version by specifying that you want to add the toolbox.

```
toolboxFile = 'My toolbox_v4.0.mltbx';
installOption = 'add';
matlab.addons.toolbox.install(toolboxFile, installOption)
```

```
ans =

  1×4 table

          Name            Version    Enabled                   Identifier
    ________________      _______    _______    ____________________________________

    "My toolbox_v4.0"      "4.0"      true      "6de8682e-9c3c-407e-bad7-aa103d738d08"
```

If the toolbox contains a license agreement, a dialog box opens to prompt you to agree to the license before installation.

Verify that the previous version of the toolbox is still installed.

```
addons = matlab.addons.installedAddons
```

```
addons = 

  2×4 table

            Name                    Version     Enabled                   Identifier
    ________________________________ _________ _________ ______________________________________
    "My Toolbox_v4.0"               "4.0"      true      "6de8682e-9c3c-407e-bad7-aa103d738d08"
    "My Toolbox"                    "2.0"      false     "6de8682e-9c3c-407e-bad7-aa103d738d08"
```

Input Arguments

`fileName` — File name

character vector | string

File name to install, specified as a character vector or string. File name must be a valid toolbox installation file (`*.mltbx`). You can specify the file name as an absolute or relative path.

`agreeToLicense` — Whether to accept license agreement

`false` (default) | `true`

Whether to accept license agreement, specified as `true` or `false`. By default, `agreeToLicense` is `false`.

If an add-on contains a license agreement:

- If `agreeToLicense` is `false`, then MATLAB displays a dialog box where you can accept the license or cancel installation.
- If `agreeToLicense` is `true`, then MATLAB installs the add-on without opening the license agreement dialog box. By setting `agreeToLicense` to `true`, you accept the terms of the license agreement. Be sure that you have reviewed the license agreement before installing the add-on.

If an add-on does not have a license agreement, the value of `agreeToLicense` has no effect on installation.

`installOption` — Whether to overwrite or add the add-on

`'overwrite'` (default) | `'add'`

Whether to overwrite or add the add-on, specified as `'overwrite'` or `'add'`. By default, `installOption` is set to `'overwrite'`.

If a version of an add-on is already installed:

- If `installOption` is `'overwrite'`, then MATLAB overwrites all previously installed versions of the add-on.
- If `installOption` is `'add'`, then MATLAB installs the add-on without overwriting previously installed versions.

Output Arguments

newAddon — New add-on information
table

New add-on information, specified as a table of strings. The table has these columns.

Field	Description
Name	Name of the add-on
Version	Version of the add-on
Enabled	Whether the add-on is enabled
Identifier	Unique identifier of the add-on

See Also

`matlab.addons.installedAddons` | `matlab.addons.uninstall`

Topics

"Get and Manage Add-Ons"

Introduced in R2018b

matlab.addons.installedAddons

Get list of installed add-ons

Syntax

```
addons = matlab.addons.installedAddons
```

Description

`addons = matlab.addons.installedAddons` returns a list of currently installed add-ons.

Examples

Get List of Installed Add-Ons

Suppose that you have an add-on called `Random File Name Creator` and versions `2.0` and `4.0` of an add-on called `My Toolbox` installed on your system. Get the list of installed add-ons.

```
addons = matlab.addons.installedAddons
```

```
addons =

  1×4 table

            Name                    Version     Enabled                   Identifier
    ____________________________    _________    _______    ________________________________________

    "Random File Name Creator"      "1.0"         true      "75442144-f751-4011-bm0e-32b6fb2f1433"
```

```
"My Toolbox_v4.0"              "4.0"         true      "6de8682e-9c3c-407e-bad7-aa103d738d08"
"My Toolbox"                   "2.0"         false     "6de8682e-9c3c-407e-bad7-aa103d738d08"
```

Output Arguments

addons — Installed add-ons information
table

Installed add-ons information, specified as a table of strings with one row for each add-on. The table has these columns.

Field	Description
Name	Name of the add-on
Version	Version of the add-on
Enabled	Whether the add-on is enabled
Identifier	Unique identifier of the add-on

See Also

`matlab.addons.disableAddon` | `matlab.addons.enableAddon` | `matlab.addons.isAddonEnabled`

Topics

"Get and Manage Add-Ons"

Introduced in R2017b

matlab.addons.isAddonEnabled

Determine if add-on is enabled

Syntax

```
isenabled = matlab.addons.isAddonEnabled(name)
isenabled = matlab.addons.isAddonEnabled(identifier)
isenabled = matlab.addons.isAddonEnabled( ___ ,version)
```

Description

`isenabled = matlab.addons.isAddonEnabled(name)` returns `true` if the add-on with the specified name is enabled. Otherwise, it returns `false`.

If multiple versions of an add-on are installed, you must specify a `version`.

`isenabled = matlab.addons.isAddonEnabled(identifier)` returns `true` if the add-on with the specified identifier is enabled. Otherwise, it returns `false`.

`isenabled = matlab.addons.isAddonEnabled( ___ ,version)` also specifies the version of the add-on. Use this syntax with any of the arguments in previous syntaxes.

Examples

Determine If Add-On Is Enabled

Suppose that you have an add-on called `Random File Name Creator` installed on your system. Get the list of installed add-ons and determine if `Random File Name Creator` is enabled.

Get the list of installed add-ons.

```
addons = matlab.addons.installedAddons
```

```
addons =

  1×4 table

            Name                    Version     Enabled                   Identifier
    ___________________________     _______     _______     ____________________________________

    "Random File Name Creator"      "1.0"        true       "75442144-f751-4011-bm0e-32b6fb2f1433"
```

Determine if `Random File Name Creator` is enabled. MATLAB confirms that the add-on is enabled.

```
matlab.addons.isAddonEnabled('Random File Name Creator')
```

```
ans =

  logical

   1
```

Disable Specified Version of Add-On

Suppose that you have versions `2.0` and `4.0` of an add-on called `My Toolbox` installed on your system. Disable version `4.0` of the add-on and confirm that it is disabled.

Get the list of installed add-ons.

```
addons = matlab.addons.installedAddons
```

```
addons =

  2×4 table

            Name                    Version     Enabled                   Identifier
    ___________________________     _______     _______     ____________________________________

    "My Toolbox_v4.0"               "4.0"        true       "6de8682e-9c3c-407e-bad7-aa103d738d08"
    "My Toolbox"                    "2.0"        false      "6de8682e-9c3c-407e-bad7-aa103d738d08"
```

Disable `My Toolbox` version `4.0` by specifying the first item in the returned list of identifiers. Then, confirm that it is disabled.

```
matlab.addons.disableAddon(addons.Identifier(1),'4.0')
matlab.addons.isAddonEnabled(addons.Identifier(1),'4.0')
```

```
ans =

  logical
```

```
0
```

Input Arguments

`name` — Name of add-on
character vector | string

Name of the add-on, specified as a character vector or string. To determine the name of an add-on, use the `matlab.addons.installedAddons` function.

`identifier` — Unique identifier of add-on
character vector | string

Unique identifier of the add-on, specified as a character vector or string. To determine the unique identifier of an add-on, use the `matlab.addons.installedAddons` function.

`version` — Version of add-on
character vector | string

Version of the add-on, specified as a character vector or string. To determine the version of an add-on, use the `matlab.addons.installedAddons` function.

See Also

`matlab.addons.disableAddon` | `matlab.addons.enableAddon` | `matlab.addons.installedAddons`

Topics

"Get and Manage Add-Ons"

Introduced in R2017b

matlab.addons.toolbox.installedToolboxes

Return information about installed toolboxes

Syntax

```
toolboxes = matlab.addons.toolbox.installedToolboxes
```

Description

`toolboxes = matlab.addons.toolbox.installedToolboxes` returns a structure array containing information about the installed toolboxes. The structure array does not include information about MathWorks toolboxes.

Examples

Display Installed Toolbox Information

Assume that you previously installed two toolboxes, `myToolbox` and `myOtherToolbox`. Store information about these toolboxes in a variable.

```
toolboxes = matlab.addons.toolbox.installedToolboxes
```

```
toolboxes = 

1x2 struct array with fields:

    Name
    Version
    Guid
```

Use `struct2table` to format this stored information as a readable table.

```
struct2table(toolboxes)
```

```
ans = 
```

```
       Name          Version                    Guid
_______________      _______    ________________________________________

'myToolbox'          '1.5.2'    'd0169b4a-fe74-463f-981a-26160c94cbe5'
'myOtherToolbox'     '1.0'      '1deb72c1-725f-4e1b-a1a7-dcc8e75453bb'
```

Output Arguments

`toolboxes` — Information about installed toolboxes
structure array

Information about installed toolboxes, returned as a structure array with one element for each installed toolbox. Each element of the structure array has the following fields.

Field	Description
`Name`	Name of the toolbox
`Version`	Toolbox version
`Guid`	Unique toolbox identifier

Tips

- To view information about MathWorks toolboxes, see `ver`.

Alternatives

You can query your installed toolboxes from the Add-On Manager UI. For more information, see “Get and Manage Add-Ons”.

See Also

`matlab.addons.toolbox.installToolbox` | `matlab.addons.toolbox.packageToolbox` | `matlab.addons.toolbox.toolboxVersion` | `matlab.addons.toolbox.uninstallToolbox`

Introduced in R2016a

matlab.addons.toolbox.installToolbox

Install toolbox file

Syntax

```
installedToolbox = matlab.addons.toolbox.installToolbox(toolboxFile)
installedToolbox = matlab.addons.toolbox.installToolbox(toolboxFile,
agreeToLicense)
```

Description

`installedToolbox = matlab.addons.toolbox.installToolbox(toolboxFile)` installs the toolbox file (`.mltbx` file) specified by `toolboxFile` and returns information about the toolbox.

`installedToolbox = matlab.addons.toolbox.installToolbox(toolboxFile, agreeToLicense)` accepts the license agreement before installing the toolbox if `agreeToLicense` is `true`.

Examples

Install Toolbox

Assume that you have the `myToolbox.mltbx` toolbox file in your current working folder. Install the toolbox.

```
toolboxFile = 'MyToolbox.mltbx';
installedToolbox = matlab.addons.toolbox.installToolbox(toolboxFile)
```

```
installedToolbox = 

       Name: 'myToolbox'
    Version: '2.0'
       Guid: 'd0169b4a-fe74-463f-981a-26160c94cbe5'
```

If the toolbox contains a licenses agreement, a dialog box opens to prompt you to agree to the license before installation.

Accept License Agreement and Install Toolbox

Assume that you have the `myToolbox.mltbx` toolbox file in your `C:\Downloads\` folder, and that the toolbox contains a license agreement. Install the toolbox and prevent MATLAB from opening the license agreement dialog box by indicating you accept the license.

```
toolboxFile = 'C:\Downloads\MyToolbox.mltbx';
agreeToLicense = true;
installedToolbox = matlab.addons.toolbox.installToolbox(toolboxFile,agreeToLicense)
```

```
installedToolbox = 

       Name: 'myToolbox'
    Version: '2.0'
       Guid: 'd0169b4a-fe74-463f-981a-26160c94cbe5'
```

Input Arguments

`toolboxFile` — Name of toolbox file
character vector | string scalar

Name of the toolbox file (`.mltbx` file), specified as a character vector or string scalar. The name includes the relative or absolute path to the file.

Example: `toolboxFile = 'myToolbox.mltbx'`

Example: `toolboxFile = "C:\Work\myOtherToolbox.mltbx"`

`agreeToLicense` — Indicator to accept license agreement
`false` (default) | `true`

Indicator to accept license agreement, specified as `false` or `true`. By default, `agreeToLicense` is `false`. If a toolbox contains a license agreement:

- If `agreeToLicense` is `false`, then MATLAB displays a dialog box where you can accept the license or cancel installation.
- If `agreeToLicense` is `true`, then MATLAB installs the toolbox without opening the license agreement dialog box. By setting `agreeToLicense` to `true`, you accept the

terms of the license agreement. Be sure that you have reviewed the license agreement before installing the toolbox.

If a toolbox does not have a license agreement, the value of `agreeToLicense` has no effect on installation.

Data Types: `logical`

Output Arguments

`installedToolbox` — Information about installed toolbox
structure

Information about the installed toolbox, returned as a structure. The structure has the following fields.

Field	Description
`Name`	Name of the toolbox
`Version`	Toolbox version
`Guid`	Unique toolbox identifier

Alternatives

You can install toolboxes from the Add-On Explorer UI. For more information, see “Get and Manage Add-Ons”.

See Also

`matlab.addons.toolbox.installedToolboxes` | `matlab.addons.toolbox.packageToolbox` | `matlab.addons.toolbox.toolboxVersion` | `matlab.addons.toolbox.uninstallToolbox`

Introduced in R2016a

matlab.addons.toolbox.packageToolbox

Package toolbox project

Syntax

```
matlab.addons.toolbox.packageToolbox(projectFile)
matlab.addons.toolbox.packageToolbox(projectFile,outputFile)
```

Description

`matlab.addons.toolbox.packageToolbox(projectFile)` packages the toolbox project file (`.prj` file) into a MATLAB toolbox file (`.mltbx` file) of the same name. For you to package a toolbox, the toolbox root folder and the toolbox files must be in the same location as when you created the toolbox project file.

`matlab.addons.toolbox.packageToolbox(projectFile,outputFile)` packages the toolbox and saves the `.mltbx` file with the name and location specified by `outputFile`.

Examples

Package Toolbox

Assume that you have the `myToolbox.prj` toolbox project file in your current working folder. Package the toolbox in the same folder.

```
projectFile = 'myToolbox.prj';
matlab.addons.toolbox.packageToolbox(projectFile)
```

Specify Output File Name and Package Toolbox

Assume that you have the `myToolbox.prj` toolbox project file in your current working folder. Package the toolbox as `myFavoriteToolbox.mltbx`.

```
projectFile = 'myToolbox.prj';
outputFile = 'myFavoriteToolbox';
matlab.addons.toolbox.packageToolbox(projectFile,outputFile)
```

Input Arguments

`projectFile` — Name of toolbox project file

character vector | string scalar

Name of the toolbox project file (`.prj` file), specified as a character vector or string scalar. The name includes the relative or absolute path to the file.

Example: `projectFile = 'myToolbox.prj'`

Example: `projectFile = "C:\Work\myOtherToolbox.prj"`

`outputFile` — Name of output MATLAB toolbox file

character vector | string scalar

Name of the output MATLAB toolbox file (`.mltbx` file), specified as a character vector or string scalar. The name includes the relative or absolute path to the file. If the value of `outputFile` does not contain the `.mltbx` extension, the `packageToolbox` function appends the extension.

Example: `outputFile = "myToolbox.mltbx"`

Example: `outputFile = 'C:\Work\myOtherToolbox'`

Alternatives

You can package toolboxes from the Package a Toolbox UI. For more information, see “Create and Share Toolboxes”.

See Also

`matlab.addons.toolbox.installToolbox` | `matlab.addons.toolbox.installedToolboxes` | `matlab.addons.toolbox.toolboxVersion` | `matlab.addons.toolbox.uninstallToolbox`

Introduced in R2016a

matlab.addons.toolbox.toolboxVersion

Query or modify version of toolbox

Syntax

```
currentVersion = matlab.addons.toolbox.toolboxVersion(toolboxFile)
previousVersion = matlab.addons.toolbox.toolboxVersion(toolboxFile,
newVersion)
```

Description

`currentVersion = matlab.addons.toolbox.toolboxVersion(toolboxFile)` queries `toolboxFile` for the current version number. You do not need to install the toolbox to query the version number.

`previousVersion = matlab.addons.toolbox.toolboxVersion(toolboxFile, newVersion)` sets the toolbox version number to `newVersion` and returns the previous version number.

Examples

Query Toolbox Version Number

Assume that you have the `myToolbox.mltbx` toolbox file on your MATLAB search path. Query the version number.

```
toolboxFile = 'myToolbox.mltbx';
currentVersion = matlab.addons.toolbox.toolboxVersion(toolboxFile)
```

```
currentVersion =

1.5.2
```

Set Toolbox Version Number

Assume that you have the `myToolbox.prj` toolbox project file on your MATLAB search path. Update the `myToolbox` version number to `2.0`.

```
toolboxFile = 'myToolbox.prj';
newVersion = '2.0';
previousVersion = matlab.addons.toolbox.toolboxVersion(toolboxFile,newVersion)
```

```
previousVersion =

1.5.2
```

Verify that the version number is now `2.0`.

```
currentVersion = matlab.addons.toolbox.toolboxVersion(toolboxFile)
```

```
currentVersion =

2.0
```

The version number is updated for the toolbox project file, but not for the MATLAB toolbox file (`.mltbx`).

```
matlab.addons.toolbox.toolboxVersion('myToolbox.mltbx')
```

```
ans =

1.5.2
```

Repackage the toolbox to update the `.mltbx` file. For more information, see `matlab.addons.toolbox.packageToolbox`.

Input Arguments

`toolboxFile` — Name of toolbox file
character vector | string scalar

Name of the toolbox file, specified as a character vector or string scalar. The name includes the relative or absolute path to the toolbox file.

If you are only querying the toolbox version, `toolboxFile` can be a toolbox project file (`.prj`) or a MATLAB toolbox file (`.mltbx`).

If you are setting the toolbox version, `toolboxFile` must be a toolbox project file (`.prj`).

Example: `toolboxFile = 'myToolbox.mltbx'`

Example: `toolboxFile = "C:\Work\myOtherToolbox.prj"`

`newVersion` — New toolbox version number
character vector | string scalar

New toolbox version number, specified as a character vector or string scalar. Enter the toolbox version number in the *`Major.Minor.Bug.Build`* format. The *`Bug`* and *`Build`* segments are optional. Each segment of the version number is an integer.

Example: `newVersion = "2.0"`

Example: `newVersion = '3.7.12.2'`

Alternatives

You can query the version number of your toolbox from the Add-On Manager UI. For more information, see “Get and Manage Add-Ons”.

You can update the version number of your toolbox from the Package a Toolbox UI. For more information, see “Create and Share Toolboxes”.

See Also

`matlab.addons.toolbox.installToolbox` | `matlab.addons.toolbox.installedToolboxes` | `matlab.addons.toolbox.packageToolbox` | `matlab.addons.toolbox.uninstallToolbox`

Introduced in R2016a

matlab.addons.toolbox.uninstallToolbox

Uninstall toolbox

Syntax

```
matlab.addons.toolbox.uninstallToolbox(installedToolbox)
```

Description

`matlab.addons.toolbox.uninstallToolbox(installedToolbox)` uninstalls the specified toolbox. MATLAB removes all the files and folders associated with the toolbox from the path and then deletes them.

Examples

Uninstall Toolbox

Assume that you previously installed two toolboxes, `myToolbox` and `myOtherToolbox`, and you want to uninstall the `myToolbox` toolbox.

Obtain the structure array of installed toolboxes, and display it as a table.

```
toolboxes = matlab.addons.toolbox.installedToolboxes;
struct2table(toolboxes)
```

```
ans =

          Name           Version                     Guid
    ________________    _______    ____________________________________

    'myToolbox'         '1.5.2'    'd0169b4a-fe74-463f-981a-26160c94cbe5'
    'myOtherToolbox'    '1.0'      '1deb72c1-725f-4e1b-a1a7-dcc8e75453bb'
```

Read the table and note the location of the toolbox you want to uninstall. Since `myToolbox` is listed first in the table, information about it can be found in `toolboxes(1)`.

```
installedToolbox = toolboxes(1)
```

```
installedToolbox = 

       Name: 'myToolbox'
    Version: '1.5.2'
       Guid: 'd0169b4a-fe74-463f-981a-26160c94cbe5'
```

Uninstall the toolbox. If the uninstallation process is successful, MATLAB does not display output.

```
matlab.addons.toolbox.uninstallToolbox(installedToolbox)
```

Input Arguments

`installedToolbox` — Information about toolbox to uninstall
structure

Information about the toolbox to uninstall, specified as a structure. The structure is an element of the structure array returned by the `matlab.addons.toolbox.installedToolboxes` function, and it has the following fields.

Field	Description
`Name`	Name of the toolbox
`Version`	Toolbox version
`Guid`	Unique toolbox identifier

MATLAB uses the `Guid` field to determine which toolbox to uninstall.

Data Types: `struct`

Alternatives

You can uninstall toolboxes from the Add-On Manager UI. For more information, see “Get and Manage Add-Ons”.

See Also

`matlab.addons.toolbox.installToolbox` | `matlab.addons.toolbox.installedToolboxes` | `matlab.addons.toolbox.packageToolbox` | `matlab.addons.toolbox.toolboxVersion`

Introduced in R2016a

matlab.addons.uninstall

Uninstall add-on

Syntax

```
matlab.addons.uninstall(name)
matlab.addons.uninstall(identifier)
```

```
matlab.addons.uninstall( ___ ,version)
matlab.addons.uninstall( ___ ,'All')
```

Description

`matlab.addons.uninstall(name)` uninstalls the add-on with the specified name. `matlab.addons.uninstall` only supports uninstalling toolboxes.

If multiple versions of an add-on are installed, you must either specify a `version` or `'All'` to uninstall all versions.

`matlab.addons.uninstall(identifier)` uninstalls the add-on with the specified identifier.

`matlab.addons.uninstall( ___ ,version)` also specifies the version of the add-on to uninstall. Use this syntax with any of the arguments in previous syntaxes.

`matlab.addons.uninstall( ___ ,'All')` uninstalls all installed versions of the specified add-on, if multiple versions are installed.

Examples

Uninstall Toolbox

Suppose that you have a toolbox called `Random File Name Creator` installed on your system. Get the list of installed add-ons and uninstall the `Random File Name Creator` toolbox.

Get the list of installed add-ons.

```
addons = matlab.addons.installedAddons
```

```
addons =

  1×4 table

            Name                    Version     Enabled                  Identifier
    ________________________________    _________    _______    ____________________________________

    "Random File Name Creator"      "1.0"         true      "75442144-f751-4011-bm0e-32b6fb2f1433"
```

Uninstall the `Random File Name Creator` toolbox.

```
matlab.addons.uninstall('Random File Name Creator');
```

Uninstall Specific Version of Toolbox

Suppose that you have version `2.0` and version `4.0` of a toolbox called `My Toolbox` installed on your system. After working with version `4.0`, you decide that you no longer need version `2.0` and want to uninstall it.

Get the list of installed add-ons.

```
addons = matlab.addons.installedAddons
```

```
addons =

  2×4 table

            Name                    Version     Enabled                  Identifier
    ________________________________    _________    _______    ____________________________________

    "My Toolbox_v4.0"               "4.0"         false     "6de8682e-9c3c-407e-bad7-aa103d738d08"
    "My Toolbox"                    "2.0"         true      "6de8682e-9c3c-407e-bad7-aa103d738d08"
```

Uninstall `My Toolbox` version `2.0` and confirm that it is uninstalled.

```
matlab.addons.uninstall('My Toolbox','2.0');
addons = matlab.addons.installedAddons
```

```
addons =

  1×4 table

            Name                    Version     Enabled                  Identifier
    ________________________________ _________ _________ ______________________________________
    "My Toolbox_v4.0"               "4.0"       false     "6de8682e-9c3c-407e-bad7-aa103d738d08"
```

Enable `My Toolbox` version `4.0` by specifying the first item in the returned list of identifiers. Then, confirm that it is enabled.

```
matlab.addons.enableAddon('My Toolbox_v4.0','4.0')
matlab.addons.isAddonEnabled('My Toolbox_v4.0','4.0')
```

```
ans =

  logical

   1
```

Uninstall All Versions of a Toolbox

Suppose that you have version `2.0` and version `4.0` of a toolbox called `My Toolbox` installed on your system. You decide that you no longer need `My Toolbox` and want to uninstall all versions of the toolbox.

Get the list of installed add-ons.

```
addons = matlab.addons.installedAddons
```

```
addons =

  2×4 table

            Name                    Version     Enabled                  Identifier
    ________________________________ _________ _________ ______________________________________

    "My Toolbox_v4.0"               "4.0"       true      "6de8682e-9c3c-407e-bad7-aa103d738d08"
    "My Toolbox"                    "2.0"       false     "6de8682e-9c3c-407e-bad7-aa103d738d08"
```

Uninstall all versions of `My Toolbox`.

```
matlab.addons.uninstall(addons.Identifier(1),'All')
```

Input Arguments

`name` — Name of add-on
character vector | string

Name of the add-on that you want to uninstall, specified as a character vector or string. To determine the name of an add-on, use the `matlab.addons.installedAddons` function.

`identifier` — Unique identifier of add-on
character vector | string

Unique identifier of the add-on that you want to uninstall, specified as a character vector or string. To determine the unique identifier of an add-on, use the `matlab.addons.installedAddons` function.

`version` — Version of add-on
character vector | string

Version of the add-on that you want to uninstall, specified as a character vector or string. To determine the version of an add-on, use the `matlab.addons.installedAddons` function.

See Also

`matlab.addons.install` | `matlab.addons.installedAddons`

Topics

"Get and Manage Add-Ons"

Introduced in R2018b

matlab.codetools.requiredFilesAndProducts

List dependencies of MATLAB program files

Syntax

```
fList = matlab.codetools.requiredFilesAndProducts(files)
[fList, pList] = matlab.codetools.requiredFilesAndProducts(files)
[fList, pList] = matlab.codetools.requiredFilesAndProducts( ___
,'toponly')
```

Description

`fList = matlab.codetools.requiredFilesAndProducts(files)` returns a list of the MATLAB program files required to run the program files specified by `files`.

The `matlab.codetools.requiredFilesAndProducts` function is intended to provide you with information to pass on to consumers of your MATLAB program files.

`[fList, pList] = matlab.codetools.requiredFilesAndProducts(files)` also returns a list of the MathWorks products possibly required to run the program files specified by `files`.

If you use the `matlab.codetools.requiredFilesAndProducts` function on MATLAB code that you received, `plist` only includes the required toolboxes that are installed on your system. In this case, `plist` can be incomplete.

`[fList, pList] = matlab.codetools.requiredFilesAndProducts( ___ ,'toponly')` indicates that for a file or product to be included in the output, it must be used directly by at least one file specified in `files`. The `'toponly'` input option is case insensitive.

Examples

Identify Required Files and Products for MATLAB Toolbox Function

Determine the required files and products for the `edge` function in the Image Processing Toolbox.

```
[fList,pList] = matlab.codetools.requiredFilesAndProducts('edge.m')
```

```
fList =

     {}


pList =

1x2 struct array with fields:

    Name
    Version
    ProductNumber
    Certain
```

There are no required MATLAB files, but there are two required products.

List the required products.

```
{pList.Name}'
```

```
ans =

    'MATLAB'
    'Image Processing Toolbox'
```

Identify Required Files and Products for Your MATLAB Program Files

In your current working folder, create a function in the file `getRandomNumber.m`.

```
function a = getRandomNumber
    rng shuffle
    a = rand;
end
```

Now, at the command line, determine the required files and products for `getRandomNumber.m`.

```
[fList,pList] = matlab.codetools.requiredFilesAndProducts('getRandomNumber.m')
```

```
fList =

    'C:\work\getRandomNumber.m'


pList =

             Name: 'MATLAB'
          Version: '8.5'
    ProductNumber: 1
          Certain: 1
```

The only file required to run the `getRandomNumber` function is the function file itself. The only required MathWorks product is MATLAB.

In your current working folder, create a function in the file `displayNumber.m`.

```
function displayNumber
    a = getRandomNumber;
    disp(['Your number is ' num2str(a)])
end
```

Now, at the command line, determine the required files and products for `displayNumber.m`.

```
[fList,pList] = matlab.codetools.requiredFilesAndProducts('displayNumber.m')
```

```
fList =

    'C:\work\displayNumber.m'    'C:\work\getRandomNumber.m'


pList =

             Name: 'MATLAB'
          Version: '8.5'
    ProductNumber: 1
          Certain: 1
```

In addition to the function file itself, the `displayNumber` function requires the `getRandomNumber.m` file. The only required MathWorks product is MATLAB.

Identify Top-Level Dependencies Only

In your current working folder, create a handle class in the file `ExampleHandle.m`.

```
classdef ExampleHandle < handle
    % class content
end
```

In your current working folder, create a class in the file `AnotherExampleHandle.m` that inherits from `ExampleHandle`.

```
classdef AnotherExampleHandle < ExampleHandle
    % class content
end
```

In your current working folder, create a function in the file `getHandles.m` that instantiates `AnotherExampleHandle` objects.

```
function [h1,h2] = getHandles()
    h1 = AnotherExampleHandle;
    h2 = AnotherExampleHandle;
end
```

Now, at the command line, determine the required files for `getHandles.m`.

```
[fList,~] = matlab.codetools.requiredFilesAndProducts('getHandles.m');
fList'
```

```
ans =

    'C:\work\AnotherExampleHandle.m'
    'C:\work\ExampleHandle.m'
    'C:\work\getHandles.m'
```

Determine the required files that are directly required for `getHandles.m`.

```
[fList,~] = matlab.codetools.requiredFilesAndProducts('getHandles.m','toponly')
```

```
fList =

    'C:\work\AnotherExampleHandle.m'    'C:\work\getHandles.m'
```

Although `AnotherExampleHandle.m` requires `ExampleHandle.m`, that file is not a direct requirement for `getHandles.m`.

Input Arguments

`files` — List of files for analysis

character vector | cell array of character vectors | string array

List of files for analysis, specified as a character vector, a cell array of character vectors, or a string array. Each element is the name of a single MATLAB program file. For example, `files` is a list of MATLAB program files that you intend to provide to other users. The `matlab.codetools.requiredFilesAndProducts` function provides you with requirements information to pass along with your files.

To ensure an accurate dependency report, `files` and dependencies must be on the MATLAB path. `matlab.codetools.requiredFilesAndProducts` does not return information about dependent files not on the path.

Example: `'myFile.m'` or `"C:\Program Files\MATLAB\R2014a\my_work\myFile.m"`

Example: `{'myFile.m','myOtherFile.m'}`

Example: `cellstr(ls('*.m'))`

Output Arguments

`fList` — List of user-authored MATLAB program files

cell array of character vectors

List of user-authored MATLAB program files required by `files`, returned as a cell array of character vectors. Each character vector indicates the full path of the required file. `fList` does not include built-in MATLAB files, since these files are installed with the products listed in `pList`.

`fList` includes dependent files that are accessed using standard file format and low-level I/O functions. These dependent files include text files, spreadsheets, images, audio, video, and XML files. For example, if you are analyzing a file that contains the code `load('mydata.mat')`, the `matlab.codetools.requiredFilesAndProducts` function includes `mydata.mat` in `fList`.

Functions that support automatic detection of dependent files include `audioinfo`, `audioread`, `csvread`, `daqread`, `dlmread`, `fileread`, `fopen`, `imfinfo`, `importdata`, `imread`, `load`, `matfile`, `mmfileinfo`, `open`, `readtable`, `type`, `VideoReader`, `xlsfinfo`, `xlsread`, `xmlread`, and `xslt`.

`pList` — List of MathWorks products
structure or array of structures

List of MathWorks products possibly required by `files`, returned as a structure or array of structures. Each product is described by name (`Name` field), version (`Version` field), product number (`ProductNumber` field), and a certainty indicator (`Certain` field). The `Certain` field has a value of 1 if `matlab.codetools.requiredFilesAndProducts` determines the product is required by the specified program files, `files`, or a value of 0 if the product is possibly required.

The `matlab.codetools.requiredFilesAndProducts` function is intended to provide you with information to pass on to consumers of your MATLAB program files. The version numbers indicate the version of the products you have installed when you execute the function. `Version` is not an indicator of backward compatibility.

See Also

Topics

"Identify Program Dependencies"

Introduced in R2014a

matlab.engine.engineName

Return name of shared MATLAB session

Syntax

```
name = matlab.engine.engineName
```

Description

`name = matlab.engine.engineName` returns the name of the MATLAB session if the session is shared. Otherwise, `matlab.engine.engineName` returns the empty value `''`.

Examples

Name of Shared MATLAB Session

Share the MATLAB session and return its name.

```
matlab.engine.shareEngine('Engine_1')
name = matlab.engine.engineName
```

```
name =

Engine_1
```

See Also

`matlab.engine.isEngineShared` | `matlab.engine.shareEngine`

Topics

"Connect Python to Running MATLAB Session"
"Connect Java to Running MATLAB Session"

Introduced in R2015b

matlab.engine.isEngineShared

Determine if MATLAB session is shared

Syntax

```
tf = matlab.engine.isEngineShared
```

Description

`tf = matlab.engine.isEngineShared` returns logical 1 (true) if a MATLAB session is shared. It returns logical 0 (false) if the session is not shared. By default, MATLAB sessions are not shared.

Examples

Test If MATLAB Session Is Shared

Determine if the current MATLAB session is shared for connections to other processes.

```
tf = matlab.engine.isEngineShared
```

```
tf =

     0
```

By default, the session is not shared.

See Also

`matlab.engine.engineName` | `matlab.engine.shareEngine`

Topics

"Connect Python to Running MATLAB Session"

"Connect Java to Running MATLAB Session"

Introduced in R2015b

matlab.engine.shareEngine

Convert running MATLAB session to shared session

Syntax

```
matlab.engine.shareEngine
matlab.engine.shareEngine(name)
```

Description

`matlab.engine.shareEngine` converts the current MATLAB session to a shared session with a default name. The default name is `'MATLAB_<process_ID>'`, where *<process_ID>* is a number that indicates the process ID of the current MATLAB session.

`matlab.engine.shareEngine(name)` converts the current MATLAB session to a shared session and gives it the specified name.

If there is already a shared MATLAB session on your local machine that has the specified name, then `matlab.engine.shareEngine` converts the current session to a shared session with a default name.

Examples

Share MATLAB Session

Convert the current MATLAB session to a shared session with a default name.

```
matlab.engine.shareEngine
matlab.engine.engineName
```

```
ans =

MATLAB_49593
```

`matlab.engine.shareEngine` gives the shared session the default name `MATLAB_49593`, where 49593 is the process ID of the MATLAB session. The process ID is different each time you start MATLAB.

Name and Share Your MATLAB Session

Give a name to the current MATLAB session and share the session.

```
matlab.engine.shareEngine('Engine_1')
matlab.engine.engineName
```

```
ans =

Engine_1
```

Input Arguments

`name` — Name for shared MATLAB session
character array | string

Name for the shared MATLAB session, specified as a character array or string. `name` must be a valid MATLAB variable name. For information on valid names, see “Variable Names”.

Data Types: `char` | `string`

Tips

You also can use a MATLAB startup option to start a shared session. To start a shared MATLAB session, type this command at the operating system prompt:

```
matlab -r "matlab.engine.shareEngine"
```

See Also

`matlab.engine.engineName` | `matlab.engine.isEngineShared`

Topics

“Connect Python to Running MATLAB Session”

"Connect Java to Running MATLAB Session"
"Connect C++ to Running MATLAB Session"

Introduced in R2015b

matfile

Access and change variables in MAT-file without loading file into memory

Description

Use a MAT-file object to access and change variables in a MAT-file without loading the file into memory. You can load or save parts of variables. Partial loading and saving of variables using a MAT-file object requires less memory than the `load` and `save` commands.

Creation

Syntax

```
matObj = matfile(filename)
matObj = matfile(filename,'Writable',isWritable)
```

Description

`matObj = matfile(filename)` creates a `matlab.io.MatFile` object connected to the MAT-file specified by `filename`.

The MAT-file object allows you to access and change variables directly in a MAT-file, without having to load the variables into memory.

`matObj = matfile(filename,'Writable',isWritable)` enables or disables write access to the file. Specify `isWritable` as `true` or `false`.

Input Arguments

`filename` — Name of MAT-file
character vector | string scalar

Name of the MAT-file, specified as a character vector or string scalar.

- `filename` can include a full or partial path; Otherwise, `matfile` searches for the file along the MATLAB search path.
- If `filename` does not include an extension, `matfile` appends `.mat`.
- If the file does not exist, `matfile` creates a Version 7.3 MAT-file on the first assignment to a variable.

`isWritable` — Enable write access
`true` | `false`

Enable write access, specified as `true` or `false`. The default value is `true` for new files, `false` for existing files.

- `true` — Enable write access to the MAT-file from the MAT-file object. If the file is read-only, change the system permissions by using the `fileattrib` function.
- `false` — Disable write access to the MAT-file from the MAT-file object.

Properties

Access MAT-File Object Properties

Access MAT-file object properties using the syntax *`ObjectName`*`.Properties.`*`PropertyName`*, where *`PropertyName`* is the name of a property. For example, you can access the source file path of the MAT-file connected to the MAT-file object `mObj` using the syntax `mObj.Properties.Source`.

`Source` — File path
character vector

File path, returned as character vector containing the fully qualified path to the MAT-file.

`Writable` — Write access status
`true` | `false`

Write access status of the MAT-file object, returned as either `true` or `false`.

- `true` — MAT-file object has write access to the MAT-file.
- `false` — MAT-file object has read-only access to the MAT-file.

Object Functions

`size`	Get array dimensions of variable in MAT-file `allDims = size(matObj,variable)` returns the size of each dimension of the specified variable in the file corresponding to `matObj`. The output `allDims` is a `1`-by-`m` vector, where `m = ndims(variable)`. `[dim1,...,dimN] = size(matObj,variable)` returns the sizes of each dimension in separate output variables `dim1,...,dimN`. `selectedDim = size(matObj,variable,dim)` returns the size of the specified dimension. **Note:** Do not call `size` with the syntax `size(matObj.variable)`. This syntax loads the entire contents of the variable into memory. For very large variables, this load operation results in `Out of Memory` errors.
`who`	Get list of variables in MAT-file `varlist = who(matObj)` lists alphabetically all variables in the MAT-file associated with `matObj`. Optionally, returns the list in cell array `varlist`. `varlist = who(matObj,variables)` lists the specified variables.
`whos`	Get list of variables in MAT-file, with sizes and types `details = whos(matObj)` returns information about all variables in the MAT-file associated with `matObj`. `details = whos(matObj,VarName1,...,VarNameN)` returns information about the specified variables.

Examples

Create MAT-file Object

Create a MAT-file object for your file. The `matfile` function constructs a `matlab.io.MatFile` object that corresponds to a MAT-File.

```
matObj = matfile('myFile.mat')
```

Enable Write Access to MAT-file

Enable write access to the MAT-file, `myFile.mat`, when you create the object.

```
m = matfile('myFile.mat','Writable',true);
```

Alternatively, you can enable write access after you create the MAT-file object by setting the `Writable` property.

```
m.Properties.Writable = true;
```

Load Entire Variable

Open the sample MAT-file `topography.mat`. Read the variable `topo` from the file using the MAT-fie object. MATLAB® loads the entire variable, `topo`, into the workspace.

```
m = matfile('topography.mat');
topo = m.topo;
```

Copyright 2012 The MathWorks, Inc.

Save Entire Variable to Existing MAT-file

Create a MAT-file containing an array and then add another array to the file using the MAT-file object.

Create a MAT file by generating a 20-by-20 array, `x`, and saving it to `myFile.mat`.

```
x = magic(20);
save('myFile.mat','x');
```

Create a MAT-file object connected to the existing MAT-file. Enable write access by setting `Writable` to `true`.

```
m = matfile('myFile.mat','Writable',true);
```

Generate another 15-by-15 array, `y`. Save `y` to the MAT-file using the MAT-file object. Specify the variable in the MAT-file using dot notation similar to accessing fields of structure arrays. MATLAB® adds a variable named `y` to the file.

```
y = magic(15);
m.y = y;
```

Display all variables stored in the MAT-file, `myFile.mat`.

```
whos('-file','myFile.mat')
```

```
  Name       Size            Bytes  Class     Attributes

  x         20x20             3200  double
  y         15x15             1800  double
```

Load and Save Parts of Variables

Access specific parts of a variable in a MAT-file using the MAT-file object. For example, you can save data to a subset of a variable in a MAT-file or read a subset of a variable into the MATLAB® workspace.

Save data to a subset of a variable `y` in the file `myFile2.mat` using the MAT-file object. First, create the MAT-file object `m`.

```
m = matfile('myFile2.mat');
```

Next, create and save an array to part of the variable y. Specify the variable in the MAT-file using dot notation similar to accessing fields of structure arrays. MATLAB® inserts the 20-by-20 array into the elements of `y` specified by the indices `(81:100,81:100)`.

```
m.y(81:100,81:100) = magic(20);
```

Read a subset of array `y` into a new workspace variable, `z`. MATLAB® reads the 10-by-10 subarray specified by the indices `(85:94,85:94)` from the MAT-file into workspace variable `z`.

```
z = m.y(85:94,85:94);
```

Determine Size of Variables

Determine the size of a variable, and then calculate the average of each column.

Open the example MAT-file, `stocks.mat`.

```
filename = 'stocks.mat';
m = matfile(filename);
```

Determine the size of the variable, `stocks`, in `stocks.mat`.

```
[nrows,ncols] = size(m,'stocks');
```

Compute the average of each column of the variable `stocks`.

```
avgs = zeros(1,ncols);
for i = 1:ncols
    avgs(i) = mean(m.stocks(:,i));
end
```

Limitations

- When accessing parts of variables in MAT-files by indexing:
  - Linear indexing is not supported. You must specify indices for all dimensions.
  - Assigning complex values to indexed portions of a real array is not supported.
- For sparse arrays in a MAT-file, the MAT-file object:
  - Supports reading a subset of a sparse array by indexing.
  - Does not support writing (assigning values) to a subset of a sparse array by indexing.

- The MAT-file object does not support indexing into:
  - Variables of tables
  - Cells of cell arrays
  - String arrays
  - Fields of structure arrays
  - User-defined classes
- The MAT-file object does not support evaluating function handles using the `m` output. For example, if your MAT-file contains function handle `myfunc`, the syntax `m.myfunc()` attempts to index into the function handle, and does not invoke the function.

Tips

- Efficient partial loading and saving requires Version 7.3 MAT-files. To create a Version 7.3 MAT-file, call the `save` function with the `'-v7.3'` option. For example, to convert an existing MAT-file named `durer.mat` to Version 7.3, call:

  ```
  load('durer.mat');
  save('mycopy_durer.mat','-v7.3');
  ```
- Using the `end` keyword as part of an index causes MATLAB to load the entire variable into memory. For very large variables, this load operation results in `Out of Memory` errors. Rather than using `end`, determine the extent of a variable with the `size` method:

  ```
  sizeMyVar = size(matObj,'myVar')
  ```

See Also

`load` | `save`

Topics

"Save and Load Parts of Variables in MAT-Files"
MAT-File Versions

Introduced before R2006a

matlab.io.fits.closeFile

Close FITS file

Syntax

```
closeFile(fptr)
```

Description

`closeFile(fptr)` closes an open FITS file.

This function corresponds to the `fits_close_file (ffclos)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits','READONLY');
fits.closeFile(fptr);
```

See Also

`createFile` | `openFile`

matlab.io.fits.createFile

Create FITS file

Syntax

```
fptr = createFile(filename)
```

Description

`fptr = createFile(filename)` creates a FITS file. An error will be returned if the specified file already exists, unless the filename is prefixed with an exclamation point (!). In that case CFITSIO will overwrite (delete) any existing file with the same name. Specify `filename` as a character vector or string scalar.

This function corresponds to the `fits_create_file (ffinit)` function in the CFITSIO library C API.

Examples

Create a new FITS file.

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'uint8',[256 512]);
fits.closeFile(fptr);
fitsdisp('myfile.fits');
```

See Also

`closeFile` | `createImg` | `createTbl` | `openFile`

matlab.io.fits.deleteFile

Delete FITS file

Syntax

```
deleteFile(fptr)
```

Description

`deleteFile(fptr)` closes and deletes an open FITS file. This can be useful if a FITS file cannot be properly closed.

This function corresponds to the `fits_delete_file (ffdelt)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','tst0012.fits');
copyfile(srcFile,'myfile.fits');
fileattrib('myfile.fits','+w');
fptr = fits.openFile('myfile.fits','readwrite');
fits.deleteFile(fptr);
fptrs = fits.getOpenFiles()
```

See Also

`closeFile` | `createFile`

matlab.io.fits.fileName

Name of FITS file

Syntax

```
name = fileName(fptr)
```

Description

`name = fileName(fptr)` returns the name of the FITS file associated with the file handle.

This function corresponds to the `fits_file_name (ffflnm)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits','READONLY');
name = fits.fileName(fptr);
fits.closeFile(fptr);
```

See Also

`createFile` | `openFile`

matlab.io.fits.fileMode

I/O mode of FITS file

Syntax

```
mode = fileMode(fptr)
```

Description

`mode = fileMode(fptr)` returns the I/O mode of the opened FITS file. Possible values returned for `mode` are `'READONLY'` or `'READWRITE'`.

This function corresponds to the `fits_file_mode (ffflmd)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
mode = fits.fileMode(fptr);
fits.closeFile(fptr);
```

See Also

`createFile` | `openFile`

matlab.io.fits.openFile

Open FITS file

Syntax

```
fptr = openFile(filename)
fptr = openFile(filename,mode)
```

Description

`fptr = openFile(filename)` opens an existing FITS file in read-only mode and returns a file pointer, `fptr`, which references the primary array (first header data unit, or "HDU"). The `openFile` function supports the extended file name syntax. Specify `filename` as a character vector or string scalar.

This function corresponds to the `fits_open_file (ffopen)` function in the CFITSIO library C API.

The `openFile` function is similar to the function `openDiskFile`. In addition to opening FITS files, the `openFile` function supports the extended file name syntax in the input file name. If the filename (or folder path) contains square or curly brace characters that would confuse the extended filename parser, then use `openDiskFile`.

`fptr = openFile(filename,mode)` opens an existing FITS file according to the `mode`, which describes the type of access. `mode` may be either `'readonly'` or `'readwrite'`.

Examples

Open a file in read-only mode and read image data from the primary array.

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
imagedata = fits.readImg(fptr);
fits.closeFile(fptr);
```

Open a file in read/write mode and add a comment to the primary array.

```
import matlab.io.*
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','tst0012.fits');
copyfile(srcFile,'myfile.fits');
fileattrib('myfile.fits','+w');
fptr = fits.openFile('myfile.fits','readwrite');
fits.writeComment(fptr,'This is just a comment.');
fits.closeFile(fptr);
```

See Also

`closeFile` | `createFile` | `openDiskFile`

matlab.io.fits.openDiskFile

Open FITS file

Syntax

```
fptr = openDiskFile(filename)
fptr = openDiskFile(filename,mode)
```

Description

`fptr = openDiskFile(filename)` opens an existing FITS file in read-only mode and returns a file pointer `fptr`, which is the first header data unit (HDU).. The `openDiskFile` function does not support the extended-file-name syntax.

This function corresponds to the `fits_open_diskfile (ffdkopen)` function in the CFITSIO library C API.

The `openDiskFile` function is similar to the `openFile` function, except that `openDiskFile` does not support the extended-file-name syntax in the input file name. Use `openDiskFile` in cases where the file name (or folder path) contains square or curly brace characters that would confuse the extended-file-name parser.

`fptr = openDiskFile(filename,mode)` opens an existing FITS file according to the type of access specified by mode.

Examples

Read and Write FITS Files

Open a FITS file to read image data, create a copy of the file, and then write a comment to the primary array.

Open a file in read-only mode and read image data from the primary array.

```
import matlab.io.*
fptr = fits.openDiskFile('tst0012.fits');
imagedata = fits.readImg(fptr); % read image from primary array
fits.closeFile(fptr);
```

Create a new file in read/write mode, copy data into the file, and then add a comment to the primary array.

```
srcFile = fullfile(matlabroot,'toolbox',...
                   'matlab','demos','tst0012.fits');
copyfile(srcFile,'myfile.fits');
fileattrib('myfile.fits','+w');
fptr = fits.openDiskFile('myfile.fits','readwrite');
fits.writeComment(fptr,'This is just a comment.');
fits.closeFile(fptr);
```

Input Arguments

`filename` — Name of file to read

character vector

Name of the file to read, specified as a character vector. The `openDiskFile` function does not support the extended file name syntax.

Example: If the `filename` is `'temp(1)\tst0012.fits'`, then `openDiskFile` writes the file `'tst0012.fits'` to the folder `temp(1)`.

Data Types: `char`

`mode` — File access type

`'readonly'` (default) | `'readwrite'`

File access type, specified as a character vector containing `'readonly'` or `'readwrite'`.

- `'readonly'` — Open file for reading.
- `'readwrite'` — Open file for reading and writing.

Example: `'readwrite'`

Data Types: `char`

See Also

`closeFile` | `createFile` | `openFile`

Introduced in R2018a

matlab.io.fits.createImg

Create FITS image

Syntax

```
createImg(fptr,bitpix,naxes)
```

Description

`createImg(fptr,bitpix,naxes)` creates a new primary image or image extension with a specified datatype `bitpix` and size `naxes`. If the FITS file is currently empty then a primary array is created, otherwise a new image extension is appended to the file.

The first two elements of `naxes` correspond to the NAXIS2 and NAXIS1 keywords, while any additional elements correspond to the NAXIS3, NAXIS4 ... NAXISn keywords.

Specify the datatype `bitpix` as a character vector or string scalar containing either a CFITSIO name or the corresponding MATLAB datatype.

`'byte_img'`	`'uint8'`
`'short_img'`	`'int16'`
`'long_img'`	`'int32'`
`'longlong_img'`	`'int64'`
`'float_img'`	`'single'`
`'double_img'`	`'double'`

This function corresponds to the `fits_create_imgll(ffcrimll)` function in the CFITSIO library C API.

Examples

Create two images in a new FITS file. There are 100 rows (NAXIS2 keyword) and 200 columns (NAXIS1 keyword) in the first image, and 256 rows (NAXIS2 keyword), 512 columns (NAXIS1 keyword), and 3 planes (NAXIS3 keyword) in the second image.

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'int16',[100 200]);
fits.createImg(fptr,'byte_img',[256 512 3]);
fits.closeFile(fptr);
fitsdisp('myfile.fits');
```

See Also

`createTbl` | `insertImg` | `readImg` | `setCompressionType` | `writeImg`

matlab.io.fits.getImgSize

Size of image

Syntax

```
imagesize = getImgSize(fptr)
```

Description

`imagesize = getImgSize(fptr)` returns the number of rows and columns of an image. This function corresponds to the `fits_get_img_size (ffgisz)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*;
fptr = fits.openFile('tst0012.fits');
hdus = [1 3 4];
for j = hdus;
    htype = fits.movAbsHDU(fptr,j);
    sz = fits.getImgSize(fptr);
    fprintf('HDU %d:  "%s", [', j, htype);
    for k = 1:numel(sz)
        fprintf(' %d ', sz(k));
    end
    fprintf(']\n');
end
fits.closeFile(fptr);
```

See Also

`createImg` | `getImgType`

matlab.io.fits.getImgType

Data type of image

Syntax

```
datatype = getImgType(fptr)
```

Description

`datatype = getImgType(fptr)` gets the data type of an image. `datatype` can be one of the following:

```
'BYTE_IMG'
'SHORT_IMG'
'LONG_IMG'
'LONGLONG_IMG'
'FLOAT_IMG'
'DOUBLE_IMG'
```

This function corresponds to the `fits_get_img_type (ffgidt)` function in the CFITSIO library C API.

Examples

```
fptr = fits.openFile('tst0012.fits');
hdus = [1 3 4];
for j = hdus;
    htype = fits.movAbsHDU(fptr,j);
    dtype = fits.getImgType(fptr);
    fprintf('HDU %d:  "%s", "%s"\n', j, htype, dtype);
end
fits.closeFile(fptr);
```

See Also

`getImgSize`

matlab.io.fits.insertImg

Insert FITS image after current image

Syntax

```
insertImage(fptr,bitpix,naxes)
```

Description

`insertImage(fptr,bitpix,naxes)` inserts a new image extension immediately following the current HDU. If the file has just been created, a new primary array is inserted at the beginning of the file. Any following extensions in the file will be shifted down to make room for the new extension. If the current HDU is the last HDU in the file, then the new image extension will be appended to the end of the file.

This function corresponds to the `fits_insert_imgll (ffiimgll)` function in the CFITSIO library C API.

Examples

Create a 150x300 image between the 1st and 2nd images in a FITS file.

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'byte_img',[100 200]);
fits.createImg(fptr,'byte_img',[200 400]);
fits.movAbsHDU(fptr,1);
fits.insertImg(fptr,'byte_img',[150 300]);
fits.closeFile(fptr);
fitsdisp('myfile.fits','mode','min');
```

See Also

`createImg`

matlab.io.fits.readImg

Read image data

Syntax

```
imgdata = readImg(fptr)
imgdata = readImg(fptr,fpixel,lpixel)
imgdata = readImg(fptr,fpixel,lpixel,inc)
```

Description

`imgdata = readImg(fptr)` reads the entire current image. The number of rows in `imgdata` will correspond to the value of the NAXIS2 keyword, while the number of columns will correspond to the value of the NAXIS1 keyword. Any further dimensions of `imgdata` will correspond to NAXIS3, NAXIS4, and so on.

`imgdata = readImg(fptr,fpixel,lpixel)` reads the subimage defined by pixel coordinates `fpixel` and `lpixel`. The `fpixel` argument is the coordinate of the first pixel and `lpixel` is the coordinate of the last pixel. `fpixel` and `lpixel` are one-based.

`imgdata = readImg(fptr,fpixel,lpixel,inc)` reads the subimage defined by `fpixel`, `lpixel`, and `inc`. The `inc` argument denotes the inter- element spacing along each extent.

This function corresponds to the `fits_read_subset (ffgsv)` function in the CFITSIO library C API.

Examples

Read an entire image.

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
data = fits.readImg(fptr);
fits.closeFile(fptr);
```

Read a 70x80 image subset.

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
img = fits.readImg(fptr,[11 11],[80 90]);
fits.closeFile(fptr);
```

See Also

`createImg` | `writeImg`

matlab.io.fits.setBscale

Reset image scaling

Syntax

```
setBscale(fptr,BSCALE,BZERO)
```

Description

`setBscale(fptr,BSCALE,BZERO)` resets the scaling factors in the primary array or image extension according to the equation

```
output = (FITS array) * BSCALE + BZERO
```

The inverse formula is used when writing data values to the FITS file.

This only affects the automatic scaling performed when the data elements are read. It does not change the BSCALE and BZERO keyword values.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.setBscale(fptr,2.0,0.5);
data = fits.readImg(fptr);
fits.closeFile(fptr);
```

See Also

`readImg`

matlab.io.fits.writeImg

Write to FITS image

Syntax

```
writeImg(fptr,data)
writeImg(fptr,data,fpixel)
```

Description

`writeImg(fptr,data)` writes an entire image to the FITS data array. The number of rows and columns in `data` must equal the values of the NAXIS2 and NAXIS1 keywords, respectively. Any further extents must correspond to the NAXIS3, NAXIS4 ... NAXISn keywords respectively.

`writeImg(fptr,data,fpixel)` writes a subset of an image to the FITS data array. `fpixel` gives the coordinate of the first pixel in the image region.

This function corresponds to the `fits_write_subset (ffpss)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'long_img',[256 512]);
data = reshape(1:256*512,[256 512]);
data = int32(data);
fits.writeImg(fptr,data);
fits.closeFile(fptr);
```

Create an 80x40 `uint8` image and set all but the outermost pixels to 1.

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
```

```
fits.createImg(fptr,'uint8',[80 40]);
data = ones(78,38);
fits.writeImg(fptr,data,[1 1]);
fits.closeFile(fptr);
```

See Also

`createImg` | `readImg`

matlab.io.fits.deleteKey

Delete key by name

Syntax

```
deleteKey(fptr,keyname)
```

Description

`deleteKey(fptr,keyname)` deletes a keyword by name. Specify `keyname` as a character vector or string scalar.

This function corresponds to the `fits_delete_key(ffdrec)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','tst0012.fits');
copyfile(srcFile,'myfile.fits');
fileattrib('myfile.fits','+w');
fprintf('Before key deletion...\n');
fitsdisp('myfile.fits','index',1);
fptr = fits.openFile('myfile.fits','readwrite');
fits.deleteKey(fptr,'DATE');
fits.closeFile(fptr);
fprintf('\n\n\nAfter key deletion...\n');
fitsdisp('myfile.fits','index',1);
```

See Also

`deleteRecord` | `writeKey`

matlab.io.fits.deleteRecord

Delete key by record number

Syntax

```
deleteRecord(fptr,keynum)
```

Description

`deleteRecord(fptr,keynum)` deletes a keyword by record number.

This function corresponds to the `fits_delete_record (ffdrec)` function in the CFITSIO library C API.

Examples

Delete the 18th keyword ("ORIGIN") in a primary array.

```
import matlab.io.*
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','tst0012.fits');
copyfile(srcFile,'myfile.fits');
fileattrib('myfile.fits','+w');
fptr = fits.openFile('myfile.fits','readwrite');
card = fits.readRecord(fptr,18);
fits.deleteRecord(fptr,18);
fits.closeFile(fptr);
```

See Also

`deleteKey` | `readRecord`

matlab.io.fits.getHdrSpace

Number of keywords in header

Syntax

```
[nkeys,morekeys] = fits.getHdrSpace(fptr)
```

Description

`[nkeys,morekeys] = fits.getHdrSpace(fptr)` returns the number of existing keywords (not counting the END keyword) and the amount of space currently available for more keywords. It returns `morekeys = -1` if the header has not yet been closed. Note that the CFITSIO library will dynamically add space if required when writing new keywords to a header so in practice there is no limit to the number of keywords that can be added to a header.

This function corresponds to the `fits_get_hdrspace (ffghsp)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
[nkeys,morekeys] = fits.getHdrSpace(fptr);
fits.closeFile(fptr);
```

matlab.io.fits.readCard

Header record of keyword

Syntax

```
card = readCard(fptr,keyname)
```

Description

`card = readCard(fptr,keyname)` returns the entire 80-character header record of the keyword, with any trailing blank characters stripped off. Specify `keyname` as a character vector or string scalar.

This function corresponds to the `fits_read_card (ffgcrd)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
n = fits.getNumHDUs(fptr);
for j = 1:n
    fits.movAbsHDU(fptr,j);
    card = fits.readCard(fptr,'NAXIS');
    fprintf('HDU %d:  ''%s''\n', j, card);
end
fits.closeFile(fptr);
```

See Also

`readKey` | `readRecord`

matlab.io.fits.readKey

Keyword

Syntax

```
[value,comment] = readKey(fptr,keyname)
```

Description

`[value,comment] = readKey(fptr,keyname)` returns the specified key and comment. Specify `keyname` as a character vector or string scalar. `value` and `comment` are returned as character vectors.

This function corresponds to the `fits_read_key_str (ffgkys)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
n = fits.getNumHDUs(fptr);
for j = 1:n
    fits.movAbsHDU(fptr,j);
    [key,comment] = fits.readKey(fptr,'NAXIS');
    fprintf('HDU %d:  NAXIS %s, "%s"\n', j, key, comment);
end
fits.closeFile(fptr);
```

See Also

`readKeyCmplx` | `readKeyDbl` | `readKeyLongLong`

matlab.io.fits.readKeyCmplx

Keyword as complex scalar value

Syntax

```
[value,comment] = readKeyCmplx(fptr,keyname)
```

Description

`[value,comment] = readKeyCmplx(fptr,keyname)` returns the specified key and comment. `value` is returned as a double precision complex scalar value.

This function corresponds to the `fits_read_key_dblcmp (ffgkym)` function in the CFITSIO library C API.

See Also

`readKey` | `readKeyDbl` | `readKeyLongLong`

matlab.io.fits.readKeyDbl

Keyword as double precision value

Syntax

```
[value,comment] = readKeyDbl(fptr,keyname)
```

Description

`[value,comment] = readKeyDbl(fptr,keyname)` returns the specified key and comment.

This function corresponds to the `fits_read_key_dbl (ffgkyd)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
n = fits.getNumHDUs(fptr);
for j = 1:n
    fits.movAbsHDU(fptr,j);
    [key,comment] = fits.readKeyDbl(fptr,'NAXIS');
    fprintf('HDU %d:  NAXIS %s, "%s"\n', j, key, comment);
end
fits.closeFile(fptr);
```

See Also

`readKey` | `readKeyCmplx` | `readKeyLongLong`

matlab.io.fits.readKeyLongLong

Keyword as `int64`

Syntax

```
[value,comment] = readKeyLongLong(fptr,keyname)
```

Description

`[value,comment] = readKeyLongLong(fptr,keyname)` returns the specified key and comment. `value` is returned an `int64` scalar value.

This function corresponds to the `fits_read_key_lnglng (ffgkyjj)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
n = fits.getNumHDUs(fptr);
for j = 1:n
    fits.movAbsHDU(fptr,j);
    [key,comment] = fits.readKeyLongLong(fptr,'NAXIS');
    fprintf('HDU %d:  NAXIS %d, "%s"\n', j, key, comment);
end
fits.closeFile(fptr);
```

See Also

`readKey` | `readKeyCmplx` | `readKeyDbl`

matlab.io.fits.readKeyLongStr

Long string value

Syntax

```
[value,comment] = readKeyLongStr(fptr,keyname)
```

Description

`[value,comment] = readKeyLongStr(fptr,keyname)` returns the specified long string value and comment as character vectors.

This function corresponds to the `fits_read_key_longstr (ffgkls)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
idata = repmat(char(97:106),1,10);
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'byte_img',[100 200]);
fits.writeKey(fptr,'mykey',idata);
odata1 = fits.readKey(fptr,'mykey');
odata2 = fits.readKeyLongStr(fptr,'mykey');
fits.closeFile(fptr);
```

See Also

`readKey`

matlab.io.fits.readKeyUnit

Physical units string from keyword

Syntax

```
units = readKeyUnit(fptr,keyname)
```

Description

`units = readKeyUnit(fptr,keyname)` returns the physical units from an existing keyword. If no units are defined, `units` is returned as an empty character vector.

This function corresponds to the `fits_read_key_unit (ffgunt)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'long_img',[10 20]);
fits.writeKey(fptr,'VELOCITY',12.3,'orbital speed');
fits.writeKeyUnit(fptr,'VELOCITY','km/s');
units = fits.readKeyUnit(fptr,'VELOCITY');
fits.closeFile(fptr);
```

See Also

`readKey` | `writeKeyUnit`

matlab.io.fits.readRecord

Header record specified by number

Syntax

```
card = readRecord(fptr,keynum)
```

Description

`card = readRecord(fptr,keynum)` returns the entire 80-character header record identified by the numeric `keynum`. Trailing blanks are truncated.

This function corresponds to the `fits_read_record (ffgrec)` function in the CFITSIO library C API.

Examples

Read the second record in each HDU.

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
n = fits.getHdrSpace(fptr);
for j = 1:n
    card = fits.readRecord(fptr,j);
    fprintf('record %d:  "%s"\n', j, card);
end
fits.closeFile(fptr);
```

See Also

`deleteRecord` | `readCard` | `readKey`

matlab.io.fits.writeComment

Write or append COMMENT keyword to CHU

Syntax

`writeComment(fptr,comment)`

Description

`writeComment(fptr,comment)` writes (appends) a COMMENT keyword to the CHU. The comment keyword, specified as a character vector or string scalar, will be continued over multiple keywords if it is longer than 70 characters.

This function corresponds to the `fits_write_comment (ffpcom)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'byte_img',[100 200]);
fits.writeComment(fptr,'this is a comment');
fits.writeComment(fptr,'this is another comment');
fits.closeFile(fptr);
fitsdisp('myfile.fits','mode','full');
```

See Also

`writeDate` | `writeHistory`

matlab.io.fits.writeDate

Write DATE keyword to CHU

Syntax

`writeDate(FPTR)`

Description

`writeDate(FPTR)` writes the DATE keyword to the CHU.

This function corresponds to the `fits_write_date (ffpdat)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'byte_img',[100 200]);
fits.writeDate(fptr);
fits.closeFile(fptr);
fitsdisp('myfile.fits','mode','full');
```

See Also

`writeComment` | `writeHistory`

matlab.io.fits.writeKey

Update or add new keyword into current HDU

Syntax

```
writeKey(fptr,keyname,value,comment)
writeKey(fptr,keyname,value,comment,decimals)
```

Description

`writeKey(fptr,keyname,value,comment)` adds a new record in the current HDU, or updates it if it already exists. `comment` is optional.

`writeKey(fptr,keyname,value,comment,decimals)` adds a new floating point keyword in the current HDU, or updates it if it already exists. You must use this syntax to write a keyword with imaginary components. `decimals` is ignored otherwise.

If a character `value` exceeds 68 characters in length, the LONGWARN convention is automatically employed.

This function corresponds to the `fits_write_key (ffpky)` and `fits_update_key (ffuky)` family of functions in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'byte_img',[100 200]);
fits.writeKey(fptr,'mykey1','a char value','with a comment');
fits.writeKey(fptr,'mykey2',int32(1));
fits.writeKey(fptr,'mykey3',5+7*j,'with another comment');
fits.writeKey(fptr,'mykey4',4/3,'with yet another comment',2);
fits.closeFile(fptr);
fitsdisp('myfile.fits','mode','full');
```

See Also

`deleteKey` | `readKey` | `readRecord`

matlab.io.fits.writeKeyUnit

Write physical units string

Syntax

```
writeKeyUnit(fptr,keyname,unit)
```

Description

`writeKeyUnit(fptr,keyname,unit)` writes the physical units into an existing keyword. `fptr` is created using the `fits.createFile` function. Specify `keyname` and `unit` as a character vector or string scalar.

This function corresponds to the `fits_write_key_unit (ffpunt)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.createFile('myFitsFile.fits');
fits.createImg(fptr,'long_img',[10 20]);
fits.writeKey(fptr,'VELOCITY',12.3,'orbital speed');
fits.writeKeyUnit(fptr,'VELOCITY','km/s');
fits.closeFile(fptr);
```

See Also

`readKeyUnit`

matlab.io.fits.writeHistory

Write or append HISTORY keyword to CHU

Syntax

```
writeHistory(fptr,history)
```

Description

`writeHistory(fptr,history)` writes (appends) a HISTORY keyword to the CHU. The history keyword, specified as a character vector or string scalar, is continued over multiple keywords if it is longer than 70 characters.

This function corresponds to the `fits_write_history (ffphis)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'byte_img',[100 200]);
fits.writeHistory(fptr,'this is a history keyword');
fits.closeFile(fptr);
fitsdisp('myfile.fits','mode','full');
```

See Also

`writeComment` | `writeDate`

matlab.io.fits.copyHDU

Copy current HDU from one file to another

Syntax

```
copyHDU(infptr,outfptr)
```

Description

`copyHDU(infptr,outfptr)` copies the current HDU from the FITS file associated with `infptr` and appends it to the end of the FITS file associated with `outfptr`.

This function corresponds to the `fits_copy_hdu (ffcopy)` function in the CFITSIO library C API.

Examples

Copy the first, third, and fifth HDUs from one file to another.

```
import matlab.io.*
infptr = fits.openFile('tst0012.fits');
outfptr = fits.createFile('myfile.fits');
fits.copyHDU(infptr,outfptr);
fits.movAbsHDU(infptr,3);
fits.copyHDU(infptr,outfptr);
fits.movAbsHDU(infptr,5);
fits.copyHDU(infptr,outfptr);
fits.closeFile(infptr);
fits.closeFile(outfptr);
fitsdisp('tst0012.fits','mode','min','index',[1 3 5]);
fitsdisp('myfile.fits','mode','min');
```

See Also

`deleteHDU`

matlab.io.fits.deleteHDU

Delete current HDU in FITS file

Syntax

```
HDU_TYPE = deleteHDU(fptr)
```

Description

`HDU_TYPE = deleteHDU(fptr)` deletes the current HDU in the FITS file. Any following HDUs will be shifted forward in the file, filling the gap created by the deleted HDU. In the case of deleting the primary array (the first HDU in the file) then the current primary array will be replaced by a null primary array containing the minimum set of required keywords and no data. If there are more HDUs in the file following the HDU being deleted, then the current HDU will be redefined to point to the following HDU. If there are no following HDUs then the current HDU will be redefined to point to the previous HDU. `HDU_TYPE` returns the type of the new current HDU.

This function corresponds to the `fits_delete_hdu (ffdhdu)` function in the CFITSIO library C API.

Examples

Delete the second HDU in a FITS file.

```
import matlab.io.*
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','tst0012.fits');
copyfile(srcFile,'myfile.fits');
fileattrib('myfile.fits','+w');
fitsdisp('myfile.fits','mode','min');
fptr = fits.openFile('myfile.fits','readwrite');
fits.movAbsHDU(fptr,2);
new_current_hdu = fits.deleteHDU(fptr);
fits.closeFile(fptr);
fitsdisp('myfile.fits','mode','min');
```

See Also

copyHDU

matlab.io.fits.getHDUnum

Number of current HDU in FITS file

Syntax

```
N = getHDUnum(fptr)
```

Description

`N = getHDUnum(fptr)` returns the number of the current HDU in the FITS file. The primary array has HDU number 1.

This function corresponds to the `fits_get_hdu_num (ffghdn)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
n = fits.getHDUnum(fptr);
fits.closeFile(fptr);
```

See Also

`getHDUtype` | `getNumHDUs`

matlab.io.fits.getHDUtype

Type of current HDU

Syntax

```
htype = getHDUtype(fptr)
```

Description

`htype = getHDUtype(fptr)` returns the type of the current HDU in the FITS file. The possible values for `htype` are:

```
'IMAGE_HDU'
'ASCII_TBL'
'BINARY_TBL'
```

This function corresponds to the `fits_get_hdu_type (ffghdt)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
n = fits.getNumHDUs(fptr);
for j = 1:n
    fits.getHDUtype(fptr);
end
fits.closeFile(fptr);
```

See Also

`getHDUnum`

matlab.io.fits.getNumHDUs

Total number of HDUs in FITS file

Syntax

```
N = getNumHDUs(fptr)
```

Description

`N = getNumHDUs(fptr)` returns the number of completely defined HDUs in a FITS file. If a new HDU has just been added to the FITS file, then that last HDU will only be counted if it has been closed, or if data has been written to the HDU. The current HDU remains unchanged by this routine.

This function corresponds to the `fits_get_num_hdus (ffthdu)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
n = fits.getNumHDUs(fptr);
fits.closeFile(fptr);
```

See Also

`getHDUnum`

matlab.io.fits.movAbsHDU

Move to absolute HDU number

Syntax

```
htype = fits.movAbsHDU(fptr,HDUNUM)
```

Description

`htype = fits.movAbsHDU(fptr,HDUNUM)` moves to a specified absolute HDU number (starting with 1 for the primary array) in the FITS file. The possible values for `htype` are:

```
'IMAGE_HDU'
'ASCII_TBL'
'BINARY_TBL'
```

This function corresponds to the `fits_move_abs_hdu` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
n = fits.getNumHDUs(fptr);
for j = 1:n
    htype = fits.movAbsHDU(fptr,j);
    fprintf('HDU %d:  "%s"\n',j,htype);
end
fits.closeFile(fptr);
```

See Also

`getNumHDUs` | `movNamHDU` | `movRelHDU`

matlab.io.fits.movNamHDU

Move to first HDU having specific type and keyword values

Syntax

`movNamHDU(fptr,hdutype,EXTNAME,EXTVER)`

Description

`movNamHDU(fptr,hdutype,EXTNAME,EXTVER)` moves to the first HDU which has the specified extension type and `EXTNAME` and `EXTVER` keyword values (or `HDUNAME` and `HDUVER` keywords).

Specify the `hdutype` parameter as one of these character vectors or string scalars.

`'IMAGE_HDU'`

`'ASCII_TBL'`

`'BINARY_TBL'`

`'ANY_HDU'`

If `hdutype` is `'ANY_HDU'`, only the `EXTNAME` and `EXTVER` values are used to locate the correct extension. If the input value of `EXTVER` is 0, then the `EXTVER` keyword is ignored and the first HDU with a matching `EXTNAME` (or `HDUNAME`) keyword will be found.

This function corresponds to the `fits_movnam_hdu (ffmnhd)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movNamHDU(fptr,'IMAGE_HDU','quality',1);
fits.closeFile(fptr);
```

See Also

`movAbsHDU` | `movRelHDU`

matlab.io.fits.movRelHDU

Move relative number of HDUs from current HDU

Syntax

```
htype = moveRelHDU(fptr,nmove)
```

Description

`htype = moveRelHDU(fptr,nmove)` moves a relative number of HDUs forward or backward from the current HDU and returns the HDU type, `htype`, of the resulting HDU. The possible values for `htype` are:

```
'IMAGE_HDU'
'ASCII_TBL'
'BINARY_TBL'
```

This function corresponds to the `fits_movrel_hdu (ffmrhd)` function in the CFITSIO library C API.

Examples

Move through each HDU in succession, then move backwards twice by two HDUs.

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
n = fits.getNumHDUs(fptr);
for j = 1:n
    htype = fits.movAbsHDU(fptr,j);
    fprintf('HDU %d:  "%s"\n',j,htype);
end
htype = fits.movRelHDU(fptr,-2);
n = fits.getHDUnum(fptr);
fprintf('HDU %d:  "%s"\n',n,htype);
```

```
htype = fits.movRelHDU(fptr,-2);
n = fits.getHDUnum(fptr);
fprintf('HDU %d:  "%s"\n',n,htype);
fits.closeFile(fptr);
```

See Also

movAbsHDU | movNamHDU

matlab.io.fits.writeChecksum

Compute and write checksum for current HDU

Syntax

```
writeChecksum(fptr)
```

Description

`writeChecksum(fptr)` computes and writes the `DATASUM` and `CHECKSUM` keyword values for the current HDU into the current header. If the keywords already exist, their values are updated only if necessary (for example, if the file has been modified since the original keyword values were computed).

This function corresponds to the `fits_write_chksum (ffpcks)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'long_img',[10 20]);
fits.writeChecksum(fptr)
fits.closeFile(fptr);
fitsdisp('myfile.fits','mode','full');
```

See Also

`fitsdisp`

matlab.io.fits.imgCompress

Compress HDU from one file into another

Syntax

```
imgCompress(infptr,outfptr)
```

Description

`imgCompress(infptr,outfptr)` initializes the output HDU, copies all the keywords, and loops through the input image, compressing the data and writing the compressed data to the output HDU.

This function corresponds to the `fits_img_compress` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
infptr = fits.openFile('tst0012.fits');
outfptr = fits.createFile('myfile.fits');
fits.setCompressionType(outfptr,'rice');
fits.imgCompress(infptr,outfptr);
fits.closeFile(infptr);
fits.closeFile(outfptr);
```

See Also

`setCompressionType`

matlab.io.fits.isCompressedImg

Determine if current image is compressed

Syntax

```
TF = isCompressedImg(fptr)
```

Description

`TF = isCompressedImg(fptr)` returns true if the image in the current HDU is compressed.

This function corresponds to the `fits_is_compressed_image` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
bool = fits.isCompressedImg(fptr);
fits.closeFile(fptr);
```

See Also

`setCompressionType`

matlab.io.fits.setCompressionType

Set image compression type

Syntax

```
setCompressionType(fptr,comptype)
```

Description

`setCompressionType(fptr,comptype)` specifies the image compression algorithm that should be used when writing a FITS image.

Supported values for `comptype` include:

```
'GZIP'
'GZIP2'
'RICE'
'PLIO'
'HCOMPRESS'
'NOCOMPRESS'
```

This function corresponds to the `fits_set_compression_type` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.setCompressionType(fptr,'GZIP2');
fits.createImg(fptr,'long_img',[256 512]);
data = reshape(1:256*512,[256 512]);
data = int32(data);
fits.writeImg(fptr,data);
```

```
fits.closeFile(fptr);
fitsdisp('myfile.fits','mode','full');
```

See Also

`createImg` | `setTileDim`

matlab.io.fits.setHCompScale

Set scale parameter for HCOMPRESS algorithm

Syntax

```
setHCompScale(fptr,scale)
```

Description

`setHCompScale(fptr,scale)` sets the scale parameter to be used with the HCOMPRESS compression algorithm. Setting the scale parameter causes the algorithm to operate in lossy mode.

This function corresponds to the `fits_set_hcomp_scale` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
data = 50*ones(256,512,'double') + 10 * rand([256 512]);
fptr = fits.createFile('myfile.fits');
fits.setCompressionType(fptr,'HCOMPRESS_1');
fits.setHCompScale(fptr,2.5);
fits.createImg(fptr,'double_img',[256 512]);
fits.writeImg(fptr,data);
fits.closeFile(fptr);
fitsdisp('myfile.fits','mode','full');
```

See Also

`setCompressionType` | `setHCompSmooth`

matlab.io.fits.setHCompSmooth

Set smoothing for images compressed with HCOMPRESS

Syntax

```
setHCompSmooth(fptr,smooth)
```

Description

`setHCompSmooth(fptr,smooth)` sets the smoothing to be used when compressing an image with the HCOMPRESS algorithm. Setting either the scale or smoothing parameter causes the algorithm to operate in lossy mode.

This function corresponds to the `fits_set_hcomp_smooth` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
data = int32(50*ones(256,512,'double') + 10 * rand([256 512]));
fptr = fits.createFile('myfile.fits');
fits.setCompressionType(fptr,'HCOMPRESS');
fits.setHCompSmooth(fptr,1);
fits.createImg(fptr,'long_img',[256 512]);
fits.writeImg(fptr,data);
fits.closeFile(fptr);
fitsdisp('myfile.fits','mode','full');
```

See Also

`setCompressionType` | `setHCompScale`

matlab.io.fits.setTileDim

Set tile dimensions

Syntax

```
fits.setTileDim(fptr,tiledims)
```

Description

`fits.setTileDim(fptr,tiledims)` specifies the size of the image compression tiles to be used when creating a compressed image.

This function corresponds to the `fits_set_tile_dim` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.setCompressionType(fptr,'RICE_1');
fits.setTileDim(fptr,[64 128]);
fits.createImg(fptr,'byte_img',[256 512]);
data = ones(256,512,'uint8');
fits.writeImg(fptr,data);
fits.closeFile(fptr);
fitsdisp('myfile.fits','mode','full');
```

See Also

`setCompressionType`

matlab.io.fits.createTbl

Create new ASCII or binary table extension

Syntax

```
fptr = createTbl(fptr,tbltype,nrows,ttype,tform,tunit,extname)
```

Description

`fptr = createTbl(fptr,tbltype,nrows,ttype,tform,tunit,extname)` creates a new ASCII or bintable table extension. `ttype` must be either `'binary'` or `'ascii'`. The `nrows` argument gives the initial number of rows to be created in the table and should normally be zero. `tunit` specifies the units for each column, but can be an empty cell array if no units are desired. `extname` specifies the extension name, but can be omitted.

`tform` contains the format of the column, specified as a cell array of character vectors or a string array. For binary tables, the values should be in the form of `'rt'`, where `'r'` is the repeat count and `'t'` is one of the following letters.

`'A'`	ASCII character
`'B'`	Byte or `uint8`
`'C'`	Complex (single precision)
`'D'`	Double precision
`'E'`	Single precision
`'I'`	`int16`
`'J'`	`int32`
`'K'`	`int64`
`'L'`	Logical
`'M'`	Complex (double precision)
`'X'`	Bit (`int8` zeros and ones)

A column can also be specified as having variable-width if the `tform` value has the form `'1Pt'` or `'1Qt'`, where `'t'` specifies the data type as above.

For ASCII tables, `tform` contains values that take the form:

'Iw'	`int16` column with width `'w'`
'Aw'	ASCII column with width `'w'`
'Fww.dd'	Fixed point
'Eww.dd'	Single precision with width `'ww'` and precision `'dd'`
'Dww.dd'	Double precision with width `'ww'` and precision `'dd'`

This function corresponds to the `fits_create_tbl(ffcrtb)` function in the CFITSIO library C API.

Examples

Create a binary table. The first column contains strings of nine characters each. The second column contains four-element sequences of bits. The third column contains three-element sequences of `uint8` values. The fourth column contains double-precision scalars.

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
ttype = {'Col1','Col2','Col3','Col4'};
tform = {'9A','4X','3B','1D'};
tunit = {'m/s','kg','kg/m^3','candela'};
fits.createTbl(fptr,'binary',10,ttype,tform,tunit,'my-table');
fits.closeFile(fptr);
fitsdisp('myfile.fits');
```

Create a two-column table where the first column has a single double-precision value, but the second column has a variable-length double-precision value.

```
import matlab.io.*
fptr = fits.createFile('myfile2.fits');
ttype = {'Col1','Col2'};
tform = {'1D','1PD'};
fits.createTbl(fptr,'binary',0,ttype,tform);
```

```
fits.closeFile(fptr);
fitsdisp('myfile2.fits');
```

See Also

`createImg` | `insertATbl` | `insertBTbl` | `readCol` | `writeCol`

matlab.io.fits.deleteCol

Delete column from table

Syntax

```
deleteCol(fptr,colnum)
```

Description

`deleteCol(fptr,colnum)` deletes the column from an ASCII or binary table.

This function corresponds to the `fits_delete_col (ffdcol)` function in the CFITSIO library C API.

Examples

Delete the second column in a binary table.

```
import matlab.io.*
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','tst0012.fits');
copyfile(srcFile,'myfile.fits');
fileattrib('myfile.fits','+w');
fprintf('Before:  '); fitsdisp('myfile.fits','index',2,'mode','min');
fptr = fits.openFile('myfile.fits','readwrite');
fits.movAbsHDU(fptr,2);
fits.deleteCol(fptr,2);
fits.closeFile(fptr);
fprintf('After :  '); fitsdisp('myfile.fits','index',2,'mode','min');
```

See Also

`deleteRows`

matlab.io.fits.deleteRows

Delete rows from table

Syntax

```
deleteRows(fptr,firstrow,nrows)
```

Description

`deleteRows(fptr,firstrow,nrows)` deletes rows from an ASCII or binary table.

This function corresponds to the `fits_delete_rows (ffdrow)` function in the CFITSIO library C API.

Examples

Delete the second, third, and fourth rows in a binary table (second HDU).

```
import matlab.io.*
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','tst0012.fits');
copyfile(srcFile,'myfile.fits');
fileattrib('myfile.fits','+w');
fprintf('Before:  '); fitsdisp('myfile.fits','index',2,'mode','min');
fptr = fits.openFile('myfile.fits','readwrite');
fits.movAbsHDU(fptr,2);
fits.deleteRows(fptr,2,2);
fits.closeFile(fptr);
fprintf('After :  '); fitsdisp('myfile.fits','index',2,'mode','min');
```

See Also

`deleteCol` | `insertRows`

matlab.io.fits.insertRows

Insert rows into table

Syntax

```
insertRows(fptr,firstrow,nrows)
```

Description

`insertRows(fptr,firstrow,nrows)` inserts rows into an ASCII or binary table. `firstrow` is a one-based number.

This function corresponds to the `fits_insert_rows (ffirow)` function in the CFITSIO library C API.

Examples

Insert five rows into an empty table.

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
ttype = {'Col1','Col2'};
tform = {'3A','1D'};
tunit = {'m/s','candela'};
fits.createTbl(fptr,'binary',0,ttype,tform,tunit,'my-table');
fits.insertRows(fptr,1,5);
fits.closeFile(fptr);
fitsdisp('myfile.fits','index',2);
```

See Also

`deleteRows` | `insertCol`

matlab.io.fits.getAColParms

ASCII table information

Syntax

```
[ttype,tbcol,tunit,tform,scale,zero,nulstr,tdisp] =
getAColParms(fptr,colnum)
```

Description

`[ttype,tbcol,tunit,tform,scale,zero,nulstr,tdisp] = getAColParms(fptr,colnum)` gets information about an existing ASCII table column.

This function corresponds to the `fits_get_acolparms (fffacl)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movAbsHDU(fptr,5);
[ttype,tbcol,tunit,tform,scale,zero,nulstr,tdisp] = fits.getAColParms(fptr,2);
fits.closeFile(fptr);
```

See Also

`getBColParms`

matlab.io.fits.getBColParms

Binary table information

Syntax

```
[ttype,tunit,typechar,repeat,scale,zero,nulval,tdisp] =
getBColParms(fptr,colnum)
```

Description

`[ttype,tunit,typechar,repeat,scale,zero,nulval,tdisp] = getBColParms(fptr,colnum)` gets information about an existing binary table column.

This function corresponds to the `fits_get_bcolparms (ffgbcl)` function in the CFITSIO library C API.

Examples

Get information about the second column in a binary table.

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movAbsHDU(fptr,2);
[ttype,tunit,typechar,repeat,scale,zero,nulval,tdisp]= fits.getBColParms(fptr,2);
fits.closeFile(fptr);
```

See Also

`getAColParms`

matlab.io.fits.getColName

Table column name

Syntax

```
[colnum,colname] = getColNum(fptr,templt,casesen)
```

Description

`[colnum,colname] = getColNum(fptr,templt,casesen)` gets the table column numbers and names of the columns whose names match an input template name. If `casesen` is true, then the column name match is case-sensitive. `casesen` defaults to false.

Specify the input column name template `templt` as a character vector or string scalar. `templt` may be either the exact name of the column to be searched for, or it may contain wildcard characters (*, ?, or #), or it may contain the integer number of the desired column (with the first column = 1). The `'*'` wildcard character matches any sequence of characters (including zero characters) and the `'?'` character matches any single character. The # wildcard matches any sequence of consecutive decimal digit characters (0-9).

Examples

Return all the columns starting with the letter `'C'`.

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movAbsHDU(fptr,2);
[nums,names] = fits.getColName(fptr,'C*');
fits.closeFile(fptr);
```

See Also

`getAColParms` | `getBColParms`

matlab.io.fits.getColType

Scaled column data type, repeat value, width

Syntax

```
[dtype,repeat,width] = getColType(fptr,colnum)
```

Description

`[dtype,repeat,width] = getColType(fptr,colnum)` returns the data type, vector repeat value, and the width in bytes of a column in an ASCII or binary table.

This function corresponds to the `fits_get_coltypell (ffgtclll)` function in the CFITSIO library C API.

Examples

Get information about the 'FLUX' column in the second HDU.

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movAbsHDU(fptr,2);
[dtype,repeat,width] = fits.getColType(fptr,5);
fits.closeFile(fptr);
```

See Also

`getEqColType`

matlab.io.fits.getEqColType

Column data type, repeat value, width

Syntax

```
[dtype,repeat,width] = getEqColType(fptr,colnum)
```

Description

`[dtype,repeat,width] = getEqColType(fptr,colnum)` returns the equivalent data type needed to store the scaled column data type, the vector repeat value, and the width in bytes of a column in an ASCII or binary table.

This function corresponds to the `fits_get_eqcoltypell (ffeqtyll)` function in the CFITSIO library C API.

Examples

Get information about the 'FLUX' column in the second HDU.

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movAbsHDU(fptr,2);
[dtype,repeat,width] = fits.getEqColType(fptr,5);
fits.closeFile(fptr);
```

See Also

`getColType`

matlab.io.fits.getNumCols

Number of columns in table

Syntax

```
ncols = getNumCols(fptr)
```

Description

`ncols = getNumCols(fptr)` gets the number of columns in the current FITS table. This function corresponds to the `fits_get_num_cols (ffgncl)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movAbsHDU(fptr,2);
ncols = fits.getNumCols(fptr);
nrows = fits.getNumRows(fptr);
fits.closeFile(fptr);
```

See Also

`getNumRows`

matlab.io.fits.getNumRows

Number of rows in table

Syntax

```
nrows = getNumRows(fptr)
```

Description

`nrows = getNumRows(fptr)` gets the number of rows in the current FITS table. This function corresponds to the `fits_get_num_rowsll` (ffgnrwll) function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movAbsHDU(fptr,2);
ncols = fits.getNumCols(fptr);
nrows = fits.getNumRows(fptr);
fits.closeFile(fptr);
```

See Also

`getNumCols`

matlab.io.fits.insertCol

Insert column into table

Syntax

```
insertCol(fptr,colnum,ttype,tform)
```

Description

`insertCol(fptr,colnum,ttype,tform)` inserts a column into an ASCII or binary table. Specify `ttype` and `tform` as character vectors or string scalars.

This function corresponds to the `fits_insert_col (fficol)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
ttype = {'Col1','Col2'};
tform = {'3A','1D'};
tunit = {'m/s','candela'};
fits.createTbl(fptr,'binary',0,ttype,tform,tunit,'my-table');
fits.insertCol(fptr,3,'Col3','3D');
fits.closeFile(fptr);
fitsdisp('myfile.fits','index',2);
```

See Also

`insertRows`

matlab.io.fits.insertATbl

Insert ASCII table after current HDU

Syntax

```
insertATbl(fptr,rowlen,nrows,ttype,tbcol,tform,tunit,extname)
```

Description

`insertATbl(fptr,rowlen,nrows,ttype,tbcol,tform,tunit,extname)` inserts a new ASCII table extension immediately following the current HDU. Any following extensions are shifted down to make room for the new extension. If there are no other following extensions, then the new table extension is simply appended to the end of the file. If the FITS file is currently empty then this routine creates a dummy primary array before appending the table to it. The new extension becomes the current HDU. If `rowlen` is 0, then CFITSIO calculates the default `rowlen` based on the `tbcol` and `ttype` values.

Specify `tform` as a cell array of character vectors or a string array that can take the following forms. In each case, `'w'` and `'ww'` represent the widths of the ASCII columns.

'Iw'	`int16` column
'Aw'	ASCII column
'Fww.dd'	Fixed point with `'dd'` digits after the decimal point
'Eww.dd'	Single precision with `'dd'` digits of precision
'Dww.dd'	Double precision with `'dd'` digits of precision

Binary tables are recommended instead of ASCII tables.

This function corresponds to the `fits_insert_atbl(ffitab)` function in the CFITSIO library C API.

Examples

Create an ASCII table between two images.

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
fits.createImg(fptr,'uint8',[20 30]);
fits.createImg(fptr,'int16',[30 40]);
fits.movRelHDU(fptr,-1);
ttype = {'Name','Short','Fix','Double'};
tbcol = [1 17 28 43];
tform = {'A15','I10','F14.2','D12.4'};
tunit = {'','m**2','cm','km/s'};
fits.insertATbl(fptr,0,0,ttype,tbcol,tform,tunit,'my-table');
fits.writeCol(fptr,1,1,char('abracadabra','hocus-pocus'));
fits.writeCol(fptr,2,1,int16([0; 1]));
fits.writeCol(fptr,3,1,[12.4; 4/3]);
fits.writeCol(fptr,4,1,[12.4; 4e8/3]);
fits.closeFile(fptr);
fitsdisp('myfile.fits','mode','min');
```

See Also

`createTbl` | `insertBTbl`

matlab.io.fits.insertBTbl

Insert binary table after current HDU

Syntax

```
insertBTbl(fptr,nrows,ttype,tform,tunit,extname,pcount)
```

Description

`insertBTbl(fptr,nrows,ttype,tform,tunit,extname,pcount)` inserts a new binary table extension immediately following the current HDU. Any following extensions are shifted down to make room for the new extension. If there are no other following extensions then the new table extension is simply appended to the end of the file. If the FITS file is currently empty then this routine creates a dummy primary array before appending the table to it. The new extension becomes the CHDU. If there are following extensions in the file and if the table contains variable-length array columns then `pcount` must specify the expected final size of the data heap. Otherwise, `pcount` must be zero.

Specify `ttype`, `tform`, and `tunits` as cell array of character vectors or a string array.

This function corresponds to the `fits_insert_btbl (ffibin)` function in the CFITSIO library C API.

Examples

Create a table following the primary array. Then, insert a new table just before it.

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
ttype = {'Col1','Col2'};
tform = {'9A','1D'};
tunit = {'m/s','candela'};
fits.createTbl(fptr,'binary',10,ttype,tform,tunit,'my-table');
fits.movRelHDU(fptr,-1);
fits.insertBTbl(fptr,5,ttype,tform,tunit,'my-new-table',0);
```

```
fits.closeFile(fptr);
fitsdisp('myfile.fits');
```

See Also

`createTbl` | `insertATbl`

matlab.io.fits.readATblHdr

Read header information from current ASCII table

Syntax

```
[rowlen,nrows,ttype,tbcol,tform,tunit,extname] = readATblHdr(fptr)
```

Description

`[rowlen,nrows,ttype,tbcol,tform,tunit,extname] = readATblHdr(fptr)` reads header information for the current ASCII table.

This function corresponds to the `fits_read_atblhdrll (ffghtbll)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movAbsHDU(fptr,5);
[rowlen,nrows,ttype,tbcol,tform,tunit,extname] = fits.readATblHdr(fptr);
fits.closeFile(fptr);
```

See Also

`readBTblHdr`

matlab.io.fits.readBTblHdr

Read header information from current binary table

Syntax

```
[nrows,ttype,tform,tunit,extname,pcount] = readBTblHdr(fptr)
```

Description

`[nrows,ttype,tform,tunit,extname,pcount] = readBTblHdr(fptr)` reads header information for the current binary table.

This function corresponds to the `fits_read_btblhdrll (ffghbnll)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movAbsHDU(fptr,2);
[nrows,ttype,tform,tunit,extname,pcount] = fits.readBTblHdr(fptr);
fits.closeFile(fptr);
```

See Also

`readATblHdr`

matlab.io.fits.readCol

Read rows of ASCII or binary table column

Syntax

```
[coldata,nullval] = readCol(fptr,colnum)
[coldata,nullval] = readCol(fptr,colnum,firstrow,numrows)
```

Description

`[coldata,nullval] = readCol(fptr,colnum)` reads an entire column from an ASCII or binary table column. `nullval` is a logical array specifying if a particular element of `coldata` should be treated as undefined. It is the same size as `coldata`.

`[coldata,nullval] = readCol(fptr,colnum,firstrow,numrows)` reads a subsection of rows from an ASCII or binary table column.

The MATLAB data type returned by `readCol` corresponds to the data type returned by `getEqColType`.

This function corresponds to the `fits_read_col (ffgcv)` function in the CFITSIO library C API.

Examples

Read an entire column.

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movAbsHDU(fptr,2);
colnum = fits.getColName(fptr,'flux');
fluxdata = fits.readCol(fptr,colnum);
fits.closeFile(fptr);
```

Read the first five rows in a column.

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movAbsHDU(fptr,2);
colnum = fits.getColName(fptr,'flux');
fluxdata = fits.readCol(fptr,colnum,1,5);
fits.closeFile(fptr);
```

See Also

writeCol

matlab.io.fits.setTscale

Reset image scaling

Syntax

```
setTscale(fptr,colnum,tscale,tzero)
```

Description

`setTscale(fptr,colnum,tscale,tzero)` resets the scaling factors for a table column according to the equation:

```
output = (FITS array) * tscale + tzero
```

The inverse formula is used when writing data values to the FITS file.

This only affects the automatic scaling performed when the data elements are read. It does not change the `tscale` and `tzero` keyword values.

Examples

Turn off automatic scaling in a table column where the `tscale` and `tzero` keywords are present.

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
fits.movAbsHDU(fptr,2);
scaled_data = fits.readCol(fptr,3);
fits.setTscale(fptr,3,1.0,0.0);
unscaled_data = fits.readCol(fptr,3);
fits.closeFile(fptr);
```

See Also

`readImg`

matlab.io.fits.writeCol

Write elements into ASCII or binary table column

Syntax

```
writeCol(fptr,colnum,firstrow,coldata)
```

Description

`writeCol(fptr,colnum,firstrow,coldata)` writes elements into an ASCII or binary table extension column.

Note If you use the `writeCol` function to write complex data to a column defined by a noncomplex data type, then `writeCol` ignores the imaginary part and only writes the real part to the column. To write complex data correctly, define the format of the table column appropriately using the `tform` argument of the `createTbl` function.

When writing rows of data to a variable length field, `coldata` must be a cell array.

This function corresponds to the `fits_write_col (ffpcl)` function in the CFITSIO library C API.

Examples

Write to a table with ASCII, `uint8`, double-precision, and variable-length double-precision columns.

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
ttype = {'Col1','Col2','Col3','Col4'};
tform = {'3A','3B','1D','1PD'};
tunit = {'m/s','kg/m^3','candela','parsec'};
fits.createTbl(fptr,'binary',0,ttype,tform,tunit,'my-table');
```

```
fits.writeCol(fptr,1,1,['dog'; 'cat']);
fits.writeCol(fptr,2,1,[0 1 2; 3 4 5; 6 7 8; 9 10 11]);
fits.writeCol(fptr,3,1,[1; 2; 3; 4]);
fits.writeCol(fptr,4,1,{1;[1 2];[1 2 3];[1 2 3 4]});
fits.closeFile(fptr);
fitsdisp('myfile.fits','index',2,'mode','full');
```

Write to a table with logical, bit, double precision, and variable-length complex single-precision columns.

```
import matlab.io.*
fptr = fits.createFile('myfile.fits');
ttype = {'Col1','Col2','Col3','Col4'};
tform = {'2L','3X','1D','1PC'};
tunit = {'','kg/m^3','candela','parsec'};
fits.createTbl(fptr,'binary',0,ttype,tform,tunit,'my-table');
fits.writeCol(fptr,1,1,[false false; true false]);
fits.writeCol(fptr,2,1,int8([0 1 1; 1 1 1; 1 1 1; 1 0 1]));
fits.writeCol(fptr,3,1,[1; 2; 3; 4]);
data = cell(4,1);
data{1} = single(1);
data{2} = single(1+2j);
data{3} = single([1j 2 3+j]);
data{4} = single([1 2+3j 3 4]);
fits.writeCol(fptr,4,1,data);
fits.closeFile(fptr);
fitsdisp('myfile.fits','index',2,'mode','full');
```

See Also

`createTbl` | `readCol`

matlab.io.fits.getConstantValue

Numeric value of named constant

Syntax

```
N = getConstantValue(name)
```

Description

`N = getConstantValue(name)` returns the numeric value corresponding to the named CFITSIO constant specified as a character vector or string scalar.

Examples

```
import matlab.io.*
n = fits.getConstantValue('BYTE_IMG');
```

matlab.io.fits.getVersion

Revision number of the CFITSIO library

Syntax

```
V = getVersion()
```

Description

`V = getVersion()` returns the revision number of the CFITSIO library. This function corresponds to the `fits_get_version (ffvers)` function in the CFITSIO library C API.

Examples

```
import matlab.io.*
v = fits.getVersion();
```

matlab.io.fits.getOpenFiles

List of open FITS files

Syntax

```
fptrs = getOpenFiles()
```

Description

`fptrs = getOpenFiles()` returns a list of file pointers of all open FITS files.

Examples

```
import matlab.io.*
fptr = fits.openFile('tst0012.fits');
clear fptr;
fptr = fits.getOpenFiles();
fits.closeFile(fptr);
```

See Also

`closeFile` | `openDiskFile` | `openFile`

matlab.io.hdf4.sd

Interact directly with HDF4 multifile scientific data set (SD) interface

Description

To use these MATLAB functions, you should be familiar with the HDF SD C API. In most cases, the syntax of the MATLAB function is similar to the syntax of the corresponding HDF library function. The functions are implemented as the package `matlab.io.hdf4.sd`. To use this package, prefix the function name with a package path, or use the import function to add the package to the current import list, prior to calling the function, for example,

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','read');
```

Access

matlab.io.hdf4.sd.close	Terminate access to SD interface
matlab.io.hdf4.sd.endAccess	Terminate access to data set
matlab.io.hdf4.sd.getFilename	Name of file
matlab.io.hdf4.sd.select	Identifier of data set with specified index
matlab.io.hdf4.sd.setExternalFile	Store data in external file
matlab.io.hdf4.sd.start	Open HDF file and initialize SD interface

Read/Write

matlab.io.hdf4.sd.create	Create new data set
matlab.io.hdf4.sd.readData	Read subsample of data
matlab.io.hdf4.sd.setFillMode	Set current fill mode of file
matlab.io.hdf4.sd.writeData	Write to data set

Inquiry

matlab.io.hdf4.sd.fileInfo	Number of data sets and global attributes in file
matlab.io.hdf4.sd.getCompInfo	Information about data set compression
matlab.io.hdf4.sd.getFillValue	Fill value for data set
matlab.io.hdf4.sd.getInfo	Information about data set
matlab.io.hdf4.sd.idToRef	Reference number corresponding to data set identifier
matlab.io.hdf4.sd.idType	Type of object
matlab.io.hdf4.sd.isCoordVar	Determine if data set is a coordinate variable
matlab.io.hdf4.sd.isRecord	Determine if data set is appendable
matlab.io.hdf4.sd.nameToIndex	Index value of named data set
matlab.io.hdf4.sd.nameToIndices	List of data sets with same name
matlab.io.hdf4.sd.refToIndex	Index of data set corresponding to reference number

Dimensions

matlab.io.hdf4.sd.dimInfo	Information about dimension
matlab.io.hdf4.sd.getDimID	Dimension identifier
matlab.io.hdf4.sd.getDimScale	Scale data for dimension
matlab.io.hdf4.sd.setDimName	Associate name with dimension
matlab.io.hdf4.sd.setDimScale	Set scale values for dimension

User-defined Attributes

matlab.io.hdf4.sd.attrInfo	Information about attribute
matlab.io.hdf4.sd.findAttr	Index of specified attribute
matlab.io.hdf4.sd.readAttr	Read attribute value
matlab.io.hdf4.sd.setAttr	Write attribute value

Predefined Attributes

matlab.io.hdf4.sd.getCal	Data set calibration information
matlab.io.hdf4.sd.getDataStrs	Predefined attributes for data set
matlab.io.hdf4.sd.getDimStrs	Predefined attributes for dimension
matlab.io.hdf4.sd.getFillValue	Fill value for data set
matlab.io.hdf4.sd.getRange	Maximum and minimum range values
matlab.io.hdf4.sd.setCal	Set data set calibration information
matlab.io.hdf4.sd.setDataStrs	Set predefined attributes for data set
matlab.io.hdf4.sd.setDimStrs	Set label, unit, and format attributes
matlab.io.hdf4.sd.setFillValue	Set fill value for data set
matlab.io.hdf4.sd.setRange	Set maximum and minimum range value for data set

Chunking/Tiling Operations

matlab.io.hdf4.sd.getChunkInfo	Chunk size for data set
matlab.io.hdf4.sd.readChunk	Read chunk from data set
matlab.io.hdf4.sd.setChunk	Set chunk size and compression method of data set
matlab.io.hdf4.sd.writeChunk	Write chunk to data set

Compression

matlab.io.hdf4.sd.setCompress	Set compression method of data set
matlab.io.hdf4.sd.setNBitDataSet	Specify nonstandard bit length for data set values

matlab.io.hdf4.sd.attrInfo

Package: `matlab.io.hdf4.sd`

Information about attribute

Syntax

```
[name,datatype,nelts] = attrInfo(objID,idx)
```

Description

`[name,datatype,nelts] = attrInfo(objID,idx)` returns the name, data type, and number of elements in the specified attribute. The attribute is specified by its zero-based index value. `objID` can be either an SD interface identifier, a data set identifier, or a dimension identifier.

This function corresponds to the `SDattrinfo` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.findAttr(sdID,'creation_date');
[name,datatype,nelts] = sd.attrInfo(sdID,idx);
data = sd.readAttr(sdID,idx);
sd.close(sdID);
```

See Also

`sd.findAttr`

matlab.io.hdf4.sd.close

Package: `matlab.io.hdf4.sd`

Terminate access to SD interface

Syntax

```
sd.close(sdID)
```

Description

`sd.close(sdID)` closes the file identified by `sdID`.

This function corresponds to the `SDend` function in the HDF C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.start`

matlab.io.hdf4.sd.create

Package: `matlab.io.hdf4.sd`

Create new data set

Syntax

```
sdsID = create(sdID,name,datatype,dims)
```

Description

`sdsID = create(sdID,name,datatype,dims)` creates a data set with the given name `name`, data type `datatype`, and dimension sizes `dims`.

To create a data set with an unlimited dimension, the last value in `dims` should be set to 0.

This function corresponds to the `SDcreate` function in the HDF library C API, but because MATLAB uses FORTRAN-style ordering, the `dims` parameter is reversed with respect to the C library API.

Examples

Create a 3D data set with an unlimited dimension.

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[10 20 0]);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.endAccess`

matlab.io.hdf4.sd.dimInfo

Package: `matlab.io.hdf4.sd`

Information about dimension

Syntax

```
[name,dimlen,datatype,nattrs] = dimInfo(dimID)
```

Description

`[name,dimlen,datatype,nattrs] = dimInfo(dimID)` returns the name, length, data type, and number of attributes of the specified dimension.

This function corresponds to the `SDdiminfo` function in the HDF library C API.

Examples

Read a 2-by-3 portion of a data set.

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'latitude');
sdsID = sd.select(sdID,idx);
dimID = sd.getDimID(sdsID,0);
[name,dimlen,datatype,nattrs] = sd.dimInfo(dimID);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.getDimID`

matlab.io.hdf4.sd.endAccess

Package: `matlab.io.hdf4.sd`

Terminate access to data set

Syntax

```
sd.endAccess(sdsID)
```

Description

`sd.endAccess(sdsID)` terminates access to the data set identified by `sdsID`. Failing to call this function after all operations on the specified data set are complete may result in loss of data.

This function corresponds to the `SDendaccess` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.close` | `sd.select`

matlab.io.hdf4.sd.fileInfo

Package: `matlab.io.hdf4.sd`

Number of data sets and global attributes in file

Syntax

```
[ndatasets,ngatts] = fileInfo(sdID)
```

Description

`[ndatasets,ngatts] = fileInfo(sdID)` returns the number of data sets `ndatasets` and the number of global attributes `ngatts` in the file identified by `sdID`.

`ndatasets` includes the number of coordinate variable data sets.

This function corresponds to the `SDfileinfo` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
[ndatasets,ngatts] = sd.fileInfo(sdID);
sd.close(sdID);
```

See Also

`sd.getInfo`

matlab.io.hdf4.sd.findAttr

Package: `matlab.io.hdf4.sd`

Index of specified attribute

Syntax

```
idx = findAttr(objID,attrname)
```

Description

`idx = findAttr(objID,attrname)` returns the index of the attribute specified by `attrname`. The `objID` input can be either an SD interface identifier, a data set identifier, or a dimension identifier.

The function corresponds to the `SDfindattr` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.findAttr(sdID,'creation_date');
data = sd.readAttr(sdID,idx);
sd.close(sdID);
```

See Also

`sd.getDimID` | `sd.readAttr` | `sd.select` | `sd.start`

matlab.io.hdf4.sd.getCal

Package: `matlab.io.hdf4.sd`

Data set calibration information

Syntax

```
[cal,calErr,offset,offsetErr,datatype] = getCal(sdsID)
```

Description

`[cal,calErr,offset,offsetErr,datatype] = getCal(sdsID)` retrieves the calibration information associated with a data set.

This function corresponds to the `SDgetcal` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
[cal,calErr,offset,offsetErr,dtype] = sd.getCal(sdsID);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.setCal`

matlab.io.hdf4.sd.getChunkInfo

Package: `matlab.io.hdf4.sd`

Chunk size for data set

Syntax

```
chunkDims = getChunkInfo(sdsID)
```

Description

`chunkDims = getChunkInfo(sdsID)` returns the chunk size for the data set specified by `sdsID`. If a data set is chunked, the dimensions of the chunks is returned in `chunkDims`. Otherwise `chunkDims` is `[]`.

This function corresponds to the `SDgetchunkinfo` function in the HDF library C API, but because MATLAB uses FORTRAN-style ordering, the `chunkDims` parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
cdims = sd.getChunkInfo(sdsID);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.getCompInfo` | `sd.setChunk`

matlab.io.hdf4.sd.getCompInfo

Package: `matlab.io.hdf4.sd`

Information about data set compression

Syntax

```
[comptype,compparms] = getCompType(sdsID)
```

Description

`[comptype,compparms] = getCompType(sdsID)` retrieves the compression type and compression information for a data set. `comptype` can be one of the following values.

`'none'`	No compression
`'rle'`	Run-length encoding
`'nbit'`	NBIT compression
`'skphuff'`	Skipping Huffman compression
`'deflate'`	GZIP compression
`'szip'`	SZIP compression

If `comptype` is `'none'` or `'rle'`, then `compparms` is `[]`.

If `comptype` is `'nbit'`, then `compparms` is a 4-element array.

`compparm(1)`	sign_ext
`compparm(2)`	fill_one
`compparm(3)`	start_bit
`compparm(4)`	bit_len

If `comptype` is `'deflate'`, then `compparms` contains the deflation value, a number between 0 and 9.

If `comptype` is `'szip'`, them `compparms` is a 5-element array. Consult the HDF Reference Manual for details on SZIP compression.

This function corresponds to the `SDgetcompinfo` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[100 50]);
sd.setCompress(sdsID,'deflate',5);
[comptype,compparm] = sd.getCompInfo(sdsID);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.setCompress` | `sd.setNBitDataSet`

matlab.io.hdf4.sd.getDataStrs

Package: `matlab.io.hdf4.sd`

Predefined attributes for data set

Syntax

```
[label,unit,format,coordsys] = getDataStrs(sdsID)
[label,unit,format,coordsys] = getDataStrs(sdsID,maxlen)
```

Description

`[label,unit,format,coordsys] = getDataStrs(sdsID)` returns the label, unit, format, and coordsys attributes for the data set identified by `sdsID`.

`[label,unit,format,coordsys] = getDataStrs(sdsID,maxlen)` returns the label, unit, format, and coordsys attributes for the data set identified by `sdsID`. The `maxlen` input is the maximum length of the attribute text. It defaults to 1000 if not specified.

This function corresponds to the `SDgetdatastrs` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
[label,unit,fmt,coordsys] = sd.getDataStrs(sdsID);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.setDataStrs`

matlab.io.hdf4.sd.getDimID

Package: `matlab.io.hdf4.sd`

Dimension identifier

Syntax

```
dimID = getDimID(sdsID,dimnumber)
```

Description

`dimID = getDimID(sdsID,dimnumber)` returns the identifier of the dimension given its index.

Note MATLAB uses Fortran-style indexing while the HDF library uses C-style indexing. The order of the dimension identifiers retrieved with `sd.getDimID` are reversed from what would be retrieved via the C API.

This function corresponds to the `SDgetdimid` function in the HDF library C API.

Examples

Read an entire data set.

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
dimID0 = sd.getDimID(sdsID,0);
dimID1 = sd.getDimID(sdsID,1);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.setDimName`

matlab.io.hdf4.sd.getDimScale

Package: `matlab.io.hdf4.sd`

Scale data for dimension

Syntax

```
scale = getDimScale(dimID)
```

Description

`scale = getDimScale(dimID)` returns the scale values of the dimension identified by `dimID`.

This function corresponds to the `SDgetdimscale` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',20);
dimID = sd.getDimID(sdsID,0);
sd.setDimName(dimID,'x');
sd.setDimScale(dimID,0:5:95);
sd.endAccess(sdsID);
sd.close(sdID);
sdID = sd.start('myfile.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
dimID = sd.getDimID(sdsID,0);
scale = sd.getDimScale(dimID);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.dimInfo` | `sd.setDimScale`

matlab.io.hdf4.sd.getDimStrs

Package: `matlab.io.hdf4.sd`

Predefined attributes for dimension

Syntax

```
[label,unit,format] = getDimStrs(dimID)
```

Description

`[label,unit,format] = getDimStrs(dimID)` returns the label, unit, and format attributes for the dimension identified by `dimID`.

This function corresponds to the `SDgetdimstrs` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',20);
dimID = sd.getDimID(sdsID,0);
sd.setDimName(dimID,'x');
sd.setDimStrs(dimID,'xdim','none','%d');
sd.endAccess(sdsID);
sd.close(sdID);
sdID = sd.start('myfile.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
dimID = sd.getDimID(sdsID,0);
[label,unit,fmt] = sd.getDimStrs(dimID);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.setDimStrs`

matlab.io.hdf4.sd.getFilename

Package: `matlab.io.hdf4.sd`

Name of file

Syntax

```
filename = getFilename(sdID)
```

Description

`filename = getFilename(sdID)` retrieves the name of a file previously opened with the `sd` package with identifier `sdID`.

This function corresponds to the `SDgetfilename` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
filename = sd.getFilename(sdID);
sd.close(sdID);
```

See Also

`sd.getInfo` | `sd.start`

matlab.io.hdf4.sd.getFillValue

Package: `matlab.io.hdf4.sd`

Fill value for data set

Syntax

```
fillvalue = getFillValue(sdsID)
```

Description

`fillvalue = getFillValue(sdsID)` returns the fill value for a data set.

This function corresponds to the `SDgetfillvalue` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
fillvalue = sd.getFillValue(sdsID);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.setFillValue`

matlab.io.hdf4.sd.getInfo

Package: `matlab.io.hdf4.sd`

Information about data set

Syntax

```
[name,dims,datatype,nattrs] = getInfo(sdsID)
```

Description

`[name,dims,datatype,nattrs] = getInfo(sdsID)` returns the name, extents, and number of attributes of the data set identified by `sdsID`.

This function corresponds to the `SDgetinfo` function in the HDF library C API, but because MATLAB uses FORTRAN-style ordering, the dims parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
[name,dims,datatype,nattrs] = sd.getInfo(sdsID);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.attrInfo` | `sd.dimInfo` | `sd.fileInfo`

matlab.io.hdf4.sd.getRange

Package: `matlab.io.hdf4.sd`

Maximum and minimum range values

Syntax

```
[maxval,minval] = getRange(sdsID)
```

Description

`[maxval,minval] = getRange(sdsID)` retrieves the "valid_range" two-element attribute value.

This function corresponds to the `SDgetrange` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
[maxval,minval] = sd.getRange(sdsID);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.setRange`

matlab.io.hdf4.sd.idToRef

Package: `matlab.io.hdf4.sd`

Reference number corresponding to data set identifier

Syntax

```
ref = idToRef(sdsID)
```

Description

`ref = idToRef(sdsID)` returns the reference number corresponding to the data set.

This function corresponds to the `SDidtoref` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
ref = sd.idToRef(sdsID);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.refToIndex`

matlab.io.hdf4.sd.idType

Package: `matlab.io.hdf4.sd`

Type of object

Syntax

```
objtype = idType(objID)
```

Description

`objtype = idType(objID)` returns the type of object that `objID` represents. Possible values for `objtype` are:

`'NOT_SDAPI_ID'`	The object is not an HDF SD identifier.
`'SD_ID'`	The object is an SD identifier (file handle) .
`'SDS_ID'`	The object is a data set identifier.
`'DIM_ID'`	The object is a dimension identifier.

This function corresponds to the `SDidtype` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
objType = sd.idType(sdID);
sd.close(sdID);
```

matlab.io.hdf4.sd.isCoordVar

Package: `matlab.io.hdf4.sd`

Determine if data set is a coordinate variable

Syntax

```
TF = isCoordVar(sdsID)
```

Description

`TF = isCoordVar(sdsID)` returns `true` if a data set is a coordinate variable and returns `false` otherwise.

This function corresponds to the `SDiscoordvar` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
ndataset = sd.fileInfo(sdID);
for idx = 0:ndataset-1
    sdsID = sd.select(sdID,idx);
    sdsName = sd.getInfo(sdsID);
    fprintf('%s (index %d) ', sdsName, idx);
    if ( sd.isCoordVar(sdsID) )
        fprintf('is a coordinate variable.\n');
    else
        fprintf('is not a coordinate variable.\n');
    end
    sd.endAccess(sdsID);
end
sd.close(sdID);
```

See Also

`sd.isRecord`

matlab.io.hdf4.sd.isRecord

Package: `matlab.io.hdf4.sd`

Determine if data set is appendable

Syntax

```
TF = isRecord(sdsID)
```

Description

`TF = isRecord(sdsID)` determines if the data set specified by `sdsID` is appendable, meaning that the slowest changing dimension is unlimited.

This function corresponds to the `SDisrecord` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
ndataset = sd.fileInfo(sdID);
for idx = 0:ndataset-1
    sdsID = sd.select(sdID,idx);
    sdsName = sd.getInfo(sdsID);
    if sd.isRecord(sdsID)
        fprintf('%s is a record variable.\n',sdsName);
    else
        fprintf('%s is not a record variable.\n',sdsName);
    end
    sd.endAccess(sdsID);
end
sd.close(sdID);
```

See Also

`sd.isCoordVar`

matlab.io.hdf4.sd.nameToIndex

Package: `matlab.io.hdf4.sd`

Index value of named data set

Syntax

```
idx = nameToIndex(sdID,sdsname)
```

Description

`idx = nameToIndex(sdID,sdsname)` returns the index of the data set with the name specified by `sdsname`. If there is more than one data set with the same name, the routine returns the index of the first one.

This function corresponds to the `SDnametoindex` function in the HDF C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf','read');
idx = sd.nameToIndex(sdID,'temperature');
sd.close(sdID);
```

See Also

`sd.select`

matlab.io.hdf4.sd.nameToIndices

Package: `matlab.io.hdf4.sd`

List of data sets with same name

Syntax

```
varstruct = nameToIndices(sdID,sdsname)
```

Description

`varstruct = nameToIndices(sdID,sdsname)` returns a structure array for all data sets with the same name. Each element of `varstruct` has two fields.

`'index'`	Index of data set
`'type'`	Type of data set, either `'SDSVAR'`, `'COORDVAR'`, or `'UNKNOWN'`

This function corresponds to the `SDnametoindices` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
varlist = sd.nameToIndices(sdID,'latitude');
sd.close(sdID);
```

See Also

`sd.isCoordVar` | `sd.setDimScale`

matlab.io.hdf4.sd.readAttr

Package: `matlab.io.hdf4.sd`

Read attribute value

Syntax

```
data = readAttr(objID,idx)
```

Description

`data = readAttr(objID,idx)` reads the value of the attribute specified by index `idx`. The `objID` input can be an SD interface identifier, a data set identifier, or a dimension identifier. `idx` is a zero-based index.

This function corresponds to the `SDreadattr` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.findAttr(sdID,'creation_date');
data = sd.readAttr(sdID,idx);
sd.close(sdID);
```

See Also

`sd.findAttr` | `sd.setAttr`

matlab.io.hdf4.sd.readChunk

Package: `matlab.io.hdf4.sd`

Read chunk from data set

Syntax

```
datachunk = readChunk(sdsID,origin)
```

Description

`datachunk = readChunk(sdsID,origin)` reads an entire chunk of data from the data set identified by `sdsID`. The `origin` input specifies the location of the chunk in zero-based chunking coordinates, not in data set coordinates.

This function corresponds to the `SDreadchunk` function in the HDF library C API, but because MATLAB uses FORTRAN-style ordering, the origin parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
dataChunk = sd.readChunk(sdsID,[0 1]);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.writeChunk` | `sd.writeData`

matlab.io.hdf4.sd.readData

Package: `matlab.io.hdf4.sd`

Read subsample of data

Syntax

```
data = readData(sdsID)
data = readData(sdsID,start,count)
data = readData(sdsID,start,count,stride)
```

Description

`data = readData(sdsID)` reads all of the data for the data set identified by `sdsID`.

`data = readData(sdsID,start,count)` reads a contiguous hyperslab of data from the data set identified by `sdsID`. The `start` input specifies the starting position from where the hyperslab is read. `count` specifies the number of values to read along each data set dimension.

`data = readData(sdsID,start,count,stride)` reads a strided hyperslab of data from the data set identified by `sdsID`.

`start`, `count`, and `stride` use zero-based indexing.

This function corresponds to the `SDreaddata` function in the HDF library C API, but because MATLAB uses FORTRAN-style ordering, the `start`, `count`, and `stride` parameters are reversed with respect to the C library API.

Examples

Read an entire data set.

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
```

```
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
data = sd.readData(sdsID);
sd.endAccess(sdsID);
sd.close(sdID);
```

Read a 2-by-3 portion of a data set.

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
data = sd.readData(sdsID,[0 0],[2 3]);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.writeData`

matlab.io.hdf4.sd.refToIndex

Package: `matlab.io.hdf4.sd`

Index of data set corresponding to reference number

Syntax

```
idx = refToIndex(sdID,ref)
```

Description

`idx = refToIndex(sdID,ref)` returns the index of the data set identified by its reference number `ref`. The `idx` output can then be passed to `sd.select`, to obtain a data set identifier.

This function corresponds to the `SDreftoindex` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf','read');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
ref = sd.idToRef(sdsID);
idx2 = sd.refToIndex(sdID,ref);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.idToRef` | `sd.select`

matlab.io.hdf4.sd.select

Package: `matlab.io.hdf4.sd`

Identifier of data set with specified index

Syntax

```
sdsID = select(sdID,IDX)
```

Description

`sdsID = select(sdID,IDX)` returns the identifier of the data set specified by its index.

This function corresponds to the `SDselect` function in the HDF C library.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf','read');
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.endAccess` | `sd.nametoIndex`

matlab.io.hdf4.sd.setAttr

Package: `matlab.io.hdf4.sd`

Write attribute value

Syntax

```
setAttr(objID,name,value)
```

Description

`setAttr(objID,name,value)` attaches an attribute to the object specified by `objID`. If `objID` is the SD interface identifier, then a global attribute is created. If a data identifier is specified, then the attribute is attached to the data set. If a dimension identifier is specified, then the attribute is attached to the dimension.

This function corresponds to the `SDsetattr` function in the HDF library C API.

Examples

Attach attributes to a file, a data set, and to a dimension.

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sd.setAttr(sdID,'creation_date',datestr(now));
sdsID = sd.create(sdID,'temperature','double',[10 20]);
sd.setAttr(sdsID,'long_name','Temperature in sunlight.');
dimID0 = sd.getDimID(sdsID,0);
sd.setAttr(dimID0,'long_name','latitude');
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.findAttr` | `sd.readAttr`

matlab.io.hdf4.sd.setCal

Package: `matlab.io.hdf4.sd`

Set data set calibration information

Syntax

```
setCal(sdsID,cal,calErr,offset,offsetErr,datatype)
```

Description

`setCal(sdsID,cal,calErr,offset,offsetErr,datatype)` sets the calibration information for a data set.

This function corresponds to the `SDsetcal` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[10 20]);
sd.setDataStrs(sdsID,'Temperature','degrees_kelvin','%.3f','spherical');
sd.setCal(sdsID,1,0,273,0,'double');
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.getCal`

matlab.io.hdf4.sd.setChunk

Package: `matlab.io.hdf4.sd`

Set chunk size and compression method of data set

Syntax

```
setChunk(sdsID,chunkSize,comptype,compparm)
```

Description

`setChunk(sdsID,chunkSize,comptype,compparm)` makes the data set specified by `sdsID` a chunked data set with chunk size given by `chunkSize` and compression specified by `comptype` and `compparm`. The `comptype` input can be one of the following values.

`'none'`	No compression
`'skphuff'`	Skipping Huffman compression
`'deflate'`	GZIP compression
`'rle'`	Run-length encoding

- If `comptype` is `'none'` or `'rle'`, then `compparm` need not be specified.
- If `comptype` is `'skphuff'`, then `compparm` is the skipping size.
- If `comptype` is `'deflate'`, then `compparm` is the deflate level, which must be between 0 and 9.

This function corresponds to the `SDsetchunk` function in the HDF library C API, but because MATLAB uses FORTRAN-style ordering, the `chunkSize` parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
```

```
sdsID = sd.create(sdID,'temperature','double',[200 100]);
sd.setChunk(sdsID,[20 10],'skphuff',16);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

sd.readChunk | sd.writeChunk

matlab.io.hdf4.sd.setCompress

Package: `matlab.io.hdf4.sd`

Set compression method of data set

Syntax

```
setCompress(sdsID,comptype,compparm)
```

Description

`setCompress(sdsID,comptype,compparm)` sets the compression scheme for the specified data set. The compression must be done before writing the data set. Specify `comptype` as one of these values.

`'none'`	No compression
`'skphuff'`	Skipping Huffman compression
`'deflate'`	GZIP compression
`'rle'`	Run-length encoding

- If `comptype` is `'none'` or `'rle'`, then `compparm` need not be specified.
- If `comptype` is `'skphuff'`, then `compparm` is the skipping size.
- If `comptype` is `'deflate'`, then `compparm` is the deflate level, which must be between 0 and 9.

This function corresponds to the `SDsetcompress` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[200 100]);
sd.setCompress(sdsID,'deflate',5);
```

```
data = rand(200,100);
sd.writeData(sdsID,[0 0],data);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

sd.setChunk

matlab.io.hdf4.sd.setDataStrs

Package: `matlab.io.hdf4.sd`

Set predefined attributes for data set

Syntax

```
setDataStrs(sdsID,label,unit,format,coordsys)
```

Description

`setDataStrs(sdsID,label,unit,format,coordsys)` sets the predefined attributes `'long_name'`, `'units'`, `'format'`, and `'coordsys'` for a data set.

This function corresponds to the `SDsetdatastrs` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[10 20]);
sd.setDataStrs(sdsID,'degrees_celsius','degrees_east','','geo');
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.getDataStrs` | `sd.setDimStrs`

matlab.io.hdf4.sd.setDimName

Package: `matlab.io.hdf4.sd`

Associate name with dimension

Syntax

`setDimName(dimID,dimname)`

Description

`setDimName(dimID,dimname)` sets the name of the dimension identified by `dimID` to `dimname`.

This function corresponds to the `SDsetdimname` function in the HDF library C API.

Examples

Create a 2D data set with dimensions `'lat'` and `'lon'`.

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[10 20]);
dimID = sd.getDimID(sdsID,0);
sd.setDimName(dimID,'lat');
dimID = sd.getDimID(sdsID,1);
sd.setDimName(dimID,'lon');
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.dimInfo`

matlab.io.hdf4.sd.setDimScale

Package: `matlab.io.hdf4.sd`

Set scale values for dimension

Syntax

```
setDimScale(dimID,scaledata)
```

Description

`setDimScale(dimID,scaledata)` sets the scale values for a dimension.

This function corresponds to the `SDsetdimscale` function in the HDF library C API.

Examples

Create a 2D data set with dimensions `'lat'` and `'lon'`.

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[10 20]);
dimID = sd.getDimID(sdsID,0);
sd.setDimName(dimID,'lat');
sd.setDimScale(dimID,0:10:90);
dimID = sd.getDimID(sdsID,1);
sd.setDimName(dimID,'lon');
sd.setDimScale(dimID, -180:18:179);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.getDimScale`

matlab.io.hdf4.sd.setDimStrs

Package: `matlab.io.hdf4.sd`

Set label, unit, and format attributes

Syntax

```
setDimStrs(dimID,label,unit,format)
```

Description

`setDimStrs(dimID,label,unit,format)` sets the label, unit, and format attributes for the dimension identified by `dimID`.

This function corresponds to the `SDsetdimstrs` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[10 20]);
dimID = sd.getDimID(sdsID,0);
sd.setDimName(dimID,'lat');
dimID = sd.getDimID(sdsID,1);
sd.setDimName(dimID,'lon');
sd.setDimStrs(dimID,'Degrees of Longitude','degrees_east','%.2f');
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.getDimStrs`

matlab.io.hdf4.sd.setExternalFile

Package: `matlab.io.hdf4.sd`

Store data in external file

Syntax

```
setExternalFile(sdsID,extfile,offset)
```

Description

`setExternalFile(sdsID,extfile,offset)` moves data values (not metadata) into the external data file `extfile` starting at the byte offset, `offset`.

Data can only be moved once for any given data set. The external file should be kept with the main file.

This function corresponds to the `SDsetexternalfile` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[10 20]);
sd.setExternalFile(sdsID,'myExternalFile.dat',0);
sd.writeData(sdsID,[0 0],rand(10,20));
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.create` | `sd.writeData`

matlab.io.hdf4.sd.setFillMode

Package: `matlab.io.hdf4.sd`

Set current fill mode of file

Syntax

```
prevmode = setFillMode(sdID,fillmode)
```

Description

`prevmode = setFillMode(sdID,fillmode)` returns the previous fill mode of a file and resets it to `fillmode`. This setting applies to all data sets contained in the file identified by `sdID`.

Possible values of `fillmode` are `'fill'`, and `'nofill'`. `'fill'` is the default mode and indicates that fill values will be written when the data set is created. `'nofill'` indicates that the fill values will not be written.

When a fixed-size data set is created, the first call to `sd.writeData` will fill the entire data set with the default or user-defined fill value if `fillmode` is `'fill'`. In data sets with an unlimited dimension, if a new write operation takes place along the unlimited dimension beyond the last location of the previous write operation, the array locations between these written areas will be initialized to the user-defined fill value, or the default fill value if a user-defined fill value has not been specified.

If it is certain that all data set values will be written before any read operation takes place, there is no need to write the fill values. Calling `sd.setFillMode` with `'nofill'` can improve performance in this case.

This function corresponds to the `SDsetfillmode` function in the HDF library C API.

Examples

Write two partial records. Write the first in `'nofill'` mode, and the second with `'fill'` mode.

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sd.setFillMode(sdID,'nofill');
sdsID = sd.create(sdID,'temperature','double',[10 10 0]);
sd.writeData(sdsID,[0 0 0], rand(5,5));
sd.setFillMode(sdID,'fill');
sd.setFillValue(sdsID,-999);
sd.writeData(sdsID,[0 0 1], rand(5,5));
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.getFillValue` | `sd.setFillValue`

matlab.io.hdf4.sd.setFillValue

Package: `matlab.io.hdf4.sd`

Set fill value for data set

Syntax

```
setFillValue(sdsID,fillValue)
```

Description

`setFillValue(sdsID,fillValue)` sets the fill value for a data set. The fill value must have the same data type as the data set.

This function corresponds to the `SDsetfillvalue` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[10 20]);
sd.setFillValue(sdsID,-999);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.getFillValue`

matlab.io.hdf4.sd.setNBitDataSet

Package: `matlab.io.hdf4.sd`

Specify nonstandard bit length for data set values

Syntax

```
setNBitDataSet(sdsID,startBit,bitlen,ext,fillone)
```

Description

`setNBitDataSet(sdsID,startBit,bitlen,ext,fillone)` specifies that the integer data set identified by `sdsID` contains data of a non-standard length defined by `startBit` and `bitlen`.

Any length between 1 and 32 bits can be specified. After `setNBitDataset` has been called for the data set array, any read or write operation will involve conversion between the new data length of the data set array and the data length of the read or write buffer.

Bit lengths of all data types are counted from the right of the bit field starting with 0. In a bit field containing the values 01111011, bits 2 and 7 are set to 0 and all the other bits are set to 1. The least significant bit is bit 0.

The `startBit` parameter specifies the left-most position of the variable-length bit field to be written. For example, in the bit field described in the preceding paragraph a `startBit` parameter set to 4 would correspond to the fourth bit value of 1 from the right.

The parameter `bitlen` specifies the number of bits of the variable-length bit field to be written. This number includes the starting bit and the count proceeds toward the right end of the bit field - toward the lower-bit numbers. For example, starting at bit 5 and writing 4 bits of the bit field described in the preceding paragraph would result in the bit field 1110 being written to the data set. This would correspond to a `startBit` value of 5 and a `bitlen` value of 4.

The parameter `ext` specifies whether to use the left-most bit of the variable-length bit field to sign-extend to the left-most bit of the data set data. For example, if 9-bit signed

integer data is extracted from bits 17-25 and the bit in position 25 is 1, then when the data is read back from disk, bits 26-31 will be set to 1. Otherwise bit 25 will be 0 and bits 26-31 will be set to 0. The `ext` parameter can be set to `true` (or 1) or `false` (or 0); specify `true` to sign-extend.

The parameter `fillone` specifies whether to fill the "background" bits with the value 1 or 0. This parameter is also set to either `true` (or 1) or `false` (or 0).

The "background" bits of a non-standard length data set are the bits that fall outside of the non-standard length bit field stored on disk. For example, if five bits of an unsigned 16-bit integer data set located in bits 5 to 9 are written to disk with the parameter `fillone` set to `true` (or 1), then when the data is reread into memory bits 0 to 4 and 10 to 15 would be set to 1. If the same 5-bit data was written with a `fillone` value of `false` (or 0), then bits 0 to 4 and 10 to 15 would be set to 0.

The operation on `fillone` is performed before the operation on `ext`. For example, using the `ext` example above, bits 0 to 16 and 26 to 31 will first be set to the background bit value, and then bits 26 to 31 will be set to 1 or 0 based on the value of the 25th bit.

This function corresponds to the `SDsetnbitdataset` in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','int32',[10 20]);
sd.setNBitDataSet(sdsID,6,4,0,0);
data = int32([1:200]);
data = reshape(data,10,20);
sd.writeData(sdsID,[0 0],data);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.setCompress`

matlab.io.hdf4.sd.setRange

Package: `matlab.io.hdf4.sd`

Set maximum and minimum range value for data set

Syntax

```
setRange(sdsID,maxval,minval)
```

Description

`setRange(sdsID,maxval,minval)` sets the maximum and minimum range values of the data set identified by `sdsID`. These values form the "valid_range" attribute for `sdsID`.

The actual maximum and minimum values of the data set are not computed. The "valid_range" attribute is for informational purposes only.

This function corresponds to the `SDsetrange` function in the HDF library C interface.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[10 20]);
sd.setDataStrs(sdsID,'Temperature','degrees_celsius','%.2f','');
sd.setRange(sdsID,1000,-273.15);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.getRange`

matlab.io.hdf4.sd.start

Package: `matlab.io.hdf4.sd`

Open HDF file and initialize SD interface

Syntax

```
sdID = start(filename)
sdID = start(filename,access)
```

Description

`sdID = start(filename)` opens the file `filename` in read-only mode. This routine must be called for each file before any other `sd` calls can be made on that file.

`sdID = start(filename,access)` opens the file `filename` with the access mode specified by `access`. This routine must be called before any other SD interface operations can be made on that file. Specify `access` as one these values:

- `'read'`
- `'write'`
- `'create'`

`access` defaults to `'read'` if not supplied.

This function corresponds to the `SDstart` function in the HDF library C API.

Examples

```
import matlab.io.hdf4.*
sdID = sd.start('sd.hdf');
sd.close(sdID);
```

See Also

`sd.close`

matlab.io.hdf4.sd.writeChunk

Package: `matlab.io.hdf4.sd`

Write chunk to data set

Syntax

`writeChunk(sdsID,origin,dataChunk)`

Description

`writeChunk(sdsID,origin,dataChunk)` writes an entire chunk of data to the data set identified by `sdsID`. The `origin` input specifies the location of the chunk in chunking coordinates, not in data set coordinates.

This function corresponds to the `SDwritechunk` function in the HDF library C API, but because MATLAB uses FORTRAN-style ordering, the origin parameter is reversed with respect to the C library API.

Examples

Write to a 2D chunked and compressed data set. The chunked layout constitutes a 10-by-5 grid.

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[100 50]);
sd.setChunk(sdsID,[10 10],'deflate',5);
for j = 0:9
    for k = 0:4
        origin = [j k];
        data = (1:100) + k*1000 + j*10000;
        data = reshape(data,10,10);
        sd.writeChunk(sdsID,origin,data);
    end
```

```
end
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

sd.readChunk | sd.writeData

matlab.io.hdf4.sd.writeData

Package: `matlab.io.hdf4.sd`

Write to data set

Syntax

```
writeData(sdsID,data)
writeData(sdsID,start,data)
writeData(sdsID,start,stride,data)
```

Description

`writeData(sdsID,data)` writes all the data to the data set identified by `sdsID`.

`writeData(sdsID,start,data)` writes a contiguous hyperslab to the data set. `start` specifies the zero-based starting index. The number of values along each dimension is inferred from the size of `data`.

`writeData(sdsID,start,stride,data)` writes a strided hyperslab of data to a grid datafield. The number of elements to write along each dimension is inferred either from the size of `data` or from the data set itself.

`start` and `stride` use zero-based indexing.

This function corresponds to the `SDreadchunk` function in the HDF library C API, but because MATLAB uses FORTRAN-style ordering, the `start` and `stride` parameters are reversed with respect to the C library API.

Examples

Write to a 2D data set.

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
```

```
sdsID = sd.create(sdID,'temperature','double',[10 20]);
data = rand(10,20);
sd.writeData(sdsID,[0 0],data);
sd.endAccess(sdsID);
sd.close(sdID);
```

Write to a 2D unlimited data set.

```
import matlab.io.hdf4.*
sdID = sd.start('myfile.hdf','create');
sdsID = sd.create(sdID,'temperature','double',[10 0]);
data = rand(10,20);
sd.writeData(sdsID,[0 0],data);
data = rand(10,30);
sd.writeData(sdsID,[0 20],data);
sd.endAccess(sdsID);
sd.close(sdID);
```

See Also

`sd.readData`

matlab.io.hdfeos.gd

Low-level access to HDF-EOS grid data

Description

To use these MATLAB functions, you must be familiar with the HDF-EOS library C interface. In most cases, the syntax of the MATLAB function is similar to the syntax of the corresponding HDF-EOS library function. The functions are implemented as the package `matlab.io.hdfeos.gd`. To use this package, prefix the function name with a package path, or use the import function to add the package to the current import list, prior to calling the function, for example,

```
import matlab.io.hdfeos.*
gfid = gd.open(filename,'read');
```

Access

matlab.io.hdfeos.gd.attach	Attach to existing grid
matlab.io.hdfeos.gd.close	Close HDF-EOS grid file
matlab.io.hdfeos.gd.detach	Detach from existing grid
matlab.io.hdfeos.gd.open	Open grid file

Definition

matlab.io.hdfeos.gd.create	Create new grid structure
matlab.io.hdfeos.gd.defComp	Set grid field compression
matlab.io.hdfeos.gd.defDim	Define new dimension within grid
matlab.io.hdfeos.gd.defField	Define new data field within grid
matlab.io.hdfeos.gd.defOrigin	Define origin of pixels in grid
matlab.io.hdfeos.gd.defPixReg	Define pixel registration within grid
matlab.io.hdfeos.gd.defProj	Define grid projection
matlab.io.hdfeos.gd.writeBlkSomOffset	Write Block SOM offset

Basic I/O

matlab.io.hdfeos.gd.getFillValue	Fill value for specified field
matlab.io.hdfeos.gd.readAttr	Read grid attribute
matlab.io.hdfeos.gd.readField	Read data from grid field
matlab.io.hdfeos.gd.setFillValue	Set fill value for specified field
matlab.io.hdfeos.gd.writeAttr	Write grid attribute
matlab.io.hdfeos.gd.writeField	Write data to grid field

Inquiry

matlab.io.hdfeos.gd.compInfo	Compression information for field
matlab.io.hdfeos.gd.dimInfo	Length of dimension
matlab.io.hdfeos.gd.fieldInfo	Information about data field
matlab.io.hdfeos.gd.gridInfo	Position and size of grid
matlab.io.hdfeos.gd.inqAttrs	Names of grid attributes
matlab.io.hdfeos.gd.inqDims	Information about dimensions defined in grid
matlab.io.hdfeos.gd.inqFields	Information about data fields defined in grid
matlab.io.hdfeos.gd.inqGrid	Names of grids in file
matlab.io.hdfeos.gd.nEntries	Number of specified objects
matlab.io.hdfeos.gd.originInfo	Origin code
matlab.io.hdfeos.gd.pixRegInfo	Pixel registration code
matlab.io.hdfeos.gd.projInfo	GCTP projection information about grid
matlab.io.hdfeos.gd.readBlkSomOffset	Read Block SOM offset

Subsetting

matlab.io.hdfeos.gd.defBoxRegion	Define region of interest by latitude and longitude
matlab.io.hdfeos.gd.defVrtRegion	Define vertical subset region
matlab.io.hdfeos.gd.extractRegion	Read region of interest from field
matlab.io.hdfeos.gd.getPixels	Pixel rows and columns for latitude/longitude pairs
matlab.io.hdfeos.gd.getPixValues	Read data values for specified pixels
matlab.io.hdfeos.gd.interpolate	Bilinear interpolation on a grid field
matlab.io.hdfeos.gd.regionInfo	Information about subsetted region

Tiling

matlab.io.hdfeos.gd.defTile	Define tiling parameters
matlab.io.hdfeos.gd.readTile	Read single tile of data from field
matlab.io.hdfeos.gd.setTileComp	Set tiling and compression for field with fill value
matlab.io.hdfeos.gd.tileInfo	Tile size of grid field
matlab.io.hdfeos.gd.writeTile	Write tile to field

Utility

matlab.io.hdfeos.gd.ij2ll	Convert row and column space to latitude and longitude
matlab.io.hdfeos.gd.ll2ij	Convert latitude and longitude to row and column space
matlab.io.hdfeos.gd.sphereCodeToName	Name corresponding to GCTP sphere code
matlab.io.hdfeos.gd.sphereNameToCode	Numeric GCTP code corresponding to sphere name

matlab.io.hdfeos.gd.attach

Package: `matlab.io.hdfeos.gd`

Attach to existing grid

Syntax

```
gridID = attach(gfID,gridName)
```

Description

`gridID = attach(gfID,gridName)` attaches to the grid dataset identified by `gridName` in the file identified by `gfID`. The `gridID` output is the identifier for the grid dataset.

This function corresponds to the `GDattach` function in the HDF-EOS library C API.

Examples

Attach to the grid named `'PolarGrid'` in the file `'grid.hdf'`.

```
import matlab.io.hdfeos.*
gfID = gd.open('grid.hdf');
gridID = gd.attach(gfID,'PolarGrid');
gd.detach(gridID);
gd.close(gfID);
```

See Also

`gd.detach` | `gd.inqGrid` | `gd.readField`

matlab.io.hdfeos.gd.close

Package: `matlab.io.hdfeos.gd`

Close HDF-EOS grid file

Syntax

```
close(gfID)
```

Description

`close(gfID)` closes an HDF-EOS grid file identified by `gfID`.

This function corresponds to the `GDclose` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfID = gd.open('grid.hdf');
gd.close(gfID);
```

See Also

`gd.create` | `gd.open`

matlab.io.hdfeos.gd.compInfo

Package: `matlab.io.hdfeos.gd`

Compression information for field

Syntax

```
[compCode,parms] = compInfo(gridID,fieldname)
```

Description

`[compCode,parms] = compInfo(gridID,fieldname)` returns the compression code and compression parameters for a given field. Refer to `gd.defComp` for a description of various compression schemes and parameters.

This function corresponds to the `GDcompinfo` function in the HDF-EOS library C API.

Examples

Get compression information for the `ice_temp` field.

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
[compCode,compParms] = gd.compInfo(gridID,'ice_temp');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defComp`

matlab.io.hdfeos.gd.create

Package: `matlab.io.hdfeos.gd`

Create new grid structure

Syntax

```
gridID = create(gfID,gridName,xdim,ydim,upLeft,lowRight)
```

Description

`gridID = create(gfID,gridName,xdim,ydim,upLeft,lowRight)` creates a new grid structure where `gfID` is the grid file identifier. `gridName` is the name of the new grid. `xdim` and `ydim` define the size of the grid. `upLeft` is a two-element vector containing the location of the upper left pixel, and `lowRight` is a two-element vector containing the location of the lower right pixel.

Note `upLeft` and `lowRight` are in units of meters for all GCTP projections other than the geographic and bcea projections, which should have units of packed degrees.

Note For certain projections, `upLeft` and `lowRight` can be given as `[]`.

- Polar Stereographic projection of an entire hemisphere.
- Goode Homolosine projection of the entire globe.
- Lambert Azimuthal entire polar or equatorial projection.

Note MATLAB uses Fortran-style ordering, but the HDF-EOS library uses C-style ordering.

This function corresponds to the `GDcreate` function in the HDF-EOS library C API.

Examples

Create a polar stereographic grid of the northern hemisphere.

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
gridID = gd.create(gfid,'PolarGrid',100,100,[],[]);
gd.detach(gridID);
gd.close(gfid);
```

Create a UTM grid bounded by 54 E to 60 E longitude and 20 N to 30 N latitude. Divide the grid into 120 bins along the x-axis and 200 bins along the y-axis.

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
uplft = [210584.50041 3322395.95445];
lowrgt = [813931.10959 2214162.53278];
gridID = gd.create(gfid,'UTMGrid',120,200,uplft,lowrgt);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defProj` | `gd.detach` | `gd.gridInfo`

matlab.io.hdfeos.gd.defBoxRegion

Package: `matlab.io.hdfeos.gd`

Define region of interest by latitude and longitude

Syntax

```
regionID = defBoxRegion(gridID,cornerLat,cornerLon)
```

Description

`regionID = defBoxRegion(gridID,cornerLat,cornerLon)` defines a latitude-longitude box region as a subset region for a grid. `regionID` can be used to read all the entries of a data field within the region.

This function corresponds to the `GDdefboxregion` function in the HDF-EOS library C API.

Examples

Define a region of interest between 20 and 50 degrees latitude and between -90 and -60 degrees longitude.

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf','read');
gridID = gd.attach(gfid,'PolarGrid');
cornerlat = [20 50];
cornerlon = [-90 -60];
regionID = gd.defBoxRegion(gridID,cornerlat,cornerlon);
data = gd.extractRegion(gridID,regionID,'ice_temp');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.extractRegion`

matlab.io.hdfeos.gd.defComp

Package: `matlab.io.hdfeos.gd`

Set grid field compression

Syntax

```
defComp(gridID,compscheme,compparm)
```

Description

`defComp(gridID,compscheme,compparm)` sets the HDF field compression for subsequent field definitions. The compression scheme does not apply to one-dimensional fields. `compscheme` can be one of the following values.

`'rle'`	Run-length encoding
`'skphuff'`	Skipping Huffman
`'deflate'`	Gzip deflate
`'none'`	No compression

When the compression scheme is `'deflate'`, `compparm` is the deflate compression level, an integer between 0 and 9. `compparm` can be omitted for the other compression schemes.

If a field is defined with compression, it must be written with a single call to `gd.writeField`. If this is not possible, you should consider using tiling.

This function corresponds to the `GDdefcomp` function in the HDF-EOS library C API.

Examples

Create a grid with a polar stereographic `Pressure` field using run-length encoding, and then an `Opacity` field with deflate compression.

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
gridID = gd.create(gfid,'PolarGrid',100,100,[],[]);
projparm = zeros(1,13);
projparm(6) = 90000000;
gd.defProj(gridID,'ps',[],'WGS 84',projparm);
dims = { 'XDim', 'YDim' };
gd.defComp(gridID,'rle');
gd.defField(gridID,'Pressure',dims,'float');
gd.defComp(gridID,'deflate',5);
gd.defField(gridID,'Opacity',dims,'float');
gd.detach(gridID);
gd.close(gfid);
```

See Also

gd.defField | gd.defTile

matlab.io.hdfeos.gd.defDim

Package: `matlab.io.hdfeos.gd`

Define new dimension within grid

Syntax

`defDim(gridID,dimname,dimlen)`

Description

`defDim(gridID,dimname,dimlen)` defines a new dimension named `dimname` with length `dimlen` in the grid structure identified by `gridID`.

To specify an unlimited dimension, you can use either 0 or `'unlimited'` for `dimlen`.

This function corresponds to the `GDdefdim` function in the HDF-EOS library C API.

Examples

Define a dimension `'Band'` with length of 15 and an unlimited dimension `'Time'`.

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
gridID = gd.create(gfid,'PolarGrid',100,100,[],[]);
gd.defDim(gridID,'Band',15);
gd.defDim(gridID,'Time',0);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defField` | `gd.dimInfo`

matlab.io.hdfeos.gd.defField

Package: `matlab.io.hdfeos.gd`

Define new data field within grid

Syntax

```
defField(gridID,fieldname,dimlist,dtype)
defField(gridID,fieldname,dimlist,dtype,mergeCode)
```

Description

`defField(gridID,fieldname,dimlist,dtype)` defines data fields for a grid specified by `gridID`. The `fieldname` input is the name of the new field. `dimlist` is a cell array of geolocation dimensions and should be listed in FORTRAN-style order, that is, the fastest varying dimension should be listed first. Specify `dimlist` as a cell array of character vectors or a string array containing text which describes the dimensions or just a character vector or string scalar when there is only one dimension. `dtype` is the data type of the field.

`defField(gridID,fieldname,dimlist,dtype,mergeCode)` defines a data field with a specific merge code. `mergeCode` can be either `'nomerge'` or `'automerge'`. The `mergeCode` input defaults to `'nomerge'` if not provided.

This function corresponds to the `GDdeffield` function in the HDF library C API, but because MATLAB uses FORTRAN-style ordering, the `dimlist` parameter is reversed with respect to the C library API.

Examples

Define a single precision grid field `'Temperature'` with dimensions `'XDim'` and `'YDim'`. Then define a single precision field `'Spectra'` with dimensions `'XDim'`, `'YDim'`, and `'Bands'`.

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
xdim = 120; ydim = 200;
gridID = gd.create(gfid,'geo',xdim,ydim,[],[]);
gd.defProj(gridID,'geo',[],[],[]);
dimlist = {'XDim','YDim'};
gd.defField(gridID,'Temperature',dimlist,'single');
gd.defDim(gridID,'Bands',3);
dimlist = {'XDim','YDim','Bands'};
gd.defField(gridID,'Spectra',dimlist,'uint8');
gd.detach(gridID);
gd.close(gfid);
```

See Also

gd.create | gd.defDim

matlab.io.hdfeos.gd.defOrigin

Package: `matlab.io.hdfeos.gd`

Define origin of pixels in grid

Syntax

`defOrigin(gridID,originCode)`

Description

`defOrigin(gridID,originCode)` defines the origin of pixels in a grid. `gridID` is the identifier of the grid, and `originCode` can be one of the following four values.

`'ul'`	Upper-left
`'ur'`	Upper-right
`'ll'`	Lower-left
`'lr'`	Lower-right

You can select any corner of the grid pixel as the origin. If this routine is not invoked, the grid defaults to using the upper-left corner for the origin.

This function corresponds to the `GDdeforigin` function in the HDF-EOS library C API.

Examples

Create a polar stereographic grid with the origin of the grid pixel in the lower right corner.

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
gridID = gd.create(gfid,'PolarGrid',100,100,[],[]);
projparm = zeros(1,13);
projparm(6) = 90000000;
```

```
gd.defProj(gridID,'ps',[],'WGS 84',projparm);
gd.defOrigin(gridID,'lr');
gd.detach(gridID);
gd.close(gfid);
```

See Also

gd.defPixReg | gd.originInfo

matlab.io.hdfeos.gd.defPixReg

Package: `matlab.io.hdfeos.gd`

Define pixel registration within grid

Syntax

```
defPixReg(gridID,pixRegCode)
```

Description

`defPixReg(gridID,pixRegCode)` defines whether the pixel center or pixel corner is used when requesting the location (longitude and latitude) of a given pixel. `pixRegCode` can be one of the following values.

`'center'`	Center of pixel cell
`'corner'`	Corner of pixel cell

If this routine is not invoked, the pixel registration is `'center'`.

This function corresponds to the `GDdefpixreg` function in the HDF-EOS library.

Examples

Define a grid with pixel registration in the center.

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
gridID = gd.create(gfid,'PolarGrid',100,100,[],[]);
projparm = zeros(1,13);
projparm(6) = 90000000;
gd.defProj(gridID,'ps',[],'WGS 84',projparm);
gd.defPixReg(gridID,'corner');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defOrigin` | `gd.pixRegInfo`

matlab.io.hdfeos.gd.defProj

Package: `matlab.io.hdfeos.gd`

Define grid projection

Syntax

```
defProj(gridID,projCode,zoneCode,sphereCode,projParm)
```

Description

`defProj(gridID,projCode,zoneCode,sphereCode,projParm)` defines a GCTP projection on the grid specified by `gridID`. The `projCode` argument can be one of these values.

`'geo'`	Geographic
`'utm'`	Universal Transverse Mercator
`'albers'`	Albers Conical Equal Area
`'lamcc'`	Lambert Conformal Conic
`'ps'`	Polar Stereographic
`'polyc'`	Polyconic
`'tm'`	Transverse Mercator
`'lamaz'`	Lambert Azimuthal Equal Area
`'snsoid'`	Sinusoidal
`'hom'`	Hotine Oblique Mercator
`'som'`	Space Oblique Mercator
`'good'`	Interrupted Goode Homolosine
`'cea'`	Cylindrical Equal Area
`'bcea'`	Behrmann Cylindrical Equal Area
`'isinus'`	Integerized Sinusoidal

If `projCode` is `'geo'`, then `zoneCode`, `sphereCode`, and `projParm` should be specified as `[]`. Any other values for these parameters are ignored.

`zoneCode` is the Universal Transverse Mercator zone code. It should be specified as -1 for other projections.

`sphereCode` is the name of the GCTP spheroid or the corresponding numeric code.

`projParm` is a vector of up to 13 elements containing projection-specific parameters. For more details about `projCode`, `zoneCode`, `sphereCode`, and `projParm`, see Chapter 6 of HDF-EOS Library Users Guide for the ECS Project, Volume 1: Overview and Examples.

This function corresponds to the `GDdefproj` function in the HDF library C API.

Examples

Create a UTM grid bounded by 54 E to 60 E longitude and 20 N to 30 N latitude (zone 40). Divide the grid into 120 bins along the x-axis and 200 bins along the y-axis.

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
uplft = [210584.50041 3322395.95445];
lowrgt = [813931.10959 2214162.53278];
gridID = gd.create(gfid,'UTMGrid',120,200,uplft,lowrgt);
gd.defProj(gridID,'utm',40,'Clarke 1866',[]);
gd.detach(gridID);
gd.close(gfid);
```

Add a polar stereographic projection of the northern hemisphere with true scale at 90 N, 0 longitude below the pole using the WGS 84 spheroid.

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
gridID = gd.create(gfid,'PolarGrid',100,100,[],[]);
projparm = zeros(1,13);
projparm(6) = 90000000;
gd.defProj(gridID,'ps',[],'WGS 84',projparm);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.create` | `gd.projInfo` | `gd.sphereCodeToName`

matlab.io.hdfeos.gd.defTile

Package: `matlab.io.hdfeos.gd`

Define tiling parameters

Syntax

```
defTile(gridID,tileDims)
```

Description

`defTile(gridID,tileDims)` defines tiling dimensions for subsequent field definitions. If `tileDims` is `[]`, then subsequently defined fields will have no tiling.

This function corresponds to the `GDdeftile` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `tileDims` parameter is reversed with respect to the C library API.

Examples

Define a field with tiling, then a subsequent field with no tiling.

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
gridID = gd.create(gfid,'GeoGrid',120,200,[],[]);
gd.defDim(gridID,'Bands',3);
gd.defProj(gridID,'geo',[],[],[]);
gd.defTile(gridID,[30 50 1]);
dimlist = {'XDim','YDim','Bands'};
gd.defField(gridID,'Spectra',dimlist,'float');
gd.defTile(gridID,[]);
dimlist = {'XDim','YDim'};
gd.defField(gridID,'Temperature',dimlist,'int32');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defField` | `gd.tileInfo`

matlab.io.hdfeos.gd.defVrtRegion

Package: `matlab.io.hdfeos.gd`

Define vertical subset region

Syntax

```
out_RID = defVrtRegion(gridID,regionID,vobj,vRange)
```

Description

`out_RID = defVrtRegion(gridID,regionID,vobj,vRange)` defines a vertical subset region and can be used on either a monotonic field or contiguous elements of a dimension.

`regionID` should be `'noprevsub'` if no prior subsetting has occurred. Otherwise it should be a value as returned from a previous subsetting routine.

`vobj` is the name of either the dimension or field to subset. If `vobj` is a dimension, it should be prefixed with `'DIM:'`.

`vRange` is the minimum and maximum range for the vertical subset.

This function corresponds to the `GDdefvrtregion` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
range = [333 667];
regionID = gd.defVrtRegion(gridID,'noprevsub','Height',range);
data = gd.extractRegion(gridID,regionID,'pressure');
gd.detach(gridID);
gd.close(gfid);
```

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
range = [3 5];
regionID = gd.defVrtRegion(gridID,'noprevsub','DIM:Height',range);
data = gd.extractRegion(gridID,regionID,'pressure');
gd.detach(gridID);
gd.close(gfid);
```

See Also

gd.extractRegion

matlab.io.hdfeos.gd.detach

Package: `matlab.io.hdfeos.gd`

Detach from existing grid

Syntax

```
detach(gridID)
```

Description

`detach(gridID)` detaches from the grid identified by `gridID`.

This function corresponds to the `GDdetach` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfID = gd.open('grid.hdf');
gridID = gd.attach(gfID,'PolarGrid');
gd.detach(gridID);
gd.close(gfID);
```

See Also

`gd.attach`

matlab.io.hdfeos.gd.dimInfo

Package: `matlab.io.hdfeos.gd`

Length of dimension

Syntax

```
dimlen = diminfo(gridID,dimname)
```

Description

`dimlen = diminfo(gridID,dimname)` retrieves the length of the specified user-defined dimension.

Please note that the two extents used to create the grid are not considered user-defined dimensions. To retrieve the length of `XDim` and `YDim`, use `gd.gridInfo`. This function corresponds to the `GDdiminfo` function in the HDF-EOS library C API.

Examples

Inquire about a `'Bands'` dimension.

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
dimlen = gd.dimInfo(gridID,'Height');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defDim` | `gd.gridInfo`

matlab.io.hdfeos.gd.extractRegion

Package: `matlab.io.hdfeos.gd`

Read region of interest from field

Syntax

```
data = extractRegion(gridID,regionID,fieldname)
```

Description

`data = extractRegion(gridID,regionID,fieldname)` extract data from a subsetted region.

This routine corresponds to the `GDextractregion` function in the HDF-EOS library C API.

Examples

Define and extract a region of interest between 20 and 50 degrees latitude and between -90 and -60 degrees longitude.

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf','read');
gridID = gd.attach(gfid,'PolarGrid');
cornerlat = [20 50];
cornerlon = [-90 -60];
regionID = gd.defBoxRegion(gridID,cornerlat,cornerlon);
data = gd.extractRegion(gridID,regionID,'ice_temp');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defBoxRegion` | `gd.defVrtRegion`

matlab.io.hdfeos.gd.fieldInfo

Package: `matlab.io.hdfeos.gd`

Information about data field

Syntax

```
[dims,ntype,dimlist] = fieldInfo(gridID,fieldname)
```

Description

`[dims,ntype,dimlist] = fieldInfo(gridID,fieldname)` returns information about a specific geolocation or data field in the grid. `dims` is a vector containing the dimension sizes of the field. `ntype` is a character vector or string scalar containing the HDF number type of the field. `dimlist` is a cell array of character vectors or a string array containing the dimension names.

This function corresponds to the `GDfieldinfo` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `dimlist` parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf','read');
gridID = gd.attach(gfid,'PolarGrid');
[dims,ntype,dimlist] = gd.fieldInfo(gridID,'ice_temp');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defField`

matlab.io.hdfeos.gd.getFillValue

Package: `matlab.io.hdfeos.gd`

Fill value for specified field

Syntax

```
fillvalue = getFillValue(gridID,fieldname)
```

Description

`fillvalue = getFillValue(gridID,fieldname)` retrieves the fill value for the specified field.

This function corresponds to the `GDgetfillvalue` function in the HDF-EOS library C API.

Examples

Return the fill value for the `'ice_temp'` field in the `'PolarGrid'` grid.

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
fillvalue = gd.getFillValue(gridID,'ice_temp');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.setFillValue`

matlab.io.hdfeos.gd.getPixels

Package: `matlab.io.hdfeos.gd`

Pixel rows and columns for latitude/longitude pairs

Syntax

```
[row,col] = getPixels(gridID,lat,lon)
```

Description

`[row,col] = getPixels(gridID,lat,lon)` converts latitude/longitude pairs into zero-based pixel row and column coordinates. The origin is the upper left-hand corner of the grid pixel. If the latitude/longitude pairs are outside the grid, then `row` and `col` are -1.

This function corresponds to the `GDgetpixels` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf','read');
gridID = gd.attach(gfid,'PolarGrid');
cornerlat = [20 50];
cornerlon = [-90 -60];
[row,col] = gd.getPixels(gridID,cornerlat,cornerlon);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.getPixValues`

matlab.io.hdfeos.gd.getPixValues

Package: `matlab.io.hdfeos.gd`

Read data values for specified pixels

Syntax

```
data = getPixValues(gridID,rows,cols,fieldname)
```

Description

`data = getPixValues(gridID,rows,cols,fieldname)` reads data values for the pixels specified by the zero-based `rows` and `cols` coordinates. All entries along the non-geographic dimensions, i.e. NOT `XDim` and `YDim`, are returned.

This function corresponds to the `GDgetpixvalues` function in the HDF-EOS library C API.

Examples

Read the grid field's corner values.

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf','read');
gridID = gd.attach(gfid,'PolarGrid');
rows = [0 99 99  0];
cols = [0  0 99 99];
data = gd.getPixValues(gridID,rows,cols,'ice_temp');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defBoxRegion` | `gd.extractRegion` | `gd.getPixels` | `gd.readField`

matlab.io.hdfeos.gd.gridInfo

Package: `matlab.io.hdfeos.gd`

Position and size of grid

Syntax

```
[xDim,yDim,upLeft,lowRight] = gridInfo(gridID)
```

Description

`[xDim,yDim,upLeft,lowRight] = gridInfo(gridID)` returns the size of a grid as well as the upper left and lower right corners of the grid.

Note `upLeft` and `lowRight` are in units of meters for all GCTP projections other than the geographic and bcea projections, which will have units of packed degrees.

This function corresponds to the `GDgridinfo` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
[xdimsize,ydimsize,upleft,lowright] = gd.gridInfo(gridID);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.create`

matlab.io.hdfeos.gd.ij2ll

Package: `matlab.io.hdfeos.gd`

Convert row and column space to latitude and longitude

Syntax

```
[lat,lon] = ij2ll(gridID,row,col)
```

Description

`[lat,lon] = ij2ll(gridID,row,col)` converts a grid's row and column coordinates to latitude and longitude in decimal degrees.

`row` and `col` are zero-based and defined such that `col` increases monotonically with the `XDim` dimension and `row` increases monotonically with the `YDim` dimension in the HD-EOS library.

This routine corresponds to the `GDij2ll` function in the HDF-EOS C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
[xdim,ydim] = gd.gridInfo(gridID);
r = 0:(xdim-1);
c = 0:(ydim-1);
[Col,Row] = meshgrid(c,r);
[lat,lon] = gd.ij2ll(gridID,Row,Col);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.ll2ij` | `gd.readField`

matlab.io.hdfeos.gd.inqAttrs

Package: `matlab.io.hdfeos.gd`

Names of grid attributes

Syntax

```
attrList = inqAttrs(gridID)
```

Description

`attrList = inqAttrs(gridID)` returns the list of grid attribute names. `attrList` is a cell array.

This function corresponds to the `GDinqattrs` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
attrList = gd.inqAttrs(gridID);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.readAttr` | `gd.writeAttr`

matlab.io.hdfeos.gd.inqDims

Package: `matlab.io.hdfeos.gd`

Information about dimensions defined in grid

Syntax

```
[dimnames,dimlens] = inqDims(gridID)
```

Description

`[dimnames,dimlens] = inqDims(gridID)` returns the names of the dimensions `dimnames` in a cell array and their respective lengths `dimlens`. This does not include the grid extent dimensions `XDim` and `YDim`.

This function corresponds to the `GDinqdims` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `dimnames` and `dimlens` parameters are reversed with respect to the C library API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf','read');
gridID = gd.attach(gfid,'PolarGrid');
[dims,dimlens] = gd.inqDims(gridID);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defDim`

matlab.io.hdfeos.gd.inqFields

Package: `matlab.io.hdfeos.gd`

Information about data fields defined in grid

Syntax

```
[fldList,fldRank,fldType] = inqFields(gridID)
```

Description

`[fldList,fldRank,fldType] = inqFields(gridID)` returns the list of fields `fldList` as a cell array. `fldRank` contains the rank of each data field. `fldType` is a cell array containing the data type of each data field.

This function corresponds to the `GDinqfields` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
[fldlist,fldrank,fldtype] = gd.inqFields(gridID);
gd.detach(gridID);
gd.close(gfid);
for j = 1:numel(fldrank)
    fprintf('%s: Rank %d, datatype %s\n', fldlist{j},fldrank(j),fldtype{j});
end
```

See Also

`gd.defField`

matlab.io.hdfeos.gd.inqGrid

Package: `matlab.io.hdfeos.gd`

Names of grids in file

Syntax

```
grids = inqGrid(filename)
```

Description

`grids = inqGrid(filename)` returns the names of all grids in the given file. `grids` is a cell array.

This function corresponds to the `GDinqgrid` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
grids = gd.inqGrid('grid.hdf');
```

See Also

`gd.create` | `sw.inqSwath`

matlab.io.hdfeos.gd.interpolate

Package: `matlab.io.hdfeos.gd`

Bilinear interpolation on a grid field

Syntax

```
data = interpolate(gridID,lat,lon,fieldname)
```

Description

`data = interpolate(gridID,lat,lon,fieldname)` performs bilinear interpolation on lat/lon pairs from the data in the grid field.

`data` contains the interpolated field values.

This function corresponds to the `GDinterpolate` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('example.hdf');
gridID = gd.attach(gfid,'MonthlyRain');
[lat,lon] = gd.ij2ll(gridID,[36 36],[14 15]);
data = gd.interpolate(gridID,lat,lon,'TbOceanRain');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.ij2ll`

matlab.io.hdfeos.gd.ll2ij

Package: `matlab.io.hdfeos.gd`

Convert latitude and longitude to row and column space

Syntax

```
[row,col] = ll2ij(gridID,lat,lon)
```

Description

`[row,col] = ll2ij(gridID,lat,lon)` converts latitude and longitude coordinates to a pre-defined grid's row and column coordinates.

`row` and `col` are zero-based and defined such that `col` increases monotonically with the `XDim` dimension and `row` increases monotonically with the `YDim` dimension in the HD-EOS library.

This routine corresponds to the `GDll2ij` function in the HDF-EOS C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
lat = [46 46 42 42];
lon = [-71 -67 -67 -71];
[row,col] = gd.ll2ij(gridID,lat,lon);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.ij2ll`

matlab.io.hdfeos.gd.nEntries

Package: `matlab.io.hdfeos.gd`

Number of specified objects

Syntax

```
nentries = nEntries(gridID,entType)
```

Description

`nentries = nEntries(gridID,entType)` returns the number of specified objects in a grid. `entType` can be either `'dims'` or `'fields'`.

This function corresponds to the `GDnentries` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
fid = gd.open('grid.hdf');
gridID = gd.attach(fid,'PolarGrid');
ndims = gd.nEntries(gridID,'dims');
nflds = gd.nEntries(gridID,'fields');
gd.detach(gridID);
gd.close(fid);
fprintf('The number of dimensions is %d.\n', ndims);
fprintf('The number of fields is %d.\n', nflds);
```

See Also

`gd.inqGrid`

matlab.io.hdfeos.gd.open

Package: `matlab.io.hdfeos.gd`

Open grid file

Syntax

```
gfid = open(filename,access)
```

Description

`gfid = open(filename,access)` opens or creates an HDF-EOS grid file identified by `filename` and returns a file ID. `access` can be one of the following values:

`'read'`	Read-only
`'rdwr'`	Read-write
`'create'`	Creates a file, deleting it if it already exists

If `access` is not provided, it defaults to `'read'`.

This function corresponds to the `GDopen` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gd.close(gfid);
```

See Also

`gd.attach` | `gd.close`

matlab.io.hdfeos.gd.originInfo

Package: `matlab.io.hdfeos.gd`

Origin code

Syntax

```
originCode = originInfo(gridID)
```

Description

`originCode = originInfo(gridID)` retrieves the origin code for the grid specified by `gridID`. The `originCode` output is one of the following four values.

`'ul'`	Upper-left
`'ur'`	Upper-right
`'ll'`	Lower-left
`'lr'`	Lower-right

This function corresponds to the `GDorigininfo` routine in the HDF-EOS library.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
origin = gd.originInfo(gridID);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defOrigin`

matlab.io.hdfeos.gd.pixRegInfo

Package: `matlab.io.hdfeos.gd`

Pixel registration code

Syntax

```
pixRegCode = pixRegInfo(gridID)
```

Description

`pixRegCode = pixRegInfo(gridID)` retrieve the pixel registration code for the grid identified by `gridID`. The `pixRegCode` output can be one of the following values.

`'center'`	Center of pixel cell
`'corner'`	Corner of pixel cell

This function corresponds to the `GDpixreginfo` routine in the HDF-EOS library.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
code = gd.pixRegInfo(gridID);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defPixReg`

matlab.io.hdfeos.gd.projInfo

Package: `matlab.io.hdfeos.gd`

GCTP projection information about grid

Syntax

```
[projCode,zoneCode,sphereName,projParm] = projInfo(gridID)
```

Description

`[projCode,zoneCode,sphereName,projParm] = projInfo(gridID)` returns the GCTP projection code, zone code, spheroid, and projection parameters for the grid identified by `gridID`.

`zoneCode` is -1 if `projCode` is anything other than `'UTM'`.

This function corresponds to the `GDprojinfo` function in the HDF-EOS library C API.

For details about the GCTP projection code, zone code, spheroid code, and projection parameters, please consult the HDF-EOS User's Guide.

Examples

```
import matlab.io.hdfeos.*
fid = gd.open('grid.hdf');
gridID = gd.attach(fid,'PolarGrid');
[projCode,zoneCode,sphereCode,projParm] = gd.projInfo(gridID);
gd.detach(gridID);
gd.close(fid);
```

See Also

`gd.defProj` | `gd.sphereCodeToName` | `gd.sphereNameToCode`

matlab.io.hdfeos.gd.readAttr

Package: `matlab.io.hdfeos.gd`

Read grid attribute

Syntax

```
data = readAttr(gridID,attrname)
```

Description

`data = readAttr(gridID,attrname)` reads a grid attribute.

This function corresponds to the `GDreadattr` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf','read');
gridID = gd.attach(gfid,'PolarGrid');
data = gd.readAttr(gridID,'creation_date');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.writeAttr`

matlab.io.hdfeos.gd.readBlkSomOffset

Package: `matlab.io.hdfeos.gd`

Read Block SOM offset

Syntax

```
offset = readBlkSomOffset(GID)
```

Description

`offset = readBlkSomOffset(GID)` reads the block SOM offset values, in pixels, from a standard SOM (Space Oblique Mercator) projection. `offset` is a vector of offset values for SOM projection data. This routine can only be used with grids that use the SOM projection.

This function corresponds to the `GDblkSOMoffset` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
lowright = [30521379.68485 1152027.64253];
upleft = [-11119487.42844 8673539.24806];
gridID = gd.create(gfid,'SOM',120,60,upleft,lowright);
projparm(1) = 6378137;
projparm(2) = 0.006694348;
projparm(4) = 98096360;  % 98.161 in DDDMMMSSS
projparm(5) = 87069061;  % 87.112 in DDDMMMSSS
projparm(9) = 0.068585416*1440;
projparm(10) = 0.0;
projparm(12) = 6;
gd.defProj(gridID,'som',[],[],projparm);
gd.writeBlkSomOffset(gridID,[5 10 12 8 2]);
gd.detach(gridID);
```

```
gd.close(gfid);
gfid = gd.open('myfile.hdf');
gridID = gd.attach(gfid,'SOM');
blk = gd.readBlkSomOffset(gridID);
gd.detach(gridID);
gd.close(gfid);
```

See Also

gd.writeBlkSomOffset

matlab.io.hdfeos.gd.readField

Package: `matlab.io.hdfeos.gd`

Read data from grid field

Syntax

```
data = readField(gridID,fieldname)
data = readField(gridID,fieldname,start,count)
data = readField(gridID,fieldname,start,count,stride)
[data,lat,lon] = readField( ___ )
```

Description

`data = readField(gridID,fieldname)` reads the entire grid field identified by `fieldname` in the grid identified by `gridID`.

`data = readField(gridID,fieldname,start,count)` reads a contiguous hyperslab of data from the field. `start` specifies the zero-based starting index of the hyperslab. `count` specifies the number of values to read along each dimension.

`data = readField(gridID,fieldname,start,count,stride)` reads a strided hyperslab of data from the field. `stride` specifies the inter-element spacing along each dimension.

`[data,lat,lon] = readField( ___ )` reads the data and the associated geo-coordinates from the grid field. This syntax is only allowed when the leading two dimensions of the grid are `'XDim'` and `'YDim'`.

This function corresponds to the `GDreadfield` function in the HDF-EOS library C API.

Examples

Read the data, latitude, and longitude for the `'ice_temp'` field.

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
[data,lat,lon] = gd.readField(gridID,'ice_temp');
gd.detach(gridID);
gd.close(gfid);
```

Read only the first 4x4 hyperslab of data, latitude, and longitude for the `'ice_temp'` field.

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
[data2,lat2,lon2] = gd.readField(gridID,'ice_temp',[0 0], [4 4]);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.writeField`

matlab.io.hdfeos.gd.readTile

Package: `matlab.io.hdfeos.gd`

Read single tile of data from field

Syntax

```
data = readTile(gridID,fieldname,tileCoords)
```

Description

`data = readTile(gridID,fieldname,tileCoords)` reads a single of data from a field. If the data is to be read tile by tile, this routine is more efficient than `gd.readField`. In all other cases, use `gd.readField`. The `tileCoords` argument has the form `[rownum colnum]` and is defined in terms of the tile coordinates, not the data elements.

This function corresponds to the `GDreadtile` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `tileCoords` parameter is reversed with respect to the C library API.

Examples

Define a field with a 2-by-3 tiling scheme.

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf','read');
gridID = gd.attach(gfid,'PolarGrid');
for h = 0:9
    data = gd.readTile(gridID,'pressure',[0 0 h]);
end
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.tileInfo` | `gd.writeTile`

matlab.io.hdfeos.gd.regionInfo

Package: `matlab.io.hdfeos.gd`

Information about subsetted region

Syntax

```
[dims,upLeft,lowRight] = regionInfo(gridID,regionID,fieldname)
```

Description

`[dims,upLeft,lowRight] = regionInfo(gridID,regionID,fieldname)` returns the dimensions and corner points for the specified field of a subsetted region identified by `regionID` in the grid identified by `gridID`.

This function corresponds to the `GDregioninfo` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf','read');
gridID = gd.attach(gfid,'PolarGrid');
cornerlat = [20 50];
cornerlon = [-90 -60];
regionID = gd.defBoxRegion(gridID,cornerlat,cornerlon);
[dims,upleft,lowright] = gd.regionInfo(gridID,regionID,'ice_temp');
data = gd.extractRegion(gridID,regionID,'ice_temp');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defBoxRegion` | `gd.defVrtRegion`

matlab.io.hdfeos.gd.setFillValue

Package: `matlab.io.hdfeos.gd`

Set fill value for specified field

Syntax

```
setFillValue(gridID,fieldname,fillvalue)
```

Description

`setFillValue(gridID,fieldname,fillvalue)` sets the fill value for the specified field. The fill value should have the same data type as the field.

This function corresponds to the `GDsetfillvalue` function in the HDF-EOS library C API.

Examples

Create a new double-precision field with a fill value of -1.

```
import matlab.io.hdfeos.*
srcFile = fullfile(matlabroot,'toolbox','matlab','imagesci','grid.hdf');
copyfile(srcFile,'myfile.hdf');
fileattrib('myfile.hdf','+w');
gfid = gd.open('myfile.hdf','rdwr');
gridID = gd.attach(gfid,'PolarGrid');
gd.defComp(gridID,'none');
gd.defField(gridID,'newfield',{'XDim','YDim'},'double');
gd.setFillValue(gridID,'newfield',-1);
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.getFillValue`

matlab.io.hdfeos.gd.setTileComp

Package: `matlab.io.hdfeos.gd`

Set tiling and compression for field with fill value

Syntax

```
setTileComp(gridID,fieldname,tilesize,compCode,compParm)
```

Description

`setTileComp(gridID,fieldname,tilesize,compCode,compParm)` sets the tiling and compression for a field that had a fill value. This function must be applied after `gd.defField` and `gd.setFillValue`. The `compCode` argument can be one of the following values.

`'rle'`	Run-length encoding
`'skphuff'`	Skipping Huffman
`'deflate'`	Deflate
`'none'`	No compression

`compParm` need only be specified when the compression scheme is `'deflate'`, and then must be an integer between 0 and 9.

This function corresponds to the `GDsettilecomp` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `tilesize` parameter is reversed with respect to the C library API.

Examples

Define a temperature field with a 2-by-2 tiling scheme, a fill value of -999, and deflate compression.

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
upleft = [210584.50041 3322395.95445];
lowright = [813931.10959 2214162.53278];
gridID = gd.create(gfid,'UTMGrid',120,200,upleft,lowright);
spherecode = 0; zonecode = 40;
projparm = zeros(1,13);
gd.defProj(gridID,'utm',zonecode,spherecode,projparm);
gd.defDim(gridID,'Time',10);
gd.defField(gridID,'Pollution',{'XDim','YDim','Time'},'float');
gd.setFillValue(gridID,'Pollution',single(7));
gd.setTileComp(gridID,'Pollution',[40 20 1],'deflate',5);
gd.detach(gridID);
gd.close(gfid);
```

See Also

gd.defComp | gd.defTile

matlab.io.hdfeos.gd.sphereCodeToName

Package: `matlab.io.hdfeos.gd`

Name corresponding to GCTP sphere code

Syntax

```
name = sphereCodeToName(code)
```

Description

`name = sphereCodeToName(code)` returns the name for the spheroid corresponding to the spheroid code. The list of supported GCTP spheroids is as follows:

GCTP Spheroid Code	Spheroid Name
0	'Clarke 1866'
1	'Clarke 1880'
2	'Bessel'
3	'International 1967'
4	'International 1909'
5	'WGS 72'
6	'Everest'
7	'WGS 66'
8	'GRS 1980'
9	'Airy'
10	'Modified Airy'
11	'Modified Everest'
12	'WGS 84'
13	'Southeast Asia'

GCTP Spheroid Code	Spheroid Name
14	'Australian National'
15	'Krassovsky'
16	'Hough'
17	'Mercury 1960'
18	'Modified Mercury 1968'
19	'Sphere of radius 6370997m'
20	'Sphere of radius 6371228m'
21	'Sphere of radius 6371007.181m'

See Also

`gd.defProj` | `gd.sphereNameToCode`

matlab.io.hdfeos.gd.sphereNameToCode

Package: `matlab.io.hdfeos.gd`

Numeric GCTP code corresponding to sphere name

Syntax

```
code = sphereNameToCode(name)
```

Description

`code = sphereNameToCode(name)` returns the numeric GCTP code corresponding to the named spheroid. The list of supported GCTP spheroids is as follows:

GCTP Spheroid Code	Spheroid Name
0	'Clarke 1866'
1	'Clarke 1880'
2	'Bessel'
3	'International 1967'
4	'International 1909'
5	'WGS 72'
6	'Everest'
7	'WGS 66'
8	'GRS 1980'
9	'Airy'
10	'Modified Airy'
11	'Modified Everest'
12	'WGS 84'
13	'Southeast Asia'

GCTP Spheroid Code	Spheroid Name
14	'Australian National'
15	'Krassovsky'
16	'Hough'
17	'Mercury 1960'
18	'Modified Mercury 1968'
19	'Sphere of radius 6370997m'
20	'Sphere of radius 6371228m'
21	'Sphere of radius 6371007.181m'

See Also

`gd.defProj` | `gd.sphereCodeToName`

matlab.io.hdfeos.gd.tileInfo

Package: `matlab.io.hdfeos.gd`

Tile size of grid field

Syntax

```
tileDims = tileInfo(gridID,fieldname)
```

Description

`tileDims = tileInfo(gridID,fieldname)` returns the tile dimensions of the field specified by `fieldname` in the grid specified by `gridID`. If the field is not tiled, then `tileDims` is `[]`.

This function corresponds to the `GDtileinfo` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `tileDims` parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('grid.hdf');
gridID = gd.attach(gfid,'PolarGrid');
tileDims = gd.tileInfo(gridID,'pressure');
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.defTile`

matlab.io.hdfeos.gd.writeAttr

Package: `matlab.io.hdfeos.gd`

Write grid attribute

Syntax

```
writeAttr(gridID,attrname,data)
```

Description

`writeAttr(gridID,attrname,data)` writes an attribute to a grid. If the attribute does not exist, it is created. If the attribute exists, it can be modified in place, but it cannot be recreated with a different data type or length.

This function corresponds to the `GDwriteattr` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
srcFile = fullfile(matlabroot,'toolbox','matlab','imagesci','grid.hdf');
copyfile(srcFile,'myfile.hdf');
fileattrib('myfile.hdf','+w');
gfid = gd.open('myfile.hdf','rdwr');
gridID = gd.attach(gfid,'PolarGrid');
gd.writeAttr(gridID,'modification_date',datestr(now));
gd.detach(gridID);
gd.close(gfid);
```

See Also

`gd.readAttr`

matlab.io.hdfeos.gd.writeBlkSomOffset

Package: `matlab.io.hdfeos.gd`

Write Block SOM offset

Syntax

```
writeBlkSomOffset(gridID,offset)
```

Description

`writeBlkSomOffset(gridID,offset)` writes the block SOM offset values n pixels for a standard Solar Oblique Mercator (SOM) projection. `offset` is a vector of offset values for SOM projection data. This routine can only be used with grids that use the SOM projection. You must take care to use this function properly in conjunction with `gd.defProj`. The 12th element of the projection parameters must be set to the total number of blocks to be defined. `offset` starts by listing the offset to the second block, so the 12th element of the projection parameters is always one more than the length of `offset`.

All fields defined after writing the block SOM offset values will automatically include "SOMBlockDim" as the slowest varying dimension.

This function corresponds to the `GDblkSOMoffset` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
lowright = [30521379.68485 1152027.64253];
upleft = [-11119487.42844 8673539.24806];
gridID = gd.create(gfid,'SOM',120,60,upleft,lowright);
projparm(1) = 6378137;
projparm(2) = 0.006694348;
```

```
projparm(4) = 98096360;  % 98.161 in DDDMMMSSS
projparm(5) = 87069061;  % 87.112 in DDDMMMSSS
projparm(9) = 0.068585416*1440;
projparm(10) = 0.0;
projparm(12) = 6;
gd.defProj(gridID,'som',[],[],projparm);
gd.writeBlkSomOffset(gridID,[5 10 12 8 2]);
gd.detach(gridID);
gd.close(gfid);
```

See Also

gd.readBlkSomOffset

matlab.io.hdfeos.gd.writeField

Package: `matlab.io.hdfeos.gd`

Write data to grid field

Syntax

```
writeField(gridID,fieldname,data)
writeField(gridID,fieldname,start,data)
writeField(gridID,fieldname,start,stride,data)
```

Description

`writeField(gridID,fieldname,data)` writes all the data to a grid field. The field is identified by `fieldname` and the grid is identified by `gridID`.

`writeField(gridID,fieldname,start,data)` writes a contiguous hyperslab to the grid field. `start` specifies the zero-based starting index.

`writeField(gridID,fieldname,start,stride,data)` writes a strided hyperslab of data to a grid data field. `stride` specifies the inter-element spacing along each dimension. The number of elements to write along each dimension is inferred from the size of `data`.

This function corresponds to the `GDwritefield` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `start` and `stride` parameters are reversed with respect to the C library API.

Examples

Write all the data to a grid field.

```
import matlab.io.hdfeos.*
srcFile = fullfile(matlabroot,'toolbox','matlab','imagesci','grid.hdf');
copyfile(srcFile,'myfile.hdf');
```

```
fileattrib('myfile.hdf','+w');
gfid = gd.open('myfile.hdf','rdwr');
gridID = gd.attach(gfid,'PolarGrid');
data = zeros(100,100,'uint16');
gd.writeField(gridID,'ice_temp',data);
gd.detach(gridID);
gd.close(gfid);
```

See Also

gd.readField

matlab.io.hdfeos.gd.writeTile

Package: `matlab.io.hdfeos.gd`

Write tile to field

Syntax

```
writeTile(gridID,fieldname,tileCoords,data)
```

Description

`writeTile(gridID,fieldname,tileCoords,data)` writes a single tile of data to a field. If the field data can be arranged tile by tile, this routine is more efficient than `gd.writeField`. In all other cases, use `gd.writeField`. The `tileCoords` argument has the form `[rownum colnum]` and is defined in terms of the tile coordinates, not the data elements.

This function corresponds to the `GDwritetile` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `tileCoords` parameter is reversed with respect to the C library API.

Examples

Define a field with a 2-by-3 tiling scheme.

```
import matlab.io.hdfeos.*
gfid = gd.open('myfile.hdf','create');
xdim = 200; ydim = 180;
gridID = gd.create(gfid,'PolarGrid',xdim,ydim,[],[]);
zonecode = 40;
spherecode = 0;
projParm = zeros(1,13);
projParm(6) =  90000000;
gd.defProj(gridID,'ps',[],spherecode,projParm);
tileSize = [100 60];
```

```
gd.defTile(gridID,tileSize);
dimlist = {'XDim','YDim'};
gd.defField(gridID,'Pressure',dimlist,'int32');
for c = 0:2
    for r = 0:1
        data = (r+c)*ones(tileSize,'int32');
        gd.writeTile(gridID,'Pressure',[r c],data);
    end
end
gd.detach(gridID);
gd.close(gfid);
```

See Also

gd.readTile

matlab.io.hdfeos.sw

Low-level access to HDF-EOS swath files

Description

To use these MATLAB functions, you must be familiar with the HDF-EOS library C interface. In most cases, the syntax of the MATLAB function is similar to the syntax of the corresponding HDF-EOS library function. The functions are implemented as the package `matlab.io.hdfeos.sw`. To use this package, prefix the function name with a package path, or use the import function to add the package to the current import list, prior to calling the function, for example,

```
import matlab.io.hdfeos.*
fileId = sw.open(filename);
```

Access

matlab.io.hdfeos.sw.attach	Attach to swath data set
matlab.io.hdfeos.sw.close	Close swath file
matlab.io.hdfeos.sw.create	Create new swath structure
matlab.io.hdfeos.sw.detach	Detach from swath
matlab.io.hdfeos.sw.open	Open swath file

Definition

matlab.io.hdfeos.sw.defComp	Set grid field compression
matlab.io.hdfeos.sw.defDataField	Define new data field within swath
matlab.io.hdfeos.sw.defDim	Define new dimension within swath
matlab.io.hdfeos.sw.defDimMap	Define mapping between geolocation and data dimensions
matlab.io.hdfeos.sw.defGeoField	Define new data field within swath

Basic I/O

matlab.io.hdfeos.sw.getFillValue	Fill value for specified field
matlab.io.hdfeos.sw.readAttr	Read swath attribute
matlab.io.hdfeos.sw.readField	Read data from swath field
matlab.io.hdfeos.sw.setFillValue	Set fill value for the specified field
matlab.io.hdfeos.sw.writeAttr	Write swath attribute
matlab.io.hdfeos.sw.writeField	Write data to swath field

Inquiry

matlab.io.hdfeos.sw.compInfo	Compression information for field
matlab.io.hdfeos.sw.dimInfo	Size of dimension
matlab.io.hdfeos.sw.fieldInfo	Information about swath field
matlab.io.hdfeos.sw.geoMapInfo	Type of dimension mapping for named dimension
matlab.io.hdfeos.sw.idxMapInfo	Indexed array of geolocation mapping
matlab.io.hdfeos.sw.inqAttrs	Names of swath attributes
matlab.io.hdfeos.sw.inqDataFields	Information about geolocation fields
matlab.io.hdfeos.sw.inqDims	Information about dimensions defined in swath
matlab.io.hdfeos.sw.inqGeoFields	Information about geolocation fields
matlab.io.hdfeos.sw.inqIdxMaps	Information about swath indexed geolocation mapping
matlab.io.hdfeos.sw.inqMaps	Information about swath geolocation relations
matlab.io.hdfeos.sw.inqSwath	Names of swaths in file
matlab.io.hdfeos.sw.mapInfo	Offset and increment of specific geolocation mapping
matlab.io.hdfeos.sw.nEntries	Number of entries for specific type

Subsetting

matlab.io.hdfeos.sw.defBoxRegion	Define latitude-longitude region for swath
matlab.io.hdfeos.sw.defTimePeriod	Define time period of interest
matlab.io.hdfeos.sw.defVrtRegion	Subset on monotonic field or dimension
matlab.io.hdfeos.sw.extractPeriod	Read data from subsetted time period
matlab.io.hdfeos.sw.extractRegion	Read subsetted region
matlab.io.hdfeos.sw.periodInfo	Information about subsetted period
matlab.io.hdfeos.sw.regionInfo	Information about subsetted region

matlab.io.hdfeos.sw.attach

Package: `matlab.io.hdfeos.sw`

Attach to swath data set

Syntax

```
swathID = attach(swfID,swathname)
```

Description

`swathID = attach(swfID,swathname)` attaches to the swath identified by `swathname` in the file identified by `swfID`. The `swathID` output is the identifier for the named swath.

This function corresponds to the `SWattach` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.detach`

matlab.io.hdfeos.sw.close

Package: `matlab.io.hdfeos.sw`

Close swath file

Syntax

```
close(swfID)
```

Description

`close(swfID)` closes an HDF-EOS swath file identified by `swfID`.

This function corresponds to the `SWclose` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('myfile.hdf','create');
swathID = sw.create(swfid,'ExampleSwath');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.create` | `sw.open`

matlab.io.hdfeos.sw.compInfo

Package: `matlab.io.hdfeos.sw`

Compression information for field

Syntax

```
[code,parms] = compInfo(swathID,fieldname)
```

Description

`[code,parms] = compInfo(swathID,fieldname)` returns the compression code and compression parameters for a given field. Refer to `sw.defComp` for a description of various compression schemes and parameters.

This function corresponds to the `SWcompinfo` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
[compCode,parms] = sw.compInfo(swathID,'Spectra');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defComp`

matlab.io.hdfeos.sw.create

Package: `matlab.io.hdfeos.sw`

Create new swath structure

Syntax

```
swathID = create(swfID,swathname)
```

Description

`swathID = create(swfID,swathname)` creates a new swath structure where `swfID` is the swath file identifier and `swathname` is the name of the new swath. The swath is created as a Vgroup with the HDF file with the name `swathname` and HDF Vgroup class `'SWATH'`.

This function corresponds to the `SWcreate` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('myfile.hdf','create');
swathID = sw.create(swfid,'ExampleSwath');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.detach`

matlab.io.hdfeos.sw.defBoxRegion

Package: `matlab.io.hdfeos.sw`

Define latitude-longitude region for swath

Syntax

```
regionID = defBoxRegion(swathID,lat,lon,mode)
```

Description

`regionID = defBoxRegion(swathID,lat,lon,mode)` defines a latitude-longitude box region for a swath. `lat` and `lon` are two-element arrays containing the latitude and longitude in decimal degrees of the box corners. A cross track is determined to be within the box if a condition is met according to the value of `mode`:

`'MIDPOINT'`	The cross track midpoint is within the box.
`'ENDPOINT'`	Either endpoint is within the box.
`'ANYPOINT'`	Any point of the cross track is within the box.

All elements of a cross track are within the region if the condition is met. The swath must have both Longitude and Latitude (or Colatitude) defined.

`regionID` is an identifier to be used by `sw.extractRegion` to read all the entries of a data field within the region.

This function corresponds to the `SWdefboxregion` and `SWregionindex` functions in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
```

```
lat = [34 44];
lon = [16 24];
regionID = sw.defBoxRegion(swathID,lat,lon,'MIDPOINT');
data = sw.extractRegion(swathID,regionID,'Temperature');
sw.detach(swathID);
sw.close(swfid);
```

See Also

sw.extractRegion

matlab.io.hdfeos.sw.defComp

Package: `matlab.io.hdfeos.sw`

Set grid field compression

Syntax

```
defComp(swathID,compscheme,compparm)
```

Description

`defComp(swathID,compscheme,compparm)` sets the field compression for subsequent definitions. The compression scheme does not apply to one-dimensional fields. `compscheme` can be one of these values:

`'rle'`	Run-length encoding
`'skphuff'`	Skipping Huffman
`'deflate'`	Gzip compression
`'none'`	No compression

When the compression scheme is `'deflate'`, the `compparm` input is the deflate compression level, an integer between 0 and 9. `compparm` can be omitted for the other compression schemes.

Fields defined with compression must be written with a single call to `sw.writeField`.

This function corresponds to the `SWdefcomp` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('myfile.hdf','create');
swathID = sw.create(swfid,'MySwath');
sw.defDim(swathID,'Track',4000);
```

```
sw.defDim(swathID,'Xtrack',2000);
sw.defDim(swathID,'Bands',3);
sw.defComp(swathID,'rle');
dims = {'Xtrack','Track'};
sw.defDataField(swathID,'Pressure',dims,'float');
sw.defComp(swathID,'deflate',5);
sw.defDataField(swathID,'Opacity',dims,'float');
sw.defComp(swathID,'skphuff');
dims = {'Xtrack','Track','Bands'};
sw.defDataField(swathID,'Spectra',dims,'float');
sw.defComp(swathID,'none');
dims = {'Xtrack','Track'};
sw.defDataField(swathID,'Temperature',dims,'float');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.compInfo`

matlab.io.hdfeos.sw.defDataField

Package: `matlab.io.hdfeos.sw`

Define new data field within swath

Syntax

```
defDataField(swathID,fieldname,dimlist,dtype)
defDataField(swathID,fieldname,dimlist,dtype,mergeCode)
```

Description

`defDataField(swathID,fieldname,dimlist,dtype)` defines a data field to be stored in the swath identified by `swathID`.

The `dimlist` input can be a cell array character vectors or a string array containing dimension names, or a single character vector or string scalar if there is only one dimension. `dimlist` should be ordered such that the fastest varying dimension is listed first. This is opposite from the order in which the dimensions are listed in the C API.

`dtype` is the data type of the field and can be one of the following values.

- `'double'`
- `'single'`
- `'int32'`
- `'uint32'`
- `'int16'`
- `'uint16'`
- `'int8'`
- `'uint8'`
- `'char'`

`defDataField(swathID,fieldname,dimlist,dtype,mergeCode)` defines a data field that can be merged with other data fields according to the value of `mergeCode`. The

`mergeCode` input can be `'automerge'` or `'nomerge'`. If `mergeCode` is `'automerge'`, then the HDF-EOS library will attempt to merge swath fields into a single object. This should not be done if you wish to access the swath fields individually with the another interface. By default, `mergeCode` is `'nomerge'`.

Note To assure that the fields defined by `sw.defDataField` are properly established in the file, the swath should be detached and then reattached before writing to any fields.

This function corresponds to the `SWdefdatafield` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `dimlist` parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('myfile.hdf','create');
swathID = sw.create(swfid,'MySwath');
sw.defDim(swathID,'GeoTrack',2000);
sw.defDim(swathID,'GeoXtrack',1000);
sw.defDim(swathID,'DataTrack',4000);
sw.defDim(swathID,'DataXtrack',2000);
sw.defDim(swathID,'Bands',3);
sw.defDimMap(swathID,'GeoTrack','DataTrack',0,2);
sw.defDimMap(swathID,'GeoXtrack','DataXtrack',1,2);
dims = {'GeoXtrack','GeoTrack'};
sw.defGeoField(swathID,'Longitude',dims,'float');
sw.defGeoField(swathID,'Latitude',dims,'float');
dims = {'DataXtrack','DataTrack','Bands'};
sw.defDataField(swathID,'Spectra',dims,'float');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defGeoField` | `sw.inqDataFields`

matlab.io.hdfeos.sw.defDim

Package: `matlab.io.hdfeos.sw`

Define new dimension within swath

Syntax

`defDim(swathID,dimname,dimlen)`

Description

`defDim(swathID,dimname,dimlen)` defines a new dimension named `dimname` with length `dimlen` in the swath structure identified by `swathID`.

To specify an unlimited dimension, use either 0 or `'unlimited'` for `dimlen`.

This function corresponds to the `SWdefdim` function in the HDF-EOS library.

Examples

Define a dimension `'Band'` with length of 15 and an unlimited dimension `'Time'`.

```
import matlab.io.hdfeos.*
swfid = sw.open('myfile.hdf','create');
swathID = sw.create(swfid,'MySwath');
sw.defDim(swathID,'GeoTrack',2000);
sw.defDim(swathID,'GeoXtrack',1000);
sw.defDim(swathID,'DataTrack',4000);
sw.defDim(swathID,'DataXtrack',2000);
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.dimInfo`

matlab.io.hdfeos.sw.defDimMap

Package: `matlab.io.hdfeos.sw`

Define mapping between geolocation and data dimensions

Syntax

```
defDimMap(swathID,geoDim,dataDim,offset,increment)
```

Description

`defDimMap(swathID,geoDim,dataDim,offset,increment)` defines a monotonic mapping between the geolocation and data dimensions, which usually have differing lengths. `offset` gives the index of the data element corresponding to the first geolocation element, and `increment` gives the number of data elements to skip for each geolocation element. If the geolocation dimension begins before the data dimension, then `offset` is negative. Similarly, if the geolocation dimension has higher resolution than the data dimension, then `increment` is negative.

This function corresponds to the `SWdefdimmap` function in the HDF-EOS library.

Examples

Create a dimension mapping such that the first element of the GeoTrack dimension corresponds to the first element of the DataTrack Dimension and such that the data dimension has twice the resolution as the geolocation dimension. Also create a dimension mapping such that the first element of the GeoXtrack dimension corresponds to the second element of the DataXtrack dimensions and such that the data dimension has twice the resolution as the geolocation dimension.

```
import matlab.io.hdfeos.*
swfid = sw.open('myfile.hdf','create');
swathID = sw.create(swfid,'MySwath');
sw.defDim(swathID,'GeoTrack',2000);
sw.defDim(swathID,'GeoXtrack',1000);
```

```
sw.defDim(swathID,'DataTrack',4000);
sw.defDim(swathID,'DataXtrack',2000);
sw.defDimMap(swathID,'GeoTrack','DataTrack',0,2);
sw.defDimMap(swathID,'GeoXtrack','DataXtrack',1,2);
sw.detach(swathID);
sw.close(swfid);
```

See Also

sw.defDim | sw.mapInfo

matlab.io.hdfeos.sw.defGeoField

Package: `matlab.io.hdfeos.sw`

Define new data field within swath

Syntax

```
defGeoField(swathID,fieldname,dimlist,dtype)
defGeoField(swathID,fieldname,dimlist,dtype,mergeCode)
```

Description

`defGeoField(swathID,fieldname,dimlist,dtype)` defines a geolocation field to be stored in the swath identified by `swathID`.

The `dimlist` input can be a cell array character vectors or a string array containing dimension names, or a single character vector or string scalar if there is only one dimension. `dimlist` should be ordered such that the fastest varying dimension is listed first. This is opposite from the order in which the dimensions are listed in the C API.

`dtype` is the data type of the field.

`defGeoField(swathID,fieldname,dimlist,dtype,mergeCode)` defines a geolocation field that may be merged with other geolocation fields according to the value of `mergeCode`. The `mergeCode` argument can `'automerge'` or `'nomerge'`. If `mergeCode` is `'automerge'`, then the HDF-EOS library will attempt to merge swath fields into a single object. This should not be done if you wish to access the swath fields individually with the another interface. By default, `mergeCode` is `'nomerge'`.

This function corresponds to the `SWdefgeofield` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `dimlist` parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('myfile.hdf','create');
swathID = sw.create(swfid,'MySwath');
sw.defDim(swathID,'GeoTrack',2000);
sw.defDim(swathID,'GeoXtrack',1000);
sw.defDim(swathID,'DataTrack',4000);
sw.defDim(swathID,'DataXtrack',2000);
sw.defDimMap(swathID,'GeoTrack','DataTrack',0,2);
sw.defDimMap(swathID,'GeoXtrack','DataXtrack',1,2);
dims = {'GeoXtrack','GeoTrack'};
sw.defGeoField(swathID,'Longitude',dims,'float');
sw.defGeoField(swathID,'Latitude',dims,'float');
sw.detach(swathID);
sw.close(swfid);
```

See Also

sw.defDataField | sw.inqGeoFields

matlab.io.hdfeos.sw.defTimePeriod

Package: `matlab.io.hdfeos.sw`

Define time period of interest

Syntax

```
outpID = defTimePeriod(swathID,start,stop,mode)
```

Description

`outpID = defTimePeriod(swathID,start,stop,mode)` defines a time period for a swath. `outpID` is a swath period ID that can be used to read all the entries of a data field within the time period. The swath structure must have the `'Time'` field defined. A cross track is within a time period if a condition is met according to the value of `mode`:

`'MIDPOINT'`	The midpoint is within the time period.
`'ENDPOINT'`	Either endpoint is within the time period.

This function corresponds to the `SWdeftimeperiod` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
starttime =  25;
stoptime = 425;
periodID = sw.defTimePeriod(swathID,starttime,stoptime,'MIDPOINT');
data = sw.extractPeriod(swathID,periodID,'Temperature');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defBoxRegion` | `sw.defVrtRegion` | `sw.extractPeriod`

matlab.io.hdfeos.sw.defVrtRegion

Package: `matlab.io.hdfeos.sw`

Subset on monotonic field or dimension

Syntax

```
regionID_out = defVrtRegion(swathID,regionID,vertObj,range)
```

Description

`regionID_out = defVrtRegion(swathID,regionID,vertObj,range)` subsets on a monotonic field or contiguous elements of a dimension. Whereas `defBoxRegion` and `defTimePeriod` subset along the `'Track'` dimension, this routine allows the user to subset along any dimension. `regionID` specifies the subsetted region from a previous call. `vertObj` specifies the dimension by which to subset. `range` specifies the minimum and maximum values for `vertObj`.

If there is no current subsetted region, `regionID` should be `'noprevsub'`.

`vertObj` can be either a dimension or a field. If it is a dimension, then `range` should consist of dimension indices. If `vertObj` corresponds to a field, then `range` should consist of the minimum and maximum field values. `vertObj` must be one-dimensional in this case, and the its values must be monotonic.

This function corresponds to the `SWdefvrtregion` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
regionID = sw.defVrtRegion(swathID,'noprevsub','Bands',[450 600]);
data = sw.extractRegion(swathID,regionID,'Spectra');
```

```
sw.detach(swathID);
sw.close(swfid);
```

See Also

sw.defBoxRegion | sw.defTimePeriod

matlab.io.hdfeos.sw.detach

Package: `matlab.io.hdfeos.sw`

Detach from swath

Syntax

```
detach(swathID)
```

Description

`detach(swathID)` detaches from the swath identified by `swathID`.

This function corresponds to the `SWdetach` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.attach` | `sw.create`

matlab.io.hdfeos.sw.dimInfo

Package: `matlab.io.hdfeos.sw`

Size of dimension

Syntax

```
dimlen = dimInfo(swathID,dimname)
```

Description

`dimlen = dimInfo(swathID,dimname)` returns the length of the specified dimension.

This function corresponds to the `SWdiminfo` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
dimlen = sw.dimInfo(swathID,'GeoTrack');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defDim`

matlab.io.hdfeos.sw.extractPeriod

Package: `matlab.io.hdfeos.sw`

Read data from subsetted time period

Syntax

```
data = extractPeriod(swathID,periodID,fieldname)
```

Description

`data = extractPeriod(swathID,periodID,fieldname)` reads data for the given field for the time period specified by `periodID`.

This routine corresponds to the `SWextractperiod` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
starttime =  25;
stoptime = 425;
periodID = sw.defTimePeriod(swathID,starttime,stoptime,'MIDPOINT');
data = sw.extractPeriod(swathID,periodID,'Temperature');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defTimePeriod`

matlab.io.hdfeos.sw.extractRegion

Package: `matlab.io.hdfeos.sw`

Read subsetted region

Syntax

```
data = extractRegion(swathID,regionID,fieldname)
```

Description

`data = extractRegion(swathID,regionID,fieldname)` reads data for a specified field from a subsetted region identified by `regionID`.

This function corresponds to the `SWextractregion` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
lat = [34 44];
lon = [16 24];
regionID = sw.defBoxRegion(swathID,lat,lon,'MIDPOINT');
data = sw.extractRegion(swathID,regionID,'Temperature');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defBoxRegion` | `sw.defVrtRegion`

matlab.io.hdfeos.sw.fieldInfo

Package: `matlab.io.hdfeos.sw`

Information about swath field

Syntax

```
[dimsizes,ntype,dimlist] = fieldInfo(swathID,fieldname)
```

Description

`[dimsizes,ntype,dimlist] = fieldInfo(swathID,fieldname)` returns the size, data type, and list of named dimensions for the specified swath geolocation or data field.

This function corresponds to the `SWfieldinfo` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `dimlist` parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
[fieldSize,ntype,dimlist] = sw.fieldInfo(swathID,'Spectra');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.inqDataFields` | `sw.inqGeoFields`

matlab.io.hdfeos.sw.geoMapInfo

Package: `matlab.io.hdfeos.sw`

Type of dimension mapping for named dimension

Syntax

```
mappingType = geoMapInfo(swathID,dimname)
```

Description

`mappingType = geoMapInfo(swathID,dimname)` returns the type of dimension mapping for the named dimension. `mappingType` is one of these values: `'indexed'`, `'regular'`, or `'unmapped'`.

This routine corresponds to the `SWgeomapinfo` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
maptype = sw.geoMapInfo(swathID,'GeoTrack');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defDimMap`

matlab.io.hdfeos.sw.getFillValue

Package: `matlab.io.hdfeos.sw`

Fill value for specified field

Syntax

```
fillvalue = getFillValue(swathID,fieldname)
```

Description

`fillvalue = getFillValue(swathID,fieldname)` returns the fill value for the specified field.

This function corresponds to the `SWgetfillvalue` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
fv = sw.getFillValue(swathID,'Spectra');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.setFillValue`

matlab.io.hdfeos.sw.idxMapInfo

Package: `matlab.io.hdfeos.sw`

Indexed array of geolocation mapping

Syntax

```
idx = idxMapInfo(swathID,geodim,datadim)
```

Description

`idx = idxMapInfo(swathID,geodim,datadim)` retrieves the indexed elements of the geolocation mapping between `geodim` and `datadim`.

This function corresponds to the `SWidxmapinfo` function in the HDF-EOS C library API.

See Also

`sw.geoMapInfo`

matlab.io.hdfeos.sw.inqAttrs

Package: `matlab.io.hdfeos.sw`

Names of swath attributes

Syntax

```
attrlist = inqAttrs(swathID)
```

Description

`attrlist = inqAttrs(swathID)` returns the list of swath attribute names. `attrlist` is a cell array.

This function corresponds to the `SWinqattrs` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
attrList = sw.inqAttrs(swathID);
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.readAttr` | `sw.writeAttr`

matlab.io.hdfeos.sw.inqDataFields

Package: `matlab.io.hdfeos.sw`

Information about geolocation fields

Syntax

```
[fields,rank,datatype] = inqDataFields(swathID)
```

Description

`[fields,rank,datatype] = inqDataFields(swathID)` returns the list of geolocation field names, the rank of each field, and the data type of each field.

This function corresponds to the `SWinqdatafields` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `fields` parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
[fields,rank,datatype] = sw.inqDataFields(swathID);
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defDataField` | `sw.inqGeoFields`

matlab.io.hdfeos.sw.inqDims

Package: `matlab.io.hdfeos.sw`

Information about dimensions defined in swath

Syntax

```
[dimnames,dimlens] = inqDims(swathID)
```

Description

`[dimnames,dimlens] = inqDims(swathID)` returns the names of the dimensions `dimnames` as a cell array. The length of each respective dimension is returned in `dimlens`.

This function corresponds to the `SWinqdims` routine in the HDF-EOS library.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
[dimnames,dimlens] = sw.inqDims(swathID);
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defDim`

matlab.io.hdfeos.sw.inqGeoFields

Package: `matlab.io.hdfeos.sw`

Information about geolocation fields

Syntax

```
[fields,rank,datatype] = inqGeoFields(swathID)
```

Description

`[fields,rank,datatype] = inqGeoFields(swathID)` returns the list of geolocation fields `fields`, the rank of each field, and the data type of each field.

This function corresponds to the `SWinqgeofields` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `fields` parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
[fields,rank,datatypes] = sw.inqGeoFields(swathID);
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defGeoField` | `sw.inqDataFields`

matlab.io.hdfeos.sw.inqIdxMaps

Package: `matlab.io.hdfeos.sw`

Information about swath indexed geolocation mapping

Syntax

```
[idxMap,idxSize] = inqIdxMaps(swathID)
```

Description

`[idxMap,idxSize] = inqIdxMaps(swathID)` retrieves all indexed geolocation/data mappings defined in the swath. `idxMap` is a cell array with each element consisting of the names of the dimensions of a mapping, separated by a `'/'`. `idxSize` contains the size of the index arrays corresponding to each mapping.

This function corresponds to the `SWinqidxmaps` routine in the HDF-EOS library.

See Also

`sw.inqMaps`

matlab.io.hdfeos.sw.inqMaps

Package: `matlab.io.hdfeos.sw`

Information about swath geolocation relations

Syntax

```
[map,offset,increment] = inqMaps(swathID)
```

Description

`[map,offset,increment] = inqMaps(swathID)` returns the dimension mapping list, the offset of each geolocation relation, and the increment of each geolocation relation. These mappings are not indexed. `map` is a cell array where each element contains the names of the dimensions for each mapping, separated by a slash. `offset` and `increment` contain the offset and increment of each geolocation relation.

This function corresponds to the `SWinqmaps` routine in the HDF-EOS library.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
[dimmap,offset,increment] = sw.inqMaps(swathID);
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defDimMap` | `sw.inqDims` | `sw.inqIdxMaps`

matlab.io.hdfeos.sw.inqSwath

Package: `matlab.io.hdfeos.sw`

Names of swaths in file

Syntax

```
swaths = inqSwath(filename)
```

Description

`swaths = inqSwath(filename)` returns a cell array containing the names of all the swaths in a file.

This function corresponds to the `SWinqswath` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swaths = sw.inqSwath('swath.hdf');
```

See Also

`gd.inqGrid`

matlab.io.hdfeos.sw.mapInfo

Package: `matlab.io.hdfeos.sw`

Offset and increment of specific geolocation mapping

Syntax

```
[offset,increment] = mapInfo(swathID,geodim,datadim)
```

Description

`[offset,increment] = mapInfo(swathID,geodim,datadim)` retrieves the offset and increment of the geolocation mapping between the specified geolocation dimension and the specified data dimension.

This function corresponds to the `SWmapinfo` function in the HDF-EOS C library API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('myfile.hdf','create');
swathID = sw.create(swfid,'MySwath');
sw.defDim(swathID,'GeoTrack',2000);
sw.defDim(swathID,'GeoXtrack',1000);
sw.defDim(swathID,'DataTrack',4000);
sw.defDim(swathID,'DataXtrack',2000);
sw.defDimMap(swathID,'GeoTrack','DataTrack',0,2);
sw.defDimMap(swathID,'GeoXtrack','DataXtrack',1,2);
sw.detach(swathID);
sw.close(swfid);
swfid = sw.open('myfile.hdf','read');
swathID = sw.attach(swfid,'MySwath');
[offset,increment] = sw.mapInfo(swathID,'GeoTrack','DataTrack');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defDimMap`

matlab.io.hdfeos.sw.nEntries

Package: `matlab.io.hdfeos.sw`

Number of entries for specific type

Syntax

```
nEnts = nEntries(swathID,type)
```

Description

`nEnts = nEntries(swathID,type)` returns the number of entries in a swath. Valid inputs for `type` include:

`'dims'`	or `'HDFE_NENTDIM'`
`'maps'`	or `'HDFE_NENTMAP'`
`'imaps'`	or `'HDFE_NENTIMAP'`
`'geofields'`	or `'HDFE_NENTGFLD'`
`'datafields'`	or `'HDFE_NENTFLD'`

This function corresponds to the `SWnentries` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('myfile.hdf','create');
swathID = sw.create(swfid,'MySwath');
sw.defDim(swathID,'GeoTrack',2000);
sw.defDim(swathID,'GeoXtrack',1000);
sw.defDim(swathID,'DataTrack',4000);
sw.defDim(swathID,'DataXtrack',2000);
ndims = sw.nEntries(swathID,'dims');
```

```
sw.detach(swathID);
sw.close(swfid);
```

matlab.io.hdfeos.sw.open

Package: `matlab.io.hdfeos.sw`

Open swath file

Syntax

```
swfID = open(filename)
swfID = open(filename,access)
```

Description

`swfID = open(filename)` opens an HDF-EOS swath file for read-only access.

`swfID = open(filename,access)` opens or creates an HDF-EOS swath file identified by `filename` and returns a file ID. `access` can be one of the following values.

`'read'` (default)	Read-only
`'rdwr'`	Read-write
`'create'`	Creates a file, deleting it if it already exists

This routine corresponds to the `SWopen` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
sw.close(swfid);
```

See Also

`sw.close`

matlab.io.hdfeos.sw.periodInfo

Package: `matlab.io.hdfeos.sw`

Information about subsetted period

Syntax

```
[datatype,dims] = periodInfo(swathID,periodID,fieldname)
```

Description

`[datatype,dims] = periodInfo(swathID,periodID,fieldname)` retrieves information about the period defined for the given field. `datatype` is the data type of the field. `dims` is the dimensions of the subsetted region.

This function corresponds to the `SWperiodinfo` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `dims` parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
starttime =  25;
stoptime = 425;
periodID = sw.defTimePeriod(swathID,starttime,stoptime,'MIDPOINT');
[ntype,dims] = sw.periodInfo(swathID,periodID,'Temperature');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defTimePeriod` | `sw.extractPeriod`

matlab.io.hdfeos.sw.readAttr

Package: `matlab.io.hdfeos.sw`

Read swath attribute

Syntax

```
data = readAttr(swathID,attrname)
```

Description

`data = readAttr(swathID,attrname)` reads a swath attribute.

This function corresponds to the `SWreadAttr` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
value = sw.readAttr(swathID,'creation_date');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.writeAttr`

matlab.io.hdfeos.sw.readField

Package: `matlab.io.hdfeos.sw`

Read data from swath field

Syntax

```
data = readField(swathID,fieldname)
data = readField(swathID,fieldname,start,count)
data = readField(swathID,fieldname,start,count,stride)
```

Description

`data = readField(swathID,fieldname)` reads an entire swath field.

`data = readField(swathID,fieldname,start,count)` reads a contiguous hyperslab of data from the swath field `fieldname`. The `start` input specifies the zero-based index of the first element to be read. `count` specifies the number of elements along each dimension to read.

`data = readField(swathID,fieldname,start,count,stride)` reads a strided hyperslab of data from the swath field `fieldname`. The `stride` input specifies the inter-element spacing along each dimension.

This function corresponds to the `SWreadfield` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `start`, `count`, and `stride` parameters are reversed with respect to the C library API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
data = sw.readField(swathID,'Longitude');
```

```
sw.detach(swathID);
sw.close(swfid);
```

See Also

sw.writeField

matlab.io.hdfeos.sw.regionInfo

Package: `matlab.io.hdfeos.sw`

Information about subsetted region

Syntax

```
[datatype,extent] = regionInfo(swathID,regionID,fieldname)
```

Description

`[datatype,extent] = regionInfo(swathID,regionID,fieldname)` returns the data type and extent of a subsetted region of a field. `regionID` is the identifier for the subsetted region.

This function corresponds to the `SWregioninfo` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `extent` parameter is reversed with respect to the C library API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('swath.hdf');
swathID = sw.attach(swfid,'Example Swath');
lat = [34 44];
lon = [16 24];
regionID = sw.defBoxRegion(swathID,lat,lon,'MIDPOINT');
[ntype,dims] = sw.regionInfo(swathID,regionID,'Temperature');
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.defBoxRegion` | `sw.defVrtRegion`

matlab.io.hdfeos.sw.setFillValue

Package: `matlab.io.hdfeos.sw`

Set fill value for the specified field

Syntax

```
setFillValue(swathID,fieldname,fillvalue)
```

Description

`setFillValue(swathID,fieldname,fillvalue)` sets the fill value for the specified field. The field must have more than two dimensions.

This function corresponds to the `SWsetfillvalue` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('myfile.hdf','create');
swathID = sw.create(swfid,'MySwath');
sw.defDim(swathID,'Track',400);
sw.defDim(swathID,'Xtrack',200);
dims = {'Track','Xtrack'};
sw.defDataField(swathID,'Temperature',dims,'float');
sw.setFillValue(swathID,'Temperature',single(-999));
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.getFillValue`

matlab.io.hdfeos.sw.writeAttr

Package: `matlab.io.hdfeos.sw`

Write swath attribute

Syntax

`writeAttr(swathID,attrname,data)`

Description

`writeAttr(swathID,attrname,data)` writes an attribute to a swath. If the attribute does not exist, it is created. If the attribute exists, it can be modified in place, but it cannot be recreated with a different data type or length.

This function corresponds to the `SWwriteattr` function in the HDF-EOS library C API.

Examples

```
import matlab.io.hdfeos.*
swfid = sw.open('myfile.hdf','create');
swathID = sw.create(swfid,'MySwath');
sw.writeAttr(swathID,'creation_date', datestr(now));
sw.detach(swathID);
sw.close(swfid);
```

See Also

`sw.readAttr`

matlab.io.hdfeos.sw.writeField

Package: `matlab.io.hdfeos.sw`

Write data to swath field

Syntax

```
writeField(swathID,fieldname,data)
writeField(swathID,fieldname,start,data)
writeField(swathID,fieldname,start,stride,data)
```

Description

`writeField(swathID,fieldname,data)` writes an entire swath data field.

`writeField(swathID,fieldname,start,data)` writes a contiguous hyperslab to a swath field. `start` specifies the index of the first element to write. The number of elements along each dimension is inferred from either the size of `data` or from the swath field itself.

`writeField(swathID,fieldname,start,stride,data)` writes a strided hyperslab to a swath field. `stride` specifies the inter-element spacing along each dimension.

This function corresponds to the `SWwritefield` function in the HDF-EOS library C API, but because MATLAB uses FORTRAN-style ordering, the `start` and `stride` parameters are reversed with respect to the C library API.

Examples

Write data to a geolocation field `'Longitude'`.

```
lon = [-50:49];
data = repmat(lon(:),1,100);
data = single(data);
import matlab.io.hdfeos.*
```

```
srcFile = fullfile(matlabroot,'toolbox','matlab','imagesci','swath.hdf');
copyfile(srcFile,'myfile.hdf');
fileattrib('myfile.hdf','+w');
swfid = sw.open('myfile.hdf','rdwr');
swathID = sw.attach(swfid,'Example Swath');
sw.writeField(swathID,'Longitude',data);
sw.detach(swathID);
sw.close(swfid);
```

See Also

sw.readField

matlab.io.saveVariablesToScript

Save workspace variables to MATLAB script

Syntax

```
matlab.io.saveVariablesToScript(filename)
matlab.io.saveVariablesToScript(filename,varnames)
matlab.io.saveVariablesToScript(filename,Name,Value)
[r1,r2] = matlab.io.saveVariablesToScript(filename)
```

Description

`matlab.io.saveVariablesToScript(filename)` saves variables in the current workspace to a MATLAB script named `filename.m`. The filename can include the `.m` suffix. If you do not include it, the function adds it when it creates the file.

Variables that MATLAB cannot generate code for are saved to a MAT-file named `filename.mat`.

If a file with the same name already exists, it is overwritten.

`matlab.io.saveVariablesToScript(filename,varnames)` saves only workspace variables specified by `varnames` to the MATLAB script.

`matlab.io.saveVariablesToScript(filename,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`[r1,r2] = matlab.io.saveVariablesToScript(filename)` additionally returns two cell arrays:

- `r1` for variables that were saved to the MATLAB script
- `r2` for variables that were saved to a MAT-file

Examples

Save Workspace Variables to MATLAB Script

Save variables from a workspace to a MATLAB script, `test.m`.

```
matlab.io.saveVariablesToScript('test.m')
```

Save Specific Workspace Variables to MATLAB Script

Create and save variable `myVar` from a workspace to a MATLAB script, `test.m`.

```
myVar = 55.3;
matlab.io.saveVariablesToScript('test.m','myVar')
```

Append Specific Variables to Existing MATLAB Script

Create two variables, `a` and `b`, and save them to an existing MATLAB script `abfile.m`.

```
a = 72.3;
b = pi;
matlab.io.saveVariablesToScript('abfile.m',{'a','b'},...
'SaveMode','append')
```

Update Specific Variables in Existing MATLAB Script

Update and save two variables, `y` and `z`, to an existing MATLAB script `yzfile.m`.

```
y = 15.7;
z = 3 * pi;
matlab.io.saveVariablesToScript('yzfile.m',{'y','z'},...
'SaveMode','update')
```

Specify MATLAB Script Configuration for Saving Variable

Update and save variable `resistance` to an existing MATLAB script `designData.m` while specifying the configuration of the script file.

```
resistance = [10 20.5 11 13.7 15.1 7.7];
matlab.io.saveVariablesToScript('designData.m','resistance',...
```

```
'SaveMode','Update','MaximumArraySize',5,...
'MaximumNestingLevel',5,'MaximumTextWidth',30)
```

Specify 2-D Slice for Saving 3-D Array in MATLAB Script

Specify a 2-D slice for the output of the 3-D array `my3Dtable`, such that the 2-D slice expands along the first and third dimensions. Save the 2-D slice in the MATLAB script `sliceData.m`.

```
level1 = [1 2; 3 4];
level2 = [5 6; 7 8];
my3Dtable( :, :, 1) = level1;
my3Dtable( :, :, 2) = level2;
matlab.io.saveVariablesToScript('sliceData.m','MultidimensionalFormat',[1,3])
```

The resulting MATLAB code is similar to the following:

```
level1 = ...
    [1 2;
    3 4];

level2 = ...
    [5 6;
    7 8];
my3Dtable = zeros(2, 2, 2);
my3Dtable(:,1,:) = ...
    [1 5;
    3 7];

my3Dtable(:,2,:) = ...
    [2 6;
    4 8];
```

Save Variables Matching a Regular Expression

Save variables that match the expression `autoL*` to a MATLAB script `autoVariables.m`.

```
matlab.io.saveVariablesToScript('autoVariables.m','RegExp','autoL*')
```

Save Variables to Version 7.3 MATLAB Script

Create two variables, `p` and `q`, and save them to a version 7.3 MATLAB script `version73.m`.

```
p = 49;
q = 35.5;
matlab.io.saveVariablesToScript('version73.m','p','q',...
'MATFileVersion','v7.3')
```

Return Variables Saved to MATLAB Script

Save variables that were saved to a MATLAB script to the variable `r1`, and those that were saved to a MAT-file to the variable `r2`.

```
[r1,r2] = matlab.io.saveVariablesToScript('mydata.m')
```

```
r1 =

    'level1'
    'level2'
    'level3'
    'my3Dtable'


r2 =

   Empty cell array: 0-by-1
```

Input Arguments

`filename` — Name of MATLAB script for saving variables
filename | variable

Name of MATLAB script for saving variables, specified as a string giving a file name or a variable containing the file name.

Example: `matlab.io.saveVariablesToScript('myVariables.m')`

`varnames` — Name of variables to save
string | cell array

Name of variables to save, specified as a string or a cell array.

Example: `{'X','Y','Z'}`

Data Types: `char` | `cell`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'MaximumArraySize',500,'MATFileVersion','v4'` specifies that the maximum number of array elements to save is `500` using MATLAB version 4 syntax.

`MATFileVersion` — MATLAB version whose syntax to use
`'v7.3'` (default) | `'v4'` | `'v6'` | `'v7'`

MATLAB version whose syntax to use for saving MAT-files, specified as the comma-separated pair consisting of `'MATFileVersion'` and one of the following version numbers:

- `'v4'`
- `'v6'`
- `'v7'`
- `'v7.3'`

Example: `'MATFileVersion','v6'`

Data Types: `char`

`MaximumArraySize` — Maximum array elements to save
`1000` (default) | integer

Maximum array elements to save, specified as the comma-separated pair consisting of `'MaximumArraySize'` and an integer in the range of 1 to 10,000.

Example: `'MaximumArraySize',1050`

`MaximumNestingLevel` — Maximum number of object levels or array hierarchies to save
`20` (default) | integer

Maximum number of object levels or array hierarchies to save, specified as the comma-separated pair consisting of `'MaximumNestingLevel'` and an integer in the range of 1 to 200.

Example: `'MaximumNestingLevel',67`

`MaximumTextWidth` — Text wrap width during save
76 (default) | integer

Text wrap width during save, specified as the comma-separated pair consisting of `'MaximumTextWidth'` and an integer in the range of 32 to 256.

Example: `'MaximumTextWidth',82`

`MultidimensionalFormat` — Dimensions of 2-D slices that represent n-D arrays of char, logic, or numeric data
`'rowvector'` (default) | integer cell array

Dimensions of 2-D slices that represent n-D arrays of char, logic, or numeric data, specified as the comma-separated pair consisting of `'MultidimensionalFormat'` and one of these values:

- `'rowvector'` — Save multidimensional variables as a single row vector.
- `integer cell array` — Save a 2-D slice of multidimensional variables, where the dimensions satisfy all the following criteria:
  - Two positive integers represent dimensions.
  - The two integers are less than or equal to the dimensions of the n-D array.
  - The second integer is greater than the first.

Example: `'MultidimensionalFormat',[1,3]`

`RegExp` — Regular expression for matching
string

Regular expression for matching, specified as the comma-separated pair consisting of `'RegExp'` and one or more expressions given as a string.

Example: `'RegExp','level*'`

Data Types: `char`

`SaveMode` — Mode to save MATLAB script
`'create'` (default) | `'update'` | `'append'`

Mode to save MATLAB script, specified as the comma-separated pair consisting of `SaveMode` and one of these values:

- `'create'` — Save variables to a new MATLAB script.
- `'update'` — Only update variables that are already present in a MATLAB script.
- `'append'` — Update variables that are already present in a MATLAB script and append new variables to the end of the script.

Example: `'SaveMode','Update'`

Output Arguments

`r1` — Variables that were saved to a MATLAB script
cell array

Variables that were saved to a MATLAB script, returned as a cell array of variable names.

`r2` — Variables that were saved to a MAT-file
cell array

Variables that were saved to a MAT-file, returned as a cell array of variable names.

Limitations

- `matlab.io.saveVariablesToScript` does not save the following variables to a MATLAB script or a MAT-file.
  - Java objects
  - .NET objects
  - Python objects
- `matlab.io.saveVariablesToScript` saves the following variables only to a MAT-file.
  - MATLAB objects
  - Function handles
  - Anonymous functions

If you have Simulink, you can use `matlab.io.saveVariablesToScript` to save the variables that your models use. However, if you save many variables, the generated MATLAB file can contain many lines of code and take a long time to execute. To avoid the long execution time, consider these alternatives:

- Permanently store variables in a data dictionary instead of using `Simulink.saveVars`. A data dictionary also provides more tools for managing variables. See “Determine Where to Store Variables and Objects for Simulink Models” (Simulink).
- Save variables in a MAT-file by using the `save` function.

See Also

Introduced in R2014a

matlab.lang.makeUniqueStrings

Construct unique strings from input strings

Syntax

```
U = matlab.lang.makeUniqueStrings(S)

U = matlab.lang.makeUniqueStrings(S,excludedStrings)
U = matlab.lang.makeUniqueStrings(S,whichStringsIdx)
U = matlab.lang.makeUniqueStrings(S, ___ , maxStringLength)

[U, modified] = matlab.lang.makeUniqueStrings( ___ )
```

Description

`U = matlab.lang.makeUniqueStrings(S)` constructs unique strings or character vectors, `U`, from input strings or character vectors, `S`, by appending an underscore and a number to duplicates.

`U = matlab.lang.makeUniqueStrings(S,excludedStrings)` constructs strings or character vectors that are unique within `U` and with respect to `excludedStrings`. The `makeUniqueStrings` function does not check `excludedStrings` for uniqueness.

`U = matlab.lang.makeUniqueStrings(S,whichStringsIdx)` specifies the subset of `S` to make unique within the entire set. `makeUniqueStrings` makes the elements in `S(whichStringsIdx)` unique among themselves and with respect to the remaining elements. `makeUniqueStrings` returns the remaining elements unmodified in `U`. Use this syntax when you have an string array or an array of character vectors, and need to check that only some elements are unique.

`U = matlab.lang.makeUniqueStrings(S, ___ , maxStringLength)` specifies the maximum length, `maxStringLength`, of elements in `U`. If `makeUniqueStrings` cannot make elements in `S` unique without exceeding `maxStringLength`, it returns an error. You can use this syntax with any of the input arguments of the previous syntaxes.

`[U, modified] = matlab.lang.makeUniqueStrings( ___ )` returns a logical array, `modified`, indicating the modified elements.

Examples

Construct Unique Character Vectors

Create a cell array of names and make each element unique.

```
S = {'John' 'Sue' 'Nick' 'John' 'Campion' 'John' 'Jason'};
U = matlab.lang.makeUniqueStrings(S)
```

```
U = 1x7 cell array
  Columns 1 through 6

    {'John'}    {'Sue'}    {'Nick'}    {'John_1'}    {'Campion'}    {'John_2'}

  Column 7

    {'Jason'}
```

The `makeUniqueStrings` function appends the duplicate names in elements 3 and 5 with underscores and incrementing numbers.

Construct Unique Character Vectors and Specify Exclusions

Without specifying excluded values, make the character vectors in `U` unique.

```
S = {'John' 'Sue' 'Nick' 'John' 'Campion' 'John' 'Jason'};
U = matlab.lang.makeUniqueStrings(S)
```

```
U = 1x7 cell array
  Columns 1 through 6

    {'John'}    {'Sue'}    {'Nick'}    {'John_1'}    {'Campion'}    {'John_2'}

  Column 7
```

```
    {'Jason'}
```

Specify that the character vector, `'Nick'`, should be excluded from the output.

```
U = matlab.lang.makeUniqueStrings(S, 'Nick')
```

```
U = 1x7 cell array
  Columns 1 through 5

    {'John'}    {'Sue'}    {'Nick_1'}    {'John_1'}    {'Campion'}

  Columns 6 through 7

    {'John_2'}    {'Jason'}
```

`makeUniqueStrings` excludes `'Nick'` from `U` and instead modifies the first duplicate, found in element 3, to be `'Nick_1'`.

Exclude workspace variables from the unique cell array.

```
Sue = 42;
U = matlab.lang.makeUniqueStrings(S, who)
```

```
U = 1x7 cell array
  Columns 1 through 5

    {'John'}    {'Sue_1'}    {'Nick'}    {'John_1'}    {'Campion'}

  Columns 6 through 7

    {'John_2'}    {'Jason'}
```

Since `'Sue'` exists in the workspace, `makeUniqueStrings` makes this character vector unique by appending an underscore and number.

Construct Unique Elements for Specified Array Indices

Create an array of character vectors and make only the first four elements unique.

```
S = {'quiz' 'quiz' 'quiz' 'exam' 'quiz' 'exam'};
U = matlab.lang.makeUniqueStrings(S, 1:4)
```

```
U = 1x6 cell array
  Columns 1 through 5

    {'quiz_1'}    {'quiz_2'}    {'quiz_3'}    {'exam_1'}    {'quiz'}

  Column 6

    {'exam'}
```

The first four elements in `U` are unique among themselves, and among the remaining character vectors in elements 5 and 6 (`'quiz'` and `'exam'`). Alternatively, you can use a logical array instead of a range of linear indices to achieve the same results: `U = matlab.lang.makeUniqueStrings(S, [true true true true false false])` or `U = matlab.lang.makeUniqueStrings(S, logical([1 1 1 1 0 0]))`.

Append a duplicate `'quiz'` onto the end of `S` and make the first four elements unique.

```
S{end+1} = 'quiz'
```

```
S = 1x7 cell array
  Columns 1 through 6

    {'quiz'}    {'quiz'}    {'quiz'}    {'exam'}    {'quiz'}    {'exam'}

  Column 7

    {'quiz'}
```

```
U = matlab.lang.makeUniqueStrings(S, 1:4)
```

```
U = 1x7 cell array
  Columns 1 through 5

    {'quiz_1'}    {'quiz_2'}    {'quiz_3'}    {'exam_1'}    {'quiz'}

  Columns 6 through 7

    {'exam'}    {'quiz'}
```

The character vectors that `makeUniqueStrings` checks are still unique among themselves and among the remaining elements. Since `makeUniqueStrings` does not check any elements after element 4, duplicate character vectors remain.

Construct Unique Character Vectors with Maximum Length

Create an array from `S` where the first three elements are unique and the maximum length of each string is 5.

```
S = {'sampleData' 'sampleData' 'sampleData' 'sampleData'};
U = matlab.lang.makeUniqueStrings(S, 1:3, 5)
```

```
U = 1x4 cell array
    {'sampl'}    {'sam_1'}    {'sam_2'}    {'sampleData'}
```

The first element is truncated to 5 characters. The second and third elements are truncated to 3 characters to allow `makeUniqueStrings` to append an underscore and number, and still not exceed 5 characters.

Determine Modified Character Vectors

```
S = {'a%name', 'name_1', '2_name'};
[N, modified] = matlab.lang.makeValidName(S)
```

```
N = 1x3 cell array
    {'a_name'}    {'name_1'}    {'x2_name'}


modified = 1x3 logical array

   1   0   1
```

`makeValidName` did not modify the second element.

Input Arguments

S — Input values
character vector | cell array of character vectors | string array

Input strings, specified as a character vector, cell array of character vectors, or string array.

excludedStrings — Strings to exclude
character vector | cell array of character vectors | string array

Character vectors to exclude from U, specified as a character vector, cell array of character vectors, or string array.

Example: `'dontDuplicateThis'`

Example: `{'excludeS1' 'excludeS2'}`

Example: `["excludeThis" "andThis"]`

Example: `who`

whichStringsIdx — Subset of strings to make unique
range of linear indices or logical array

Subset of Sto make unique within the entire set, specified as a range of linear indices or as a logical array with the same size and shape as S. If there are duplicates in S, the `makeUniqueStrings` function only modifies those specified by `whichStringsIdx`.

If `whichStringsIdx` is a logical array, elements are checked for uniqueness when the array element in the same position has a value of `true`.

Example: `1:5`, `logical([1 0 1])`, `[true false true]`

maxStringLength — Maximum length of strings
integer

Maximum length of strings in U, specified as an integer. If `makeUniqueStrings` cannot make elements in S unique without exceeding `maxStringLength`, it returns an error.

Output Arguments

U — Unique strings
character vector | cell array of character vectors | string array

Unique strings, returned as a character vector, cell array of character vectors, or string array. The output has the same dimension as the input, S.

modified — Indicator of modified elements
logical scalar | logical array

Indicator of modified elements, returned as a logical scalar or array and having the same dimension as the input, S. A value of 1 (true) indicates that makeUniqueStrings modified the element in the corresponding location. A value of 0 (false) indicates that makeUniqueStrings did not need to modify the element in the corresponding location.

Tips

- To ensure that input values are valid and unique, use matlab.lang.makeValidName before matlab.lang.makeUniqueStrings.

```
S = {'my.Name','my_Name','my_Name'};
validValues = matlab.lang.makeValidName(S)
validUniqueValues = matlab.lang.makeUniqueStrings(validValues,...
    {},namelengthmax)
```

```
validValues =

    'my_Name'    'my_Name'    'my_Name'


validUniqueValues =

    'my_Name'    'my_Name_1'    'my_Name_2'
```

See Also

matlab.lang.makeValidName | namelengthmax | who

Introduced in R2014a

matlab.lang.makeValidName

Construct valid MATLAB identifiers from input strings

Syntax

```
N = matlab.lang.makeValidName(S)
N = matlab.lang.makeValidName(S,Name,Value)
[N, modified] = matlab.lang.makeValidName( ___ )
```

Description

`N = matlab.lang.makeValidName(S)` constructs valid MATLAB identifiers, `N`, from input strings, `S`. The `makeValidName` function does not guarantee the strings in `N` are unique.

A valid MATLAB identifier is a character vector of alphanumerics (A–Z, a–z, 0–9) and underscores, such that the first character is a letter and the length of the character vector is less than or equal to `namelengthmax`.

`makeValidName` deletes any whitespace characters before replacing any characters that are not alphanumerics or underscores. If a whitespace character is followed by a lowercase letter, `makeValidName` converts the letter to the corresponding uppercase character.

`N = matlab.lang.makeValidName(S,Name,Value)` includes additional options specified by one or more `Name,Value` pair arguments.

`[N, modified] = matlab.lang.makeValidName( ___ )` returns a logical array, `modified`, indicating modified elements. You can use this syntax with any of the input arguments of the previous syntaxes.

Examples

Construct Valid MATLAB Identifiers

```
S = {'Item_#','Price/Unit','1st order','Contact'};
N = matlab.lang.makeValidName(S)
```

```
N = 1x4 cell array
    {'Item__'}    {'Price_Unit'}    {'x1stOrder'}    {'Contact'}
```

In the first and second elements, `makeValidName` replaced the invalid characters (`#` and `/`), with underscores. In the third element, `makeValidName` appended a prefix because the character vector does not begin with a letter, deleted the empty space, and capitalized the character following the deleted space.

Construct Valid MATLAB Identifiers Using Specified Replacement Style

Replace invalid characters with the corresponding hexadecimal representation.

```
S = {'Item_#','Price/Unit','1st order','Contact'};
N = matlab.lang.makeValidName(S,'ReplacementStyle','hex')
```

```
N = 1x4 cell array
    {'Item_0x23'}    {'Price0x2FUnit'}    {'x1stOrder'}    {'Contact'}
```

In the first and second elements, `makeValidName` replaced the invalid characters (`#` and `/`), with their hexadecimal representation. In the third element, `makeValidName` appended a prefix because the character vector does not begin with a letter, deleted the empty space, and capitalized the character following the deleted space.

Delete invalid characters.

```
N = matlab.lang.makeValidName(S,'ReplacementStyle','delete')
```

```
N = 1x4 cell array
    {'Item_'}    {'PriceUnit'}    {'x1stOrder'}    {'Contact'}
```

`makeValidName` deleted the invalid characters (`#` and `/`). In the third element, `makeValidName` appended a prefix because the character vector does not begin with a letter, deleted the empty space, and capitalized the character following the deleted space.

Construct Valid MATLAB Identifiers Using Specified Prefix

```
S = {'1stMeasurement','2ndMeasurement','Control'};
N = matlab.lang.makeValidName(S,'Prefix','m_')
```

```
N = 1x3 cell array
    {'m_1stMeasurement'}    {'m_2ndMeasurement'}    {'Control'}
```

Only the elements that do not start with a letter are prepended with a prefix.

Determine Modified Character Vectors

```
S = {'a%name', 'name_1', '2_name'};
[N, modified] = matlab.lang.makeValidName(S)
```

```
N = 1x3 cell array
    {'a_name'}    {'name_1'}    {'x2_name'}


modified = 1x3 logical array

   1   0   1
```

`makeValidName` did not modify the second element.

Input Arguments

S — Input strings
character vector | cell array of character vectors | string array

Input strings, specified as a character vector, cell array of character vectors, or string array.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'ReplacementStyle','delete'` deletes invalid characters.

`ReplacementStyle` — Replacement style
`'underscore'` (default) | `'delete'` | `'hex'`

Replacement style, specified as `'underscore'`, `'delete'`, or `'hex'`. The value controls how MATLAB replaces nonalphanumeric characters. For all values of `ReplacementStyle`, MATLAB deletes whitespace characters and changes a lowercase letter following a whitespace to uppercase.

`ReplacementStyle` Value	Description
`'underscore'` (default)	Replaces all characters that are not alphanumerics or underscores with underscores. `'underscore'`.
`'hex'`	Replaces each character that is not an alphanumeric or underscore with its corresponding hexadecimal representation. `'hex'`.
`'delete'`	Deletes all characters that are not alphanumerics or underscores. `'delete'`.

`Prefix` — Characters to prefix
`'x'` (default) | character vector | string scalar

Characters to prefix to inputs that do not begin with a letter after `makeValidName` replaces nonalphanumeric characters, specified as a character vector or string scalar. For example, by default, `makeValidName` prefixes characters to an input `'*hello'` because, after it replaces nonalphanumeric characters, the input does not begin with a letter (`'_hello'`). However, if you specify a replacement style that deletes nonalphanumeric characters, `makeValidName` does not prefix characters. After it replaces nonalphanumeric characters, the input begins with a letter (`'hello'`).

A valid prefix must meet the following conditions.

- Start with a letter.
- Contain only alphanumeric characters and underscores.
- Not be a MATLAB keyword.
- Not be longer than the value of `namelengthmax`.

Output Arguments

`N` — Valid MATLAB identifiers
character vector | cell array of character vectors | string array

Valid MATLAB identifiers, returned as a character vector, cell array of character vectors, or string array. The output has the same number of dimensions as the input, `S`.

`modified` — Indicator of modified elements
logical scalar | logical array

Indicator of modified elements, returned as a logical scalar or array and having the same number of dimensions as the input, `S`. A value of `1` (`true`) indicates that `makeValidName` modified the input in the corresponding location. A value of `0` (`false`) indicates that `makeValidName` did not need to modify the input in the corresponding location.

Tips

- To ensure that input values are valid and unique, use `matlab.lang.makeUniqueStrings` after `matlab.lang.makeValidName`.

```
S = {'my.Name','my_Name','my_Name'};
validValues = matlab.lang.makeValidName(S)
validUniqueValues = matlab.lang.makeUniqueStrings(validValues,{},...
    namelengthmax)
```

```
validValues =

    'my_Name'    'my_Name'    'my_Name'


validUniqueValues =

    'my_Name'    'my_Name_1'    'my_Name_2'
```

- To customize an invalid character replacement, first use functions such as `strrep` or `regexprep` to convert to valid characters. For example, convert `'@'` characters in `S` to `'At'` using `strrep(S,'@','At')`. Then, use `matlab.lang.makeValidName` to ensure that all characters in `S` are valid.

See Also

`iskeyword` | `isletter` | `isvarname` | `matlab.lang.makeUniqueStrings` | `namelengthmax` | `regexp` | `regexprep` | `strrep` | `who`

Introduced in R2014a

matlab.mock.actions.AssignOutputs class

Package: `matlab.mock.actions`

Define return values for method called or property accessed

Description

To specify the values that the framework returns when a mock object method is invoked or a mock object property is accessed, use the `AssignOutputs` class.

If you call a mocked method with fewer outputs than specified by `AssignOutputs`, the remaining outputs are discarded. If you call a mocked method with more outputs than specified by `AssignOutputs`, MATLAB throws an error.

Construction

`action = AssignOutputs(A1,...,An)` defines the values for the mock object to return for a method call or property access. The `AssignOutputs` constructor accepts one or more values, `A1,...,An`, that correspond to the values for the output arguments of the mock object method or the value of the mock object property.

Input Arguments

`A1,...,An` — Return values
scalar | vector | matrix | multidimensional array

Return values, specified as scalars, vectors, matrices, or multidimensional arrays. Return values can be any data type.

Example: `"mySpecifiedValue"`

Example: `7,13,42`

Example: `[1 2 3;4 5 6]`

Properties

`Outputs` — Predefined return values
cell array

Predefined return values, returned as a cell array of arguments. Return values are specified during class construction, and can be any data type.

Data Types: `cell`

Methods

repeat	Repeat same action multiple times
then	Specify action

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Assign Outputs to Mock

Create a mock for a quadrilateral class.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedMethods',"sideLengths");
```

Set up behavior.

```
import matlab.mock.actions.AssignOutputs;
when(withAnyInputs(behavior.sideLengths),AssignOutputs(2,2,4,4));
```

Use the mock.

```
[a,b,c,d] = mock.sideLengths
```

Tips

- The `AssignOutputs` action defines the value of the output at instance creation time. To specify that a mock object method invoke a function handle to determine output values when a mock object method is called, use the `Invoke` class.

Alternatives

For simple definition of outputs, you can use the `assignOutputsWhen` method of the `matlab.mock.TestCase` class. However, there is more functionality when you use the `AssignOutputs` action.

See Also

`matlab.mock.TestCase` | `matlab.mock.actions.Invoke` | `matlab.mock.actions.ThrowException` | `when` | `when`

Introduced in R2017a

repeat

Class: `matlab.mock.actions.AssignOutputs`
Package: `matlab.mock.actions`

Repeat same action multiple times

Syntax

```
repeat(n,action)
```

Description

`repeat(n,action)` repeats the same action multiple times.

Input Arguments

`n` — Times to repeat
integer

Number of times to repeat action, specified as an integer.

Example: 5

`action` — Defined action
instance of `matlab.mock.actions.AssignOutputs`

Defined action, specified as an instance of `matlab.mock.actions.AssignOutputs`.

Example: `AssignOutputs(true)`

Example: `AssignOutputs(7,13,42)`

Examples

Assign Repeating Outputs to Mock

Create a mock for a bank account class.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedMethods',"isOpen");
```

Specify behavior.

```
import matlab.mock.actions.AssignOutputs
when(withExactInputs(behavior.isOpen),then(repeat(2,AssignOutputs(true)), ...
    then(AssignOutputs(false))));
```

Use the mock.

```
for i = 1:3
    isAccountOpen = mock.isOpen
end
```

```
isAccountOpen = logical
   1


isAccountOpen = logical
   1


isAccountOpen = logical
   0
```

Tips

- If you repeat an action, and do not follow it with a call to the `then` method, the mock continues to return the repeated value. For example, consider the following mock of a bank account class.

  ```
  import matlab.mock.actions.AssignOutputs
  testCase = matlab.mock.TestCase.forInteractiveUse;
  [mock,behavior] = testCase.createMock('AddedProperties',"IsJointAccount");
  ```

If you repeat an action to return a property value of `true` twice, the following code, which goes on to get the property value a third and fourth time, returns `true` all four times.

```
when(get(behavior.IsJointAccount),then(repeat(2,AssignOutputs(true))));
for i = 1:4
    tf = mock.IsJointAccount
end
```

But the following code returns `true` twice and `false` twice.

```
when(get(behavior.IsJointAccount),then(repeat(2,AssignOutputs(true)), ...
    then(AssignOutputs(false))));
for i = 1:4
    tf = mock.IsJointAccount
end
```

See Also

Introduced in R2017a

then

Class: `matlab.mock.actions.AssignOutputs`
Package: `matlab.mock.actions`

Specify action

Syntax

```
then(action1)
then(action1,action2)
```

Description

`then(action1)` specifies an action for mock object interactions.

`then(action1,action2)` specifies an action and a subsequent action for mock object interactions.

Input Arguments

`action1` — Defined action
instance of `matlab.mock.actions.AssignOutputs`

Defined action, specified as an instance of `matlab.mock.actions.AssignOutputs`.

Example: `AssignOutputs(true)`

Example: `AssignOutputs(7,13,42)`

`action2` — Second defined action
instance of `matlab.mock.actions.AssignOutputs` | instance of `matlab.mock.actions.Invoke` | instance of `matlab.mock.actions.ThrowException`

Second defined action, specified as an instance of `matlab.mock.actions.AssignOutputs`, `matlab.mock.actions.Invoke`, or `matlab.mock.actions.ThrowException`.

Example: `AssignOutputs(7,13,42)`

Example: `ThrowException`

Examples

Assign Series of Outputs to Mock

Create a mock for a bank account class.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedMethods',"isOpen");
```

Set up behavior to return `true`, `true`, and then `false`.

```
import matlab.mock.actions.AssignOutputs;
when(withExactInputs(behavior.isOpen),then(AssignOutputs(true), ...
    then(AssignOutputs(true),then(AssignOutputs(false)))));
```

Use the mock.

```
isAccountOpen = mock.isOpen
```

```
isAccountOpen = logical
   1
```

```
isAccountOpen = mock.isOpen
```

```
isAccountOpen = logical
   1
```

```
isAccountOpen = mock.isOpen
```

```
isAccountOpen = logical
   0
```

Tips

- Each call to `then` accepts up to two actions. To specify more subsequent actions, use multiple calls to `then`. For example, to specify three actions use `then(action1,then(action2,action3))`.

See Also

Introduced in R2017a

matlab.mock.actions.Invoke class

Package: `matlab.mock.actions`

Invoke function handle when method is called

Description

To specify that the framework invokes a function handle to determine outputs when a mock object method is called, use the `Invoke` class. This action differs from the `AssignOutputs` action, which returns values that are defined when you create the `AssignOutputs` instance.

Construction

`action = invoke(fh)` invokes the function specified by `fh` when a method is called.

Input Arguments

`fh` — Function to invoke
function handle

Function to invoke when a mock object method is called, specified as a function handle.

The framework passes the function handle the same inputs as it passes to the mock method. Therefore, `fh` typically contains an argument list. If the function does not interact with the mock, for example by accessing a property, you can use tilde (~) in the argument list to ignore the mock object. The framework requests the same number of outputs as the mock method call.

Example: `@(~)randi(6)`

Example: `@(cmock)myFunction('hello')`

Example: `@isempty`

Properties

`Function` — Function to invoke
function handle

Function to invoke when a mock object method is called, stored as a function handle.

Methods

repeat	Repeat same action multiple times
then	Specify action

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Invoke Function Handle at Method Call Time

Create a mock for a class that represents a 6-sided die and includes a mocked `roll` method.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock("AddedMethods","roll");
```

Define behavior to return a random integer 1 through 6 each time the `roll` method is called.

```
import matlab.mock.actions.Invoke
when(withExactInputs(behavior.roll),Invoke(@(~)randi(6)));
```

Call the mocked `roll` method.

```
val = mock.roll
```

```
val = 5
```

See Also

`matlab.mock.TestCase` | `matlab.mock.actions.AssignOutputs` | `when`

Introduced in R2018b

repeat

Class: `matlab.mock.actions.Invoke`
Package: `matlab.mock.actions`

Repeat same action multiple times

Syntax

```
repeat(n,action)
```

Description

`repeat(n,action)` repeats the same action multiple times.

Input Arguments

`n` — Times to repeat
integer

Number of times to repeat action, specified as an integer.

Example: 5

`action` — Defined action
instance of `matlab.mock.actions.Invoke`

Defined action, specified as an instance of `matlab.mock.actions.Invoke`.

Example: `Invoke(@isempty)`

Example: `Invoke(@(~)randi(10))`

Examples

Assign Repeating Outputs to Mock

Create a mock for a class that represents a 12-sided die, including a mocked `roll` method.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock("AddedMethods","roll");
```

Set up behavior for the `roll` method to return the value from the `randi` function twice and then return 0.

```
import matlab.mock.actions.AssignOutputs
import matlab.mock.actions.Invoke

when(withExactInputs(behavior.roll),...
    then(repeat(2,Invoke(@(~)randi(12))),...
        then(AssignOutputs(0))));
```

Call the mocked `roll` method four times.

```
val = mock.roll
```

```
val = 10
```

```
val = mock.roll
```

```
val = 11
```

```
val = mock.roll
```

```
val = 0
```

```
val = mock.roll
```

```
val = 0
```

See Also

`matlab.mock.actions.Invoke`

Introduced in R2018b

then

Class: `matlab.mock.actions.Invoke`
Package: `matlab.mock.actions`

Specify action

Syntax

```
then(action1)
then(action1,action2)
```

Description

`then(action1)` specifies an action for mock object interactions.

`then(action1,action2)` specifies an action and a subsequent action for mock object interactions.

Input Arguments

`action1` — Defined action
instance of `matlab.mock.actions.Invoke`

Defined action, specified as an instance of `matlab.mock.actions.Invoke`.

`action2` — Second defined action
instance of `matlab.mock.actions.Invoke` | instance of `matlab.mock.actions.AssignOutputs` | instance of `matlab.mock.actions.ThrowException`

Second defined action, specified as an instance of `matlab.mock.actions.Invoke`, `matlab.mock.actions.AssignOutputs`, or `matlab.mock.actions.ThrowException`.

Examples

Assign Series of Outputs to Mock

Create a mock for a class that represents a 6-sided die, including a mocked `roll` method.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedMethods',"roll");
```

Set up behavior for the `roll` method. Return 0 the first time it runs. On subsequent runs, invoke the `randi` function to return a random integer between 1 and 6.

```
import matlab.mock.actions.AssignOutputs
import matlab.mock.actions.Invoke
when(withExactInputs(behavior.roll),...
    then(AssignOutputs(0),...
        then(Invoke(@(~)randi(6)))))
```

Call the mocked `roll` method three times.

```
val = mock.roll
```

```
val = 0
```

```
val = mock.roll
```

```
val = 5
```

```
val = mock.roll
```

```
val = 6
```

Tips

- Each call to `then` accepts up to two actions. To specify more subsequent actions, use multiple calls to `then`. For example, to specify three actions use `then(action1,then(action2,action3))`.

See Also

`matlab.mock.actions.Invoke`

Introduced in R2018b

matlab.mock.actions.ReturnStoredValue class

Package: `matlab.mock.actions`

Return stored property value

Description

The `ReturnStoredValue` action specifies that the stored value is returned when accessing a property.

By default, when you access property values in strict mocks, MATLAB throws an exception. To specify that the framework returns the stored value of a mock object property, use the `ReturnStoredValue` class.

Construction

`action = ReturnStoredValue` returns the stored property value.

Methods

repeat	Repeat same action multiple times
then	Specify action

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Return Mock Property Value

Create a strict mock. By default, property interactions throw exceptions.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
properties = ["PropA" "PropB" "PropC"];
[mock,behavior] = testCase.createMock('AddedProperties',properties,"Strict",true);
```

Enable access to `PropA`.

```
import matlab.mock.actions.ReturnStoredValue;
when(get(behavior.PropA),ReturnStoredValue);
```

Use the mock.

```
val1 = mock.PropA
```

```
val1 =

     []
```

```
val2 = mock.PropB
```

```
Interactive assertion failed.

----------------
Test Diagnostic:
----------------
Unexpected access of strict mock property 'PropB'.
Assertion failed.
```

Alternatives

For a simple directive to return property values, you can use the `returnStoredValueWhen` method of the `matlab.mock.TestCase` class. However, there is more functionality when you use the `ReturnStoredValue` action.

See Also

`matlab.mock.TestCase` | `matlab.mock.actions.AssignOutputs` | `matlab.mock.actions.StoreValue` | `matlab.mock.actions.ThrowException` | `when`

Introduced in R2017a

repeat

Class: `matlab.mock.actions.ReturnStoredValue`
Package: `matlab.mock.actions`

Repeat same action multiple times

Syntax

```
repeat(n,action)
```

Description

`repeat(n,action)` repeats the same action multiple times.

Input Arguments

`n` — Times to repeat
integer

Number of times to repeat action, specified as an integer.

Example: 5

`action` — Defined action
instance of `matlab.mock.actions.ReturnStoredValue`

Defined action, specified as an instance of `matlab.mock.actions.ReturnStoredValue`.

Examples

Repeat Return of Stored Property Value

Create a strict mock for a bank account class.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties',"isOpen",'Strict',true);
```

Specify behavior. Return the stored property value, which is empty by default, twice and then assign an output of `false`.

```
import matlab.mock.actions.ReturnStoredValue
import matlab.mock.actions.AssignOutputs
when(get(behavior.isOpen),then(repeat(2,ReturnStoredValue), ...
    then(AssignOutputs(false))));
```

Use the mock.

```
for i = 1:3
    isOpen = mock.isOpen
end
```

```
isOpen =

     []


isOpen =

     []

isOpen = logical
   0
```

See Also

Introduced in R2017a

then

Class: `matlab.mock.actions.ReturnStoredValue`
Package: `matlab.mock.actions`

Specify action

Syntax

```
then(action1)
then(action1,action2)
```

Description

`then(action1)` specifies an action for mock object interactions.

`then(action1,action2)` specifies an action and a subsequent action for mock object interactions.

Input Arguments

`action1` — Defined action
instance of `matlab.mock.actions.ReturnStoredValue`

Defined action, specified as an instance of `matlab.mock.actions.ReturnStoredValue`.

`action2` — Second defined action
instance of `matlab.mock.actions.AssignOutputs` | instance of `matlab.mock.actions.ThrowException`

Second defined action, specified as an instance of `matlab.mock.actions.AssignOutputs`, or `matlab.mock.actions.ThrowException`.

Example: `ReturnStoredValue`

Example: `AssignOutputs(7,13,42)`

Example: `ThrowException`

Examples

Assign Series of Outputs to Mock

Create a strict mock for a bank account class.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties',"isOpen",'Strict',true);
```

Set up behavior to return empty once and then `true`.

```
import matlab.mock.actions.ReturnStoredValue
import matlab.mock.actions.AssignOutputs
when(get(behavior.isOpen),then(ReturnStoredValue,AssignOutputs(true)))
```

Use the mock.

```
isAccountOpen = mock.isOpen
```

```
isAccountOpen =

     []
```

```
isAccountOpen = mock.isOpen
```

```
isAccountOpen = logical
   1


isAccountOpen = mock.isOpen
```

```
isAccountOpen = logical
   1
```

Tips

- Each call to `then` accepts up to two actions. To specify more subsequent actions, use multiple calls to `then`. For example, to specify three actions use `then(action1,then(action2,action3))`.

See Also

Introduced in R2017a

matlab.mock.actions.StoreValue class

Package: `matlab.mock.actions`

Store property value

Description

The `StoreValue` action specifies that the specified value is stored when setting a property.

By default, when you set property values in strict mocks, MATLAB throws an exception. To specify that the framework stores the value of a mock object property, use the `StoreValue` class.

Construction

`action = StoreValue` stores a property value.

Methods

repeat	Repeat same action multiple times
then	Specify action

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Store Mock Property Value

Create a strict mock. By default, property interactions throw exceptions.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
properties = ["PropA" "PropB" "PropC"];
[mock, behavior] = testCase.createMock('AddedProperties',properties,"Strict",true);
```

Enable set access to `PropA`.

```
import matlab.mock.actions.StoreValue;
when(set(behavior.PropA),StoreValue);
```

Use the mock.

```
mock.PropA = 1;
```

Use the mock.

```
mock.PropB = 2;
```

```
Interactive assertion failed.

----------------
Test Diagnostic:
----------------
Unexpected set of strict mock property 'PropB'.
Assertion failed.
```

Alternatives

For a simple directive to store property values, you can use the `storeValueWhen` method of the `matlab.mock.TestCase` class. However, there is more functionality when you use the `StoreValue` action.

See Also

`matlab.mock.TestCase` | `matlab.mock.actions.ReturnStoredValue` | `matlab.mock.actions.ThrowException` | `when`

Introduced in R2017a

repeat

Class: `matlab.mock.actions.StoreValue`
Package: `matlab.mock.actions`

Repeat same action multiple times

Syntax

```
repeat(n,action)
```

Description

`repeat(n,action)` repeats the same action multiple times.

Input Arguments

`n` — Times to repeat
integer

Number of times to repeat action, specified as an integer.

Example: 5

`action` — Defined action
instance of `matlab.mock.actions.StoreValue`

Defined action, specified as an instance of `matlab.mock.actions.StoreValue`.

Examples

Repeat Storing of Property Value

Create a mock for a bank account class.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties',"IsOpen");
```

Specify behavior. Allow the `IsOpen` property to be set twice, then throw an exception.

```
import matlab.mock.actions.StoreValue
import matlab.mock.actions.ThrowException
when(set(behavior.IsOpen),then(repeat(2,StoreValue), ...
    then(ThrowException(MException('Account:setValue:tooMany', ...
    'Value set too many times.')))));
```

Use the mock.

```
for i = 1:3
    mock.IsOpen = i
end
```

```
mock =

  Mock with properties:

    IsOpen: 1


mock =

  Mock with properties:

    IsOpen: 2

Error using matlab.mock.internal.MockContext>mockPropertySetCallback (line 706)
Value set too many times.

Error in matlab.mock.internal.MockContext>@(name,obj,value)mockPropertySetCallback(name
(line 284)
            propertySetCallback = @(name, obj, value)mockPropertySetCallback(name, obj,
```

See Also

Introduced in R2017a

then

Class: `matlab.mock.actions.StoreValue`
Package: `matlab.mock.actions`

Specify action

Syntax

```
then(action1)
then(action1,action2)
```

Description

`then(action1)` specifies an action for mock object interactions.

`then(action1,action2)` specifies an action and a subsequent action for mock object interactions.

Input Arguments

`action1` — Defined action
instance of `matlab.mock.actions.StoreValue`

Defined action, specified as an instance of `matlab.mock.actions.StoreValue`.

`action2` — Second defined action
instance of `matlab.mock.actions.ThrowException`

Second defined action, specified as an instance of `matlab.mock.actions.ThrowException`.

Example: `StoreValue`

Example: `ThrowException(MException('Account:deposit:Negative','Deposit amount must be positive.'))`

Examples

Store Series of Values to Mock Property

Create a mock for a bank account class.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties',"IsOpen");
```

Specify behavior. Allow the `IsOpen` property to be set once, and then throw an exception.

```
import matlab.mock.actions.StoreValue
import matlab.mock.actions.ThrowException
when(set(behavior.IsOpen),then(StoreValue,ThrowException( ...
    MException('Account:setValue:tooMany','Value set too many times.'))));
```

Use the mock.

```
mock.IsOpen = false
```

```
mock =

  Mock with properties:

    IsOpen: 0
```

```
mock.IsOpen = true
```

```
Error using matlab.mock.internal.MockContext>mockPropertySetCallback (line 706)
Value set too many times.

Error in matlab.mock.internal.MockContext>@(name,obj,value)mockPropertySetCallback(name
(line 284)
            propertySetCallback = @(name, obj, value)mockPropertySetCallback(name, obj
```

Tips

- Each call to `then` accepts up to two actions. To specify more subsequent actions, use multiple calls to `then`. For example, to specify three actions use `then(action1,then(action2,action3))`.

See Also

Introduced in R2017a

matlab.mock.actions.ThrowException class

Package: `matlab.mock.actions`

Throw exception when method is called or when property is set or accessed

Description

To specify that the framework throws an exception when a mock object method is invoked or when a mock object property is set or accessed, use the `ThrowException` class. You can use this action to inject error conditions into the system under test.

Construction

`action = ThrowException` throws an exception when a method is called or a property is set or accessed.

`action = ThrowException(exception)` specifies the exception that the mock throws.

Input Arguments

`exception` — Exception to throw
scalar `MException` object

Exception for the framework to throw at method call or property interaction, specified as a scalar `MException` object.

Example: `MException('MyProduct:myID','My exception message.')`

Methods

repeat	Repeat same action multiple times
then	Specify action

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Throw Exception for Particular Mock Method Inputs

Create a mock for a bank account class.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedMethods',"deposit");
```

Set up behavior.

```
import matlab.mock.actions.ThrowException;
import matlab.unittest.constraints.IsLessThan;
when(behavior.deposit(IsLessThan(0)), ...
    ThrowException(MException('Account:deposit:Negative', ...
    'Deposit amount must be positive.')));
```

Use the mock.

```
mock.deposit(10)
mock.deposit(-10)
```

```
Error using matlab.mock.internal.MockContext>mockMethodCallback (line 663)
Deposit amount must be positive.

Error in matlab.mock.internal.MockContext>@(data)mockMethodCallback(data,catalog) (line
            methodCallback = @(data)mockMethodCallback(data, catalog);
```

Alternatives

For simple definition of outputs, you can use the `throwExceptionWhen` method of the `matlab.mock.TestCase` class. However, there is more functionality when you use the `ThrowException` action.

See Also

`matlab.mock.TestCase` | `matlab.mock.actions.AssignOutputs` | `when` | `when` | `when`

Introduced in R2017a

repeat

Class: `matlab.mock.actions.ThrowException`
Package: `matlab.mock.actions`

Repeat same action multiple times

Syntax

```
repeat(n,action)
```

Description

`repeat(n,action)` repeats the same action multiple times.

Input Arguments

n — Times to repeat
integer

Number of times to repeat action, specified as an integer.

Example: 5

action — Defined action
instance of `matlab.mock.actions.ThrowException`

Defined action, specified as an instance of `matlab.mock.actions.ThrowException`.

Example: `ThrowException`

Example: `ThrowException(MException('Account:deposit:Negative','Deposit amount must be positive.'))`

Examples

Throwing Repeated Exceptions for Mock Method Calls

Create a mock for a bank account class.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedMethods',"isOpen");
```

Specify behavior.

```
import matlab.mock.actions.ThrowException
import matlab.mock.actions.AssignOutputs
when(withExactInputs(behavior.isOpen),then(repeat(2,ThrowException), ...
    then(AssignOutputs(false))));
```

Use the mock.

```
isAccountOpen = mock.isOpen
```

```
Error using matlab.mock.internal.MockContext>mockMethodCallback (line 663)
The following method call was specified to throw an exception:
    isOpen([1×1 matlab.mock.classes.Mock])

Error in matlab.mock.internal.MockContext>@(data)mockMethodCallback(data,catalog) (line
            methodCallback = @(data)mockMethodCallback(data, catalog);
```

```
isAccountOpen = mock.isOpen
```

```
Error using matlab.mock.internal.MockContext>mockMethodCallback (line 663)
The following method call was specified to throw an exception:
    isOpen([1×1 matlab.mock.classes.Mock])

Error in matlab.mock.internal.MockContext>@(data)mockMethodCallback(data,catalog) (line
            methodCallback = @(data)mockMethodCallback(data, catalog);
```

```
isAccountOpen = mock.isOpen
```

```
isAccountOpen =

  logical
```

```
0
```

Tips

- If you repeat an action, and do not follow it with a call to the `then` method, the mock continues to return the repeated value. For example, consider the following mock of a bank account class.

  ```
  import matlab.mock.actions.ThrowException
  testCase = matlab.mock.TestCase.forInteractiveUse;
  [mock, behavior] = testCase.createMock('AddedProperties',"IsJointAccount");
  ```

  If you repeat an action to throw an exception twice, the framework continues to throw an exception in the following code, which goes on to get the property value a third time.

  ```
  when(get(behavior.IsJointAccount),then(repeat(2,ThrowException)));
  tf = mock.IsJointAccount
  tf = mock.IsJointAccount
  tf = mock.IsJointAccount
  ```

  But the following code throws an exception twice and the returns `false`.

  ```
  import matlab.mock.actions.AssignOutputs
  when(get(behavior.IsJointAccount),then(repeat(2,ThrowException), ...
      then(AssignOutputs(false))));
  tf = mock.IsJointAccount
  tf = mock.IsJointAccount
  tf = mock.IsJointAccount
  ```

See Also

Introduced in R2017a

then

Class: `matlab.mock.actions.ThrowException`
Package: `matlab.mock.actions`

Specify action

Syntax

```
then(action1)
then(action1,action2)
```

Description

`then(action1)` specifies an action for mock object interactions.

`then(action1,action2)` specifies an action and a subsequent action for mock object interactions.

Input Arguments

`action1` — Defined action
instance of `matlab.mock.actions.ThrowException`

Defined action, specified as an instance of `matlab.mock.actions.ThrowException`.

Example: `ThrowException`

Example: `ThrowException(MException('Account:deposit:Negative','Deposit amount must be positive.'))`

`action2` — Second defined action
instance of `matlab.mock.actions.ThrowException` | instance of `matlab.mock.actions.AssignOutputs` | instance of `matlab.mock.actions.Invoke` | instance of `matlab.mock.actions.ReturnStoredValue` | instance of `matlab.mock.actions.StoreValue`

Second defined action, specified as an instance of `matlab.mock.actions.ThrowException`, `matlab.mock.actions.AssignOutputs`, `matlab.mock.actions.Invoke`, `matlab.mock.actions.ReturnStoredValue`, or `matlab.mock.actions.StoreValue`.

Example: `ThrowException`

Example: `AssignOutputs(7,13,42)`

Examples

Throw Exceptions for Mock Method Calls

Create a mock for a bank account class.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedMethods',"isOpen");
```

Specify behavior.

```
import matlab.mock.actions.ThrowException
import matlab.mock.actions.AssignOutputs
when(withExactInputs(behavior.isOpen),then(ThrowException,AssignOutputs(false)));
```

Use the mock.

```
isAccountOpen = mock.isOpen
```

```
Error using matlab.mock.internal.MockContext>mockMethodCallback (line 663)
The following method call was specified to throw an exception:
    isOpen([1×1 matlab.mock.classes.Mock])

Error in matlab.mock.internal.MockContext>@(data)mockMethodCallback(data,catalog) (line
            methodCallback = @(data)mockMethodCallback(data, catalog);
```

Use the mock.

```
isAccountOpen = mock.isOpen
```

```
isAccountOpen =

  logical
```

```
0
```

Tips

- Each call to `then` accepts up to two actions. To specify more subsequent actions, use multiple calls to `then`. For example, to specify three actions use `then(action1,then(action2,action3))`.

See Also

Introduced in R2017a

matlab.mock.AnyArguments class

Package: `matlab.mock`

Match any number of arguments

Description

Use the `AnyArguments` class to match any number of arguments when specifying mock behavior or qualifying mock interactions.

Construction

`AnyArguments` matches an unlimited, unspecified number of arguments, including zero. When defining mock behavior or qualifying mock interactions, specify `AnyArguments` as the last argument in the argument list.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Match Any Number of Arguments

Match any number of arguments.

```
import matlab.mock.AnyArguments
import matlab.mock.actions.ThrowException

testCase = matlab.mock.TestCase.forInteractiveUse;

% Create a mock for a bank account class
[saboteurAccount,behavior] = testCase.createMock('AddedMethods',"deposit");
```

```
% Define behavior to throw exception with any input argument
when(behavior.deposit(AnyArguments),ThrowException)

% All of the following interactions throw an exception:
saboteurAccount.deposit;
saboteurAccount.deposit(-10);
saboteurAccount.deposit(10);
saboteurAccount.deposit('a','b','c');
```

Alternatives

The `AnyArguments` class is functionally similar to using the `withAnyInputs` method of the `matlab.mock.MethodCallBehavior` class. For example, the following code blocks are similar.

```
% Using the AnyArguments class
import matlab.mock.AnyArguments;
testCase.verifyCalled(behavior.myMethod(AnyArguments));

% Using the withAnyInputs method
testCase.verifyCalled(withAnyInputs(behavior.myMethod))
```

However, `AnyArguments` requires that the mock is the first input argument, and `withAnyInputs` does not. The `MethodCallBehavior` class provides additional methods to specify behavior and record interactions, such as specification of exact inputs or a number of outputs.

See Also

`matlab.mock.TestCase`

Introduced in R2017a

matlab.mock.constraints.Occurred class

Package: `matlab.mock.constraints`

Constraint qualifying mock object interactions

Description

The `Occurred` constraint qualifies the occurrence of one or more mock object interactions. It produces a qualification failure for any actual-value array that specifies at least one interaction that did not occur. The actual value must be an array of `MethodCallBehavior`, `PropertyGetBehavior`, or `PropertySetBehavior` objects that all refer to the same mock object.

Use the `Occurred` constraint to qualify any combination of method calls, property accesses, or property modifications.

By default, the constraint qualifies that all interactions occurred at least once and in any order. The `RespectingOrder` name-value pair enables qualification that the interactions occurred in the specified order.

Construction

`constraint = matlab.mock.constraints.Occurred` provides a constraint that determines if all specified interactions occurred.

`constraint = matlab.mock.constraints.Occurred('RespectingOrder',tf)` provides a constraint that respects the order of occurrence of the specified interactions.

Input Arguments

`tf` — Respect order of interactions
`false` (default) | `true`

Whether to respect the order of interactions, specified as `false` or `true`. By default, the constraint does not require that interactions occur in a specified order.

Data Types: `logical`

Properties

`RespectOrder` — Respects order of interactions
`false` (default) | `true`

This property is read-only.

Whether the constraint respects the order of interactions, stored as `false` or `true`. The `RespectOrder` property is `false` by default, but can be set to `true` during construction of the constraint by using the `'RespectingOrder'` name-value pair.

Data Types: `logical`

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Qualify Mock Interactions

Create a mock for a person class.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[fakePerson,behavior] = testCase.createMock("AddedProperties",["Name","Age"], ...
    "AddedMethods","speak");
```

Use the mock by calling the `speak` method, accessing the `Age` property, and setting the `Name` property.

```
fakePerson.speak("hello");
age = fakePerson.Age;
fakePerson.Name = "Zed";
```

Verify that a call to the `speak` method with the input `"hello"` occurred.

```
import matlab.mock.constraints.Occurred;
testCase.verifyThat(behavior.speak("hello"),Occurred)
```

```
Verification passed.
```

Verify that the `Age` property was accessed, the `speak` method was called with `"hello"`, and the `Name` property was set to `"Zed"`.

```
testCase.verifyThat([get(behavior.Age), ...
    behavior.speak("hello"), ...
    behavior.Name.setToValue("Zed")],Occurred)
```

```
Verification passed.
```

Repeat the verification but require the constraint is satisfied only if the interactions occurred in the specified order. This test fails because the `speak` method was called before the `Age` property was set.

```
testCase.verifyThat([get(behavior.Age), ...
    behavior.speak("hello"), ...
    behavior.Name.setToValue("Zed")],Occurred('RespectingOrder',true))
```

```
Verification failed.

    ---------------------
    Framework Diagnostic:
    ---------------------
    Occurred failed.
    --> All specified interactions occurred.
    --> The interactions did not occur in the specified order.
        Actual order:
            speak([1×1 matlab.mock.classes.Mock], "hello")
            <Mock>.Age
            <Mock>.Name = "Zed"

    Specified interactions:
      1×3 heterogeneous InteractionBehavior (PropertyGetBehavior, MethodCallBehavior, PropertySetBehavior) a

        <Mock>.Age
        [...] = speak(<Mock>, "hello")
        <Mock>.Name = "Zed"
```

Repeat the verification and specify interactions should occur in a different order.

```
testCase.verifyThat([behavior.speak("hello"), ...
    get(behavior.Age), ...
    behavior.Name.setToValue("Zed")],Occurred('RespectingOrder',true))
```

```
Verification passed.
```

See Also

matlab.mock.MethodCallBehavior | matlab.mock.PropertyGetBehavior | matlab.mock.PropertySetBehavior | matlab.mock.TestCase

Introduced in R2018b

matlab.mock.constraints.WasAccessed class

Package: `matlab.mock.constraints`

Constraint determining property get access

Description

The `WasAccessed` constraint produces a qualification failure if an actual value is not a `PropertyBehavior` instance, or if the property that corresponds to the `PropertyBehavior` was not accessed the specified number of times.

The number of times a property is accessed includes the number of times that MATLAB implicitly accesses the property. For example, if you display a mock object, MATLAB accesses the object properties to display their values.

Construction

`constraint = WasAccessed` provides a constraint that determines the property get access. If a property was accessed at least once, the constraint is satisfied. To qualify that a property was not accessed, negate the `WasAccessed` constraint with the tilde (~) operator.

`constraint = WasAccessed('WithCount',n)` provides a constraint that is satisfied when a property was accessed exactly `n` times.

If you negate `WasAccessed` with this syntax, if the property was not accessed exactly `n` times, the constraint passes. For example, if a property was accessed 4 times, `~WasAccessed('WithCount',3)` passes and `~WasAccessed('WithCount',4)` fails.

Input Arguments

`n` — Number of times of property get access
integer

Number of times of property get access, specified as an integer.

Properties

`Count` — Property access count
integer

Property access count, returned as an integer. This property is read-only once the constraint is constructed. You can specify it during constraint construction.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Qualify Mock Property Get Access

Create a mock for a person class.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[fakePerson,behavior] = testCase.createMock('AddedProperties',["Name" "Age"]);
fakePerson.Name = 'David';
fprintf(1,'The person''s name is %s.\n',fakePerson.Name);
```

```
The person's name is David.
```

Construct passing cases.

```
import matlab.mock.constraints.WasAccessed
testCase.verifyThat(behavior.Name,WasAccessed)
```

```
Interactive verification passed.
```

```
testCase.verifyThat(behavior.Age,~WasAccessed)
```

```
Interactive verification passed.
```

```
testCase.verifyThat(behavior.Name,WasAccessed('WithCount',1))
```

```
Interactive verification passed.
```

Construct failing cases.

```
testCase.verifyThat(behavior.Name,~WasAccessed)
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
Negated WasAccessed failed.
--> Property 'Name' was unexpectedly accessed 1 time(s).

Specified property access:
    PropertyGetBehavior
        <Mock>.Name
```

```
testCase.verifyThat(behavior.Age,WasAccessed)
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
WasAccessed failed.
--> Property 'Age' was never accessed.

Specified property access:
    PropertyGetBehavior
        <Mock>.Age
```

```
testCase.verifyThat(behavior.Name,WasAccessed('WithCount',5))
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
WasAccessed failed.
--> Property 'Name' was not accessed the expected number of times.

    Actual property access count:
              1
    Expected property access count:
              5

Specified property access:
```

```
PropertyGetBehavior
    <Mock>.Name
```

See Also

matlab.mock.PropertyBehavior | matlab.mock.TestCase

Introduced in R2017a

matlab.mock.constraints.WasCalled class

Package: `matlab.mock.constraints`

Constraint determining method call

Description

The `WasCalled` constraint produces a qualification failure if an actual value is not a `MethodCallBehavior` instance, or if the method that corresponds to the `MethodCallBehavior` was not called the specified number of times.

Construction

`constraint = WasCalled` provides a constraint that determines a method call. If a method was called at least once, the constraint is satisfied. To qualify that a method was not called, negate the `WasCalled` constraint with the tilde (~) operator.

`constraint = WasCalled('WithCount',n)` provides a constraint that is satisfied when a method is called exactly `n` times.

If you negate `WasCalled` with this syntax, if the method was not called exactly `n` times, the constraint passes. For example, if a method was called four times, `~WasCalled('WithCount',3)` passes and `~WasCalled('WithCount',4)` fails.

Input Arguments

`n` — Number of method calls
integer

Number of method calls, specified as an integer.

Properties

`Count` — Method call count
integer

Method call count, returned as an integer. This property is read-only. You can specify it during constraint construction.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Qualify Mock Method Calls

Create a mock for a bank account class.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[fakeAccount,behavior] = testCase.createMock('AddedMethods',"deposit");
```

Use the mock account.

```
fakeAccount.deposit(10);
fakeAccount.deposit(20);
fakeAccount.deposit(10);
```

Construct passing cases.

```
import matlab.mock.constraints.WasCalled
testCase.verifyThat(behavior.deposit(10),WasCalled)
```

```
Interactive verification passed.
```

```
testCase.verifyThat(behavior.deposit(10),WasCalled('WithCount',2))
```

```
Interactive verification passed.
```

```
import matlab.unittest.constraints.IsGreaterThan
testCase.verifyThat(behavior.deposit(IsGreaterThan(100)),~WasCalled)
```

```
Interactive verification passed.
```

Construct failing cases.

```
testCase.verifyThat(behavior.deposit(100),WasCalled);
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
WasCalled failed.
--> Method 'deposit' was not called with the specified signature.
--> Observed method call(s) with any signature:
        deposit([1×1 matlab.mock.classes.Mock], 10)
        deposit([1×1 matlab.mock.classes.Mock], 20)
        deposit([1×1 matlab.mock.classes.Mock], 10)

Specified method call:
    MethodCallBehavior
        [...] = deposit(<Mock>, 100)
```

```
testCase.verifyThat(behavior.deposit(20),WasCalled('WithCount',2))
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
WasCalled failed.
--> Method 'deposit' was not called the expected number of times with the specified sig

    Actual method call count:
              1
    Expected method call count:
              2
--> Observed method call(s) with any signature:
        deposit([1×1 matlab.mock.classes.Mock], 10)
        deposit([1×1 matlab.mock.classes.Mock], 20)
        deposit([1×1 matlab.mock.classes.Mock], 10)

Specified method call:
    MethodCallBehavior
        [...] = deposit(<Mock, 20)
```

```
testCase.verifyThat(behavior.deposit(IsGreaterThan(50)),WasCalled)
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
WasCalled failed.
--> Method 'deposit' was not called with the specified signature.
--> Observed method call(s) with any signature:
        deposit([1×1 matlab.mock.classes.Mock], 10)
        deposit([1×1 matlab.mock.classes.Mock], 20)
        deposit([1×1 matlab.mock.classes.Mock], 10)

Specified method call:
    MethodCallBehavior
        [...] = deposit(<Mock>, <IsGreaterThan constraint>)
```

See Also

`matlab.mock.MethodCallBehavior` | `matlab.mock.TestCase`

Introduced in R2017a

matlab.mock.constraints.WasSet class

Package: `matlab.mock.constraints`

Constraint determining property set interaction

Description

The `WasSet` constraint produces a qualification failure if an actual value is not a `PropertyBehavior` instance, or if the property that corresponds to the `PropertyBehavior` was not set the specified number of times.

Construction

`constraint = WasSet` provides a constraint that determines property set interaction. If a property value was set at least once, the constraint is satisfied. To qualify that a property was not set, negate the `WasSet` constraint with the tilde (~) operator.

`constraint = WasSet(Name,Value)` provides a constraint with additional options specified by one or more `Name,Value` pair arguments. For example, `WasSet('ToValue',42)` constructs a constraint that is satisfied if a property value is set to 42, and `WasSet('ToValue',42,'WithCount',3)` constructs a constraint that is satisfied if a property value is set to 42 exactly 3 times.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`ToValue` — Specified property value
scalar | vector | matrix | multidimensional array

Specified property values, specified as a scalar, vector, matrix, or multidimensional array. Values can be any data type, and relate to the property specified by behavior.

Example: `'Joe'`

Example: `[1 2 3;4 5 6]`

`WithCount` — Number of times of property was set
integer

Number of times the property was set, specified as an integer.

If you negate `WasSet` with this syntax, if the property value was not set exactly `n` times, the constraint passes. For example, if a property was set four times, `~WasSet('WithCount',3)` passes and `~WasSet('WithCount',4)` fails.

Example: 5

Properties

`Value` — Property values
scalar | vector | matrix | multidimensional array

Property values, specified as a scalar, vector, matrix, or multidimensional array. Values can be any data type, and relate to the property specified by behavior.

`Count` — Property set access count
integer

Property set access count, returned as an integer. This property is read-only once the constraint is constructed. You can specify it during constraint construction.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Qualify Mock Property Set Access

Create a mock for a person class.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[fakePerson,behavior] = testCase.createMock('AddedProperties',["Name" "Age"]);
```

Use the mock.

```
fakePerson.Name = 'David';
```

Construct passing cases.

```
import matlab.mock.constraints.WasSet
testCase.verifyThat(behavior.Name,WasSet)
```

```
Interactive verification passed.
```

```
testCase.verifyThat(behavior.Age,~WasSet)
```

```
Interactive verification passed.
```

```
testCase.verifyThat(behavior.Name,WasSet('ToValue','David'))
```

```
Interactive verification passed.
```

```
testCase.verifyThat(behavior.Name,WasSet('WithCount',1))
```

```
Interactive verification passed.
```

Construct failing cases.

```
testCase.verifyThat(behavior.Name,~WasSet)
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
Negated WasSet failed.
--> Property 'Name' was unexpectedly set to the specified value 1 time(s).
--> Observed property set(s) to any value:
        <Mock>.Name = 'David'

Specified property set:
    PropertySetBehavior
        <Mock>.Name = <IsAnything constraint>
```

```
testCase.verifyThat(behavior.Age,WasSet)
```

```
Interactive verification failed.
```

```
----------------------
Framework Diagnostic:
----------------------
WasSet failed.
--> Property 'Age' was never set.

Specified property set:
    PropertySetBehavior
        <Mock>.Age = <IsAnything constraint>
```

```
testCase.verifyThat(behavior.Name,WasSet('ToValue','Andy'))
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
WasSet failed.
--> Property 'Name' was not set to the specified value.
--> Observed property set(s) to any value:
        <Mock>.Name = 'David'

Specified property set:
    PropertySetBehavior
        <Mock>.Name = 'Andy'
```

```
testCase.verifyThat(behavior.Name,WasSet('WithCount',5))
```

```
Interactive verification failed.

--------------------
Framework Diagnostic:
--------------------
WasSet failed.
--> Property 'Name' was not set to the specified value the expected number of times.

    Actual property set count:
             1
    Expected property set count:
             5
--> Observed property set(s) to any value:
        <Mock>.Name = 'David'

Specified property set:
    PropertySetBehavior
        <Mock>.Name = <IsAnything constraint>
```

See Also

matlab.mock.PropertyBehavior | matlab.mock.TestCase

Introduced in R2017a

matlab.mock.history Package

Summary of classes representing mock object interaction history

Description

The `matlab.mock.history` package consists of the following classes.

Classes

matlab.mock.history.MethodCall	Representation of mock object method call
matlab.mock.history.PropertyAccess	Representation of mock object property access
matlab.mock.history.PropertyModification	Representation of mock object property modification
matlab.mock.history.SuccessfulMethodCall	Representation of successful mock object method call
matlab.mock.history.SuccessfulPropertyAccess	Representation of successful mock object property access
matlab.mock.history.SuccessfulPropertyModification	Representation of successful mock object property modification
matlab.mock.history.UnsuccessfulMethodCall	Representation of unsuccessful mock object method call
matlab.mock.history.UnsuccessfulPropertyAccess	Representation of unsuccessful mock object property access
matlab.mock.history.UnsuccessfulPropertyModification	Representation of unsuccessful mock object property modification

See Also

`matlab.mock.InteractionHistory`

Introduced in R2018a

matlab.mock.history.MethodCall class

Package: `matlab.mock.history`
Superclasses: `matlab.mock.InteractionHistory`

Representation of mock object method call

Description

A `MethodCall` instance represents a call to a mock object method. The framework constructs instances of the class, so there is no need to construct this class directly.

Properties

`Name` — Name of mock object method

string scalar

This property is read-only.

Name of the mock object method called during an interaction, returned as a string scalar.

`Inputs` — Inputs passed to mock object method

cell vector

This property is read-only.

Inputs passed to mock object method, returned as a cell vector.

See Also

`matlab.mock.InteractionHistory.forMock` | `matlab.mock.history`

Introduced in R2018a

matlab.mock.history.PropertyAccess class

Package: `matlab.mock.history`
Superclasses: `matlab.mock.InteractionHistory`

Representation of mock object property access

Description

A `PropertyAccess` instance represents the access of a mock object property value. The framework constructs instances of the class, so there is no need to construct this class directly.

Properties

`Name` — Name of mock object property
string scalar

This property is read-only.

Name of the mock object property involved in an interaction, returned as a string scalar.

See Also

`matlab.mock.InteractionHistory.forMock` | `matlab.mock.history`

Introduced in R2018a

matlab.mock.history.PropertyModification class

Package: `matlab.mock.history`
Superclasses: `matlab.mock.InteractionHistory`

Representation of mock object property modification

Description

A `PropertyModification` instance represents the modification of a mock object property value. The framework constructs instances of the class, so there is no need to construct this class directly.

Properties

`Name` — Name of mock object property
string scalar

This property is read-only.

Name of the mock object property involved in an interaction, returned as a string scalar.

`Value` — Value assigned to mock object property
scalar | array

This property is read-only.

Value assigned to mock object property, returned as a scalar or an array. `Value` can have any data type.

See Also

`matlab.mock.InteractionHistory.forMock` | `matlab.mock.history`

Introduced in R2018a

matlab.mock.history.SuccessfulMethodCall class

Package: `matlab.mock.history`
Superclasses: `matlab.mock.history.MethodCall`

Representation of successful mock object method call

Description

A `SuccessfulMethodCall` instance represents a call to a mock object method that ran to completion. The framework constructs instances of the class, so there is no need to construct this class directly.

Properties

`Name` — Name of mock object method
string scalar

This property is read-only.

Name of the mock object method called during an interaction, returned as a string scalar.

`Inputs` — Inputs passed to mock object method
cell vector

This property is read-only.

Inputs passed to mock object method, returned as a cell vector.

`Outputs` — Outputs returned from mock object method
cell vector

This property is read-only.

Outputs returned from a mock object method, returned as a cell vector.

See Also

`matlab.mock.InteractionHistory.forMock` | `matlab.mock.history`

Introduced in R2018a

matlab.mock.history.SuccessfulPropertyAccess class

Package: `matlab.mock.history`
Superclasses: `matlab.mock.history.PropertyAccess`

Representation of successful mock object property access

Description

A `SuccessfulPropertyAccess` instance represents the successful access of a mock object property value. The framework constructs instances of the class, so there is no need to construct this class directly.

Properties

`Name` — Name of mock object property
string scalar

This property is read-only.

Name of the mock object property involved in an interaction, returned as a string scalar.

`Value` — Return value of mock object property
scalar | array

This property is read-only.

Return value of the mock object property, returned as a scalar or an array. `Value` can have any data type.

See Also

`matlab.mock.InteractionHistory.forMock` | `matlab.mock.history`

Introduced in R2018a

matlab.mock.history.SuccessfulPropertyModification class

Package: `matlab.mock.history`
Superclasses: `matlab.mock.history.PropertyModification`

Representation of successful mock object property modification

Description

A `SuccessfulPropertyModification` instance represents the successful modification of a mock object property value. The framework constructs instances of the class, so there is no need to construct this class directly.

Properties

`Name` — Name of mock object property
string scalar

This property is read-only.

Name of the mock object property involved in an interaction, returned as a string scalar.

`Value` — Value assigned to mock object property
scalar | array

This property is read-only.

Value assigned to mock object property, returned as a scalar or an array. `Value` can have any data type.

See Also

`matlab.mock.InteractionHistory.forMock` | `matlab.mock.history`

Introduced in R2018a

matlab.mock.history.UnsuccessfulMethodCall class

Package: `matlab.mock.history`
Superclasses: `matlab.mock.history.MethodCall`

Representation of unsuccessful mock object method call

Description

An `UnsuccessfulMethodCall` instance represents a call to a mock object method that threw an exception. The framework constructs instances of the class, so there is no need to construct this class directly.

Properties

`Name` — Name of mock object method
string scalar

This property is read-only.

Name of the mock object method called during an interaction, returned as a string scalar.

`Inputs` — Inputs passed to mock object method
cell vector

This property is read-only.

Inputs passed to mock object method, returned as a cell vector.

`Exception` — Exception produced by mock object interaction
scalar `MException` object

This property is read-only.

Exception produced by mock object interaction, returned as a scalar `MException` object.

See Also

`matlab.mock.InteractionHistory.forMock` | `matlab.mock.history`

Introduced in R2018a

matlab.mock.history.UnsuccessfulPropertyAccess class

Package: `matlab.mock.history`
Superclasses: `matlab.mock.history.PropertyAccess`

Representation of unsuccessful mock object property access

Description

An `UnsuccessfulPropertyAccess` instance represents the access of a mock object property that threw an exception. The framework constructs instances of the class, so there is no need to construct this class directly.

Properties

`Name` — Name of mock object property
string scalar

This property is read-only.

Name of the mock object property involved in an interaction, returned as a string scalar.

`Exception` — Exception produced by mock property access
scalar `MException` object

This property is read-only.

Exception produced by mock property access, returned as a scalar `MException` object.

See Also

`matlab.mock.InteractionHistory.forMock` | `matlab.mock.history`

Introduced in R2018a

matlab.mock.history.UnsuccessfulPropertyModification class

Package: `matlab.mock.history`
Superclasses: `matlab.mock.history.PropertyModification`

Representation of unsuccessful mock object property modification

Description

An `UnsuccessfulPropertyModification` instance represents the modification of a mock object property value that threw an exception. The framework constructs instances of the class, so there is no need to construct this class directly.

Properties

`Name` — Name of mock object property

string scalar

This property is read-only.

Name of the mock object property involved in an interaction, returned as a string scalar.

`Value` — Value assigned to mock object property

scalar | array

This property is read-only.

Value assigned to mock object property, returned as a scalar or an array. `Value` can have any data type.

`Exception` — Exception produced by mock property modification

scalar `MException` object

This property is read-only.

Exception produced by mock property modification, returned as a scalar `MException` object.

See Also

`matlab.mock.InteractionHistory.forMock` | `matlab.mock.history`

Introduced in R2018a

matlab.mock.InteractionHistory class

Package: `matlab.mock`

Interface for mock object interaction history

Description

`matlab.mock.InteractionHistory` is the interface for representing interactions with mock objects. Interactions include method calls, property modifications, and property accesses. The framework constructs instances of the class, so there is no need to construct it directly.

Properties

`Name` — Method or property name
string scalar

This property is read-only.

Method or property name, returned as a string scalar. `Name` indicates the mock object method or property that was involved in the interaction.

Methods

forMock	Return history from mock object

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

See Also

`matlab.mock.history`

Introduced in R2018a

matlab.mock.InteractionHistory.forMock

Class: `matlab.mock.InteractionHistory`
Package: `matlab.mock`

Return history from mock object

Syntax

```
history = matlab.mock.InteractionHistory.forMock(mock)
```

Description

`history = matlab.mock.InteractionHistory.forMock(mock)` returns the history from a mock object. `history` is an array of `matlab.mock.InteractionHistory` objects. Each element in `history` corresponds to one method call, property access, or property modification. The array elements are ordered, with the first element indicating the first recorded interaction. This method returns interactions with publicly visible methods and properties only. For example, the following interactions are not recorded:

- Calls to `Hidden` methods
- Calls to `Sealed` superclass methods
- Accesses or modifications of concrete superclass properties

Input Arguments

`mock` — Mock to return history
mock object

Mock to return history of interactions, specified as a mock object.

Examples

Obtain History of Mock Interactions

Construct a mock with a `computeValue` method and two properties. Assign a default value of `false` to `Prop2`.

```
tc = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = tc.createMock( ...
    'AddedMethods',{'computeValue'}, ...
    'AddedProperties',{'Prop1','Prop2'}, ...
    'DefaultPropertyValues',struct('Prop2',false));
```

Set up the behavior of the `computeValue` method to return the value 42, regardless of input values.

```
import matlab.mock.actions.AssignOutputs;
when(withAnyInputs(behavior.computeValue),AssignOutputs(42));
```

Interact with the mock. First call the `computeValue` method. Then display the value of `Prop2`. Finally, set the value of `Prop1`.

```
n = mock.computeValue('hello');
mock.Prop2
mock.Prop1 = 13;
```

```
ans =

  logical

   0
```

Obtain the interaction history for the mock.

```
h = matlab.mock.InteractionHistory.forMock(mock)
```

```
h =

  1×3 heterogeneous InteractionHistory (SuccessfulMethodCall, SuccessfulPropertyAccess, SuccessfulPropertyMo

    Name

Interaction summary:
  computeValue([1×1 matlab.mock.classes.Mock], 'hello')
  <Mock>.Prop2
  <Mock>.Prop1 = 13
```

Examine the first `InteractionHistory` object. The method was called with the mock object and the character vector `'hello'` as inputs. The method output the value 42.

```
h(1)

ans =

  SuccessfulMethodCall with properties:

       Name: "computeValue"
     Inputs: {[1×1 matlab.mock.classes.Mock]  'hello'}
    Outputs: {[42]}

Interaction summary:
  computeValue([1×1 matlab.mock.classes.Mock], 'hello')
```

Alternatives

You can obtain the same history of interactions using the `getMockHistory` method on a `matlab.mock.TestCase` instance. For example, if you have a `matlab.mock.TestCase` instance `tc`, and a mock object `mock`, the following method calls are equivalent.

```
h = matlab.mock.InteractionHistory.forMock(mock);
h = tc.getMockHistory(mock);
```

However, you do not need access to the `matlab.mock.TestCase` instance to use the `forMock` method.

See Also

`matlab.mock.history`

Introduced in R2018a

matlab.mock.MethodCallBehavior class

Package: `matlab.mock`

Specify mock object method behavior and qualify method calls

Description

Use the `MethodCallBehavior` object to specify behavior for a mock object method and to qualify method calls.

There are several ways to define behavior for a mock object method.

- Pass a mock object action, such as `matlab.mock.actions.AssignOutputs`, to the `when` method of the `MethodCallBehavior` class.
- Pass a `MethodCallBehavior` object to a method of the `matlab.mock.TestCase` class, such as `assignOutputsWhen`.

There are several ways to qualify interactions with the mock object method.

- Pass a mock object constraint, such as `matlab.mock.constraints.WasCalled`, to the `verifyThat`, `assertThat`, `fatalAssertThat`, or `assumeThat` method of the `matlab.unittest.TestCase`.
- Pass a `MethodCallBehavior` object to a method of the `matlab.mock.TestCase` class, such as `verifyCalled` or `assumeNotCalled`.

Construction

The mocking framework constructs a `MethodCallBehavior` instance when you call a method of the mock behavior object.

Methods

when	Specify mock object method behavior
withAnyInputs	Specify mock object method call with any number of inputs with any value
withExactInputs	Specify mock object method call with only object as input
withNargout	Specify mock object method call with defined number of output arguments

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

See Also

`matlab.mock.TestCase` | `matlab.mock.constraints.Occurred` | `matlab.mock.constraints.WasCalled`

Introduced in R2017a

when

Class: `matlab.mock.MethodCallBehavior`
Package: `matlab.mock`

Specify mock object method behavior

Syntax

```
when(behavior,action)
```

Description

`when(behavior,action)` specifies the action that a mock object method takes when it is called with the inputs defined by `behavior`.

Input Arguments

`behavior` — Behavior of mock
`matlab.mock.MethodCallBehavior` instance

Behavior of the mock, specified as a `matlab.mock.MethodCallBehavior` instance. To create an instance of `matlab.mock.MethodCallBehavior`, call a method of the behavior object.

Example: `withExactInputs(myMockBehavior.myMockedMethod)`

`action` — Defined action
instance of `matlab.mock.actions.AssignOutputs` | instance of `matlab.mock.actions.ThrowException` | instance of `matlab.mock.actions.Invoke`

Defined action, specified as an instance of `matlab.mock.actions.AssignOutputs`, `matlab.mock.actions.ThrowException`, or `matlab.mock.actions.Invoke`.

Example: `AssignOutputs(7,13,42)`

Example: `ThrowException(MException('Account:deposit:Negative','Deposit amount must be positive.'))`

Examples

Specify Mock Method Behavior

Create a mock for a triangle class. The mock has one method, `sideLengths`.

```
import matlab.mock.actions.AssignOutputs;
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedMethods',"sideLengths");
```

Set up behavior. Regardless of the inputs to the `sideLengths` method, the mock returns the values 2, 3, and 4.

```
when(withAnyInputs(behavior.sideLengths),AssignOutputs(2,3,4))
```

Call the `sideLengths` method of the mock object.

```
[a,b,c] = mock.sideLengths
```

```
a = 2
```

```
b = 3
```

```
c = 4
```

Call the `sideLengths` method again using different inputs and only two outputs.

```
[a,b] = mock.sideLengths(13,"inputText")
```

```
a = 2
```

```
b = 3
```

See Also

`matlab.mock.actions.AssignOutputs` | `matlab.mock.actions.Invoke` | `matlab.mock.actions.ThrowException`

Introduced in R2017a

withAnyInputs

Class: `matlab.mock.MethodCallBehavior`
Package: `matlab.mock`

Specify mock object method call with any number of inputs with any value

Syntax

```
withAnyInputs(behavior)
```

Description

`withAnyInputs(behavior)` specifies a mock object method call with any number of inputs that can have any value.

Input Arguments

`behavior` — Behavior of mock
`matlab.mock.MethodCallBehavior` instance

Behavior of the mock, specified as a `matlab.mock.MethodCallBehavior` instance. To create an instance of `matlab.mock.MethodCallBehavior`, call a method of the behavior object.

Example: `myMockBehavior.myMockedMethod`

Examples

Specify Method Call with Any Inputs

Create a mock with a `myMethod` method.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedMethods',"myMethod");
```

Set up the behavior. Regardless of the inputs to the method, it returns `"hello"`.

```
testCase.assignOutputsWhen(withAnyInputs(behavior.myMethod),"hello")
```

Call the method with various inputs.

```
output = mock.myMethod
```

```
output =
"hello"
```

```
output = mock.myMethod(123)
```

```
output =
"hello"
```

```
output = myMethod("abc",mock)
```

```
output =
"hello"
```

Verify that the method was called at least once with some set of inputs.

```
testCase.verifyCalled(withAnyInputs(behavior.myMethod))
```

```
Verification passed.
```

See Also

`withExactInputs`

Introduced in R2017a

withExactInputs

Class: `matlab.mock.MethodCallBehavior`
Package: `matlab.mock`

Specify mock object method call with only object as input

Syntax

```
withExactInputs(behavior)
```

Description

`withExactInputs(behavior)` specifies a mock object method call with only the object as an input.

Input Arguments

`behavior` — Behavior of mock
`matlab.mock.MethodCallBehavior` instance

Behavior of the mock, specified as a `matlab.mock.MethodCallBehavior` instance. To create an instance of `matlab.mock.MethodCallBehavior`, call a method of the behavior object.

Example: `myMockBehavior.myMockedMethod`

Examples

Specify Method Call with Object Only

Create a mock with a `myMethod` method.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedMethods',{'myMethod'});
```

Set up behavior. If the method is called with only the object as input, return `"hello"`.

```
testCase.assignOutputsWhen(withExactInputs(behavior.myMethod),"hello")
```

Call the method with only the object as an input.

```
output = myMethod(mock)
```

```
output =
"hello"
```

Call the method with additional inputs. Since this behavior is not defined, the mock returns the default value.

```
output = mock.myMethod(123)
```

```
output =

     []
```

Verify that the method was called at least once with only the object as an input.

```
testCase.verifyCalled(withExactInputs(behavior.myMethod))
```

```
Verification passed.
```

See Also

`withAnyInputs`

Introduced in R2017a

withNargout

Class: `matlab.mock.MethodCallBehavior`
Package: `matlab.mock`

Specify mock object method call with defined number of output arguments

Syntax

```
withNargout(n,behavior)
```

Description

`withNargout(n,behavior)` specifies a mock object method call with a defined number of output arguments.

Input Arguments

n — Number of outputs
integer

Number of outputs from mock object method, specified as an integer.

Example: 3

behavior — Behavior of mock
`matlab.mock.MethodCallBehavior` instance

Behavior of the mock, specified as a `matlab.mock.MethodCallBehavior` instance. To create an instance of `matlab.mock.MethodCallBehavior`, call a method of the behavior object.

Example: `myMockBehavior.myMockedMethod`

Examples

Specify Method Call with Specific Number of Outputs

Create a mock for a triangle class.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mockQuad,behavior] = testCase.createMock('AddedMethods',"sideLengths");
```

Set up behavior. If the `sideLengths` method is called with only the object as input and with one output, then return an array of three values. If it is called with only the object as input and with three outputs, then return the three values. Otherwise, return the default value of an empty array.

```
import matlab.mock.actions.AssignOutputs
when(withNargout(1,withExactInputs(behavior.sideLengths)), ...
    AssignOutputs([1 2 3]))
when(withNargout(3,withExactInputs(behavior.sideLengths)), ...
    AssignOutputs(1,2,3))
```

Call the `sideLengths` method with only the object as an input and one output.

```
len = mockQuad.sideLengths
```

```
len = 1×3

     1     2     3
```

Verify that the `sideLengths` method was called at least once with one output argument.

```
import matlab.mock.constraints.WasCalled
testCase.verifyThat(withNargout(1, ...
    withExactInputs(behavior.sideLengths)),WasCalled)
```

```
Verification passed.
```

Verify that the `sideLengths` method was not called with three output arguments.

```
testCase.verifyThat(withNargout(3, ...
    withExactInputs(behavior.sideLengths)),~WasCalled)
```

```
Verification passed.
```

See Also

Introduced in R2017a

matlab.mock.PropertyBehavior class

Package: `matlab.mock`

Specify mock object property behavior and qualify interactions

Description

Use the `PropertyBehavior` class to specify mock object property behavior and qualify interactions.

There are several ways to qualify interactions with mock object properties.

- Pass the `PropertyBehavior` instance to a method of the `matlab.mock.TestCase`, such as `verifyAccessed` or `assumeNotSet`.
- Pass the `PropertyBehavior` instance and a mock object constraint, such as `matlab.mock.constraints.WasAccessed`, to the `verifyThat`, `assertThat`, `fatalAssertThat`, or `assumeThat` method of the `matlab.unittest.TestCase` class.

To define behavior for a mock object property, you first need a `PropertyGetBehavior` or `PropertySetBehavior` instance. To create one of these instances, call a `PropertyBehavior` method. For more information on defining property behavior, see `matlab.mock.PropertyGetBehavior` or `matlab.mock.PropertySetBehavior`.

Construction

The mocking framework constructs a `PropertyBehavior` instance when you access a property on the mock behavior object.

Methods

get	Construct object to define mock property get behavior
setToValue	Construct object to define behavior when mocked property is set to specific value
set	Construct object to define mock property set behavior

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

See Also

`matlab.mock.PropertyGetBehavior` | `matlab.mock.PropertySetBehavior` | `matlab.mock.constraints.Occurred` | `matlab.mock.constraints.WasAccessed` | `matlab.mock.constraints.WasSet`

Introduced in R2017a

get

Class: `matlab.mock.PropertyBehavior`
Package: `matlab.mock`

Construct object to define mock property get behavior

Syntax

```
getBehavior = get(behavior)
```

Description

`getBehavior = get(behavior)` constructs a `PropertyGetBehavior` object to define mock property get behavior. Typically you use the `get` method to construct the `PropertyGetBehavior` implicitly when you define mock behavior.

Input Arguments

`behavior` — Behavior of mock
`matlab.mock.PropertyBehavior` instance

Behavior of the mock, specified as a `matlab.mock.PropertyBehavior` instance. To create an instance of `matlab.mock.PropertyBehavior`, access a property of the behavior object.

Example: `myMockBehavior.MyProperty`

Examples

Define Behavior with `PropertyGetBehavior` Object

Create a mock for a person class with a `Name` property.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties',"Name");
```

Create a `PropertyGetBehavior` and set up the behavior. The mock returns `"David"` when you get the value of the `Name` property.

```
getBehavior = get(behavior.Name);
testCase.assignOutputsWhen(getBehavior,"David")
```

Alternatively, you can create the `PropertyGetBehavior` object implicitly with the behavior definition.

```
testCase.assignOutputsWhen(get(behavior.Name),"David")
```

Access the `Name` property.

```
name = mock.Name
```

```
name =
"David"
```

See Also

`matlab.mock.PropertyGetBehavior`

Introduced in R2017a

setToValue

Class: `matlab.mock.PropertyBehavior`
Package: `matlab.mock`

Construct object to define behavior when mocked property is set to specific value

Syntax

```
setBehavior = setToValue(behavior,value)
```

Description

`setBehavior = setToValue(behavior,value)` constructs a `PropertySetBehavior` object to define behavior when mocked property is set to a specific value. Typically you use the `setToValue` method to construct the `PropertySetBehavior` implicitly when you define mock behavior.

Input Arguments

`behavior` — Behavior of mock
`matlab.mock.PropertyBehavior` instance

Behavior of the mock, specified as a `matlab.mock.PropertyBehavior` instance. To create an instance of `matlab.mock.PropertyBehavior`, access a property of the behavior object.

Example: `myMockBehavior.MyProperty`

`value` — Property values
scalar | vector | matrix | multidimensional array | constraint | behavior object

Property values that the mock property must be set to, specified as a scalar, vector, matrix, multidimensional array, constraint, or behavior object. Values can be any data type, and relate to the property specified by behavior.

Example: `"hello"`

Example: 42

Example: [1 2 3]

Example: matlab.unittest.constraints.IsLessThan(10)

Examples

Define Behavior for Specific Property Values

Create a mock for a person class with a Name property.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties',"Name");
```

Create a PropertySetBehavior object and set up the behavior. The mock throws an exception when you set the value of the Name property to "David".

```
setBehavior = setToValue(behavior.Name,"David");
testCase.throwExceptionWhen(setBehavior)
```

Alternatively, you can create the PropertySetBehavior object implicitly with the behavior definition.

```
testCase.throwExceptionWhen(behavior.Name.setToValue("David"))
```

Set the value of the Name property.

```
mock.Name = "Andy";
mock.Name = "David";
```

```
Error using matlab.mock.internal.MockContext/createMockObject/mockPropertySetCallback (
The following property set was specified to throw an exception:
    <Mock>.Name = "David"
```

See Also

matlab.mock.PropertySetBehavior

Introduced in R2017a

set

Class: `matlab.mock.PropertyBehavior`
Package: `matlab.mock`

Construct object to define mock property set behavior

Syntax

```
setBehavior = set(behavior)
```

Description

`setBehavior = set(behavior)` constructs a `PropertySetBehavior` object to define mock property set behavior. Typically you use the `set` method to construct the `PropertySetBehavior` implicitly when you define mock behavior.

Input Arguments

`behavior` — Behavior of mock
`matlab.mock.PropertyBehavior` instance

Behavior of the mock, specified as a `matlab.mock.PropertyBehavior` instance. To create an instance of `matlab.mock.PropertyBehavior`, access a property of the behavior object.

Example: `myMockBehavior.MyProperty`

Examples

Define Behavior with `PropertySetBehavior` Object

Create a mock for a person class with a `Name` property.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties',"Name");
```

Create a `PropertySetBehavior` object and set up the behavior. The mock throws an exception when you set the value of the `Name` property.

```
setBehavior = set(behavior.Name);
testCase.throwExceptionWhen(setBehavior)
```

Alternatively, you can create the `PropertySetBehavior` object implicitly with the behavior definition.

```
testCase.throwExceptionWhen(set(behavior.Name))
```

Set the value of the `Name` property.

```
mock.Name = "Andy";
```

```
Error using matlab.mock.internal.MockContext/createMockObject/mockPropertySetCallback (
The following property set was specified to throw an exception:
    <Mock>.Name = "Andy"
```

See Also

`matlab.mock.PropertySetBehavior`

Introduced in R2017a

matlab.mock.PropertyGetBehavior class

Package: `matlab.mock`

Specify mock property get behavior

Description

Use the `PropertyGetBehavior` class to specify mock object get behavior. There are several ways to specify get behavior.

- Pass a mock object action, such as `matlab.mock.actions.AssignOutputs`, to the `when` method.
- Pass a `PropertyGetBehavior` object to a method of the `matlab.mock.TestCase` class, such as `assignOutputsWhen`.

To qualify mock property interactions, see `matlab.mock.PropertyBehavior`.

Construction

The mocking framework creates a `PropertyGetBehavior` instance when you call the `get` method. Typically, you construct the `PropertyGetBehavior` implicitly. For example, `testCase.assignOutputsWhen(get(behavior.MyProperty),'abc')`.

Methods

when	Specify mock object property access action

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

See Also

`matlab.mock.PropertyBehavior` | `matlab.mock.TestCase`

Introduced in R2017a

when

Class: `matlab.mock.PropertyGetBehavior`
Package: `matlab.mock`

Specify mock object property access action

Syntax

```
when(behavior,action)
```

Description

`when(behavior,action)` specifies the action that a mock object property takes when it is accessed.

Input Arguments

`behavior` — Behavior of mock
`matlab.mock.PropertyGetBehavior` instance

Behavior of the mock, specified as a `matlab.mock.PropertyGetBehavior` instance. To create an instance of `matlab.mock.PropertyGetBehavior`, call the `matlab.mock.PropertyBehavior.get` method with the behavior object.

Example: `get(myMockBehavior.MyProperty)`

`action` — Defined action
instance of `matlab.mock.actions.ReturnStoredValue` | instance of `matlab.mock.actions.ThrowException`

Defined action, specified as an instance of `matlab.mock.actions.ReturnStoredValue` or `matlab.mock.actions.ThrowException`.

Example: `ReturnStoredValue`

Example: `ThrowException(MException('Account:deposit:Negative','Deposit amount must be positive.'))`

Examples

Specify Mock Property Access Behavior

Create a mock for a person class. The mock has one property, `Name`.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties',"Name");
```

Set up the behavior. When the property is accessed, return the value `"David"`.

```
import matlab.mock.actions.AssignOutputs
when(get(behavior.Name),AssignOutputs("David"))
```

Access the property.

```
name = mock.Name
```

```
name =
"David"
```

See Also

`get` | `matlab.mock.actions.ReturnStoredValue` | `matlab.mock.actions.ThrowException`

Introduced in R2017a

matlab.mock.PropertySetBehavior class

Package: `matlab.mock`

Specify mock object set behavior

Description

Use the `PropertySetBehavior` class to specify mock object set behavior. There are several ways to specify property set behavior.

- Pass a mock object action, such as `matlab.mock.actions.StoreValue`, to the `when` method.
- Pass a `PropertySetBehavior` object to a method of the `matlab.mock.TestCase` class, such as `storeValueWhen`.

To qualify mock property interactions, see `matlab.mock.PropertyBehavior`.

Construction

The mocking framework creates a `PropertySetBehavior` instance when you call the `matlab.mock.PropertyBehavior.set` or `matlab.mock.PropertyBehavior.setToValue` method. Typically, you construct the `PropertySetBehavior` implicitly. For example, `testCase.throwExceptionWhen(set(behavior.MyProperty))`.

Methods

when	Specify mock object property set action

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

See Also

`matlab.mock.PropertyBehavior` | `matlab.mock.TestCase`

Introduced in R2017a

when

Class: `matlab.mock.PropertySetBehavior`
Package: `matlab.mock`

Specify mock object property set action

Syntax

```
when(behavior,action)
```

Description

`when(behavior,action)` specifies the action that a mock object property takes when it is set.

Input Arguments

`behavior` — Behavior of mock
`matlab.mock.PropertySetBehavior` instance

Behavior of the mock, specified as a `matlab.mock.PropertySetBehavior` instance. To create an instance of `matlab.mock.PropertySetBehavior`, call the `matlab.mock.PropertyBehavior.set` or `matlab.mock.PropertyBehavior.setToValue` method with the behavior object.

Example: `set(myMockBehavior.MyProperty)`

Example: `setToValue(myMockBehavior.MyProperty,13)`

`action` — Defined action
instance of `matlab.mock.actions.StoreValue` | instance of `matlab.mock.actions.ThrowException`

Defined action, specified as an instance of `matlab.mock.actions.StoreValue` or `matlab.mock.actions.ThrowException`.

Example: `StoreValue`

Example: `ThrowException(MException('Account:deposit:Negative','Deposit amount must be positive.'))`

Examples

Specify Mock Property Set Behavior

Create a strict mock for a person class. The mock has one property, `Name`.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties',"Name",'Strict',true);
```

Set up the behavior. When the property is set, store the value, and when the property is accessed, return the value. Without defining this behavior, strict mocks throw an exception when setting or accessing the property.

```
import matlab.mock.actions.StoreValue
import matlab.mock.actions.ReturnStoredValue
when(set(behavior.Name),StoreValue)
when(get(behavior.Name),ReturnStoredValue)
```

Set the property. The property access is implicit because we are displaying the result.

```
mock.Name = "David"
```

```
mock 
  Mock with properties:

    Name: "David"
```

See Also

`matlab.mock.actions.StoreValue` | `matlab.mock.actions.ThrowException` | `set` | `setToValue`

Introduced in R2017a

matlab.mock.TestCase class

Package: `matlab.mock`
Superclasses: `matlab.unittest.TestCase`

`TestCase` to write tests with mocking framework

Description

Use the `matlab.mock.TestCase` class to write tests that use the mocking framework. The `matlab.mock.TestCase` derives from the `matlab.unittest.TestCase` class.

Construction

The testing framework constructs the `matlab.mock.TestCase` instances.

Methods

assertAccessed	Assert that a property was accessed
assertCalled	Assert that a method was called with certain input values
assertSet	Assert that a property was set
assertNotAccessed	Assert that a property was not accessed
assertNotCalled	Assert that a method was not called with certain input values
assertNotSet	Assert that a property was not set
assignOutputsWhen	Define return values for method call or property access
assumeAccessed	Assume that a property was accessed
assumeCalled	Assume that a method was called with certain input values
assumeSet	Assume that a property was set
assumeNotAccessed	Assume that a property was not accessed
assumeNotCalled	Assume that a method was not called with certain input values
assumeNotSet	Assume that a property was not set
clearMockHistory	Clear history of mock object interactions
createMock	Create mock object
fatalAssertAccessed	Fatally assert that a property was accessed
fatalAssertCalled	Fatally assert that a method was called with certain input values
fatalAssertSet	Fatally assert that a property was set
fatalAssertNotAccessed	Fatally assert that a property was not accessed
fatalAssertNotCalled	Fatally assert that a method was not called with certain input values
fatalAssertNotSet	Fatally assert that a property was not set
getMockHistory	Return history from mock object
forInteractiveUse	Create `TestCase` for interactive use
returnStoredValueWhen	Return stored property value
storeValueWhen	Store property value
throwExceptionWhen	Throw exception for method call or property interaction
verifyAccessed	Verify that a property was accessed
verifyCalled	Verify that a method was called with certain input values
verifySet	Verify that a property was set
verifyNotAccessed	Verify that a property was not accessed
verifyNotCalled	Verify that a method was not called with certain input values
verifyNotSet	Verify that a property was not set

Inherited Methods

addTeardown	Dynamically add teardown routine
applyFixture	Use fixture with `TestCase`
forInteractiveUse	Create TestCase for interactive use
getSharedTestFixtures	Provide access to shared test fixtures
log	Record diagnostic information
onFailure	Dynamically add diagnostics for test failures
run	Run `TestCase` test

Also, the `TestCase` class inherits methods from these classes:

matlab.unittest.qualifications.Assertable	Qualification to validate preconditions of a test
matlab.unittest.qualifications.Assumable	Qualification to filter test content
matlab.unittest.qualifications.FatalAssertable	Qualification to abort test execution
matlab.unittest.qualifications.Verifiable	Qualification to produce soft failure conditions

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Write Test Using Mock

Write a test using a mock.

```
import matlab.unittest.constraints.IsLessThan;
testCase = matlab.mock.TestCase.forInteractiveUse;

% Create a mock for a bank account class
[mock, behavior] = testCase.createMock('AddedMethods',["deposit" "isOpen"]);

% Set up behavior
testCase.throwExceptionWhen(behavior.deposit(IsLessThan(0)), ...
    MException('Account:deposit:Negative', ...
    'Deposit amount must be positive.'));

% Use mock object
mock.deposit(100);
testCase.verifyError(@() mock.deposit(-10), 'Account:deposit:Negative');

% Passing verifications
testCase.verifyCalled(behavior.deposit(100),...
    'A $100 deposit should have been made.');
testCase.assertNotCalled(behavior.deposit(0));
testCase.assertCalled(behavior.deposit(IsLessThan(0)));

% Failing assertion
testCase.assertCalled(withExactInputs(behavior.isOpen));
```

See Also

`matlab.unittest.TestCase`

Topics

"Create Mock Object"
"Specify Mock Object Behavior"
"Qualify Mock Object Interaction"
"Write Test That Uses App Testing and Mocking Frameworks"

Introduced in R2017a

assertAccessed

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Assert that a property was accessed

Syntax

```
assertAccessed(testcase,behavior)
assertAccessed(testcase,behavior,diagnostic)
```

Description

`assertAccessed(testcase,behavior)` asserts that a property was accessed.

`assertAccessed(testcase,behavior,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.PropertyBehavior` instance

Behavior of the mock, specified as a `matlab.mock.PropertyBehavior` instance. To create an instance of `matlab.mock.PropertyBehavior`, access a property of the behavior object.

Example: `behavior.PropertyFoo`

`diagnostic` — Diagnostic information to display
string array | character array | function handle | `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic information to display, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` object. Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Example: `"My diagnostic message."`

Example: `@() datetime('now')`

Examples

Assert Property Was Accessed

Create a mock with an implicit interface for interactive use. The interface includes `PropertyFoo` and `PropertyBar` properties. Access `PropertyFoo` by storing its value.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties', ...
    ["PropertyFoo","PropertyBar"]);
value = mock.PropertyFoo;
```

Assert that `PropertyFoo` was accessed.

```
testCase.assertAccessed(behavior.PropertyFoo);
```

```
Assertion passed.
```

Display Diagnostic When Assertion Fails

Assert that `PropertyBar` was accessed. Display the diagnostic information upon failure.

```
testCase.assertAccessed(behavior.PropertyBar, ...
    'PropertyBar should have been accessed.');
```

```
Assertion failed.
    ----------------
```

```
    Test Diagnostic:
    ----------------
    PropertyBar should have been accessed.
    ----------------------
    Framework Diagnostic:
    ----------------------
    assertAccessed failed.
    --> Property 'PropertyBar' was never accessed.

    Specified property access operation:
    PropertyGetBehavior
        <Mock>.PropertyBar
Assertion failed.
```

Tips

Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point marks the current test method as failed and incomplete.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

Alternatives

Using the `assertAccessed` method is functionally equivalent to using the `matlab.mock.constraints.WasAccessed` constraint with the `assertThat` method of

the `Assertable` class. For example, the following code blocks are functionally equivalent.

```
% Using the assertAccessed method
testCase.assertAccessed(behavior.PropertyFoo);

% Using the WasAccessed constraint with assertThat method
import matlab.mock.constraints.WasAccessed;
testCase.assertThat(behavior.PropertyFoo,WasAccessed);
```

However, there is more functionality when you use the `WasAccessed` constraint. For instance, you can specify a property was accessed a certain number of times.

See Also

`matlab.mock.constraints.WasAccessed`

Introduced in R2017a

assertCalled

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Assert that a method was called with certain input values

Syntax

```
assertCalled(testcase,behavior)
assertCalled(testcase,behavior,diagnostic)
```

Description

`assertCalled(testcase,behavior)` asserts that a method was called with certain input values.

`assertCalled(testcase,behavior,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.MethodCallBehavior` instance

Behavior of the mock, specified as a `matlab.mock.MethodCallBehavior` instance. To create an instance of `matlab.mock.MethodCallBehavior`, call a method of the behavior object.

Example: `withExactInputs(myMockBehavior.myMockedMethod)`

`diagnostic` — Diagnostic information to display
string array | character array | function handle | `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic information to display, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` object. Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Example: `"My diagnostic message."`

Example: `@() datetime('now')`

Examples

Assert Method Was Called with Certain Input

Create a mock with an implicit interface for interactive use. The interface includes `foo` and `bar` methods. Call `foo` by specifying its input.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedMethods',["foo","bar"]);
mock.foo(123);
```

Assert that `foo` was called with input 123.

```
testCase.assertCalled(behavior.foo(123));
```

```
Assertion passed.
```

Display Diagnostic When Assertion Fails

Assert that `foo` was called with input 456. Display the diagnostic information upon failure.

```
testCase.assertCalled(behavior.foo(456), ...
    'Method foo should have been called with input 456.');
```

```
Assertion failed.
    ----------------
```

```
    Test Diagnostic:
    ----------------
    Method foo should have been called with input 456.
    ----------------------
    Framework Diagnostic:
    ----------------------
    assertCalled failed.
    --> Method 'foo' was not called with the specified signature.
    --> All observed method call(s) with any signature are:
            foo([1×1 matlab.mock.classes.Mock], 123)

    Specified method call:
    MethodCallBehavior
        [...] = foo(<Mock>, 456)
Assertion failed.
```

Assert that `bar` was called at least once with only the object as an input.

```
testCase.assertCalled(withExactInputs(behavior.bar), ...
    'Method bar should have been called.');
```

```
Assertion failed.
    ----------------
    Test Diagnostic:
    ----------------
    Method bar should have been called.
    ----------------------
    Framework Diagnostic:
    ----------------------
    assertCalled failed.
    --> Method 'bar' was never called.

    Specified method call:
    MethodCallBehavior
        [...] = bar(<Mock>)
Assertion failed.
```

Tips

Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point marks the current test method as failed and incomplete.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

Alternatives

Using the `assertCalled` method is functionally equivalent to using the `matlab.mock.constraints.WasCalled` constraint with the `assertThat` method of the `Assertable` class. For example, the following code blocks are functionally equivalent.

```
% Using the assertCalled method
testCase.assertCalled(behavior.foo(123), ...
    'Method foo should have been called with input 123.')

% Using the WasCalled constraint with assertThat method
import matlab.mock.constraints.WasCalled;
testCase.assertThat(behavior.foo(123),WasCalled, ...
    'Method foo should have been called with input 123.');
```

However, there is more functionality when you use the `WasCalled` constraint. For instance, you can specify a method was called a certain number of times.

See Also

`matlab.mock.constraints.WasCalled`

Introduced in R2017a

assertSet

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Assert that a property was set

Syntax

```
assertSet(testcase,behavior)
assertSet(testcase,behavior,diagnostic)
```

Description

`assertSet(testcase,behavior)` asserts that a property was set.

`assertSet(testcase,behavior,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.PropertyBehavior` instance

Behavior of the mock, specified as a `matlab.mock.PropertyBehavior` instance. To create an instance of `matlab.mock.PropertyBehavior`, access a property of the behavior object.

Example: `behavior.PropertyFoo`

`diagnostic` — Diagnostic information to display
string array | character array | function handle | `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic information to display, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` object. Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Example: `"My diagnostic message."`

Example: `@() datetime('now')`

Examples

Assert Property Was Set

Create a mock with an implicit interface for interactive use. The interface includes `PropertyFoo` and `PropertyBar` properties. Assign a value to `PropertyFoo`.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties', ...
    ["PropertyFoo","PropertyBar"]);
mock.PropertyFoo = 123;
```

Assert that `PropertyFoo` was set.

```
testCase.assertSet(behavior.PropertyFoo);
```

```
Assertion passed.
```

Display Diagnostic When Assertion Fails

Assert that `PropertyBar` was set. Display the diagnostic information upon failure.

```
testCase.assertSet(behavior.PropertyBar, ...
    'PropertyBar should have been set.');
```

```
Assertion failed.
    ----------------
```

```
    Test Diagnostic:
    ----------------
    PropertyBar should have been set.
    ---------------------
    Framework Diagnostic:
    ---------------------
    assertSet failed.
    --> Property 'PropertyBar' was never set.

    Specified property set operation:
    PropertySetBehavior
        <Mock>.PropertyBar = <IsAnything constraint>
Assertion failed.
```

Tips

Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point marks the current test method as failed and incomplete.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

Alternatives

Using the `assertSet` method is functionally equivalent to using the `matlab.mock.constraints.WasSet` constraint with the `assertThat` method of the `Assertable` class. For example, the following code blocks are functionally equivalent.

```
% Using the assertSet method
testCase.assertSet(behavior.PropertyFoo);

% Using the WasSet constraint with assertThat method
import matlab.mock.constraints.WasSet;
testCase.assertThat(behavior.PropertyFoo, WasSet);
```

However, there is more functionality when you use the `WasSet` constraint. For instance, you can specify a property was set a certain number of times.

See Also

`matlab.mock.constraints.WasSet`

Introduced in R2017a

assertNotAccessed

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Assert that a property was not accessed

Syntax

```
assertNotAccessed(testcase,behavior)
assertNotAccessed(testcase,behavior,diagnostic)
```

Description

`assertNotAccessed(testcase,behavior)` asserts that a property was not accessed.

`assertNotAccessed(testcase,behavior,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.PropertyBehavior` instance

Behavior of the mock, specified as a `matlab.mock.PropertyBehavior` instance. To create an instance of `matlab.mock.PropertyBehavior`, access a property of the behavior object.

Example: `behavior.PropertyFoo`

`diagnostic` — Diagnostic information to display
string array | character array | function handle | `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic information to display, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` object. Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Example: `"My diagnostic message."`

Example: `@() datetime('now')`

Examples

Assert Property Was Not Accessed

Create a mock with an implicit interface for interactive use. The interface includes `PropertyFoo` and `PropertyBar` properties. Access `PropertyFoo` by storing its value.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties',...
    ["PropertyFoo","PropertyBar"]);
value = mock.PropertyFoo;
```

Assert that `PropertyBar` was not accessed.

```
testCase.assertNotAccessed(behavior.PropertyBar);
```

```
Assertion passed.
```

Display Diagnostic When Assertion Fails

Assert that `PropertyFoo` was not accessed. Display the diagnostic information upon failure.

```
testCase.assertNotAccessed(behavior.PropertyFoo, ...
    'PropertyFoo should not have been accessed.');
```

```
Assertion failed.
    ----------------
```

```
    Test Diagnostic:
    ----------------
    PropertyFoo should not have been accessed.
    ----------------------
    Framework Diagnostic:
    ----------------------
    assertNotAccessed failed.
    --> Property 'PropertyFoo' was unexpectedly accessed 1 time(s).

    Specified property access operation:
    PropertyGetBehavior
        <Mock>.PropertyFoo
Assertion failed.
```

Tips

Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point marks the current test method as failed and incomplete.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

Alternatives

Using the `assertNotAccessed` method is functionally equivalent to using the negated `matlab.mock.constraints.WasAccessed` constraint with the `assertThat` method of the `Assertable` class. For example, the following code blocks are functionally equivalent.

```
% Using the assertNotAccessed method
testCase.assertNotAccessed(behavior.PropertyBar);

% Using the WasAccessed constraint with assertThat method
import matlab.mock.constraints.WasAccessed;
testCase.assertThat(behavior.PropertyBar,~WasAccessed);
```

However, there is more functionality when you use the `WasAccessed` constraint. For instance, you can specify a property was not accessed a certain number of times.

See Also

`matlab.mock.constraints.WasAccessed`

Introduced in R2017a

assertNotCalled

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Assert that a method was not called with certain input values

Syntax

```
assertNotCalled(testcase,behavior)
assertNotCalled(testcase,behavior,diagnostic)
```

Description

`assertNotCalled(testcase,behavior)` asserts that a method was not called with certain input values.

`assertNotCalled(testcase,behavior,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.MethodCallBehavior` instance

Behavior of the mock, specified as a `matlab.mock.MethodCallBehavior` instance. To create an instance of `matlab.mock.MethodCallBehavior`, call a method of the behavior object.

Example: `withExactInputs(myMockBehavior.myMockedMethod)`

`diagnostic` — Diagnostic information to display
string array | character array | function handle | `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic information to display, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` object. Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Example: `"My diagnostic message."`

Example: `@() datetime('now')`

Examples

Assert Method Was Not Called with Certain Input

Create a mock with an implicit interface for interactive use. The interface includes a `foo` method. Call the method by specifying its input.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedMethods',"foo");
mock.foo(123);
```

Assert that `foo` was not called with input `456`.

```
testCase.assertNotCalled(behavior.foo(456));
```

```
Assertion passed.
```

Assert that `foo` was not called with only the object as an input.

```
testCase.assertNotCalled(withExactInputs(behavior.foo));
```

```
Assertion passed.
```

Display Diagnostic When Assertion Fails

Assert that `foo` was not called with input `123`. Display the diagnostic information upon failure.

```
testCase.assertNotCalled(behavior.foo(123), ...
    'Method foo should not have been called with input 123.');
```

```
Assertion failed.
    ----------------
    Test Diagnostic:
    ----------------
    Method foo should not have been called with input 123.
    ----------------------
    Framework Diagnostic:
    ----------------------
    assertNotCalled failed.
    --> Method 'foo' was unexpectedly called 1 time(s) with the specified signature.
    --> All observed method call(s) with any signature are:
            foo([1×1 matlab.mock.classes.Mock], 123)

    Specified method call:
    MethodCallBehavior
        [...] = foo(<Mock>, 123)
Assertion failed.
```

Tips

Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point marks the current test method as failed and incomplete.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

Alternatives

Using the `assertNotCalled` method is functionally equivalent to using the negated `matlab.mock.constraints.WasCalled` constraint with the `assertThat` method of the `Assertable` class. For example, the following code blocks are functionally equivalent.

```
% Using the assertNotCalled method
testCase.assertNotCalled(behavior.foo(456), ...
    'Method foo should not have been called with input 456.')

% Using the WasCalled constraint with assertThat method
import matlab.mock.constraints.WasCalled;
testCase.assertThat(behavior.foo(456),~WasCalled, ...
    'Method foo should not have been called with input 456.')
```

However, there is more functionality when you use the `WasCalled` constraint. For instance, you can specify a method was not called a certain number of times.

See Also

`matlab.mock.constraints.WasCalled`

Introduced in R2017a

assertNotSet

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Assert that a property was not set

Syntax

```
assertNotSet(testcase,behavior)
assertNotSet(testcase,behavior,diagnostic)
```

Description

`assertNotSet(testcase,behavior)` asserts that a property was not set.

`assertNotSet(testcase,behavior,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.PropertyBehavior` instance

Behavior of the mock, specified as a `matlab.mock.PropertyBehavior` instance. To create an instance of `matlab.mock.PropertyBehavior`, access a property of the behavior object.

Example: `behavior.PropertyFoo`

`diagnostic` — Diagnostic information to display
string array | character array | function handle | `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic information to display, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` object. Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Example: `"My diagnostic message."`

Example: `@() datetime('now')`

Examples

Assert Property Was Not Set

Create a mock with an implicit interface for interactive use. The interface includes `PropertyFoo` and `PropertyBar` properties. Assign a value to `PropertyFoo`.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties', ...
    ["PropertyFoo","PropertyBar"]);
mock.PropertyFoo = 123;
```

Assert that `PropertyBar` was not set.

```
testCase.assertNotSet(behavior.PropertyBar);
```

```
Assertion passed.
```

Display Diagnostic When Assertion Fails

Assert that `PropertyFoo` was not set. Display the diagnostic information upon failure.

```
testCase.assertNotSet(behavior.PropertyFoo, ...
    'PropertyFoo should not have been set.');
```

```
Assertion failed.
    ----------------
```

```
    Test Diagnostic:
    ----------------
    PropertyFoo should not have been set.
    ---------------------
    Framework Diagnostic:
    ---------------------
    assertNotSet failed.
    --> Property 'PropertyFoo' was unexpectedly set to the specified value 1 time(s).
    --> All observed property set operation(s) where the property was set to any value
            <Mock>.PropertyFoo = 123

    Specified property set operation:
    PropertySetBehavior
        <Mock>.PropertyFoo = <IsAnything constraint>
Assertion failed.
```

Tips

Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point marks the current test method as failed and incomplete.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

Alternatives

Using the `assertNotSet` method is functionally equivalent to using the negated `matlab.mock.constraints.WasSet` constraint with the `assertThat` method of the `Assertable` class. For example, the following code blocks are functionally equivalent.

```
% Using the assertNotSet method
testCase.assertNotSet(behavior.PropertyBar);

% Using the WasSet constraint with assertThat method
import matlab.mock.constraints.WasSet;
testCase.assertThat(behavior.PropertyBar,~WasSet);
```

However, there is more functionality when you use the `WasSet` constraint. For instance, you can specify a property was not set a certain number of times.

See Also

`matlab.mock.constraints.WasSet`

Introduced in R2017a

assignOutputsWhen

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Define return values for method call or property access

Syntax

```
assignOutputsWhen(testcase,behavior,A1,...,An)
```

Description

`assignOutputsWhen(testcase,behavior,A1,...,An)` defines values to return for a method called or property accessed.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.MethodCallBehavior` instance | `matlab.mock.PropertyGetBehavior` instance

Behavior of mock, specified as a `matlab.mock.MethodCallBehavior` or a `matlab.mock.PropertyGetBehavior` instance. To create an instance of `matlab.mock.MethodCallBehavior`, call a method of the behavior object. To create an instance of `matlab.mock.PropertyGetBehavior`, call the `get` method on a property of the behavior object.

Example: `get(behavior.MyMockedProperty)`

Example: `withExactInputs(behavior.myMockedMethod)`

A1,...,An — Defined return values
scalar | vector | matrix | multidimensional array

Defined return values, specified as scalars, vectors, matrices, or multidimensional arrays. Return values can be any data type, and relate to the property or method specified by `behavior`.

Example: `"mySpecifiedValue"`

Example: `7,13,42`

Example: `[1 2 3;4 5 6]`

Examples

Assign Outputs

Assign outputs.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock, behavior] = testCase.createMock('AddedProperties',"PropertyFoo", ...
    'AddedMethods',"methodBar");
testCase.assignOutputsWhen(get(behavior.PropertyFoo), 'abc');
testCase.assignOutputsWhen(withExactInputs(behavior.methodBar), 1, 2, 3);

% Carry out actions
mock.PropertyFoo
[out1,out2,out3] = mock.methodBar
```

Alternatives

Using the `assignOutputsWhen` method is functionally equivalent to using the `matlab.mock.actions.AssignOutputs` action with the `when` method of the `MethodCallBehavior` or `PropertyGetBehavior` class. For example, the following code blocks are functionally equivalent.

```
% Using the assignOutputsWhen method
testCase.assignOutputsWhen(get(behavior.PropertyFoo),'abc');
testCase.assignOutputsWhen(withExactInputs(behavior.methodBar),1,2,3);
```

```
% Using the AssignOutputs action with the when function
import matlab.mock.actions.AssignOutputs;
when(get(behavior.PropertyFoo),AssignOutputs('abc'));
when(withExactInputs(behavior.methodBar),AssignOutputs(1,2,3));
```

However, there is more functionality when you use the `AssignOutputs` action. For instance, you can specify different subsequent behavior for the same mocked object interaction.

See Also

`matlab.mock.actions.AssignOutputs`

Introduced in R2017a

assumeAccessed

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Assume that a property was accessed

Syntax

```
assumeAccessed(testcase,behavior)
assumeAccessed(testcase,behavior,diagnostic)
```

Description

`assumeAccessed(testcase,behavior)` assumes that a property was accessed.

`assumeAccessed(testcase,behavior,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.PropertyBehavior` instance

Behavior of the mock, specified as a `matlab.mock.PropertyBehavior` instance. To create an instance of `matlab.mock.PropertyBehavior`, access a property of the behavior object.

Example: `behavior.PropertyFoo`

`diagnostic` — Diagnostic information to display
string array | character array | function handle | `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic information to display, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` object. Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Example: `"My diagnostic message."`

Example: `@() datetime('now')`

Examples

Assume Property Was Accessed

Create a mock with an implicit interface for interactive use. The interface includes `PropertyFoo` and `PropertyBar` properties. Access `PropertyFoo` by storing its value.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties', ...
    ["PropertyFoo","PropertyBar"]);
value = mock.PropertyFoo;
```

Assume that `PropertyFoo` was accessed.

```
testCase.assumeAccessed(behavior.PropertyFoo);
```

```
An assumption was met.
```

Display Diagnostic When Assumption Fails

Assume that `PropertyBar` was accessed. Display the diagnostic information upon failure.

```
testCase.assumeAccessed(behavior.PropertyBar, ...
    'PropertyBar should have been accessed.');
```

```
An assumption was not met.
    ----------------
```

```
    Test Diagnostic:
    ----------------
    PropertyBar should have been accessed.
    ----------------------
    Framework Diagnostic:
    ----------------------
    assumeAccessed failed.
    --> Property 'PropertyBar' was never accessed.

    Specified property access operation:
    PropertyGetBehavior
        <Mock>.PropertyBar
Assumption failed.
```

Tips

Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point marks the current test method as failed and incomplete.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

Alternatives

Using the `assumeAccessed` method is functionally equivalent to using the `matlab.mock.constraints.WasAccessed` constraint with the `assumeThat` method of the `Assumable` class. For example, the following code blocks are functionally equivalent.

```
% Using the assumeAccessed method
testCase.assumeAccessed(behavior.PropertyFoo);

% Using the WasAccessed constraint with assumeThat method
import matlab.mock.constraints.WasAccessed;
testCase.assumeThat(behavior.PropertyFoo,WasAccessed);
```

However, there is more functionality when you use the `WasAccessed` constraint. For instance, you can specify a property was accessed a certain number of times.

See Also

`matlab.mock.constraints.WasAccessed`

Introduced in R2017a

assumeCalled

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Assume that a method was called with certain input values

Syntax

```
assumeCalled(testcase,behavior)
assumeCalled(testcase,behavior,diagnostic)
```

Description

`assumeCalled(testcase,behavior)` assumes that a method was called with certain input values.

`assumeCalled(testcase,behavior,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.MethodCallBehavior` instance

Behavior of the mock, specified as a `matlab.mock.MethodCallBehavior` instance. To create an instance of `matlab.mock.MethodCallBehavior`, call a method of the behavior object.

Example: `withExactInputs(myMockBehavior.myMockedMethod)`

`diagnostic` — Diagnostic information to display
string array | character array | function handle | `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic information to display, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` object. Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Example: `"My diagnostic message."`

Example: `@() datetime('now')`

Examples

Assume Method Was Called with Certain Input

Create a mock with an implicit interface for interactive use. The interface includes `foo` and `bar` methods. Call `foo` by specifying its input.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedMethods',["foo","bar"]);
mock.foo(123);
```

Assume that `foo` was called with input 123.

```
testCase.assumeCalled(behavior.foo(123));
```

```
An assumption was met.
```

Display Diagnostic When Assumption Fails

Assume that `foo` was called with input 456. Display the diagnostic information upon failure.

```
testCase.assumeCalled(behavior.foo(456), ...
    'Method foo should have been called with input 456.');
```

```
An assumption was not met.
    ----------------
```

```
    Test Diagnostic:
    ----------------
    Method foo should have been called with input 456.
    ----------------------
    Framework Diagnostic:
    ----------------------
    assumeCalled failed.
    --> Method 'foo' was not called with the specified signature.
    --> All observed method call(s) with any signature are:
            foo([1×1 matlab.mock.classes.Mock], 123)

    Specified method call:
    MethodCallBehavior
        [...] = foo(<Mock>, 456)
Assumption failed.
```

Assume that `bar` was called at least once with only the object as an input.

```
testCase.assumeCalled(withExactInputs(behavior.bar), ...
    'Method bar should have been called.');
```

```
An assumption was not met.
    ----------------
    Test Diagnostic:
    ----------------
    Method bar should have been called.
    ----------------------
    Framework Diagnostic:
    ----------------------
    assumeCalled failed.
    --> Method 'bar' was never called.

    Specified method call:
    MethodCallBehavior
        [...] = bar(<Mock>)
Assumption failed.
```

Tips

Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point marks the current test method as failed and incomplete.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

Alternatives

Using the `assumeCalled` method is functionally equivalent to using the `matlab.mock.constraints.WasCalled` constraint with the `assumeThat` method of the `Assumable` class. For example, the following code blocks are functionally equivalent.

```
% Using the assumeCalled method
testCase.assumeCalled(behavior.foo(123), ...
    'Method foo should have been called with input 123.')

% Using the WasCalled constraint with assumeThat method
import matlab.mock.constraints.WasCalled;
testCase.assumeThat(behavior.foo(123),WasCalled, ...
    'Method foo should have been called with input 123.');
```

However, there is more functionality when you use the `WasCalled` constraint. For instance, you can specify a method was called a certain number of times.

See Also

`matlab.mock.constraints.WasCalled`

Introduced in R2017a

assumeSet

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Assume that a property was set

Syntax

```
assumeSet(testcase,behavior)
assumeSet(testcase,behavior,diagnostic)
```

Description

`assumeSet(testcase,behavior)` assumes that a property was set.

`assumeSet(testcase,behavior,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.PropertyBehavior` instance

Behavior of the mock, specified as a `matlab.mock.PropertyBehavior` instance. To create an instance of `matlab.mock.PropertyBehavior`, access a property of the behavior object.

Example: `behavior.PropertyFoo`

`diagnostic` — Diagnostic information to display
string array | character array | function handle | `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic information to display, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` object. Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Example: `"My diagnostic message."`

Example: `@() datetime('now')`

Examples

Assume Property Was Set

Create a mock with an implicit interface for interactive use. The interface includes `PropertyFoo` and `PropertyBar` properties. Assign a value to `PropertyFoo`.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties', ...
    ["PropertyFoo","PropertyBar"]);
mock.PropertyFoo = 123;
```

Assume that `PropertyFoo` was set.

```
testCase.assumeSet(behavior.PropertyFoo);
```

```
An assumption was met.
```

Display Diagnostic When Assumption Fails

Assume that `PropertyBar` was set. Display the diagnostic information upon failure.

```
testCase.assumeSet(behavior.PropertyBar, ...
    'PropertyBar should have been set.');
```

```
An assumption was not met.
    ----------------
```

```
    Test Diagnostic:
    ----------------
    PropertyBar should have been set.
    ---------------------
    Framework Diagnostic:
    ---------------------
    assumeSet failed.
    --> Property 'PropertyBar' was never set.

    Specified property set operation:
    PropertySetBehavior
        <Mock>.PropertyBar = <IsAnything constraint>
Assumption failed.
```

Tips

Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point marks the current test method as failed and incomplete.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

Alternatives

Using the `assumeSet` method is functionally equivalent to using the `matlab.mock.constraints.WasSet` constraint with the `assertThat` method of the `Assumable` class. For example, the following code blocks are functionally equivalent.

```
% Using the assertSet method
testCase.assertSet(behavior.PropertyFoo);

% Using the WasSet constraint with assertThat method
import matlab.mock.constraints.WasSet;
testCase.assertThat(behavior.PropertyFoo, WasSet);
```

However, there is more functionality when you use the `WasSet` constraint. For instance, you can specify a property was set a certain number of times.

See Also

`matlab.mock.constraints.WasSet`

Introduced in R2017a

assumeNotAccessed

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Assume that a property was not accessed

Syntax

```
assumeNotAccessed(testcase,behavior)
assumeNotAccessed(testcase,behavior,diagnostic)
```

Description

`assumeNotAccessed(testcase,behavior)` assumes that a property was not accessed.

`assumeNotAccessed(testcase,behavior,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.PropertyBehavior` instance

Behavior of the mock, specified as a `matlab.mock.PropertyBehavior` instance. To create an instance of `matlab.mock.PropertyBehavior`, access a property of the behavior object.

Example: `behavior.PropertyFoo`

`diagnostic` — Diagnostic information to display
string array | character array | function handle | `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic information to display, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` object. Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Example: `"My diagnostic message."`

Example: `@() datetime('now')`

Examples

Assume Property Was Not Accessed

Create a mock with an implicit interface for interactive use. The interface includes `PropertyFoo` and `PropertyBar` properties. Access `PropertyFoo` by storing its value.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties',...
    ["PropertyFoo","PropertyBar"]);
value = mock.PropertyFoo;
```

Assume that `PropertyBar` was not accessed.

```
testCase.assumeNotAccessed(behavior.PropertyBar);
```

```
An assumption was met.
```

Display Diagnostic When Assumption Fails

Assume that `PropertyFoo` was not accessed. Display the diagnostic information upon failure.

```
testCase.assumeNotAccessed(behavior.PropertyFoo, ...
    'PropertyFoo should not have been accessed.');
```

```
An assumption was not met.
    ----------------
```

```
    Test Diagnostic:
    ----------------
    PropertyFoo should not have been accessed.
    ----------------------
    Framework Diagnostic:
    ----------------------
    assumeNotAccessed failed.
    --> Property 'PropertyFoo' was unexpectedly accessed 1 time(s).

    Specified property access operation:
    PropertyGetBehavior
        <Mock>.PropertyFoo
Assumption failed.
```

Tips

Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point marks the current test method as failed and incomplete.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

Alternatives

Using the `assumeNotAccessed` method is functionally equivalent to using the negated `matlab.mock.constraints.WasAccessed` constraint with the `assumeThat` method of the `Assumable` class. For example, the following code blocks are functionally equivalent.

```
% Using the assumeNotAccessed method
testCase.assumeNotAccessed(behavior.PropertyBar);

% Using the WasAccessed constraint with assumeThat method
import matlab.mock.constraints.WasAccessed;
testCase.assumeThat(behavior.PropertyBar,~WasAccessed);
```

However, there is more functionality when you use the `WasAccessed` constraint. For instance, you can specify a property was not accessed a certain number of times.

See Also

`matlab.mock.constraints.WasAccessed`

Introduced in R2017a

assumeNotCalled

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Assume that a method was not called with certain input values

Syntax

```
assumeNotCalled(testcase,behavior)
assumeNotCalled(testcase,behavior,diagnostic)
```

Description

`assumeNotCalled(testcase,behavior)` assumes that a method was not called with certain input values.

`assumeNotCalled(testcase,behavior,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.MethodCallBehavior` instance

Behavior of the mock, specified as a `matlab.mock.MethodCallBehavior` instance. To create an instance of `matlab.mock.MethodCallBehavior`, call a method of the behavior object.

Example: `withExactInputs(myMockBehavior.myMockedMethod)`

`diagnostic` — Diagnostic information to display
string array | character array | function handle | `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic information to display, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` object. Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Example: `"My diagnostic message."`

Example: `@() datetime('now')`

Examples

Assume Method Was Not Called with Certain Input

Create a mock with an implicit interface for interactive use. The interface includes a `foo` method. Call the method by specifying its input.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedMethods',"foo");
mock.foo(123);
```

Assume that `foo` was not called with input `456`.

```
testCase.assumeNotCalled(behavior.foo(456));
```

```
An assumption was met.
```

Assume that `foo` was not called with only the object as an input.

```
testCase.assumeNotCalled(withExactInputs(behavior.foo));
```

```
An assumption was met.
```

Display Diagnostic When Assumption Fails

Assume that `foo` was not called with input `123`. Display the diagnostic information upon failure.

```
testCase.assumeNotCalled(behavior.foo(123), ...
    'Method foo should not have been called with input 123.');

An assumption was not met.
    ----------------
    Test Diagnostic:
    ----------------
    Method foo should not have been called with input 123.
    ---------------------
    Framework Diagnostic:
    ---------------------
    assumeNotCalled failed.
    --> Method 'foo' was unexpectedly called 1 time(s) with the specified signature.
    --> All observed method call(s) with any signature are:
            foo([1×1 matlab.mock.classes.Mock], 123)

    Specified method call:
    MethodCallBehavior
        [...] = foo(<Mock>, 123)
Assumption failed.
```

Tips

Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point marks the current test method as failed and incomplete.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does

not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

Alternatives

Using the `assumeNotCalled` method is functionally equivalent to using the negated `matlab.mock.constraints.WasCalled` constraint with the `assumeThat` method of the `Assumable` class. For example, the following code blocks are functionally equivalent.

```
% Using the assumeNotCalled method
testCase.assumeNotCalled(behavior.foo(456), ...
    'Method foo should not have been called with input 456.')

% Using the WasCalled constraint with assumeThat method
import matlab.mock.constraints.WasCalled;
testCase.assumeThat(behavior.foo(456),~WasCalled, ...
    'Method foo should not have been called with input 456.')
```

However, there is more functionality when you use the `WasCalled` constraint. For instance, you can specify a method was not called a certain number of times.

See Also

`matlab.mock.constraints.WasCalled`

Introduced in R2017a

assumeNotSet

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Assume that a property was not set

Syntax

```
assumeNotSet(testcase,behavior)
assumeNotSet(testcase,behavior,diagnostic)
```

Description

`assumeNotSet(testcase,behavior)` assumes that a property was not set.

`assumeNotSet(testcase,behavior,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.PropertyBehavior` instance

Behavior of the mock, specified as a `matlab.mock.PropertyBehavior` instance. To create an instance of `matlab.mock.PropertyBehavior`, access a property of the behavior object.

Example: `behavior.PropertyFoo`

`diagnostic` — Diagnostic information to display
string array | character array | function handle | `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic information to display, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` object. Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Example: `"My diagnostic message."`

Example: `@() datetime('now')`

Examples

Assume Property Was Not Set

Create a mock with an implicit interface for interactive use. The interface includes `PropertyFoo` and `PropertyBar` properties. Assign a value to `PropertyFoo`.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties', ...
    ["PropertyFoo","PropertyBar"]);
mock.PropertyFoo = 123;
```

Assume that `PropertyBar` was not set.

```
testCase.assumeNotSet(behavior.PropertyBar);
```

```
An assumption was met.
```

Display Diagnostic When Assumption Fails

Assert that `PropertyFoo` was not set. Display the diagnostic information upon failure.

```
testCase.assumeNotSet(behavior.PropertyFoo, ...
    'PropertyFoo should not have been set.');
```

```
An assumption was not met.
    ----------------
```

```
    Test Diagnostic:
    ----------------
    PropertyFoo should not have been set.
    ---------------------
    Framework Diagnostic:
    ---------------------
    assumeNotSet failed.
    --> Property 'PropertyFoo' was unexpectedly set to the specified value 1 time(s).
    --> All observed property set operation(s) where the property was set to any value
            <Mock>.PropertyFoo = 123

    Specified property set operation:
    PropertySetBehavior
        <Mock>.PropertyFoo = <IsAnything constraint>
Assumption failed.
```

Tips

Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point marks the current test method as failed and incomplete.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

Alternatives

Using the `assumeNotSet` method is functionally equivalent to using the negated `matlab.mock.constraints.WasSet` constraint with the `assumeThat` method of the `Assumable` class. For example, the following code blocks are functionally equivalent.

```
% Using the assumeNotSet method
testCase.assumeNotSet(behavior.PropertyBar);

% Using the WasSet constraint with assumeThat method
import matlab.mock.constraints.WasSet;
testCase.assumeThat(behavior.PropertyBar,~WasSet);
```

However, there is more functionality when you use the `WasSet` constraint. For instance, you can specify a property was not set a certain number of times.

See Also

`matlab.mock.constraints.WasSet`

Introduced in R2017a

clearMockHistory

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Clear history of mock object interactions

Syntax

```
clearMockHistory(testCase,mock)
```

Description

`clearMockHistory(testCase,mock)` clears the history of recorded mock object interactions. The `clearMockHistory` method does not clear mock object behaviors. To clear both interactions and behaviors, create a new mock.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`mock` — Mock to clear history
mock object

Mock to clear history of interactions from, specified as a mock object.

Examples

Clear Mock History

Construct a mock with a `myMethod` method.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock, behavior] = testCase.createMock("AddedMethods","myMethod");
```

Interact with the mock by calling the method. Then verify that the method was called.

```
mock.myMethod('abc');
testCase.verifyCalled(behavior.myMethod('abc'))
```

```
Verification passed.
```

View the interaction history.

```
h = testCase.getMockHistory(mock)
```

```
h = 

  SuccessfulMethodCall with properties:

       Name: "myMethod"
     Inputs: {[1×1 matlab.mock.classes.Mock]  'abc'}
    Outputs: {[]}

Interaction summary:
  myMethod([1×1 matlab.mock.classes.Mock], 'abc')
```

Clear the history of the recorded interaction and retest whether the method was called. The verification now fails.

```
testCase.clearMockHistory(mock)
testCase.verifyCalled(behavior.myMethod('abc'))
```

```
Verification failed.

    ----------------------
    Framework Diagnostic:
    ----------------------
    verifyCalled failed.
    --> Method 'myMethod' was never called.

    Specified method call:
    MethodCallBehavior
        [...] = myMethod(<Mock>, 'abc')
```

View the interaction history again. It is empty.

```
h = testCase.getMockHistory(mock)
```

```
h =

  1×0 InteractionHistory array with properties:

    Name
```

See Also

matlab.mock.history

Introduced in R2018b

createMock

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Create mock object

Syntax

```
[mock,behavior] = createMock(testcase)
[mock,behavior] = createMock(testcase,superclass)
[mock,behavior] = createMock( ___ ,Name,Value)
```

Description

`[mock,behavior] = createMock(testcase)` creates a mock object and an associated behavior object.

`[mock,behavior] = createMock(testcase,superclass)` creates a mock that derives from the `superclass` class.

`[mock,behavior] = createMock( ___ ,Name,Value)` creates a mock with additional options specified by one or more `Name,Value` pair arguments. You can use this syntax with any of the arguments from the previous syntaxes.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`superclass` — Superclass for mock
scalar `meta.class` object

Superclass for mock, specified as a scalar `meta.class` object. The mock object implements all the abstract properties and methods of this class.

Example: `?MyIterfaceClass`

Example: `?MException`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

For example, `[mock,behavior] = testCase.createMock('AddedProperties', {'Prop1','Prop2'})` creates a mock and adds the `Prop1` and `Prop2` properties to it.

`AddedMethods` — Names of methods to add to mock
string array | cell array of character vectors

Names of methods to add to the mock, specified as a string array or a cell array of character vectors. Unless the mock is strict or the mock behavior has been defined, calling these methods on the mock returns an empty array.

Example: `["methodA","methodB","methodC"]`

Example: `{'method1','method2'}`

`AddedProperties` — Properties to add to mock
string array | cell array of character vectors

Names of properties to add to the mock, specified as a string array or a cell array of character vectors. If a mock is not strict, you can set and get their values. However, if a mock is strict, by default MATLAB produces an assertion failure if you set or get a property value.

Example: `"MyProperty"`

Example: `{'Prop1','Prop2'}`

`DefaultPropertyValues` — Default property values
scalar struct

Default property values, specified as a scalar struct. Use this name-value pair argument to specify default values for properties implemented by the mock object class. These

properties include `Abstract` superclass properties and properties added with the `'AddedProperties'` name-value pair argument. Each field refers to the name of a property implemented on the mock class, and the corresponding value represents the default value for that property.

Example: `struct('PropA',123,'PropB',true)`

`MockedMethods` — Methods to mock
mock all possible methods (default) | `string.empty` | `{}` | string array | `cellstr`

Methods to mock, specified using the method names in a string array or cell array of character vectors. To specify that no methods are mocked, use an empty value specified as `string.empty`, or `{}`. By default, all methods are mocked.

`MockedMethods` can include any subset of added methods, abstract superclass methods, and concrete superclass methods that can be overridden (`Sealed` attribute value of `false`). In general, you include only those methods that you want to stub or spy on.

Specifying `MockedMethods` enables tests to mock only those methods that are important to the test case. Limiting the methods that are mocked can improve test performance when superclasses define many methods.

Example: ["foo", "bar"]

Data Types: `char` | `string` | `cell`

`Strict` — Indicator if mock is strict
`false` (default) | `true`

Indicator if mock is strict, specified as `false` or `true`. By default, a mock method returns an empty array if the behavior is undefined. If you set `Strict` to `true`, the framework produces an assertion failure for undefined behavior for

- All `abstract` methods and properties of the specified interface.
- Methods added to the mock with the `AddedMethods` argument.
- Properties added to the mock with the `AddedProperties` argument.

Data Types: `logical`

`ConstructorInputs` — Inputs to pass to `superclass` constructor
cell array of values

Inputs to pass to the `superclass` constructor, specified as a cell array of values.

Example: If you construct a mock where you define `superclass` to be `?MException`, `'ConstructorInputs'` could be `{'My:ID','My message'}`.

Output Arguments

`mock` — Implementation of abstract methods and properties
mock object

Implementation of the abstract methods and properties of the interface specified by the `superclass` input, returned as a mock object. If a mock is constructed without defining a superclass, it does not have an explicit interface.

Note: You cannot save and load mock objects.

`behavior` — Definition of mock behavior
behavior object

Definition of the mock behavior, returned as a behavior object. Use `behavior` to define mock actions and verify interactions.

Note: You cannot save and load behavior objects.

Examples

Construct Mocks

Construct a strict mock.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedMethods',"foo",'Strict',true);
```

Construct a mock with specific methods.

```
[mock,behavior] = testCase.createMock('AddedMethods', ...
    {'one', 'two', 'three'});
```

Construct a mock with constructor inputs.

```
[mock,behavior] = testCase.createMock(?MException,'ConstructorInputs', ...
    {'My:ID','My message'});
```

Construct a mock with two properties. `Prop2` has a default value of `false`.

```
mock = testCase.createMock('AddedProperties',{'Prop1','Prop2'},...
    'DefaultPropertyValues',struct('Prop2',false))
```

```
mock =

  Mock with properties:

    Prop1: []
    Prop2: 0
```

Construct a mock that overrides the `isnan` and `isinf` methods of the class `double`.

```
[mock,behavior] = testCase.createMock(?double,"MockedMethods",["isnan","isinf"],...
    "ConstructorInputs",{123});
```

See Also

Topics

“Create Mock Object”
“Specify Mock Object Behavior”
“Qualify Mock Object Interaction”
“Write Test That Uses App Testing and Mocking Frameworks”

Introduced in R2017a

fatalAssertAccessed

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Fatally assert that a property was accessed

Syntax

```
fatalAssertAccessed(testcase,behavior)
fatalAssertAccessed(testcase,behavior,diagnostic)
```

Description

`fatalAssertAccessed(testcase,behavior)` fatally asserts that a property was accessed.

`fatalAssertAccessed(testcase,behavior,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.PropertyBehavior` instance

Behavior of the mock, specified as a `matlab.mock.PropertyBehavior` instance. To create an instance of `matlab.mock.PropertyBehavior`, access a property of the behavior object.

Example: `behavior.PropertyFoo`

`diagnostic` — Diagnostic information to display
string array | character array | function handle | `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic information to display, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` object. Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Example: `"My diagnostic message."`

Example: `@() datetime('now')`

Examples

Fatally Assert Property Was Accessed

Create a mock with an implicit interface for interactive use. The interface includes `PropertyFoo` and `PropertyBar` properties. Access `PropertyFoo` by storing its value.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties', ...
    ["PropertyFoo","PropertyBar"]);
value = mock.PropertyFoo;
```

Fatally assert that `PropertyFoo` was accessed.

```
testCase.fatalAssertAccessed(behavior.PropertyFoo);
```

```
Fatal assertion passed.
```

Display Diagnostic When Fatal Assertion Fails

Fatally assert that `PropertyBar` was accessed. Display the diagnostic information upon failure.

```
testCase.fatalAssertAccessed(behavior.PropertyBar, ...
    'PropertyBar should have been accessed.');
```

```
Fatal assertion failed.
    ----------------
```

```
    Test Diagnostic:
    ----------------
    PropertyBar should have been accessed.
    ----------------------
    Framework Diagnostic:
    ----------------------
    fatalAssertAccessed failed.
    --> Property 'PropertyBar' was never accessed.

    Specified property access operation:
    PropertyGetBehavior
        <Mock>.PropertyBar
Fatal assertion failed.
```

Tips

Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point marks the current test method as failed and incomplete.

Alternatives

Using the `fatalAssertAccessed` method is functionally equivalent to using the `matlab.mock.constraints.WasAccessed` constraint with the `fatalAssertThat` method of the `FatalAssertable` class. For example, the following code blocks are functionally equivalent.

```
% Using the fatalAssertAccessed method
testCase.fatalAssertAccessed(behavior.PropertyFoo);

% Using the WasAccessed constraint with fatalAssertThat method
import matlab.mock.constraints.WasAccessed;
testCase.fatalAssertThat(behavior.PropertyFoo,WasAccessed);
```

However, there is more functionality when you use the `WasAccessed` constraint. For instance, you can specify a property was accessed a certain number of times.

See Also

`matlab.mock.constraints.WasAccessed`

Introduced in R2017a

fatalAssertCalled

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Fatally assert that a method was called with certain input values

Syntax

```
fatalAssertCalled(testcase,behavior)
fatalAssertCalled(testcase,behavior,diagnostic)
```

Description

`fatalAssertCalled(testcase,behavior)` fatally asserts that a method was called with certain input values.

`fatalAssertCalled(testcase,behavior,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.MethodCallBehavior` instance

Behavior of the mock, specified as a `matlab.mock.MethodCallBehavior` instance. To create an instance of `matlab.mock.MethodCallBehavior`, call a method of the behavior object.

Example: `withExactInputs(myMockBehavior.myMockedMethod)`

`diagnostic` — Diagnostic information to display
string array | character array | function handle | `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic information to display, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` object. Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Example: `"My diagnostic message."`

Example: `@() datetime('now')`

Examples

Fatally Assert Method Was Called with Certain Input

Create a mock with an implicit interface for interactive use. The interface includes `foo` and `bar` methods. Call `foo` by specifying its input.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedMethods',["foo","bar"]);
mock.foo(123);
```

Fatally assert that `foo` was called with input `123`.

```
testCase.fatalAssertCalled(behavior.foo(123));
```

```
Fatal assertion passed.
```

Display Diagnostic When Fatal Assertion Fails

Fatally assert that `foo` was called with input `456`. Display the diagnostic information upon failure.

```
testCase.fatalAssertCalled(behavior.foo(456), ...
    'Method foo should have been called with input 456.');
```

```
Fatal assertion failed.
    ----------------
```

```
    Test Diagnostic:
    ----------------
    Method foo should have been called with input 456.
    ----------------------
    Framework Diagnostic:
    ----------------------
    fatalAssertCalled failed.
    --> Method 'foo' was not called with the specified signature.
    --> All observed method call(s) with any signature are:
            foo([1×1 matlab.mock.classes.Mock], 123)

    Specified method call:
    MethodCallBehavior
        [...] = foo(<Mock>, 456)
Fatal assertion failed.
```

Fatally assert that `bar` was called at least once with only the object as an input.

```
testCase.fatalAssertCalled(withExactInputs(behavior.bar), ...
    'Method bar should have been called.');
```

```
Fatal assertion failed.
    ----------------
    Test Diagnostic:
    ----------------
    Method bar should have been called.
    ----------------------
    Framework Diagnostic:
    ----------------------
    fatalAssertCalled failed.
    --> Method 'bar' was never called.

    Specified method call:
    MethodCallBehavior
        [...] = bar(<Mock>)
Fatal assertion failed.
```

Tips

Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not

restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point marks the current test method as failed and incomplete.

Alternatives

Using the `fatalAssertCalled` method is functionally equivalent to using the `matlab.mock.constraints.WasCalled` constraint with the `fatalAssertThat` method of the `FatalAssertable` class. For example, the following code blocks are functionally equivalent.

```
% Using the fatalAssertCalled method
testCase.fatalAssertCalled(behavior.foo(123), ...
    'Method foo should have been called with input 123.')

% Using the WasCalled constraint with fatalAssertThat method
import matlab.mock.constraints.WasCalled;
testCase.fatalAssertThat(behavior.foo(123),WasCalled, ...
    'Method foo should have been called with input 123.');
```

However, there is more functionality when you use the `WasCalled` constraint. For instance, you can specify a method was called a certain number of times.

See Also

`matlab.mock.constraints.WasCalled`

Introduced in R2017a

fatalAssertSet

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Fatally assert that a property was set

Syntax

```
fatalAssertSet(testcase,behavior)
fatalAssertSet(testcase,behavior,diagnostic)
```

Description

`fatalAssertSet(testcase,behavior)` fatally asserts that a property was set.

`fatalAssertSet(testcase,behavior,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.PropertyBehavior` instance

Behavior of the mock, specified as a `matlab.mock.PropertyBehavior` instance. To create an instance of `matlab.mock.PropertyBehavior`, access a property of the behavior object.

Example: `behavior.PropertyFoo`

`diagnostic` — Diagnostic information to display
string array | character array | function handle | `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic information to display, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` object. Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Example: `"My diagnostic message."`

Example: `@() datetime('now')`

Examples

Fatally Assert Property Was Set

Create a mock with an implicit interface for interactive use. The interface includes `PropertyFoo` and `PropertyBar` properties. Assign a value to `PropertyFoo`.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties', ...
    ["PropertyFoo","PropertyBar"]);
mock.PropertyFoo = 123;
```

Fatally assert that `PropertyFoo` was set.

```
testCase.fatalAssertSet(behavior.PropertyFoo);
```

```
Fatal assertion passed.
```

Display Diagnostic When Fatal Assertion Fails

Fatally assert that `PropertyBar` was set. Display the diagnostic information upon failure.

```
testCase.fatalAssertSet(behavior.PropertyBar, ...
    'PropertyBar should have been set.');
```

```
Fatal assertion failed.
    ----------------
```

```
    Test Diagnostic:
    ----------------
    PropertyBar should have been set.
    ---------------------
    Framework Diagnostic:
    ---------------------
    fatalAssertSet failed.
    --> Property 'PropertyBar' was never set.

    Specified property set operation:
    PropertySetBehavior
        <Mock>.PropertyBar = <IsAnything constraint>
Fatal assertion failed.
```

Tips

Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point marks the current test method as failed and incomplete.

Alternatives

Using the `fatalAssertSet` method is functionally equivalent to using the `matlab.mock.constraints.WasSet` constraint with the `fatalAssertThat` method of the `FatalAssertable` class. For example, the following code blocks are functionally equivalent.

```
% Using the fatalAssertSet method
testCase.fatalAssertSet(behavior.PropertyFoo);

% Using the WasSet constraint with fatalAssertThat method
import matlab.mock.constraints.WasSet;
testCase.fatalAssertThat(behavior.PropertyFoo, WasSet);
```

However, there is more functionality when you use the `WasSet` constraint. For instance, you can specify a property was set a certain number of times.

See Also

`matlab.mock.constraints.WasSet`

Introduced in R2017a

fatalAssertNotAccessed

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Fatally assert that a property was not accessed

Syntax

```
fatalAssertNotAccessed(testcase,behavior)
fatalAssertNotAccessed(testcase,behavior,diagnostic)
```

Description

`fatalAssertNotAccessed(testcase,behavior)` fatally asserts that a property was not accessed.

`fatalAssertNotAccessed(testcase,behavior,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.PropertyBehavior` instance

Behavior of the mock, specified as a `matlab.mock.PropertyBehavior` instance. To create an instance of `matlab.mock.PropertyBehavior`, access a property of the behavior object.

Example: `behavior.PropertyFoo`

`diagnostic` — Diagnostic information to display
string array | character array | function handle | `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic information to display, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` object. Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Example: `"My diagnostic message."`

Example: `@() datetime('now')`

Examples

Fatally Assert Property Was Not Accessed

Create a mock with an implicit interface for interactive use. The interface includes `PropertyFoo` and `PropertyBar` properties. Access `PropertyFoo` by storing its value.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties',...
    ["PropertyFoo","PropertyBar"]);
value = mock.PropertyFoo;
```

Fatally assert that `PropertyBar` was not accessed.

```
testCase.fatalAssertNotAccessed(behavior.PropertyBar);
```

```
Fatal assertion passed.
```

Display Diagnostic When Fatal Assertion Fails

Fatally assert that `PropertyFoo` was not accessed. Display the diagnostic information upon failure.

```
testCase.fatalAssertNotAccessed(behavior.PropertyFoo, ...
    'PropertyFoo should not have been accessed.');
```

```
Fatal assertion failed.
    ----------------
```

```
    Test Diagnostic:
    ----------------
    PropertyFoo should not have been accessed.
    ----------------------
    Framework Diagnostic:
    ----------------------
    fatalAssertNotAccessed failed.
    --> Property 'PropertyFoo' was unexpectedly accessed 1 time(s).

    Specified property access operation:
    PropertyGetBehavior
        <Mock>.PropertyFoo
Fatal assertion failed.
```

Tips

Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point marks the current test method as failed and incomplete.

Alternatives

Using the `fatalAssertNotAccessed` method is functionally equivalent to using the negated `matlab.mock.constraints.WasAccessed` constraint with the `fatalAssertThat` method of the `FatalAssertable` class. For example, the following code blocks are functionally equivalent.

```
% Using the fatalAssertNotAccessed method
testCase.fatalAssertNotAccessed(behavior.PropertyBar);

% Using the WasAccessed constraint with fatalAssertThat method
import matlab.mock.constraints.WasAccessed;
testCase.fatalAssertThat(behavior.PropertyBar,~WasAccessed);
```

However, there is more functionality when you use the `WasAccessed` constraint. For instance, you can specify a property was not accessed a certain number of times.

See Also

`matlab.mock.constraints.WasAccessed`

Introduced in R2017a

fatalAssertNotCalled

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Fatally assert that a method was not called with certain input values

Syntax

```
fatalAssertNotCalled(testcase,behavior)
fatalAssertNotCalled(testcase,behavior,diagnostic)
```

Description

`fatalAssertNotCalled(testcase,behavior)` fatally asserts that a method was not called with certain input values.

`fatalAssertNotCalled(testcase,behavior,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.MethodCallBehavior` instance

Behavior of the mock, specified as a `matlab.mock.MethodCallBehavior` instance. To create an instance of `matlab.mock.MethodCallBehavior`, call a method of the behavior object.

Example: `withExactInputs(myMockBehavior.myMockedMethod)`

`diagnostic` — Diagnostic information to display
string array | character array | function handle | `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic information to display, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` object. Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Example: `"My diagnostic message."`

Example: `@() datetime('now')`

Examples

Fatally Assert Method Was Not Called with Certain Input

Create a mock with an implicit interface for interactive use. The interface includes a `foo` method. Call the method by specifying its input.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedMethods',"foo");
mock.foo(123);
```

Fatally assert that `foo` was not called with input `456`.

```
testCase.fatalAssertNotCalled(behavior.foo(456));
```

```
Fatal assertion passed.
```

Fatally assert that `foo` was not called with only the object as an input.

```
testCase.fatalAssertNotCalled(withExactInputs(behavior.foo));
```

```
Fatal assertion passed.
```

Display Diagnostic When Fatal Assertion Fails

Fatally assert that `foo` was not called with input `123`. Display the diagnostic information upon failure.

```
testCase.fatalAssertNotCalled(behavior.foo(123), ...
    'Method foo should not have been called with input 123.');

Fatal assertion failed.
    ----------------
    Test Diagnostic:
    ----------------
    Method foo should not have been called with input 123.
    ---------------------
    Framework Diagnostic:
    ---------------------
    fatalAssertNotCalled failed.
    --> Method 'foo' was unexpectedly called 1 time(s) with the specified signature.
    --> All observed method call(s) with any signature are:
            foo([1×1 matlab.mock.classes.Mock], 123)

    Specified method call:
    MethodCallBehavior
        [...] = foo(<Mock>, 123)
Fatal assertion failed.
```

Tips

Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test

methods. A failure at the assertion point marks the current test method as failed and incomplete.

Alternatives

Using the `fatalAssertNotCalled` method is functionally equivalent to using the negated `matlab.mock.constraints.WasCalled` constraint with the `fatalAssertThat` method of the `FatalAssertable` class. For example, the following code blocks are functionally equivalent.

```
% Using the fatalAssertNotCalled method
testCase.fatalAssertNotCalled(behavior.foo(456), ...
    'Method foo should not have been called with input 456.')

% Using the WasCalled constraint with fatalAssertThat method
import matlab.mock.constraints.WasCalled;
testCase.fatalAssertThat(behavior.foo(456),~WasCalled, ...
    'Method foo should not have been called with input 456.')
```

However, there is more functionality when you use the `WasCalled` constraint. For instance, you can specify a method was not called a certain number of times.

See Also

`matlab.mock.constraints.WasCalled`

Introduced in R2017a

fatalAssertNotSet

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Fatally assert that a property was not set

Syntax

```
fatalAssertNotSet(testcase,behavior)
fatalAssertNotSet(testcase,behavior,diagnostic)
```

Description

`fatalAssertNotSet(testcase,behavior)` fatally asserts that a property was not set.

`fatalAssertNotSet(testcase,behavior,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.PropertyBehavior` instance

Behavior of the mock, specified as a `matlab.mock.PropertyBehavior` instance. To create an instance of `matlab.mock.PropertyBehavior`, access a property of the behavior object.

Example: `behavior.PropertyFoo`

`diagnostic` — Diagnostic information to display
string array | character array | function handle | `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic information to display, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` object. Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Example: `"My diagnostic message."`

Example: `@() datetime('now')`

Examples

Fatally Assert Property Was Not Set

Create a mock with an implicit interface for interactive use. The interface includes `PropertyFoo` and `PropertyBar` properties. Assign a value to `PropertyFoo`.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties', ...
    ["PropertyFoo","PropertyBar"]);
mock.PropertyFoo = 123;
```

Fatally assert that `PropertyBar` was not set.

```
testCase.fatalAssertNotSet(behavior.PropertyBar);
```

```
Fatal assertion passed.
```

Display Diagnostic When Fatal Assertion Fails

Fatally assert that `PropertyFoo` was not set. Display the diagnostic information upon failure.

```
testCase.fatalAssertNotSet(behavior.PropertyFoo, ...
    'PropertyFoo should not have been set.');
```

```
Fatal assertion failed.
    ----------------
```

```
    Test Diagnostic:
    ----------------
    PropertyFoo should not have been set.
    ---------------------
    Framework Diagnostic:
    ---------------------
    fatalAssertNotSet failed.
    --> Property 'PropertyFoo' was unexpectedly set to the specified value 1 time(s).
    --> All observed property set operation(s) where the property was set to any value
            <Mock>.PropertyFoo = 123

    Specified property set operation:
    PropertySetBehavior
        <Mock>.PropertyFoo = <IsAnything constraint>
Fatal assertion failed.
```

Tips

Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point marks the current test method as failed and incomplete.

Alternatives

Using the `fatalAssertNotSet` method is functionally equivalent to using the negated `matlab.mock.constraints.WasSet` constraint with the `fatalAssertThat` method of the `FatalAssertable` class. For example, the following code blocks are functionally equivalent.

```
% Using the fatalAssertNotSet method
testCase.fatalAssertNotSet(behavior.PropertyBar);

% Using the WasSet constraint with fatalAssertThat method
import matlab.mock.constraints.WasSet;
testCase.fatalAssertThat(behavior.PropertyBar,~WasSet);
```

However, there is more functionality when you use the `WasSet` constraint. For instance, you can specify a property was not set a certain number of times.

See Also

`matlab.mock.constraints.WasSet`

Introduced in R2017a

getMockHistory

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Return history from mock object

Syntax

```
history = getMockHistory(testcase,mock)
```

Description

`history = getMockHistory(testcase,mock)` returns the history from a mock object. `history` is an array of `matlab.mock.InteractionHistory` objects. Each element in `history` corresponds to one method call, property access, or property modification. The array elements are ordered, with the first element indicating the first recorded interaction. This method returns interactions with publicly visible methods and properties only. For example, the following interactions are not recorded:

- Calls to `Hidden` methods
- Calls to `Sealed` superclass methods
- Accesses or modifications of concrete superclass properties

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`mock` — Mock to return history
mock object

Mock to return history of interactions, specified as a mock object.

Examples

Obtain History of Mock Interactions

Construct a mock with a `computeValue` method and two properties. Assign a default value of `false` to `Prop2`.

```
tc = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = tc.createMock( ...
    'AddedMethods',{'computeValue'}, ...
    'AddedProperties',{'Prop1','Prop2'}, ...
    'DefaultPropertyValues',struct('Prop2',false));
```

Set up the behavior of the `computeValue` method to return the value 42, regardless of input values.

```
import matlab.mock.actions.AssignOutputs;
when(withAnyInputs(behavior.computeValue),AssignOutputs(42));
```

Interact with the mock. First call the `computeValue` method. Then display the value of `Prop2`. Finally, set the value of `Prop1`.

```
n = mock.computeValue('hello');
mock.Prop2
mock.Prop1 = 13;
```

```
ans =

  logical

   0
```

Obtain the interaction history for the mock.

```
h = tc.getMockHistory(mock)
```

```
h =

  1×3 heterogeneous InteractionHistory (SuccessfulMethodCall, SuccessfulPropertyAccess, SuccessfulPropertyMo

    Name

Interaction summary:
  computeValue([1×1 matlab.mock.classes.Mock], 'hello')
  <Mock>.Prop2
  <Mock>.Prop1 = 13
```

Examine the first `InteractionHistory` object. The method was called with the mock object and the character vector `'hello'` as inputs. The method output the value 42.

```
h(1)
```

```
ans = 

  SuccessfulMethodCall with properties:

       Name: "computeValue"
     Inputs: {[1×1 matlab.mock.classes.Mock]  'hello'}
    Outputs: {[42]}

Interaction summary:
  computeValue([1×1 matlab.mock.classes.Mock], 'hello')
```

Alternatives

You can obtain the same history of interactions using the `matlab.mock.InteractionHistory.forMock` method. For example, if you have a `matlab.mock.TestCase` instance `tc`, and a mock object `mock`, the following method calls are equivalent.

```
h = matlab.mock.InteractionHistory.forMock(mock);
h = tc.getMockHistory(mock);
```

However, you do not need access to the `matlab.mock.TestCase` instance to use the `matlab.mock.InteractionHistory.forMock` method.

See Also

`matlab.mock.history`

Introduced in R2018a

matlab.mock.TestCase.forInteractiveUse

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Create `TestCase` for interactive use

Syntax

```
tc = matlab.mock.TestCase.forInteractiveUse
```

Description

`tc = matlab.mock.TestCase.forInteractiveUse` creates a `matlab.mock.TestCase` instance for interactive use. The `TestCase` is configured so you can experiment with it at the MATLAB command prompt. The `TestCase` reacts to qualification failures and successes by displaying messages to the screen for both passing and failing conditions.

Examples

Construct Interactive TestCase

Create a TestCase configured for interactive use at the MATLAB command prompt.

```
import matlab.mock.TestCase;
testCase = TestCase.forInteractiveUse;
```

Create a mock.

```
[mock,behavior] = testCase.createMock('AddedMethods',"myMethod");
```

Produce a failing verification.

```
testCase.verifyCalled(behavior.myMethod(123));
```

See Also

Introduced in R2017a

returnStoredValueWhen

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Return stored property value

Syntax

```
returnStoredValueWhen(testcase,behavior)
```

Description

`returnStoredValueWhen(testcase,behavior)` specifies that the mock returns the stored property value when a property is accessed. If the mock is strict and the property is an abstract property of the mock interface, the framework produces an assertion failure when it accesses a property. To enable access to the property in a strict mock, use the `returnStoredValueWhen` method.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.PropertyGetBehavior` instance

Behavior of the mock, specified as a `matlab.mock.PropertyGetBehavior` instance. To create an instance of `matlab.mock.PropertyGetBehavior`, call the `get` method on a property of the behavior object.

Example: `get(behavior.MyMockedProperty)`

Examples

Return Property Value

Create a strict mock. All property interactions throw exceptions by default.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties',"PropertyFoo",...
    'Strict',true);
```

Enable `PropertyFoo` to be accessed instead of throwing an exception.

```
testCase.returnStoredValueWhen(get(behavior.PropertyFoo));
```

Alternatives

Using the `returnStoredValueWhen` method is functionally equivalent to using the `matlab.mock.actions.ReturnStoredValue` action with the `when` method of the `PropertyGetBehavior` class. For example, the following code blocks are functionally equivalent.

```
% Using the returnStoredValueWhen method
testCase.returnStoredValueWhen(get(behavior.PropertyFoo));

% Using the ReturnStoredValue action with the when function
import matlab.mock.actions.ReturnStoredValue;
when(get(behavior.PropertyFoo),ReturnStoredValue);
```

However, there is more functionality when you use the `ReturnStoredValue` action. For instance, you can specify different subsequent behavior for the same mocked object interaction.

See Also

`matlab.mock.actions.ReturnStoredValue`

Introduced in R2017a

storeValueWhen

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Store property value

Syntax

```
storeValueWhen(testcase,behavior)
```

Description

`storeValueWhen(testcase,behavior)` specifies that the mock should store the property value when a property is set. If the mock is strict and the property is an abstract property of the mock interface, the framework produces an assertion failure at property set access. To enable the property to be set in a strict mock, use the `storeValueWhen` method.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.PropertySetBehavior` instance

Behavior of the mock, specified as a `matlab.mock.PropertySetBehavior` instance. To create an instance of `matlab.mock.PropertySetBehavior`, call the `set` method on a property of the behavior object.

Example: `set(behavior.MyMockedProperty)`

Examples

Store Property Value

Create a strict mock. All property interactions throw exceptions by default.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties',"PropertyFoo", ...
    'Strict',true);
```

Enable `PropertyFoo` to be set instead of throwing an exception.

```
testCase.storeValueWhen(set(behavior.PropertyFoo));
```

Alternatives

Using the `storeValueWhen` method is functionally equivalent to using the `matlab.mock.actions.StoreValue` action with the `when` method of the `PropertySetBehavior` class. For example, the following code blocks are functionally equivalent.

```
% Using the storeValueWhen method
testCase.storeValueWhen(set(behavior.PropertyFoo));

% Using the StoreValue action with the when function
import matlab.mock.actions.StoreValue;
when(set(behavior.PropertyFoo),StoreValue);
```

However, there is more functionality when you use the `StoreValue` action. For instance, you can specify different subsequent behavior for the same mocked object interaction.

See Also

`matlab.mock.actions.StoreValue`

Introduced in R2017a

throwExceptionWhen

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Throw exception for method call or property interaction

Syntax

```
throwExceptionWhen(testcase,behavior)
throwExceptionWhen(testcase,behavior,exception)
```

Description

`throwExceptionWhen(testcase,behavior)` specifies that the mock should throw an exception when a method is called or a property is accessed or set.

`throwExceptionWhen(testcase,behavior,exception)` specifies the exception that the mock throws.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.MethodCallBehavior` instance |
`matlab.mock.PropertyGetBehavior` instance |
`matlab.mock.PropertySetBehavior` instance

Behavior of the mock, specified as a `matlab.mock.MethodCallBehavior`, `matlab.mock.PropertyGetBehavior`, or `matlab.mock.PropertySetBehavior` instance. To create an instance of `matlab.mock.MethodCallBehavior`, call a method of the behavior object. To create an instance of `matlab.mock.PropertyGetBehavior`,

call the `get` method on a property of the behavior object. To create an instance of `matlab.mock.PropertySetBehavior`, call the `set` method on a property of the behavior object.

Example: `withExactInputs(behavior.myMockedMethod)`

Example: `get(behavior.MyMockedProperty)`

Example: `set(behavior.MyMockedProperty)`

`exception` — Exception to throw at method call or property interaction
scalar `MException` object

Exception for the framework to throw at the method call or property interaction, specified as a scalar `MException` object.

Example: `MException('MyProduct:myID','My exception message.')`

Examples

Throw Exception

Throw an exception when a method is called or when a property is accessed or set.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties',"PropertyFoo", ...
    'AddedMethods',"methodBar");
testCase.throwExceptionWhen(get(behavior.PropertyFoo));
testCase.throwExceptionWhen(set(behavior.PropertyFoo), ...
    MException('PropertyFoo:set', 'Do not change PropertyFoo'));
testCase.throwExceptionWhen(withAnyInputs(behavior.methodBar));

% Carry out actions
mock.PropertyFoo
mock.PropertyFoo = 123;
mock.methodBar;
```

Alternatives

Using the `throwExceptionWhen` method is functionally equivalent to using the `matlab.mock.actions.ThrowException` action with the `when` method of the

`MethodCallBehavior`, `PropertyGetBehavior`, or `PropertySetBehavior` class. For example, the following code blocks are functionally equivalent.

```
% Using the throwExceptionWhen method
testCase.throwExceptionWhen(behavior.deposit(IsLessThan(0)), ...
    MException('Account:deposit:Negative', ...
    'Deposit amount must be positive.'));

% Using the ThrowException action with the when function
import matlab.mock.actions.ThrowException
when(behavior.deposit(IsLessThan(0)),ThrowException( ...
    MException('Account:deposit:Negative', ...
    'Deposit amount must be positive.')))
```

However, there is more functionality when you use the `ThrowException` action. For instance, you can specify different subsequent behavior for the same mocked object interaction.

See Also

`matlab.mock.actions.ThrowException`

Introduced in R2017a

verifyAccessed

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Verify that a property was accessed

Syntax

```
verifyAccessed(testcase,behavior)
verifyAccessed(testcase,behavior,diagnostic)
```

Description

`verifyAccessed(testcase,behavior)` verifies that a property was accessed.

`verifyAccessed(testcase,behavior,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.PropertyBehavior` instance

Behavior of the mock, specified as a `matlab.mock.PropertyBehavior` instance. To create an instance of `matlab.mock.PropertyBehavior`, access a property of the behavior object.

Example: `behavior.PropertyFoo`

`diagnostic` — Diagnostic information to display
string array | character array | function handle | `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic information to display, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` object. Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Example: `"My diagnostic message."`

Example: `@() datetime('now')`

Examples

Verify Property Was Accessed

Create a mock with an implicit interface for interactive use. The interface includes `PropertyFoo` and `PropertyBar` properties. Access `PropertyFoo` by storing its value.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties', ...
    ["PropertyFoo","PropertyBar"]);
value = mock.PropertyFoo;
```

Verify that `PropertyFoo` was accessed.

```
testCase.verifyAccessed(behavior.PropertyFoo);
```

```
Verification passed.
```

Display Diagnostic When Verification Fails

Verify that `PropertyBar` was accessed. Display the diagnostic information upon failure.

```
testCase.verifyAccessed(behavior.PropertyBar, ...
    'PropertyBar should have been accessed.');
```

```
Verification failed.
    ----------------
```

```
Test Diagnostic:
----------------
PropertyBar should have been accessed.
----------------------
Framework Diagnostic:
----------------------
verifyAccessed failed.
--> Property 'PropertyBar' was never accessed.

Specified property access operation:
PropertyGetBehavior
    <Mock>.PropertyBar
```

Tips

Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point marks the current test method as failed and incomplete.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

Alternatives

Using the `verifyAccessed` method is functionally equivalent to using the `matlab.mock.constraints.WasAccessed` constraint with the `verifyThat` method of

the `Verifiable` class. For example, the following code blocks are functionally equivalent.

```
% Using the verifyAccessed method
testCase.verifyAccessed(behavior.PropertyFoo);

% Using the WasAccessed constraint with verifyThat method
import matlab.mock.constraints.WasAccessed;
testCase.verifyThat(behavior.PropertyFoo,WasAccessed);
```

However, there is more functionality when you use the `WasAccessed` constraint. For instance, you can specify a property was accessed a certain number of times.

See Also

`matlab.mock.constraints.WasAccessed`

Introduced in R2017a

verifyCalled

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Verify that a method was called with certain input values

Syntax

```
verifyCalled(testcase,behavior)
verifyCalled(testcase,behavior,diagnostic)
```

Description

`verifyCalled(testcase,behavior)` verifies that a method was called with certain input values.

`verifyCalled(testcase,behavior,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.MethodCallBehavior` instance

Behavior of the mock, specified as a `matlab.mock.MethodCallBehavior` instance. To create an instance of `matlab.mock.MethodCallBehavior`, call a method of the behavior object.

Example: `withExactInputs(myMockBehavior.myMockedMethod)`

`diagnostic` — Diagnostic information to display
string array | character array | function handle | `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic information to display, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` object. Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Example: `"My diagnostic message."`

Example: `@() datetime('now')`

Examples

Verify Method Was Called with Certain Input

Create a mock with an implicit interface for interactive use. The interface includes `foo` and `bar` methods. Call `foo` by specifying its input.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedMethods',["foo","bar"]);
mock.foo(123);
```

Verify that `foo` was called with input 123.

```
testCase.verifyCalled(behavior.foo(123));
```

```
Verification passed.
```

Display Diagnostic When Verification Fails

Verify that `foo` was called with input 456. Display the diagnostic information upon failure.

```
testCase.verifyCalled(behavior.foo(456), ...
    'Method foo should have been called with input 456.');
```

```
Verification failed.
    ----------------
```

```
    Test Diagnostic:
    ----------------
    Method foo should have been called with input 456.
    ----------------------
    Framework Diagnostic:
    ----------------------
    verifyCalled failed.
    --> Method 'foo' was not called with the specified signature.
    --> All observed method call(s) with any signature are:
            foo([1×1 matlab.mock.classes.Mock], 123)

    Specified method call:
    MethodCallBehavior
        [...] = foo(<Mock>, 456)
```

Verify that `bar` was called at least once with only the object as an input.

```
testCase.verifyCalled(withExactInputs(behavior.bar), ...
    'Method bar should have been called.');
```

```
Verification failed.
    ----------------
    Test Diagnostic:
    ----------------
    Method bar should have been called.
    ----------------------
    Framework Diagnostic:
    ----------------------
    verifyCalled failed.
    --> Method 'bar' was never called.

    Specified method call:
    MethodCallBehavior
        [...] = bar(<Mock>)
```

Tips

Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point marks the current test method as failed and incomplete.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

Alternatives

Using the `verifyCalled` method is functionally equivalent to using the `matlab.mock.constraints.WasCalled` constraint with the `verifyThat` method of the `Verifiable` class. For example, the following code blocks are functionally equivalent.

```
% Using the verifyCalled method
testCase.verifyCalled(behavior.foo(123), ...
    'Method foo should have been called with input 123.')

% Using the WasCalled constraint with verifyThat method
import matlab.mock.constraints.WasCalled;
testCase.verifyThat(behavior.foo(123),WasCalled, ...
    'Method foo should have been called with input 123.');
```

However, there is more functionality when you use the `WasCalled` constraint. For instance, you can specify a method was called a certain number of times.

See Also

`matlab.mock.constraints.WasCalled`

Introduced in R2017a

verifySet

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Verify that a property was set

Syntax

```
verifySet(testcase,behavior)
verifySet(testcase,behavior,diagnostic)
```

Description

`verifySet(testcase,behavior)` verifies that a property was set.

`verifySet(testcase,behavior,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.PropertyBehavior` instance

Behavior of the mock, specified as a `matlab.mock.PropertyBehavior` instance. To create an instance of `matlab.mock.PropertyBehavior`, access a property of the behavior object.

Example: `behavior.PropertyFoo`

`diagnostic` — Diagnostic information to display
string array | character array | function handle | `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic information to display, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` object. Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Example: `"My diagnostic message."`

Example: `@() datetime('now')`

Examples

Verify Property Was Set

Create a mock with an implicit interface for interactive use. The interface includes `PropertyFoo` and `PropertyBar` properties. Assign a value to `PropertyFoo`.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties', ...
    ["PropertyFoo","PropertyBar"]);
mock.PropertyFoo = 123;
```

Verify that `PropertyFoo` was set.

```
testCase.verifySet(behavior.PropertyFoo);
```

```
Verification passed.
```

Display Diagnostic When Verification Fails

Verify that `PropertyBar` was set. Display the diagnostic information upon failure.

```
testCase.verifySet(behavior.PropertyBar, ...
    'PropertyBar should have been set.');
```

```
Verification failed.
    ----------------
```

```
Test Diagnostic:
----------------
PropertyBar should have been set.
---------------------
Framework Diagnostic:
---------------------
verifySet failed.
--> Property 'PropertyBar' was never set.

Specified property set operation:
PropertySetBehavior
    <Mock>.PropertyBar = <IsAnything constraint>
```

Tips

Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point marks the current test method as failed and incomplete.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

Alternatives

Using the `verifySet` method is functionally equivalent to using the `matlab.mock.constraints.WasSet` constraint with the `verifyThat` method of the `Verifiable` class. For example, the following code blocks are functionally equivalent.

```
% Using the verifySet method
testCase.verifySet(behavior.PropertyFoo);

% Using the WasSet constraint with verifyThat method
import matlab.mock.constraints.WasSet;
testCase.verifyThat(behavior.PropertyFoo, WasSet);
```

However, there is more functionality when you use the `WasSet` constraint. For instance, you can specify a property was set a certain number of times.

See Also

`matlab.mock.constraints.WasSet`

Introduced in R2017a

verifyNotAccessed

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Verify that a property was not accessed

Syntax

```
verifyNotAccessed(testcase,behavior)
verifyNotAccessed(testcase,behavior,diagnostic)
```

Description

`verifyNotAccessed(testcase,behavior)` verifies that a property was not accessed.

`verifyNotAccessed(testcase,behavior,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.PropertyBehavior` instance

Behavior of the mock, specified as a `matlab.mock.PropertyBehavior` instance. To create an instance of `matlab.mock.PropertyBehavior`, access a property of the behavior object.

Example: `behavior.PropertyFoo`

`diagnostic` — Diagnostic information to display
string array | character array | function handle | `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic information to display, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` object. Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Example: `"My diagnostic message."`

Example: `@() datetime('now')`

Examples

Verify Property Was Not Accessed

Create a mock with an implicit interface for interactive use. The interface includes `PropertyFoo` and `PropertyBar` properties. Access `PropertyFoo` by storing its value.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties',...
    ["PropertyFoo","PropertyBar"]);
value = mock.PropertyFoo;
```

Verify that `PropertyBar` was not accessed.

```
testCase.verifyNotAccessed(behavior.PropertyBar);
```

```
Verification passed.
```

Display Diagnostic When Verification Fails

Verify that `PropertyFoo` was not accessed. Display the diagnostic information upon failure.

```
testCase.verifyNotAccessed(behavior.PropertyFoo, ...
    'PropertyFoo should not have been accessed.');
```

```
Verification failed.
    ----------------
```

```
Test Diagnostic:
----------------
PropertyFoo should not have been accessed.
----------------------
Framework Diagnostic:
----------------------
verifyNotAccessed failed.
--> Property 'PropertyFoo' was unexpectedly accessed 1 time(s).

Specified property access operation:
PropertyGetBehavior
    <Mock>.PropertyFoo
```

Tips

Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point marks the current test method as failed and incomplete.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

Alternatives

Using the `verifyNotAccessed` method is functionally equivalent to using the negated `matlab.mock.constraints.WasAccessed` constraint with the `verifyThat` method of

the `Verifiable` class. For example, the following code blocks are functionally equivalent.

```
% Using the verifyNotAccessed method
testCase.verifyNotAccessed(behavior.PropertyBar);

% Using the WasAccessed constraint with verifyThat method
import matlab.mock.constraints.WasAccessed;
testCase.verifyThat(behavior.PropertyBar,~WasAccessed);
```

However, there is more functionality when you use the `WasAccessed` constraint. For instance, you can specify a property was not accessed a certain number of times.

See Also

`matlab.mock.constraints.WasAccessed`

Introduced in R2017a

verifyNotCalled

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Verify that a method was not called with certain input values

Syntax

```
verifyNotCalled(testcase,behavior)
verifyNotCalled(testcase,behavior,diagnostic)
```

Description

`verifyNotCalled(testcase,behavior)` verifies that a method was not called with certain input values.

`verifyNotCalled(testcase,behavior,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.MethodCallBehavior` instance

Behavior of the mock, specified as a `matlab.mock.MethodCallBehavior` instance. To create an instance of `matlab.mock.MethodCallBehavior`, call a method of the behavior object.

Example: `withExactInputs(myMockBehavior.myMockedMethod)`

`diagnostic` — Diagnostic information to display
string array | character array | function handle | `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic information to display, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` object. Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Example: `"My diagnostic message."`

Example: `@() datetime('now')`

Examples

Verify Method Was Not Called with Certain Input

Create a mock with an implicit interface for interactive use. The interface includes a `foo` method. Call the method by specifying its input.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedMethods',"foo");
mock.foo(123);
```

Verify that `foo` was not called with input `456`.

```
testCase.verifyNotCalled(behavior.foo(456));
```

```
Verification passed.
```

Verify that `foo` was not called with only the object as an input.

```
testCase.verifyNotCalled(withExactInputs(behavior.foo));
```

```
Verification passed.
```

Display Diagnostic When Verification Fails

Verify that `foo` was not called with input `123`. Display the diagnostic information upon failure.

```
testCase.verifyNotCalled(behavior.foo(123), ...
    'Method foo should not have been called with input 123.');
```

```
Verification failed.
    ----------------
    Test Diagnostic:
    ----------------
    Method foo should not have been called with input 123.
    ----------------------
    Framework Diagnostic:
    ----------------------
    verifyNotCalled failed.
    --> Method 'foo' was unexpectedly called 1 time(s) with the specified signature.
    --> All observed method call(s) with any signature are:
            foo([1×1 matlab.mock.classes.Mock], 123)

    Specified method call:
    MethodCallBehavior
        [...] = foo(<Mock>, 123)
```

Tips

Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point marks the current test method as failed and incomplete.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

Alternatives

Using the `verifyNotCalled` method is functionally equivalent to using the negated `matlab.mock.constraints.WasCalled` constraint with the `verifyThat` method of the `Verifiable` class. For example, the following code blocks are functionally equivalent.

```
% Using the verifyNotCalled method
testCase.verifyNotCalled(behavior.foo(456), ...
    'Method foo should not have been called with input 456.')

% Using the WasCalled constraint with verifyThat method
import matlab.mock.constraints.WasCalled;
testCase.verifyThat(behavior.foo(456),~WasCalled, ...
    'Method foo should not have been called with input 456.')
```

However, there is more functionality when you use the `WasCalled` constraint. For instance, you can specify a method was not called a certain number of times.

See Also

`matlab.mock.constraints.WasCalled`

Introduced in R2017a

verifyNotSet

Class: `matlab.mock.TestCase`
Package: `matlab.mock`

Verify that a property was not set

Syntax

```
verifyNotSet(testcase,behavior)
verifyNotSet(testcase,behavior,diagnostic)
```

Description

`verifyNotSet(testcase,behavior)` verifies that a property was not set.

`verifyNotSet(testcase,behavior,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`testcase` — Instance of test case
`matlab.mock.TestCase` object

Instance of the test case, specified as a `matlab.mock.TestCase` object.

`behavior` — Behavior of mock
`matlab.mock.PropertyBehavior` instance

Behavior of the mock, specified as a `matlab.mock.PropertyBehavior` instance. To create an instance of `matlab.mock.PropertyBehavior`, access a property of the behavior object.

Example: `behavior.PropertyFoo`

`diagnostic` — Diagnostic information to display
string array | character array | function handle | `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic information to display, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` object. Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Example: `"My diagnostic message."`

Example: `@() datetime('now')`

Examples

Verify Property Was Not Set

Create a mock with an implicit interface for interactive use. The interface includes `PropertyFoo` and `PropertyBar` properties. Assign a value to `PropertyFoo`.

```
testCase = matlab.mock.TestCase.forInteractiveUse;
[mock,behavior] = testCase.createMock('AddedProperties', ...
    ["PropertyFoo","PropertyBar"]);
mock.PropertyFoo = 123;
```

Verify that `PropertyBar` was not set.

```
testCase.verifyNotSet(behavior.PropertyBar);
```

```
Verification passed.
```

Display Diagnostic When Verification Fails

Verify that `PropertyFoo` was not set. Display the diagnostic information upon failure.

```
testCase.verifyNotSet(behavior.PropertyFoo, ...
    'PropertyFoo should not have been set.');
```

```
Verification failed.
    ----------------
```

```
Test Diagnostic:
----------------
PropertyFoo should not have been set.
---------------------
Framework Diagnostic:
---------------------
verifyNotSet failed.
--> Property 'PropertyFoo' was unexpectedly set to the specified value 1 time(s).
--> All observed property set operation(s) where the property was set to any value
        <Mock>.PropertyFoo = 123

Specified property set operation:
PropertySetBehavior
    <Mock>.PropertyFoo = <IsAnything constraint>
```

Tips

Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualifications for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point marks the current test method as failed and incomplete.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

Alternatives

Using the `verifyNotSet` method is functionally equivalent to using the negated `matlab.mock.constraints.WasSet` constraint with the `verifyThat` method of the `Verifiable` class. For example, the following code blocks are functionally equivalent.

```
% Using the verifyNotSet method
testCase.verifyNotSet(behavior.PropertyBar);

% Using the WasSet constraint with verifyThat method
import matlab.mock.constraints.WasSet;
testCase.verifyThat(behavior.PropertyBar,~WasSet);
```

However, there is more functionality when you use the `WasSet` constraint. For instance, you can specify a property was not set a certain number of times.

See Also

`matlab.mock.constraints.WasSet`

Introduced in R2017a

matlabrc

System administrator-defined start up script for MATLAB

Syntax

```
matlabrc
```

Description

`matlabrc` initializes the MATLAB program. At startup, MATLAB automatically executes `matlabrc` and then the `startup` script, if it exists.

For system administrators managing multi-user or networked systems, modify `matlabrc` to add messages, definitions, or other code that applies to all users. The `matlabrc` script is in the *`matlabroot`*`/toolbox/local` folder.

For individual systems, use the `startup` script to customize MATLAB startup. Put this script in the `userpath` folder.

Tips

You also can start MATLAB using options that you define at the command prompt. For more information, see “Specify Startup Options”.

See Also

`matlabroot` | `quit` | `restoredefaultpath` | `startup` | `userpath`

Topics

“Specify Startup Options”
“MATLAB Startup Folder”

Introduced before R2006a

matlabroot

MATLAB root folder

Syntax

```
mr = matlabroot
```

Description

`mr = matlabroot` returns a character vector giving the full path to the folder where MATLAB is installed. Use `matlabroot` to create a path to MATLAB and toolbox folders that is independent of platform, MATLAB version, or installation location.

Examples

Get MATLAB Install Location

Get the location where MATLAB is installed.

```
matlabroot
```

```
ans =

    'C:\Program Files\MATLAB\R2017b'
```

Get Full Path to Folder

Get the full path to the `toolbox/matlab/general` folder for the current system.

```
fullfile(matlabroot,'toolbox','matlab','general')
```

```
ans =

    'C:\Program Files\MATLAB\R2017b\toolbox\matlab\general'
```

Set Current Folder to MATLAB Root

```
cd(matlabroot)
```

Add Folder to Path

Add the folder `myfiles` to the MATLAB search path.

```
addpath([matlabroot '/toolbox/local/myfiles'])
```

More About

matlabroot

The term *`matlabroot`* can also refer to the folder where MATLAB files are installed.

For example, in the documentation, the phrase "save to *`matlabroot`*`/toolbox/local`" means save to the `toolbox/local` folder in the MATLAB root folder. If your MATLAB root folder is `C:\Program Files\MATLAB\R2017b`, then you would save to the folder `C:\Program Files\MATLAB\R2017b\toolbox\local`.

$matlabroot

The macro `$matlabroot` can be used literally in some types of files to represent the full path to the MATLAB root folder.

For example, including the line `$matlabroot/toolbox/local/myfile.jar` in `javaclasspath.txt`, adds the full path of `myfile.jar` to the static Java class path. For more information, see “Static Path”.

Sometimes, particularly in older code examples, the term `$matlabroot` or `$MATLABROOT` is also used to represent the value returned by the `matlabroot` function.

Algorithms

matlabroot on Macintosh

On Macintosh systems running R2008b (V7.7) or newer versions, running `matlabroot` returns, for example

```
/Applications/MATLAB_R2008b.app
```

On Mac systems running R2008a (V7.6) or earlier versions, `matlabroot` returns, for example

```
/Applications/MATLAB_R2008a
```

On Mac systems, you cannot directly view the contents of the MATLAB root folder. For more information, including how to view the contents, see “Navigating Within the MATLAB Root Folder on macOS Platforms”.

See Also

`fullfile` | `path` | `toolboxdir`

Introduced before R2006a

matlabshared.supportpkg.checkForUpdate

List of support packages that can be updated (Not recommended)

Note `checkForUpdate` will be removed in a future release. To check for updates to support packages, on the MATLAB® **Home** tab, in the **Environment** section, click **Help** > **Check for Updates**.

Syntax

```
matlabshared.supportpkg.checkForUpdate
info = matlabshared.supportpkg.checkForUpdate
```

Description

`matlabshared.supportpkg.checkForUpdate` displays information about support package updates in the MATLAB Command Window. If an update is available, use `supportPackageInstaller` to install the updates.

`info = matlabshared.supportpkg.checkForUpdate` returns a structured array of information about installed support packages.

Examples

Check for support package updates

```
matlabshared.supportpkg.checkForUpdate
```

```
No support packages need updates.
```

If one or more updates are available, the command line displays that information in the response.

Get a structured array of support package updates

```
info = matlabshared.supportpkg.checkForUpdate
```

```
info = 

                Name: 'Arduino'
    InstalledVersion: '3.0'
         BaseProduct: 'Simulink'
```

Output Arguments

`info` — Return argument from function
structure created using `matlabshared.supportpkg.checkForUpdate`

Information about support package updates, returned as a structured array.

See Also

`matlabshared.supportpkg.getInstalled` | `supportPackageInstaller`

Introduced in R2014b

matlabshared.supportpkg.getInstalled

List of installed support packages

Syntax

```
matlabshared.supportpkg.getInstalled
info = matlabshared.supportpkg.getInstalled
```

Description

`matlabshared.supportpkg.getInstalled` displays information about installed support packages in the MATLAB Command Window.

`info = matlabshared.supportpkg.getInstalled` returns a structured array of information about installed support packages.

Examples

Get a list of installed support packages

```
matlabshared.supportpkg.getInstalled
```

```
Name   Version Base Product
------- ------- ------------
Arduino 3.0     Simulink
```

Get a structured array of installed support packages

```
info = matlabshared.supportpkg.getInstalled
```

```
info =

                 Name: 'Arduino'
```

```
    InstalledVersion: '3.0'
         BaseProduct: 'Simulink'
```

Output Arguments

`info` — Return argument from function
structure created using `matlabshared.supportpkg.getInstalled`

Information about installed support packages, returned as a structured array.

See Also

`matlabshared.supportpkg.checkForUpdate` | `supportPackageInstaller`

Introduced in R2014b

matlabshared.supportpkg.getSupportPackageRoot

Get root folder of support packages

Syntax

```
installDir = matlabshared.supportpkg.getSupportPackageRoot
```

Description

`installDir = matlabshared.supportpkg.getSupportPackageRoot` gets root folder of the current support packages. Support Package Installer installs all support packages to this location.

Examples

Set and Get Custom Support Package Root

Use the `setSupportPackageRoot` function to change the installation folder for support packages. You need administrative privileges to use this function.

```
matlabshared.supportpkg.setSupportPackageRoot('C:\MATLAB\CustomSupportPackageRoot')
```

Use `getSupportPackageRoot` to check that the change was successful.

```
matlabshared.supportpkg.getSupportPackageRoot
```

```
ans =

C:\MATLAB\CustomSupportPackageRoot
```

Output Arguments

`installDir` — Root folder of support packages
string

Root folder of support packages, returned as a string. By default, the directory is set to:

- Windows – `'C:\MATLAB\SupportPackage\release'`
- For Unix – `'/home/user/Documents/MATLAB/SupportPackages/release'`

Here, *`release`* is your specific MATLAB release and *`user`* is your Unix user name. If the default location exists and is not empty, the location becomes *`release_num`*.

Example: `'C:\MATLAB\SupportPackage\R2016a'`

Data Types: `char`

See Also

`matlabshared.supportpkg.setSupportPackageRoot`

Topics

"Support Package Installation"

Introduced in R2016a

matlabshared.supportpkg.setSupportPackageRoot

Set root folder of support packages

Syntax

```
matlabshared.supportpkg.setSupportPackageRoot(installDir)
```

Description

`matlabshared.supportpkg.setSupportPackageRoot(installDir)` sets the root folder of the current support packages. Support Package Installer installs all support packages to this location.

You must have administrative privileges on the folder that you set the root to. You must also have administrative privileges to modify `matlabroot`, because the root folder of the support package is recorded under `matlabroot`.

When you change the root folder, support packages installed in the new location become available on the MATLAB path. Any previously installed support packages are removed from the MATLAB path, but the files are still available.

Examples

Set and Get Custom Support Package Root

Use the `setSupportPackageRoot` function to change the installation folder for support packages. You need administrative privileges to use this function.

```
matlabshared.supportpkg.setSupportPackageRoot('C:\MATLAB\CustomSupportPackageRoot')
```

Use `getSupportPackageRoot` to check that the change was successful.

```
matlabshared.supportpkg.getSupportPackageRoot
```

```
ans =

C:\MATLAB\CustomSupportPackageRoot
```

Input Arguments

`installDir` — Root folder of support packages
string

Root folder of support packages, specified as a string. By default, the directory is set to:

- Windows – `'C:\MATLAB\SupportPackage\release'`
- For Unix – `'/home/user/Documents/MATLAB/SupportPackages/release'`

Here, *`release`* is your specific MATLAB release and *`user`* is your Unix user name. If the default location exists and is not empty, the location becomes *`release_num`*.

Note The folder path cannot contain any space characters.

Example: `'C:\MATLAB\SupportPackage\R2016a'`

Data Types: `char`

See Also

`matlabshared.supportpkg.getSupportPackageRoot`

Topics

"Support Package Installation"

Introduced in R2016a

matlab (Linux)

Start MATLAB program from Linux system prompt

Syntax

```
matlab
matlab option1 ... optionN
```

Description

`matlab` is a Bourne shell script that launches the MATLAB program from a Linux system prompt. Here the term `matlab` refers to this script and MATLAB refers to the program.

The `matlab` script:

- Determines the MATLAB root folder, the value returned by the `matlabroot` function
- Processes command-line options, if any
- Reads the MATLAB start-up file, `.matlab7rc.sh`
- Sets MATLAB environment variables

`matlab option1 ... optionN` launches MATLAB with the specified start-up options.

Alternatively, assign start-up options in the MATLAB “.matlab7rc.sh Start-up File” on page 1-9503. Modifying the `.matlab7rc.sh` file defines start-up options every time you start MATLAB.

MATLAB uses the Java Virtual Machine (JVM) software to run the desktop and to display graphics. The `-nojvm` option enables you to start MATLAB without the JVM. Using this option minimizes memory usage and improves initial start-up speed, but restricts functionality.

Input Arguments

`option1 ... optionN` — One or more start-up options
strings

One or more start-up options, specified as strings corresponding to valid start-up options from the following tables.

Mode Options

Option	Result
`-desktop`	Start MATLAB without a controlling terminal. Use this option when you start MATLAB from a window manager menu or desktop icon.
`-nodesktop`	Run the JVM software without opening the MATLAB desktop. You can use development environment tools by calling them as functions. To run in batch processing mode, use the `-batch` option. If you use the > constructor to pipe to MATLAB, then the `nodesktop` option is used automatically. MATLAB provides a command window-only interface in the desktop environment. On the **Home** tab, click **Layout**. Then, under **Select Layout**, select **Command Window Only**.
`-nojvm`	Start MATLAB without the JVM software. Features that require Java software (such as the desktop tools and graphics) are not supported.

Display Options

Option	Result
`-noFigureWindows`	Disable the display of figure windows in MATLAB.
`-nosplash`	Do not display the splash screen during startup.
`-nodisplay`	Start the JVM software without starting the MATLAB desktop. This option does not display X commands. It overrides the `DISPLAY` environment variable.

Option	Result
`-display xDisp`	Send X commands to X Window Server display `xDisp`. This option overrides the `DISPLAY` environment variable.

Set Initial Working Folder

The initial working folder is the current folder when MATLAB starts. For more information, see “MATLAB Startup Folder”.

Option	Result
`-sd` *`folder`*	Set the MATLAB folder to *`folder`*, specified as a string. Example: `matlab -sd "C:\work"`
`-useStartupFolderPref`	Set the MATLAB folder to the value specified by the **Initial working folder** option in the General Preferences panel.

Specify MATLAB Version

Option	Result
`v=`*`variant`*	Start the version of MATLAB in the `bin/`*`arch`*`/`*`variant`* folder instead of the `bin/`*`arch`* folder, where: • *`arch`* is the system architecture, the value returned by the `computer('arch')` function • *`variant`* is a string representing a MATLAB version

Debugging Options

Option	Result
`-logfile` *`filename`*	Copy Command Window output, including error reports, into *`filename`*, specified as a string. Example: `-logfile output.log`

Option	Result
`-n`	Display, without starting MATLAB, the final values of the environment variables and arguments passed to the MATLAB executable. This option also displays other diagnostic information for use when working with a Technical Support Representative.
`-e`	Display, without starting MATLAB, all environment variables and their values to standard output. If the exit status is not `0` on return, then the variables and values might not be correct.
`-D`*`debugger debugopts`	Start MATLAB in debug mode. This option uses the debugger program name, *`debugger`*, specified as a string, for example, `gdb`, `lldb`, or `dbx`. You can specify the full path to the debugger. This option must be the first option in the `matlab` script. Debugger program command-line options, *`debugopts`*, specified as a string of valid options for *`debugger`*. See your debugger documentation for details. Do not use any other `matlab` script options when using *`debugopts`*. Do not add a space between **D** and *`debugger`*. Example: `-Dgdb`
`-jdb` *`portnumber`*	Enable use of the Java debugger. The Java debugger uses the default *`portnumber`* value 4444 to communicate with MATLAB. The port number is optional. However, to use the Java debugger while running multiple MATLAB sessions, you must specify a port number. The *`portnumber`* value must be an integer in the range 0–65535. The integer cannot be reserved or currently in use by another application on your system.
`-debug`	Display information for debugging X-based problems. Use this option only when working with a Technical Support Representative from MathWorks, Inc.

Execute MATLAB Script or Function

Option	Result
`-batch` *`statement`*	Execute MATLAB script, statement, or function non-interactively. MATLAB: • Starts without the desktop • Does not display the splash screen • Executes *`statement`* • Disables changes to preferences • Disables toolbox caching • Logs text to `stdout` and `stderr` • Does not display dialog boxes • Exits automatically with exit code 0 if *`script`* executes successfully. Otherwise, MATLAB terminates with a non-zero exit code. *`statement`* is MATLAB code enclosed in double quotation marks. If *`statement`* is the name of a MATLAB function or script, do not specify the file extension. Any required file must be on the MATLAB search path or in the startup folder. Use the `-batch` option in non-interactive scripting or command line work flows. Do not use this option with the `-r` option. To test if a session of MATLAB is running in batch mode, call the `batchStartupOptionUsed` function. Example: `-batch "myscript"`

Option	Result
`-r statement`	Execute the MATLAB *`statement`*. Use this option for interactive work flows. Do not use this option with the `-batch` option. **Note** To set the initial working folder, use the `-sd` option. For example: `-sd folder` Example: `-r "disp(['Current folder: ' pwd])"` Example: `-r "myscript"`

Use Single Computational Thread

By default, MATLAB uses the multithreading capabilities of the computer on which it is running.

Option	Result
`-singleCompThread`	Limit MATLAB to a single computational thread.

Disable Searching Custom Java Class Path

Option	Result
`-nouserjavapath`	Disable use of `javaclasspath.txt` and `javalibrarypath.txt` files. For more information, see “Specifying Java Startup Options”.

OpenGL Library Options

These options control the use of software OpenGL libraries when MATLAB detects a graphics driver with known issues. For more information, see “Graphics Features That Have Specific Requirements”.

Option	Result
`-softwareopengl`	Force MATLAB to start with software OpenGL libraries.

Option	Result
`-nosoftwareopengl`	Disable auto-selection of OpenGL software.

Specify License File

Option	Result
`-c` *`license`*	Use the specified license file, *`license`*, specified as a string, a colon-separated list of license file names, or a `port@host` entry. For more information, see "Understanding Network License Files" (Installation, Licensing, and Activation).

Help Options

Option	Result
`-h`	Display startup options without starting MATLAB.
`-help`	Same as `-h` option.

Examples

Start MATLAB Without Desktop

```
matlab -nojvm -nodisplay -nosplash
```

Display Current Folder at Start-up

```
matlab -r "disp(['Current folder: ' pwd])"
```

More About

.matlab7rc.sh Start-up File

The `.matlab7rc.sh` shell script contains variable definitions used by the `matlab` script.

Use the `.matlab7rc.sh` file to redefine variables defined in the `matlab` script. `matlab` looks in these folders for the first occurrence of `.matlab7rc.sh` in the following order:

1. Current folder
2. Home folder (`$HOME`)
3. *`matlabroot`*`/bin` folder

To edit the `.matlab7rc.sh` file, use the template located in the *`matlabroot`*`/bin` folder.

This table lists the variables. For more information, see the comments in the `.matlab7rc.sh` file.

Variable	Definition and Standard Assignment Behavior
`ARCH`	Machine architecture MATLAB checks these values in this order: 1. `ARCH`, passed with the `-arch` or `-arch/ext` argument to the script 2. Value of the environment variable `MATLAB_ARCH`
`DISPLAY`	Host name of the X Window display MATLAB uses for output The value of `Xdisplay` passed with the `-display` argument to the script is used; otherwise, the value in the environment is used. MATLAB ignores `DISPLAY` if the `-nodisplay` argument is passed.
`LD_LIBRARY_PATH`	Final Load library path The final value is normally a colon-separated list of four sublists, each of which could be empty. The sublists are: • Defined in `.matlab7rc.sh` as `LDPATH_PREFIX` • Computed in the script and includes folders inside the MATLAB root folder and relevant Java folders • Contains any nonempty value of `LD_LIBRARY_PATH` from the environment possibly augmented in `.matlab7rc.sh` • Defined in `.matlab7rc.sh` as `LDPATH_SUFFIX`

Variable	Definition and Standard Assignment Behavior
MATLAB	MATLAB root folder MATLAB uses the default computed by the script unless MATLABdefault is reset in .matlab7rc.sh. Currently MATLABdefault is not reset in the shipping .matlab7rc.sh.
MATLABPATH	MATLAB search path The final value is a colon-separated list with the MATLABPATH from the environment prepended to a list of computed defaults. At start-up, you can add subfolders of userpath to the MATLAB search path. See userpath for details.
SHELL	Shell to use with MATLAB "!" or unix commands This value is taken from the environment, unless SHELL is reset in .matlab7rc.sh. The default .matlab7rc.sh file does not reset SHELL.
MATLAB_SHELL	Shell to use instead of SHELL MATLAB checks for MATLAB_SHELL first. If it is empty or not defined, then checks SHELL. If SHELL is also empty or not defined, then MATLAB uses the Bourne shell, /bin/sh. Use an absolute path for the value of MATLAB_SHELL, that is, /bin/sh, not simply sh. The default .matlab7rc.sh file does not reference or set MATLAB_SHELL.
TOOLBOX	Path of the toolbox folder **1** Value in the environment, if not empty **2** Value of TOOLBOX in .matlab7rc.sh, if reset **3** Value of *matlabroot*/toolbox computed by the script The MATLAB version of .matlab7rc.sh does not reset TOOLBOX.

The `matlab` script determines the path of the MATLAB root folder by looking up the folder tree from the *`matlabroot`*`/bin` folder (where the `matlab` script is located). MATLAB use the `MATLAB` variable to locate all files within the MATLAB folder tree.

You can change the definition of `MATLAB`. For example, you might change the definition:

- To run a different version of MATLAB
- When your system uses certain types of automounting schemes and the path determined by the `matlab` script is not correct

Compatibility Considerations

`-r` option for non-interactive use is not recommended

Not recommended starting in R2019a

To start MATLAB non-interactively, use the `-batch` option. To start MATLAB interactively, use the `-r` option.

See Also

`batchStartupOptionUsed` | `computer` | `matlab (Windows)` | `matlab (macOS)` | `quit` | `unix`

Topics

"Start MATLAB on Linux Platforms"
"Specify Startup Options"
General Preferences

matlab (macOS)

Start MATLAB program from macOS Terminal

Syntax

```
matlab
matlab option1 ... optionN
```

Description

`matlab` is a Bourne shell script that starts the MATLAB program from a macOS system prompt. Here the term `matlab` refers to this script and MATLAB refers to the program.

The `matlab` script is located in the MATLAB application package, `/Applications/`*`matlabroot`*`/MATLAB_`*`release`*`.app/bin`, where *`matlabroot`* is the name of the folder in which you installed MATLAB and *`release`* is the MATLAB release number.

The `matlab` script:

- Determines the MATLAB root folder, the value returned by the `matlabroot` function
- Processes command-line options, if any
- Reads the MATLAB startup file, `.matlab7rc.sh`
- Sets MATLAB environment variables

`matlab option1 ... optionN` starts MATLAB with the specified startup options.

Alternatively, assign startup options in the MATLAB “.matlab7rc.sh Startup File” on page 1-9514. Modifying the `.matlab7rc.sh` file defines startup options every time you start MATLAB. On macOS platforms, the `.matlab7rc.sh` file applies only when starting MATLAB from Terminal. The file does not apply when starting MATLAB by double-clicking the MATLAB icon in the Applications folder.

MATLAB uses the Java Virtual Machine (JVM) software to run the desktop and to display graphics. The `-nojvm` option enables you to start MATLAB without the JVM. Using this option minimizes memory usage and improves initial startup speed, but restricts functionality.

Input Arguments

`option1 ... optionN` — One or more startup options
strings

One or more startup options, specified as strings corresponding to valid startup options from the following tables.

Mode Options

Option	Result
`-desktop`	Start MATLAB without a controlling terminal. Use this option when you start MATLAB from a window manager menu or desktop icon.
`-nodesktop`	Run the JVM software without opening the MATLAB desktop. You can use development environment tools by calling them as functions. Use this option to run in batch processing mode. If you use the > constructor to pipe to MATLAB, then the `nodesktop` option is used automatically. MATLAB provides a command window-only interface in the desktop environment. On the **Home** tab, click **Layout**. Then, under **Select Layout**, select **Command Window Only**.
`-nojvm`	Start MATLAB without the JVM software. Features that require Java software (such as the desktop tools and graphics) are not supported.

Display Options

Option	Result
`-noFigureWindows`	Disable the display of figure windows in MATLAB.
`-nosplash`	Do not display the splash screen during startup.
`-nodisplay`	Start the JVM software without starting the MATLAB desktop.

Set Initial Working Folder

The initial working folder is the current folder when MATLAB starts. For more information, see “MATLAB Startup Folder”.

Option	Result
`-sd` *`folder`*	Set the MATLAB folder to *`folder`*, specified as a string. Example: `matlab -sd "C:\work"`
`-useStartupFolderPref`	Set the MATLAB folder to the value specified by the **Initial working folder** option in the General Preferences panel.

Specify MATLAB Version

Option	Result
`v=`*`variant`*	Start the version of MATLAB in the `bin/`*`arch`*`/`*`variant`* folder instead of the `bin/`*`arch`* folder, where: • *`arch`* is the system architecture, the value returned by the `computer('arch')` function • *`variant`* is a string representing a MATLAB version

Debugging Options

Option	Result
`-logfile` *`filename`*	Copy Command Window output, including error reports, into *`filename`*, specified as a string. Example: `-logfile output.log`
`-n`	Display, without starting MATLAB, the final values of the environment variables and arguments passed to the MATLAB executable. This option also displays other diagnostic information for use when working with a Technical Support Representative.
`-e`	Display, without starting MATLAB, all environment variables and their values to standard output. If the exit status is not `0` on return, then the variables and values might not be correct.

Option	Result
`-D`*`debugger debugopts*	Start MATLAB in debug mode. This option uses the debugger program name, *`debugger`*, specified as a string, for example, `gdb`, `lldb`, or `dbx`. You can specify the full path to the debugger. This option must be the first option in the `matlab` script. Debugger program command-line options, *`debugopts`*, specified as a string of valid options for *`debugger`*. See your debugger documentation for details. Do not use any other `matlab` script options when using *`debugopts`*. Do not add a space between **D** and *`debugger`*. Example: `-Dlldb`
`-jdb` *`portnumber`*	Enable use of the Java debugger. The Java debugger uses the default *`portnumber`* value 4444 to communicate with MATLAB. The port number is optional. However, to use the Java debugger while running multiple MATLAB sessions, you must specify a port number. The *`portnumber`* value must be an integer in the range 0–65535. The integer cannot be reserved or currently in use by another application on your system.

Execute MATLAB Script or Function

Option	Result
`-batch` *`statement`*	Execute MATLAB script, statement, or function non-interactively. MATLAB: • Starts without the desktop • Does not display the splash screen • Executes *`statement`* • Disables changes to preferences • Disables toolbox caching • Logs text to `stdout` and `stderr` • Does not display dialog boxes • Exits automatically with exit code 0 if *`script`* executes successfully. Otherwise, MATLAB terminates with a non-zero exit code. *`statement`* is MATLAB code enclosed in double quotation marks. If *`statement`* is the name of a MATLAB function or script, do not specify the file extension. Any required file must be on the MATLAB search path or in the startup folder. Use the `-batch` option in non-interactive scripting or command line work flows. Do not use this option with the `-r` option. To test if a session of MATLAB is running in batch mode, call the `batchStartupOptionUsed` function. Example: `-batch "myscript"`

Option	Result
`-r statement`	Execute the MATLAB *`statement`*. Use this option for interactive work flows. Do not use this option with the `-batch` option. **Note** To set the initial working folder, use the `-sd` option. For example: `-sd folder` Example: `-r "disp(['Current folder: ' pwd])"` Example: `-r "myscript"`

Use Single Computational Thread

By default, MATLAB uses the multithreading capabilities of the computer on which it is running.

Option	Result
`-singleCompThread`	Limit MATLAB to a single computational thread.

Disable Searching Custom Java Class Path

Option	Result
`-nouserjavapath`	Disable use of `javaclasspath.txt` and `javalibrarypath.txt` files. For more information, see "Specifying Java Startup Options".

Specify License File

Option	Result
`-c license`	Use the specified license file, *`license`*, specified as a string, a colon-separated list of license file names, or a `port@host` entry. For more information, see "Understanding Network License Files" (Installation, Licensing, and Activation).

Help Options

Option	Result
`-h`	Display startup options without starting MATLAB.
`-help`	Same as `-h` option.

Examples

Start MATLAB R2015a from Applications Folder

Move to the `bin` folder within the application package for MATLAB installed in a folder named `MyMATLAB`.

```
cd /Applications/MyMATLAB/MATLAB_R2015a.app/bin
```

Preface the `matlab` script with `./` characters.

```
./matlab -nosplash
```

Start MATLAB from Any Folder

Start MATLAB R2015a from any Terminal folder by specifying the full path name. Assume MATLAB is installed in a folder named `MyMATLAB`.

```
/Applications/MyMATLAB/MATLAB_R2015a.app/bin/matlab
```

Start MATLAB Without Desktop

```
matlab -nojvm -nodisplay -nosplash
```

Display Current Folder at Startup

```
matlab -r "disp(['Current folder: ' pwd])"
```

More About

.matlab7rc.sh Startup File

The `.matlab7rc.sh` shell script contains variable definitions used by the `matlab` script. To redefine variables defined in the `matlab` script, modify the `.matlab7rc.sh` file. The `.matlab7rc.sh` file applies only when starting MATLAB from Terminal; it does not apply when starting MATLAB by double-clicking the MATLAB icon in the Applications folder.

`matlab` looks in these folders for the first occurrence of `.matlab7rc.sh` in the following order:

1. Current folder
2. Home folder (`$HOME`)
3. *`matlabroot`*`/bin` folder

To edit the `.matlab7rc.sh` file, use the template located in the *`matlabroot`*`/bin` folder.

The following table lists the variables. For more information, see the comments in the `.matlab7rc.sh` file.

`.matlab7rc.sh` Variable	Definition and Standard Assignment Behavior
`ARCH`	Machine architecture MATLAB checks these values in this order: • The value `ARCH` passed with the `-arch` or `-arch/ext` argument to the script. • The value of the environment variable `MATLAB_ARCH`.

`.matlab7rc.sh` Variable	Definition and Standard Assignment Behavior
`DYLD_LIBRARY_PATH`	Final Load library path The final value is normally a colon-separated list of four sublists, each of which could be empty. The sublists are: • Defined in `.matlab7rc.sh` as `LDPATH_PREFIX` • Computed in the script and includes folders inside the MATLAB root folder and relevant Java folders • Contains any nonempty value of `DYLD_LIBRARY_PATH` from the environment possibly augmented in `.matlab7rc.sh` • Defined in `.matlab7rc.sh` as `LDPATH_SUFFIX`
`MATLAB`	MATLAB root folder MATLAB uses the default computed by the script unless `MATLABdefault` is reset in `.matlab7rc.sh`. Currently `MATLABdefault` is not reset in the shipping `.matlab7rc.sh`.
`MATLABPATH`	MATLAB search path The final value is a colon-separated list with the `MATLABPATH` from the environment prepended to a list of computed defaults. At startup, you can add subfolders of `userpath` to the MATLAB search path. See `userpath` for details.
`SHELL`	Shell to use with MATLAB “`!`” or `unix` commands This value is taken from the environment, unless `SHELL` is reset in `.matlab7rc.sh`. The default `.matlab7rc.sh` file does not reset `SHELL`.

`.matlab7rc.sh` Variable	Definition and Standard Assignment Behavior
`MATLAB_SHELL`	Shell to use instead of `SHELL` MATLAB checks for `MATLAB_SHELL` first and, if empty or not defined, checks `SHELL`. If `SHELL` is also empty or not defined, MATLAB uses the Bourne shell, `/bin/sh`. Use an absolute path for the value of `MATLAB_SHELL`, that is, `/bin/sh`, not simply `sh`. The default `.matlab7rc.sh` file does not reference or set `MATLAB_SHELL`.
`TOOLBOX`	Path of the toolbox folder **1** Value in the environment, if not empty **2** Value of `TOOLBOX` in `.matlab7rc.sh`, if reset **3** Value of *`matlabroot`*`/toolbox` computed by the script The MATLAB version of `.matlab7rc.sh` does not reset `TOOLBOX`.

The `matlab` script determines the path of the MATLAB root folder by looking up the folder tree from the *`matlabroot`*`/bin` folder (where the `matlab` script is located). MATLAB use the `MATLAB` variable to locate all files within the MATLAB folder tree.

You can change the definition of `MATLAB`. For example, you might change the definition:

- To run a different version of MATLAB
- When your system uses certain types of automounting schemes and the path determined by the `matlab` script is not correct

Compatibility Considerations

`-r` option for non-interactive use is not recommended

Not recommended starting in R2019a

To start MATLAB non-interactively, use the `-batch` option. To start MATLAB interactively, use the `-r` option.

See Also

`batchStartupOptionUsed` | `matlab (Linux)` | `matlab (Windows)` | `matlabroot` | `quit` | `unix`

Topics

"Start MATLAB on macOS Platforms"
"Specify Startup Options"
General Preferences

matlab (Windows)

Start MATLAB program from Windows system prompt

Syntax

```
matlab
matlab option1 ... optionN
```

Description

`matlab` starts the MATLAB program from the Microsoft Windows system prompt. In this topic the term `matlab` refers to the command you type, and MATLAB refers to the program.

The `matlab` command:

- Determines the MATLAB root folder, the value returned by the `matlabroot` function.
- Processes command-line options and passes other options to MATLAB.

To capture the exit code, start MATLAB with the `-wait` option.

`matlab option1 ... optionN` starts MATLAB with the specified startup options.

Input Arguments

`option1 ... optionN` — One or more startup options
strings

One or more startup options, specified as strings corresponding to valid startup options from the following tables.

Display Options

Option	Result
`-noFigureWindows`	Disable the display of figure windows in MATLAB.
`-nosplash`	Do not display the splash screen during startup.

Set Initial Working Folder

The initial working folder is the current folder when MATLAB starts. For more information, see “MATLAB Startup Folder”.

Option	Result
`-sd` *`folder`*	Set the MATLAB folder to *`folder`*, specified as a string. Example: `matlab -sd "C:\work"`
`-useStartupFolderPref`	Set the MATLAB folder to the value specified by the **Initial working folder** option in the General Preferences panel.

Debugging Options

Option	Result
`-logfile` *`filename`*	Copy Command Window output, including error log reports, in to *`filename`*, specified as a string. Example: `-logfile output.log`
`-jdb` *`portnumber`*	Enable use of the Java debugger. The Java debugger uses the default *`portnumber`* value 4444 to communicate with MATLAB. The port number is optional. However, to use the Java debugger while running multiple MATLAB sessions, you must provide a port number. The *`portnumber`* value must be an integer in the range 0–65535. The integer cannot be reserved or currently in use by another application on your system.

Execute MATLAB Script or Function

Option	Result
`-batch` *`statement`*	Execute MATLAB script, statement, or function non-interactively. MATLAB: • Starts without the desktop • Does not display the splash screen • Executes *`statement`* • Disables changes to preferences • Disables toolbox caching • Logs text to `stdout` and `stderr` • Does not display dialog boxes • Exits automatically with exit code 0 if *`script`* executes successfully. Otherwise, MATLAB terminates with a non-zero exit code. *`statement`* is MATLAB code enclosed in double quotation marks. If *`statement`* is the name of a MATLAB function or script, do not specify the file extension. Any required file must be on the MATLAB search path or in the startup folder. Use the `-batch` option in non-interactive scripting or command line work flows. Do not use this option with the `-r` option. To test if a session of MATLAB is running in batch mode, call the `batchStartupOptionUsed` function. Example: `-batch "myscript"`

Option	Result
`-r statement`	Execute the MATLAB *`statement`*. Use this option for interactive work flows. Do not use this option with the `-batch` option. **Note** To set the initial working folder, use the `-sd` option. For example: `-sd folder` Example: `-r "disp(['Current folder: ' pwd])"` Example: `-r "myscript"`

Use Single Computational Thread

By default, MATLAB uses the multithreading capabilities of the computer on which it is running.

Option	Result
`-singleCompThread`	Limit MATLAB to a single computational thread.

Disable Searching Custom Java Class Path

Option	Result
`-nouserjavapath`	Disable use of `javaclasspath.txt` and `javalibrarypath.txt` files. For more information, see “Specifying Java Startup Options”.

OpenGL Library Options

These options control the use of software OpenGL libraries when MATLAB detects a graphics driver with known issues. For more information, see “Graphics Features That Have Specific Requirements”.

Option	Result
`-softwareopengl`	Force MATLAB to start with software OpenGL libraries.

Option	Result
`-nosoftwareopengl`	Disable auto-selection of OpenGL software.

COM Automation Server Options

Option	Result
`-automation`	Start MATLAB as a Component Object Model (COM) Automation server. MATLAB does not display the splash screen and minimizes the window. Use for a single call to MATLAB.
`-regserver`	Register MATLAB as a COM Automation server in the Windows registry. You must have administrator privileges to change the Windows registry. Based on your User Account Control (UAC) settings, you might need to right-click a Windows Command Processor and select **Run as administrator**. If that option is not available, contact your system administrator. Alternatively, you can register MATLAB from the MATLAB command prompt. Type: `regmatlabserver` For more information, see “Register MATLAB as Automation Server” and `regmatlabserver`.
`-unregserver`	Remove MATLAB COM server entries from the registry.

Wait for MATLAB to Terminate

By default, when you call the `matlab` command from a script, the command starts MATLAB and then immediately executes the next statements in the script. The `-wait` option pauses the script until MATLAB terminates.

Option	Result
`-wait`	Use in a script to process the results from MATLAB. Calling MATLAB with this option blocks the script from continuing until the results are generated.

Specify License File

For more information, see “Understanding Network License Files” (Installation, Licensing, and Activation).

Option	Result
`-c` *`license`*	Use the License File, *`license`*, specified as a string, a semicolon-separated list of license file names, or a `port@host` entry. If specifying multiple files, separate the names by semicolons and enclose the entire list in quotation marks. If the path to your license file contains a space, enclose the path name in quotation marks. Example: `-c "c:\TMW license \license_agreement.txt"`

Help Options

Option	Result
`-h`	Display options without starting MATLAB.
`-help`	Same as `-h` option.
`-?`	Same as `-h` option.

Examples

Startup Without Splash Screen

```
matlab -nosplash
```

Copy Command Window Output into `output.log` File

```
matlab -logfile output.log
```

Return Exit Code on Windows

To return an exit status on the command line, start MATLAB with the `-wait` option.

From the operating system prompt, type:

```
matlab -wait
```

Tips

- To add folders to the MATLAB search path at , set the `MATLABPATH` environment variable before running the `matlab` command. For more information, see "Add Folders to the MATLAB Search Path at Startup".

Compatibility Considerations

`-r` option for non-interactive use is not recommended

Not recommended starting in R2019a

To start MATLAB non-interactively, use the `-batch` option. To start MATLAB interactively, use the `-r` option.

See Also

`batchStartupOptionUsed` | `matlab (Linux)` | `matlab (macOS)` | `matlabroot` | `memory` | `quit` | `userpath`

Topics

"Start MATLAB on Windows Platforms"
"MATLAB Startup Folder"
"Specify Startup Options"

max

Maximum elements of an array

Syntax

```
M = max(A)
M = max(A,[],dim)
M = max(A,[],nanflag)
M = max(A,[],dim,nanflag)
[M,I] = max( ___ )

M = max(A,[],'all')
M = max(A,[],vecdim)
M = max(A,[],'all',nanflag)
M = max(A,[],vecdim,nanflag)

[M,I] = max(A,[], ___ ,'linear')

C = max(A,B)
C = max(A,B,nanflag)
```

Description

`M = max(A)` returns the maximum elements of an array.

- If `A` is a vector, then `max(A)` returns the maximum of `A`.
- If `A` is a matrix, then `max(A)` is a row vector containing the maximum value of each column.
- If `A` is a multidimensional array, then `max(A)` operates along the first array dimension whose size does not equal `1`, treating the elements as vectors. The size of this dimension becomes `1` while the sizes of all other dimensions remain the same. If `A` is an empty array whose first dimension has zero length, then `max(A)` returns an empty array with the same size as `A`.

`M = max(A,[],dim)` returns the maximum element along dimension `dim`. For example, if `A` is a matrix, then `max(A,[],2)` is a column vector containing the maximum value of each row.

`M = max(A,[],nanflag)` specifies whether to include or omit `NaN` values in the calculation. For example, `max(A,[],'includenan')` includes all `NaN` values in `A` while `max(A,[],'omitnan')` ignores them.

`M = max(A,[],dim,nanflag)` also specifies the dimension to operate along when using the `nanflag` option.

`[M,I] = max( ___ )` also returns the index into the operating dimension that corresponds to the maximum value of `A` for any of the previous syntaxes.

`M = max(A,[],'all')` finds the maximum over all elements of `A`. This syntax is valid for MATLAB versions R2018b and later.

`M = max(A,[],vecdim)` computes the maximum over the dimensions specified in the vector `vecdim`. For example, if `A` is a matrix, then `max(A,[],[1 2])` computes the maximum over all elements in `A`, since every element of a matrix is contained in the array slice defined by dimensions 1 and 2.

`M = max(A,[],'all',nanflag)` computes the maximum over all elements of `A` when using the `nanflag` option.

`M = max(A,[],vecdim,nanflag)` specifies multiple dimensions to operate along when using the `nanflag` option.

`[M,I] = max(A,[], ___ ,'linear')` returns the linear index into `A` that corresponds to the maximum value in `A`.

`C = max(A,B)` returns an array with the largest elements taken from `A` or `B`.

`C = max(A,B,nanflag)` also specifies how to treat `NaN` values.

Examples

Largest Vector Element

Create a vector and compute its largest element.

```
A = [23 42 37 18 52];
M = max(A)
```

```
M = 52
```

Largest Complex Element

Create a complex vector and compute its largest element, that is, the element with the largest magnitude.

```
A = [-2+2i 4+i -1-3i];
max(A)
```

```
ans = 4.0000 + 1.0000i
```

Largest Element in Each Matrix Column

Create a matrix and compute the largest element in each column.

```
A = [2 8 4; 7 3 9]
```

```
A = 2×3

     2     8     4
     7     3     9
```

```
M = max(A)
```

```
M = 1×3

     7     8     9
```

Largest Element in Each Matrix Row

Create a matrix and compute the largest element in each row.

```
A = [1.7 1.2 1.5; 1.3 1.6 1.99]
```

```
A = 2×3

    1.7000    1.2000    1.5000
    1.3000    1.6000    1.9900
```

```
M = max(A,[],2)
```

```
M = 2×1

    1.7000
    1.9900
```

Largest Element Involving NaN

Create a vector and compute its maximum, excluding NaN values.

```
A = [1.77 -0.005 3.98 -2.95 NaN 0.34 NaN 0.19];
M = max(A,[],'omitnan')
```

```
M = 3.9800
```

`max(A)` will also produce this result since `'omitnan'` is the default option.

Use the `'includenan'` flag to return NaN.

```
M = max(A,[],'includenan')
```

```
M = NaN
```

Largest Element Indices

Create a matrix A and compute the largest elements in each column, as well as the row indices of A in which they appear.

```
A = [1 9 -2; 8 4 -5]
```

```
A = 2×3

     1     9    -2
     8     4    -5
```

```
[M,I] = max(A)
```

```
M = 1×3

     8     9    -2


I = 1×3

     2     1     1
```

Maximum of Array Page

Create a 3-D array and compute the maximum over each page of data (rows and columns).

```
A(:,:,1) = [2 4; -2 1];
A(:,:,2) = [9 13; -5 7];
A(:,:,3) = [4 4; 8 -3];
M1 = max(A,[],[1 2])
```

```
M1 =
M1(:,:,1) =

     4


M1(:,:,2) =

    13


M1(:,:,3) =

     8
```

Starting in R2018b, to compute the maximum over all dimensions of an array, you can either specify each dimension in the vector dimension argument, or use the `'all'` option.

```
M2 = max(A,[],[1 2 3])
```

```
M2 = 13
```

```
Mall = max(A,[],'all')
```

```
Mall = 13
```

Return Linear Indices

Create a matrix `A` and return the maximum value of each row in the matrix `M`. Use the `'linear'` option to also return the linear indices `I` such that `M = A(I)`.

```
A = [1 2 3; 4 5 6]
```

```
A = 2×3

     1     2     3
     4     5     6
```

```
[M,I] = max(A,[],2,'linear')
```

```
M = 2×1

     3
     6


I = 2×1

     5
     6
```

```
maxvals = A(I)
```

```
maxvals = 2×1

     3
```

```
     6
```

Largest Element Comparison

Create a matrix and return the largest value between each of its elements compared to a scalar.

```
A = [1 7 3; 6 2 9]
```

```
A = 2×3

     1     7     3
     6     2     9
```

```
B = 5;
C = max(A,B)
```

```
C = 2×3

     5     7     5
     6     5     9
```

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

- If `A` is complex, then `max(A)` returns the complex number with the largest magnitude. If magnitudes are equal, then `max(A)` returns the value with the largest magnitude and the largest phase angle.
- If `A` is a scalar, then `max(A)` returns `A`.
- If `A` is a 0-by-0 empty array, then `max(A)` is as well.

If `A` has type `categorical`, then it must be ordinal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `categorical` | `datetime` | `duration`
Complex Number Support: Yes

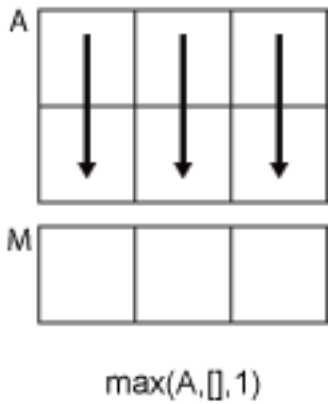
`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

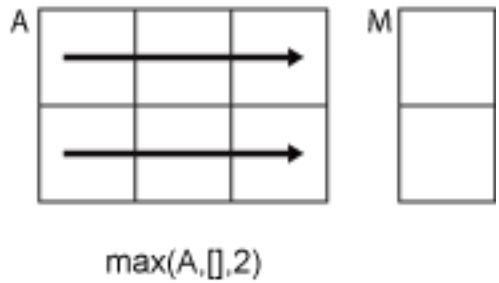
Dimension `dim` indicates the dimension whose length reduces to `1`. The `size(M,dim)` is `1`, while the sizes of all other dimensions remain the same, unless `size(A,dim)` is `0`. If `size(A,dim)` is `0`, then `max(A,dim)` returns an empty array with the same size as `A`.

Consider a two-dimensional input array, `A`:

- If `dim = 1`, then `max(A,[],1)` returns a row vector containing the largest element in each column.



- If `dim = 2`, then `max(A,[],2)` returns a column vector containing the largest element in each row.

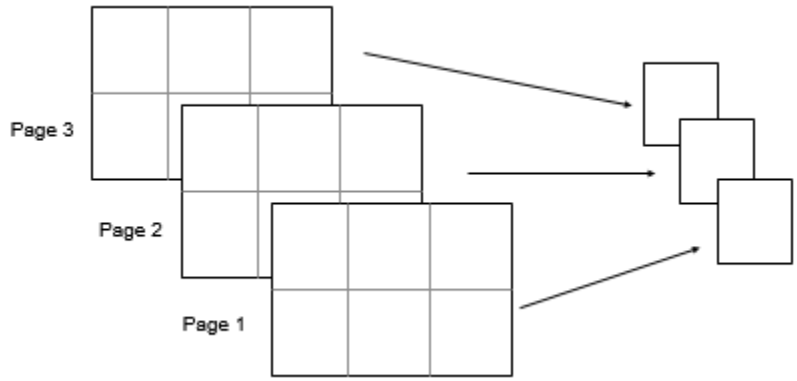


`max` returns `A` if `dim` is greater than `ndims(A)`.

`vecdim` — Vector of dimensions
vector of positive integers

Vector of dimensions, specified as a vector of positive integers. Each element represents a dimension of the input array. The lengths of the output in the specified operating dimensions are 1, while the others remain the same.

Consider a 2-by-3-by-3 input array, `A`. Then `max(A,[],[1 2])` returns a 1-by-1-by-3 array whose elements are the maximums computed over each page of `A`.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

B — Additional input array
scalar | vector | matrix | multidimensional array

Additional input array, specified as a scalar, vector, matrix, or multidimensional array. Numeric inputs A and B must either be the same size or have sizes that are compatible (for example, A is an M-by-N matrix and B is a scalar or 1-by-N row vector). For more information, see “Compatible Array Sizes for Basic Operations”.

If A and B are datetime, duration, or categorical arrays, then they must be the same size unless one is a scalar.

- A and B must be the same data type unless one is a `double`. In that case, the data type of the other array can be `single`, `duration`, or any integer type.
- If A and B are ordinal categorical arrays, they must have the same sets of categories with the same order.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `categorical` | `datetime` | `duration`
Complex Number Support: Yes

`nanflag` — NaN condition
`'omitnan'` (default) | `'includenan'`

NaN condition, specified as one of these values:

- `'omitnan'` — Ignore all NaN values in the input. If all elements are NaN, then `max` returns the first one.
- `'includenan'` — Include the NaN values in the input for the calculation.

For `datetime` arrays, you can also use `'omitnat'` or `'includenat'` to omit and include NaT values, respectively.

For `categorical` arrays, you can also use `'omitundefined'` or `'includeundefined'` to omit and include undefined values, respectively.

Data Types: `char`

Output Arguments

`M` — Maximum values
scalar | vector | matrix | multidimensional array

Maximum values, returned as a scalar, vector, matrix, or multidimensional array. `size(M,dim)` is `1`, while the sizes of all other dimensions match the size of the

corresponding dimension in `A`, unless `size(A,dim)` is `0`. If `size(A,dim)` is `0`, then `M` is an empty array with the same size as `A`.

`I` — Index
scalar | vector | matrix | multidimensional array

Index, returned as a scalar, vector, matrix, or multidimensional array. `I` is the same size as the first output.

When `'linear'` is not specified, `I` is the index into the operating dimension. When `'linear'` is specified, `I` contains the linear indices of `A` corresponding to the maximum values.

If the largest element occurs more than once, then `I` contains the index to the first occurrence of the value.

`C` — Maximum elements from `A` or `B`
scalar | vector | matrix | multidimensional array

Maximum elements from `A` or `B`, returned as a scalar, vector, matrix, or multidimensional array. The size of `C` is determined by implicit expansion of the dimensions of `A` and `B`. For more information, see “Compatible Array Sizes for Basic Operations”.

The data type of `C` depends on the data types of `A` and `B`:

- If `A` and `B` are the same data type, then `C` matches the data type of `A` and `B`.
- If either `A` or `B` is `single`, then `C` is `single`.
- If either `A` or `B` is an integer data type with the other a scalar `double`, then `C` assumes the integer data type.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If you specify `dim`, then it must be a constant.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).
- See “Code Generation for Complex Data with Zero-Valued Imaginary Parts” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bounds` | `maxk` | `mean` | `median` | `min` | `sort`

Topics

“Array Indexing”

Introduced before R2006a

MaximizeCommandWindow

Open Automation server window

Syntax

IDL Method Signature

```
HRESULT MaximizeCommandWindow(void)
```

Microsoft Visual Basic Client

```
MaximizeCommandWindow
```

MATLAB Client

```
MaximizeCommandWindow(h)
```

Description

`MaximizeCommandWindow(h)` displays the window for the server attached to handle `h`, and makes it the currently active window on the desktop.

`MaximizeCommandWindow` restores the window to the size it had at the time it was minimized, not to the maximum size on the desktop. If the server window was not previously in a minimized state, `MaximizeCommandWindow` does nothing.

Examples

Adjust MATLAB Command Window in Visual Basic .NET

This example shows how to minimize and maximize the command window in a MATLAB Automation server. Create an application from the following code.

```
type adjustcommandwindow.vb
```

```
Dim Matlab As Object

Matlab = CreateObject("matlab.application")
Matlab.MinimizeCommandWindow

'Now return the server window to its former state on
'the desktop and make it the currently active window.

Matlab.MaximizeCommandWindow
```

Adjust MATLAB Command Window in VBA

This example shows how to minimize and maximize the command window in a MATLAB Automation server. Create an application from the following code.

```
type adjustcommandwindow.vba
```

```
Dim Matlab As Object

Set Matlab = CreateObject("matlab.application")
Matlab.MinimizeCommandWindow

'Now return the server window to its former state on
'the desktop and make it the currently active window.

Matlab.MaximizeCommandWindow
```

See Also

MinimizeCommandWindow

Introduced before R2006a

maxk

Find k largest elements of array

Syntax

```
B = maxk(A,k)
B = maxk(A,k,dim)
B = maxk( ___ ,'ComparisonMethod',c)
[B,I] = maxk( ___ )
```

Description

`B = maxk(A,k)` returns the `k` largest elements of `A`.

- If `A` is a vector, then `maxk` returns a vector containing the `k` largest elements of `A`.
- If `A` is a matrix, then `maxk` returns a matrix whose columns contain the `k` largest elements of each column of `A`.
- If `A` is a multidimensional array, then `maxk` returns the `k` largest elements along the first dimension whose size does not equal 1.

`B = maxk(A,k,dim)` determines the `k` largest elements of `A` along dimension `dim`.

`B = maxk( ___ ,'ComparisonMethod',c)` optionally specifies how to compare elements of `A` for any of the previous syntaxes. For example, `maxk(A,k,'ComparisonMethod','abs')` returns the `k` largest elements of `A` according to their absolute values.

`[B,I] = maxk( ___ )` finds the indices of the largest `k` values of `A` and returns them in `I`.

Examples

Largest Vector Elements

Compute the largest 3 elements of a vector.

```
A = 1:10;
B = maxk(A,3)
```

```
B = 1×3

    10     9     8
```

Largest Elements of Matrix Rows

Compute the largest 3 elements of each row of a matrix.

```
A = magic(5)
```

```
A = 5×5

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

```
B = maxk(A,3,2)
```

```
B = 5×3

    24    17    15
    23    16    14
    22    20    13
    21    19    12
    25    18    11
```

Complex Vector

Compute the 2 largest elements of a complex vector according to their magnitude, and return the indices where they are located in the input vector.

```
A = [2-2i 5+i -7-3i -1+i]
```

```
A = 1×4 complex

   2.0000 - 2.0000i   5.0000 + 1.0000i  -7.0000 - 3.0000i  -1.0000 + 1.0000i
```

```
[B,I] = maxk(A,2,'ComparisonMethod','abs')
```

```
B = 1×2 complex

  -7.0000 - 3.0000i   5.0000 + 1.0000i


I = 1×2

     3     2
```

Input Arguments

A — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

- If A is a vector, then maxk returns a vector containing the k largest elements of A.
- If A is a matrix, then maxk returns a matrix whose columns contain the k largest elements of each column of A.
- If A is a multidimensional array, then maxk returns the k largest elements along the first dimension whose size does not equal 1.

If A has type categorical, then it must be ordinal.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char | categorical | datetime | duration

Complex Number Support: Yes

k — Number of maxima
nonnegative integer scalar

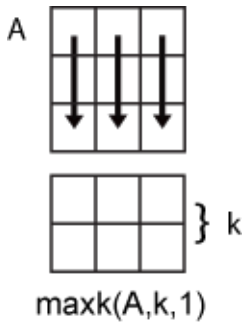
Number of maxima to return, specified as a positive integer scalar. If `k` is greater than or equal to the number of elements in the operating dimension, then `maxk` sorts the input array along that dimension.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

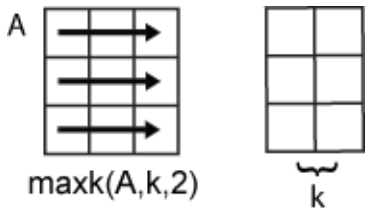
`dim` — Operating dimension
positive integer scalar

Operating dimension, specified as a positive integer scalar. By default, `maxk` operates along the first dimension whose size does not equal 1.

For example, if `A` is a matrix, then `maxk(A,k,1)` operates along the rows of `A`, computing maximums for each column.



`maxk(A,k,2)` operates along the columns of `A`, computing maximums for each row.



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

c — Comparison method
`'auto'` (default) | `'real'` | `'abs'`

Comparison method, specified as one of the following:

- `'auto'` — Compare elements of input `A` by `real(A)` when `A` is real, and by `abs(A)` when `A` is complex.
- `'real'` — Compare elements of input `A` by `real(A)` when `A` is real or complex. If `A` has elements with equal real parts, then use `imag(A)` to break ties.
- `'abs'` — Compare elements of input `A` by `abs(A)` when `A` is real or complex. If `A` has elements with equal magnitude, then use `angle(A)` in the interval (-π,π] to break ties.

Output Arguments

B — Output array
scalar | vector | matrix | multidimensional array

Output array, returned as a scalar, vector, matrix, or multidimensional array. `maxk` returns the `k` elements in order from largest to smallest. The order of the elements in `B` preserves the order of any equal elements in `A`.

I — Index array
scalar | vector | matrix | multidimensional array

Index array, returned as a vector, matrix, or multidimensional array. `I` is the same size as `B`. If the output array `B` contains repeated elements, then the order of their indices in `I` matches the order in which they appear in the input array.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If `A` is complex with all zero imaginary parts, then MATLAB might convert `A` to `real(A)` before calling `maxk(A)`. In this case, MATLAB compares elements of `A` by `real(A)`, but the generated code compares elements of `A` by `abs(A)`. To make the generated code match MATLAB, use `maxk(real(A),k)` or `maxk(A,k,'ComparisonMethod','real')`. See “Code Generation for Complex Data with Zero-Valued Imaginary Parts” (MATLAB Coder).
- If you supply `dim`, then it must be constant.
- For limitations related to variable-size inputs, see “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`max` | `mink` | `topkrows`

Introduced in R2017b

maxNumCompThreads

Control maximum number of computational threads

Syntax

```
N = maxNumCompThreads
LASTN = maxNumCompThreads(N)
LASTN = maxNumCompThreads('automatic')
```

Description

`N = maxNumCompThreads` returns the current maximum number of computational threads `N`.

`LASTN = maxNumCompThreads(N)` sets the maximum number of computational threads to `N`, and returns the previous maximum number of computational threads, `LASTN`.

`LASTN = maxNumCompThreads('automatic')` sets the maximum number of computational threads using what the MATLAB software determines to be the most desirable. It additionally returns the previous maximum number of computational threads, `LASTN`.

Currently, the maximum number of computational threads is equal to the number of physical cores on your machine.

Note Setting the maximum number of computational threads using `maxNumCompThreads` does not propagate to your next MATLAB session.

You can set the `-singleCompThread` option when starting MATLAB to limit MATLAB to a single computational thread. By default, MATLAB makes use of the multithreading capabilities of the computer on which it is running.

On Windows platforms with multiple processor groups, the default maximum number of computational threads is equal to the number of physical cores within the processor group that is running MATLAB.

Introduced in R2007b

mean

Average or mean value of array

Syntax

```
M = mean(A)
M = mean(A,'all')
M = mean(A,dim)
M = mean(A,vecdim)
M = mean( ___ ,outtype)
M = mean( ___ ,nanflag)
```

Description

`M = mean(A)` returns the mean on page 1-9554 of the elements of `A` along the first array dimension whose size does not equal 1.

- If `A` is a vector, then `mean(A)` returns the mean of the elements.
- If `A` is a matrix, then `mean(A)` returns a row vector containing the mean of each column.
- If `A` is a multidimensional array, then `mean(A)` operates along the first array dimension whose size does not equal 1, treating the elements as vectors. This dimension becomes `1` while the sizes of all other dimensions remain the same.

`M = mean(A,'all')` computes the mean over all elements of `A`. This syntax is valid for MATLAB versions R2018b and later.

`M = mean(A,dim)` returns the mean along dimension `dim`. For example, if `A` is a matrix, then `mean(A,2)` is a column vector containing the mean of each row.

`M = mean(A,vecdim)` computes the mean based on the dimensions specified in the vector `vecdim`. For example, if `A` is a matrix, then `mean(A,[1 2])` is the mean of all elements in `A`, since every element of a matrix is contained in the array slice defined by dimensions 1 and 2.

`M = mean( ___ ,outtype)` returns the mean with a specified data type, using any of the input arguments in the previous syntaxes. `outtype` can be `'default'`, `'double'`, or `'native'`.

`M = mean( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `mean(A,'includenan')` includes all `NaN` values in the calculation while `mean(A,'omitnan')` ignores them.

Examples

Mean of Matrix Columns

Create a matrix and compute the mean of each column.

```
A = [0 1 1; 2 3 2; 1 3 2; 4 2 2]
```

```
A = 4×3

     0     1     1
     2     3     2
     1     3     2
     4     2     2
```

```
M = mean(A)
```

```
M = 1×3

    1.7500    2.2500    1.7500
```

Mean of Matrix Rows

Create a matrix and compute the mean of each row.

```
A = [0 1 1; 2 3 2; 3 0 1; 1 2 3]
```

```
A = 4×3
```

```
     0     1     1
     2     3     2
     3     0     1
     1     2     3
```

```
M = mean(A,2)
```

```
M = 4×1

    0.6667
    2.3333
    1.3333
    2.0000
```

Mean of 3-D Array

Create a 4-by-2-by-3 array of integers between 1 and 10 and compute the mean values along the second dimension.

```
A = gallery('integerdata',10,[4,2,3],1);
M = mean(A,2)
```

```
M =
M(:,:,1) =

    9.5000
    6.5000
    9.5000
    6.0000


M(:,:,2) =

    1.5000
    4.0000
    7.5000
    7.5000


M(:,:,3) =
```

```
    7.0000
    2.5000
    4.0000
    5.5000
```

Mean of Array Page

Create a 3-D array and compute the mean over each page of data (rows and columns).

```
A(:,:,1) = [2 4; -2 1];
A(:,:,2) = [9 13; -5 7];
A(:,:,3) = [4 4; 8 -3];
M1 = mean(A,[1 2])
```

```
M1 =
M1(:,:,1) =

    1.2500


M1(:,:,2) =

     6


M1(:,:,3) =

    3.2500
```

Starting in R2018b, to compute the mean over all dimensions of an array, you can either specify each dimension in the vector dimension argument, or use the `'all'` option.

```
M2 = mean(A,[1 2 3])
```

```
M2 = 3.5000
```

```
Mall = mean(A,'all')
```

```
Mall = 3.5000
```

Mean of Single-Precision Array

Create a single-precision vector of ones and compute its single-precision mean.

```
A = single(ones(10,1));
M = mean(A,'native')
```

```
M = single
    1
```

The result is also in single precision.

```
class(M)
```

```
ans =
'single'
```

Mean Excluding NaN

Create a vector and compute its mean, excluding `NaN` values.

```
A = [1 0 0 1 NaN 1 NaN 0];
M = mean(A,'omitnan')
```

```
M = 0.5000
```

If you do not specify `'omitnan'`, then `mean(A)` returns `NaN`.

Input Arguments

A — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

- If `A` is a scalar, then `mean(A)` returns `A`.
- If `A` is an empty 0-by-0 matrix, then `mean(A)` returns `NaN`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `datetime` | `duration`

`dim` — Dimension to operate along

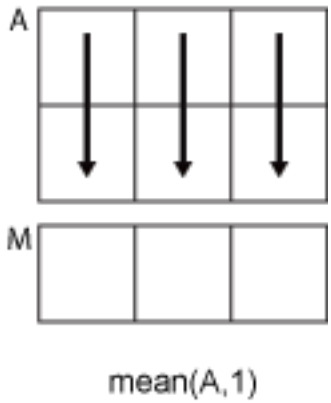
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal `1`.

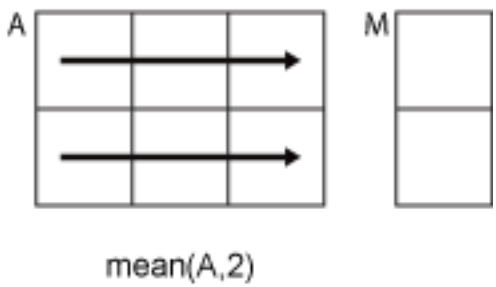
Dimension `dim` indicates the dimension whose length reduces to `1`. The `size(M,dim)` is `1`, while the sizes of all other dimensions remain the same.

Consider a two-dimensional input array, `A`.

- If `dim = 1`, then `mean(A,1)` returns a row vector containing the mean of the elements in each column.



- If `dim = 2`, then `mean(A,2)` returns a column vector containing the mean of the elements in each row.



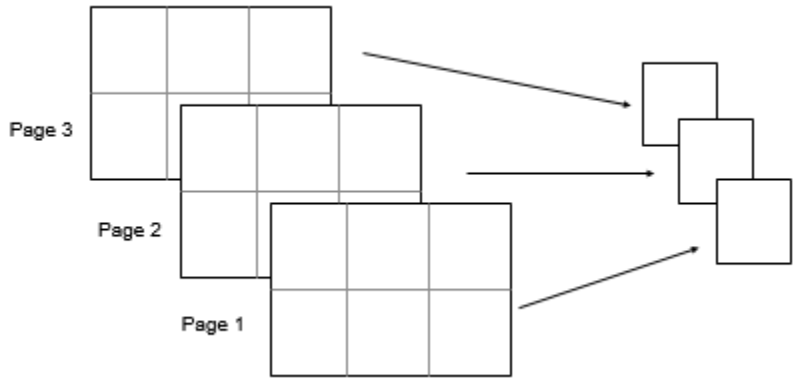
`mean` returns `A` when `dim` is greater than `ndims(A)` or when `size(A,dim)` is `1`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`vecdim` — Vector of dimensions
vector of positive integers

Vector of dimensions, specified as a vector of positive integers. Each element represents a dimension of the input array. The lengths of the output in the specified operating dimensions are 1, while the others remain the same.

Consider a 2-by-3-by-3 input array, `A`. Then `mean(A,[1 2])` returns a 1-by-1-by-3 array whose elements are the means over each page of `A`.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`outtype` — Output data type
`'default'` (default) | `'double'` | `'native'`

Output data type, specified as `'default'`, `'double'`, or `'native'`. These options also specify the data type in which the operation is performed.

`outtype`	Output data type
`'default'`	`double`, unless the input data type is `single`, `duration`, or `datetime`, in which case, the output is `'native'`

outtype	Output data type
'double'	double, unless the data type is duration or datetime, in which case, 'double' is not supported
'native'	same data type as the input, unless • Input data type is logical, in which case, the output is double • Input data type is char, in which case, 'native'is not supported

Data Types: char

nanflag — NaN condition
'includenan' (default) | 'omitnan'

NaN condition, specified as one of these values:

- 'includenan' — Include NaN values when computing the mean, resulting in NaN.
- 'omitnan' — Ignore all NaN values in the input.

For datetime arrays, you can also use 'omitnat' or 'includenat' to omit and include NaT values, respectively.

Data Types: char

More About

Mean

For a random variable vector A made up of N scalar observations, the mean is defined as

$$\mu = \frac{1}{N}\sum_{i=1}^{N} A_i .$$

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If you specify `dim`, then it must be a constant.
- The `outtype` and `nanflag` options must be constant character vectors.
- Integer types do not support the `'native'` output data type option.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- If you specify `dim`, then it must be a constant.
- The `outtype` and `nanflag` options must be constant character vectors.
- Integer types do not support the `'native'` output data type option.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `'native'` option is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `'native'` option is not supported.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`median` | `mode` | `std` | `sum` | `var`

Introduced before R2006a

matlab.unittest.measurement.chart.ComparisonPlot class

Package: `matlab.unittest.measurement.chart`
Superclasses: `matlab.graphics.chartcontainer.ChartContainer`, `matlab.graphics.chartcontainer.mixin.Legend`

Visually compare two sets of time experiment results

Description

Objects of the `ComparisonPlot` class provide visualization to compare the time measurement results of two equal-sized sets of performance tests. `ComparisonPlot` objects map baseline and measurement `TimeResult` instances to the *x*- and *y*-axes of 2-D scatter plots, respectively.

The `matlab.unittest.measurement.chart.ComparisonPlot` class is a `handle` class.

Class Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Creation

To create `ComparisonPlot` instances, use the `comparisonPlot` method.

Properties

`Scale` — Scale of *x*- and *y*-axes
`'log'` (default) | `'linear'`

Scale of the *x*- and *y*-axes of the `ComparisonPlot` object, specified as `'log'` or `'linear'`.

`SimilarityTolerance` — Allowable deviation of statistics ratio from one
`0.1` (default) | numeric scalar

Allowable deviation of the statistics ratio from one, for a pair of similar performance tests, specified as a numeric value between `0` and `1`.

`SimilarityTolerance` specifies the borders of a shaded region in the `ComparisonPlot` object. Data points that fall inside this region represent similar baseline and measurement entries.

See Also

`comparisonPlot` | `matlab.perftest.TimeResult`

Introduced in R2019b

matlab.unittest.measurement.MeasurementResult class

Package: `matlab.unittest.measurement`

Base class for classes holding measurement results

Description

The `MeasurementResult` class is the base class for classes that hold the results from running a measurement experiment on test suites. These classes contains the measurement values, categories, and timestamps, and information about the functional test results.

When you run an experiment, the testing framework returns an object of a concrete class that is derived from the `MeasurementResult` class. One object for each `Test` object in the suite.

Properties

`Name` — Name of the test suite element

character vector

Name of the test suite element that corresponds to the measurement result, returned as a character vector.

`Valid` — Indicator if measurement is valid

`logical` 0 or 1

Indicator if the measurement is valid, returned as `logical` 0 or 1. A measurement is valid (`logical` 1) if it is collected from a passing test. Otherwise it is invalid (`logical` 0).

`Samples` — Information about sample measurements

table

Information about sample measurements, returned as a table containing these columns:

Column Name	Description
Name	Name of the test suite element.
MeasuredValue	Value of the measurement. Experiments can give this column a more specific name, such as `MeasuredTime`.
Timestamp	Time of measurement.
Host	Machine name.
Platform	Platform architecture.
Version	MATLAB Version.
RunIdentifier	Unique identifier of measurement run.

The `Samples` table does not include information from warm-up runs.

`TestActivity` — Information about all test activity
table

Information about all test activity, including the warm-up measurements, returned as a table. This table is useful when you want all the data, such as during debugging or when performing a more involved data analysis. The `TestActivity` table contains these columns in addition to the columns described for `Samples`:

Column Name	Description
Passed	Logical value indicating if the test passed
Failed	Logical value indicating if the test failed
Incomplete	Logical value indicating if test run is incomplete
Objective	Categorical value indicating the type of measurement: `sample` or `warmup`
TestResult	`TestResult` object from the functional test

Methods

samplefun	Apply function across samples of `MeasurementResult` array
sampleSummary	Create table of summary statistics from `MeasurementResult` array

See Also

`matlab.perftest.TimeResult` | `matlab.unittest.TestResult` | `matlab.unittest.TestSuite` | `matlab.unittest.measurement.DefaultMeasurementResult`

Introduced in R2016a

matlab.unittest.measurement.DefaultMeasurementResult class

Package: `matlab.unittest.measurement`
Superclasses: `matlab.unittest.measurement.MeasurementResult`

Default implementation of `MeasurementResult` class

Description

`MeasurementResult` objects saved in releases before R2019a are loaded as `DefaultMeasurementResult` objects. These objects contain the data save in the original `MeasurementResult` object.

For information on `DefaultMeasurementResult` objects, see `matlab.unittest.measurement.MeasurementResult`.

Creation

The test framework creates objects of this class when loading previously save `MeasurementResult` objects.

See Also

`matlab.perftest.TimeResult` | `matlab.unittest.measurement.MeasurementResult`

Topics

"Test Performance Using Scripts or Functions"
"Test Performance Using Classes"

Introduced in R2019a

samplefun

Class: `matlab.unittest.measurement.MeasurementResult`
Package: `matlab.unittest.measurement`

Apply function across samples of `MeasurementResult` array

Syntax

```
[B1,...,Bm] = samplefun(fh,R)
[B1,...,Bm] = samplefun(fh,R,'UniformOutput',tf)
```

Description

`[B1,...,Bm] = samplefun(fh,R)` applies a function `fh` across the samples on each element of a `MeasurementResult` array. Each output argument from `samplefun` corresponds to an output argument from `fh` and has the same size and shape as `R`.

`[B1,...,Bm] = samplefun(fh,R,'UniformOutput',tf)` indicates if the output of `fh` can be returned without encapsulation in a cell array. By default, `fh` must return scalar values that can be concatenated into an array.

Input Arguments

`fh` — Function to apply
function handle

Function to apply across the samples on each element of a `MeasurementResult` array, specified as a function handle.

`R` — Results from running a measurement experiment on a test suite
`MeasurementResult` array

Results from running a measurement experiment on a test suite, specified as a `MeasurementResult` array.

`tf` — Indicator whether `fh` returns uniform values
`true` (default) | `false`

Indicator whether `fh` returns uniform values, specified as `true` or `false`. `tf` is `true` by default, which indicates that `fh` returns scalar values that can be concatenated into an array. If `tf` is `false`, the outputs of `fh` can have different sizes and data types. `samplefun` returns these nonuniform outputs in a cell array.

Examples

Apply Function to Measurement Results

In your current working folder, create a class-based test, `preallocationTest.m`, that compares different methods of preallocation.

```
classdef preallocationTest < matlab.perftest.TestCase
    methods(Test)
        function testOnes(testCase)
            x = ones(1,1e7);
        end

        function testIndexingWithVariable(testCase)
            id = 1:1e7;
            x(id) = 1;
        end

        function testIndexingOnLHS(testCase)
            x(1:1e7) = 1;
        end

        function testForLoop(testCase)
            for i=1:1e7
                x(i) = 1;
            end
        end

    end
end
```

Create a test suite.

```
suite = testsuite('preallocationTest');
```

Construct a variable time experiment, and run the tests.

```
import matlab.perftest.TimeExperiment
experiment = TimeExperiment.limitingSamplingError;
R = run(experiment,suite);
```

```
Running preallocationTest
..........
..........
..........
......Warning: Target Relative Margin of Error not met after running the MaxSamples
for preallocationTest/testOnes.
....
..........
..........
..........
..........
.....
Done preallocationTest
__________
```

For each test element, find the mean time of the samples.

```
M = samplefun(@mean,R)
```

```
M =

    0.0350    0.1351    0.0789    0.7337
```

For each test element, find the minimum time and index to the minimum time.

```
[M,I] = samplefun(@min,R)
```

```
M =

    0.0258    0.1169    0.0691    0.6531


I =
```

```
    27     3     1     1
```

Apply Custom Function to Measurement Results

In your current working folder, create a class-based test, `preallocationTest.m`, that compares different methods of preallocation.

```
classdef preallocationTest < matlab.perftest.TestCase
    methods(Test)
        function testOnes(testCase)
            x = ones(1,1e7);
        end

        function testIndexingWithVariable(testCase)
            id = 1:1e7;
            x(id) = 1;
        end

        function testIndexingOnLHS(testCase)
            x(1:1e7) = 1;
        end

        function testForLoop(testCase)
            for i=1:1e7
                x(i) = 1;
            end
        end

    end
end
```

Create a test suite.

```
suite = testsuite('preallocationTest');
```

Construct a fixed time experiment with 26 sample measurements, and run the tests.

```
import matlab.perftest.TimeExperiment
experiment = TimeExperiment.withFixedSampleSize(26);
R = run(experiment,suite);
```

```
Running preallocationTest
..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
....
Done preallocationTest
__________
```

In your current working folder, create a function, `customSampleFun`, that computes the mean of each of the 26 samples, converts the mean to milliseconds, and returns a character vector indicating if the mean time was fast or slow.

```
function [mean_ms,speed] = customSampleFun(S)
threshold_ms = 100;
mean_ms = mean(S)*1e3;
if mean_ms < threshold_ms
    speed = 'fast';
else
    speed = 'slow';
end
end
```

Apply `customSampleFun` to each element in the `MeasurementResult` array. Since the character vectors aren't scalar, specify `UniformOutput` as false.

```
[mean_ms,speed] = samplefun(@customSampleFun,R,'UniformOutput',false)
```

```
mean_ms =

  1×4 cell array

    [30.9500]    [142.7037]    [83.9830]    [806.3446]
```

```
speed =

  1×4 cell array

    'fast'    'slow'    'fast'    'slow'
```

See Also

sampleSummary

Introduced in R2017a

sampleSummary

Class: `matlab.unittest.measurement.MeasurementResult`
Package: `matlab.unittest.measurement`

Create table of summary statistics from `MeasurementResult` array

Syntax

```
T = sampleSummary(R)
```

Description

`T = sampleSummary(R)` creates a table of summary statistics from a `MeasurementResult` array.

Input Arguments

R — Results array
`MeasurementResult` array

Results array from running a measurement experiment on a test suite, specified as a `MeasurementResult` array.

Output Arguments

T — Measurement sample summary
table

Measurement sample summary, returned as a table. The table contains the following columns: `Name`, `SampleSize`, `Mean`, `StandardDeviation`, `Min`, `Median`, and `Max`.

Examples

Generate Table of Summary Statistics

In your current working folder, create a class-based test, `preallocationTest.m`, that compares different methods of preallocation.

```
classdef preallocationTest < matlab.perftest.TestCase
    methods(Test)
        function testOnes(testCase)
            x = ones(1,1e7);
        end

        function testIndexingWithVariable(testCase)
            id = 1:1e7;
            x(id) = 1;
        end

        function testIndexingOnLHS(testCase)
            x(1:1e7) = 1;
        end

        function testForLoop(testCase)
            for i=1:1e7
                x(i) = 1;
            end
        end

    end
end
```

Create a test suite.

```
suite = testsuite('preallocationTest');
```

Construct a time experiment with a variable number of sample measurements, and run the tests.

```
import matlab.perftest.TimeExperiment
experiment = TimeExperiment.limitingSamplingError;
R = run(experiment,suite);
```

```
Running preallocationTest
..........
..........
..........
..........
..........
.....
Done preallocationTest
__________
```

Create a table of summary statistics from the result array R.

```
T = sampleSummary(R)
```

```
T =

  4×7 table array

                       Name                       SampleSize      Mean      StandardDev
    ____________________________________________   __________   ________   ___________

    preallocationTest/testOnes                      4            0.02649    0.00086703
    preallocationTest/testIndexingWithVariable     16            0.13356      0.014525
    preallocationTest/testIndexingOnLHS            13           0.073571     0.0073962
    preallocationTest/testForLoop                   6            0.74768       0.03897
```

See Also

samplefun

Introduced in R2017a

median

Median value of array

Syntax

```
M = median(A)
M = median(A,'all')
M = median(A,dim)
M = median(A,vecdim)
M = median( ___ ,nanflag)
```

Description

`M = median(A)` returns the median value of `A`.

- If `A` is a vector, then `median(A)` returns the median value of `A`.
- If `A` is a nonempty matrix, then `median(A)` treats the columns of `A` as vectors and returns a row vector of median values.
- If `A` is an empty 0-by-0 matrix, `median(A)` returns `NaN`.
- If `A` is a multidimensional array, then `median(A)` treats the values along the first array dimension whose size does not equal `1` as vectors. The size of this dimension becomes `1` while the sizes of all other dimensions remain the same.

`median` computes natively in the numeric class of `A`, such that `class(M) = class(A)`.

`M = median(A,'all')` computes the median over all elements of `A`. This syntax is valid for MATLAB versions R2018b and later.

`M = median(A,dim)` returns the median of elements along dimension `dim`. For example, if `A` is a matrix, then `median(A,2)` is a column vector containing the median value of each row.

`M = median(A,vecdim)` computes the median based on the dimensions specified in the vector `vecdim`. For example, if `A` is a matrix, then `median(A,[1 2])` is the median over

all elements in `A`, since every element of a matrix is contained in the array slice defined by dimensions 1 and 2.

`M = median( ___ ,nanflag)` optionally specifies whether to include or omit `NaN` values in the median calculation for any of the previous syntaxes. For example, `median(A,'omitnan')` ignores all `NaN` values in `A`.

Examples

Median of Matrix Columns

Define a 4-by-3 matrix.

```
A = [0 1 1; 2 3 2; 1 3 2; 4 2 2]
```

```
A = 4×3

     0     1     1
     2     3     2
     1     3     2
     4     2     2
```

Find the median value of each column.

```
M = median(A)
```

```
M = 1×3

    1.5000    2.5000    2.0000
```

For each column, the median value is the mean of the middle two numbers in sorted order.

Median of Matrix Rows

Define a 2-by-3 matrix.

```
A = [0 1 1; 2 3 2]
```

```
A = 2×3

     0     1     1
     2     3     2
```

Find the median value of each row.

```
M = median(A,2)
```

```
M = 2×1

     1
     2
```

For each row, the median value is the middle number in sorted order.

Median of 3-D Array

Create a 1-by-3-by-4 array of integers between `1` and `10`.

```
A = gallery('integerdata',10,[1,3,4],1)
```

```
A =
A(:,:,1) =

    10     8    10


A(:,:,2) =

     6     9     5


A(:,:,3) =

     9     6     1


A(:,:,4) =
```

```
     4     9     5
```

Find the median values of this 3-D array along the second dimension.

```
M = median(A)
```

```
M =
M(:,:,1) =

    10


M(:,:,2) =

     6


M(:,:,3) =

     6


M(:,:,4) =

     5
```

This operation produces a 1-by-1-by-4 array by computing the median of the three values along the second dimension. The size of the second dimension is reduced to 1.

Compute the median along the first dimension of A.

```
M = median(A,1);
isequal(A,M)
```

```
ans = logical
   1
```

This command returns the same array as A because the size of the first dimension is 1.

Median of Array Page

Create a 3-D array and compute the median over each page of data (rows and columns).

```
A(:,:,1) = [2 4; -2 1];
A(:,:,2) = [6 2; -5 3];
A(:,:,3) = [4 4; 7 -3];
M1 = median(A,[1 2])
```

```
M1 =
M1(:,:,1) =

    1.5000


M1(:,:,2) =

    2.5000


M1(:,:,3) =

     4
```

Starting in R2018b, to compute the median over all dimensions of an array, you can either specify each dimension in the vector dimension argument, or use the `'all'` option.

```
M2 = median(A,[1 2 3])
```

```
M2 = 2.5000
```

```
Mall = median(A,'all')
```

```
Mall = 2.5000
```

Median of 8-Bit Integer Array

Define a 1-by-4 vector of 8-bit integers.

```
A = int8(1:4)
```

```
A = 1x4 int8 row vector
```

```
   1   2   3   4
```

Compute the median value.

```
M = median(A),
```

```
M = int8
    3
```

```
class(M)
```

```
ans =
'int8'
```

`M` is the mean of the middle two numbers in sorted order returned as an 8-bit integer.

Median Excluding NaN

Create a vector and compute its median, excluding `NaN` values.

```
A = [1.77 -0.005 3.98 -2.95 NaN 0.34 NaN 0.19];
M = median(A,'omitnan')
```

```
M = 0.2650
```

Input Arguments

`A` — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array. `A` can be a numeric array, ordinal `categorical` array, `datetime` array, or `duration` array.

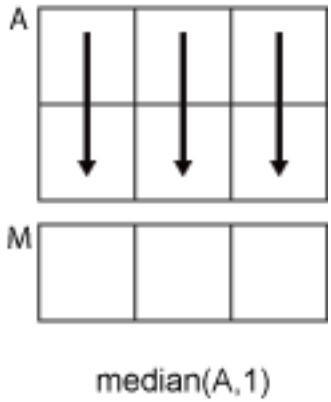
`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

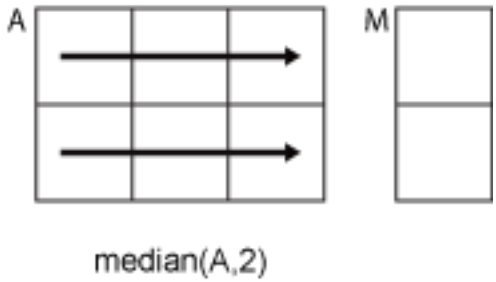
Dimension `dim` indicates the dimension whose length reduces to `1`. The `size(M,dim)` is `1`, while the sizes of all other dimensions remain the same.

Consider a two-dimensional input array, `A`.

- If `dim = 1`, then `median(A,1)` returns a row vector containing the median of the elements in each column.



- If `dim = 2`, then `median(A,2)` returns a column vector containing the median of the elements in each row.

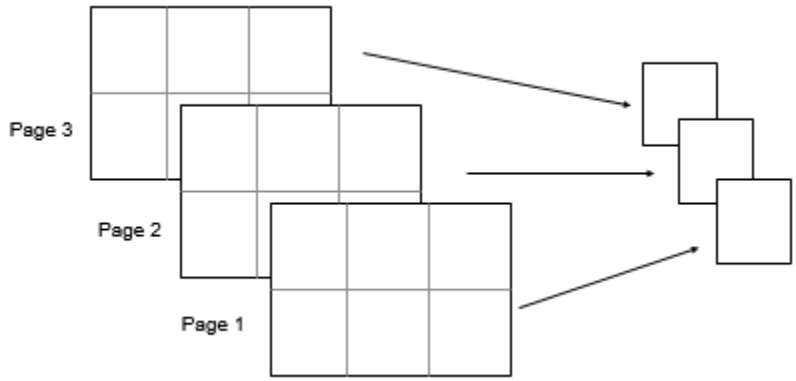


`median` returns `A` when `dim` is greater than `ndims(A)`.

`vecdim` — Vector of dimensions
vector of positive integers

Vector of dimensions, specified as a vector of positive integers. Each element represents a dimension of the input array. The lengths of the output in the specified operating dimensions are 1, while the others remain the same.

Consider a 2-by-3-by-3 input array, `A`. Then `median(A,[1 2])` returns a 1-by-1-by-3 array whose elements are the medians of each page of `A`.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`nanflag` — `NaN` condition
`'includenan'` (default) | `'omitnan'`

`NaN` condition, specified as one of these values:

- `'includenan'` — the median of input containing `NaN` values is also `NaN`.
- `'omitnan'` — all `NaN` values appearing in the input are ignored. Note: the `NaN` flags are not set to `0`.

You also can specify additional values for some data types.

- `'includeundefined'` and `'omitundefined'` — `categorical` input
- `'includenat'` and `'omitnat'` — `datetime` input

Algorithms

For ordinal categorical arrays, MATLAB interprets the median of an even number of elements as follows:

If the number of categories between the middle two values is ...	Then the median is ...
zero (values are from consecutive categories)	larger of the two middle values
an odd number	value from category occurring midway between the two middle values
an even number	value from larger of the two categories occurring midway between the two middle values

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

Input `A` must be a column vector to compute `median` in the first dimension.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If specified, `dim` must be a constant.
- “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder)
- “Code Generation for Complex Data with Zero-Valued Imaginary Parts” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`corrcoef` | `cov` | `max` | `mean` | `min` | `mode` | `std` | `var`

Introduced before R2006a

memmapfile

Create memory map to a file

Syntax

```
m = memmapfile(filename)
m = memmapfile(filename,Name,Value)
```

Description

`m = memmapfile(filename)` maps an existing file, `filename`, to memory and returns the memory map, `m`.

Memory-mapping is a mechanism that maps a portion of a file, or an entire file, on disk to a range of memory addresses within the MATLAB address space. Then, MATLAB can access files on disk in the same way it accesses dynamic memory, accelerating file reading and writing. Memory-mapping allows you to work with data in a file as if it were a MATLAB array.

`m = memmapfile(filename,Name,Value)` specifies the properties of `m` using one or more name-value pair arguments. For example, you can specify the format of the data in the file.

Examples

Map Entire File of `uint8` Data

At the command prompt, create a sample file in your current folder called `records.dat`, containing 10 `uint8` values.

```
myData = uint8(1:10)';

fileID = fopen('records.dat','w');
```

```
fwrite(fileID, myData,'uint8');
fclose(fileID);
```

Create a map for `records.dat`. When using `memmapfile`, the default data format is `uint8` so the file name is the only required input argument in this case.

```
m = memmapfile('records.dat')
```

```
m = 

    Filename: 'd:\matlab\records.dat'
    Writable: false
      Offset: 0
      Format: 'uint8'
      Repeat: Inf
        Data: 10x1 uint8 array
```

MATLAB maps the entire `records.dat` file to memory, setting all properties of the memory map to their default values. The memory map is assigned to the variable, `m`. In this example, the command maps the entire file as a sequence of unsigned 8-bit integers and gives the caller read-only access to its contents.

View the mapped data by accessing the `Data` property of `m`.

```
m.Data
```

```
ans = 

    1
    2
    3
    4
    5
    6
    7
    8
    9
   10
```

Map Entire File of Double-Precision Data

Create a memory map for double-precision data. The syntax is similar when specifying other data types.

At the command prompt, create a sample file in your current folder called `records.dat`, containing 10 `double` values.

```
myData = (1:10)';

fileID = fopen('records.dat','w');
fwrite(fileID,myData,'double');
fclose(fileID);
```

Create a memory map for `records.dat`, and set the `Format` property for the output to `'double'`.

```
m = memmapfile('records.dat','Format','double') ;
```

The memmapfile, `m`, contains the following properties: `Filename`, `Writable`, `Offset`, `Format`, `Repeat`, and `Data`. To display any one property, for example `Format`, type `m.Format` in the command window.

```
m.Format
```

```
ans =
'double'
```

The `Data` property contains the 10 double-precision values in `records.dat`.

Map and Change Part of a File

Create a memory map for a large array of `int32` data. Specify write access, and nondefault `Format` and `Offset` values.

At the command prompt, create a sample file in your current folder called `records.dat`, containing 10,000 `int32` values.

```
myData = int32([1:10000]);

fileID = fopen('records.dat','w');
fwrite(fileID,myData,'int32');
fclose(fileID);
```

Create a memory map for `records.dat`, and set the `Format` property for the output to `int32`. Also, set the `Offset` property to disregard the first 9000 bytes in the file, and the `Writable` property to permit write access.

```
m = memmapfile('records.dat',...
               'Offset',9000,...
               'Format','int32',...
               'Writable',true);
```

An `Offset` value of `9000` indicates that the first 9000 bytes of `records.dat` are not mapped.

Type the name of the memory map to see the current settings for all properties.

```
m
```

```
m = 
    Filename: 'd:\matlab\records.dat'
    Writable: true
      Offset: 9000
      Format: 'int32'
      Repeat: Inf
        Data: 7750x1 int32 array
```

The `Format` property indicates that any read or write operation made via the memory map reads and writes the file contents as a sequence of signed 32-bit integers. The `Data` property contains only 7750 elements because the first 9000 bytes of `records.dat`, representing the first 2250 values in the file, are not mapped.

View the first five elements of the mapped data by accessing the `Data` property of `m`.

```
m.Data(1:5)
```

```
ans = 

        2251
        2252
        2253
        2254
        2255
```

Map Region of File to Specific Array Shape

Create a memory map for a region of a file containing 100 double-precision values.

At the command prompt, create a sample file in your current folder called `mybinary.bin`, containing 100 double-precision values.

```
randData = gallery('uniformdata',[100,1],0,'double');

fileID = fopen('mybinary.bin','w');
fwrite(fileID,randData,'double');
fclose(fileID);
```

Map the first 75 values in `mybinary.bin` to a 5-by-5-by-3 array of double-precision values that can be referenced in the structure of the memory map using the field name `x`. Specify these parameters with the `Format` name-value pair argument.

```
m = memmapfile('mybinary.bin',...
               'Format',{'double',[5 5 3],'x'})
```

```
m = 

    Filename: 'd:\matlab\mybinary.bin'
    Writable: false
      Offset: 0
      Format: {'double' [5 5 3] 'x'}
      Repeat: Inf
        Data: 1x1 struct array with fields:
                x
```

The `Data` property is a structure array that contains the mapped values in the field, `x`.

Assign the mapped data to a variable, `A`. Because the `Data` property is a structure array, you must index into the field, `x`, to access the data.

```
A = m.Data.x;
```

View information about `A`.

```
whos A
```

```
  Name      Size             Bytes  Class     Attributes

  A         5x5x3              600  double
```

Map Segments of File to Multiple Arrays

Map segments of a file with different array shapes and data types to memory.

At the command prompt, create a sample file in your current folder called `mybinary.bin`. Write `uint16` data and double-precision data representing sample

pressure, temperature, and volume values into the file. In this case, each of the `uint16` arrays are 50-by-1 and the double-precision arrays are 5-by-10. `k` is a sample scaling factor.

```
k = 8.21;
[pres1,temp1] = gallery('integerdata',[1,300],[50,1],0,'uint16');
vol1 = double(reshape(k*temp1./pres1,5,10));
[pres2,temp2] = gallery('integerdata',[5,500],[50,1],5,'uint16');
vol2 = double(reshape(k*temp2./pres2,5,10));

fileID = fopen('mybinary.bin','w');
fwrite(fileID,pres1,'uint16');
fwrite(fileID,temp1,'uint16');
fwrite(fileID,vol1,'double');
fwrite(fileID,pres2,'uint16');
fwrite(fileID,temp2,'uint16');
fwrite(fileID,vol2,'double');
fclose(fileID);
```

Map the file to arrays accessible by unique names. Define a field, `pressure`, containing a 50-by-1 array of `uint16` values, followed by a field, `temperature`, containing 50-by-1 `uint16` values. Define a field, `volume`, containing a 5-by-10 array of double-precision values. Use a cell array to define the format of the mapped region and repeat the pattern twice.

```
m = memmapfile('mybinary.bin',...
'Format',{'uint16',[50 1],'pressure';...
'uint16',[50,1],'temperature';...
'double',[5,10],'volume'},'Repeat',2)
```

```
m =

    Filename: 'd:\matlab\mybinary.bin'
    Writable: false
      Offset: 0
      Format: {'uint16' [50 1] 'pressure'
               'uint16' [50 1] 'temperature'
               'double' [5 10] 'volume'}
      Repeat: 2
        Data: 2x1 struct array with fields:
         pressure
      temperature
           volume
```

The `Data` property of the memory map, `m`, is a 2-by-1 structure array because the `Format` is applied twice.

Copy the `Data` property to a variable, `A`. Then, view the last block of `double` data, which you can access using the field name, `volume`.

```
A = m.Data;
myVolume = A(2).volume
```

```
myVolume =

     2    13    32     5     5    16     4    22     3     8
     2     9    53    38    13    19    23    85     2   120
    29    10     6     1     2     5     6    58    20    11
     7    15     4     1     5    18     1     4    14     8
     9     8     4     2     0     9     8     6     3     3
```

Input Arguments

`filename` — Name of file to map
character vector | string scalar

Name of the file to map including the file extension, specified as a character vector or string scalar. The `filename` argument cannot include any wildcard characters (for example, `*` or `?`).

Example: `'myFile.dat'`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `m = memmapfile('myFile.dat','Format','int32','Offset',255)` maps `int32` data in the file, `myFile.dat`, to memory starting from the 256th byte.

`Writable` — Type of access allowed to mapped region
`false` (default) | `true`

Type of access allowed to the mapped region, specified as the comma-separated pair consisting of `'Writable'` and either `true` or `false`. If the `Writable` property is set to `false`, the mapped region is read-only. If `true`, then write access is allowed.

Example: `'Writable',true`

Data Types: `logical`

`Offset` — Number of bytes from start of file to start of mapped region
0 (default) | nonnegative integer

Number of bytes from the start of the file to the start of the mapped region, specified as the comma-separated pair consisting of `'Offset'` and a nonnegative integer. This value is zero-based. That is, an `Offset` value of 0 represents the start of the file.

Example: `'Offset',1024`

Data Types: `double`

`Format` — Format of mapped region
`'uint8'` (default) | character vector | string scalar | `n`-by-3 cell array

Format of the mapped region contents, specified as the comma-separated pair consisting of `'Format'` and a single character vector or string scalar, or an `n`-by-3 cell array.

- If the file region you are mapping contains data of only one type, specify the `Format` value as a character vector or string scalar identifying that type.
  Example: `'int16'`
- To specify an array shape to apply to the data read or written to the mapped file, and a field name to reference this array, specify the `Format` value as a 1-by-3 cell array. The first cell contains a character vector or string scalar identifying the data type to apply to the mapped region. The second cell contains the array dimensions to apply to the region. The third cell specifies the field name to use in the `Data` structure array of the memory map. Specify the field name as a string scalar or a character vector.
  Example: `{'uint64',[30 4 10],'x'}`
- If the region you are mapping is composed of segments of varying data types or array shapes, you can specify an individual format for each segment using an `n`-by-3 cell array, where `n` is the number of segments.
  Example: `{'uint64',[30 4 10],'x'; 'uint32',[30 4 6],'y'}`

You can use any of the following data types when you specify a `Format` value:

- 'int8'
- 'int16'
- 'int32'
- 'int64'
- 'uint8'
- 'uint16'
- 'uint32'
- 'uint64'
- 'single'
- 'double'

Data Types: char | string | cell

Repeat — Number of times to apply Format parameter
Inf (default) | positive integer

Number of times to apply the Format parameter to the mapped region of the file, specified as the comma-separated pair consisting of 'Repeat' and a positive integer. If the value of Repeat is Inf, then memmapfile applies the Format parameter until the end of the file.

Example: 'Repeat',2000

Data Types: double

Output Arguments

m — Memory map
memmapfile object

Memory map, returned as a memmapfile object with the following properties.

Property	Description
Filename	Path and name of the mapped file
Writable	Type of access allowed to the mapped region

Property	Description
`Offset`	Number of bytes from the start of the file to the start of the mapped region
`Format`	Format of the contents of the mapped region, including data type, array size, and field name by which to access the data
`Repeat`	Number of times to apply the pattern specified by the `Format` property to the mapped region of the file
`Data`	Memory-mapped data from the file. `Data` can be a numeric array or a structure array with field names specified in the `Format` property

The values for any property (except for `Data`) are set at the time you call `memmapfile`, using name-value pair arguments.

Access any property of `m` with dot notation similar to accessing fields of a structure array. For example, to access the memory-mapped data in the `Data` property, do one of the following:

- If `Data` is a numeric array, call `m.Data`.
- If `Data` is a scalar (1-by-1) structure array, call `m.Data.`*`fieldname`*, where *`fieldname`* is the name of a field.
- If `Data` is a nonscalar structure array, call `m.Data(`*`index`*`).`*`fieldname`* where *`index`* is the index for the element in the structure array, and *`fieldname`* is the name of a field. For example, to access the file data in the `temperature` field of the first element of `Data`, call `m.Data(1).temperature`.

After you create a memory map, `m`, you can change the value of any of its properties, except for `Data`. To assign a new value, use dot notation. For example, to set a new `Offset` value for `m`, type:

```
m.Offset = 2048;
```

Tips

- You can map only an existing file. You cannot create a new file and map that file to memory in one operation. Use the MATLAB file I/O functions to create the file before attempting to map it to memory.

- After `memmapfile` locates the file, MATLAB stores the file's absolute pathname internally, and then uses this stored path to locate the file from that point on. As a result, you can work in other directories outside your current work directory and retain access to the mapped file.
- `memmapfile` does not expand or append to a mapped file. Use instead standard file I/O functions like `fopen` and `fwrite`.

Algorithms

The actual mapping of a file to the MATLAB address space does not take place when you construct a `memmapfile` object. A memory map, based on the information currently stored in the mapped object, is generated the first time you reference or modify the `Data` property for that object.

See Also

Topics

"Map File to Memory"
"Overview of Memory-Mapping"

Introduced before R2006a

memoize

Add memoization semantics to function handle

Syntax

```
memoizedFcn = memoize(fh)
```

Description

Memoization is an optimization technique used to speed up programs by caching the results of expensive function calls and returning the cached result when the program is called with the same inputs.

Consider memoizing a function call if all of the following are true:

- Performance is important.
- The function is time consuming.
- The function has return values that are determined entirely by the input values, and has no side effects.
- System memory is adequate to store unique input and output combinations.

`memoizedFcn = memoize(fh)` adds memoization semantics to the input function handle, and returns a `MemoizedFunction` object. Invoke `memoizedFcn` as you would invoke `fh`. However, `memoizedFcn` is not a function handle.

The `MemoizedFunction` object maintains the cache of inputs and the corresponding outputs. When it is invoked, MATLAB returns the associated cached output values if the following conditions are true.

1. The input arguments are numerically equal to cached inputs. When comparing input values, MATLAB treats `NaN`s as equal.
2. The number of requested output arguments matches the number of cached outputs associated with the inputs.

The memoization of a function is associated with the input function and not with the `MemoizedFunction` object. Therefore, keep the following in mind.

- Constructing a new `MemoizedFunction` object to the same function creates another reference to the same data. Two variables that memoize the same function share a cache and object property values, such as cache size. In the following example, the variables `a` and `b` share a cache and have the same value for cache size.

  ```
  a = memoize(@svd);
  b = memoize(@svd);
  ```

  Similarly, clearing the cache for `b` (`b.clearCache`) also clears the cache for `a`, and any other variables that memoize the `svd` function. `clearCache` is a `MemoizedFunction` object function.
- Assigning a `MemoizedFunction` object to a new variable creates another reference to the same data. In the following example, the variables `c` and `d` share data.

  ```
  c = memoize(@svd);
  d = c;
  ```
- Clearing a variable does not clear the cache associated with the input function. To clear the cache for a `MemoizedFunction` object that no longer exists in the workspace, create a new `MemoizedFunction` object to the same function, and use the `clearCache` function on the new object. Alternatively, you can clear caches for all `MemoizedFunction` objects using the `clearAllMemoizedCaches` function.

Caution A `MemoizedFunction` object is not aware of updates to the underlying function. If you modify the function associated with the memoized function, clear the cache with the `clearCache` object function.

Examples

Cache Results from MATLAB Built-in Function

To speed up performing a singular value decomposition when you could be operating on the same inputs multiple times, memoize the `svd` function.

```
fh = @svd;
memoizedFcn = memoize(fh);
```

Create a matrix and cache the results of the singular value decomposition. Time the function call.

```
X = magic(1234);
tic
[U,S,V]= memoizedFcn(X);
preCachedTime = toc
```

```
preCachedTime = 0.7452
```

Call the memoized function again using the same inputs. To observe the speed improvement using cached results, time the function call again.

```
tic
[U,S,V]= memoizedFcn(X);
postCachedTime = toc
```

```
postCachedTime = 0.0081
```

Cache Results from User-Defined Function

In your current working folder, create a file `computeNumberCombinations.m` that contains the following function to compute the number of combinations of `n` items taken `k` at a time.

```
type computeNumberCombinations.m
```

```
function c = computeNumberCombinations(n,k)
% Calculate number of combinations of n items taken k at a time
c = fact(n)/(fact(n-k)*fact(k));
end

function f = fact(n)
f = 1;
for m = 2:n
    f = f*m;
end
end
```

Clear the cache for any `MemoizedFunction` objects in your workspace.

```
clearAllMemoizedCaches
```

Memoize the `computeNumberCombinations` function to speed up computation for repeated input values.

```
fh = @computeNumberCombinations;
memoizedFcn = memoize(fh);
```

Call the memoized function and time the function call. This function call caches the results for the specified inputs.

```
tic
c = memoizedFcn(42e5,137);
preCachedTime = toc
```

```
preCachedTime = 0.0457
```

Call the memoized function and time the function call again. This function call uses the cached results and does not execute the function.

```
tic
c = memoizedFcn(42e5,137);
postCachedTime = toc
```

```
postCachedTime = 0.0100
```

Input Arguments

`fh` — Function to memoize
function handle

Function to memoize, specified as a function handle.

Example: `memoizedEigs = memoize(@eigs)`

Data Types: `function_handle`

Tips

- Multiple calls to `memoize` with the same function handle return the same `MemoizedFunction` object. For example:

```
x = memoize(@plus);
y = memoize(@plus);
x == y

ans =

  logical

   1
```

- You should not memoize a function with side effects such as setting some global state or performing I/O operations. Side effects are not repeated on subsequent calls to the memoized function with the same inputs. For example, if you memoize the `randi` function, the memoized function always returns the same value when called with the same input argument.

```
fh = @randi;
memoized_fh = memoize(fh);

fh_result = [fh(100) fh(100) fh(100)]
memoized_result = [memoized_fh(100) memoized_fh(100) memoized_fh(100)]

fh_result =

    18    71     4


memoized_result =

    28    28    28
```

See Also

Functions
`clearAllMemoizedCaches`

Objects
`MemoizedFunction`

Introduced in R2017a

MemoizedFunction

Call memoized function and cache results

Description

A `MemoizedFunction` object maintains the memoization semantics of a function handle and a cache of the function call results. It has the same calling syntax as the function handle specified in the `Function` property. However, the `MemoizedFunction` object is not a function handle.

The first time you call the memoized function with a certain set of input values, MATLAB executes the function specified by the `Function` property and caches the results. In later calls to the memoized function with the same set of inputs, MATLAB returns the cached results instead of executing the function again.

The `MemoizedFunction` object maintains the cache of inputs and the corresponding outputs. When it is invoked, MATLAB returns the associated cached output values if the following conditions are true.

1. The input arguments are numerically equal to cached inputs. When comparing input values, MATLAB treats `NaN`s as equal.
2. The number of requested output arguments matches the number of cached outputs associated with the inputs.

Caution A `MemoizedFunction` object is not aware of updates to the underlying function. If you modify the function associated with the memoized function, clear the cache with the `clearCache` object function.

Creation

To create a `MemoizedFunction` object, call the `memoize` function.

The memoization of a function is associated with the input function and not with the `MemoizedFunction` object. Therefore, keep the following in mind.

- Constructing a new `MemoizedFunction` object to the same function creates another reference to the same data. Two variables that memoize the same function share a cache and object property values, such as cache size. In the following example, the variables `a` and `b` share a cache and have the same value for cache size.

  ```
  a = memoize(@svd);
  b = memoize(@svd);
  ```

  Similarly, clearing the cache for `b` (`b.clearCache`) also clears the cache for `a`, and any other variables that memoize the `svd` function. `clearCache` is a `MemoizedFunction` object function.
- Assigning a `MemoizedFunction` object to a new variable creates another reference to the same data. In the following example, the variables `c` and `d` share data.

  ```
  c = memoize(@svd);
  d = c;
  ```
- Clearing a variable does not clear the cache associated with the input function. To clear the cache for a `MemoizedFunction` object that no longer exists in the workspace, create a new `MemoizedFunction` object to the same function, and use the `clearCache` function on the new object. Alternatively, you can clear caches for all `MemoizedFunction` objects using the `clearAllMemoizedCaches` function.

Properties

`MemoizedFunction` properties control the behavior of the memoized function. You can access or modify properties of the memoized function. Use dot notation to refer to a particular object and property:

```
m = memoize(@ones);
m.CacheSize = 25;
```

`Function` — Function with memoization semantics applied
function handle

Function with memoization semantics applied, returned as a function handle. This property is read only.

Data Types: `function_handle`

`CacheSize` — Maximum number of cached input and output combinations
10 (default) | positive integer

Maximum number of cached input and output combinations, specified as a positive integer.

Data Types: `double`

Enabled — Caching state
`true` (default) | `false`

Caching state, specified as `true` or `false`. To instruct MATLAB to call the function specified by the `Function` property regardless of whether the results are cached, and not to cache results, set this property to `false`.

Data Types: `logical`

Object Functions

clearCache	Clear cache for MemoizedFunction object
stats	Return cached values and statistics for MemoizedFunction object

Examples

Change Cache Size of Memoized Function

Create a `MemoizedFunction` object by memozing the `datetime` function.

```
mf = memoize(@datetime)
```

```
mf = 
  MemoizedFunction with properties:

     Function: @datetime
      Enabled: 1
    CacheSize: 10
```

Change the maximum number of cached input and output combinations.

```
mf.CacheSize = 2
```

```
mf = 
  MemoizedFunction with properties:
```

```
     Function: @datetime
      Enabled: 1
    CacheSize: 2
```

Call the memoized function with three different input values.

```
a = mf('today');
b = mf('yesterday');
c = mf('tomorrow');
```

Call the `stats` function to investigate the cached results.

```
s = stats(mf);
s.Cache.Inputs{:}
```

```
ans = 1x1 cell array
    {'yesterday'}


ans = 1x1 cell array
    {'tomorrow'}
```

The results of calling the memoized function with `'today'` are not cached because the `CacheSize` is 2.

Disable Memoization

In your current working folder, create the following file `memoizeSquareExample.m` that contains a function to compute the square of a number. When the function is called, if MATLAB returns cached results, `msg` is not displayed.

```
type memoizeSquareExample.m
```

```
function m = memoizeSquareExample(n)

m = n^2;
msg = "The square of " + string(n) + " is " + string(m) +".";
disp(msg)

end
```

Memoize the function. By default, memoization is enabled.

```
mf = memoize(@memoizeSquareExample);
```

Call the memoized function twice with the same input value. `msg` is displayed only once because the second function call returns cached results.

```
a = mf(42);
```

```
The square of 42 is 1764.
```

```
b = mf(42);
```

Disable memoization and call the memoized function with a repeat input value. Although the results for an input of 42 are cached, `msg` is displayed because memoization is disabled.

```
mf.Enabled = false;
c = mf(42);
```

```
The square of 42 is 1764.
```

Call the memoized function with a different set of inputs.

```
d = mf(13);
```

```
The square of 13 is 169.
```

Call the `stats` function to investigate the cached results. MATLAB does not return cached results while memoization is disabled or collect statistics, it continues to store input and output values.

```
s = mf.stats();
s.Cache.Inputs{:}
```

```
ans = 1x1 cell array
    {[42]}
```

See Also

`clearAllMemoizedCaches` | `memoize`

Introduced in R2017a

memory

Display memory information

Syntax

```
memory
userview = memory
[userview,systemview] = memory
```

Limitations

- The `memory` function is available only on Microsoft Windows systems. Results are dependent on your computer hardware and the load on your computer.

Description

`memory` displays information showing how much memory is available and how much the MATLAB software is currently using. The information displayed at your computer screen includes the following items, each of which is described in a section below:

- “Maximum Possible Array” on page 1-9605
- “Memory Available for All Arrays” on page 1-9606
- “Memory Used By MATLAB” on page 1-9606
- “Physical Memory (RAM)” on page 1-9607

`userview = memory` returns user-focused information on memory use in structure `userview`. The information returned in `userview` includes the following items, each of which is described in a section below:

- “Maximum Possible Array” on page 1-9605
- “Memory Available for All Arrays” on page 1-9606
- “Memory Used By MATLAB” on page 1-9606

`[userview,systemview] = memory` returns both user- and system-focused information on memory use in structures `userview` and `systemview`, respectively. The `userview` structure is described in the command syntax above. The information returned in `systemview` includes the following items, each of which is described in a section below:

- “Virtual Address Space” on page 1-9607
- “System Memory” on page 1-9608
- “Physical Memory (RAM)” on page 1-9607

Output Arguments

Each of the sections below describes a value that is displayed or returned by the `memory` function.

Maximum Possible Array

Maximum Possible Array is the size of the largest contiguous free memory block. As such, it is an upper bound on the largest single array MATLAB can create at this time.

MATLAB derives this number from the smaller of the following two values:

- The largest contiguous memory block found in the MATLAB virtual address space
- The total available system memory

To see how many array elements this number represents, divide by the number of bytes in the array class. For example, for a `double` array, divide by 8. The actual number of elements MATLAB can create is always fewer than this number.

When you enter the `memory` command without assigning its output, MATLAB displays this information in the Command Window. When you do assign the output, MATLAB returns the information in a structure field. See the table below.

Command	Returned in
`memory`	Character vector labeled `Maximum possible array:`
`user = memory`	Structure field `user.MaxPossibleArrayBytes`

All values are double-precision and in units of bytes.

Footnotes

When you enter the `memory` command without specifying any outputs, MATLAB may also display one of the following footnote on 64-bit systems:

`Limited by System Memory (physical + swap file) available.`

There is insufficient system memory to allow mapping of all virtual addresses in the largest available block of the MATLAB process.

Memory Available for All Arrays

Memory Available for All Arrays is the total amount of memory available to hold data. The amount of memory available is guaranteed to be at least as large as this field.

MATLAB derives this number from the smaller of the following two values:

- The total available MATLAB virtual address space
- The total available system memory

When you enter the `memory` command without assigning its output, MATLAB displays this information in the Command Window. When you do assign the output, MATLAB returns the information in a structure field. See the table below.

Command	Returned in
`memory`	Character vector labeled `Memory available for all arrays:`
`user = memory`	Structure field `user.MemAvailableAllArrays`

Footnotes

When you enter the `memory` command without specifying any outputs, MATLAB may also display the following footnote on 64-bit systems:

`Limited by System Memory (physical + swap file) available.`

There is insufficient system memory to allow mapping of all available virtual addresses in the MATLAB process.

Memory Used By MATLAB

Memory Used By MATLAB is the total amount of system memory reserved for the MATLAB process. It is the sum of the physical memory and potential swap file usage.

When you enter the `memory` command without assigning its output, MATLAB displays this information in the Command Window. When you do assign the output, MATLAB returns the information in a structure field. See the table below.

Command	Returned in
`memory`	Character vector labeled `Memory used by MATLAB:`
`user = memory`	Structure field `user.MemUsedMATLAB`

Physical Memory (RAM)

Physical Memory is the available and total amounts of physical memory (RAM) on the computer running MATLAB.

When you enter the `memory` command without assigning its output, MATLAB displays the total memory in the Command Window. When you do assign the output, MATLAB returns both the available and total memory in a structure field. See the table below.

Command	Value	Returned in
`memory`	Total memory	Character vector labeled `Physical Memory (RAM):`
`[user,sys] = memory`	Available memory	Structure field `sys.PhysicalMemory.Available`
	Total memory	Structure field `sys.PhysicalMemory.Total`

Available physical memory is the same as **Available** found in the Windows Task Manager: Performance/Physical Memory, and the total physical memory is the same as **Total**.

You can use the amount of available physical memory as a measure of how much data you can access quickly.

Virtual Address Space

Virtual Address Space is the amount of available and total virtual memory for the MATLAB process. MATLAB returns the information in two fields of the return structure: `Available` and `Total`.

Command	Return Value	Returned in Structure Field
`[user,sys] = memory`	Available memory	`sys.VirtualAddressSpace.Available`
	Total memory	`sys.VirtualAddressSpace.Total`

You can monitor the difference:

```
VirtualAddressSpace.Total - VirtualAddressSpace.Available
```

as the Virtual Bytes counter in the Windows Performance Monitor (e.g., Windows 7 Control Panel/Administrative Tools/Performance Monitor). If you add a counter, the **Virtual Bytes** counter is found under the **Process** menu. Select the counter, and then select MATLAB from the **Instances of selected object** pane.

System Memory

System Memory is the amount of available system memory on your computer system. This number includes the amount of available physical memory and the amount of available swap file space on the computer running MATLAB. MATLAB returns the information in the `SystemMemory` field of the return structure.

Command	Return Value	Returned in Structure Field
`[user,sys] = memory`	Available memory	`sys.SystemMemory`

This is the same as the difference:

```
limit - total (in bytes)
```

found in the Windows Task Manager: Performance/Commit Charge.

Examples

Display memory statistics on a 64-bit Windows system:

```
memory
```

```
Maximum possible array:            14253 MB (1.495e+10 bytes) *
Memory available for all arrays:           14253 MB (1.495e+10 bytes) *
Memory used by MATLAB:              747 MB (7.833e+08 bytes)
Physical Memory (RAM):            12279 MB (1.288e+10 bytes)

*  Limited by System Memory (physical + swap file) available.
```

Return in the structure `userview`, information on the largest array MATLAB can create at this time, how much memory is available to hold data, and the amount of memory currently being used by your MATLAB process:

```
userview = memory
```

```
userview =

    MaxPossibleArrayBytes: 1.4957e+10
    MemAvailableAllArrays: 1.4957e+10
            MemUsedMATLAB: 784044032
```

Assign the output to two structures, `user` and `sys`, to obtain the information shown here:

```
[user,sys] = memory;

%  --- Largest array MATLAB can create ---
user.MaxPossibleArrayBytes

ans =
   1.4956e+10

%  --- Memory available for data ---
user.MemAvailableAllArrays

ans =
   1.4956e+10

%  --- Memory used by MATLAB process ---
user.MemUsedMATLAB

ans =
   784039936

%  --- Virtual memory for MATLAB process ---
sys.VirtualAddressSpace

ans =
    Available: 8.7910e+12
        Total: 8.7961e+12

%  --- Physical memory and paging file ---
sys.SystemMemory

ans =
    Available: 1.4956e+10
```

```
%   --- Computer's physical memory ---
sys.PhysicalMemory

ans =
    Available: 2.7093e+09
        Total: 1.2876e+10
```

Tips

Details on Memory Used By MATLAB

MATLAB computes the value for Memory Used By MATLAB by walking the MATLAB process memory structures and summing all the sections that have physical storage allocated in memory or in the paging file on disk.

Using the Windows Task Manager, you have for the MATLAB.exe image:

`Mem Usage < MemUsedMATLAB < Mem Usage + VM Size (in bytes)`

where both of the following are true:

- `Mem Usage` is the working set size in kilobytes.
- `VM Size` is the page file usage, or private bytes, in kilobytes.

The working set size is the portion of the MATLAB virtual address space that is *currently* resident in RAM and can be referenced without a memory page fault. The page file usage gives the portion of the MATLAB virtual address space that requires a backup that doesn't already exist. Another name for page file usage is private bytes. It includes all MATLAB variables and workspaces. Since some of the pages in the page file may also be part of the working set, this sum is an overestimate of `MemUseMATLAB`. Note that there are virtual pages in the MATLAB process space that already have a backup. For example, code loaded from EXEs and DLLs and memory-mapped files. If any part of those files is in memory when the memory builtin is called, that memory will be counted as part of `MemUsedMATLAB`.

Reserved Addresses

Reserved addresses are addresses sets aside in the process virtual address space for some specific future use. These reserved addresses reduce the size of

`MemAvailableAllArrays` and can reduce the size of the current or future value of `MaxPossibleArrayBytes`.

Example 1 — Java Virtual Machine (JVM)

At MATLAB startup, part of the MATLAB virtual address space is reserved by the Java Virtual Machine (JVM) and cannot be used for storing MATLAB arrays.

Example 2 — Standard Windows Heap Manager

MATLAB, by default, uses the standard Windows heap manager except for a set of small preselected allocation sizes. One characteristic of this heap manager is that its behavior depends upon whether the requested allocation is less than or greater than the fixed number of 524,280 bytes. For, example, if you create a sequence of MATLAB arrays, each less than 524,280 bytes, and then clear them all, the `MemUsedMATLAB` value before and after shows little change, and the `MemAvailableAllArrays` value is now smaller by the total space allocated.

The result is that, instead of globally freeing the extra memory, the memory becomes reserved. It can *only* be reused for arrays less than 524,280 bytes. You cannot reclaim this memory for a larger array except by restarting MATLAB.

See Also

`clear` | `inmem` | `load` | `mlock` | `munlock` | `pack` | `save` | `whos`

Topics

"How MATLAB Allocates Memory"
"Strategies for Efficient Use of Memory"
"Resolve "Out of Memory" Errors"

Introduced in R2008a

menu

(Not recommended) Create multiple-choice dialog box

Note The `menu` function is not recommended. Use the `listdlg` function instead.

Syntax

```
choice = menu(message,opt1,opt2,...,optn)
choice = menu(message,options)
```

Description

`choice = menu(message,opt1,opt2,...,optn)` displays a modal menu dialog box containing the text in `message` and the choices specified by `opt1`, `opt2`,... `optn`. The `menu` function returns the number of the selected menu item, or 0 if the user clicks the close button on the window. Specify `message` as a character vector or string scalar. Specify `opt1`, `opt2`,... `optn` as character vectors or string scalars.

`choice = menu(message,options)` specifies the choices as a cell array of character vectors or string array.

If the user's terminal provides a graphics capability, `menu` displays the menu items as push buttons in a figure window (Example 1). Otherwise. they will be given as a numbered list in the Command Window (Example 2).

Examples

Example 1

On a system with a display, `menu` displays choices as buttons in a dialog box:

```
choice = menu('Choose a color','Red','Blue','Green')
```

displays the following dialog box.

The number entered by the user in response to the prompt is returned as `choice` (i.e., `choice = 2` implies that the user selected `Blue`).

After input is accepted, the dialog box closes, returning the output in `choice`. You can use `choice` to control the color of a graph:

```
t = 0:.1:60;
s = sin(t);
color = ['r','b','g']
plot(t,s,color(choice))
```

Example 2

On a system without a display, `menu` displays choices in the Command Window:

```
choice = menu('Choose a color','Red','Blue','Green')
```

displays the following text.

```
----- Choose a color -----
1) Red
2) Blue
3) Green
Select a menu number:
```

Tips

To call `menu` from within a callback of a `UIControl` or other UI component, set that object's `Interruptible` property to `'on'`. For more information, see Uicontrol.

See Also

`dialog` | `guide` | `input` | `uicontrol` | `uimenu`

Introduced before R2006a

Menu Properties

Control appearance and behavior of menu

Description

Menus display drop-down lists of options at the top of an app window. Call the `uimenu` function to create a menu or add a submenu to an existing menu. Properties control the appearance and behavior of a menu. Use dot notation to refer to a specific object and property.

```
uf = uifigure;
m = uimenu(uf);
m.Text = 'Open Selection';
```

The properties listed here are valid for menus in App Designer, or in apps created with the `uifigure` function. For menus used in GUIDE, or in apps created with the `figure` function, see Uimenu.

Properties

Menu

`Text` — Menu label

character vector | string scalar

Menu label, specified as a character vector or string scalar. This property specifies the label that appears on the menu or menu item.

Avoid using these case-sensitive reserved words: "default", "remove", and "factory". If you must use a reserved word, then specify a backslash character before the word. For instance, specify "default" as `'\default'`.

You can specify a mnemonic keyboard shortcut (**Alt+***`mnemonic`*) by using the ampersand (&) character in the text for the label. The character that follows the ampersand appears underlined in the menu when **Alt** is pressed. You can select the menu item by holding down the **Alt** key and typing the character shown.

To use mnemonics, you must specify a mnemonic for all menus and menu items that you define in the app. If you define mnemonics only for some menus or menu items, pressing the **Alt** key does not have any effect.

The table shows some examples:

Text Value	Menu Label with Mnemonic Hints
`'&Open Selection'`	Open Selection
`'O&pen Selection'`	Open Selection
`'&Save && Go'`	Save & Go

`Accelerator` — Keyboard shortcut
character

Keyboard shortcut, specified as a character. Use this property to define a keyboard shortcut for selecting a menu item.

Example: `mitem.Accelerator = 'H'`

Specifying an accelerator value enables users to select the menu item by pressing a character and another key, instead of using the mouse. The key sequence is platform specific.

- Windows systems: **Ctrl+***accelerator*
- Macintosh systems: **Command+***accelerator*
- Linux systems: **Ctrl+***accelerator*

Things to keep in mind when using accelerators:

- The app window must be in focus when entering the accelerator key sequence.
- Accelerators cannot be used on top-level menus.
- Accelerators only work when the menu item meets all these criteria.
  - It does not contain any submenu items.
  - It executes a callback function.
  - It has the `Visible` property set to `'on'`.

`Separator` — Separator line mode

`'off'` (default) | `'on'`

Separator line mode, specified as `'off'` or `'on'`. Setting this property to `'on'` draws a dividing line above the menu item.

Note The `Separator` property is ignored when the menu item is a top-level menu item.

`Checked` — Menu check indicator

`'off'` (default) | `'on'`

Menu check indicator, specified as `'off'` or `'on'`. Setting this property to `'on'` places a check mark next to the corresponding menu item. Setting it to `'off'` removes the check mark. You can use this feature to show the state of menu items that enable or disable functionality in your application.

Note The `Checked` property is ignored when the menu item is:

- A top-level menu item
- A menu item that contains one or more child menu items

`ForegroundColor` — Menu label color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Menu label color, specified as an RGB triplet, a hexadecimal color code or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Interactivity

`Visible` — Menu visibility

`'on'` (default) | `'off'`

Menu visibility, specified as `'on'` or `'off'`. When the `Visible` property is set to `'off'`, the menu is not visible, but you can query and set its properties.

`Enable` — Operational state of menu

`'on'` (default) | `'off'`

Operational state of menu, specified as `'on'` or `'off'`. This property controls whether the user can select a menu item. When the value is `'off'`, the menu label appears dimmed, indicating that the user cannot select it.

`Tooltip` — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Callbacks

`MenuSelectedFcn` — Menu selected callback

`''` (default) | function handle | cell array | character vector

Menu selected callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

The callback responds depending on the location of the menu item and the type of interaction:

- Left-clicking a menu expands that menu and triggers its callback.
- While any menu is expanded, hovering any other parent menu (or top-level menu) expands that menu and triggers its callback.

Note Do not use a callback to dynamically change menu items. Deleting, adding, and replacing menu items in a callback can result in a blank menu. Instead, use the `Visible` property to hide or show menu items. You can also enable and disable menu items by setting the `Enable` property. To fully repopulate menu items, delete and create them outside the callback.

For more information about writing callbacks, see “Write Callbacks in App Designer”.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent object

`Figure` | `Menu`

Parent object, specified as a `Figure` created using the `uifigure` function, or a `Menu` object. You can move a menu item to a different window, or move it under a different menu by setting this property.

`Children` — Menu children

empty `GraphicsPlaceholder` array (default) | 1-D array of `Menu` objects

Menu children, returned as an empty `GraphicsPlaceholder` or a 1-D array of `Menu` objects.

You cannot add or remove child components using the `Children` property. Use this property to view the list of children or to reorder the child menu items.

To add a child menu to this list, set the `Parent` property of another `Menu` object to this `Menu` object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uimenu'`

This property is read-only.

Type of graphics object, returned as `'uimenu'`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`appdesigner` | `uimenu`

Introduced in R2017b

mergecats

Merge categories in categorical array

Syntax

```
B = mergecats(A,oldcats)
B = mergecats(A,oldcats,newcat)
```

Description

`B = mergecats(A,oldcats)` merges two or more categories in `A` into the first category, `oldcats(1)`. Any values in `A` from `oldcats` become `oldcats(1)` in `B`.

`B = mergecats(A,oldcats,newcat)` merges `oldcats` into a single new category, `newcat`. Any values in `A` from `oldcats` become `newcat` in `B`.

Examples

Merge Two Categories into One

Create a categorical array containing various colors.

```
A = categorical({'red';'blue';'pink';'red';'blue';'red'})
```

```
A = 6x1 categorical array
     red
     blue
     pink
     red
     blue
     red
```

`A` is a 6-by-1 categorical array.

Display the categories of A.

```
categories(A)
```

```
ans = 3x1 cell array
    {'blue'}
    {'pink'}
    {'red' }
```

The three categories are in alphabetical order.

Merge the categories red and pink into the category red. Specify red first in oldcats to use it as the merged category.

```
oldcats = {'red','pink'};
B = mergecats(A,oldcats)
```

```
B = 6x1 categorical array
     red
     blue
     red
     red
     blue
     red
```

mergecats replaces the value pink from A(3) with red.

Display the categories of B.

```
categories(B)
```

```
ans = 2x1 cell array
    {'blue'}
    {'red' }
```

B has two categories instead of three.

Merge Alphabetically Listed Categories

Create a categorical array containing various items.

```
A = categorical({'shirt' 'pants'; 'shoes' 'shirt'; 'dress' 'belt'})
```

```
A = 3x2 categorical array
     shirt      pants
     shoes      shirt
     dress      belt
```

Display the categories of A.

```
categories(A)
```

```
ans = 5x1 cell array
    {'belt' }
    {'dress'}
    {'pants'}
    {'shirt'}
    {'shoes'}
```

The five categories are in alphabetical order.

Merge the categories `belt` and `shoes` into a new category called `other`.

```
B = mergecats(A,{'belt' 'shoes'},'other')
```

```
B = 3x2 categorical array
     shirt      pants
     other      shirt
     dress      other
```

The value `other` replaces all instances of `belt` and `shoes`.

Display the categories of B.

```
categories(B)
```

```
ans = 4x1 cell array
    {'other'}
    {'dress'}
    {'pants'}
    {'shirt'}
```

B has four categories and the order is no longer alphabetical. `other` appears in place of `belt`.

Merge Categories of Ordinal Categorical Array

Create an ordinal categorical array.

```
A = categorical([1 2 3 2 1],1:3,{'poor','fair','good'},'Ordinal',true)
```

```
A = 1x5 categorical array
     poor      fair      good      fair      poor
```

Display the categories of A.

```
categories(A)
```

```
ans = 3x1 cell array
    {'poor'}
    {'fair'}
    {'good'}
```

Since A is ordinal, the categories have the mathematical ordering `poor < fair < good`.

Consider all `fair` or `poor` values to be `bad`. Since A is ordinal, the categories to merge must be consecutive.

```
B = mergecats(A,{'fair' 'poor'},'bad')
```

```
B = 1x5 categorical array
     bad       bad       good      bad       bad
```

The value `bad` replaces all instances of `fair` and `poor`.

Display the categories of B.

```
categories(B)
```

```
ans = 2x1 cell array
    {'bad' }
```

```
{'good'}
```

B has two categories with the mathematical ordering: `bad < good`.

Input Arguments

`A` — Categorical array
vector | matrix | multidimensional array

Categorical array, specified as a vector, matrix, or multidimensional array.

`oldcats` — Categories to merge
cell array of character vectors | string array

Categories to merge, specified as a cell array of character vectors or a string array. If `A` is ordinal, then the categories to merge must be consecutive.

`newcat` — New category
`oldcats(1)` (default) | character vector | string scalar

New category, specified as a character vector or a string scalar.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`addcats` | `categories` | `iscategory` | `removecats` | `renamecats` | `reordercats` | `setcats`

Introduced in R2013b

mergevars

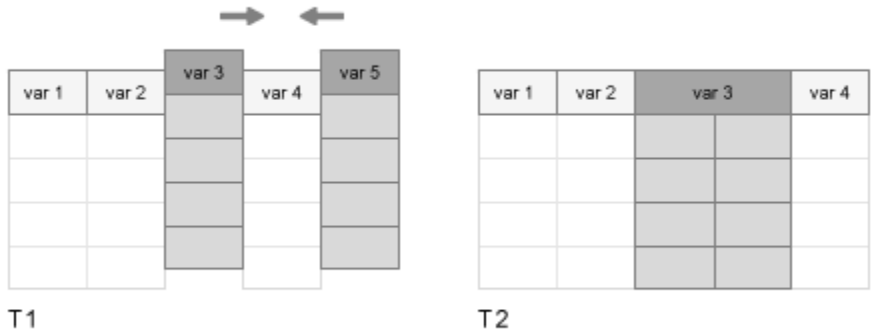
Combine table or timetable variables into multicolumn variable

Syntax

```
T2 = mergevars(T1,vars)
T2 = mergevars(T1,vars,'NewVariableName',newName)
T2 = mergevars( ___ ,'MergeAsTable',true)
```

Description

`T2 = mergevars(T1,vars)` combines the table variables specified by `vars` to create one multicolumn variable in `T2` (see diagram). All other variables from `T1` are unaltered. You can specify variables by name, by position, or using logical indices.



By default, the name of the merged variable in `T2` takes the form `Var`*N*, where *N* is the position of the merged variable. For example, if the merged variable is the third variable in `T2`, then its name is `Var3`.

To split multicolumn variables, use the `splitvars` function.

`T2 = mergevars(T1,vars,'NewVariableName',newName)` specifies a name for the multicolumn variable.

`T2 = mergevars( ___ ,'MergeAsTable',true)` merges the specified variables into a table, instead of an array. The new table is itself a variable of the output table `T2`. Use this syntax to combine variables that cannot be concatenated into an array. You can use this syntax with any of the input arguments from the previous syntaxes.

Examples

Merge Variables

Create a table from workspace variables.

```
A = [1:3]';
B = [5 11 12]';
C = [3.14 2.72 1.37]';
D = {'a';'b';'c'};
T1 = table(A,B,C,D)
```

```
T1=3×4 table
    A    B      C        D
    _    __    ____    _____

    1     5    3.14    {'a'}
    2    11    2.72    {'b'}
    3    12    1.37    {'c'}
```

Merge the second and third variables. The new variable has two columns.

```
T2 = mergevars(T1,[2 3])
```

```
T2=3×3 table
    A       Var2         D
    _    __________    _____

    1     5    3.14    {'a'}
    2    11    2.72    {'b'}
    3    12    1.37    {'c'}
```

New Name for Merged Variable

Create a table using arrays of data from the `patients.mat` file. Display the first three rows of the table.

```
load patients
T1 = table(LastName,Gender,Age,Height,Weight,Systolic,Diastolic);
head(T1,3)
```

```
ans=3×7 table
      LastName         Gender      Age    Height    Weight    Systolic    Diastolic
    ____________    __________    ___    ______    ______    ________    _________

    {'Smith'   }    {'Male'  }    38       71       176        124          93
    {'Johnson' }    {'Male'  }    43       69       163        109          77
    {'Williams'}    {'Female'}    38       64       131        125          83
```

Merge the variables `Systolic` and `Diastolic` into one variable with two columns. Name it `BloodPressure`.

```
T2 = mergevars(T1,{'Systolic','Diastolic'},'NewVariableName','BloodPressure');
head(T2,3)
```

```
ans=3×6 table
      LastName         Gender      Age    Height    Weight    BloodPressure
    ____________    __________    ___    ______    ______    _____________

    {'Smith'   }    {'Male'  }    38       71       176       124     93
    {'Johnson' }    {'Male'  }    43       69       163       109     77
    {'Williams'}    {'Female'}    38       64       131       125     83
```

Merged Variable as Table

Read in a table from a spreadsheet. Display the first three rows.

```
T1 = readtable('outages.csv');
head(T1,3)
```

```
ans =

  3x6 table

      Region           OutageTime        Loss     Customers     RestorationTime
    _____________    _________________    ______    __________    _________________    __

    {'SouthWest'}    2002-02-01 12:18    458.98    1.8202e+06    2002-02-07 16:50    {
    {'SouthEast'}    2003-01-23 00:49    530.14    2.1204e+05                NaT    {
    {'SouthEast'}    2003-02-07 21:15     289.4    1.4294e+05    2003-02-17 08:14    {
```

Merge `Cause`, `Loss`, and `RestorationTime`. Because these variables have different types, merge them into a table within a table.

```
T2 = mergevars(T1,{'Cause','Loss','RestorationTime'},...
               'NewVariableName','LossData','MergeAsTable',true);
head(T2,3)
```

```
ans =

  3x4 table

      Region           OutageTime       Customers                        LossData
                                                                 Cause     Loss     Re
    _____________    _________________    __________    ____________________________

    {'SouthWest'}    2002-02-01 12:18    1.8202e+06    {'winter storm'}    458.98    20
    {'SouthEast'}    2003-01-23 00:49    2.1204e+05    {'winter storm'}    530.14
    {'SouthEast'}    2003-02-07 21:15    1.4294e+05    {'winter storm'}     289.4    20
```

Input Arguments

T1 — Input table
table | timetable

Input table, specified as a table or timetable.

`vars` — Variables in input table
character vector | cell array of character vectors | string array | numeric array | logical array

Variables in the input table, specified as a character vector, cell array of character vectors, string array, numeric array, or logical array.

`newName` — Name of merged variable
character vector | string scalar

Name of the merged variable, specified as a character vector or string scalar.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

See Also

`addvars` | `movevars` | `removevars` | `splitvars`

Topics

“Add, Delete, and Rearrange Table Variables”
“Add and Delete Table Rows”
“Access Data in Tables”
“Modify Units, Descriptions, and Table Variable Names”
“Clean Messy and Missing Data in Tables”

Introduced in R2018a

mesh

Mesh surface plot

Syntax

```
mesh(X,Y,Z)
```

```
mesh(Z)
mesh(Z,C)
mesh( ___ ,C)
```

```
mesh(ax, ___ )
mesh( ___ ,Name,Value)
s = mesh( ___ )
```

Description

`mesh(X,Y,Z)` creates a mesh plot, which is a three-dimensional surface that has solid edge colors and no face colors. The function plots the values in matrix `Z` as heights above a grid in the *x*-*y* plane defined by `X` and `Y`. The edge colors vary according to the heights specified by `Z`.

`mesh(Z)` creates a mesh plot and uses the column and row indices of the elements in `Z` as the *x*- and *y*-coordinates.

`mesh(Z,C)` additionally specifies the color of the edges.

`mesh( ___ ,C)` additionally specifies the color of the edges.

`mesh(ax, ___ )` plots into the axes specified by `ax` instead of the current axes. Specify the axes as the first input argument.

`mesh( ___ ,Name,Value)` specifies surface properties using one or more name-value pair arguments. For example, `'FaceAlpha',0.5` creates a semitransparent mesh plot.

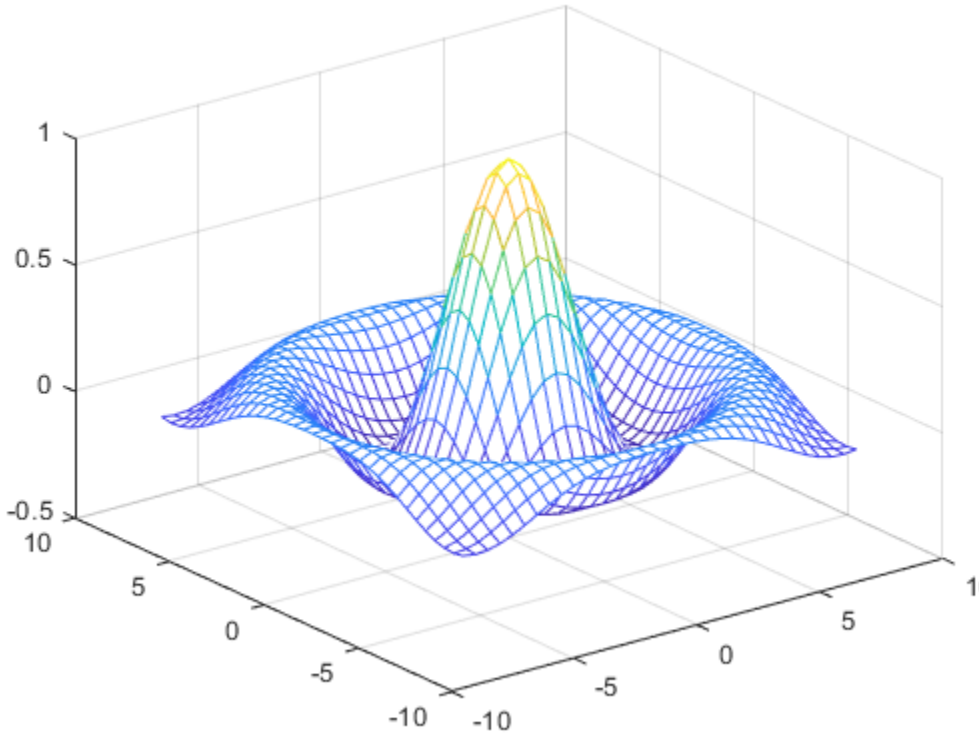
`s = mesh( ___ )` returns the chart surface object. Use `s` to modify the mesh plot after it is created. For a list of properties, see Chart Surface.

Examples

Create Mesh Plot

Create three matrices of the same size. Then plot them as a mesh plot. The plot uses Z for both height and color.

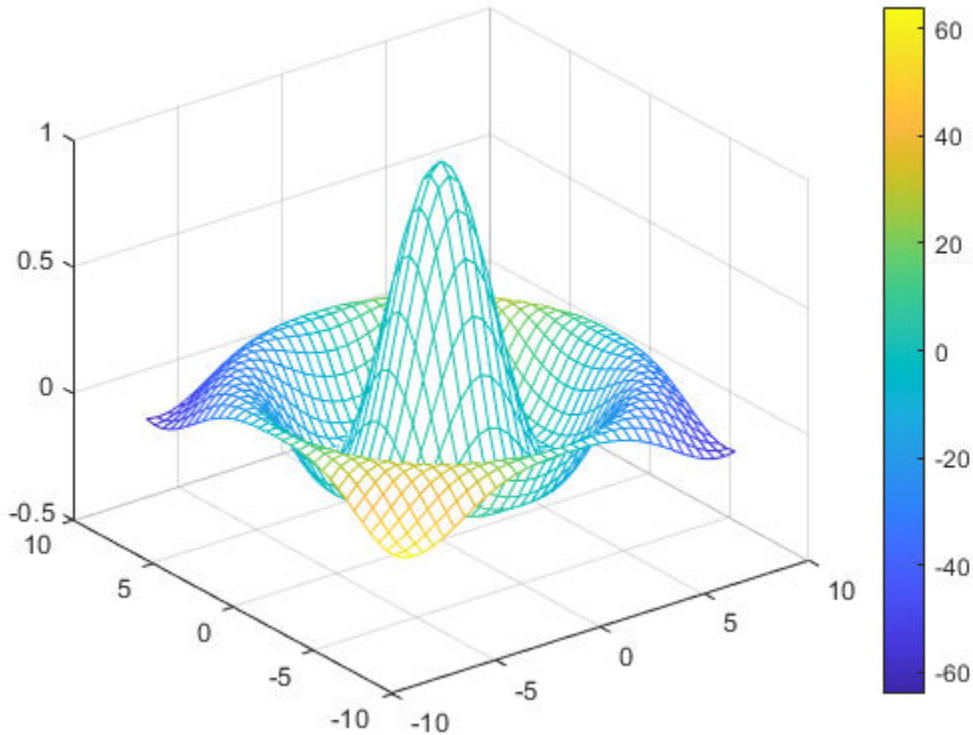
```
[X,Y] = meshgrid(-8:.5:8);
R = sqrt(X.^2 + Y.^2) + eps;
Z = sin(R)./R;
mesh(X,Y,Z)
```



Specify Colormap Colors for Mesh Plot

Specify the colors for a mesh plot by including a fourth matrix input, `C`. The mesh plot uses `Z` for height and `C` for color. Specify the colors using a *colormap*, which uses single numbers to stand for colors on a spectrum. When you use a colormap, `C` is the same size as `Z`. Add a color bar to the graph to show how the data values in `C` correspond to the colors in the colormap.

```
[X,Y] = meshgrid(-8:.5:8);
R = sqrt(X.^2 + Y.^2) + eps;
Z = sin(R)./R;
C = X.*Y;
mesh(X,Y,Z,C)
colorbar
```

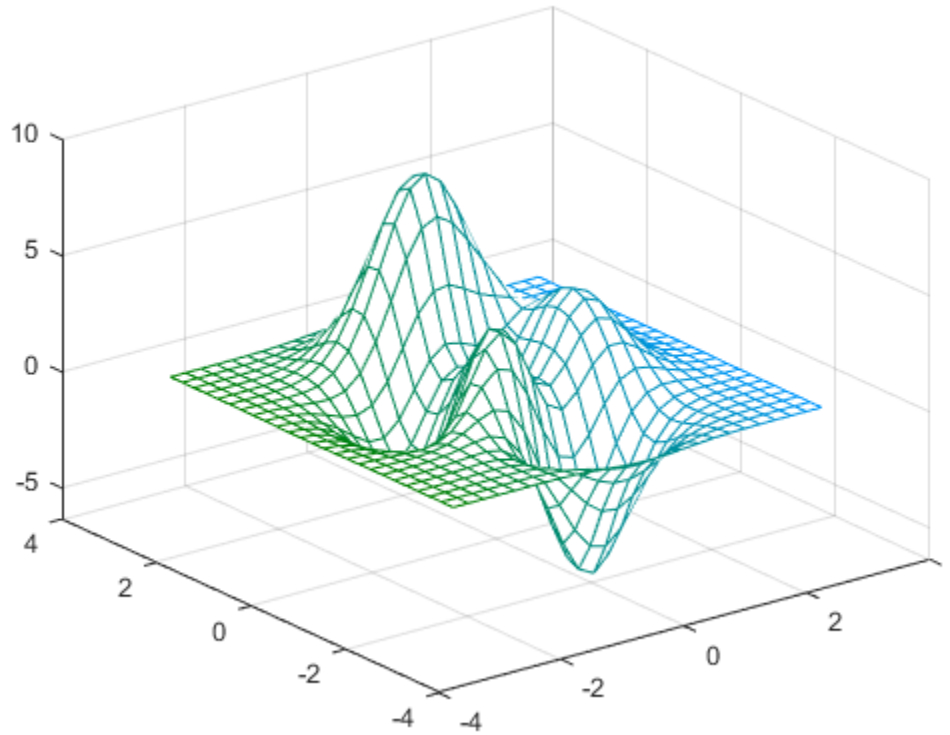


Specify True Colors for Mesh Plot

Specify the colors for a mesh plot by including a fourth matrix input, `C0`. The mesh plot uses `Z` for height and `C0` for color. Specify the colors using *truecolor*, which uses triplets of numbers to stand for all possible colors. When you use truecolor, if `Z` is `m`-by-`n`, then `C0` is `m`-by-`n`-by-3. The first page of the array indicates the red component for each color, the second page indicates the green component, and the third page indicates the blue component.

```
[X,Y,Z] = peaks(25);
C0(:,:,1) = zeros(25); % red
```

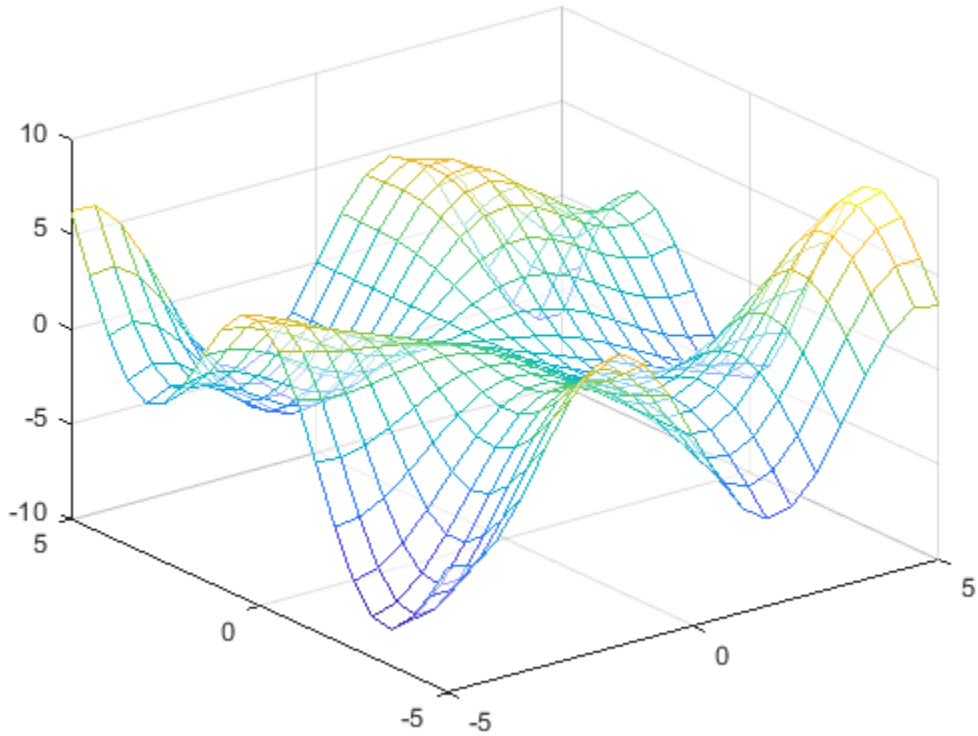
```
C0(:,:,2) = ones(25).*linspace(0.5,0.6,25); % green
C0(:,:,3) = ones(25).*linspace(0,1,25); % blue
mesh(X,Y,Z,C0)
```



Modify Mesh Plot Appearance

Create a semitransparent mesh surface by specifying the `FaceAlpha` name-value pair with `0.5` as the value. To allow further modifications, assign the surface object to the variable `s`.

```
[X,Y] = meshgrid(-5:.5:5);
Z = Y.*sin(X) - X.*cos(Y);
s = mesh(X,Y,Z,'FaceAlpha','0.5')
```

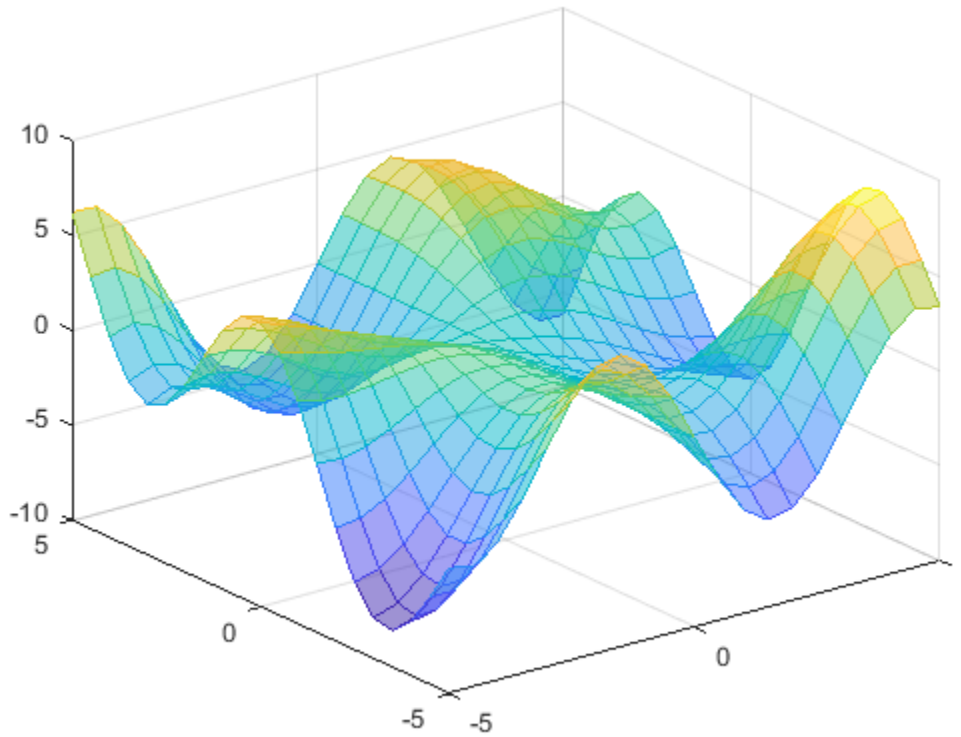



```
s =
  Surface with properties:

       EdgeColor: 'flat'
       LineStyle: '-'
       FaceColor: [1 1 1]
   FaceLighting: 'none'
       FaceAlpha: 0.5000
           XData: [21x21 double]
           YData: [21x21 double]
```

```
            ZData: [21x21 double]
            CData: [21x21 double]

  Show all properties
```

Use `s` to access and modify properties of the mesh plot after it is created. For example, add color to the face of the mesh plot by setting the `FaceColor` property.

```
s.FaceColor = 'flat';
```



Input Arguments

X — *x*-coordinates
matrix | vector

x-coordinates, specified as a matrix the same size as Z, or as a vector with length n, where `[m,n] = size(Z)`. If you do not specify values for X and Y, `mesh` uses the vectors `(1:n)` and `(1:m)`.

You can use the `meshgrid` function to create X and Y matrices.

The `XData` property of the surface object stores the *x*-coordinates.

Example: `X = 1:10`

Example: `X = [1 2 3; 1 2 3; 1 2 3]`

Example: `[X,Y] = meshgrid(-5:0.5:5)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

Y — *y*-coordinates
matrix | vector

y-coordinates, specified as a matrix the same size as Z or as a vector with length m, where `[m,n] = size(Z)`. If you do not specify values for X and Y, `mesh` uses the vectors `(1:n)` and `(1:m)`.

You can use the `meshgrid` function to create the X and Y matrices.

The `YData` property of the surface object stores the *y*-coordinates.

Example: `Y = 1:10`

Example: `Y = [1 1 1; 2 2 2; 3 3 3]`

Example: `[X,Y] = meshgrid(-5:0.5:5)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

Z — *z*-coordinates
matrix

z-coordinates, specified as a matrix. Z must have at least two rows and two columns.

`Z` specifies the height of the mesh plot at each *x*-*y* coordinate. If you do not specify the colors, then `Z` also specifies the mesh edge colors.

The `ZData` property of the surface object stores the *z*-coordinates.

Example: `Z = [1 2 3; 4 5 6]`

Example: `Z = sin(x) + cos(y)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`C` — Color array
matrix | `m`-by-`n`-by-`3` array of RGB triplets

Color array, specified as an `m`-by-`n` matrix of colormap indices or as an `m`-by-`n`-by-`3` array of RGB triplets, where `Z` is `m`-by-`n`.

- To use colormap colors, specify `C` as a matrix. For each grid point on the mesh surface, `C` indicates a color in the colormap. The `CDataMapping` property of the surface object controls how the values in `C` correspond to colors in the colormap.
- To use truecolor colors, specify `C` as an array of RGB triplets.

For more information, see “Differences Between Colormaps and Truecolor”.

The `CData` property of the surface object stores the color array. For additional control over the surface coloring, use the `FaceColor` and `EdgeColor` properties.

`ax` — Axes to plot in
axes object

Axes to plot in, specified as an `axes` object. If you do not specify the axes, then `mesh` plots into the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

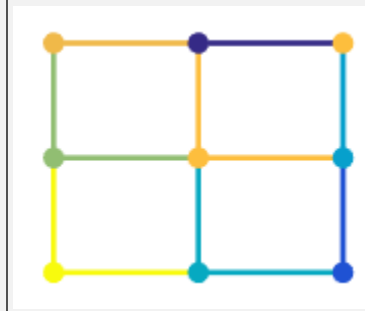
Note The properties listed here are only a subset. For a full list, see Chart Surface.

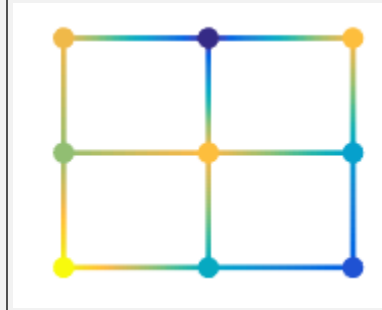
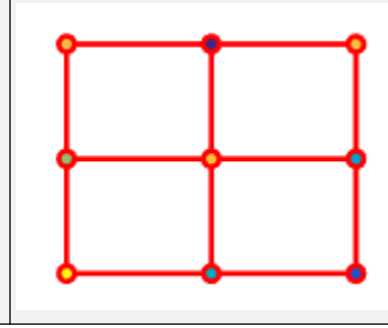
Example: `mesh(X,Y,Z,'FaceAlpha',0.5)` creates a semitransparent mesh plot.

`EdgeColor` — Edge line color

`[0 0 0]` (default) | `'none'` | `'flat'` | `'interp'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Edge line color, specified as one of the values listed here. The default color of `[0 0 0]` corresponds to black edges.

Value	Description
`'none'`	Do not draw the edges.
`'flat'`	Use a different color for each edge based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color value at the first vertex of each face (in the positive *x* and *y* directions) determines the color for the adjacent edges. You cannot use this value when the `EdgeAlpha` property is set to `'interp'`. 

Value	Description
`'interp'`	Use interpolated coloring for each edge based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color varies across each edge by linearly interpolating the color values at the vertices. You cannot use this value when the `EdgeAlpha` property is set to `'flat'`. 
RGB triplet, hexadecimal color code, or color name	Use the specified color for all the edges. This option does not use the color values in the `CData` property. 

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

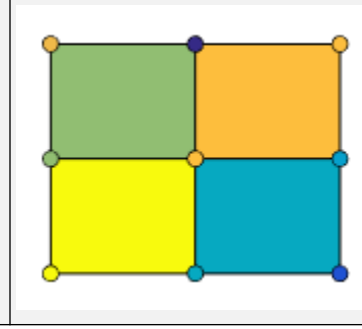
Line style, specified as one of the options listed in this table.

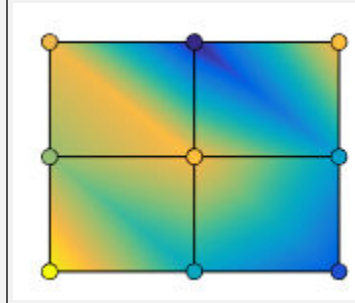
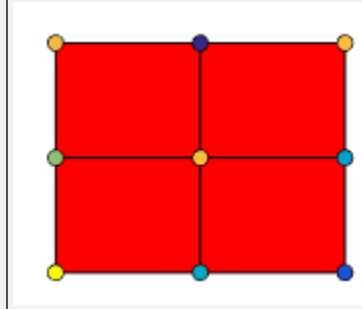
Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`FaceColor` — Face color

`'flat'` (default) | `'interp'` | `'none'` | `'texturemap'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Face color, specified as one of the values in this table.

Value	Description
`'flat'`	Use a different color for each face based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color value at the first vertex of each face (in the positive *x* and *y* directions) determines the color for the entire face. You cannot use this value when the `FaceAlpha` property is set to `'interp'`. 

Value	Description
`'interp'`	Use interpolated coloring for each face based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color varies across each face by interpolating the color values at the vertices. You cannot use this value when the `FaceAlpha` property is set to `'flat'`. 
RGB triplet, hexadecimal color code, or color name	Use the specified color for all the faces. This option does not use the color values in the `CData` property. 
`'texturemap'`	Transform the color data in `CData` so that it conforms to the surface.
`'none'`	Do not draw the faces.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

FaceAlpha — Face transparency
1 (default) | scalar in range `[0,1]` | `'flat'` | `'interp'` | `'texturemap'`

Face transparency, specified as one of these values:

- Scalar in range `[0,1]` — Use uniform transparency across all the faces. A value of `1` is fully opaque and `0` is completely transparent. Values between `0` and `1` are semitransparent. This option does not use the transparency values in the `AlphaData` property.
- `'flat'` — Use a different transparency for each face based on the values in the `AlphaData` property. The transparency value at the first vertex determines the transparency for the entire face. First you must specify the `AlphaData` property as a matrix the same size as the `ZData` property. The `FaceColor` property also must be set to `'flat'`.
- `'interp'` — Use interpolated transparency for each face based on the values in `AlphaData` property. The transparency varies across each face by interpolating the values at the vertices. First you must specify the `AlphaData` property as a matrix the same size as the `ZData` property. The `FaceColor` property also must be set to `'interp'`.
- `'texturemap'` — Transform the data in `AlphaData` so that it conforms to the surface.

FaceLighting — Effect of light objects on faces
`'flat'` (default) | `'gouraud'` | `'none'`

Effect of light objects on faces, specified as one of these values:

- `'flat'` — Apply light uniformly across each face. Use this value to view faceted objects.
- `'gouraud'` — Vary the light across the faces. Calculate the light at the vertices and then linearly interpolate the light across the faces. Use this value to view curved surfaces.
- `'none'` — Do not apply light from light objects to the faces.

To add a light object to the axes, use the `light` function.

Note The `'phong'` value has been removed. Use `'gouraud'` instead.

Tips

- To remove hidden lines from the plot, use the `hidden` function.
- To control the color shading of the plot surfaces, use the `shading` function.
- To create a three-dimensional surface with face colors, use the `surf` function.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions
`colormap` | `griddata` | `imagesc` | `meshc` | `meshgrid` | `meshz` | `pcolor` | `shading` | `surf` | `view`

Properties
Surface

Topics

"Representing Data as a Surface"
"How Surface Plot Data Relates to a Colormap"

Introduced before R2006a

meshc

Contour plot under mesh surface plot

Syntax

```
meshc(X,Y,Z)
meshc(Z)
meshc( ___ ,C)

meshc(ax, ___ )
sc = meshc( ___ )
```

Description

`meshc(X,Y,Z)` creates a mesh plot with a contour plot underneath. A mesh plot is a three-dimensional surface that has solid edge colors and no face colors. The function plots the values in matrix `Z` as heights above a grid in the *x*-*y* plane defined by `X` and `Y`. The edge colors vary according to the heights specified by `Z`.

`meshc(Z)` creates a mesh and contour plot and uses the column and row indices of the elements in `Z` as the *x*- and *y*-coordinates.

`meshc( ___ ,C)` additionally specifies the color of the edges.

`meshc(ax, ___ )` plots into the axes specified by `ax` instead of the current axes. Specify the axes as the first input argument.

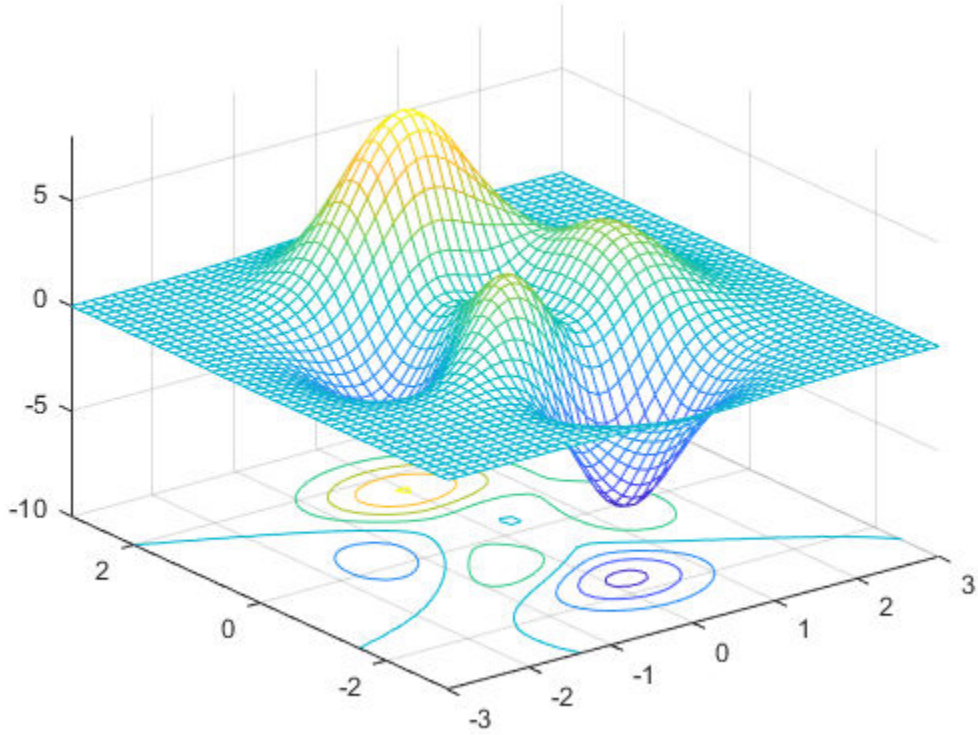
`sc = meshc( ___ )` returns a graphics array that includes the chart surface object and the contour object. Use `sc` to modify the mesh and contour plots after they are created. For a list of properties, see Chart Surface and Contour Properties.

Examples

Display Contour Plot Under Mesh Plot

Create three matrices of the same size. Then plot them as a mesh plot with a contour plot underneath. The mesh plot uses Z for both height and color.

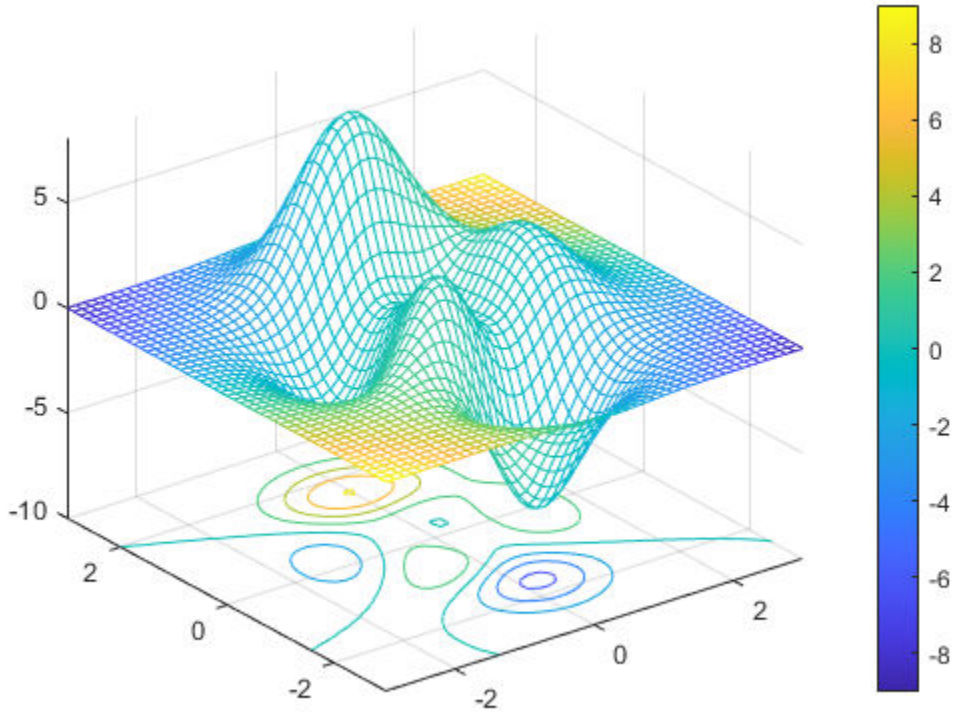
```
[X,Y] = meshgrid(-3:.125:3);
Z = peaks(X,Y);
meshc(X,Y,Z)
```



Specify Colormap Colors for Mesh and Contour Plot

Specify the colors for a mesh and contour plot by including a fourth matrix input, `C`. The plots use `Z` for height and `C` for color. Specify the colors using a *colormap*, which uses single numbers to stand for colors on a spectrum. When you use a colormap, `C` is the same size as `Z`. Add a color bar to the graph to show how the data values in `C` correspond to the colors in the colormap.

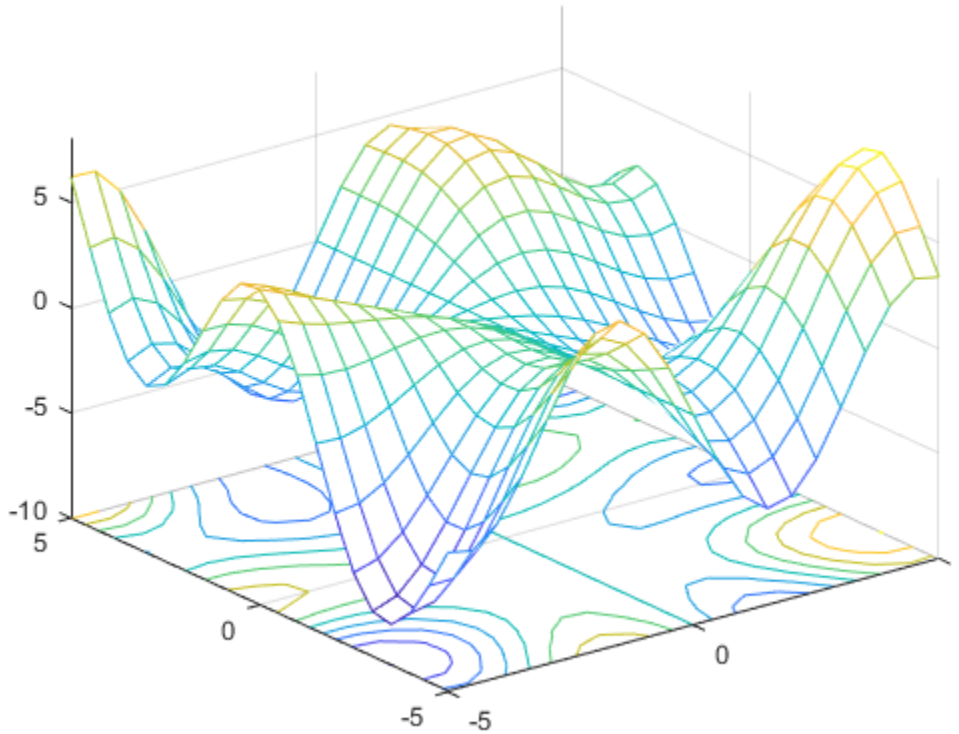
```
[X,Y] = meshgrid(-3:.125:3);
Z = peaks(X,Y);
C = X.*Y;
meshc(X,Y,Z,C)
colorbar
```



Modify Appearance of Mesh and Contour Plots

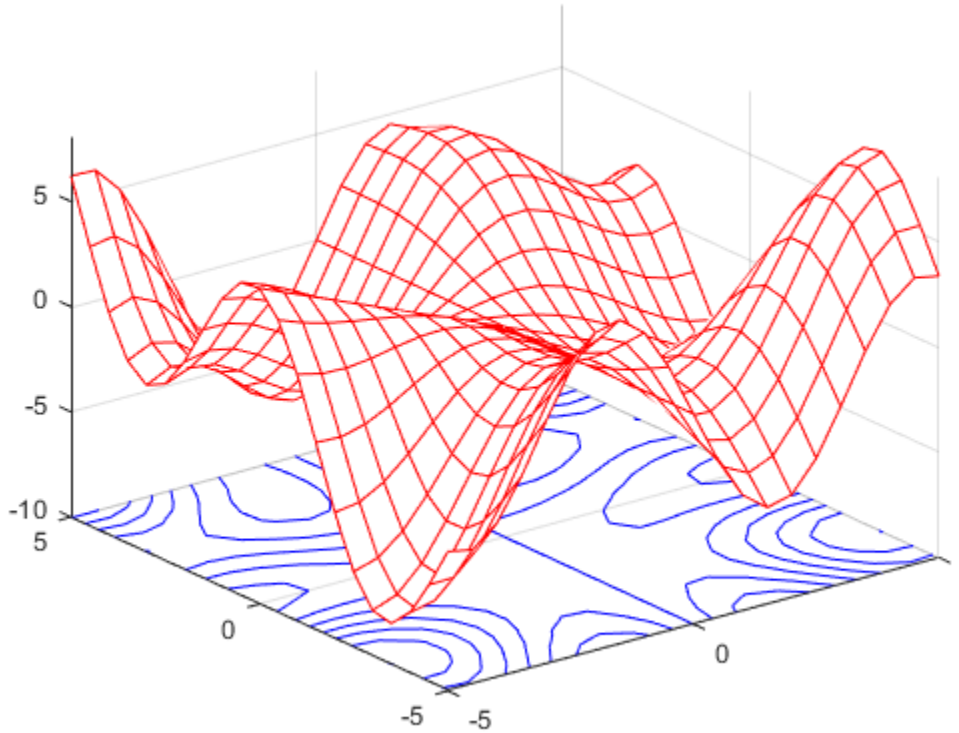
Create a mesh plot with a contour plot underneath it. To allow further modifications, assign the graphics array containing the surface and contour objects to the variable `sc`.

```
[X,Y] = meshgrid(-5:.5:5);
Z = Y.*sin(X) - X.*cos(Y);
sc = meshc(X,Y,Z);
```



Index into `sc` to access and modify properties of the mesh and contour plots after they are created. The mesh plot is accessible as `sc(1)` and the contour plot as `sc(2)`. For example, change the edge colors of the two plots by setting the `EdgeColor` properties.

```
sc(1).EdgeColor = 'r';
sc(2).EdgeColor = 'b';
```



Input Arguments

X — *x*-coordinates
matrix | vector

x-coordinates, specified as a matrix the same size as `Z`, or as a vector with length `n`, where `[m,n] = size(Z)`. If you do not specify values for `X` and `Y`, `meshc` uses the vectors `(1:n)` and `(1:m)`.

When X is a matrix, the values must be strictly increasing or decreasing along one dimension and remain constant along the other dimension. The dimension that varies must be the opposite of the dimension that varies in Y. You can use the `meshgrid` function to create X and Y matrices.

When X is a vector, the values must be strictly increasing or decreasing.

The `XData` properties of the surface and contour objects store the *x*-coordinates.

Example: `X = 1:10`

Example: `X = [1 2 3; 1 2 3; 1 2 3]`

Example: `[X,Y] = meshgrid(-5:0.5:5)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical`

Y — *y*-coordinates
matrix | vector

y-coordinates, specified as a matrix the same size as Z or as a vector with length `m`, where `[m,n] = size(Z)`. If you do not specify values for X and Y, `meshc` uses the vectors `(1:n)` and `(1:m)`.

When Y is a matrix, the values must be strictly increasing or decreasing along one dimension and remain constant along the other dimension. The dimension that varies must be the opposite of the dimension that varies in X. You can use the `meshgrid` function to create X and Y matrices.

When Y is a vector, the values must be strictly increasing or decreasing.

The `YData` properties of the surface and contour objects store the *y*-coordinates.

Example: `Y = 1:10`

Example: `Y = [1 1 1; 2 2 2; 3 3 3]`

Example: `[X,Y] = meshgrid(-5:0.5:5)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical`

Z — *z*-coordinates
matrix

z-coordinates, specified as a matrix. `Z` must have at least two rows and two columns.

`Z` specifies the height of the mesh plot at each *x*-*y*-coordinate. If you do not specify the colors, then `Z` also specifies the mesh edge colors.

The `ZData` properties of the surface and contour objects store the *z*-coordinates.

Example: `Z = [1 2 3; 4 5 6]`

Example: `Z = sin(x) + cos(y)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical`

`C` — Color array
matrix | `m`-by-`n`-by-`3` array of RGB triplets

Color array, specified as an `m`-by-`n` matrix of colormap indices or as an `m`-by-`n`-by-`3` array of RGB triplets, where `Z` is `m`-by-`n`.

- To use colormap colors, specify `C` as a matrix. For each grid point on the mesh surface, `C` indicates a color in the colormap. The `CDataMapping` property of the surface object controls how the values in `C` correspond to colors in the colormap.
- To use truecolor colors, specify `C` as an array of RGB triplets.

For more information, see “Differences Between Colormaps and Truecolor”.

The `CData` property of the surface object stores the color array. For additional control over the surface coloring, use the `FaceColor` and `EdgeColor` properties.

`ax` — Axes to plot in
axes object

Axes to plot in, specified as an `axes` object. If you do not specify the axes, then `meshc` plots into the current axes.

Tips

- To remove hidden lines from the plot, use the `hidden` function.
- To control the color shading of the plot surfaces, use the `shading` function.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions

`colormap` | `contour` | `imagesc` | `mesh` | `meshgrid` | `meshz` | `pcolor` | `shading` | `surfc` | `view`

Properties

Chart Surface | Contour

Topics

“Representing Data as a Surface”
“How Surface Plot Data Relates to a Colormap”

Introduced before R2006a

meshz

Mesh surface plot with curtain

Syntax

```
meshz(X,Y,Z)
meshz(X,Y,Z,C)

meshz(Z)
meshz(Z,C)

meshz(ax, ___ )
s = meshz( ___ )
```

Description

`meshz(X,Y,Z)` creates a mesh plot with a curtain around it. A mesh plot is a three-dimensional surface that has solid edge colors and no face colors. The function plots the values in matrix `Z` as heights above a grid in the *x*-*y* plane defined by `X` and `Y`. The edge colors vary according to the heights specified by `Z`.

`meshz(X,Y,Z,C)` additionally specifies the color of the edges.

`meshz(Z)` creates a mesh plot with a curtain, and uses the column and row indices of the elements in `Z` as the *x*- and *y*-coordinates.

`meshz(Z,C)` additionally specifies the color of the edges.

`meshz(ax, ___ )` plots into the axes specified by `ax` instead of the current axes. Specify the axes as the first input argument.

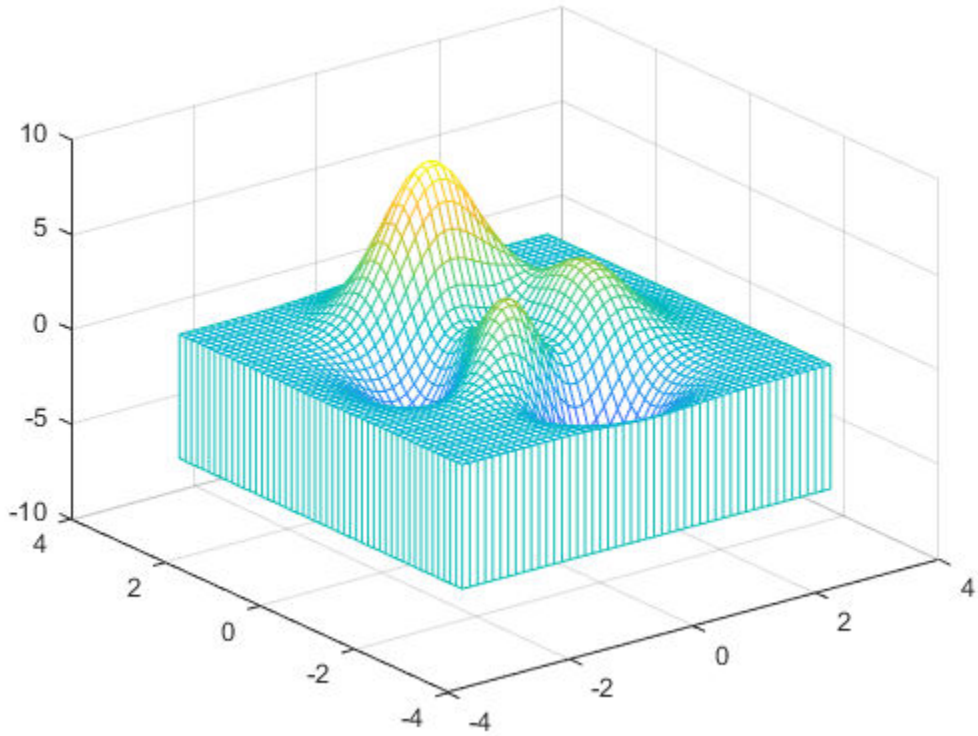
`s = meshz( ___ )` returns the chart surface object. Use `s` to modify the mesh plot after it is created. For a list of properties, see Chart Surface.

Examples

Display Curtain Around Mesh Plot

Create three matrices of the same size. Then plot them as a mesh plot with a curtain. The mesh plot uses Z for both height and color.

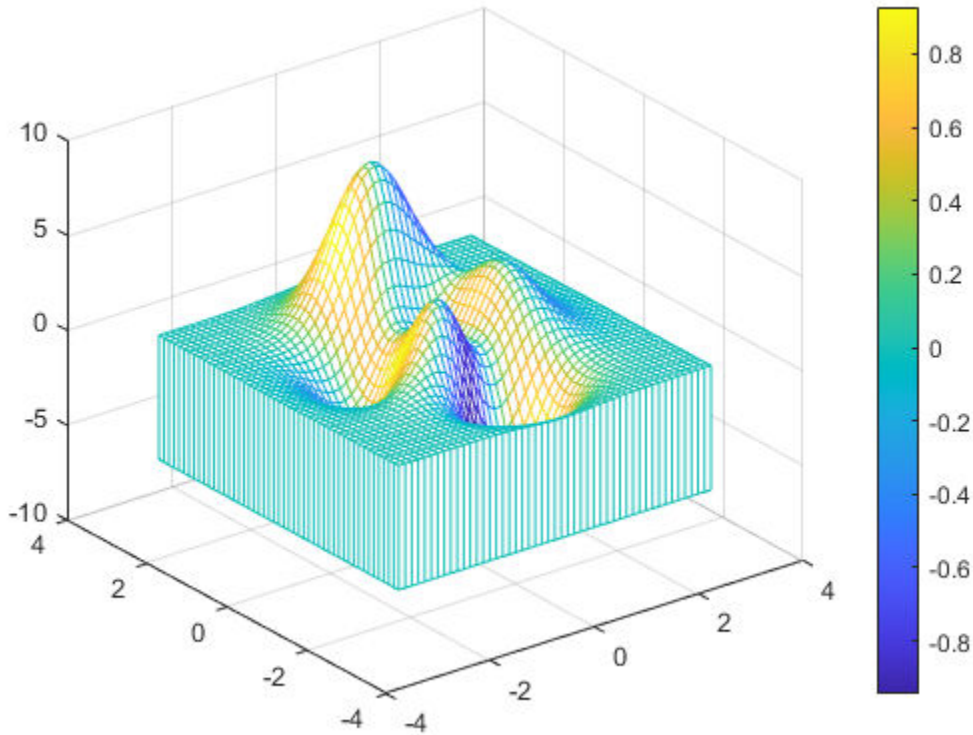
```
[X,Y] = meshgrid(-3:.125:3);
Z = peaks(X,Y);
meshz(X,Y,Z)
```



Specify Colormap Colors for Mesh Plot With Curtain

Specify the colors for a mesh plot and surrounding curtain by including a fourth matrix input, C. The mesh plot uses Z for height and C for color. Specify the colors using a *colormap*, which uses single numbers to stand for colors on a spectrum. When you use a colormap, C is the same size as Z. Add a color bar to the graph to show how the data values in C correspond to the colors in the colormap.

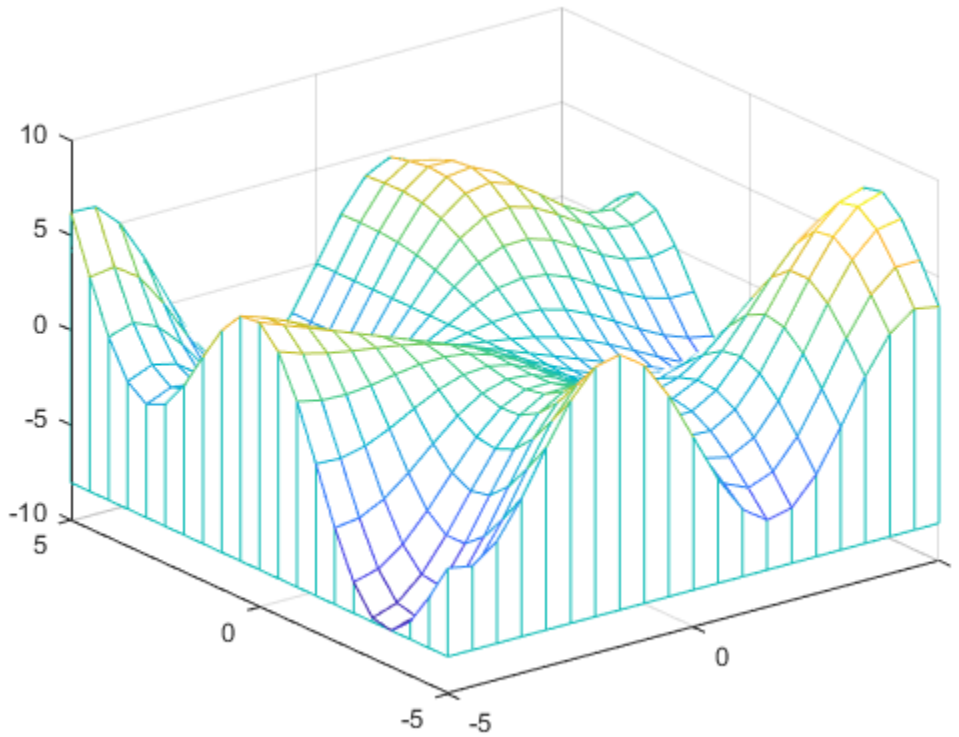
```
[X,Y] = meshgrid(-3:.125:3);
Z = peaks(X,Y);
C = gradient(Z);
meshz(X,Y,Z,C)
colorbar
```



Modify Appearance of Mesh Plot With Curtain

Create a mesh plot with a curtain around it. To allow further modifications, assign the surface object to the variable `s`.

```
[X,Y] = meshgrid(-5:.5:5);
Z = Y.*sin(X) - X.*cos(Y);
s = meshz(X,Y,Z)
```

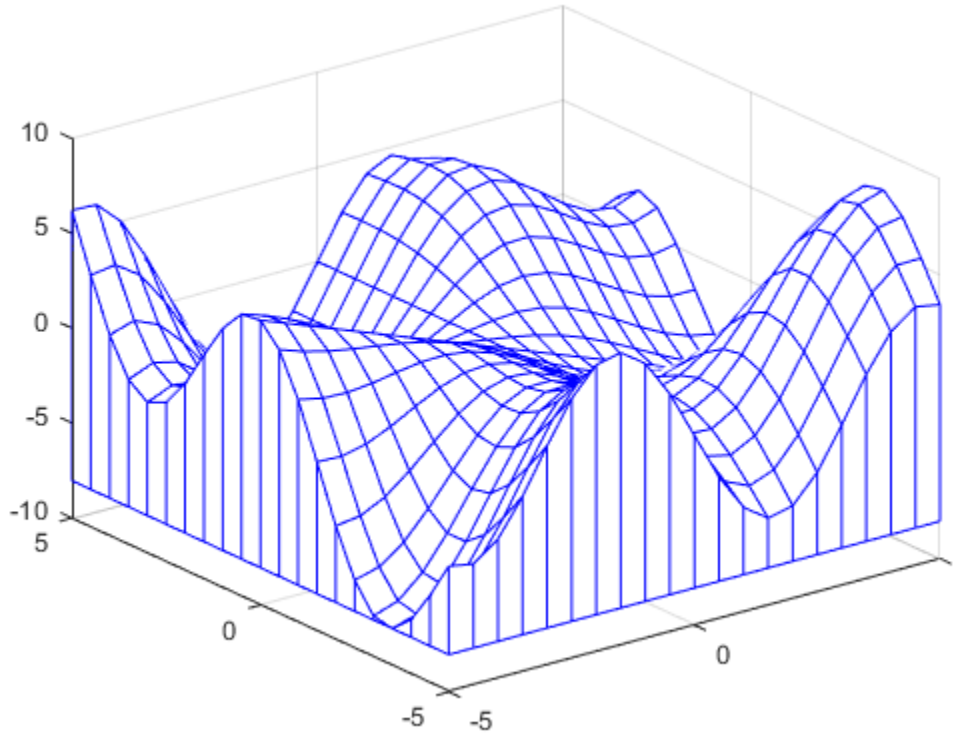



```
s =
  Surface (meshz) with properties:
```

```
       EdgeColor: 'flat'
       LineStyle: '-'
       FaceColor: [1 1 1]
    FaceLighting: 'none'
       FaceAlpha: 1
           XData: [25x25 double]
           YData: [25x25 double]
           ZData: [25x25 double]
           CData: [25x25 double]

  Show all properties
```

Use `s` to access and modify properties of the mesh plot after it is created. For example, change the color of the mesh plot edges and surrounding curtain by setting the `EdgeColor` property.

```
s.EdgeColor = 'b';
```



Input Arguments

X — *x*-coordinates
matrix | vector

x-coordinates, specified as a matrix the same size as `Z`, or as a vector with length `n`, where `[m,n] = size(Z)`. If you do not specify values for `X` and `Y`, `meshz` uses the vectors `(1:n)` and `(1:m)`.

When `X` is a matrix, the values must be strictly increasing or decreasing along one dimension and remain constant along the other dimension. The dimension that varies

must be the opposite of the dimension that varies in Y. You can use the `meshgrid` function to create X and Y matrices.

When X is a vector, the values must be strictly increasing or decreasing.

The `XData` properties of the surface object stores the *x*-coordinates.

Example: `X = 1:10`

Example: `X = [1 2 3; 1 2 3; 1 2 3]`

Example: `[X,Y] = meshgrid(-5:0.5:5)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical`

Y — *y*-coordinates
matrix | vector

y-coordinates, specified as a matrix the same size as Z or as a vector with length `m`, where `[m,n] = size(Z)`. If you do not specify values for X and Y, `meshz` uses the vectors `(1:n)` and `(1:m)`.

When Y is a matrix, the values must be strictly increasing or decreasing along one dimension and remain constant along the other dimension. The dimension that varies must be the opposite of the dimension that varies in X. You can use the `meshgrid` function to create X and Y matrices.

When Y is a vector, the values must be strictly increasing or decreasing.

The `YData` properties of the surface objects store the *y*-coordinates.

Example: `Y = 1:10`

Example: `Y = [1 1 1; 2 2 2; 3 3 3]`

Example: `[X,Y] = meshgrid(-5:0.5:5)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical`

Z — *z*-coordinates
matrix

z-coordinates, specified as a matrix. Z must have at least two rows and two columns.

`Z` specifies the height of the mesh plot at each *x*-*y* coordinate. If you do not specify the colors, then `Z` also specifies the mesh edge colors.

The `ZData` property of the surface object stores the *z*-coordinates.

Example: `Z = [1 2 3; 4 5 6]`

Example: `Z = sin(x) + cos(y)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical`

`C` — Color array
matrix

Color array, specified as a an m-by-n matrix of colormap indices, where `Z` is `m`-by-`n`. For each grid point on the mesh surface, `C` indicates a color in the colormap. The `CDataMapping` property of the surface object controls how the values in `C` correspond to colors in the colormap.

The `CData` property of the surface object stores the color array. For additional control over the surface coloring, use the `FaceColor` and `EdgeColor` properties.

`ax` — Axes to plot in
axes object

Axes to plot in, specified as an `axes` object. If you do not specify the axes, then `meshz` plots into the current axes.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions

`colormap` | `imagesc` | `mesh` | `meshc` | `meshgrid` | `pcolor` | `shading` | `view` | `waterfall`

Properties

Chart Surface | Contour

Topics

“Representing Data as a Surface”
“How Surface Plot Data Relates to a Colormap”

Introduced before R2006a

meshgrid

2-D and 3-D grids

Syntax

```
[X,Y] = meshgrid(x,y)
[X,Y] = meshgrid(x)

[X,Y,Z] = meshgrid(x,y,z)
[X,Y,Z] = meshgrid(x)
```

Description

`[X,Y] = meshgrid(x,y)` returns 2-D grid coordinates based on the coordinates contained in vectors `x` and `y`. `X` is a matrix where each row is a copy of `x`, and `Y` is a matrix where each column is a copy of `y`. The grid represented by the coordinates `X` and `Y` has `length(y)` rows and `length(x)` columns.

`[X,Y] = meshgrid(x)` is the same as `[X,Y] = meshgrid(x,x)`, returning square grid coordinates with grid size `length(x)`-by-`length(x)`.

`[X,Y,Z] = meshgrid(x,y,z)` returns 3-D grid coordinates defined by the vectors `x`, `y`, and `z`. The grid represented by `X`, `Y`, and `Z` has size `length(y)`-by-`length(x)`-by-`length(z)`.

`[X,Y,Z] = meshgrid(x)` is the same as `[X,Y,Z] = meshgrid(x,x,x)`, returning 3-D grid coordinates with grid size `length(x)`-by-`length(x)`-by-`length(x)`.

Examples

2-D Grid

Create 2-D grid coordinates with *x*-coordinates defined by the vector `x` and *y*-coordinates defined by the vector `y`.

```
x = 1:3;
y = 1:5;
[X,Y] = meshgrid(x,y)
```

```
X = 5×3

     1     2     3
     1     2     3
     1     2     3
     1     2     3
     1     2     3


Y = 5×3

     1     1     1
     2     2     2
     3     3     3
     4     4     4
     5     5     5
```

Evaluate the expression $x^2 + y^2$ over the 2-D grid.

```
X.^2 + Y.^2
```

```
ans = 5×3

     2     5    10
     5     8    13
    10    13    18
    17    20    25
    26    29    34
```

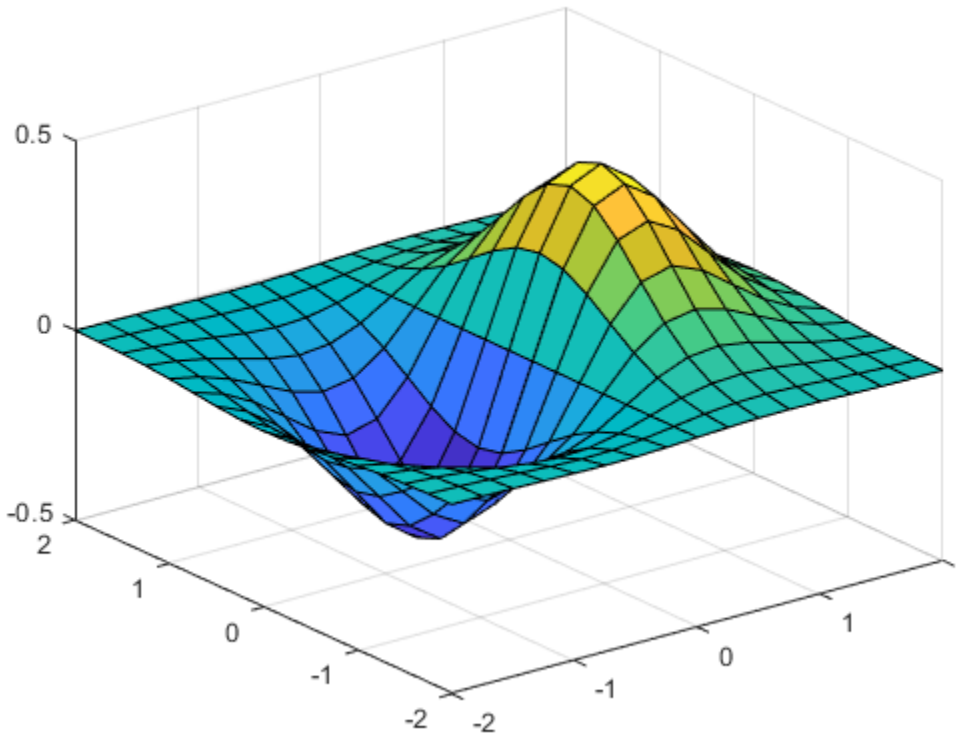
Plot Surface

Create a 2-D grid with uniformly spaced *x*-coordinates and *y*-coordinates in the interval [-2,2].

```
x = -2:0.25:2;
y = x;
[X,Y] = meshgrid(x);
```

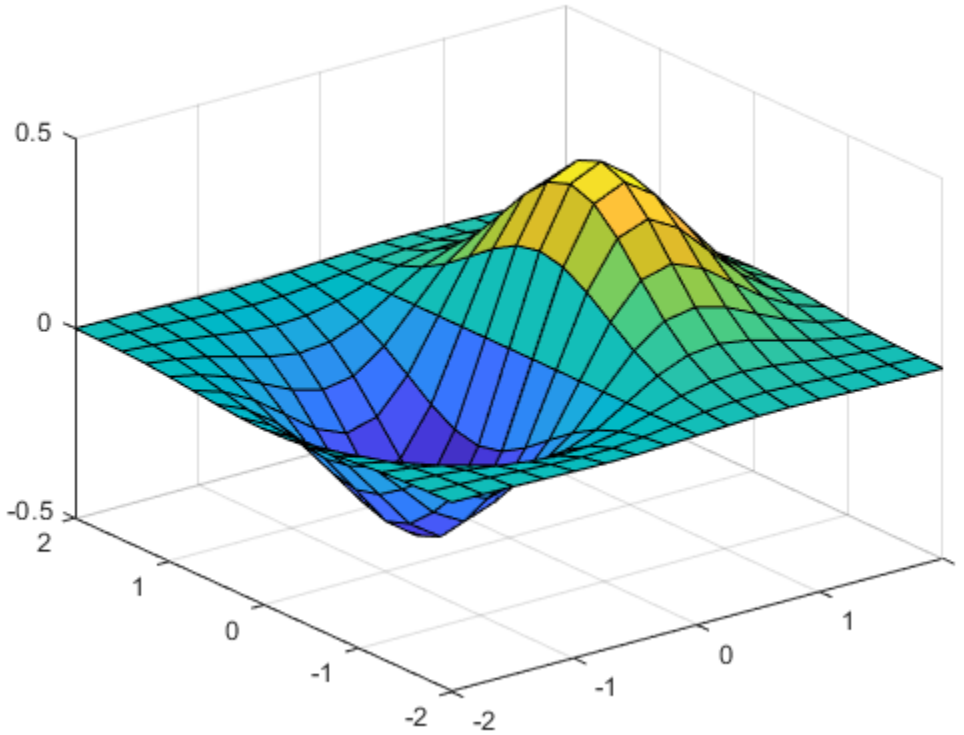
Evaluate and plot the function $f(x, y) = xe^{-x^2 - y^2}$ over the 2-D grid.

```
F = X.*exp(-X.^2-Y.^2);
surf(X,Y,F)
```



Starting in R2016b, it is not always necessary to create the grid before operating over it. For example, computing the expression $xe^{-x^2 - y^2}$ implicitly expands the vectors `x` and `y`. For more information on implicit expansion, see “Array vs. Matrix Operations”.

```
surf(x,y,x.*exp(-x.^2-(y').^2))
```



3-D Grid

Create 3-D grid coordinates from *x*-, *y*-, and *z*-coordinates defined in the interval [0,6], and evaluate the expression $x^2 + y^2 + z^2$.

```
x = 0:2:6;
y = 0:1:6;
z = 0:3:6;
[X,Y,Z] = meshgrid(x,y,z);
F = X.^2 + Y.^2 + Z.^2;
```

Determine the size of the grid. The three coordinate vectors have different lengths, forming a rectangular box of grid points.

```
gridsize = size(F)
```

```
gridsize = 1×3

     7     4     3
```

Use the single-input syntax to generate a uniformly spaced 3-D grid based on the coordinates defined in `x`. The new grid forms a cube of grid points.

```
[X,Y,Z] = meshgrid(x);
G = X.^2 + Y.^2 + Z.^2;
gridsize = size(G)
```

```
gridsize = 1×3

     4     4     4
```

Input Arguments

x — x-coordinates of points
vector

x-coordinates of points, specified as a vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

y — y-coordinates of points
vector

y-coordinates of points, specified as a vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

z — z-coordinates of points
vector

z-coordinates of points, specified as a vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

X — x-coordinates over grid
2-D or 3-D array

x-coordinates over a grid, returned as a 2-D (two inputs) or 3-D array (three inputs).

Y — y-coordinates over grid
2-D or 3-D array

y-coordinates over a grid, returned as a 2-D (two inputs) or 3-D array (three inputs).

Z — z-coordinates over grid
3-D array

z-coordinates over a grid, returned as a 3-D array.

Tips

- For additional information on creating multidimensional grids, see “Grid Representation”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The inputs must be floating-point double or single.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- The inputs must be floating-point double or single.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`griddedInterpolant` | `mesh` | `ndgrid` | `surf`

Topics

“Interpolating Gridded Data”

Introduced before R2006a

meta.class

Describe MATLAB class

Description

Instances of the `meta.class` class contain information about MATLAB classes. The read/write properties of the `meta.class` class correspond to class attributes and are set only from within class definitions on the `classdef` line. You can query the read–only properties of the `meta.class` object to obtain information that is specified syntactically by the class (for example, to obtain the name of the class).

You cannot instantiate a `meta.class` object directly. You can construct a `meta.class` object from an instance of a class or using the class name:

- `metaclass` — returns a `meta.class` object representing the object passed as an argument.
- `?`*`ClassName`* — returns a `meta.class` object representing the named class.
- `fromName` — static method returns a `meta.class` object representing the named class.

For example, the `metaclass` function returns the `meta.class` object representing `MyClass`.

```
ob = MyClass;
obmeta = metaclass(ob);
obmeta.Name
```

```
ans =
MyClass
```

You can use the class name to obtain the `meta.class` object:

```
obmeta = ?MyClass;
```

You can also use the `fromName` static method:

```
obmeta = meta.class.fromName('MyClass');
```

Properties

Property	Purpose
`Abstract` attribute, default = `false`	If `true`, this class is an abstract class (cannot be instantiated). For more information, see “Abstract Classes”.
`ConstructOnLoad` attribute, default = `false`	If `true`, MATLAB calls the class constructor automatically when loading an object from a MAT-file. Therefore, the construction must be implemented so that calling it with no arguments does not produce an error. See “Save and Load Process for Objects”
`ContainingPackage` read only	A `meta.package` object describing the package within which this class is contained, or an empty object if this class is not in a package. See “Packages Create Namespaces”.
`Description` read only	Currently not used.
`DetailedDescription` read only	Currently not used.
`Enumeration` attribute, default = `false`	If `true`, this class is an enumeration class. See “Define Enumeration Classes”.
`EventList` read only	An array of `meta.event` objects describing each event defined by this class, including all inherited events. See “Events”.
`EnumerationMemberList`	An array of `meta.EnumeratedValue` objects describing the member names defined by an enumeration class. See “Enumerations” for more information on enumeration classes.
`Hidden` attribute, default = `false`	If set to `true`, the class does not appear in the output of MATLAB commands or tools that display class names.

Property	Purpose
`InferiorClasses` attribute, default = `{}`	A cell array of `meta.class` objects defining the precedence of classes represented by the list as inferior to this class. See “Class Precedence”
`MethodList` read only	An array of `meta.method` objects describing each method defined by this class, including all inherited public and protected methods. See “Methods in Class Design”.
`Name` read only	Name of the class associated with this `meta.class` object (`char` array)
`PropertyList` read only	An array of `meta.property` objects describing each property defined by this class, including all inherited public and protected properties. See “Properties”.
`RestrictsSubclassing` read only	If the class is not `Sealed` and has an empty list of `AllowedSubclasses` in the class definition, this property is set to `false`. If the class is `Sealed` or specifies any `AllowedSubclasses`, this property is set to `true`. See “Specify Allowed Subclasses”.
`Sealed` attribute, default = `false`	If `true`, the class cannot be subclassed.
`SuperclassList` read only	An array of `meta.class` objects describing each direct superclass from which this class is derived. See “Design Subclass Constructors”.

Methods

Method	Purpose
`fromName`	Returns the `meta.class` object associated with the specified class name.
`tf = eq(Cls)`	Equality function (`a == b`). Use to test if two variables refer to equal classes (classes that contain exactly the same list of elements).
`tf = ne(Cls)`	Not equal function (`a ~= b`). Use to test if two variables refer to different meta–classes.
`tf = lt(ClsA,ClsB)`	Less than function (`ClsA < ClsB`). Use to determine if `ClsA` is a strict subclass of `ClsB` (i.e., a strict subclass means `ClsX < ClsX` is `false`).
`tf = le(ClsA,ClsB)`	Less than or equal to function (`ClsA <= ClsB`). Use to determine if `ClsA` is a subclass of `ClsB`.
`tf = gt(ClsA,ClsB)`	Greater than function (`ClsA > ClsB`). Use to determine if `ClsA` is a strict superclass of `ClsB` (i.e., a strict superclass means `ClsX > ClsX` is `false`).
`tf = ge(ClsA,ClsB)`	Greater than or equal to function (`ClsA >= ClsB`). Use to determine if `ClsA` is a superclass of `ClsB`.

Events

Event	Purpose
`InstanceCreated`	If the class is a handle class, this event occurs every time a new instance of this handle class is created, including new instances of any subclasses. The event occurs immediately after all constructor functions finish executing.
`InstanceDestroyed`	If the class is a handle class, this event occurs every time an instance of this handle class is destroyed, including all subclasses. The event occurs immediately before any destructor functions execute.

Examples

Find property attributes using the `handle` class `findobj` method and the `audioplayer` `meta.class` object. Determine if you can change the `SampleRate` property on an `audioplayer` object.

```
load('handel.mat')
player = audioplayer(y,Fs);
mc = metaclass(player);
mp = findobj(mc.PropertyList,'Name','SampleRate');
if strcmp(mp.SetAccess,'public')
   player.SampleRate = 7200;
end
```

See Also

`fromName` | `meta.event` | `meta.method` | `meta.package` | `meta.property` | `metaclass`

Topics

"Class Introspection and Metadata"

meta.class.fromName

Return `meta.class` object associated with named class

Syntax

```
mcls = meta.class.fromName(ClassName)
```

Description

`mcls = meta.class.fromName(ClassName)` is a static method that returns the `meta.class` object for the specified class. *`ClassName`* can be a character vector or a string scalar.

You can also use the `?` operator to obtain the `meta.class` object for a class name:

```
mcls = ?ClassName;
```

The equivalent call to `meta.class.fromName` is:

```
mcls = meta.class.fromName('ClassName');
```

Use `meta.class.fromName` when using a character vector or string variable for the class name:

```
function mcls = getMetaClass(clname)
   % Do error checking
   mcls = meta.class.fromName(clname);
   ...
end
```

See Also

`meta.class` | `string`

meta.DynamicProperty

Describe dynamic property of MATLAB object

Description

The `meta.DynamicProperty` class contains descriptive information about dynamic properties that you have added to an instance of a MATLAB classes. The MATLAB class must be a subclass of `dynamicprops`. The properties of the `meta.DynamicProperty` class correspond to property attributes that you specify from within class definitions. Dynamic properties are not defined in `classdef` blocks, but you can set their attributes by setting the `meta.DynamicProperty` object properties.

You add a dynamic property to an object using the `addprop` method of the `dynamicprops` class. The `addprop` method returns a `meta.DynamicProperty` instance representing the new dynamic property. You can modify the properties of the `meta.DynamicProperty` object to set the attributes of the dynamic property or to add set and get access methods, which would be defined in the `classdef` for regular properties.

You cannot instantiate the `meta.DynamicProperty` class. You must use `addprop` to obtain a `meta.DynamicProperty` object. Use `findprop` to get the `meta.DynamicProperty` object for an object with a dynamic property.

To remove the dynamic property, call the `delete` handle class method on the `meta.DynamicProperty` object.

The `dynamicprops` `addprop` method returns an array of `meta.DynamicProperty` objects, one for each dynamic property added.

See “Dynamic Properties — Adding Properties to an Instance” for more information.

Properties

Property	Purpose
`Name`	Name of the property.

Property	Purpose
`Description`	Currently not used
`DetailedDescription`	Currently not used
`AbortSet`	If `true`, then MATLAB does not set the property value if the new value is the same as the current value. This approach prevents the triggering of property `PreSet` and `PostSet` events.
`Abstract` attribute, default = `false`	If `true`, the property has no implementation, but a concrete subclass must redefine this property without `Abstract` being set to `true`. • Abstract properties cannot define set or get access methods. See “Property Access Methods” • Abstract properties cannot define initial values. “Assigning a Default Value” • All subclasses must specify the same values as the superclass for the property `SetAccess` and `GetAccess` attributes. • `Abstract=true` should be used with the class attribute `Sealed=false` (the default).

Property	Purpose
`Access`	The Access attribute a of dynamic property applies to the class of the instance that contains the dynamic property. For more information, see “Access Attribute for Dynamic Properties”. `public` – unrestricted access `protected` – access from class or subclasses `private` – access by class members only (not subclasses) List of classes that have get and set access to this property. Specify classes as `meta.class` objects in the form: • A single `meta.class` object • A cell array of `meta.class` objects. An empty cell array, `{}`, is the same as `private` access. Use `Access` to set both `SetAccess` and `GetAccess` to the same value. Query the values of `SetAccess` and `GetAccess` directly (not `Access`).
`Constant` attribute, default = `false`	Setting the `Constant` attribute of a dynamic property is not allowed. Dynamic properties cannot be constant.
`DefaultValue`	Querying this property returns an error because dynamic properties cannot define default values.
`DefiningClass`	The `meta.class` object representing the class that defines this property.
`GetAccess` attribute, default = `public`	`public` – unrestricted access `protected` – access from class or subclasses `private` – access by class members only
`SetAccess` attribute, default = `public`	`public` – unrestricted access `protected` – access from class or subclasses `private` – access by class members only

Property	Purpose
`Dependent` attribute, default = `false`	If `false`, property value is stored in object. If `true`, property value is not stored in object and the set and get functions cannot access the property by indexing into the object using the property name. See “Property Get Methods”
`Transient` attribute, default = `false`	If `true`, property value is not saved when object is saved to a file. See “Save and Load Process for Objects” for more about saving objects.
`Hidden` attribute, default = `false`	Determines whether the property should be shown in a property list (e.g., Property Inspector, call to `properties`, etc.).
`NonCopyable` default = `false`	Determines if dynamic property should be copied when object is copied. By default, dynamic properties are not copied. For more information, see “Exclude Properties from Copy”
`GetObservable` attribute, default = `false`	If `true`, then listeners can be created for access to this property. The listeners are called whenever property values are queried. See “Property-Set and Query Events”
`SetObservable` attribute, default = `false`	If `true`, then listeners can be created for access to this property. The listeners are called whenever property values are modified. See “Property-Set and Query Events”
`GetMethod`	Function handle of the get method associated with this property. Empty if there is no get method specified. See “Get Method Syntax”
`SetMethod`	Function handle of the set method associated with this property. Empty if there is no set method specified. See “Property Set Methods”
`HasDefault`	Always `false`. Dynamic properties cannot define default values.

Events

See “Listen for Changes to Property Values” for information on using property events.

Event Name	Purpose
`PreGet`	Event occurs just before property is queried.
`PostGet`	Event occurs just after property has been queried
`PreSet`	Event occurs just before this property is modified
`PostSet`	Event occurs just after this property has been modified
`ObjectBeingDestroyed`	Inherited from `handle`

See Also

`addprop` | `findprop` | `handle`

Topics

“Dynamic Properties — Adding Properties to an Instance”
“Set and Get Methods for Dynamic Properties”
“Dynamic Property Events”
“Dynamic Properties and ConstructOnLoad”

meta.EnumeratedValue

Describe enumeration member of MATLAB class

Description

The `meta.EnumeratedValue` class contains information about enumeration members defined by MATLAB classes. The properties of a `meta.EnumeratedValue` object correspond to the attributes of the enumeration member being described.

All `meta.EnumeratedValue` properties are read-only. Query the `meta.EnumeratedValue` object to obtain information about the enumeration member it describes.

Obtain a `meta.EnumeratedValue` object from the `EnumerationMemberList` property of the `meta.class` object. `EnumerationMemberList` is an array of `Meta.EnumeratedValue` instances, one per enumeration member.

The `meta.EnumeratedValue` class is a subclass of the `handle` class.

Example

To access the `meta.EnumeratedValue` objects for a class, first create a `meta.class` object for that class. For example, give the following `OnOff` class definition:

```
classdef OnOff < logical
   enumeration
      On (true)
      Off (false)
   end
end
```

Obtain a `meta.EnumeratedValue` object from the `EnumerationMemberList` property of the `meta.class` object:

```
% Obtain the meta.class instance for the OnOff class
mc = ?OnOff;
% Get the array of EnumerateValue objects
enumList = mc.EnumerationMemberList;
```

```
% Access the Name property of the first object in the array
enumList(1).Name =
ans =
On
```

Properties

Property	Purpose
`Name` read-only	Name of the enumeration member associated with this `meta.EnumeratedValue` object
`Description` read-only	Currently not used.
`DetailedDescription` read-only	Currently not used.

Methods

See the `handle` superclass for inherited methods.

Events

See the `handle` superclass for inherited events.

See Also

`meta.class` | `meta.event` | `meta.method` | `meta.property`

Topics

"Define Enumeration Classes"
"Class Introspection and Metadata"

meta.event

Describe event of MATLAB class

Description

The `meta.event` class provides information about MATLAB class events. The read/write properties of the `meta.event` class correspond to event attributes and are specified only from within class definitions.

You can query the read-only properties of the `meta.event` object to obtain information that is specified syntactically by the class (for example, to obtain the name of the class defining the event).

You cannot instantiate a `meta.event` object directly. Obtain a `meta.event` object from the `meta.class` `EventList` property, which contains an array of `meta.event` objects, one for each event defined by the class. For example, replace *`ClassName`* with the name of the class whose events you want to query:

```
mco = ?ClassName;
elist = mco.EventList;
elist(1).Name; % name of first event in list
```

Use the `metaclass` function to obtain a `meta.class` object from a class instance:

```
mco = metaclass(obj);
```

Properties

Property	Purpose
`Name` read-only	Name of the event.
`Description` read-only	This property holds the first comment line following an event name inside an events block.

Property	Purpose
`DetailedDescription` read-only	This property holds the comment lines immediately following an event definition up to the first line that does not begin with a comment. The lines are in a 1-by-n MATLAB `char` array with newline characters separating lines.
`Hidden`	If `true`, the event does not appear in the list of events returned by the `events` function (or other event listing functions or viewers)

`ListenAccess`	Determines where you can create listeners for the event.
	• `public` — unrestricted access
	• `protected` — access from methods in class or subclasses
	• `private` — access by class methods only (not from subclasses)
	• List classes that have listen access to this event. Specify classes as `meta.class` objects in the form:
	  • A single `meta.class` object
	  • A cell array of `meta.class` objects. An empty cell array, `{}`, is the same as `private` access.
	See “Class Members Access”

Property	Purpose

`NotifyAccess`	Determines where code can trigger the event.
	• `public` — any code can trigger event
	• `protected` — can trigger event from methods in class or subclasses
	• `private` — can trigger event by class methods only (not from subclasses)
	• List classes that have notify access to this event. Specify classes as `meta.class` objects in the form:
	  • A single `meta.class` object
	  • A cell array of `meta.class` objects. An empty cell array, `{}`, is the same as `private` access.
	  See “Class Members Access”

`DefiningClass`	The `meta.class` object representing the class that defines this event.

See Also

`meta.class` | `meta.method` | `meta.property` | `metaclass`

Topics

“Events”
“Class Introspection and Metadata”

meta.MetaData class

Package: `meta`
Superclasses: `matlab.mixin.Heterogeneous`

Superclass for MATLAB object metadata

Description

The `meta.MetaData` class of objects represent MATLAB class definitions and the constituent parts of those definitions, such as properties and methods. Metadata enable a program to get information about a class definition.

The `meta.MetaData` class forms the root of the metadata class hierarchy, which enables the formation of arrays of metadata objects belonging to different specific classes.

MATLAB uses instances of the `meta.MetaData` class as the default object to fill in missing array elements.

`findobj` and `findprop`, can search the metadata hierarchy and return an array of different metadata objects. These function require the ability to form heterogeneous arrays containing various metaclass objects.

See the `matlab.mixin.Heterogeneous` class for more information on heterogeneous hierarchies.

Construction

You cannot create an instance of the `meta.MetaData` class directly. MATLAB constructs instances of this class as required.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

This example shows how the `meta.MetaData` class facilitates working with metaclasses.

Create a `meta.class` instance representing the MATLAB `timeseries` class:

```
>> mc = ?timeseries;
```

MATLAB uses `meta.MetaData` objects to fill empty array elements:

```
>> m(2) = mc
>> class(m(1))

ans =

meta.MetaData
>> class(m(2))

ans =

meta.class
```

Use `findobj` to find all properties and methods that have protected access:

```
>> protectedMembers = findobj(mc,{'Access','protected'},...
'-or',{'SetAccess','protected'},...
'-or',{'GetAccess','protected'});
```

The `timeseries` class defines both properties and methods that have protected access. Therefore, `findobj` returns a heterogeneous array of class `meta.MetaData`. This array contains both `meta.property` and `meta.method` objects.

```
>> protectedMembers

protectedMembers =

  11x1 heterogeneous meta.MetaData (meta.property, meta.method)
  handle with no properties.
  Package: meta
>> class(protectedMembers(1))

ans =

meta.property

>> protectedMembers(1).Name
```

```
ans =

Length
>> protectedMembers(1).SetAccess

ans =

protected
>> protectedMembers(1).GetAccess

ans =

public
```

See Also

`handle` | `matlab.mixin.Heterogeneous`

Topics

Class Attributes
Property Attributes
"Class Introspection and Metadata"

meta.method

Describe method of MATLAB class

Description

The `meta.method` class provides information about the methods of MATLAB classes. The read/write properties of the `meta.method` class correspond to method attributes and are specified only from within class definitions.

You can query the read-only properties of the `meta.method` object to obtain information that is specified syntactically by the class (for example, to obtain the name of the class defining a method).

You cannot instantiate a `meta.method` object directly. Obtain a `meta.method` object from the `meta.class` `MethodList` property, which contains an array of `meta.method` objects, one for each class method. For example, replace *`ClassName`* with the name of the class whose methods you want to query:

```
mco = ?ClassName;
mlist = mco.MethodList;
mlist(1).Name; % name of first method in the list
```

Use the `metaclass` function to obtain a `meta.class` object from a class instance:

```
mco = metaclass(obj);
```

Properties

Property	Purpose
`Abstract`	If `true`, the method has no implementation. The method has a syntax line that can include arguments, which subclasses use when implementing the method. • Subclasses are not required to define the same number of input and output arguments. • The method can have comments after the `function` line • Does not contain `function` or `end` keywords, only the function syntax (e.g., `[a,b] = myMethod(x,y)`)

`Access` attribute, default = `public`	Determines what code can call this method.
	• `public` — unrestricted access
	• `protected` — access from methods in class or subclasses
	• `private` — access by class methods only (not from subclasses)
	• List classes that have access to this method. Specify classes as `meta.class` objects in the form:
	  • A single `meta.class` object
	  • A cell array of `meta.class` objects. An empty cell array, `{}`, is the same as `private` access.

`DefiningClass`	The meta.class object representing the class that defines this method.
`Description` read-only	Currently not used.
`DetailedDescription` read-only	Currently not used.
`Hidden` attribute, default = `false`	When `false`, the method name shows in the list of methods displayed using the `methods` or `methodsview` commands. If set to `true`, the method name is not included in these listings.

Property	Purpose
`InputNames`	Character vector or cell array containing the names of the input arguments used in the function signature.
`Name` read-only	Name of the method.
`OutputNames`	Character vector or cell array containing the names of the output arguments used in the function signature.
`Sealed` attribute, default = `false`	If `true`, the method cannot be redefined in a subclass. Attempting to define a method with the same name in a subclass causes an error.
`Static` attribute, default = `false`	Set to `true` to define a method that does not depend on an object of the class and does not require an object argument. Call static methods using the class name in place of the object: *`classname.methodname()`* Or with an instance of the class, like any method: `o.`*`methodname()`* See “Static Methods”

See Also

`meta.class` | `meta.event` | `meta.property` | `metaclass`

Topics

“Methods”
“Class Introspection and Metadata”

meta.package

Describe MATLAB package

Description

The `meta.package` class contains information about MATLAB packages.

You cannot instantiate a `meta.package` object directly. Obtain a `meta.package` object from the `meta.class` `ContainingPackage` property, which contains a `meta.package` object, or an empty object, if the class is not in a package.

Properties

Property	Purpose
`Name`	Name of the package associated with this `meta.package` object. Read only
`Description`	Currently not used. Read only
`DetailedDescription`	Currently not used. Read only
`ClassList`	List of classes that are scoped to this package. An object array of `meta.class` objects. Read only
`Classes` Use `ClassList` instead	List of classes that are scoped to this package. A cell array of `meta.class` objects. Read only
`FunctionList`	List of functions that are scoped to this package. An object array of function handles. Read only
`Functions` Use `FunctionList` instead	List of functions that are scoped to this package. A cell array of function handles. Read only

Property	Purpose
`PackageList`	List of packages that are scoped to this package. An object array of `meta.package` objects. Read only
`Packages` Use `PackageList` instead	List of packages that are scoped to this package. A cell array of `meta.package` objects. Read only
`ContainingPackage`	A `meta.package` object describing the package within which this package is contained, or an empty object if this package is not nested. Read only

Methods

Method	Purpose
`fromName`	Static method returns a `meta.package` object for a specified package name.
`getAllPackages`	Static method returns a cell array of `meta.package` objects representing all top-level packages.

See Also

`meta.class` | `meta.event` | `meta.method` | `meta.property`

Topics

"Class Introspection and Metadata"

meta.abstractDetails

Package: `meta`

Find abstract methods and properties

Syntax

```
meta.abstractDetails(ClassName)
meta.abstractDetails(mc)
absMembers = meta.abstractDetails( ___ )
```

Description

`meta.abstractDetails(ClassName)` displays a list of abstract methods and properties for the class with name `ClassName`. Use the fully specified name for classes in packages. MATLAB displays all public and protected abstract methods and properties, including those declared `Hidden`.

`meta.abstractDetails(mc)` displays a list of abstract methods and properties for the class represented by the `meta.class` object `mc`.

`absMembers = meta.abstractDetails( ___ )` returns an array of the metaclass objects corresponding to the abstract members of the class, and can include any of the input arguments in previous syntaxes. If the class has both abstract methods and abstract properties, `absMembers` is a heterogeneous array of class `meta.MetaData` containing `meta.method` and `meta.property` objects.

A class can be abstract without defining any abstract methods or properties if it declares the `Abstract` class attribute. In this case, `meta.abstractDetails` returns no abstract members for that class, but the class is abstract. See “Determine If a Class Is Abstract” for more information.

Input Arguments

ClassName

Name of the class specified as a character vector or a string scalar.

mc

`meta.class` object representing the class (for example, `?MyClass`).

Output Arguments

absMembers

Array of `meta.class` objects representing abstract class members

Examples

Display Abstract Member Names

Define the class, `AbsBase`, with an abstract property:

```
classdef AbsBase
   properties (Abstract)
      Prop1
   end
   methods(Abstract)
      result = methodOne(obj)
      output = methodTwo(obj)
   end
end
```

Pass the class name (`AbsBase`) as a `char` vector:

```
meta.abstractDetails('AbsBase')
```

`meta.abstractDetails` displays the names of the abstract properties and methods defined in the class `AbsBase`.

```
Abstract methods for class AbsBase:
    methodTwo   % defined in AbsBase
    methodOne   % defined in AbsBase

Abstract properties for class AbsBase:
    Prop1   % defined in AbsBase
```

Return Abstract Member Metaclass Objects

Pass a `meta.class` object representing the `AbsBase` class and return the metaclass objects for the abstract members. Use the definition of the `AbsBase` class from the previous example.

```
mc = ?AbsBase;
absMembers = meta.abstractDetails(mc);
```

`absMembers` is a heterogeneous array containing a `meta.property` object for the `Prop1` abstract property and `meta.method` objects for the `methodOne` and `methodTwo` abstract methods.

List the names of the metaclass objects.

```
for k = 1:length(absMembers)
   disp(absMembers(k).Name)
end
```

```
methodTwo
methodOne
Prop1
```

Find Inherited Abstract Members

Derive the `SubAbsBase` class from `AbsBase`, which is defined in a previous example.

```
classdef SubAbsBase < AbsBase
   properties
      SubProp = 1;
   end
   methods
      function result = methodOne(obj)
         result = obj.SubProp + 1;
```

```
        end
    end
end
```

Display the names of the abstract members inherited by `SubAbsBase`.

```
meta.abstractDetails('SubAbsBase')
```

```
Abstract methods for class SubAbsBase:
    methodTwo   % defined in AbsBase

Abstract properties for class SubAbsBase:
    Prop1   % defined in AbsBase
```

To make `SubAbsBase` a concrete class, you need to implement concrete versions of `methodTwo` and `Prop1` in the subclass.

See Also

`meta.class` | `meta.class.fromName` | `string`

Topics

"Abstract Classes"
"Class Introspection and Metadata"

meta.package.fromName

Return `meta.package` object for specified package

Syntax

```
mp = meta.package.fromName(packageName)
```

Description

`mp = meta.package.fromName(packageName)` returns the `meta.package` object associated with the named package. If `packageName` is a nested package, then you must provide the fully qualified name (for example, `packageName1.packageName2`).

Input Arguments

`packageName` — Package name
character vector | string scalar

Package name, specified as a character vector or a string scalar.

Data Types: `char` | `string`

Output Arguments

`mp` — `meta.package` object
object

`meta.package` object for the specified package.

Examples

Get Package Contents

Return the names of the classes contained in the event package

```
mev = meta.package.fromName('event');
for k=1:length(mev.Classes)
   disp(mev.Classes{k}.Name)
end
```

```
event.EventData
event.ClassInstanceEvent
event.PropertyEvent
event.ParentEvent
event.ChildEvent
event.DynamicPropertyEvent
event.listener
event.proplistener
```

See Also

`meta.package` | `meta.package.getAllPackages`

Introduced in R2008a

meta.package.getAllPackages

Get all top-level packages

Syntax

```
P = meta.package.getAllPackages
```

Description

`P = meta.package.getAllPackages` is a static method that returns a cell array of `meta.package` objects representing all the top-level packages that are visible on the MATLAB path or defined as top-level built-in packages. You can access subpackages using the `Packages` property of each `meta.package` object.

Note that the time required to find all the packages on the path might be excessively long in some cases. You should therefore avoid using this method in any code where execution time is a consideration. `getAllPackages` is generally intended for interactive use only.

See Also

`meta.package` | `meta.package.fromName`

meta.property

Describe property of MATLAB class

Description

The `meta.property` class provides information about the properties of MATLAB classes. The read/write properties of the `meta.property` class correspond to property attributes and are specified only from within your class definitions.

You can query the read-only properties of the `meta.property` object to obtain information that is specified syntactically by the class (for example, to obtain the function handle of a property set access method).

You cannot instantiate a `meta.property` object directly. Obtain a `meta.property` object from the `meta.class` `PropertyList` property, which contains an array of `meta.property` objects, one for each class property. For example, replace *`ClassName`* with the name of the class whose properties you want to query:

```
mco = ?ClassName;
plist = mco.PropertyList;
plist(1).Name; % name of first property
```

Use the `metaclass` function to obtain a `meta.class` object from a class instance:

```
mco = metaclass(obj);
```

Properties

Property	Purpose
`Name` read-only	Name of the property.
`Description` read-only	Currently not used.
`DetailedDescription` read-only	Currently not used.

Property	Purpose
`AbortSet` attribute, default = `false`	If `true`, and this property belongs to a handle class, then MATLAB does not set the property value if the new value is the same as the current value. This setting prevents the triggering of property `PreSet` and `PostSet` events. See “Listen for Changes to Property Values”
`Abstract` attribute, default = `false`	If `true`, the property has no implementation, but a concrete subclass must redefine this property without `Abstract` being set to `true`. • Abstract properties cannot define set or get access methods. See “Property Access Methods” • Abstract properties cannot define initial values. “Assigning a Default Value” • All subclasses must specify the same values as the superclass for the property `SetAccess` and `GetAccess` attributes. • `Abstract=true` should be used with the class attribute `Sealed=false` (the default).
`GetAccess` attribute, default = `public`	`public` – unrestricted access `protected` – access from class or subclasses `private` – access by class members only List classes that allow access for this property. Specify classes as `meta.class` objects in the form: • A single `meta.class` object • A cell array of `meta.class` objects. An empty cell array, `{}`, is the same as `private` access. See “Class Members Access”

Property	Purpose
`SetAccess` attribute, default = `public`	`public` – unrestricted access `protected` – access from class or subclasses `private` – access by class members only `immutable` — property can be set only in the constructor. See “Mutable and Immutable Properties” List classes that have set access to this property. Specify classes as `meta.class` objects in the form: • A single `meta.class` object • A cell array of `meta.class` objects. An empty cell array, `{}`, is the same as `private` access. See “Class Members Access”
`Access`	`public` – unrestricted access `protected` – access from class or subclasses `private` – access by class members only (not subclasses) List of classes that have get and set access to this property. Specify classes as `meta.class` objects in the form: • A single `meta.class` object • A cell array of `meta.class` objects. An empty cell array, `{}`, is the same as `private` access. Use `Access` to set both `SetAccess` and `GetAccess` to the same value. Query the values of `SetAccess` and `GetAccess` directly (not `Access`).

Property	Purpose
`Constant` attribute, default = `false`	Set to `true` if you want only one value for this property in all instances of the class. • Subclasses inherit constant properties, but cannot change them. • `Constant` properties cannot be `Dependent` • `SetAccess` is ignored. See “Define Class Properties with Constant Values”
`DefaultValue`	Property default value (if specified in class definition). See also `HasDefault` property. Abstract, dependent, and dynamic properties cannot specify default values.
`DefiningClass`	The `meta.class` object representing the class that defines this property.
`Dependent` attribute, default = `false`	If `false`, property value is stored in object. If `true`, the property value is not stored in the object and the set and get functions cannot access the property by indexing into the object using the property name. See “Property Get Methods”
`Transient`attribute, default = `false`	If `true`, property value is not saved when object is saved to a file. See “Save and Load Process for Objects” for more about saving objects.
`GetMethod` read-only	Function handle of the get method associated with this property. Empty if there is no get method specified. See “Property Get Methods”
`GetObservable` attribute, default = `false`	If `true`, and it is a handle class property, then listeners can be created for access to this property. The listeners are called whenever property values are queried. See “Property-Set and Query Events”
`HasDefault`	Property contains a boolean value indicating whether the property defines a default value. Test `HasDefault` before querying the `DefaultValue` property to avoid a `MATLAB:class:NoDefaultDefined` error.

Property	Purpose
`Hidden` attribute, default = `false`	Determines whether the property should be shown in a property list (e.g., Property Inspector, call to `properties`, etc.).
`NonCopyable` attribute, default = `false`	Indicate if property value should be copied when object is copied. For more information, see “Exclude Properties from Copy”
`SetMethod` read-only	Function handle of the set method associated with this property. Empty if there is no set method specified. See “Property Set Methods”
`SetObservable` attribute, default = `false`	If `true`, and it is a handle class property, then listeners can be created for access to this property. The listeners are called whenever property values are modified. See “Property-Set and Query Events”
`Validation`	`meta.Validation` object describing the validation defined by this property.

Events

See “Listen for Changes to Property Values” for information on using property events.

Event Name	Purpose
`PreGet`	Event occurs just before property is queried.
`PostGet`	Event occurs just after property has been queried
`PreSet`	Event occurs just before this property is modified
`PostSet`	Event occurs just after this property has been modified

See Also

`meta.Validation` | `meta.class` | `meta.event` | `meta.method` | `metaclass`

Topics

“Properties”

"Class Introspection and Metadata"

Introduced in R2008a

metaclass

Obtain `meta.class` object

Syntax

```
mc = metaclass(object)
mc = ?ClassName
```

Description

`mc = metaclass(object)` returns the `meta.class` object for the class of `object`. The `object` input argument can be a scalar or an array of objects. However, `metaclass` always returns a scalar `meta.class` object.

`mc = ?ClassName` returns the `meta.class` object for the class with name, *`ClassName`*. The `?` operator works only with a class name, not an object.

If you pass a class name as a `char` vector to the `metaclass` function, it returns the `meta.class` object for the `char` class. Use the `?` operator or the `meta.class.fromName` method to obtain the `meta.class` object from a class name. Use this method if you want to pass the class name in a `char` variable.

Examples

Return the `meta.class` object for an instance of the `MException` class:

```
obj = MException('Msg:ID','MsgTxt');
mc = metaclass(obj);
```

Use the `?` operator to get the `meta.class` object for the `matlab.mixin.SetGet` class:

```
mc = ?matlab.mixin.SetGet;
```

See Also

`meta.class` | `meta.class.fromName`

Topics

"Class Metadata"

meta.ArrayDimemsion class

Package: `meta`
Superclasses: `matlab.mixin.Heterogeneous`

Size information for property validation

Description

`meta.ArrayDimension` is an abstract class used to define subclasses that represent the size information defined by property validation. These subclasses enable a complete description of property size information:

- `meta.FixedDimension` - Dimension specified as a fixed numeric value
- `meta.UnrestrictedDimension` - Dimension specified as a colon, which indicates any value is allowed

Attributes

`Abstract`	`true`

For information on class attributes, see “Class Attributes”.

See Also

`meta.FixedDimension` | `meta.UnrestrictedDimension` | `meta.Validation`

Topics

“Metadata Interface to Property Validation”

Introduced in R2018a

meta.FixedDimension class

Package: `meta`
Superclasses: `meta.ArrayDimension`

Fixed dimension in property size specification

Description

The `meta.Validation` class `Size` property uses `meta.FixedDimension` objects to represent the fixed values in a property size specification. The object Length property contains the numeric value of the dimension.

Properties

`Length` — Scalar fixed dimension
1 (default)

Fixed dimension, specified as scalar `uint64`.

Attributes:

`GetAccess`	`public`
`SetAccess`	`private`

Data Types: `uint64`

Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Examples

Find the fixed dimensions in a property validation.

The `ValidationExample` class specifies the size of the property value as `(1,:)`.

```
classdef ValidationExample
   properties
      Prop (1,:) double {mustBeReal, mustBeGreaterThan(Prop, 10)} = 200;
   end
end
```

Read the dimensions into a cell array.

```
mc = ?ValidationExample;
mp = findobj(mc.PropertyList,'Name','Prop');
sz = mp.Validation.Size;
len = length(sz);
dim = cell(1:len);
for k = 1:len
   switch class(sz(k))
      case 'meta.FixedDimension'
         dim{k} = sz(k).Length;
      case 'meta.UnrestrictedDimension'
         dim{k} = ':';
   end
end
```

See Also

`meta.ArrayDimension` | `meta.UnrestrictedDimension` | `meta.Validation`

Topics

"Metadata Interface to Property Validation"

Introduced in R2018a

meta.UnrestrictedDimension class

Package: `meta`
Superclasses: `meta.ArrayDimension`

Unrestricted dimension in property size specification

Description

The `meta.Validation` class `Size` property uses `meta.UnrestrictedDimension` objects to represent the use of a colon in a property size specification. An instance of this class in the `Size` array indicates that the respective dimension in the property definition is a colon. A colon in a size specification indicates that any values is allowed.

Attributes

`Sealed`	`true`

For information on class attributes, see “Class Attributes”.

Examples

Find the fixed dimensions in a property validation.

The `ValidationExample` class specifies the size of the property value as `(1,:)`.

```
classdef ValidationExample
   properties
      Prop (1,:) double {mustBeReal, mustBeGreaterThan(Prop, 10)} = 200;
   end
end
```

Read the dimensions into a cell array.

```
mc = ?ValidationExample;
mp = findobj(mc.PropertyList,'Name','Prop');
```

```
sz = mp.Validation.Size;
len = length(sz);
dim = cell(1:len);
for k = 1:len
   switch class(sz(k))
      case 'meta.FixedDimension'
         dim{k} = sz(k).Length;
      case 'meta.UnrestrictedDimension'
         dim{k} = ':';
   end
end
```

See Also

`meta.ArrayDimension` | `meta.FixedDimension` | `meta.Validation`

Topics

"Metadata Interface to Property Validation"

Introduced in R2018a

meta.Validation class

Package: `meta`
Superclasses: `handle`

Describes property validation

Description

Instances of this class contain information about property validation that is specified in a class definition. The `meta.Validation` class enables you to obtain the following information programmatically for each property in a class definition:

- Class restriction applied to the property
- Size requirements of the property value
- Function handles referencing validation functions applied to property values

For information on property validation, see “Validate Property Values”.

Properties

`Class` — Class restriction applied to property

`meta.class` object

Class restriction applied to property, specified as a `meta.class` object. If the property definition does not contain class restriction, MATLAB sets this property to a 0-by-0 meta.class object.

Attributes:

`GetAccess`	`public`
`SetAccess`	`private`

`Size` — Dimensions of the property value

empty `meta.ArrayDimension` (default) | heterogeneous array of type `meta.ArrayDimension` | `meta.FixedDimemsion` | `meta.UnrestrictedDimension`

Dimensions of the property value, specified as a heterogeneous array of type `meta.ArrayDimension` or arrays of type `meta.FixedDimension` or `meta.UnrestrictedDimension`. If the property definition does not specify dimensions for the property, MATLAB sets this property to a 1-by-0 `meta.ArrayDimension` array.

Attributes:

GetAccess	public
SetAccess	private

`ValidationFunctions` — Validation functions
cell array of function handles

Validation functions, specified as a cell array of function handles referencing each validation function. If the property does not use validation functions, MATLAB sets this property to a 1-by-0 cell array.

Attributes:

GetAccess	public
SetAccess	private

Attributes

Sealed	true
HandleCompatible	true

For information on class attributes, see “Class Attributes”.

Methods

isValidValue

```
tf = isValidValue(metaValidationObj,value)
```

Determine if value is valid. This method returns `true` if `value` is a valid value for the property whose validation is describe by `metaValidationObj`.

Input Arguments

- `metaValidationObj` - The `meta.Validation` object for the property
- `value` - The potential property value to test for validity

Return Value

- `true` - Value is valid for this property
- `false` - Value is not a valid value for this property

validateValue

`validateValue(metaValidationObj,value)`

Test if value is valid and throw error if it is not. This method throws an error if `value` is not a valid value for the property whose validation is describe by `metaValidationObj`. The error message is the same as that thrown if the value is assigned to the property of an actual object.

Input Arguments

- `metaValidationObj` - The `meta.Validation` object for the property
- `value` - The potential property value to test for validity

Return Value

none

Examples

The `ValidationExample` class defines a property that used validation.

```
classdef ValidationExample
   properties
      Prop (1,:) double {mustBeReal, mustBeGreaterThan(Prop, 10)} = 200;
   end
end
```

The `getErrorMessage` function determines if a potential value is valid and displays the error message indicating the cause of invalid values.

```
function getErrorMessage(possibleValue)
   mc = ?ValidationExample;
   mp = findobj(mc.PropertyList,'Name','Prop');
   mv = mp.Validation;
   if ~mv.isValidValue(possibleValue)
      try
         mv.validateValue(possibleValue)
      catch errorMessage
         fprintf('This value is not valid because: %s\n',...
            errorMessage.message);
      end
   else
      fprintf('%d is OK\n',possibleValue)
   end
end
```

See Also

`meta.FixedDimension` | `meta.UnrestrictedDimension` | `meta.property`

Topics

"Metadata Interface to Property Validation"

Introduced in R2018a

methods

Class method names

Syntax

```
methods ClassName
methods(obj)
methods( ___ ,'-full')
m = methods( ___ )
```

Description

`methods ClassName` displays the names of the methods for the class `ClassName`. If `ClassName` is a MATLAB or Java class, then `methods` displays only non-hidden, public methods, including those methods inherited from superclasses.

`methods(obj)` displays the names of the methods for the class of `obj`.

`methods( ___ ,'-full')` displays a full description of the methods, including inheritance information and, for MATLAB and Java methods, method attributes and signatures. This function does not remove duplicate method names with different signatures. This option does not work with classes defined before MATLAB 7.6.

`m = methods( ___ )` returns the method names in a cell array of character vectors.

Examples

List Class Methods

Display the public methods of the MException class

```
methods MException
```

```
Methods for class MException:
```

```
addCause       eq             isequal        rethrow        throwAsCaller
addCorrection  getReport      ne             throw

Static methods:

last
```

Get Method Names from Object

Construct a java.lang.String object and display the names of the public methods of that object.

```
s = java.lang.String;
methods(s);
```

```
Methods for class java.lang.String:

String                format                replace
charAt                getBytes              replaceAll
chars                 getChars              replaceFirst
codePointAt           getClass              split
codePointBefore       hashCode              startsWith
codePointCount        indexOf               subSequence
codePoints            intern                substring
compareTo             isEmpty               toCharArray
compareToIgnoreCase   join                  toLowerCase
concat                lastIndexOf           toString
contains              length                toUpperCase
contentEquals         matches               trim
copyValueOf           notify                valueOf
endsWith              notifyAll             wait
equals                offsetByCodePoints
equalsIgnoreCase      regionMatches
```

Show method signatures

List the public methods of the MException class and show method signatures.

```
methods('MException','-full')
```

```
Methods for class MException:

MException scalar lhs1 addCause(MException scalar rhs1, MException scalar rhs2)
varargout addCorrection(MException rhs1, Correction)
logical scalar lhs1 eq(MException scalar rhs1, MException scalar rhs2)
logical lhs1 eq(MException rhs1, MException rhs2)
logical lhs1 eq(rhs1, rhs2)
unicodeString lhs1 getReport(MException scalar rhs1, asciiString rhs2, rhs3)
logical scalar lhs1 isequal(MException scalar rhs1, MException scalar rhs2)
logical scalar lhs1 isequal(MException rhs1, MException rhs2)
logical scalar lhs1 isequal(rhs1, rhs2)
Static MException scalar lhs1 last(asciiString rhs1)
logical scalar lhs1 ne(MException scalar rhs1, MException scalar rhs2)
logical lhs1 ne(MException rhs1, MException rhs2)
logical lhs1 ne(rhs1, rhs2)
rethrow(MException scalar rhs1)
throw(MException scalar rhs1)
throwAsCaller(MException scalar rhs1)
```

Store Method Names in Cell Array

Store the names of the public methods of the MException class in a cell array. Include the method signatures using the -full option.

```
m = methods('MException','-full');
```

Input Arguments

`ClassName` — Class name
character vector | string scalar

Class name, specified as a character vector or string scalar.

Data Types: `char` | `string`

`'-full'` — Display full description
'-full'

Display full description of methods, including input and output arguments

Data Types: `char`

Output Arguments

`m` — Method names
cell array

Method names returned as a cell array of character vectors.

More About

Language Dependency

This function does not show generic methods from classes based on the Microsoft .NET Framework. Use your product documentation to get information on generic methods.

Methods Keyword

The word `methods` is also a MATLAB class-definition keyword. See `classdef` for more information on class-definition keywords.

Listed Methods

This function reports the methods from all method directories together and removes all duplicate method names from the list.

See Also

`methodsview` | `properties`

Topics

"Methods"

Introduced before R2006a

methodsview

View class methods

Syntax

```
methodsview(packagename.classname)
methodsview(object)
```

Description

`methodsview(packagename.classname)` displays information about the methods in the class `classname`. If the class is in a package, include `packagename`. If `classname` is a MATLAB or Java class, `methodsview` lists only public methods, including those methods inherited from superclasses.

`methodsview` creates a window that displays the methods and information such as arguments, returned values, and superclasses. It also includes method qualifiers (for example, `abstract` or `synchronized`) and possible exceptions thrown.

`methodsview(object)` displays information for the class of `object`.

Examples

Display Methods of Java Class

List information on all methods in the `java.awt.MenuItem` class. MATLAB displays this information in a new window.

```
methodsview('java.awt.MenuItem')
```

Input Arguments

`packagename.classname` — Fully qualified class name
string | character vector

Fully qualified class name, specified as a string or a character vector.

`object` — Object name
string | character vector

Object name, specified as a string or a character vector.

See Also

`class` | `import` | `javaArray` | `methods`

Introduced before R2006a

mex

Build MEX function or engine application

Syntax

```
mex filenames
mex filenames api option1 ... optionN

mex -client engine filenames
mex -client engine filenames api option1 ... optionN

mex -setup [lang]
mex -setup -client engine [lang]
```

Description

`mex filenames` compiles and links one or more C++ source files written with the "MATLAB Data API" into a binary MEX file in the current folder. For information about writing these applications, see "C++ MEX Applications".

If writing MEX files based on the "C Matrix API" or the "Fortran Matrix API", then `mex` *`filenames`* builds one or more C, C++, or Fortran source files with the `-R2017b` `api`. In a future version of MATLAB, the default `api` option will change to use the interleaved complex API (`-R2018a`). MathWorks recommends that you create MEX files and update existing MEX files to use the interleaved complex API. Alternatively, use the `MX_HAS_INTERLEAVED_COMPLEX` macro to ensure the desired behavior across versions of MATLAB. For more information, see "MATLAB Support for Interleaved Complex API in MEX Functions".

For information about working with C, C++, and Fortran applications, see "Integrate MATLAB with External Programming Languages and Systems".

`mex filenames api option1 ... optionN` builds with the specified `api` and optional `option1 ... optionN` arguments. The `option1 ... optionN` arguments supplement or override the default `mex` build configuration.

`mex -client engine filenames` builds C++ source files written with the "MATLAB Data API" into standalone MATLAB engine applications. For more information, see "Introduction to Engine API for C++"

If writing applications based on the "MATLAB Engine API for C", the "MATLAB C API to Read MAT-File Data", the "MATLAB Engine API for Fortran", or the "MATLAB Fortran API to Read MAT-File Data", then `mex -client engine filenames` builds a standalone application with the `-R2017b` `api`. In a future version of MATLAB, the default `api` option will change to use the interleaved complex API (`-R2018a`). MathWorks recommends that you create engine applications and update existing applications to use the interleaved complex API.

`mex -client engine filenames api option1 ... optionN` builds engine applications with the specified `api` and optional `option1 ... optionN` arguments.

`mex -setup [lang]` selects a compiler for the given `lang` for building MEX files. MATLAB defines a default compiler for each supported language. If you have multiple compilers for a given language, use the `lang` option to change the default compiler for that language. For more information, see "Change Default Compiler" and "Choose a C++ Compiler".

`mex -setup -client engine [lang]` selects a compiler for building engine applications.

Examples

Build MEX File Using Interleaved Complex API

Copy the source code example from the *`matlabroot`*`/extern/examples` folder.

```
copyfile(fullfile(matlabroot,'extern','examples','mex','explore.c'),'.','f')
```

Build the MEX file. The output displays information specific to your compiler.

```
mex -R2018a explore.c
```

Test.

```
a = [1 3 5];
b = [5 3 1];
```

```
A = complex(a,b);
explore(A)
```

```
-----------------------------------------------
Name: prhs[0]
Dimensions: 1x3
Class Name: double
-----------------------------------------------
    (1,1) = 1 + 5i
    (1,2) = 3 + 3i
    (1,3) = 5 + 1i
```

Build C MEX File

Build a single C program `yprime.c` into a MEX file.

Create a writable folder on your path `c:\work` and set it as the current folder.

```
[s,msg,msgid] = mkdir('c:\work');
if (isempty(msgid))
    mkdir('c:\work')
end
cd c:\work
```

Copy the source code example from the *`matlabroot`*`/extern/examples` folder.

```
copyfile(fullfile(matlabroot,'extern','examples','mex','yprime.c'),'.','f')
```

Build the MEX file. The output displays information specific to your compiler.

```
mex yprime.c
```

```
Building with 'MinGW64 Compiler  C '.
MEX completed successfully.
```

Test.

```
T=1;
Y=1:4;
yprime(T,Y)
```

```
ans =
```

```
    2.0000    8.9685    4.0000   -1.0947
```

Display Detailed Build and Troubleshooting Information

To display the compile and link commands and other information useful for troubleshooting, use verbose mode.

```
mex -v -compatibleArrayDims yprime.c
```

The output displays information specific to your platform and compiler.

Override Default Compiler Switch Option

Build the `yprime.c` MEX file by appending the value `-Wall` to the existing compiler flag. Because the value includes a space character, you must delineate the string; the character you use depends on the platform.

At the MATLAB prompt, use MATLAB single quotes (`'`).

```
mex -v COMPFLAGS='$COMPFLAGS -Wall' yprime.c
```

For the MinGW-w64 compiler, which is based on gcc/g++, use the Linux compiler flags.

```
mex -v CXXFLAGS='$CXXFLAGS -Wall' yprime.c
```

At the Windows command prompt, use double quotes (`"`).

```
mex -v COMPFLAGS="$COMPFLAGS -Wall" yprime.c
```

At the shell command line on macOS and Linux, use single quotes (`'`).

```
mex -v CFLAGS='$CFLAGS -Wall' yprime.c
```

Build MEX File from Multiple Source Files

The MEX file example `fulltosparse` consists of two Fortran source files, `loadsparse.F` and `fulltosparse.F`. To run this example, you need a supported Fortran compiler installed on your system.

Copy the source files to the current folder.

```
copyfile(fullfile(matlabroot,'extern','examples','refbook','loadsparse.F'),'.','f')
copyfile(fullfile(matlabroot,'extern','examples','refbook','fulltosparse.F'),'.','f')
```

Build the `fulltosparse` MEX file. The MEX file name is `fulltosparse` because `fulltosparse.F` is the first file on the command line. The output contains information specific to your compiler.

```
mex -largeArrayDims fulltosparse.F loadsparse.F
```

```
Building with 'Intel Visual Fortran Composer XE 2013 with Microsoft Visual Studio 2012
MEX completed successfully.
```

Test.

```
full = eye(5);
spar = fulltosparse(full)
```

```
spar =

    1,1         1
    2,2         1
    3,3         1
    4,4         1
    5,5         1
```

Combine Source Files Using Wild Card

Combine all C source files in the current folder into MEX file `mymex`. Use the `-output` option to control the name of the MEX file.

```
mex -output mymex *.c
```

Preview Build Commands

To preview the build command details without executing the commands, use the `-n` option. The output contains information specific to your platform and compiler.

```
mex -n yprime.c
```

Create and Link to Separate Object Files

You can link to object files that you compile separately from your source MEX files.

The MEX file example fulltosparse consists of two Fortran source files. The `fulltosparse` file is the gateway routine (contains the `mexFunction` subroutine) and `loadsparse` contains the computational routine.

To run this example, you need a supported Fortran compiler installed on your system. Copy the computational subroutine to your current folder.

```
copyfile(fullfile(matlabroot,'extern','examples','refbook','loadsparse.F'),'.','f')
```

Compile the subroutine and place the object file in a separate folder, `c:\objfiles`.

```
mkdir c:\objfiles
mex -largeArrayDims -c -outdir c:\objfiles loadsparse.F
```

```
Building with 'Intel Visual Fortran Composer XE 2013 with Microsoft Visual Studio 2012
MEX completed successfully.
```

Copy the gateway subroutine to your current folder. Compile and link with the `loadsparse` object file.

```
copyfile(fullfile(matlabroot,'extern','examples','refbook','fulltosparse.F'),'.','f')
mex -largeArrayDims fulltosparse.F c:\objfiles\loadsparse.obj
```

```
Building with 'Intel Visual Fortran Composer XE 2013 with Microsoft Visual Studio 2012
MEX completed successfully.
```

Specify Path to Include File

To specify the path to include the MATLAB LAPACK library subroutines for handling complex number routines, use the `-I` option. To use these subroutines, your MEX file must access the header file `fort.h`.

Copy the `matrixDivideComplex.c` example to the current folder.

```
copyfile(fullfile(matlabroot,'extern','examples','refbook','matrixDivideComplex.c'),'.
```

Create the `-I` argument by concatenating `'-I'` with the path to `fort.h` file.

```
ipath = ['-I' fullfile(matlabroot,'extern','examples','refbook')];
```

Create variables for the names and paths to the LAPACK library file and the file, `fort.c`, containing the complex number handling routines.

```
lapacklib = fullfile(matlabroot,'extern','lib',computer('arch'),'microsoft','libmwlapac
fortfile = fullfile(matlabroot,'extern','examples','refbook','fort.c');
```

Build the MEX file.

```
mex('-v','-R2017b',ipath,'matrixDivideComplex.c',fortfile,lapacklib)
```

Specify Path to Library File

Build the `matrixDivide.c` example on a Windows platform using the `-L` and `-l` options to specify the `libmwlapack.lib` library. The library file is located in the folder, *`matlabroot`*`\extern\lib\`*`arch`*`\microsoft`.

Copy the `matrixDivide.c` example to the current folder.

```
copyfile(fullfile(matlabroot,'extern','examples','refbook','matrixDivide.c'),'.','f')
```

Capture the value of *`matlabroot`*.

```
matlabroot
```

```
ans =

C:\Program Files\MATLAB\R2014a
```

Capture the value of *`arch`*.

```
computer('arch')
```

```
ans =

win64
```

To build the MEX file, copy the values of *`matlabroot`* and *`arch`* into the `mex` command, as shown in the following statement.

```
mex '-LC:\Program Files\MATLAB\R2014a\extern\lib\win64\microsoft'...
   -llibmwlapack matrixDivide.c
```

You must use the `'` characters because `\Program Files` in the path includes a space.

Define Compiler Directive

The `mxcreatecharmatrixfromstr.c` example uses a `#define` symbol `SPACE_PADDING` to determine what character to use between character vectors in a matrix. To set the value, build the MEX file with the `-D` option.

Copy the example to the current folder.

```
copyfile(fullfile(matlabroot,'extern','examples','mx','mxcreatecharmatrixfromstr.c'),'
```

Set the `SPACE_PADDING` directive to add a space between values.

```
mex mxcreatecharmatrixfromstr.c -DSPACE_PADDING
```

```
Building with 'MinGW64 Compiler  C '.
MEX completed successfully.
```

Build Engine Application

Copy the `engwindemo.c` engine example to the current folder.

```
copyfile(fullfile(matlabroot,'extern','examples','eng_mat','engwindemo.c'),'.','f')
```

```
mex -client engine engwindemo.c
```

If you are running on a Windows platform, you must first register MATLAB as a COM server. For more information, see “Register MATLAB as a COM Server”.

Run the example.

```
!engwindemo
```

Select C Compiler

```
mex -setup
```

MATLAB displays the options for your version and system based on the list of Supported and Compatible Compilers.

Input Arguments

`filenames` — One or more file names

string | character vector

One or more file names, including name and file extension, specified as a string or a character vector. If the file is not in the current folder, specify the full path to the file.

File names can be any combination of:

- C, C++, or Fortran language source files
- Simulink S-function files
- Object files
- Static library files. `filenames` must include the fully qualified path to the library file. To link dynamic libraries, use the `-l`*`libname`* option.

The first source code file listed in `filenames` is the name of the binary MEX file or engine application. To override this naming convention, use the `'-output'` option.

Use the MATLAB Editor to write your source code. If you use an integrated development environment (IDE) such as Microsoft Visual Studio® or Xcode, then you can use the `mex` command or follow the guidelines in “Custom Build with MEX Script Options”.

MATLAB automatically selects a compiler, if installed, based on the language of the `filenames` arguments.

Data Types: `char`

`api` — Release-specific API

`-R2017b` (default) | `-R2018a` | `-largeArrayDims` | `-compatibleArrayDims`

Links with the release-specific “C Matrix API” or “Fortran Matrix API”, specified as one of these values. Do not combine these options.

Do not use this option for MEX files or engine applications using the “MATLAB Data API”.

API	Description
`-R2017b` (default)	Builds with: • Separate complex API, which contains the C and Fortran Matrix API functionality in MATLAB R2017b and earlier. • Large-array-handling API, which handles arrays with more than 2^{31}-1 elements. • Treats a handle to a graphics object as `object`, not `double`. In a future version of MATLAB, the default `api` option will change to use the interleaved complex API (`-R2018a`). MathWorks recommends that you create applications and update existing applications to use the interleaved complex API. Alternatively, use the `MX_HAS_INTERLEAVED_COMPLEX` macro to ensure the desired behavior across versions of MATLAB. For more information, see “MATLAB Support for Interleaved Complex API in MEX Functions”.
`-R2018a`	Builds with: • Interleaved complex API, which includes the typed data access functions. For more information, see “Upgrade MEX Files to Use Interleaved Complex API”. • Large-array-handling API • Treats a handle to a graphics object as `object`, not `double`. To run a Fortran MEX file built with the interleaved complex API in MATLAB R2018a, you must use MATLAB R2018a Update 3.

API	Description
`-largeArrayDims`	Builds with: • Separate complex API • Large-array-handling API • Treats a handle to a graphics object as `object`, not `double`. To treat the handle as `double`, combine this option with `-DMEX_DOUBLE_HANDLE`.
`-compatibleArrayDims`	Builds with: • Separate complex API • Version 7.2 array-handling API, which limits arrays to 2^{31}-1 elements • Treats a handle to a graphics object as `object`, not `double`. To treat the handle as `double`, combine this option with `-DMEX_DOUBLE_HANDLE`. Do not use the `-compatibleArrayDims` option when calling LAPACK or BLAS functions. Default option for C MEX S-functions only.

Example: `mex -R2018a explore.c`

`option1 ... optionN` — Optional build options
strings or character vectors corresponding to valid option flags

Optional build options, specified as one of these values. Options can appear in any order on any platform, except where indicated.

Option	Description
`@`*`rspfile`*	Uses Windows RSP file. An RSP file is a text file containing command-line options. Non-ASCII characters are not supported.
`-c`	Compiles an object file only. Does not build a binary MEX file.
`-client engine`	Build engine application.

Option	Description

`-Dsymbolname`
`-Dsymbolname=symbolvalue`
`-Usymbolname`
	The `-D` options define C preprocessor macros. Equivalent to the following in the source file:
	• `#define symbolname`
	• `#define symbolname symbolvalue`
	
	The `-U` option removes any initial definition of the C preprocessor macro, `symbolname`. Inverse of the `-D` option.
	
	Do not add a space between **D** or **U** and *`symbolname`*. Do not add spaces around the **=** sign.
	
	Example: “Define Compiler Directive” on page 1-9741

`-f filepath`	Overrides the default compiler selection. *`filepath`* is the name and full path of the configuration file, specified as a string or a character vector. For information about using a non-default compiler, see “Change Default Compiler”. Do not use the `-f` option to build engine applications. Use the `-client engine` option instead.
`-g`	Adds symbolic information and disables optimizing built object code. Use for debugging. To debug with optimization, add the `-O` option.
`-h[elp]`	Displays help for `mex`. Use from an operating system prompt.
`-Ipathname`	Adds *`pathname`* to the list of folders to search for `#include` files. Do not add a space between **I** and *`pathname`*. Example: “Specify Path to Include File” on page 1-9739

Option	Description
`-l`*`libname`* `-L`*`libfolder`* `-l`*`libname`*	Links with dynamic object library *`libname`* in (optional) *`libfolder`*. MATLAB expands *`libname`* to: • *`libname`*`.lib` or `lib`*`libname`*`.lib` — Windows systems • `lib`*`libname`*`.dylib` — macOS systems • `lib`*`libname`*`.so` — Linux systems If used, the `-L` option must precede the `-l` option. When using the `-L` option on Linux or macOS systems, you also must set the runtime library path, as explained in "Set Run-Time Library Path". Do not add a space between **l** and *`libname`* or between **L** and *`libfolder`*. Specify the `-l` option with the lowercase letter **L**. To link a static library, use the `filenames` input argument on page 1-9742. Example: "Specify Path to Library File" on page 1-9740
`-n`	Displays, but does not execute, commands that `mex` would execute. Example: "Preview Build Commands" on page 1-9738
`-O`	Optimizes the object code. Use this option to compile with optimization. Optimization is enabled by default. Optimization is disabled when the `-g` option appears without the `-O` option. Specify this option with the capital letter **O**.
`-outdir` *`dirname`*	Places all output files in folder *`dirname`*. Example: "Create and Link to Separate Object Files" on page 1-9739

Option	Description
`-output` *`mexname`*	Overrides the default MEX file naming mechanism. Creates binary MEX file named *`mexname`* with the appropriate MEX file extension. Example: “Combine Source Files Using Wild Card” on page 1-9738
`-setup` *`lang`*	Change the default compiler to build *`lang`* language MEX files or engine applications. When you use this option, `mex` ignores all other command-line options.
`-silent`	Suppresses informational messages. The `mex` command still reports errors and warnings, even when you specify `-silent`.
`-U`*`symbolname`*	Removes any initial definition of the C preprocessor macro *`symbolname`*. (Inverse of the `-D` option.) Do not add a space between **U** and *`symbolname`*.
`-v`	Builds in verbose mode. Displays values for internal variables after all command-line arguments are considered. Displays each compile and link step fully evaluated. Use for troubleshooting compiler setup problems. Example: “Display Detailed Build and Troubleshooting Information” on page 1-9737
`varname=varvalue`	Overrides default setting for variable *`varname`*. This option is processed after all command-line arguments are considered. Example: “Override Default Compiler Switch Option” on page 1-9737.

`lang` — Language

`C` (default) | `C++` | `CPP` | `Fortran`

Language, specified as one of these values.

`C`	C compilers, including C++
`C++` or `CPP`	C++ compilers

`Fortran`	Fortran compilers

Tips

- You can run `mex` from:
  - MATLAB Command Window
  - Windows system prompt
  - macOS Terminal
  - Linux shell

  For command-line usage outside of MATLAB, the `mex` program is located in the folder specified by `[matlabroot '/bin']` on UNIX and `[matlabroot '\bin\win64']` on Windows.
- The MEX file has a platform-dependent extension. You can place binary MEX files for different platforms in the same folder. To identify the MEX file extension, use the `mexext` function.

  MEX File Platform-Dependent Extension

Platform	Binary MEX File Extension
Linux (64-bit)	`mexa64`
Apple Mac (64-bit)	`mexmaci64`
Windows (64-bit)	`mexw64`
- To use `mex` to build executable files for standalone MATLAB engine applications, use the `-client engine` option.
- The `mex` command does not support folder names containing double quote (`"`) characters.

See Also

`clear` | `computer` | `dbmex` | `inmem` | `loadlibrary` | `mex.getCompilerConfigurations` | `mexext` | `pcode` | `prefdir` | `system`

Topics

"MATLAB Data API"

“C Matrix API”
“Fortran Matrix API”
“Tables of MEX Function Source Code Examples”
“Build C MEX Function”
“Build C++ MEX Programs”
“Change Default Compiler”
“Upgrade MEX Files to Use Interleaved Complex API”
“Upgrade MEX Files to Use 64-Bit API”
“Upgrade MEX Files to Use Graphics Objects”

External Websites

Supported and Compatible Compilers

Introduced before R2006a

mexhost

Create host process for C++ MEX function

Syntax

```
mh = mexhost
mh = mexhost("EnvironmentVariables",envVariables)
```

Description

`mh = mexhost` creates a MEX host process that is used to run C++ MEX functions. The default process name is `MATLABMexHost`.

Use the `feval` method of the `matlab.mex.MexHost` object returned to execute C++ MEX functions in the MEX host process.

`mh = mexhost("EnvironmentVariables",envVariables)` sets environment variable with values defined in `envVariables` for the process.

Examples

Run C++ MEX Function Out of Process

Create a host process and run a C++ MEX function in that process.

The `arrayProduct.cpp` C++ MEX file contains the source code for a function that multiplies an array by a scalar input and returns the resulting array. Open this file and save it on your MATLAB path. Build the C++ MEX source file using the `mex` command. To set up the MEX build, follow the instructions in “Build C++ MEX Programs”.

```
mex arrayProduct.cpp
```

Create a host process. The `mexhost` function returns a `matlab.mex.MexHost` object.

```
mh = mexhost;
```

Use the `feval` method of the `matlab.mex.MexHost` object to evaluate the C++ MEX function in the host process.

```
result = feval(mh,"arrayProduct",10,[2,4,6,8])
```

```
result =

    20    40    60    80
```

Find Process ID

You can use the `MexHost` object to find the identifier of the process created by the `mexhost` function.

```
mh = mexhost;
mh.ProcessIdentifier
```

```
ans =
    "13336"
```

Each call to `mexhost` creates a process.

Set Environment Variable for Library

Set the value of environment variable `envName1` to `envVal1` and the value of variable `envName2` to `envVal2`.

```
s = ["envName1","envVal1"
     "envName2","envVal2"];
mh = mexhost("EnvironmentVariables",s)
```

```
mh =

  MexHost with properties:

             ProcessName: "MATLABMexHost"
       ProcessIdentifier: "19344"
               Functions: [0×0 string]
```

```
    EnvironmentVariables: "envName1"    "envVal1"
                          "envName2"    "envVal2"
```

Input Arguments

envVariables — Environment variables and values
n-by-2 string array

Environment variables and values, specified as an n-by-2 string array. The first column is the name of the environment variable and the second column is the value.

Output Arguments

mh — Host process
`matlab.mex.MexHost` object

Host process, returned as a `matlab.mex.MexHost` object. Use this process to run a C++ MEX function outside of the MATLAB process.

More About

Process Life Cycle

MATLAB terminates the process when the object returned by `mexhost` is destroyed. MATLAB destroys the object when any of the following occur.

- The MEX host variable returned by `mexhost` goes out of scope, causing MATLAB to destroy the object.
- The `delete` method is called on the MEX host variable explicitly.
- The `clear` function is called on the MEX host variable and there are no other references to the object.
- The `clear` function is called with any of these options: `clear java`, `clear classes`, or `clear all`.

Unload C++ MEX Functions

To unload all C++ MEX functions from their host processes, call `clear mex` or `clear functions`. To unload a specific C++ MEX function all host processes running it, call `clear` on the function name. For more information, see the `clear` function and "Out-of-Process Execution of C++ MEX Functions".

See Also

`clear` | `feval` | `matlab.mex.MexHost`

Topics

"Out-of-Process Execution of C++ MEX Functions"

Introduced in R2019a

matlab.mex.MexHost class

Package: `matlab.mex`
Superclasses: `handle`

Out-of-process host for C++ MEX function execution

Description

Instances of the `matlab.mex.MexHost` class represent processes created to execute C++ MEX functions. You can execute a C++ MEX function from MATLAB in the host process using the `feval` method of this class. For information on running C++ MEX functions out of process, see “Out-of-Process Execution of C++ MEX Functions”.

The `matlab.mex.MexHost` class is a `handle` class.

Creation

Use the `mexhost` function to create a `matlab.mex.MexHost` object.

Properties

`EnvironmentVariables` — Process environment variables
string array

Process environment variables, returned as a string array.

Attributes:

`GetAccess`	`public`
`SetAccess`	`private`

Data Types: `string`

`Functions` — Names of C++ MEX functions loaded in the host process
string vector

Names of the C++ MEX functions loaded in the host process, returned as a string vector.

Attributes:

```
GetAccess                                   public
SetAccess                                   private
```

Data Types: `string`

`ProcessName` — Name of the C++ MEX host process
`MATLABMexHost` (default)

Name of the C++ MEX host process, returned as a string scalar.

Example: `MATLABMexHost`

Attributes:

```
GetAccess                                   public
SetAccess                                   private
```

Data Types: `string`

`ProcessIdentifier` — Process identifier of the C++ MEX host process
process ID

Process identifier of the C++ MEX host process, returned as a string scalar.

Attributes:

```
GetAccess                                   public
SetAccess                                   private
```

Data Types: `string`

Methods

Public Methods

feval Evaluate C++ MEX function in MEX host process

Examples

Create Host Process

Create an object of the `matlab.mex.MexHost` class using the `mexhost` function. Use the object to call the `feval`, which executes your C++ MEX function in the MEX host process.

```
mexHost = mexhost;
result = feval(mexHost,'MyMexFunction',inputs);
```

See Also

`mexhost`

Topics

"Out-of-Process Execution of C++ MEX Functions"

Introduced in R2019a

feval

Class: `matlab.mex.MexHost`
Package: `matlab.mex`

Evaluate C++ MEX function in MEX host process

Syntax

```
[result1,...,resultN] = feval(mexHost,mexFunctionName,
input1,...,inputM)
```

Description

`[result1,...,resultN] = feval(mexHost,mexFunctionName, input1,...,inputM)` evaluates the named C++ MEX function in the MEX host process and returns the results. You can run multiple C++ MEX functions in a single host process.

Input Arguments

`mexHost` — MEX host process
`matlab.mex.MexHost` object

MEX host process, specified as a `matlab.mex.MexHost` object. Create the MEX host process using the `mexhost` function.

`mexFunctionName` — Name of the C++ MEX function to execute
character vector | string scalar

Name of the C++ MEX function to execute, specified as a character vector or a string scalar.

Example: `"myMexFunction"`

Data Types: `char` | `string`

`input1,...,inputM` — Inputs to C++ MEX function
depends on function

Inputs to C++ MEX function. The number and type of values that can be specified as inputs depend on the C++ MEX function.

Output Arguments

`result1,...,resultN` — Results of evaluated C++ MEX function
depends on function

Results of evaluated C++ MEX function. The number and type of values returned depend on the C++ MEX function.

Examples

Run MEX Function Out of Process

Create a MEX host process and run a C++ MEX function in that process.

The `arrayProduct.cpp` C++ MEX file contains the source code for a function that multiplies an array by a scalar input and returns the resulting array. Open this file and save it on your MATLAB path. Build the C++ MEX source file using the `mex` command. Follow the instructions in “Build C++ MEX Programs” to set up the MEX build.

```
mex arrayProduct.cpp
```

Create a host process. The `mexhost` function returns a `matlab.mex.MexHost` object.

```
mh = mexhost;
```

Use the `feval` method of the `matlab.mex.MexHost` object to evaluate the C++ MEX function in the host process.

```
result = feval(mh,"arrayProduct",10,[2,4,6,8])
```

```
result =

    20    40    60    80
```

See Also

mexhost

Topics

"Out-of-Process Execution of C++ MEX Functions"

Introduced in R2019a

mex.getCompilerConfigurations

Get compiler configuration information for building MEX files

Syntax

```
cc = mex.getCompilerConfigurations
cc = mex.getCompilerConfigurations(lang)
cc = mex.getCompilerConfigurations(lang,list)
```

Description

`cc = mex.getCompilerConfigurations` returns an object `cc` containing information about the default compiler configurations used by the `mex` command. There is one configuration for each supported language.

`cc = mex.getCompilerConfigurations(lang)` returns an array of objects for the given language, `lang`.

`cc = mex.getCompilerConfigurations(lang,list)` returns information about the set of configurations, `list`.

Examples

Display Information for C Compiler

```
myCCompiler = mex.getCompilerConfigurations('C','Selected')
```

```
myCCompiler = 

  CompilerConfiguration with properties:

             Name: 'Microsoft Visual C++ 2010 (C)'
     Manufacturer: 'Microsoft'
         Language: 'C'
```

```
          Version: '10.0'
         Location: 'c:\Program Files (x86)\Microsoft Visual Studio 10.0'
        ShortName: 'MSVC100'
         Priority: 'A'
          Details: [1x1 mex.CompilerConfigurationDetails]
       LinkerName: 'link'
    LinkerVersion: ''
           MexOpt: 'C:\Users\auser\AppData\Roaming\MathWorks\MATLAB\R2014a\mex_C_win64
```

MATLAB displays information depending on your architecture and your version of MATLAB.

Display Number of Supported C Compilers

```
cLanguageCC = mex.getCompilerConfigurations('C','Supported');
length(cLanguageCC)
```

```
ans =
    10
```

The number of compilers for your version of MATLAB might be different.

Input Arguments

`lang` — Language
`'Any'` (default) | `'C'` | `'C++'` | `'CPP'` | `'Fortran'`

Language, specified as one of these values.

`'Any'`	Default value. All supported languages.
`'C'`	All C compiler configurations, including C++ configurations.
`'C++'` or `'CPP'`	All C++ compiler configurations.
`'Fortran'`	All Fortran compiler configurations.

`list` — Set of configurations
`'Selected'` (default) | `'Installed'` | `'Supported'`

Set of configurations, specified as one of these values.

`'Selected'`	The default compiler for each language.
`'Installed'`	All supported compilers `mex` finds installed on your system.
`'Supported'`	All compilers supported in the current release.

Output Arguments

`cc` — Compiler information
mex.CompilerConfiguration object or array of objects

Compiler information, specified as a `mex.CompilerConfiguration` object or array of `mex.CompilerConfiguration` objects. The `mex.CompilerConfiguration` class contains the following read-only properties.

Property	Purpose
`Name`	Compiler name.
`ShortName`	Text used to identify options file for the compiler.
`Manufacturer`	Name of the manufacturer of the compiler.
`Language`	Compiler language.
`Version`	(Windows platforms only) Version of the compiler.
`Location`	(Windows platforms only) Folder where compiler is installed.
`Details`	More read-only properties about the compiler configuration. These properties depend on the compiler, platform, and release of MATLAB.
`LinkerName`	Linker name.
`LinkerVersion`	(Windows platforms only) Version of the linker.
`MexOpt`	Name and full path to options file.
`Priority`	The priority of this compiler.

See Also

`mex`

External Websites

Supported and Compatible Compilers

Introduced in R2008b

MException

Capture error information

Description

Any MATLAB code that detects an error and throws an exception must construct an `MException` object. This class contains retrievable information about errors. MATLAB can throw either predefined exceptions or exceptions that you construct.

Creation

Syntax

```
ME = MException(msgID,msgtext)
ME = MException(msgID,msgtext,A1,...,An)
```

Description

`ME = MException(msgID,msgtext)` captures information about a specific error and stores it in the `MException` object, `ME`. The `MException` object is constructed with a message identifier, `msgID`, and an error message, `msgtext`.

`ME = MException(msgID,msgtext,A1,...,An)` allows formatting of the error message using text or numeric values, `A1,...,An`, to replace conversion specifiers in `msgtext` at run time.

Input Arguments

`msgID` — Identifier for error
character vector | string scalar

Identifier for the error, specified as a character vector or string scalar. Use the message identifier with exception handling to better identify the source of the error or to control a selected subset of the exceptions in your program.

The message identifier includes a `component` and `mnemonic`. The identifier must always contain a colon and follows a simple format: `component:mnemonic`. The `component` and `mnemonic` fields must each begin with a letter. The remaining characters can be alphanumerics (A–Z, a–z, 0–9) and underscores. No white-space characters can appear anywhere in `msgID`. For more information, see “Message Identifiers”.

Example: `'MyComponent:noSuchVariable'`

`msgtext` — Information about cause of error
character vector | string scalar

Information about the cause of the error and how you might correct it, specified as a character vector or string scalar. To format the text, use escape sequences, such as `\t` or `\n`. You also can use any format specifiers supported by the `sprintf` function, such as `%s` or `%d`. Specify values for the conversion specifiers via the `A1,...,An` input arguments. For more information, see “Formatting Text”.

Example: `'Error opening file.'`

Example: `'Error on line %d.'`

`A1,...,An` — Values
character vector | string scalar | numeric scalar

Values that replace the conversion specifiers in `msgtext`, specified as a character vector, string scalar, or numeric scalar.

Properties

`identifier` — Unique identifier of error
character vector

Character vector that uniquely identifies the error, specified as a character vector by the `msgID` input argument. This property is read-only. For more information, see “Message Identifiers”.

Example: `'MATLAB:test'`

Data Types: `char`

`message` — Error message
character vector

Character vector that contains the error message that is displayed when MATLAB throws the exception, specified by the `msgtext` and `A1,...,An` input arguments. This property is read only. For more information, see “Text of the Error Message”.

Example: `'Variable x not found'`

Data Types: `char`

`stack` — Stack trace information
array of structures

Structure array that contains stack trace information including the file name (`file`), function name (`name`), and line number (`line`) where MATLAB throws the exception. If the error occurs in a called function, the `stack` property also contains the file name, function name, and line number for each of the called functions. MATLAB generates the stack only when it throws the exception.

`stack` is an N-by-1 `struct` array, where N represents the depth of the call stack. This property is read-only. For more information, see “The Call Stack”.

Data Types: `struct`

`cause` — Cause of the exception
cell array of `MException` objects

Cell array of `MException` objects that caused MATLAB to create this exception. Use the `addCause` method to add an exception to the `cause` field of another exception. For more information, see “The Cause Array”.

`Correction` — Suggested fix for the exception
`matlab.lang.correction.AppendArgumentsCorrection` object | `matlab.lang.correction.ConvertToFunctionNotationCorrection` object | `matlab.lang.correction.ReplaceIdentifierCorrection` object

Suggested fix for this exception, specified as a `matlab.lang.correction.AppendArgumentsCorrection`, `matlab.lang.correction.ConvertToFunctionNotationCorrection`, or `matlab.lang.correction.ReplaceIdentifierCorrection` object. When an exception is thrown and not caught, MATLAB uses the `Correction` property to suggest a fix for the exception.

Object Functions

throw	Throw exception
MException.last	Return last uncaught exception
rethrow	Rethrow previously caught exception
throwAsCaller	Throw exception as if occurs within calling function
addCause	Record additional causes of exception
addCorrection	Provide suggested fix for exception
getReport	Get error message for exception

Examples

Create MException Object

```
msgID = 'myComponent:inputError';
msgtext = 'Input does not have the expected format.';

ME = MException(msgID,msgtext)
```

```
ME = 
  MException with properties:

    identifier: 'myComponent:inputError'
       message: 'Input does not have the expected format.'
         cause: {}
         stack: [0x1 struct]
    Correction: []
```

Create MException with Formatted Error Message

```
msgID = 'MATLAB:test';
msgtext = 'There are %d errors on this page';
A1 = 10;

ME = MException(msgID,msgtext,A1)
```

```
ME = 
  MException with properties:
```

```
    identifier: 'MATLAB:test'
       message: 'There are 10 errors on this page'
         cause: {}
         stack: [0x1 struct]
    Correction: []
```

Create and Throw MException Object

Throw an exception if an input variable name does not exist in the workspace.

```
str = input('Type a variable name: ','s');
if ~exist(str,'var')
    ME = MException('MyComponent:noSuchVariable', ...
        'Variable %s not found',str);
    throw(ME)
end
```

At the input prompt, enter any variable that does not exist in your workspace. For example, enter `notaVariable`.

```
Variable notaVariable not found
```

Since `notVariable` doesn't exist in your workspace, MATLAB creates an `MException` object, and then throws it.

Use try/catch to Capture Exception

Catch the exception generated by calling a nonexistent function, `notaFunction`. If the function is not defined, issue a warning and assign the output a value of 0.

```
try
    a = notaFunction(5,6);
catch ME
    if strcmp(ME.identifier,'MATLAB:UndefinedFunction')
        warning('Function is undefined.  Assigning a value of 0.');
    else
        rethrow(ME)
```

```
    end
end
```

```
Warning: Function is undefined.  Assigning a value of 0.
```

By itself, the call to `notaFunction` results in an error. Using `try` and `catch`, this code catches the undefined function exception and repackages it as a warning, allowing MATLAB to continue executing subsequent commands. If the caught exception has a different error identifier, MATLAB rethrows the exception.

See Also

`assert` | `dbstack` | `error` | `try, catch`

Topics

"Capture Information About Exceptions"

Introduced in R2007b

addCause

Record additional causes of exception

Syntax

```
baseException = addCause(baseException,causeException)
```

Description

`baseException = addCause(baseException,causeException)` modifies the existing `MException` object `baseException` by appending `causeException` to its `cause` property. Catching the resulting exception in a `try/catch` statement makes the base exception, along with all of the appended cause records, available to help diagnose the error.

Examples

Add Causes to Exception

Create an array, and an index into it with a logical array.

```
A = [13 42; 7 20];
idx = [1 0 1; 0 1 0];
```

Create an exception that provides general information about an error. Test the index array and add exceptions with more detailed information about the source of the failure.

```
try
    A(idx);
catch
    msgID = 'MYFUN:BadIndex';
    msg = 'Unable to index into array.';
    baseException = MException(msgID,msg);
```

```
    try
        assert(islogical(idx),'MYFUN:notLogical',...
            'Indexing array is not logical.')
    catch causeException
        baseException = addCause(baseException,causeException);
    end

    if any(size(idx) > size(A))
        msgID = 'MYFUN:incorrectSize';
        msg = 'Indexing array is too large.';
        causeException2 = MException(msgID,msg);
        baseException = addCause(baseException,causeException2);
    end
    throw(baseException)
end
```

```
Unable to index into array.

Caused by:
    Indexing array is not logical.
    Indexing array is too large.
```

Examine the `baseException` object.

```
baseException
```

```
baseException =

  MException with properties:

    identifier: 'MYFUN:BadIndex'
       message: 'Unable to index into array.'
         cause: {2x1 cell}
         stack: [0x1 struct]
```

The value of the `cause` property is a `2x1` cell array.

Examine the first cause of the exception.

```
baseException.cause{1}
```

```
ans =

  MException with properties:
```

```
    identifier: 'MYFUN:notLogical'
       message: 'Indexing array is not logical.'
         cause: {0x1 cell}
         stack: [0x1 struct]
```

Examine the second cause of the exception.

```
baseException.cause{2}
```

```
ans =

  MException with properties:

    identifier: 'MYFUN:incorrectSize'
       message: 'Indexing array is too large.'
         cause: {}
         stack: [0x1 struct]
```

Input Arguments

`baseException` — Primary exception
`MException` object

Primary exception containing the primary cause and location of an error, specified as an `MException` object.

`causeException` — Related exception
`MException` object

Related exception containing the cause and location of an error related to `baseException`, specified as an `MException` object.

See Also

`MException.last` | `assert` | `error` | `rethrow` | `throw` | `throwAsCaller` | `try, catch`

Introduced in R2007b

addCorrection

Provide suggested fix for exception

Syntax

```
meNew = addCorrection(meBase,meCorrection)
meBase = addCorrection(meBase,meCorrection)
```

Description

`meNew = addCorrection(meBase,meCorrection)` creates a `meNew` `MException` object from the `meBase` `MException` and the suggested fix `meCorrection`. To construct `meNew`, `addCorrection` makes a copy of `meBase` and appends `meCorrection` to the `meNew.Correction` property.

`meBase = addCorrection(meBase,meCorrection)` modifies the existing `meBase` `MException` object by appending `meCorrection` to its `Correction` property.

Examples

Suggest Fix When Function Called Without Arguments

The function `hello` requires one input argument. Add a suggested input argument `"world"` to the error message.

```
function hello(audience)
if nargin < 1
    me = MException('MATLAB:notEnoughInputs', 'Not enough input arguments.');
    aac = matlab.lang.correction.AppendArgumentsCorrection('"world"');
    me = me.addCorrection(aac);
    throw(me);
end
fprintf("Hello, %s!\n", audience);
end
```

Call the function without an argument.

```
hello

Error using hello (line 6)
Not enough input arguments.

Did you mean:
>> hello("world")
```

Input Arguments

meBase — Error information
MException

Error information, specified as an MException object.

meCorrection — Suggested fix for this exception
matlab.lang.correction.AppendArgumentsCorrection object | matlab.lang.correction.ConvertToFunctionNotationCorrection object | matlab.lang.correction.ReplaceIdentifierCorrection object

Suggested fix for this exception, specified as a matlab.lang.correction.AppendArgumentsCorrection, matlab.lang.correction.ConvertToFunctionNotationCorrection, or matlab.lang.correction.ReplaceIdentifierCorrection object.

See Also

Classes
matlab.lang.correction.AppendArgumentsCorrection | matlab.lang.correction.ConvertToFunctionNotationCorrection | matlab.lang.correction.ReplaceIdentifierCorrection

Functions
error

Introduced in R2019a

getReport

Get error message for exception

Syntax

```
msgText = getReport(exception)
msgText = getReport(exception,type)
msgText = getReport(exception,type,'hyperlinks',hlink)
```

Description

`msgText = getReport(exception)` gets the error message for an exception and returns it as formatted text, `msgText`. The message is the value of the `message` property of the `MException` object, `exception`. It is the same text that MATLAB displays when it throws the exception.

`msgText = getReport(exception,type)` returns the error message using the indicated level of detail, specified by `type`.

`msgText = getReport(exception,type,'hyperlinks',hlink)` uses the value of `hlink` to determine whether to include active hyperlinks to the failing lines of code within the error message.

Examples

Get Error Message from Exception

Cause MATLAB to throw an exception.

```
plus
```

```
Error using +
Not enough input arguments.
```

Get the error message from the exception.

```
exception = MException.last;
msgText = getReport(exception)
```

```
msgText =

Error using +
Not enough input arguments.
```

Specify Detail Level in Error Message

In a file in your current working folder, create the following function in `testFunc.m`.

```
function a = testFunc
try
    a = notaFunction(5,6);
catch a

end
```

Since the function, `notaFunction`, does not exist, `testFunc` returns an `MException` object.

At the command prompt, call `testFunc` and get the error message.

```
m = testFunc;
msgText = getReport(m)
```

```
msgText =

Undefined function 'notaFunction' for input arguments of type 'double'.

Error in testFunc (line 3)
    a = notaFunction(5,6);
```

Specify that the error message only contains the error message and not the stack information.

```
msgText = getReport(m,'basic')
```

```
msgText =

Undefined function 'notaFunction' for input arguments of type 'double'.
```

Turn Off Hyperlinks in Error Message

Cause MATLAB to throw an exception.

```
try
    surf
catch exception
end
```

Get the error message from the exception.

```
msgText = getReport(exception)
```

```
msgText =

Error using surf (line 49)
Not enough input arguments.
```

Get the error message without active hyperlinks to `surf.m`.

```
msgText = getReport(exception,'extended','hyperlinks','off')
```

```
msgText =

Error using surf (line 49)
Not enough input arguments.
```

Input Arguments

`exception` — Exception object that provides error message
`MException` object

Exception object that provides the error message, specified as a scalar `MException` object.

`type` — Detail indicator of message
`'extended'` (default) | `'basic'`

Detail indicator of the message returned, specified as `'extended'` or `'basic'`.

type Value	msgText Detail Level
`'extended'` (default)	`msgText` includes the line number, error message, cause, and stack summary. To display the proper stack, MATLAB first must throw an exception.
`'basic'`	`msgText` includes the error message.

`hlink` — Hyperlink indicator of message
`'on'` (default) | `'off'` | `'default'`

Hyperlink indicator of the message that includes active hyperlinks to the failing lines of code, specified as `'on'`, `'off'`, or `'default'`.

hlink Value	Action
`'on'`	Display hyperlinks to failing lines of code.
`'off'`	Do not display hyperlinks to failing lines of code.
`'default'`	Use the default for the Command Window to determine if to use hyperlinks in the error message.

See Also
`error` | `throw` | `try, catch`

Introduced in R2007b

MException.last

Return last uncaught exception

Syntax

```
exception = MException.last
MException.last('reset')
```

Description

`exception = MException.last` returns the contents of the most recently thrown, uncaught `MException` object. `MException.last` is not set if a `try/catch` statement catches the last exception. `MException.last` is a static function.

`MException.last('reset')` clears the properties of the exception returned from `MException.last`. It sets the `MException` `identifier` and `message` properties to an empty character vector, the `stack` property to a 0-by-1 structure, and the `cause` property to an empty cell array.

Examples

Get Last Uncaught Exception

Cause MATLAB to throw, but not catch, an exception.

```
A = 25;
A(2)
```

```
Index exceeds matrix dimensions.
```

Get the uncaught exception.

```
exception = MException.last
```

```
exception =
```

```
  MException with properties:

    identifier: 'MATLAB:badsubscript'
       message: 'Index exceeds matrix dimensions.'
         cause: {}
         stack: [0x1 struct]
```

Reset Last Uncaught Exception

Call the `surf` function with no input arguments.

```
surf
```

```
Error using surf (line 49)
Not enough input arguments.
```

Get the uncaught exception.

```
exception = MException.last
```

```
exception =

  MException with properties:

    identifier: 'MATLAB:narginchk:notEnoughInputs'
       message: 'Not enough input arguments.'
         cause: {}
         stack: [1x1 struct]
```

Get the last, uncaught exception.

```
MException.last('reset')
exception = MException.last
```

```
exception =

  MException with properties:

    identifier: ''
       message: ''
```

```
        cause: {0x1 cell}
        stack: [0x1 struct]
```

Tips

- Use `MException.last` only from the Command Window, not within a function.

See Also

`error` | `throw` | `try, catch`

Introduced in R2007b

rethrow

Rethrow previously caught exception

Syntax

```
rethrow(exception)
```

Description

`rethrow(exception)` rethrows a previously caught exception, `exception`. MATLAB typically responds to errors by terminating the currently running program. However, you can use a `try/catch` block to catch the exception. This interrupts the program termination so you can execute your own error handling procedures. To terminate the program and redisplay the exception, end the `catch` block with a `rethrow` statement.

`rethrow` handles the stack trace differently from `error`, `assert`, and `throw`. Instead of creating the stack from where MATLAB executes the function, `rethrow` preserves the original exception information and enables you to retrace the source of the original error.

Examples

Catch and Rethrow Exception

Cause MATLAB to throw an error by calling `surf` with no inputs. Catch the exception, display the error identifier, and rethrow the exception.

```
try
    surf
catch ME
    disp(['ID: ' ME.identifier])
    rethrow(ME)
end
```

```
ID: MATLAB:narginchk:notEnoughInputs
Error using surf (line 49)
Not enough input arguments.
```

Compare Behavior of `throw` and `rethrow`

Create a function, `combineArrays`, in your working folder.

```
function C = combineArrays(A,B)
try
    C = catAlongDim1(A,B);       % Line 3
catch exception
    throw(exception)             % Line 5
end
end

function V = catAlongDim1(V1,V2)
V = cat(1,V1,V2);                % Line 10
end
```

Call the `combineArrays` function with arrays of different sizes.

```
A = 1:5;
B = 1:4;

combineArrays(A,B)
```

```
Error using combineArrays (line 5)
Dimensions of matrices being concatenated are not consistent.
```

The stack refers to line 5 where MATLAB throws the exception.

Replace `throw(exception)` with `rethrow(exception)` on line 5 of the `combineArrays` function, and call the function again.

```
combineArrays(A,B)
```

```
Error using cat
Dimensions of matrices being concatenated are not consistent.

Error in combineArrays>catAlongDim1 (line 10)
V = cat(1,V1,V2);                % Line 10
```

```
Error in combineArrays (line 3)
    C = catAlongDim1(A,B);       % Line 3
```

The `rethrow` function maintains the original stack and indicates the error is on line 3.

Input Arguments

`exception` — Exception containing cause and location of error
`MException` object

Exception containing the cause and location of an error, specified as a scalar `MException` object.

See Also

`MException.last` | `assert` | `error` | `throw` | `throwAsCaller` | `try, catch`

Introduced in R2007b

throw

Throw exception

Syntax

`throw(exception)`

Description

`throw(exception)` throws an exception based on the information contained in the `MException` object, `exception`. The exception terminates the currently running function and returns control either to the keyboard or to an enclosing `catch` block. When you throw an exception from outside a `try/catch` statement, MATLAB displays the error message in the Command Window.

The `throw` function, unlike the `throwAsCaller` and `rethrow` functions, creates the stack trace from the location where MATLAB calls the function.

You can access the `MException` object via a `try/catch` statement or the `MException.last` function.

Examples

Create and Throw MException Object

Throw an exception if an input variable name does not exist in the workspace.

```
str = input('Type a variable name: ','s');
if ~exist(str,'var')
    ME = MException('MyComponent:noSuchVariable', ...
        'Variable %s not found',str);
    throw(ME)
end
```

At the input prompt, enter any variable that does not exist in your workspace. For example, enter `notaVariable`.

```
Variable notaVariable not found
```

Since `notVariable` doesn't exist in your workspace, MATLAB creates an `MException` object, and then throws it.

Compare Behavior of `throw` and `rethrow`

Create a function, `combineArrays`, in your working folder.

```
function C = combineArrays(A,B)
try
    C = catAlongDim1(A,B);       % Line 3
catch exception
    throw(exception)             % Line 5
end
end

function V = catAlongDim1(V1,V2)
V = cat(1,V1,V2);                % Line 10
end
```

Call the `combineArrays` function with arrays of different sizes.

```
A = 1:5;
B = 1:4;

combineArrays(A,B)
```

```
Error using combineArrays (line 5)
Dimensions of matrices being concatenated are not consistent.
```

The stack refers to line 5 where MATLAB throws the exception.

Replace `throw(exception)` with `rethrow(exception)` on line 5 of the `combineArrays` function, and call the function again.

```
combineArrays(A,B)
```

```
Error using cat
Dimensions of matrices being concatenated are not consistent.
```

```
Error in combineArrays>catAlongDim1 (line 10)
V = cat(1,V1,V2);                % Line 10

Error in combineArrays (line 3)
    C = catAlongDim1(A,B);       % Line 3
```

The `rethrow` function maintains the original stack and indicates the error is on line 3.

Input Arguments

`exception` — Exception containing cause and location of error
`MException` object

Exception containing the cause and location of an error, specified as a scalar `MException` object.

See Also

`MException.last` | `error` | `rethrow` | `throwAsCaller` | `try, catch`

Introduced in R2007b

throwAsCaller

Throw exception as if occurs within calling function

Syntax

```
throwAsCaller(exception)
```

Description

`throwAsCaller(exception)` throws an exception as if it occurs within the calling function. The exception terminates the currently running function and returns control to the keyboard or an enclosing `catch` block. When you throw an exception from outside a `try/catch` statement, MATLAB displays the error message in the Command Window.

You can access the `MException` object via a `try/catch` statement or the `MException.last` function.

Sometimes, it is more informative for the error to point to the location in the calling function that results in the exception rather than pointing to the function that actually throws the exception. You can use `throwAsCaller` to simplify the error display.

Examples

Compare Behavior of `throw` and `throwAsCaller`

Create a function, `sayHello`, in your working folder.

```
function sayHello(N)
checkInput(N)
str = ['Hello, ' N '!'];
disp(str)

function checkInput(N)
if ~ischar(N)
```

```
    ME = MException('sayHello:inputError','Input must be char.');
    throw(ME)
end
```

At the command prompt, call the function with a numeric input.

```
sayHello(42)
```

```
Error using sayHello>checkInput (line 9)
Input must be char.

Error in sayHello (line 2)
checkInput(N)
```

The top of the stack refers to line 9 because this is where MATLAB throws the exception. After the initial stack frame, MATLAB displays information from the calling function.

Replace `throw(ME)` with `throwAsCaller(ME)` in line 9 of `sayHello.m` and call the function again.

```
sayHello(42)
```

```
Error using sayHello (line 2)
Input must be char.
```

The top of the stack refers to line 2 because that is the location of the error in the calling function.

Input Arguments

`exception` — Exception containing cause and location of error
`MException` object

Exception containing the cause and location of an error, specified as a scalar `MException` object.

See Also

`MException.last` | `error` | `rethrow` | `throw` | `try, catch`

Introduced in R2007b

mexext

Binary MEX file-name extension

Syntax

```
ext = mexext
extlist = mexext('all')
```

Description

`ext = mexext` returns the file-name extension for the current platform.

`extlist = mexext('all')` returns the extensions for all platforms.

Examples

Display File Extension for Your Computer

Find the MEX file extension for the system you are currently working on.

```
ext = mexext
```

```
ext =
'mexw64'
```

Your results reflect your system.

Find MEX File Extension for Mac Platform

Get the list of file extensions for supported platforms.

```
extlist = mexext('all');
```

The `mex` command identifies a platform by its `arch` value, which is the output of the `computer('arch')` command. For Mac platforms, the value is `maci64`.

Search the `arch` field in `extlist` for `'maci64'`, and display the corresponding `ext` field.

```
for k=1:length(extlist)
   if strcmp(extlist(k).arch, 'maci64')
   disp(sprintf('Arch: %s  File Extension: %s', extlist(k).arch, extlist(k).ext))
   end
end
```

```
Arch: maci64  File Extension: mexmaci64
```

Output Arguments

`ext` — File-name extension

`mexa64` | `mexmaci64` | `mexw64`

File-name extension for MEX file, returned as one of these values.

MEX File Platform-Dependent Extension

Platform	Binary MEX File Extension
Linux (64-bit)	`mexa64`
Apple Mac (64-bit)	`mexmaci64`
Windows (64-bit)	`mexw64`

`extlist` — All file-name extensions

structure

All file-name extensions, returned as a structure with these fields:

`arch` — Platform

character vector

Platform, returned as a character vector. The name of the platform is the output of the `computer('arch')` command.

`ext` — File extension

character vector

File extension, returned as a character vector.

Tips

- To use the MEX file-name extension in makefiles or scripts outside MATLAB, type one of the following from the system command prompt. The script is located in the *matlabroot*`\bin` folder.
  - `mexext.bat`—Windows platform.
  - `mexext.sh`—UNIX platform.

    For example, the following commands are in a GNU makefile.

    ```
    ext = $(shell mexext)
    yprime.$(ext) : yprime.c
            mex yprime.c
    ```

See Also

`computer` | `mex`

Introduced before R2006a

mfilename

File name of currently running code

Syntax

```
mfilename
p = mfilename('fullpath')
c = mfilename('class')
```

Description

`mfilename` returns a character vector containing the file name of the file in which the function call occurs. When called from within the file, it returns the name of that file. This allows a script or function to determine its name.

`p = mfilename('fullpath')` returns the full path and name of the file in which the call occurs, not including the filename extension.

`c = mfilename('class')` in a method, returns the class of the method, not including the leading `@` sign. If called from a nonmethod, it yields the empty character vector.

Tips

If `mfilename` is called with any argument other than the above two, it behaves as if it were called with no argument.

When called from the command line, `mfilename` returns an empty character vector.

To get the names of the callers of a MATLAB function file, use `dbstack` with an output argument.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`dbstack` | `function` | `inputname` | `nargin` | `nargout`

Introduced before R2006a

mget

Download files from FTP server

Syntax

```
mget(ftpobj,contents)
mget(ftpobj,contents,target)
downloadPaths = mget( ___ )
```

Description

`mget(ftpobj,contents)` retrieves the files or folders specified by `contents` from the FTP server associated with `ftpobj` into the MATLAB current folder.

`mget(ftpobj,contents,target)` retrieves the files or folders into the local folder specified by the absolute or relative path in `target`. If the local folder does not exist, `mget` creates it.

`downloadPaths = mget( ___ )` also returns the paths to the downloaded files and folders as a cell array of character vectors. You can use the input arguments from either of the previous syntaxes.

Examples

Download File

Download a text file from an FTP server and display its contents.

First, connect to the National Centers for Environmental Information (NCEI) FTP server.

```
ftpobj = ftp('ftp.ngdc.noaa.gov')
```

```
ftpobj =
```

```
  FTP Object
     host: ftp.ngdc.noaa.gov
     user: anonymous
      dir: /
     mode: binary
```

Download a text file. The `mget` function downloads the file to the current folder on your machine.

```
mget(ftpobj,'README.txt');
```

Display the beginning of `README.txt`. To read the copy of `README.txt` downloaded to your computer, use the `fileread` function.

```
readme = fileread('README.txt');
readme(1:95)
```

```
ans =
    '                   Welcome to the
         NOAA/National Centers for Environmental Information (NCEI)'
```

FTP service courtesy of the National Centers for Environmental Information (NCEI). See the NCEI Privacy Policy, Disclaimer, and Copyright for NCEI terms of service.

Download File to Specified Folder

Download a text file from an FTP server to a specified folder on your local machine.

First, connect to the National Centers for Environmental Information (NCEI) FTP server.

```
ftpobj = ftp('ftp.ngdc.noaa.gov')
```

```
ftpobj =

  FTP Object
     host: ftp.ngdc.noaa.gov
     user: anonymous
      dir: /
     mode: binary
```

Download a text file to a folder named `myLocalFolder`. If this folder does not exist, then the `mget` function creates it on your local machine.

```
mget(ftpobj,'README.txt','myLocalFolder');
```

Read the beginning of `README.txt` using the `fileread` function.

```
readme = fileread('myLocalFolder/README.txt');
readme(1:95)
```

```
ans =
    '                    Welcome to the
         NOAA/National Centers for Environmental Information (NCEI)'
```

FTP service courtesy of the National Centers for Environmental Information (NCEI). See the NCEI Privacy Policy, Disclaimer, and Copyright for NCEI terms of service.

Input Arguments

`ftpobj` — Connection to FTP server
FTP object

Connection to an FTP server, specified as an FTP object.

`contents` — Remote files or folders
character vector | string scalar

Remote files or folders, specified as a character vector or string scalar.

To match multiple files or folders on the FTP server, you can include a wildcard character (*) in `contents`. For example, if you specify `contents` as `*.docx`, then `mget` downloads all files whose names end with `.docx`.

`target` — Local folder
character vector | string scalar

Local folder, specified as a character vector or string scalar. `target` can specify a relative or absolute path.

See Also

`cd` | `ftp` | `mput`

Introduced before R2006a

milliseconds

Duration in milliseconds

Syntax

```
MS = milliseconds(X)
```

Description

`MS = milliseconds(X)` returns an array of milliseconds equivalent to the values in `X`.

- If `X` is a numeric array, then `MS` is a `duration` array in units of milliseconds.
- If `X` is a `duration` array, then `MS` is a `double` array with each element equal to the number of milliseconds in the corresponding element of `X`.

Examples

Create Duration Array of Milliseconds

```
X = magic(3);
MS = milliseconds(X)
```

```
MS = 3x3 duration array
   0.008 sec   0.001 sec   0.006 sec
   0.003 sec   0.005 sec   0.007 sec
   0.004 sec   0.009 sec   0.002 sec
```

Convert Durations to Numeric Array of Milliseconds

Create a `duration` array.

```
X = minutes(2) + seconds(1:3)
```

```
X = 1x3 duration array
   2.0167 min   2.0333 min     2.05 min
```

Convert each duration in X to a number of milliseconds.

```
MS = milliseconds(X)
```

```
MS = 1×3

      121000      122000      123000
```

MS is a double array.

Input Arguments

X — Input array
numeric array | duration array | logical array

Input array, specified as a numeric array, duration array, or logical array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This function fully supports code generation. For more information, see “Code Generation for Duration Arrays” (MATLAB Coder).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`duration`

Introduced in R2015a

min

Minimum elements of an array

Syntax

```
M = min(A)
M = min(A,[],dim)
M = min(A,[],nanflag)
M = min(A,[],dim,nanflag)
[M,I] = min( ___ )

M = min(A,[],'all')
M = min(A,[],vecdim)
M = min(A,[],'all',nanflag)
M = min(A,[],vecdim,nanflag)

[M,I] = min(A,[], ___ ,'linear')

C = min(A,B)
C = min(A,B,nanflag)
```

Description

`M = min(A)` returns the minimum elements of an array.

- If `A` is a vector, then `min(A)` returns the minimum of `A`.
- If `A` is a matrix, then `min(A)` is a row vector containing the minimum value of each column.
- If `A` is a multidimensional array, then `min(A)` operates along the first array dimension whose size does not equal `1`, treating the elements as vectors. The size of this dimension becomes `1` while the sizes of all other dimensions remain the same. If `A` is an empty array with first dimension `0`, then `min(A)` returns an empty array with the same size as `A`.

`M = min(A,[],dim)` returns the minimum element along dimension `dim`. For example, if `A` is a matrix, then `min(A,[],2)` is a column vector containing the minimum value of each row.

`M = min(A,[],nanflag)` specifies whether to include or omit `NaN` values in the calculation. For example, `min(A,[],'includenan')` includes all `NaN` values in `A` while `min(A,[],'omitnan')` ignores them.

`M = min(A,[],dim,nanflag)` also specifies the dimension to operate along when using the `nanflag` option.

`[M,I] = min( ___ )` also returns the index into the operating dimension that corresponds to the minimum value of `A` for any of the previous syntaxes.

`M = min(A,[],'all')` finds the minimum over all elements of `A`. This syntax is valid for MATLAB versions R2018b and later.

`M = min(A,[],vecdim)` computes the minimum over the dimensions specified in the vector `vecdim`. For example, if `A` is a matrix, then `min(A,[],[1 2])` computes the minimum over all elements in `A`, since every element of a matrix is contained in the array slice defined by dimensions 1 and 2.

`M = min(A,[],'all',nanflag)` computes the minimum over all elements of `A` when using the `nanflag` option.

`M = min(A,[],vecdim,nanflag)` specifies multiple dimensions to operate along when using the `nanflag` option.

`[M,I] = min(A,[], ___ ,'linear')` returns the linear index into `A` that corresponds to the minimum value in `A`.

`C = min(A,B)` returns an array with the smallest elements taken from `A` or `B`.

`C = min(A,B,nanflag)` also specifies how to treat `NaN` values.

Examples

Smallest Vector Element

Create a vector and compute its smallest element.

```
A = [23 42 37 15 52];
M = min(A)
```

```
M = 15
```

Smallest Complex Element

Create a complex vector and compute its smallest element, that is, the element with the smallest magnitude.

```
A = [-2+2i 4+i -1-3i];
min(A)
```

```
ans = -2.0000 + 2.0000i
```

Smallest Element in Each Matrix Column

Create a matrix and compute the smallest element in each column.

```
A = [2 8 4; 7 3 9]
```

```
A = 2×3

     2     8     4
     7     3     9
```

```
M = min(A)
```

```
M = 1×3

     2     3     4
```

Smallest Element in Each Matrix Row

Create a matrix and compute the smallest element in each row.

```
A = [1.7 1.2 1.5; 1.3 1.6 1.99]
```

```
A = 2×3

    1.7000    1.2000    1.5000
    1.3000    1.6000    1.9900
```

```
M = min(A,[],2)
```

```
M = 2×1

    1.2000
    1.3000
```

Smallest Element Involving NaN

Create a vector and compute its minimum, excluding NaN values.

```
A = [1.77 -0.005 3.98 -2.95 NaN 0.34 NaN 0.19];
M = min(A,[],'omitnan')
```

```
M = -2.9500
```

`min(A)` will also produce this result since `'omitnan'` is the default option.

Use the `'includenan'` flag to return NaN.

```
M = min(A,[],'includenan')
```

```
M = NaN
```

Smallest Element Indices

Create a matrix A and compute the smallest elements in each column as well as the row indices of A in which they appear.

```
A = [1 9 -2; 8 4 -5]
```

```
A = 2×3

     1     9    -2
     8     4    -5
```

```
[M,I] = min(A)
```

```
M = 1×3

     1     4    -5


I = 1×3

     1     2     2
```

Minimum of Array Page

Create a 3-D array and compute the minimum over each page of data (rows and columns).

```
A(:,:,1) = [2 4; -2 1];
A(:,:,2) = [9 13; -5 7];
A(:,:,3) = [4 4; 8 -3];
M1 = min(A,[],[1 2])
```

```
M1 =
M1(:,:,1) =

    -2


M1(:,:,2) =

    -5


M1(:,:,3) =

    -3
```

Starting in R2018b, to compute the minimum over all dimensions of an array, you can either specify each dimension in the vector dimension argument, or use the `'all'` option.

```
M2 = min(A,[],[1 2 3])
```

```
M2 = -5
```

```
Mall = min(A,[],'all')
```

```
Mall = -5
```

Return Linear Indices

Create a matrix `A` and return the minimum value of each row in the matrix `M`. Use the `'linear'` option to also return the linear indices `I` such that `M = A(I)`.

```
A = [1 2 3; 4 5 6]
```

```
A = 2×3

     1     2     3
     4     5     6
```

```
[M,I] = min(A,[],2,'linear')
```

```
M = 2×1

     1
     4


I = 2×1

     1
     2
```

```
minvals = A(I)
```

```
minvals = 2×1

     1
```

```
     4
```

Smallest Element Comparison

Create a matrix and return the smallest value between each of its elements compared to a scalar.

```
A = [1 7 3; 6 2 9]
```

```
A = 2×3

     1     7     3
     6     2     9
```

```
B = 5;
C = min(A,B)
```

```
C = 2×3

     1     5     3
     5     2     5
```

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

- If `A` is complex, then `min(A)` returns the complex number with the smallest magnitude. If magnitudes are equal, then `min(A)` returns the value with the smallest magnitude and the smallest phase angle.
- If `A` is a scalar, then `min(A)` returns `A`.
- If `A` is a 0-by-0 empty array, then `min(A)` is as well.

If `A` has type `categorical`, then it must be ordinal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `categorical` | `datetime` | `duration`
Complex Number Support: Yes

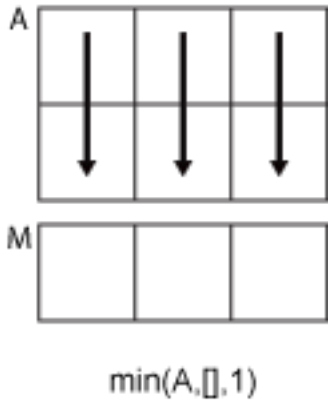
`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

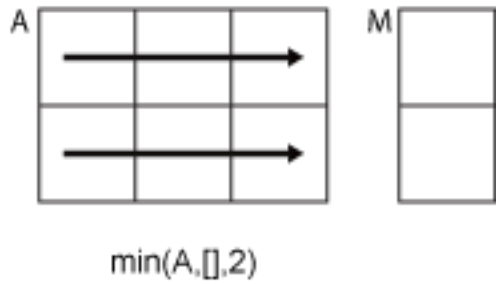
Dimension `dim` indicates the dimension whose length reduces to `1`. The `size(M,dim)` is `1`, while the sizes of all other dimensions remain the same, unless `size(A,dim)` is `0`. If `size(A,dim)` is `0`, then `min(A,dim)` returns an empty array with the same size as `A`.

Consider a two-dimensional input array, `A`:

- If `dim = 1`, then `min(A,[],1)` returns a row vector containing the smallest element in each column.



- If `dim = 2`, then `min(A,[],2)` returns a column vector containing the smallest element in each row.

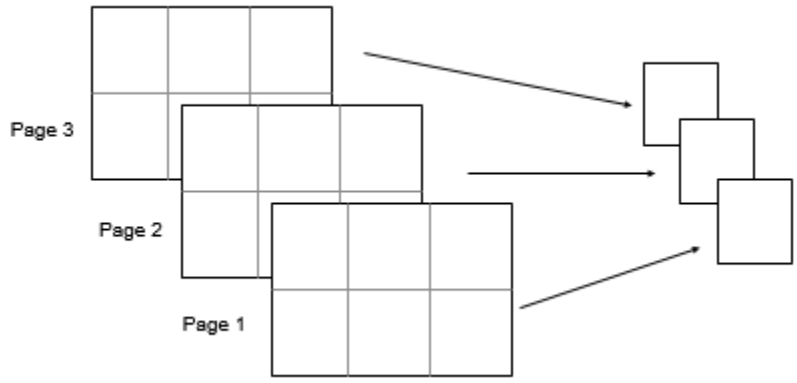


`min` returns `A` if `dim` is greater than `ndims(A)`.

`vecdim` — Vector of dimensions
vector of positive integers

Vector of dimensions, specified as a vector of positive integers. Each element represents a dimension of the input array. The lengths of the output in the specified operating dimensions are 1, while the others remain the same.

Consider a 2-by-3-by-3 input array, `A`. Then `min(A,[],[1 2])` returns a 1-by-1-by-3 array whose elements are the minimums computed over each page of `A`.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

B — Additional input array
scalar | vector | matrix | multidimensional array

Additional input array, specified as a scalar, vector, matrix, or multidimensional array. Numeric inputs A and B must either be the same size or have sizes that are compatible (for example, A is an M-by-N matrix and B is a scalar or 1-by-N row vector). For more information, see “Compatible Array Sizes for Basic Operations”.

If A and B are datetime, duration, or categorical arrays, then they must be the same size unless one is a scalar.

- A and B must be the same data type unless one is a double. In that case, the data type of the other array can be single, duration, or any integer type.
- If A and B are ordinal categorical arrays, they must have the same sets of categories with the same order.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | categorical | datetime | duration
Complex Number Support: Yes

nanflag — NaN condition
'omitnan' (default) | 'includenan'

NaN condition, specified as one of these values:

- 'omitnan' — Ignore all NaN values in the input. If all elements are NaN, then min returns the first one.
- 'includenan' — Include the NaN values in the input for the calculation.

For datetime arrays, you can also use 'omitnat' or 'includenat' to omit and include NaT values, respectively.

For categorical arrays, you can also use 'omitundefined' or 'includeundefined' to omit and include undefined values, respectively.

Data Types: char

Output Arguments

M — Minimum values
scalar | vector | matrix | multidimensional array

Minimum values, returned as a scalar, vector, matrix, or multidimensional array. size(M,dim) is 1, while the sizes of all other dimensions match the size of the

corresponding dimension in `A`, unless `size(A,dim)` is `0`. If `size(A,dim)` is `0`, then `M` is an empty array with the same size as `A`.

`I` — Index
scalar | vector | matrix | multidimensional array

Index, returned as a scalar, vector, matrix, or multidimensional array. `I` is the same size as the first output.

When `'linear'` is not specified, `I` is the index into the operating dimension. When `'linear'` is specified, `I` contains the linear indices of `A` corresponding to the minimum values.

If the smallest element occurs more than once, then `I` contains the index to the first occurrence of the value.

`C` — Minimum elements from `A` or `B`
scalar | vector | matrix | multidimensional array

Minimum elements from `A` or `B`, returned as a scalar, vector, matrix, or multidimensional array. The size of `C` is determined by implicit expansion of the dimensions of `A` and `B`. For more information, see “Compatible Array Sizes for Basic Operations”.

The data type of `C` depends on the data types of `A` and `B`:

- If `A` and `B` are the same data type, then `C` matches the data type of `A` and `B`.
- If either `A` or `B` is `single`, then `C` is `single`.
- If either `A` or `B` is an integer data type with the other a scalar `double`, then `C` assumes the integer data type.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If you specify `dim`, then it must be a constant.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).
- See “Code Generation for Complex Data with Zero-Valued Imaginary Parts” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bounds` | `max` | `mean` | `median` | `mink` | `sort`

Topics

“Array Indexing”

Introduced before R2006a

MinimizeCommandWindow

Minimize size of Automation server window

Syntax

IDL Method Signature

```
HRESULT MinimizeCommandWindow(void)
```

Microsoft Visual Basic Client

```
MinimizeCommandWindow
```

MATLAB Client

```
MinimizeCommandWindow(h)
```

Description

`MinimizeCommandWindow(h)` minimizes the window for the server attached to handle `h`, and makes it inactive.

If the server window was already in a minimized state, `MinimizeCommandWindow` does nothing.

Examples

Adjust MATLAB Command Window in Visual Basic .NET

This example shows how to minimize and maximize the command window in a MATLAB Automation server. Create an application from the following code.

```
type adjustcommandwindow.vb
```

```
Dim Matlab As Object

Matlab = CreateObject("matlab.application")
Matlab.MinimizeCommandWindow

'Now return the server window to its former state on
'the desktop and make it the currently active window.

Matlab.MaximizeCommandWindow
```

Adjust MATLAB Command Window in VBA

This example shows how to minimize and maximize the command window in a MATLAB Automation server. Create an application from the following code.

```
type adjustcommandwindow.vba
```

```
Dim Matlab As Object

Set Matlab = CreateObject("matlab.application")
Matlab.MinimizeCommandWindow

'Now return the server window to its former state on
'the desktop and make it the currently active window.

Matlab.MaximizeCommandWindow
```

See Also

`MaximizeCommandWindow`

Introduced before R2006a

mink

Find k smallest elements of array

Syntax

```
B = mink(A,k)
B = mink(A,k,dim)
B = mink( ___ ,'ComparisonMethod',c)
[B,I] = mink( ___ )
```

Description

`B = mink(A,k)` returns the `k` smallest elements of `A`.

- If `A` is a vector, then `mink` returns a vector containing the `k` smallest elements of `A`.
- If `A` is a matrix, then `mink` returns a matrix whose columns contain the `k` smallest elements of each column of `A`.
- If `A` is a multidimensional array, then `mink` returns the `k` smallest elements along the first dimension whose size does not equal 1.

`B = mink(A,k,dim)` determines the `k` smallest elements of `A` along dimension `dim`.

`B = mink( ___ ,'ComparisonMethod',c)` optionally specifies how to compare elements of `A` for any of the previous syntaxes. For example, `mink(A,k,'ComparisonMethod','abs')` returns the `k` smallest elements of `A` according to their absolute values.

`[B,I] = mink( ___ )` finds the indices of the smallest `k` values of `A` and returns them in `I`.

Examples

Smallest Vector Elements

Compute the smallest 3 elements of a vector.

```
A = 1:10;
B = mink(A,3)
```

```
B = 1×3

     1     2     3
```

Smallest Elements of Matrix Rows

Compute the smallest 3 elements of each row of a matrix.

```
A = magic(5)
```

```
A = 5×5

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

```
B = mink(A,3,2)
```

```
B = 5×3

     1     8    15
     5     7    14
     4     6    13
     3    10    12
     2     9    11
```

Complex Vector

Compute the 2 smallest elements of a complex vector according to their magnitude, and return the indices where they are located in the input vector.

```
A = [2-2i 5+i -7-3i -1+i]
```

```
A = 1×4 complex

   2.0000 - 2.0000i   5.0000 + 1.0000i  -7.0000 - 3.0000i  -1.0000 + 1.0000i
```

```
[B,I] = mink(A,2,'ComparisonMethod','abs')
```

```
B = 1×2 complex

  -1.0000 + 1.0000i   2.0000 - 2.0000i


I = 1×2

     4     1
```

Input Arguments

A — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

- If A is a vector, then `mink` returns a vector containing the k smallest elements of A.
- If A is a matrix, then `mink` returns a matrix whose columns contain the k smallest elements of each column of A.
- If A is a multidimensional array, then `mink` returns the k smallest elements along the first dimension whose size does not equal 1.

If A has type `categorical`, then it must be ordinal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `categorical` | `datetime` | `duration`

Complex Number Support: Yes

k — Number of minima
nonnegative integer scalar

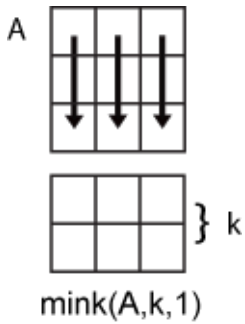
Number of minima to return, specified as a positive integer scalar. If `k` is greater than or equal to the number of elements in the operating dimension, then `mink` sorts the input array along that dimension.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

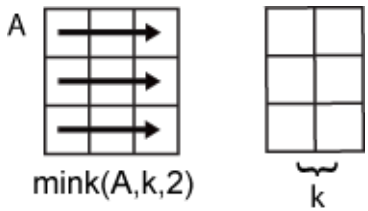
`dim` — Operating dimension
positive integer scalar

Operating dimension, specified as a positive integer scalar. By default, `mink` operates along the first dimension whose size does not equal 1.

For example, if `A` is a matrix, then `mink(A,k,1)` operates along the rows of `A`, computing minimums for each column.



`mink(A,k,2)` operates along the columns of `A`, computing minimums for each row.



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

c — Comparison method
`'auto'` (default) | `'real'` | `'abs'`

Comparison method, specified as one of the following:

- `'auto'` — Compare elements of input `A` by `real(A)` when `A` is real, and by `abs(A)` when `A` is complex.
- `'real'` — Compare elements of input `A` by `real(A)` when `A` is real or complex. If `A` has elements with equal real parts, then use `imag(A)` to break ties.
- `'abs'` — Compare elements of input `A` by `abs(A)` when `A` is real or complex. If `A` has elements with equal magnitude, then use `angle(A)` in the interval (-π,π] to break ties.

Output Arguments

B — Output array
scalar | vector | matrix | multidimensional array

Output array, returned as a scalar, vector, matrix, or multidimensional array. `mink` returns the `k` elements in order from smallest to largest.

I — Index array
scalar | vector | matrix | multidimensional array

Index array, returned as a vector, matrix, or multidimensional array. `I` is the same size as `B`. If the output array `B` contains repeated elements, then the order of their indices in `I` matches the order in which they appear in the input array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If `A` is complex with all zero imaginary parts, then MATLAB might convert `A` to `real(A)` before calling `mink(A)`. In this case, MATLAB compares elements of `A` by `real(A)`, but the generated code compares elements of `A` by `abs(A)`. To make the generated code match MATLAB, use `mink(real(A),k)` or `mink(A,k,'ComparisonMethod','real')`. See "Code Generation for Complex Data with Zero-Valued Imaginary Parts" (MATLAB Coder).
- If you supply `dim`, then it must be constant.
- For limitations related to variable-size inputs, see "Variable-Sizing Restrictions for Code Generation of Toolbox Functions" (MATLAB Coder).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see "Run MATLAB Functions with Distributed Arrays" (Parallel Computing Toolbox).

See Also

`maxk` | `min` | `topkrows`

Introduced in R2017b

minres

Minimum residual method

Syntax

```
x = minres(A,b)
minres(A,b,tol)
minres(A,b,tol,maxit)
minres(A,b,tol,maxit,M)
minres(A,b,tol,maxit,M1,M2)
minres(A,b,tol,maxit,M1,M2,x0)
[x,flag] = minres(A,b,...)
[x,flag,relres] = minres(A,b,...)
[x,flag,relres,iter] = minres(A,b,...)
[x,flag,relres,iter,resvec] = minres(A,b,...)
[x,flag,relres,iter,resvec,resveccg] = minres(A,b,...)
```

Description

`x = minres(A,b)` attempts to find a minimum norm residual solution `x` to the system of linear equations `A*x=b`. The `n`-by-`n` coefficient matrix `A` must be symmetric but need not be positive definite. It should be large and sparse. The column vector `b` must have length `n`. You can specify `A` as a function handle, `afun`, such that `afun(x)` returns `A*x`.

“Parameterizing Functions” explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `minres` converges, a message to that effect is displayed. If `minres` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`minres(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `minres` uses the default, `1e-6`.

`minres(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `minres` uses the default, `min(n,20)`.

`minres(A,b,tol,maxit,M)` and `minres(A,b,tol,maxit,M1,M2)` use symmetric positive definite preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(sqrt(M))*A*inv(sqrt(M))*y = inv(sqrt(M))*b` for `y` and then return `x = inv(sqrt(M))*y`. If `M` is `[]` then `minres` applies no preconditioner. `M` can be a function handle `mfun`, such that `mfun(x)` returns `M\x`.

`minres(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is `[]`, then `minres` uses the default, an all-zero vector.

`[x,flag] = minres(A,b,...)` also returns a convergence flag.

Flag	Convergence
`0`	`minres` converged to the desired tolerance `tol` within `maxit` iterations.
`1`	`minres` iterated `maxit` times but did not converge.
`2`	Preconditioner `M` was ill-conditioned.
`3`	`minres` stagnated. (Two consecutive iterates were the same.)
`4`	One of the scalar quantities calculated during `minres` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = minres(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = minres(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = minres(A,b,...)` also returns a vector of estimates of the `minres` residual norms at each iteration, including `norm(b-A*x0)`.

`[x,flag,relres,iter,resvec,resveccg] = minres(A,b,...)` also returns a vector of estimates of the Conjugate Gradients residual norms at each iteration.

Examples

Using minres with a Matrix Input

```
n = 100; on = ones(n,1);
A = spdiags([-2*on 4*on -2*on],-1:1,n,n);
b = sum(A,2);
tol = 1e-10;
maxit = 50;
M1 = spdiags(4*on,0,n,n);

x = minres(A,b,tol,maxit,M1);
minres converged at iteration 49 to a solution with relative
residual 4.7e-014
```

Using minres with a Function Handle

This example replaces the matrix `A` in the previous example with a handle to a matrix-vector product function `afun`. The example is contained in a file `run_minres` that

- Calls `minres` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_minres` are available to `afun`.

The following shows the code for `run_minres`:

```
function x1 = run_minres
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -2*on],-1:1,n,n);
b = sum(A,2);
tol = 1e-10;
maxit = 50;
M = spdiags(4*on,0,n,n);
x1 = minres(@afun,b,tol,maxit,M);

       function y = afun(x)
          y = 4 * x;
          y(2:n) = y(2:n) - 2 * x(1:n-1);
          y(1:n-1) = y(1:n-1) - 2 * x(2:n);
       end
end
```

When you enter

```
x1=run_minres;
```

MATLAB software displays the message

```
minres converged at iteration 49 to a solution with relative
residual 4.7e-014
```

Using minres instead of pcg

Use a symmetric indefinite matrix that fails with `pcg`.

```
A = diag([20:-1:1, -1:-1:-20]);
b = sum(A,2);      % The true solution is the vector of all ones.
x = pcg(A,b);      % Errors out at the first iteration.
```

displays the following message:

```
pcg stopped at iteration 1 without converging to the desired
tolerance 1e-006 because a scalar quantity became too small or
too large to continue computing.
The iterate returned (number 0) has relative residual 1
```

However, `minres` can handle the indefinite matrix `A`.

```
x = minres(A,b,1e-6,40);
minres converged at iteration 39 to a solution with relative
residual 1.3e-007
```

References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Paige, C. C. and M. A. Saunders, “Solution of Sparse Indefinite Systems of Linear Equations.” *SIAM J. Numer. Anal.*, Vol.12, 1975, pp. 617-629.

Extended Capabilities

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- If `M1` is a function, then it is applied independently to each row.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bicg` | `bicgstab` | `cgs` | `gmres` | `ichol` | `lsqr` | `mldivide` | `pcg` | `qmr` | `symmlq`

Topics

“Create Function Handle”

Introduced before R2006a

minus, -

Subtraction

Syntax

```
C = A - B
C = minus(A,B)
```

Description

`C = A - B` subtracts array `B` from array `A` by subtracting corresponding elements. The sizes of `A` and `B` must be the same or be compatible.

If the sizes of `A` and `B` are compatible, then the two arrays implicitly expand to match each other. For example, if `A` or `B` is a scalar, then the scalar is combined with each element of the other array. Also, vectors with different orientations (one row vector and one column vector) implicitly expand to form a matrix.

`C = minus(A,B)` is an alternate way to execute `A - B`, but is rarely used. It enables operator overloading for classes.

Examples

Subtract Scalar from Array

Create an array, `A`, and subtract a scalar value from it.

```
A = [2 1; 3 5];
C = A - 2
```

```
C = 2×2

     0    -1
     1     3
```

The scalar is subtracted from each entry of A.

Subtract Two Arrays

Create two arrays, A and B, and subtract the second, B, from the first, A.

```
A = [1 0; 2 4];
B = [5 9; 2 1];
C = A - B
```

```
C = 2×2

    -4    -9
     0     3
```

The elements of B are subtracted from the corresponding elements of A.

Use the syntax -C to negate the elements of C.

```
-C
```

```
ans = 2×2

     4     9
     0    -3
```

Subtract Row and Column Vectors

Create a 1-by-2 row vector and 3-by-1 column vector and subtract them.

```
a = 1:2;
b = (1:3)';
a - b
```

```
ans = 3×2

     0     1
    -1     0
```

```
    -2    -1
```

The result is a 3-by-2 matrix, where each (i,j) element in the matrix is equal to `a(j) - b(i)`:

$$a = [a_1\ a_2], \qquad b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}, \qquad a - b = \begin{bmatrix} a_1 - b_1 & a_2 - b_1 \\ a_1 - b_2 & a_2 - b_2 \\ a_1 - b_3 & a_2 - b_3 \end{bmatrix}.$$

Subtract Mean from Matrix

Create a matrix, `A`. Scale the elements in each column by subtracting the mean.

```
A = [1 9 3; 2 7 8]
```

```
A = 2×3

     1     9     3
     2     7     8
```

```
A - mean(A)
```

```
ans = 2×3

   -0.5000    1.0000   -2.5000
    0.5000   -1.0000    2.5000
```

Input Arguments

A, B — Operands
scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Numeric inputs `A` and `B` must either be the same size or have sizes that are compatible (for

example, `A` is an `M`-by-`N` matrix and `B` is a scalar or `1`-by-`N` row vector). For more information, see “Compatible Array Sizes for Basic Operations”.

- Operands with an integer data type cannot be complex.
- Datetime, duration, and calendar duration arrays must be the same size unless one is a scalar.
- If one input is a datetime array, duration array, or calendar duration array, then numeric values in the other input are treated as a number of 24-hour days.
- If one input is a string array, then the other input can be a numeric, logical, character, string, or cell array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `datetime` | `duration` | `calendarDuration`
Complex Number Support: Yes

Compatibility Considerations

Implicit expansion change affects arguments for operators

Behavior changed in R2016b

Starting in R2016b with the addition of implicit expansion, some combinations of arguments for basic operations that previously returned errors now produce results. For example, you previously could not add a row and a column vector, but those operands are now valid for addition. In other words, an expression like `[1 2] + [1; 2]` previously returned a size mismatch error, but now it executes.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a `try`/`catch` block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see “Compatible Array Sizes for Basic Operations”.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- 64-bit integers are not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`diff` | `plus` | `uminus`

Topics

"Array vs. Matrix Operations"
"Operator Precedence"
"MATLAB Operators and Special Characters"

Introduced before R2006a

minute

Minute number

Syntax

```
m = minute(t)
```

Description

`m = minute(t)` returns the minute numbers of the datetime values in `t`. The `m` output is a `double` array the same size as `t` and contains integer values from 0 to 59.

The `minute` function returns the minute numbers of datetime values. To assign minute values to a datetime array, `t`, use `t.Minute` and modify the `Minute` property.

Examples

Find Minute Number of Datetime Values

```
t1 = datetime('now');
t = t1 + minutes(2:4)
```

```
t = 1x3 datetime array
   26-Aug-2019 18:09:47   26-Aug-2019 18:10:47   26-Aug-2019 18:11:47
```

```
m = minute(t)
```

```
m = 1×3

     9    10    11
```

Input Arguments

t — Input date and time
`datetime` array

Input date and time, specified as a `datetime` array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`datetime` | `hms` | `hour` | `second` | `timeofday`

Introduced in R2014b

minutes

Duration in minutes

Syntax

```
M = minutes(X)
```

Description

`M = minutes(X)` returns an array of minutes equivalent to the values in `X`.

- If `X` is a numeric array, then `M` is a `duration` array in units of minutes.
- If `X` is a `duration` array, then `M` is a `double` array with each element equal to the number of minutes in the corresponding element of `X`.

The `minutes` function converts between `duration` and `double` values. To display a duration in units of minutes, set its `Format` property to `'m'`.

Examples

Create Duration Array of Minutes

```
X = magic(4);
M = minutes(X)
```

```
M = 4x4 duration array
   16 min    2 min    3 min   13 min
    5 min   11 min   10 min    8 min
    9 min    7 min    6 min   12 min
    4 min   14 min   15 min    1 min
```

Convert Durations to Numeric Array of Minutes

Create a `duration` array.

```
X = hours(2:10:38) + minutes(30)
```

```
X = 1x4 duration array
    2.5 hr   12.5 hr   22.5 hr   32.5 hr
```

Convert each duration in `X` to a number of minutes.

```
M = minutes(X)
```

```
M = 1×4

         150         750        1350        1950
```

View the data type of `M`.

```
whos M
```

```
  Name      Size            Bytes  Class     Attributes

  M         1x4                32  double
```

Input Arguments

X — Input array
numeric array | duration array | logical array

Input array, specified as a numeric array, duration array, or logical array.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This function fully supports code generation. For more information, see “Code Generation for Duration Arrays” (MATLAB Coder).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`duration`

Introduced in R2014b

mislocked

Determine if function or script is locked in memory

Syntax

```
tf = mislocked(fun)
tf = mislocked
```

Description

`tf = mislocked(fun)` returns `true` (logical `1`) if the function or script `fun` is locked in memory. Otherwise, it returns `false` (logical 0).

`tf = mislocked` returns `true` (logical `1`) if the currently running function is locked in memory. Otherwise, it returns `false` (logical 0).

Use this syntax only within a MATLAB code file.

Examples

Determine If Function Is Locked

Check if the function `plot` is locked in memory.

```
tf = mislocked('plot')
```

```
tf =

  logical

   0
```

Determine If Currently Running Function Is Locked

Create the function `myFun` in your current working folder. The function calls `mislocked` on itself to find out if it is locked in memory before and after a call to `mlock`.

```
function myFun()
    isLockedBefore = mislocked
    mlock
    isLockedAfter = mislocked
end
```

At the command prompt, call the function.

```
myFun
```

```
isLockedBefore =

  logical

   0


isLockedAfter =

  logical

   1
```

Unlock the function so it can be cleared from memory. Verify that the function is not locked.

```
munlock('myFun')
tf = mislocked('myFun')
```

```
tf =

  logical

   0
```

Input Arguments

`fun` — Name of function or script
character vector | string scalar

Name of function or script, specified as a character vector or a string scalar.

Tips

- To remove a locked function from memory, first unlock it using the `munlock` function, and then clear it from memory using the `clear` function.
- To determine if a MEX file is locked in memory, use the `mexIsLocked` function.

See Also

`inmem` | `mlock` | `munlock`

Introduced before R2006a

missing

Create missing values

Syntax

```
m = missing
```

Description

`m = missing` returns a missing value displayed as `<missing>`. You can set an element of an array or table to `missing` to represent missing data. The value of `missing` is then automatically converted to the standard missing value native to the data type of the array or table variable. For example, core data types that support `missing` and their corresponding standard missing values after assignment are as follows:

- `double`, `single`, `duration`, and `calendarDuration` convert `missing` to `NaN`
- `datetime` converts `missing` to `NaT`
- `categorical` converts `missing` to `<undefined>`
- `string` converts `missing` to `<missing>`

Examples

Missing Data in Timetable

Create a timetable containing weather data, and replace the last row with missing values. Each `missing` value is automatically replaced with the standard missing value for the relevant data type.

```
Time = datetime({'2015-12-18 08:03:05';'2015-12-18 10:03:17';'2015-12-18 12:03:13'});
Temp = [37.3;39.1;42.3];
WindDirection = categorical({'NW';'N';'NW'});
TT = timetable(Time,Temp,WindDirection)
```

```
TT=3×3 timetable
            Time            Temp    WindDirection
    ____________________    ____    _____________

    18-Dec-2015 08:03:05    37.3         NW
    18-Dec-2015 10:03:17    39.1         N
    18-Dec-2015 12:03:13    42.3         NW
```

```
TT.Time(3) = missing;
TT.Temp(3) = missing;
TT.WindDirection(3) = missing;

TT
```

```
TT=3×3 timetable
            Time            Temp    WindDirection
    ____________________    ____    _____________

    18-Dec-2015 08:03:05    37.3     NW
    18-Dec-2015 10:03:17    39.1     N
    NaT                      NaN     <undefined>
```

See Also

`fillmissing` | `ismissing` | `standardizeMissing`

Topics

"Missing Data in MATLAB"

Introduced in R2017a

Clean Missing Data

Find, fill, or remove missing data in the Live Editor

Description

The **Clean Missing Data** task lets you interactively handle missing data values such as `NaN` or `<missing>`. The task automatically generates MATLAB code for your live script.

Using this task, you can:

- Find, fill, or remove missing data in a workspace variable.
- Customize the method for filling data.
- Automatically visualize the missing data and the cleaned data.

Open the Task

To add the **Clean Missing Data** task to a live script in the MATLAB Editor:

- On the **Live Editor** tab, select **Task > Clean Missing Data**.
- In a code block in the script, type a relevant keyword, such as `missing` or `remove`. Select **Clean Missing Data** from the suggested command completions.

Parameters

`Fill method` — Method for filling missing data

`Linear interpolation` (default) | `Constant value` | `Previous value` | `Next value` | `Nearest value` | `Spline interpolation` | `Shape-preserving cubic interpolation (PCHIP)` | `Modified Akima cubic interpolation` | `Moving median` | `Moving mean` | ...

Specify the method for filling missing data using one of the following options.

Method	Description
`Linear interpolation`	Linear interpolation of neighboring, nonmissing values.
`Constant value`	Specified constant value.
`Previous value`	Previous nonmissing value.
`Next value`	Next nonmissing value.
`Nearest value`	Nearest nonmissing value.
`Spline interpolation`	Piecewise cubic spline interpolation.
`Shape-preserving cubic interpolation`	Shape-preserving piecewise cubic spline interpolation.
`Modified Akima cubic interpolation`	Modified Akima cubic Hermite interpolation.
`Moving median`	Moving median with specified window size.
`Moving mean`	Moving mean with specified window size.

`Window` — Window for moving methods
`Centered` (default) | `Asymmetric`

Specify the window type and size when the method for filling missing data is `Moving median` or `Moving mean`.

Window	Description
`Centered`	Specified window length centered about the current point
`Asymmetric`	Specified window containing the number of elements before the current point and the number of elements after the current point

Window sizes are relative to the **X-axis** variable units.

See Also

Functions
`fillmissing` | `ismissing` | `rmmissing`

Live Editor Tasks
Clean Outlier Data | Find Change Points | Find Local Extrema | Remove Trends | Smooth Data

Topics

"Add Interactive Tasks to a Live Script"
Clean Messy Data and Locate Extrema Using Live Editor Tasks
"Missing Data in MATLAB"

Introduced in R2019b

mkdir

Make new folder

Syntax

```
mkdir folderName
mkdir parentFolder folderName

status = mkdir( ___ )
[status,msg] = mkdir( ___ )
[status,msg,msgID] = mkdir( ___ )
```

Description

`mkdir folderName` creates the folder `folderName`. If `folderName` exists, MATLAB issues a warning. If the operation is not successful, `mkdir` throws an error to the Command Window.

`mkdir parentFolder folderName` creates `folderName` in `parentFolder`. If `parentFolder` does not exist, MATLAB attempts to create it.

`status = mkdir( ___ )` creates the specified folder and returns a status of `1` if the operation is successful or if the folder exists. Otherwise, `mkdir` returns `0` and does *not* throw a warning or error to the Command Window. You can use this syntax with any of the input argument combinations in the previous syntaxes.

`[status,msg] = mkdir( ___ )` also returns the message text for any warning or error that occurs.

`[status,msg,msgID] = mkdir( ___ )` additionally returns the message ID for any warning or error that occurs.

Examples

Create Subfolder in Current Folder

Create a folder called `newdir` in the current folder.

```
mkdir newdir
```

Create Subfolder in Specified Parent Folder

Create a folder called `newfolder` in the folder `testdata`. Use a relative path, where `newFolder` is at the same level as the current folder.

```
mkdir ../testdata newFolder
```

Create Folder That Already Exists

Create the same folder twice, verifying the status of the operation after each try.

Create the folder `newFolder`. The operation succeeds, returning a status of 1 with no error or warning message.

```
[status, msg, msgID] = mkdir('newFolder')
```

```
status = logical
   1


msg =

  0x0 empty char array


msgID =

  0x0 empty char array
```

Create the folder `newFolder` again. The operation succeeds again, returning a status of 1. A warning message and message ID inform you that the folder already exists.

```
[status, msg, msgID] = mkdir('newFolder')
```

```
status = logical
   1


msg =
'Directory already exists.'

msgID =
'MATLAB:MKDIR:DirectoryExists'
```

Input Arguments

`folderName` — Folder name
character vector | string scalar

Folder name, specified as a character vector or string scalar. You can specify `folderName` as an absolute or relative path, unless a parent folder is specified. If you specify a parent folder, then `folderName` must be a path relative to the parent folder.

If `folderName` contains a path that includes one or more nonexistent folders, MATLAB attempts to create the nonexistent folder. For example, for the path `myFolder\folder1\folder2\targetFolder`, if `folder1` does not exist, MATLAB creates `folder1`, creates `folder2` within `folder1`, and creates `targetFolder` within `folder2`.

Data Types: `char` | `string`

`parentFolder` — Parent folder
character vector | string scalar

Parent folder for the new folder, specified as a character vector or string scalar. Specify `parentFolder` as an absolute or relative path. If `parentFolder` does not exist, MATLAB attempts to create it.

Data Types: `char` | `string`

Output Arguments

`status` — Folder creation status
0 | 1

Folder creation status indicating whether the attempt to create the folder is successful, returned as `0` or `1`. If the attempt to create the folder is successful or the folder already exists, then the value of `status` is 1. Otherwise, the value is 0.

Data Types: `logical`

`msg` — Error message
character vector

Error message, returned as a character vector. If an error or warning occurs, `msg` contains the message text of the error or warning. Otherwise, `msg` is empty, `''`.

`msgID` — Error message identifier
character vector

Error message identifier, returned as a character vector. If an error or warning occurs, `msgID` contains the message identifier of the error or warning. Otherwise, `msgID` is empty, `''`.

See Also

`cd` | `copyfile` | `dir` | `movefile` | `rmdir`

Topics

"Manage Files and Folders"

Introduced before R2006a

mkdir

Make new folder on FTP server

Syntax

```
mkdir(ftpobj,folder)
```

Description

`mkdir(ftpobj,folder)` makes the specified folder on the FTP server associated with `ftpobj`.

Examples

Make Folder

Connect to an FTP server and make a folder. Navigate to that folder and upload a file. This example shows a hypothetical FTP session on `ftp.example.com`, a machine that does not exist. If you access an FTP server that grants you permission to upload files and make folders on that server, then you can use the `mkdir` function as shown in this example.

First, connect to the server.

```
ftpobj = ftp('ftp.example.com')
```

```
ftpobj = 

  FTP Object
     host: ftp.example.com
     user: anonymous
      dir: /
     mode: binary
```

Display the contents of the current folder on the FTP server.

```
dir(ftpobj)
```

```
myscript.m                  README.txt                pub
```

Make a folder named `scripts`.

```
mkdir(ftpobj,'scripts')
```

Display the updated contents of the current folder, including the new `scripts` folder.

```
dir(ftpobj)
```

```
myscript.m                  README.txt                pub
scripts
```

Navigate to the `scripts` folder and upload a file.

```
cd(ftpobj,'scripts');
mput(ftpobj,'myNewScript.m')
```

Input Arguments

`ftpobj` — Connection to FTP server
FTP object

Connection to an FTP server, specified as an FTP object.

`folder` — New folder
character vector | string scalar

New folder on the FTP server, specified as a character vector or string scalar.

See Also

`dir` | `ftp` | `rmdir`

Introduced before R2006a

mkpp

Make piecewise polynomial

Syntax

```
pp = mkpp(breaks,coefs)
pp = mkpp(breaks,coefs,d)
```

Description

`pp = mkpp(breaks,coefs)` builds a piecewise polynomial `pp` from its breaks and coefficients. Use `ppval` to evaluate the piecewise polynomial at specific points, or `unmkpp` to extract details about the piecewise polynomial.

`pp = mkpp(breaks,coefs,d)` specifies that the piecewise polynomial is vector-valued, such that the value of each of its coefficients is a vector of length `d`.

Examples

Create Piecewise Polynomial with Polynomials of Several Degrees

Create a piecewise polynomial that has a cubic polynomial in the interval [0,4], a quadratic polynomial in the interval [4,10], and a quartic polynomial in the interval [10,15].

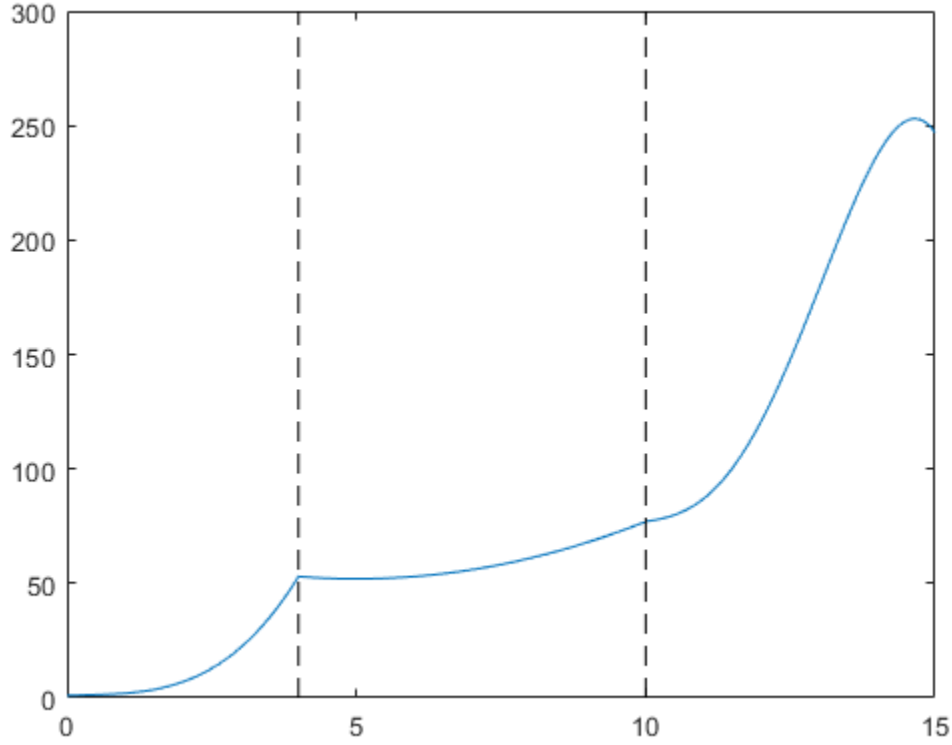
```
breaks = [0 4 10 15];
coefs = [0 1 -1 1 1; 0 0 1 -2 53; -1 6 1 4 77];
pp = mkpp(breaks,coefs)
```

```
pp = struct with fields:
      form: 'pp'
    breaks: [0 4 10 15]
     coefs: [3x5 double]
    pieces: 3
```

```
     order: 5
       dim: 1
```

Evaluate the piecewise polynomial at many points in the interval [0,15] and plot the results. Plot vertical dashed lines at the break points where the polynomials meet.

```
xq = 0:0.01:15;
plot(xq,ppval(pp,xq))
line([4 4],ylim,'LineStyle','--','Color','k')
line([10 10],ylim,'LineStyle','--','Color','k')
```



Create Piecewise Polynomial with Repeated Pieces

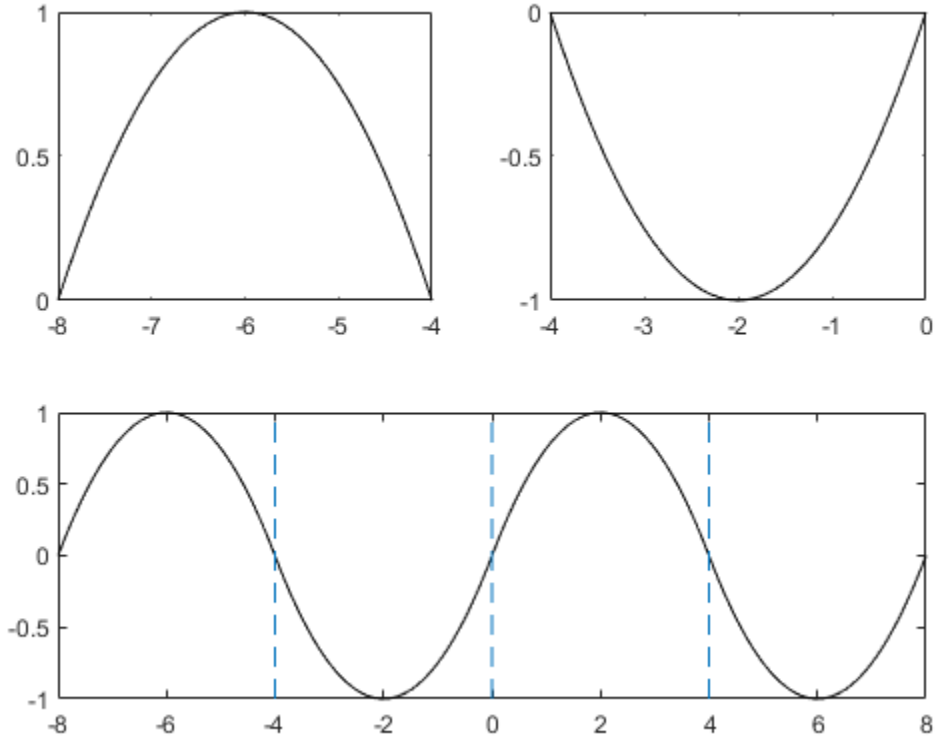
Create and plot a piecewise polynomial with four intervals that alternate between two quadratic polynomials.

The first two subplots show a quadratic polynomial and its negation shifted to the intervals [-8,-4] and [-4,0]. The polynomial is

$$1 - \left(\frac{x}{2} - 1\right)^2 = \frac{-x^2}{4} + x\,.$$

The third subplot shows a piecewise polynomial constructed by alternating these two quadratic pieces over four intervals. Vertical lines are added to show the points where the polynomials meet.

```
subplot(2,2,1)
cc = [-1/4 1 0];
pp1 = mkpp([-8 -4],cc);
xx1 = -8:0.1:-4;
plot(xx1,ppval(pp1,xx1),'k-')

subplot(2,2,2)
pp2 = mkpp([-4 0],-cc);
xx2 = -4:0.1:0;
plot(xx2,ppval(pp2,xx2),'k-')

subplot(2,1,2)
pp = mkpp([-8 -4 0 4 8],[cc;-cc;cc;-cc]);
xx = -8:0.1:8;
plot(xx,ppval(pp,xx),'k-')
hold on
line([-4 -4],ylim,'LineStyle','--')
line([0 0],ylim,'LineStyle','--')
line([4 4],ylim,'LineStyle','--')
hold off
```



Input Arguments

`breaks` — Break points
vector

Break points, specified as a vector of length `L+1` with strictly increasing elements that represent the start and end of each of `L` intervals.

Data Types: `single` | `double`

`coefs` — Polynomial coefficients
matrix

Polynomial coefficients, specified as an L-by-k matrix with the ith row `coefs(i,:)` containing the local coefficients of an order k polynomial on the ith interval, `[breaks(i), breaks(i+1)]`. In other words, the polynomial is `coefs(i,1)*(X-breaks(i))^(k-1) + coefs(i,2)*(X-breaks(i))^(k-2) + ... + coefs(i,k-1)*(X-breaks(i)) + coefs(i,k)`.

Data Types: `single` | `double`

d — Dimension
scalar | vector

Dimension, specified as a scalar or vector of integers. Specify `d` to signify that the piecewise polynomial has coefficient values of size `d`.

Data Types: `single` | `double`

Output Arguments

pp — Piecewise polynomial
structure

Piecewise polynomial, returned as a structure. Use this structure with the `ppval` function to evaluate the piecewise polynomial at one or more query points. The structure has these fields.

Field	Description
`form`	`'pp'` for *piecewise polynomial*
`breaks`	Vector of length `L+1` with strictly increasing elements that represent the start and end of each of `L` intervals
`coefs`	`L`-by-`k` matrix with each row `coefs(i,:)` containing the local coefficients of an order `k` polynomial on the ith interval, `[breaks(i),breaks(i+1)]`
`pieces`	Number of pieces, `L`
`order`	Order of the polynomials
`dim`	Dimensionality of target

Since the polynomial coefficients in `coefs` are local coefficients for each interval, you must subtract the lower endpoint of the corresponding knot interval to use the coefficients in a conventional polynomial equation. In other words, for the coefficients `[a,b,c,d]` on the interval `[x1,x2]`, the corresponding polynomial is

$$f(x) = a(x - x_1)^3 + b(x - x_1)^2 + c(x - x_1) + d \ .$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The output structure `pp` differs from the `pp` structure in MATLAB. In MATLAB, `ppval` cannot use the `pp` structure from the code generator. For code generation, `ppval` cannot use a `pp` structure created by MATLAB. `unmkpp` can use a MATLAB `pp` structure for code generation.

  To create a MATLAB `pp` structure from a `pp` structure created by the code generator:

  - In code generation, use `unmkpp` to return the piecewise polynomial details to MATLAB.
  - In MATLAB, use `mkpp` to create the `pp` structure.
- If you do not provide `d`, then `coefs` must be two-dimensional and have a fixed number of columns. In this case, the number of columns is the order.
- To define a piecewise constant polynomial, `coefs` must be a column vector or `d` must have at least two elements.
- If you provide `d` and `d` is `1`, then `d` must be a constant. Otherwise, if the input to `ppval` is nonscalar, then the shape of the output of `ppval` can differ from `ppval` in MATLAB.
- If you provide `d`, then it must have a fixed length. One of the following sets of statements must be true:

  1 Suppose that `m = length(d)` and `npieces = length(breaks) - 1`.

```
size(coefs,j) = d(j)
size(coefs,m+1) = npieces
size(coefs,m+2) = order
```

`j` = 1,2,...,m. The dimension `m+2` must be fixed length.

2. Suppose that `m = length(d)` and `npieces = length(breaks) - 1`.

```
size(coefs,1) = prod(d)*npieces
size(coefs,2) = order
```

The second dimension must be fixed length.

- If you do not provide `d`, then the following statements must be true:

  Suppose that `m = length(d)` and `npieces = length(breaks) - 1`.

  ```
  size(coefs,1) = prod(d)*npieces
  size(coefs,2) = order
  ```

  The second dimension must be fixed length.

See Also

`pchip` | `ppval` | `spline` | `unmkpp`

Introduced before R2006a

mldivide, \

Solve systems of linear equations $Ax = B$ for x

Syntax

```
x = A\B
x = mldivide(A,B)
```

Description

`x = A\B` solves the system of linear equations `A*x = B`. The matrices `A` and `B` must have the same number of rows. MATLAB displays a warning message if `A` is badly scaled or nearly singular, but performs the calculation regardless.

- If `A` is a scalar, then `A\B` is equivalent to `A.\B`.
- If `A` is a square `n`-by-`n` matrix and `B` is a matrix with `n` rows, then `x = A\B` is a solution to the equation `A*x = B`, if it exists.
- If `A` is a rectangular `m`-by-`n` matrix with `m ~= n`, and `B` is a matrix with `m` rows, then `A\B` returns a least-squares solution to the system of equations `A*x= B`.

`x = mldivide(A,B)` is an alternative way to execute `x = A\B`, but is rarely used. It enables operator overloading for classes.

Examples

System of Equations

Solve a simple system of linear equations, `A*x = B`.

```
A = magic(3);
B = [15; 15; 15];
x = A\B
```

```
x = 3×1

    1.0000
    1.0000
    1.0000
```

Linear System with Singular Matrix

Solve a linear system of equations `A*x = b` involving a singular matrix, `A`.

```
A = magic(4);
b = [34; 34; 34; 34];
x = A\b
```

```
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND

x = 4×1

    1.5000
    2.5000
   -0.5000
    0.5000
```

When `rcond` is between `0` and `eps`, MATLAB® issues a nearly singular warning, but proceeds with the calculation. When working with ill-conditioned matrices, an unreliable solution can result even though the residual `(b-A*x)` is relatively small. In this particular example, the norm of the residual is zero, and an exact solution is obtained, although `rcond` is small.

When `rcond` is equal to `0`, the singular warning appears.

```
A = [1 0; 0 0];
b = [1; 1];
x = A\b
```

```
Warning: Matrix is singular to working precision.

x = 2×1

     1
```

```
   Inf
```

In this case, division by zero leads to computations with `Inf` and/or `NaN`, making the computed result unreliable.

Least-Squares Solution of Underdetermined System

Solve a system of linear equations, `A*x = b`.

```
A = [1 2 0; 0 4 3];
b = [8; 18];
x = A\b
```

```
x = 3×1

         0
    4.0000
    0.6667
```

Linear System with Sparse Matrix

Solve a simple system of linear equations using sparse matrices.

Consider the matrix equation `A*x = B`.

```
A = sparse([0 2 0 1 0; 4 -1 -1 0 0; 0 0 0 3 -6; -2 0 0 0 2; 0 0 4 2 0]);
B = sparse([8; -1; -18; 8; 20]);
x = A\B
```

```
x =
   (1,1)       1.0000
   (2,1)       2.0000
   (3,1)       3.0000
   (4,1)       4.0000
   (5,1)       5.0000
```

Input Arguments

A, B — Operands
vectors | full matrices | sparse matrices

Operands, specified as vectors, full matrices, or sparse matrices. A and B must have the same number of rows.

- If A or B has an integer data type, the other input must be scalar. Operands with an integer data type cannot be complex.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

Output Arguments

x — Solution
vector | full matrix | sparse matrix

Solution, returned as a vector, full matrix, or sparse matrix. If A is an m-by-n matrix and B is an m-by-p matrix, then x is an n-by-p matrix, including the case when p==1.

If A has full storage, x is also full. If A is sparse, then x has the same storage as B.

Tips

- The operators / and \ are related to each other by the equation `B/A = (A'\B')'`.
- If A is a square matrix, then A\B is roughly equal to `inv(A)*B`, but MATLAB processes A\B differently and more robustly.
- If the rank of A is less than the number of columns in A, then `x = A\B` is not necessarily the minimum norm solution. You can compute the minimum norm least-squares solution using `x = lsqminnorm(A,B)` or `x = pinv(A)*B`.

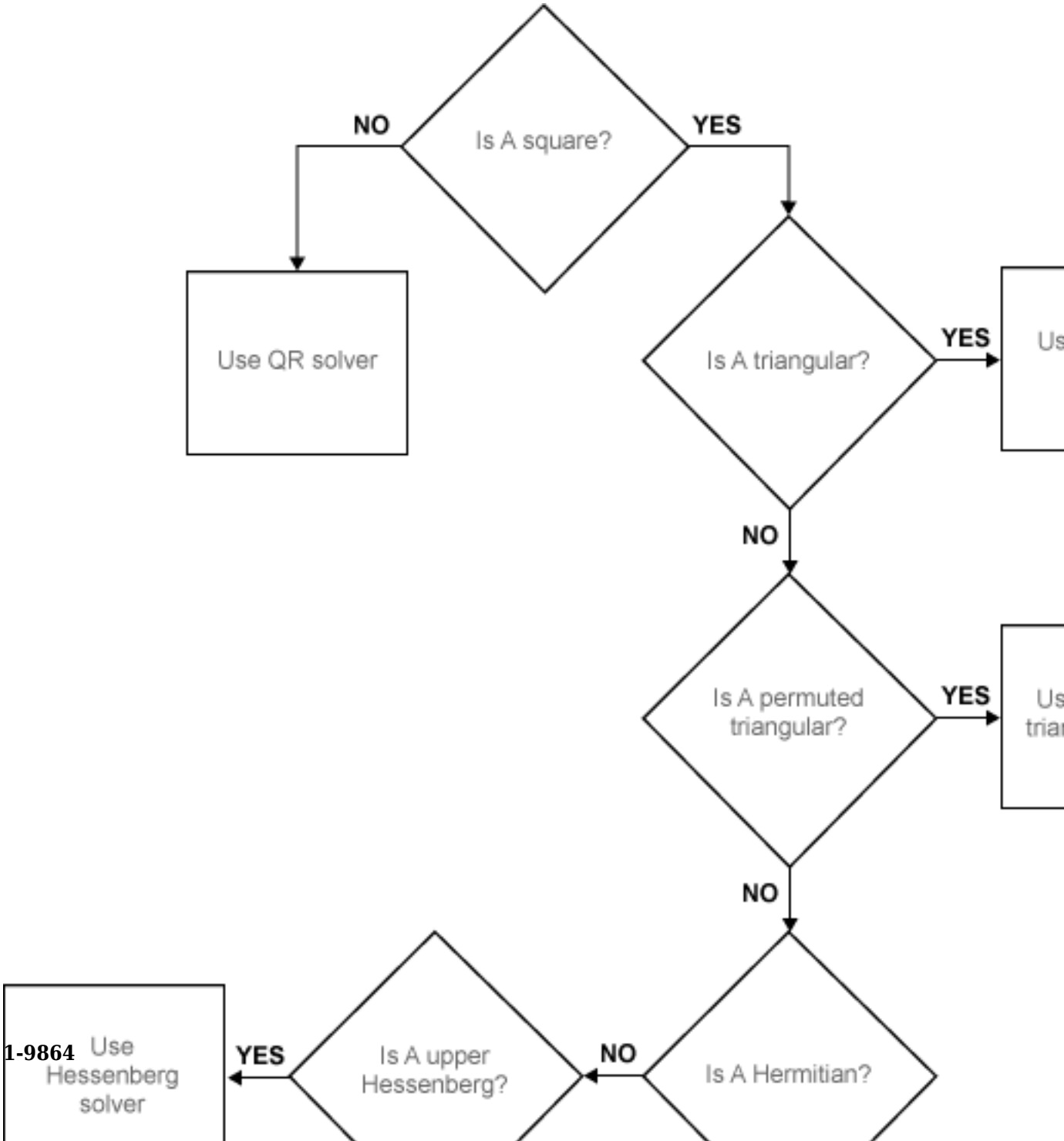
Algorithms

The versatility of `mldivide` in solving linear systems stems from its ability to take advantage of symmetries in the problem by dispatching to an appropriate solver. This

approach aims to minimize computation time. The first distinction the function makes is between *full* (also called "*dense*") and *sparse* input arrays.

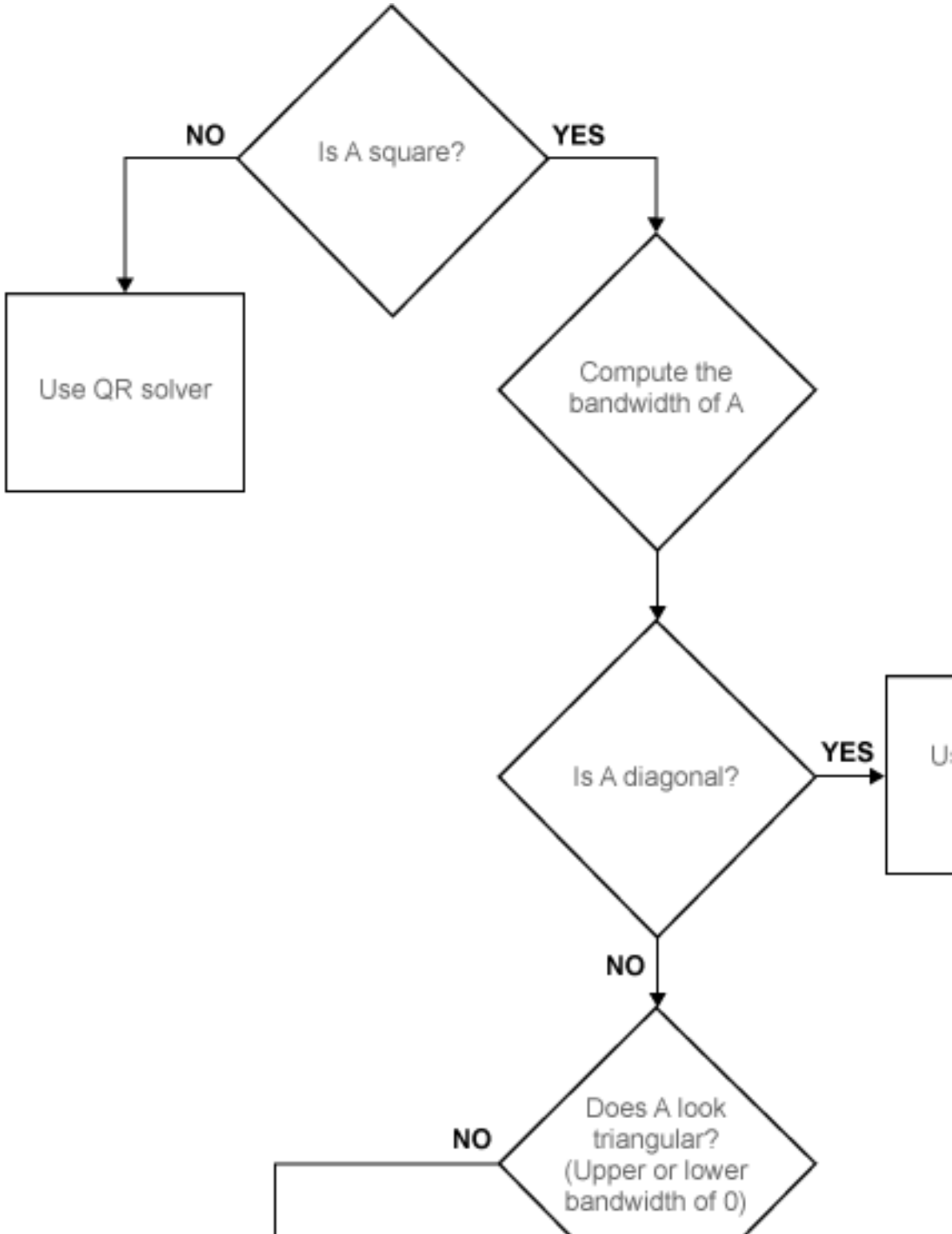
Algorithm for Full Inputs

The flow chart below shows the algorithm path when inputs A and B are **full**.


Is A square?
NO
YES
Use QR solver
Is A triangular?
YES
NO
Is A permuted triangular?
YES
NO
Is A Hermitian?
NO
Is A upper Hessenberg?
YES
Use Hessenberg solver

Algorithm for Sparse Inputs

If `A` is full and `B` is sparse then `mldivide` converts `B` to a full matrix and uses the full algorithm path (above) to compute a solution with full storage. If `A` is sparse, the storage of the solution `x` is the same as that of `B` and `mldivide` follows the algorithm path for **sparse** inputs, shown below.


Is A square?
NO
YES
Use QR solver
Compute the bandwidth of A
Is A diagonal?
YES
NO
Does A look triangular? (Upper or lower bandwidth of 0)
NO

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitation:

For the syntax `Z = X\Y`, the array `X` must be a scalar or a tall matrix with the same number of rows as `Y`.

For more information, see “Tall Arrays for Out-of-Memory Data”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For sparse matrix inputs, the standard math library must be C99 or later.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For sparse matrix inputs, the standard math library must be C99 or later.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- If `A` is rectangular, then it must also be nonsparse.

- The MATLAB `mldivide` function prints a warning if `A` is badly scaled, nearly singular, or rank deficient. The `gpuArray` `mldivide` is unable to check for this condition. Take action to avoid this condition.
- 64-bit integers are not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- If `A` is rectangular, then it must also be nonsparse.
- The MATLAB `mldivide` function prints a warning if `A` is badly scaled, nearly singular, or rank deficient. The distributed array `mldivide` is unable to check for this condition. Take action to avoid this condition.
- If `A` is an M-by-N matrix with N > M, for distributed arrays, `mldivide` computes a solution that minimizes `norm(X)`. The result is the same as the result of `PINV(A)*B`.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`chol` | `inv` | `ldivide` | `ldl` | `linsolve` | `lsqminnorm` | `lu` | `mrdivide` | `pinv` | `qr` | `rdivide` | `spparms`

Topics

“Array vs. Matrix Operations”
“Operator Precedence”
“Systems of Linear Equations”
“Operator Overloading”
“MATLAB Operators and Special Characters”

Introduced before R2006a

mrdivide, /

Solve systems of linear equations $xA = B$ for x

Syntax

```
x = B/A
x = mrdivide(B,A)
```

Description

`x = B/A` solves the system of linear equations `x*A = B` for `x`. The matrices `A` and `B` must contain the same number of columns. MATLAB displays a warning message if `A` is badly scaled or nearly singular, but performs the calculation regardless.

- If `A` is a scalar, then `B/A` is equivalent to `B./A`.
- If `A` is a square `n`-by-`n` matrix and `B` is a matrix with `n` columns, then `x = B/A` is a solution to the equation `x*A = B`, if it exists.
- If `A` is a rectangular `m`-by-`n` matrix with `m ~= n`, and `B` is a matrix with `n` columns, then `x = B/A` returns a least-squares solution of the system of equations `x*A = B`.

`x = mrdivide(B,A)` is an alternative way to execute `x = B/A`, but is rarely used. It enables operator overloading for classes.

Examples

System of Equations

Solve a system of equations that has a unique solution, `x*A = B`.

```
A = [1 1 3; 2 0 4; -1 6 -1];
B = [2 19 8];
x = B/A
```

```
x = 1×3

    1.0000    2.0000    3.0000
```

Least-Squares on an Underdetermined System

Solve an underdetermined system, `x*C = D`.

```
C = [1 0; 2 0; 1 0];
D = [1 2];
x = D/C
```

```
Warning: Rank deficient, rank = 1, tol =  1.332268e-15.

x = 1×3

         0    0.5000         0
```

MATLAB® issues a warning but proceeds with calculation.

Verify that `x` is not an exact solution.

```
x*C-D
```

```
ans = 1×2

     0    -2
```

Input Arguments

A, B — Operands
vectors | full matrices | sparse matrices

Operands, specified as vectors, full matrices, or sparse matrices. `A` and `B` must have the same number of columns.

- If A or B has an integer data type, the other input must be scalar. Operands with an integer data type cannot be complex.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char
Complex Number Support: Yes

Output Arguments

x — Solution
vector | full matrix | sparse matrix

Solution, returned as a vector, full matrix, or sparse matrix. If A is an m-by-n matrix and B is a p-by-n matrix, then x is a p-by-m matrix.

x is sparse only if both A and B are sparse matrices.

Tips

- The operators / and \ are related to each other by the equation B/A = (A'\B')'.
- If A is a square matrix, then B/A is roughly equal to B*inv(A), but MATLAB processes B/A differently and more robustly.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitation:

For the syntax Z = X/Y, the Y operand must be a scalar.

For more information, see “Tall Arrays for Out-of-Memory Data”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- If `B` is rectangular, then it must also be nonsparse.
- The MATLAB `mrdivide` function prints a warning if `B` is badly scaled, nearly singular, or rank deficient. The `gpuArray` `mrdivide` is unable to check for this condition. Take action to avoid this condition.
- 64-bit integers are not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- If `B` is rectangular, then it must also be nonsparse.

- The MATLAB `mrdivide` function prints a warning if `B` is badly scaled, nearly singular, or rank deficient. The distributed array `mrdivide` is unable to check for this condition. Take action to avoid this condition.
- If `B` is an M-by-N matrix with N > M, for distributed arrays, `mrdivide` computes a solution that minimizes `norm(X)`. The result is the same as the result of `PINV(B)*A`.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`inv` | `ldivide` | `mldivide` | `rdivide` | `transpose`

Topics

“Array vs. Matrix Operations”
“Operator Precedence”
“Systems of Linear Equations”
“MATLAB Operators and Special Characters”

Introduced before R2006a

mlint

Check MATLAB code files for possible problems

Note `mlint` is not recommended. Use `checkcode` instead.

Alternatives

For information on using the graphical user interface to the Code Analyzer, see “Check Code for Errors and Warnings”.

Syntax

```
mlint('filename')
mlint('filename','-config=settings.txt')
mlint('filename','-config=factory')
inform=mlint('filename','-struct')
msg=mlint('filename','-string')
[inform,filepaths]=mlint('filename')
inform=mlint('filename','-id')
inform=mlint('filename','-fullpath')
inform=mlint('filename','-notok')
mlint('filename','-cyc')
mlint('filename','-codegen')
mlint('filename','-eml')
```

Description

`mlint('filename')` displays messages about `filename` that report potential problems and opportunities for code improvement. These messages are sometimes referred to as Code Analyzer messages. The line number in the message is a hyperlink that you can click to go directly to that line in the Editor. The exact text of the `mlint` messages is subject to some change between versions.

Specify `filename` as one or more character vectors or string arrays, or as a cell array of character vectors. If `filename` specifies multiple character vectors or string arrays, or if `filename` is a nonscalar string array or a cell array of character vectors, MATLAB displays information for each file. You cannot combine cell arrays of character vectors and character vectors of file names. For example, you cannot have `{'lengthofline', 'buggy'}, 'collatz'` as an input.

`mlint('filename','-config=settings.txt')` overrides the default active settings file with the settings that enable or suppress messages as indicated in the specified `settings.txt` file.

Note If used, you must specify the full path to the `settings.txt` file specified with the `-config` option.

For information about creating a `settings.txt` file, see “Save and Reuse Code Analyzer Message Settings”. If you specify an invalid file, `mlint` returns a message indicating that it cannot open or read the file you specified. In that case, `mlint` uses the factory default settings.

`mlint('filename','-config=factory')` ignores all settings files and uses the factory default preference settings.

`inform=mlint('filename','-struct')` returns the information in a structure array whose length is the number of messages found. The structure has the fields that follow.

Field	Description
`message`	Message describing the suspicious construct that code analysis caught.
`line`	Vector of file line numbers to which the message refers.
`column`	Two-column array of file columns (column extents) to which the message applies. The first column of the array specifies the column in the Editor where the message begins. The second column of the array specifies the column in the Editor where the message ends. There is one row in the two-column array for each occurrence of a message.

If you specify multiple file names as input, `inform` contains a cell array of structures.

`msg=mlint('filename','-string')` returns the information as the character vector `msg`. If you specify multiple file names as input, `msg` contains information for each file, separated by 10 equal sign characters (=), a space, the file name, a space, and 10 equal sign characters.

If you omit the **`-struct`** or **`-string`** argument and you specify an output argument, the default behavior is **`-struct`**. If you omit the argument and there are no output arguments, the default behavior is to display the information to the command line.

`[inform,filepaths]=mlint('filename')` additionally returns `filepaths`, the absolute paths to the file names, in the same order as you specified them.

`inform=mlint('filename','-id')` requests the message ID, where ID is a character vector of the form `ABC...`. When returned to a structure, the output also has the `id` field, which is the ID associated with the message.

`inform=mlint('filename',`**`'-fullpath'`**`)` assumes that the input file names are absolute paths, so that `mlint` does not try to locate them.

`inform=mlint('filename','-notok')` runs `mlint` for all lines in `filename`, even those lines that end with the `mlint` suppression directive, `%#ok`.

`mlint('filename','-cyc')` displays the McCabe complexity (also referred to as cyclomatic complexity) of each function in the file. Higher McCabe complexity values indicate higher complexity, and there is some evidence to suggest that programs with higher complexity values are more likely to contain errors. Frequently, you can lower the complexity of a function by dividing it into smaller, simpler functions. In general, smaller complexity values indicate programs that are easier to understand and modify. Some people advocate splitting up programs that have a complexity rating over 10.

`mlint('filename','-codegen')` enables code generation messages for display in the Command Window.

`mlint('filename','-eml')` `'-eml'` is not recommended. Use `'-codegen'` instead.

Examples

The following examples use `lengthofline.m`, which is a sample file with MATLAB code that can be improved. You can find it in *`matlabroot`*`/help/techdoc/matlab_env/examples`. If you want to run the examples, save a copy of `lengthofline.m` to a location on your MATLAB path.

Running mlint on a File with No Options

To run `mlint` on the example file, `lengthofline.m`, run

```
mlint('lengthofline')
```

MATLAB displays the M-Lint messages for `lengthofline.m` in the Command Window:

```
L 22 (C 1-9): The value assigned here to variable 'nothandle' might never be used.
L 23 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 24 (C 5-11): 'notline' might be growing inside a loop. Consider preallocating for speed.
L 24 (C 44-49): Use STRCMPI(str1,str2) instead of using LOWER in a call to STRCMP.
L 28 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 34 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.
L 34 (C 24-31): Use dynamic fieldnames with structures instead of GETFIELD.
                Type 'doc struct' for more information.
L 38 (C 29): Use || instead of | as the OR operator in (scalar) conditional statements.
L 39 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 40 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 42 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.
L 43 (C 13-15): 'dim' might be growing inside a loop. Consider preallocating for speed.
L 45 (C 13-15): 'dim' might be growing inside a loop.Consider preallocating for speed.
L 48 (C 52): There may be a parenthesis imbalance around here.
L 48 (C 53): There may be a parenthesis imbalance around here.
L 48 (C 54): There may be a parenthesis imbalance around here.
L 48 (C 55): There may be a parenthesis imbalance around here.
L 49 (C 17): Terminate statement with semicolon to suppress output (in functions).
L 49 (C 23): Use of brackets [] is unnecessary. Use parentheses to group, if needed.
```

For details about these messages and how to improve the code, see “Changing Code Based on Code Analyzer Messages” in the MATLAB Desktop Tools and Development Environment documentation.

Running mlint with Options to Show IDs and Return Results to a Structure

To store the results to a structure and include message IDs, run

```
inform=mlint('lengthofline', '-id')
```

MATLAB returns

```
inform =

19x1 struct array with fields:
    message
    line
    column
    id
```

To see values for the first message, run

```
inform(1)
```

MATLAB displays

```
ans =

    message: 'The value assigned here to variable 'nothandle' might never be used.'
       line: 22
     column: [1 9]
         id: 'NASGU'
```

Here, the message is for the value that appears on line 22 that extends from column 1–9 in the file.`NASGU` is the ID for the message `'The value assigned here to variable 'nothandle' might never be used.'`.

Displaying McCabe Complexity with mlint

To display the McCabe complexity of a MATLAB code file, run `mlint` with the `-cyc` option, as shown in the following example (assuming you have saved `lengthofline.m` to a local folder).

```
mlint lengthofline.m -cyc
```

Results displayed in the Command Window show the McCabe complexity of the file, followed by the M-Lint messages, as shown here:

```
L 1 (C 23-34): The McCabe complexity of 'lengthofline' is 12.
L 22 (C 1-9): The value assigned here to variable 'nothandle' might never be used.
L 23 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 24 (C 5-11): 'notline' might be growing inside a loop. Consider preallocating for speed.
L 24 (C 44-49): Use STRCMPI(str1,str2) instead of using UPPER/LOWER in a call to STRCMP.
L 28 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 34 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.
L 34 (C 24-31): Use dynamic fieldnames with structures instead of GETFIELD. Type 'doc struct'
 for more information.
L 38 (C 29): Use || instead of | as the OR operator in (scalar) conditional statements.
L 39 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 40 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 42 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.
L 43 (C 13-15): 'dim' might be growing inside a loop. Consider preallocating for speed.
L 45 (C 13-15): 'dim' might be growing inside a loop. Consider preallocating for speed.
L 48 (C 52): There may be a parenthesis imbalance around here.
L 48 (C 53): There may be a parenthesis imbalance around here.
L 48 (C 54): There may be a parenthesis imbalance around here.
L 48 (C 55): There may be a parenthesis imbalance around here.
L 49 (C 17): Terminate statement with semicolon to suppress output (in functions).
L 49 (C 23): Use of brackets [] is unnecessary.  Use parentheses to group, if needed.
```

See Also

`mlintrpt`, `profile`

How To

- For information on the suppression directive, `%#ok`, and suppressing messages from within your program, see “Adjust Code Analyzer Message Indicators and Messages”.

Introduced before R2006a

mlintrpt

Run `checkcode` for file or folder

Syntax

```
mlintrpt
mlintrpt(f)
mlintrpt(f, inputType)
mlintrpt( ___ ,settingsfile)
```

Description

`mlintrpt` scans all MATLAB code files in the current folder for Code Analyzer messages and reports the results in a MATLAB web browser.

`mlintrpt(f)` scans the specified file `f`.

`mlintrpt(f, inputType)` scans the specified file or folder `f`, as specified by `inputType`. Use `'file'` to specify a file and `'dir'` to specify a folder.

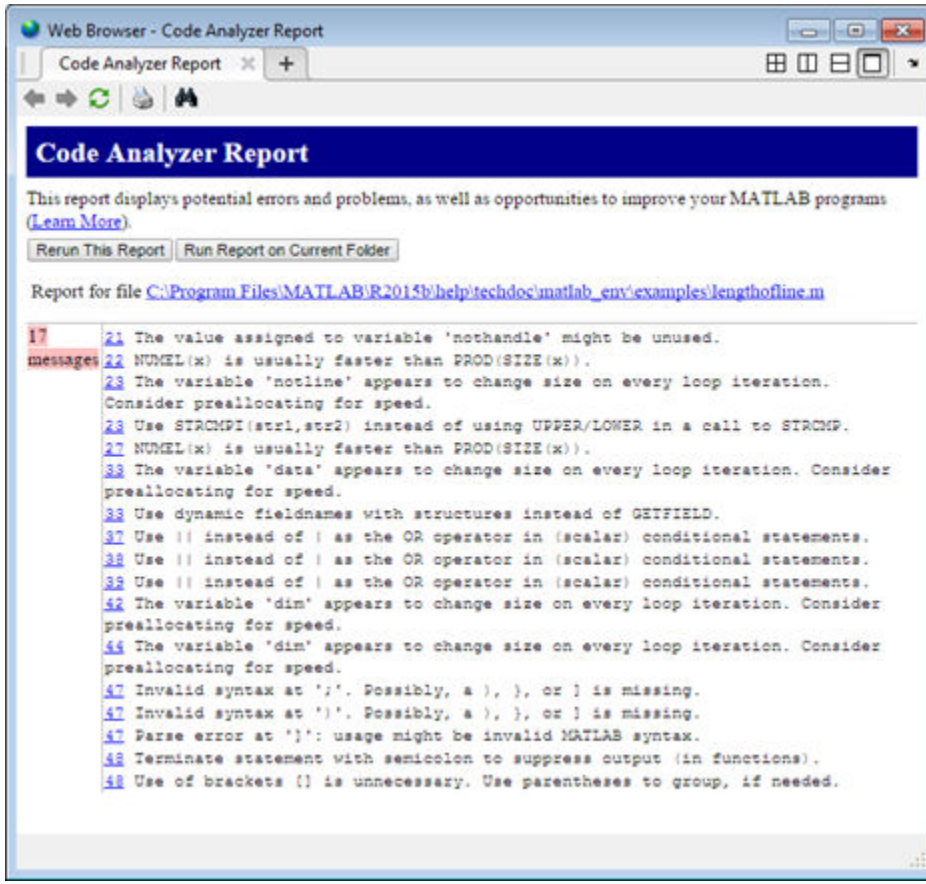
`mlintrpt( ___ ,settingsfile)` overrides the default active settings file with the specified settings file. You can specify `settingsfile` with any of the input arguments in the previous syntaxes.

Examples

Create Report for File

Run `mlintrpt` on the example file `lengthofline.m`. MATLAB® displays a report of potential problems and improvements for `lengthofline`.

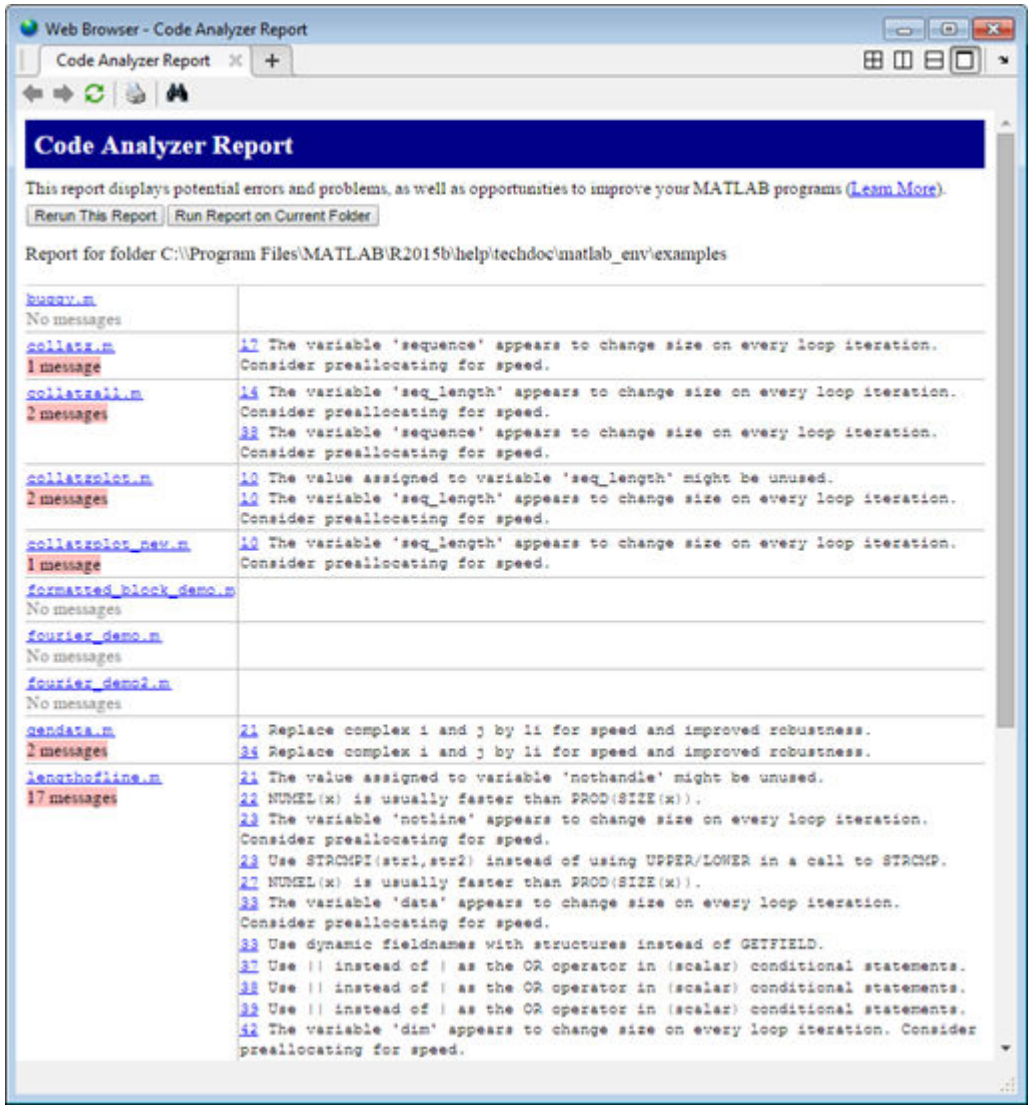
```
mlintrpt('lengthofline')
```



Create Report for All Files in Folder

Run `mlintrpt` on all examples in the MATLAB® example folder *`matlabroot`*`/matlab/help/techdoc/matlab_env/examples`. MATLAB displays a report of potential problems and improvements for all MATLAB code files in this folder.

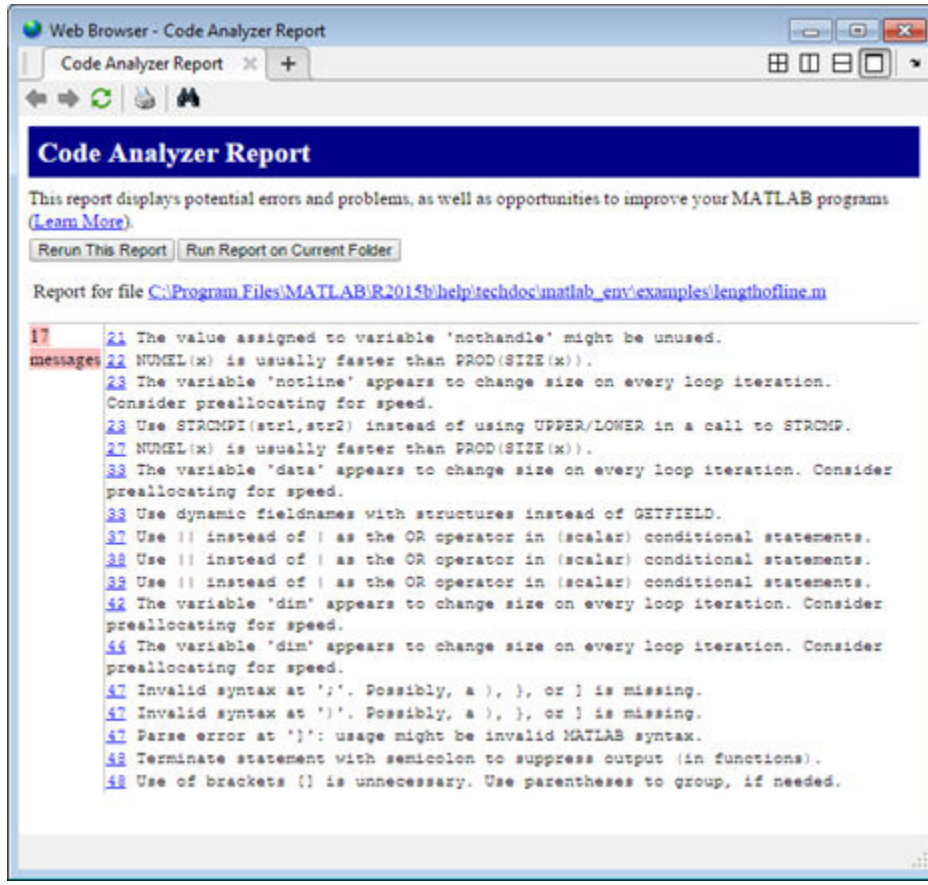
```
mlintrpt(fullfile(matlabroot,'help','techdoc','matlab_env','examples'),'dir')
```


Web Browser - Code Analyzer Report
Code Analyzer Report
Code Analyzer Report
This report displays potential errors and problems, as well as opportunities to improve your MATLAB programs (Learn More).
Rerun This Report
Run Report on Current Folder
Report for folder C:\\Program Files\MATLAB\R2015b\help\techdoc\matlab_env\examples
buggy.m
No messages
collatz.m
1 message
17 The variable 'sequence' appears to change size on every loop iteration. Consider preallocating for speed.
collatzall.m
2 messages
14 The variable 'seq_length' appears to change size on every loop iteration. Consider preallocating for speed.
38 The variable 'sequence' appears to change size on every loop iteration. Consider preallocating for speed.
collatzplot.m
2 messages
10 The value assigned to variable 'seq_length' might be unused.
10 The variable 'seq_length' appears to change size on every loop iteration. Consider preallocating for speed.
collatzplot_new.m
1 message
10 The variable 'seq_length' appears to change size on every loop iteration. Consider preallocating for speed.
formatted_block_demo.m
No messages
fourier_demo.m
No messages
fourier_demo2.m
No messages
gendata.m
2 messages
21 Replace complex i and j by 1i for speed and improved robustness.
34 Replace complex i and j by 1i for speed and improved robustness.
lengthofline.m
17 messages
21 The value assigned to variable 'nothandle' might be unused.
22 NUMEL(x) is usually faster than PROD(SIZE(x)).
23 The variable 'notline' appears to change size on every loop iteration. Consider preallocating for speed.
23 Use STRCMPI(str1,str2) instead of using UPPER/LOWER in a call to STRCMP.
27 NUMEL(x) is usually faster than PROD(SIZE(x)).
33 The variable 'data' appears to change size on every loop iteration. Consider preallocating for speed.
33 Use dynamic fieldnames with structures instead of GETFIELD.
37 Use || instead of | as the OR operator in (scalar) conditional statements.
38 Use || instead of | as the OR operator in (scalar) conditional statements.
39 Use || instead of | as the OR operator in (scalar) conditional statements.
42 The variable 'dim' appears to change size on every loop iteration. Consider preallocating for speed.

Suppress Code Analyzer Messages in Report

Suppress specific messages in a report by creating and specifying a settings file. For example, the file `lengthofline.m` includes several lines that use | instead of || as the `OR` operator. By default, `mlintrpt` flags these lines.

```
mlintrpt('lengthofline')
```



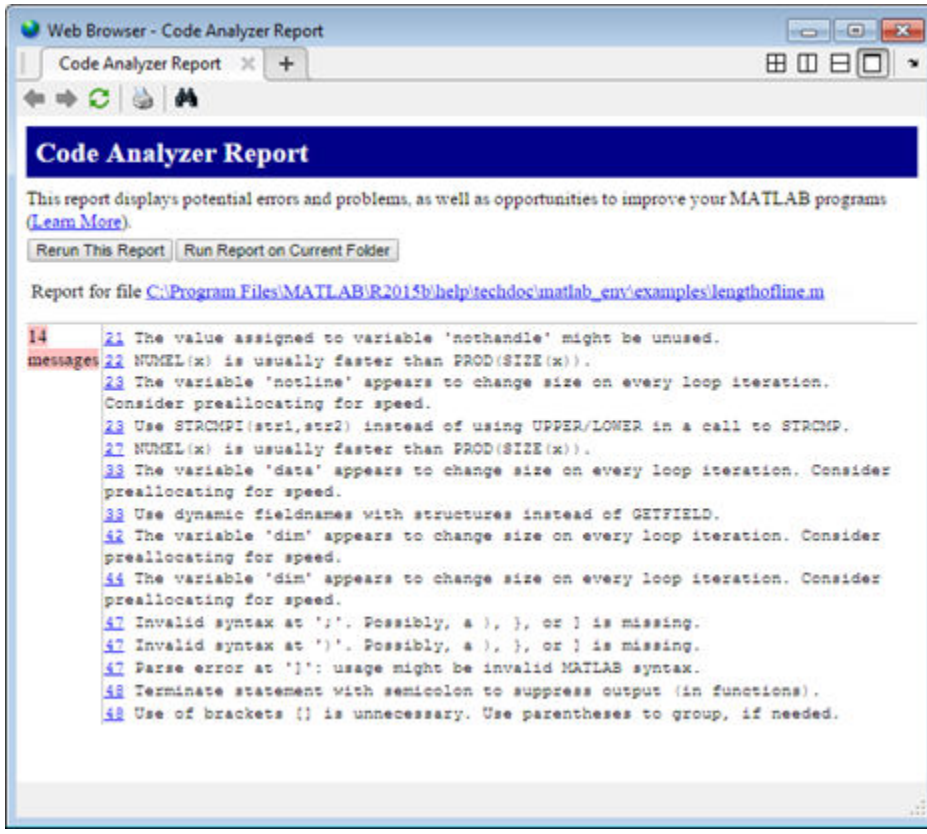
Create a settings file that suppresses the message flagging the use of | as the `OR` operator.

1. On the **Home** tab, in the **Environment** section, click the **Preferences** button.
2. Select **Code Analyzer** in the left pane.

3. Under **Default Settings**, in the **Aesthetics and Readability** section, clear the message **Use || instead of | as the OR operator in (scalar) conditional statements**.
4. Enter `mysettings.txt` as the file name and save it to your current folder.
5. Press the **Cancel** button to exit out of the preference panel without changing the active settings.

Run `mlintrpt` on the example file using the custom settings file `mysettings.txt`. The message **Use || instead of | as the OR operator in (scalar) conditional statements** is suppressed and is no longer visible in the report.

```
mlintrpt('lengthofline','mysettings.txt')
```



Input Arguments

`f` — File or folder name
character vector

File or folder name, specified as a character vector. The file name can include a partial path, but must be in a folder on the search path, or in the current folder.

`inputType` — Input type
`'file'` | `'dir'`

Input type, specified as `'file'` if the input is a file or `'dir'` if the input is a folder.

`settingsfile` — Settings file name
character vector

Settings file name, specified as a character vector. If the file is not in the current folder, provide the full path to the file.

For information about creating a settings file, see "Save and Reuse Code Analyzer Message Settings". If you specify an invalid file, the first message in the report is `0: Unable to open or read the configuration file 'mymlint.txt'--using default settings`.

See Also
`checkcode`

Topics
"Check Code for Errors and Warnings"

Introduced before R2006a

mlock

Prevent clearing function or script from memory

Syntax

```
mlock
```

Description

`mlock` locks the currently running function in memory. Locking a function prevents `clear` from removing it from memory, and prevents reinitialization of any `persistent` variables defined in the file.

Use `mlock` only within a MATLAB code file.

To remove a locked function or script from memory, first unlock it using the `munlock` command, and then use the `clear` command.

Examples

Lock Function with Persistent Variable

Create the function `myFun` in your current working folder.

```
function myFun()
    persistent n
    if isempty(n)
        n = 0;
    end
    n = n+1
end
```

At the command prompt, call `myFun` twice. Each time you call the function, the value of `n` increases because it is `persistent`.

```
myFun
myFun
myFun
```

```
n =

     1


n =

     2


n =

     3
```

Clear the function and call it another two times. Clearing the function also clears the persistent variable.

```
clear myFun
myFun
myFun
```

```
n =

     1


n =

     2
```

Edit the `myFun` function to include a call to `mlock`.

```
function myFun()
    mlock
    persistent n
    if isempty(n)
        n = 0;
    end
    n = n+1
end
```

At the command prompt, call `myFun` 3 times.

```
myFun
myFun
myFun

n =

     1


n =

     2


n =

     3
```

Try to clear the function and call it another two times. Since `myFun` is locked, clearing the function does not remove it from memory and does not clear the persistent variable.

```
clear myFun
myFun
myFun

n =

     4


n =

     5
```

Unlock `myFun` so it can be cleared from memory.

```
munlock('myFun')
```

Tips

- To lock a MEX file, use the `mexLock` function.

See Also

`inmem` | `mislocked` | `munlock` | `persistent`

Introduced before R2006a

mmfileinfo

Information about multimedia file

Syntax

info = mmfileinfo(*filename*)

Description

info = mmfileinfo(*filename*) returns a structure, *info*, with fields containing information about the contents of the multimedia file identified by *filename*. The *filename* is specified as a character vector or string scalar.

If *filename* is a URL, mmfileinfo might take a long time to return because it must first download the file. For large files, downloading can take several minutes. To avoid blocking the MATLAB command line while this processing takes place, download the file before calling mmfileinfo.

The *info* structure contains the following fields, listed in the order they appear in the structure.

Field	Description
Filename	Text indicating the name of the file.
Path	Text indicating the absolute path to the file.
Duration	Length of the file in seconds.
Audio	Structure containing information about the audio data in the file. See “Audio Data” on page 1-9891 for more information about this data structure.
Video	Structure containing information about the video data in the file. See “Video Data” on page 1-9891 for more information about this data structure.

Audio Data

The `Audio` structure contains the following fields, listed in the order they appear in the structure. If the file does not contain audio data, the fields in the structure are empty.

Field	Description
`Format`	Text indicating the audio format.
`NumChannels`	Number of audio channels.

Video Data

The `Video` structure contains the following fields, listed in the order they appear in the structure. If the file does not contain video data, the fields in the structure are empty.

Field	Description
`Format`	Text indicating the video format.
`Height`	Height of the video frame.
`Width`	Width of the video frame.

Examples

Display information about the example file `xylophone.mpg`:

```
info = mmfileinfo('xylophone.mpg')
audio = info.Audio
video = info.Video
```

MATLAB returns:

```
info = 
    Filename: 'xylophone.mpg'
        Path: 'matlabroot\toolbox\matlab\audiovideo'
    Duration: 4.7020
       Audio: [1x1 struct]
       Video: [1x1 struct]

audio = 
              Format: 'MPEG'
```

```
         NumChannels: 2

video =
    Format: 'MPEG1'
    Height: 240
     Width: 320
```

where `Path` is system-dependent.

See Also

`VideoReader`

Introduced before R2006a

mod

Remainder after division (modulo operation)

Syntax

```
b = mod(a,m)
```

Description

`b = mod(a,m)` returns the remainder after division of `a` by `m`, where `a` is the dividend and `m` is the divisor. This function is often called the modulo operation, which can be expressed as `b = a - m.*floor(a./m)`. The `mod` function follows the convention that `mod(a,0)` returns `a`.

Examples

Remainder After Division of Scalar

Compute 23 modulo 5.

```
b = mod(23,5)
```

```
b = 3
```

Remainder After Division of Vector

Find the remainder after division for a vector of integers and the divisor 3.

```
a = 1:5;
m = 3;
b = mod(a,m)
```

```
b = 1×5

     1     2     0     1     2
```

Remainder After Division for Positive and Negative Values

Find the remainder after division for a set of integers including both positive and negative values. Note that nonzero results are always positive if the divisor is positive.

```
a = [-4 -1 7 9];
m = 3;
b = mod(a,m)
```

```
b = 1×4

     2     2     1     0
```

Remainder After Division for Negative Divisor

Find the remainder after division by a negative divisor for a set of integers including both positive and negative values. Note that nonzero results are always negative if the divisor is negative.

```
a = [-4 -1 7 9];
m = -3;
b = mod(a,m)
```

```
b = 1×4

    -1    -1    -2     0
```

Remainder After Division for Floating-Point Values

Find the remainder after division for several angles using a modulus of `2*pi`. Note that `mod` attempts to compensate for floating-point round-off effects to produce exact integer results when possible.

```
theta = [0.0 3.5 5.9 6.2 9.0 4*pi];
m = 2*pi;
b = mod(theta,m)
```

```
b = 1×6

         0    3.5000    5.9000    6.2000    2.7168         0
```

Input Arguments

a — Dividend
scalar | vector | matrix | multidimensional array

Dividend, specified as a scalar, vector, matrix, or multidimensional array. `a` must be a real-valued array of any numerical type. Numeric inputs `a` and `m` must either be the same size or have sizes that are compatible (for example, `a` is an `M`-by-`N` matrix and `m` is a scalar or `1`-by-`N` row vector). For more information, see “Compatible Array Sizes for Basic Operations”.

If `a` and `m` are duration arrays, then they must be the same size unless one is a scalar. If one input is a duration array, the other input can be a duration array or a numeric array. In this context, `mod` treats numeric values as a number of standard 24-hour days.

If one input has an integer data type, then the other input must be of the same integer data type or be a scalar `double`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `duration` | `char`

m — Divisor
scalar | vector | matrix | multidimensional array

Divisor, specified as a scalar, vector, matrix, or multidimensional array. `m` must be a real-valued array of any numerical type. Numeric inputs `a` and `m` must either be the same size

or have sizes that are compatible (for example, a is an M-by-N matrix and m is a scalar or 1-by-N row vector). For more information, see “Compatible Array Sizes for Basic Operations”.

If a and m are duration arrays, then they must be the same size unless one is a scalar. If one input is a duration array, the other input can be a duration array or a numeric array. In this context, mod treats numeric values as a number of standard 24-hour days.

If one input has an integer data type, then the other input must be of the same integer data type or be a scalar double.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | duration | char

More About

Differences Between mod and rem

The concept of remainder after division is not uniquely defined, and the two functions mod and rem each compute a different variation. The mod function produces a result that is either zero or has the same sign as the divisor. The rem function produces a result that is either zero or has the same sign as the dividend.

Another difference is the convention when the divisor is zero. The mod function follows the convention that mod(a,0) returns a, whereas the rem function follows the convention that rem(a,0) returns NaN.

Both variants have their uses. For example, in signal processing, the mod function is useful in the context of periodic signals because its output is periodic (with period equal to the divisor).

Congruence Relationships

The mod function is useful for congruence relationships: a and b are congruent (mod m) if and only if mod(a,m) == mod(b,m). For example, 23 and 13 are congruent (mod 5).

References

[1] Knuth, Donald E. *The Art of Computer Programming*. Vol. 1. Addison Wesley, 1997 pp.39–40.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Arithmetic is performed using the output class. Results might not match MATLAB due to differences in rounding errors.
- If one of the inputs has type `int64` or `uint64`, both inputs must have the same type.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- 64-bit integers are not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`rem`

Introduced before R2006a

mode

Most frequent values in array

Syntax

```
M = mode(A)
M = mode(A,'all')
M = mode(A,dim)
M = mode(A,vecdim)
```

```
[M,F] = mode( ___ )
[M,F,C] = mode( ___ )
```

Description

`M = mode(A)` returns the sample mode of `A`, which is the most frequently occurring value in `A`. When there are multiple values occurring equally frequently, `mode` returns the smallest of those values. For complex inputs, the smallest value is the first value in a sorted list.

- If `A` is a vector, then `mode(A)` returns the most frequent value of `A`.
- If `A` is a nonempty matrix, then `mode(A)` returns a row vector containing the mode of each column of `A`.
- If `A` is an empty 0-by-0 matrix, `mode(A)` returns `NaN`.
- If `A` is a multidimensional array, then `mode(A)` treats the values along the first array dimension whose size does not equal `1` as vectors and returns an array of most frequent values. The size of this dimension becomes `1` while the sizes of all other dimensions remain the same.

`M = mode(A,'all')` computes the mode over all elements of `A`. This syntax is valid for MATLAB versions R2018b and later.

`M = mode(A,dim)` returns the mode of elements along dimension `dim`. For example, if `A` is a matrix, then `mode(A,2)` is a column vector containing the most frequent value of each row

`M = mode(A,vecdim)` computes the mode based on the dimensions specified in the vector `vecdim`. For example, if `A` is a matrix, then `mode(A,[1 2])` is the mode over all elements in `A`, since every element of a matrix is contained in the array slice defined by dimensions 1 and 2.

`[M,F] = mode( ___ )` also returns a frequency array `F`, using any of the input arguments in the previous syntaxes. `F` is the same size as `M`, and each element of `F` represents the number of occurrences of the corresponding element of `M`.

`[M,F,C] = mode( ___ )` also returns a cell array `C` of the same size as `M` and `F`. Each element of `C` is a sorted vector of all values that have the same frequency as the corresponding element of `M`.

Examples

Mode of Matrix Columns

Define a 3-by-4 matrix.

```
A = [3 3 1 4; 0 0 1 1; 0 1 2 4]
```

```
A = 3×4

     3     3     1     4
     0     0     1     1
     0     1     2     4
```

Find the most frequent value of each column.

```
M = mode(A)
```

```
M = 1×4

     0     0     1     4
```

Mode of Matrix Rows

Define a 3-by-4 matrix.

```
A = [3 3 1 4; 0 0 1 1; 0 1 2 4]
```

```
A = 3×4

     3     3     1     4
     0     0     1     1
     0     1     2     4
```

Find the most frequent value of each row.

```
M = mode(A,2)
```

```
M = 3×1

     3
     0
     0
```

Mode of 3-D Array

Create a 1-by-3-by-4 array of integers between 1 and 10.

```
A = gallery('integerdata',10,[1,3,4],1)
```

```
A =
A(:,:,1) =

    10     8    10


A(:,:,2) =

     6     9     5


A(:,:,3) =
```

```
     9     6     1


A(:,:,4) =

     4     9     5
```

Find the most frequent values of this 3-D array along the second dimension.

```
M = mode(A)
```

```
M =
M(:,:,1) =

    10


M(:,:,2) =

     5


M(:,:,3) =

     1


M(:,:,4) =

     4
```

This operation produces a 1-by-1-by-4 array by finding the most frequent value along the second dimension. The size of the second dimension reduces to 1.

Compute the mode along the first dimension of A.

```
M = mode(A,1);
isequal(A,M)
```

```
ans = logical
   1
```

This returns the same array as A because the size of the first dimension is 1.

Mode of Array Page

Create a 3-D array and compute the mode over each page of data (rows and columns).

```
A(:,:,1) = [2 4; 2 1];
A(:,:,2) = [6 2; 3 3];
A(:,:,3) = [4 4; 7 4];
M1 = mode(A,[1 2])
```

```
M1 =
M1(:,:,1) =

     2


M1(:,:,2) =

     3


M1(:,:,3) =

     4
```

Starting in R2018b, to compute the mode over all dimensions of an array, you can either specify each dimension in the vector dimension argument, or use the `'all'` option.

```
M2 = mode(A,[1 2 3])
```

```
M2 = 4
```

```
Mall = mode(A,'all')
```

```
Mall = 4
```

Mode of Matrix Columns with Frequency Information

Define a 3-by-4 matrix.

```
A = [3 3 1 4; 0 0 1 1; 0 1 2 4]
```

```
A = 3×4

     3     3     1     4
     0     0     1     1
     0     1     2     4
```

Find the most frequent value of each column, as well as how often it occurs.

```
[M,F] = mode(A)
```

```
M = 1×4

     0     0     1     4


F = 1×4

     2     1     2     2
```

`F(1)` is `2` since `M(1)` occurs twice in the first column.

Mode of Matrix Rows with Frequency and Multiplicity Information

Define a 3-by-4 matrix.

```
A = [3 3 1 4; 0 0 1 1; 0 1 2 4]
```

```
A = 3×4

     3     3     1     4
     0     0     1     1
     0     1     2     4
```

Find the most frequent value of each row, how often it occurs, and which values in that row occur with the same frequency.

```
[M,F,C] = mode(A,2)
```

```
M = 3×1

     3
     0
     0


F = 3×1

     2
     2
     1


C=3×1 cell
    {[       3]}
    {2x1 double}
    {4x1 double}
```

`C{2}` is the 2-by-1 vector `[0;1]` since values `0` and `1` in the second row occur with frequency `F(2)`.

`C{3}` is the 4-by-1 vector `[0;1;2;4]` since all values in the third row occur with frequency `F(3)`.

Mode of 16-bit Unsigned Integer Array

Define a 1-by-4 vector of 16-bit unsigned integers.

```
A = gallery('integerdata',10,[1,4],3,'uint16')
```

```
A = 1x4 uint16 row vector

   6   3   2   3
```

Find the most frequent value, as well as the number of times it occurs.

```
[M,F] = mode(A),
```

```
M = uint16
    3
```

```
F = 2
```

```
class(M)
```

```
ans =
'uint16'
```

`M` is the same class as the input, `A`.

Input Arguments

`A` — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array. `A` can be a numeric array, categorical array, datetime array, or duration array.

`NaN` or `NaT` (Not a Time) values in the input array, `A`, are ignored. Undefined values in categorical arrays are similar to `NaN`s in numeric arrays.

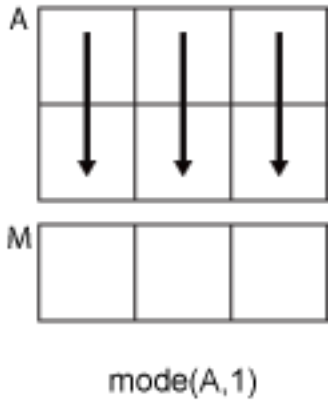
`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal `1`.

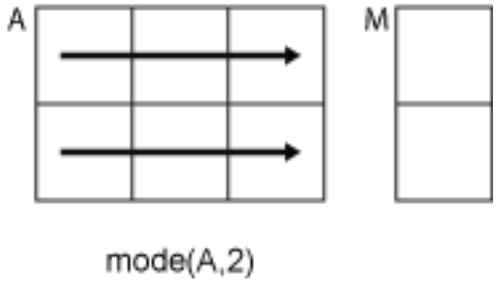
Dimension `dim` indicates the dimension whose length reduces to `1`. The `size(M,dim)` is `1`, while the sizes of all other dimensions remain the same.

Consider a two-dimensional input array, `A`.

- If `dim = 1`, then `mode(A,1)` returns a row vector containing the most frequent value in each column.



- If `dim = 2`, then `mode(A,2)` returns a column vector containing the most frequent value in each row.



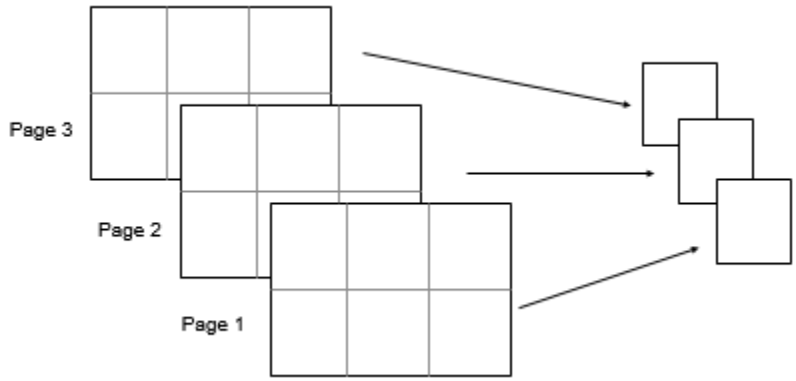
`mode` returns `A` if `dim` is greater than `ndims(A)`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`vecdim` — Vector of dimensions
vector of positive integers

Vector of dimensions, specified as a vector of positive integers. Each element represents a dimension of the input array. The lengths of the output in the specified operating dimensions are 1, while the others remain the same.

Consider a 2-by-3-by-3 input array, `A`. Then `mode(A,[1 2])` returns a 1-by-1-by-3 array whose elements are the modes of each page of `A`.



Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Output Arguments

M — Most frequent values
scalar | vector | matrix | multidimensional array

Most frequent values returned as a scalar, vector, matrix, or multidimensional array. When there are multiple values occurring equally frequently, mode returns the smallest of those values. For complex inputs, this is taken to be the first value in a sorted list of values.

The class of M is the same as the class of the input array, A.

F — Frequency array
scalar | vector | matrix | multidimensional array

Frequency array returned as a scalar, vector, matrix, or multidimensional array. The size of F is the same as the size of M, and each element of F represents the number of occurrences of the corresponding element of M.

The class of F is always double.

C — Most frequent values with multiplicity
cell array

Most frequent values with multiplicity returned as a cell array. The size of `C` is the same as the size of `M` and `F`, and each element of `C` is a sorted column vector of all values that have the same frequency as the corresponding element of `M`.

Tips

- The `mode` function is most useful with discrete or coarsely rounded data. The mode for a continuous probability distribution is defined as the peak of its density function. Applying the `mode` function to a sample from that distribution is unlikely to provide a good estimate of the peak; it would be better to compute a histogram or density estimate and calculate the peak of that estimate. Also, the `mode` function is not suitable for finding peaks in distributions having multiple modes.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Does not support the third output argument `C`.
- If supplied, `dim` must be a constant.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`histcounts` | `histogram` | `mean` | `median` | `sort`

Introduced before R2006a

month

Month number and name

Syntax

```
m = month(t)
m = month(t,monthType)
```

Description

`m = month(t)` returns the month numbers of the datetime values in `t`. The `m` output contains integer values from 1 to 12.

`m = month(t,monthType)` returns the type of month number or name specified by `monthType`.

The `month` function returns the month numbers or names of datetime values. To assign month numbers to datetime array `t`, use `t.Month` and modify the `Month` property.

Examples

Extract Month Number from Dates

Extract the month numbers from an array of dates.

```
t = datetime(2014,05,31):caldays(35):datetime(2014,10,15)
```

```
t = 1x4 datetime array
   31-May-2014   05-Jul-2014   09-Aug-2014   13-Sep-2014
```

```
m = month(t)
```

```
m = 1×4
```

```
     5     7     8     9
```

Find Month Names of Dates

Get the month names from an array of dates.

```
t = datetime(2013,01,01):calweeks(12):datetime(2013,12,31)
```

```
t = 1x5 datetime array
   01-Jan-2013   26-Mar-2013   18-Jun-2013   10-Sep-2013   03-Dec-2013
```

```
m = month(t,'name')
```

```
m = 1x5 cell array
    {'January'}    {'March'}    {'June'}    {'September'}    {'December'}
```

Input Arguments

t — Input date and time
`datetime` array

Input date and time, specified as a `datetime` array.

monthType — Type of month values
`'monthofyear'` (default) | `'name'` | `'shortname'`

Type of month values, specified as a value in the table.

Value of monthType	Description
`'monthofyear'`	Month-of-year number
`'name'`	Full month names, for example, `August` or `September`. For `NaT` datetime values, the month name is the empty character vector, `''`.

Value of monthType	Description
'shortname'	Abbreviated month names, for example, Aug or Sep. For NaT datetime values, the month name is the empty character vector, ''.

Note month returns month names in the language specified by the **Locale** option in the **Datetime format** section of the Preferences panel. To change the default datetime locale, see "Set Command Window Preferences".

Output Arguments

m — Month number or name
double array | cell array of character vectors

Month number or name, returned as a numeric array of type double, or a cell array of character vectors. m is the same size as t.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see "Tall Arrays".

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see "Run MATLAB Functions with Distributed Arrays" (Parallel Computing Toolbox).

See Also

`datetime` | `day` | `quarter` | `week` | `year` | `ymd`

Introduced in R2014b

more

Control paged output in Command Window

Syntax

```
more on
more off
more(n)
A = more( ___ )
```

Description

`more on` enables paging of the output in the MATLAB Command Window. When paging is enabled, MATLAB displays output one page at a time.

- To advance to the next page of output, press the **Space** key.
- To advance to the next line of output, press the **Return** key.
- To stop displaying the current output, press the **Q** key. Do not use **Ctrl+C** to exit `more`, otherwise MATLAB can return an error.

`more off` disables paging of the output in the Command Window. By default, paging is disabled.

`more(n)` enables paging and sets the page length to `n` lines.

By default, the length of a page is equal to the number of lines available for display in the MATLAB Command Window. If you set the page length to a specific value, MATLAB uses that value for the page size, regardless of the size of the command window.

To restore the default page length, run the command `more off` followed by the command `more on`.

`A = more( ___ )` returns the current page size in number of lines. You can use this syntax with any of the input argument combinations in the previous syntaxes.

Examples

View Help One Page at a Time

Use the `more` function to control the number of help lines displayed in the command window.

Enable paging in the Command Window and then get help on the `plot` function.

```
more on
help plot
```

```
 plot   Linear plot.
    plot(X,Y) plots vector Y versus vector X. If X or Y is a matrix,
    then the vector is plotted versus the rows or columns of the matrix,
    whichever line up.  If X is a scalar and Y is a vector, disconnected
    line objects are created and plotted as discrete points vertically at
    X.

    plot(Y) plots the columns of Y versus their index.
    If Y is complex, plot(Y) is equivalent to plot(real(Y),imag(Y)).
--more--
```

Press the **Space** key to view the help page by page until the last page is displayed. Call `more off` to disable paging.

Change Command Window Page Size

Set the page size in the Command Window and then restore the original page size.

Enable paging and set the page size to 5, storing the previous size in the variable `previousSize`.

```
previousSize = more(5)
```

```
previousSize =

     9
```

Display the help text for the `plot` function.

```
more on
help plot
```

```
plot   Linear plot.
    plot(X,Y) plots vector Y versus vector X. If X or Y is a matrix,
    then the vector is plotted versus the rows or columns of the matrix,
    whichever line up.  If X is a scalar and Y is a vector, disconnected
    line objects are created and plotted as discrete points vertically at
--more--
```

Use the **Space** key to view the help page by page until the last page is displayed.

Restore the page size and display the help for the `plot` function again.

```
more(previousSize)
help plot
```

```
 plot   Linear plot.
    plot(X,Y) plots vector Y versus vector X. If X or Y is a matrix,
    then the vector is plotted versus the rows or columns of the matrix,
    whichever line up.  If X is a scalar and Y is a vector, disconnected
    line objects are created and plotted as discrete points vertically at
    X.

    plot(Y) plots the columns of Y versus their index.
    If Y is complex, plot(Y) is equivalent to plot(real(Y),imag(Y)).
--more--
```

Use the **Space** key to view the help page by page until the last page is displayed. Call `more off` to disable paging.

Input Arguments

n — Page size
positive integer

Page size, specified as a positive integer.

Tips

- To see the status of `more`, type `get(0,'More')`. MATLAB returns either `on` or `off`, indicating whether paging of the output is enabled or disabled.

See Also

`diary`

Topics

"Format Output"

Introduced before R2006a

morebins

Package: `matlab.graphics.chart.primitive`

Increase number of histogram bins

Syntax

```
N = morebins(h)
N = morebins(h,direction)
```

Description

`N = morebins(h)` increases the number of bins in histogram `h` by 10% (rounded up to the nearest integer) and returns the new number of bins.

For bivariate histograms, this increases the bin count in both the *x* and *y* directions.

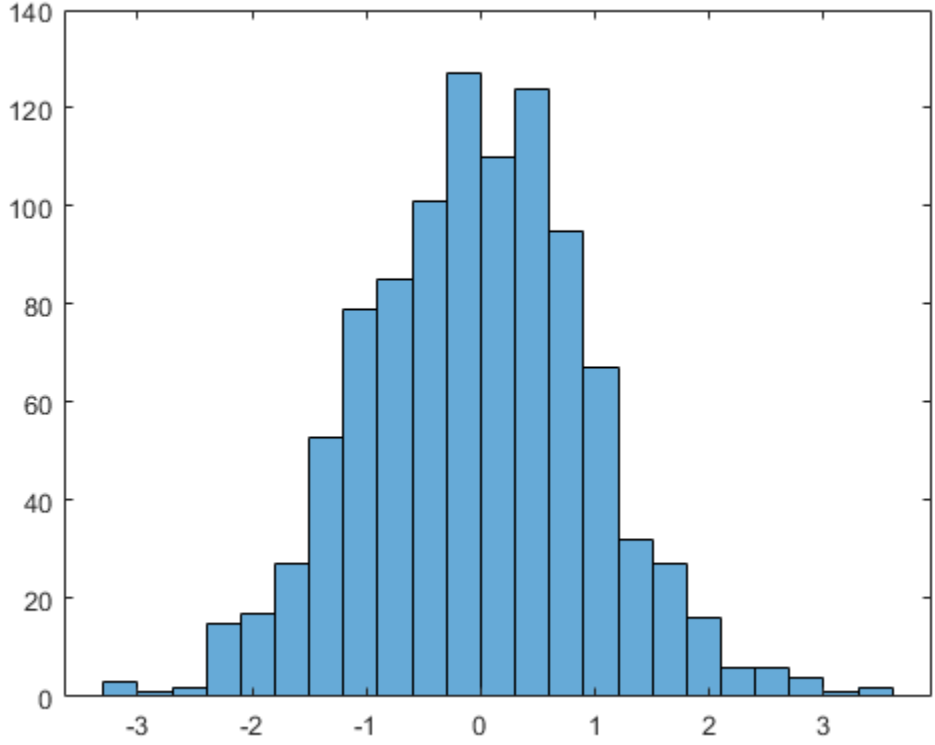
`N = morebins(h,direction)`, where `h` must be a `histogram2` object, only increases the number of bins in the dimension specified by `direction`. The `direction` option can be `'x'`, `'y'`, or `'both'`. The default value is `'both'`.

Examples

Increase Number of Histogram Bins

Plot a histogram of 1,000 random numbers and return a handle to the histogram object.

```
x = randn(1000,1);
h = histogram(x)
```

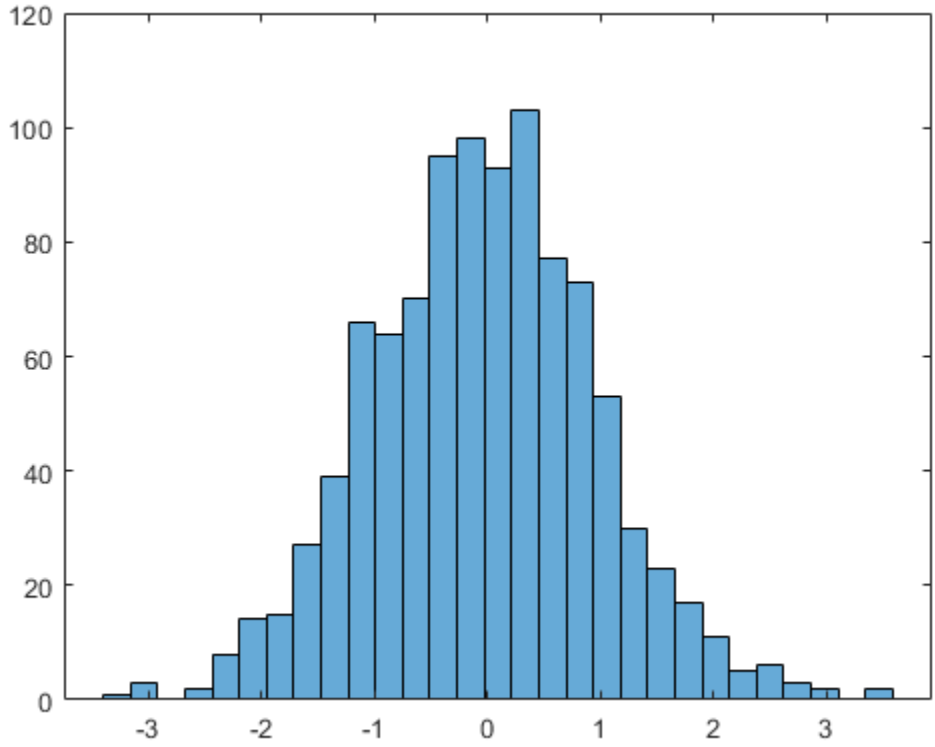



```
h =
  Histogram with properties:

            Data: [1000x1 double]
          Values: [1x23 double]
         NumBins: 23
        BinEdges: [1x24 double]
        BinWidth: 0.3000
       BinLimits: [-3.3000 3.6000]
   Normalization: 'count'
       FaceColor: 'auto'
       EdgeColor: [0 0 0]
```

```
Show all properties
```

Use `morebins` to increase the number of bins in the histogram.

```
morebins(h);
morebins(h)
```




```
ans = 29
```

Input Arguments

h — Input histogram
`histogram` object | `histogram2` object

Input histogram, specified as a `histogram` or `histogram2` object.

`h` cannot be a categorical histogram.

`direction` — Direction to increase number of bins
`'both'` (default) | `'x'` | `'y'`

Direction to increase the number of bins, specified as `'x'`, `'y'`, or `'both'`. Specify `'x'` or `'y'` to only increase the number of bins in that direction while leaving the number of bins in the other direction constant.

Output Arguments

`N` — Number of bins
scalar | vector

Number of bins, returned as a scalar or vector. `N` is the new number of bins for the histogram after increase. For bivariate histogram plots, `N` is a two-element vector, `[nx ny]`.

See Also

`fewerbins` | `histcounts` | `histcounts2` | `histogram` | `histogram2`

Introduced in R2014b

move

Move or resize control in parent window

Syntax

```
V = move(c)
V = move(c,position)
```

Description

`V = move(c)` returns a four-element vector indicating the current position of the control.

`V = move(c,position)` moves the control to the position specified by `position`.

Input Arguments

`c` — COM object
function handle

COM object, specified as a function handle.

`position` — Position
integer

New position in x,y pixel coordinates, specified as integers.

Example: `[0 0 200 200]`

Output Arguments

`V` — Position vector
integer

Position vector, specified as four-element integer vector containing the position and size of the control in the parent figure window. The elements of the vector are `[x, y,`

`width, height]`. The values `x` and `y` are offsets in pixels from the bottom left corner of the figure window to the same corner of the control. The values `width` and `height` are the size of the control itself.

See Also

`get` | `set`

Introduced before R2006a

move

Package: `matlab.settings`

Record move or rename of factory setting or group

Syntax

```
move(source,destination)
```

Description

`move(source,destination)` records the move of a factory setting or settings group with a factory settings tree. You also can use the `move` function to record the rename of a factory setting or group. Recording changes to the factory settings tree for each toolbox version ensures that users upgrading to a new version of a toolbox do not have backward incompatibility issues with their settings.

Examples

Record Move of Settings Group

Create a settings file upgrader object and record the move of a settings group in the second version of a toolbox.

Create a settings file upgrader object for version 2 of `mytoolbox`.

```
upgrader = SettingsFileUpgrader('Version2');
```

Record the move of the settings group `fonts` from the `mytoolbox` factory settings group to the `mytoolbox.mysettings` group.

```
upgrader.move('mytoolbox.fonts','mytoolbox.mysettings.fonts');
```

Record Rename of Setting

Create a settings file upgrader object and record the rename of a setting in the second version of a toolbox.

Create a settings file upgrader object for version 2 of `mytoolbox`.

```
upgrader = SettingsFileUpgrader('Version2');
```

Record the rename of the setting `MySetting` to `MyUpdatedSetting`, and the move of the setting from the `mytoolbox` group to the `mytoolbox.mysettings` group.

```
upgrader.move('mytoolbox.MySetting','mytoolbox.mysettings.MyUpdatedSetting');
```

Input Arguments

`source` — Source path
character vector | string

Source path of setting or settings group, specified as a character vector or string. Specify the full source path of the setting or group, starting from the toolbox factory settings group and including the setting or group name.

Example:
`upgrader.move('mytoolbox.MySetting','mytoolbox.MyUpdatedSetting');`

`destination` — Destination path
character vector | string scalar

Destination path of setting or settings group, specified as a character vector or string scalar. Specify the full destination path of the setting or group, starting from the toolbox factory settings group and including the setting or group name.

Example:
`upgrader.move('mytoolbox.MySetting','mytoolbox.MyUpdatedSetting');`

See Also

`SettingsFileUpgrader` | `remove`

Topics

"Create Factory Settings for Toolboxes"

"Access and Modify Settings"

Introduced in R2019b

movefile

Move or rename file or folder

Syntax

```
movefile source
movefile source destination
movefile source destination f

status = movefile( ___ )
[status,msg] = movefile( ___ )
[status,msg,msgID] = movefile( ___ )
```

Description

`movefile source` moves the file or folder `source` to the current folder. `movefile` does not preserve the archive attribute of `source`.

`movefile source destination` moves `source` to the file or folder `destination`. If `source` and `destination` are in the same location, then `movefile` renames `source` to `destination`. To rename a file or folder when moving it, make `destination` a different name from `source` and specify only one file or folder for `source`.

If `source` is a folder, then `destination` must be a folder. If `source` is a folder or specifies multiple files and `destination` does not exist, then `movefile` creates `destination`.

`movefile source destination f` performs the move, even when `destination` is not writable. The state of the read/write attribute for `destination` does not change.

`status = movefile( ___ )` moves the specified file or folder and returns a status of `1` if the operation is successful. Otherwise, `movefile` returns `0`. You can use this syntax with any of the input argument combinations in the previous syntaxes.

`[status,msg] = movefile( ___ )` also returns the message text for any warning or error that occurs.

`[status,msg,msgID] = movefile( ___ )` additionally returns the message ID for any warning or error that occurs.

Examples

Move Files to Current Folder

Move files and folders to the current folder by omitting the destination input.

Create two folders: the first, `myfiles`, containing the file `myfile1.m`, and the second, `myotherfiles`, containing the file `myfile2.m`.

```
mkdir myfiles
movefile myfile1.m myfiles
mkdir myotherfiles
movefile myfile2.m myotherfiles
```

Move `myfile1.m` to the current folder. Since a destination is not specified, MATLAB® assumes the destination is the current folder.

```
movefile myfiles/myfile1.m
```

Set the current folder to `myfiles`. Move `myotherfiles` and its contents to the current folder.

```
cd myfiles
movefile ../myotherfiles
```

Move Files and Folders to New Folder

Move files and subfolders whose names begin with `my` from the current folder to the folder `newFolder`, where `newFolder` previously does not exist.

```
movefile my* newFolder
```

Rename Folder

Create the folder `myoldfolder`, and then rename it to `mynewfolder`.

```
mkdir myoldfolder
movefile myoldfolder mynewfolder
```

Move File to Read-Only Folder

Move the file `myfile1.m` from the current folder to the read-only folder `restricted`.

Create the read-only folder `restricted`.

```
mkdir restricted
fileattrib restricted -w
```

Move the file `myfile1.m`. A status of 0 shows the copy was unsuccessful.

```
status = movefile('myfile1.m','restricted');
status
```

```
status = logical
   0
```

Move the file `myfile1.m` using the 'f' option to override the read-only status of the destination folder. A status of 1 and an empty `message` and `messageID` confirm the copy was successful.

```
[status,message,messageId] = movefile('myfile1.m','restricted','f');
status
```

```
status = logical
   1
```

```
message
```

```
message =

  0x0 empty char array
```

```
messageId
```

```
messageId =

  0x0 empty char array
```

Input Arguments

`source` — File or folder to copy

character vector | string scalar

File or folder to copy, specified as a character vector or string scalar. `source` can be an absolute or relative path. To copy multiple files or folders, use wildcards (*).

Note If `source` is a string, enclose all the inputs in parentheses. For example, `movefile("myfile.m","newfolder")`.

`destination` — File or folder destination

character vector | string scalar

File or folder destination for the copy made, specified as a character vector or string scalar. `destination` can be an absolute or relative path, but cannot include wildcards (*).

Note If `destination` is a string, enclose all the inputs in parentheses. For example, `movefile("myfile.m","newfolder")`.

Output Arguments

`status` — Move status

0 | 1

Move status, indicating if the attempt to move the file or folder is successful, returned as `0` or `1`. If the attempt is successful, the value of `status` is 1. Otherwise, the value is 0.

Data Types: `logical`

`msg` — Error message

character vector

Error message, returned as a character vector. If an error or warning occurs, `msg` contains the message text of the error or warning. Otherwise, `msg` is empty, `''`.

`msgID` — Error message identifier
character vector

Error message identifier, returned as a character vector. If an error or warning occurs, `msgID` contains the message identifier of the error or warning. Otherwise, `msgID` is empty, `''`.

See Also

`cd` | `copyfile` | `delete` | `dir` | `mkdir` | `rmdir`

Topics

"Manage Files and Folders"
"Specify File Names"

Introduced before R2006a

movegui

Move figure to specified location on screen

Syntax

```
movegui(f,position)
movegui(position)
movegui(f)
movegui
```

Description

`movegui(f,position)` moves the figure `f` to the specified screen location. The figure can be one created with either the `figure` or `uifigure` function. The position can be specified as a two-element vector or as a predefined position name.

`movegui(position)` moves the current figure or the callback figure to the specified position. To determine the current figure or the callback figure use `gcf` or `gcbf`, respectively. Note that `gcf` and `gcbf` return figures created with the `figure` function only. If one does not exist, MATLAB creates one using `figure` and moves it to the specified position.

`movegui(f)` moves the figure to the closest position that puts it entirely on screen.

`movegui` moves the current figure (`gcf`) or the callback figure (`gcbf`) to the closest position that puts it entirely on screen.

Examples

Move Figure with Position Vector

Create a figure and move it so that the bottom left corner is 300 pixels from the left side of the screen and 600 pixels from the bottom.

```
fig = uifigure;
movegui(fig,[300 600]);
```

Move Figure with Position Name

Create a figure and move it to the bottom center of the screen.

```
f = figure;
movegui(f,'south');
```

Move Current Figure

```
f1 = figure;
f2 = figure;
movegui('east');
```

`f2` moves because it is the current figure.

Move Off-Screen Figure on Screen

Create a figure that is positioned far outside the upper rightmost corner of the screen. Then, move it onscreen.

```
f = figure('Position',[10000 10000 400 300]);
movegui(f);
```

The figure moves to the top right corner of the screen because that is the position that is closest to its last location.

Input Arguments

`f` — Figure
`Figure` object

Figure, specified as a `Figure` object created with either the `figure` or `uifigure` function. Use this argument to specify a figure you want to move.

`position` — Position

two-element numeric vector | character vector | string scalar

Position of the figure on the screen, specified as a two-element numeric vector, a character vector, or a string scalar. A numeric vector specifies *x*- and *y*-values in pixels on page 1-9936, and a text argument specifies one of the predefined position names.

Example: `movegui(f,[150 -50])`

Example: `movegui('west')`

To indicate the offset of the figure from the edges of the screen in units of pixels, specify a two-element numeric vector, `[x y]`. The edge of the screen that the offset is measured from depends on the sign of the vector element.

Position Vector Element	Value Range	Description
`x`	`x >= 0`	Offset of left side from left edge of screen
	`x < 0`	Offset of right side from right edge of screen
`y`	`y >= 0`	Offset of bottom from bottom edge of screen
	`y < 0`	Offset of top from top edge of screen

You can also specify `position` as one of these position names.

Position Name	Screen Location
`'north'`	Top center
`'south'`	Bottom center
`'east'`	Right center
`'west'`	Left center
`'northeast'`	Top right corner
`'northwest'`	Top left corner
`'southeast'`	Bottom right corner
`'southwest'`	Bottom left corner

Position Name	Screen Location
`'center'`	Center
`'onscreen'`	Location nearest to the current location that is entirely on screen

More About

Pixels

Distances in pixels are independent of your system resolution on Windows and Macintosh systems:

- On Windows systems, MATLAB defines a pixel as 1/96th of an inch.
- On Macintosh systems, MATLAB defines a pixel as 1/72nd of an inch.

On Linux systems, your system resolution determines the size of a MATLAB pixel. For more information, see “DPI-Aware Behavior in MATLAB”.

Algorithms

Use with Maximized Figures

Applying `movegui` to a maximized figure window moves the window towards the taskbar and creates a gap on the opposite side of the screen about as wide as the task bar. The window might shrink in size by a few pixels. If you use the `onscreen` option with a maximized figure window, then `movegui` creates a gap on both the left and upper sides of the screen so that the top left corner of the figure is visible.

See Also

`guide` | `openfig`

Introduced before R2006a

movevars

Move variables in table or timetable

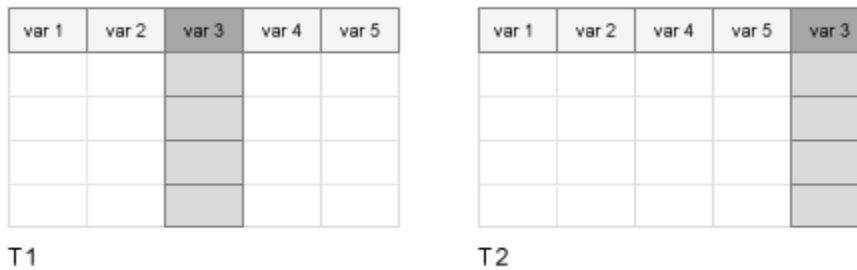
Syntax

```
T2 = movevars(T1,vars,'Before',location)
T2 = movevars(T1,vars,'After',location)
```

Description

`T2 = movevars(T1,vars,'Before',location)` moves the table variables specified by `vars` to the left of the variable specified by `location`. You can specify variables and `location` by name, by position, or using logical indices.

`T2 = movevars(T1,vars,'After',location)` moves the variables to the right of the table variable indicated by `location` (see diagram).



Examples

Move Variable

Create a table and move variables one at a time. You can specify variables by name or by position in the table.

Read data from a spreadsheet into a table. Display the first three rows.

```
T1 = readtable('outages.csv');
head(T1,3)
```

```
ans=3×6 table
       Region           OutageTime        Loss     Customers     RestorationTime
    _____________    ________________    ______    __________    ________________    __

    {'SouthWest'}    2002-02-01 12:18    458.98    1.8202e+06    2002-02-07 16:50    {
    {'SouthEast'}    2003-01-23 00:49    530.14    2.1204e+05                 NaT    {
    {'SouthEast'}    2003-02-07 21:15     289.4    1.4294e+05    2003-02-17 08:14    {
```

Move the variable that is named `Region` so that it is before the variable named `Cause`.

```
T2 = movevars(T1,'Region','Before','Cause');
head(T2,3)
```

```
ans=3×6 table
       OutageTime        Loss     Customers     RestorationTime        Region
    ________________    ______    __________    ________________    _____________    __

    2002-02-01 12:18    458.98    1.8202e+06    2002-02-07 16:50    {'SouthWest'}    {
    2003-01-23 00:49    530.14    2.1204e+05                 NaT    {'SouthEast'}    {
    2003-02-07 21:15     289.4    1.4294e+05    2003-02-17 08:14    {'SouthEast'}    {
```

Move the fourth variable so that it is after the first variable.

```
T3 = movevars(T2,4,'After',1);
head(T3,3)
```

```
ans=3×6 table
       OutageTime       RestorationTime      Loss     Customers        Region
    ________________    ________________    ______    __________    _____________    __

    2002-02-01 12:18    2002-02-07 16:50    458.98    1.8202e+06    {'SouthWest'}    {
    2003-01-23 00:49                 NaT    530.14    2.1204e+05    {'SouthEast'}    {
    2003-02-07 21:15    2003-02-17 08:14     289.4    1.4294e+05    {'SouthEast'}    {
```

Specify Multiple Variables

Move multiple table variables using the `movevars` function. You can specify variables by name or by position.

Read data from a spreadsheet into a table.

```
T1 = readtable('outages.csv');
head(T1,3)
```

```
ans=3×6 table
       Region           OutageTime        Loss     Customers     RestorationTime
    _____________    ________________    ______    __________    _______________

    {'SouthWest'}    2002-02-01 12:18    458.98    1.8202e+06    2002-02-07 16:50
    {'SouthEast'}    2003-01-23 00:49    530.14    2.1204e+05                NaT
    {'SouthEast'}    2003-02-07 21:15     289.4    1.4294e+05    2003-02-17 08:14
```

Move the variables named `Loss`, `Customer`, and `Cause` so that they are before the first variable. Specify names using a cell array of character vectors.

```
T2 = movevars(T1,{'Loss','Customers','Cause'},'Before',1);
head(T2,3)
```

```
ans=3×6 table
     Loss     Customers          Cause             Region           OutageTime
    ______    __________    _________________    _____________    ________________

    458.98    1.8202e+06    {'winter storm'}    {'SouthWest'}    2002-02-01 12:18
    530.14    2.1204e+05    {'winter storm'}    {'SouthEast'}    2003-01-23 00:49
     289.4    1.4294e+05    {'winter storm'}    {'SouthEast'}    2003-02-07 21:15
```

Move the first four variables of `T2` so that they are after `RestorationTime`.

```
T3 = movevars(T2,[1:4],'After','RestorationTime');
head(T3,3)
```

```
ans=3×6 table
       OutageTime       RestorationTime      Loss     Customers          Cause
    ________________    _______________    ______    __________    _________________
```

```
    2002-02-01 12:18    2002-02-07 16:50    458.98    1.8202e+06    {'winter storm'}
    2003-01-23 00:49                 NaT    530.14    2.1204e+05    {'winter storm'}
    2003-02-07 21:15    2003-02-17 08:14     289.4    1.4294e+05    {'winter storm'}
```

Input Arguments

`T1` — Input table
table | timetable

Input table, specified as a table or timetable.

`vars` — Variables in input table
character vector | cell array of character vectors | string array | numeric array | logical array

Variables in the input table, specified as a character vector, cell array of character vectors, string array, numeric array, or logical array.

`location` — Location to insert moved variables
character vector | string scalar | integer | logical array

Location to insert moved variables, specified as a character vector, string scalar, integer, or logical array.

- If `location` is a character vector or string scalar, then it is the name of a variable in the input table `T1`.
- If `location` is the integer `n`, then it specifies the `n`th variable in `T1`.
- If `location` is a logical array, whose `n`th element is `1` (`true`), then it specifies the `n`th variable in `T1`. All other elements of `location` must be `0` (`false`).

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`addvars` | `mergevars` | `removevars` | `splitvars`

Topics

“Add, Delete, and Rearrange Table Variables”
“Add and Delete Table Rows”
“Access Data in Tables”
“Modify Units, Descriptions, and Table Variable Names”
“Clean Messy and Missing Data in Tables”

Introduced in R2018a

movie

Play recorded movie frames

Syntax

```
movie(M)
movie(M,n)
movie(M,n,fps)
movie(h,...)
movie(h,M,n,fps,loc)
```

Description

The `movie` function plays the movie defined by a matrix whose columns are movie frames (usually produced by `getframe`).

`movie(M)` plays the movie in matrix `M` once, using the current axes as the default target. If you want to play the movie in the figure instead of the axes, specify the figure handle (or `gcf`) as the first argument: `movie(figure_handle,...)`. `M` must be an array of movie frames (usually from `getframe`).

`movie(M,n)` plays the movie `n` times. If `n` is negative, each cycle is shown forward then backward. If `n` is a vector, the first element is the number of times to play the movie, and the remaining elements make up a list of frames to play in the movie.

For example, if `M` has four frames then `n = [10 4 4 2 1]` plays the movie ten times, and the movie consists of frame `4` followed by frame `4` again, followed by frame `2` and finally frame `1`.

`movie(M,n,fps)` plays the movie at `fps` frames per second. The default is 12 frames per second. Computers that cannot achieve the specified speed play as fast as possible.

`movie(h,...)` plays the movie centered in the figure or axes identified by the handle `h`. Specifying the figure or axes enables MATLAB to fit the movie to the available size.

`movie(h,M,n,fps,loc)` specifies `loc`, a four-element location vector, `[x y 0 0]`, where the lower left corner of the movie frame is anchored (only the first two elements in

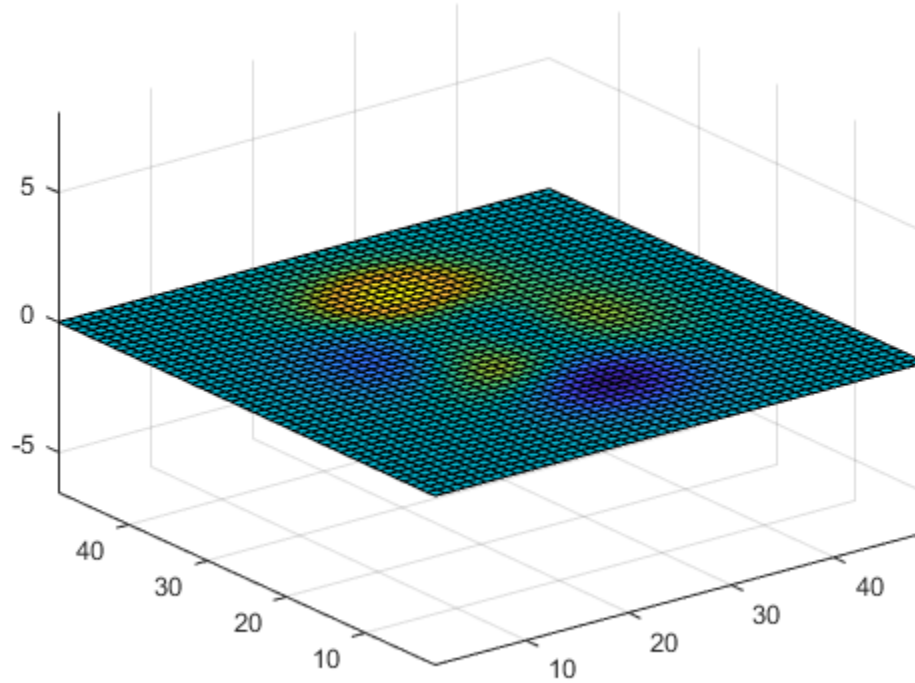
the vector are used). The location is relative to the lower left corner of the figure or axes specified by `handle` `h` and in units of pixels, regardless of the object's `Units` property.

Examples

Record Frames and Play Movie

Use the `getframe` function in a loop to record frames of the `peaks` function vibrating. Preallocate an array to store the movie frames.

```
figure
Z = peaks;
surf(Z)
axis tight manual
ax = gca;
ax.NextPlot = 'replaceChildren';


loops = 40;
F(loops) = struct('cdata',[],'colormap',[]);
for j = 1:loops
    X = sin(j*pi/10)*Z;
    surf(X,Z)
    drawnow
    F(j) = getframe;
end
```



To play the movie two times, use `movie(F,2)`.

Tips

The `movie` function uses a default figure size of 560-by-420 and does not resize figures to fit movies with larger or smaller frames. To accommodate other frame sizes, you can resize the figure to fit the movie.

`movie` only accepts 8-bit image frames; it does not accept 16-bit grayscale or 24–bit truecolor image frames.

Buffering the movie places all frames in memory. As a result, on Microsoft Windows and perhaps other platforms, a long movie (on the order of several hundred frames) can exhaust memory, depending on system resources. In such cases an error message is issued:

```
??? Error using ==> movie
Could not create movie frame
```

You can abort a movie by typing **Ctrl-C**.

`movie` is not a built-in function. Therefore, you cannot call `movie` using the `builtin` function.

Limitations with Renderer on Windows Systems

Setting the figure `Renderer` property to `painters` works around limitations of using `getframe` with the `OpenGL` renderer on some Windows systems.

See Also

`VideoReader` | `VideoWriter` | `frame2im` | `getframe` | `im2frame`

Topics

"Record Animation for Playback"

Introduced before R2006a

movmad

Moving median absolute deviation

Syntax

```
M = movmad(A,k)
M = movmad(A,[kb kf])
M = movmad( ___ ,dim)
M = movmad( ___ ,nanflag)
M = movmad( ___ ,Name,Value)
```

Description

`M = movmad(A,k)` returns an array of local `k`-point median absolute deviations (MADs) on page 1-9954, where each MAD is calculated over a sliding window of length `k` across neighboring elements of `A`. `M` is the same size as `A`.

When `k` is odd, the window is centered about the element in the current position. When `k` is even, the window is centered about the current and previous elements. The window size is automatically truncated at the endpoints when there are not enough elements to fill the window. When the window is truncated, the MAD is taken over only the elements that fill the window.

- If `A` is a vector, then `movmad` operates along the length of the vector.
- If `A` is a multidimensional array, then `movmad` operates along the first array dimension whose size does not equal 1.

`M = movmad(A,[kb kf])` computes the MAD with a window of length `kb+kf+1` that includes the element in the current position, `kb` elements backward, and `kf` elements forward.

`M = movmad( ___ ,dim)` computes the MAD along dimension `dim` for any of the previous syntaxes. For example, `movmad(A,k,2)` for a matrix `A` operates across the columns of `A`, computing the `k`-element sliding MAD for each row.

`M = movmad( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `movmad(A,k,'includenan')` includes all

`NaN` values in the calculation, which is the default. `movmad(A,k,'omitnan')` ignores them and computes the MAD over fewer points.

`M = movmad( ___ ,Name,Value)` specifies additional parameters for the moving MAD using one or more name-value pair arguments. For example, if `x` is a vector of time values, then `movmad(A,k,'SamplePoints',x)` computes the moving MAD of `A` relative to the times in `x`.

Examples

Centered Moving MAD of Vector

Compute the three-point centered moving MAD of a row vector. When there are fewer than three elements in the window at the endpoints, compute over the elements that are available.

```
A = [1 2 4 -1 -2 -3 -1 3 2 1];
M = movmad(A,3)
```

```
M = 1×10

    0.5000    1.0000    2.0000    1.0000    1.0000    1.0000    2.0000    1.0000    1.0
```

Trailing Moving MAD of Vector

Compute the three-point trailing moving MAD of a row vector. When there are fewer than three elements in the window at the endpoints, compute over the elements that are available.

```
A = [1 2 1 -1 -2 -3 -1 3 4 1];
M = movmad(A,[2 0])
```

```
M = 1×10

         0    0.5000         0    1.0000    1.0000    1.0000    1.0000    2.0000    1.0
```

Moving MAD of Matrix

Compute the 3-point centered moving MAD for each row of a matrix. The dimension argument is 2, which slides the window across the columns of `A`. The window starts on the first row, slides horizontally to the end of the row, then moves to the second row, and so on.

```
A = [1 2 1; -1 -2 -3; -1 3 4]
```

```
A = 3×3

     1     2     1
    -1    -2    -3
    -1     3     4
```

```
M = movmad(A,3,2)
```

```
M = 3×3

    0.5000         0    0.5000
    0.5000    1.0000    0.5000
    2.0000    1.0000    0.5000
```

Moving MAD of Vector with `NaN` Elements

Compute the three-point centered moving MAD of a row vector containing two `NaN` elements.

```
A = [2 1 NaN -1 -2 -3 NaN 3 4 1];
M = movmad(A,3)
```

```
M = 1×10

    0.5000       NaN       NaN       NaN    1.0000       NaN       NaN       NaN    1.0
```

Recalculate the moving MAD omitting the `NaN` values. When `movmad` discards `NaN` elements, it computes over the remaining elements in the window.

```
M = movmad(A,3,'omitnan')
```

```
M = 1×10

    0.5000    0.5000    1.0000    0.5000    1.0000    0.5000    3.0000    0.5000    1.0
```

Sample Points for Moving MAD

Compute a 3-hour centered moving MAD of the data in `A` according to the time vector `t`.

```
A = [4 8 6 -1 -2 -3];
k = hours(3);
t = datetime(2016,1,1,0,0,0) + hours(0:5)
```

```
t = 1x6 datetime array
Columns 1 through 3

   01-Jan-2016 00:00:00   01-Jan-2016 01:00:00   01-Jan-2016 02:00:00

Columns 4 through 6

   01-Jan-2016 03:00:00   01-Jan-2016 04:00:00   01-Jan-2016 05:00:00
```

```
M = movmad(A,k,'SamplePoints',t)
```

```
M = 1×6

    2.0000    2.0000    2.0000    1.0000    1.0000    0.5000
```

Return Only Full-Window MADs

Compute the three-point centered moving MAD of a row vector, but discard any calculation that uses fewer than three points from the output. In other words, return only the MADs computed from a full three-element window, discarding endpoint calculations.

```
A = [1 2 1 -1 -2 -3 -1 3 4 1];
M = movmad(A,3,'Endpoints','discard')
```

```
M = 1×8

     0     1     1     1     1     2     1     1
```

Input Arguments

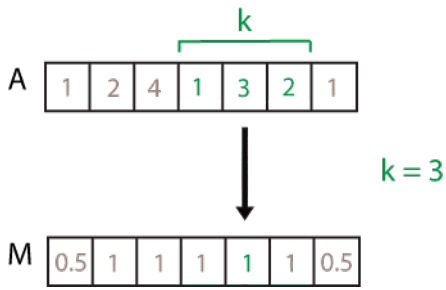
A — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

k — Window length
numeric or duration scalar

Window length, specified as a numeric or duration scalar. When `k` is a positive integer scalar, the centered MAD includes the element in the current position plus surrounding neighbors. For example, a three-point MAD defined by a window of length three results in the following calculation for a vector `A`:

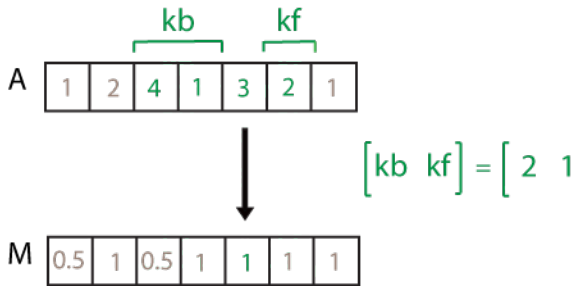


Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`[kb kf]` — Directional window length
numeric or duration row vector containing two elements

Directional window length, specified as a numeric or duration row vector containing two elements. When `kb` and `kf` are positive integer scalars, the calculation is over `kb+kf+1`

elements. The calculation includes the element in the current position, `kb` elements before the current position, and `kf` elements after the current position. For example, a four-point MAD defined by the directional window `[2 1]` results in the following calculation for a vector `A`:



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

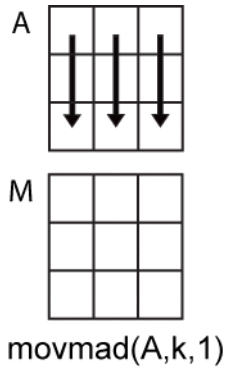
`dim` — Dimension to operate along

positive integer scalar

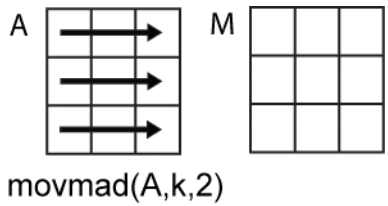
Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal `1`.

Consider a matrix `A`.

- If `dim = 1`, then `movmad(A,k,1)` starts with the first column and slides vertically over each row. The MAD is computed over `k` elements at a time. Then it moves to the second column and repeats the computation. This process continues until all columns are exhausted.



- If `dim = 2`, then `movmad(A,k,2)` starts with the first row and slides horizontally across each column, computing over `k` elements at a time. Then it moves to the second row and repeats the computation. This process continues until all rows are exhausted.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`nanflag` — `NaN` condition
`'includenan'` (default) | `'omitnan'`

`NaN` condition, specified as one of these values:

- `'includenan'` — Include `NaN` values from the input when computing the MAD, resulting in `NaN` output.
- `'omitnan'` — Ignore all `NaN` values in the input. If a window contains only `NaN` values, then `movmad` returns `NaN`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `M = movmad(A,k,'Endpoints','fill')`

`Endpoints` — Method to treat windows near endpoints
`'shrink'` (default) | `'discard'` | `'fill'` | numeric or logical scalar

Method to treat windows near endpoints, specified as the comma-separated pair consisting of `'Endpoints'` and one of the following:

`'Endpoints'` Value	Description
`'shrink'`	Shrink the window size near the endpoints of the input to include only existing elements.
`'discard'`	Do not output any MAD values when the window does not completely overlap with existing elements.
`'fill'`	Replace nonexisting elements with `NaN`.
numeric or logical scalar	Replace nonexisting elements with the specified numeric or logical value.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string`

`SamplePoints` — Sample points for computing MADs
vector

Sample points for computing MADs, specified as the comma-separated pair consisting of `'SamplePoints'` and a vector. The sample points represent the locations of the data in `A`. Sample points do not need to be uniformly sampled. By default, the sample points vector is `[1 2 3 ... ]`.

Moving windows are defined relative to the sample points, which must be sorted and contain unique elements. For example, if `t` is a vector of times corresponding to the input data, then `movmad(rand(1,10),3,'SamplePoints',t)` has a window that represents the time interval between `t(i)-1.5` and `t(i)+1.5`.

When the sample points vector has data type `datetime` or `duration`, then the moving window length must have type `duration`.

If the sample points are nonuniformly spaced and the `'Endpoints'` name-value pair is specified, then its value must be `'shrink'`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `datetime` | `duration`

More About

Median Absolute Deviation

For a random variable vector A made up of N scalar observations, the median absolute deviation (MAD) is defined as

$$\text{MAD} = \text{median}(|A_i - \text{median}(A)|)$$

for $i = 1,2,...,N$.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

The `'SamplePoints'` name-value pair is not supported.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This function supports C/C++ code generation with the limitations:

The `'SamplePoints'` name-value pair is not supported.

See Also

`median` | `movmean` | `movmedian` | `movstd` | `movvar`

Introduced in R2017a

movmax

Moving maximum

Syntax

```
M = movmax(A,k)
M = movmax(A,[kb kf])
M = movmax( ___ ,dim)
M = movmax( ___ ,nanflag)
M = movmax( ___ ,Name,Value)
```

Description

`M = movmax(A,k)` returns an array of local `k`-point maximum values, where each maximum is calculated over a sliding window of length `k` across neighboring elements of `A`. When `k` is odd, the window is centered about the element in the current position. When `k` is even, the window is centered about the current and previous elements. The window size is automatically truncated at the endpoints when there are not enough elements to fill the window. When the window is truncated, the maximum is taken over only the elements that fill the window. `M` is the same size as `A`.

- If `A` is a vector, then `movmax` operates along the length of the vector.
- If `A` is a multidimensional array, then `movmax` operates along the first array dimension whose size does not equal 1.

`M = movmax(A,[kb kf])` computes the maximum with a window of length `kb+kf+1` that includes the element in the current position, `kb` elements backward, and `kf` elements forward.

`M = movmax( ___ ,dim)` returns the array of moving maximums along dimension `dim` for any of the previous syntaxes. For example, if `A` is a matrix, then `movmax(A,k,2)` operates along the columns of `A`, computing the `k`-element sliding maximum for each row.

`M = movmax( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `movmax(A,k,'includenan')` includes all

`NaN` values in the calculation while `movmax(A,k,'omitnan')` ignores them and computes the maximum over fewer points.

`M = movmax( ___ ,Name,Value)` specifies additional parameters for the moving maximum using one or more name-value pair arguments. For example, if `x` is a vector of time values, then `movmax(A,k,'SamplePoints',x)` computes the moving maximum relative to the times in `x`.

Examples

Centered Moving Maximum of Vector

Compute the three-point centered moving maximum of a row vector. When there are fewer than three elements in the window at the endpoints, take the maximum over the elements that are available.

```
A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movmax(A,3)
```

```
M = 1×10

     8     8     8     6    -1    -1     3     4     5     5
```

Trailing Moving Maximum of Vector

Compute the three-point trailing moving maximum of a row vector. When there are fewer than three elements in the window at the endpoints, `movmax` takes the maximum over the number of elements that are available.

```
A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movmax(A,[2 0])
```

```
M = 1×10

     4     8     8     8     6    -1    -1     3     4     5
```

Moving Maximum of Matrix

Compute the three-point centered moving maximum for each row of a matrix. The window starts on the first row, slides horizontally to the end of the row, then moves to the second row, and so on. The dimension argument is two, which slides the window across the columns of A.

```
A = [4 8 6; -1 -2 -3; -1 3 4]
```

```
A = 3×3

     4     8     6
    -1    -2    -3
    -1     3     4
```

```
M = movmax(A,3,2)
```

```
M = 3×3

     8     8     8
    -1    -1    -2
     3     4     4
```

Moving Maximum of Vector with NaN Elements

Compute the three-point centered moving maximum of a row vector containing two NaN elements.

```
A = [4 8 NaN -1 -2 -3 NaN 3 4 5];
M = movmax(A,3)
```

```
M = 1×10

     8     8     8    -1    -1    -2     3     4     5     5
```

Recalculate the maximum, but include the NaN values. When taking the maximum over a group of elements containing at least one NaN value, movmax returns NaN.

```
M = movmax(A,3,'includenan')
```

```
M = 1×10

     8   NaN   NaN   NaN    -1   NaN   NaN   NaN     5     5
```

Sample Points for Moving Maximum

Compute a 3-hour centered moving maximum of the data in `A` according to the time vector `t`.

```
A = [4 8 6 -1 -2 -3];
k = hours(3);
t = datetime(2016,1,1,0,0,0) + hours(0:5)
```

```
t = 1x6 datetime array
Columns 1 through 3

   01-Jan-2016 00:00:00   01-Jan-2016 01:00:00   01-Jan-2016 02:00:00

Columns 4 through 6

   01-Jan-2016 03:00:00   01-Jan-2016 04:00:00   01-Jan-2016 05:00:00
```

```
M = movmax(A,k,'SamplePoints',t)
```

```
M = 1×6

     8     8     8     6    -1    -2
```

Return Only Full-Window Maximums

Compute the three-point centered moving maximum of a row vector, but discard any calculation that uses fewer than three points from the output. In other words, return only the maximums computed from a full three-element window, discarding endpoint calculations.

```
A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movmax(A,3,'Endpoints','discard')
```

```
M = 1×8

     8     8     6    -1    -1     3     4     5
```

Input Arguments

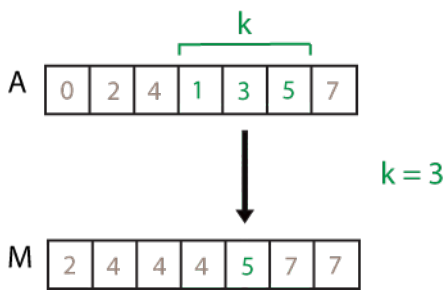
A — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

k — Window length
numeric or duration scalar

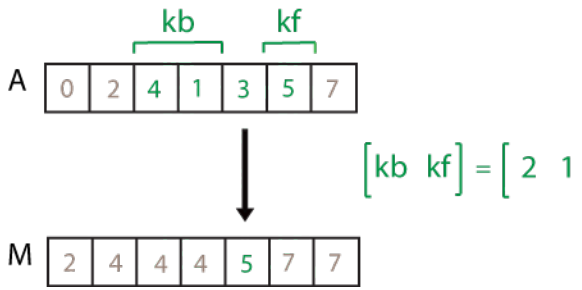
Window length, specified as a numeric or duration scalar. When `k` is a positive integer scalar, the centered maximum includes the element in the current position plus surrounding neighbors. For example, a three-point maximum defined by a window of length three results in the following calculation for a vector `A`:



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`[kb kf]` — Directional window length
numeric or duration row vector containing two elements

Directional window length, specified as a numeric or duration row vector containing two elements. When `kb` and `kf` are positive integer scalars, the calculation is over `kb+kf+1` elements. The calculation includes the element in the current position, `kb` elements before the current position, and `kf` elements after the current position. For example, a four-point maximum defined by the directional window `[2 1]` results in the following calculation for a vector `A`:



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`dim` — Dimension to operate along

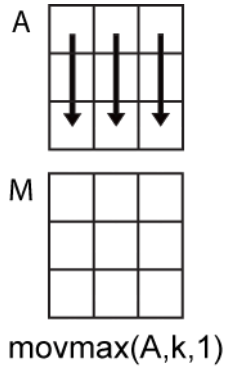
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

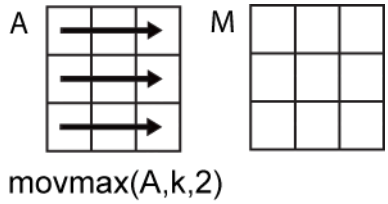
Dimension `dim` indicates the dimension that `movmax` operates along, that is, the direction in which the specified window slides.

Consider a two-dimensional input array, `A`.

- If `dim = 1`, then `movmax(A,k,1)` starts with the first column and slides vertically over each row to compute. The maximum is computed over `k` elements at a time. Then it moves to the second column and repeats the computation. This process continues until all columns are exhausted.



- If `dim = 2`, then `movmax(A,k,2)` starts with the first row and slides horizontally across each column. The maximum is computed over `k` elements at a time. Then it moves to the second row and repeats the computation. This process continues until all rows are exhausted.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`nanflag` — `NaN` condition
`'omitnan'` (default) | `'includenan'`

`NaN` condition, specified as one of these values:

- `'omitnan'` — Ignore all `NaN` values in the input. If a window includes only `NaN` values, then `movmax` returns `NaN`.
- `'includenan'` — Include `NaN` values from the input when computing the maximum, resulting in the output `NaN`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `M = movmax(A,k,'Endpoints','fill')`

`Endpoints` — Method to treat leading and trailing windows

`'shrink'` (default) | `'discard'` | `'fill'` | numeric or logical scalar

Method to treat leading and trailing windows, specified as the comma-separated pair consisting of `'Endpoints'` and one of the following:

`'Endpoints'` Value	Description
`'shrink'`	Shrink the window size near the endpoints of the input to include only existing elements.
`'discard'`	Do not output any maximum values when the window does not completely overlap with existing elements.
`'fill'`	Substitute nonexisting elements with `-Inf`.
numeric or logical scalar	Substitute nonexisting elements with a specified numeric or logical value.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string`

`SamplePoints` — Sample points for computing maximums

vector

Sample points for computing maximums, specified as the comma-separated pair consisting of `'SamplePoints'` and a vector. The sample points represent the location of the data in `A`. Sample points do not need to be uniformly sampled. By default, the sample points vector is `[1 2 3 ... ]`.

Moving windows are defined relative to the sample points, which must be sorted and contain unique elements. For example, if `t` is a vector of times corresponding to the input data, then `movmax(rand(1,10),3,'SamplePoints',t)` has a window that represents the time interval between `t(i)-1.5` and `t(i)+1.5`.

When the sample points vector has data type `datetime` or `duration`, then the moving window length must have type `duration`.

If the sample points are nonuniformly spaced and the `'Endpoints'` name-value pair is specified, then its value must be `'shrink'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `datetime` | `duration`

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

The `'SamplePoints'` name-value pair is not supported.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This function supports C/C++ code generation with the limitations:

The `'SamplePoints'` name-value pair is not supported.

See Also

`cummax` | `max` | `movmin` | `movsum`

Introduced in R2016a

movmean

Moving mean

Syntax

```
M = movmean(A,k)
M = movmean(A,[kb kf])
M = movmean( ___ ,dim)
M = movmean( ___ ,nanflag)
M = movmean( ___ ,Name,Value)
```

Description

`M = movmean(A,k)` returns an array of local `k`-point mean on page 1-9973 values, where each mean is calculated over a sliding window of length `k` across neighboring elements of `A`. When `k` is odd, the window is centered about the element in the current position. When `k` is even, the window is centered about the current and previous elements. The window size is automatically truncated at the endpoints when there are not enough elements to fill the window. When the window is truncated, the average is taken over only the elements that fill the window. `M` is the same size as `A`.

- If `A` is a vector, then `movmean` operates along the length of the vector.
- If `A` is a multidimensional array, then `movmean` operates along the first array dimension whose size does not equal 1.

`M = movmean(A,[kb kf])` computes the mean with a window of length `kb+kf+1` that includes the element in the current position, `kb` elements backward, and `kf` elements forward.

`M = movmean( ___ ,dim)` returns the array of moving averages along dimension `dim` for any of the previous syntaxes. For example, if `A` is a matrix, then `movmean(A,k,2)` operates along the columns of `A`, computing the `k`-element sliding mean for each row.

`M = movmean( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `movmean(A,k,'includenan')` includes all

`NaN` values in the calculation while `movmean(A,k,'omitnan')` ignores them and computes the mean over fewer points.

`M = movmean( ___ ,Name,Value)` specifies additional parameters for the moving average using one or more name-value pair arguments. For example, if `x` is a vector of time values, then `movmean(A,k,'SamplePoints',x)` computes the moving average relative to the times in `x`.

Examples

Centered Moving Average of Vector

Compute the three-point centered moving average of a row vector. When there are fewer than three elements in the window at the endpoints, take the average over the elements that are available.

```
A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movmean(A,3)
```

```
M = 1×10

    6.0000    6.0000    4.3333    1.0000   -2.0000   -2.0000   -0.3333    2.0000    4.0
```

Trailing Moving Average of Vector

Compute the three-point trailing moving average of a row vector. When there are fewer than three elements in the window at the endpoints, take the average over the elements that are available.

```
A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movmean(A,[2 0])
```

```
M = 1×10

    4.0000    6.0000    6.0000    4.3333    1.0000   -2.0000   -2.0000   -0.3333    2.0
```

Moving Average of Matrix

Compute the three-point centered moving average for each row of a matrix. The window starts on the first row, slides horizontally to the end of the row, then moves to the second row, and so on. The dimension argument is two, which slides the window across the columns of A.

```
A = [4 8 6; -1 -2 -3; -1 3 4]
```

```
A = 3×3

     4     8     6
    -1    -2    -3
    -1     3     4
```

```
M = movmean(A,3,2)
```

```
M = 3×3

    6.0000    6.0000    7.0000
   -1.5000   -2.0000   -2.5000
    1.0000    2.0000    3.5000
```

Moving Average of Vector with NaN Elements

Compute the three-point centered moving average of a row vector containing two NaN elements.

```
A = [4 8 NaN -1 -2 -3 NaN 3 4 5];
M = movmean(A,3)
```

```
M = 1×10

    6.0000       NaN       NaN       NaN   -2.0000       NaN       NaN       NaN    4.0
```

Recalculate the average, but omit the NaN values. When movmean discards NaN elements, it takes the average over the remaining elements in the window.

```
M = movmean(A,3,'omitnan')
```

```
M = 1×10

    6.0000    6.0000    3.5000   -1.5000   -2.0000   -2.5000         0    3.5000    4.0
```

Sample Points for Moving Average

Compute a 3-hour centered moving average of the data in `A` according to the time vector `t`.

```
A = [4 8 6 -1 -2 -3];
k = hours(3);
t = datetime(2016,1,1,0,0,0) + hours(0:5)
```

```
t = 1x6 datetime array
Columns 1 through 3

   01-Jan-2016 00:00:00   01-Jan-2016 01:00:00   01-Jan-2016 02:00:00

Columns 4 through 6

   01-Jan-2016 03:00:00   01-Jan-2016 04:00:00   01-Jan-2016 05:00:00
```

```
M = movmean(A,k,'SamplePoints',t)
```

```
M = 1×6

    6.0000    6.0000    4.3333    1.0000   -2.0000   -2.5000
```

Return Only Full-Window Averages

Compute the three-point centered moving average of a row vector, but discard any calculation that uses fewer than three points from the output. In other words, return only the averages computed from a full three-element window, discarding endpoint calculations.

```
A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movmean(A,3,'Endpoints','discard')
```

```
M = 1×8

    6.0000    4.3333    1.0000   -2.0000   -2.0000   -0.3333    2.0000    4.0000
```

Input Arguments

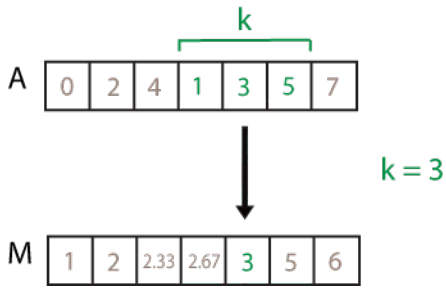
A — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

k — Window length
numeric or duration scalar

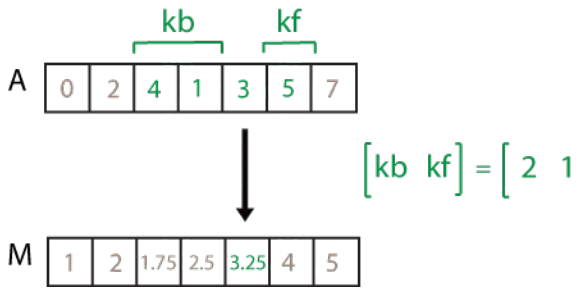
Window length, specified as a numeric or duration scalar. When `k` is a positive integer scalar, the centered average includes the element in the current position plus surrounding neighbors. For example, a three-point average defined by a window of length three results in the following calculation for a vector `A`:



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`[kb kf]` — Directional window length
numeric or duration row vector containing two elements

Directional window length, specified as a numeric or duration row vector containing two elements. When `kb` and `kf` are positive integer scalars, the calculation is over `kb+kf+1` elements. The calculation includes the element in the current position, `kb` elements before the current position, and `kf` elements after the current position. For example, a four-point average defined by the directional window `[2 1]` results in the following calculation for a vector `A`:



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`dim` — Dimension to operate along

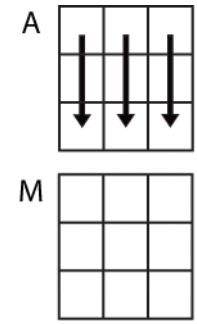
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Dimension `dim` indicates the dimension that `movmean` operates along, that is, the direction in which the specified window slides.

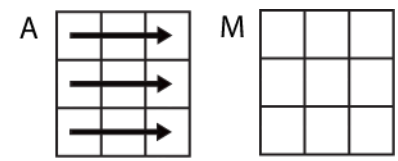
Consider a two-dimensional input array, `A`.

- If `dim = 1`, then `movmean(A,k,1)` starts with the first column and slides vertically over each row. The mean is computed over `k` elements at a time. Then it moves to the second column and repeats the computation. This process continues until all columns are exhausted.



movmean(A,k,1)

- If `dim = 2`, then `movmean(A,k,2)` starts with the first row and slides horizontally across each column. The mean is computed over `k` elements at a time. Then it moves to the second row and repeats the computation. This process continues until all rows are exhausted.



movmean(A,k,2)

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`nanflag` — `NaN` condition
`'includenan'` (default) | `'omitnan'`

`NaN` condition, specified as one of these values:

- `'includenan'` — Include `NaN` values from the input when computing the mean, resulting in `NaN` output.
- `'omitnan'` — Ignore all `NaN` values in the input. If a window contains only `NaN` values, then `movmean` returns `NaN`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `M = movmean(A,k,'Endpoints','fill')`

`Endpoints` — Method to treat leading and trailing windows

`'shrink'` (default) | `'discard'` | `'fill'` | numeric or logical scalar

Method to treat leading and trailing windows, specified as the comma-separated pair consisting of `'Endpoints'` and one of the following:

`'Endpoints'` Value	Description
`'shrink'`	Shrink the window size near the endpoints of the input to include only existing elements.
`'discard'`	Do not output any averages when the window does not completely overlap with existing elements.
`'fill'`	Substitute nonexisting elements with `NaN`.
numeric or logical scalar	Substitute nonexisting elements with a specified numeric or logical value.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string`

`SamplePoints` — Sample points for computing averages

vector

Sample points for computing averages, specified as the comma-separated pair consisting of `'SamplePoints'` and a vector. The sample points represent the location of the data in `A`. Sample points do not need to be uniformly sampled. By default, the sample points vector is `[1 2 3 ... ]`.

Moving windows are defined relative to the sample points, which must be sorted and contain unique elements. For example, if `t` is a vector times corresponding to the input data, then `movmean(rand(1,10),3,'SamplePoints',t)` has a window that represents the time interval between `t(i)-1.5` and `t(i)+1.5`.

When the sample points vector has data type `datetime` or `duration`, then the moving window length must have type `duration`.

If the sample points are nonuniformly spaced and the `'Endpoints'` name-value pair is specified, then its value must be `'shrink'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `datetime` | `duration`

More About

Mean

For a random variable vector A made up of N scalar observations, the mean is defined as

$$\mu = \frac{1}{N}\sum_{i=1}^{N} A_i .$$

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

The `'SamplePoints'` name-value pair is not supported.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This function supports C/C++ code generation with the limitations:

The `'SamplePoints'` name-value pair is not supported.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `'SamplePoints'` name-value pair is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`mean` | `movmad` | `movmedian` | `movstd` | `movsum` | `movvar` | `smoothdata`

Topics

“Data Smoothing and Outlier Detection”

Introduced in R2016a

movmedian

Moving median

Syntax

```
M = movmedian(A,k)
M = movmedian(A,[kb kf])
M = movmedian( ___ ,dim)
M = movmedian( ___ ,nanflag)
M = movmedian( ___ ,Name,Value)
```

Description

`M = movmedian(A,k)` returns an array of local `k`-point median values, where each median is calculated over a sliding window of length `k` across neighboring elements of `A`. When `k` is odd, the window is centered about the element in the current position. When `k` is even, the window is centered about the current and previous elements. The window size is automatically truncated at the endpoints when there are not enough elements to fill the window. When the window is truncated, the median is taken over only the elements that fill the window. `M` is the same size as `A`.

- If `A` is a vector, then `movmedian` operates along the length of the vector.
- If `A` is a multidimensional array, then `movmedian` operates along the first array dimension whose size does not equal 1.

`M = movmedian(A,[kb kf])` computes the median with a window of length `kb+kf+1` that includes the element in the current position, `kb` elements backward, and `kf` elements forward.

`M = movmedian( ___ ,dim)` returns the array of moving medians along dimension `dim` for any of the previous syntaxes. For example, if `A` is a matrix, then `movmedian(A,k,2)` operates along the columns of `A`, computing the `k`-element sliding median for each row.

`M = movmedian( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `movmedian(A,k,'includenan')`

includes all `NaN` values in the calculation while `movmedian(A,k,'omitnan')` ignores them and computes the median over fewer points.

`M = movmedian( ___ ,Name,Value)` specifies additional parameters for the moving median using one or more name-value pair arguments. For example, if `x` is a vector of time values, then `movmedian(A,k,'SamplePoints',x)` computes the moving median relative to the times in `x`.

Examples

Centered Moving Median of Vector

Compute the three-point centered moving median of a row vector. When there are fewer than three elements in the window at the endpoints, take the average over the elements that are available.

```
A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movmedian(A,3)
```

```
M = 1×10

    6.0000    6.0000    6.0000   -1.0000   -2.0000   -2.0000   -1.0000    3.0000    4.0
```

Trailing Moving Median of Vector

Compute the three-point trailing moving median of a row vector. When there are fewer than three elements in the window at the endpoints, take the average over the elements that are available.

```
A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movmedian(A,[2 0])
```

```
M = 1×10

     4     6     6     6    -1    -2    -2    -1     3     4
```

Moving Median of Matrix

Compute the three-point centered moving median for each row of a matrix. The window starts on the first row, slides horizontally to the end of the row, then moves to the second row, and so on. The dimension argument is two, which slides the window across the columns of `A`.

```
A = [4 8 6; -1 -2 -3; -1 3 4]
```

```
A = 3×3

     4     8     6
    -1    -2    -3
    -1     3     4
```

```
M = movmedian(A,3,2)
```

```
M = 3×3

    6.0000    6.0000    7.0000
   -1.5000   -2.0000   -2.5000
    1.0000    3.0000    3.5000
```

Moving Median of Vector with `NaN` Elements

Compute the three-point centered moving median of a row vector containing `NaN` elements.

```
A = [4 8 NaN -1 -2 -3 NaN 3 4 5];
M = movmedian(A,3)
```

```
M = 1×10

    6.0000       NaN       NaN       NaN   -2.0000       NaN       NaN       NaN    4.0
```

Recalculate the median, but omit the `NaN` values. When `movmedian` discards `NaN` elements, it takes the median over the remaining elements in the window.

```
M = movmedian(A,3,'omitnan')
```

```
M = 1×10

    6.0000    6.0000    3.5000   -1.5000   -2.0000   -2.5000         0    3.5000    4.0
```

Sample Points for Moving Median

Compute a 3-hour centered moving median of the data in `A` according to the time vector `t`.

```
A = [4 8 6 -1 -2 -3];
k = hours(3);
t = datetime(2016,1,1,0,0,0) + hours(0:5)
```

```
t = 1x6 datetime array
Columns 1 through 3

   01-Jan-2016 00:00:00   01-Jan-2016 01:00:00   01-Jan-2016 02:00:00

Columns 4 through 6

   01-Jan-2016 03:00:00   01-Jan-2016 04:00:00   01-Jan-2016 05:00:00
```

```
M = movmedian(A,k,'SamplePoints',t)
```

```
M = 1×6

    6.0000    6.0000    6.0000   -1.0000   -2.0000   -2.5000
```

Return Only Full-Window Medians

Compute the three-point centered moving median of a row vector, but discard any calculation that uses fewer than three points from the output. In other words, return only the medians computed from a full three-element window, discarding endpoint calculations.

```
A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movmedian(A,3,'Endpoints','discard')
```

```
M = 1×8

     6     6    -1    -2    -2    -1     3     4
```

Input Arguments

A — Input array

vector | matrix | multidimensional array

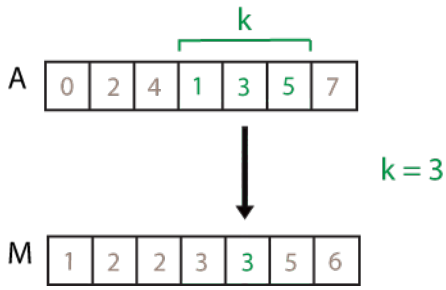
Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

k — Window length

numeric or duration scalar

Window length, specified as a numeric or duration scalar. When `k` is a positive integer scalar, the centered median includes the element in the current position plus surrounding neighbors. For example, a three-point median defined by a window of length three results in the following calculation for a vector `A`:

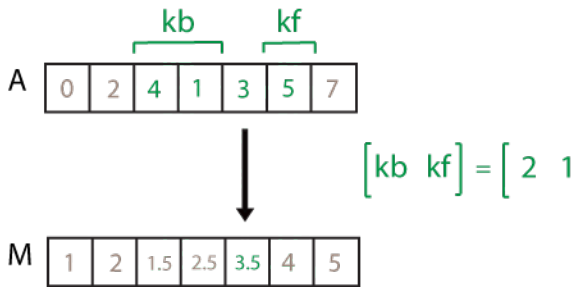


Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

[kb kf] — Directional window length

numeric or duration row vector containing two elements

Directional window length, specified as a numeric or duration row vector containing two elements. When `kb` and `kf` are positive integer scalars, the calculation is over `kb+kf+1` elements. The calculation includes the element in the current position, `kb` elements before the current position, and `kf` elements after the current position. For example, a four-point median defined by the directional window `[2 1]` results in the following calculation for a vector `A`:



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`dim` — Dimension to operate along

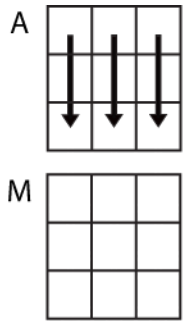
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Dimension `dim` indicates the dimension that `movmedian` operates along, that is, the direction in which the specified window slides.

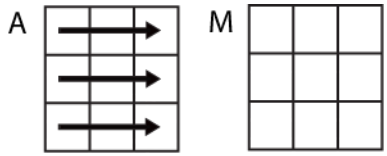
Consider a two-dimensional input array, `A`.

- If `dim = 1`, then `movmedian(A,k,1)` starts with the first column and slides vertically over each row. The median is computed over `k` elements at a time. Then it moves to the second column and repeats the computation. This process continues until all columns are exhausted.



movmedian(A,k,1)

- If `dim = 2`, then `movmedian(A,k,2)` starts with the first row and slides horizontally across each column. The median is computed over `k` elements at a time. Then it moves to the second row and repeats the computation. This process continues until all rows are exhausted.



movmedian(A,k,2)

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`nanflag` — `NaN` condition
`'includenan'` (default) | `'omitnan'`

`NaN` condition, specified as one of these values:

- `'includenan'` — Include `NaN` values from the input when computing the median, resulting in the output `NaN`.
- `'omitnan'` — Ignore all `NaN` values in the input. If a window contains only `NaN` values, then `movmedian` returns `NaN`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `M = movmedian(A,k,'Endpoints','fill')`

`Endpoints` — Method to treat leading and trailing windows

`'shrink'` (default) | `'discard'` | `'fill'` | numeric or logical scalar

Method to treat leading and trailing windows, specified as the comma-separated pair consisting of `'Endpoints'` and one of the following:

`'Endpoints'` Value	Description
`'shrink'`	Shrink the window size near the endpoints of the input to include only existing elements.
`'discard'`	Do not output any medians when the window does not completely overlap with existing elements.
`'fill'`	Substitute nonexisting elements with `NaN`.
numeric or logical scalar	Substitute nonexisting elements with a specified numeric or logical value.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string`

`SamplePoints` — Sample points for computing medians

vector

Sample points for computing medians, specified as the comma-separated pair consisting of `'SamplePoints'` and a vector. The sample points represent the location of the data in `A`. Sample points do not need to be uniformly sampled. By default, the sample points vector is `[1 2 3 ... ]`.

Moving windows are defined relative to the sample points, which must be sorted and contain unique elements. For example, if `t` is a vector of times corresponding to the input data, then `movmedian(rand(1,10),3,'SamplePoints',t)` has a window that represents the time interval between `t(i)-1.5` and `t(i)+1.5`.

When the sample points vector has data type `datetime` or `duration`, then the moving window length must have type `duration`.

If the sample points are nonuniformly spaced and the `'Endpoints'` name-value pair is specified, then its value must be `'shrink'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `datetime` | `duration`

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

The `'SamplePoints'` name-value pair is not supported.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This function supports C/C++ code generation with the limitations:

The `'SamplePoints'` name-value pair is not supported.

See Also

`median` | `movmad` | `movmean` | `movstd` | `movvar`

Topics

“Data Smoothing and Outlier Detection”

Introduced in R2016a

movmin

Moving minimum

Syntax

```
M = movmin(A,k)
M = movmin(A,[kb kf])
M = movmin( ___ ,dim)
M = movmin( ___ ,nanflag)
M = movmin( ___ ,Name,Value)
```

Description

`M = movmin(A,k)` returns an array of local `k`-point centered minimum values, where each minimum is calculated over a sliding window of length `k` across neighboring elements of `A`. When `k` is odd, the window is centered about the element in the current position. When `k` is even, the window is centered about the current and previous elements. The window size is automatically truncated at the endpoints when there are not enough elements to fill the window. When the window is truncated, the minimum is taken over only the elements that fill the window. `M` is the same size as `A`.

- If `A` is a vector, then `movmin` operates along the length of the vector.
- If `A` is a multidimensional array, then `movmin` operates along the first array dimension whose size does not equal 1.

`M = movmin(A,[kb kf])` computes the minimum with a window of length `kb+kf+1` that includes the element in the current position, `kb` elements backward, and `kf` elements forward.

`M = movmin( ___ ,dim)` returns the array of sliding minimums along dimension `dim` for any of the previous syntaxes. For example, if `A` is a matrix, then `movmin(A,k,2)` operates along the columns of `A`, computing the `k`-element sliding minimum for each row.

`M = movmin( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `movmin(A,k,'includenan')` includes all

`NaN` values in the calculation while `movmin(A,k,'omitnan')` ignores them and computes the minimum over fewer points.

`M = movmin( ___ ,Name,Value)` specifies additional parameters for the minimum using one or more name-value pair arguments. For example, if `x` is a vector of time values, then `movmin(A,k,'SamplePoints',x)` computes the moving minimum relative to the times in `x`.

Examples

Centered Moving Minimum of Vector

Compute the three-point centered moving minimum of a row vector. When there are fewer than three elements in the window at the endpoints, take the minimum over the elements that are available.

```
A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movmin(A,3)
```

```
M = 1×10

     4     4    -1    -2    -3    -3    -3    -1     3     4
```

Trailing Moving Minimum of Vector

Compute the three-point trailing moving minimum of a row vector. When there are fewer than three elements in the window at the endpoints, take the minimum over the elements that are available.

```
A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movmin(A,[2 0])
```

```
M = 1×10

     4     4     4    -1    -2    -3    -3    -3    -1     3
```

Moving Minimum of Matrix

Compute the three-point centered moving minimum for each row of a matrix. The window starts on the first row, slides horizontally to the end of the row, then moves to the second row, and so on. The dimension argument is two, which slides the window across the columns of `A`.

```
A = [4 8 6; -1 -2 -3; -1 3 4]
```

```
A = 3×3

     4     8     6
    -1    -2    -3
    -1     3     4
```

```
M = movmin(A,3,2)
```

```
M = 3×3

     4     4     6
    -2    -3    -3
    -1    -1     3
```

Moving Minimum of Vector with `NaN` Elements

Compute the three-point centered moving minimum of a row vector containing two `NaN` elements.

```
A = [4 8 NaN -1 -2 -3 NaN 3 4 5];
M = movmin(A,3)
```

```
M = 1×10

     4     4    -1    -2    -3    -3    -3     3     3     4
```

Recalculate the minimum, but include the `NaN` values. When taking the minimum over a group of elements containing at least one `NaN` value, `movmin` returns `NaN`.

```
M = movmin(A,3,'includenan')
```

```
M = 1×10

     4   NaN   NaN   NaN    -3   NaN   NaN   NaN     3     4
```

Sample Points for Moving Minimum

Compute a 3-hour centered moving minimum of the data in `A` according to the time vector `t`.

```
A = [4 8 6 -1 -2 -3];
k = hours(3);
t = datetime(2016,1,1,0,0,0) + hours(0:5)
```

```
t = 1x6 datetime array
Columns 1 through 3

   01-Jan-2016 00:00:00   01-Jan-2016 01:00:00   01-Jan-2016 02:00:00

Columns 4 through 6

   01-Jan-2016 03:00:00   01-Jan-2016 04:00:00   01-Jan-2016 05:00:00
```

```
M = movmin(A,k,'SamplePoints',t)
```

```
M = 1×6

     4     4    -1    -2    -3    -3
```

Return Only Full-Window Minimums

Compute the three-point centered moving minimum of a row vector, but discard any calculation that uses fewer than three points from the output. In other words, return only the minimums computed from a full three-element window, discarding endpoint calculations.

```
A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movmin(A,3,'Endpoints','discard')
```

```
M = 1×8

     4    -1    -2    -3    -3    -3    -1     3
```

Input Arguments

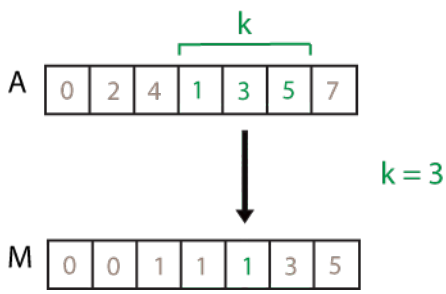
A — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

k — Window length
numeric or duration scalar

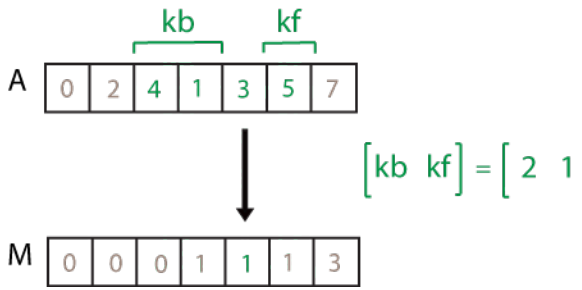
Window length, specified as a numeric or duration scalar. When `k` is a positive integer scalar, the centered minimum includes the element in the current position plus surrounding neighbors. For example, a three-point minimum defined by a window of length three results in the following calculation for a vector `A`:



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`[kb kf]` — Directional window length
numeric or duration row vector containing two elements

Directional window length, specified as a numeric or duration row vector containing two elements. When `kb` and `kf` are positive integer scalars, the calculation is over `kb+kf+1` elements. The calculation includes the element in the current position, `kb` elements before the current position, and `kf` elements after the current position. For example, a four-point minimum defined by the directional window `[2 1]` results in the following calculation for a vector `A`:



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`dim` — Dimension to operate along

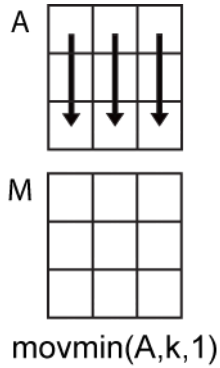
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

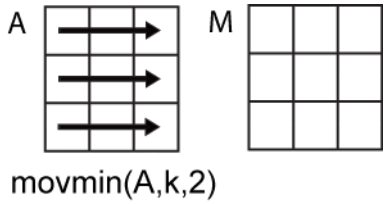
Dimension `dim` indicates the dimension that `movmin` operates along, that is, the direction in which the specified window slides.

Consider a two-dimensional input array, `A`.

- If `dim = 1`, then `movmin(A,k,1)` starts with the first column and slides vertically over each row. The minimum is computed over `k` elements at a time. Then it moves to the second column and repeats the computation. This process continues until all columns are exhausted.



- If `dim = 2`, then `movmin(A,k,2)` starts with the first row and slides horizontally across each column. The minimum is computed over `k` elements at a time. Then it moves to the second row and repeats the computation. This process continues until all rows are exhausted.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`nanflag` — `NaN` condition
`'omitnan'` (default) | `'includenan'`

`NaN` condition, specified as one of these values:

- `'omitnan'` — Ignore all `NaN` values in the input. If a window contains only `NaN` values, then `movmin` returns `NaN`.
- `'includenan'` — Include `NaN` values from the input when computing the minimum, resulting in the output `NaN`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `M = movmin(A,k,'Endpoints','fill')`

`Endpoints` — Method to treat leading and trailing windows

`'shrink'` (default) | `'discard'` | `'fill'` | numeric or logical scalar

Method to treat leading and trailing windows, specified as the comma-separated pair consisting of `'Endpoints'` and one of the following:

`'Endpoints'` Value	Description
`'shrink'`	Shrink the window size near the endpoints of the input to include only existing elements.
`'discard'`	Do not output any minimums when the window does not completely overlap with existing elements.
`'fill'`	Substitute nonexisting elements with `-Inf`.
numeric or logical scalar	Substitute nonexisting elements with a specified numeric or logical value.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string`

`SamplePoints` — Sample points for computing minimums

vector

Sample points for computing minimums, specified as the comma-separated pair consisting of `'SamplePoints'` and a vector. The sample points represent the location of the data in `A`. Sample points do not need to be uniformly sampled. By default, the sample points vector is `[1 2 3 ... ]`.

Moving windows are defined relative to the sample points, which must be sorted and contain unique elements. For example, if `t` is a vector of times corresponding to the input data, then `movmin(rand(1,10),3,'SamplePoints',t)` has a window that represents the time interval between `t(i)-1.5` and `t(i)+1.5`.

When the sample points vector has data type `datetime` or `duration`, then the moving window length must have type `duration`.

If the sample points are nonuniformly spaced and the `'Endpoints'` name-value pair is specified, then its value must be `'shrink'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `datetime` | `duration`

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

The `'SamplePoints'` name-value pair is not supported.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This function supports C/C++ code generation with the limitations:

The `'SamplePoints'` name-value pair is not supported.

See Also

`cummin` | `min` | `movmax` | `movsum`

Introduced in R2016a

movprod

Moving product

Syntax

```
M = movprod(A,k)
M = movprod(A,[kb kf])
M = movprod( ___ ,dim)
M = movprod( ___ ,nanflag)
M = movprod( ___ ,Name,Value)
```

Description

`M = movprod(A,k)` returns an array of local `k`-point products, where each product is calculated over a sliding window of length `k` across neighboring elements of `A`. `M` is the same size as `A`.

When `k` is odd, the window is centered about the element in the current position. When `k` is even, the window is centered about the current and previous elements. The window size is automatically truncated at the endpoints when there are not enough elements to fill the window. When the window is truncated, the product is taken over only the elements that fill the window.

- If `A` is a vector, then `movprod` operates along the length of the vector.
- If `A` is a multidimensional array, then `movprod` operates along the first array dimension whose size does not equal 1.

`M = movprod(A,[kb kf])` computes the product with a window of length `kb+kf+1` that includes the element in the current position, `kb` elements backward, and `kf` elements forward.

`M = movprod( ___ ,dim)` computes the product along dimension `dim` for any of the previous syntaxes. For example, if `A` is a matrix, then `movprod(A,k,2)` operates across the columns of `A`, computing the `k`-element sliding product for each row.

`M = movprod( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `movprod(A,k,'includenan')` includes all

`NaN` values in the calculation, which is the default. `movprod(A,k,'omitnan')` ignores them and computes the product over fewer points.

`M = movprod( ___ ,Name,Value)` specifies additional parameters for the moving product using one or more name-value pair arguments. For example, if `x` is a vector of time values, then `movprod(A,k,'SamplePoints',x)` computes the moving product relative to the times in `x`.

Examples

Centered Moving Product of Vector

Compute the three-point centered moving product of a row vector. When there are fewer than three elements in the window at the endpoints, take the product over the elements that are available.

```
A = [1 2 4 -1 -2 -3 -1 3 2 1];
M = movprod(A,3)
```

```
M = 1×10

     2     8    -8     8    -6    -6     9    -6     6     2
```

Trailing Moving Product of Vector

Compute the three-point trailing moving product of a row vector. When there are fewer than three elements in the window at the endpoints, take the product over the elements that are available.

```
A = [1 2 1 -1 -2 -3 -1 3 4 1];
M = movprod(A,[2 0])
```

```
M = 1×10

     1     2     2    -2     2    -6    -6     9   -12    12
```

Moving Product of Matrix

Compute the 3-point centered moving product for each row of a matrix. The dimension argument is 2, which slides the window across the columns of A. The window starts on the first row, slides horizontally to the end of the row, then moves to the second row, and so on.

```
A = [1 2 1; -1 -2 -3; -1 3 4]
```

```
A = 3×3

     1     2     1
    -1    -2    -3
    -1     3     4
```

```
M = movprod(A,3,2)
```

```
M = 3×3

     2     2     2
     2    -6     6
    -3   -12    12
```

Moving Product of Vector with NaN Elements

Compute the three-point centered moving product of a row vector containing two NaN elements.

```
A = [2 1 NaN -1 -2 -3 NaN 3 4 1];
M = movprod(A,3)
```

```
M = 1×10

     2   NaN   NaN   NaN    -6   NaN   NaN   NaN    12     4
```

Recalculate the product, but omit the NaN values. When movprod discards NaN elements, it takes the product over the remaining elements in the window.

```
M = movprod(A,3,'omitnan')
```

```
M = 1×10

     2     2    -1     2    -6     6    -9    12    12     4
```

Sample Points for Moving Product

Compute a 3-hour centered moving product of the data in `A` according to the time vector `t`.

```
A = [4 8 6 -1 -2 -3];
k = hours(3);
t = datetime(2016,1,1,0,0,0) + hours(0:5)
```

```
t = 1x6 datetime array
Columns 1 through 3

   01-Jan-2016 00:00:00   01-Jan-2016 01:00:00   01-Jan-2016 02:00:00

Columns 4 through 6

   01-Jan-2016 03:00:00   01-Jan-2016 04:00:00   01-Jan-2016 05:00:00
```

```
M = movprod(A,k,'SamplePoints',t)
```

```
M = 1×6

    32   192   -48    12    -6     6
```

Return Only Full-Window Products

Compute the three-point centered moving product of a row vector, but discard any calculation that uses fewer than three points from the output. In other words, return only the products computed from a full three-element window, discarding endpoint calculations.

```
A = [1 2 1 -1 -2 -3 -1 3 4 1];
M = movprod(A,3,'Endpoints','discard')
```

```
M = 1×8

     2    -2     2    -6    -6     9   -12    12
```

Input Arguments

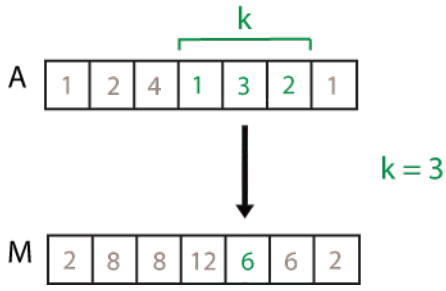
A — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

k — Window length
numeric or duration scalar

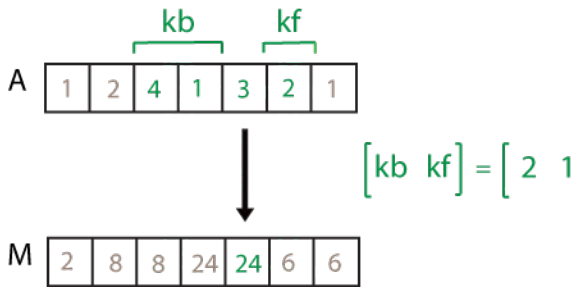
Window length, specified as a numeric or duration scalar. When `k` is a positive integer scalar, the centered product includes the element in the current position plus surrounding neighbors. For example, a three-point product defined by a window of length three results in the following calculation for a vector `A`:



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`[kb kf]` — Directional window length
numeric or duration row vector containing two elements

Directional window length, specified as a numeric or duration row vector containing two elements. When `kb` and `kf` are positive integer scalars, the calculation is over `kb+kf+1` elements. The calculation includes the element in the current position, `kb` elements before the current position, and `kf` elements after the current position. For example, a four-point MAD defined by the directional window `[2 1]` results in the following calculation for a vector `A`:



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

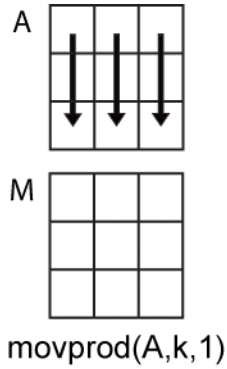
`dim` — Dimension to operate along

positive integer scalar

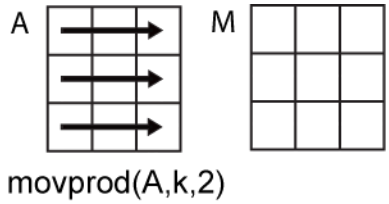
Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a matrix `A`.

- If `dim = 1`, then `movprod(A,k,1)` starts with the first column and slides vertically over each row. The product is computed over `k` elements at a time. Then it moves to the second column and repeats the computation. This process continues until all columns are exhausted.



- If `dim = 2`, then `movprod(A,k,2)` starts with the first row and slides horizontally across each column. The product is computed over `k` elements at a time. Then it moves to the second row and repeats the computation. This process continues until all rows are exhausted.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`nanflag` — `NaN` condition
`'includenan'` (default) | `'omitnan'`

`NaN` condition, specified as one of these values:

- `'includenan'` — Include `NaN` values from the input when computing the product, resulting in `NaN` output.
- `'omitnan'` — Ignore all `NaN` values in the input. If a window contains only `NaN` values, then `movprod` returns 1.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `M = movprod(A,k,'Endpoints','fill')`

`Endpoints` — Method to treat windows near endpoints

`'shrink'` (default) | `'discard'` | `'fill'` | numeric or logical scalar

Method to treat windows near endpoints, specified as the comma-separated pair consisting of `'Endpoints'` and one of the following:

`'Endpoints'` Value	Description
`'shrink'`	Shrink the window size near the endpoints of the input to include only existing elements.
`'discard'`	Do not output any products when the window does not completely overlap with existing elements.
`'fill'`	Replace nonexisting elements with `NaN`.
numeric or logical scalar	Replace nonexisting elements with the specified numeric or logical value.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string`

`SamplePoints` — Sample points for computing products

vector

Sample points for computing products, specified as the comma-separated pair consisting of `'SamplePoints'` and a vector. The sample points represent the locations of the data in `A`. Sample points do not need to be uniformly sampled. By default, the sample points vector is `[1 2 3 ... ]`.

Moving windows are defined relative to the sample points, which must be sorted and contain unique elements. For example, if `t` is a vector of times corresponding to the input data, then `movprod(rand(1,10),3,'SamplePoints',t)` has a window that represents the time interval between `t(i)-1.5` and `t(i)+1.5`.

When the sample points vector has data type `datetime` or `duration`, then the moving window length must have type `duration`.

If the sample points are nonuniformly spaced and the `'Endpoints'` name-value pair is specified, then its value must be `'shrink'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `datetime` | `duration`

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

The `'SamplePoints'` name-value pair is not supported.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This function supports C/C++ code generation with the limitations:

The `'SamplePoints'` name-value pair is not supported.

See Also

`cumprod` | `movsum` | `prod`

Introduced in R2017a

movstd

Moving standard deviation

Syntax

```
M = movstd(A,k)
M = movstd(A,[kb kf])
M = movstd( ___ ,w)
M = movstd( ___ ,w,dim)
M = movstd( ___ ,nanflag)
M = movstd( ___ ,Name,Value)
```

Description

`M = movstd(A,k)` returns an array of local `k`-point standard deviation on page 1-10010 values. Each standard deviation is calculated over a sliding window of length `k` across neighboring elements of `A`. When `k` is odd, the window is centered about the element in the current position. When `k` is even, the window is centered about the current and previous elements. The window size is automatically truncated at the endpoints when there are not enough elements to fill the window. When the window is truncated, the standard deviation is taken over only the elements that fill the window. `M` is the same size as `A`.

- If `A` is a vector, then `movstd` operates along the length of the vector.
- If `A` is a multidimensional array, then `movstd` operates along the first array dimension whose size does not equal 1.

`M = movstd(A,[kb kf])` computes the standard deviation with a window of length `kb +kf+1`. The calculation includes the element in the current position, `kb` elements backward, and `kf` elements forward.

`M = movstd( ___ ,w)` specifies a normalization factor for any of the previous syntaxes. When `w = 0` (default), `M` is normalized by `k-1` for window length `k`. When `w = 1`, `M` is normalized by `k`.

`M = movstd( ___ ,w,dim)` returns the array of sliding standard deviations along dimension `dim` for any of the previous syntaxes. Always specify the weight `w` from the previous syntax when specifying `dim`. For example, `movstd(A,k,0,2)` operates along the columns of a matrix `A`, computing the `k`-element sliding standard deviation for each row. The normalization factor is the default, `k-1`.

`M = movstd( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `movstd(A,k,'includenan')` includes all `NaN` values in the calculation while `movstd(A,k,'omitnan')` ignores them and computes the standard deviation over fewer points.

`M = movstd( ___ ,Name,Value)` specifies additional parameters for the standard deviation using one or more name-value pair arguments. For example, if `x` is a time vector, then `movstd(A,k,'SamplePoints',x)` computes the moving standard deviation relative to the times in `x`.

Examples

Centered Moving Standard Deviation of Vector

Compute the three-point centered moving standard deviation of a row vector. When there are fewer than three elements in the window at the endpoints, take the standard deviation over the elements that are available.

```
A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movstd(A,3)
```

```
M = 1×10

    2.8284    2.0000    4.7258    4.3589    1.0000    1.0000    3.0551    2.6458    1.0
```

Trailing Moving Standard Deviation of Vector

Compute the three-point trailing moving standard deviation of a row vector. When there are fewer than three elements in the window at the endpoints, take the standard deviation over the elements that are available.

```
A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movstd(A,[2 0])
```

```
M = 1×10

         0    2.8284    2.0000    4.7258    4.3589    1.0000    1.0000    3.0551    2.6
```

Specify Normalization for Moving Standard Deviation

Compute the three-point centered moving standard deviation of a row vector and normalize each standard deviation by the number of elements in the window.

```
A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movstd(A,3,1)
```

```
M = 1×10

    2.0000    1.6330    3.8586    3.5590    0.8165    0.8165    2.4944    2.1602    0.8
```

Moving Standard Deviation of Matrix

Compute the three-point centered moving standard deviation for each row of a matrix. The window starts on the first row, slides horizontally to the end of the row, then moves to the second row, and so on. The dimension argument is two, which slides the window across the columns of A. Always specify the normalization factor when specifying the dimension.

```
A = [4 8 6; -1 -2 -3; -1 3 4];
M = movstd(A,3,0,2)
```

```
M = 3×3

    2.8284    2.0000    1.4142
    0.7071    1.0000    0.7071
    2.8284    2.6458    0.7071
```

Moving Standard Deviation of Vector with NaN Elements

Compute the three-point centered moving standard deviation of a row vector containing two NaN elements.

```
A = [4 8 NaN -1 -2 -3 NaN 3 4 5];
M = movstd(A,3)
```

```
M = 1×10

    2.8284       NaN       NaN       NaN    1.0000       NaN       NaN       NaN    1.0
```

Recalculate the standard deviation, but omit the NaN values. When movstd discards NaN elements, it takes the standard deviation over the remaining elements in the window.

```
M = movstd(A,3,'omitnan')
```

```
M = 1×10

    2.8284    2.8284    6.3640    0.7071    1.0000    0.7071    4.2426    0.7071    1.0
```

Sample Points for Moving Standard Deviation

Compute a 3-hour centered moving standard deviation of the data in A according to the time vector t.

```
A = [4 8 6 -1 -2 -3];
k = hours(3);
t = datetime(2016,1,1,0,0,0) + hours(0:5)
```

```
t = 1x6 datetime array
Columns 1 through 3

   01-Jan-2016 00:00:00   01-Jan-2016 01:00:00   01-Jan-2016 02:00:00

Columns 4 through 6

   01-Jan-2016 03:00:00   01-Jan-2016 04:00:00   01-Jan-2016 05:00:00
```

```
M = movstd(A,k,'SamplePoints',t)
```

```
M = 1×6

    2.8284    2.0000    4.7258    4.3589    1.0000    0.7071
```

Return Only Full-Window Standard Deviations

Compute the three-point centered moving standard deviation of a row vector, but discard any calculation that uses fewer than three points from the output. In other words, return only the standard deviations computed from a full three-element window, discarding endpoint calculations.

```
A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movstd(A,3,'Endpoints','discard')
```

```
M = 1×8

    2.0000    4.7258    4.3589    1.0000    1.0000    3.0551    2.6458    1.0000
```

Input Arguments

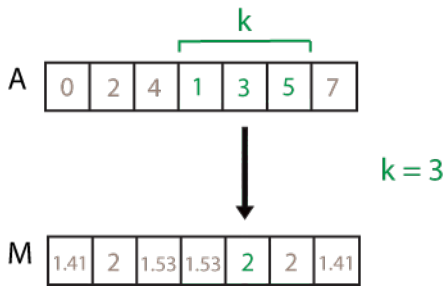
A — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `logical`

k — Window length
numeric or duration scalar

Window length, specified as a numeric or duration scalar. When `k` is a positive integer scalar, the centered standard deviation includes the element in the current position plus surrounding neighbors. For example, a three-point standard deviation defined by a window of length three results in the following calculation for a vector `A`:

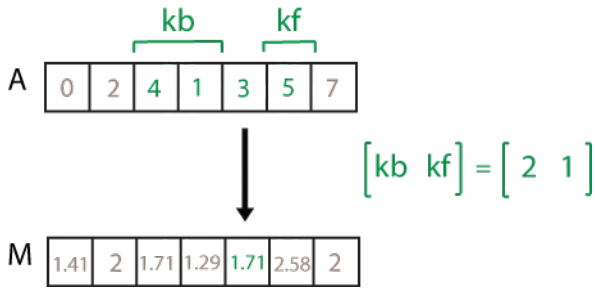


Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`[kb kf]` — Directional window length

numeric or duration row vector containing two elements

Directional window length, specified as a numeric or duration row vector containing two elements. When `kb` and `kf` are positive integer scalars, the calculation is over `kb+kf+1` elements. The calculation includes the element in the current position, `kb` elements before the current position, and `kf` elements after the current position. For example, a four-point standard deviation defined by the directional window `[2 1]` results in the following calculation for a vector `A`:



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`w` — Weight

`0` (default) | `1`

Weight, specified as one of these values:

- `0` — Normalize by `k-1`, where `k` is the window length. If `k=1`, the weight is `k`.
- `1` — Normalize by `k`.

Data Types: `single` | `double`

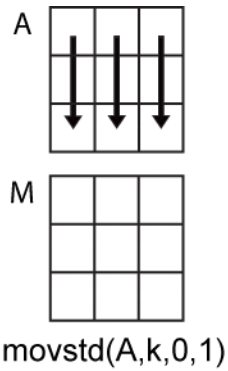
`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

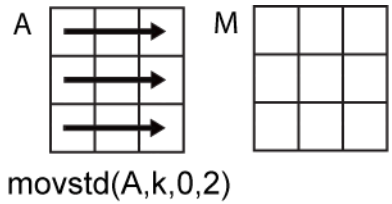
Dimension `dim` indicates the dimension that `movstd` operates along, that is, the direction in which the specified window slides.

Consider a two-dimensional input array, `A`.

- If `dim = 1`, then `movstd(A,k,0,1)` starts with the first column and slides vertically over each row. The standard deviation is taken over `k` elements at a time. Then it moves to the second column and repeats the computation. This process continues until all columns are exhausted. The argument `0` specifies the default weight, which is required when specifying `dim`.



- If `dim = 2`, then `movstd(A,k,0,2)` starts with the first row and slides horizontally across each column. The standard deviation is taken over `k` elements at a time. Then it moves to the second row and repeats the computation. This process continues until all rows are exhausted. The argument `0` specifies the default weight, which is required when specifying `dim`.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`nanflag` — `NaN` condition
`'includenan'` (default) | `'omitnan'`

`NaN` condition, specified as one of these values:

- `'includenan'` — Include `NaN` values from the input when computing the standard deviation, resulting in the output `NaN`.
- `'omitnan'` — Ignore all `NaN` values in the input. If a window contains only `NaN` values, then `movstd` returns `NaN`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `M = movstd(A,k,'Endpoints','fill')`

`Endpoints` — Method to treat leading and trailing windows
`'shrink'` (default) | `'discard'` | `'fill'` | numeric or logical scalar

Method to treat leading and trailing windows, specified as the comma-separated pair consisting of `'Endpoints'` and one of the following:

`'Endpoints'` Value	Description
`'shrink'`	Shrink the window size near the endpoints of the input to include only existing elements.

'Endpoints' Value	Description
'discard'	Do not output any standard deviations when the window does not completely overlap with existing elements.
'fill'	Substitute nonexisting elements with NaN.
numeric or logical scalar	Substitute nonexisting elements with a specified numeric or logical value.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char | string

SamplePoints — Sample points for computing standard deviations
vector

Sample points for computing standard deviations, specified as the comma-separated pair consisting of 'SamplePoints' and a vector. The sample points represent the location of the data in A. Sample points do not need to be uniformly sampled. By default, the sample points vector is [1 2 3 ...].

Moving windows are defined relative to the sample points, which must be sorted and contain unique elements. For example, if t is a vector of times corresponding to the input data, then movstd(rand(1,10),3,'SamplePoints',t) has a window that represents the time interval between t(i)-1.5 and t(i)+1.5.

When the sample points vector has data type datetime or duration, then the moving window length must have type duration.

If the sample points are nonuniformly spaced and the 'Endpoints' name-value pair is specified, then its value must be 'shrink'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | datetime | duration

More About

Standard Deviation

For a random variable vector A made up of N scalar observations, the standard deviation is defined as

$$S = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}|A_i - \mu|^2},$$

where μ is the mean of A:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} A_i .$$

The standard deviation is the square root of the variance. Some definitions of standard deviation use a normalization factor of *N* instead of *N-1*, which you can specify by setting `w` to `1`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

The `'SamplePoints'` name-value pair is not supported.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This function supports C/C++ code generation with the limitations:

The `'SamplePoints'` name-value pair is not supported.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `'SamplePoints'` name-value pair is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`movmad` | `movmean` | `movmedian` | `movvar` | `std`

Introduced in R2016a

movsum

Moving sum

Syntax

```
M = movsum(A,k)
M = movsum(A,[kb kf])
M = movsum( ___ ,dim)
M = movsum( ___ ,nanflag)
M = movsum( ___ ,Name,Value)
```

Description

`M = movsum(A,k)` returns an array of local `k`-point sums, where each sum is calculated over a sliding window of length `k` across neighboring elements of `A`. When `k` is odd, the window is centered about the element in the current position. When `k` is even, the window is centered about the current and previous elements. The window size is automatically truncated at the endpoints when there are not enough elements to fill the window. When the window is truncated, the sum is taken over only the elements that fill the window. `M` is the same size as `A`.

- If `A` is a vector, then `movsum` operates along the length of the vector.
- If `A` is a multidimensional array, then `movsum` operates along the first array dimension whose size does not equal 1.

`M = movsum(A,[kb kf])` computes the sum with a window of length `kb+kf+1` that includes the element in the current position, `kb` elements backward, and `kf` elements forward.

`M = movsum( ___ ,dim)` returns the array of sliding sums along dimension `dim` for any of the previous syntaxes. For example, if `A` is a matrix, then `movsum(A,k,2)` operates along the columns of `A`, computing the `k`-element sliding sum for each row.

`M = movsum( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `movsum(A,k,'includenan')` includes all

`NaN` values in the calculation while `movsum(A,k,'omitnan')` ignores them and computes the sum over fewer points.

`M = movsum( ___ ,Name,Value)` specifies additional parameters for the sum using one or more name-value pair arguments. For example, if `x` is a time vector, then `movsum(A,k,'SamplePoints',x)` computes the moving sum of `A` relative to the times in `x`.

Examples

Centered Moving Sum of Vector

Compute the three-point centered moving sum of a row vector. When there are fewer than three elements in the window at the endpoints, take the sum over the elements that are available.

```
A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movsum(A,3)
```

```
M = 1×10

    12    18    13     3    -6    -6    -1     6    12     9
```

Trailing Moving Sum of Vector

Compute the three-point trailing moving sum of a row vector. When there are fewer than three elements in the window at the endpoints, `movsum` takes the sum over the number of elements that are available.

```
A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movsum(A,[2 0])
```

```
M = 1×10

     4    12    18    13     3    -6    -6    -1     6    12
```

Moving Sum of Matrix

Compute the three-point centered moving sum for each row of a matrix. The window starts on the first row, slides horizontally to the end of the row, then moves to the second row, and so on. The dimension argument is two, which slides the window across the columns of `A`.

```
A = [4 8 6; -1 -2 -3; -1 3 4]
```

```
A = 3×3

     4     8     6
    -1    -2    -3
    -1     3     4
```

```
M = movsum(A,3,2)
```

```
M = 3×3

    12    18    14
    -3    -6    -5
     2     6     7
```

Moving Sum of Vector with `NaN` Elements

Compute the three-point centered moving sum of a row vector containing two `NaN` elements.

```
A = [4 8 NaN -1 -2 -3 NaN 3 4 5];
M = movsum(A,3)
```

```
M = 1×10

    12   NaN   NaN   NaN    -6   NaN   NaN   NaN    12     9
```

Recalculate the sum, but omit the `NaN` values. When `movsum` discards `NaN` elements, it takes the sum over the remaining elements in the window.

```
M = movsum(A,3,'omitnan')
```

```
M = 1×10

    12    12     7    -3    -6    -5     0     7    12     9
```

Sample Points for Moving Sum

Compute a 3-hour centered moving sum of the data in `A` according to the time vector `t`.

```
A = [4 8 6 -1 -2 -3];
k = hours(3);
t = datetime(16,1,1,0,0,0) + hours(0:5)
```

```
t = 1x6 datetime array
Columns 1 through 3

   01-Jan-0016 00:00:00   01-Jan-0016 01:00:00   01-Jan-0016 02:00:00

Columns 4 through 6

   01-Jan-0016 03:00:00   01-Jan-0016 04:00:00   01-Jan-0016 05:00:00
```

```
M = movsum(A,k,'SamplePoints',t)
```

```
M = 1×6

    12    18    13     3    -6    -5
```

Return Only Full-Window Sums

Compute the three-point centered moving sum of a row vector, but discard any calculation that uses fewer than three points from the output. In other words, return only the sums computed from a full three-element window, discarding endpoint calculations.

```
A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movsum(A,3,'Endpoints','discard')
```

```
M = 1×8

    18    13     3    -6    -6    -1     6    12
```

Input Arguments

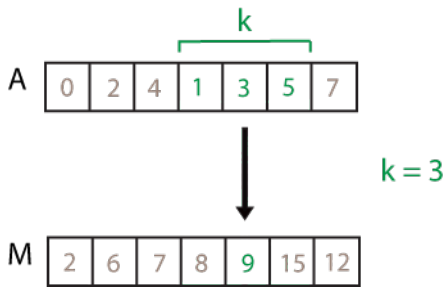
A — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

k — Window length
numeric or duration scalar

Window length, specified as a numeric or duration scalar. When `k` is a positive integer scalar, the centered sum includes the element in the current position plus surrounding neighbors. For example, a three-point sum defined by a window of length three results in the following calculation for a vector `A`:

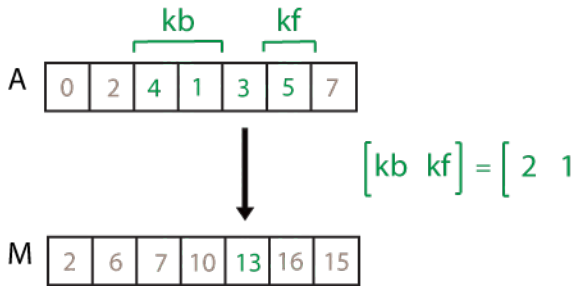


Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`[kb kf]` — Directional window length
numeric or duration row vector containing two elements

Directional window length, specified as a numeric or duration row vector containing two elements. When `kb` and `kf` are positive integer scalars, the calculation is over `kb+kf+1`

elements. The calculation includes the element in the current position, `kb` elements before the current position, and `kf` elements after the current position. For example, a four-point sum defined by the directional window `[2 1]` results in the following calculation for a vector `A`:



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`dim` — Dimension to operate along

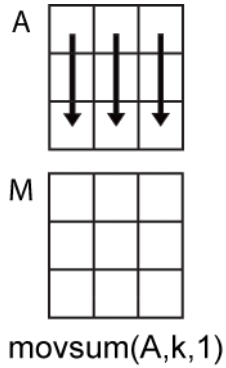
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

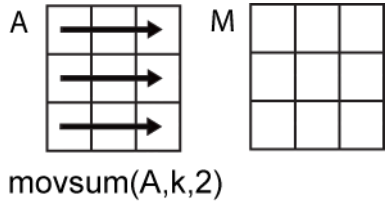
Dimension `dim` indicates the dimension that `movsum` operates along, that is, the direction in which the specified window slides.

Consider a two-dimensional input array, `A`.

- If `dim = 1`, then `movsum(A,k,1)` starts with the first column and slides vertically over each row. The sum is computed over `k` elements at a time. Then it moves to the second column and repeats the computation. This process continues until all columns are exhausted.



- If `dim = 2`, then `movsum(A,k,2)` starts with the first row and slides horizontally across each column. The sum is computed over `k` elements at a time. Then it moves to the second row and repeats the computation. This process continues until all rows are exhausted.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`nanflag` — `NaN` condition
`'includenan'` (default) | `'omitnan'`

`NaN` condition, specified as one of these values:

- `'includenan'` — Include `NaN` values when computing the sum, resulting in `NaN`.
- `'omitnan'` — Ignore all `NaN` values in the input. If a window contains only `NaN` values, then `movsum` returns 0.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `M = movsum(A,k,'Endpoints','fill')`

`Endpoints` — Method to treat leading and trailing windows
`'shrink'` (default) | `'discard'` | `'fill'` | numeric or logical scalar

Method to treat leading and trailing windows, specified as the comma-separated pair consisting of `'Endpoints'` and one of the following:

`'Endpoints'` Value	Description
`'shrink'`	Shrink the window size near the endpoints of the input to include only existing elements.
`'discard'`	Do not output any sums when the window does not completely overlap with existing elements.
`'fill'`	Substitute nonexisting elements with `NaN`.
numeric or logical scalar	Substitute nonexisting elements with a specified numeric or logical value.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string`

`SamplePoints` — Sample points for computing sums
vector

Sample points for computing sums, specified as the comma-separated pair consisting of `'SamplePoints'` and a vector. The sample points represent the location of the data in `A`. Sample points do not need to be uniformly sampled. By default, the sample points vector is `[1 2 3 ... ]`.

Moving windows are defined relative to the sample points, which must be sorted and contain unique elements. For example, if `t` is a vector of times corresponding to the input data, then `movsum(rand(1,10),3,'SamplePoints',t)` has a window that represents the time interval between `t(i)-1.5` and `t(i)+1.5`.

When the sample points vector has data type `datetime` or `duration`, then the moving window length must have type `duration`.

If the sample points are nonuniformly spaced and the `'Endpoints'` name-value pair is specified, then its value must be `'shrink'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `datetime` | `duration`

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

The `'SamplePoints'` name-value pair is not supported.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This function supports C/C++ code generation with the limitations:

The `'SamplePoints'` name-value pair is not supported.

Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `'SamplePoints'` name-value pair is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`cumsum` | `movmax` | `movmean` | `movmin` | `movprod` | `sum`

Introduced in R2016a

movvar

Moving variance

Syntax

```
M = movvar(A,k)
M = movvar(A,[kb kf])
M = movvar( ___ ,w)
M = movvar( ___ ,w,dim)
M = movvar( ___ ,nanflag)
M = movvar( ___ ,Name,Value)
```

Description

`M = movvar(A,k)` returns an array of local `k`-point variance on page 1-10031 values, where each variance is calculated over a sliding window of length `k` across neighboring elements of `A`. When `k` is odd, the window is centered about the element in the current position. When `k` is even, the window is centered about the current and previous elements. The window size is automatically truncated at the endpoints when there are not enough elements to fill the window. When the window is truncated, the variance is taken over only the elements that fill the window. `M` is the same size as `A`.

- If `A` is a vector, then `movvar` operates along the length of the vector.
- If `A` is a multidimensional array, then `movvar` operates along the first array dimension whose size does not equal 1.

`M = movvar(A,[kb kf])` computes the variance with a window of length `kb+kf+1` that includes the element in the current position, `kb` elements backward, and `kf` elements forward.

`M = movvar( ___ ,w)` specifies a normalization factor for any of the previous syntaxes. When `w = 0` (default), `M` is normalized by `k-1` for window length `k`. When `w = 1`, `M` is normalized by `k`.

`M = movvar( ___ ,w,dim)` returns the array of sliding variances along dimension `dim` for any of the previous syntaxes. Always specify the weight `w` from the previous syntax

when specifying `dim`. For example, if `A` is a matrix, then `movvar(A,k,0,2)` operates along the columns of `A`, computing the `k`-element sliding variance for each row. The normalization factor is the default, `k-1`.

`M = movvar( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `movvar(A,k,'includenan')` includes all `NaN` values in the calculation while `movvar(A,k,'omitnan')` ignores them and computes the variance over fewer points.

`M = movvar( ___ ,Name,Value)` specifies additional parameters for the variance using one or more name-value pair arguments. For example, if `x` is a vector of time values, then `movvar(A,k,'SamplePoints',x)` computes the moving variance relative to the times in `x`.

Examples

Centered Moving Variance of Vector

Compute the three-point centered moving variance of a row vector. When there are fewer than three elements in the window at the endpoints, take the variance over the elements that are available.

```
A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movvar(A,3)
```

```
M = 1×10

    8.0000    4.0000   22.3333   19.0000    1.0000    1.0000    9.3333    7.0000    1.0
```

Trailing Moving Variance of Vector

Compute the three-point trailing moving variance of a row vector. When there are fewer than three elements in the window at the endpoints, take the variance over the elements that are available.

```
A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movvar(A,[2 0])
```

```
M = 1×10

         0    8.0000    4.0000   22.3333   19.0000    1.0000    1.0000    9.3333    7.0
```

Specify Normalization for Moving Variance

Compute the three-point centered moving variance of a row vector and normalize each variance by the number of elements in the window.

```
A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movvar(A,3,1)
```

```
M = 1×10

    4.0000    2.6667   14.8889   12.6667    0.6667    0.6667    6.2222    4.6667    0.6
```

Moving Variance of Matrix

Compute the three-point centered moving variance for each row of a matrix. The window starts on the first row, slides horizontally to the end of the row, then moves to the second row, and so on. The dimension argument is two, which slides the window across the columns of `A`. Always specify the normalization factor when specifying the dimension.

```
A = [4 8 6; -1 -2 -3; -1 3 4];
M = movvar(A,3,0,2)
```

```
M = 3×3

    8.0000    4.0000    2.0000
    0.5000    1.0000    0.5000
    8.0000    7.0000    0.5000
```

Moving Variance of Vector with NaN Elements

Compute the three-point centered moving variance of a row vector containing two NaN elements.

```
A = [4 8 NaN -1 -2 -3 NaN 3 4 5];
M = movvar(A,3)
```

```
M = 1×10

    8.0000       NaN       NaN       NaN    1.0000       NaN       NaN       NaN    1.0
```

Recalculate the variance, but omit the NaN values. When movvar discards NaN elements, it takes the variance over the remaining elements in the window.

```
M = movvar(A,3,'omitnan')
```

```
M = 1×10

    8.0000    8.0000   40.5000    0.5000    1.0000    0.5000   18.0000    0.5000    1.0
```

Sample Points for Moving Variance

Compute a 3-hour centered moving variance of the data in A according to the time vector t.

```
A = [4 8 6 -1 -2 -3];
k = hours(3);
t = datetime(2016,1,1,0,0,0) + hours(0:5)
```

```
t = 1x6 datetime array
Columns 1 through 3

   01-Jan-2016 00:00:00   01-Jan-2016 01:00:00   01-Jan-2016 02:00:00

Columns 4 through 6

   01-Jan-2016 03:00:00   01-Jan-2016 04:00:00   01-Jan-2016 05:00:00
```

```
M = movvar(A,k,'SamplePoints',t)
```

```
M = 1×6

    8.0000    4.0000   22.3333   19.0000    1.0000    0.5000
```

Return Only Full-Window Variances

Compute the three-point centered moving variance of a row vector, but discard any calculation that uses fewer than three points from the output. In other words, return only the variances computed from a full three-element window, discarding endpoint calculations.

```
A = [4 8 6 -1 -2 -3 -1 3 4 5];
M = movvar(A,3,'Endpoints','discard')
```

```
M = 1×8

    4.0000   22.3333   19.0000    1.0000    1.0000    9.3333    7.0000    1.0000
```

Input Arguments

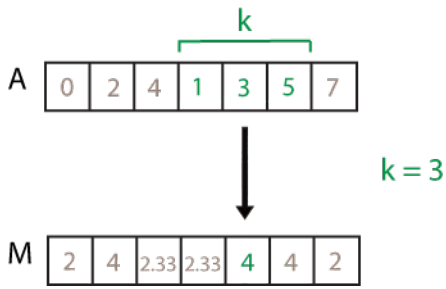
A — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `logical`

k — Window length
numeric or duration scalar

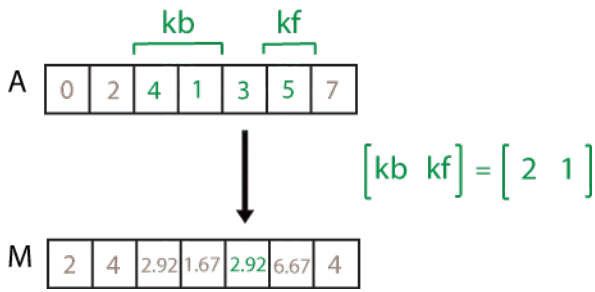
Window length, specified as a numeric or duration scalar. When `k` is a positive integer scalar, the centered variance includes the element in the current position plus surrounding neighbors. For example, a three-point variance defined by a window of length three results in the following calculation for a vector `A`:



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`[kb kf]` — Directional window length
numeric or duration row vector containing two elements

Directional window length, specified as a numeric or duration row vector containing two elements. When `kb` and `kf` are positive integer scalars, the calculation is over `kb+kf+1` elements. The calculation includes the element in the current position, `kb` elements before the current position, and `kf` elements after the current position. For example, a four-point variance defined by the directional window `[2 1]` results in the following calculation for a vector `A`:



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`w` — Weight
`0` (default) | `1`

Weight, specified as one of these values:

- `0` — Normalize by `k-1`, where `k` is the window length. If `k=1`, the weight is `k`.
- `1` — Normalize by `k`.

Data Types: `single` | `double`

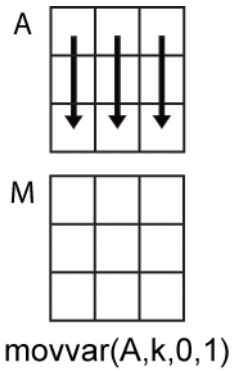
`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

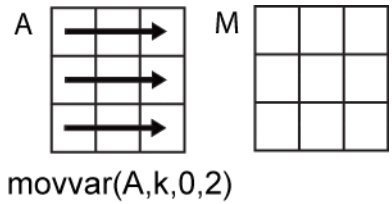
Dimension `dim` indicates the dimension that `movvar` operates along, that is, the direction in which the specified window slides.

Consider a two-dimensional input array, `A`.

- If `dim = 1`, then `movvar(A,k,0,1)` starts with the first column and slides vertically over each row. The variance is computed over `k` elements at a time. Then it moves to the second column and repeats the computation. This process continues until all columns are exhausted. The argument `0` specifies the default weight, which is required when specifying `dim`.



- If `dim = 2`, then `movvar(A,k,0,2)` starts with the first row and slides horizontally across each column. The variance is computed over `k` elements at a time. Then it moves to the second row and repeats the computation. This process continues until all rows are exhausted. The argument `0` specifies the default weight, which is required when specifying `dim`.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`nanflag` — `NaN` condition
`'includenan'` (default) | `'omitnan'`

`NaN` condition, specified as one of these values:

- `'includenan'` — Include `NaN` values from the input when computing the variance, resulting in the output `NaN`.
- `'omitnan'` — Ignore all `NaN` values in the input. If a window contains only `NaN` values, then `movvar` returns `NaN`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `M = movvar(A,k,'Endpoints','fill')`

`Endpoints` — Method to treat leading and trailing windows
`'shrink'` (default) | `'discard'` | `'fill'` | numeric or logical scalar

Method to treat leading and trailing windows, specified as the comma-separated pair consisting of `'Endpoints'` and one of the following:

`'Endpoints'` Value	Description
`'shrink'`	Shrink the window size near the endpoints of the input to include only existing elements.

'Endpoints' Value	Description
'discard'	Do not output any variances when the window does not completely overlap with existing elements.
'fill'	Substitute nonexisting elements with NaN.
numeric or logical scalar	Substitute nonexisting elements with a specified numeric or logical value.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char | string

SamplePoints — Sample points for computing variances
vector

Sample points for computing variances, specified as the comma-separated pair consisting of 'SamplePoints' and a vector. The sample points represent the location of the data in A. Sample points do not need to be uniformly sampled. By default, the sample points vector is [1 2 3 ...].

Moving windows are defined relative to the sample points, which must be sorted and contain unique elements. For example, if t is a vector of times corresponding to the input data, then movvar(rand(1,10),3,'SamplePoints',t) has a window that represents the time interval between t(i)-1.5 and t(i)+1.5.

When the sample points vector has data type datetime or duration, then the moving window length must have type duration.

If the sample points are nonuniformly spaced and the 'Endpoints' name-value pair is specified, then its value must be 'shrink'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | datetime | duration

More About

Variance

For a random variable vector A made up of N scalar observations, the variance is defined as

$$V = \frac{1}{N-1}\sum_{i=1}^{N}|A_i - \mu|^2$$

where μ is the mean of A,

$$\mu = \frac{1}{N}\sum_{i=1}^{N}A_i .$$

Some definitions of variance use a normalization factor of N instead of $N-1$. You can specify a factor of N by setting `w` to `1`. In either case, the mean is assumed to have the usual normalization factor N.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

The `'SamplePoints'` name-value pair is not supported.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This function supports C/C++ code generation with the limitations:

The `'SamplePoints'` name-value pair is not supported.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `'SamplePoints'` name-value pair is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`movmad` | `movmean` | `movmedian` | `movstd` | `var`

Introduced in R2016a

mpower, ^

Matrix power

Syntax

```
C = A^B
C = mpower(A,B)
```

Description

`C = A^B` computes `A` to the `B` power and returns the result in `C`.

`C = mpower(A,B)` is an alternate way to execute `A^B`, but is rarely used. It enables operator overloading for classes.

Examples

Square a Matrix

Create a 2-by-2 matrix and square it.

```
A = [1 2; 3 4];
C = A^2
```

```
C = 2×2

     7    10
    15    22
```

The syntax `A^2` is equivalent to `A*A`.

Matrix Exponents

Create a 2-by-2 matrix and use it as the exponent for a scalar.

```
B = [0 1; 1 0];
C = 2^B
```

```
C = 2×2

    1.2500    0.7500
    0.7500    1.2500
```

Compute `C` by first finding the eigenvalues `D` and eigenvectors `V` of the matrix `B`.

```
[V,D] = eig(B)
```

```
V = 2×2

   -0.7071    0.7071
    0.7071    0.7071


D = 2×2

    -1     0
     0     1
```

Next, use the formula `2^B = V*2^D/V` to compute the power.

```
C = V*2^D/V
```

```
C = 2×2

    1.2500    0.7500
    0.7500    1.2500
```

Input Arguments

A, B — Operands
scalar | matrix

Operands, specified as scalars or matrices. Inputs `A` and `B` must be one of the following combinations:

- Base `A` is a square matrix and exponent `B` is a scalar. If `B` is a positive integer, the power is computed by repeated squaring. For other values of `B` the calculation involves eigenvalues and eigenvectors.
- Base `A` is a scalar and exponent `B` is a square matrix. The calculation uses eigenvalues and eigenvectors.

Operands with an integer data type cannot be complex.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

Tips

- MATLAB computes `X^(-1)` and `inv(X)` in the same manner, and both are subject to the same limitations. For more information, see `inv`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If `A` is a 2-by-2 or larger matrix and `B` is `Inf` or `-Inf`, then `A^B` returns a matrix of `NaN` values.
- For `A^b`, if `b` is a noninteger scalar, then at least one of `A` or `b` must be complex.
- Code generation does not support sparse matrix inputs for this function.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- If `A` is a 2-by-2 or larger matrix and `B` is `Inf` or `-Inf`, then `A^B` returns a matrix of `NaN` values.
- For `A^b`, if `b` is a noninteger scalar, then at least one of `A` or `b` must be complex.
- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`mtimes` | `power` | `times`

Topics

“Array vs. Matrix Operations”
“Operator Precedence”
“MATLAB Operators and Special Characters”

Introduced before R2006a

mput

Upload file or folder to FTP server

Syntax

```
mput(ftpobj,contents)
uploadPaths = mput(ftpobj,contents)
```

Description

`mput(ftpobj,contents)` uploads the files or folders specified by `contents` to the current folder on the FTP server associated with `ftpobj`.

`uploadPaths = mput(ftpobj,contents)` returns a cell array that lists the paths to the uploaded files on the server.

Examples

Upload Files

Connect to an FTP server and upload files from your local machine to the server. This example shows a hypothetical FTP session on `ftp.example.com`, a machine that does not exist. If you have an account on an FTP server that grants you permission to upload files to that server, then you can use the `mput` function as shown in this example.

First, connect to the server.

```
ftpobj = ftp('ftp.example.com')
```

```
ftpobj = 

  FTP Object
     host: ftp.example.com
     user: anonymous
```

```
      dir: /
     mode: binary
```

Display the contents of the current folder on the FTP server.

```
dir(ftpobj)
```

```
myscript.m                  README.txt                pub
```

Upload a file from your local machine to the FTP server.

```
mput(ftpobj,'myNewScript.m');
```

Display the updated contents on the FTP server, including the new `myNewScript.m` file.

```
dir(ftpobj)
```

```
myscript.m                  myNewScript.m           README.txt
pub
```

Upload multiple files. If the file names have some text in common, then you can use a wildcard character (*) to match the rest of the names. In this example, use a wildcard to match the file names `image001.png`, `image002.png`, and `image003.png` on your local machine.

```
mput(ftpobj,'image*png');
```

Display the updated contents on the FTP server.

```
dir(ftpobj)
```

```
image001.png                image002.png            image003.png
myscript.m                  myNewScript.m           README.txt
pub
```

Input Arguments

`ftpobj` — Connection to FTP server
FTP object

Connection to an FTP server, specified as an FTP object.

`contents` — Local file or folder
character vector | string scalar

Local files or folders, specified as a character vector or string scalar.

To match multiple files or folders on your local machine, you can include a wildcard character (`*`) in `contents`. For example, if you specify `contents` as `*.docx`, then `mput` uploads all files whose names end with `.docx`.

See Also

`ftp` | `mget` | `mkdir` | `rename`

Introduced before R2006a

msgbox

Create message dialog box

Note **If you are using App Designer or creating apps with the `uifigure` function, then use `uialert` instead.** For more information, see “GUIDE Migration Strategies”.

Syntax

```
f = msgbox(message)
f = msgbox(message,title)
f = msgbox(message,title,icon)
f = msgbox(message,title,'custom',icondata,iconcmap)
f = msgbox( ___ ,createmode)
```

Description

`f = msgbox(message)` creates a message dialog box that automatically wraps `message` to fit an appropriately sized figure.

`f = msgbox(message,title)` specifies the title of the message box.

`f = msgbox(message,title,icon)` specifies a predefined icon to display in the message dialog box.

`f = msgbox(message,title,'custom',icondata,iconcmap)` specifies a custom icon to include in the message dialog box. `icondata` is the image data that defines the icon. `iconcmap` is the colormap used for the image. If `icondata` is a truecolor image array, you do not need to specify `iconcmap`.

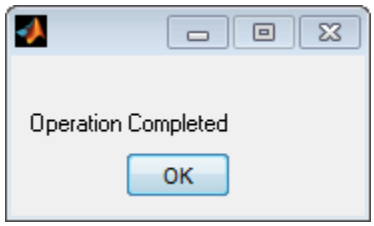
`f = msgbox( ___ ,createmode)` specifies the window mode for the dialog box. Alternatively, you can specify a structure array that specifies the window mode and the interpreter for the `message`.

Examples

Simple Message Dialog Box

Specify the text you want displayed in the message dialog box.

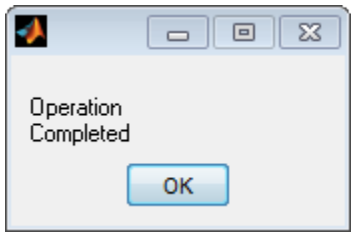
```
f = msgbox('Operation Completed');
```



Message Dialog Box Text with Line Breaks

Specify the message dialog box text using a cell array of character vectors.

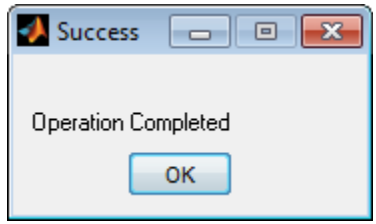
```
f = msgbox({'Operation';'Completed'});
```



Message Dialog Box with a Title

Specify the message dialog box text and give the dialog box a title, `Success`.

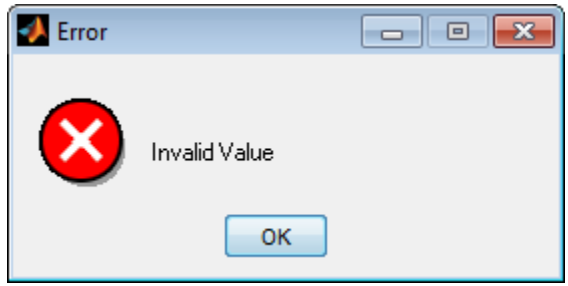
```
f = msgbox('Operation Completed','Success');
```



Message Dialog Box That Uses a Built-in Icon

Include a built-in error icon with an error message in a message dialog box entitled Error.

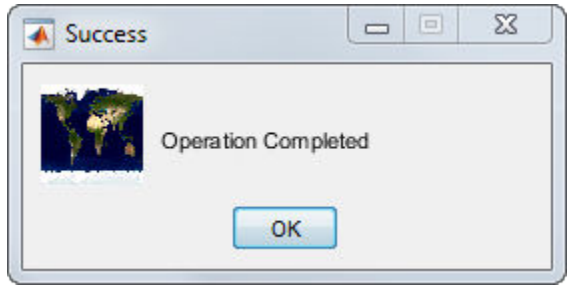
```
f = msgbox('Invalid Value', 'Error','error');
```



Message Dialog Box That Uses a Truecolor Custom Icon

Read an RGB image into the workspace. Then, specify it as a custom icon in the dialog box.

```
myicon = imread('landOcean.jpg');
h=msgbox('Operation Completed','Success','custom',myicon);
```



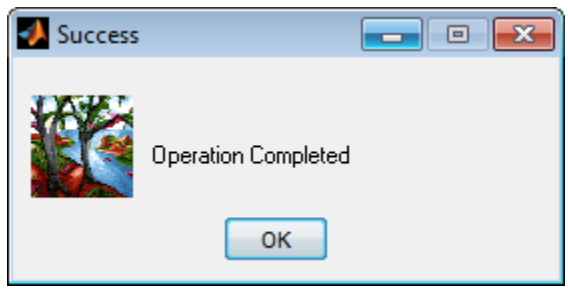
Message Dialog Box That Uses an Indexed Color Icon

Read the image data and colormap into the workspace.

```
[icondata,iconcmap] = imread('trees.tif');
```

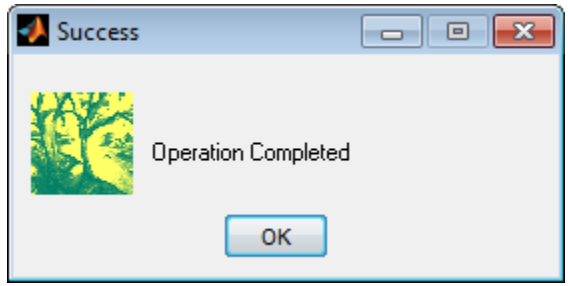
Create the message dialog box, including the custom icon.

```
h=msgbox('Operation Completed',...
         'Success','custom',icondata,iconcmap);
```



Adjust the image colors by specifying a different colormap. For instance, specify the MATLAB built-in colormap, `summer`

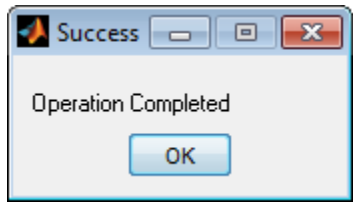
```
h=msgbox('Operation Completed','Success','custom',...
          icondata,summer);
```



Modal Message Dialog Box

Create a modal message dialog box, wrapping the call to `msgbox` with `uiwait` to make the message dialog box block MATLAB execution until the user responds to the message dialog box.

```
uiwait(msgbox('Operation Completed','Success','modal'));
```



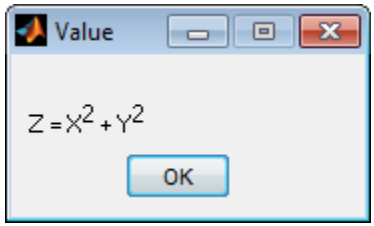
Modal Message Dialog Box That Uses a TeX Formatted Message

Create a structure to specify that the user must click OK before interacting with another window and that MATLAB interpret the message text as TeX format.

```
CreateStruct.Interpreter = 'tex';
CreateStruct.WindowStyle = 'modal';
```

Create the message dialog box.

```
h=msgbox('Z = X^2 + Y^2','Value',CreateStruct);
```



Input Arguments

`message` — Dialog box text
character vector | cell array of character vectors | string array

Dialog box text, specified as a character vector, cell array of character vectors, or string array.

Example: `'Operation Completed'`

Example: `{'Operation'; 'Completed'}`

`title` — Title
character vector | string scalar

Title, specified as a character vector or string scalar.

Example: `'Success'`

`icon` — Icon
`'none'` (default) | `'help'` | `'warn'` | `'error'` | `'custom'`

Icon, specified as `'help'`, `'warn'`, or `'error'` or `'none'`.

This table lists the values and the corresponding icons.

Value	Icon
`'help'`	
`'warn'`	

Value	Icon
`'error'`	
`'none'`	No icon displays.

`icondata` — Image array
m-by-n array | m-by-n-by-3 truecolor image array

Image array, specified as an m-by-n array or an m-by-n-by-3 truecolor image array. You can also use `imread` to get the image array from a file.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`iconcmap` — Colormap
three-column matrix of RGB triplets

Colormap, specified as a three-column matrix of RGB triplets. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of a color. The intensities must be in the range [0, 1].

Data Types: `single` | `double`

`createmode` — Window mode
`'non-modal'` (default) | `'modal'` | `'replace'` | structure array

Window mode, specified as `'non-modal'`, `'modal'`, `'replace'`, or a structure array.

- If `createmode` is `'non-modal'`, MATLAB creates a new nonmodal message box with the specified parameters. Existing message boxes with the same `title` remain.
- If `createmode` is `'modal'`, MATLAB replaces the existing message box with the specified `title` that was last created or clicked on with the specified modal dialog box. MATLAB deletes all other message boxes with the same title. The replaced message box can be either modal or nonmodal.
- If `createmode` is `'replace'`, MATLAB replaces the message box having the specified `title` that was last created or clicked on with a nonmodal message box as specified. MATLAB deletes all other message boxes with the same title. The replaced message box can be either modal or nonmodal.
- If `createmode` is a structure array, it must have the fields `WindowStyle` and `Interpreter`. The value of the `WindowStyle` field must be `'non-modal'`, `'modal'`,

or `'replace'`. The value of the `Interpreter` field must be `'tex'` or `'none'`. If the `Interpreter` value is `'tex'`, MATLAB interprets the `message` value as TeX. The default value for `Interpreter` is `'none'`.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text.

Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they modify only the next character or the characters within the curly braces. When you set the `Interpreter` property to `'tex'`, the supported modifiers are as follows.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{`*`specifier`*`}`	Font name — Replace *`specifier`* with the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{`*`specifier`*`}`	Font size —Replace *`specifier`* with a numeric scalar value in point units.	`'\fontsize{15} text'`

Modifier	Description	Example
`\color{specifier}`	Font color — Replace *`specifier`* with one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Replace *`specifier`* with a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	β	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ϵ	`\Delta`	Δ	`\spadesuit`	♠
`\zeta`	ζ	`\Theta`	Θ	`\leftrightarrow`	↔
`\eta`	η	`\Lambda`	Λ	`\leftarrow`	←
`\theta`	θ	`\Xi`	Ξ	`\Leftarrow`	⇐
`\vartheta`	ϑ	`\Pi`	Π	`\uparrow`	↑
`\iota`	ι	`\Sigma`	Σ	`\rightarrow`	→

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	μ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

Output Arguments

f — Figure object
`Figure` object

`Figure` object for the dialog box. Use `f` to query and modify the properties of the dialog box.

More About

modal dialog box

A modal dialog box prevents a user from interacting with other windows before responding to the modal dialog box.

Tips

- Program execution continues even when a modal dialog box is active. To block MATLAB program execution until the user responds to the modal dialog box, use the `uiwait` function.
- Modal dialogs (created using `errordlg`, `msgbox`, or `warndlg`) replace any existing dialogs created with these functions that also have the same name.

See Also

`errordlg` | `helpdlg` | `warndlg`

Introduced before R2006a

mtimes, *

Matrix multiplication

Syntax

```
C = A*B
C = mtimes(A,B)
```

Description

`C = A*B` is the matrix product of `A` and `B`. If `A` is an m-by-p and `B` is a p-by-n matrix, then `C` is an m-by-n matrix defined by

$$C(i,j) = \sum_{k=1}^{p} A(i,k)B(k,j).$$

This definition says that `C(i,j)` is the inner product of the `i`th row of `A` with the `j`th column of `B`. You can write this definition using the MATLAB colon operator as

```
C(i,j) = A(i,:)*B(:,j)
```

For nonscalar `A` and `B`, the number of columns of `A` must equal the number of rows of `B`. Matrix multiplication is *not* universally commutative for nonscalar inputs. That is, `A*B` is typically not equal to `B*A`. If at least one input is scalar, then `A*B` is equivalent to `A.*B` and is commutative.

`C = mtimes(A,B)` is an alternative way to execute `A*B`, but is rarely used. It enables operator overloading for classes.

Examples

Multiply Two Vectors

Create a 1-by-4 row vector, `A`, and a 4-by-1 column vector, `B`.

```
A = [1 1 0 0];
B = [1; 2; 3; 4];
```

Multiply A times B.

```
C = A*B
```

```
C = 3
```

The result is a 1-by-1 scalar, also called the *dot product* or *inner product* of the vectors A and B. Alternatively, you can calculate the dot product $A \cdot B$ with the syntax `dot(A,B)`.

Multiply B times A.

```
C = B*A
```

```
C = 4×4

     1     1     0     0
     2     2     0     0
     3     3     0     0
     4     4     0     0
```

The result is a 4-by-4 matrix, also called the *outer product* of the vectors A and B. The outer product of two vectors, $A \otimes B$, returns a matrix.

Multiply Two Arrays

Create two arrays, A and B.

```
A = [1 3 5; 2 4 7];
B = [-5 8 11; 3 9 21; 4 0 8];
```

Calculate the product of A and B.

```
C = A*B
```

```
C = 2×3

    24    35   114
    30    52   162
```

Calculate the inner product of the second row of A and the third column of B.

```
A(2,:)*B(:,3)
```

```
ans = 162
```

This answer is the same as C(2,3).

Input Arguments

A, B — Operands
scalars | vectors | matrices

Operands, specified as scalars, vectors, or matrices.

- A and B must be 2-D arrays, or one of them can be scalar.
- For nonscalar inputs, the number of columns in A must be equal to the number of rows in B.
- If one of A or B is an integer class (int16, uint8, ...), then the other input must be a scalar. Operands with an integer data type cannot be complex.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char | duration | calendarDuration

Output Arguments

C — Product
scalar | vector | matrix

Product, returned as a scalar, vector, or matrix. Array C has the same number of rows as input A and the same number of columns as input B. For example, if A is an m-by-0 empty matrix and B is a 0-by-n empty matrix, then A*B is an m-by-n matrix of zeros.

Tips

- With chained matrix multiplications such as A*B*C, you might be able to improve execution time by using parentheses to dictate the order of the operations. Consider

the case of multiplying three matrices with `A*B*C`, where `A` is 500-by-2, `B` is 2-by-500, and `C` is 500-by-2.

- With no parentheses, the order of operations is left to right so `A*B` is calculated first, which forms a 500-by-500 matrix. This matrix is then multiplied with `C` to arrive at the 500-by-2 result.
- If you instead specify `A*(B*C)`, then `B*C` is multiplied first, producing a 2-by-2 matrix. The small matrix then multiplies `A` to arrive at the same 500-by-2 result, but with fewer operations and less intermediate memory usage.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

- For `A*B` where `A` and `B` are both tall arrays, one of them must be a scalar.
- For `A'*B`, both `A` and `B` must be tall vectors or matrices with a common size in the first dimension.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Multiplication of pure imaginary numbers by non-finite numbers might not match MATLAB. The code generator does not specialize multiplication by pure imaginary numbers—it does not eliminate calculations with the zero real part. For example, `(Inf + 1i)*1i = (Inf*0 - 1*1) + (Inf*1 + 1*0)i = NaN + Infi`.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Multiplication of pure imaginary numbers by non-finite numbers might not match MATLAB. The code generator does not specialize multiplication by pure imaginary numbers—it does not eliminate calculations with the zero real part. For example, `(Inf + 1i)*1i = (Inf*0 - 1*1) + (Inf*1 + 1*0)i = NaN + Infi`.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- 64-bit integers are not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`colon` | `cross` | `dot` | `times`

Topics

“Array vs. Matrix Operations”
“Operator Precedence”
“MATLAB Operators and Special Characters”

Introduced before R2006a

mu2lin

Convert mu-law audio signal to linear

Syntax

```
y = mu2lin(mu)
```

Description

`y = mu2lin(mu)` converts mu-law encoded 8-bit audio signals, stored as “flints” in the range `0` ≤ `mu` ≤ `255`, to linear signal amplitude in the range `-s < Y < s` where `s = 32124/32768 ~= .9803`. The input `mu` is often obtained using `fread(...,'uchar')` to read byte-encoded audio files. "Flints" are MATLAB integers — floating-point numbers whose values are integers.

See Also

`audioread` | `lin2mu`

Introduced before R2006a

multibandread

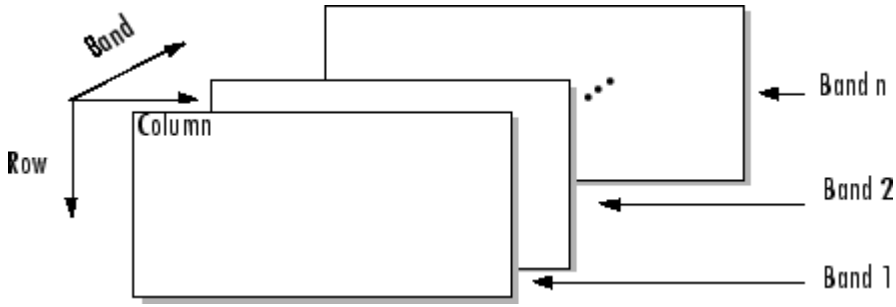
Read band-interleaved data from binary file

Syntax

```
X =
multibandread(filename,size,precision,offset,interleave,byteorder)
X = multibandread(...,subset1,subset2,subset3)
```

Description

`X = multibandread(filename,size,precision,offset,interleave,byteorder)` reads band-sequential (BSQ), band-interleaved-by-line (BIL), or band-interleaved-by-pixel (BIP) data from the binary file `filename`. The `filename` input is specified as a character vector or string scalar. This function defines *band* as the third dimension in a 3-D array, as shown in this figure.



You can use the parameters of `multibandread` to specify many aspects of the read operation, such as which bands to read. See “Parameters” on page 1-10060 for more information.

`X` is a 2-D array if only one band is read; otherwise it is 3-D. `X` is returned as an array of data type `double` by default. Use the `precision` parameter to map the data to a different data type.

`X = multibandread(...,subset1,subset2,subset3)` reads a subset of the data in the file. You can use up to three subsetting parameters to specify the data subset along row, column, and band dimensions. See “Subsetting Parameters” on page 1-10061 for more information.

Note In addition to BSQ, BIL, and BIP files, multiband imagery may be stored using the TIFF file format. In that case, use the `imread` function to import the data.

Parameters

This table describes the arguments accepted by `multibandread`.

Argument	Description
`filename`	Character vector or string scalar containing the name of the file to be read.
`size`	Three-element vector of integers consisting of `[height, width, N]`, where • `height` is the total number of rows • `width` is the total number of elements in each row • `N` is the total number of bands. This will be the dimensions of the data if it is read in its entirety.
`precision`	Character vector or string scalar specifying the format of the data to be read, such as `'uint8'`, `'double'`, `'integer*4'`, or any of the other precisions supported by the `fread` function. Note: You can also use the `precision` parameter to specify the format of the output data. For example, to read `uint8` data and output a `uint8` array, specify a precision of `'uint8=>uint8'` (or `'*uint8'`). To read `uint8` data and output it in the MATLAB software in single precision, specify `'uint8=>single'`. See `fread` for more information.
`offset`	Scalar specifying the zero-based location of the first data element in the file. This value represents the number of bytes from the beginning of the file to where the data begins.

Argument	Description
`interleave`	Format in which the data is stored, specified as one of these values: • `'bsq'` — Band-Sequential • `'bil'`— Band-Interleaved-by-Line • `'bip'`— Band-Interleaved-by-Pixel For more information about these interleave methods, see the `multibandwrite` reference page.
`byteorder`	Character vector or string scalar specifying the byte ordering (machine format) in which the data is stored, such as • `'ieee-le'` — Little-endian • `'ieee-be'` — Big-endian See `fopen` for a complete list of supported formats.

Subsetting Parameters

You can specify up to three subsetting parameters. Each subsetting parameter is a three-element cell array, `{`*`dim`*`,`*`method`*`,index}`, where

Parameter	Description
`dim`	The dimension to subset along. Specified as any of these values: • `'Column'` • `'Row'` • `'Band'`
`method`	The subsetting method. Specified as either of these values: • `'Direct'` • `'Range'` If you leave out this element of the subset cell array, `multibandread` uses `'Direct'` as the default.

Parameter	Description
`index`	If `method` is `'Direct'`, `index` is a vector specifying the indices to read along the Band dimension. If `method` is `'Range'`, `index` is a three-element vector of `[start, increment, stop]` specifying the range and step size to read along the dimension specified in `dim`. If `index` is a two-element vector, `multibandread` assumes that the value of `increment` is `1`.

Examples

Example 1

Setup initial parameters for a data set.

```
rows=3; cols=3; bands=5;
filename = tempname;
```

Define the data set.

```
fid = fopen(filename, 'w', 'ieee-le');
fwrite(fid, 1:rows*cols*bands, 'double');
fclose(fid);
```

Read every other band of the data using the Band-Sequential format.

```
im1 = multibandread(filename, [rows cols bands], ...
                    'double', 0, 'bsq', 'ieee-le', ...
                    {'Band', 'Range', [1 2 bands]} )
```

Read the first two rows and columns of data using Band-Interleaved-by-Pixel format.

```
im2 = multibandread(filename, [rows cols bands], ...
                    'double', 0, 'bip', 'ieee-le', ...
                    {'Row', 'Range', [1 2]}, ...
                    {'Column', 'Range', [1 2]} )
```

Read the data using Band-Interleaved-by-Line format.

```
im3 = multibandread(filename, [rows cols bands], ...
                    'double', 0, 'bil', 'ieee-le')
```

Delete the file created in this example.

```
delete(filename);
```

Example 2

Read `int16` BIL data from the FITS file `tst0012.fits`, starting at byte 74880.

```
im4 = multibandread('tst0012.fits', [31 73 5], ...
                    'int16', 74880, 'bil', 'ieee-be', ...
                    {'Band', 'Range', [1 3]} );
im5 = double(im4)/max(max(max(im4)));
imagesc(im5);
```

See Also

`fread` | `fwrite` | `imread` | `memmapfile` | `multibandwrite`

Introduced before R2006a

multibandwrite

Write band-interleaved data to file

Syntax

```
multibandwrite(data,filename,interleave)
multibandwrite(data,filename,interleave,start,totalsize)
multibandwrite(...,param,value...)
```

Description

`multibandwrite(data,filename,interleave)` writes `data`, a two- or three-dimensional numeric or logical array, to the binary file specified by `filename`. The `filename` is specified as a character vector or string scalar. The length of the third dimension of `data` determines the number of bands written to the file. The bands are written to the file in the form specified by `interleave`. See “Interleave Methods” on page 1-10066 for more information about this argument.

If filename already exists, `multibandwrite` overwrites it unless you specify the optional `offset` parameter. For information about other optional parameters, see the last syntax and its description.

`multibandwrite(data,filename,interleave,start,totalsize)` writes `data` to the binary file `filename` in chunks. In this syntax, `data` is a subset of the complete data set.

`start` is a 1-by-3 array `[firstrow firstcolumn firstband]` that specifies the location to start writing data. `firstrow` and `firstcolumn` specify the location of the upper left image pixel. `firstband` gives the index of the first band to write. For example, `data(I,J,K)` contains the data for the pixel at `[firstrow+I-1, firstcolumn+J-1]` in the `(firstband+K-1)`-th band.

`totalsize` is a 1-by-3 array, `[totalrows,totalcolumns,totalbands]`, which specifies the full, three-dimensional size of the data to be written to the file.

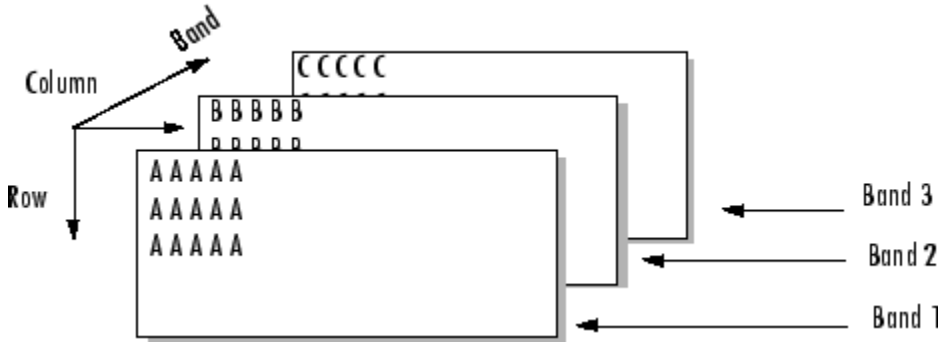
Note In this syntax, you must call `multibandwrite` multiple times to write all the data to the file. The first time it is called, `multibandwrite` writes the complete file, using the fill value for all values outside the data subset. In each subsequent call, `multibandwrite` overwrites these fill values with the data subset in `data`. The parameters `filename`, `interleave`, `offset`, and `totalsize` must remain constant throughout the writing of the file.

`multibandwrite(...,param,value...)` writes the multiband data to a file, specifying any of these optional parameter/value pairs.

Parameter	Description
`'precision'`	Character vector or string scalar specifying the form and size of each element written to the file. See the help for `fwrite` for a list of valid values. The default precision is the class of the data.
`'offset'`	The number of bytes to skip before the first data element. If the file does not already exist, `multibandwrite` writes ASCII null values to fill the space. To specify a different fill value, use the parameter `'fillvalue'`. This option is useful when you are writing a header to the file before or after writing the data. When writing the header to the file after the data is written, open the file with `fopen` using `'r+'` permission.
`'machfmt'`	Character vector or string scalar to control the format in which the data is written to the file. Typical values are `'ieee-le'` for little endian and `'ieee-be'` for big endian. See the help for `fopen` for a complete list of available formats. The default machine format is the local machine format.
`'fillvalue'`	A number specifying the value to use in place of missing data. `'fillvalue'` can be a single number, specifying the fill value for all missing data, or a 1-by-Number-of-bands vector of numbers specifying the fill value for each band. This value is used to fill space when data is written in chunks.

Interleave Methods

`interleave` is a character vector or string scalar that specifies how `multibandwrite` interleaves the bands as it writes data to the file. If `data` is two-dimensional, `multibandwrite` ignores the `interleave` argument. The following table lists the supported methods and uses this example multiband file to illustrate each method.



Supported methods of interleaving bands include those listed below.

Method	Specified as	Description	Example
Band-Interleaved-by-Line	`'bil'`	Write an entire row from each band	`AAAAABBBBBCCCCC` `AAAAABBBBBCCCCC` `AAAAABBBBBCCCCC`
Band-Interleaved-by-Pixel	`'bip'`	Write a pixel from each band	`ABCABCABCABCABC...`

Band-Sequential	`'bsq'`	Write each band in its entirety	`AAAAA`
			`AAAAA`
			`AAAAA`
			`BBBBB`
			`BBBBB`
			`BBBBB`
			`CCCCC`
			`CCCCC`
			`CCCCC`

Examples

Note To run these examples successfully, you must be in a writable folder.

Example 1

Write all data (interleaved by line) to the file in one call.

```
data = reshape(uint16(1:600), [10 20 3]);
multibandwrite(data,'data.bil','bil');
```

Example 2

Write a single-band tiled image with one call for each tile. This is only useful if a subset of each band is available at each call to `multibandwrite`.

```
numBands = 1;
dataDims = [1024 1024 numBands];
data = reshape(uint32(1:(1024 * 1024 * numBands)), dataDims);

for band = 1:numBands
   for row = 1:2
      for col = 1:2

         subsetRows = ((row - 1) * 512 + 1):(row * 512);
         subsetCols = ((col - 1) * 512 + 1):(col * 512);

         upperLeft = [subsetRows(1), subsetCols(1), band];
         multibandwrite(data(subsetRows, subsetCols, band), ...
                        'banddata.bsq', 'bsq', upperLeft, dataDims);

      end
   end
end
```

See Also

`fread` | `fwrite` | `multibandread`

Introduced before R2006a

munlock

Allow clearing function or script from memory

Syntax

```
munlock(fun)
munlock
```

Description

`munlock(fun)` unlocks the MATLAB code file named `fun`. A function that is locked using `mlock` cannot be removed from memory with the `clear` command unless you first unlock it using `munlock`.

`munlock` unlocks the currently running file. Use this syntax only within a MATLAB code file.

Examples

Unlock Function in Memory

Create the function `lockFun` in your current working folder.

```
function lockFun()
    mlock
end
```

At the command prompt, call the `lockFun` function. Check that the function is locked.

```
lockFun
tf = mislocked('lockFun')
```

```
tf =

  logical
```

```
   1
```

Unlock the function so it can be cleared from memory. Check that the function is not locked.

```
munlock('lockFun')
tf = mislocked('lockFun')
```

```
tf =

  logical

   0
```

Input Arguments

`fun` — Name of function or script
character vector | string scalar

Name of function or script, specified as a character vector or a string scalar.

Tips

- To determine if a file is locked, use the `mislocked` function.
- To unlock a MEX file, use the `mexUnlock` function.

See Also

`inmem` | `mislocked` | `mlock` | `persistent`

Introduced before R2006a

mustBePositive

Validate that value is positive or issue error

Syntax

```
mustBePositive(value)
```

Description

`mustBePositive(value)` issues an error if `value` is not positive. Values are positive when they are real, numeric, and greater than zero. This function does not return a value.

`mustBePositive` accepts user-defined objects if the class of the object implements these methods:

- `gt`
- `isreal`
- `isnumeric` or `islogical`

This function ignores input arguments that are empty values. Therefore, no error is thrown when the property or function argument value is empty.

Examples

Validate That Values Are Positive

Use `mustBePositive` to validate that the input contains only positive values.

The `rand` function creates a uniformly distributed random number.

```
A = rand(1,5) -0.5;
```

Validate that array elements are positive.

```
mustBePositive(A)
```

```
Error using mustBePositive (line 14)
Value must be positive.
```

The result of subtracting `0.5` from the array return by `rand` can contain negative numbers. When a value is negative, `mustBePositive` issues an error.

Restrict Property Values

This class restricts the value of `Prop1` to positive values.

```
classdef MyClass
   properties
      Prop1 {mustBePositive}
   end
end
```

Create an object and assign a value to its property.

```
obj = MyClass;
obj.Prop1 = 0;
```

```
Error setting property 'Prop1' of class 'MyClass':
Value must be positive.
```

When you assign a value to the property, MATLAB calls `mustBePositive` with the value being assigned to the property. `mustBePositive` issues an error because the value `0` is not positive.

Restrict Function Argument Values

This function declares two input arguments. Input `A` must be a numeric vector. Input `ix` must be a positive integer.

```
function r = mbPositive(A,ix)
    arguments
        A (1,:) {mustBeNumeric}
        ix {mustBePositive, mustBeInteger}
    end
```

```
    r = A(ix);
end
```

Calling the function with a value for `ix` that does not meet the requirement of `mustBePositive` results in an error.

```
A = 1:10;
ix = 0;
r = mbPositive(A,ix);
```

```
Error using mbPositive
Invalid input argument at position 2. Value must be positive.
```

Input Arguments

`value` — Value to validate
scalar | array

Value to validate, specified as a scalar or an array of one of the following:

- `logical` or numeric class
- MATLAB classes that implement `gt`, `isreal`, and either `isnumeric` or `islogical`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`arguments` | `mustBeNonnegative` | `mustBeNonzero`

Topics
"Property Validation Functions"
"Function Argument Validation"

Introduced in R2017a

mustBeNonpositive

Validate that value is nonpositive or issue error

Syntax

```
mustBeNonpositive(value)
```

Description

`mustBeNonpositive(value)` issues an error if `value` is positive. Values are positive when they are greater than zero. This function does not return a value.

`mustBeNonpositive` accepts user-defined objects if the class of the object implements these methods:

- `le`
- `isreal`
- `isnumeric` or `islogical`

Examples

Validate That Values Are Not Positive

Use `mustBeNonpositive` to validate that the input contains only nonpositive values.

```
A = 1 < 10;
mustBeNonpositive(A)
```

```
Error using mustBeNonpositive (line 14)
Value must not be positive.
```

Because the expression `1 < 10` returns logical `1`, the value of `A` is positive and causes `mustBeNonpositive` to issue an error.

Restrict Property Values

This class restricts the value of `Prop1` to a nonpositive values.

```
classdef MyClass
   properties
      Prop1 {mustBeNonpositive}
   end
end
```

Create an object and assign a value to its property.

```
obj = MyClass;
obj.Prop1 = 10;
```

```
Error setting property 'Prop1' of class 'MyClass':
Value must not be positive.
```

When you assign a value to the property, MATLAB calls `mustBeNonpositive` with the value being assigned to the property. `mustBeNonpositive` issues an error because the value `10` is positive.

Restrict Function Argument Values

This function declares two input arguments. Input `lower` must not be positive and input `upper` must be positive.

```
function r = mbNonpositive(lower,upper)
    arguments
        lower {mustBeNonpositive}
        upper {mustBePositive}
    end
    x = lower*pi:upper*pi;
    r = sin(x);
end
```

Calling the function with a value for `lower` that does not meet the requirements of `mustBeNonpositive` results an error.

```
r = mbNonpositive(2,4);
```

```
Error using mbNonpositive
Invalid input argument at position 1. Value must not be positive.
```

Input Arguments

`value` — Value to validate
scalar | array

Value to validate, specified as a scalar or an array of one of the following:

- `logical` or numeric class
- MATLAB classes that implement `le`, `isreal`, and either `isnumeric` or `islogical`

Example: `value = -1` does not generate an error.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`arguments` | `mustBeNegative` | `mustBeNonzero`

Topics
"Property Validation Functions"
"Function Argument Validation"

Introduced in R2017a

mustBeFinite

Validate that value is finite or issue error

Syntax

```
mustBeFinite(value)
```

Description

`mustBeFinite(value)` issues an error if `value` is not finite. A numeric value is finite if it is not `NaN` or `Inf`. This function does not return a value.

This function accepts user-defined objects if the class of the object implements the following method:

- `isfinite`

This function ignores input arguments that are empty values. Therefore, no error is thrown when the property or function argument value is empty.

Examples

Validate That Values Are Not NaN or Inf

Use `mustBeFinite` to validate that no array elements are `NaN` or `Inf`.

```
d = 0:9;
A = 1./d;
mustBeFinite(A)
```

```
Error using mustBeFinite (line 13)
Value must be finite.
```

The division by `d` resulted in one element becoming `Inf`, which causes an error.

Restrict Property to Finite Values

This class restricts the value of `Prop1` to finite values.

```
classdef MyClass
   properties
      Prop1 {mustBeFinite}
   end
end
```

Create an object and assign a value to its property.

```
d = 0:9;
obj = MyClass;
obj.Prop1 = 1./d;
```

```
Error setting property 'Prop1' of class 'MyClass':
Value must be finite.
```

When you assign a value to the property, MATLAB calls `mustBeFinite` with the value being assigned to the property. `mustBeFinite` issues an error because the result of division by `0` is `Inf`.

Restrict Function Argument to Finite Values

This function declares an input argument that must be a vector of doubles that contain no `Inf` or `NaN` elements.

```
function s = mbFinite(x)
    arguments
        x (1,:) double {mustBeFinite}
    end
    n = length(x);
    m = sum(x)/n;
    s = sqrt(sum((x-m).^2/n));
end
```

Calling the function with an input that does not meet the requirement of `mustBeFinite` results in an error.

```
values = [12.7, 45.4, 98.9, Inf, 53.1];
s = mbFinite(values);
```

```
Error using mbFinite
Invalid input argument at position 1. Value must be finite.
```

Input Arguments

value — Value to validate
scalar | array

Value to validate, specified as a scalar or an array of one of the following:

- `logical` or numeric class
- MATLAB classes that implement `isfinite`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`arguments` | `mustBeNonNan` | `mustBeReal`

Topics

"Property Validation Functions"
"Function Argument Validation"

Introduced in R2017a

mustBeNonNan

Validate that value is nonNaN

Syntax

```
mustBeNonNan(value)
```

Description

`mustBeNonNan(value)` issues an error if `value` is `NaN`. This function does not return a value.

`mustBeNonNan` accepts user-defined objects if the class of the object implements this method:

- `isnan`

Examples

Validate That Values Are Not NaN

Use `mustBeNonNan` to validate that no array elements are `NaN`.

```
A = 0./[-2 -1 0 1 2];
mustBeNonNan(A)
```

```
Error using mustBeNonNan (line 13)
Value must not be NaN.
```

Division of `0` by `0` is equal to `NaN` so the array value contains one element that is `NaN`, which causes an error.

Restrict Property Values

This class restricts the value of `Prop1` to nonNaN values.

```
classdef MyClass
   properties
      Prop1 {mustBeNonNan}
   end
end
```

Create an object and assign a value to `Prop1`.

```
obj = MyClass;
obj.Prop1 = 0./[-2 -1 0 1 2];
```

```
Error setting property 'Prop1' of class 'MyClass':
Value must not be NaN.
```

When you assign a value to the property, MATLAB calls `mustBeNonNan` with the value being assigned to the property. `mustBeNonNan` issues an error because division of `0` by `0` is `NaN`.

Restrict Function Argument Values

This function declares an input argument that must be a vector of doubles containing no `NaN` elements.

```
function s = mbNonNan(x)
    arguments
        x (1,:) double {mustBeNonNan}
    end
    n = length(x);
    m = sum(x)/n;
    s = sqrt(sum((x-m).^2/n));
end
```

Calling the function with an input that does not meet the requirement of `mustBeNonNan` results in an error.

```
values = [12.7, 45.4, 98.9, NaN, 53.1];
s = mbNonNan(values);
```

```
Error using mbNonNan
Invalid input argument at position 1. Value must not be NaN.
```

Input Arguments

`value` — Value to validate
scalar | array

Value to validate, specified as a scalar or an array of one of the following:

- All MATLAB numeric classes and `logical`.
- MATLAB user-defined classes that implement `isnan`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`arguments` | `mustBeFinite` | `mustBeNonempty`

Topics

"Property Validation Functions"
"Function Argument Validation"

Introduced in R2017a

mustBeNonnegative

Validate that value is nonnegative or issue error

Syntax

```
mustBeNonnegative(value)
```

Description

`mustBeNonnegative(value)` issues an error if `value` is negative. This function does not return a value.

`mustBeNonnegative` accepts user-defined objects if the class of the object implements these methods:

- `ge`
- `isreal`
- `isnumeric` or `islogical`

Examples

Validate That Values Are Not Negative

Use `mustBeNonnegative` to validate that the input contains only nonnegative values.

The `randn` function creates normally distributed random numbers.

```
A = randn(1,5);
```

Validate that the random numbers are nonnegative.

```
mustBeNonnegative(A)
```

```
Error using mustBeNonnegative (line 14)
Value must be nonnegative.
```

Restrict Property Values

This class restricts the value of `Prop1` to nonnegative values.

```
classdef MyClass
   properties
      Prop1 {mustBeNonnegative}
   end
end
```

Create an object and assign a value to its property.

```
obj = MyClass;
obj.Prop1 = -10;
```

```
Error using mustBeNonnegative (line 14)
Value must be nonnegative.
```

When you assign a value to the property, MATLAB calls `mustBeNonnegative` with the value being assigned to the property. `mustBeNonnegative` issues an error because the value `-10` is negative.

Restrict Function Argument Values

This function declares two input arguments. Input `lower` must be not be positive and input `upper` must be positive.

```
function r = mbNonnegative(lower,upper)
    arguments
        lower {mustBeNonpositive}
        upper {mustBeNonnegative}
    end
    x = lower*pi:upper*pi;
    r = sin(x);
end
```

Calling the function with a value for `upper` that does not meet the requirements of `mustBeNonnegative` results an error.

```
r = mbNonnegative(-12,-4);
```

```
Error using mbNonnegative
Invalid input argument at position 2. Value must be nonnegative.
```

Input Arguments

`value` — Value to validate
scalar | array

Value to validate, specified as a scalar or an array of one of the following:

- `logical` or numeric class
- MATLAB classes that implement `ge`, `isreal`, and either `isnumeric` or `islogical`

Example: `value = 1` does not generate an error.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`arguments` | `mustBeNonzero` | `mustBePositive`

Topics
"Property Validation Functions"
"Function Argument Validation"

Introduced in R2017a

mustBeGreaterThan

Validate that value is greater than another value or issue error

Syntax

```
mustBeGreaterThan(value,c)
```

Description

`mustBeGreaterThan(value,c)` issues an error if any elements in `value` are not greater than the scalar `c`. This function does not return a value.

`mustBeGreaterThan` accepts user-defined objects if the class of the object implements these methods:

- `gt`
- `isreal`
- `isnumeric` or `islogical`

This function ignores empty values in the first input argument. Therefore, no error is thrown when the property or function argument value is empty.

Examples

Validate First Input Greater Than Second

Use `mustBeGreaterThan` to validate that the values in the first input are greater than the value of the second input.

```
mustBeGreaterThan([2 3 4],2)
```

```
Error using mustBeGreaterThan (line 19)
Value must be greater than 2.
```

Restrict Property Values

This class restricts the value of `Prop1` to be greater than 2.

```
classdef MyClass
   properties
      Prop1 {mustBeGreaterThan(Prop1,2)}
   end
end
```

Create an object and assign a value to its property.

```
obj = MyClass;
Obj.Prop1 = 2;
```

```
Error setting property 'Prop1' of class 'MyClass':
Value must be greater than 2.
```

When you assign a value to the property, MATLAB calls `mustBeGreaterThan` with the value being assigned to the property. `mustBeGreaterThan` issues an error because the value 2 is not greater than 2.

Restrict Function Argument Values

This function restricts the input argument to be values that are greater than 5.

```
function r = mbGreaterThan(x)
    arguments
        x {mustBeGreaterThan(x,5)}
    end
    r = x - 5;
end
```

Calling the function with a vector that contains values that are less than or equal to 5 does not meet the requirements defined with `mustBeGreaterThan` and results in an error.

```
x = [12.7, 45.4, 5.0, 77.1, 53.1];
r = mbGreaterThan(x);
```

```
Error using mbGreaterThan
Invalid input argument at position 1. Value must be greater than 5.
```

Input Arguments

`value` — Value to validate
scalar | array

Value to validate, specified as a scalar or an array of one of the following:

- `logical`, `char`, `string`, or numeric class
- MATLAB classes that implement `gt`, `isreal`, and either `isnumeric`, or `islogical`

`c` — Constant value that the `value` argument must be greater than
scalar

Constant value that the `value` argument must be greater than, specified as a scalar of one of the following:

- `logical`, `char`, `string`, and numeric class
- MATLAB classes that implement `gt`, `isreal`, and either `isnumeric`, or `islogical`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`arguments` | `mustBeGreaterThanOrEqual` | `mustBeLessThan`

Topics

"Property Validation Functions"

“Function Argument Validation”

Introduced in R2017a

mustBeGreaterThanOrEqual

Validate that value is greater than or equal to another value or issue error

Syntax

```
mustBeGreaterThanOrEqual(value,c)
```

Description

`mustBeGreaterThanOrEqual(value,c)` issues an error if any elements in `value` are not greater than or equal to the scalar `c`. This function does not return a value.

`mustBeGreaterThanOrEqual` accepts user-defined objects if the class of the object implements these methods:

- `ge`
- `isreal`
- `isnumeric` or `islogical`

This function ignores empty values in the first input argument. Therefore, no error is thrown when the property or function argument value is empty.

Examples

Validate That Input Is Greater Than Or Equal To Specified Value

Use `mustBeGreaterThanOrEqual` to validate that the values in the array are greater than or equal to 3.

```
mustBeGreaterThanOrEqual([2,3,4],3)
```

```
Error using mustBeGreaterThanOrEqual (line 19)
Value must be greater than or equal to 3.
```

Restrict Property Values

Restrict property values to be greater than or equal to a specified value.

This class constrains the value of `Prop1` to be greater than or equal to `3`.

```
classdef MyClass
   properties
      Prop1 {mustBeGreaterThanOrEqual(Prop1,3)}
   end
end
```

Create an object and assign a value to its property.

```
obj = MyClass;
obj.Prop1 = 2;
```

```
Error setting property 'Prop1' of class 'MyClass':
Value must be greater than or equal to 3.
```

When you assign a value to the property, MATLAB calls `mustBeGreaterThanOrEqual` with the value being assigned to the property. `mustBeGreaterThanOrEqual` issues an error because the value `2` is not greater than or equal to `3`.

Restrict Function Argument Values

This function restricts the input argument to be values that are less than or equal to 5.

```
function r = mbGreaterThanOrEqual(x)
    arguments
        x {mustBeGreaterThanOrEqual(x,5)}
    end
    r = x - 5;
end
```

Calling the function with a vector that contains values that are greater than 5 does not meet the requirements defined with `mustBeGreaterThanOrEqual` and results in an error.

```
x = [12.7, 45.4, 4.9, 77.1, 53.1];
r = mbGreaterThanOrEqual(x);
```

```
Error using mbGreaterThanOrEqual
Invalid input argument at position 1. Value must be greater than or equal to 5.
```

Input Arguments

`value` — Value to validate
scalar | array

Value to validate, specified as a scalar or an array of one of the following:

- `logical`, `char`, `string`, and numeric class
- MATLAB classes that implement `ge`, `isreal`, and either `isnumeric` or `islogical`

`c` — Constant value that the `value` argument must be greater than or equal to
scalar

Constant value that the `value` argument must be greater than or equal to, specified as a scalar of one of the following:

- Any numeric class
- Logical class
- MATLAB classes that implement `ge`, `isreal`, and either `isnumeric` or `islogical`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`arguments` | `mustBeGreaterThan` | `mustBeLessThanOrEqual`

Topics

"Property Validation Functions"
"Function Argument Validation"

Introduced in R2017a

mustBeLessThan

Validate that value is less than another value or issue error

Syntax

```
mustBeLessThan(value,c)
```

Description

`mustBeLessThan(value,c)` issues an error if any elements in `value` are not less than the scalar `c`. This function does not return a value.

`mustBeLessThan` accepts user-defined objects if the class of the object implements these methods:

- `lt`
- `isreal`
- `isnumeric` or `islogical`

This function ignores empty values in the first input argument. Therefore, no error is thrown when the property or function argument value is empty.

Examples

Validate First Input Less Than Second

Use `mustBeLessThan` to validate that the values in the first input are less than the value of the second input.

```
mustBeLessThan([2 3 4],2)
```

```
Error using mustBeLessThan (line 19)
Value must be less than 2.
```

Restrict Property Values

Restrict property values to be less than a specified value.

This class constrains the value of `Prop1` to be less than 2.

```
classdef MyClass
   properties
      Prop1 {mustBeLessThan(Prop1,2)}
   end
end
```

Create an object and assign a value to its property.

```
obj = MyClass;
obj.Prop1 = 2;
```

```
Error setting property 'Prop1' of class 'MyClass':
Value must be less than 2.
```

When you assign a value to the property, MATLAB calls `mustBeLessThan` with the value being assigned to the property. `mustBeLessThan` issues an error because the value 2 is not less than 2.

Restrict Function Argument Values

This function restricts the input argument to be values that are less than 5.

```
function r = mbLessThan(x)
    arguments
        x {mustBeLessThan(x,5)}
    end
    r = x + 5;
end
```

Calling the function with a vector that contains values that are greater than or equal to 5 does not meet the requirements defined with `mustBeLessThan` and results in an error.

```
x = [1.27, 4.54, 3.9, 5.0, .531];
r = mbLessThan(x);
```

```
Error using mbLessThan
Invalid input argument at position 1. Value must be less than 5.
```

Input Arguments

`value` — Value to validate
scalar | array

Value to validate, specified as a scalar or an array of one of the following:

- `logical`, `char`, `string`, or numeric class
- MATLAB classes that implement `lt`, `isscalar`, `isreal`, and either `isnumeric` or `islogical`

When using `mustBeLessThan` as a property validator, this argument must be the property name, specified without quotation marks.

`c` — Constant value that the `value` argument must be less than
scalar

Constant value that the `value` argument must be less than, specified as a scalar of one of the following:

- Any numeric class
- Logical class
- MATLAB classes that implement `lt`, `isscalar`, `isreal`, and either `isnumeric` or `islogical`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`arguments` | `mustBeGreaterThan` | `mustBeLessThanOrEqual`

Topics

"Property Validation Functions"
"Function Argument Validation"

Introduced in R2017a

mustBeLessThanOrEqual

Validate that value is less than or equal to another value or issue error

Syntax

```
mustBeLessThanOrEqual(value,c)
```

Description

`mustBeLessThanOrEqual(value,c)` issues an error if any elements in `value` are not less than or equal to the scalar `c`. This function does not return a value.

`mustBeLessThanOrEqual` accepts user-defined objects if the class of the object implements these methods:

- `le`
- `isreal`
- `isnumeric` or `islogical`

This function ignores empty values in the first input argument. Therefore, no error is thrown when the property or function argument value is empty.

Examples

Validate First Input Less Than or Equal to Second

Use `mustBeLessThanOrEqual` to validate that the values in the first input are less than or equal to the value of the second input.

```
mustBeLessThanOrEqual([3 4 5],2)
```

```
Error using mustBeLessThanOrEqual (line 18)
Values must be less than or equal to 2.
```

Restrict Property Values

Restrict property values to be less than or equal to a specified value.

This class constrains the value of `Prop1` to be less than or equal to `2`.

```
classdef MyClass
   properties
      Prop1 {mustBeLessThanOrEqual(Prop1,2)}
   end
end
```

Create an object and assign a value to its property.

```
obj = MyClass;
obj.Prop1 = 3;
```

```
Error setting 'Prop1' property of 'MyClass' class:
Values must be less than or equal to 2.
```

When you assign a value to the property, MATLAB calls `mustBeLessThanOrEqual` with the value being assigned to the property. `mustBeLessThanOrEqual` issues an error because the value `3` is not less than or equal to `2`.

Restrict Function Argument Values

This function restricts the input argument to be values that are less than or equal to 5.

```
function r = mbLessThanOrEqual(x)
    arguments
        x {mustBeLessThanOrEqual(x,5)}
    end
    r = x + 5;
end
```

Calling the function with a vector that contains values that are not less than or equal to 5 does not meet the requirements defined with `mustBeLessThan` and results in an error.

```
x = [1.27, 4.54, 3.9, 5.1, .531];
r = mbLessThanOrEqual(x);
```

```
Error using mbLessThanOrEqual
Invalid input argument at position 1. Value must be less than or equal to 5.
```

Input Arguments

value — Value to validate

scalar | array

Value to validate, specified as a scalar or an array of one of the following:

- `logical`, `char`, or numeric classes
- MATLAB classes that implement `le`, `isscalar`, `isreal`, and either `isnumeric` or `islogical`

c — Constant value that the value argument must be less than or equal to

scalar

Constant value that the `value` argument must be less than or equal to, specified as a scalar of one of the following:

- `logical`, `char`, `string`, and numeric class
- MATLAB classes that implement `le`, `isscalar`, `isreal`, and either `isnumeric` or `islogical`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`arguments` | `mustBeGreaterThanOrEqual` | `mustBeLessThan`

Topics

"Property Validation Functions"
"Function Argument Validation"

Introduced in R2017a

mustBeNegative

Validate that value is negative or issue error

Syntax

```
mustBeNegative(value)
```

Description

`mustBeNegative(value)` issues an error if `value` is not negative. A value is negative if it is less than zero. This function does not return a value.

`mustBeNegative` accepts user-defined objects if the class of the object implements these methods:

- `lt`
- `isreal`
- `isnumeric` or `islogical`

This function ignores input arguments that are empty values. Therefore, no error is thrown when the property or function argument value is empty.

Examples

Validate That Values Are Negative

Use `mustBeNegative` to validate that the input contains only negative values.

The `rand` function creates uniformly distributed random numbers in the interval `(0,1)`. Test the array after the subtraction to validate that all values are negative. If not, `mustBeNegative` issues an error.

```
A = rand(1,5) - 0.75;
mustBeNegative(A)
```

```
Error using mustBeNegative (line 14)
Value must be negative.
```

Restrict Property to Negative Values

This class restricts the value of `Prop1` to negative values.

```
classdef MyClass
   properties
      Prop1 {mustBeNegative}
   end
end
```

Create an object and assign a value to `Prop1`.

```
obj = MyClass;
obj.Prop1 = rand(1,5) - 0.75;
```

```
Error setting property 'Prop1' of class 'MyClass':
Value must be negative.
```

When you assign a value to the property, MATLAB calls `mustBeNegative` with the value being assigned to the property. `mustBeNegative` issues an error if the any of the elements in the array are not negative.

Restrict Function Argument to Negative Values

This function declares two input arguments. Input `lower` must be negative and input `upper` must not be negative.

```
function r = mbNegative(lower,upper)
    arguments
        lower {mustBeNegative}
        upper {mustBeNonnegative}
    end
    x = lower*pi:upper*pi;
    r = sin(x);
end
```

Calling the function with a value for `lower` that does not meet the requirements of `mustBeNegative` results an error.

```
mbNegative(0,4)
```

```
Error using mbNegative
Invalid input argument at position 1. Value must be negative.
```

Input Arguments

value — Value to validate
scalar | array

Value to validate, specified as a scalar or an array of one of the following:

- `logical` or numeric class
- MATLAB classes that implement `lt`, `isreal`, and either `isnumeric` or `islogical`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`arguments` | `mustBeNonnegative`

Topics
"Property Validation Functions"
"Function Argument Validation"

Introduced in R2017a

mustBeNonempty

Validate that value is nonempty or issue error

Syntax

```
mustBeNonempty(value)
```

Description

`mustBeNonempty(value)` issues an error if `value` is empty. This function does not return a value.

`mustBeNonempty` accepts user-defined objects if the class of the object implements this method:

- `isempty`

Examples

Validate Input Not Empty

Use `mustBeNonempty` to validate that the input is not empty.

Create a `containers.Map`.

```
A = containers.Map;
```

Validate that `A` is not an empty value.

```
mustBeNonempty(A)
```

```
Error using mustBeNonempty (line 13)
Values must not be empty.
```

Creating a `containers.Map` with no input arguments results in an empty object.

Restrict Property Values

This class restricts the value of `Prop1` to nonempty values. The default value must also be nonempty.

```
classdef MyClass
   properties
      Prop1 {mustBeNonempty} = containers.Map(1,'First')
   end
end
```

Create an object and assign a value to its property.

```
obj = MyClass;
obj.Prop1 = containers.Map;
```

```
Error setting 'Prop1' property of 'MyClass' class:
Values must not be empty.
```

When you assign a value to the property, MATLAB calls `mustBeNonempty` with the value being assigned to the property. `mustBeNonempty` issues an error because the value assigned to `Prop1` is empty.

Restrict Function Argument Values

This function restricts the input argument to be a nonempty character vector.

```
function str = mbNonempty(name)
    arguments
        name (1,:) char  {mustBeNonempty}
    end
    str = sprintf('Name: %s',name);
end
```

Calling this function with an empty character vector results in an error being thrown by `mustBeNonempty`.

```
mbNonempty('')
```

```
Error using mbNonempty
Invalid input argument at position 1. Value must not be empty.
```

Input Arguments

`value` — Value to validate
scalar | array

Value to validate, specified as a scalar or array of any one of the following:

- Any MATLAB data type
- MATLAB classes that implement `isempty`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration`
Complex Number Support: Yes

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`arguments` | `mustBeMember` | `mustBeNonzero`

Topics
"Property Validation Functions"
"Function Argument Validation"

Introduced in R2017a

mustBeNonsparse

Validate that value is nonsparse or issue error

Syntax

```
mustBeNonsparse(value)
```

Description

`mustBeNonsparse(value)` issues an error if `value` is sparse. This function does not return a value.

`mustBeNonsparse` accepts user-defined objects if the class of the object implements this method:

- `issparse`

Examples

Validate Input Is Not Sparse

Use `mustBeNonsparse` to validate that the input is nonsparse.

Use the `sparse` function to create a sparse matrix.

```
A = [ 0   0   0   5
      0   2   0   0
      1   3   0   0
      0   0   4   0];
S = sparse(A);
```

Validate that `S` is nonsparse.

```
mustBeNonsparse(S)
```

```
Error using mustBeNonsparse (line 13)
Values must not be sparse.
```

Restrict Property Values

This class restricts the value of `Prop1` to nonsparse values.

```
classdef MyClass
   properties
      Prop1 {mustBeNonsparse}
   end
end
```

Create an object and assign a value to its property.

```
obj = MyClass;
A = [ 0   0   0   5
      0   2   0   0
      1   3   0   0
      0   0   4   0];
obj.Prop1 = sparse(A);
```

```
Error setting 'Prop1' property of 'MyClass' class:
Values must not be sparse.
```

When you assign a value to the property, MATLAB calls `mustBeNonsparse` with the value being assigned to the property. `mustBeNonsparse` issues an error because the value assigned to `Prop1` is sparse.

Restrict Function Argument Values

This function restricts the input argument to be any nonsparse value.

```
function mbNonsparse(S)
    arguments
        S {mustBeNonsparse}
    end
    disp(S)
end
```

Calling this function with a sparse array results in an error being thrown by `mustBeNonsparse`.

```
A = [ 0   0   0   5
      0   2   0   0
      1   3   0   0
      0   0   4   0];
S = sparse(A);
mbNonsparse(S)
```

```
Error using mbNonsparse
Invalid input argument at position 1. Value must not be sparse.
```

Input Arguments

`value` — Value to validate
scalar | array

Value to validate, specified as a scalar or array of any one of the following:

- `logical` or numeric class
- MATLAB classes that implement `issparse`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`arguments` | `mustBeNonempty` | `mustBeNumeric`

Topics

"Property Validation Functions"
"Function Argument Validation"

Introduced in R2017a

mustBeNonzero

Validate that value is nonzero or issue error

Syntax

```
mustBeNonzero(value)
```

Description

`mustBeNonzero(value)` issues an error if `value` is zero. This function does not return a value.

`mustBeNonzero` accepts user-defined objects if the class of the object implements these methods:

- `eq`
- `isnumeric` or `islogical`

Examples

Validate That Values Are Not Zero

Use `mustBeNonzero` to validate that the input does not contain values that are zero.

`A` is an array of numbers that is the result of a calculation:

```
A = sin([-1,0,1]);
msutBeNonzero(A)
```

```
Error using mustBeNonzero (line 14)
Value must not be zero.
```

Restrict Property Values

This class restrict the value of `Prop1` to be nonzero.

```
classdef MyClass
   properties
      Prop1 {mustBeNonzero}
   end
end
```

Create an object and assign a value to its property.

```
obj = MyClass;
obj.Prop1 = sin([-1,0,1]);
```

```
Error setting property 'Prop1' of class 'MyClass':
Value must not be zero.
```

When you assign a value to the property, MATLAB calls `mustBeNonzero` with the value being assigned to the property. `mustBeNonzero` issues an error because the one of the values in `A` is zero.

Restrict Function Argument Values

This function declares two input arguments. Input `A` must be a numeric vector and input `offset` must be a scalar that is not equal to zero.

```
function r = mbNonzero(A,offset)
    arguments
        A {mustBeNumeric}
        offset (1,1) {mustBeNonzero}
    end
    r = A + offset;
end
```

Calling the function with a value for `offset` equal to zero does not meet the requirements of `mustBeNonzero` and results in an error.

```
A = [12.7, 45.4, 98.9, 77.1, 53.1];
r = mbNonzero(A,0);
```

```
Error using mbNonzero
Invalid input argument at position 2. Value must not be zero.
```

Input Arguments

value — Value to validate
scalar | array

Value to validate, specified as a scalar or an array of one of the following:

- `logical` or numeric class
- MATLAB classes that implement `eq`, and either `isnumeric` or `islogical`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`arguments` | `mustBeNonnegative` | `mustBePositive`

Topics

"Property Validation Functions"
"Function Argument Validation"

Introduced in R2017a

mustBeNumeric

Validate that value is numeric or issue error

Syntax

```
mustBeNumeric(value)
```

Description

`mustBeNumeric(value)` issues an error if `value` is not numeric. This function does not return a value.

`mustBeNumeric` accepts user-defined objects if the class of the object implements this method:

- `isnumeric`

This function ignores input arguments that are empty values. Therefore, no error is thrown when the property or function argument value is empty.

Examples

Validate That Input Is Numeric

Validate that the result of an operation is numeric.

```
mustBeNumeric(5 < 10)
```

```
Error using mustBeNumeric (line 10)
Values must be numeric.
```

The relational operator for less than returns a logical value.

Restrict Property Values

This class restricts the value of `Prop1` to numeric values.

```
classdef MyClass
   properties
      Prop1 {mustBeNumeric}
   end
end
```

Create an object and assign a value to its property.

```
obj = MyClass;
obj.Prop1 = isprime(29);
```

```
Error setting 'Prop1' property of 'MyClass' class:
Values must be numeric.
```

When you assign a value to the property, MATLAB calls `mustBeNumeric` with the value being assigned to the property. `mustBeNumeric` issues an error because the value assigned to `Prop1` is a logical.

Restrict Function Values

This function restricts the input argument to a numeric vector.

```
function r = mbNumeric(x)
    arguments
        x (1,:) {mustBeNumeric}
    end
    p = [3 2 1];
    r = polyval(p,x);
end
```

Calling this function with a character vector results in an error being thrown by `mustBeNumeric`.

```
x = '4 3 2';
r = mbNumeric(x);
```

```
Error using mbNumeric
Invalid input argument at position 1. Value must be numeric.
```

Input Arguments

`value` — Value to validate
scalar | array

Value to validate, specified as a scalar or array of any of the following:

- Any MATLAB numeric class
- MATLAB classes that implement `isnumeric`

Other data types cause an error.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`
Complex Number Support: Yes

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`arguments` | `mustBeNumericOrLogical`

Topics
"Property Validation Functions"
"Function Argument Validation"

Introduced in R2017a

mustBeNumericOrLogical

Validate that value is numeric or logical or issue error

Syntax

```
mustBeNumericOrLogical(value)
```

Description

`mustBeNumericOrLogical(value)` issues an error if `value` is not numeric or logical. This function does not return a value.

`mustBeNumericOrLogical` accepts user-defined objects if the class of the object implements these methods:

- `isnumeric` or `islogical`

This function ignores input arguments that are empty values. Therefore, no error is thrown when the property or function argument value is empty.

Examples

Validate That Input Is Numeric or Logical

Validate that the value of an object property is a numeric or logical value.

```
a = axes;
mustBeNumericOrLogical(a.Visible)
```

```
Error using mustBeNumericOrLogical (line 11)
Values must be numeric or logical.
```

In this case, the value of the `Visible` property is a `char` vector, which results in an error.

Restrict Property Values

This class restricts the value of `Prop1` to numeric values.

```
classdef MyClass
   properties
      Prop1 {mustBeNumericOrLogical}
   end
end
```

Create an object and assign a value to its property.

```
obj = MyClass;
obj.Prop1 = get(gca,'Visible');
```

```
Error setting 'Prop1' property of 'MyClass' class:
Values must be numeric or logical.
```

When you assign a value to the property, MATLAB calls `mustBeNumericOrLogical` with the value being assigned to the property. `mustBeNumericOrLogical` issues an error because the value assigned to `Prop1` is a `char` vector.

Restrict Function Argument Values

This function restricts the input argument to a numeric or logical vector.

```
function r = mbNumericOrLogical(x)
    arguments
        x (1,:) {mustBeNumericOrLogical}
    end
    p = [3 2 1];
    r = polyval(p,x);
end
```

Calling this function with a character vector results in an error being thrown by `mustBeNumericOrLogical`.

```
x = '4 3 2';
r = mbNumericOrLogical(x);
```

```
Error using mbNumericOrLogical
Invalid input argument at position 1. Value must be numeric or logical.
```

Input Arguments

value — Value to validate
scalar | array

Value to validate, specified as a scalar or array of any one of the following:

- `logical` or numeric class
- MATLAB classes that implement `isnumeric` or `islogical`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`arguments` | `mustBeNumeric`

Topics

"Property Validation Functions"
"Function Argument Validation"

Introduced in R2017a

mustBeReal

Validate that value is real or issue error

Syntax

```
mustBeReal(value)
```

Description

`mustBeReal(value)` issues an error if `value` is not a real number. This function does not return a value.

`mustBeReal` accepts user-defined objects if the class of the object implements this method:

- `isreal`

This function ignores input arguments that are empty values. Therefore, no error is thrown when the property or function argument value is empty.

Examples

Validate Input Is Real

Use `mustBeReal` to validate that the input is a real number.

Validate that a calculation results in real numbers.

```
x = -2:2;
z = x.^exp(-x.^2);
mustBeReal(z)
```

```
Error using mustBeReal (line 13)
values must be real.
```

The value `z` is complex.

Restrict Property Values

This class restricts the value of `Prop1` to real values.

```
classdef MyClass
   properties
      Prop1 {mustBeReal}
   end
end
```

Create an object and assign a value to its property.

```
x = -2:0.1:2;
obj = MyClass;
obj.Prop1 = x.^exp(-x.^2);
```

```
Error setting 'Prop1' property of 'MyClass' class:
values must be real.
```

When you assign a value to the property, MATLAB calls `mustBeReal` with the value being assigned to the property. `mustBeReal` issues an error because the value assigned to `Prop1` is complex.

Restrain Function Argument Values

This function restricts the input argument to a scalar, real value.

```
function r = mbReal(z)
    arguments
        z (1,1) {mustBeReal}
    end
    r = randi(9,z);
end
```

Calling this function with a complex value results in an error being thrown by `mustBeReal`.

```
x = -1;
z = x.^exp(-x.^2);
r = mbReal(z);
```

```
Error using mbReal
Invalid input argument at position 1. Value must be real.
```

Input Arguments

`value` — Value to validate
scalar | array

Value to validate, specified as a scalar or array of any one of the following:

- `logical`, `char`, or numeric class
- MATLAB classes that implement `isreal`

Other data types cause an error.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`arguments` | `mustBeFinite` | `mustBeNonNan`

Topics
"Property Validation Functions"
"Function Argument Validation"

Introduced in R2017a

mustBeInteger

Validate that value is integer or issue error

Syntax

```
mustBeInteger(value)
```

Description

`mustBeInteger(value)` issues an error if `value` does not contain integer values. This function tests for numbers with no fractional part. The data type of the values is not considered. `mustBeInteger` does not return a value.

This function accepts user-defined objects if the class of the object implements these methods:

- `isreal`
- `isfinite`
- `floor`
- `eq`
- `isnumeric` or `islogical`

This function ignores input arguments that are empty values. Therefore, no error is thrown when the property or function argument value is empty.

Examples

Validate That Input Contains Only Integers

Validate that a calculation results in integer values.

```
A = randi(9)/randi(9);
mustBeInteger(A)
```

```
Error using mustBeInteger (line 14)
Values must be integer.
```

If the result of dividing the two random integers contains a fractional part, `mustBeInteger` issues an error.

Restrict Property to Integer Values

This class restricts the value of the property to be only integers.

```
classdef MyClass
   properties
      Prop1 {mustBeInteger}
   end
end
```

Create an object and assign a value to its property. The result of dividing two random integers can result in an integer or a number with a fractional part.

```
obj = MyClass;
obj.Prop1 = randi(9)/randi(9);
```

```
Error setting 'Prop1' property of 'MyClass' class:
Value must be integer.
```

When you assign a value to the property, MATLAB calls `mustBeInteger` with the value being assigned to the property. `mustBeInteger` issues an error when the value resulting from the division of these particular random integers does not result in an integer.

Restrict Function Argument Values

This function restricts the input argument to a value that has not fractional part.

```
function r = mbInteger(A,idx)
    arguments
        A (1,:) {mustBeNumeric}
        idx (1,1) {mustBeInteger}
    end
    r = A(idx).^2;
end
```

Calling this function with a noninteger value results in an error being thrown by `mustBeInteger`. The result of dividing two random integers can result in an integer or a number with a fractional part.

```
idx = randi(9)/randi(9);
A = rand(1,10);
r = mbInteger(A,idx);
```

```
Error using mbInteger
Invalid input argument at position 2. Value must be integer.
```

Input Arguments

`value` — Value to validate
scalar | array

Value to validate, specified as a scalar or array of any of the following:

- `logical`, `char`, or numeric class
- MATLAB classes that implement `eq`, `isreal`, `isfinite`, `floor`, and either `isnumeric` or `islogical`

Other data types cause an error.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`arguments` | `mustBeNumeric` | `mustBePositive`

Topics

"Property Validation Functions"

"Function Argument Validation"

Introduced in R2017a

mustBeMember

Validate that value is member of specified set

Syntax

```
mustBeMember(value,S)
```

Description

`mustBeMember(value,S)` issues an error if `value` is not a member of the set of values specified by `S`. The set of values specified by `value` must be a case-sensitive, exact match. `mustBeMember` does not return a value.

When using `mustBeMember` as a property or function argument validation function, ensure that the property default value is a member of the set.

This function accepts user-defined objects if the class of the object implements the following method:

- `ismember`

Examples

Validate That First Input Is Member of Set

Use `mustBeMember` to validate that the first input is a member of the set of values specified by the second input.

Validate that the character vector `'red'` is a member of set of character vectors, `'yellow'`, `'green'`, and `'blue'`.

```
A = 'red';
B = {'yellow','green','blue'};
mustBeMember(A,B)
```

```
Error using mustBeMember (line 14)
Value must be a member of this set
    'yellow'
    'green'
    'blue'
```

The validation failed because `'red'` is not a member of the set. MATLAB returns an error message listing the allowed values.

Restrict Property Values

This class restricts the value of the property to a specific set of values.

The value of `Prop1` must be `'yellow'`, `'green'`, or `'blue'`.

```
classdef MyClass
   properties
      Prop1 {mustBeMember(Prop1,{'yellow','green','blue'})} = 'yellow'
   end
end
```

The default property value must comply with the restrictions imposed by the validation function. Therefore, you must explicitly assign a default value that is a member of the set.

Create an object and assign a value to its property.

```
obj = MyClass
obj.Prop1 = 'red';
```

```
Error setting 'Prop1' property of 'MyClass' class:
Value must be a member of this set
    'yellow'
    'green'
    'blue'
```

The validation failed because `'red'` is not a member of the set. MATLAB returns an error message listing the allowed values.

Restrict Function Argument Values

This function declares two input arguments. Input `n` must be a scalar, numeric value, and input `typename` must be either of the character vectors `single` or `double`.

```
function r = mbMember(n,typename)
    arguments
        n (1,1) {mustBeNumeric}
        typename {mustBeMember(typename,{'single','double'})} = 'single'
    end
    r = rand(n,typename);
end
```

This call to the function uses a value for `typename` that does not meet the requirements defined with `mustBeMember` and results an error.

```
r = mbMember(5,'int32');
```

```
Error using mbMember
Invalid input argument at position 2. Value must be a member of this set:
    'single'
    'double'
```

Input Arguments

`value` — Value to validate
scalar | array

Value to validate, specified as a scalar or an array of one of the following:

- `logical`, `string`, `char`, or any numeric class
- Logical array, string array, `cellstr`, or numeric array
- MATLAB classes that implement `ismember`

When using `mustBeMember` as a property validator, this argument must be the property name, specified without quotation marks.

Example: `PropName {mustBeMember(PropName,{'High','Medium','Low'})} = 'Low'`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string`
Complex Number Support: Yes

S — Set of values to which `value` must belong
scalar | array

Set of values to which `value` must belong, specified as any of the following:

- `logical`, `string`, `char`, or any numeric class
- Logical array, string array, `cellstr`, or numeric array
- MATLAB classes that implement `ismember`

Example: Property with cell array of `char` vectors: `PropName {mustBeMember(PropName,{'yellow','green','blue'})} = 'blue'`

Example: Property with string array: `PropName {mustBeMember(PropName, ["yellow","green","blue"])} = "blue"`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string`
Complex Number Support: Yes

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`arguments` | `mustBeNonempty`

Topics

"Property Validation Functions"
"Function Argument Validation"

Introduced in R2017a

matlab.settings.mustBeLogicalScalar

Package: `matlab.settings`

Validate that setting value is logical array with one element

Syntax

```
matlab.settings.mustBeLogicalScalar(settingvalue)
```

Description

`matlab.settings.mustBeLogicalScalar(settingvalue)` issues an error if `settingvalue` is not a logical array with one element. This function does not return a value.

This function ignores input arguments that are empty values. Therefore, no error is thrown when the setting value is empty.

Examples

Validate That Setting Value Is Logical Scalar

Create a setting and specify a function to validate that the setting value is a logical scalar.

Create the settings group `mysettings`.

```
s = settings;
addGroup(s,'mysettings');
```

Add the setting `MyLogicalSetting` to `mysettings` and specify the validation function `matlab.settings.mustBeLogicalScalar`.

```
addSetting(s.mysettings,'MyLogicalSetting','ValidationFcn',@matlab.settings.mustBeLogicalScalar);
```

Now, test whether the validation function works. Set the value of `MyLogicalSetting` to a nonlogical value. As expected, MATLAB throws an error.

```
s.mysettings.MyLogicalSetting.PersonalValue = 'Hello';
```

```
Error setting 'MyLogicalSetting' in group 'mysettings': Value must be logical.
```

Input Arguments

`settingvalue` — Setting value to validate

scalar

Setting value to validate, specified as a scalar of one of the following:

- Any MATLAB logical class
- MATLAB classes that implement `isscalar` and `islogical`

Other data types cause an error.

See Also

`matlab.settings.mustBeIntegerScalar` | `matlab.settings.mustBeNumericScalar` | `matlab.settings.mustBeStringScalar`

Topics

"Access and Modify Settings"
"Create Custom Settings"
"Create Factory Settings for Toolboxes"

Introduced in R2019b

matlab.settings.mustBeNumericScalar

Package: `matlab.settings`

Validate that setting value is numeric array with one element

Syntax

```
matlab.settings.mustBeNumericScalar(settingvalue)
```

Description

`matlab.settings.mustBeNumericScalar(settingvalue)` issues an error if `settingvalue` is not a numeric array with one element. This function does not return a value.

This function ignores input arguments that are empty values. Therefore, no error is thrown when the setting value is empty.

Examples

Validate That Setting Value Is Numeric Scalar

Create a setting and specify a function to validate that the setting value is a numeric scalar.

Create the settings group `mysettings`.

```
s = settings;
addGroup(s,'mysettings');
```

Add the setting `MyNumericSetting` to `mysettings` and specify the validation function `matlab.settings.mustBeNumericScalar`.

```
addSetting(s.mysettings,'MyNumericSetting','ValidationFcn',@matlab.settings.mustBeNumericScalar);
```

Now, test whether the validation function works. Set the value of `MyNumericSetting` to a nonnumeric value. As expected, MATLAB throws an error.

```
s.mysettings.MyNumericSetting.PersonalValue = 'Hello';
```

```
Error setting 'MyNumericSetting' in group 'mysettings': Value must be numeric.
```

Input Arguments

`settingvalue` — Setting value to validate

scalar

Setting value to validate, specified as a scalar of one of the following:

- Any MATLAB numeric class
- MATLAB classes that implement `isscalar` and `isnumeric`

Other data types cause an error.

See Also

`matlab.settings.mustBeIntegerScalar` | `matlab.settings.mustBeLogicalScalar` | `matlab.settings.mustBeStringScalar`

Topics

"Access and Modify Settings"
"Create Custom Settings"
"Create Factory Settings for Toolboxes"

Introduced in R2019b

matlab.settings.mustBeStringScalar

Package: `matlab.settings`

Validate that setting value is string array with one element

Syntax

```
matlab.settings.mustBeStringScalar(settingvalue)
```

Description

`matlab.settings.mustBeStringScalar(settingvalue)` issues an error if `settingvalue` is not a string array with one element. This function does not return a value.

This function ignores input arguments that are empty values. Therefore, no error is thrown when the setting value is empty.

Examples

Validate That Setting Value Is String Scalar

Create a setting and specify a function to validate that the setting value is a string scalar.

Create the settings group `mysettings`.

```
s = settings;
addGroup(s,'mysettings');
```

Add the setting `MyStringSetting` to `mysettings` and specify the validation function `matlab.settings.mustBeStringScalar`.

```
addSetting(s.mysettings,'MyStringSetting','ValidationFcn',@matlab.settings.mustBeStringScalar);
```

Now, test whether the validation function works. Set the value of `MyStringSetting` to a nonstring value. As expected, MATLAB throws an error.

```
s.mysettings.MyStringSetting.PersonalValue = 11;
```

```
Error setting 'MyStringSetting' in group 'mysettings': Value must be string or convertible to string.
```

Input Arguments

`settingvalue` — Setting value to validate

scalar

Setting value to validate, specified as a scalar of one of the following:

- `string` class
- MATLAB classes that implement `isStringScalar`

Other data types cause an error.

See Also

`matlab.settings.mustBeIntegerScalar` | `matlab.settings.mustBeLogicalScalar` | `matlab.settings.mustBeNumericScalar`

Topics

"Access and Modify Settings"
"Create Custom Settings"
"Create Factory Settings for Toolboxes"

Introduced in R2019b

matlab.settings.mustBeIntegerScalar

Package: `matlab.settings`

Validate that setting value is integer array with one element

Syntax

```
matlab.settings.mustBeIntegerScalar(settingvalue)
```

Description

`matlab.settings.mustBeIntegerScalar(settingvalue)` issues an error if `settingvalue` is not an integer array with one element. This function does not return a value.

This function ignores input arguments that are empty values. Therefore, no error is thrown when the setting value is empty.

Examples

Validate That Setting Value Is an Integer Scalar

Create a setting and specify a function to validate that the setting value is an integer scalar.

Create the settings group `mysettings`.

```
s = settings;
addGroup(s,'mysettings');
```

Add the setting `MyIntegerSetting` to `mysettings` and specify the validation function `matlab.settings.mustBeIntegerScalar`.

```
addSetting(s.mysettings,'MyIntegerSetting','ValidationFcn',@matlab.settings.mustBeIntegerScalar);
```

Now, test whether the validation function works. Set the value of `MyIntegerSetting` to a noninteger value. As expected, MATLAB throws an error.

```
s.mysettings.MyIntegerSetting.PersonalValue = 'Hello';
```

```
Error setting 'MyIntegerSetting' in group 'mysettings': Value must be integer.
```

Input Arguments

`settingvalue` — Setting value to validate

scalar

Setting value to validate, specified as a scalar of one of the following:

- `logical`, `char`, or numeric class
- MATLAB classes that implement `isscalar`, `isreal`, `isfinite`, `floor`, `eq`, and either `isnumeric` or `islogical`

Other data types cause an error.

See Also

`matlab.settings.mustBeLogicalScalar` | `matlab.settings.mustBeNumericScalar` | `matlab.settings.mustBeStringScalar`

Topics

"Access and Modify Settings"
"Create Custom Settings"
"Create Factory Settings for Toolboxes"

Introduced in R2019b

namedargs2cell

Convert structure containing name-value pairs to cell array

Syntax

```
C = namedargs2cell(S)
```

Description

`C = namedargs2cell(S)` converts a scalar structure array containing name-value pairs to a cell array containing the names and values. This function converts a 1-by-1 structure with `n` number of fields to a 1-by-2`n` cell array with interleaved names and values.

Examples

Convert Structure to Interleaved Cell Array

Create a structure with the fields `XLim`, `Color`, and `Box` and assign values to each field. Use `namedargs2cell` to convert the structure to a cell array that interleaves the field names and the values.

```
S.XLim = [1,100];
S.Color = "red";
S.Box = "on";
C = namedargs2cell(S)
```

```
C = 1×6 cell array
    {'XLim'}    {1×2 double}    {'Color'}    {["red"]}    {'Box'}    {["on"]}
```

Convert Name-Value Structure for Name-Value Pairs

Write a function that accepts name-value pairs for specific axes object properties and impose additional restrictions on those property values.

Convert a name-value structure to a cell array containing interleaved names and values. Pass the cell array to the `axes` function, which accepts name-value pairs in a cell array. Return the handle to the axes object.

```
function a = myAxes(axesProps)
    arguments
        axesProps.XLim (1,2) {mustBeNumeric}
        axesProps.Color string {mustBeMember(axesProps.Color,["red","green","blue"])}
        axesProps.Box matlab.lang.OnOffSwitchState
    end
    nvCell = namedargs2cell(axesProps);
    a = axes(nvCell{:});
end
```

The output of the `namedarg2cell` function converts the `axesProps` name-value structure defined in the `arguments` block to an interleaved cell array of names and values. Pass the `nvCell` cell array to the axes function, which accepts name-value pairs as a cell array.

```
a = myAxes("XLim",[1,10],"Color","red","Box","on");
```

Input Arguments

S — Name-value structure

structure

Name-value structure, specified as a scalar structure array.

Data Types: `struct`

Output Arguments

C — Cell array of interleaved names and values

cell array

Cell array of interleaved names and values, returned as a 1-by-2n cell array, where n is the number of fields in the input structure.

Data Types: `cell`

See Also

`cell` | `struct`

Topics

"Function Argument Validation"

Introduced in R2019b

namelengthmax

Maximum identifier length

Syntax

```
l = namelengthmax
```

Description

`l = namelengthmax` returns the maximum length allowed for MATLAB identifiers. MATLAB identifiers include:

- Variable names
- Structure field names
- Script, function, and class names
- Model names

Use the `namelengthmax` function instead of hard-coding a specific maximum name length into your programs. Then, if the identifier length changes in a future MATLAB release, it is not necessary to update this limit in your code.

Examples

Determine Maximum MATLAB Identifier Length

```
l = namelengthmax
```

```
l = 63
```

See Also

`isvarname` | `matlab.lang.makeUniqueStrings` | `matlab.lang.makeValidName`

Introduced before R2006a

NaN

Create array of all NaN values

Syntax

```
X = NaN
X = NaN(n)
X = NaN(sz1,...,szN)
X = NaN(sz)

X = NaN(___,typename)
X = NaN(___,'like',p)
```

Description

`X = NaN` returns the scalar representation of "not a number". Operations return `NaN` when they have undefined numeric results, such as `0/0` or `0*Inf`.

`X = NaN(n)` returns an `n`-by-`n` matrix of `NaN` values.

`X = NaN(sz1,...,szN)` returns an `sz1`-by-...-by-`szN` array of `NaN` values, where `sz1,...,szN` indicate the size of each dimension. For example, `NaN(3,4)` returns a 3-by-4 matrix.

`X = NaN(sz)` returns an array of `NaN` values, where the size vector `sz` defines `size(X)`. For example, `NaN([3 4])` returns a 3-by-4 matrix.

`X = NaN(___,typename)` returns an array of `NaN` values of data type `typename`, which can be either `'single'` or `'double'`.

`X = NaN(___,'like',p)` returns an array of `NaN` values of the same data type, sparsity, and complexity (real or complex) as `p`. You can specify `typename` or `'like'` but not both.

Examples

Matrix of NaN Values

Create a 3-by-3 matrix of NaN values.

```
X = NaN(3)
```

```
X = 3×3

   NaN   NaN   NaN
   NaN   NaN   NaN
   NaN   NaN   NaN
```

3-D Array of NaN Values

Create a 2-by-3-by-4 array of NaN values and display its size.

```
X = NaN(2,3,4);
size(X)
```

```
ans = 1×3

     2     3     4
```

Clone Size from Existing Array

Create an array of NaN values that is the same size as an existing array.

```
A = [1 4; 2 5; 3 6];
sz = size(A);
X = NaN(sz)
```

```
X = 3×2

   NaN   NaN
```

```
   NaN   NaN
   NaN   NaN
```

It is a common pattern to combine the previous two lines of code into a single line.

```
X = NaN(size(A));
```

Specify Data Type of NaN Values

Create a 1-by-3 vector of NaN values whose elements are of type `single`.

```
X = NaN(1,3,'single')
```

```
X = 1x3 single row vector

   NaN   NaN   NaN
```

You can also specify the output type based on the type of another variable. Create a variable p of type `single`. Then, create a vector of NaN values with the same size and type as p.

```
p = single([1 2 3]);
X = NaN(size(p),'like',p)
```

```
X = 1x3 single row vector

   NaN   NaN   NaN
```

Input Arguments

n — Size of square matrix
integer

Size of square matrix, specified as an integer.

- If n is 0, then X is an empty matrix.

- If n is negative, then it is treated as 0.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

sz1,...,szN — Size of each dimension in a list
integers

Size of each dimension in a list, specified as separate integer arguments.

- If the size of any dimension is 0, then X is an empty array.
- If the size of any dimension is negative, then it is treated as 0.
- Beyond the second dimension, NaN ignores trailing dimensions of length 1. For example, NaN(3,1,1) creates a 3-by-1 vector of NaN values.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

sz — Size of each dimension in a vector
row vector of integers

Size of each dimension in a vector, specified as a row vector of integers.

- If the size of any dimension is 0, then X is an empty array.
- If the size of any dimension is negative, then it is treated as 0.
- Beyond the second dimension, NaN ignores trailing dimensions of length 1. For example, NaN([3 1 1]) creates a 3-by-1 vector of NaN values.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

typename — Data type
'double' (default) | single'

Data type to create, specified as 'double' or 'single'.

p — Prototype of array
array

Prototype of array to create, specified as an array.

Data Types: double | single

Complex Number Support: Yes

Tips

- `X = NaN` returns the scalar, type `double`, IEEE representation of "not a number". The exact bit-wise hexadecimal representation of this value is `fff8000000000000`. MATLAB preserves the "not a number" status of alternate `NaN` representations and treats all representations equivalently. In some special cases, due to hardware limitations for example, MATLAB does not preserve the exact bit pattern of the alternate representations during computation, and instead uses the canonical `NaN` bit pattern previously described.
- `NaN` values are not equal to each other. As a result, comparison operations involving `NaN` return false, except for the not equal operator ~=. For example, `NaN == NaN` returns logical 0 (`false`) but `NaN ~= NaN` returns logical 1 (`true`).
- `NaN` values in a vector are treated as different unique elements. For example, `unique([1 1 NaN NaN])` returns the row vector `[1 NaN NaN]`.
- Use the `isnan` or `ismissing` function to detect `NaN` values in an array. The `rmmissing` function detects and removes `NaN` values, and the `fillmissing` function detects `NaN` values and replaces them with non-`NaN` values.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Dimensions must be real, nonnegative, integers.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Dimensions must be real, nonnegative, integers.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `NaN` in the Parallel Computing Toolbox documentation.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `NaN` in the Parallel Computing Toolbox documentation.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`Inf` | `fillmissing` | `ismissing` | `isnan` | `missing` | `rmmissing`

Topics

“Missing Data in MATLAB”

Introduced before R2006a

nargchk

Validate number of input arguments

Note `nargchk` is not recommended. Use `narginchk` instead.

Syntax

```
msgText = nargchk(minArgs,maxArgs,numArgs)
msgStruct = nargchk(minArgs,maxArgs,numArgs,'struct')
```

Description

`msgText = nargchk(minArgs,maxArgs,numArgs)` validates the number of input arguments and returns a message if the number of inputs, `numArgs`, is fewer than `minArgs` or greater than `maxArgs`.

This syntax is the same as `msgText = nargchk(minArgs,maxArgs,numArgs,'string')`.

`msgStruct = nargchk(minArgs,maxArgs,numArgs,'struct')` returns a message structure instead of a character vector.

Examples

Check Number of Function Inputs

In a file named `checkInputs`, create a function that uses `nargchk` to verify that the function has been called with a valid number of inputs.

```
function checkInputs(varargin)
    msgTxt = nargchk(2,3,nargin)
end
```

Call the `checkInputs` function with a valid number of inputs. `nargchk` returns an empty character vector.

```
checkInputs(13,7)
```

```
msgTxt =

     []
```

Call the `checkInputs` function with too few inputs.

```
checkInputs(42)
```

```
msgTxt =

    'Not enough input arguments.'
```

Call the `checkInputs` function with too many inputs.

```
checkInputs(0,1,1,2,3)
```

```
msgTxt =

    'Too many input arguments.'
```

Pass `nargchk` Output Structure to `error` Function

In a file named `checkInputs`, create a function that uses `nargchk` with the `'struct'` parameter to verify that the function has been called with a valid number of inputs.

```
function checkInputs(varargin)
    msgStruct = nargchk(2,3,nargin,'struct');
    error(msgStruct)
end
```

At the command prompt, call the `checkInputs` function with an accepted number of inputs. `nargchk` does not throw an error.

```
checkInputs(13,7)
```

Call the `checkInputs` function with too few inputs.

```
checkInputs(42)
```

```
Error using checkInputs (line 3)
Not enough input arguments.
```

Call the `checkInputs` function with too many inputs.

```
checkInputs(0,1,1,2,3)
```

```
Error using checkInputs (line 3)
Too many input arguments.
```

Input Arguments

`minArgs` — Minimum number of accepted inputs
scalar

Minimum number of accepted inputs, specified as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`maxArgs` — Maximum number of accepted inputs
scalar

Maximum number of accepted inputs, specified as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`numArgs` — Number of function inputs
scalar

Number of function inputs, specified as a scalar. Typically, you use the `nargin` function to determine the number of input arguments specified in the function call.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`msgText` — Message text
`'Not enough input arguments.'` | `'Too many input arguments.'` | empty matrix

Message text, returned as `'Not enough input arguments.'`, `'Too many input arguments.'`, or an empty matrix.

If `numArgs` is less than `minArgs`, then `nargchk` returns the character vector `'Not enough input arguments.'` If `numArgs` is greater than `maxArgs`, then `nargchk` returns the character vector `'Too many input arguments.'` Otherwise, `nargchk` returns an empty matrix.

`msgStruct` — Message and identifier
structure

Message and identifier, returned as a structure with `message` and `identifier` fields. If `numArgs` is less than `minArgs`, then `nargchk` returns this structure:

```
       message: 'Not enough input arguments.'
    identifier: 'MATLAB:nargchk:notEnoughInputs'
```

If `numArgs` is greater than `maxArgs`, then `nargchk` returns this structure:

```
       message: 'Too many input arguments.'
    identifier: 'MATLAB:nargchk:tooManyInputs'
```

Otherwise, `nargchk` returns an empty structure.

Tips

- `nargchk` is often used with the `error` function. The `error` function accepts either type of return value from `nargchk`: a message character vector or message structure. For example, this command uses the output message structure from `nargchk` as the input to the `error` function.

  ```
  error(nargchk(2,4,nargin,'struct'))
  ```

  If the number of input arguments is within the expected range, then `nargchk` returns an empty character vector or structure. When the `error` function receives an empty character vector or structure, it does not throw an error.

See Also

`error` | `nargin` | `narginchk` | `nargout` | `nargoutchk` | `varargin` | `varargout`

Topics

"Validate Number of Function Arguments"

Introduced before R2006a

nargin

Number of function input arguments

Syntax

```
nargin
nargin(fun)
```

Description

`nargin` returns the number of function input arguments given in the call to the currently executing function. Use this syntax in the body of a function only. When using an `arguments` validation block, the value returned by `nargin` within a function is the number of positional arguments provided when the function is called. For more information, see “nargin in Argument Validation”.

`nargin(fun)` returns the number of input arguments that appear in the `fun` function definition. If the function includes `varargin` in its definition, then `nargin` returns the negative of the number of inputs. For example, if function `myFun` declares inputs `a`, `b`, and `varargin`, then `nargin('myFun')` returns `-3`.

If the input argument refers to a function that uses an `arguments` validation block, then the returned value is the number of declared positional arguments in the function definition as a non-negative value.

Examples

Inputs to Current Function

In a file named `addme.m`, create a function that accepts up to two inputs. Use `nargin` in the body of the function to determine the number of inputs.

```
type addme.m
```

```
function c = addme(a,b)
    switch nargin
        case 2
            c = a + b;
        case 1
            c = a + a;
        otherwise
            c = 0;
    end
end
```

At the command prompt, call the `addme` function with two inputs.

```
c = addme(13,42)
```

```
c = 55
```

Call the function with one input.

```
c = addme(13)
```

```
c = 26
```

Inputs Defined for a Function

Determine how many inputs a function accepts.

The function `addme` created in the previous example, has two inputs in its declaration statement (`a` and `b`). Define the name of the function as a character vector and use it as input for `nargin`.

```
fun = 'addme';
nargin(fun)
```

```
ans = 2
```

Inputs to Function That Uses `varargin`

Determine how many inputs a function that uses `varargin` can accept.

In a file named `mynewplot.m`, create a function that accepts numeric inputs `x` and `y` and any number of additional plot inputs using `varargin`.

```
type mynewplot.m
```

```
function mynewplot(x,y,varargin)
    figure
    plot(x,y,varargin{:})
    title('My New Plot')
end
```

Query how many inputs `newplot` can accept.

```
fx = 'mynewplot';
nargin(fx)
```

```
ans = -3
```

The minus sign indicates that the third input is `varargin`. The `mynewplot` function can accept an indeterminate number of additional input arguments.

Input Arguments

`fun` — Function
function handle | character vector | string scalar

Function for which `nargin` returns the number of input arguments from its definition, specified as a function handle, a character vector, or a string scalar.

Example: `@cos`

Example: `'plot'`

Data Types: `char` | `function_handle`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For the syntax `nargin(fun)`, if `fun` is a function handle or a function name that C/C++ code generation does not support, then the generated code for `nargin` returns 0.

See Also

`narginchk` | `nargout` | `varargin`

Topics

"Find Number of Function Arguments"
"Function Argument Validation"
"Argument Checking in Nested Functions"

Introduced before R2006a

narginchk

Validate number of input arguments

Syntax

`narginchk(minArgs,maxArgs)`

Description

`narginchk(minArgs,maxArgs)` validates the number of input arguments in the call to the currently executing function. `narginchk` throws an error if the number of inputs specified in the call is fewer than `minArgs` or greater than `maxArgs`. If the number of inputs is between `minArgs` and `maxArgs` (inclusive), then `narginchk` does nothing.

Examples

Verify Number of Inputs Is Within Expected Range

Verify that a function is called with a minimum of two and maximum of five input arguments.

In a file named `checkInputs.m`, create a function that uses `narginchk` to verify that the function has been called with a valid number of inputs. The function signature indicates that `checkInputs` requires two input arguments and accepts up to three additional, optional arguments.

```
function checkInputs(A,B,varargin)
    minArgs=2;
    maxArgs=5;
    narginchk(minArgs,maxArgs)

    fprintf('Received 2 required and %d optional inputs\n', length(varargin))
end
```

Call the function with one input argument.

```
checkInputs(13)
```

```
Error using checkInputs (line 4)
Not enough input arguments.
```

Call the function again with five input arguments.

```
checkInputs(13,7,42,1701,5)
```

```
Received 2 required and 3 optional inputs
```

Call the function again with six input arguments.

```
checkInputs(13,7,42,1701,5,88)
```

```
Error using checkInputs (line 4)
Too many input arguments.
```

Input Arguments

`minArgs` — Minimum number of accepted inputs
scalar

Minimum number of accepted inputs, specified as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`maxArgs` — Maximum number of accepted inputs
scalar

Maximum number of accepted inputs, specified as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Tips

- To verify that you have a minimum number of arguments, but no maximum number, set `maxArgs` to `inf`. For example: `narginchk(5,inf)` throws an error when there are fewer than five inputs.

- To verify that you have an exact number of arguments, specify the same value for `minArgs` and `maxArgs`. For example: `narginchk(3,3)` throws an error if you do not have exactly three inputs.

  If you call a function with too few inputs, the message identifier and message are:

  ```
  identifier: 'MATLAB:narginchk:notEnoughInputs'
     message: 'Not enough input arguments.'
  ```

  When too many inputs are supplied, the message identifier and message are:

  ```
  identifier: 'MATLAB:narginchk:tooManyInputs'
     message: 'Too many input arguments.'
  ```

- If `minArgs` is 0 and `maxArgs` is `nargin(fun)`, then you do not need to use `narginchk`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`nargin` | `varargin`

Topics

"Validate Number of Function Arguments"

Introduced in R2011b

nargout

Number of function output arguments

Syntax

```
nargout
nargout(fun)
```

Description

`nargout` returns the number of function output arguments specified in the call to the currently executing function. Use this syntax in the body of a function only.

`nargout(fun)` returns the number of outputs that appear in the `fun` function definition. If the function includes `varargout` in its definition, then `nargout` returns the negative of the number of outputs. For example, if function `myFun` declares outputs `y`, `z`, and `varargout`, then `nargout('myFun')` returns `-3`.

Examples

Outputs for Current Function

In a file named `subtract.m`, create a function that calculates a second return value, `absdif`, only if requested.

```
type subtract.m
```

```
function [dif,absdif] = subtract(y,x)
    dif = y-x;
    if nargout > 1
        disp('Calculating absolute value')
        absdif = abs(dif);
    end
end
```

At the command prompt, call the `subtract` function with one return value.

```
diff = subtract(42,13)
```

```
diff = 29
```

Call the `subtract` function again with two return values.

```
[dif,absdif] = subtract(2,5)
```

```
Calculating absolute value

dif = -3

absdif = 3
```

Outputs Defined for a Function

Determine how many outputs a function can return.

The function `subtract` created in the previous example has two outputs in its declaration statement (`dif` and `absdif`).

```
fun = @subtract;
nargout(fun)
```

```
ans = 2
```

Outputs from Function That Uses `varargout`

Determine how many outputs a function that uses `varargout` can return.

In a file named `mySize.m`, create a function that returns a vector of dimensions from the `size` function and the individual dimensions using `varargout`.

```
type mySize.m
```

```
function [sizeVector,varargout] = mySize(x)
    sizeVector = size(x);
    varargout = cell(1,nargout-1);
```

```
    for k = 1:length(varargout)
        varargout{k} = sizeVector(k);
    end
end
```

Query how many outputs `mySize` can return.

```
fun = 'mySize';
nargout(fun)
```

```
ans = -2
```

The minus sign indicates that the second output is `varargout`. The `mySize` function can return an indeterminate number of additional outputs.

Input Arguments

`fun` — Function
function handle | character vector | string scalar

Function for which `nargout` returns the number of output arguments from its definition, specified as a function handle, a character vector, or a string scalar.

Example: `@rand`

Example: `'sortrows'`

Data Types: `char` | `function_handle`

Tips

- When you use a function as part of an expression, such as an `if` statement, then MATLAB calls the function with one output argument. Therefore, the `nargout` function returns `1` within expressions.
- If you check for a `nargout` value of 0 within a function and you specify the value of the output, MATLAB populates `ans`. However, if you check `nargout` and do not specify a value for the output, then MATLAB does not modify `ans`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For code generation, when you use `nargout` in an entry-point (top-level) function, either the number of outputs in the function definition or the `codegen -nargout` syntax determine the `nargout` result.
- For the syntax `nargout(fun)`, if `fun` is a function handle or function name that C/C++ code generation does not support, then the generated code for `nargout` returns 0.

See Also

`nargoutchk` | `varargout`

Topics

"Find Number of Function Arguments"
"Argument Checking in Nested Functions"

Introduced before R2006a

nargoutchk

Validate number of output arguments

Syntax

```
nargoutchk(minArgs,maxArgs)
msgText = nargoutchk(minArgs,maxArgs,numArgs)
msgStruct = nargoutchk(minArgs,maxArgs,numArgs,'struct')
```

Description

`nargoutchk(minArgs,maxArgs)` validates the number of output arguments specified in the current function call. `nargoutchk` throws an error if the number of outputs is less than `minArgs` or greater than `maxArgs`. If the number of outputs is between `minArgs` and `maxArgs` (inclusive), then `nargoutchk` does nothing.

`msgText = nargoutchk(minArgs,maxArgs,numArgs)` validates the value of `numArgs`, and returns a message if `numArgs` is less than `minArgs` or greater than `maxArgs`. The use of this syntax is not recommended.

`msgStruct = nargoutchk(minArgs,maxArgs,numArgs,'struct')` returns a message structure. The use of this syntax is not recommended.

Examples

Number of Outputs Within Expected Range

Verify that a function is called with a minimum of two and maximum of five output arguments.

In a file named `checkOutputs.m`, create a function that uses `nargoutchk` to verify that the function has been called with a valid number of outputs. The function signature indicates that `checkOutputs` accepts a variable number of output arguments.

```
function varargout = checkOutputs(varargin)
minArgs=2;
maxArgs=5;
nargoutchk(minArgs,maxArgs)

disp("You requested " + nargout + " outputs.")

varargout = cell(nargout,1);
for k=1:nargout
    varargout{k} = randi(100);
end
end
```

Call the function with one output argument.

```
a = checkOutputs(13)
```

```
Error using checkOutputs (line 4)
Not enough output arguments.
```

Call the function again with five output arguments.

```
[a,b,c,d,e] = checkOutputs(7,42);
```

```
You requested 5 outputs.
```

Call the function again with six output arguments.

```
[a,b,c,d,e,f] = checkOutputs(7,42);
```

```
Error using checkOutputs (line 4)
Too many output arguments.
```

Input Arguments

`minArgs` — Minimum number of accepted outputs
scalar

Minimum number of accepted outputs, specified as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`maxArgs` — Maximum number of accepted outputs
scalar

Maximum number of accepted outputs, specified as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`numArgs` — Number of function outputs
scalar

Note Use of `numArgs` is not recommended.

Number of function outputs, specified as a scalar. Typically, you use the `nargout` function to determine the number of output arguments specified in the function call.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`msgText` — Message text
`'Not enough output arguments.'` | `'Too many output arguments.'` | empty matrix

Note Use of `msgText` is not recommended.

Message text, returned as `'Not enough output arguments.'`, `'Too many output arguments.'`, or an empty matrix.

If `numArgs` is less than `minArgs`, then `nargoutchk` returns the character vector `'Not enough output arguments.'` If `numArgs` is greater than `maxArgs`, then `nargoutchk` returns the character vector `'Too many output arguments.'` Otherwise, `nargoutchk` returns an empty matrix.

`msgStruct` — Message and identifier
structure

Note Use of `msgStruct` is not recommended.

Message and identifier, returned as a structure with `message` and `identifier` fields. If `numArgs` is less than `minArgs`, `nargoutchk` returns this structure:

```
       message: 'Not enough output arguments.'
    identifier: 'MATLAB:nargoutchk:notEnoughOutputs'
```

If `numArgs` is greater than `maxArgs`, `nargoutchk` returns this structure:

```
       message: 'Too many output arguments.'
    identifier: 'MATLAB:nargoutchk:tooManyOutputs'
```

Otherwise, `nargoutchk` returns an empty structure.

Tips

- To verify that you have a minimum number of arguments, but no maximum number, set `maxArgs` to `inf`. For example: `nargoutchk(5,inf)` throws an error when there are fewer than five outputs.
- To verify that you have an exact number of arguments, specify the same value for `minArgs` and `maxArgs`. For example: `nargoutchk(3,3)` throws an error when you do not have exactly three outputs.
- If `minArgs` is 0 and `maxArgs` is `nargout`, then you do not need to use `nargoutchk`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`nargout` | `varargout`

Topics

"Validate Number of Function Arguments"

Introduced before R2006a

NaT

Not-a-Time

Syntax

```
NaT
t = NaT(n)
t = NaT(sz1,...,szN)
t = NaT(sz)

t = NaT( ___ ,'Format',fmt)
t = NaT( ___ ,'TimeZone',tz)
```

Description

`NaT` is the representation for Not-a-Time, a value that can be stored in a `datetime` array to indicate an unknown or missing `datetime` value.

The `datetime` function creates a `NaT` value automatically when it cannot convert text to a `datetime` value, or for elements in a `datetime` array where the `Year`, `Month`, `Day`, `Hour`, `Minute`, or `Second` properties are set to `NaN`. You also can assign the character vector, `'NaT'`, to elements of an existing `datetime` array. Use the `NaT` function to create a new `datetime` array containing only `NaT` values.

`NaT` returns a scalar Not-a-Time (`NaT`) `datetime` value.

`t = NaT(n)` returns an `n`-by-`n` matrix of `NaT` values.

`t = NaT(sz1,...,szN)` returns a `sz1`-by-...-by-`szN` array of `NaT` values where `sz1,...,szN` indicates the size of each dimension. For example, `NaT(3,4)` returns a 3-by-4 array of `NaT` values.

`t = NaT(sz)` returns an array of `NaT` values where the size vector, `sz`, defines `size(t)`. For example, `NaT([3,4])` returns a 3-by-4 array of `NaT` values.

`t = NaT( ___ ,'Format',fmt)` returns a `datetime` array with the specified display format. Use this syntax to initialize a `datetime` array. Not-a-Time values always display as `NaT`, but non-`NaT` values assigned to the array will display using the specified format.

`t = NaT( ___ ,'TimeZone',tz)` returns an array of `NaT` values in the time zone specified by `tz`.

Examples

Square NaT Matrix

Create a 3-by-3 matrix of `NaT` values.

```
t = NaT(3)
```

```
t = 3x3 datetime array
   NaT   NaT   NaT
   NaT   NaT   NaT
   NaT   NaT   NaT
```

Initialize Rectangular datetime Array and Specify Format

Create a 2-by-3 array of `NaT` values and specify a date format.

```
t = NaT(2,3,'Format','dd/MM/yyyy')
```

```
t = 2x3 datetime array
   NaT   NaT   NaT
   NaT   NaT   NaT
```

Assign a `datetime` value to an element of `t`.

```
t(1,2) = datetime('today')
```

```
t = 2x3 datetime array
   NaT          27/08/2019   NaT
   NaT          NaT          NaT
```

Input Arguments

n — Size of square matrix
integer

Size of square matrix, specified as an integer.

- If `n` is `0`, then `t` is an empty matrix.
- If `n` is negative, then it is treated as `0`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

sz1,...,szN — Size of each dimension (as separate arguments)
integer values

Size of each dimension, specified as separate arguments of integer values.

- If the size of any dimension is `0`, then `t` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

sz — Size of each dimension (as a row vector)
integer values

Size of each dimension, specified as a row vector of integer values. Each element of this vector indicates the size of the corresponding dimension:

- If the size of any dimension is `0`, then `t` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.

Example: `sz = [2,3,4]` creates a 2-by-3-by-4 array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

fmt — Date format
`'default'` (default) | `'defaultdate'` | character vector | string scalar

Date format, specified as a character vector or string scalar. Use the letters `A-Z` and `a-z` to define the format. For a complete list of valid letter identifiers, see the `Format` property for datetime arrays.

Alternatively, use one of the following character vectors to specify a default format.

Value of `fmt`	Description
`'default'`	Use the default display format.
`'defaultdate'`	Use the default display format for datetime values created without time components.

Example: `'yyyy MMM dd'`

Data Types: `char` | `string`

`tz` — Time zone region

`''` (default) | character vector | string scalar

Time zone region, specified as a character vector or string scalar.

The value of `tz` can be:

- `''`, to create an “unzoned” `datetime` array that does not belong to a specific time zone.
- The name of a time zone region from the IANA Time Zone Database, for example, `'America/Los_Angeles'`. The name of a time zone region accounts for the current and historical rules for standard and daylight offsets from UTC that are observed in a geographic region.
- An ISO 8601 character vector of the form `+HH:mm` or `-HH:mm`, for example, `'+01:00'`, to specify a time zone that is a fixed offset from UTC.
- `'UTC'`, to create a `datetime` array in Universal Coordinated Time.
- `'UTCLeapSeconds'`, to create a `datetime` array in Universal Coordinated Time that accounts for leap seconds.
- `'local'`, to create a `datetime` array in the system time zone.

This table lists some common names of time zone regions from the IANA Time Zone Database.

Value of TimeZone	UTC Offset	UTC DST Offset
'Africa/Johannesburg'	+02:00	+02:00
'America/Chicago'	−06:00	−05:00
'America/Denver'	−07:00	−06:00
'America/Los_Angeles'	−08:00	−07:00
'America/New_York'	−05:00	−04:00
'America/Sao_Paulo'	−03:00	−02:00
'Asia/Hong_Kong'	+08:00	+08:00
'Asia/Kolkata'	+05:30	+05:30
'Asia/Tokyo'	+09:00	+09:00
'Australia/Sydney'	+10:00	+11:00
'Europe/London'	+00:00	+01:00
'Europe/Zurich'	+01:00	+02:00

Data Types: `char` | `string`

See Also

`datetime` | `isnat` | `nan`

Introduced in R2015b

native2unicode

Convert numeric bytes to Unicode character representation

Syntax

```
unicodestr = native2unicode(bytes)
unicodestr = native2unicode(bytes, encoding)
```

Description

`unicodestr = native2unicode(bytes)` converts a numeric vector, `bytes`, from the user default encoding to a Unicode character representation. `native2unicode` treats `bytes` as a vector of 8-bit bytes, and each value must be in the range [0,255]. The output argument `unicodestr` is a character vector having the same general array shape as `bytes`.

`unicodestr = native2unicode(bytes, encoding)` converts `bytes` to a Unicode representation with the assumption that `bytes` is in the character encoding scheme specified by `encoding`. The input argument `encoding` must have no characters (`''`) or it must be a name or alias for an encoding scheme. Some examples are `'UTF-8'`, `'latin1'`, `'US-ASCII'`, and `'Shift_JIS'`. If `encoding` is unspecified or has no characters (`''`), the default encoding scheme is used. `encoding` can be a character vector or a string scalar.

Note If `bytes` is a character vector or a string scalar, it is returned unchanged.

Examples

This example begins with a vector of bytes in an unknown character encoding scheme. The user-written function `detect_encoding` determines the encoding scheme. If successful, it returns the encoding scheme name or alias as a character vector. If unsuccessful, it throws an error represented by an MException object, `ME`. The example calls `native2unicode` to convert the bytes to Unicode representation:

```
try
    enc = detect_encoding(bytes);
    str = native2unicode(bytes, enc);
    disp(str);
catch ME
    rethrow(ME);
end
```

Note that the computer must be configured to display text in a language represented by the detected encoding scheme for the output of `disp(str)` to be correct.

See Also

`unicode2native`

Introduced before R2006a

nchoosek

Binomial coefficient or all combinations

Syntax

```
b = nchoosek(n,k)
C = nchoosek(v,k)
```

Description

`b = nchoosek(n,k)` returns the binomial coefficient, defined as

$$ {}_nC_k = \binom{n}{k} = \frac{n!}{(n-k)!\ k!} \ . $$

This is the number of combinations of `n` items taken `k` at a time. `n` and `k` must be nonnegative integers.

`C = nchoosek(v,k)` returns a matrix containing all possible combinations of the elements of vector `v` taken `k` at a time. Matrix `C` has `k` columns and $m!/((m–k)!\ k!)$ rows, where m is `length(v)`.

Examples

Binomial Coefficient, "5 Choose 4"

```
b = nchoosek(5,4)
```

```
b = 5
```

All Combinations of Five Numbers Taken Four at a Time

```
v = 2:2:10;
C = nchoosek(v,4)
```

```
C = 5×4

     2     4     6     8
     2     4     6    10
     2     4     8    10
     2     6     8    10
     4     6     8    10
```

All Combinations of Three Unsigned Integers Taken Two at a Time

```
v = uint16([10 20 30]);
C = nchoosek(v,uint16(2))
```

```
C = 3x2 uint16 matrix

   10   20
   10   30
   20   30
```

Input Arguments

n — Number of possible choices
nonnegative integer scalar

Number of possible choices, specified as a nonnegative integer scalar. `n` can be any numeric type, but must be real.

Example: `10`

Example: `int16(10)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

k — Number of selected choices
nonnegative integer scalar

Number of selected choices, specified as a nonnegative integer scalar. k can be any numeric type, but must be real. `nchoosek(n,k)` requires that n and k be the same type or that at least one of them be of type `double`.

There are no restrictions on combining inputs of different types for `nchoosek(v,k)`.

Example: 3

Example: `int16(3)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

v — Set of all choices
vector

Set of all choices, specified as a vector.

Example: `[1 2 3 4 5]`

Example: `[1+1i 2+1i 3+1i 4+1i]`

Example: `int16([1 2 3 4 5])`

Example: `[true false true false]`

Example: `['abcd']`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

Output Arguments

b — Binomial coefficient
nonnegative scalar value

Binomial coefficient, returned as a nonnegative scalar value. b is the same type as n and k. If n and k are of different types, then b is returned as the nondouble type.

C — All combinations of v
matrix

All combinations of `v`, returned as a matrix of the same type as `v`. Matrix `C` has `k` columns and $n!/((n–k)!\ k!)$ rows, where n is `length(v)`.

Each row of `C` contains a combination of `k` items chosen from `v`. The elements in each row of `C` are listed in the same order as they appear in `v`.

If `k > numel(v)`, then `C` is an empty matrix.

Limitations

- When `b = nchoosek(n,k)` is sufficiently large, `nchoosek` displays a warning that the result might not be exact. In this case, the result is only accurate to 15 digits for double-precision inputs, or 8 digits for single-precision inputs.
- `C = nchoosek(v,k)` is only practical for situations where `length(v)` is less than about `15`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- When the first input, `x`, is a scalar, `nchoosek` returns a binomial coefficient. In this case, `x` must be a nonnegative integer. It cannot have type `int64` or `uint64`.
- When the first input, `x`, is a vector, `nchoosek` treats it as a set. In this case, `x` can have type `int64` or `uint64`.
- The second input, `k`, cannot have type `int64` or `uint64`.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

See Also

`perms`

Introduced before R2006a

ndgrid

Rectangular grid in N-D space

Syntax

```
[X1,X2,...,Xn] = ndgrid(x1,x2,...,xn)
[X1,X2,...,Xn] = ndgrid(xg)
```

Description

`[X1,X2,...,Xn] = ndgrid(x1,x2,...,xn)` replicates the grid vectors `x1,x2,...,xn` to produce an `n`-dimensional full grid.

`[X1,X2,...,Xn] = ndgrid(xg)` specifies a single grid vector `xg` to use for all dimensions. The number of output arguments you specify determines the dimensionality `n` of the output.

Examples

Create 2-D Grid

Create a 2-D grid from the vectors `[1 3 5 7 9 11 13 15 17 19]` and `[2 4 6 8 10 12]`.

```
[X,Y] = ndgrid(1:2:19,2:2:12)
```

```
X = 10×6

     1     1     1     1     1     1
     3     3     3     3     3     3
     5     5     5     5     5     5
     7     7     7     7     7     7
     9     9     9     9     9     9
    11    11    11    11    11    11
```

```
    13    13    13    13    13    13
    15    15    15    15    15    15
    17    17    17    17    17    17
    19    19    19    19    19    19


Y = 10×6

     2     4     6     8    10    12
     2     4     6     8    10    12
     2     4     6     8    10    12
     2     4     6     8    10    12
     2     4     6     8    10    12
     2     4     6     8    10    12
     2     4     6     8    10    12
     2     4     6     8    10    12
     2     4     6     8    10    12
     2     4     6     8    10    12
```

Evaluate Function Over Gridded Domain

Evaluate and plot the function

$$x_1 e^{-x_1^2 - x_2^2}$$

over the gridded domain

$-2 < x_1 < 2$ and $-2 < x_2 < 2$.

Create a grid of values for the domain.

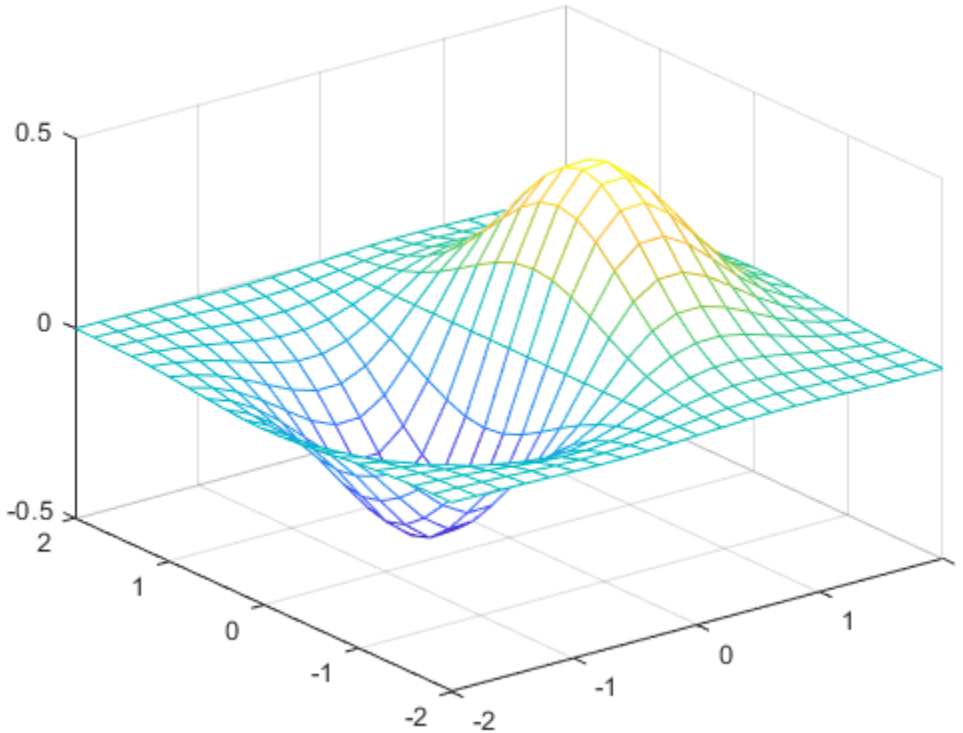
```
[X1,X2] = ndgrid(-2:.2:2);
```

Evaluate the function over the domain.

```
Z = X1 .* exp(-X1.^2 - X2.^2);
```

Generate a mesh plot of the function.

```
mesh(X1,X2,Z)
```



In R2016b and later releases, this task does not require the use of `ndgrid`. Instead, you can construct the grid using implicit expansion with these commands:

```
x = -2:.2:2;

Z1 = x.' .* exp(-(x.').^2 - x.^2);
```

Interpolate Data

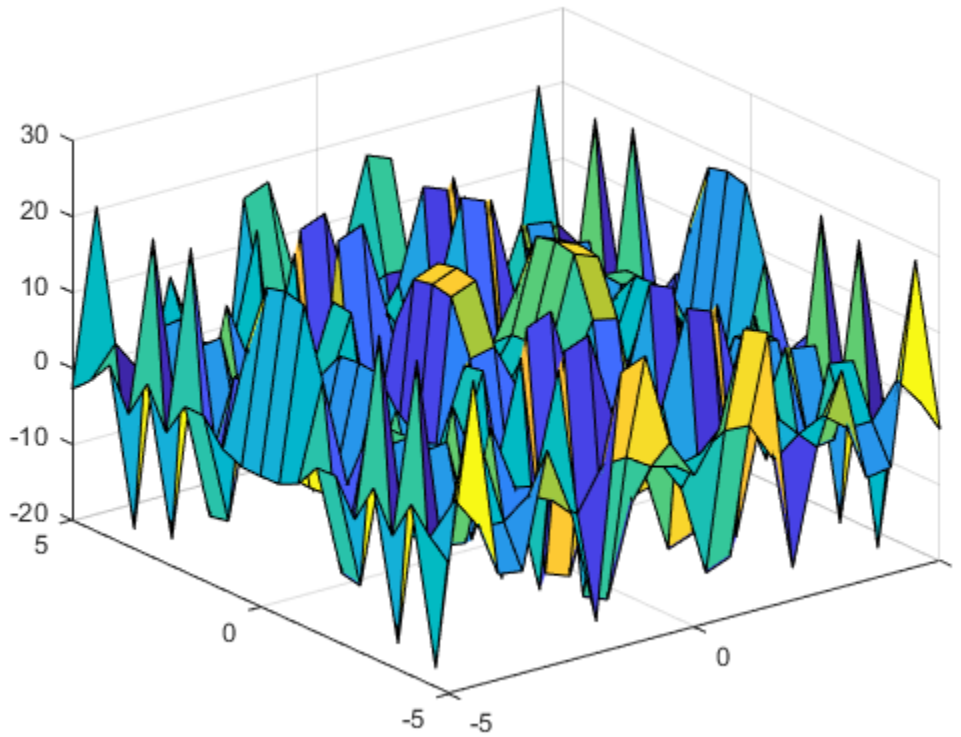
Create a 2-D grid and calculate some function values on the grid. Interpolate between the assigned values to refine the grid.

Create a coarse grid for (x, y) in the range $[-5, 5]$.

```
[X,Y] = ndgrid(-5:0.5:5);
```

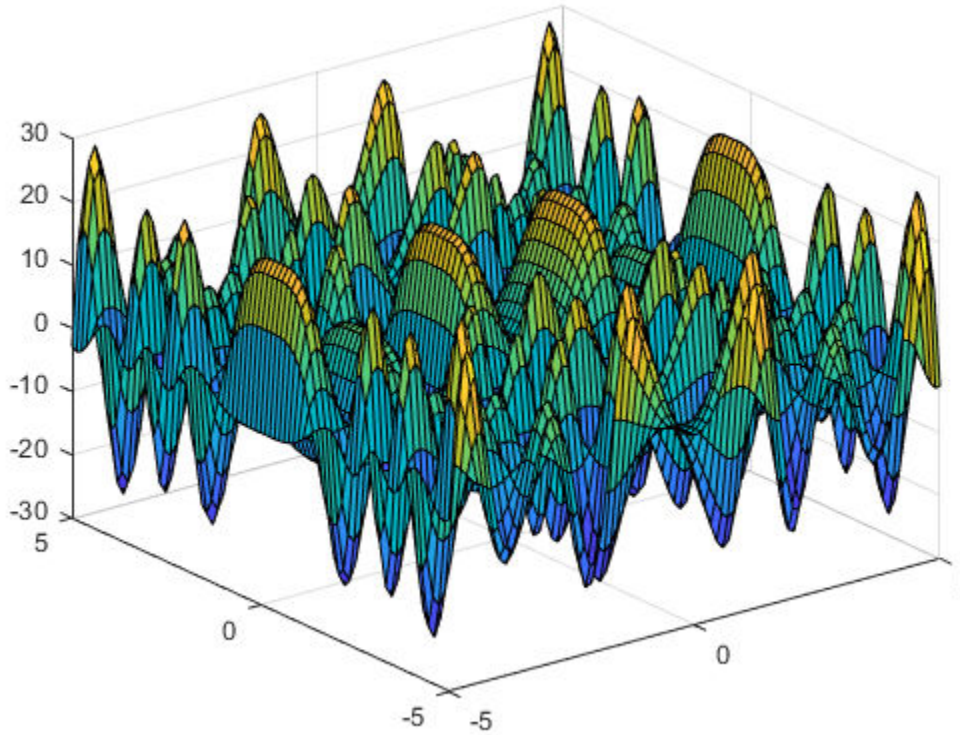
Calculate some function values on the grid and plot the function.

```
f = sin(X.^2) * cos(Y.^2);
surf(X,Y,f)
```



Interpolate between the points using a more refined grid and plot the result.

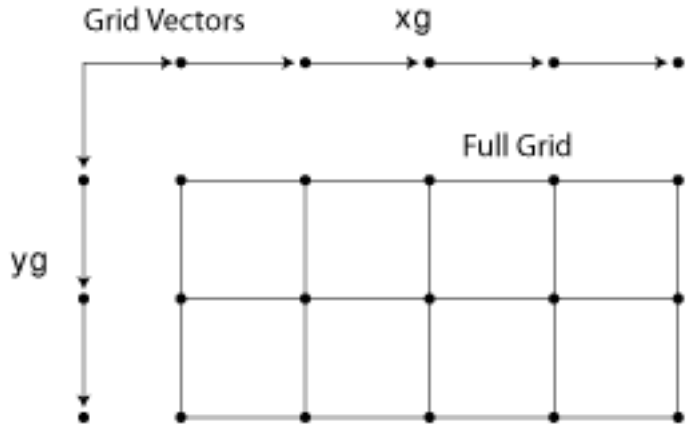
```
[X1,Y1] = ndgrid(-5:0.125:5);
F = interpn(X,Y,f,X1,Y1,'spline');

surf(X1,Y1,F)
```



Input Arguments

x1,x2,...,xn — Grid vectors (as separate arguments)
vectors

Grid vectors, specified as vectors containing grid coordinates for each dimension. The grid vectors implicitly define the grid. For example, in 2-D:



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`
Complex Number Support: Yes

xg — Grid vector for all dimensions
vectors

Grid vector for all dimensions, specified as a vector containing grid coordinates. `ndgrid` uses `xg` as the grid vector for each dimension.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`
Complex Number Support: Yes

Output Arguments

`X1,X2,...,Xn` — Full grid representation
array

Full grid representation, returned as separate arrays. For each output array `Xi`, the `i`th dimension contains copies of the grid vector `xi`.

Tips

- The `ndgrid` function is similar to `meshgrid`. However, `ndgrid` supports 1-D to N-D while `meshgrid` is restricted to 2-D and 3-D.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The 1-D syntax, `X = ndgrid(x)`, returns a `gpuArray` column vector `X` that contains the elements of the input `gpuArray` `x` for use as a one-dimensional grid.
- The inputs must be floating-point double or single.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- The 1-D syntax, `X = ndgrid(x)`, returns a distributed array column vector `X` that contains the elements of the input distributed array `x` for use as a one-dimensional grid.
- The inputs must be floating-point double or single.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`griddedInterpolant` | `mesh` | `meshgrid` | `surf`

Topics

“Interpolating Gridded Data”

Introduced before R2006a

ndims

Number of array dimensions

Syntax

```
N = ndims(A)
```

Description

`N = ndims(A)` returns the number of dimensions in the array `A`. The number of dimensions is always greater than or equal to *2*. The function ignores trailing singleton dimensions, for which `size(A,dim) = 1`.

Examples

Find Dimensions of Vector

Create a row vector.

```
A = 1:5;
```

Find the number of dimensions in the vector.

```
ndims(A)
```

```
ans = 2
```

The result is 2 because the vector has a size of 1-by-5.

Find Dimensions of Cell Array

Create a cell array of character vectors.

```
A{1,1,1} = 'cell_1';
A{1,1,2} = 'cell_2';
A{1,1,3} = 'cell_3'
```

```
A = 1x1x3 cell array
A(:,:,1) =

    {'cell_1'}


A(:,:,2) =

    {'cell_2'}


A(:,:,3) =

    {'cell_3'}
```

Find the number of dimensions of the cell array.

```
ndims(A)
```

```
ans = 3
```

The result is 3 because the cell array has a size of 1-by-1-by-3.

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array | table | timetable

Input array, specified as a scalar, vector, matrix, multidimensional array, table, or timetable.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `cell` | `table` | `timetable` | `categorical` | `datetime` | `duration` | `calendarDuration`

Algorithms

The number of dimensions in an array is the same as the length of the size vector of the array. In other words, `ndims(A) = length(size(A))`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`length` | `size`

Introduced before R2006a

ne, ~=

Determine inequality

Syntax

```
A ~= B
ne(A,B)
```

Description

`A ~= B` returns a logical array with elements set to logical `1` (`true`) where arrays `A` and `B` are not equal; otherwise, the element is logical `0` (`false`). The test compares both real and imaginary parts of numeric arrays. `ne` returns logical `1` (`true`) where `A` or `B` have NaN or undefined categorical elements.

`ne(A,B)` is an alternative way to execute `A ~= B`, but is rarely used. It enables operator overloading for classes.

Examples

Inequality of Two Vectors

Create two vectors containing both real and imaginary numbers, then compare the vectors for inequality.

```
A = [1+i 3 2 4+i];
B = [1 3+i 2 4+i];
A ~= B
```

```
ans = 1x4 logical array

   1   1   0   0
```

The `ne` function tests both real and imaginary parts for inequality, and returns logical `1` (`true`) where one or both parts are not equal.

Find Characters

Create a character vector.

```
M = 'masterpiece';
```

Test for the presence of a specific character using `~=`.

```
M ~= 'n'
```

```
ans = 1x11 logical array

   1   1   1   1   1   1   1   1   1   1   1
```

The value of logical `1` (`true`) indicates the absence of the character `'n'`. The character is not present in the vector.

Find Values in Categorical Array

Create a categorical array with two values: `'heads'` and `'tails'`.

```
A = categorical({'heads' 'heads' 'tails'; 'tails' 'heads' 'tails'})
```

```
A = 2x3 categorical array
     heads      heads      tails
     tails      heads      tails
```

Find all values not in the `'heads'` category.

```
A ~= 'heads'
```

```
ans = 2x3 logical array

   0   0   1
```

```
   1   0   1
```

A value of logical `1` (`true`) indicates a value not in the category. Since `A` only has two categories, `A ~= 'heads'` returns the same answer as `A == 'tails'`.

Compare the rows of `A` for inequality.

```
A(1,:) ~= A(2,:)
```

```
ans = 1x3 logical array

   1   0   0
```

A value of logical `1` (`true`) indicates where the rows have unequal category values.

Compare Floating-Point Numbers

Many numbers expressed in decimal text cannot be represented exactly as binary floating numbers. This leads to small differences in results that the ~= operator reflects.

Perform a few subtraction operations on numbers expressed in decimal and store the result in `C`.

```
C = 0.5-0.4-0.1
```

```
C = -2.7756e-17
```

With exact decimal arithmetic, `C` should be equal to *exactly* `0`. Its small value is due to the nature of binary floating-point arithmetic.

Compare `C` to `0` for inequality.

```
C ~= 0
```

```
ans = logical
   1
```

Compare floating-point numbers using a tolerance, `tol`, instead of using ~=.

```
tol = eps(0.5);
abs(C-0) > tol
```

```
ans = logical
   0
```

The two numbers, `C` and `0`, are closer to one another than two consecutive floating-point numbers near `0.5`. In many situations, `C` may act like `0`.

Inequality of Two Datetime Arrays

Compare the elements of two `datetime` arrays for inequality.

Create two `datetime` arrays in different time zones.

```
t1 = [2014,04,14,9,0,0;2014,04,14,10,0,0];
A = datetime(t1,'TimeZone','America/Los_Angeles');
A.Format = 'd-MMM-y HH:mm:ss Z'
```

```
A = 2x1 datetime array
   14-Apr-2014 09:00:00 -0700
   14-Apr-2014 10:00:00 -0700
```

```
t2 = [2014,04,14,12,0,0;2014,04,14,12,30,0];
B = datetime(t2,'TimeZone','America/New_York');
B.Format = 'd-MMM-y HH:mm:ss Z'
```

```
B = 2x1 datetime array
   14-Apr-2014 12:00:00 -0400
   14-Apr-2014 12:30:00 -0400
```

Check where elements in `A` and `B` are not equal.

```
A~=B
```

```
ans = 2x1 logical array

   0
   1
```

Input Arguments

A, B — Operands
scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Numeric or string inputs A and B must either be the same size or have sizes that are compatible (for example, A is an M-by-N matrix and B is a scalar or 1-by-N row vector). For more information, see “Compatible Array Sizes for Basic Operations”.

You can compare numeric inputs of any type, and the comparison does not suffer loss of precision due to type conversion.

If A and B are categorical, datetime, or duration arrays, then they must be the same size unless one is a scalar.

- If one input is a categorical array, the other input can be a categorical array, a cell array of character vectors, or a single character vector. A single character vector expands into a cell array of character vectors of the same size as the other input. If both inputs are ordinal categorical arrays, they must have the same sets of categories, including their order. If both inputs are categorical arrays that are not ordinal, they can have different sets of categories. See “Compare Categorical Array Elements” for more details.
- If one input is a datetime array, the other input can be a datetime array, a character vector, or a cell array of character vectors.
- If one input is a duration array, the other input can be a duration array or a numeric array. The operator treats each numeric value as a number of standard 24-hour days.
- If one input is a string array, the other input can be a string array, a character vector, or a cell array of character vectors. The corresponding elements of A and B are compared lexicographically.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char | string | categorical | datetime | duration
Complex Number Support: Yes

Compatibility Considerations

Implicit expansion change affects arguments for operators

Behavior changed in R2016b

Starting in R2016b with the addition of implicit expansion, some combinations of arguments for basic operations that previously returned errors now produce results. For example, you previously could not add a row and a column vector, but those operands are now valid for addition. In other words, an expression like `[1 2] + [1; 2]` previously returned a size mismatch error, but now it executes.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a `try/catch` block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see "Compatible Array Sizes for Basic Operations".

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see "Tall Arrays".

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support using `ne` to test inequality between an enumeration member and a string array, a character array, or a cell array of character arrays.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Code generation does not support using `ne` to test inequality between an enumeration member and a string array, a character array, or a cell array of character arrays.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`eq` | `ge` | `gt` | `le` | `lt`

Topics

“Array Comparison with Relational Operators”
“MATLAB Operators and Special Characters”

Introduced before R2006a

nearestNeighbor

Class: `DelaunayTri`

(Not recommended) Point closest to specified location

Note `nearestNeighbor(DelaunayTri)` is not recommended. Use `nearestNeighbor(triangulation)` instead.

`DelaunayTri` is not recommended. Use `delaunayTriangulation` instead.

Syntax

```
PI = nearestNeighbor(DT,QX)
PI = nearestNeighbor(DT,QX,QY)
PI = nearestNeighbor(DT,QX,QY,QZ)
[PI,D] = nearestNeighbor(DT,QX,...)
```

Description

`PI = nearestNeighbor(DT,QX)` returns the index of the nearest point in `DT.X` for each query point location in `QX`.

`PI = nearestNeighbor(DT,QX,QY)` and `PI = nearestNeighbor(DT,QX,QY,QZ)` allow the query points to be specified in column vector format when working in 2-D and 3-D.

`[PI,D] = nearestNeighbor(DT,QX,...)` returns the index of the nearest point in `DT.X` for each query point location in `QX`. The corresponding Euclidean distances between the query points and their nearest neighbors are returned in `D`.

Note `nearestNeighbor` is not supported for 2-D triangulations that have constrained edges.

Input Arguments

DT	Delaunay triangulation.
QX	The matrix QX is of size mpts-by-ndim, mpts being the number of query points and ndim the dimension of the space where the points reside.

Output Arguments

PI	PI is a column vector of point indices that index into the points DT.X. The length of PI is equal to the number of query points mpts
D	D is a column vector of length mpts.

Examples

Create a Delaunay triangulation:

```
x = rand(10,1);
y = rand(10,1);
dt = DelaunayTri(x,y);
```

Create query points:

```
qrypts = [0.25 0.25; 0.5 0.5];
```

Find the nearest neighbors to the query points:

```
pid = nearestNeighbor(dt, qrypts)
```

See Also

delaunayTriangulation | pointLocation | triangulation

neighbors

Class: `TriRep`

(Not recommended) Simplex neighbor information

Note `neighbors(TriRep)` is not recommended. Use `neighbors(triangulation)` instead.

`TriRep` is not recommended. Use `triangulation` instead.

Syntax

```
SN = neighbors(TR, SI)
```

Description

`SN = neighbors(TR, SI)` returns the simplex neighbor information for the specified simplices `SI`.

Input Arguments

`TR`	Triangulation representation.
`SI`	`SI` is a column vector of simplex indices that index into the triangulation matrix `TR.Triangulation`. If `SI` is not specified the neighbor information for the entire triangulation is returned, where the neighbors associated with simplex `i` are defined by the `i`'th row of `SN`.

Output Arguments

`SN`	`SN` is an `m`-by-`n` matrix, where `m = length(SI)` , the number of specified simplices, and `n` is the number of neighbors per simplex. Each row `SN(i,:)` represents the neighbors of the simplex `SI(i)`. By convention, the simplex opposite `vertex(j)` of simplex `SI(i)` is `SN(i,j)`. If a simplex has one or more boundary facets, the nonexistent neighbors are represented by `NaN`.

Examples

Example 1

Load a 3-D triangulation and use `TriRep` to compute the neighbors of all tetrahedra.

```
load tetmesh
trep = TriRep(tet, X)
nbrs = neighbors(trep)
```

Example 2

Query a 2-D triangulation created using `DelaunayTri`.

```
x = rand(10,1)
y = rand(10,1)
dt = DelaunayTri(x,y)
```

Find the neighbors of the first triangle:

```
n1 = neighbors(dt, 1)
```

More About

Simplex

A simplex is a triangle/tetrahedron or higher-dimensional equivalent.

Facet

A facet is an edge of a triangle or a face of a tetrahedron.

See Also

`delaunayTriangulation` | `triangulation`

NET

Summary of functions in MATLAB .NET interface

Description

Use the following functions to bring assemblies from the Microsoft .NET Framework into the MATLAB environment. The functions are implemented as a package called NET. To use these functions, prefix the function name with package name NET.

BeginInvoke	Initiate asynchronous .NET delegate call
Combine	Convenience function for static .NET System.Delegate Combine method
enableNETfromNetworkDrive	Enable access to .NET commands from network drive
EndInvoke	Retrieve result of asynchronous call initiated by .NET System.Delegate BeginInvoke method
NET.addAssembly	Make .NET assembly visible to MATLAB
NET.Assembly	Members of .NET assembly
NET.convertArray	Convert numeric MATLAB array to .NET array
NET.createArray	Array for nonprimitive .NET types
NET.createGeneric	Create instance of specialized .NET generic type
NET.disableAutoRelease	Lock .NET object representing a RunTime Callable Wrapper (COM Wrapper) so that MATLAB does not release COM object
NET.enableAutoRelease	Unlock .NET object representing a RunTime Callable Wrapper (COM Wrapper) so that MATLAB releases COM object
NET.GenericClass	Represent parameterized generic type definitions
NET.invokeGenericMethod	Invoke generic method of object
NET.isNETSupported	Check for supported Microsoft .NET Framework
NET.NetException	Capture error information for .NET exception
NET.setStaticProperty	Static property or field name
Remove	Convenience function for static .NET System.Delegate Remove method
RemoveAll	Convenience function for static .NET System.Delegate RemoveAll method

See Also

Topics

".NET Libraries"

NET.addAssembly

Package: NET

Make .NET assembly visible to MATLAB

Syntax

```
asmInfo = NET.addAssembly(globalName)
asmInfo = NET.addAssembly(privateName)
```

Description

`asmInfo = NET.addAssembly(globalName)` loads a global .NET assembly into MATLAB.

`asmInfo = NET.addAssembly(privateName)` loads a private .NET assembly.

Limitations

- `NET.addAssembly` does not support assemblies generated by the MATLAB Compiler SDK product.

Input Arguments

`globalName`

One of the following:

- String or character vector representing the name of a global assembly.
- Instance of `System.Reflection.AssemblyName` class.

Default:

privateName

String or character vector representing the full path of a private assembly.

Default:

Output Arguments

asmInfo

`NET.Assembly` object containing names of the members of the assembly.

Examples

Display today's date using `System.DateTime` in the `mscorlib` assembly.

```
System.DateTime.Now.ToLongDateString
```

Call the `System.Windows.Forms.MessageBox.Show` method in the global assembly `System.Windows.Forms`.

```
asm = NET.addAssembly('System.Windows.Forms');
import System.Windows.Forms.*
MessageBox.Show('Simple Message Box')
```

Display classes in the private assembly `NetSample.dll`.

```
asm = NET.addAssembly('c:\work\NetSample.dll');
asm.Classes
```

Tips

- You do not need to call `NET.addAssembly` to access classes in the `mscorlib.dll` and `system.dll` assemblies. MATLAB dynamically loads these assemblies from the .NET Framework class library the first time you type "`NET.`" or "`System.`".
- Refer to your .NET product documentation for the name of the assembly and its deployment type (global or private).

See Also

`NET.Assembly`

Topics

"Assembly Is Library of .NET Classes"

External Websites

MSDN AssemblyName Class

Introduced in R2009a

NET.Assembly class

Package: NET

Members of .NET assembly

Description

NET.Assembly object returns names of the members of an assembly.

Construction

The NET.addAssembly function creates an instance of this class.

Properties

AssemblyHandle

Instance of System.Reflection.Assembly class of the added assembly.

Classes

nClassx1 cell array of class names of the assembly, where nClass is the number of classes.

Enums

nEnumx1 cell array of enumerations of the assembly, where nEnum is the number of enumerations.

Structures

nStructx1 cell array of structures of the assembly, where nStruct is the number of structures.

`GenericTypes`

`nGenTypex1` cell array of generic types of the assembly, where `nGenType` is the number of generic types.

`Interfaces`

`nInterfacex1` cell array of interface names of the assembly, where `nInterface` is the number of interfaces.

`Delegates`

`nDelegatex1` cell array of delegates of the assembly, where `nDelegate` is the number of delegates.

See Also

`NET.addAssembly`

Topics

"What Classes Are in a .NET Assembly?"

NET.convertArray

Package: NET

Convert numeric MATLAB array to .NET array

Note MATLAB automatically converts arrays to .NET types. For information, see "Use Arrays with .NET Applications".

Syntax

```
arrObj = NET.convertArray(V,'arrType',[m,n])
```

Description

`arrObj = NET.convertArray(V,'arrType',[m,n])` converts a MATLAB array `V` to a .NET array. Optional value `arrType` is a string or character vector representing a namespace-qualified .NET array type. To convert a MATLAB vector to a two-dimensional .NET array (either 1-by-n or m-by-1), use optional values `m,n`. If `V` is a MATLAB vector and you do not specify the number of dimensions and their sizes, the output `arrObj` is a one-dimensional .NET array.

If you do not specify `arrType`, MATLAB converts the type according to the MATLAB Primitive Type Conversion Table. See "Pass Primitive .NET Types".

Examples

Create a list `L` of random `System.Int32` integers using the `System.Collections.Generic.List` class, and then sort the results.

```
% Create array R of random integers
nInt = 5;
R = randi(100,1,nInt);
```

```
% Create .NET array A
A = NET.convertArray(R,'System.Int32');
% Put A into L, a generic collections list
L = NET.createGeneric('System.Collections.Generic.List',{'System.Int32'},A.Length);
L.AddRange(A)
% Sort the values in L
L.Sort
```

See Also

NET.createArray

Introduced in R2009a

NET.createArray

Package: NET

Array for nonprimitive .NET types

Syntax

```
array = NET.createArray(typeName,[m,n,p,...])
array = NET.createArray(typeName,m,n,p,...)
```

Description

`array = NET.createArray(typeName,[m,n,p,...])` creates an m-by-n-by-p-by-... array of type `typeName`, which is either a fully qualified .NET array type name (`namespace` and array type name) or an instance of the `NET.GenericClass` class, in case of arrays of generic type. `m,n,p,...` are the number of elements in each dimension of the array.

`array = NET.createArray(typeName,m,n,p,...)` alternative syntax for creating an array.

You cannot specify the lower bound of an array.

Examples

Create .NET Array of Generic Type

This example creates a .NET array of `List<Int32>` generic type.

```
genType = NET.GenericClass('System.Collections.Generic.List',...
  'System.Int32');
arr = NET.createArray(genType, 5)
```

```
arr =
```

```
  List<System*Int32>[] with properties:

            Length: 5
        LongLength: 5
              Rank: 1
          SyncRoot: [1x1 System.Collections.Generic.List<System*Int32>[]]
        IsReadOnly: 0
       IsFixedSize: 1
    IsSynchronized: 0
```

Create and Initialize Jagged Array

This example creates a jagged .NET array of three elements.

```
jaggedArray = NET.createArray('System.Double[]', 3)
```

```
jaggedArray =

  Double[][] with properties

            Length: 3
        LongLength: 3
              Rank: 1
          SyncRoot: [1x1 System.Double[][]]
        IsReadOnly: 0
       IsFixedSize: 1
    IsSynchronized: 0
```

Assign values:

```
jaggedArray(1) = [1, 3, 5, 7, 9];
jaggedArray(2) = [0, 2, 4, 6];
jaggedArray(3) = [11, 22];
```

Access first value of third array:

```
jaggedArray(3,1)
```

```
ans =
    11
```

Create Jagged Array of Generic Type

This example creates a jagged array of `List<Double>` generic type.

```
genCls = NET.GenericClass('System.Collections.Generic.List[]',...
  'System.Double');
```

Create the array, genArr.

```
genArr = NET.createArray(genCls,3)
```

```
genArr =

  List<System*Double>[][] with properties:

           Length: 3
       LongLength: 3
             Rank: 1
         SyncRoot: [1x1 System.Collections.Generic.List`1[][]]
       IsReadOnly: 0
      IsFixedSize: 1
    IsSynchronized: 0
```

Create Nested Jagged Array

This command creates a jagged array of type System.Double[][][].

```
netArr = NET.createArray('System.Double[][]', 3)
```

```
netArr =

  Double[][][] with properties:

           Length: 3
       LongLength: 3
             Rank: 1
         SyncRoot: [1x1 System.Double[][][]]
       IsReadOnly: 0
      IsFixedSize: 1
    IsSynchronized: 0
```

See Also

NET.convertArray | NET.createGeneric

Introduced in R2009a

NET.createGeneric

Package: NET

Create instance of specialized .NET generic type

Syntax

```
genObj = createGeneric(className,paramTypes,varargin ctorArgs)
```

Description

`genObj = createGeneric(className,paramTypes,varargin ctorArgs)` creates an instance `genObj` of generic type `className`.

Input Arguments

`className`	String or character vector containing the fully qualified generic type name.
`paramTypes`	Allowed cell types are: strings or character vectors with fully qualified parameter type names and instances of the `NET.GenericClass` class when parameterization with another parameterized type is needed.
`ctorArgs`	Optional, variable length (0 to N) list of constructor arguments matching the arguments of the .NET generic class constructor intended to be invoked.

Output Arguments

`genObj`	Handle to the specialized generic class instance.

Examples

Create List of System.Double Objects

Create a strongly typed list dblLst of objects of type System.Double.

```
t = NET.createGeneric('System.Collections.Generic.List',{'System.Double'},10);
```

Create List with Key/Value Pairs

Create the kvpType generic association where Key is of System.Int32 type and Value is a System.String.

```
kvpType = NET.GenericClass(...
    'System.Collections.Generic.KeyValuePair',...
    'System.Int32','System.String');
```

Create the list kvpList with initial storage capacity for 10 key-value pairs.

```
kvpList = NET.createGeneric('System.Collections.Generic.List',{kvpType},10);
```

Add Item to List

Create a KeyValuePair item.

```
kvpItem = NET.createGeneric(...
    'System.Collections.Generic.KeyValuePair',...
    {'System.Int32','System.String'},...
    42,'myString');
```

Add this item to the list kvpList.

```
Add(kvpList,kvpItem)
```

See Also

NET.GenericClass

Introduced in R2009a

NET.disableAutoRelease

Package: NET

Lock .NET object representing a RunTime Callable Wrapper (COM Wrapper) so that MATLAB does not release COM object

Syntax

```
A = NET.disableAutoRelease(obj)
```

Description

`A = NET.disableAutoRelease(obj)` locks a .NET object representing a RunTime Callable Wrapper (COM Wrapper) so that MATLAB does not release the COM object. `obj` is a .NET object representing a COM Wrapper.

Before passing a .NET object representing a COM Wrapper to another process, lock the object using this function so that MATLAB does not release it. After using the object, call `NET.enableAutoRelease` to release the COM object.

Examples

The following user-defined function, `GetComApp.m`, has access to a COM object defined in the **pseudo-class**`ComNamespace.ComClass`. One of its methods is `readData`, with the signature:

```
System.String RetVal readData(ComNamespace.ComClass this, System.String strIn)
```

The input argument is defined in the **pseudo-class** `NetDocTest.MyClass`, which has a property named `MyApp`.

```
function GetComApp(obj)
comObj = ComNamespace.ComClass;
obj.MyApp = comObj;
% To pass a COM object to another process, lock the object
```

```
NET.disableAutoRelease(comObj);
end
```

The example in `NET.enableAutoRelease` shows how to call the `GetComApp` function.

See Also

`NET.enableAutoRelease`

Topics

"How MATLAB Handles System.__ComObject"

Introduced in R2010b

NET.enableAutoRelease

Package: NET

Unlock .NET object representing a RunTime Callable Wrapper (COM Wrapper) so that MATLAB releases COM object

Syntax

```
A = NET.enableAutoRelease(obj)
```

Description

`A = NET.enableAutoRelease(obj)` releases the COM wrapper when the object goes out of scope, where `obj` is a .NET object representing a COM Wrapper.

Call this function only if the object was locked using `NET.disableAutoRelease.`

Examples

The following **pseudo-code** shows how to call a function (`GetComApp.m`, described in `NET.disableAutoRelease`) which returns a COM object. The object, `mainObj` of type `NetDocTest.MyClass`, has a property, `MyApp`. Call `GetComApp` to get a COM object, and use its `readData` method.

```
mainObj = NetDocTest.MyClass;
GetComApp(mainObj);
app = mainObj.MyApp;
app.readData('hello');
% Unlock the COM object
NET.enableAutoRelease(mainObj.MyApp);
```

See Also

`NET.disableAutoRelease`

Topics

"How MATLAB Handles System.__ComObject"

Introduced in R2010b

NET.GenericClass class

Package: NET

Represent parameterized generic type definitions

Description

The `NET.createGeneric` function uses instances of this class to create a generic specialization that requires parameterization with another parameterized type.

Construction

```
genType = NET.GenericClass(className,paramTypes)
```

Input Arguments

`className`

String or character vector containing the fully qualified generic type name.

Default:

`paramTypes`

Optional, variable length (1 to N) list of types for the generic class parameterization. Allowed argument types are:

- String or character vector containing the fully qualified generic type name.
- Instance of the `NET.GenericClass` class when deeper nested parameterization with another parameterized type is needed.

Default:

Examples

Create an instance of `System.Collections.Generic.List` of `System.Collections.Generic.KeyValuePair` generic associations where `Key` is of `System.Int32` type and `Value` is a `System.String` class with initial storage capacity for 10 key-value pairs.

```
kvpType = NET.GenericClass(...
    'System.Collections.Generic.KeyValuePair',...
    'System.Int32', 'System.String');
kvpList = NET.createGeneric('System.Collections.Generic.List',...
    {kvpType}, 10);
```

See Also

`NET.createArray` | `NET.createGeneric` | `NET.invokeGenericMethod`

Topics

".NET Generic Classes"

Introduced in R2009a

NET.invokeGenericMethod

Package: NET

Invoke generic method of object

Syntax

```
[varargout] =
NET.invokeGenericMethod(obj,'genericMethodName',paramTypes,args,...)
```

Description

`[varargout] = NET.invokeGenericMethod(obj,'genericMethodName',paramTypes,args,...)` calls instance or static generic method *genericMethodName*.

Input Arguments

`obj`	Allowed argument types are: • Instances of class containing the generic method • Strings or character vectors with fully qualified class name, if calling static generic methods • Instances of `NET.GenericClass` definitions, if calling static generic methods of a generic class
genericMethodName	Generic method name to invoke

`paramTypes`	Cell vector (1 to N) with the types for generic method parameterization, where allowed cell types are: • Strings or character vectors with fully qualified parameter type name. • Instances of `NET.GenericClass` definitions, if using nested parameterization with another parameterized type
`args`	Optional, variable length (0 to N) list of method arguments

Output Arguments

`varargout`	Variable-length output argument list, `varargout`, from method *`genericMethodName`*

Examples

The following syntax calls a generic method that takes two parameterized types and returns a parameterized type:

```
a = NET.invokeGenericMethod(obj, ...
  'myGenericSwapMethod', ...
  {'System.Double', 'System.Double'}, ...
  5, 6);
```

To display generic methods in MATLAB, see the example “Display .NET Generic Methods Using Reflection”.

See Also

`NET.GenericClass` | `NET.createGeneric` | `varargout`

Topics

“Call .NET Generic Methods”

Introduced in R2009b

NET.isNETSupported

Check for supported Microsoft .NET Framework

Syntax

```
tf = NET.isNETSupported
```

Description

`tf = NET.isNETSupported` returns logical `1` (`true`) if a supported version of the Microsoft .NET Framework is found. Otherwise, it returns logical `0` (`false`) and you cannot use the .NET Framework in MATLAB.

Examples

Display Message If Supported Microsoft .NET Framework Not Found

```
if ~NET.isNETSupported
    disp('Supported .NET Framework not found')
end
```

See Also

Introduced in R2013a

NET.NetException class

Package: NET

Capture error information for .NET exception

Description

Process information from a `NET.NetException` object to handle .NET errors. This class is derived from `MException`.

Construction

`e = NET.NetException(msgID,errMsg,netObj)` constructs instance `e` of `NET.NetException` class.

Input Arguments

`msgID`

message identifier

`errMsg`

error message text

`netObj`

`System.Exception` object that caused the exception

Properties

`ExceptionObject`

`System.Exception` class causing the error.

Methods

Inherited Methods

See the methods of the base class `MException`.

Examples

Display error information after trying to load an unknown assembly:

```
try
  NET.addAssembly('C:\Work\invalidfile.dll')
catch e
  e.message;
  if(isa(e, 'NET.NetException'))
    eObj = e.ExceptionObject
  end
end
```

```
ans =
Message: Could not load file or assembly
  'file:///C:\Work\invalidfile.dll' or
  one of its dependencies. The system cannot
  find the file specified.
Source: mscorlib
HelpLink:

eObj =
  FileNotFoundException with properties:

           Message: [1x1 System.String]
          FileName: [1x1 System.String]
         FusionLog: [1x1 System.String]
              Data: [1x1 System.Collections.ListDictionaryInternal]
    InnerException: []
        TargetSite: [1x1 System.Reflection.RuntimeMethodInfo]
        StackTrace: [1x1 System.String]
          HelpLink: []
            Source: [1x1 System.String]
```

See Also

`MException`

Topics

Class Attributes
Property Attributes

Introduced in R2009b

NET.setStaticProperty

Package: NET

Static property or field name

Syntax

```
NET.setStaticProperty('propName', value)
```

Description

`NET.setStaticProperty('propName', value)` sets the static property or field name specified in the string or character vector `propName` to the given `value`.

Examples

To set the `myStaticProperty` in the given class and namespace, use the syntax:

```
NET.setStaticProperty('MyTestObject.MyClass.myStaticProperty', 5);
```

Introduced in R2009b

nccreate

Create variable in NetCDF file

Syntax

```
nccreate(filename,varname)
nccreate(filename,varname,Name,Value)
```

Description

`nccreate(filename,varname)` creates a scalar `double` variable named `varname` in the NetCDF file specified by `filename`. If `filename` does not exist, then `nccreate` creates the file using the `netcdf4_classic` format.

`nccreate(filename,varname,Name,Value)` creates a variable with additional options specified by one or more name-value pair arguments. For example, to create a nonscalar variable, use the `Dimensions` name-value pair argument.

Examples

Create New Variables in NetCDF File

Create a NetCDF file named `myexample.nc` that contains a variable named `Var1`.

```
nccreate('myexample.nc','Var1')
```

Create a second variable in the same file.

```
nccreate('myexample.nc','Var2')
```

Display the contents of the NetCDF file.

```
ncdisp('myexample.nc')
```

```
Source:
           pwd\myexample.nc
```

```
Format:
           netcdf4_classic
Variables:
    Var1
           Size:       1x1
           Dimensions:
           Datatype:   double
    Var2
           Size:       1x1
           Dimensions:
           Datatype:   double
```

Create Variable and Specify Dimensions and File Format

Create a new two-dimensional variable named `peaks` in a classic (NetCDF 3) format file named `myncclassic.nc`. Use the `'Dimensions'` name-value pair argument to specify the names and lengths of the two dimensions. Use the `'Format'` name-value pair argument to specify the file format.

```
nccreate('myncclassic.nc','peaks',...
          'Dimensions',{'r',200,'c',200},...
          'Format','classic')
```

Write data to the variable.

```
ncwrite('myncclassic.nc','peaks',peaks(200))
```

Display the contents of the NetCDF file.

```
ncdisp('myncclassic.nc')
```

```
Source:
           pwd\myncclassic.nc
Format:
           classic
Dimensions:
           r = 200
           c = 200
Variables:
    peaks
           Size:       200x200
```

```
Dimensions: r,c
Datatype:   double
```

Input Arguments

`filename` — File name

character vector | string scalar

File name, specified as a character vector or string scalar. The file is an existing NetCDF file, or the name you want to assign to a new NetCDF file.

Example: `'myFile.nc'`

`varname` — Name of new variable

character vector | string scalar

Name of the new variable, specified as a character vectorr or string scalar.

Example: `'myVar'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
`nccreate('myFile.nc','Var1','Datatype','double','Format','classic')` creates a variable named `Var1` of type `NC_DOUBLE` in a NetCDF 3 file named `myFile.nc`.

`Dimensions` — Dimensions of variable

cell array

Dimensions of the new variable, specified as the comma-separated pair consisting of `'Dimensions'` and a cell array. The cell array lists the dimension name as a character vector or string scalar followed by its numerical length, in this form: `{dname1,dlength1,dname2,dlength2, ...}`. The `dname1` input is the name of the first dimension specified as a character vector or string scalar, `dlength1` is the length of the first dimension, `dname2` is the name of the second dimension, and so on. If a dimension exists, specifying its length is optional. A variable with a single dimension is always treated as a column vector.

Use `Inf` to specify an unlimited dimension. A `netcdf4` format file can have any number of unlimited dimensions in any order. All other formats can have only one unlimited dimension per file and it must be specified last in the cell array.

`nccreate` creates the dimension at the same location as the variable. For `netcdf4` format files, you can specify a different location for the dimension using a fully qualified dimension name.

Example: `'Dimensions',{'dim1',100,'dim2',150,'dim3',Inf}`

`Datatype` — MATLAB data type
`'double'` (default) | character vector | string scalar

MATLAB data type, specified as the comma-separated pair consisting of `'Datatype'` and a character vector or string scalar containing the name of the data type. When `nccreate` creates the variable in the NetCDF file, it uses a corresponding NetCDF datatype. This table lists valid values for `'Datatype'` and the corresponding NetCDF variable type that `nccreate` creates.

Value of `Datatype`	NetCDF Variable Type
`'double'`	`NC_DOUBLE`
`'single'`	`NC_FLOAT`
`'int64'`	`NC_INT64*`
`'uint64'`	`NC_UINT64*`
`'int32'`	`NC_INT`
`'uint32'`	`NC_UINT*`
`'int16'`	`NC_SHORT`
`'uint16'`	`NC_USHORT*`
`'int8'`	`NC_BYTE`
`'uint8'`	`NC_UBYTE*`
`'char'`	`NC_CHAR`

* These data types are only available when the file is a `netcdf4` format file.

Example: `'Datatype','uint16'`

`Format` — NetCDF file format
`'netcdf4_classic'` (default) | character vector | string scalar

NetCDF file format, specified as the comma-separated pair consisting of `'Format'` and one of these values.

Value of `Format`	Description
`'classic'`	NetCDF 3
`'64bit'`	NetCDF 3, with 64-bit offsets
`'netcdf4_classic'`	NetCDF 4 classic model
`'netcdf4'`	NetCDF 4 model (Use this format to enable group hierarchy)

If `varname` specifies a group (for example,`'/grid3/temperature'`), then `nccreate` sets the value of `Format` to `'netcdf4'`.

Example: `'Format','classic'`

`FillValue` — Replacement value for missing values
scalar | `'disable'`

Replacement value for missing values, specified as the comma-separated pair consisting of `'FillValue'` and a scalar or `'disable'`. The default value is specified by the NetCDF library. To disable replacement values, specify `'FillValue','disable'`.

This argument is available for `netcdf4` or `netcdf4_classic` formats only.

Example: `'FillValue',NaN`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char`

`ChunkSize` — Chunk size along each dimension
vector

Chunk size along each dimension, specified as the comma-separated pair consisting of `'ChunkSize'` and a vector. The first element specifies the number of rows, the second element specifies the number of columns, the third element specifies the length of the third dimension, and so on. The default value is specified by the NetCDF library.

This argument is available for `netcdf4` or `netcdf4_classic` formats only.

Example: `'ChunkSize',[5 6 9]`

Data Types: `double`

`DeflateLevel` — Amount of compression
`0` (default) | scalar value between `0` and `9`

Amount of compression, specified as the comma-separated pair consisting of `'DeflateLevel'` and a scalar value between `0` and `9`. `0` indicates no compression and `9` indicates the most compression.

This argument is available for `netcdf4` or `netcdf4_classic` formats only.

Example: `'DeflateLevel',5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Shuffle` — Status of shuffle filter
`false` (default) | `true`

Status of the shuffle filter, specified as the comma-separated pair consisting of `'Shuffle'` and `false` or `true`. `false` disables the shuffle filter and `true` enables it. The shuffle filter can assist with the compression of integer data by changing the byte order in the data stream.

This argument is available for `netcdf4` or `netcdf4_classic` formats only.

Example: `'Shuffle',true`

Data Types: `logical`

See Also
`ncdisp` | `ncinfo` | `ncwrite` | `ncwriteschema`

Topics
"Export to NetCDF Files"

Introduced in R2011a

ncdisp

Display contents of NetCDF data source in Command Window

Syntax

```
ncdisp(source)
ncdisp(source,location)
ncdisp(source,location,dispFormat)
```

Description

`ncdisp(source)` displays all the groups, dimensions, variable definitions, and all attributes in the NetCDF data source, `source`, as text in the Command Window.

`ncdisp(source,location)` displays information about the variable or group specified by `location`.

`ncdisp(source,location,dispFormat)` displays the contents of the NetCDF data source, in the display format specified by `dispFormat`.

Examples

Display Contents of NetCDF File

Display the contents of the example NetCDF file, `example.nc`.

```
ncdisp('example.nc')
```

```
Source:
           matlabroot\toolbox\matlab\demos\example.nc
Format:
           netcdf4
Global Attributes:
           creation_date = '29-Mar-2010'
```

```
Dimensions:
           x = 50
           y = 50
           z = 5
Variables:
    avagadros_number
           Size:       1x1
           Dimensions:
           Datatype:   double
           Attributes:
                       description = 'this variable has no dimensions'
    temperature
           Size:       50x1
           Dimensions: x
           Datatype:   int16
           Attributes:
                       scale_factor = 1.8
                       add_offset   = 32
                       units        = 'degrees_fahrenheight'
    peaks
           Size:       50x50
           Dimensions: x,y
           Datatype:   int16
           Attributes:
                       description = 'z = peaks(50);'
Groups:
    /grid1/
        Attributes:
                   description = 'This is a group attribute.'
        Dimensions:
                   x    = 360
                   y    = 180
                   time = 0     (UNLIMITED)
        Variables:
            temp
                   Size:       []
                   Dimensions: x,y,time
                   Datatype:   int16

    /grid2/
        Attributes:
                   description = 'This is another group attribute.'
        Dimensions:
                   x    = 360
```

```
                    y    = 180
                    time = 0     (UNLIMITED)
        Variables:
            temp
                    Size:       []
                    Dimensions: x,y,time
                    Datatype:   int16
```

MATLAB displays all the groups, dimensions, and variable definitions in `example.nc`.

Display Contents of NetCDF Variable

Display the contents of the variable `peaks` in the file, `example.nc`.

```
ncdisp('example.nc','peaks')
```

```
Source:
           matlabroot\toolbox\matlab\demos\example.nc
Format:
           netcdf4
Dimensions:
           x = 50
           y = 50
Variables:
    peaks
           Size:       50x50
           Dimensions: x,y
           Datatype:   int16
           Attributes:
                       description = 'z = peaks(50);'
```

Display Contents of NetCDF File and Hide Attributes

Display only the group hierarchy and variable definitions of the example file, `example.nc`.

```
ncdisp('example.nc','/','min')
```

```
Source:
           matlabroot\toolbox\matlab\demos\example.nc
Format:
```

```
           netcdf4
Variables:
    avagadros_number
           Size:       1x1
           Dimensions:
           Datatype:   double
    temperature
           Size:       50x1
           Dimensions: x
           Datatype:   int16
    peaks
           Size:       50x50
           Dimensions: x,y
           Datatype:   int16
Groups:
    /grid1/
        Variables:
            temp
                   Size:       []
                   Dimensions: x,y,time
                   Datatype:   int16

    /grid2/
        Variables:
            temp
                   Size:       []
                   Dimensions: x,y,time
                   Datatype:   int16
```

Input Arguments

`source` — Name of NetCDF file

character vector | string scalar

Name of a NetCDF file, specified as a character vector or string scalar. `source` also can be the URL of an OPeNDAP NetCDF data source that resolves to a NetCDF file or a variable in a NetCDF file.

Example: `'myNetCDFfile.nc'`

Data Types: `char` | `string`

`location` — Location of variable or group
`'/'` (default) | character vector | string scalar

Location of a variable or group in the NetCDF file, specified as a character vector or string scalar. Set `location` to `'/'`(forward slash) to display the entire contents of the file.

Data Types: `char` | `string`

`dispFormat` — Display format
`'full'` (default) | `'min'`

Display format, specified as one of these values.

`'full'`	Display group hierarchy with dimensions, attributes, and variable definitions.
`'min'`	Display group hierarchy and variable definitions.

Data Types: `char` | `string`

Tips

- If `source` is an OPeNDAP URL with a constraint expression, use the syntax, `ncdisp(source)` with no other input arguments.

See Also

`ncinfo` | `ncread` | `ncreadatt` | `ncwrite`

Topics

"Import NetCDF Files and OPeNDAP Data"

Introduced in R2011a

ncinfo

Return information about NetCDF data source

Syntax

```
finfo = ncinfo(source)
vinfo = ncinfo(source,varname)
ginfo = ncinfo(source,groupname)
```

Description

`finfo = ncinfo(source)` returns information in the structure `finfo` about the entire NetCDF data source specified by `source`, where source can be the name of a NetCDF file or the URL of an OPeNDAP NetCDF data source.

`vinfo = ncinfo(source,varname)` returns information in the structure `vinfo` about the variable `varname` in `source`.

`ginfo = ncinfo(source,groupname)` returns information in the structure `ginfo` about the group `groupname` in `source` (only NetCDF4 data sources).

Note Use `ncdisp` for visual inspection of a NetCDF source.

Input Arguments

`source`

Character vector or string scalar specifying the name of a NetCDF file or the URL of an OPeNDAP NetCDF data source.

Default:

`varname`

Character vector or string scalar specifying the name of a variable in a NetCDF file or OPeNDAP data source.

Default:

`groupname`

Character vector or string scalar specifying the name of a group in a NetCDF file or OPeNDAP data source.

Default:

Output Arguments

`finfo`

A structure with the following fields.

Field	Description	
`Filename`	NetCDF file name or OPeNDAP URL	
`Name`	`'/'`, indicating the full file	
`Dimensions`	An array of structures with these fields:	
	`Name`	Dimension name
	`Length`	Current length of dimension
	`Unlimited`	Boolean flag, true for unlimited dimensions
`Variables`	An array of structures with these fields:	
	`Name`	Variable name
	`Dimensions`	Associated dimensions
	`Size`	Current variable size
	`Datatype`	MATLAB datatype
	`Attributes`	Associated variable attributes

Field	Description	
	`ChunkSize`	Chunk size, if defined. `[]` otherwise
	`FillValue`	Fill value of the variable.
	`DeflateLevel`	Deflate filter level, if enabled.
	`Shuffle`	Shuffle filter enabled flag
`Attributes`	An array of global attributes with these fields:	
	`Name`	Attribute name
	`Value`	Attribute value
`Groups`	An array of groups present in the file, for `netcdf4` files; An empty array (`[]`) for all other NetCDF file formats.	
`Format`	The format of the NetCDF file	

`vinfo`

A structure containing only the variable fields from `finfo`.

Field	Description
`Filename`	NetCDF file name
`Name`	Name of the variable
`Dimensions`	Dimensions of the variable
`Size`	Size of the current variable
`Datatype`	MATLAB datatype
`Attributes`	Attributes associated with the variable
`ChunkSize`	Chunk size, if defined. `[]` otherwise.
`FillValue`	Fill value used in the variable.
`DeflateLevel`	Deflate filter level, if enabled.
`Shuffle`	Shuffle filter enabled flag
`Format`	The format of the NetCDF file

`ginfo`

A structure containing only the group fields from `finfo`.

Field	Description
`Filename`	NetCDF file name
`Name`	Name of the group
`Dimensions`	Only dimensions defined in the specified group
`Variables`	Only variables defined in the specified group
`Attributes`	Attributes associated with the variable
`Groups`	Names of groups, if defined. `[]` otherwise.
`Format`	The format of the NetCDF file

Examples

Search for dimensions with names that start with the character `x` in the file.

```
finfo = ncinfo('example.nc');
disp(finfo);
dimNames = {finfo.Dimensions.Name};
dimMatch = strncmpi(dimNames,'x',1);
disp(finfo.Dimensions(dimMatch));
```

Obtain the size of a variable and check if it has any unlimited dimensions.

```
vinfo = ncinfo('example.nc','peaks');
varSize = vinfo.Size;
disp(vinfo);
hasUnLimDim = any([vinfo.Dimensions.Unlimited]);
```

Find all unlimited dimensions defined in a group.

```
ginfo = ncinfo('example.nc','/grid2/');
unlimDims = [ginfo.Dimensions.Unlimited];
disp(ginfo.Dimensions(unlimDims));
```

See Also

`ncdisp` | `ncread` | `ncwrite` | `ncwriteschema`

Topics

“Import NetCDF Files and OPeNDAP Data”

Introduced in R2011a

ncread

Read data from variable in NetCDF data source

Syntax

```
vardata = ncread(source,varname)
vardata = ncread(source,varname,start,count)
vardata = ncread(source,varname,start,count,stride)
```

Description

`vardata = ncread(source,varname)` reads all the data from the variable `varname` contained in the NetCDF file or an OPeNDAP NetCDF data source specified by `source`.

`vardata = ncread(source,varname,start,count)` reads data beginning at the location specified in `start`. The `count` argument specifies the number of elements to read along each dimension.

`vardata = ncread(source,varname,start,count,stride)` returns data with the interval between the indices of each dimension of the variable specified by `stride`.

Examples

Read NetCDF Data

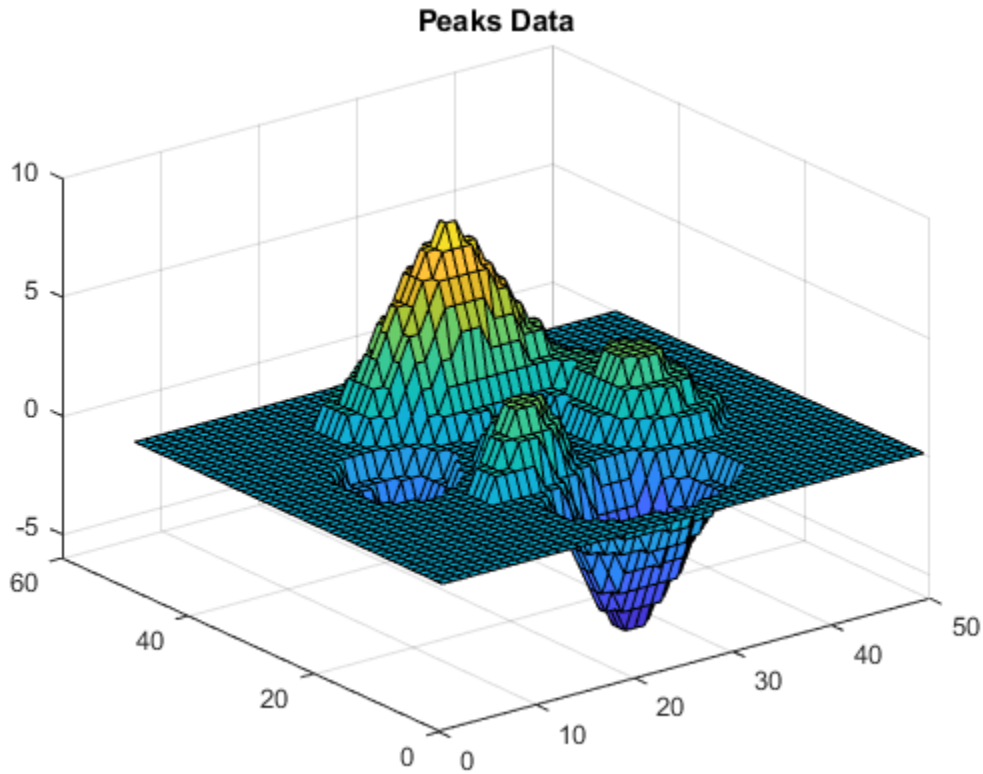
Read and plot variable named `peaks` from the file `example.nc`.

```
peaksData  = ncread('example.nc','peaks');
whos peaksData
```

```
  Name            Size            Bytes  Class    Attributes

  peaksData      50x50             5000  int16
```

Plot `peaksData` and add a title.

```
surf(double(peaksData));
title('Peaks Data');
```



Read Portion of Data from Variable

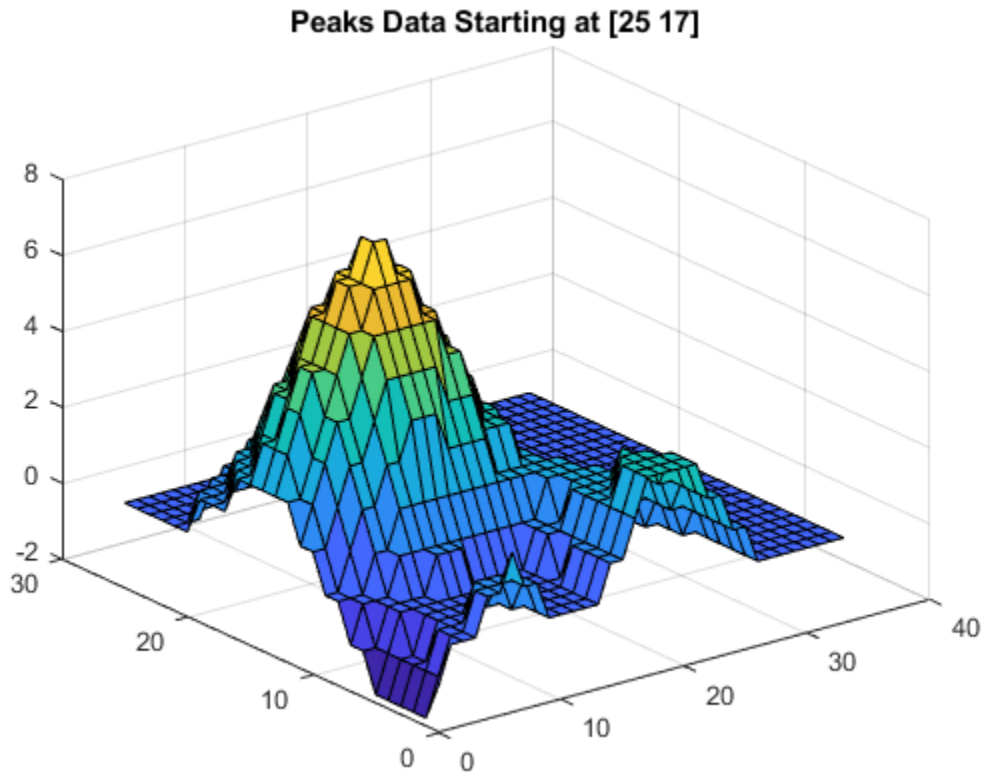
Read and plot only a subset of the variable data starting from the location `[25 17]` until the end of each dimension.

```
startLoc = [25 17]; % Start location along each coordinate
count  = [Inf Inf]; % Read until the end of each dimension
peaksData  = ncread('example.nc','peaks',startLoc,count);
whos peaksData
```

```
  Name            Size            Bytes  Class    Attributes

  peaksData      26x34             1768  int16
```

Plot the data.

```
surf(double(peaksData));
title('Peaks Data Starting at [25 17]');
```



Read Data with Specified Spacing Between Variable Indices

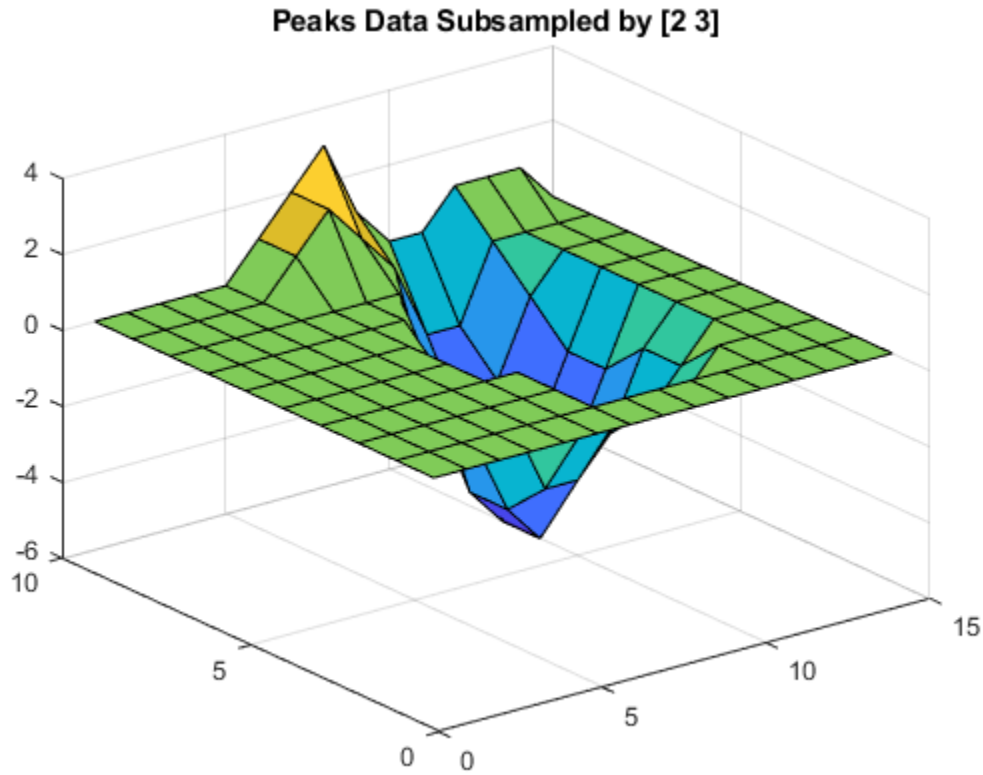
Read and plot data, where the data is sampled at a specified spacing between variable indices along each dimension. Start reading from the location in `startLoc` and read variable data at intervals specified in `stride`. A value of `1` in `stride`, accesses adjacent values in the corresponding dimension. Whereas, a value of `2` accesses every other value in the corresponding dimension, and so on.

```
startLoc = [1 1];
count  = [10 15];
stride = [2 3];
sampledPeaksData  = ncread('example.nc','peaks',startLoc,count,stride);
whos sampledPeaksData
```

```
  Name                   Size            Bytes  Class    Attributes

  sampledPeaksData      10x15              300  int16
```

Plot the data.

```
surf(double(sampledPeaksData));
title('Peaks Data Subsampled by [2 3]');
```



Input Arguments

`source` — Source name
character vector | string scalar

Source name, specified as a character vector or string scalar containing the name of a NetCDF file or the URL of an OPeNDAP NetCDF data source.

Data Types: `char` | `string`

`varname` — Variable name
character vector | string scalar

Variable name, specified as a character vector or string scalar containing the name of a variable in the NetCDF file or OPeNDAP NetCDF data source.

Data Types: `char` | `string`

`start` — Starting location
vector of ones (default) | numeric vector

Starting location, specified as a numeric vector of positive integers. For an `N`-dimensional variable, `start` is a vector of length *N* containing 1-based indices.

If you do not specify `start`, then the `ncread` function starts reading the variable from the first index along each dimension.

Data Types: `double`

`count` — Number of elements
vector of `Inf`'s (default) | numeric vector

Number of elements to read, specified as a numeric vector of positive integers. For an `N`-dimensional variable, `count` is a vector of length `N`, specifying the number of elements to read along each dimension. If any element of `count` is `Inf`, then `ncread` reads until the end of the corresponding dimension.

If you do not specify `count`, then the `ncread` function reads the variable data until end of each dimension.

Data Types: `double`

`stride` — Space between variable indices
vector of ones (default) | numeric vector

Space between the variable indices along each dimension, specified as a numeric vector of integers. For an `N`-dimensional variable, `stride` is vector of length `N`. The elements of the `stride` vector correspond, in order, to the variable's dimensions. A value of `1` accesses adjacent values of the NetCDF variable in the corresponding dimension. Whereas, a value of `2` accesses every other value of the NetCDF variable in the corresponding dimension, and so on.

If you do not specify `stride`, then the `ncread` function reads the data with a default spacing of `1` along each dimension.

Data Types: `double`

Output Arguments

`vardata` — Variable data
numeric | text

Variable data, returned as text or numeric arrays.

In most cases, the `ncread` function uses the MATLAB datatype that is the closest type to the corresponding NetCDF datatype.

When at least one of the variable attributes `_FillValue`, `scale_factor`, or `add_offset` is present, then `ncread` returns `vardata` of type `double`. In addition, `ncread` applies these conventions:

- If the `_FillValue` attribute exists, then `ncread` replaces `vardata` values equal to `_FillValue` values with `NaN`s. If the`_FillValue` attribute does not exist, then `ncread` queries the NetCDF library for the variable's fill value.
- If the `scale_factor` attribute exists, then `ncread` multiplies variable data by the value of the `scale_factor` attribute.
- If the `add_offset` attribute exists, then `ncread` adds the value of the `add_offset` attribute to the variable data.

Note For variable data containing text, the `ncread` function supports reading only of `vardata` that is ASCII encoded.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`

See Also

`ncdisp` | `ncinfo` | `ncreadatt` | `ncwrite`

Topics

“Import NetCDF Files and OPeNDAP Data”

Introduced in R2011a

ncreadatt

Read attribute value from NetCDF data source

Syntax

```
attvalue = ncreadatt(source,location,attname)
```

Description

`attvalue = ncreadatt(source,location,attname)` reads the attribute `attname` from the group or variable specified by `location` in `source`, where source is the name of a NetCDF file or the URL of a NetCDF data source.

Input Arguments

`source`

Character vector or string scalar specifying the name of a NetCDF file or the URL of an OPeNDAP NetCDF data source.

`location`

Character vector or string scalar specifying a group or variable in the NetCDF data source. To read global attributes, set `location` to `'/'` (forward slash).

`attname`

Character vector or string scalar specifying the name of an attribute that you want to read in the NetCDF data source.

Output Arguments

attvalue

Data associated with the attribute.

Examples

Read Global Attribute

```
creation_date = ncreadatt('example.nc','/','creation_date')
```

```
creation_date =
'29-Mar-2010'
```

Read Attribute Associated with Variable

Read an attribute associated with a variable.

```
scale_factor = ncreadatt('example.nc','temperature','scale_factor')
```

```
scale_factor = 1.8000
```

Read Attribute Associated with Group

Read an attribute associated with a group. This example applies to netcdf4 format files only.

```
desc_value = ncreadatt('example.nc','/grid2','description')
```

```
desc_value =
'This is another group attribute.'
```

See Also

`ncdisp` | `ncinfo` | `ncread` | `ncwriteatt`

Topics

"Import NetCDF Files and OPeNDAP Data"

Introduced in R2011a

ncwrite

Write data to NetCDF file

Syntax

```
ncwrite(filename,varname,vardata)
ncwrite(filename,varname,vardata,start)
ncwrite(filename,varname,vardata,start,stride)
```

Description

`ncwrite(filename,varname,vardata)` writes the text or numeric data in `vardata` to an existing variable `varname` in the NetCDF file `filename`.

The `ncwrite` function writes the data in `vardata` starting at the beginning of the variable and, if needed, automatically extends the unlimited dimensions. For more information on unlimited dimensions, see the `Dimensions` argument of the `nccreate` function.

`ncwrite(filename,varname,vardata,start)` writes data to an existing variable, beginning at the location specified by `start`. Use this syntax to append data to an existing variable or to write partial data.

`ncwrite(filename,varname,vardata,start,stride)` writes data with the interval between the indices of each dimension specified by `stride`.

Examples

Write Variable to NetCDF File

Create a NetCDF file that contains a variable, and then write data to that variable.

Create a new file `myfile.nc` containing a variable named `pi`.

```
nccreate('myfile.nc','pi');
```

Write a scalar data with no dimensions to the variable `pi`.

```
ncwrite('myfile.nc','pi',3.1416);
```

Read and display the variable from the file.

```
valPi = ncread('myfile.nc','pi')
```

```
valPi = 3.1416
```

Write Data to Portion of Variable

Write data to a portion of a variable in a NetCDF file starting at a specified location.

Create a file `myncfile.nc` with an empty 3-by-6 numeric variable `vmark`. To disable the default fill value for missing or empty variables, set the value of the `FillValue` name-value pair argument to `disable`.

```
nccreate('myncfile.nc','vmark',...
         'Dimensions', {'x',3,'y',6},...
         'FillValue','disable');
```

Write a 3-by-3 array to the variable, and then read and display `vmark` from the file. The `ncwrite` function writes data starting at the beginning of the variable.

```
ncwrite('myncfile.nc','vmark',3*eye(3));
varData = ncread('myncfile.nc','vmark');
display(varData)
```

```
varData = 3×6

     3     0     0     0     0     0
     0     3     0     0     0     0
     0     0     3     0     0     0
```

Add another 3-by-3 array to the variable `vmark` starting at the fourth column of the first row. Next, read and display `vmark` from the file. The `ncrwrite` function writes the array starting at the location `[1 4]`.

```
ncwrite('myncfile.nc','vmark',5*eye(3),[1 4]);
varData = ncread('myncfile.nc','vmark');
display(varData)
```

```
varData = 3×6

     3     0     0     5     0     0
     0     3     0     0     5     0
     0     0     3     0     0     5
```

Write Data with Specified Spacing

Write data with specified spacing between the variable indices along each dimension.

First, create a file `myncfile.nc` with an empty 6-by-6 numeric variable `vmark`. To disable the default fill value for missing or empty variables, set the value of the `FillValue` name-value pair argument to `disable`.

```
nccreate('myncfile.nc','vmark',...
         'Dimensions', {'x',6,'y',6},...
         'FillValue','disable');
```

Next, write a 3-by-3 numeric array to the variable `vmark` starting at the location `[1 1]` with a spacing of `2` between the variable indices along each dimension. Read and display `vmark` from the file.

```
ncwrite('myncfile.nc','vmark',3*eye(3),[1 1],[2 2]);
varData = ncread('myncfile.nc','vmark');
display(varData)
```

```
varData = 6×6

     3     0     0     0     0     0
     0     0     0     0     0     0
     0     0     3     0     0     0
     0     0     0     0     0     0
     0     0     0     0     3     0
     0     0     0     0     0     0
```

Input Arguments

`filename` — File name
character vector | string scalar

File name of an existing NetCDF file, specified as a character vector or string scalar.

If the NetCDF file or variable do not exist, then use the `nccreate` function to create them first.

Data Types: `char` | `string`

`varname` — Variable name
character vector | string scalar

Variable name, specified as a character vector or string containing the name of a variable in the NetCDF file.

Data Types: `char` | `string`

`vardata` — Variable data
numeric | text

Variable data, specified as numeric data or text.

Note For variable values containing text data, the `vardata` input must have only ASCII encoded characters.

If the variable `varname` exists, then `ncwrite` expects the data type of `vardata` to match the NetCDF variable data type.

If the variable `varname` has attributes `_FillValue`, `scale_factor`, or `add_offset`, then the `ncwrite` function expects the data to be of the data type `double`. To cast `vardata` into the NetCDF data type, the `ncwrite` function applies these attribute conventions in a sequence:

1. Subtract the value of the `add_offset` attribute from `vardata`.
2. Divide `vardata` by the value of the `scale_factor` attribute.
3. Replace any `NaN` in `vardata` with the value contained in the `_FillValue` attribute. If this attribute does not exist, then `ncwrite` uses the fill value for this variable as specified by the NetCDF library.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`

`start` — Starting location
vector of ones (default) | numeric vector

Starting location of the data in the variable, specified as a vector of indices. For an `N`-dimensional variable, `start` is a vector of length *N* containing 1-based indices.

If you do not specify `start`, then the `ncwrite` function starts writing the variable from the first index along each dimension.

Data Types: `double`

`stride` — Space between variable indices
vector of ones (default) | numeric vector

Space between the variable indices along each dimension, specified as a numeric vector of integers. For an `N`-dimensional variable, `stride` is a vector of length `N`. The elements of the `stride` vector correspond, in order, to the variable's dimensions. A value of `1` writes adjacent values of the NetCDF variable in the corresponding dimension. Where as, a value of `2` writes every other value of the NetCDF variable in the corresponding dimension, and so on.

If you do not specify `stride`, then the `ncwrite` function writes the data with a default spacing of `1` along each dimension.

Data Types: `double`

See Also

`nccreate` | `ncdisp` | `ncinfo` | `ncread` | `ncwriteatt`

Topics

"Export to NetCDF Files"

Introduced in R2011a

ncwriteatt

Write attribute to NetCDF file

Syntax

```
ncwriteatt(filename,location,attname,attvalue)
```

Description

`ncwriteatt(filename,location,attname,attvalue)` creates or modifies the attribute specified by `attname` in the group or variable specified by `location`, in the NetCDF file specified by `filename`. `attvalue` can be a a character vector or string scalar, or a numeric vector.

Input Arguments

`filename`

Character vector or string scalar specifying the name of a NetCDF file

`location`

Character vector or string scalar specifying a group or variable in the NetCDF file. To write global attributes, set `location` to `'/'` (forward slash).

`attname`

Character vector or string scalar specifying the name of an existing attribute in a NetCDF file or the name of the attribute that you want to create.

`attvalue`

Character vector or string scalar, or a numeric vector.

Examples

Create a global attribute.

```
copyfile(which('example.nc'),'myfile.nc');
fileattrib('myfile.nc','+w');
ncdisp('myfile.nc');
ncwriteatt('myfile.nc','/','creation_date',datestr(now));
ncdisp('myfile.nc');
```

Modify an existing attribute.

```
copyfile(which('example.nc'),'myfile.nc');
fileattrib('myfile.nc','+w');
ncdisp('myfile.nc','peaks');
ncwriteatt('myfile.nc','peaks','description','Output of PEAKS');
ncdisp('myfile.nc','peaks');
```

See Also

`nccreate` | `ncdisp` | `ncread` | `ncreadatt` | `ncwrite`

Topics

"Export to NetCDF Files"

Introduced in R2011a

ncwriteschema

Add NetCDF schema definitions to NetCDF file

Syntax

```
ncwriteschema(filename,schema)
```

Description

`ncwriteschema(filename,schema)` creates or adds attributes, dimensions, variable definitions and group structure defined in `schema` to the file `filename`.

Use `ncwriteschema` in combination with `ncinfo` to create a new NetCDF file based on the schema of an existing file. You can also use `ncwriteschema` to add variable definitions, attributes, dimensions, or group structure to an existing file.

Note `ncwriteschema` does not write variable data. Use `ncwrite` to write data to the created variables. Created unlimited dimensions will have an initial size of 0 until you write data.

Note `ncwriteschema` cannot change the format of an existing file. It cannot redefine existing variables and dimensions in `filename`. If your schema contains attributes, dimensions, variable definitions, or a group structure that already exist in the file, `writeschema` issues a warning but continues processing.

Input Arguments

`filename`

Character vector or string scalar containing the name of a NetCDF file. If `filename` does not exist, `ncwriteschema` creates a new file using the `netcdf4_classic` format, unless the `Format` field in `schema` specifies another format.

Default:

`schema`

A structure, or array of structures, representing either a dimension, variable, an entire NetCDF file, or a `netcdf4` group. A group or file schema can contain a dimension or variable schema, or both. You can use the output returned by `ncinfo` as a `schema` structure. The following table lists the fields in the various types of schema structures. Optional fields are marked with asterisk (*).

Schema Type	Structure Field	Description
Group/File Schema	`Name`	Character vector or string scalar identifying the group name. Use `'/'` to indicate the entire file.
	`Dimensions*`	Dimension schema
	`Variables*`	Variable schema
	`Attributes*`	Structure array of group/global attributes with Name and Value fields
	`Format*`	Character vector or string scalar identifying a NetCDF file format
Dimension schema	`Name`	Character vector or string scalar identifying the dimension
	`Length`	Length of the dimension. Can be `Inf`.
	`Unlimited*`	Boolean flag indicating if the dimension is unlimited
	`Format*`	Character vector or string scalar identifying a NetCDF file format
Variable schema	`Name`	Character vector or string scalar identifying a variable name
	`Dimensions`	Variable's dimension schema
	`Datatype`	Character vector or string scalar identifying a MATLAB datatype
	`Attributes*`	Structure array of variable attributes with Name and Value fields

Schema Type	Structure Field	Description
	`ChunkSize*`	Numeric value specifying chunk size of the variable
	`FillValue*`	Text or numeric fill value
	`DeflateValue*`	Deflate compression level
	`Shuffle*`	Boolean flag to turn on the Shuffle filter
	`Format*`	Character vector or string scalar identifying a NetCDF file format

Default:

Examples

Create a classic format file with two dimension definitions.

```
mySchema.Name   = '/';
mySchema.Format = 'classic';
mySchema.Dimensions(1).Name   = 'time';
mySchema.Dimensions(1).Length = Inf;
mySchema.Dimensions(2).Name   = 'rows';
mySchema.Dimensions(2).Length = 10;
ncwriteschema('emptyFile.nc', mySchema);
ncdisp('emptyFile.nc');
```

Create a `netcdf4_classic` format file to store a single variable from an existing file. First use `ncinfo` to get the schema of the peaks variable from the file. Then use `ncwriteschema` to create a NetCDF file, defining the peaks variable. Use `ncread` to get the data associated with the peaks variable and then use `ncwrite` to write the data to the variable in the new NetCDF file.

```
myVarSchema = ncinfo('example.nc','peaks');
ncwriteschema('peaksFile.nc',myVarSchema);
peaksData   = ncread('example.nc','peaks');
ncwrite('peaksFile.nc','peaks',peaksData);
ncdisp('peaksFile.nc');
```

See Also

`ncdisp` | `ncinfo` | `ncread` | `ncwrite`

Topics

"Export to NetCDF Files"

Introduced in R2011a

netcdf.abort

Revert recent netCDF file definitions

Syntax

```
netcdf.abort(ncid)
```

Description

`netcdf.abort(ncid)` reverts a netCDF file to its previous state, backing out any definitions made since the file last entered define mode. A file enters define mode when you create it (using `netcdf.create`) or when you explicitly enter define mode (using `netcdf.redef`). Once you leave define mode (using `netcdf.endDef`), you cannot revert the definitions you made while in define mode. `ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`. A call to `netcdf.abort` closes the file.

This function corresponds to the `nc_abort` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm.

Examples

This example creates a new file, performs an operation on the file, and then reverts the file back to its original state. To run this example, you must have write permission in your current directory.

```
% Create a netCDF file
ncid = netcdf.create('foo.nc','NC_NOCLOBBER');

% Perform an operation, such as defining a dimension.
dimid = netcdf.defDim(ncid, 'lat', 50);

% Revert the file back to its previous state.
netcdf.abort(ncid)

% Verify that the file is now closed.
```

```
dimid = netcdf.defDim(ncid, 'lat', 50); % should fail
??? Error using ==> netcdflib
NetCDF: Not a valid ID

Error in ==> defDim at 22
dimid = netcdflib('def_dim', ncid,dimname,dimlen);
```

See Also

netcdf.create | netcdf.endDef | netcdf.reDef

netcdf.close

Close netCDF file

Syntax

```
netcdf.close(ncid)
```

Description

`netcdf.close(ncid)` terminates access to the netCDF file identified by `ncid`.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_close` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm.

Examples

This example creates a new netCDF file, and then closes the file. You must have write permission in your current directory to run this example.

```
ncid = netcdf.open('foo.nc','NC_WRITE')

netcdf.close(ncid)
```

See Also

`netCDF.open` | `netcdf.create`

netcdf.copyAtt

Copy attribute to new location

Syntax

```
netcdf.copyAtt(ncid_in,varid_in,attname,ncid_out,varid_out)
```

Description

`netcdf.copyAtt(ncid_in,varid_in,attname,ncid_out,varid_out)` copies an attribute from one variable to another, possibly across files. `ncid_in` and `ncid_out` are netCDF file identifiers returned by `netcdf.create` or `netcdf.open`. `varid_in` identifies the variable with an attribute that you want to copy. `varid_out` identifies the variable to which you want to associate a copy of the attribute.

This function corresponds to the `nc_copy_att` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm.

Examples

This example makes a copy of the attribute associated with the first variable in the netCDF example file, `example.nc`, in a new file. To run this example, you must have write permission in your current directory.

```
% Open example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get identifier for a variable in the file.
varid = netcdf.inqVarID(ncid,'avagadros_number');

% Create new netCDF file.
ncid2 = netcdf.create('foo.nc','NC_NOCLOBBER');

% Define a dimension in the new file.
dimid2 = netcdf.defDim(ncid2,'x',50);
```

```
% Define a variable in the new file.
varid2 = netcdf.defVar(ncid2,'myvar','double',dimid2);

% Copy the attribute named 'description' into the new file,
% associating the attribute with the new variable.
netcdf.copyAtt(ncid,varid,'description',ncid2,varid2);
%
% Check the name of the attribute in new file.
attname = netcdf.inqAttName(ncid2,varid2,0)

attname =

description
```

See Also

netcdf.inqAtt | netcdf.inqAttID | netcdf.inqAttName | netcdf.putAtt | netcdf.renameAtt

netcdf.create

Create new NetCDF dataset

Syntax

```
ncid = netcdf.create(filename,cmode)
[chunksize_out,ncid] =
netcdf.create(filename,cmode,initsz,chunksize)
```

Description

`ncid = netcdf.create(filename,cmode)` creates a new NetCDF file according to the file creation mode. The return value `ncid` is a file ID. The `cmode` parameter determines the type of file access. Specify `cmode` as one of these values.

Value of cmode	Description
`'NOCLOBBER'`	Prevent overwriting of existing file with the same name.
`'CLOBBER'`	Overwrite any existing file with the same name.
`'SHARE'`	Allow synchronous file updates.
`'64BIT_OFFSET'`	Allow easier creation of files and variables which are larger than two gigabytes.
`'NETCDF4'`	Create a NetCDF-4/HDF5 file
`'CLASSIC_MODEL'`	Enforce the classic model; has no effect unless used in a bitwise-or with `NETCDF4`

Note You can specify the mode as a numeric value, retrieved using the `netcdf.getConstant` function. To specify more than one mode, use a bitwise-OR of the numeric values of the modes.

`[chunksize_out,ncid] = netcdf.create(filename,cmode,initsz,chunksize)` creates a new NetCDF file,

but with additional performance tuning parameters. `initsz` sets the initial size of the file. `chunksize` can affect I/O performance. The actual value chosen by the NetCDF library might not correspond to the input value.

This function corresponds to the `nc_create` and `nc__create` functions in the NetCDF library C API. To use this function, you should be familiar with the NetCDF programming paradigm.

Examples

Create NetCDF File Without Overwriting Existing File

Create a NetCDF dataset named `foo.nc`, only if no other file with the same name exists in the current directory. To run this example, you must have write permission in your current directory.

```
ncid = netcdf.create('foo.nc','NOCLOBBER')
```

```
ncid = 65536
```

`netcdf.create` returns a file identifier.

Close the file

```
netcdf.close(ncid)
```

Create NetCDF-4 File Using Classic Model

Get the numeric values corresponding to the `NETCDF4` and `CLASSIC_MODEL` constants defined by the NetCDF library. Use a bitwise-OR of the numeric values to specify more than one creation mode.

```
cmode = netcdf.getConstant('NETCDF4');
cmode = bitor(cmode,netcdf.getConstant('CLASSIC_MODEL'));
```

Create a NetCDF-4 file that uses the classic model by specifying the creation mode value, `cmode`.

```
ncid = netcdf.create('myfile.nc',cmode);
```

Close the file.

```
netcdf.close(ncid);
```

See Also

`netcdf.getConstant` | `netcdf.open`

netcdf.defDim

Create netCDF dimension

Syntax

```
dimid = netcdf.defDim(ncid,dimname,dimlen)
```

Description

`dimid = netcdf.defDim(ncid,dimname,dimlen)` creates a new dimension in the netCDF file specified by `ncid`, where `dimname` is a character vector or string scalar that specifies the name of the dimension and `dimlen` is a numeric value that specifies its length. To define an unlimited dimension, specify the predefined constant `'NC_UNLIMITED'` for `dimlen`, using `netcdf.getConstant` to retrieve the value.

`netcdf.defDim` returns `dimid`, a numeric ID corresponding to the new dimension.

This function corresponds to the `nc_def_dim` function in the netCDF library C API.To use this function, you should be familiar with the netCDF programming paradigm.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

Examples

Create a new file and define two dimensions in the file. One dimension is an unlimited dimension. To run this example, you must have write permission in your current folder.

```
% Create a netCDF file.
ncid = netcdf.create('foo.nc','NC_NOCLOBBER')

% Define a dimension.
lat_dimID = netcdf.defDim(ncid,'latitude',360);

% Define an unlimited dimension.
```

```
long_dimID = netcdf.defDim(ncid,'longitude',...
        netcdf.getConstant('NC_UNLIMITED'));
```

See Also

netcdf.getConstant

netcdf.defGrp

Create group in NetCDF file

Syntax

```
childGrpID = netcdf.defGrp(parentGroupId,childGroupName)
```

Description

`childGrpID = netcdf.defGrp(parentGroupId,childGroupName)` creates a child group with the name specified by `childGroupName`, that is the child of the parent group specified by `parentGroupId`

Input Arguments

`parentGroupId`

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

`childGroupName`

Character vector or string scalar specifying the name that you want to assign to the group.

Default:

Output Arguments

`childGrpID`

Identifier of a NetCDF group.

Examples

This example creates a NetCDF dataset and then defines a group.

```
ncid = netcdf.create('myfile.nc','netcdf4');
childGroupId = netcdf.defGrp(ncid,'mygroup');
netcdf.close(ncid);
```

References

This function corresponds to the `nc_def_grp` function in the NetCDF library C API.

For copyright information, read the files `netcdfcopyright.txt` and `mexnccopyright.txt`.

See Also

`netcdf.inqGrps`

netcdf.defVar

Create NetCDF variable

Syntax

```
varid = netcdf.defVar(ncid,varname,xtype,dimids)
```

Description

`varid = netcdf.defVar(ncid,varname,xtype,dimids)` creates a new variable in the dataset identified by `ncid`.

`varname` is a character vector or string scalar that specifies the name of the variable.

`xtype` specifies the NetCDF data type of the variable, using one of these values.

Value of `xtype`	MATLAB Class
`'NC_BYTE'`	`int8` or `uint8`[a]
`'NC_CHAR'`	`char`
`'NC_SHORT'`	`int16`
`'NC_INT'`	`int32`
`'NC_FLOAT'`	`single`
`'NC_DOUBLE'`	`double`

a. NetCDF interprets byte data as either signed or unsigned.

Alternatively, `xtype` can be the numeric equivalent returned by the `netcdf.getConstant` function.

`dimids` specifies a list of dimension IDs.

`netcdf.defVar` returns `varid`, a numeric identifier for the new variable.

This function corresponds to the `nc_def_var` function in the NetCDF library C API. Because MATLAB uses FORTRAN-style ordering, the fastest-varying dimension comes

first and the slowest comes last. Any unlimited dimension is therefore last in the list of dimension IDs. This ordering is the reverse of that found in the C API.To use this function, you should be familiar with the NetCDF programming paradigm.

Examples

Define Dimension and Variable in NetCDF File

Create a new NetCDF file, define a dimension in the file, and then define a variable on that dimension. In NetCDF files, you must create a dimension before you can create a variable. To run this example, you must have write permission in your current folder.

Create a new NetCDF file named `foo.nc`.

```
ncid = netcdf.create('foo.nc','NC_NOCLOBBER');
```

Define a dimension in the new file.

```
dimid = netcdf.defDim(ncid,'x',50);
```

Define a variable in the new file using `netcdf.defVar`.

```
varid = netcdf.defVar(ncid,'myvar','NC_DOUBLE',dimid)
```

```
varid = 0
```

`netcdf.defVar` returns a numeric identifier for the new variable.

Close the file.

```
netcdf.close(ncid)
```

See Also

`netCDF.getConstant` | `netCDF.inqVar` | `netCDF.putVar`

netcdf.defVarChunking

Define chunking behavior for NetCDF variable

Syntax

```
netcdf.defVarChunking(ncid,varid,storage,chunkDims)
```

Description

`netcdf.defVarChunking(ncid,varid,storage,chunkDims)` sets the chunk settings for the variable specified by `varid`. Chunking is a technique to improve performance. `storage` specifies the type of chunking to use and `chunkDims` specifies the extents of the chunk size. You must specify the chunk size used with a variable after creating the variable but before you write data to the variable.

You cannot specify the chunk size for variables in a NetCDF file created with the netCDF-3 mode (`CLASSIC_MODEL`).

Input Arguments

`ncid`

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

`varid`

Identifier of a NetCDF variable, returned by `netcdf.defVar`.

Default:

`storage`

Character vector or string scalar specifying whether NetCDF should break the variable into chunks when writing to a file. If set to `'CHUNKED'`, NetCDF breaks the variable into chunks; if set to `'CONTIGUOUS'`, NetCDF does not break the data into chunks.

Default:

`chunkDims`

Array specifying the dimensions of the chunk.

Because MATLAB uses FORTRAN-style ordering, the order of dimensions in `chunkdims` is reversed relative to what would be in the C API.

If storage is `'CONTIGUOUS'`, you can omit `chunkDims`.

Default: Chunk size determined by the NetCDF library.

Examples

This example creates a NetCDF file and specifies the chunking behavior of a variable.

```
ncid = netcdf.create('myfile.nc','NETCDF4');
latdimid = netcdf.defDim(ncid,'lat',1800);
londimid = netcdf.defDim(ncid,'col',3600);
varid = netcdf.defVar(ncid,'earthgrid','double',[latdimid londimid]);
netcdf.defVarChunking(ncid,varid,'CHUNKED',[180 360]);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_def_var_chunking` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf.inqVarChunking`

netcdf.defVarDeflate

Define compression parameters for NetCDF variable

Syntax

```
netcdf.defVarDeflate(ncid,varid,shuffle,deflate,deflateLevel)
```

Description

`netcdf.defVarDeflate(ncid,varid,shuffle,deflate,deflateLevel)` sets the compression parameters for the NetCDF variable specified by `varid` in the location specified by `ncid`.

Input Arguments

`ncid`

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

`varid`

Identifier of a NetCDF variable, returned by `netcdf.defVar`.

Default:

`shuffle`

Boolean value. To turn on the shuffle filter, set this argument to `true`. The shuffle filter can assist with the compression of integer data by changing the byte order in the data stream.

Default:

deflate

Boolean value. To turn on compression, set this argument to `true` and set the `deflateLevel` argument to the desired compression level.

Default:

deflateLevel

Numeric value between `0` and `9` specifying the amount of compression, where `0` is no compression and `9` is the most compression.

Default:

Examples

This example create a variable with dimensions [1800 3600] and a compression level of 5. This results in a chunked layout that is a 10-by-10 grid. Use `netcdf.defVarChunking` to define your own chunking, otherwise `netcdf.defVarDeflate` uses the library default values.

```
ncid = netcdf.create('myfile.nc','NETCDF4');
latdimid = netcdf.defDim(ncid,'lat',1800);
londimid = netcdf.defDim(ncid,'col',3600);
varid = netcdf.defVar(ncid,'earthgrid','double',[latdimid londimid]);
netcdf.defVarChunking(ncid,varid,'CHUNKED',[180 360]);
netcdf.defVarDeflate(ncid,varid,true,true,5);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_def_var_deflate` function in the netCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf.inqVarDeflate`

netcdf.defVarFill

Define fill parameters for NetCDF variable

Syntax

```
netcdf.defVarFill(ncid,varid,noFillMode,fillValue)
```

Description

`netcdf.defVarFill(ncid,varid,noFillMode,fillValue)` sets the fill parameters for a variable in a NetCDF-4 file. `varid` identifies the variable and the `ncid` identifies the NetCDF-4 file.

For netCDF files, you can only specify fill values when the NetCDF is in definition mode (before calling `netcdf.endDef`). For NetCDF files in classic and 64-bit offset modes, you can turn no-fill mode on and off at any time.

Input Arguments

`ncid`

Identifier of a NetCDF-4 file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

`varid`

Identifier of a NetCDF variable, returned by `netcdf.defVar`.

Default:

`noFillMode`

Boolean value. When set to `true`, turns off use of fill values for the variable, which can be helpful in high performance applications. When `true`, `netcdf.defVarFill` ignores the value of the `fillValue` argument. To use the fill value, set this to `false`.

Default:

`fillValue`

Specifies the value to use in the variable when no other value is specified. The data type must be the same data type as the variable.

Default:

Examples

This example creates a NetCDF-4 file and defines a fill value for a variable.

```
ncid = netcdf.create('myfile.nc','NETCDF4');
dimid = netcdf.defDim(ncid,'latitude',180);
varid = netcdf.defVar(ncid,'latitude','double',dimid);
netcdf.defVarFill(ncid,varid,false,-999);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_def_var_fill` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf.inqVarFill` | `netcdf.setFill`

netcdf.defVarFletcher32

Define checksum parameters for NetCDF variable

Syntax

```
netcdf.defVarFletcher32(ncid,varid,setting)
```

Description

`netcdf.defVarFletcher32(ncid,varid,setting)` defines the checksum settings for the NetCDF variable specified by `varid` in the file specified by `ncid`.

Input Arguments

`ncid`

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

`varid`

Identifier of a NetCDF variable, returned by `netcdf.defVar`.

Default:

`setting`

Character vector or string scalar specifying whether Fletcher32 checksum error detection is used with the variable. To turn on Fletcher32 checksum, specify the value `'FLETCHER32'`. To turn off the use of checksum error detection, specify the value `'NOCHECKSUM'`.

Default:

Examples

This example creates a NetCDF dataset and turns on the Fletcher32 checksum for a variable.

```
ncid = netcdf.create('myfile.nc','NETCDF4');
latdimid = netcdf.defDim(ncid,'lat',1800);
londimid = netcdf.defDim(ncid,'col',3600);
varid = netcdf.defVar(ncid,'earthgrid','double',[latdimid londimid]);
netcdf.defVarFletcher32(ncid,varid,'FLETCHER32');
netcdf.close(ncid);
```

References

This function corresponds to the `nc_def_var_fletcher32` function in the NetCDF library C API.

For copyright information, read the files `netcdfcopyright.txt` and `mexnccopyright.txt`.

See Also

`netcdf.inqVarFletcher32`

netcdf.delAtt

Delete netCDF attribute

Syntax

```
netcdf.delAtt(ncid,varid,attName)
```

Description

`netcdf.delAtt(ncid,varid,attName)` deletes the attribute identified by `attName`. Specify `attName` as a character vector or string scalar.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`varid` is a numeric value that identifies the variable. To delete a global attribute, use `netcdf.getConstant('GLOBAL')` for the `varid`. You must be in define mode to delete an attribute.

This function corresponds to the `nc_del_att` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm.

Examples

This example opens a local copy of the example netCDF file included with MATLAB, `example.nc`.

```
% Open a netCDF file.
ncid = netcdf.open('my_example.nc','NC_WRITE')

% Determine number of global attributes in file.
[numdims numvars numatts unlimdimID] = netcdf.inq(ncid);

numatts =

   1
```

```
% Get name of attribute; it is needed for deletion.
attname = netcdf.inqAttName(ncid,netcdf.getConstant('NC_GLOBAL'),0)

% Put file in define mode to delete an attribute.
netcdf.reDef(ncid);

% Delete the global attribute in the netCDF file.
netcdf.delAtt(ncid,netcdf.getConstant('GLOBAL'),attname);

% Verify that the global attribute was deleted.
[numdims numvars numatts unlimdimID] = netcdf.inq(ncid);

numatts =

   0
```

See Also

netcdf.getConstant | netcdf.inqAttName

netcdf.endDef

End netCDF file define mode

Syntax

```
netcdf.endDef(ncid)
netcdf.endDef(ncid,h_minfree,v_align,v_minfree,r_align)
```

Description

`netcdf.endDef(ncid)` takes a netCDF file out of define mode and into data mode. `ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`netcdf.endDef(ncid,h_minfree,v_align,v_minfree,r_align)` takes a netCDF file out of define mode, specifying four additional performance tuning parameters. For example, one reason for using the performance parameters is to reserve extra space in the netCDF file header using the `h_minfree` parameter:

```
ncid = netcdf.endDef(ncid,20000,4,0,4);
```

This reserves 20,000 bytes in the header, which can be used later when adding attributes. This can be extremely efficient when working with very large netCDF 3 files. To understand how to use these performance tuning parameters, see the netCDF library documentation.

This function corresponds to the `nc_enddef` and `nc__enddef` functions in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm.

Examples

Take File out of Define Mode

When you create a file using `netcdf.create`, the functions opens the file in define mode. This example uses `netcdf.endDef` to take the file out of define mode.

Create a netCDF file.

```
ncid = netcdf.create('foo.c','NC_NOCLOBBER');
```

Define a dimension.

```
dimid = netcdf.defDim(ncid,'lat',50);
```

Leave define mode.

```
netcdf.endDef(ncid)
```

Making a change, when not in define mode, will return an error. For this example the define mode was ended, using the `netcdf.endDef` function, therefore typing `dimid = netcdf.defDim(ncid,'lon',50)` in the command window should return an error.

See Also

`netcdf.create` | `netcdf.reDef`

netcdf.getAtt

Return netCDF attribute

Syntax

```
attrvalue = netcdf.getAtt(ncid,varid,attname)
attrvalue = netcdf.getAtt(ncid,varid,attname,output_datatype)
```

Description

`attrvalue = netcdf.getAtt(ncid,varid,attname)` returns `attrvalue`, the value of the attribute name contained in `attname`. Specify `attname` as a character vector or string scalar. When it chooses the data type of `attrvalue`, MATLAB attempts to match the netCDF class of the attribute. For example, if the attribute has the netCDF data type `NC_INT`, MATLAB uses the `int32` class for the output data. If an attribute has the netCDF data type `NC_BYTE`, the class of the output data is `int8` value.

`attrvalue = netcdf.getAtt(ncid,varid,attname,output_datatype)` returns `attrvalue`, the value of the attribute name specified by `attname`, using the output class specified by `output_datatype`. Specify the output data type using one of these values.

`'int'`	`'double'`	`'int16'`
`'short'`	`'single'`	`'int8'`
`'float'`	`'int32'`	`'uint8'`

This function corresponds to several attribute I/O functions in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`, and gets the value of the attribute associated with the first variable. The example also gets the value of the global variable in the file.

```
% Open a netCDF file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get name of first variable.
[varname vartype vardimIDs varatts] = netcdf.inqVar(ncid,0);

% Get ID of variable, given its name.
varid = netcdf.inqVarID(ncid,varname);

% Get attribute name, given variable id.
attname = netcdf.inqAttName(ncid,varid,0);

% Get value of attribute.
attval = netcdf.getAtt(ncid,varid,attname);

% Get name of global attribute
gattname = netcdf.inqAttName(ncid,netcdf.getConstant('NC_GLOBAL'),0);

% Get value of global attribute.
gattval = netcdf.getAtt(ncid,netcdf.getConstant('NC_GLOBAL'),gattname)

gattval =

09-Jun-2008
```

See Also

netcdf.inqAtt | netcdf.putAtt

netcdf.getChunkCache

Retrieve chunk cache settings for NetCDF library

Syntax

```
[csize, nelems, premp] = netcdf.getChunkCache()
```

Description

`[csize, nelems, premp] = netcdf.getChunkCache()` returns the default chunk cache settings.

Output Arguments

`csize`

Scalar double specifying the total size of the raw data chunk cache in bytes.

`nelems`

Scalar double specifying the number of chunk slots in the raw data chunk cache hash table.

`premp`

Double, between 0 and 1, inclusive, that specifies how the library handles preempting fully read chunks in the chunk cache. A value of zero means fully read chunks are treated no differently than other chunks, that is, preemption occurs solely based on the Least Recently Used (LRU) algorithm. A value of 1 means fully read chunks are always preempted before other chunks.

Examples

Determine information about the chunk cache size used by the NetCDF library.

```
[csize, nelems, premp] = netcdf.getChunkCache();
```

References

This function corresponds to the `nc_get_chunk_cache` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf.setChunkCache`

netcdf.getConstant

Return numeric value of named constant

Syntax

```
val = netcdf.getConstant(param_name)
```

Description

`val = netcdf.getConstant(param_name)` returns the numeric value corresponding to the name of a constant defined by the netCDF library. For example, `netcdf.getConstant('NC_NOCLOBBER')` returns the numeric value corresponding to the netCDF constant `NC_NOCLOBBER`.

The value for `param_name` can be either upper- or lowercase, and does not need to include the leading three characters `'NC_'`. To retrieve a list of all the names defined by the netCDF library, use the `netcdf.getConstantNames` function.

This function has no direct equivalent in the netCDF C interface.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`.

```
% Open example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Determine contents of the file.
[ndims nvars natts dimm] = netcdf.inq(ncid);

% Get name of global attribute.
% Note: You must use netcdf.getConstant to specify NC_GLOBAL.
attname = netcdf.inqattname(ncid,netcdf.getConstant('NC_GLOBAL'),0)

attname =
```

```
creation_date
```

See Also

netcdf.getConstantNames

netcdf.getConstantNames

Return list of constants known to netCDF library

Syntax

```
val = netcdf.getConstantNames(param_name)
```

Description

`val = netcdf.getConstantNames(param_name)` returns a list of names of netCDF library constants, definitions, and enumerations. When these names, returned as character vectors, are supplied as actual parameters to MATLAB netCDF package functions, the functions automatically convert the constant to the appropriate numeric value.

This MATLAB function has no direct equivalent in the netCDF C interface.

Examples

```
nc_constants = netcdf.getConstantNames

nc_constants =

    'NC2_ERR'
    'NC_64BIT_OFFSET'
    'NC_BYTE'
    'NC_CHAR'
    'NC_CLOBBER'
    'NC_DOUBLE'
    'NC_EBADDIM'
    'NC_EBADID'
    'NC_EBADNAME'
    'NC_EBADTYPE'
        ...
```

See Also

`netCDF.getConstantNames`

netcdf.getVar

Read data from NetCDF variable

Syntax

```
data = netcdf.getVar(ncid,varid)
data = netcdf.getVar(ncid,varid,start)
data = netcdf.getVar(ncid,varid,start,count)
data = netcdf.getVar(ncid,varid,start,count,stride)
data = netcdf.getVar( ___ ,output_type)
```

Description

`data = netcdf.getVar(ncid,varid)` returns `data`, the value of the variable specified by `varid`. MATLAB attempts to match the class of the output data to NetCDF class of the variable.

Note For variable values containing text, the `netcdf.getVar` function supports reading only of `data` that is ASCII encoded.

`ncid` is a NetCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`data = netcdf.getVar(ncid,varid,start)` returns a single value starting at the specified index, `start`.

`data = netcdf.getVar(ncid,varid,start,count)` returns a contiguous section of a variable. `start` specifies the starting point and `count` specifies the amount of data to return.

`data = netcdf.getVar(ncid,varid,start,count,stride)` returns a subset of a section of a variable. `start` specifies the starting point, `count` specifies the extent of the section, and `stride` specifies which values to return.

`data = netcdf.getVar( ___ ,output_type)` specifies the data type of the return value `data`. Specify `output_type` as one of these values.

'int8'
'uint8'
'int16'
'int32'
'single'
'double'

This function corresponds to several functions in the NetCDF library C API. To use this function, you should be familiar with the NetCDF programming paradigm.

Examples

Read Value of Variable in NetCDF File

Open the example file, `example.nc`.

```
ncid = netcdf.open('example.nc','NC_NOWRITE');
```

Get the name of the first variable in the file.

```
varname = netcdf.inqVar(ncid,0)
```

```
varname =
'avagadros_number'
```

Get variable ID of the first variable, given its name.

```
varid = netcdf.inqVarID(ncid,varname)
```

```
varid = 0
```

Get the value of the variable. Use the variable ID as the second input to the `netcdf.getVar` function.

```
data = netcdf.getVar(ncid,varid)
```

```
data = 6.0221e+23
```

Display the data type of the output value.

```
whos data
```

```
  Name      Size            Bytes  Class     Attributes

  data      1x1                 8  double
```

Get the value of the `avogadros_number` variable again, specifying that the output data type should be `single`.

```
data = netcdf.getVar(ncid,varid,'single');
```

Display the data type of the output value.

```
whos data
```

```
  Name      Size            Bytes  Class     Attributes

  data      1x1                 4  single
```

Close the NetCDF file.

```
netcdf.close(ncid)
```

See Also

`netcdf.create` | `netcdf.inqVarID` | `netcdf.open`

netcdf.inq

Return information about netCDF file

Syntax

```
[ndims,nvars,ngatts,unlimdimid] = netcdf.inq(ncid)
```

Description

`[ndims,nvars,ngatts,unlimdimid] = netcdf.inq(ncid)` returns the number of dimensions, variables, and global attributes in a netCDF file. The function also returns the ID of the dimension defined with unlimited length, if one exists.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`. You can call `netcdf.inq` in either define mode or data mode.

This function corresponds to the `nc_inq` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`, and uses the `netcdf.inq` function to get information about the contents of the file.

```
% Open netCDF example file.
ncid = netcdf.open('example.nc','NC_NOWRITE')

% Get information about the contents of the file.
[numdims, numvars, numglobalatts, unlimdimID] = netcdf.inq(ncid)

numdims =

     4
```

```
numvars =

     4


numglobalatts =

     1


unlimdimID =

     3
```

See Also

netcdf.create | netcdf.open

netcdf.inqDimIDs

Retrieve list of dimension identifiers in group

Syntax

```
dimIDs = netcdf.inqDimIDs(ncid)
dimIDs = netcdf.inqDimIDs(ncid,includeParents)
```

Description

`dimIDs = netcdf.inqDimIDs(ncid)` returns a list of dimension identifiers in the group specified by `ncid`.

`dimIDs = netcdf.inqDimIDs(ncid,includeParents)` includes all dimensions in all parent groups if `includeParents` is `true`.

Input Arguments

`ncid`

Identifier of a NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

`includeParents`

Boolean value. If set to `true`, `netcdf.inqDimIDs` includes the dimensions of all parent groups.

Default: `false`

Output Arguments

dimIDs

Array of dimension IDs

Examples

This example opens the NetCDF sample file and gets the IDs of all the dimensions.

```
ncid = netcdf.open('example.nc','NOWRITE');
gid = netcdf.inqNcid(ncid,'grid1');
dimids = netcdf.inqDimIDs(gid);
dimids_all = netcdf.inqDimIDS(gid, true);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_dimids` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf.inqVarIDs`

netcdf.inqFormat

Determine format of NetCDF file

Syntax

```
format = netcdf.inqFormat(ncid)
```

Description

`format = netcdf.inqFormat(ncid)` returns the format for the file specified by NetCDF file identifier, `ncid`.

Input Arguments

`ncid`

Identifier of a NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

Output Arguments

`format`

Character vector that specifies the format of the NetCDF file. Returned values include:

Format	Description
`'FORMAT_CLASSIC'`	Classic format — Original NetCDF format, used by all NetCDF files created between 1989 and 2004

Format	Description
'FORMAT_64BIT'	Classic format, 64–bit — Original format with 64–bit addressing capability to allow creation and access of much larger files.
'FORMAT_NETCDF4'	Enhanced model, HDF5-based — Introduced in 2008, NetCDF, version 4, extends the classic model and is based on HDF5.
'FORMAT_NETCDF4_CLASSIC'	Classic model, HDF5-based — Introduced in 2008, NetCDF, version 4, implements classic model but is based on HDF5.

Examples

This example opens the sample NetCDF file and determines the format.

```
ncid = netcdf.open('example.nc','NOWRITE');
fmt = netcdf.inqFormat(ncid)

format =

FORMAT_NETCDF4

netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_format` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf.getConstant`

netcdf.inqGrpName

Retrieve name of group

Syntax

```
groupName = netcdf.inqGrpName(ncid)
```

Description

`groupName = netcdf.inqGrpName(ncid)` returns the name of a group specified by `ncid`.

Input Arguments

`ncid`

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

Output Arguments

`groupName`

Character vector containing the name of the group specified by `ncid`. The root group has the name `'/'`.

Examples

This example opens the NetCDF sample file and gets the names of groups in the dataset.

```
ncid = netcdf.open('example.nc','nowrite');
name = netcdf.inqGrpName(ncid);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_grpname` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf.inqGrpNameFull`

netcdf.inqGrpNameFull

Complete pathname of group

Syntax

```
groupName = netcdf.inqGrpNameFull(ncid)
```

Description

`groupName = netcdf.inqGrpNameFull(ncid)` returns the complete pathname of the group specified by `ncid`.

Input Arguments

`ncid`

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

Output Arguments

`groupName`

Character vector specifying complete path of group.

The root group has the name `'/'`. The names of parent groups and child groups use the forward slash `'/'` separator, as in UNIX folder names, for example, `/group1/subgrp2/subsubgrp3`.

Examples

Open the NetCDF sample dataset and retrieve the names of all groups.

```
ncid = netcdf.open('example.nc','NOWRITE');
gid = netcdf.inqNcid(ncid,'grid2');
fullName = netcdf.inqGrpNameFull(gid);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_grpname_full` function in the netCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf.inqGrpName`

netcdf.inqGrpParent

Retrieve ID of parent group.

Syntax

```
parentGroupID = netcdf.inqGrpParent(ncid)
```

Description

`parentGroupID = netcdf.inqGrpParent(ncid)` returns the ID of the parent group given the location of the child group, specified by `ncid`.

Input Arguments

`ncid`

Identifier of a NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

Output Arguments

`parentGroupID`

Identifier of the NetCDF group or file that is the parent of the specified file or group.

Examples

This example opens the NetCDF sample file and gets the full path of the parent of the specified group.

```
ncid = netcdf.open('example.nc','NOWRITE');
gid = netcdf.inqNcid(ncid,'grid2');
parentId = netcdf.inqGrpParent(gid);
fullName = netcdf.inqGrpNameFull(parentId);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_grp_parent` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf.inqGrps`

netcdf.inqGrps

Retrieve array of child group IDs

Syntax

```
childGrps = netcdf.inqGrps(ncid)
```

Description

`childGrps = netcdf.inqGrps(ncid)` returns all the child group IDs in the parent group, specified by `ncid`.

Input Arguments

`ncid`

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

Output Arguments

`childGrps`

Array containing identifiers of child groups in the specified NetCDF file or group.

Examples

This example opens the sample NetCDF file and then gets information about the groups it contains.

```
ncid = netcdf.open('example.nc','nowrite');
childGroups = netcdf.inqGrps(ncid);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_grps` function in the netCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf.inqNcid`

netcdf.inqNcid

Return ID of named group

Syntax

```
childGroupId = netcdf.inqNcid(ncid,childGroupName)
```

Description

`childGroupId = netcdf.inqNcid(ncid,childGroupName)` returns the ID of the child group, specified by the name `childGroupName`, in the file or group specified by `ncid`.

Input Arguments

`ncid`

Identifier of a NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

`childGroupName`

Character vector or string scalar specifying the name of a NetCDF group.

Default:

Output Arguments

`childGroupID`

Identifier of a NetCDF group.

Examples

This example opens the sample NetCDF dataset and then gets the ID of a group in the dataset.

```
ncid = netcdf.open('example.nc','nowrite');
gid = netcdf.inqNcid(ncid,'grid1');
netcdf.close(ncid);
```

References

This function corresponds to the nc_inq_ncid function in the netCDF library C API. Read the files `netcdfcopyright.txt` and `mexnccopyright.txt` for more information.

See Also

`netcdf.inqGrpName` | `netcdf.inqGrpNameFull`

netcdf.inqUnlimDims

Return list of unlimited dimensions in group

Syntax

```
unlimdimIDs = netcdf.inqUnlimDims(ncid)
```

Description

`unlimdimIDs = netcdf.inqUnlimDims(ncid)` returns the IDs of all unlimited dimensions in the group specified by `ncid`.

Input Arguments

`ncid`

Identifier of a NetCDF file, returned by `netcdf.create` or `netcdf.open`, or group, returned by `netcdf.defGrp`.

Default:

Output Arguments

`unlimDimIDs`

An array containing the identifiers of each unlimited dimension. `unlimDimIDs` is empty if there are no unlimited dimensions.

Examples

This example opens the NetCDF sample dataset and gets the IDs of all the unlimited dimensions.

```
ncid = netcdf.open('example.nc','NOWRITE');
dimids = netcdf.inqUnlimDims(ncid)

dimids =

     []

netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_unlim_dims` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf.defDim` | `netcdf.inqDim` | `netcdf.inqDimID` | `netcdf.inqDimIDs` | `netcdf.renameDim`

netcdf.inqVarIDs

IDs of all variables in group

Syntax

```
varids = netcdf.inqVarIDs(ncid)
```

Description

`varids = netcdf.inqVarIDs(ncid)` returns IDs of the all the variables in the group specified by `ncid`.

Input Arguments

`ncid`

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

Output Arguments

`varids`

Array containing identifiers of variables in a NetCDF file or group.

Examples

This example opens the NetCDF sample file and gets the IDs of all the variables in a group.

```
ncid = netcdf.open('example.nc','NOWRITE');
gid = netcdf.inqNcid(ncid,'grid1');
varids = netcdf.inqVarIDs(gid);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_varids` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf.inqDimIDs` | `netcdf.inqVarID`

netcdf.inqVarChunking

Determine chunking settings for NetCDF variable

Syntax

```
[storage,chunkSizes] = netcdf.inqVarChunking(ncid,varid)
```

Description

`[storage,chunkSizes] = netcdf.inqVarChunking(ncid,varid)` returns the type of chunking and the dimensions of a chunk for the NetCDF variable specified by `varid`, in the file or group specified by `ncid`.

Input Arguments

`ncid`

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

`varid`

Identifier of NetCDF variable, returned by `netcdf.defVar`.

Default:

Output Arguments

storage

Character vector specifying if NetCDF breaks the data into chunks when writing to a file. The value `'CHUNKED'` indicates the data is chunked; `'CONTIGUOUS'` indicates that the data is not chunked.

chunkSizes

Array specifying the dimensions of the chunk.

Because MATLAB uses FORTRAN-style ordering, the order of dimensions in `chunkdims` is reversed relative to what would be in the NetCDF C API.

If the storage type specified is `CONTIGUOUS`, `netcdf.inqVarChunking` returns an empty array, `[]`.

Examples

This example opens the NetCDF sample dataset and gets the values of chunking parameters associated with a variable.

```
ncid = netcdf.open('example.nc','NOWRITE');
groupid = netcdf.inqNcid(ncid,'grid1');
varid = netcdf.inqVarID(groupid,'temp');
[storage,chunkSize] = netcdf.inqVarChunking(groupid,varid);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_var_chunking` function in the netCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf.defVar` | `netcdf.defVarChunking`

netcdf.inqVarDeflate

Determine compression settings for NetCDF variable

Syntax

```
[shuffle,deflate,deflateLevel] = netcdf.inqVarDeflate(ncid,varid)
```

Description

`[shuffle,deflate,deflateLevel] = netcdf.inqVarDeflate(ncid,varid)` returns the compression parameters for the NetCDF variable specified by `varid` in the location specified by `ncid`.

Input Arguments

`ncid`

Identifier of a NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Default:

`varid`

Identifier of NetCDF variable, returned by `netcdf.defVar`.

Default:

Output Arguments

shuffle

Boolean value. `true` indicates that the shuffle filter is enabled for the specified variable. The shuffle filter can assist with the compression of integer data by changing the byte order in the data stream.

deflate

Boolean value. `true` indicates that compression is enabled for this variable. The `deflateLevel` argument specifies the level of compression.

deflateLevel

Scalar value between `0` and `9` specifying the amount of compression, where `0` is no compression and `9` is the most compression

Examples

This example opens the NetCDF sample file and gets information about variable compression.

```
ncid = netcdf.open('example.nc','NOWRITE');
groupid = netcdf.inqNcid(ncid,'grid1');
varid = netcdf.inqVarID(groupid,'temp');
[shuffle,deflate,deflateLevel] = netcdf.inqVarDeflate(groupid,varid);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_var_deflate` function in the netCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf.defVarDeflate`

netcdf.inqVarFill

Determine values of fill parameters for NetCDF variable

Syntax

```
[noFillMode,fillValue] = netcdf.inqVarFill(ncid,varid)
```

Description

`[noFillMode,fillValue] = netcdf.inqVarFill(ncid,varid)` returns the fill mode and the fill value for the variable `varid` in the NetCDF-4 file or group specified by `ncid`.

Input Arguments

`ncid`

Identifier of a NetCDF-4 file, returned by `netcdf.create` or `netcdf.open`, or a NetCDF group, returned by `netcdf.defGrp`.

Default:

`varid`

Identifier of NetCDF variable.

Default:

Output Arguments

`noFillMode`

Boolean value. `true` indicates that use of the fill values for the variable has been disabled.

fillValue

Specifies the value to use in the variable when no other value is specified and use of fill values has been enabled.

Examples

This example opens the NetCDF sample dataset and gets the fill mode and fill value used with a variable.

```
ncid = netcdf.open('example.nc','NOWRITE');
varid = netcdf.inqVarID(ncid,'temperature');
[noFillMode,fillValue] = netcdf.inqVarFill(ncid,varid);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_var_fill` function in the netCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf.defVarFill` | `netcdf.setFill`

netcdf.inqVarFletcher32

Fletcher32 checksum setting for NetCDF variable

Syntax

```
setting = netcdf.inqVarFletcher32(ncid,varid)
```

Description

`setting = netcdf.inqVarFletcher32(ncid,varid)` returns the Fletcher32 checksum setting for the NetCDF variable specified by `varid` in the file or group specified by `ncid`.

Input Arguments

`ncid`

Identifier for NetCDF file, returned by `netcdf.create` or `netcdf.open`, or group, returned by `netcdf.defGrp`.

Default:

`varid`

Identifier of NetCDF variable.

Default:

Output Arguments

setting

Character vector specifying whether the Fletcher32 checksum is turned on for the specified variable. `netcdf.inqVarFletcher32` returns `'FLETCHER32'` or `'NOCHECKSUM'`.

- `'FLETCHER32'` — If the checksum is turned on for the variable.
- `'NOCHECKSUM'` — Otherwise.

Examples

This example opens the sample NetCDF file and gets information about the checksum setting for a variable.

```
ncid = netcdf.open('example.nc','NOWRITE');
varid = netcdf.inqVarID(ncid,'temperature');
setting = netcdf.inqVarFletcher32(ncid,varid);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_var_fletcher32` function in the netCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` for more information.

See Also

`netcdf.defVarFletcher32`

netcdf.inqAtt

Return information about netCDF attribute

Syntax

```
[xtype,attlen] = netcdf.inqAtt(ncid,varid,attname)
```

Description

`[xtype,attlen] = netcdf.inqAtt(ncid,varid,attname)` returns the data type, `xtype`, and length, `attlen`, of the attribute identified in `attname`. Specify `attname` as a character vector or string scalar.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`varid` identifies the variable that the attribute is associated with. To get information about a global attribute, specify `netcdf.getConstant('NC_GLOBAL')` in place of `varid`.

This function corresponds to the `nc_inq_att` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`, and gets information about an attribute in the file.

```
% Open netCDF example file.
ncid = netcdf.open('example.nc','NOWRITE');

% Get identifier of a variable in the file, given its name.
varid = netcdf.inqVarID(ncid,'avagadros_number');

% Get attribute name, given variable id and attribute number.
attname = netcdf.inqAttName(ncid,varid,0);
```

```
% Get information about the attribute.
[xtype,attlen] = netcdf.inqAtt(ncid,varid,'description')

xtype =

     2


attlen =

    31

% Get name of global attribute
gattname = netcdf.inqAttName(ncid,netcdf.getConstant('NC_GLOBAL'),0);

% Get information about global attribute.
[gxtype gattlen] = netcdf.inqAtt(ncid,netcdf.getConstant('NC_GLOBAL'),gattname)

gxtype =

     2


gattlen =

    11
```

See Also

netcdf.inqAttID | netcdf.inqAttName

netcdf.inqAttID

Return ID of netCDF attribute

Syntax

```
attnum = netcdf.inqAttID(ncid,varid,attname)
```

Description

`attnum = netcdf.inqAttID(ncid,varid,attname)` retrieves `attnum`, the identifier of the attribute indicated by `attname`. Specify `attname` as a character vector or string scalar.

`varid` specifies the variable the attribute is associated with.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_inq_attid` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm.

Examples

This example opens the netCDF example file included with MATLAB, `example.nc`.

```
% Open the netCDF example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get the identifier of a variable in the file.
varid = netcdf.inqVarID(ncid,'avagadros_number');

% Retrieve the identifier of the attribute associated with the variable.
attid = netcdf.inqAttID(ncid,varid,'description');
```

See Also

`netcdf.inqAtt` | `netcdf.inqAttName`

netcdf.inqAttName

Return name of netCDF attribute

Syntax

```
attname = netcdf.inqAttName(ncid,varid,attnum)
```

Description

`attname = netcdf.inqAttName(ncid,varid,attnum)` returns `attname`, a character vector containing the name of an attribute.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`varid` is a numeric identifier of a variable in the file. If you want to get the name of a global attribute in the file, use `netcdf.getConstant('NC_GLOBAL')` in place of `attnum` is a zero-based numeric value specifying the attribute, with 0 indicating the first attribute, 1 the second attribute, and so on.

This function corresponds to the `nc_inq_attname` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`.

```
% Open netCDF example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get identifier of a variable in the file.
varid = netcdf.inqVarID(ncid,'avagadros_number')

% Get the name of the attribute associated with the variable.
attname = netcdf.inqAttName(ncid,varid,0)
```

```
attname =

description

% Get the name of the global attribute associated with the variable.
gattname = netcdf.inqAttName(ncid,netcdf.getConstant('NC_GLOBAL'),0)

gattname =

creation_date
```

See Also

netcdf.inqAtt | netcdf.inqAttID

netcdf.inqDim

Return netCDF dimension name and length

Syntax

```
[dimname, dimlen] = netcdf.inqDim(ncid,dimid)
```

Description

`[dimname, dimlen] = netcdf.inqDim(ncid,dimid)` returns the name, `dimname`, and length, `dimlen`, of the dimension specified by `dimid`. If `ndims` is the number of dimensions defined for a netCDF file, each dimension has an ID between 0 and `ndims-1`. For example, the dimension identifier of the first dimension is 0, the second dimension is 1, and so on.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_inq_dim` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm.

Examples

The example opens the example netCDF file include with MATLAB, `example.nc`.

```
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get name and length of first dimension
[dimname, dimlen] = netcdf.inqDim(ncid,0)

dimname =

x


dimlen =
```

```
50
```

See Also

netcdf.inqDimID

netcdf.inqDimID

Return dimension ID

Syntax

```
dimid = netcdf.inqDimID(ncid,dimname)
```

Description

`dimid = netcdf.inqDimID(ncid,dimname)` returns `dimid` corresponding to `dimname`. Specify `dimname` as a character vector or string scalar. You can use the `netcdf.inqDim` function to retrieve the dimension name. `ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_inq_dimid` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`.

```
% Open netCDF example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get name and length of first dimension
[dimname, dimlen] = netcdf.inqDim(ncid,0);

% Retrieve identifier of dimension.
dimid = netcdf.inqDimID(ncid,dimname)

dimid =

     0
```

See Also

`netcdf.inqDim`

netcdf.inqLibVers

Return NetCDF library version information

Syntax

```
libvers = netcdf.inqLibVers
```

Description

`libvers = netcdf.inqLibVers` returns a character vector identifying the version of the NetCDF library.

This function corresponds to the `nc_inq_libvers` function in the NetCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm.

Examples

```
libvers = netcdf.inqLibVers

libvers =

4.6.1
```

netcdf.inqVar

Information about variable

Syntax

```
[varname,xtype,dimids,natts] = netcdf.inqVar(ncid,varid)
```

Description

`[varname,xtype,dimids,natts] = netcdf.inqVar(ncid,varid)` returns information about the variable identified by `varid`. The argument, `ncid`, is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

The output argument, `varname`, is the name of the variable. `xtype` is the data type, `dimids` is the dimension IDs, and `natts` is the number of attributes associated with the variable. Dimension IDs are zero-based.

This function corresponds to the `nc_inq_var` function in the netCDF library C API. Because MATLAB uses FORTRAN-style ordering, however, the order of the dimension IDs is reversed relative to what would be obtained from the C API. To use this function, you should be familiar with the netCDF programming paradigm.

Examples

Open the example netCDF file included with MATLAB, `example.nc`, and get information about a variable in the file.

```
% Open the example netCDF file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get information about third variable in the file.
[varname, xtype, dimids, numatts] = netcdf.inqVar(ncid,2)

varname =
```

```
peaks

xtype =

     5


dimids =

     0     1


numatts =

     1     1
```

See Also

netcdf.create | netcdf.inqVarID | netcdf.open

netcdf.inqVarID

Return ID associated with variable name

Syntax

```
varid = netcdf.inqVarID(ncid,varname)
```

Description

`varid = netcdf.inqVarID(ncid,varname)` returns `varid` corresponding to `varname`. Specify `varname` as character vector or string scalar.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_inq_varid` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`, and uses several inquiry functions to get the ID of the first variable.

```
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get information about first variable in the file.
[varname, xtype, dimids, atts] = netcdf.inqVar(ncid,0);

% Get variable ID of the first variable, given its name
varid = netcdf.inqVarID(ncid,varname)

varid =

     0
```

See Also

`netcdf.create` | `netcdf.inqVar` | `netcdf.open`

netcdf.open

Open NetCDF data source

Syntax

```
ncid = netcdf.open(source)
ncid = netcdf.open(source,mode)
[actualChunksize,ncid] = netcdf.open(source,mode,chunksize)
```

Description

`ncid = netcdf.open(source)` opens the `source` for read-only access and returns a NetCDF identifier in `ncid`. The`source` specified can be the name of a NetCDF file or the URL of an OPeNDAP NetCDF data source.

`ncid = netcdf.open(source,mode)` opens `source` with the type of access specified by `mode`. Mode values are `'WRITE'`, `'SHARE'`, or `'NOWRITE'`.

`[actualChunksize,ncid] = netcdf.open(source,mode,chunksize)` opens an existing NetCDF data source, with a specified `chunksize`. The `chunksize` parameter enables I/O performance tuning.

Examples

Read from NetCDF File

Open the sample NetCDF file, `example.nc` with read-only access, read a variable, and then close the file. The `netcdf.open` function returns a file identifier.

```
ncid = netcdf.open('example.nc')
```

```
ncid = 65536
```

Read the data associated with the variable, `avagadros_number`, in the example file, using the `netcdf.getVar` function. The second input to `netcdf.getVar` is the variable

ID, which is a zero-based index that identifies the variable. The `avagadros_number` variable has the index value `0`.

```
A_number = netcdf.getVar(ncid,0)
```

```
A_number = 6.0221e+23
```

Close the NetCDF file.

```
netcdf.close(ncid)
```

Open NetCDF File for Writing

Open an NetCDF and assign a value to a variable, and then display it.

Create NetCDF File `myexample.nc` containing a variable `pi`.

```
nccreate('myexample.nc','pi')
```

Open the file for writing and write the value `3.1416` to the variable `pi`.

```
ncid = netcdf.open('myexample.nc','WRITE')
```

```
ncid = 65536
```

```
varInd = 0; % index of the first variable in the file
varValue = 3.1416 ;
netcdf.putVar(ncid,varInd,varValue);
```

Read and display the value of the variable `pi` from the file.

```
netcdf.getVar(ncid,varInd)
```

```
ans = 3.1416
```

Close the file.

```
netcdf.close(ncid);
```

Input Arguments

source — Source name

character vector | string scalar

Source name, specified as a character vector or string scalar containing the name of a NetCDF file or the URL of an OPeNDAP NetCDF data source.

Data Types: `char` | `string`

mode — Access type

`'NOWRITE'` (default) | `'WRITE'` | `'SHARE'` | numeric value

Access type, specified as a character vector or string scalar, or a numeric value. The `mode` input can one of these values.

Value	Description
`'WRITE'`	Read-write access
`'SHARE'`	Synchronous file updates
`'NOWRITE'`	Read-only access (Default)

You also can specify `mode` with a numeric value. Retrieve the numerical value for `mode` by using `netcdf.getConstant`. For example, `netcdf.getConstant('WRITE')` returns the numeric value for `mode`. Use these numeric values when you want to specify a bitwise-OR of several modes.

Data Types: `char` | `string` | `double`

chunksize — Chunk size

integer

Chunk size parameter for performance tuning, specified as an integer.

The `chunksize` parameter controls the space-versus-time tradeoff, memory that the `netcdf.open` function allocates in the NetCDF library versus the number of system calls. Because of internal requirements, the actual value of the chunk size used by the `netcdf.open` function can be different than the input value you specify. For the actual value used by the function, see the `actualChunksize` argument.

Example: `1024`

Example: `8192`

Data Types: `double`

Output Arguments

`ncid` — File identifier
integer

File identifier of an open NetCDF file or OPeNDAP NetCDF data source, returned as an integer.

Data Types: `double`

`actualChunksize` — Actual chunk size
integer

Actual chunk size used by the `netcdf.open` function, returned as an integer.

Data Types: `double`

Algorithms

This function corresponds to the `nc_open` and `nc__open` functions in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm.

See Also

`netcdf.close` | `netcdf.getConstant`

Introduced in R2011a

netcdf.putAtt

Write netCDF attribute

Syntax

```
netcdf.putAtt(ncid,varid,attrname,attrvalue)
```

Description

`netcdf.putAtt(ncid,varid,attrname,attrvalue)` writes the attribute named `attrname` with value `attrvalue` to the netCDF variable specified by `varid`. To specify a global attribute, use `netcdf.getConstant('NC_GLOBAL')` for `varid`.

`ncid` is a netCDF file identifier returned by `netCDF.create` or `netCDF.open`.

Note You cannot use `netcdf.putAtt` to set the `'_FillValue'` attribute of NetCDF4 files. Use the `netcdf.defVarFill` function to set the fill value for a variable.

This function corresponds to several attribute I/O functions in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm.

Examples

This example creates a new netCDF file, defines a dimension and a variable, adds data to the variable, and then creates an attribute associated with the variable. To run this example, you must have writer permission in your current directory.

```
% Create a variable in the workspace.
my_vardata = linspace(0,50,50);

% Create a netCDF file.
ncid = netcdf.create('foo.nc','NC_WRITE');
```

```
% Define a dimension in the file.
dimid = netcdf.defDim(ncid,'my_dim',50);

% Define a new variable in the file.
varid = netcdf.defVar(ncid,'my_var','double',dimid);

% Leave define mode and enter data mode to write data.
netcdf.endDef(ncid);

% Write data to variable.
netcdf.putVar(ncid,varid,my_vardata);

% Re-enter define mode.
netcdf.reDef(ncid);

% Create an attribute associated with the variable.
netcdf.putAtt(ncid,0,'my_att',10);

% Verify that the attribute was created.
[xtype xlen] = netcdf.inqAtt(ncid,0,'my_att')

xtype =

     6


xlen =

     1
```

This example creates a new netCDF file, specifies a global attribute, and assigns a value to the attribute.

```
ncid = netcdf.create('myfile.nc','CLOBBER');
varid = netcdf.getConstant('GLOBAL');
netcdf.putAtt(ncid,varid,'creation_date',datestr(now));
netcdf.close(ncid);
```

See Also

`netcdf.defVarFill` | `netcdf.getAtt` | `netcdf.getConstant`

netcdf.putVar

Write data to netCDF variable

Syntax

```
netcdf.putVar(ncid,varid,data)
netcdf.putVar(ncid,varid,start,data)
netcdf.putVar(ncid,varid,start,count,data)
netcdf.putVar(ncid,varid,start,count,stride,data)
```

Description

`netcdf.putVar(ncid,varid,data)` writes `data` to a netCDF variable identified by `varid`.

Note For variable values containing text data, the `data` input must have only ASCII encoded characters.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`netcdf.putVar(ncid,varid,start,data)` writes a single data value into the variable at the index specified by `start`.

`netcdf.putVar(ncid,varid,start,count,data)` writes a section of values into the netCDF variable at the index specified by the vector `start` to the extent specified by the vector `count`, along each dimension of the specified variable.

`netcdf.putVar(ncid,varid,start,count,stride,data)` writes the subsection specified by sampling interval, `stride`, of the values in the section of the variable beginning at the index `start` and to the extent specified by `count`.

This function corresponds to several variable I/O functions in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm.

Examples

Write Variable to New netCDF File

Create a new netCDF file and write a variable to the file.

Create a 50 element vector for a variable.

```
my_vardata = linspace(0,50,50);
```

Open the netCDF file.

```
ncid = netcdf.create('foo.nc','NOCLOBBER');
```

Define the dimensions of the variable.

```
dimid = netcdf.defDim(ncid,'my_dim',50);
```

Define a new variable in the file.

```
my_varID = netcdf.defVar(ncid,'my_var','double',dimid);
```

Leave define mode and enter data mode to write data.

```
netcdf.endDef(ncid);
```

Write data to variable.

```
netcdf.putVar(ncid,my_varID,my_vardata);
```

Verify that the variable was created.

```
[varname xtype dimid natts ] = netcdf.inqVar(ncid,0)
```

```
varname =
'my_var'

xtype = 6

dimid = 0

natts = 0
```

Close the file.

```
netcdf.close(ncid)
```

Write Elements of Variable

Write to the first ten elements of the example `temperature` variable.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.nc');
copyfile(srcFile,'myfile.nc');
fileattrib('myfile.nc','+w');
ncid = netcdf.open('myfile.nc','WRITE');
varid = netcdf.inqVarID(ncid,'temperature');
data = [100:109];
netcdf.putVar(ncid,varid,0,10,data);
netcdf.close(ncid);
```

See Also

`netcdf.getVar`

netcdf.reDef

Put open netCDF file into define mode

Syntax

`netcdf.reDef(ncid)`

Description

`netcdf.reDef(ncid)` puts an open netCDF file into define mode so that dimensions, variables, and attributes can be added or renamed. Attributes can also be deleted in define mode. `ncid` is a valid NetCDF file ID, returned from a previous call to `netcdf.open` or `netcdf.create`.

This function corresponds to the `nc_redef` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm.

Examples

This example opens a local copy of the example netCDF file included with MATLAB, `example.nc`.

```
% Open a netCDF file.
ncid = netcdf.open('my_example.nc','NC_WRITE')

% Try to define a dimension.
dimid = netcdf.defdim(ncid, 'lat', 50); % should fail.
??? Error using ==> netcdflib
NetCDF: Operation not allowed in data mode

Error in ==> defDim at 22
dimid = netcdflib('def_dim', ncid,dimname,dimlen);

% Put file in define mode.
netcdf.reDef(ncid);
```

```
% Try to define a dimension again. Should succeed.
dimid = netcdf.defDim(ncid, 'lat', 50);
```

See Also

netcdf.create | netcdf.endDef | netcdf.open

netcdf.renameAtt

Change name of attribute

Syntax

`netcdf.renameAtt(ncid,varid,oldName,newName)`

Description

`netcdf.renameAtt(ncid,varid,oldName,newName)` changes the name of the attribute specified in `oldName` to the name specified in `newName`.

Specify `oldName` and `newName` as character vectors or string scalars.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`varid` identifies the variable to which the attribute is associated. To specify a global attribute, use `netcdf.getConstant('NC_GLOBAL')` for `varid`.

This function corresponds to the `nc_rename_att` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm.

Examples

This example modifies a local copy of the example netCDF file included with MATLAB, `example.nc`.

```
% Open netCDF file.
ncid = netcdf.open('my_example.nc','NC_WRITE')

% Get the ID of a variable the attribute is associated with.
varID = netcdf.inqVarID(ncid,'avagadros_number')

% Rename the attribute.
netcdf.renameAtt(ncid,varID,'description','Description');
```

```
% Verify that the name changed.
attname = netcdf.inqAttName(ncid,varID,0)

attname =

Description
```

See Also

netcdf.inqAttName

netcdf.renameDim

Change name of netCDF dimension

Syntax

```
netcdf.renameDim(ncid,dimid,newName)
```

Description

`netcdf.renameDim(ncid,dimid,newName)` renames the dimension identified by the dimension identifier, `dimid`.

`newName` is a character vector or string scalar specifying the new name. `ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`

This function corresponds to the `nc_rename_dim` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm.

Examples

This example modifies a local copy of the example netCDF file included with MATLAB, `example.nc`.

```
% Open netCDF file.
ncid = netcdf.open('my_example.nc','NC_WRITE')

% Put file is define mode.
netcdf.reDef(ncid)

% Get the identifier of a dimension to rename.
dimid = netcdf.inqDimID(ncid,'x');

% Rename the dimension.
netcdf.renameDim(ncid,dimid,'Xdim')

% Verify that the name changed.
```

```
data = netcdf.inqDim(ncid,dimid)

data =

Xdim
```

See Also

netcdf.defDim

netcdf.renameVar

Change name of netCDF variable

Syntax

```
netcdf.renameVar(ncid,varid,newName)
```

Description

`netcdf.renameVar(ncid,varid,newName)` renames the variable identified by `varid` in the netCDF file identified by `ncid`.`newName` is a character vector or string scalar specifying the new name.

This function corresponds to the `nc_rename_var` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm.

Examples

This example modifies a local copy of the example netCDF file included with MATLAB, `example.nc`.

```
% Open netCDF file.
ncid = netcdf.open('my_example.nc','NC_WRITE')

% Put file in define mode.
netcdf.redef(ncid)

% Get name of first variable
[varname, xtype, varDimIDs, varAtts] = netcdf.inqVar(ncid,0);

varname

varname =

avagadros_number
```

```
% Rename the variable, using a capital letter to start the name.
netcdf.renameVar(ncid,0,'Avagadros_number')

% Verify that the name of the variable changed.
[varname, xtype, varDimIDs, varAtts] = netcdf.inqVar(ncid,0);

varname

varname =

Avagadros_number
```

See Also

netCDF.defVar | netCDF.inqVar | netCDF.putVar

netcdf.setChunkCache

Set default chunk cache settings for NetCDF library

Syntax

```
netcdf.setChunkCache(csize,nelems,premp)
```

Description

`netcdf.setChunkCache(csize,nelems,premp)` sets the default chunk cache settings used by the NetCDF library.

Settings apply for subsequent file open or create operations, for the remainder of the MATLAB session or until you issue a `clear mex` call. This function does not change the chunk cache settings of files already open.

Input Arguments

`csize`

Scalar double specifying the total size of the raw data chunk cache in bytes.

Default:

`nelems`

Scalar double specifying the number of chunk slots in the raw data chunk cache hash table.

Default:

`premp`

Scalar double, between `0` and `1`, inclusive, that specifies how the library handles preempting fully read chunks in the chunk cache. A value of `0` means fully read chunks

are treated no differently than other chunks, that is, preemption occurs solely based on the Least Recently Used (LRU) algorithm. A value of 1 means fully read chunks are always preempted before other chunks.

Default:

Examples

This example sets the cache chunk size used by the NetCDF library.

```
netcdf.setChunkCache(32000000, 2003, .75)
```

References

This function corresponds to the `nc_set_chunk_cache` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf.getChunkCache`

netcdf.setDefaultFormat

Change default netCDF file format

Syntax

```
oldFormat = netcdf.setDefaultFormat(newFormat)
```

Description

`oldFormat = netcdf.setDefaultFormat(newFormat)` changes the default format used by `netCDF.create` when creating new netCDF files, and returns the value of the old format. You can use this function to change the format used by a netCDF file without having to change the creation mode flag used in each call to `netCDF.create`. This setting persists for the remainder of the MATLAB session or until you issue the command `clear mex`.

`newFormat` can be one of these values.

Value	Description
`'NC_FORMAT_CLASSIC'`	Classic format — Original NetCDF format, used by all NetCDF files created between 1989 and 2004.
`'NC_FORMAT_64BIT'`	Classic format, 64–bit — Original format with 64–bit addressing capability to allow creation and access of much larger files.
`'NC_FORMAT_NETCDF4'`	Enhanced model, HDF5-based — Introduced in 2008, NetCDF, version 4, extends the classic model and is based on HDF5.
`'NC_FORMAT_NETCDF4_CLASSIC'`	Classic model, HDF5-based — Introduced in 2008, NetCDF, version 4, implements classic model but is based on HDF5.

You can also specify the numeric equivalent of these values, as retrieved by `netcdf.getConstant`.

This function corresponds to the `nc_set_default_format` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm.

Examples

```
oldFormat = netcdf.setDefaultFormat('NC_FORMAT_64BIT');
```

See Also

`netcdf.create`

netcdf.setFill

Set netCDF fill mode

Syntax

```
old_mode = netcdf.setFill(ncid,new_mode)
```

Description

`old_mode = netcdf.setFill(ncid,new_mode)` sets the fill mode for a netCDF file identified by `ncid`.

`new_mode` can be either `'FILL'` or `'NOFILL'` or their numeric equivalents, as retrieved by `netcdf.getConstant`. The default mode is `'FILL'`. netCDF pre-fills data with fill values. Specifying `'NOFILL'` can be used to enhance performance, because it avoids the duplicate writes that occur when the netCDF writes fill values that are later overwritten with data.

This function corresponds to the `nc_set_fill` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm.

Examples

This example creates a new file and specifies the fill mode used by netCDF with the file.

```
ncid = netcdf.open('foo.nc','NC_WRITE');

% Set filling behavior
old_mode = netcdf.setFill(ncid,'NC_NOFILL');
```

See Also

`netcdf.getConstant`

netcdf.sync

Synchronize netCDF file to disk

Syntax

```
netcdf.sync(ncid)
```

Description

`netcdf.sync(ncid)` synchronizes the state of a netCDF file to disk. The netCDF library normally buffers accesses to the underlying netCDF file, unless you specify the `NC_SHARE` mode when you opened the file with `netcdf.open` or `netcdf.create`. To call `netcdf.sync`, the netCDF file must be in data mode.

This function corresponds to the `nc_sync` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm.

Examples

This example creates a new netCDF file for write access, performs an operation on the file, takes the file out of define mode, and then synchronizes the file to disk.

```
% Create a netCDF file.
ncid = netcdf.create('foo.nc','NC_WRITE');

% Perform an operation.
dimid = netcdf.defDim(ncid,'Xdim',50);

% Take file out of define mode.
netcdf.endDef(ncid);

% Synchronize the file to disk.
netcdf.sync(ncid)
```

See Also

`netcdf.close` | `netcdf.create` | `netcdf.endDef` | `netcdf.open`

newline

Create newline character

Syntax

```
c = newline
```

Description

`c = newline` creates a newline character. `newline` is equivalent to `char(10)` or `sprintf('\n')`. Use `newline` to concatenate a newline character onto a character vector or a string, or to split text on newline characters.

Examples

Add Newline to Character Vector and String

Create a newline character with `newline` and concatenate it onto a character vector.

```
chr = 'Whose woods these are I think I know.';
chr = [chr newline 'His house is in the village though']
```

```
chr =
    'Whose woods these are I think I know.
     His house is in the village though'
```

Although `chr` displays on two lines, `chr` is a 1-by-73 character vector that contains the two sentences, separated by a newline.

Starting in R2017a, you can create strings using double quotes. Create a newline character. Then use + to concatenate the newline character and more text onto the end of a string.

```
str = "In Xanadu did Kubla Khan";
str = str + newline + "A stately pleasure-dome decree"
```

```
str =
    "In Xanadu did Kubla Khan
     A stately pleasure-dome decree"
```

Although `str` displays on two lines, `str` is a 1-by-1 string.

Tips

- The `newline` function does not return a carriage return character. A carriage return is equivalent to `char(13)` or `sprintf('\r')`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`char` | `compose` | `join` | `split` | `splitlines` | `sprintf` | `string`

Introduced in R2016b

newplot

Determine where to draw graphics objects

Syntax

```
newplot
h = newplot
h = newplot(sv)
```

Description

`newplot` prepares a figure and axes for subsequent graphics commands.

`h = newplot` prepares a figure and axes for subsequent graphics commands and returns the current axes.

`h = newplot(sv)` does not delete any objects specified by the `sv` argument, which can be a vector of graphics objects. The figure and axes containing the objects in `sv` are prepared for plotting instead of the current axes of the current figure. If `sv` is empty, `newplot` behaves as if it were called without any inputs.

Tips

To create a simple 2-D plot, use the `plot` function instead.

Use `newplot` at the beginning of high-level graphics code to determine which figure and axes to target for graphics output. Calling `newplot` can change the current figure and current axes. Basically, there are three options when you are drawing graphics in existing figures and axes:

- Add the new graphics without changing any properties or deleting any objects.
- Delete all existing objects whose handles are not hidden before drawing the new objects.

- Delete all existing objects regardless of whether or not their handles are hidden, and reset most properties to their defaults before drawing the new objects (refer to the following table for specific information).

The figure and axes `NextPlot` properties determine how `newplot` behaves. The following two tables describe this behavior with various property values.

First, `newplot` reads the current figure's `NextPlot` property and acts accordingly.

NextPlot	What Happens
`new`	Create a new figure and use it as the current figure.
`add`	Draw to the current figure without clearing any graphics objects already present.
`replacechildren`	Remove all child objects whose `HandleVisibility` property is set to `on` and reset figure `NextPlot` property to `add`. This clears the current figure and is equivalent to issuing the `clf` command.
`replace`	Remove all child objects (regardless of the setting of the `HandleVisibility` property) and reset figure properties to their defaults, except `NextPlot` is reset to `add` regardless of user-defined defaults. • `Position`, `Units`, `PaperPosition`, and `PaperUnits` are not reset. This clears and resets the current figure and is equivalent to issuing the `clf reset` command.

After `newplot` establishes which figure to draw in, it reads the current axes' `NextPlot` property and acts accordingly.

NextPlot	Description
`add`	Add new plots to the existing axes. Do not delete existing plots or reset axes properties before displaying the new plot.

NextPlot	Description
`replacechildren`	Delete existing plots before displaying the new plot. Reset the `ColorOrderIndex` and `LineStyleOrderIndex` axes properties to 1, but do not reset other axes properties. The next plot added to the axes uses the first color and line style based on the `ColorOrder` and `LineStyleOrder` properties of the axes. This value is similar to using `cla` before every new plot.
`replace`	Delete existing plots and reset all axes properties, except `Position` and `Units`, to their default values before displaying the new plot.
`replaceall`	Delete existing plots and reset all axes properties, except `Position` and `Units`, to their default values before displaying the new plot. This value is similar to using `cla reset` before every new plot. For axes with only one *y*-axes, the `replace` and `replaceall` values are equivalent. For axes with two *y*-axes, the `replace` value affects only the active side while the `replaceall` value affects both sides.

See Also

`axes` | `cla` | `clf` | `figure` | `hold` | `ishold` | `plot` | `reset`

Topics

"Control Graph Display"

Introduced before R2006a

nextDirectory

Make next IFD the current IFD

Syntax

```
nextDirectory(t)
```

Description

`nextDirectory(t)` makes the next image file directory (IFD) of the TIFF file the current IFD.

The `Tiff` object functions operate on the current IFD. Use this function to navigate among IFDs in a TIFF file containing multiple images.

Examples

Navigate to Next Image in TIFF File

Get information on the `Photometric` tag of the image in the current IFD, navigate to the next image in the file, and examine the same tag for the next image.

Create a `Tiff` object for the file `example.tif`.

```
t = Tiff('example.tif','r');
```

Display the current directory number and get the value of the `Photometric` tag from the current IFD. A value of 2 for the `Photometric` tag corresponds to the colorspace `RGB` in the TIFF specifications.

```
dirNum = currentDirectory(t)
```

```
dirNum = 1
```

```
value = getTag(t,'Photometric')
```

```
value = 2
```

Navigate to the next IFD

```
nextDirectory(t);
```

Display the current directory number and get the value of the `Photometric` tag from this IFD. A value of `1` for the `Photometric` tag corresponds to the colorspace `MinIsBlack` in the TIFF specifications.

```
dirNum = currentDirectory(t)
```

```
dirNum = 2
```

```
value = getTag(t,'Photometric')
```

```
value = 1
```

Close the `Tiff` object.

```
close(t);
```

Input Arguments

t — Tiff object
`Tiff` object

`Tiff` object representing a TIFF file. Use the `Tiff` function to create the object.

Algorithms

References

This function corresponds to the `TIFFReadDirectory` function in the LibTIFF C API. To use this function, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff` | `setDirectory`

Topics

"Importing Images"
"Exporting to Images"

Introduced in R2009b

nextpow2

Exponent of next higher power of 2

Syntax

```
P = nextpow2(A)
```

Description

`P = nextpow2(A)` returns the exponents for the smallest powers of two that satisfy

$$2^p \geq |A|$$

for each element in `A`. By convention, `nextpow2(0)` returns zero.

You can use `nextpow2` to pad the signal you pass to `fft`. Doing so can speed up the computation of the FFT when the signal length is not an exact power of 2.

Examples

Next Power of 2 of Double Integer Values

Define a vector of `double` integer values and calculate the exponents for the next power of 2 higher than those values.

```
a = [1 -2 3 -4 5 9 519];
p = nextpow2(a)
```

```
p = 1×7

     0     1     2     2     3     4    10
```

Calculate the positive next powers of 2.

```
np2 = 2.^p
```

```
np2 = 1×7

           1           2           4           4           8          16        1024
```

Preserve the sign of the original input values.

```
np2.*sign(a)
```

```
ans = 1×7

           1          -2           4          -4           8          16        1024
```

Next Power of 2 of Unsigned Integer Values

Define a vector of unsigned integers and calculate the exponents for the next power of 2 higher than those values.

```
a = uint32([1020 4000 32700]);
p = nextpow2(a)
```

```
p = 1x3 uint32 row vector

   10   12   15
```

Calculate the next powers of 2 higher than the values in `a`.

```
2.^p
```

```
ans = 1x3 uint32 row vector

    1024    4096   32768
```

Optimize FFT with Padding

Use the `nextpow2` function to increase the performance of `fft` when the length of a signal is not a power of 2.

Create a 1-D vector containing 8191 sample values.

```
x = gallery('uniformdata',[1,8191],0);
```

Calculate the next power of 2 higher than 8191.

```
p = nextpow2(8191);
n = 2^p
```

```
n = 8192
```

Pass the signal and the next power of 2 to the `fft` function.

```
y = fft(x,n);
```

Input Arguments

A — Input values
scalar, vector, or array of real numbers

Input values, specified as a scalar, vector, or array of real numbers of any numeric type.

Example: 15

Example: `[-15.123 32.456 63.111]`

Example: `int16([-15 32 63])`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`fft` | `log2` | `pow2`

Introduced before R2006a

nnz

Number of nonzero matrix elements

Syntax

```
N = nnz(X)
```

Description

`N = nnz(X)` returns the number of nonzero elements in matrix `X`.

Examples

Number of Nonzeros

Create an identity matrix and determine the number of nonzeros it contains.

```
X = eye(4)
```

```
X = 4×4

     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
```

```
N = nnz(X)
```

```
N = 4
```

Number of Elements Meeting a Condition

Use `nnz` in conjunction with a relational operator to determine how many matrix elements meet a condition. Since relational operators produce logical matrices of 1s and 0s, the `nnz` function counts the 1s where the condition is true.

Create a matrix and determine how many elements are greater than 10.

```
X = magic(5)
```

```
X = 5×5

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

```
nnz(X>10)
```

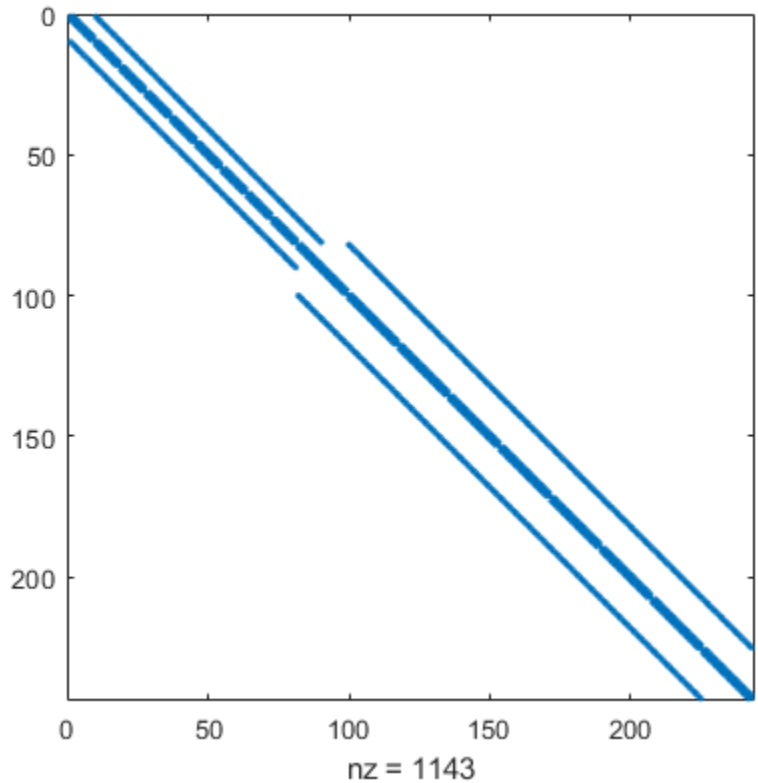
```
ans = 15
```

Density of Sparse Matrix

The density of a matrix is the ratio of nonzeros to the total number of elements, `nnz(X)/numel(X)`.

Create a sparse matrix representing the finite difference Laplacian on an L-shaped domain and calculate its density.

```
X = delsq(numgrid('L',20));
spy(X)
```




```
d = nnz(X)/numel(X)
```

```
d = 0.0194
```

The result indicates that only about 2% of the elements in the matrix are nonzero.

Input Arguments

X — Input matrix
matrix

Input matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `duration` | `calendarDuration`
Complex Number Support: Yes

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`find` | `nonzeros` | `numel` | `nzmax` | `size`

Introduced before R2006a

nonzeros

Nonzero matrix elements

Syntax

```
v = nonzeros(A)
```

Description

`v = nonzeros(A)` returns a full column vector of the nonzero elements in `A`. The elements in `v` are ordered by columns.

Examples

Nonzero Matrix Elements

Use `nonzeros` to return the nonzero elements in a sparse matrix.

Create a 10-by-10 sparse matrix that contains a few nonzero elements. The typical display of sparse matrices shows a list of the nonzero values and their locations.

```
A = sparse([1 3 2 1],[1 1 2 3],1:4,10,10)
```

```
A =
   (1,1)        1
   (3,1)        2
   (2,2)        3
   (1,3)        4
```

Find the values of the nonzero elements.

```
v = nonzeros(A)
```

```
v = 4×1
```

```
     1
     2
     3
     4
```

Location and Count of Nonzeros

Use `nonzeros`, `nnz`, and `find` to locate and count nonzero matrix elements.

Create a 10-by-10 random sparse matrix with 7% density of nonzeros.

```
A = sprand(10,10,0.07);
```

Use `nonzeros` to find the values of the nonzero elements.

```
v = nonzeros(A)
```

```
v = 7×1

    0.9595
    0.4218
    0.7922
    0.8003
    0.1419
    0.9157
    0.6557
```

Use `nnz` to count the number of nonzeros.

```
n = nnz(A)
```

```
n = 7
```

Use `find` to get the indices and values of the nonzeros.

```
[i,j,v] = find(A)
```

```
i = 7×1

    10
     3
```

```
     9
     1
     2
     7
    10


j = 7×1

     2
     5
     6
    10
    10
    10
    10


v = 7×1

    0.9595
    0.4218
    0.7922
    0.8003
    0.1419
    0.9157
    0.6557
```

Input Arguments

A — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array. A can be full or sparse.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

Output Arguments

v — Nonzero elements
column vector

Nonzero elements, returned as a column vector. `v` is returned in full-storage regardless of whether `A` is full or sparse. The elements in `v` are ordered first by column subscript and then by row subscript.

`nonzeros` gives the `v`, but not the indices `i` and `j`, from `[i,j,v] = find(A)`. Generally,

```
length(v) = nnz(A) <= nzmax(A) <= prod(size(A))
```

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`find` | `isa` | `nnz` | `nzmax` | `size` | `whos`

Introduced before R2006a

norm

Vector and matrix norms

Syntax

```
n = norm(v)
n = norm(v,p)

n = norm(X)
n = norm(X,p)
n = norm(X,'fro')
```

Description

`n = norm(v)` returns the Euclidean norm on page 1-10409 of vector `v`. This norm is also called the 2-norm, vector magnitude, or Euclidean length.

`n = norm(v,p)` returns the generalized vector p-norm on page 1-10409.

`n = norm(X)` returns the 2-norm or maximum singular value of matrix `X`, which is approximately `max(svd(X))`.

`n = norm(X,p)` returns the *p*-norm of matrix `X`, where `p` is `1`, `2`, or `Inf`:

- If `p = 1`, then `n` is the maximum absolute column sum on page 1-10409 of the matrix.
- If `p = 2`, then `n` is approximately `max(svd(X))`. This is equivalent to `norm(X)`.
- If `p = Inf`, then `n` is the maximum absolute row sum on page 1-10409 of the matrix.

`n = norm(X,'fro')` returns the Frobenius norm on page 1-10410 of matrix `X`.

Examples

Vector Magnitude

Create a vector and calculate the magnitude.

```
v = [1 -2 3];
n = norm(v)
```

```
n = 3.7417
```

1-Norm of Vector

Calculate the 1-norm of a vector, which is the sum of the element magnitudes.

```
X = [-2 3 -1];
n = norm(X,1)
```

```
n = 6
```

Euclidean Distance Between Two Points

Calculate the distance between two points as the norm of the difference between the vector elements.

Create two vectors representing the (x,y) coordinates for two points on the Euclidean plane.

```
a = [0 3];
b = [-2 1];
```

Use `norm` to calculate the distance between the points.

```
d = norm(b-a)
```

```
d = 2.8284
```

Geometrically, the distance between the points is equal to the magnitude of the vector that extends from one point to the other.

$$a = 0\hat{i} + 3\hat{j}$$
$$b = -2\hat{i} + 1\hat{j}$$
$$\begin{aligned} d_{(a,b)} &= ||b - a|| \\ &= \sqrt{(-2-0)^2 + (1-3)^2} \\ &= \sqrt{8} \end{aligned}$$

2-Norm of Matrix

Calculate the 2-norm of a matrix, which is the largest singular value.

```
X = [2 0 1;-1 1 0;-3 3 0];
n = norm(X)
```

```
n = 4.7234
```

Frobenius Norm of Sparse Matrix

Use `'fro'` to calculate the Frobenius norm of a sparse matrix, which calculates the 2-norm of the column vector, `S(:)`.

```
S = sparse(1:25,1:25,1);
n = norm(S,'fro')
```

```
n = 5
```

Input Arguments

v — Input vector
vector

Input vector.

Data Types: `single` | `double`

Complex Number Support: Yes

X — Input matrix
matrix

Input matrix.

Data Types: `single` | `double`
Complex Number Support: Yes

p — Norm type
2 (default) | positive integer scalar | `Inf` | `-Inf`

Norm type, specified as `2` (default), a different positive integer scalar, `Inf`, or `-Inf`. The valid values of `p` and what they return depend on whether the first input to `norm` is a matrix or vector, as shown in the table.

Note This table does not reflect the actual algorithms used in calculations.

p	Matrix	Vector
`1`	`max(sum(abs(X)))`	`sum(abs(X))`
`2`	`max(svd(X))`	`sum(abs(X).^2)^(1/2)`
Positive, real-valued numeric `p`	—	`sum(abs(X).^p)^(1/p)`
`Inf`	`max(sum(abs(X')))`	`max(abs(X))`
`-Inf`	—	`min(abs(X))`

Output Arguments

n — Matrix or vector norm
scalar

Matrix or vector norm, returned as a scalar. The norm gives a measure of the magnitude of the elements. By convention, `norm` returns `NaN` if the input contains `NaN` values.

More About

Euclidean Norm

The Euclidean norm (also called the vector magnitude, Euclidean length, or 2-norm) of a vector `v` with `N` elements is defined by

$$\|v\| = \sqrt{\sum_{k=1}^{N} |v_k|^2}\ .$$

General Vector Norm

The general definition for the p-norm of a vector `v` that has `N` elements is

$$\|v\|_p = \left[\sum_{k=1}^{N} |v_k|^p\right]^{1/p},$$

where `p` is any positive real value, `Inf`, or `-Inf`. Some interesting values of `p` are:

- If `p = 1`, then the resulting 1-norm is the sum of the absolute values of the vector elements.
- If `p = 2`, then the resulting 2-norm gives the vector magnitude or Euclidean length of the vector.
- If `p = Inf`, then $\|v\|_\infty = \max_i(|v(i)|)$.
- If `p = -Inf`, then $\|v\|_{-\infty} = \min_i(|v(i)|)$.

Maximum Absolute Column Sum

The maximum absolute column sum of an `m`-by-`n` matrix `X` (with `m,n >= 2`) is defined by

$$\|X\|_1 = \max_{1 \le j \le n}\left(\sum_{i=1}^{m} |a_{ij}|\right).$$

Maximum Absolute Row Sum

The maximum absolute row sum of an `m`-by-`n` matrix `X` (with `m,n >= 2`) is defined by

$$\|X\|_\infty = \max_{1 \le i \le m}\left(\sum_{j=1}^{n} |a_{ij}|\right).$$

Frobenius Norm

The Frobenius norm of an `m`-by-`n` matrix `X` (with `m,n >= 2`) is defined by

$$\|X\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2} = \sqrt{\mathrm{trace}\left(X^\dagger X\right)}.$$

Tips

- Use `vecnorm` to treat a matrix or array as a collection of vectors and calculate the norm along a specified dimension. For example, `vecnorm` can calculate the norm of each column in a matrix.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cond` | `condest` | `hypot` | `normest` | `rcond` | `vecnorm`

Introduced before R2006a

normalize

Normalize data

Syntax

```
N = normalize(A)
N = normalize(A,dim)
N = normalize( ___ ,method)
N = normalize( ___ ,method,methodtype)
N = normalize( ___ ,'DataVariables',datavars)
```

Description

`N = normalize(A)` returns the vectorwise *z*-score on page 1-10418 of the data in `A` with center 0 and standard deviation 1.

- If `A` is a vector, then `normalize` operates on the entire vector.
- If `A` is a matrix, table, or timetable, then `normalize` operates on each column of data separately.
- If `A` is a multidimensional array, then `normalize` operates along the first array dimension whose size does not equal 1.

`N = normalize(A,dim)` returns the *z*-score along dimension `dim`. For example, `normalize(A,2)` normalizes each row.

`N = normalize( ___ ,method)` specifies a normalization method for either of the previous syntaxes. For example, `normalize(A,'norm')` normalizes the data in `A` by the Euclidean norm (2-norm).

`N = normalize( ___ ,method,methodtype)` specifies the type of normalization for the given `method`. For example, `normalize(A,'norm',Inf)` normalizes the data in `A` using the infinity norm.

`N = normalize( ___ ,'DataVariables',datavars)` specifies variables to operate on when the input data is in a table or timetable.

Examples

Vector and Matrix Data

Normalize data in a vector and matrix by computing the z-score.

Create a vector v and compute the z-score, normalizing the data to have mean 0 and standard deviation 1.

```
v = 1:5;
N = normalize(v)
```

```
N = 1×5

   -1.2649   -0.6325         0    0.6325    1.2649
```

Create a matrix B and compute the z-score for each column. Then, normalize each row.

```
B = magic(3)
```

```
B = 3×3

     8     1     6
     3     5     7
     4     9     2
```

```
N1 = normalize(B)
```

```
N1 = 3×3

    1.1339   -1.0000    0.3780
   -0.7559         0    0.7559
   -0.3780    1.0000   -1.1339
```

```
N2 = normalize(B,2)
```

```
N2 = 3×3

    0.8321   -1.1094    0.2774
   -1.0000         0    1.0000
```

```
   -0.2774    1.1094   -0.8321
```

Scale Data

Scale a vector A by its standard deviation.

```
A = 1:5;
Ns = normalize(A,'scale')
```

```
Ns = 1×5

    0.6325    1.2649    1.8974    2.5298    3.1623
```

Scale A so that its range is in the interval [0,1].

```
Nr = normalize(A,'range')
```

```
Nr = 1×5

         0    0.2500    0.5000    0.7500    1.0000
```

Specify Method Type

Create a vector A and normalize it by its 1-norm.

```
A = 1:5;
Np = normalize(A,'norm',1)
```

```
Np = 1×5

    0.0667    0.1333    0.2000    0.2667    0.3333
```

Center the data in A so that it has mean 0.

```
Nc = normalize(A,'center','mean')
```

```
Nc = 1×5

    -2    -1     0     1     2
```

Table Variables

Create a table containing height information for five people.

```
LastName = {'Sanchez';'Johnson';'Lee';'Diaz';'Brown'};
Height = [71;69;64;67;64];
T = table(LastName,Height)
```

```
T=5×2 table
    LastName     Height
    _________    ______

    'Sanchez'      71
    'Johnson'      69
    'Lee'          64
    'Diaz'         67
    'Brown'        64
```

Normalize the height data by the maximum height.

```
N = normalize(T,'norm',Inf,'DataVariables','Height')
```

```
N=5×2 table
    LastName     Height
    _________    _______

    'Sanchez'          1
    'Johnson'    0.97183
    'Lee'        0.90141
    'Diaz'       0.94366
    'Brown'      0.90141
```

Input Arguments

A — Input data

scalar | vector | matrix | multidimensional array | table | timetable

Input data, specified as a scalar, vector, matrix, multidimensional array, table, or timetable.

If A is a numeric array and has type `single`, then the output also has type `single`. Otherwise, the output has type `double`.

`normalize` ignores `NaN` values in A.

Data Types: `double` | `single` | `table` | `timetable`
Complex Number Support: Yes

dim — Dimension

positive integer scalar

Dimension to operate along, specified as a positive integer scalar.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

method — Normalization method

`'zscore'` (default) | `'norm'` | `'scale'` | `'range'` | `'center'`

Normalization method, specified as one of the following options:

Method	Description
`'zscore'`	z-score with mean 0 and standard deviation 1
`'norm'`	2-norm
`'scale'`	Scale by standard deviation
`'range'`	Scale range of data to [0,1]
`'center'`	Center data to have mean 0

methodtype — Method type

scalar | 2-element row vector | character vector

Method type, specified as a scalar, a 2-element row vector, or a character vector, depending on the specified method:

Method	Method Type Options	Description
`'zscore'`	`'std'` (default)	Center and scale to have mean 0 and standard deviation 1
	`'robust'`	Center and scale to have mean 0 and median absolute deviation 1
`'norm'`	Positive numeric scalar (default is 2)	*p*-norm on page 1-10418
	`Inf`	Infinity norm
`'scale'`	`'std'` (default)	Scale by standard deviation
	`'mad'`	Scale by median absolute deviation
	`'first'`	Scale by first element of data
	Numeric scalar	Scale data by numeric value
`'range'`	2-element row vector (default is [0 1])	Interval of the form `[a b]` where `a < b`
`'center'`	`'mean'`	Center to have mean 0
	`'median'`	Center to have median 0
	Numeric scalar	Shift center by numeric value

`datavars` — Table variables

scalar | vector | cell array | function handle

Table variables, specified as the comma-separated pair consisting of `'DataVariables'` and a scalar, vector, cell array, or function handle. The `'DataVariables'` value indicates which columns of the input table to operate on, and can be one of the following:

- A character vector or scalar string specifying a single table variable name
- A cell array of character vectors or string array where each element is a table variable name

- A vector of table variable indices
- A logical vector whose elements each correspond to a table variable, where `true` includes the corresponding variable and `false` excludes it
- A function handle that takes the table as input and returns a logical scalar

Example: `'Age'`

Example: `{'Height','Weight'}`

Example: `@isnumeric`

Data Types: `char` | `string` | `cell` | `double` | `single` | `logical` | `function_handle`

More About

Z-Score

For a random variable X with mean μ and standard deviation σ, the z-score of a value x is

$$z = \frac{(x-\mu)}{\sigma}.$$

For sample data with mean $\bar{X}$ and standard deviation S, the z-score of a data point x is

$$z = \frac{(x-\bar{X})}{S}.$$

z-scores measure the distance of a data point from the mean in terms of the standard deviation. The standardized data set has mean 0 and standard deviation 1, and retains the shape properties of the original data set (same skewness and kurtosis).

P-Norm

The general definition for the p-norm of a vector v that has N elements is

$$\|v\|_p = \left[\sum_{k=1}^{N} |v_k|^p\right]^{1/p},$$

where p is any positive real value, `Inf`, or `-Inf`. Some common values of p are:

- If p is 1, then the resulting 1-norm is the sum of the absolute values of the vector elements.
- If p is 2, then the resulting 2-norm gives the vector magnitude or Euclidean length of the vector.
- If p is `Inf`, then $\|v\|_\infty = \max_i(|v(i)|)$.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- Normalization methods that require calculation of the median only support vector data. This includes the methods `normalize(___,'zscore','robust')`, `normalize(___,'scale','mad')`, and `normalize(___,'center','median')`.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Tables are not supported.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`norm` | `rescale` | `vecnorm`

Introduced in R2018a

normest

2-norm estimate

Syntax

```
nrm = normest(S)
nrm = normest(S,tol)
[nrm,count] = normest(...)
```

Description

This function is intended primarily for sparse matrices, although it works correctly and may be useful for large, full matrices as well.

`nrm = normest(S)` returns an estimate of the 2-norm of the matrix `S`.

`nrm = normest(S,tol)` uses relative error `tol` instead of the default tolerance `1.e-6`. The value of `tol` determines when the estimate is considered acceptable.

`[nrm,count] = normest(...)` returns an estimate of the 2-norm and also gives the number of power iterations used.

Algorithms

The power iteration involves repeated multiplication by the matrix `S` and its transpose, `S'`. The iteration is carried out until two successive estimates agree to within the specified relative tolerance.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cond` | `condest` | `norm` | `rcond` | `svd`

Introduced before R2006a

not, ~

Find logical NOT

Syntax

```
~A
not(A)
```

Description

`~A` returns a logical array of the same size as `A`. The array contains logical `1` (`true`) values where `A` is zero and logical `0` (`false`) values where `A` is nonzero.

`not(A)` is an alternate way to execute `~A`, but is rarely used. It enables operator overloading for classes.

Examples

Logical Negation of Matrix

Create a 3-by-3 identity matrix.

```
A = eye(3)
```

```
A = 3×3

     1     0     0
     0     1     0
     0     0     1
```

Find the logical negation of `A`. The new matrix has type `logical`.

```
B = ~A
```

```
B = 3x3 logical array

   0   1   1
   1   0   1
   1   1   0
```

Conditional Code Execution

Execute code based on a condition using the logical not operator in the context of an `if` loop.

Create a logical variable `A`.

```
A = false;
```

Use `A` to write an if/else code block. Wrap the if/else block in a `for` loop so that it executes four times.

```
for k = 1:4
    if ~A
        disp('IF block')
        A = true;
    else
        disp('ELSE block')
    end
end
```

```
IF block
ELSE block
ELSE block
ELSE block
```

On the first iteration, `A` is `false`, so the `if` block executes since `~A` is `true`. However, the `if` block also changes the value of `A` to `true`. In the remaining iterations, `~A` is `false` and the `else` block executes.

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a numeric scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

Tips

- You also can use the ~ symbol as a placeholder output argument in a function call. For example, `[~,i] = max(A)` suppresses the first output of the `max` function, returning only the indices of the maximum values. For more information, see “Ignore Function Inputs”.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`all` | `and` | `any` | `bitcmp` | `or` | `xor`

Topics

“Operator Precedence”
“MATLAB Operators and Special Characters”

Introduced before R2006a

notebook

(Removed) Open MATLAB Notebook in Microsoft Word software (on Microsoft Windows platforms)

Note `notebook` has been removed. To create a document that combines code, formatted text, and output, use the Live Editor instead.

Syntax

```
notebook
notebook('filename')
notebook('-setup')
```

Description

`notebook` starts Microsoft Word software and creates a new MATLAB Notebook titled `Document 1`.

Note The `notebook` command is available only on Windows systems that have a 32–bit version of Microsoft Word installed. The `notebook` command is not available for 64–bit versions of Microsoft Word.

`notebook('filename')` starts Microsoft Word and opens the notebook `filename`, where `filename` is either in the MATLAB current folder or is a full path. If `filename` does not exist, MATLAB creates a new notebook titled `filename`. If the file name extension is not specified, MATLAB assumes `.doc`.

`notebook('-setup')` runs an interactive setup function for MATLAB Notebook. It copies the notebook template, `m-book.dot`, to the Microsoft Word template folder, whose location MATLAB automatically determines from the Windows system registry. Upon completion, MATLAB displays a message indicating whether or not the setup was successful.

See Also

Topics

"What Is a Live Script or Function?"

Introduced before R2006a

now

Current date and time as serial date number

Syntax

```
t = now
```

Description

`t = now` returns the current date and time as a serial date number. A serial date number represents the whole and fractional number of days starting from a fixed, preset date (January 0, 0000 on page 1-10430).

Examples

Return and Convert Serial Date Numbers

Change the output display for numbers to long, fixed-decimal format. Then return the current date and time as a serial date number.

```
format longG
t = now
```

```
t =
          737664.113578692
```

The whole part of `t` corresponds to the date, and the fractional part corresponds to the time of day. One way to show the date and time is to convert `t` using the `datetime` function.

```
d = datetime(t,'ConvertFrom','datenum')
```

```
d = datetime
   27-Aug-2019 02:43:33
```

To represent the date alone, without the time of day, use the `floor` function. Convert the result to a `datetime` value for display.

```
t2 = floor(t)
```

```
t2 =
      737664
```

```
d2 = datetime(t2,'ConvertFrom','datenum')
```

```
d2 = datetime
   27-Aug-2019 00:00:00
```

Limitations

- MATLAB Online returns the current date and time in Coordinated Universal Time (UTC) rather than local time.

More About

January 0, 0000

Date specified to include a year 0 in the proleptic Gregorian calendar. For more information, see January 0 and Year zero.

Tips

- To represent the current date and time as a serial date number, as text, or as a `datetime` value, use the function calls shown in the table.

Function Call	Output
`floor(now)`	Current date as a serial date number
`rem(now,1)`	Current time as a serial date number
`datestr(now)`	Current date and time as text
`char(datetime('now'))`	
`string(datetime('now'))`	
`datetime('now')`	Current date and time as a `datetime` value
`datetime(now,'ConvertFrom','datenum')`	

See Also

`char` | `date` | `datenum` | `datestr` | `datetime` | `string`

Topics

"Represent Dates and Times in MATLAB"
"Convert Between Datetime Arrays, Numbers, and Text"

Introduced before R2006a

nsidedpoly

Regular polygon

Syntax

```
pgon = nsidedpoly(n)
pgon = nsidedpoly(n,Name,Value)
```

Description

`pgon = nsidedpoly(n)` returns a regular polygon with `n` equal-length sides. The center of `pgon` is at the point (0,0), and the circumscribed circle of the polygon has radius 1.

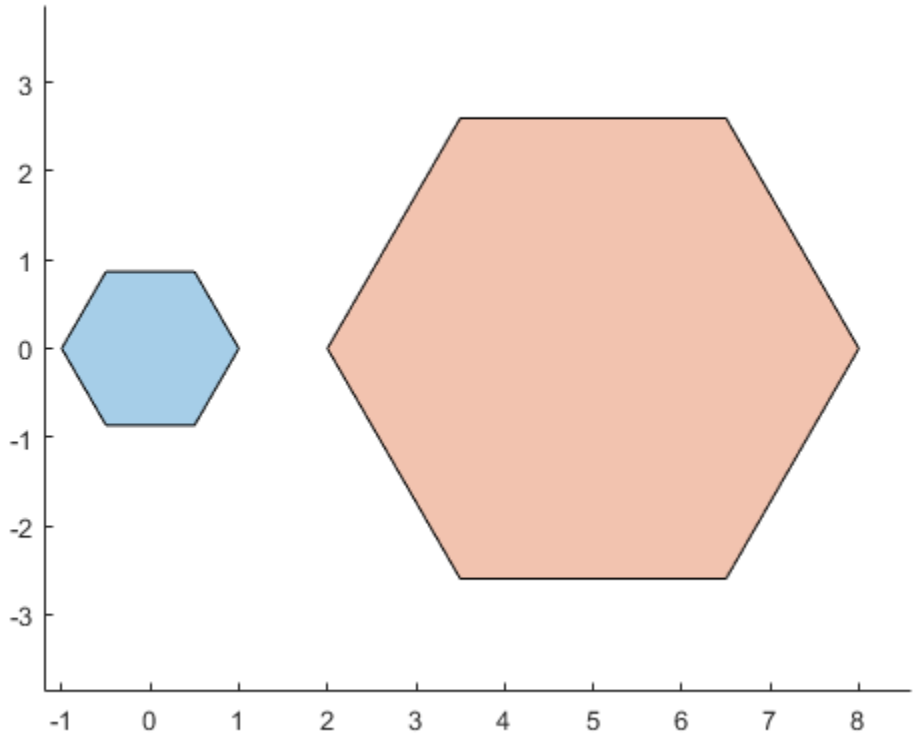
`pgon = nsidedpoly(n,Name,Value)` specifies additional properties of the polygon using one or more name-value pair arguments. For example, `pgon = nsidedpoly(4,'SideLength',5)` creates a square centered at (0,0) with sides of length 5.

Examples

Hexagon

Create a hexagon with sides of length 1 centered at the point (0,0). Then, create a hexagon with sides of length 3 centered at (5,0).

```
pgon1 = nsidedpoly(6);
pgon2 = nsidedpoly(6,'Center',[5 0],'SideLength',3);
plot([pgon1 pgon2])
axis equal
```



Input Arguments

n — Number of sides
positive scalar integer

Number of sides of the polygon, specified as a positive scalar integer greater than 2.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `pgon = nsidedpoly(3,'Radius',10)`

`Center` — Center point
1-by-2 row vector

Center point of the polygon, specified as the comma-separated pair consisting of `'Center'` and a 1-by-2 row vector whose first element is the *x*-coordinate of the point and whose second element is the *y*-coordinate.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Radius` — Radius
positive scalar

Radius of the circumscribed circle of the polygon, specified as the comma-separated pair consisting of `'Radius'` and a positive scalar. This name-value pair cannot be combined with the `'SideLength'` name-value pair.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`SideLength` — Side length
positive scalar

Side length of polygon, specified as the comma-separated pair consisting of `'SideLength'` and a positive scalar. This name-value pair cannot be combined with the `'Radius'` name-value pair.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

See Also

`alphaShape` | `polyshape`

Introduced in R2017b

nthroot

Real nth root of real numbers

Syntax

```
Y = nthroot(X,N)
```

Description

`Y = nthroot(X,N)` returns the real nth root of the elements of `X`. Both `X` and `N` must be real scalars or arrays of the same size. If an element in `X` is negative, then the corresponding element in `N` must be an odd integer.

Examples

Calculate Real Root of Negative Number

Find the real cube root of `-27`.

```
nthroot(-27, 3)
```

```
ans = -3
```

For comparison, also calculate `(-27)^(1/3)`.

```
(-27)^(1/3)
```

```
ans = 1.5000 + 2.5981i
```

The result is the complex cube root of `-27`.

Calculate Several Real Roots of Scalar

Create a vector of roots to calculate, N.

```
N = [5 3 -1];
```

Use `nthroot` to calculate several real roots of `-8`.

```
Y = nthroot(-8,N)
```

```
Y = 1×3

   -1.5157   -2.0000   -0.1250
```

The result is a vector of the same size as N.

Element-wise Roots of Matrix

Create a matrix of bases, X, and a matrix of nth roots, N.

```
X = [-2 -2 -2; 4 -3 -5]
```

```
X = 2×3

    -2    -2    -2
     4    -3    -5
```

```
N = [1 -1 3; 1/2 5 3]
```

```
N = 2×3

    1.0000   -1.0000    3.0000
    0.5000    5.0000    3.0000
```

Each element in X corresponds to an element in N.

Calculate the real nth roots of the elements in X.

```
Y = nthroot(X,N)
```

```
Y = 2×3

   -2.0000   -0.5000   -1.2599
   16.0000   -1.2457   -1.7100
```

Except for the signs (which are treated separately), the result is comparable to `abs(X).^(1./N)`. By contrast, you can calculate the complex roots using `X.^(1./N)`.

Input Arguments

`X` — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. `X` can be either a scalar or an array of the same size as `N`. The elements of `X` must be real.

Data Types: `single` | `double`

`N` — Roots to calculate
scalar | array of same size as `X`

Roots to calculate, specified as a scalar or array of the same size as `X`. The elements of `N` must be real. If an element in `X` is negative, the corresponding element in `N` must be an odd integer.

Data Types: `single` | `double`

Tips

- While `power` is a more efficient function for computing the roots of numbers, in cases where both real and complex roots exist, `power` returns only the complex roots. In these cases, use `nthroot` to obtain the real roots.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`power` | `sqrt`

Introduced before R2006a

null

Null space of matrix

Syntax

```
Z = null(A)
Z = null(A,'r')
```

Description

`Z = null(A)` returns an orthonormal basis for the null space of `A`.

`Z = null(A,'r')` returns a "rational" basis for the null space of `A` that is typically not orthonormal. If `A` is a small matrix with small integer elements, then the elements of `Z` are ratios of small integers. This method is numerically less accurate than `null(A)`.

Examples

Null Space of Matrix

Use the `null` function to calculate orthonormal and rational basis vectors for the null space of a matrix. The null space of a matrix contains vectors *x* that satisfy Ax = 0.

Create a 4-by-4 magic square matrix. This matrix is rank deficient, with one of the singular values being equal to zero.

```
A = magic(4)
```

```
A = 4×4

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Calculate an orthonormal basis for the null space of A. Confirm that $Ax_1 = 0$, within roundoff error.

```
x1 = null(A)
```

```
x1 = 4×1

    0.2236
    0.6708
   -0.6708
   -0.2236
```

```
norm(A*x1)
```

```
ans = 4.4019e-15
```

Now calculate a rational basis for the null space. Confirm that $Ax_2 = 0$.

```
x2 = null(A,'r')
```

```
x2 = 4×1

    -1
    -3
     3
     1
```

```
norm(A*x2)
```

```
ans = 0
```

x1 and x2 are similar, but are normalized differently.

General Solution of Underdetermined System of Equations

Find one particular solution to an underdetermined system, and then obtain the general form for all solutions.

Underdetermined linear systems Ax = *b* involve more unknowns than equations. An underdetermined system can have infinitely many solutions or no solution. When the

system has infinitely many solutions, they all lie on a line. The points on the line are all obtained with linear combinations of the null space vectors.

Create a 2-by-4 coefficient matrix and use backslash to solve the equation $Ax_0 = b$, where b is a vector of ones. Backslash calculates a least-squares solution to the problem.

```
A = [1 8 15 67; 7 14 16 3]
```

```
A = 2×4

     1     8    15    67
     7    14    16     3
```

```
b = ones(2,1);
x0 = A\b
```

```
x0 = 4×1

         0
         0
    0.0623
    0.0010
```

The complete general solution to the underdetermined system has the form $x = x_0 + Ny$, where:

- N is the null space of A.
- y is any vector of proper length.
- x_0 is the solution computed by backslash.

Calculate the null space of A, and then use the result to construct another solution to the system of equations. Check that the new solution satisfies Ax = b, up to roundoff error.

```
N = null(A)
```

```
N = 4×2

   -0.2977   -0.8970
   -0.6397    0.4397
    0.7044    0.0157
   -0.0769   -0.0426
```

```
x = x0 + N*[1; -2]
```

```
x = 4×1

    1.4963
   -1.5192
    0.7354
    0.0093
```

```
norm(A*x-b)
```

```
ans = 1.8291e-14
```

Input Arguments

A — Input matrix
matrix

Input matrix.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Z — Null space basis vectors
matrix

Null space basis vectors, returned in the columns of a matrix. `Z` satisfies the properties:

- `A*Z` has negligible elements.
- `size(Z,2)` is an estimate of the nullity of `A`.

Algorithms

`null(A)` calculates the singular value decomposition of the matrix, `[U,S,V] = svd(A,0)`. The columns of `V` that do not correspond to nonzero singular values form a set of orthonormal basis vectors for the null space.

The "rational" basis for the null space `null(A,'r')` is obtained from the reduced row echelon form of `A`, as calculated by `rref`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Generated code might return a different basis than MATLAB
- Code generation does not support the rational basis option (second input).
- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The syntax `Z = null(A, 'r')` is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`orth` | `rank` | `rref` | `svd`

Introduced before R2006a

numArgumentsFromSubscript

Number of arguments for customized indexing methods

Syntax

```
n = numArgumentsFromSubscript(obj,s,indexingContext)
```

Description

`n = numArgumentsFromSubscript(obj,s,indexingContext)` returns the number of expected inputs to `subsasgn` or the number of expected outputs from `subsref`.

Overload `numArgumentsFromSubscript` to describe the number of values to return from indexing expressions that return or assign to a comma-separated list. That is, indexing expressions that end in `'{}'` or `'.'` indexing. The `numArgumentsFromSubscript` function can:

- Access the indexing operations and indices used in the indexing expression.
- Determine if an indexing operation is made in the context of a reference statement, an expression passed to a function, or an assignment.

If a class overloads `numArgumentsFromSubscript`, MATLAB calls it to determine the number of array elements involved in an indexing operation when the number of elements is greater than one. For example, these `'.'` indexing operations generate a call to `numArgumentsFromSubscript`:

- `objArray.a` — Number of elements referenced in a statement (`Statement`)
- `func(objArray.a)` — Number of elements returned in an expression (`Expression`)
- `[objArray.a] = rhs` — Number of values assigned with a comma-separated list (`Assignment`)

MATLAB uses the calling context to determine when to apply the value returned by `numArgumentsFromSubscript`. Your implementation of `numArgumentsFromSubscript` can provide different outputs for the three types of indexing statements. For example, this overload of `numArgumentsFromSubscript`:

- Changes the expected number of output arguments from `subsref` for indexing expressions that are passed to functions.
- Uses the indexing substructure `s` to determine the number of arguments required by the indexing operation

```
function n = numArgumentsFromSubscript(obj,s,indexingContext)
   if indexingContext == matlab.mixin.util.IndexingContext.Expression
      n = 1;
   else
      n = length(s(1).subs{:});
   end
end
```

Implement the `subsref` method with a `varargout` output to enable MATLAB to call this method with the specified number of output arguments.

```
function varargout = subsref(obj,s)
   ...
end
```

Examples

Indexing with Character Array Keywords

Use words as indices for object properties.

By default, MATLAB treats each character in a character array as a separate element. Therefore, MATLAB interprets a character array index as multiple index values.

Without overloading `numArgumentsFromSubscript`, keyword indexing statements request too many arguments for `subsref` and `subsasgn` methods. To implement indexing that uses character arrays as a single keyword, define a `numArgumentsFromSubscript` method to return `1` as the number of elements referenced or assigned.

Overload `numArgumentsFromSubscript` to support the following kinds of indexed reference and assignment.

Index reference with character array:

```
obj{'keyword'}
```

Index reference in function argument:

```
func(obj{'keyword'})
```

Assign comma-separated list to object array:

```
[obj{'keyword'}] = right-side values;
```

In these cases, `subsref` and `subsasgn` cannot determine the number of required output and input arguments, respectively, from the calling context. Provide this information using the `numArgumentsFromSubscript` function.

Define the `KeyIndex` class with two properties. `KeyCell` stores key names. `DataArray` stores corresponding data. To customize indexing:

- Implement a `subsref` method that returns the data values that correspond to the specified keyword.
- Implement a `subsasgn` method that assigns the specified values to the corresponding element in `DataArray` properties.
- Overload `numArgumentsFromSubscript` to return a value of `1` or scalar objects and a value of `numel(objArray)` for object arrays.

Specify a cell array containing the names of months for the `KeyCell` property. Specify a numeric array of the snow fall for the corresponding months in the `DataArray` property.

```
snow15 = KeyIndex({'Jan','Feb','March'},[36 42 2])
```

```
snow15 =

  KeyIndex with properties:

      KeyCell: {'Jan'  'Feb'  'March'}
    DataArray: [36 42 2]
```

Reference the snow amount for a particular month using the keyword:

```
snow15{'Feb'}
```

```
ans =

    42
```

Assign to the corresponding element using `'{}'` indexing with the keyword:

```
snow15{'Feb'} = 52;
```

The value corresponding to the key `'Feb'` changes:

```
snow15{'Feb'}
```

```
ans =

    52
```

Create an array with the snow fall for 2 years:

```
snow14 = KeyIndex({'Jan','Feb','March'},[12 8 2]);
sTotal = [snow14,snow15];
```

Show the snow fall for both years for February:

```
sTotal{'Feb'}
```

```
ans =

     8


ans =

    52
```

Update the values of snow fall for both years for the month of February. Indexing into a cell array returns a comma-separated list:

```
c{1} = 34;
c{2} = 56;
[sTotal{'Feb'}] = c{:};
```

Here is the `KeyIndex` class. This class shows a particular programming technique. It does not include error checking and other features that your class can require.

```
classdef KeyIndex
   properties
      KeyCell
      DataArray
   end
   methods
      function obj = KeyIndex(key,data)
```

```
    if nargin > 0
       obj.KeyCell = key;
       obj.DataArray = data;
    end
end
function n = numArgumentsFromSubscript(obj,~,~)
    n = numel(obj);
end
function varargout = subsref(obj,s)
    a = numel(obj);
    switch s(1).type
       case '{}'
          varargout = cell(1,a);
          for j = 1:a
             for k = 1:numel(obj(j).KeyCell)
                if strcmp(s.subs(:),obj(j).KeyCell{k})
                   varargout{j} = obj(j).DataArray(k);
                end
             end
          end
       case '.'
          varargout = {builtin('subsref',obj,s)};
       case '()'
          varargout = {builtin('subsref',obj,s)};
    end
end
function obj = subsasgn(obj,s,varargin)
    a = numel(varargin);
    switch s(1).type
       case '{}'
          for j = 1:a
             for k = 1:numel(obj(j).KeyCell)
                if strcmp(s.subs(:),obj(j).KeyCell{k})
                   obj(j).DataArray(k) = varargin{j};
                end
             end
          end
       case '.'
          obj = builtin('subsasgn',obj,s,varargin{:});
       case '()'
          obj = builtin('subsasgn',obj,s,varargin{:});
    end
end
```

```
    end
end
```

Reference Property Value Once Per Array

Define a class that supports per-element and per-array indexing for properties.

The `numArgumentsFromSubscript` function enables classes to customize references to specific properties. The indexing substructure argument (`s`) to `numArgumentsFromSubscript` contains the name of the property referenced in a dot indexing expression (that is, a reference of the form `obj.PropertyName`). Using the indexing substructure, `numArgumentsFromSubscript` can return a unique value for any given property.

The `PerArray` class uses the information in the indexing substructure to specify the number of outputs for each property reference.

Overload `numArgumentsFromSubscript` to return:

- A comma-separated list of values for the `ByElement` property, one value per array element
- A single value for the `ByArray` property for the entire array

Create an array of `PerArray` objects:

```
for k = 1:4
   pa(k) = PerArray(rand(k));
end
```

Query the `ByElement` property to return a comma-separated list of property values:

```
pa.ByElement
```

```
ans =

    0.7513


ans =

    0.2551    0.6991
    0.5060    0.8909
```

```
ans =

    0.9593    0.1493    0.2543
    0.5472    0.2575    0.8143
    0.1386    0.8407    0.2435


ans =

    0.9293    0.6160    0.5853    0.7572
    0.3500    0.4733    0.5497    0.7537
    0.1966    0.3517    0.9172    0.3804
    0.2511    0.8308    0.2858    0.5678
```

Query the `ByArray` property to return a single value for the array:

```
pa.ByArray
```

```
ans =

    3.1416
```

Here is the `PerArray` class. This class shows a particular programming technique. It does not include error checking and other features that your class can require.

```
classdef PerArray
   properties
      ByElement
   end
   properties (Constant)
      ByArray = pi;
   end
   methods
      function obj = PerArray(be)
         if nargin > 0
            obj.ByElement = be;
         end
      end
      function n = numArgumentsFromSubscript(obj,s,indexingContext)
         import matlab.mixin.util.IndexingContext
         if (indexingContext ~= IndexingContext.Assignment)
            if length(s) < 2
               switch (s.subs(:)')
```

```
                    case 'ByElement'
                       n = numel(obj);
                    case 'ByArray'
                       if strcmp(s(1).type,'.')
                          n = 1;
                       end
                 end
              elseif length(s) > 1
                 n = length([s(1).subs{:}]);
              end
           end
        end
        function varargout = subsref(obj,s)

           switch s(1).type
              case '.'
                 varargout = cell(1,numel(obj));
                 for k = 1:numel(obj)
                    varargout{k} = obj(k).(s.subs(:));
                 end
              case '()'
                 if length([s(1).subs{:}]) > 1
                    ind = [s(1).subs{:}];
                    numInd = length(ind);
                    varargout = cell(1,numInd);
                    for k = 1:numInd
                       varargout{k} = obj(ind(k)).(s(2).subs(:));
                    end
                 else
                    varargout = {builtin('subsref',obj,s)};
                 end
           end
        end
     end
end
```

Input Arguments

obj — Object of overloading class
object

Object of overloading class used in the indexing operation. The class of this object determines which `subsref` or `subsasgn` method MATLAB calls as a result of an indexing operation.

`s` — Indexing structure
`struct`

Indexing structure or array of indexing structures containing information about the specific indexing expression. Each structure has two fields:

- `type` — Indexing expression can be `'()'`, `'{}'`, `'.'`
- `subs` — Subscript values (property name or cell array of index numbers)

Data Types: `struct`

`indexingContext` — Context in which result applies
`matlab.mixin.util.IndexingContext`

Context in which result applies, specified as one of these enumerations:

- `matlab.mixin.util.IndexingContext.Statement` — Indexed reference used as a statement (`obj.a`)
- `matlab.mixin.util.IndexingContext.Expression` — Indexed reference used as an argument to a function (`func(obj.a)`)
- `matlab.mixin.util.IndexingContext.Assignment` — Indexed assignment (`[obj.a] = x`).

Output Arguments

`n` — Number of arguments
scalar integer

Number of arguments returned by overloaded `subsref` or passed to overloaded `subsasgn`. Overload `numArgumentsFromSubscript` to return the values required by your class for various indexing scenarios.

Tips

- Overload `numArgumentsFromSubscript` instead of `numel` to control the results from overloaded `subsref` and `subsasgn`. Overloading `numArgumentsFromSubscript` can avoid errors caused by overloading `numel`.

See Also

`subsasgn` | `subsref`

Topics

"Number of Arguments for subsref and subsasgn"
"Modify nargout and nargin for Indexing Methods"
"Comma-Separated Lists"

Introduced in R2015b

num2cell

Convert array to cell array with consistently sized cells

Syntax

```
C = num2cell(A)
C = num2cell(A,dim)
```

Description

`C = num2cell(A)` converts array `A` into cell array `C` by placing each element of `A` into a separate cell in `C`.

The `num2cell` function converts an array that has **any** data type—even a nonnumeric type.

`C = num2cell(A,dim)` splits the contents of `A` into separate cells of `C`, where `dim` specifies which dimensions of `A` to include in each cell. `dim` can be a scalar or a vector of dimensions. For example, if `A` has 2 rows and 3 columns, then:

- `num2cell(A,1)` creates a 1-by-3 cell array `C`, where each cell contains a 2-by-1 column of `A`.
- `num2cell(A,2)` creates a 2-by-1 cell array `C`, where each cell contains a 1-by-3 row of `A`.
- `num2cell(A,[1 2])` creates a 1-by-1 cell array `C`, where the cell contains the entire array `A`.

Examples

Convert Arrays to Cell Array

Place all elements of a numeric array into separate cells.

```
a = magic(3)
```

```
a = 3×3

     8     1     6
     3     5     7
     4     9     2
```

```
c = num2cell(a)
```

```
c=3×3 cell
    {[8]}    {[1]}    {[6]}
    {[3]}    {[5]}    {[7]}
    {[4]}    {[9]}    {[2]}
```

Place individual letters of a word into separate cells of an array.

```
a = ['four';'five';'nine']
```

```
a = 3x4 char array
    'four'
    'five'
    'nine'
```

```
c = num2cell(a)
```

```
c = 3x4 cell array
    {'f'}    {'o'}    {'u'}    {'r'}
    {'f'}    {'i'}    {'v'}    {'e'}
    {'n'}    {'i'}    {'n'}    {'e'}
```

Create Cell Array of Numeric Arrays

Generate a 4-by-3-by-2 numeric array, and then create a 1-by-3-by-2 cell array of 4-by-1 column vectors.

```
A = reshape(1:12,4,3);
A(:,:,2) = A*10
```

```
A =
A(:,:,1) =

     1     5     9
     2     6    10
     3     7    11
     4     8    12


A(:,:,2) =

    10    50    90
    20    60   100
    30    70   110
    40    80   120
```

```
C = num2cell(A,1)
```

```
C = 1x3x2 cell array
C(:,:,1) =

    {4x1 double}    {4x1 double}    {4x1 double}


C(:,:,2) =

    {4x1 double}    {4x1 double}    {4x1 double}
```

Each 4-by-1 vector contains elements from along the *first* dimension of A:

```
C{1}
```

```
ans = 4×1

     1
     2
     3
     4
```

Create a 4-by-1-by-2 cell array of 1-by-3 numeric arrays.

```
C = num2cell(A,2)
```

```
C = 4x1x2 cell array
C(:,:,1) =

    {1x3 double}
    {1x3 double}
    {1x3 double}
    {1x3 double}


C(:,:,2) =

    {1x3 double}
    {1x3 double}
    {1x3 double}
    {1x3 double}
```

Each 1-by-3 row vector contains elements from along the *second* dimension of A:

```
C{1}
```

```
ans = 1×3

     1     5     9
```

Finally, create a 4-by-3 cell array of 1-by-1-by-2 numeric arrays.

```
C = num2cell(A,3)
```

```
C=4×3 cell
    {1x1x2 double}    {1x1x2 double}    {1x1x2 double}
    {1x1x2 double}    {1x1x2 double}    {1x1x2 double}
    {1x1x2 double}    {1x1x2 double}    {1x1x2 double}
    {1x1x2 double}    {1x1x2 double}    {1x1x2 double}
```

Each 1-by-1-by-2 vector contains elements from along the *third* dimension of A:

```
C{1}
```

```
ans =
ans(:,:,1) =

     1
```

```
ans(:,:,2) =

    10
```

Combine Across Multiple Dimensions

Create a cell array by combining elements into numeric arrays along several dimensions.

```
A = reshape(1:12,4,3);
A(:,:,2) = A*10
```

```
A =
A(:,:,1) =

     1     5     9
     2     6    10
     3     7    11
     4     8    12


A(:,:,2) =

    10    50    90
    20    60   100
    30    70   110
    40    80   120
```

```
c = num2cell(A,[1 3])
```

```
c=1×3 cell
    {4x1x2 double}    {4x1x2 double}    {4x1x2 double}
```

Each 4-by-1-by-2 array contains elements from along the first and third dimension of A:

```
c{1}
```

```
ans =
ans(:,:,1) =
```

```
     1
     2
     3
     4


ans(:,:,2) =

    10
    20
    30
    40
```

```
c = num2cell(A,[2 3])
```

```
c=4×1 cell
    {1x3x2 double}
    {1x3x2 double}
    {1x3x2 double}
    {1x3x2 double}
```

Input Arguments

A — Input
any type of multidimensional array

Input, specified as any type of multidimensional array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `cell` | `categorical` | `datetime` | `duration` | `calendarDuration` | `function_handle`

`dim` — Dimension of `A`
positive integer | positive vector of integers

Dimension of `A`, specified as a positive integer or a vector of positive integers. `dim` must be between 1 and `ndims(A)`.

Elements do not need to be in numeric order. However, `num2cell` permutes the dimensions of the arrays in each cell of `C` to match the order of the specified dimensions.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

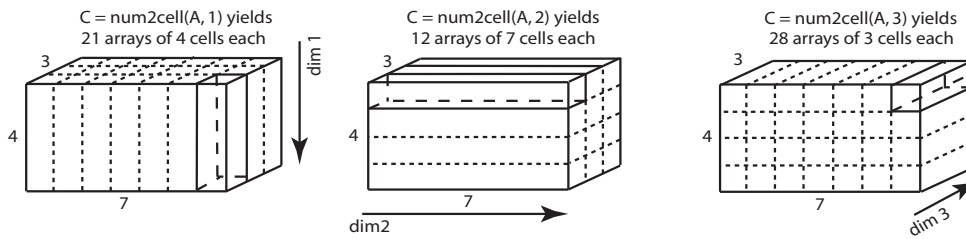
Output Arguments

`C` — Resulting array
cell array

Resulting array, returned as a cell array. The size of `C` depends on the size of `A` and the values of `dim`.

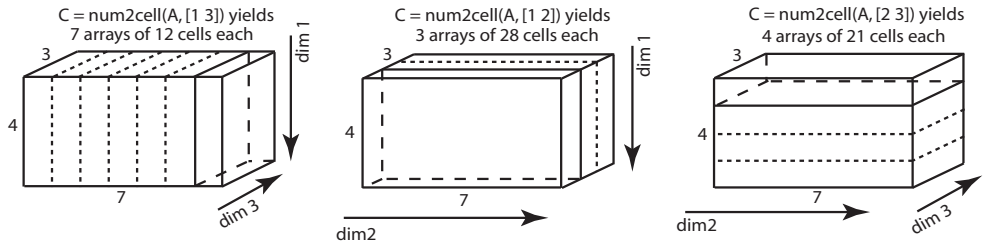
- If `dim` is not specified, then `C` is the same size as `A`.
- If `dim` is a scalar, then `C` contains `numel(A)/size(A,dim)` cells. If `dim` is 1 or 2, then each cell contains a column or row vector, respectively. If `dim` > 2, then each cell contains an array whose `dim`th dimensional length is `size(A,dim)`, and whose other dimensions are all singletons.

  For example, given a 4-by-7-by-3 array, `A`, this figure shows how `num2cell` creates cells corresponding to `dim` values of `1`, `2`, and `3`.



- If `dim` is a vector containing `N` values, then `C` has `numel(A)/prod([size(A,dim(1)),...,size(A,vdim(N))])` cells. Each cell contains an array whose `dim(i)`th dimension has a length of `size(A,dim(i))` and whose other dimensions are singletons.

  For example, given a 4-by-7-by-3 array, you can specify `dim` as a positive integer vector to create cell arrays of different dimensions.



Data Types: `cell`

Extended Capabilities

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cat` | `cell2mat` | `mat2cell`

Introduced before R2006a

num2hex

Convert single- and double-precision numbers to IEEE hexadecimal format

Syntax

```
hexStr = num2hex(X)
```

Description

`hexStr = num2hex(X)` returns a representation of `X` in IEEE format using hexadecimal digits.

- If `X` is a double-precision number, then `hexStr` has 16 characters.
- If `X` is a single-precision number, then `hexStr` has 8 characters.
- If `X` is a vector, matrix, or multidimensional array, then `hexStr` is a two-dimensional character array. Each row of `hexStr` represents an element of `X`.

Unlike the `dec2hex` function, `num2hex` can convert floating-point values with fractional parts. Also, `num2hex` handles `NaN` and `Inf` values and denormal numbers correctly.

The format produced by `num2hex` is identical to the one produced by the `format hex` command.

Examples

Convert Double-Precision Number with Fractional Part

Create a number and represent it in its IEEE® format using hexadecimal digits. `X` is a double-precision number and has a fractional part.

```
X = 3.1416;
hexStr = num2hex(X)
```

```
hexStr =
'400921ff2e48e8a7'
```

Since `hexStr` is in IEEE format, it also represents the fractional part of `X`. You can convert `hexStr` back to a number using the `hex2num` function.

```
hex2num(hexStr)
```

```
ans = 3.1416
```

Convert Single-Precision Number

Create a single-precision number.

```
X = single(2.7182818)
```

```
X = single
    2.7183
```

Convert `X`. The `num2hex` function represents a single-precision number using only eight hexadecimal digits.

```
hexStr = num2hex(X)
```

```
hexStr =
'402df854'
```

Represent Negative Number, NaN, and Inf

Convert a negative number and represent it in IEEE® format.

```
num2hex(-1)
```

```
ans =
'bff0000000000000'
```

Convert `NaN`.

```
num2hex(NaN)
```

```
ans =
'fff8000000000000'
```

Convert `Inf`.

```
num2hex(Inf)
```

```
ans =
'7ff0000000000000'
```

Represent Array of Floating-Point Numbers

Create a numeric array.

```
X = [3.1416 1.37e-5 1023.92];
```

Convert the elements of `X`. The `num2hex` function returns a character array that represents these values.

```
hexStr = num2hex(X)
```

```
hexStr = 3x16 char array
    '400921ff2e48e8a7'
    '3eecbb21a99df39b'
    '408fff5c28f5c28f'
```

Input Arguments

X — Input array
numeric array

Input array, specified as a numeric array.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`dec2hex` | `format` | `hex2num`

Topics

"Hexadecimal and Binary Values"

External Websites

IEEE 754

Introduced before R2006a

num2ruler

Convert numeric data for use with specific ruler

Syntax

```
data = num2ruler(num,ruler)
```

Description

`data = num2ruler(num,ruler)` converts the numeric values in `num` to the appropriate values for the specified ruler. The resulting values depend on both the type and limits of the `ruler` input. The output `data` is an array the same size as `num`.

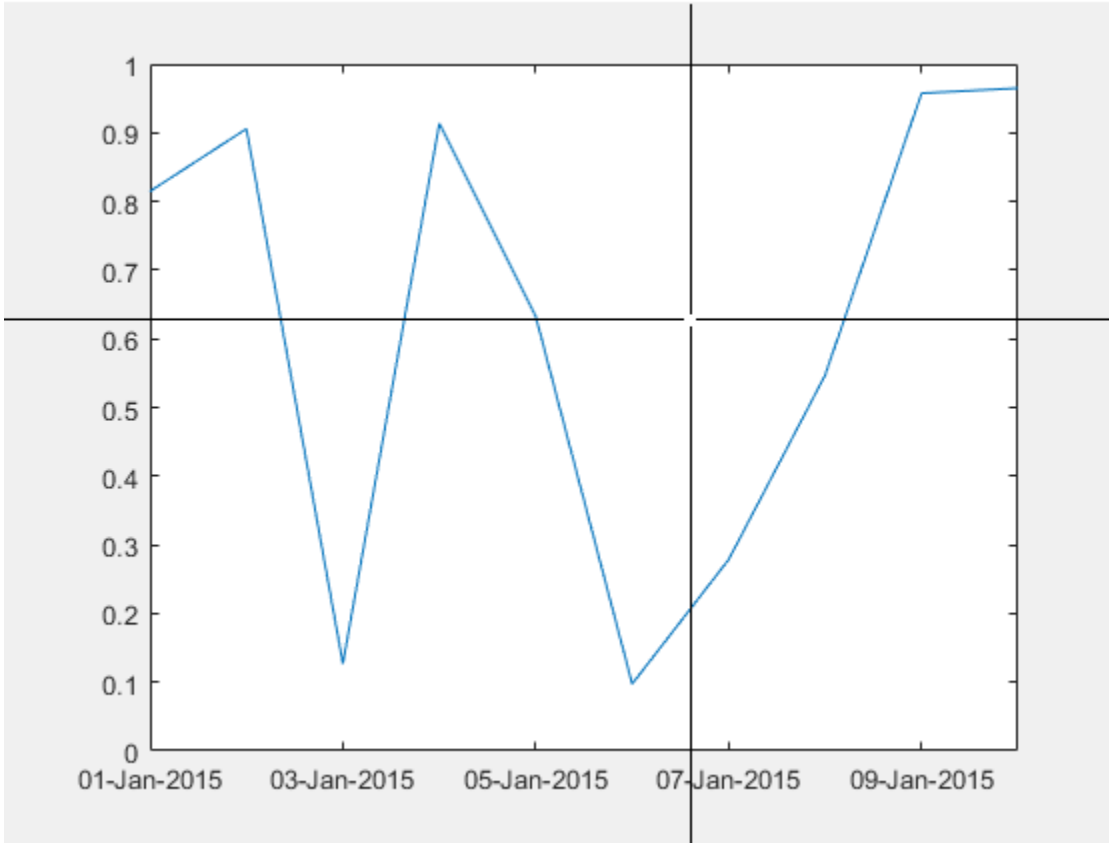
Examples

Convert Numeric Value to Equivalent Date Value

Convert the output of `ginput` from a numeric value to the equivalent date value.

Create a plot with dates along the *x*-axis. Then, click a point in the axes and use the `ginput` function to return the location of the selected point.

```
t = datetime(2015,1,1:10);
y = rand(10,1);
plot(t,y)
ax = gca;
[x,y] = ginput(1)
```




```
x =

    5.8168


y =

    0.6458
```

The `ginput` function returns the location as numeric values. Use `num2ruler` to determine the date value that is equivalent to the numeric value returned for `x`.

```
xdate = num2ruler(x,ax.XAxis)
```

```
xdate =

   06-Jan-2015 19:36:13
```

Input Arguments

num — Numeric data
array

Numeric data, specified as an array.

Example: `num = [1 2 3]`

ruler — Ruler that determines data mapping
`DurationRuler` object | `DatetimeRuler` object | `NumericRuler` object | `GeographicRuler` object

Ruler that determines data mapping, specified as one of these types of ruler objects:

- `DurationRuler` object — Convert the input `num` to duration values.
- `DatetimeRuler` object — Convert the input `num` to datetime values.
- `NumericRuler` object — No conversion. For this case, the output `data` is equal to the input `num`.
- `GeographicRuler` object — No conversion. For this case, the output `data` is equal to the input `num`.

Specify the ruler object by referring to either the `XAxis`, `YAxis`, or `ZAxis` property of the `Axes` object.

```
ax = gca;
data = num2ruler(num,ax.XAxis)
```

See Also

`axes` | `datetime` | `duration` | `ruler2num`

Introduced in R2016b

num2str

Convert numbers to character array

Syntax

```
s = num2str(A)
s = num2str(A,precision)
s = num2str(A,formatSpec)
```

Description

`s = num2str(A)` converts a numeric array into a character array that represents the numbers. The output format depends on the magnitudes of the original values. `num2str` is useful for labeling and titling plots with numeric values.

`s = num2str(A,precision)` returns a character array that represents the numbers with the maximum number of significant digits specified by `precision`.

`s = num2str(A,formatSpec)` applies a format specified by `formatSpec` to all elements of `A`.

Examples

Default Conversions of Floating-Point Values

Convert the floating-point values returned by `pi` and `eps` to character vectors.

```
s = num2str(pi)
```

```
s =
'3.1416'
```

```
s = num2str(eps)
```

```
s =
'2.2204e-16'
```

Specify Precision

Specify the maximum number of significant digits for floating-point values.

```
A = gallery('normaldata',[2,2],0);
s = num2str(A,3)
```

```
s = 2x16 char array
    '-0.433     0.125'
    ' -1.67     0.288'
```

Specify Formatting

Specify the width, precision, and other formatting for an array of floating-point values.

```
A = gallery('uniformdata',[2,3],0) * 9999;
s = num2str(A,'%10.5e\n')
```

```
s = 2x35 char array
    '9.50034e+03...'
    '2.31115e+03...'
```

The format `'%10.5e'` prints each value in exponential format with five decimal places, and `'\n'` prints a new line character.

Input Arguments

A — Input array
numeric array

Input array, specified as a numeric array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

`precision` — Maximum number of significant digits
positive integer

Maximum number of significant digits in the output string, specified as a positive integer.

Note If you specify `precision` to exceed the precision of the input floating-point data type, the results might not match the input values to the precision you specified. The result depends on your computer hardware and operating system.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

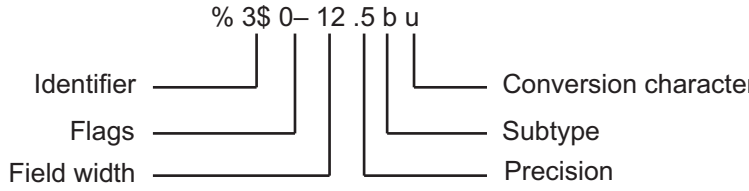
`formatSpec` — Format of output fields
formatting operators

Format of the output fields, specified using formatting operators. `formatSpec` also can include ordinary text and special characters.

`formatSpec` can be a character vector in single quotes, or, starting in R2016b, a string scalar.

Formatting Operator

A formatting operator starts with a percent sign, `%`, and ends with a conversion character. The conversion character is required. Optionally, you can specify identifier, flags, field width, precision, and subtype operators between `%` and the conversion character. (Spaces are invalid between operators and are shown here only for readability).



Conversion Character

This table shows conversion characters to format numeric and character data as text.

Value Type	Conversion	Details
Integer, signed	`%d` or `%i`	Base 10
Integer, unsigned	`%u`	Base 10
	`%o`	Base 8 (octal)
	`%x`	Base 16 (hexadecimal), lowercase letters `a–f`
	`%X`	Same as `%x`, uppercase letters `A–F`
Floating-point number	`%f`	Fixed-point notation (Use a precision operator to specify the number of digits after the decimal point.)
	`%e`	Exponential notation, such as `3.141593e+00` (Use a precision operator to specify the number of digits after the decimal point.)
	`%E`	Same as `%e`, but uppercase, such as `3.141593E+00` (Use a precision operator to specify the number of digits after the decimal point.)
	`%g`	The more compact of `%e` or `%f`, with no trailing zeros (Use a precision operator to specify the number of significant digits.)
	`%G`	The more compact of `%E` or `%f`, with no trailing zeros (Use a precision operator to specify the number of significant digits.)
Characters or strings	`%c`	Single character
	`%s`	Character vector or string array. The type of the output text is the same as the type of `formatSpec`.

Optional Operators

The optional identifier, flags, field width, precision, and subtype operators further define the format of the output text.

- **Identifier**

  Order for processing the function input arguments. Use the syntax *n*`$`, where *n* represents the positions of the other input arguments in the function call.

  Example: `('%3$s %2$s %1$s %2$s','A','B','C')` prints input arguments `'A'`, `'B'`, `'C'` as follows: `C B A B`.

  Note: If an input argument is an array, you cannot use identifiers to specify particular array elements from that input argument.

- **Flags**

`'–'`	Left-justify. **Example:** `%-5.2f` **Example:** `%-10s`
`'+'`	Always print a sign character (+ or –) for any numeric value. **Example:** `%+5.2f` Right-justify text. **Example:** `%+10s`
`' '`	Insert a space before the value. **Example:** `% 5.2f`
`'0'`	Pad to field width with zeros before the value. **Example:** `%05.2f`
`'#'`	Modify selected numeric conversions: • For `%o`, `%x`, or `%X`, print `0`, `0x`, or `0X` prefix. • For `%f`, `%e`, or `%E`, print decimal point even when precision is 0. • For `%g` or `%G`, do not remove trailing zeros or decimal point. **Example:** `%#5.0f`

- **Field Width**

  Minimum number of characters to print. The field width operator can be a number, or an asterisk (*) to refer to an input argument.

When you specify * as the field width operator, the other input arguments must provide both a width and a value to be printed. Widths and values can be pairs of arguments or pairs within a numeric array. With * as the field width operator, you can print different values with different widths.

Example: The input arguments `('%12d',intmax)` are equivalent to `('%*d',12,intmax)`.

Example: The input arguments `('%*d',[2 10 5 100])` return `'10 100'`, with two spaces allocated for `10` and five spaces for `100`. As an alternative, you also can specify the field widths and values as multiple arguments, as in `('%*d',2,10,5,100)`.

The function pads to field width with spaces before the value unless otherwise specified by flags.

- **Precision**

For `%f`, `%e`, or `%E`	Number of digits to the right of the decimal point **Example:** `'%.4f'` prints `pi` as `'3.1416'`
For `%g` or `%G`	Number of significant digits **Example:** `'%.4g'` prints `pi` as `'3.142'`

The precision operator can be a number, or an asterisk (*) to refer to an argument.

When you specify * as the field precision operator, the other input arguments must provide both a precision and a value to be printed. Precisions and values can be pairs of arguments, or pairs within a numeric array. With * as the precision operator, you can print different values to different precisions.

When you specify `*.*` as field width and precision operators, you must specify field widths, precisions, and values as triplets.

Example: The input arguments `('%.4f',pi)` are equivalent to `('%.*f',4,pi)`.

Example: The input arguments `('%6.4f',pi)` are equivalent to `('%.*f',6,4,pi)`.

Example: The input arguments `('%*.*f',6,4,pi,9,6,exp(1))` return `'3.1416 2.718282'`, with `9` and `6` as the field width and precision for the output of `exp(1)`.

Note If you specify a precision operator for floating-point values that exceeds the precision of the input numeric data type, the results might not match the input values

to the precision you specified. The result depends on your computer hardware and operating system.

- **Subtypes**

  You can use a subtype operator to print a floating-point value as its octal, decimal, or hexadecimal value. The subtype operator immediately precedes the conversion character. This table shows the conversions that can use subtypes.

Input Value Type	Subtype and Conversion Character	Output Value Type

Floating-point number	`%bx` or `%bX`
	`%bo`
	`%bu`
		Double-precision hexadecimal, octal, or decimal value
		Example: `%bx` prints `pi` as `400921fb54442d18`

	`%tx` or `%tX`
	`%to`
	`%tu`
		Single-precision hexadecimal, octal, or decimal value
		Example: `%tx` prints `pi` as `40490fdb`

Text Before or After Formatting Operators

`formatSpec` can also include additional text before a percent sign, `%`, or after a conversion character. The text can be:

- Ordinary text to print.
- Special characters that you cannot enter as ordinary text. This table shows how to represent special characters in `formatSpec`.

Special Character	Representation
Single quotation mark	`''`
Percent character	`%%`
Backslash	`\\`
Alarm	`\a`
Backspace	`\b`

Special Character	Representation
Form feed	`\f`
New line	`\n`
Carriage return	`\r`
Horizontal tab	`\t`
Vertical tab	`\v`
Character whose Unicode numeric value can be represented by the hexadecimal number, `N`	`\xN` **Example:** `sprintf('\x5A')` returns `'Z'`
Character whose Unicode numeric value can be represented by the octal number, `N`	`\N` **Example:** `sprintf('\132')` returns `'Z'`

Notable Behavior of Conversions with Formatting Operators

- Numeric conversions print only the real component of complex numbers.
- If you specify a conversion that does not fit the data, such as a text conversion for a numeric value, MATLAB overrides the specified conversion, and uses `%e`.

  Example: `'%s'` converts `pi` to `3.141593e+00`.
- If you apply a text conversion (either `%c` or `%s`) to integer values, MATLAB converts values that correspond to valid character codes to characters.

  Example: `'%s'` converts `[65 66 67]` to `ABC`.

Data Types: `char` | `string`

Output Arguments

s — Text representation of input array
character array

Text representation of the input array, returned as a character array.

Tips

`num2str` does not accept positional identifiers in the `formatSpec` input argument. For example, `num2str([14 15],'%2$X %1$o)` returns an error.

Positional identifiers specify the order in which the formatting operator processes input arguments of the function, not the elements of an input array. When you call `num2str`, there is only one input argument that has numbers to convert.

Algorithms

`num2str` trims any leading spaces from a character array, even when `formatSpec` includes a space character flag. For example, `num2str(42.67,'% 10.2f')` returns a 1-by-5 character array `'42.67'`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The input arguments must be constants.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`cast` | `int2str` | `mat2str` | `sprintf` | `str2num`

Introduced before R2006a

numberOfStrips

Total number of strips in image

Syntax

```
numStrips = numberOfStrips(t)
```

Description

`numStrips = numberOfStrips(t)` returns the total number of strips in the image specified in the `Tiff` object `t`.

Examples

Determine Number of Strips in Image

Determine the number of strips in the second image of a TIFF file.

Create a `Tiff` object for the file `example.tif`.

```
t = Tiff('example.tif','r');
```

When you create a `Tiff` object, the current image file directory points to the first image in the TIFF file. To get data on the second image, change the image file directory to point to the second image.

```
nextDirectory(t)
```

Get the number of strips in the image.

```
if ~isTiled(t) % if the image does not have a tiled layout
    numStrips = numberOfStrips(t)
end
```

```
numStrips = 7
```

Close the `Tiff` object.

```
close(t);
```

Input Arguments

`t` — Tiff object
`Tiff` object

`Tiff` object representing a TIFF file. Use the `Tiff` function to create the object.

Algorithms

References

This function corresponds to the `TIFFNumberOfStrips` function in the LibTIFF C API. To use this function, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`isTiled` | `numberOfTiles`

Topics

"Importing Images"
"Exporting to Images"

Introduced in R2009b

numberOfTiles

Total number of tiles in image

Syntax

```
numTiles = numberOfTiles(t)
```

Description

`numTiles = numberOfTiles(t)` returns the total number of tiles in the image specified in the `Tiff` object `t`.

Examples

Determine Number of Tiles in Image

Create a `Tiff` object for the file `example.tif`.

```
t = Tiff('example.tif','r');
```

Check if the image has a tiled organization, and then get the number of tiles in the image.

```
if isTiled(t) % If the image has tiled layout
    nTiles = numberOfTiles(t)
end
```

```
nTiles = 110
```

Close the `Tiff` object.

```
close(t);
```

Input Arguments

t — Tiff object
`Tiff` object

`Tiff` object representing a TIFF file. Use the `Tiff` function to create the object.

Algorithms

References

This function corresponds to the `TIFFNumberOfTiles` function in the LibTIFF C API. To use this function, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`isTiled` | `numberOfStrips`

Topics

"Importing Images"
"Exporting to Images"

Introduced in R2009b

numel

Number of array elements

Syntax

```
n = numel(A)
```

Description

`n = numel(A)` returns the number of elements, `n`, in array `A`, equivalent to `prod(size(A))`.

Examples

Number of Elements in 3-D Matrix

Create a 4-by-4-by-2 matrix.

```
A = magic(4);
A(:,:,2) = A'
```

```
A =
A(:,:,1) =

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1


A(:,:,2) =

    16     5     9     4
     2    11     7    14
     3    10     6    15
```

```
    13     8    12     1
```

`numel` counts 32 elements in the matrix.

```
n = numel(A)
```

```
n = 32
```

String Array

Create a string array and compute the number of elements in the array.

```
A = ["a" "b" "c"; "d" "e" "f"]
```

```
A = 2x3 string array
    "a"    "b"    "c"
    "d"    "e"    "f"
```

```
n = numel(A)
```

```
n = 6
```

Number of Elements in Cell Array

Create a cell array of character vectors.

```
A = {'dog','cat','fish','horse'};
```

`numel` counts 4 elements in the array.

```
n = numel(A)
```

```
n = 4
```

Number of Elements in Table

Create a table with four variables listing patient information for five people.

```
LastName = {'Smith';'Johnson';'Williams';'Jones';'Brown'};
Age = [38;43;38;40;49];
Height = [71;69;64;67;64];
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];

A = table(Age,Height,Weight,BloodPressure,'RowNames',LastName)
```

```
A=5×4 table
                Age    Height    Weight    BloodPressure
                ___    ______    ______    _____________

    Smith       38       71       176       124     93
    Johnson     43       69       163       109     77
    Williams    38       64       131       125     83
    Jones       40       67       133       117     75
    Brown       49       64       119       122     80
```

Find the number of elements in the table.

```
n = numel(A)
```

```
n = 20
```

`numel` returns a value equivalent to `prod(size(A))` corresponding to the 5 rows and 4 variables.

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array | table | timetable

Input array, specified as a scalar, vector, matrix, multidimensional array, table, or timetable.

Tips

- If `A` is a table, `numel` returns the number of elements in the table, `A`, equivalent to `prod(size(A))`. Variables in a table can have multiple columns, but `numel(A)` only accounts for the number of rows and number of variables.
- If `A` is a character vector of type `char`, then `numel` returns the number of characters. However, if `A` is a string scalar, `numel` returns `1` because it is a single element of a string array. For example, compare the output of `numel` for a character vector and string:

```
nchar = numel('mytext')

nchar =

     6

nstr = numel("mytext")

nstr =

     1
```

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`prod` | `size` | `subsref`

Topics

“Modify nargout and nargin for Indexing Methods”

Introduced before R2006a

NumericRuler Properties

Control axis with numeric values

Description

`NumericRuler` properties control the appearance and behavior of an *x*-axis, *y*-axis, or *z*-axis that shows numeric values. Each individual axis has its own ruler object. By changing property values of the ruler, you can modify certain aspects of a specific axis.

Use dot notation to refer to a particular ruler and property. Access the ruler objects through the `XAxis`, `YAxis`, and `ZAxis` properties of the `Axes` object.

```
ax = gca;
co = ax.XAxis.Color;
ax.XAxis.Color = 'blue';
```

Properties

Appearance

`Color` — Color of axis line and labels

RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of the axis line and labels, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Note The `Color` property for the ruler and the associated `XColor`, `YColor`, or `ZColor` property for the parent axes always have the same value. Setting one also sets the other.

`LineWidth` — Width of axis line and tick marks

`0.5` (default) | positive value

Width of axis line and tick marks, specified as a positive value in point units. One point equals 1/72 inch.

Example: `ax.XAxis.LineWidth = 2;`

Note Setting the `LineWidth` property for the parent axes sets the `LineWidth` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

`Label` — Axis label

text object (default)

Axis label, which is a text object. To display text or change existing text, set the `String` property for the text object. Use other properties to change the text appearance, such as the font style or color.

```
ax = gca;
ax.XAxis.Label.String = 'X Axis';
ax.XAxis.Label.FontSize = 12;
```

For a full list of options, see Text.

Alternatively, add or modify the axis labels using the `xlabel`, `ylabel`, and `zlabel` functions.

Note The text object is not a child of the ruler object, so it cannot be returned by `findobj` and it does not use the default text property values.

`Visible` — Axis visibility

`'on'` (default) | `'off'`

Axis visibility, specified as one of these values:

- `'on'` — Display the axis.
- `'off'` — Hide the axis without deleting it. You still can access properties of an invisible axis using the ruler object.

Example: `ax.XAxis.Visible = 'off';`

Scale and Direction

`Limits` — Minimum and maximum axis limits
`[0 1]` (default) | two-element vector of the form `[min max]`

Minimum and maximum axis limits, specified as a two-element vector of the form `[min max]`.

If you assign a value to this property, then MATLAB sets the associated mode to `'manual'`.

Alternatively, set the limits using the `xlim`, `ylim`, and `zlim` functions.

Note The `Limits` property for the ruler and the associated `XLim`, `YLim`, or `ZLim` property for the parent axes always have the same value. Setting one also sets the other.

Example: `ax.XAxis.Limits = [-10 10];`

`LimitsMode` — Selection mode for `Limits` property
`'auto'` (default) | `'manual'`

Selection mode for the `Limits` property, specified as one of these values:

- `'auto'` — Automatically select the axis limits based on the data plotted.
- `'manual'` — Use axis limit values that you specify. To specify the axis limits, set the `Limits` property.

Note The `LimitsMode` property for the ruler and the associated `XLimMode`, `YLimMode`, or `ZLimMode` property for the parent axes always have the same value. Setting one also sets the other.

`Scale` — Scale of values along axis
`'linear'` (default) | `'log'`

Scale of values along axis, specified as `'linear'` or `'log'`.

Note The `Scale` property for the ruler and the associated `XScale`, `YScale`, or `ZScale` property for the parent axes always have the same value. Setting one also sets the other.

Example: `ax.XAxis.Scale = 'log';`

`Direction` — Direction of increasing values

`'normal'` (default) | `'reverse'`

Direction of increasing values, specified as one of these values:

- `'normal'` — Values increase from left to right or bottom to top.
- `'reverse'` — Values increase from right to left or top to bottom.

Note The `Direction` property for the ruler and the associated `XDir`, `YDir`, or `ZDir` property for the parent axes always have the same value. Setting one also sets the other.

Example: `ax.XAxis.Direction = 'reverse';`

Tick Values and Labels

`TickValues` — Tick mark locations along the axis

`[]` (default) | vector of increasing values

Tick mark locations along the axis, specified as a vector of increasing values.

Example: `ax.XAxis.TickValues = [2 4 6 8 10];`

Example: `ax.XAxis.TickValues = 0:10:100;`

If you assign a value to this property, then MATLAB sets the `TickValuesMode` property to `'manual'`.

Alternatively, use the `xticks`, `yticks`, and `zticks` functions.

Note The `TickValues` property for the ruler and the associated `XTick`, `YTick`, or `ZTick` property for the parent axes always have the same value. Setting one also sets the other.

`TickValuesMode` — Selection mode for `TickValues` property

`'auto'` (default) | `'manual'`

Selection mode for the `TickValues` property, specified as one of these values:

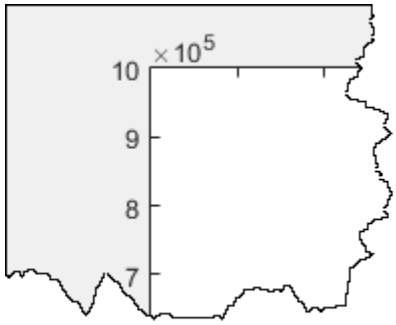
- `'auto'` — Automatically select the tick values based on the data plotted.
- `'manual'` — Use tick values that you specify. To specify the values, set the `TickValues` property.

Note The `TickValuesMode` property for the ruler and the associated `XTickMode`, `YTickMode`, or `ZTickMode` property for the parent axes always have the same value. Setting one also sets the other.

`Exponent` — Exponential notation common to all tick values

integer value

Exponential notation common to all tick values, specified as an integer value. The axis displays an exponent label showing the base and exponent value, for example:



The base value is always 10. However, you can change the exponent value by setting the `Exponent` property. For example, change the exponent to 2.

```
ax = gca;
ax.YAxis.Exponent = 2;
```

If the exponent value is `0`, then the exponent label does not display.

If you assign a value to this property, then MATLAB sets the `ExponentMode` property to `'manual'`. If the axis has a log scale, then the `Exponent` property has no effect.

`ExponentMode` — Selection mode for `Exponent` property

`'auto'` (default) | `'manual'`

Selection mode for the `Exponent` property, specified as one of these values:

- `'auto'` — Automatically select the exponent value based on the axis limits.
- `'manual'` — Use an exponent value that you specify. To specify the value, set the `Exponent` property.

`TickLabels` — Tick mark labels

`''` (default) | cell array of character vectors | string array | categorical array

Tick mark labels, specified as a cell array of character vectors, string array, or categorical array. If you do not specify enough labels for all of the tick values, then the labels repeat. The labels support TeX and LaTeX markup. See the `TickLabelInterpreter` property for more information.

Example: `ax.XAxis.TickLabels = {'January','February','March','April','May'}';`

Example: `ax.YAxis.TickLabels = {'\pi','2\pi','3\pi'}'`

If you set this property, then MATLAB sets the `TickLabelsMode` property to `'manual'`.

Alternatively, specify the tick labels using the `xticklabels`, `yticklabels`, and `zticklabels` functions.

Note

- If you specify this property as a categorical array, MATLAB uses the values in the array, not the categories.
- The `TickLabels` property for the ruler and the associated `XTickLabel`, `YTickLabel`, or `ZTickLabel` property for the parent axes always have the same value. Setting one also sets the other.

Data Types: `char` | `string` | `categorical`

`TickLabelsMode` — Selection mode for `TickLabels` property

`'auto'` (default) | `'manual'`

Selection mode for the `TickLabels` property, specified as one of these values:

- `'auto'` — Automatically select the tick labels.
- `'manual'` — Use tick labels that you specify. To specify the labels, set the `TickLabels` property.

Note The `TickLabelsMode` property for the ruler and the associated `XTickLabelMode`, `YTickLabelMode`, or `ZTickLabelMode` property for the parent axes always have the same value. Setting one also sets the other.

`TickLabelInterpreter` — Interpretation of tick label characters

`'tex'` (default) | `'latex'` | `'none'`

Interpretation of tick label characters, specified as one of these values:

- `'tex'` — Interpret characters using a subset of TeX markup.
- `'latex'` — Interpret characters using LaTeX markup.
- `'none'` — Display literal characters.

Note Setting the `TickLabelInterpreter` property for the parent axes sets the `TickLabelInterpreter` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.XAxis.TickLabelInterpreter = 'latex';`

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the text type and color, and include special characters in the text.

This table lists the supported modifiers with the `TickLabelInterpreter` property set to `'tex'`. Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they only modify the next character or the text within the curly braces `{}`.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`

Modifier	Description	Example
`\sl`	Oblique font (rarely available)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{specifier}`	Set `specifier` as the name of a font family to change the font style. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{specifier}`	Set `specifier` as a scalar numeric value to change the font size.	`'\fontsize{15} text'`
`\color{specifier}`	Set `specifer` as one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Set `specifier` as a three-element RGB triplet to change the font color.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	β	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\epsilon	ϵ	\Delta	Δ	\spadesuit	♠
\zeta	ζ	\Theta	Θ	\leftrightarrow	↔
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	μ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `TickLabelInterpreter` property to `'latex'`. Use dollar symbols around the labels, for example, use `'$\int_1^{20} x^2 dx$'` for inline mode or `'$$\int_1^{20} x^2 dx$$'` for display mode.

The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup within the text. The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters. For multiline text, the maximum size of the text reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project website at https://www.latex-project.org/.

`TickLabelFormat` — Tick label format and decimal precision

character vector | string

Tick label format and decimal precision, specified as a character vector or string. For example, you can display the tick labels in a currency format, control the number of decimals that appear in each label, or add text after all the labels.

Example: `ax.XAxis.TickLabelFormat = '%g%%';` displays a percent sign after all the tick labels.

This table lists some common formats.

Common Format	Character Vector to Use	Examples of Resulting Format

Currency — Display dollar sign before values, use two decimal places, and use commas.	`'$%,.2f'`	`$0.01`
		`$1.00`
		`$1,000.00`

Common Format	Character Vector to Use	Examples of Resulting Format

Temperatures — Display degree symbol after values.	`'%g\\circ'`	`0.01°`
		`1°`
		`1000°`

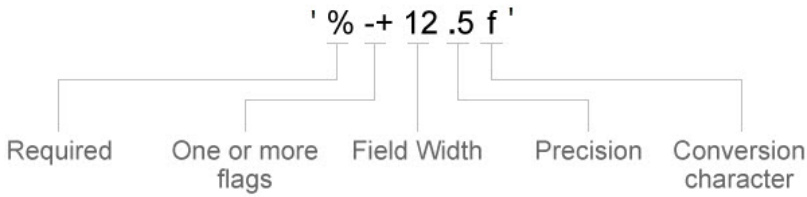
Percentages — Display percent sign after values.	`'%g%%'`	`0.01%`
		`1%`
		`1000%`

Commas — Display commas in the thousandth place.	`'%,g'`	`0.01`
		`1`
		`1,000`

Alternatively, specify the tick label format using the `xtickformat`, `ytickformat`, and `ztickformat` functions.

Custom Format

If none of the formats mentioned in the table gives your desired format, then create a custom character vector or string with identifiers.



Identifiers are optional, except the percent sign and conversion character. Construct the format in this order:

- One or more flags — Options such as adding a plus sign before positive values. For a full list of options, see the table of Optional Flags.
- Field width — Minimum number of characters to print in the tick label. Specify the field width as an integer value. If the number of significant digits in the tick value is smaller than the field width, then the label is padded with spaces.
- Precision — Number of digits to the right of the decimal point or the number of significant digits, depending on the conversion character. Specify the precision as an integer value.

- Conversion character — Value type. For a full list of options, see the table of Conversion Characters. If you specify a conversion that does not fit the data, MATLAB overrides the specified conversion and uses `%e`.

Also, you can specify literal text at the beginning or end of the format. To print a single quotation mark, use `''`. To print a percent character, use `%%`.

- `'$%.2f'` — Display a dollar sign before each value and use fixed-point notation with two decimal values.
- `'%.3f Million'` — Display `Million` after each value and use fixed-point notation with three decimal values.

Optional Flags

Identifier	Description	Example of Numeric Format
`,`	Display commas every three digits, such as `'1,000'`.	`'%,4.4g'`
`+`	Print the sign character (`+`) for positive values, such as `'+100'`.	`'%+4.4g'`
`0`	Pad the field width with leading zeros instead of spaces, such as `'0100'`.	`'%04.4g'`
`-`	Left-justify, which pads the end of the value with spaces instead of the beginning. For example, if the field width is 4, then this flag formats the label as `'100 '` instead of `' 100'`.	`'%-4.4g'`
`#`	For the `%f`, `%e`, and `%g` conversion characters, print the decimal point even when the precision is 0, such as `'100.'`. For `%g`, do not remove trailing zeros.	`'%#4.4g'`

Conversion Characters

Identifier	Description	Example
`d` or `i`	Signed integer with base 10. The precision value indicates the number of significant digits.	`'%.4d'` displays π as `0003`.
`f`	Fixed-point notation. The precision value indicates the number of decimal places.	`'%.4f'` displays π as `3.1416`.
`e`	Exponential notation. The precision value indicates the number of decimal places.	`'%.4e'` displays π as 3.1416×10^0.
`g`	The more compact version of `e` or `f`, with no trailing zeros. The precision value indicates the maximum number of decimal places.	`'%.4g'` displays π as `3.1416`.

`TickLabelRotation` — Rotation of tick labels
scalar value in degrees

Rotation of tick labels, specified as a scalar value in degrees. Positive values give counterclockwise rotation. Negative values give clockwise rotation.

Alternatively, you can rotate the tick labels using the `xtickangle`, `ytickangle`, and `ztickangle` functions.

Note The `TickLabelRotation` property for the ruler and the associated `XTickLabelRotation`, `YTickLabelRotation`, or `ZTickLabelRotation` property for the parent axes always have the same value. Setting one also sets the other.

Example: `ax.XAxis.TickLabelRotation = 45;`

Example: `ax.YAxis.TickLabelRotation = -45;`

`TickLabelRotationMode` — Selection mode for `TickLabelRotation` property
`'auto'` (default) | `'manual'`

Selection mode for the `TickLabelRotation` property, specified as one of these values:

- `'auto'` — Automatically select the tick label rotation.
- `'manual'` — Use a tick label rotation that you specify. To specify the rotation, set the `TickLabelRotation` property.

`TickDirection` — Tick mark direction
`'in'` (default) | `'out'` | `'both'`

Tick mark direction, specified as one of these values:

- `'in'` — Direct the tick marks inward from the axis lines. This is the default for 2-D views.
- `'out'` — Direct the tick marks outward from the axis lines. This is the default for 3-D views.
- `'both'` — Center the tick marks over the axis lines.

If you assign a value to this property, then MATLAB sets the `TickDirectionMode` property to `'manual'`.

Note Setting the `TickDir` property for the parent axes sets the `TickDirection` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.YAxis.TickDirection = 'out';`

`TickDirectionMode` — Selection mode for `TickDirection` property
`'auto'` (default) | `'manual'`

Selection mode for the `TickDirection` property, specified as one of these values:

- `'auto'` — Automatically select the tick direction.
- `'manual'` — Use a tick direction that you specify. To specify the tick direction, set the `TickDirection` property.

`TickLength` — Tick mark length
two-element vector

Tick mark length, specified as a two-element vector of the form `[2Dlength 3Dlength]`. The first element is the tick mark length in 2-D views. The second element is the tick mark length in 3-D views. Specify the values in units normalized relative to the longest axes dimension.

Note Setting the `TickLength` property for the parent axes sets the `TickLength` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.YAxis.TickLength = [0.02 0.035];`

`MinorTick` — Minor tick mark display

`'off'` (default) | `'on'`

Minor tick mark display, specified as one of these values:

- `'off'` — Do not display minor tick marks. This is the default value for an axis with a linear scale.
- `'on'` — Display minor tick marks between the major tick marks on the axis. This is the default value for an axis with a log scale. The space between the major tick marks determines the number of minor tick marks. If the `MinorTickValues` property is set to empty `[]`, then no minor tick marks appear. Specify the tick mark locations by setting the `MinorTickValues` property.

Note The `MinorTick` property for the ruler and the associated `XMinorTick`, `YMinorTick`, or `ZMinorTick` property for the parent axes always have the same value. Setting one also sets the other.

Example: `ax.XAxis.MinorTick = 'on';`

`MinorTickValues` — Minor tick mark locations

`[]` (default) | vector of increasing values

Minor tick mark locations, specified as a vector of increasing values.

Example: `ax.XAxis.MinorTickValues = [0 0.5 1 1.5 2 2.5];`

Example: `ax.YAxis.MinorTickValues = 0:5:100;`

If you assign values to this property, then MATLAB sets the `MinorTickValuesMode` property to `'manual'`.

`MinorTickValuesMode` — Selection mode for `MinorTickValues` property

`'auto'` (default) | `'manual'`

Selection mode for the `MinorTickValues` property, specified as one of these values:

- `'auto'` — Use automatically calculated minor tick values.
- `'manual'` — Use minor tick values that you specify. To specify the values, set the `MinorTickValues` property.

Font

`FontName` — Font name

system supported font name | `'FixedWidth'`

Font name, specified as a system supported font name or `'FixedWidth'`. The default font depends on the specific system and locale. To use a fixed-width font that renders well, specify `'FixedWidth'`. The actual fixed-width font used depends on the `FixedWidthFontName` property of the root object.

Note Setting the `FontName` property for the parent axes sets the `FontName` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.XAxis.FontName = 'Cambria';`

`FontSize` — Font size

scalar numeric value

Font size, specified as a scalar numeric value. The default font size depends on the specific operating system and locale.

Note Setting the `FontSize` property for the parent axes sets the `FontSize` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.XAxis.FontSize = 12;`

`FontWeight` — Thickness of text characters
`'normal'` (default) | `'bold'`

Thickness of the text characters, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font.
- `'bold'` — Thicker character outlines than normal.

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold font weight. Therefore, specifying a bold font weight still can result in the normal font weight.

Note Setting the `FontWeight` property for the parent axes sets the `FontWeight` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.XAxis.FontWeight = 'bold';`

`FontAngle` — Text character slant
`'normal'` (default) | `'italic'`

Character slant, specified as `'normal'` or `'italic'`. Not all fonts have both font styles. Therefore, the italic font can look the same as the normal font.

Note Setting the `FontAngle` property for the parent axes sets the `FontAngle` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.XAxis.FontAngle = 'italic';`

`FontSmoothing` — Text antialiasing
`'on'` (default) | `'off'`

Text smoothing, specified as one of these values:

- `'on'` — Enable text antialiasing to reduce the jagged appearance of text characters and make the text easier to read. In certain cases, smoothed text blends against the background color and can make the text appear blurry.
- `'off'` — Disable text antialiasing. Use this setting if the text seems blurry.

Note Setting the `FontSmoothing` property for the parent axes sets the `FontSmoothing` property for the ruler to the same value. However, setting the ruler property does not set the axes property. To prevent the axes property value from overriding the ruler property value, set the axes value first, and then set the ruler value.

Example: `ax.XAxis.FontSmoothing = 'off';`

Parent/Child

`Parent` — Ruler parent

`Axes` object

Ruler parent, specified as an `Axes` object.

Note Ruler objects are not listed in the `Children` property of the parent `Axes` object.

`Children` — Ruler children

empty `GraphicsPlaceholder` array

The ruler has no children. You cannot set this property.

See Also

Axes

Introduced in R2015b

nzmax

Amount of storage allocated for nonzero matrix elements

Syntax

```
n = nzmax(S)
```

Description

`n = nzmax(S)` returns the amount of storage allocated for nonzero elements in sparse matrix `S`. For sparse matrices, `nzmax(S) >= 1`.

- If `S` is a full matrix, then `nzmax(S)` is equal to `prod(size(S))`.

Often, `nnz(S)` and `nzmax(S)` are the same. But if `S` is created by an operation which produces fill-in matrix elements, such as sparse matrix multiplication or sparse LU factorization, then more storage might be allocated than is actually required, and `nzmax(S)` reflects this. Alternatively, `sparse(i,j,s,m,n,nzmax)` or its simpler form, `spalloc(m,n,nzmax)`, can set `nzmax` in anticipation of later fill-in.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`find` | `isa` | `nnz` | `nonzeros` | `size` | `whos`

Introduced before R2006a

ode15i

Solve fully implicit differential equations — variable order method

Syntax

```
[t,y] = ode15i(odefun,tspan,y0,yp0)
[t,y] = ode15i(odefun,tspan,y0,yp0,options)
[t,y,te,ye,ie] = ode15i(odefun,tspan,y0,yp0,options)
sol = ode15i( ___ )
```

Description

`[t,y] = ode15i(odefun,tspan,y0,yp0)`, where `tspan = [t0 tf]`, integrates the system of differential equations $f(t, y, y') = 0$ from `t0` to `tf` with initial conditions `y0` and `yp0`. Each row in the solution array `y` corresponds to a value returned in column vector `t`.

`[t,y] = ode15i(odefun,tspan,y0,yp0,options)` also uses the integration settings defined by `options`, which is an argument created using the `odeset` function. For example, use the `AbsTol` and `RelTol` options to specify absolute and relative error tolerances, or the `Jacobian` option to provide the Jacobian matrix.

`[t,y,te,ye,ie] = ode15i(odefun,tspan,y0,yp0,options)` additionally finds where functions of `(t,y,y')`, called event functions, are zero. In the output, `te` is the time of the event, `ye` is the solution at the time of the event, and `ie` is the index of the triggered event.

For each event function, specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the `'Events'` property to a function, such as `myEventFcn` or `@myEventFcn`, and creating a corresponding function: `[value,isterminal,direction] = myEventFcn(t,y,yp)`. For more information, see “ODE Event Location”.

`sol = ode15i( ___ )` returns a structure that you can use with `deval` to evaluate the solution at any point on the interval `[t0 tf]`. You can use any of the input argument combinations in previous syntaxes.

Examples

Solve Weissinger Implicit ODE

Calculate consistent initial conditions and solve an implicit ODE with `ode15i`.

Weissinger's equation is

$$ty^2(y')^3 - y^3(y')^2 + t\left(t^2 + 1\right)y' - t^2 y = 0.$$

Since the equation is in the generic form $f(t, y, y') = 0$, you can use the `ode15i` function to solve the implicit differential equation.

Code Equation

To code the equation in a form suitable for `ode15i`, you need to write a function with inputs for t, y, and y' that returns the residual value of the equation. The function `@weissinger` encodes this equation. View the function file.

```
type weissinger
```

```
function res = weissinger(t,y,yp)
%WEISSINGER  Evaluate the residual of the Weissinger implicit ODE
%
%   See also ODE15I.

%   Jacek Kierzenka and Lawrence F. Shampine
%   Copyright 1984-2014 The MathWorks, Inc.

res = t*y^2 * yp^3 - y^3 * yp^2 + t*(t^2 + 1)*yp - t^2 * y;
```

Calculate Consistent Initial Conditions

The `ode15i` solver requires *consistent* initial conditions, that is, the initial conditions supplied to the solver must satisfy

$$f(t_0, y, y') = 0.$$

Since it is possible to supply inconsistent initial conditions, and `ode15i` does not check for consistency, it is recommended that you use the helper function `decic` to compute such conditions. `decic` holds some specified variables fixed and computes consistent initial values for the unfixed variables.

In this case, fix the initial value $y(t_0) = \sqrt{\frac{3}{2}}$ and let `decic` compute a consistent initial value for the derivative $y'(t_0)$, starting from an initial guess of $y'(t_0) = 0$.

```
t0 = 1;
y0 = sqrt(3/2);
yp0 = 0;
[y0,yp0] = decic(@weissinger,t0,y0,1,yp0,0)
```

```
y0 = 1.2247

yp0 = 0.8165
```

Solve Equation

Use the consistent initial conditions returned by `decic` with `ode15i` to solve the ODE over the time interval [1 10].

```
[t,y] = ode15i(@weissinger,[1 10],y0,yp0);
```

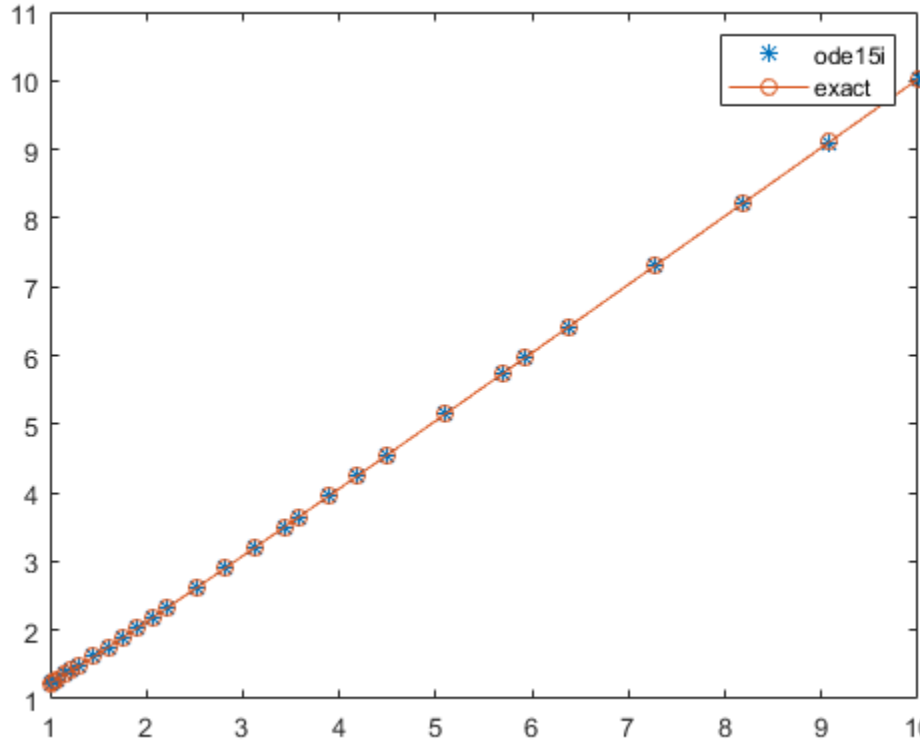
Plot Results

The exact solution of this ODE is

$$y(t) = \sqrt{t^2 + \frac{1}{2}}.$$

Plot the numerical solution `y` computed by `ode15i` against the analytical solution `ytrue`.

```
ytrue = sqrt(t.^2 + 0.5);
plot(t,y,'*',t,ytrue,'-o')
legend('ode15i', 'exact')
```



Solve Robertson Problem as Implicit Differential Algebraic Equations (DAEs)

This example reformulates a system of ODEs as a fully implicit system of differential algebraic equations (DAEs). The Robertson problem coded by hb1ode.m is a classic test problem for programs that solve stiff ODEs. The system of equations is

$$\begin{aligned} y_1' &= -0.04y_1 + 10^4 y_2 y_3 \\ y_2' &= 0.04y_1 - 10^4 4 y_2 y_3 - (3 \times 10^7) y_2^2 \\ y_3' &= (3 \times 10^7) y_2^2. \end{aligned}$$

`hb1ode` solves this system of ODEs to steady state with the initial conditions $y_1 = 1$, $y_2 = 0$, and $y_3 = 0$. But the equations also satisfy a linear conservation law,

$$y_1' + y_2' + y_3' = 0.$$

In terms of the solution and initial conditions, the conservation law is

$$y_1 + y_2 + y_3 = 1.$$

The problem can be rewritten as a system of DAEs by using the conservation law to determine the state of y_3. This reformulates the problem as the implicit DAE system

$$\begin{aligned} 0 &= y_1' + 0.04y_1 - 10^4 y_2 y_3 \\ 0 &= y_2' - 0.04y_1 + 10^4 y_2 y_3 + (3 \times 10^7) y_2^2 \\ 0 &= y_1 + y_2 + y_3 - 1. \end{aligned}$$

The function `robertsidae` encodes this DAE system.

```
function res = robertsidae(t,y,yp)
res = [yp(1) + 0.04*y(1) - 1e4*y(2)*y(3);
   yp(2) - 0.04*y(1) + 1e4*y(2)*y(3) + 3e7*y(2)^2;
   y(1) + y(2) + y(3) - 1];
```

The full example code for this formulation of the Robertson problem is available in ihb1dae.m.

Set the error tolerances and the value of $\partial f / \partial y'$.

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-6 1e-10 1e-6], ...
   'Jacobian',{[],[1 0 0; 0 1 0; 0 0 0]});
```

Use `decic` to compute consistent initial conditions from guesses. Fix the first two components of `y0` to get the same consistent initial conditions as found by `ode15s` in hb1dae.m, which formulates this problem as a semi-explicit DAE system.

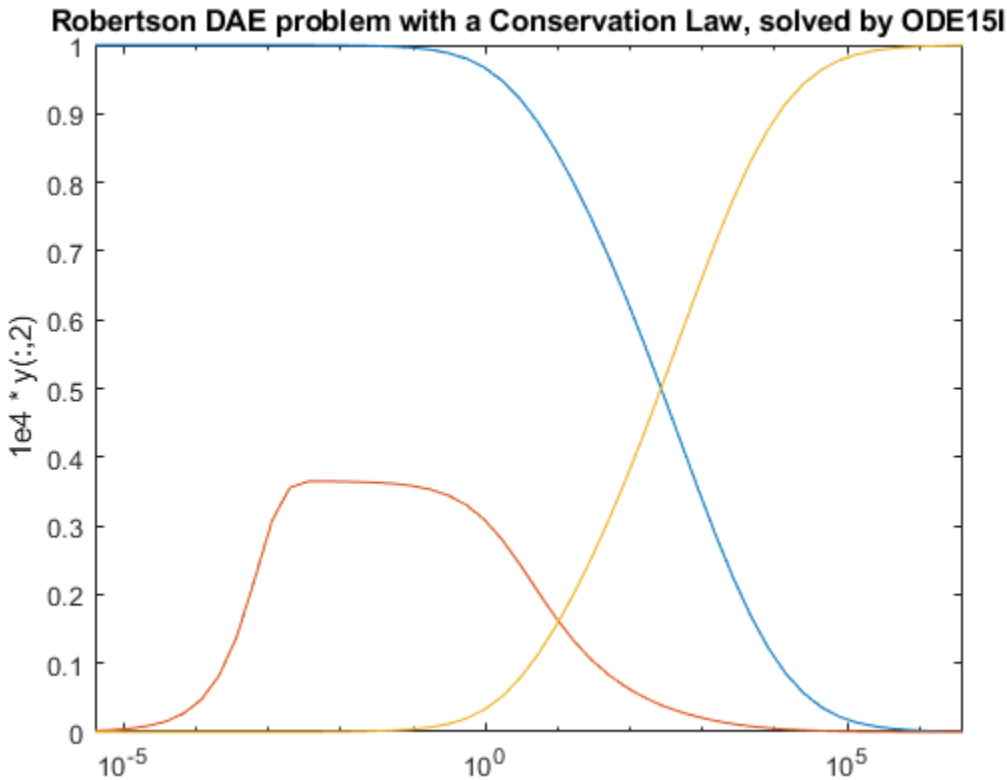
```
y0 = [1; 0; 1e-3];
yp0 = [0; 0; 0];
[y0,yp0] = decic(@robertsidae,0,y0,[1 1 0],yp0,[],options);
```

Solve the system of DAEs using `ode15i`.

```
tspan = [0 4*logspace(-6,6)];
[t,y] = ode15i(@robertsidae,tspan,y0,yp0,options);
```

Plot the solution components. Since the second solution component is small relative to the others, multiply it by `1e4` before plotting.

```
y(:,2) = 1e4*y(:,2);
semilogx(t,y)
ylabel('1e4 * y(:,2)')
title('Robertson DAE problem with a Conservation Law, solved by ODE15I')
```



Input Arguments

odefun — Functions to solve
function handle

Functions to solve, specified as a function handle that defines the functions to be integrated.

The function `f = odefun(t,y,yp)`, for a scalar `t` and column vectors `y` and `yp`, must return a column vector `f` of data type `single` or `double` that corresponds to $f(t, y, y')$. `odefun` must accept the three inputs for `t`, `y`, and `yp` even if one of the inputs is not used in the function.

For example, to solve $y' - y = 0$, use this function.

```
function f = odefun(t,y,yp)
f = yp - y;
```

For a system of equations, the output of `odefun` is a vector. Each equation becomes an element in the solution vector. For example, to solve

$$y'_1 - y_2 = 0$$
$$y'_2 + 1 = 0\,,$$

use this function.

```
function dy = odefun(t,y,yp)
dy = zeros(2,1);
dy(1) = yp(1)-y(2);
dy(2) = yp(2)+1;
```

For information on how to provide additional parameters to the function `odefun`, see "Parameterizing Functions".

Example: `@myFcn`

Data Types: `function_handle`

`tspan` — Interval of integration
vector

Interval of integration, specified as a vector. At minimum, `tspan` must be a two element vector `[t0 tf]` specifying the initial and final times. To obtain solutions at specific times between `t0` and `tf`, use a longer vector of the form `[t0,t1,t2,...,tf]`. The elements in `tspan` must be all increasing or all decreasing.

The solver imposes the initial conditions given by `y0` at the initial time `tspan(1)`, then integrates from `tspan(1)` to `tspan(end)`:

- If `tspan` has two elements, `[t0 tf]`, then the solver returns the solution evaluated at each internal integration step within the interval.
- If `tspan` has more than two elements `[t0,t1,t2,...,tf]`, then the solver returns the solution evaluated at the given points. However, the solver does not step precisely to each point specified in `tspan`. Instead, the solver uses its own internal steps to compute the solution, then evaluates the solution at the requested points in `tspan`.

The solutions produced at the specified points are of the same order of accuracy as the solutions computed at each internal step.

Specifying several intermediate points has little effect on the efficiency of computation, but for large systems it can affect memory management.

The values of `tspan` are used by the solver to calculate suitable values for `InitialStep` and `MaxStep`:

- If `tspan` contains several intermediate points `[t0,t1,t2,...,tf]`, then the specified points give an indication of the scale for the problem, which can affect the value of `InitialStep` used by the solver. Therefore, the solution obtained by the solver might be different depending on whether you specify `tspan` as a two-element vector or as a vector with intermediate points.
- The initial and final values in `tspan` are used to calculate the maximum step size `MaxStep`. Therefore, changing the initial or final values in `tspan` could lead to the solver using a different step sequence, which might change the solution.

Example: `[1 10]`

Example: `[1 3 5 7 9 10]`

Data Types: `single` | `double`

`y0` — Initial conditions for *y*
vector

Initial conditions for *y*, specified as a vector. `y0` must be the same length as the vector output of `odefun`, so that `y0` contains an initial condition for each equation defined in `odefun`.

The initial conditions for `y0` and `yp0` must be consistent, meaning that $f(t_0, y_0, y'_0) = 0$. Use the `decic` function to compute consistent initial conditions close to guessed values.

Data Types: `single` | `double`

`yp0` — Initial conditions for *y'*
vector

Initial conditions for *y'*, specified as a vector. `yp0` must be the same length as the vector output of `odefun`, so that `yp0` contains an initial condition for each variable defined in `odefun`.

The initial conditions for `y0` and `yp0` must be consistent, meaning that $f(t_0, y_0, y'_0) = 0$. Use the `decic` function to compute consistent initial conditions close to guessed values.

Data Types: `single` | `double`

`options` — Option structure
structure array

Option structure, specified as a structure array. Use the `odeset` function to create or modify the option structure.

See "Summary of ODE Options" for a list of which options are compatible with each ODE solver.

Example: `options = odeset('RelTol',1e-5,'Stats','on','OutputFcn',@odeplot)` specifies a relative error tolerance of `1e-5`, turns on the display of solver statistics, and specifies the output function `@odeplot` to plot the solution as it is computed.

Data Types: `struct`

Output Arguments

`t` — Evaluation points
column vector

Evaluation points, returned as a column vector.

- If `tspan` contains two elements, `[t0 tf]`, then `t` contains the internal evaluation points used to perform the integration.
- If `tspan` contains more than two elements, then `t` is the same as `tspan`.

`y` — Solutions
array

Solutions, returned as an array. Each row in `y` corresponds to the solution at the value returned in the corresponding row of `t`.

`te` — Time of events
column vector

Time of events, returned as a column vector. The event times in `te` correspond to the solutions returned in `ye`, and `ie` specifies which event occurred.

`ye` — Solution at time of events
array

Solution at time of events, returned as an array. The event times in `te` correspond to the solutions returned in `ye`, and `ie` specifies which event occurred.

`ie` — Index of triggered event function
column vector

Index of triggered event function, returned as a column vector. The event times in `te` correspond to the solutions returned in `ye`, and `ie` specifies which event occurred.

`sol` — Structure for evaluation
structure array

Structure for evaluation, returned as a structure array. Use this structure with the `deval` function to evaluate the solution at any point in the interval `[t0 tf]`. The `sol` structure array always includes these fields:

Structure Field	Description
`sol.x`	Row vector of the steps chosen by the solver.
`sol.y`	Solutions. Each column `sol.y(:,i)` contains the solution at time `sol.x(i)`.
`sol.solver`	Solver name.

Additionally, if you specify the `Events` option and events are detected, then `sol` also includes these fields:

Structure Field	Description
`sol.xe`	Points when events occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.

Structure Field	Description
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

Tips

- Providing the Jacobian matrix to `ode15i` is critical for reliability and efficiency. Alternatively, if the system is large and sparse, then providing the Jacobian sparsity pattern also assists the solver. In either case, use `odeset` to pass in the matrices using the `Jacobian` or `JPattern` options.

Algorithms

`ode15i` is a variable-step, variable-order (VSVO) solver based on the backward differentiation formulas (BDFs) of orders 1 to 5. `ode15i` is designed to be used with fully implicit differential equations and index-1 differential algebraic equations (DAEs). The helper function `decic` computes consistent initial conditions that are suitable to be used with `ode15i` [1].

References

[1] Lawrence F. Shampine, “Solving 0 = F(t, y(t), y′(t)) in MATLAB,” *Journal of Numerical Mathematics*, Vol.10, No.4, 2002, pp. 291-310.

See Also

`decic` | `deval` | `ode15s` | `ode23t` | `odeget` | `odeset`

Topics

“Choose an ODE Solver”
“Summary of ODE Options”
“Solve Differential Algebraic Equations (DAEs)”
“Troubleshoot Common ODE Problems”

Introduced before R2006a

ode15s

Solve stiff differential equations and DAEs — variable order method

Syntax

```
[t,y] = ode15s(odefun,tspan,y0)
[t,y] = ode15s(odefun,tspan,y0,options)
[t,y,te,ye,ie] = ode15s(odefun,tspan,y0,options)
sol = ode15s( ___ )
```

Description

`[t,y] = ode15s(odefun,tspan,y0)`, where `tspan = [t0 tf]`, integrates the system of differential equations $y' = f(t, y)$ from `t0` to `tf` with initial conditions `y0`. Each row in the solution array `y` corresponds to a value returned in column vector `t`.

All MATLAB ODE solvers can solve systems of equations of the form $y' = f(t, y)$, or problems that involve a mass matrix, $M(t, y)y' = f(t, y)$. The solvers all use similar syntaxes. The `ode23s` solver only can solve problems with a mass matrix if the mass matrix is constant. `ode15s` and `ode23t` can solve problems with a mass matrix that is singular, known as differential-algebraic equations (DAEs). Specify the mass matrix using the `Mass` option of `odeset`.

`[t,y] = ode15s(odefun,tspan,y0,options)` also uses the integration settings defined by `options`, which is an argument created using the `odeset` function. For example, use the `AbsTol` and `RelTol` options to specify absolute and relative error tolerances, or the `Mass` option to provide a mass matrix.

`[t,y,te,ye,ie] = ode15s(odefun,tspan,y0,options)` additionally finds where functions of (t,y), called event functions, are zero. In the output, `te` is the time of the event, `ye` is the solution at the time of the event, and `ie` is the index of the triggered event.

For each event function, specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the `'Events'` property to a function, such as `myEventFcn` or `@myEventFcn`, and creating a

corresponding function: [value,isterminal,direction] = myEventFcn(t,y). For more information, see “ODE Event Location”.

sol = ode15s(___) returns a structure that you can use with deval to evaluate the solution at any point on the interval [t0 tf]. You can use any of the input argument combinations in previous syntaxes.

Examples

ODE With Single Solution Component

Simple ODEs that have a single solution component can be specified as an anonymous function in the call to the solver. The anonymous function must accept two inputs (t,y) even if one of the inputs is not used.

Solve the ODE

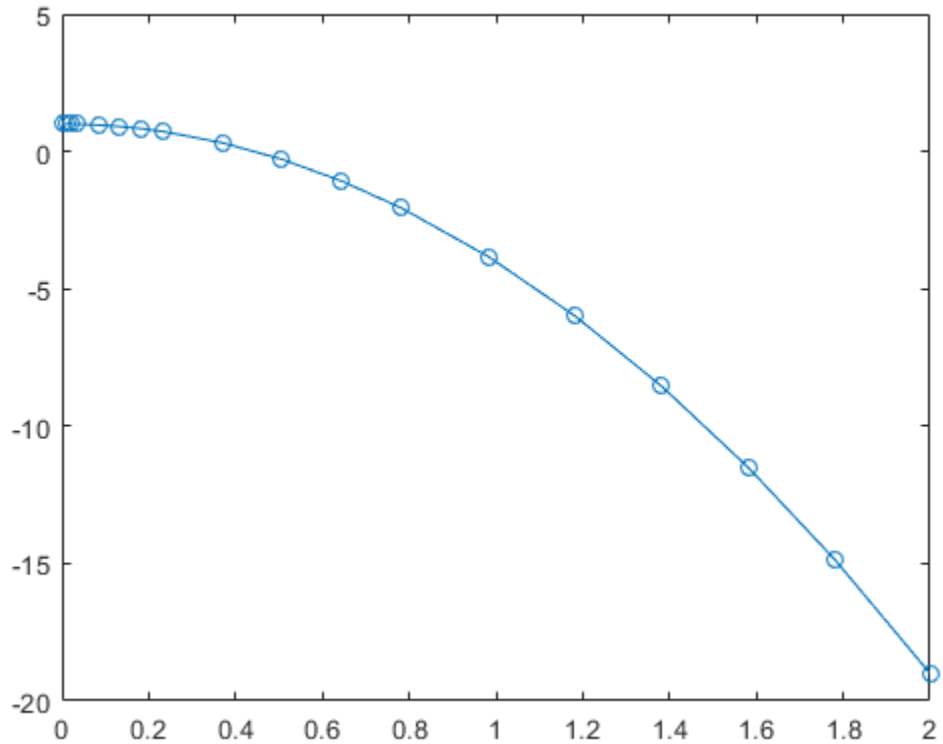
$$y' = -10t.$$

Use a time interval of [0,2] and the initial condition y0 = 1.

```
tspan = [0 2];
y0 = 1;
[t,y] = ode15s(@(t,y) -10*t, tspan, y0);
```

Plot the solution.

```
plot(t,y,'-o')
```



Solve Stiff ODE

An example of a stiff system of equations is the van der Pol equations in relaxation oscillation. The limit cycle has regions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

The system of equations is:

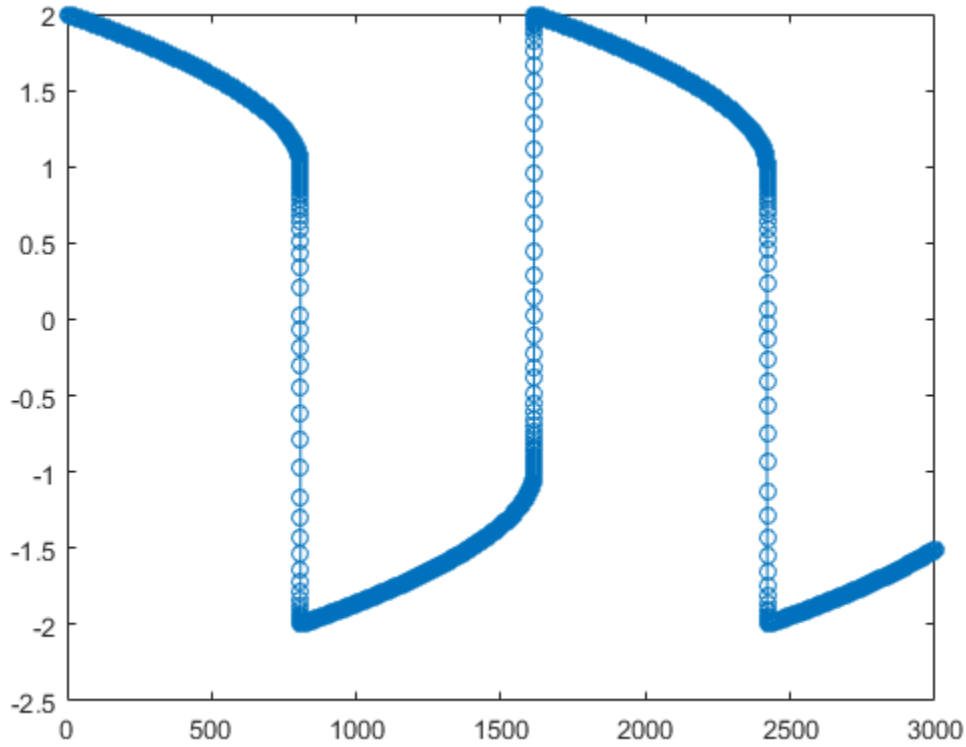
$$\begin{aligned} y_1' &= y_2 \\ y_2' &= 1000(1 - y_1^2)y_2 - y_1 \end{aligned}$$

The initial conditions are $y_1(0) = 2$ and $y_2(0) = 0$. The function `vdp1000` ships with MATLAB® and encodes the equations.

```
function dydt = vdp1000(t,y)
%VDP1000  Evaluate the van der Pol ODEs for mu = 1000.
%
%   See also ODE15S, ODE23S, ODE23T, ODE23TB.

%   Jacek Kierzenka and Lawrence F. Shampine
%   Copyright 1984-2014 The MathWorks, Inc.

dydt = [y(2); 1000*(1-y(1)^2)*y(2)-y(1)];
```

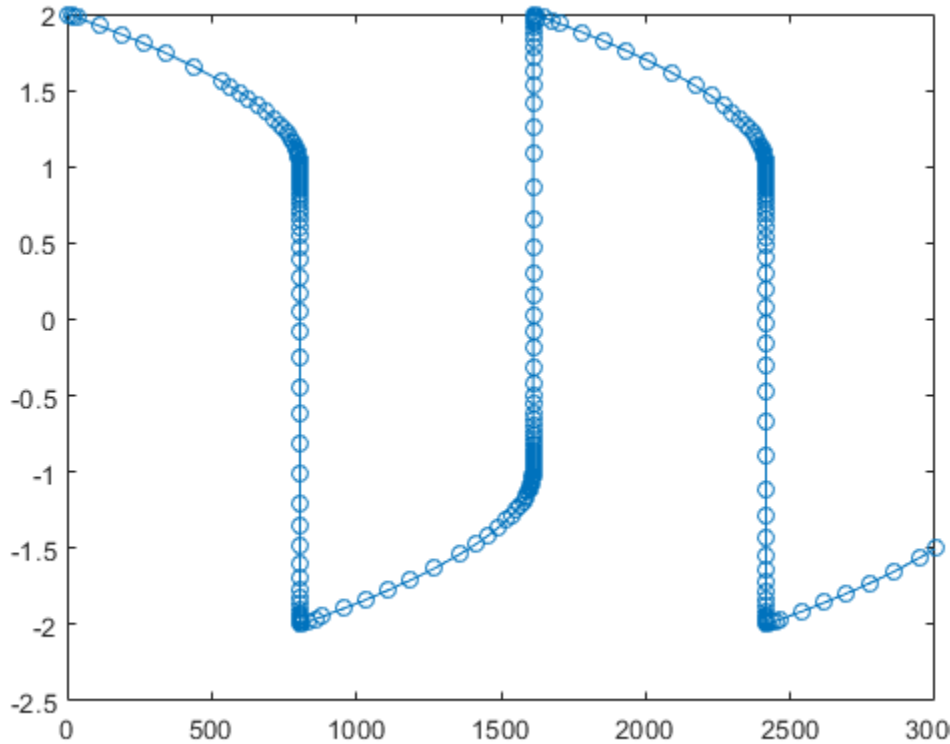
Solving this system using `ode45` with the default relative and absolute error tolerances (`1e-3` and `1e-6`, respectively) is extremely slow, requiring several minutes to solve and plot the solution. `ode45` requires millions of time steps to complete the integration, due to the areas of stiffness where it struggles to meet the tolerances.

This is a plot of the solution obtained by `ode45`, which takes a long time to compute. Notice the enormous number of time steps required to pass through areas of stiffness.



Solve the stiff system using the `ode15s` solver, and then plot the first column of the solution `y` against the time points `t`. The `ode15s` solver passes through stiff areas with far fewer steps than `ode45`.

```
[t,y] = ode15s(@vdp1000,[0 3000],[2 0]);
plot(t,y(:,1),'-o')
```



Pass Extra Parameters to ODE Function

`ode15s` only works with functions that use two input arguments, `t` and `y`. However, you can pass in extra parameters by defining them outside the function and passing them in when you specify the function handle.

Solve the ODE

$$y'' = \frac{A}{B} t y.$$

Rewriting the equation as a first-order system yields

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= \frac{A}{B} t y_1. \end{aligned}$$

`odefcn.m` represents this system of equations as a function that accepts four input arguments: `t`, `y`, `A`, and `B`.

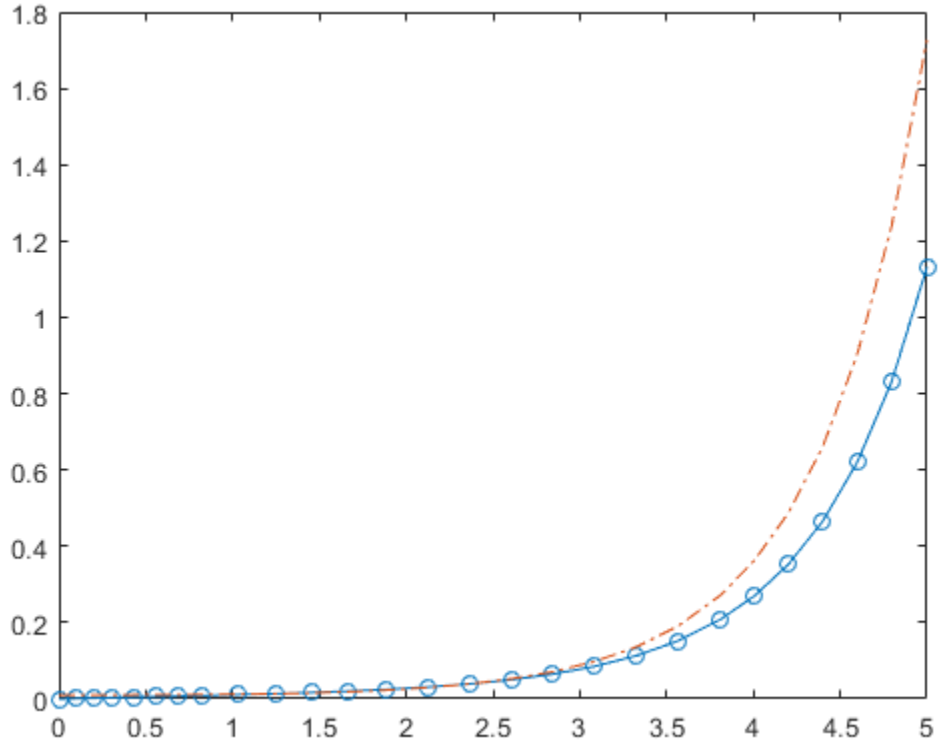
```
function dydt = odefcn(t,y,A,B)
dydt = zeros(2,1);
dydt(1) = y(2);
dydt(2) = (A/B)*t.*y(1);
```

Solve the ODE using `ode15s`. Specify the function handle such that it passes in the predefined values for `A` and `B` to `odefcn`.

```
A = 1;
B = 2;
tspan = [0 5];
y0 = [0 0.01];
[t,y] = ode15s(@(t,y) odefcn(t,y,A,B), tspan, y0);
```

Plot the results.

```
plot(t,y(:,1),'-o',t,y(:,2),'-.')
```



Compare Stiff ODE Solvers

The `ode15s` solver is a good first choice for most stiff problems. However, the other stiff solvers might be more efficient for certain types of problems. This example solves a stiff test equation using all four stiff ODE solvers.

Consider the test equation

$$y' = -\lambda y.$$

The equation becomes increasingly stiff as the magnitude of λ increases. Use $\lambda = 1 \times 10^9$ and the initial condition $y(0) = 1$ over the time interval `[0 0.5]`. With these values, the

problem is stiff enough that `ode45` and `ode23` struggle to integrate the equation. Also, use `odeset` to pass in the constant Jacobian $J = \frac{\partial f}{\partial y} = -\lambda$ and turn on the display of solver statistics.

```
lambda = 1e9;
y0 = 1;
tspan = [0 0.5];
opts = odeset('Jacobian',-lambda,'Stats','on');
```

Solve the equation with `ode15s`, `ode23s`, `ode23t`, and `ode23tb`. Make subplots for comparison.

```
subplot(2,2,1)
tic, ode15s(@(t,y) -lambda*y, tspan, y0, opts), toc
```

```
104 successful steps
1 failed attempts
212 function evaluations
0 partial derivatives
21 LU decompositions
210 solutions of linear systems
Elapsed time is 1.800760 seconds.
```

```
title('ode15s')
subplot(2,2,2)
tic, ode23s(@(t,y) -lambda*y, tspan, y0, opts), toc
```

```
63 successful steps
0 failed attempts
191 function evaluations
0 partial derivatives
63 LU decompositions
189 solutions of linear systems
Elapsed time is 0.472553 seconds.
```

```
title('ode23s')
subplot(2,2,3)
tic, ode23t(@(t,y) -lambda*y, tspan, y0, opts), toc
```

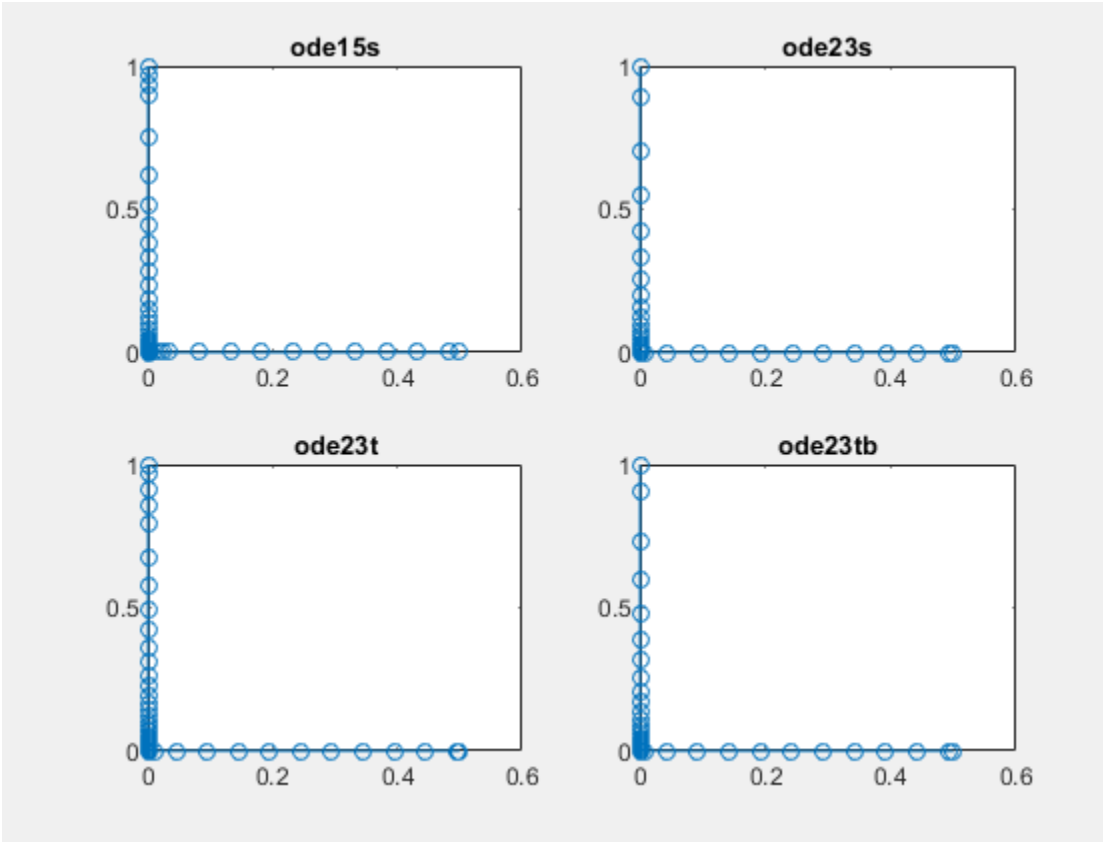
```
95 successful steps
0 failed attempts
125 function evaluations
0 partial derivatives
28 LU decompositions
```

```
123 solutions of linear systems
Elapsed time is 0.689998 seconds.
```

```
title('ode23t')
subplot(2,2,4)
tic, ode23tb(@(t,y) -lambda*y, tspan, y0, opts), toc
```

```
71 successful steps
0 failed attempts
167 function evaluations
0 partial derivatives
23 LU decompositions
236 solutions of linear systems
Elapsed time is 0.553235 seconds.
```

```
title('ode23tb')
```



The stiff solvers all perform well, but `ode23s` completes the integration with the fewest steps and runs the fastest for this particular problem. Since the constant Jacobian is specified, none of the solvers need to calculate partial derivatives to compute the solution. Specifying the Jacobian benefits `ode23s` the most since it normally evaluates the Jacobian in every step.

For general stiff problems, the performance of the stiff solvers varies depending on the format of the problem and specified options. Providing the Jacobian matrix or sparsity pattern always improves solver efficiency for stiff problems. But since the stiff solvers use the Jacobian differently, the improvement can vary significantly. Practically speaking, if a system of equations is very large or needs to be solved many times, then it is worthwhile to investigate the performance of the different solvers to minimize execution time.

Evaluate and Extend Solution Structure

The van der Pol equation is a second order ODE

$$y_1'' - \mu\left(1 - y_1^2\right)y_1' + y_1 = 0\,.$$

Solve the van der Pol equation with $\mu = 1000$ using `ode15s`. The function `vdp1000.m` ships with MATLAB® and encodes the equations. Specify a single output to return a structure containing information about the solution, such as the solver and evaluation points.

```
tspan = [0 3000];
y0 = [2 0];
sol = ode15s(@vdp1000,tspan,y0)
```

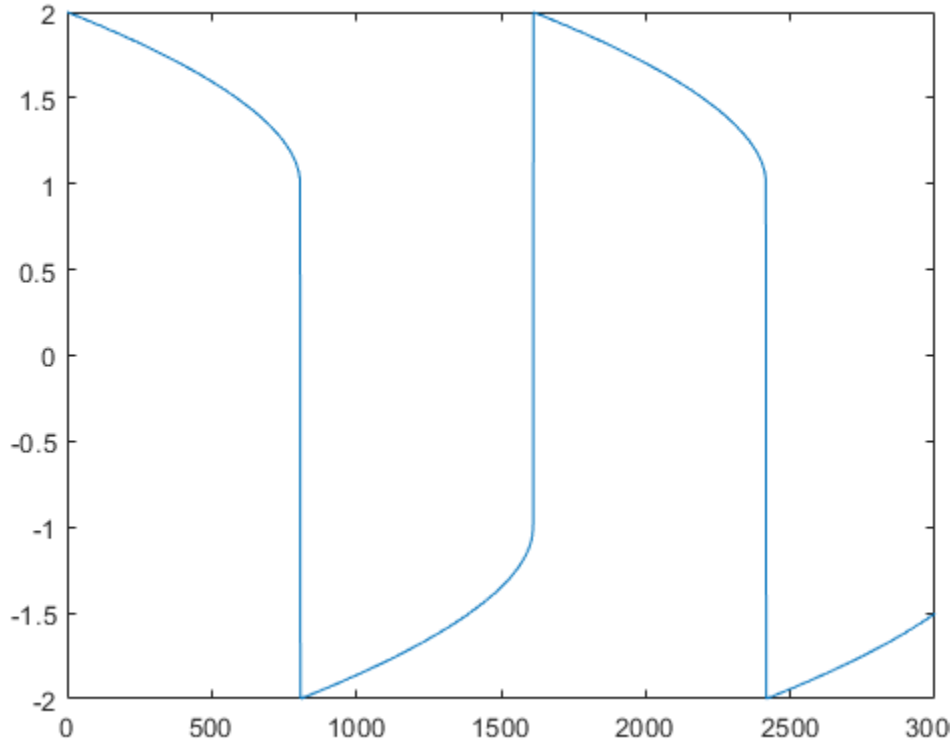
```
sol = struct with fields:
     solver: 'ode15s'
    extdata: [1x1 struct]
          x: [1x592 double]
          y: [2x592 double]
      stats: [1x1 struct]
      idata: [1x1 struct]
```

Use `linspace` to generate 2500 points in the interval `[0 3000]`. Evaluate the first component of the solution at these points using `deval`.

```
x = linspace(0,3000,2500);
y = deval(sol,x,1);
```

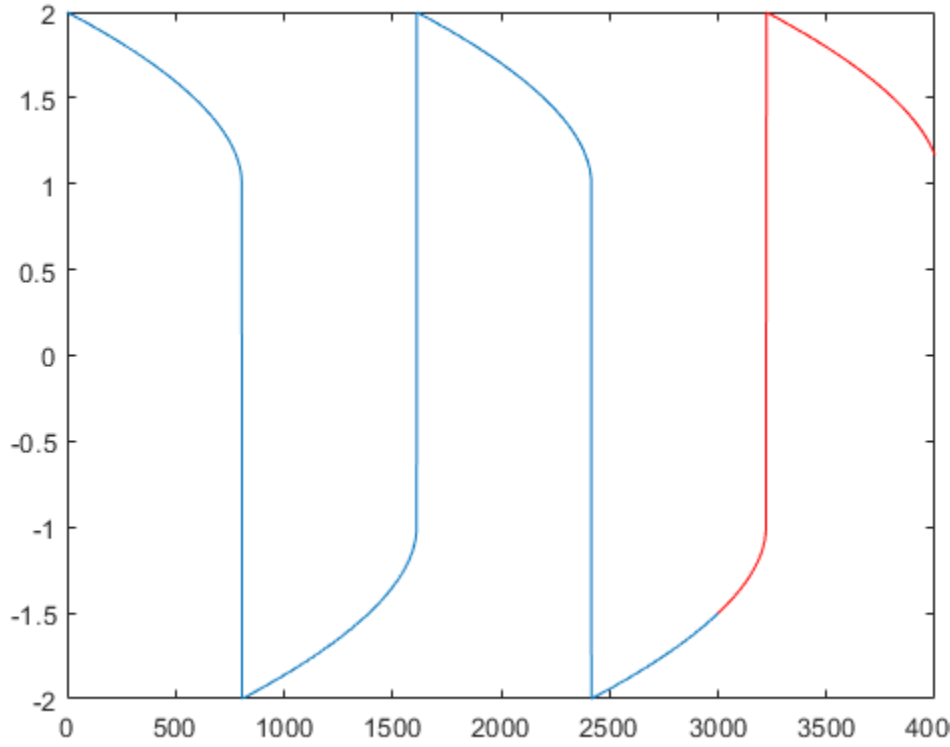
Plot the solution.

```
plot(x,y)
```



Extend the solution to $t_f = 4000$ using `odextend` and add the result to the original plot.

```
tf = 4000;
sol_new = odextend(sol,@vdp1000,tf);
x = linspace(3000,tf,350);
y = deval(sol_new,x,1);
hold on
plot(x,y,'r')
```



Solve Robertson Problem as Semi-Explicit Differential Algebraic Equations (DAEs)

This example reformulates a system of ODEs as a system of differential algebraic equations (DAEs). The Robertson problem found in hb1ode.m is a classic test problem for programs that solve stiff ODEs. The system of equations is

$$\begin{aligned} y_1' &= -0.04y_1 + 10^4 y_2 y_3 \\ y_2' &= 0.04y_1 - 10^4 y_2 y_3 - (3 \times 10^7) y_2^2 \\ y_3' &= (3 \times 10^7) y_2^2. \end{aligned}$$

`hb1ode` solves this system of ODEs to steady state with the initial conditions $y_1 = 1$, $y_2 = 0$, and $y_3 = 0$. But the equations also satisfy a linear conservation law,

$$y_1' + y_2' + y_3' = 0.$$

In terms of the solution and initial conditions, the conservation law is

$$y_1 + y_2 + y_3 = 1.$$

The system of equations can be rewritten as a system of DAEs by using the conservation law to determine the state of y_3. This reformulates the problem as the DAE system

$$\begin{aligned} y_1' &= -0.04y_1 + 10^4 y_2 y_3 \\ y_2' &= 0.04y_1 - 10^4 y_2 y_3 - (3 \times 10^7) y_2^2 \\ 0 &= y_1 + y_2 + y_3 - 1. \end{aligned}$$

The differential index of this system is 1, since only a single derivative of y_3 is required to make this a system of ODEs. Therefore, no further transformations are required before solving the system.

The function `robertsdae` encodes this DAE system. Save `robertsdae.m` in your current folder to run the example.

```
function out = robertsdae(t,y)
out = [-0.04*y(1) + 1e4*y(2).*y(3)
   0.04*y(1) - 1e4*y(2).*y(3) - 3e7*y(2).^2
   y(1) + y(2) + y(3) - 1 ];
```

The full example code for this formulation of the Robertson problem is available in hb1dae.m.

Solve the DAE system using `ode15s`. Consistent initial conditions for `y0` are obvious based on the conservation law. Use `odeset` to set the options:

- Use a constant mass matrix to represent the left hand side of the system of equations.

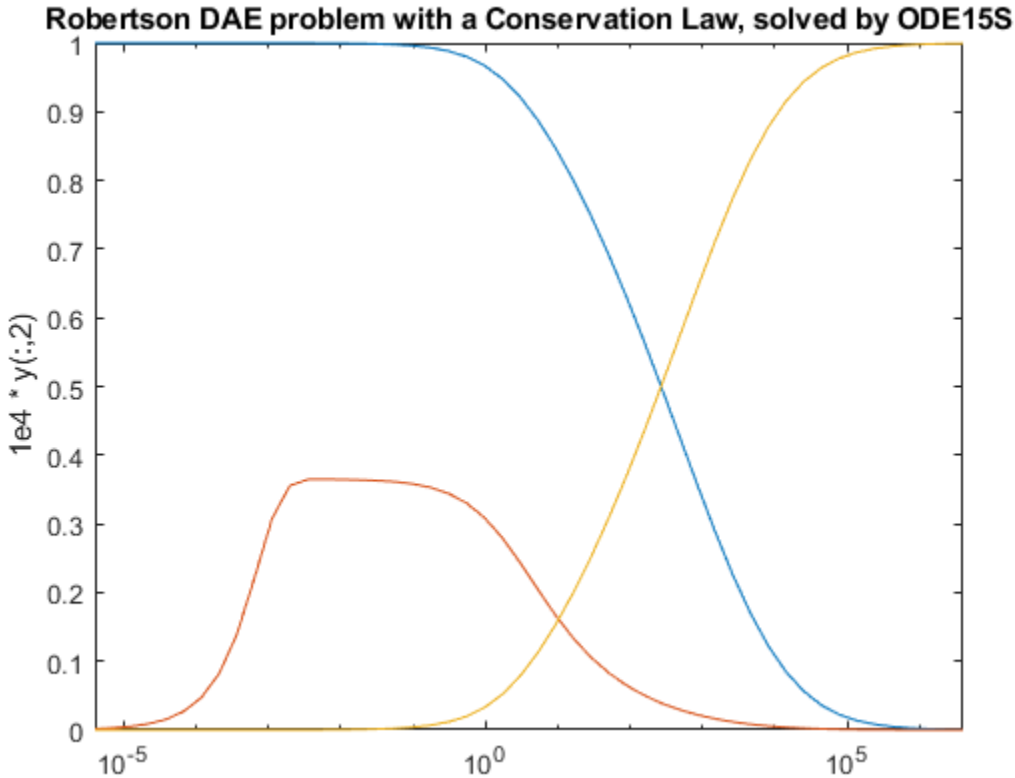
$$\begin{pmatrix} y_1' \\ y_2' \\ 0 \end{pmatrix} = My' \rightarrow M = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

- Set the relative error tolerance to `1e-4`.
- Use an absolute tolerance of `1e-10` for the second solution component, since the scale varies dramatically from the other components.
- Leave the `'MassSingular'` option at its default value `'maybe'` to test the automatic detection of a DAE.

```
y0 = [1; 0; 0];
tspan = [0 4*logspace(-6,6)];
M = [1 0 0; 0 1 0; 0 0 0];
options = odeset('Mass',M,'RelTol',1e-4,'AbsTol',[1e-6 1e-10 1e-6]);
[t,y] = ode15s(@robertsdae,tspan,y0,options);
```

Plot the solution.

```
y(:,2) = 1e4*y(:,2);
semilogx(t,y);
ylabel('1e4 * y(:,2)');
title('Robertson DAE problem with a Conservation Law, solved by ODE15S');
```



Input Arguments

odefun — Functions to solve
function handle

Functions to solve, specified as a function handle which defines the functions to be integrated.

The function `dydt = odefun(t,y)`, for a scalar `t` and a column vector `y`, must return a column vector `dydt` of data type `single` or `double` that corresponds to $f(t, y)$. `odefun` must accept both input arguments, `t` and `y`, even if one of the arguments is not used in the function.

For example, to solve $y' = 5y - 3$, use the function:

```
function dydt = odefun(t,y)
dydt = 5*y-3;
```

For a system of equations, the output of `odefun` is a vector. Each element in the vector is the solution to one equation. For example, to solve

$$y'_1 = y_1 + 2y_2$$
$$y'_2 = 3y_1 + 2y_2$$

use the function:

```
function dydt = odefun(t,y)
dydt = zeros(2,1);
dydt(1) = y(1)+2*y(2);
dydt(2) = 3*y(1)+2*y(2);
```

For information on how to provide additional parameters to the function `odefun`, see "Parameterizing Functions".

Example: `@myFcn`

Data Types: `function_handle`

`tspan` — Interval of integration
vector

Interval of integration, specified as a vector. At minimum, `tspan` must be a two element vector `[t0 tf]` specifying the initial and final times. To obtain solutions at specific times between `t0` and `tf`, use a longer vector of the form `[t0,t1,t2,...,tf]`. The elements in `tspan` must be all increasing or all decreasing.

The solver imposes the initial conditions given by `y0` at the initial time `tspan(1)`, then integrates from `tspan(1)` to `tspan(end)`:

- If `tspan` has two elements, `[t0 tf]`, then the solver returns the solution evaluated at each internal integration step within the interval.
- If `tspan` has more than two elements `[t0,t1,t2,...,tf]`, then the solver returns the solution evaluated at the given points. However, the solver does not step precisely to each point specified in `tspan`. Instead, the solver uses its own internal steps to compute the solution, then evaluates the solution at the requested points in `tspan`.

The solutions produced at the specified points are of the same order of accuracy as the solutions computed at each internal step.

Specifying several intermediate points has little effect on the efficiency of computation, but for large systems it can affect memory management.

The values of `tspan` are used by the solver to calculate suitable values for `InitialStep` and `MaxStep`:

- If `tspan` contains several intermediate points `[t0,t1,t2,...,tf]`, then the specified points give an indication of the scale for the problem, which can affect the value of `InitialStep` used by the solver. Therefore, the solution obtained by the solver might be different depending on whether you specify `tspan` as a two-element vector or as a vector with intermediate points.
- The initial and final values in `tspan` are used to calculate the maximum step size `MaxStep`. Therefore, changing the initial or final values in `tspan` could lead to the solver using a different step sequence, which might change the solution.

Example: `[1 10]`

Example: `[1 3 5 7 9 10]`

Data Types: `single` | `double`

`y0` — Initial conditions
vector

Initial conditions, specified as a vector. `y0` must be the same length as the vector output of `odefun`, so that `y0` contains an initial condition for each equation defined in `odefun`.

Data Types: `single` | `double`

`options` — Option structure
structure array

Option structure, specified as a structure array. Use the `odeset` function to create or modify the options structure. See “Summary of ODE Options” for a list of the options compatible with each solver.

Example: `options = odeset('RelTol',1e-5,'Stats','on','OutputFcn',@odeplot)` specifies a relative error tolerance of `1e-5`, turns on the display of solver statistics, and specifies the output function `@odeplot` to plot the solution as it is computed.

Data Types: `struct`

Output Arguments

`t` — Evaluation points
column vector

Evaluation points, returned as a column vector.

- If `tspan` contains two elements, `[t0 tf]`, then `t` contains the internal evaluation points used to perform the integration.
- If `tspan` contains more than two elements, then `t` is the same as `tspan`.

`y` — Solutions
array

Solutions, returned as an array. Each row in `y` corresponds to the solution at the value returned in the corresponding row of `t`.

`te` — Time of events
column vector

Time of events, returned as a column vector. The event times in `te` correspond to the solutions returned in `ye`, and `ie` specifies which event occurred.

`ye` — Solution at time of events
array

Solution at time of events, returned as an array. The event times in `te` correspond to the solutions returned in `ye`, and `ie` specifies which event occurred.

`ie` — Index of triggered event function
column vector

Index of triggered event function, returned as a column vector. The event times in `te` correspond to the solutions returned in `ye`, and `ie` specifies which event occurred.

`sol` — Structure for evaluation
structure array

Structure for evaluation, returned as a structure array. Use this structure with the `deval` function to evaluate the solution at any point in the interval `[t0 tf]`. The `sol` structure array always includes these fields:

Structure Field	Description
`sol.x`	Row vector of the steps chosen by the solver.
`sol.y`	Solutions. Each column `sol.y(:,i)` contains the solution at time `sol.x(i)`.
`sol.solver`	Solver name.

Additionally, if you specify the `Events` option and events are detected, then `sol` also includes these fields:

Structure Field	Description
`sol.xe`	Points when events occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

Algorithms

`ode15s` is a variable-step, variable-order (VSVO) solver based on the numerical differentiation formulas (NDFs) of orders 1 to 5. Optionally, it can use the backward differentiation formulas (BDFs, also known as Gear's method) that are usually less efficient. Like `ode113`, `ode15s` is a multistep solver. Use `ode15s` if `ode45` fails or is very inefficient and you suspect that the problem is stiff, or when solving a differential-algebraic equation (DAE) [1], [2].

References

[1] Shampine, L. F. and M. W. Reichelt, “The MATLAB ODE Suite,” *SIAM Journal on Scientific Computing*, Vol. 18, 1997, pp. 1–22.

[2] Shampine, L. F., M. W. Reichelt, and J.A. Kierzenka, “Solving Index-1 DAEs in MATLAB and Simulink,” *SIAM Review*, Vol. 41, 1999, pp. 538–552.

See Also

`deval` | `ode23s` | `ode23t` | `ode23tb` | `ode45` | `odeget` | `odeset`

Topics

“Choose an ODE Solver”
“Summary of ODE Options”
“Solve Stiff ODEs”
“Solve Differential Algebraic Equations (DAEs)”
“Troubleshoot Common ODE Problems”

Introduced before R2006a

ode23

Solve nonstiff differential equations — low order method

Syntax

```
[t,y] = ode23(odefun,tspan,y0)
[t,y] = ode23(odefun,tspan,y0,options)
[t,y,te,ye,ie] = ode23(odefun,tspan,y0,options)
sol = ode23( ___ )
```

Description

`[t,y] = ode23(odefun,tspan,y0)`, where `tspan = [t0 tf]`, integrates the system of differential equations $y' = f(t, y)$ from `t0` to `tf` with initial conditions `y0`. Each row in the solution array `y` corresponds to a value returned in column vector `t`.

All MATLAB ODE solvers can solve systems of equations of the form $y' = f(t, y)$, or problems that involve a mass matrix, $M(t, y)y' = f(t, y)$. The solvers all use similar syntaxes. The `ode23s` solver only can solve problems with a mass matrix if the mass matrix is constant. `ode15s` and `ode23t` can solve problems with a mass matrix that is singular, known as differential-algebraic equations (DAEs). Specify the mass matrix using the `Mass` option of `odeset`.

`[t,y] = ode23(odefun,tspan,y0,options)` also uses the integration settings defined by `options`, which is an argument created using the `odeset` function. For example, use the `AbsTol` and `RelTol` options to specify absolute and relative error tolerances, or the `Mass` option to provide a mass matrix.

`[t,y,te,ye,ie] = ode23(odefun,tspan,y0,options)` additionally finds where functions of (t,y), called event functions, are zero. In the output, `te` is the time of the event, `ye` is the solution at the time of the event, and `ie` is the index of the triggered event.

For each event function, specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the `'Events'` property to a function, such as `myEventFcn` or `@myEventFcn`, and creating a

corresponding function: [value,isterminal,direction] = myEventFcn(t,y). For more information, see “ODE Event Location”.

sol = ode23(___) returns a structure that you can use with deval to evaluate the solution at any point on the interval [t0 tf]. You can use any of the input argument combinations in previous syntaxes.

Examples

ODE with Single Solution Component

Simple ODEs that have a single solution component can be specified as an anonymous function in the call to the solver. The anonymous function must accept two inputs (t,y) even if one of the inputs is not used.

Solve the ODE

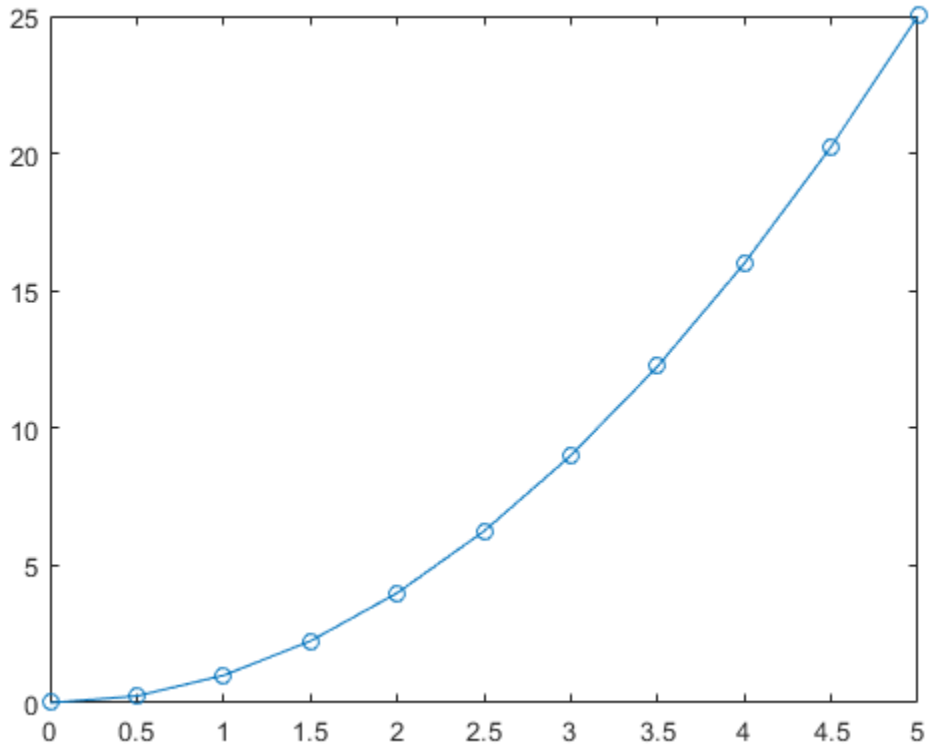
$$y' = 2t.$$

Use a time interval of [0,5] and the initial condition y0 = 0.

```
tspan = [0 5];
y0 = 0;
[t,y] = ode23(@(t,y) 2*t, tspan, y0);
```

Plot the solution.

```
plot(t,y,'-o')
```



Solve Nonstiff Equation

The van der Pol equation is a second order ODE

$$y_1'' - \mu\left(1 - y_1^2\right) y_1' + y_1 = 0,$$

where $\mu > 0$ is a scalar parameter. Rewrite this equation as a system of first-order ODEs by making the substitution $y_1' = y_2$. The resulting system of first-order ODEs is

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= \mu(1 - y_1^2)y_2 - y_1. \end{aligned}$$

The function file `vdp1.m` represents the van der Pol equation using $\mu = 1$. The variables y_1 and y_2 are the entries `y(1)` and `y(2)` of a two-element vector, `dydt`.

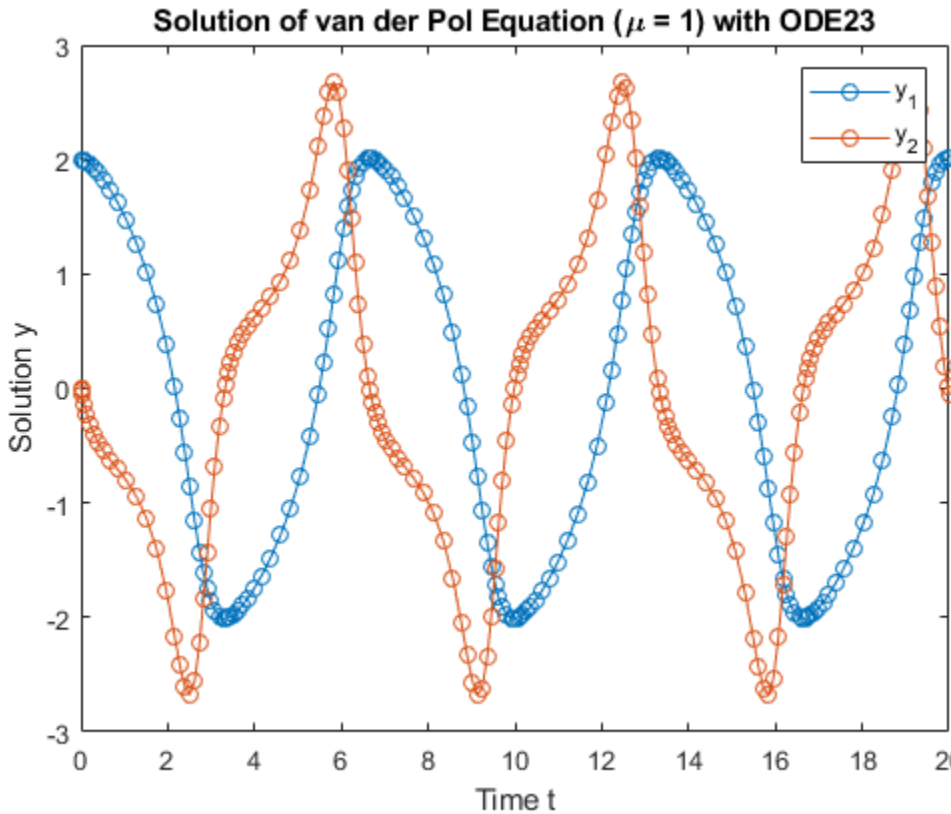
```
function dydt = vdp1(t,y)
%VDP1  Evaluate the van der Pol ODEs for mu = 1
%
%   See also ODE113, ODE23, ODE45.

%   Jacek Kierzenka and Lawrence F. Shampine
%   Copyright 1984-2014 The MathWorks, Inc.

dydt = [y(2); (1-y(1)^2)*y(2)-y(1)];
```

Solve the ODE using the `ode23` function on the time interval `[0 20]` with initial values `[2 0]`. The resulting output is a column vector of time points `t` and a solution array `y`. Each row in `y` corresponds to a time returned in the corresponding row of `t`. The first column of `y` corresponds to y_1, and the second column to y_2.

```
[t,y] = ode23(@vdp1,[0 20],[2; 0]);
```

Plot the solutions for y_1 and y_2 against `t`.

```
plot(t,y(:,1),'-o',t,y(:,2),'-o')
title('Solution of van der Pol Equation (\mu = 1) with ODE23');
xlabel('Time t');
ylabel('Solution y');
legend('y_1','y_2')
```



Pass Extra Parameters to ODE Function

`ode23` only works with functions that use two input arguments, `t` and `y`. However, you can pass in extra parameters by defining them outside the function and passing them in when you specify the function handle.

Solve the ODE

$$y'' = \frac{A}{B} t y.$$

Rewriting the equation as a first-order system yields

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= \frac{A}{B} t y_1. \end{aligned}$$

`odefcn.m` represents this system of equations as a function that accepts four input arguments: `t`, `y`, `A`, and `B`.

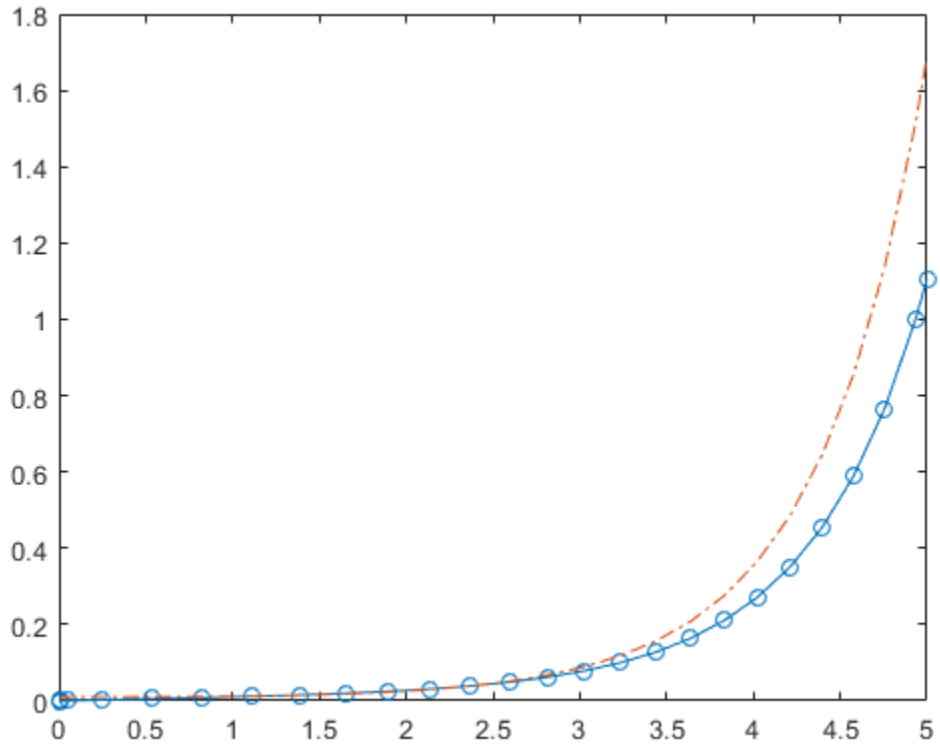
```
function dydt = odefcn(t,y,A,B)
dydt = zeros(2,1);
dydt(1) = y(2);
dydt(2) = (A/B)*t.*y(1);
```

Solve the ODE using `ode23`. Specify the function handle such that it passes in the predefined values for `A` and `B` to `odefcn`.

```
A = 1;
B = 2;
tspan = [0 5];
y0 = [0 0.01];
[t,y] = ode23(@(t,y) odefcn(t,y,A,B), tspan, y0);
```

Plot the results.

```
plot(t,y(:,1),'-o',t,y(:,2),'-.')
```



ODE With Crude Error Threshold

Compared to `ode45`, the `ode23` solver is better at solving problems with crude error tolerances.

Compare the performance of `ode45` and `ode23` by solving the moderately-stiff ODE

$$y' = -\lambda y$$

for $\lambda = 1000$. This ODE is a test equation that becomes increasingly stiff as λ increases. Use `odeset` to turn on the display of solver statistics.

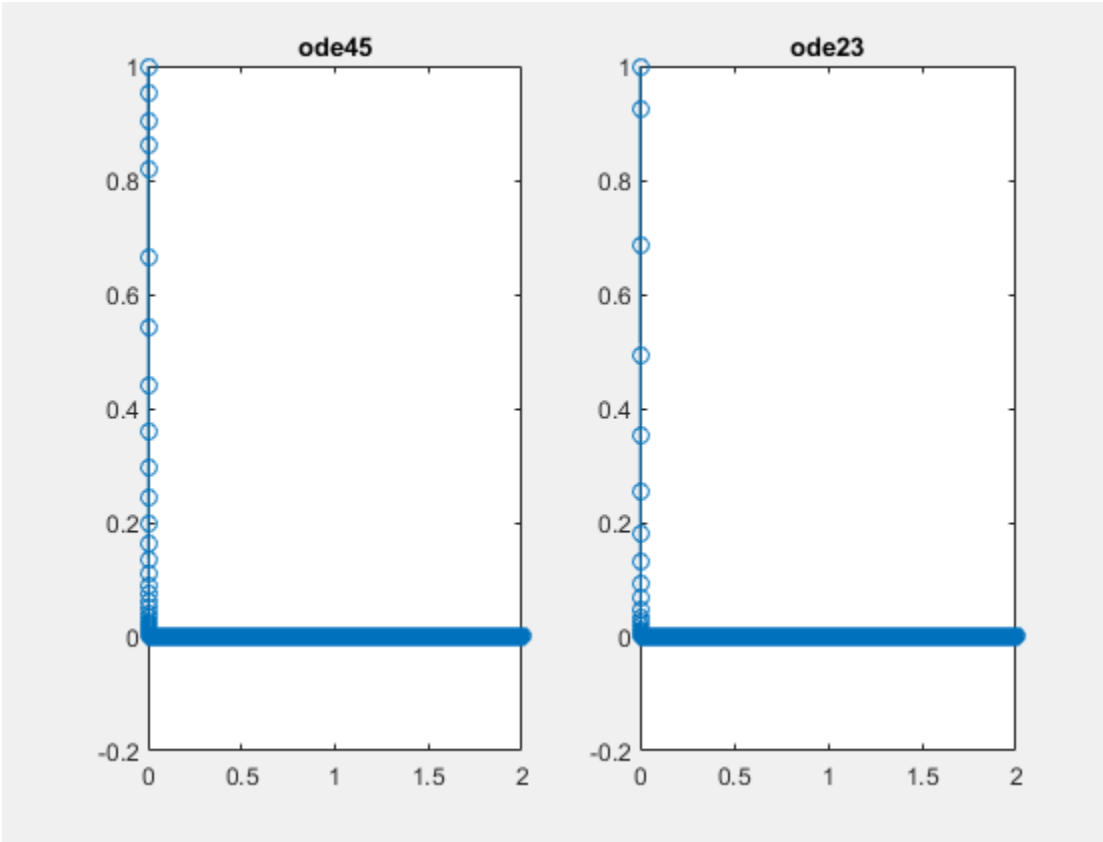
```
opts = odeset('Stats','on');
tspan = [0 2];
y0 = 1;
lambda = 1e3;
subplot(1,2,1)
tic, ode45(@(t,y) -lambda*y, tspan, y0, opts), toc
```

```
615 successful steps
35 failed attempts
3901 function evaluations
Elapsed time is 2.200831 seconds.
```

```
title('ode45')
subplot(1,2,2)
tic, ode23(@(t,y) -lambda*y, tspan, y0, opts), toc
```

```
822 successful steps
2 failed attempts
2473 function evaluations
Elapsed time is 1.502456 seconds.
```

```
title('ode23')
```



For this moderately stiff problem, `ode23` executes slightly faster than `ode45` and also has fewer failed steps. The step sizes taken by `ode45` and `ode23` for this problem are limited by the stability requirements of the equation rather than by accuracy. Since steps taken by `ode23` are cheaper than with `ode45`, the `ode23` solver executes quicker even though it takes more steps.

Input Arguments

`odefun` — Functions to solve
function handle

Functions to solve, specified as a function handle which defines the functions to be integrated.

The function `dydt = odefun(t,y)`, for a scalar `t` and a column vector `y`, must return a column vector `dydt` of data type `single` or `double` that corresponds to $f(t, y)$. `odefun` must accept both input arguments, `t` and `y`, even if one of the arguments is not used in the function.

For example, to solve $y' = 5y - 3$, use the function:

```
function dydt = odefun(t,y)
dydt = 5*y-3;
```

For a system of equations, the output of `odefun` is a vector. Each element in the vector is the solution to one equation. For example, to solve

$$y'_1 = y_1 + 2y_2$$
$$y'_2 = 3y_1 + 2y_2$$

use the function:

```
function dydt = odefun(t,y)
dydt = zeros(2,1);
dydt(1) = y(1)+2*y(2);
dydt(2) = 3*y(1)+2*y(2);
```

For information on how to provide additional parameters to the function `odefun`, see "Parameterizing Functions".

Example: `@myFcn`

Data Types: `function_handle`

`tspan` — Interval of integration
vector

Interval of integration, specified as a vector. At minimum, `tspan` must be a two element vector `[t0 tf]` specifying the initial and final times. To obtain solutions at specific times between `t0` and `tf`, use a longer vector of the form `[t0,t1,t2,...,tf]`. The elements in `tspan` must be all increasing or all decreasing.

The solver imposes the initial conditions given by `y0` at the initial time `tspan(1)`, then integrates from `tspan(1)` to `tspan(end)`:

- If `tspan` has two elements, `[t0 tf]`, then the solver returns the solution evaluated at each internal integration step within the interval.
- If `tspan` has more than two elements `[t0,t1,t2,...,tf]`, then the solver returns the solution evaluated at the given points. However, the solver does not step precisely to each point specified in `tspan`. Instead, the solver uses its own internal steps to compute the solution, then evaluates the solution at the requested points in `tspan`. The solutions produced at the specified points are of the same order of accuracy as the solutions computed at each internal step.

  Specifying several intermediate points has little effect on the efficiency of computation, but for large systems it can affect memory management.

The values of `tspan` are used by the solver to calculate suitable values for `InitialStep` and `MaxStep`:

- If `tspan` contains several intermediate points `[t0,t1,t2,...,tf]`, then the specified points give an indication of the scale for the problem, which can affect the value of `InitialStep` used by the solver. Therefore, the solution obtained by the solver might be different depending on whether you specify `tspan` as a two-element vector or as a vector with intermediate points.
- The initial and final values in `tspan` are used to calculate the maximum step size `MaxStep`. Therefore, changing the initial or final values in `tspan` could lead to the solver using a different step sequence, which might change the solution.

Example: `[1 10]`

Example: `[1 3 5 7 9 10]`

Data Types: `single` | `double`

`y0` — Initial conditions
vector

Initial conditions, specified as a vector. `y0` must be the same length as the vector output of `odefun`, so that `y0` contains an initial condition for each equation defined in `odefun`.

Data Types: `single` | `double`

`options` — Option structure
structure array

Option structure, specified as a structure array. Use the `odeset` function to create or modify the options structure. See “Summary of ODE Options” for a list of the options compatible with each solver.

Example: `options = odeset('RelTol',1e-5,'Stats','on','OutputFcn',@odeplot)` specifies a relative error tolerance of `1e-5`, turns on the display of solver statistics, and specifies the output function `@odeplot` to plot the solution as it is computed.

Data Types: `struct`

Output Arguments

`t` — Evaluation points
column vector

Evaluation points, returned as a column vector.

- If `tspan` contains two elements, `[t0 tf]`, then `t` contains the internal evaluation points used to perform the integration.
- If `tspan` contains more than two elements, then `t` is the same as `tspan`.

`y` — Solutions
array

Solutions, returned as an array. Each row in `y` corresponds to the solution at the value returned in the corresponding row of `t`.

`te` — Time of events
column vector

Time of events, returned as a column vector. The event times in `te` correspond to the solutions returned in `ye`, and `ie` specifies which event occurred.

`ye` — Solution at time of events
array

Solution at time of events, returned as an array. The event times in `te` correspond to the solutions returned in `ye`, and `ie` specifies which event occurred.

`ie` — Index of triggered event function
column vector

Index of triggered event function, returned as a column vector. The event times in `te` correspond to the solutions returned in `ye`, and `ie` specifies which event occurred.

`sol` — Structure for evaluation
structure array

Structure for evaluation, returned as a structure array. Use this structure with the `deval` function to evaluate the solution at any point in the interval `[t0 tf]`. The `sol` structure array always includes these fields:

Structure Field	Description
`sol.x`	Row vector of the steps chosen by the solver.
`sol.y`	Solutions. Each column `sol.y(:,i)` contains the solution at time `sol.x(i)`.
`sol.solver`	Solver name.

Additionally, if you specify the `Events` option and events are detected, then `sol` also includes these fields:

Structure Field	Description
`sol.xe`	Points when events occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

Algorithms

`ode23` is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than `ode45` at crude tolerances and in the presence of moderate stiffness. `ode23` is a single-step solver [1], [2].

References

[1] Bogacki, P. and L. F. Shampine, "A 3(2) pair of Runge-Kutta formulas," *Appl. Math. Letters*, Vol. 2, 1989, pp. 321–325.

[2] Shampine, L. F. and M. W. Reichelt, "The MATLAB ODE Suite," *SIAM Journal on Scientific Computing*, Vol. 18, 1997, pp. 1–22.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- All `odeset` option arguments must be constant.
- Code generation does not support a constant mass matrix in the options structure. Provide a mass matrix as a function.
- You must provide at least the two output arguments `T` and `Y`.
- Input types must be homogeneous—all double or all single.
- Variable-sizing support must be enabled. Code generation requires dynamic memory allocation when `tspan` has two elements or you use event functions.

See Also

`deval` | `ode45` | `odeget` | `odeset` | `odextend`

Topics

"Choose an ODE Solver"
"Summary of ODE Options"
"Solve Nonstiff ODEs"
"Troubleshoot Common ODE Problems"

Introduced before R2006a

ode23s

Solve stiff differential equations — low order method

Syntax

```
[t,y] = ode23s(odefun,tspan,y0)
[t,y] = ode23s(odefun,tspan,y0,options)
[t,y,te,ye,ie] = ode23s(odefun,tspan,y0,options)
sol = ode23s( ___ )
```

Description

`[t,y] = ode23s(odefun,tspan,y0)`, where `tspan = [t0 tf]`, integrates the system of differential equations $y' = f(t, y)$ from `t0` to `tf` with initial conditions `y0`. Each row in the solution array `y` corresponds to a value returned in column vector `t`.

All MATLAB ODE solvers can solve systems of equations of the form $y' = f(t, y)$, or problems that involve a mass matrix, $M(t, y)y' = f(t, y)$. The solvers all use similar syntaxes. The `ode23s` solver only can solve problems with a mass matrix if the mass matrix is constant. `ode15s` and `ode23t` can solve problems with a mass matrix that is singular, known as differential-algebraic equations (DAEs). Specify the mass matrix using the `Mass` option of `odeset`.

`[t,y] = ode23s(odefun,tspan,y0,options)` also uses the integration settings defined by `options`, which is an argument created using the `odeset` function. For example, use the `AbsTol` and `RelTol` options to specify absolute and relative error tolerances, or the `Mass` option to provide a mass matrix.

`[t,y,te,ye,ie] = ode23s(odefun,tspan,y0,options)` additionally finds where functions of (t,y), called event functions, are zero. In the output, `te` is the time of the event, `ye` is the solution at the time of the event, and `ie` is the index of the triggered event.

For each event function, specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the `'Events'` property to a function, such as `myEventFcn` or `@myEventFcn`, and creating a

corresponding function: [value,isterminal,direction] = myEventFcn(t,y). For more information, see “ODE Event Location”.

sol = ode23s(___) returns a structure that you can use with deval to evaluate the solution at any point on the interval [t0 tf]. You can use any of the input argument combinations in previous syntaxes.

Examples

ODE With Single Solution Component

Simple ODEs that have a single solution component can be specified as an anonymous function in the call to the solver. The anonymous function must accept two inputs (t,y) even if one of the inputs is not used.

Solve the ODE

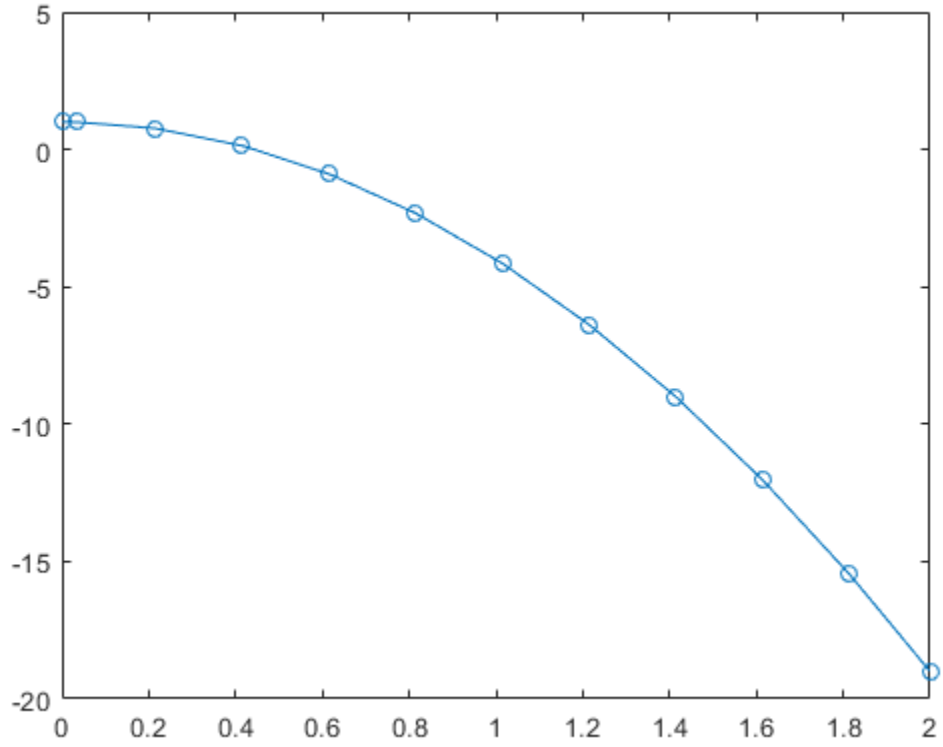
$$y' = -10t.$$

Use a time interval of [0,2] and the initial condition y0 = 1.

```
tspan = [0 2];
y0 = 1;
[t,y] = ode23s(@(t,y) -10*t, tspan, y0);
```

Plot the solution.

```
plot(t,y,'-o')
```



Solve Stiff ODE

An example of a stiff system of equations is the van der Pol equations in relaxation oscillation. The limit cycle has regions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

The system of equations is:

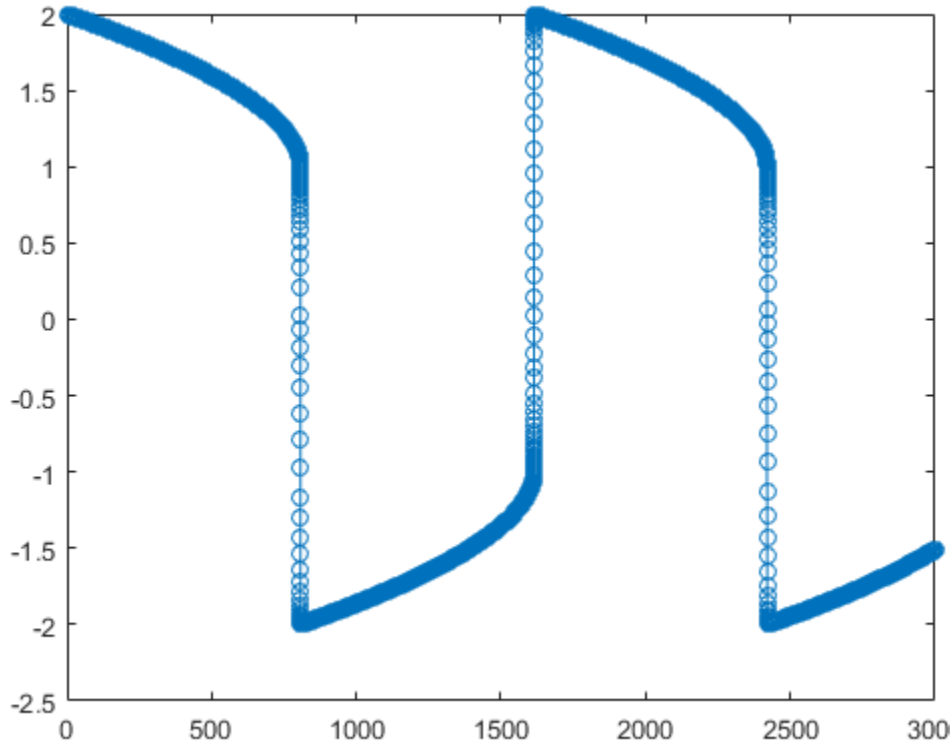
$$\begin{aligned} y_1' &= y_2 \\ y_2' &= 1000(1 - y_1^2)y_2 - y_1 \end{aligned}$$

The initial conditions are $y_1(0) = 2$ and $y_2(0) = 0$. The function `vdp1000` ships with MATLAB® and encodes the equations.

```
function dydt = vdp1000(t,y)
%VDP1000  Evaluate the van der Pol ODEs for mu = 1000.
%
%   See also ODE15S, ODE23S, ODE23T, ODE23TB.

%   Jacek Kierzenka and Lawrence F. Shampine
%   Copyright 1984-2014 The MathWorks, Inc.

dydt = [y(2); 1000*(1-y(1)^2)*y(2)-y(1)];
```

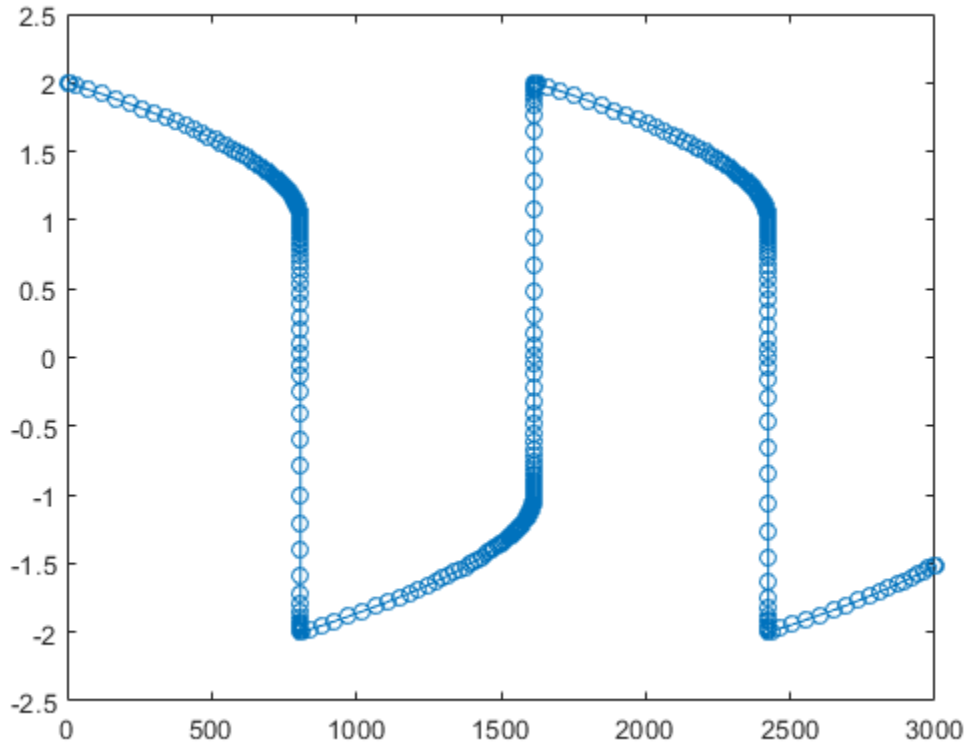
Solving this system using `ode45` with the default relative and absolute error tolerances (`1e-3` and `1e-6`, respectively) is extremely slow, requiring several minutes to solve and plot the solution. `ode45` requires millions of time steps to complete the integration, due to the areas of stiffness where it struggles to meet the tolerances.

This is a plot of the solution obtained by `ode45`, which takes a long time to compute. Notice the enormous number of time steps required to pass through areas of stiffness.



Solve the stiff system using the `ode23s` solver, and then plot the first column of the solution `y` against the time points `t`. The `ode23s` solver passes through stiff areas with far fewer steps than `ode45`.

```
[t,y] = ode23s(@vdp1000,[0 3000],[2 0]);
plot(t,y(:,1),'-o')
```



Pass Extra Parameters to ODE Function

`ode23s` only works with functions that use two input arguments, `t` and `y`. However, you can pass in extra parameters by defining them outside the function and passing them in when you specify the function handle.

Solve the ODE

$$y'' = \frac{A}{B} t y.$$

Rewriting the equation as a first-order system yields

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= \frac{A}{B} t y_1. \end{aligned}$$

`odefcn.m` represents this system of equations as a function that accepts four input arguments: `t`, `y`, `A`, and `B`.

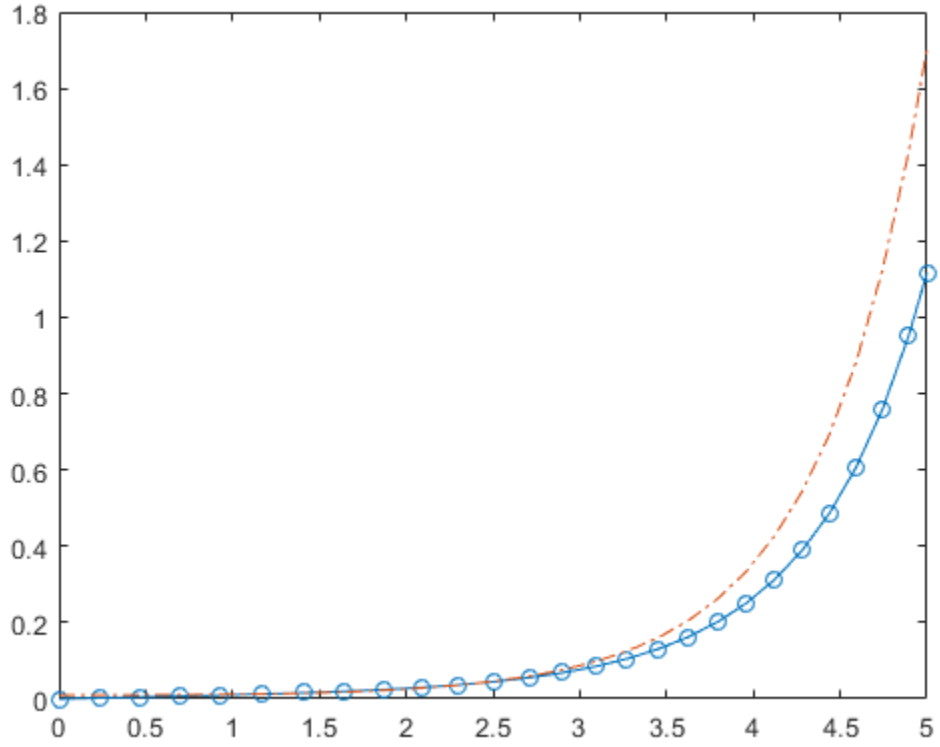
```
function dydt = odefcn(t,y,A,B)
dydt = zeros(2,1);
dydt(1) = y(2);
dydt(2) = (A/B)*t.*y(1);
```

Solve the ODE using `ode23s`. Specify the function handle such that it passes in the predefined values for `A` and `B` to `odefcn`.

```
A = 1;
B = 2;
tspan = [0 5];
y0 = [0 0.01];
[t,y] = ode23s(@(t,y) odefcn(t,y,A,B), tspan, y0);
```

Plot the results.

```
plot(t,y(:,1),'-o',t,y(:,2),'-.')
```



Compare Stiff ODE Solvers

The `ode15s` solver is a good first choice for most stiff problems. However, the other stiff solvers might be more efficient for certain types of problems. This example solves a stiff test equation using all four stiff ODE solvers.

Consider the test equation

$$y' = -\lambda y.$$

The equation becomes increasingly stiff as the magnitude of λ increases. Use $\lambda = 1 \times 10^9$ and the initial condition $y(0) = 1$ over the time interval `[0 0.5]`. With these values, the

problem is stiff enough that `ode45` and `ode23` struggle to integrate the equation. Also, use `odeset` to pass in the constant Jacobian $J = \frac{\partial f}{\partial y} = -\lambda$ and turn on the display of solver statistics.

```
lambda = 1e9;
y0 = 1;
tspan = [0 0.5];
opts = odeset('Jacobian',-lambda,'Stats','on');
```

Solve the equation with `ode15s`, `ode23s`, `ode23t`, and `ode23tb`. Make subplots for comparison.

```
subplot(2,2,1)
tic, ode15s(@(t,y) -lambda*y, tspan, y0, opts), toc
```

```
104 successful steps
1 failed attempts
212 function evaluations
0 partial derivatives
21 LU decompositions
210 solutions of linear systems
Elapsed time is 1.800760 seconds.
```

```
title('ode15s')
subplot(2,2,2)
tic, ode23s(@(t,y) -lambda*y, tspan, y0, opts), toc
```

```
63 successful steps
0 failed attempts
191 function evaluations
0 partial derivatives
63 LU decompositions
189 solutions of linear systems
Elapsed time is 0.472553 seconds.
```

```
title('ode23s')
subplot(2,2,3)
tic, ode23t(@(t,y) -lambda*y, tspan, y0, opts), toc
```

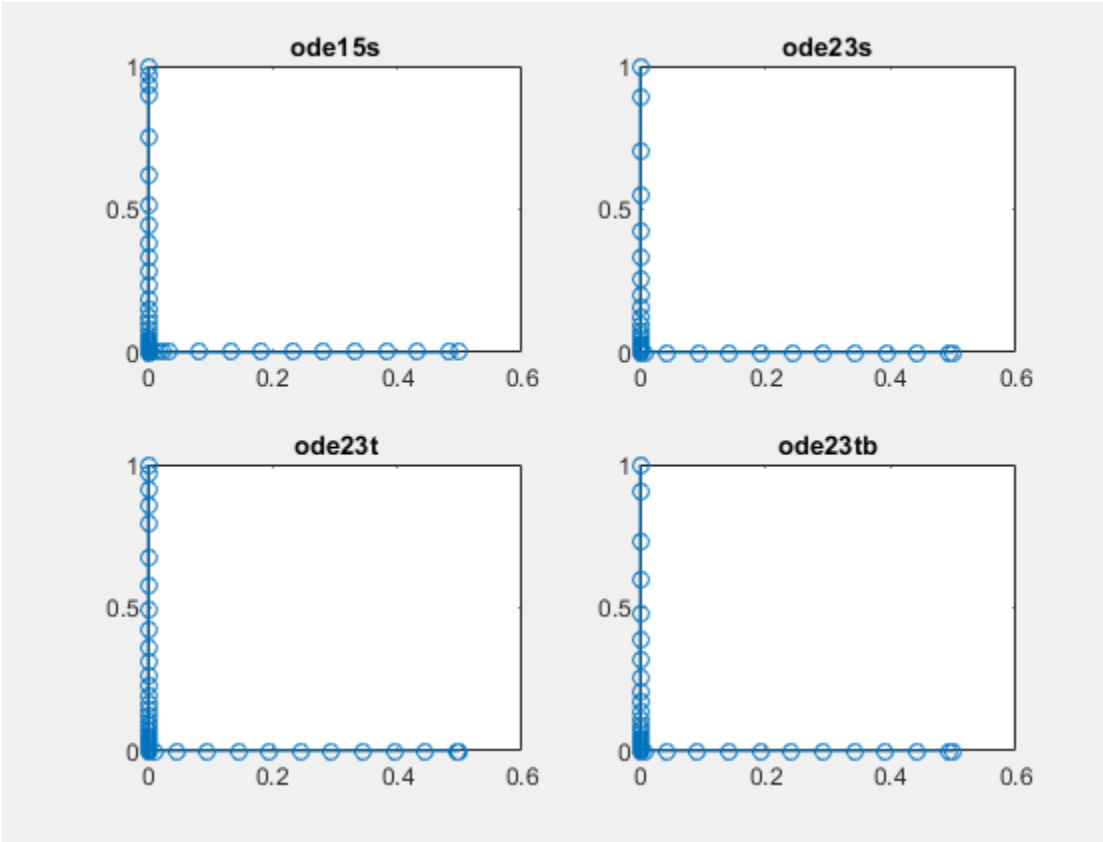
```
95 successful steps
0 failed attempts
125 function evaluations
0 partial derivatives
28 LU decompositions
```

```
123 solutions of linear systems
Elapsed time is 0.689998 seconds.
```

```
title('ode23t')
subplot(2,2,4)
tic, ode23tb(@(t,y) -lambda*y, tspan, y0, opts), toc
```

```
71 successful steps
0 failed attempts
167 function evaluations
0 partial derivatives
23 LU decompositions
236 solutions of linear systems
Elapsed time is 0.553235 seconds.
```

```
title('ode23tb')
```



The stiff solvers all perform well, but `ode23s` completes the integration with the fewest steps and runs the fastest for this particular problem. Since the constant Jacobian is specified, none of the solvers need to calculate partial derivatives to compute the solution. Specifying the Jacobian benefits `ode23s` the most since it normally evaluates the Jacobian in every step.

For general stiff problems, the performance of the stiff solvers varies depending on the format of the problem and specified options. Providing the Jacobian matrix or sparsity pattern always improves solver efficiency for stiff problems. But since the stiff solvers use the Jacobian differently, the improvement can vary significantly. Practically speaking, if a system of equations is very large or needs to be solved many times, then it is worthwhile to investigate the performance of the different solvers to minimize execution time.

Input Arguments

odefun — Functions to solve
function handle

Functions to solve, specified as a function handle which defines the functions to be integrated.

The function `dydt = odefun(t,y)`, for a scalar `t` and a column vector `y`, must return a column vector `dydt` of data type `single` or `double` that corresponds to $f(t, y)$. `odefun` must accept both input arguments, `t` and `y`, even if one of the arguments is not used in the function.

For example, to solve $y' = 5y - 3$, use the function:

```
function dydt = odefun(t,y)
dydt = 5*y-3;
```

For a system of equations, the output of `odefun` is a vector. Each element in the vector is the solution to one equation. For example, to solve

$$y'_1 = y_1 + 2y_2$$
$$y'_2 = 3y_1 + 2y_2$$

use the function:

```
function dydt = odefun(t,y)
dydt = zeros(2,1);
dydt(1) = y(1)+2*y(2);
dydt(2) = 3*y(1)+2*y(2);
```

For information on how to provide additional parameters to the function `odefun`, see “Parameterizing Functions”.

Example: `@myFcn`

Data Types: `function_handle`

tspan — Interval of integration
vector

Interval of integration, specified as a vector. At minimum, `tspan` must be a two element vector `[t0 tf]` specifying the initial and final times. To obtain solutions at specific times

between `t0` and `tf`, use a longer vector of the form `[t0,t1,t2,...,tf]`. The elements in `tspan` must be all increasing or all decreasing.

The solver imposes the initial conditions given by `y0` at the initial time `tspan(1)`, then integrates from `tspan(1)` to `tspan(end)`:

- If `tspan` has two elements, `[t0 tf]`, then the solver returns the solution evaluated at each internal integration step within the interval.
- If `tspan` has more than two elements `[t0,t1,t2,...,tf]`, then the solver returns the solution evaluated at the given points. However, the solver does not step precisely to each point specified in `tspan`. Instead, the solver uses its own internal steps to compute the solution, then evaluates the solution at the requested points in `tspan`. The solutions produced at the specified points are of the same order of accuracy as the solutions computed at each internal step.

  Specifying several intermediate points has little effect on the efficiency of computation, but for large systems it can affect memory management.

The values of `tspan` are used by the solver to calculate suitable values for `InitialStep` and `MaxStep`:

- If `tspan` contains several intermediate points `[t0,t1,t2,...,tf]`, then the specified points give an indication of the scale for the problem, which can affect the value of `InitialStep` used by the solver. Therefore, the solution obtained by the solver might be different depending on whether you specify `tspan` as a two-element vector or as a vector with intermediate points.
- The initial and final values in `tspan` are used to calculate the maximum step size `MaxStep`. Therefore, changing the initial or final values in `tspan` could lead to the solver using a different step sequence, which might change the solution.

Example: `[1 10]`

Example: `[1 3 5 7 9 10]`

Data Types: `single` | `double`

`y0` — Initial conditions
vector

Initial conditions, specified as a vector. `y0` must be the same length as the vector output of `odefun`, so that `y0` contains an initial condition for each equation defined in `odefun`.

Data Types: `single` | `double`

options — Option structure
structure array

Option structure, specified as a structure array. Use the `odeset` function to create or modify the options structure. See “Summary of ODE Options” for a list of the options compatible with each solver.

Example: `options = odeset('RelTol',1e-5,'Stats','on','OutputFcn',@odeplot)` specifies a relative error tolerance of `1e-5`, turns on the display of solver statistics, and specifies the output function `@odeplot` to plot the solution as it is computed.

Data Types: `struct`

Output Arguments

t — Evaluation points
column vector

Evaluation points, returned as a column vector.

- If `tspan` contains two elements, `[t0 tf]`, then `t` contains the internal evaluation points used to perform the integration.
- If `tspan` contains more than two elements, then `t` is the same as `tspan`.

y — Solutions
array

Solutions, returned as an array. Each row in `y` corresponds to the solution at the value returned in the corresponding row of `t`.

te — Time of events
column vector

Time of events, returned as a column vector. The event times in `te` correspond to the solutions returned in `ye`, and `ie` specifies which event occurred.

ye — Solution at time of events
array

Solution at time of events, returned as an array. The event times in `te` correspond to the solutions returned in `ye`, and `ie` specifies which event occurred.

`ie` — Index of triggered event function
column vector

Index of triggered event function, returned as a column vector. The event times in `te` correspond to the solutions returned in `ye`, and `ie` specifies which event occurred.

`sol` — Structure for evaluation
structure array

Structure for evaluation, returned as a structure array. Use this structure with the `deval` function to evaluate the solution at any point in the interval `[t0 tf]`. The `sol` structure array always includes these fields:

Structure Field	Description
`sol.x`	Row vector of the steps chosen by the solver.
`sol.y`	Solutions. Each column `sol.y(:,i)` contains the solution at time `sol.x(i)`.
`sol.solver`	Solver name.

Additionally, if you specify the `Events` option and events are detected, then `sol` also includes these fields:

Structure Field	Description
`sol.xe`	Points when events occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

Algorithms

`ode23s` is based on a modified Rosenbrock formula of order 2. Because it is a single-step solver, it may be more efficient than `ode15s` at solving problems that permit crude

tolerances or problems with solutions that change rapidly. It can solve some kinds of stiff problems for which `ode15s` is not effective. The `ode23s` solver evaluates the Jacobian during each step of the integration, so supplying it with the Jacobian matrix is critical to its reliability and efficiency [1].

References

[1] Shampine, L. F. and M. W. Reichelt, "The MATLAB ODE Suite," *SIAM Journal on Scientific Computing*, Vol. 18, 1997, pp. 1–22.

See Also

`deval` | `ode15s` | `odeget` | `odeset`

Topics

"Choose an ODE Solver"
"Summary of ODE Options"
"Solve Stiff ODEs"
"Troubleshoot Common ODE Problems"

Introduced before R2006a

ode23t

Solve moderately stiff ODEs and DAEs — trapezoidal rule

Syntax

```
[t,y] = ode23t(odefun,tspan,y0)
[t,y] = ode23t(odefun,tspan,y0,options)
[t,y,te,ye,ie] = ode23t(odefun,tspan,y0,options)
sol = ode23t( ___ )
```

Description

`[t,y] = ode23t(odefun,tspan,y0)`, where `tspan = [t0 tf]`, integrates the system of differential equations $y' = f(t, y)$ from `t0` to `tf` with initial conditions `y0`. Each row in the solution array `y` corresponds to a value returned in column vector `t`.

All MATLAB ODE solvers can solve systems of equations of the form $y' = f(t, y)$, or problems that involve a mass matrix, $M(t, y)y' = f(t, y)$. The solvers all use similar syntaxes. The `ode23s` solver only can solve problems with a mass matrix if the mass matrix is constant. `ode15s` and `ode23t` can solve problems with a mass matrix that is singular, known as differential-algebraic equations (DAEs). Specify the mass matrix using the `Mass` option of `odeset`.

`[t,y] = ode23t(odefun,tspan,y0,options)` also uses the integration settings defined by `options`, which is an argument created using the `odeset` function. For example, use the `AbsTol` and `RelTol` options to specify absolute and relative error tolerances, or the `Mass` option to provide a mass matrix.

`[t,y,te,ye,ie] = ode23t(odefun,tspan,y0,options)` additionally finds where functions of (t,y), called event functions, are zero. In the output, `te` is the time of the event, `ye` is the solution at the time of the event, and `ie` is the index of the triggered event.

For each event function, specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the `'Events'` property to a function, such as `myEventFcn` or `@myEventFcn`, and creating a

corresponding function: [value,isterminal,direction] = myEventFcn(t,y). For more information, see “ODE Event Location”.

sol = ode23t(___) returns a structure that you can use with deval to evaluate the solution at any point on the interval [t0 tf]. You can use any of the input argument combinations in previous syntaxes.

Examples

ODE With Single Solution Component

Simple ODEs that have a single solution component can be specified as an anonymous function in the call to the solver. The anonymous function must accept two inputs (t,y) even if one of the inputs is not used.

Solve the ODE

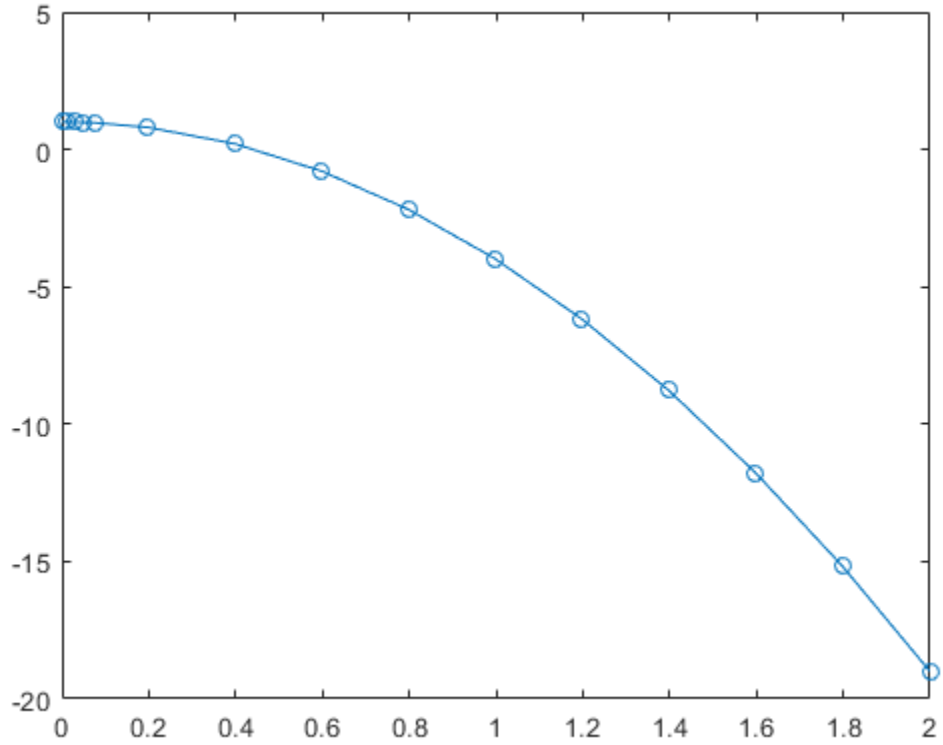
$$y' = -10t.$$

Use a time interval of [0,2] and the initial condition y0 = 1.

```
tspan = [0 2];
y0 = 1;
[t,y] = ode23t(@(t,y) -10*t, tspan, y0);
```

Plot the solution.

```
plot(t,y,'-o')
```



Solve Stiff ODE

An example of a stiff system of equations is the van der Pol equations in relaxation oscillation. The limit cycle has regions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

The system of equations is:

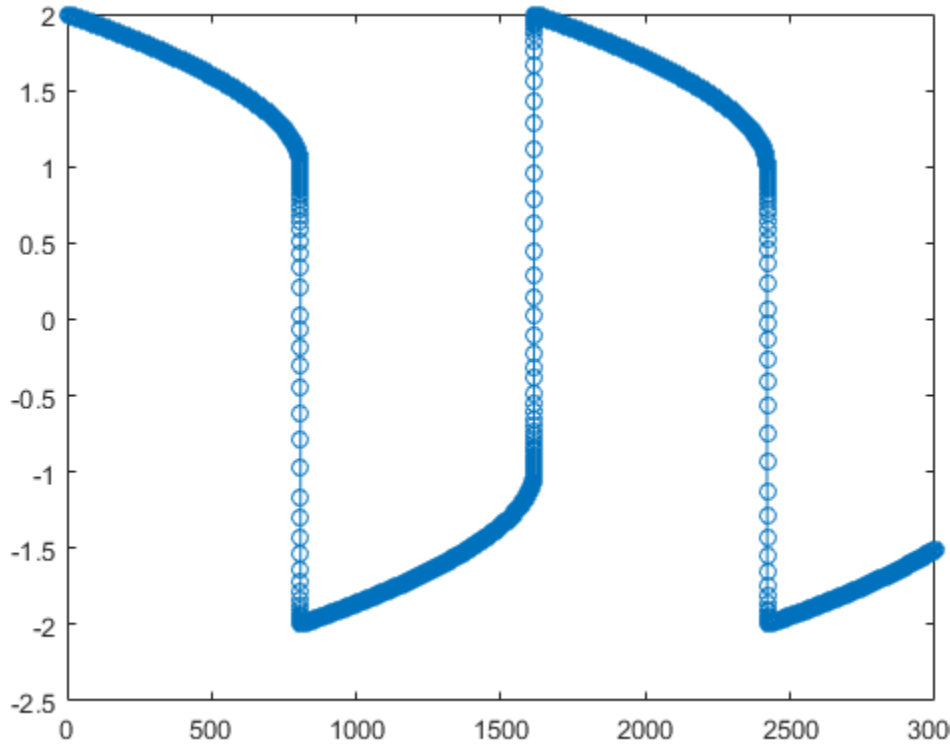
$$\begin{aligned} y_1' &= y_2 \\ y_2' &= 1000(1 - y_1^2)y_2 - y_1 \end{aligned}$$

The initial conditions are $y_1(0) = 2$ and $y_2(0) = 0$. The function `vdp1000` ships with MATLAB® and encodes the equations.

```
function dydt = vdp1000(t,y)
%VDP1000  Evaluate the van der Pol ODEs for mu = 1000.
%
%   See also ODE15S, ODE23S, ODE23T, ODE23TB.

%   Jacek Kierzenka and Lawrence F. Shampine
%   Copyright 1984-2014 The MathWorks, Inc.

dydt = [y(2); 1000*(1-y(1)^2)*y(2)-y(1)];
```

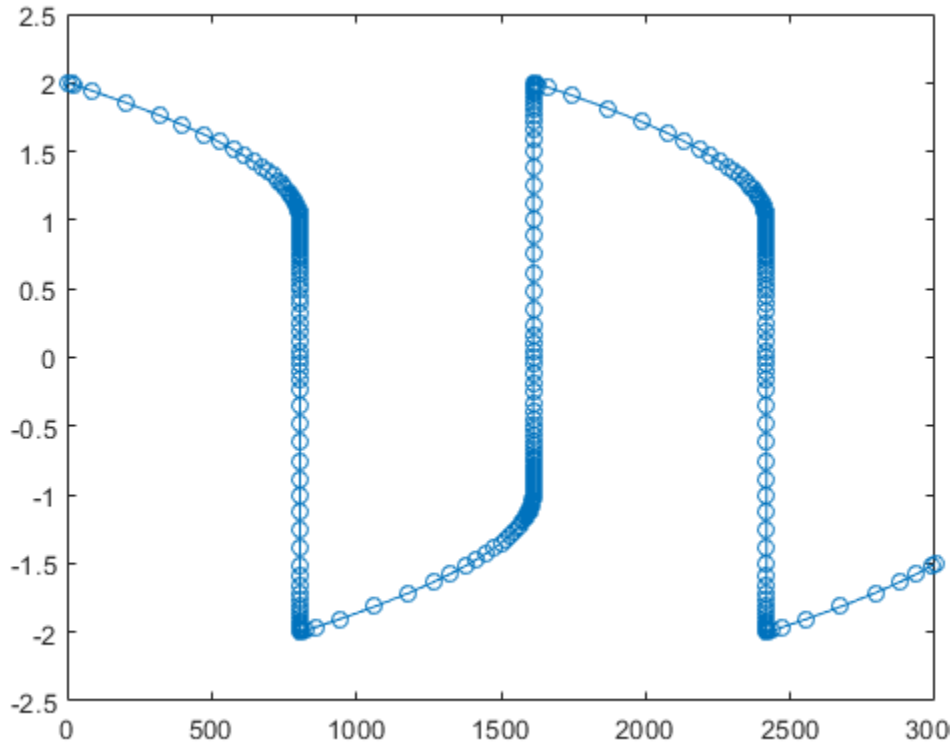
Solving this system using `ode45` with the default relative and absolute error tolerances (`1e-3` and `1e-6`, respectively) is extremely slow, requiring several minutes to solve and plot the solution. `ode45` requires millions of time steps to complete the integration, due to the areas of stiffness where it struggles to meet the tolerances.

This is a plot of the solution obtained by `ode45`, which takes a long time to compute. Notice the enormous number of time steps required to pass through areas of stiffness.



Solve the stiff system using the `ode23t` solver, and then plot the first column of the solution `y` against the time points `t`. The `ode23t` solver passes through stiff areas with far fewer steps than `ode45`.

```
[t,y] = ode23t(@vdp1000,[0 3000],[2 0]);
plot(t,y(:,1),'-o')
```



Pass Extra Parameters to ODE Function

`ode23t` only works with functions that use two input arguments, `t` and `y`. However, you can pass in extra parameters by defining them outside the function and passing them in when you specify the function handle.

Solve the ODE

$$y'' = \frac{A}{B} t y.$$

Rewriting the equation as a first-order system yields

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= \frac{A}{B} t y_1. \end{aligned}$$

`odefcn.m` represents this system of equations as a function that accepts four input arguments: `t`, `y`, `A`, and `B`.

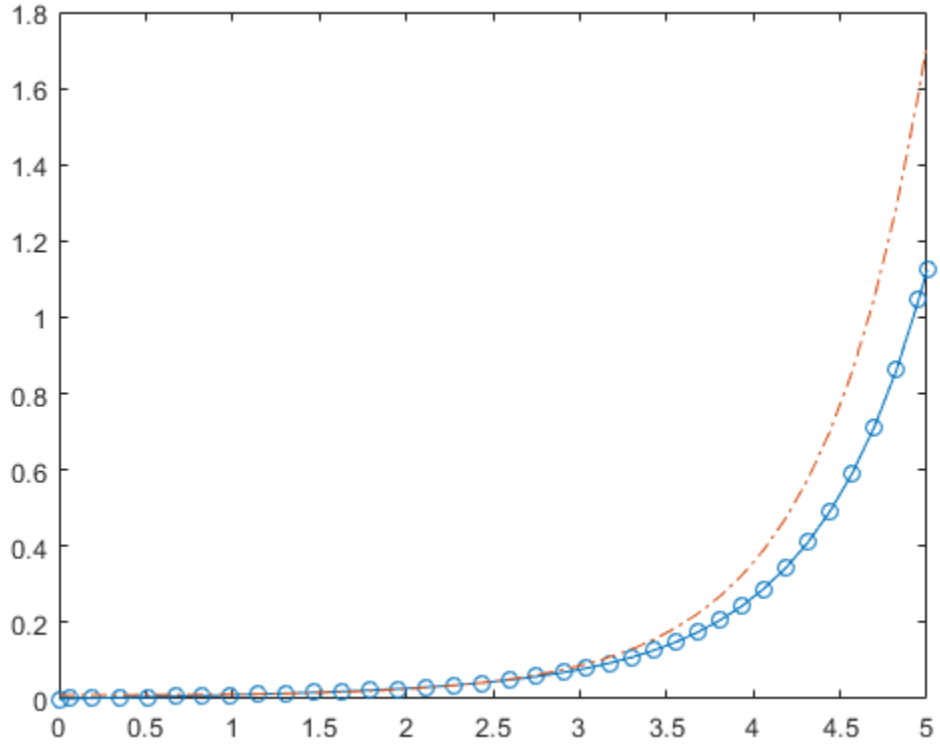
```
function dydt = odefcn(t,y,A,B)
dydt = zeros(2,1);
dydt(1) = y(2);
dydt(2) = (A/B)*t.*y(1);
```

Solve the ODE using `ode23t`. Specify the function handle such that it passes in the predefined values for `A` and `B` to `odefcn`.

```
A = 1;
B = 2;
tspan = [0 5];
y0 = [0 0.01];
[t,y] = ode23t(@(t,y) odefcn(t,y,A,B), tspan, y0);
```

Plot the results.

```
plot(t,y(:,1),'-o',t,y(:,2),'-.')
```



Compare Stiff ODE Solvers

The `ode15s` solver is a good first choice for most stiff problems. However, the other stiff solvers might be more efficient for certain types of problems. This example solves a stiff test equation using all four stiff ODE solvers.

Consider the test equation

$$y' = -\lambda y.$$

The equation becomes increasingly stiff as the magnitude of λ increases. Use $\lambda = 1 \times 10^9$ and the initial condition $y(0) = 1$ over the time interval `[0 0.5]`. With these values, the

problem is stiff enough that `ode45` and `ode23` struggle to integrate the equation. Also, use `odeset` to pass in the constant Jacobian $J = \frac{\partial f}{\partial y} = -\lambda$ and turn on the display of solver statistics.

```
lambda = 1e9;
y0 = 1;
tspan = [0 0.5];
opts = odeset('Jacobian',-lambda,'Stats','on');
```

Solve the equation with `ode15s`, `ode23s`, `ode23t`, and `ode23tb`. Make subplots for comparison.

```
subplot(2,2,1)
tic, ode15s(@(t,y) -lambda*y, tspan, y0, opts), toc
```

```
104 successful steps
1 failed attempts
212 function evaluations
0 partial derivatives
21 LU decompositions
210 solutions of linear systems
Elapsed time is 1.800760 seconds.
```

```
title('ode15s')
subplot(2,2,2)
tic, ode23s(@(t,y) -lambda*y, tspan, y0, opts), toc
```

```
63 successful steps
0 failed attempts
191 function evaluations
0 partial derivatives
63 LU decompositions
189 solutions of linear systems
Elapsed time is 0.472553 seconds.
```

```
title('ode23s')
subplot(2,2,3)
tic, ode23t(@(t,y) -lambda*y, tspan, y0, opts), toc
```

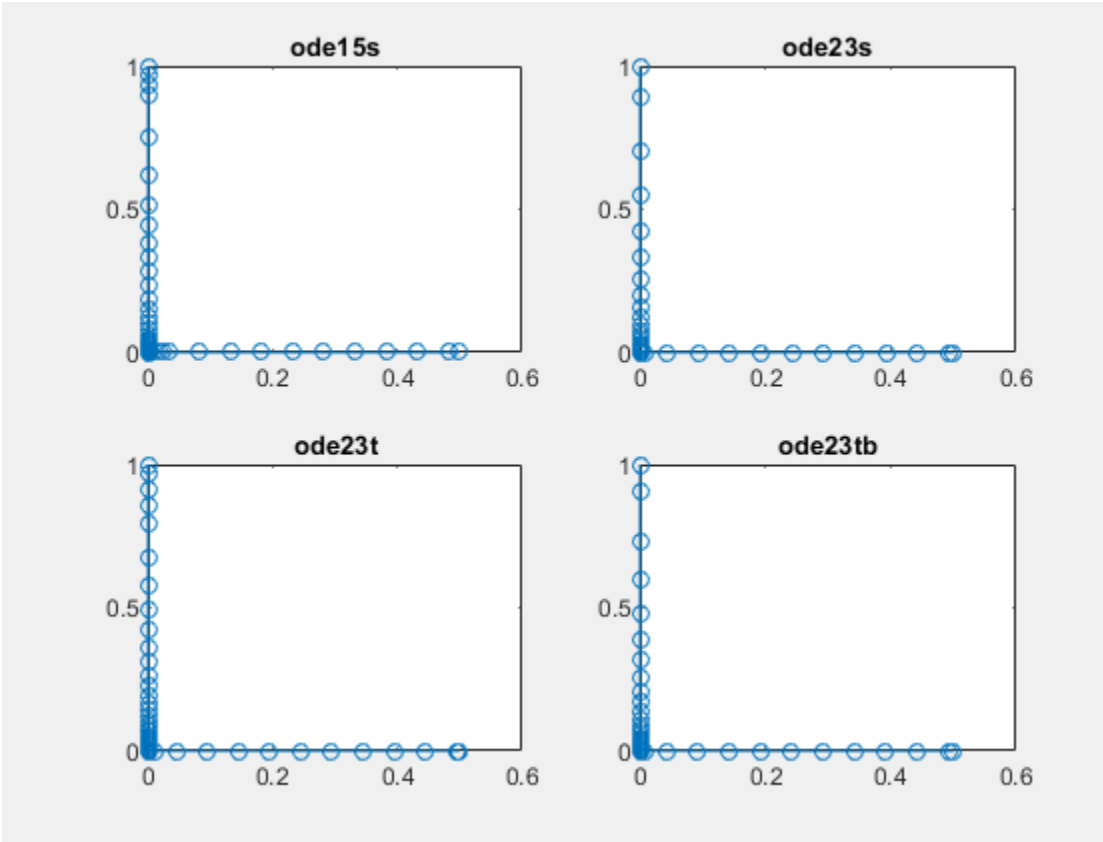
```
95 successful steps
0 failed attempts
125 function evaluations
0 partial derivatives
28 LU decompositions
```

```
123 solutions of linear systems
Elapsed time is 0.689998 seconds.
```

```
title('ode23t')
subplot(2,2,4)
tic, ode23tb(@(t,y) -lambda*y, tspan, y0, opts), toc
```

```
71 successful steps
0 failed attempts
167 function evaluations
0 partial derivatives
23 LU decompositions
236 solutions of linear systems
Elapsed time is 0.553235 seconds.
```

```
title('ode23tb')
```



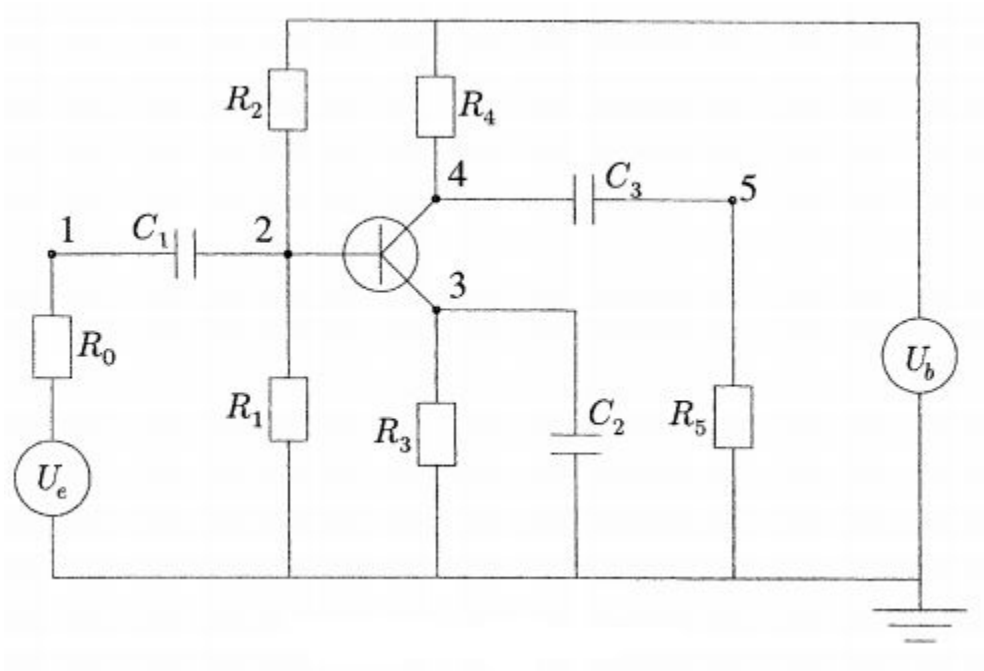
The stiff solvers all perform well, but `ode23s` completes the integration with the fewest steps and runs the fastest for this particular problem. Since the constant Jacobian is specified, none of the solvers need to calculate partial derivatives to compute the solution. Specifying the Jacobian benefits `ode23s` the most since it normally evaluates the Jacobian in every step.

For general stiff problems, the performance of the stiff solvers varies depending on the format of the problem and specified options. Providing the Jacobian matrix or sparsity pattern always improves solver efficiency for stiff problems. But since the stiff solvers use the Jacobian differently, the improvement can vary significantly. Practically speaking, if a system of equations is very large or needs to be solved many times, then it is worthwhile to investigate the performance of the different solvers to minimize execution time.

Solve Stiff Differential Algebraic Equation

This example shows how to use `ode23t` to solve a stiff differential algebraic equation (DAE) problem that arises from an electrical circuit. This problem is originally from p.377 of E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, 2nd ed. (Berlin: Springer, 1996).

The one-transistor amplifier problem coded by amp1dae.m can be rewritten in semi-explicit form, but this example solves it in its original form $Mu' = \phi(u)$. The problem includes a constant, singular mass matrix that is not diagonal. The transistor amplifier circuit contains six resistors, three capacitors, and a transistor.



The initial voltage signal is $U_e(t) = 0.4 \sin(200\pi t)$, and the other parameters are constant. The goal is to solve for the output voltage through node 5.

Set the values of the initial and operating voltages.

```
Ue = @(t) 0.4*sin(200*pi*t);
Ub = 6;
```

Using Kirchoff's law to equalize the current through each node (1 through 5), you obtain a system of five equations. The mass matrix of this system has the form

$$M = \begin{pmatrix} -c_1 & c_1 & 0 & 0 & 0 \\ c_1 & -c_1 & 0 & 0 & 0 \\ 0 & 0 & -c_2 & 0 & 0 \\ 0 & 0 & 0 & -c_3 & c_3 \\ 0 & 0 & 0 & c_3 & -c_3 \end{pmatrix},$$

where $c_k = k \times 10^{-6}$ for $k = 1, 2, 3$.

Use the `odeset` function to pass in the mass matrix. Leave the `'MassSingular'` option at its default value of `'maybe'` to test the automatic detection of a DAE problem by the solver.

```
c = 1e-6 * (1:3);
M = zeros(5,5);
M(1,1) = -c(1);
M(1,2) =  c(1);
M(2,1) =  c(1);
M(2,2) = -c(1);
M(3,3) = -c(2);
M(4,4) = -c(3);
M(4,5) =  c(3);
M(5,4) =  c(3);
M(5,5) = -c(3);
opts = odeset('Mass',M);
```

The function `transampdae.m` contains the system of equations for this example. Save this function in your current folder to run the example.

```
function dudt = transampdae(t,u)
% Define voltages and constant parameters
Ue = @(t) 0.4*sin(200*pi*t);
Ub = 6;
R0 = 1000;
R15 = 9000;
alpha = 0.99;
beta = 1e-6;
```

```
Uf = 0.026;

f23 = beta*(exp((u(2) - u(3))/Uf) - 1);
dudt = [ -(Ue(t) - u(1))/R0
    -(Ub/R15 - u(2)*2/R15 - (1-alpha)*f23)
    -(f23 - u(3)/R15)
    -((Ub - u(4))/R15 - alpha*f23)
    (u(5)/R15) ];
end
```

Set the initial conditions. This example uses the consistent initial conditions computed by Hairer and Wanner.

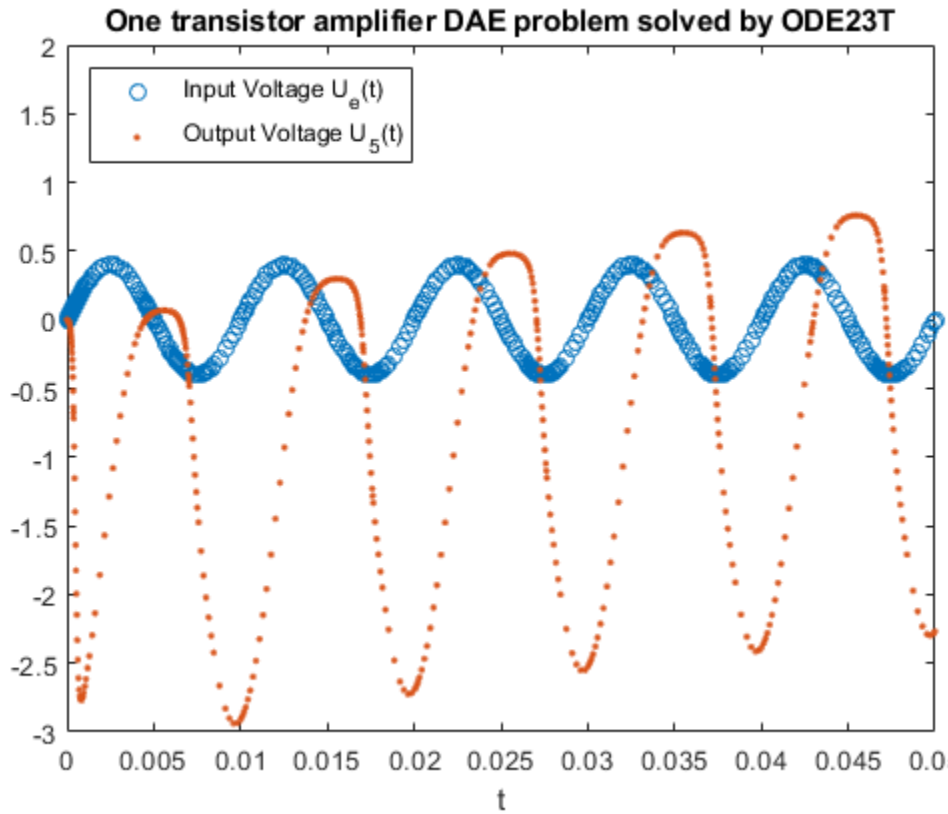
```
u0(1) = 0;
u0(2) = Ub/2;
u0(3) = Ub/2;
u0(4) = Ub;
u0(5) = 0;
```

Solve the DAE system over the time interval `[0 0.05]` using `ode23t`.

```
tspan = [0 0.05];
[t,u] = ode23t(@transampdae,tspan,u0,opts);
```

Plot the initial voltage $U_e(t)$ and output voltage $U_5(t)$.

```
plot(t,Ue(t),'o',t,u(:,5),'.')
axis([0 0.05 -3 2]);
legend('Input Voltage U_e(t)','Output Voltage U_5(t)', 'Location', 'NorthWest');
title('One transistor amplifier DAE problem solved by ODE23T');
xlabel('t');
```



Input Arguments

odefun — Functions to solve
function handle

Functions to solve, specified as a function handle which defines the functions to be integrated.

The function `dydt = odefun(t,y)`, for a scalar `t` and a column vector `y`, must return a column vector `dydt` of data type `single` or `double` that corresponds to $f(t, y)$. `odefun` must accept both input arguments, `t` and `y`, even if one of the arguments is not used in the function.

For example, to solve $y' = 5y - 3$, use the function:

```
function dydt = odefun(t,y)
dydt = 5*y-3;
```

For a system of equations, the output of `odefun` is a vector. Each element in the vector is the solution to one equation. For example, to solve

$$y'_1 = y_1 + 2y_2$$
$$y'_2 = 3y_1 + 2y_2$$

use the function:

```
function dydt = odefun(t,y)
dydt = zeros(2,1);
dydt(1) = y(1)+2*y(2);
dydt(2) = 3*y(1)+2*y(2);
```

For information on how to provide additional parameters to the function `odefun`, see "Parameterizing Functions".

Example: `@myFcn`

Data Types: `function_handle`

`tspan` — Interval of integration
vector

Interval of integration, specified as a vector. At minimum, `tspan` must be a two element vector `[t0 tf]` specifying the initial and final times. To obtain solutions at specific times between `t0` and `tf`, use a longer vector of the form `[t0,t1,t2,...,tf]`. The elements in `tspan` must be all increasing or all decreasing.

The solver imposes the initial conditions given by `y0` at the initial time `tspan(1)`, then integrates from `tspan(1)` to `tspan(end)`:

- If `tspan` has two elements, `[t0 tf]`, then the solver returns the solution evaluated at each internal integration step within the interval.
- If `tspan` has more than two elements `[t0,t1,t2,...,tf]`, then the solver returns the solution evaluated at the given points. However, the solver does not step precisely to each point specified in `tspan`. Instead, the solver uses its own internal steps to compute the solution, then evaluates the solution at the requested points in `tspan`.

The solutions produced at the specified points are of the same order of accuracy as the solutions computed at each internal step.

Specifying several intermediate points has little effect on the efficiency of computation, but for large systems it can affect memory management.

The values of `tspan` are used by the solver to calculate suitable values for `InitialStep` and `MaxStep`:

- If `tspan` contains several intermediate points `[t0,t1,t2,...,tf]`, then the specified points give an indication of the scale for the problem, which can affect the value of `InitialStep` used by the solver. Therefore, the solution obtained by the solver might be different depending on whether you specify `tspan` as a two-element vector or as a vector with intermediate points.
- The initial and final values in `tspan` are used to calculate the maximum step size `MaxStep`. Therefore, changing the initial or final values in `tspan` could lead to the solver using a different step sequence, which might change the solution.

Example: `[1 10]`

Example: `[1 3 5 7 9 10]`

Data Types: `single` | `double`

`y0` — Initial conditions
vector

Initial conditions, specified as a vector. `y0` must be the same length as the vector output of `odefun`, so that `y0` contains an initial condition for each equation defined in `odefun`.

Data Types: `single` | `double`

`options` — Option structure
structure array

Option structure, specified as a structure array. Use the `odeset` function to create or modify the options structure. See “Summary of ODE Options” for a list of the options compatible with each solver.

Example: `options = odeset('RelTol',1e-5,'Stats','on','OutputFcn',@odeplot)` specifies a relative error tolerance of `1e-5`, turns on the display of solver statistics, and specifies the output function `@odeplot` to plot the solution as it is computed.

Data Types: `struct`

Output Arguments

`t` — Evaluation points
column vector

Evaluation points, returned as a column vector.

- If `tspan` contains two elements, `[t0 tf]`, then `t` contains the internal evaluation points used to perform the integration.
- If `tspan` contains more than two elements, then `t` is the same as `tspan`.

`y` — Solutions
array

Solutions, returned as an array. Each row in `y` corresponds to the solution at the value returned in the corresponding row of `t`.

`te` — Time of events
column vector

Time of events, returned as a column vector. The event times in `te` correspond to the solutions returned in `ye`, and `ie` specifies which event occurred.

`ye` — Solution at time of events
array

Solution at time of events, returned as an array. The event times in `te` correspond to the solutions returned in `ye`, and `ie` specifies which event occurred.

`ie` — Index of triggered event function
column vector

Index of triggered event function, returned as a column vector. The event times in `te` correspond to the solutions returned in `ye`, and `ie` specifies which event occurred.

`sol` — Structure for evaluation
structure array

Structure for evaluation, returned as a structure array. Use this structure with the `deval` function to evaluate the solution at any point in the interval `[t0 tf]`. The `sol` structure array always includes these fields:

Structure Field	Description
`sol.x`	Row vector of the steps chosen by the solver.
`sol.y`	Solutions. Each column `sol.y(:,i)` contains the solution at time `sol.x(i)`.
`sol.solver`	Solver name.

Additionally, if you specify the `Events` option and events are detected, then `sol` also includes these fields:

Structure Field	Description
`sol.xe`	Points when events occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

Algorithms

`ode23t` is an implementation of the trapezoidal rule using a “free” interpolant. This solver is preferred over `ode15s` if the problem is only moderately stiff and you need a solution without numerical damping. `ode23t` also can solve differential algebraic equations (DAEs) [1], [2].

References

[1] Shampine, L. F., M. W. Reichelt, and J.A. Kierzenka, “Solving Index-1 DAEs in MATLAB and Simulink”, *SIAM Review*, Vol. 41, 1999, pp. 538–552.

[2] Shampine, L. F. and M. E. Hosea, “Analysis and Implementation of TR-BDF2,” *Applied Numerical Mathematics 20*, 1996.

See Also

`deval` | `ode15s` | `odeget` | `odeset`

Topics

“Choose an ODE Solver”
“Summary of ODE Options”
“Solve Stiff ODEs”
“Solve Differential Algebraic Equations (DAEs)”
“Troubleshoot Common ODE Problems”

Introduced before R2006a

ode23tb

Solve stiff differential equations — trapezoidal rule + backward differentiation formula

Syntax

```
[t,y] = ode23tb(odefun,tspan,y0)
[t,y] = ode23tb(odefun,tspan,y0,options)
[t,y,te,ye,ie] = ode23tb(odefun,tspan,y0,options)
sol = ode23tb( ___ )
```

Description

`[t,y] = ode23tb(odefun,tspan,y0)`, where `tspan = [t0 tf]`, integrates the system of differential equations $y' = f(t, y)$ from `t0` to `tf` with initial conditions `y0`. Each row in the solution array `y` corresponds to a value returned in column vector `t`.

All MATLAB ODE solvers can solve systems of equations of the form $y' = f(t, y)$, or problems that involve a mass matrix, $M(t, y)y' = f(t, y)$. The solvers all use similar syntaxes. The `ode23s` solver only can solve problems with a mass matrix if the mass matrix is constant. `ode15s` and `ode23t` can solve problems with a mass matrix that is singular, known as differential-algebraic equations (DAEs). Specify the mass matrix using the `Mass` option of `odeset`.

`[t,y] = ode23tb(odefun,tspan,y0,options)` also uses the integration settings defined by `options`, which is an argument created using the `odeset` function. For example, use the `AbsTol` and `RelTol` options to specify absolute and relative error tolerances, or the `Mass` option to provide a mass matrix.

`[t,y,te,ye,ie] = ode23tb(odefun,tspan,y0,options)` additionally finds where functions of (t,y), called event functions, are zero. In the output, `te` is the time of the event, `ye` is the solution at the time of the event, and `ie` is the index of the triggered event.

For each event function, specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the `'Events'` property to a function, such as `myEventFcn` or `@myEventFcn`, and creating a

corresponding function: [value,isterminal,direction] = myEventFcn(t,y). For more information, see “ODE Event Location”.

sol = ode23tb(___) returns a structure that you can use with deval to evaluate the solution at any point on the interval [t0 tf]. You can use any of the input argument combinations in previous syntaxes.

Examples

ODE With Single Solution Component

Simple ODEs that have a single solution component can be specified as an anonymous function in the call to the solver. The anonymous function must accept two inputs (t,y) even if one of the inputs is not used.

Solve the ODE

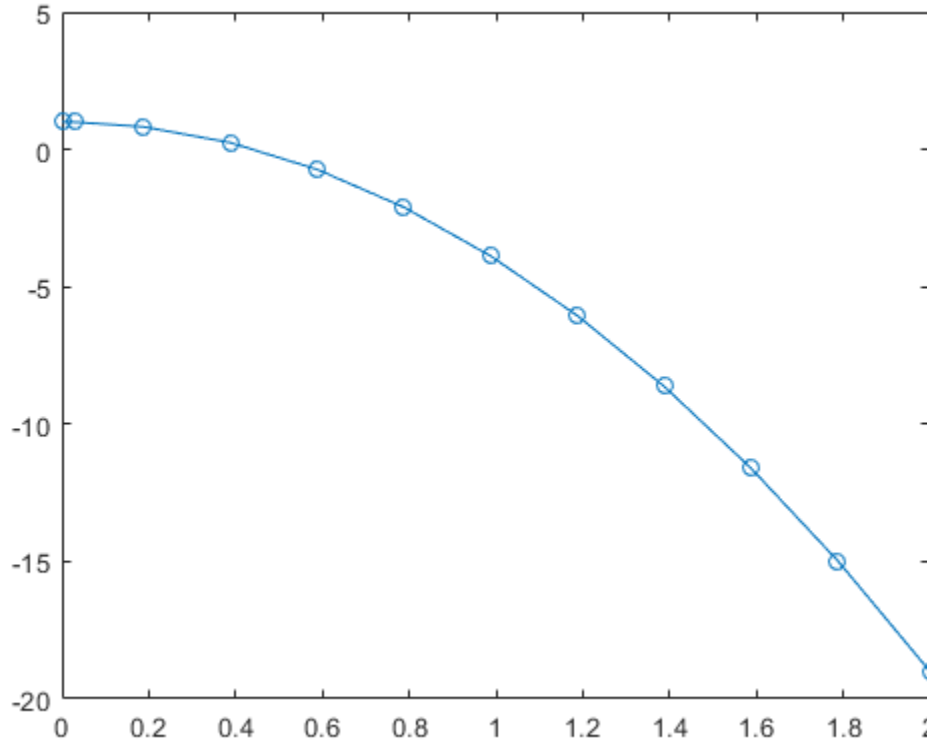
$$y' = -10t.$$

Use a time interval of [0,2] and the initial condition y0 = 1.

```
tspan = [0 2];
y0 = 1;
[t,y] = ode23tb(@(t,y) -10*t, tspan, y0);
```

Plot the solution.

```
plot(t,y,'-o')
```



Solve Stiff ODE

An example of a stiff system of equations is the van der Pol equations in relaxation oscillation. The limit cycle has regions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

The system of equations is:

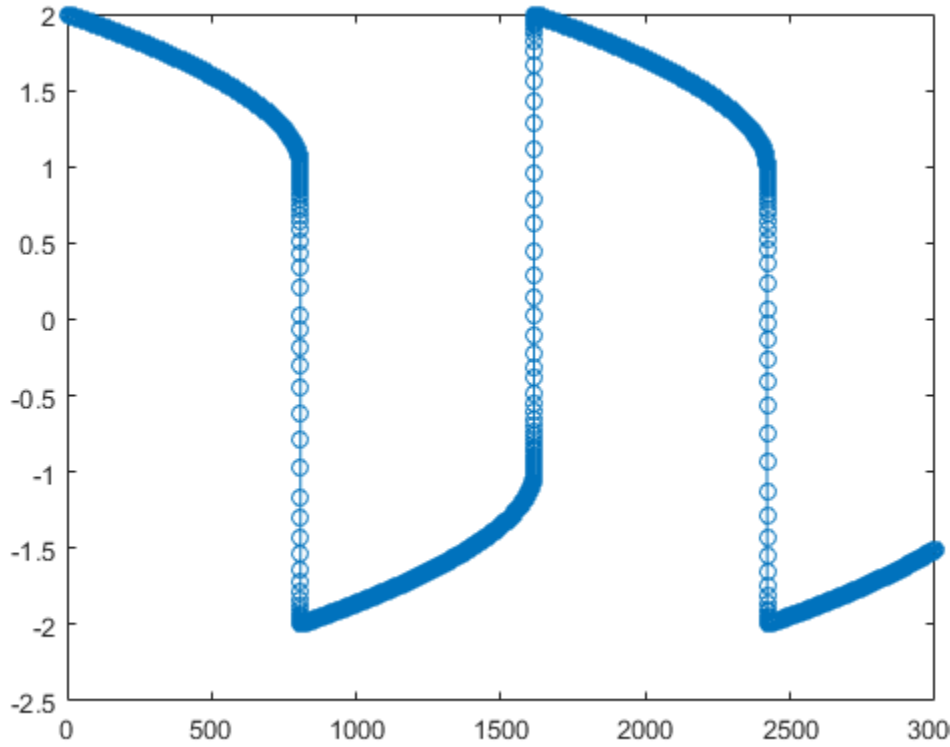
$$\begin{aligned} y_1' &= y_2 \\ y_2' &= 1000(1 - y_1^2)y_2 - y_1 \end{aligned}$$

The initial conditions are $y_1(0) = 2$ and $y_2(0) = 0$. The function `vdp1000` ships with MATLAB® and encodes the equations.

```
function dydt = vdp1000(t,y)
%VDP1000  Evaluate the van der Pol ODEs for mu = 1000.
%
%   See also ODE15S, ODE23S, ODE23T, ODE23TB.

%   Jacek Kierzenka and Lawrence F. Shampine
%   Copyright 1984-2014 The MathWorks, Inc.

dydt = [y(2); 1000*(1-y(1)^2)*y(2)-y(1)];
```

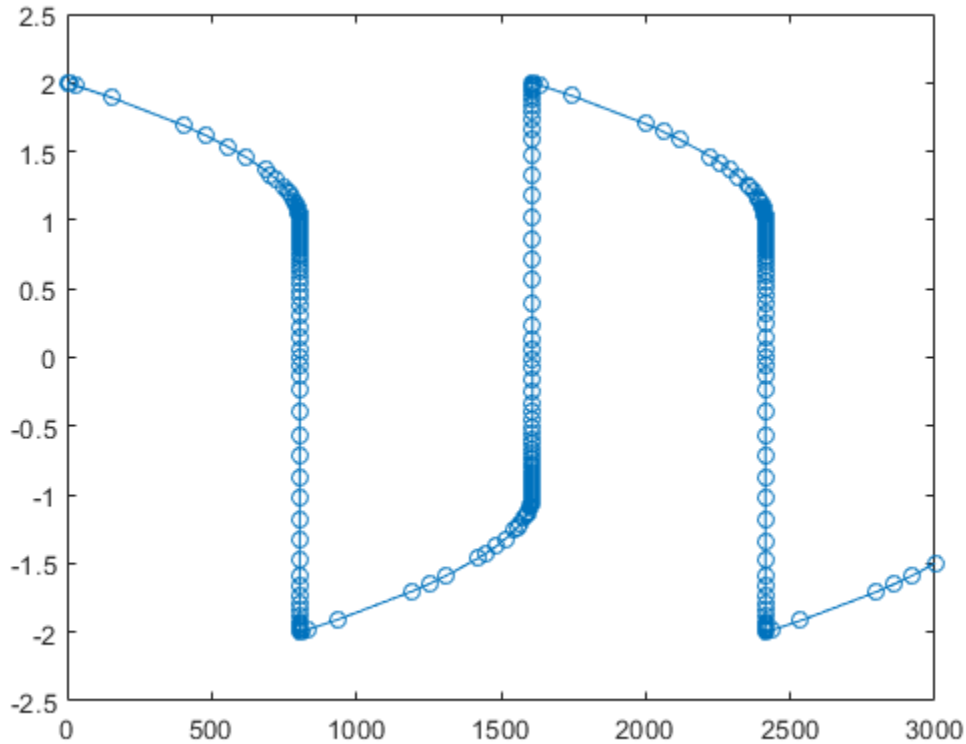
Solving this system using `ode45` with the default relative and absolute error tolerances (`1e-3` and `1e-6`, respectively) is extremely slow, requiring several minutes to solve and plot the solution. `ode45` requires millions of time steps to complete the integration, due to the areas of stiffness where it struggles to meet the tolerances.

This is a plot of the solution obtained by `ode45`, which takes a long time to compute. Notice the enormous number of time steps required to pass through areas of stiffness.



Solve the stiff system using the `ode23tb` solver, and then plot the first column of the solution `y` against the time points `t`. The `ode23tb` solver passes through stiff areas with far fewer steps than `ode45`.

```
[t,y] = ode23tb(@vdp1000,[0 3000],[2 0]);
plot(t,y(:,1),'-o')
```



Pass Extra Parameters to ODE Function

`ode23s` only works with functions that use two input arguments, `t` and `y`. However, you can pass in extra parameters by defining them outside the function and passing them in when you specify the function handle.

Solve the ODE

$$y'' = \frac{A}{B} t y.$$

Rewriting the equation as a first-order system yields

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= \frac{A}{B} t y_1. \end{aligned}$$

`odefcn.m` represents this system of equations as a function that accepts four input arguments: `t`, `y`, `A`, and `B`.

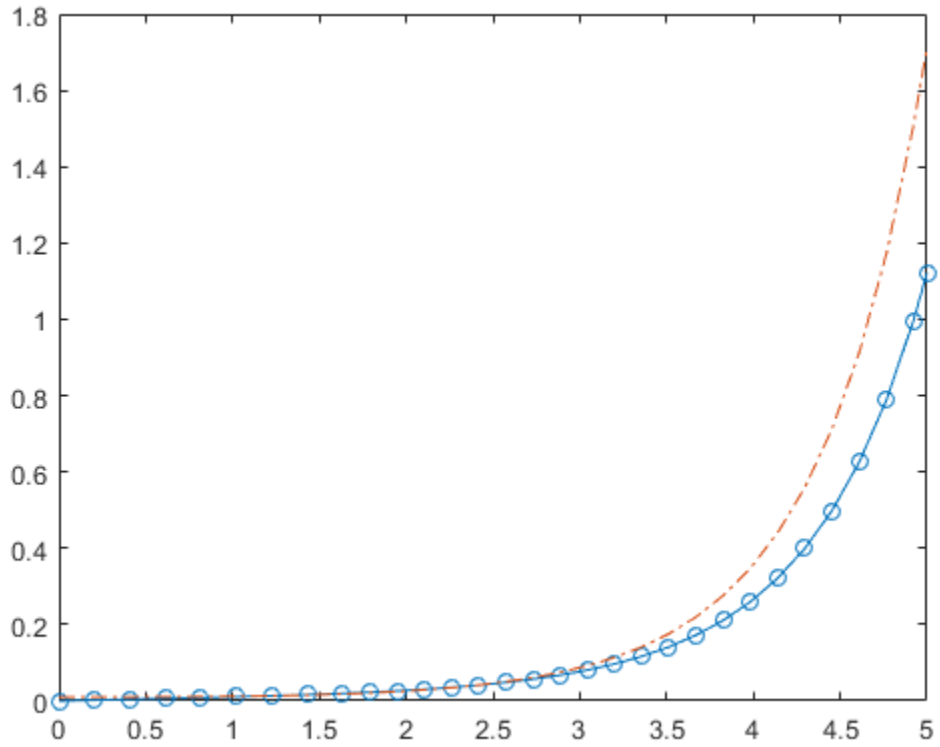
```
function dydt = odefcn(t,y,A,B)
dydt = zeros(2,1);
dydt(1) = y(2);
dydt(2) = (A/B)*t.*y(1);
```

Solve the ODE using `ode23tb`. Specify the function handle such that it passes in the predefined values for `A` and `B` to `odefcn`.

```
A = 1;
B = 2;
tspan = [0 5];
y0 = [0 0.01];
[t,y] = ode23tb(@(t,y) odefcn(t,y,A,B), tspan, y0);
```

Plot the results.

```
plot(t,y(:,1),'-o',t,y(:,2),'-.')
```



Compare Stiff ODE Solvers

The `ode15s` solver is a good first choice for most stiff problems. However, the other stiff solvers might be more efficient for certain types of problems. This example solves a stiff test equation using all four stiff ODE solvers.

Consider the test equation

$$y' = -\lambda y.$$

The equation becomes increasingly stiff as the magnitude of λ increases. Use $\lambda = 1 \times 10^9$ and the initial condition $y(0) = 1$ over the time interval `[0 0.5]`. With these values, the

problem is stiff enough that `ode45` and `ode23` struggle to integrate the equation. Also, use `odeset` to pass in the constant Jacobian $J = \frac{\partial f}{\partial y} = -\lambda$ and turn on the display of solver statistics.

```
lambda = 1e9;
y0 = 1;
tspan = [0 0.5];
opts = odeset('Jacobian',-lambda,'Stats','on');
```

Solve the equation with `ode15s`, `ode23s`, `ode23t`, and `ode23tb`. Make subplots for comparison.

```
subplot(2,2,1)
tic, ode15s(@(t,y) -lambda*y, tspan, y0, opts), toc
```

```
104 successful steps
1 failed attempts
212 function evaluations
0 partial derivatives
21 LU decompositions
210 solutions of linear systems
Elapsed time is 1.800760 seconds.
```

```
title('ode15s')
subplot(2,2,2)
tic, ode23s(@(t,y) -lambda*y, tspan, y0, opts), toc
```

```
63 successful steps
0 failed attempts
191 function evaluations
0 partial derivatives
63 LU decompositions
189 solutions of linear systems
Elapsed time is 0.472553 seconds.
```

```
title('ode23s')
subplot(2,2,3)
tic, ode23t(@(t,y) -lambda*y, tspan, y0, opts), toc
```

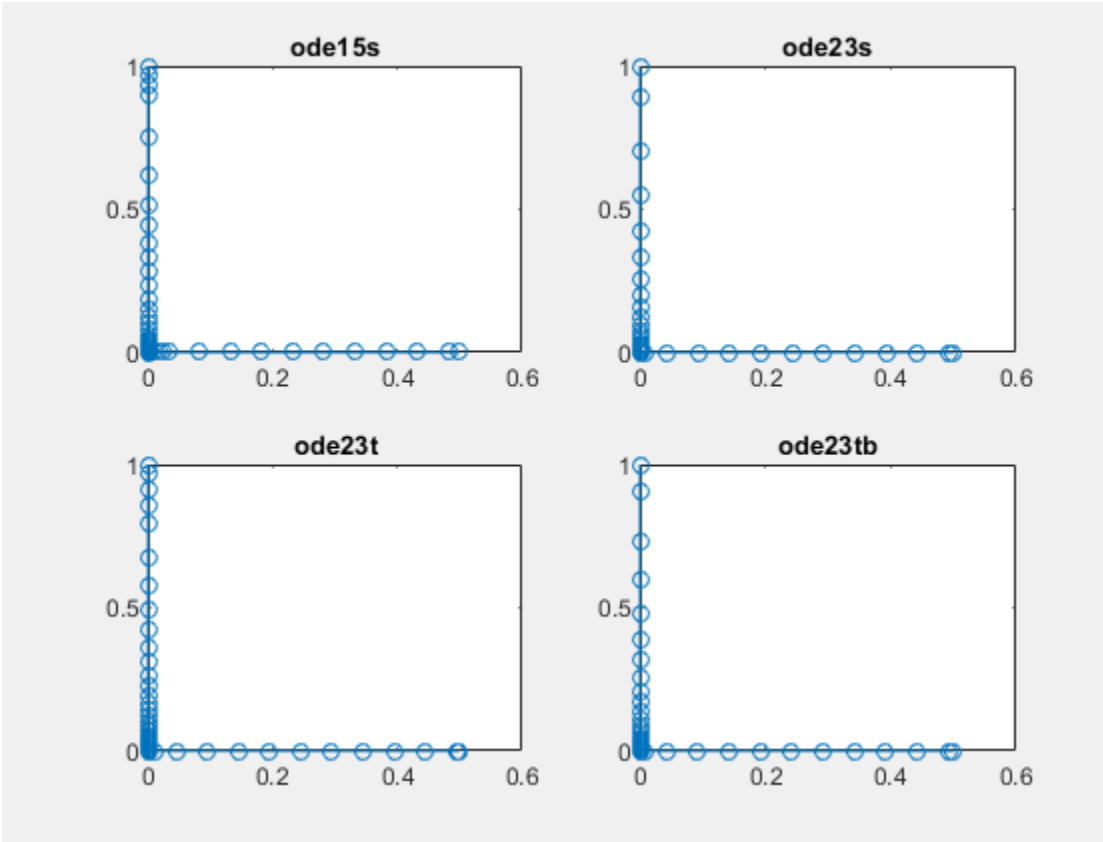
```
95 successful steps
0 failed attempts
125 function evaluations
0 partial derivatives
28 LU decompositions
```

```
123 solutions of linear systems
Elapsed time is 0.689998 seconds.
```

```
title('ode23t')
subplot(2,2,4)
tic, ode23tb(@(t,y) -lambda*y, tspan, y0, opts), toc
```

```
71 successful steps
0 failed attempts
167 function evaluations
0 partial derivatives
23 LU decompositions
236 solutions of linear systems
Elapsed time is 0.553235 seconds.
```

```
title('ode23tb')
```



The stiff solvers all perform well, but `ode23s` completes the integration with the fewest steps and runs the fastest for this particular problem. Since the constant Jacobian is specified, none of the solvers need to calculate partial derivatives to compute the solution. Specifying the Jacobian benefits `ode23s` the most since it normally evaluates the Jacobian in every step.

For general stiff problems, the performance of the stiff solvers varies depending on the format of the problem and specified options. Providing the Jacobian matrix or sparsity pattern always improves solver efficiency for stiff problems. But since the stiff solvers use the Jacobian differently, the improvement can vary significantly. Practically speaking, if a system of equations is very large or needs to be solved many times, then it is worthwhile to investigate the performance of the different solvers to minimize execution time.

Input Arguments

odefun — Functions to solve
function handle

Functions to solve, specified as a function handle which defines the functions to be integrated.

The function `dydt = odefun(t,y)`, for a scalar `t` and a column vector `y`, must return a column vector `dydt` of data type `single` or `double` that corresponds to $f(t, y)$. `odefun` must accept both input arguments, `t` and `y`, even if one of the arguments is not used in the function.

For example, to solve $y' = 5y - 3$, use the function:

```
function dydt = odefun(t,y)
dydt = 5*y-3;
```

For a system of equations, the output of `odefun` is a vector. Each element in the vector is the solution to one equation. For example, to solve

$$y'_1 = y_1 + 2y_2$$
$$y'_2 = 3y_1 + 2y_2$$

use the function:

```
function dydt = odefun(t,y)
dydt = zeros(2,1);
dydt(1) = y(1)+2*y(2);
dydt(2) = 3*y(1)+2*y(2);
```

For information on how to provide additional parameters to the function `odefun`, see “Parameterizing Functions”.

Example: `@myFcn`

Data Types: `function_handle`

tspan — Interval of integration
vector

Interval of integration, specified as a vector. At minimum, `tspan` must be a two element vector `[t0 tf]` specifying the initial and final times. To obtain solutions at specific times

between `t0` and `tf`, use a longer vector of the form `[t0,t1,t2,...,tf]`. The elements in `tspan` must be all increasing or all decreasing.

The solver imposes the initial conditions given by `y0` at the initial time `tspan(1)`, then integrates from `tspan(1)` to `tspan(end)`:

- If `tspan` has two elements, `[t0 tf]`, then the solver returns the solution evaluated at each internal integration step within the interval.
- If `tspan` has more than two elements `[t0,t1,t2,...,tf]`, then the solver returns the solution evaluated at the given points. However, the solver does not step precisely to each point specified in `tspan`. Instead, the solver uses its own internal steps to compute the solution, then evaluates the solution at the requested points in `tspan`. The solutions produced at the specified points are of the same order of accuracy as the solutions computed at each internal step.

  Specifying several intermediate points has little effect on the efficiency of computation, but for large systems it can affect memory management.

The values of `tspan` are used by the solver to calculate suitable values for `InitialStep` and `MaxStep`:

- If `tspan` contains several intermediate points `[t0,t1,t2,...,tf]`, then the specified points give an indication of the scale for the problem, which can affect the value of `InitialStep` used by the solver. Therefore, the solution obtained by the solver might be different depending on whether you specify `tspan` as a two-element vector or as a vector with intermediate points.
- The initial and final values in `tspan` are used to calculate the maximum step size `MaxStep`. Therefore, changing the initial or final values in `tspan` could lead to the solver using a different step sequence, which might change the solution.

Example: `[1 10]`

Example: `[1 3 5 7 9 10]`

Data Types: `single` | `double`

`y0` — Initial conditions
vector

Initial conditions, specified as a vector. `y0` must be the same length as the vector output of `odefun`, so that `y0` contains an initial condition for each equation defined in `odefun`.

Data Types: `single` | `double`

options — Option structure
structure array

Option structure, specified as a structure array. Use the `odeset` function to create or modify the options structure. See “Summary of ODE Options” for a list of the options compatible with each solver.

Example: `options = odeset('RelTol',1e-5,'Stats','on','OutputFcn',@odeplot)` specifies a relative error tolerance of `1e-5`, turns on the display of solver statistics, and specifies the output function `@odeplot` to plot the solution as it is computed.

Data Types: `struct`

Output Arguments

t — Evaluation points
column vector

Evaluation points, returned as a column vector.

- If `tspan` contains two elements, `[t0 tf]`, then `t` contains the internal evaluation points used to perform the integration.
- If `tspan` contains more than two elements, then `t` is the same as `tspan`.

y — Solutions
array

Solutions, returned as an array. Each row in `y` corresponds to the solution at the value returned in the corresponding row of `t`.

te — Time of events
column vector

Time of events, returned as a column vector. The event times in `te` correspond to the solutions returned in `ye`, and `ie` specifies which event occurred.

ye — Solution at time of events
array

Solution at time of events, returned as an array. The event times in `te` correspond to the solutions returned in `ye`, and `ie` specifies which event occurred.

`ie` — Index of triggered event function

column vector

Index of triggered event function, returned as a column vector. The event times in `te` correspond to the solutions returned in `ye`, and `ie` specifies which event occurred.

`sol` — Structure for evaluation

structure array

Structure for evaluation, returned as a structure array. Use this structure with the `deval` function to evaluate the solution at any point in the interval `[t0 tf]`. The `sol` structure array always includes these fields:

Structure Field	Description
`sol.x`	Row vector of the steps chosen by the solver.
`sol.y`	Solutions. Each column `sol.y(:,i)` contains the solution at time `sol.x(i)`.
`sol.solver`	Solver name.

Additionally, if you specify the `Events` option and events are detected, then `sol` also includes these fields:

Structure Field	Description
`sol.xe`	Points when events occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

Algorithms

`ode23tb` is an implementation of TR-BDF2, an implicit Runge-Kutta formula with a trapezoidal rule step as its first stage and a backward differentiation formula of order two

as its second stage. By construction, the same iteration matrix is used in evaluating both stages. Like `ode23s` and `ode23t`, this solver may be more efficient than `ode15s` for problems with crude tolerances [1], [2].

References

[1] Bank, R. E., W. C. Coughran, Jr., W. Fichtner, E. Grosse, D. Rose, and R. Smith, "Transient Simulation of Silicon Devices and Circuits," *IEEE Trans. CAD*, 4 (1985), pp. 436–451.

[2] Shampine, L. F. and M. E. Hosea, "Analysis and Implementation of TR-BDF2," *Applied Numerical Mathematics 20*, 1996.

See Also

`deval` | `ode15s` | `odeget` | `odeset`

Topics

"Choose an ODE Solver"
"Summary of ODE Options"
"Solve Stiff ODEs"
"Troubleshoot Common ODE Problems"

Introduced before R2006a

ode45

Solve nonstiff differential equations — medium order method

Syntax

```
[t,y] = ode45(odefun,tspan,y0)
[t,y] = ode45(odefun,tspan,y0,options)
[t,y,te,ye,ie] = ode45(odefun,tspan,y0,options)
sol = ode45( ___ )
```

Description

`[t,y] = ode45(odefun,tspan,y0)`, where `tspan = [t0 tf]`, integrates the system of differential equations $y' = f(t, y)$ from `t0` to `tf` with initial conditions `y0`. Each row in the solution array `y` corresponds to a value returned in column vector `t`.

All MATLAB ODE solvers can solve systems of equations of the form $y' = f(t, y)$, or problems that involve a mass matrix, $M(t, y)y' = f(t, y)$. The solvers all use similar syntaxes. The `ode23s` solver only can solve problems with a mass matrix if the mass matrix is constant. `ode15s` and `ode23t` can solve problems with a mass matrix that is singular, known as differential-algebraic equations (DAEs). Specify the mass matrix using the `Mass` option of `odeset`.

`ode45` is a versatile ODE solver and is the first solver you should try for most problems. However, if the problem is stiff or requires high accuracy, then there are other ODE solvers that might be better suited to the problem. See “Choose an ODE Solver” for more information.

`[t,y] = ode45(odefun,tspan,y0,options)` also uses the integration settings defined by `options`, which is an argument created using the `odeset` function. For example, use the `AbsTol` and `RelTol` options to specify absolute and relative error tolerances, or the `Mass` option to provide a mass matrix.

`[t,y,te,ye,ie] = ode45(odefun,tspan,y0,options)` additionally finds where functions of (t,y), called event functions, are zero. In the output, `te` is the time of the

event, `ye` is the solution at the time of the event, and `ie` is the index of the triggered event.

For each event function, specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the `'Events'` property to a function, such as `myEventFcn` or `@myEventFcn`, and creating a corresponding function: `[value,isterminal,direction] = myEventFcn(t,y)`. For more information, see “ODE Event Location”.

`sol = ode45( ___ )` returns a structure that you can use with `deval` to evaluate the solution at any point on the interval `[t0 tf]`. You can use any of the input argument combinations in previous syntaxes.

Examples

ODE with Single Solution Component

Simple ODEs that have a single solution component can be specified as an anonymous function in the call to the solver. The anonymous function must accept two inputs `(t,y)` even if one of the inputs is not used.

Solve the ODE

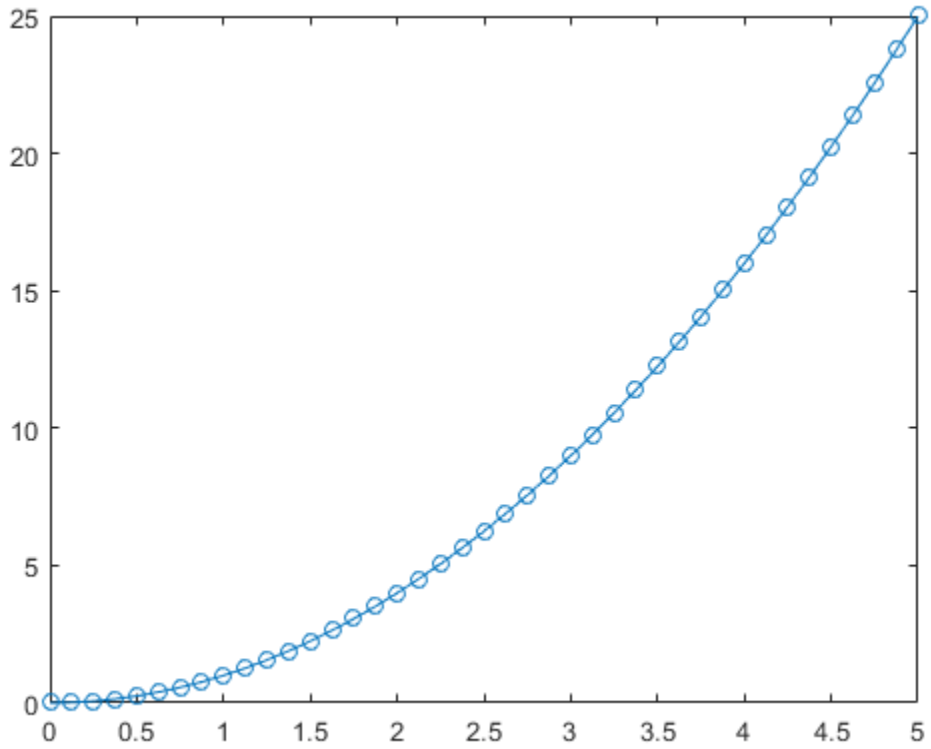
$$y' = 2t.$$

Use a time interval of `[0,5]` and the initial condition `y0 = 0`.

```
tspan = [0 5];
y0 = 0;
[t,y] = ode45(@(t,y) 2*t, tspan, y0);
```

Plot the solution.

```
plot(t,y,'-o')
```



Solve Nonstiff Equation

The van der Pol equation is a second order ODE

$$y_1'' - \mu\left(1 - y_1^2\right) y_1' + y_1 = 0,$$

where $\mu > 0$ is a scalar parameter. Rewrite this equation as a system of first-order ODEs by making the substitution $y_1' = y_2$. The resulting system of first-order ODEs is

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= \mu(1 - y_1^2)y_2 - y_1. \end{aligned}$$

The function file `vdp1.m` represents the van der Pol equation using $\mu = 1$. The variables y_1 and y_2 are the entries `y(1)` and `y(2)` of a two-element vector, `dydt`.

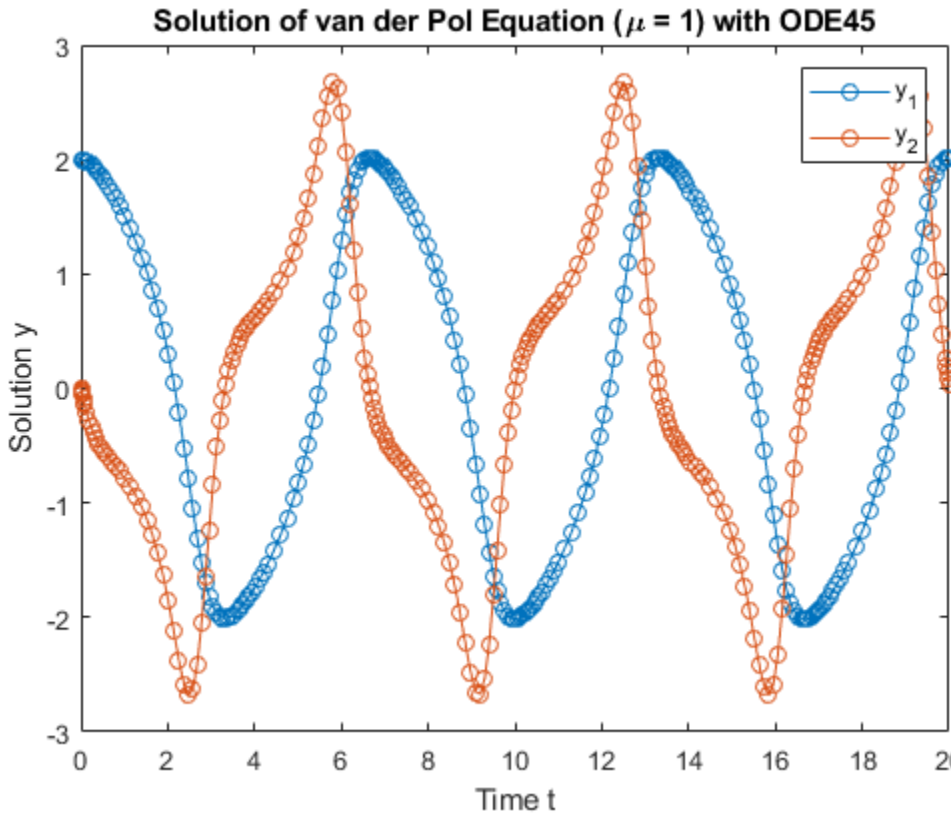
```
function dydt = vdp1(t,y)
%VDP1  Evaluate the van der Pol ODEs for mu = 1
%
%   See also ODE113, ODE23, ODE45.

%   Jacek Kierzenka and Lawrence F. Shampine
%   Copyright 1984-2014 The MathWorks, Inc.

dydt = [y(2); (1-y(1)^2)*y(2)-y(1)];
```

Solve the ODE using the `ode45` function on the time interval `[0 20]` with initial values `[2 0]`. The resulting output is a column vector of time points `t` and a solution array `y`. Each row in `y` corresponds to a time returned in the corresponding row of `t`. The first column of `y` corresponds to y_1, and the second column to y_2.

```
[t,y] = ode45(@vdp1,[0 20],[2; 0]);
```

Plot the solutions for y_1 and y_2 against `t`.

```
plot(t,y(:,1),'-o',t,y(:,2),'-o')
title('Solution of van der Pol Equation (\mu = 1) with ODE45');
xlabel('Time t');
ylabel('Solution y');
legend('y_1','y_2')
```



Pass Extra Parameters to ODE Function

`ode45` only works with functions that use two input arguments, `t` and `y`. However, you can pass in extra parameters by defining them outside the function and passing them in when you specify the function handle.

Solve the ODE

$$y'' = \frac{A}{B} t y.$$

Rewriting the equation as a first-order system yields

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= \frac{A}{B} t y_1. \end{aligned}$$

`odefcn.m` represents this system of equations as a function that accepts four input arguments: `t`, `y`, `A`, and `B`.

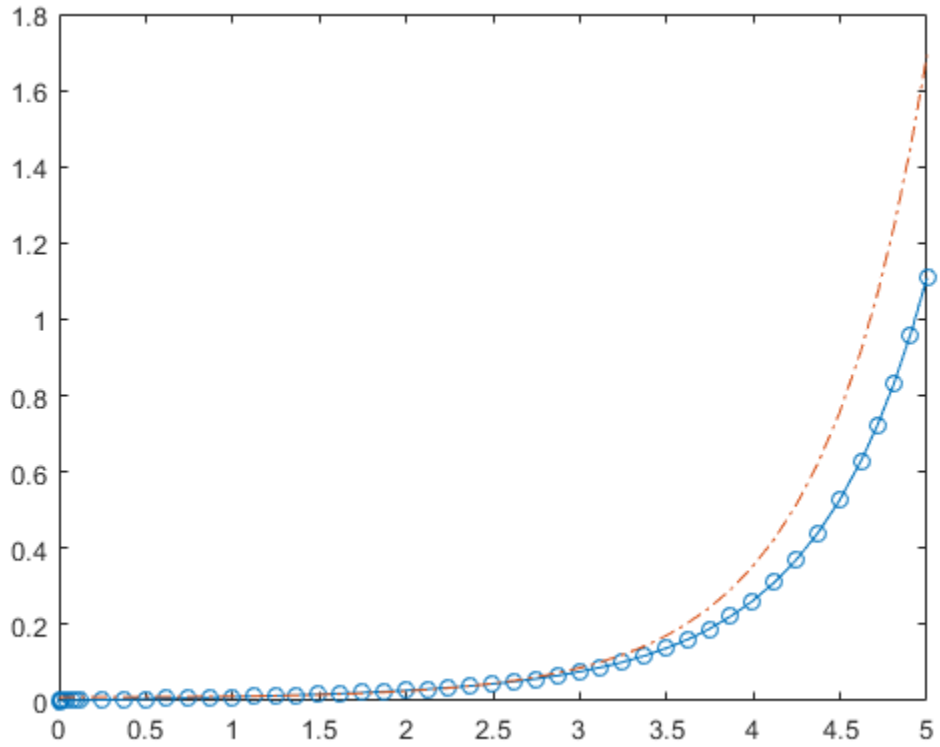
```
function dydt = odefcn(t,y,A,B)
dydt = zeros(2,1);
dydt(1) = y(2);
dydt(2) = (A/B)*t.*y(1);
```

Solve the ODE using `ode45`. Specify the function handle such that it passes in the predefined values for `A` and `B` to `odefcn`.

```
A = 1;
B = 2;
tspan = [0 5];
y0 = [0 0.01];
[t,y] = ode45(@(t,y) odefcn(t,y,A,B), tspan, y0);
```

Plot the results.

```
plot(t,y(:,1),'-o',t,y(:,2),'-.')
```



ODE with Time-Dependent Terms

Consider the following ODE with time-dependent parameters

$$y'(t) + f(t)y(t) = g(t).$$

The initial condition is $y_0 = 1$. The function `f(t)` is defined by the n-by-1 vector `f` evaluated at times `ft`. The function `g(t)` is defined by the m-by-1 vector `g` evaluated at times `gt`.

Create the vectors `f` and `g`.

```
ft = linspace(0,5,25);
f = ft.^2 - ft - 3;

gt = linspace(1,6,25);
g = 3*sin(gt-0.25);
```

Write a function named `myode` that interpolates `f` and `g` to obtain the value of the time-dependent terms at the specified time. Save the function in your current folder to run the rest of the example.

The `myode` function accepts extra input arguments to evaluate the ODE at each time step, but `ode45` only uses the first two input arguments `t` and `y`.

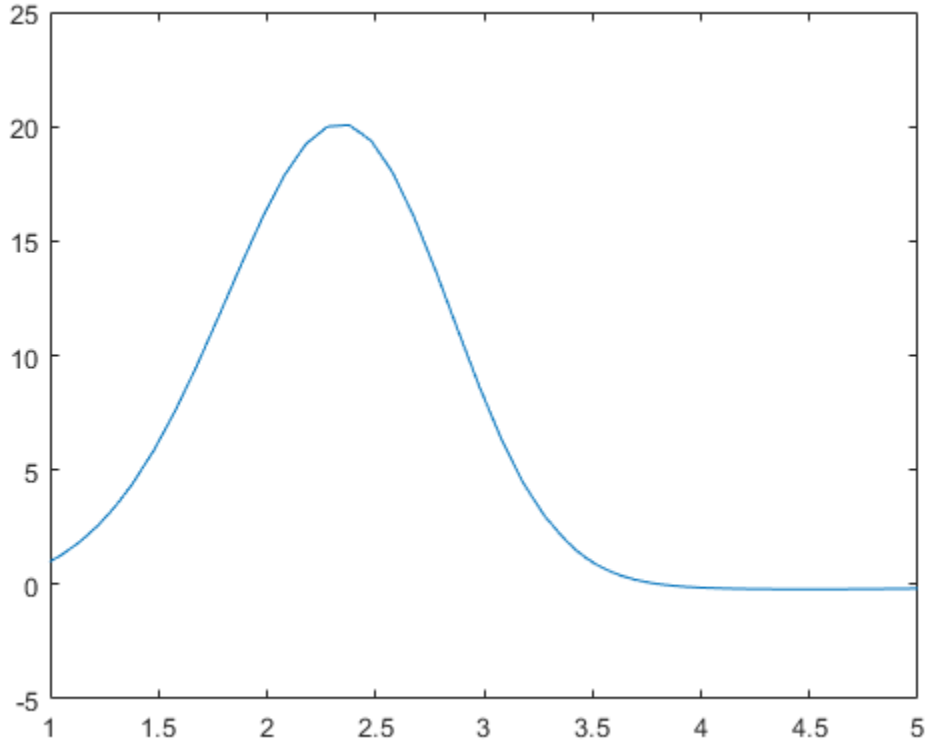
```
function dydt = myode(t,y,ft,f,gt,g)
f = interp1(ft,f,t); % Interpolate the data set (ft,f) at time t
g = interp1(gt,g,t); % Interpolate the data set (gt,g) at time t
dydt = -f.*y + g; % Evaluate ODE at time t
```

Solve the equation over the time interval `[1 5]` using `ode45`. Specify the function using a function handle so that `ode45` uses only the first two input arguments of `myode`. Also, loosen the error thresholds using `odeset`.

```
tspan = [1 5];
ic = 1;
opts = odeset('RelTol',1e-2,'AbsTol',1e-4);
[t,y] = ode45(@(t,y) myode(t,y,ft,f,gt,g), tspan, ic, opts);
```

Plot the solution, `y`, as a function of the time points, `t`.

```
plot(t,y)
```



Evaluate and Extend Solution Structure

The van der Pol equation is a second order ODE

$$y_1'' - \mu\left(1 - y_1^2\right)y_1' + y_1 = 0\,.$$

Solve the van der Pol equation with $\mu = 1$ using `ode45`. The function `vdp1.m` ships with MATLAB® and encodes the equations. Specify a single output to return a structure containing information about the solution, such as the solver and evaluation points.

```
tspan = [0 20];
y0 = [2 0];
sol = ode45(@vdp1,tspan,y0)
```

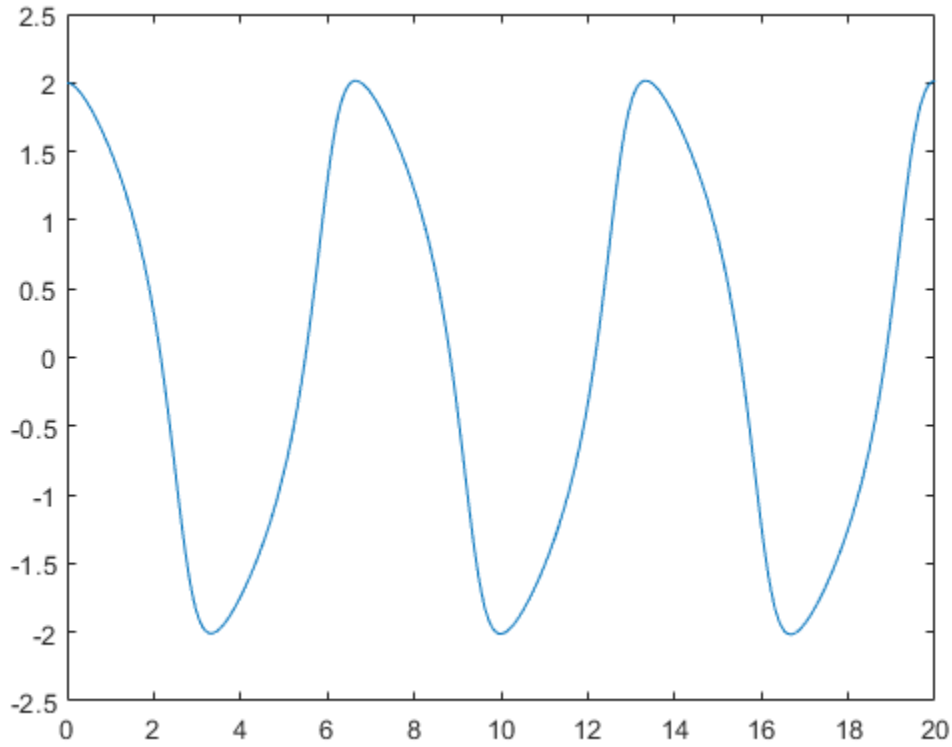
```
sol = struct with fields:
     solver: 'ode45'
    extdata: [1x1 struct]
          x: [1x60 double]
          y: [2x60 double]
      stats: [1x1 struct]
      idata: [1x1 struct]
```

Use `linspace` to generate 250 points in the interval `[0 20]`. Evaluate the solution at these points using `deval`.

```
x = linspace(0,20,250);
y = deval(sol,x);
```

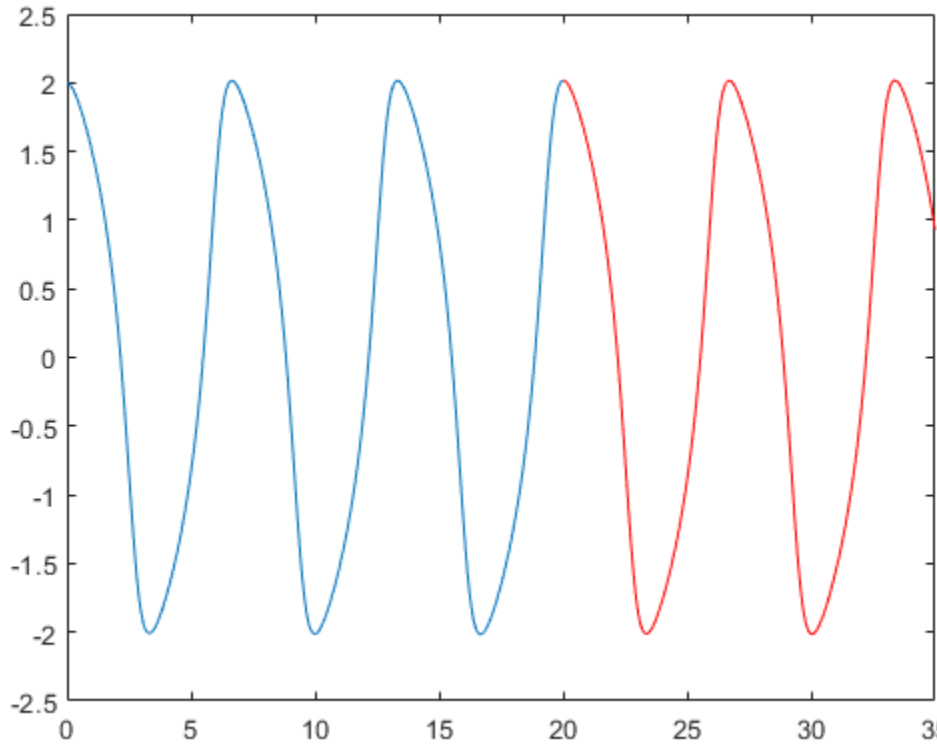
Plot the first component of the solution.

```
plot(x,y(1,:))
```



Extend the solution to $t_f = 35$ using `odextend` and add the result to the original plot.

```
sol_new = odextend(sol,@vdp1,35);
x = linspace(20,35,350);
y = deval(sol_new,x);
hold on
plot(x,y(1,:),'r')
```



Input Arguments

odefun — Functions to solve
function handle

Functions to solve, specified as a function handle which defines the functions to be integrated.

The function `dydt = odefun(t,y)`, for a scalar `t` and a column vector `y`, must return a column vector `dydt` of data type `single` or `double` that corresponds to $f(t, y)$. `odefun`

must accept both input arguments, `t` and `y`, even if one of the arguments is not used in the function.

For example, to solve $y' = 5y - 3$, use the function:

```
function dydt = odefun(t,y)
dydt = 5*y-3;
```

For a system of equations, the output of `odefun` is a vector. Each element in the vector is the solution to one equation. For example, to solve

$$y'_1 = y_1 + 2y_2$$
$$y'_2 = 3y_1 + 2y_2$$

use the function:

```
function dydt = odefun(t,y)
dydt = zeros(2,1);
dydt(1) = y(1)+2*y(2);
dydt(2) = 3*y(1)+2*y(2);
```

For information on how to provide additional parameters to the function `odefun`, see "Parameterizing Functions".

Example: `@myFcn`

Data Types: `function_handle`

`tspan` — Interval of integration
vector

Interval of integration, specified as a vector. At minimum, `tspan` must be a two element vector `[t0 tf]` specifying the initial and final times. To obtain solutions at specific times between `t0` and `tf`, use a longer vector of the form `[t0,t1,t2,...,tf]`. The elements in `tspan` must be all increasing or all decreasing.

The solver imposes the initial conditions given by `y0` at the initial time `tspan(1)`, then integrates from `tspan(1)` to `tspan(end)`:

- If `tspan` has two elements, `[t0 tf]`, then the solver returns the solution evaluated at each internal integration step within the interval.
- If `tspan` has more than two elements `[t0,t1,t2,...,tf]`, then the solver returns the solution evaluated at the given points. However, the solver does not step precisely

to each point specified in `tspan`. Instead, the solver uses its own internal steps to compute the solution, then evaluates the solution at the requested points in `tspan`. The solutions produced at the specified points are of the same order of accuracy as the solutions computed at each internal step.

Specifying several intermediate points has little effect on the efficiency of computation, but for large systems it can affect memory management.

The values of `tspan` are used by the solver to calculate suitable values for `InitialStep` and `MaxStep`:

- If `tspan` contains several intermediate points `[t0,t1,t2,...,tf]`, then the specified points give an indication of the scale for the problem, which can affect the value of `InitialStep` used by the solver. Therefore, the solution obtained by the solver might be different depending on whether you specify `tspan` as a two-element vector or as a vector with intermediate points.
- The initial and final values in `tspan` are used to calculate the maximum step size `MaxStep`. Therefore, changing the initial or final values in `tspan` could lead to the solver using a different step sequence, which might change the solution.

Example: `[1 10]`

Example: `[1 3 5 7 9 10]`

Data Types: `single` | `double`

`y0` — Initial conditions
vector

Initial conditions, specified as a vector. `y0` must be the same length as the vector output of `odefun`, so that `y0` contains an initial condition for each equation defined in `odefun`.

Data Types: `single` | `double`

`options` — Option structure
structure array

Option structure, specified as a structure array. Use the `odeset` function to create or modify the options structure. See “Summary of ODE Options” for a list of the options compatible with each solver.

Example: `options = odeset('RelTol',1e-5,'Stats','on','OutputFcn',@odeplot)` specifies a

relative error tolerance of `1e-5`, turns on the display of solver statistics, and specifies the output function `@odeplot` to plot the solution as it is computed.

Data Types: `struct`

Output Arguments

`t` — Evaluation points
column vector

Evaluation points, returned as a column vector.

- If `tspan` contains two elements, `[t0 tf]`, then `t` contains the internal evaluation points used to perform the integration.
- If `tspan` contains more than two elements, then `t` is the same as `tspan`.

`y` — Solutions
array

Solutions, returned as an array. Each row in `y` corresponds to the solution at the value returned in the corresponding row of `t`.

`te` — Time of events
column vector

Time of events, returned as a column vector. The event times in `te` correspond to the solutions returned in `ye`, and `ie` specifies which event occurred.

`ye` — Solution at time of events
array

Solution at time of events, returned as an array. The event times in `te` correspond to the solutions returned in `ye`, and `ie` specifies which event occurred.

`ie` — Index of triggered event function
column vector

Index of triggered event function, returned as a column vector. The event times in `te` correspond to the solutions returned in `ye`, and `ie` specifies which event occurred.

`sol` — Structure for evaluation
structure array

Structure for evaluation, returned as a structure array. Use this structure with the `deval` function to evaluate the solution at any point in the interval `[t0 tf]`. The `sol` structure array always includes these fields:

Structure Field	Description
`sol.x`	Row vector of the steps chosen by the solver.
`sol.y`	Solutions. Each column `sol.y(:,i)` contains the solution at time `sol.x(i)`.
`sol.solver`	Solver name.

Additionally, if you specify the `Events` option and events are detected, then `sol` also includes these fields:

Structure Field	Description
`sol.xe`	Points when events occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

Algorithms

`ode45` is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a single-step solver – in computing $y(t_n)$, it needs only the solution at the immediately preceding time point, $y(t_{n-1})$ [1], [2].

References

[1] Dormand, J. R. and P. J. Prince, "A family of embedded Runge-Kutta formulae," *J. Comp. Appl. Math.*, Vol. 6, 1980, pp. 19–26.

[2] Shampine, L. F. and M. W. Reichelt, "The MATLAB ODE Suite," *SIAM Journal on Scientific Computing*, Vol. 18, 1997, pp. 1–22.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- All `odeset` option arguments must be constant.
- Code generation does not support a constant mass matrix in the options structure. Provide a mass matrix as a function.
- You must provide at least the two output arguments `T` and `Y`.
- Input types must be homogeneous—all double or all single.
- Variable-sizing support must be enabled. Code generation requires dynamic memory allocation when `tspan` has two elements or you use event functions.

See Also

`deval` | `ode113` | `ode15s` | `ode23` | `odeget` | `odeset` | `odextend`

Topics

"Choose an ODE Solver"
"Summary of ODE Options"
"Solve Nonstiff ODEs"
"Troubleshoot Common ODE Problems"

Introduced before R2006a

ode113

Solve nonstiff differential equations — variable order method

Syntax

```
[t,y] = ode113(odefun,tspan,y0)
[t,y] = ode113(odefun,tspan,y0,options)
[t,y,te,ye,ie] = ode113(odefun,tspan,y0,options)
sol = ode113( ___ )
```

Description

`[t,y] = ode113(odefun,tspan,y0)`, where `tspan = [t0 tf]`, integrates the system of differential equations $y' = f(t, y)$ from `t0` to `tf` with initial conditions `y0`. Each row in the solution array `y` corresponds to a value returned in column vector `t`.

All MATLAB ODE solvers can solve systems of equations of the form $y' = f(t, y)$, or problems that involve a mass matrix, $M(t, y)y' = f(t, y)$. The solvers all use similar syntaxes. The `ode23s` solver only can solve problems with a mass matrix if the mass matrix is constant. `ode15s` and `ode23t` can solve problems with a mass matrix that is singular, known as differential-algebraic equations (DAEs). Specify the mass matrix using the `Mass` option of `odeset`.

`[t,y] = ode113(odefun,tspan,y0,options)` also uses the integration settings defined by `options`, which is an argument created using the `odeset` function. For example, use the `AbsTol` and `RelTol` options to specify absolute and relative error tolerances, or the `Mass` option to provide a mass matrix.

`[t,y,te,ye,ie] = ode113(odefun,tspan,y0,options)` additionally finds where functions of (t,y), called event functions, are zero. In the output, `te` is the time of the event, `ye` is the solution at the time of the event, and `ie` is the index of the triggered event.

For each event function, specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the `'Events'` property to a function, such as `myEventFcn` or `@myEventFcn`, and creating a

corresponding function: [value,isterminal,direction] = myEventFcn(t,y). For more information, see “ODE Event Location”.

sol = ode113(___) returns a structure that you can use with deval to evaluate the solution at any point on the interval [t0 tf]. You can use any of the input argument combinations in previous syntaxes.

Examples

ODE with Single Solution Component

Simple ODEs that have a single solution component can be specified as an anonymous function in the call to the solver. The anonymous function must accept two inputs (t,y) even if one of the inputs is not used.

Solve the ODE

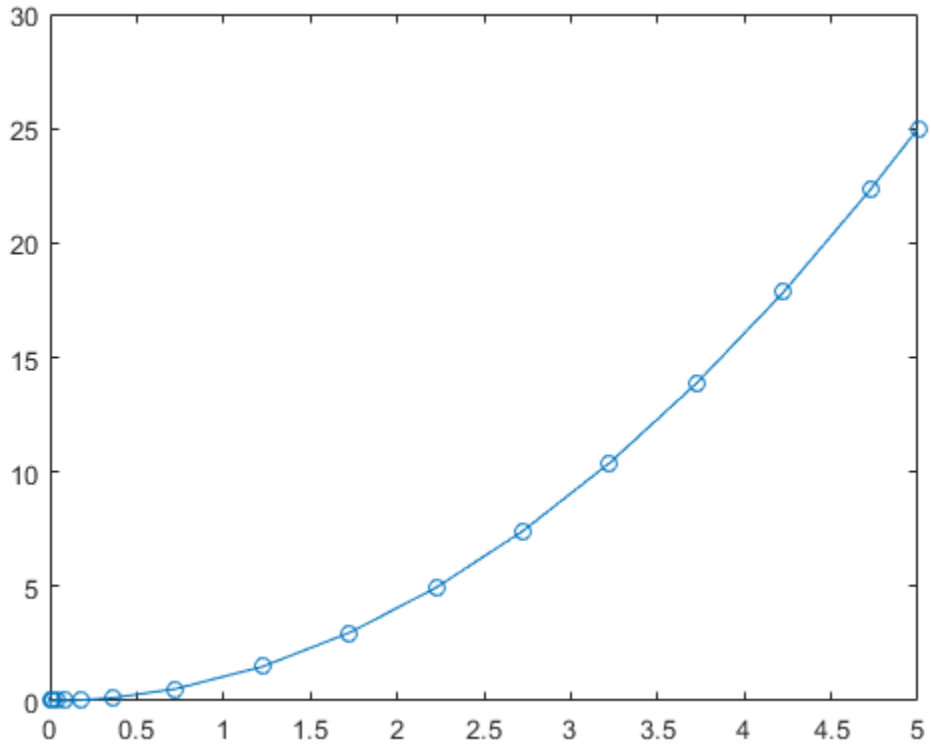
$$y' = 2t\,.$$

Use a time interval of [0,5] and the initial condition y0 = 0.

```
tspan = [0 5];
y0 = 0;
[t,y] = ode113(@(t,y) 2*t, tspan, y0);
```

Plot the solution.

```
plot(t,y,'-o')
```



Solve Nonstiff Equation

The van der Pol equation is a second order ODE

$$y_1'' - \mu\left(1 - y_1^2\right) y_1' + y_1 = 0,$$

where $\mu > 0$ is a scalar parameter. Rewrite this equation as a system of first-order ODEs by making the substitution $y_1' = y_2$. The resulting system of first-order ODEs is

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= \mu(1 - y_1^2)y_2 - y_1. \end{aligned}$$

The function file `vdp1.m` represents the van der Pol equation using $\mu = 1$. The variables y_1 and y_2 are the entries `y(1)` and `y(2)` of a two-element vector, `dydt`.

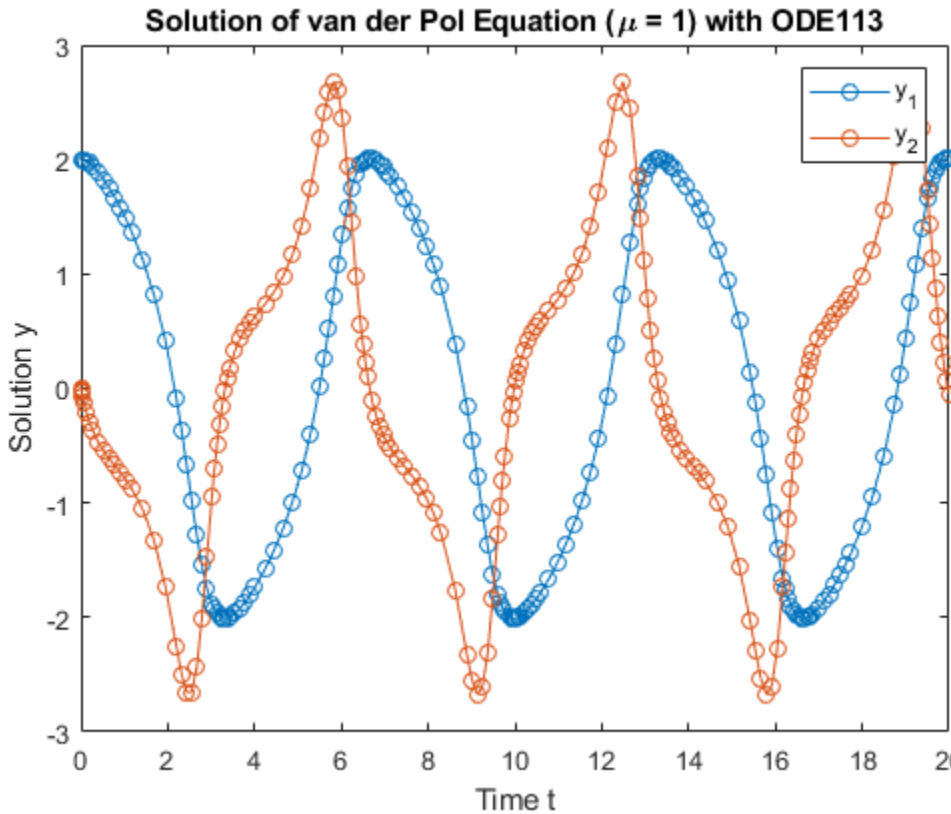
```
function dydt = vdp1(t,y)
%VDP1  Evaluate the van der Pol ODEs for mu = 1
%
%   See also ODE113, ODE23, ODE45.

%   Jacek Kierzenka and Lawrence F. Shampine
%   Copyright 1984-2014 The MathWorks, Inc.

dydt = [y(2); (1-y(1)^2)*y(2)-y(1)];
```

Solve the ODE using the `ode113` function on the time interval `[0 20]` with initial values `[2 0]`. The resulting output is a column vector of time points `t` and a solution array `y`. Each row in `y` corresponds to a time returned in the corresponding row of `t`. The first column of `y` corresponds to y_1, and the second column to y_2.

```
[t,y] = ode113(@vdp1,[0 20],[2; 0]);
```

Plot the solutions for y_1 and y_2 against `t`.

```
plot(t,y(:,1),'-o',t,y(:,2),'-o')
title('Solution of van der Pol Equation (\mu = 1) with ODE113');
xlabel('Time t');
ylabel('Solution y');
legend('y_1','y_2')
```



Pass Extra Parameters to ODE Function

`ode113` only works with functions that use two input arguments, `t` and `y`. However, you can pass in extra parameters by defining them outside the function and passing them in when you specify the function handle.

Solve the ODE

$$y'' = \frac{A}{B} t y.$$

Rewriting the equation as a first-order system yields

$$
\begin{aligned}
y_1' &= y_2 \\
y_2' &= \frac{A}{B} t y_1.
\end{aligned}
$$

`odefcn.m` represents this system of equations as a function that accepts four input arguments: `t`, `y`, `A`, and `B`.

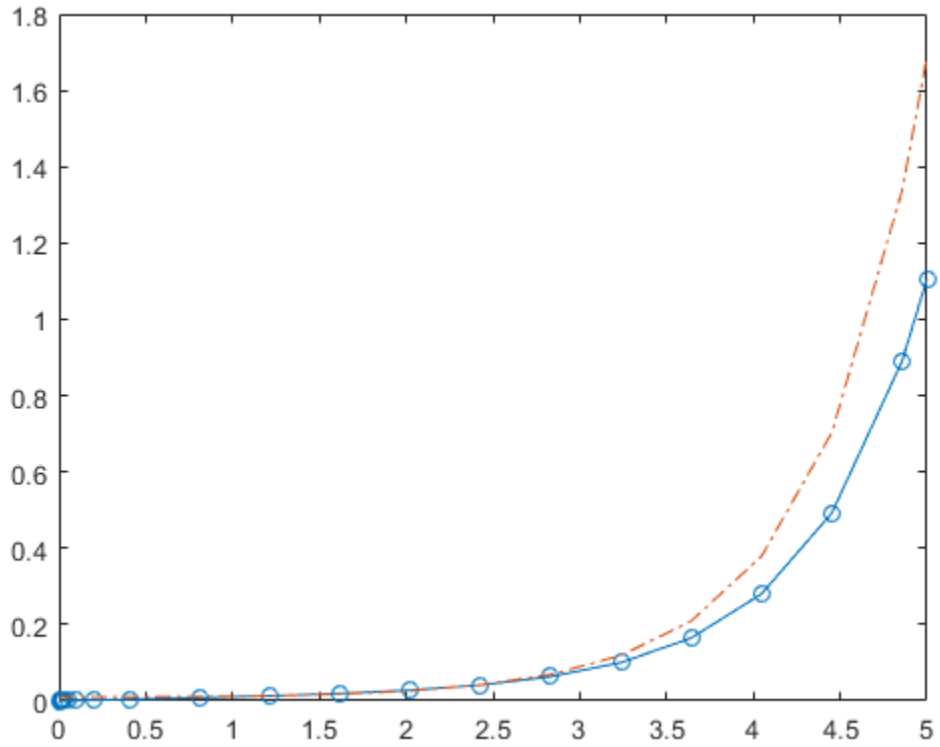
```
function dydt = odefcn(t,y,A,B)
dydt = zeros(2,1);
dydt(1) = y(2);
dydt(2) = (A/B)*t.*y(1);
```

Solve the ODE using `ode113`. Specify the function handle such that it passes in the predefined values for `A` and `B` to `odefcn`.

```
A = 1;
B = 2;
tspan = [0 5];
y0 = [0 0.01];
[t,y] = ode113(@(t,y) odefcn(t,y,A,B), tspan, y0);
```

Plot the results.

```
plot(t,y(:,1),'-o',t,y(:,2),'-.')
```



ODE with Stringent Error Threshold

Compared to `ode45`, the `ode113` solver is better at solving problems with stringent error tolerances. A common situation where `ode113` excels is in orbital dynamics problems, where the solution curve is smooth and requires high accuracy.

The two-body problem considers two interacting masses `m1` and `m2` orbiting in a common plane. In this example, one of the masses is significantly larger than the other. With the heavy body at the origin, the equations of motion are

$$
\begin{aligned}
x'' &= -x/r^3 \\
y'' &= -y/r^3,
\end{aligned}
$$

where

$$r = \sqrt{x^2 + y^2}.$$

To solve the system, first convert to a system of four first-order ODEs using the substitutions

$$
\begin{aligned}
y_1 &= x \\
y_2 &= x' \\
y_3 &= y \\
y_4 &= y'.
\end{aligned}
$$

The substitutions produce the first-order system

$$
\begin{aligned}
y_1' &= y_2 \\
y_2' &= -y_1/r^3 \\
y_3' &= y_4 \\
y_4' &= -y_3/r^3.
\end{aligned}
$$

The function `twobodyode` codes the system of equations for the two-body problem.

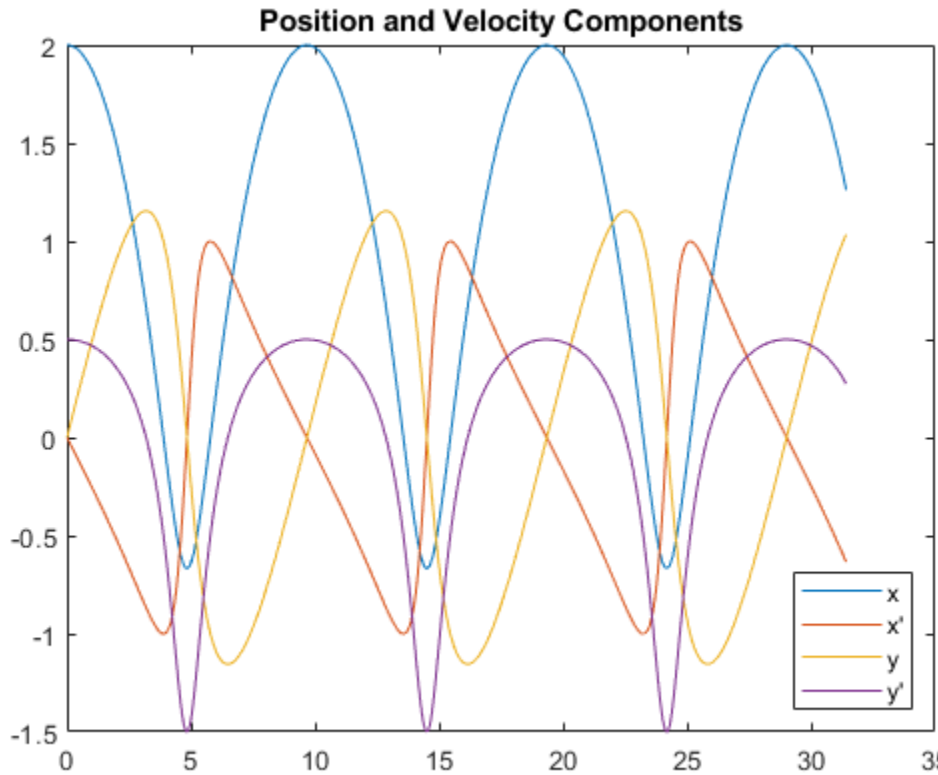
```
function dy = twobodyode(t,y)
% Two body problem with one mass much larger than the other.
r = sqrt(y(1)^2 + y(3)^2);
dy = [y(2);
    -y(1)/r^3;
    y(4);
    -y(3)/r^3];
```

Save `twobodyode.m` in your working directory, then solve the ODE using `ode113`. Use stringent error tolerances of `1e-13` for `RelTol` and `1e-14` for `AbsTol`.

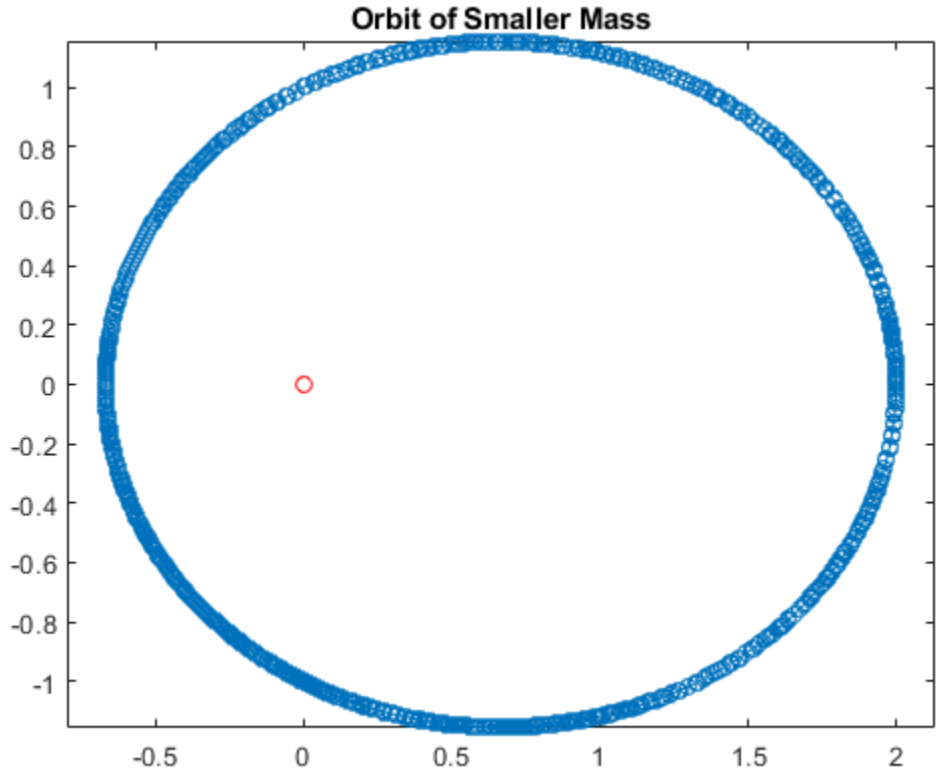
```
opts = odeset('Reltol',1e-13,'AbsTol',1e-14,'Stats','on');
tspan = [0 10*pi];
y0 = [2 0 0 0.5];

[t,y] = ode113(@twobodyode, tspan, y0, opts);
```

```
plot(t,y)
legend('x','x''','y','y''','Location','SouthEast')
title('Position and Velocity Components')
```

```
924 successful steps
4 failed attempts
1853 function evaluations
```




```
figure
plot(y(:,1),y(:,3),'-o',0,0,'ro')
axis equal
title('Orbit of Smaller Mass')
```



Compared to `ode45`, the `ode113` solver is able to obtain the solution faster and with fewer function evaluations.

Input Arguments

`odefun` — Functions to solve
function handle

Functions to solve, specified as a function handle which defines the functions to be integrated.

The function `dydt = odefun(t,y)`, for a scalar `t` and a column vector `y`, must return a column vector `dydt` of data type `single` or `double` that corresponds to $f(t, y)$. `odefun` must accept both input arguments, `t` and `y`, even if one of the arguments is not used in the function.

For example, to solve $y' = 5y - 3$, use the function:

```
function dydt = odefun(t,y)
dydt = 5*y-3;
```

For a system of equations, the output of `odefun` is a vector. Each element in the vector is the solution to one equation. For example, to solve

$$y'_1 = y_1 + 2y_2$$
$$y'_2 = 3y_1 + 2y_2$$

use the function:

```
function dydt = odefun(t,y)
dydt = zeros(2,1);
dydt(1) = y(1)+2*y(2);
dydt(2) = 3*y(1)+2*y(2);
```

For information on how to provide additional parameters to the function `odefun`, see "Parameterizing Functions".

Example: `@myFcn`

Data Types: `function_handle`

`tspan` — Interval of integration
vector

Interval of integration, specified as a vector. At minimum, `tspan` must be a two element vector `[t0 tf]` specifying the initial and final times. To obtain solutions at specific times between `t0` and `tf`, use a longer vector of the form `[t0,t1,t2,...,tf]`. The elements in `tspan` must be all increasing or all decreasing.

The solver imposes the initial conditions given by `y0` at the initial time `tspan(1)`, then integrates from `tspan(1)` to `tspan(end)`:

- If `tspan` has two elements, `[t0 tf]`, then the solver returns the solution evaluated at each internal integration step within the interval.

- If `tspan` has more than two elements `[t0,t1,t2,...,tf]`, then the solver returns the solution evaluated at the given points. However, the solver does not step precisely to each point specified in `tspan`. Instead, the solver uses its own internal steps to compute the solution, then evaluates the solution at the requested points in `tspan`. The solutions produced at the specified points are of the same order of accuracy as the solutions computed at each internal step.

  Specifying several intermediate points has little effect on the efficiency of computation, but for large systems it can affect memory management.

The values of `tspan` are used by the solver to calculate suitable values for `InitialStep` and `MaxStep`:

- If `tspan` contains several intermediate points `[t0,t1,t2,...,tf]`, then the specified points give an indication of the scale for the problem, which can affect the value of `InitialStep` used by the solver. Therefore, the solution obtained by the solver might be different depending on whether you specify `tspan` as a two-element vector or as a vector with intermediate points.
- The initial and final values in `tspan` are used to calculate the maximum step size `MaxStep`. Therefore, changing the initial or final values in `tspan` could lead to the solver using a different step sequence, which might change the solution.

Example: `[1 10]`

Example: `[1 3 5 7 9 10]`

Data Types: `single` | `double`

`y0` — Initial conditions
vector

Initial conditions, specified as a vector. `y0` must be the same length as the vector output of `odefun`, so that `y0` contains an initial condition for each equation defined in `odefun`.

Data Types: `single` | `double`

`options` — Option structure
structure array

Option structure, specified as a structure array. Use the `odeset` function to create or modify the options structure. See “Summary of ODE Options” for a list of the options compatible with each solver.

Example: `options = odeset('RelTol',1e-5,'Stats','on','OutputFcn',@odeplot)` specifies a relative error tolerance of `1e-5`, turns on the display of solver statistics, and specifies the output function `@odeplot` to plot the solution as it is computed.

Data Types: `struct`

Output Arguments

`t` — Evaluation points
column vector

Evaluation points, returned as a column vector.

- If `tspan` contains two elements, `[t0 tf]`, then `t` contains the internal evaluation points used to perform the integration.
- If `tspan` contains more than two elements, then `t` is the same as `tspan`.

`y` — Solutions
array

Solutions, returned as an array. Each row in `y` corresponds to the solution at the value returned in the corresponding row of `t`.

`te` — Time of events
column vector

Time of events, returned as a column vector. The event times in `te` correspond to the solutions returned in `ye`, and `ie` specifies which event occurred.

`ye` — Solution at time of events
array

Solution at time of events, returned as an array. The event times in `te` correspond to the solutions returned in `ye`, and `ie` specifies which event occurred.

`ie` — Index of triggered event function
column vector

Index of triggered event function, returned as a column vector. The event times in `te` correspond to the solutions returned in `ye`, and `ie` specifies which event occurred.

`sol` — Structure for evaluation
structure array

Structure for evaluation, returned as a structure array. Use this structure with the `deval` function to evaluate the solution at any point in the interval `[t0 tf]`. The `sol` structure array always includes these fields:

Structure Field	Description
`sol.x`	Row vector of the steps chosen by the solver.
`sol.y`	Solutions. Each column `sol.y(:,i)` contains the solution at time `sol.x(i)`.
`sol.solver`	Solver name.

Additionally, if you specify the `Events` option and events are detected, then `sol` also includes these fields:

Structure Field	Description
`sol.xe`	Points when events occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

Algorithms

`ode113` is a variable-step, variable-order (VSVO) Adams-Bashforth-Moulton PECE solver of orders 1 to 13. The highest order used appears to be 12, however, a formula of order 13 is used to form the error estimate and the function does local extrapolation to advance the integration at order 13.

`ode113` may be more efficient than `ode45` at stringent tolerances or if the ODE function is particularly expensive to evaluate. `ode113` is a multistep solver — it normally needs the solutions at several preceding time points to compute the current solution [1], [2].

References

[1] Shampine, L. F. and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, San Francisco, 1975.

[2] Shampine, L. F. and M. W. Reichelt, "The MATLAB ODE Suite," *SIAM Journal on Scientific Computing*, Vol. 18, 1997, pp. 1–22.

See Also

`deval` | `ode45` | `odeget` | `odeset` | `odextend`

Topics

"Choose an ODE Solver"
"Summary of ODE Options"
"Solve Nonstiff ODEs"
"Troubleshoot Common ODE Problems"

Introduced before R2006a

odeget

Extract ODE option values

Syntax

```
v = odeget(options,'Name')
v = odeget(options,'Name',default)
```

Description

`v = odeget(options,'Name')` extracts the value of the named option from `options`, which is a structure containing option values. Use `odeset` to create or update the options structure.

`v = odeget(options,'Name',default)` returns the value `v = default` if the named option does not have a value specified in `options`.

Examples

Extract ODE Option Values

Create an options structure using `odeset` that contains several values for different options.

```
M = @(t) [t 0; 0 -t];
options = odeset('RelTol',1e-4,'AbsTol',1e-5,'OutputFcn',@odephas2,...
    'Mass',M,'MassSingular','no','MStateDependence','none')
```

```
options = struct with fields:
              AbsTol: 1.0000e-05
                 BDF: []
              Events: []
         InitialStep: []
            Jacobian: []
```

```
           JConstant: []
            JPattern: []
                Mass: @(t)[t,0;0,-t]
        MassSingular: 'no'
            MaxOrder: []
             MaxStep: []
         NonNegative: []
         NormControl: []
           OutputFcn: @odephas2
           OutputSel: []
              Refine: []
              RelTol: 1.0000e-04
               Stats: []
          Vectorized: []
    MStateDependence: 'none'
           MvPattern: []
        InitialSlope: []
```

Use `odeget` to extract the value of the `OutputFcn` field from the options structure.

```
v = odeget(options,'OutputFcn')
```

```
v = function_handle with value:
    @odephas2
```

Now extract the value of the `Refine` field. Since this field is not set, `odeget` returns an empty matrix `[]`.

```
v = odeget(options,'Refine')
```

```
v =

     []
```

You can specify a third input to `odeget` to change the default return value. This ensures that `v` is never empty.

```
v = odeget(options,'Refine',1)
```

```
v = 1
```

Input Arguments

`options` — Options structure
structure

Options structure. Use `odeset` to create or modify the options structure.

Example: `options = odeset('RelTol',1e-4,'AbsTol',1e-5)` returns an options structure with values specified for the relative and absolute error tolerances.

Data Types: `struct`

`Name` — Option name
any valid option from `odeset`

Option name, specified as any valid option accepted by `odeset`:

- Error control — `'AbsTol'`, `'RelTol'`, `'NormControl'`
- Output control — `'NonNegative'`, `'OutputFcn'`, `'OutputSel'`, `'Refine'`, `'Stats'`
- Solver steps — `'InitialStep'`, `'MaxStep'`
- Event functions — `'Events'`
- Jacobian matrix — `'Jacobian'`, `'JPattern'`, `'Vectorized'`
- Mass matrix — `'Mass'`, `'MStateDependence'`, `'MvPattern'`, `'MassSingular'`, `'InitialSlope'`
- Algorithm options for `ode15s` and `ode15i` — `'MaxOrder'`, `'BDF'`

Example: `v = odeget(options,'AbsTol')`

Data Types: `char` | `string`

`default` — Default return value
any MATLAB object

Default return value, specified as any valid MATLAB object.

Example: `v = odeget(options,'AbsTol',1e-6)` returns `1e-6` if `options` does not have a value set for `AbsTol`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell` | `function_handle` | `categorical`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The `name` argument must be constant.

See Also

`deval` | `odeset` | `odextend`

Topics

"Choose an ODE Solver"
"Summary of ODE Options"
"Troubleshoot Common ODE Problems"

Introduced before R2006a

odeset

Create or modify options structure for ODE and PDE solvers

Syntax

```
options = odeset(Name,Value,...)
options = odeset(oldopts,Name,Value,...)
options = odeset(oldopts,newopts)
odeset
```

Description

`options = odeset(Name,Value,...)` creates an options structure that you can pass as an argument to ODE and PDE solvers. In the structure, `options`, the named options have the specified values. Any unspecified options have default values. For example, `options = odeset('RelTol',1e-3)` returns an options structure with `RelTol` set to `1e-3`.

`options = odeset(oldopts,Name,Value,...)` modifies an existing options structure, `oldopts`, using the newly specified name-value pairs. This overwrites any old values of the specified options, and adds values for new options to the structure.

`options = odeset(oldopts,newopts)` modifies an existing options structure, `oldopts`, by combining it with a new options structure, `newopts`. Any new options not equal to `[]` overwrite the corresponding options in `oldopts`.

`odeset` with no input arguments displays all possible option names and their possible values. Default values are indicated with `{}`, where applicable.

Examples

Set and Update ODE Options

Create an options structure that contains values for `RelTol` and `AbsTol`.

```
options = odeset('RelTol',1e-8,'AbsTol',1e-10);
```

Update the value of `AbsTol` in the existing options structure.

```
options = odeset(options,'AbsTol',1e-9)
```

```
options = struct with fields:
              AbsTol: 1.0000e-09
                 BDF: []
              Events: []
         InitialStep: []
            Jacobian: []
           JConstant: []
            JPattern: []
                Mass: []
        MassSingular: []
            MaxOrder: []
             MaxStep: []
         NonNegative: []
         NormControl: []
           OutputFcn: []
           OutputSel: []
              Refine: []
              RelTol: 1.0000e-08
               Stats: []
          Vectorized: []
    MStateDependence: []
            MvPattern: []
        InitialSlope: []
```

Combine Options Structures

Create two options structures.

```
opts_1 = odeset('RelTol',1e-8,'AbsTol',1e-9,'OutputFcn',@odeplot,'Stats','on');
```

```
opts_2 = odeset('Mass',@(t) [t 0; 0 -t],'MStateDependence','none',...
    'MassSingular','no','OutputFcn',@odephas2);
```

Combine the options structures, giving preference to `opts_2`. Since both structures contain different values for `OutputFcn`, the value in `opts_2` overrides the one in `opts_1`.

```
opts = odeset(opts_1,opts_2)
```

```
opts = struct with fields:
              AbsTol: 1.0000e-09
                 BDF: []
              Events: []
         InitialStep: []
            Jacobian: []
           JConstant: []
            JPattern: []
                Mass: @(t)[t,0;0,-t]
        MassSingular: 'no'
            MaxOrder: []
             MaxStep: []
         NonNegative: []
         NormControl: []
           OutputFcn: @odephas2
           OutputSel: []
              Refine: []
              RelTol: 1.0000e-08
               Stats: 'on'
          Vectorized: []
    MStateDependence: 'none'
           MvPattern: []
        InitialSlope: []
```

Input Arguments

`oldopts` — Old options structure
structure

Old options structure, specified as a structure previously created using `odeset`.

Data Types: `struct`

`newopts` — New options structure
structure

New options structure, specified as a structure previously created using `odeset`.

Data Types: `struct`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `options = odeset('AbsTol',1e-3,'Reltol',1e-2,'Jacobian',@J,'Mass',M)` specifies thresholds for the absolute and relative error tolerances, a function that returns the Jacobian, and a mass matrix.

Error Control

`RelTol` — Relative error tolerance
`1e-3` (default) | positive scalar

Relative error tolerance, specified as the comma-separated pair consisting of `'RelTol'` and a positive scalar. This tolerance measures the error relative to the magnitude of each solution component. Roughly speaking, it controls the number of correct digits in all solution components, except those smaller than the absolute tolerance `AbsTol`.

At each step, the ODE solver estimates the local error `e` in the `i`th component of the solution. To be successful, the step must have acceptable error, as determined by both the relative and absolute error tolerances:

```
|e(i)| <= max(RelTol*abs(y(i)),AbsTol(i))
```

Example: `opts = odeset('RelTol',1e-5,'AbsTol',1e-7)`

Data Types: `single` | `double`

`AbsTol` — Absolute error tolerance
`1e-6` (default) | positive scalar | vector

Absolute error tolerance, specified as the comma-separated pair consisting of `'AbsTol'` and a positive scalar or vector. This tolerance is a threshold below which the value of the solution becomes unimportant. If the solution `|y|` is smaller than `AbsTol`, then the solver does not need to obtain any correct digits in `|y|`. For this reason, the value of `AbsTol` should take into account the scale of the solution components.

If `AbsTol` is a vector, then it must be the same length as the solution. If `AbsTol` is a scalar, then the value applies to all solution components.

At each step, the ODE solver estimates the local error `e` in the `i`th component of the solution. To be successful, the step must have acceptable error, as determined by both the relative and absolute error tolerances:

```
|e(i)| <= max(RelTol*abs(y(i)),AbsTol(i))
```

Example: `opts = odeset('RelTol',1e-5,'AbsTol',1e-7)`

Data Types: `single` | `double`

`NormControl` — Control error relative to norm
`'off'` (default) | `'on'`

Control error relative to the norm of the solution, specified as the comma-separated pair consisting of `'NormControl'` and `'on'` or `'off'`. When `NormControl` is `'on'`, the solvers control the error `e` at each step using the norm of the solution rather than its absolute value:

```
norm(e(i)) <= max(RelTol*norm(y(i)),AbsTol(i))
```

Example: `opts = odeset('NormControl','on')`

Data Types: `char` | `string`

Solver Output

`NonNegative` — Nonnegative solution components
`[]` (default) | scalar | vector

Nonnegative solution components, specified as the comma-separated pair consisting of `'NonNegative'` and a scalar or vector. The scalar or vector selects which solution components must be nonnegative.

Note `NonNegative` is not available for `ode23s` or `ode15i`. Additionally, for `ode15s`, `ode23t`, and `ode23tb` it is not available for problems where there is a mass matrix.

Example: `opts = odeset('NonNegative',1)` specifies that the first solution component must be nonnegative.

Data Types: `single` | `double`

`OutputFcn` — Output function
`[]` or `@odeplot` (default) | function handle

Output function, specified as the comma-separated pair consisting of `'OutputFcn'` and a function handle. The ODE solver calls the output function after each successful time step. If you call an ODE solver with no outputs, then the output function defaults to `@odeplot`, which plots all of the solution components as they are computed. Otherwise, the default is `[]`.

These are the built-in output functions that you can use with `OutputFcn`:

Function Name	Description
`odeplot`	Plot all components of the solution vs. time
`odephas2`	2-D phase plane plot of the first two solution components
`odephas3`	3-D phase plane plot of the first three solution components
`odeprint`	Print solution and time step

If you write a custom output function, then it must be of the form

```
status = myOutputFcn(t,y,flag)
```

The output function must also respond appropriately to these flags:

Flag	Description
`'init'`	The solver calls `myOutputFcn([tspan(1) tspan(end)],y0,'init')` before beginning the integration to allow the output function to initialize. `tspan` and `y0` are the input arguments to the ODE solver.

Flag	Description
`[]`	The solver calls `status = myOutputFcn(t,y,[])` after each integration step for which output is requested. `t` contains points where output was generated during the step, and `y` is the numerical solution at the points in `t`. If `t` is a vector, then the `i`th column of `y` corresponds to the `i`th element of `t`. • If `length(tspan) > 2`, then the output is produced at every point in `tspan`. • If `length(tspan) = 2`, then the output is produced according to the `Refine` option. `myOutputFcn` must return a status of `0` or `1`. If `status = 1`, then the solver halts integration. You can use this mechanism, for instance, to implement a **Stop** button.
`'done'`	The solver calls `myOutputFcn([],[],'done')` once integration is complete to allow the output function to perform cleanup tasks.

Data Types: `function_handle`

`OutputSel` — Component selection for output function
vector of indices

Component selection for output function, specified as the comma-separated pair consisting of `'OutputSel'` and a vector of indices. The vector specifies which components of the solution to pass to the output function.

Example: `opts = odeset('OutputFcn',@myFcn,'OutputSel',[1 3])` passes the first and third components of the solution to the output function.

`Refine` — Solution refinement factor

1 (most solvers) or 4 (`ode45` only) (default) | scalar

Solution refinement factor, specified as the comma-separated pair consisting of `'Refine'` and a scalar. The scalar specifies a factor by which the number of output points should increase in each step.

- If the refinement factor is `1`, then the solver returns solutions only at the end of each step.
- If the refinement factor is `n>1`, then the solver subdivides each step into `n` smaller intervals and returns solutions at each point.

The default value for most solvers is `1`, but `ode45` uses a default of `4` to compensate for large step sizes.

The extra values produced by the refinement factor are computed by means of continuous extension formulas. These are specialized formulas used by the ODE solvers to obtain accurate solutions between computed time steps without significant increase in computation time.

Note `Refine` does not apply when `length(tspan) > 2`, or when the ODE solver returns the solution as a structure.

Example: `opts = odeset('Refine',5)` increases the number of output points by a factor of five.

`Stats` — Solver statistics

`'off'` (default) | `'on'`

Solver statistics, specified as the comma-separated pair consisting of `'Stats'` and `'on'` or `'off'`. When `'on'`, the solver displays information after completing the solution:

- The number of successful steps
- The number of failed attempts
- The number of times the ODE function was called to evaluate $f(t, y)$

Implicit solvers display additional information about the solution:

- The number of times that the partial derivatives matrix $\partial f / \partial y$ was formed

- The number of LU decompositions
- The number of solutions of linear systems

Example: `opts = odeset('Stats','on')`

Data Types: `char` | `string`

Step-Size

`InitialStep` — Suggested initial step size

scalar

Suggested initial step size, specified as the comma-separated pair consisting of `'InitialStep'` and a positive scalar. `InitialStep` sets an upper bound on the magnitude of the first step size that the solver tries.

If you do not specify an initial step size, then the solver bases the initial step size on the slope of the solution at the initial time point, `tspan(1)`. If the slope of all solution components is zero, then the solver might try a step size that is too large. If you are aware that this is occurring, or if you want to be sure that the solver resolves important behavior at the beginning of the integration, then use `InitialStep` to provide a suitable initial step size.

Example: `opts = odeset('InitialStep',1e-3)` sets an upper bound of `1e-3` on the size of the initial step.

`MaxStep` — Maximum step size

`0.1*abs(t0-tf)` (default) | scalar

Maximum step size, specified as the comma-separated pair consisting of `'MaxStep'` and a positive scalar. `MaxStep` sets an upper bound on the size of any step taken by the solver. If the equation has periodic behavior, for example, then setting `MaxStep` to a fraction of the period ensures that the solver does not enlarge the step so much that it steps over an area of interest.

- Do not use `MaxStep` just to obtain more output points, as it can significantly slow down the integration. Instead, use the `Refine` option to compute additional points at low computational cost.
- Do not use `MaxStep` to increase the accuracy of the solution. If the solution is not accurate enough, then reduce the value of `RelTol` and use the solution to determine a suitable value for `AbsTol`.

- Avoid using `MaxStep` to make sure the solver does not step over some behavior that occurs only once in the integration interval. If you know the time at which the behavior occurs, then break the interval into two pieces and call the solver twice. If you do not know the time at which the change occurs, try reducing `RelTol` and `AbsTol`. Use `MaxStep` only as a last resort in this case.

Example: `opts = odeset('MaxStep',1e-2)`

Event Location

`Events` — Event function

function handle

Event function, specified as the comma-separated pair consisting of `'Events'` and a function handle such as `@myEventsFcn`.

Function Signature

For ODEs: The event function specified by the function handle must have the general form

```
[value,isterminal,direction] = myEventsFcn(t,y)
```

For PDEs: The event function specified by the function handle must have the general form

```
[value,isterminal,direction] = myEventsFcn(m,t,xmesh,umesh)
```

In both cases, `value`, `isterminal`, and `direction` are vectors whose `i`th element corresponds to the `i`th event function:

- `value(i)` is the value of the `i`th event function.
- `isterminal(i) = 1` if the integration is to terminate at a zero of this event function. Otherwise, it is `0`.
- `direction(i) = 0` if all zeros are to be located (the default). A value of `+1` locates only zeros where the event function is increasing, and `-1` locates only zeros where the event function is decreasing.

See “Parameterizing Functions” to see how to pass in additional inputs to the events function.

Events Output

If you specify an events function, you can call the solver with three extra output arguments, such as

```
[t,y,te,ye,ie] = odeXY(odefun,tspan,y0,options)
```

The three additional outputs returned by the solver correspond to the detected events:

- `te` is a column vector of the times at which events occurred.
- `ye` is the solution value corresponding to the event times in `te`.
- `ie` are indices into the vector returned by the events function. The values indicate which event the solver detected.

Alternatively, you can call the solver with a single output as

```
sol = odeXY(odefun,tspan,y0,options)
```

In this case, the event information is stored in the structure as `sol.te`, `sol.ye`, and `sol.ie`.

Diagnostics

The root finding mechanism employed by the ODE/PDE solver in conjunction with the event function has these limitations:

- If a terminal event occurs during the first step of the integration, then the solver registers the event as nonterminal and continues integrating.
- If more than one terminal event occurs during the first step, then only the first event registers and the solver continues integrating.
- Zeros are determined by sign crossings between steps. Therefore, zeros with an even number of crossings between steps can be missed.

If the solver steps past events, try reducing `RelTol` and `AbsTol` to improve accuracy. Alternatively, set `MaxStep` to place an upper bound on the step size. Adjusting `tspan` does not change the steps taken by the solver.

Examples

- See “ODE Event Location” for examples of ODE event functions that detect the bounces of a ball and orbits of celestial bodies.
- See “Solve Oscillatory PDE with Event Logging” on page 1-11047 for an example of a PDE event function detecting the zero-crossings of an oscillatory solution.

Data Types: `function_handle`

Jacobian Matrix

`Jacobian` — Jacobian matrix

matrix | function handle | cell array

Jacobian matrix, specified as the comma-separated pair consisting of `'Jacobian'` and a matrix or function that evaluates the Jacobian. The Jacobian is a matrix of partial derivatives of the function that defines the differential equations.

$$J = \frac{\partial f}{\partial y} = \begin{bmatrix} \frac{\partial f_1}{\partial y_1} & \frac{\partial f_1}{\partial y_2} & \cdots \\ \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial y_2} & \cdots \\ \vdots & \vdots & \end{bmatrix}$$

For the stiff ODE solvers (`ode15s`, `ode23s`, `ode23t`, `ode23tb`, and `ode15i`), providing information about the Jacobian matrix is critical for reliability and efficiency. If you do not provide the Jacobian, then the ODE solver approximates it numerically using finite differences.

For `ode15i` only: The `Jacobian` option must specify matrices for both $\frac{\partial f}{\partial y}$ and $\frac{\partial f}{\partial y'}$. You can provide these matrices as a cell array of two constant matrices $\left\{\frac{\partial f}{\partial y}, \frac{\partial f}{\partial y'}\right\}$, or as a function that computes the matrices and has the general form

```
[dfdy, dfdp] = Fjac(t,y,yp)
```

For very large systems where it is not feasible to provide the entire analytic Jacobian, use the `JPattern` property to pass in the sparsity pattern of the Jacobian matrix. The solver uses the sparsity pattern to calculate a sparse Jacobian.

Example: `opts = odeset('Jacobian',@J)` specifies the function `J` that calculates the Jacobian matrix.

Example: `opts = odeset('Jacobian',[0 1; -2 1])` specifies a constant Jacobian matrix.

Example: `opts = odeset('Jacobian',{A,Ap})` specifies two constant Jacobian matrices for use with `ode15i`.

Data Types: `single` | `double` | `cell` | `function_handle`

`JPattern` — Jacobian sparsity pattern
sparse matrix | cell array

Jacobian sparsity pattern, specified as the comma-separated pair consisting of `'JPattern'` and a sparse matrix. The sparse matrix contains 1s where there might be nonzero entries in the Jacobian. The ODE solver uses the sparsity pattern to generate a sparse Jacobian matrix numerically. Use this option to improve execution time when the ODE system is large and you cannot provide an analytic Jacobian.

For `ode15i` only: Set the `JPattern` option using a cell array containing two sparse matrices `{dfdyPattern, dfdypPattern}`, which are the sparsity patterns for $\frac{\partial f}{\partial y}$ and $\frac{\partial f}{\partial y'}$.

Note If you specify a Jacobian matrix using `Jacobian`, then the solver ignores any setting for `JPattern`.

Example: `opts = odeset('JPattern',S)` specifies the Jacobian sparsity pattern using sparse matrix `S`.

Example: `opts = odeset('JPattern',{dFdy, dFdyp})` specifies two constant Jacobian sparsity patterns for use with `ode15i`.

Data Types: `double` | `cell`

`Vectorized` — Vectorized function toggle
`'off'` (default) | `'on'` | cell array

Vectorized function toggle, specified as the comma-separated pair consisting of `'Vectorized'` and either `'off'` or `'on'`. Use this option to inform the ODE solver that the function is coded so that it accepts and returns vectors for the second argument. That is, `f(t,[y1 y2 y3...])` returns `[f(t,y1) f(t,y2) f(t,y3) ...]`. Compared to evaluating values one at a time, this vectorization allows the solver to reduce the number of function evaluations required to compute all the columns of the Jacobian matrix, and might significantly reduce solution time. See “Array vs. Matrix Operations” for a description of the element-wise operators that support vectorization.

For `ode15i` only: Set the `Vectorized` option using a two-element cell array. Set the first element to `'on'` if `f(t,[y1,y2,...],yp)` returns `[f(t,y1,yp), f(t,y2,yp), ...]`. Set the second element to `'on'` if `f(t,y,[yp1,yp2,...])` returns `[f(t,y,yp1), f(t,y,yp2), ...]`. The default value of `Vectorized` in this case is `{'off','off'}`.

Note If you specify a Jacobian matrix using `Jacobian`, then the solver ignores a setting of `'on'` for `Vectorized`.

Example: `opts = odeset('JPattern',S,'Vectorized','on')` specifies that the function is vectorized and sets the Jacobian sparsity pattern.

Example: `opts = odeset('JPattern',{dy,dyp},'Vectorized',{'on','on'})` specifies that the function is vectorized with respect to `y` and `yp`, and also sets the Jacobian sparsity pattern for use with `ode15i`.

Data Types: `char` | `cell` | `string`

Mass Matrix and DAEs (do not apply to ode15i)

`Mass` — Mass matrix

matrix | function handle

Mass matrix, specified as the comma-separated pair consisting of `'Mass'` and a matrix or function handle. The ODE solvers can solve problems containing a mass matrix of the form $M(t,y)\,y' = f(t,y)$, where $M(t,y)$ is a mass matrix that can be full or sparse (the `ode23s` solver can solve only equations with constant mass matrices).

- When the mass matrix is nonsingular, the equation simplifies to $y' = M^{-1} f(t,y)$ and the ODE has a solution for any initial value. However, it is often more convenient and natural to express the model in terms of the mass matrix directly using $M(t,y)\,y' = f(t,y)$, and avoiding the computation of the matrix inverse reduces the storage and execution time needed to solve the problem.
- When $M(t,y)$ is a singular matrix, then the problem is a system of differential algebraic equations (DAEs). A DAE has a solution only when `y0` is consistent; that is, there exists an initial slope `yp0` such that `M(t0,y0)yp0 = f(t0,y0)`, where `yp0` is specified using the `InitialSlope` option. DAEs are characterized by their differential index, or the number of derivatives required to simplify the system to an equivalent system of ODEs. For DAEs of index 1, solving an initial value problem with consistent initial conditions is much like solving an ODE. The `ode15s` and `ode23t` solvers can solve

DAEs of index 1. When solving DAEs, it is advantageous to formulate the problem so that the mass matrix is a diagonal matrix (a semiexplicit DAE).

In all cases, mass matrices that are time- or state-dependent (instead of constant) require the use of additional options:

- For problems of the form $M(t)y' = f(t, y)$, set the `MStateDependence` option to `'none'`. This ensures that the solver calls the mass matrix function with a single input argument for `t`.
- If the mass matrix depends on `y`, then set `MStateDependence` to either `'weak'` (default) or `'strong'`. In both cases the solver calls the mass matrix function with two inputs `(t,y)`, but the `'weak'` option results in implicit solvers using approximations when solving algebraic equations.
- If the system contains many equations with a strongly state-dependent mass matrix $M(t, y)$, then set `MvPattern` to a sparse matrix `S` to specify the sparsity pattern.

Example: The example files `fem2ode` and `batonode` illustrate different uses of the mass matrix.

Data Types: `single` | `double` | `function_handle`

`MStateDependence` — State dependence of mass matrix
`'weak'` (default) | `'none'` | `'strong'`

State dependence of mass matrix, specified as the comma-separated pair consisting of `'MStateDependence` and `'weak'`, `'strong'`, or `'none'`.

- For problems of the form $M(t)y' = f(t, y)$, set the `MStateDependence` option to `'none'`. This ensures that the solver calls the mass matrix function with a single input argument for `t`.
- If the mass matrix depends on `y`, then set `MStateDependence` to either `'weak'` (default) or `'strong'`. In both cases the solver calls the mass matrix function with two inputs `(t,y)`, but the `'weak'` option results in implicit solvers using approximations when solving algebraic equations.

Example: `opts = odeset('Mass',@M,'MStateDependence','none')` specifies that the mass matrix `M` depends only on `t`.

Data Types: `char` | `string`

`MvPattern` — Mass matrix sparsity pattern
sparse matrix

Mass matrix sparsity pattern, specified as the comma-separated pair consisting of `'MvPattern'` and a sparse matrix. Use this option to specify the sparsity pattern of the matrix $\frac{\partial}{\partial y}[M(t,y)v]$. The sparse matrix `S` has `S(i,j) = 1` if for any `k`, the `(i,k)` component of $M(t,y)$ depends on component `j` of `y`.

Note `MvPattern` is for use by `ode15s`, `ode23t`, and `ode23tb` when `MStateDependence` is `'strong'`.

Example: `opts = odeset('MStateDependence','strong','MvPattern',S)`

`MassSingular` — Singular mass matrix toggle
`'maybe'` (default) | `'yes'` | `'no'`

Singular mass matrix toggle, specified as the comma-separated pair consisting of `'MassSingular'` and `'maybe'`, `'yes'`, or `'no'`. The default value of `'maybe'` causes the solver to test whether the problem is a DAE, by testing whether the mass matrix is singular. Avoid this check by specifying `'yes'` if you know the system is a DAE, or `'no'` if it is not.

Data Types: `char` | `string`

`InitialSlope` — Consistent initial slope
vector of zeros (default) | vector

Consistent initial slope, specified as the comma-separated pair consisting of `'InitialSlope'` and a vector. Use this option with the `ode15s` and `ode23t` solvers when solving DAEs. The specified vector is the initial slope y'_0 such that $M(t_0, y_0)y'_0 = f(t_0, y_0)$. If the specified initial conditions are not consistent, then the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem.

Data Types: `single` | `double`

Only for ode15s and ode15i

`MaxOrder` — Maximum order of formula
5 (default) | 4 | 3 | 2 | 1

Maximum order of formula, specified as the comma-separated pair consisting of `'MaxOrder'` and an integer between `1` and `5`. Use this option to specify the maximum

order used in the numerical differentiation formulas (NDFs) or backward differentiation formulas (BDFs) that are used by the variable-order solvers `ode15s` and `ode15i`.

`BDF` — Toggle to use BDFs with `ode15s`
`'off'` (default) | `'on'`

Toggle to use backward differentiation formulas (BDFs) with `ode15s`, specified as the comma-separated pair consisting of `'BDF'` and either `'off'` or `'on'`. The default numerical differentiation formulas (NDFs) are generally more efficient than BDFs, but the two are closely related.

Example: `opts = odeset('BDF','on','MaxOrder',4)` enables the use of BDFs by `ode15s` with a maximum order of 4.

Data Types: `char` | `string`

Output Arguments

`options` — Options structure
structure

Options structure. `options` can be used as a fourth input argument to `ode45`, `ode23`, `ode113`, `ode15s`, `ode23s`, `ode23t`, `ode23tb`, or `ode15i`.

Tips

- See “Summary of ODE Examples and Files” for a list of ODE examples that illustrate the use of various options.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- All inputs must be constant.

See Also

`deval` | `odeget` | `odextend`

Topics

"Choose an ODE Solver"
"Summary of ODE Options"
"ODE Event Location"
"Nonnegative ODE Solution"
"Troubleshoot Common ODE Problems"

Introduced before R2006a

odextend

Extend solution to ODE

Syntax

```
solext = odextend(sol,odefun,tfinal)
solext = odextend(sol,[],tfinal)
solext = odextend(sol,odefun,tfinal,y0)
solext = odextend(sol,odefun,tfinal,y0,options)
```

Description

`solext = odextend(sol,odefun,tfinal)` extends the solution `sol` by integrating `odefun` from `sol.x(end)` to `tfinal`, using the same ODE solver that created `sol`. The function `odefun` can be different than the original function used to compute `sol`. The lower bound for the independent variable in `solext` is the same as in `sol`, that is, `sol.x(1)`. By default, `odextend` uses:

- The initial conditions `y = sol.y(:,end)` for the subsequent integration.
- The same integration properties and additional input arguments that the ODE solver originally used to compute `sol`. This information is stored in the solution structure `sol` and is later returned in `solext`. Unless you want to change these values, you do not need to pass them to `odextend`.

`solext = odextend(sol,[],tfinal)` extends the solution to the same ODE function that was solved to obtain `sol`.

`solext = odextend(sol,odefun,tfinal,y0)` specifies new initial conditions `y0` for the extended integration instead of using `sol.y(:,end)`.

For the `ode15i` solver: `y0` must be an m-by-2 matrix containing column vectors of initial conditions for the solution components and their derivatives, `y0 = [yinit ypinit]`.

`solext = odextend(sol,odefun,tfinal,y0,options)` uses the integration options defined by `options`, which is an argument created using the `odeset` function. The

specified options override the options that the ODE solver originally used to compute `sol`. You can optionally specify `y0 = []` to use default initial conditions.

Examples

Evaluate and Extend Solution Structure

The van der Pol equation is a second order ODE

$$y_1'' - \mu\left(1 - y_1^2\right)y_1' + y_1 = 0\,.$$

Solve the van der Pol equation with $\mu = 1$ using `ode45`. The function `vdp1.m` ships with MATLAB® and encodes the equations. Specify a single output to return a structure containing information about the solution, such as the solver and evaluation points.

```
tspan = [0 20];
y0 = [2 0];
sol = ode45(@vdp1,tspan,y0)
```

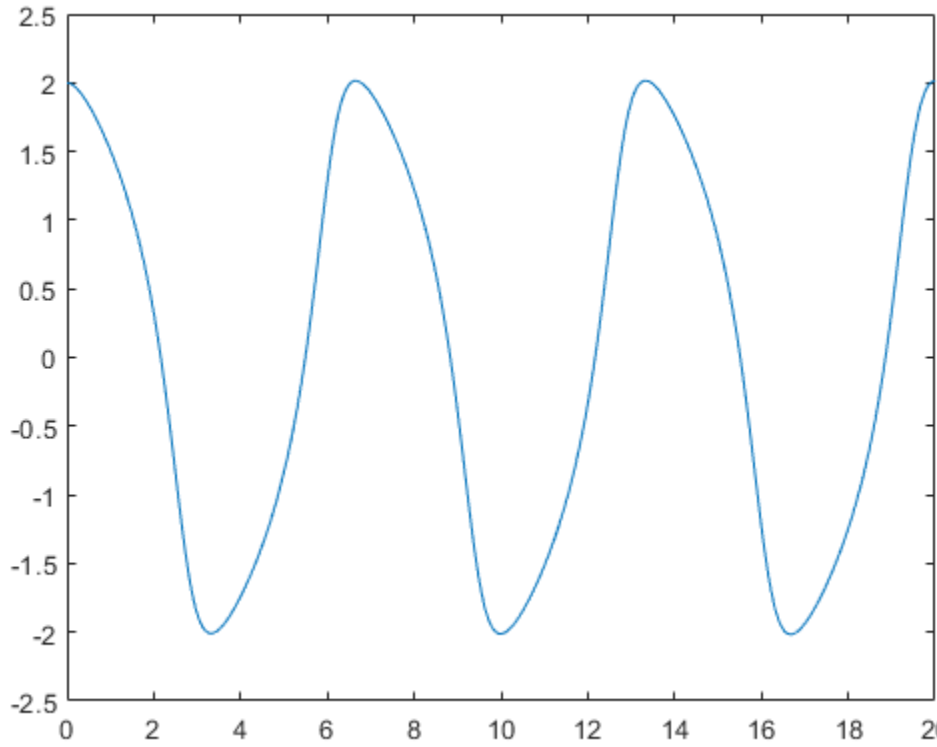
```
sol = struct with fields:
     solver: 'ode45'
    extdata: [1x1 struct]
          x: [1x60 double]
          y: [2x60 double]
      stats: [1x1 struct]
      idata: [1x1 struct]
```

Use `linspace` to generate 250 points in the interval `[0 20]`. Evaluate the solution at these points using `deval`.

```
x = linspace(0,20,250);
y = deval(sol,x);
```

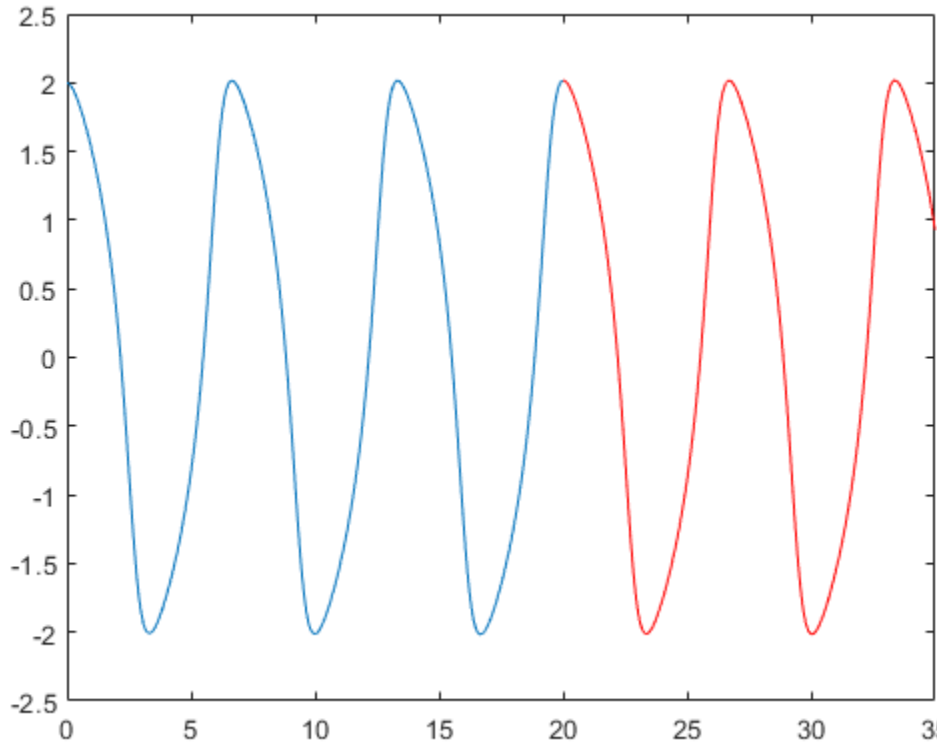
Plot the first component of the solution.

```
plot(x,y(1,:))
```



Extend the solution to $t_f = 35$ using `odextend` and add the result to the original plot.

```
sol_new = odextend(sol,@vdp1,35);
x = linspace(20,35,350);
y = deval(sol_new,x);
hold on
plot(x,y(1,:),'r')
```



Input Arguments

`sol` — Solution structure
structure

Solution structure, specified as a structure returned by an ODE solver. When you call an ODE solver with a single output argument, it returns a solution structure.

Example: `sol = ode45(myode,tspan,y0)`

Data Types: `struct`

`odefun` — Function to solve
`[]` | function handle

Function to solve, specified as a function handle. Use this input to extend the solution using a new or modified ODE function. To continue using the original ODE function used to create the solution structure `sol`, specify `odefun` as an empty input `[]`.

Data Types: `function_handle`

`tfinal` — Final integration time
scalar

Final integration time, specified as a scalar.

Data Types: `single` | `double`

`y0` — Initial conditions
`[]` | scalar | vector | matrix

Initial conditions, specified as a scalar, vector, or matrix. By default `odextend` uses the initial conditions `y = sol.y(:,end)` to extend the integration. Use this input to specify new initial conditions for the extended integration.

For the `ode15i` solver: `y0` must be an m-by-2 matrix containing column vectors of initial conditions for the solution components and their derivatives, `y0 = [yinit ypinit]`.

Data Types: `single` | `double`

`options` — Options structure
structure

Options structure. By default, `odextend` uses the same options and additional inputs as the ODE solver originally used to compute `sol`. Use this input to specify a new options structure that overrides the options used to create `sol`.

Use the `odeset` function to create or modify an ODE options structure.

Data Types: `struct`

Output Arguments

`solext` — Extended solution
structure

Extended solution, returned as a structure. Use this structure with the `deval` function to evaluate the solution at any point in the interval `[t0 tf]`. The `solext` structure array always includes these fields:

Structure Field	Description
`sol.x`	Row vector of the steps chosen by the solver.
`sol.y`	Each column `sol.y(:,i)` contains the solution at time `sol.x(i)`.
`sol.solver`	Solver name.

Additionally, if you specify the `Events` option and events are detected, then `solext` also includes these fields:

Structure Field	Description
`sol.xe`	Points when events occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

See Also

`deval` | `odeget` | `odeset`

Topics

"Choose an ODE Solver"
"Summary of ODE Options"
"Troubleshoot Common ODE Problems"

Introduced before R2006a

onCleanup

Cleanup tasks upon function completion

Syntax

```
cleanupObj = onCleanup(cleanupFun)
```

Description

`cleanupObj = onCleanup(cleanupFun)` creates an object that, when destroyed, executes the function `cleanupFun`. MATLAB implicitly clears all local variables at the termination of a function, whether by normal completion, or a forced exit, such as an error, or **Ctrl+C**.

If you reference or pass `cleanupObj` outside your function, then `cleanupFun` does not run when that function terminates. Instead, it runs whenever MATLAB destroys the object.

Examples

Close a Figure After Executing Function

Save the following code in `action.m` and type `action` in the Command Window.

```
function action
disp('Display Figure')
f = figure;
cleanup = onCleanup(@()myCleanupFun(f));
pause(1)
end

function myCleanupFun(f)
disp('Close Figure')
close(f)
end
```

```
Display Figure
Close Figure
```

Switch Directories After Executing Function

Pass your own script to the `onCleanup` object so that it executes when MATLAB destroys the cleanup object.

Save the following code in `cleanup.m`.

```
cd(tempdir)
disp('You are now in the temporary folder')
```

Save the following code in `youraction.m` and type `youraction` in the Command Window.

```
function youraction
cleanup = onCleanup(@cleanup);
disp('Execute Code')
end
```

```
Execute Code
You are now in the temporary folder
```

Input Arguments

`cleanupFun` — Cleanup task
function handle

Cleanup task, specified as a handle to a function.

You can declare any number of `onCleanup` objects in a program file. However, if the cleanup tasks depend on the order of execution, then you should define only one object that calls a script or function, containing the relevant cleanup commands.

You should use an anonymous function handle to call your cleanup task. This allows you to pass arguments to your cleanup function.

Example: `@()fclose('file.m')`

Example: `@user_script`

Data Types: `function_handle`

Tips

- Avoid using nested functions during cleanup. MATLAB can clear variables used in nested functions before the cleanup function tries to read from them.
- If your program contains multiple cleanup objects, MATLAB does not guarantee the order that it destroys these objects. If the order of your cleanup functions matters, define one `onCleanup` object for all the tasks.
- If you save an `onCleanup` object, MATLAB displays a warning and does not save the `cleanupFun` cleanup task.

See Also

`clear` | `clearvars`

Topics

"Clean Up When Functions Complete"
"Object Lifecycle"
"Function Handles"
"What Are Anonymous Functions?"

Introduced in R2008a

ones

Create array of all ones

Syntax

```
X = ones
X = ones(n)
X = ones(sz1,...,szN)
X = ones(sz)

X = ones( ___ ,typename)
X = ones( ___ ,'like',p)
```

Description

`X = ones` returns the scalar `1`.

`X = ones(n)` returns an `n`-by-`n` matrix of ones.

`X = ones(sz1,...,szN)` returns an `sz1`-by-...-by-`szN` array of ones where `sz1,...,szN` indicates the size of each dimension. For example, `ones(2,3)` returns a 2-by-3 array of ones.

`X = ones(sz)` returns an array of ones where the size vector, `sz`, defines `size(X)`. For example, `ones([2,3])` returns a 2-by-3 array of ones.

`X = ones( ___ ,typename)` also specifies the data type (class) of `X` for any of the previous syntaxes. For example, `ones(5,'int8')` returns a 5-by-5 matrix of 8-bit integers.

`X = ones( ___ ,'like',p)` specifies that `X` has the same data type, sparsity, and complexity (real or complex) as the numeric variable `p`.

Examples

Square Array of Ones

Create a 4-by-4 array of ones.

```
X = ones(4)
```

```
X = 4×4

     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1
```

3-D Array of Ones

Create a 2-by-3-by-4 array of ones.

```
X = ones(2,3,4);
```

```
size(X)
```

```
ans = 1×3

     2     3     4
```

Size Defined by Existing Array

Define a 3-by-2 array A.

```
A = [1 4 ; 2 5 ; 3 6];
```

```
sz = size(A)
```

```
sz = 1×2

     3     2
```

Create an array of ones that is the same size as A

```
X = ones(sz)
```

```
X = 3×2

     1     1
     1     1
     1     1
```

Nondefault Numeric Data Type

Create a 1-by-3 vector of ones whose elements are 16-bit unsigned integers.

```
X = ones(1,3,'uint16'),
```

```
X = 1x3 uint16 row vector

   1   1   1
```

```
class(X)
```

```
ans =
'uint16'
```

Complex One

Create a scalar 1 that is not real valued, but instead is complex like an existing array.

Define a complex vector.

```
p = [1+2i 3i];
```

Create a scalar 1 that is complex like p.

```
X = ones('like',p)
```

```
X = 1.0000 + 0.0000i
```

Size and Numeric Data Type Defined by Existing Array

Define a 2-by-3 array of 8-bit unsigned integers.

```
p = uint8([1 3 5 ; 2 4 6]);
```

Create an array of ones that is the same size and data type as `p`.

```
X = ones(size(p),'like',p),
```

```
X = 2x3 uint8 matrix

   1   1   1
   1   1   1
```

```
class(X)
```

```
ans =
'uint8'
```

Input Arguments

`n` — Size of square matrix
integer value

Size of square matrix, specified as an integer value, defines the output as a square, n-by-n matrix of ones.

- If `n` is `0`, then `X` is an empty matrix.
- If `n` is negative, then it is treated as `0`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz1,...,szN` — Size of each dimension
two or more integer values

Size of each dimension, specified as two or more integer values, defines X as a sz1-by...-by-szN array.

- If the size of any dimension is 0, then X is an empty array.
- If the size of any dimension is negative, then it is treated as 0.
- If any trailing dimensions greater than 2 have a size of 1, then the output, X, does not include those dimensions.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

sz — Output size
row vector of integer values

Output size, specified as a row vector of integer values. Each element of this vector indicates the size of the corresponding dimension.

- If the size of any dimension is 0, then X is an empty array.
- If the size of any dimension is negative, then it is treated as 0.
- If any trailing dimensions greater than 2 have a size of 1, then the output, X, does not include those dimensions.

Example: sz = [2,3,4] defines X as a 2-by-3-by-4 array.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

typename — Output class
'double' (default) | 'single' | 'logical' | 'int8' | 'uint8' | ...

Output class, specified as 'double', 'single', 'logical', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'int64', or 'uint64'.

p — Prototype
variable

Prototype, specified as a variable.

Data Types: double | single | logical | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64
Complex Number Support: Yes

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Dimensions must be real, nonnegative integers.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Dimensions must be real, nonnegative integers.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `ones` in the Parallel Computing Toolbox documentation.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `ones` in the Parallel Computing Toolbox documentation.

See Also

`complex` | `eye` | `false` | `rand` | `randn` | `size` | `zeros`

Topics

"Class Support for Array-Creation Functions"
"Preallocation"

Introduced before R2006a

matlab.lang.OnOffSwitchState class

Package: `matlab.lang`

Represent on and off states with logical values

Description

`matlab.lang.OnOffSwitchState` is an enumeration class that derives from the `logical` class. Use this class to specify the data type of properties that accept values of `'on'` and `'off'` and logical `true`, `false`, `1`, or `0`.

Use this class to constrain property values to any of these values:

Class	Logical True	Logical False
Character vector	`'on'`	`'off'`
String scalar	`"on"`	`"off"`
Logical	`true`	`false`
Logical and numeric	`1`	`0`
Enumeration member	`matlab.lang.OnOffSwitchState.on`	`matlab.lang.OnOffSwitchState.off`

Because `OnOffSwitchState` derives from the `logical` class, you can use these enumeration members in logical expressions.

Enumeration Members

`off`	Logical false
`on`	Logical true

Examples

Enable Property Values to Be Logical or on/off

Create a class to represent the state of a computer whose power and monitor can be turned on and off separately. Use `OnOffSwitchState` to define the class of these properties.

```
classdef SystemState
   properties
      Power matlab.lang.OnOffSwitchState = 'off'
      Monitor matlab.lang.OnOffSwitchState = 'off'
   end
   methods
      function val = isOn(obj)
         if ~(obj.Power && obj.Monitor)
            val = matlab.lang.OnOffSwitchState.off;
         else
            val = matlab.lang.OnOffSwitchState.on;
         end
      end
   end
end
```

Create a `SystemState` object and set the property values to `'on'`.

```
a = SystemState;
a.Power = 'on';
a.Monitor = 'on';
```

Call the `isOn` method to determine the state of the system. The method returns a logical value provided by the `OnOffSwitchState` enumeration.

```
if isOn(a)
   ...% System is ready to use
end
```

More About

Specialized Concatenation Behavior

The `OnOffSwitchState` class enables you to form text expressions by concatenating enumeration members with character vectors or strings. For example, if an object

property named `Power` contains an enumeration member of the `OnOffSwitchState` class, you can form a character array with a character vector and the property value:

```
a = SystemState;
a.Power = true;
['The power switch is currently ' a.Power]
```

```
ans =

    'The power switch is currently on'
```

Normally, when forming an array by concatenating character vectors or strings with enumeration members, MATLAB attempts to convert the text to the class of the enumeration member. However, the `OnOffSwitchState` class defines specialized behavior that enables concatenation of enumeration members and text in cases where the text does not correspond to enumeration members defined by the class. In these cases, MATLAB creates an array of the same type as the text.

Here are the rules that MATLAB applies when concatenating character vectors or strings with enumeration members of the `OnOffSwitchState` class.

- If all array elements are either `OnOffSwitchState` enumeration members or character vectors, all enumeration members are converted to their `char` equivalent representation. The resulting array is of type `char`.
- If all array elements are either `OnOffSwitchState` enumeration members or strings, all enumeration members are converted to their `string` equivalent representation. The resulting array is of type `string`.
- If all array elements are either `OnOffSwitchState` enumeration members, strings, or character vectors, all enumeration members and character vectors are converted to their `string` equivalent representation. The resulting array is of type `string`.
- In all other cases, normal concatenation rules apply. For more information, see “Concatenating Objects of Different Classes”.

Testing for Equality

The `OnOffSwitchState` class supports the `eq` (`==`) and `isequal` functions for testing equality between enumeration members and char, string, logical, or numeric types having values of 1 or 0.

See Also

`char` | `logical` | `string`

Topics

"Validate Property Values"

Introduced in R2017a

open

Open file in appropriate application

Syntax

```
open name
A = open(name)
```

Description

`open name` opens the specified file or variable in the appropriate application.

You can extend the functionality of `open` by defining your own file-handling function of the form `open`*xxx*, where *xxx* is a file extension. For example, if you create a function `openlog`, then the `open` function calls `openlog` to process any files with the `.log` extension. The `open` function returns any single output defined by your function.

`A = open(name)` returns a structure if `name` is a MAT-file, or it returns a figure handle if `name` is a figure. Otherwise, `open` returns an empty array. For increased flexibility and options, use the `load` function to open MAT-files, and the `openfig` function to open figures.

Examples

Open File

Open the file `num2str.m` in the Editor. MATLAB opens the file *`matlabroot`*`\toolbox\matlab\strfun\num2str.m`. If a file called `num2str.m` exists in a folder that is above `toolbox\matlab\strfun` on the MATLAB path, then MATLAB opens that file instead.

```
open num2str.m
```

Open File Not on Path

Open a file not on the MATLAB path by including the complete file specification. If the file does not exist, MATLAB displays an error message.

```
open C:\temp\data.mat
```

Create Function to Handle an Extension

Create a function called `opentxt` to handle files with a `.txt` extension.

Create the function `opentxt`.

```
function opentxt(filename)
   [~, name, ext] = fileparts(filename);
   fprintf('You have requested file: %s\n', [name ext]);

   if exist(filename, 'file') == 2
     fprintf('Opening in MATLAB Editor: %s\n', [name ext]);
     edit(filename);
   else
      wh = which(filename);
      if ~isempty(wh)
         fprintf('Opening in MATLAB Editor: %s\n', wh);
         edit(wh);
      else
        warning('MATLAB:fileNotFound', ...
                'File was not found: %s', [name ext]);
      end
   end

end
```

Create the text file `myTestFile.txt`.

```
dlmwrite('myTestFile.txt',[1,2,3,4]);
```

Read the data from the file. The `open` function calls the function `opentxt` to open the file.

```
open('myTestFile.txt');
```

```
You have requested file: myTestFile.txt
Opening in MATLAB Editor: myTestFile.txt
```

Input Arguments

name — File or variable name
character array | string scalar

File or variable name, specified as a character array or string scalar. If `name` does not include an extension, then MATLAB searches for variables and files according to the "Function Precedence Order". If `name` is a variable, the `open` function opens it in the Variables editor. Otherwise, the `open` function performs one of these actions based on the file extension.

`.m` or `.mlx`	Open code file in MATLAB Editor.
`.mat`	Return variables in structure `A` when called with the syntax `A = open(name)`.
`.fig`	Open figure in Figure window.
`.mdl` or `.slx`	Open model in Simulink.
`.prj`	Open project in the MATLAB Compiler Deployment Tool.
`.doc*`	Open document in Microsoft Word.
`.exe`	Run executable file (only on Windows systems).
`.pdf`	Open document in Adobe® Acrobat®.
`.ppt*`	Open document in Microsoft PowerPoint.
`.xls*`	Start MATLAB Import Wizard.
`.htm` or `.html`	Open document in MATLAB browser.
`.slxc`	Open report file for the Simulink cache file.

In MATLAB Online, `open` only supports opening MAT-files, figures, code files (`.m` or `.mlx`), and HTML documents.

Data Types: `char` | `string`

See Also

`edit` | `load` | `openfig` | `openvar` | `which` | `winopen`

Introduced before R2006a

openfig

Open figure saved in FIG-file

Syntax

```
openfig(filename)
openfig(filename,copies)
openfig( ___ ,visibility)

fig = openfig( ___ )
```

Description

`openfig(filename)` opens the figure saved in the MATLAB figure file (FIG-file) called `filename`.

`openfig(filename,copies)` specifies whether to open a new copy of the figure in the case that a copy is already open. If you do not want to create a new copy, set `copies` to `'reuse'`. The `'reuse'` option brings the existing figure to the front of the screen. To open a new copy of the figure regardless of whether a copy is already open, set `copies` to `'new'` . The `'new'` option is the default behavior.

`openfig( ___ ,visibility)` specifies whether to open the figure in a visible or invisible state. To display the figure, set `visibility` to `'visible'`. If you do not want to display the figure, use the `'invisible'` setting. You can use this option with any of the input argument combinations in the previous syntaxes.

`fig = openfig( ___ )` returns the figure object. Set properties of the figure object to modify its appearance or behavior. For a list of properties, see Figure.

Examples

Open Figure Saved in MATLAB Figure File

Create a surface plot and save the figure as a MATLAB figure file. Then, close the figure.

```
surf(peaks)
savefig('MySavedPlot.fig')
close(gcf)
```

Open the saved figure.

```
openfig('MySavedPlot.fig')
```

Open Invisible Figure in Visible State

Create a surface plot and make the figure invisible. Then, save the figure as a MATLAB figure file. Close the invisible figure.

```
surf(peaks)
set(gcf,'Visible','off')
savefig('MySavedPlot.fig')
close(gcf)
```

Open the saved figure and make it visible on the screen.

```
openfig('MySavedPlot.fig','visible')
```

Input Arguments

`filename` — File name of saved figure
character vector | string

File name of saved figure, specified as a character vector or string. You do not have to specify the full file path, as long as it is on your MATLAB path. Including `.fig` in the file name is optional.

Example: `openfig('MySavedFigure.fig')`

`copies` — Control for opening multiple copies of figure
`'new'` (default) | `'reuse'`

Control for opening multiple copies of the figure, specified as one of these values:

- `'new'` — Open a new copy of the figure, even if a copy already exists on the screen.
- `'reuse'` — Open a new copy of the figure only if one does not exist. If a copy exists, then bring the existing copy to the front of the screen. If the figure is off the screen, then `'reuse'` repositions the figure so that it is completely on the screen. This option helps provide compatibility with different screen sizes and resolutions by ensuring that the figure displays on screen.

Example: `openfig('MySavedFigure.fig','reuse')`

`visibility` — Figure visibility
`'visible'` | `'invisible'`

Figure visibility, specified as one of these values:

- `'visible'` — Open the saved figure in a visible state. If the MATLAB figure file contains an invisible figure, then you can use this option to make the figure visible when it opens.
- `'invisible'` — Open the saved figure in an invisible state.

Example: `openfig('MySavedFigure.fig','invisible')`

Output Arguments

`fig` — Figure object
figure object

Figure object. Set properties of the figure to change the appearance or behavior of the opened figure. For a list of properties, see Figure.

Limitations

- Do not use `openfig` to open FIG-files created with GUIDE. Use the `guide` function instead.

See Also

`open` | `saveas` | `savefig`

Topics

"Save Figure to Reopen in MATLAB Later"

Introduced before R2006a

opengl

Control OpenGL rendering

Syntax

```
opengl info
d = opengl('data')

opengl software
opengl hardware
opengl hardwarebasic

opengl('save',pref)
```

Description

`opengl info` prints information about the OpenGL implementation currently in use by MATLAB, such as the version, vendor, and graphics features that it supports. Using this command loads OpenGL. Starting in R2019a, this syntax is no longer recommended. For more information, see “Compatibility Considerations” on page 1-10703.

`d = opengl('data')` returns the same data provided with `opengl info`, but stores it in a structure. Starting in R2019a, this syntax is no longer recommended.

`opengl software` uses a software version of OpenGL to render subsequent graphics for the current MATLAB session. This command works only on Windows systems.

`opengl hardware` uses a hardware-accelerated version of OpenGL to render subsequent graphics. If your graphics hardware does not support hardware-accelerated OpenGL, then MATLAB uses a software version instead.

`opengl hardwarebasic` uses a hardware-accelerated version of OpenGL, but disables some advanced graphics features that are unstable with certain graphics drivers. If your graphics hardware does not support hardware-accelerated OpenGL, then MATLAB uses a software version instead.

`opengl('save',pref)` sets your preferences so that future sessions of MATLAB on this computer use the preferred version of OpenGL. Specify `pref` as `'software'`, `'hardware'`, `'hardwarebasic'`, or `'none'`. This command does not affect the current session.

Note The `autoselect`, `neverselect`, `advise`, `quiet`, `verbose`, and `DriverBugWorkaround` inputs have been removed in R2014b. For more information about the behavior of these syntaxes in previous releases, see `opengl` for R2014a.

Examples

Display Information About OpenGL

Using the `opengl info` command, display information about the OpenGL implementation currently in use by MATLAB. For example, display the vendor, the version, and the supported graphics features. Also display whether MATLAB is using a hardware-accelerated implementation or a software implementation of OpenGL.

```
opengl info
```

```
                          Version: '3.3.0'
                           Vendor: 'NVIDIA Corporation'
                         Renderer: 'Quadro 400/PCIe/SSE2'
             RendererDriverVersion: '9.18.13.4192'
         RendererDriverReleaseDate: '13-Oct-2015'
                    MaxTextureSize: 8192
                            Visual: 'Visual 0x07, (RGBA 32 bits (8 8 8 8), Z depth 16 bi
                          Software: 'false'
              HardwareSupportLevel: 'full'
         SupportsGraphicsSmoothing: 1
    SupportsDepthPeelTransparency: 1
       SupportsAlignVertexCenters: 1
                        Extensions: {249x1 cell}
               MaxFrameBufferSize: 8192
```

Determine Graphics Hardware

Using the `opengl info` command, determine your graphics hardware by checking the `Vendor` and `Renderer` fields.

```
opengl info
```

```
                          Version: '3.3.0'
                           Vendor: 'NVIDIA Corporation'
                         Renderer: 'Quadro 400/PCIe/SSE2'
             RendererDriverVersion: '9.18.13.3182'
         RendererDriverReleaseDate: '11-Nov-2013'
                    MaxTextureSize: 8192
                            Visual: 'Visual 0x07, (RGBA 32 bits...'
                         Software: 'false'
              HardwareSupportLevel: 'full'
         SupportsGraphicsSmoothing: 1
     SupportsDepthPeelTransparency: 1
        SupportsAlignVertexCenters: 1
                       Extensions: {248x1 cell}
                MaxFrameBufferSize: 8192
```

Note If the returned fields contain the line `Software: 'true'`, then you are using software OpenGL and the name listed in the `Vendor` field is not your graphics hardware vendor. Instead, the `Vendor` field indicates the manufacturer of the software OpenGL implementation.

Determine Graphics Hardware When Using Software OpenGL (Windows)

Close all figures, switch to hardware OpenGL, and issue the `opengl info` command. Then, switch back to software OpenGL.

```
close all
opengl hardware
opengl info
opengl software
```

Determine Graphics Hardware When Using Software OpenGL (Linux)

Start MATLAB with the `-nosoftwareopengl` flag. Then, issue the `opengl info` command.

Use Software OpenGL for Current Session

Switch to using software OpenGL to render graphics in the current session.

```
opengl software
```

This command works only on Windows systems.

Use Software OpenGL for Future Sessions

Set your preferences so that MATLAB uses software OpenGL to render graphics in all future sessions. This command does not affect the current session.

```
opengl('save','software')
```

Input Arguments

`info` — Information about OpenGL implementation

`info`

Information about the OpenGL implementation currently in use by MATLAB, specified as `info`. The `opengl info` command returns the fields listed in this table.

Field	Description
`Version`	Version of the OpenGL implementation.
`Vendor`	Manufacturer of the OpenGL implementation.
`RendererDriverVersion`	Version of the OpenGL driver (Windows systems with hardware-accelerated OpenGL only). This field does not display on virtual machines.

Field	Description
`RendererDriverReleaseDate`	Release date of the OpenGL driver (Windows systems with hardware-accelerated OpenGL only). This field does not display on virtual machines.
`Renderer`	Description of the OpenGL renderer. If you are using hardware-accelerated OpenGL, this field is the graphics card model name.
`MaxTextureSize`	Maximum texture size supported by the OpenGL implementation.
`Visual`	Display properties of the OpenGL implementation.
`Software`	Software OpenGL enabled, returned as `'true'` or `'false'`.
`HardwareSupportLevel`	Hardware support level, returned as one of these values: • `'full'` — Hardware-accelerated OpenGL (all graphics features enabled) • `'basic'` — Basic hardware-accelerated OpenGL (some graphics features disabled) • `'none'` — Software OpenGL If MATLAB detects an unsupported driver, this field also contains `'known graphics driver issues'`.
`SupportsGraphicsSmoothing`	Graphics smoothing feature support, returned as `1` if supported or `0` otherwise.
`SupportsDepthPeelTransparency`	Depth peel transparency feature support, returned as `1` if supported or `0` otherwise.
`SupportsAlignVertexCenters`	Align vertex centers feature support, returned as `1` if supported or `0` otherwise.
`Extensions`	Extended capabilities supported by the OpenGL implementation.

Field	Description
`MaxFrameBufferSize`	Maximum frame buffer size supported by the OpenGL implementation.

For more information about the graphics smoothing, depth peel transparency, and align vertex centers features, see “Advanced Graphics Features” on page 1-10699.

`software` — Software OpenGL
`software`

Software OpenGL, specified as `software`.

To switch to software OpenGL:

- On Windows systems, execute `opengl software`.
- On Linux systems, start MATLAB with the `-softwareopengl` flag.
- Macintosh systems do not support software OpenGL.

Software OpenGL can be slower than hardware-accelerated OpenGL and does not support all graphics features. For a table of supported features, see “Advanced Graphics Features” on page 1-10699.

`hardware` — Hardware-accelerated OpenGL
`hardware`

Hardware-accelerated OpenGL, specified as `hardware`. All systems support using the `opengl hardware` command to switch from basic hardware to hardware OpenGL. However, only Windows systems support using the `opengl hardware` command to switch from software to hardware OpenGL. To switch from software to hardware on Linux systems, start MATLAB with the `-nosoftwareopengl` flag.

If your system automatically switched to using software OpenGL, then forcing your system to use hardware OpenGL can cause instabilities.

`hardwarebasic` — Basic version of hardware-accelerated OpenGL
`hardwarebasic`

Basic version of hardware-accelerated OpenGL, specified as `hardwarebasic`. This version of hardware-accelerated OpenGL uses your graphics hardware, but disables graphics features that are unstable with some graphics drivers. The disabled features might change in future releases as graphics features change and graphics hardware

evolves. For a table of disabled features, see “Advanced Graphics Features” on page 1-10699.

pref — OpenGL version preference for future sessions
`'software'` | `'hardware'` | `'hardwarebasic'` | `'none'`

OpenGL version preference for future sessions, specified as one of these options:

- `'software'` — Software OpenGL. This option is not available on Macintosh systems.
- `'hardware'` — Hardware-accelerated OpenGL.
- `'hardwarebasic'` — Hardware-accelerated OpenGL with some advanced graphics features disabled. For more information, see “Advanced Graphics Features” on page 1-10699.
- `'none'` — Default value for your system.

More About

Advanced Graphics Features

Advanced graphics features are features that require certain implementations of OpenGL. These features are graphics smoothing on page 1-10700, depth peel transparency on page 1-10701, align vertex centers on page 1-10702, and hardware-accelerated markers on page 1-10702. Support for these features depends on:

- Whether you are using hardware, basic hardware, or software OpenGL. To determine which implementation you are using, check the `HardwareSupportLevel` field returned by `opengl info`.
- The version of the OpenGL implementation. To determine the version in use, check the `Version` field returned by `opengl info`. To get the latest version available for your graphics hardware, upgrade your graphics drivers from your computer manufacturer website. For more information on upgrading graphics drivers, see “System Requirements for Graphics”.

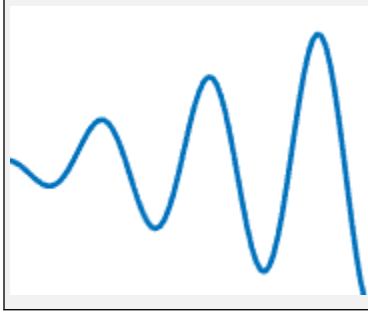
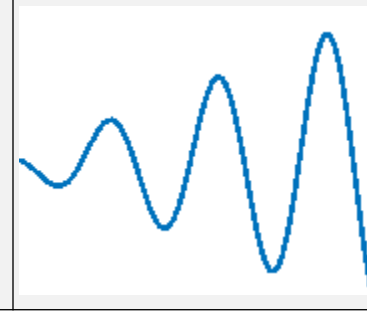
This table lists the advanced graphics features and the circumstances under which they are supported.

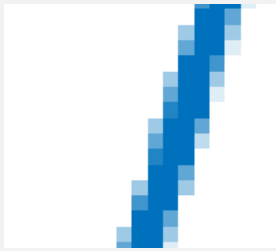
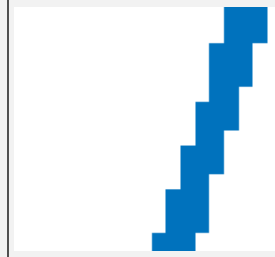
Graphics Feature	Hardware OpenGL	Basic Hardware OpenGL	Software OpenGL on Windows	Software OpenGL on Linux
Graphics Smoothing	Supported for OpenGL 2.1 or higher	Supported for OpenGL 2.1 or higher	Not supported	Not supported
Depth Peel Transparency	Supported for OpenGL 2.1 or higher	Disabled	Not supported	Supported
Align Vertex Centers	Supported for OpenGL 2.1 or higher	Disabled	Not supported	Not supported
Hardware-accelerated markers	Supported for OpenGL 4.0 or higher	Disabled	Not supported	Not supported

Graphics Smoothing

Graphics smoothing is a technique to improve the appearance of plots by reducing the appearance of jagged lines. By default, this feature is enabled if your system supports it.

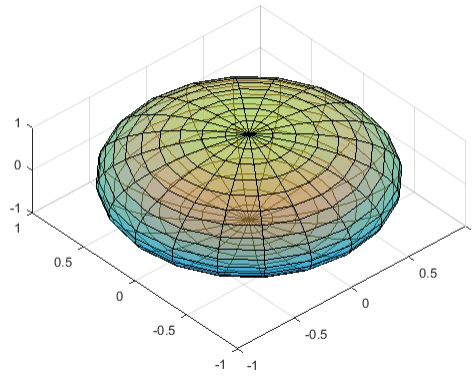
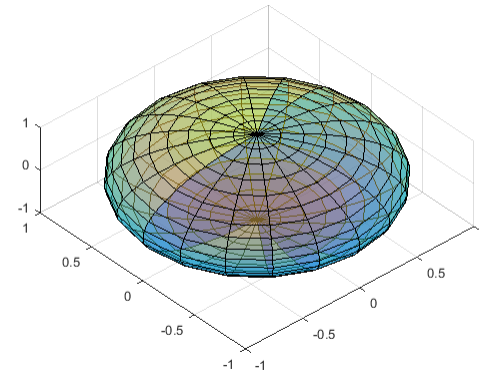
This table shows the difference when the feature is enabled or disabled. To turn off this feature for a particular figure, set the `GraphicsSmoothing` property of the figure to `'off'`.

When Supported and Enabled	When Not Supported or Disabled
	

When Supported and Enabled	When Not Supported or Disabled
(Zoomed-in view) 	(Zoomed-in view) 

Depth Peel Transparency

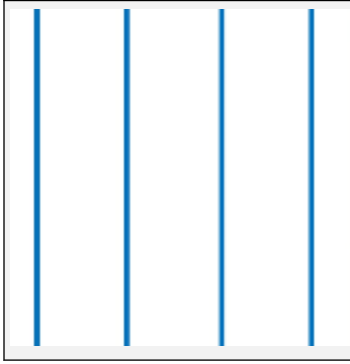
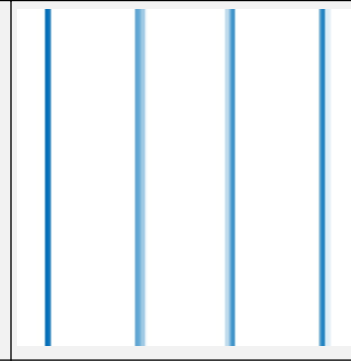
Depth peel transparency is a feature for correctly drawing transparent 3-D objects or complex plots that contain intersecting transparent objects. In the table, the left image shows the result of using transparency on a sphere when the depth peel transparency feature is supported. The right image shows the same sphere with unexpected shaded areas that occur when the feature is not supported.

When Supported	When Not Supported
	

Align Vertex Centers

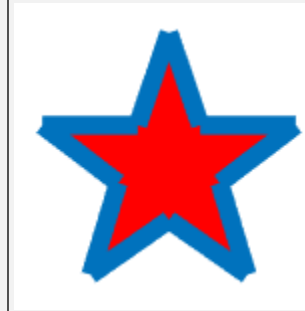
Align vertex centers is a feature for sharp vertical and horizontal lines. If graphics smoothing is enabled, it can cause horizontal and vertical lines to appear uneven in thickness or color. The align vertex centers feature eliminates the uneven appearance. By default, the align vertex centers feature is not enabled. However, if your system supports this feature, then you can turn it on for objects that have an `AlignVertexCenters` property by setting the property to `'on'`.

This table shows the difference when the feature is enabled or disabled.

When Supported and Enabled	When Not Supported or Disabled
	

Hardware-Accelerated Markers

Hardware-accelerated markers take advantage of your graphics hardware for improved performance and quality. This table shows the difference when the feature is supported or not supported.

When Supported	When Not Supported
	

Tips

- Painters is an alternate rendering method for screen display and printing. For more information, see the figure's `Renderer` property.
- By default, MATLAB uses hardware-accelerated OpenGL if your graphics hardware supports it. However, in some cases, MATLAB automatically switches to software OpenGL, for example, if it detects:
  - You are using a graphics driver with known issues or graphics virtualization.
  - A previous MATLAB session crashed due to a graphics issue. If the previous session was using software OpenGL and crashed, then subsequent sessions use a more stable version of software OpenGL that has fewer capabilities.
  - You do not have graphics hardware or your graphics hardware does not support hardware OpenGL.

Compatibility Considerations

Using the `opengl` function to get information about the graphics renderer is not recommended

Not recommended starting in R2019a

Using the `opengl` function to get information about the graphics renderer is not recommended. Specifically, these syntaxes are not recommended:

- `opengl info`
- `d = openg('data')`

There are no plans to remove support for these syntaxes at this time. Instead of calling `opengl` to get the renderer information, call the `rendererinfo` function instead:

```
info = rendererinfo(ax)
```

Specify `ax` as any type of axes or a chart that can be a child of a figure (such as a heatmap). The output is a structure containing most of the same information as the `opengl` function provides.

Fields in opengl Structure	Corresponding Fields in rendererinfo Structure
`d.Version`	`info.Version`
`d.Vendor`	`info.Vendor`
`d.Renderer`	`info.RendererDevice`
`d.RendererDriverVersion`	`info.Details.RendererDriverVersion`
`d.RendererDriverReleaseDate`	`info.Details.RendererDriverReleaseDate`
`d.MaxTextureSize`	`info.Details.MaxTextureSize`
`d.Visual`	No longer needed
`d.Software`	This information is stored in `info.GraphicsRenderer`, but to get the equivalent logical value, use `strcmp(info.GraphicsRenderer,'OpenGL Software')`
`d.HardwareSupportLevel`	`info.Details.HardwareSupportLevel`
`d.SupportsGraphicsSmoothing`	`info.Details.SupportsGraphicsSmoothing`
`d.SupportsDepthPeelTransparency`	`info.Details.SupportsDepthPeelTransparency`
`d.SupportsAlignVertexCenters`	`info.Details.SupportsAlignVertexCenters`

Fields in opengl Structure	Corresponding Fields in rendererinfo Structure
`d.Extensions`	No longer needed
`d.MaxFrameBufferSize`	`info.Details.MaxFrameBufferSize`

See Also

`matlab (Linux)` | `matlab (Windows)` | `matlab (macOS)` | `rendererinfo`

Topics

"System Requirements for Graphics"
"Resolving Low-Level Graphics Issues"

Introduced before R2006a

openProject

Package: `matlab.project`

Load an existing project

Syntax

```
proj = openProject(projectPath)
```

Description

`proj = openProject(projectPath)` loads the project in the specified file or folder. If any projects are currently open, MATLAB closes them before loading the specified project.

Examples

Open a Project

Open an existing project from a folder called `"C:/projects/project1/"`.

```
proj = openProject("C:/projects/project1/")
```

Input Arguments

`projectPath` — Full path to project file or folder
character vector | string scalar

Full path to the project `.prj` file or project root folder, specified as a character vector or string scalar.

Example: `"C:/projects/project1/myProject.prj"`

Example: `"C:/projects/project1/"`

Output Arguments

`proj` — Project
`matlab.project.Project` object

Project, returned as a `matlab.project.Project` object. Use the `matlab.project.Project` object to programmatically manipulate the currently open project.

See Also

`currentProject`

Topics

"Create and Edit Projects Programmatically"

Introduced in R2019a

openvar

Open workspace variable in Variables editor or other graphical editing tool

Syntax

`openvar(varname)`

Description

`openvar(varname)` opens the workspace variable `varname`, in the Variables editor for graphical editing. Changes that you make to variables in the Variables editor occur in the workspace as soon as you enter them.

In some toolboxes, `openvar` opens a tool appropriate for viewing or editing objects indicated by `varname` instead of opening the Variables editor.

MATLAB does not impose any limitation on the size of a variable that you can open in the Variables editor. However, your operating system or the amount of physical memory installed on your computer can impose such limits.

Examples

Identify Outliers in a Linked Graph

Use data brushing to identify observations in a vector or matrix that might warrant further analysis.

Make a scatter plot of data in the sample MAT-file `count.dat`, and open the variable `count` in the Variables editor.

```
load count.dat
scatter(count(:,1),count(:,2))
openvar('count')
```

Right-click a cell in the Variables editor and select **Brushing > Brushing On**. This turns on data brushing in the Variables editor.

Select the rows 7, 8, and 20. (Select noncontiguous rows by holding down the **Ctrl** key and clicking in each row.)

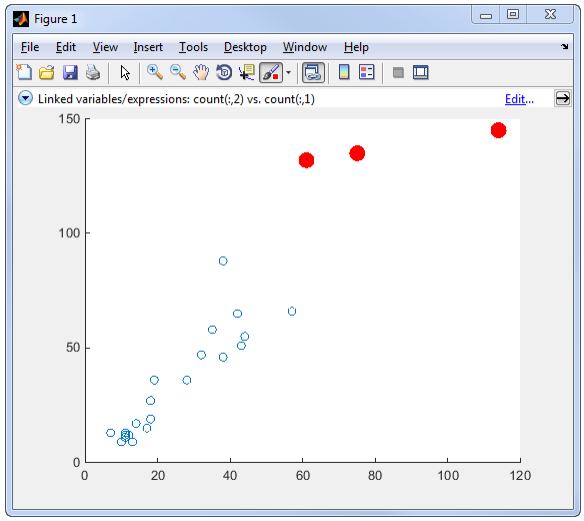
count

24x3 double

	1	2	3	4	5
1	11	11	9		
2	7	13	11		
3	14	17	20		
4	11	13	9		
5	43	51	69		
6	38	46	76		
7	61	132	186		
8	75	135	180		
9	38	88	115		
10	28	36	55		
11	12	12	14		
12	18	27	30		
13	18	19	29		
14	17	15	18		
15	19	36	48		
16	32	47	10		
17	42	65	92		
18	57	66	151		
19	44	55	90		
20	114	145	257		
21	35	58	68		
22	11	12	15		
23	13	9	15		
24	10	9	7		
25					

In the Figure window with the scatter plot, click **Brush/Select Data** to enable data brushing, and **Link Plot** to enable data linking.

The data observations you brushed in the Variables editor appear highlighted in the scatter plot.



As long as data linking is enabled in the figure, observations that you brush in the scatter plot are highlighted in the Variables editor. When a figure is not linked to its data sources, you can still brush its graphs and you can brush the same data in the Variables editor, but only the display that you brush responds by highlighting.

Input Arguments

`varname` — Variable name

character vector

Variable name, specified as a character vector. The named variable can be an array, character vector, cell array, structure, or an object and its properties. If the named variable is a multidimensional array, then you can only view the array in the Variables editor, and not edit it.

Example: `'myVariable'`

Example: `'A'`

Tips

- As an alternative to the `openvar` function, double-click a variable in the Workspace browser.

See Also

Workspace Browser | `brush` | `linkdata` | `load` | `save`

Topics

"Create and Edit Variables"
"Automatically Refresh Plot After Changing Data"

Introduced before R2006a

OperationResult

Operation result object

Description

An `OperationResult` object represents the status of an individual operation when upgrading the personal settings of a toolbox.

Creation

Create a `ReleaseCompatibilityResults` object for a specific toolbox version number by using the `matlab.settings.loadSettingsCompatibilityResults` function. Access the `Results` property and the resulting object `VersionChanges` property to get an array of `OperationResult` objects.

For example, this code gets the array of `OperationResult` objects for version 2 of the toolbox `mytoolbox`.

```
myCompatibilityResults = matlab.settings.loadSettingsCompatibilityResults('mytoolbox','Version2');
myCompatibilityResults.Results.VersionChanges
```

```
ans =
  1×2 OperationResult array with properties:
    Operation
    Status
    ExceptionLog
```

Properties

`Operation` — Upgrade operation performed

string scalar

Upgrade operation performed, specified as a string scalar.

Example: `"move mytoolbox.font.MyFontSize mytoolbox.font.FontSize"`

`Status` — Status of upgrade operation
`"Succeeded"` | `"Skipped"` | `"Failed"`

Status of upgrade operation, specified as `"Succeeded"`, `"Skipped"`, or `"Failed"`. This table describes each status and possible causes.

Status	Cause	
`"Succeeded"`	N/A	
`"Skipped"`	Setting or setting group specified in the operation does not exist in the personal settings to be upgraded.	
`"Failed"`	Move operations	• Upgraded settings tree already contains a setting or group with the specified name. • Specified setting or settings group path is not a valid path.
	Remove operations	• Specified setting or settings group path is not a valid path.

`ExceptionLog` — Log of exceptions
array of release compatibility exception objects

Log of exceptions that occurred while performing the upgrade operation, specified as an array of release compatibility exception objects. If no exception occurs, MATLAB returns a `0`-by-`0` array.

Examples

Get Results of Upgrading Personal Settings

Create functions to create and then upgrade a toolbox factory tree and then test that the upgrade completes successfully.

Create the function `createMyToolboxFactoryTree` that creates the factory settings tree for the toolbox `mytoolbox`.

```
function myToolboxFactoryTree = createMyToolboxFactoryTree()
    myToolboxFactoryTree = matlab.settings.FactoryGroup.createToolboxGroup('mytoolbox', ...
        'Hidden',false);

    toolboxFontGroup = addGroup(myToolboxFactoryTree,'font','Hidden',false)
    addSetting(toolboxFontGroup,'MyFontSize','FactoryValue',11,'Hidden',false, ...
        'ValidationFcn',@matlab.settings.mustBeNumericScalar)
    addSetting(toolboxFontGroup,'MyFontColor','FactoryValue','Black', ...
        'Hidden',false,'ValidationFcn',@matlab.settings.mustBeStringScalar);
end
```

Create the function `createMyToolboxSettingsFileUpgraders` with an empty settings file upgrader object.

```
function upgraders = createMyToolboxSettingsFileUpgraders()
    upgraders = matlab.settings.SettingsFileUpgrader.empty;
end
```

Create the `settingsInfo.json` file for the toolbox. Specify `mytoolbox` as the root settings group name, `createMyToolboxFactoryTree` as the settings tree creation function, and `createMyToolboxSettingsFileUpgraders` as the settings tree upgrade function. Place `settingsInfo.json` in the toolbox `resources` folder.

```
{
"ToolboxGroupName" : "mytoolbox",
"Hidden" : "false",
"CreateTreeFcn" : "createMyToolboxFactoryTree",
"CreateUpgradersFcn" : "createMyToolboxSettingsFileUpgraders"
}
```

Add the folder that contains the settings tree creation function and the toolbox resources folder to the MATLAB path. Then, load the factory settings tree for `mytoolbox`.

```
matlab.settings.reloadFactoryFile('mytoolbox');
```

Use the `settings` function to access the root of the settings tree and set the personal value for the `MyFontSize` setting.

```
s = settings;
s.mytoolbox.font.MyFontSize.PersonalValue = 15;
```

Change the settings names in `createMyToolboxFactoryTree` to be `FontSize` and `FontColor`.

```
function myToolboxFactoryTree = createMyToolboxFactoryTree()
    myToolboxFactoryTree = matlab.settings.FactoryGroup.createToolboxGroup('mytoolbox', ...
        'Hidden',false);

    toolboxFontGroup = addGroup(myToolboxFactoryTree,'font','Hidden',false)
    addSetting(toolboxFontGroup,'FontSize','FactoryValue',11,'Hidden',false, ...
        'ValidationFcn',@matlab.settings.mustBeNumericScalar)
```

```
    addSetting(toolboxFontGroup,'FontColor','FactoryValue','Black', ...
        'Hidden',false,'ValidationFcn',@matlab.settings.mustBeStringScalar);
end
```

Record the rename of the two settings in the `createMyToolboxSettingsFileUpgraders` function as changes to the settings tree for version 2 of `mytoolbox`.

```
function upgraders = createMyToolboxSettingsFileUpgraders()
    upgraders = matlab.settings.SettingsFileUpgrader('Version2');
    upgraders.move('mytoolbox.font.MyFontSize','mytoolbox.font.FontSize');
    upgraders.move('mytoolbox.font.MyFontColor','mytoolbox.font.FontColor');
end
```

Reload the factory settings tree for `mytoolbox`.

```
matlab.settings.reloadFactoryFile('mytoolbox');
```

Use the `settings` function to access the root of the settings tree and verify that the personal value for the `FontSize` setting was correctly moved from the `MyFontSize` setting.

```
s = settings;
s.mytoolbox.font.FontSize
```

```
ans = 
  Setting 'mytoolbox.font.FontSize' with properties:
       ActiveValue: 15
     TemporaryValue: <no value>
      PersonalValue: 15
       FactoryValue: 11
```

Get the result of the first upgrade operation for version 2 of `mytoolbox`.

```
compatibilityResults.matlab.settings.loadSettingsCompatibilityResults("mytoolbox","Version2");
compatibilityResults.Results.VersionChanges(1)
```

```
ans = 
  OperationResult with properties:
       Operation: "move mytoolbox.font.MyFontSize mytoolbox.font.FontSize"
          Status: "Succeeded"
    ExceptionLog: [0×0 matlab.settings.ReleaseCompatibilityException]
```

See Also

`VersionResults` | `matlab.settings.loadSettingsCompatibilityResults`

Topics

“Create Factory Settings for Toolboxes”

"Access and Modify Settings"

Introduced in R2019b

optimget

Optimization options values

Syntax

```
val = optimget(options,'param')
val = optimget(options,'param',default)
```

Description

`val = optimget(options,'param')` returns the value of the specified parameter in the optimization options structure `options`. You need to type only enough leading characters to define the parameter name uniquely. Case is ignored for parameter names.

`val = optimget(options,'param',default)` returns `default` if the specified parameter is not defined in the optimization options structure `options`. Note that this form of the function is used primarily by other optimization functions.

Examples

This statement returns the value of the `Display` optimization options parameter in the structure called `my_options`.

```
val = optimget(my_options,'Display')
```

This statement returns the value of the `Display` optimization options parameter in the structure called `my_options` (as in the previous example) except that if the `Display` parameter is not defined, it returns the value `'final'`.

```
optnew = optimget(my_options,'Display','final');
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Input parameter names must be constant.

See Also

`fminbnd` | `fminsearch` | `fzero` | `lsqnonneg` | `optimset`

Introduced before R2006a

optimset

Create or modify optimization options structure

Syntax

```
options = optimset(Name,Value)
optimset
options = optimset
options = optimset(optimfun)
options = optimset(oldopts,Name,Value)
options = optimset(oldopts,newopts)
```

Description

`options = optimset(Name,Value)` returns `options` with specified parameters set using one or more name-value pair arguments.

`optimset` (with no input or output arguments) displays a complete list of parameters with their valid values.

`options = optimset` (with no input arguments) creates an options structure `options` where all parameters are set to `[]`.

`options = optimset(optimfun)` creates `options` with all parameter names and default values relevant to the optimization function `optimfun`.

`options = optimset(oldopts,Name,Value)` creates a copy of `oldopts` and modifies the specified parameters using one or more name-value pair arguments.

`options = optimset(oldopts,newopts)` combines an existing options structure `oldopts` with a new options structure `newopts`. Any parameters in `newopts` with nonempty values overwrite the corresponding parameters in `oldopts`.

Examples

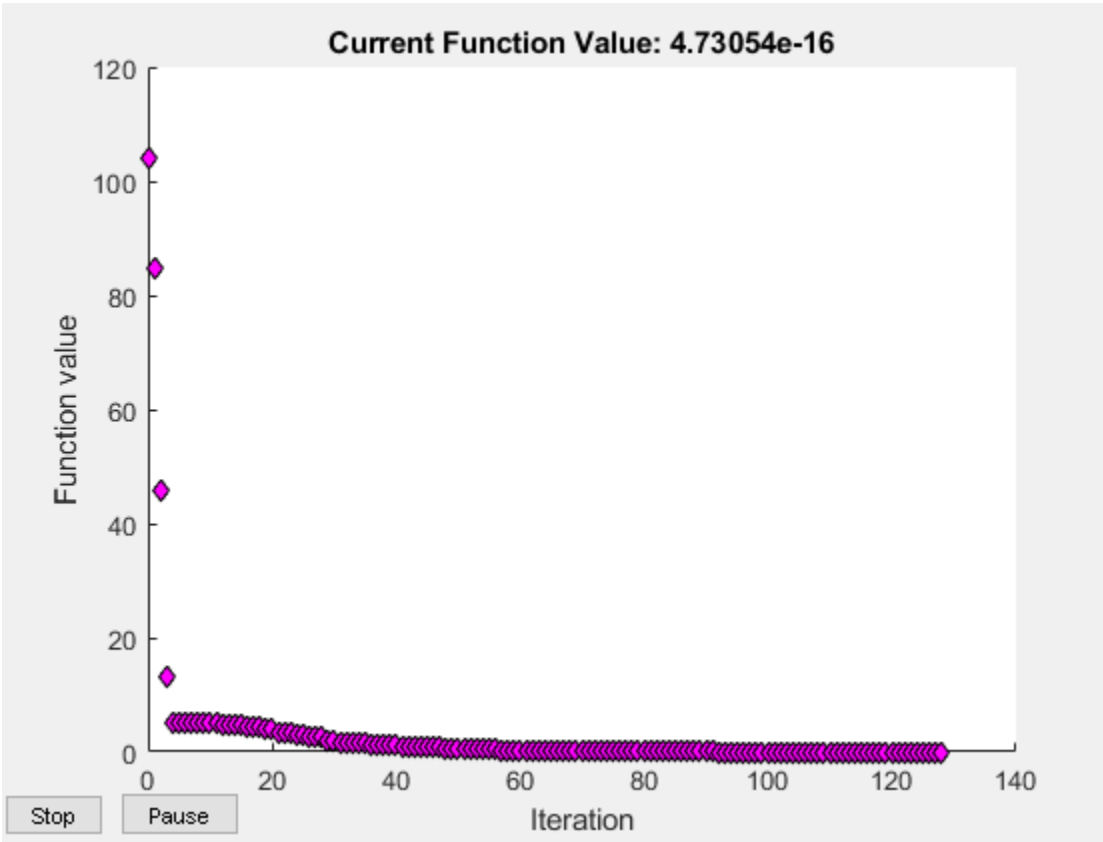
Create Nondefault Options

Set options for `fminsearch` to use a plot function and a stricter stopping condition than the default.

```
options = optimset('PlotFcns','optimplotfval','TolX',1e-7);
```

Minimize Rosenbrock's function starting from the point (–1,2), and monitor the minimization process by using the options. Rosenbrock's function has a minimum value of 0 at the point (1,1).

```
fun = @(x)100*((x(2) - x(1)^2)^2) + (1 - x(1))^2; % Rosenbrock's function
x0 = [-1,2];
[x,fval] = fminsearch(fun,x0,options)
```



```
x = 1×2

    1.0000    1.0000

fval = 4.7305e-16
```

Create Default Options for Solver

Create a structure containing the default options for the `fzero` solver.

```
options = optimset('fzero');
```

View the default value of the `TolX` option for `fzero`.

```
tol = options.TolX
```

```
tol = 2.2204e-16
```

Modify Options

Set options to use a function tolerance of `1e-6`.

```
oldopts = optimset('TolFun',1e-6);
```

Modify options in `oldopts` to use the `'optimplotfval'` plot function and a `TolX` value of `1e-6`.

```
options = optimset(oldopts,'PlotFcns','optimplotfval','TolX',1e-6);
```

View the three options that you set.

```
disp(options.TolFun);
```

```
   1.0000e-06
```

```
disp(options.PlotFcns);
```

```
optimplotfval
```

```
disp(options.TolX);
```

```
1.0000e-06
```

Update Options Structure Using New Options Structure

Overwrite the corresponding parts of one options structure with a different options structure by using `optimset`.

```
oldopts = optimset('Display','iter','TolX',1e-6);
newopts = optimset('PlotFcns','optimplotfval','Display','off');
options = optimset(oldopts,newopts);
```

Both `oldopts` and `newopts` set the value of the `Display` option. Check that `newopts` overwrites `oldopts` for this option.

```
options.Display
```

```
ans =
'off'
```

Check the values of the other two options.

```
options.TolX
```

```
ans = 1.0000e-06
```

```
options.PlotFcns
```

```
ans =
'optimplotfval'
```

Input Arguments

`optimfun` — Optimization solver
name | function handle

Optimization solver, specified as a name or function handle. The returned options structure has nonempty entries for the specified solver only.

Example: `options = optimset('fzero')`

Example: `options = optimset(@fminsearch)`

Data Types: `char` | `string` | `function_handle`

`oldopts` — Previous optimization options
structure

Previous optimization options, specified as a structure. The output `options` is the same as `oldopts`, except for the specified parameters.

Example: `options = optimset(oldopts,'TolX',1e-6)`

Data Types: `struct`

`newopts` — New optimization options
structure

New optimization options, specified as a structure. The output `options` is the same as `newopts`, and also includes nonempty parameters of `oldopts` that are empty in `newopts`.

Example: `options = optimset(oldopts,newopts)`

Data Types: `struct`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `options = optimset('TolX',1e-6,'PlotFcns',@optimplotfval)`

`Display` — Level of display
`'notify'` (default) | `'final'` | `'off'` | `'none'` | `'iter'`

Level of display, specified as the comma-separated pair consisting of 'Display' and one of these values:

- `'notify'` — Display output only if the function does not converge.
- `'final'` — Display just the final output.
- `'off'` or `'none'` — Display no output.
- `'iter'` — Display output at each iteration (not available for `lsqnonneg`).

`Display` is available for all optimization solvers.

Example: `options = optimset('Display','iter')`

Data Types: `char` | `string`

`FunValCheck` — Check whether function values are valid
`'off'` (default) | `'on'`

Flag to check whether function values are valid, specified as the comma-separated pair consisting of `'FunValCheck'` and the value `'off'` or `'on'`. When the value is `'on'`, solvers display an error when the objective function returns a value that is complex or `NaN`.

`FunValCheck` is available for `fminbnd`, `fminsearch`, and `fzero`.

Example: `options = optimset('FunValCheck','on')`

Data Types: `char` | `string`

`MaxFunEvals` — Maximum number of function evaluations
`500` for `fminbnd`, `200*(number of variables)` for `fminsearch` (default) | positive integer

Maximum number of function evaluations, specified as the comma-separated pair consisting of `'MaxFunEvals'` and a positive integer.

`MaxFunEvals` is available for `fminbnd` and `fminsearch`.

Example: `options = optimset('MaxFunEvals',2e3)`

Data Types: `single` | `double`

`MaxIter` — Maximum number of iterations
`500` for `fminbnd`, `200*(number of variables)` for `fminsearch` (default) | positive integer

Maximum number of iterations, specified as the comma-separated pair consisting of `'MaxIter'` and a positive integer.

`MaxIter` is available for `fminbnd` and `fminsearch`.

Example: `options = optimset('MaxIter',2e3)`

Data Types: `single` | `double`

`OutputFcn` — Output function
`[]` (default) | function name | function handle | cell array of function handles

Output function, specified as the comma-separated pair consisting of `'OutputFcn'` and a function name or function handle. Specify multiple output functions as a cell array of function handles. An output function runs after each iteration, enabling you to monitor the solution process or stop the iterations. For more information, see “Output Functions”.

`OutputFcn` is available for `fminbnd`, `fminsearch`, and `fzero`.

Example: `options = optimset('OutputFcn',{@outfun1,@outfun2})`

Data Types: `char` | `string` | `cell` | `function_handle`

`PlotFcns` — Plot functions
`[]` (default) | function name | function handle | cell array of function handles

Plot functions, specified as the comma-separated pair consisting of `'PlotFcns'` and a function name or function handle. Specify multiple plot functions as a cell array of function handles. A plot function runs after each iteration, enabling you to monitor the solution process or stop the iterations. For more information, see “Plot Functions”.

The built-in plot functions are as follows:

- `@optimplotx` plots the current point.
- `@optimplotfval` plots the function value.
- `@optimplotfunccount` plots the function count (not available for `fzero`).

The `PlotFcns` option is available for `fminbnd`, `fminsearch`, and `fzero`.

Example: `options = optimset('PlotFcns','optimplotfval')`

Data Types: `char` | `string` | `cell` | `function_handle`

`TolFun` — Termination tolerance on function value
`1e-4` (default) | nonnegative scalar

Termination tolerance on the function value, specified as the comma-separated pair consisting of `'TolFun'` and a nonnegative scalar. Iterations end when the current function value differs from the previous value by less than `TolFun`, relative to the initial function value. See “Tolerances and Stopping Criteria”.

`TolFun` is available for `fminsearch` only.

Example: `options = optimset('TolFun',2e-6)`

Data Types: `single` | `double`

`TolX` — Termination tolerance on `x`, the current point
`1e-4` for `fminbnd` and `fminsearch`, `eps` for `fzero`, `10*eps*norm(c,1)*length(c)` for `lsqnonneg` (default) | nonnegative scalar

Termination tolerance on `x`, the current point, specified as the comma-separated pair consisting of `'TolX'` and a nonnegative scalar. Iterations end when the current point differs from the previous point by less than `TolX`, relative to the size of `x`. See "Tolerances and Stopping Criteria".

`TolX` is available for all solvers.

Example: `options = optimset('TolFun',2e-6)`

Data Types: `single` | `double`

Output Arguments

`options` — Optimization options
structure

Optimization options, returned as a structure. Values for parameters you do not set are `[]`, which causes solvers to use the default values of these parameters.

Limitations

- `optimset` sets options for the four MATLAB optimization solvers: `fminbnd`, `fminsearch`, `fzero`, and `lsqnonneg`. To set options for Optimization Toolbox™ or Global Optimization Toolbox solvers, the recommended function is `optimoptions`.
- `optimset` cannot set options for some Optimization Toolbox solvers, such as `intlinprog`. Use `optimoptions` instead.
- `optimset` cannot set most options for Global Optimization Toolbox solvers. Use `optimoptions` instead.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support the syntax that has no input or output arguments:

  `optimset`
- Functions specified in the `options` must be supported for code generation.
- The input argument `optimfun` must be a function that is supported for code generation.
- The fields of the options structure `oldopts` must be fixed-size fields.
- Code generation ignores the `Display` option.
- Code generation does not support the additional options in an options structure created by the Optimization Toolbox `optimset` function. If an input options structure includes the additional Optimization Toolbox options, then the output structure does not include them.

See Also

`fminbnd` | `fminsearch` | `fzero` | `lsqnonneg` | `optimget` | `optimoptions`

Topics

"Set Options"
"Plot Functions"
"Optimization"

Introduced before R2006a

or, |

Find logical OR

Syntax

```
A | B
or(A,B)
```

Description

`A | B` performs a logical OR of arrays `A` and `B` and returns an array containing elements set to either logical 1 (`true`) or logical 0 (`false`). An element of the output array is set to logical 1 (`true`) if either `A` or `B` contain a nonzero element at that same array location. Otherwise, the array element is set to 0.

`or(A,B)` is an alternate way to execute `A | B`, but is rarely used. It enables operator overloading for classes.

Examples

Locate Zeros in Matrices

Find the logical OR of two matrices. The result contains logical `1` (`true`) where either matrix contains a nonzero value. The zeros in the result indicate spots where both arrays have a value of zero.

```
A = [5 7 0; 0 2 9; 5 0 0]
```

```
A = 3×3

     5     7     0
     0     2     9
     5     0     0
```

```
B = [6 6 0; 1 3 5; -1 0 0]
```

```
B = 3×3

     6     6     0
     1     3     5
    -1     0     0
```

```
A | B
```

```
ans = 3x3 logical array

   1   1   0
   1   1   1
   1   0   0
```

Truth Table for Logical OR

Create a truth table for `or`.

```
A = [true false]
```

```
A = 1x2 logical array

   1   0
```

```
B = [true; false]
```

```
B = 2x1 logical array

   1
   0
```

```
C = A|B
```

```
C = 2x2 logical array

   1   1
   1   0
```

Input Arguments

A, B — Operands
scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs `A` and `B` must either be the same size or have sizes that are compatible (for example, `A` is an `M`-by-`N` matrix and `B` is a scalar or `1`-by-`N` row vector). For more information, see "Compatible Array Sizes for Basic Operations".

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Tips

- You can chain together several logical operations, for example, `A & B | C`.
- The symbols `|` and `||` perform different operations in MATLAB. The element-wise OR operator described here is `|`. The short-circuit OR operator is `||`.
- When you use the element-wise & and `|` operators in the context of an `if` or `while` loop expression (and only in that context), they use short-circuiting to evaluate expressions. Otherwise, you must specify && or `||` to opt-in to short-circuiting behavior. See `Logical Operators: Short Circuit` for more information.

Compatibility Considerations

Implicit expansion change affects arguments for operators

Behavior changed in R2016b

Starting in R2016b with the addition of implicit expansion, some combinations of arguments for basic operations that previously returned errors now produce results. For example, you previously could not add a row and a column vector, but those operands are now valid for addition. In other words, an expression like `[1 2] + [1; 2]` previously returned a size mismatch error, but now it executes.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a `try/catch` block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see “Compatible Array Sizes for Basic Operations”.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`all` | `and` | `any` | `bitor` | `not` | `xor`

Topics

"MATLAB Operators and Special Characters"

Introduced before R2006a

ordeig

Eigenvalues of quasitriangular matrices

Syntax

```
E = ordeig(T)
E = ordeig(AA,BB)
```

Description

`E = ordeig(T)` takes a quasitriangular Schur matrix `T`, typically produced by `schur`, and returns the vector `E` of eigenvalues in their order of appearance down the diagonal of T.

`E = ordeig(AA,BB)` takes a quasitriangular matrix pair `AA` and `BB`, typically produced by `qz`, and returns the generalized eigenvalues in their order of appearance down the diagonal of `AA-λ*BB`.

`ordeig` is an order-preserving version of `eig` for use with `ordschur` and `ordqz`. It is also faster than `eig` for quasitriangular matrices.

Examples

Example 1

```
T=diag([1 -1 3 -5 2]);
```

`ordeig(T)` returns the eigenvalues of `T` in the same order they appear on the diagonal.

```
ordeig(T)

ans =

     1
    -1
```

```
     3
    -5
     2
```

`eig(T)`, on the other hand, returns the eigenvalues in order of increasing magnitude.

```
eig(T)

ans =

    -5
    -1
     1
     2
     3
```

Example 2

```
A = rand(10);
[U, T] = schur(A);
abs(ordeig(T))

ans =

    5.3786
    0.7564
    0.7564
    0.7802
    0.7080
    0.7080
    0.5855
    0.5855
    0.1445
    0.0812
% Move eigenvalues with magnitude < 0.5 to the
% upper-left corner of T.
[U,T] = ordschur(U,T,abs(E)<0.5);
abs(ordeig(T))

ans =

    0.1445
    0.0812
    5.3786
```

```
    0.7564
    0.7564
    0.7802
    0.7080
    0.7080
    0.5855
    0.5855
```

See Also

eig | ordqz | ordschur | qz | schur

Introduced before R2006a

orderfields

Order fields of structure array

Syntax

```
S = orderfields(S1)
S = orderfields(S1,S2)
S = orderfields(S1,C)
S = orderfields(S1,P)
[S,Pout] = orderfields( ___ )
```

Description

`S = orderfields(S1)` orders the fields in `S1` by name.

Since field names can contain only letters, digits, and underscores, this syntax sorts field names in ASCII order. All uppercase letters come before all lowercase letters.

`S = orderfields(S1,S2)` returns a copy of `S1` with its fields reordered to match the order of the fields of `S2`. The input structure arrays `S1` and `S2` must have the same field names.

`S = orderfields(S1,C)` matches the order of the names specified in the input array `C`. The name of every field in `S1` must appear once in `C`.

`S = orderfields(S1,P)` matches the order specified by the permutation vector `P`.

If `S1` has `n` fields, then the elements of `P` are the integers from `1` through `n`, arranged in any order. For example, if `S1` has three fields and `P` is `[3 1 2]`, then the third field of `S1` is the first field of the output `S`. This syntax is useful for ordering multiple structure arrays in the same way.

`[S,Pout] = orderfields( ___ )` also returns a permutation vector. The elements of `Pout` are the integers from `1` through `n`, arranged in an order that represents the change in order of the fields. You can use this syntax with any of the input arguments of the previous syntaxes.

Examples

Order Fields by Name

Create a structure with several fields.

```
S1 = struct('b',1,'B',2,'a',3,'A',4)
```

```
S1 = struct with fields:
    b: 1
    B: 2
    a: 3
    A: 4
```

Order the fields. This syntax orders the fields by their names, in ASCII order.

```
S = orderfields(S1)
```

```
S = struct with fields:
    A: 4
    B: 2
    a: 3
    b: 1
```

Order Fields Using Another Structure

Create two structures that have the same fields, in different orders. The field names are the same, but the field values are different.

```
S1 = struct('b',1,'B',2,'a',3,'A',4)
```

```
S1 = struct with fields:
    b: 1
    B: 2
    a: 3
    A: 4
```

```
S2 = struct('a',0,'b',20,'B',10,'A',0)
```

```
S2 = struct with fields:
    a: 0
    b: 20
    B: 10
    A: 0
```

Order the fields in S1 to match the order of fields in S2.

```
S = orderfields(S1,S2)
```

```
S = struct with fields:
    a: 3
    b: 1
    B: 2
    A: 4
```

List Field Names in Cell Array

Create a structure.

```
data.x = linspace(0,2*pi);
data.y = sin(data.x);
data.title = 'y = sin(x)'
```

```
data = struct with fields:
        x: [1x100 double]
        y: [1x100 double]
    title: 'y = sin(x)'
```

Order the fields by listing their names in a cell array.

```
C = {'title','x','y'};
data = orderfields(data,C)
```

```
data = struct with fields:
    title: 'y = sin(x)'
        x: [1x100 double]
        y: [1x100 double]
```

List Fields by Position Using Permutation Vector

Create a structure.

```
data.x = linspace(0,2*pi);
data.y = sin(data.x);
data.title = 'y = sin(x)'
```

```
data = struct with fields:
        x: [1x100 double]
        y: [1x100 double]
    title: 'y = sin(x)'
```

Order the fields by listing their original positions in a different order. For example, move the third field so that it is the first field of the output structure.

```
P = [3 1 2];
data = orderfields(data,P)
```

```
data = struct with fields:
    title: 'y = sin(x)'
        x: [1x100 double]
        y: [1x100 double]
```

Reorder Using Permutation Vector from Another Structure

Create a structure.

```
data1.x = linspace(0,2*pi);
data1.y = sin(data1.x);
data1.title = 'y = sin(x)';
```

Reorder the structure using the `orderfields` function. Store the new field order in a permutation vector `Pout`.

```
[S,Pout] = orderfields(data1,{'title','x','y'})
```

```
S = struct with fields:
    title: 'y = sin(x)'
```

```
        x: [1x100 double]
        y: [1x100 double]


Pout = 3×1

     3
     1
     2
```

Create a second structure with the same fields.

```
data2.x = data1.x;
data2.y = cos(data2.x);
data2.title = 'y = cos(x)';
```

Reorder the fields of `data2` using `Pout`. If you have many structures with the same field names, then you can use Pout to reorder them all in the same way.

```
S2 = orderfields(data2,Pout)
```

```
S2 = struct with fields:
    title: 'y = cos(x)'
        x: [1x100 double]
        y: [1x100 double]
```

Input Arguments

S1 — Input structure

structure array

Input structure, specified as a structure array.

S2 — Field order by structure

structure array

Field order by structure, specified as a structure array. `S2` has the same fields as `S1` but specifies them in a different order.

`C` — Field order by name
cell array of character vectors | string array

Field order by name, specified as a cell array of character vectors or a string array. The names in `C` must match the field names of `S1`.

`P` — Field order by number
numeric vector

Field order by number, specified as a numeric vector. The numbers must be the integers from `1` through `n`, where `n` is the number of fields of `S1`.

Output Arguments

`S` — Reordered structure
structure array

Reordered structure, returned as a structure array. `S` has the same fields as `S1` but they might be in a different order.

`Pout` — Output field order
numeric vector

Output field order, returned as a numeric vector. The elements of `Pout` are the integers from `1` through `n`, where `n` is the number of fields of `S1`. The permutation of the integers represents the change in the order of the fields.

Tips

- The `orderfields` function only orders top-level fields. It is not recursive.

See Also

`cell2struct` | `fieldnames` | `getfield` | `isfield` | `rmfield` | `setfield` | `struct` | `struct2cell`

Topics

"Access Data in Structure Array"

"Generate Field Names from Variables"

Introduced before R2006a

ordqz

Reorder eigenvalues in QZ factorization

Syntax

```
[AAS,BBS,QS,ZS] = ordqz(AA,BB,Q,Z,select)
[AAS,BBS,QS,ZS] = ordqz(AA,BB,Q,Z,keyword)
[AAS,BBS,QS,ZS] = ordqz(AA,BB,Q,Z,clusters)
```

Description

`[AAS,BBS,QS,ZS] = ordqz(AA,BB,Q,Z,select)` reorders the QZ factorization `Q*A*Z = AA` and `Q*B*Z = BB` produced by `[AA,BB,Q,Z] = qz(A,B)` and returns the reordered matrix pair `(AAS,BBS)` along with orthogonal matrices `(QS,ZS)`, such that `QS*A*ZS = AAS` and `QS*B*ZS = BBS`.

In this reordering, the selected cluster of eigenvalues appears in the leading (upper left) diagonal blocks of the quasitriangular on page 1-10748 pair `(AAS,BBS)`. The leading columns of `ZS` span the corresponding invariant subspace. The logical vector `select` specifies the selected cluster as `e(select)`, where `e = ordeig(AA,BB)`.

`[AAS,BBS,QS,ZS] = ordqz(AA,BB,Q,Z,keyword)` sets the selected cluster to include all eigenvalues in the region specified by `keyword`.

`[AAS,BBS,QS,ZS] = ordqz(AA,BB,Q,Z,clusters)` reorders multiple clusters simultaneously. `ordqz` sorts the specified clusters in descending order along the diagonal of `(AAS,BBS)`, with the cluster of highest index appearing in the upper left corner.

Examples

Reorder QZ Factorization

Compute the QZ factorization of a pair of matrices, and then reorder the factors according to a specified ordering of the eigenvalues.

Find the QZ factorization, or generalized Schur decomposition, of a pair of matrices A and B. This decomposition results in the factors AA = QAZ and BB = QBZ.

```
A = magic(5);
B = hilb(5);
[AA,BB,Q,Z] = qz(A,B)
```

```
AA = 5×5

   14.5272   -2.3517    8.5757   -0.2350   -1.4432
         0  -19.7471    2.1824    4.5417    7.2059
         0         0  -17.9538    8.9292   -9.6961
         0         0         0   30.3449  -47.9191
         0         0         0         0   32.4399


BB = 5×5

    0.0000    0.0005    0.0018    0.0465    0.2304
         0    0.0008    0.0199    0.1662    0.7320
         0         0    0.0210    0.1006   -0.1341
         0         0         0    0.0623   -1.1380
         0         0         0         0    0.7434


Q = 5×5

   -0.1743   -0.1099   -0.0789   -0.4690    0.8552
   -0.7567   -0.1151   -0.0846    0.6172    0.1617
   -0.4010    0.6782    0.5478   -0.2664   -0.0901
    0.4178   -0.0297    0.6473    0.4883    0.4089
   -0.2484   -0.7168    0.5173   -0.2995   -0.2593


Z = 5×5

    0.0057   -0.0424   -0.2914   -0.5860   -0.7549
   -0.1125    0.4109    0.7635    0.1734   -0.4533
    0.4995   -0.6746    0.1486    0.4053   -0.3303
   -0.7694   -0.2140   -0.2614    0.4749   -0.2616
    0.3818    0.5731   -0.4917    0.4866   -0.2173
```

Since AA and BB are triangular, use `ordeig` to extract the eigenvalues from the diagonal blocks of AA and BB.

```
e = ordeig(AA,BB)
```

```
e = 5×1
10^6 ×

    2.8871
   -0.0257
   -0.0009
    0.0005
    0.0000
```

Separate the eigenvalues into clusters, with the real positive eigenvalues ($e > 0$) forming the leading cluster. Reorder the matrices AA, BB, Q, and Z according to this ordering of the eigenvalues.

```
[AAS,BBS,QS,ZS] = ordqz(AA,BB,Q,Z,'rhp')
```

```
AAS = 5×5

   14.5272    1.2849    1.0391    7.6821    4.4119
         0   21.7128   19.1784   -1.8380   -9.1187
         0         0   60.3083   -8.4452   -6.4304
         0         0         0  -18.2081   -3.3783
         0         0         0         0  -14.6375


BBS = 5×5

    0.0000   -0.0114    0.1908   -0.1119    0.0788
         0    0.0446   -0.0377    0.1107   -0.1978
         0         0    1.3820   -0.6325    0.2807
         0         0         0    0.0007    0.0137
         0         0         0         0    0.0171


QS = 5×5

   -0.1743   -0.1099   -0.0789   -0.4690    0.8552
    0.6353   -0.1853   -0.4099   -0.5765   -0.2483
   -0.7034   -0.4518   -0.3456   -0.2295   -0.3591
   -0.1415    0.2036    0.7054   -0.6065   -0.2703
   -0.2263    0.8414   -0.4568   -0.1647   -0.0705
```

```
ZS = 5×5

    0.0057    0.0088   -0.5288    0.3591   -0.7690
   -0.1125    0.6095   -0.3858    0.4737    0.4926
    0.4995   -0.6478   -0.2711    0.3644    0.3529
   -0.7694   -0.4176   -0.4090   -0.1750    0.1890
    0.3818    0.1855   -0.5752   -0.6952    0.0758
```

Examine the new eigenvalue order.

```
E2 = ordeig(AAS,BBS)
```

```
E2 = 5×1
10^6 ×

    2.8871
    0.0005
    0.0000
   -0.0257
   -0.0009
```

Input Arguments

AA, BB — Matrix factors
matrices

Matrix factors, specified as the matrices returned by `[AA,BB,Q,Z] = qz(A,B)`. These matrices satisfy `Q*A*Z = AA` and `Q*B*Z = BB`. For complex matrices, `AA` and `BB` are triangular.

If `AA` and `BB` do not form a valid QZ decomposition, then `ordqz` does not produce an error and returns incorrect results.

Data Types: `single` | `double`
Complex Number Support: Yes

Q, Z — Unitary matrices
matrices

Unitary matrices, specified as the matrices returned by `[AA,BB,Q,Z] = qz(A,B)`. These matrices satisfy `Q*A*Z = AA` and `Q*B*Z = BB`.

Data Types: `single` | `double`
Complex Number Support: Yes

`select` — Cluster selector
logical vector

Cluster selector, specified as a logical vector with length equal to the number of generalized eigenvalues. The generalized eigenvalues appear along the diagonal of `AA-λ*BB`.

Data Types: `logical`

`keyword` — Eigenvalue region keyword
`'lhp'` | `'rhp'` | `'udi'` | `'udo'`

Eigenvalue region keyword, specified as one of the options in this table.

Option	Selected Region (`e = ordeig(AA,BB)`)
`'lhp'`	Left-half plane (`real(e) < 0`)
`'rhp'`	Right-half plane (`real(e) >= 0`)
`'udi'`	Interior of unit disk (`abs(e) < 1`)
`'udo'`	Exterior of unit disk (`abs(e) >= 1`)

`clusters` — Cluster indices
vector of positive integers

Cluster indices, specified as a vector of positive integers with length equal to the number of eigenvalues. `clusters` assigns each eigenvalue returned by `e = ordeig(AA,BB)` to a different cluster. All eigenvalues with the same index value in `clusters` form one cluster.

Example: `ordqz(AA,BB,Q,Z,[1 1 2 3 3])` groups five eigenvalues into three clusters.

Data Types: `single` | `double`

Output Arguments

AAS, BBS, QS, ZS — Reordered matrices
matrices

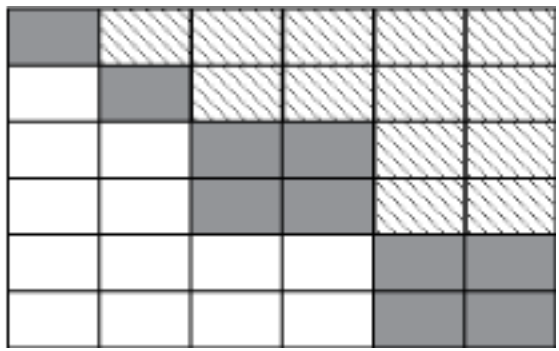
Reordered matrices, returned as matrices that satisfy `QS*A*ZS = AAS` and `QS*B*ZS = BBS`.

`QS` and `ZS` are unitary, while `AAS` is quasitriangular on page 1-10748 and `BBS` is triangular.

More About

Quasitriangular

An upper quasitriangular matrix can result from the Schur decomposition or generalized Schur (QZ) decomposition of real matrices. These matrices are block upper triangular, with 1-by-1 and 2-by-2 blocks along the diagonal.



The eigenvalues of these diagonal blocks are also the eigenvalues of the matrix. The 1-by-1 blocks correspond to real eigenvalues, and the 2-by-2 blocks correspond to complex conjugate eigenvalue pairs.

Tips

- If AA has complex conjugate pairs (nonzero elements on the subdiagonal), then you should move the pair to the same cluster. Otherwise, `ordqz` acts to keep the pair together:
  - If `select` is not the same for two eigenvalues in a conjugate pair, then `ordqz` treats both as selected.
  - If `clusters` is not the same for two eigenvalues in a conjugate pair, then `ordqz` treats both as part of the cluster with larger index.

See Also

`ordeig` | `ordschur` | `qz`

Introduced before R2006a

ordschur

Reorder eigenvalues in Schur factorization

Syntax

```
[US,TS] = ordschur(U,T,select)
[US,TS] = ordschur(U,T,keyword)
[US,TS] = ordschur(U,T,clusters)
```

Description

`[US,TS] = ordschur(U,T,select)` reorders the Schur factorization `X = U*T*U'` produced by `[U,T] = schur(X)` and returns the reordered Schur matrix `TS` and the orthogonal matrix `US`, such that `X = US*TS*US'`.

In this reordering, the selected cluster of eigenvalues appears in the leading (upper left) diagonal blocks of the quasitriangular on page 1-10754 Schur matrix `TS`. The leading columns of `US` span the corresponding invariant subspace. The logical vector `select` specifies the selected cluster as `e(select)`, where `e = ordeig(T)`.

`[US,TS] = ordschur(U,T,keyword)` sets the selected cluster to include all eigenvalues in the region specified by `keyword`.

`[US,TS] = ordschur(U,T,clusters)` reorders multiple clusters simultaneously. `ordschur` sorts the specified clusters in descending order along the diagonal of `TS`, with the cluster of highest index appearing in the upper left corner.

Examples

Reorder Schur Factorization

Compute the Schur factors of a matrix, then reorder the factors according to a specified ordering of the eigenvalues.

Find the Schur factorization of a matrix X. The Schur factorization produces an upper quasitriangular matrix T and a unitary matrix U such that $X = UTU^*$.

```
X = magic(6);
[U,T] = schur(X)
```

```
U = 6×6

    0.4082   -0.2887    0.4082    0.5749    0.5000   -0.0530
    0.4082    0.5774    0.4082   -0.3333   -0.0000   -0.4714
    0.4082   -0.2887    0.4082   -0.2416   -0.5000    0.5244
    0.4082    0.2887   -0.4082    0.5749   -0.5000   -0.0530
    0.4082   -0.5774   -0.4082   -0.3333   -0.0000   -0.4714
    0.4082    0.2887   -0.4082   -0.2416    0.5000    0.5244


T = 6×6

  111.0000    0.0000   -0.0000    0.0000    0.0000   -0.0000
         0   27.0000   -0.0000  -28.3164  -15.5885  -13.0454
         0         0  -27.0000   18.0000   22.0454  -12.7279
         0         0         0    9.7980    0.0000    6.9282
         0         0         0         0   -0.0000    0.0000
         0         0         0         0         0   -9.7980
```

Since T is triangular, the diagonal of T contains the eigenvalues of the original matrix X.

Reorder the Schur factorization so that the eigenvalues are in two clusters, with the cluster of negative eigenvalues appearing first along the diagonal of TS.

```
[US,TS] = ordschur(U,T,'lhp')
```

```
US = 6×6

    0.4082   -0.2887    0.2746    0.4082   -0.4826    0.5244
    0.4082   -0.2887    0.2990    0.4082    0.5213   -0.4714
    0.4082    0.5774   -0.5736    0.4082   -0.0386   -0.0530
   -0.4082    0.2887    0.2075    0.4082    0.5151    0.5244
   -0.4082    0.2887    0.3662    0.4082   -0.4765   -0.4714
   -0.4082   -0.5774   -0.5736    0.4082   -0.0386   -0.0530


TS = 6×6
```

```
  -27.0000  -19.0919   18.6997    0.0000   -9.7888   12.7279
         0   -0.0000   -0.3800    0.0000  -15.6493   15.5227
         0         0   -9.7980   -0.0000   -2.4773    8.7185
         0         0         0  111.0000    0.0000    0.0000
         0         0         0         0   27.0000  -26.3600
         0         0         0         0         0    9.7980
```

Input Arguments

U — Unitary matrix
matrix

Unitary matrix, specified as the matrix U returned by `[U,T] = schur(X)`. The matrix U satisfies the properties `X = U*T*U'` and `U'*U = eye(size(X))`.

If U and T do not form a valid Schur decomposition, then `ordschur` does not produce an error and returns incorrect results.

Data Types: `single` | `double`
Complex Number Support: Yes

T — Schur matrix
matrix

Schur matrix, specified as the upper quasitriangular on page 1-10754 matrix T returned by `[U,T] = schur(X)`. The matrix T satisfies `X = U*T*U'`.

If U and T do not form a valid Schur decomposition, then `ordschur` does not produce an error and returns incorrect results.

Data Types: `single` | `double`
Complex Number Support: Yes

`select` — Cluster selector
logical vector

Cluster selector, specified as a logical vector with length equal to the number of eigenvalues. The eigenvalues appear along the diagonal of the matrix T produced by `[U,T] = schur(X)`.

Data Types: `logical`

keyword — Eigenvalue region keyword
`'lhp'` | `'rhp'` | `'udi'` | `'udo'`

Eigenvalue region keyword, specified as one of the options in this table.

Option	Selected Region (`e = ordeig(T)`)
`'lhp'`	Left-half plane (`real(e) < 0`)
`'rhp'`	Right-half plane (`real(e) >= 0`)
`'udi'`	Interior of unit disk (`abs(e) < 1`)
`'udo'`	Exterior of unit disk (`abs(e) >= 1`)

`clusters` — Cluster indices
vector of positive integers

Cluster indices, specified as a vector of positive integers with length equal to the number of eigenvalues. `clusters` assigns each eigenvalue returned by `e = ordeig(T)` to a different cluster. All eigenvalues with the same index value in `clusters` form one cluster.

Example: `ordschur(U,T,[1 1 2 3 3])` groups five eigenvalues into three clusters.

Data Types: `single` | `double`

Output Arguments

US, TS — Reordered matrices
matrix

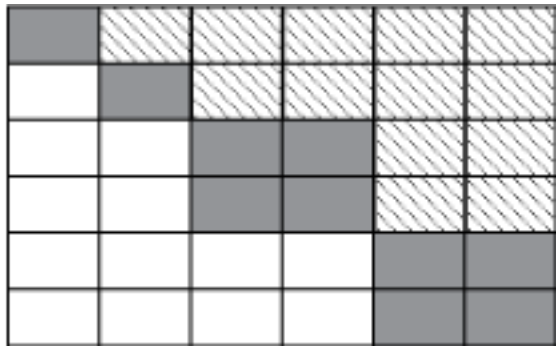
Reordered matrices, returned as matrices that satisfy `X = US*TS*US'`.

`US` is a unitary matrix, and `TS` is quasitriangular on page 1-10754.

More About

Quasitriangular

An upper quasitriangular matrix can result from the Schur decomposition or generalized Schur (QZ) decomposition of real matrices. These matrices are block upper triangular, with 1-by-1 and 2-by-2 blocks along the diagonal.



The eigenvalues of these diagonal blocks are also the eigenvalues of the matrix. The 1-by-1 blocks correspond to real eigenvalues, and the 2-by-2 blocks correspond to complex conjugate eigenvalue pairs.

Tips

- If `T` has complex conjugate pairs (nonzero elements on the subdiagonal), then you should move the pair to the same cluster. Otherwise, `ordschur` acts to keep the pair together:
  - If `select` is not the same for two eigenvalues in a conjugate pair, then `ordschur` treats both as selected.
  - If `clusters` is not the same for two eigenvalues in a conjugate pair, then `ordschur` treats both as part of the cluster with larger index.

See Also

`ordeig` | `ordqz` | `schur`

Introduced before R2006a

orient

Paper orientation for printing or saving

Syntax

```
orient ornt
orient(fig,ornt)

or = orient
or = orient(fig)
```

Description

`orient ornt` specifies the paper orientation to use when printing or saving the current figure to a paged format, such as PDF. Specify the orientation as either `portrait`, `landscape`, or `tall`. Figure property values related to printing, such as the `PaperPositionMode` value, affect the behavior.

`orient(fig,ornt)` sets the paper orientation for the figure or Simulink model specified by `fig`. Use single quotes around the orientation option, such as `'portrait'`.

`or = orient` returns the paper orientation value for the current figure as either `'portrait'`, `'landscape'`, or `'tall'`.

`or = orient(fig)` returns the paper orientation value for the figure or Simulink model specified by `fig`.

Examples

Set Landscape Orientation with Current Figure Size

Create a figure with a surface plot and save the figure to a PDF format. Save the figure with the same size that it appears on screen and use a landscape orientation for the page.

```
fig = figure;
surf(peaks)
orient(fig,'landscape')
print(fig,'LandscapePage.pdf','-dpdf')
```

Set Landscape Orientation with Expanded Figure Size

Create a figure with a surface plot and save the figure to a PDF format. Use landscape orientation for the page and expand the figure size to fill page.

```
fig = figure;
surf(peaks)
fig.PaperPositionMode = 'manual';
orient(fig,'landscape')
print(fig,'LandscapePage_ExpandedFigure.pdf','-dpdf')
```

Return Current Paper Orientation Value

Create a surface plot. Return the current paper orientation value for the figure.

```
figure
surf(peaks)
or = orient
```

```
or =

    'portrait'
```

Input Arguments

`ornt` — Paper orientation
`portrait` (default) | `landscape` | `tall`

Paper orientation, specified as one of the values in this table.

Value	Description
`landscape`	Landscape orientation. Set the `PaperOrientation` property to `'landscape'`. • If the `PaperPositionMode` property of the figure is set to `'auto'` (the default), then print or save the figure using the size of the figure on the screen. • If the `PaperPositionMode` property of the figure is set to `'manual'`, then resize the figure to fit across the entire page with a 0.25-inch margin. Set the `PaperPosition` property accordingly.
`tall`	Tall portrait orientation. Set the `PaperOrientation` property to `'portrait'` and resize the figure to fit across the entire page with a 0.25-inch margin. Set the `PaperPosition` property accordingly.
`portrait`	Portrait orientation. Set the `PaperOrientation` property to `'portrait'`. • If the `PaperPositionMode` property of the figure is set to `'auto'`, then print or save the figure using the size of the figure on the screen. • If the `PaperPositionMode` property is `'manual'`, then set the `PaperPosition` property based on the default `PaperType` and `PaperOrientation` values. For details, see Effect of Default Values on `portrait` option.

Effect of Default Values on portrait Option

The default `PaperType` and `PaperOrientation` properties affect the result of the `portrait` option in these ways.

Default Versus Current Figure PaperType Value	Default PaperOrientation Value	Current Figure PaperOrientation Value	Result
Same value	`'portrait'`	Not applicable	Position the figure on the page using the default `PaperPosition` value for figures. In inches, it typically is `[0.25 2.5 8 6]`.
Same value	`'landscape'`	Not applicable	Position the figure on the page using the reverse of the default `PaperPosition` value for figures. In inches, it typically is `[2.5 0.25 6 8]`.
Different values	Not applicable	`'portrait'`	Position the figure on the page using the current `PaperPosition` value for the figure.
Different values	Not applicable	`'landscape'`	Position the figure on the page using the reverse of the current `PaperPosition` value for the figure, such as`[y x height width]`.

`fig` — Target figure or Simulink model
figure object | Simulink model

Target figure or Simulink model, specified as a figure object or Simulink model. If you do not specify the figure or model, then `orient` uses the current figure.

See Also

`print` | `printpreview`

Introduced before R2006a

orth

Orthonormal basis for range of matrix

Syntax

```
Q = orth(A)
```

Description

`Q = orth(A)` returns an orthonormal basis for the range on page 1-10763 of `A`. The columns of `Q` are vectors, which span the range of `A`. The number of columns in `Q` is equal to the rank on page 1-10763 of `A`.

Examples

Basis for Full Rank Matrix

Calculate and verify the orthonormal basis vectors for the range of a full rank matrix.

Define a matrix and find the rank.

```
A = [1 0 1;-1 -2 0; 0 1 -1];
r = rank(A)
```

```
r = 3
```

Since `A` is a square matrix of full rank, the orthonormal basis calculated by `orth(A)` matches the matrix `U` calculated in the singular value decomposition, `[U,S] = svd(A,'econ')`. This is because the singular values of `A` are all nonzero.

Calculate the orthonormal basis for the range of `A` using `orth`.

```
Q = orth(A)
```

```
Q = 3×3
```

```
   -0.1200   -0.8097    0.5744
    0.9018    0.1531    0.4042
   -0.4153    0.5665    0.7118
```

The number of columns in `Q` is equal to `rank(A)`. Since `A` is of full rank, `Q` and `A` are the same size.

Verify that the basis, `Q`, is orthogonal and normalized within a reasonable error range.

```
E = norm(eye(r)-Q'*Q,'fro')
```

```
E = 1.0857e-15
```

The error is on the order of `eps`.

Basis for Rank Deficient Matrix

Calculate and verify the orthonormal basis vectors for the range of a rank deficient matrix.

Define a singular matrix and find the rank.

```
A = [1 0 1; 0 1 0; 1 0 1];
r = rank(A)
```

```
r = 2
```

Since `A` is rank deficient, the orthonormal basis calculated by `orth(A)` matches only the first `r = 2` columns of matrix `U` calculated in the singular value decomposition, `[U,S] = svd(A,'econ')`. This is because the singular values of `A` are *not* all nonzero.

Calculate the orthonormal basis for the range of `A` using `orth`.

```
Q = orth(A)
```

```
Q = 3×2

   -0.7071         0
         0    1.0000
   -0.7071         0
```

Since A is rank deficient, Q contains one fewer column than A.

Input Arguments

A — Input matrix
matrix

Input matrix.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Range

The column space, or *range*, of a matrix A is the collection of all linear combinations of the columns of A. Any vector, b, that is a solution to the linear equation, `A*x = b`, is included in the range of A since you can also write it as a linear combination of the columns of A.

Rank

The `rank` of a matrix is equal to the dimension of the range.

Algorithms

`orth` is obtained from U in the singular value decomposition, `[U,S] = svd(A,'econ')`. If `r = rank(A)`, the first `r` columns of U form an orthonormal basis for the range of A.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Generated code can return a different basis than MATLAB returns.
- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`null` | `rank` | `svd`

Introduced before R2006a

outerjoin

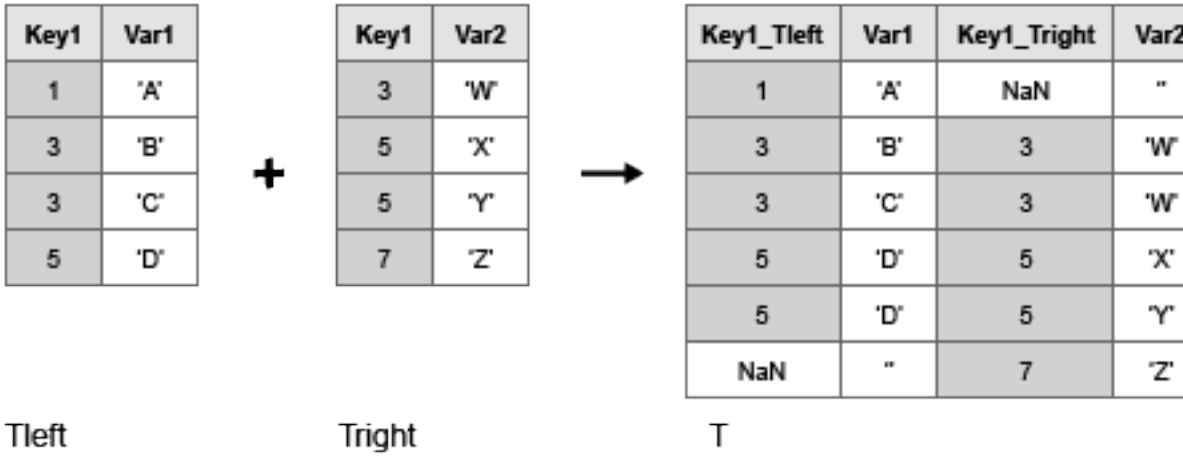
Outer join between two tables or timetables

Syntax

```
T = outerjoin(Tleft,Tright)
T = outerjoin(Tleft,Tright,Name,Value)
[T,ileft,iright] = outerjoin( ___ )
```

Description

`T = outerjoin(Tleft,Tright)` creates the table or timetable, `T`, as the *outer join* of `Tleft` and `Tright` using key variables on page 1-10781. All variables with the same names in both tables are key variables. An outer join combines table rows where the key variables have matching values, but it also includes rows where the key variables from one input table have no matches in the other input table. For example, if `Tleft` has variables named `Key1` and `Var1`, and `Tright` has variables `Key1` and `Var2`, then `T=outerjoin(Tleft,Tright)` uses `Key1` as a key variable.



The matching values of the key variables in the left and right tables do not have to be in the same order. Outer joins can perform one-to-many and many-to-one matches between the key variables of the two tables. That is, a value that occurs once in a key variable of the left table can have multiple matches in the right table. Similarly, a value that occurs once in a key variable of the right table can have multiple matches in the left table.

You can perform outer joins only on certain combinations of tables and timetables.

- If `Tleft` is a table, then `Tright` must be a table. `outerjoin` returns `T` as a table.
- If `Tleft` is a timetable, then `Tright` can be either a table or a timetable. `outerjoin` returns `T` as a timetable for either combination of inputs.

The vectors of row labels of `Tleft` and `Tright` can be key variables. Row labels are the row names of a table, or the row times of a timetable.

`T = outerjoin(Tleft,Tright,Name,Value)` performs the outer-join operation with additional options specified by one or more `Name,Value` pair arguments.

`[T,ileft,iright] = outerjoin( ___ )` also returns index vectors, `ileft` and `iright`, indicating the correspondence between rows in `T` and rows in `Tleft` and `Tright` respectively. You can use this syntax with any of the input arguments in the previous syntaxes.

Examples

Outer-Join Operation of Tables with One Variable in Common

Create a table, `Tleft`.

```
Tleft = table([5;12;23;2;15;6],...
        {'cheerios';'pizza';'salmon';'oreos';'lobster';'pizza'},...
        'VariableNames',{'Age','FavoriteFood'},...
        'RowNames',{'Amy','Bobby','Holly','Harry','Marty','Sally'})
```

```
Tleft=6×2 table
             Age    FavoriteFood
             ___    ____________

    Amy       5     {'cheerios'}
    Bobby    12     {'pizza'   }
    Holly    23     {'salmon'  }
    Harry     2     {'oreos'   }
    Marty    15     {'lobster' }
    Sally     6     {'pizza'   }
```

Create a table, `Tright`, with one variable in common with `Tleft`, called `FavoriteFood`.

```
Tright = table({'cheerios';'oreos';'pizza';'salmon';'cake'},...
         [110;160;140;367;243],...
         {'A-';'D';'B';'B';'C-'},...
         'VariableNames',{'FavoriteFood','Calories','NutritionGrade'})
```

```
Tright=5×3 table
    FavoriteFood    Calories    NutritionGrade
    ____________    ________    ______________

    {'cheerios'}      110           {'A-'}
    {'oreos'   }      160           {'D' }
    {'pizza'   }      140           {'B' }
    {'salmon'  }      367           {'B' }
    {'cake'    }      243           {'C-'}
```

Use the `outerjoin` function to create a new table, `T`, with data from tables `Tleft` and `Tright`.

```
T = outerjoin(Tleft,Tright)
```

```
T=7×5 table
    Age    FavoriteFood_Tleft    FavoriteFood_Tright    Calories    NutritionGrade
    ___    __________________    ___________________    ________    ______________

    NaN       {0x0 char  }          {'cake'    }          243         {'C-'    }
      5       {'cheerios'}          {'cheerios'}          110         {'A-'    }
     15       {'lobster' }          {0x0 char  }          NaN         {0x0 char}
      2       {'oreos'   }          {'oreos'   }          160         {'D'     }
     12       {'pizza'   }          {'pizza'   }          140         {'B'     }
      6       {'pizza'   }          {'pizza'   }          140         {'B'     }
     23       {'salmon'  }          {'salmon'  }          367         {'B'     }
```

Table `T` contains a separate variable for the key variable from `Tleft`, called `FavoriteFood_Tleft`, and the key variable from `Tright`, called `FavoriteFood_Tright`.

Merge Key Variable Pair to Single Variable

Create a table, `Tleft`.

```
Tleft = table({'a' 'b' 'c' 'e' 'h'}',[1 2 3 11 17]',...
        'VariableNames',{'Key1' 'Var1'})
```

```
Tleft=5×2 table
    Key1     Var1
    _____    ____

    {'a'}      1
    {'b'}      2
    {'c'}      3
    {'e'}     11
    {'h'}     17
```

Create a table, `Tright`, with common values in the variable `Key1` between tables `Tleft` and `Tright`, but also containing rows with values of `Key1` not present in `Tleft`.

```
Tright = table({'a','b','d','e'}',[4;5;6;7],...
         'VariableNames',{'Key1' 'Var2'})
```

```
Tright=4×2 table
    Key1     Var2
    _____    ____

    {'a'}     4
    {'b'}     5
    {'d'}     6
    {'e'}     7
```

Use the `outerjoin` function to create a new table, `T`, with data from tables `Tleft` and `Tright`. Merge the key values into a single variable in the output table, `T`.

```
T = outerjoin(Tleft,Tright,'MergeKeys',true)
```

```
T=6×3 table
    Key1     Var1    Var2
    _____    ____    ____

    {'a'}      1       4
    {'b'}      2       5
    {'c'}      3     NaN
    {'d'}    NaN       6
    {'e'}     11       7
    {'h'}     17     NaN
```

Variables in table `T` that came from `Tleft` contain null values in the rows that have no match from `Tright`. Similarly, variables in `T` that came from `Tright` contain null values in those rows that had no match from `Tleft`.

Outer-Join Operation of Tables and Indices to Values

Create a table, `Tleft`.

```
Tleft = table({'a' 'b' 'c' 'e' 'h'}',[1 2 3 11 17]',...
        'VariableNames',{'Key1' 'Var1'})
```

```
Tleft=5×2 table
    Key1     Var1
    _____    ____
```

```
    {'a'}      1
    {'b'}      2
    {'c'}      3
    {'e'}     11
    {'h'}     17
```

Create a table, `Tright`, with common values in the variable `Key1` between tables `Tleft` and `Tright`, but also containing rows with values of `Key1` not present in `Tleft`.

```
Tright = table({'a','b','d','e'}',[4;5;6;7],...
         'VariableNames',{'Key1' 'Var2'})
```

```
Tright=4×2 table
    Key1     Var2
    _____    ____

    {'a'}     4
    {'b'}     5
    {'d'}     6
    {'e'}     7
```

Use the `outerjoin` function to create a new table, `T`, with data from tables `Tleft` and `Tright`. Match up rows with common values in the key variable, `Key1`, but also retain rows whose key values don't have a match.

Also, return index vectors, `ileft` and `iright` indicating the correspondence between rows in `T` and rows in `Tleft` and `Tright` respectively.

```
[T,ileft,iright] = outerjoin(Tleft,Tright)
```

```
T=6×4 table
    Key1_Tleft    Var1    Key1_Tright    Var2
    __________    ____    ___________    ____

    {'a'     }      1     {'a'     }       4
    {'b'     }      2     {'b'     }       5
    {'c'     }      3     {0x0 char}     NaN
    {0x0 char}    NaN     {'d'     }       6
    {'e'     }     11     {'e'     }       7
    {'h'     }     17     {0x0 char}     NaN


ileft = 6×1
```

```
     1
     2
     3
     0
     4
     5


iright = 6×1

     1
     2
     0
     3
     4
     0
```

The index vectors `ileft` and `iright` contain zeros to indicate the rows in table `T` that do not correspond to rows in tables `Tleft` or `Tright`, respectively.

Left Outer-Join Operation of Tables and Indices to Values

Create a table, `Tleft`.

```
Tleft = table({'a' 'b' 'c' 'e' 'h'}',[1 2 3 11 17]',...
        'VariableNames',{'Key1' 'Var1'})
```

```
Tleft=5×2 table
    Key1     Var1
    _____    ____

    {'a'}      1
    {'b'}      2
    {'c'}      3
    {'e'}     11
    {'h'}     17
```

Create a table, `Tright`, with common values in the variable `Key1` between tables `Tleft` and `Tright`, but also containing rows with values of `Key1` not present in `Tleft`.

```
Tright = table({'a','b','d','e'}',[4;5;6;7],...
         'VariableNames',{'Key1' 'Var2'})
```

```
Tright=4×2 table
    Key1     Var2
    _____    ____

    {'a'}     4
    {'b'}     5
    {'d'}     6
    {'e'}     7
```

Use the `outerjoin` function to create a new table, `T`, with data from tables `Tleft` and `Tright`. Ignore rows in `Tright` whose key values do not match any rows in `Tleft`.

Also, return index vectors, `ileft` and `iright` indicating the correspondence between rows in `T` and rows in `Tleft` and `Tright` respectively.

```
[T,ileft,iright] = outerjoin(Tleft,Tright,'Type','left')
```

```
T=5×4 table
    Key1_Tleft    Var1    Key1_Tright    Var2
    __________    ____    ___________    ____

      {'a'}         1     {'a'     }       4
      {'b'}         2     {'b'     }       5
      {'c'}         3     {0x0 char}     NaN
      {'e'}        11     {'e'     }       7
      {'h'}        17     {0x0 char}     NaN


ileft = 5×1

     1
     2
     3
     4
     5


iright = 5×1

     1
     2
```

```
     0
     4
     0
```

All values of `ileft` are nonzero indicating that all rows in `T` have corresponding rows in `Tleft`.

Outer-Join Operation of Timetables

Create two timetables, `Tleft` and `Tright`. They have some row times in common, but each also includes row times that are not in the other timetable.

```
Tleft = timetable(seconds([1;2;4;6]),[1 2 3 11]')
```

```
Tleft=4×2 timetable
    Time     Var1
    _____    ____

    1 sec      1
    2 sec      2
    4 sec      3
    6 sec     11
```

```
Tright = timetable(seconds([2;4;6;7]),[4 5 6 7]')
```

```
Tright=4×2 timetable
    Time     Var1
    _____    ____

    2 sec     4
    4 sec     5
    6 sec     6
    7 sec     7
```

Combine `Tleft` and `Tright` with an outer join. `T1` matches up the rows with common row times, but also includes the rows that do not have matches.

```
T1 = outerjoin(Tleft,Tright)
```

```
T1=5×3 timetable
    Time     Var1_Tleft    Var1_Tright
    _____    __________    ___________

    1 sec         1            NaN
    2 sec         2              4
    4 sec         3              5
    6 sec        11              6
    7 sec       NaN              7
```

Combine `Tleft` and `Tright`, but ignore rows in `Tright` whose row times do not match any row times in `Tleft`.

```
T2 = outerjoin(Tleft,Tright,'Type','left')
```

```
T2=4×3 timetable
    Time     Var1_Tleft    Var1_Tright
    _____    __________    ___________

    1 sec         1            NaN
    2 sec         2              4
    4 sec         3              5
    6 sec        11              6
```

Input Arguments

`Tleft` — Left table
table | timetable

Left table, specified as a table or a timetable.

`Tright` — Right table
table | timetable

Right table, specified as a table or a timetable.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Keys',2` uses the second variable in `Tleft` and the second variable in `Tright` as key variables.

Keys — Variables to use as keys
positive integer | vector of positive integers | character vector | cell array of character vectors | string array | logical vector

Variables to use as keys, specified as the comma-separated pair consisting of `'Keys'` and a positive integer, vector of positive integers, character vector, cell array of character vectors, string array, or logical vector.

You cannot use the `'Keys'` name-value pair argument with the `'LeftKeys'` and `'RightKeys'` name-value pair arguments.

A vector of row labels can be a key, alone or in combination with other key variables. For more information, see the “Tips” on page 1-10781 section.

Example: `'Keys',[1 3]` uses the first and third variables in `Tleft` and `Tright` as key variables.

Example: `'Keys',{'X','Y'}` uses the variables named `X` and `Y` in `Tleft` and `Tright` as key variables.

Example: `'Keys','Row'` uses the vectors of row names of `Tleft` and `Tright` as key variables, if both `Tleft` and `Tright` are tables with row names.

`LeftKeys` — Variables to use as keys in `Tleft`
positive integer | vector of positive integers | character vector | cell array of character vectors | string array | logical vector

Variables to use as keys in `Tleft`, specified as the comma-separated pair consisting of `'LeftKeys'` and a positive integer, vector of positive integers, character vector, cell array of character vectors, string array, or logical vector.

You must use the `'LeftKeys'` name-value pair argument in conjunction with the `'RightKeys'` name-value pair argument. `'LeftKeys'` and `'RightKeys'` both must specify the same number of key variables. `outerjoin` pairs key values based on their order.

A vector of row labels can be a key, alone or in combination with other key variables. For more information, see the “Tips” on page 1-10781 section.

Example: `'LeftKeys',1` uses only the first variable in `Tleft` as a key variable.

`RightKeys` — Variables to use as keys in `Tright`

positive integer | vector of positive integers | character vector | cell array of character vectors | string array | logical vector

Variables to use as keys in `Tright`, specified as the comma-separated pair consisting of `'RightKeys'` and a positive integer, vector of positive integers, character vector, cell array of character vectors, string array, or logical vector.

You must use the `'RightKeys'` name-value pair argument in conjunction with the `'LeftKeys'` name-value pair argument. `'LeftKeys'` and `'RightKeys'` both must specify the same number of key variables. `outerjoin` pairs key values based on their order.

A vector of row labels can be a key, alone or in combination with other key variables. For more information, see the “Tips” on page 1-10781 section.

Example: `'RightKeys',3` uses only the third variable in `Tright` as a key variable.

`MergeKeys` — Merge keys flag

`false` (default) | `true` | `0` | `1`

Merge keys flag, specified as the comma-separated pair consisting of `'MergeKeys'` and either `false`, `true`, `0` or `1`.

`false`	`outerjoin` includes two separate variables in the output table, `T`, for each key variable pair from tables `Tleft` and `Tright`. This behavior is the default behavior.

`true`	`outerjoin` includes a single variable in the output table, `T`, for each key variable pair from tables `Tleft` and `Tright`. `outerjoin` creates the single variable by merging the key values from `Tleft` and `Tright`, taking values from `Tleft` where a corresponding row exists in `Tleft`, and taking values from `Tright` otherwise. • If you specify `'LeftVariables'` or `'RightVariables'` to include only one key from a key variable pair, then `outerjoin` includes the merged key—containing values from both key variables—in the output table. • If you specify `'LeftVariables'` and `'RightVariables'` to exclude both keys from a key variable pair, then `outerjoin` does not include the merged key variable in the output table.

`LeftVariables` — Variables from `Tleft` to include in `T`

positive integer | vector of positive integers | character vector | cell array of character vectors | string array | logical vector

Variables from `Tleft` to include in `T`, specified as the comma-separated pair consisting of `'LeftVariables'` and a positive integer, vector of positive integers, character vector, cell array of character vectors, string array, or logical vector.

You can use `'LeftVariables'` to include or exclude key variables as well as nonkey variables from the output, `T`.

By default, `outerjoin` includes all variables from `Tleft`.

`RightVariables` — Variables from `Tright` to include in `T`

positive integer | vector of positive integers | character vector | cell array of character vectors | string array | logical vector

Variables from `Tright` to include in `T`, specified as the comma-separated pair consisting of `'RightVariables'` and a positive integer, vector of positive integers, character vector, cell array of character vectors, string array, or logical vector.

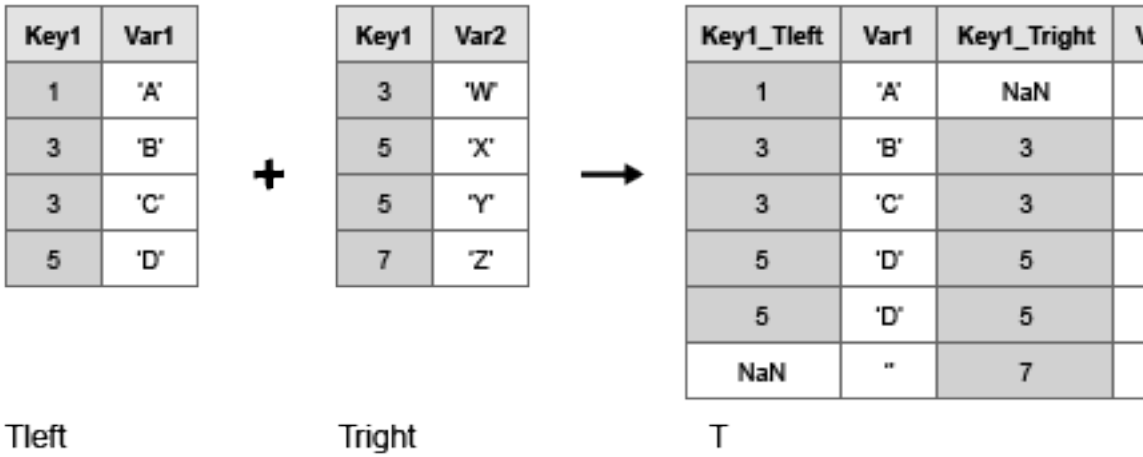
You can use `'RightVariables'` to include or exclude key variables as well as nonkey variables from the output, `T`.

By default, `outerjoin` includes all the variables from `Tright`.

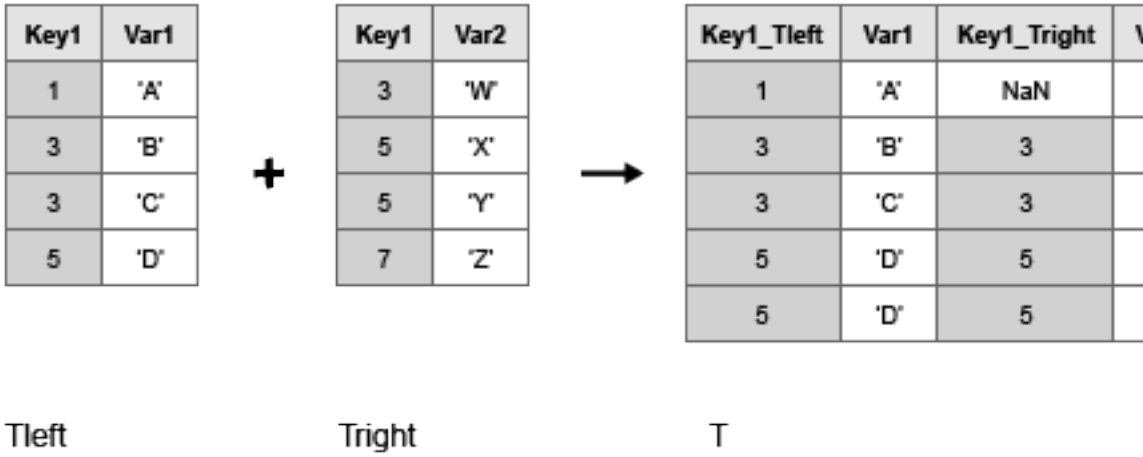
Type — Type of outer join operation
`'full'` (default) | `'left'` | `'right'`

Type of outer join operation, specified as the comma-separated pair consisting of `'Type'` and either `'full'` (meaning a full outer join), `'left'` (left outer join), or `'right'` (right outer join).

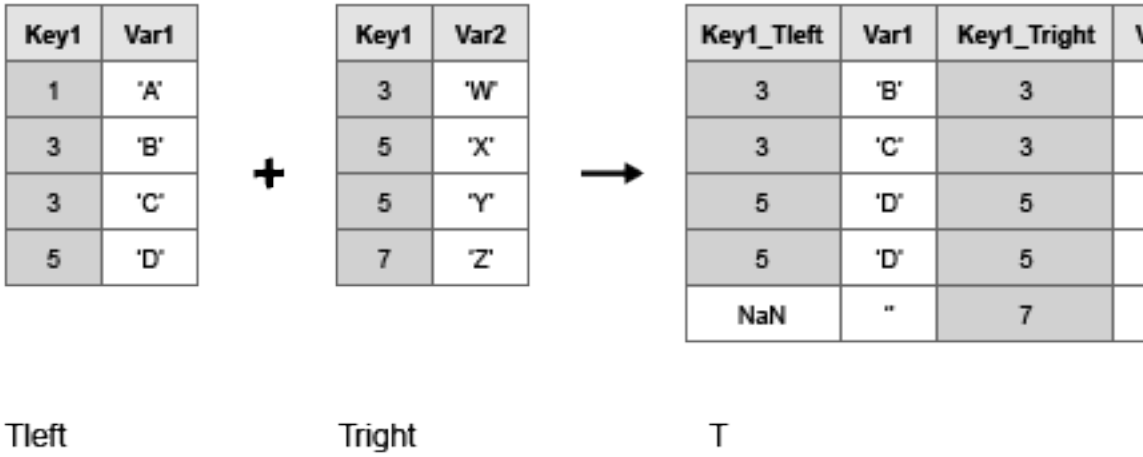
- For a full outer join, `T` includes all key values from both tables, as shown in the diagram below.



- For a left outer join, `T` includes key values in `Tleft` that do not match any key values in `Tright`, but excludes key values in `Tright` that do not match any key values in `Tleft`. For example, in the diagram below, the left outer join excludes the key value 7 from the right table.



- For a right outer join, `T` includes key values in `Tright` that do not match any key values in `Tleft`, but excludes key values in `Tleft` that do not match any key values in `Tright`. For example, in the diagram below, the right outer join excludes the key value `1` from the left table.



By default, `outerjoin` does a full outer join and includes unmatched rows from both `Tleft` and `Tright`.

Output Arguments

`T` — Outer join from `Tleft` and `Tright`
table

Outer join from `Tleft` and `Tright`, returned as a table. The output table, `T`, contains one row for each pair of rows in tables `Tleft` and `Tright` that share the same combination of key values. If `Tleft` and `Tright` contain variables with the same name, `outerjoin` adds a unique suffix to the corresponding variable names in `T`. Variables in `T` that came from `Tleft` contain null values in those rows that had no match from `Tright`. Similarly, variables in `T` that came from `Tright` contain null values in those rows that had no match from `Tleft`.

In general, if there are `m` rows in table `Tleft` and `n` rows in table `Tright` that all contain the same combination of values in the key variables, table `T` contains `m*n` rows for that combination. `T` also contains rows corresponding to key value combinations in one input table that do not match any row the other input table.

T contains the horizontal concatenation of `Tleft(ileft,LeftVars)` and `Tright(iright,RightVars)` sorted by the values in the key variables. By default, `LeftVars` consists of all the variables of `Tleft`, and `RightVars` consists of all the variables from `Tright`. Otherwise, `LeftVars` consists of the variables specified by the `'LeftVariables'` name-value pair argument, and `RightVars` consists of the variables specified by the `'RightVariables'` name-value pair argument.

You can store additional metadata such as descriptions, variable units, variable names, and row names in the table. For more information, see the Properties section of `table`.

`ileft` — Index to `Tleft`
column vector

Index to `Tleft`, returned as a column vector. Each element of `ileft` identifies the row in table `Tleft` that corresponds to that row in the output table, `T`. The vector `ileft` contains zeros to indicate the rows in `T` that do not correspond to rows in `Tleft`.

`iright` — Index to `Tright`
column vector

Index to `Tright`, returned as a column vector. Each element of `iright` identifies the row in table `Tright` that corresponds to that row in the output table, `T`. The vector `iright` contains zeros to indicate the rows in `T` that do not correspond to rows in `Tright`.

More About

Key Variable

Variable used to match and combine data between the input tables, `Tleft` and `Tright`.

Tips

The vector of row labels from an input table or timetable can be a key, alone or in combination with other key variables. Row labels are the row names of a table or the row times of a timetable. To use this vector as a key, specify it as `'Row'` (for the row names of a table), as the name of a timetable vector of row times, or as the value of *`T`*`.Properties.DimensionNames{1}`, where *`T`* is the table or timetable.

In general, `outerjoin` copies row labels from the input table `Tleft` to the output table `T`.

- If `Tleft` has no row labels, then `T` has no row labels.
- If `Tleft` has row labels, then `outerjoin` copies row labels from `Tleft` to create row labels in `T`.
  - If you specify row labels from both `Tleft` and `Tright` as a key pair, then `outerjoin` merges row labels from `Tright` into row labels of `T` where needed.
  - If you specify row labels of `Tleft` as a key, but do not specify row labels of `Tright` as the matching key, then `outerjoin` creates default row labels in `T` where needed.
  - If both `Tleft` and `Tright` are tables, but you do not specify either input table's row names as a key, then `outerjoin` does not create row names in `T`.

You cannot perform an outer join using the row labels of `Tleft` as the left key and a variable of `Tright` as the right key. To perform the outer join, convert the row labels of `Tleft` to a table variable and use the new table variable as a key.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see "Tall Arrays".

See Also

Join Tables | `innerjoin` | `join`

Introduced in R2013b

Clean Outlier Data

Find, fill, or remove outliers in the Live Editor

Description

The **Clean Outlier Data** task lets you interactively handle outliers in data. The task automatically generates MATLAB code for your live script.

Using this task, you can:

- Find, fill, or remove outliers from data in a workspace variable.
- Customize the methods for finding and filling outliers.
- Automatically visualize the outlier data and cleaned data.

Open the Task

To add the **Clean Outlier Data** task to a live script in the MATLAB Editor:

- On the **Live Editor** tab, select **Task > Clean Outlier Data**.
- In a code block in the script, type a relevant keyword, such as `outlier` or `clean`. Select **Clean Outlier Data** from the suggested command completions.

Parameters

`Cleaning method` — Cleaning method for filling outliers

`Linear interpolation` (default) | `Constant value` | `Center value` | `Clip to threshold value` | `Previous value` | `Next value` | `Nearest value` | `Spline interpolation` | `Shape-preserving cubic interpolation (PCHIP)` | `Modified Akima cubic interpolation` | …

Specify the method for filling outliers using one of the following options.

Fill Method	Description
`Linear interpolation`	Linear interpolation of neighboring, nonoutlier values.
`Constant value`	Specified scalar value, which is `0` by default.
`Center value`	Center value determined by the find method.
`Clip to threshold value`	Fills lower threshold value for elements smaller than the lower threshold determined by the find method. Fills with the upper threshold value for elements larger than the upper threshold determined by the find method.
`Previous value`	Previous nonoutlier value.
`Next value`	Next nonoutlier value.
`Nearest value`	Nearest nonoutlier value.
`Spline interpolation`	Piecewise cubic spline interpolation.
`Shape-preserving cubic interpolation (PCHIP)`	Shape-preserving piecewise cubic spline interpolation.
`Modified Akima cubic interpolation`	Modified Akima cubic Hermite interpolation.

`Detection method` — Method for detecting outliers

`Median` (default) | `Mean` | `Quartiles` | `Grubbs` | `Generalized extreme studentized deviate (GESD)` | `Moving median` | `Moving mean` | ...

Specify the detection method for finding outliers using one of the following options.

Method	Description
`Median`	Outliers are defined as elements more than the specified threshold of scaled median absolute deviations (MAD) from the median, which is `3` by default. For input data `A`, the scaled MAD is defined as `c*median(abs(A-median(A)))`, where `c=-1/(sqrt(2)*erfcinv(3/2))`.
`Mean`	Outliers are defined as elements more than the specified threshold of standard deviations from the mean, which is `3` by default. This method is faster but less robust than `Median`.
`Quartiles`	Outliers are defined as elements more than the specified threshold of interquartile ranges above the upper quartile (75 percent) or below the lower quartile (25 percent), which is `1.5` by default. This method is useful when the input data is not normally distributed.
`Grubbs`	Outliers are detected using Grubbs's test, which removes one outlier per iteration based on hypothesis testing. This method assumes that the input data is normally distributed.
`Generalized extreme studentized deviate (GESD)`	Outliers are detected using the generalized extreme studentized deviate test for outliers. This iterative method is similar to `Grubbs`, but can perform better when multiple outliers are masking each other.
`Moving median`	Outliers are defined as elements more than the specified threshold of local scaled MAD from the local median over a specified window. The default threshold is `3`.

Method	Description
`Moving mean`	Outliers are defined as elements more than the specified threshold of local standard deviations from the local mean over a specified window. The default threshold is 3.

`Moving window` — Window for moving methods
`Centered` (default) | `Asymmetric`

Specify the window type and size when the method for detecting outliers is `Moving median` or `Moving mean`.

Window	Description
`Centered`	Specified window length centered about the current point.
`Asymmetric`	Specified window containing the number of elements before the current point and the number of elements after the current point.

Window sizes are relative to the **X-axis** variable units.

See Also

Functions
`filloutliers` | `isoutlier` | `rmoutliers`

Live Editor Tasks
Clean Missing Data | **Find Change Points** | **Find Local Extrema** | **Remove Trends** | **Smooth Data**

Topics

"Add Interactive Tasks to a Live Script"
Clean Messy Data and Locate Extrema Using Live Editor Tasks
"Data Smoothing and Outlier Detection"

Introduced in R2019b

pack

Consolidate workspace memory

Syntax

```
pack
pack filename
pack('filename')
```

Description

`pack` frees up needed space by reorganizing information so that it only uses the minimum memory required. All variables from your base and global workspaces that are less than 2GB are preserved. Any persistent variables that are defined at the time are set to their default value (the empty matrix, `[]`).

Caution The `pack` function is unable to preserve variables larger than 2GB. It clears them from your workspace.

Do not use the `pack` function with objects that define events and listeners. The `pack` function causes the destructor of any listeners defined for the objects in the workspace. For information on restoring listeners when saving objects, see “Restore Listeners”.

The MATLAB software temporarily stores your workspace data in a file called `tp######.mat` (where `######` is a numeric value) that is located in your temporary folder. (You can use the command `dir(tempdir)` to see the files in this folder).

`pack filename` frees space in memory, temporarily storing workspace data in a file specified by `filename`. This file resides in your current working folder and, unless specified otherwise, has a `.mat` file extension.

`pack('filename')` is the function form of `pack`.

Examples

Change the current folder to one that is writable, run `pack`, and return to the previous folder.

```
cwd = pwd;
cd(tempdir);
pack
cd(cwd)
```

Tips

You can only run `pack` from the MATLAB command line.

If you specify a `filename` argument, that file must reside in a folder for which you have write permission.

The `pack` function does not affect the amount of memory allocated to the MATLAB process. You must quit MATLAB to free up this memory.

Since MATLAB uses a heap method of memory management, extended MATLAB sessions may cause memory to become fragmented. When memory is fragmented, there may be plenty of free space, but not enough contiguous memory to store a new large variable.

If you get the `Out of memory` message from MATLAB, the `pack` function may find you some free memory without forcing you to delete variables.

The `pack` function frees space by

- Saving all variables in the base and global workspaces to a temporary file.
- Clearing all variables and functions from memory.
- Reloading the base and global workspace variables back from the temporary file and then deleting the file.

If you use `pack` and there is still not enough free memory to proceed, you must clear some variables. If you run out of memory often, you can allocate larger matrices earlier in the MATLAB session and use these system-specific tips.

- When running MATLAB on The Open Group UNIX platforms, ask your system manager to increase your swap space.

- On Microsoft Windows platforms, increase virtual memory using the Windows Control Panel.

To maintain persistent variables when you run pack, use `mlock` in the function.

See Also

`clear` | `memory`

Introduced before R2006a

pad

Add leading or trailing characters to strings

Syntax

```
newStr = pad(str)
newStr = pad(str,numberOfCharacters)
newStr = pad(str,side)
newStr = pad(str,numberOfCharacters,side)
newStr = pad( ___ ,padCharacter)
```

Description

`newStr = pad(str)` adds space characters to the ends of the strings in `str`, except for the longest one.

- If `str` is a string array or cell array of character vectors with multiple elements, then `pad` adds space characters. All of the strings in `newStr` are as long as the longest element in `str`.
- If `str` is a character vector, or a string array or cell array of character vectors with one element, then `pad` returns `str` unaltered.

`newStr = pad(str,numberOfCharacters)` adds space characters so the strings in `newStr` have the length specified by `numberOfCharacters`. If any strings in `str` have more characters than `numberOfCharacters`, then `pad` does not modify them.

`newStr = pad(str,side)` adds space characters to the side specified by `side`. The `side` argument can be `'left'`, `'right'`, or `'both'`.

`newStr = pad(str,numberOfCharacters,side)` adds space characters to the side specified by `side`, up to the length specified by `numberOfCharacters`.

`newStr = pad( ___ ,padCharacter)` pads strings with the character specified by `padCharacter` instead of the space character. You can use any of the input arguments in the previous syntaxes.

If `str` contains only one piece of text, then `pad(str,padCharacter)` returns `str` unaltered.

Examples

Pad String Array with Spaces

Create a string array. Starting in R2017a, you can create strings using double quotes.

```
str = ["Mercury","Gemini","Apollo";
       "Skylab","Skylab B","ISS"]
```

```
str = 2x3 string array
    "Mercury"    "Gemini"      "Apollo"
    "Skylab"     "Skylab B"    "ISS"
```

Pad the elements of `str` with space characters.

```
newStr = pad(str)
```

```
newStr = 2x3 string array
    "Mercury "    "Gemini  "    "Apollo  "
    "Skylab  "    "Skylab B"    "ISS     "
```

Pad String Array to Specified Length

Create a string array. Starting in R2017a, you can create strings using double quotes.

```
str = ["Mercury","Gemini","Apollo";
       "Skylab","Skylab B","ISS"]
```

```
str = 2x3 string array
    "Mercury"    "Gemini"      "Apollo"
    "Skylab"     "Skylab B"    "ISS"
```

Specify the length so that even the longest string is padded with spaces.

```
newStr = pad(str,12)
```

```
newStr = 2x3 string array
    "Mercury     "    "Gemini      "    "Apollo      "
    "Skylab      "    "Skylab B    "    "ISS         "
```

Pad Strings to Different Sides

Create a string array. Starting in R2017a, you can create strings using double quotes.

```
str = ["Mary";"Elizabeth";"James"]
```

```
str = 3x1 string array
    "Mary"
    "Elizabeth"
    "James"
```

Pad the strings to the left.

```
newStr = pad(str,'left')
```

```
newStr = 3x1 string array
    "     Mary"
    "Elizabeth"
    "    James"
```

Pad both sides.

```
newStr = pad(str,'both')
```

```
newStr = 3x1 string array
    "  Mary   "
    "Elizabeth"
    "  James  "
```

Pad String Array with Different Character

Create a string array representing numbers and pad the strings with leading zeroes instead of space characters.

```
A = [69.45 31.71 95.36 3.44 7.82];
A = A';
str = string(A)
```

```
str = 5x1 string array
    "69.45"
    "31.71"
    "95.36"
    "3.44"
    "7.82"
```

```
newStr = pad(str,7,'left','0')
```

```
newStr = 5x1 string array
    "0069.45"
    "0031.71"
    "0095.36"
    "0003.44"
    "0007.82"
```

Input Arguments

`str` — Input text
string array | character vector | cell array of character vectors

Input text, specified as a string array, a character vector, or a cell array of character vectors.

Data Types: `string` | `char` | `cell`

`numberOfCharacters` — Total number of characters in output strings
positive integer

Total number of characters in output strings, specified as a positive integer.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`side` — Side of string to pad
`'right'` (default) | `'left'` | `'both'`

Side of string to pad, specified as `'left'`, `'right'`, or `'both'`. The default behavior is to pad the right side of the string.

Data Types: `char` | `string`

`padCharacter` — Character to pad with
`' '` (default) | character | string

Character to pad with, specified as a character or as a string that contains one character.

Data Types: `char` | `string`

Output Arguments

`newStr` — Output text
string array | character vector | cell array of character vectors

Output text, returned as a string array, a character vector, or a cell array of character vectors. `str` and `newStr` are the same data type.

Data Types: `string` | `char` | `cell`

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitation:

If you do not specify `numberOfCharacters`, then a full pass through the data is required to determine it.

For more information, see “Tall Arrays for Out-of-Memory Data”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`erase` | `join` | `replace` | `split` | `strip` | `strsplit`

Topics

“Create String Arrays”
“Search and Replace Text”

Introduced in R2016b

padecoef

Padé approximation of time delays

Syntax

```
[num,den] = padecoef(T,N)
```

Description

`[num,den] = padecoef(T,N)` returns the `N`th-order Padé Approximation on page 1-10798 of the continuous-time delay `exp(-T*s)` in transfer function form. The row vectors `num` and `den` contain the numerator and denominator coefficients in descending powers of `s`. Both are `N`th-order polynomials.

Examples

Second Order Pade Approximation

Use `padecoef` to estimate the value of e^{-2s} to second order.

```
[a,b] = padecoef(2,2)
```

```
a = 1×3

     1    -3     3


b = 1×3

     1     3     3
```

The result indicates that the second order approximation is

$$f(s) \approx \frac{a}{b} = \frac{s^2 - 3s + 3}{s^2 + 3s + 3}.$$

Compare the approximation to the actual value at $s = 0.25$.

```
f_approx = @(s) (s^2 - 3*s+3)/(s^2 + 3*s + 3);
f_actual = @(s) exp(-2*s);
abs(f_approx(0.25) - f_actual(0.25))
```

```
ans = 2.6717e-05
```

Input Arguments

T — Time delay
scalar

Time delay, specified as a real numeric scalar.

Data Types: `single` | `double`

N — Order of approximation
scalar

Order of approximation, specified as a real numeric scalar.

Data Types: `single` | `double`

More About

Padé Approximation

The Laplace transform of a time delay of `T` seconds is `exp(-Ts)`. The `padecoef` function approximates this exponential transfer function by a rational transfer function using Padé approximation formulas. [1]

References

[1] Golub, G. H. and C. F. Van Loan. *Matrix Computations*. 4th ed. Johns Hopkins University Press, Baltimore: 2013, pp. 530–532.

See Also

pade

Introduced in R2008a

pagesetupdlg

Page setup dialog box

Syntax

```
dlg = pagesetupdlg(fig)
```

Note `pagesetupdlg` is no longer supported. Use `printpreview` instead.

Description

`dlg = pagesetupdlg(fig)` creates a dialog box from which a set of page layout properties for the figure window, `fig`, can be set.

`pagesetupdlg` implements the "Page Setup..." option in the **Figure File Menu**.

`pagesetupdlg` supports setting the layout for a single figure. `fig` must be a single figure handle, not a vector of figures or a Simulink diagram.

See Also

`printdlg` | `printopt` | `printpreview`

Introduced before R2006a

pan

Pan view of graph interactively

Syntax

```
pan on
pan xon
pan yon
pan off
pan
pan(fig,...)
h = pan(fig)
```

Description

`pan on` turns on pan mode for axes in the current figure.

`pan xon` turns on pan mode and enables panning only in the *x* direction for axes in a 2-D view in the current figure.

`pan yon` turns on pan mode and enables panning only in the *y* direction for axes in a 2-D view in the current figure.

`pan off` turns off pan mode for axes in the current figure. Starting in R2018b, some pan interactions are enabled by default, regardless of the pan mode. If you want to disable these default interactions, then use the `disableDefaultInteractivity` function.

`pan` toggles the pan mode for axes in the current figure to `on` or `off`.

`pan(fig,...)` sets the pan mode for axes in the specified figure.

`h = pan(fig)` returns the figure's pan *mode object* for the figure `fig` for you to customize the mode's behavior.

Using Pan Mode Objects

Access the following properties of pan mode objects.

- *Enable* `'on'|'off'` — Specifies whether this figure mode is currently enabled on the figure.
- *Motion* `'horizontal'|'vertical'|'both'` — The type of panning enabled for the figure. This property only affects axes in a 2-D view (`[0 90]`).
- `FigureHandle <handle>` — The associated figure handle, a read-only property that cannot be set.

Pan Mode Callbacks

You can program the following callbacks for pan mode operations.

- `ButtonDownFilter <function_handle>` — Function to intercept `ButtonDown` events

  The application can inhibit the panning operation under circumstances the programmer defines, depending on what the callback returns. The input function handle should reference a function with two implicit arguments (similar to graphics object callbacks):

  ```
  function [res] = myfunction(obj,event_obj)
  % obj          handle to the object clicked on
  % event_obj    event data (empty in this release)
  % res [output] a logical flag to determine whether the pan
  %              operation should take place(for 'res' set to 'false')
  %              or the 'ButtonDownFcn' property of the object should
  %              take precedence (when 'res' is 'true')
  ```

- `ActionPreCallback <function_handle>` — Function to execute before panning

  Set this callback to if you need to execute code when a pan operation begins. The function handle should reference a function with two implicit arguments (similar to graphics object callbacks):

  ```
  function myfunction(obj,event_obj)
  % obj         handle to the figure that has been clicked on
  % event_obj   object containing struct of event data
  ```

  The event data struct has the following field:

`Axes`	The handle of the axes that is being panned

- `ActionPostCallback <function_handle>` — Function to execute after panning

  Set this callback if you need to execute code when a pan operation ends. The function handle should reference a function with two implicit arguments (similar to graphics object callbacks):

  ```
  function myfunction(obj,event_obj)
  % obj         handle to the figure that has been clicked on
  % event_obj   object containing struct of event data
  %             (same as the event data of the
  %             'ActionPreCallback' callback)
  ```

Pan Mode Utility Functions

The following functions in pan mode query and set certain of its properties.

- `flags = isAllowAxesPan(h,ax)` — Function querying permission to pan axes.

  Calling the function `isAllowAxesPan` on the pan object, `h`, with a vector of axes handles, `ax`, as input returns a logical array of the same dimension as the axes handle vector, which indicates whether a pan operation is permitted on the axes objects.
- `setAllowAxesPan(h,ax,flag)` — Function to set permission to pan axes.

  Calling the function `setAllowAxesPan` on the pan object, `h`, with a vector of axes handles, `ax`, and a logical scalar, `flag`, either allows or disallows a pan operation on the axes objects.
- `cn = getAxesPanConstraint(h,ax)` — Function to get constraints of pan operations.

  Calling the function `getAxesPanConstraint` on the pan object, `h`, with an axes object, `ax`, as input returns the constraint for the axes. The returned constraint is one of these values: `'x'`, `'y'`, `'z'`, `'xy'`, `'xz'`, `'yz'`, or `'unconstrained'`.
- `setAxesPanConstraint(h,ax,cnstr)` — Function to set constraints of pan operations.

  Calling the function `setAxesPanConstraint` on the pan object, `h`, with an axes object, `ax`, and a constraint option, `cnstr`, sets the constraint for the axes. Specify the constraint as one of these values: `'x'`, `'y'`, `'z'`, `'xy'`, `'xz'`, `'yz'`, or `'unconstrained'`.

- `sty = getAxes3DPanAndZoomStyle(h,ax)` — Function to get style of pan operations.

  Calling the function `getAxes3DPanAndZoomStyle` on the pan object, `h`, with a vector of axes handles, `ax`, as input returns the style of panning for each axes. The returned value for each axes is either `'limits'` or `'camera'`.
- `setAxes3DPanAndZoomStyle(h,ax,style)` — Function to set style of pan operations.

  Calling the function `setAxes3DPanAndZoomStyle` on the pan object, `h`, with a vector of axes handles, `ax`, and a character array, `style`, sets the style of panning on each axes. Specify the style as either `'limits'` or `'camera'`.
- `cns = getAxesPanMotion(h,ax)` — Function to get constraints of pan operations (not recommended, use `getAxesPanConstraint`).

  Calling the function `getAxesPanMotion` on the pan object, `h`, with a vector of axes objects, `ax`, as input returns a character cell array of the same dimension as `ax`, which indicates the constraint for each axes. The returned value for each axes is `'horizontal'`, `'vertical'` or `'both'`.
- `setAxesPanMotion(h,ax,constraints)` — Function to set constraints of pan operations (not recommended, use `setAxesPanConstraint`).

  Calling the function `setAxesPanMotion` on the pan object, `h`, with a vector of axes objects, `ax`, and a character array, `constraints`, sets the constraint for each axes. Specify the constraints as `'horizontal'`, `'vertical'` or `'both'`.

Examples

Example 1 — Entering Pan Mode

Plot a graph and turn on Pan mode:

```
plot(magic(10));
pan on
% pan on the plot
```

Example 2 — Constrained Pan

Constrain pan to *x*-axis using `set`:

```
plot(magic(10));
h = pan;
h.Motion = 'horizontal';
h.Enable = 'on';
% pan on the plot in the horizontal direction.
```

Example 3 — Constrained Pan in Subplots

Create four axes as subplots and give each one a different panning behavior:

```
ax1 = subplot(2,2,1);
plot(1:10);
h = pan;
ax2 = subplot(2,2,2);
plot(rand(3));
setAllowAxesPan(h,ax2,false);
ax3 = subplot(2,2,3);
plot(peaks);
setAxesPanMotion(h,ax3,'horizontal');
ax4 = subplot(2,2,4);
contour(peaks);
setAxesPanMotion(h,ax4,'vertical');
% pan on the plots.
```

Example 4 — Coding a ButtonDown Callback

Create a buttonDown callback for pan mode objects to trigger. Copy the following code to a new file, execute it, and observe panning behavior:

```
function demo
% Allow a line to have its own 'ButtonDownFcn' callback.
hLine = plot(rand(1,10));
hLine.ButtonDownFcn = 'disp(''This executes'')';
hLine.Tag = 'DoNotIgnore';
h = pan;
h.ButtonDownFilter = @mycallback;
h.Enable = 'on';
% mouse click on the line
%
function [flag] = mycallback(obj,event_obj)
% If the tag of the object is 'DoNotIgnore', then
% return true.
```

```
% Indicate what the target is.
disp(['Clicked ' obj.Type ' object'])
objTag = obj.Tag;
if strcmpi(objTag,'DoNotIgnore')
   flag = true;
else
   flag = false;
end
```

Example 5 — Coding Pre- and Post-Callback Behavior

Create callbacks for pre- and post-ButtonDown events for pan mode objects to trigger. Copy the following code to a new file, execute it, and observe panning behavior:

```
function demo
% Listen to pan events
plot(1:10);
h = pan;
h.ActionPreCallback = @myprecallback;
h.ActionPostCallback = @mypostcallback;
h.Enable = 'on';
%
function myprecallback(obj,evd)
disp('A pan is about to occur.');
%
function mypostcallback(obj,evd)
newLim = evd.Axes.XLim;
msgbox(sprintf('The new X-Limits are [%.2f,%.2f].',newLim));
```

Example 6 — Creating a Context Menu for Pan Mode

Coding a context menu that lets the user to switch to zoom mode by right-clicking:

```
figure
plot(magic(10));
hCM = uicontextmenu;
hMenu = uimenu('Parent',hCM,'Label','Switch to zoom',...
        'Callback','zoom(gcbf,''on'')');
hPan = pan(gcf);
hPan.UIContextMenu = hCM;
pan('on')
```

You cannot add items to the built-in pan context menu, but you can replace it with your own.

Tips

You can create a pan mode object once and use it to customize the behavior of different axes, as Example 3 illustrates. You can also change its callback functions on the fly.

Note **Do not change figure callbacks within an interactive mode.** While a mode is active (when panning, zooming, etc.), you will receive a warning if you attempt to change any of the figure's callbacks and the operation will not succeed. The one exception to this rule is the figure `WindowButtonMotionFcn` callback, which can be changed from within a mode. Therefore, if you are creating a UI that updates a figure's callbacks, the UI should some keep track of which interactive mode is active, if any, before attempting to do this.

When you assign different pan behaviors to different `subplot` axes via a mode object and then link them using the `linkaxes` function, the behavior of the axes you manipulate with the mouse carries over to the linked axes, regardless of the behavior you previously set for the other axes.

Alternatives

Use the Pan tool on the toolbar to enable and disable pan mode on a plot, or select **Pan** from the figure's **Tools** menu.

See Also

`linkaxes` | `rotate3d` | `zoom`

Topics

"Interactively Explore Plotted Data"

Introduced before R2006a

panInteraction

Pan interaction

Description

A pan interaction allows you to pan within a chart without having to select any buttons in the axes toolbar. To enable panning, set the `Interactions` property of the axes to a `panInteraction` object. When this interaction is enabled, you can pan by dragging within the chart.

To enable multiple interactions, set the `Interactions` property to an array of objects.

Creation

Syntax

```
p = panInteraction
p = panInteraction('Dimensions',d)
```

Description

`p = panInteraction` creates a pan interaction object.

`p = panInteraction('Dimensions',d)` sets the `Dimensions` property. Use this property to constrain panning to specific dimensions. For example, `p = panInteraction('Dimensions','x')` constrains panning to the *x*-dimension.

Properties

`Dimensions` — Dimensions
`'xyz'` (default) | `'xy'` | `'yz'` | `'xz'` | `'x'` | `'y'` | `'z'`

Dimensions to allow panning, specified as one of these values:

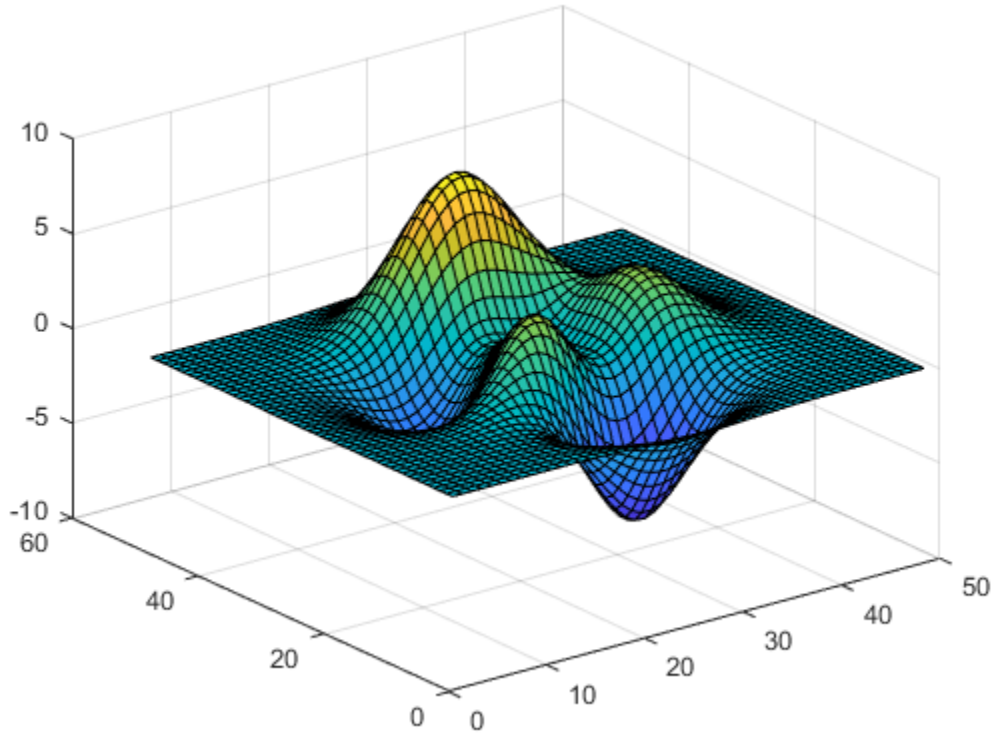
- `'xyz'` — Allows panning in all dimensions.
- `'xy'` — Allows panning in the *x* and *y* dimensions only.
- `'yz'` — Allows panning in the *y* and *z* dimensions only.
- `'xz'` — Allows panning in the *x* and *z* dimensions only.
- `'x'` — Allows panning in the *x* dimension only.
- `'y'` — Allows panning in the *y* dimension only.
- `'z'` — Allows panning in the *z* dimension only.

Examples

Axes with Pan and Data Tip Interactions

Create a surface plot. Get the current axes and replace the default interactions with the pan and data tip interactions. Then hover over the surface to display data tips. Click and drag or tap and drag to pan.

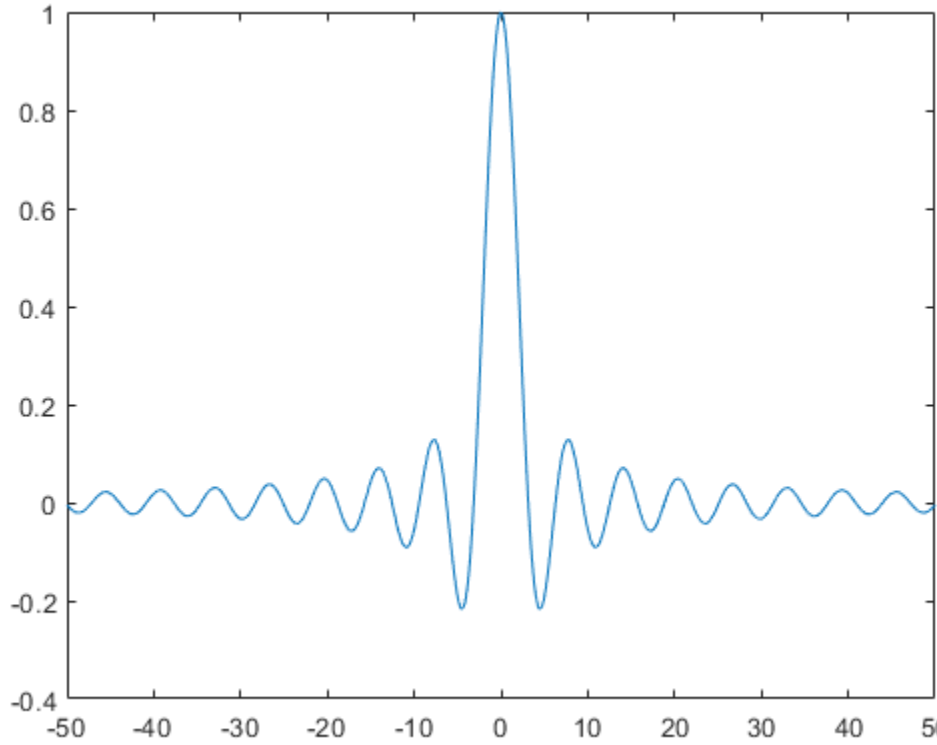
```
surf(peaks)
ax = gca;
ax.Interactions = [panInteraction dataTipInteraction];
```



Restrict Panning to X-Dimension

Plot a set of *x* and *y* values, and adjust the x-axis limits to narrow the view of the plot. Then replace the default set of interactions with a pan interaction that is restricted to the *x*-dimension. Drag within the plot to pan. Notice that you cannot pan vertically.

```
x = linspace(-500,500,5000);
y = sin(x)./x;
plot(x,y)
xlim([-50 50])
ax = gca;
ax.Interactions = panInteraction('Dimensions','x');
```



Limitations

The `panInteraction` function is not supported in the Live Editor.

Tips

In most cases, the axes have a default set of interactions which depend on the type of chart you are displaying. You can replace the default set with a new set of interactions, but you cannot access or modify any of the interactions in the default set.

See Also

`dataTipInteraction` | `disableDefaultInteractivity` | `enableDefaultInteractivity` | `regionZoomInteraction` | `rotateInteraction` | `rulerPanInteraction` | `zoomInteraction`

Topics

"Control Chart Interactivity"

Introduced in R2019a

ParallelCoordinatesPlot Properties

Control parallel coordinates plot appearance and behavior

Description

`ParallelCoordinatesPlot` properties control the appearance and behavior of a `ParallelCoordinatesPlot` object. By changing property values, you can modify certain aspects of the plot display. For example, you can add a title:

```
p = parallelplot(rand(10,4));
p.Title = 'My Title';
```

Properties

Labels

`Title` — Plot title

`''` (default) | character vector | string array | cell array of character vectors | categorical array

Plot title, specified as a character vector, string array, cell array of character vectors, or categorical array. By default, the plot has no title.

To create a multiline title, specify a string array or cell array of character vectors. Each element in the array corresponds to a line of text.

If you specify the title as a categorical array, MATLAB uses the values in the array, not the categories.

Example: `p = parallelplot(__,'Title','My Title Text')`

Example: `p.Title = 'My Title Text'`

Example: `p.Title = {'My','Title'}`

`CoordinateLabel` — Horizontal axis label

`''` (default) | character vector | string array | cell array of character vectors | categorical array

Horizontal axis label for describing the set of coordinate variables, specified as a character vector, string array, cell array of character vectors, or categorical array. The `CoordinateLabel` value appears below the `CoordinateTickLabels` value in the plot.

To create a multiline label, specify a string array or cell array of character vectors. Each element in the array corresponds to a line of text.

If you specify the label as a categorical array, MATLAB uses the values in the array, not the categories.

Example: `p = parallelplot(__,'CoordinateLabel','Measurements')`

Example: `p.CoordinateLabel = 'Measurements'`

Example: `p.CoordinateLabel = {'Measurements','(in inches)'}`

`CoordinateTickLabels` — Coordinate variable labels
string array | cell array of character vectors | categorical array

Coordinate variable labels, specified as a string array, cell array of character vectors, or categorical array. Each coordinate tick label appears directly below the ruler displaying the values for that coordinate variable.

If you specify the labels as a categorical array, MATLAB uses the values in the array, not the categories.

Example: `p = parallelplot(__,'CoordinateTickLabels', {'Height','Age','Smoker'})`

Example: `p.CoordinateTickLabels = {'Height','Age','Smoker'}`

`DataLabel` — Vertical axis label
`''` (default) | character vector | string array | cell array of character vectors | categorical array

Vertical axis label for describing the rows of the data in `SourceTable` or `Data`, specified as a character vector, string array, cell array of character vectors, or categorical array.

To create a multiline label, specify a string array or cell array of character vectors. Each element in the array corresponds to a line of text.

If you specify the label as a categorical array, MATLAB uses the values in the array, not the categories.

Example: `p = parallelplot(__,'DataLabel','Patients')`

Example: `p.DataLabel = 'Patients'`

Example: `p.DataLabel = {'Patients','(data set 1)'}`

`LegendTitle` — Legend title
character vector | string array | cell array of character vectors | categorical array

Legend title, specified as a character vector, string array, cell array of character vectors, or categorical array. Use `''` for no title.

To create a multiline title, specify a string array or cell array of character vectors. Each element in the array corresponds to a line of text.

If you specify the title as a categorical array, MATLAB uses the values in the array, not the categories.

Example: `p = parallelplot(__,'LegendTitle','My Title Text')`

Example: `p.LegendTitle = 'My Title Text'`

Example: `p.LegendTitle = {'My','Title'}`

`LegendVisible` — Visibility of legend
`'on'` | `'off'`

Visibility of legend, specified as `'on'` or `'off'`. Set `LegendVisible` to `'on'` to display the legend or `'off'` to hide the legend.

If `GroupData` is empty (`[]`), then `parallelplot` does not display a legend. Otherwise, `parallelplot` displays a legend by default.

In the legend, `parallelplot` displays the group names in order of their first appearance in `GroupData`.

Example: `p = parallelplot(__,'LegendVisible','off')`

Example: `p.LegendVisible = 'off'`

Data Display

`DataNormalization` — Normalization method for coordinates
`'range'` (default) | `'none'` | `'zscore'` | `'scale'` | `'center'` | `'norm'`

Normalization method for coordinates with numeric values, specified as one of the following options.

Method	Description
`'range'`	Display raw data along coordinate rulers that have independent minimum and maximum limits
`'none'`	Display raw data along coordinate rulers that have the same minimum and maximum limits
`'zscore'`	Display z-scores (with a mean of 0 and a standard deviation of 1) along each coordinate ruler
`'scale'`	Display values scaled by standard deviation along each coordinate ruler
`'center'`	Display data centered to have a mean of 0 along each coordinate ruler
`'norm'`	Display 2-norm values along each coordinate ruler

For more information about these methods, see `normalize`.

For a coordinate variable that is a logical vector, datetime array, duration array, categorical array, string array, or cell array of character vectors, `parallelplot` evenly distributes the unique possible values along the coordinate ruler, regardless of the normalization method.

Example: `p = parallelplot(__,'DataNormalization','none')`

Example: `p.DataNormalization = 'zscore'`

`Jitter` — Data displacement distance

`0.1` (default) | numeric scalar in the interval [0,1]

Data displacement distance along the coordinate rulers, specified as a numeric scalar in the interval [0,1]. The `Jitter` value determines the maximum distance to displace plot lines from their true value along the coordinate rulers, where the displacement is a uniform random amount. If you set the `Jitter` property to `1`, then adjacent jitter regions just touch. Set the `Jitter` property to `0` to display the true data values.

Some amount of jitter is particularly helpful for visualizing categorical data because the jittering enables you to distinguish between plot lines more easily. However, the `Jitter` value affects all coordinate variables, including numeric variables.

Example: `p = parallelplot(__,'Jitter',0.5)`

Example: `p.Jitter = 0.2`

Color and Styling

`Color` — Group color

character vector | string array | cell array of character vectors | matrix of RGB values

Group color, specified in one of these forms:

- Character vector designating a color name, short name, or hexadecimal color code. A hexadecimal color code starts with a hash symbol (`#`) and is followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.
- String array or cell array of character vectors designating one or more color names, short names, or hexadecimal color codes.
- Three-column matrix of RGB values in the range [0,1]. The three columns represent the R value, G value, and B value.

Choose among these predefined colors, their equivalent RGB triplets, and their hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

By default, `parallelplot` assigns a maximum of seven unique group colors. When the total number of groups exceeds the number of specified colors, `parallelplot` cycles through the specified colors.

Example: `p = parallelplot(__,'Color',{'blue','black','green'})`

Example: `p.Color = [0 0 1; 0 0.5 0.5; 0.5 0.5 0.5]`

Example: `p.Color = {'#EDB120','#77AC30','#7E2F8E'}`

`LineStyle` — Group line style

character vector | string array | cell array of character vectors

Group line style, specified in one of these forms:

- Character vector designating one line style
- String array or cell array of character vectors designating one or more line styles

Choose among these line style options.

Line Style	Description	Resulting Line
'-'	Solid line	
'--'	Dashed line	
':'	Dotted line	
'-.'	Dash-dotted line	

Line Style	Description	Resulting Line
`'none'`	No line	No line

By default, `parallelplot` assigns a line style of `'-'` to each group of plot lines. When the total number of groups exceeds the number of specified line styles, `parallelplot` cycles through the specified line styles.

Example: `p = parallelplot(__,'LineStyle',':')`

Example: `p.LineStyle = {'-.','-','--'}`

`LineWidth` — Group line width

positive scalar | positive vector

Group line width, specified as a positive scalar or positive vector in points. By default, `parallelplot` assigns a line width of `1` to each group of plot lines.

When the total number of groups exceeds the number of specified line widths, `parallelplot` cycles through the specified line widths.

Example: `p = parallelplot(__,'LineWidth',1.5)`

Example: `p.LineWidth = [0.5 1.5 0.5]`

`LineAlpha` — Group line transparency

numeric scalar | numeric vector

Group line transparency, specified as a numeric scalar or numeric vector with values between 0 and 1. Values closer to 0 specify more transparent lines, and values closer to 1 specify more opaque lines. By default, `parallelplot` assigns a `LineAlpha` value of `0.7` to each group of plot lines.

Example: `p = parallelplot(__,'LineAlpha',0.75)`

Example: `p.LineAlpha = [0.2 0.7 0.4]`

`MarkerStyle` — Group marker symbol

character vector | string array | cell array of character vectors

Group marker symbol for data values along the coordinate rulers, specified in one of these forms:

- Character vector designating a marker style

- String array or cell array of character vectors designating one or more marker styles

Choose among these marker options.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

By default, `parallelplot` assigns `'none'` as the marker style for each group. When the total number of groups exceeds the number of specified symbols, `parallelplot` cycles through the specified symbols.

Example: `p = parallelplot(__,'MarkerStyle','x')`

Example: `p.MarkerStyle = {'x','o'}`

`MarkerSize` — Group marker size

nonnegative scalar | nonnegative vector

Group marker size for data values along the coordinate rulers, specified as a nonnegative scalar or nonnegative vector, with values measured in points. By default, `parallelplot` assigns `6` as the marker size for each group. When the total number of groups exceeds the number of specified values, `parallelplot` cycles through the specified values.

Example: `p = parallelplot(__,'MarkerSize',10)`

Example: `p.MarkerSize = [4 6 8]`

Font

`FontName` — Font name

system-supported font name

Font name, specified as a system-supported font name. The same font is used for the title, labels, legend title, and group names. The default font depends on the specific operating system and locale.

Example: `p = parallelplot(__,'FontName','Cambria')`

Example: `p.FontName = 'Helvetica'`

`FontSize` — Font size

positive scalar

Font size, specified as a positive scalar. `FontSize` is the same for the title, labels, legend title, and group names. The default font size depends on the specific operating system and locale.

As you adjust the size of plot elements, the software automatically updates the font size. However, changing the `FontSize` property disables this automatic resizing.

Example: `p = parallelplot(__,'FontSize',12)`

Example: `p.FontSize = 10`

Position

`ActivePositionProperty` — Position property to hold constant

`'outerposition'` | `'innerposition'`

Position property to hold constant when adding, removing, or changing decorations, specified as one of the following values:

- `'outerposition'` — The `OuterPosition` property remains constant when you add, remove, or change decorations such as a title or an axis label. If any positional adjustments are needed, MATLAB adjusts the `InnerPosition` property.
- `'innerposition'` — The `InnerPosition` property remains constant when you add, remove, or change decorations such as a title or an axis label. If any positional adjustments are needed, MATLAB adjusts the `OuterPosition` property.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`InnerPosition` — Inner size and position

four-element numeric vector

Inner size and position of the plot within the parent container (typically a figure, panel, or tab), specified as a four-element numeric vector of the form `[left bottom width height]`. The inner position does not include the plot title, labels, or legend.

- The `left` and `bottom` elements define the distance from the lower left corner of the container to the lower left corner of the plot.
- The `width` and `height` elements are the dimensions of the plot.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`OuterPosition` — Outer size and position

`[0 0 1 1]` (default) | four-element numeric vector

Outer size and position of the full parallel coordinates plot within the parent container (typically a figure, panel, or tab), specified as a four-element numeric vector of the form `[left bottom width height]`. The default value of `[0 0 1 1]` includes the whole interior of the container.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`Position` — Inner size and position

four-element numeric vector

Inner size and position of the plot within the parent container (typically a figure, panel, or tab), specified as a four-element numeric vector of the form `[left bottom width height]`. This property is equivalent to the `InnerPosition` property.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`Units` — Position units

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'points'` | `'pixels'` | `'characters'`

Position units, specified as one of these values.

Value	Description
`'normalized'` (default)	Normalized with respect to the container, which is typically the figure or a panel. The lower left corner of the container maps to `(0,0)`, and the upper right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the default uicontrol font of the graphics root object: • Character width is equal to the width of the letter `x` • Character height is equal to the distance between the baselines of two lines of text
`'points'`	Typography points. One point equals 1/72 inch.

Value	Description
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96 inch • On Macintosh systems, a pixel is 1/72 inch On Linux systems, the size of a pixel is determined by your system resolution.

When specifying the units as a name-value pair during object creation, you must set the `Units` property before specifying the properties that you want to use these units for, such as `OuterPosition`.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `TiledChartLayoutOptions` object

Layout options, specified as a `TiledChartLayoutOptions` object. This property specifies options when the chart is a child of a tiled chart layout. Use this property to position the chart within the layout by setting the `Tile` and `TileSpan` properties on the `TiledChartLayoutOptions` object.

For example, this code places chart `c` in the third tile of a tiled chart layout.

```
c.Layout.Tile = 3;
```

To make the chart span multiple tiles, specify the `TileSpan` property as a two-element vector. For example, this chart spans `2` rows and `3` columns of tiles.

```
c.Layout.TileSpan = [2 3];
```

If the chart is not a child of a tiled chart layout (for example, if it is a child of a figure or panel), then this property is empty and has no effect.

`Visible` — Visibility of object
`'on'` (default) | `'off'`

Visibility of object, specified as one of these values:

- `'on'` — Display the `ParallelCoordinatesPlot` object.
- `'off'` — Hide the `ParallelCoordinatesPlot` object without deleting it. You can still access the properties of an invisible `ParallelCoordinatesPlot` object.

Table Data

`SourceTable` — Source table

table

Source table, specified as a table.

You can create a table from workspace variables using the `table` function, or you can import data as a table using the `readtable` function.

Note The property is ignored and read-only when you use matrix data instead of tabular data.

`CoordinateVariables` — Table variables to display as coordinates

numeric vector | string array | cell array of character vectors | logical vector

Table variables to display as coordinates, specified in one of these forms:

- Numeric vector — Indicating the indices of the table variables
- String array or cell array of character vectors — Indicating the names of the table variables
- Logical vector — Containing `true` elements for the selected table variables

If you set the `CoordinateVariables` property value, then the `CoordinateData` and `CoordinateTickLabels` properties update to appropriate values.

Note The property is ignored and read-only when you use matrix data instead of tabular data.

Example: `p.CoordinateVariables = {'Age','Height','Weight'}` specifies the variables named `'Age'`, `'Height'`, and `'Weight'`.

`GroupVariable` — Table variable for grouping data

character vector | string scalar | numeric scalar | logical vector

Table variable for grouping data, specified in one of these forms:

- Character vector or string scalar — Indicating one of the table variable names
- Numeric scalar — Indicating the table variable index
- Logical vector — Containing one `true` element for the table variable

The values associated with your table variable must form a numeric vector, logical vector, categorical array, string array, or cell array of character vectors.

`GroupVariable` splits the rows in `SourceTable` into unique groups. `parallelplot` assigns each group a default color. In the legend, `parallelplot` displays the group names in order of their first appearance in `GroupData`.

When you specify `GroupVariable`, MATLAB updates the `GroupData` property values.

Note This property is ignored and read-only when you use matrix data instead of tabular data.

Example: `p.GroupVariable = 'Smoker'`

Matrix Data

`Data` — Input data

numeric matrix

Input data, specified as a numeric matrix.

Note If you are using tabular data, you cannot set this property.

`CoordinateData` — Variables displayed as coordinates

numeric vector | logical vector

Variables displayed as coordinates, specified in one of these forms:

- Numeric vector — Indicating the columns of the source table or input data matrix

- Logical vector — Containing `true` elements for the selected columns of the source table or input data matrix

Note If you are using tabular data, you cannot set this property. The value of `CoordinateData` automatically updates based on the table variables you select with the `CoordinateVariables` property.

Example: `p.CoordinateData = [1 5:7]`

`GroupData` — Values for grouping data

numeric vector | logical vector | categorical array | string array | cell array of character vectors

Values for grouping data, specified as a numeric vector, logical vector, categorical array, string array, or cell array of character vectors.

`GroupData` splits the rows in `SourceTable` or `Data` into unique groups. `parallelplot` assigns each group a default color. In the legend, `parallelplot` displays the group names in order of their first appearance in `GroupData`.

Note If you are using tabular data, you cannot set this property. The `GroupData` values automatically populate based on the table variable you select with the `GroupVariable` property.

Example: `p.GroupData = [1 2 1 3 2 1 3 3 2 3]`

Example: `p.GroupData = categorical({'blue','red','yellow','blue','yellow','red','red','yellow','blue','red'})`

Parent/Child

`Parent` — Parent container

`Figure` object | `Panel` object | `Tab` object | `TiledChartLayout` object

Parent container, specified as a `Figure`, `Panel`, `Tab`, or `TiledChartLayout` object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle for `ScatterHistogramChart` in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is always invisible. This option is useful for preventing unintended changes to the UI by another function. To temporarily hide the handle during the execution of that function, set the `HandleVisibility` to `'off'`.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but allows callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return the object. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles, regardless of their `HandleVisibility` property setting.

See Also

`parallelplot`

Topics

"Access Property Values"

Introduced in R2019a

parallelplot

Create parallel coordinates plot

Syntax

```
p = parallelplot(tbl)
p = parallelplot(tbl,'CoordinateVariables',coordvars)
p = parallelplot( ___ ,'GroupVariable',grpvar)

p = parallelplot(data)
p = parallelplot(data,'CoordinateData',coorddata)
p = parallelplot( ___ ,'GroupData',grpdata)

p = parallelplot( ___ ,Name,Value)
p = parallelplot(parent, ___ )
```

Description

`p = parallelplot(tbl)` creates a parallel coordinates plot from the table `tbl` and returns a `ParallelCoordinatesPlot` object. Each line in the plot represents a row in the table, and each coordinate variable in the plot corresponds to a column in the table. The software plots all table columns by default.

Use `p` to modify the object after you create it. For a list of properties, see ParallelCoordinatesPlot.

`p = parallelplot(tbl,'CoordinateVariables',coordvars)` creates a parallel coordinates plot from the `coordvars` variables in the table `tbl`.

`p = parallelplot( ___ ,'GroupVariable',grpvar)` uses the table variable specified by `grpvar` to group the lines in the plot. Specify this option after any of the input argument combinations in the previous syntaxes.

`p = parallelplot(data)` creates a parallel coordinates plot from the numeric matrix `data`.

`p = parallelplot(data,'CoordinateData',coorddata)` creates a parallel coordinates plot from the `coorddata` columns in the matrix `data`.

`p = parallelplot( ___ ,'GroupData',grpdata)` uses the data in `grpdata` to group the lines in the plot. Specify this option after any of the previous input argument combinations for numeric matrix data.

`p = parallelplot( ___ ,Name,Value)` specifies additional options using one or more name-value pair arguments. For example, you can specify the data normalization method for coordinates with numeric values. For a list of properties, see ParallelCoordinatesPlot.

`p = parallelplot(parent, ___ )` creates the parallel coordinates plot in the figure, panel, or tab specified by `parent`.

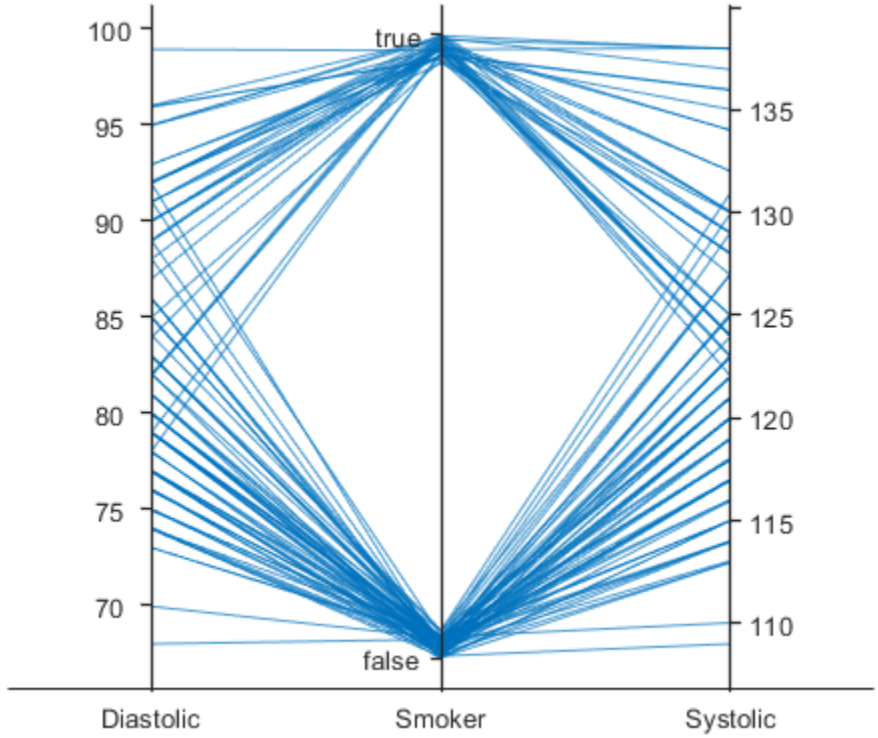
Examples

Parallel Coordinates Plot with Tabular Data

Create a parallel coordinates plot from a table of medical patient data.

Load the `patients` data set, and create a table from a subset of the variables loaded into the workspace. Create a parallel coordinates plot using the table. The lines in the plot correspond to individual patients. Use the plot to observe trends in the data. For example, the plot indicates that smokers tend to have higher blood pressure values (both diastolic and systolic).

```
load patients
tbl = table(Diastolic,Smoker,Systolic);
p = parallelplot(tbl)
```



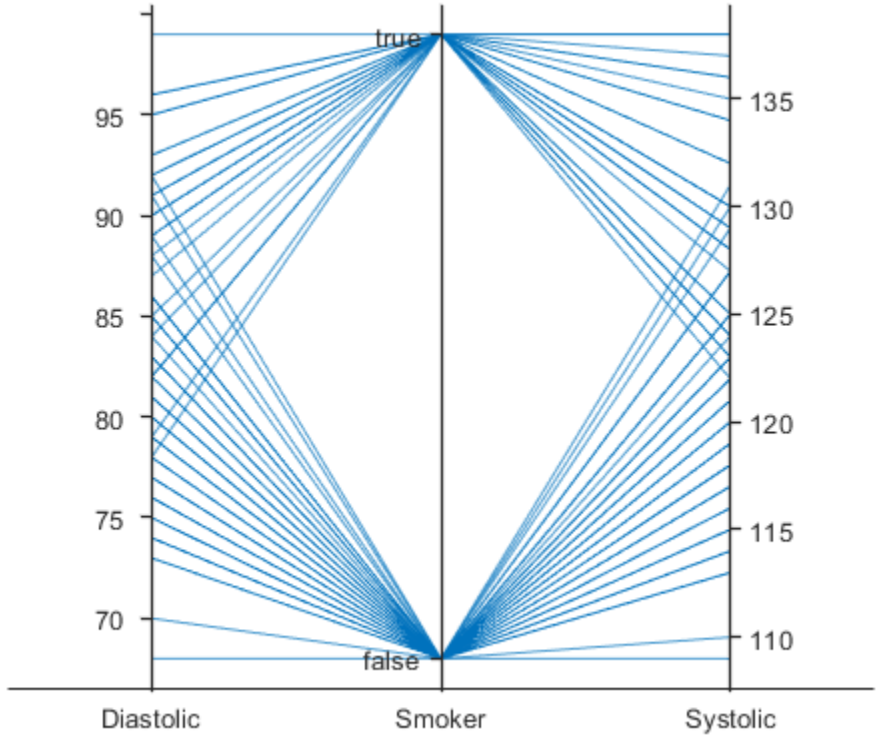

```
p =
  ParallelCoordinatesPlot with properties:

            SourceTable: [100x3 table]
    CoordinateVariables: {'Diastolic'  'Smoker'  'Systolic'}
          GroupVariable: ''

  Show all properties
```

By default, the software randomly jitters plot lines so that they are unlikely to overlap perfectly along coordinate rulers. This jittering is particularly helpful for visualizing categorical data because it enables you to distinguish between plot lines more easily. For

example, observe the plot lines along the `Smoker` coordinate ruler; the plot lines are not flush with either the `true` or `false` tick marks.

To disable the default jittering, set the `Jitter` property to `0`.

```
p.Jitter = 0;
```



Specify Coordinate and Group Variables

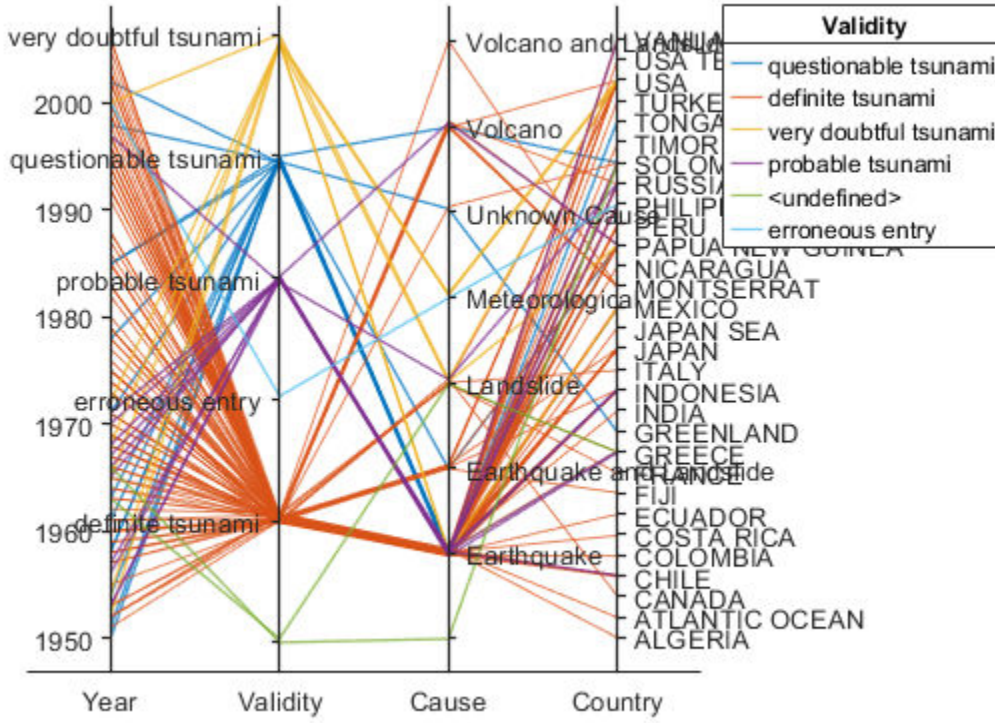
Create a parallel coordinates plot from a table of tsunami data. Specify the table variables to display and their order, and group the lines in the plot according to one of the variables.

Read the tsunami data into the workspace as a table.

```
tsunamis = readtable('tsunamis.xlsx');
```

Create a parallel coordinates plot using a subset of the variables in the table. First, increase the figure window size to prevent overcrowding in the plot. Then, to specify the variables and their order, use the `'CoordinateVariables'` name-value pair argument. To group occurrences according to their validity, set the `'GroupVariable'` name-value pair argument to `'Validity'`. The lines in the plot correspond to individual tsunami occurrences. The plot indicates that most of the occurrences in the data set that have a `Validity` value are considered definite tsunamis.

```
figure('Units','normalized','Position',[0.3 0.3 0.45 0.4])
coordvars = {'Year','Validity','Cause','Country'};
p = parallelplot(tsunamis,'CoordinateVariables',coordvars,'GroupVariable','Validity');
```



Parallel Coordinates Plot with Binned Data

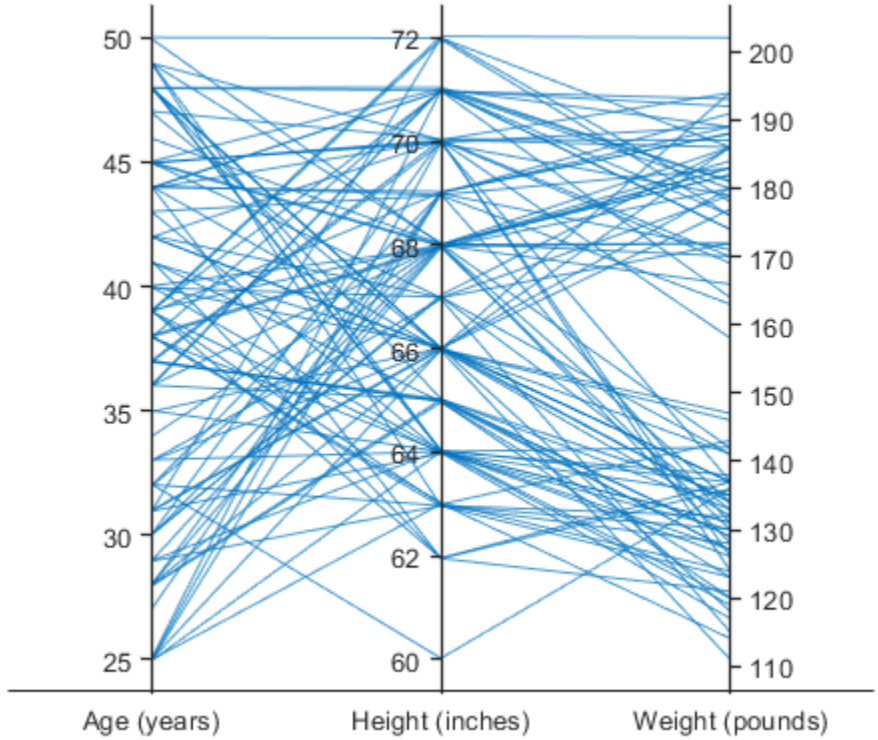
Create a parallel coordinates plot from a matrix containing medical patient data. Bin the values in one of the columns in the matrix, and group the lines in the plot using the binned values.

Load the `patients` data set, and create a matrix from the `Age`, `Height`, and `Weight` values. Create a parallel coordinates plot using the matrix data. Label the coordinate variables in the plot. The lines in the plot correspond to individual patients.

```
load patients
X = [Age Height Weight];
p = parallelplot(X)
```

```
p = 
  ParallelCoordinatesPlot with properties:

              Data: [100x3 double]
    CoordinateData: [1 2 3]
         GroupData: []

  Show all properties
```

```
p.CoordinateTickLabels = {'Age (years)','Height (inches)','Weight (pounds)'};
```



Create a new categorical variable that groups each patient into one of three categories: `short`, `average`, or `tall`. Set the bin edges such that they include the minimum and maximum `Height` values.

```
min(Height)
```

```
ans = 60
```

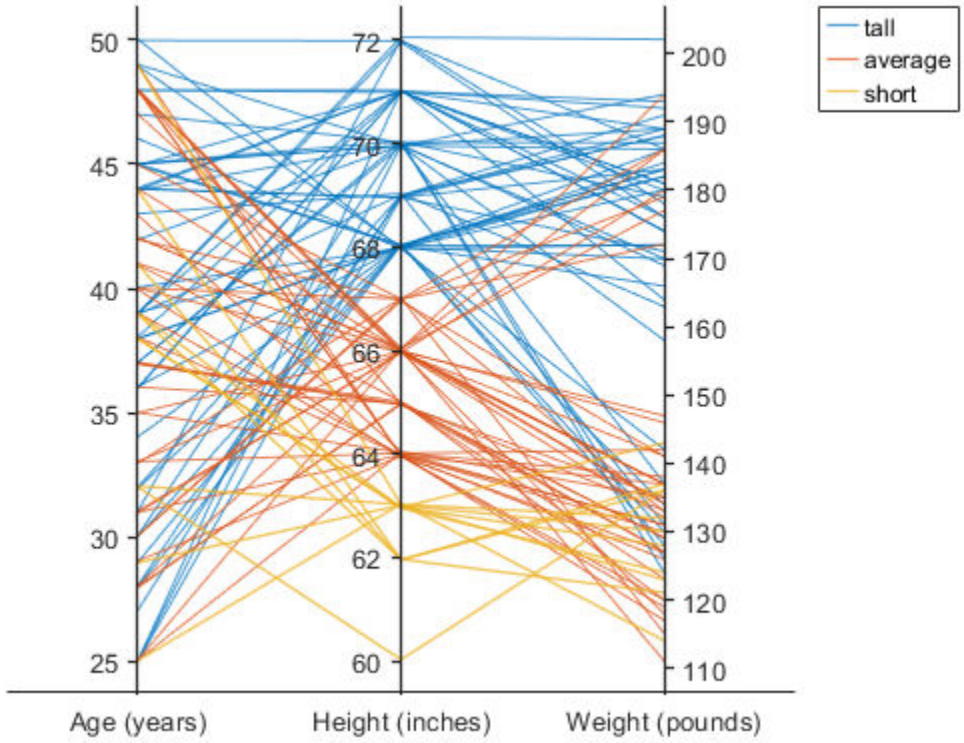
```
max(Height)
```

```
ans = 72
```

```
binEdges = [60 64 68 72];
bins = {'short','average','tall'};
groupHeight = discretize(Height,binEdges,'categorical',bins);
```

Now use the `groupHeight` values to group the lines in the parallel coordinates plot. The plot indicates that `short` patients tend to weigh less than `tall` patients.

```
p.GroupData = groupHeight;
```



Specify Coordinate and Group Data

Create parallel coordinates plots from a matrix containing medical patient data. For each plot, specify the columns of the matrix to display, and group the lines in the plot according to a separate variable.

Load the `patients` data set, and create a matrix from some of the variables loaded into the workspace.

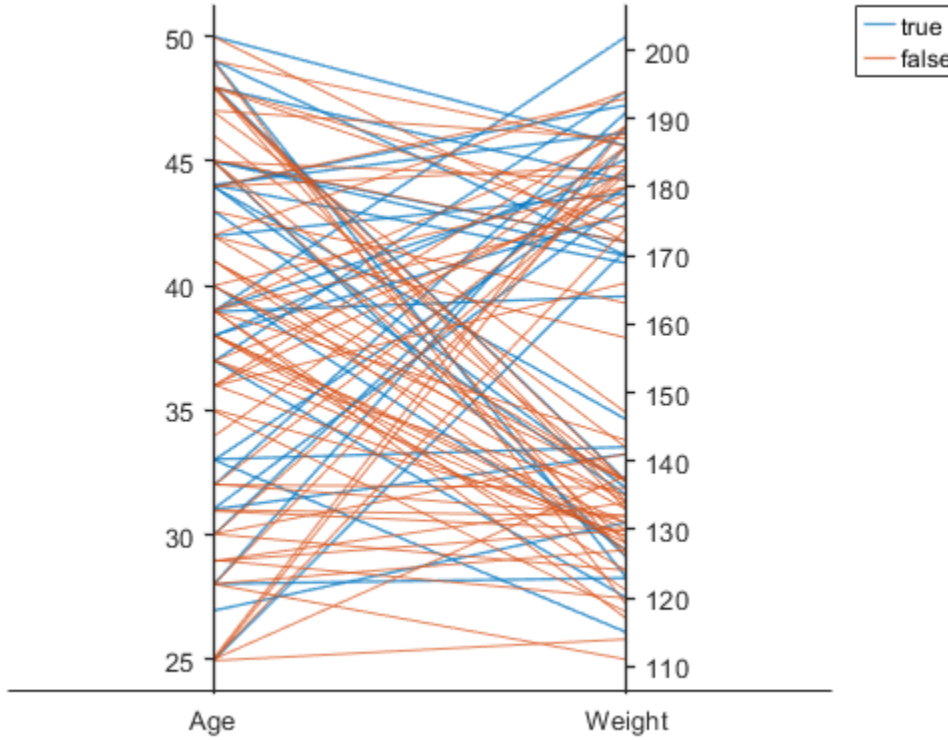
```
load patients
X = [Age Height Weight];
```

Create a parallel coordinates plot using a subset of the columns in the matrix `X`. To specify the columns and their order, use the `'CoordinateData'` name-value pair argument. Group patients according to their smoker status by passing the `Smoker` values to the `'GroupData'` name-value pair argument. The lines in the plot correspond to individual patients. The plot indicates that no clear relationship exists between smoker status and either age or weight.

```
coorddata = [1 3];
p = parallelplot(X,'CoordinateData',coorddata,'GroupData',Smoker)
```

```
p = 
  ParallelCoordinatesPlot with properties:

              Data: [100x3 double]
    CoordinateData: [1 3]
         GroupData: [100x1 logical]

  Show all properties
```

```
p.CoordinateTickLabels = {'Age','Weight'};
```

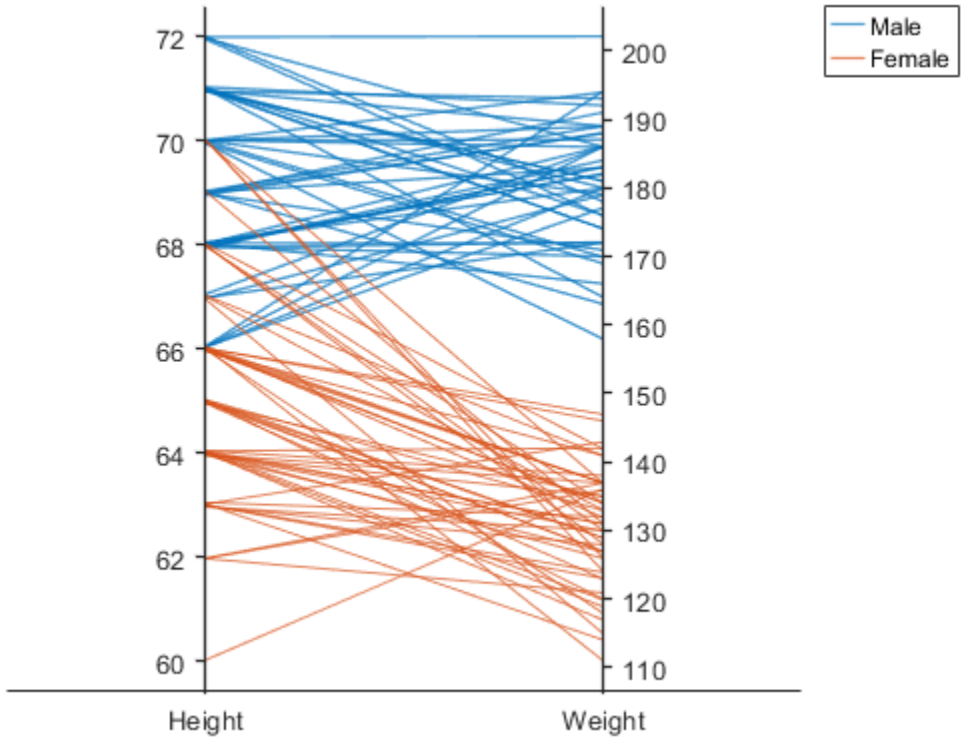


Create another parallel coordinates plot using a different subset of the columns in X. Group the patients according to their gender. The plot indicates that the men are taller and weigh more than the women.

```
coorddata2 = [2 3];
p2 = parallelplot(X,'CoordinateData',coorddata2,'GroupData',Gender)
```

```
p2 = 
  ParallelCoordinatesPlot with properties:

              Data: [100x3 double]
    CoordinateData: [2 3]
         GroupData: {100x1 cell}
```

```
Show all properties

p2.CoordinateTickLabels = {'Height','Weight'};
```



Change Data Normalization in Plot

Create a parallel coordinates plot from a table of power outage data. Change the normalization method for the numeric coordinate variables.

Read the power outage data into the workspace as a table. Display the first few rows of the table.

```
outages = readtable('outages.csv');
head(outages)
```

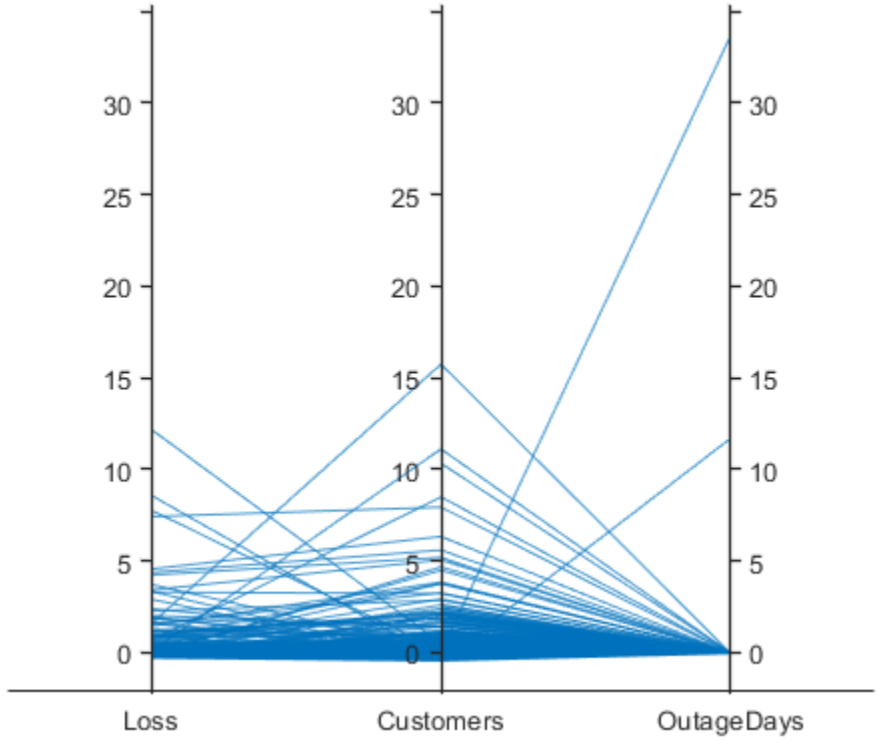
```
ans=8×6 table
       Region           OutageTime        Loss     Customers     RestorationTime
    _____________    _________________    ______    __________    _________________

    {'SouthWest'}    2002-02-01 12:18    458.98    1.8202e+06    2002-02-07 16:50
    {'SouthEast'}    2003-01-23 00:49    530.14    2.1204e+05                 NaT
    {'SouthEast'}    2003-02-07 21:15     289.4    1.4294e+05    2003-02-17 08:14
    {'West'     }    2004-04-06 05:44    434.81    3.4037e+05    2004-04-06 06:10
    {'MidWest'  }    2002-03-16 06:18    186.44    2.1275e+05    2002-03-18 23:23
    {'West'     }    2003-06-18 02:49         0             0    2003-06-18 10:54
    {'West'     }    2004-06-20 14:39    231.29           NaN    2004-06-20 19:16
    {'West'     }    2002-06-06 19:28    311.86           NaN    2002-06-07 00:51
```

Create a new variable called `OutageDuration` that indicates how long each power outage lasted. Convert `OutageDuration` to the number of days each power outage lasted. Add the new variable to the `outages` table, and call it `OutageDays`.

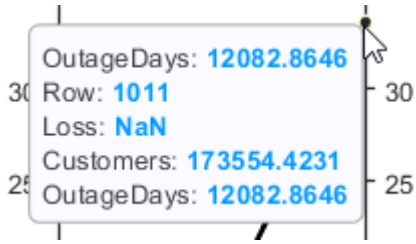
```
OutageDuration = outages.RestorationTime - outages.OutageTime;
outages.OutageDays = days(OutageDuration);
```

Create a parallel coordinates plot using the `Loss`, `Customers`, and `OutageDays` variables. Because the coordinate variables are numeric, display the values in the plot as z-scores, without any jittering, using the `'DataNormalization'` and `'Jitter'` name-value pair arguments.

```
coordvars = {'Loss','Customers','OutageDays'};
p = parallelplot(outages,'CoordinateVariables',coordvars,'DataNormalization','zscore',
```



The `OutageDays` variable contains one value that is more than 30 standard deviations away from the mean `OutageDays` value and another value that is more than 10 standard deviations away from the mean. Hover over the values in the plot to display data tips. Each data tip indicates the row in the table corresponding to the line in the plot.



Find the rows in the `outages` table that have the identified extreme `OutageDays` values. Notice that the `RestorationTime` values for these two power outages are suspicious.

```
outliers = outages([1011 269],:)
```

```
outliers=2×7 table
       Region           OutageTime        Loss     Customers     RestorationTime
    _____________    ________________    ______    __________    ________________    __

    {'NorthEast'}    2009-08-20 02:46       NaN    1.7355e+05    2042-09-18 23:31    {
    {'MidWest'  }    2008-02-07 06:18    2378.7             0    2019-08-14 16:16    {
```

Reorder Categories of Coordinate Variable in Plot

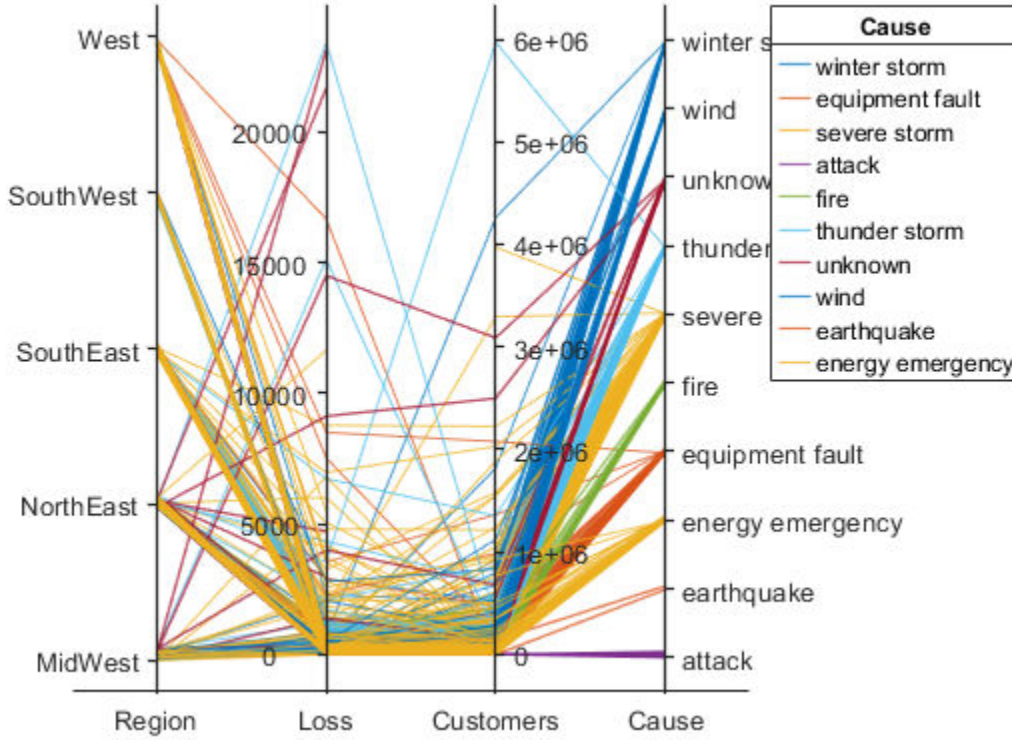
Create a parallel coordinates plot. Reorder the categories of one of the coordinate variables.

Read data on power outages into the workspace as a table.

```
outages = readtable('outages.csv');
```

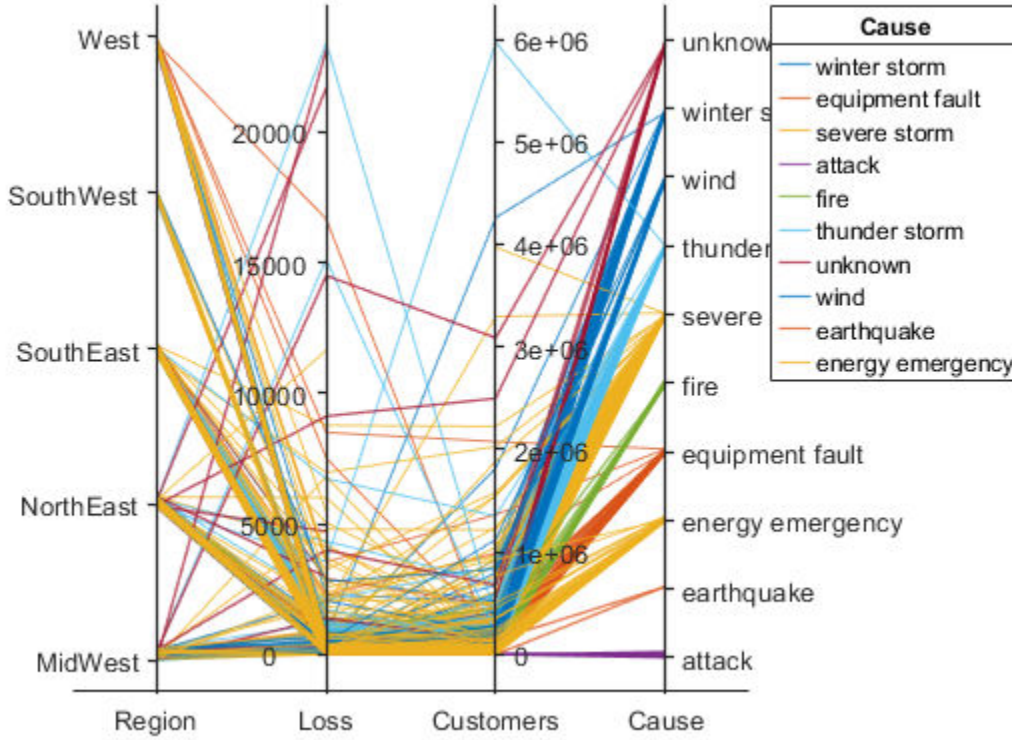
Create a parallel coordinates plot using a subset of the columns in the table. Group the lines in the plot according to the event that caused the power outage.

```
coordvars = [1 3 4 6];
p = parallelplot(outages,'CoordinateVariables',coordvars,'GroupVariable','Cause');
```



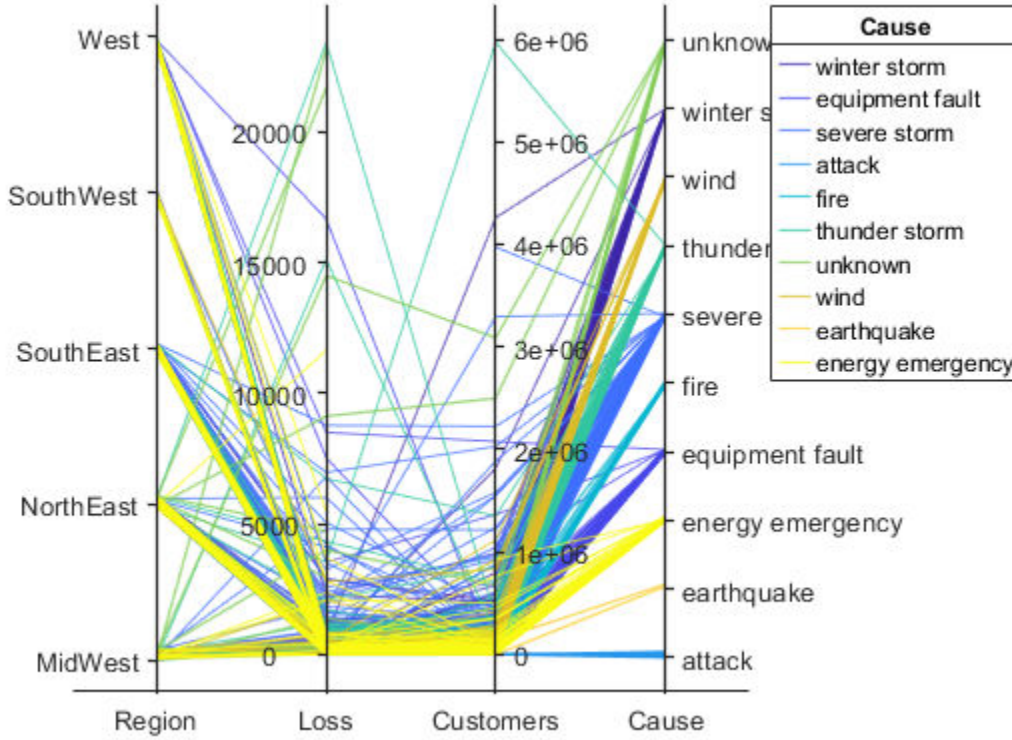
Change the order of the events in `Cause` by updating the source table. First, convert `Cause` to a `categorical` variable, specify the new order of the events, and use the `reordercats` function to create a new variable called `orderCause`. Then, replace the original `Cause` variable with the new `orderCause` variable in the source table of the plot.

```
categoricalCause = categorical(p.SourceTable.Cause);
newOrder = {'attack','earthquake','energy emergency','equipment fault', ...
    'fire','severe storm','thunder storm','wind','winter storm','unknown'};
orderCause = reordercats(categoricalCause,newOrder);
p.SourceTable.Cause = orderCause;
```



Because the `Cause` variable contains more than seven categories, some of the groups have the same color in the plot. Assign distinct colors to every group by changing the `Color` property of `p`.

```
p.Color = parula(10);
```



Input Arguments

`tbl` — Source table
table

Source table, specified as a table.

You can create a table from workspace variables using the `table` function, or you can import data as a table using the `readtable` function.

The `SourceTable` property of the `ParallelCoordinatesPlot` object stores the source table.

`coordvars` — Table variables to display as coordinates

numeric vector | string array | cell array of character vectors | logical vector

Table variables to display as coordinates, specified in one of these forms:

- Numeric vector — Indicating the indices of the table variables. For example, `parallelplot(tbl,'CoordinateVariables',[1 5:7])` selects the first, fifth, sixth, and seventh variables in the table to display as coordinates.
- String array or cell array of character vectors — Indicating the names of the table variables. For example, `parallelplot(tbl,'CoordinateVariables', {'Age','Weight','Height'})` selects the variables named `'Age'`, `'Weight'`, and `'Height'` to display as coordinates.
- Logical vector — Containing `true` elements for the selected table variables.

The `CoordinateVariables` property of the `ParallelCoordinatesPlot` object stores the `coordvars` value. The `CoordinateTickLabels` property stores the selected variable names.

`grpvar` — Table variable for grouping data

character vector | string scalar | numeric scalar | logical vector

Table variable for grouping data, specified in one of these forms:

- Character vector or string scalar — Indicating one of the table variable names
- Numeric scalar — Indicating the table variable index
- Logical vector — Containing one `true` element for the table variable

The values associated with your table variable must form a numeric vector, logical vector, categorical array, string array, or cell array of character vectors.

`grpvar` splits the rows in `tbl` into unique groups. By default, the software colors the associated plot lines according to their group value. Plot lines corresponding to the same group have the same color. However, `parallelplot` assigns a maximum of seven unique group colors. When the total number of groups exceeds the number of specified colors, `parallelplot` cycles through the specified colors.

In the legend, `parallelplot` displays the group names in order of their first appearance in the `GroupData` property of `ParallelCoordinatesPlot`.

Example: `'Smoker'`

Example: 3

data — Input data
numeric matrix

Input data, specified as a numeric matrix.

The `Data` property of the `ParallelCoordinatesPlot` object stores the `data` values.

coorddata — Matrix columns to display as coordinates
numeric vector | logical vector

Matrix columns to display as coordinates, specified in one of these forms:

- Numeric vector — Indicating the columns of the input data matrix. For example, `parallelplot(data,'CoordinateData',[1 5:7])` selects the first, fifth, sixth, and seventh columns in `data` to display as coordinates.
- Logical vector — Containing `true` elements for the selected columns of the input data matrix.

The `CoordinateData` property of the `ParallelCoordinatesPlot` object stores the `coorddata` value.

grpdata — Values for grouping matrix data
numeric vector | logical vector | categorical array | string array | cell array of character vectors

Values for grouping matrix data, specified as a numeric vector, logical vector, categorical array, string array, or cell array of character vectors.

`grpdata` splits the rows in `data` into unique groups. By default, the software colors the associated plot lines according to their group value. Plot lines corresponding to the same group have the same color. However, `parallelplot` assigns a maximum of seven unique group colors. When the total number of groups exceeds the number of specified colors, `parallelplot` cycles through the specified colors.

In the legend, `parallelplot` displays the group names in order of their first appearance in the `GroupData` property of `ParallelCoordinatesPlot`.

Example: `[1 2 1 3 2 1 3 3 2 3]`

Example:
```
categorical({'blue','red','yellow','blue','yellow','red','red','yell
ow','blue','red'})
```

parent — Parent container
`Figure` object | `Panel` object | `Tab` object | `TiledChartLayout` object

Parent container, specified as a `Figure`, `Panel`, `Tab`, or `TiledChartLayout` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `parallelplot(data,'GroupData',grpdata,'DataNormalization','zscore','Jitter',0)` specifies to group the numeric data in `data` by using `grpdata` and to display the data as z-scores, without any jittering.

`Title` — Plot title
`''` (default) | character vector | string array | cell array of character vectors | categorical array

Plot title, specified as a character vector, string array, cell array of character vectors, or categorical array. By default, the plot has no title.

To create a multiline title, specify a string array or cell array of character vectors. Each element in the array corresponds to a line of text.

If you specify the title as a categorical array, MATLAB uses the values in the array, not the categories.

Example: `p = parallelplot(__,'Title','My Title Text')`

Example: `p.Title = 'My Title Text'`

Example: `p.Title = {'My','Title'}`

`DataNormalization` — Normalization method for coordinates
`'range'` (default) | `'none'` | `'zscore'` | `'scale'` | `'center'` | `'norm'`

Normalization method for coordinates with numeric values, specified as one of the following options.

Method	Description
`'range'`	Display raw data along coordinate rulers that have independent minimum and maximum limits
`'none'`	Display raw data along coordinate rulers that have the same minimum and maximum limits
`'zscore'`	Display z-scores (with a mean of 0 and a standard deviation of 1) along each coordinate ruler
`'scale'`	Display values scaled by standard deviation along each coordinate ruler
`'center'`	Display data centered to have a mean of 0 along each coordinate ruler
`'norm'`	Display 2-norm values along each coordinate ruler

For more information about these methods, see `normalize`.

For a coordinate variable that is a logical vector, datetime array, duration array, categorical array, string array, or cell array of character vectors, `parallelplot` evenly distributes the unique possible values along the coordinate ruler, regardless of the normalization method.

Example: `p = parallelplot(__,'DataNormalization','none')`

Example: `p.DataNormalization = 'zscore'`

`Jitter` — Data displacement distance

`0.1` (default) | numeric scalar in the interval [0,1]

Data displacement distance along the coordinate rulers, specified as a numeric scalar in the interval [0,1]. The `Jitter` value determines the maximum distance to displace plot lines from their true value along the coordinate rulers, where the displacement is a uniform random amount. If you set the `Jitter` property to `1`, then adjacent jitter regions just touch. Set the `Jitter` property to `0` to display the true data values.

Some amount of jitter is particularly helpful for visualizing categorical data because the jittering enables you to distinguish between plot lines more easily. However, the `Jitter` value affects all coordinate variables, including numeric variables.

Example: `p = parallelplot(__,'Jitter',0.5)`

Example: `p.Jitter = 0.2`

`Color` — Group color

character vector | string array | cell array of character vectors | matrix of RGB values

Group color, specified in one of these forms:

- Character vector designating a color name, short name, or hexadecimal color code. A hexadecimal color code starts with a hash symbol (#) and is followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.
- String array or cell array of character vectors designating one or more color names, short names, or hexadecimal color codes.
- Three-column matrix of RGB values in the range [0,1]. The three columns represent the R value, G value, and B value.

Choose among these predefined colors, their equivalent RGB triplets, and their hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

By default, `parallelplot` assigns a maximum of seven unique group colors. When the total number of groups exceeds the number of specified colors, `parallelplot` cycles through the specified colors.

Example: `p = parallelplot(__,'Color',{'blue','black','green'})`

Example: `p.Color = [0 0 1; 0 0.5 0.5; 0.5 0.5 0.5]`

Example: `p.Color = {'#EDB120','#77AC30','#7E2F8E'}`

Tips

- To interactively explore the data in your `ParallelCoordinatesPlot` object, use these options (some are not available in the Live Editor):
  - Zoom — Use the scroll wheel to zoom.
  - Pan — Click and drag the parallel coordinates plot to pan.
  - Data tips — Hover over the parallel coordinates plot to display a data tip. The software highlights the corresponding line in the plot. For an example, see “Change Data Normalization in Plot” on page 1-10839.
  - Rearrange coordinates — Click and drag a coordinate tick label horizontally to move the corresponding coordinate ruler to a different position. For an example, see “Explore Table Data Using Parallel Coordinates Plot”.

See Also

Functions

`categorical` | `discretize` | `reordercats` | `table`

Properties
ParallelCoordinatesPlot

Topics
"Explore Table Data Using Parallel Coordinates Plot"
"Access Data in Tables"

Introduced in R2019a

matlab.unittest.parameters Package

Summary of classes associated with MATLAB Unit Test parameters

Description

The `matlab.unittest.parameters` package consists of the following classes used in parameterized testing.

Classes

matlab.unittest.parameters.ClassSetupParameter	Specification of Class Setup Parameter
matlab.unittest.parameters.EmptyParameter	Empty parameter implementation
matlab.unittest.parameters.MethodSetupParameter	Specification of Method Setup Parameter
matlab.unittest.parameters.Parameter	Base class for parameters
matlab.unittest.parameters.TestParameter	Specification of Test Parameter

See Also

`selectIf`

Topics

"Create Basic Parameterized Test"
"Create Advanced Parameterized Test"

Introduced in R2014a

matlab.unittest.parameters.EmptyParameter class

Package: `matlab.unittest.parameters`
Superclasses: `matlab.unittest.parameters.Parameter`

Empty parameter implementation

Description

The `matlab.unittest.parameters.EmptyParameter` class is a `Parameter` implementation that provides no parameter information. There is no need for test authors to interact with this `Parameter` directly. This class provides an empty parameter instance to the `Parameterization` property of a nonparameterized test element.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

See Also

`matlab.unittest.TestCase`

Topics

"Create Basic Parameterized Test"
"Create Advanced Parameterized Test"

Introduced in R2014a

matlab.unittest.parameters.ClassSetupParameter class

Package: `matlab.unittest.parameters`
Superclasses: `matlab.unittest.parameters.Parameter`

Specification of Class Setup Parameter

Description

The `matlab.unittest.parameters.ClassSetupParameter` class holds information about a single value of a Class Setup Parameter.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

See Also

`matlab.unittest.TestCase`

Topics

"Create Basic Parameterized Test"
"Create Advanced Parameterized Test"

Introduced in R2014a

matlab.unittest.parameters.MethodSetupParameter class

Package: `matlab.unittest.parameters`
Superclasses: `matlab.unittest.parameters.Parameter`

Specification of Method Setup Parameter

Description

The `matlab.unittest.parameters.MethodSetupParameter` class holds information about a single value of a Method Setup Parameter.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

See Also

`matlab.unittest.TestCase`

Topics

"Create Basic Parameterized Test"
"Create Advanced Parameterized Test"

Introduced in R2014a

matlab.unittest.parameters.Parameter class

Package: `matlab.unittest.parameters`

Base class for parameters

Description

In parameterized testing, use parameters to pass data to test methods.

Construction

Instantiate a `Parameter` using the static `fromData` method.

Properties

`Property` — Name of property that defines `Parameter`
character vector

This property is read-only.

Name of the property that defines the `Parameter`, stored as a character vector.

`Name` — Parameter value name
character vector

This property is read-only.

Parameter value name, stored as a character vector. `Name` uniquely identifies a particular value for a parameter.

`Value` — Parameter value
any type of array

This property is read-only.

Parameter value, stored as any type of array. `Value` holds the data that the `TestRunner` passes into a parameterized method.

Methods

fromData	Create parameters from data

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Test Using Parameters External to Test Class

In your working folder, create `testZeros.m`. This class contains five test methods, resulting in eleven parameterized tests.

```
classdef testZeros < matlab.unittest.TestCase
    properties (TestParameter)
        type = {'single','double','uint16'};
        outSize = struct('s2d',[3 3], 's3d',[2 5 4]);
    end

    methods (Test)
        function testClass(testCase, type, outSize)
            testCase.verifyClass(zeros(outSize,type), type);
        end

        function testSize(testCase, outSize)
            testCase.verifySize(zeros(outSize), outSize);
        end

        function testDefaultClass(testCase)
            testCase.verifyClass(zeros, 'double');
        end
```

```
        function testDefaultSize(testCase)
            testCase.verifySize(zeros, [1 1]);
        end

        function testDefaultValue(testCase)
            testCase.verifyEqual(zeros,0);
        end
    end
end
```

Redefine the `type` parameter so that the test uses `uint64` and `int64` data types in the parameterization instead of `single`, `double`, and `uint16`. Create parameters.

```
import matlab.unittest.parameters.Parameter
newType = {'int64','uint64'};
param = Parameter.fromData('type',newType);
```

Create a test suite that injects the `param` parameters. View the names of the tests in the suite. The injected parameters are indicated by `#ext`.

```
import matlab.unittest.TestSuite
suite = TestSuite.fromClass(?testZeros,'ExternalParameters',param);
{suite.Name}'
```

```
ans =

  9×1 cell array

    {'testZeros/testClass(type=int64#ext,outSize=s2d)' }
    {'testZeros/testClass(type=int64#ext,outSize=s3d)' }
    {'testZeros/testClass(type=uint64#ext,outSize=s2d)'}
    {'testZeros/testClass(type=uint64#ext,outSize=s3d)'}
    {'testZeros/testSize(outSize=s2d)'                  }
    {'testZeros/testSize(outSize=s3d)'                  }
    {'testZeros/testDefaultClass'                       }
    {'testZeros/testDefaultSize'                        }
    {'testZeros/testDefaultValue'                       }
```

Run the suite.

```
results = suite.run;
```

```
Running testZeros
.........
Done testZeros
__________
```

Redefine the `outSize` parameter so that the test parameterizes for 1-d and 4-d arrays. Create parameters from `newType` and `newSize`.

```
newSize = struct('s2d',[5 3],'s4d',[2 3 2 4]);
param = Parameter.fromData('type',newType,'outSize',newSize);
```

Create a test suite that injects the `param` parameters. View the names of the tests in the suite. The injected parameters are indicated by `#ext`.

```
import matlab.unittest.TestSuite
suite = TestSuite.fromClass(?testZeros,'ExternalParameters',param);
{suite.Name}'
```

```
ans =

  9×1 cell array

    {'testZeros/testClass(type=int64#ext,outSize=s2d#ext)' }
    {'testZeros/testClass(type=int64#ext,outSize=s4d#ext)' }
    {'testZeros/testClass(type=uint64#ext,outSize=s2d#ext)'}
    {'testZeros/testClass(type=uint64#ext,outSize=s4d#ext)'}
    {'testZeros/testSize(outSize=s2d#ext)'                  }
    {'testZeros/testSize(outSize=s4d#ext)'                  }
    {'testZeros/testDefaultClass'                           }
    {'testZeros/testDefaultSize'                            }
    {'testZeros/testDefaultValue'                           }
```

Run the suite.

```
results = suite.run;
```

```
Running testZeros
.........
Done testZeros
```

See Also

Topics

"Use External Parameters in Parameterized Test"

Introduced in R2018b

matlab.unittest.parameters.Parameter.fromData

Class: `matlab.unittest.parameters.Parameter`
Package: `matlab.unittest.parameters`

Create parameters from data

Syntax

```
param = matlab.unittest.parameters.Parameter.fromData(prop,nameVal)
param =
matlab.unittest.parameters.Parameter.fromData(prop1,nameVal1,...,pro
pN,nameValN)
```

Description

`param = matlab.unittest.parameters.Parameter.fromData(prop,nameVal)` creates an array of `Parameter` instances where `prop` defines the parameterizing property name for all `Parameter` elements and `nameVal` defines the name and value for each `Parameter` element.

Using `fromData` is analogous to defining parameters within a class-based test using a `properties` block. For example:

```
properties (TestParameter)
    prop = nameVal
end
```

However, with `fromData` you can redefine existing test parameters from outside the test class.

Use the `fromData` method to redefine parameters defined in the `TestParameter`, `MethodSetupParameter`, or `ClassSetupParameter` `properties` block of a parameterized test.

```
param =
matlab.unittest.parameters.Parameter.fromData(prop1,nameVal1,...,pro
```

`pN,nameValN)` defines `Parameter` instances with multiple parameterizing property names.

Input Arguments

`prop` — Parameterizing property name
character vector | string

Parameterizing property name, specified as a character vector or string.

Example: `'myParam'`

`nameVal` — Parameter name and value
cell array | struct

Parameter name and value, specified as a nonempty cell array or struct.

If `nameVal` is a struct, the struct field represents the parameter name and the struct value represents the parameter value. If `nameVal` is a cell array, each element represents a parameter value.

If `nameVal` is a cell array of character vectors, the parameter names are generated from the values. Otherwise, MATLAB generates parameter names.

Example: `struct('small', 1, 'medium', 10, 'large', 100)`

Example: `{42,7,13}`

Example: `{'double','single','uint16'}`

Examples

Test Using Parameters External to Test Class

In your working folder, create `testZeros.m`. This class contains five test methods, resulting in eleven parameterized tests.

```
classdef testZeros < matlab.unittest.TestCase
    properties (TestParameter)
```

```
        type = {'single','double','uint16'};
        outSize = struct('s2d',[3 3], 's3d',[2 5 4]);
    end

    methods (Test)
        function testClass(testCase, type, outSize)
            testCase.verifyClass(zeros(outSize,type), type);
        end

        function testSize(testCase, outSize)
            testCase.verifySize(zeros(outSize), outSize);
        end

        function testDefaultClass(testCase)
            testCase.verifyClass(zeros, 'double');
        end
        function testDefaultSize(testCase)
            testCase.verifySize(zeros, [1 1]);
        end

        function testDefaultValue(testCase)
            testCase.verifyEqual(zeros,0);
        end
    end
end
```

Redefine the `type` parameter so that the test uses `uint64` and `int64` data types in the parameterization instead of `single`, `double`, and `uint16`. Create parameters.

```
import matlab.unittest.parameters.Parameter
newType = {'int64','uint64'};
param = Parameter.fromData('type',newType);
```

Create a test suite that injects the `param` parameters. View the names of the tests in the suite. The injected parameters are indicated by `#ext`.

```
import matlab.unittest.TestSuite
suite = TestSuite.fromClass(?testZeros,'ExternalParameters',param);
{suite.Name}'
```

```
ans =

  9×1 cell array
```

```
    {'testZeros/testClass(type=int64#ext,outSize=s2d)' }
    {'testZeros/testClass(type=int64#ext,outSize=s3d)' }
    {'testZeros/testClass(type=uint64#ext,outSize=s2d)'}
    {'testZeros/testClass(type=uint64#ext,outSize=s3d)'}
    {'testZeros/testSize(outSize=s2d)'                 }
    {'testZeros/testSize(outSize=s3d)'                 }
    {'testZeros/testDefaultClass'                      }
    {'testZeros/testDefaultSize'                       }
    {'testZeros/testDefaultValue'                      }
```

Run the suite.

```
results = suite.run;
```

```
Running testZeros
.........
Done testZeros
__________
```

Redefine the `outSize` parameter so that the test parameterizes for 1-d and 4-d arrays. Create parameters from `newType` and `newSize`.

```
newSize = struct('s2d',[5 3],'s4d',[2 3 2 4]);
param = Parameter.fromData('type',newType,'outSize',newSize);
```

Create a test suite that injects the `param` parameters. View the names of the tests in the suite. The injected parameters are indicated by `#ext`.

```
import matlab.unittest.TestSuite
suite = TestSuite.fromClass(?testZeros,'ExternalParameters',param);
{suite.Name}'
```

```
ans =

  9×1 cell array

    {'testZeros/testClass(type=int64#ext,outSize=s2d#ext)' }
    {'testZeros/testClass(type=int64#ext,outSize=s4d#ext)' }
    {'testZeros/testClass(type=uint64#ext,outSize=s2d#ext)'}
    {'testZeros/testClass(type=uint64#ext,outSize=s4d#ext)'}
    {'testZeros/testSize(outSize=s2d#ext)'                 }
```

```
    {'testZeros/testSize(outSize=s4d#ext)'                 }
    {'testZeros/testDefaultClass'                          }
    {'testZeros/testDefaultSize'                           }
    {'testZeros/testDefaultValue'                          }
```

Run the suite.

```
results = suite.run;
```

```
Running testZeros
.........
Done testZeros
__________
```

See Also

Topics

"Use External Parameters in Parameterized Test"

Introduced in R2018b

matlab.unittest.parameters.TestParameter class

Package: `matlab.unittest.parameters`
Superclasses: `matlab.unittest.parameters.Parameter`

Specification of Test Parameter

Description

The `matlab.unittest.parameters.TestParameter` class holds information about a single value of a Test Parameter.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

See Also

`matlab.unittest.TestCase`

Topics

"Create Basic Parameterized Test"
"Create Advanced Parameterized Test"

Introduced in R2014a

ParameterizedFunctionLine Properties

Parameterized line chart appearance and behavior

Description

`ParameterizedFunctionLine` properties control the appearance and behavior of a `ParameterizedFunctionLine` object. By changing property values, you can modify certain aspects of the line chart.

Properties

Color and Styling

`Color` — Line color

`[0 0 1]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: 'blue'

Example: [0 0 1]

Example: '#0000FF'

ColorMode — Control how Color is set
'auto' (default) | 'manual'

Control how the Color property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the `Color` property by selecting a color from the `ColorOrder` property of the axes.
- `'manual'` — You control the value of the `Color` property manually, either by setting the value of the `Color` property directly on the object, or by including a color in the `LineSpec` argument when you call a plotting function.

If you change the value of the `Color` property manually, MATLAB changes the value of the `ColorMode` property to `'manual'`.

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineStyleMode` — Control how `LineStyle` is set
`'auto'` (default) | `'manual'`

Control how the `LineStyle` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the `LineStyle` property by selecting a line style from the `LineStyleOrder` property of the axes.
- `'manual'` — You control the value of the `LineStyle` property manually, either by setting the value of the `LineStyle` property directly on the object, or by specifying the `LineSpec` argument when you call a plotting function.

If you change the value of the `LineStyle` property manually, MATLAB changes the value of the `LineStyleMode` property to `'manual'`.

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

Markers

`Marker` — Marker symbol

`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | ...

Marker symbol, specified as one of the values listed in this table. By default, the object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerMode` — Control how `Marker` is set

`'auto'` (default) | `'manual'`

Control how the `Marker` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the object's `Marker` property.

- `'manual'` — You control the value of the `Marker` property manually, either by setting the value of the `Marker` property directly on the object, or by including a marker in the `LineSpec` argument when you call a plotting function.

If you change the value of the `Marker` property manually, MATLAB changes the value of the `MarkerMode` property to `'manual'`.

`MarkerSize` — Marker size
6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

`MarkerEdgeColor` — Marker outline color
`'auto'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'auto'` uses the same color as the `Color` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` value uses the same color as the `MarkerEdgeColor` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Example: `'#D2F9A7'`

Function

`XFunction` — Parametric input for x-coordinates
function handle | anonymous function | symbolic expression | symbolic function

Parametric input for x-coordinates, specified as a function handle, anonymous function, or a symbolic expression or function.

`YFunction` — Parametric input for y-coordinates
function handle | anonymous function | symbolic expression | symbolic function

Parametric input for y-coordinates, specified as a function handle, anonymous function, or a symbolic expression or function.

`ZFunction` — Parametric input for z-coordinates
function handle | anonymous function | symbolic expression | symbolic function

Parametric input for z-coordinates, specified as a function handle, anonymous function, or a symbolic expression or function. For a 2-D plot, `ZFunction` is empty.

`MeshDensity` — Number of evaluation points
23 (default) | number

Number of evaluation points, specified as a number. The default is `23`. Because `ParameterizedFunctionLine` uses adaptive evaluation, the actual number of evaluation points is greater.

`TRange` — Range of values of parameter
[–5 5] (default) | vector of two numbers

Range of values of parameter, specified as a vector of two numbers. The default range is `[-5 5]`.

`TRangeMode` — Selection mode for `TRange`
`'auto'` (default) | `'manual'`

Selection mode for `TRange`, specified as one of these values:

- `'auto'` — Use the default value `[-5 5]`.
- `'manual'` — Use manually specified values. To specify the values, set the `TRange` property or specify the input argument `[tmin tmax]` to the plotting function.

Coordinate Data

XData — x values
vector

This property is read-only.

x values, specified as a vector. `XData`, `YData`, and `ZData` have equal lengths.

YData — y values
vector

y values, specified as a vector. `XData`, `YData`, and `ZData` have equal lengths.

ZData — z values
vector

z values, specified as a vector. `XData`, `YData`, and `ZData` have equal lengths.

Legend

DisplayName — Text for legend label
autogenerated label (default) | character vector | string

Text for legend label, specified as a custom character vector or string. The default label is autogenerated from the `Function` property and the `texlabel` function. The legend does not appear until you call the `legend` function.

Data Types: `char` | `string`

Annotation — Control for including or excluding object from legend
`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`DataTipTemplate` — Data tip content

`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

`Clipping` — Clipping of object to axes limits

`'on'` (default) | `'off'`

Clipping of the object to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the object that are outside the axes limits.
- `'off'` — Display the entire object, even if parts of it appear outside the axes limits. Parts of the object might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the object so that it is larger than the original plot.

The `Clipping` property of the axes that contains the object must be set to `'on'`. Otherwise, this property has no effect. For more information about the clipping behavior, see the `Clipping` property of the axes.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see "Callback Definition".

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you

do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks when visible. The `Visible` property must be set to `'on'` and you must click a part of the `ParameterizedFunctionLine` object that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The `HitTest` property determines if the `ParameterizedFunctionLine` object responds to the click or if an ancestor does.
- `'all'` — Capture mouse clicks regardless of visibility. The `Visible` property can be set to `'on'` or `'off'` and you can click a part of the `ParameterizedFunctionLine` object that has no color. The `HitTest` property determines if the `ParameterizedFunctionLine` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `ParameterizedFunctionLine` object passes the click through it to the object below it in the current view of the figure window. The `HitTest` property has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `ParameterizedFunctionLine` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `ParameterizedFunctionLine` object that has one of these:
  - `HitTest` property set to `'on'`
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

Note The `PickableParts` property determines if the `ParameterizedFunctionLine` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent

`Axes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `Group`, or `Transform` object.

`Children` — Children

empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the chart object.

HandleVisibility — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'parameterizedfunctionline'`

This property is read-only.

Type of graphics object, returned as `'parameterizedfunctionline'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
[ ] (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`fplot` | `fplot3`

Topics

“Access Property Values”
“Graphics Object Properties”

Introduced in R2016a

ParameterizedFunctionSurface Properties

Parameterized surface chart appearance and behavior

Description

`ParameterizedFunctionSurface` properties control the appearance and behavior of `ParameterizedFunctionSurface` objects. By changing property values, you can modify certain aspects of the surface chart.

Properties

Faces

`FaceColor` — Face color
`'interp'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Face color, specified as `'interp'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'interp'` interpolates the colors based on the `ZData` values.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

FaceAlpha — Face transparency

1 (default) | scalar in range [0 1]

Face transparency, specified as a scalar in the range [0,1]. Use uniform transparency across all of the faces. A value of 1 is fully opaque and 0 is completely transparent. Values between 0 and 1 are semitransparent.

Edges

EdgeColor — Line color

`[0 0 0]` (default) | `'interp'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as `'interp'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default RGB triplet value of `[0 0 0]` corresponds to black. The `'interp'` value colors the edges based on the `ZData` values.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
'-'	Solid line	
'--'	Dashed line	
':'	Dotted line	
'-.'	Dash-dotted line	
'none'	No line	No line

`LineWidth` — Line width

`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

Markers

`Marker` — Marker symbol

`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | `'x'` | `...`

Marker symbol, specified as one of the markers listed in this table. By default, the object does not display markers. Specifying a marker symbol adds markers at the intersection points of mesh lines.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'+'`

Example: `'diamond'`

`MarkerSize` — Marker size
6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

`MarkerEdgeColor` — Marker outline color
`'auto'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'auto'` uses the same color as the `EdgeColor` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

Example: `'#D2F9A7'`

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` value uses the same color as the `MarkerEdgeColor` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Example: `'#D2F9A7'`

Function

`XFunction` — Parametric input for x-coordinates

function handle | anonymous function | symbolic expression | symbolic function

Parametric input for x-coordinates, specified as a function handle, anonymous function, or a symbolic expression or function.

`YFunction` — Parametric input for y-coordinates

function handle | anonymous function | symbolic expression | symbolic function

Parametric input for y-coordinates, specified as a function handle, anonymous function, or a symbolic expression or function.

`ZFunction` — Parametric input for z-coordinates

function handle | anonymous function | symbolic expression | symbolic function

Parametric input for z-coordinates, specified as a function handle, anonymous function, or a symbolic expression or function. For a 2-D plot, `ZFunction` is empty.

URange — Range of first parameter

[–5 5] (default) | vector of two numbers

Range of the first parameter, specified as a vector of two numbers. The default range is `[-5 5]`.

`URangeMode` — Selection mode for `URange`
`'auto'` (default) | `'manual'`

Selection mode for `URange`, specified as one of these values:

- `'auto'` — Use the default value `[-5 5]`.
- `'manual'` — Use manually specified values. To specify the values, set the `URange` property or specify the input argument `[umin umax vmin vmax]` to the plotting function.

`VRange` — Range of second parameter
[–5 5] (default) | vector of two numbers

Range of the second parameter, specified as a vector of two numbers. The default range is `[-5 5]`.

`VRangeMode` — Selection mode for `VRange`
`'auto'` (default) | `'manual'`

Selection mode for `VRange`, specified as one of these values:

- `'auto'` — Use the default value `[-5 5]`.
- `'manual'` — Use manually specified values. To specify the values, set the `VRange` property or specify the input argument `[umin umax vmin vmax]` to the plotting function.

`MeshDensity` — Number of evaluation points per direction
35 (default) | number

Number of evaluation points per direction, specified as a number. The default is `35`. Because `ParameterizedFunctionSurface` objects use adaptive evaluation, the actual number of evaluation points is greater.

Example: `100`

`ShowContours` — Display contour plot under plot
`'off'` (default) | `'on'`

Display contour plot under plot, specified as `'off'` (default) or `'on'`.

Data

`XData` — x values
matrix

This property is read-only.

x values specified as a matrix. `XData` is at least a 2-by-2 matrix. `size(XData)`, `size(YData)`, and `size(ZData)` are equal.

`YData` — y values
matrix

This property is read-only.

y values, specified as a matrix. `YData` is at least a 2-by-2 matrix. `size(XData)`, `size(YData)`, and `size(ZData)` are equal.

`ZData` — Data that defines surface to contour
matrix

This property is read-only.

Data that defines the surface to contour, specified as a matrix. `ZData` is at least a 2-by-2 matrix. `size(XData)`, `size(YData)`, and `size(ZData)` are equal.

Lighting

`AmbientStrength` — Strength of ambient light
`0.3` (default) | scalar in range `[0,1]`

Strength of ambient light, specified as a scalar value in the range `[0,1]`. Ambient light is a nondirectional light that illuminates the entire scene. There must be at least one visible light object in the axes for the ambient light to be visible.

The `AmbientLightColor` property for the axes sets the color of the ambient light. The color is the same for all objects in the axes.

Example: `0.5`

Data Types: `double`

`DiffuseStrength` — Strength of diffuse light
`0.6` (default) | scalar in range `[0,1]`

Strength of diffuse light, specified as a scalar value in the range `[0,1]`. Diffuse light is the nonspecular reflectance from light objects in the axes.

Example: `0.3`

Data Types: `double`

`SpecularStrength` — Strength of specular reflection
`0.9` (default) | scalar in range `[0,1]`

Strength of specular reflection, specified as a scalar value in the range `[0,1]`. Specular reflections are the bright spots on the surface from light objects in the axes.

Example: `0.3`

Data Types: `double`

`SpecularExponent` — Size of specular spot
10 (default) | scalar greater than or equal to 1

Size of specular spot, specified as a scalar value greater than or equal to 1. Most materials have exponents in the range `[5 20]`.

Example: `7`

Data Types: `double`

`SpecularColorReflectance` — Color of specular reflections
1 (default) | scalar in range `[0,1]`

Color of specular reflections, specified as a scalar value in the range `[0,1]`. A value of `1` sets the color using only the color of the light source. A value of `0` sets the color using both the color of the object from which it reflects and the color of the light source. The `Color` property of the light contains the color of the light source. The proportions vary linearly for values in between.

Example: `0.5`

Data Types: `double`

Legend

`DisplayName` — Text for legend label
autogenerated label (default) | character vector | string

Text for legend label, specified as a custom character vector or string. The default label is autogenerated from the `Function` property and the `texlabel` function. The legend does not appear until you call the `legend` function.

Data Types: `char` | `string`

`Annotation` — Control for including or excluding object from legend
`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility
`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`DataTipTemplate` — Data tip content
`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

BusyAction — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

PickableParts — Ability to capture mouse clicks
`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks when visible. The `Visible` property must be set to `'on'` and you must click a part of the `ParameterizedFunctionSurface` object that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The `HitTest` property determines if the `ParameterizedFunctionSurface` object responds to the click or if an ancestor does.
- `'all'` — Capture mouse clicks regardless of visibility. The `Visible` property can be set to `'on'` or `'off'` and you can click a part of the `ParameterizedFunctionSurface` object that has no color. The `HitTest` property determines if the `ParameterizedFunctionSurface` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `ParameterizedFunctionSurface` object passes the click through it to the object below it in the current view of the figure window. The `HitTest` property has no effect.

HitTest — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `ParameterizedFunctionSurface` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `ParameterizedFunctionSurface` object that has one of these:
  - `HitTest` property set to `'on'`
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

---

Note The `PickableParts` property determines if the `ParameterizedFunctionSurface` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

BeingDeleted — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

Parent — Parent

`Axes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `Group`, or `Transform` object.

Children — Children

empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the chart object.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object
`'parameterizedfunctionsurface'`

This property is read-only.

Type of graphics object, returned as `'parameterizedfunctionsurface'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[ ]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`fmesh` | `fsurf`

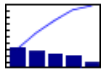
Topics

“Access Property Values”
“Graphics Object Properties”

Introduced in R2016a

pareto

Pareto chart



Syntax

```
pareto(Y)
pareto(Y,names)
pareto(Y,X)
H = pareto(...)
[H,ax] = pareto(...)
```

Description

Pareto charts display the values in the vector `Y` as bars drawn in descending order. Values in `Y` must be nonnegative and not include `NaN`s. Only the first 95% of the cumulative distribution is displayed.

`pareto(Y)` labels each bar with its element index in `Y` and also plots a line displaying the cumulative sum of `Y`.

`pareto(Y,names)` labels each bar with the associated text in the matrix or cell array `names`.

`pareto(Y,X)` labels each bar with the associated value from `X`.

`pareto(ax,..)` plots into the axes `ax` rather than the current axes, `gca`.

`H = pareto(...)` returns the primitive `Line` and `Bar` objects created.

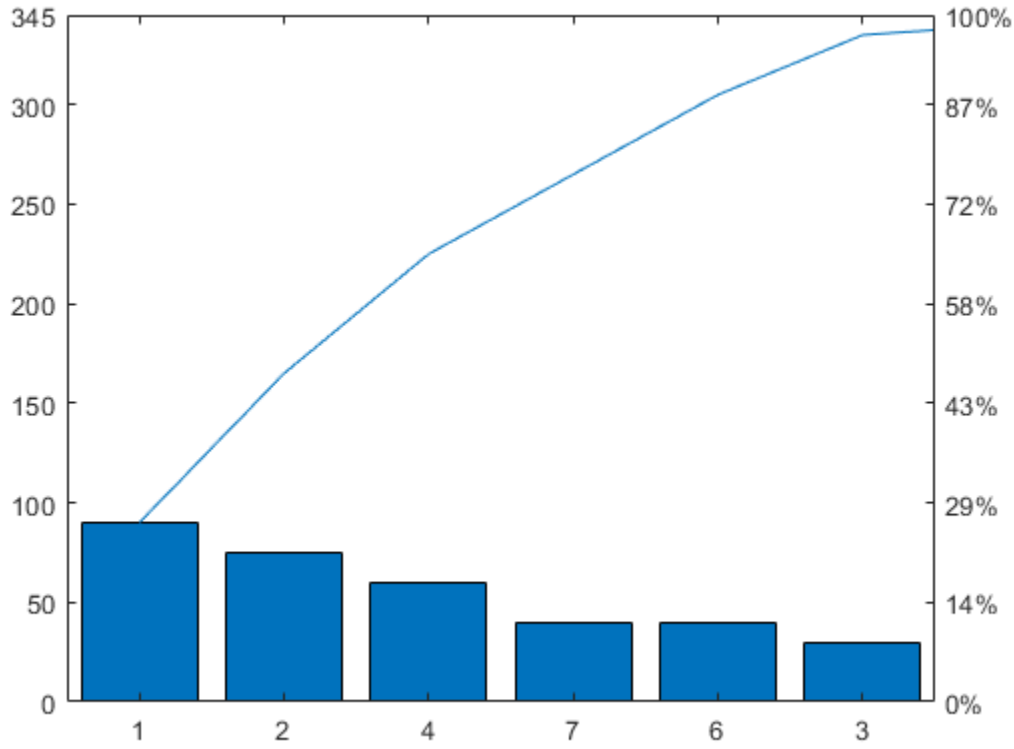
`[H,ax] = pareto(...)` additionally returns the two axes objects created.

Examples

Create Pareto Chart

Create a Pareto chart of vector `y`.

```
y = [90,75,30,60,5,40,40,5];
figure
pareto(y)
```

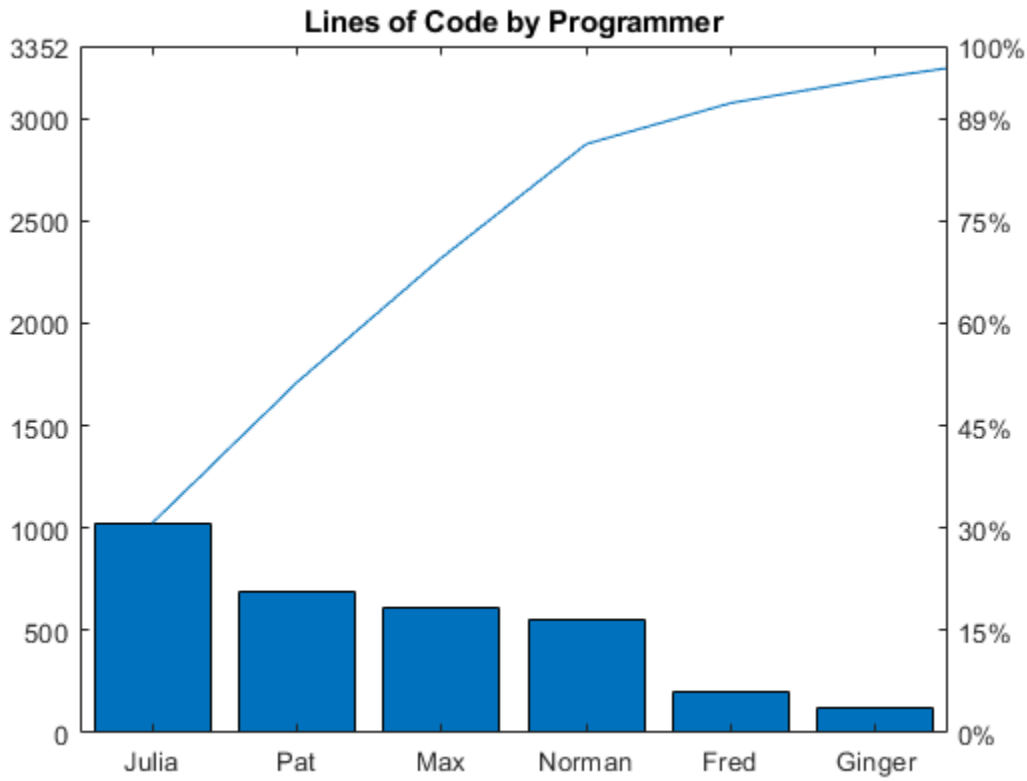


`pareto` displays the elements in `y` as bars in descending order and labels each bar with its index in `y`. Since `pareto` displays only the first 95% of the cumulative distribution, some elements in `y` are not displayed.

Label Bars in Pareto Chart

Examine the cumulative productivity of a group of programmers to see how normal its distribution is. Label each bar with the name of the programmer.

```
codelines = [200 120 555 608 1024 101 57 687];
coders = {'Fred','Ginger','Norman','Max','Julia','Wally','Heidi','Pat'};

figure
pareto(codelines, coders)
title('Lines of Code by Programmer')
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions
`bar` | `histogram` | `line`

Properties
Bar | Primitive Line

Introduced before R2006a

parfor

Parallel for loop

Syntax

```
parfor loopvar = initval:endval; statements; end
parfor (loopvar = initval:endval, M); statements; end
```

Description

`parfor loopvar = initval:endval; statements; end` executes a series of MATLAB statements for values of `loopvar` between `initval` and `endval`, inclusive, which specify a vector of increasing integer values. The loop runs in parallel when you have the Parallel Computing Toolbox or when you create a MEX function or standalone code with MATLAB Coder. Unlike a traditional `for`-loop, iterations are not executed in a guaranteed order. You cannot call scripts directly in a `parfor`-loop. However, you can call functions that call scripts.

`parfor (loopvar = initval:endval, M); statements; end` executes statements in a loop using a maximum of `M` workers or threads, where `M` is a nonnegative integer.

Examples

Perform three large eigenvalue computations using three workers or cores with Parallel Computing Toolbox software:

```
parpool(3)
parfor i=1:3, c(:,i) = eig(rand(1000)); end
```

Tips

- If you have Parallel Computing Toolbox software, see the function reference pages for `parfor` and `parpool` for additional information.

- If you have MATLAB Coder software, see the `parfor` function reference page for additional information.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Treated as a `for`-loop in a MATLAB Function block or when used with `fiaccel`.
- See `parfor` in MATLAB Coder.
- See “Generate Code with Parallel for-Loops (parfor)” (MATLAB Coder).

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

If you have Parallel Computing Toolbox installed, then when you use `parfor`, MATLAB automatically opens a parallel pool of workers on your local machine. MATLAB runs the loop across the available workers. Control parallel behavior with the parallel preferences, including scaling up to a cluster.

For details, see “Parallel for-Loops (parfor)” (Parallel Computing Toolbox).

See Also

`for`

Introduced in R2008a

parseSoapResponse

Convert response string from SOAP (Simple Object Access Protocol) server into MATLAB types

Note `parseSoapResponse` will be removed in a future release. Use `matlab.wsdl.createWSDLClient` instead.

Syntax

```
data = parseSoapResponse(response)
```

Description

`data = parseSoapResponse(response)` extracts data from SOAP server `response` and converts to MATLAB types.

Examples

Retrieve Book Information from Library Database

This example assumes the library is on a local intranet and does not use an actual endpoint; therefore, you cannot run it.

Retrieve the name of the author of a book titled “In the Fall.” The relative path of the library service is `urn:LibraryCatalog`. To get the author's name, use the `getAuthor` function, which takes the book name as the input value. The `getAuthor` parameter is `nameToLookUp`. The XML data type for title is `{http://www.w3.org/2001/XMLSchema}string`. The SOAP message style is `rpc` by default.

Create the SOAP message.

```
message = createSoapMessage(...
  'urn:LibraryCatalog',...
```

```
  'getAuthor',...
  {'In the Fall'},...
  {'nameToLookUp'},...
  {'{http://www.w3.org/2001/XMLSchema}string'})
```

```
message =

[#document: null]
```

This response does not necessarily indicate that the message is valid, although certain input problems produce error messages.

Send the message to the server for processing, and get the author's name back. The server endpoint is `http://test/soap/services/LibraryCatalog`. The server method is `urn:LibraryCatalog#getAuthor`.

```
response = callSoapService(...
  'http://test/soap/services/LibraryCatalog',...
  'urn:LibraryCatalog#getAuthor',...
  message)
```

```
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body>
<getAuthorResponse xmlns="urn:LibraryCatalog">
<ns1:getAuthorReturn xmlns:ns1="http://latestversion.soap.test">
Kate Alvin
</ns1:getAuthorReturn>
</getAuthorResponse>
</soapenv:Body>
</soapenv:Envelope>
```

MATLAB returns the message in a single line, displayed here on separate lines for legibility.

Extract the author's name.

```
author = parseSoapResponse(response)
```

```
author = Kate Alvin
```

MATLAB automatically converted the XML string data type to `char`.

Input Arguments

`response` — Data from SOAP server
string

Data from SOAP server, specified as a string. `response` is the output from the `callSoapService` function.

Output Arguments

`data` — Output of SOAP service call
cell array of any valid MATLAB type

Output of SOAP service call, returned as a cell array of any valid MATLAB type. For information about `data`, see the documentation for the SOAP service used in the `callSoapService` function.

See Also

`callSoapService` | `createSoapMessage` | `matlab.wsdl.createWSDLClient` | `urlread` | `xmlread`

Introduced before R2006a

pascal

Pascal matrix

Syntax

```
P = pascal(n)
P = pascal(n,1)
P = pascal(n,2)
P = pascal( ___ ,classname)
```

Description

`P = pascal(n)` returns a “Pascal’s Matrix” on page 1-10917 of order `n`. `P` is a symmetric positive definite matrix with integer entries taken from Pascal's triangle. The inverse of `P` has integer entries.

`P = pascal(n,1)` returns the lower triangular Cholesky factor (up to the signs of the columns) of the Pascal matrix. `P` is *involutary*, that is, it is its own inverse.

`P = pascal(n,2)` returns a transposed and permuted version of `pascal(n,1)`. In this case, `P` is a cube root of the identity matrix.

`P = pascal( ___ ,classname)` returns a matrix of class `classname` using any of the input argument combinations in previous syntaxes. `classname` can be `'single'` or `'double'`.

Examples

Matrix from Pascal's Triangle

Compute the fourth-order Pascal matrix.

```
A = pascal(4)
```

```
A = 4×4

     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    20
```

Compute the lower triangular Cholesky factor of the third-order Pascal matrix, and verify it is involutory.

```
A = pascal(3,1)
```

```
A = 3×3

     1     0     0
     1    -1     0
     1    -2     1
```

```
inv(A)
```

```
ans = 3×3

     1     0     0
     1    -1     0
     1    -2     1
```

Input Arguments

n — Matrix order
scalar, nonnegative integer

Matrix order, specified as a scalar, nonnegative integer.

Example: `pascal(10)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`

classname — Matrix class
`'double'` (default) | `'single'`

Matrix class, specified as either `'double'` or `'single'`.

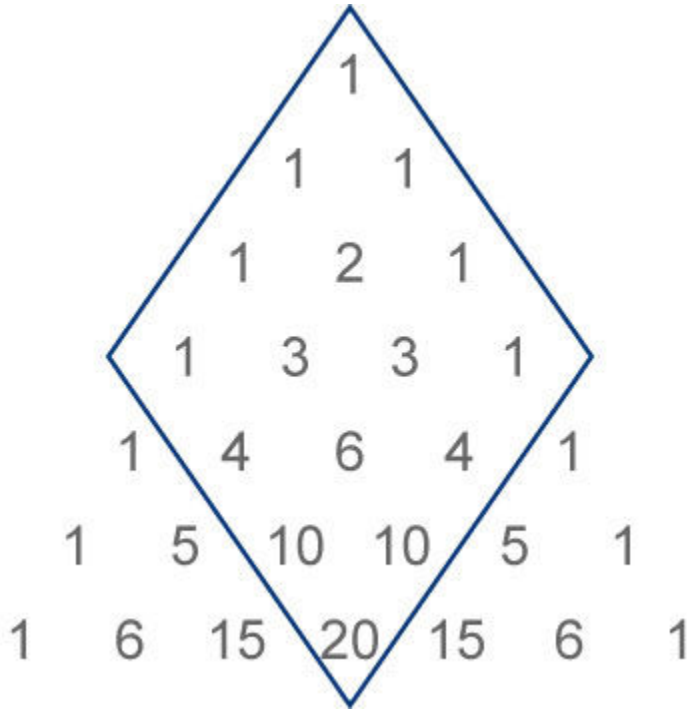
Example: `pascal(10,'single')`

Data Types: `char`

More About

Pascal’s Matrix

Pascal’s triangle is a triangle formed by rows of numbers. The first row has entry `1`. Each succeeding row is formed by adding adjacent entries of the previous row, substituting a `0` where no adjacent entry exists. The `pascal` function forms Pascal’s matrix by selecting the portion of Pascal’s triangle that corresponds to the specified matrix dimensions, as outlined in the graphic. The matrix outlined corresponds to the MATLAB command `pascal(4)`.



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`chol` | `gallery` | `vander`

Introduced before R2006a

patch

Plot one or more filled polygonal regions

Syntax

```
patch(X,Y,C)
patch(X,Y,Z,C)
patch('XData',X,'YData',Y)
patch('XData',X,'YData',Y,'ZData',Z)

patch('Faces',F,'Vertices',V)

patch(S)

patch( ___ ,Name,Value)
patch(ax, ___ )

p = patch( ___ )
```

Description

`patch(X,Y,C)` plots one or more filled polygonal regions using the elements of `X` and `Y` as the coordinates for each vertex. `patch` connects the vertices in the order that you specify them. To create one polygon, specify `X` and `Y` as vectors. To create multiple polygons, specify `X` and `Y` as matrices where each column corresponds to a polygon. `C` determines the polygon colors.

`patch(X,Y,Z,C)` creates the polygons in 3-D coordinates using `X`, `Y`, and `Z`. To view the polygons in a 3-D view, use the `view(3)` command. `C` determines the polygon colors.

`patch('XData',X,'YData',Y)` is similar to `patch(X,Y,C)`, except that you do not have to specify color data for the 2-D coordinates.

`patch('XData',X,'YData',Y,'ZData',Z)` is similar to `patch(X,Y,Z,C)`, except that you do not have to specify color data for the 3-D coordinates.

`patch('Faces',F,'Vertices',V)` creates one or more polygons where `V` specifies vertex values and `F` defines which vertices to connect. Specifying only unique vertices and their connection matrix can reduce the size of the data when there are many polygons. Specify one vertex per row in `V`. To create one polygon, specify `F` as a vector. To create multiple polygons, specify `F` as a matrix with one row per polygon. Each face does not have to have the same number of vertices. To specify different numbers of vertices, pad `F` with `NaN` values.

`patch(S)` creates one or more polygons using structure `S`. The structure fields correspond to patch property names and the field values corresponding to property values. For example, `S` can contain the fields `Faces` and `Vertices`.

`patch( ___ ,Name,Value)` creates polygons and specifies one or more patch properties using name-value pair arguments. A patch is the object that contains the data for all of the polygons created. You can specify patch properties with any of the input argument combinations in the previous syntaxes. For example, `'LineWidth',2` sets the outline width for all of the polygons to 2 points.

`patch(ax, ___ )` creates the patch in the axes specified by `ax` instead of in the current axes (`gca`). The option `ax` can precede any of the input argument combinations in the previous syntaxes.

`p = patch( ___ )` returns the patch object that contains the data for all the polygons. Use `p` to query and modify properties of the patch object after it is created. For a list of properties and descriptions, see Patch.

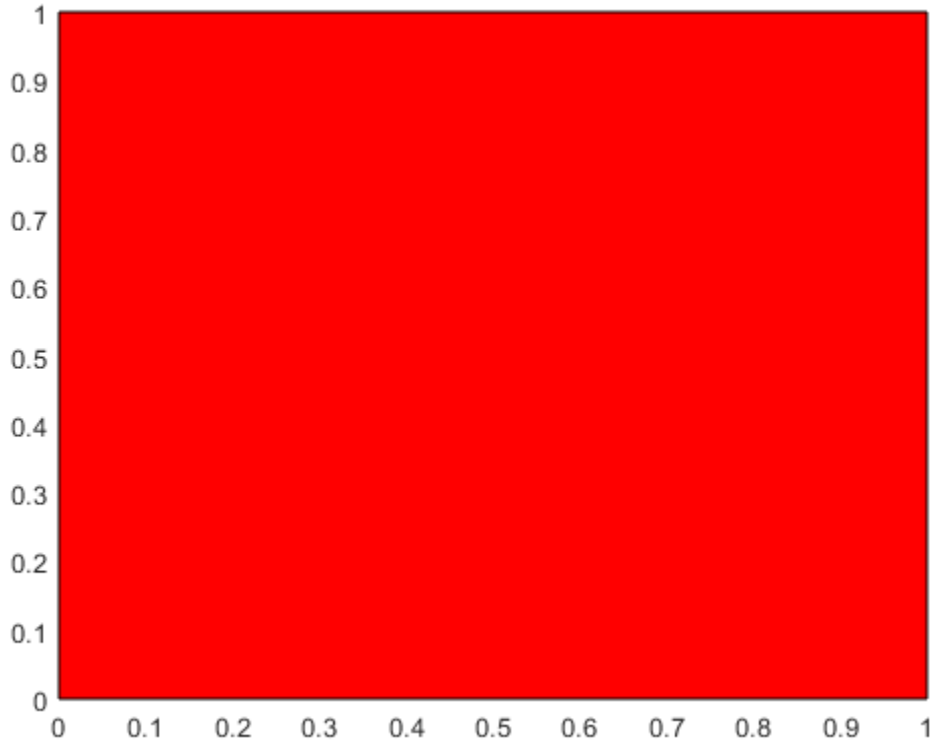
Examples

Specifying Coordinates

Create a single polygon by specifying the (*x*,*y*) coordinates of each vertex. Then, add two more polygons to the figure.

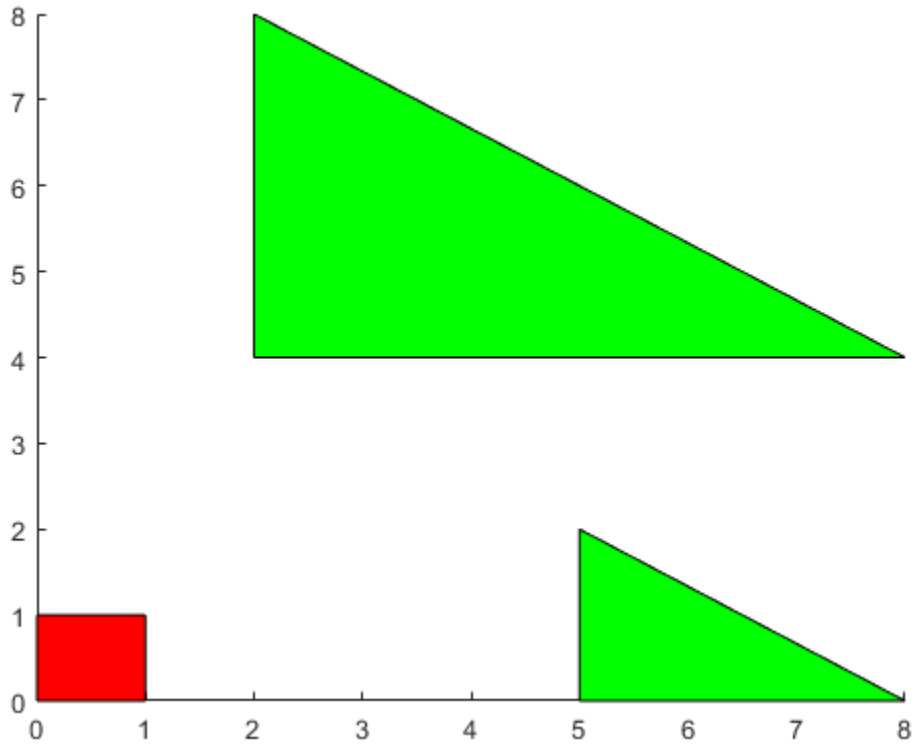
Create a red square with vertices at `(0,0)`, `(1,0)`, `(1,1)`, and `(0,1)`. Specify `x` as the *x*-coordinates of the vertices and `y` as the *y*-coordinates. `patch` automatically connects the last (*x*,*y*) coordinate with the first (*x*,*y*) coordinate.

```
x = [0 1 1 0];
y = [0 0 1 1];
patch(x,y,'red')
```



Create two polygons by specifying `x` and `y` as two-column matrices. Each column defines the coordinates for one of the polygons. `patch` adds the polygons to the current axes without clearing the axes.

```
x2 = [2 5; 2 5; 8 8];
y2 = [4 0; 8 2; 4 0];
patch(x2,y2,'green')
```

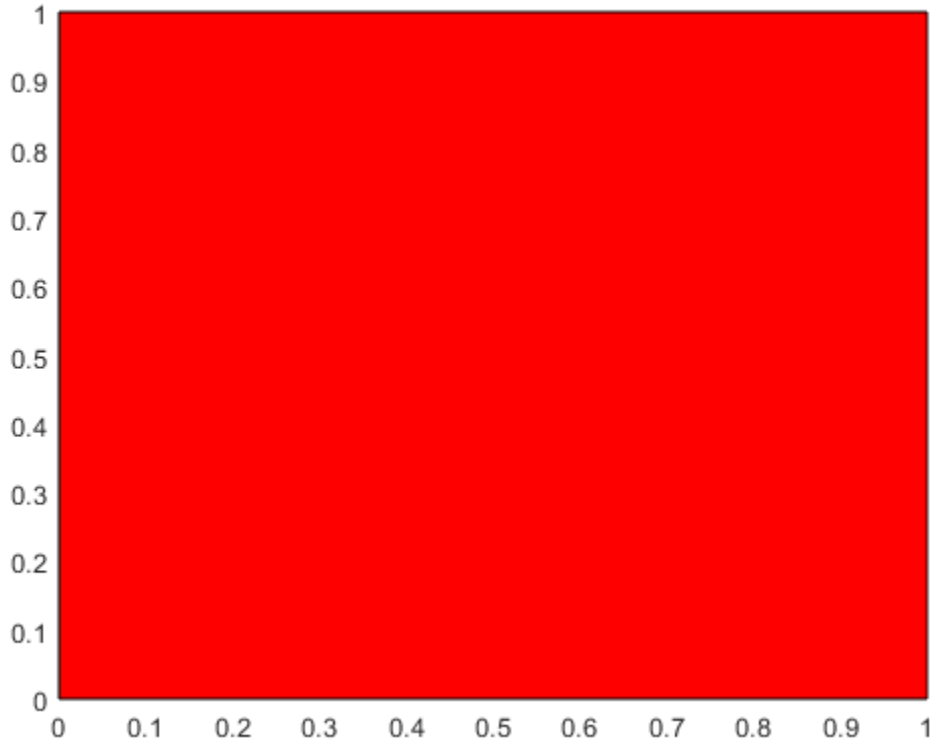


Specifying Faces and Vertices

Create a single polygon by specifying the coordinates of each unique vertex and a matrix that defines how to connect them. Then, add two more polygons to the figure.

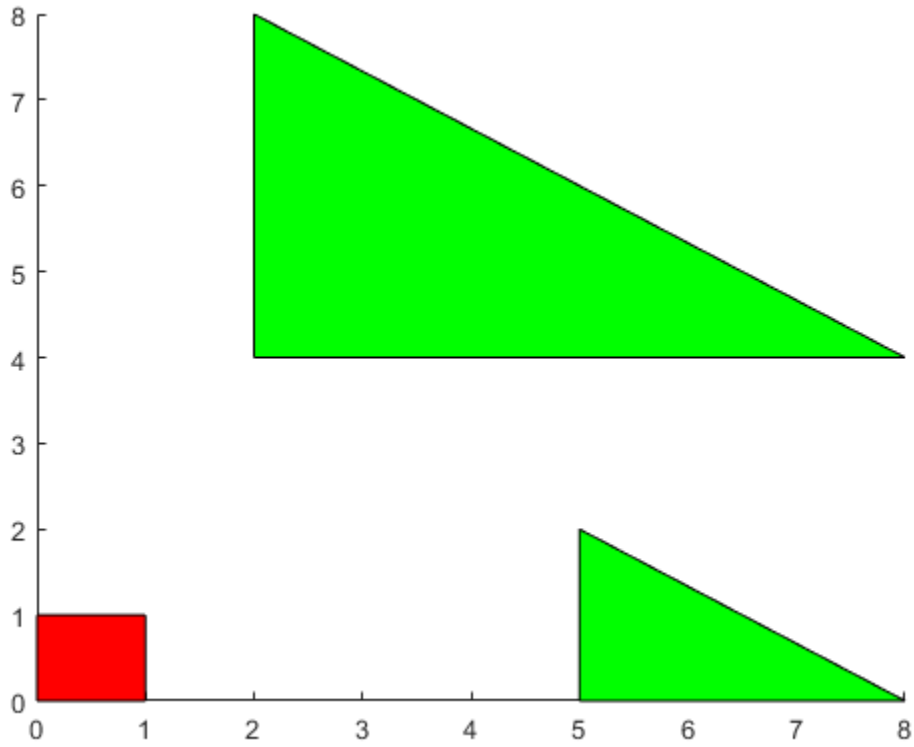
Create a red square with corners at `(0,0)`, `(1,0)`, `(1,1)`, and `(0,1)`. Specify `v` so that each row defines the (*x*,*y*) coordinates for one vertex. Then, specify `f` as the vertices to connect. Set the color by specifying the `FaceColor` property.

```
v = [0 0; 1 0; 1 1; 0 1];
f = [1 2 3 4];
patch('Faces',f,'Vertices',v,'FaceColor','red')
```



Create two polygons by specifying `f` as a two-row matrix. Each row defines the face for one patch.

```
v2 = [2 4; 2 8; 8 4; 5 0; 5 2; 8 0];
f2 = [1 2 3;
    4 5 6];
patch('Faces',f2,'Vertices',v2,'FaceColor','green')
```



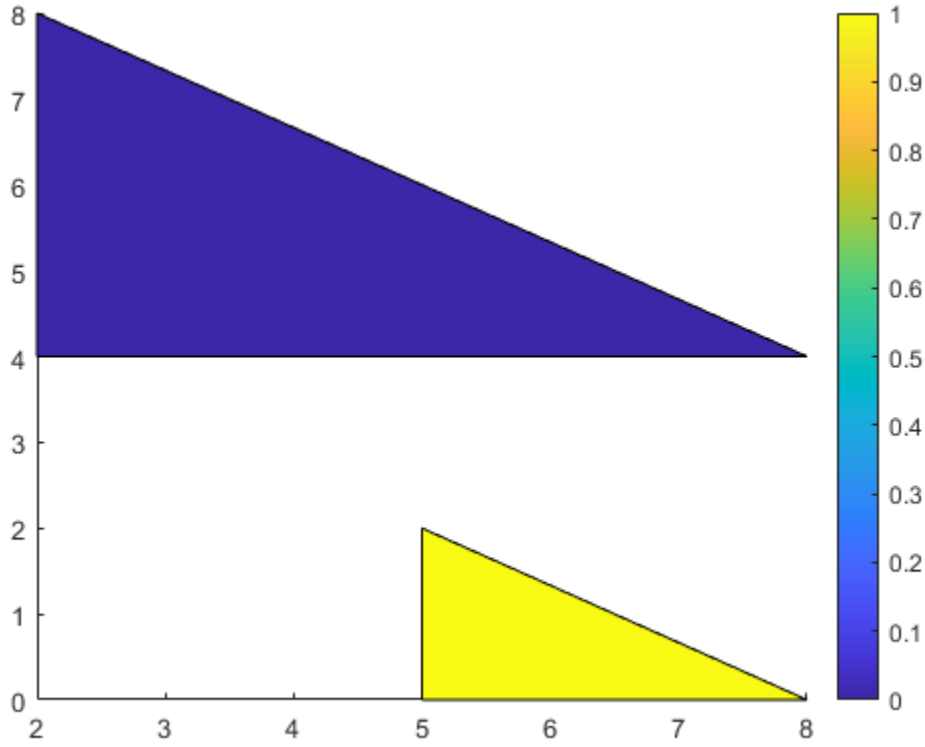
Different Polygon Face Colors

Create two polygons and use a different color for each polygon face. Use a colorbar to show how the colors map into the colormap.

Create the polygons using matrices `x` and `y`. Specify `c` as an column vector with two elements since there are two polygon faces, and add a colorbar.

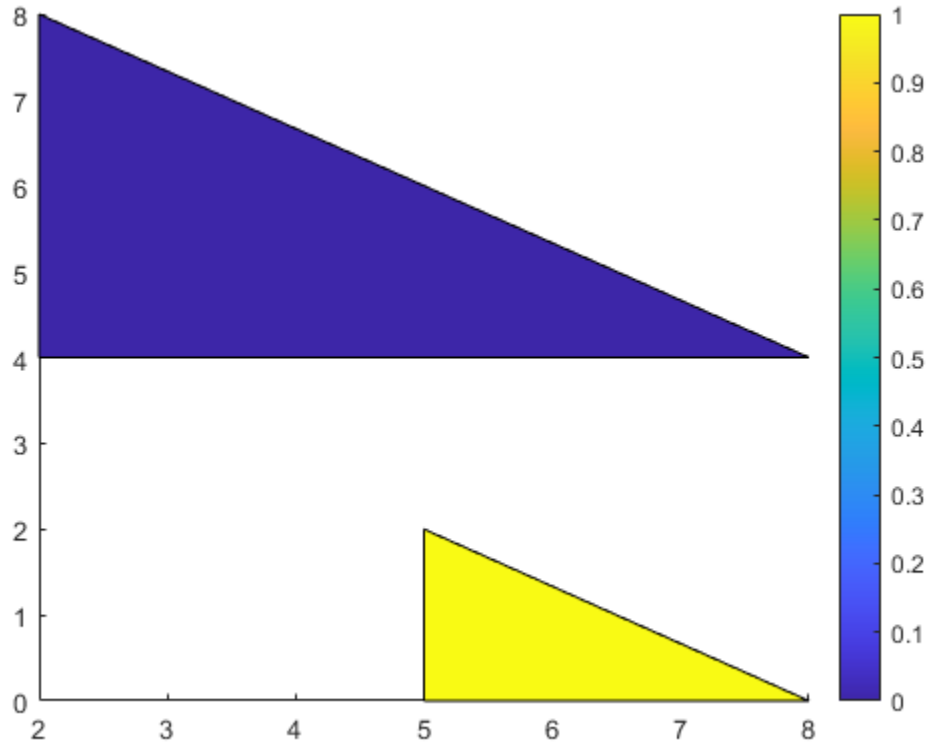
```
x = [2 5; 2 5; 8 8];
y = [4 0; 8 2; 4 0];
c = [0; 1];
```

```
figure
patch(x,y,c)
colorbar
```



Alternatively, you can get the same result when using `f` and `v` instead. When you create the polygons, set `FaceVertexCData` to a column vector with two elements since there are two polygon faces. Set `FaceColor` to `'flat'`.

```
v = [2 4; 2 8; 8 4; 5 0; 5 2; 8 0];
f = [1 2 3; 4 5 6];
col = [0; 1];
figure
patch('Faces',f,'Vertices',v,'FaceVertexCData',col,'FaceColor','flat');
colorbar
```



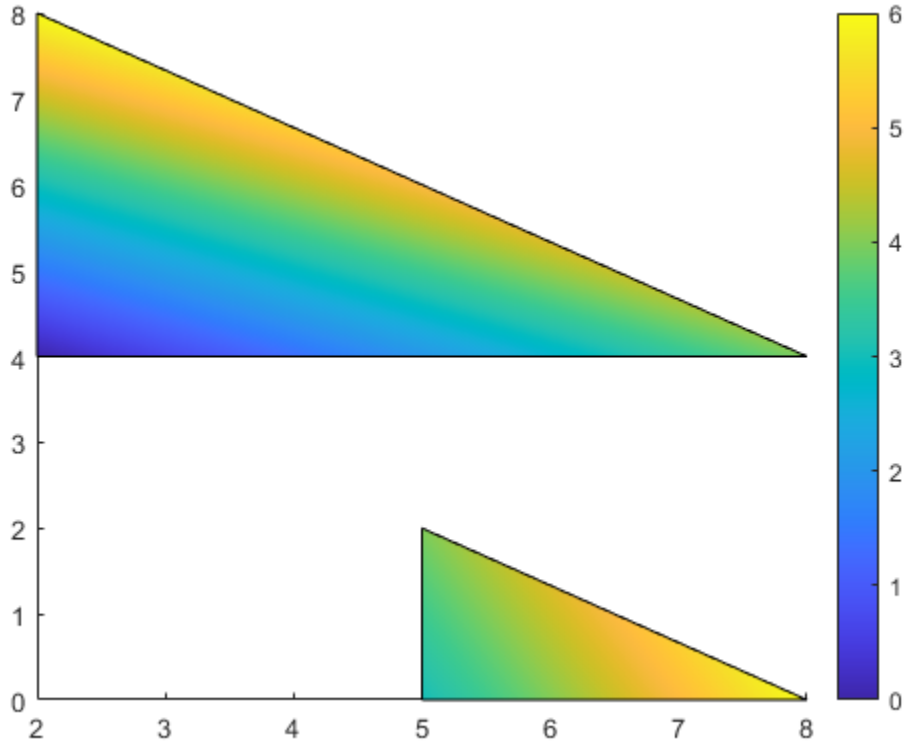
Interpolated Polygon Face Colors

Interpolate colors across polygon faces by specifying a color at each polygon vertex, and use a colorbar to show how the colors map into the colormap.

Create the polygons using matrices `x` and `y`. Specify `c` as a matrix the same size as `x` and `y` defining one color per vertex, and add a colorbar.

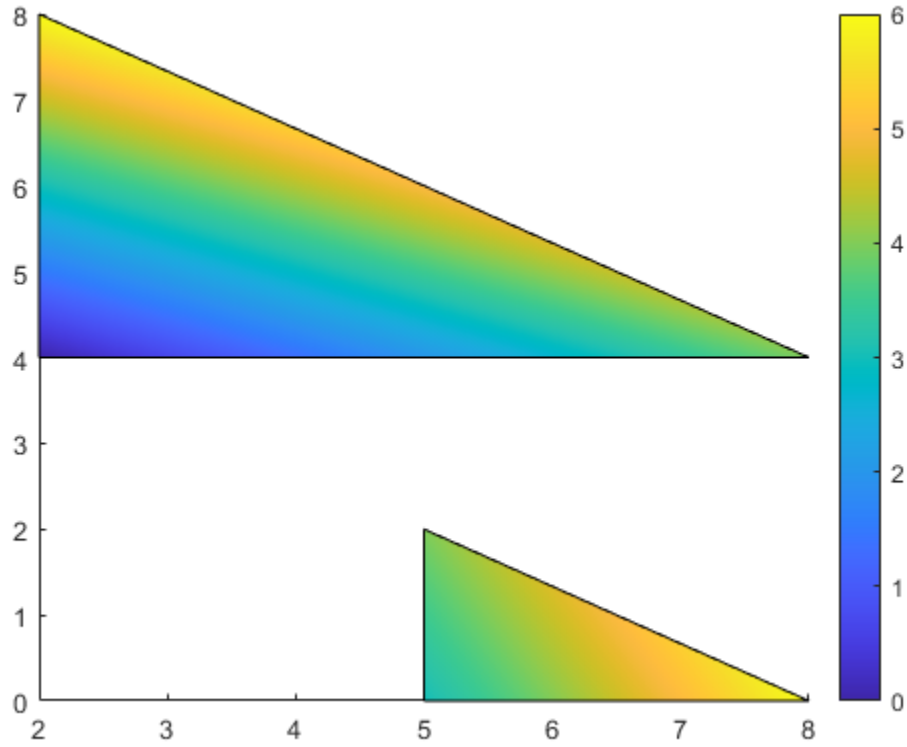
```
x = [2 5; 2 5; 8 8];
y = [4 0; 8 2; 4 0];
c = [0 3; 6 4; 4 6];
```

```
figure
patch(x,y,c)
colorbar
```



Alternatively, you can get the same result using `f` and `v` instead. When you create the polygons, set `FaceVertexCData` to a column vector with one value per vertex and set `FaceColor` to `'interp'`.

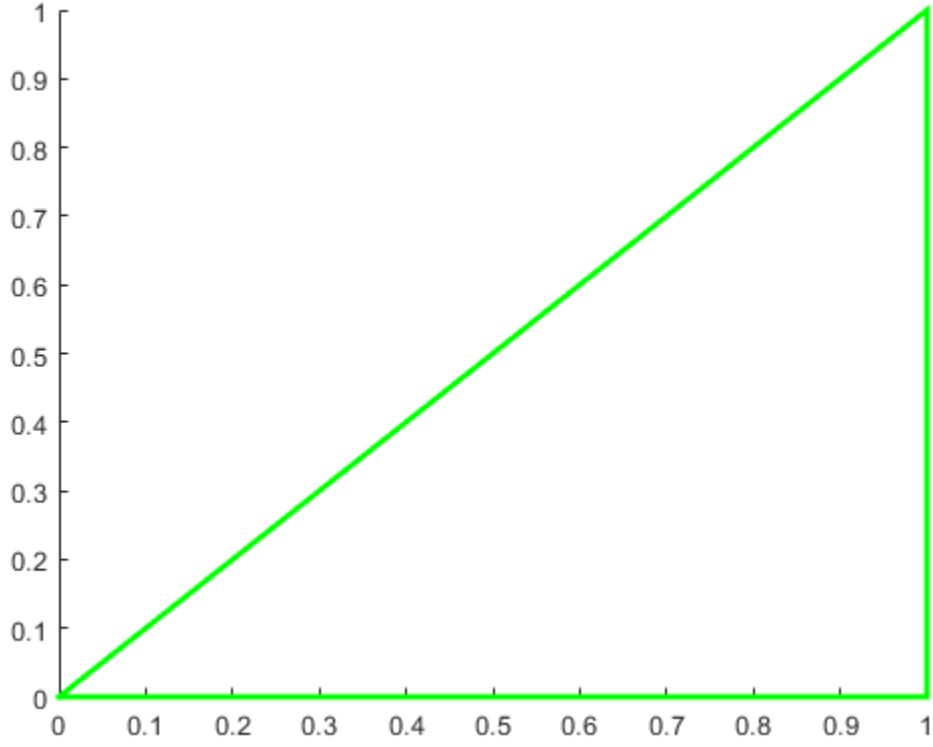
```
v = [2 4; 2 8; 8 4; 5 0; 5 2; 8 0];
f = [1 2 3; 4 5 6];
col = [0; 6; 4; 3; 4; 6];
figure
patch('Faces',f,'Vertices',v,'FaceVertexCData',col,'FaceColor','interp');
colorbar
```



Polygon Edges Without Faces

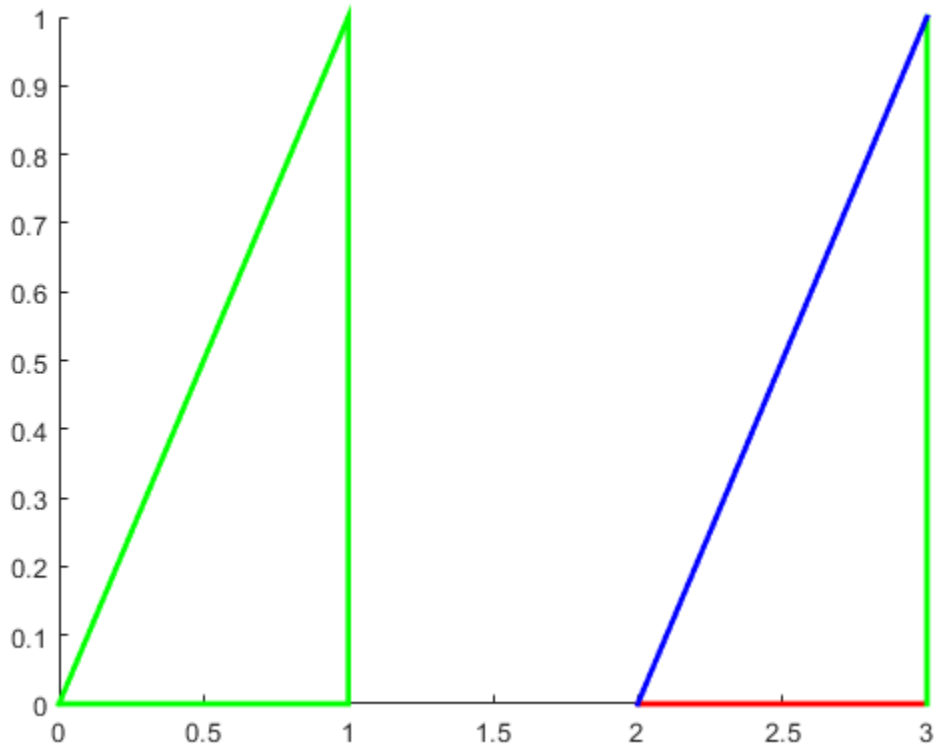
Create a polygon with green edges and do not display the face. Then, create a second polygon with a different color for each edge.

```
v = [0 0; 1 0; 1 1];
f = [1 2 3];
figure
patch('Faces',f,'Vertices',v,...
    'EdgeColor','green','FaceColor','none','LineWidth',2);
```



Use a different color for each edge by specifying a color for each vertex and setting `EdgeColor` to `'flat'`.

```
v = [2 0; 3 0; 3 1];
f = [1 2 3];
c = [1 0 0; % red
    0 1 0; % green
    0 0 1]; % blue
patch('Faces',f,'Vertices',v,'FaceVertexCData',c,...
    'EdgeColor','flat','FaceColor','none','LineWidth',2);
```

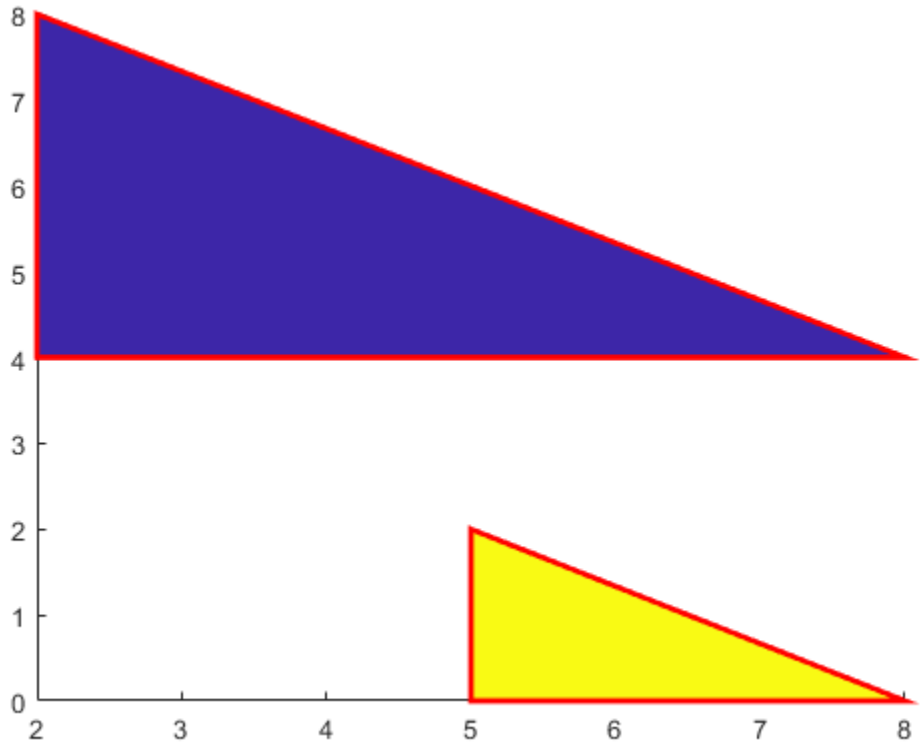


Polygons Using Structure

Use a structure to create two polygons. First, create a structure with fields names that match patch property names. Then, use the structure to create the polygons.

```
clear S
S.Vertices = [2 4; 2 8; 8 4; 5 0; 5 2; 8 0];
S.Faces = [1 2 3; 4 5 6];
S.FaceVertexCData = [0; 1];
S.FaceColor = 'flat';
S.EdgeColor = 'red';
```

```
S.LineWidth = 2;
figure
patch(S)
```

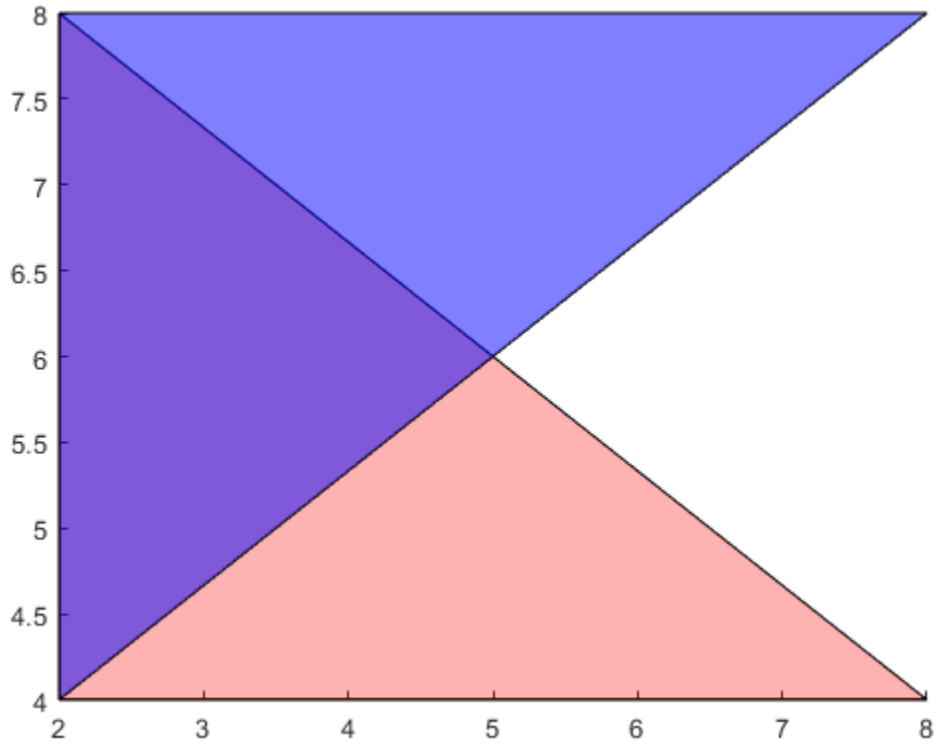


Semitransparent Polygons

Create two semitransparent polygons by setting the `FaceAlpha` property to a value between `0` and `1`.

```
v1 = [2 4; 2 8; 8 4];
f1 = [1 2 3];
```

```
figure
patch('Faces',f1,'Vertices',v1,'FaceColor','red','FaceAlpha',.3);

v2 = [2 4; 2 8; 8 8];
f2 = [1 2 3];
patch('Faces',f2,'Vertices',v2,'FaceColor','blue','FaceAlpha',.5);
```



Create Multicolored Line

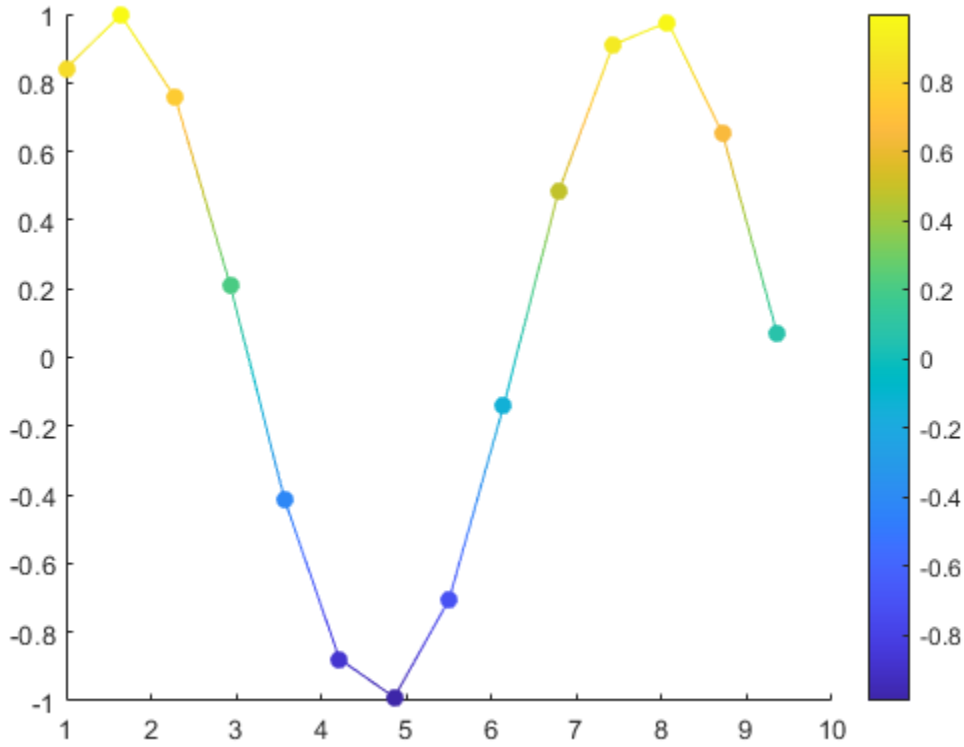
Create a multicolored line with markers at each vertex. Interpolate the colors and use a colorbar to show how the values map to the colormap.

Create the data. Set the last entry of `y` to `NaN` so that `patch` creates a line instead of a closed polygon. Define a color for each vertex using the `y` values. The values in `c` map to colors in the colormap.

```
x = linspace(1,10,15);
y = sin(x);
y(end) = NaN;
c = y;
```

Create the line. Show markers at each vertex and set the `EdgeColor` to `'interp'` to interpolate the colors between vertices. Add a colorbar.

```
figure
patch(x,y,c,'EdgeColor','interp','Marker','o','MarkerFaceColor','flat');
colorbar;
```



Input Arguments

X — *x*-coordinates for vertices
vector | matrix

x-coordinates for the vertices, specified in one of these forms:

- Vector — Create one polygon.
- Matrix — Create `n` polygons with `m` vertices each, where `[m,n] = size(X)`. Each column in the matrix corresponds to one polygon.

If the data does not define closed polygons, then `patch` closes the polygons. If the edges of an individual polygon intersect themselves, the resulting polygons might be partly filled. In that case, it is better to divide the patch object into smaller polygons.

When you specify `X`, the `patch` function sets the `XData` property for the patch object to the same value. The patch object automatically calculates the face and vertex data and sets the `Faces` and `Vertices` properties to the appropriate values.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Y — *y*-coordinates for vertices
vector | matrix

y-coordinates for the vertices, specified in one of these forms:

- Vector — Create one polygon.
- Matrix — Create `n` polygons with `m` vertices each, where `[m,n] = size(Y)`. Each column in the matrix corresponds to one polygon.

If the data does not define closed polygons, then `patch` closes the polygons. If the edges of an individual polygon intersect themselves, the resulting polygons might be partly filled. In that case, it is better to divide the patch object into smaller polygons.

When you specify `Y`, the `patch` function sets the `YData` property for the patch object to the same value. The patch object automatically calculates the face and vertex data and sets the `Faces` and `Vertices` properties to the appropriate values.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Z — *z*-coordinates for vertices
vector | matrix

z-coordinates for the vertices, specified in one of these forms:

- Vector — Create one polygon.
- Matrix — Create `m` polygons with `n` vertices each, where `[m,n] = size(Z)`. Each column in the matrix corresponds to one polygon.

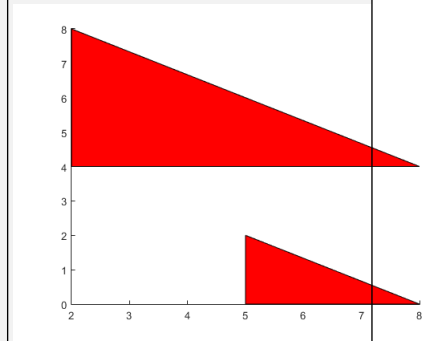
When you specify `Z`, the `patch` function sets the `ZData` property for the patch object to the same value. The patch object automatically calculates the face and vertex data and sets the `Faces` and `Vertices` properties to the appropriate values.

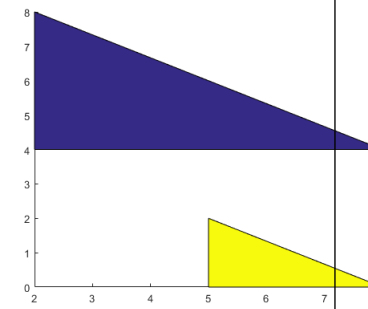
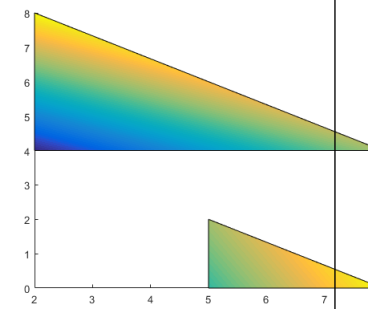
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`C` — Polygon colors

scalar | vector | matrix | RGB triplet | `'r'` | `'g'` | `'b'` | ...

Polygon colors, specified as a scalar, vector, matrix, or a color name. The format of the input determines whether all polygons have the same color, one color per face, or interpolated face colors.

Desired Effect	Use One of These Formats	Results
Single color for all faces	• Color name, for example, `'r'` for red. • RGB triplet, for example, `[0 .5 .5]`. • Scalar value, for example, `2`. The `CDataMapping` property determines how the value maps into the colormap. For an example, see “Specifying Coordinates” on page 1-10920.	 • Sets the `FaceColor` property to the specified color.

Desired Effect	Use One of These Formats	Results
One color per face	• `n`-by-1 vector of colormap colors, where `n` is the number of faces. The `CDataMapping` property determines how the values map into the colormap. • `n`-by-1-by-3 array of RGB values. The first page of the array defines the red components of the colors, the second page defines the blue, and the third page defines the green. For an example, see “Different Polygon Face Colors” on page 1-10924.	 • Sets the `FaceColor` property to `'flat'`. • Sets the `CData` and `FaceVertexCData` properties using the specified color values.
Interpolated face colors	• `m`-by-`n` matrix of colormap values, where `[m,n] = size(X)`. Specify one color per vertex. • `m`-by-`n`-by-3 array of RGB values. For an example, see “Interpolated Polygon Face Colors” on page 1-10926.	 • Sets the `FaceColor` property to `'interp'`. • Sets the `CData` and `FaceVertexCData` properties using the specified color values.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`

`V` — Polygon vertices

two-column or three-column matrix

Polygon vertices, specified in one of these forms:

- Two-column matrix — Each row contains the (*x*,*y*) coordinates for a vertex.
- Three-column matrix — Each row contains the (*x*,*y*,*Z*) coordinates for a vertex.

Specify only unique vertices. You can refer to a vertex more than once when defining the face definitions in `F`.

When you specify `V`, the `patch` function sets the `Vertices` property for the patch object to the same value. The patch object automatically calculates the coordinate data and sets the `XData`, `YData`, and `ZData` to the appropriate values.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`F` — Face definitions

row vector | matrix

Face definitions, specified in one of these forms:

- Row vector — Create a single polygon.
- Matrix — Create multiple polygons where each row corresponds to a polygon.

For example, this code defines three vertices in `V` and creates one polygon by connecting vertex 1 to 2, 2 to 3, and 3 to 1.

```
V = [1 1; 2 1; 2 2];
F = [1 2 3 1];
patch('Faces',F,'Vertices',V)
```

When you specify `F`, the `patch` function sets the `Faces` property for the patch object to the same value. The patch object automatically calculates the coordinate data and sets the `XData`, `YData`, and `ZData` to the appropriate values.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

S — Patch definition
structure

Patch definition, specified as a structure with fields that correspond patch property names and field values that correspond to patch property values.

ax — Axes object
axes object

Axes object. If you do not specify an axes object, then `patch` uses the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `patch(x,y,c,'FaceAlpha',.5,'LineStyle',':')` creates semitransparent polygons with dotted edges.

The properties listed here are only a subset of patch properties. For a complete list, see Patch.

FaceColor — Face color

`[0 0 0]` (default) | `'interp'` | `'flat'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Face color, specified as `'interp'`, `'flat'` an RGB triplet, a hexadecimal color code, a color name, or a short name.

To create a different color for each face, specify the `CData` or `FaceVertexCData` property as an array containing one color per face or one color per vertex. The colors can be interpolated from the colors of the surrounding vertices of each face, or they can be uniform. For interpolated colors, specify this property as `'interp'`. For uniform colors, specify this property as `'flat'`. If you specify `'flat'` and a different color for each vertex, the color of the first vertex you specify determines the face color.

To designate a single color for all of the faces, specify this property as an RGB triplet, a hexadecimal color code, a color name, or a short name.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`FaceAlpha` — Face transparency

1 (default) | scalar in range `[0,1]` | `'flat'` | `'interp'`

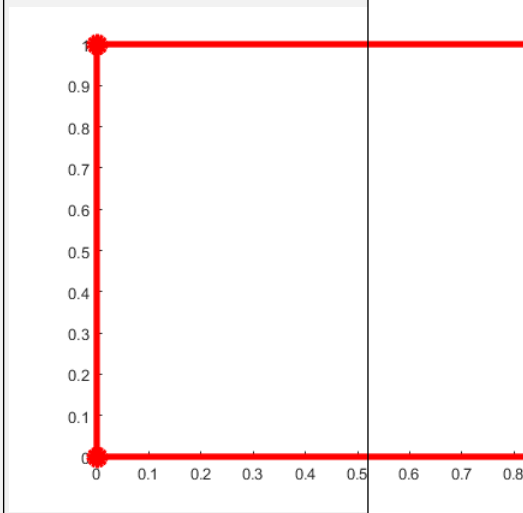
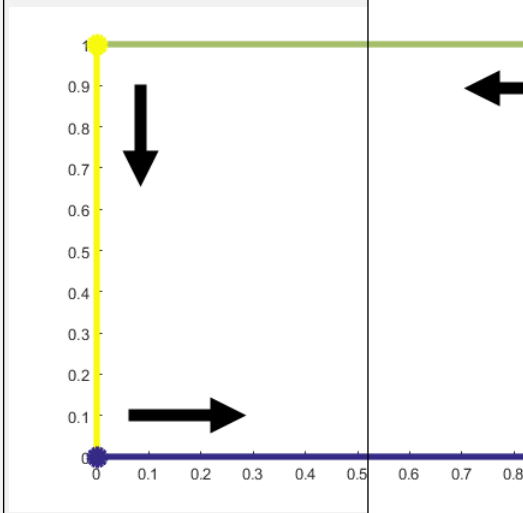
Face transparency, specified as one of these values:

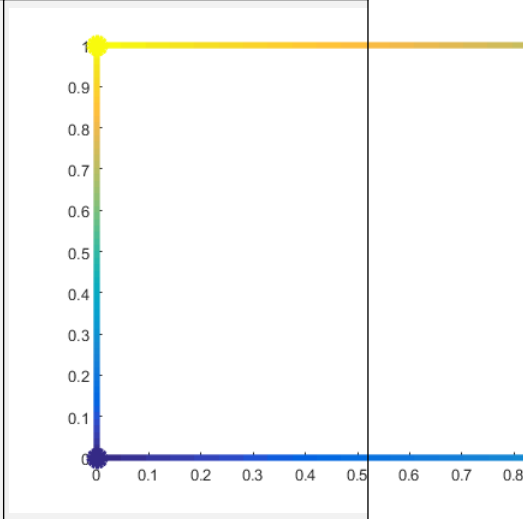
- Scalar in range `[0,1]` — Use uniform transparency across all of the faces. A value of `1` is fully opaque and `0` is completely transparent. This option does not use the transparency values in the `FaceVertexAlphaData` property.
- `'flat'` — Use a different transparency for each face based on the values in the `FaceVertexAlphaData` property. First you must specify the `FaceVertexAlphaData` property as a vector containing one transparency value per face or vertex. The transparency value at the first vertex determines the transparency for the entire face.
- `'interp'` — Use interpolated transparency for each face based on the values in `FaceVertexAlphaData` property. First you must specify the `FaceVertexAlphaData` property as a vector containing one transparency value per vertex. The transparency varies across each face by interpolating the values at the vertices.

`EdgeColor` — Edge colors

`[0 0 0]` (default) | `'none'` | `'flat'` | `'interp'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Edge colors, specified as one of the values in this table. The default edge color is black with a value of `[0 0 0]`. If multiple polygons share an edge, then the first polygon drawn controls the displayed edge color.

Value	Description	Result
RGB triplet, hexadecimal color code, or color name	Single color for all of the edges. See the following table for more details.	
`'flat'`	Different color for each edge. Use the vertex colors to set the color of the edge that follows it. You must first specify `CData` or `FaceVertexCData` as an array containing one color per vertex. The edge color depends on the order in which you specify the vertices.	

Value	Description	Result
`'interp'`	Interpolated edge color. You must first specify `CData` or `FaceVertexCData` as an array containing one color per vertex. Determine the edge color by linearly interpolating the values at the two bounding vertices.	
`'none'`	No edges displayed.	No edges displayed.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
'-'	Solid line	
'--'	Dashed line	
':'	Dotted line	
'-.'	Dash-dotted line	
'none'	No line	No line

Output Arguments

p — patch object
scalar

Patch object, returned as a scalar. Each patch object can consist of one or more polygons. Use p to query or change properties of the patch object after it is created.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions
`area` | `fill` | `fill3`

Properties
Patch

Topics

"Multifaceted Patches"
"How Patch Data Relates to a Colormap"
"Differences Between Colormaps and Truecolor"

Introduced before R2006a

Patch Properties

Patch appearance and behavior

Description

`Patch` properties control the appearance and behavior of `Patch` objects. By changing property values, you can modify certain aspects of the patch.

Starting in R2014b, you can use dot notation to query and set properties.

```
p = patch;
c = p.CData;
p.CDataMapping = 'scaled';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Color

`FaceColor` — Face color

`[0 0 0]` (default) | `'interp'` | `'flat'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Face color, specified as `'interp'`, `'flat'` an RGB triplet, a hexadecimal color code, a color name, or a short name.

To create a different color for each face, specify the `CData` or `FaceVertexCData` property as an array containing one color per face or one color per vertex. The colors can be interpolated from the colors of the surrounding vertices of each face, or they can be uniform. For interpolated colors, specify this property as `'interp'`. For uniform colors, specify this property as `'flat'`. If you specify `'flat'` and a different color for each vertex, the color of the first vertex you specify determines the face color.

To designate a single color for all of the faces, specify this property as an RGB triplet, a hexadecimal color code, a color name, or a short name.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

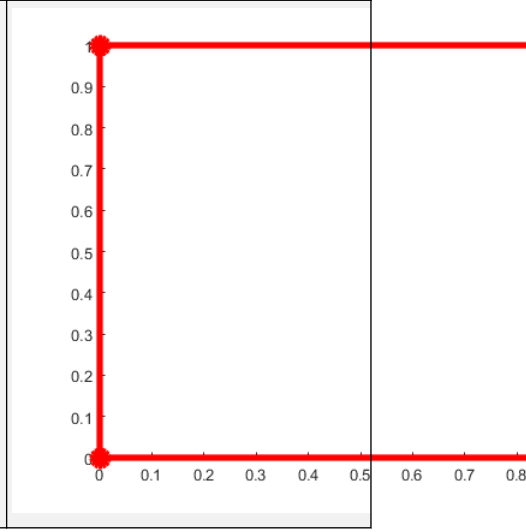
Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

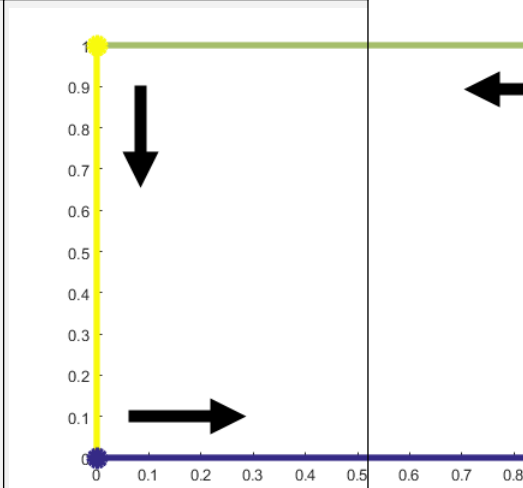
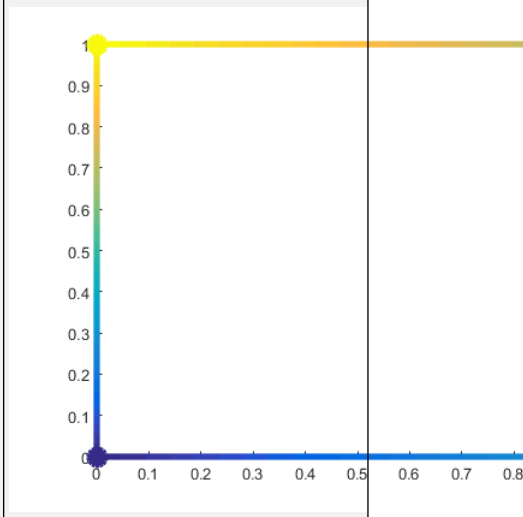
RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`EdgeColor` — Edge colors

`[0 0 0]` (default) | `'none'` | `'flat'` | `'interp'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Edge colors, specified as one of the values in this table. The default edge color is black with a value of `[0 0 0]`. If multiple polygons share an edge, then the first polygon drawn controls the displayed edge color.

Value	Description	Result
RGB triplet, hexadecimal color code, or color name	Single color for all of the edges. See the following table for more details.	

Value	Description	Result
`'flat'`	Different color for each edge. Use the vertex colors to set the color of the edge that follows it. You must first specify `CData` or `FaceVertexCData` as an array containing one color per vertex. The edge color depends on the order in which you specify the vertices.	
`'interp'`	Interpolated edge color. You must first specify `CData` or `FaceVertexCData` as an array containing one color per vertex. Determine the edge color by linearly interpolating the values at the two bounding vertices.	
`'none'`	No edges displayed.	No edges displayed.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

CData — Patch color data

single color for entire patch | one color per face | one color per vertex

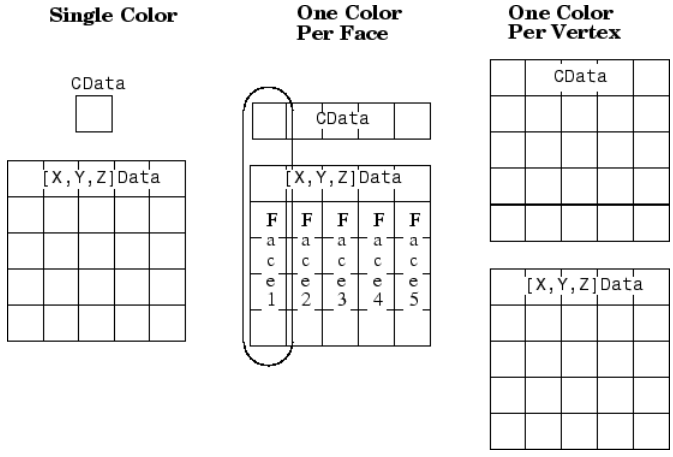
Patch color data, specified as a single color for the entire patch, one color per face, or one color per vertex.

The way the `patch` function interprets `CData` depends on the type of data supplied. Specify `CData` in one of these forms:

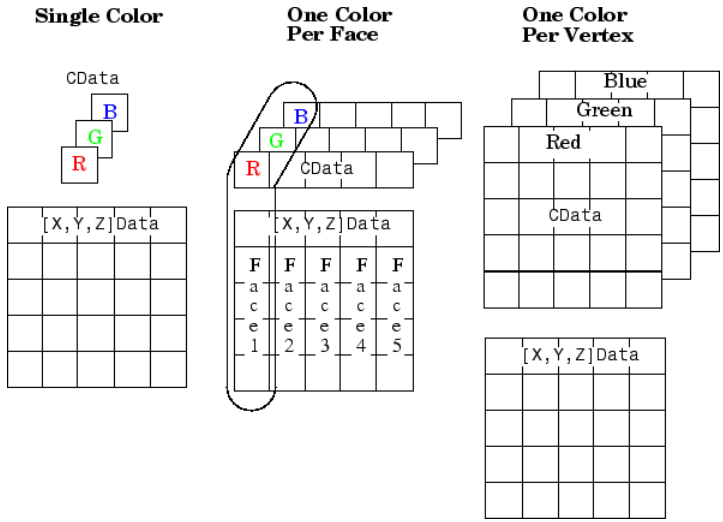
- Numeric values that are scaled to map linearly into the current colormap.
- Integer values that are used directly as indices into the current colormap.
- Arrays of RGB triplets. RGB triplets are not mapped into the current colormap, but interpreted as the colors defined.

The following diagrams illustrate the dimensions of `CData` with respect to the arrays in the `XData`, `YData`, and `ZData` properties.

These diagrams illustrates the use of indexed color.



These diagrams illustrates the use of true color. True color requires either a single RGB triplet or an array of RGB triplets.



If `CData` contains NaNs, then `patch` does not color the faces.

An alternative method for defining patches uses the `Faces`, `Vertices`, and `FaceVertexCData` properties.

Example: `[1,0,0]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`FaceVertexCData` — Face and vertex colors

`[]` (default) | single color for entire patch | one color per face | one color per vertex

Face and vertex colors, specified as a single color for the entire patch, one color per face, or one color per vertex for interpolated face color.

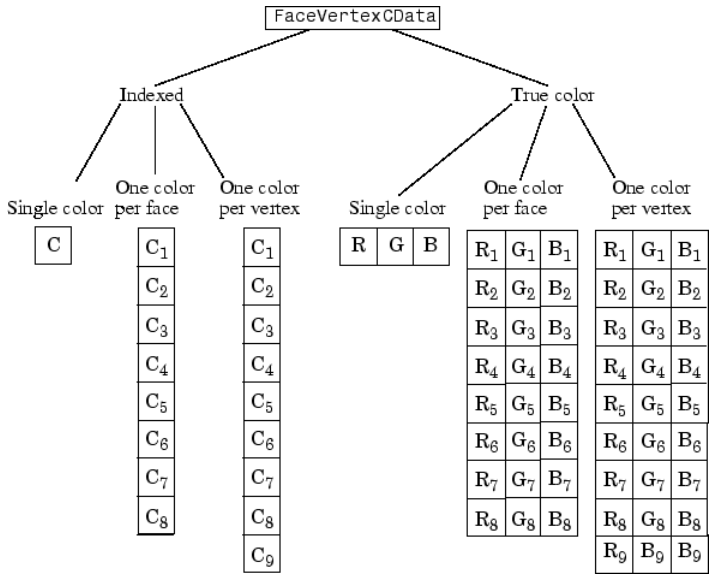
If you want to use indexed colors, then specify `FaceVertexCData` in one of these forms:

- For one color for the entire patch, use a single value.
- For one color per face, use an m-by-1 column vector, where m is the number of rows in the `Faces` property.
- For interpolated face color, use an m-by–1 column vector where m is the number of rows in the `Vertices` property.

If you want to use true colors, then specify `FaceVertexCData` in one of these forms:

- For one color for all the faces, specify a three-element row vector that defines an RGB triplet. When you do this, you must also set the `FaceColor` to `'flat'` and the `EdgeColor` to a value other than `'flat'` or `'interp'`.
- For one color per face, use an m-by-3 array of RGB triplets, where m is the number of rows in the `Faces` property.
- For interpolated face color, use an m-by-3 array, where m is the number of rows in the `Vertices` property.

The following diagram illustrates the various forms of the `FaceVertexCData` property for a patch having eight faces and nine vertices. The `CDataMapping` property determines how MATLAB interprets the `FaceVertexCData` property when you specify indexed colors.



`CDataMapping` — Direct or scaled color data mapping
`'scaled'` (default) | `'direct'`

Direct or scaled color data mapping, specified as `'scaled'` (the default) or `'direct'`. The `CData` and `FaceVertexCData` properties contains color data. If you use true color specification for `CData` or `FaceVertexCData`, then this property has no effect.

- `'direct'` — Interpret the values as indices into the current colormap. Values with a decimal portion are fixed to the nearest lower integer.

- If the values are of type `double` or `single`, then values of `1` or less map to the first color in the colormap. Values equal to or greater than the length of the colormap map to the last color in the colormap.
- If the values are of type `uint8`, `uint16`, `uint32`, `uint64` , `int8`, `int16`, `int32`, or `int64`, then values of `0` or less map to the first color in the colormap. Values equal to or greater than the length of the colormap map to the last color in the colormap (or up to the range limits of the type).
- If the values are of type `logical`, then values of `0` map to the first color in the colormap and values of `1` map to the second color in the colormap.

- `'scaled'` — Scale the values to range between the minimum and maximum color limits. The `CLim` property of the axes contains the color limits.

Transparency

`FaceAlpha` — Face transparency

1 (default) | scalar in range `[0,1]` | `'flat'` | `'interp'`

Face transparency, specified as one of these values:

- Scalar in range `[0,1]` — Use uniform transparency across all of the faces. A value of `1` is fully opaque and `0` is completely transparent. This option does not use the transparency values in the `FaceVertexAlphaData` property.
- `'flat'` — Use a different transparency for each face based on the values in the `FaceVertexAlphaData` property. First you must specify the `FaceVertexAlphaData` property as a vector containing one transparency value per face or vertex. The transparency value at the first vertex determines the transparency for the entire face.
- `'interp'` — Use interpolated transparency for each face based on the values in `FaceVertexAlphaData` property. First you must specify the `FaceVertexAlphaData` property as a vector containing one transparency value per vertex. The transparency varies across each face by interpolating the values at the vertices.

`EdgeAlpha` — Edge line transparency

1 (default) | scalar value in range `[0,1]` | `'flat'` | `'interp'`

Edge line transparency, specified as one of these values:

- Scalar value in range `[0,1]` — Use uniform transparency across all of the edges. A value of `1` is fully opaque and `0` is completely transparent. This option does not use the transparency values in the `FaceVertexAlphaData` property.

- `'flat'` — Use a different transparency for each edge based on the values in the `FaceVertexAlphaData` property. First you must specify the `FaceVertexAlphaData` property as a vector containing one transparency value per face or vertex. The transparency value at the first vertex determines the transparency for the edge.
- `'interp'` — Use interpolated transparency for each edge based on the values in `FaceVertexAlphaData` property. First you must specify the `FaceVertexAlphaData` property as a vector containing one transparency value per vertex. Vary the transparency across each edge by interpolating the values at the vertices.

`FaceVertexAlphaData` — Face and vertex transparency values
`[ ]` (default) | scalar | vector with one value per face | vector with one value per vertex

Face and vertex transparency values, specified as a scalar, a vector with one value per face, or a vector with one value per vertex.

- For uniform transparency across all of the faces or edges, specify a scalar value. Then, set the `FaceAlpha` or `EdgeAlpha` property to `'flat'`.
- For a different transparency for each face or edge, specify an `m`-by-1 vector, where `m` is the number of faces. Then, set the `FaceAlpha` or `EdgeAlpha` property to `'flat'`. To determine the number of faces, query the number of rows in the `Faces` property.
- For interpolated transparency across each face or edge, specify an `n`-by-1 vector, where `n` is the number of vertices. Then, set the `FaceAlpha` or `EdgeAlpha` property to `'interp'`. To determine the number of vertices, query the number of rows in the `Vertices` property.

The `AlphaDataMapping` property determines how the patch interprets the `FaceVertexAlphaData` property values.

Note If the `FaceAlpha` and `EdgeAlpha` properties are both set to scalar values, then the patch does not use the `FaceVertexAlphaData` values.

`AlphaDataMapping` — Interpretation of `FaceVertexAlphaData` values
`'scaled'` (default) | `'direct'` | `'none'`

Interpretation of `FaceVertexAlphaData` values, specified as one of these values:

- `'none'` — Interpret the values as transparency values. A value of 1 or greater is completely opaque, a value of 0 or less is completely transparent, and a value between 0 and 1 is semitransparent.

- `'scaled'` — Map the values into the figure's alphamap. The minimum and maximum alpha limits of the axes determine the alpha data values that map to the first and last elements in the alphamap, respectively. For example, if the alpha limits are `[3 5]`, then alpha data values less than or equal to `3` map to the first element in the alphamap. Alpha data values greater than or equal to `5` map to the last element in the alphamap. The `ALim` property of the axes contains the alpha limits. The `Alphamap` property of the figure contains the alphamap.
- `'direct'` — Interpret the values as indices into the figure's alphamap. Values with a decimal portion are fixed to the nearest lower integer.
  - If the values are of type `double` or `single`, then values of 1 or less map to the first element in the alphamap. Values equal to or greater than the length of the alphamap map to the last element in the alphamap.
  - If the values are of integer type, then values of 0 or less map to the first element in the alphamap. Values equal to or greater than the length of the alphamap map to the last element in the alphamap (or up to the range limits of the type). The integer types are `uint8`, `uint16`, `uint32`, `uint64` , `int8`, `int16`, `int32`, and `int64`.
  - If the values are of type `logical`, then values of 0 map to the first element in the alphamap and values of 1 map to the second element in the alphamap.

Line Styling

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width

`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

`AlignVertexCenters` — Sharp vertical and horizontal lines

`'off'` (default) | `'on'`

Sharp vertical and horizontal lines, specified as `'off'` or `'on'`.

If the associated figure has a `GraphicsSmoothing` property set to `'on'` and a `Renderer` property set to `'opengl'`, then the figure applies a smoothing technique to plots. In some cases, this smoothing technique can cause vertical and horizontal lines to appear uneven in thickness or color. Use the `AlignVertexCenters` property to eliminate the uneven appearance.

- `'off'` — Do not sharpen vertical or horizontal lines. The lines might appear uneven in thickness or color.
- `'on'` — Sharpen vertical and horizontal lines to eliminate an uneven appearance.

Note You must have a graphics card that supports this feature. To see if the feature is supported, call the `rendererinfo` function. If it is supported, `rendererinfo` returns value of `1` for `info.Details.SupportsAlignVertexCenters`.

Markers

`Marker` — Marker symbol

`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | ...

Marker symbol, specified as one of the values listed in this table. By default, the object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross

Value	Description
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerSize` — Marker size

6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

`MarkerEdgeColor` — Marker outline color

`'auto'` (default) | `'flat'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'`

Marker outline color, specified as `'auto'`, `'flat'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` option uses the same color as the `EdgeColor` property. The `'flat'` option uses the `CData` value at the vertex to set the color.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | `'flat'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, `'flat'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` option uses the same color as the `Color` property for the axes. The `'flat'` option uses the `CData` value of the vertex to set the color.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

This property affects only the circle, square, diamond, pentagram, hexagram, and the four triangle marker types.

Example: [0.3 0.2 0.1]

Example: 'green'

Example: '#D2F9A7'

Data

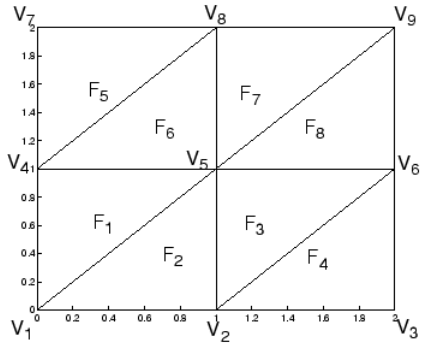
Faces — Vertex connection defining each face

vector | matrix

Vertex connection defining each face, specified as a vector or a matrix defining the vertices in the Vertices property that are to be connected to form each face. The Faces and Vertices properties provide an alternative way to specify a patch that can be more efficient than using XData, YData, and ZData coordinates in most cases.

Each row in the faces array designates the connections for a single face, and the number of elements in that row that are not NaN defines the number of vertices for that face. Therefore, an m-by-n Faces array defines m faces with up to n vertices each.

For example, consider the following patch. It is composed of eight triangular faces defined by nine vertices. The corresponding Faces and Vertices properties are shown to the right of the patch. Note how some faces share vertices with other faces. For example, the fifth vertex (V5) is used six times, once each by faces one, two, three, six, seven, and eight. Without sharing vertices, this same patch requires 24 vertex definitions.



Faces property

F_1	V_1	V_4	V_5
F_2	V_1	V_5	V_2
F_3	V_2	V_5	V_6
F_4	V_2	V_6	V_3
F_5	V_4	V_7	V_8
F_6	V_4	V_8	V_5
F_7	V_5	V_8	V_9
$F8$	V_5	V_9	V_6

Vertices property

V_1	X_1	Y_1	Z_1
V_2	X_2	Y_2	Z_2
V_3	X_3	Y_3	Z_3
V_4	X_4	Y_4	Z_4
V_5	X_5	Y_5	Z_5
V_6	X_6	Y_6	Z_6
V_7	X_7	Y_7	Z_7
V_8	X_8	Y_8	Z_8
V_9	X_9	Y_9	Z_9

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Vertices` — Vertex coordinates

vector | matrix

Vertex coordinates, specified as a vector or a matrix defining the (*x*,*y*,*z*) coordinates of each vertex. The `Faces` and `Vertices` properties provide an alternative way to specify a patch that can be more efficient than using `XData`, `YData`, and `ZData` coordinates in most cases. See the `Faces` property for a description of how the vertex data is used.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`XData` — x-coordinates of the patch vertices

vector | matrix

The *x*-coordinates of the patch vertices, specified as a vector or a matrix. If `XData` is a matrix, then each column represents the *x*-coordinates of a single face of the patch. In this case, `XData`, `YData`, and `ZData` must have the same dimensions.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`YData` — y-coordinates of the patch vertices

vector | matrix

The *y*-coordinates defining the patch, specified as a vector or a matrix. If `YData` is a matrix, then each column represents the *y*-coordinates of a single face of the patch. In this case, `XData`, `YData`, and `ZData` must have the same dimensions.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ZData` — z-coordinates of the patch vertices
vector | matrix

The *z*-coordinates of the patch vertices, specified as a vector or a matrix. If `ZData` is a matrix, then each column represents the *z*-coordinates of a single face of the patch. In this case, `XData`, `YData`, and `ZData` must have the same dimensions.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Normals

`VertexNormals` — Vertex normal vectors
m-by-n-by-3 array (default) | array of normal vectors

Vertex normal vectors, specified as an array of normal vectors with one normal vector one per patch vertex. Define one normal per patch vertex, as determined by the size of the `Vertices` property value. Vertex normals determine the shape and orientation of the patch. This data is used for lighting calculations.

Specifying values for this property sets the associated mode to manual. If you do not specify normal vectors, then the patch generates this data when the axes contains light objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`VertexNormalsMode` — Selection mode for `VertexNormals`
`'auto'` (default) | `'manual'`

Selection mode for `VertexNormals`, specified as one of these values:

- `'auto'` — The `patch` function calculates vertex normals when you add a light to the scene.
- `'manual'` — Use the vertex normal data specified by the `VertexNormals` property. Assigning values to the `VertexNormals` property sets `VertexNormalsMode` to `'manual'`.

`FaceNormals` — Face normal vectors

m-by-n-by-3 array (default) | array of normal vectors

Face normal vectors, specified as an array of normal vectors with one normal vector one per patch face. Define one normal per patch face, as determined by the size of the `Faces` property value. Face normals determine the orientation of each patch face. This data is used for lighting calculations.

Specifying values for this property sets the associated mode to manual. If you do not specify normal vectors, then the patch generates this data when the axes contains light objects. The patch computes face normals using Newell's method.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`FaceNormalsMode` — Selection mode for `FaceNormals`

`'auto'` (default) | `'manual'`

Selection mode for `FaceNormals`, specified as one of these values:

- `'auto'` — The `patch` function calculates face normals when you add a light to the scene.
- `'manual'` — Use the face normal data specified by the `FaceNormals` property. Assigning values to the `FaceNormals` property sets `FaceNormalsMode` to `'manual'`.

Lighting

`FaceLighting` — Effect of light objects on faces

`'flat'` (default) | `'gouraud'` | `'none'`

Effect of light objects on faces, specified as one of these values:

- `'flat'` — Apply light uniformly across each face. Use this value to view faceted objects.
- `'gouraud'` — Vary the light across the faces. Calculate the light at the vertices and then linearly interpolate the light across the faces. Use this value to view curved surfaces.
- `'none'` — Do not apply light from light objects to the faces.

To add a light object to the axes, use the `light` function.

Note The `'phong'` value has been removed. Use `'gouraud'` instead.

BackFaceLighting — Face lighting when normals point away from camera
`'reverselit'` (default) | `'unlit'` | `'lit'`

Face lighting when the vertex normals point away from camera, specified as one of these values:

- `'reverselit'` — Light the face as if the vertex normal pointed towards the camera.
- `'unlit'` — Do not light the face.
- `'lit'` — Light the face according to the vertex normal.

Use this property to discriminate between the internal and external surfaces of an object. For an example, see “Back Face Lighting”.

EdgeLighting — Effect of light objects on edges
`'none'` (default) | `'flat'` | `'gouraud'`

Effect of light objects on edges, specified as one of these values:

- `'flat'` — Apply light uniformly across the each edges.
- `'none'` — Do not apply lights from light objects to the edges.
- `'gouraud'` — Calculate the light at the vertices, and then linearly interpolate across the edges.

Note The `'phong'` value has been removed. Use `'gouraud'` instead.

AmbientStrength — Strength of ambient light
`0.3` (default) | scalar in range `[0,1]`

Strength of ambient light, specified as a scalar value in the range `[0,1]`. Ambient light is a nondirectional light that illuminates the entire scene. There must be at least one visible light object in the axes for the ambient light to be visible.

The `AmbientLightColor` property for the axes sets the color of the ambient light. The color is the same for all objects in the axes.

Example: `0.5`

Data Types: `double`

DiffuseStrength — Strength of diffuse light
`0.6` (default) | scalar in range `[0,1]`

Strength of diffuse light, specified as a scalar value in the range `[0,1]`. Diffuse light is the nonspecular reflectance from light objects in the axes.

Example: `0.3`

Data Types: `double`

`SpecularStrength` — Strength of specular reflection
`0.9` (default) | scalar in range `[0,1]`

Strength of specular reflection, specified as a scalar value in the range `[0,1]`. Specular reflections are the bright spots on the surface from light objects in the axes.

Example: `0.3`

Data Types: `double`

`SpecularExponent` — Expansiveness of specular reflection
`10` (default) | scalar value greater than `0`

Expansiveness of specular reflection, specified as a scalar value greater than `0`. `SpecularExponent` controls the size of the specular reflection spot. Greater values produce less specular reflection.

Most materials have exponents in the range of `5` to `20`.

Example: `17`

Data Types: `double`

`SpecularColorReflectance` — Color of specular reflections
`1` (default) | scalar between `0` and `1` inclusive

Color of specular reflections, specified as a scalar between `0` and `1` inclusive.

- `0` — The color of the specular reflection depends on both the color of the object from which it reflects and the color of the light source.
- `1` — The color of the specular reflection depends only on the color or the light source (that is, the light object `Color` property).

The contributions from the light source color and the patch color to the specular reflection color vary linearly for values between `0` and `1`.

Example: `0.5`

Data Types: `single` | `double`

Legend

DisplayName — Legend label

`''` (default) | character vector | string scalar

Legend label, specified as a character vector or string scalar. The legend does not display until you call the `legend` command. If you do not specify the text, then `legend` sets the label using the form `'dataN'`.

Annotation — Control for including or excluding object from legend

`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

Visible — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

DataTipTemplate — Data tip content

`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

This property applies only to patches with pinned data tips.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

UIContextMenu — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

Selected — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

SelectionHighlight — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

`Clipping` — Clipping of object to axes limits
`'on'` (default) | `'off'`

Clipping of the object to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the object that are outside the axes limits.
- `'off'` — Display the entire object, even if parts of it appear outside the axes limits. Parts of the object might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the object so that it is larger than the original plot.

The `Clipping` property of the axes that contains the object must be set to `'on'`. Otherwise, this property has no effect. For more information about the clipping behavior, see the `Clipping` property of the axes.

Callbacks

`ButtonDownFcn` — Mouse-click callback
`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.

  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.

- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks when visible. The `Visible` property must be set to `'on'` and you must click a part of the `Patch` object that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The `HitTest` property determines if the `Patch` object responds to the click or if an ancestor does.
- `'all'` — Capture mouse clicks regardless of visibility. The `Visible` property can be set to `'on'` or `'off'` and you can click a part of the `Patch` object that has no color. The `HitTest` property determines if the `Patch` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `Patch` object passes the click through it to the object below it in the current view of the figure window. The `HitTest` property has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `Patch` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `Patch` object that has one of these:
  - `HitTest` property set to `'on'`
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

---

Note The `PickableParts` property determines if the `Patch` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent

`Axes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `Group`, or `Transform` object.

`Children` — Children

empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the chart object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object
`'patch'`

This property is read-only.

Type of graphics object, returned as `'patch'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[ ]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`patch`

Topics

"Access Property Values"
"Graphics Object Properties"

Introduced before R2006a

path

View or change search path

Syntax

```
path
path(newpath)
path(oldpath,newfolder)
path(newfolder,oldpath)
p = path( ___ )
```

Description

`path` displays the MATLAB search path, which is stored in `pathdef.m`.

`path(newpath)` changes the search path to `newpath`.

`path(oldpath,newfolder)` adds the folder `newfolder` to the end of the search path. If `newfolder` is already on the search path, then `path(oldpath,newfolder)` moves `newfolder` to the end of the search path. To add multiple folders, use the `addpath` function.

`path(newfolder,oldpath)` adds the folder `newfolder` to the beginning of the search path. If `newfolder` is already on the search path, then `path(oldpath,newfolder)` moves `newfolder` to the beginning of the search path.

`p = path( ___ )` returns the MATLAB search path as a character vector. You can use this syntax with any of the input argument combinations in the previous syntaxes.

Examples

Display Search Path

Suppose you have MATLAB R2017a installed. Display the search path.

```
path

        MATLABPATH

    C:\Program Files\MATLAB\R2017a\toolbox\matlab\datafun
    C:\Program Files\MATLAB\R2017a\toolbox\matlab\datatypes
    C:\Program Files\MATLAB\R2017a\toolbox\matlab\elfun
    C:\Program Files\MATLAB\R2017a\toolbox\matlab\elmat
    C:\Program Files\MATLAB\R2017a\toolbox\matlab\funfun
    C:\Program Files\MATLAB\R2017a\toolbox\matlab\general
    C:\Program Files\MATLAB\R2017a\toolbox\matlab\iofun
    C:\Program Files\MATLAB\R2017a\toolbox\matlab\lang
    C:\Program Files\MATLAB\R2017a\toolbox\matlab\matfun
...
```

Add New Folder to Search Path on Windows

Add the folder `c:\tools\goodstuff` to the bottom of the search path on a Microsoft Windows system.

```
oldpath = path;
path(oldpath,'c:\tools\goodstuff')
```

Add New Folder to Search Path on UNIX

Add the folder `/home/tools/goodstuff` to the beginning of the search path on a UNIX system.

```
oldpath = path;
path('/home/tools/goodstuff',oldpath)
```

Temporarily Add Folder to Search Path

Temporarily add the folder `my_files` to the search path and then run `my_function` in `my_files`.

```
oldpath = path;
path(oldpath,'my_files')
my_function
```

Restore the previous search path.

```
path(oldpath)
```

Input Arguments

`newpath` — New MATLAB search path
character array | string array

New MATLAB search path, specified as a character array or string array.

MATLAB resolves all path names containing `'.'`, `'..'`, and symbolic links to their target location before adding them to the path. This ensures that each entry in the MATLAB path represents a unique folder location. For example, if `newpath` contains `c:\matlab\..\work`, the new MATLAB search path contains `c:\work`.

Data Types: `char` | `string`

`oldpath` — Existing MATLAB search path
character array | string array

Existing MATLAB search path, specified as a character array or string array returned by the `path` command.

Data Types: `char` | `string`

`newfolder` — New folder
character vector | string scalar

New folder to add to the MATLAB search path, specified as a character vector or string scalar.

Data Types: `char` | `string`

Alternative Functionality

As an alternative to the `path` function, use the Set Path dialog box. To open the Set Path dialog box, on the **Home** tab, in the **Environment** section, click **Set Path**.

You can also use the `addpath` function to add multiple folders to the search path.

See Also

`addpath` | `matlabroot` | `pathtool` | `rehash` | `restoredefaultpath` | `rmpath`

Topics

"What Is the MATLAB Search Path?"
"Files and Folders that MATLAB Accesses"

Introduced before R2006a

path2rc

Save current search path to `pathdef.m` file

Syntax

```
path2rc
```

Description

`path2rc` runs `savepath`. The `savepath` function is replacing `path2rc`. Use `savepath` instead of `path2rc` and replace instances of `path2rc` with `savepath`.

Introduced before R2006a

pathsep

Search path separator for current platform

Syntax

```
c = pathsep
```

Description

`c = pathsep` returns the search path separator character for the current platform. The search path separator is the character that separates path names in the `pathdef.m` file, as returned by the `path` function. On Windows, the character is a semicolon (`;`). On UNIX the character is a colon (`:`). Use `pathsep` to work programmatically with the content of the search path file.

Examples

Get List of Subfolders Within Path Name

Use `genpath` in conjunction with `regexp` and `pathsep` to get a list of individual folders within a path name.

Create a folder `myfolder` containing a subfolder `mysubfolder`.

```
mkdir myfolder;
cd myfolder;
mkdir mysubfolder;
cd ..
```

Generate a path name that includes `myfolder` and all folders below it.

```
p = genpath('myfolder')
```

```
p =
'myfolder;myfolder\mysubfolder;'
```

Use the `regexp` function to split the path into individual folders.

```
s = regexp(p, pathsep, 'split')
```

```
s = 1x3 cell array
    {'myfolder'}    {'myfolder\mysubf...'}    {0x0 char}
```

See Also

`fileparts` | `filesep` | `fullfile` | `path`

Topics

"What Is the MATLAB Search Path?"

Introduced before R2006a

pathtool

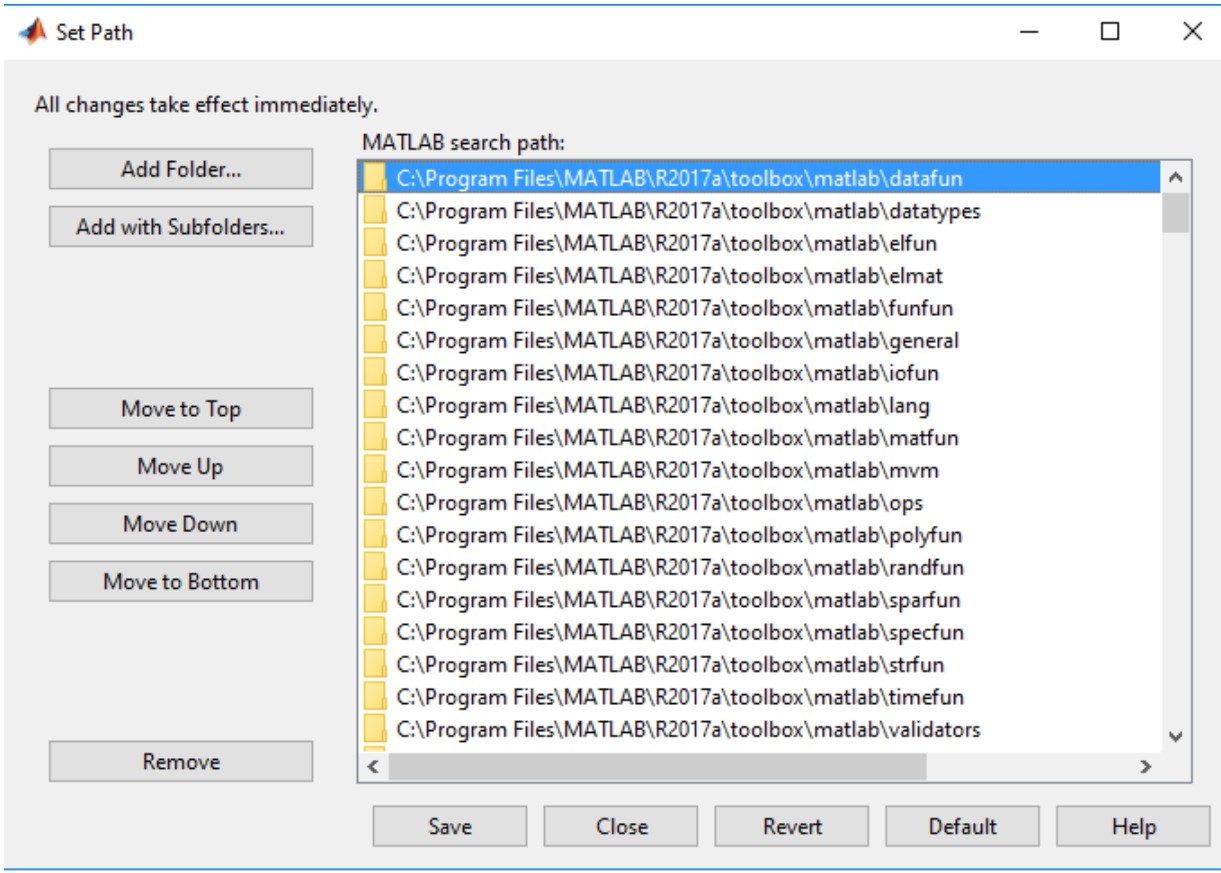
Open Set Path dialog box to view and change search path

Syntax

```
pathtool
```

Description

`pathtool` opens the Set Path dialog box. You can use the Set Path dialog box to view, modify, and save the MATLAB search path.



Alternative Functionality

Alternatively, to open the Set Path dialog box, go the **Home** tab and, in the **Environment** section, click **Set Path**.

See Also

`addpath` | `path` | `rehash` | `restoredefaultpath` | `rmpath` | `savepath`

Topics

"What Is the MATLAB Search Path?"

Introduced before R2006a

pause

Stop MATLAB execution temporarily

Syntax

```
pause

pause(n)

pause(state)
oldState = pause(state)
```

Description

`pause` temporarily stops MATLAB execution and waits for the user to press any key. The `pause` function also temporarily stops the execution of Simulink models, but does not pause their repainting.

Note If you previously disabled the pause setting, reenable it using `pause('on')` for this call to take effect.

`pause(n)` pauses execution for `n` seconds before continuing. Pausing must be enabled for this call to take effect.

`pause(state)` enables, disables, or displays the current pause setting.

`oldState = pause(state)` returns the current pause setting and sets the pause state as indicated by `state`. For example, if pausing is enabled, `oldState = pause('off')` returns `'on'` in `oldState` and disables pausing.

Examples

Pause Execution

Pause execution for 5 seconds. MATLAB blocks, or hides, the command prompt (>>) while it pauses execution.

```
n = 5;
pause(n)
```

Disable Pause Setting

Disable the pause setting and query the current state.

```
pause('off')
pause('query')
```

```
ans =
'off'
```

Pause execution for 100 seconds. Since the pause setting is off, MATLAB ignores the request to pause execution, and immediately returns the command prompt.

```
pause(100)
```

Enable the pause setting.

```
pause('on')
```

Save and Restore Pause State

Store the current pause setting and then disable the ability to pause execution.

```
oldState = pause('off')
```

```
oldState =
'on'
```

Query the current pause setting.

```
pause('query')
```

```
ans =
'off'
```

Restore the initial pause state.

```
pause(oldState)
pause('query')
```

```
ans =
'on'
```

Alternatively, you can store the queried value of the pause state and then disable the ability to pause execution.

```
oldState = pause('query');
pause('off')
```

Restore the initial pause state.

```
pause(oldState)
```

Input Arguments

n — Number of seconds
nonnegative, real number

Number of seconds to pause execution specified as a nonnegative, real number.

Typing `pause(inf)` puts you into an infinite loop. To return to the MATLAB prompt, type **Ctrl+C**.

Example: `pause(3)` pauses for 3 seconds.

Example: `pause(5/1000)` pauses for 5 milliseconds.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

state — Pause setting
`'on'` | `'off'` | `'query'`

Pause control indicator specified as `'on'`, `'off'`, or `'query'`. Use `'on'` or `'off'` to control whether the `pause` function is able to pause MATLAB execution. Use `'query'` to query the current state of the pause setting.

To run interactive code unattended, disable the pause setting.

Tips

- The accuracy of the `pause` function is subject to the scheduling resolution of your operating system, and to other concurrent system activity. The accuracy is not guaranteed, and finer resolution results in higher relative error.
- While MATLAB is paused, the following continue to execute:
  - Repainting of figure windows, Simulink block diagrams, and Java windows
  - Callbacks from figure windows
  - Event handling from Java windows

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The use of `pause` in `parfor` loops is not supported for MEX code generation.
- The generated code truncates pause delay values to `uint32` range during run-time execution.
- In generated standalone (`lib`, `dll`, or `exe`) code, the `pause` syntax with no input arguments has different behavior than in MATLAB.
  - In MATLAB, after `pause` execution you can press any key to continue. When running generated code from the terminal, you must press enter.
  - If you call the generated code from MATLAB with the `system` command or exclamation point character, with SIL, or with PIL, input characters may not be

correctly redirected from the MATLAB environment to the generated code. Therefore the generated code might not resume.

- The C/C++ implementation for `pause` in the generated code differs depending on the hardware settings stored in the code generation configuration object. By default, the hardware settings are configured for the host platform, with `Hardware Board` in the MATLAB Coder app set to `MATLAB Host Computer`.
  - When generating code on Windows with Windows-compatible hardware settings, the generated C/C++ implementation uses the Windows API function `Sleep`.
  - In all other cases, the implementation uses the POSIX API `nanosleep`. When compiling code that uses the POSIX API, the preprocessor macro `_POSIX_C_SOURCE` must be set to an integer greater than or equal to `199309L`. The code generator sets the macro to `199309L` for compilation.

See Also

`drawnow` | `input` | `keyboard`

Introduced before R2006a

pbaspect

Control relative lengths of each axis

Syntax

```
pbaspect(ratio)
pb = pbaspect

pbaspect auto
pbaspect manual
m = pbaspect('mode')

___ = pbaspect(ax, ___ )
```

Description

`pbaspect(ratio)` sets the plot box aspect ratio for the current axes. The plot box aspect ratio is the relative length of the *x*-axis, *y*-axis, and *z*-axis. Specify `ratio` as a three-element vector of positive values that represent the ratio of the *x*-axis, *y*-axis, and *z*-axis lengths. For example, `[3 1 1]` specifies that the length of the *x*-axis is equal to three times the length of the *y*-axis and *z*-axis. For equal axis lengths in all directions, use `[1 1 1]`.

`pb = pbaspect` returns the plot box aspect ratio for the current axes.

`pbaspect auto` sets an automatic mode, enabling the axes to choose the appropriate plot box aspect ratio. The mode must be automatic to enable the “stretch-to-fill on page 1-11003” feature of the axes.

`pbaspect manual` sets a manual mode and uses the ratio stored in the `PlotBoxAspectRatio` property of the `Axes` object. When the mode is manual, it disables the stretch-to-fill feature of the axes. Specifying a value for the plot box aspect ratio sets the mode to manual.

`m = pbaspect('mode')` returns the current mode, which is either `'auto'` or `'manual'`. By default, the mode is automatic unless you specify the plot box aspect ratio or set the mode to manual.

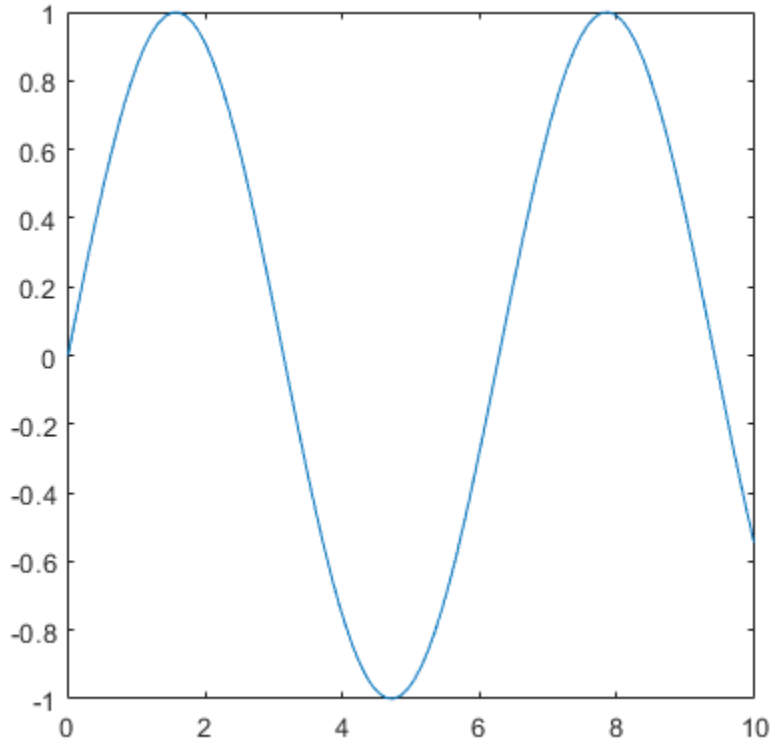
`___ = pbaspect(ax, ___ )` uses the axes specified by `ax` instead of the current axes. Specify the axes as the first input argument for any of the previous syntaxes. Use single quotes around the `'auto'` and `'manual'` inputs.

Examples

Use Equal Axis Lengths

Create a simple line plot. Make the *x*-axis, *y*-axis, and *z*-axis (not shown) equal lengths.

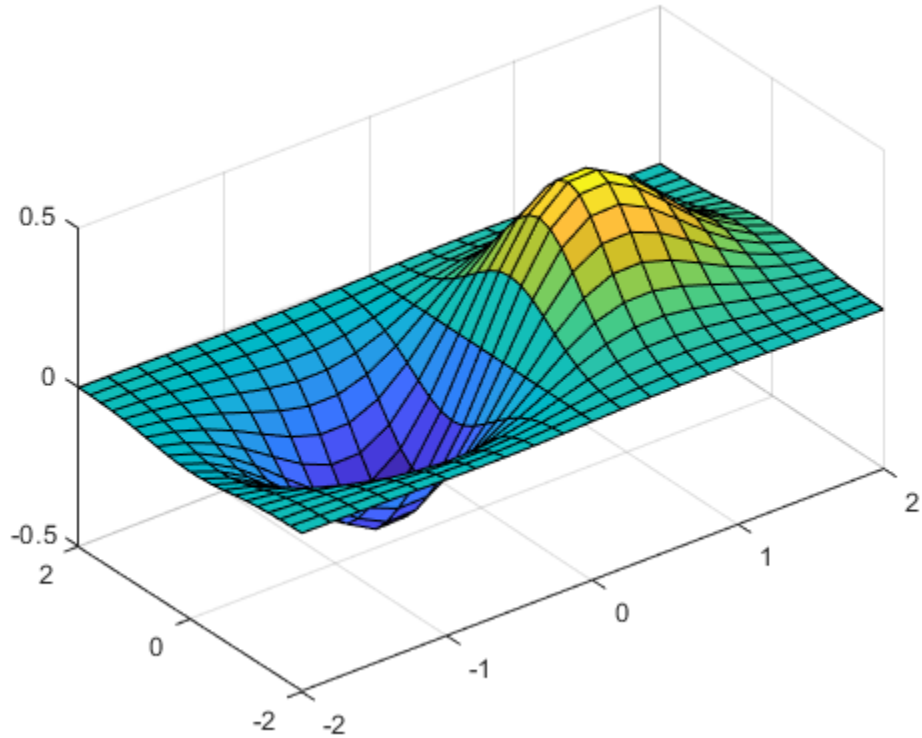
```
x = linspace(0,10);
y = sin(x);
plot(x,y)
pbaspect([1 1 1])
```



Use Different Axis Lengths

Plot a surface and make the *x*-axis twice as long as the *y*-axis and *z*-axis.

```
[x,y] = meshgrid(-2:.2:2);
z = x.*exp(-x.^2 - y.^2);
surf(x,y,z)
pbaspect([2 1 1])
```



When you specify the plot box aspect ratio, the associated mode changes to manual. Query the current plot box aspect ratio mode.

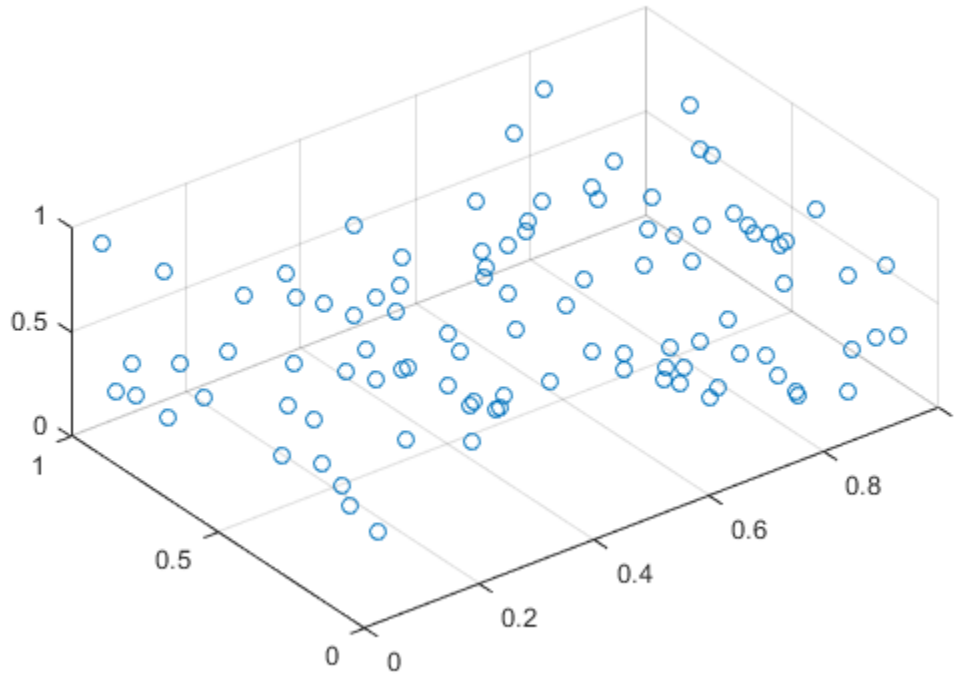
```
m = pbaspect('mode')
```

```
m =
'manual'
```

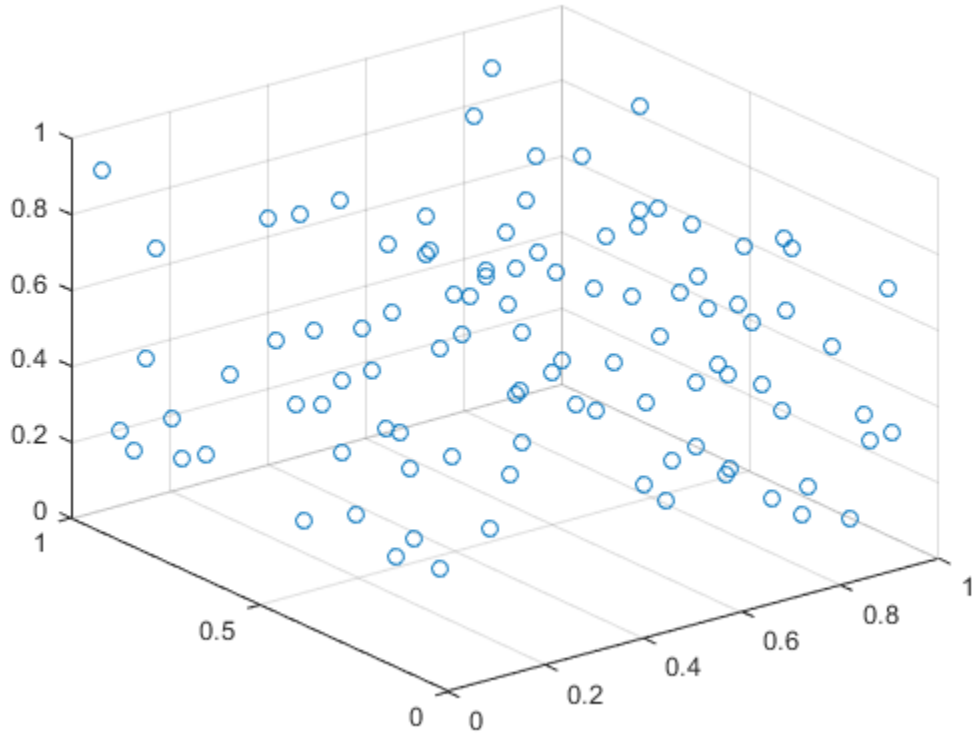
Revert Back to Default Plot Box Aspect Ratio

Create a 3-D scatter chart of random data and set the plot box aspect ratio. Then revert back to the default plot box aspect ratio.

```
X = rand(100,1);
Y = rand(100,1);
Z = rand(100,1);
scatter3(X,Y,Z)
pbaspect([3 2 1])
```



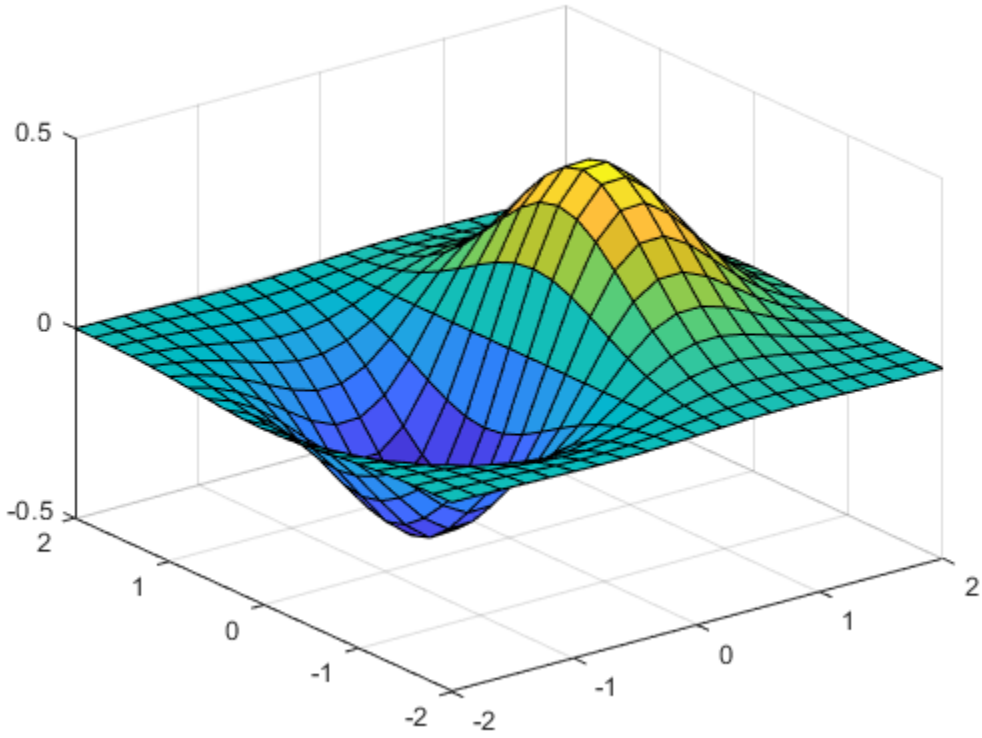

```
pbaspect auto
```



Query Plot Box Aspect Ratio

Create a surface plot.

```
[x,y] = meshgrid(-2:.2:2);
z = x.*exp(-x.^2 - y.^2);
surf(x,y,z)
```



Query the plot box aspect ratio. The values indicate the ratio of the *x*-axis length to *y*-axis length to *z*-axis length.

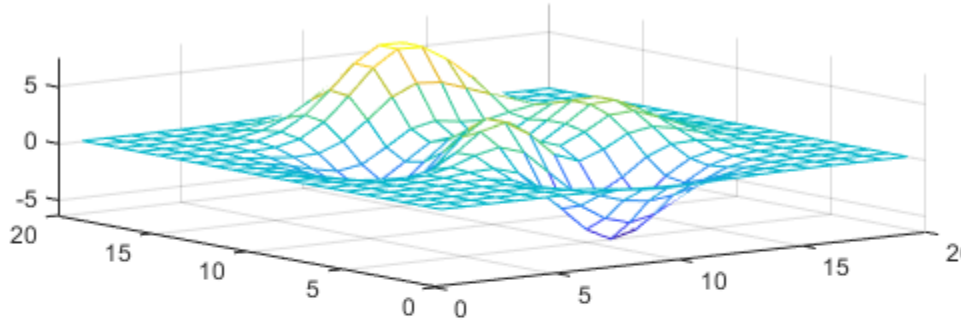
```
pb = pbaspect
```

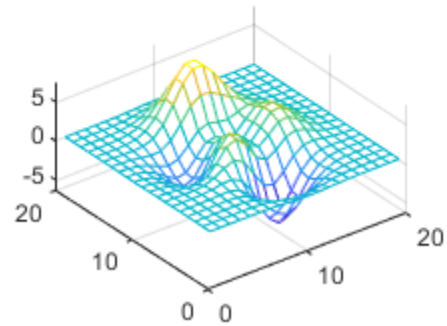
```
pb = 1×3

    1.0000    0.9419    0.8518
```

Set Plot Box Aspect Ratio for Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Plot into each of the axes. Then set the plot box aspect ratio for lower plot by specifying `ax2` as the first input argument to the `pbaspect` function.

```
tiledlayout(2,1)
ax1 = nexttile;
mesh(peaks(20))

ax2 = nexttile;
mesh(peaks(20))
pbaspect(ax2,[2 2 1])
```





Input Arguments

`ratio` — Plot box aspect ratio
three-element vector

Plot box aspect ratio, specified as a three-element vector of positive values. For example, `pbaspect([3 2 1])` specifies that the ratio of the *x*-axis length to *y*-axis length to *z*-axis length is 3 to 2 to 1. Thus, the *x*-axis is the longest and the *z*-axis is the shortest.

Specifying the ratio sets the `PlotBoxAspectRatio` property for the `Axes` object to the specified value. The `PlotBoxAspectRatio` property interacts with the

`DataAspectAspectRatio`, `XLim`, `YLim`, and `ZLim` properties to control the length and scale of the *x*-axis, *y*-axis, and *z*-axis.

Specifying the ratio sets the `PlotBoxAspectRatioMode` property to `'manual'` and disables the stretch-to-fill on page 1-11003 behavior of the axes.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ax` — Target axes
`Axes` object

Target axes, specified as an `Axes` object. If you do not specify the axes, then `pbaspect` uses the current axes (`gca`).

Output Arguments

`pb` — Current plot box aspect ratio values
three-element vector

Current plot box aspect ratio, returned as a three-element vector of values representing the ratio of the *x*-axis, *y*-axis, and *z*-axis lengths.

Querying the plot box aspect ratio returns the `PlotBoxAspectRatio` property value for the `Axes` object.

`m` — Current plot box aspect ratio mode
`'auto'` | `'manual'`

Current plot box aspect ratio mode, returned as either `'auto'` or `'manual'`. When the mode is automatic, MATLAB determines the appropriate plot box aspect ratio value. If you specify a value, then the mode changes to manual.

Querying the plot box aspect ratio mode returns the `PlotBoxAspectRatioMode` property value for the `Axes` object.

More About

Stretch-to-Fill

When the “stretch-to-fill” behavior is enabled, MATLAB stretches the axes to fill the available space. The axes might not exactly match the data aspect ratio, plot box aspect ratio, and camera-view angle values stored in its `DataAspectRatio`, `PlotBoxAspectRatio`, and `CameraViewAngle` properties.

If you specify the data aspect ratio, plot box aspect ratio, or camera-view angle, then the “stretch-to-fill” behavior is disabled. When stretch-to-fill is disabled, MATLAB makes the axes as large as possible within the available space and strictly adheres to the property values so that there is no distortion. For more information, see “Control Axes Layout”.

Algorithms

The `pbaspect` function sets and queries several axes properties related to the data aspect ratio.

- `PlotBoxAspectRatio` — Property that stores the plot box aspect ratio value.
- `PlotBoxAspectRatioMode` — Property that stores the plot box aspect ratio mode. When you set the ratio, this property changes to `'manual'`.

See Also

`axis` | `daspect` | `xlim` | `ylim` | `zlim`

Topics

“Control Ratio of Axis Lengths and Data Unit Lengths”

Introduced before R2006a

pcg

Preconditioned conjugate gradients method

Syntax

```
x = pcg(A,b)
pcg(A,b,tol)
pcg(A,b,tol,maxit)
pcg(A,b,tol,maxit,M)
pcg(A,b,tol,maxit,M1,M2)
pcg(A,b,tol,maxit,M1,M2,x0)
[x,flag] = pcg(A,b,...)
[x,flag,relres] = pcg(A,b,...)
[x,flag,relres,iter] = pcg(A,b,...)
[x,flag,relres,iter,resvec] = pcg(A,b,...)
```

Description

`x = pcg(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be symmetric and positive definite, and should also be large and sparse. The column vector `b` must have length `n`. You also can specify `A` to be a function handle, `afun`, such that `afun(x)` returns `A*x`.

"Parameterizing Functions" explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `pcg` converges, a message to that effect is displayed. If `pcg` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`pcg(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `pcg` uses the default, `1e-6`.

`pcg(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `pcg` uses the default, `min(n,20)`.

`pcg(A,b,tol,maxit,M)` and `pcg(A,b,tol,maxit,M1,M2)` use symmetric positive definite preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `pcg` applies no preconditioner. `M` can be a function handle `mfun` such that `mfun(x)` returns `M\x`.

`pcg(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is `[]`, then `pcg` uses the default, an all-zero vector.

`[x,flag] = pcg(A,b,...)` also returns a convergence flag.

Flag	Convergence
`0`	`pcg` converged to the desired tolerance `tol` within `maxit` iterations.
`1`	`pcg` iterated `maxit` times but did not converge.
`2`	Preconditioner `M` was ill-conditioned.
`3`	`pcg` stagnated. (Two consecutive iterates were the same.)
`4`	One of the scalar quantities calculated during `pcg` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = pcg(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = pcg(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = pcg(A,b,...)` also returns a vector of the residual norms at each iteration including `norm(b-A*x0)`.

Examples

Using pcg with Large Matrices

This example shows how to use pcg with a matrix input and with a function handle.

```
n1 = 21;
A = gallery('moler',n1);
```

```
b1 = sum(A,2);
tol = 1e-6;
maxit = 15;
M1 = spdiags((1:n1)',0,n1,n1);
[x1,flag1,rr1,iter1,rv1] = pcg(A,b1,tol,maxit,M1);
```

Alternatively, you can use the following function in place of the matrix `A`:

```
function y = applyMoler(x)
y = x;
y(end-1:-1:1) = y(end-1:-1:1) - cumsum(y(end:-1:2));
y(2:end) = y(2:end) - cumsum(y(1:end-1));
```

By using this function, you can solve larger systems more efficiently as there is no need to store the entire matrix `A`:

```
n2 = 21;
b2 = applyMoler(ones(n2,1));
tol = 1e-6;
maxit = 15;
M2 = spdiags((1:n2)',0,n2,n2);
[x2,flag2,rr2,iter2,rv2] = pcg(@applyMoler,b2,tol,maxit,M2);
```

Using `pcg` with a Preconditioner

This example demonstrates how to use a preconditioner matrix with `pcg`.

Create an input matrix and try to solve the system with `pcg`.

```
A = delsq(numgrid('S',100));
b = ones(size(A,1),1);
[x0,fl0,rr0,it0,rv0] = pcg(A,b,1e-8,100);
```

`fl0` is `1` because `pcg` does not converge to the requested tolerance of `1e-8` within the requested maximum 100 iterations. A preconditioner can make the system converge more quickly.

Use `ichol` with only one input argument to construct an incomplete Cholesky factorization with zero fill.

```
L = ichol(A);
[x1,fl1,rr1,it1,rv1] = pcg(A,b,1e-8,100,L,L');
```

`fl1` is `0` because `pcg` drives the relative residual to `9.8e-09` (the value of `rr1`) which is less than the requested tolerance of `1e-8` at the seventy-seventh iteration (the value of

`it1`) when preconditioned by the zero-fill incomplete Cholesky factorization. `rv1(1) = norm(b)` and `rv1(78) = norm(b-A*x1)`.

The previous matrix represents the discretization of the Laplacian on a 100x100 grid with Dirichlet boundary conditions. This means that a modified incomplete Cholesky preconditioner might perform even better.

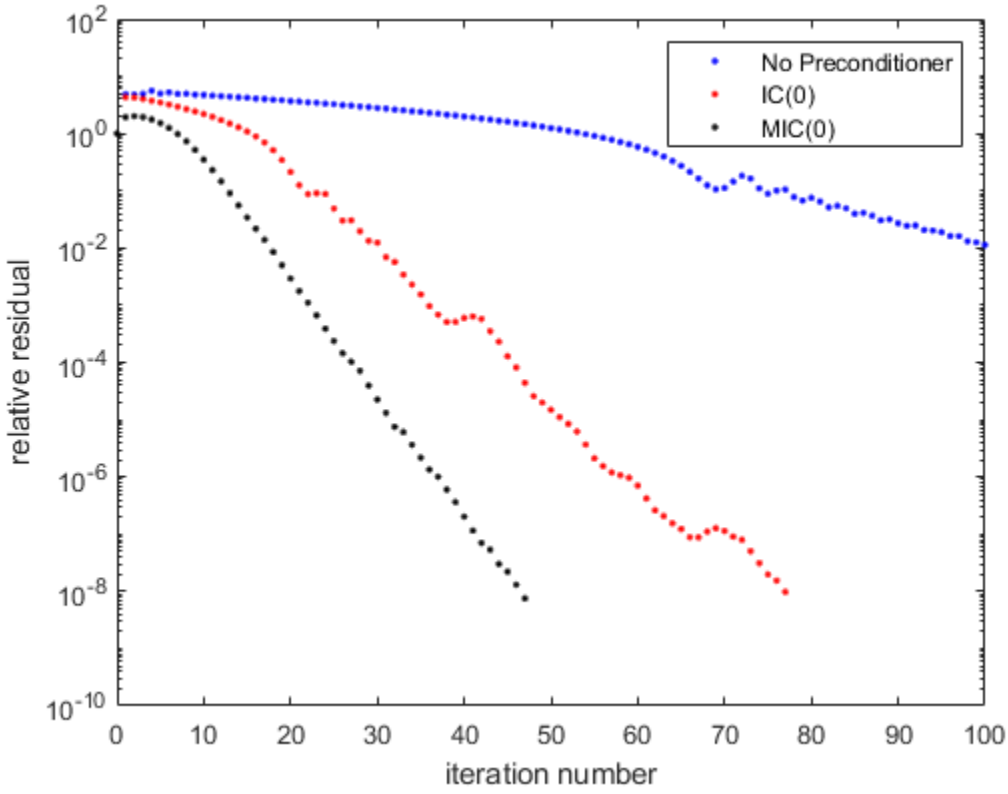
Use the `michol` option to create a modified incomplete Cholesky preconditioner.

```
L = ichol(A,struct('michol','on'));
[x2,fl2,rr2,it2,rv2] = pcg(A,b,1e-8,100,L,L');
```

In this case you attain convergence in only forty-seven iterations.

You can see how the preconditioners affect the rate of convergence of `pcg` by plotting each of the residual histories starting from the initial estimate (iterate number `0`).

```
figure;
semilogy(0:it0,rv0/norm(b),'b.');
hold on;
semilogy(0:it1,rv1/norm(b),'r.');
semilogy(0:it2,rv2/norm(b),'k.');
legend('No Preconditioner','IC(0)','MIC(0)');
xlabel('iteration number');
ylabel('relative residual');
hold off;
```



References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When input `A` is a sparse matrix:
  - Only one sparse matrix preconditioner `M` is supported.
  - If you use two preconditioners, `M1` and `M2`, then both of them must be functions.
  - For GPU arrays, `pcg` does not detect stagnation (Flag 3). Instead, it reports failure to converge (Flag 1).

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- If `M1` is a function, then it is applied independently to each row.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bicg` | `bicgstab` | `cgs` | `gmres` | `ichol` | `lsqr` | `minres` | `mldivide` | `qmr` | `symmlq`

Topics

“Create Function Handle”

Introduced before R2006a

pchip

Piecewise Cubic Hermite Interpolating Polynomial (PCHIP)

Syntax

```
p = pchip(x,y,xq)
pp = pchip(x,y)
```

Description

`p = pchip(x,y,xq)` returns a vector of interpolated values `p` corresponding to the query points in `xq`. The values of `p` are determined by shape-preserving piecewise cubic interpolation on page 1-11017 of `x` and `y`.

`pp = pchip(x,y)` returns a piecewise polynomial structure for use with `ppval` and the spline utility `unmkpp`.

Examples

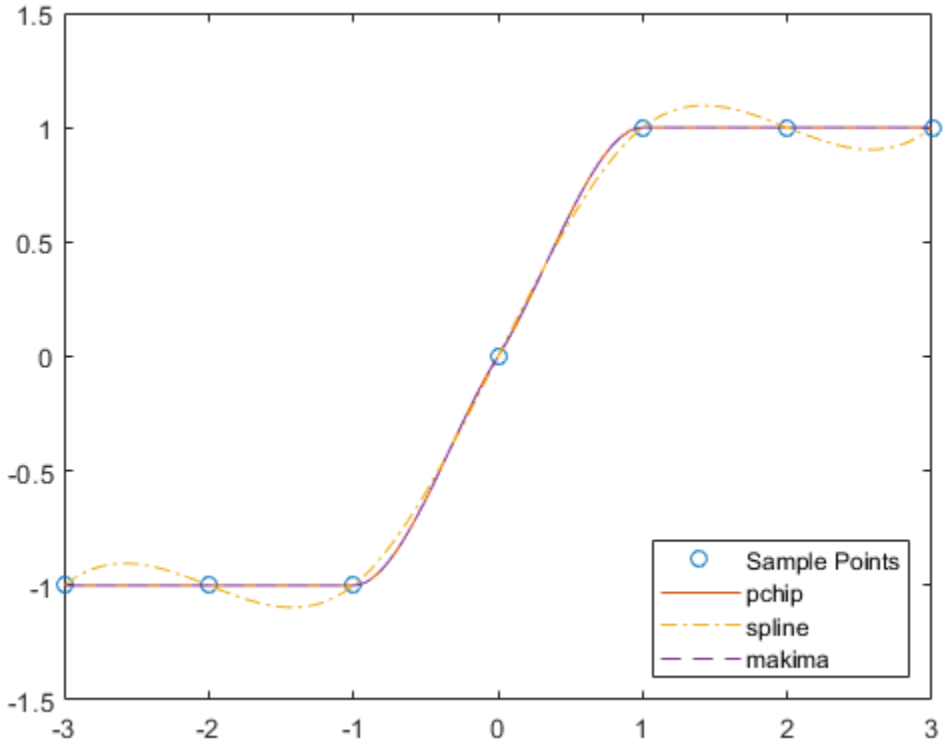
Data Interpolation with `spline`, `pchip`, and `makima`

Compare the interpolation results produced by `spline`, `pchip`, and `makima` for two different data sets. These functions all perform different forms of piecewise cubic Hermite interpolation. Each function differs in how it computes the slopes of the interpolant, leading to different behaviors when the underlying data has flat areas or undulations.

Compare the interpolation results on sample data that connects flat regions. Create vectors of `x` values, function values at those points `y`, and query points `xq`. Compute interpolations at the query points using `spline`, `pchip`, and `makima`. Plot the interpolated function values at the query points for comparison.

```
x = -3:3;
y = [-1 -1 -1 0 1 1 1];
xq1 = -3:.01:3;
```

```
p = pchip(x,y,xq1);
s = spline(x,y,xq1);
m = makima(x,y,xq1);
plot(x,y,'o',xq1,p,'-',xq1,s,'-.',xq1,m,'--')
legend('Sample Points','pchip','spline','makima','Location','SouthEast')
```

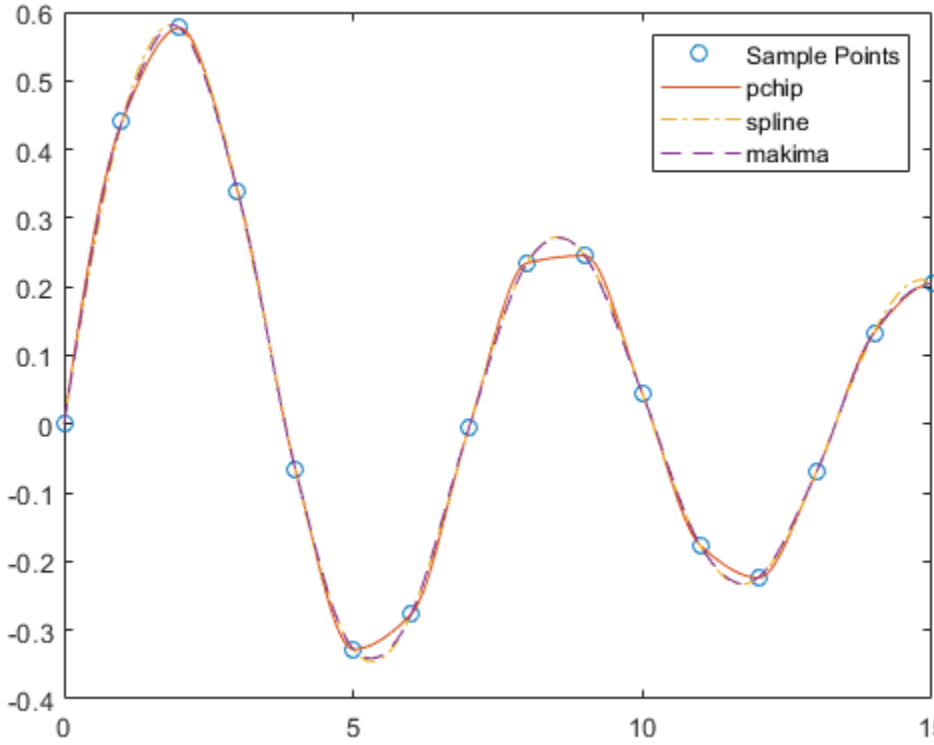


In this case, `pchip` and `makima` have similar behavior in that they avoid overshoots and can accurately connect the flat regions.

Perform a second comparison using an oscillatory sample function.

```
x = 0:15;
y = besselj(1,x);
xq2 = 0:0.01:15;
```

```
p = pchip(x,y,xq2);
s = spline(x,y,xq2);
m = makima(x,y,xq2);
plot(x,y,'o',xq2,p,'-',xq2,s,'-.',xq2,m,'--')
legend('Sample Points','pchip','spline','makima')
```



When the underlying function is oscillatory, `spline` and `makima` capture the movement between points better than `pchip`, which is aggressively flattened near local extrema.

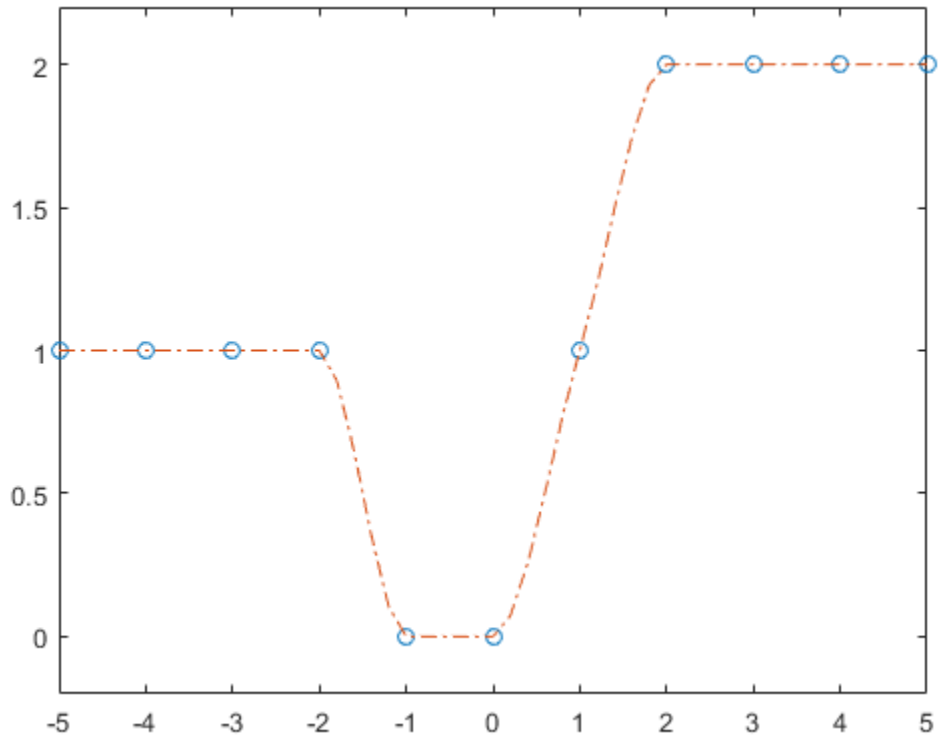
Interpolation with Piecewise Polynomial Structure

Create vectors for the `x` values and function values `y`, and then use `pchip` to construct a piecewise polynomial structure.

```
x = -5:5;
y = [1 1 1 1 0 0 1 2 2 2 2];
p = pchip(x,y);
```

Use the structure with `ppval` to evaluate the interpolation at several query points. Plot the results.

```
xq = -5:0.2:5;
pp = ppval(p,xq);
plot(x,y,'o',xq,pp,'-.')
ylim([-0.2 2.2])
```



Input Arguments

x — Sample points
vector

Sample points, specified as a vector. The vector x specifies the points at which the data y is given. The elements of x must be unique.

Data Types: `single` | `double`

y — Function values at sample points
vector | matrix | array

Function values at sample points, specified as a numeric vector, matrix, or array. `x` and `y` must have the same length.

If `y` is a matrix or array, then the values in the last dimension, `y(:,...,:,j)`, are taken as the values to match with `x`. In that case, the last dimension of `y` must be the same length as `x`.

Data Types: `single` | `double`

`xq` — Query points
scalar | vector | matrix | array

Query points, specified as a scalar, vector, matrix, or array. The points specified in `xq` are the *x*-coordinates for the interpolated function values `yq` computed by `pchip`.

Data Types: `single` | `double`

Output Arguments

`p` — Interpolated values at query points
scalar | vector | matrix | array

Interpolated values at query points, returned as a scalar, vector, matrix, or array. The size of `p` is related to the sizes of `y` and `xq`:

- If `y` is a vector, then `p` has the same size as `xq`.
- If `y` is an array of size `Ny = size(y)`, then these conditions apply:
  - If `xq` is a scalar or vector, then `size(p)` returns `[Ny(1:end-1) length(xq)]`.
  - If `xq` is an array, then `size(p)` returns `[Ny(1:end-1) size(xq)]`.

`pp` — Piecewise polynomial
structure

Piecewise polynomial, returned as a structure. Use this structure with the `ppval` function to evaluate the interpolating polynomials at one or more query points. The structure has these fields.

Field	Description
`form`	`'pp'` for *piecewise polynomial*

Field	Description
`breaks`	Vector of length `L+1` with strictly increasing elements that represent the start and end of each of `L` intervals
`coefs`	`L`-by-`k` matrix with each row `coefs(i,:)` containing the local coefficients of an order `k` polynomial on the `i`th interval, `[breaks(i),breaks(i+1)]`
`pieces`	Number of pieces, `L`
`order`	Order of the polynomials
`dim`	Dimensionality of target

Since the polynomial coefficients in `coefs` are local coefficients for each interval, you must subtract the lower endpoint of the corresponding knot interval to use the coefficients in a conventional polynomial equation. In other words, for the coefficients `[a,b,c,d]` on the interval `[x1,x2]`, the corresponding polynomial is

$$f(x) = a(x - x_1)^3 + b(x - x_1)^2 + c(x - x_1) + d \ .$$

More About

Shape-Preserving Piecewise Cubic Interpolation

`pchip` interpolates using a piecewise cubic polynomial $P(x)$ with these properties:

- On each subinterval $x_k \le x \le x_{k+1}$, the polynomial $P(x)$ is a cubic Hermite interpolating polynomial for the given data points with specified derivatives (slopes) at the interpolation points.
- $P(x)$ interpolates y, that is, $P(x_j) = y_j$, and the first derivative $\frac{dP}{dx}$ is continuous. The second derivative $\frac{d^2P}{dx^2}$ is probably not continuous so jumps at the x_j are possible.
- The cubic interpolant $P(x)$ is shape preserving. The slopes at the x_j are chosen in such a way that $P(x)$ preserves the shape of the data and respects monotonicity. Therefore,

on intervals where the data is monotonic, so is $P(x)$, and at points where the data has a local extremum, so does $P(x)$.

Note If y is a matrix, $P(x)$ satisfies these properties for each row of y.

Tips

- `spline` constructs $S(x)$ in almost the same way `pchip` constructs $P(x)$. However, `spline` chooses the slopes at the x_j differently, namely to make even $S''(x)$ continuous. This difference has several effects:
  - `spline` produces a smoother result, such that $S''(x)$ is continuous.
  - `spline` produces a more accurate result if the data consists of values of a smooth function.
  - `pchip` has no overshoots and less oscillation if the data is not smooth.
  - `pchip` is less expensive to set up.
  - The two are equally expensive to evaluate.

References

[1] Fritsch, F. N. and R. E. Carlson. "Monotone Piecewise Cubic Interpolation." *SIAM Journal on Numerical Analysis*. Vol. 17, 1980, pp.238–246.

[2] Kahaner, David, Cleve Moler, Stephen Nash. *Numerical Methods and Software*. Upper Saddle River, NJ: Prentice Hall, 1988.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Input `x` must be strictly increasing.
- Code generation does not remove `y` entries with `NaN` values.
- If you generate code for the `pp = pchip(x,y)` syntax, then you cannot input `pp` to the `ppval` function in MATLAB. To create a MATLAB `pp` structure from a `pp` structure created by the code generator:
  - In code generation, use `unmkpp` to return the piecewise polynomial details to MATLAB.
  - In MATLAB, use `mkpp` to create the `pp` structure.

See Also

`interp1` | `makima` | `ppval` | `spline`

Introduced before R2006a

pcode

Create content-obscured, executable files

Syntax

```
pcode(fun)
pcode(fun1,...,funN)
pcode(fun,'-inplace')
```

Description

`pcode(fun)` obfuscates the code in `fun.m` and produces a file called `fun.p`, known as a P-file. If `fun` is a folder, then all the script or function files in that folder are obfuscated in P-files. MATLAB creates the P-files in the current folder. The original `.m` file or folder can be anywhere on the search path.

`pcode(fun1,...,funN)` creates `N` P-files from the listed files. If any inputs are folders, then MATLAB creates a P-file for every `.m` file the folders contain.

`pcode(fun,'-inplace')` creates P-files in the same folder as the script or function files.

Note

- The `pcode` function *obfuscates* your code files, it does not *encrypt* them. While the content in a `.p` file is difficult to understand, it should not be considered secure. It is not recommended that you P-code files to protect your intellectual property.
- The `pcode` function does not support live scripts or live functions (`.mlx`).

Input Arguments

fun

MATLAB file or directory containing MATLAB files. If `fun` resides within a package and/or class folder, then `pcode` creates the same package and/or class structure to house the resulting P-files.

An input argument with no file extension and that is not a folder must be a function in the MATLAB path or in the current folder.

When using wild cards `*`, `pcode` ignores all files with extensions other than `.m`.

Default:

Examples

P-Coding Multiple Files

Convert selected files from the `sparfun` folder into P-files.

Create a temporary folder and define an existing path to `.m` files.

```
tmp = tempname;
mkdir(tmp)
cd(tmp)
fun = fullfile(matlabroot,'toolbox','matlab','sparfun','spr*.m');
```

Create the P-files.

```
pcode(fun)
dir(tmp)
```

```
.           ..          sprand.p     sprandn.p    sprandsym.p  sprank.p
```

The temporary folder now contains encoded P-files.

P-Coding Files That Belong to a Package and/or Class

Generate P-files from input files that are part of a package and/or class. This example uses an existing MATLAB example class.

Define `funclass` as an existing a class folder that contains `.m` files.

```
funclass = fullfile(docroot, 'techdoc', 'matlab_oop', ...
    'examples', '@BankAccount')
dir(funclass)
```

```
funclass =

C:\Program Files\MATLAB\R2013a\help\techdoc\matlab_oop\examples\@BankAccount


.            ..           BankAccount.m
```

Create a temporary folder. This folder has no package or class structure at this time.

```
tmp = tempname;
mkdir(tmp);
cd(tmp);
dir(tmp)
```

```
.             ..
```

Create a P-file for every `.m` file in the path `funclass`. Because the input files are part of a package and/or class, MATLAB creates a folder structure so that the output file belongs to the same package and/or class.

```
pcode(funclass)
dir(tmp)
```

```
.             ..            @BankAccount
```

You see that the P-file resides in the same folder structure.

```
dir('@BankAccount')
```

```
.              ..             BankAccount.p
```

P-Coding In Place

Generate P-files in the same folder as the input files using the option `inplace`.

Copy several MATLAB files to a temporary folder.

```
fun = fullfile(matlabroot,'toolbox','matlab','sparfun','spr*.m');
tmp = tempname;
mkdir(tmp);
copyfile(fun,tmp)
dir(tmp)
```

```
.          ..          sprand.m     sprandn.m    sprandsym.m  sprank.m
```

Create P-files in the same folder as the original `.m` files.

```
pcode(tmp,'-inplace')
dir(tmp)
```

```
.            sprand.m     sprandn.m    sprandsym.m  sprank.m
..           sprand.p     sprandn.p    sprandsym.p  sprank.p
```

Tips

- The `pcode` algorithm was redesigned in MATLAB 7.5 (Release R2007b). If your P-file was generated prior to MATLAB 7.5, it will not run in MATLAB 8.6 (Release R2015b) or later. Files generated in 7.5, or later versions, cannot run in MATLAB 7.4 or earlier.
- When obfuscating all files in a folder, `pcode` does not obfuscate any files within subfolders.
- A P-file takes precedence over the corresponding MATLAB code file (`.m`) for execution, even after modifications to the code file.
- MATLAB does not display any of the help comments that might be in the original MATLAB code file (`.m`).

See Also

`matlab.codetools.requiredFilesAndProducts`

Topics

"Protect Your Source Code"

Introduced before R2006a

pcolor

Pseudocolor plot

Syntax

```
pcolor(C)
pcolor(X,Y,C)
pcolor(ax, ___ )
s = pcolor( ___ )
```

Description

`pcolor(C)` creates a pseudocolor plot using the values in matrix `C`. A pseudocolor plot displays matrix data as an array of colored cells (known as faces). MATLAB creates this plot as a flat surface in the *x*-*y* plane. The surface is defined by a grid of *x*- and *y*-coordinates that correspond to the corners (or vertices) of the faces. The grid covers the region `X=1:n` and `Y=1:m`, where `[m,n] = size(C)`. Matrix `C` specifies the colors at the vertices. The color of each face depends on the color at one of its four surrounding vertices. Of the four vertices, the one that comes first in the *x*-*y* grid determines the color of the face.

`pcolor(X,Y,C)` specifies the *x*- and *y*-coordinates for the vertices. The size of `C` must match the size of the *x*-*y* coordinate grid. For example, if `X` and `Y` define an *m*-by-*n* grid, then `C` must be an *m*-by-*n* matrix.

`pcolor(ax, ___ )` specifies the target axes for the plot. Specify `ax` as the first argument in any of the previous syntaxes.

`s = pcolor( ___ )` returns a `Surface` object. Use `s` to set properties on the plot after creating it. For a list of properties, see Surface Properties.

Examples

Plot Four Faces with Four Colors

Create coordinate vectors `X` and `Y` and a colormap called `mymap` containing five colors: red, green, blue, yellow, and black.

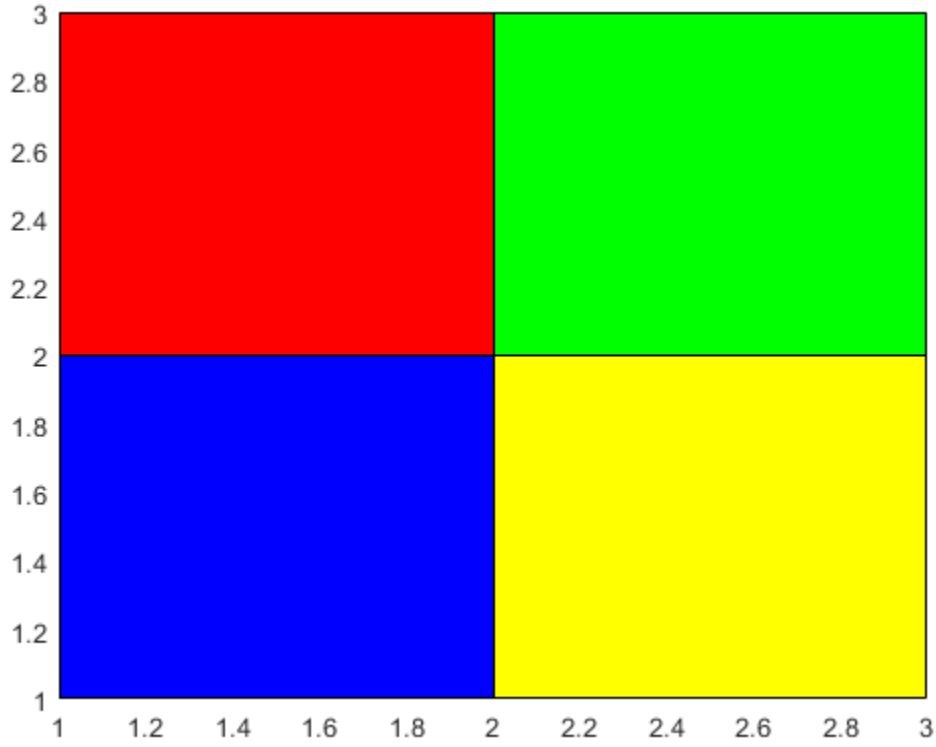
```
X = [1 2 3; 1 2 3; 1 2 3];
Y = X';
mymap = [1 0 0; 0 1 0; 0 0 1; 1 1 0; 0 0 0];
```

Create matrix `C` that maps the colormap colors to the nine vertices. Four of the nine vertices determine the colors of the faces. Specify the colors at those vertices to make the faces red (1), green (2), blue (3), and yellow (4), respectively. Set the colors at the other vertices to black (5).

```
C = [3 4 5; 1 2 5; 5 5 5];
```

Plot the faces, and call the `colormap` function to replace the default colormap with `mymap`.

```
pcolor(X,Y,C)
colormap(mymap)
```



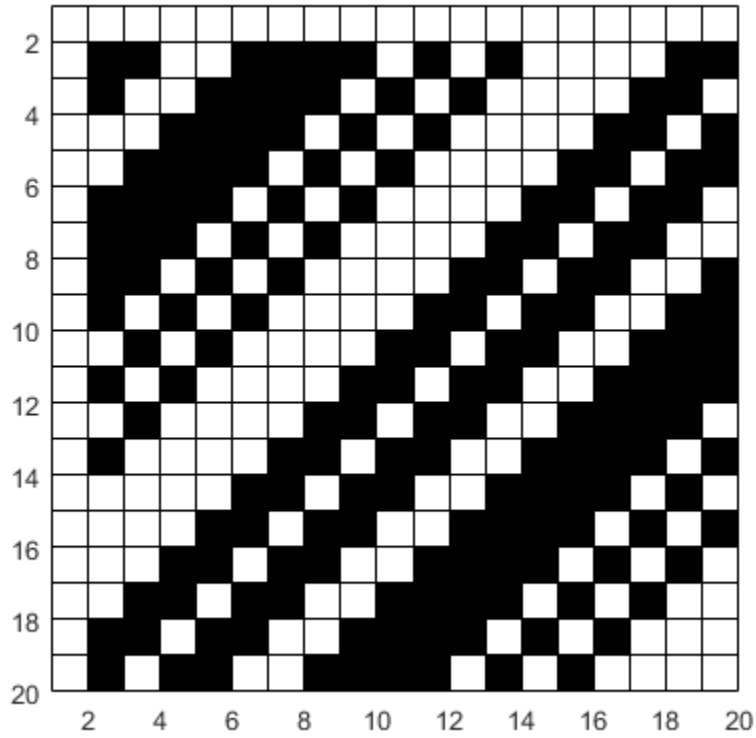
Plot Hadamard Matrix

A Hadamard matrix has elements that are either `1` or `-1`. A good way to visualize this matrix is with a two-color colormap.

Create a 20-by-20 Hadamard matrix. Then plot the matrix using a black and white colormap. Use the `axis` function to reverse the direction of the *y*-axis and set the axis lines to equal lengths.

```
C = hadamard(20);
pcolor(C)
```

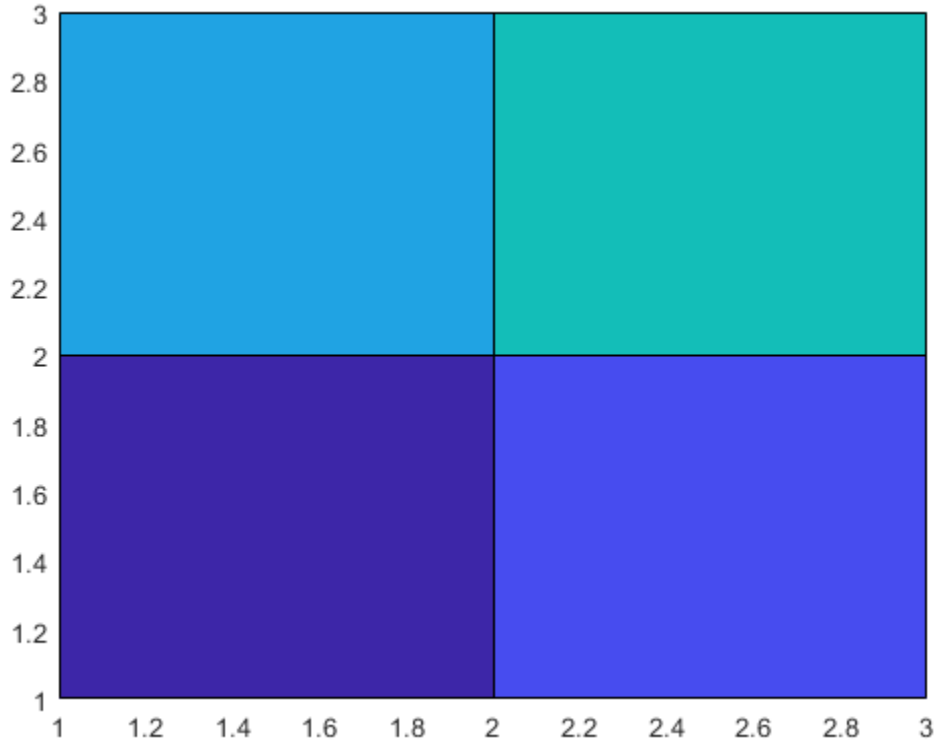
```
colormap(gray(2))
axis ij
axis square
```



Modify Borders

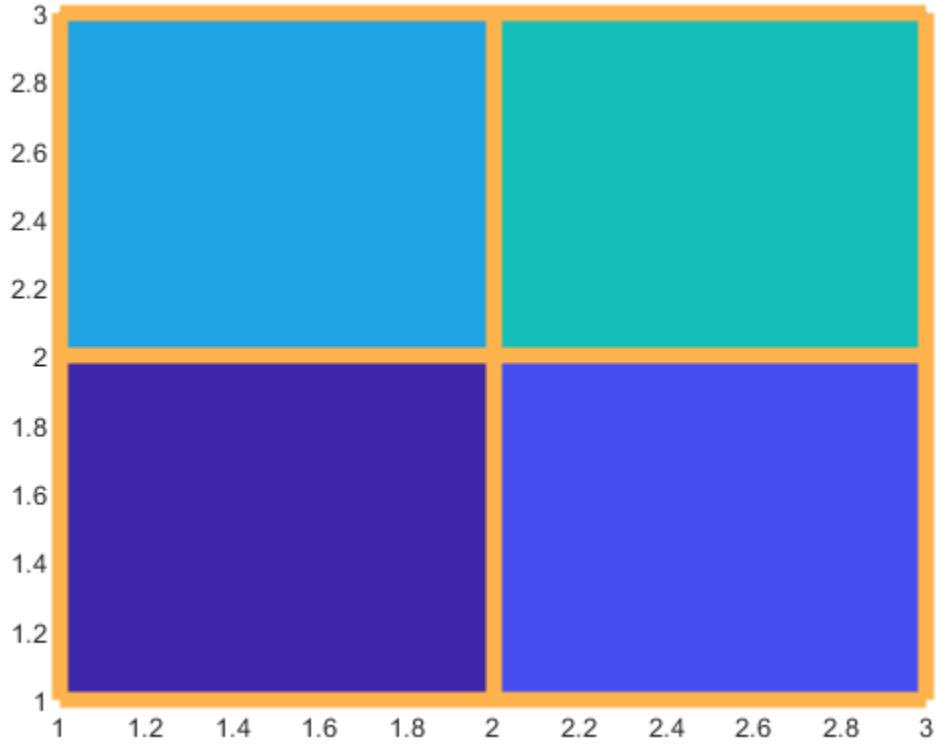
Create color matrix `C`. Then create a pseudocolor plot of `C`, and store the `Surface` object in the return argument `s`.

```
C = [1 2 3; 4 5 6; 7 8 9];
s = pcolor(C);
```



Change the border color by setting the `EdgeColor` property of `s`. Make the border thicker by setting the `LineWidth` property.

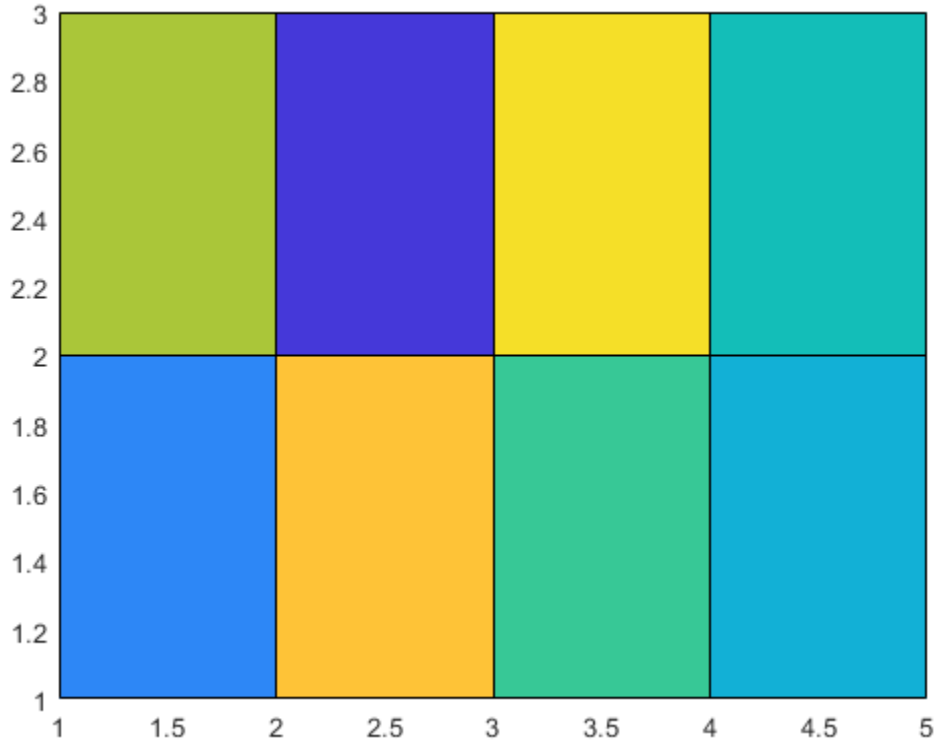
```
s.EdgeColor = [1 0.7 0.3];
s.LineWidth = 6;
```



Interpolate Colors Across Faces

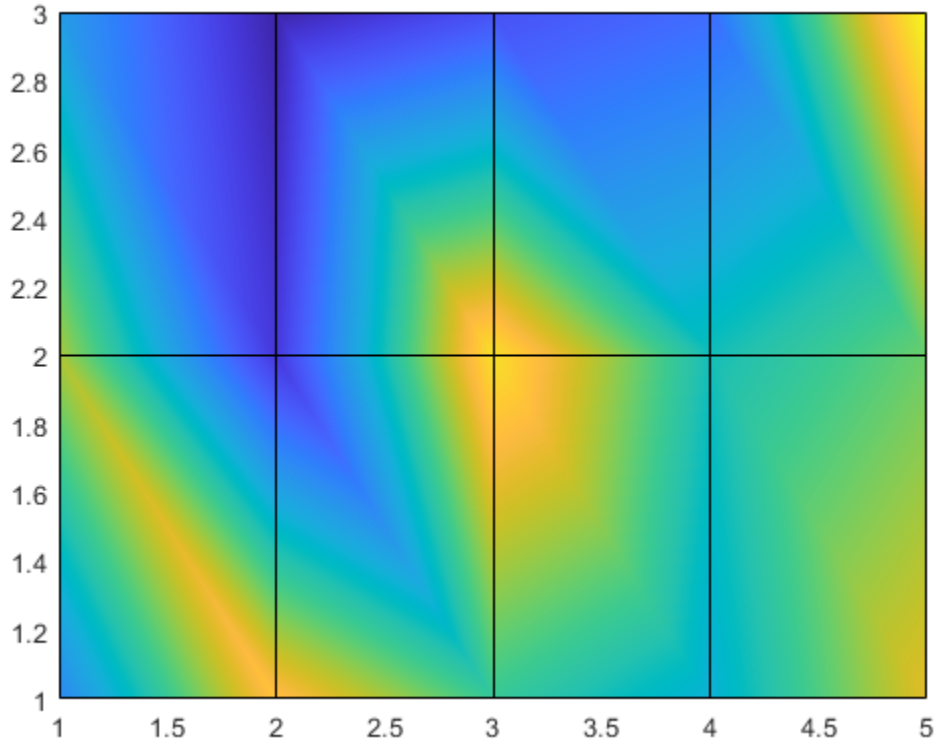
Create color matrix `C`. Then create a pseudocolor plot of `C`, and store the `Surface` object in the return argument `s`.

```
C = [5 13 9 7 12; 11 2 14 8 10; 6 1 3 4 15];
s = pcolor(C);
```



To interpolate the colors across the faces, set the `FaceColor` propery of `s` to `'interp'`.

```
s.FaceColor = 'interp';
```



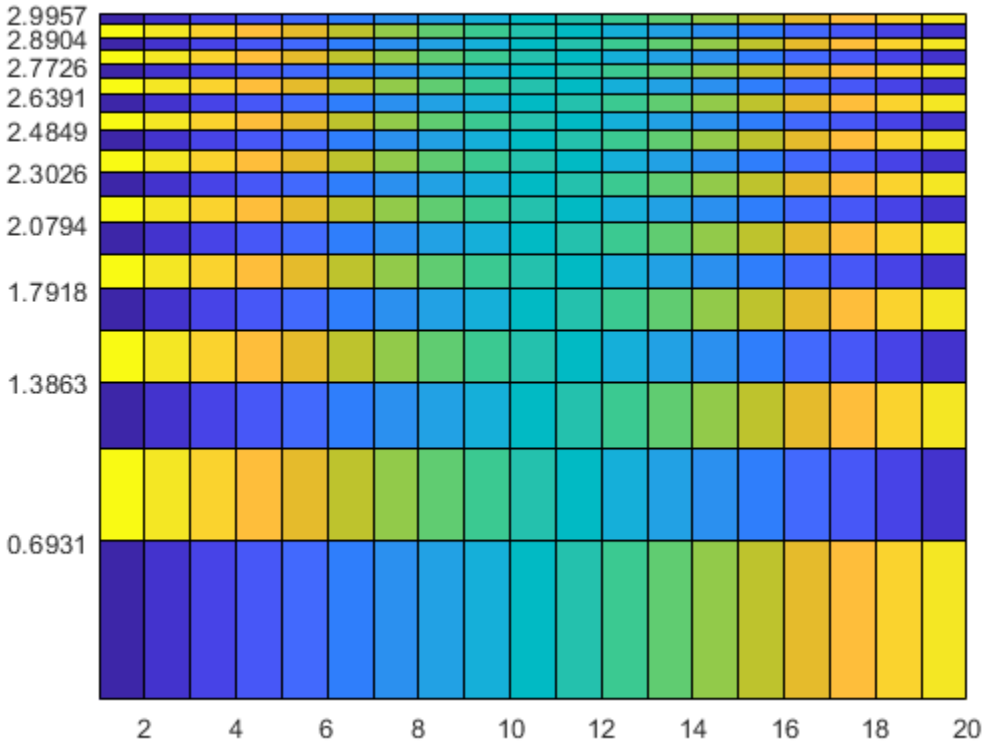
Specify Semilogarithmic Grid

Create matrices X and Y, which define a regularly spaced grid of vertices. Calculate matrix LY as the log of Y. Then create matrix C containing alternating pairs of rows of color indices.

```
[X,Y] = meshgrid(1:20);
LY = log(Y);
colorscale = [1:20; 20:-1:1];
C = repmat(colorscale,10,1);
```

Plot X and LY, using the colors specified in C. Then adjust the tick labels on the *y*-axis.

```
s = pcolor(X,LY,C);
tickvals = LY(2:2:20,1)';
set(gca,'YTick',tickvals);
```



Specify Parametric Grid

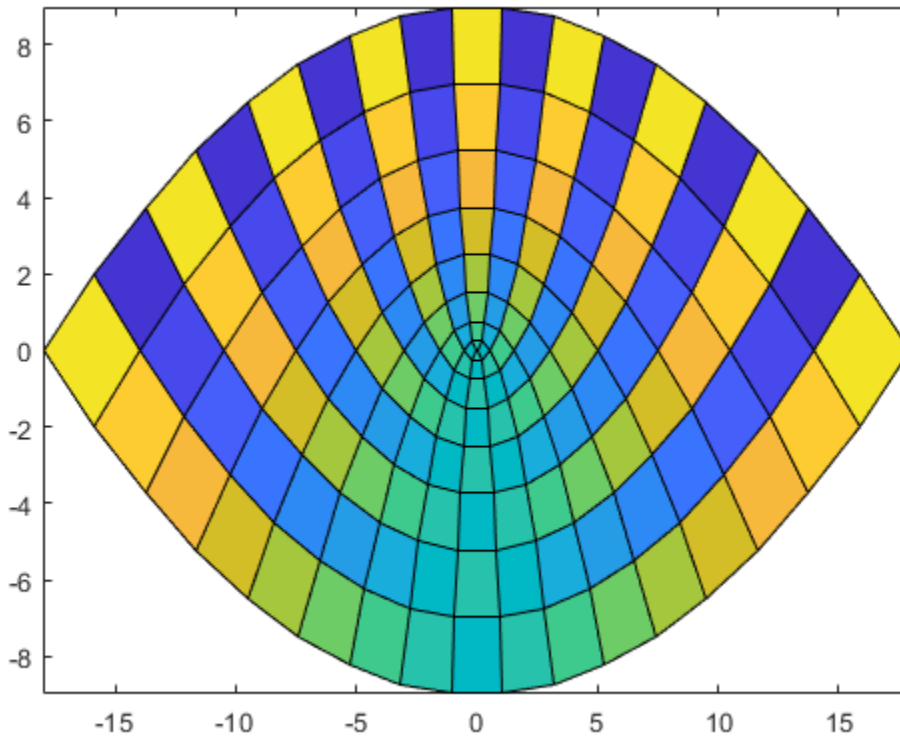
Create matrices X and Y, which define a regularly spaced grid of vertices. Calculate matrices XX and YY as functions of X and Y. Then create matrix C containing alternating pairs of rows of color indices.

```
[X,Y] = meshgrid(-3:6/17:3);
XX = 2*X.*Y;
```

```
YY = X.^2 - Y.^2;
colorscale = [1:18; 18:-1:1];
C = repmat(colorscale,9,1);
```

Plot XX and YY using the colors in C.

```
pcolor(XX,YY,C);
```

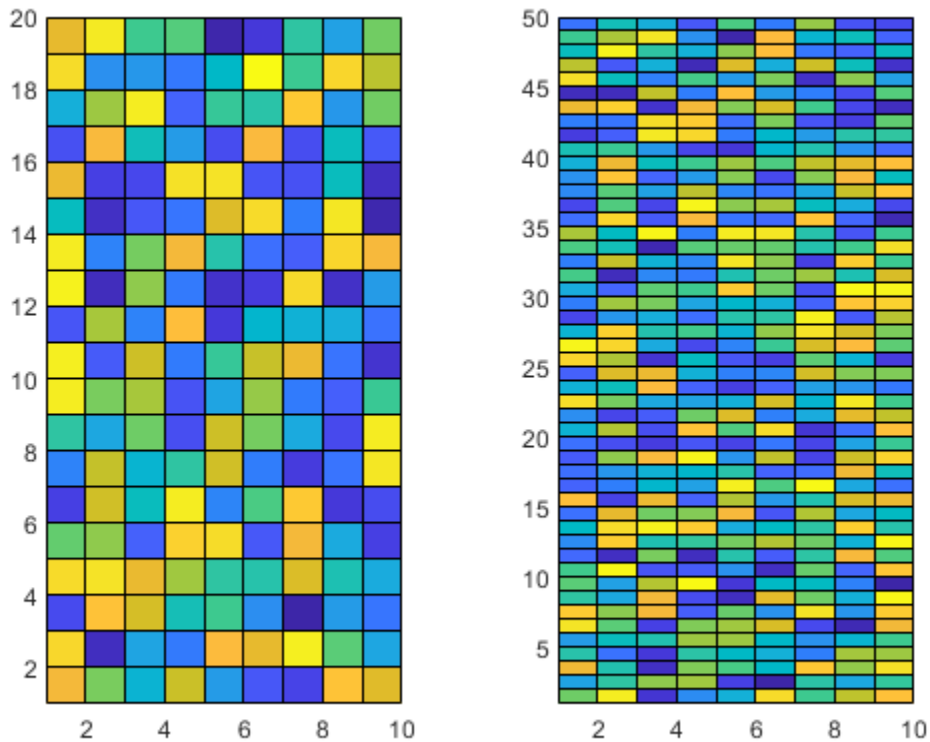


Specify Target Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 1-by-2 tiled chart layout.

Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Create two pseudocolor plots by specifying the axes as the first argument to `pcolor`.

```
tiledlayout(1,2)

% Left plot
ax1 = nexttile;
C1 = rand(20,10);
pcolor(ax1,C1)

% Right plot
ax2 = nexttile;
C2 = rand(50,10);
pcolor(ax2,C2)
```

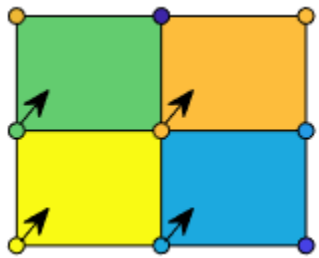


Input Arguments

C — Color matrix

matrix

Color matrix containing indices into the colormap. The values in `C` map colors in the colormap array to the vertices surrounding each face. The color of a face depends on the color at one of its four vertices. Of the four vertices, the one that come first in `X` and `Y` determines the color of the face. If you do not specify `X` and `Y`, MATLAB uses `X=1:n` and `Y=1:m`, where `[m,n] = size(C)`. Because of this relationship between the vertex colors and face colors, none of the values in the last row and column of `C` are represented in the plot.



Note The first vertex of a face is the one that is closest to the upper-left corner of the corresponding matrix. However, because the *y*-axis increases from bottom to top, the first vertex shown in the plot is typically the one in the lower-left corner of the face. To get the effect you want, you might have to change the orientation of the *y*-axis or the orientation of matrix `C`.

The values in `C` scale to the full range of the colormap. The smallest value in `C` maps to the first row in the colormap array. The largest value in `C` maps to the last row in the colormap array. The intermediate values in `C` map linearly to the intermediate rows of the colormap array. You can adjust this mapping using the `caxis` function.

The `CData` property of the `Surface` object stores the values of `C`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

X — *x*-coordinates
matrix | vector

x-coordinates, specified as a matrix the same size as `C`, or as a vector of length `n`, where `[m,n] = size(C)`. The default value of `X` is the vector `(1:n)`.

To create a rectangular grid of vertices, specify `X` as either of the following:

- A vector containing values that are increasing or decreasing.
- A matrix that is increasing or decreasing along one dimension and is constant along the other dimension. Set the dimension that varies to the opposite of the dimension that varies in matrix `Y`. You can use the `meshgrid` function to create the `X` and `Y` matrices.

To create a parametric grid, create a rectangular grid and pass it through a mathematical function.

Example: `X = 1:10`

Example: `X = [1 2 3; 1 2 3; 1 2 3]`

Example: `[X,Y] = meshgrid(1:10)`

The `XData` property of the `Surface` object stores the *x*-coordinates.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

Y — *y*-coordinates
matrix | vector

y-coordinates, specified as a matrix the same size as `C`, or as a vector of length `m`, where `[m,n] = size(C)`. The default value of `Y` is the vector `(1:m)`.

To create a rectangular grid of vertices, specify `Y` as either of the following:

- A vector containing values that are increasing or decreasing.
- A matrix that is increasing or decreasing along one dimension and is constant along the other dimension. Set the dimension that varies to the opposite of the dimension that varies in matrix `X`. You can use the `meshgrid` function to create the `X` and `Y` matrices.

To create a parametric grid, create a rectangular grid and pass it through a mathematical function.

Example: `Y = 1:10`

Example: `Y = [1 1 1; 2 2 2; 3 3 3]`

Example: `[X,Y] = meshgrid(1:10)`

The `YData` property of the `Surface` object stores the *y*-coordinates.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

ax — Target axes
`Axes` object

Target axes, specified as an `Axes` object. If you do not specify the axes, then `pcolor` plots into the current axes.

Algorithms

Use the `pcolor`, `image`, and `imagesc` functions to display rectangular arrays of colored cells. The relationship between the color matrix `C` and the colored cells is different in each case.

- `pcolor(C)` uses the values in `C` to define the vertex colors by scaling the values to the full range of the colormap. The size of `C` determines the number of vertices. The values in `C` map colors from the current colormap to the vertices surrounding each cell.
- `image(C)` uses `C` to define the cell colors by mapping the values directly into the colormap. The size of `C` determines the number of cells.
- `imagesc(C)` uses `C` to define the cell colors by scaling the values to the full range of the colormap. The size of `C` determines the number of cells.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions

`colormap` | `image` | `imagesc` | `surface`

Properties

Surface Properties

Topics

“Change Color Scheme Using a Colormap”
“How Surface Plot Data Relates to a Colormap”
“Control Colormap Limits”

Introduced before R2006a

pdepe

Solve 1-D parabolic and elliptic PDEs

Syntax

```
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options)
[sol,tsol,sole,te,ie] = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,
options)
```

Description

`sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)` solves a system of parabolic and elliptic PDEs with one spatial variable *x* and time *t*. At least one equation must be parabolic. The scalar `m` represents the symmetry of the problem (slab, cylindrical, or spherical). The equations being solved are coded in `pdefun`, the initial value is coded in `icfun`, and the boundary conditions are coded in `bcfun`. The ordinary differential equations (ODEs) resulting from discretization in space are integrated to obtain approximate solutions at the times specified in `tspan`. The `pdepe` function returns values of the solution on a mesh provided in `xmesh`.

`sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options)` also uses the integration settings defined by `options`, which is created using the `odeset` function. For example, use the `AbsTol` and `RelTol` options to specify absolute and relative error tolerances.

`[sol,tsol,sole,te,ie] = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan, options)` also finds where functions of $(t,u(x,t))$, called event functions, are zero. In the output, `te` is the time of the event, `sole` is the solution at the time of the event, and `ie` is the index of the triggered event. `tsol` is a column vector of times specified in `tspan`, prior to the first terminal event.

For each event function, specify whether the integration is to terminate at a zero and whether the direction of the zero-crossing matters. Do this by setting the `'Events'` option of `odeset` to a function, such as `@myEventFcn`, and creating a corresponding

function: [value,isterminal,direction] = myEventFcn(m,t,xmesh,umesh). The xmesh input contains the spatial mesh and umesh is the solution at the mesh points.

Examples

Solve Heat Equation in Cylindrical Coordinates

Solve the heat equation in cylindrical coordinates using pdepe, and plot the solution.

In cylindrical coordinates with angular symmetry the heat equation is

$$\frac{\partial u}{\partial t} = \frac{1}{x}\frac{\partial}{\partial x}\left(x\frac{\partial u}{\partial x}\right).$$

The equation is defined for $0 \le x \le 1$ at times $t \ge 0$. The initial condition is defined in terms of the bessel function $J_0(x)$ and its first zero $n = 2.404825557695773$ as

$$u(x,0) = J_0(\mathrm{nx}).$$

Since this problem is in cylindrical coordinates (m = 1), pdepe automatically enforces the symmetry condition at $x = 0$. The right boundary condition is

$$u(1,t) = J_0(n)e^{-n^2 t}.$$

The initial and boundary conditions are selected to be consistent with the analytic solution to the problem,

$$u(x,t) = J_0(\mathrm{nx})e^{-n^2 t}.$$

To solve this equation in MATLAB, you need to code the equation, initial conditions, and boundary conditions, then select a suitable solution mesh before calling the solver pdepe. You either can include the required functions as local functions at the end of a file (as done here), or save them as separate, named files in a directory on the MATLAB path.

Code Equation

Before you can code the equation, you need to rewrite it in a form that the pdepe solver expects. The standard form that pdepe expects is

$$c\left(x,t,u,\frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left(x^m f\left(x,t,u,\frac{\partial u}{\partial x}\right)\right) + s\left(x,t,u,\frac{\partial u}{\partial x}\right).$$

Written in this form, the PDE becomes

$$\frac{\partial u}{\partial t} = x^{-1}\frac{\partial}{\partial x}\left(x^1\frac{\partial u}{\partial x}\right)$$

With the equation in the proper form you can read off the relevant terms:

- $m = 1$
- $c\left(x,t,u,\frac{\partial u}{\partial x}\right) = 1$
- $f\left(x,t,u,\frac{\partial u}{\partial x}\right) = \frac{\partial u}{\partial x}$
- $s\left(x,t,u,\frac{\partial u}{\partial x}\right) = 0$

Now you can create a function to code the equation. The function should have the signature `[c,f,s] = heatcyl(x,t,u,dudx)`:

- `x` is the independent spatial variable.
- `t` is the independent time variable.
- `u` is the dependent variable being differentiated with respect to `x` and `t`.
- `dudx` is the partial spatial derivative $\partial u/\partial x$.
- The outputs `c`, `f`, and `s` correspond to coefficients in the standard PDE equation form expected by `pdepe`. These coefficients are coded in terms of the input variables `x`, `t`, `u`, and `dudx`.

As a result, the equation in this example can be represented by the function:

```
function [c,f,s] = heatcyl(x,t,u,dudx)
c = 1;
f = dudx;
s = 0;
end
```

(Note: All functions are included as local functions at the end of the example.)

Code Initial Conditions

Next, write a function that returns the initial condition. The initial condition is applied at the first time value `tspan(1)`. The function should have the signature `u0 = heatic(x)`.

The corresponding function for $u(x, 0) = J_0(\mathrm{nx})$ is

```
function u0 = heatic(x)
n = 2.404825557695773;
u0 = besselj(0,n*x);
end
```

Code Boundary Conditions

Now, write a function that evaluates the boundary condition

$$u(1, t) = J_0(n)e^{-n^2 t}.$$

Since this problem is in cylindrical coordinates (`m = 1`), `pdepe` automatically enforces the symmetry condition at $x = 0$, so you do not need to specify a left boundary condition.

For problems posed on the interval $a \le x \le b$, the boundary conditions apply for all t and either $x = a$ or $x = b$. The standard form for the boundary conditions expected by the solver is

$$p(x, t, u) + q(x, t)f\left(x, t, u, \frac{\partial u}{\partial x}\right) = 0.$$

Written in this form, the boundary conditions for the partial derivatives of u need to be expressed in terms of the flux $f\left(x, t, u, \frac{\partial u}{\partial x}\right)$. So the right boundary condition for this problem is

$$u(1, t) - J_0(n)e^{-n^2 t} + 0 \cdot f\left(x, t, u, \frac{\partial u}{\partial x}\right) = 0.$$

The boundary function should have the function signature `[pl,ql,pr,qr] = heatbc(xl,ul,xr,ur,t)`:

- The inputs `xl` and `ul` correspond to u and x for the left boundary.
- The inputs `xr` and `ur` correspond to u and x for the right boundary.
- `t` is the independent time variable.

- The outputs `pl` and `ql` correspond to $p_L(x,t,u)$ and $q_L(x,t)$ for the left boundary ($x = 0$ for this problem).
- The outputs `pr` and `qr` correspond to $p_R(x,t,u)$ and $q_R(x,t)$ for the right boundary ($x = 1$ for this problem).

The boundary conditions in this example are represented by the function:

```
function [pl,ql,pr,qr] = heatbc(xl,ul,xr,ur,t)
n = 2.404825557695773;
pl = 0; %ignored by solver since m=1
ql = 0; %ignored by solver since m=1
pr = ur-besselj(0,n)*exp(-n^2*t);
qr = 0;
end
```

Select Solution Mesh

Before solving the equation you need to specify the mesh points (t,x) at which you want `pdepe` to evaluate the solution. Specify the points as vectors `t` and `x`. The vectors `t` and `x` play different roles in the solver. In particular, the cost and accuracy of the solution depend strongly on the length of the vector `x`. However, the computation is much less sensitive to the values in the vector `t`.

For this problem, use a mesh with 25 equally spaced points in the spatial interval [0,1] for both `x` and `t`.

```
x = linspace(0,1,25);
t = linspace(0,1,25);
```

Solve Equation

Finally, solve the equation using the symmetry `m`, the PDE equation, the initial conditions, the boundary conditions, and the meshes for `x` and `t`.

```
m = 1;
sol = pdepe(m,@heatcyl,@heatic,@heatbc,x,t);
```

`pdepe` returns the solution in a 3-D array `sol`, where `sol(i,j,k)` approximates the `k`th component of the solution u_k evaluated at `t(i)` and `x(j)`. The size of `sol` is `length(t)`-by-`length(x)`-by-`length(u0)`, since `u0` specifies an initial condition for each solution component. For this problem, `u` has only one component, so `sol` is a 25-by-25 matrix, but in general you can extract the `k`th solution component with the command `u = sol(:,:,k)`.

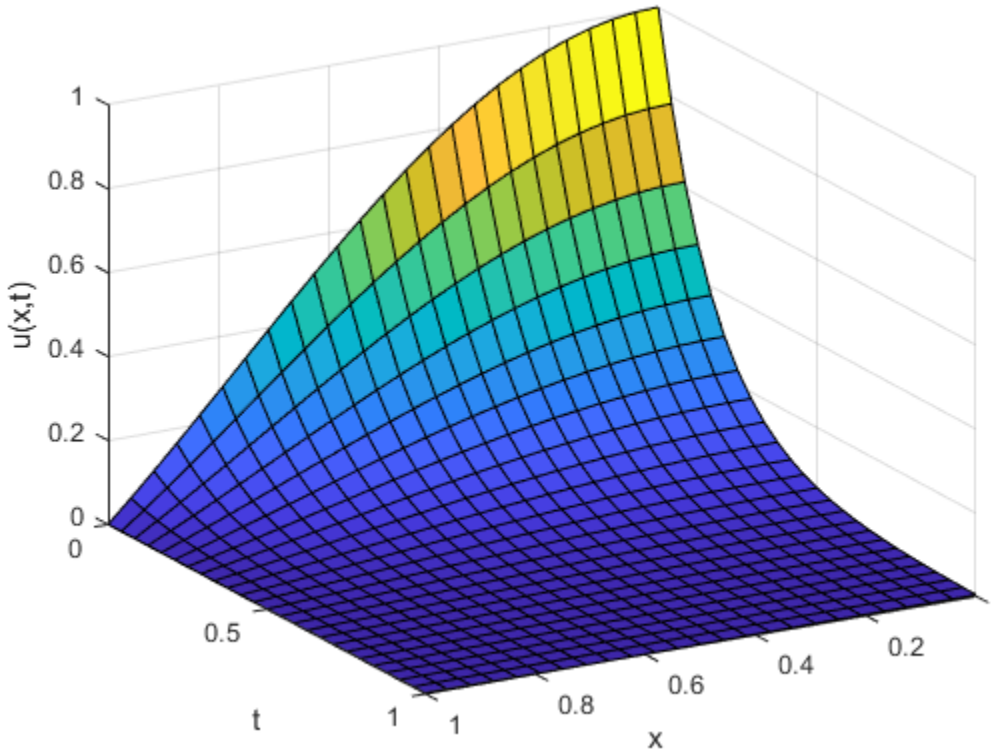
Extract the first solution component from `sol`.

```
u = sol(:,:,1);
```

Plot Solution

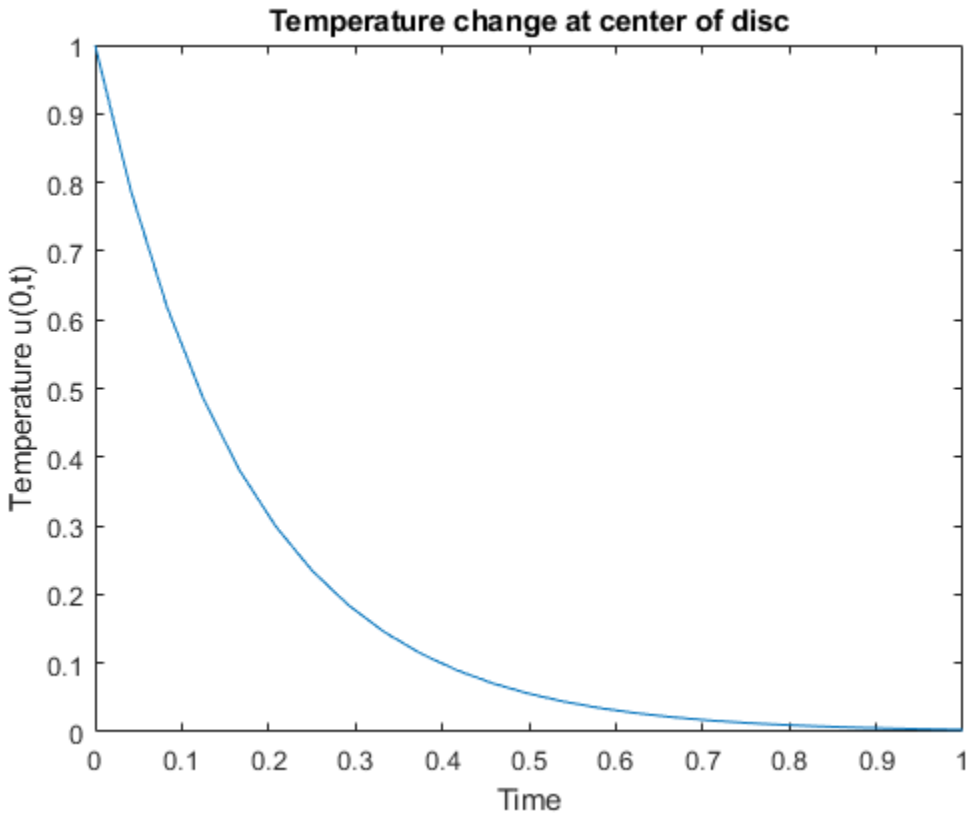
Create a surface plot of the solution. Since the problem is posed with cylindrical coordinates on a disc, the `x` values show the temperature on the disc some distance from the center, and the `t` values show how the temperature at a particular location changes over time.

```
surf(x,t,u)
xlabel('x')
ylabel('t')
zlabel('u(x,t)')
view([150 25])
```



Plot the temperature change at the center ($x = 0$) of the disc.

```
plot(t,sol(:,1))
xlabel('Time')
ylabel('Temperature u(0,t)')
title('Temperature change at center of disc')
```



Local Functions

Listed here are the local helper functions that the PDE solver `pdepe` calls to calculate the solution. Alternatively, you can save these functions as their own files in a directory on the MATLAB path.

```
function [c,f,s] = heatcyl(x,t,u,dudx)
c = 1;
f = dudx;
s = 0;
end
%---------------------------------------------
function u0 = heatic(x)
n = 2.404825557695773;
```

```
u0 = besselj(0,n*x);
end
%--------------------------------------------
function [pl,ql,pr,qr] = heatbc(xl,ul,xr,ur,t)
n = 2.404825557695773;
pl = 0; %ignored by solver since m=1
ql = 0; %ignored by solver since m=1
pr = ur-besselj(0,n)*exp(-n^2*t);
qr = 0;
end
%--------------------------------------------
```

Solve Oscillatory PDE with Event Logging

Solve a partial differential equation and use an event function to log zero-crossings in the oscillatory solution.

Consider the equation

$$\frac{1}{x}\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(\frac{1}{t}u\right).$$

The equation is defined for $0 \le x \le 1$ and $t \ge 0.1$. The initial condition is

$$u(x, 0.1) = 1.$$

The boundary conditions are

$$u(0, t) = 1,$$

$$u(1, t) = \cos(\pi t).$$

Additionally, the zero-crossings of the solution are of interest.

To solve this equation in MATLAB, you need to code the equation, initial conditions, boundary conditions, and event function, then select a suitable solution mesh before calling the solver `pdepe`. You either can include the required functions as local functions at the end of a file (as done here), or save them as separate, named files in a directory on the MATLAB path.

Code Equation

Before you can code the equation, you need to rewrite it in a form that the `pdepe` solver expects. The standard form that `pdepe` expects is

$$c\left(x,t,u,\frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left(x^m f\left(x,t,u,\frac{\partial u}{\partial x}\right)\right) + s\left(x,t,u,\frac{\partial u}{\partial x}\right).$$

The PDE equation is already in this form:

$$\frac{1}{x}\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(\frac{1}{t}u\right).$$

So you can read off the relevant terms:

- $m = 0$
- $c\left(x,t,u,\frac{\partial u}{\partial x}\right) = \frac{1}{x}$
- $f\left(x,t,u,\frac{\partial u}{\partial x}\right) = \frac{1}{t}u$
- $s\left(x,t,u,\frac{\partial u}{\partial x}\right) = 0$

Now you can create a function to code the equation. The function should have the signature `[c,f,s] = oscpde(x,t,u,dudx)`:

- `x` is the independent spatial variable.
- `t` is the independent time variable.
- `u` is the dependent variable being differentiated with respect to `x` and `t`.
- `dudx` is the partial spatial derivative $\partial u/\partial x$.
- The outputs `c`, `f`, and `s` correspond to coefficients in the standard PDE equation form expected by `pdepe`. These coefficients are coded in terms of the input variables `x`, `t`, `u`, and `dudx`.

As a result, the equation in this example can be represented by the function:

```
function [c,f,s] = oscpde(x,t,u,dudx)
c = 1/x;
f = u/t;
s = 0;
end
```

(Note: All functions are included as local functions at the end of the example.)

Code Initial Conditions

Next, write a function that returns the initial condition. The initial condition is applied at the first time value `tspan(1)`. The function should have the signature `u0 = oscic(x)`.

The corresponding function for $u(x, 0.1) = 1$ is

```
function u0 = oscic(x)
u0 = 1;
end
```

Code Boundary Conditions

Now, write a function that evaluates the boundary conditions

$$u(0, t) = 1,$$

$$u(1, t) = \cos(\pi t).$$

For problems posed on the interval $a \le x \le b$, the boundary conditions apply for all t and either $x = a$ or $x = b$. The standard form for the boundary conditions expected by the solver is

$$p\left(x, t, u, \frac{\partial u}{\partial x}\right) + q(x, t) f\left(x, t, u, \frac{\partial u}{\partial x}\right) = 0.$$

Written in this form, the boundary conditions for this problem become

$$u(0, t) - 1 + 0 \cdot f\left(x, t, u, \frac{\partial u}{\partial x}\right) = 0,$$

$$u(1, t) - \cos(\pi x) + 0 \cdot f\left(x, t, u, \frac{\partial u}{\partial x}\right) = 0.$$

The boundary function should have the function signature `[pl,ql,pr,qr] = oscbc(xl,ul,xr,ur,t)`:

- The inputs `xl` and `ul` correspond to u and x for the left boundary.
- The inputs `xr` and `ur` correspond to u and x for the right boundary.
- `t` is the independent time variable.

- The outputs `pl` and `ql` correspond to $p_L(x,t,u)$ and $q_L(x,t)$ for the left boundary ($x = 0$ for this problem).
- The outputs `pr` and `qr` correspond to $p_R(x,t,u)$ and $q_R(x,t)$ for the right boundary ($x = 1$ for this problem).

The boundary conditions in this example are represented by the function:

```
function [pl,ql,pr,qr] = oscbc(xl,ul,xr,ur,t)
pl = ul - 1;
ql = 0;
pr = ur - cos(pi*t);
qr = 0;
end
```

Code Event Function

Use an event function to log the zero-crossings of the solution in the integration. The event function has a function signature of `[value,isterminal,direction] = pdevents(m,t,xmesh,umesh)`:

- `m` is the coordinate symmetry specified as the first input to `pdepe`.
- `t` is the current time (a scalar).
- `xmesh` is the spatial mesh.
- `umesh` contains the solution at the mesh points.
- `value` is an equation of interest, usually expressed in terms of the solution `umesh`. When `value` equals 0, an event occurs.
- `isterminal` specifies whether an event stops the integration. If `isterminal` is 0, events are logged but the integration does not stop. If `isterminal` is 1, then the integration stops when an event occurs.
- `direction` specifies the direction of zero crossing. If 1, only zero-crossings with a positive slope trigger an event. If -1, the zero-crossings must have a negative slope. If 0, then any zero-crossing triggers an event.

At each time in the integration, the solver calls the event function to check for zero crossings. To log all zero crossings, `value` should look for changes in sign in the solution vector `umesh`. Specify `isterminal` and `direction` as vectors of zeros with the same size, since the events in this example are not terminal and the zero-crossings can occur with any slope.

The event function for this problem is

```
function [value,isterminal,direction] = pdevents(m,t,xmesh,umesh)
value = umesh;
isterminal = zeros(size(umesh));
direction = zeros(size(umesh));
end
```

Select Solution Mesh

Before solving the equation you need to specify the mesh points (t, x) at which you want `pdepe` to evaluate the solution. For this problem, use a fine mesh of 50 points in the intervals $0 \le x \le 1$ and $0.1 \le t \le 1$. A fine mesh gives good resolution of the oscillatory solution.

```
x = linspace(0,1,50);
t = linspace(0.1,pi,50);
```

Solve Equation

Finally, solve the equation using the symmetry `m`, the PDE equation, the initial conditions, the boundary conditions, the event function, and the meshes for `x` and `t`. Use `odeset` to create an options structure that references the events function, and pass in the structure as the last input argument to `pdepe`. Specify five output arguments to return information from both the event function and the solver:

- `sol` is the solution computed by `pdepe`.
- `tsol` is a vector of times before a terminal event. `tsol` is equal to `t` when no events are terminal.
- `sole` is the solution at the time of each event.
- `te` is the time of each event.
- `ie` is the index of each event. Since `values = umesh` in the event function, `ie` gives the indices of `umesh` that triggered an event at each time step.

```
m = 0;
options = odeset('Events',@pdevents);
[sol,tsol,sole,te,ie] = pdepe(m,@oscpde,@oscic,@oscbc,x,t,options);
```

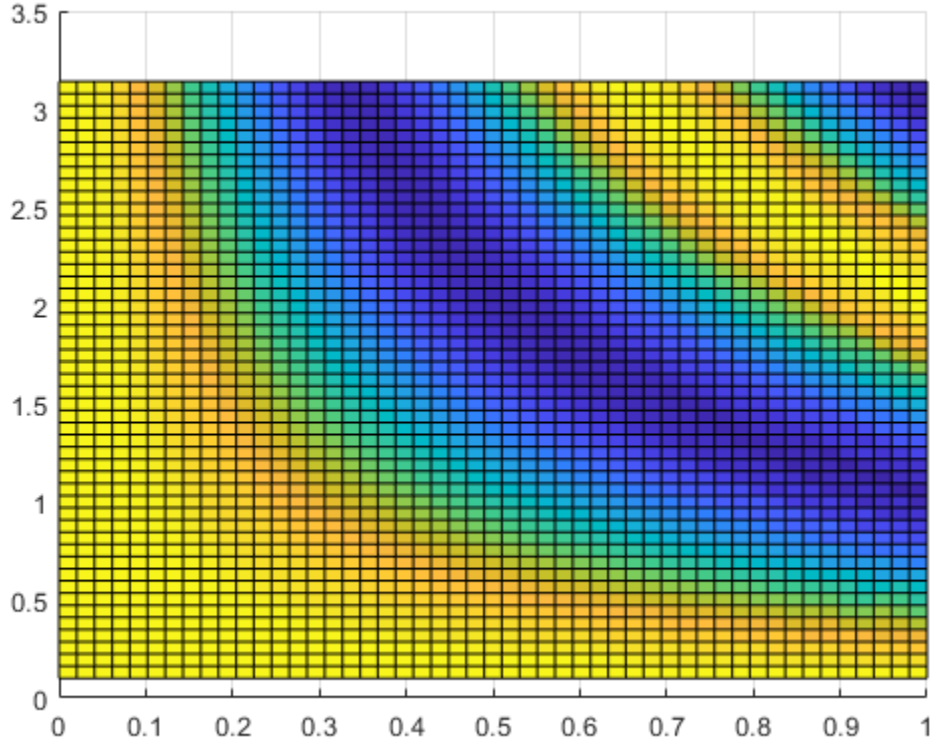
Extract the solution as a matrix `u`.

```
u = sol(:,:,1);
```

Plot Solution

Create a surface plot of the solution and view the plot from above.

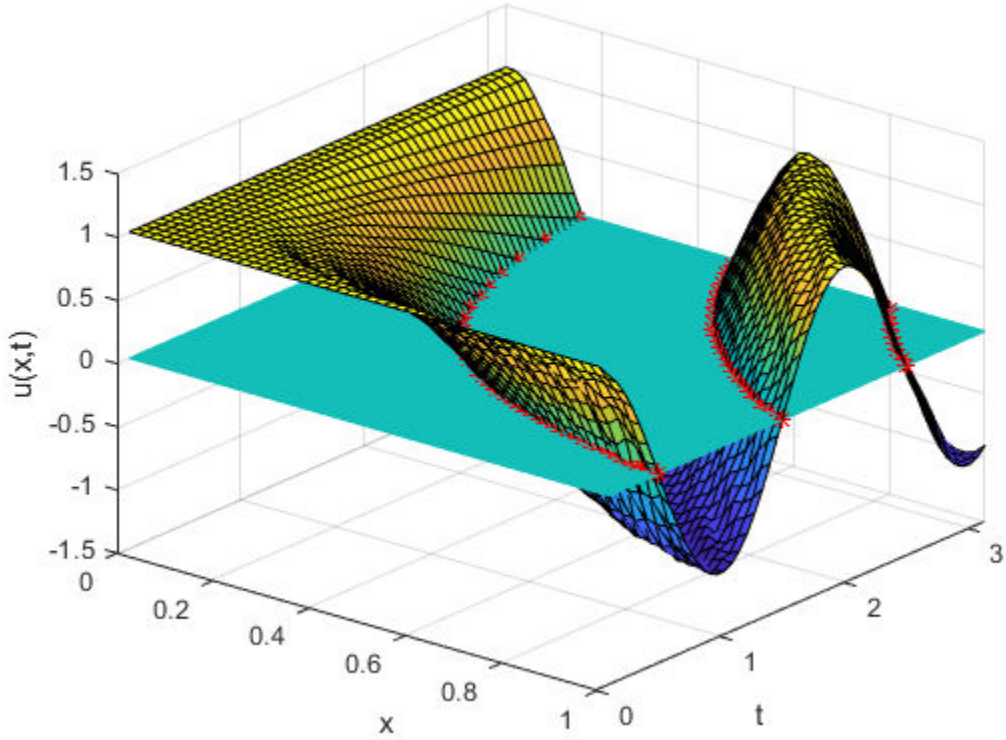
```
surf(x,t,u)
view(2)
```



Plot the points where events occurred, with a surface $f(x,t) = 0$ for reference. The output index vector `ie` is useful to pick out the event locations. The expression `x(ie)'` gives the x-values where events occurred, and the expression `sole(x==x(ie)')` gives the corresponding solution values.

```
view([39 30])
xlabel('x')
ylabel('t')
zlabel('u(x,t)')
hold on
plot3(x(ie)',te,sole(x==x(ie)'),'r*')
```

```
surf(x,t,zeros(size(u)),'EdgeColor','flat')
hold off
```



Local Functions

Listed here are the local helper functions that the PDE solver pdepe calls to calculate the solution. Alternatively, you can save these functions as their own files in a directory on the MATLAB path.

```
function [c,f,s] = oscpde(x,t,u,dudx)
c = 1/x;
f = u/t;
s = 0;
end
```

```
%--------------------------------------------
function u0 = oscic(x)
u0 = 1;
end
%--------------------------------------------
function [pl,ql,pr,qr] = oscbc(xl,ul,xr,ur,t)
pl = ul - 1;
ql = 0;
pr = ur - cos(pi*t);
qr = 0;
end
%--------------------------------------------
function [value, isterminal, direction] = pdevents(m,t,xmesh,umesh)
value = umesh;
isterminal = zeros(size(umesh));
direction = zeros(size(umesh));
end
%--------------------------------------------
```

Input Arguments

m — Symmetry constant

0 | 1 | 2

Symmetry constant, specified as one of the values in this table. `m` specifies the type of problem in the equations specified by `pdefun`. Once you rewrite the equations in the form expected by `pdepe`, you can read the value of `m` from the equation.

Value	Description	Laplacian of flux
0	1-D Cartesian coordinates with no symmetry	$\Delta f = \frac{\partial^2 f}{\partial x^2}$
1	2-D Cylindrical coordinates with azimuthal angular symmetry	$\Delta f = \frac{1}{\rho}\frac{\partial}{\partial \rho}\left(\rho\frac{\partial f}{\partial \rho}\right) + \frac{1}{\rho^2}\frac{\partial^2 f}{\partial \varphi^2}$, where $\frac{\partial f}{\partial \varphi} = 0$ (azimuthal symmetry).

Value	Description	Laplacian of flux
2	3-D Spherical coordinates with azimuthal and zenith angular symmetries	$\Delta f = \frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial f}{\partial r}\right)$, $+ \frac{1}{r^2\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial f}{\partial\theta}\right)$ $+ \frac{1}{r^2\sin^2\theta}\frac{\partial^2 f}{\partial\varphi^2}$ where $\frac{\partial f}{\partial\theta} = \frac{\partial f}{\partial\varphi} = 0$ (azimuthal and zenith angular symmetries).

When `m > 0`, `pdepe` requires the left spatial boundary a to be ≥ 0, and it imposes the left boundary condition automatically to account for the coordinate discontinuity at the origin. In that case, `pdepe` ignores any specified left boundary conditions in `bcfun`.

`pdefun` — PDE function for equations
function handle

PDE function for equations, specified as a function handle that defines the coefficients of the PDE.

`pdefun` encodes the coefficients for PDEs of the form

$$c\left(x,t,u,\frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left(x^m f\left(x,t,u,\frac{\partial u}{\partial x}\right)\right) + s\left(x,t,u,\frac{\partial u}{\partial x}\right).$$

The terms in the equation are:

- $f\left(x,t,u,\frac{\partial u}{\partial x}\right)$ is a flux term.
- $s\left(x,t,u,\frac{\partial u}{\partial x}\right)$ is a source term.
- The coupling of the partial derivatives with respect to time is restricted to multiplication by a diagonal matrix $c\left(x,t,u,\frac{\partial u}{\partial x}\right)$. The diagonal elements of this matrix are either zero or positive. An element that is zero corresponds to an elliptic equation, and all other elements correspond to parabolic equations. There must be at least one

parabolic equation. An element of c that corresponds to a parabolic equation can vanish at isolated values of x if those values of x are mesh points.

- Discontinuities in c and s due to material interfaces are permitted provided that a mesh point is placed at each interface.

The PDEs hold for $t_0 \le t \le t_f$ and $a \le x \le b$. The values `tspan(1)` and `tspan(end)` correspond to t_0 and t_f, while `xmesh(1)` and `xmesh(end)` correspond to a and b. The interval $[a,b]$ must be finite. `m` can be 0, 1, or 2, corresponding to slab, cylindrical, or spherical symmetry, respectively. If $m > 0$, then a must be ≥ 0.

Coding the Equation

Write a function `pdefun` that calculates the values of the coefficients c, f, and s. Use the function signature

```
[c,f,s] = pdefun(x,t,u,dudx)
```

The input arguments to `pdefun` are scalars `x` and `t` and vectors `u` and `dudx`. The vector `u` approximates the solution u, and `dudx` approximates its partial derivative with respect to x. The output arguments `c`, `f`, and `s` are column vectors with a number of elements equal to the number of equations. `c` stores the diagonal elements of the matrix c.

Example

The heat equation $\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$ can be rewritten in the form expected by the solver as

$$1 \cdot \frac{\partial u}{\partial t} = x^0 \frac{\partial}{\partial x}\left(x^0 \frac{\partial u}{\partial x}\right).$$

In this form you can read off the value of the coefficients as $m = 0$, $c = 1$, $f = \partial u/\partial x$, and $s = 0$. The function for this equation is:

```
function [c,f,s] = heatpde(x,t,u,dudx)
  c = 1;
  f = dudx;
  s = 0;
end
```

Data Types: `function_handle`

`icfun` — Initial condition function
function handle

Initial condition function, specified as a function handle that defines the initial condition.

For $t = t_0$ and all x, the solution components satisfy initial conditions of the form

$$u(x, t_0) = u_0(x).$$

Coding the Initial Condition

Write a function `icfun` that defines the initial conditions. Use the function signature:

```
function u0 = icfun(x)
```

When called with an argument `x`, the function `icfun` evaluates and returns the initial values of the solution components at `x` in the column vector `u0`. The number of elements in `u0` is equal to the number of equations.

Example

The constant initial condition $u(x, 0) = 1/2$ corresponds to this function:

```
function u0 = heatic(x)
  u0 = 0.5;
end
```

Data Types: `function_handle`

`bcfun` — Boundary condition function

function handle

Boundary condition function, specified as a function handle that defines the boundary conditions.

For all t and either $x = a$ or $x = b$, the solution components satisfy a boundary condition of the form

$$p(x, t, u) + q(x, t) f\left(x, t, u, \frac{\partial u}{\partial x}\right) = 0.$$

Elements of q are either zero or never zero. Note that the boundary conditions are expressed in terms of the flux f rather than $\partial u/\partial x$. Also, of the two coefficients, only p can depend on u.

Coding the Boundary Conditions

Write a function `bcfun` that defines the terms p and q of the boundary conditions. Use the function signature

```
function [pl,ql,pr,qr] = bcfun(xl,ul,xr,ur,t)
```

- `ul` is the approximate solution at the left boundary `xl = a`, and `ur` is the approximate solution at the right boundary `xr = b`.
- `pl` and `ql` are column vectors corresponding to p and q evaluated at `xl`. Similarly, `pr` and `qr` correspond to `xr`.
- The number of elements in the outputs `pl`, `ql`, `pr`, and `qr` is equal to the number of equations.

When $m > 0$ and $a = 0$, boundedness of the solution near $x = 0$ requires that the flux f vanish at $a = 0$. `pdepe` imposes this boundary condition automatically, and it ignores values returned in `pl` and `ql`.

Example

Consider the boundary conditions $u(0,t) = 0, \quad u(1,t) = 1$ on the interval $0 \le x \le 1$. In the form expected by the solver, the boundary conditions are

$$u(0,t) + 0 \cdot f\left(0, t, u, \frac{\partial u}{\partial x}\right) = 0,$$

$$u(1,t) - 1 + 0 \cdot f\left(0, t, u, \frac{\partial u}{\partial x}\right) = 0.$$

In this form it is clear that both q terms are zero. The p terms are nonzero to reflect the values of u. The corresponding function is

```
function [pl,ql,pr,qr] = heatbc(xl,ul,xr,ur,t)
  pl = ul;
  ql = 0;
  pr = ur - 1;
  qr = 0;
end
```

Data Types: `function_handle`

`xmesh` — Spatial mesh
vector

Spatial mesh, specified as a vector `[x0 x1 ... xn]` specifying the points at which a numerical solution is requested for every value in `tspan`. Second-order approximations to the solution are made on the mesh specified in `xmesh`. `pdepe` does *not* select the mesh in x automatically. You must provide an appropriate fixed mesh in `xmesh`.

The elements of `xmesh` must satisfy `x0 < x1 < ... < xn`. Generally, it is best to use closely spaced mesh points where the solution changes rapidly. The length of `xmesh` must be >=3, and the computational cost of the solution depends strongly on the length of `xmesh`.

For problems with discontinuities, you should place mesh points at the discontinuities so that the problem is smooth within each subinterval. However, when $m > 0$, it is not necessary to use a fine mesh near $x = 0$ to account for the coordinate singularity.

Example: `xmesh = linspace(0,5,25)` uses a spatial mesh of 25 points between 0 and 5.

Data Types: `single` | `double`

`tspan` — Time span of integration
vector

Time span of integration, specified as a vector `[t0 t1 ... tf]` specifying the points at which a solution is requested for every value in `xmesh`. The `pdepe` function performs the time integration with an ODE solver that selects both the time step and the formula dynamically. The elements of `tspan` merely specify where you want answers, and the computational cost of the solution therefore depends only weakly on the length of `tspan`.

The elements of `tspan` must satisfy `t0 < t1 < ... < tf`. The length of `tspan` must be >=3.

Example: `tspan = linspace(0,5,5)` uses five time points between 0 and 5.

Data Types: `single` | `double`

`options` — Option structure
structure array

Option structure, specified as a structure array. Use the `odeset` function to create or modify the option structure. `pdepe` supports these options:

- `'RelTol'`
- `'AbsTol'`

- `'NormControl'`
- `'InitialStep'`
- `'MaxStep'`
- `'Events'`

In most cases, default values for these options provide satisfactory solutions.

Example: `options = odeset('RelTol',1e-5)` specifies a relative error tolerance of `1e-5`.

Data Types: `struct`

Output Arguments

`sol` — Solution array
3-D array

Solution array, returned as a 3-D array.

`pdepe` returns the solution as a multidimensional array. u_i = `ui = sol(:,:,i)` is an approximation to the `i`th component of the solution vector u. The element `ui(j,k) = sol(j,k,i)` approximates u_i at (t,x) = `(tspan(j),xmesh(k))`.

`ui = sol(j,:,i)` approximates component `i` of the solution at time `tspan(j)` and mesh points `xmesh(:)`. Use `pdeval` to compute the approximation and its partial derivative $\partial u_i/\partial x$ at points not included in `xmesh`. See `pdeval` for details.

`tsol` — Solution times
column vector

Solution times, returned as a column vector of times specified in `tspan` that are prior to the first terminal event.

`sole` — Solution at time of events
array

Solution at time of events, returned as an array. The event times in `te` correspond to solutions returned in `sole`, and `ie` specifies which event occurred.

`te` — Time of events
column vector

Time of events, returned as a column vector. The event times in `te` correspond to solutions returned in `sole`, and `ie` specifies which event occurred.

`ie` — Index of triggered event function
column vector

Index of triggered event function, returned as a column vector. The event times in `te` correspond to solutions returned in `sole`, and `ie` specifies which event occurred.

Tips

- If `uji = sol(j,:,i)` approximates component `i` of the solution at time `tspan(j)` and mesh points `xmesh`, then `pdeval` evaluates the approximation and its partial derivative $\partial u_i/\partial x$ at the array of points `xout` and returns them in `uout` and `duoutdx`: `[uout,duoutdx] = pdeval(m,xmesh,uji,xout)`. The `pdeval` function evaluates the partial derivative $\partial u_i/\partial x$ rather than the flux. The flux is continuous, but at a material interface the partial derivative may have a jump.

Algorithms

The time integration is done with the `ode15s` solver. `pdepe` exploits the capabilities of `ode15s` for solving the differential-algebraic equations that arise when the PDE contains elliptic equations, and for handling Jacobians with a specified sparsity pattern.

After discretization, elliptic equations give rise to algebraic equations. If the elements of the initial-conditions vector that correspond to elliptic equations are not consistent with the discretization, `pdepe` tries to adjust them before beginning the time integration. For this reason, the solution returned for the initial time may have a discretization error comparable to that at any other time. If the mesh is sufficiently fine, `pdepe` can find consistent initial conditions close to the given ones. If `pdepe` displays a message that it has difficulty finding consistent initial conditions, try refining the mesh. No adjustment is necessary for elements of the initial conditions vector that correspond to parabolic equations.

References

[1] Skeel, R. D. and M. Berzins, "A Method for the Spatial Discretization of Parabolic Equations in One Space Variable," *SIAM Journal on Scientific and Statistical Computing*, Vol. 11, 1990, pp.1–32.

See Also

`ode15s` | `odeget` | `odeset` | `pdeval`

Topics

“Solving Partial Differential Equations”
“Solve Single PDE”
“Solve System of PDEs”

Introduced before R2006a

pdeval

Interpolate numerical solution of PDE

Syntax

```
[u,dudx] = pdeval(m,xmesh,usol,xq)
```

Description

`[u,dudx] = pdeval(m,xmesh,usol,xq)` interpolates a numerical solution returned by `pdepe` at new query points `xq`, and returns the interpolated values of the solution `u` and their partial derivative `dudx`. The `m`, `xmesh`, and `usol` arguments are reused from a previous call to `pdepe`:

- The numerical solution produced by `sol = pdepe(m,@pdefun,@pdeic,@pdebc,xmesh,tspan)` uses the coordinate symmetry `m` and spatial mesh `xmesh` to return a 3-D matrix of the solution values `sol`. Reuse the `m` and `xmesh` inputs used to calculate the solution when you call `pdeval`.
- The input vector `usol = sol(i,:,k)` is the value of component `k` of the solution at time `tspan(i)`. When there is only one solution component, `usol` is a row extracted from the solution matrix `usol = sol(i,:)`.

Examples

Interpolate PDE Solution

Use `pdepe` to solve a partial differential equation, and then use `pdeval` to evaluate the solution at additional points.

Solve PDE

Use `pdepe` to solve the `pdex1` example problem. You can type `edit pdex1` to see more details on the problem, or see `pdepe` for details on the PDE solution process. The

required functions to solve the `pdex1` problem are included at the end of this example as local functions.

```
m = 0;
x = linspace(0,1,20);
t = linspace(0,2,5);
sol = pdepe(m,@pdex1pde,@pdex1ic,@pdex1bc,x,t);
```

Interpolate Solution

The solution `sol` generated by `pdepe` uses 20 points for `x`, evenly spaced between 0 and 1. Create a vector of query points that are located midway between the points used by `pdepe`.

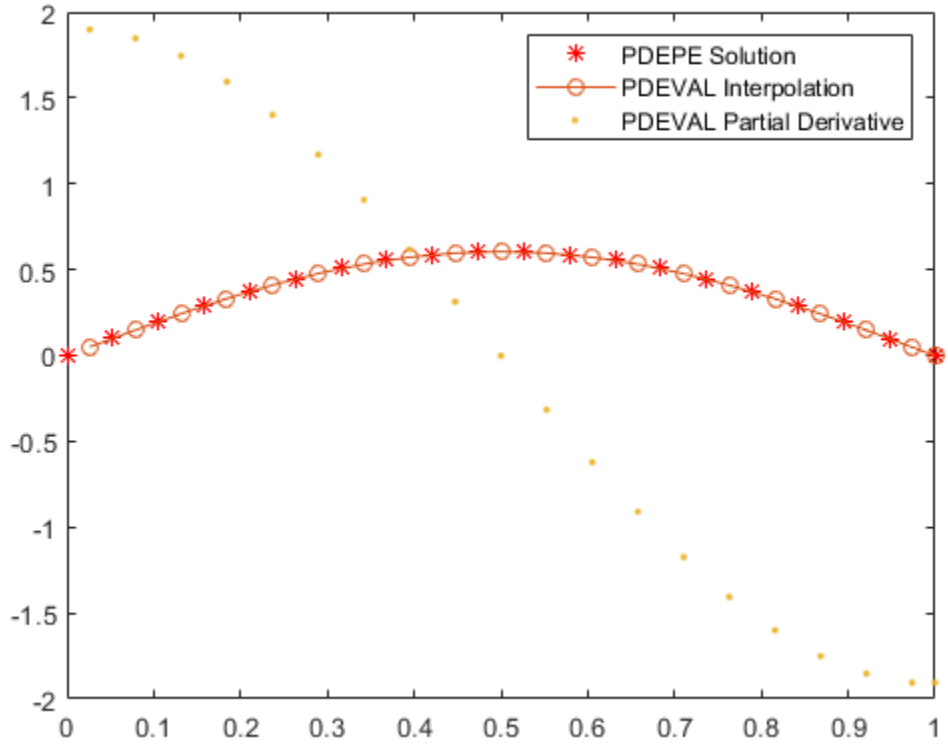
```
xq = x;
xq(1:end-1) = xq(1:end-1) + diff(xq)./2;
```

Use `pdeval` to interpolate the solution at the query points. Since there is only one solution component, you can extract a row from `sol` to operate on, such as `sol(2,:)`.

```
[u,dudx] = pdeval(m,x,sol(2,:),xq);
```

Plot the solution computed by `pdepe`, as well as the interpolated solution and its partial derivative computed by `pdeval`.

```
plot(x,sol(2,:),'r*')
hold on
plot(xq,u,'-o')
plot(xq,dudx,'.')
hold off
legend('PDEPE Solution', 'PDEVAL Interpolation', 'PDEVAL Partial Derivative')
```



Local Functions

Listed here are the local helper functions that the PDE solver `pdepe` calls to calculate the solution.

```
function [c,f,s] = pdex1pde(x,t,u,DuDx)
c = pi^2;
f = DuDx;
s = 0;
end
% --------------------------------------------------------
function u0 = pdex1ic(x)
u0 = sin(pi*x);
end
```

```
% --------------------------------------------------------
function [pl,ql,pr,qr] = pdex1bc(xl,ul,xr,ur,t)
pl = ul;
ql = 0;
pr = pi * exp(-t);
qr = 1;
end
```

Input Arguments

m — Coordinate symmetry used with pdepe

0 | 1 | 2

Coordinate symmetry used with pdepe, specified as one of the values in this table. Specify the same coordinate symmetry you used in the initial call to pdepe.

Value	Symmetry
0	Slab/Cartesian
1	Cylindrical
2	Spherical

xmesh — Spatial mesh used with pdepe

vector

Spatial mesh used with pdepe, specified as a vector [x0 x1 ... xn] containing the points at which a numerical solution was computed. Specify the same spatial mesh you used in the initial call to pdepe.

Data Types: single | double

usol — Extracted solution component

vector

Extracted solution component, specified as a vector of values computed by pdepe for one solution component at a particular time.

pdepe returns the solution in a 3-D array sol, where sol(i,j,k) approximates the kth component of the solution u_k evaluated at time t(i) and spatial point xmesh(j). You can create the input usol with the command usol = sol(i,:,k), where sol(i,:,k) is the value of component k of the solution at time tspan(i), evaluated on the entire

spatial mesh `xmesh`. When there is only one solution component, `usol` is a row extracted from the solution matrix `usol = sol(i,:)`.

Example: `usol = sol(10,:,2)` extracts the second solution component calculated at time `tspan(10)`.

Example: `usol = sol(5,:)` extracts the solution calculated at time `tspan(5)`.

Data Types: `single` | `double`
Complex Number Support: Yes

xq — Query points
scalar | vector

Query points, specified as a scalar or vector of *x*-coordinates. The points specified in `xq` must lie within the interval `[min(xmesh) max(xmesh)]`.

Typically, the points in `xq` are specified between the values in `xmesh` to enable `pdeval` to evaluate the solution produced by `pdepe` on a finer mesh.

Example: `xq = linspace(0,1,100)` specifies 100 evenly spaced query points between 0 and 1.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

u — Interpolated solution
vector

Interpolated solution, returned as a vector with the same size as `xq`.

dudx — Partial derivative of interpolated solution
vector

Partial derivative of interpolated solution, returned as a vector with the same size as `xq`.

Tips

- `pdeval` evaluates the partial derivative $\frac{\partial u}{\partial x}$ rather than the flux $f\left(x, t, u, \frac{\partial u}{\partial x}\right)$. Although the flux is continuous, the partial derivative can have a jump at a material interface.

See Also

pdepe

Topics

"Solving Partial Differential Equations"

Introduced before R2006a

peaks

Example function of two variables

Syntax

```
Z = peaks;
Z = peaks(n);
Z = peaks(V);
Z = peaks(X,Y);
peaks(...)
[X,Y,Z] = peaks(...);
```

Description

`peaks` is a function of two variables, obtained by translating and scaling Gaussian distributions, which is useful for demonstrating `mesh`, `surf`, `pcolor`, `contour`, and so on.

`Z = peaks;` returns a 49-by-49 matrix.

`Z = peaks(n);` returns an `n`-by-`n` matrix.

`Z = peaks(V);` returns an `n`-by-`n` matrix, where `n = length(V)`.

`Z = peaks(X,Y);` evaluates `peaks` at the given `X` and `Y` (which must be the same size) and returns a matrix the same size.

`peaks(...)` (with no output argument) plots the peaks function with `surf`. Use any of the input argument combinations in the previous syntaxes.

`[X,Y,Z] = peaks(...);` returns two additional matrices, `X` and `Y`, for parametric plots, for example, `surf(X,Y,Z,del2(Z))`. If not given as input, the underlying matrices `X` and `Y` are

```
[X,Y] = meshgrid(V,V)
```

where V is a given vector, or V is a vector of length n with elements equally spaced from -3 to 3. If no input argument is given, the default n is 49.

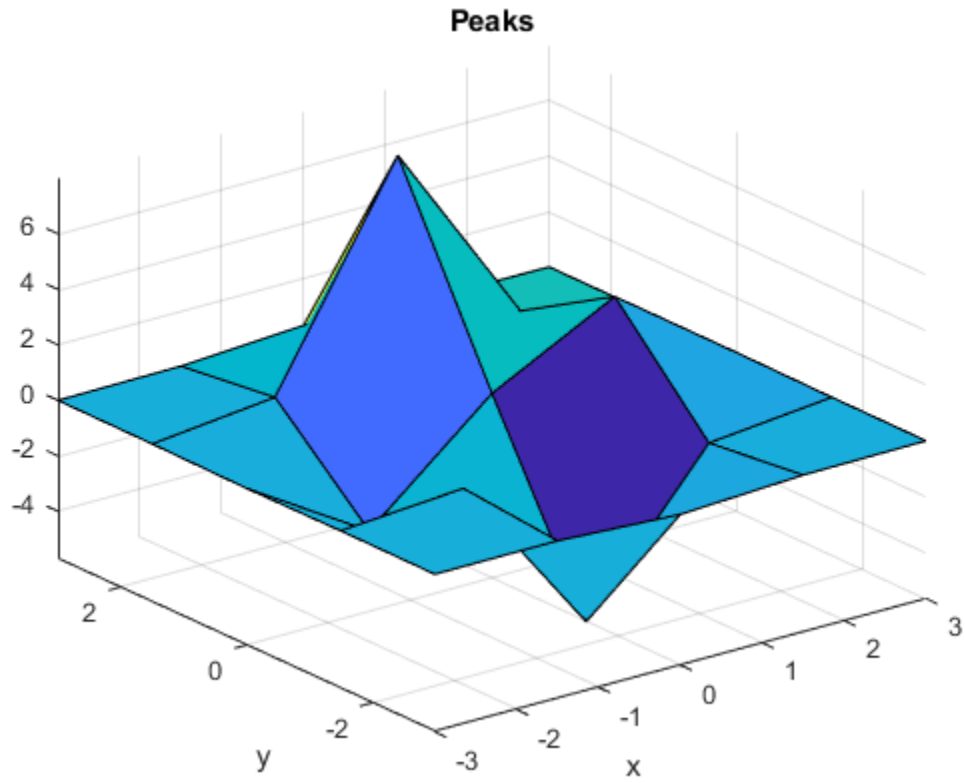
Examples

Peaks Surface

Create a 5-by-5 matrix of peaks and display the surface.

```
figure
peaks(5);
```

```
z =  3*(1-x).^2.*exp(-(x.^2) - (y+1).^2) ...
   - 10*(x/5 - x.^3 - y.^5).*exp(-x.^2-y.^2) ...
   - 1/3*exp(-(x+1).^2 - y.^2)
```



See Also

`meshgrid` | `surf`

Introduced before R2006a

matlab.perftest.FixedTimeExperiment class

Package: `matlab.perftest`
Superclasses: `matlab.perftest.TimeExperiment`

`TimeExperiment` that collects fixed number of measurements

Description

A `FixedTimeExperiment` is an instance of the `TimeExperiment` class that collects a fixed number of measurements.

To create a `FixedTimeExperiment` instance, use the `matlab.perftest.TimeExperiment.withFixedSampleSize` static method.

Properties

`NumWarmups` — Number of warm-up measurements
0 (default) | nonnegative integer

Number of warm-up measurements, specified as a nonnegative integer. `NumWarmups` defines the number of times that the test framework runs the test code to warm it up.

`SampleSize` — Number of sample measurements to collect after warm-up
positive integer

Number of sample measurements to collect after warm-up, specified as a positive integer.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

See Also

`matlab.perftest.TimeExperiment` | `matlab.unittest.measurement.MeasurementResult`

Introduced in R2016a

matlab.perftest.FrequentistTimeExperiment class

Package: `matlab.perftest`
Superclasses: `matlab.perftest.TimeExperiment`

`TimeExperiment` that collects variable number of measurements

Description

A `FrequentistTimeExperiment` is an instance of the `TimeExperiment` class that collects a variable number of measurements. The test framework collects a variable number of measurements to reach a specified relative margin of error at a specified confidence level.

To create a `FrequentistTimeExperiment` instance, use the `matlab.perftest.TimeExperiment.limitingSamplingError` static method.

Properties

`NumWarmups` — Number of warm-up measurements
4 (default) | nonnegative integer

Number of warm-up measurements, specified as a nonnegative integer. `NumWarmups` defines the number of times that the test framework runs the test code to warm it up.

`MinSamples` — Minimum number of samples
4 (default) | integer greater than 1

Minimum number of sample measurements, specified as an integer greater than 1. `MinSamples` defines the minimum number of times that the test framework runs the test code after `NumWarmups`. The test framework exercises the test code at least `MinSamples` times, regardless of whether the experiment meets the statistical objectives.

`MaxSamples` — Maximum number of samples
256 (default) | integer greater than or equal to `MinSamples`

Maximum number of sample measurements, specified as an integer greater than or equal to `MinSamples`. `MaxSamples` defines the maximum number of times that the test framework runs the test code after `NumWarmups`. The test framework will stop after it takes `MaxSamples` samples, even if the experiment does not meet the statistical objectives.

`RelativeMarginOfError` — Goal relative margin of error for samples
0.05 (default) | positive number

Goal relative margin of error for samples, specified as a positive number.

The framework calculates the Relative Margin of Error for a sample `X` using the equation

$$\text{relMoE} = \frac{\text{T}*\text{std(X)}}{\text{mean(X)} * \sqrt{\text{length(X)}}}$$

where `T` is the T-score from Student's T distribution using the specified `ConfidenceLevel` and `length(X)-1` degrees of freedom.

`ConfidenceLevel` — Confidence level for samples to be within relative margin of error
0.95 (default) | number between 0 and 1

Confidence level for the samples to be within the relative margin of error, specified as a number between 0 and 1.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

See Also

`matlab.perftest.TimeExperiment` | `matlab.unittest.measurement.MeasurementResult`

Introduced in R2016a

matlab.perftest.TestCase class

Package: `matlab.perftest`
Superclasses: `matlab.unittest.TestCase`

Superclass of `matlab.perftest` performance test classes

Description

Use the `matlab.perftest.TestCase` class to write class-based performance tests that can define measurement boundaries. By default, the framework measures performance around the test method boundary. However, test classes that inherit from `matlab.perftest.TestCase` can use the `startMeasuring` and `stopMeasuring` methods to define boundaries to measure specific code segments.

The `matlab.perftest.TestCase` derives from the `matlab.unittest.TestCase` class.

Construction

The testing framework constructs the `matlab.perftest.TestCase` instances.

Methods

keepMeasuring	Measure code with automatic looping
startMeasuring	Designate start of measurement boundary
stopMeasuring	Designate end of measurement boundary

Inherited Methods

addTeardown	Dynamically add teardown routine
applyFixture	Use fixture with `TestCase`
forInteractiveUse	Create TestCase for interactive use
getSharedTestFixtures	Provide access to shared test fixtures
log	Record diagnostic information
onFailure	Dynamically add diagnostics for test failures
run	Run `TestCase` test

Also, the `TestCase` class inherits methods from these classes:

matlab.unittest.qualifications.Assertable	Qualification to validate preconditions of a test
matlab.unittest.qualifications.Assumable	Qualification to filter test content
matlab.unittest.qualifications.FatalAssertable	Qualification to abort test execution
matlab.unittest.qualifications.Verifiable	Qualification to produce soft failure conditions

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Create Performance Test Class

Create a performance test class, `preallocationTest`. The performance testing framework measures the time for each of the four test methods.

```
classdef preallocationTest < matlab.perftest.TestCase
    methods(Test)
        function testOnes(testCase)
            x = ones(1,1e7);
        end

        function testIndexingWithVariable(testCase)
            id = 1:1e7;
            x(id) = 1;
        end

        function testIndexingOnLHS(testCase)
            x(1:1e7) = 1;
        end

        function testForLoop(testCase)
            for i=1:1e7
                x(i) = 1;
            end
        end

    end
end
```

Specify Boundaries for Performance Test Class

Create a performance test class, `fprintfTest`. The performance testing framework measures the code between the calls to the `startMeasuring` and `stopMeasuring` methods. This boundary restricts the performance testing framework to measuring only the call to the `fprintf` function. It excludes setup and teardown actions, and qualifications testing.

```
classdef fprintfTest < matlab.perftest.TestCase
    methods(Test)
        function testPrintingToFile(testCase)
            file = tempname;
            fid = fopen(file, 'w');
            testCase.assertNotEqual(fid, -1, 'IO Problem');

            stringToWrite = repmat('abcdef', 1, 1000000);

            testCase.startMeasuring();
```

```
            fprintf(fid, '%s', stringToWrite);
            testCase.stopMeasuring();

            testCase.verifyEqual(fileread(file), stringToWrite);
            fclose(fid);
        end
    end
end
```

See Also

matlab.perftest.TimeExperiment

Introduced in R2016a

keepMeasuring

Class: `matlab.perftest.TestCase`
Package: `matlab.perftest`

Measure code with automatic looping

Syntax

```
keepMeasuring(testcase)
keepMeasuring(testcase,label)
```

Description

`keepMeasuring(testcase)` instructs the testing framework to iterate through a `while` loop as many times as it needs to get an accurate measurement of performance.

Performance tests that execute too quickly for MATLAB to time accurately are filtered with an assumption failure. With the `keepMeasuring` method, the testing framework can measure significantly faster code by automatically determining the number of times to iterate through code and measuring the average performance.

You cannot put a `keepMeasuring-while` loop between calls to `startMeasuring` and `stopMeasuring`. Similarly, you cannot call the `startMeasuring` and `stopMeasuring` methods inside a `keepMeasuring-while` loop.

`keepMeasuring(testcase,label)` labels the measurement with `label`. Measurements generated in the same test method and with the same label are accumulated and summed. The label is appended in angle brackets to the test element name in the `Samples` and `TestActivity` properties of the `MeasurementResult` object.

Input Arguments

`testcase` — Instance of test case
`matlab.perftest.TestCase` object

Instance of the test case, specified as a `matlab.perftest.TestCase` object.

`label` — Measurement boundary label
valid MATLAB identifier

Measurement boundary label, specified as a valid MATLAB identifier. A valid MATLAB identifier is a character vector or string scalar of alphanumerics (`A`–`Z`, `a`–`z`, `0`–`9`) and underscores, such that the first character is a letter and the length of the character vector is less than or equal to `namelengthmax`.

Examples

Measure Fast Code

Create a performance test class, `ZerosTest`. This parameterized performance test measures the creation of three different sizes of arrays of zeros.

```
classdef ZerosTest < matlab.perftest.TestCase
    properties (TestParameter)
        Size = {1e2,1e3,1e4};
    end

    methods(Test)
        function testOne(testCase,Size)
            A = zeros(Size);
        end
    end
end
```

Run the performance test. The time to create the first two arrays is too close to the precision of the framework and the tests are filtered.

```
results = runperf('ZerosTest');
```

```
Running ZerosTest
........
==============================================================================
ZerosTest/testOne(Size=value1) was filtered.
    Test Diagnostic: The MeasuredTime should not be too close to the precision of the framework.
Details
==============================================================================
.. ......
==============================================================================
ZerosTest/testOne(Size=value2) was filtered.
```

```
    Test Diagnostic: The MeasuredTime should not be too close to the precision of the framework.
Details
==============================================================================
.... ....
Done ZerosTest
_________

Failure Summary:

     Name                            Failed  Incomplete  Reason(s)
    ==============================================================================
     ZerosTest/testOne(Size=value1)              X       Filtered by assumption.
    ------------------------------------------------------------------------------
     ZerosTest/testOne(Size=value2)              X       Filtered by assumption.
```

To get an accurate measurement, modify the `ZerosTest` class to use a `keepMeasuring-while` loop. The performance testing framework measures the code within the `keepMeasuring-while` loop.

```
classdef ZerosTest < matlab.perftest.TestCase
    properties (TestParameter)
        Size = {1e2,1e3,1e4};
    end

    methods(Test)
        function testOne(testCase,Size)
            while(testCase.keepMeasuring)
                A = zeros(Size);
            end
        end
    end
end
```

Rerun the performance test.

```
results = runperf('ZerosTest');
```

```
Running ZerosTest
.......... .......... ....
Done ZerosTest
__________
```

See Also

`matlab.perftest.TimeExperiment` | `runperf`

Topics

“Measure Fast Executing Test Code”

Introduced in R2018b

startMeasuring

Class: `matlab.perftest.TestCase`
Package: `matlab.perftest`

Designate start of measurement boundary

Syntax

```
startMeasuring(testcase)
startMeasuring(testcase,label)
```

Description

`startMeasuring(testcase)` designates the start of a measurement boundary. Invoke this method and the `stopMeasuring` method to restrict measurements to the code between the `startMeasuring` and `stopMeasuring` method calls. Defining this boundary allows you to exclude setup, verification, and teardown code from the measurement.

The performance framework permits multiple, nonnested calls to the `startMeasuring` and `stopMeasuring` methods within each method that is tagged with the `Test` attribute. When creating bounded performance tests, keep the following in mind:

- A test method that calls the `startMeasuring` method must call the `stopMeasuring` method in the scope of the same test method.
- A call to the `startMeasuring` method must have a subsequent call to `stopMeasuring` method in the scope of the same test method. Similarly, a call to the `stopMeasuring` method must have a preceding call to `startMeasuring`.
- You cannot call the `startMeasuring` and `stopMeasuring` methods inside a `while` loop that has the `keepMeasuring` method in the condition. Similarly, you cannot have a `while` loop that has the `keepMeasuring` condition between calls to `startMeasuring` and `stopMeasuring`.
- If a test method has multiple calls to `startMeasuring` and `stopMeasuring`, then the performance framework accumulates and sums the measurements.

If the framework encounters unsupported use of `startMeasuring` and `stopMeasuring` within a test method, it marks the corresponding `MeasurementResult` instance invalid.

`startMeasuring(testcase,label)` designates the start of a measurement boundary and labels the measurement with `label`. Specifying a measurement boundary with a label is similar to specifying one without a label. A call to `startMeasuring` with a label must have a subsequent call to `stopMeasuring` with the same label in the scope of the same test method. If a test method has multiple boundaries with the same label, then the performance framework accumulates the measurements by label and computes the sum. The performance framework does not support nested measurement boundaries.

The label is appended in angle brackets to the test element name in the `Samples` and `TestActivity` properties of the `MeasurementResult`.

Input Arguments

`testcase` — Instance of test case
`matlab.perftest.TestCase` object

Instance of the test case, specified as a `matlab.perftest.TestCase` object.

`label` — Measurement boundary label
valid MATLAB identifier

Measurement boundary label, specified as a valid MATLAB identifier. A valid MATLAB identifier is a character vector or string scalar of alphanumerics (`A`–`Z`, `a`–`z`, `0`–`9`) and underscores, such that the first character is a letter and the length of the character vector is less than or equal to `namelengthmax`.

Examples

Specify Boundaries for Performance Test Class

Create a performance test class, `fprintfTest`. The performance testing framework measures the code between the calls to the `startMeasuring` and `stopMeasuring` methods. This boundary restricts the performance testing framework to measuring only the call to the `fprintf` function. It excludes setup and teardown actions, and qualifications testing.

```
classdef fprintfTest < matlab.perftest.TestCase
    methods(Test)
        function testPrintingToFile(testCase)
            file = tempname;
            fid = fopen(file, 'w');
            testCase.assertNotEqual(fid, -1, 'IO Problem');

            stringToWrite = repmat('abcdef', 1, 1000000);

            testCase.startMeasuring();
            fprintf(fid, '%s', stringToWrite);
            testCase.stopMeasuring();

            testCase.verifyEqual(fileread(file), stringToWrite);
            fclose(fid);
        end
    end
end
```

Refine Measured Code Using Multiple Boundaries

Create a performance test class, `fprintfTest2`. Multiple boundaries (calls to `startMeasuring` and `stopMeasuring`) enable the performance framework to measure the code that opens the file, writes to the file, and closes the file.

```
classdef fprintfTest2 < matlab.perftest.TestCase
    methods(Test)
        function testPrintingToFile(testCase)
            file = tempname;

            testCase.startMeasuring();
            fid = fopen(file,'w');
            testCase.stopMeasuring();

            testCase.assertNotEqual(fid,-1,'IO Problem');
            stringToWrite = repmat('abcdef',1,1000000);

            testCase.startMeasuring();
            fprintf(fid,'%s',stringToWrite);
            testCase.stopMeasuring();

            testCase.verifyEqual(fileread(file),stringToWrite);
```

```
            testCase.startMeasuring();
            fclose(fid);
            testCase.stopMeasuring();
        end
    end
end
```

Run the performance test and view the sample summary. The performance framework measured that the mean time to open, write to, and close the file for the `testPrintingToFile` test was approximately 0.02 seconds. Your results might vary.

```
results = runperf('fprintfTest2');
T = sampleSummary(results)
```

```
Running fprintfTest2
........
Done fprintfTest2
__________


T =

  1×7 table

                  Name                   SampleSize      Mean      StandardDeviation
    ______________________________       __________    ________    _________________

    fprintfTest2/testPrintingToFile          4         0.017003        0.0004943
```

Specify Labeled Measurement Boundaries

Create a performance test class, `examplePerfTest`. The first test has labeled test boundaries for generating an array of random numbers, measuring a call to `svd` with a single output, and measuring a call to `svd` with multiple outputs. The second test has an unlabeled boundary around the call to `svd`.

```
classdef examplePerfTest < matlab.perftest.TestCase
    methods(Test)
        function testSVD1(testCase)
            testCase.startMeasuring('arrayGen')
            X = rand(1000);
```

```
            testCase.stopMeasuring('arrayGen')

            testCase.startMeasuring('SVD_1out')
            S = svd(X);
            testCase.stopMeasuring('SVD_1out')

            testCase.startMeasuring("SVD_3out")
            [U2,S2,V2] = svd(X);
            testCase.stopMeasuring("SVD_3out")

            testCase.verifyEqual(S,diag(S2),'RelTol',1e-14)
        end

        function testSVD2(testCase)
            sz = 732;
            X = rand(sz);

            testCase.startMeasuring()
            [U,S,V] = svd(X);
            testCase.stopMeasuring()

            testCase.verifyTrue(isdiag(S))
            testCase.verifyTrue(issorted(diag(S),'descend'))
            testCase.verifySize(S,[sz sz 1])
        end
    end
end
```

Run the performance test and view the sample summary. Your results might vary. The labels from `testSVD1` are appended in angle brackets to the test element name in the results.

```
results = runperf('examplePerfTest');
T = sampleSummary(results)
```

```
Running examplePerfTest
..........
..........
..........
..........
Done examplePerfTest
__________


T =

  4×7 table
```

Name	SampleSize	Mean	StandardDeviation	Min	Med
examplePerfTest/testSVD1 <arrayGen>	21	0.0096508	0.0012428	0.0087596	0.00
examplePerfTest/testSVD1 <SVD_1out>	21	0.11978	0.0098172	0.10585	0.
examplePerfTest/testSVD1 <SVD_3out>	21	0.30664	0.020991	0.26882	0
examplePerfTest/testSVD2	11	0.13294	0.011135	0.11127	0.

See Also

`matlab.perftest.TimeExperiment` | `stopMeasuring`

Introduced in R2016a

stopMeasuring

Class: `matlab.perftest.TestCase`
Package: `matlab.perftest`

Designate end of measurement boundary

Syntax

```
stopMeasuring(testcase)
stopMeasuring(testcase,label)
```

Description

`stopMeasuring(testcase)` designates the end of a measurement boundary. Invoke this method and the `startMeasuring` method to restrict measurements to the code between the `startMeasuring` and `stopMeasuring` method calls. Defining this boundary allows you to exclude setup, verification, and teardown code from the measurement.

The performance framework permits multiple, nonnested calls to the `startMeasuring` and `stopMeasuring` methods within each method that is tagged with the `Test` attribute. When creating bounded performance tests, keep the following in mind:

- A test method that calls the `startMeasuring` method must call the `stopMeasuring` method in the scope of the same test method.
- A call to the `startMeasuring` method must have a subsequent call to `stopMeasuring` method in the scope of the same test method. Similarly, a call to the `stopMeasuring` method must have a preceding call to `startMeasuring`.
- You cannot call the `startMeasuring` and `stopMeasuring` methods inside a `while` loop that has the `keepMeasuring` method in the condition. Similarly, you cannot have a `while` loop that has the `keepMeasuring` condition between calls to `startMeasuring` and `stopMeasuring`.
- If a test method has multiple calls to `startMeasuring` and `stopMeasuring`, then the performance framework accumulates and sums the measurements.

If the framework encounters unsupported use of `startMeasuring` and `stopMeasuring` within a test method, it marks the corresponding `MeasurementResult` instance invalid.

`stopMeasuring(testcase,label)` designates the end of a measurement boundary and labels the measurement with `label`. Specifying a measurement boundary with a label is similar to specifying one without a label. A call to `stopMeasuring` with a label must have a preceding call to `startMeasuring` with the same label in the scope of the same test method. If a test method has multiple boundaries with the same label, then the performance framework accumulates the measurements by label and computes the sum. The performance framework does not support nested measurement boundaries.

The label is appended in angle brackets to the test element name in the `Samples` and `TestActivity` properties of the `MeasurementResult`.

Input Arguments

`testcase` — Instance of test case
`matlab.perftest.TestCase` object

Instance of the test case, specified as a `matlab.perftest.TestCase` object.

`label` — Measurement boundary label
valid MATLAB identifier

Measurement boundary label, specified as a valid MATLAB identifier. A valid MATLAB identifier is a character vector or string scalar of alphanumerics (`A`–`Z`, `a`–`z`, `0`–`9`) and underscores, such that the first character is a letter and the length of the character vector is less than or equal to `namelengthmax`.

Examples

Specify Boundaries for Performance Test Class

Create a performance test class, `fprintfTest`. The performance testing framework measures the code between the calls to the `startMeasuring` and `stopMeasuring` methods. This boundary restricts the performance testing framework to measuring only the call to the `fprintf` function. It excludes setup and teardown actions, and qualifications testing.

```
classdef fprintfTest < matlab.perftest.TestCase
    methods(Test)
        function testPrintingToFile(testCase)
            file = tempname;
            fid = fopen(file, 'w');
            testCase.assertNotEqual(fid, -1, 'IO Problem');

            stringToWrite = repmat('abcdef', 1, 1000000);

            testCase.startMeasuring();
            fprintf(fid, '%s', stringToWrite);
            testCase.stopMeasuring();

            testCase.verifyEqual(fileread(file), stringToWrite);
            fclose(fid);
        end
    end
end
```

Refine Measured Code Using Multiple Boundaries

Create a performance test class, `fprintfTest2`. Multiple boundaries (calls to `startMeasuring` and `stopMeasuring`) enable the performance framework to measure the code that opens the file, writes to the file, and closes the file.

```
classdef fprintfTest2 < matlab.perftest.TestCase
    methods(Test)
        function testPrintingToFile(testCase)
            file = tempname;

            testCase.startMeasuring();
            fid = fopen(file,'w');
            testCase.stopMeasuring();

            testCase.assertNotEqual(fid,-1,'IO Problem');
            stringToWrite = repmat('abcdef',1,1000000);

            testCase.startMeasuring();
            fprintf(fid,'%s',stringToWrite);
            testCase.stopMeasuring();

            testCase.verifyEqual(fileread(file),stringToWrite);
```

```
            testCase.startMeasuring();
            fclose(fid);
            testCase.stopMeasuring();
        end
    end
end
```

Run the performance test and view the sample summary. The performance framework measured that the mean time to open, write to, and close the file for the `testPrintingToFile` test was approximately 0.02 seconds. Your results might vary.

```
results = runperf('fprintfTest2');
T = sampleSummary(results)
```

```
Running fprintfTest2
........
Done fprintfTest2
__________


T =

  1×7 table

                  Name                   SampleSize      Mean      StandardDeviation
    ____________________________________  __________  ________  _________________  _

    fprintfTest2/testPrintingToFile          4        0.017003       0.0004943       (
```

Specify Labeled Measurement Boundaries

Create a performance test class, `examplePerfTest`. The first test has labeled test boundaries for generating an array of random numbers, measuring a call to `svd` with a single output, and measuring a call to `svd` with multiple outputs. The second test has an unlabeled boundary around the call to `svd`.

```
classdef examplePerfTest < matlab.perftest.TestCase
    methods(Test)
        function testSVD1(testCase)
            testCase.startMeasuring('arrayGen')
            X = rand(1000);
```

```
            testCase.stopMeasuring('arrayGen')

            testCase.startMeasuring('SVD_1out')
            S = svd(X);
            testCase.stopMeasuring('SVD_1out')

            testCase.startMeasuring("SVD_3out")
            [U2,S2,V2] = svd(X);
            testCase.stopMeasuring("SVD_3out")

            testCase.verifyEqual(S,diag(S2),'RelTol',1e-14)
        end

        function testSVD2(testCase)
            sz = 732;
            X = rand(sz);

            testCase.startMeasuring()
            [U,S,V] = svd(X);
            testCase.stopMeasuring()

            testCase.verifyTrue(isdiag(S))
            testCase.verifyTrue(issorted(diag(S),'descend'))
            testCase.verifySize(S,[sz sz 1])
        end
    end
end
```

Run the performance test and view the sample summary. Your results might vary. The labels from `testSVD1` are appended in angle brackets to the test element name in the results.

```
results = runperf('examplePerfTest');
T = sampleSummary(results)
```

```
Running examplePerfTest
..........
..........
..........
..........
Done examplePerfTest
__________


T =

  4×7 table
```

Name	SampleSize	Mean	StandardDeviation	Min	Med
examplePerfTest/testSVD1 <arrayGen>	21	0.0096508	0.0012428	0.0087596	0.00
examplePerfTest/testSVD1 <SVD_1out>	21	0.11978	0.0098172	0.10585	0.
examplePerfTest/testSVD1 <SVD_3out>	21	0.30664	0.020991	0.26882	0
examplePerfTest/testSVD2	11	0.13294	0.011135	0.11127	0.

See Also

`matlab.perftest.TimeExperiment` | `startMeasuring`

Introduced in R2016a

matlab.perftest.TimeExperiment class

Package: `matlab.perftest`

Interface for measuring execution time of code under test

Description

The `matlab.perftest.TimeExperiment` class is the principal interface for measuring the execution time of code under test. The performance testing framework runs a test suite and returns a `matlab.unittest.measurement.MeasurementResult` object for each element of the suite. For each test suite element, the testing framework collects multiple measurements and reports them in the corresponding `MeasurementResult`.

Construction

Instantiate a `TimeExperiment` object using one of its static methods.

Methods

limitingSamplingError	Construct time experiment for specified margin of error and confidence level
run	Run time experiment on test suite
withFixedSampleSize	Construct time experiment with fixed number of measurements

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

See Also

`matlab.perftest.FixedTimeExperiment` | `matlab.perftest.FrequentistTimeExperiment` | `matlab.unittest.TestRunner` | `matlab.unittest.measurement.MeasurementResult`

Introduced in R2016a

matlab.perftest.TimeResult class

Package: `matlab.perftest`
Superclasses: `matlab.unittest.measurement.MeasurementResult`

Result from running time experiment

Description

Objects of the `TimeResult` class hold information describing the result from running a time experiment on a test suite using the `matlab.perftest.TimeExperiment` class. Objects contain information about the time measurements and the corresponding functional test results.

The test results are available as a table.

Creation

The MATLAB testing framework creates an instance of the `TimeResult` class when you run a time experiment.

Properties

`Name` — Name of the test suite element that corresponds to the time results
character vector

Name of the test suite element that corresponds to the time results, returned as a character vector.

Example: 'MyTestSuite'

Attributes:

`GetAccess`	`public`
`SetAccess`	`protected`

Data Types: `char`

`Valid` — Indicator if measurement is valid

`logical` 0 or 1

Indicator if the measurement is valid, returned as `logical` 0 or 1. A measurement is valid (`logical` 1) if it is collected from a passing test. Otherwise it is invalid (`logical` 0).

Attributes:

`GetAccess`	`public`
`SetAccess`	`private`
`Dependent`	`true`
`NonCopyable`	`true`

Data Types: `logical`

`Samples` — Information about sample measurements

table

Information about sample measurements, returned as a table with the following information.

Column Name	Description
Name	Name of the test suite element. If a measurement label is specified, `Name` is appended with the measurement label delimited by angle brackets.
MeasuredValue	Value of the measurement. Experiments can give this column a more specific name, such as `MeasuredTime`.
Timestamp	Time when measurement is collected.
Host	Machine name where measurement is collected.
Platform	Platform architecture on which measurement is collected.
Version	MATLAB Version used in test.
RunIdentifier	Unique identifier of measurement run.

Attributes:

```
GetAccess                                       public
SetAccess                                       private
Dependent                                       true
NonCopyable                                     true
```

Data Types: `table`

`TestActivity` — Information about all test activity
table

Information about all test activity, including the warm-up measurements, returned as a table. This table is useful when you want all the data, such as during debugging or when performing a more involved data analysis. The `TestActivity` table contains these columns in addition to the columns described for `Samples`:

Column Name	Description
Passed	Logical value indicating if the test passed
Failed	Logical value indicating if the test failed
Incomplete	Logical value indicating if test run is incomplete
Objective	Categorical value indicating the type of measurement: `sample` or `warmup`
TestResult	`TestResult` object from the functional test

Attributes:

```
GetAccess                                       public
SetAccess                                       private
Dependent                                       true
NonCopyable                                     true
```

Data Types: `table`

Methods

Public Methods

`comparisonPlot`	Use the `comparisonPlot` method to visually compare the time measurement results of two equal-sized sets of performance tests.
`sampleSummary`	The `TestResult` class inherits this `MeasurementResult` method. Use the `sampleSummary` method as described with the `TestResult` object returned by the `matlab.perftest.TimeExperiment` `run` method.
`samplefun`	The `TestResult` class inherits this `MeasurementResult` method. Use the `samplefun` method as described with the `TestResult` object returned by the `matlab.perftest.TimeExperiment` `run` method.

See Also

`matlab.unittest.TestResult` | `matlab.unittest.TestSuite` | `matlab.unittest.measurement.DefaultMeasurementResult` | `matlab.unittest.measurement.chart.ComparisonPlot`

Topics

"Test Performance Using Scripts or Functions"
"Test Performance Using Classes"

Introduced in R2019a

comparisonPlot

Class: `matlab.perftest.TimeResult`
Package: `matlab.perftest`

Create plot to compare baseline and measurement test results

Syntax

```
comparisonPlot(Baseline,Measurement)
comparisonPlot(Baseline,Measurement,stat)
comparisonPlot( ___ ,Name,Value)
cp = comparisonPlot( ___ )
```

Description

`comparisonPlot(Baseline,Measurement)` creates a plot that visually compares each `TimeResult` object in the arrays `Baseline` and `Measurement`. The comparison is based on the minimum of sample measurement times.

`comparisonPlot(Baseline,Measurement,stat)` specifies the statistic applied to the sample measurement times of each `TimeResult` object in `Baseline` and `Measurement`.

`comparisonPlot( ___ ,Name,Value)` creates a comparison plot with additional options specified by one or more `Name,Value` pair arguments. For example, `comparisonPlot(Baseline,Measurement,'Scale','linear')` creates a plot with a linear scale for the *x*- and *y*-axes.

`cp = comparisonPlot( ___ )` returns the `ComparisonPlot` object, specified as an instance of the `matlab.unittest.measurement.chart.ComparisonPlot` class. Use `cp` to modify the properties of a specific comparison plot after you create it.

Input Arguments

`Baseline` — Results from running primary time experiment
`matlab.perftest.TimeResult` array

Results from running a primary time experiment on a test suite, specified as a `TimeResult` array. `Baseline` must have the same length as `Measurement`.

`Measurement` — Results from running secondary time experiment
`matlab.perftest.TimeResult` array

Results from running a secondary time experiment on a test suite, specified as a `TimeResult` array. `Measurement` must have the same length as `Baseline`.

`stat` — Statistic applied to sample measurement times
`'min'` (default) | 'max' | 'mean' | 'median'

Statistic applied to the sample measurement times of each `TimeResult` object in `Baseline` and `Measurement`, specified as `'min'`, `'max'`, `'mean'`, or `'median'`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `cp = comparisonPlot(Baseline,Measurement,'SimilarityTolerance',0.05,'Scale','linear')`

`Scale` — Scale of *x*- and *y*-axes
'log' (default) | 'linear'

Scale of the *x*- and *y*-axes of the `ComparisonPlot` object, specified as `'log'` or `'linear'`.

`SimilarityTolerance` — Allowable deviation of statistics ratio from one
`0.1` (default) | numeric scalar

Allowable deviation of the statistics ratio from one, for a pair of similar performance tests, specified as a numeric value between `0` and `1`.

`SimilarityTolerance` specifies the borders of a shaded region in the comparison plot. Data points that fall inside this region represent similar `Baseline` and `Measurement` entries.

Parent — Parent container
`Figure` object (default) | `Panel` object | `Tab` object

Parent container in which to plot, specified as a `Figure` object created with either the `figure` or `uifigure` function, a `Panel` object created with the `uipanel` function, or a `Tab` object created with the `uitab` function.

Examples

Visually Compare Sorting Algorithm Performance

Visualize the computational complexity of two sorting algorithms, bubble sort and merge sort, which sort list elements in ascending order. Bubble sort is a simple sorting algorithm that repeatedly steps through a list, compares adjacent pairs of elements, and swaps elements if they are in the wrong order. Merge sort is a "divide and conquer" algorithm that takes advantage of the ease of merging sorted sublists into a new sorted list.

In your current folder, save the following code in `bubbleSort.m`.

```
function y = bubbleSort(x)
% Sorting algorithm with O(n^2) complexity
n = length(x);
swapped = true;

while swapped
    swapped = false;
    for i = 2:n
        if x(i-1) > x(i)
            temp = x(i-1);
            x(i-1) = x(i);
            x(i) = temp;
            swapped = true;
        end
    end
end

y=x;
end
```

Save the following code in `mergeSort.m`.

```
function y = mergeSort(x)
% Sorting algorithm with O(n*logn) complexity
y = x; % A list of one element is considered sorted

if length(x) > 1
    mid = floor(length(x)/2);
    L = x(1:mid);
    R = x((mid+1):end);

    % Sort left and right sublists recursively
    L = mergeSort(L);
    R = mergeSort(R);

    % Merge the sorted left (L) and right (R) sublists
    i = 1;
    j = 1;
    k = 1;
    while i <= length(L) && j <= length(R)
        if L(i) < R(j)
            y(k) = L(i);
            i = i + 1;
        else
            y(k) = R(j);
            j = j + 1;
        end
        k = k + 1;
    end

    % At this point, either L or R is empty
    while i <= length(L)
        y(k) = L(i);
        i = i + 1;
        k = k + 1;
    end

    while j <= length(R)
        y(k) = R(j);
        j = j + 1;
        k = k + 1;
    end
end

end
```

Create the following parameterized test class, `TestSort`, which compares the performance of the bubble sort and merge sort algorithms. The `len` property contains the number of list elements you want to test.

```
classdef TestSort < matlab.perftest.TestCase
    properties
        Data
        SortedData
    end

    properties(ClassSetupParameter)
        % Create 25 logarithmically spaced values between 10^2 and 10^4
        len = num2cell(round(logspace(2,4,25)));
    end

    methods(TestClassSetup)
        function ClassSetup(testCase,len)
            orig = rng;
            testCase.addTeardown(@rng,orig)
            rng('default')
            testCase.Data = rand(1,len);
            testCase.SortedData = sort(testCase.Data);
        end
    end

    methods(Test)
        function testBubbleSort(testCase)
            while testCase.keepMeasuring
                y = bubbleSort(testCase.Data);
            end
            testCase.verifyEqual(y,testCase.SortedData);
        end

        function testMergeSort(testCase)
            while testCase.keepMeasuring
                y = mergeSort(testCase.Data);
            end
            testCase.verifyEqual(y,testCase.SortedData);
        end

    end
end
```

Run performance tests for all test elements that correspond to the `'testBubbleSort'` method, and save the results in the `Baseline` array. Your results might vary from the results shown.

```
Baseline = runperf('TestSort','ProcedureName','testBubbleSort');
```

```
Running TestSort
.......... .......... .......... .......... ..........
.......... .......... .......... .......... ..........
.......... .......... .......... .......... ..........
.......... .......... .......... .......... ..........
.......... .......... .......... .......... ..........
.......... .......... .....
Done TestSort
__________
```

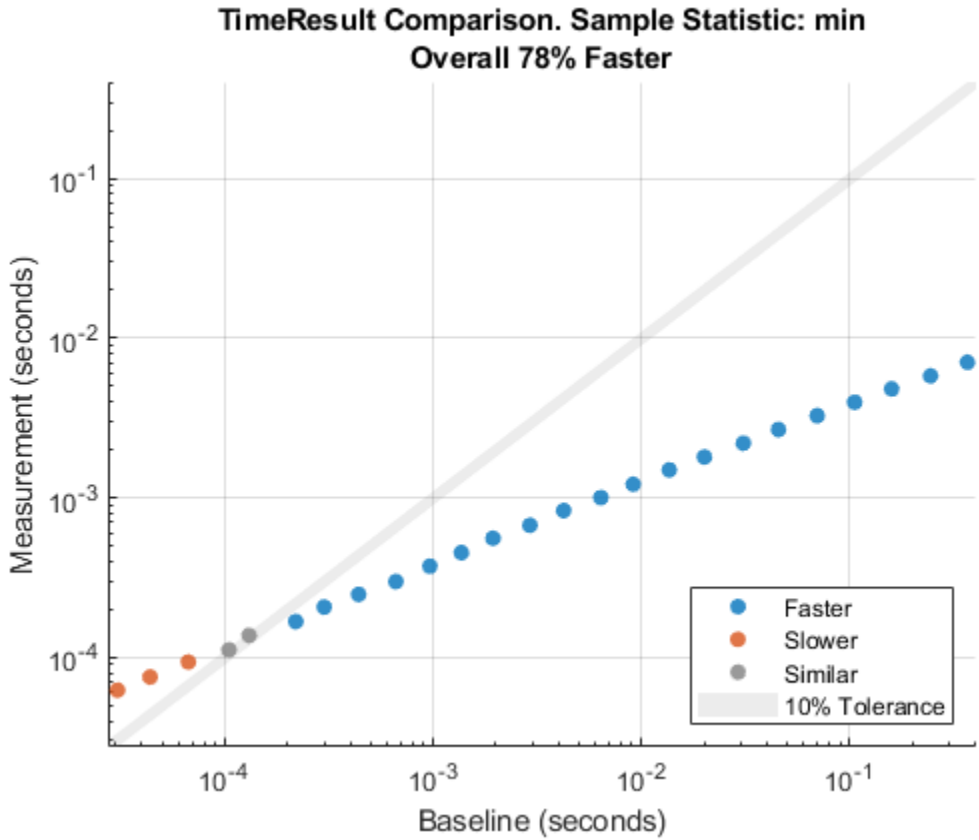
Run performance tests for all elements that correspond to the `'testMergeSort'` method, and save the results in the `Measurement` array.

```
Measurement = runperf('TestSort','ProcedureName','testMergeSort');
```

```
Running TestSort
.......... .......... .......... .......... ..........
.......... .......... .......... .......... ..........
.......... .......... .......... .......... ..........
.......... .......... .......... .......... ..........
Done TestSort
__________
```

Visually compare the minimum of the `MeasuredTime` column in the `Samples` table for each corresponding pair of `Baseline` and `Measurement` objects.

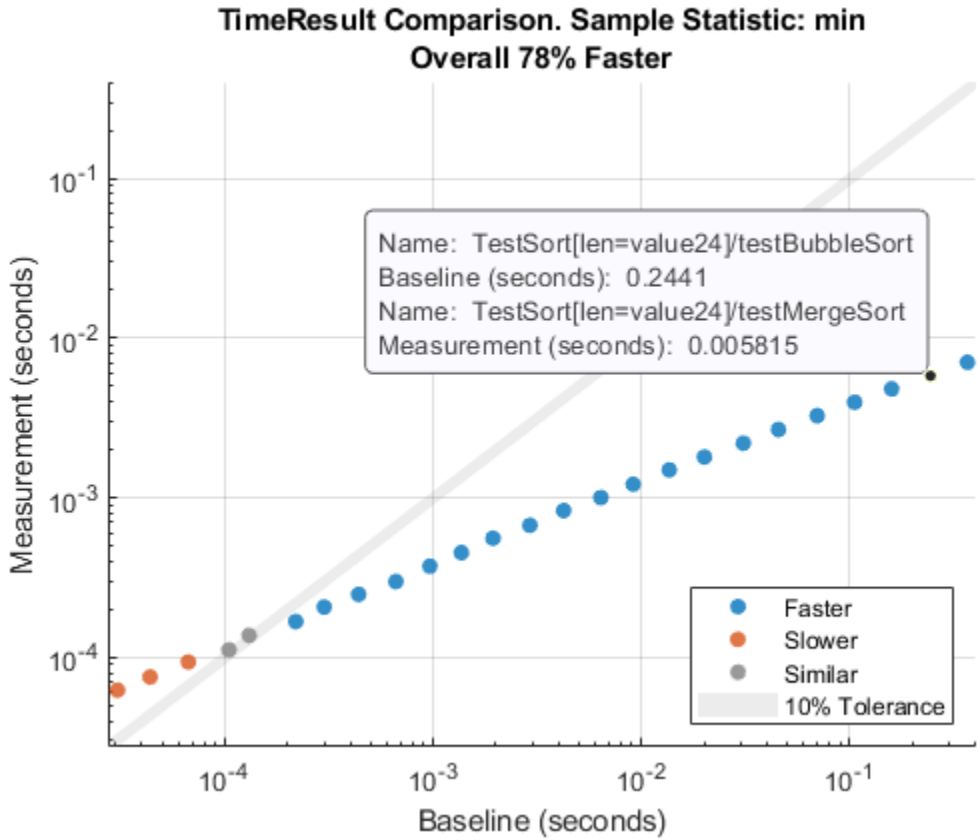
```
comparisonPlot(Baseline,Measurement);
```



In this comparison plot, most data points are blue because they are below the shaded similarity region. This result indicates the superior performance of merge sort for the majority of test cases. However, for small enough lists, bubble sort performs better than or comparable to merge sort, as shown by the orange and gray points on the plot.

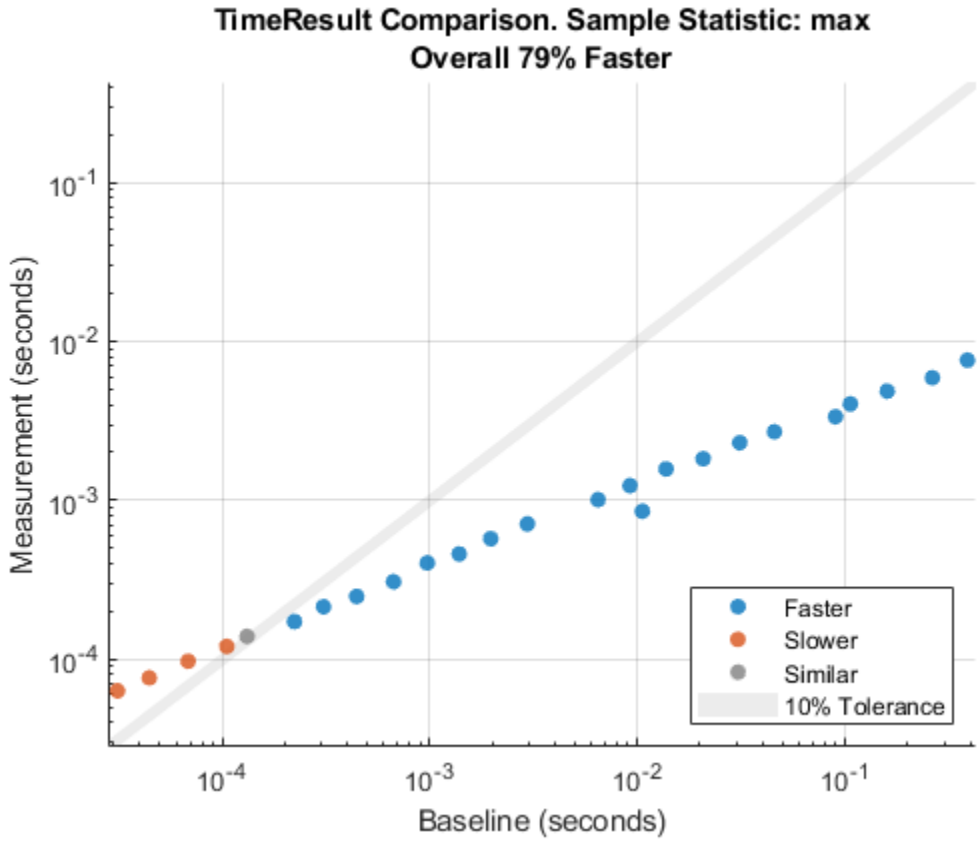
As a comparison summary, the plot reports a 78% performance improvement due to merge sort. This value is the geometric mean of the improvement percentages corresponding to all data points. If the comparison plot contained invalid data points, a comparison summary would not be generated.

You can press or hover over any data point to view detailed information about the time measurement results being compared.



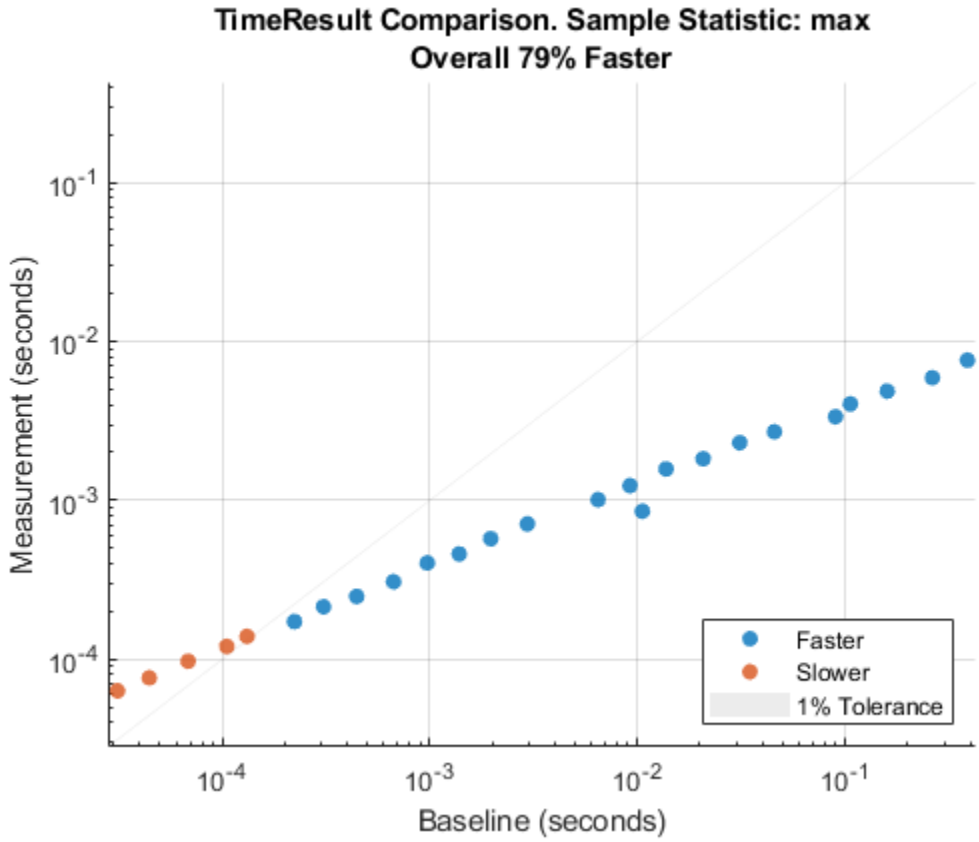
To study the worst-case sorting algorithm performance for different list lengths, create a comparison plot based on the maximum of sample measurement times.

```
comparisonPlot(Baseline,Measurement,'max');
```



Reduce `SimilarityTolerance` to `0.01` when comparing the maximum of sample measurement times. Return the `ComparisonPlot` object in `cp` so that you can modify its properties later.

```
cp = comparisonPlot(Baseline,Measurement,'max','SimilarityTolerance',0.01);
```



See Also

`matlab.unittest.measurement.chart.ComparisonPlot` | `runperf`

Introduced in R2019b

matlab.perftest.TimeExperiment.limitingSamplingError

Class: `matlab.perftest.TimeExperiment`
Package: `matlab.perftest`

Construct time experiment for specified margin of error and confidence level

Syntax

```
experiment = matlab.perftest.TimeExperiment.limitingSamplingError
experiment = matlab.perftest.TimeExperiment.limitingSamplingError(
Name,Value)
```

Description

`experiment = matlab.perftest.TimeExperiment.limitingSamplingError` constructs a time experiment for each test suite element, with the specified statistical objectives (such as margin of error and confidence level). This method returns an instance of `FrequentistTimeExperiment`. This syntax uses the following defaults to determine the number of sample measurements.

- Number of warm-up measurements: 4
- Minimum number of samples: 4
- Maximum number of samples collected in the event other statistical objectives are not met: 256
- Objective relative margin of error for samples: 0.05 (5%)
- Confidence level for samples to be within relative margin of error: 0.95 (95%)

`experiment = matlab.perftest.TimeExperiment.limitingSamplingError(Name,Value)` constructs a time experiment with additional options specified by one or more `Name,Value` pair arguments. Use this syntax to override the defaults listed above.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `experiment = matlab.perftest.TimeExperiment.limitingSamplingError('RelativeMarginOfError',0.12,'MaxSamples',100)` creates a time experiment that collects sample measurements until samples have a relative margin of error of 12%, or until it collects 100 measurements.

`NumWarmups` — Number of warm-up measurements
4 (default) | nonnegative integer

Number of warm-up measurements, specified as a nonnegative integer. The value defines the number of times that the test framework exercises the test code to warm it up.

`MinSamples` — Minimum number of samples
4 (default) | positive integer

Minimum number of sample measurements, specified as a positive integer. The value defines the minimum number of times that the test framework exercises the test code after any warm-up runs. The test framework exercises the test code at least `MinSamples` times, regardless of whether the experiment meets the statistical objectives.

`MaxSamples` — Maximum number of samples
256 (default) | positive integer

Maximum number of sample measurements, specified as a positive integer. The value defines the maximum number of times that the test framework exercises the test code after `NumWarmups`. If the experiment does not meet the statistical objectives, the testing framework collects up to `MaxSamples`.

`RelativeMarginOfError` — Objective relative margin of error for samples
0.05 (default) | positive number

Objective relative margin of error for samples, specified as a positive number.

The testing framework calculates the relative margin of error for a sample X using the equation

$$\text{relMoE} = \frac{\text{T}*\text{std(X)}}{\text{mean(X)} * \sqrt{\text{length(X)}}}$$

where T is the T-score from Student's T distribution using the specified `ConfidenceLevel` and `length(X)-1` degrees of freedom.

`ConfidenceLevel` — Confidence level for samples to be within relative margin of error

0.95 (default) | number between 0 and 1

Confidence level for the samples to be within the relative margin of error, specified as a number from 0 through 1.

Examples

Performance Test with Variable Number of Measurements

In your current working folder, create a class-based test, `preallocationTest.m`, that compares different methods of preallocation.

```
classdef preallocationTest < matlab.perftest.TestCase
    methods(Test)
        function testOnes(testCase)
            x = ones(1,1e7);
        end
        function testIndexingWithVariable(testCase)
            id = 1:1e7;
            x(id) = 1;
        end
        function testIndexingOnLHS(testCase)
            x(1:1e7) = 1;
        end
        function testForLoop(testCase)
            for i=1:1e7
                x(i) = 1;
            end
        end
```

```
    end
end
```

Create a test suite.

```
suite = testsuite('preallocationTest');
```

Construct a time experiment with a variable number of sample measurements, and run the tests.

```
import matlab.perftest.TimeExperiment
experiment = TimeExperiment.limitingSamplingError;
result = run(experiment,suite);
```

```
Running preallocationTest
.......... .......... .......... ..
Done preallocationTest
__________
```

View the test activity for the first test. Your results might vary.

```
result(1).TestActivity
```

```
ans =

  8×12 table

                Name                  Passed    Failed    Incomplete    MeasuredTime    Obj
    ____________________________    ______    ______    __________    ____________    ___

    preallocationTest/testOnes      true      false       false         0.056052       wa
    preallocationTest/testOnes      true      false       false         0.056227       wa
    preallocationTest/testOnes      true      false       false         0.055969       wa
    preallocationTest/testOnes      true      false       false         0.054961       wa
    preallocationTest/testOnes      true      false       false         0.052572       sa
    preallocationTest/testOnes      true      false       false         0.051743       sa
    preallocationTest/testOnes      true      false       false         0.051709       sa
    preallocationTest/testOnes      true      false       false         0.051256       sa
```

For this test, the performance testing framework collected 4 warm-up measurements (the default), and 11 sample measurements. After 11 sample measurements, the performance testing framework satisfied the default statistical objectives.

Construct a time experiment that collects two warm-up measurements and runs the tests a variable number of times to reach a sample mean with a 10% relative margin of error within a 90% confidence level.

```
experiment = TimeExperiment.limitingSamplingError('NumWarmups',2,...
    'RelativeMarginOfError',0.10, 'ConfidenceLevel', 0.90);
result = run(experiment, suite);
```

```
Running preallocationTest
.......... .......... ....
Done preallocationTest
__________
```

View the test activity for the first test. Your results might vary.

```
result(1).TestActivity
```

```
ans =
  6×12 table

                Name                Passed    Failed    Incomplete    MeasuredTime    Obj
    ____________________________    ______    ______    __________    ____________    ___

    preallocationTest/testOnes      true      false       false         0.053963       wa
    preallocationTest/testOnes      true      false       false         0.053086       wa
    preallocationTest/testOnes      true      false       false         0.052502       sa
    preallocationTest/testOnes      true      false       false          0.05252       sa
    preallocationTest/testOnes      true      false       false         0.052048       sa
    preallocationTest/testOnes      true      false       false         0.052434       sa
```

For this test, the performance testing framework collected two warm-up measurements and nine sample measurements. After nine sample measurements, the performance testing framework satisfied the specified statistical objectives.

See Also

`matlab.perftest.FrequentistTimeExperiment` | `matlab.perftest.TimeExperiment.withFixedSampleSize`

Topics

"Overview of Performance Testing Framework"

Introduced in R2016a

run

Class: `matlab.perftest.TimeExperiment`
Package: `matlab.perftest`

Run time experiment on test suite

Syntax

```
results = run(experiment,suite)
```

Description

`results = run(experiment,suite)` runs a time experiment on a test suite, and returns an array of `TimeResult` objects. Each element in `results` corresponds to an element in `suite`.

Input Arguments

`experiment` — Experiment to collect measurements on
`matlab.perftest.TimeExperiment` instance

Experiment to collect measurements on, specified as a `matlab.perftest.TimeExperiment` instance.

`suite` — Suite of tests
`matlab.unittest.Test` array

Suite of tests, specified as a `matlab.unittest.Test` array.

Examples

Run Time Experiment

In your current working folder, create a class-based test, preallocationTest.m, that compares different methods of preallocation.

```
classdef preallocationTest < matlab.perftest.TestCase
    methods(Test)
        function testOnes(testCase)
            x = ones(1,1e7);
        end

        function testIndexingWithVariable(testCase)
            id = 1:1e7;
            x(id) = 1;
        end

        function testIndexingOnLHS(testCase)
            x(1:1e7) = 1;
        end

        function testForLoop(testCase)
            for i=1:1e7
                x(i) = 1;
            end
        end

    end
end
```

Create a test suite.

```
suite = testsuite('preallocationTest');
```

Construct a time experiment with a fixed number of sample measurements, and run the tests.

```
import matlab.perftest.TimeExperiment
numSamples = 10;
experiment = TimeExperiment.withFixedSampleSize(numSamples);
result = run(experiment,suite)
```

```
Running preallocationTest
.......... .......... .......... ..........
```

```
Done preallocationTest
__________


result =

  1x4 TimeResult array with properties:

    Name
    Valid
    Samples
    TestActivity

Totals:
   4 Valid, 0 Invalid.
   30.7234 seconds testing time.
```

Alternatives

If you do not need to create an experiment and test suite explicitly, you can use `runperf`.

See Also

`runperf`

Topics

"Overview of Performance Testing Framework"

Introduced in R2016a

matlab.perftest.TimeExperiment.withFixedSampleSize

Class: `matlab.perftest.TimeExperiment`
Package: `matlab.perftest`

Construct time experiment with fixed number of measurements

Syntax

```
experiment = matlab.perftest.TimeExperiment.withFixedSampleSize(
numSamples)
```

Description

`experiment = matlab.perftest.TimeExperiment.withFixedSampleSize(numSamples)` constructs a time experiment with a fixed number of measurements. This method returns an instance of `FixedTimeExperiment`.

`experiment = matlab.perftest.TimeExperiment.withFixedSampleSize(numSamples,'NumWarmups',numWarmups)` configures the time experiment to first warm up the code by exercising it `numWarmups` times.

Input Arguments

`numSamples` — Number of sample measurements to collect
positive integer

Number of sample measurements to collect, specified as a positive integer. If you specified a number of warm-ups, the testing framework first exercises the code `numWarmups` times before collecting `numSamples` measurements.

`numWarmups` — Number of warm-up measurements
0 (default) | nonnegative integer

Number of warm-up measurements, specified as a nonnegative integer. `numWarmups` defines the number of times that the test framework exercises the test code to warm it up. Warming up the code gives a more realistic analysis of typical execution time, since it minimizes the effects of first-time run costs.

Example: `experiment = matlab.perftest.TimeExperiment.withFixedSampleSize(24,'NumWarmups',8)` constructs a `FixedTimeExperiment` that exercises the code 8 times to warm it up and then exercises the code 24 times to collect sample measurements.

Examples

Performance Test with Fixed Number of Measurements

In your current working folder, create a class-based test, `preallocationTest.m`, that compares different methods of preallocation.

```
classdef preallocationTest < matlab.perftest.TestCase
    methods(Test)
        function testOnes(testCase)
            x = ones(1,1e7);
        end
        function testIndexingWithVariable(testCase)
            id = 1:1e7;
            x(id) = 1;
        end
        function testIndexingOnLHS(testCase)
            x(1:1e7) = 1;
        end
        function testForLoop(testCase)
            for i=1:1e7
                x(i) = 1;
            end
        end

    end
end
```

Create a test suite.

```
suite = testsuite('preallocationTest');
```

Construct a time experiment with a fixed number of sample measurements, and run the tests.

```
import matlab.perftest.TimeExperiment
numSamples = 6;
experiment = TimeExperiment.withFixedSampleSize(numSamples);
result = run(experiment,suite);
```

```
Running preallocationTest
..........
..........
....
Done preallocationTest
__________
```

View the test activity for the fourth test.

```
result(4).TestActivity
```

```
ans =

                Name                 Passed    Failed    Incomplete    MeasuredTime
    ____________________________     ______    ______    __________    ____________

    preallocationTest/testForLoop    true      false     false         0.90553
    preallocationTest/testForLoop    true      false     false         0.86564
    preallocationTest/testForLoop    true      false     false         0.75888
    preallocationTest/testForLoop    true      false     false         0.74051
    preallocationTest/testForLoop    true      false     false          0.8735
    preallocationTest/testForLoop    true      false     false         0.83188
```

The performance testing framework collected six sample measurements for each test.

Construct a time experiment that also runs the code 3 times to warm it up. Run the tests.

```
numWarmups = 3;
experiment = TimeExperiment.withFixedSampleSize(numSamples,'NumWarmups',numWarmups);
result = run(experiment,suite);
```

```
Running preallocationTest
..........
..........
....
Done preallocationTest
__________
```

View the test activity for the fourth test.

```
result(4).TestActivity
```

```
ans =

                Name                 Passed    Failed    Incomplete    MeasuredTime
    _____________________________    ______    ______    __________    ____________

    preallocationTest/testForLoop    true      false     false         0.82972
    preallocationTest/testForLoop    true      false     false         0.85917
    preallocationTest/testForLoop    true      false     false         0.85857
    preallocationTest/testForLoop    true      false     false         0.85307
    preallocationTest/testForLoop    true      false     false         0.86655
    preallocationTest/testForLoop    true      false     false         0.81533
    preallocationTest/testForLoop    true      false     false         0.88266
    preallocationTest/testForLoop    true      false     false         0.94436
    preallocationTest/testForLoop    true      false     false          1.0375
```

For each test, the performance testing framework collected three warm-up measurements in addition to the six sample measurements.

See Also

`matlab.perftest.FixedTimeExperiment` | `matlab.perftest.TimeExperiment.limitingSamplingError`

Topics

"Overview of Performance Testing Framework"

Introduced in R2016a

perl

Call Perl script using operating system executable

Syntax

```
perl(perlfile)
perl(perlfile,arg1,...,argN)
result = perl( ___ )
[result, status] = perl( ___ )
```

Description

`perl(perlfile)` calls the Perl script `perlfile`.

On Microsoft Windows systems, MATLAB ships with Perl, which is available from the Perl.org website. For information about using the Perl programming language, Perl source code, and a standard distribution of Perl, see www.perl.org.

On Linux and Mac systems, MATLAB calls the Perl interpreter available with the operating system.

`perl(perlfile,arg1,...,argN)` calls the script with the arguments `arg1,...,argN`.

`result = perl( ___ )` returns the results. Use this option with any of the input arguments from the previous syntaxes.

`[result, status] = perl( ___ )` returns the exit status in `status`.

Examples

Run `hello.pl` Perl Script

Create a file `hello.pl` containing these statements:

```
$input = $ARGV[0];
print "Hello $input.";
```

Save the file on your MATLAB path.

At the MATLAB command line, type:

```
perl('hello.pl','World')
```

```
ans =
Hello World.
```

Input Arguments

`perlfile` — Perl script
string | character vector

Perl script file name, specified as a string or character vector.

Example: `'myscript.pl'`

`arg1,...,argN` — Perl script input arguments
string | character vector

Perl script input arguments 1 through N (if any) required by `perlfile`, specified as strings or character vectors. The script specifies the arguments.

Output Arguments

`result` — Script output
string

Script output, returned as a string.

`status` — Script exit status
integer

Script exit status, returned as an integer.

Tips

- Consider using Perl scripts instead of MATLAB code when:
  - You want to use an existing Perl script.
  - You want to preprocess data quickly. MATLAB reads Perl script formatting more easily than other formatting.
  - Perl has features not supported by MATLAB.

Compatibility Considerations

MATLAB ships with Perl version 5.26.1

Behavior changed in R2018a

To use the `perl` command on Windows platforms, go to https://perldoc.perl.org/perl.html and select Perl version 5.26.1 for information about using this version.

MATLAB ships with Perl version 5.24.1

Behavior changed in R2017b

To use the `perl` command on Windows platforms, go to https://perldoc.perl.org/perl.html and select Perl version 5.24 for information about using this version.

See Also

`!` (exclamation point) | `dos` | `regexp` | `system` | `unix`

Introduced before R2006a

perms

All possible permutations

Syntax

```
P = perms(v)
```

Description

`P = perms(v)` returns a matrix containing all permutations of the elements of vector `v` in reverse lexicographic order. Each row of `P` contains a different permutation of the *n* elements in `v`. Matrix `P` has the same data type as `v`, and it has *n*! rows and *n* columns.

Examples

All Permutations of Double Integers

```
v = [2 4 6];
P = perms(v)
```

```
P = 6×3

     6     4     2
     6     2     4
     4     6     2
     4     2     6
     2     6     4
     2     4     6
```

All Permutations of Unsigned Integers

```
v = uint16([1023 4095 65535]);
P = perms(v)
```

```
P = 6x3 uint16 matrix

   65535    4095    1023
   65535    1023    4095
    4095   65535    1023
    4095    1023   65535
    1023   65535    4095
    1023    4095   65535
```

All Permutations of Complex Numbers

```
v = [1+1i 2+1i 3+1i];
P = perms(v)
```

```
P = 6×3 complex

   3.0000 + 1.0000i   2.0000 + 1.0000i   1.0000 + 1.0000i
   3.0000 + 1.0000i   1.0000 + 1.0000i   2.0000 + 1.0000i
   2.0000 + 1.0000i   3.0000 + 1.0000i   1.0000 + 1.0000i
   2.0000 + 1.0000i   1.0000 + 1.0000i   3.0000 + 1.0000i
   1.0000 + 1.0000i   3.0000 + 1.0000i   2.0000 + 1.0000i
   1.0000 + 1.0000i   2.0000 + 1.0000i   3.0000 + 1.0000i
```

Input Arguments

v — Set of items
vector of numeric, `logical`, or `char` values

Set of items, specified as a vector of numeric, `logical`, or `char` values.

Example: `[1 2 3 4]`

Example: `[1+1i 2+1i 3+1i 4+1i]`

Example: `int16([1 2 3 4])`

Example: `['abcd']`

Example: `[true false true false]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

Limitations

`perms(v)` is practical when `length(v)` is less than about 10.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`nchoosek` | `permute` | `randperm`

Introduced before R2006a

permute

Permute array dimensions

Syntax

```
B = permute(A,dimorder)
```

Description

`B = permute(A,dimorder)` rearranges the dimensions of an array in the order specified by the vector `dimorder`. For example, `permute(A,[2 1])` switches the row and column dimensions of a matrix `A`.

Examples

3-D Array

Create a 3-by-4-by-2 array and permute it so that the first and third dimensions are switched, resulting in a 2-by-4-by-3 array.

```
rng default
A = rand(3,4,2)
```

```
A =
A(:,:,1) =

    0.8147    0.9134    0.2785    0.9649
    0.9058    0.6324    0.5469    0.1576
    0.1270    0.0975    0.9575    0.9706


A(:,:,2) =

    0.9572    0.1419    0.7922    0.0357
    0.4854    0.4218    0.9595    0.8491
```

```
    0.8003    0.9157    0.6557    0.9340
```

```
B = permute(A,[3 2 1])
```

```
B =
B(:,:,1) =

    0.8147    0.9134    0.2785    0.9649
    0.9572    0.1419    0.7922    0.0357


B(:,:,2) =

    0.9058    0.6324    0.5469    0.1576
    0.4854    0.4218    0.9595    0.8491


B(:,:,3) =

    0.1270    0.0975    0.9575    0.9706
    0.8003    0.9157    0.6557    0.9340
```

```
szB = size(B)
```

```
szB = 1×3

     2     4     3
```

Input Arguments

A — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

`dimorder` — Dimension order
row vector

Dimension order, specified as a row vector with unique, positive integer elements that represent the dimensions of the input array.

Compatibility Considerations

Noninteger or complex dimension order arguments produce error

Behavior changed in R2019b

Starting in R2019b, the syntax `permute(A,dimorder)` produces an error when `dimorder` is a noninteger or complex value. Instead, specify real, positive integer values for `dimorder`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitation:

Permuting the tall dimension (dimension one) is not supported.

For more information, see “Tall Arrays for Out-of-Memory Data”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Does not support cell arrays for the first argument.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`ipermute` | `reshape` | `shiftdim` | `transpose`

Introduced before R2006a

persistent

Define persistent variable

Syntax

```
persistent var1 ... varN
```

Description

`persistent var1 ... varN` declares variables `var1 ... varN` as persistent. Persistent variables are local to the function in which they are declared, yet their values are retained in memory between calls to the function. Code at the MATLAB command line and in other functions cannot change persistent variables.

When MATLAB first encounters a particular `persistent` statement, it initializes the persistent variable to an empty matrix (`[]`).

MATLAB clears persistent variables when you clear or modify a function that is in memory. To keep a function in memory, use `mlock`.

Examples

Count Calls to Function

Create the function `myFun` in your current working folder. Each time you call the function, the value of `n` increases.

```
function myFun()
    persistent n
    if isempty(n)
        n = 0;
    end
    n = n+1
end
```

At the command prompt, call `myFun` three times.

```
myFun
myFun
myFun
```

```
n =

     1


n =

     2


n =

     3
```

Clear `myFun` and call it another two times. Clearing the function also clears the persistent variable.

```
clear myFun
myFun
myFun
```

```
n =

     1


n =

     2
```

Log Data at Specified Time Interval

Write a function that logs data if at least three seconds have passed since the last log entry. Define `logTime` as a persistent variable that stores the last time `logData` wrote to the file.

In a file in your current working folder, define the `logData` function.

```
function logData(fname,n)
    persistent logTime
```

```
    currTime = datetime;

    if isempty(logTime)
        logTime = currTime;
        disp('Logging initial value.')
        dlmwrite(fname,n)
        return
    end

    dt = currTime - logTime;
    if dt > seconds(3)
        disp('Logging.')
        dlmwrite(fname,n,'-append')
        logTime = currTime;
    else
      disp(['Not logging. ' num2str(seconds(dt)) ' sec since last log.'])
    end
end
```

At the command prompt, call `logData` in a loop. The loop has 10 iterations, and each iteration takes approximately 1 second. Therefore, MATLAB writes 4 values to `myLog.txt` (at approximately 0, 3, 6, and 9 seconds).

```
for n = 1:10
    pause(1)
    logData('myLog.txt',rand)
end
```

```
Logging initial value.
Not logging. 1.005 sec since last log.
Not logging. 2.009 sec since last log.
Logging.
Not logging. 1.007 sec since last log.
Not logging. 2.013 sec since last log.
Logging.
Not logging. 1.005 sec since last log.
Not logging. 2.007 sec since last log.
Logging.
```

Call the `logData` function again to append another value.

```
logData('myLog.txt',rand)
```

```
Logging.
```

Clear the `logData` function to reinitialize the persistent variable. Call the `logData` function again. This time, the function overwrites `myLog.txt` instead of appending a value.

```
clear logData
logData('myLog.txt',rand)
```

```
Logging initial value.
```

Tips

- Persistent variables are similar to global variables because MATLAB creates permanent storage for both. They differ from global variables because persistent variables are known only to the function that declares them. Therefore, code at the MATLAB command line or other functions cannot change persistent variables.
- Since MATLAB initializes a persistent variable to an empty matrix (`[]`), typically functions check to see if a persistent variable is empty, and, if so, initialize it.

```
function myFun()
    persistent n
    if isempty(n)
        n = 0;
    end
    n = n+1;
end
```

- The declaration of a variable as persistent must precede any other references to the variable, including input or output arguments. For example, the `persistent` declarations in the following functions are invalid.

```
function myfunA(x)
    persistent x
end

function myfunB
    x = 0;
    persistent x
end
```

- To clear a persistent variable, use `clear` with the name of the function that declares the variable. For example, `clear myFun`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Persistent variables in the generated code do not share values with MATLAB.
- Every generated MEX function has its own copy of persistent data. Each invocation of a particular MEX function reuses values from prior invocations.
- In a Simulink model, each MATLAB Function block contains its own copy of persistent data. If a MATLAB function that contains a persistent variable is called from two different blocks, the model has two persistent variables. Also, each run of the simulation creates a new copy of the persistent data.
- In the MATLAB code intended for code generation, a persistent variable `p` must be assigned before `p` is used. The only exception is a check using `isempty(p)` that can be performed before assignment. You can use this check to make sure that `p` is assigned before it is used in your MATLAB code.
- If the parent function declares a persistent variable, the function must assign the variable before it calls a nested function that uses the persistent variable.
- Persistent variables cannot be renamed in generated code. See “Reuse the Same Variable with Different Properties” (MATLAB Coder).
- The body of a `parfor`-loop cannot contain a persistent variable declaration.
- The generated code does not enforce order of evaluation in expressions. For most expressions, the order of evaluation is not significant. However, for expressions with side effects, the generated code can produce the side effects in different order from the original MATLAB code. Expressions that produce side effects include those that modify persistent variables. See “Differences Between Generated Code and MATLAB Code” (MATLAB Coder).
- A handle object that a persistent variable refers to must be a singleton object. See “Handle Object Limitations for Code Generation” (MATLAB Coder).
- The code generator computes class initial values at class loading time before code generation. If you use persistent variables in MATLAB class property initialization, the value of the persistent variable that is computed when the class loads belongs to MATLAB. It is not the value that is used at code generation time.

  In particular, if you use `coder.target` in a MATLAB class property initialization, `coder.target('MATLAB')` returns `true`.

  See “MATLAB Classes Definition for Code Generation” (MATLAB Coder).

See Also

`clear` | `global` | `isempty` | `mislocked` | `mlock` | `munlock`

Introduced before R2006a

pi

Ratio of circle's circumference to its diameter

Syntax

```
p = pi
```

Description

`p = pi` returns the floating-point number nearest to the value of π in IEEE double-precision. For more information on floating-point numbers, see “Floating-Point Numbers”.

Examples

Value of Pi

Return the value of π in double precision, which has 15 digits after the decimal point.

```
format long
p = pi
```

```
p =
   3.141592653589793
```

Area of Circle

Calculate the area of a circle with radius 7.

```
r = 7;
A = pi*r^2
```

```
A = 153.9380
```

Surface Area and Volume of Sphere

Calculate the surface area and volume of a sphere with radius 5.

```
r = 5;
SA = 4*pi*r^2
```

```
SA = 314.1593
```

```
V = 4/3*pi*r^3
```

```
V = 523.5988
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`cos` | `cospi` | `rad2deg` | `sin` | `sinpi`

Topics

"Floating-Point Numbers"

Introduced before R2006a

pie

Pie chart

Syntax

```
pie(X)
pie(X,explode)
pie(X,labels)
pie(X,explode,labels)
```

```
pie(ax, ___ )
```

```
p = pie( ___ )
```

Description

`pie(X)` draws a pie chart using the data in `X`. Each slice of the pie chart represents an element in `X`.

- If `sum(X) ≤ 1`, then the values in `X` directly specify the areas of the pie slices. `pie` draws only a partial pie if `sum(X) < 1`.
- If `sum(X) > 1`, then `pie` normalizes the values by `X/sum(X)` to determine the area of each slice of the pie.
- If `X` is of data type `categorical`, the slices correspond to categories. The area of each slice is the number of elements in the category divided by the number of elements in `X`.

`pie(X,explode)` offsets slices from the pie. `explode` is a vector or matrix of zeros and nonzeros that correspond to `X`. The `pie` function offsets slices for the nonzero elements only in `explode`.

If `X` is of data type `categorical`, then `explode` can be a vector of zeros and nonzeros corresponding to categories, or a cell array of the names of categories to offset.

`pie(X,labels)` specifies text labels for the slices. `X` must be numeric. The number of labels must equal the number of elements in `X`.

`pie(X,explode,labels)` offsets slices and specifies the text labels. `X` can be numeric or categorical. For numeric `X`, the number of labels must equal the number of elements in `X`. For categorical `X`, the number of labels must equal the number of categories.

`pie(ax, ___ )` plots into the axes specified by `ax` instead of into the current axes (`gca`). The option `ax` can precede any of the input argument combinations in the previous syntaxes.

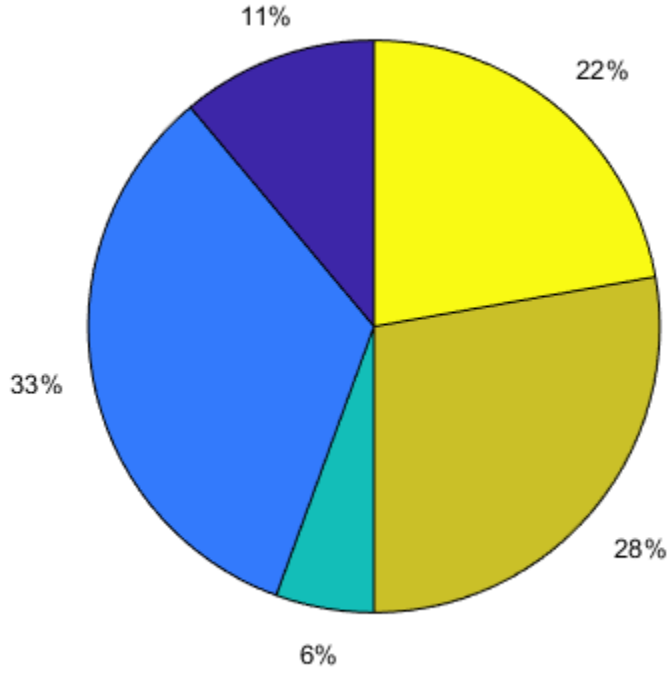
`p = pie( ___ )` returns a vector of patch and text graphics objects. The input can be any of the input argument combinations in the previous syntaxes.

Examples

Create Pie Chart with Offset Slices

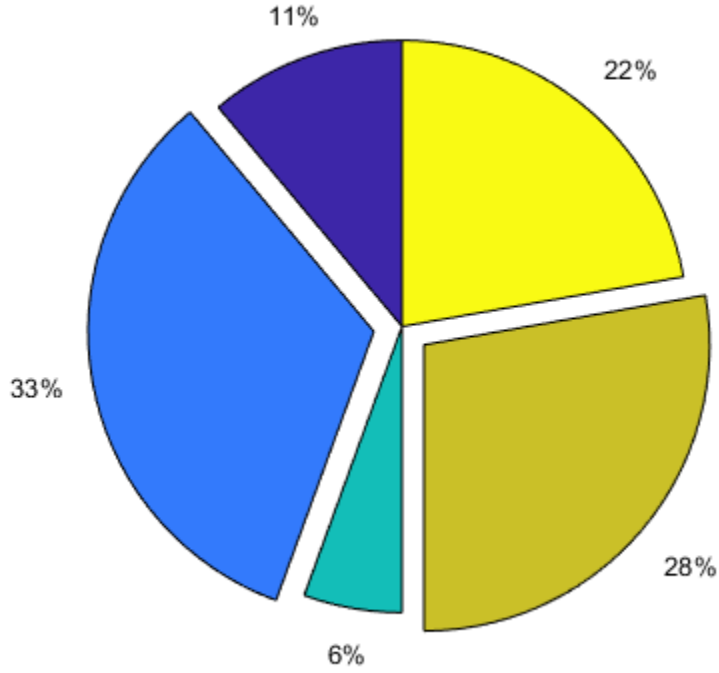
Create a pie chart of vector `X`.

```
X = [1 3 0.5 2.5 2];
pie(X)
```



Offset the second and fourth pie slices by setting the corresponding `explode` elements to 1.

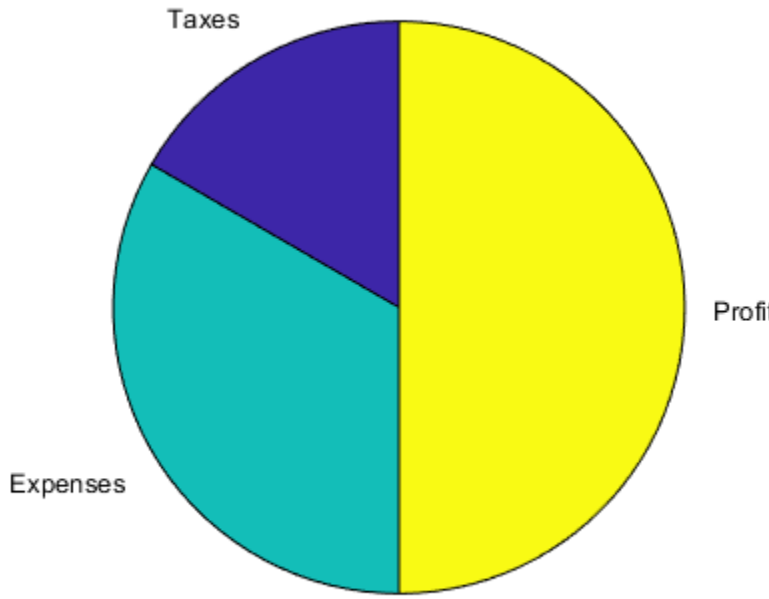
```
explode = [0 1 0 1 0];
pie(X,explode)
```



Specify Text Labels for Pie Chart

Create a pie chart of vector X and label the slices.

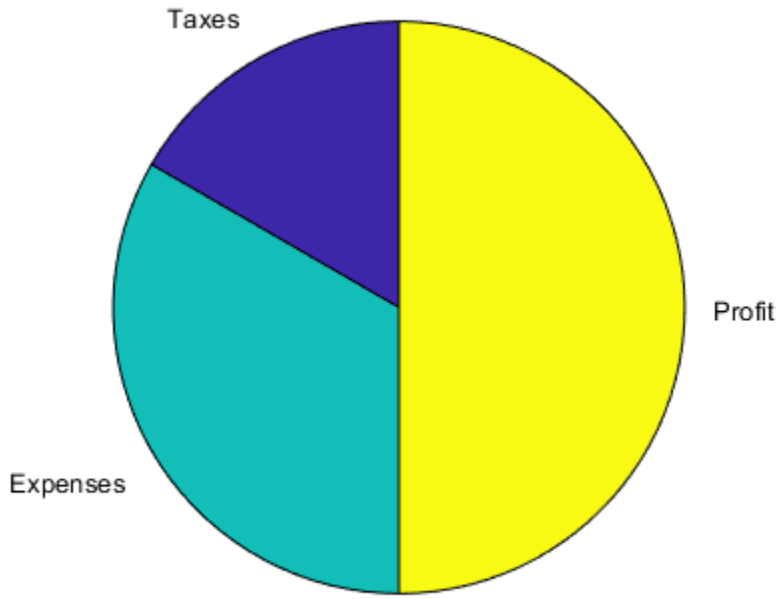
```
X = 1:3;
labels = {'Taxes','Expenses','Profit'};
pie(X,labels)
```



Modify Text Label for Pie Chart

Create a labeled pie chart, and then modify the color and font size of the text labels.

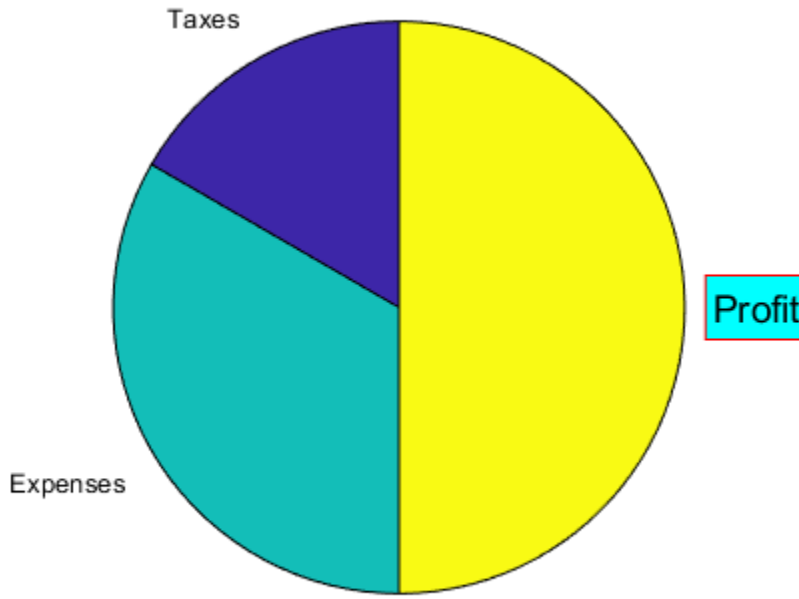
```
X = 1:3;
labels = {'Taxes','Expenses','Profit'};
p = pie(X,labels)
```




```
p =
  1x6 graphics array:

    Patch    Text     Patch    Text     Patch    Text
```

Get the text object for the label 'Profit'. Change its color and font size. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead.

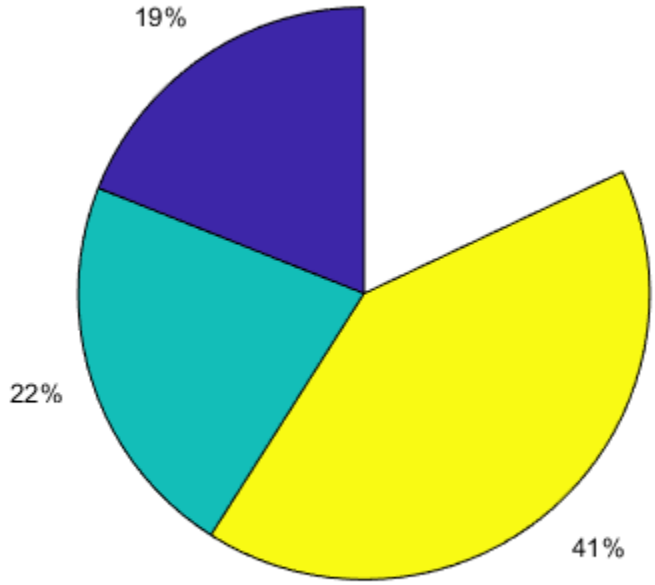
```
t = p(6);
t.BackgroundColor = 'cyan';
t.EdgeColor = 'red';
t.FontSize = 14;
```



Plot Partial Pie Chart

Create a pie chart of vector X where the sum of the elements is less than 1.

```
X = [0.19 0.22 0.41];
pie(X)
```



`pie` draws a partial pie because the sum of the elements is less than 1.

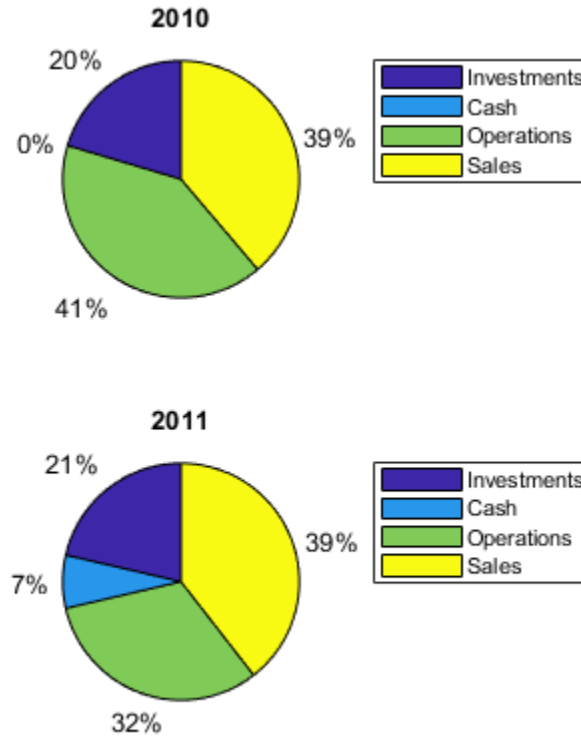
Compare Two Pie Charts

Create vectors `y2010` and `y2011`, that contain financial data for two years. Then create a cell array containing the labels for the values.

```
y2010 = [50 0 100 95];
y2011 = [65 22 97 120];
labels = {'Investments','Cash','Operations','Sales'};
```

Create a 2-by-1 tiled chart layout, and display a pie chart and legend for `y2010` data in the first tile. Then display a pie chart and legend for the `y2011` data in the second tile.

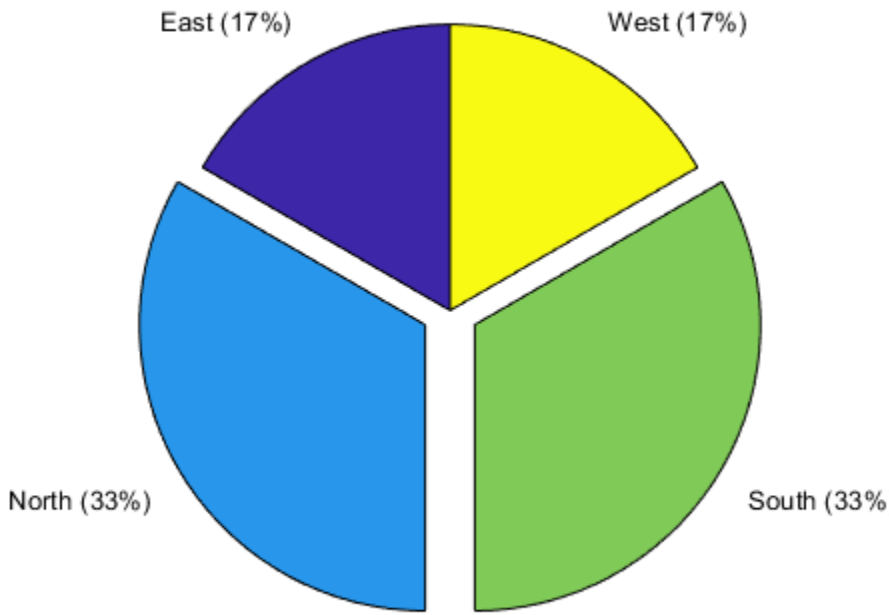
```
t = tiledlayout(2,1);
ax1 = nexttile;
pie(ax1,y2010)
legend(labels)
title('2010')

ax2 = nexttile;
pie(ax2,y2011)
legend(labels)
title('2011')
```



Plot Categorical Pie Chart with Offsets

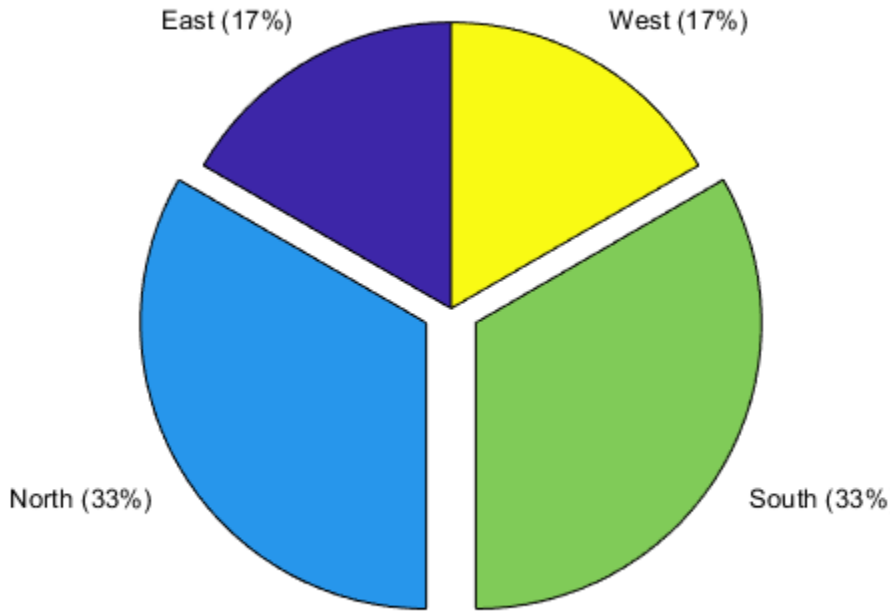
Plot a categorical pie chart with offset slices corresponding to categories.

```
X = categorical({'North','South','North','East','South','West'});
explode = {'North','South'};
pie(X,explode)
```



Now, use a logical vector to offset the same slices.

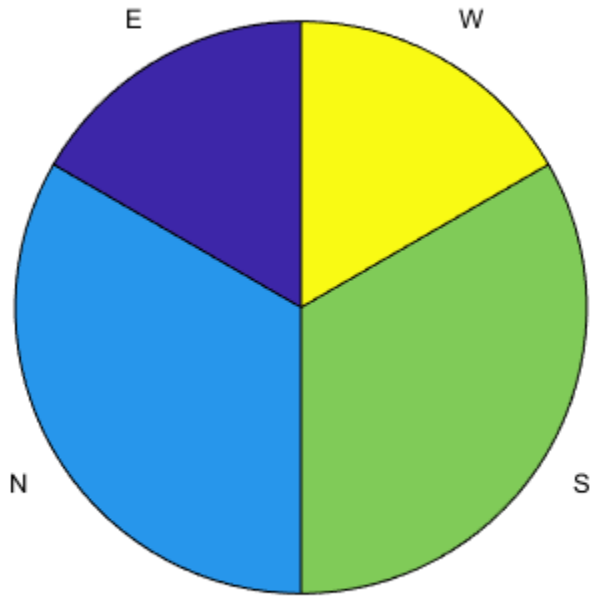
```
explode = [0 1 1 0];
pie(X,explode)
```



Plot Categorical Pie Chart with Labels

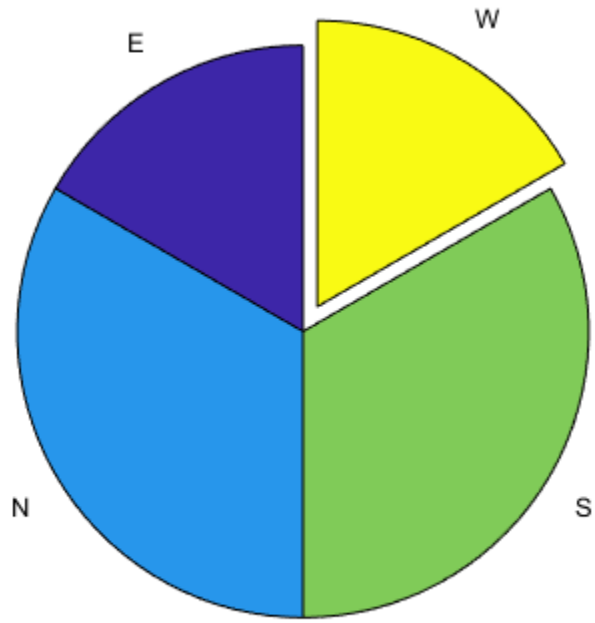
Plot a categorical pie chart without any offset slices and label the slices. When `X` is of data type `categorical` you must specify the input argument `explode`. To specify labels without any offset slices, specify `explode` as an empty cell array, and `labels` as the labels.

```
X = categorical({'North','South','North','East','South','West'});
explode = {};
labels = {'E','N','S','W'};
pie(X,explode,labels)
```



Now, offset a slice and label all slices.

```
X = categorical({'North','South','North','East','South','West'});
explode = {'West'};
labels = {'E','N','S','W'};
pie(X,explode,labels)
```



Input Arguments

X — Input array
vector | matrix

Input vector or matrix.

- If X is numeric, then all values in X must be finite.
- If X is categorical, then `pie` ignores undefined elements.

Data Types: `double|categorical`

`explode` — Offset slices

numeric vector or matrix | cell array of character vectors | string array

Offset slices, specified as a numeric vector or matrix, a cell array of character vectors, or a string array.

- If `X` is numeric, then `explode` must be a logical or numeric vector or matrix of zeros and nonzeros that correspond to `X`. A true (nonzero) value offsets the corresponding slice from the center of the pie chart, so that `X(i,j)` is offset from the center if `explode(i,j)` is nonzero. `explode` must be the same size as `X`.
- If `X` is categorical, then `explode` can be a cell array of character vectors that are category names or a string array of category names. `pie` offsets slices corresponding to categories in `explode`.
- If `X` is categorical, then `explode` also can be a logical or numeric vector with elements that correspond to each category in `X`. The `pie` function offsets slices corresponding to true (nonzero) in category order.

`labels` — Text labels

cell array of character vectors | string array

Text labels for slices, specified as a cell array of character vectors or a string array.

`ax` — Axes

axes object

Axes object. Use `ax` to plot the pie chart in a specific axes instead of the current axes (`gca`).

Output Arguments

`p` — `Patch` and `Text` objects

vector

`Patch` and `Text` objects, returned as a vector. For more information, see Patch Properties and Text Properties.

Compatibility Considerations

Pie Charts Display Zero Values

Behavior changed in R2019b

When you call the `pie` function, and specify data that contains zero values, your pie chart shows the zero values and corresponding labels. If you call `pie` with an output argument, the output includes objects for each zero value.

In previous releases, `pie` omitted the zero values from the chart, and it did not return any objects that corresponded to those values. If you do not want to display zero values or return the corresponding objects, then remove the zeros from your data.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the following limitations:

`X` must be a tall categorical array.

For more information, see “Tall Arrays”.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`pie3`

Topics

“Offset Pie Slice with Greatest Contribution”
“Label Pie Chart With Text and Percentages”

Introduced before R2006a

pie3

3-D pie chart

Syntax

```
pie3(X)
pie3(X,explode)
pie3(...,labels)
pie3(axes_handle,...)
h = pie3(...)
```

Description

`pie3(X)` draws a three-dimensional pie chart using the data in `X`. Each element in `X` is represented as a slice in the pie chart.

- If `sum(X) ≤ 1`, then the values in `X` directly specify the area of the pie slices. `pie3` draws only a partial pie if `sum(X) < 1`.
- If the sum of the elements in `X` is greater than one, then `pie3` normalizes the values by `X/sum(X)` to determine the area of each slice of the pie.

`pie3(X,explode)` specifies whether to offset a slice from the center of the pie chart. `X(i,j)` is offset from the center of the pie chart if `explode(i,j)` is nonzero. `explode` must be the same size as `X`.

`pie3(...,labels)` specifies text labels for the slices. The number of labels must equal the number of elements in `X`.

`pie3(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

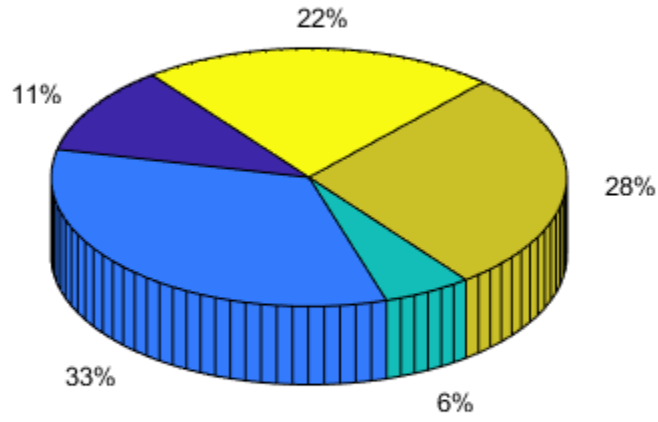
`h = pie3(...)` returns a vector of handles to patch, surface, and text graphics objects.

Examples

Create 3-D Pie Chart

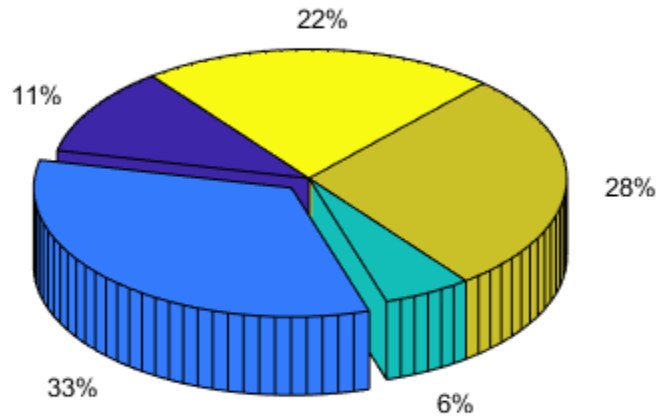
Create a 3-D pie chart of vector `x`.

```
x = [1,3,0.5,2.5,2];
figure
pie3(x)
```



To offset the second pie slice, set the corresponding `explode` element to 1.

```
explode = [0,1,0,0,0];
figure
pie3(x,explode)
```

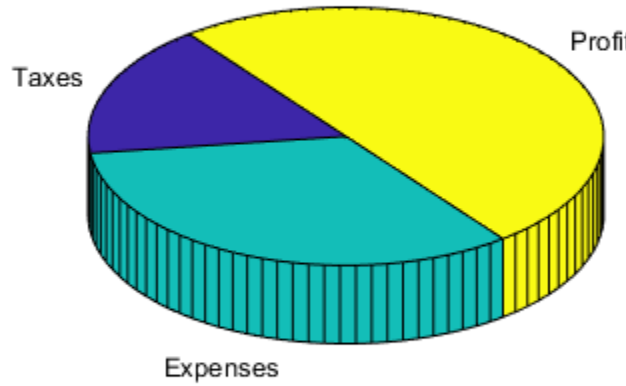


Specify Text Labels for 3-D Pie Chart

Create a 3-D pie chart and specify the text labels.

```
x = 1:3;
labels = {'Taxes','Expenses','Profit'};
```

```
figure
pie3(x,labels)
```



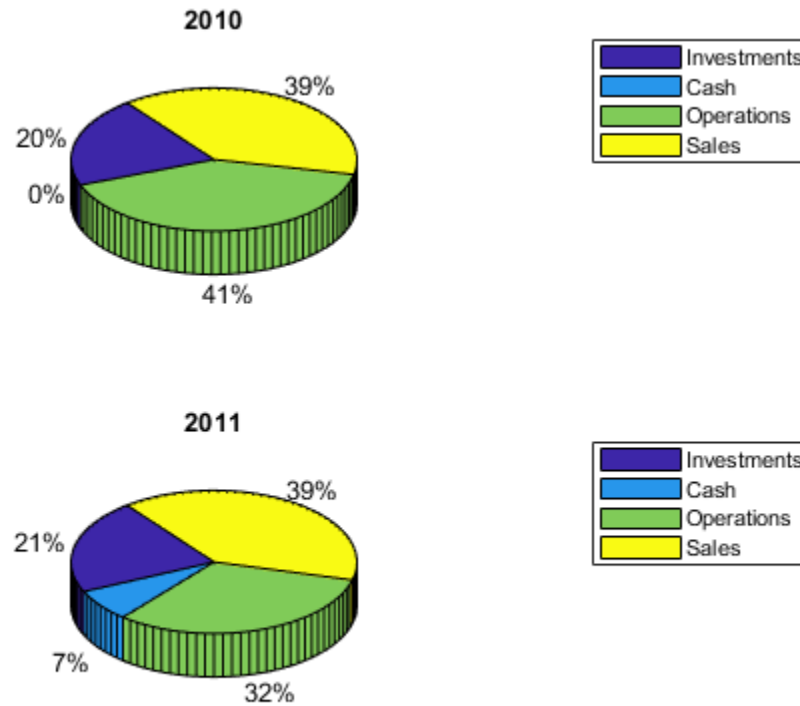
Compare Two Pie Charts

Create vectors `y2010` and `y2011`, that contain financial data for two years. Then create a cell array containing the labels for the values.

```
y2010 = [50 0 100 95];
y2011 = [65 22 97 120];
labels = {'Investments','Cash','Operations','Sales'};
```

Create a 2-by-1 tiled chart layout, and display a pie chart and legend for `y2010` data in the first tile. Then display a pie chart and legend for the `y2011` data in the second tile.

```
t = tiledlayout (2,1);
ax1 = nexttile;
pie3(ax1,y2010)
title('2010')
legend(labels)

ax2 = nexttile;
pie3(ax2,y2011)
title('2011')
legend(labels)
```



Compatibility Considerations

Pie Charts Display Zero Values

Behavior changed in R2019b

When you call the `pie3` function, and specify data that contains zero values, your pie chart shows the zero values and corresponding labels. If you call `pie3` with an output argument, the output includes objects for each zero value.

In previous releases, `pie3` omitted the zero values from the chart, and it did not return any objects that corresponded to those values. If you do not want to display zero values or return the corresponding objects, then remove the zeros from your data.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`pie`

Introduced before R2006a

pink

Pink colormap array

Syntax

```
c = pink
c = pink(m)
```

Description

`c = pink` returns the pink colormap as a three-column array with the same number of rows as the colormap for the current figure. If no figure exists, then the number of rows is equal to the default length of 256. Each row in the array contains the red, green, and blue intensities for a specific color. The intensities are in the range [0,1], and the color scheme looks like this image.

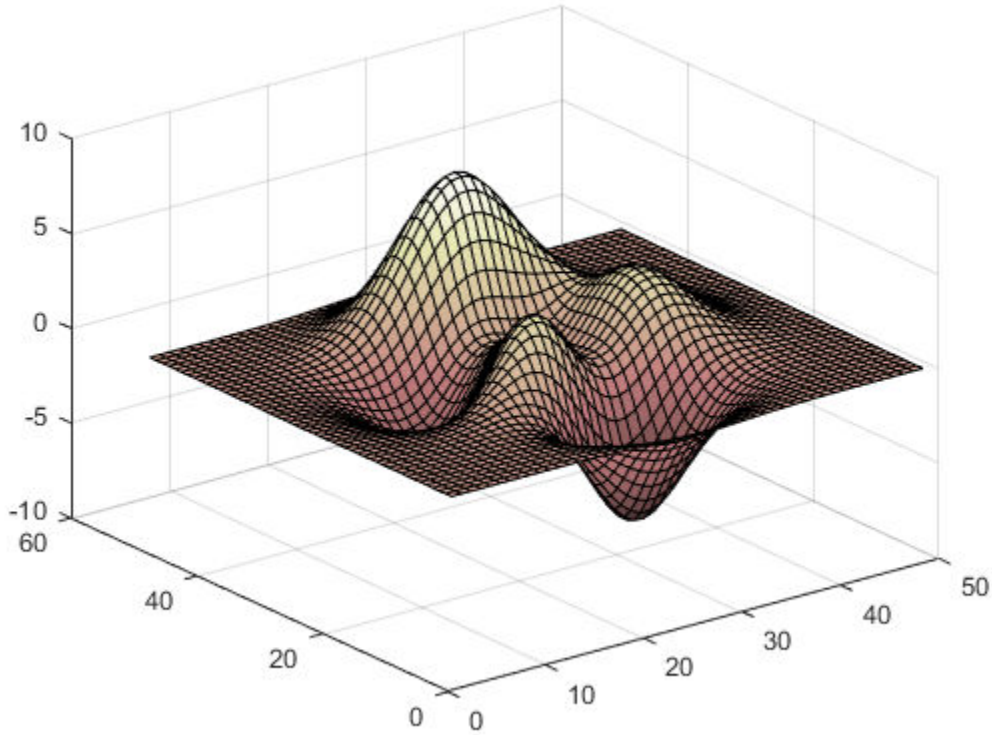
`c = pink(m)` returns the colormap with `m` colors.

Examples

Reverse the Colormap

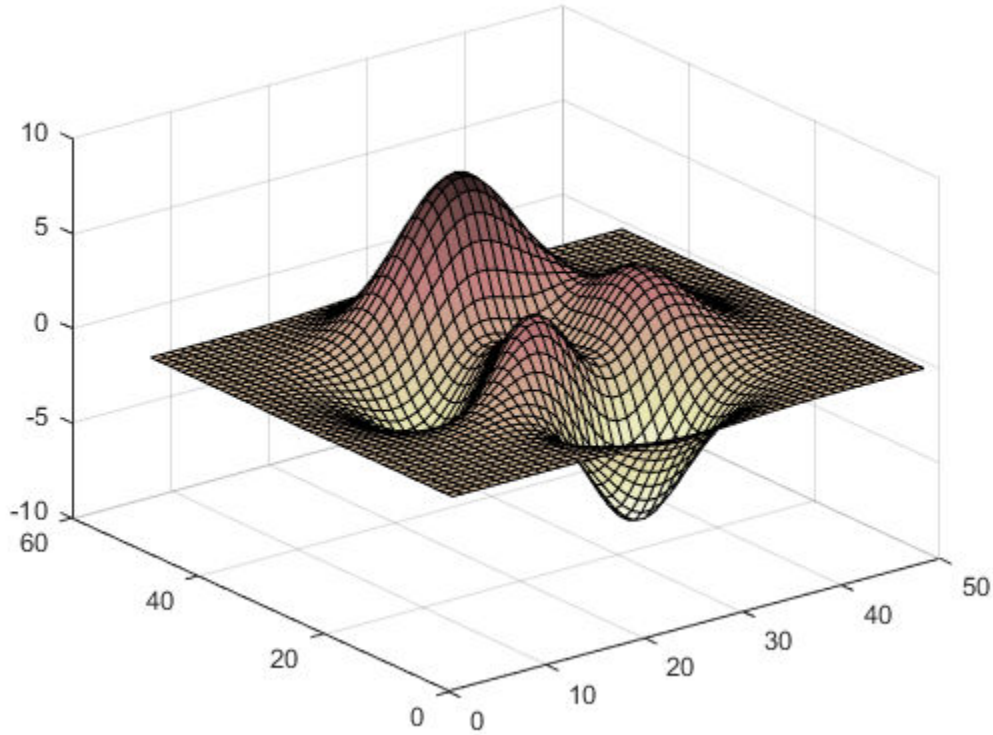
Plot a surface and assign the pink colormap.

```
surf(peaks);
colormap('pink');
```



Get the pink colormap array and reverse the order. Then apply the modified colormap to the surface.

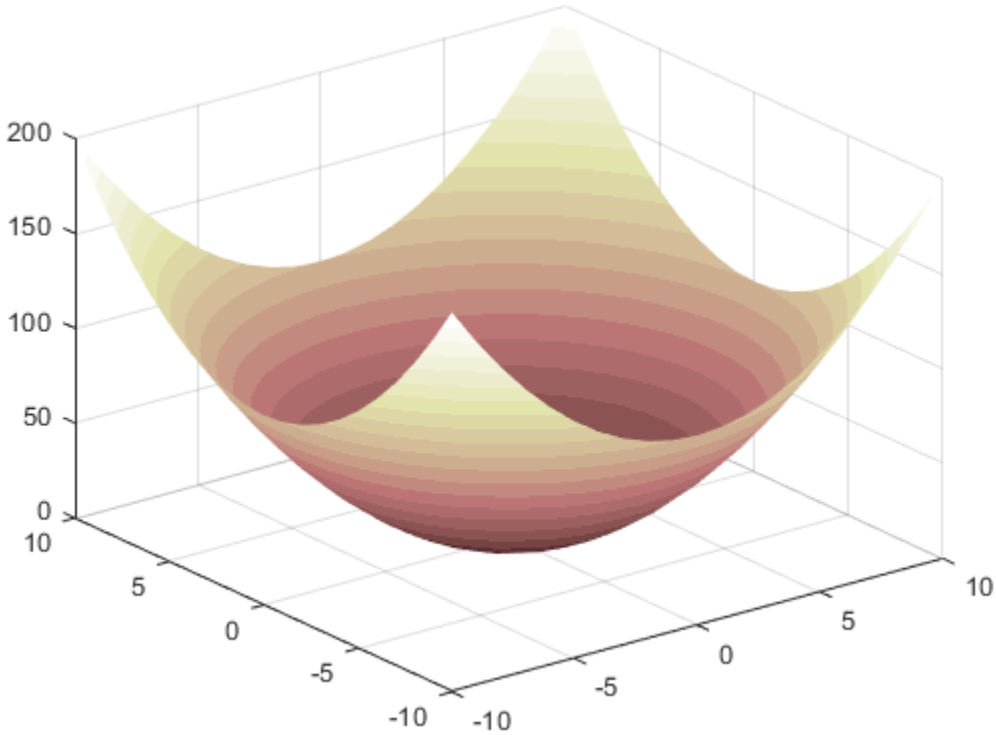
```
c = pink;
c = flipud(c);
colormap(c);
```



Downsample the Pink Colormap

Get a downsampled version of the pink colormap containing only twenty colors. Then display the contours of a paraboloid by applying the colormap and interpolated shading.

```
c = pink(20);
[X,Y] = meshgrid(-10:1:10);
Z = X.^2 + Y.^2;
surf(X,Y,Z);
colormap(c);
shading interp;
```



Input Arguments

`m` — Number of colors
256 (default) | scalar integer value

Number of colors, specified as a scalar integer value. The default value of `m` is equal to the length of the colormap for the current figure. If no figure exists, the default value is 256.

Data Types: `single` | `double`

Compatibility Considerations

Colormaps Have 256 Colors by Default

Behavior changed in R2019b

Starting in R2019b, colormaps have 256 colors by default.

In R2019a and previous releases, the default size is 64. If you have code that depends on a colormap having 64 colors, specify the number of colors when you set the colormap for the figure, axes, or chart. For example, `colormap(pink(64))` sets the figure's colormap to the 64-color `pink` colormap.

Alternatively, you can change the default colormap for all figures within your MATLAB session:

```
set(groot,'defaultFigureColormap',pink(64))
```

See Also

`colormap`

Topics

"Change Color Scheme Using a Colormap"

Introduced before R2006a

pinv

Moore-Penrose pseudoinverse

Syntax

```
B = pinv(A)
B = pinv(A,tol)
```

Description

`B = pinv(A)` returns the “Moore-Penrose Pseudoinverse” on page 1-11174 of matrix `A`.

`B = pinv(A,tol)` specifies a value for the tolerance. `pinv` treats singular values of `A` that are smaller than the tolerance as zero.

Examples

Solve System of Linear Equations Using Pseudoinverse

Compare solutions to a system of linear equations obtained by backslash (`\`) and `pinv`.

If a rectangular coefficient matrix `A` is of low rank, then the least-squares problem of minimizing `norm(A*x-b)` has infinitely many solutions. Two solutions are returned by `x1 = A\b` and `x2 = pinv(A)*b`. The distinguishing properties of these solutions are that `x1` has only `rank(A)` nonzero components, and `norm(x2)` is smaller than for any other solution.

Create an 8-by-6 matrix that has `rank(A) = 3`.

```
A = magic(8);
A = A(:,1:6)
```

```
A = 8×6
```

```
    64     2     3    61    60     6
     9    55    54    12    13    51
    17    47    46    20    21    43
    40    26    27    37    36    30
    32    34    35    29    28    38
    41    23    22    44    45    19
    49    15    14    52    53    11
     8    58    59     5     4    62
```

Create a vector for the right-hand side of the system of equations.

```
b = 260*ones(8,1)
```

```
b = 8×1

   260
   260
   260
   260
   260
   260
   260
   260
```

The number chosen for the right-hand side, 260, is the value of the 8-by-8 magic sum for `A`. If A were still an 8-by-8 matrix, then one solution for x would be a vector of 1s. With only six columns, a solution exists since the equations are still consistent, but the solution is not all 1s. Since the matrix is of low rank, there are infinitely many solutions.

Solve for two of the solutions using backslash and `pinv`.

```
x1 = A\b
```

```
Warning: Rank deficient, rank = 3, tol =  1.882938e-13.
```

```
x1 = 6×1

    3.0000
    4.0000
         0
         0
    1.0000
         0
```

```
x2 = pinv(A)*b
```

```
x2 = 6×1

    1.1538
    1.4615
    1.3846
    1.3846
    1.4615
    1.1538
```

Both of these solutions are exact, in the sense that `norm(A*x1-b)` and `norm(A*x2-b)` are on the order of roundoff error. The solution `x1` is special because it has only three nonzero elements. The solution `x2` is special because `norm(x2)` is smaller than it is for any other solution, including `norm(x1)`.

```
norm(x1)
```

```
ans = 5.0990
```

```
norm(x2)
```

```
ans = 3.2817
```

Input Arguments

A — Input matrix
matrix

Input matrix.

Data Types: `single` | `double`
Complex Number Support: Yes

`tol` — Singular value tolerance
scalar

Singular value tolerance, specified as a scalar. `pinv` treats singular values that are smaller than `tol` as zeros during the computation of the pseudoinverse.

The default tolerance is `max(size(A))*eps(norm(A))`.

Example: `pinv(A,1e-4)`

More About

Moore-Penrose Pseudoinverse

The Moore-Penrose pseudoinverse is a matrix that can act as a partial replacement for the matrix inverse in cases where it does not exist. This matrix is frequently used to solve a system of linear equations when the system does not have a unique solution or has many solutions.

For any matrix `A`, the pseudoinverse `B` exists, is unique, and has the same dimensions as `A'`. If `A` is square and not singular, then `pinv(A)` is simply an expensive way to compute `inv(A)`. However, if `A` is not square, or is square and singular, then `inv(A)` does not exist. In these cases, `pinv(A)` has some (but not all) of the properties of `inv(A)`:

1. $ABA = A$
2. $BAB = B$
3. $(AB)^* = AB$ (AB Hermitian)
4. $(BA)^* = BA$ (BA Hermitian)

The pseudoinverse computation is based on `svd(A)`. The calculation treats singular values less than `tol` as zero.

Tips

- You can replace most uses of `pinv` applied to a vector `b`, as in `pinv(A)*b`, with `lsqminnorm(A,b)` to get the minimum-norm least-squares solution of a system of linear equations. `lsqminnorm` is generally more efficient than `pinv`, and it also supports sparse matrices.

Algorithms

`pinv` uses the singular value decomposition to form the pseudoinverse of `A`. Singular values along the diagonal of `S` that are smaller than `tol` are treated as zeros, and the representation of `A` becomes:

$$A = USV^* = [U_1 \;\; U_2]\begin{bmatrix} S_1 & 0 \\ 0 & 0 \end{bmatrix}[V_1 \;\; V_2]^*$$

$$A = U_1 S_1 V_1^* \; .$$

The pseudoinverse of A is then equal to:

$$B = V_1 S_1^{-1} U_1^* \; .$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`decomposition` | `inv` | `lsqminnorm` | `qr` | `rank` | `svd`

Introduced before R2006a

planerot

Givens plane rotation

Syntax

```
[G,y] = planerot(x)
```

Description

`[G,y] = planerot(x)` where `x` is a 2-component column vector, returns a 2-by-2 orthogonal matrix G so that `y = G*x` has `y(2) = 0`.

Examples

```
x = [3 4];
[G,y] = planerot(x')

G  =
    0.6000    0.8000
   -0.8000    0.6000

y =
     5
     0
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).
- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`qrdelete` | `qrinsert`

Introduced before R2006a

play

Play audio from `audioplayer` object

Syntax

```
play(playerObj)
play(playerObj,start)
play(playerObj,[start,stop])
```

Description

`play(playerObj)` plays the audio associated with `audioplayer` object `playerObj` from beginning to end.

`play(playerObj,start)` plays audio from the sample indicated by `start` to the end.

`play(playerObj,[start,stop])` plays audio from the sample indicated by `start` to the sample indicated by `stop`.

Examples

Play with and without Blocking

Play two audio samples with and without blocking using the `play` and `playblocking` methods.

Load data from example files `chirp.mat` and `gong.mat`.

```
chirpData = load('chirp.mat');
chirpObj = audioplayer(chirpData.y,chirpData.Fs);

gongData = load('gong.mat');
gongObj = audioplayer(gongData.y,gongData.Fs);
```

Play the samples with blocking, one after the other.

```
playblocking(chirpObj);
playblocking(gongObj);
```

Play without blocking. The audio can overlap.

```
play(chirpObj);
play(gongObj);
```

Starting Sample

Play audio from the example file `handel.mat` starting 4 seconds from the beginning.

```
load handel.mat;
playerObj = audioplayer(y,Fs);
start = playerObj.SampleRate * 4;

play(playerObj,start);
```

Sample Range

Play the first 3 seconds of audio from the example file `handel.mat`.

```
load handel.mat;
playerObj = audioplayer(y,Fs);
start = 1;
stop = playerObj.SampleRate * 3;

play(playerObj,[start,stop]);
```

See Also

`audioplayer` | `playblocking`

Topics

"Play Audio"

playblocking

Play audio from `audioplayer` object, hold control until playback completes

Syntax

```
playblocking(playerObj)
playblocking(playerObj,start)
playblocking(playerObj,[start,stop])
```

Description

`playblocking(playerObj)` plays the audio associated with `audioplayer` object `playerObj` from beginning to end. `playblocking` does not return control until playback completes.

`playblocking(playerObj,start)` plays audio from the sample indicated by `start` to the end.

`playblocking(playerObj,[start,stop])` plays audio from the sample indicated by `start` to the sample indicated by `stop`.

Examples

Play with and without Blocking

Play two audio samples with and without blocking using the `play` and `playblocking` methods.

Load data from example files `chirp.mat` and `gong.mat`.

```
chirpData = load('chirp.mat');
chirpObj = audioplayer(chirpData.y,chirpData.Fs);
```

```
gongData = load('gong.mat');
gongObj = audioplayer(gongData.y,gongData.Fs);
```

Play the samples with blocking, one after the other.

```
playblocking(chirpObj);
playblocking(gongObj);
```

Play without blocking. The audio can overlap.

```
play(chirpObj);
play(gongObj);
```

Starting Sample

Play audio from the example file `handel.mat` starting 4 seconds from the beginning.

```
load handel.mat;
playerObj = audioplayer(y,Fs);
start = playerObj.SampleRate * 4;

play(playerObj,start);
```

Sample Range

Play the first 3 seconds of audio from the example file `handel.mat`.

```
load handel.mat;
playerObj = audioplayer(y,Fs);
start = 1;
stop = playerObj.SampleRate * 3;

play(playerObj,[start,stop]);
```

See Also

`audioplayer` | `play`

Topics

"Play Audio"

pause

Pause playback or recording

Syntax

```
pause(audioObj)
```

Description

`pause(audioObj)` pauses the playback or recording.

Examples

Control Audio Recording and Playback

Initiate a recording session, pause the recorder, and then continue the session to finish recording a full audio clip.

Create an audio recording object and start recording using a microphone on your system.

```
recObj = audiorecorder;
record(recObj);
disp('Recording in progress now ...')
```

```
Recording in progress now ...
```

Pause the recording and check the status of the `audiorecorder` object. The `isrecording` function returns `1` when recording is progress and `0` when the recording is not in progress.

```
pause(recObj);
isrecording(recObj)
```

```
ans =
```

```
  logical

   0
```

Listen to the audio that has been recorded so far.

```
playerObj= getplayer(recObj);
play(playerObj);
```

Before proceeding with the recording session, check the status of the `audioplayer` object. The `isplaying` function returns `1` when playback is progress and `0` when the playback is not in progress.

```
isplaying(playerObj)
```

```
ans =

  logical

   0
```

Next, resume the recording and capture more audio in the `audiorecorder` object.

```
resume(recObj)
```

To finalize the audio clip, after you have recorded more material, stop the recording.

```
stop(recObj)
```

Get the full audio clip from the `audiorecorder` object and play it.

```
playerObj= getplayer(recObj)
play(playerObj)
```

Input Arguments

`audioObj` — Audio object
`audioplayer` object | `audiorecorder` object

Audio object, specified as an `audioplayer` or `audiorecorder` object.

- If `audioObj` is an `audioplayer` object, then the `pause` function pauses the audio playback that is in progress.

- If `audioObj` is an `audiorecorder` object, then the `pause` function pauses the audio recording that is in progress.

To create an `audioplayer` object, use the `audioplayer` function.

To create an `audiorecorder` object, use the `audiorecorder` function.

See Also

`audioplayer` | `audiorecorder` | `resume` | `stop`

Introduced in R2006a

resume

Resume playback or recording from paused state

Syntax

```
resume(audioObj)
```

Description

`resume(audioObj)` resumes the playback or recording from the paused state.

Examples

Control Audio Recording and Playback

Initiate a recording session, pause the recorder, and then continue the session to finish recording a full audio clip.

Create an audio recording object and start recording using a microphone on your system.

```
recObj = audiorecorder;
record(recObj);
disp('Recording in progress now ...')
```

```
Recording in progress now ...
```

Pause the recording and check the status of the `audiorecorder` object. The `isrecording` function returns `1` when recording is progress and `0` when the recording is not in progress.

```
pause(recObj);
isrecording(recObj)
```

```
ans =
```

```
  logical

   0
```

Listen to the audio that has been recorded so far.

```
playerObj= getplayer(recObj);
play(playerObj);
```

Before proceeding with the recording session, check the status of the `audioplayer` object. The `isplaying` function returns `1` when playback is progress and `0` when the playback is not in progress.

```
isplaying(playerObj)
```

```
ans =

  logical

   0
```

Next, resume the recording and capture more audio in the `audiorecorder` object.

```
resume(recObj)
```

To finalize the audio clip, after you have recorded more material, stop the recording.

```
stop(recObj)
```

Get the full audio clip from the `audiorecorder` object and play it.

```
playerObj= getplayer(recObj)
play(playerObj)
```

Input Arguments

`audioObj` — Audio object
`audioplayer` object | `audiorecorder` object

Audio object, specified as an `audioplayer` or `audiorecorder` object.

- If `audioObj` is an `audioplayer` object, then the `resume` function resumes the audio playback that was paused using the `pause` function.

- If `audioObj` is an `audiorecorder` object, then the `resume` function resumes the audio recording that was paused using the `pause` function.

To create an `audioplayer` object, use the `audioplayer` function.

To create an `audiorecorder` object, use the `audiorecorder` function.

See Also

`audioplayer` | `audiorecorder` | `pause` | `stop`

Introduced in R2006a

stop

Stop playback or recording

Syntax

`stop(audioObj)`

Description

`stop(audioObj)` stops audio playback or recording in progress.

Examples

Control Audio Recording and Playback

Initiate a recording session, pause the recorder, and then continue the session to finish recording a full audio clip.

Create an audio recording object and start recording using a microphone on your system.

```
recObj = audiorecorder;
record(recObj);
disp('Recording in progress now ...')
```

```
Recording in progress now ...
```

Pause the recording and check the status of the `audiorecorder` object. The `isrecording` function returns `1` when recording is progress and `0` when the recording is not in progress.

```
pause(recObj);
isrecording(recObj)
```

```
ans =
```

```
  logical

   0
```

Listen to the audio that has been recorded so far.

```
playerObj= getplayer(recObj);
play(playerObj);
```

Before proceeding with the recording session, check the status of the `audioplayer` object. The `isplaying` function returns `1` when playback is progress and `0` when the playback is not in progress.

```
isplaying(playerObj)
```

```
ans =

  logical

   0
```

Next, resume the recording and capture more audio in the `audiorecorder` object.

```
resume(recObj)
```

To finalize the audio clip, after you have recorded more material, stop the recording.

```
stop(recObj)
```

Get the full audio clip from the `audiorecorder` object and play it.

```
playerObj= getplayer(recObj)
play(playerObj)
```

Input Arguments

`audioObj` — Audio object
`audioplayer` object | `audiorecorder` object

Audio object, specified as an `audioplayer` or `audiorecorder` object.

- If `audioObj` is an `audioplayer` object, then the `stop` function stops the audio playback in progress.

- If `audioObj` is an `audiorecorder` object, then the `stop` function stops the audio recording in progress.

To create an `audioplayer` object, use the `audioplayer` function.

To create an `audiorecorder` object, use the `audiorecorder` function.

See Also

`audioplayer` | `audiorecorder` | `pause` | `resume`

Introduced in R2006a

isplaying

Determine if playback is in progress

Syntax

```
isplaying(playerObj)
```

Description

`isplaying(playerObj)` determines if playback is in progress for the `audioplayer` object specified in `playerObj`.

Examples

Control Audio Recording and Playback

Initiate a recording session, pause the recorder, and then continue the session to finish recording a full audio clip.

Create an audio recording object and start recording using a microphone on your system.

```
recObj = audiorecorder;
record(recObj);
disp('Recording in progress now ...')
```

```
Recording in progress now ...
```

Pause the recording and check the status of the `audiorecorder` object. The `isrecording` function returns `1` when recording is progress and `0` when the recording is not in progress.

```
pause(recObj);
isrecording(recObj)
```

```
ans =
```

```
  logical

   0
```

Listen to the audio that has been recorded so far.

```
playerObj= getplayer(recObj);
play(playerObj);
```

Before proceeding with the recording session, check the status of the `audioplayer` object. The `isplaying` function returns `1` when playback is progress and `0` when the playback is not in progress.

```
isplaying(playerObj)
```

```
ans =

  logical

   0
```

Next, resume the recording and capture more audio in the `audiorecorder` object.

```
resume(recObj)
```

To finalize the audio clip, after you have recorded more material, stop the recording.

```
stop(recObj)
```

Get the full audio clip from the `audiorecorder` object and play it.

```
playerObj= getplayer(recObj)
play(playerObj)
```

Input Arguments

`playerObj` — Audio player object
`audioplayer` object

Audio player object, specified as an `audioplayer` object.

To create an `audioplayer` object, use the `audioplayer` function.

See Also

`audioplayer`

Introduced in R2006a

play

Play audio from `audiorecorder` object

Syntax

```
player = play(recObj)
player = play(recObj, start)
player = play(recObj, [start stop])
```

Description

player = `play(`*recObj*`)` plays the audio associated with `audiorecorder` object *recObj* from beginning to end, and returns an `audioplayer` object.

player = `play(`*recObj*`, `*start*`)` plays audio from the sample indicated by *start* to the end.

player = `play(`*recObj*`, [`*start stop*`])` plays audio from the sample indicated by *start* to the sample indicated by *stop*.

Examples

Record 5 seconds of your speech with a microphone, and play it back. Display the properties of the `audioplayer` object.

```
myVoice = audiorecorder;

disp('Start speaking.');
recordblocking(myVoice, 5);
disp('End of recording. Playing back ...');

playerObj = play(myVoice);

disp('Properties of playerObj:');
get(playerObj)
```

Play back only the first 3 seconds of the speech recorded in the previous example:

```
play(myVoice, [1 myVoice.SampleRate*3]);
```

See Also

`audioplayer` | `audiorecorder`

isrecording

Determine if recording is in progress

Syntax

```
isrecording(recorder)
```

Description

`isrecording(recorder)` determines if recording is in progress for the `audiorecorder` object specified in `recorder`.

Examples

Control Audio Recording and Playback

Initiate a recording session, pause the recorder, and then continue the session to finish recording a full audio clip.

Create an audio recording object and start recording using a microphone on your system.

```
recObj = audiorecorder;
record(recObj);
disp('Recording in progress now ...')
```

```
Recording in progress now ...
```

Pause the recording and check the status of the `audiorecorder` object. The `isrecording` function returns `1` when recording is progress and `0` when the recording is not in progress.

```
pause(recObj);
isrecording(recObj)
```

```
ans =
```

```
  logical

   0
```

Listen to the audio that has been recorded so far.

```
playerObj= getplayer(recObj);
play(playerObj);
```

Before proceeding with the recording session, check the status of the `audioplayer` object. The `isplaying` function returns `1` when playback is progress and `0` when the playback is not in progress.

```
isplaying(playerObj)
```

```
ans =

  logical

   0
```

Next, resume the recording and capture more audio in the `audiorecorder` object.

```
resume(recObj)
```

To finalize the audio clip, after you have recorded more material, stop the recording.

```
stop(recObj)
```

Get the full audio clip from the `audiorecorder` object and play it.

```
playerObj= getplayer(recObj)
play(playerObj)
```

Input Arguments

`recorder` — Audio recorder object
`audiorecorder` object

Audio recorder object, specified as an `audiorecorder` object. Use the `audiorecorder` function to create the object.

See Also

`audiorecorder`

Introduced in R2006a

getplayer

Creates associated `audioplayer` object

Syntax

```
playerObject = getplayer(recorder)
```

Description

`playerObject = getplayer(recorder)` creates the `audioplayer` object associated with the specified `audiorecorder` object.

Examples

Control Audio Recording and Playback

Initiate a recording session, pause the recorder, and then continue the session to finish recording a full audio clip.

Create an audio recording object and start recording using a microphone on your system.

```
recObj = audiorecorder;
record(recObj);
disp('Recording in progress now ...')
```

```
Recording in progress now ...
```

Pause the recording and check the status of the `audiorecorder` object. The `isrecording` function returns `1` when recording is progress and `0` when the recording is not in progress.

```
pause(recObj);
isrecording(recObj)
```

```
ans =
```

```
  logical

   0
```

Listen to the audio that has been recorded so far.

```
playerObj= getplayer(recObj);
play(playerObj);
```

Before proceeding with the recording session, check the status of the `audioplayer` object. The `isplaying` function returns `1` when playback is progress and `0` when the playback is not in progress.

```
isplaying(playerObj)
```

```
ans =

  logical

   0
```

Next, resume the recording and capture more audio in the `audiorecorder` object.

```
resume(recObj)
```

To finalize the audio clip, after you have recorded more material, stop the recording.

```
stop(recObj)
```

Get the full audio clip from the `audiorecorder` object and play it.

```
playerObj= getplayer(recObj)
play(playerObj)
```

Input Arguments

`recorder` — Audio recorder object
`audiorecorder` object

Audio recorder object, specified as an `audiorecorder` object. Use the `audiorecorder` function to create the object.

See Also

`audioplayer` | `audiorecorder`

Introduced in R2006a

plot

2-D line plot

Syntax

```
plot(X,Y)
plot(X,Y,LineSpec)
plot(X1,Y1,...,Xn,Yn)
plot(X1,Y1,LineSpec1,...,Xn,Yn,LineSpecn)
```

```
plot(Y)
plot(Y,LineSpec)
```

```
plot( ___ ,Name,Value)
plot(ax, ___ )
```

```
h = plot( ___ )
```

Description

`plot(X,Y)` creates a 2-D line plot of the data in `Y` versus the corresponding values in `X`.

- If `X` and `Y` are both vectors, then they must have equal length. The `plot` function plots `Y` versus `X`.
- If `X` and `Y` are both matrices, then they must have equal size. The `plot` function plots columns of `Y` versus columns of `X`.
- If one of `X` or `Y` is a vector and the other is a matrix, then the matrix must have dimensions such that one of its dimensions equals the vector length. If the number of matrix rows equals the vector length, then the `plot` function plots each matrix column versus the vector. If the number of matrix columns equals the vector length, then the function plots each matrix row versus the vector. If the matrix is square, then the function plots each column versus the vector.
- If one of `X` or `Y` is a scalar and the other is either a scalar or a vector, then the `plot` function plots discrete points. However, to see the points you must specify a marker symbol, for example, `plot(X,Y,'o')`.

`plot(X,Y,LineSpec)` sets the line style, marker symbol, and color.

`plot(X1,Y1,...,Xn,Yn)` plots multiple `X`, `Y` pairs using the same axes for all lines.

`plot(X1,Y1,LineSpec1,...,Xn,Yn,LineSpecn)` sets the line style, marker type, and color for each line. You can mix `X`, `Y`, `LineSpec` triplets with `X`, `Y` pairs. For example, `plot(X1,Y1,X2,Y2,LineSpec2,X3,Y3)`.

`plot(Y)` creates a 2-D line plot of the data in `Y` versus the index of each value.

- If `Y` is a vector, then the *x*-axis scale ranges from 1 to `length(Y)`.
- If `Y` is a matrix, then the `plot` function plots the columns of `Y` versus their row number. The *x*-axis scale ranges from 1 to the number of rows in `Y`.
- If `Y` is complex, then the `plot` function plots the imaginary part of `Y` versus the real part of `Y`, such that `plot(Y)` is equivalent to `plot(real(Y),imag(Y))`.

`plot(Y,LineSpec)` sets the line style, marker symbol, and color.

`plot( ___ ,Name,Value)` specifies line properties using one or more `Name,Value` pair arguments. For a list of properties, see Line. Use this option with any of the input argument combinations in the previous syntaxes. Name-value pair settings apply to all the lines plotted.

`plot(ax, ___ )` creates the line in the axes specified by `ax` instead of in the current axes (`gca`). The option `ax` can precede any of the input argument combinations in the previous syntaxes.

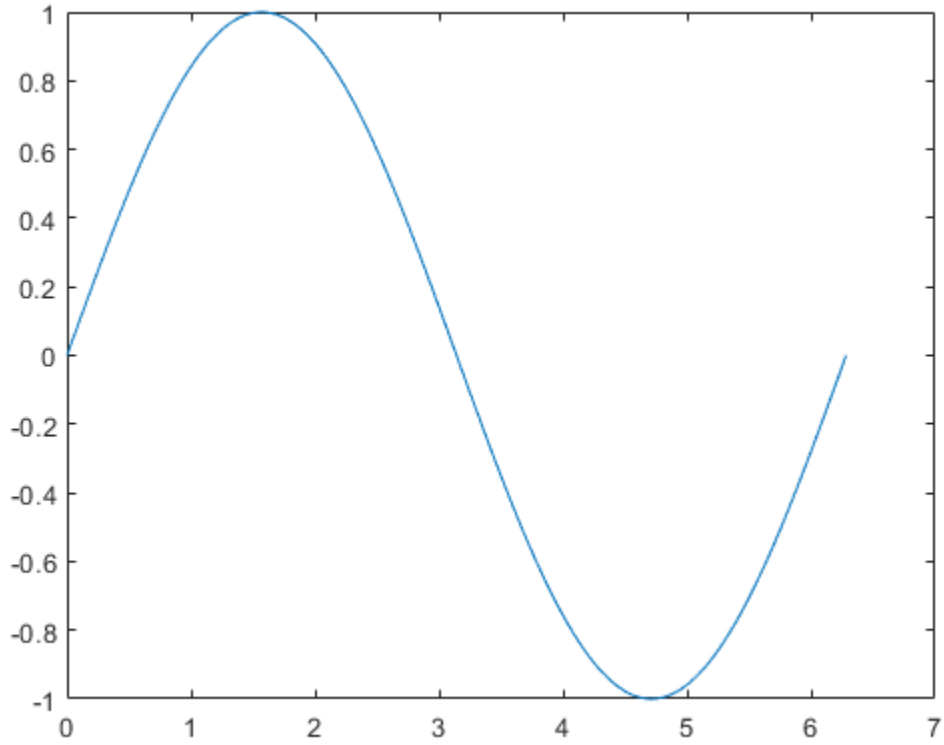
`h = plot( ___ )` returns a column vector of chart line objects. Use `h` to modify properties of a specific chart line after it is created. For a list of properties, see Line.

Examples

Create Line Plot

Create `x` as a vector of linearly spaced values between 0 and 2π. Use an increment of $\pi/100$ between the values. Create `y` as sine values of `x`. Create a line plot of the data.

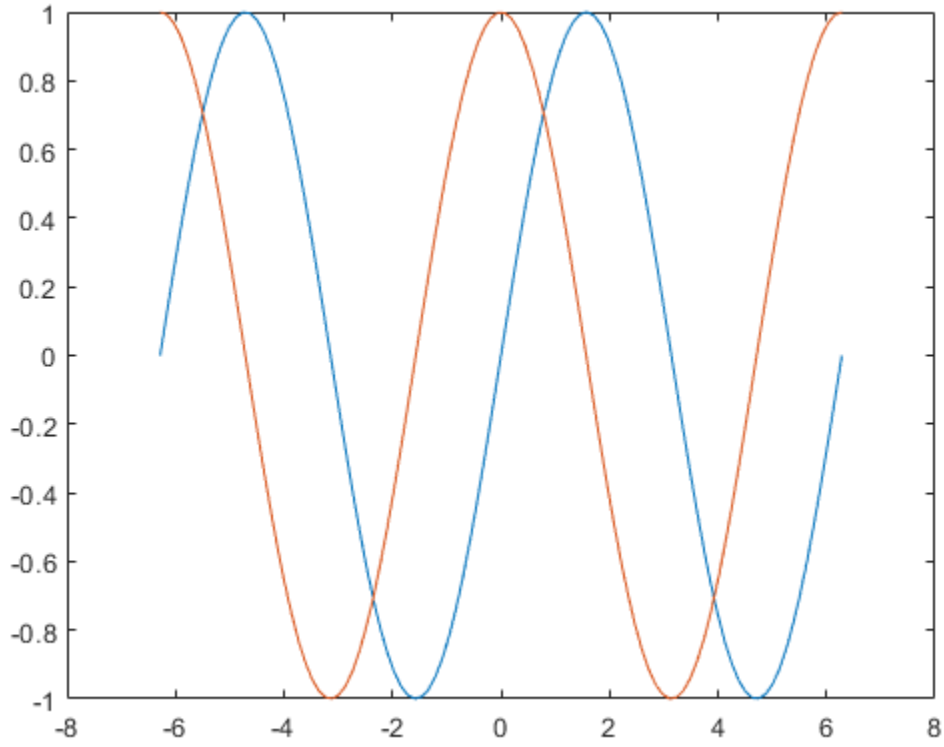
```
x = 0:pi/100:2*pi;
y = sin(x);
plot(x,y)
```



Plot Multiple Lines

Define `x` as 100 linearly spaced values between -2π and 2π. Define `y1` and `y2` as sine and cosine values of `x`. Create a line plot of both sets of data.

```
x = linspace(-2*pi,2*pi);
y1 = sin(x);
y2 = cos(x);

figure
plot(x,y1,x,y2)
```



Create Line Plot From Matrix

Define `Y` as the 4-by-4 matrix returned by the `magic` function.

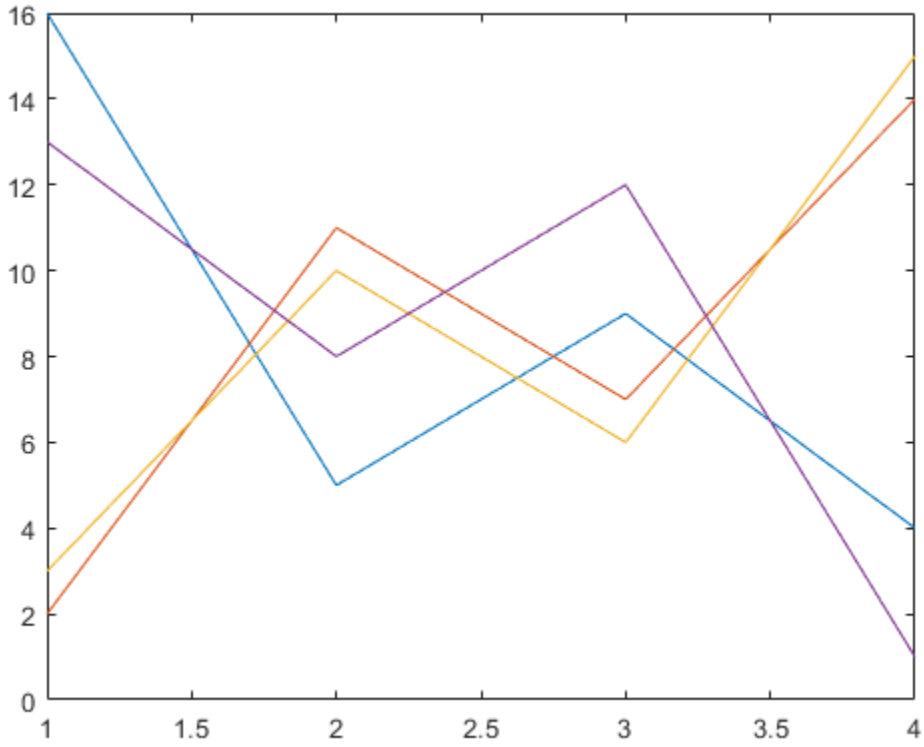
```
Y = magic(4)
```

```
Y = 4×4

    16     2     3    13
     5    11    10     8
     9     7     6    12
```

```
     4    14    15     1
```

Create a 2-D line plot of Y. MATLAB® plots each matrix column as a separate line.

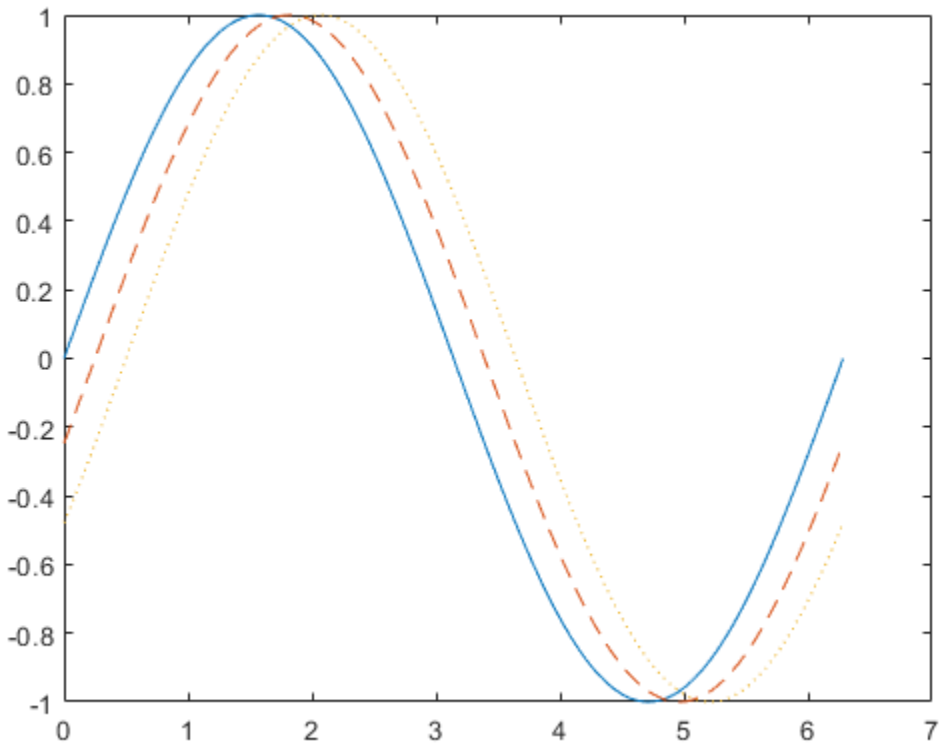
```
figure
plot(Y)
```



Specify Line Style

Plot three sine curves with a small phase shift between each line. Use the default line style for the first line. Specify a dashed line style for the second line and a dotted line style for the third line.

```
x = 0:pi/100:2*pi;
y1 = sin(x);
y2 = sin(x-0.25);
y3 = sin(x-0.5);

figure
plot(x,y1,x,y2,'--',x,y3,':')
```

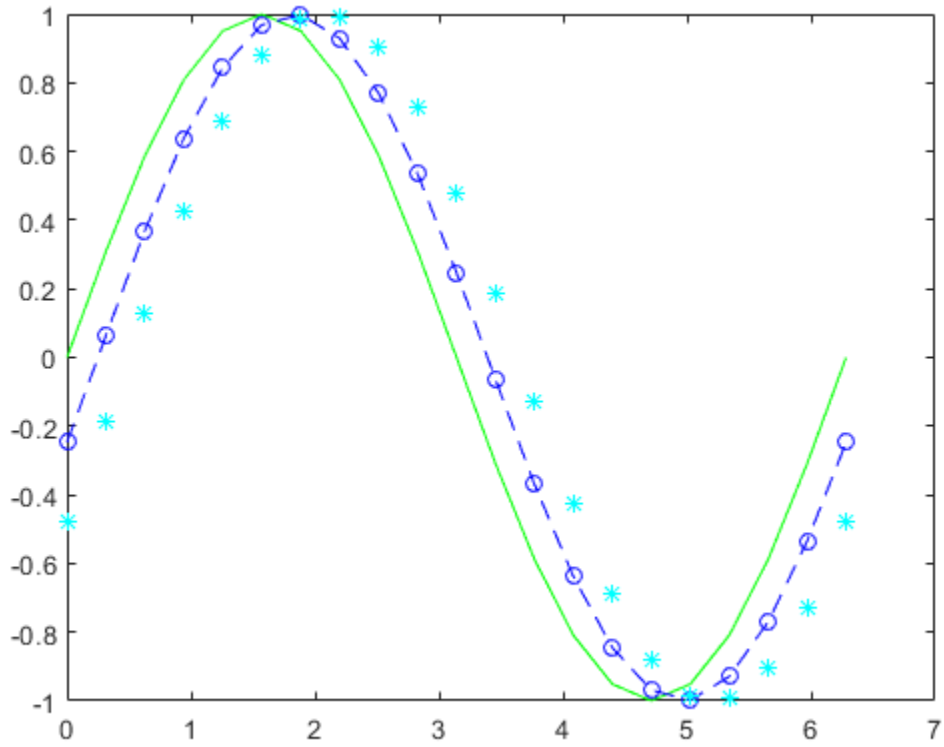


MATLAB® cycles the line color through the default color order.

Specify Line Style, Color, and Marker

Plot three sine curves with a small phase shift between each line. Use a green line with no markers for the first sine curve. Use a blue dashed line with circle markers for the second sine curve. Use only cyan star markers for the third sine curve.

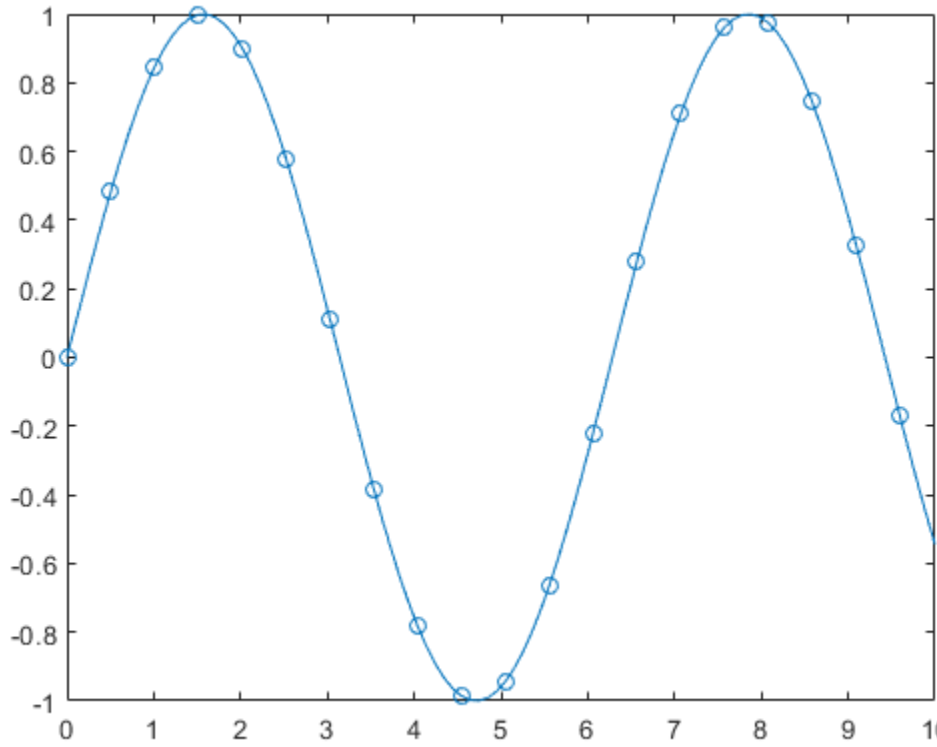
```
x = 0:pi/10:2*pi;
y1 = sin(x);
y2 = sin(x-0.25);
y3 = sin(x-0.5);

figure
plot(x,y1,'g',x,y2,'b--o',x,y3,'c*')
```



Display Markers at Specific Data Points

Create a line plot and display markers at every fifth data point by specifying a marker symbol and setting the `MarkerIndices` property as a name-value pair.

```
x = linspace(0,10);
y = sin(x);
plot(x,y,'-o','MarkerIndices',1:5:length(y))
```

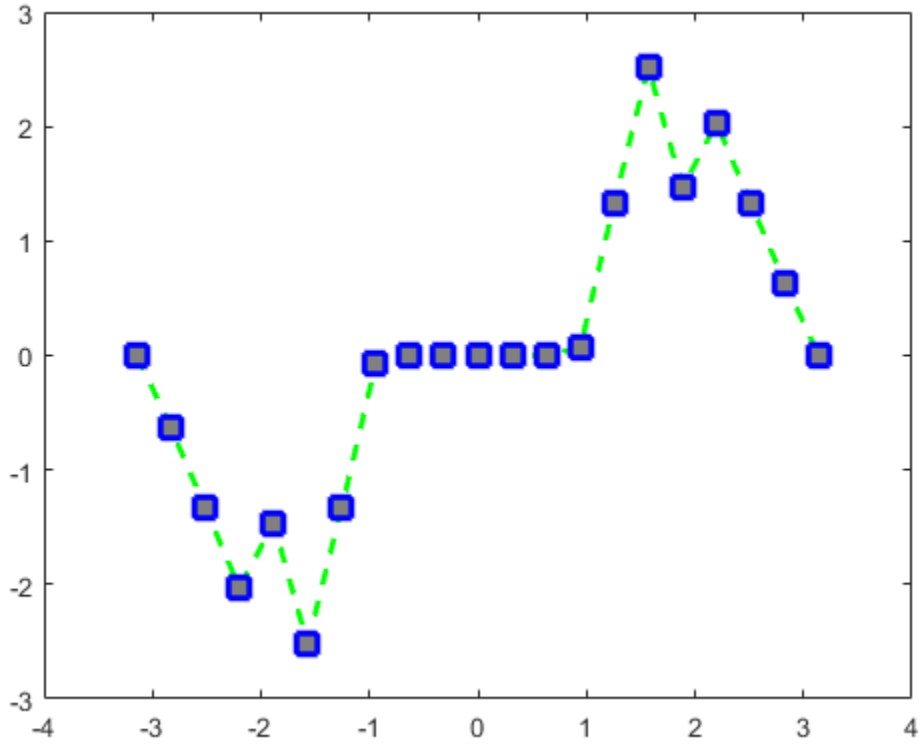


Specify Line Width, Marker Size, and Marker Color

Create a line plot and use the `LineSpec` option to specify a dashed green line with square markers. Use `Name,Value` pairs to specify the line width, marker size, and marker colors. Set the marker edge color to blue and set the marker face color using an RGB color value.

```
x = -pi:pi/10:pi;
y = tan(sin(x)) - sin(tan(x));

figure
```

```
plot(x,y,'--gs',...
    'LineWidth',2,...
    'MarkerSize',10,...
    'MarkerEdgeColor','b',...
    'MarkerFaceColor',[0.5,0.5,0.5])
```



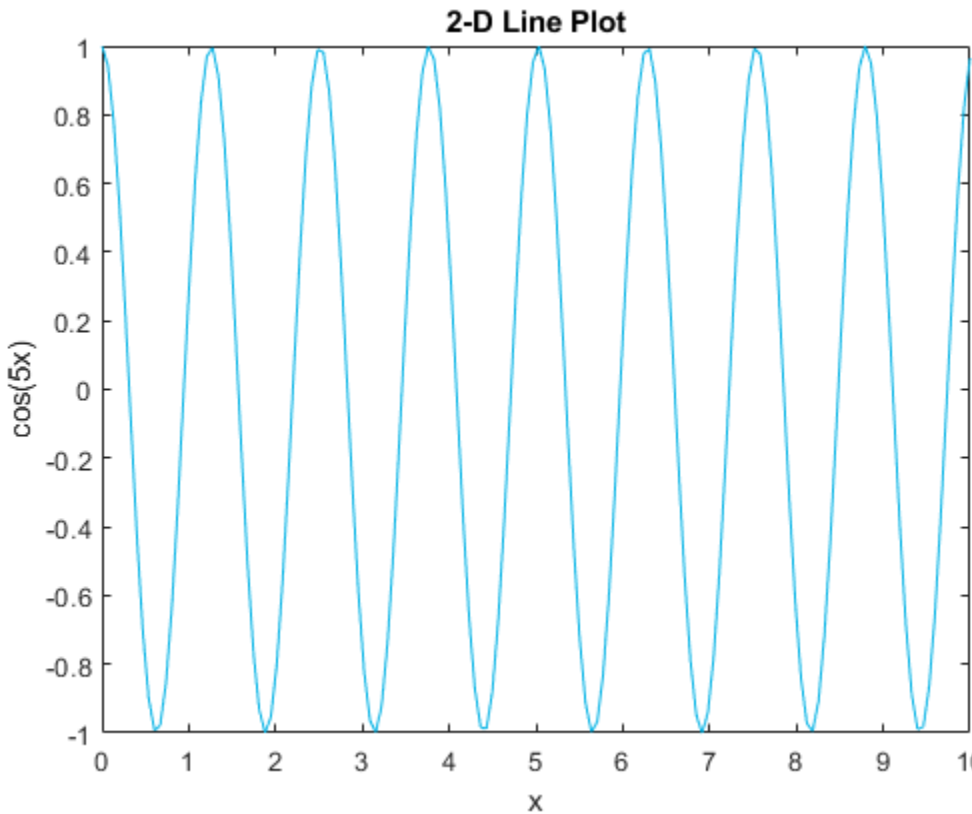
Add Title and Axis Labels

Use the `linspace` function to define `x` as a vector of 150 values between 0 and 10. Define `y` as cosine values of `x`.

```
x = linspace(0,10,150);
y = cos(5*x);
```

Create a 2-D line plot of the cosine curve. Change the line color to a shade of blue-green using an RGB color value. Add a title and axis labels to the graph using the `title`, `xlabel`, and `ylabel` functions.

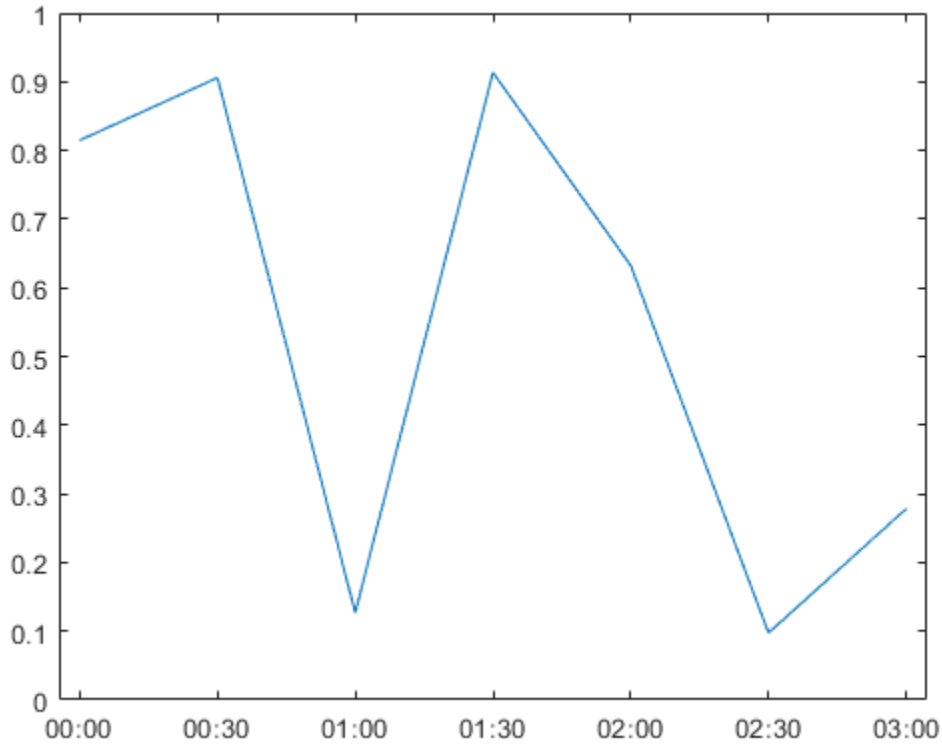
```
figure
plot(x,y,'Color',[0,0.7,0.9])

title('2-D Line Plot')
xlabel('x')
ylabel('cos(5x)')
```



Plot Durations and Specify Tick Format

Define `t` as seven linearly spaced `duration` values between 0 and 3 minutes. Plot random data and specify the format of the `duration` tick marks using the `'DurationTickFormat'` name-value pair argument.

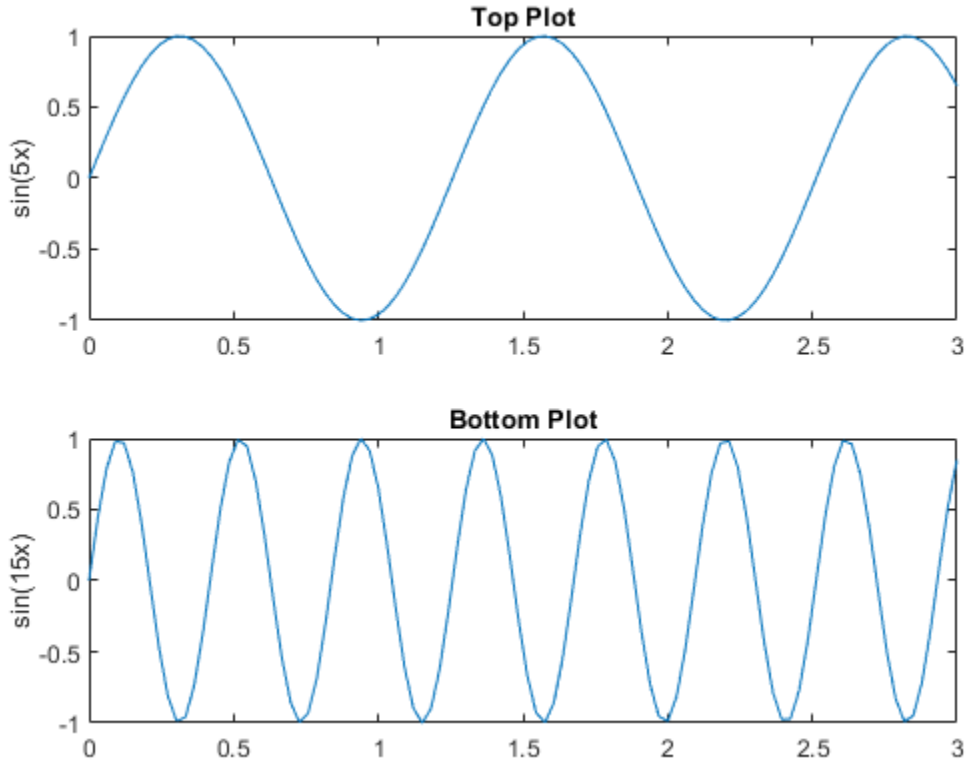
```
t = 0:seconds(30):minutes(3);
y = rand(1,7);

plot(t,y,'DurationTickFormat','mm:ss')
```



Specify Axes for Line Plot

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create an axes object and return the object as `ax1`. Create the top plot by passing `ax1` to the `plot` function. Add a title and *y*-axis label to the plot by passing the axes to the `title` and `ylabel` functions. Repeat the process to create the bottom plot.

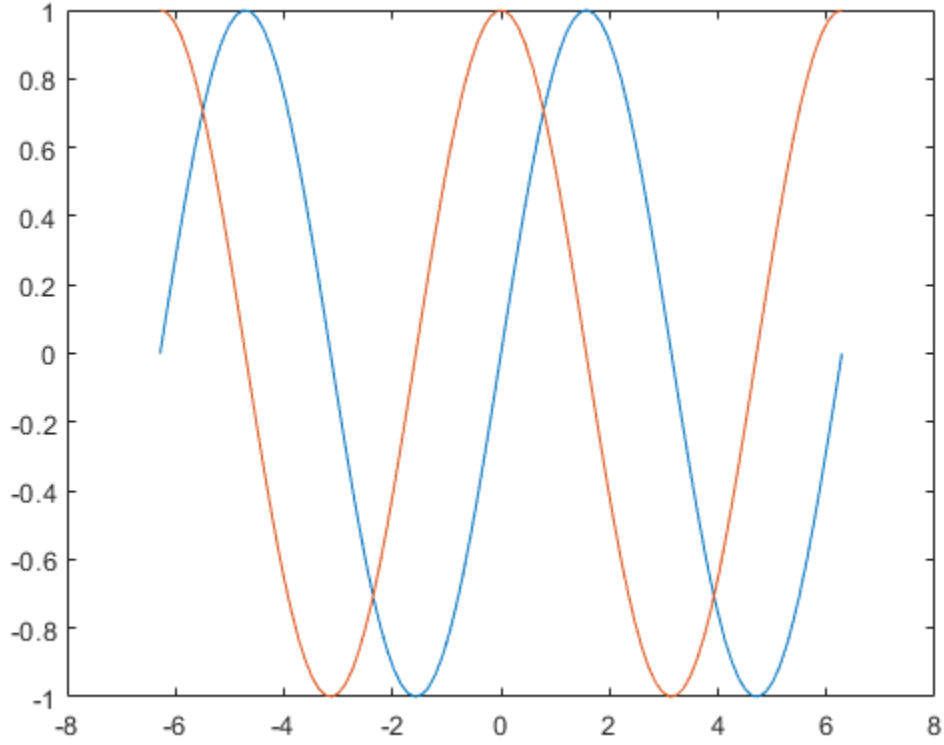
```
% Create data and 2-by-1 tiled chart layout
x = linspace(0,3);
y1 = sin(5*x);
y2 = sin(15*x);
tiledlayout(2,1)

% Top plot
ax1 = nexttile;
plot(ax1,x,y1)
title(ax1,'Top Plot')
ylabel(ax1,'sin(5x)')

% Bottom plot
ax2 = nexttile;
plot(ax2,x,y2)
title(ax2,'Bottom Plot')
ylabel(ax2,'sin(15x)')
```



Modify Lines After Creation

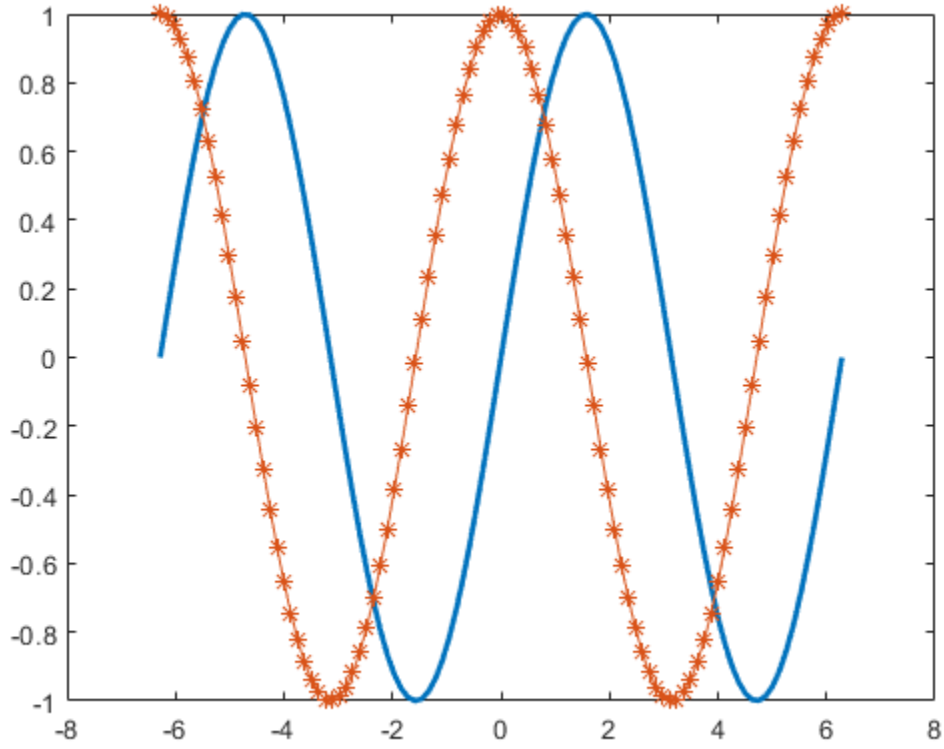
Define x as 100 linearly spaced values between -2π and 2π. Define y1 and y2 as sine and cosine values of x. Create a line plot of both sets of data and return the two chart lines in p.

```
x = linspace(-2*pi,2*pi);
y1 = sin(x);
y2 = cos(x);
p = plot(x,y1,x,y2);
```



Change the line width of the first line to 2. Add star markers to the second line. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead.

```
p(1).LineWidth = 2;
p(2).Marker = '*';
```

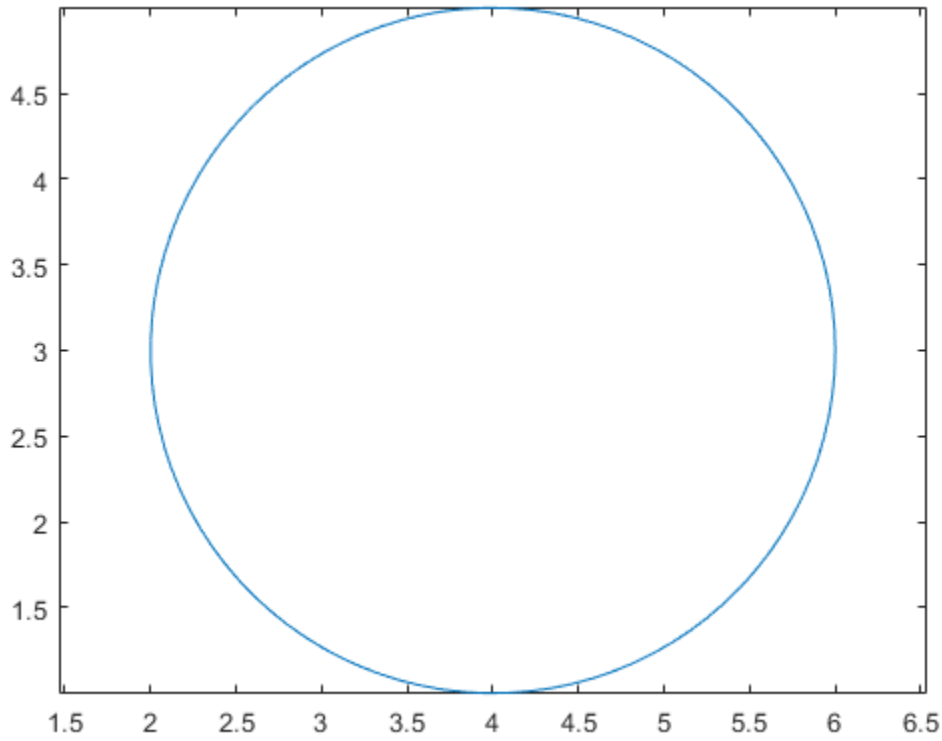


Plot Circle

Plot a circle centered at the point (4,3) with a radius equal to 2. Use `axis equal` to use equal data units along each coordinate direction.

```
r = 2;
xc = 4;
yc = 3;

theta = linspace(0,2*pi);
x = r*cos(theta) + xc;
```

```
y = r*sin(theta) + yc;
plot(x,y)
axis equal
```



Input Arguments

Y — y values
scalar | vector | matrix

y values, specified as a scalar, a vector, or a matrix. To plot against specific *x* values you must also specify X.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

X — x values
scalar | vector | matrix

x values, specified as a scalar, a vector, or a matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`LineSpec` — Line style, marker, and color
character vector | string

Line style, marker, and color, specified as a character vector or string containing symbols. The symbols can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the marker and no line.

Example: `'--or'` is a red dashed line with circle markers

Line Style	Description
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Marker	Description
`o`	Circle
`+`	Plus sign
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle

Marker	Description
v	Downward-pointing triangle
>	Right-pointing triangle
<	Left-pointing triangle
p	Pentagram
h	Hexagram

Color	Description
y	yellow
m	magenta
c	cyan
r	red
g	green
b	blue
w	white
k	black

ax — Target axes
`Axes` object | `PolarAxes` object | `GeographicAxes` object

Target axes, specified as an `Axes` object, a `PolarAxes` object, or a `GeographicAxes` object. If you do not specify the axes and if the current axes are Cartesian axes, then the `plot` function uses the current axes. To plot into polar axes, specify the `PolarAxes` object as the first input argument or use the `polarplot` function. To plot into a geographic axes, specify the `GeographicAxes` object as the first input argument or use the `geoplot` function.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The chart line properties listed here are only a subset. For a complete list, see Chart Line.

Example: `'Marker','o','MarkerFaceColor','red'`

`Color` — Line color
`[0 0.4470 0.7410]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

`Marker` — Marker symbol
`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | `'x'` | ...

Marker symbol, specified as one of the markers in this table. By default, a chart line does not have markers. Add markers at each data point along the line by specifying a marker symbol.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'Marker','+'`

Example: `'Marker','diamond'`

`MarkerIndices` — Indices of data points at which to display markers
`1:length(YData)` (default) | vector of positive integers | scalar positive integer

Indices of data points at which to display markers, specified as a vector of positive integers. If you do not specify the indices, then MATLAB displays a marker at every data point.

Note To see the markers, you must also specify a marker symbol.

Example: `plot(x,y,'-o','MarkerIndices',[1 5 10])` displays a circle marker at the first, fifth, and tenth data points.

Example: `plot(x,y,'-x','MarkerIndices',1:3:length(y))` displays a cross marker every three data points.

Example: `plot(x,y,'Marker','square','MarkerIndices',5)` displays one square marker at the fifth data point.

`MarkerEdgeColor` — Marker outline color

`'auto'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'auto'` uses the same color as the `Color` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` option uses the same color as the `Color` property of the parent axes. If you specify `'auto'` and the axes plot box is invisible, the marker fill color is the color of the figure.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`MarkerSize` — Marker size
6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

`DatetimeTickFormat` — Format for `datetime` tick labels
character vector | string

Format for `datetime` tick labels, specified as the comma-separated pair consisting of `'DatetimeTickFormat'` and a character vector or string containing a date format. Use

the letters A-Z and a-z to construct a custom format. These letters correspond to the Unicode Locale Data Markup Language (LDML) standard for dates. You can include non-ASCII letter characters such as a hyphen, space, or colon to separate the fields.

If you do not specify a value for 'DatetimeTickFormat', then plot automatically optimizes and updates the tick labels based on the axis limits.

Example: 'DatetimeTickFormat','eeee, MMMM d, yyyy HH:mm:ss' displays a date and time such as Saturday, April 19, 2014 21:41:06.

The following table shows several common display formats and examples of the formatted output for the date, Saturday, April 19, 2014 at 9:41:06 PM in New York City.

Value of DatetimeTickFormat	Example
'yyyy-MM-dd'	2014-04-19
'dd/MM/yyyy'	19/04/2014
'dd.MM.yyyy'	19.04.2014
'yyyy 年 MM 月 dd 日'	2014 年 04 月 19 日
'MMMM d, yyyy'	April 19, 2014
'eeee, MMMM d, yyyy HH:mm:ss'	Saturday, April 19, 2014 21:41:06
'MMMM d, yyyy HH:mm:ss Z'	April 19, 2014 21:41:06 -0400

For a complete list of valid letter identifiers, see the Format property for datetime arrays.

DatetimeTickFormat is not a chart line property. You must set the tick format using the name-value pair argument when creating a plot. Alternatively, set the format using the xtickformat and ytickformat functions.

The TickLabelFormat property of the datetime ruler stores the format.

DurationTickFormat — Format for duration tick labels
character vector | string

Format for duration tick labels, specified as the comma-separated pair consisting of 'DurationTickFormat' and a character vector or string containing a duration format.

If you do not specify a value for 'DurationTickFormat', then plot automatically optimizes and updates the tick labels based on the axis limits.

To display a duration as a single number that includes a fractional part, for example, 1.234 hours, specify one of the values in this table.

Value of `DurationTickFormat`	Description
`'y'`	Number of exact fixed-length years. A fixed-length year is equal to 365.2425 days.
`'d'`	Number of exact fixed-length days. A fixed-length day is equal to 24 hours.
`'h'`	Number of hours
`'m'`	Number of minutes
`'s'`	Number of seconds

Example: `'DurationTickFormat','h'` displays duration values in terms of fixed-length days.

To display a duration in the form of a digital timer, specify one of these values.

- `'dd:hh:mm:ss'`
- `'hh:mm:ss'`
- `'mm:ss'`
- `'hh:mm'`

In addition, you can display up to nine fractional second digits by appending up to nine `S` characters.

Example: `'DurationTickFormat','hh:mm:ss.SSS'` displays the milliseconds of a duration value to three digits.

`DurationTickFormat` is not a chart line property. You must set the tick format using the name-value pair argument when creating a plot. Alternatively, set the format using the `xtickformat` and `ytickformat` functions.

The `TickLabelFormat` property of the duration ruler stores the format.

Output Arguments

h — One or more chart line objects
scalar | vector

One or more chart line objects, returned as a scalar or a vector. These are unique identifiers, which you can use to query and modify properties of a specific chart line. For a list of properties, see Chart Line.

Tips

- Use `NaN` and `Inf` values to create breaks in the lines. For example, this code plots the first two elements, skips the third element, and draws another line using the last two elements:

  ```
  plot([1,2,NaN,4,5])
  ```

- `plot` uses colors and line styles based on the `ColorOrder` and `LineStyleOrder` properties of the axes. `plot` cycles through the colors with the first line style. Then, it cycles through the colors again with each additional line style.

  Starting in R2019b, you can change the colors and the line styles after plotting by setting the `ColorOrder` or `LineStyleOrder` properties on the axes. You can also call the `colororder` function to change the color order for all the axes in the figure.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- Supported syntaxes for tall arrays `X` and `Y` are:
  - `plot(X,Y)`
  - `plot(Y)`
  - `plot(___,LineSpec)`
  - `plot(___,Name,Value)`
  - `plot(ax,___)`
- `X` must be in monotonically increasing order.

- Categorical inputs are not supported.
- Tall inputs must be real column vectors.
- With tall arrays, the `plot` function plots in iterations, progressively adding to the plot as more data is read. During the updates, a progress indicator shows the proportion of data that has been plotted. Zooming and panning is supported during the updating process, before the plot is complete. To stop the update process, press the pause button in the progress indicator.

For more information, see “Visualization of Tall Arrays”.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions

`gca` | `hold` | `legend` | `loglog` | `plot3` | `title` | `xlabel` | `xlim` | `ylabel` | `ylim` | `yyaxis`

Properties
Chart Line

Topics
"Plot Dates and Durations"
"Plot Categorical Data"

External Websites
MATLAB Plot Gallery

Introduced before R2006a

plot3

3-D point or line plot

Syntax

```
plot3(X,Y,Z)
plot3(X,Y,Z,LineSpec)
plot3(X1,Y1,Z1,...,Xn,Yn,Zn)
plot3(X1,Y1,Z1,LineSpec1,...,Xn,Yn,Zn,LineSpecn)
plot3( ___ ,Name,Value)
plot3(ax, ___ )
p = plot3( ___ )
```

Description

`plot3(X,Y,Z)` plots coordinates in 3-D space.

- To plot a set of coordinates connected by line segments, specify `X`, `Y`, and `Z` as vectors of the same length.
- To plot multiple sets of coordinates on the same set of axes, specify at least one of `X`, `Y`, or `Z` as a matrix and the others as vectors.

`plot3(X,Y,Z,LineSpec)` creates the plot using the specified line style, marker, and color.

`plot3(X1,Y1,Z1,...,Xn,Yn,Zn)` plots multiple sets of coordinates on the same set of axes. Use this syntax as an alternative to specifying multiple sets as matrices.

`plot3(X1,Y1,Z1,LineSpec1,...,Xn,Yn,Zn,LineSpecn)` assigns specific line styles, markers, and colors to each `XYZ` triplet. You can specify `LineSpec` for some triplets and omit it for others. For example, `plot3(X1,Y1,Z1,'o',X2,Y2,Z2)` specifies markers for the first triplet but not the for the second triplet.

`plot3( ___ ,Name,Value)` specifies `Line` properties using one or more name-value pair arguments. Specify the properties after all other input arguments. For a list of properties, see Line.

`plot3(ax, ___ )` displays the plot in the target axes. Specify the axes as the first argument in any of the previous syntaxes.

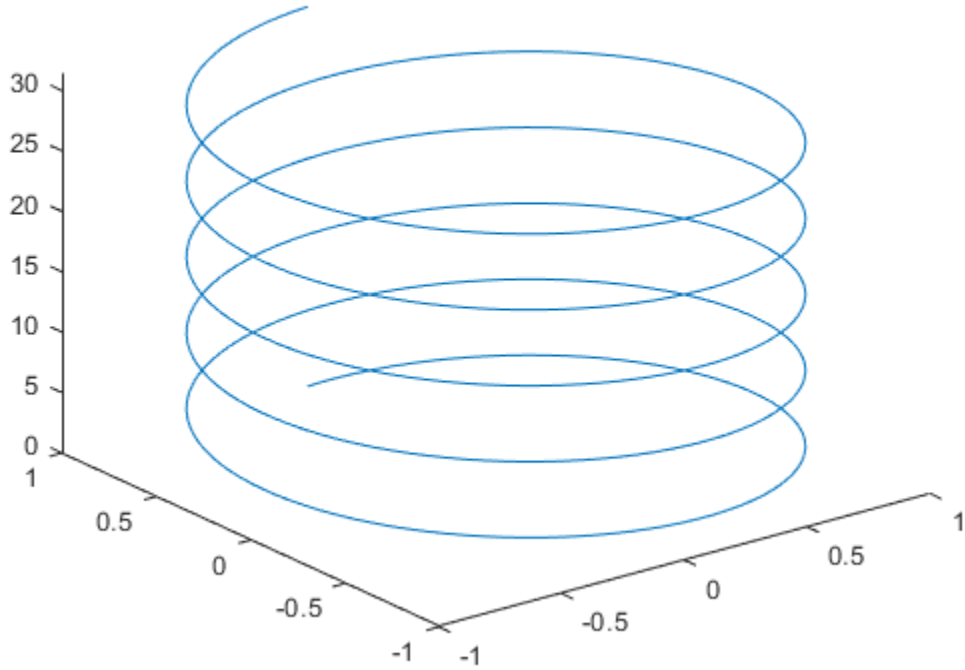
`p = plot3( ___ )` returns a `Line` object or an array of `Line` objects. Use `p` to modify properties of the plot after creating it. For a list of properties, see Line.

Examples

Plot 3-D Helix

Define `t` as a vector of values between 0 and 10π. Define `st` and `ct` as vectors of sine and cosine values. Then plot `st`, `ct`, and `t`.

```
t = 0:pi/50:10*pi;
st = sin(t);
ct = cos(t);
plot3(st,ct,t)
```



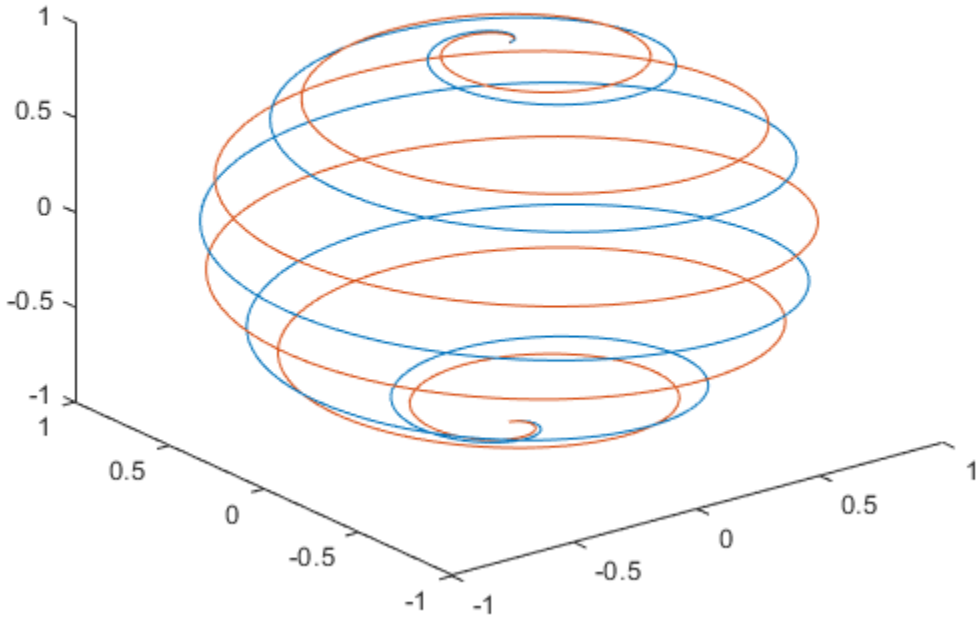
Plot Multiple Lines

Create two sets of *x*-, *y*-, and *z*-coordinates.

```
t = 0:pi/500:pi;
xt1 = sin(t).*cos(10*t);
yt1 = sin(t).*sin(10*t);
zt1 = cos(t);

xt2 = sin(t).*cos(12*t);
yt2 = sin(t).*sin(12*t);
zt2 = cos(t);
```

Call the `plot3` function, and specify consecutive XYZ triplets.

```
plot3(xt1,yt1,zt1,xt2,yt2,zt2)
```



Plot Multiple Lines Using Matrices

Create matrix X containing three rows of *x*-coordinates. Create matrix Y containing three rows of *y*-coordinates.

```
t = 0:pi/500:pi;
X(1,:) = sin(t).*cos(10*t);
X(2,:) = sin(t).*cos(12*t);
```

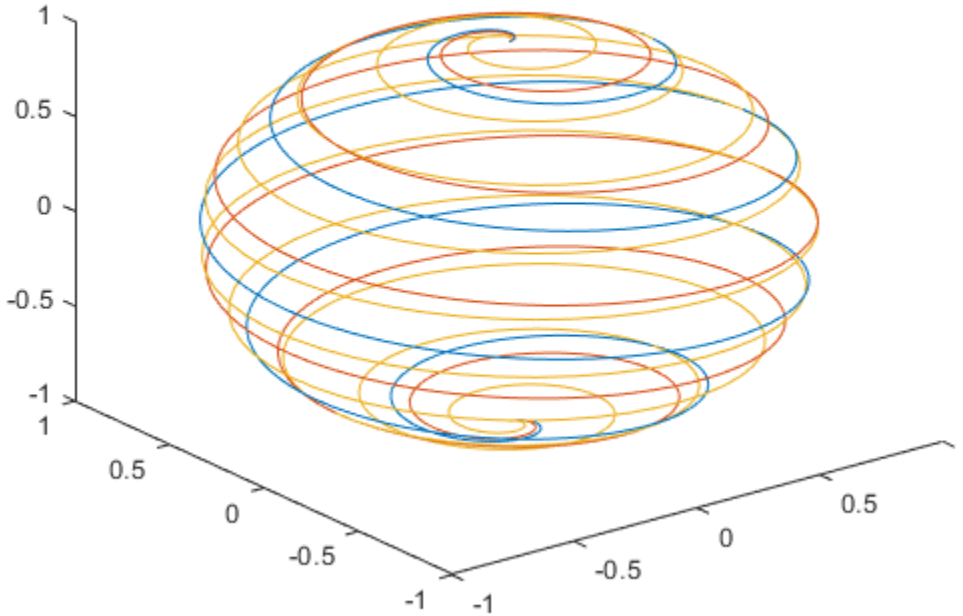
```
X(3,:) = sin(t).*cos(20*t);

Y(1,:) = sin(t).*sin(10*t);
Y(2,:) = sin(t).*sin(12*t);
Y(3,:) = sin(t).*sin(20*t);
```

Create matrix Z containing the *z*-coordinates for all three sets.

```
Z = cos(t);
```

Plot all three sets of coordinates on the same set of axes.

```
plot3(X,Y,Z)
```



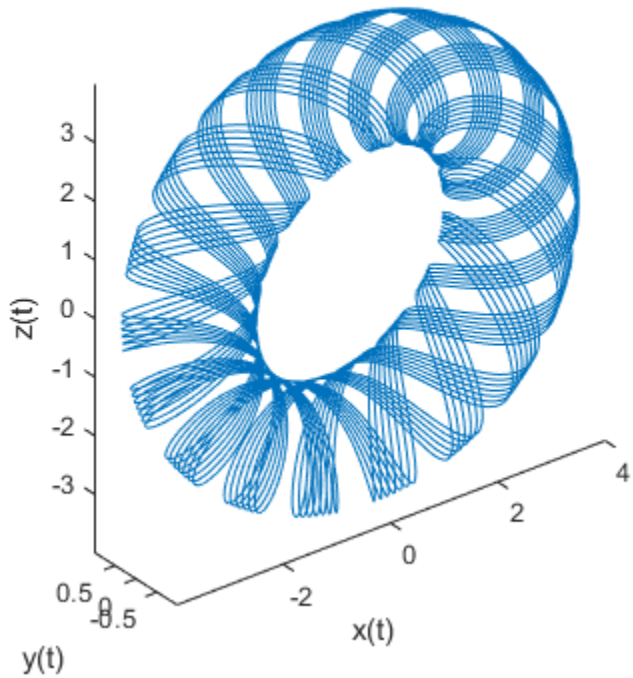
Specify Equally-Spaced Tick Units and Axis Labels

Create vectors `xt`, `yt`, and `zt`.

```
t = 0:pi/500:40*pi;
xt = (3 + cos(sqrt(32)*t)).*cos(t);
yt = sin(sqrt(32) * t);
zt = (3 + cos(sqrt(32)*t)).*sin(t);
```

Plot the data, and use the `axis equal` command to space the tick units equally along each axis. Then specify the labels for each axis.

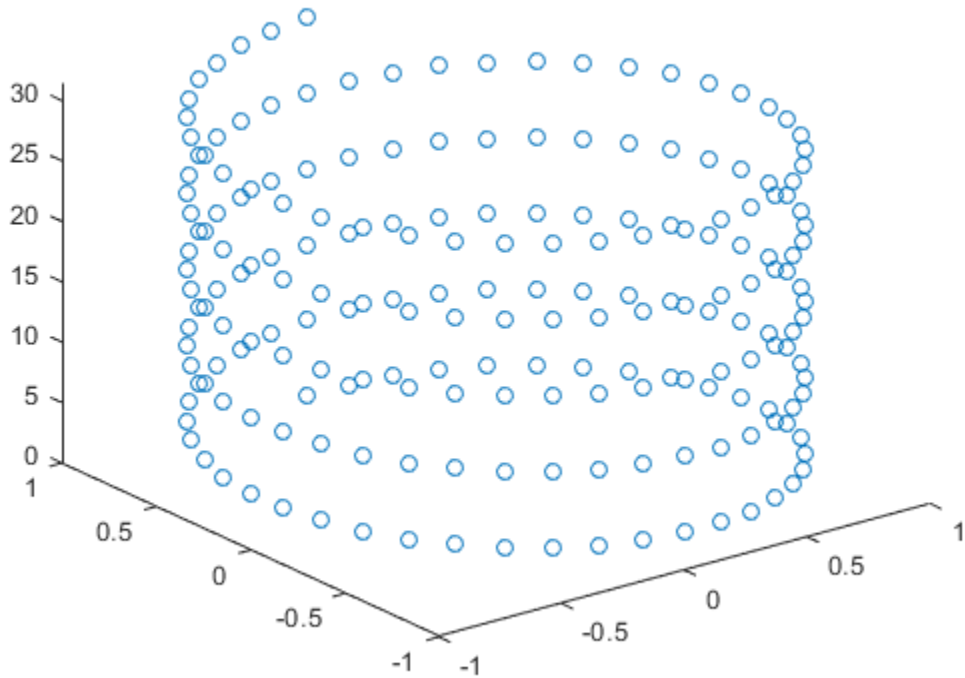
```
plot3(xt,yt,zt)
axis equal
xlabel('x(t)')
ylabel('y(t)')
zlabel('z(t)')
```



Plot Points as Markers Without Lines

Create vectors `t`, `xt`, and `yt`, and plot the points in those vectors using circular markers.

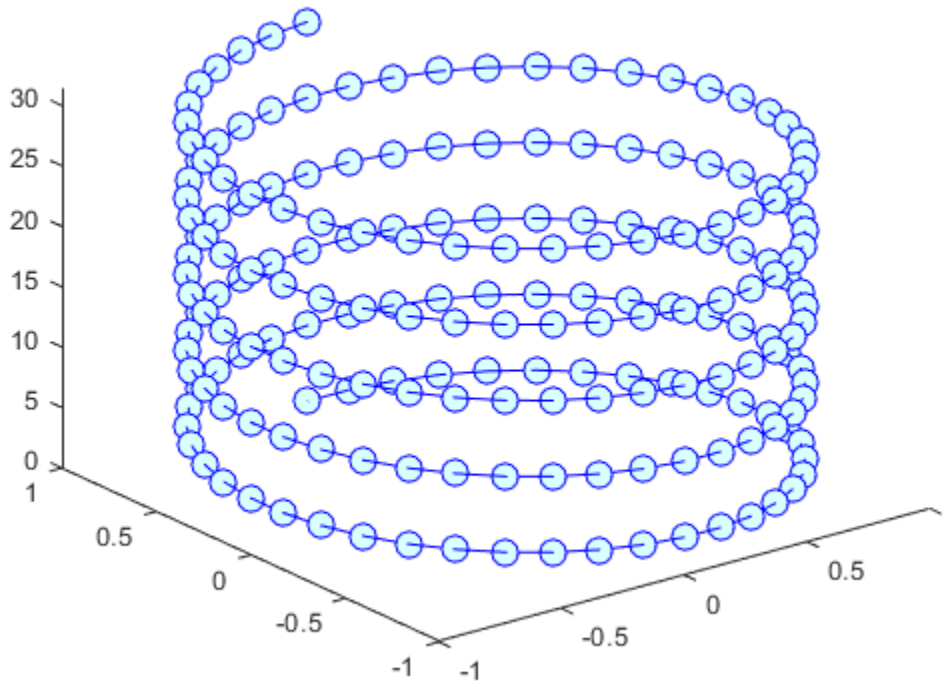
```
t = 0:pi/20:10*pi;
xt = sin(t);
yt = cos(t);
plot3(xt,yt,t,'o')
```



Customize Color and Marker

Create vectors `t`, `xt`, and `yt`, and plot the points in those vectors as a blue line with 10-point circular markers. Use a hexadecimal color code to specify a light blue fill color for the markers.

```
t = 0:pi/20:10*pi;
xt = sin(t);
yt = cos(t);
plot3(xt,yt,t,'-o','Color','b','MarkerSize',10,'MarkerFaceColor','#D9FFFF')
```



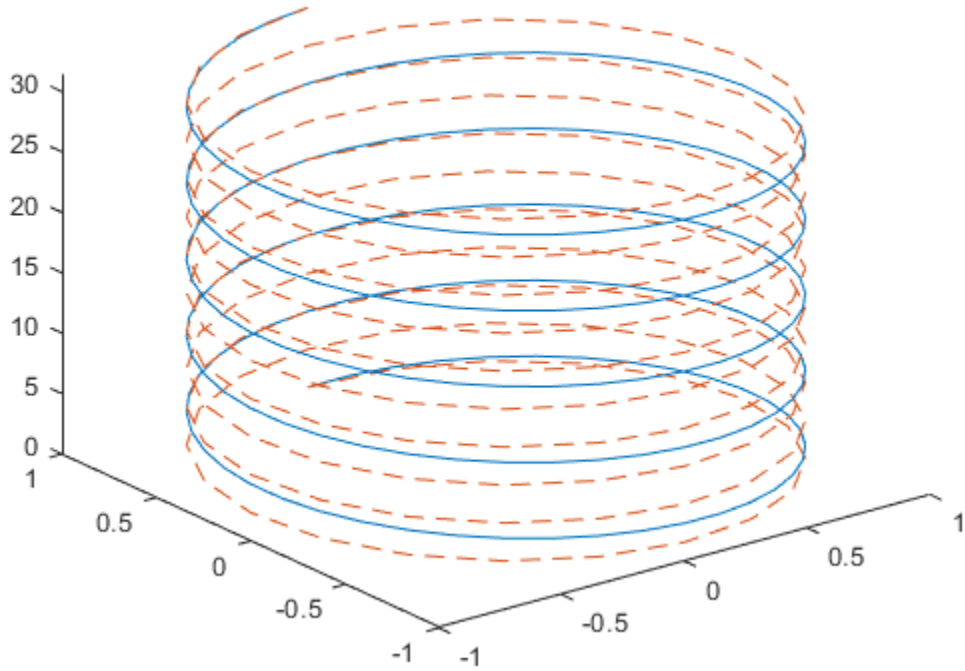
Specify Line Style

Create vector `t`. Then use `t` to calculate two sets of *x* and *y* values.

```
t = 0:pi/20:10*pi;
xt1 = sin(t);
yt1 = cos(t);

xt2 = sin(2*t);
yt2 = cos(2*t);
```

Plot the two sets of values. Use the default line for the first set, and specify a dashed line for the second set.

```
plot3(xt1,yt1,t,xt2,yt2,t,'--')
```

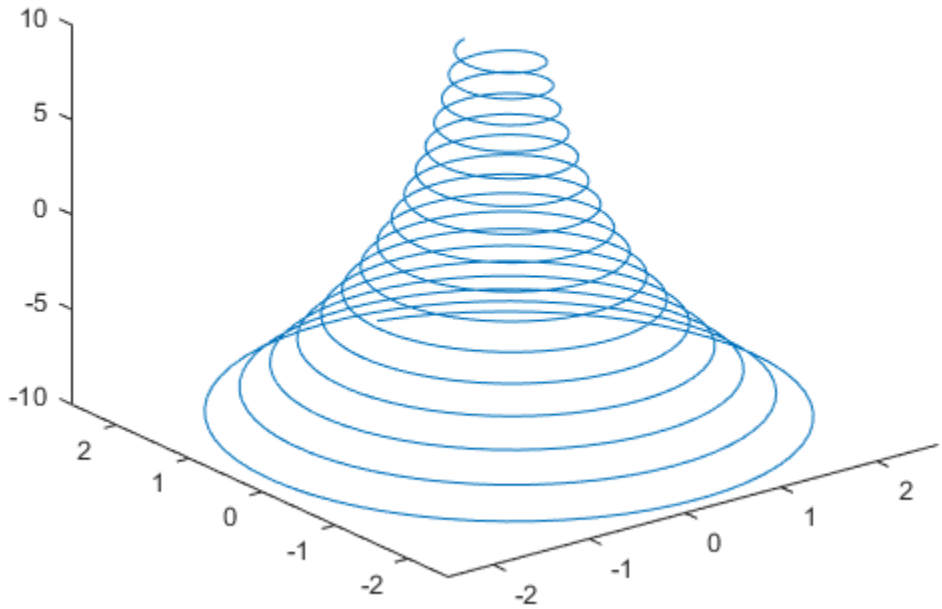


Modify Line After Plotting

Create vectors `t`, `xt`, and `yt`, and plot the data in those vectors. Return the chart line in the output variable `p`.

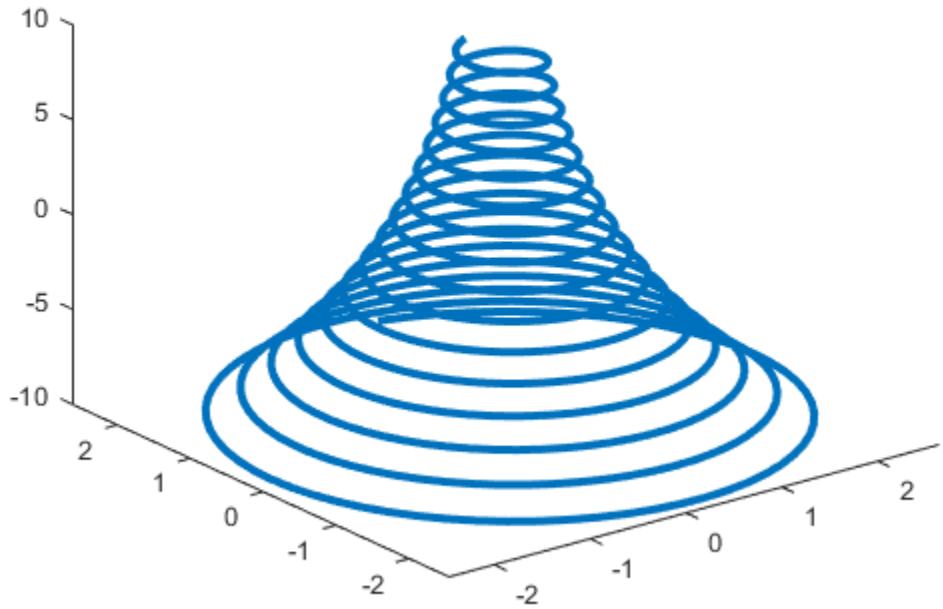
```
t = linspace(-10,10,1000);
xt = exp(-t./10).*sin(5*t);
```

```
yt = exp(-t./10).*cos(5*t);
p = plot3(xt,yt,t);
```



Change the line width to 3.

```
p.LineWidth = 3;
```

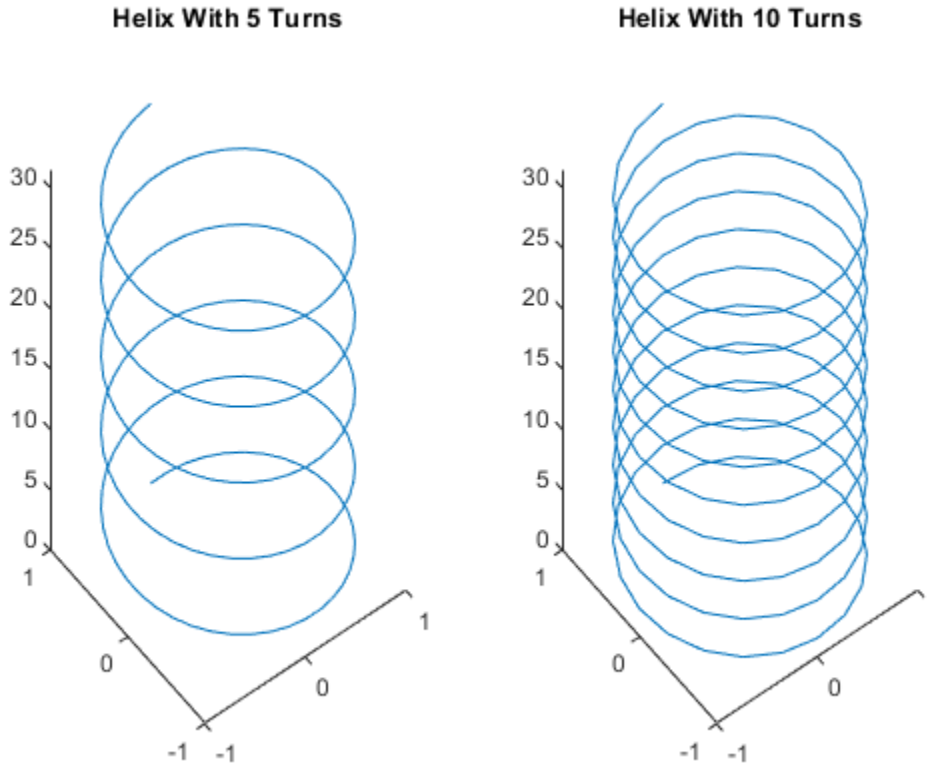


Specify Target Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 1-by-2 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Create separate line plots in the axes by specifying the axes object as the first argument to `plot3`.

```
tiledlayout(1,2)

% Left plot
ax1 = nexttile;
```

```
t = 0:pi/20:10*pi;
xt1 = sin(t);
yt1 = cos(t);
plot3(ax1,xt1,yt1,t)
title(ax1,'Helix With 5 Turns')

% Right plot
ax2 = nexttile;
t = 0:pi/20:10*pi;
xt2 = sin(2*t);
yt2 = cos(2*t);
plot3(ax2,xt2,yt2,t)
title(ax2,'Helix With 10 Turns')
```



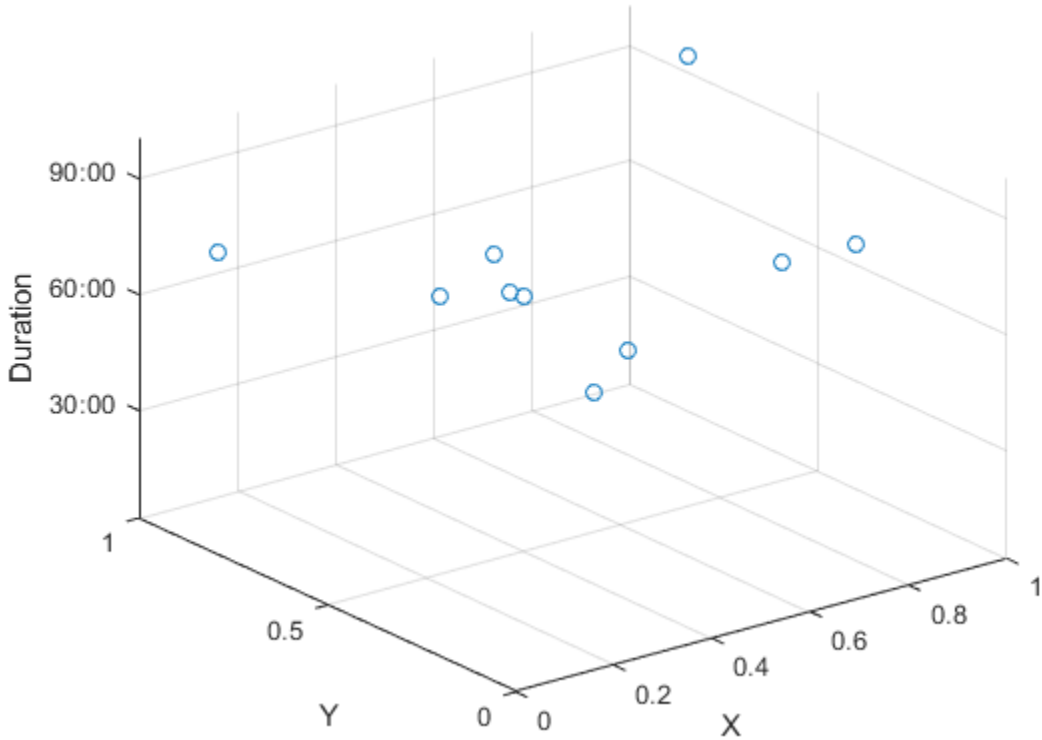
Plot Duration Data with Custom Tick Format

Create x and y as vectors of random values between 0 and 1. Create z as a vector of random duration values.

```
x = rand(1,10);
y = rand(1,10);
z = duration(rand(10,1),randi(60,10,1),randi(60,10,1));
```

Plot x, y, and z, and specify the format for the *z*-axis as minutes and seconds. Then add axis labels, and turn on the grid to make it easier to visualize the points within the plot box.

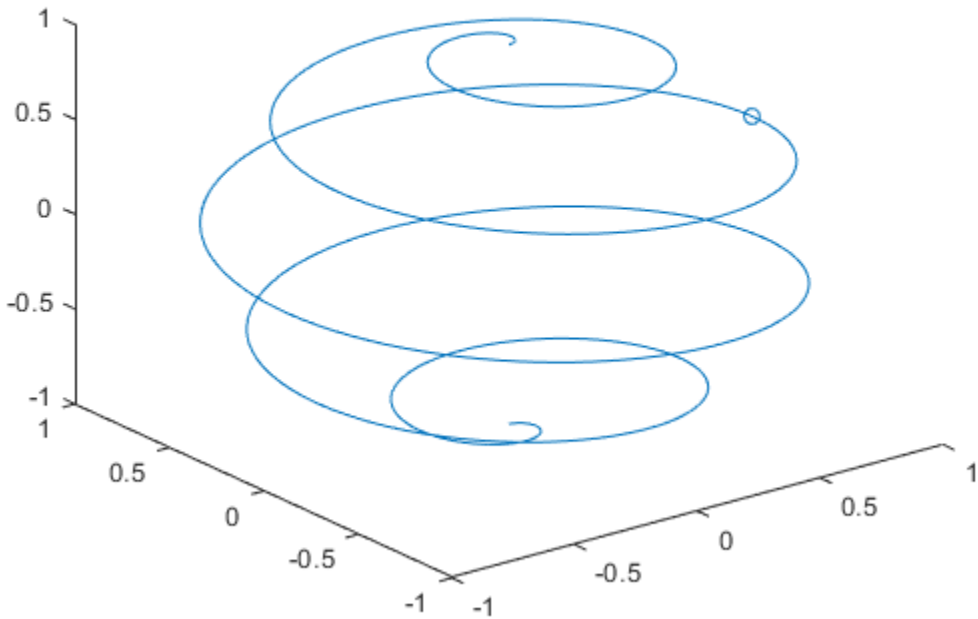
```
plot3(x,y,z,'o','DurationTickFormat','mm:ss')
xlabel('X')
ylabel('Y')
zlabel('Duration')
grid on
```



Plot Line With Marker at One Data Point

Create vectors `xt`, `yt`, and `zt`. Plot the values, specifying a solid line with circular markers using the `LineSpec` argument. Specify the `MarkerIndices` property to place one marker at the 200th data point.

```
t = 0:pi/500:pi;
xt(1,:) = sin(t).*cos(10*t);
yt(1,:) = sin(t).*sin(10*t);
zt = cos(t);
plot3(xt,yt,zt,'-o','MarkerIndices',200)
```



Input Arguments

X — *x*-coordinates
scalar | vector | matrix

x-coordinates, specified as a scalar, vector, or matrix. The size and shape of X depends on the shape of your data and the type of plot you want to create. This table describes the most common situations.

Type of Plot	How to Specify Coordinates
Single point	Specify X, Y, and Z as scalars and include a marker. For example: `plot3(1,2,3,'o')`
One set of points	Specify X, Y, and Z as any combination of row or column vectors of the same length. For example: `plot3([1 2 3],[4; 5; 6],[7 8 9])`
Multiple sets of points (using vectors)	Specify consecutive sets of X, Y, and Z vectors. For example: `plot3([1 2 3],[4 5 6],[7 8 9],[1 2 3],[4 5 6],[10 11 12])`
Multiple sets of points (using matrices)	Specify at least one of X, Y, or Z as a matrix, and the others as vectors. Each of X, Y, and Z must have at least one dimension that is same size. For best results, specify all vectors of the same shape and all matrices of the same shape. For example: `plot3([1 2 3],[4 5 6],[7 8 9; 10 11 12])`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

Y — *y*-coordinates
scalar | vector | matrix

y-coordinates, specified as a scalar, vector, or matrix. The size and shape of Y depends on the shape of your data and the type of plot you want to create. This table describes the most common situations.

Type of Plot	How to Specify Coordinates
Single point	Specify X, Y, and Z as scalars and include a marker. For example: `plot3(1,2,3,'o')`
One set of points	Specify X, Y, and Z as any combination of row or column vectors of the same length. For example: `plot3([1 2 3],[4; 5; 6],[7 8 9])`

Type of Plot	How to Specify Coordinates
Multiple sets of points (using vectors)	Specify consecutive sets of X, Y, and Z vectors. For example: `plot3([1 2 3],[4 5 6],[7 8 9],[1 2 3],[4 5 6],[10 11 12])`
Multiple sets of points (using matrices)	Specify at least one of X, Y, or Z as a matrix, and the others as vectors. Each of X, Y, and Z must have at least one dimension that is same size. For best results, specify all vectors of the same shape and all matrices of the same shape. For example: `plot3([1 2 3],[4 5 6],[7 8 9; 10 11 12])`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

Z — *z*-coordinates
scalar | vector | matrix

z-coordinates, specified as a scalar, vector, or matrix. The size and shape of Z depends on the shape of your data and the type of plot you want to create. This table describes the most common situations.

Type of Plot	How to Specify Coordinates
Single point	Specify X, Y, and Z as scalars and include a marker. For example: `plot3(1,2,3,'o')`
One set of points	Specify X, Y, and Z as any combination of row or column vectors of the same length. For example: `plot3([1 2 3],[4; 5; 6],[7 8 9])`
Multiple sets of points (using vectors)	Specify consecutive sets of X, Y, and Z vectors. For example: `plot3([1 2 3],[4 5 6],[7 8 9],[1 2 3],[4 5 6],[10 11 12])`
Multiple sets of points (using matrices)	Specify at least one of X, Y, or Z as a matrix, and the others as vectors. Each of X, Y, and Z must have at least one dimension that is same size. For best results, specify all vectors of the same shape and all matrices of the same shape. For example: `plot3([1 2 3],[4 5 6],[7 8 9; 10 11 12])`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`LineSpec` — Line style, marker, and color
character vector | string

Line style, marker, and color, specified as a character vector or string containing symbols. The symbols can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the marker and no line.

Example: `'--or'` is a red dashed line with circle markers

Line Style	Description
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Marker	Description
`o`	Circle
`+`	Plus sign
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Pentagram
`h`	Hexagram

Color	Description
y	yellow
m	magenta
c	cyan
r	red
g	green
b	blue
w	white
k	black

ax — Target axes
`Axes` object

Target axes, specified as an `Axes` object. If you do not specify the axes and if the current axes are Cartesian axes, then the `plot` function uses the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `plot3([1 2],[3 4],[5 6],'Color','red')` specifies a red line for the plot.

Note The properties listed here are only a subset. For a complete list, see Chart Line.

`Color` — Color
`[0 0.4470 0.7410]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The color you specify sets the line color. It also sets the marker edge color when the `MarkerEdgeColor` property is set to `'auto'`.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

LineWidth — Line width
0.5 (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

MarkerSize — Marker size
6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

MarkerEdgeColor — Marker outline color
'auto' (default) | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...

Marker outline color, specified as 'auto', an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of 'auto' uses the same color as the Color property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` option uses the same color as the `Color` property of the parent axes. If you specify `'auto'` and the axes plot box is invisible, the marker fill color is the color of the figure.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Tips

- Use `NaN` or `Inf` to create breaks in the lines. For example, this code plots a line with a break between `z=2` and `z=4`.

  ```
  plot3([1 2 3 4 5],[1 2 3 4 5],[1 2 NaN 4 5])
  ```

- `plot3` uses colors and line styles based on the `ColorOrder` and `LineStyleOrder` properties of the axes. `plot3` cycles through the colors with the first line style. Then, it cycles through the colors again with each additional line style.

  Starting in R2019b, you can change the colors and the line styles after plotting by setting the `ColorOrder` or `LineStyleOrder` properties on the axes. You can also call the `colororder` function to change the color order for all the axes in the figure.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions
`axis` | `bar3` | `grid` | `nexttile` | `scatter3` | `tiledlayout`

Properties
Chart Line

Topics

"Plot Dates and Durations"
"Plot Categorical Data"

Introduced before R2006a

plotbrowser

Show or hide figure **Plot Browser**

Syntax

```
plotbrowser('on')
plotbrowser('off')
plotbrowser('toggle')
plotbrowser
plotbrowser(figure_handle,...)
```

Description

`plotbrowser('on')` displays the Plot Browser on the current figure.

`plotbrowser('off')` hides the Plot Browser on the current figure.

`plotbrowser('toggle')` toggles the visibility of the Plot Browser on the current figure.

`plotbrowser` is the same as `plotbrowser('on')`.

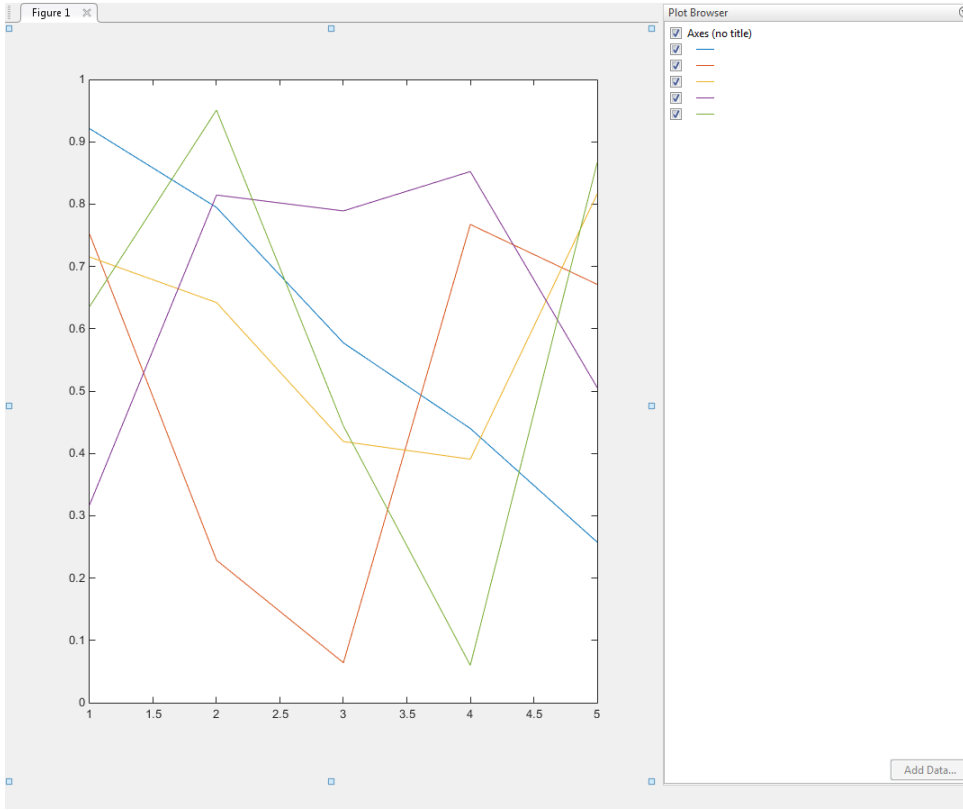
`plotbrowser(figure_handle,...)` shows or hides the Plot Browser on the figure specified by `figure_handle`.

Examples

Open Plot Browser

Plot a 5-by-5 matrix of random numbers. Then, open the plot browser.

```
plot(rand(5))
plotbrowser('on')
```



Tips

If you call `plotbrowser` in a MATLAB program and subsequent lines depend on the Plot Browser being fully initialized, follow it by `drawnow` to ensure complete initialization.

Alternatives

Open or close the **Plot Browser** tool from the figure's **View** menu.

See Also

`figurepalette` | `plottools` | `propertyeditor`

Introduced before R2006a

plotedit

Interactively edit and annotate plots

Syntax

```
plotedit on
plotedit off
plotedit
plotedit(h)
plotedit(state)
plotedit(h,state)
```

Description

`plotedit on` starts plot edit mode for the current figure, allowing you to use a graphical interface to annotate and edit plots easily. In plot edit mode, you can label axes, change line styles, and add text, line, and arrow annotations.

`plotedit off` ends plot mode for the current figure.

`plotedit` toggles the plot edit mode for the current figure.

`plotedit(h)` toggles the plot edit mode for the figure specified by figure handle `h`.

`plotedit(state)` specifies the `plotedit` state for the current figure. Values for `state` can be as shown.

Value for state	Description
`'on'`	Starts plot edit mode
`'off'`	Ends plot edit mode
`'showtoolsmenu'`	Displays the **Tools** menu in the menu bar
`'hidetoolsmenu'`	Removes the **Tools** menu from the menu bar

Note `'hidetoolsmenu'` is intended for UI developers who do not want the **Tools** menu to appear in applications that use the figure window.

`plotedit(h,state)` specifies the `plotedit` `state` for figure handle `h`.

Examples

Start plot edit mode for figure 2.

```
plotedit(2)
```

End plot edit mode for figure 2.

```
plotedit(2, 'off')
```

Hide the **Tools** menu for the current figure:

```
plotedit('hidetoolsmenu')
```

See Also

`axes` | `line` | `open` | `plot` | `print` | `propedit` | `saveas` | `text`

Introduced before R2006a

plotmatrix

Scatter plot matrix

Syntax

```
plotmatrix(X,Y)
plotmatrix(X)
plotmatrix( ___ ,LineSpec)
```

```
[S,AX,BigAx,H,HAx] = plotmatrix( ___ )
```

Description

`plotmatrix(X,Y)` creates a matrix of subaxes containing scatter plots of the columns of `X` against the columns of `Y`. If `X` is *p*-by-*n* and `Y` is *p*-by-*m*, then `plotmatrix` produces an *n*-by-*m* matrix of subaxes.

`plotmatrix(X)` is the same as `plotmatrix(X,X)` except that the subaxes along the diagonal are replaced with histogram plots of the data in the corresponding column of `X`. For example, the subaxes along the diagonal in the *i*th column is replaced by `histogram(X(:,i))`.

`plotmatrix( ___ ,LineSpec)` specifies the line style, marker symbol, and color for the scatter plots. The option `LineSpec` can be preceded by any of the input argument combinations in the previous syntaxes.

`[S,AX,BigAx,H,HAx] = plotmatrix( ___ )` returns the graphic objects created as follows:

- `S` – Chart line objects for the scatter plots
- `AX` – Axes objects for each subaxes
- `BigAx` – Axes object for big axes that frames the subaxes
- `H` – Histogram objects for the histogram plots
- `HAx` – Axes objects for the invisible histogram axes

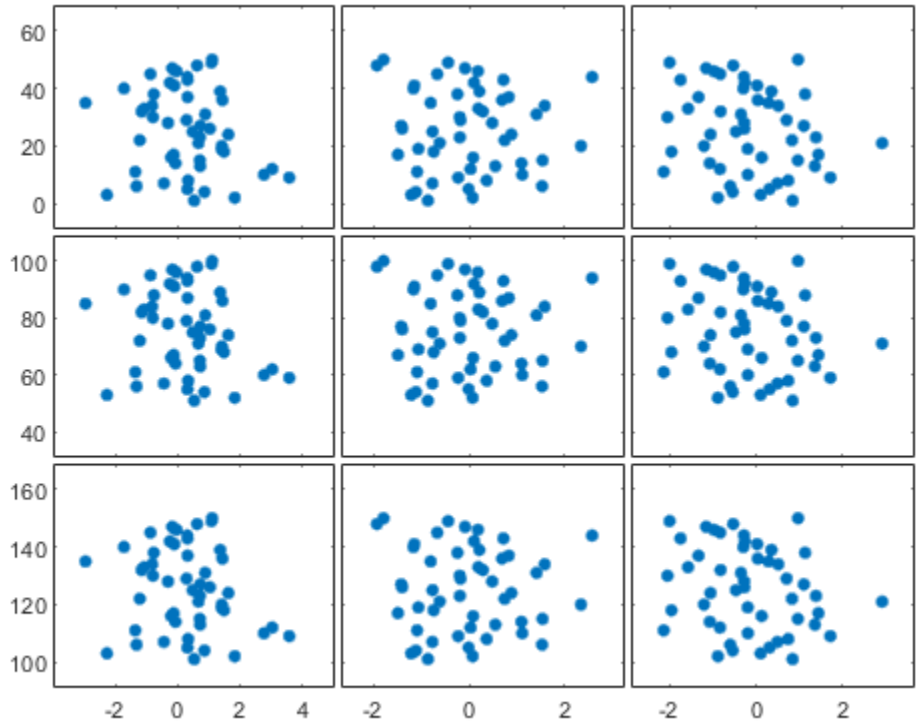
`BigAx` is left as the current axes (`gca`) so that a subsequent `title`, `xlabel`, or `ylabel` command centers text with respect to the big axes.

Examples

Create Scatter Plot Matrix with Two Matrix Inputs

Create `X` as a matrix of random data and `Y` as a matrix of integer values. Then, create a scatter plot matrix of the columns of `X` against the columns of `Y`.

```
X = randn(50,3);
Y = reshape(1:150,50,3);
plotmatrix(X,Y)
```

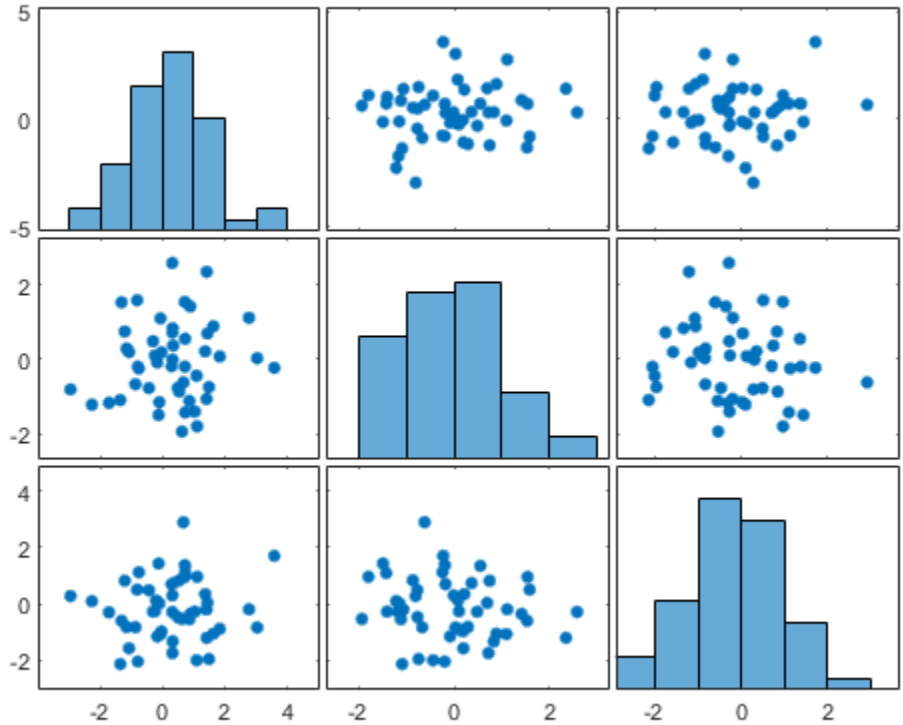


The subplot in the ith row, jth column of the figure is a scatter plot of the ith column of Y against the jth column of X.

Create Scatter Plot Matrix with One Matrix Input

Create a scatter plot matrix of random data. The subplot in the ith row, jth column of the matrix is a scatter plot of the ith column of X against the jth column of X. Along the diagonal are histogram plots of each column of X.

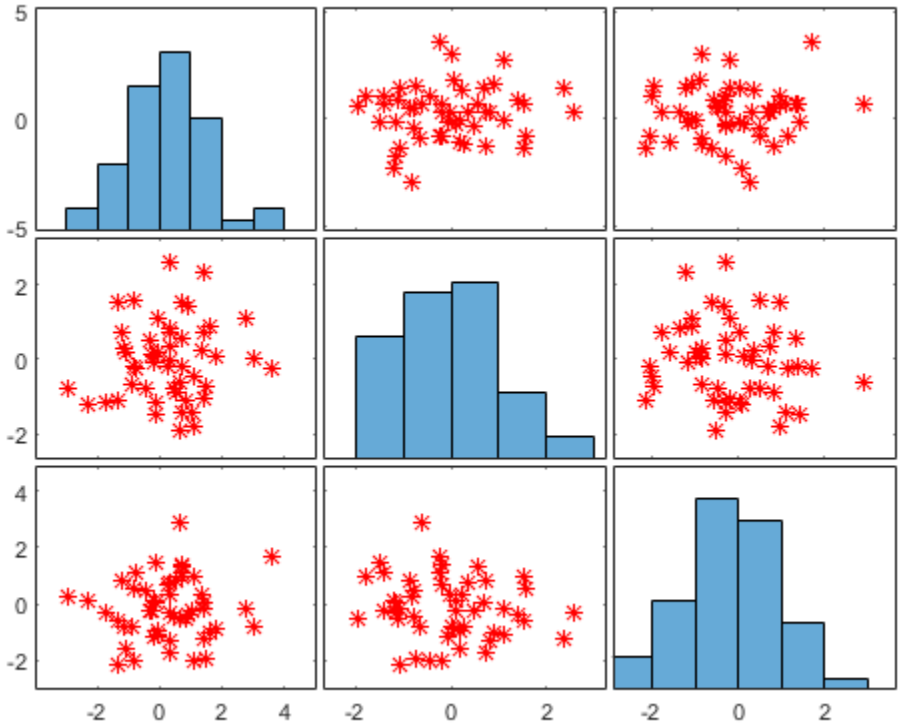
```
X = randn(50,3);
plotmatrix(X)
```



Specify Marker Type and Color

Create a scatter plot matrix of random data. Specify the marker type and the color for the scatter plots.

```
X = randn(50,3);
plotmatrix(X,'*r')
```

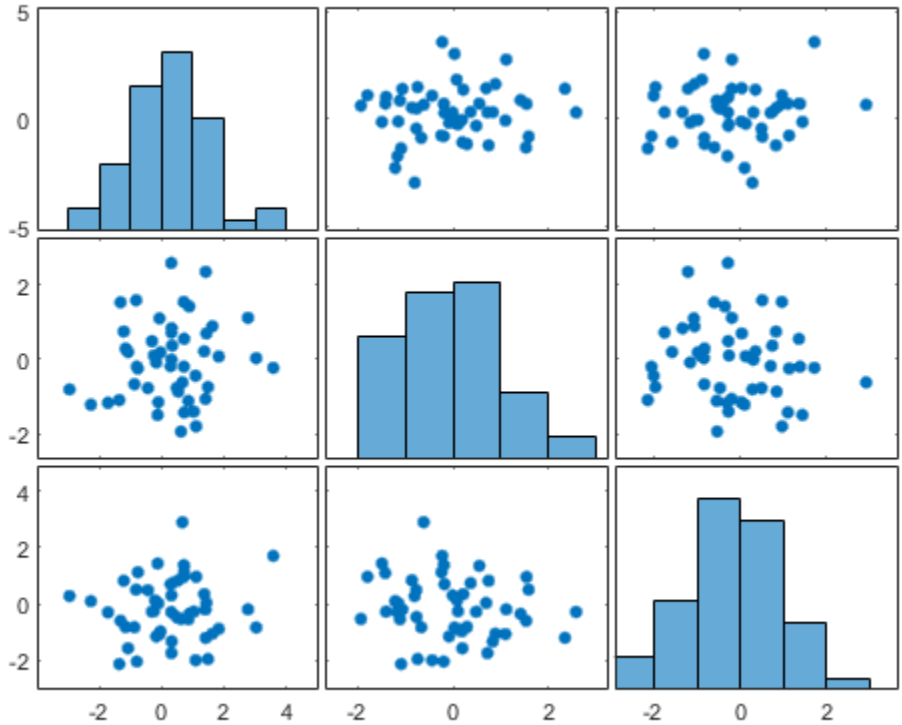


The `LineSpec` option sets properties for the scatter plots. To set properties for the histogram plots, return the histogram objects.

Modify Scatter Plot Matrix After Creation

Create a scatter plot matrix of random data.

```
rng default
X = randn(50,3);
[S,AX,BigAx,H,HAx] = plotmatrix(X);
```

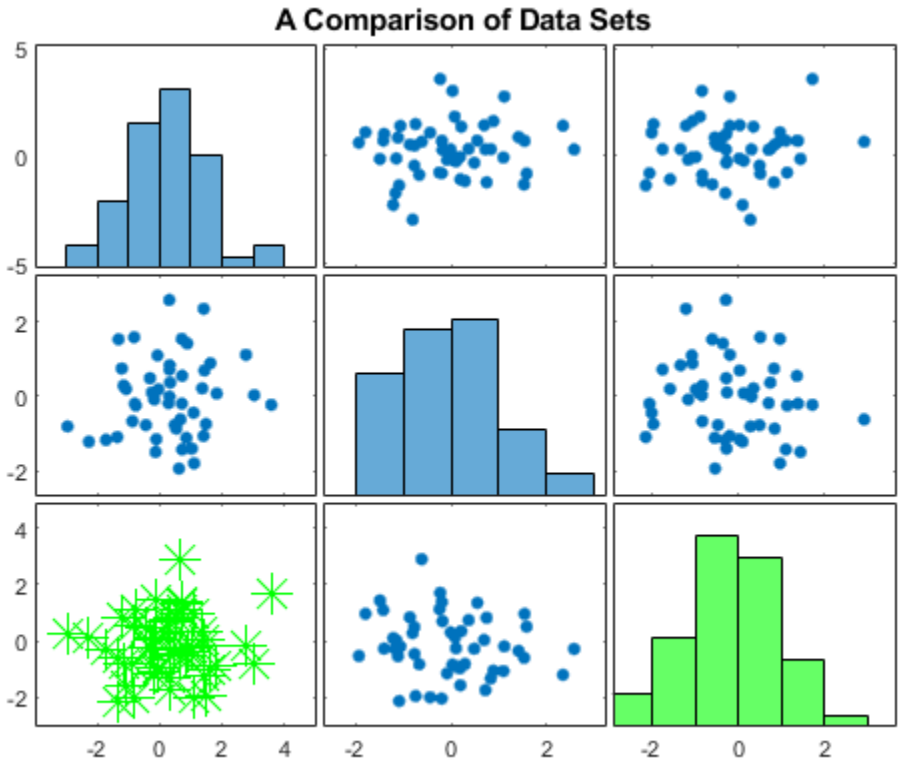


To set properties for the scatter plots, use `S`. To set properties for the histograms, use `H`. To set axes properties, use `AX`, `BigAx`, and `HAx`. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead.

Set the color and marker type for the scatter plot in the lower left corner of the figure. Set the color for the histogram plot in the lower right corner. Use the `title` command to title the figure.

```
S(3).Color = 'g';
S(3).Marker = '*';

H(3).EdgeColor = 'k';
H(3).FaceColor = 'g';
```

```
title(BigAx,'A Comparison of Data Sets')
```



Input Arguments

X — Data to display
matrix

Data to display, specified as a matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Y — Data to plot against X
matrix

Data to plot against X, specified as a matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`LineSpec` — Line style, marker symbol, and color for scatter plots
character vector | string

Line style, marker symbol, and color for the scatter plots, specified as a character vector or string. For more information on line style, marker symbol, and color options see `LineSpec`.

Example: `':*r'`

Output Arguments

S — Chart line objects for scatter plots
matrix

Chart line objects for the scatter plots, returned as a matrix. These are unique identifiers, which you can use to query and modify the properties of a specific scatter plot.

`AX` — Axes objects for subaxes
matrix

Axes objects for the subaxes, returned as a matrix. These are unique identifiers, which you can use to query and modify the properties of a specific subaxes.

`BigAx` — Axes object for big axes
scalar

Axes object for big axes, returned as a scalar. This is a unique identifier, which you can use to query and modify properties of the big axes.

`H` — Histogram objects
vector | []

Histogram objects, returned as a vector or `[]`. These are unique identifiers, which you can use to query and modify the properties of a specific histogram object. If no histogram plots are created, then `H` is returned as empty brackets.

Note Starting in R2015b, `H` is a vector of histogram objects, In previous releases, it was a vector of patch objects.

`HAx` — Axes objects for invisible histogram axes
vector | []

Axes objects for invisible histogram axes, returned as a vector or `[]`. These are unique identifiers, which you can use to query and modify the properties of a specific axes. If no histogram plots are created, then `HAx` is returned as empty brackets.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`nexttile` | `scatter` | `scatter3` | `tiledlayout`

Introduced before R2006a

plottools

Show or hide plot tools

Syntax

```
plottools('on')
plottools('off')
plottools('toggle')
plottools
plottools(figure_handle,...)
plottools(...,'tool')
```

Description

`plottools('on')` displays the Figure Palette, Plot Browser, and Property Editor on the current figure, configured as you last used them.

`plottools('off')` hides the Figure Palette, Plot Browser, and Property Editor on the current figure.

`plottools('toggle')` toggles the visibility of the Figure Palette, Plot Browser, and Property Editor on the current figure.

`plottools` is the same as `plottools('on')`.

`plottools(figure_handle,...)` displays or hides the plot tools on the specified figure instead of on the current figure.

`plottools(...,'tool')` operates on the specified tool only. `'tool'` can be one of these values:

- `'figurepalette'`
- `'plotbrowser'`
- `'propertyeditor'`

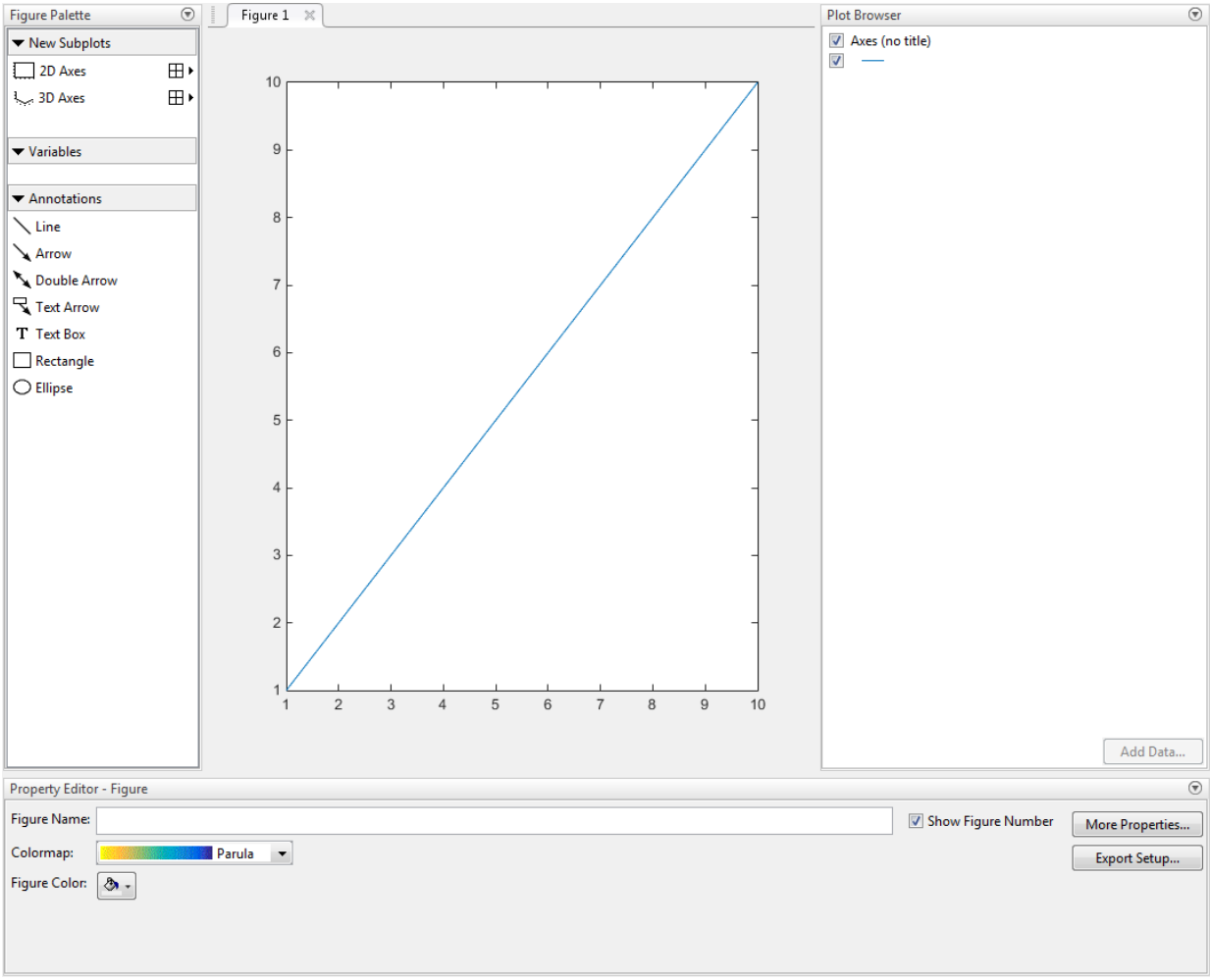
Note The first time you open the plotting tools, all three of them appear, grouped around the current figure as shown above. If you close, move, or undock any of the tools, MATLAB remembers the configuration you left them in and restores it when you invoke the tools for subsequent figures, both within and across MATLAB sessions.

Examples

Open Plot Tools

Create a line plot and open the plot tools.

```
plot(1:10);
plottools('on')
```



Alternatives

Select the **Figure Palette**, **Plot Browser**, and **Property Editor** tools from the figure's **View** menu.

See Also

`figurepalette` | `plotbrowser` | `propertyeditor`

Topics

"Interactively Explore Plotted Data"

Introduced before R2006a

plotyy

(Not recommended) Create graph with two y-axes

Note `plotyy` is not recommended. Use `yyaxis` instead. For more information, see "Compatibility Considerations".

Syntax

```
plotyy(X1,Y1,X2,Y2)
plotyy(X1,Y1,X2,Y2,function)
plotyy(X1,Y1,X2,Y2,'function1','function2')
plotyy(AX1,___)
[AX,H1,H2] = plotyy(___)
```

Description

`plotyy(X1,Y1,X2,Y2)` plots `Y1` versus `X1` with *y*-axis labeling on the left and plots `Y2` versus `X2` with *y*-axis labeling on the right.

`plotyy(X1,Y1,X2,Y2,function)` uses the specified plotting function to produce the graph.

`function` can be either a function handle or a character vector specifying `plot`, `semilogx`, `semilogy`, `loglog`, `stem`, or any MATLAB function that accepts the syntax

```
h = function(x,y)
```

For example,

```
plotyy(x1,y1,x2,y2,@loglog) % function handle
plotyy(x1,y1,x2,y2,'loglog') % character vector
```

Function handles enable you to access user-defined local functions and can provide other advantages. For more information on using function handles, see "Create Function Handle".

`plotyy(X1,Y1,X2,Y2,'function1','function2')` uses `function1(X1,Y1)` to plot the data for the left axis and `function2(X2,Y2)` to plot the data for the right axis.

`plotyy(AX1, ___ )` plots the data using the axes specified by `AX1` for the first set of data, instead of using the current axes. Specify `AX1` as a single axes object or a vector of the two axes objects returned by a previous call to `plotyy`. If you specify a vector, then `plotyy` uses the first axes object in the vector. Use this option with any of the input argument combinations in the previous syntaxes.

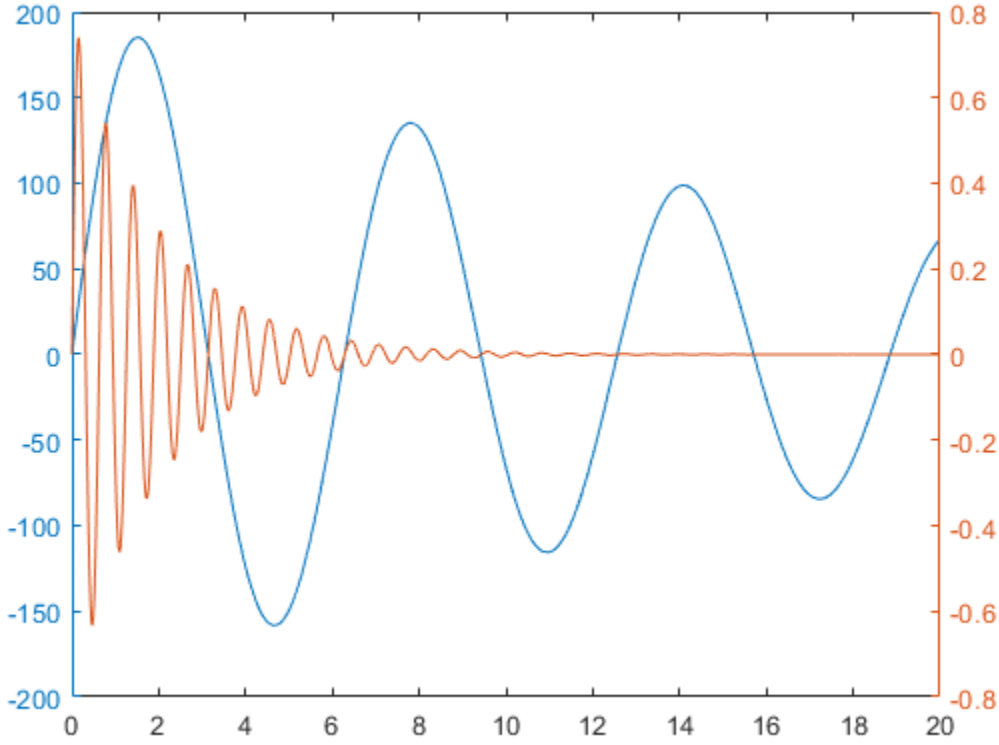
`[AX,H1,H2] = plotyy( ___ )` returns the handles of the two axes created in `AX` and the handles of the graphics objects from each plot in `H1` and `H2`. `AX(1)` is the left axes and `AX(2)` is the right axes.

Examples

Plot Two Data Sets with Different y-Axes

Plot two data sets on one graph using two *y*-axes.

```
x = 0:0.01:20;
y1 = 200*exp(-0.05*x).*sin(x);
y2 = 0.8*exp(-0.5*x).*sin(10*x);

figure % new figure
plotyy(x,y1,x,y2)
```

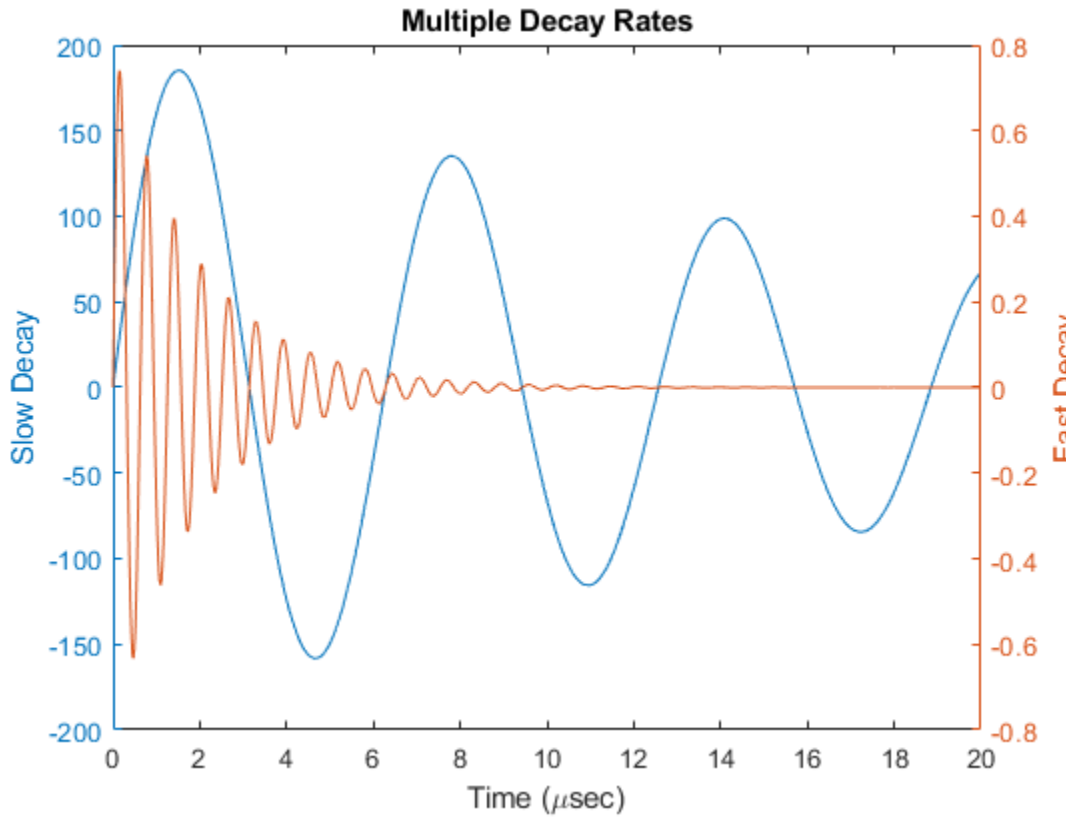


Add Title and Axis Labels

Plot two data sets using a graph with two *y*-axes. Add a title and axis labels.

```
x = 0:0.01:20;
y1 = 200*exp(-0.05*x).*sin(x);
y2 = 0.8*exp(-0.5*x).*sin(10*x);

figure % new figure
[hAx,hLine1,hLine2] = plotyy(x,y1,x,y2);
```

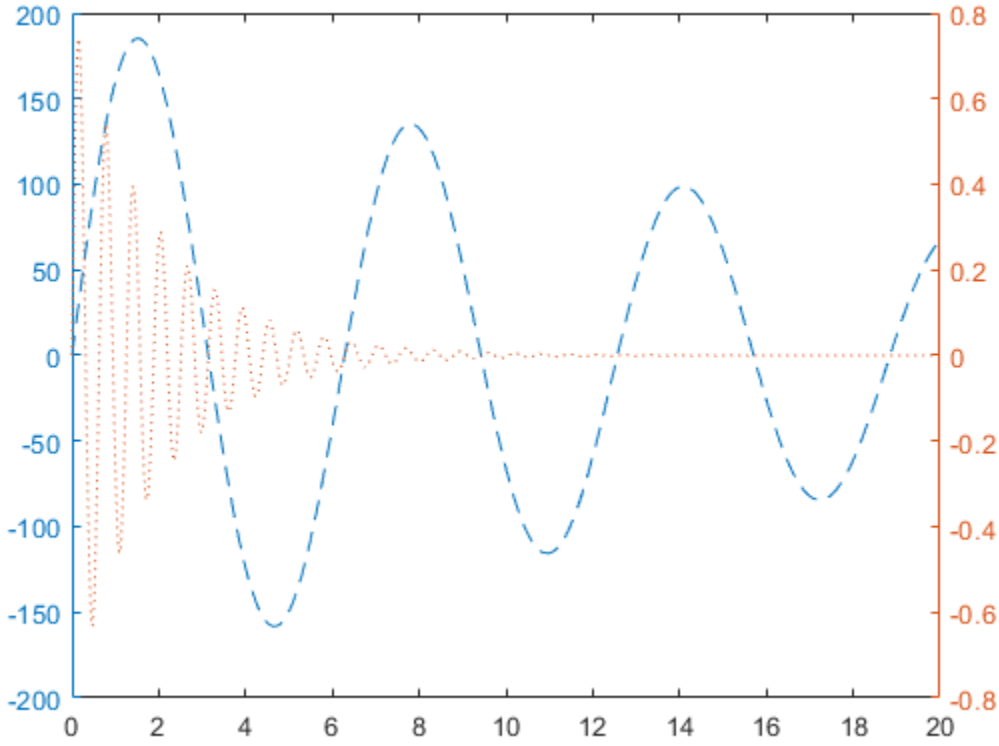
```
title('Multiple Decay Rates')
xlabel('Time (\musec)')

ylabel(hAx(1),'Slow Decay') % left y-axis
ylabel(hAx(2),'Fast Decay') % right y-axis
```



Change Line Styles

Plot two data sets using a graph with two *y*-axes. Change the line styles. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead.

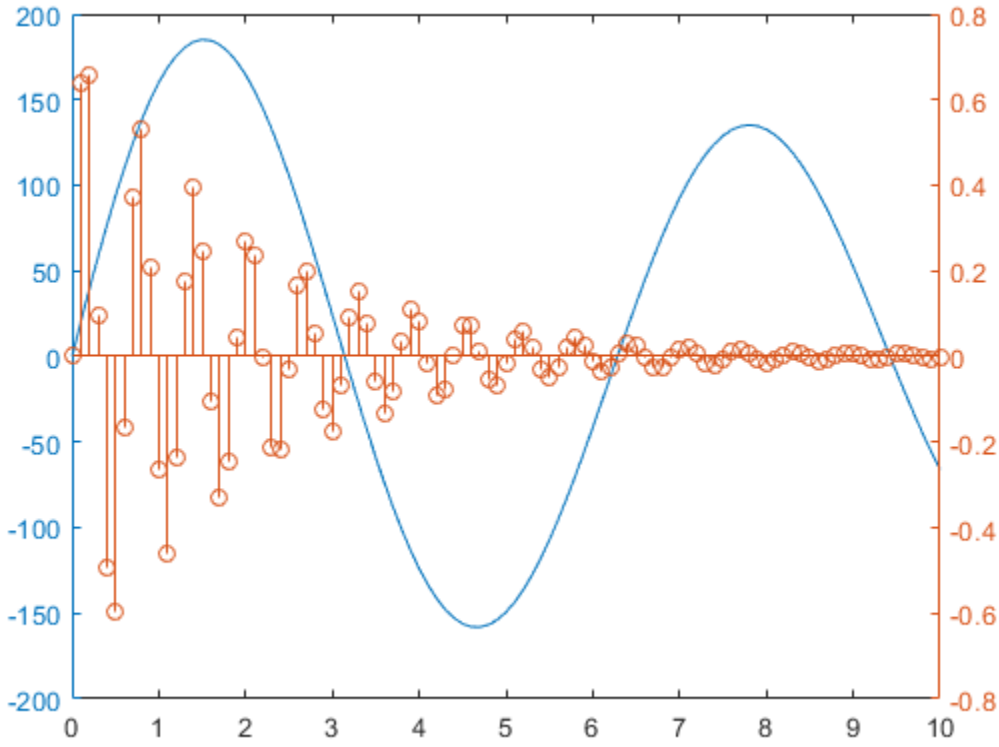
```
x = 0:0.01:20;
y1 = 200*exp(-0.05*x).*sin(x);
y2 = 0.8*exp(-0.5*x).*sin(10*x);

[hAx,hLine1,hLine2] = plotyy(x,y1,x,y2);
hLine1.LineStyle = '--';
hLine2.LineStyle = ':';
```



Combine Different Types of Plots

Plot two data sets using a graph with two y-axes. Use a line plot for the data associated with the left *y*-axes. Use a stem plot for the data associated with the right *y*-axes.

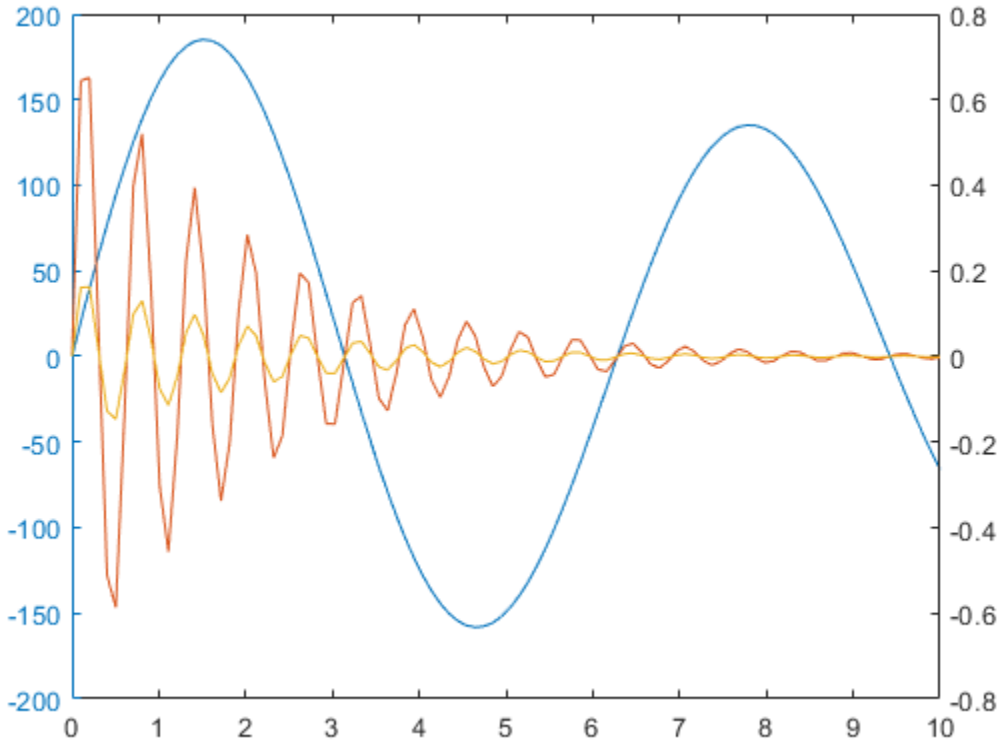
```
x = 0:0.1:10;
y1 = 200*exp(-0.05*x).*sin(x);
y2 = 0.8*exp(-0.5*x).*sin(10*x);

figure
plotyy(x,y1,x,y2,'plot','stem')
```



Use Right y-Axis for Two Data Sets

Plot three data sets using a graph with two *y*-axes. Plot one set of data associated with the left *y*-axis. Plot two sets of data associated with the right *y*-axis by using two-column matrices.

```
x = linspace(0,10);
y1 = 200*exp(-0.05*x).*sin(x);
y2 = 0.8*exp(-0.5*x).*sin(10*x);
y3 = 0.2*exp(-0.5*x).*sin(10*x);

figure
[hAx,hLine1,hLine2] = plotyy(x,y1,[x',x'],[y2',y3']);
```



Compatibility Considerations

plotyy is not recommended

Not recommended starting in R2016a

`plotyy` is not recommended. Use `yyaxis` instead. There are no plans to remove `plotyy`.

Starting in R2016a, use the `yyaxis` function to create charts with two y-axes. The `yyaxis` function has several advantages over the `plotyy` function.

- Unlike `plotyy`, the `yyaxis` function creates one `Axes` object with two y-axes. `plotyy` creates two overlaid `Axes` objects that can get out of sync.
- You can use `yyaxis` with any 2-D plotting function. `plotyy` is limited to working with plotting functions of the form `function(x,y)`. It does not work with other plotting functions, such as `errorbar`.

This table shows some typical usages of `plotyy` and how to update your code to use `yyaxis` instead.

Not Recommended	Recommended

`plotyy(x1,y1,x2,y2)`	`yyaxis left`
	`plot(x1,y1)`
	`yyaxis right`
	`plot(x2,y2)`

`plotyy(x1,y1,x2,y2,...`
`'function1','function2')`
	`yyaxis left`
	`function1(x1,y1)`
	`yyaxis right`
	`function2(x2,y2)`

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions

`linkaxes` | `linkprop` | `loglog` | `plot` | `semilogx` | `semilogy` | `yyaxis`

Topics

“Create Chart with Two y-Axes”
“Create Chart with Multiple x-Axes and y-Axes”

Introduced before R2006a

matlab.unittest.plugins Package

Summary of classes in MATLAB Plugins Interface

Description

Plugins customize a `TestRunner` object. The `matlab.unittest.plugins` package consists of the following customized MATLAB plugins.

Classes

Fundamental Plugin Related Interfaces

matlab.unittest.plugins.OutputStream	Interface that determines where to send text output
matlab.unittest.plugins.Parallelizable	Interface for plugins that support running tests in parallel
matlab.unittest.plugins.QualifyingPlugin	Interface for plugins that perform system-wide qualifications
matlab.unittest.plugins.TestRunnerPlugin	Plugin interface for extending TestRunner

Plugin Implementations

Diagnostic & Progress Information

matlab.unittest.plugins.DiagnosticsOutputPlugin	Plugin to direct diagnostics to output stream
matlab.unittest.plugins.DiagnosticsRecordingPlugin	Plugin to record diagnostics on test results
matlab.unittest.plugins.LoggingPlugin	Plugin to report diagnostic messages
matlab.unittest.plugins.TestRunProgressPlugin	Plugin that reports test run progress

Debugging & Qualification

matlab.unittest.plugins.DiagnosticsValidationPlugin	Plugin to help validate diagnostic code
matlab.unittest.plugins.FailOnWarningsPlugin	Plugin to fail tests that issue warnings
matlab.unittest.plugins.StopOnFailuresPlugin	Plugin to debug test failures

Output Formats & Continuous Integration

matlab.unittest.plugins.TAPPlugin	Plugin that produces Test Anything Protocol stream
matlab.unittest.plugins.XMLPlugin	Plugin that writes test results in XML format

Reporting

matlab.unittest.plugins.CodeCoveragePlugin	Plugin that produces a code coverage report
matlab.unittest.plugins.TestReportPlugin	Plugin that produces a test result report

Output Streams

matlab.unittest.plugins.ToFile	Output stream to write text output to file
matlab.unittest.plugins.ToStandardOutput	Output stream to display text information to screen
matlab.unittest.plugins.ToUniqueFile	Output stream to write text output to unique file

Packages

matlab.unittest.plugins.codecoverage	Summary of classes associated with code coverage report generation
matlab.unittest.plugins.diagnosticrecord	Summary of classes associated with MATLAB Plugin Diagnostic Records
matlab.unittest.plugins.plugindata	Summary of classes in MATLAB Plugin Data Interface
matlab.unittest.plugins.testreport	Summary of classes in MATLAB Plugin Test Report Generation

See Also

Topics

"Add Plugin to Test Runner"

"Write Plugins to Extend TestRunner"
"Create Custom Plugin"
"Run Tests in Parallel with Custom Plugin"
"Write Plugin to Save Diagnostic Details"

Introduced in R2013a

matlab.unittest.plugins.codecoverage Package

Summary of classes associated with code coverage report generation

Description

The `codecoverage` classes create formats used by the `CodeCoveragePlugin` to generate code coverage reports. The `matlab.unittest.plugins.codecoverage` package consists of the following MATLAB® classes.

Classes

matlab.unittest.plugins.codecoverage.CoverageReport	Format for code coverage report
matlab.unittest.plugins.codecoverage.CoberturaFormat	Format to create code coverage report using Cobertura XML format
matlab.unittest.plugins.codecoverage.ProfileReport	Format to create MATLAB Profiler Coverage Report

See Also

`matlab.unittest.plugins` | `matlab.unittest.plugins.CodeCoveragePlugin`

Introduced in R2017b

matlab.unittest.plugins.codecoverage.CoberturaFormat class

Package: `matlab.unittest.plugins.codecoverage`

Format to create code coverage report using Cobertura XML format

Construction

`reportFormat = CoberturaFormat(filename)` constructs a `CoberturaFormat` format and returns it as `reportFormat`. When used with the `CodeCoveragePlugin`, the code coverage results are saved to the file `filename`.

Typically, you construct this class implicitly by instantiating it when you create a `CodeCoveragePlugin` instance. For example, `p = CodeCoveragePlugin.forFile('ProductTest.m','Producing',CoberturaFormat('reportFile.xml'))` creates a plugin that produces a code coverage report for `ProductTest.m` in the Cobertura XML format named `reportFile.xml`.

Input Arguments

`filename` — Name of file
string | character vector

Name of file, specified as a string or character vector.

Example: `'product1_coverageReults.xml'`

See Also

`matlab.unittest.plugins.CodeCoveragePlugin` | `matlab.unittest.plugins.codecoverage.CoverageReport`

Introduced in R2017b

matlab.unittest.plugins.codecoverage.CoverageReport class

Package: `matlab.unittest.plugins.codecoverage`

Format for code coverage report

Description

To display code coverage metrics in the MATLAB browser, use an instance of the `CoverageReport` class with the plugin `CodeCoveragePlugin`.

Creation

Description

`reportFormat = CoverageReport` constructs a `CoverageReport` format. Use this report format with `matlab.unittest.plugins.CodeCoveragePlugin` to produce an HTML file in a temporary folder that contains the code coverage report.

`reportFormat = CoverageReport(reportFolder)` constructs a `CoverageReport` format. Use this report format with `matlab.unittest.plugins.CodeCoveragePlugin` to produce an HTML file in the specified folder that contains the code coverage report. Generating the report overwrites the contents of the folder specified by `reportFolder`.

`reportFormat = CoverageReport(___,'MainFile',mainFileName)` constructs a `CoverageReport` format with `mainFileName` as the name of the main HTML file containing the code coverage report.

Input Arguments

`reportFolder` — Folder containing code coverage report files
character vector | string scalar

Folder containing code coverage report files, specified as the path to a folder relative to the current folder or as an absolute path. If the folder does not exist, `CoverageReport` creates it.

`mainFileName` — Name of the main HTML file containing code coverage report
character vector | string scalar

Name of the main HTML file containing code coverage report, specified as a file name with a `.html` extension.

Properties

`MainFile` — Name of the main HTML file that contains the code coverage report
index.html (default)

Name of the main HTML file that contains the code coverage report, specified as a file name with a `.html` extension. To set the property, pass the file name to the constructor as a `'MainFile','`*`fileName`*`'` pair.

Example: myReport.html

Attributes:

`GetAccess`	`public`
`SetAccess`	`immutable`

Data Types: `char` | `string`

Examples

Create Test Report

Create a test report using the `CoverageReport` format. Specify the folder and HTML file name for the report.

Create a test for this function.

```
function res = add5(x)
   % Increment input by 5.
```

```
   if ~isa(x,'numeric')
      error('add5:InputMustBeNumeric','Input must be numeric.')
   end
   res = x + 5;
end
```

Derive a class from `matlab.unittest.TestCase` to implement test methods.

```
classdef Add5Test < matlab.unittest.TestCase
   methods (Test)
      function testDoubleOut(testCase)
         actOutput = add5(1);
         testCase.verifyClass(actOutput,'double')
      end
      function testNonNumericInput(testCase)
         testCase.verifyError(@()add5('0'),'add5:InputMustBeNumeric')
      end
   end
end
```

This test script creates a test suite for a single file, adds the `CodeCoveragePlugin` with the `CoverageReport` format to the test runner, and runs the tests. The format on the plugin specifies the name of the main test results HTML file and the folder containing the results.

```
import matlab.unittest.TestSuite
import matlab.unittest.TestRunner
import matlab.unittest.plugins.CodeCoveragePlugin
import matlab.unittest.plugins.codecoverage.CoverageReport

suite = TestSuite.fromFile('Add5Test.m');
runner = TestRunner.withNoPlugins;
runner.addPlugin(CodeCoveragePlugin.forFile('Add5Test.m', ...
   'Producing',CoverageReport('Add5Results', ...
   'MainFile','Add5TestResults.html')))
runner.run(suite)
```

```
Code coverage report has been saved to:
 C:\myWorkingFolder\Add5Results\Add5TestResults.html
```

See Also

`matlab.unittest.plugins.CodeCoveragePlugin` | `matlab.unittest.plugins.codecoverage.CoberturaFormat` | `runtests`

Introduced in R2019a

matlab.unittest.plugins.codecoverage.ProfileReport class

Package: `matlab.unittest.plugins.codecoverage`

Format to create MATLAB Profiler Coverage Report

Construction

`reportFormat = ProfileReport` constructs a `ProfileReport` format and returns it as `reportFormat`. When used with the `CodeCoveragePlugin`, the code coverage results open in a MATLAB Profiler Coverage Report.

Typically, you construct this class implicitly by instantiating it when you create a `CodeCoveragePlugin` instance. For example, `p = CodeCoveragePlugin.forFolder('myTests','Producing',ProfileReport)` creates a plugin that produces a MATLAB Profiler Coverage Report for code in the `myTests` folder.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

See Also

`matlab.unittest.plugins.CodeCoveragePlugin`

Introduced in R2017b

matlab.unittest.plugins.CodeCoveragePlugin class

Package: `matlab.unittest.plugins`

Plugin that produces a code coverage report

Description

To produce a line coverage report for MATLAB source code, add the `CodeCoveragePlugin` to the `TestRunner`. The testing framework runs the tests, and the resulting coverage report indicates the executed lines of code. The coverage report is based on source code located in one or more folders or packages. The source code must be on the MATLAB path and remain on the path during the test run.

The `CodeCoveragePlugin` uses the MATLAB profiler to determine which lines of code the tests execute. The tests and source code should not interact with the profiler. Before running a suite of tests, the plugin clears any data collected by the profiler.

Construction

Instantiate a `CodeCoveragePlugin` using one of its static methods.

To report on source code within one or more files, use the `forFile` static method. To report on source code within one or more folders, use the `forFolder` static method. To report on source code within one or more packages, use the `forPackage` static method.

Methods

forFile	Construct `CodeCoveragePlugin` for files
forFolder	Construct `CodeCoveragePlugin` for folders
forPackage	Construct `CodeCoveragePlugin` for packages

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Generate Code Coverage Report

In a new file, `quadraticSolver.m`, in your working folder, create the following function.

```
function roots = quadraticSolver(a,b,c)
% quadraticSolver returns solutions to the
% quadratic equation a*x^2 + b*x + c = 0.

checkInputs

roots(1) = (-b + sqrt(b^2 - 4*a*c)) / (2*a);
roots(2) = (-b - sqrt(b^2 - 4*a*c)) / (2*a);

    function checkInputs
        if ~isa(a,'numeric') || ~isa(b,'numeric') || ~isa(c,'numeric')
            error('quadraticSolver:InputMustBeNumeric', ...
                'Coefficients must be numeric.')
        end
    end
end
```

Create a test for the quadratic solver. In a `tests` package (a `+tests` subfolder), create `SolverTest.m` containing the following test class.

```
classdef SolverTest < matlab.unittest.TestCase
    % SolverTest tests solutions to the quadratic equation
    % a*x^2 + b*x + c = 0

    methods (Test)
        function testRealSolution(testCase)
            actSolution = quadraticSolver(1,-3,2);
            expSolution = [2,1];
            testCase.verifyEqual(actSolution,expSolution)
        end
        function testImaginarySolution(testCase)
```

```
            actSolution = quadraticSolver(1,2,10);
            expSolution = [-1+3i, -1-3i];
            testCase.verifyEqual(actSolution,expSolution)
        end
    end

end
```

At the command prompt from within your original working folder, create a test suite from the `tests` package.

```
import matlab.unittest.TestSuite
import matlab.unittest.TestRunner
import matlab.unittest.plugins.CodeCoveragePlugin

suite = TestSuite.fromPackage('tests');
```

Create a test runner.

```
runner = TestRunner.withTextOutput;
```

Add `CodeCoveragePlugin` to the runner and run the tests. Specify that the source code folder is your current working folder. If you have other source code files in your current working folder, they show up in the coverage report. The folder that contains the source code (`quadraticSolver.m`) must be on your MATLAB path.

```
runner.addPlugin(CodeCoveragePlugin.forFolder(pwd))
result = runner.run(suite);
```

```
Running tests.SolverTest
..
Done tests.SolverTest
__________
```

MATLAB creates a Code Coverage Report for the quadratic solver function in a temporary folder.

MATLAB® Code Coverage Report

Overall Summary

TOTAL FILES	LINE COVERAGE	EXECUTABLE LINES	EXECUTED LINES	MISSED LINES
1	80	5	4	1

Breakdown by Source

Root Folder - `H:\UnitTest_Examples\`

FILE NAME	LINE COVERAGE	EXECUTABLE LINES	EXECUTED LINES	MISSED LINES
quadraticSolver.m	80	5	4	1

Source Coverage

Source Filename - `H:\UnitTest_Examples\quadraticSolver.m`

```
1        function roots = quadraticSolver(a,b,c)
2           % quadraticSolver returns solutions to the
3           % quadratic equation a*x^2 + b*x + c = 0.
4
5     2     checkInputs
6
7     2     roots(1) = (-b + sqrt(b^2 - 4*a*c)) / (2*a);
8     2     roots(2) = (-b - sqrt(b^2 - 4*a*c)) / (2*a);
9
10          function checkInputs
11    2        if ~isa(a,'numeric') || ~isa(b,'numeric') || ~isa(c,'numeric')
12    0           error('quadraticSolver:InputMustBeNumeric', ...
13                   'Coefficients must be numeric.')
14           end
15        end
16     end
```

The `checkinputs` nested function does not have complete code coverage. Since the tests in `SolverTest.m` do not pass nonnumeric input to `quadraticSolver`, MATLAB does not exercise the code that throws an error if the inputs are not numeric. To address the missing coverage, add a test method to test the error condition.

Generate Report in Cobertura Format

In your working folder, create a file `quadraticSolver.m` with the following function.

```
function roots = quadraticSolver(a,b,c)
    % quadraticSolver returns solutions to the
    % quadratic equation a*x^2 + b*x + c = 0.
```

```
    checkInputs
    roots(1) = (-b + sqrt(b^2 - 4*a*c)) / (2*a);
    roots(2) = (-b - sqrt(b^2 - 4*a*c)) / (2*a);

    function checkInputs
        if ~isa(a,'numeric') || ~isa(b,'numeric') || ~isa(c,'numeric')
            error('quadraticSolver:InputMustBeNumeric', ...
                'Coefficients must be numeric.')
        end
    end
end
```

Create a test class for the quadratic solver in `ForFileSolverTest.m`.

```
classdef ForFileSolverTest < matlab.unittest.TestCase
    % SolverTest tests solutions to the quadratic equation
    % a*x^2 + b*x + c = 0

    methods (Test)
        function testRealSolution(testCase)
            actSolution = quadraticSolver(1,-3,2);
            expSolution = [2,1];
            testCase.verifyEqual(actSolution,expSolution)
        end
        function testImaginarySolution(testCase)
            actSolution = quadraticSolver(1,2,10);
            expSolution = [-1+3i, -1-3i];
            testCase.verifyEqual(actSolution,expSolution)
        end
    end
end
```

At the command prompt, create a test suite for `ForFileSolverTest.m`, and create a test runner.

```
import matlab.unittest.TestRunner

testFile = 'ForFileSolverTest.m';
suite = testsuite(testFile);
runner = TestRunner.withTextOutput;
```

Create a plugin that outputs a code coverage report for the source code in Cobertura format.

```
import matlab.unittest.plugins.CodeCoveragePlugin
import matlab.unittest.plugins.codecoverage.CoberturaFormat

sourceCodeFile = 'quadraticSolver.m';
reportFile = 'CoverageResults.xml';
reportFormat = CoberturaFormat(reportFile);
plugin = CodeCoveragePlugin.forFile(sourceCodeFile,'Producing',reportFormat);
```

Add the plugin to the test runner and run the tests.

```
runner.addPlugin(plugin);
result = runner.run(suite);
```

```
Running ForFileSolverTest
..
Done ForFileSolverTest
__________
```

Display the contents of the file with the code coverage results.

```
disp(fileread(reportFile))
```

```
<?xml version="1.0" encoding="utf-8"?>
<coverage branch-rate="NaN" branches-covered="NaN" branches-valid="NaN" complexity="NaN" line-rate="0.8" lin
   <sources>
      <source>C:\work\</source>
   </sources>
   <packages>
      <package branch-rate="NaN" complexity="NaN" line-rate="0.8" name="">
         <classes>
            <class branch-rate="NaN" complexity="NaN" filename="quadraticSolver.m" line-rate="0.8" name="qua
               <methods/>
               <lines>
                  <line branch="NaN" condition-coverage="NaN" hits="2" number="4"/>
                  <line branch="NaN" condition-coverage="NaN" hits="2" number="5"/>
                  <line branch="NaN" condition-coverage="NaN" hits="2" number="6"/>
                  <line branch="NaN" condition-coverage="NaN" hits="2" number="9"/>
                  <line branch="NaN" condition-coverage="NaN" hits="0" number="10"/>
               </lines>
            </class>
         </classes>
      </package>
   </packages>
</coverage>
```

See Also

`matlab.unittest.plugins.TestRunnerPlugin` | `profile`

Introduced in R2014b

matlab.unittest.plugins.CodeCoveragePlugin.forFile

Class: `matlab.unittest.plugins.CodeCoveragePlugin`
Package: `matlab.unittest.plugins`

Construct `CodeCoveragePlugin` for files

Syntax

```
matlab.unittest.plugins.CodeCoveragePlugin.forFile(file,'Producing',
reportFormat)
```

Description

`matlab.unittest.plugins.CodeCoveragePlugin.forFile(file,'Producing', reportFormat)` creates a plugin that produces a code coverage report for one or more `.m` or `.mlx` files.

Input Arguments

`file` — Name of files to analyze
character vector | cell array of character vectors | string scalar | string array

Name of files to analyze, specified as a character vector, cell array of character vectors, string scalar, or string array. `file` is the absolute or relative path to one or more `.m` or `.mlx` files.

Example: `'../thisTest.m'`

Example: `{'Test_featureB.m','Test_featureA.m'}`

Data Types: `char` | `string`

`reportFormat` — Report format
instance of a code coverage report format class

Report format, specified as an instance of the `matlab.unittest.plugins.codecoverage.CoverageReport` or the `matlab.unittest.plugins.codecoverage.CoberturaFormat` class.

Example: `matlab.unittest.plugins.codecoverage.CoberturaFormat('CoverageResults.xml')`

Examples

Generate Report in Cobertura Format

In your working folder, create a file `quadraticSolver.m` with the following function.

```
function roots = quadraticSolver(a,b,c)
    % quadraticSolver returns solutions to the
    % quadratic equation a*x^2 + b*x + c = 0.
    checkInputs
    roots(1) = (-b + sqrt(b^2 - 4*a*c)) / (2*a);
    roots(2) = (-b - sqrt(b^2 - 4*a*c)) / (2*a);

    function checkInputs
        if ~isa(a,'numeric') || ~isa(b,'numeric') || ~isa(c,'numeric')
            error('quadraticSolver:InputMustBeNumeric', ...
                'Coefficients must be numeric.')
        end
    end
end
```

Create a test class for the quadratic solver in `ForFileSolverTest.m`.

```
classdef ForFileSolverTest < matlab.unittest.TestCase
    % SolverTest tests solutions to the quadratic equation
    % a*x^2 + b*x + c = 0

    methods (Test)
        function testRealSolution(testCase)
            actSolution = quadraticSolver(1,-3,2);
            expSolution = [2,1];
            testCase.verifyEqual(actSolution,expSolution)
        end
        function testImaginarySolution(testCase)
```

```
            actSolution = quadraticSolver(1,2,10);
            expSolution = [-1+3i, -1-3i];
            testCase.verifyEqual(actSolution,expSolution)
        end
    end
end
```

At the command prompt, create a test suite for `ForFileSolverTest.m`, and create a test runner.

```
import matlab.unittest.TestRunner

testFile = 'ForFileSolverTest.m';
suite = testsuite(testFile);
runner = TestRunner.withTextOutput;
```

Create a plugin that outputs a code coverage report for the source code in Cobertura format.

```
import matlab.unittest.plugins.CodeCoveragePlugin
import matlab.unittest.plugins.codecoverage.CoberturaFormat

sourceCodeFile = 'quadraticSolver.m';
reportFile = 'CoverageResults.xml';
reportFormat = CoberturaFormat(reportFile);
plugin = CodeCoveragePlugin.forFile(sourceCodeFile,'Producing',reportFormat);
```

Add the plugin to the test runner and run the tests.

```
runner.addPlugin(plugin);
result = runner.run(suite);
```

```
Running ForFileSolverTest
..
Done ForFileSolverTest
__________
```

Display the contents of the file with the code coverage results.

```
disp(fileread(reportFile))
```

```
<?xml version="1.0" encoding="utf-8"?>
<coverage branch-rate="NaN" branches-covered="NaN" branches-valid="NaN" complexity="NaN" line-rate="0.8" lin
   <sources>
      <source>C:\work\</source>
   </sources>
```

```
   <packages>
      <package branch-rate="NaN" complexity="NaN" line-rate="0.8" name="">
         <classes>
            <class branch-rate="NaN" complexity="NaN" filename="quadraticSolver.m" line-rate="0.8" name="qua
               <methods/>
               <lines>
                  <line branch="NaN" condition-coverage="NaN" hits="2" number="4"/>
                  <line branch="NaN" condition-coverage="NaN" hits="2" number="5"/>
                  <line branch="NaN" condition-coverage="NaN" hits="2" number="6"/>
                  <line branch="NaN" condition-coverage="NaN" hits="2" number="9"/>
                  <line branch="NaN" condition-coverage="NaN" hits="0" number="10"/>
               </lines>
            </class>
         </classes>
      </package>
   </packages>
</coverage>
```


See Also

matlab.unittest.plugins.CodeCoveragePlugin.forFolder | matlab.unittest.plugins.CodeCoveragePlugin.forPackage | matlab.unittest.plugins.codecoverage.CoberturaFormat | matlab.unittest.plugins.codecoverage.CoverageReport

Introduced in R2017b

matlab.unittest.plugins.CodeCoveragePlugin.forFolder

Class: `matlab.unittest.plugins.CodeCoveragePlugin`
Package: `matlab.unittest.plugins`

Construct `CodeCoveragePlugin` for folders

Syntax

```
matlab.unittest.plugins.CodeCoveragePlugin.forFolder(folder)
matlab.unittest.plugins.CodeCoveragePlugin.forFolder(folder,
Name,Value)
```

Description

`matlab.unittest.plugins.CodeCoveragePlugin.forFolder(folder)` creates a plugin that produces a code coverage report for one or more folders. The plugin reports on the source code inside `folder`.

`matlab.unittest.plugins.CodeCoveragePlugin.forFolder(folder, Name,Value)` produces a code coverage report with additional options specified by one or more `Name,Value` pair arguments. For example, `matlab.unittest.plugins.CodeCoveragePlugin.forFolder(pwd,'IncludingSubfolders',true)` produces a code coverage report for source code in the current folder and its subfolders.

Input Arguments

`folder` — Locations of folder containing source code
character vector | cell array of character vectors | string array

Locations of folder containing source code, specified as a character vector, a cell array of character vectors, or a string array. `folder` is the absolute or relative path to one or

more folders. If you specify multiple folders, MATLAB opens a profile coverage report for each folder.

The source code folders must be on the MATLAB path and remain on the path during the test run.

Example: `'C:\projects\myproj'`

Example: `pwd`

Example: `{'C:\projects\myprojA','myprojB'}`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `matlab.unittest.plugins.CodeCoveragePlugin.forFolder(pwd,'IncludingSubfolders',true)` produces a code coverage report for source code in the current folder and its subfolders.

`IncludingSubfolders` — Include source code in subfolders
`false` (default) | `true`

Setting to include source code in subfolders of `folder`, specified by `false` or `true`. By default, `CodeCoveragePlugin` does not report on source code in subfolders.

Data Types: `logical`

`Producing` — Report format
`matlab.unittest.plugins.codecoverage.CoverageReport` instance (default) | instance of class in `matlab.unittest.plugins.codecoverage` package

Report format, specified as either a `matlab.unittest.plugins.codecoverage.CoverageReport` instance or an instance of a different class in the `matlab.unittest.plugins.codecoverage` package. By default, the report format is `matlab.unittest.plugins.codecoverage.CoverageReport`, which displays a MATLAB Code Coverage Report.

Example:
`matlab.unittest.plugins.codecoverage.CoberturaFormat('CoverageResults.xml')`

Examples

Generate Code Coverage Report

In a new file, `quadraticSolver.m`, in your working folder, create the following function.

```
function roots = quadraticSolver(a,b,c)
% quadraticSolver returns solutions to the
% quadratic equation a*x^2 + b*x + c = 0.

checkInputs

roots(1) = (-b + sqrt(b^2 - 4*a*c)) / (2*a);
roots(2) = (-b - sqrt(b^2 - 4*a*c)) / (2*a);

    function checkInputs
        if ~isa(a,'numeric') || ~isa(b,'numeric') || ~isa(c,'numeric')
            error('quadraticSolver:InputMustBeNumeric', ...
                'Coefficients must be numeric.')
        end
    end
end
```

Create a test for the quadratic solver. In a `tests` package (a `+tests` subfolder), create `SolverTest.m` containing the following test class.

```
classdef SolverTest < matlab.unittest.TestCase
    % SolverTest tests solutions to the quadratic equation
    % a*x^2 + b*x + c = 0

    methods (Test)
        function testRealSolution(testCase)
            actSolution = quadraticSolver(1,-3,2);
            expSolution = [2,1];
            testCase.verifyEqual(actSolution,expSolution)
        end
        function testImaginarySolution(testCase)
```

```
            actSolution = quadraticSolver(1,2,10);
            expSolution = [-1+3i, -1-3i];
            testCase.verifyEqual(actSolution,expSolution)
        end
    end

end
```

At the command prompt from within your original working folder, create a test suite from the `tests` package.

```
import matlab.unittest.TestSuite
import matlab.unittest.TestRunner
import matlab.unittest.plugins.CodeCoveragePlugin

suite = TestSuite.fromPackage('tests');
```

Create a test runner.

```
runner = TestRunner.withTextOutput;
```

Add `CodeCoveragePlugin` to the runner and run the tests. Specify that the source code folder is your current working folder. If you have other source code files in your current working folder, they show up in the coverage report. The folder that contains the source code (`quadraticSolver.m`) must be on your MATLAB path.

```
runner.addPlugin(CodeCoveragePlugin.forFolder(pwd))
result = runner.run(suite);
```

```
Running tests.SolverTest
..
Done tests.SolverTest
__________
```

MATLAB creates a Code Coverage Report for the quadratic solver function in a temporary folder.

MATLAB® Code Coverage Report

Overall Summary

TOTAL FILES	LINE COVERAGE	EXECUTABLE LINES	EXECUTED LINES	MISSED LINES
1	80	5	4	1

Breakdown by Source

Root Folder - `H:\UnitTest_Examples\`

FILE NAME	LINE COVERAGE	EXECUTABLE LINES	EXECUTED LINES	MISSED LINES
quadraticSolver.m	80	5	4	1

Source Coverage

Source Filename - `H:\UnitTest_Examples\quadraticSolver.m`

```
1        function roots = quadraticSolver(a,b,c)
2           % quadraticSolver returns solutions to the
3           % quadratic equation a*x^2 + b*x + c = 0.
4
5     2     checkInputs
6
7     2     roots(1) = (-b + sqrt(b^2 - 4*a*c)) / (2*a);
8     2     roots(2) = (-b - sqrt(b^2 - 4*a*c)) / (2*a);
9
10          function checkInputs
11    2        if ~isa(a,'numeric') || ~isa(b,'numeric') || ~isa(c,'numeric')
12    0           error('quadraticSolver:InputMustBeNumeric', ...
13                   'Coefficients must be numeric.')
14           end
15        end
16     end
```

The `checkinputs` nested function does not have complete code coverage. Since the tests in `SolverTest.m` do not pass nonnumeric input to `quadraticSolver`, MATLAB does not exercise the code that throws an error if the inputs are not numeric. To address the missing coverage, add a test method to test the error condition.

See Also

`matlab.unittest.plugins.CodeCoveragePlugin.forFile` | `matlab.unittest.plugins.CodeCoveragePlugin.forPackage`

Introduced in R2014b

matlab.unittest.plugins.CodeCoveragePlugin.forPackage

Class: `matlab.unittest.plugins.CodeCoveragePlugin`
Package: `matlab.unittest.plugins`

Construct `CodeCoveragePlugin` for packages

Syntax

```
matlab.unittest.plugins.CodeCoveragePlugin.forPackage(package)
matlab.unittest.plugins.CodeCoveragePlugin.forPackage(package,
Name,Value)
```

Description

`matlab.unittest.plugins.CodeCoveragePlugin.forPackage(package)` creates a plugin that produces a code coverage report for one or more packages. The plugin reports on the source code inside `package`.

`matlab.unittest.plugins.CodeCoveragePlugin.forPackage(package, Name,Value)` produces a code coverage report with additional options specified by one or more `Name,Value` pair arguments. For example, `matlab.unittest.plugins.CodeCoveragePlugin.forPackage('myprojA','IncludingSubpackages',true)` produces a code coverage report for source code in the `myprojA` package and its subpackages.

Input Arguments

package — Names of package containing source code
character vector | cell array of character vectors | string array

Names of package containing source code, specified as a character vector, a cell array of character vectors, or a string array. If you specify multiple packages, MATLAB opens a profile coverage report for each package.

The parent folder of the top-level source code package must be on the MATLAB path and remain on the path during the test run.

Example: `'myproject.controller'`

Example: `{'myprojA','myprojB'}`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `matlab.unittest.plugins.CodeCoveragePlugin.forPackage('myprojA','IncludingSubpackages',true)` produces a code coverage report for source code in the `myprojA` package and its subpackages.

`IncludingSubpackages` — Include source code in subpackages
`false` (default) | `true`

Setting to include source code in subpackages of `package`, specified by `false` or `true`. By default, `CodeCoveragePlugin` does not report on source code in subpackages.

Data Types: `logical`

`Producing` — Report format
`matlab.unittest.plugins.codecoverage.CoverageReport` instance (default) | instance of class in `matlab.unittest.plugins.codecoverage` package

Report format, specified as either a `matlab.unittest.plugins.codecoverage.CoverageReport` instance or an instance of a different class in the `matlab.unittest.plugins.codecoverage` package. By default, the report format is `matlab.unittest.plugins.codecoverage.CoverageReport`, which displays a MATLAB Code Coverage Report.

Example: `matlab.unittest.plugins.codecoverage.CoberturaFormat('CoverageResults.xml')`

See Also

`matlab.unittest.plugins.CodeCoveragePlugin.forFile` | `matlab.unittest.plugins.CodeCoveragePlugin.forFolder`

Introduced in R2014b

matlab.unittest.plugins.DiagnosticsOutputPlugin class

Package: `matlab.unittest.plugins`

Plugin to direct diagnostics to output stream

Description

The `DiagnosticsOutputPlugin` class creates a plugin to direct diagnostics to an output stream. To configure the type of diagnostics and detail level that the testing framework outputs, add this plugin to a `TestRunner` instance.

Construction

`matlab.unittest.plugins.DiagnosticsOutputPlugin` creates a plugin that directs diagnostics for failed events and for events logged at the `Verbosity.Terse` level to the `ToStandardOutput` stream.

`matlab.unittest.plugins.DiagnosticsOutputPlugin(stream)` redirects diagnostics to the specified output stream. For example, you can redirect output to a stream creating using `ToFile`.

`matlab.unittest.plugins.DiagnosticsOutputPlugin(___,Name,Value)` creates a plugin with additional options specified by one or more `Name,Value` pair arguments. For example, `DiagnosticsOutputPlugin('LoggingLevel',4,'IncludingPassingDiagnostics',true)` creates a plugin that displays diagnostics logged at any level and also displays passing diagnostics.

Input Arguments

`stream` — Output location
`ToStandardOutput` (default) | instance of `matlab.unittest.plugins.OutputStream`

Output location, specified as an instance of the `OutputStream` class. The plugin directs diagnostic information to the specified location. By default, the plugin uses the `matlab.unittest.plugins.ToStandardOutput` stream.

Example: `matlab.unittest.plugins.ToFile('myFile.txt')`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `DiagnosticsOutputPlugin('IncludingPassingDiagnostics',true,'OutputDetail',4)` creates a plugin that includes passing diagnostics and displays diagnostics at a verbose detail level.

`ExcludingFailureDiagnostics` — Exclude diagnostics from failing events
false (default) | true

Whether to exclude diagnostics from failing events, specified as `false` or `true`. By default the plugin includes diagnostics from failing events.

Data Types: `logical`

`IncludingPassingDiagnostics` — Include passing event diagnostics
`false` (default) | `true`

Whether to include passing event diagnostics, specified as `false` or `true`. By default the plugin does not include diagnostics from passing events.

Data Types: `logical`

`LoggingLevel` — Maximum level of logged diagnostics
1 (default) | 0 | 2 | 3 | 4 | `matlab.unittest.Verbosity` enumeration | enumeration name as string or char vector

Maximum level at which logged diagnostics are included by the plugin instance, specified as an integer value from 0 through 4, a `matlab.unittest.Verbosity` enumeration object, or a string scalar or character vector corresponding to one of the predefined enumeration member names. The plugin includes diagnostics that are logged at this level and below. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration.

Numeric Representation	Enumeration Member Name	Verbosity Description
0	None	No information
1	Terse	Minimal information
2	Concise	Moderate amount of information
3	Detailed	Some supplemental information
4	Verbose	Lots of supplemental information

By default the plugin includes diagnostics logged at the `matlab.unittest.Verbosity.Terse` level (level 1). To exclude logged diagnostics, specify `LoggingLevel` as `Verbosity.None` (level 0).

Logged diagnostics are diagnostics that you supply to the testing framework with a call to the `log` or `log` method.

`OutputDetail` — Detail level for reported events
3 (default) | 0 | 1 | 2 | 4 | `matlab.unittest.Verbosity` enumeration | enumeration name as string or char vector

Detail level for reported events, specified as an integer value from 0 through 4, a `matlab.unittest.Verbosity` enumeration object, or a string scalar or character vector corresponding to one of the predefined enumeration member names. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration.

The plugin reports passing, failing, and logged events with the amount of detail specified by `OutputDetail`. By default the plugin records events at the `matlab.unittest.Verbosity.Detailed` level (level 3).

Numeric Representation	Enumeration Member Name	Verbosity Description
0	None	No information
1	Terse	Minimal information
2	Concise	Moderate amount of information
3	Detailed	Some supplemental information

Numeric Representation	Enumeration Member Name	Verbosity Description
4	`Verbose`	Lots of supplemental information

Properties

`ExcludeFailureDiagnostics` — Indicator if diagnostics for failing events are excluded
`false` (default) | `true`

This property is read-only.

Indicator if diagnostics for failing events are excluded, specified as `false` or `true` (`logical` 0 or 1). By default, `ExcludeFailureDiagnostics` is `false` and the diagnostics from failing events are included in the output. To exclude diagnostics from failing events from the output, specify `ExcludeFailureDiagnostics` as `true` during plugin construction.

`IncludePassingDiagnostics` — Indicator if diagnostics for passing events are included
`false` (default) | `true`

This property is read-only.

Indicator if diagnostics for passing events are included, specified as `false` or `true` (`logical` 0 or 1). By default, `IncludePassingDiagnostics` is `false` and the diagnostics from passing events are excluded from the output. To include diagnostics from passing events in the output, specify `IncludePassingDiagnostics` as `true` during plugin construction.

`LoggingLevel` — Maximum verbosity level for logged diagnostics included by the plugin
`matlab.unittest.Verbosity.Terse` (default) | `matlab.unittest.Verbosity` enumeration object

This property is read-only.

Maximum verbosity level for logged diagnostics included by the plugin, returned as a `matlab.unittest.Verbosity` enumeration object. The plugin includes diagnostics that

are logged at this level and below. By default this property value is `matlab.unittest.Verbosity.Terse`. You can specify a different logging level during plugin construction.

Logged diagnostics are diagnostics that you supply to the testing framework with a call to the `log` or `log` method.

`OutputDetail` — Detail level for reported events
`Detailed` (default) | matlab.unittest.Verbosity instance

This property is read-only.

Detail level for reported events, returned as a `matlab.unittest.Verbosity` enumeration object. By default this property value is `matlab.unittest.Verbosity.Detailed`. You can specify a different output detail level during plugin construction.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Create Plugin to Customize Diagnostics Display

Create a file `ExampleDiagOutputTest.m` containing the following test class.

```
classdef ExampleDiagOutputTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase)
            import matlab.unittest.Verbosity
            testCase.log(Verbosity.Detailed,'Testing failing event')
            testCase.verifyEqual(42,13,'42 == 13')
        end
        function testTwo(testCase)
            testCase.log(3,'Testing passing event')
            testCase.verifyTrue(true,'true is true')
        end
    end
end
```

Create a test suite from the `ExampleDiagOutputTest` class. Create a test runner with no plugins.

```
import matlab.unittest.TestRunner
import matlab.unittest.TestSuite
import matlab.unittest.Verbosity
import matlab.unittest.plugins.DiagnosticsOutputPlugin

suite = TestSuite.fromClass(?ExampleDiagOutputTest);
runner = TestRunner.withNoPlugins();
```

Create a default `DiagnosticsOutputPlugin`, add it to the runner, and run the tests.

```
plugin = DiagnosticsOutputPlugin;
runner.addPlugin(plugin);
result = runner.run(suite);
```

```
================================================================================
Verification failed in ExampleDiagOutputTest/testOne.

    ----------------
    Test Diagnostic:
    ----------------
    42 == 13

    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyEqual failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
            Actual    Expected    Error     RelativeError
            ______    ________    _____    _______________

              42         13        29      2.23076923076923

    Actual Value:
        42
    Expected Value:
        13

    ------------------
    Stack Information:
    ------------------
    In C:\work\ExampleDiagOutputTest.m (ExampleDiagOutputTest.testOne) at 5
================================================================================
Failure Summary:

     Name                           Failed  Incomplete  Reason(s)
    ================================================================================
     ExampleDiagOutputTest/testOne    X                 Failed by verification.
```

Create another test runner and a `DiagnosticsOutputPlugin` that displays diagnostics, including passing diagnostics, at a `Terse` level, and displays diagnostics that are logged at a `Detailed` level or lower. Add it to the runner and rerun the tests.

```
runner = TestRunner.withNoPlugins();
plugin = DiagnosticsOutputPlugin('OutputDetail',Verbosity.Terse, ...
    'LoggingLevel',3,'IncludingPassingDiagnostics',true);
runner.addPlugin(plugin);
result = runner.run(suite);
```

```
[Detailed] Diagnostic logged (2018-04-13 13:47:34): Testing failing event

FAIL: ExampleDiagOutputTest/testOne in ExampleDiagOutputTest.testOne at 6 :: verifyEqual failed.

[Detailed] Diagnostic logged (2018-04-13 13:47:34): Testing passing event

PASS: ExampleDiagOutputTest/testTwo in ExampleDiagOutputTest.testTwo at 10 :: verifyTrue passed.
```

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.Verbosity` | `matlab.unittest.plugins.OutputStream`

Introduced in R2018b

matlab.unittest.plugins.DiagnosticsRecordingPlugin class

Package: `matlab.unittest.plugins`

Plugin to record diagnostics on test results

Description

The `DiagnosticsRecordingPlugin` enables programmatic access to the diagnostic information from unit tests.

This class creates a plugin to record diagnostics on test results. The `TestRunner` records these diagnostics as `DiagnosticRecord` arrays in the `Details` property of the `TestResult` object. Each element of the `DiagnosticRecord` array corresponds to an event in an individual test.

If you run tests with the `runtests` function or the `run` method of `TestSuite` or `TestCase`, the test framework uses this plugin by default. Also, if you run performance tests with the `runperf` function or the `run` method of `TimeExperiment`, the test framework uses this plugin by default.

Construction

`matlab.unittest.plugins.DiagnosticsRecordingPlugin` creates a plugin to record diagnostics on test results. By default, the `DiagnosticsRecordingPlugin` records qualification failures and logged events.

`matlab.unittest.plugins.DiagnosticsRecordingPlugin(Name,Value)` creates a plugin with additional options specified by one or more `Name,Value` pair arguments. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `matlab.unittest.plugins.DiagnosticsRecordingPlugin('IncludingPassingDiagnostics',true)` creates a plugin that records passing diagnostics in addition to diagnostics for failing qualifications and logged events.

`IncludingPassingDiagnostics` — Indicator to record diagnostics from passing tests
`false` (default) | `true`

Whether to record diagnostics from passing tests, specified as `false` or `true`. By default the plugin does not record diagnostics from passing tests.

Data Types: `logical`

`LoggingLevel` — Maximum level at which logged diagnostics are recorded
1 (default) | 0 | 2 | 3 | 4 | `matlab.unittest.Verbosity` enumeration

Maximum level at which logged diagnostics are recorded by the plugin instance, specified as an integer value from 0 through 4, or as a `matlab.unittest.Verbosity` enumeration object. The plugin records diagnostics that are logged at this level and below. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration.

By default the plugin records diagnostics logged at the `matlab.unittest.Verbosity.Terse` level (level 1). To exclude logged diagnostics, specify `LoggingLevel` as `Verbosity.None` (level 0).

Logged diagnostics are diagnostics that you supply to the testing framework with a call to the `log` or `log` method.

Numeric Representation	Enumeration Member Name	Verbosity Description
0	None	No information
1	Terse	Minimal information
2	Concise	Moderate amount of information
3	Detailed	Some supplemental information
4	Verbose	Lots of supplemental information

`OutputDetail` — Detail level for recorded events
3 (default) | 0 | 1 | 2 | 4 | `matlab.unittest.Verbosity` enumeration

Detail level for recorded events, specified as an integer value from 0 through 4, or as a `matlab.unittest.Verbosity` enumeration object. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration.

The plugin records passing, failing, and logged events with the amount of detail specified by `OutputDetail`. By default the plugin records events at the `matlab.unittest.Verbosity.Detailed` level (level 3).

Numeric Representation	Enumeration Member Name	Verbosity Description
0	None	No information
1	Terse	Minimal information
2	Concise	Moderate amount of information
3	Detailed	Some supplemental information
4	Verbose	Lots of supplemental information

Properties

`IncludePassingDiagnostics` — Indicator if diagnostics for passing events are recorded
`false` (default) | `true`

This property is read-only.

Indicator if diagnostics for passing events are recorded, returned as `false` or `true`. This property is `false` by default. You can specify it as `true` during construction.

Data Types: `logical`

`LoggingLevel` — Maximum verbosity level for logged diagnostics
`matlab.unittest.Verbosity` enumeration object

This property is read-only.

Maximum verbosity level for logged diagnostics recorded by the plugin, returned as a `matlab.unittest.Verbosity` enumeration object. The plugin records diagnostics that are logged at this level and below. By default this property value is `matlab.unittest.Verbosity.Terse`. You can specify a different logging level during plugin construction.

Logged diagnostics are diagnostics that you supply to the testing framework with a call to the `log` or `log` method.

`OutputDetail` — Display level for event details
`matlab.unittest.Verbosity` enumeration object

This property is read-only.

Display level for event details, returned as a `matlab.unittest.Verbosity` enumeration object. The plugin displays passing, failing, and logged events with the amount of detail specified by `OutputDetail`. By default this property value is `matlab.unittest.Verbosity.Detailed`. You can specify a different output detail during plugin construction.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Record Diagnostics on Test Result

In your working folder, create a file, `ExampleTest.m`, containing the following test class. The intent of this test is to illustrate how to use the `DiagnosticsRecordingPlugin` plugin, and it is not intended to be a representative unit test.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods (Test)
        function testA(testCase)
            testCase.log(1,'Terse log message')     % logs
            testCase.log(3,'Detailed log message')     % logs
            testCase.verifyEqual(3+2,5)             % passes
            testCase.assumeTrue(true)               % passes
            testCase.verifyGreaterThan(5, 9)        % fails
            testCase.assertEqual(3.14,pi)           % fails/incomplete
        end
        function testB(testCase)
            % This test contains an intentional error - passing a character
            % instead of a variable to the ones function.
            a = [1 2];
            testCase.verifyEqual(ones('a'),[1 1]);  % errors
        end
    end
end
```

At the command prompt, create a test suite from the `ExampleTest` class.

```
suite   = testsuite('ExampleTest');
```

Create a test runner with no plugins. This code creates a silent runner and provides you with complete control over the installed plugins. Add a `DiagnosticsRecordingPlugin` to the test runner.

```
import matlab.unittest.TestRunner;
import matlab.unittest.plugins.DiagnosticsRecordingPlugin;

runner = TestRunner.withNoPlugins;
runner.addPlugin(DiagnosticsRecordingPlugin);
```

Run the tests.

```
results = runner.run(suite);
```

Display the result from the second test. The test fails and is incomplete.

```
results(2)
```

```
ans =

  TestResult with properties:
```

```
          Name: 'ExampleTest/testB'
        Passed: 0
        Failed: 1
    Incomplete: 1
      Duration: 7.8912e-04
       Details: [1×1 struct]

Totals:
   0 Passed, 1 Failed, 1 Incomplete.
   0.00078912 seconds testing time.
```

Index into the diagnostic record to display more information.

```
results(2).Details.DiagnosticRecord
```

```
ans =

  ExceptionDiagnosticRecord with properties:

                        Event: 'ExceptionThrown'
                   EventScope: TestMethod
                 EventLocation: 'ExampleTest/testB'
                     Exception: [1×1 MException]
    AdditionalDiagnosticResults: [1×0 matlab.unittest.diagnostics.DiagnosticResult]
                        Stack: [1×1 struct]
                       Report: 'Error occurred in ExampleTest/testB and it did not run to completion…'
```

The test throws an uncaught exception.

Collect the diagnostic records for the first test, `testA`.

```
testA_records = results(1).Details.DiagnosticRecord
```

```
testA_records =

  1×3 heterogeneous DiagnosticRecord (LoggedDiagnosticRecord, QualificationDiagnosticRe

    Event
    EventScope
    EventLocation
    Stack
    Report
```

View the events that the plugin recorded for `testA`.

```
{testA_records.Event}'
```

```
ans =

  3×1 cell array

    {'DiagnosticLogged'  }
    {'VerificationFailed'}
    {'AssertionFailed'   }
```

The plugin records the message logged at a `Terse` level of verbosity, and the verification and assertion failures.

Create a plugin that records messages at all verbosity levels and includes passing diagnostics. Rerun the tests and collect the diagnostic records for `testA`.

```
runner = TestRunner.withNoPlugins;
runner.addPlugin(DiagnosticsRecordingPlugin(...
    'IncludingPassingDiagnostics',true,'OutputDetail',4,'LoggingLevel',4));
results = runner.run(suite);
testA_records = results(1).Details.DiagnosticRecord;
```

View the events that the plugin recorded for `testA`.

```
{testA_records.Event}'
```

```
ans =

  6×1 cell array

    {'DiagnosticLogged'  }
    {'DiagnosticLogged'  }
    {'VerificationPassed'}
    {'AssumptionPassed'  }
    {'VerificationFailed'}
    {'AssertionFailed'   }
```

The plugin records diagnostic information for all the qualifications and calls to the `log` method.

Select all the records with failing event diagnostics.

```
failedRecords = selectFailed(testA_records)
```

```
failedRecords =

  1×2 QualificationDiagnosticRecord array with properties:

    Event
```

```
    EventScope
    EventLocation
    TestDiagnosticResults
    FrameworkDiagnosticResults
    AdditionalDiagnosticResults
    Stack
    Report
```

Select all the records with passing event diagnostics and display the report for the first record.

```
passedRecords = selectPassed(testA_records);
passedRecords(1).Report
```

```
ans =

    'Verification passed in ExampleTest/testA.

         ---------------------
         Framework Diagnostic:
         ---------------------
         verifyEqual passed.
         --> The values are equal using "isequaln".

         Actual Value:
              5
         Expected Value:
              5

         ------------------
         Stack Information:
         ------------------
         In C:\work\ExampleTest.m (ExampleTest.testA) at 6'
```

Select all the records for incomplete events.

```
incompleteRecords = selectIncomplete(testA_records)
```

```
incompleteRecords =

  QualificationDiagnosticRecord with properties:

                        Event: 'AssertionFailed'
                   EventScope: TestMethod
                EventLocation: 'ExampleTest/testA'
         TestDiagnosticResults: [1×0 matlab.unittest.diagnostics.DiagnosticResult]
     FrameworkDiagnosticResults: [1×1 matlab.unittest.diagnostics.DiagnosticResult]
```

```
    AdditionalDiagnosticResults: [1×0 matlab.unittest.diagnostics.DiagnosticResult]
                          Stack: [1×1 struct]
                         Report: 'Assertion failed in ExampleTest/testA and it did not run to completion…'
```

Since this event is an assertion failure, the framework also returns this record with the failing diagnostics as `failedRecords(2)`.

Select all the records with logged events and display the logged messages.

```
loggedRecords = selectLogged(testA_records);
{loggedRecords.Report}'
```

```
ans =

  2×1 cell array

    {'[Terse] Diagnostic logged (2018-04-12 13:15:23): Terse log message'      }
    {'[Detailed] Diagnostic logged (2018-04-12 13:15:23): Detailed log message'}
```

See Also

`matlab.unittest.plugins` | `matlab.unittest.plugins.DiagnosticsOutputPlugin` | `matlab.unittest.plugins.LoggingPlugin` | `matlab.unittest.plugins.diagnosticrecord` | `matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord`

Topics

"Programmatically Access Test Diagnostics"

Introduced in R2016a

selectFailed

Class: `matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord`
Package: `matlab.unittest.plugins.diagnosticrecord`

Return diagnostic records for failed events

Syntax

```
selectedRecords = selectFailed(records)
```

Description

`selectedRecords = selectFailed(records)` returns the diagnostic records for failed events as an array of `matlab.unittest.plugins.diagnosticrecord.QualificationDiagnosticRecord` and `matlab.unittest.plugins.diagnosticrecord.ExceptionDiagnosticRecord` instances.

Failed events are events that result in a failure on the `TestResult`. These events include verification failures, assertion failures, and uncaught `MException` objects.

Input Arguments

`records` — Recorded diagnostics on test result
array of `matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord` instances

Recorded diagnostics on a test result, specified as an array of `matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord` instances. Access recorded diagnostics via the `DiagnosticRecord` field in the `Details` property on `TestResult`. For example, if your test results are stored in the variable `results`, find the recorded diagnostics for the second test by invoking `records = result(2).Details.DiagnosticRecord`.

Examples

Select Diagnostics on Test Result

In your working folder, create a file, `ExampleTest.m`, containing the following test class. The intent of this test is to illustrate how to use the `DiagnosticsRecordingPlugin` plugin, and it is not intended to be a representative unit test.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods (Test)
        function testA(testCase)
            testCase.log(1,'Terse log message')     % logs
            testCase.log(3,'Detailed log message')     % logs
            testCase.verifyEqual(3+2,5)             % passes
            testCase.assumeTrue(true)               % passes
            testCase.verifyGreaterThan(5, 9)        % fails
            testCase.assertEqual(3.14,pi)           % fails/incomplete
        end
        function testB(testCase)
            % This test contains an intentional error - passing a character
            % instead of a variable to the ones function.
            a = [1 2];
            testCase.verifyEqual(ones('a'),[1 1]);  % errors
        end
    end
end
```

At the command prompt, create a test suite from the `ExampleTest` class.

```
suite   = testsuite('ExampleTest');
```

Create a test runner with no plugins. This code creates a silent runner and provides you with complete control over the installed plugins. Add a `DiagnosticsRecordingPlugin` to the test runner.

```
import matlab.unittest.TestRunner;
import matlab.unittest.plugins.DiagnosticsRecordingPlugin;

runner = TestRunner.withNoPlugins;
runner.addPlugin(DiagnosticsRecordingPlugin);
```

Run the tests.

```
results = runner.run(suite);
```

Display the result from the second test. The test fails and is incomplete.

```
results(2)
```

```
ans = 

  TestResult with properties:

          Name: 'ExampleTest/testB'
        Passed: 0
        Failed: 1
    Incomplete: 1
      Duration: 7.8912e-04
       Details: [1×1 struct]

Totals:
   0 Passed, 1 Failed, 1 Incomplete.
   0.00078912 seconds testing time.
```

Index into the diagnostic record to display more information.

```
results(2).Details.DiagnosticRecord
```

```
ans = 

  ExceptionDiagnosticRecord with properties:

                        Event: 'ExceptionThrown'
                   EventScope: TestMethod
                 EventLocation: 'ExampleTest/testB'
                     Exception: [1×1 MException]
    AdditionalDiagnosticResults: [1×0 matlab.unittest.diagnostics.DiagnosticResult]
                        Stack: [1×1 struct]
                       Report: 'Error occurred in ExampleTest/testB and it did not run to completion…'
```

The test throws an uncaught exception.

Collect the diagnostic records for the first test, `testA`.

```
testA_records = results(1).Details.DiagnosticRecord
```

```
testA_records = 

  1×3 heterogeneous DiagnosticRecord (LoggedDiagnosticRecord, QualificationDiagnosticRe

    Event
    EventScope
    EventLocation
    Stack
    Report
```

View the events that the plugin recorded for `testA`.

```
{testA_records.Event}'
```

```
ans =

  3×1 cell array

    {'DiagnosticLogged'  }
    {'VerificationFailed'}
    {'AssertionFailed'   }
```

The plugin records the message logged at a `Terse` level of verbosity, and the verification and assertion failures.

Create a plugin that records messages at all verbosity levels and includes passing diagnostics. Rerun the tests and collect the diagnostic records for `testA`.

```
runner = TestRunner.withNoPlugins;
runner.addPlugin(DiagnosticsRecordingPlugin(...
    'IncludingPassingDiagnostics',true,'OutputDetail',4,'LoggingLevel',4));
results = runner.run(suite);
testA_records = results(1).Details.DiagnosticRecord;
```

View the events that the plugin recorded for `testA`.

```
{testA_records.Event}'
```

```
ans =

  6×1 cell array

    {'DiagnosticLogged'  }
    {'DiagnosticLogged'  }
    {'VerificationPassed'}
    {'AssumptionPassed'  }
    {'VerificationFailed'}
    {'AssertionFailed'   }
```

The plugin records diagnostic information for all the qualifications and calls to the `log` method.

Select all the records with failing event diagnostics.

```
failedRecords = selectFailed(testA_records)
```

```
failedRecords =

  1×2 QualificationDiagnosticRecord array with properties:
```

```
    Event
    EventScope
    EventLocation
    TestDiagnosticResults
    FrameworkDiagnosticResults
    AdditionalDiagnosticResults
    Stack
    Report
```

Select all the records with passing event diagnostics and display the report for the first record.

```
passedRecords = selectPassed(testA_records);
passedRecords(1).Report
```

```
ans =

    'Verification passed in ExampleTest/testA.

         ---------------------
         Framework Diagnostic:
         ---------------------
         verifyEqual passed.
         --> The values are equal using "isequaln".

         Actual Value:
              5
         Expected Value:
              5

         ------------------
         Stack Information:
         ------------------
         In C:\work\ExampleTest.m (ExampleTest.testA) at 6'
```

Select all the records for incomplete events.

```
incompleteRecords = selectIncomplete(testA_records)
```

```
incompleteRecords =

  QualificationDiagnosticRecord with properties:

                        Event: 'AssertionFailed'
                   EventScope: TestMethod
```

```
                 EventLocation: 'ExampleTest/testA'
          TestDiagnosticResults: [1×0 matlab.unittest.diagnostics.DiagnosticResult]
     FrameworkDiagnosticResults: [1×1 matlab.unittest.diagnostics.DiagnosticResult]
    AdditionalDiagnosticResults: [1×0 matlab.unittest.diagnostics.DiagnosticResult]
                          Stack: [1×1 struct]
                         Report: 'Assertion failed in ExampleTest/testA and it did not run to completion…'
```

Since this event is an assertion failure, the framework also returns this record with the failing diagnostics as `failedRecords(2)`.

Select all the records with logged events and display the logged messages.

```
loggedRecords = selectLogged(testA_records);
{loggedRecords.Report}'
```

```
ans =

  2×1 cell array

    {'[Terse] Diagnostic logged (2018-04-12 13:15:23): Terse log message'      }
    {'[Detailed] Diagnostic logged (2018-04-12 13:15:23): Detailed log message'}
```

See Also

`matlab.unittest.plugins.DiagnosticsRecordingPlugin` | `matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord` | `matlab.unittest.plugins.diagnosticrecord.ExceptionDiagnosticRecord` | `matlab.unittest.plugins.diagnosticrecord.QualificationDiagnosticRecord` | `selectIncomplete` | `selectLogged` | `selectPassed`

Topics

"Programmatically Access Test Diagnostics"

Introduced in R2016a

selectIncomplete

Class: `matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord`
Package: `matlab.unittest.plugins.diagnosticrecord`

Return diagnostic records for incomplete events

Syntax

```
selectedRecords = selectIncomplete(records)
```

Description

`selectedRecords = selectIncomplete(records)` returns the diagnostic records for incomplete events as an array of `matlab.unittest.plugins.diagnosticrecord.QualificationDiagnosticRecord` and `matlab.unittest.plugins.diagnosticrecord.ExceptionDiagnosticRecord` instances.

Incomplete events are events that result in a incomplete test on the `TestResult`. These events include assumption failures, assertion failures, and uncaught `MException` objects.

Input Arguments

`records` — Recorded diagnostics on test result
array of `matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord` instances

Recorded diagnostics on a test result, specified as an array of `matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord` instances. Access recorded diagnostics via the `DiagnosticRecord` field in the `Details` property on `TestResult`. For example, if your test results are stored in the variable `results`, find the recorded diagnostics for the second test by invoking `records = result(2).Details.DiagnosticRecord`.

Examples

Select Diagnostics on Test Result

In your working folder, create a file, `ExampleTest.m`, containing the following test class. The intent of this test is to illustrate how to use the `DiagnosticsRecordingPlugin` plugin, and it is not intended to be a representative unit test.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods (Test)
        function testA(testCase)
            testCase.log(1,'Terse log message')     % logs
            testCase.log(3,'Detailed log message')     % logs
            testCase.verifyEqual(3+2,5)             % passes
            testCase.assumeTrue(true)               % passes
            testCase.verifyGreaterThan(5, 9)        % fails
            testCase.assertEqual(3.14,pi)           % fails/incomplete
        end
        function testB(testCase)
            % This test contains an intentional error - passing a character
            % instead of a variable to the ones function.
            a = [1 2];
            testCase.verifyEqual(ones('a'),[1 1]);  % errors
        end
    end
end
```

At the command prompt, create a test suite from the `ExampleTest` class.

```
suite   = testsuite('ExampleTest');
```

Create a test runner with no plugins. This code creates a silent runner and provides you with complete control over the installed plugins. Add a `DiagnosticsRecordingPlugin` to the test runner.

```
import matlab.unittest.TestRunner;
import matlab.unittest.plugins.DiagnosticsRecordingPlugin;

runner = TestRunner.withNoPlugins;
runner.addPlugin(DiagnosticsRecordingPlugin);
```

Run the tests.

```
results = runner.run(suite);
```

Display the result from the second test. The test fails and is incomplete.

```
results(2)
```

```
ans = 

  TestResult with properties:

          Name: 'ExampleTest/testB'
        Passed: 0
        Failed: 1
    Incomplete: 1
      Duration: 7.8912e-04
       Details: [1×1 struct]

Totals:
   0 Passed, 1 Failed, 1 Incomplete.
   0.00078912 seconds testing time.
```

Index into the diagnostic record to display more information.

```
results(2).Details.DiagnosticRecord
```

```
ans = 

  ExceptionDiagnosticRecord with properties:

                        Event: 'ExceptionThrown'
                   EventScope: TestMethod
                 EventLocation: 'ExampleTest/testB'
                     Exception: [1×1 MException]
    AdditionalDiagnosticResults: [1×0 matlab.unittest.diagnostics.DiagnosticResult]
                        Stack: [1×1 struct]
                       Report: 'Error occurred in ExampleTest/testB and it did not run to completion…'
```

The test throws an uncaught exception.

Collect the diagnostic records for the first test, `testA`.

```
testA_records = results(1).Details.DiagnosticRecord
```

```
testA_records = 

  1×3 heterogeneous DiagnosticRecord (LoggedDiagnosticRecord, QualificationDiagnosticRe

    Event
    EventScope
    EventLocation
    Stack
    Report
```

View the events that the plugin recorded for `testA`.

```
{testA_records.Event}'
```

```
ans =

  3×1 cell array

    {'DiagnosticLogged'  }
    {'VerificationFailed'}
    {'AssertionFailed'   }
```

The plugin records the message logged at a `Terse` level of verbosity, and the verification and assertion failures.

Create a plugin that records messages at all verbosity levels and includes passing diagnostics. Rerun the tests and collect the diagnostic records for `testA`.

```
runner = TestRunner.withNoPlugins;
runner.addPlugin(DiagnosticsRecordingPlugin(...
    'IncludingPassingDiagnostics',true,'OutputDetail',4,'LoggingLevel',4));
results = runner.run(suite);
testA_records = results(1).Details.DiagnosticRecord;
```

View the events that the plugin recorded for `testA`.

```
{testA_records.Event}'
```

```
ans =

  6×1 cell array

    {'DiagnosticLogged'  }
    {'DiagnosticLogged'  }
    {'VerificationPassed'}
    {'AssumptionPassed'  }
    {'VerificationFailed'}
    {'AssertionFailed'   }
```

The plugin records diagnostic information for all the qualifications and calls to the `log` method.

Select all the records with failing event diagnostics.

```
failedRecords = selectFailed(testA_records)
```

```
failedRecords =

  1×2 QualificationDiagnosticRecord array with properties:
```

```
    Event
    EventScope
    EventLocation
    TestDiagnosticResults
    FrameworkDiagnosticResults
    AdditionalDiagnosticResults
    Stack
    Report
```

Select all the records with passing event diagnostics and display the report for the first record.

```
passedRecords = selectPassed(testA_records);
passedRecords(1).Report
```

```
ans =

    'Verification passed in ExampleTest/testA.

         ---------------------
         Framework Diagnostic:
         ---------------------
         verifyEqual passed.
         --> The values are equal using "isequaln".

         Actual Value:
              5
         Expected Value:
              5

         ------------------
         Stack Information:
         ------------------
         In C:\work\ExampleTest.m (ExampleTest.testA) at 6'
```

Select all the records for incomplete events.

```
incompleteRecords = selectIncomplete(testA_records)
```

```
incompleteRecords =

  QualificationDiagnosticRecord with properties:

                        Event: 'AssertionFailed'
                   EventScope: TestMethod
```

```
                 EventLocation: 'ExampleTest/testA'
          TestDiagnosticResults: [1×0 matlab.unittest.diagnostics.DiagnosticResult]
     FrameworkDiagnosticResults: [1×1 matlab.unittest.diagnostics.DiagnosticResult]
    AdditionalDiagnosticResults: [1×0 matlab.unittest.diagnostics.DiagnosticResult]
                          Stack: [1×1 struct]
                         Report: 'Assertion failed in ExampleTest/testA and it did not run to completion…'
```

Since this event is an assertion failure, the framework also returns this record with the failing diagnostics as `failedRecords(2)`.

Select all the records with logged events and display the logged messages.

```
loggedRecords = selectLogged(testA_records);
{loggedRecords.Report}'
```

```
ans =

  2×1 cell array

    {'[Terse] Diagnostic logged (2018-04-12 13:15:23): Terse log message'      }
    {'[Detailed] Diagnostic logged (2018-04-12 13:15:23): Detailed log message'}
```

See Also

`matlab.unittest.plugins.DiagnosticsRecordingPlugin` | `matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord` | `matlab.unittest.plugins.diagnosticrecord.ExceptionDiagnosticRecord` | `matlab.unittest.plugins.diagnosticrecord.QualificationDiagnosticRecord` | `selectFailed` | `selectLogged` | `selectPassed`

Topics

"Programmatically Access Test Diagnostics"

Introduced in R2016a

selectLogged

Class: `matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord`
Package: `matlab.unittest.plugins.diagnosticrecord`

Return diagnostic records for logged events

Syntax

```
selectedRecords = selectLogged(records)
```

Description

`selectedRecords = selectLogged(records)` returns the diagnostic records for logged events as an array of `matlab.unittest.plugins.diagnosticrecord.LoggedDiagnosticRecord` instances. Logged events are calls to the `log` method within the test.

Input Arguments

`records` — Recorded diagnostics on test result
array of `matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord` instances

Recorded diagnostics on a test result, specified as an array of `matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord` instances. Access recorded diagnostics via the `DiagnosticRecord` field in the `Details` property on `TestResult`. For example, if your test results are stored in the variable `results`, find the recorded diagnostics for the second test by invoking `records = result(2).Details.DiagnosticRecord`.

Examples

Select Diagnostics on Test Result

In your working folder, create a file, `ExampleTest.m`, containing the following test class. The intent of this test is to illustrate how to use the `DiagnosticsRecordingPlugin` plugin, and it is not intended to be a representative unit test.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods (Test)
        function testA(testCase)
            testCase.log(1,'Terse log message')     % logs
            testCase.log(3,'Detailed log message')     % logs
            testCase.verifyEqual(3+2,5)             % passes
            testCase.assumeTrue(true)               % passes
            testCase.verifyGreaterThan(5, 9)        % fails
            testCase.assertEqual(3.14,pi)           % fails/incomplete
        end
        function testB(testCase)
            % This test contains an intentional error - passing a character
            % instead of a variable to the ones function.
            a = [1 2];
            testCase.verifyEqual(ones('a'),[1 1]);  % errors
        end
    end
end
```

At the command prompt, create a test suite from the `ExampleTest` class.

```
suite   = testsuite('ExampleTest');
```

Create a test runner with no plugins. This code creates a silent runner and provides you with complete control over the installed plugins. Add a `DiagnosticsRecordingPlugin` to the test runner.

```
import matlab.unittest.TestRunner;
import matlab.unittest.plugins.DiagnosticsRecordingPlugin;

runner = TestRunner.withNoPlugins;
runner.addPlugin(DiagnosticsRecordingPlugin);
```

Run the tests.

```
results = runner.run(suite);
```

Display the result from the second test. The test fails and is incomplete.

```
results(2)
```

```
ans =

  TestResult with properties:
```

```
          Name: 'ExampleTest/testB'
        Passed: 0
        Failed: 1
    Incomplete: 1
      Duration: 7.8912e-04
       Details: [1×1 struct]

Totals:
   0 Passed, 1 Failed, 1 Incomplete.
   0.00078912 seconds testing time.
```

Index into the diagnostic record to display more information.

```
results(2).Details.DiagnosticRecord
```

```
ans =

  ExceptionDiagnosticRecord with properties:

                        Event: 'ExceptionThrown'
                   EventScope: TestMethod
                 EventLocation: 'ExampleTest/testB'
                     Exception: [1×1 MException]
    AdditionalDiagnosticResults: [1×0 matlab.unittest.diagnostics.DiagnosticResult]
                        Stack: [1×1 struct]
                       Report: 'Error occurred in ExampleTest/testB and it did not run to completion…'
```

The test throws an uncaught exception.

Collect the diagnostic records for the first test, `testA`.

```
testA_records = results(1).Details.DiagnosticRecord
```

```
testA_records =

  1×3 heterogeneous DiagnosticRecord (LoggedDiagnosticRecord, QualificationDiagnosticRe

    Event
    EventScope
    EventLocation
    Stack
    Report
```

View the events that the plugin recorded for `testA`.

```
{testA_records.Event}'
```

```
ans =

  3×1 cell array

    {'DiagnosticLogged'  }
    {'VerificationFailed'}
    {'AssertionFailed'   }
```

The plugin records the message logged at a `Terse` level of verbosity, and the verification and assertion failures.

Create a plugin that records messages at all verbosity levels and includes passing diagnostics. Rerun the tests and collect the diagnostic records for `testA`.

```
runner = TestRunner.withNoPlugins;
runner.addPlugin(DiagnosticsRecordingPlugin(...
    'IncludingPassingDiagnostics',true,'OutputDetail',4,'LoggingLevel',4));
results = runner.run(suite);
testA_records = results(1).Details.DiagnosticRecord;
```

View the events that the plugin recorded for `testA`.

```
{testA_records.Event}'
```

```
ans =

  6×1 cell array

    {'DiagnosticLogged'  }
    {'DiagnosticLogged'  }
    {'VerificationPassed'}
    {'AssumptionPassed'  }
    {'VerificationFailed'}
    {'AssertionFailed'   }
```

The plugin records diagnostic information for all the qualifications and calls to the `log` method.

Select all the records with failing event diagnostics.

```
failedRecords = selectFailed(testA_records)
```

```
failedRecords =

  1×2 QualificationDiagnosticRecord array with properties:

    Event
```

```
    EventScope
    EventLocation
    TestDiagnosticResults
    FrameworkDiagnosticResults
    AdditionalDiagnosticResults
    Stack
    Report
```

Select all the records with passing event diagnostics and display the report for the first record.

```
passedRecords = selectPassed(testA_records);
passedRecords(1).Report
```

```
ans =

    'Verification passed in ExampleTest/testA.

         ---------------------
         Framework Diagnostic:
         ---------------------
         verifyEqual passed.
         --> The values are equal using "isequaln".

         Actual Value:
              5
         Expected Value:
              5

         ------------------
         Stack Information:
         ------------------
         In C:\work\ExampleTest.m (ExampleTest.testA) at 6'
```

Select all the records for incomplete events.

```
incompleteRecords = selectIncomplete(testA_records)
```

```
incompleteRecords =

  QualificationDiagnosticRecord with properties:

                        Event: 'AssertionFailed'
                   EventScope: TestMethod
                EventLocation: 'ExampleTest/testA'
         TestDiagnosticResults: [1×0 matlab.unittest.diagnostics.DiagnosticResult]
     FrameworkDiagnosticResults: [1×1 matlab.unittest.diagnostics.DiagnosticResult]
```

```
    AdditionalDiagnosticResults: [1×0 matlab.unittest.diagnostics.DiagnosticResult]
                          Stack: [1×1 struct]
                         Report: 'Assertion failed in ExampleTest/testA and it did not run to completion…'
```

Since this event is an assertion failure, the framework also returns this record with the failing diagnostics as `failedRecords(2)`.

Select all the records with logged events and display the logged messages.

```
loggedRecords = selectLogged(testA_records);
{loggedRecords.Report}'
```

```
ans =

  2×1 cell array

    {'[Terse] Diagnostic logged (2018-04-12 13:15:23): Terse log message'      }
    {'[Detailed] Diagnostic logged (2018-04-12 13:15:23): Detailed log message'}
```

See Also

`matlab.unittest.plugins.DiagnosticsRecordingPlugin` | `matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord` | `matlab.unittest.plugins.diagnosticrecord.LoggedDiagnosticRecord` | `selectFailed` | `selectIncomplete` | `selectPassed`

Topics

"Programmatically Access Test Diagnostics"

Introduced in R2016a

selectPassed

Class: `matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord`
Package: `matlab.unittest.plugins.diagnosticrecord`

Return diagnostic records for passed events

Syntax

```
selectedRecords = selectPassed(records)
```

Description

`selectedRecords = selectPassed(records)` returns the diagnostic records for passed events as an array of `matlab.unittest.plugins.diagnosticrecord.QualificationDiagnosticRecord` instances.

Input Arguments

`records` — Recorded diagnostics on test result
array of `matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord` instances

Recorded diagnostics on a test result, specified as an array of `matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord` instances. Access recorded diagnostics via the `DiagnosticRecord` field in the `Details` property on `TestResult`. For example, if your test results are stored in the variable `results`, find the recorded diagnostics for the second test by invoking `records = result(2).Details.DiagnosticRecord`.

Examples

Select Diagnostics on Test Result

In your working folder, create a file, `ExampleTest.m`, containing the following test class. The intent of this test is to illustrate how to use the `DiagnosticsRecordingPlugin` plugin, and it is not intended to be a representative unit test.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods (Test)
        function testA(testCase)
            testCase.log(1,'Terse log message')     % logs
            testCase.log(3,'Detailed log message')     % logs
            testCase.verifyEqual(3+2,5)             % passes
            testCase.assumeTrue(true)               % passes
            testCase.verifyGreaterThan(5, 9)        % fails
            testCase.assertEqual(3.14,pi)           % fails/incomplete
        end
        function testB(testCase)
            % This test contains an intentional error - passing a character
            % instead of a variable to the ones function.
            a = [1 2];
            testCase.verifyEqual(ones('a'),[1 1]);  % errors
        end
    end
end
```

At the command prompt, create a test suite from the `ExampleTest` class.

```
suite   = testsuite('ExampleTest');
```

Create a test runner with no plugins. This code creates a silent runner and provides you with complete control over the installed plugins. Add a `DiagnosticsRecordingPlugin` to the test runner.

```
import matlab.unittest.TestRunner;
import matlab.unittest.plugins.DiagnosticsRecordingPlugin;

runner = TestRunner.withNoPlugins;
runner.addPlugin(DiagnosticsRecordingPlugin);
```

Run the tests.

```
results = runner.run(suite);
```

Display the result from the second test. The test fails and is incomplete.

```
results(2)
```

```
ans =

  TestResult with properties:
```

```
          Name: 'ExampleTest/testB'
        Passed: 0
        Failed: 1
    Incomplete: 1
      Duration: 7.8912e-04
       Details: [1×1 struct]

Totals:
   0 Passed, 1 Failed, 1 Incomplete.
   0.00078912 seconds testing time.
```

Index into the diagnostic record to display more information.

```
results(2).Details.DiagnosticRecord
```

```
ans =

  ExceptionDiagnosticRecord with properties:

                        Event: 'ExceptionThrown'
                   EventScope: TestMethod
                 EventLocation: 'ExampleTest/testB'
                     Exception: [1×1 MException]
    AdditionalDiagnosticResults: [1×0 matlab.unittest.diagnostics.DiagnosticResult]
                        Stack: [1×1 struct]
                       Report: 'Error occurred in ExampleTest/testB and it did not run to completion…'
```

The test throws an uncaught exception.

Collect the diagnostic records for the first test, `testA`.

```
testA_records = results(1).Details.DiagnosticRecord
```

```
testA_records =

  1×3 heterogeneous DiagnosticRecord (LoggedDiagnosticRecord, QualificationDiagnosticRe

    Event
    EventScope
    EventLocation
    Stack
    Report
```

View the events that the plugin recorded for `testA`.

```
{testA_records.Event}'
```

```
ans =

  3×1 cell array

    {'DiagnosticLogged'  }
    {'VerificationFailed'}
    {'AssertionFailed'   }
```

The plugin records the message logged at a `Terse` level of verbosity, and the verification and assertion failures.

Create a plugin that records messages at all verbosity levels and includes passing diagnostics. Rerun the tests and collect the diagnostic records for `testA`.

```
runner = TestRunner.withNoPlugins;
runner.addPlugin(DiagnosticsRecordingPlugin(...
    'IncludingPassingDiagnostics',true,'OutputDetail',4,'LoggingLevel',4));
results = runner.run(suite);
testA_records = results(1).Details.DiagnosticRecord;
```

View the events that the plugin recorded for `testA`.

```
{testA_records.Event}'
```

```
ans =

  6×1 cell array

    {'DiagnosticLogged'  }
    {'DiagnosticLogged'  }
    {'VerificationPassed'}
    {'AssumptionPassed'  }
    {'VerificationFailed'}
    {'AssertionFailed'   }
```

The plugin records diagnostic information for all the qualifications and calls to the `log` method.

Select all the records with failing event diagnostics.

```
failedRecords = selectFailed(testA_records)
```

```
failedRecords =

  1×2 QualificationDiagnosticRecord array with properties:

    Event
```

```
    EventScope
    EventLocation
    TestDiagnosticResults
    FrameworkDiagnosticResults
    AdditionalDiagnosticResults
    Stack
    Report
```

Select all the records with passing event diagnostics and display the report for the first record.

```
passedRecords = selectPassed(testA_records);
passedRecords(1).Report
```

```
ans =

    'Verification passed in ExampleTest/testA.

         ---------------------
         Framework Diagnostic:
         ---------------------
         verifyEqual passed.
         --> The values are equal using "isequaln".

         Actual Value:
              5
         Expected Value:
              5

         ------------------
         Stack Information:
         ------------------
         In C:\work\ExampleTest.m (ExampleTest.testA) at 6'
```

Select all the records for incomplete events.

```
incompleteRecords = selectIncomplete(testA_records)
```

```
incompleteRecords =

  QualificationDiagnosticRecord with properties:

                        Event: 'AssertionFailed'
                   EventScope: TestMethod
                EventLocation: 'ExampleTest/testA'
         TestDiagnosticResults: [1×0 matlab.unittest.diagnostics.DiagnosticResult]
     FrameworkDiagnosticResults: [1×1 matlab.unittest.diagnostics.DiagnosticResult]
```

```
    AdditionalDiagnosticResults: [1×0 matlab.unittest.diagnostics.DiagnosticResult]
                          Stack: [1×1 struct]
                         Report: 'Assertion failed in ExampleTest/testA and it did not run to completion…'
```

Since this event is an assertion failure, the framework also returns this record with the failing diagnostics as `failedRecords(2)`.

Select all the records with logged events and display the logged messages.

```
loggedRecords = selectLogged(testA_records);
{loggedRecords.Report}'
```

```
ans =

  2×1 cell array

    {'[Terse] Diagnostic logged (2018-04-12 13:15:23): Terse log message'      }
    {'[Detailed] Diagnostic logged (2018-04-12 13:15:23): Detailed log message'}
```

See Also

`matlab.unittest.plugins.DiagnosticsRecordingPlugin` | `matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord` | `matlab.unittest.plugins.diagnosticrecord.QualificationDiagnosticRecord` | `selectFailed` | `selectIncomplete` | `selectLogged`

Topics

"Programmatically Access Test Diagnostics"

Introduced in R2016a

matlab.unittest.plugins.DiagnosticsValidationPlugin class

Package: `matlab.unittest.plugins`

Plugin to help validate diagnostic code

Description

The `DiagnosticsValidationPlugin` creates a plugin to help validate diagnostic code.

Add the `DiagnosticsValidationPlugin` to the `TestRunner` to confirm that user-supplied diagnostics execute correctly. This plugin is useful because typically tests do not encounter failure conditions. A failure can result in unexercised diagnostic code. If a programming error exists in this diagnostic code, the error is not evident unless the test fails. However, at this point in the testing process, the diagnostics for the failure condition are lost due to the error in the diagnostic code.

Use this plugin to unconditionally evaluate the diagnostics supplied by the test writer, regardless of whether the test results in a passing or failing condition. This approach helps you to confirm that all of the diagnostic code is free from programming errors.

The diagnostic analysis can reduce the test performance and can result in very verbose text output. Be aware of these impacts before using this plugin for routine testing.

Construction

`matlab.unittest.plugins.DiagnosticsValidationPlugin` creates a plugin to help validate diagnostic code.

`matlab.unittest.plugins.DiagnosticsValidationPlugin(stream)` redirects all the text output to the output stream, `stream`. If you do not specify the output stream, the plugin uses the default `ToStandardOutput` stream.

Input Arguments

stream

Location where the plugin directs text output, specified as an `OutputStream`.

Default: `ToStandardOutput`

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Add Plugin to TestRunner

In your working folder, create a file, `ExampleTest.m`, containing the following test class. In this example, the `testThree` method has an intentional error. The method should use a function handle to the `dir` function as a `FunctionHandleDiagnostic`, but `dir` is misspelled.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase)
            % test code
        end
        function testTwo(testCase)
            % test code
        end
        function testThree(testCase)
            % The following should use @dir as a function handle,
            % but there is a typo
            testCase.verifyEqual('myfile','myfile', @dri)
        end
    end
end
```

All of the tests in `ExampleTest.m` result in a passing condition, but there is an error in the diagnostic.

At the command prompt, create a test suite from the `ExampleTest` class.

```
import matlab.unittest.TestRunner
import matlab.unittest.TestSuite
import matlab.unittest.plugins.DiagnosticsValidationPlugin

suite = TestSuite.fromClass(?ExampleTest);
```

Create a test runner configured with text output.

```
runner = TestRunner.withTextOutput;
```

Run the tests.

```
result1 = runner.run(suite);
```

```
Running ExampleTest
...
Done ExampleTest
__________
```

No diagnostic output is displayed because all the tests passed. The testing framework does not encounter the bug in the `FunctionHandleDiagnostic` of `testThree`.

Add `DiagnosticValidationPlugin` to the runner and run the tests.

```
runner.addPlugin(DiagnosticsValidationPlugin)
result2 = runner.run(suite);
```

```
Running ExampleTest
..
------------------------------
Validation of Test Diagnostic:
------------------------------
Error occurred while capturing diagnostics:
Error using evalc
Undefined function or variable 'dri'.

Error in ExampleTest/testThree (line 12)
            testCase.verifyEqual('myfile','myfile', @dri);

.
Done ExampleTest
__________
```

The framework executes the diagnostic provided by the `FunctionHandleDiagnostic`, even though none of the tests fails. Without this plugin, the test framework only encounters the bug if the test fails.

See Also

`OutputStream` | `ToStandardOutput` | `matlab.unittest.diagnostics` | `matlab.unittest.plugins`

matlab.unittest.plugins.FailureDiagnosticsPlugin class

Package: `matlab.unittest.plugins`

(Not recommended) Plugin to show diagnostics on failure

Note `matlab.unittest.plugins.FailureDiagnosticsPlugin` class is not recommended. Use `matlab.unittest.plugins.DiagnosticsOutputPlugin` class instead.

Description

The `FailureDiagnosticsPlugin` creates a plugin to show diagnostics upon encountering a test failure. Add it to the `TestRunner` to output test failure diagnostics to the Command Window. This plugin is used by default when you construct a test runner using `TestRunner.withTextOutput`.

Construction

`matlab.unittest.plugins.FailureDiagnosticsPlugin` creates a plugin to show diagnostics upon encountering a test failure.

`matlab.unittest.plugins.FailureDiagnosticsPlugin(stream)` redirects all the text output to the output stream, `stream`. If you do not specify the output stream, the plugin uses the `ToStandardOutput` stream.

Input Arguments

`stream`

Location where the plugin directs text output, specified as an `OutputStream`.

Default: `ToStandardOutput`

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Add Plugin to TestRunner

In your working folder, create a file, `ExampleTest.m`, containing the following test class.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testPathAdd(testCase)
            % test code
        end
        function testOne(testCase)  % Test fails
            testCase.verifyEqual(5, 4, 'Testing 5==4')
        end
         function testTwo(testCase)  % Test passes
             testCase.verifyEqual(5, 5, 'Testing 5==5')
        end
    end
end
```

The `verifyEqual` qualification in `testOne` causes a test failure. The qualifications in `testOne` and `testTwo` include an instance of a `matlab.unittest.diagnostics.StringDiagnostic`.

At the command prompt, create a test suite from the `ExampleTest` class.

```
import matlab.unittest.TestRunner
import matlab.unittest.TestSuite
import matlab.unittest.plugins.FailureDiagnosticsPlugin

suite = TestSuite.fromClass(?ExampleTest);
```

Create a test runner with no plugins. This code creates a silent runner and provides you with complete control over the installed plugins.

```
runner = TestRunner.withNoPlugins;
```

Run the tests.

```
result1 = runner.run(suite);
```

No output is displayed, but `result1` contains information about the failed test.

Add `FailureDiagnosticsPlugin` to the runner and run the tests.

```
runner.addPlugin(FailureDiagnosticsPlugin)
result2 = runner.run(suite);
```

```
==============================================================================
Verification failed in ExampleTest/testOne.

    ----------------
    Test Diagnostic:
    ----------------
    Testing 5==4

    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyEqual failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
                Actual    Expected    Error    RelativeError
                ______    ________    _____    _____________

                5         4           1        0.25

    Actual double:
             5
    Expected double:
             4

    ------------------
    Stack Information:
    ------------------
    In C:\work\ExampleTest.m (ExampleTest.testOne) at 7
==============================================================================
Failure Summary:

     Name                 Failed  Incomplete  Reason(s)
    ================================================================
     ExampleTest/testOne    X                 Failed by verification.
```

The framework displays the `DiagnosticResult` of the `StringDiagnostic` for failed tests only. It also displays additional framework diagnostics. The `TestResult` object, `result2`, is the same as `result1`.

See Also

`matlab.unittest.plugins.DiagnosticsOutputPlugin`

Introduced in R2013a

matlab.unittest.plugins.FailOnWarningsPlugin class

Package: `matlab.unittest.plugins`

Plugin to fail tests that issue warnings

Description

The `FailOnWarningsPlugin` creates a plugin that, when added to the `TestRunner`, fails any test that issues a warning. The plugin produces a qualification failure in the test scope that issues the warning. For example, if a shared test fixture issues a warning, the plugin produces a qualification failure on the fixture and fails all tests that share the fixture.

The `FailOnWarningsPlugin` plugin does not produce a failure if:

- A test accounts for the warning through a constraint such as `IssuesWarnings` or `IssuesNoWarnings`, regardless of whether the constraint is satisfied or not.
- A warning is disabled. For example, if you disable a warning using the `SupressedWarningsFixture`.

Construction

`matlab.unittest.plugins.FailOnWarningsPlugin` creates a plugin that fails any test that issues a warning.

`matlab.unittest.plugins.FailOnWarningsPlugin('Ignoring',warnIDs)` creates a plugin that does not fail for the specified warning identifiers, `warnIDs`.

Input Arguments

`warnIDs` — Identifiers for warnings to ignore
cell array of character vectors

Identifiers for warnings to ignore, specified as a cell array of character vectors. The plugin does not fail a test for the warnings with identifiers included in `warnIDs`.

Example: `FailOnWarningsPlugin('Ignoring',{'MATLAB:singularMatrix'})`

Properties

`Ignore` — Values to ignore
empty cell array (default) | cell array of character vectors

The `Ignore` property is empty by default. To specify the property as a cell array of character vectors, use the `'Ignoring'` syntax when you construct the plugin instance.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Fail Tests If They Issue Warnings

Create the `solve` function to solve a set of linear equations. There is an intentional error in the `solve` function — the `assert` call that checks whether the matrix is singular should use `rcond` instead of `det`.

```
function x = solve(A,b)

assert(abs(det(A)) > 1e-12,... % intentional bug for illustrative purposes
    'The matrix is singular or nearly singular');

x = A\b;
```

Create the following test class. In `testTwo`, the `A` matrix is singular, but since there is a bug in the `solve` function, the `assert` call does not catch it.

```
classdef TestSolve < matlab.unittest.TestCase

    methods(Test)
```

```
        function testOne(testCase)
            A = eye(3);
            b = [3; 4; 1];
            testCase.verifyEqual(solve(A,b),b);
        end

        function testTwo(testCase)
            A = [1e-100 0; 0 1e100];
            b = [5; 5];
            expX = [5e100 5e-100];
            testCase.verifyEqual(solve(A,b),expX);
        end
    end
end
```

At the command prompt, create a test suite, and test runner.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;
import matlab.unittest.plugins.FailOnWarningsPlugin;

suite = TestSuite.fromClass(?TestSolve);
runner = TestRunner.withTextOutput;
```

Add the `FailOnWarningsPlugin` plugin, and run the tests. `testTwo` fails because the `solve` function issues a warning. Without `FailOnWarningsPlugin`, the `solve` function issues the warning, but both tests pass.

```
runner.addPlugin(FailOnWarningsPlugin);
result = runner.run(suite);
```

```
Running TestSolve
.Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND
.
================================================================================
Verification failed in TestSolve/testTwo.

    ----------------------
    Framework Diagnostic:
    ----------------------
    TestSolve/testTwo issued warnings:

        ---------------------------
        MATLAB:nearlySingularMatrix
        ---------------------------
```

```
        Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =
            In C:\work\solve.m (solve) at 6
            In C:\work\TestSolve.m (TestSolve.testTwo) at 14

    ------------------
    Stack Information:
    ------------------
    In C:\Program Files\MATLAB\R2015b\toolbox\matlab\testframework\+matlab\+unittest\+p
==============================================================================

Done TestSolve
_________

Failure Summary:

     Name               Failed  Incomplete  Reason(s)
    ===============================================================
     TestSolve/testTwo    X                 Failed by verification.
```

Tips

- Set the warning stack trace display using the `warning` function. For example, `warning('off','backtrace')`.
- Disable warnings using the `SuppressedWarningsFixture` class.

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.constraints.IssuesNoWarnings` | `matlab.unittest.constraints.IssuesWarnings` | `matlab.unittest.fixtures.SuppressedWarningsFixture` | `matlab.unittest.plugins.TestRunnerPlugin` | `warning`

Introduced in R2015b

matlab.unittest.plugins.LoggingPlugin class

Package: `matlab.unittest.plugins`

Plugin to report diagnostic messages

Description

The `LoggingPlugin` creates a plugin to report diagnostic messages that are created by the `log` method of a `TestCase` or `Fixture`.

Construction

Instantiate a `LoggingPlugin` using one of its static methods.

Use the `withVerbosity` static method to configure a plugin to respond to messages of a particular verbosity. Also, the `withVerbosity` method accepts a number of name/value pairs to configure the format for reporting logged messages.

Properties

`Description` — Logged diagnostic message description

`'Diagnostic logged'` (default) | character vector | string scalar

Logged diagnostic message description, specified as a character vector or string scalar. The value of this property is printed alongside each logged diagnostic message. `Description` is read only, and its value is set during construction.

`HideLevel` — Indicator to display verbosity level

`false` (default) | `true`

Indicator to display the verbosity level alongside each logged diagnostic, specified as `false` (`logical(0)`) or `true` (`logical(1)`). By default, this property is `false` and the test framework displays the verbosity level. `HideLevel` is read only, and its value is set during construction.

`HideTimestamp` — Indicator to display timestamp
`false` (default) | `true`

Indicator to display the timestamp from when the test framework generates the logged message alongside each logged diagnostic, specified as `false` (`logical(0)`) or `true` (`logical(1)`). By default, this property is `false` and the test framework displays the timestamp. `HideTimestamp` is read only, and its value is set during construction.

`NumStackFrames` — Number of stack frames to display
`0` (default) | integer value | `Inf`

Number of stack frames to display after each logged diagnostic message, specified as an integer value. By default, this property is `0`, and the test framework does not display stack information. If `NumStackFrames` is `Inf`, the test framework displays all available stack frames. `NumStackFrames` is read only, and its value is set during construction.

`Verbosity` — Verbosity levels supported by plugin instance
array of `matlab.unittest.Verbosity` instances

Verbosity levels supported by plugin instance, specified as an array of `matlab.unittest.Verbosity` instances. The plugin reacts to diagnostics that are logged at a verbosity level listed in this array. `Verbosity` is read only, and its value is set during construction.

Methods

withVerbosity	Construct LoggingPlugin for messages of specified verbosity

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

See Also

`log` | `log` | `matlab.unittest.Verbosity`

Introduced in R2014b

matlab.unittest.plugins.LoggingPlugin.withVerbosity

Class: `matlab.unittest.plugins.LoggingPlugin`
Package: `matlab.unittest.plugins`

Construct LoggingPlugin for messages of specified verbosity

Syntax

```
matlab.unittest.plugins.LoggingPlugin.withVerbosity(v)
matlab.unittest.plugins.LoggingPlugin.withVerbosity(v,stream)
matlab.unittest.plugins.LoggingPlugin.withVerbosity(v,Name,Value)
```

Description

`matlab.unittest.plugins.LoggingPlugin.withVerbosity(v)` constructs a `LoggingPlugin` for messages of the specified verbosity.

`matlab.unittest.plugins.LoggingPlugin.withVerbosity(v,stream)` redirects the text output to the output stream.

`matlab.unittest.plugins.LoggingPlugin.withVerbosity(v,Name,Value)` includes additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

v — Verbosity levels supported by plugin instance
`0` | `1` | `2` | `3` | `4` | `matlab.unittest.Verbosity` enumeration | enumeration name as string or char vector

Verbosity levels supported by the plugin instance, specified as an integer value between 0 and 4, a `matlab.unittest.Verbosity` enumeration object, or a string scalar or character vector corresponding to one of the predefined enumeration member names.

The plugin reacts to diagnostics that are logged at this level and lower. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration.

Numeric Representation	Enumeration Member Name	Verbosity Description
`0`	`None`	No information
`1`	`Terse`	Minimal information
`2`	`Concise`	Moderate amount of information
`3`	`Detailed`	Some supplemental information
`4`	`Verbose`	Lots of supplemental information

`stream` — Location where plugin directs text output
`ToStandardOutput` instance (default) | `OutputStream` instance

Location where the plugin directs text output, specified as an `OutputStream` instance. By default, the plugin uses the `OutputStream` subclass `ToStandardOutput` as the stream.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`Description` — Logged diagnostic message description
`'Diagnostic logged'` (default) | character vector | string scalar

Logged diagnostic message description, specified as a character vector or string scalar. This value is printed alongside each logged diagnostic message. If the value empty, the test framework does not display a description.

`ExcludingLowerLevels` — Indicator to display messages logged at levels lower than the verbosity level
`false` (default) | `true`

Indicator to display messages logged at levels lower than the verbosity level, `v`, specified as `false` or `true` (`logical(0)` or `logical(1)`). By default, the value is `false` and the

plugin reacts to all messages logged at level `v` or lower. If the value is `true`, the plugin reacts only to messages logged at level `v`.

`HideLevel` — Indicator to display verbosity level
`false` (default) | `true`

Indicator to display the verbosity level alongside each logged diagnostic, specified as `false` or `true` (`logical(0)` or `logical(1)`). By default, the value is `false` and the test framework displays the verbosity level.

`HideTimestamp` — Indicator to display timestamp
`false` (default) | `true`

Indicator to display the timestamp from when the test framework generates the logged message alongside each logged diagnostic, specified as `false` or `true` (`logical(0)` or `logical(1)`). By default, the value is `false` and the test framework displays the timestamp.

`NumStackFrames` — Number of stack frames to display
`0` (default) | integer value | `Inf`

Number of stack frames to display after each logged diagnostic message, specified as an integer value. By default, the value is `0`, and the test framework does not display stack information. If `NumStackFrames` is `Inf`, the test framework displays all available stack frames.

Examples

Create Logging Plugin

Create a function-based test in a file, `sampleLogTest.m`, in your working folder.

```
function tests = sampleLogTest
tests = functiontests(localfunctions);

function svdTest(testCase)
import matlab.unittest.Verbosity

log(testCase,'Generating matrix.');
m = rand(1000);
```

```
log(testCase,1,'About to call SVD.');
[U,S,V] = svd(m);

log(testCase,Verbosity.Terse,'SVD finished.');

verifyEqual(testCase,U*S*V',m,'AbsTol',1e-6)
```

At the command prompt, run the test.

```
results = run(sampleLogTest);
```

```
Running sampleLogTest
   [Terse] Diagnostic logged (2014-04-14T14:20:59): About to call SVD.
   [Terse] Diagnostic logged (2014-04-14T14:20:59): SVD finished.
.
Done sampleLogTest
__________
```

The default runner reports the diagnostics at level 1 (`Terse`).

Create a test runner to report the diagnostics at levels 1 and 2, and rerun the test.

```
import matlab.unittest.TestRunner
import matlab.unittest.plugins.LoggingPlugin

runner = TestRunner.withNoPlugins;
p = LoggingPlugin.withVerbosity(2);
runner.addPlugin(p);

results = runner.run(sampleLogTest);
```

```
 [Concise] Diagnostic logged (2014-04-14T14:28:14): Generating matrix.
   [Terse] Diagnostic logged (2014-04-14T14:28:14): About to call SVD.
   [Terse] Diagnostic logged (2014-04-14T14:28:15): SVD finished.
```

Configure Logged Message Output

Create the following class In a file in your current working folder, `ExampleLogTest.m`.

```
classdef ExampleLogTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase)  % Test fails
            log(testCase,3,'Starting Test')
            log(testCase,'Testing 5==4')
```

```
            testCase.verifyEqual(5,4)
            log(testCase,4,'Test Complete')
        end
        function testTwo(testCase)  % Test passes
            log(testCase,matlab.unittest.Verbosity.Detailed,'Starting Test')
            log(testCase,'Testing 5==5')
            testCase.verifyEqual(5,5)
            log(testCase,matlab.unittest.Verbosity.Verbose,'Test Complete')
        end
    end
end
```

The log messages in `testTwo` uses `Verbosity` enumerations instead of the corresponding integers.

At the command prompt, create the test suite and a runner at verbosity level 4, and then run the test.

```
import matlab.unittest.TestSuite
import matlab.unittest.TestRunner
import matlab.unittest.plugins.LoggingPlugin
suite   = TestSuite.fromClass(?ExampleLogTest);

runner = TestRunner.withNoPlugins;
p = LoggingPlugin.withVerbosity(4);
runner.addPlugin(p);

results = runner.run(suite);
```

```
[Detailed] Diagnostic logged (2014-04-14T15:24:03): Starting Test
 [Concise] Diagnostic logged (2014-04-14T15:24:03): Testing 5==4
 [Verbose] Diagnostic logged (2014-04-14T15:24:03): Test Complete
[Detailed] Diagnostic logged (2014-04-14T15:24:03): Starting Test
 [Concise] Diagnostic logged (2014-04-14T15:24:03): Testing 5==5
 [Verbose] Diagnostic logged (2014-04-14T15:24:03): Test Complete
```

Create a new plugin to direct the output to a file, `myOutput.log`, and rerun the tests.

```
import matlab.unittest.plugins.ToFile
outFile = 'myOutput.log';

runner = TestRunner.withNoPlugins;
p = LoggingPlugin.withVerbosity(4,ToFile(outFile));
runner.addPlugin(p);

results = runner.run(suite);
```

Observe the contents in the file created by the plugin.

```
disp(fileread(outFile))
```

```
[Detailed] Diagnostic logged (2014-04-14T15:27:44): Starting Test
 [Concise] Diagnostic logged (2014-04-14T15:27:44): Testing 5==4
 [Verbose] Diagnostic logged (2014-04-14T15:27:44): Test Complete
[Detailed] Diagnostic logged (2014-04-14T15:27:44): Starting Test
 [Concise] Diagnostic logged (2014-04-14T15:27:44): Testing 5==5
 [Verbose] Diagnostic logged (2014-04-14T15:27:44): Test Complete
```

Create a new plugin that does not display level 4 messages. Do not display the verbosity level or timestamp. Rerun the tests.

```
runner = TestRunner.withNoPlugins;
p = LoggingPlugin.withVerbosity(matlab.unittest.Verbosity.Detailed,...
    'HideLevel',true,'HideTimestamp',true);
runner.addPlugin(p);

results = runner.run(suite);
```

```
Diagnostic logged: Starting Test
Diagnostic logged: Testing 5==4
Diagnostic logged: Starting Test
Diagnostic logged: Testing 5==5
```

See Also

`log` | `log` | `matlab.unittest.Verbosity` | `matlab.unittest.plugins.OutputStream` | `matlab.unittest.plugins.ToStandardOutput`

Introduced in R2014b

matlab.unittest.plugins.OutputStream class

Package: `matlab.unittest.plugins`

Interface that determines where to send text output

Description

The `OutputStream` interface is an abstract interface class that you can use as a base class to specify where plugins direct their text output. To create a custom output stream, implement a `print` method that correctly handles the formatted text information the testing framework passes to it. Many text-oriented plugins accept an `OutputStream` to redirect the text they produce in a configurable manner.

Methods

print	Print text to output stream

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Create Custom Output Stream

In a file in your working directory, create a new output stream class in the file `ToFigure.m`. This class allows plugin output to be redirected to a figure.

```
classdef ToFigure < matlab.unittest.plugins.OutputStream

    properties(SetAccess=private)
        Figure
```

```
    end
    properties(Access=private)
        ListBox
    end
```

This class uses two properties. `Figure` is the figure that receives and displays the output. `ListBox` is a handle to the list box that displays the text.

In the same file, add the following `methods` block.

```
    methods
        function print(stream,formatSpec,varargin)
            % Create the figure
            if isempty(stream.Figure) || ~ishghandle(stream.Figure)
                stream.createFigure
            end
            newStr = sprintf(formatSpec,varargin{:});
            oldStr = strjoin(stream.ListBox.String', '\n');

            % Create the full message
            fullStr = [oldStr,newStr];
            fullStrCell = strsplit(fullStr,'\n','CollapseDelimiters',false);

            % Set the string and selection
            stream.ListBox.String = fullStrCell';
            stream.ListBox.Value = numel(fullStrCell);
            drawnow
        end
    end
```

You must implement the `print` method for any subclass of `OutputStream`. In this example, the method creates a new figure (if necessary), formats the incoming text, and then adds it to the output stream.

In the same file, add the following `methods` block containing a helper function to create the figure.

```
    methods(Access=private)
        function createFigure(stream)
            stream.Figure = figure(...
                'Name',         'Unit Test Output', ...
                'WindowStyle',  'docked');

            stream.ListBox = uicontrol( ...
```

```
                'Parent',       stream.Figure, ...
                'Style',        'listbox', ...
                'String',       {}, ...
                'Units',        'normalized', ...
                'Position',     [.05 .05 .9 .9], ...
                'Max',          2, ...
                'FontName',     'Monospaced', ...
                'FontSize',      13);
        end
    end
end
```

In an new file in your working folder, create `ExampleTest.m` containing the following test class.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase)  % Test fails
            testCase.verifyEqual(5, 4, 'Testing 5==4' );
        end
        function testTwo(testCase)  % Test passes
            testCase.verifyEqual(5, 5, 'Testing 5==5' );
        end
        function testThree(testCase)
            % test code
        end
    end
end
```

The `verifyEqual` qualification in `testOne` causes a test failure. The qualifications in `testOne` and `testTwo` include an instance of a `matlab.unittest.diagnostics.StringDiagnostic`.

At the command prompt, create a test suite from the `ExampleTest` class.

```
import matlab.unittest.TestSuite
import matlab.unittest.TestRunner
import matlab.unittest.plugins.DiagnosticsValidationPlugin

suite = TestSuite.fromClass(?ExampleTest);
```

Create a test runner that displays output to the command window.

```
runner = TestRunner.withTextOutput;
```

Create a `DiagnosticsValidationPlugin` that explicitly specifies that its output should go to a figure via the `ToFigure` output stream.

```
plugin = DiagnosticsValidationPlugin(ToFigure);
```

Add the plugin to the `TestRunner` and run the suite.

```
runner.addPlugin(plugin)
result = runner.run(suite);
```

```
Running ExampleTest

================================================================================
Verification failed in ExampleTest/testOne.

    ----------------
    Test Diagnostic:
    ----------------
    Testing 5==4

    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyEqual failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
                Actual    Expected    Error    RelativeError
                ______    ________    _____    _____________

                5         4           1        0.25

    Actual double:
             5
    Expected double:
             4

    ------------------
    Stack Information:
    ------------------
    In C:\work\ExampleTest.m (ExampleTest.testOne) at 4
================================================================================
...
Done ExampleTest
__________

Failure Summary:

     Name                 Failed  Incomplete  Reason(s)
    ===================================================================
     ExampleTest/testOne    X                 Failed by verification.
```

Only the test failures produce output to the screen. By default, `TestRunner.withTextOutput` uses a `DiagnosticsOutputPlugin` to display output on the screen.

In addition to the default text output being displayed on the screen, the `DiagnosticsValidationPlugin` output is directed to a docked figure. The figure shows the following text.

```
------------------------------
Validation of Test Diagnostic:
------------------------------
Testing 5==4
------------------------------
Validation of Test Diagnostic:
------------------------------
Testing 5==5
```

The `DiagnosticsValidationPlugin` displays the diagnostic information regardless of whether the tests encounter failure conditions.

See Also

`fprintf` | `matlab.unittest.plugins` | `matlab.unittest.plugins.ToStandardOutput`

Introduced in R2014a

print

Class: `matlab.unittest.plugins.OutputStream`
Package: `matlab.unittest.plugins`

Print text to output stream

Syntax

```
print(stream,formatSpec,A1,...,An)
```

Description

`print(stream,formatSpec,A1,...,An)` formats the data in arrays `A1,...,An` according to `formatSpec`, and sends the result to the output stream, `stream`. Assign `formatSpec` and `A1,...,An` using the same interface that you use for `sprintf` and `fprintf`.

Input Arguments

`stream`

Output stream, specified as an instance of the `OutputStream` class

`formatSpec`

Format of text in output stream, specified as a character vector. For information on the construction of `formatSpec`, see the input argument entry on the `fprintf` or `sprint` reference pages.

`A`

Numeric or character arrays, specified as a scalar, vector, matrix, or multidimensional array.

Examples

Create Custom Output Stream

In a file in your working directory, create a new output stream class in the file `ToFigure.m`. This class allows plugin output to be redirected to a figure.

```
classdef ToFigure < matlab.unittest.plugins.OutputStream

    properties(SetAccess=private)
        Figure
    end
    properties(Access=private)
        ListBox
    end
```

This class uses two properties. `Figure` is the figure that receives and displays the output. `ListBox` is a handle to the list box that displays the text.

In the same file, add the following `methods` block.

```
    methods
        function print(stream,formatSpec,varargin)
            % Create the figure
            if isempty(stream.Figure) || ~ishghandle(stream.Figure)
                stream.createFigure
            end
            newStr = sprintf(formatSpec,varargin{:});
            oldStr = strjoin(stream.ListBox.String', '\n');

            % Create the full message
            fullStr = [oldStr,newStr];
            fullStrCell = strsplit(fullStr,'\n','CollapseDelimiters',false);

            % Set the string and selection
            stream.ListBox.String = fullStrCell';
            stream.ListBox.Value = numel(fullStrCell);
            drawnow
        end
    end
```

You must implement the `print` method for any subclass of `OutputStream`. In this example, the method creates a new figure (if necessary), formats the incoming text, and then adds it to the output stream.

In the same file, add the following `methods` block containing a helper function to create the figure.

```
    methods(Access=private)
        function createFigure(stream)
            stream.Figure = figure(...
                'Name',          'Unit Test Output', ...
                'WindowStyle',   'docked');

            stream.ListBox = uicontrol( ...
                'Parent',        stream.Figure, ...
                'Style',         'listbox', ...
                'String',        {}, ...
                'Units',         'normalized', ...
                'Position',      [.05 .05 .9 .9], ...
                'Max',           2, ...
                'FontName',      'Monospaced', ...
                'FontSize',       13);
        end
    end
end
```

In an new file in your working folder, create `ExampleTest.m` containing the following test class.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase)  % Test fails
            testCase.verifyEqual(5, 4, 'Testing 5==4' );
        end
        function testTwo(testCase)  % Test passes
            testCase.verifyEqual(5, 5, 'Testing 5==5' );
        end
        function testThree(testCase)
            % test code
        end
    end
end
```

The `verifyEqual` qualification in `testOne` causes a test failure. The qualifications in `testOne` and `testTwo` include an instance of a `matlab.unittest.diagnostics.StringDiagnostic`.

At the command prompt, create a test suite from the `ExampleTest` class.

```
import matlab.unittest.TestSuite
import matlab.unittest.TestRunner
import matlab.unittest.plugins.DiagnosticsValidationPlugin

suite = TestSuite.fromClass(?ExampleTest);
```

Create a test runner that displays output to the command window.

```
runner = TestRunner.withTextOutput;
```

Create a `DiagnosticsValidationPlugin` that explicitly specifies that its output should go to a figure via the `ToFigure` output stream.

```
plugin = DiagnosticsValidationPlugin(ToFigure);
```

Add the plugin to the `TestRunner` and run the suite.

```
runner.addPlugin(plugin)
result = runner.run(suite);
```

```
Running ExampleTest

================================================================================
Verification failed in ExampleTest/testOne.

    ----------------
    Test Diagnostic:
    ----------------
    Testing 5==4

    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyEqual failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
                Actual    Expected    Error    RelativeError
                ______    ________    _____    _____________

                5         4           1        0.25

    Actual double:
             5
    Expected double:
             4

    ------------------
```

```
    Stack Information:
    ------------------
    In C:\work\ExampleTest.m (ExampleTest.testOne) at 4
==============================================================================
...
Done ExampleTest
__________

Failure Summary:

     Name                 Failed  Incomplete  Reason(s)
    ===============================================================
     ExampleTest/testOne    X                 Failed by verification.
```

Only the test failures produce output to the screen. By default, `TestRunner.withTextOutput` uses a `DiagnosticsOutputPlugin` to display output on the screen.

In addition to the default text output being displayed on the screen, the `DiagnosticsValidationPlugin` output is directed to a docked figure. The figure shows the following text.

```
------------------------------
Validation of Test Diagnostic:
------------------------------
Testing 5==4
------------------------------
Validation of Test Diagnostic:
------------------------------
Testing 5==5
```

The `DiagnosticsValidationPlugin` displays the diagnostic information regardless of whether the tests encounter failure conditions.

See Also

`fprintf` | `sprintf`

Introduced in R2014a

matlab.unittest.plugins.Parallelizable class

Package: `matlab.unittest.plugins`

Interface for plugins that support running tests in parallel

Description

The `matlab.unittest.plugins.Parallelizable` class is an interface for `TestRunnerPlugin` instances that support dividing the test suite into separate groups and running each group on the current parallel pool. `Parallelizable` enables you to transfer data from MATLAB workers to the MATLAB client. Additionally, you can use `Parallelizable` to specify the circumstances in which a plugin is not allowed to support running tests in parallel.

To create a plugin that supports running tests in parallel, subclass both the `TestRunnerPlugin` and `Parallelizable` classes. A `TestRunner` enhanced by a parallelizable plugin can run tests in both serial and parallel modes. To run tests in parallel, which requires Parallel Computing Toolbox, invoke the `runInParallel` method on the `TestRunner`.

The `matlab.unittest.plugins.Parallelizable` class is a `handle` class.

Methods

Public Methods

supportsParallel	Specify when plugin supports running tests in parallel

Protected Methods

storeIn	Store data collected for group of tests
retrieveFrom	Retrieve data stored for group of tests

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.plugins.TestRunnerPlugin` | `matlab.unittest.plugins.plugindata.CommunicationBuffer` | `runInParallel`

Topics

"Run Tests in Parallel with Custom Plugin"

Introduced in R2019b

retrieveFrom

Class: `matlab.unittest.plugins.Parallelizable`
Package: `matlab.unittest.plugins`

Retrieve data stored for group of tests

Syntax

```
data = retrieveFrom(plugin,communicationBuffer)
```

Description

`data = retrieveFrom(plugin,communicationBuffer)` retrieves the `data` collected by `plugin` from the buffer `communicationBuffer`. The data must be stored in the buffer by a MATLAB worker while running a group of tests.

To run tests in parallel, the testing framework divides the original `TestSuite` array into separate groups and assigns them to workers on the current parallel pool (requires Parallel Computing Toolbox). To enable the MATLAB client to retrieve data from the workers, invoke `retrieveFrom` within the scope of the `reportFinalizedSuite` method of `TestRunnerPlugin`. The workers must store their data by invoking the `storeIn` method within the scope of `runTestSuite`.

Input Arguments

`plugin` — Plugin object
`Parallelizable` object

Plugin object, specified as an instance of the plugin class that subclasses the `matlab.unittest.plugins.Parallelizable` interface.

`communicationBuffer` — Buffer from which client retrieves data
`matlab.unittest.plugins.plugindata.CommunicationBuffer` object

Buffer from which the client retrieves data, specified as an instance of the `matlab.unittest.plugins.plugindata.CommunicationBuffer` class. `communicationBuffer` represents a property of the `matlab.unittest.plugins.plugindata.FinalizedSuitePluginData` instance that is passed to the `reportFinalizedSuite` method.

Output Arguments

`data` — Data retrieved from buffer
scalar | array

Data retrieved from the buffer, returned as a scalar or an array of objects.

Attributes

`Access`	`protected`
`Sealed`	`true`

To learn about attributes of methods, see Method Attributes.

Examples

Retrieve Number of TestSuite Elements

Invoke the `retrieveFrom` method in the plugin class to retrieve the number of elements in the finalized `TestSuite` array.

```
classdef ExamplePlugin < ...
        matlab.unittest.plugins.TestRunnerPlugin & ...
        matlab.unittest.plugins.Parallelizable

    methods (Access = protected)
        function runTestSuite(plugin, pluginData)

            % Display and store running TestSuite size
            suiteSize = numel(pluginData.TestSuite);
            fprintf('### Running %d tests\n', suiteSize)
            plugin.storeIn(pluginData.CommunicationBuffer, suiteSize);

            runTestSuite@matlab.unittest.plugins.TestRunnerPlugin(...
```

```
            plugin, pluginData);
        end

        function reportFinalizedSuite(plugin, pluginData)

            % Retrieve and display finalized TestSuite size
            suiteSize = plugin.retrieveFrom(pluginData.CommunicationBuffer);
            fprintf('### Finished running %d tests\n', suiteSize)

            reportFinalizedSuite@matlab.unittest.plugins.TestRunnerPlugin(...
                plugin, pluginData);
        end
    end

end
```

See Also

matlab.unittest.plugins.TestRunnerPlugin | matlab.unittest.plugins.plugindata.CommunicationBuffer | matlab.unittest.plugins.plugindata.FinalizedSuitePluginData | reportFinalizedSuite | runInParallel | runTestSuite | storeIn

Topics

"Run Tests in Parallel with Custom Plugin"

Introduced in R2019b

storeIn

Class: `matlab.unittest.plugins.Parallelizable`
Package: `matlab.unittest.plugins`

Store data collected for group of tests

Syntax

```
storeIn(plugin,communicationBuffer,data)
```

Description

`storeIn(plugin,communicationBuffer,data)` stores the `data` collected by `plugin` in the buffer `communicationBuffer`. The data is collected by a MATLAB worker while running a group of tests.

To run tests in parallel, the testing framework divides the original `TestSuite` array into separate groups and assigns them to workers on the current parallel pool (requires Parallel Computing Toolbox). To enable workers to store their data, invoke `storeIn` within the scope of the `runTestSuite` method of `TestRunnerPlugin`. The MATLAB client can retrieve the stored data by invoking the `retrieveFrom` method within the scope of `reportFinalizedSuite`.

Note Each new call to `storeIn` overwrites the previously stored data. Therefore, write to the buffer only one time within the scope of the `runTestSuite` method. If you need to store several data items, include them all in `data` using an appropriate data type.

Input Arguments

`plugin` — Plugin object
`Parallelizable` object

Plugin object, specified as an instance of the plugin class that subclasses the `matlab.unittest.plugins.Parallelizable` interface.

`communicationBuffer` — Buffer in which worker stores data
`matlab.unittest.plugins.plugindata.CommunicationBuffer` object

Buffer in which the worker stores data, specified as an instance of the `matlab.unittest.plugins.plugindata.CommunicationBuffer` class. `communicationBuffer` represents a property of the `matlab.unittest.plugins.plugindata.TestSuiteRunPluginData` instance that is passed to the `runTestSuite` method.

`data` — Data to store in buffer
scalar | array

Data to store in the buffer, specified as a scalar or an array of objects. For example, you can specify `data` as a numeric scalar, string array, cell array, structure, or class object.

Parallel Computing Toolbox serializes `data` on the worker side, saves it in `communicationBuffer`, and deserializes it on the client side. `data` must properly facilitate being saved and loaded. For more information, see “Save and Load Process for Objects”.

Attributes

`Access`	`protected`
`Sealed`	`true`

To learn about attributes of methods, see Method Attributes.

Examples

Store Number of TestSuite Elements

Invoke the `storeIn` method in the plugin class to store the number of elements in the running `TestSuite` array.

```
classdef ExamplePlugin < ...
        matlab.unittest.plugins.TestRunnerPlugin & ...
        matlab.unittest.plugins.Parallelizable

    methods (Access = protected)
```

```
        function runTestSuite(plugin, pluginData)

            % Display and store running TestSuite size
            suiteSize = numel(pluginData.TestSuite);
            fprintf('### Running %d tests\n', suiteSize)
            plugin.storeIn(pluginData.CommunicationBuffer, suiteSize);

            runTestSuite@matlab.unittest.plugins.TestRunnerPlugin(...
                plugin, pluginData);
        end

        function reportFinalizedSuite(plugin, pluginData)

            % Retrieve and display finalized TestSuite size
            suiteSize = plugin.retrieveFrom(pluginData.CommunicationBuffer);
            fprintf('### Finished running %d tests\n', suiteSize)

            reportFinalizedSuite@matlab.unittest.plugins.TestRunnerPlugin(...
                plugin, pluginData);
        end
    end

end
```

See Also

matlab.unittest.plugins.TestRunnerPlugin | matlab.unittest.plugins.plugindata.CommunicationBuffer | matlab.unittest.plugins.plugindata.TestSuiteRunPluginData | reportFinalizedSuite | retrieveFrom | runInParallel | runTestSuite

Topics

"Run Tests in Parallel with Custom Plugin"

Introduced in R2019b

supportsParallel

Class: `matlab.unittest.plugins.Parallelizable`
Package: `matlab.unittest.plugins`

Specify when plugin supports running tests in parallel

Syntax

```
tf = supportsParallel(plugin)
```

Description

`tf = supportsParallel(plugin)` returns logical `1` (`true`) if `plugin` supports running tests in parallel, and returns logical `0` (`false`) if `plugin` supports running tests only in serial mode.

In certain cases, the `TestRunner` cannot run tests in parallel when it is extended by a plugin. Override `supportsParallel` to specify the circumstances in which the plugin cannot be used to run tests in parallel.

Running tests in parallel requires Parallel Computing Toolbox.

Input Arguments

`plugin` — Plugin object
`Parallelizable` object

Plugin object, specified as an instance of the plugin class that subclasses the `matlab.unittest.plugins.Parallelizable` interface.

Examples

Specify When Plugin Supports Parallel Test Runs

`ExamplePlugin` is a parallelizable plugin that directs text output to the screen by default. Override `supportsParallel` so that tests run only in serial mode when the plugin writes text output to a file.

```
classdef ExamplePlugin < ...
        matlab.unittest.plugins.TestRunnerPlugin & ...
        matlab.unittest.plugins.Parallelizable

    properties
        Output (1,1) string = "StandardOutput"
    end

    methods
        function plugin = ExamplePlugin(stream)
            if nargin == 1
                plugin.Output = stream;
            end
        end
        function tf = supportsParallel(plugin)
            tf = (plugin.Output == "StandardOutput");
        end
    end
end
```

See Also

`matlab.unittest.plugins.OutputStream` | `matlab.unittest.plugins.TestRunnerPlugin`

Topics

"Run Tests in Parallel with Custom Plugin"

Introduced in R2019b

matlab.unittest.plugins.QualifyingPlugin class

Package: `matlab.unittest.plugins`
Superclasses: `matlab.unittest.plugins.TestRunnerPlugin`

Interface for plugins that perform system-wide qualifications

Description

Use qualifying plugins to produce test failures apart from your test content. Qualifications at the plugin level are useful because you can avoid repeating the same qualification in every test. You can decide to apply system-wide qualifications to the test suite periodically by simply adding the plugin to the test runner for a particular test session.

The `QualifyingPlugin` interface enables test runner plugin authors to implement plugins that perform system-wide qualifications on a test suite. You can perform verifications, assumptions, assertions, and fatal assertions in these inherited methods:

- `setupTestClass`
- `teardownTestClass`
- `setupTestMethod`
- `teardownTestMethod`

You can perform only assumptions, assertions, and fatal assertions in these inherited methods:

- `setupSharedTestFixture`
- `teardownSharedTestFixture`

Methods

assertUsing	Assert that value satisfies given constraint
assumeUsing	Assume that value satisfies given constraint
fatalAssertUsing	Fatally assert that value satisfies given constraint
verifyUsing	Verify that value satisfies given constraint

Inherited Methods

createSharedTestFixture	Extend creation of shared test fixture instances
createTestClassInstance	Extend creation of class-level `TestCase` instances
createTestMethodInstance	Extend creation of method-level `TestCase` instances
reportFinalizedResult	Enable reporting of finalized test results
reportFinalizedSuite	Extend reporting of finalized `TestSuite` array
runSession	Extend running of test session
runTest	Extend running of single `Test` element
runTestClass	Extend running of `Test` elements from same class or function
runTestSuite	Extend running of `TestSuite` array
runTestMethod	Extend running of single test method
setupSharedTestFixture	Extend setting up shared test fixture
setupTestClass	Extend setting up test class
setupTestMethod	Extend setting up test method
teardownSharedTestFixture	Extend tearing down shared test fixture
teardownTestClass	Extend tearing down test class
teardownTestMethod	Extend tearing down test method

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Test Using Qualification in Plugin

Create a plugin to ensure that test files leaves the MATLAB path unchanged. If the path after running the test file is different from the starting path, the test fails.

Create a class, `VerifyNoPathChangePlugin`, that inherits from the `matlab.unittest.plugins.QualifyingPlugin` class.

```
classdef VerifyNoPathChangePlugin < matlab.unittest.plugins.QualifyingPlugin
    properties (Access=private)
        OriginalPath
    end

    methods (Access=protected)
        function setupTestClass(plugin, pluginData)
            plugin.OriginalPath = path;
            setupTestClass@matlab.unittest.plugins.QualifyingPlugin(plugin,pluginData)
        end
        function teardownTestClass(plugin, pluginData)
            import matlab.unittest.constraints.IsEqualTo;
            teardownTestClass@matlab.unittest.plugins.QualifyingPlugin(plugin,pluginDat
            plugin.verifyUsing(pluginData.QualificationContext, ...
                path, IsEqualTo(plugin.OriginalPath), ...
                sprintf('%s modified the path.', pluginData.Name));
        end
    end
end
```

Create the following test class. The test modifies the path, but does not restore the original path.

```
classdef LeavesModifiedPath < matlab.unittest.TestCase
    methods (Test)
        function test1(~)
            addpath(pwd);
        end
    end
end
```

For purposes of this example, at the command prompt, remove the present working folder from the path.

```
rmpath(pwd)
```

Create a test suite, add the plugin to the test runner, and run the suite. The test fails because the path after the test is different from the starting path.

```
suite = matlab.unittest.TestSuite.fromClass(?LeavesModifiedPath);
runner = matlab.unittest.TestRunner.withTextOutput;
runner.addPlugin(VerifyNoPathChangePlugin);
runner.run(suite);
```

```
Running LeavesModifiedPath
.
================================================================================
Verification failed while setting up or tearing down LeavesModifiedPath.
As a result, all LeavesModifiedPath tests failed.

    ----------------
    Test Diagnostic:
    ----------------
    LeavesModifiedPath modified the path.

    ---------------------
    Framework Diagnostic:
    ---------------------
    IsEqualTo failed.
    --> StringComparator failed.
        --> The character arrays are not equal.

    Actual char:
        C:\work;C:\Program Files\MATLAB\R2015b\toolbox\matlab\...
   Expected char:
        C:\Program Files\MATLAB\R2015b\toolbox\matlab\...

    ------------------
    Stack Information:
    ------------------
    In C:\work\VerifyNoPathChangePlugin.m (VerifyNoPathChangePlugin.teardownTestClass)
================================================================================

Done LeavesModifiedPath
_________

Failure Summary:

     Name                      Failed  Incomplete  Reason(s)
```

```
    ==========================================================================
     LeavesModifiedPath/test1    X                 Failed by verification.
```

See Also

`matlab.unittest.plugins.FailOnWarningsPlugin` | `matlab.unittest.plugins.TestRunnerPlugin` | `matlab.unittest.plugins.plugindata.QualificationContext`

Topics

"Add Plugin to Test Runner"
"Write Plugins to Extend TestRunner"
"Create Custom Plugin"

Introduced in R2015b

assertUsing

Class: `matlab.unittest.plugins.QualifyingPlugin`
Package: `matlab.unittest.plugins`

Assert that value satisfies given constraint

Syntax

```
assertUsing(plugin,context,actual,constraint)
assertUsing(plugin,context,actual,constraint,diagnostic)
```

Description

`assertUsing(plugin,context,actual,constraint)` asserts that `actual` is a value that satisfies the given constraint, `constraint`, using the qualification `context`. If the actual value does not satisfy the constraint, the testing framework reports an assertion failure.

You can perform assertions in these `QualifyingPlugin` methods:

- `setupSharedTestFixture`
- `teardownSharedTestFixture`
- `setupTestClass`
- `teardownTestClass`
- `setupTestMethod`
- `teardownTestMethod`

`assertUsing(plugin,context,actual,constraint,diagnostic)` displays the diagnostic information, `diagnostic`, upon failure.

Input Arguments

`plugin` — Qualifying plugin instance
`matlab.unittest.plugins.QualifyingPlugin`

Plugin instance, specified as an instance of the `matlab.unittest.plugins.QualifyingPlugin` class.

`context` — Context for plugins
`matlab.unittest.plugins.plugindata.QualificationContext`

Context for plugins, specified as an instance of `matlab.unittest.plugins.plugindata.QualificationContext` class. This instance provides the context for the plugin to perform qualifications on test suites. The plugin obtains this context from the plugin data.

`actual` — Value to test
workspace variable

Value to test, specified as a workspace variable. `actual` can be any data type. The qualification passes when `actual` satisfies the specified constraint, `constraint`.

`constraint` — Condition test must satisfy
instance of class in `matlab.unittest.constraints` package

Condition the test must satisfy, specified as an instance of a class in the `matlab.unittest.constraints` package.

`diagnostic` — Diagnostic information to display
string | character vector | function handle | instance of class in `matlab.unittest.diagnostics` package

Diagnostic information to display upon failure, specified as a string, a character vector, a function handle, or an instance of a class in the `matlab.unittest.diagnostics` package.

Examples

Use Qualifications in Plugin

See examples for the `QualifyingPlugin` class and replace calls to `verifyUsing` with `assertUsing`.

Tips

Use the `QualifyingPlugin.assertUsing` method when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the associated test content as failed and incomplete. Alternatively,

- Use the `QualifyingPlugin.verifyUsing` method to produce and record failures without throwing an exception. Because a `QualifyingPlugin` performs additional qualifications beyond the ones defined in the unit tests, it typically uses verifications. Verifications do not cause an early exit from the test, ensuring that the test framework executes all test content. Use other qualification types to test for violation of preconditions or incorrect test setup.
- Use the `QualifyingPlugin.assumeUsing` method to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the associated test content as `Incomplete`.
- Use the `QualifyingPlugin.fatalAssertUsing` method to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

See Also

`matlab.unittest.plugins.QualifyingPlugin`

Introduced in R2015b

assumeUsing

Class: `matlab.unittest.plugins.QualifyingPlugin`
Package: `matlab.unittest.plugins`

Assume that value satisfies given constraint

Syntax

```
assumeUsing(plugin,context,actual,constraint)
assumeUsing(plugin,context,actual,constraint,diagnostic)
```

Description

`assumeUsing(plugin,context,actual,constraint)` assumes that `actual` is a value that satisfies the given constraint, `constraint`, using the qualification `context`. If the actual value does not satisfy the constraint, the testing framework reports an assumption failure.

You can perform assumptions in these `QualifyingPlugin` methods:

- `setupSharedTestFixture`
- `teardownSharedTestFixture`
- `setupTestClass`
- `teardownTestClass`
- `setupTestMethod`
- `teardownTestMethod`

`assumeUsing(plugin,context,actual,constraint,diagnostic)` displays the diagnostic information, `diagnostic`, upon failure.

Input Arguments

`plugin` — Qualifying plugin instance
`matlab.unittest.plugins.QualifyingPlugin`

Plugin instance, specified as an instance of the `matlab.unittest.plugins.QualifyingPlugin` class.

`context` — Context for plugins
`matlab.unittest.plugins.plugindata.QualificationContext`

Context for plugins, specified as an instance of `matlab.unittest.plugins.plugindata.QualificationContext` class. This instance provides the context for the plugin to perform qualifications on test suites. The plugin obtains this context from the plugin data.

`actual` — Value to test
workspace variable

Value to test, specified as a workspace variable. `actual` can be any data type. The qualification passes when `actual` satisfies the specified constraint, `constraint`.

`constraint` — Condition test must satisfy
instance of class in `matlab.unittest.constraints` package

Condition the test must satisfy, specified as an instance of a class in the `matlab.unittest.constraints` package.

`diagnostic` — Diagnostic information to display
string | character vector | function handle | instance of class in `matlab.unittest.diagnostics` package

Diagnostic information to display upon failure, specified as a string, a character vector, a function handle, or an instance of a class in the `matlab.unittest.diagnostics` package.

Examples

Use Qualifications in Plugin

See examples for the `QualifyingPlugin` class and replace calls to `verifyUsing` with `assumeUsing`.

Tips

Use the `QualifyingPlugin.assumeUsing` method to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the associated test content as `Incomplete`. Alternatively,

- Use the `QualifyingPlugin.verifyUsing` method to produce and record failures without throwing an exception. Because a `QualifyingPlugin` performs additional qualifications beyond the ones defined in the unit tests, it typically uses verifications. Verifications do not cause an early exit from the test, ensuring that the test framework executes all test content. Use other qualification types to test for violation of preconditions or incorrect test setup.
- Use the `QualifyingPlugin.assertUsing` method when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the associated test content as failed and incomplete.
- Use the `QualifyingPlugin.fatalAssertUsing` method to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

See Also

`matlab.unittest.plugins.QualifyingPlugin`

Introduced in R2015b

fatalAssertUsing

Class: `matlab.unittest.plugins.QualifyingPlugin`
Package: `matlab.unittest.plugins`

Fatally assert that value satisfies given constraint

Syntax

```
fatalAssertUsing(plugin,context,actual,constraint)
fatalAssertUsing(plugin,context,actual,constraint,diagnostic)
```

Description

`fatalAssertUsing(plugin,context,actual,constraint)` fatally asserts that `actual` is a value that satisfies the given constraint, `constraint`, using the qualification `context`. If the actual value does not satisfy the constraint, the testing framework reports a fatal assertion failure.

You can perform fatal assertions in these `QualifyingPlugin` methods:

- `setupSharedTestFixture`
- `teardownSharedTestFixture`
- `setupTestClass`
- `teardownTestClass`
- `setupTestMethod`
- `teardownTestMethod`

`fatalAssertUsing(plugin,context,actual,constraint,diagnostic)` displays the diagnostic information, `diagnostic`, upon failure.

Input Arguments

`plugin` — Qualifying plugin instance
`matlab.unittest.plugins.QualifyingPlugin`

Plugin instance, specified as an instance of the `matlab.unittest.plugins.QualifyingPlugin` class.

`context` — Context for plugins
`matlab.unittest.plugins.plugindata.QualificationContext`

Context for plugins, specified as an instance of `matlab.unittest.plugins.plugindata.QualificationContext` class. This instance provides the context for the plugin to perform qualifications on test suites. The plugin obtains this context from the plugin data.

`actual` — Value to test
workspace variable

Value to test, specified as a workspace variable. `actual` can be any data type. The qualification passes when `actual` satisfies the specified constraint, `constraint`.

`constraint` — Condition test must satisfy
instance of class in `matlab.unittest.constraints` package

Condition the test must satisfy, specified as an instance of a class in the `matlab.unittest.constraints` package.

`diagnostic` — Diagnostic information to display
string | character vector | function handle | instance of class in `matlab.unittest.diagnostics` package

Diagnostic information to display upon failure, specified as a string, a character vector, a function handle, or an instance of a class in the `matlab.unittest.diagnostics` package.

Examples

Use Qualifications in Plugin

See examples for the `QualifyingPlugin` class and replace calls to `verifyUsing` with `fatalAssertUsing`.

Tips

Use the `QualifyingPlugin.fatalAssertUsing` method to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

- Use the `QualifyingPlugin.verifyUsing` method to produce and record failures without throwing an exception. Because a `QualifyingPlugin` performs additional qualifications beyond the ones defined in the unit tests, it typically uses verifications. Verifications do not cause an early exit from the test, ensuring that the test framework executes all test content. Use other qualification types to test for violation of preconditions or incorrect test setup.
- Use the `QualifyingPlugin.assumeUsing` method to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the associated test content as `Incomplete`.
- Use the `QualifyingPlugin.assertUsing` method when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the associated test content as failed and incomplete.

See Also

`matlab.unittest.plugins.QualifyingPlugin`

Introduced in R2015b

verifyUsing

Class: `matlab.unittest.plugins.QualifyingPlugin`
Package: `matlab.unittest.plugins`

Verify that value satisfies given constraint

Syntax

```
verifyUsing(plugin,context,actual,constraint)
verifyUsing(plugin,context,actual,constraint,diagnostic)
```

Description

`verifyUsing(plugin,context,actual,constraint)` verifies that `actual` is a value that satisfies the given constraint, `constraint`, using the qualification `context`. If the actual value does not satisfy the constraint, the testing framework reports a verification failure.

You can perform verifications in these `QualifyingPlugin` methods:

- `setupTestClass`
- `teardownTestClass`
- `setupTestMethod`
- `teardownTestMethod`

`verifyUsing(plugin,context,actual,constraint,diagnostic)` displays the diagnostic information, `diagnostic`, upon failure.

Input Arguments

`plugin` — Qualifying plugin instance
`matlab.unittest.plugins.QualifyingPlugin`

Plugin instance, specified as an instance of the `matlab.unittest.plugins.QualifyingPlugin` class.

`context` — Context for plugins
`matlab.unittest.plugins.plugindata.QualificationContext`

Context for plugins, specified as an instance of `matlab.unittest.plugins.plugindata.QualificationContext` class. This instance provides the context for the plugin to perform qualifications on test suites. The plugin obtains this context from the plugin data.

`actual` — Value to test
workspace variable

Value to test, specified as a workspace variable. `actual` can be any data type. The qualification passes when `actual` satisfies the specified constraint, `constraint`.

`constraint` — Condition test must satisfy
instance of class in `matlab.unittest.constraints` package

Condition the test must satisfy, specified as an instance of a class in the `matlab.unittest.constraints` package.

`diagnostic` — Diagnostic information to display
string | character vector | function handle | instance of class in `matlab.unittest.diagnostics` package

Diagnostic information to display upon failure, specified as a string, a character vector, a function handle, or an instance of a class in the `matlab.unittest.diagnostics` package.

Examples

Use Qualifications in Plugin

See examples for the `QualifyingPlugin` class.

Tips

Use the `QualifyingPlugin.verifyUsing` method to produce and record failures without throwing an exception. Because a `QualifyingPlugin` performs additional qualifications beyond the ones defined in the unit tests, it typically uses verifications. Verifications do not cause an early exit from the test, ensuring that the test framework executes all test content. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

- Use the `QualifyingPlugin.assumeUsing` method to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`.
- Use the `QualifyingPlugin.assertUsing` method when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete.
- Use the `QualifyingPlugin.fatalAssertUsing` method to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session.

See Also

`matlab.unittest.plugins.QualifyingPlugin`

Introduced in R2015b

matlab.unittest.plugins.StopOnFailuresPlugin class

Package: `matlab.unittest.plugins`

Plugin to debug test failures

Description

The `StopOnFailuresPlugin` class provides a plugin to help debug test failures. Adding `StopOnFailuresPlugin` to the test runner pauses execution of a test if it encounters a qualification failure or uncaught error and puts MATLAB into debug mode.

If `StopOnFailuresPlugin` encounters a qualification failure or uncaught error in a test, you can use MATLAB debugging commands, such as `dpbup`, `dbstep`, `dbcont`, and `dbquit`, to investigate the cause of the test failure.

If `StopOnFailuresPlugin` encounters an uncaught error in a test, you cannot use `dbup` to shift context to the source of the error because the error disrupts the stack.

Construction

`matlab.unittest.plugins.StopOnFailuresPlugin` creates a plugin to debug test failures.

`matlab.unittest.plugins.StopOnFailuresPlugin('IncludingAssumptionFailures',tf)` indicates whether to react to assumption failures. By default, `StopOnFailuresPlugin` reacts to only uncaught errors and verification, assertion, and fatal assertion qualification errors. However, when `'IncludingAssumptionFailures'` is set to `true`, the plugin also reacts to assumption failures.

Input Arguments

`tf` — Indicator to react to assumption failures
`FALSE` (default) | `TRUE`

Indicator to react to assumption failures, specified as logical `false` or `true`. When this value is `true`, the test runner reacts to assumption failures. When the value is `false`, the plugin ignores assumption failures.

Properties

`IncludeAssumptionFailures`

When this property value is `true`, the instance reacts to assumption failures. When the value is `false`, the instance ignores assumption failures. The `IncludeAssumptionFailures` property is `false` by default. To specify the property as `true`, use the `IncludingAssumptionFailures` input when you construct the instance.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Add Plugin to TestRunner

In your working folder, create the file `ExampleTest.m` containing the following test class.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase)  % Test fails
            act = 3.1416;
            exp = pi;
            testCase.verifyEqual(act, exp)
        end
        function testTwo(testCase)  % Test does not complete
            testCase.assumeEqual(5, 4)
        end
    end
end
```

At the command prompt, create a test suite from the `ExampleTest` class and a test runner.

```
import matlab.unittest.TestRunner
import matlab.unittest.TestSuite
import matlab.unittest.plugins.StopOnFailuresPlugin

suite = TestSuite.fromClass(?ExampleTest);
runner = TestRunner.withTextOutput;
```

Run the tests.

```
result = runner.run(suite);
```

```
Running ExampleTest

================================================================================
Verification failed in ExampleTest/testOne.

    ----------------------
    Framework Diagnostic:
    ----------------------
    verifyEqual failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
                Actual        Expected               Error               RelativeError
                ______    _________________    ____________________    _________________

                3.1416    3.14159265358979    7.34641020683213e-06    2.33843499679617e

    Actual double:
           3.141600000000000
    Expected double:
           3.141592653589793

    ------------------
    Stack Information:
    ------------------
    In C:\work\ExampleTest.m (ExampleTest.testOne) at 6
================================================================================
.
================================================================================
ExampleTest/testTwo was filtered.
    Details
================================================================================
```

```
.
Done ExampleTest
__________

Failure Summary:

     Name                 Failed  Incomplete  Reason(s)
    ================================================================
     ExampleTest/testOne    X                 Failed by verification.
    ----------------------------------------------------------------
     ExampleTest/testTwo              X       Filtered by assumption.
```

As a result of the qualifications in the test class, the first test fails, and the second test does not complete.

Add the `StopOnFailuresPlugin` to the runner and run the tests.

```
runner.addPlugin(StopOnFailuresPlugin)
result = runner.run(suite);
```

```
Running ExampleTest
Test execution paused due to failure. Either click here or execute DBUP to shift contex
```

During the test execution, when the failure occurs, MATLAB enters debug mode.

Click on the hyperlinked word `'here'` to shift debug context to your work source. If necessary, make the command window your current window.

```
In workspace belonging to ExampleTest>ExampleTest.testOne at 6
```

Examine the variables in the workspace.

```
whos
```

```
  Name          Size            Bytes  Class           Attributes

  act           1x1                 8  double
  exp           1x1                 8  double
  testCase      1x1               112  ExampleTest
```

Now, you can investigate the cause of the test failure.

For example, see if the test passes when you specify a relative tolerance of `100*eps`.

```
testCase.verifyEqual(act,exp,'RelTol',100*eps)
```

```
Verification failed in ExampleTest/testOne.

    ----------------------
    Framework Diagnostic:
    ----------------------
    verifyEqual failed.
    --> The values are not equal using "isequaln".
    --> The error was not within relative tolerance.
    --> Failure table:
                Actual        Expected               Error               RelativeError
                ______    _________________    ____________________    _________________

                3.1416    3.14159265358979    7.34641020683213e-06    2.33843499679617e

    Actual double:
           3.141600000000000
    Expected double:
           3.141592653589793

    ------------------
    Stack Information:
    ------------------
    In C:\work\ExampleTest.m (ExampleTest.testOne) at 6
================================================================================
```

The test fails even with the specified tolerance.

Exit out of debug mode.

```
dbquit
```

```
================================================================================
Verification failed in ExampleTest/testOne.

    ----------------------
    Framework Diagnostic:
    ----------------------
    verifyEqual failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
                Actual        Expected               Error               RelativeError
                ______    _________________    ____________________    _________________

                3.1416    3.14159265358979    7.34641020683213e-06    2.33843499679617e
```

```
    Actual double:
           3.141600000000000
    Expected double:
           3.141592653589793

    ------------------
    Stack Information:
    ------------------
    In C:\work\ExampleTest.m (ExampleTest.testOne) at 6
================================================================================
.
================================================================================
ExampleTest/testTwo was filtered.
    Details
================================================================================
.
Done ExampleTest
__________

Failure Summary:

     Name                 Failed  Incomplete  Reason(s)
    ===================================================================
     ExampleTest/testOne    X                 Failed by verification.
    -------------------------------------------------------------------
     ExampleTest/testTwo              X       Filtered by assumption.
```

To enter debug mode for tests that fail by assumption, such as `testTwo` in the `ExampleTest` class, include `'IncludingAssumptionFailures'` option for the plugin.

```
runner = TestRunner.withTextOutput;
runner.addPlugin(StopOnFailuresPlugin(...
    'IncludingAssumptionFailures', true))
```

If you run the test runner, you enter debug mode for both `testOne` and `testTwo`.

See Also

`dbcont` | `dbquit` | `dbstep` | `dbup` | `matlab.unittest.plugins`

Introduced in R2013b

matlab.unittest.plugins.TAPPlugin class

Package: `matlab.unittest.plugins`

Plugin that produces Test Anything Protocol stream

Description

The `TAPPlugin` creates a plugin that produces a Test Anything Protocol (TAP) stream. Using this plugin, you can integrate MATLAB Unit Test results into third-party systems that recognize the TAP protocol. For example, you can integrate test results with continuous integration systems like Jenkins™ or TeamCity®.

Construction

Instantiate a `TAPPlugin` using one of its static methods.

To produce output in the original TAP format (version 12), use the `producingOriginalFormat` static method. To produce output with TAP version 13 format, use the `producingVersion13` static method. TAP version 13 output includes test diagnostics in YAML blocks.

Methods

producingOriginalFormat	Construct TAPPlugin for original TAP format
producingVersion13	Construct TAPPlugin for version 13 TAP format

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

See Also

`matlab.unittest.plugins.OutputStream` | `matlab.unittest.plugins.TestRunnerPlugin` | `matlab.unittest.plugins.ToFile`

External Websites

Jenkins
TeamCity
testanything.org

Introduced in R2014a

matlab.unittest.plugins.TAPPlugin.producingOriginalFormat

Class: `matlab.unittest.plugins.TAPPlugin`
Package: `matlab.unittest.plugins`

Construct TAPPlugin for original TAP format

Syntax

```
matlab.unittest.plugins.TAPPlugin.producingOriginalFormat
matlab.unittest.plugins.TAPPlugin.producingOriginalFormat(stream)
matlab.unittest.plugins.TAPPlugin.producingOriginalFormat( ___ ,
Name,Value)
```

Description

`matlab.unittest.plugins.TAPPlugin.producingOriginalFormat` creates a plugin that produces output in the form of the original Test Anything Protocol (TAP) format (version 12). By default, the plugin uses the `ToStandardOutput` stream, and the output appears on the screen. In this case, other output sent to the screen can invalidate the TAP stream.

`matlab.unittest.plugins.TAPPlugin.producingOriginalFormat(stream)` redirects all the text output to a specified output stream. For example, you can redirect the output to the `ToFile` stream.

`matlab.unittest.plugins.TAPPlugin.producingOriginalFormat( ___ , Name,Value)` creates a plugin with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`stream` — Location where the plugin directs text output
`matlab.unittest.plugins.ToStandardOutput` (default) | instance of `matlab.unittest.plugins.OutputStream` class

Location where the plugin directs text output, specified as an instance of the `OutputStream` class. By default, the plugin uses the `ToStandardOutput` stream.

Example: `stream = matlab.unittest.plugins.ToStandardOutput`

Example: `stream = matlab.unittest.plugins.ToFile('myFile.tap')`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `TAPPlugin.producingOriginalFormat('LoggingLevel', Verbosity.Detailed)` creates a plugin that includes diagnostics logged at and below the `Detailed` level.

`IncludingPassingDiagnostics` — Include passing event diagnostics
`false` (default) | `true`

Whether to include passing event diagnostics, specified as `false` or `true`. By default the plugin does not include diagnostics from passing events.

Data Types: `logical`

`LoggingLevel` — Maximum level of logged diagnostics
1 (default) | 0 | 2 | 3 | 4 | `matlab.unittest.Verbosity` enumeration | enumeration name as string or char vector

Maximum level at which logged diagnostics are included by the plugin instance, specified as an integer value from 0 through 4, a `matlab.unittest.Verbosity` enumeration object, or a string scalar or character vector corresponding to one of the predefined enumeration member names. The plugin includes diagnostics that are logged at this level and below. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration.

Numeric Representation	Enumeration Member Name	Verbosity Description
0	None	No information
1	Terse	Minimal information
2	Concise	Moderate amount of information
3	Detailed	Some supplemental information
4	Verbose	Lots of supplemental information

By default the plugin includes diagnostics logged at the `matlab.unittest.Verbosity.Terse` level (level 1). To exclude logged diagnostics, specify `LoggingLevel` as `Verbosity.None` (level 0).

Logged diagnostics are diagnostics that you supply to the testing framework with a call to the `log` or `log` method.

`OutputDetail` — Detail level for reported events
3 (default) | 0 | 1 | 2 | 4 | `matlab.unittest.Verbosity` enumeration | enumeration name as string or char vector

Detail level for reported events, specified as an integer value from 0 through 4, a `matlab.unittest.Verbosity` enumeration object, or a string scalar or character vector corresponding to one of the predefined enumeration member names. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration.

The plugin reports passing, failing, and logged events with the amount of detail specified by `OutputDetail`. By default the plugin records events at the `matlab.unittest.Verbosity.Detailed` level (level 3).

Numeric Representation	Enumeration Member Name	Verbosity Description
0	None	No information
1	Terse	Minimal information
2	Concise	Moderate amount of information
3	Detailed	Some supplemental information

Numeric Representation	Enumeration Member Name	Verbosity Description
4	`Verbose`	Lots of supplemental information

Examples

Create TAP Plugin

In a new file in your working folder, create `ExampleTest.m` containing the following test class.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase)  % Test fails
            testCase.verifyEqual(5,4,'Testing 5==4')
        end
        function testTwo(testCase)  % Test passes
            testCase.verifyEqual(5,5,'Testing 5==5')
        end
        function testThree(testCase)
            % test code
        end
    end
end
```

At the command prompt, create a test suite from the `ExampleTest` class.

```
import matlab.unittest.TestRunner
import matlab.unittest.TestSuite
import matlab.unittest.plugins.TAPPlugin
import matlab.unittest.plugins.ToFile

suite   = TestSuite.fromClass(?ExampleTest);
```

Create a test runner that displays output to the command window using the default plugin.

```
runner = TestRunner.withTextOutput;
```

Create a `TAPPlugin` that sends output to the file `MyTapOutput.tap`.

```
tapFile = 'MyTAPOutput.tap';
plugin = TAPPlugin.producingOriginalFormat(ToFile(tapFile));
```

Add the plugin to the `TestRunner` and run the suite.

```
runner.addPlugin(plugin)
result = runner.run(suite);
```

```
Running ExampleTest

================================================================================
Verification failed in ExampleTest/testOne.

    ----------------
    Test Diagnostic:
    ----------------
    Testing 5==4

    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyEqual failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
                Actual    Expected    Error    RelativeError
                ______    ________    _____    _____________

                5         4           1        0.25

    Actual double:
             5
    Expected double:
             4

    ------------------
    Stack Information:
    ------------------
    In C:\work\ExampleTest.m (ExampleTest.testOne) at 4
================================================================================
...
Done ExampleTest
_________

Failure Summary:
```

```
     Name                 Failed  Incomplete  Reason(s)
    ===================================================================
     ExampleTest/testOne    X                 Failed by verification.
```

Display the file created by the plugin.

```
disp(fileread(tapFile))
```

```
1..3
not ok 1 - ExampleTest/testOne
# ==============================================================================
# Verification failed in ExampleTest/testOne.
#
#     ----------------
#     Test Diagnostic:
#     ----------------
#     Testing 5==4
#
#     ---------------------
#     Framework Diagnostic:
#     ---------------------
#     verifyEqual failed.
#     --> The values are not equal using "isequaln".
#     --> Failure table:
#                 Actual    Expected    Error    RelativeError
#                 ______    ________    _____    _____________
#
#                 5         4           1        0.25
#
#     Actual double:
#              5
#     Expected double:
#              4
#
#     ------------------
#     Stack Information:
#     ------------------
#     In C:\work\ExampleTest.m (ExampleTest.testOne) at 4
# ==============================================================================
#
ok 2 - ExampleTest/testTwo
ok 3 - ExampleTest/testThree
```

You can use the `TAPPlugin` directed to standard output. However, any other text displayed to standard output (such as failed test information) interrupts the stream and has the potential to invalidate it.

See Also

`matlab.unittest.plugins.OutputStream` | `matlab.unittest.plugins.TAPPlugin.producingVersion13` | `matlab.unittest.plugins.ToFile`

External Websites

testanything.org

Introduced in R2014a

matlab.unittest.plugins.TAPPlugin.producingVersion13

Class: `matlab.unittest.plugins.TAPPlugin`
Package: `matlab.unittest.plugins`

Construct TAPPlugin for version 13 TAP format

Syntax

```
matlab.unittest.plugins.TAPPlugin.producingVersion13
matlab.unittest.plugins.TAPPlugin.producingVersion13(stream)
matlab.unittest.plugins.TAPPlugin.producingVersion13( ___ ,Name,Value)
```

Description

`matlab.unittest.plugins.TAPPlugin.producingVersion13` creates a plugin that produces output in version 13 of the Test Anything Protocol (TAP) format. The TAP version 13 output includes test diagnostics in YAML blocks. By default, the plugin uses the `ToStandardOutput` stream, and the output appears on the screen. In this case, other output sent to the screen can invalidate the TAP stream.

`matlab.unittest.plugins.TAPPlugin.producingVersion13(stream)` redirects all the text output to a specified output stream. For example, you can redirect the output to the `ToFile` stream.

`matlab.unittest.plugins.TAPPlugin.producingVersion13( ___ ,Name,Value)` creates a plugin with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`stream` — Location where the plugin directs text output
`matlab.unittest.plugins.ToStandardOutput` (default) | instance of `matlab.unittest.plugins.OutputStream` class

Location where the plugin directs text output, specified as an instance of the `OutputStream` class. By default, the plugin uses the `ToStandardOutput` stream.

Example: `stream = matlab.unittest.plugins.ToStandardOutput`

Example: `stream = matlab.unittest.plugins.ToFile('myFile.tap')`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `TAPPlugin.producingVersion13('LoggingLevel', Verbosity.Detailed)` creates a plugin that includes diagnostics logged at and below the `Detailed` level.

`IncludingPassingDiagnostics` — Include passing event diagnostics
`false` (default) | `true`

Whether to include passing event diagnostics, specified as `false` or `true`. By default the plugin does not include diagnostics from passing events.

Data Types: `logical`

`LoggingLevel` — Maximum level of logged diagnostics
1 (default) | 0 | 2 | 3 | 4 | `matlab.unittest.Verbosity` enumeration | enumeration name as string or char vector

Maximum level at which logged diagnostics are included by the plugin instance, specified as an integer value from 0 through 4, a `matlab.unittest.Verbosity` enumeration object, or a string scalar or character vector corresponding to one of the predefined enumeration member names. The plugin includes diagnostics that are logged at this level and below. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration.

Numeric Representation	Enumeration Member Name	Verbosity Description
`0`	`None`	No information
`1`	`Terse`	Minimal information

Numeric Representation	Enumeration Member Name	Verbosity Description
2	`Concise`	Moderate amount of information
3	`Detailed`	Some supplemental information
4	`Verbose`	Lots of supplemental information

By default the plugin includes diagnostics logged at the `matlab.unittest.Verbosity.Terse` level (level 1). To exclude logged diagnostics, specify `LoggingLevel` as `Verbosity.None` (level 0).

Logged diagnostics are diagnostics that you supply to the testing framework with a call to the `log` or `log` method.

`OutputDetail` — Detail level for reported events
3 (default) | 0 | 1 | 2 | 4 | `matlab.unittest.Verbosity` enumeration | enumeration name as string or char vector

Detail level for reported events, specified as an integer value from 0 through 4, a `matlab.unittest.Verbosity` enumeration object, or a string scalar or character vector corresponding to one of the predefined enumeration member names. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration.

The plugin reports passing, failing, and logged events with the amount of detail specified by `OutputDetail`. By default the plugin records events at the `matlab.unittest.Verbosity.Detailed` level (level 3).

Numeric Representation	Enumeration Member Name	Verbosity Description
0	`None`	No information
1	`Terse`	Minimal information
2	`Concise`	Moderate amount of information
3	`Detailed`	Some supplemental information
4	`Verbose`	Lots of supplemental information

Examples

Create TAP Plugin

In a new file in your working folder, create `ExampleTest.m` containing the following test class.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase)  % Test fails
            testCase.verifyEqual(5,4,'Testing 5==4')
        end
        function testTwo(testCase)  % Test passes
            testCase.verifyEqual(5,5,'Testing 5==5')
        end
        function testThree(testCase)
            % test code
        end
    end
end
```

At the command prompt, create a test suite from the `ExampleTest` class.

```
import matlab.unittest.TestRunner
import matlab.unittest.TestSuite
import matlab.unittest.plugins.TAPPlugin
import matlab.unittest.plugins.ToFile

suite = TestSuite.fromClass(?ExampleTest);
```

Create a test runner that displays output to the command window using the default plugin.

```
runner = TestRunner.withTextOutput;
```

Create a `TAPPlugin` that sends output to the file `MyTapOutput.tap`.

```
tapFile = 'MyTAPOutput.tap';
plugin = TAPPlugin.producingVersion13(ToFile(tapFile));
```

Add the plugin to the `TestRunner` and run the suite.

```
runner.addPlugin(plugin)
result = runner.run(suite);
```

```
Running ExampleTest

================================================================================
Verification failed in ExampleTest/testOne.

    ----------------
    Test Diagnostic:
    ----------------
    Testing 5==4

    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyEqual failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
                Actual    Expected    Error    RelativeError
                ______    ________    _____    _____________

                5         4           1        0.25

    Actual double:
             5
    Expected double:
             4

    ------------------
    Stack Information:
    ------------------
    In C:\work\ExampleTest.m (ExampleTest.testOne) at 4
================================================================================
...
Done ExampleTest
__________

Failure Summary:

     Name                 Failed  Incomplete  Reason(s)
    ===================================================================
     ExampleTest/testOne    X                 Failed by verification.
```

Display the file created by the plugin.

```
disp(fileread(tapFile))
```

```
TAP version 13
1..3
not ok 1 - ExampleTest/testOne
    ---
    Event:
        Event Name: 'VerificationFailed'
        Event Location: 'ExampleTest/testOne'
        Test Diagnostic: |
            Testing 5==4
        Framework Diagnostic: |
            verifyEqual failed.
            --> The values are not equal using "isequaln".
            --> Failure table:
                    Actual    Expected    Error    RelativeError
                    ______    ________    _____    _____________

                    5         4           1        0.25

            Actual Value:
                 5
            Expected Value:
                 4
        Stack: |
            In C:\work\ExampleTest.m (ExampleTest.testOne) at 4
    ...
ok 2 - ExampleTest/testTwo
ok 3 - ExampleTest/testThree
```

You can use the `TAPPlugin` directed to standard output. However, any other text displayed to standard output (such as failed test information) interrupts the stream and has the potential to invalidate it.

See Also

`matlab.unittest.plugins.OutputStream` | `matlab.unittest.plugins.TAPPlugin.producingVersion13` | `matlab.unittest.plugins.ToFile`

External Websites

testanything.org
yaml.org

Introduced in R2016b

matlab.unittest.plugins.ToFile class

Package: `matlab.unittest.plugins`
Superclasses: `matlab.unittest.plugins.OutputStream`

Output stream to write text output to file

Description

The `ToFile` class creates an output stream that writes text output to a UTF-8 encoded file. Whenever text prints to this stream, the output stream opens the file, appends the text, and closes the file.

Construction

`matlab.unittest.plugins.ToFile(fname)` creates an `OutputStream` that writes text output to the file, `fname`.

Input Arguments

`fname`

Name of file to write the output text, specified as a character vector or string scalar. If `fname` exists, the text from the stream is appended to the file.

Properties

`Filename`

Name of file to redirect text output from the plugin, specified in the input argument, `fname`.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Direct TAPPlugin Output Text to Separate File

In your working folder, create the file `ExampleTest.m` containing the following test class.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase)  % Test fails
            testCase.verifyEqual(5, 4, 'Testing 5==4')
        end
        function testTwo(testCase)  % Test passes
            testCase.verifyEqual(5, 5, 'Testing 5==5')
        end
        function testThree(testCase)
            % test code
        end
    end
end
```

The `verifyEqual` qualification in `testOne` causes a test failure. The qualifications in `testOne` and `testTwo` include an instance of a `matlab.unittest.diagnostics.StringDiagnostic`.

At the command prompt, create a test suite from the `ExampleTest` class.

```
import matlab.unittest.TestRunner
import matlab.unittest.TestSuite
import matlab.unittest.plugins.TAPPlugin
import matlab.unittest.plugins.ToFile

suite   = TestSuite.fromClass(?ExampleTest);
```

Create a test runner that displays output to the command window.

```
runner = TestRunner.withTextOutput;
```

Create a `TAPPlugin` that explicitly specifies that its output should go to the file, `MyTapOutput.tap`.

```
filename = 'MyTapOutput.tap';
plugin = TAPPlugin.producingOriginalFormat(ToFile(filename));
```

Add the plugin to the `TestRunner` and run the suite.

```
runner.addPlugin(plugin)
result = runner.run(suite);
```

```
Running ExampleTest

================================================================================
Verification failed in ExampleTest/testOne.

    ----------------
    Test Diagnostic:
    ----------------
    Testing 5==4

    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyEqual failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
                Actual    Expected    Error    RelativeError
                ______    ________    _____    _____________

                5         4           1        0.25

    Actual double:
             5
    Expected double:
             4

    ------------------
    Stack Information:
    ------------------
    In C:\work\ExampleTest.m (ExampleTest.testOne) at 4
================================================================================
...
Done ExampleTest
__________

Failure Summary:

     Name                 Failed  Incomplete  Reason(s)
    ================================================================
     ExampleTest/testOne    X                 Failed by verification.
```

Only the test failures produce output to the screen. By default, `TestRunner.withTextOutput` uses a `DiagnosticsOutputPlugin` to display output on the screen.

Observe contents in the file created by the plugin.

```
disp(fileread(filename))
```

```
1..3
not ok 1 - ExampleTest/testOne
# ================================================================================
# Verification failed in ExampleTest/testOne.
#
#     ----------------
#     Test Diagnostic:
#     ----------------
#     Testing 5==4
#
#     ---------------------
#     Framework Diagnostic:
#     ---------------------
#     verifyEqual failed.
#     --> The values are not equal using "isequaln".
#     --> Failure table:
#                 Actual    Expected    Error    RelativeError
#                 ______    ________    _____    _____________
#
#                 5         4           1        0.25
#
#     Actual double:
#              5
#     Expected double:
#              4
#
#     ------------------
#     Stack Information:
#     ------------------
#     In C:\work\ExampleTest.m (ExampleTest.testOne) at 4
# ================================================================================
#
ok 2 - ExampleTest/testTwo
ok 3 - ExampleTest/testThree
```

See Also

fopen | fprintf | matlab.unittest.plugins | matlab.unittest.plugins.OutputStream | matlab.unittest.plugins.ToUniqueFile

Introduced in R2014a

matlab.unittest.plugins.ToStandardOutput class

Package: `matlab.unittest.plugins`
Superclasses: `matlab.unittest.plugins.OutputStream`

Output stream to display text information to screen

Description

The `ToStandardOutput` class creates an output stream to display text output to the screen. Many plugins that accept an output stream use `ToStandardOutput` as their default stream.

Construction

`matlab.unittest.plugins.ToStandardOutput` creates an `OutputStream` that prints text output to the screen.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Direct Plugin Output Text to Standard Output

In your working folder, create the file `ExampleTest.m` containing the following test class.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
```

```
        function testOne(testCase)  % Test fails
            testCase.verifyEqual(5, 4, 'Testing 5==4')
        end
        function testTwo(testCase)  % Test passes
            testCase.verifyEqual(5, 5, 'Testing 5==5')
        end
        function testThree(testCase)
            % test code
        end
    end
end
```

The `verifyEqual` qualification in `testOne` causes a test failure. The qualifications in `testOne` and `testTwo` include an instance of a `matlab.unittest.diagnostics.StringDiagnostic`.

At the command prompt, create a test suite from the `ExampleTest` class.

```
import matlab.unittest.TestRunner
import matlab.unittest.TestSuite
import matlab.unittest.plugins.DiagnosticsOutputPlugin
import matlab.unittest.plugins.ToStandardOutput

suite   = TestSuite.fromClass(?ExampleTest);
```

Create a test runner with no plugins. This code creates a silent runner and provides you with complete control over the installed plugins.

```
runner = TestRunner.withNoPlugins;
```

Create a `DiagnosticsOutputPlugin` that explicitly specifies that its output should go to the screen.

```
plugin = DiagnosticsOutputPlugin(ToStandardOutput);
```

Add the plugin to the `TestRunner` and run the suite.

```
runner.addPlugin(plugin)
result = runner.run(suite);
```

```
==============================================================================
Verification failed in ExampleTest/testOne.

    ----------------
    Test Diagnostic:
    ----------------
    Testing 5==4
```

```
    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyEqual failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
            Actual    Expected    Error    RelativeError
            ______    ________    _____    _____________

              5          4          1          0.25

    Actual Value:
         5
    Expected Value:
         4

    ------------------
    Stack Information:
    ------------------
    In C:\work\ExampleTest.m (ExampleTest.testOne) at 4
================================================================================
Failure Summary:

     Name                 Failed  Incomplete  Reason(s)
    ===============================================================
     ExampleTest/testOne    X                 Failed by verification.
```

Only the test failures produce output to the screen.

See Also

`fprintf` | `matlab.unittest.plugins` | `matlab.unittest.plugins.OutputStream`

Introduced in R2014a

matlab.unittest.plugins.ToUniqueFile class

Package: `matlab.unittest.plugins`
Superclasses: `matlab.unittest.plugins.OutputStream`

Output stream to write text output to unique file

Description

The `ToUniqueFile` creates an output stream that writes text output to a unique, UTF-8 encoded file. Whenever text prints to this stream, the output stream opens the file, appends the text, and closes the file. Each instance of `ToUniqueFile` creates a file with a unique file name. This output stream is useful for running tests in parallel while redirecting output to a file.

MATLAB creates the unique file name for the output stream, but you can specify a file prefix and extension.

Construction

`matlab.unittest.plugins.ToUniqueFile(folder)` creates an `OutputStream` that writes text output to a unique file in the specified folder.

`matlab.unittest.plugins.ToUniqueFile(folder,Name,Value)` creates a unique file with additional options specified by one or more `Name,Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

`folder` — Name of folder
character vector | string scalar

Name of existing folder, specified as a character vector or string scalar. The output stream writes to a file in `folder`.

Example: `'myOutput'`

Example: `pwd`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `matlab.unittest.plugins.ToFile(pwd,'WithPrefix','myOutput_')` creates an output stream that writes to a file in the present working folder that starts with `'myOutput_'`.

`WithPrefix` — Prefix for file name
character vector | string scalar

Prefix for the file name, specified as a character vector or string scalar.

Example: 'outputA_'

`WithExtension` — Extension for file name
`'.txt'` (default) | character vector | string scalar

Extension for the file name, specified as a character vector or string scalar. By default, the file name has the extension `'.txt'`. Extensions must begin with a period.

Example: `'.xml'`

Properties

`Filename` — Full name of file
string scalar

Full name of the file to redirect text output from the plugin, returned as a string scalar.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Direct Output from Tests Run in Parallel

Produce TAP output for a suite of tests that you run in parallel. To avoid having the framework overwrite the TAP file, direct the output from each group of tests to a unique file. To run tests in parallel, this example requires Parallel Computing Toolbox.

Create the following parameterized test in a file in your current working folder.

```
classdef TestRand < matlab.unittest.TestCase
    properties (TestParameter)
        dim1 = createDimensionSizes;
        dim2 = createDimensionSizes;
        dim3 = createDimensionSizes;
        type = {'single','double'};
    end

    methods (Test)
        function testRepeatable(testCase,dim1,dim2,dim3)
            state = rng;
            firstRun = rand(dim1,dim2,dim3);
            rng(state)
            secondRun = rand(dim1,dim2,dim3);
            testCase.verifyEqual(firstRun,secondRun);
        end
        function testClass(testCase,dim1,dim2,type)
            testCase.verifyClass(rand(dim1,dim2,type),type)
        end
    end
end

function sizes = createDimensionSizes
    % Create logarithmicly spaced sizes up to 100
    sizes = num2cell(round(logspace(0,2,10)));
end
```

At the command prompt, create a folder for output files.

```
mkdir myOutput
```

Create a suite from `TestRand.m` and a test runner with terse output. The suite contains 1200 test elements.

```
suite = matlab.unittest.TestSuite.fromClass(?TestRand);
runner = matlab.unittest.TestRunner.withTextOutput('OutputDetail',1);
```

Create an instance of a stream that writes to a unique file in the `myOutput` folder. Then create a `TAPPlugin` and direct the output to the stream. Each created file begins with `'myTapFile_'` and ends with the `'.tap'` extension.

```
import matlab.unittest.plugins.ToUniqueFile;
import matlab.unittest.plugins.TAPPlugin;

stream = ToUniqueFile('myOutput','WithPrefix','myTapFile_','WithExtension','.tap');
plugin = TAPPlugin.producingOriginalFormat(stream);
```

Add the plugin to the test runner, and run the test suite in parallel. Your test groups and the output file names might vary.

```
runner.addPlugin(plugin);
result = runner.runInParallel(suite);
```

```
Split tests into 18 groups and running them on 6 workers.
----------------
Finished Group 5
----------------
..................................................
........................

----------------
Finished Group 6
----------------
..................................................
......................

----------------
Finished Group 4
----------------
..................................................
.........................

----------------
Finished Group 1
----------------
..................................................
...............................

----------------
Finished Group 3
```

```
----------------
..................................................
...........................

----------------
Finished Group 2
----------------
..................................................
............................

----------------
Finished Group 7
----------------
..................................................
......................

----------------
Finished Group 8
----------------
..................................................
....................

-----------------
Finished Group 11
-----------------
..................................................
................

----------------
Finished Group 9
----------------
..................................................
...................

-----------------
Finished Group 12
-----------------
..................................................
..............

-----------------
Finished Group 10
-----------------
..................................................
```

```
................

-----------------
Finished Group 16
-----------------
..................................................
........

-----------------
Finished Group 18
-----------------
..................................................
....

-----------------
Finished Group 17
-----------------
..................................................
.......

-----------------
Finished Group 13
-----------------
..................................................
............

-----------------
Finished Group 14
-----------------
..................................................
...........

-----------------
Finished Group 15
-----------------
..................................................
.........
```

View the output files in the `myOutput` folder. Since MATLAB ran the tests in 18 groups, the framework created 18 instances of the `ToUniqueFile` output stream. There are 18 associated output files.

```
dir myOutput
```

```
.
..
myTapFile_22dc996d-e1e9-44e2-af3a-e8e9c68c6941.tap
myTapFile_2de69eb4-591f-4456-9890-31626e57792f.tap
myTapFile_3e88b1fb-5679-4489-b9c9-a2b22ac76cb1.tap
myTapFile_4b660dae-9e33-4e89-bd1b-27c874749476.tap
myTapFile_56d584a8-2bf2-4677-ad25-5f268628c179.tap
myTapFile_632c3723-c300-40a6-8ffa-a7dbd0d07d65.tap
myTapFile_66276292-5062-489a-9219-cc2664f48fb8.tap
myTapFile_78d69693-720a-4a14-86b3-de687b1ddf91.tap
myTapFile_7df3915c-60de-4c7f-8968-b5260d4c2933.tap
myTapFile_827a6d46-54c6-4ee3-bfbb-0d46d4024fcf.tap
myTapFile_97af3692-7b4a-4f80-a81a-96fc0f86beed.tap
myTapFile_9cb0cdb1-4f30-40a0-8f5f-51da8af7bb86.tap
myTapFile_b8820e56-7c92-41eb-b040-94f55399766b.tap
myTapFile_c1bfd286-7fa4-4365-b456-4babf2a995da.tap
myTapFile_c53c2906-14a7-41eb-a87c-d1fd727e6d9e.tap
myTapFile_decbc713-84c0-4139-b3e4-d2b8c2e273bf.tap
myTapFile_e1687147-f8a7-4f6d-aea0-a3e885246dd6.tap
myTapFile_f78c98a7-6c14-4981-ae03-fc8ffbeddaf8.tap
```

See Also

`matlab.unittest.plugins` | `matlab.unittest.plugins.OutputStream` | `runInParallel`

Introduced in R2018a

matlab.unittest.plugins.testreport Package

Summary of classes in MATLAB Plugin Test Report Generation

Description

The `testreport` classes create plugins to generate test result reports. The `matlab.unittest.plugins.testreport` package consists of the following MATLAB classes.

Classes

matlab.unittest.plugins.testreport.DOCXTestReportPlugin	TestReportPlugin that creates a test report in .docx format
matlab.unittest.plugins.testreport.HTMLTestReportPlugin	TestReportPlugin that creates a test report in .html format
matlab.unittest.plugins.testreport.PDFTestReportPlugin	TestReportPlugin that creates a test report in .pdf format

See Also

`TestReportPlugin` | `matlab.unittest.plugins`

Introduced in R2016b

matlab.unittest.plugins.testreport.DOCXTestReportPlugin class

Package: `matlab.unittest.plugins.testreport`
Superclasses: `matlab.unittest.plugins.TestReportPlugin`

`TestReportPlugin` that creates a test report in `.docx` format

Description

A `DOCXTestReportPlugin` is an instance of the `TestReportPlugin` class that creates a test result report in `.docx` format. To create a `DOCXTestReportPlugin` instance, use the `matlab.unittest.plugins.TestReportPlugin.producingDOCX` static method.

Properties

`PageOrientation` — Report orientation
`'portrait'` (default) | `'landscape'`

Report orientation, specified as `'portrait'` or `'landscape'`. This property is read only. You can specify it during plugin construction.

These properties are inherited from `matlab.unittest.plugins.TestReportPlugin`:

`IncludeCommandWindowText` — Indicator if text output from Command Window is included
`false` (default) | `true`

This property is read-only.

Indicator if text output from Command Window is included, specified as `false` or `true` (`logical` 0 or 1). By default, `IncludeCommandWindowText` is `false` and the text output from the Command Window is excluded from the report. To include Command Window text in the report, specify `IncludeCommandWindowText` as `true` during plugin construction.

`IncludePassingDiagnostics` — Indicator if diagnostics for passing events are included
`false` (default) | `true`

This property is read-only.

Indicator if diagnostics for passing events are included, specified as `false` or `true` (`logical` 0 or 1). By default, `IncludePassingDiagnostics` is `false` and the diagnostics from passing events are excluded from the output. To include diagnostics from passing events in the output, specify `IncludePassingDiagnostics` as `true` during plugin construction.

`LoggingLevel` — Maximum verbosity level for logged diagnostics included by the plugin
`matlab.unittest.Verbosity.Terse` (default) | `matlab.unittest.Verbosity` enumeration object

This property is read-only.

Maximum verbosity level for logged diagnostics included by the plugin, returned as a `matlab.unittest.Verbosity` enumeration object. The plugin includes diagnostics that are logged at this level and below. By default this property value is `matlab.unittest.Verbosity.Terse`. You can specify a different logging level during plugin construction.

Logged diagnostics are diagnostics that you supply to the testing framework with a call to the `log` or `log` method.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

See Also

`matlab.unittest.plugins.TestReportPlugin` | `matlab.unittest.plugins.TestReportPlugin.producingDOCX`

Introduced in R2016b

matlab.unittest.plugins.testreport.HTMLTestReportPlugin class

Package: `matlab.unittest.plugins.testreport`
Superclasses: `matlab.unittest.plugins.TestReportPlugin`

`TestReportPlugin` that creates a test report in `.html` format

Description

An `HTMLTestReportPlugin` is an instance of the `TestReportPlugin` class that creates a test result report in `.html` format. To create a `HTMLTestReportPlugin` instance, use the `matlab.unittest.plugins.TestReportPlugin.producingHTML` static method.

Properties

`MainFile` — Name of main file
character vector

This property is read-only.

Name of the main file of the HTML report, specified as a character vector. Specify `MainFile` during plugin construction.

Example: `'index.html'`

Example: `'main.html'`

These properties are inherited from `matlab.unittest.plugins.TestReportPlugin`:

`IncludeCommandWindowText` — Indicator if text output from Command Window is included
`false` (default) | `true`

This property is read-only.

Indicator if text output from Command Window is included, specified as `false` or `true` (`logical` 0 or 1). By default, `IncludeCommandWindowText` is `false` and the text

output from the Command Window is excluded from the report. To include Command Window text in the report, specify `IncludeCommandWindowText` as `true` during plugin construction.

`IncludePassingDiagnostics` — Indicator if diagnostics for passing events are included
`false` (default) | `true`

This property is read-only.

Indicator if diagnostics for passing events are included, specified as `false` or `true` (`logical` 0 or 1). By default, `IncludePassingDiagnostics` is `false` and the diagnostics from passing events are excluded from the output. To include diagnostics from passing events in the output, specify `IncludePassingDiagnostics` as `true` during plugin construction.

`LoggingLevel` — Maximum verbosity level for logged diagnostics included by the plugin
`matlab.unittest.Verbosity.Terse` (default) | `matlab.unittest.Verbosity` enumeration object

This property is read-only.

Maximum verbosity level for logged diagnostics included by the plugin, returned as a `matlab.unittest.Verbosity` enumeration object. The plugin includes diagnostics that are logged at this level and below. By default this property value is `matlab.unittest.Verbosity.Terse`. You can specify a different logging level during plugin construction.

Logged diagnostics are diagnostics that you supply to the testing framework with a call to the `log` or `log` method.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

See Also

`matlab.unittest.plugins.TestReportPlugin` | `matlab.unittest.plugins.TestReportPlugin.producingHTML`

Introduced in R2017b

matlab.unittest.plugins.testreport.PDFTestReportPlugin class

Package: `matlab.unittest.plugins.testreport`
Superclasses: `matlab.unittest.plugins.TestReportPlugin`

`TestReportPlugin` that creates a test report in `.pdf` format

Description

A `PDFTestReportPlugin` is an instance of the `TestReportPlugin` class that creates a test result report in `PDF` format. To create a `PDFTestReportPlugin` instance, use the `matlab.unittest.plugins.TestReportPlugin.producingPDF` static method.

Properties

`PageOrientation` — Report orientation
`'portrait'` (default) | `'landscape'`

Report orientation, specified as `'portrait'` or `'landscape'`. This property is read only. You can specify it during plugin construction.

These properties are inherited from `matlab.unittest.plugins.TestReportPlugin`:

`IncludeCommandWindowText` — Indicator if text output from Command Window is included
`false` (default) | `true`

This property is read-only.

Indicator if text output from Command Window is included, specified as `false` or `true` (`logical` 0 or 1). By default, `IncludeCommandWindowText` is `false` and the text output from the Command Window is excluded from the report. To include Command Window text in the report, specify `IncludeCommandWindowText` as `true` during plugin construction.

`IncludePassingDiagnostics` — Indicator if diagnostics for passing events are included
`false` (default) | `true`

This property is read-only.

Indicator if diagnostics for passing events are included, specified as `false` or `true` (`logical` 0 or 1). By default, `IncludePassingDiagnostics` is `false` and the diagnostics from passing events are excluded from the output. To include diagnostics from passing events in the output, specify `IncludePassingDiagnostics` as `true` during plugin construction.

`LoggingLevel` — Maximum verbosity level for logged diagnostics included by the plugin
`matlab.unittest.Verbosity.Terse` (default) | `matlab.unittest.Verbosity` enumeration object

This property is read-only.

Maximum verbosity level for logged diagnostics included by the plugin, returned as a `matlab.unittest.Verbosity` enumeration object. The plugin includes diagnostics that are logged at this level and below. By default this property value is `matlab.unittest.Verbosity.Terse`. You can specify a different logging level during plugin construction.

Logged diagnostics are diagnostics that you supply to the testing framework with a call to the `log` or `log` method.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Tips

- PDF test reports are generated based on your system locale and the font families installed on your machine. When generating a report with a non-English locale, unless your machine has the Noto Sans CJK font families installed, the report may have pound sign characters (#) in place of Chinese, Japanese, and Korean characters.

See Also

`matlab.unittest.plugins.TestReportPlugin` | `matlab.unittest.plugins.TestReportPlugin.producingPDF`

Introduced in R2016b

matlab.unittest.plugins.TestReportPlugin class

Package: `matlab.unittest.plugins`

Plugin that produces a test result report

Description

`TestReportPlugin` creates a plugin that directs the `TestRunner` to produce a test result report. Using this plugin, you can produce readable and archivable test reports.

Construction

Instantiate a `TestReportPlugin` using one of its static methods:

- To produce a `.docx` report, use the `producingDOCX` static method.
- To produce a `PDF` report, use the `producingPDF` static method.

Properties

`IncludeCommandWindowText` — Indicator if text output from Command Window is included

`false` (default) | `true`

This property is read-only.

Indicator if text output from Command Window is included, specified as `false` or `true` (`logical` 0 or 1). By default, `IncludeCommandWindowText` is `false` and the text output from the Command Window is excluded from the report. To include Command Window text in the report, specify `IncludeCommandWindowText` as `true` during plugin construction.

`IncludePassingDiagnostics` — Indicator if diagnostics for passing events are included
`false` (default) | `true`

This property is read-only.

Indicator if diagnostics for passing events are included, specified as `false` or `true` (`logical` 0 or 1). By default, `IncludePassingDiagnostics` is `false` and the diagnostics from passing events are excluded from the output. To include diagnostics from passing events in the output, specify `IncludePassingDiagnostics` as `true` during plugin construction.

`LoggingLevel` — Maximum verbosity level for logged diagnostics included by the plugin
`matlab.unittest.Verbosity.Terse` (default) | `matlab.unittest.Verbosity` enumeration object

This property is read-only.

Maximum verbosity level for logged diagnostics included by the plugin, returned as a `matlab.unittest.Verbosity` enumeration object. The plugin includes diagnostics that are logged at this level and below. By default this property value is `matlab.unittest.Verbosity.Terse`. You can specify a different logging level during plugin construction.

Logged diagnostics are diagnostics that you supply to the testing framework with a call to the `log` or `log` method.

Methods

producingDOCX	Constructs plugin that produces `.docx` report
producingHTML	Constructs plugin that produces `.html` report
producingPDF	Constructs plugin that produces `.pdf` report

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Generate Test Result Report in `.docx` Format

Create a test suite from two test files, run the suite, and generate a `.docx` report of the results.

Create a new file in your working folder named `ScriptBasedTest.m` containing the following test script. The script includes two failing and incomplete tests.

```
%% Test double class
expSolution = 'double';
actSolution = ones;
assert(isa(actSolution,expSolution))

%% Test single class
expSolution = 'single';
actSolution = ones('single');
assert(isa(actSolution,expSolution))

%% Test uint16 class
expSolution = 'uint16';
actSolution = ones('uint16');
assert(isa(actSolution,expSolution))

%% Test that fails
assert(false==true);

%% Another test that fails
assert(strcmp('correlation','causation'))
```

Create a file named `ClassBasedTest.m` containing the following test class. The class includes a failing test that, with parameterization, results in nine failed test results.

```
classdef ClassBasedTest < matlab.unittest.TestCase
    properties (ClassSetupParameter)
        generator = {'twister','combRecursive','multFibonacci'};
    end
    properties (MethodSetupParameter)
        seed = {0,123,4294967295};
    end
    properties (TestParameter)
        dim1 = struct('small',1,'medium',2,'large',3);
```

```
        dim2 = struct('small',2,'medium',3,'large',4);
        dim3 = struct('small',3,'medium',4,'large',5);
        type = {'single','double'};
    end
    methods (TestClassSetup)
        function ClassSetup(testCase,generator)
            orig = rng;
            testCase.addTeardown(@rng,orig)
            rng(0, generator)
        end
    end
    methods (TestMethodSetup)
        function MethodSetup(testCase,seed)
            orig = rng;
            testCase.addTeardown(@rng,orig)
            rng(seed)
        end
    end
    methods (Test, ParameterCombination='sequential')
        function testSize(testCase,dim1,dim2,dim3)
            testCase.verifySize(rand(dim1,dim2,dim3),[dim1 dim2 dim3])
        end
    end
    methods (Test, ParameterCombination='pairwise')
        function testRepeatable(testCase,dim1,dim2,dim3)
            state = rng;
            firstRun = rand(dim1,dim2,dim3);
            rng(state)
            secondRun = rand(dim1,dim2,dim3);
            testCase.verifyEqual(firstRun,secondRun);
        end
    end
    methods (Test)
        function testClass(testCase,dim1,dim2,type)
            testCase.verifyClass(rand(dim1,dim2,type),type)
        end
    end
end
```

At the command prompt, create a test suite from both test files.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;
import matlab.unittest.plugins.TestReportPlugin;
```

```
suite = testsuite({'ScriptBasedTest','ClassBasedTest'})
```

```
suite = 

  1×284 Test array with properties:

    Name
    ProcedureName
    TestClass
    BaseFolder
    Parameterization
    SharedTestFixtures
    Tags

Tests Include:
   17 Unique Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

Create a silent test runner, so that there is no information output to the command window. Create a `TestReportPlugin` that sends output to the file `MyTestReport.docx`.

```
runner = TestRunner.withNoPlugins;
docxFile = 'MyTestReport.docx';
plugin = TestReportPlugin.producingDOCX(docxFile);
```

Add the plugin to the `TestRunner` and run the suite.

```
runner.addPlugin(plugin);
result = runner.run(suite)
```

```
Generating report. Please wait.
    Preparing content for the report.
    Adding content to the report.
    Writing report to file.
Report has been saved to: C:\work\MyTestReport.docx

result = 

  1×284 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
```

```
    Duration
    Details

Totals:
   282 Passed, 2 Failed, 2 Incomplete.
   0.73477 seconds testing time.
```

Open the test report.

```
open(docxFile)
```

Generate `.pdf` Report That Includes Passing Diagnostics

Create a test suite from a function-based test, run the suite, and generate a report of the results. Include passing diagnostics and the text output to the Command Window.

Create a new file in your working folder named `FunctionBasedTest.m` containing the following function-based test. The test file includes two failing tests.

```
%% Main function to generate tests
function tests = FunctionBasedTest
tests = functiontests(localfunctions);
end

%% Test Functions
function passingTest(testCase)
actSolution = 13*3+7*5;
expSolution = 74;
verifyEqual(testCase,actSolution,expSolution)
end

function failingTest(testCase)
actSolution = single(1);
verifyTrue(testCase,actSolution)
end

function anotherPassingTest(testCase)
verifyClass(testCase,string('some text'),'string')
end

function anotherFailingTest(testCase)
verifyTrue(testCase,strcmp('42','everything'))
end
```

At the command prompt, create a test suite from `FunctionBasedTest.m`. Create a test runner that displays output to the command window using the default plugin.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;
import matlab.unittest.plugins.TestReportPlugin;

suite = testsuite('FunctionBasedTest');
runner = TestRunner.withTextOutput;
```

Create a `TestReportPlugin` that sends output to the file `MyTestReport2.pdf`. Include passing diagnostics and text output from the Command Window in the report.

```
pdfFile = 'MyTestReport2.pdf';
plugin = TestReportPlugin.producingPDF(pdfFile,...
    'IncludingPassingDiagnostics',true,'IncludingCommandWindowText',true);
```

Add the plugin to the `TestRunner` and run the suite.

```
runner.addPlugin(plugin);
result = runner.run(suite);
```

```
Running FunctionBasedTest
.
================================================================================
Verification failed in FunctionBasedTest/failingTest.

    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyTrue failed.
    --> The value must be logical. It is of type "single".

    Actual single:
             1

    ------------------
    Stack Information:
    ------------------
    In C:\Work\FunctionBasedTest.m (failingTest) at 15
================================================================================
..
================================================================================
Verification failed in FunctionBasedTest/anotherFailingTest.
```

```
    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyTrue failed.
    --> The value must evaluate to "true".

    Actual logical:
           0

    ------------------
    Stack Information:
    ------------------
    In C:\Work\FunctionBasedTest.m (anotherFailingTest) at 23
================================================================================
.
Done FunctionBasedTest
__________

Failure Summary:

     Name                                  Failed  Incomplete  Reason(s)
    ===========================================================================================
     FunctionBasedTest/failingTest           X                 Failed by verification.
    -------------------------------------------------------------------------------------------
     FunctionBasedTest/anotherFailingTest    X                 Failed by verification.

Generating report. Please wait.
    Preparing content for the report.
    Adding content to the report.
    Writing report to file.
Report has been saved to: C:\Work\MyTestReport2.pdf
```

Open the test report.

```
open(pdfFile)
```

Create Test Report That Includes Artifacts

In a file in your current working folder, create the `FigurePropTest` test class. If the `failingTest` test method fails (it always does in this example), it uses a `FigureDiagnostic` to save the figure so you can examine it later.

```
classdef FigurePropTest < matlab.unittest.TestCase
    properties
        TestFigure
    end
    methods(TestMethodSetup)
        function createFigure(testCase)
            testCase.TestFigure = figure;
        end
    end
    methods(TestMethodTeardown)
        function closeFigure(testCase)
            close(testCase.TestFigure)
        end
    end
    methods(Test)
        function defaultCurrentPoint(testCase)
            cp = testCase.TestFigure.CurrentPoint;
            testCase.verifyEqual(cp,[0 0], ...
                'Default current point is incorrect')
        end
        function defaultCurrentObject(testCase)
            import matlab.unittest.constraints.IsEmpty
            co = testCase.TestFigure.CurrentObject;
            testCase.verifyThat(co,IsEmpty, ...
                'Default current object should be empty')
        end
        function failingTest(testCase)
            import matlab.unittest.diagnostics.FigureDiagnostic
            fig = testCase.TestFigure;
            ax = axes(fig);
            surf(ax,peaks)
            testCase.verifyEmpty(testCase.TestFigure.Children, ...
                FigureDiagnostic(testCase.TestFigure))
        end
    end
end
```

At the command prompt, create a test suite.

```
suite = testsuite('FigurePropTest');
```

Create a silent test runner that records diagnostics and generates a PDF report.

```
import matlab.unittest.plugins.DiagnosticsRecordingPlugin
import matlab.unittest.plugins.TestReportPlugin
```

```
runner = matlab.unittest.TestRunner.withNoPlugins;
runner.addPlugin(DiagnosticsRecordingPlugin);
runner.addPlugin(TestReportPlugin.producingPDF('MyTestReport.pdf'));
```

Change the default artifact root to your current working folder.

```
runner.ArtifactsRootFolder = pwd;
```

Run the tests. The third test fails.

```
results = runner.run(suite)
```

```
Generating report. Please wait.
    Preparing content for the report.
    Adding content to the report.
    Writing report to file.
Report has been saved to: C:\work\MyTestReport.pdf

results =

  1×3 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
   2 Passed, 1 Failed, 0 Incomplete.
   1.2295 seconds testing time.
```

Display the test diagnostic results for the third test. The test framework saved two artifacts related to the third test. By default, a `FigureDiagnostic` object saves a figure as both a PNG file and a FIG file.

```
results(3).Details.DiagnosticRecord.TestDiagnosticResults
```

```
ans =

  DiagnosticResult with properties:

         Artifacts: [1×2 matlab.unittest.diagnostics.FileArtifact]
    DiagnosticText: 'Figure saved to:↵--> C:\work\715b5416-5c52-4a53-bbec-837a5db57392\
```

Display the stored location of the first artifact.

```
results(3).Details.DiagnosticRecord.TestDiagnosticResults.Artifacts(1)
```

```
ans = 

  FileArtifact with properties:

        Name: "Figure_284d9501-2121-45a1-bf5c-18904ce11e8f.fig"
    Location: "C:\work\715b5416-5c52-4a53-bbec-837a5db57392"
    FullPath: "C:\work\715b5416-5c52-4a53-bbec-837a5db57392\Figure_284d9501-2121-45a1-b
```

To inspect the image related to the failed test, open the file at the location shown in the `FullPath` field. Additionally, since you generated a PDF test report, the image is captured in `MyTestReport.pdf`. The test report also contains the path to the artifacts.

See Also

`matlab.unittest.plugins.testreport.DOCXTestReportPlugin` | `matlab.unittest.plugins.testreport.HTMLTestReportPlugin` | `matlab.unittest.plugins.testreport.PDFTestReportPlugin`

Introduced in R2016b

matlab.unittest.plugins.TestReportPlugin.producingDOCX

Class: `matlab.unittest.plugins.TestReportPlugin`
Package: `matlab.unittest.plugins`

Constructs plugin that produces `.docx` report

Syntax

```
matlab.unittest.plugins.TestReportPlugin.producingDOCX
matlab.unittest.plugins.TestReportPlugin.producingDOCX(docxFile)
matlab.unittest.plugins.TestReportPlugin.producingDOCX(docxFile,
Name,Value)
```

Description

`matlab.unittest.plugins.TestReportPlugin.producingDOCX` constructs a plugin that produces a `.docx` report of test results in a temporary folder. This syntax is equivalent to `matlab.unittest.plugins.TestReportPlugin.producingDOCX([tempname '.docx'])`.

`matlab.unittest.plugins.TestReportPlugin.producingDOCX(docxFile)` saves the report to the file `docxFile`.

`matlab.unittest.plugins.TestReportPlugin.producingDOCX(docxFile, Name,Value)` constructs a plugin with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`docxFile` — Name of test report
character vector ending in `.docx`

Name of the test report that the plugin creates, specified as a character vector ending in `.docx`.

Example: `'myReportFile.docx'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `TestReportPlugin.producingDOCX(docxFile,'LoggingLevel',Verbosity.Detailed)` creates a plugin that includes diagnostics logged at and below the `Detailed` level.

`PageOrientation` — Report orientation
`'portrait'` (default) | `'landscape'`

Report orientation, specified as `'portrait'` or `'landscape'`. By default the plugin produces a report with portrait orientation.

`IncludingCommandWindowText` — Include Command Window text in report
`false` (default) | `true`

Include Command Window text in report, specified as `false` or `true`. By default, `IncludingCommandWindowText` is `false` and the text output from the Command Window is excluded from the report. To include Command Window text in the report, specify `IncludingCommandWindowText` as `true`.

Data Types: `logical`

`IncludingPassingDiagnostics` — Include passing event diagnostics
`false` (default) | `true`

Include passing event diagnostics, specified as `false` or `true`. By default, `IncludingPassingDiagnostics` is `false` and the diagnostics from passing events are excluded from the report. To include diagnostics from passing events in the report, specify `IncludingPassingDiagnostics` as `true`.

Data Types: `logical`

`LoggingLevel` — Maximum level of logged diagnostics
1 (default) | 0 | 2 | 3 | 4 | `matlab.unittest.Verbosity` enumeration | enumeration name as string or char vector

Maximum level at which logged diagnostics are included by the plugin instance, specified as an integer value from 0 through 4, a `matlab.unittest.Verbosity` enumeration object, or a string scalar or character vector corresponding to one of the predefined enumeration member names. The plugin includes diagnostics that are logged at this level and below. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration.

Numeric Representation	Enumeration Member Name	Verbosity Description
`0`	`None`	No information
`1`	`Terse`	Minimal information
`2`	`Concise`	Moderate amount of information
`3`	`Detailed`	Some supplemental information
`4`	`Verbose`	Lots of supplemental information

By default the plugin includes diagnostics logged at the `matlab.unittest.Verbosity.Terse` level (level 1). To exclude logged diagnostics, specify `LoggingLevel` as `Verbosity.None` (level 0).

Logged diagnostics are diagnostics that you supply to the testing framework with a call to the `log` or `log` method.

Examples

Generate Test Result Report in `.docx` Format

Create a test suite from two test files, run the suite, and generate a `.docx` report of the results.

Create a new file in your working folder named `ScriptBasedTest.m` containing the following test script. The script includes two failing and incomplete tests.

```
%% Test double class
expSolution = 'double';
actSolution = ones;
assert(isa(actSolution,expSolution))

%% Test single class
expSolution = 'single';
actSolution = ones('single');
assert(isa(actSolution,expSolution))

%% Test uint16 class
expSolution = 'uint16';
actSolution = ones('uint16');
assert(isa(actSolution,expSolution))

%% Test that fails
assert(false==true);

%% Another test that fails
assert(strcmp('correlation','causation'))
```

Create a file named `ClassBasedTest.m` containing the following test class. The class includes a failing test that, with parameterization, results in nine failed test results.

```
classdef ClassBasedTest < matlab.unittest.TestCase
    properties (ClassSetupParameter)
        generator = {'twister','combRecursive','multFibonacci'};
    end
    properties (MethodSetupParameter)
        seed = {0,123,4294967295};
    end
    properties (TestParameter)
        dim1 = struct('small',1,'medium',2,'large',3);
        dim2 = struct('small',2,'medium',3,'large',4);
        dim3 = struct('small',3,'medium',4,'large',5);
        type = {'single','double'};
    end
    methods (TestClassSetup)
        function ClassSetup(testCase,generator)
            orig = rng;
            testCase.addTeardown(@rng,orig)
            rng(0, generator)
        end
    end
    methods (TestMethodSetup)
```

```
        function MethodSetup(testCase,seed)
            orig = rng;
            testCase.addTeardown(@rng,orig)
            rng(seed)
        end
    end
    methods (Test, ParameterCombination='sequential')
        function testSize(testCase,dim1,dim2,dim3)
            testCase.verifySize(rand(dim1,dim2,dim3),[dim1 dim2 dim3])
        end
    end
    methods (Test, ParameterCombination='pairwise')
        function testRepeatable(testCase,dim1,dim2,dim3)
            state = rng;
            firstRun = rand(dim1,dim2,dim3);
            rng(state)
            secondRun = rand(dim1,dim2,dim3);
            testCase.verifyEqual(firstRun,secondRun);
        end
    end
    methods (Test)
        function testClass(testCase,dim1,dim2,type)
            testCase.verifyClass(rand(dim1,dim2,type),type)
        end
    end
end
```

At the command prompt, create a test suite from both test files.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;
import matlab.unittest.plugins.TestReportPlugin;

suite = testsuite({'ScriptBasedTest','ClassBasedTest'})
```

```
suite =

  1×284 Test array with properties:

    Name
    ProcedureName
    TestClass
    BaseFolder
    Parameterization
    SharedTestFixtures
```

```
    Tags

Tests Include:
   17 Unique Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

Create a silent test runner, so that there is no information output to the command window. Create a `TestReportPlugin` that sends output to the file `MyTestReport.docx`.

```
runner = TestRunner.withNoPlugins;
docxFile = 'MyTestReport.docx';
plugin = TestReportPlugin.producingDOCX(docxFile);
```

Add the plugin to the `TestRunner` and run the suite.

```
runner.addPlugin(plugin);
result = runner.run(suite)
```

```
Generating report. Please wait.
    Preparing content for the report.
    Adding content to the report.
    Writing report to file.
Report has been saved to: C:\work\MyTestReport.docx

result =

  1×284 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
   282 Passed, 2 Failed, 2 Incomplete.
   0.73477 seconds testing time.
```

Open the test report.

```
open(docxFile)
```

Generate .docx Report That Includes Passing Diagnostics

Create a test suite from a function-based test, run the suite, and generate a report of the results. Include passing diagnostics and the text output to the Command Window.

Create a new file in your working folder named `FunctionBasedTest.m` containing the following function-based test. The test file includes two failing tests.

```
%% Main function to generate tests
function tests = FunctionBasedTest
tests = functiontests(localfunctions);
end

%% Test Functions
function passingTest(testCase)
actSolution = 13*3+7*5;
expSolution = 74;
verifyEqual(testCase,actSolution,expSolution)
end

function failingTest(testCase)
actSolution = single(1);
verifyTrue(testCase,actSolution)
end

function anotherPassingTest(testCase)
verifyClass(testCase,string('some text'),'string')
end

function anotherFailingTest(testCase)
verifyTrue(testCase,strcmp('42','everything'))
end
```

At the command prompt, create a test suite from `FunctionBasedTest.m`. Create a test runner that displays output to the command window using the default plugin.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;
import matlab.unittest.plugins.TestReportPlugin;

suite = testsuite('FunctionBasedTest');
runner = TestRunner.withTextOutput;
```

Create a `TestReportPlugin` that sends output to the file `MyTestReport2.docx`. Include passing diagnostics and text output from the Command Window in the report.

```
docxFile = 'MyTestReport2.docx';
plugin = TestReportPlugin.producingDOCX(docxFile,...
    'IncludingPassingDiagnostics',true,'IncludingCommandWindowText',true);
```

Add the plugin to the `TestRunner` and run the suite.

```
runner.addPlugin(plugin);
result = runner.run(suite);
```

```
Running FunctionBasedTest
.
================================================================================
Verification failed in FunctionBasedTest/failingTest.

    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyTrue failed.
    --> The value must be logical. It is of type "single".

    Actual single:
             1

    ------------------
    Stack Information:
    ------------------
    In C:\Work\FunctionBasedTest.m (failingTest) at 15
================================================================================
..
================================================================================
Verification failed in FunctionBasedTest/anotherFailingTest.

    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyTrue failed.
    --> The value must evaluate to "true".

    Actual logical:
           0

    ------------------
    Stack Information:
    ------------------
    In C:\Work\FunctionBasedTest.m (anotherFailingTest) at 23
```

```
==============================================================================
.
Done FunctionBasedTest
_________

Failure Summary:

     Name                                  Failed  Incomplete  Reason(s)
    ==============================================================================
     FunctionBasedTest/failingTest           X                 Failed by verification.
    ------------------------------------------------------------------------------
     FunctionBasedTest/anotherFailingTest    X                 Failed by verification.

Generating report. Please wait.
    Preparing content for the report.
    Adding content to the report.
    Writing report to file.
Report has been saved to: C:\Work\MyTestReport2.docx
```

Open the test report.

```
open(docxFile)
```

See Also

`matlab.unittest.plugins.testreport.DOCXTestReportPlugin`

Introduced in R2016b

matlab.unittest.plugins.TestReportPlugin.producingHTML

Class: `matlab.unittest.plugins.TestReportPlugin`
Package: `matlab.unittest.plugins`

Constructs plugin that produces `.html` report

Syntax

```
matlab.unittest.plugins.TestReportPlugin.producingHTML
matlab.unittest.plugins.TestReportPlugin.producingHTML(htmlFolder)
matlab.unittest.plugins.TestReportPlugin.producingHTML( ___ ,
Name,Value)
```

Description

`matlab.unittest.plugins.TestReportPlugin.producingHTML` constructs a plugin that produces an `.html` report of test results in a temporary folder. Within that folder, the main file of the report is `index.html`. If you rerun the test suite with this plugin, then MATLAB overwrites the contents in the folder.

This syntax is equivalent to `matlab.unittest.plugins.TestReportPlugin.producingHTML(tempname)`.

`matlab.unittest.plugins.TestReportPlugin.producingHTML(htmlFolder)` saves the report to the `htmlFolder` folder.

`matlab.unittest.plugins.TestReportPlugin.producingHTML( ___ , Name,Value)` constructs a plugin with additional options specified by one or more `Name,Value` pair arguments. You can use this syntax with any of the arguments from the previous syntaxes.

Input Arguments

`htmlFolder` — Output folder
character vector | string scalar

Output folder, specified as a character vector or string scalar. `htmlFolder` can be a relative or absolute path. By default, within the folder, the main file of the report is `index.html`. To change the name of the main file, use the `'MainFile'` name-value pair argument.

Example: `'TestRunOutput'`

Example: `'C:\myWork\testResults'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `TestReportPlugin.producingHTML('myTestOutput','MainFile','main.html')` creates a plugin that outputs results to the `myTestOutput` folder, with a main file named `main.html` instead of `index.html`.

`MainFile` — Main file name
character vector | string scalar

Main `HTML` file name, specified as a character vector or string scalar.

Example: `'main.html'`

`IncludingCommandWindowText` — Include Command Window text in report
`false` (default) | `true`

Include Command Window text in report, specified as `false` or `true`. By default, `IncludingCommandWindowText` is `false` and the text output from the Command Window is excluded from the report. To include Command Window text in the report, specify `IncludingCommandWindowText` as `true`.

Data Types: `logical`

`IncludingPassingDiagnostics` — Include passing event diagnostics
`false` (default) | `true`

Include passing event diagnostics, specified as `false` or `true`. By default, `IncludingPassingDiagnostics` is `false` and the diagnostics from passing events are excluded from the report. To include diagnostics from passing events in the report, specify `IncludingPassingDiagnostics` as `true`.

Data Types: `logical`

`LoggingLevel` — Maximum level of logged diagnostics
1 (default) | 0 | 2 | 3 | 4 | `matlab.unittest.Verbosity` enumeration | enumeration name as string or char vector

Maximum level at which logged diagnostics are included by the plugin instance, specified as an integer value from 0 through 4, a `matlab.unittest.Verbosity` enumeration object, or a string scalar or character vector corresponding to one of the predefined enumeration member names. The plugin includes diagnostics that are logged at this level and below. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration.

Numeric Representation	Enumeration Member Name	Verbosity Description
`0`	`None`	No information
`1`	`Terse`	Minimal information
`2`	`Concise`	Moderate amount of information
`3`	`Detailed`	Some supplemental information
`4`	`Verbose`	Lots of supplemental information

By default the plugin includes diagnostics logged at the `matlab.unittest.Verbosity.Terse` level (level 1). To exclude logged diagnostics, specify `LoggingLevel` as `Verbosity.None` (level 0).

Logged diagnostics are diagnostics that you supply to the testing framework with a call to the `log` or `log` method.

Examples

Generate Test Result Report in `.html` Format

Create a test suite from two test files, run the suite, and generate a `.html` report of the results.

Create a new file in your working folder named `ScriptBasedTest.m` containing the following test script. The script includes two failing and incomplete tests.

```
%% Test double class
expSolution = 'double';
actSolution = ones;
assert(isa(actSolution,expSolution))

%% Test single class
expSolution = 'single';
actSolution = ones('single');
assert(isa(actSolution,expSolution))

%% Test uint16 class
expSolution = 'uint16';
actSolution = ones('uint16');
assert(isa(actSolution,expSolution))

%% Test that fails
assert(false==true);

%% Another test that fails
assert(strcmp('correlation','causation'))
```

Create a file named `ClassBasedTest.m` containing the following test class. The class includes a failing test that, with parameterization, results in nine failed tests.

```
classdef ClassBasedTest < matlab.unittest.TestCase
    properties (ClassSetupParameter)
        generator = {'twister','combRecursive','multFibonacci'};
    end
    properties (MethodSetupParameter)
        seed = {0,123,4294967295};
    end
    properties (TestParameter)
        dim1 = struct('small',1,'medium',2,'large',3);
```

```
        dim2 = struct('small',2,'medium',3,'large',4);
        dim3 = struct('small',3,'medium',4,'large',5);
        type = {'single','double'};
    end
    methods (TestClassSetup)
        function ClassSetup(testCase,generator)
            orig = rng;
            testCase.addTeardown(@rng,orig)
            rng(0, generator)
        end
    end
    methods (TestMethodSetup)
        function MethodSetup(testCase,seed)
            orig = rng;
            testCase.addTeardown(@rng,orig)
            rng(seed)
        end
    end
    methods (Test, ParameterCombination='sequential')
        function testSize(testCase,dim1,dim2,dim3)
            testCase.verifySize(rand(dim1,dim2,dim3),[dim1 dim2 dim3])
        end
    end
    methods (Test, ParameterCombination='pairwise')
        function testRepeatable(testCase,dim1,dim2,dim3)
            state = rng;
            firstRun = rand(dim1,dim2,dim3);
            rng(state)
            secondRun = rand(dim1,dim2,dim3);
            testCase.verifyEqual(firstRun,secondRun);
        end
    end
    methods (Test)
        function testClass(testCase,dim1,dim2,type)
            testCase.verifyClass(rand(dim1,dim2,type),type)
        end
    end
end
```

At the command prompt, create a test suite from both test files.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;
import matlab.unittest.plugins.TestReportPlugin;
```

```
suite = testsuite({'ScriptBasedTest','ClassBasedTest'})
```

```
suite = 

  1×284 Test array with properties:

    Name
    ProcedureName
    TestClass
    BaseFolder
    Parameterization
    SharedTestFixtures
    Tags

Tests Include:
   17 Unique Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

Create a silent test runner, so that there is no information output to the Command Window. Create a `TestReportPlugin` that generates a `.html` test report in a folder named `myResults`.

```
runner = TestRunner.withNoPlugins;
htmlFolder = 'myResults';
plugin = TestReportPlugin.producingHTML(htmlFolder);
```

Add the plugin to the `TestRunner` and run the suite.

```
runner.addPlugin(plugin);
result = runner.run(suite)
```

```
Generating report. Please wait.
    Preparing content for the report.
    Adding content to the report.
    Writing report to file.
Report has been saved to: C:\work\myResults\index.html

result = 

  1×284 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
```

```
    Duration
    Details

Totals:
   282 Passed, 2 Failed, 2 Incomplete.
   1.6712 seconds testing time.
```

Open the test report by clicking the name of the saved file. In this example the file name is `C:\work\myResults\index.html`.

See Also

`matlab.unittest.plugins.testreport.HTMLTestReportPlugin`

Introduced in R2017b

matlab.unittest.plugins.TestReportPlugin.producingPDF

Class: `matlab.unittest.plugins.TestReportPlugin`
Package: `matlab.unittest.plugins`

Constructs plugin that produces `.pdf` report

Syntax

```
matlab.unittest.plugins.TestReportPlugin.producingPDF
matlab.unittest.plugins.TestReportPlugin.producingPDF(pdfFile)
matlab.unittest.plugins.TestReportPlugin.producingPDF(pdfFile,
Name,Value)
```

Description

`matlab.unittest.plugins.TestReportPlugin.producingPDF` constructs a plugin that produces a `PDF` report of test results in a temporary folder. This syntax is equivalent to `matlab.unittest.plugins.TestReportPlugin.producingPDF([tempname '.pdf'])`.

`matlab.unittest.plugins.TestReportPlugin.producingPDF(pdfFile)` saves the report to the file `pdfFile`.

`matlab.unittest.plugins.TestReportPlugin.producingPDF(pdfFile, Name,Value)` constructs a plugin with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`pdfFile` — Name of test report
character vector ending in `.pdf`

Name of the test report that the plugin creates, specified as a character vector ending in `.pdf`.

Example: `pdfFile = 'myReportFile.pdf'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `TestReportPlugin.producingPDF(pdfFile,'PageOrientation','landscape')` creates a plugin that generates a report in landscape orientation.

`PageOrientation` — Report orientation
`'portrait'` (default) | `'landscape'`

Report orientation, specified as `'portrait'` or `'landscape'`. By default the plugin produces a report with portrait orientation.

`IncludingCommandWindowText` — Include Command Window text in report
`false` (default) | `true`

Include Command Window text in report, specified as `false` or `true`. By default, `IncludingCommandWindowText` is `false` and the text output from the Command Window is excluded from the report. To include Command Window text in the report, specify `IncludingCommandWindowText` as `true`.

Data Types: `logical`

`IncludingPassingDiagnostics` — Include passing event diagnostics
`false` (default) | `true`

Include passing event diagnostics, specified as `false` or `true`. By default, `IncludingPassingDiagnostics` is `false` and the diagnostics from passing events are excluded from the report. To include diagnostics from passing events in the report, specify `IncludingPassingDiagnostics` as `true`.

Data Types: `logical`

`LoggingLevel` — Maximum level of logged diagnostics

1 (default) | 0 | 2 | 3 | 4 | `matlab.unittest.Verbosity` enumeration | enumeration name as string or char vector

Maximum level at which logged diagnostics are included by the plugin instance, specified as an integer value from 0 through 4, a `matlab.unittest.Verbosity` enumeration object, or a string scalar or character vector corresponding to one of the predefined enumeration member names. The plugin includes diagnostics that are logged at this level and below. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration.

Numeric Representation	Enumeration Member Name	Verbosity Description
`0`	`None`	No information
`1`	`Terse`	Minimal information
`2`	`Concise`	Moderate amount of information
`3`	`Detailed`	Some supplemental information
`4`	`Verbose`	Lots of supplemental information

By default the plugin includes diagnostics logged at the `matlab.unittest.Verbosity.Terse` level (level 1). To exclude logged diagnostics, specify `LoggingLevel` as `Verbosity.None` (level 0).

Logged diagnostics are diagnostics that you supply to the testing framework with a call to the `log` or `log` method.

Examples

Generate Test Result Report in `.pdf` Format

Create a test suite from two test files, run the suite, and generate a `.pdf` report of the results.

Create a new file in your working folder named `ScriptBasedTest.m` containing the following test script. The script includes two failing and incomplete tests.

```
%% Test double class
expSolution = 'double';
actSolution = ones;
assert(isa(actSolution,expSolution))

%% Test single class
expSolution = 'single';
actSolution = ones('single');
assert(isa(actSolution,expSolution))

%% Test uint16 class
expSolution = 'uint16';
actSolution = ones('uint16');
assert(isa(actSolution,expSolution))

%% Test that fails
assert(false==true);

%% Another test that fails
assert(strcmp('correlation','causation'))
```

Create a file named `ClassBasedTest.m` containing the following test class. The class includes a failing test that, with parameterization, results in nine failed test results.

```
classdef ClassBasedTest < matlab.unittest.TestCase
    properties (ClassSetupParameter)
        generator = {'twister','combRecursive','multFibonacci'};
    end
    properties (MethodSetupParameter)
        seed = {0,123,4294967295};
    end
    properties (TestParameter)
        dim1 = struct('small',1,'medium',2,'large',3);
        dim2 = struct('small',2,'medium',3,'large',4);
        dim3 = struct('small',3,'medium',4,'large',5);
        type = {'single','double'};
    end
    methods (TestClassSetup)
        function ClassSetup(testCase,generator)
            orig = rng;
            testCase.addTeardown(@rng,orig)
            rng(0, generator)
        end
    end
    methods (TestMethodSetup)
```

```
        function MethodSetup(testCase,seed)
            orig = rng;
            testCase.addTeardown(@rng,orig)
            rng(seed)
        end
    end
    methods (Test, ParameterCombination='sequential')
        function testSize(testCase,dim1,dim2,dim3)
            testCase.verifySize(rand(dim1,dim2,dim3),[dim1 dim2 dim3])
        end
    end
    methods (Test, ParameterCombination='pairwise')
        function testRepeatable(testCase,dim1,dim2,dim3)
            state = rng;
            firstRun = rand(dim1,dim2,dim3);
            rng(state)
            secondRun = rand(dim1,dim2,dim3);
            testCase.verifyEqual(firstRun,secondRun);
        end
    end
    methods (Test)
        function testClass(testCase,dim1,dim2,type)
            testCase.verifyClass(rand(dim1,dim2,type),type)
        end
    end
end
```

At the command prompt, create a test suite from both test files.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;
import matlab.unittest.plugins.TestReportPlugin;

suite = testsuite({'ScriptBasedTest','ClassBasedTest'})
```

```
suite =

  1×284 Test array with properties:

    Name
    ProcedureName
    TestClass
    BaseFolder
    Parameterization
    SharedTestFixtures
```

```
    Tags

Tests Include:
   17 Unique Parameterizations, 0 Shared Test Fixture Classes, 0 Tags..
```

Create a silent test runner, so that there is no information output to the command window. Create a `TestReportPlugin` that sends output to the file `MyTestReport.pdf`.

```
runner = TestRunner.withNoPlugins;
pdfFile = 'MyTestReport.pdf';
plugin = TestReportPlugin.producingPDF(pdfFile);
```

Add the plugin to the `TestRunner` and run the suite.

```
runner.addPlugin(plugin);
result = runner.run(suite)
```

```
Generating report. Please wait.
    Preparing content for the report.
    Adding content to the report.
    Writing report to file.
Report has been saved to: C:\work\MyTestReport.pdf

result =

  1×284 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
   282 Passed, 2 Failed, 2 Incomplete.
   2.2054 seconds testing time.
```

Open the test report.

```
open(pdfFile)
```

Generate .pdf Report That Includes Passing Diagnostics

Create a test suite from a function-based test, run the suite, and generate a report of the results. Include passing diagnostics and the text output to the Command Window.

Create a new file in your working folder named `FunctionBasedTest.m` containing the following function-based test. The test file includes two failing tests.

```
%% Main function to generate tests
function tests = FunctionBasedTest
tests = functiontests(localfunctions);
end

%% Test Functions
function passingTest(testCase)
actSolution = 13*3+7*5;
expSolution = 74;
verifyEqual(testCase,actSolution,expSolution)
end

function failingTest(testCase)
actSolution = single(1);
verifyTrue(testCase,actSolution)
end

function anotherPassingTest(testCase)
verifyClass(testCase,string('some text'),'string')
end

function anotherFailingTest(testCase)
verifyTrue(testCase,strcmp('42','everything'))
end
```

At the command prompt, create a test suite from `FunctionBasedTest.m`. Create a test runner that displays output to the command window using the default plugin.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;
import matlab.unittest.plugins.TestReportPlugin;

suite = testsuite('FunctionBasedTest');
runner = TestRunner.withTextOutput;
```

Create a `TestReportPlugin` that sends output to the file `MyTestReport2.pdf`. Include passing diagnostics and text output from the Command Window in the report.

```
pdfFile = 'MyTestReport2.pdf';
plugin = TestReportPlugin.producingPDF(pdfFile,...
    'IncludingPassingDiagnostics',true,'IncludingCommandWindowText',true);
```

Add the plugin to the `TestRunner` and run the suite.

```
runner.addPlugin(plugin);
result = runner.run(suite);
```

```
Running FunctionBasedTest
.
================================================================================
Verification failed in FunctionBasedTest/failingTest.

    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyTrue failed.
    --> The value must be logical. It is of type "single".

    Actual single:
             1

    ------------------
    Stack Information:
    ------------------
    In C:\Work\FunctionBasedTest.m (failingTest) at 15
================================================================================
..
================================================================================
Verification failed in FunctionBasedTest/anotherFailingTest.

    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyTrue failed.
    --> The value must evaluate to "true".

    Actual logical:
           0

    ------------------
    Stack Information:
    ------------------
    In C:\Work\FunctionBasedTest.m (anotherFailingTest) at 23
```

```
==============================================================================
.
Done FunctionBasedTest
_________

Failure Summary:

     Name                                  Failed  Incomplete  Reason(s)
    ==============================================================================
     FunctionBasedTest/failingTest           X                 Failed by verification.
    ------------------------------------------------------------------------------
     FunctionBasedTest/anotherFailingTest    X                 Failed by verification.

Generating report. Please wait.
    Preparing content for the report.
    Adding content to the report.
    Writing report to file.
Report has been saved to: C:\Work\MyTestReport2.pdf
```

Open the test report.

```
open(pdfFile)
```

Tips

- PDF test reports are generated based on your system locale and the font families installed on your machine. When generating a report with a non-English locale, unless your machine has the Noto Sans CJK font families installed, the report may have pound sign characters (#) in place of Chinese, Japanese, and Korean characters.

See Also

`matlab.unittest.plugins.testreport.PDFTestReportPlugin`

Introduced in R2016b

matlab.unittest.plugins.TestRunnerPlugin class

Package: `matlab.unittest.plugins`

Plugin interface for extending TestRunner

Description

The `TestRunnerPlugin` interface enables extension of the `matlab.unittest.TestRunner`. To customize a test run, create a subclass of `TestRunnerPlugin` and override select methods. `TestRunnerPlugin` provides you with a default implementation. Override only the methods that are required to achieve your customization. Every method you implement must invoke its corresponding superclass method, passing along the same instance of `pluginData` that it receives.

To run tests with this extension, add the custom `TestRunnerPlugin` to the `TestRunner` by using the `addPlugin` method of `TestRunner`.

Methods

createSharedTestFixture	Extend creation of shared test fixture instances
createTestClassInstance	Extend creation of class-level `TestCase` instances
createTestMethodInstance	Extend creation of method-level `TestCase` instances
reportFinalizedResult	Enable reporting of finalized test results
reportFinalizedSuite	Extend reporting of finalized `TestSuite` array
runSession	Extend running of test session
runTest	Extend running of single `Test` element
runTestClass	Extend running of `Test` elements from same class or function
runTestSuite	Extend running of `TestSuite` array
runTestMethod	Extend running of single test method
setupSharedTestFixture	Extend setting up shared test fixture
setupTestClass	Extend setting up test class
setupTestMethod	Extend setting up test method
teardownSharedTestFixture	Extend tearing down shared test fixture
teardownTestClass	Extend tearing down test class
teardownTestMethod	Extend tearing down test method

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Tips

- To run tests in parallel with a `TestRunnerPlugin` instance, your plugin should subclass the `matlab.unittest.plugins.Parallelizable` interface. Running tests in parallel requires Parallel Computing Toolbox.

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.plugins.Parallelizable` | `matlab.unittest.plugins.plugindata`

Topics

"Write Plugins to Extend TestRunner"
"Create Custom Plugin"
"Run Tests in Parallel with Custom Plugin"

Introduced in R2014a

createSharedTestFixture

Class: `matlab.unittest.plugins.TestRunnerPlugin`
Package: `matlab.unittest.plugins`

Extend creation of shared test fixture instances

Syntax

```
f = createSharedTestFixture(plugin,pluginData)
```

Description

`f = createSharedTestFixture(plugin,pluginData)` extends the creation of shared test fixtures and returns the modified `Fixture` instance, `f`. The testing framework uses the fixture instance to customize running tests that use shared fixtures. The framework evaluates this method within the scope of the `runTestSuite` method of `TestRunnerPlugin` for each shared test fixture it needs to set up. A typical implementation of this method is to add listeners to various events originating from the shared test fixture instance. Since the `Fixture` inherits from the `handle` class, add listeners by calling the `addlistener` method from within the `createSharedTestFixture` method.

Input Arguments

`plugin`

Plugin object, specified as an instance of the `matlab.unittest.plugins.TestRunnerPlugin` class.

`pluginData`

Shared test fixture creation information, specified as an instance of the `matlab.unittest.plugins.plugindata.TestContentCreationPluginData` class. The testing framework uses this information to describe the test content to the plugin.

Examples

Extend Creation of Shared Test Fixture Instances

Extend the running of tests to count the number of shared test fixture assertion failures.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin

    properties (SetAccess = private)
        FixtureAssertionFailureData = {};
    end

    methods (Access = protected)
        function fixture = createSharedTestFixture(plugin, pluginData)
            % Invoke the super class method
            fixture = createSharedTestFixture@...
                matlab.unittest.plugins.TestRunnerPlugin(plugin, pluginData);

            % Get the fixture name
            fixtureName = pluginData.Name;

            % Add a listener to fixture assertion failures
            % and capture the qualification failure information
            fixture.addlistener('AssertionFailed', @(~,evd) ...
                plugin.captureFixtureAssertionFailureData(evd, fixtureName))
        end
    end

    methods (Access = private)
        function captureFixtureAssertionFailureData(plugin, eventData, fixtureName)
            plugin.FixtureAssertionFailureData{end+1} = struct(...
                'FixtureName', fixtureName, ...
                'ActualValue', eventData.ActualValue, ...
                'Constraint' , eventData.Constraint, ...
                'Stack'      , eventData.Stack);
        end
    end
end
```

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.fixtures.Fixture` | `matlab.unittest.plugins.plugindata.TestContentCreationPluginData` | `matlab.unittest.qualifications.ExceptionEventData` | `matlab.unittest.qualifications.QualificationEventData`

Topics

"Write Plugins to Extend TestRunner"
"Create Custom Plugin"
"Run Tests in Parallel with Custom Plugin"

Introduced in R2014a

createTestClassInstance

Class: `matlab.unittest.plugins.TestRunnerPlugin`
Package: `matlab.unittest.plugins`

Extend creation of class-level `TestCase` instances

Syntax

```
tc = createTestClassInstance(plugin,pluginData)
```

Description

`tc = createTestClassInstance(plugin,pluginData)` extends the creation of class-level `TestCase` instances, and returns the modified `TestCase` instance, `tc`. The testing framework uses the `TestCase` instance to customize running tests that belong to the same test class. The framework evaluates this method within the scope of the `runTestClass` method of `TestRunnerPlugin`. A typical implementation of this method is to add listeners to various events originating from the class level instance. Since the `TestCase` inherits from the `handle` class, add listeners by calling the `addlistener` method from within the `createTestClassInstance` method. For each class, the testing framework passes the instance to any method with the `TestClassSetup` or `TestClassTeardown` attribute.

Input Arguments

`plugin`

Plugin object, specified as an instance of the `matlab.unittest.plugins.TestRunnerPlugin` class.

`pluginData`

Class-level `TestCase` creation information, specified as an instance of the `matlab.unittest.plugins.plugindata.TestContentCreationPluginData` class. The testing framework uses this information to describe the test content to the plugin.

Examples

Extend Creation of Class-Level TestCase Instances

Extend the running of tests to count the number of class-level assumption failures.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin

    properties (SetAccess = private)
        TestClassAssumptionFailureData = {};
    end

    methods (Access = protected)
        function testCase = createTestClassInstance(plugin,pluginData)
            % Invoke super class method
            testCase = createTestClassInstance@...
                matlab.unittest.plugins.TestRunnerPlugin(plugin,pluginData);

            % Get the test class name
            instanceName = pluginData.Name;

            % Add a listener to capture assumption failures
            testCase.addlistener('AssumptionFailed', @(~,evd) ...
                plugin.captureClassLevelAssumptionFailureData(evd,instanceName))
        end
    end

    methods (Access = private)
        function captureClassLevelAssumptionFailureData(plugin,eventData,instanceName)
            plugin.TestClassAssumptionFailureData{end+1} = struct(...
                'InstanceName', instanceName, ...
                'ActualValue'  , eventData.ActualValue, ...
                'Constraint'   , eventData.Constraint, ...
                'Stack'        , eventData.Stack);
        end
    end
end
```

See Also

`matlab.unittest.TestCase` | `matlab.unittest.TestRunner` | `matlab.unittest.plugins.plugindata.TestContentCreationPluginData` | `matlab.unittest.qualifications.ExceptionEventData` | `matlab.unittest.qualifications.QualificationEventData`

Topics

"Write Plugins to Extend TestRunner"
"Create Custom Plugin"
"Run Tests in Parallel with Custom Plugin"

Introduced in R2014a

createTestMethodInstance

Class: `matlab.unittest.plugins.TestRunnerPlugin`
Package: `matlab.unittest.plugins`

Extend creation of method-level `TestCase` instances

Syntax

```
tc = createTestMethodInstance(plugin,pluginData)
```

Description

`tc = createTestMethodInstance(plugin,pluginData)` extends the creation of method-level `TestCase` instances, and returns the modified `TestCase` instance, `tc`. The testing framework evaluates this method within the scope of the `runTest` method of `TestRunnerPlugin`. A typical implementation of this method is to add listeners to various events originating from the method-level instance. Since the `TestCase` inherits from the `handle` class, add listeners by calling the `addlistener` method from within the `createTestMethodInstance` method. The testing framework creates instances for every element of the `matlab.unittest.Test` array and passes each instance to its corresponding `Test` methods and to any method with the `TestMethodSetup` or `TestMethodTeardown` attribute.

Input Arguments

plugin

Plugin object, specified as an instance of the `matlab.unittest.plugins.TestRunnerPlugin` class.

pluginData

Method-level `TestCase` creation information, specified as an instance of the `matlab.unittest.plugins.plugindata.TestContentCreationPluginData` class. The testing framework uses this information to describe the test content to the plugin.

Examples

Implement createMethodInstance Method

Add a listener to listen for assumption failures. Use the helper method, `captureMethodLevelAssumptionFailureData`, to populate the `TestMethodAssumptionFailureData` property.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin

    properties (SetAccess = private)
        TestMethodAssumptionFailureData = {};
    end

    methods (Access = protected)
        function testCase = createTestMethodInstance(plugin, pluginData)
            testCase = createTestMethodInstance@...
                matlab.unittest.plugins.TestRunnerPlugin(plugin, pluginData);

            instanceName = pluginData.Name;
            testCase.addlistener('AssumptionFailed', @(~,evd) ...
                plugin.captureMethodLevelAssumptionFailureData(evd,instanceName))
        end
    end

    methods (Access = private)
        function captureMethodLevelAssumptionFailureData(...
                plugin, eventData, instanceName)
            plugin.TestMethodAssumptionFailureData{end+1} = struct(...
                'InstanceName', instanceName, ...
                'ActualValue'     , eventData.ActualValue, ...
                'Constraint'      , eventData.Constraint, ...
                'Stack'           , eventData.Stack);
        end
    end

end
```

See Also

`createTestClassInstance` | `matlab.unittest.TestCase` | `matlab.unittest.TestRunner` | `matlab.unittest.plugins.plugindata.TestContentCreationPluginData` | `matlab.unittest.qualifications.ExceptionEventData` | `matlab.unittest.qualifications.QualificationEventData`

Topics

"Write Plugins to Extend TestRunner"
"Create Custom Plugin"

"Run Tests in Parallel with Custom Plugin"

Introduced in R2014a

reportFinalizedResult

Class: `matlab.unittest.plugins.TestRunnerPlugin`
Package: `matlab.unittest.plugins`

Enable reporting of finalized test results

Syntax

```
reportFinalizedResult(plugin,pluginData)
```

Description

`reportFinalizedResult(plugin,pluginData)` enables the reporting of finalized test results. A test result is finalized when no remaining test content can modify it. The `TestRunner` might modify previously run test results when it executes code inside `TestClassTeardown` methods or tears down shared test fixtures, for example.

A plugin that overrides the `reportFinalizedResult` method is recommended for streaming or inline reporting of test results. If you implement this method, the `TestRunner` reports the results as soon as they are finalized. The plugin can then report test results while the test suite is still running, rather than waiting until the entire suite is complete. The testing framework can evaluate `reportFinalizedResult` within the scope of the `runTestSuite`, `runTestClass`, or `runTest` methods of `TestRunnerPlugin`.

Input Arguments

`plugin` — plugin object
`matlab.unittest.plugins.TestRunnerPlugin` object

Plugin object, specified as an instance of the `matlab.unittest.plugins.TestRunnerPlugin` class.

`pluginData` — Finalized test information
`matlab.unittest.plugins.plugindata.FinalizedResultPluginData` object

Finalized test information, specified as an instance of the `matlab.unittest.plugins.plugindata.FinalizedResultPluginData` class. The testing framework uses this information to describe the test content to the plugin.

Examples

Display Status of Each Finalized Test Element

Display the status of each `Test` element by using the `reportFinalizedResult` method.

Create the plugin file `ExamplePlugin.m` in your current folder.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin
    methods (Access = protected)
        function reportFinalizedResult(plugin, pluginData)
            thisResult = pluginData.TestResult;
            if thisResult.Passed
                status = 'PASSED';
            elseif thisResult.Failed
                status = 'FAILED';
            elseif thisResult.Incomplete
                status = 'SKIPPED';
            end
            fprintf('%s: %s in %f seconds.\n',thisResult.Name,...
                status,thisResult.Duration)
            reportFinalizedResult@...
                matlab.unittest.plugins.TestRunnerPlugin(plugin,pluginData);
        end
    end
end
```

Create the test class file `ExampleTest.m` in your current folder.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase)
            testCase.assertGreaterThan(5,1)
        end
        function testTwo(testCase)
            wrongAnswer = 'wrong';
            testCase.verifyEmpty(wrongAnswer,'Not Empty')
            testCase.verifyClass(wrongAnswer,'double','Not double')
        end
        function testThree(testCase)
            testCase.assumeEqual(7*2,13,'Values not equal')
        end
        function testFour(testCase)
```

```
            testCase.verifyEqual(3+2,5);
        end
    end
end
```

At the command prompt, create a test suite, add the plugin to the `TestRunner`, and run the tests.

```
import matlab.unittest.TestSuite
import matlab.unittest.TestRunner

suite = TestSuite.fromClass(?ExampleTest);
runner = TestRunner.withNoPlugins;
runner.addPlugin(ExamplePlugin)
result = runner.run(suite);
```

```
ExampleTest/testOne: PASSED in 0.002216 seconds.
ExampleTest/testTwo: FAILED in 0.006105 seconds.
ExampleTest/testThree: SKIPPED in 0.007458 seconds.
ExampleTest/testFour: PASSED in 0.004865 seconds.
```

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.plugins.plugindata.FinalizedResultPluginData`

Topics

"Write Plugins to Extend TestRunner"
"Create Custom Plugin"
"Run Tests in Parallel with Custom Plugin"
"Plugin to Generate Custom Test Output Format"

Introduced in R2015b

reportFinalizedSuite

Class: `matlab.unittest.plugins.TestRunnerPlugin`
Package: `matlab.unittest.plugins`

Extend reporting of finalized `TestSuite` array

Syntax

```
reportFinalizedSuite(plugin,pluginData)
```

Description

`reportFinalizedSuite(plugin,pluginData)` extends the reporting of a finalized portion of the original test suite. A test suite portion is finalized when no remaining test content can modify the results associated with its elements. The testing framework evaluates this method as many times as the number of groups into which the entire `TestSuite` array is divided.

An example of reporting test results for finalized test suite portions is when tests run in parallel (requires Parallel Computing Toolbox). In this case, the testing framework divides the original `TestSuite` array into separate groups and assigns them to workers on the current parallel pool. A plugin that overrides the `reportFinalizedSuite` method can report test group results as soon as they are finalized by a worker, rather than waiting until the entire test suite is complete.

Input Arguments

`plugin` — Plugin object
`matlab.unittest.plugins.TestRunnerPlugin` object

Plugin object, specified as an instance of the `matlab.unittest.plugins.TestRunnerPlugin` class.

`pluginData` — Finalized test suite portion information
`matlab.unittest.plugins.plugindata.FinalizedSuitePluginData` object

Finalized test suite portion information, specified as an instance of the `matlab.unittest.plugins.plugindata.FinalizedSuitePluginData` class. The testing framework uses this information to describe the test content to the plugin.

Examples

Display Finalized Test Suite Information

Override the `reportFinalizedSuite` method in the plugin class to display a summary for a group of tests once the group is finalized.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin
    methods (Access = protected)
        function reportFinalizedSuite(plugin, pluginData)

            % Inspect pluginData to get finalized TestSuite information
            groupNumber = pluginData.Group;
            totalGroups = pluginData.NumGroups;
            suiteSize = numel(pluginData.TestSuite);
            fprintf('### Finished Running %d tests in group %d out of %d groups\n',...
                suiteSize, groupNumber, totalGroups)

            % Invoke the superclass method
            reportFinalizedSuite@matlab.unittest.plugins.TestRunnerPlugin(plugin, pluginData);
        end
    end
end
```

See Also

`matlab.unittest.TestResult` | `matlab.unittest.TestSuite` | `matlab.unittest.plugins.plugindata.FinalizedSuitePluginData` | `retrieveFrom`

Topics

"Write Plugins to Extend TestRunner"
"Create Custom Plugin"
"Run Tests in Parallel with Custom Plugin"

Introduced in R2019b

runSession

Class: `matlab.unittest.plugins.TestRunnerPlugin`
Package: `matlab.unittest.plugins`

Extend running of test session

Syntax

```
runSession(plugin,pluginData)
```

Description

`runSession(plugin,pluginData)` extends the running of the original `TestSuite` array passed by the testing framework to the `TestRunner` within a test session. The framework evaluates this method one time for the entire `TestSuite` array.

Input Arguments

`plugin` — Plugin object
`matlab.unittest.plugins.TestRunnerPlugin` object

Plugin object, specified as an instance of the `matlab.unittest.plugins.TestRunnerPlugin` class.

`pluginData` — Entire test suite information
`matlab.unittest.plugins.plugindata.RunPluginData` object

Entire test suite information, specified as an instance of the `matlab.unittest.plugins.plugindata.RunPluginData` class. The testing framework uses this information to describe the test content to the plugin.

Examples

Display Number of Running Test Elements

Override the `runSession` method in the plugin class to display the number of elements in the entire `TestSuite` array.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin

    methods (Access = protected)
        function runSession(plugin, pluginData)

            % Inspect pluginData to get TestSuite size
            suiteSize = numel(pluginData.TestSuite);
            fprintf('### Running a total of %d tests\n', suiteSize)

            % Invoke the superclass method
            runSession@matlab.unittest.plugins.TestRunnerPlugin(plugin, pluginData);
        end
    end

end
```

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.TestSuite` | `matlab.unittest.plugins.Parallelizable` | `matlab.unittest.plugins.plugindata.RunPluginData`

Topics

"Write Plugins to Extend TestRunner"
"Create Custom Plugin"
"Run Tests in Parallel with Custom Plugin"

Introduced in R2019b

runTest

Class: `matlab.unittest.plugins.TestRunnerPlugin`
Package: `matlab.unittest.plugins`

Extend running of single `Test` element

Syntax

```
runTest(plugin,pluginData)
```

Description

`runTest(plugin,pluginData)` extends the running of a single test suite element. This method allows you to override the method that runs a scalar `Test` element in the `TestSuite` array, including the creation of the `TestCase`, and the `TestMethodSetup` and `TestMethodTeardown` routines. Provided the testing framework completes all fixture setup, it invokes this method one time per `Test` element.

Input Arguments

`plugin`

Plugin object, specified as an instance of the `matlab.unittest.plugins.TestRunnerPlugin` class.

`pluginData`

`Test` element information, specified as an instance of the `matlab.unittest.plugins.plugindata.RunPluginData` class. The testing framework uses this information to describe the test content to the plugin.

Examples

Extend runTest Method

Print the label of the test element at run time.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin
    methods (Access = protected)
        function runTest(plugin, pluginData)
            fprintf('### Running test: %s\n', pluginData.Name)
            runTest@matlab.unittest.plugins.TestRunnerPlugin(...
                plugin, pluginData);
        end
    end
end
```

See Also

`matlab.unittest.TestResult` | `matlab.unittest.TestRunner` | `matlab.unittest.TestSuite` | `matlab.unittest.plugins.plugindata.RunPluginData`

Topics

"Write Plugins to Extend TestRunner"
"Create Custom Plugin"
"Run Tests in Parallel with Custom Plugin"

Introduced in R2014a

runTestClass

Class: `matlab.unittest.plugins.TestRunnerPlugin`
Package: `matlab.unittest.plugins`

Extend running of `Test` elements from same class or function

Syntax

```
runTestClass(plugin,pluginData)
```

Description

`runTestClass(plugin,pluginData)` extends the running of tests that belong to the same test class, function, or script. This method applies to a subset of the full `TestSuite` being run by the `TestRunner`. The testing framework evaluates this method within the scope of the `runTestSuite` method of `TestRunnerPlugin`. It evaluates this method between setting up and tearing down the shared test fixture (`setupSharedTestFixture` and `teardownSharedTestFixture`). Provided the testing framework completes shared test fixture setup, it invokes this method one time per test class.

Input Arguments

`plugin`

Plugin object, specified as an instance of the `matlab.unittest.plugins.TestRunnerPlugin` class.

`pluginData`

Information about the `Test` elements being run, specified as an instance of the `matlab.unittest.plugins.plugindata.RunPluginData` class. The testing framework uses this information to describe the test content to the plugin.

Examples

Extend runTestClass Method

Print the label of the test content at run time.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin
    methods (Access = protected)
        function runTestClass(plugin, pluginData)
            fprintf('### Running test class: %s\n', pluginData.Name)

            runTestClass@matlab.unittest.plugins.TestRunnerPlugin(...
                plugin, pluginData);
        end
    end
end
```

See Also

`matlab.unittest.TestResult` | `matlab.unittest.TestRunner` | `matlab.unittest.TestSuite` | `matlab.unittest.plugins.plugindata.RunPluginData`

Topics

"Write Plugins to Extend TestRunner"
"Create Custom Plugin"
"Run Tests in Parallel with Custom Plugin"

Introduced in R2014a

runTestSuite

Class: `matlab.unittest.plugins.TestRunnerPlugin`
Package: `matlab.unittest.plugins`

Extend running of `TestSuite` array

Syntax

```
runTestSuite(plugin,pluginData)
```

Description

`runTestSuite(plugin,pluginData)` extends the running of the portion of the `TestSuite` array that is passed to the `TestRunner`. The testing framework evaluates this method within the scope of the `runSession` method of `TestRunnerPlugin`.

An example of running different test suite portions is when tests run in parallel (requires Parallel Computing Toolbox). In this case, the testing framework divides the original test suite into separate groups and assigns them to workers on the current parallel pool. The framework evaluates `runTestSuite` as many times as the number of groups into which the entire `TestSuite` array is divided.

Input Arguments

`plugin`

Plugin object, specified as an instance of the `matlab.unittest.plugins.TestRunnerPlugin` class.

`pluginData`

`TestSuite` portion information, specified as an instance of the `matlab.unittest.plugins.plugindata.TestSuiteRunPluginData` class. The testing framework uses this information to describe the test content to the plugin.

Examples

Display Running TestSuite Information

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin
    methods (Access = protected)
        function runTestSuite(plugin, pluginData)

            % Inspect pluginData to get TestSuite size and group
            groupNumber = pluginData.Group;
            totalGroups = pluginData.NumGroups;
            suiteSize = numel(pluginData.TestSuite);
            fprintf('### Running %d tests in group %d out of %d groups\n',...
                suiteSize, groupNumber, totalGroups)

            % Invoke the superclass method
            runTestSuite@matlab.unittest.plugins.TestRunnerPlugin(plugin, pluginData);
        end
    end
end
```

See Also

`matlab.unittest.TestResult` | `matlab.unittest.TestRunner` | `matlab.unittest.TestSuite` | `matlab.unittest.plugins.plugindata.TestSuiteRunPluginData` | `storeIn`

Topics

"Write Plugins to Extend TestRunner"
"Create Custom Plugin"
"Run Tests in Parallel with Custom Plugin"

Introduced in R2014a

runTestMethod

Class: `matlab.unittest.plugins.TestRunnerPlugin`
Package: `matlab.unittest.plugins`

Extend running of single test method

Syntax

```
runTestMethod(plugin,pluginData)
```

Description

`runTestMethod(plugin,pluginData)` extends the running of a single test method. The testing framework evaluates this method within the scope of the `runTest` method of `TestRunnerPlugin`. It evaluates this method between setting up and tearing down the scalar `TestSuite` element (`setupTestMethod` and `teardownTestMethod`).

Input Arguments

`plugin`

Plugin object, specified as an instance of the `matlab.unittest.plugins.TestRunnerPlugin` class.

`pluginData`

Test method information, specified as an instance of the `matlab.unittest.plugins.plugindata.RunPluginData` class. The testing framework uses this information to describe the test content to the plugin.

Examples

Extend runTestMethod Method

Print the time taken to evaluate the test method.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin
    methods (Access = protected)
        function runTestMethod(plugin, pluginData)
            tic

            runTestMethod@matlab.unittest.plugins.TestRunnerPlugin(...
                plugin, pluginData);

            fprintf('### %s ran in %f seconds excluding fixture time.',...
                pluginData.Name, toc)
        end
    end
end
```

See Also

`matlab.unittest.TestResult` | `matlab.unittest.TestRunner` | `matlab.unittest.plugins.plugindata.RunPluginData` | `runTest` | `runTestClass`

Topics

"Write Plugins to Extend TestRunner"
"Create Custom Plugin"
"Run Tests in Parallel with Custom Plugin"

Introduced in R2014a

setupSharedTestFixture

Class: `matlab.unittest.plugins.TestRunnerPlugin`
Package: `matlab.unittest.plugins`

Extend setting up shared test fixture

Syntax

```
setupSharedTestFixture(plugin,pluginData)
```

Description

`setupSharedTestFixture(plugin,pluginData)` extends the setting up of a shared test fixture. This method defines how the `TestRunner` performs shared fixture setup. The testing framework evaluates this method one time for each shared test fixture, within the scope of the `runTestSuite` method of `TestRunnerPlugin`.

Input Arguments

`plugin`

Plugin object, specified as an instance of the `matlab.unittest.plugins.TestRunnerPlugin` class.

`pluginData`

Shared test fixture setup information, specified as an instance of the `matlab.unittest.plugins.plugindata.SharedTestFixturePluginData` class. The testing framework uses this information to describe the test content to the plugin.

Examples

Implement setupSharedTestFixture Method

Display the shared test fixture name at setup time.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin

    methods (Access = protected)
        function setupSharedTestFixture(plugin, pluginData)
            fprintf('### Setting up: %s\n', pluginData.Name)
            setupSharedTestFixture@matlab.unittest.plugins.TestRunnerPlugin...
                (plugin, pluginData);
        end
    end
end
```

See Also

`createSharedTestFixture` | `matlab.unittest.TestRunner` | `matlab.unittest.plugins.plugindata.SharedTestFixturePluginData`

Topics

"Write Plugins to Extend TestRunner"
"Create Custom Plugin"
"Run Tests in Parallel with Custom Plugin"

Introduced in R2014a

setupTestClass

Class: `matlab.unittest.plugins.TestRunnerPlugin`
Package: `matlab.unittest.plugins`

Extend setting up test class

Syntax

```
setupTestClass(plugin,pluginData)
```

Description

`setupTestClass(plugin,pluginData)` extends the setting up of a test class. This method defines how the `TestRunner` performs test class setup. The testing framework evaluates this method within the scope of the `runTestClass` method of `TestRunnerPlugin`. If the test class contains properties with the `ClassSetupParameter` attribute, the testing framework evaluates the `setupTestClass` method as many times as the class setup parameterization dictates.

Input Arguments

`plugin`

Plugin object, specified as an instance of the `matlab.unittest.plugins.TestRunnerPlugin` class.

`pluginData`

Test class setup information, specified as an instance of the `matlab.unittest.plugins.plugindata.ImplicitFixturePluginData`. The testing framework uses this information to describe the test content to the plugin.

Examples

Implement setupTestClass Method

Display the test class name at setup time.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin

    methods (Access = protected)
        function setupTestClass(plugin, pluginData)
            fprintf('### Setting up: %s\n', pluginData.Name)
            setupTestClass@matlab.unittest.plugins.TestRunnerPlugin...
                (plugin, pluginData);
        end
    end
end
```

See Also

`createTestClassInstance` | `matlab.unittest.TestRunner` | `matlab.unittest.plugins.plugindata.ImplicitFixturePluginData`

Topics

"Write Plugins to Extend TestRunner"
"Create Custom Plugin"
"Run Tests in Parallel with Custom Plugin"

Introduced in R2014a

setupTestMethod

Class: `matlab.unittest.plugins.TestRunnerPlugin`
Package: `matlab.unittest.plugins`

Extend setting up test method

Syntax

```
setupTestMethod(plugin,pluginData)
```

Description

`setupTestMethod(plugin,pluginData)` extends the setting up of a test method. This method defines how the `TestRunner` performs test method setup for the single test suite element. The testing framework evaluates this method within the scope of the `runTest` method of `TestRunnerPlugin`.

Input Arguments

`plugin`

Plugin object, specified as an instance of the `matlab.unittest.plugins.TestRunnerPlugin` class.

`pluginData`

Test method setup information, specified as an instance of the `matlab.unittest.plugins.plugindata.ImplicitFixturePluginData` class. The testing framework uses this information to describe the test content to the plugin.

Examples

Implement setupTestMethod

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin

    methods (Access = protected)
        function setupTestMethod(plugin, pluginData)
            fprintf('### Setting up: %s\n', pluginData.Name)
            setupTestMethod@matlab.unittest.plugins.TestRunnerPlugin...
                (plugin, pluginData);
        end
    end
end
```

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.plugins.plugindata.ImplicitFixturePluginData`

Topics

"Write Plugins to Extend TestRunner"
"Create Custom Plugin"
"Run Tests in Parallel with Custom Plugin"

Introduced in R2014a

teardownSharedTestFixture

Class: `matlab.unittest.plugins.TestRunnerPlugin`
Package: `matlab.unittest.plugins`

Extend tearing down shared test fixture

Syntax

```
teardownSharedTestFixture(plugin,pluginData)
```

Description

`teardownSharedTestFixture(plugin,pluginData)` extends the tearing down of a shared test fixture. This method defines how the `TestRunner` performs shared fixture teardown. The testing framework evaluates this method one time for each shared test fixture, within the scope of the `runTestSuite` method of `TestRunnerPlugin`.

Input Arguments

`plugin`

Plugin object, specified as an instance of the `matlab.unittest.plugins.TestRunnerPlugin` class.

`pluginData`

Shared test fixture teardown information, specified as an instance of the `matlab.unittest.plugins.plugindata.SharedTestFixturePluginData` class. The testing framework uses this information to describe the test content to the plugin.

Examples

Implement teardownSharedTestFixture Method

Display the shared test fixture name at teardown time.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin

    methods (Access = protected)
        function teardownSharedTestFixture(plugin, pluginData)
            fprintf('### Tearing down: %s\n', pluginData.Name)
            teardownSharedTestFixture@matlab.unittest.plugins.TestRunnerPlugin...
                (plugin, pluginData);
        end
    end
end
```

See Also

`createSharedTestFixture` | `matlab.unittest.TestRunner` | `matlab.unittest.plugins.plugindata.SharedTestFixturePluginData`

Topics

"Write Plugins to Extend TestRunner"
"Create Custom Plugin"
"Run Tests in Parallel with Custom Plugin"

Introduced in R2014a

teardownTestClass

Class: `matlab.unittest.plugins.TestRunnerPlugin`
Package: `matlab.unittest.plugins`

Extend tearing down test class

Syntax

```
teardownTestClass(plugin,pluginData)
```

Description

`teardownTestClass(plugin,pluginData)` extends the tearing down of a test class. This method defines how the `TestRunner` performs test class teardown. The testing framework evaluates this method within the scope of the `runTestClass` method of `TestRunnerPlugin`. If the test class contains properties with the `ClassSetupParameter` attribute, the testing framework evaluates the `teardownTestClass` method as many times as the class setup parameterization dictates.

Input Arguments

`plugin`

Plugin object, specified as an instance of the `matlab.unittest.plugins.TestRunnerPlugin` class.

`pluginData`

Test class teardown information, specified as an instance of the `matlab.unittest.plugins.plugindata.ImplicitFixturePluginData` class. The testing framework uses this information to describe the test content to the plugin.

Examples

Implement teardownTestClass Method

Display the test class name at teardown time.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin

    methods (Access = protected)
        function teardownTestClass(plugin, pluginData)
            fprintf('### Tearing down: %s\n', pluginData.Name)
            teardownTestClass@matlab.unittest.plugins.TestRunnerPlugin...
                (plugin, pluginData);
        end
    end
end
```

See Also

`createTestClassInstance` | `matlab.unittest.TestRunner` | `matlab.unittest.plugins.plugindata.ImplicitFixturePluginData`

Topics

"Write Plugins to Extend TestRunner"
"Create Custom Plugin"
"Run Tests in Parallel with Custom Plugin"

Introduced in R2014a

teardownTestMethod

Class: `matlab.unittest.plugins.TestRunnerPlugin`
Package: `matlab.unittest.plugins`

Extend tearing down test method

Syntax

```
teardownTestMethod(plugin,pluginData)
```

Description

`teardownTestMethod(plugin,pluginData)` extends the tearing down of a test method. This method defines how the `TestRunner` performs test method teardown for a single test suite element. The testing framework evaluates this method within the scope of the `runTest` method of `TestRunnerPlugin`.

Input Arguments

`plugin`

Plugin object, specified as an instance of the `matlab.unittest.plugins.TestRunnerPlugin` class.

`pluginData`

Test method teardown information, specified as an instance of the `matlab.unittest.plugins.plugindata.ImplicitFixturePluginData` class. The testing framework uses this information to describe the test content to the plugin.

Examples

Implement teardownTestMethod Method

Display the test method name at teardown time.

```
classdef ExamplePlugin < matlab.unittest.plugins.TestRunnerPlugin
    methods (Access = protected)
        function teardownTestMethod(plugin, pluginData)
            fprintf('### Tearing down: %s\n', pluginData.Name)
            teardownTestMethod@matlab.unittest.plugins.TestRunnerPlugin...
                (plugin, pluginData);
        end
    end
end
```

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.plugins.plugindata.ImplicitFixturePluginData` | `setupTestMethod`

Topics

"Write Plugins to Extend TestRunner"
"Create Custom Plugin"
"Run Tests in Parallel with Custom Plugin"

Introduced in R2014a

matlab.unittest.plugins.TestRunProgressPlugin class

Package: `matlab.unittest.plugins`

Plugin that reports test run progress

Description

The `TestRunProgressPlugin` class creates a plugin that reports on test run progress.

Construction

`matlab.unittest.plugins.TestRunProgressPlugin.withVerbosity(v)` constructs a `TestRunProgressPlugin` for the specified verbosity.

`matlab.unittest.plugins.TestRunProgressPlugin.withVerbosity(v, stream)` redirects the text output to the output stream.

Input Arguments

v — Verbosity level
`0` | `1` | `2` | `3` | `4` | `matlab.unittest.Verbosity` enumeration | enumeration name as string or char vector

Verbosity level, specified as an integer value between 0 and 4, a `matlab.unittest.Verbosity` enumeration object, or a string scalar or character vector corresponding to one of the predefined enumeration member names. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration.

Numeric Representation	Enumeration Member Name	Verbosity Description
`0`	None	No information

Numeric Representation	Enumeration Member Name	Verbosity Description
1	`Terse`	Minimal information
2	`Concise`	Moderate amount of information
3	`Detailed`	Some supplemental information
4	`Verbose`	Lots of supplemental information

`stream` — Location where plugin directs text output
`ToStandardOutput` instance (default) | `OutputStream` instance

Location where the plugin directs text output, specified as an `OutputStream` instance. By default, the plugin uses the `OutputStream` subclass `ToStandardOutput` as the stream.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Create Test Run Progress Plugin

Create a function-based test called `cylinderPlotTest` in a file in your working folder.

```
function tests = cylinderPlotTest
tests = functiontests(localfunctions);
end

function setupOnce(testCase)
testCase.TestData.Figure = figure;
addTeardown(testCase,@close,testCase.TestData.Figure)
end

function setup(testCase)
testCase.TestData.Axes = axes('Parent',testCase.TestData.Figure);
addTeardown(testCase,@clf,testCase.TestData.Figure)
cylinder(testCase.TestData.Axes,10)
end
```

```
function testXLim(testCase)
xlim = testCase.TestData.Axes.XLim;
verifyLessThanOrEqual(testCase,xlim(1),-10,'Minimum x-limit too large')
verifyGreaterThanOrEqual(testCase,xlim(2),10,'Maximum x-limit too small')
end

function zdataTest(testCase)
s = findobj(testCase.TestData.Axes,'Type','surface');
verifyEqual(testCase,min(s.ZData(:)),0,'Min cylinder value is incorrect')
verifyEqual(testCase,max(s.ZData(:)),1,'Max cylinder value is incorrect')
end
```

At the command prompt, run the test.

```
results = run(cylinderPlotTest);
```

```
Running cylinderPlotTest
..
Done cylinderPlotTest
__________
```

By default, the test runner uses verbosity level 2.

Create a test runner to report the diagnostics at level 1, and rerun the test.

```
import matlab.unittest.TestRunner
import matlab.unittest.plugins.TestRunProgressPlugin

runner = TestRunner.withNoPlugins;
p = TestRunProgressPlugin.withVerbosity(1);
runner.addPlugin(p);

results = runner.run(cylinderPlotTest);
```

```
..
```

Create a test runner to report the diagnostics at level 4, and rerun the test.

```
runner = TestRunner.withNoPlugins;
p = TestRunProgressPlugin.withVerbosity(4);
runner.addPlugin(p);

results = runner.run(cylinderPlotTest);
```

```
 Running cylinderPlotTest
  Setting up cylinderPlotTest
    Evaluating TestClassSetup: setupOnce
  Done setting up cylinderPlotTest in 0.067649 seconds
   Running cylinderPlotTest/testXLim
```

```
    Evaluating TestMethodSetup: setup
    Evaluating Test: testXLim
    Evaluating TestMethodTeardown: teardown
    Evaluating addTeardown function: clf
   Done cylinderPlotTest/testXLim in 0.053834 seconds
   Running cylinderPlotTest/zdataTest
    Evaluating TestMethodSetup: setup
    Evaluating Test: zdataTest
    Evaluating TestMethodTeardown: teardown
    Evaluating addTeardown function: clf
   Done cylinderPlotTest/zdataTest in 0.037715 seconds
  Tearing down cylinderPlotTest
    Evaluating TestClassTeardown: teardownOnce
    Evaluating addTeardown function: close
  Done tearing down cylinderPlotTest in 0.022783 seconds
 Done cylinderPlotTest in 0.18198 seconds
__________
```

Configure Progress Message Output

Create a class named `ExampleProgressTest` in a file in your current working folder.

```
classdef ExampleProgressTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase)  % Test fails
            testCase.verifyEqual(5,4)
        end
        function testTwo(testCase)  % Test passes
            testCase.verifyEqual(5,5)
        end
    end
end
```

At the command prompt, create the test suite and a runner at verbosity level 3, and then run the test.

```
import matlab.unittest.TestSuite
import matlab.unittest.TestRunner
import matlab.unittest.plugins.TestRunProgressPlugin

suite = TestSuite.fromClass(?ExampleProgressTest);

runner = TestRunner.withNoPlugins;
```

```
p = TestRunProgressPlugin.withVerbosity(3);
runner.addPlugin(p);
results = runner.run(suite);
```

```
 Running ExampleProgressTest
  Setting up ExampleProgressTest
  Done setting up ExampleProgressTest in 0 seconds
   Running ExampleProgressTest/testOne
   Done ExampleProgressTest/testOne in 0.0049988 seconds
   Running ExampleProgressTest/testTwo
   Done ExampleProgressTest/testTwo in 0.0044541 seconds
  Tearing down ExampleProgressTest
  Done tearing down ExampleProgressTest in 0 seconds
 Done ExampleProgressTest in 0.0094529 seconds
__________
```

Create a new plugin to direct the output to a file named `myOutput.log`, and rerun the tests.

```
import matlab.unittest.plugins.ToFile
outFile = 'myOutput.log';

runner = TestRunner.withNoPlugins;
p = TestRunProgressPlugin.withVerbosity(3,ToFile(outFile));
runner.addPlugin(p);

results = runner.run(suite);
```

Observe the contents of the file created by the plugin.

```
disp(fileread(outFile))
```

```
 Running ExampleProgressTest
  Setting up ExampleProgressTest
  Done setting up ExampleProgressTest in 0 seconds
   Running ExampleProgressTest/testOne
   Done ExampleProgressTest/testOne in 0.0050172 seconds
   Running ExampleProgressTest/testTwo
   Done ExampleProgressTest/testTwo in 0.0049449 seconds
  Tearing down ExampleProgressTest
  Done tearing down ExampleProgressTest in 0 seconds
```

```
 Done ExampleProgressTest in 0.009962 seconds
__________
```

See Also

matlab.unittest.TestRunner | matlab.unittest.Verbosity | matlab.unittest.plugins.OutputStream | matlab.unittest.plugins.TestRunnerPlugin | matlab.unittest.plugins.ToStandardOutput

Introduced in R2014b

matlab.unittest.plugins.XMLPlugin class

Package: `matlab.unittest.plugins`

Plugin that writes test results in XML format

Description

The `XMLPlugin` class creates a plugin that writes test results to a file in XML format.

Construction

Instantiate an `XMLPlugin` using the `matlab.unittest.plugins.XMLPlugin.producingJUnitFormat` method.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Output Results to JUnit-style XML File

Create `ExampleTest.m` containing the following test class.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase)  % Test fails
            testCase.verifyEqual(5,4,'Testing 5==4')
        end
        function testTwo(testCase)  % Test passes
            testCase.verifyEqual(5,5,'Testing 5==5')
        end
        function testThree(testCase) % Test fails
```

```
            testCase.assumeTrue(false)
        end
    end
end
```

Create a test suite from the `ExampleTest` class. Create a silent test runner.

```
import matlab.unittest.TestRunner
import matlab.unittest.TestSuite
import matlab.unittest.plugins.XMLPlugin

suite = TestSuite.fromClass(?ExampleTest);
runner = TestRunner.withNoPlugins;
```

Create an `XMLPlugin` that writes test results to the file `myTestResults.xml`.

```
xmlFile = 'myTestResults.xml';
p = XMLPlugin.producingJUnitFormat(xmlFile);
```

Add the plugin to the test runner and run the suite.

```
runner.addPlugin(p)
results = runner.run(suite);
table(results)
```

```
ans =

            Name              Passed    Failed    Incomplete    Duration       Details
    ______________________    ______    ______    __________    ________    ____________

    'ExampleTest/testOne'      false     true      false          0.078297    [1x1 struct]
    'ExampleTest/testTwo'      true      false     false         0.0038761    [1x1 struct]
    'ExampleTest/testThree'    false     false     true           0.049763    [1x1 struct]
```

View the contents in the file created by the plugin.

```
disp(fileread(xmlFile))
```

```
<?xml version="1.0" encoding="utf-8"?>
<testsuites>
   <testsuite errors="0" failures="1" name="" skipped="1" tests="3" time="0.0493">
      <testcase classname="ExampleTest" name="testOne" time="0.030643">
         <failure type="VerificationFailure">==================================================================
Verification failed in ExampleTest/testOne.

    ----------------
    Test Diagnostic:
    ----------------
    Testing 5==4
```

```
    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyEqual failed.
    --&gt; The values are not equal using "isequaln".
    --&gt; Failure table:
                Actual    Expected    Error    RelativeError
                ______    ________    _____    _____________

                5         4           1        0.25

    Actual double:
              5
    Expected double:
              4

    ------------------
    Stack Information:
    ------------------
    In C:\work\ExampleTest.m (ExampleTest.testOne) at 4
================================================================================
</failure>
      </testcase>
      <testcase classname="ExampleTest" name="testTwo" time="0.004332"/>
      <testcase classname="ExampleTest" name="testThree" time="0.014325">
         <skipped>==============================================================================
ExampleTest/testThree was filtered.
================================================================================
</skipped>
      </testcase>
   </testsuite>
</testsuites>
```

Tips

- If your test file is a script-based or function-based test, the value of the `classname` attribute of the `<testcase>` element is the test file name.

See Also

External Websites

JUnit

Introduced in R2015b

matlab.unittest.plugins.XMLPlugin.producingJUnitFormat

Class: `matlab.unittest.plugins.XMLPlugin`
Package: `matlab.unittest.plugins`

Construct plugin to write test results in XML format

Syntax

```
xmlPlugin = matlab.unittest.plugins.XMLPlugin.producingJUnitFormat(
xmlFile)
xmlPlugin = matlab.unittest.plugins.XMLPlugin.producingJUnitFormat(
xmlFile,'OutputDetail',OutputDetail)
```

Description

`xmlPlugin = matlab.unittest.plugins.XMLPlugin.producingJUnitFormat(xmlFile)` returns a plugin that produces JUnit-style XML output to the specified XML file. Using this plugin, you can integrate MATLAB Unit Test results into third-party systems that recognize JUnit-style XML. For example, you can integrate test results with continuous integration systems like Jenkins, TeamCity, or Microsoft Team Foundation Server.

The XML file is overwritten every time you run the test suite with this plugin.

`xmlPlugin = matlab.unittest.plugins.XMLPlugin.producingJUnitFormat(xmlFile,'OutputDetail',OutputDetail)` returns a plugin that displays failing events with the amount of detail specified by `OutputDetail`.

Input Arguments

`xmlFile` — Name of test-result file
character vector | string scalar

Name of the test-result file, specified as a character vector or string scalar. If `xmlFile` exists, the test framework overwrites the file.

Example: `XMLPlugin.producingJUnitFormat('myTestResults.xml')`

Data Types: `char` | `string`

`OutputDetail` — Detail level for recorded events
3 (default) | 0 | 1 | 2 | 4 | `matlab.unittest.Verbosity` enumeration member | string or character vector corresponding to the name of a `matlab.unittest.Verbosity` enumeration member.

Detail level for recorded events, specified as an integer value from 0 through 4, as a `matlab.unittest.Verbosity` enumeration member, or as a string or character vector corresponding to the name of a `matlab.unittest.Verbosity` enumeration member. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration.

The plugin records failing events with the amount of detail specified by `OutputDetail`. By default, the plugin records events at the `matlab.unittest.Verbosity.Detailed` level (level 3).

Numeric Representation	Enumeration Member Name	Verbosity Description
`0`	`None`	No information
`1`	`Terse`	Minimal information
`2`	`Concise`	Moderate amount of information
`3`	`Detailed`	Some supplemental information
`4`	`Verbose`	Lots of supplemental information

Output Arguments

`xmlPlugin` — Plugin
`matlab.unittest.plugins.XMLPlugin`

Plugin that writes test results to a file in JUnit-style XML format, returned as a `matlab.unittest.plugins.XMLPlugin` object.

Attributes

`Static`	`true`

To learn about attributes of methods, see Method Attributes.

Examples

Create Plugin to Create XML Output File

```
import matlab.unittest.plugins.XMLPlugin;
import matlab.unittest.Verbosity;

% Create a XML plugin that sends XML Output to a file
plugin = XMLPlugin.producingJUnitFormat('MyXMLFile.xml');

% Create a XML plugin that produces a concise amount of output detail
plugin = XMLPlugin.producingJUnitFormat('MyXMLFile.xml','OutputDetail',Verbosity.Concise);
```

For more examples, see `matlab.unittest.plugins.XMLPlugin`.

See Also

`matlab.unittest.Verbosity` | `matlab.unittest.plugins.XMLPlugin`

Introduced in R2015b

matlab.unittest.plugins.diagnosticrecord Package

Summary of classes associated with MATLAB Plugin Diagnostic Records

Description

The `matlab.unittest.plugins.diagnosticrecord` package consists of the following MATLAB® classes.

Classes

matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord	Diagnostic information
matlab.unittest.plugins.diagnosticrecord.ExceptionDiagnosticRecord	Diagnostic information about errors
matlab.unittest.plugins.diagnosticrecord.LoggedDiagnosticRecord	Diagnostic information about logged events
matlab.unittest.plugins.diagnosticrecord.QualificationDiagnosticRecord	Diagnostic information about qualification events

See Also

`DiagnosticsRecordingPlugin` | `matlab.unittest.plugins`

Introduced in R2016a

matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord class

Package: `matlab.unittest.plugins.diagnosticrecord`

Diagnostic information

Description

The `DiagnosticRecord` class defines the diagnostic information that the `DiagnosticsRecordingPlugin` includes on the `TestResult`. The `DiagnosticsRecordingPlugin` creates this class, so there is no need for test authors to construct the class directly.

Properties

`Event` — Name of recorded event

character vector

Name of recorded event, returned as a character vector. This property corresponds to the event on the `TestCase` instance.

Example: `'VerificationPassed'`

Example: `'ExceptionThrown'`

Example: `'DiagnosticLogged'`

Data Types: `char`

`EventLocation` — Location of event

character vector

Location of event, returned as a character vector. This property corresponds to the label of the test content.

Example: `'myTestClass'`

Example: `'myTestClass/testMethod1'`

Example: `'myTestClass[classSetupParam=value1]/testMethod1'`

Example: `'mySharedTestFixture'`

Data Types: `char`

`EventScope` — Scope where event originated
instance of `matlab.unittest.Scope`

Scope where event originated, returned as an instance of `matlab.unittest.Scope`. This property corresponds to the label of the test content.

Example: `matlab.unittest.Scope.TestMethod`

Example: `matlab.unittest.Scope.TestClass`

Example: `matlab.unittest.Scope.SharedTestFixture`

`Stack` — Stack trace to diagnostic
structure

Stack trace to the location of the diagnostic event, returned as a structure containing the fields `file`, `name`, and `line`.

Data Types: `struct`

`Report` — Diagnostic information
character vector

All diagnostic information, returned as a character vector. The report provides an overall summary of the event.

Data Types: `char`

Methods

selectFailed	Return diagnostic records for failed events
selectIncomplete	Return diagnostic records for incomplete events
selectLogged	Return diagnostic records for logged events
selectPassed	Return diagnostic records for passed events

See Also

`matlab.unittest.plugins.DiagnosticsRecordingPlugin` | `matlab.unittest.plugins.diagnosticrecord`

Introduced in R2016a

matlab.unittest.plugins.diagnosticrecord.ExceptionDiagnosticRecord class

Package: `matlab.unittest.plugins.diagnosticrecord`
Superclasses:
`matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord`

Diagnostic information about errors

Description

The `ExceptionDiagnosticRecord` class defines the diagnostic information about errors that the `DiagnosticsRecordingPlugin` includes on the `TestResult`. The `DiagnosticsRecordingPlugin` creates this class, so there is no need for test authors to construct the class directly.

Properties

`Exception` — Error information
`MException` object

Error information, returned as an `MException` object.

`AdditionalDiagnosticResults` — Results of additional diagnostics specified in the test content
array of `DiagnosticResult` instances

Results of additional diagnostics specified in the test content, represented as an array of `DiagnosticResult` instances. For example, `AdditionalDiagnosticResults` includes results from diagnostics added using the `matlab.unittest.TestCase.onFailure` method.

These properties are inherited from `matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord`:

`Event` — Name of recorded event
character vector

Name of recorded event, returned as a character vector. This property corresponds to the event on the `TestCase` instance.

Example: `'VerificationPassed'`

Example: `'ExceptionThrown'`

Example: `'DiagnosticLogged'`

Data Types: `char`

`EventLocation` — Location of event
character vector

Location of event, returned as a character vector. This property corresponds to the label of the test content.

Example: `'myTestClass'`

Example: `'myTestClass/testMethod1'`

Example: `'myTestClass[classSetupParam=value1]/testMethod1'`

Example: `'mySharedTestFixture'`

Data Types: `char`

`EventScope` — Scope where event originated
instance of `matlab.unittest.Scope`

Scope where event originated, returned as an instance of `matlab.unittest.Scope`. This property corresponds to the label of the test content.

Example: `matlab.unittest.Scope.TestMethod`

Example: `matlab.unittest.Scope.TestClass`

Example: `matlab.unittest.Scope.SharedTestFixture`

`Stack` — Stack trace to diagnostic
structure

Stack trace to the location of the diagnostic event, returned as a structure containing the fields `file`, `name`, and `line`.

Data Types: `struct`

`Report` — Diagnostic information
character vector

All diagnostic information, returned as a character vector. The report provides an overall summary of the event.

Data Types: `char`

Methods

Inherited Methods

selectFailed	Return diagnostic records for failed events
selectIncomplete	Return diagnostic records for incomplete events
selectLogged	Return diagnostic records for logged events
selectPassed	Return diagnostic records for passed events

See Also

`matlab.unittest.plugins.DiagnosticsRecordingPlugin` | `matlab.unittest.plugins.diagnosticrecord`

Introduced in R2016a

matlab.unittest.plugins.diagnosticrecord.LoggedDiagnosticRecord class

Package: `matlab.unittest.plugins.diagnosticrecord`
Superclasses:
`matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord`

Diagnostic information about logged events

Description

The `LoggedDiagnosticRecord` class defines the diagnostic information about logged events that the `DiagnosticsRecordingPlugin` includes on the `TestResult`. The `DiagnosticsRecordingPlugin` creates this class, so there is no need for test authors to construct the class directly.

Properties

`LoggedDiagnosticResult` — Logged diagnostic result
cell array of character vectors

Logged diagnostic result, returned as a cell array of character vectors. A single `log` call can result in multiple logged diagnostics.

Data Types: `char`

`Timestamp` — Time the `log` method was called
date and time

The time the `log` method was called, returned as a date and time.

Data Types: `datetime`

`Verbosity` — Verbosity level of the logged message
`matlab.unittest.Verbosity` enumeration

Verbosity level of the logged message, returned as a `matlab.unittest.Verbosity` enumeration object.

Example: `Terse`

These properties are inherited from `matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord`:

`Event` — Name of recorded event
character vector

Name of recorded event, returned as a character vector. This property corresponds to the event on the `TestCase` instance.

Example: `'VerificationPassed'`

Example: `'ExceptionThrown'`

Example: `'DiagnosticLogged'`

Data Types: `char`

`EventLocation` — Location of event
character vector

Location of event, returned as a character vector. This property corresponds to the label of the test content.

Example: `'myTestClass'`

Example: `'myTestClass/testMethod1'`

Example: `'myTestClass[classSetupParam=value1]/testMethod1'`

Example: `'mySharedTestFixture'`

Data Types: `char`

`EventScope` — Scope where event originated
instance of `matlab.unittest.Scope`

Scope where event originated, returned as an instance of `matlab.unittest.Scope`. This property corresponds to the label of the test content.

Example: `matlab.unittest.Scope.TestMethod`

Example: `matlab.unittest.Scope.TestClass`

Example: `matlab.unittest.Scope.SharedTestFixture`

`Stack` — Stack trace to diagnostic
structure

Stack trace to the location of the diagnostic event, returned as a structure containing the fields `file`, `name`, and `line`.

Data Types: `struct`

`Report` — Diagnostic information
character vector

All diagnostic information, returned as a character vector. The report provides an overall summary of the event.

Data Types: `char`

Methods

Inherited Methods

selectFailed	Return diagnostic records for failed events
selectIncomplete	Return diagnostic records for incomplete events
selectLogged	Return diagnostic records for logged events
selectPassed	Return diagnostic records for passed events

See Also

`matlab.unittest.plugins.DiagnosticsRecordingPlugin` | `matlab.unittest.plugins.diagnosticrecord`

Introduced in R2016a

matlab.unittest.plugins.diagnosticrecord.QualificationDiagnosticRecord class

Package: `matlab.unittest.plugins.diagnosticrecord`
Superclasses:
`matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord`

Diagnostic information about qualification events

Description

The `QualificationDiagnosticRecord` class defines the diagnostic information about qualification events that the `DiagnosticsRecordingPlugin` includes on the `TestResult`. The `DiagnosticsRecordingPlugin` creates this class, so there is no need for test authors to construct the class directly.

Properties

`TestDiagnosticResults` — Test diagnostic results
cell array of character vectors

Test diagnostic results, returned as a cell array of character vectors. A single qualification can result in multiple test diagnostics.

Data Types: `char`

`FrameworkDiagnosticResults` — Framework diagnostic results
cell array of character vectors

Framework diagnostic results, returned as a cell array of character vectors. A single qualification can result in multiple framework diagnostics.

Example: `{'verifyClass failed.…'}`

Data Types: `char`

`AdditionalDiagnosticResults` — Results of additional diagnostics specified in the test content
array of `DiagnosticResult` instances

Results of additional diagnostics specified in the test content, represented as an array of `DiagnosticResult` instances. For example, `AdditionalDiagnosticResults` includes results from diagnostics added using the `matlab.unittest.TestCase.onFailure` method.

These properties are inherited from `matlab.unittest.plugins.diagnosticrecord.DiagnosticRecord`:

`Event` — Name of recorded event
character vector

Name of recorded event, returned as a character vector. This property corresponds to the event on the `TestCase` instance.

Example: `'VerificationPassed'`

Example: `'ExceptionThrown'`

Example: `'DiagnosticLogged'`

Data Types: `char`

`EventLocation` — Location of event
character vector

Location of event, returned as a character vector. This property corresponds to the label of the test content.

Example: `'myTestClass'`

Example: `'myTestClass/testMethod1'`

Example: `'myTestClass[classSetupParam=value1]/testMethod1'`

Example: `'mySharedTestFixture'`

Data Types: `char`

`EventScope` — Scope where event originated
instance of `matlab.unittest.Scope`

Scope where event originated, returned as an instance of `matlab.unittest.Scope`. This property corresponds to the label of the test content.

Example: `matlab.unittest.Scope.TestMethod`

Example: `matlab.unittest.Scope.TestClass`

Example: `matlab.unittest.Scope.SharedTestFixture`

`Stack` — Stack trace to diagnostic
structure

Stack trace to the location of the diagnostic event, returned as a structure containing the fields `file`, `name`, and `line`.

Data Types: `struct`

`Report` — Diagnostic information
character vector

All diagnostic information, returned as a character vector. The report provides an overall summary of the event.

Data Types: `char`

Methods

Inherited Methods

selectFailed	Return diagnostic records for failed events
selectIncomplete	Return diagnostic records for incomplete events
selectLogged	Return diagnostic records for logged events
selectPassed	Return diagnostic records for passed events

See Also

`matlab.unittest.plugins.DiagnosticsRecordingPlugin` | `matlab.unittest.plugins.diagnosticrecord`

Introduced in R2016a

matlab.unittest.plugins.plugindata Package

Summary of classes in MATLAB Plugin Data Interface

Description

The `plugindata` classes store information about test content for use by the plugins and plugin methods. The `TestRunner` passes instances of these classes to various plugin methods. The `matlab.unittest.plugins.plugindata` package consists of the following MATLAB plugin data classes.

Classes

matlab.unittest.plugins.plugindata.CommunicationBuffer	Buffer for parallelizable plugins
matlab.unittest.plugins.plugindata.FinalizedResultPluginData	Plugin data containing finalized test result information
matlab.unittest.plugins.plugindata.FinalizedSuitePluginData	Plugin data containing finalized TestSuite information
matlab.unittest.plugins.plugindata.ImplicitFixturePluginData	Plugin data containing test setup and teardown information
matlab.unittest.plugins.plugindata.PluginData	Data object passed to TestRunnerPlugin methods
matlab.unittest.plugins.plugindata.QualificationContext	Context for qualifying plugins
matlab.unittest.plugins.plugindata.RunPluginData	Plugin data describing Test elements being run

matlab.unittest.plugins.plugindata.SharedTestFixturePluginData	Plugin data containing shared test fixture information
matlab.unittest.plugins.plugindata.TestContentCreationPluginData	Plugin data containing test content creation information
matlab.unittest.plugins.plugindata.TestSuiteRunPluginData	Plugin data describing TestSuite portion being run

See Also

`matlab.unittest.plugins` | `matlab.unittest.plugins.TestRunnerPlugin`

Introduced in R2014a

matlab.unittest.plugins.plugindata.CommunicationBuffer class

Package: `matlab.unittest.plugins.plugindata`

Buffer for parallelizable plugins

Description

The `CommunicationBuffer` class enables data storage and retrieval by the methods of plugins that support running tests in parallel. In particular, `CommunicationBuffer` allows you to store data within the `runTestSuite` method and retrieve data within the `reportFinalizedSuite` method of `TestRunnerPlugin`. The `TestRunner` instantiates this class, so you are not required to create an object of the class directly.

See Also

`matlab.unittest.plugins.Parallelizable` | `matlab.unittest.plugins.TestRunnerPlugin`

Topics

"Run Tests in Parallel with Custom Plugin"

Introduced in R2019b

matlab.unittest.plugins.plugindata.FinalizedResultPluginData class

Package: `matlab.unittest.plugins.plugindata`

Plugin data containing finalized test result information

Description

The `FinalizedResultPluginData` class defines the data passed by the `TestRunner` to plugin methods related to finalized test results. The `TestRunner` instantiates this class, so you are not required to create an object of the class directly.

Properties

`Index` — Location of finalized `Test` element
integer scalar

Location of the finalized `Test` element relative to the entire test suite, specified as an integer scalar.

`TestSuite` — Specification of finalized `Test` element
`matlab.unittest.TestSuite` scalar

Specification of the finalized `Test` element, specified as a `matlab.unittest.TestSuite` scalar.

`TestResult` — Result from finalized `Test` element
`matlab.unittest.TestResult` scalar

Result from the finalized `Test` element, specified as a `matlab.unittest.TestResult` scalar.

See Also

`matlab.unittest.plugins.TestRunnerPlugin` | `reportFinalizedResult`

Introduced in R2015b

matlab.unittest.plugins.plugindata.FinalizedSuitePluginData class

Package: `matlab.unittest.plugins.plugindata`
Superclasses: `matlab.unittest.plugins.plugindata.PluginData`

Plugin data containing finalized `TestSuite` information

Description

The `FinalizedSuitePluginData` class defines the data passed by the `TestRunner` to plugin methods related to finalized test suite portions. The `TestRunner` instantiates this class, so you are not required to create an object of the class directly.

The `matlab.unittest.plugins.plugindata.FinalizedSuitePluginData` class is a `handle` class.

Properties

`NumGroups` — Number of groups for entire test suite
positive integer scalar

Number of groups into which the entire test suite is divided, specified as a positive integer scalar. `NumGroups` is equal to `1` when tests run in serial mode.

`Group` — Identifier of finalized test suite portion
positive integer scalar

Identifier of the finalized test suite portion, specified as a positive integer scalar between `1` and `NumGroups`.

`TestSuite` — Specification of finalized `Test` elements
`matlab.unittest.TestSuite` array

Specification of the finalized `Test` elements in the current group, specified as a `matlab.unittest.TestSuite` array.

`TestResult` — Results from finalized test suite portion
`matlab.unittest.TestResult` array

Results from the finalized test suite portion, specified as a `matlab.unittest.TestResult` array.

`SuiteIndices` — Locations of finalized `Test` elements
numeric array

Locations of the finalized `Test` elements relative to the entire test suite passed to the `TestRunner` within the test session, specified as a numeric array. `SuiteIndices` specifies the locations of the finalized results in the entire array of test results. `SuiteIndices` can be unsorted and can contain noncontiguous indices.

`CommunicationBuffer` — Buffer for finalized test suite portion
`matlab.unittest.plugins.plugindata.CommunicationBuffer` object

Buffer for storing and retrieving data related to the finalized test suite portion, specified as an instance of the `matlab.unittest.plugins.plugindata.CommunicationBuffer` class. Use `CommunicationBuffer` with the `storeIn` and `retrieveFrom` methods of the `matlab.unittest.plugins.Parallelizable` interface.

See Also

`matlab.unittest.TestResult` | `matlab.unittest.TestRunner` | `matlab.unittest.TestSuite` | `matlab.unittest.plugins.Parallelizable` | `matlab.unittest.plugins.TestRunnerPlugin` | `matlab.unittest.plugins.plugindata.CommunicationBuffer`

Introduced in R2019b

matlab.unittest.plugins.plugindata.ImplicitFixturePluginData class

Package: `matlab.unittest.plugins.plugindata`

Plugin data containing test setup and teardown information

Description

The `ImplicitFixturePluginData` class defines the data passed by the `TestRunner` to plugin methods related to setting up and tearing down tests. The `TestRunner` instantiates this class, so you are not required to create an object of the class directly.

Properties

`Name` — Label of test content `TestRunner` sets up or tears down
character vector

Label of test content `TestRunner` sets up or tears down, specified as a character vector. Use the `Name` property for informational, labeling, and display purposes. You cannot use this property to inspect the test content.

`QualificationContext` — Context for plugins to perform qualifications on test content
`matlab.unittest.plugins.plugindata.QualificationContext` object

Context for plugins to perform qualifications on the test content, specified as an instance of the `matlab.unittest.plugins.plugindata.QualificationContext` class. For your plugin to use qualifications, it must be a subclass of `matlab.unittest.plugins.QualifyingPlugin` and provide this context in the call to one of its qualifications methods.

See Also

`matlab.unittest.plugins.TestRunnerPlugin` | `setupTestClass` | `setupTestMethod` | `teardownTestClass` | `teardownTestMethod`

Introduced in R2015a

matlab.unittest.plugins.plugindata.PluginData class

Package: `matlab.unittest.plugins.plugindata`

Data object passed to `TestRunnerPlugin` methods

Description

The `PluginData` class defines the data passed by the `TestRunner` to various plugin methods. The `TestRunner` instantiates this class, so you are not required to create an object of the class directly.

Properties

`Name`

Label of test content executed by the `TestRunner` within the scope of a plugin method, specified as a character vector. Use the `Name` property for informational, labeling, and display purposes. You cannot use this property to inspect the test content.

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.plugins.TestRunnerPlugin`

Introduced in R2014a

matlab.unittest.plugins.plugindata.QualificationContext class

Package: `matlab.unittest.plugins.plugindata`

Context for qualifying plugins

Description

The `QualificationContext` class provides the context required for qualifying plugins to perform qualifications. The `TestRunner` instantiates this class and stores it as part of the relevant plugin data, so you are not required to create an object of the class directly.

See Also

`matlab.unittest.plugins.QualifyingPlugin` | `matlab.unittest.plugins.plugindata.ImplicitFixturePluginData` | `matlab.unittest.plugins.plugindata.SharedTestFixturePluginData`

Introduced in R2015b

matlab.unittest.plugins.plugindata.RunPluginData class

Package: `matlab.unittest.plugins.plugindata`
Superclasses: `matlab.unittest.plugins.plugindata.PluginData`

Plugin data describing `Test` elements being run

Description

The `RunPluginData` class defines the data passed by the `TestRunner` to plugin methods that extend the running of tests related to a method, class, or function. The `TestRunner` instantiates this class, so you are not required to create an object of the class directly.

The `matlab.unittest.plugins.plugindata.RunPluginData` class is a `handle` class.

Properties

`Name` — Name of `Test` elements

character vector

Name of the `Test` elements being run within the scope of a plugin method, specified as a character vector. The `Name` property is for informational, labeling, and display purposes. You cannot use this property to inspect the content of the `Test` elements being run.

`TestSuite` — Specification of `Test` elements

`matlab.unittest.TestSuite` array

Specification of the `Test` elements being run within the scope of a plugin method, specified as a `matlab.unittest.TestSuite` array.

`TestResult` — Results from running `Test` elements

`matlab.unittest.TestResult` array

Results from running the `Test` elements of the `TestSuite` property, specified as a `matlab.unittest.TestResult` array.

See Also

`matlab.unittest.TestResult` | `matlab.unittest.TestSuite` | `matlab.unittest.plugins.TestRunnerPlugin`

Introduced in R2019b

matlab.unittest.plugins.plugindata.SharedTestFixturePluginData class

Package: `matlab.unittest.plugins.plugindata`

Plugin data containing shared test fixture information

Description

The `SharedTestFixturePluginData` defines the data passed by the `TestRunner` to plugin methods related to shared test fixtures. The `TestRunner` instantiates this class, so you are not required to create an object of the class directly.

Properties

`Name`

Label of shared test fixture, specified as a character vector. The `Name` property is for informational, labeling, and display purposes. You cannot use this property to inspect the test content.

`Description`

Description of action performed during setup and teardown of a shared test fixture, specified as a character vector.

`QualificationContext`

Context for plugins to perform qualifications on fixtures, specified as an instance of the `matlab.unittest.plugins.plugindata.QualificationContext` class. For your plugin to use qualifications, it must be a subclass of `matlab.unittest.plugins.QualifyingPlugin` and provide this context in the call to one of its qualifications methods.

See Also

`matlab.unittest.fixtures.Fixture` | `matlab.unittest.plugins.TestRunnerPlugin`

Introduced in R2014a

matlab.unittest.plugins.plugindata.TestContentCreationPluginData class

Package: `matlab.unittest.plugins.plugindata`

Plugin data containing test content creation information

Description

The `TestContentCreationPluginData` class defines the data passed by the `TestRunner` to plugin methods related to test content creation. The `TestRunner` instantiates this class, so you are not required to create an object of the class directly.

Properties

`Name` — Name of content `TestRunner` executes
string scalar

Name of content the `TestRunner` executes within the scope of a plugin method, specified as a string scalar. The `Name` property is for informational, labeling, and display purposes. You cannot use this property to inspect the test content.

See Also

`matlab.unittest.TestRunner` | `matlab.unittest.plugins.TestRunnerPlugin`

Introduced in R2017a

matlab.unittest.plugins.plugindata.TestSuiteRunPluginData class

Package: `matlab.unittest.plugins.plugindata`

Plugin data describing `TestSuite` portion being run

Description

The `TestSuiteRunPluginData` class defines the data passed by the `TestRunner` to plugin methods that extend the running of test suite portions. The `TestRunner` instantiates this class, so you are not required to create an object of the class directly.

Properties

`Name`

Name of the test suite portion being run by the `TestRunner`, specified as a character vector. Use the `Name` property for informational, labeling, and display purposes. You cannot use this property to inspect the test content.

`TestSuite`

Specification of the elements in the test suite portion being run, specified as a `matlab.unittest.TestSuite` array.

`TestResult`

Results from the test suite portion being run, specified as a `matlab.unittest.TestResult` array.

`NumGroups`

Number of groups into which the entire test suite is divided, specified as a positive integer scalar. `NumGroups` is equal to `1` when tests run in serial mode.

`Group`

Identifier of the test suite portion being run, specified as a positive integer scalar between `1` and `NumGroups`.

`CommunicationBuffer`

Buffer for storing and retrieving data related to the test suite portion being run, specified as an instance of the `matlab.unittest.plugins.plugindata.CommunicationBuffer` class. Use `CommunicationBuffer` with the `storeIn` and `retrieveFrom` methods of the `matlab.unittest.plugins.Parallelizable` interface.

See Also

`matlab.unittest.TestResult` | `matlab.unittest.TestRunner` | `matlab.unittest.TestSuite` | `matlab.unittest.plugins.Parallelizable` | `matlab.unittest.plugins.TestRunnerPlugin` | `matlab.unittest.plugins.plugindata.CommunicationBuffer`

Introduced in R2014a

plus, +

Addition

Syntax

```
C = A + B
C = plus(A,B)
```

Description

`C = A + B` adds arrays `A` and `B` by adding corresponding elements. The sizes of `A` and `B` must be the same or be compatible.

If the sizes of `A` and `B` are compatible, then the two arrays implicitly expand to match each other. For example, if one of `A` or `B` is a scalar, then the scalar is combined with each element of the other array. Also, vectors with different orientations (one row vector and one column vector) implicitly expand to form a matrix.

`C = plus(A,B)` is an alternate way to execute `A + B`, but is rarely used. It enables operator overloading for classes.

Examples

Add Scalar to Array

Create an array, `A`, and add a scalar value to it.

```
A = [0 1; 1 0];
C = A + 2
```

```
C = 2×2

     2     3
     3     2
```

The scalar value is added to each entry of A.

Add Two Arrays

Create two arrays, A and B, and add them together.

```
A = [1 0; 2 4];
B = [5 9; 2 1];
C = A + B
```

```
C = 2×2

     6     9
     4     5
```

The elements of A are added to the corresponding elements of B.

Add Row and Column Vectors

Create a 1-by-2 row vector and 3-by-1 column vector and add them.

```
a = 1:2;
b = (1:3)';
a + b
```

```
ans = 3×2

     2     3
     3     4
     4     5
```

The result is a 3-by-2 matrix, where each (i,j) element in the matrix is equal to `a(j) + b(i)`:

$$a = [a_1\, a_2], \qquad b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}, \qquad a + b = \begin{bmatrix} a_1 + b_1 & a_2 + b_1 \\ a_1 + b_2 & a_2 + b_2 \\ a_1 + b_3 & a_2 + b_3 \end{bmatrix}.$$

Add Vector to Matrix

Create an array, A, and add a column vector to it. The vector is treated as though it is a matrix of the same size as A, so that each element in the vector is added to a row in A.

```
A = [1 2 3; 4 5 6]
```

```
A = 2×3

     1     2     3
     4     5     6
```

```
b = [10; 100]
```

```
b = 2×1

    10
   100
```

```
A + b
```

```
ans = 2×3

    11    12    13
   104   105   106
```

Concatenate Strings

Create two 1-by-3 string arrays, then concatenate similarly located strings in the arrays.

```
s1 = string({'Red' 'Blue' 'Green'})
```

```
s1 = 1x3 string array
    "Red"    "Blue"    "Green"
```

```
s2 = string({'Truck' 'Sky' 'Tree'})
```

```
s2 = 1x3 string array
    "Truck"    "Sky"    "Tree"
```

```
s = s1 + s2
```

```
s = 1x3 string array
    "RedTruck"    "BlueSky"    "GreenTree"
```

Input Arguments

A, B — Operands
scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Numeric or string inputs A and B must either be the same size or have sizes that are compatible (for example, A is an M-by-N matrix and B is a scalar or 1-by-N row vector). For more information, see “Compatible Array Sizes for Basic Operations”.

- Operands with an integer data type cannot be complex.
- Datetime, duration, and calendar duration arrays must be the same size unless one is a scalar.
- If one input is a datetime array, duration array, or calendar duration array, then numeric values in the other input are treated as a number of 24-hour days.
- If one input is a string array, then the other input can be a numeric, logical, character, string, or cell array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `datetime` | `duration` | `calendarDuration`
Complex Number Support: Yes

Compatibility Considerations

Implicit expansion change affects arguments for operators

Behavior changed in R2016b

Starting in R2016b with the addition of implicit expansion, some combinations of arguments for basic operations that previously returned errors now produce results. For example, you previously could not add a row and a column vector, but those operands are now valid for addition. In other words, an expression like `[1 2] + [1; 2]` previously returned a size mismatch error, but now it executes.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a `try/catch` block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see "Compatible Array Sizes for Basic Operations".

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see "Tall Arrays".

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- 64-bit integers are not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cumsum` | `minus` | `sum` | `uplus`

Topics

“Array vs. Matrix Operations”
“Operator Precedence”
“MATLAB Operators and Special Characters”

Introduced before R2006a

pointLocation

Class: `DelaunayTri`

(Not recommended) Simplex containing specified location

Note `pointLocation(DelaunayTri)` is not recommended. Use `pointLocation(triangulation)` instead.

`DelaunayTri` is not recommended. Use `delaunayTriangulation` instead.

Syntax

```
SI = pointLocation(DT,QX)
SI = pointLocation(DT,QX,QY)
SI = pointLocation(DT,QX,QY,QZ)
[SI, BC] = pointLocation(DT,...)
```

Description

`SI = pointLocation(DT,QX)` returns the indices `SI` of the enclosing simplex (triangle/tetrahedron) for each query point location in `QX`. The enclosing simplex for point `QX(k,:)` is `SI(k)`. `pointLocation` returns `NaN` for all points outside the convex hull.

`SI = pointLocation(DT,QX,QY)` and `SI = pointLocation(DT,QX,QY,QZ)` allow the query point locations to be specified in alternative column vector format when working in 2-D and 3-D.

`[SI, BC] = pointLocation(DT,...)` returns the barycentric coordinates `BC`.

Input Arguments

`DT`	Delaunay triangulation.
`QX`	Matrix of size `mpts`-by-`ndim`, `mpts` being the number of query points.

Output Arguments

`SI`	Column vector of length `mpts` containing the indices of the enclosing simplex for each query point. `mpts` is the number of query points.
`BC`	`BC` is a `mpts`-by-`ndim` matrix, each row `BC(i,:)` represents the barycentric coordinates of `QX(i,:)` with respect to the enclosing simplex `SI(i)`.

Examples

Example 1

Create a 2-D Delaunay triangulation:

```
X = rand(10,2);
dt = DelaunayTri(X);
```

Find the triangles that contain specified query points:

```
qrypts = [0.25 0.25; 0.5 0.5];
triids = pointLocation(dt, qrypts)
```

Example 2

Create a 3-D Delaunay triangulation:

```
x = rand(10,1);
y = rand(10,1);
z = rand(10,1);
dt = DelaunayTri(x,y,z);
```

Find the triangles that contain specified query points and evaluate the barycentric coordinates:

```
qrypts = [0.25 0.25 0.25; 0.5 0.5 0.5];
[tetids, bcs] = pointLocation(dt, qrypts)
```

See Also

`delaunayTriangulation` | `nearestNeighbor` | `triangulation`

pol2cart

Transform polar or cylindrical coordinates to Cartesian

Syntax

```
[x,y] = pol2cart(theta,rho)
[x,y,z] = pol2cart(theta,rho,z)
```

Description

`[x,y] = pol2cart(theta,rho)` transforms corresponding elements of the polar coordinate arrays `theta` and `rho` to two-dimensional Cartesian, or *xy*, coordinates.

`[x,y,z] = pol2cart(theta,rho,z)` transforms corresponding elements of the cylindrical coordinate arrays `theta`, `rho`, and `z` to three-dimensional Cartesian, or *xyz*, coordinates.

Examples

Polar to Cartesian Coordinates

Convert the polar coordinates defined by corresponding entries in the matrices `theta` and `rho` to two-dimensional Cartesian coordinates `x` and `y`.

```
theta = [0 pi/4 pi/2 pi]
```

```
theta = 1×4

         0    0.7854    1.5708    3.1416
```

```
rho = [5 5 10 10]
```

```
rho = 1×4
```

```
     5     5    10    10
```

```
[x,y] = pol2cart(theta,rho)
```

```
x = 1×4

    5.0000    3.5355    0.0000  -10.0000


y = 1×4

         0    3.5355   10.0000    0.0000
```

Cylindrical to Cartesian Coordinates

Convert the cylindrical coordinates defined by corresponding entries in the matrices `theta`, `rho`, and `z` to three-dimensional Cartesian coordinates `x`, `y`, and `z`.

```
theta = [0 pi/4 pi/2 pi]'
```

```
theta = 4×1

         0
    0.7854
    1.5708
    3.1416
```

```
rho = [1 3 4 5]'
```

```
rho = 4×1

     1
     3
     4
     5
```

```
z = [7 8 9 10]'
```

```
z = 4×1
```

```
     7
     8
     9
    10
```

```
[x,y,z] = pol2cart(theta,rho,z)
```

```
x = 4×1

    1.0000
    2.1213
    0.0000
   -5.0000


y = 4×1

         0
    2.1213
    4.0000
    0.0000


z = 4×1

     7
     8
     9
    10
```

Input Arguments

`theta` — Angular coordinate
scalar | vector | matrix | multidimensional array

Angular coordinate, specified as a scalar, vector, matrix, or multidimensional array. `theta`, `rho`, and `z` must be the same size, or any of them can be scalar.

`theta` is the counterclockwise angle in the *x*-*y* plane measured in radians from the positive *x*-axis.

Data Types: `single` | `double`
Complex Number Support: Yes

`rho` — Radial coordinate
scalar | vector | matrix | multidimensional array

Radial coordinate, specified as a scalar, vector, matrix, or multidimensional array. `theta`, `rho`, and `z` must be the same size, or any of them can be scalar.

`rho` is the distance from the origin to a point in the *x*-*y* plane.

Data Types: `single` | `double`
Complex Number Support: Yes

`z` — Elevation coordinate
scalar | vector | matrix | multidimensional array

Elevation coordinate, specified as a scalar, vector, matrix, or multidimensional array. `theta`, `rho`, and `z` must be the same size, or any of them can be scalar.

`z` is the height above the *x*-*y* plane.

Data Types: `single` | `double`
Complex Number Support: Yes

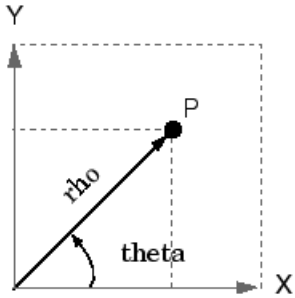
Output Arguments

`x`, `y`, `z` — Cartesian coordinates
arrays

Cartesian coordinates, returned as arrays.

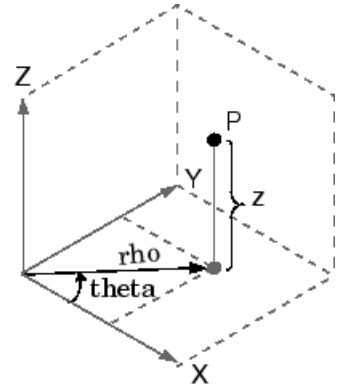
Algorithms

The mapping from polar and cylindrical coordinates to Cartesian coordinates is:



Polar to Cartesian Mapping

```
x = rho*cos(theta)
y = rho*sin(theta)
```



Polar to Cartesian Mapping

```
x = rho*cos(theta)
y = rho*sin(theta)
z = z
```

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cart2pol` | `cart2sph` | `sph2cart`

Introduced before R2006a

polar

(Not recommended) Polar coordinate plot

Note `polar` is not recommended. Use `polarplot` instead.

Syntax

```
polar(theta,rho)
polar(theta,rho,LineSpec)
polar(axes_handle,...)
h = polar(...)
```

Description

The `polar` function accepts polar coordinates, plots them in a Cartesian plane, and draws the polar grid on the plane.

`polar(theta,rho)` creates a polar coordinate plot of the angle `theta` versus the radius `rho`. `theta` is the angle from the *x*-axis to the radius vector specified in radians; `rho` is the length of the radius vector specified in dataspace units.

`polar(theta,rho,LineSpec)` `LineSpec` specifies the line type, plot symbol, and color for the lines drawn in the polar plot.

`polar(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

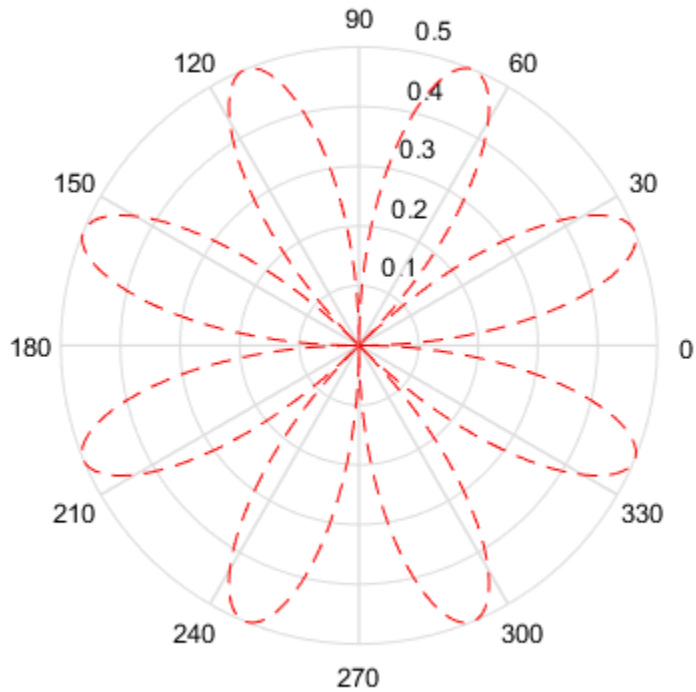
`h = polar(...)` returns the line object in `h`.

Examples

Simple Polar Plot

Create a simple polar plot using a dashed red line.

```
theta = 0:0.01:2*pi;
rho = sin(2*theta).*cos(2*theta);

figure
polar(theta,rho,'--r')
```



Tips

Negative `r` values reflect through the origin, rotating by `pi` (since `(theta,r)` transforms to `(r*cos(theta), r*sin(theta))`). If you want different behavior, you can manipulate `r` prior to plotting. For example, you can make `r` equal to `max(0,r)` or `abs(r)`.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cart2pol` | `compass` | `pol2cart` | `polarplot` | `rose`

Introduced before R2006a

polaraxes

Create polar axes

Syntax

```
polaraxes
polaraxes(Name,Value)
polaraxes(parent, ___ )
pax = polaraxes( ___ )

polaraxes(pax_in)
```

Description

`polaraxes` creates the default polar axes in the current figure.

`polaraxes(Name,Value)` specifies properties for the `PolarAxes` object using one or more name-value pair arguments, for example, `'ThetaDir','clockwise'`. For a list of properties, see PolarAxes.

`polaraxes(parent, ___ )` creates the polar axes in the figure, panel, or tab specified by `parent`, instead of in the current figure. Use this option alone or with name-value pair arguments.

`pax = polaraxes( ___ )`returns the `PolarAxes` object created. Use `pax` to query and set properties of the `PolarAxes` object after it is created. For a list of properties, see PolarAxes.

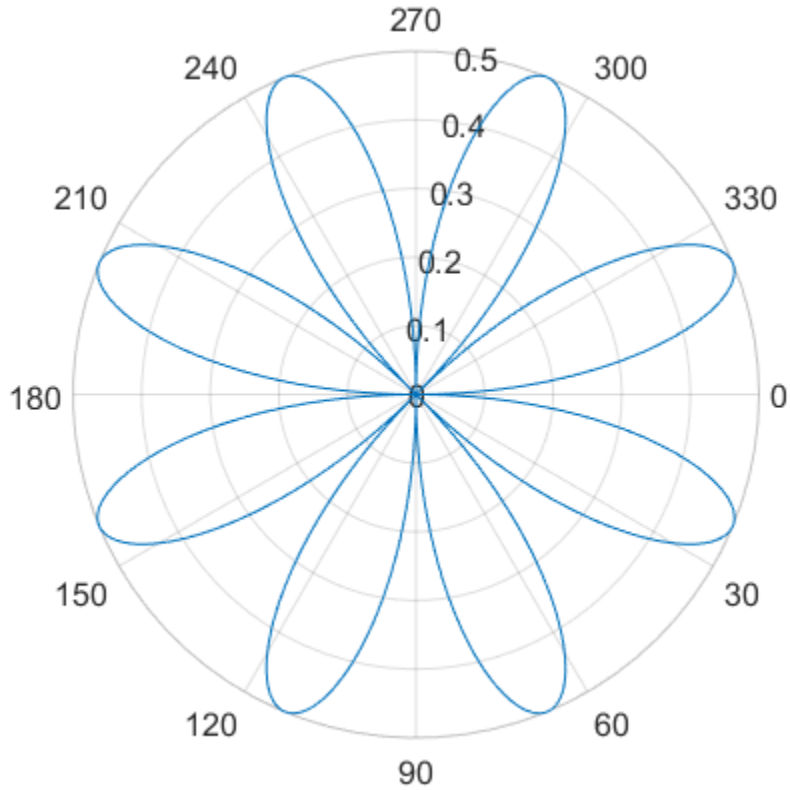
`polaraxes(pax_in)` makes the `PolarAxes` object `pax_in` the current axes.

Examples

Modify Polar Axes Properties

Create a new figure with polar axes and assign the polar axes object to `pax`. Add a plot to the axes. Then, use `pax` to modify axes properties.

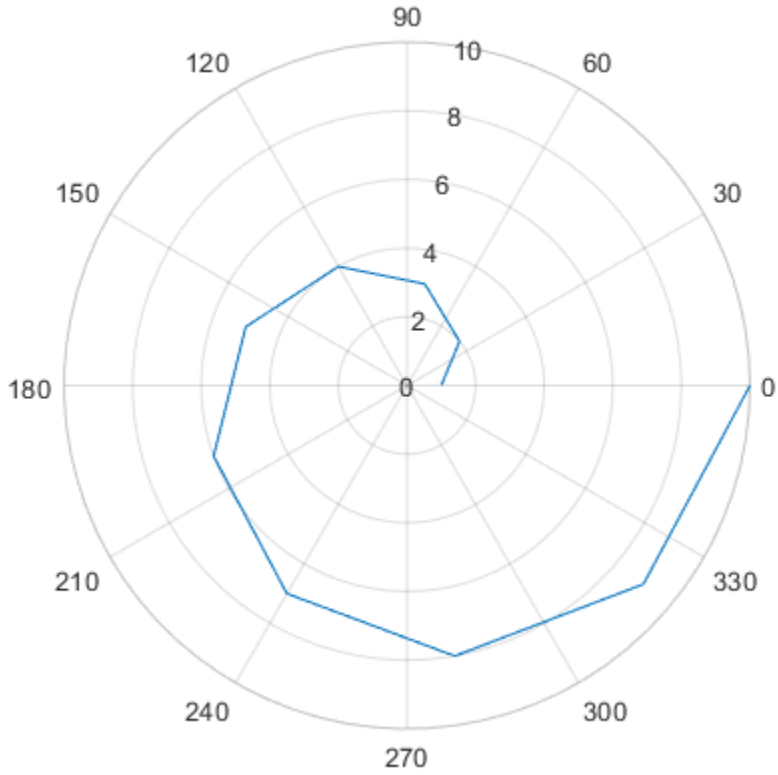
```
figure
pax = polaraxes;
theta = 0:0.01:2*pi;
rho = sin(2*theta).*cos(2*theta);
polarplot(theta,rho)

pax.ThetaDir = 'clockwise';
pax.FontSize = 12;
```



Make Polar Axes Current Axes

Create a figure with polar axes and assign the polar axes object to `pax`. Then, ensure `pax` is the current axes before calling the `polarplot` function.

```
figure
pax = polaraxes;

polaraxes(pax)
polarplot(1:10)
```



Input Arguments

parent — Parent container
`Figure` object | `Panel` object | `Tab` object | `TiledChartLayout` object

Parent container, specified as a `Figure`, `Panel`, `Tab`, or `TiledChartLayout` object.

pax_in — Polar axes to make current
`PolarAxes` object

Polar axes to make current, specified as a `PolarAxes` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The properties listed here are only a subset. For a complete list, see PolarAxes. Some graphics functions reset axes properties when plotting. To avoid graphics functions from overriding the property values, set axes properties after plotting.

Example: `'ThetaZeroLocation','top','ThetaDir','clockwise'`

ThetaAxisUnits — Units for angle values
`'degrees'` (default) | `'radians'`

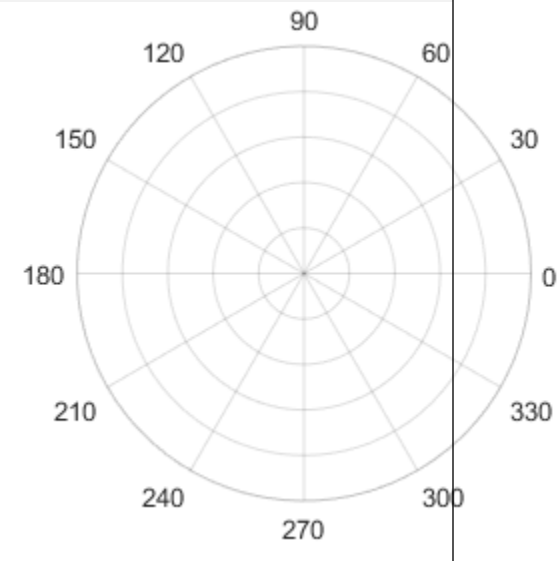
Units for angle values, specified as one of these values:

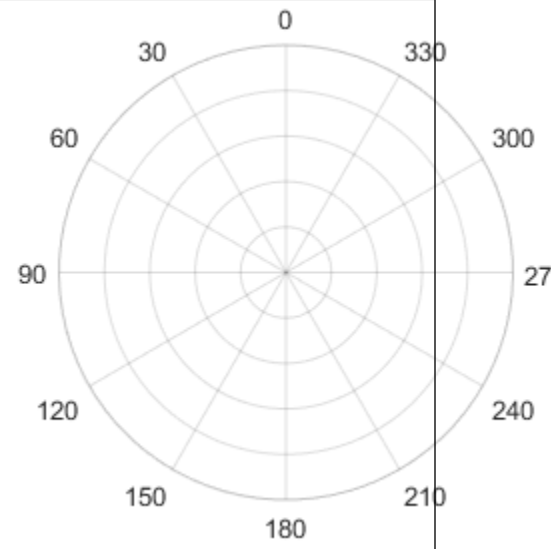
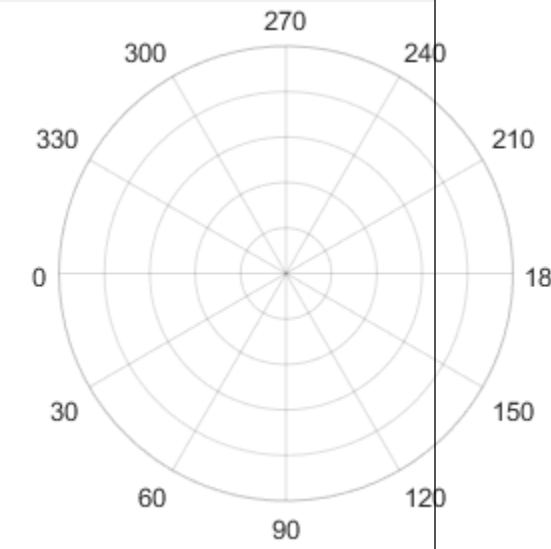
- `'degrees'` — Label the angles in degrees, and interpret the `ThetaLim`, `ThetaTick`, and `RAxisLocation` property values in degrees. When you switch the units from radians to degrees, MATLAB converts the radian values in those three properties to the equivalent degree values.
- `'radians'` — Label the angles in radians, and interpret the `ThetaLim`, `ThetaTick`, and `RAxisLocation` property values in radians. When you switch the units from degrees to radians, MATLAB converts the degree values in those three properties to the equivalent radian values.

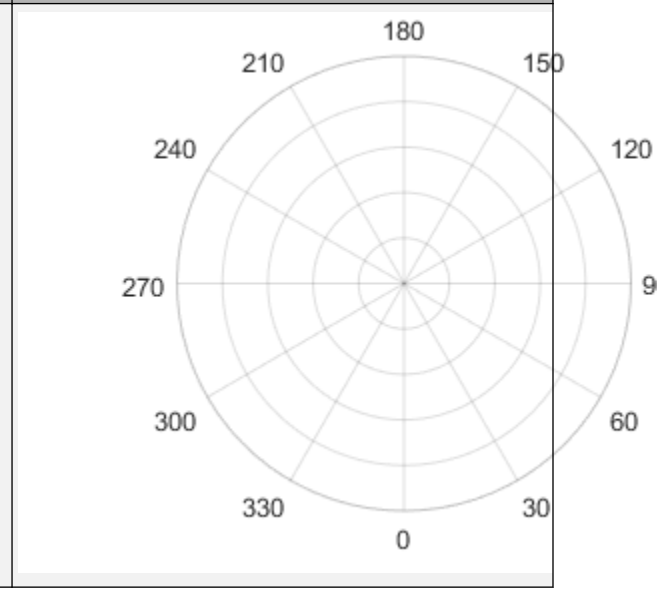
Example: `ax.ThetaAxisUnits = 'radians';`

`ThetaZeroLocation` — Location of the zero reference axis
`'right'` (default) | `'top'` | `'left'` | `'bottom'`

Location of the zero reference axis, specified as one of the values in this table.

Value	Result
`'right'`	

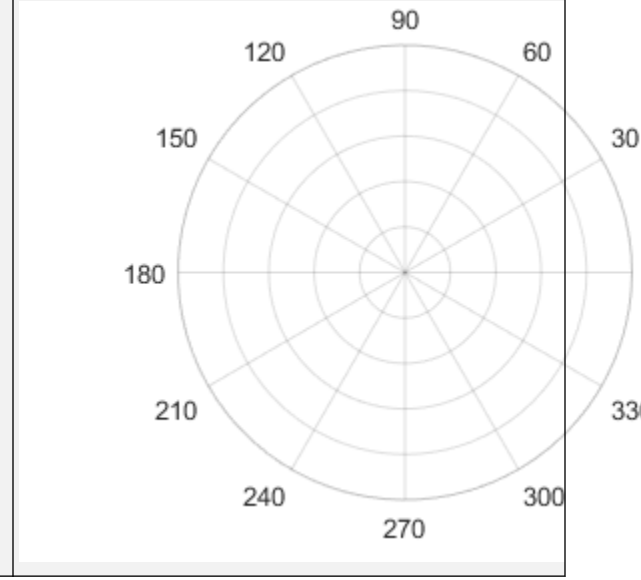
Value	Result
`'top'`	
`'left'`	

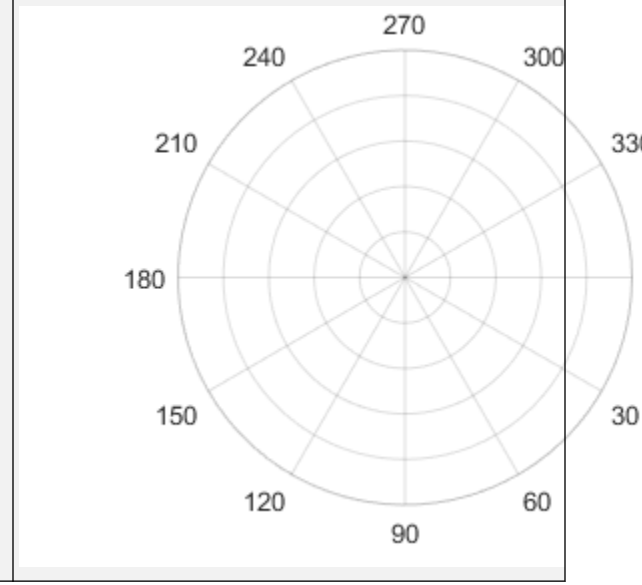
Value	Result
`'bottom'`	

Example: `ax.ThetaZeroLocation = 'left';`

`ThetaDir` — Direction of increasing angles
`'counterclockwise'` (default) | `'clockwise'`

Direction of increasing angles, specified as one of the values in this table.

Value	Result
`'counterclockwise'`	Angles increase in a counterclockwise direction. 

Value	Result
`'clockwise'`	Angles increase in a clockwise direction. 

Example: `ax.ThetaDir = 'clockwise';`

`ThetaLim` — Minimum and maximum angle values
`[0 360]` (default) | two-element vector of the form `[thmin thmax]`

Minimum and maximum angle values, specified as a two-element vector of the form `[thmin thmax]`. If the difference between the values is less than 360 degrees, then the theta-axis is a partial circle.

MATLAB interprets the values in units determined by the `ThetaAxisUnits` property.

Example: `ax.ThetaLim = [0 180];`

`ThetaTick` — Angles at which to display lines
`[0 30 60 ... 300 330 360]` (default) | vector of increasing values

Angles at which to display lines extending from the origin, specified as a vector of increasing values. MATLAB labels the lines with the appropriate angle values, unless you specify different labels using the `ThetaTickLabel` property.

MATLAB interprets the values in units determined by the `ThetaAxisUnits` property.

Example: `ax.ThetaTick = [0 90 180 270];`

Alternatively, specify the values using the `thetaticks` function.

`ThetaColor` — Color of *theta*-axis
`[0.15 0.15 0.15]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of the *theta*-axis, including the *theta*-axis grid lines, tick marks, tick labels. Specify this value as an RGB triplet, a hexadecimal color code, a color name, or a short name.

Note If you specify the `GridColor` property, then the grid lines use the color in the `GridColor` property instead. See `GridColorMode` for more information.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

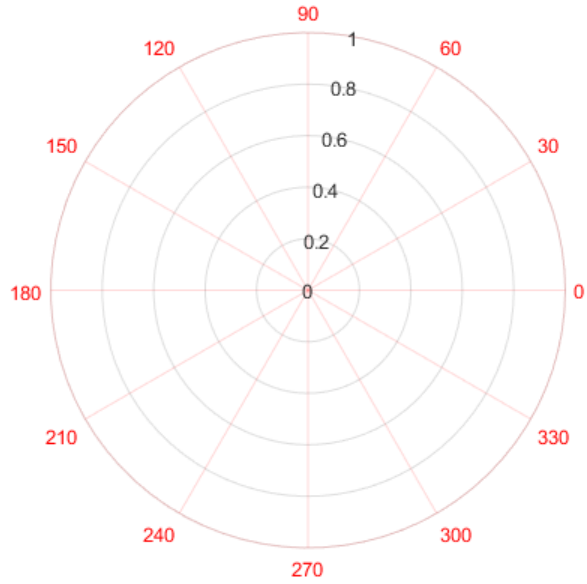
Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

For example, `ax.ThetaColor = 'r'` changes the color to red.



`RAxisLocation` — Location of r-axis
`80` (default) | scalar angle value

Location of the r-axis, specified a scalar angle value. MATLAB interprets the values in units determined by the `ThetaAxisUnits` property.

Example: `ax.RAxisLocation = 90;`

`RDir` — Direction of increasing values along r-axis
`'normal'` (default) | `'reverse'`

Direction of increasing values along the r-axis, specified as one of these values:

- `'normal'` — Values increase outward from the center of the chart. The radius at the origin corresponds to the minimum value stored in the `RLim` property (typically 0).
- `'reverse'` — Values decrease outward from the center of the chart. The radius at the origin corresponds to the maximum value stored in the `RLim` property.

Example: `ax.RDir = 'reverse';`

RLim — Minimum and maximum radius limits
`[0 1]` (default) | two-element vector of the form `[rmin rmax]`

Minimum and maximum radius limits, specified as a two-element vector of the form `[rmin rmax]`, where `rmax` is a numeric value greater than `rmin`. You can specify both limits, or specify one limit and let the axes automatically calculate the other.

- To automatically set the minimum limit to the minimum data value, specify the first element as `-inf`, for example, `[-inf 0]`.
- To automatically set the maximum limit to the maximum data value, specify the second element as `inf`, for example, `[0 inf]`.

Alternatively, use the `rlim` function to set the limits.

Example: `ax.RLim = [0 6];`

RTick — Radius tick values
`[0 0.2 0.4 0.6 0.8 1]` (default) | vector of increasing values

Radius tick values, specified as a vector of increasing values. The radius tick values are the locations along the *r*-axis where the circular lines appear. The radius tick labels are the labels that you see next to each line. Use the `RTickLabels` property to specify the associated labels.

Example: `ax.RTick = [0 2 4 6];`

Alternatively, use the `rticks` function to specify the tick values.

RColor — Color of *r*-axis
`[0.15 0.15 0.15]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of the *r*-axis, including the *r*-axis grid lines, tick marks, and tick labels. Specify this value as an RGB triplet, a hexadecimal color code, a color name, or a short name.

Note If you specify the `GridColor` property, then the grid lines use the color in the `GridColor` property instead. See `GridColorMode` for more information.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

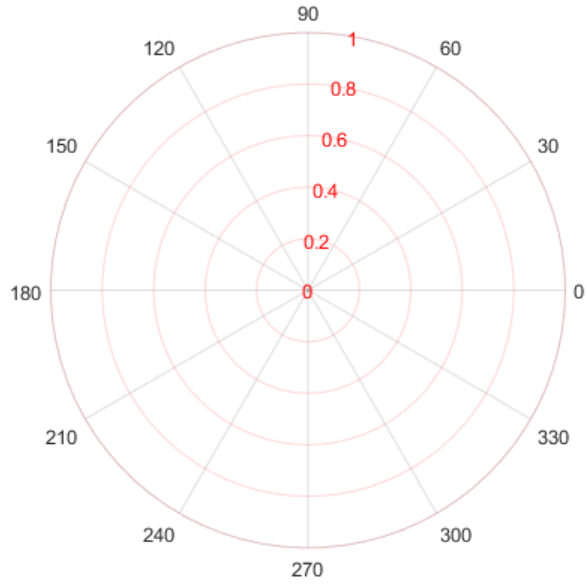
Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

For example, `ax.RColor = 'r'` changes the color to red.



See Also

PolarAxes | `polarplot` | `rlim`

Introduced in R2016a

PolarAxes Properties

Polar axes appearance and behavior

Description

`PolarAxes` properties control the appearance and behavior of a `PolarAxes` object. By changing property values, you can modify certain aspects of the polar axes. Set axes properties after plotting since some graphics functions reset axes properties.

Some graphics functions create polar axes when plotting. Use `gca` to access the newly created axes. To create empty polar axes, use the `polaraxes` function.

```
polarplot([0 pi/2 pi],[1 2 3])
ax = gca;
d = ax.ThetaDir;
ax.ThetaDir = 'clockwise';
```

Properties

Font

`FontName` — Font name

supported font name | `'FixedWidth'`

Font name, specified as a supported font name or `'FixedWidth'`. To display and print text properly, you must choose a font that your system supports. The default font depends on your operating system and locale.

To use a fixed-width font that looks good in any locale, use `'FixedWidth'`. The fixed-width font relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`FontSize` — Font size

scalar numeric value

Font size, specified as a scalar numeric value. The font size affects the title and tick labels. It also affects any legends or colorbars associated with the axes. The default font

size depends on the specific operating system and locale. By default, the font size is measured in points. To change the units, set the `FontUnits` property.

MATLAB automatically scales some of the text to a percentage of the axes font size.

- Titles — 110% of the axes font size by default. To control the scaling, use the `TitleFontSizeMultiplier` and `LabelFontSizeMultiplier` properties.
- Legends and colorbars — 90% of the axes font size by default. To specify a different font size, set the `FontSize` property for the `Legend` or `Colorbar` object instead.

Example: `ax.FontSize = 12`

`FontSizeMode` — Selection mode for font size
`'auto'` (default) | `'manual'`

Selection mode for the font size, specified as one of these values:

- `'auto'` — Font size specified by MATLAB. If you resize the axes to be smaller than the default size, the font size might scale down to improve readability and layout.
- `'manual'` — Font size specified manually. Do not scale the font size as the axes size changes. To specify the font size, set the `FontSize` property.

`FontWeight` — Character thickness
`'normal'` (default) | `'bold'`

Character thickness, specified as `'normal'` or `'bold'`.

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.

`FontAngle` — Character slant
`'normal'` (default) | `'italic'`

Character slant, specified as `'normal'` or `'italic'`.

Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.

`TitleFontSizeMultiplier` — Scale factor for title font size
`1.1` (default) | numeric value greater than 0

Scale factor for the title font size, specified as a numeric value greater than 0. The scale factor is applied to the value of the `FontSize` property to determine the font size for the title.

Example: `ax.TitleFontSizeMultiplier = 1.75`

`TitleFontWeight` — Title character thickness
`'bold'` (default) | `'normal'`

Title character thickness, specified as one of these values:

- `'bold'` — Thicker characters outlines than normal
- `'normal'` — Default weight as defined by the particular font

Example: `ax.TitleFontWeight = 'normal'`

`FontUnits` — Font size units
`'points'` (default) | `'inches'` | `'centimeters'` | `'normalized'` | `'pixels'`

Font size units, specified as one of these values.

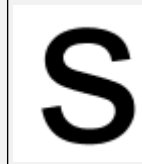
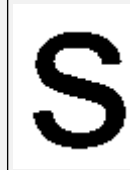
Units	Description
`'points'`	Points. One point equals 1/72 inch.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'normalized'`	Interpret font size as a fraction of the axes height. If you resize the axes, the font size modifies accordingly. For example, if the `FontSize` is `0.1` in normalized units, then the text is 1/10 of the height value stored in the axes `Position` property.

Units	Description
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems. • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. • On Linux systems, the size of a pixel is determined by your system resolution.

To set both the font size and the font units in a single function call, you first must set the `FontUnits` property so that the `Axes` object correctly interprets the specified font size.

`FontSmoothing` — Character smoothing
`'on'` (default) | `'off'`

Character smoothing, specified as `'on'` or `'off'`.

Value	Description	Result
`'on'`	Use antialiasing to make text appear smoother on the screen. **Example:** `ax.FontSmoothing = 'on'`	
`'off'`	Do not use antialiasing. Use this setting if the text seems blurry. **Example:** `ax.FontSmoothing = 'off'`	

Ticks

RTick — Radius tick values

`[0 0.2 0.4 0.6 0.8 1]` (default) | vector of increasing values

Radius tick values, specified as a vector of increasing values. The radius tick values are the locations along the *r*-axis where the circular lines appear. The radius tick labels are the labels that you see next to each line. Use the `RTickLabels` property to specify the associated labels.

Example: `ax.RTick = [0 2 4 6];`

Alternatively, use the `rticks` function to specify the tick values.

RTickMode — Selection mode for radius tick values

`'auto'` (default) | `'manual'`

Selection mode for the radius tick values, specified as one of these values:

- `'auto'` — Automatically select the tick values based on the range of data for the axis.
- `'manual'` — Manually specify the tick values. To specify the values, set the `RTick` property.

Example: `ax.RTickMode = 'auto'`

RTickLabel — Radius tick labels

`{'0'; '0.2'; '0.4'; '0.6'; '0.8'; '1'}` (default) | cell array of character vectors | string array | categorical array

Radius tick labels, specified as a cell array of character vectors, string array, or categorical array. If you do not want tick labels to show, then specify an empty cell array `{}`. If you do not specify enough labels for all the ticks values, then the labels repeat.

Tick labels support TeX and LaTeX markup. See the `TickLabelInterpreter` property for more information.

If you specify this property as a categorical array, MATLAB uses the values in the array, not the categories.

Example: `ax.RTickLabel = {'one','two','three','four'};`

Alternatively, use the `rticklabels` function.

RTickLabelMode — Selection mode for RTickLabel
`'auto'` (default) | `'manual'`

Selection mode for the `RTickLabel` property value, specified as one of these values:

- `'auto'` — Automatically select the tick labels.
- `'manual'` — Manually specify the tick labels. To specify the labels, set the `RTickLabel` property.

ThetaTick — Angles at which to display lines
`[0 30 60 ... 300 330 360]` (default) | vector of increasing values

Angles at which to display lines extending from the origin, specified as a vector of increasing values. MATLAB labels the lines with the appropriate angle values, unless you specify different labels using the `ThetaTickLabel` property.

MATLAB interprets the values in units determined by the `ThetaAxisUnits` property.

Example: `ax.ThetaTick = [0 90 180 270];`

Alternatively, specify the values using the `thetaticks` function.

ThetaTickMode — Selection mode for ThetaTick
`'auto'` (default) | `'manual'`

Selection mode for the `ThetaTick` property value, specified as one of these values:

- `'auto'` — Automatically select the property value.
- `'manual'` — Use the specified property value. To specify the value, set the `ThetaTick` property.

ThetaTickLabel — Labels for angle lines
`{'0'; '30'; '60'; ... '300'; '330'; '360'}` (default) | cell array of character vectors | string array | categorical array

Labels for angle lines, specified as a cell array of character vectors, string array, or categorical array.

If you do not specify enough labels for all the lines, then the labels repeat. Labels support TeX and LaTeX markup. See the `TickLabelInterpreter` property for more information.

If you specify this property as a categorical array, MATLAB uses the values in the array, not the categories.

Example: `ax.ThetaTickLabel = {'right','top','left','bottom'};`

Alternatively, specify the values using the `thetaticklabels` function.

`ThetaTickLabelMode` — Selection mode for `ThetaTickLabel`
`'auto'` (default) | `'manual'`

Selection mode for the `ThetaTickLabel` property value, specified as one of these values:

- `'auto'` — Automatically select the property value.
- `'manual'` — Use the specified property value. To specify the value, set the `ThetaTickLabel` property.

`RTickLabelRotation` — Rotation of *r*-axis tick labels
`0` (default) | scalar value in degrees

Rotation of *r*-axis tick labels, specified as a scalar value in degrees. Positive values give counterclockwise rotation. Negative values give clockwise rotation.

Example: `ax.RTickLabelRotation = 45;`

Alternatively, use the `rtickangle` function.

`RMinorTick` — Minor tick marks along *r*-axis
`'off'` | `'on'`

Minor tick marks along *r*-axis, specified as one of these values:

- `'off'` — Do not display minor tick marks.
- `'on'` — Display minor tick marks. The space between the major tick marks and grid lines determines the number of minor tick marks. This property value has a visual effect only if the tick length is positive (controlled by the `TickLength` property) and if the polar axes is a full circle (controlled by the `ThetaLim` property).

Example: `ax.RMinorTick = 'on';`

`ThetaMinorTick` — Minor tick marks between angle lines
`'off'` (default) | `'on'`

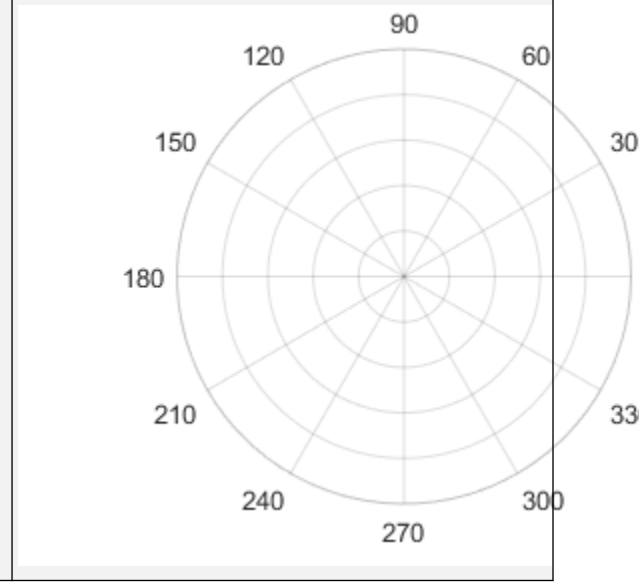
Minor tick marks between angle lines, specified as one of these values:

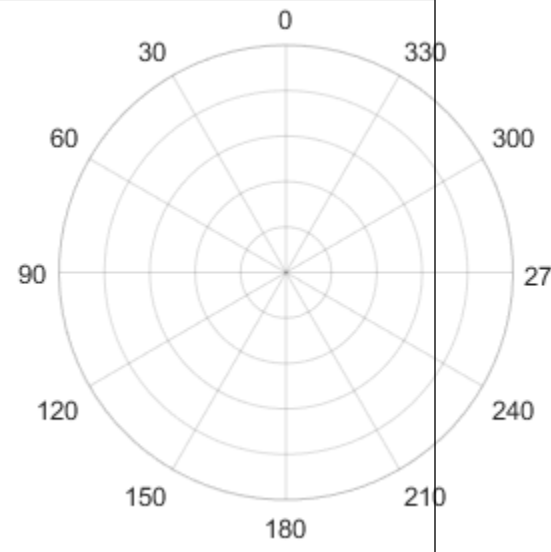
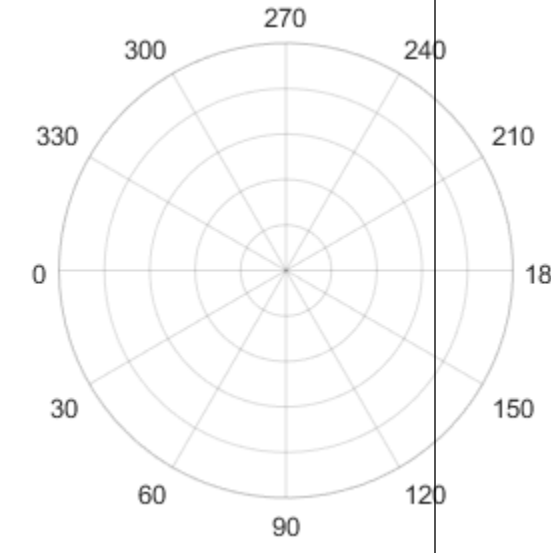
- `'off'` — Do not display minor tick marks.

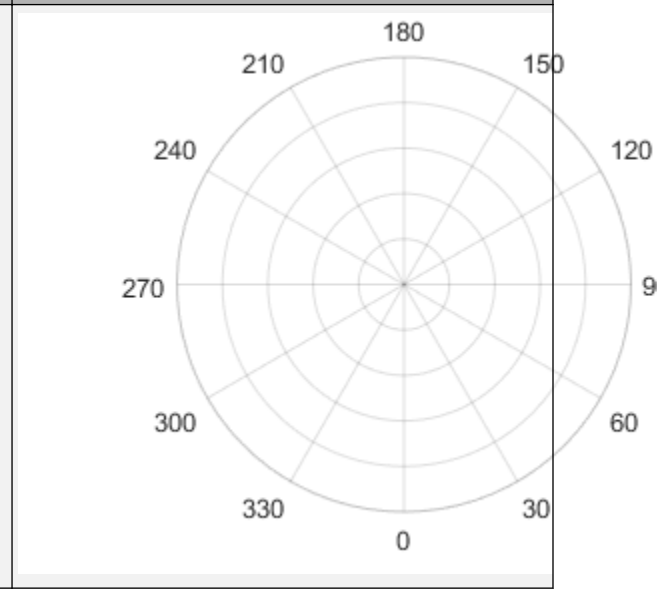
- `'on'` — Display minor tick marks. The space between the lines determines the number of minor tick marks. This property value has a visual effect only if the tick length is positive. To set the tick length, use the `TickLength` property, for example, `ax.TickLength = [0.02 0]`.

`ThetaZeroLocation` — Location of the zero reference axis
`'right'` (default) | `'top'` | `'left'` | `'bottom'`

Location of the zero reference axis, specified as one of the values in this table.

Value	Result
`'right'`	

Value	Result
`'top'`	
`'left'`	

Value	Result
`'bottom'`	

Example: `ax.ThetaZeroLocation = 'left';`

`TickDir` — Tick mark direction
`'in'` (default) | `'out'` | `'both'`

Tick mark direction, specified as one of these values:

- `'in'` — Direct the tick marks inward from the axes outline.
- `'out'` — Direct the tick marks outward from the axes outline.
- `'both'` — Center the tick marks over the axes outline.

Example: `ax.TickDir = 'out';`

`TickDirMode` — Selection mode for `TickDir`
`'auto'` (default) | `'manual'`

Selection mode for the `TickDir` property, specified as one of these values:

- `'auto'` — Automatically select the tick direction based on the current view.
- `'manual'` — Manually specify the tick direction. To specify the tick direction, set the `TickDir` property.

Example: `ax.TickDirMode = 'auto'`

`TickLabelInterpreter` — Interpretation of characters in tick labels

`'tex'` (default) | `'latex'` | `'none'`

Interpretation of tick label characters, specified as one of these values:

- `'tex'` — Interpret characters using a subset of TeX markup.
- `'latex'` — Interpret characters using a subset of LaTeX markup. When you specify the tick labels, use dollar signs around each element in the cell array.
- `'none'` — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the style and color, and include special characters in the label.

This table lists the supported modifiers when the `TickLabelInterpreter` property is set to `'tex'`, which is the default value. Modifiers remain in effect until the end of the text, except for superscripts and subscripts which only modify the next character or the text within the curly braces `{}`.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (rarely available)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{specifier}`	Set `specifier` as the name of a font family to change the font style. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`

Modifier	Description	Example
`\fontsize{specifier}`	Set `specifier` as a scalar numeric value to change the font size.	`'\fontsize{15} text'`
`\color{specifier}`	Set `specifer` as one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Set `specifier` as a three-element RGB triplet to change the font color.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	β	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ϵ	`\Delta`	Δ	`\spadesuit`	♠
`\zeta`	ζ	`\Theta`	Θ	`\leftrightarrow`	↔
`\eta`	η	`\Lambda`	Λ	`\leftarrow`	←
`\theta`	θ	`\Xi`	Ξ	`\Leftarrow`	⇐
`\vartheta`	ϑ	`\Pi`	Π	`\uparrow`	↑

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	μ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `TickLabelInterpreter` property to `'latex'`. When you specify the tick labels, use dollar signs around each element in the cell array, for example:

```
ax = polaraxes;
ax.ThetaTickLabels = {'$\sum_1^n$','$\int_1^n$','$\frac{1}{2}$'};
ax.TickLabelInterpreter = 'latex';
```

The displayed labels uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup. The maximum size of the labels that you can use with the LaTeX interpreter is 1200 characters. For multiline labels, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project website at https://www.latex-project.org/.

`TickLength` — Tick mark length

`[0 0]` (default) | two-element vector

Tick mark length, specified as a two-element vector. The first element determines the tick length. The second element is ignored.

Example: `ax.TickLength = [0.02 0];`

Rulers

`RLim` — Minimum and maximum radius limits

`[0 1]` (default) | two-element vector of the form `[rmin rmax]`

Minimum and maximum radius limits, specified as a two-element vector of the form `[rmin rmax]`, where `rmax` is a numeric value greater than `rmin`. You can specify both limits, or specify one limit and let the axes automatically calculate the other.

- To automatically set the minimum limit to the minimum data value, specify the first element as `-inf`, for example, `[-inf 0]`.
- To automatically set the maximum limit to the maximum data value, specify the second element as `inf`, for example, `[0 inf]`.

Alternatively, use the `rlim` function to set the limits.

Example: `ax.RLim = [0 6];`

RLimMode — Selection mode for RLim

`'auto'` (default) | `'manual`

Selection mode for the `RLim` property value, specified as one of these values:

- `'auto'`— Automatically set the property value.
- `'manual'`— Use the specified property value. To specify the value, set the `RLim` property.

ThetaLim — Minimum and maximum angle values

`[0 360]` (default) | two-element vector of the form `[thmin thmax]`

Minimum and maximum angle values, specified as a two-element vector of the form `[thmin thmax]`. If the difference between the values is less than 360 degrees, then the theta-axis is a partial circle.

MATLAB interprets the values in units determined by the `ThetaAxisUnits` property.

Example: `ax.ThetaLim = [0 180];`

ThetaLimMode — Selection mode for ThetaLim

`'auto'` (default) | `'manual`

Selection mode for the `ThetaLim` property value, specified as one of these values:

- `'auto'` — Automatically select the property value.
- `'manual'` — Use the specified property value. To specify the value, set the `ThetaLim` property.

RAxis — Component that controls appearance and behavior of r-axis

ruler object (default)

Component that controls the appearance and behavior of the *r*-axis, returned as a ruler object. When MATLAB creates polar axes, it automatically creates a ruler for the *r*-axis. Modify the appearance and behavior of this axis by accessing the associated ruler and setting ruler properties. For a list of options, see NumericRuler.

For example, change the color of the *r*-axis to red.

```
ax = polaraxes;
ax.RAxis.Color = 'r';
```

Use the `RAxis` properties to access the ruler objects and set ruler properties. If you want to set polar axes properties, set them directly on the `PolarAxes` object.

`ThetaAxis` — Component that controls appearance and behavior of theta-axis
ruler object (default)

Component that controls the appearance and behavior of the theta-axis, returned as a ruler object. When MATLAB creates polar axes, it automatically creates a numeric ruler for the *theta*-axis. Modify the appearance and behavior of this axis by accessing the associated ruler and setting ruler properties. For a list of options, see NumericRuler.

For example, change the color of the *theta*-axis to red.

```
ax = polaraxes;
ax.ThetaAxis.Color = 'r';
```

Use the `ThetaAxis` property to access the ruler object and set ruler properties. If you want to set polar axes properties, set them directly on the `PolarAxes` object.

`RAxisLocation` — Location of r-axis
`80` (default) | scalar angle value

Location of the r-axis, specified a scalar angle value. MATLAB interprets the values in units determined by the `ThetaAxisUnits` property.

Example: `ax.RAxisLocation = 90;`

`RAxisLocationMode` — Selection mode for `RAxisLocation`
`'auto'` (default) | `'manual'`

Selection mode for the `RAxisLocation` property value, specified as one of these values:

- `'auto'` — Automatically select the property value.
- `'manual'` — Use the specified property value. To specify the value, set the `RAxisLocation` property.

`RColor` — Color of *r*-axis
`[0.15 0.15 0.15]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of the *r*-axis, including the *r*-axis grid lines, tick marks, and tick labels. Specify this value as an RGB triplet, a hexadecimal color code, a color name, or a short name.

Note If you specify the `GridColor` property, then the grid lines use the color in the `GridColor` property instead. See `GridColorMode` for more information.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

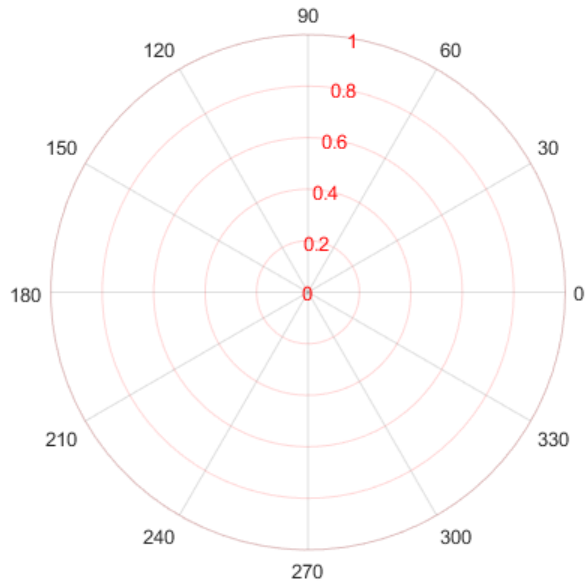
Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

For example, `ax.RColor = 'r'` changes the color to red.



RColorMode — Property for setting *r*-axis grid color
`'auto'` (default) | `'manual'`

Property for setting *r*-axis grid color, specified `'auto'` or `'manual'`. The mode value only affects the *r*-axis grid color. The *r*-axis tick labels always use the `RColor` value, regardless of the mode.

The *r*-axis grid color depends on both the `RColorMode` property and the `GridColorMode` property, as shown here.

RColorMode	GridColorMode	r-Axis Grid Color
'auto'	'auto'	GridColor property
	'manual'	GridColor property
'manual'	'auto'	RColor property
	'manual'	GridColor property

The *r*-axis minor grid color depends on both the RColorMode property and the MinorGridColorMode property, as shown here.

RColorMode	MinorGridColorMode	r-Axis Minor Grid Color
'auto'	'auto'	MinorGridColor property
	'manual'	MinorGridColor property
'manual'	'auto'	RColor property
	'manual'	MinorGridColor property

ThetaColor — Color of *theta*-axis

[0.15 0.15 0.15] (default) | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...

Color of the *theta*-axis, including the *theta*-axis grid lines, tick marks, tick labels. Specify this value as an RGB triplet, a hexadecimal color code, a color name, or a short name.

Note If you specify the GridColor property, then the grid lines use the color in the GridColor property instead. See GridColorMode for more information.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

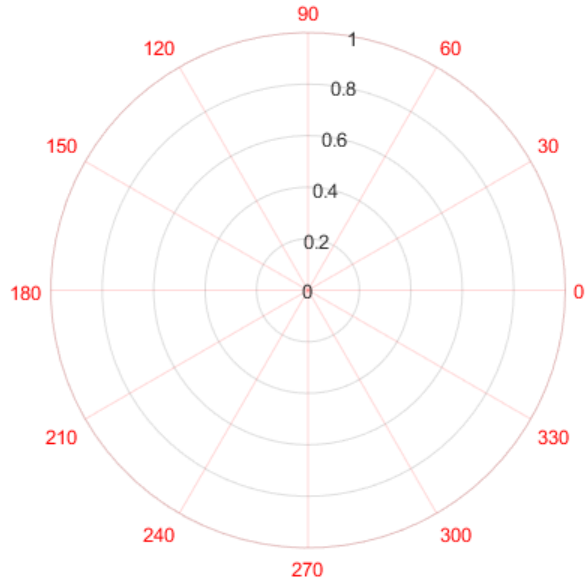
Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

For example, `ax.ThetaColor = 'r'` changes the color to red.



`ThetaColorMode` — Property for setting *theta*-axis grid color
`'auto'` (default) | `'manual'`

Property for setting *theta*-axis grid color, specified `'auto'` or `'manual'`. The mode value only affects the *theta*-axis grid color. The *theta*-axis line, tick marks, and labels always use the `ThetaColor` value, regardless of the mode.

The *theta*-axis grid color depends on both the `ThetaColorMode` property and the `GridColorMode` property, as shown here.

ThetaColorMode	GridColorMode	theta-Axis Grid Color
`'auto'`	`'auto'`	`GridColor` property
	`'manual'`	`GridColor` property
`'manual'`	`'auto'`	`ThetaColor` property
	`'manual'`	`GridColor` property

The *theta*-axis minor grid color depends on both the `ThetaColorMode` property and the `MinorGridColorMode` property, as shown here.

ThetaColorMode	MinorGridColorMode	theta-Axis Minor Grid Color
'auto'	'auto'	MinorGridColor property
	'manual'	MinorGridColor property
'manual'	'auto'	ThetaColor property
	'manual'	MinorGridColor property

RDir — Direction of increasing values along r-axis
'normal' (default) | 'reverse'

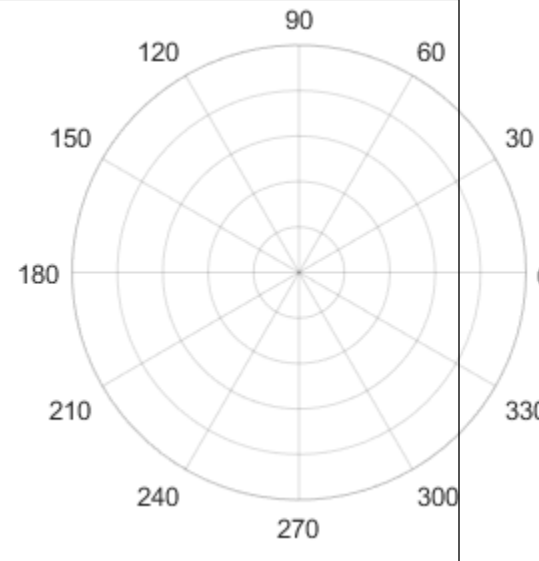
Direction of increasing values along the r-axis, specified as one of these values:

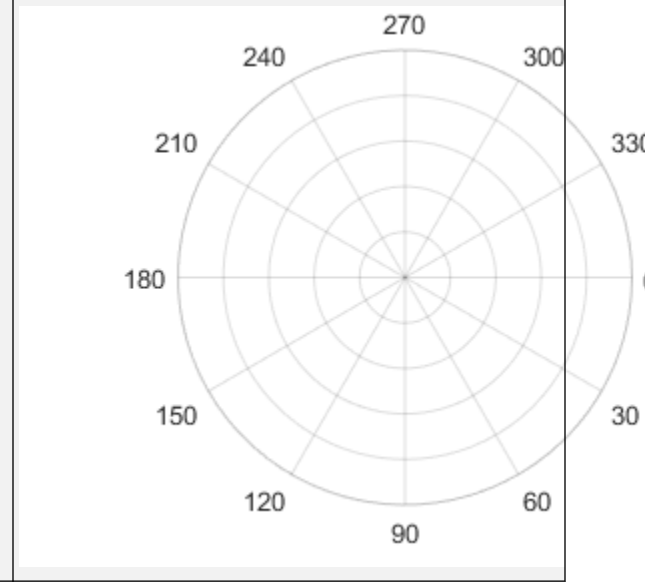
- 'normal' — Values increase outward from the center of the chart. The radius at the origin corresponds to the minimum value stored in the RLim property (typically 0).
- 'reverse' — Values decrease outward from the center of the chart. The radius at the origin corresponds to the maximum value stored in the RLim property.

Example: ax.RDir = 'reverse';

ThetaDir — Direction of increasing angles
'counterclockwise' (default) | 'clockwise'

Direction of increasing angles, specified as one of the values in this table.

Value	Result
`'counterclockwise'`	Angles increase in a counterclockwise direction. 

Value	Result
`'clockwise'`	Angles increase in a clockwise direction. 

Example: `ax.ThetaDir = 'clockwise';`

`ThetaAxisUnits` — Units for angle values
`'degrees'` (default) | `'radians'`

Units for angle values, specified as one of these values:

- `'degrees'` — Label the angles in degrees, and interpret the `ThetaLim`, `ThetaTick`, and `RAxisLocation` property values in degrees. When you switch the units from radians to degrees, MATLAB converts the radian values in those three properties to the equivalent degree values.
- `'radians'` — Label the angles in radians, and interpret the `ThetaLim`, `ThetaTick`, and `RAxisLocation` property values in radians. When you switch the units from degrees to radians, MATLAB converts the degree values in those three properties to the equivalent radian values.

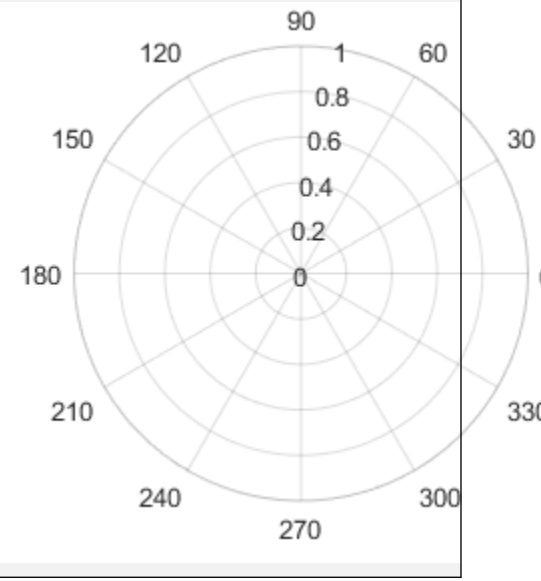
Example: `ax.ThetaAxisUnits = 'radians';`

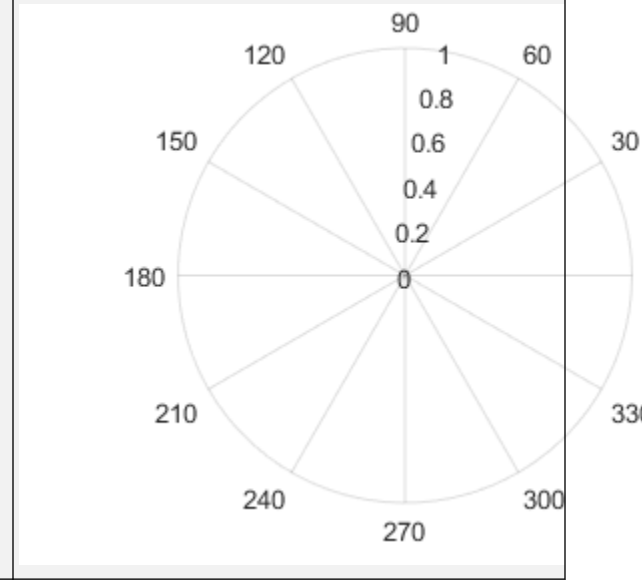
Grid Lines

`RGrid` — Display of *r*-axis grid lines

`'on'` (default) | `'off'`

Display of *r*-axis grid lines, specified as one of the values in this table.

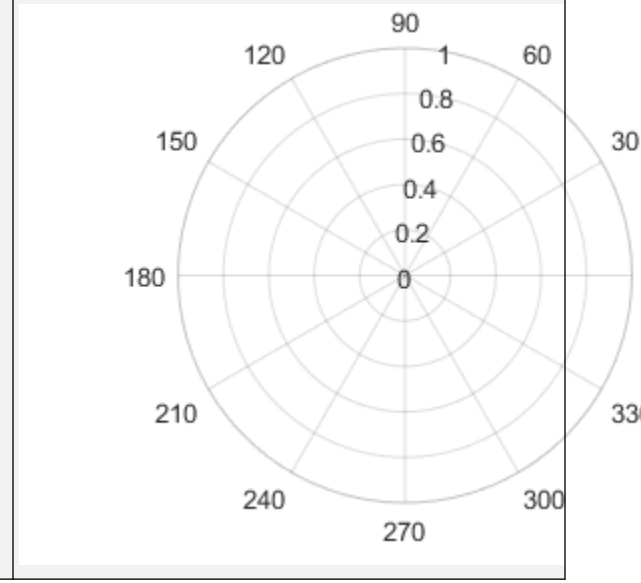
Value	Result
`'on'`	Display the lines. 

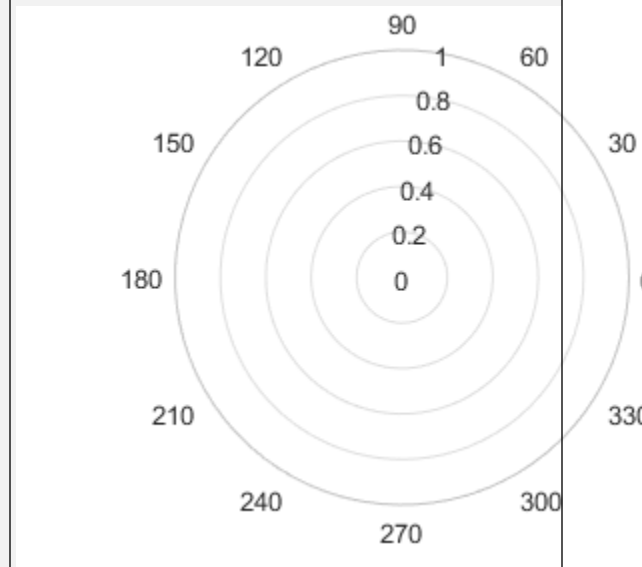
Value	Result
`'off'`	Do not display the lines. 

Example: `ax.RGrid = 'off';`

`ThetaGrid` — Display of *theta*-axis grid lines
`'on'` (default) | `'off'`

Display of *theta*-axis grid lines, specified as one of the values in this table.

Value	Result
`'on'`	Display the lines. 

Value	Result
`'off'`	Do not display the lines. 

Example: `ax.ThetaGrid = 'off';`

Layer — Placement of grid lines and tick marks
`'bottom'` (default) | `'top'`

Placement of grid lines and tick marks in relation to graphic objects, specified as one of these values:

- `'bottom'` — Display tick marks and grid lines under graphics objects.
- `'top'` — Display tick marks and grid lines over graphics objects.

This property affects only 2-D views.

Example: `ax.Layer = 'top'`

`GridLineStyle` — Line style for grid lines
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style used for grid lines, specified as one of the line styles in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

To display grid lines, use the `grid on` command or set the `ThetaGrid` or `RGrid` property to `'on'`.

Example: `ax.GridLineStyle = '--';`

`GridColor` — Color of grid lines

`[0.15 0.15 0.15]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of the grid lines, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The actual grid color depends on the values of the `GridColorMode`, `ThetaColorMode`, and `RColorMode` properties. See `GridColorMode` for more information.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `ax.GridColor = [0 0 1]`

Example: `ax.GridColor = 'blue'`

Example: `ax.GridColor = '#0000FF'`

`GridColorMode` — Property for setting grid color
`'auto'` (default) | `'manual'`

Property for setting the grid color, specified as one of these values:

- `'auto'` — Check the values of the `RColorMode` and `ThetaColorMode` properties to determine the grid line colors for the *r* and *theta* directions.
- `'manual'` — Use `GridColor` to set the grid line color for all directions.

`GridAlpha` — Grid-line transparency
`0.15` (default) | value in the range `[0,1]`

Grid-line transparency, specified as a value in the range `[0,1]`. A value of `1` means opaque and a value of `0` means completely transparent.

Example: `ax.GridAlpha = 0.5`

`GridAlphaMode` — Selection mode for `GridAlpha`
`'auto'` (default) | `'manual'`

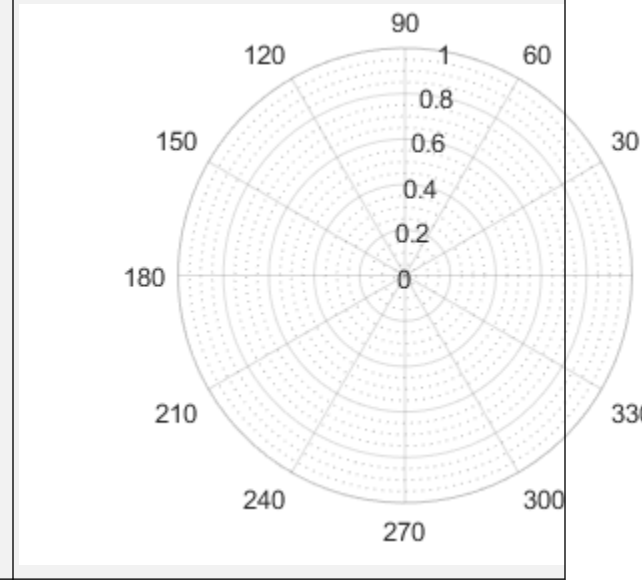
Selection mode for the `GridAlpha` property, specified as one of these values:

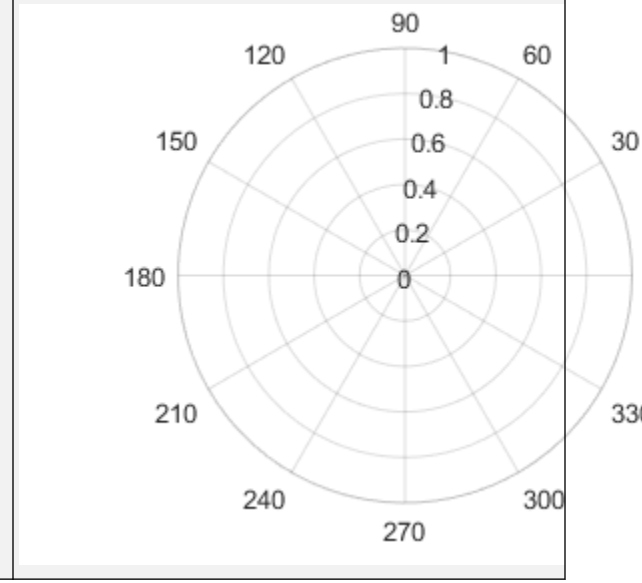
- `'auto'` — Default transparency value of `0.15`.
- `'manual'` — Manually specify the transparency value. To specify the value, set the `GridAlpha` property.

Example: `ax.GridAlphaMode = 'auto'`

`RMinorGrid` — Display of *r*-axis minor grid lines
`'off'` (default) | `'on'`

Display of *r*-axis minor grid lines, specified as one of the values in this table.

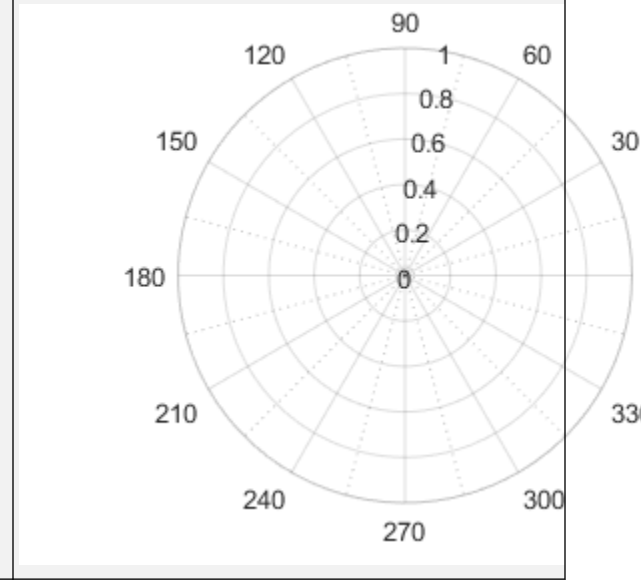
Value	Result
`'on'`	Display the lines. 

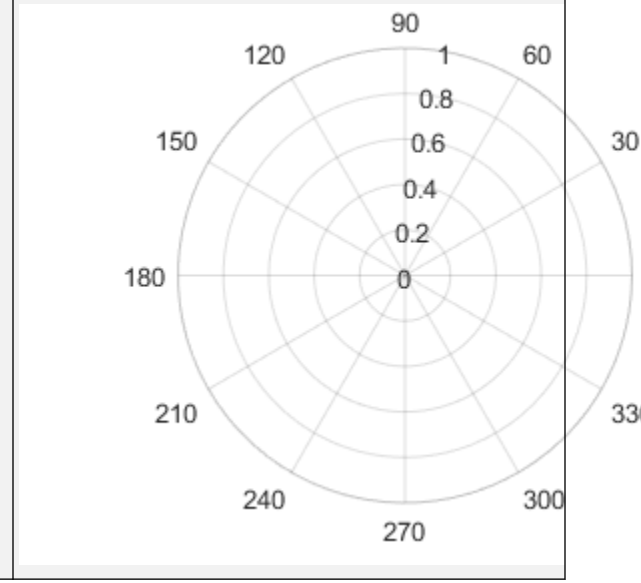
Value	Result
`'off'`	Do not display the lines. 

Example: `ax.RMinorGrid = 'on';`

`ThetaMinorGrid` — Display of *theta*-axis minor grid lines
`'off'` (default) | `'on'`

Display of *theta*-axis minor grid lines, specified as one of the values in this table.

Value	Result
`'on'`	Display the lines. 

Value	Result
`'off'`	Do not display the lines. 

Example: `ax.ThetaMinorGrid = 'on';`

`MinorGridLineStyle` — Line style for minor grid lines
`':'` (default) | `'-'` | `'--'` | `'-.'` | `'none'`

Line style used for minor grid lines, specified as one of the line styles in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

To display the grid lines, use the `grid minor` command or set the `ThetaMinorGrid` or `RMinorGrid` property to `'on'`.

Example: `ax.MinorGridLineStyle = '-.';`

`MinorGridColor` — Color of minor grid lines

`[0.1 0.1 0.1]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of minor grid lines, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The actual grid color depends on the values of the `MinorGridColorMode`, `ThetaColorMode`, and `RColorMode` properties. See `MinorGridColorMode` for more information.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `ax.MinorGridColor = [0 0 1]`

Example: `ax.MinorGridColor = 'blue'`

Example: `ax.MinorGridColor = '#0000FF'`

`MinorGridColorMode` — Property for setting minor grid color

`'auto'` (default) | `'manual'`

Property for setting the minor grid color, specified as one of these values:

- `'auto'` — Check the values of the `RColorMode` and `ThetaColorMode` properties to determine the grid line colors for the *r* and *theta* directions.
- `'manual'` — Use `MinorGridColor` to set the grid line color for all directions.

`MinorGridAlpha` — Minor grid line transparency

`0.25` (default) | value in the range `[0,1]`

Minor grid line transparency, specified as a value in the range `[0,1]`. A value of `1` means opaque and a value of `0` means completely transparent.

Example: `ax.MinorGridAlpha = 0.5`

MinorGridAlphaMode — Selection mode for MinorGridAlpha

'auto' (default) | 'manual'

Selection mode for the MinorGridAlpha property, specified as one of these values:

- 'auto' — Default transparency value of 0.25.
- 'manual' — Manually specify the transparency value. To specify the value, set the MinorGridAlpha property.

Example: ax.MinorGridAlphaMode = 'auto'

Labels

Title — Text object for axes title

text object (default)

Text object for axes title. To add a title, set the String property of the text object. To change the title appearance, such as the font style or color, set other properties. For a list, see Text.

```
ax = polaraxes;
ax.Title.String = 'My Title';
ax.Title.Color = 'red';
```

Alternatively, use the title to add a title and control the appearance.

```
title('My Title','Color','red')
```

Legend — Legend associated with the axes

empty GraphicsPlaceholder (default) | legend object

This property is read-only.

Legend associated with the axes, specified as a legend object. You can use this property to determine if the axes has a legend.

```
ax = gca;
lgd = ax.Legend
if ~isempty(lgd)
    disp('Legend Exists')
end
```

You also can use this property to access properties of an existing legend. For a list of properties, see Legend.

```
polarplot(1:10)
legend({'Line 1'},'FontSize',12)
ax = gca;
ax.Legend.TextColor = 'red';
```

Multiple Plots

`ColorOrder` — Color order

seven predefined colors (default) | three-column matrix of RGB triplets

Color order, specified as a three-column matrix of RGB triplets. This property defines the palette of colors MATLAB uses to create plot objects such as `Line`, `Scatter`, and `Bar` objects. Each row of the array is an RGB triplet. An RGB triplet is a three-element vector whose elements specify the intensities of the red, green, and blue components of a color. The intensities must be in the range [0, 1]. This table lists the default colors.

Colors	`ColorOrder` Matrix

	`[ 0 0.4470 0.7410`
	`0.8500 0.3250 0.0980`
	`0.9290 0.6940 0.1250`
	`0.4940 0.1840 0.5560`
	`0.4660 0.6740 0.1880`
	`0.3010 0.7450 0.9330`
	`0.6350 0.0780 0.1840]`

MATLAB assigns colors to objects according to their order of creation. For example, when plotting lines, the first line uses the first color, the second line uses the second color, and so on. If there are more lines than colors, then the cycle repeats.

Changing the Color Order Before or After Plotting

Starting in R2019b, you can change the color order in either of the following ways:

- Call the `colororder` function to change the color order for all the axes in a figure. The colors of existing plots in the figure update immediately. If you place additional axes into the figure, those axes also use the new color order. If you continue to call plotting commands, those commands also use the new colors.
- Set the `ColorOrder` property on the axes, call the `hold` function to set the axes hold state to `'on'`, and then call the desired plotting functions. This is like calling the

`colororder` function, but in this case you are setting the color order for the specific axes, not the entire figure. Setting the `hold` state to `'on'` is necessary to ensure that subsequent plotting commands do not reset the axes to use the default color order.

If you are using R2019a or an earlier release, changing the color order matrix does not affect existing plots. To change colors in an existing plot, you must set the `ColorOrder` property, and then set the axes hold state to `'on'` before calling any plotting functions.

The behavior of this property changed in R2019b. For more information, see:

- "Changing ColorOrder or LineStyleOrder on the axes affects existing plots immediately" on page 1-11671
- "Indexing scheme for ColorOrder and LineStyleOrder might change plot colors and line styles" on page 1-11671

`ColorOrderIndex` — Color order index

1 (default) | positive integer

Color order index, specified as a positive integer. This property specifies the next color MATLAB selects from the axes `ColorOrder` property when it creates the next plot object such as a `Line`, `Scatter`, or `Bar` object. For example, if the color order index value is `1`, then the next object added to the axes uses the first color in the `ColorOrder` matrix. If the index value exceeds the number of colors in the `ColorOrder` matrix, then the index value modulo of the number of colors in the `ColorOrder` matrix determines the color of the next object.

When the `NextPlot` property of the axes is set to `'add'`, then the color order index value increases every time you add a new plot to the axes. To start again with first color, set the `ColorOrderIndex` property to `1`.

The behavior of this property changed in R2019b. For more information, see "Indexing scheme for ColorOrder and LineStyleOrder might change plot colors and line styles" on page 1-11671.

`LineStyleOrder` — Line style order

`'-'` solid line (default) | character vector | cell array of character vectors | string array

Line style order, specified as a character vector, a cell array of character vectors, or a string array. This property lists the line styles that MATLAB uses to display multiple plot lines in the axes. MATLAB assigns styles to lines according to their order of creation. It changes to the next line style only after cycling through all the colors in the `ColorOrder`

property with the current line style. The default `LineStyleOrder` has only one line style, `'-'`.

To customize the line style order, create a cell array of character vectors or a string array. Specify each element of the array as a line specifier or marker specifier from the following tables. You can combine a line and a marker specifier into a single element, such as `'-*'`.

Line Specifier	Description
`'-'` (default)	Solid line
`'--'`	Dashed line
`':'`	Dotted line
`'-.'`	Dash-dotted line

Marker Specifier	Description
`'+'`	Plus sign markers
`'o'`	Circle markers
`'*'`	Star markers
`'.'`	Point markers
`'x'`	Cross markers
`'s'`	Square markers
`'d'`	Diamond markers
`'^'`	Upward-pointing triangle markers
`'v'`	Downward-pointing triangle markers
`'>'`	Right-pointing triangle markers
`'<'`	Left-pointing triangle markers
`'p'`	Five-pointed star (pentagram) markers
`'h'`	Six-pointed star (hexagram) markers

Changing Line Style Order Before or After Plotting

Starting in R2019b, you can change the line style order before or after plotting into the axes. When you set the `LineStyleOrder` property to a new value, MATLAB updates the

styles of any lines that are in the axes. If you continue plotting into the axes, your plotting commands continue using the line styles from the updated list.

If you are using R2019a or an earlier release, you must change the line style order before plotting. Set the value of the `LineStyleOrder` property, and then call the `hold` function to set the axes hold state to `'on'` before calling any plotting functions.

The behavior of this property changed in R2019b. For more information, see:

- "Changing ColorOrder or LineStyleOrder on the axes affects existing plots immediately" on page 1-11671
- "Indexing scheme for ColorOrder and LineStyleOrder might change plot colors and line styles" on page 1-11671

`LineStyleOrderIndex` — Line style order index

1 (default) | positive integer

Line style order index, specified as a positive integer. This property specifies the next line style MATLAB selects from the axes `LineStyleOrder` property to create the next plot line. For example, if this property is set to `1`, then the next plot line you add to the axes uses the first item in the `LineStyleOrder` property. If the index value exceeds the number of line styles in the `LineStyleOrder` array, then the index value modulo of the number of elements in the `LineStyleOrder` array determines the style of the next line.

When the `NextPlot` property of the axes is set to `'add'`, MATLAB increments the index value after cycling through all the colors in the `ColorOrder` property with the current line style. To start again with first line style, set the `LineStyleOrderIndex` property to `1`.

The behavior of this property changed in R2019b. For more information, see "Indexing scheme for ColorOrder and LineStyleOrder might change plot colors and line styles" on page 1-11671.

`NextPlot` — Properties to reset

`'replace'` (default) | `'add'` | `'replacechildren'` | `'replaceall'`

Properties to reset when adding a new plot to the axes, specified as one of these values:

- `'add'` — Add new plots to the existing axes. Do not delete existing plots or reset axes properties before displaying the new plot.
- `'replacechildren'` — Delete existing plots before displaying the new plot. Reset the `ColorOrderIndex` and `LineStyleOrderIndex` properties to `1`, but do not reset

other axes properties. The next plot added to the axes uses the first color and line style based on the `ColorOrder` and `LineStyle` order properties. This value is similar to using `cla` before every new plot.

- `'replace'` — Delete existing plots and reset axes properties, except `Position` and `Units`, to their default values before displaying the new plot.
- `'replaceall'` — Delete existing plots and reset axes properties, except `Position` and `Units`, to their default values before displaying the new plot. This value is similar to using `cla reset` before every new plot.

Note For `Axes` objects with only one *y*-axis, the `'replace'` and `'replaceall'` property values are equivalent. For `Axes` objects with two *y*-axes, the `'replace'` value affects only the active side while the `'replaceall'` value affects both sides.

Figures also have a `NextPlot` property. Alternatively, you can use the `newplot` function to prepare figures and axes for subsequent graphics commands.

`SortMethod` — Order for rendering objects

`'depth'` | `'childorder'`

Order for rendering objects, specified as one of these values:

- `'depth'` — Draw objects in back-to-front order based on the current view. Use this value to ensure that objects in front of other objects are drawn correctly.
- `'childorder'` — Draw objects in the order in which they are created by graphics functions, without considering the relationship of the objects in three dimensions. This value can result in faster rendering, particularly if the figure is very large, but also can result in improper depth sorting of the objects displayed.

Color and Transparency Maps

`Colormap` — Color map

parula (default) | `m`-by-`3` array of RGB triplets

Color map, specified as an `m`-by-`3` array of RGB (red, green, blue) triplets that define `m` individual colors.

Example: `ax.Colormap = [1 0 1; 0 0 1; 1 1 0]` sets the color map to three colors: magenta, blue, and yellow.

MATLAB accesses these colors by their row number.

Alternatively, use the `colormap` function to change the color map.

`ColorScale` — Scale for color mapping
`'linear'` (default) | `'log'`

Scale for color mapping, specified as one of these values:

- `'linear'` — Linear scale. The tick values along the colorbar also use a linear scale.
- `'log'` — Log scale. The tick values along the colorbar also use a log scale.

Example: `ax.ColorScale = 'log'`

`CLim` — Color limits for colormap
`[0 1]` (default) | two-element vector of the form `[cmin cmax]`

Color limits for the colormap, specified as a two-element vector of the form `[cmin cmax]`.

If the associated mode property is set to `'auto'`, then MATLAB chooses the color limits. If you assign a value to this property, then MATLAB sets the mode to `'manual'` and does not automatically choose the color limits.

`CLimMode` — Selection mode for `CLim`
`'auto'` (default) | `'manual'`

Selection mode for the `CLim` property, specified as one of these values:

- `'auto'` — Automatically select the limits based on the color data of the graphics objects contained in the axes.
- `'manual'` — Manually specify the values. To specify the values, set the `CLim` property. The values do not change when the limits of the axes children change.

`Alphamap` — Transparency map
array of 64 values from `0` to `1` (default) | array of finite alpha values from `0` to `1`

Transparency map, specified as an array of finite alpha values that progress linearly from `0` to `1`. The size of the array can be m-by-1 or 1-by-m. MATLAB accesses alpha values by their index in the array. Alphamaps can be any length.

`AlphaScale` — Scale for transparency mapping
`'linear'` (default) | `'log'`

Scale for transparency mapping, specified as one of these values:

- `'linear'` — Linear scale
- `'log'` — Log scale

Example: `ax.AlphaScale = 'log'`

`ALim` — Alpha limits for alphamap

`[0 1]` (default) | two-element vector of the form `[amin amax]`

Alpha limits for alphamap, specified as a two-element vector of the form `[amin amax]`.

If the associated mode property is set to `'auto'`, then MATLAB chooses the alpha limits. If you set this property, then MATLAB sets the mode to `'manual'` and does not automatically choose the alpha limits.

`ALimMode` — Selection mode for `ALim`

`'auto'` (default) | `'manual'`

Selection mode for the `ALim` property, specified as one of these values:

- `'auto'` — Automatically select the limits based on the `AlphaData` values of the graphics objects contained in the axes.
- `'manual'` — Manually specify the alpha limits. To specify the alpha limits, set the `ALim` property.

Box Styling

`Color` — Background color

`[1 1 1]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `ax.Color = 'none'`

`LineWidth` — Width of circular and angle lines

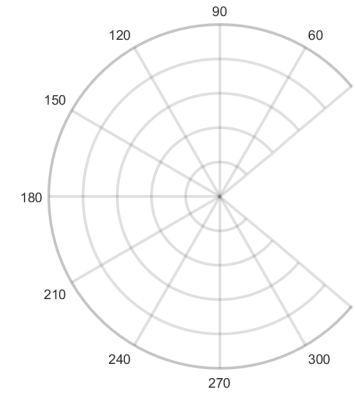
`0.5` (default) | scalar value

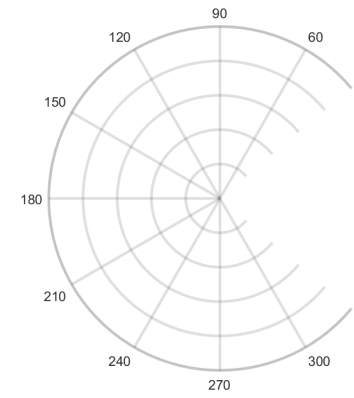
Width of circular and angle lines, specified as a scalar value in point units. One point equals 1/72 inch.

Example: `ax.LineWidth = 1.5`

Box — Outline around polar axes
`'off'` (default) | `'on'`

Outline around the polar axes, specified as either `'on'` or `'off'`. The difference between the values is most noticeable when the *theta*-axis limits do not span 360 degrees.

Value	Result
`'on'`	Display the full outline around the polar axes. 

Value	Result
`'off'`	Do not display the full outline around the polar axes. 

Example: `ax.Box = 'on'`

`Clipping` — Clipping of objects to polar axes boundary
`'on'` (default) | `'off'`

Clipping of objects to the polar axes boundary, specified as either `'on'` or `'off'`. The clipping behavior of an object in the polar axes depends on both the `Clipping` property of the polar axes and the `Clipping` property of the individual object. The property value of the polar axes has these effects:

- `'on'` — Allow each individual object in the polar axes to control its own clipping behavior based on the `Clipping` property value for the object.
- `'off'` — Disable clipping for all objects in the polar axes, regardless of the `Clipping` property value for the individual objects. Parts of objects can appear outside of the polar axes limits. For example, parts can appear outside the limits if you create a plot, set `hold on`, freeze the axis scaling, and then add a plot that is larger than the original plot.

This table lists the results for different combinations of `Clipping` property values.

Clipping Property for Axes Object	Clipping Property for Individual Object	Result
`'on'`	`'on'`	Individual object is clipped. Others might or might not be.
`'on'`	`'off'`	Individual object is not clipped. Others might or might not be.
`'off'`	`'on'`	Individual object and other objects are not clipped.
`'off'`	`'off'`	Individual object and other objects are not clipped.

Thick lines and markers might display outside the polar axes limits, even if clipping is enabled. If a plot contains markers, then as long as the data point lies within the polar axes, MATLAB draws the entire marker.

Position

`OuterPosition` — Size and position of polar axes, including labels and margins

`[0 0 1 1]` (default) | four-element vector

Size and position of polar axes, including the labels and margins, specified as a four-element vector of the form `[left bottom width height]`. This vector defines the extents of the rectangle that encloses the outer bounds of the polar axes. The `left` and `bottom` elements define the distance from the lower-left corner of the figure or uipanel that contains the polar axes to the lower-left corner of the rectangle. The `width` and `height` elements are the rectangle dimensions.

By default, the values are measured in units normalized to the container. To change the units, set the `Units` property. The default value of `[0 0 1 1]` includes the whole interior of the container.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`Position` — Size and position of polar axes, not including labels or margins

`[0.1300 0.1100 0.7750 0.8150]` (default) | four-element vector

Size and position of the polar axes, not including labels or margins, specified as a four-element vector of the form `[left bottom width height]`. This vector defines the extents of the tightest bounding rectangle that encloses the polar axes. The `left` and `bottom` elements define the distance from the lower-left corner of the container to the lower-left corner of the rectangle. The `width` and `height` elements are the rectangle dimensions.

By default, the values are measured in units normalized to the container. To change the units, set the `Units` property.

Example: `ax.Position = [0 0 1 1]`

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`TightInset` — Margins for text labels
four-element vector of the form `[left bottom right top]`

This property is read-only.

Margins for the text labels, returned as a four-element vector of the form `[left bottom right top]`. The elements define the distances between the bounds of the `Position` property and the extent of the polar axes text labels and title. By default, the values are measured in units normalized to the figure or uipanel that contains the polar axes. To change the units, set the `Units` property.

The `Position` property and the `TightInset` property define the tightest bounding box that encloses the polar axes and its labels and title.

`ActivePositionProperty` — Active position property
`'outerposition'` | `'position'`

Position property to hold constant when adding, removing, or changing decorations, specified as one of the following values:

- `'outerposition'` — The `OuterPosition` property remains constant when you add, remove, or change decorations such as a title or an axis label. If any positional adjustments are needed, MATLAB adjusts the `Position` property.
- `'position'` — The `Position` property remains constant when you add, remove, or change decorations such as a title or an axis label. If any positional adjustments are needed, MATLAB adjusts the `OuterPosition` property.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`Units` — Position units

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'points'` | `'pixels'` | `'characters'`

Position units, specified as one of these values.

Units	Description
`'normalized'` (default)	Normalized with respect to the container, which is typically the figure or a panel. The lower left corner of the container maps to `(0,0)` and the upper right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the default uicontrol font of the graphics root object: • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.
`'points'`	Typography points. One point equals 1/72 inch.

Units	Description
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems. • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. • On Linux systems, the size of a pixel is determined by your system resolution.

When specifying the units as a `Name,Value` pair during object creation, you must set the `Units` property before specifying the properties that you want to use these units, such as `Position`.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `TiledChartLayoutOptions` object

Layout options, specified as a `TiledChartLayoutOptions` object. This property specifies options when the axes is a child of a tiled chart layout. Use this property to position the axes within the layout by setting the `Tile` and `TileSpan` properties on the `TiledChartLayoutOptions` object.

For example, this code places axes `ax` in the third tile of a tiled chart layout.

```
ax.Layout.Tile = 3;
```

To make the axes span multiple tiles, specify the `TileSpan` property as a two-element vector. For example, this axes spans `2` rows and `3` columns of tiles.

```
ax.Layout.TileSpan = [2 3];
```

If the axes is not a child of a tiled chart layout (for example, if it is a child of a figure or panel), then this property is empty and has no effect.

Interactivity

`Toolbar` — Data exploration toolbar
`AxesToolbar` object (default)

Data exploration toolbar, which is an `AxesToolbar` object. The toolbar appears at the top-right corner of the axes when you hover over it and includes options for exporting and data tips.

You can customize the toolbar buttons using the `axtoolbar` and `axtoolbarbtn` functions.

If you do not want the toolbar to appear when you hover over the axes, set the `Visible` property of the `AxesToolbar` object to `'off'`.

```
ax = gca;
ax.Toolbar.Visible = 'off';
```

For more information, see AxesToolbar.

`Interactions` — Interactions
`DataTipInteraction` object (default) | `[]`

Interactions, specified as a `DataTipInteraction` object or an empty array. When the value of this property is a `DataTipInteraction` object, you can display data tips within your chart without selecting any of the axes toolbar buttons.

To remove all interactions from the axes, set this property to an empty array. To temporarily disable the current set of interactions, call the `disableDefaultInteractivity` function. You can reenable them by calling the `enableDefaultInteractivity` function.

Note

- Setting this property is not supported in the Live Editor.
- The `DataTipInteraction` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

For more information about chart interactions, see “Control Chart Interactivity”.

`Visible` — State of visibility
`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`UIContextMenu` — Context menu
empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state
`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles
`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

Callbacks

`ButtonDownFcn` — Mouse-click callback
`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see "Callback Definition".

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks only when visible. The `Visible` property must be set to `'on'`. The `HitTest` property determines if the `PolarAxes` object responds to the click or if an ancestor does.
- `'all'` — Capture mouse clicks regardless of visibility. The `Visible` property can be set to `'on'` or `'off'`. The `HitTest` property determines if the `PolarAxes` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `PolarAxes` object passes the click to the object below it in the current view of the figure window, which is typically the axes or the figure. The `HitTest` property has no effect.

If you want an object to be clickable when it is underneath other objects that you do not want to be clickable, then set the `PickableParts` property of the other objects to `'none'` so that the click passes through them.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `PolarAxes` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `PolarAxes` object that has one of these:

- `HitTest` property set to `'on'`
- `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

Note The `PickableParts` property determines if the `PolarAxes` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container

`Figure` object | `Panel` object | `Tab` object | `TiledChartLayout` object

Parent container, specified as a `Figure`, `Panel`, `Tab`, or `TiledChartLayout` object.

`Children` — Children

empty `GraphicsPlaceholder` array | array of graphics objects

Children, returned as an array of graphics objects. Use this property to view a list of the children or to reorder the children by setting the property to a permutation of itself.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the child graphics object to the `PolarAxes` object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object
`'polaraxes'` (default)

This property is read-only.

Type of graphics object, returned as `'polaraxes'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

Compatibility Considerations

Changing ColorOrder or LineStyleOrder on the axes affects existing plots immediately

Behavior changed in R2019b

Starting in R2019b, if you change the axes `ColorOrder` or `LineStyleOrder` properties after plotting into the axes, the colors and line styles in your plot update immediately. In R2019a and previous releases, the new colors and line styles affect only subsequent plots, not the existing plots.

To preserve the original behavior, set the axes `ColorOrderIndex` or `LineStyleOrderIndex` property to any value (such as its current value) before changing the `ColorOrder` or `LineStyleOrder` property.

Indexing scheme for ColorOrder and LineStyleOrder might change plot colors and line styles

Behavior changed in R2019b

Starting in R2019b, there is a new indexing scheme that enables you to change the colors and line styles of existing plots by setting the `ColorOrder` or `LineStyleOrder` properties. MATLAB applies this indexing scheme to all objects that have a `ColorMode`, `FaceColorMode`, `MarkerFaceColorMode`, or `CDataMode`. As a result, your code might produce plots that cycle though the colors and line styles differently than in previous releases.

In R2019a and earlier releases, MATLAB uses a different indexing scheme which does not allow you to change the colors of existing plots.

To preserve the way your plots cycle through colors and line styles, set the axes `ColorOrderIndex` or `LineStyleOrderIndex` property to any value (such as its current value) before plotting into the axes.

See Also

`cla` | `gca` | `polaraxes`

Topics

"Access Property Values"
"Graphics Object Properties"

Introduced before R2006a

polarhistogram

Histogram chart in polar coordinates

Syntax

```
polarhistogram(theta)
polarhistogram(theta,nbins)
polarhistogram(theta,edges)
polarhistogram('BinEdges',edges,'BinCounts',counts)

polarhistogram( ___ ,Name,Value)
polarhistogram(pax, ___ )
h = polarhistogram( ___ )
```

Description

`polarhistogram(theta)` creates a histogram plot in polar coordinates by sorting the values in `theta` into equally spaced bins. Specify the values in radians.

`polarhistogram(theta,nbins)` uses the number of bins specified by the positive integer, `nbins`.

`polarhistogram(theta,edges)` sorts `theta` into bins with bin edges specified by the vector, `edges`. All bins include the left edge, but only the last bin includes the right edge. In other words, the last bin includes both edges.

`polarhistogram('BinEdges',edges,'BinCounts',counts)` uses the manually specified bin edges and associated bin counts. The `polarhistogram` function does not do any data binning.

`polarhistogram( ___ ,Name,Value)` specifies additional options using one or more name-value pair arguments. For example, you can use semitransparent bars by specifying `'FaceAlpha'` and a scalar value between 0 and 1.

`polarhistogram(pax, ___ )` plots into the polar axes specified by `pax` instead of into the current axes.

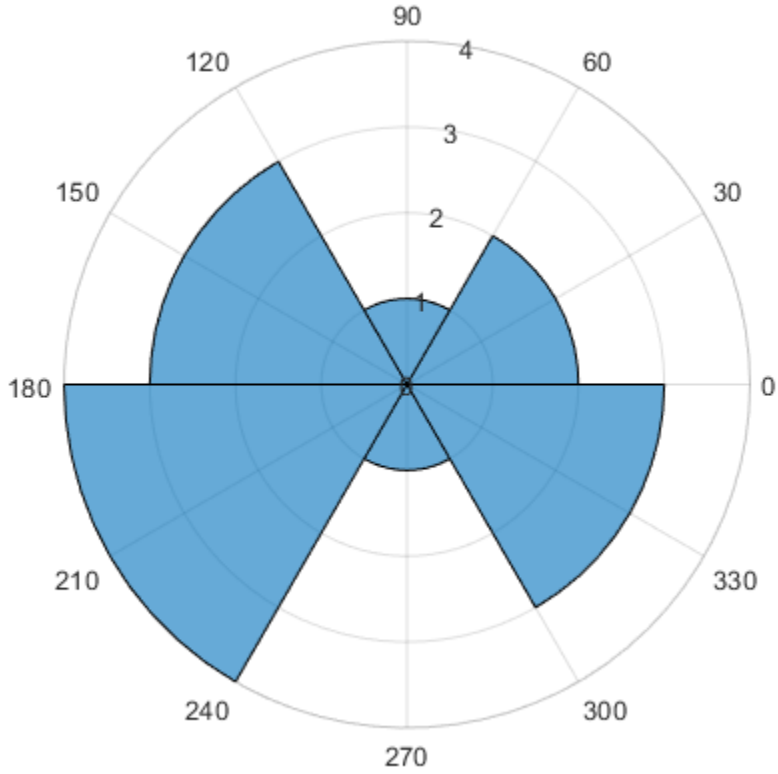
`h = polarhistogram( ___ )` returns the `Histogram` object. Use `h` to modify the histogram after it is created. For a list of properties, see Histogram.

Examples

Create Histogram Chart in Polar Coordinates

Create a vector of values between 0 and 2π. Create a histogram chart that shows the data sorted into six bins.

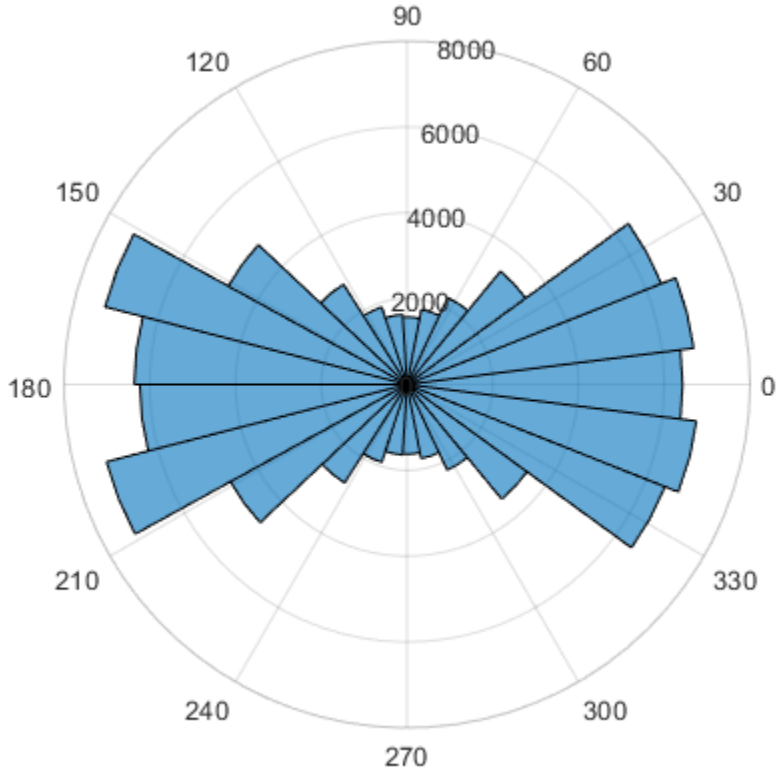
```
theta = [0.1 1.1 5.4 3.4 2.3 4.5 3.2 3.4 5.6 2.3 2.1 3.5 0.6 6.1];
polarhistogram(theta,6)
```



Specify Number of Bins for Polar Histogram Chart

Create a histogram plot from 100,000 values between $-\pi$ and π, and sort the data into 25 bins.

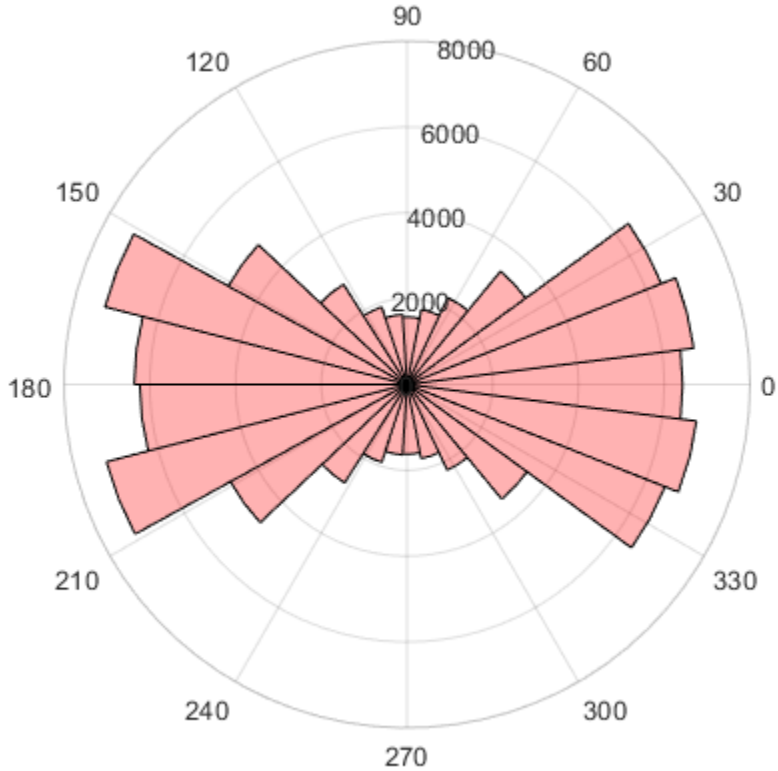
```
theta = atan2(rand(100000,1)-0.5,2*(rand(100000,1)-0.5));
polarhistogram(theta,25);
```



Modify Appearance of Histogram Chart

Create a histogram chart in polar coordinates, and then change its appearance. Specify the bar colors by setting the `FaceColor` property to a character vector of a color name, such as `'red'`, or an RGB triplet. Specify the transparency by setting the `FaceAlpha` property to a value between 0 and 1.

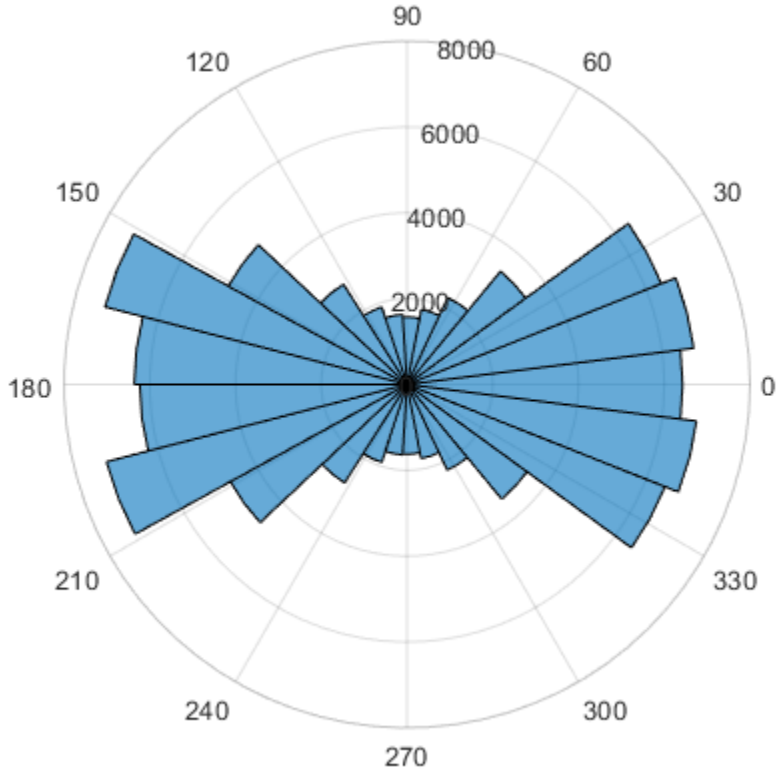
```
theta = atan2(rand(100000,1)-0.5,2*(rand(100000,1)-0.5));
polarhistogram(theta,25,'FaceColor','red','FaceAlpha',.3);
```



Modify Appearance of Histogram Chart After Creation

Create a histogram chart in polar coordinates. Assign the histogram object to the variable `h`.

```
theta = atan2(rand(100000,1)-0.5,2*(rand(100000,1)-0.5));
h = polarhistogram(theta,25)
```

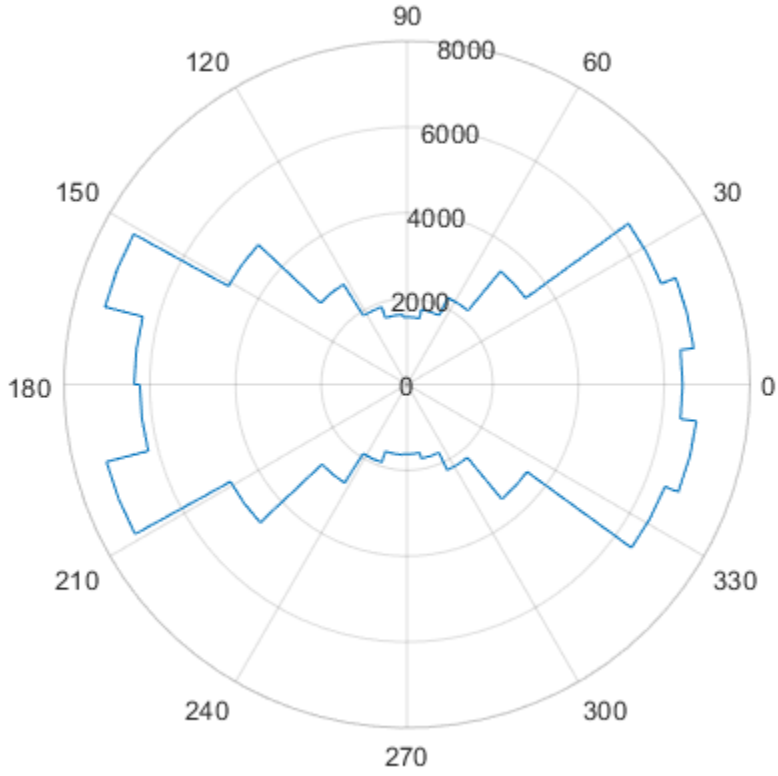



```
h =
  Histogram with properties:

            Data: [100000x1 double]
          Values: [1x25 double]
         NumBins: 25
        BinEdges: [1x26 double]
        BinWidth: 0.2513
       BinLimits: [-3.1416 3.1416]
    Normalization: 'count'
       FaceColor: 'auto'
       EdgeColor: [0 0 0]
```

```
Show all properties
```

Use `h` to access and modify properties of the histogram object after it is created. For example, show just the histogram outline by setting the `DisplayStyle` property of the histogram object.

```
h.DisplayStyle = 'stairs';
```



Input Arguments

theta — Data to distribute among bins
vector | matrix

Data to distribute among bins, specified as a vector or a matrix. `polarhistogram` creates one histogram, regardless of whether you specify a vector or a matrix. Specify the values in radians. To convert degrees to radians, use `deg2rad`.

Values that correspond to the same angle direction differ by exactly 2π, and are sorted into the same bin. `polarhistogram` does not include `NaN`, `Inf`, and `-Inf` values in any bin.

Example: `theta = [0 0.4 0.5 0.7 2.3 3.0 1.7 0.3];`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

nbins — Number of bins
positive integer

Number of bins, specified as a positive integer. If you do not specify `nbins`, then `polarhistogram` automatically calculates how many bins to use based on the values in `theta`.

Example: `nbins = 15;`

edges — Bin edges
vector

Bin edges, specified as a vector. The difference between the minimum and maximum edge values must be less than or equal to 2π.

Example: `polarhistogram('BinEdges',[0 pi/3 pi 3*pi/2 2*pi],'BinCounts',[5 3 4 6])`

counts — Bin counts
vector

Bin counts, specified as a vector. Use this option if you perform the bin counts calculation separately and you do not want `polarhistogram` to do any data binning.

Example: `polarhistogram('BinEdges',[0 pi/3 pi 3*pi/2 2*pi],'BinCounts',[5 3 4 6])`

pax — PolarAxes object
PolarAxes object

PolarAxes object. If you do not specify the polar axes, then polarhistogram uses the current axes. polarhistogram does not support plotting into Cartesian axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

The histogram properties listed here are only a subset. For a complete list, see Histogram.

Example: polarhistogram(theta,'FaceAlpha',0.3) creates a histogram chart with semitransparent bars.

BinLimits — Bin limits
two-element vector

Bin limits, specified as a two-element vector of the form [bmin,bmax], where bmin is less than bmax.

This option plots a histogram using the input array values that fall between bmin and bmax inclusive. That is, theta(theta>=bmin & theta<=bmax).

Example: polarhistogram(theta,'BinLimits',[-pi/2 pi/2]) plots a histogram using only the values in theta that are between -pi/2 and pi/2 inclusive.

BinWidth — Width across top of bins
scalar less than 2π

Width across the top of the bins, specified as a scalar less than 2π.

polarhistogram uses a maximum of 65,536 bins (or 2^{16}). If the specified bin width requires more bins, then polarhistogram uses the maximum number of bins and adjust the bin width accordingly.

Example: polarhistogram(theta,'BinWidth',pi) uses bins with a width of π.

`Normalization` — Type of normalization
`'count'` (default) | `'probability'` | `'countdensity'` | `'pdf'` | `'cumcount'` | `'cdf'`

Type of normalization, specified as one of the values in this table.

Value	Description
`'count'`	Default normalization scheme. The height of each bar is the number of observations in each bin. The sum of the bar heights is `numel(theta)`.
`'probability'`	The height of each bar is the relative number of observations, (number of observations in bin/total number of observations). The sum of the bar heights is 1.
`'countdensity'`	The height of each bar is the number of observations in bin/width of bin.
`'pdf'`	Probability density function estimate. The height of each bar is, (number of observations in the bin)/(total number of observations * width of bin). The area of each bar is the relative number of observations. The sum of the bar areas is 1.
`'cumcount'`	The height of each bar is the cumulative number of observations in each bin and all previous bins. The height of the last bar is `numel(theta)`.
`'cdf'`	Cumulative density function estimate. The height of each bar is equal to the cumulative relative number of observations in the bin and all previous bins. The height of the last bar is 1.

Example: `polarhistogram(theta,'Normalization','pdf')` plots an estimate of the probability density function for `theta`.

`DisplayStyle` — Histogram display style
`'bar'` (default) | `'stairs'`

Histogram display style, specified as one of these values:

- `'stairs'` — Display the histogram outline only.
- `'bar'` — Show each individual bar with a filled interior.

Example: `polarhistogram(theta,'DisplayStyle','stairs')` plots the outline of the histogram.

`FaceAlpha` — Transparency of histogram bars

`0.6` (default) | scalar value between `0` and `1` inclusive

Transparency of histogram bars, specified as a scalar value between `0` and `1` inclusive. `polarhistogram` uses the same transparency for all the bars of the histogram. A value of `1` means fully opaque and `0` means completely transparent (invisible).

Example: `polarhistogram(theta,'FaceAlpha',.5)` creates a histogram plot with semi-transparent bars.

`FaceColor` — Histogram bar color

`'auto'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Histogram bar color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'auto'` lets the histogram choose the color.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

If you specify `DisplayStyle` as `'stairs'`, then `polarhistogram` does not use the `FaceColor` property.

Example: `polarhistogram(theta,'FaceColor','g')` creates a histogram plot with green bars.

Limitations

- `polarhistogram` does not support creating histograms of categorical data in polar axes.

See Also

Functions
`histogram` | `polarplot` | `polarscatter`

Properties
Histogram | PolarAxes

Introduced in R2016b

polarplot

Plot line in polar coordinates

Syntax

```
polarplot(theta,rho)
polarplot(theta,rho,LineSpec)
polarplot(theta1,rho1,...,thetaN,rhoN)
polarplot(theta1,rho1,LineSpec1,...,thetaN,rhoN,LineSpecN)
```

```
polarplot(rho)
polarplot(rho,LineSpec)
```

```
polarplot(Z)
polarplot(Z,LineSpec)
```

```
polarplot( ___ ,Name,Value)
polarplot(pax, ___ )
p = polarplot( ___ )
```

Description

`polarplot(theta,rho)` plots a line in polar coordinates, with `theta` indicating the angle in radians and `rho` indicating the radius value for each point. The inputs must be vectors with equal length or matrices with equal size. If the inputs are matrices, then `polarplot` plots columns of `rho` versus columns of `theta`. Alternatively, one of the inputs can be a vector and the other a matrix as long as the vector is the same length as one dimension of the matrix.

`polarplot(theta,rho,LineSpec)` sets the line style, marker symbol, and color for the line.

`polarplot(theta1,rho1,...,thetaN,rhoN)` plots multiple `rho,theta` pairs.

`polarplot(theta1,rho1,LineSpec1,...,thetaN,rhoN,LineSpecN)` specifies the line style, marker symbol, and color for each line.

`polarplot(rho)` plots the radius values in `rho` at evenly spaced angles between 0 and 2π.

`polarplot(rho,LineSpec)` sets the line style, marker symbol, and color for the line.

`polarplot(Z)` plots the complex values in `Z`.

`polarplot(Z,LineSpec)` sets the line style, marker symbol, and color for the line.

`polarplot( ___ ,Name,Value)` specifies properties of the chart line using one or more `Name,Value` pair arguments. The property settings apply to all the lines. You cannot specify different property values for different lines using `Name,Value` pairs.

`polarplot(pax, ___ )` uses the `PolarAxes` object specified by `pax`, instead of the current axes.

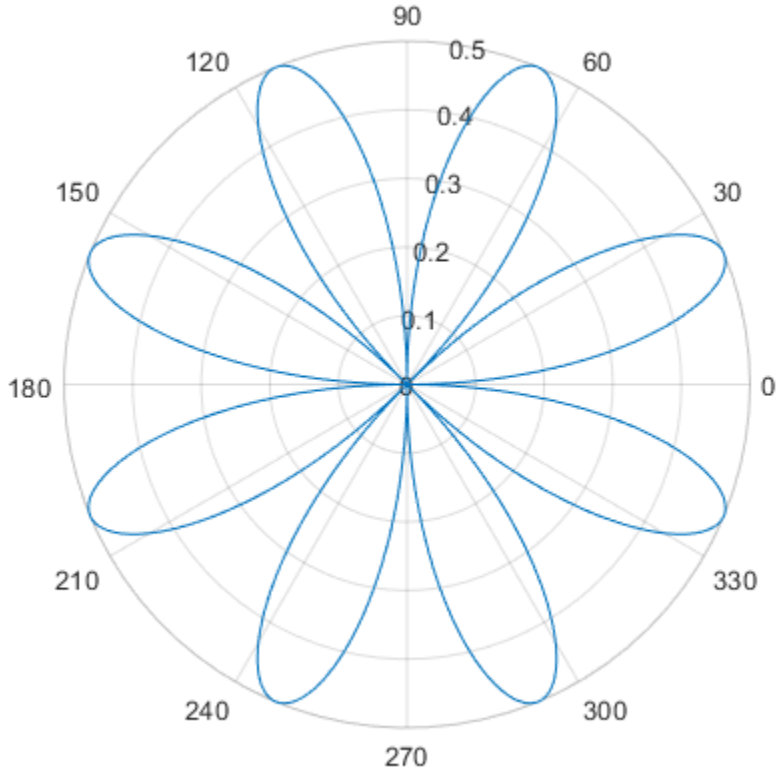
`p = polarplot( ___ )` returns one or more chart line objects. Use `p` to set properties of a specific chart line object after it is created. For a list of properties, see Chart Line.

Examples

Create Polar Plot

Plot a line in polar coordinates.

```
theta = 0:0.01:2*pi;
rho = sin(2*theta).*cos(2*theta);
polarplot(theta,rho)
```



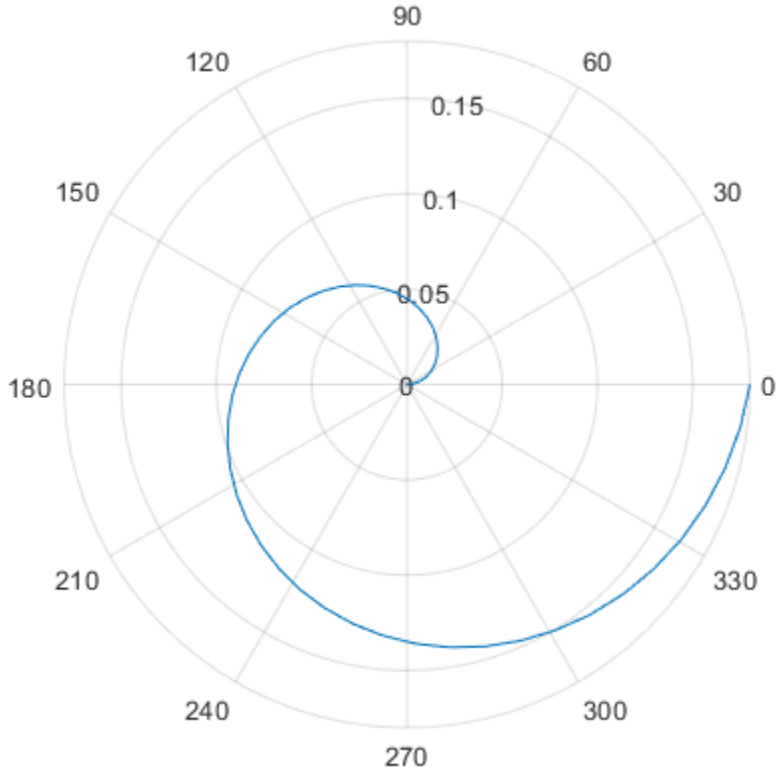
Convert from Degrees to Radians Before Plotting

Create the data to plot.

```
theta = linspace(0,360,50);
rho = 0.005*theta/10;
```

Convert the values in `theta` from degrees to radians. Then, plot the data in polar coordinates.

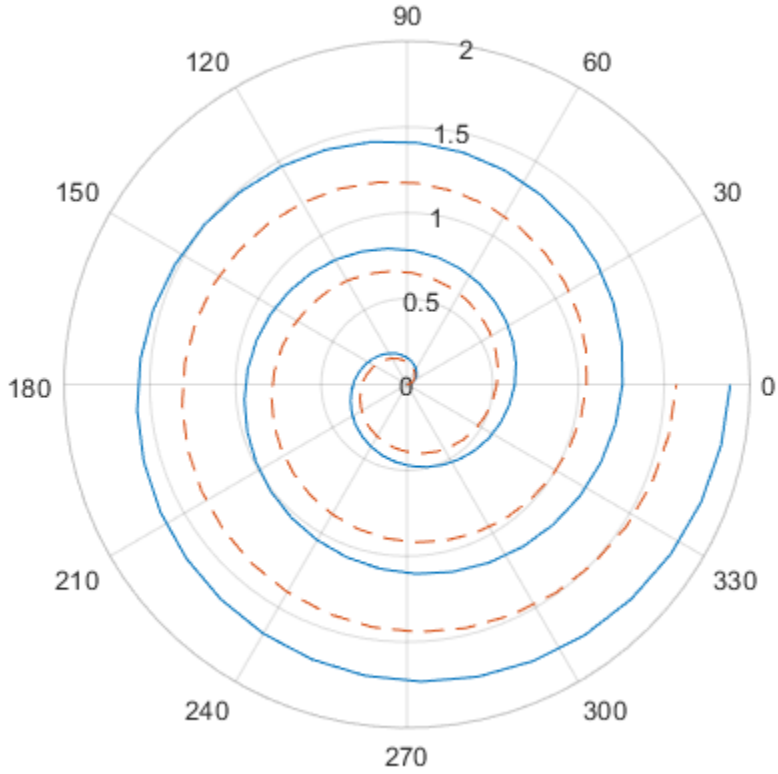
```
theta_radians = deg2rad(theta);
polarplot(theta_radians,rho)
```



Plot Multiple Lines in Polar Coordinates

Plot two lines in polar coordinates. Use a dashed line for the second line.

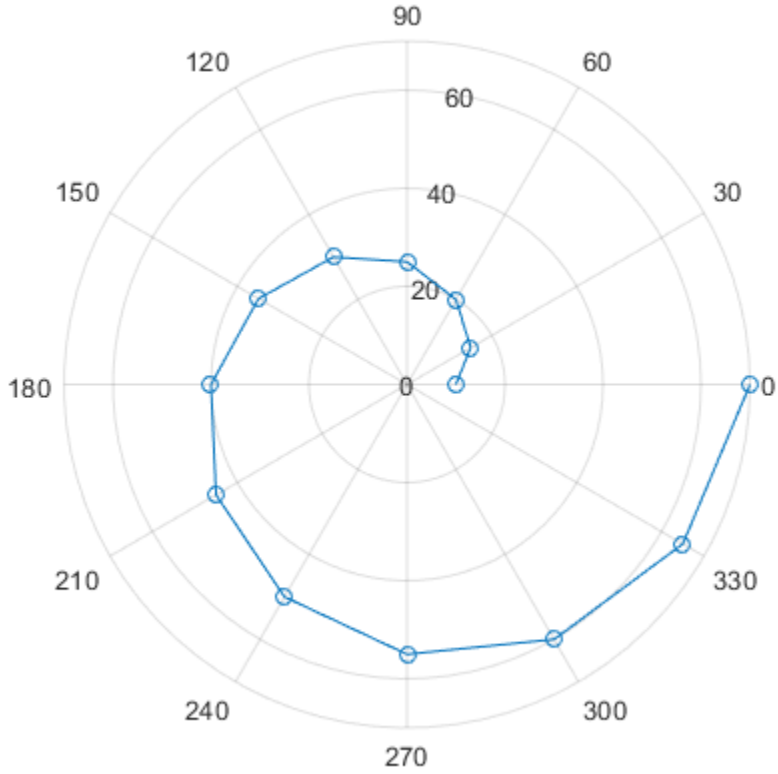
```
theta = linspace(0,6*pi);
rho1 = theta/10;
polarplot(theta,rho1)

rho2 = theta/12;
hold on
polarplot(theta,rho2,'--')
hold off
```



Plot Radius Values at Equally Spaced Angles

Specify only the radius values, without specifying the angle values. `polarplot` plots the radius values at equally spaced angles that span from 0 to 2π. Display a circle marker at each data point.

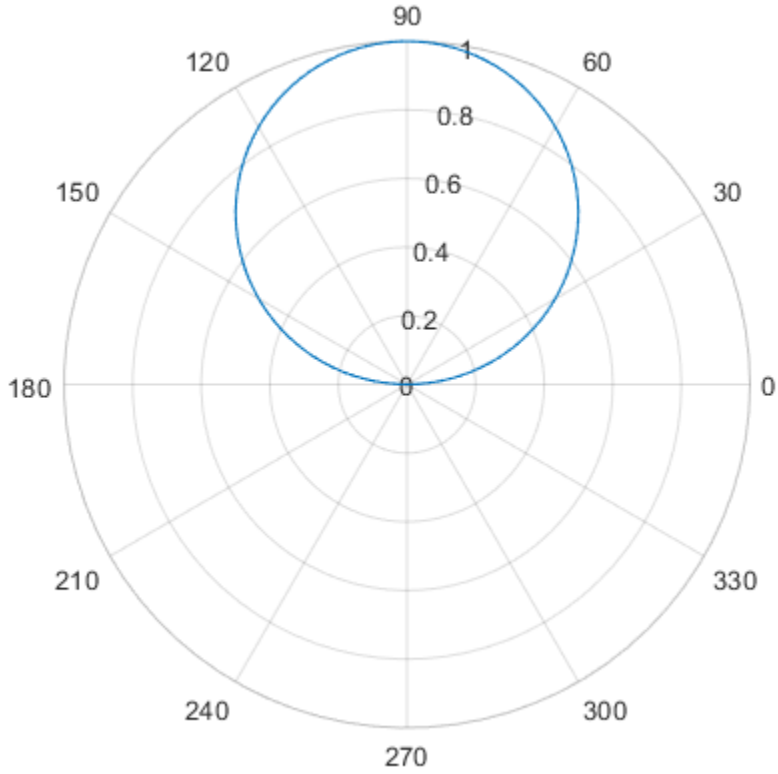
```
rho = 10:5:70;
polarplot(rho,'-o')
```



Plot Negative Radius Values

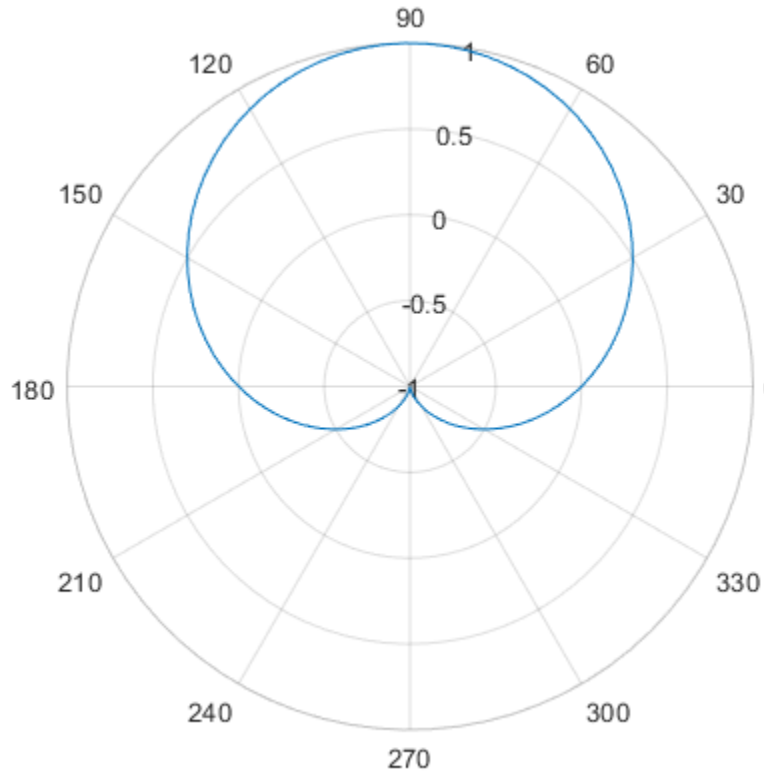
Create a polar plot using negative radius values. By default, `polarplot` reflects negative values through the origin.

```
theta = linspace(0,2*pi);
rho = sin(theta);
polarplot(theta,rho)
```



Change the limits of the *r*-axis so it ranges from -1 to 1.

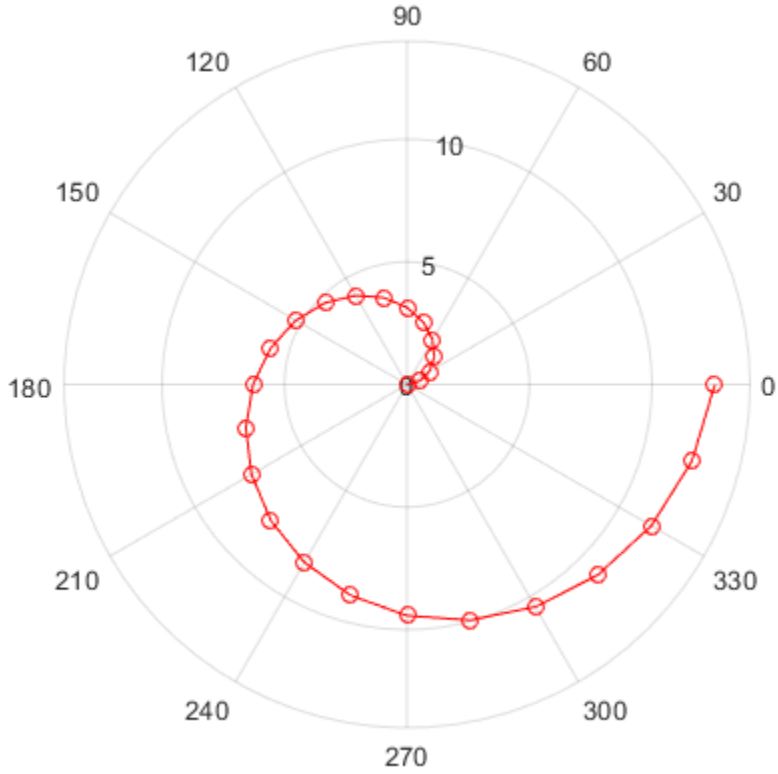
```
rlim([-1 1])
```



Specify Line Color for Polar Plot

Create a polar plot using a red line with circle markers.

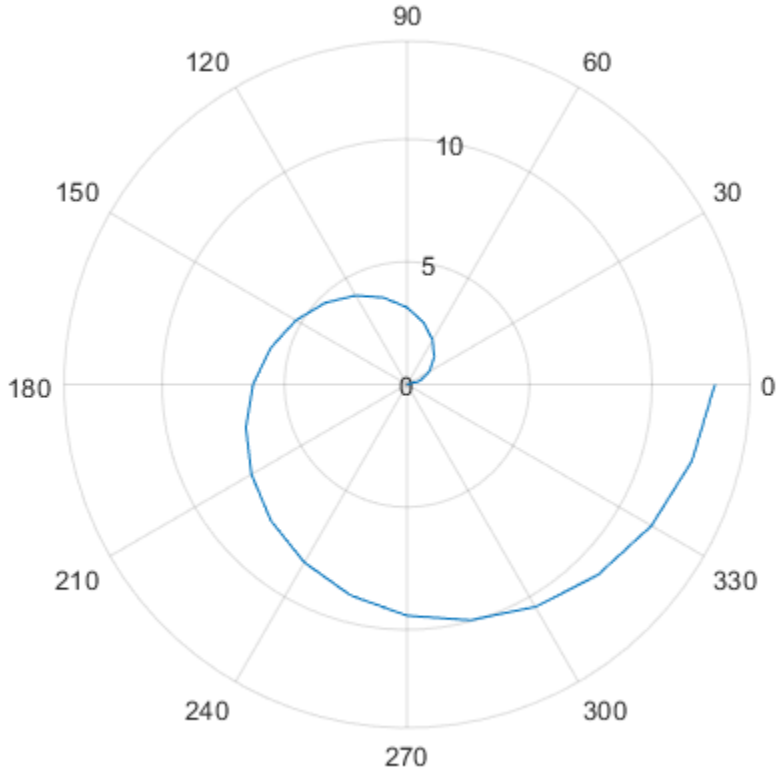
```
theta = linspace(0,2*pi,25);
rho = 2*theta;
polarplot(theta,rho,'r-o')
```



Specify Line Color After Creation

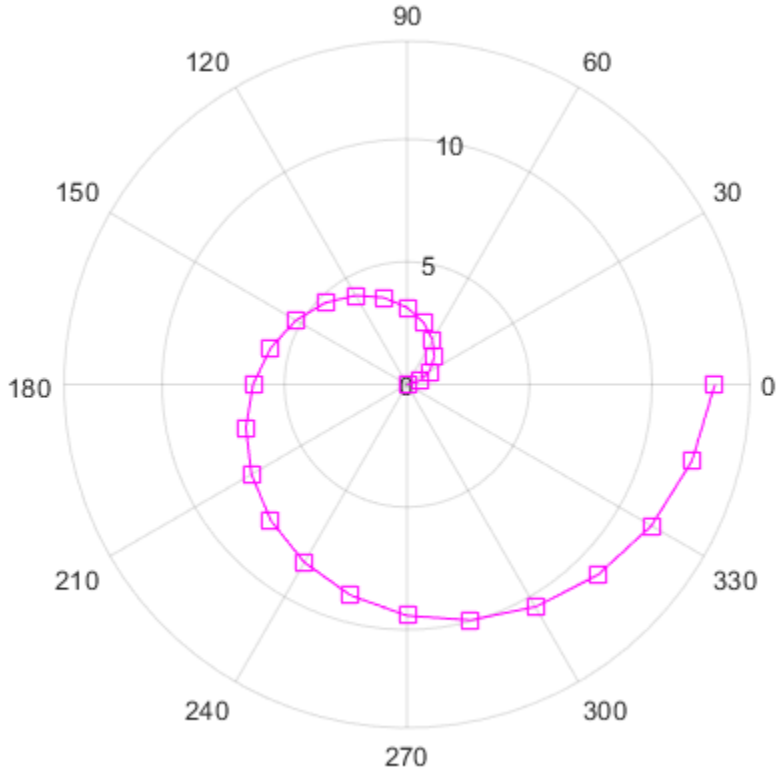
Create a polar plot and return the chart line object.

```
theta = linspace(0,2*pi,25);
rho = 2*theta;
p = polarplot(theta,rho);
```



Change the line color and width and add markers.

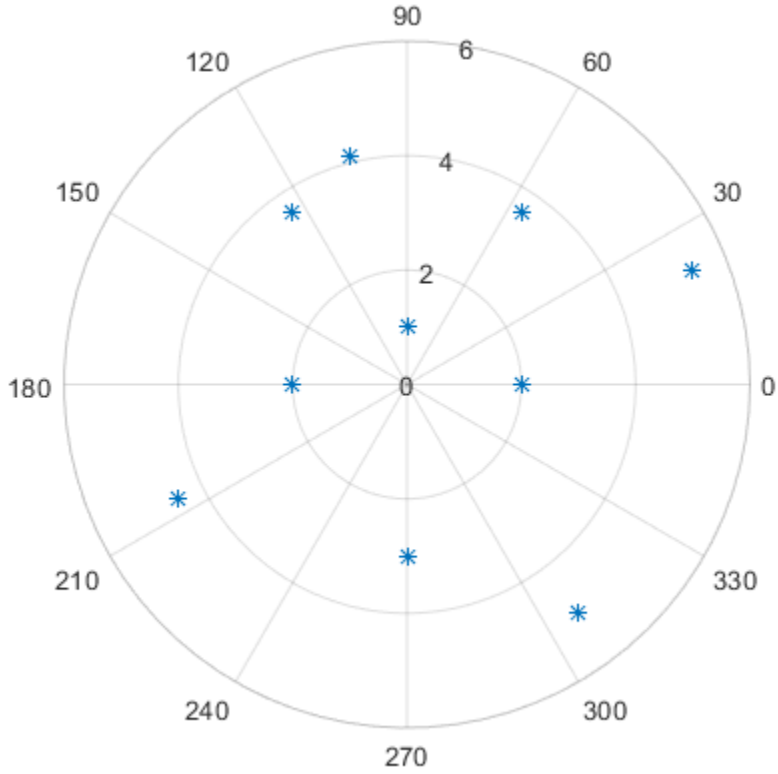
```
p.Color = 'magenta';
p.Marker = 'square';
p.MarkerSize = 8;
```



Create Polar Plot with Complex Values

Plot complex values in polar coordinates. Display markers at each point without a line connecting them.

```
Z = [2+3i 2 -1+4i 3-4i 5+2i -4-2i -2+3i -2 -3i 3i-2i];
polarplot(Z,'*')
```



Input Arguments

`theta` — Angle values
vector | matrix

Angle values, specified as a vector or matrix. Specify the values in radians. To convert data from degrees to radians, use `deg2rad`.

To change the limits of the theta-axis, use `thetalim`.

Example: `[0 pi/2 pi 3*pi/2 2*pi]`

`rho` — Radius values
vector | matrix

Radius values, specified as a vector or matrix. By default, negative values are reflected through 0. A point is reflected by taking the absolute value of its radius, and adding 180 degrees to its angle.

To change the limits of the *r*-axis, use `rlim`.

Example: `[1 2 3 4 5]`

`Z` — Complex values
vector | matrix

Complex values, specified as a vector or matrix where each element is of the form `rho*`$e^{i*theta}$, or `x+iy`, where:

- `rho = sqrt(x^2+y^2)`
- `theta = atan(y/x)`

Example: `[1+2i 3+4i 3i]`

`LineSpec` — Line specification
character vector | string

Line specification, specified as a character vector or string containing a line style, marker, and color specifier. The elements can appear in any order, and you can omit one or more options. If you omit the line style and specify the marker, then the plot shows only markers with no line connecting them.

Example: `'--or'` is a red dashed line with circle markers

Specifier	Line Style
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Specifier	Marker
`o`	Circle

Specifier	Marker
`+`	Plus sign
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Pentagram
`h`	Hexagram

Specifier	Color
`y`	yellow
`m`	magenta
`c`	cyan
`r`	red
`g`	green
`b`	blue
`w`	white
`k`	black

`pax` — `PolarAxes` object
`PolarAxes` object

`PolarAxes` object. You can modify the appearance and behavior of a `PolarAxes` object by setting its properties. For a list of properties, see PolarAxes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`Name,Value` pair settings apply to all the lines plotted. You cannot specify different `Name,Value` pairs for each line using this syntax. Instead, return the chart line objects and use dot notation to set the properties for each line.

The properties listed here are only a subset. For a full list, see Chart Line.

Example: `'LineWidth',3`

`Color` — Line color
`[0 0.4470 0.7410]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
'-'	Solid line	
'--'	Dashed line	
':'	Dotted line	

Line Style	Description	Resulting Line
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

`Marker` — Marker symbol
`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | `'x'` | `...`

Marker symbol, specified as one of the markers in this table. By default, a chart line does not have markers. Add markers at each data point along the line by specifying a marker symbol.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerSize` — Marker size
6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

`MarkerFaceColor` — Marker fill color
`'none'` (default) | `'auto'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` option uses the same color as the `Color` property of the parent axes. If you specify `'auto'` and the axes plot box is invisible, the marker fill color is the color of the figure.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Tips

- To convert data from degrees to radians, use `deg2rad`. To convert data from radians to degrees, use `rad2deg`.
- You can modify polar axes properties to customize the chart. For a list of properties, see Polar Axes.
- To plot additional data in the polar axes, use the `hold on` command. However, you cannot plot data that requires Cartesian axes in a polar chart.

See Also

Functions
`deg2rad` | `rad2deg` | `rlim`

Properties
Chart Line | PolarAxes

Introduced in R2016a

polarscatter

Scatter chart in polar coordinates

Syntax

```
polarscatter(th,r)
polarscatter(th,r,sz)
polarscatter(th,r,sz,c)

polarscatter( ___ ,mkr)
polarscatter( ___ ,'filled')
polarscatter( ___ ,Name,Value)
polarscatter(pax, ___ )
ps = polarscatter( ___ )
```

Description

`polarscatter(th,r)` plots `th` versus `r` and displays a circle at each data point. `th` and `r` must be vectors of the same length. You must specify `th` in radians.

`polarscatter(th,r,sz)` sets the marker sizes, where `sz` specifies the area of each marker in points squared. To draw all the markers with the same size, specify `sz` as a scalar. To draw the markers with different sizes, specify `sz` as a vector the same length as `th`.

`polarscatter(th,r,sz,c)` sets the marker colors, where `c` is either a vector, a three-column matrix, an RGB triplet, or a color name, such as `'red'`.

`polarscatter( ___ ,mkr)` sets the marker symbol. For example, `'+'` displays cross markers. Specify the marker symbol after any of the input argument combinations in the previous syntaxes.

`polarscatter( ___ ,'filled')` fills the marker interiors.

`polarscatter( ___ ,Name,Value)` modifies the appearance of the scatter plot using one or more name-value pair arguments. For example, you can use semi-transparent markers by specifying `'FaceAlpha'` and a scalar value between `0` and `1`.

`polarscatter(pax, ___ )` plots into the polar axes specified by `pax` instead of into the current axes.

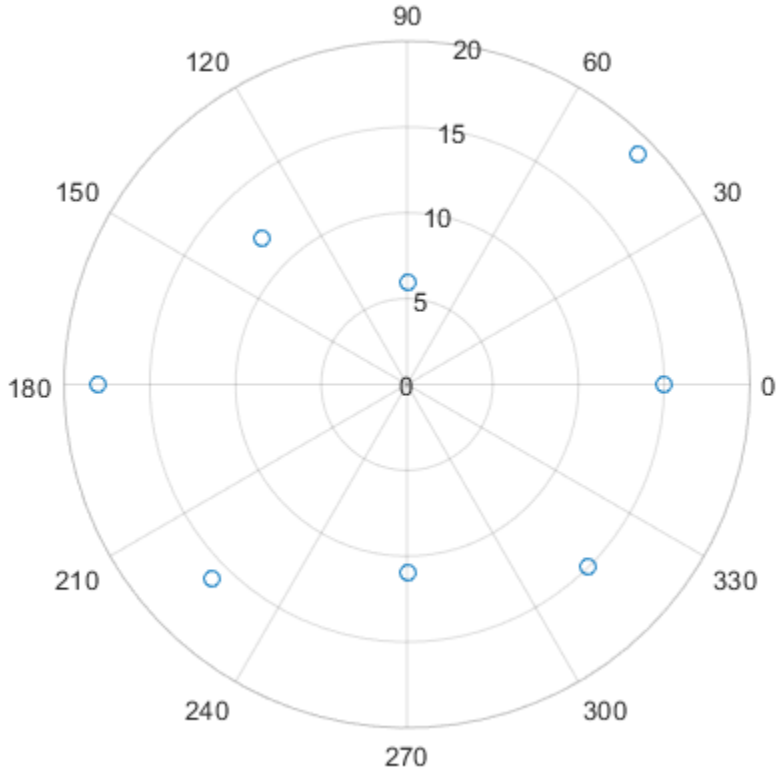
`ps = polarscatter( ___ )` returns the `Scatter` object. Use `ps` to modify the appearance of the `Scatter` object after it is created. For a list of properties, see Scatter.

Examples

Create Scatter Chart

Create a scatter chart in polar coordinates.

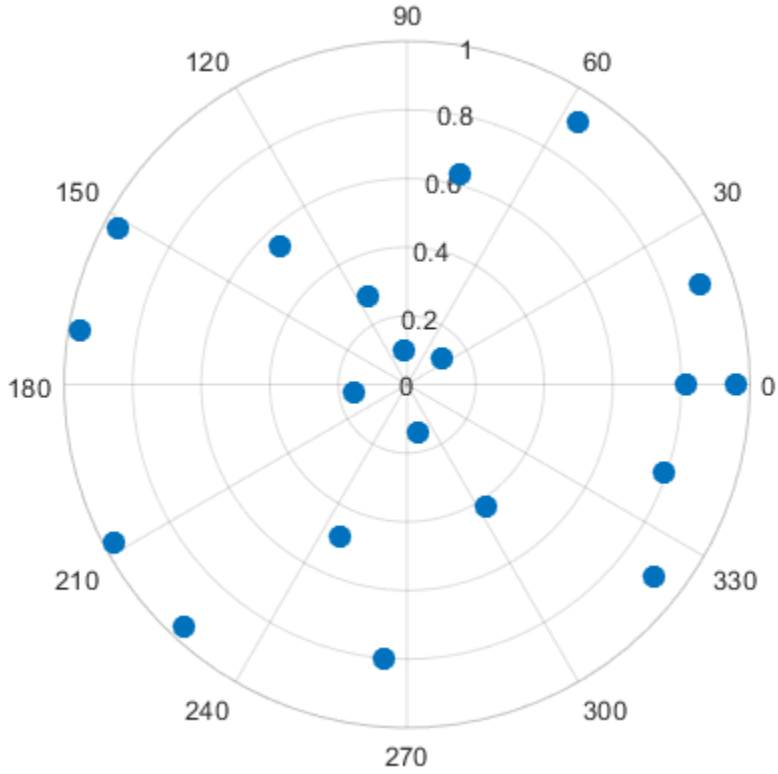
```
th = pi/4:pi/4:2*pi;
r = [19 6 12 18 16 11 15 15];
polarscatter(th,r)
```



Use Filled Markers and Set Marker Size

Create a scatter chart that uses filled markers by specifying the optional input argument, `'filled'`. Set the marker size to 75 points squared.

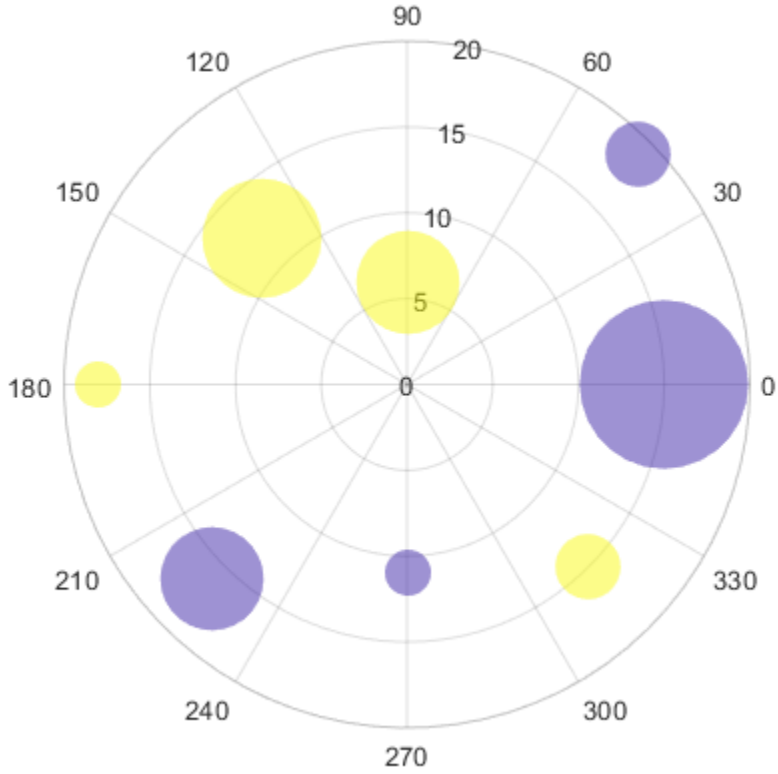
```
th = linspace(0,2*pi,20);
r = rand(1,20);
sz = 75;
polarscatter(th,r,sz,'filled')
```



Use Markers with Varying Sizes and Colors

Create a scatter chart with markers of varying sizes and colors. Specify the optional size and color input arguments as vectors. Use unique values in the color vector to specify the different colors you want. The values map to colors in the colormap.

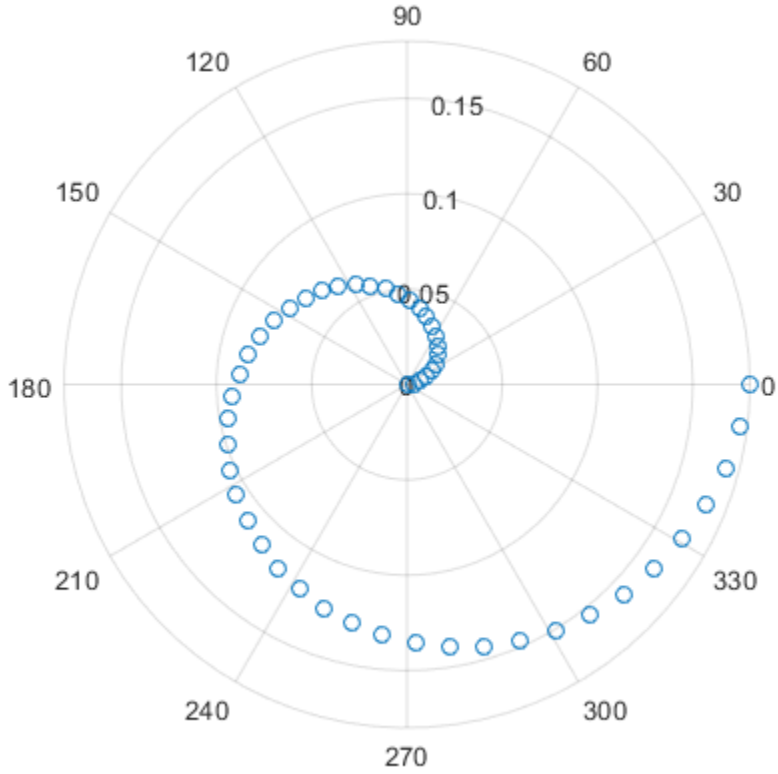
```
th = pi/4:pi/4:2*pi;
r = [19 6 12 18 16 11 15 15];
sz = 100*[6 15 20 3 15 3 6 40];
c = [1 2 2 2 1 1 2 1];
polarscatter(th,r,sz,c,'filled','MarkerFaceAlpha',.5)
```



Convert from Degrees to Radians Before Plotting

Create data where the angle values are in degrees. Since `polarscatter` requires angle values in radians, convert the values to radians before plotting using `deg2rad`.

```
th = linspace(0,360,50);
r = 0.005*th/10;
th_radians = deg2rad(th);
polarscatter(th_radians,r)
```

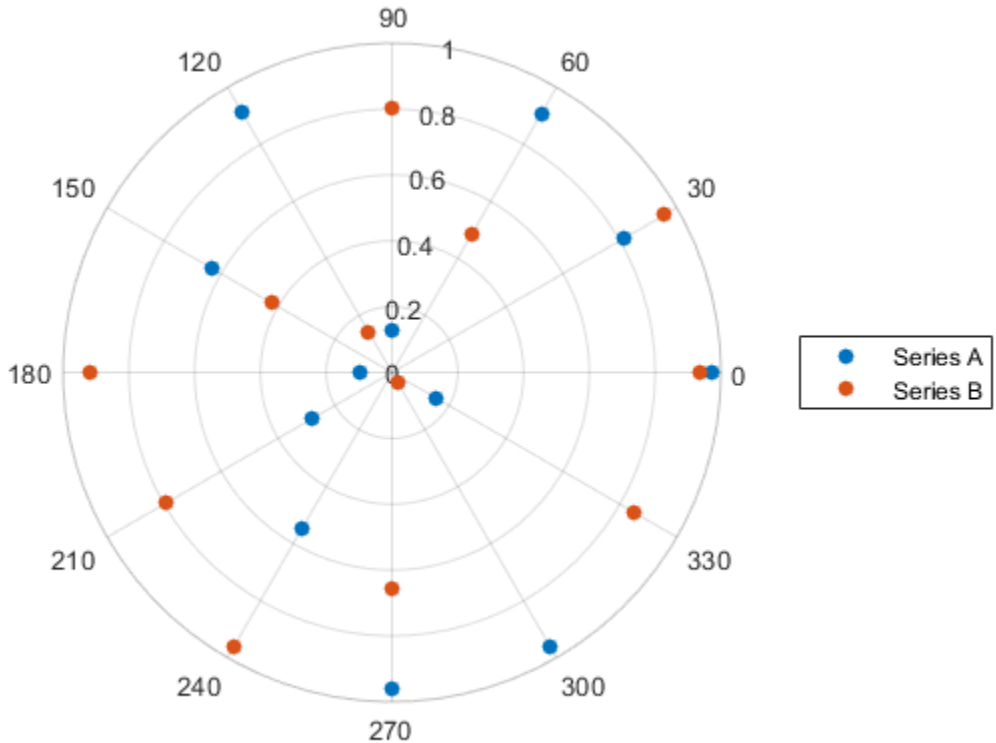


Combine Two Scatter Charts

Combine two scatter charts in the same polar axes using the `hold` command. Add a legend with a description of each chart.

```
th = pi/6:pi/6:2*pi;
r1 = rand(12,1);
polarscatter(th,r1,'filled')

hold on
r2 = rand(12,1);
```

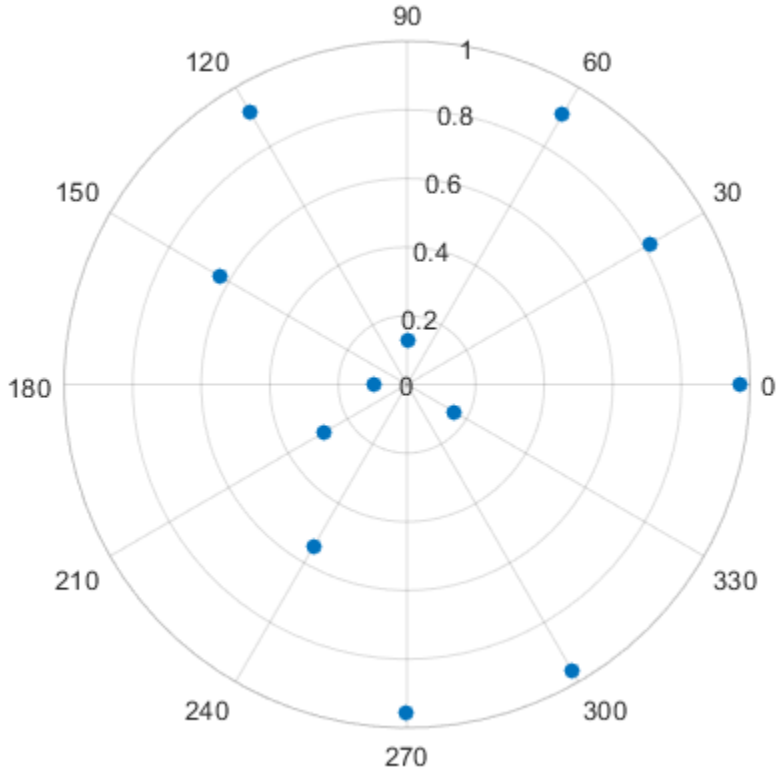
```
polarscatter(th,r2,'filled')
hold off

legend('Series A','Series B')
```



Modify Scatter Chart After Creation

Create a scatter chart and assign the scatter object to the variable `ps`.

```
th = pi/6:pi/6:2*pi;
r = rand(12,1);
ps = polarscatter(th,r,'filled')
```

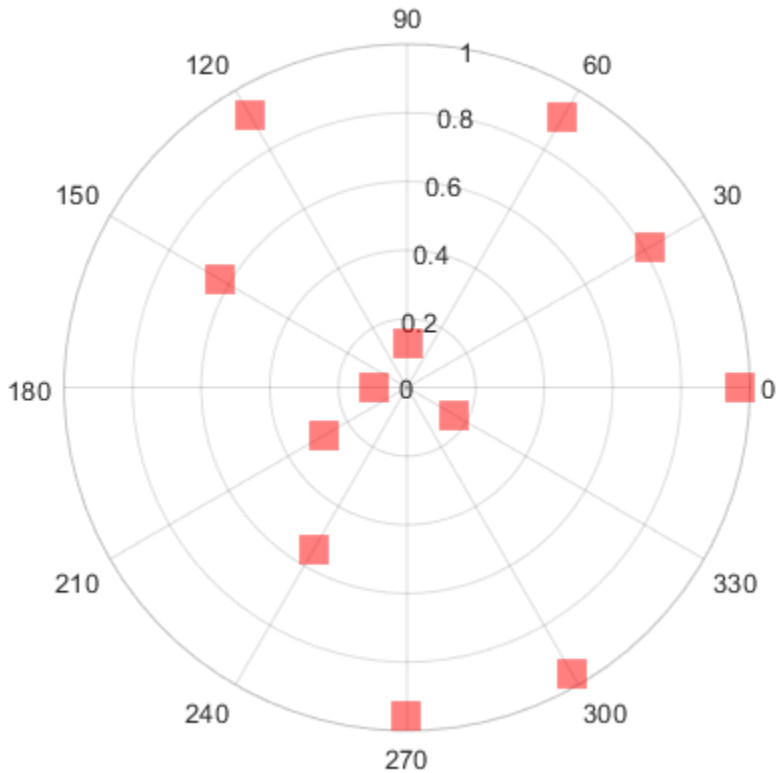



```
ps =
  Scatter with properties:

             Marker: 'o'
    MarkerEdgeColor: 'none'
    MarkerFaceColor: 'flat'
           SizeData: 36
          LineWidth: 0.5000
          ThetaData: [1x12 double]
              RData: [1x12 double]
              ZData: [1x0 double]
              CData: [0 0.4470 0.7410]
```

```
Show all properties
```

Use `ps` to modify properties of the scatter object after it is created.

```
ps.Marker = 'square';
ps.SizeData = 200;
ps.MarkerFaceColor = 'red';
ps.MarkerFaceAlpha = .5;
```



Input Arguments

`th` — *theta* values

vector | scalar

theta values, specified as a vector or a scalar in radians. `th` and `r` must be the same size. To convert degrees to radians, use `deg2rad`.

The `ThetaData` property of the scatter object stores the *theta* values.

Example: `th = [3 2 4 5 7 2 3];`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`r` — *rho* values

vector | scalar

rho values, specified as a vector or a scalar. `th` and `r` must be the same size.

The `RData` property of the scatter object stores the *rho* values.

Example: `r = 0:pi/2:2*pi;`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz` — Marker sizes in points squared

`36` (default) | scalar | vector | `[]`

Marker sizes in points squared, specified in one of these forms:

- Scalar — Uniform marker size. For example, `sz = 100` creates all markers with an area of 100 points squared.
- Vector — Different marker size for each data point. The vector must be the same length as `th` and `r`.
- Empty brackets `[]` — Default marker size with an area of 36 points squared. Use this option if you want to specify the color input argument, but use the default marker area; for example, `polarscatter(th,r,[],c)`.

The `SizeData` property of the scatter object stores the marker sizes.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

c — Marker colors

three-column matrix of RGB triplets | vector | RGB triplet | `'r''g''b'` | ...

Marker colors, specified in one of these forms.

- RGB triplet or color name — Uniform marker color, such as `'red'`.
- Vector — Different marker colors for each data point. The values in `c` are linearly mapped to colors in the colormap. The vector must be the same length as `th` and `r`.
- Three-column matrix of RGB triplets — Different marker colors for each data point using RGB triplet colors. The length of the matrix must be the same length as `th` and `r`.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`

When you specify the marker colors, `polarscatter` sets the `MarkerFaceColor` property of the scatter object to `'flat'` and stores the marker colors in the `CData` property.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

mkr — Marker symbol

`'o'` (default) | `'+'` | `'*'` | `'.'` | `'x'` | `'s'` | ...

Marker symbol, specified as one of the marker symbols listed in this table.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

The `Marker` property of the scatter object stores the marker symbol.

`pax` — `PolarAxes` object
`PolarAxes` object

`PolarAxes` object. If you do not specify the polar axes, then `polarscatter` uses the current axes. `polarscatter` does not support plotting into Cartesian axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The scatter object properties listed here are only a subset. For a complete list, see Scatter.

Example: `polarscatter(th,r,'filled','MarkerFaceAlpha',.5)` creates filled, semi-transparent markers.

`MarkerFaceAlpha` — Marker face transparency
`1` (default) | scalar in range `[0,1]`

Marker face transparency, specified as a scalar in the range `[0,1]`. A value of 1 is opaque and 0 is transparent. Values between 0 and 1 are semi-transparent.

`MarkerEdgeColor` — Marker outline color
`'flat'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified `'flat'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'flat'` uses colors from the `CData` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: [0.5 0.5 0.5]

Example: 'blue'

Example: '#D2F9A7'

MarkerFaceColor — Marker fill color

'none' (default) | 'flat' | 'auto' | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...

Marker fill color, specified as 'flat', 'auto', an RGB triplet, a hexadecimal color code, a color name, or a short name. The 'flat' option uses the CData values. The 'auto' option uses the same color as the Color property for the axes.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to

F. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Example: `'#D2F9A7'`

`LineWidth` — Width of marker edge
`0.5` (default) | positive value

Width of marker edge, specified as a positive value in point units.

Example: `0.75`

See Also

Functions
`hold` | `legend` | `polarhistogram` | `polarplot` | `scatter`

Properties
PolarAxes | Scatter

Introduced in R2016b

poly

Polynomial with specified roots or characteristic polynomial

Syntax

```
p = poly(r)
p = poly(A)
```

Description

`p = poly(r)`, where `r` is a vector, returns the coefficients of the polynomial whose roots are the elements of `r`.

`p = poly(A)`, where `A` is an `n`-by-`n` matrix, returns the `n+1` coefficients of the characteristic polynomial of the matrix, `det`($\lambda I - A$).

Examples

Characteristic Polynomial from Eigenvalues

Calculate the eigenvalues of a matrix, `A`.

```
A = [1 8 -10; -4 2 4; -5 2 8]
```

```
A = 3×3

     1     8   -10
    -4     2     4
    -5     2     8
```

```
e = eig(A)
```

```
e = 3×1 complex
```

```
  11.6219 + 0.0000i
  -0.3110 + 2.6704i
  -0.3110 - 2.6704i
```

Since the eigenvalues in `e` are the roots of the characteristic polynomial of `A`, use `poly` to determine the characteristic polynomial from the values in `e`.

```
p = poly(e)
```

```
p = 1×4

    1.0000  -11.0000   -0.0000  -84.0000
```

Characteristic Polynomial of Matrix

Use `poly` to calculate the characteristic polynomial of a matrix, `A`.

```
A = [1 2 3; 4 5 6; 7 8 0]
```

```
A = 3×3

     1     2     3
     4     5     6
     7     8     0
```

```
p = poly(A)
```

```
p = 1×4

    1.0000   -6.0000  -72.0000  -27.0000
```

Calculate the roots of `p` using `roots`. The roots of the characteristic polynomial are the eigenvalues of matrix `A`.

```
r = roots(p)
```

```
r = 3×1

   12.1229
```

```
-5.7345
-0.3884
```

Input Arguments

r — Polynomial roots

vector

Polynomial roots, specified as a vector.

Example: `poly([2 -3])`

Example: `poly([2 -2 3 -3])`

Example: `poly(roots(k))`

Example: `poly(eig(A))`

Data Types: `single` | `double`
Complex Number Support: Yes

A — Input matrix

matrix

Input matrix.

Example: `poly([0 -1; 1 0])`

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

p — Polynomial coefficients

row vector

Polynomial coefficients, returned as a row vector.

- If the input is a square n-by-n matrix, A, then p contains the coefficients for the characteristic polynomial of A.

- If the input is a vector of roots, `r`, then `p` contains the coefficients for the polynomial whose roots are in `r`.

In each case, the `n+1` coefficients in `p` describe the polynomial

$$p_1x^n + p_2x^{n-1} + \ldots + p_nx + p_{n+1} \ .$$

Tips

- For vectors, `r = roots(p)` and `p = poly(r)` are inverse functions of each other, up to roundoff error, ordering, and scaling.

Algorithms

The algorithms employed for `poly` and `roots` illustrate an interesting aspect of the modern approach to eigenvalue computation. `poly(A)` generates the characteristic polynomial of `A`, and `roots(poly(A))` finds the roots of that polynomial, which are the eigenvalues of `A`. But both `poly` and `roots` use `eig`, which is based on similarity transformations. The classical approach, which characterizes eigenvalues as roots of the characteristic polynomial, is actually reversed.

If `A` is an `n`-by-`n` matrix, `poly(A)` produces the coefficients `p(1)` through `p(n+1)`, with `p(1) = 1`, in

$$\det(\lambda I - A) = p_1\lambda^n + \ldots + p_n\lambda + p_{n+1} \ .$$

The algorithm is

```
z = eig(A);
p = zeros(n+1,1);
p(1) = 1;
for j = 1:n
    p(2:j+1) = p(2:j+1)-z(j)*p(1:j);
end
```

This recursion is derived by expanding the product,

$$(\lambda - \lambda_1)(\lambda - \lambda_2)\ldots(\lambda - \lambda_n) \ .$$

It is possible to prove that `poly(A)` produces the coefficients in the characteristic polynomial of a matrix within roundoff error of `A`. This is true even if the eigenvalues of `A` are badly conditioned. The traditional algorithms for obtaining the characteristic polynomial do not use the eigenvalues, and do not have such satisfactory numerical properties.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not discard nonfinite input values.
- Complex input produces complex output.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`conv` | `polyval` | `polyvalm` | `residue` | `roots`

Topics

“Create and Evaluate Polynomials”

Introduced before R2006a

polyarea

Area of polygon

Syntax

```
a = polyarea(x,y)
a = polyarea(x,y,dim)
```

Description

`a = polyarea(x,y)` returns the area of the 2-D polygon defined by the vertices in vectors `x` and `y`.

- If `x` and `y` are vectors of the same length, then `polyarea` returns the scalar area of the polygon defined by `x` and `y`.
- If `x` and `y` are matrices of the same size, then `polyarea` returns a row vector containing the areas of each polygon defined by the columnwise pairs in `x` and `y`.
- If `x` and `y` are multidimensional arrays, then `polyarea` operates along the first dimension whose length is not equal to 1.

`a = polyarea(x,y,dim)` operates along dimension `dim`.

Examples

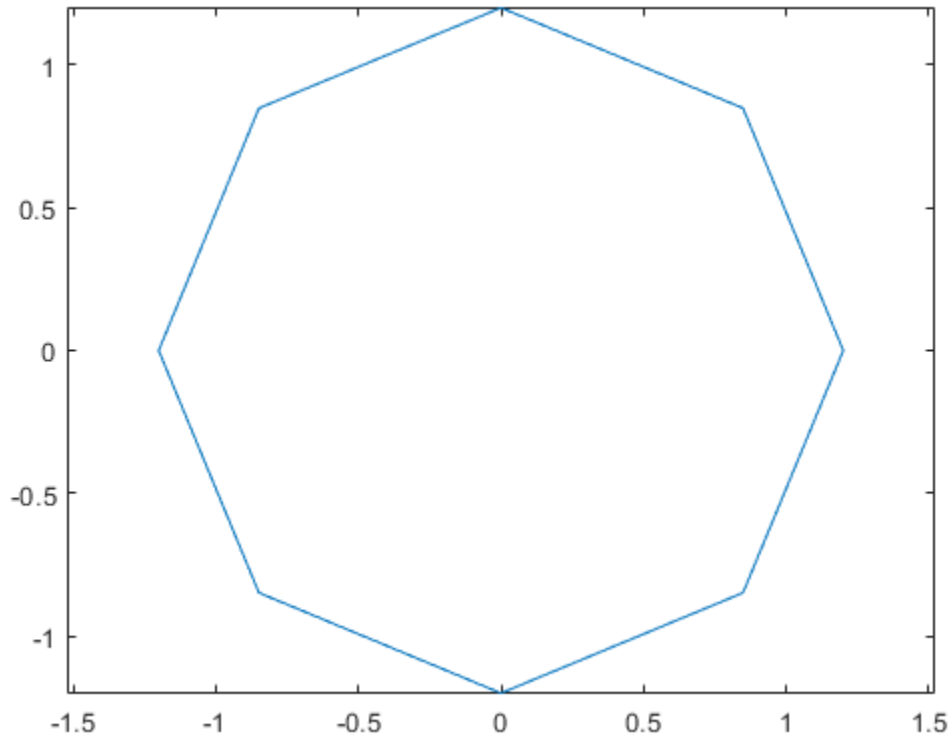
Area of Polygon

Create a set of points that define a 2-D polygon.

```
p = linspace(0,2.*pi,9);
x = 1.2*cos(p)';
y = 1.2*sin(p)';
```

Plot the polygon.

```
plot(x,y);
axis equal
```



Compute the area of the polygon.

```
a = polyarea(x,y)
```

```
a = 4.0729
```

Input Arguments

x — *x*-coordinates
vector | matrix | multidimensional array

x-coordinates, specified as a vector, matrix, or multidimensional array.

y — *y*-coordinates
vector | matrix | multidimensional array

y-coordinates, specified as a vector, matrix, or multidimensional array.

`dim` — Dimension
positive integer scalar

Dimension to operate along, specified as a positive integer scalar.

Tips

- The area value returned by `polyarea` can depend on the order of the input points.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`inpolygon` | `nsidedpoly` | `polyshape` | `rectint`

Introduced before R2006a

polybuffer

Create buffer around points, lines, or `polyshape` objects

Syntax

```
polyout = polybuffer(P,'points',d)
polyout = polybuffer(P,'lines',d)
polyout = polybuffer(P,'lines',d,'JointType',jt)
polyout = polybuffer(P,'lines',d,'JointType','miter','MiterLimit',
lim)
```

```
polyout = polybuffer(polyin,d)
polyout = polybuffer(polyin,d,'JointType',jt)
polyout = polybuffer(polyin,d,'JointType','miter','MiterLimit',lim)
```

Description

`polyout = polybuffer(P,'points',d)` returns a `polyshape` object with boundaries that buffer the 2-D points in `P` by a distance `d`. The `polybuffer` function computes the buffer by taking the union of circles of radius `d` centered at each point in `P`. The first column of the matrix `P` contains the *x*-coordinates of the points to buffer and the second column contains the corresponding *y*-coordinates.

Once you create the `polyshape` object `polyout`, you can analyze its properties or perform additional computations using `polyshape` functions. For example, you can access the vertices that define the buffered shape with the property `polyout.Vertices`, and you can plot the shape using the command `plot(polyout)`.

`polyout = polybuffer(P,'lines',d)` returns a `polyshape` object with boundaries that buffer the line segments defined by the 2-D points in `P`. The `polybuffer` function computes the buffer by moving a circle of radius `d` along the line segments created by the input points, centered on the line segments.

`polyout = polybuffer(P,'lines',d,'JointType',jt)` specifies how the meeting points of line segment endpoints (joints) are treated. For example, `polybuffer(P,'lines',d,'JointType','square')` squares off the joints.

`polyout = polybuffer(P,'lines',d,'JointType','miter','MiterLimit', lim)` specifies a miter limit when the joint type is specified as `'miter'`. The miter limit is the maximum allowable ratio between the distance a joint vertex is moved and the buffer distance `d`. The limit `lim` must be greater than or equal to 2.

`polyout = polybuffer(polyin,d)` returns a `polyshape` object with boundaries that buffer the input `polyshape` object `polyin` by a distance `d`. For positive values of `d`, solid region boundaries of `polyin` expand by `d` units, and hole boundaries shrink by `d` units. Negative values of `d` shrink solid boundaries and expand hole boundaries.

`polyout = polybuffer(polyin,d,'JointType',jt)` specifies how the joints of the `polyshape` are treated when the first input argument is a `polyshape`.

`polyout = polybuffer(polyin,d,'JointType','miter','MiterLimit',lim)` specifies a miter limit when the joint type is specified as `'miter'` and the first input argument is a `polyshape`.

Examples

Buffer 2-D Points

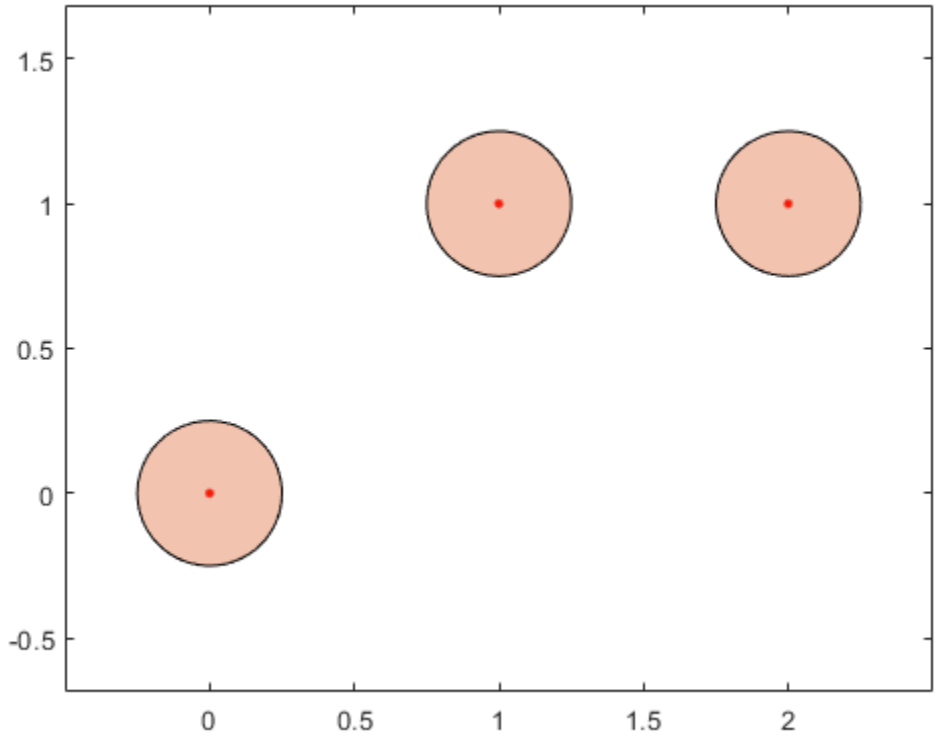
Create a matrix that contains the 2-D coordinates of three points. Then, create a `polyshape` object that buffers each point by a distance of 0.25 units.

```
P = [0 0; 1 1; 2 1];
polyout = polybuffer(P,'points',0.25)
```

```
polyout 
  polyshape with properties:

      Vertices: [542x2 double]
    NumRegions: 3
      NumHoles: 0
```

Plot the original points and their buffer regions.

```
plot(P(:,1),P(:,2),'r.','MarkerSize',10)
hold on
plot(polyout)
axis equal
```



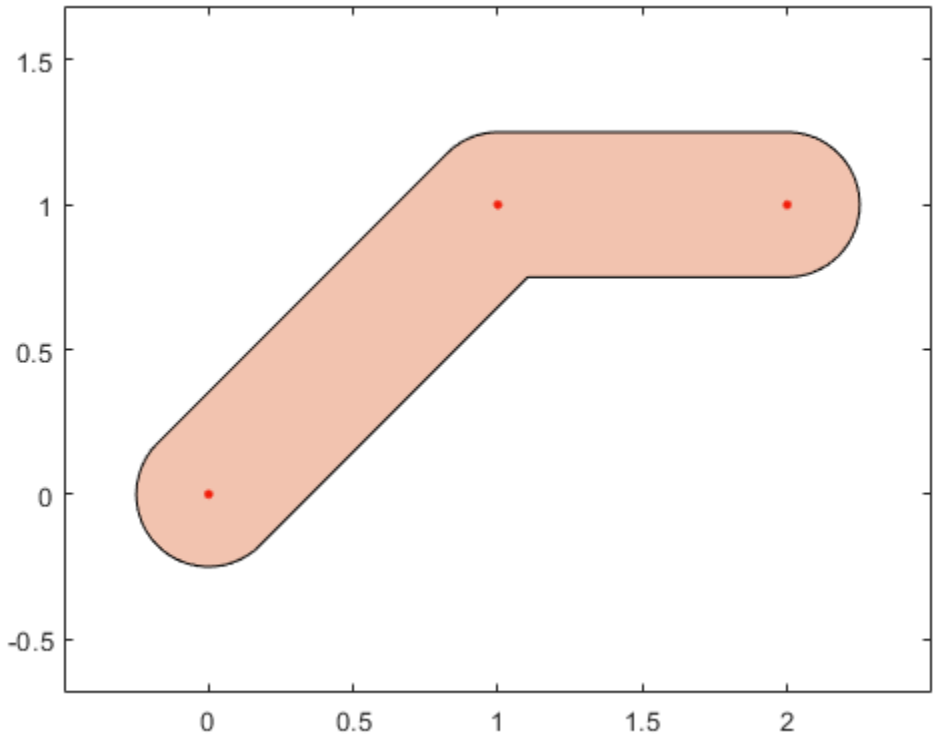
Buffer Line Segments

Create buffer regions surrounding line segments.

Create a matrix of 2-D points, and compute a `polyshape` object that buffers the line segments connecting the points.

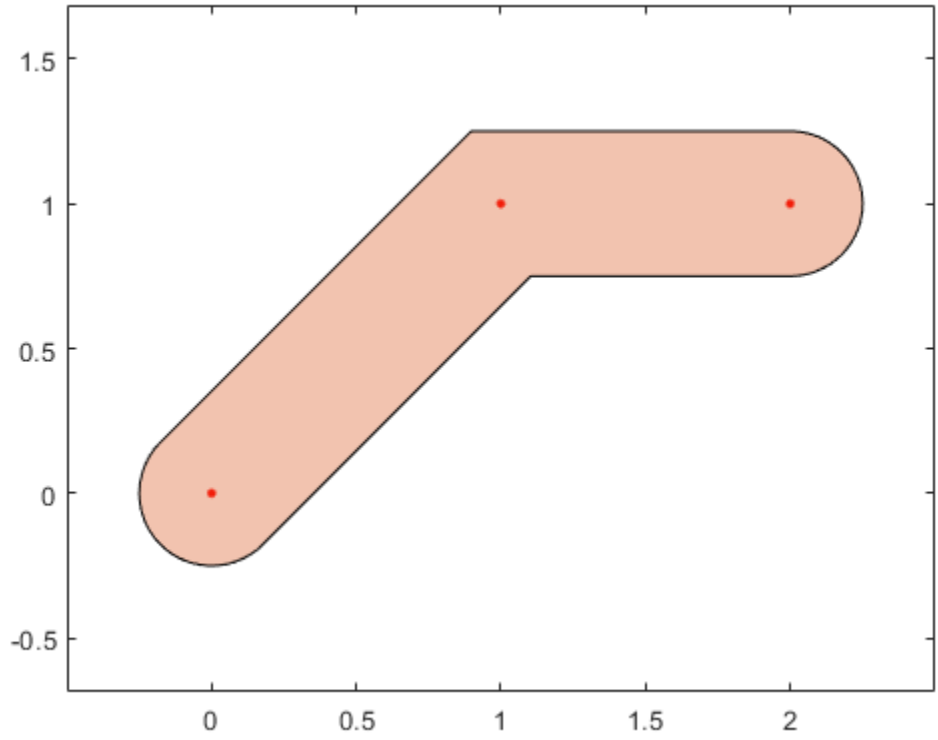
```
P = [0 0; 1 1; 2 1];
polyout1 = polybuffer(P,'lines',0.25);
plot(P(:,1),P(:,2),'r.','MarkerSize',10)
hold on
```

```
plot(polyout1)
axis equal
hold off
```



By default, `polybuffer` rounds out the joints that connect line segments. You can control the shape of joints by using the `'JointType'` name-value pair. For example, use the value `'miter'` to preserve the angle connecting the two line segments.

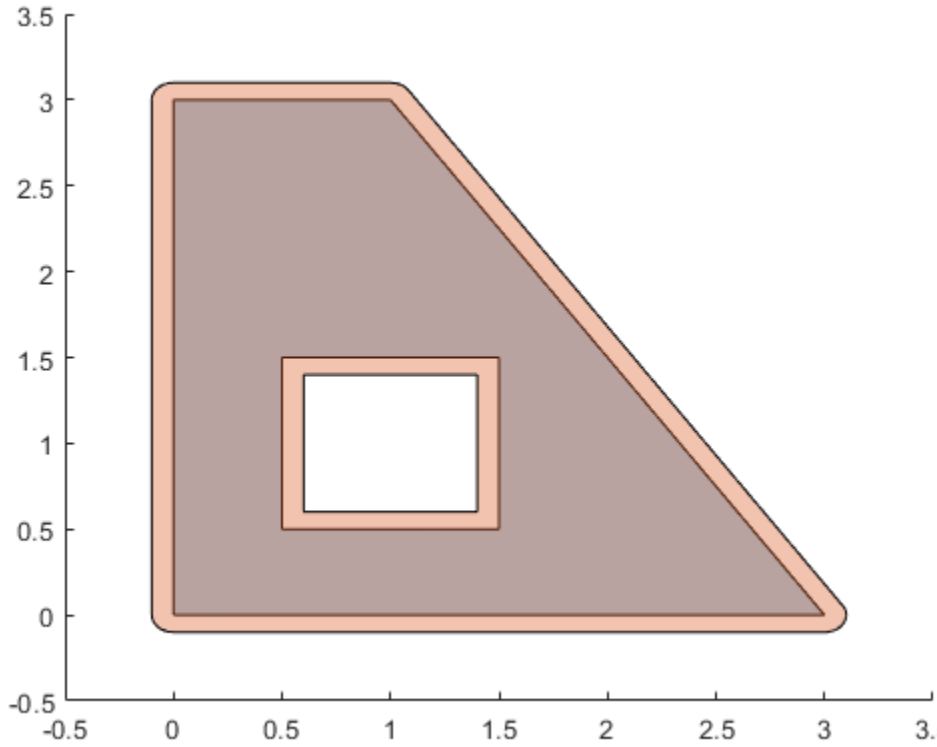
```
figure
polyout2 = polybuffer(P,'lines',0.25,'JointType','miter');
plot(P(:,1),P(:,2),'r.','MarkerSize',10)
hold on
plot(polyout2)
axis equal
```



Buffer Polygon

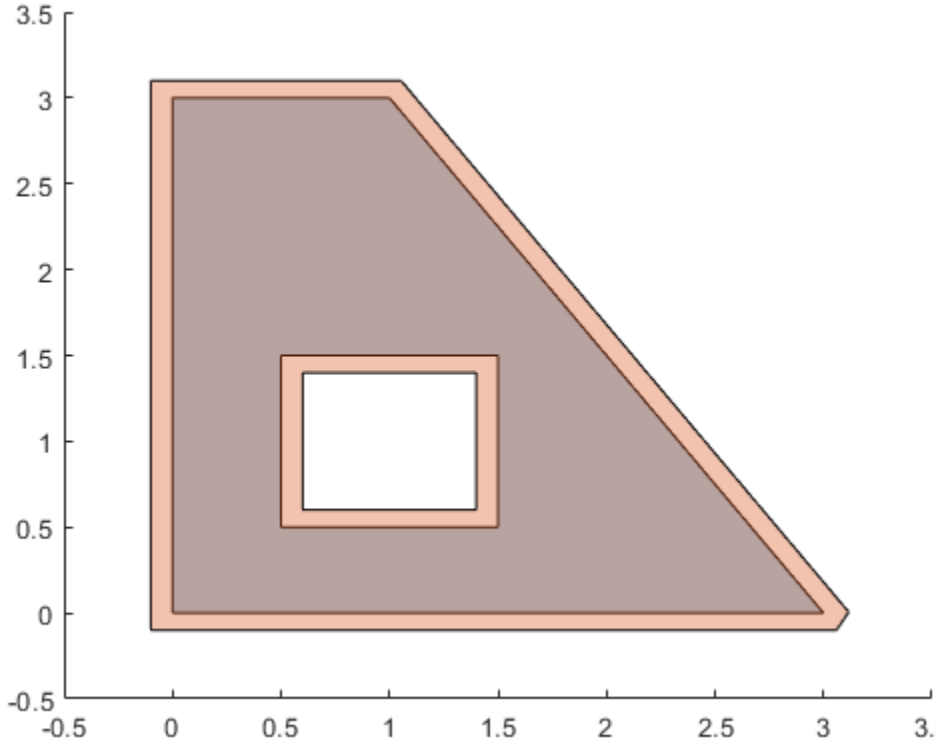
Create a polygon with a solid boundary and a hole boundary. Then, create a buffer at a distance of 0.1 from the boundaries. By default the buffer has rounded joints.

```
polyin = polyshape({[0 0 1 3],[0.5 1.5 1.5 0.5]},{[0 3 3 0],[0.5 0.5 1.5 1.5]});
plot(polyin)
polyout1 = polybuffer(polyin,0.1);
hold on
plot(polyout1)
hold off
```



Create a buffer using a miter limit of 2. The miter limit controls the pointiness of the joints.

```
polyout2 = polybuffer(polyin,0.1,'JointType','miter','MiterLimit',2);
plot(polyin)
hold on
plot(polyout2)
```



Input Arguments

P — Input vertices
2-column numeric matrix

Input vertices of 2-D points, specified as a 2-column numeric matrix. The first column of P contains the *x*-coordinates and the second column contains the corresponding *y*-coordinates.

`polyin` — Input `polyshape`
scalar | vector | matrix | multidimensional array

Input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array. When `polyin` is an array of `polyshape` objects, `polybuffer` applies the specified buffer parameters to each element.

`d` — Buffer distance
numeric scalar

Buffer distance, specified as a numeric scalar.

When the first input argument is a set of vertices, `d` must be a positive numeric scalar.

When the first input argument is a `polyshape`, `d` can be positive, negative, or zero:

- If `d>0`, then solid boundaries grow by a distance `d` and hole boundaries shrink.
- If `d<0`, then solid boundaries shrink by a distance `d` and hole boundaries grow.
- If `d=0`, then there is no change to the input boundaries.

`jt` — Joint type
`'round'` (default) | `'square'` | `'miter'`

Joint type for buffer boundaries, specified as one of the following:

- `'round'` — Round out boundary corners.
- `'square'` — Square off boundary corners.
- `'miter'` — Limit the ratio between the distance a joint vertex is moved and the buffer distance to 3. This limit prevents excessive pointiness.

When the first input argument is a set of vertices, `polybuffer` applies the joint type only where endpoints of two line segments meet, and not at an open endpoint of a line segment.

When the first input argument is a `polyshape`, `polybuffer` only applies the joint type to solid boundaries when the buffer distance is positive, or for hole boundaries when the buffer distance is negative.

`lim` — Miter limit
positive numeric scalar

Miter limit, specified as a positive numeric scalar greater than or equal to 2. The miter limit is the ratio between the distance a joint vertex is moved and the buffer distance. Setting a miter limit controls the pointiness of boundary joints.

See Also

`convhull` | `polyshape`

Introduced in R2017b

polyder

Polynomial differentiation

Syntax

```
k = polyder(p)
k = polyder(a,b)
[q,d] = polyder(a,b)
```

Description

`k = polyder(p)` returns the derivative of the polynomial represented by the coefficients in `p`,

$$k(x) = \frac{d}{dx}p(x) \ .$$

`k = polyder(a,b)` returns the derivative of the product of the polynomials `a` and `b`,

$$k(x) = \frac{d}{dx}[a(x)b(x)] \ .$$

`[q,d] = polyder(a,b)` returns the derivative of the quotient of the polynomials `a` and `b`,

$$\frac{q(x)}{d(x)} = \frac{d}{dx}\left[\frac{a(x)}{b(x)}\right] \ .$$

Examples

Differentiate Polynomial

Create a vector to represent the polynomial $p(x) = 3x^5 - 2x^3 + x + 5$.

```
p = [3 0 -2 0 1 5];
```

Use `polyder` to differentiate the polynomial. The result is $q(x) = 15x^4 - 6x^2 + 1$.

```
q = polyder(p)
```

```
q = 1×5

    15     0    -6     0     1
```

Differentiate Product of Polynomials

Create two vectors to represent the polynomials $a(x) = x^4 - 2x^3 + 11$ and $b(x) = x^2 - 10x + 15$.

```
a = [1 -2 0 0 11];
b = [1 -10 15];
```

Use `polyder` to calculate

$$q(x) = \frac{d}{dx}[a(x)b(x)].$$

```
q = polyder(a,b)
```

```
q = 1×6

     6   -60   140   -90    22  -110
```

The result is

$$q(x) = 6x^5 - 60x^4 + 140x^3 - 90x^2 + 22x - 110.$$

Differentiate Quotient of Polynomials

Create two vectors to represent the polynomials in the quotient,

$$\frac{x^4 - 3x^2 - 1}{x + 4}.$$

```
p = [1 0 -3 0 -1];
v = [1 4];
```

Use `polyder` with two output arguments to calculate

$$\frac{q(x)}{d(x)} = \frac{d}{dx}\left[\frac{p(x)}{v(x)}\right].$$

```
[q,d] = polyder(p,v)
```

```
q = 1×5

     3    16    -3   -24     1


d = 1×3

     1     8    16
```

The result is

$$\frac{q(x)}{d(x)} = \frac{3x^4 + 16x^3 - 3x^2 - 24x + 1}{x^2 + 8x + 16}.$$

Input Arguments

p — Polynomial coefficients
vector

Polynomial coefficients, specified as a vector. For example, the vector `[1 0 1]` represents the polynomial $x^2 + 1$, and the vector `[3.13 -2.21 5.99]` represents the polynomial $3.13x^2 - 2.21x + 5.99$.

For more information, see “Create and Evaluate Polynomials”.

Data Types: `single` | `double`
Complex Number Support: Yes

a,b — Polynomial coefficients (as separate arguments)
row vectors

Polynomial coefficients, specified as two separate arguments of row vectors.

For more information, see “Create and Evaluate Polynomials”.

Example: `polyder([1 0 -1],[10 2])`

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

k — Differentiated polynomial coefficients
row vector

Differentiated polynomial coefficients, returned as a row vector.

q — Numerator polynomial
row vector

Numerator polynomial, returned as a row vector.

d — Denominator polynomial
row vector

Denominator polynomial, returned as a row vector.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The output can contain fewer `NaN`s than the MATLAB output. However, if the input contains a `NaN`, the output contains at least one `NaN`.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`conv` | `deconv` | `polyint` | `polyval`

Topics

“Create and Evaluate Polynomials”
“Integrate and Differentiate Polynomials”

Introduced before R2006a

polyeig

Polynomial eigenvalue problem

Syntax

```
e = polyeig(A0,A1,...,Ap)
[X,e] = polyeig(A0,A1,...,Ap)
[X,e,s] = polyeig(A0,A1,...,Ap)
```

Description

`e = polyeig(A0,A1,...,Ap)` returns the eigenvalues for the polynomial eigenvalue problem on page 1-11749 of degree `p`.

`[X,e] = polyeig(A0,A1,...,Ap)` also returns matrix `X`, of size `n`-by-`n*p`, whose columns are the eigenvectors.

`[X,e,s] = polyeig(A0,A1,...,Ap)` additionally returns vector `s`, of length `p*n`, containing condition numbers for the eigenvalues. At least one of `A0` and `Ap` must be nonsingular. Large condition numbers imply that the problem is close to a problem with repeated eigenvalues.

Examples

Quadratic Eigenvalue Problem with Mass, Damping, and Stiffness Matrices

Solve a quadratic eigenvalue problem involving a mass matrix `M`, damping matrix `C`, and stiffness matrix `K`. This quadratic eigenvalue problem arises from the equation of motion:

$$M\frac{d^2y}{dt^2} + C\frac{dy}{dt} + Ky = f(t)$$

This equation applies to a broad range of oscillating systems, including a dynamic mass-spring system or RLC electronic network. The fundamental solution is $y(t) = xe^{\lambda t}$, so both λ and x must solve the quadratic eigenvalue problem (QEP),

$$\left(M\lambda^2 + C\lambda + K\right)x = 0$$

Create coefficient matrices M, C, and K to represent a mass-spring system with four-degrees-of-freedom. The coefficient matrices are all symmetric and positive semidefinite, and M is a diagonal matrix.

```
M = diag([3 1 3 1])
```

```
M = 4×4

     3     0     0     0
     0     1     0     0
     0     0     3     0
     0     0     0     1
```

```
C = [0.4 0 -0.3 0; 0 0 0 0; -0.3 0 0.5 -0.2; 0 0 -0.2 0.2]
```

```
C = 4×4

    0.4000         0   -0.3000         0
         0         0         0         0
   -0.3000         0    0.5000   -0.2000
         0         0   -0.2000    0.2000
```

```
K = [-7 2 4  0; 2 -4 2 0; 4 2 -9 3; 0 0 3 -3]
```

```
K = 4×4

    -7     2     4     0
     2    -4     2     0
     4     2    -9     3
     0     0     3    -3
```

Solve the QEP for the eigenvalues, eigenvectors, and condition numbers using `polyeig`.

```
[X,e,s] = polyeig(K,C,M)
```

```
X = 4×8

    0.1828    0.3421    0.3989    0.0621    0.3890   -0.4143   -0.4575    0.4563
    0.3530   -0.9296    0.3330   -0.8571   -0.6366   -0.2717   -0.4981    0.4985
   -0.5360   -0.0456   -0.1724    0.3509   -0.3423    0.1666   -0.5106    0.5107
    0.7448    0.1295   -0.8368   -0.3720    0.5712    0.8525   -0.5309    0.5315


e = 8×1

   -2.4498
   -2.1536
   -1.6248
    2.2279
    2.0364
    1.4752
    0.3353
   -0.3466


s = 8×1

    0.5813
    0.8609
    1.2232
    0.7855
    0.7012
    1.2922
   10.1097
   10.0519
```

Check that the first eigenvalue, `e(1)`, and first eigenvector, `X(:,1)`, satisfy the QEP equation. The result is close to, but not exactly, zero.

```
lambda = e(1);
x = X(:,1);
(M*lambda^2 + C*lambda + K)*x
```

```
ans = 4×1
10^-13 ×

   -0.0133
   -0.0466
    0.1465
```

```
-0.0622
```

Input Arguments

`A0,A1,...,Ap` — Square coefficient matrices (as separate arguments)
matrices

Square coefficient matrices, specified as separate arguments. The matrices must all have the same order, `n`.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

`e` — Eigenvalues
vector

Eigenvalues, returned as a vector.

`X` — Eigenvectors
matrix

Eigenvectors, returned in the columns of a matrix. The first eigenvector is `X(:,1)`, the second is `X(:,2)`, and so on.

`s` — Condition numbers
vector

Condition numbers, returned as a vector. The condition numbers in `s` correspond to similarly located eigenvalues in `e`. Large condition numbers indicate that the problem is close to having repeated eigenvalues.

More About

Polynomial Eigenvalue Problem

The polynomial eigenvalue problem is a variant of the standard eigenvalue problem, $Ax = \lambda x$, but instead involves polynomials rather than linear terms.

As with the standard eigenvalue problem, the solution involves finding the eigenvalues and eigenvectors that satisfy the equation,

$$\left(A_0 + \lambda A_1 + \ldots + \lambda^P A_p\right)x = 0 \,,$$

where the polynomial degree, `p`, is a nonnegative integer, and `A0,A1,...Ap` are square coefficient matrices of order `n`.

The most common form is the *quadratic* polynomial eigenvalue problem, which is

$$\left(A_2\lambda^2 + A_1\lambda + A_0\right)x = 0 \ .$$

One major difference between the quadratic eigenvalue problem and the standard (or generalized) eigenvalue problem is that there can be up to `2n` eigenvalues with up to `2n` right and left eigenvectors. In cases where there are more than `n` eigenvectors, the eigenvectors do not form a linearly independent set. See [1] and [2] for more detailed information about the quadratic eigenvalue problem.

Tips

- `polyeig` handles the following simplified cases:
  - `p = 0`, or `polyeig(A)`, is the standard eigenvalue problem, `eig(A)`.
  - `p = 1`, or `polyeig(A,B)`, is the generalized eigenvalue problem, `eig(A,-B)`.
  - `n = 0`, or `polyeig(a0,a1,...,ap)`, is the standard polynomial problem, `roots([ap ... a1 a0])`, where `a0,a1,...,ap` are scalars.

Algorithms

The `polyeig` function uses the QZ factorization to find intermediate results in the computation of generalized eigenvalues. `polyeig` uses the intermediate results to

determine if the eigenvalues are well-determined. See the descriptions of `eig` and `qz` for more information.

The computed solutions might not exist or be unique, and can also be computationally inaccurate. If both `A0` and `Ap` are singular matrices, then the problem might be ill-posed. If only one of `A0` and `Ap` is singular, then some of the eigenvalues might be `0` or `Inf`.

Scaling `A0,A1,...,Ap` to have `norm(Ai)` roughly equal to `1` might increase the accuracy of `polyeig`. In general, however, this improved accuracy is not achievable. (See Tisseur [3] for details).

References

[1] Dedieu, Jean-Pierre, and Francoise Tisseur. “Perturbation theory for homogeneous polynomial eigenvalue problems.” *Linear Algebra Appl.* Vol. 358, 2003, pp. 71–94.

[2] Tisseur, Francoise, and Karl Meerbergen. “The quadratic eigenvalue problem.” *SIAM Rev.* Vol. 43, Number 2, 2001, pp. 235–286.

[3] Francoise Tisseur. “Backward error and condition of polynomial eigenvalue problems.” *Linear Algebra Appl.* Vol. 309, 2000, pp. 339–361.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The basis of the eigenvectors can be different in the generated code than in MATLAB. In general, in the eigenvalues output, the eigenvalues for real inputs are not sorted so that complex conjugate pairs are adjacent.
- Differences in eigenvectors and ordering of eigenvalues can lead to differences in the condition numbers output.
- Code generation does not support sparse matrix inputs for this function.

See Also

`cond` | `condeig` | `eig` | `qz`

Topics

"Create and Evaluate Polynomials"

Introduced before R2006a

polyfit

Polynomial curve fitting

Syntax

```
p = polyfit(x,y,n)
[p,S] = polyfit(x,y,n)
[p,S,mu] = polyfit(x,y,n)
```

Description

`p = polyfit(x,y,n)` returns the coefficients for a polynomial `p(x)` of degree `n` that is a best fit (in a least-squares sense) for the data in `y`. The coefficients in `p` are in descending powers, and the length of `p` is `n+1`

$$p(x) = p_1x^n + p_2x^{n-1} + ... + p_nx + p_{n+1}.$$

`[p,S] = polyfit(x,y,n)` also returns a structure `S` that can be used as an input to `polyval` to obtain error estimates.

`[p,S,mu] = polyfit(x,y,n)` also returns `mu`, which is a two-element vector with centering and scaling values. `mu(1)` is `mean(x)`, and `mu(2)` is `std(x)`. Using these values, `polyfit` centers `x` at zero and scales it to have unit standard deviation,

$$\widehat{x} = \frac{x - \bar{x}}{\sigma_x}.$$

This centering and scaling transformation improves the numerical properties of both the polynomial and the fitting algorithm.

Examples

Fit Polynomial to Trigonometric Function

Generate 10 points equally spaced along a sine curve in the interval `[0,4*pi]`.

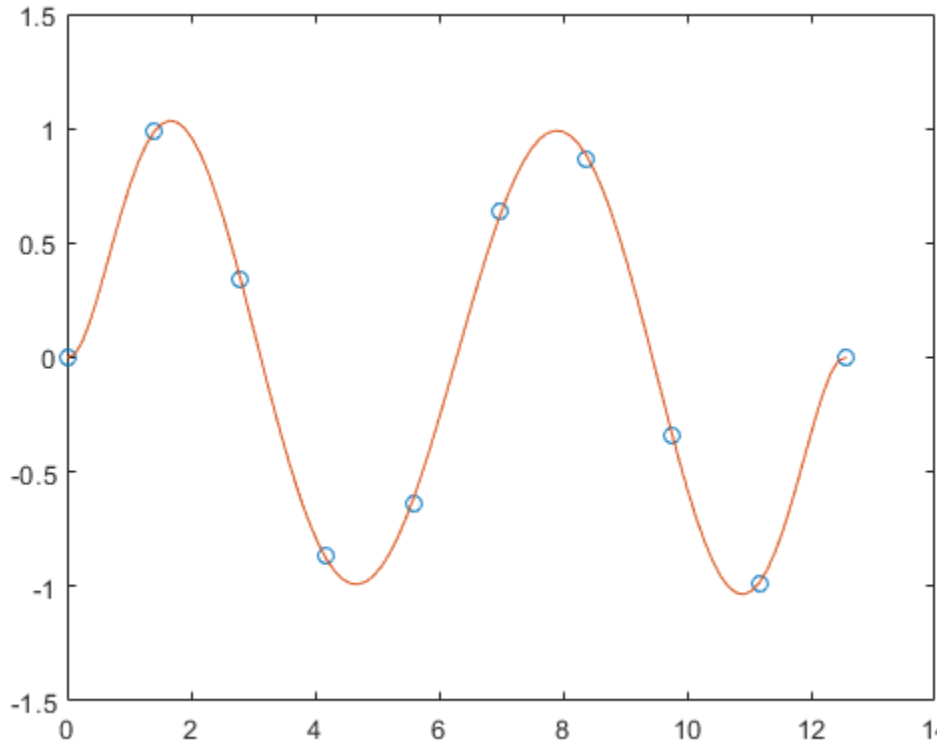
```
x = linspace(0,4*pi,10);
y = sin(x);
```

Use `polyfit` to fit a 7th-degree polynomial to the points.

```
p = polyfit(x,y,7);
```

Evaluate the polynomial on a finer grid and plot the results.

```
x1 = linspace(0,4*pi);
y1 = polyval(p,x1);
figure
plot(x,y,'o')
hold on
plot(x1,y1)
hold off
```



Fit Polynomial to Set of Points

Create a vector of 5 equally spaced points in the interval `[0,1]`, and evaluate $y(x) = (1 + x)^{-1}$ at those points.

```
x = linspace(0,1,5);
y = 1./(1+x);
```

Fit a polynomial of degree 4 to the 5 points. In general, for `n` points, you can fit a polynomial of degree `n-1` to exactly pass through the points.

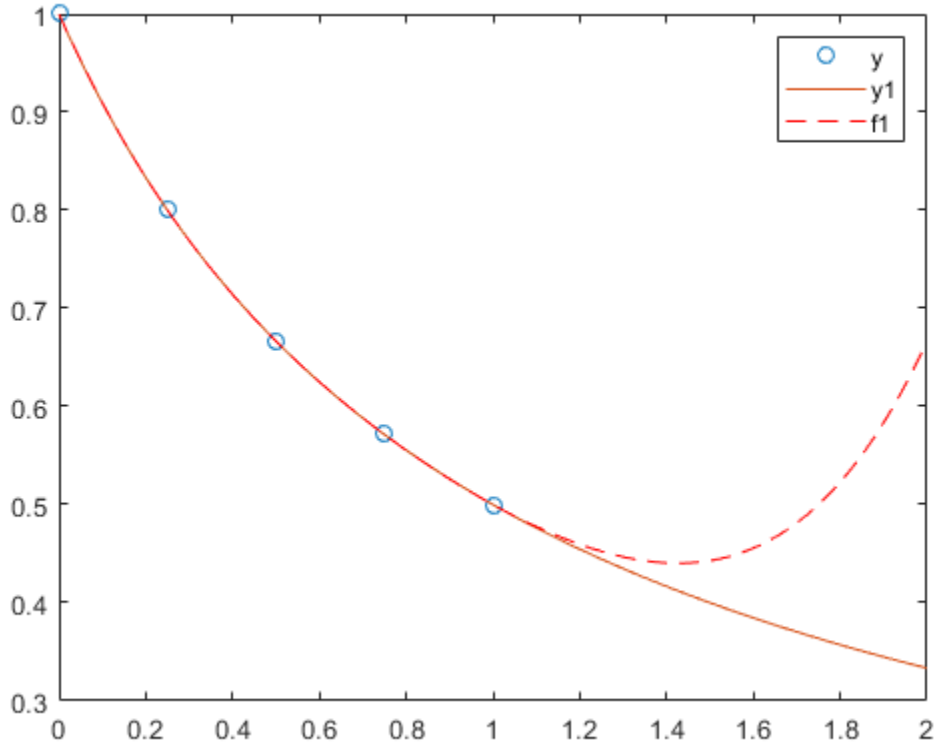
```
p = polyfit(x,y,4);
```

Evaluate the original function and the polynomial fit on a finer grid of points between 0 and 2.

```
x1 = linspace(0,2);
y1 = 1./(1+x1);
f1 = polyval(p,x1);
```

Plot the function values and the polynomial fit in the wider interval `[0,2]`, with the points used to obtain the polynomial fit highlighted as circles. The polynomial fit is good in the original `[0,1]` interval, but quickly diverges from the fitted function outside of that interval.

```
figure
plot(x,y,'o')
hold on
plot(x1,y1)
plot(x1,f1,'r--')
legend('y','y1','f1')
```



Fit Polynomial to Error Function

First generate a vector of `x` points, equally spaced in the interval `[0,2.5]`, and then evaluate `erf(x)` at those points.

```
x = (0:0.1:2.5)';
y = erf(x);
```

Determine the coefficients of the approximating polynomial of degree 6.

```
p = polyfit(x,y,6)
```

```
p = 1×7

    0.0084   -0.0983    0.4217   -0.7435    0.1471    1.1064    0.0004
```

To see how good the fit is, evaluate the polynomial at the data points and generate a table showing the data, fit, and error.

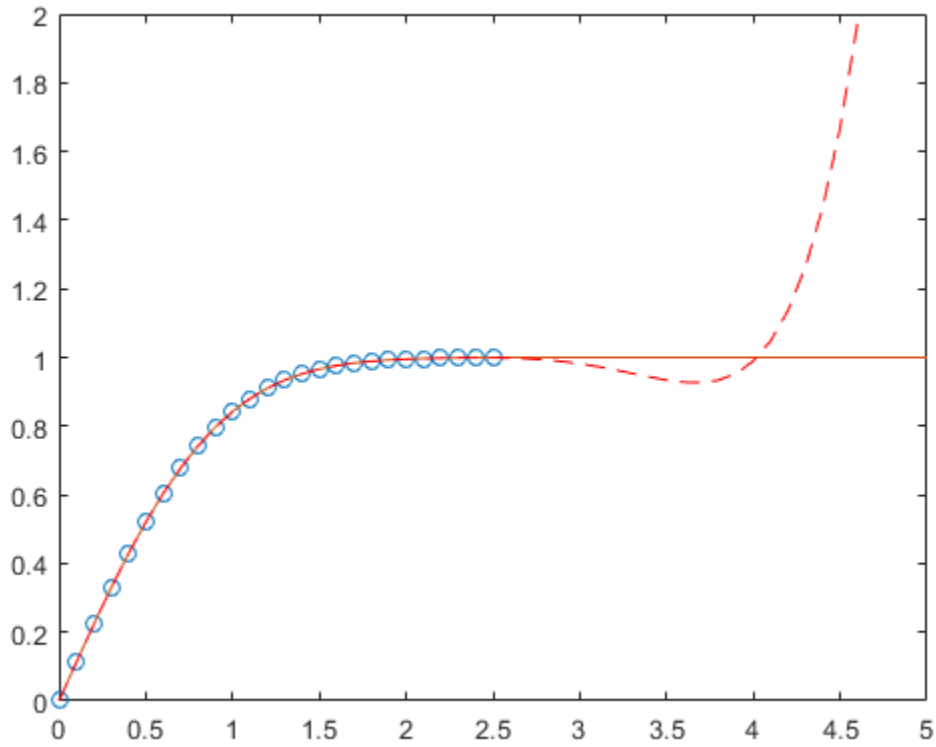
```
f = polyval(p,x);
T = table(x,y,f,y-f,'VariableNames',{'X','Y','Fit','FitError'})
```

```
T=26×4 table
     X        Y          Fit         FitError
    ___    _______    __________    __________

      0          0    0.00044117    -0.00044117
    0.1    0.11246       0.11185     0.00060836
    0.2     0.2227       0.22231     0.00039189
    0.3    0.32863       0.32872    -9.7429e-05
    0.4    0.42839        0.4288    -0.00040661
    0.5     0.5205       0.52093    -0.00042568
    0.6    0.60386       0.60408    -0.00022824
    0.7     0.6778       0.67775     4.6383e-05
    0.8     0.7421       0.74183     0.00026992
    0.9    0.79691       0.79654     0.00036515
      1     0.8427       0.84238      0.0003164
    1.1    0.88021       0.88005     0.00015948
    1.2    0.91031       0.91035    -3.9919e-05
    1.3    0.93401       0.93422      -0.000211
    1.4    0.95229       0.95258    -0.00029933
    1.5    0.96611       0.96639    -0.00028097
      ⋮
```

In this interval, the interpolated values and the actual values agree fairly closely. Create a plot to show how outside this interval, the extrapolated values quickly diverge from the actual data.

```
x1 = (0:0.1:5)';
y1 = erf(x1);
f1 = polyval(p,x1);
figure
plot(x,y,'o')
hold on
plot(x1,y1,'-')
```

```
plot(x1,f1,'r--')
axis([0  5  0  2])
hold off
```



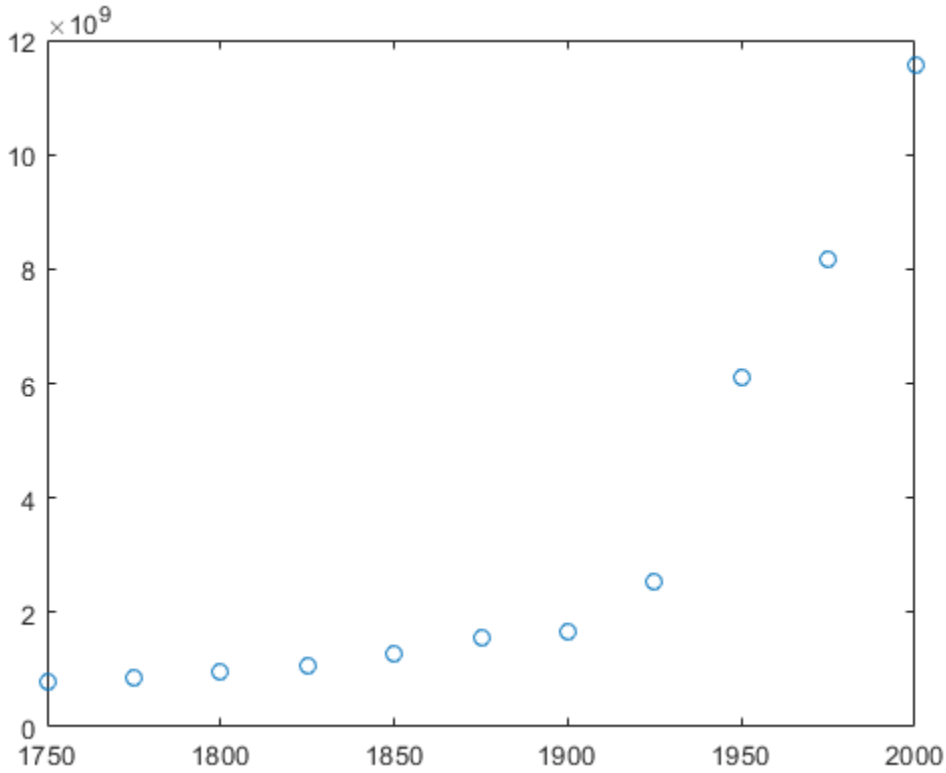
Use Centering and Scaling to Improve Numerical Properties

Create a table of population data for the years 1750 - 2000 and plot the data points.

```
year = (1750:25:2000)';
pop = 1e6*[791 856 978 1050 1262 1544 1650 2532 6122 8170 11560]';
T = table(year, pop)
```

```
T=11×2 table
    year       pop
    ____    _________

    1750     7.91e+08
    1775     8.56e+08
    1800     9.78e+08
    1825     1.05e+09
    1850    1.262e+09
    1875    1.544e+09
    1900     1.65e+09
    1925    2.532e+09
    1950    6.122e+09
    1975     8.17e+09
    2000    1.156e+10
```

```
plot(year,pop,'o')
```

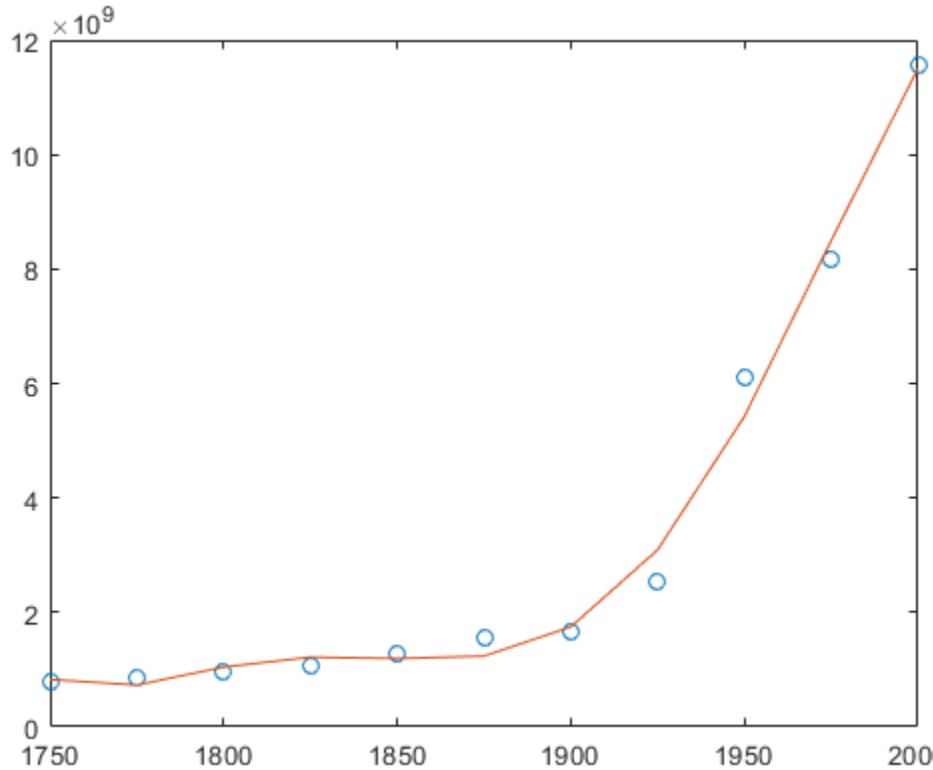


Use `polyfit` with three outputs to fit a 5th-degree polynomial using centering and scaling, which improves the numerical properties of the problem. `polyfit` centers the data in `year` at 0 and scales it to have a standard deviation of 1, which avoids an ill-conditioned Vandermonde matrix in the fit calculation.

```
[p,~,mu] = polyfit(T.year, T.pop, 5);
```

Use `polyval` with four inputs to evaluate `p` with the scaled years, `(year-mu(1))/mu(2)`. Plot the results against the original years.

```
f = polyval(p,year,[],mu);
hold on
plot(year,f)
hold off
```



Simple Linear Regression

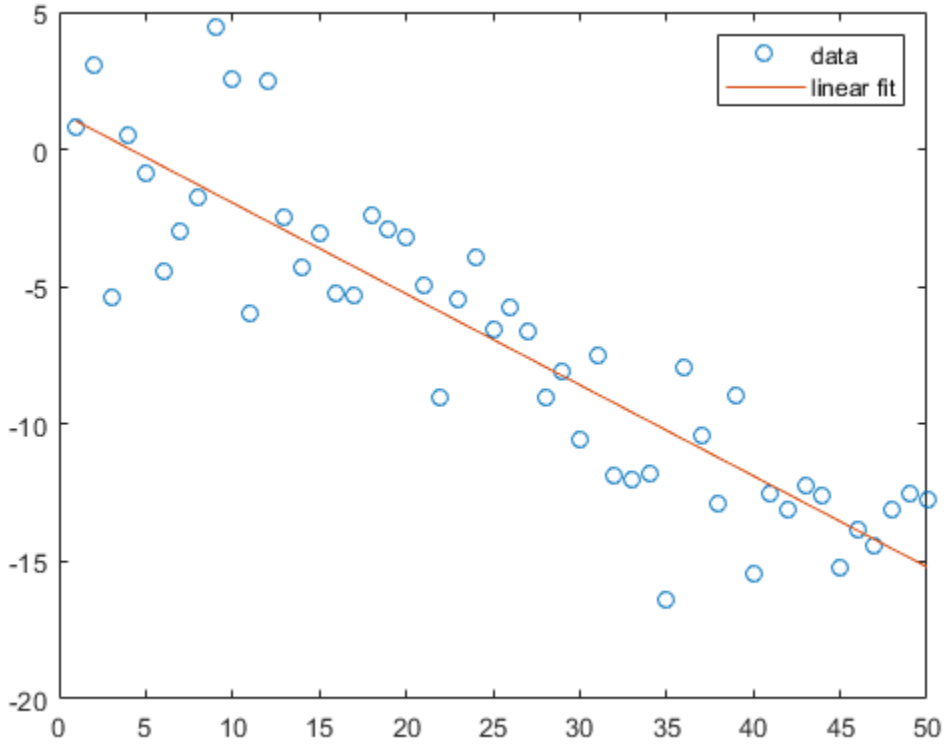
Fit a simple linear regression model to a set of discrete 2-D data points.

Create a few vectors of sample data points *(x,y)*. Fit a first degree polynomial to the data.

```
x = 1:50;
y = -0.3*x + 2*randn(1,50);
p = polyfit(x,y,1);
```

Evaluate the fitted polynomial `p` at the points in `x`. Plot the resulting linear regression model with the data.

```
f = polyval(p,x);
plot(x,y,'o',x,f,'-')
legend('data','linear fit')
```



Linear Regression With Error Estimate

Fit a linear model to a set of data points and plot the results, including an estimate of a 95% prediction interval.

Create a few vectors of sample data points *(x,y)*. Use `polyfit` to fit a first degree polynomial to the data. Specify two outputs to return the coefficients for the linear fit as well as the error estimation structure.

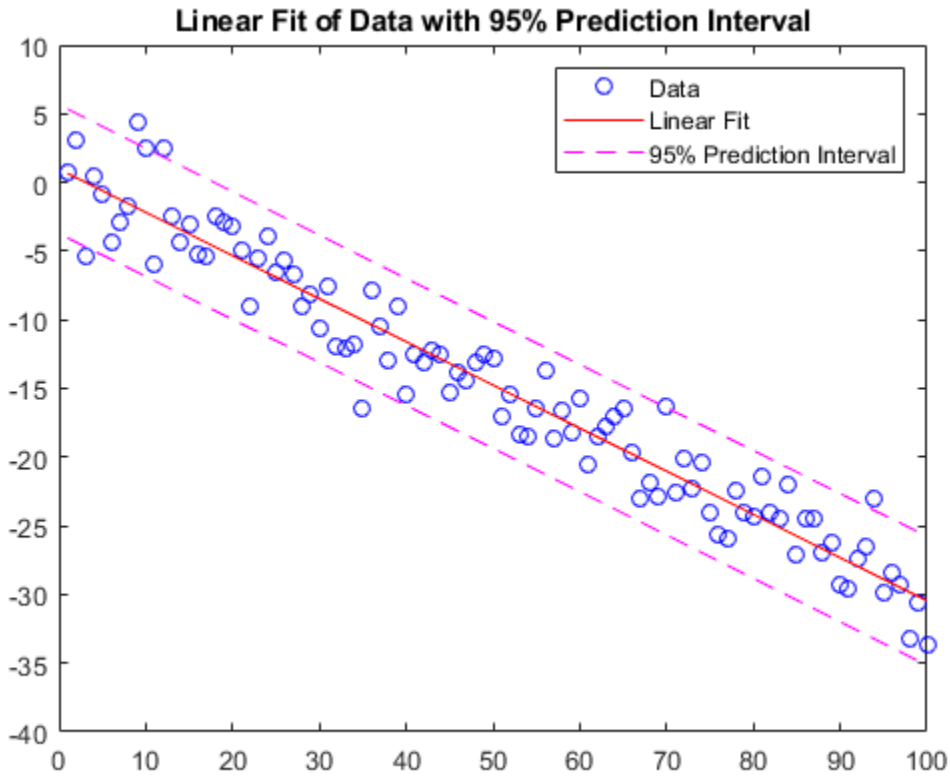
```
x = 1:100;
y = -0.3*x + 2*randn(1,100);
[p,S] = polyfit(x,y,1);
```

Evaluate the first-degree polynomial fit in `p` at the points in `x`. Specify the error estimation structure as the third input so that `polyval` calculates an estimate of the standard error. The standard error estimate is returned in `delta`.

```
[y_fit,delta] = polyval(p,x,S);
```

Plot the original data, linear fit, and 95% prediction interval $y \pm 2\Delta$.

```
plot(x,y,'bo')
hold on
plot(x,y_fit,'r-')
plot(x,y_fit+2*delta,'m--',x,y_fit-2*delta,'m--')
title('Linear Fit of Data with 95% Prediction Interval')
legend('Data','Linear Fit','95% Prediction Interval')
```



Input Arguments

x — Query points
vector

Query points, specified as a vector. The points in `x` correspond to the fitted function values contained in `y`. If `x` is not a vector, then `polyfit` converts it into a column vector `x(:)`.

Warning messages result when `x` has repeated (or nearly repeated) points or if `x` might need centering and scaling.

Data Types: `single` | `double`
Complex Number Support: Yes

y — Fitted values at query points
vector

Fitted values at query points, specified as a vector. The values in `y` correspond to the query points contained in `x`. If `y` is not a vector, then `polyfit` converts it into a column vector `y(:)`.

Data Types: `single` | `double`
Complex Number Support: Yes

n — Degree of polynomial fit
positive integer scalar

Degree of polynomial fit, specified as a positive integer scalar. `n` specifies the polynomial power of the left-most coefficient in `p`.

Output Arguments

p — Least-squares fit polynomial coefficients
vector

Least-squares fit polynomial coefficients, returned as a vector. `p` has length `n+1` and contains the polynomial coefficients in descending powers, with the highest power being `n`. If either `x` or `y` contain `NaN` values and `n < length(x)`, then all elements in `p` are `NaN`.

Use `polyval` to evaluate `p` at query points.

S — Error estimation structure
structure

Error estimation structure. This optional output structure is primarily used as an input to the `polyval` function to obtain error estimates. `S` contains the following fields:

Field	Description
R	Triangular factor from a QR decomposition of the Vandermonde matrix of `x`

Field	Description
`df`	Degrees of freedom
`normr`	Norm of the residuals

If the data in `y` is random, then an estimate of the covariance matrix of `p` is `(Rinv*Rinv')*normr^2/df`, where `Rinv` is the inverse of `R`.

If the errors in the data in `y` are independent and normal with constant variance, then `[y,delta] = polyval(...)` produces error bounds that contain at least 50% of the predictions. That is, `y` ± `delta` contains at least 50% of the predictions of future observations at `x`.

`mu` — Centering and scaling values
two-element vector

Centering and scaling values, returned as a two-element vector. `mu(1)` is `mean(x)`, and `mu(2)` is `std(x)`. These values center the query points in `x` at zero with unit standard deviation.

Use `mu` as the fourth input to `polyval` to evaluate `p` at the scaled points, `(x - mu(1))/mu(2)`.

Limitations

- In problems with many points, increasing the degree of the polynomial fit using `polyfit` does not always result in a better fit. High-order polynomials can be oscillatory between the data points, leading to a *poorer* fit to the data. In those cases, you might use a low-order polynomial fit (which tends to be smoother between points) or a different technique, depending on the problem.
- Polynomials are unbounded, oscillatory functions by nature. Therefore, they are not well-suited to extrapolating bounded data or monotonic (increasing or decreasing) data.

Algorithms

`polyfit` uses `x` to form Vandermonde matrix `V` with `n+1` columns and `m = length(x)` rows, resulting in the linear system

$$\begin{pmatrix} x_1^n & x_1^{n-1} & \cdots & 1 \\ x_2^n & x_2^{n-1} & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ x_m^n & x_m^{n-1} & \cdots & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_{n+1} \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix},$$

which `polyfit` solves with `p = V\y`. Since the columns in the Vandermonde matrix are powers of the vector `x`, the condition number of `V` is often large for high-order fits, resulting in a singular coefficient matrix. In those cases centering and scaling can improve the numerical properties of the system to produce a more reliable fit.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

`X` and `Y` must be column vectors.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`cov` | `lscov` | `poly` | `polyder` | `polyint` | `polyval` | `roots`

Topics

“Programmatic Fitting”

Introduced before R2006a

Polygon Properties

Polygon appearance and behavior

Description

`Polygon` properties control the appearance and behavior of a `Polygon` object. By changing property values, you can modify certain aspects of the polygon.

```
ps = polyshape([0 0 1 1],[1 0 0 1]);
pg = plot(ps);
pg.FaceAlpha = 0.5;
lw = pg.LineWidth;
```

Properties

Color and Styling

`FaceColor` — Filled region color

`[0.85 0.85 0.85]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Filled region color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `pg.FaceColor = [0.1 0.5 0.2];`

Example: `pg.FaceColor = 'white';`

Example: `pg.FaceColor = 'none';`

`EdgeColor` — Color of outer boundaries

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of the outer boundaries of the filled regions, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default color of `[0 0 0]` corresponds to black boundaries.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `pg.EdgeColor = [0.1 0.5 0.2];`

Example: `pg.EdgeColor = 'white';`

Example: `pg.EdgeColor = 'none';`

`HoleEdgeColor` — Color of interior boundaries

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of the interior boundaries of the filled regions, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. If you do not specify the interior boundary color, then it uses the same color as the `EdgeColor` property. The default color of `[0 0 0]` corresponds to black boundaries.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `pg.HoleEdgeColor = [0.1 0.5 0.2];`

Example: `pg.HoleEdgeColor = 'white';`

Example: `pg.HoleEdgeColor = '#D9A2E9';`

Example: `pg.HoleEdgeColor = 'none';`

FaceAlpha — Filled region transparency
1 (default) | scalar in range `[0,1]`

Filled region transparency, specified as a scalar in the range `[0,1]`. A value of 1 is opaque, whereas 0 is completely transparent. Values between 0 and 1 are semitransparent.

Example: `pg.FaceAlpha = 0.5;`

`EdgeAlpha` — Transparency of outer boundaries
`1` (default) | scalar in range `[0,1]`

Transparency of the outer boundaries of the filled regions, specified as a scalar in the range `[0,1]`. A value of 1 is opaque, whereas 0 is completely transparent. Values between 0 and 1 are semitransparent.

Example: `pg.EdgeAlpha = 0.5;`

`HoleEdgeAlpha` — Transparency of interior boundaries
`1` (default) | scalar in range `[0,1]`

Transparency of the interior boundaries of the filled regions, specified as a scalar in the range `[0,1]`. A value of 1 is opaque, whereas 0 is completely transparent. Values between 0 and 1 are semitransparent.

Example: `pg.HoleEdgeAlpha = 0.5;`

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points.

Example: `pg.LineWidth = 0.75;`

`AlignVertexCenters` — Sharp vertical and horizontal lines

`'off'` (default) | `'on'`

Sharp vertical and horizontal lines, specified as `'off'` or `'on'`.

If the associated figure has a `GraphicsSmoothing` property set to `'on'` and a `Renderer` property set to `'opengl'`, then the figure applies a smoothing technique to plots. In some cases, this smoothing technique can cause vertical and horizontal lines to appear uneven in thickness or color. Use the `AlignVertexCenters` property to eliminate the uneven appearance.

- `'off'` — Do not sharpen vertical or horizontal lines. The lines might appear uneven in thickness or color.
- `'on'` — Sharpen vertical and horizontal lines to eliminate an uneven appearance.

Note You must have a graphics card that supports this feature. To see if the feature is supported, call the `rendererinfo` function. If it is supported, `rendererinfo` returns value of `1` for `info.Details.SupportsAlignVertexCenters`.

Shape

Shape — Polygon geometry

`polyshape` object

Polygon geometry, specified as a `polyshape` object. To create a `polyshape` object, use the `polyshape` function.

Legend

`DisplayName` — Legend label

`''` (default) | character vector | string scalar

Legend label, specified as a character vector or string scalar. The legend does not display until you call the `legend` command. If you do not specify the text, then `legend` sets the label using the form `'dataN'`.

`Annotation` — Control for including or excluding object from legend

`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`DataTipTemplate` — Data tip content

`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

This property applies only to polygons with pinned data tips.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

UIContextMenu — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

Selected — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

SelectionHighlight — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

Clipping — Clipping of object to axes limits

`'on'` (default) | `'off'`

Clipping of the object to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the object that are outside the axes limits.
- `'off'` — Display the entire object, even if parts of it appear outside the axes limits. Parts of the object might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the object so that it is larger than the original plot.

The `Clipping` property of the axes that contains the object must be set to `'on'`. Otherwise, this property has no effect. For more information about the clipping behavior, see the `Clipping` property of the axes.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.

- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks

`'all'` (default) | `'visible'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'all'` — Can capture mouse clicks regardless of visibility. The `Visible` property can be set to `'on'` or `'off'` and you can click a part of the `Polygon` object that has no color. The `HitTest` property determines if the `Polygon` object responds to the click or if an ancestor does.
- `'visible'` — Can capture mouse clicks when visible. The `Visible` property must be set to `'on'` and you must click a part of the `Polygon` object that has a defined color. You cannot click a part that has an associated color property set to `'none'`. The `HitTest` property determines if the `Polygon` object responds to the click or if an ancestor does.

- `'none'` — Cannot capture mouse clicks. Clicking the `Polygon` object passes the click through it to the object below it in the current view of the figure window. The `HitTest` property has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `Polygon` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `Polygon` object that has one of these:
  - `HitTest` property set to `'on'`
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

Note The `PickableParts` property determines if the `Polygon` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent
`Axes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `Group`, or `Transform` object.

`Children` — Children

empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the chart object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'polygon'`

This property is read-only.

Type of graphics object, returned as `'polygon'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`polyshape`

Introduced in R2017b

polyint

Polynomial integration

Syntax

```
q = polyint(p,k)
q = polyint(p)
```

Description

`q = polyint(p,k)` returns the integral of the polynomial represented by the coefficients in `p` using a constant of integration `k`.

`q = polyint(p)` assumes a constant of integration `k = 0`.

Examples

Integrate Quartic Polynomial

Evaluate the definite integral

$$I = \int_{-1}^{3} \left(3x^4 - 4x^2 + 10x - 25\right) dx \, .$$

Create a vector to represent the polynomial integrand $3x^4 - 4x^2 + 10x - 25$. The x^3 term is absent and thus has a coefficient of 0.

```
p = [3 0 -4 10 -25];
```

Use `polyint` to integrate the polynomial using a constant of integration equal to `0`.

```
q = polyint(p)
```

```
q = 1×6
```

```
    0.6000         0   -1.3333    5.0000  -25.0000         0
```

Find the value of the integral by evaluating `q` at the limits of integration.

```
a = -1;
b = 3;
I = diff(polyval(q,[a b]))
```

```
I = 49.0667
```

Integrate Product of Two Polynomials

Evaluate

$$I = \int_0^2 \left(x^5 - x^3 + 1\right)\left(x^2 + 1\right)dx$$

Create vectors to represent the polynomials $p(x) = x^5 - x^3 + 1$ and $v(x) = x^2 + 1$.

```
p = [1 0 -1 0 0 1];
v = [1 0 1];
```

Multiply the polynomials and integrate the resulting expression using a constant of integration `k = 3`.

```
k = 3;
q = polyint(conv(p,v),k)
```

```
q = 1×9

    0.1250         0         0         0   -0.2500    0.3333         0    1.0000    3.0
```

Find the value of `I` by evaluating `q` at the limits of integration.

```
a = 0;
b = 2;
I = diff(polyval(q,[a b]))
```

```
I = 32.6667
```

Input Arguments

p — Polynomial coefficients
vector

Polynomial coefficients, specified as a vector. For example, the vector `[1 0 1]` represents the polynomial $x^2 + 1$, and the vector `[3.13 -2.21 5.99]` represents the polynomial $3.13x^2 - 2.21x + 5.99$.

For more information, see “Create and Evaluate Polynomials”.

Data Types: `single` | `double`
Complex Number Support: Yes

k — Constant of integration
numeric scalar

Constant of integration, specified as a numeric scalar.

Example: `polyint([1 0 0],3)`

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

q — Integrated polynomial coefficients
row vector

Integrated polynomial coefficients, returned as a row vector. For more information, see “Create and Evaluate Polynomials”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`polyder` | `polyfit` | `polyval` | `polyvalm`

Topics

“Analytic Solution to Integral of Polynomial”
“Create and Evaluate Polynomials”
“Integrate and Differentiate Polynomials”

Introduced before R2006a

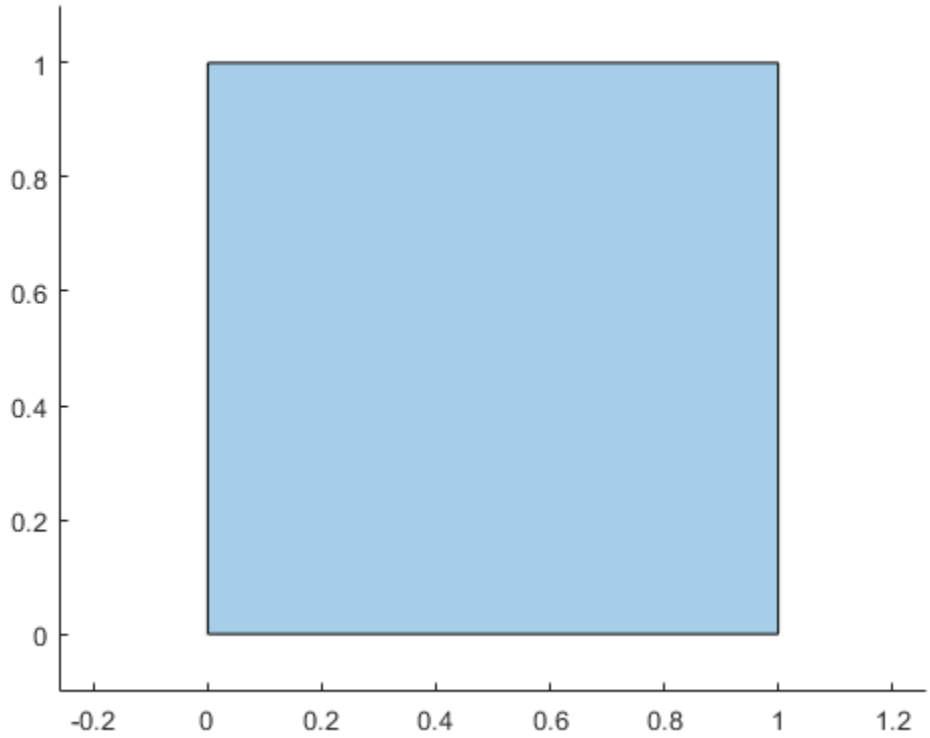
polyshape

2-D polygons

Description

The `polyshape` function creates a polygon defined by 2-D vertices, and returns a `polyshape` object with properties describing its vertices, solid regions, and holes. For example, `pgon = polyshape([0 0 1 1],[1 0 0 1])` creates the solid square defined by the four points (0,1), (0,0), (1,0), and (1,1).

```
>> pgon = polyshape([0 0 1 1],[1 0 0 1])

pgon =

  polyshape with properties:

      Vertices: [4×2 double]
    NumRegions: 1
      NumHoles: 0

>> plot(pgon)
```



Creation

When creating a `polyshape` object, the input vertices should define one or more boundaries of a polygon that have no intersections and are properly nested. Otherwise, the `polyshape` function automatically alters the input vertices as needed to create a well-defined polygon.

If you want to create a regular polygon, you can also use the `nsidedpoly` function. `nsidedpoly` enables you to optionally specify parameters such as the center point and side length of the polygon.

Syntax

```
pgon = polyshape()
pgon = polyshape(x,y)
pgon = polyshape(P)
pgon = polyshape(X,Y)
pgon = polyshape( ___ ,Name,Value)
```

Description

`pgon = polyshape()` creates an empty `polyshape` object.

`pgon = polyshape(x,y)` creates a `polyshape` from 2-D vertices defined by a vector of *x*-coordinates and a vector of corresponding *y*-coordinates. `x` and `y` must be the same length with at least three elements.

`pgon = polyshape(P)` creates a `polyshape` from the 2-D vertices defined in the *N*-by-2 matrix `P`, where *N* is the number of vertices. The first column of `P` defines the *x*-coordinates, and the second column defines the `y`-coordinates.

`pgon = polyshape(X,Y)`, where `X` and `Y` are 1-by-*M* cell arrays of vectors for the *x*- and `y`-coordinates, creates a polygon consisting of *M* boundaries. Each vector in `X` must have the same length as the corresponding vector in `Y`, but the number of vertices can vary between boundaries.

`pgon = polyshape( ___ ,Name,Value)` specifies additional parameters for creating a `polyshape` object for any of the previous syntaxes.

Input Arguments

x — *x*-coordinates
vector

x-coordinates of polygon vertices, specified as a vector. You can represent the coordinates of multiple boundaries at a time by placing a `NaN` between each boundary. For example, `pgon = polyshape([0 0 1 NaN 1 5 5],[1 0 0 NaN 5 5 1])` creates a `polyshape` made up of two solid triangles.

Numeric input coordinates that are not of type `double` are automatically converted to type `double`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

y — *y*-coordinates
vector

y-coordinates of polygon vertices, specified as a vector. You can represent the coordinates of multiple boundaries at a time by placing a `NaN` between each boundary. For example, `pgon = polyshape([0 0 1 NaN 1 5 5],[1 0 0 NaN 5 5 1])` creates a `polyshape` made up of two solid triangles.

Numeric input coordinates that are not of type `double` are automatically converted to type `double`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

P — 2-D vertices
2-column matrix

2-D vertices of the polygon, specified as a 2-column matrix. The first column of `P` contains the *x*-coordinates of the vertices, and the second column contains the *y*-coordinates. `P` must have at least 3 rows.

You can represent the coordinates of multiple boundaries at a time by placing a `NaN` between each boundary. For example, `pgon = polyshape([1 0; 0 0; 0 1; NaN NaN; 1 5; 5 5; 5 1])` creates a `polyshape` made up of two solid triangles.

Numeric input coordinates that are not of type `double` are automatically converted to type `double`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

X — *x*-coordinates
cell array of vectors

x-coordinates of *M* boundaries, specified as a cell array of vectors. The length of each vector can vary, but must match the length of the corresponding *y* vector.

Numeric input coordinates that are not of type `double` are automatically converted to type `double`.

Data Types: `cell`

Y — *y*-coordinates
cell array of vectors

y-coordinates of *M* boundaries, specified as a cell array of vectors. The length of each vector can vary, but must match the length of the corresponding *y* vector.

Numeric input coordinates that are not of type `double` are automatically converted to type `double`.

Data Types: `cell`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify name and value pair arguments in either order as `Name1,Value1,Name2,Value2`.

Example: `pgon = polyshape(P,'SolidBoundaryOrientation','ccw')`

`SolidBoundaryOrientation` — Boundary orientation
`'auto'` (default) | `'cw'` | `'ccw'`

Boundary orientation, specified as the comma-separated pair consisting of `'SolidBoundaryOrientation'` and one of these values:

- `'auto'` — Automatically determine the boundary type based on boundary nesting.
- `'cw'` — Treat clockwise vertex orientation as a solid boundary.
- `'ccw'` — Treat counterclockwise vertex orientation as a solid boundary.

This parameter is typically specified for consistency purposes when polygon data is produced by other software using a particular convention.

`Simplify` — Vertex alteration
`true` (default) | `false`

Vertex alteration, specified as the comma-separated pair consisting of `'Simplify'` and one of the following:

- `true` — Alter polygon vertices to produce a well-defined polygon when the input vertices produce intersections or improper nesting.
- `false` — Do not alter input vertices regardless of intersections or improper nesting. Computing with ill-defined polygons can lead to inaccurate or unexpected results.

Data Types: `logical`

`KeepCollinearPoints` — Collinear vertices
`false` (default) | `true`

Collinear vertices, specified as the comma-separated pair consisting of `'KeepCollinearPoints'` and one of the following:

- `false` — Remove collinear points when creating the `polyshape` so that it contains the fewest vertices necessary to define the boundaries.
- `true` — Keep all collinear points as vertices when creating the `polyshape`.

After creating a `polyshape`, the value of `'KeepCollinearPoints'` is automatically carried over when you use the `addboundary` or `simplify` object functions.

Data Types: `logical`

Properties

`Vertices` — 2-D vertices
2-column matrix

2-D vertices of the polygon, specified as a 2-column matrix. The vertex matrix contains the vertices for each boundary of the polygon. `NaN` values separate each boundary's set of vertices.

Data Types: `double`

`NumRegions` — Number of regions
scalar integer

This property is read-only.

Number of regions making up the polygon, specified as a scalar integer. A region is an area bounded by an outer boundary, which may contain hole boundaries that lie entirely inside the outer boundary.

Data Types: `double`

`NumHoles` — Number of holes
scalar integer

This property is read-only.

Number of holes in the polygon, specified as a scalar integer.

Data Types: `double`

Object Functions

Modify Polyshape

addboundary	Add polyshape boundary
polybuffer	Create buffer around points, lines, or polyshape objects
rmboundary	Remove polyshape boundary
rmholes	Remove holes in polyshape
rmslivers	Remove polyshape boundary outliers
rotate	Rotate polyshape
scale	Scale polyshape
simplify	Simplify polyshape boundaries
sortboundaries	Sort polyshape boundaries
sortregions	Sort polyshape regions
translate	Translate polyshape

Query and Visualize

boundary	Vertex coordinates of polyshape boundary
holes	Convert polyshape hole boundaries to array of polyshape objects
ishole	Determine if polyshape boundary is a hole
isinterior	Query points inside polyshape
issimplified	Determine if polyshape is well-defined
nearestvertex	Query nearest polyshape vertex
numboundaries	Number of polyshape boundaries
numsides	Number of polyshape sides
overlaps	Determine whether polyshape objects overlap
plot	Plot polyshape
regions	Access polyshape regions

Geometric Quantities

area	Area of polyshape
boundingbox	Bounding box of polyshape

centroid	Centroid of polyshape
convhull	Convex hull of polyshape
perimeter	Perimeter of polyshape
triangulation	Triangulate polyshape
turningdist	Compute turning distance between polyshape objects

Boolean Operations

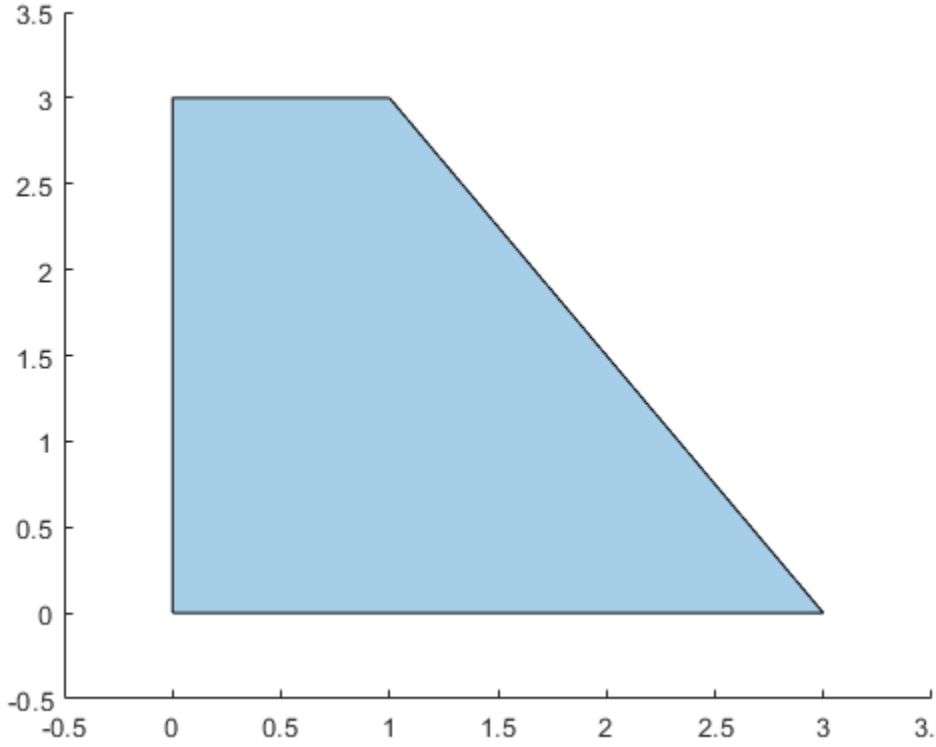
intersect	Intersection of polyshape objects
subtract	Difference of two polyshape objects
union	Union of polyshape objects
xor	Exclusive OR of two polyshape objects

Examples

Simple Quadrilateral

Create and plot a polygon made up of four points, and compute its area, perimeter, and centroid coordinates.

```
pgon = polyshape([0 0 1 3], [0 3 3 0]);
plot(pgon)
```




```
A = area(pgon)
```

```
A = 6
```

```
P = perimeter(pgon)
```

```
P = 10.6056
```

```
[Cx Cy] = centroid(pgon)
```

```
Cx = 1.0833
```

```
Cy = 1.2500
```

Area of Bow Tie

Analyze a polygon with intersecting boundaries.

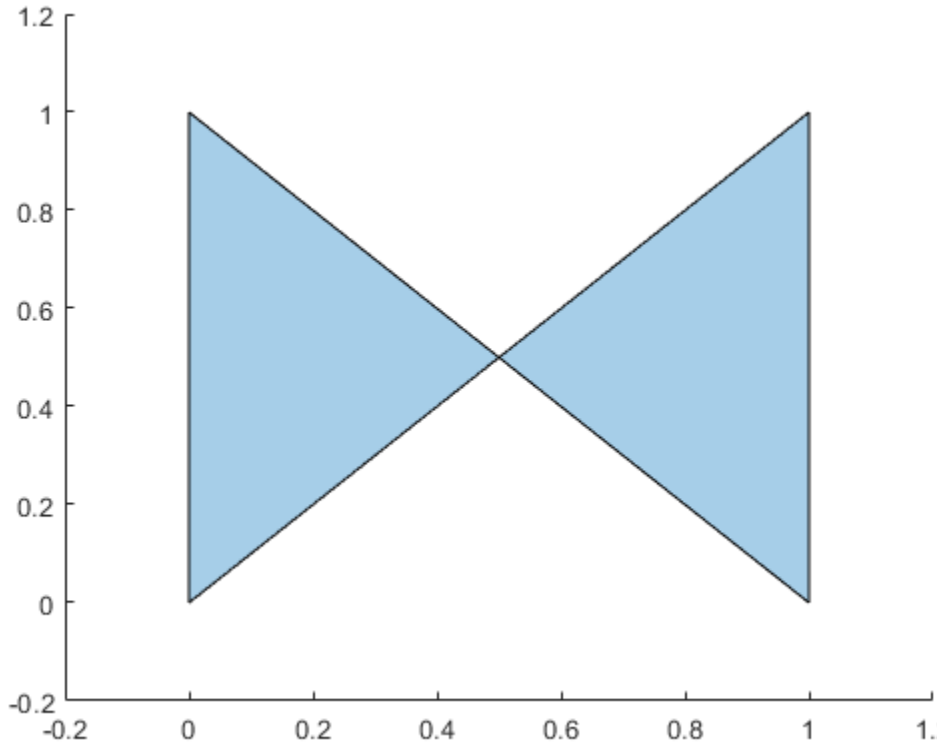
Create a polygon whose boundary contains a self-intersection. By default, the `polyshape` function splits the boundary into two distinct boundaries in order to create a well-defined polygon.

```
P = [0 0; 1 1; 1 0; 0 1; 0 0];
pgon = polyshape(P)
```

```
Warning: Polyshape has duplicate vertices, intersections, or other inconsistencies that

pgon = 
  polyshape with properties:

      Vertices: [7x2 double]
    NumRegions: 2
      NumHoles: 0
```

Plot the polygon and compute its area.

```
plot(pgon)
```




```
A = area(pgon)
```

```
A = 0.5000
```

If you do not simplify the original input vertices, the areas of the two triangular regions of the polygon cancel each other out.

```
pgon2 = polyshape(P,'Simplify',false)
```

```
pgon2 
  polyshape with properties:

      Vertices: [4x2 double]
    NumRegions: 1
```

```
      NumHoles: 0
```

```
A2 = area(pgon2)
```

```
A2 = 0
```

Polygon with Hole

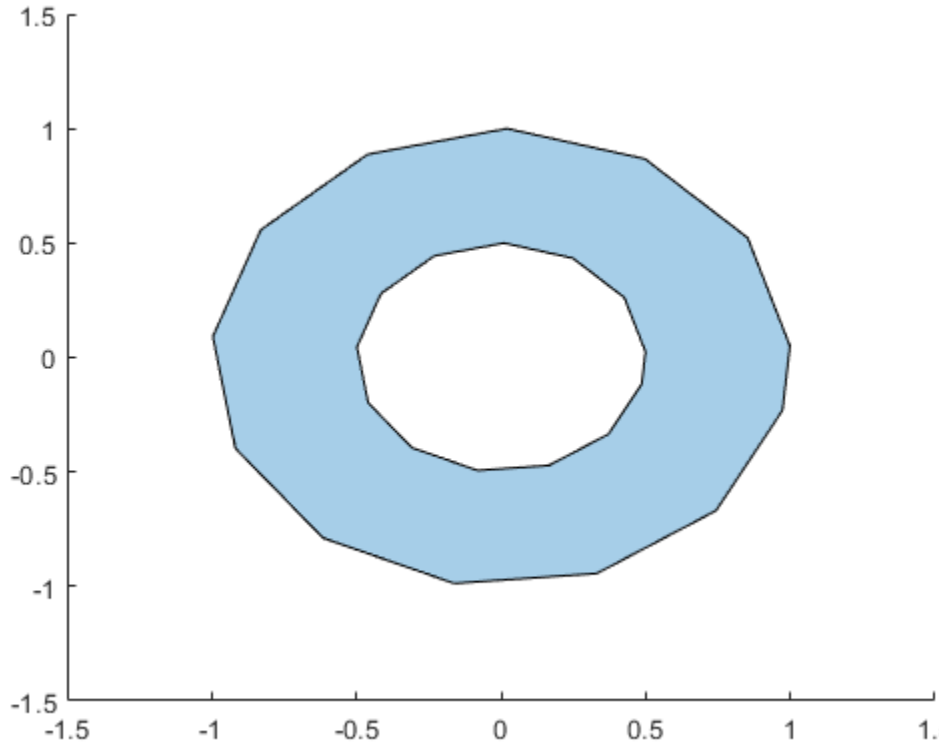
Analyze polygons with nested boundaries.

Create a polygon with two properly nested boundaries. `polyshape` defines the outermost boundary as the exterior bound of a solid region. Working inward, the next boundary defines the start of a hole.

```
t = 0.05:0.5:2*pi;
x1 = cos(t);
y1 = sin(t);
x2 = 0.5*cos(t);
y2 = 0.5*sin(t);
pgon = polyshape({x1,x2},{y1,y2})
```

```
pgon = 
  polyshape with properties:

      Vertices: [27x2 double]
    NumRegions: 1
      NumHoles: 1
```

```
plot(pgon)
```

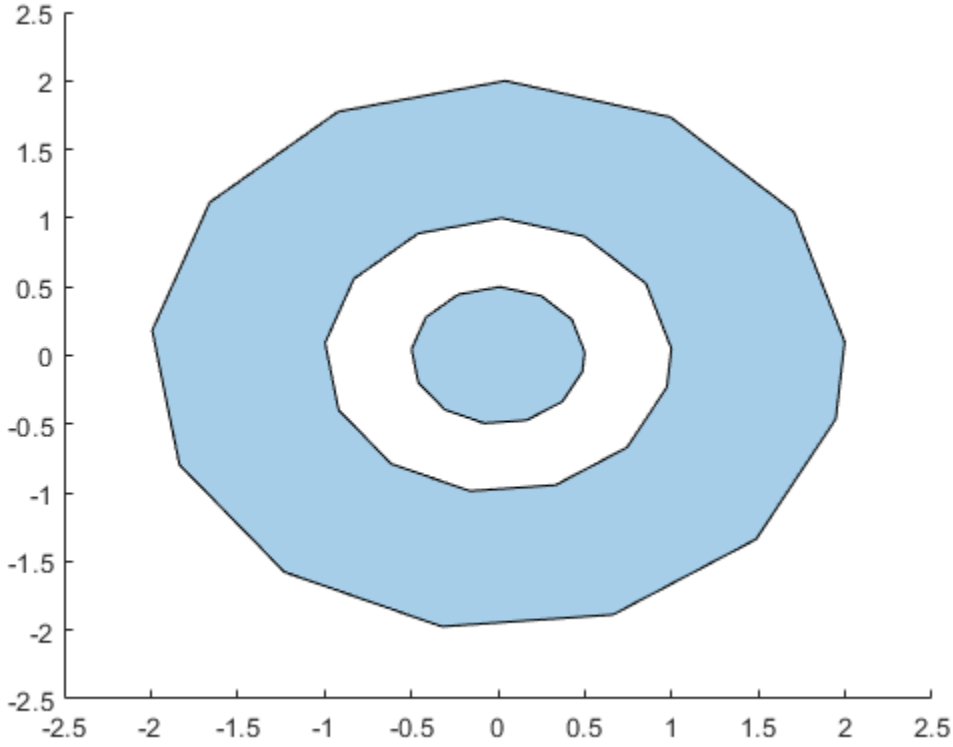


Use the `addboundary` function to create a second polygon that adds a third, outer boundary to `pgon`.

```
x3 = 2*cos(t);
y3 = 2*sin(t);
pgon2 = addboundary(pgon,x3,y3)
```

```
pgon2 = 
  polyshape with properties:

      Vertices: [41x2 double]
    NumRegions: 2
      NumHoles: 1
```

```
plot(pgon2)
```



Adding a third, outer boundary requires `polyshape` to reorganize the solid and hole boundaries based on the new nesting pattern. Again, `polyshape` starts with the outermost boundary that indicates the start of a solid region, then alternates between hole and solid with each nested boundary, working inward. The new polygon now has two solid regions and one hole.

See Also

`alphaShape` | `nsidedpoly`

Introduced in R2017b

addboundary

Add `polyshape` boundary

Syntax

```
polyout = addboundary(polyin,x,y)
polyout = addboundary(polyin,P)
polyout = addboundary(polyin,{x1,x2,...,xM},{y1,y2,...,yM})
polyout = addboundary( ___ ,Name,Value)
```

Description

`polyout = addboundary(polyin,x,y)` returns a `polyshape` object that is made up of an existing `polyshape` plus an additional boundary defined by the *x*-coordinates and the *y*-coordinates contained in the vectors `x` and `y`.

`polyout = addboundary(polyin,P)` adds the boundary defined by the coordinates contained in the *N*-by-2 matrix `P`, where *N* is the number of vertices.

`polyout = addboundary(polyin,{x1,x2,...,xM},{y1,y2,...,yM})` adds *M* boundaries, where the vectors of *x*-coordinates for each boundary are listed together in a cell array. The corresponding vectors of *y*-coordinates also are listed together in a cell array. Each x_i must have the same length as the corresponding y_i, but the number of vertices can vary among the boundaries.

`polyout = addboundary( ___ ,Name,Value)` specifies additional parameters for adding boundaries to a `polyshape` for any of the previous syntaxes.

Examples

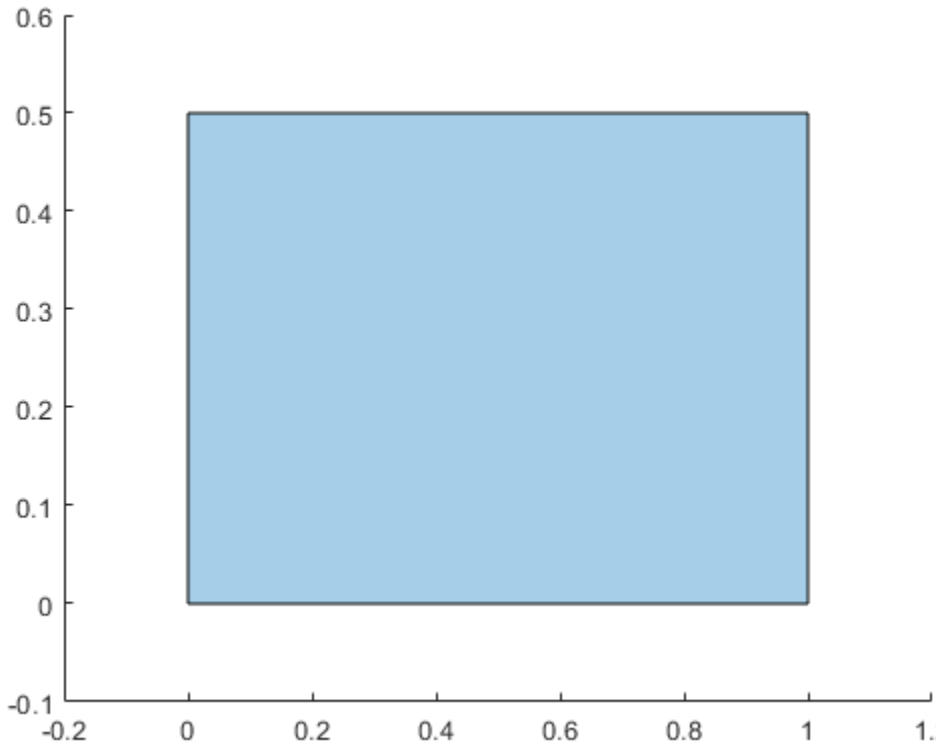
Add Polygon Boundary

Create a rectangle, and then create a second polygon made up of the rectangle plus a triangle.

```
polyin = polyshape([0 0 1 1],[0 0.5 0.5 0])
```

```
polyin = 
  polyshape with properties:

      Vertices: [4x2 double]
    NumRegions: 1
      NumHoles: 0
```

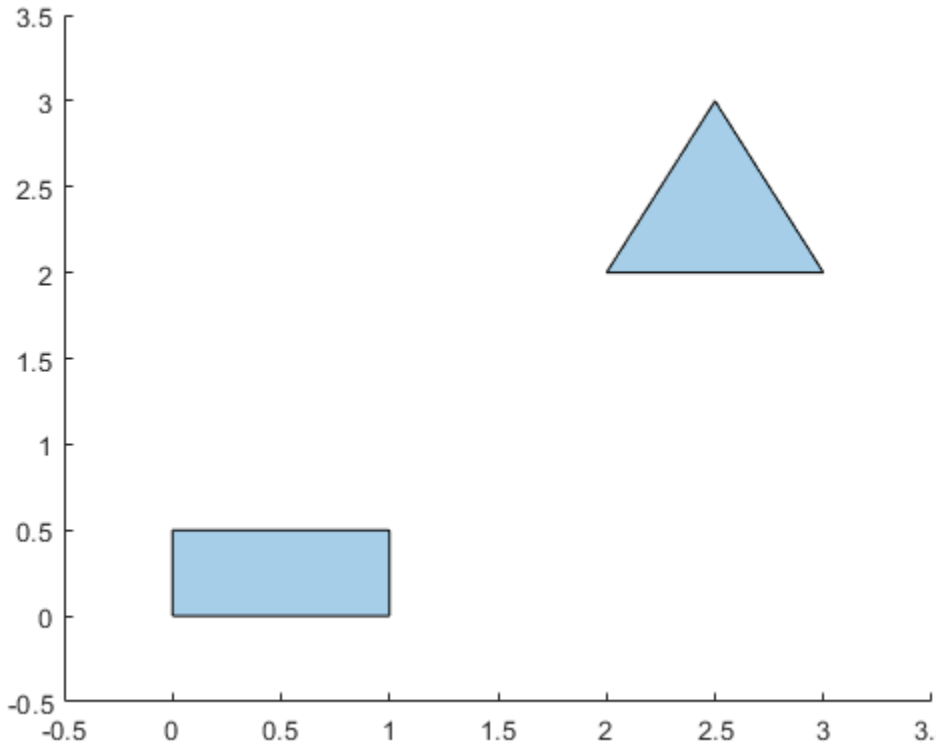
```
plot(polyin)
```




```
polyout = addboundary(polyin,[2 3 2.5],[2 2 3])
```

```
polyout =
  polyshape with properties:

      Vertices: [8x2 double]
    NumRegions: 2
      NumHoles: 0
```

```
plot(polyout)
```



Input Arguments

`polyin` — Input `polyshape`
scalar

Input `polyshape`, specified as a scalar.

Data Types: `polyshape`

`x` — *x*-coordinates
vector

x-coordinates of boundary vertices, specified as a vector. You can represent the coordinates of multiple boundaries simultaneously by placing a `NaN` between each boundary. For example, `polyout = addboundary(polyin,[0 0 1 NaN 1 5 5],[1 0 0 NaN 5 5 1])` returns a `polyshape` object made up of `polyin` plus two additional triangles.

Numeric input vertices that are not of type `double` are automatically converted to type `double`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`y` — *y*-coordinates
vector

y-coordinates of boundary vertices, specified as a vector. You can represent the coordinates of multiple boundaries simultaneously by placing a `NaN` between each boundary. For example, `polyout = addboundary(polyin,[0 0 1 NaN 1 5 5],[1 0 0 NaN 5 5 1])` returns a `polyshape` object made up of `polyin` plus two additional triangles.

Numeric input vertices that are not of type `double` are automatically converted to type `double`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`P` — Boundary vertices
2-column matrix

Boundary vertices, specified as a 2-column matrix. The first column of `P` contains the *x*-coordinates of the vertices, and the second column contains the *y*-coordinates. `P` must have at least 3 rows.

You can represent the coordinates of multiple boundaries simultaneously by placing a `NaN` between each boundary. For example, `polyout = addboundary(polyin,[1 0; 0 0; 0 1; NaN NaN; 1 5; 5 5; 5 1])` returns a `polyshape` object made up of `polyin`, plus two additional triangles.

Numeric input vertices that are not of type `double` are automatically converted to type `double`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`{x1,x2,...,xM}` — *x*-coordinates
cell array of vectors

x-coordinates for *M* boundaries, specified as a cell array of vectors. The length of each x_i can vary, but must match the length of the corresponding y_i vector.

Numeric input vertices that are not of type `double` are automatically converted to type `double`.

Data Types: `cell`

`{y1,y2,...,yM}` — *y*-coordinates
cell array of vectors

y-coordinates of *M* boundaries, specified as a cell array of vectors. The length of each y_i can vary, but must match the length of the corresponding x_i vector.

Numeric input vertices that are not of type `double` are automatically converted to type `double`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Example: `polyout = addboundary(polyin,x,y,'SolidBoundaryOrientation','ccw')`

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`SolidBoundaryOrientation` — Boundary orientation
`'auto'` (default) | `'cw'` | `'ccw'`

Boundary orientation, specified as the comma-separated pair consisting of `'SolidBoundaryOrientation'` and one of these values:

- `'auto'` — Automatically choose vertex orientation as clockwise or counterclockwise for solid boundaries.
- `'cw'` — Treat clockwise vertex orientation as a solid boundary.
- `'ccw'` — Treat counterclockwise vertex orientation as a solid boundary.

The vertex orientation is not important for most applications and is primarily an algorithmic tool for determining boundary nesting. This parameter is typically specified for consistency and efficiency purposes when polygon data is produced by other software using a particular convention.

`Simplify` — Vertex alteration
`true` (default) | `false`

Vertex alteration, specified as the comma-separated pair consisting of `'Simplify'` and one of these values:

- `true` — Alter polygon vertices to produce a well-defined polygon when the input vertices produce intersections or improper nesting.
- `false` — Do not alter input vertices regardless of intersections or improper nesting. Computing with ill-defined polygons can lead to inaccurate or unexpected results.

Data Types: `logical`

`KeepCollinearPoints` — Collinear vertices
`false` | `true`

Collinear vertices, specified as the comma-separated pair consisting of `'KeepCollinearPoints'` and one of the following:

- `false` — Remove collinear points so that the output `polyshape` contains the fewest vertices necessary to define the boundaries.

- `true` — Keep all collinear points as vertices.

When `'KeepCollinearPoints'` is not specified, its value is automatically set to the value used when creating the input `polyshape`.

Data Types: `logical`

See Also

`boundary` | `polyshape` | `rmboundary`

Introduced in R2017b

area

Area of `polyshape`

Syntax

```
A = area(polyin)
A = area(polyin,I)
```

Description

`A = area(polyin)` returns the total area of a `polyshape` object, which is the sum of the areas of the solid regions that make up the `polyshape`.

`A = area(polyin,I)` returns the area of the `I`th boundary of `polyin`. The area of a hole boundary is negative.

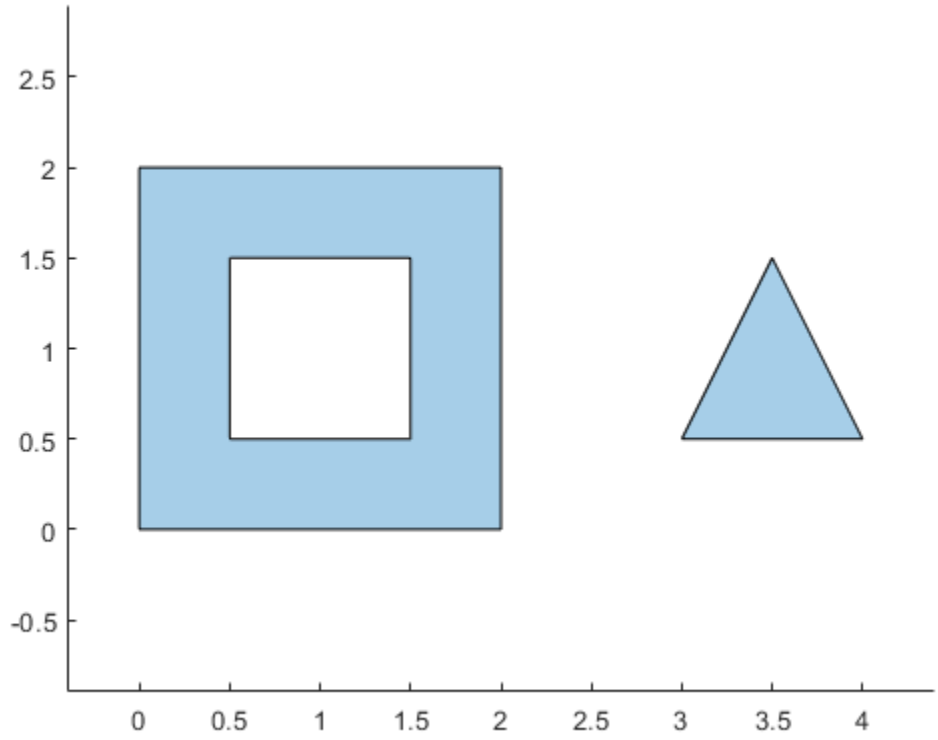
This syntax is only supported when `polyin` is a scalar `polyshape` object.

Examples

Area of Polygon

Create a polygon containing multiple solid regions and one hole, and compute its area.

```
P = [0 0; 0 2; 2 2; 2 0; NaN NaN; 0.5 0.5; 0.5 1.5; 1.5 1.5; 1.5 0.5; NaN NaN; 3 0.5; 3
polyin = polyshape(P);
plot(polyin)
axis equal
```




```
A = area(polyin)
```

```
A = 3.5000
```

Input Arguments

`polyin` — Input `polyshape`
scalar | vector | matrix | multidimensional array

Input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `polyshape`

`I` — Boundary index
scalar integer | vector of integers

Boundary index, specified as a scalar integer or vector of integers. Each element of `I` corresponds to a single boundary of the input `polyshape`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

See Also

`centroid` | `perimeter` | `polyshape`

Introduced in R2017b

boundary

Vertex coordinates of `polyshape` boundary

Syntax

```
[x,y] = boundary(polyin)
[x,y] = boundary(polyin,I)
```

Description

`[x,y] = boundary(polyin)` returns the *x*-coordinates and the *y*-coordinates of each boundary of a `polyshape`. The vectors `x` and `y` contain the list of coordinates for each boundary, delimited by `NaN`. The `boundary` function automatically appends the first vertex coordinates of `polyin` to the end of `x` and `y` to close the polygon.

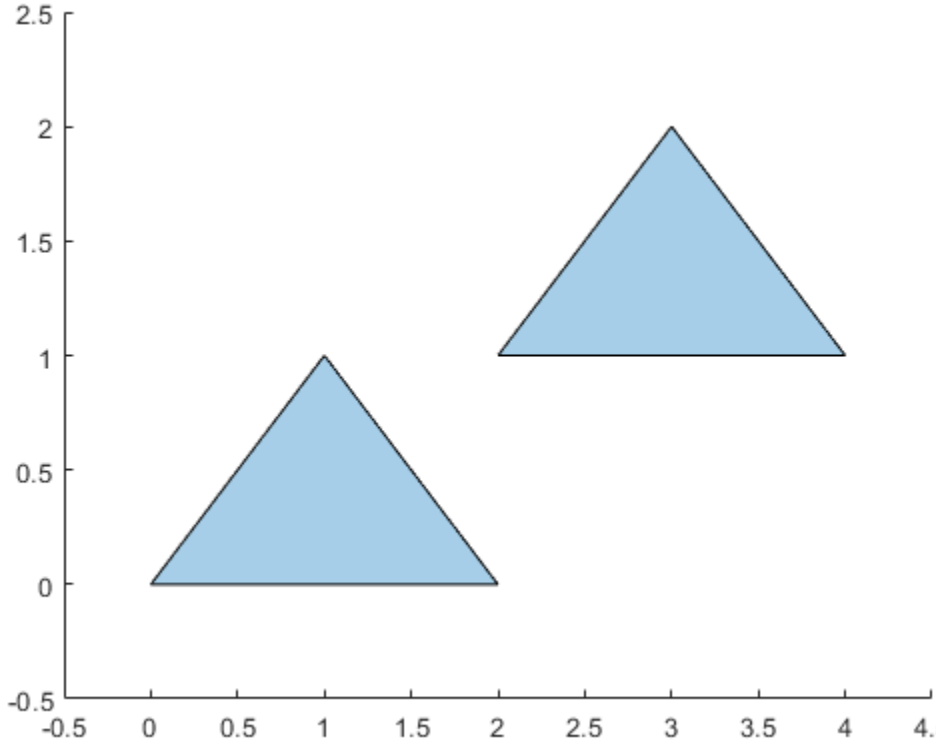
`[x,y] = boundary(polyin,I)` returns the coordinates of the `I`th boundary of a `polyshape`.

Examples

Polygon Boundary Vertices

Create a polygon containing two boundaries, and display the vertex coordinates of both boundaries. The list of vertices for the boundaries are delimited by `NaN`.

```
x1 = [0 1 2];
y1 = [0 1 0];
x2 = [2 3 4];
y2 = [1 2 1];
polyin = polyshape({x1,x2},{y1,y2});
plot(polyin)
```




```
[x,y] = boundary(polyin)
```

```
x = 9×1

     0
     1
     2
     0
   NaN
     2
     3
     4
     2
```

```
y = 9×1

     0
     1
     0
     0
   NaN
     1
     2
     1
     1
```

Input Arguments

polyin — Input polyshape
scalar

Input `polyshape`, specified as a scalar.

Data Types: `polyshape`

I — Boundary index
scalar integer | vector of integers

Boundary index, specified as a scalar integer or vector of integers. Each element of `I` corresponds to a single boundary of the input `polyshape`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

x — *x*-coordinates
column vector

x-coordinates of `polyshape` boundaries, returned as a column vector. When returning multiple boundaries, `boundary` places `NaN` values between each boundary's set of coordinates.

Data Types: `double`

y — *y*-coordinates
column vector

y-coordinates of `polyshape` boundary, returned as a column vector. When returning multiple boundaries, `boundary` places `NaN` values between each boundary's set of coordinates.

Data Types: `double`

See Also

`addboundary` | `numboundaries` | `polyshape` | `rmboundary` | `sortboundaries`

Introduced in R2017b

boundingbox

Bounding box of `polyshape`

Syntax

```
[xlim,ylim] = boundingbox(polyin)
[xlim,ylim] = boundingbox(polyin,I)
```

Description

`[xlim,ylim] = boundingbox(polyin)` returns the *x* and *y* bounds of the smallest rectangle enclosing a `polyshape`. `xlim` and `ylim` are two-element row vectors whose first elements correspond to the lower *x* and *y* bounds, and whose second elements correspond to the upper *x* and *y* bounds.

When `polyin` is an array of `polyshape` objects, `xlim` and `ylim` describe the bounding box enclosing all `polyshape` elements of `polyin`.

`[xlim,ylim] = boundingbox(polyin,I)` returns the bounding box limits of the `I`th boundary of `polyin`.

This syntax is only supported when `polyin` is a scalar `polyshape` object.

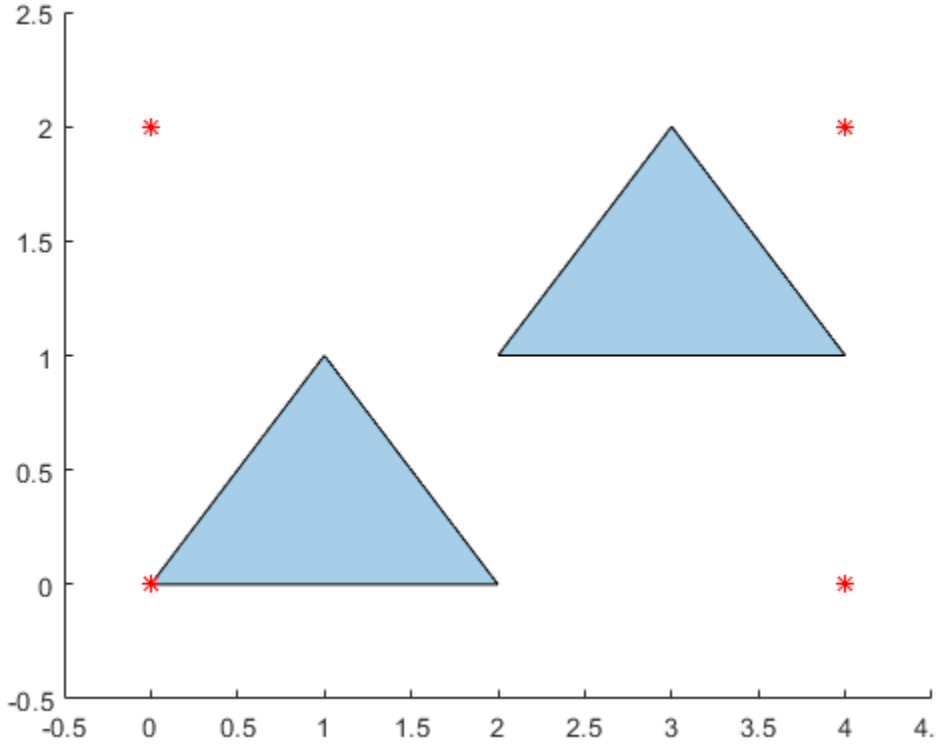
Examples

Bounding Box of Polygon

Create a polygon containing two solid regions, and compute the lower and upper x and y bounds that enclose it.

```
x1 = [0 1 2];
y1 = [0 1 0];
x2 = [2 3 4];
y2 = [1 2 1];
```

```
polyin = polyshape({x1,x2},{y1,y2});
[xlim,ylim] = boundingbox(polyin);
plot(polyin)
hold on
plot(xlim,ylim,'r*',xlim,fliplr(ylim),'r*')
```



Input Arguments

`polyin` — Input `polyshape`

scalar | vector | matrix | multidimensional array

Input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `polyshape`

`I` — Boundary index
scalar integer | vector of integers

Boundary index, specified as a scalar integer or vector of integers. Each element of `I` corresponds to a single boundary of the input `polyshape`.

When `I` is a vector of indices, `boundingbox` returns the limits enclosing all boundaries indexed by the elements of `I`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`xlim` — *x* limits
two-element row vector

x limits bounding a `polyshape`, returned as a two-element row vector. The first element of `xlim` is the lower *x* bound, and the second element is the upper *x* bound.

Data Types: `double`

`ylim` — *y* limits
two-element row vector

y limits bounding a `polyshape`, returned as a two-element row vector. The first element of `ylim` is the lower *y* bound, and the second element is the upper *y* bound.

Data Types: `double`

See Also

`convhull` | `polyshape`

Introduced in R2017b

centroid

Centroid of `polyshape`

Syntax

```
[x,y] = centroid(polyin)
[x,y] = centroid(polyin,I)
```

Description

`[x,y] = centroid(polyin)` returns the *x*-coordinates and the *y*-coordinates of the centroid of a `polyshape`.

`[x,y] = centroid(polyin,I)` returns the coordinates of the centroid of the `I`th boundary of `polyin`.

This syntax is only supported when `polyin` is a scalar `polyshape` object.

Examples

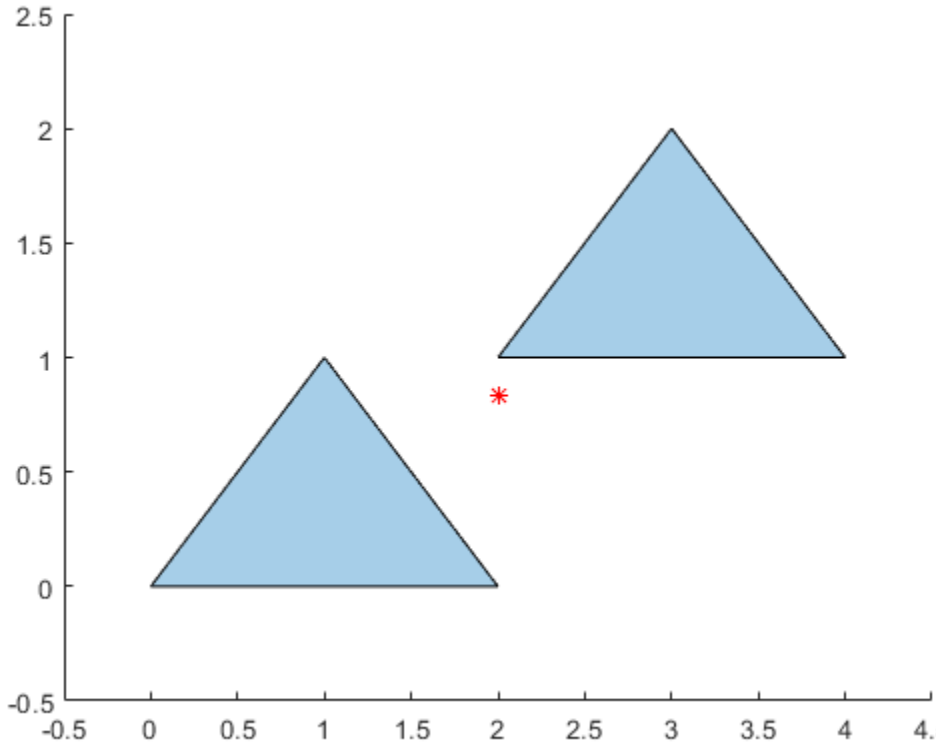
Centroid of Polygon

Compute the centroids associated with a multiregion polygon.

Create a polygon containing two solid regions, and then compute its centroid.

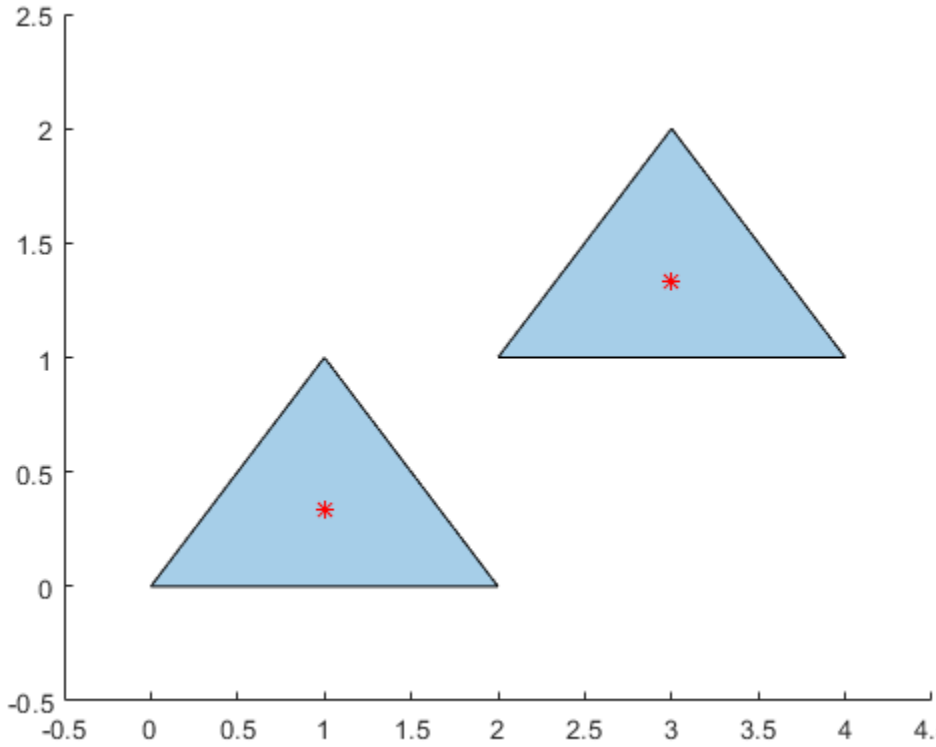
```
x1 = [0 1 2];
y1 = [0 1 0];
x2 = [2 3 4];
y2 = [1 2 1];
polyin = polyshape({x1,x2},{y1,y2});
[x,y] = centroid(polyin);
plot(polyin)
hold on
```

```
plot(x,y,'r*')
hold off
```



To compute the centroid of each region separately, specify the boundary indices of each region in the second argument.

```
[x,y] = centroid(polyin,[1 2]);
plot(polyin)
hold on
plot(x(1),y(1),'r*',x(2),y(2),'r*')
hold off
```



Alternatively, you can compute the centroid of each region by first separating the regions into separate `polyshape` objects. Use the `regions` function to create an array of `polyshape` objects, where each element of the array is a `polyshape` defining one of the triangular regions.

```
polyarray = regions(polyin)
```

```
polyarray = 
  2x1 polyshape array with properties:

    Vertices
    NumRegions
    NumHoles
```

```
[x,y] = centroid(polyarray)
```

```
x = 2×1

     1
     3


y = 2×1

    0.3333
    1.3333
```

Input Arguments

polyin — Input polyshape
scalar | vector | matrix | multidimensional array

Input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `polyshape`

I — Boundary index
scalar integer | vector of integers

Boundary index, specified as a scalar integer or vector of integers. Each element of `I` corresponds to a single boundary of the input `polyshape`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

x — *x*-coordinate
scalar | vector | matrix | multidimensional array

x-coordinate of the centroid of a `polyshape`, returned as a scalar, vector, matrix, or multidimensional array. If the input is an array of `polyshape` objects, then `x` contains the *x*-coordinates of the centroid of each `polyshape`.

y — *y*-coordinate
scalar | vector | matrix | multidimensional array

y-coordinate of the centroid of a `polyshape`, returned as a scalar, vector, matrix, or multidimensional array. If the input is an array of `polyshape` objects, then `y` contains the *y*-coordinates of the centroid of each `polyshape`.

See Also

`area` | `perimeter` | `polyshape`

Introduced in R2017b

convhull

Convex hull of `polyshape`

Syntax

```
polyout = convhull(polyin)
```

Description

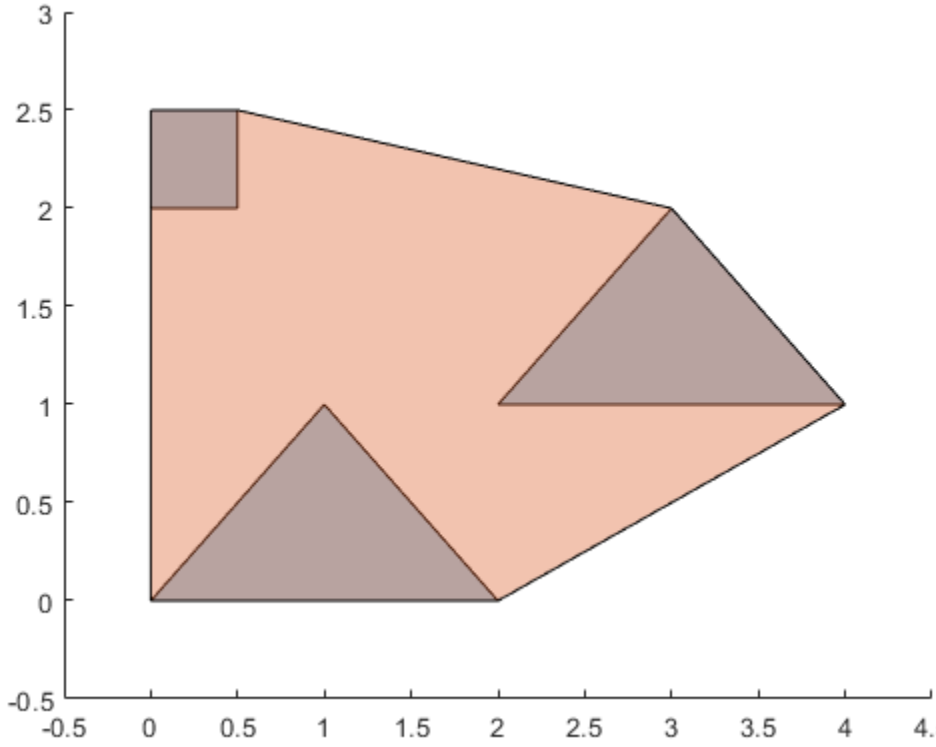
`polyout = convhull(polyin)` returns the convex hull of a `polyshape` object. `polyout` is a `polyshape` object or an array of `polyshape` objects the same size as `polyin`.

Examples

Convex Hull of Polygon

Compute the convex hull of a polygon containing three solid regions.

```
x1 = [0 1 2];
y1 = [0 1 0];
x2 = [2 3 4];
y2 = [1 2 1];
x3 = [0 0.5 0.5 0];
y3 = [2 2 2.5 2.5];
polyin = polyshape({x1,x2,x3},{y1,y2,y3});
plot(polyin)
hold on
polyout = convhull(polyin);
plot(polyout)
```



Input Arguments

`polyin` — Input `polyshape`

scalar | vector | matrix | multidimensional array

Input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `polyshape`

See Also

`boundingbox` | `polyshape`

Introduced in R2017b

holes

Convert `polyshape` hole boundaries to array of `polyshape` objects

Syntax

```
polyout = holes(polyin)
```

Description

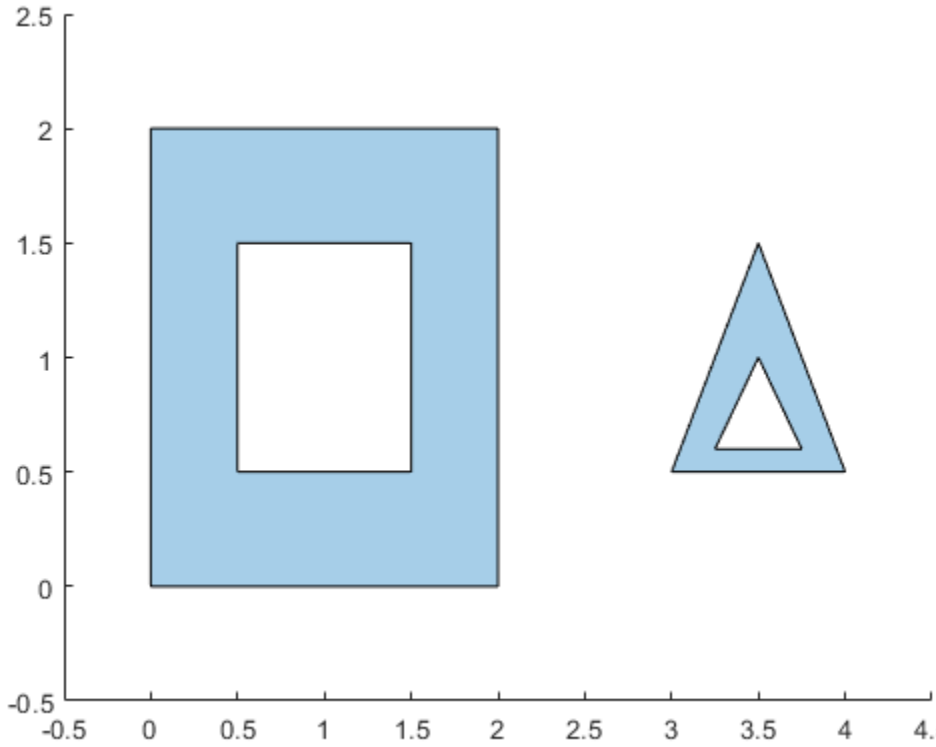
`polyout = holes(polyin)` returns the hole boundaries of a `polyshape` object as an array of `polyshape` objects. The number of elements in `polyout` is equal to the number of hole boundaries in `polyin`.

Examples

Array of Hole Boundaries

Create a polygon containing two solid regions and two holes. Convert the hole boundaries to an array of `polyshape` objects.

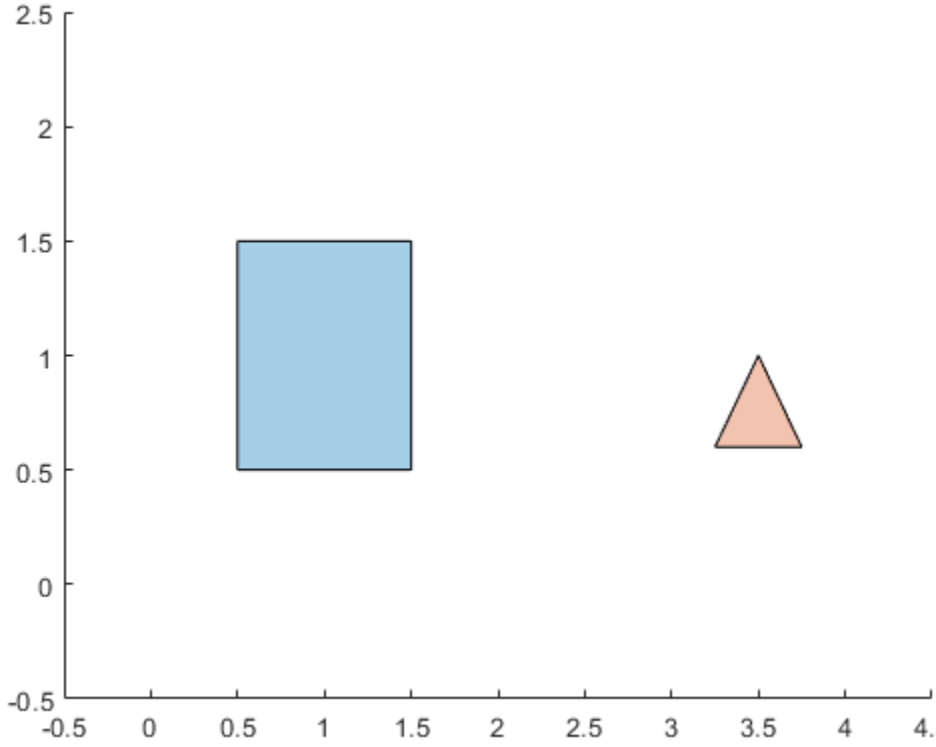
```
P = [0 0; 0 2; 2 2; 2 0; NaN NaN; 0.5 0.5; 0.5 1.5; 1.5 1.5; 1.5 0.5; ...
    NaN NaN; 3 0.5; 3.5 1.5; 4 0.5; NaN NaN; 3.25 0.6; 3.75 0.6; 3.5 1];
polyin = polyshape(P);
plot(polyin)
xlim([-0.5 4.5])
ylim([-0.5 2.5])
```




```
polyout = holes(polyin)
```

```
polyout =
  2x1 polyshape array with properties:

    Vertices
    NumRegions
    NumHoles
```

```
plot(polyout)
xlim([-0.5 4.5])
ylim([-0.5 2.5])
```



Input Arguments

`polyin` — Input `polyshape`
scalar

Input `polyshape`, specified as a scalar.

Data Types: `polyshape`

See Also

`boundary` | `ishole` | `polyshape` | `rmholes`

Introduced in R2017b

intersect

Intersection of `polyshape` objects

Syntax

```
polyout = intersect(poly1,poly2)
polyout = intersect(polyvec)
[polyout,shapeID,vertexID] = intersect(poly1,poly2)
[polyout,shapeID,vertexID] = intersect(polyvec)
___ = intersect( ___ ,'KeepCollinearPoints',TF)
[in,out] = intersect(poly1,lineseg)
```

Description

`polyout = intersect(poly1,poly2)` returns a `polyshape` object whose regions are the geometric intersection of two `polyshape` objects. The intersection contains the regions of `poly1` and `poly2` that overlap. `poly1` and `poly2` must have compatible array sizes.

`polyout = intersect(polyvec)` returns a `polyshape` object whose regions are the intersection of all the `polyshape` objects in the vector `polyvec`. The intersection contains the regions where all the `polyshape` objects in `polyvec` overlap.

`[polyout,shapeID,vertexID] = intersect(poly1,poly2)` also returns vertex mapping information from the vertices in `polyout` to the vertices in `poly1` and `poly2`. The `intersect` function only supports this syntax when `poly1` and `poly2` are scalar `polyshape` objects.

The `shapeID` elements identify whether the corresponding vertex in `polyout` originated in `poly1`, `poly2`, or was created from the intersection. `vertexID` maps the vertices of `polyout` to the vertices of `poly1`, `poly2`, or the intersection.

`[polyout,shapeID,vertexID] = intersect(polyvec)` returns vertex mapping information from `polyout` to each element of the vector of `polyshape` objects `polyvec`.

`___ = intersect( ___ ,'KeepCollinearPoints',TF)` specifies whether to keep or remove collinear points in `polyout` for any of the previous syntaxes.

`[in,out] = intersect(poly1,lineseg)` returns the line segments of `lineseg` that are inside and outside of `poly1`. The matrix `lineseg` has two columns. The first column defines the *x*-coordinates of the line segments and the second column defines the corresponding *y*-coordinates.

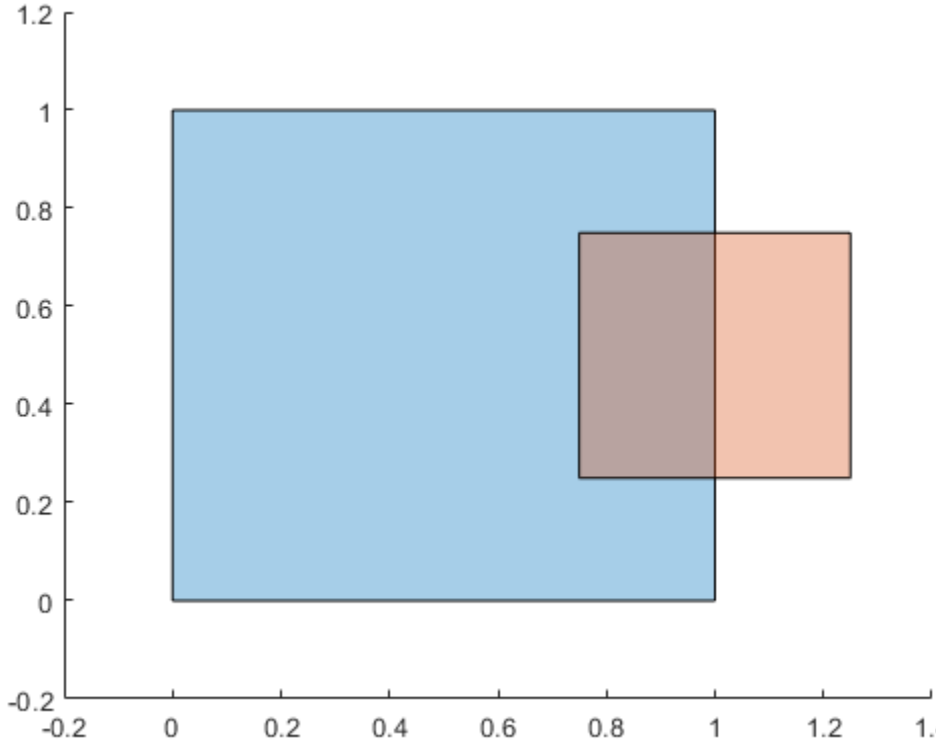
`intersect` supports this syntax only when `poly1` is a scalar `polyshape` and `lineseg` contains no self-intersections.

Examples

Intersection of Two Polygons

Create and plot two polygons.

```
poly1 = polyshape([0 0 1 1],[1 0 0 1]);
poly2 = polyshape([0.75 1.25 1.25 0.75],[0.25 0.25 0.75 0.75]);
plot(poly1)
hold on
plot(poly2)
```



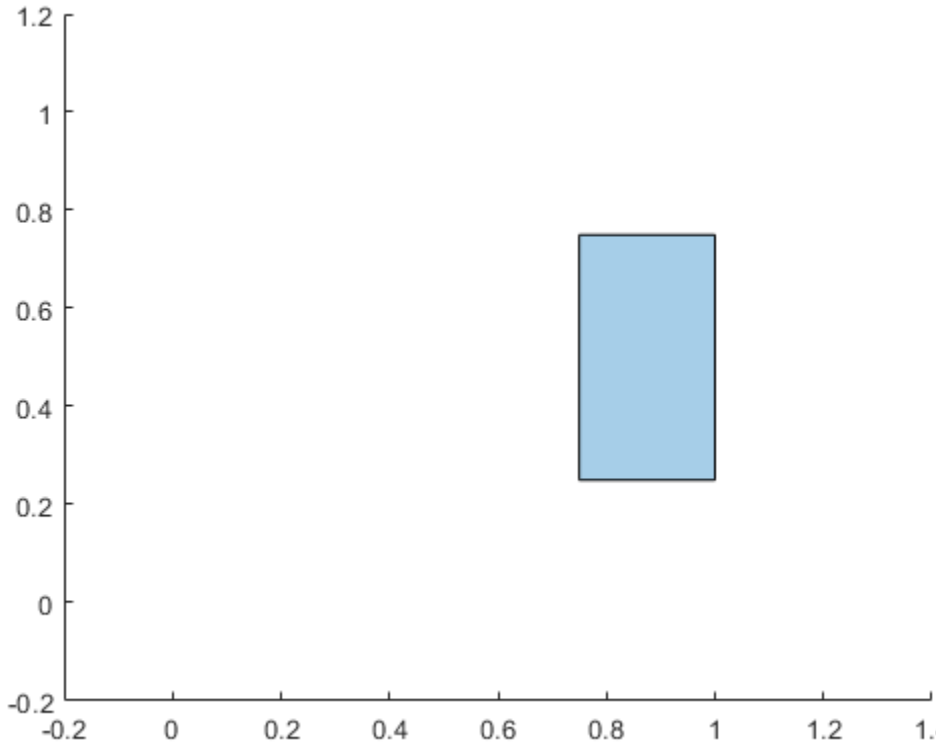

```
figure
```

Compute and plot the intersection of `poly1` and `poly2`.

```
polyout = intersect(poly1,poly2)
```

```
polyout 
  polyshape with properties:

      Vertices: [4x2 double]
    NumRegions: 1
      NumHoles: 0
```

```
plot(polyout)
xlim([-0.2 1.4]);
ylim([-0.2 1.2]);
```



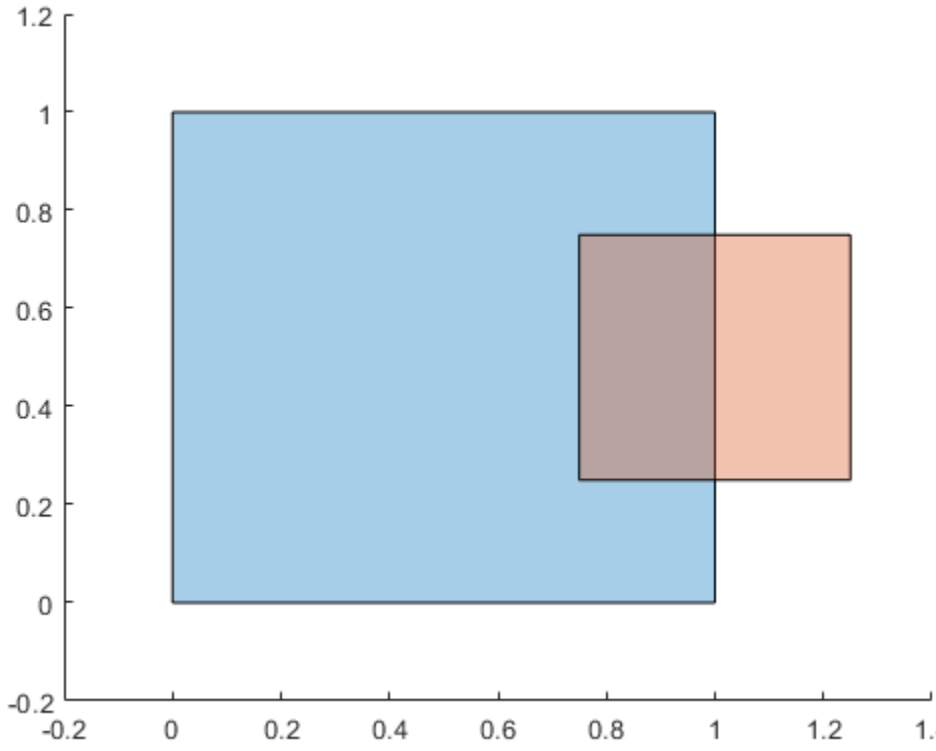
Vector of Polygons

Create a vector containing two polygons.

```
polyarray1 = polyshape([0 0 1 1],[1 0 0 1]);
polyarray2 = polyshape([0.75 1.25 1.25 0.75],[0.25 0.25 0.75 0.75]);
poly1 = [polyarray1 polyarray2]
```

```
poly1 =
  1x2 polyshape array with properties:

    Vertices
    NumRegions
    NumHoles
```

```
plot(poly1(1))
hold on
plot(poly1(2))
```



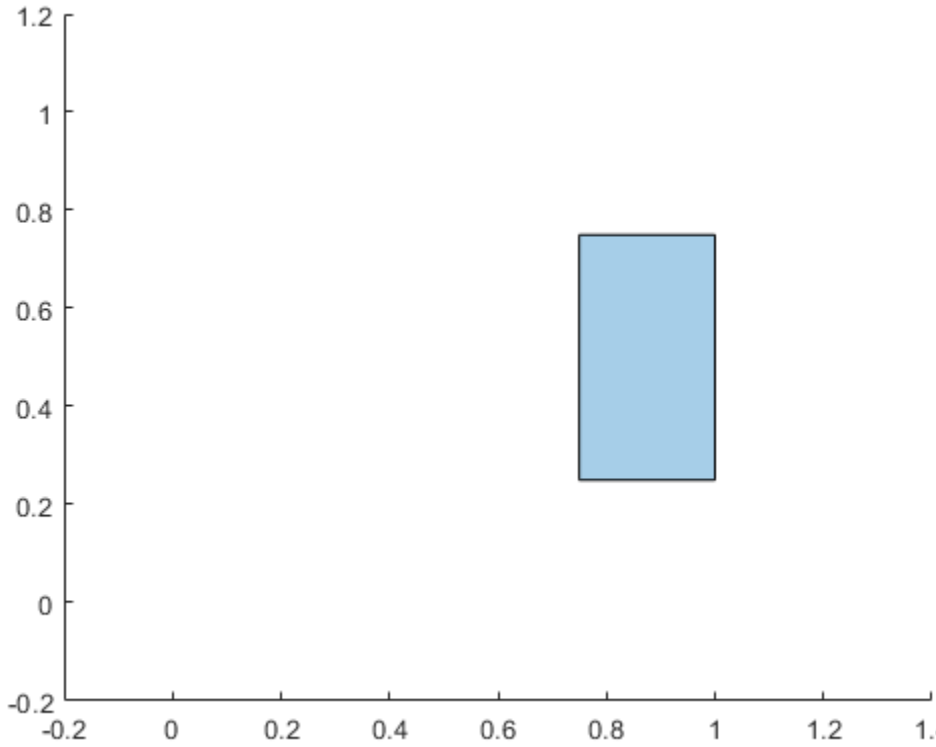

```
figure
```

Compute the intersection of the elements of `poly1`.

```
polyout = intersect(poly1)
```

```
polyout 
  polyshape with properties:

      Vertices: [4x2 double]
    NumRegions: 1
      NumHoles: 0
```

```
plot(polyout)
xlim([-0.2 1.4]);
ylim([-0.2 1.2]);
```



Vertex Mapping

Create two polygons and compute their intersection. Display the vertex coordinates of the intersection and the corresponding vertex mapping information.

```
poly1 = polyshape([0 0 1 1],[1 0 0 1]);
poly2 = polyshape([0.75 1.25 1.25 0.75],[0.25 0.25 0.75 0.75]);
[polyout,shapeID,vertexID] = intersect(poly1,poly2);
```

```
[polyout.Vertices shapeID vertexID]
```

```
ans = 4×4

    0.7500    0.2500    2.0000    1.0000
    0.7500    0.7500    2.0000    2.0000
    1.0000    0.7500         0         0
    1.0000    0.2500         0         0
```

The first two vertices of the intersection originated in `poly2`, since the corresponding values in `shapeID` are 2.These vertices are the first and second vertices in the property `poly2.Vertices`, respectively, since their corresponding values in `vertexID` are 1 and 2. The last two vertices of `polyout` were created from the intersection because the corresponding values in `shapeID` and `vertexID` are 0.

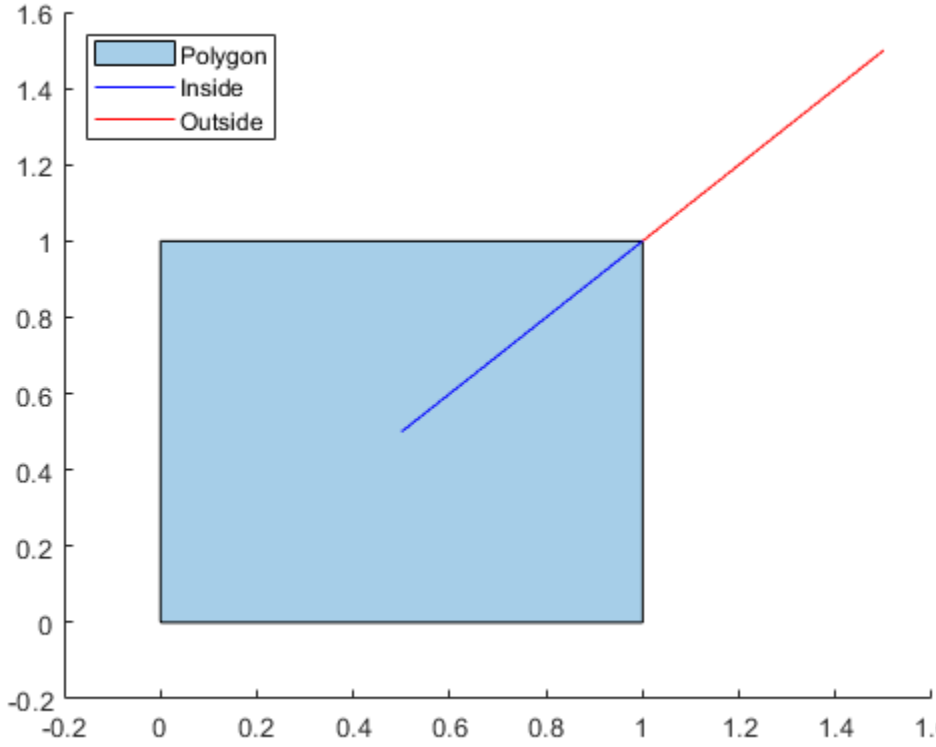
Intersection of Polygon and Line

Create a rectangular polygon and a line segment.

```
poly1 = polyshape([0 0 1 1],[1 0 0 1]);
lineseg = [0.5 0.5; 1.5 1.5];
```

Compute the intersection of the polygon with the line segment, and determine which sections of the line segment are inside or outside of the polygon.

```
[in,out] = intersect(poly1,lineseg);
plot(poly1)
hold on
plot(in(:,1),in(:,2),'b',out(:,1),out(:,2),'r')
legend('Polygon','Inside','Outside','Location','NorthWest')
```



Input Arguments

poly1 — First input polyshape
scalar | vector | matrix | multidimensional array

First input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `polyshape`

poly2 — Second input polyshape
scalar | vector | matrix | multidimensional array

Second input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `polyshape`

`polyvec` — `polyshape` vector
vector

`polyshape` vector.

Data Types: `polyshape`

`lineseg` — Line segment coordinates
two-column matrix

Line segment coordinates, specified as a two-column matrix. The first column defines the *x*-coordinates of the line segments and the second column defines the *y*-coordinates. `lineseg` must have at least two rows and contain no self-intersections.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

TF — Collinear vertices indicator
`false` | `true`

Collinear vertices indicator, specified as `false` or `true`:

- `false` — Remove collinear points so that the output `polyshape` contains the fewest vertices necessary to define the boundaries.
- `true` — Keep all collinear points as vertices.

When the `'KeepCollinearPoints'` parameter is not specified, its value is assigned according to the values used when creating the input `polyshape` objects:

- If the value was `true` for each input `polyshape` when they were created, then the value for the output `polyshape` is set to `true`.
- If the value was `false` for each input `polyshape` when they were created, then the value for the output `polyshape` is set to `false`.
- If the values for the input `polyshape` objects do not match, then the value for the output `polyshape` is set to `false`.

Data Types: `logical`

Output Arguments

`polyout` — Output `polyshape`
scalar | vector | matrix | multidimensional array

Output `polyshape`, returned as a scalar, vector, matrix, or multidimensional array.

- If you input two `polyshape` arguments, then they must have compatible sizes. For example, if two input `polyshape` vectors have different lengths *M* and *N*, then they must have different orientations (one must be a row vector and one must be a column vector). `polyout` is then *M*-by-*N* or *N*-by-*M* depending on the orientation of each input vector. For more information on compatible array sizes, see “Compatible Array Sizes for Basic Operations”.
- If you provide a single input argument `polyvec`, then `polyout` is a scalar `polyshape` object.

`shapeID` — Shape ID
column vector

Shape ID, returned as a column vector whose elements each represent the origin of a vertex in the intersection.

- The length of `shapeID` is equal to the number of rows in the `Vertices` property of the output `polyshape`.
- The elements of `shapeID` depend on the number of input arguments:
  - If you provide two input arguments `poly1` and `poly2`, then they must be scalar `polyshape` objects. The value of an element in `shapeID` is 0 when the corresponding vertex of the output `polyshape` was created by the intersection. An element is 1 when the corresponding vertex originated from `poly1`, and 2 when it originated from `poly2`.
  - If you provide one input argument `polyvec` that is a vector of `polyshape` objects, then `shapeID` contains the element index of `polyvec` from which the corresponding output vertex originated. The value of an element is 0 when the corresponding vertex was created by the intersection.

Data Types: `double`

`vertexID` — Vertex ID
column vector

Vertex ID, returned as a column vector whose elements map the vertices in the output `polyshape` to the vertices in the `polyshape` of origin. The elements of `vertexID` contain the row numbers of the corresponding vertices in the `Vertices` property of the input `polyshape`. An element is 0 when the corresponding vertex of the output `polyshape` was created by the intersection.

The length of `vertexID` is equal to the number of rows in the `Vertices` property of the output `polyshape`. If you provide two input `polyshape` objects, then `intersect` only supports this output argument if they are scalar.

Data Types: `double`

`in` — Inside coordinates
two-column matrix

Inside line segment coordinates, returned as a two-column matrix. The first column of `in` contains the *x*-coordinates of the line segments inside the input `polyshape`, and the second column contains the corresponding *y*-coordinates.

Data Types: `double`

`out` — Outside coordinates
two-column matrix

Outside line segment coordinates, returned as a two-column matrix. The first column of `out` contains the *x*-coordinates of the line segments outside of the input `polyshape`, and the second column contains the corresponding *y*-coordinates.

Data Types: `double`

See Also

`polyshape` | `xor`

Introduced in R2017b

ishole

Determine if `polyshape` boundary is a hole

Syntax

```
TF = ishole(polyin)
TF = ishole(polyin,I)
```

Description

`TF = ishole(polyin)` returns a logical vector whose elements are 1 (`true`) if the corresponding boundary of `polyin` is a hole.

`TF = ishole(polyin,I)` returns a logical vector corresponding to the boundaries of `polyin` indexed by `I`. `TF` is the same length as `I`.

Examples

Determine Polygon Holes

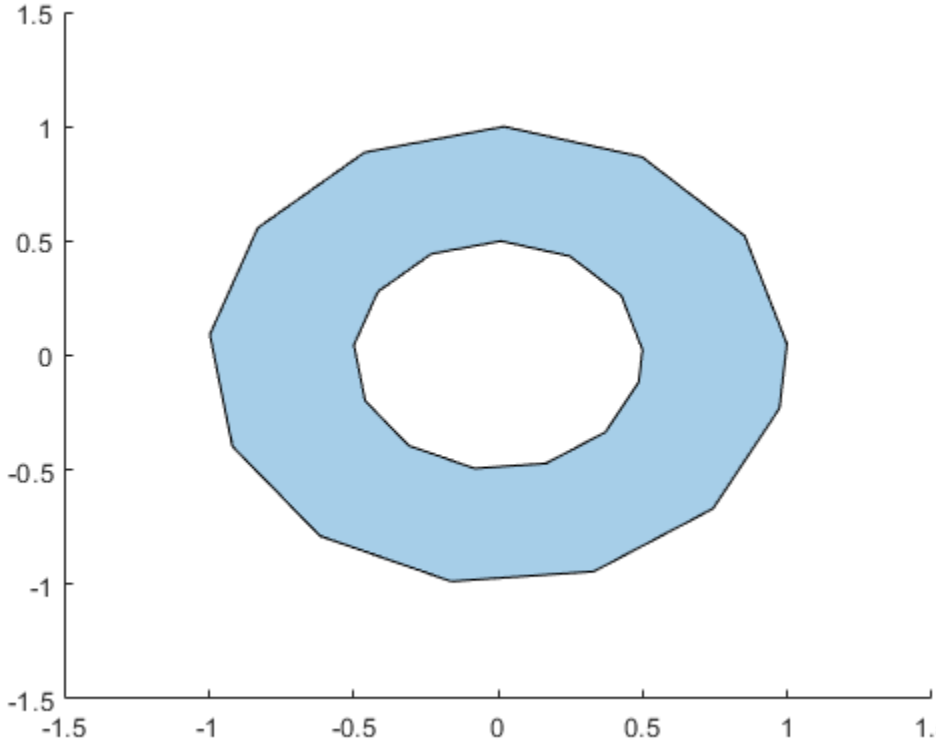
Create a polygon containing one solid region and one hole, and determine which region is a hole.

```
t = 0.05:0.5:2*pi;
x1 = cos(t);
y1 = sin(t);
x2 = 0.5*cos(t);
y2 = 0.5*sin(t);
polyin = polyshape({x1,x2},{y1,y2})
```

```
polyin = 
  polyshape with properties:

      Vertices: [27x2 double]
    NumRegions: 1
```

```
    NumHoles: 1
```

```
plot(polyin)
```




```
TF = ishole(polyin)
```

```
TF = 2x1 logical array

   0
   1
```

To query one boundary at a time, use the boundary index as a second argument.

```
TF = ishole(polyin,2)
```

```
TF = logical
   1
```

Input Arguments

polyin — Input polyshape
scalar

Input `polyshape`, specified as a scalar.

Data Types: `polyshape`

I — Boundary index
scalar integer | vector of integers

Boundary index, specified as a scalar integer or vector of integers. Each element of `I` corresponds to a single boundary of the input `polyshape`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

See Also

`holes` | `polyshape`

Introduced in R2017b

isinterior

Query points inside `polyshape`

Syntax

```
TFin = isinterior(polyin,x,y)
TFin = isinterior(polyin,P)
[TFin,TFon] = isinterior( ___ )
```

Description

`TFin = isinterior(polyin,x,y)` returns a logical vector whose elements are 1 (`true`) when the corresponding *x*-coordinates and *y*-coordinates in the vectors `x` and `y` are in a `polyshape`. A point is in a `polyshape` if it is either in a solid region or on one of the boundaries.

`TFin = isinterior(polyin,P)` represents the query points in a 2-column matrix `P`. The first column of `P` contains the *x*-coordinates of the query points, and the second column contains the corresponding *y*-coordinates.

`[TFin,TFon] = isinterior( ___ )` returns an additional logical vector for either of the previous syntaxes. The elements of `TFon` are 1 when the corresponding query points are on a boundary of `polyin`.

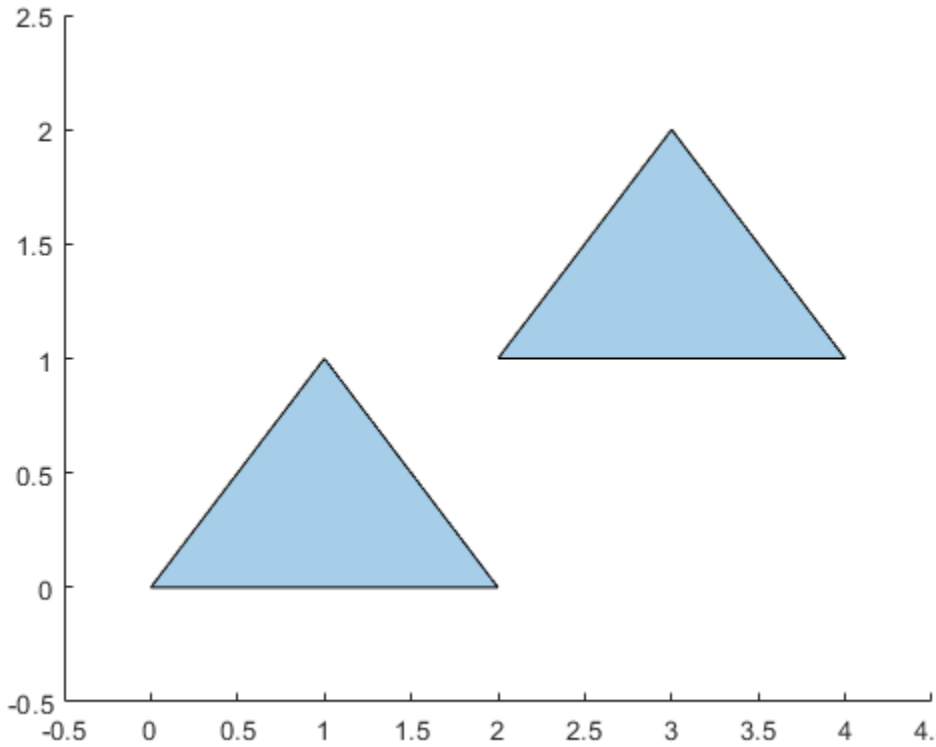
Examples

Query Points Inside Polygon

Create a polygon containing two solid regions.

```
x1 = [0 1 2];
y1 = [0 1 0];
x2 = [2 3 4];
y2 = [1 2 1];
```

```
polyin = polyshape({x1,x2},{y1,y2});
plot(polyin)
hold on
```

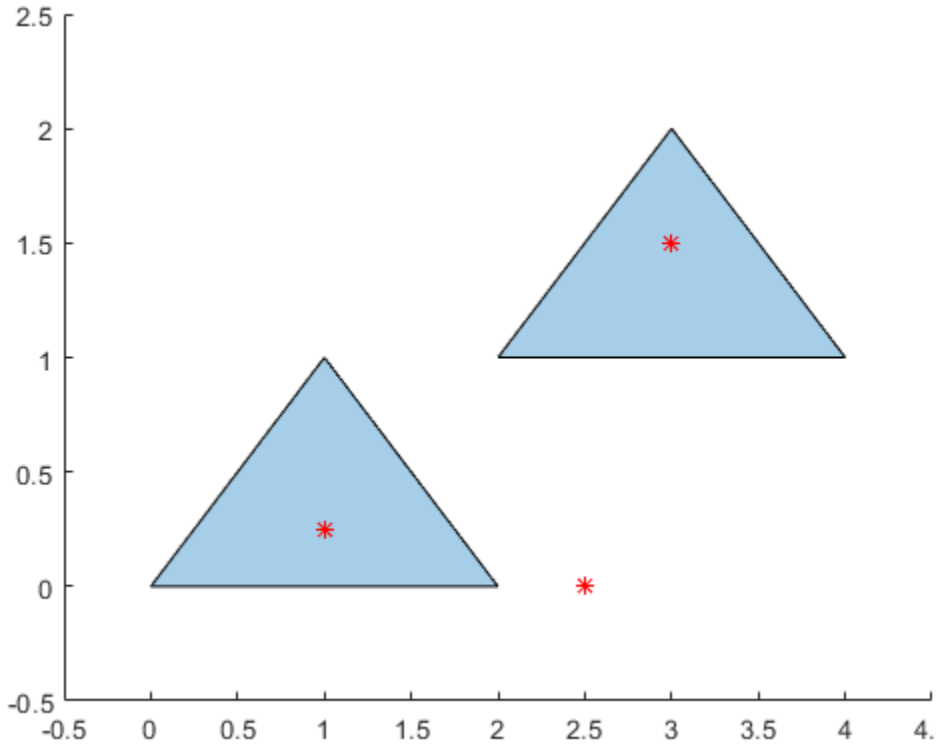


Define three points, and determine if they are in the polygon.

```
x = [1 2.5 3];
y = [0.25 0 1.5];
TFin = isinterior(polyin,x,y)
```

```
TFin = 3x1 logical array

   1
   0
```

```
1

plot(x,y,'r*')
```



Input Arguments

`polyin` — Input `polyshape`
scalar

Input `polyshape`, specified as a scalar.

Data Types: `polyshape`

x — *x*-coordinates
scalar | vector

x-coordinates of query points, specified as a scalar or vector.

Data Types: `double` | `single`

y — *y*-coordinates
scalar | vector

y-coordinates of query points, specified as a scalar or vector.

Data Types: `double` | `single`

P — 2-D vertices
2-column matrix

2-D vertices, specified as a 2-column matrix. The first column of `P` contains the *x*-coordinates of the query points, and the second column contains the *y*-coordinates.

Data Types: `double` | `single`

Output Arguments

`TFin` — Interior point indicator
scalar | vector

Interior point indicator, returned as a scalar or vector. An element of `TFin` is 1 when the corresponding query point is either in a solid region of the input `polyshape` or on a boundary.

Data Types: `logical`

`TFon` — Boundary point indicator
scalar | vector

Boundary point indicator, returned as a scalar or vector. An element of `TFon` is 1 when the corresponding query point is strictly on a boundary of the `polyshape`.

Data Types: `logical`

See Also

`boundary` | `polyshape`

Introduced in R2017b

issimplified

Determine if `polyshape` is well-defined

Syntax

```
TF = issimplified(polyin)
```

Description

`TF = issimplified(polyin)` returns a logical array whose elements are 1 (`true`) if the corresponding elements of `polyin` are well-defined polygons. Well-defined polygons have boundaries that contain no intersections and are properly nested.

Examples

Simplified Polygons

Create a row vector of two polygons: one that is ill-defined, and one that is well-defined. Use the `issimplified` function to verify their status.

```
poly1 = polyshape(rand(20,2),'Simplify',false);
poly2 = polyshape(rand(20,2));
```

```
Warning: Polyshape has duplicate vertices, intersections, or other inconsistencies that
```

```
polyin = [poly1 poly2];
TF = issimplified(polyin)
```

```
TF = 1x2 logical array

   0   1
```

Input Arguments

`polyin` — Input `polyshape`
scalar | vector | matrix | multidimensional array

Input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `polyshape`

See Also

`polyshape` | `simplify`

Introduced in R2017b

nearestvertex

Query nearest `polyshape` vertex

Syntax

```
[vertexid,boundaryid,ind] = nearestvertex(polyin,x,y)
[vertexid,boundaryid,ind] = nearestvertex(polyin,P)
```

Description

`[vertexid,boundaryid,ind] = nearestvertex(polyin,x,y)` finds the vertex of a scalar `polyshape` object that is nearest to each 2-D query point. The coordinates of the query points are specified in `x` and `y`. The output contains three vectors:

- `vertexid` contains the row number in `polyin.Vertices` corresponding to the nearest vertex for each query point.
- `boundaryid` contains the boundary index of `polyin` corresponding to the nearest vertex for each query point.
- `ind` contains the nearest vertex index in the boundary `boundaryid` for each query point.

`[vertexid,boundaryid,ind] = nearestvertex(polyin,P)` specifies the query points as a 2-column matrix `P` whose first column contains the *x*-coordinates and whose second column contains the corresponding *y*-coordinates.

Examples

Nearest Vertex

Create a `polyshape` object and compute the nearest vertex of the `polyshape` to the query point (2,1.8).

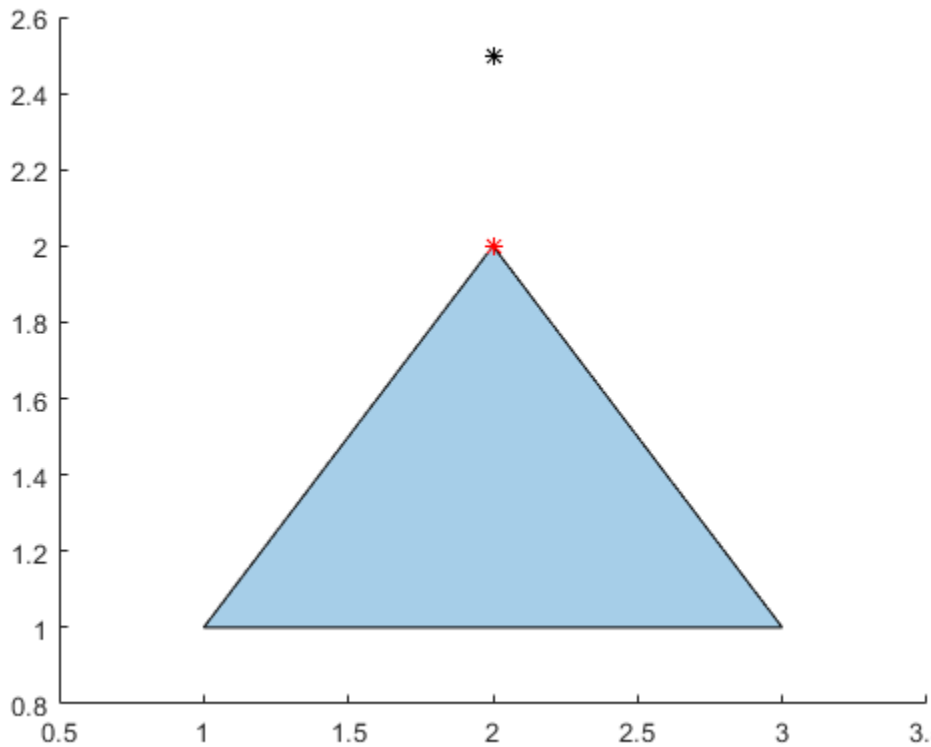
```
P = [1 1; 3 1; 2 2];
polyin = polyshape(P);
[vertexid,boundaryid,ind] = nearestvertex(polyin,2,1.8);
```

Plot the `polyshape` and the query point in black.

```
plot(polyin)
hold on
plot(2,2.5,'k*')
```

Now plot the nearest vertex in red. The `vertexid` value enables you to index into the row of `polyin.Vertices` corresponding to the nearest vertex, accessing its coordinates.

```
plot(polyin.Vertices(vertexid,1),polyin.Vertices(vertexid,2),'r*')
hold off
```



Input Arguments

`polyin` — Input `polyshape`
scalar

Input `polyshape`, specified as a scalar `polyshape` object.

`x,y` — Input coordinates
scalars | vectors

Input coordinates of 2-D query points, specified as scalars or vectors. `x` and `y` must have the same length.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`P` — Input vertices
2-column matrix

Input vertices of 2-D query points, specified as a 2-column matrix. The first column of `P` contains the *x*-coordinates and the second column contains the corresponding *y*-coordinates.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`vertexid` — Vertex ID
scalar | vector

Vertex ID, returned as a scalar or vector. `vertexid` contains the row number in the input `polyshape` corresponding to the nearest vertex for each query point. `vertexid` is the same length as the query point vectors or array.

Data Types: `double`

`boundaryid` — Boundary ID
scalar | vector

Boundary ID, returned as a scalar or vector. `boundaryid` contains the boundary index of the input `polyshape` corresponding to the nearest vertex for each query point. `boundaryid` is the same length as the query point vectors or array.

Data Types: `double`

`ind` — Index
scalar | vector

Index, returned as a scalar or vector. `ind` contains the nearest vertex index in the boundary `boundaryid` for each query point.

Data Types: `double`

See Also

`convhull` | `overlaps` | `perimeter` | `polyshape`

Introduced in R2018a

numboundaries

Number of `polyshape` boundaries

Syntax

```
N = numboundaries(polyin)
```

Description

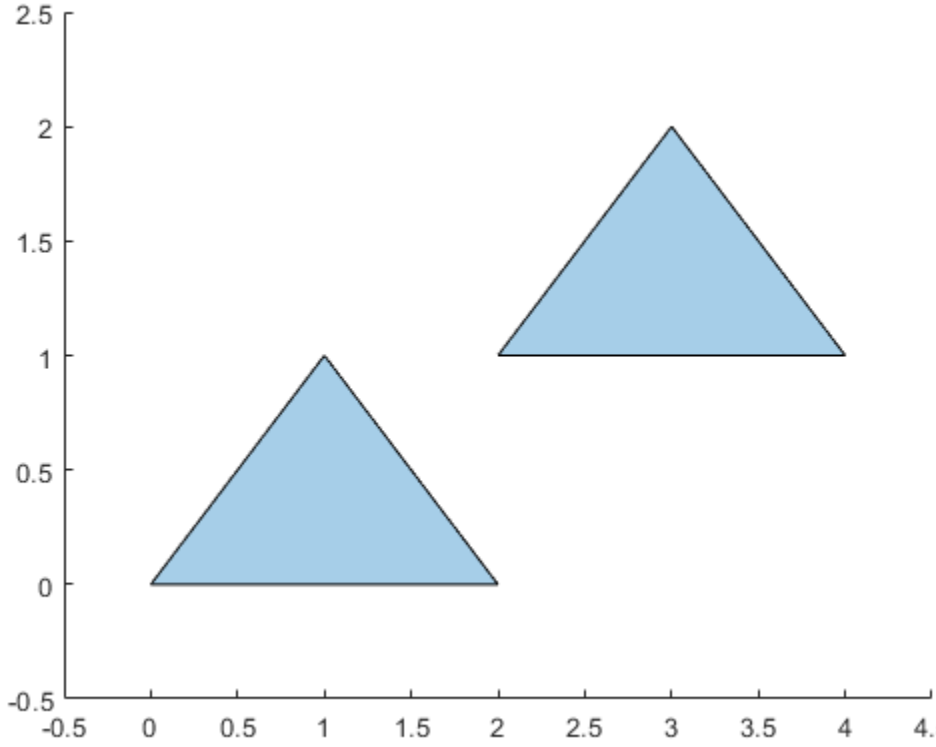
`N = numboundaries(polyin)` returns the number of boundaries of a `polyshape` object.

Examples

Number of Polygon Boundaries

Create a polygon and determine its number of boundaries.

```
x1 = [0 1 2];
y1 = [0 1 0];
x2 = [2 3 4];
y2 = [1 2 1];
polyin = polyshape({x1,x2},{y1,y2});
plot(polyin)
```




```
N = numboundaries(polyin)
```

```
N = 2
```

Input Arguments

`polyin` — Input `polyshape`
scalar | vector | matrix | multidimensional array

Input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `polyshape`

See Also

`boundary` | `numsides` | `polyshape`

Introduced in R2017b

numsides

Number of `polyshape` sides

Syntax

```
N = numsides(polyin)
N = numsides(polyin,I)
```

Description

`N = numsides(polyin)` returns the number of sides of a `polyshape` object.

`N = numsides(polyin,I)` returns the number of sides of the `I`th boundary of `polyin`.

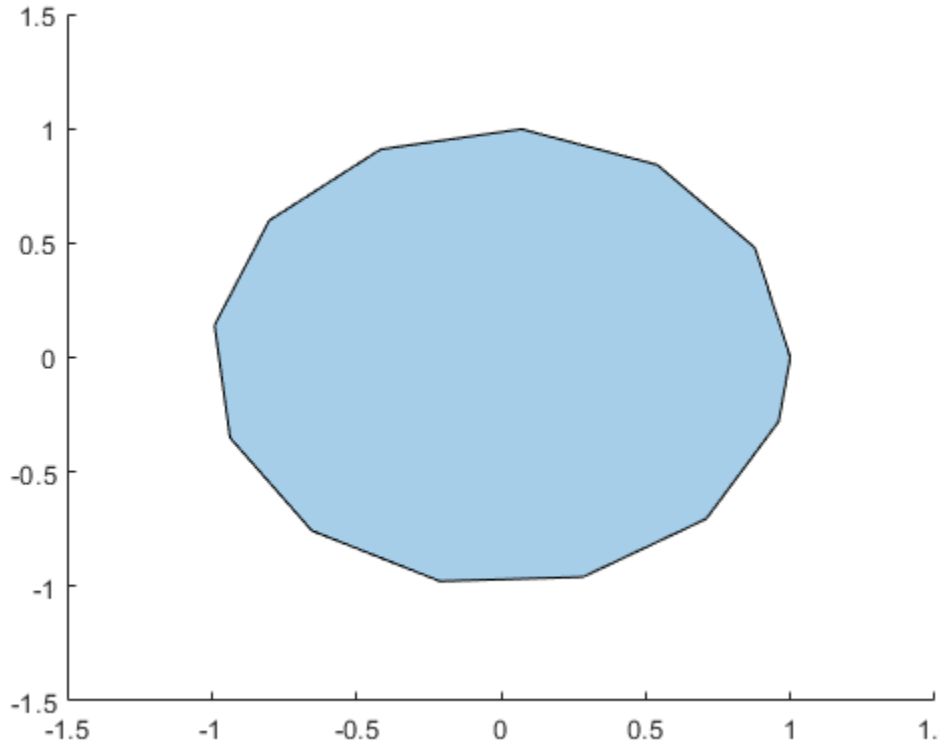
This syntax is only supported when `polyin` is a scalar `polyshape` object.

Examples

Number of Polygon Sides

Create a polygon and determine its number of sides.

```
t = 0:0.5:2*pi;
P = [cos(t)' sin(t)'];
polyin = polyshape(P);
plot(polyin)
```




```
N = numsides(polyin)
```

```
N = 13
```

Input Arguments

`polyin` — Input `polyshape`
scalar | vector | matrix | multidimensional array

Input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `polyshape`

`I` — Boundary index
scalar integer | vector of integers

Boundary index, specified as a scalar integer or vector of integers. Each element of `I` corresponds to a single boundary of the input `polyshape`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

See Also

`numboundaries` | `polyshape`

Introduced in R2017b

overlaps

Determine whether `polyshape` objects overlap

Syntax

```
TF = overlaps(poly1,poly2)
TF = overlaps(polyvec)
```

Description

`TF = overlaps(poly1,poly2)` returns a logical array whose elements are 1 when the corresponding element pairs of two `polyshape` arrays with compatible sizes overlap. *TF(i,j)* is 1 when the *i*th `polyshape` in `poly1` overlaps the *j*th `polyshape` in `poly2`.

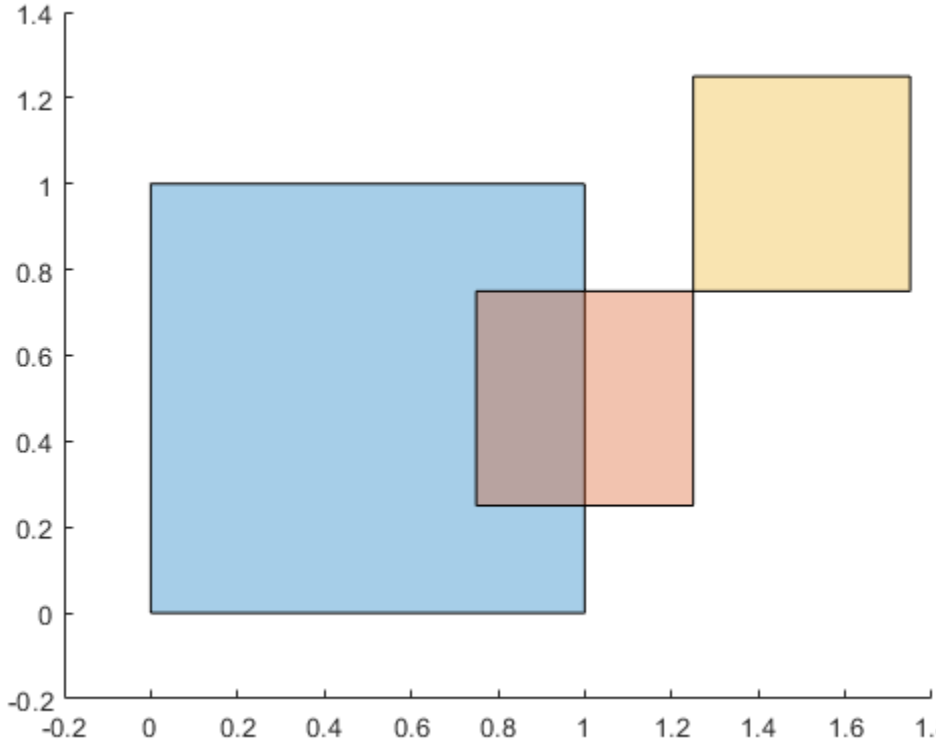
`TF = overlaps(polyvec)` returns a logical array whose elements are 1 (`true`) when the corresponding element pairs of a vector of `polyshape` objects overlap. *TF(i,j)* is 1 when the *i*th `polyshape` in `polyvec` overlaps the *j*th `polyshape`.

Examples

Overlapping Polygons

Create and plot a 1-by-3 vector of polygons, then determine which pairs of polygons overlap.

```
p1 = polyshape([0 0 1 1],[1 0 0 1]);
p2 = polyshape([0.75 1.25 1.25 0.75],[0.25 0.25 0.75 0.75]);
p3 = polyshape([1.25 1.25 1.75 1.75],[0.75 1.25 1.25 0.75]);
polyvec = [p1 p2 p3];
plot(polyvec)
```




```
TF = overlaps(polyvec)
```

```
TF = 3x3 logical array

   1   1   0
   1   1   0
   0   0   1
```

Since the third polygon does not overlap either the first or second polygon, `TF(1,3)`, `TF(2,3)`, `TF(3,1)`, and `TF(3,2)` are 0. The single point shared by the second and third polygon is not considered an overlapping region.

Input Arguments

poly1 — First input polyshape
scalar | vector | matrix | multidimensional array

First input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

poly2 — Second input polyshape
scalar | vector | matrix | multidimensional array

Second input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

polyvec — polyshape vector
vector

`polyshape` vector.

Output Arguments

TF — Overlap indicator
scalar | vector | matrix | multidimensional array

Overlap indicator, returned as a scalar, vector, matrix, or multidimensional array.

- If you input two `polyshape` arguments, then they must have compatible sizes. For example, if two `polyshape` vectors have different lengths M and N, then they must have different orientations (one must be a row vector and one must be a column vector). `TF` is then M-by-N or N-by-M depending on the orientation of each input vector. For more information on compatible array sizes, see “Compatible Array Sizes for Basic Operations”.
- If you input a single `polyshape` vector with length N, then `TF` is N-by-N.

Data Types: `logical`

See Also

`intersect` | `polyshape` | `subtract` | `union`

Introduced in R2018a

perimeter

Perimeter of `polyshape`

Syntax

```
P = perimeter(polyin)
P = perimeter(polyin,I)
```

Description

`P = perimeter(polyin)` returns the perimeter of a `polyshape` object, which is the sum of the lengths of its boundaries.

`P = perimeter(polyin,I)` returns the perimeter of the `I`th boundary of `polyin`.

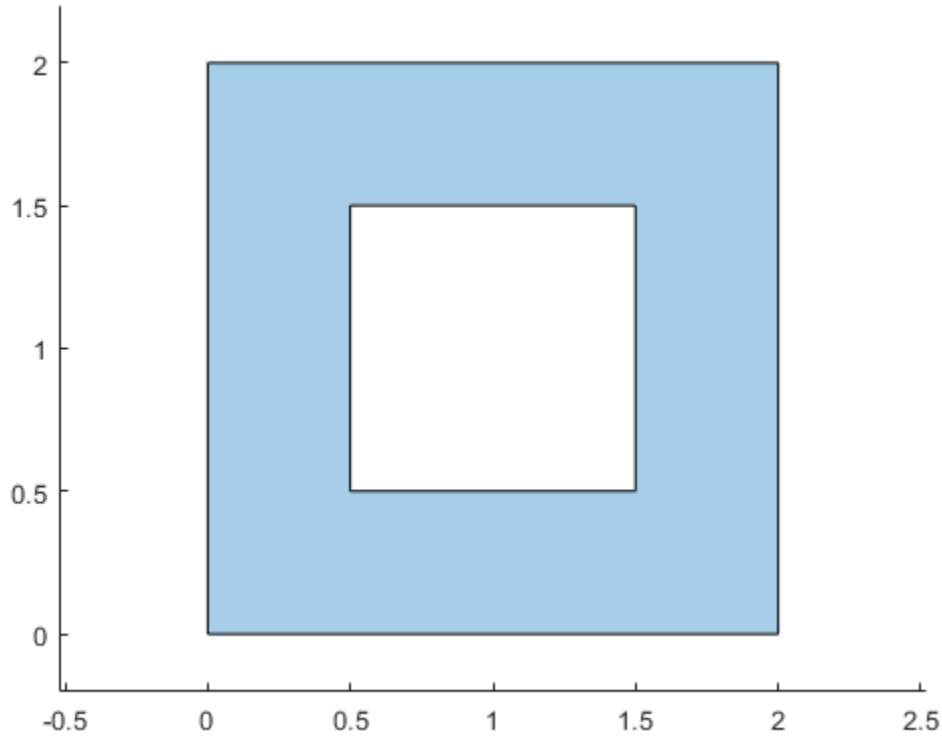
This syntax is only supported when `polyin` is a scalar `polyshape` object.

Examples

Perimeter of Polygon

Create a polygon with one hole, and compute its perimeter.

```
V = [0 0; 0 2; 2 2; 2 0; NaN NaN; 0.5 0.5; 0.5 1.5; 1.5 1.5; 1.5 0.5];
polyin = polyshape(V);
plot(polyin)
axis equal
```




```
P = perimeter(polyin)
```

```
P = 12
```

Input Arguments

`polyin` — Input `polyshape`
scalar | vector | matrix | multidimensional array

Input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `polyshape`

`I` — Boundary index
scalar integer | vector of integers

Boundary index, specified as a scalar integer or vector of integers. Each element of `I` corresponds to a single boundary of the input `polyshape`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

See Also

`area` | `centroid` | `polyshape`

Introduced in R2017b

plot

Plot `polyshape`

Syntax

```
plot(pgon)
plot(pgon,Name,Value)
h = plot( ___ )
```

Description

`plot(pgon)` plots a `polyshape` object.

`plot(pgon,Name,Value)` specifies additional properties of the `polyshape` plot using one or more name-value pairs. For example, `plot(pgon,'FaceColor','green')` shades the solid regions of a `polyshape` green.

When `pgon` is an ill-defined polygon, the shading of the plot can be inaccurate.

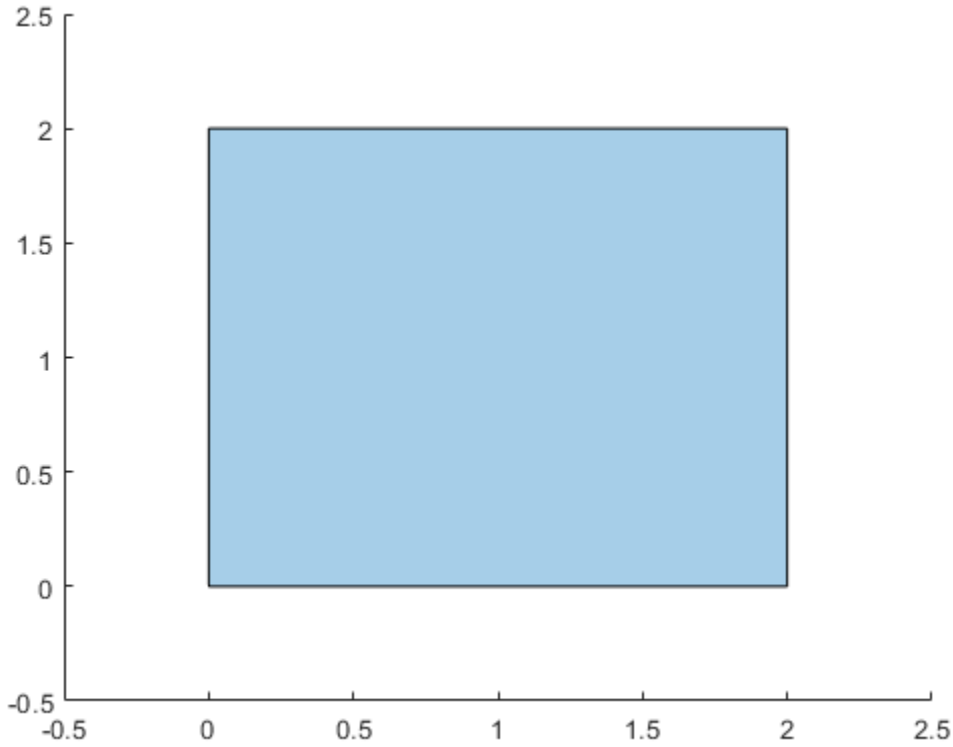
`h = plot( ___ )` returns a `Polygon` graphics object for either of the previous syntaxes. You can change the appearance and behavior of the plot by directly changing the properties of `h`. For more information, see Polygon Properties.

Examples

Simple Rectangle

Plot a rectangular polygon.

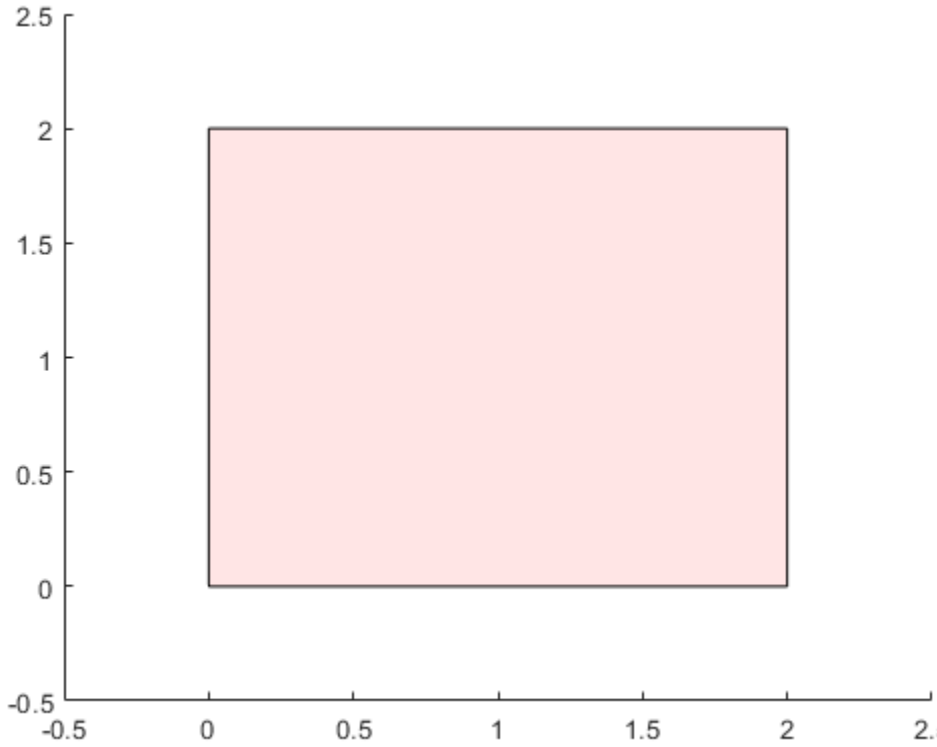
```
pgon = polyshape([0 0 2 2],[2 0 0 2]);
plot(pgon)
```



Modify Plot

Create a rectangular polygon. Plot the rectangle, specifying the color and transparency factor.

```
pgon = polyshape([0 0 2 2],[2 0 0 2]);
plot(pgon,'FaceColor','red','FaceAlpha',0.1)
```



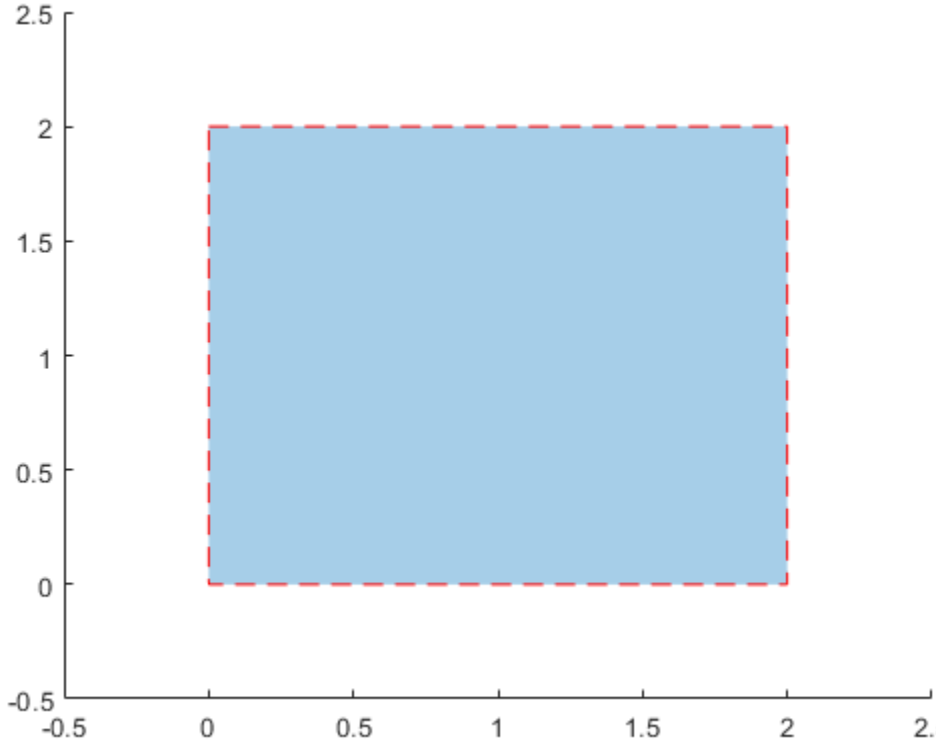
You can also change the appearance of the plot by assigning property values to the graphics object.

```
h = plot(pgon)
```

```
h 
  Polygon with properties:

    FaceColor: [0 0.4470 0.7410]
    FaceAlpha: 0.3500
    EdgeColor: [0 0 0]
    LineWidth: 0.5000
    LineStyle: '-'
        Shape: [1x1 polyshape]
```

```
Show all properties
```

```
h.LineStyle = '--';
h.EdgeColor = 'red';
```



Input Arguments

pgon — Input polyshape
scalar | vector | matrix | multidimensional array

Input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `polyshape`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `plot(pgon,'LineStyle','--')`

The plot properties listed here are only a subset. For a complete list, see Polygon Properties.

`FaceColor` — Color for solid regions
`[0.85 0.85 0.85]` (default) | color name | RGB triplet | `'none'`

Color for solid regions of the polygon, specified as the comma-separated pair consisting of `'FaceColor'` and a color name, an RGB triplet, or `'none'`.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. This table lists the long and short color name options and the equivalent RGB triplet values.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black'`	`'k'`	`[0 0 0]`

`FaceAlpha` — Face transparency
`1` (default) | scalar in range `[0,1]`

Face transparency, specified as the comma-separated pair consisting of `'FaceAlpha'` and a scalar in the range `[0,1]`. A value of 1 is opaque and 0 is completely transparent. Values between 0 and 1 are semitransparent.

`EdgeColor` — Color for boundaries
`[0 0 0]` (default) | color name | RGB triplet | `'none'`

Color for boundaries of the polygon, specified as the comma-separated pair consisting of `'EdgeColor'` and a color name, an RGB triplet, or `'none'`. The default color of `[0 0 0]` corresponds to black boundaries.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. This table lists the long and short color name options and the equivalent RGB triplet values.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black'`	`'k'`	`[0 0 0]`

`EdgeAlpha` — Edge transparency
`1` (default) | scalar in range `[0,1]`

Edge transparency, specified as the comma-separated pair consisting of `'EdgeAlpha'` and a scalar in the range `[0,1]`. A value of 1 is opaque and 0 is completely transparent. Values between 0 and 1 are semitransparent.

`LineWidth` — Line width
`0.5` (default) | positive numeric value

Line width, specified as the comma-separated pair consisting of `'LineWidth'` and a positive numeric value in points.

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

Output Arguments

`h` — `Polygon` graphics object
scalar | vector | matrix | multidimensional array

`Polygon` graphics object, specified as a scalar, vector, matrix, or multidimensional array. For more information on the properties of `h`, see Polygon Properties.

Data Types: `Polygon`

See Also

Polygon Properties | `polyshape`

Introduced in R2017b

regions

Access `polyshape` regions

Syntax

```
polyout = regions(polyin)
```

Description

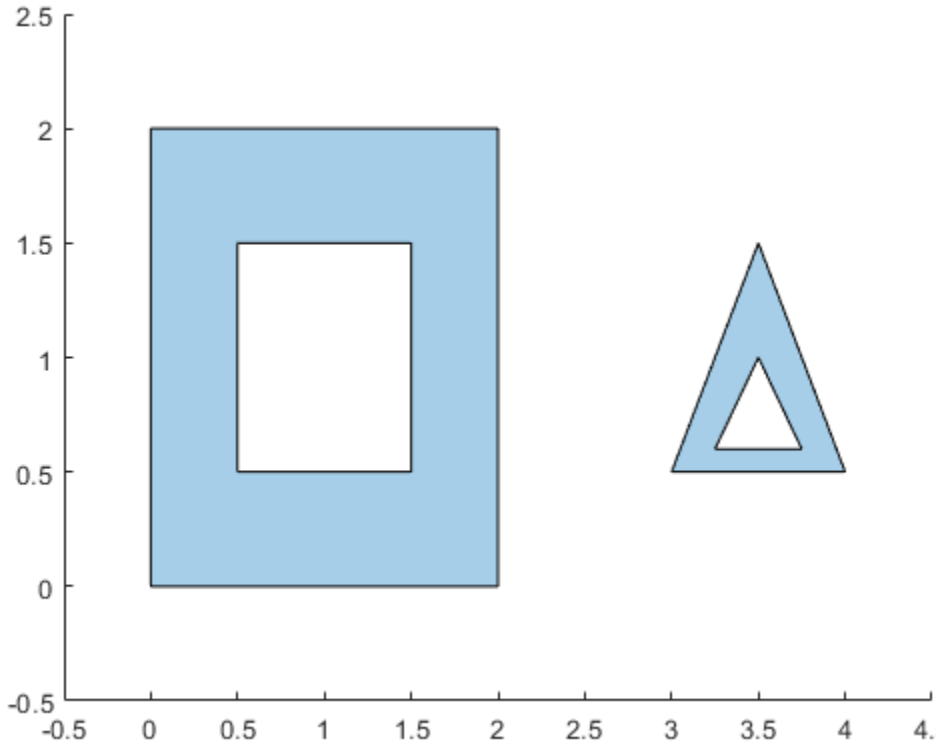
`polyout = regions(polyin)` returns a vector of `polyshape` objects whose elements are the solid regions of the input `polyshape`.

Examples

Array of Polygon Regions

Compute a polygon with two solid regions. Then, create a vector whose elements are a single region of the polygon.

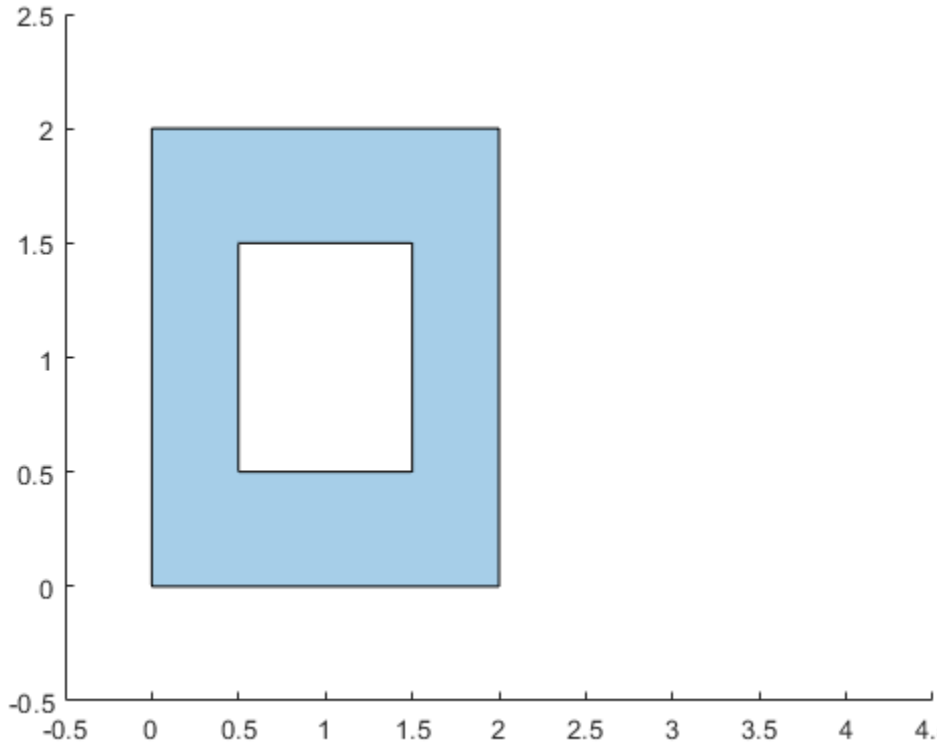
```
P = [0 0; 0 2; 2 2; 2 0; NaN NaN; 0.5 0.5; 0.5 1.5; 1.5 1.5; 1.5 0.5; NaN NaN; 3 0.5; 3
polyin = polyshape(P);
plot(polyin)
xlim([-0.5 4.5])
ylim([-0.5 2.5])
```



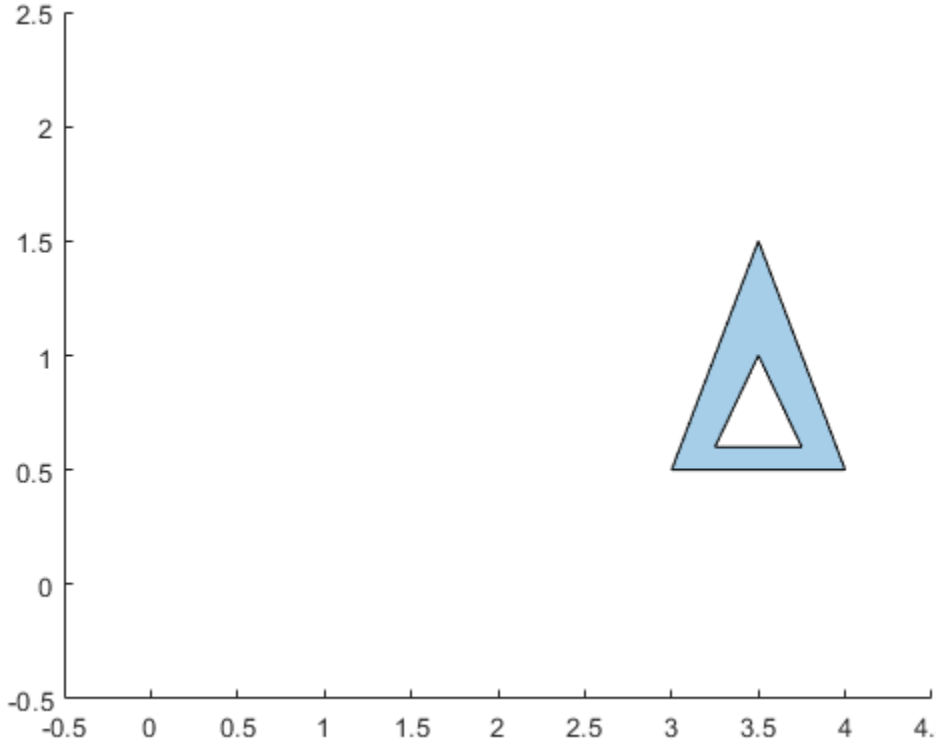

```
polyout = regions(polyin)
```

```
polyout =
  2x1 polyshape array with properties:

    Vertices
    NumRegions
    NumHoles
```

```
plot(polyout(1))
xlim([-0.5 4.5])
ylim([-0.5 2.5])
```




```
plot(polyout(2))
xlim([-0.5 4.5])
ylim([-0.5 2.5])
```



Input Arguments

`polyin` — Input `polyshape`
scalar

Input `polyshape`, specified as a scalar.

Data Types: `polyshape`

See Also

`boundary` | `holes` | `polyshape` | `sortregions`

Introduced in R2017b

rmboundary

Remove `polyshape` boundary

Syntax

```
polyout = rmboundary(polyin,I)
polyout = rmboundary(polyin,I,'Simplify',TF)
```

Description

`polyout = rmboundary(polyin,I)` returns a `polyshape` object made up of the polygon `polyin` with the `I`th boundary removed.

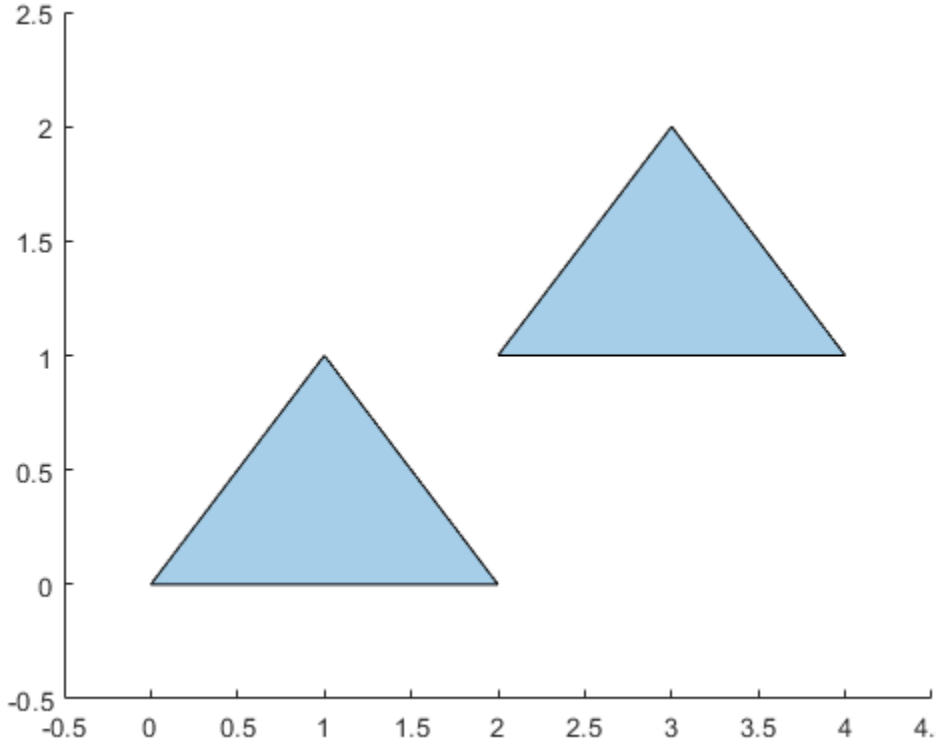
`polyout = rmboundary(polyin,I,'Simplify',TF)` specifies how to treat ill-defined polygons.

Examples

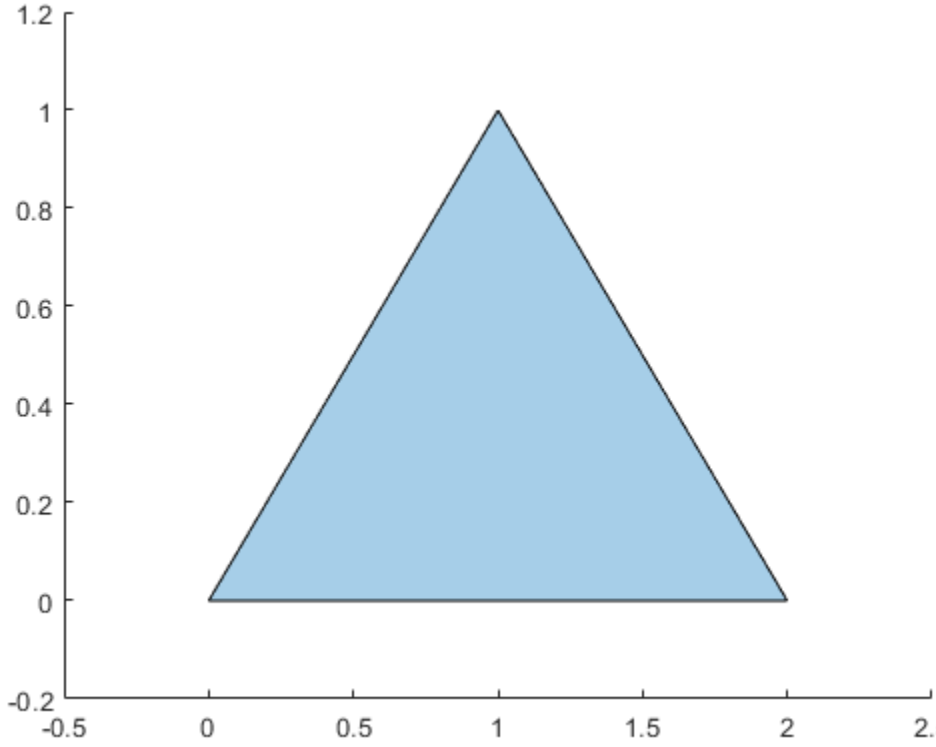
Remove Polygon Boundary

Create a polygon made up of two triangles, and then remove the triangle whose centroid is nearer to the point (4,2). You can use the `sortboundaries` function to access the boundary index corresponding to the triangle closer to the reference point, which is 1 after sorting.

```
x1 = [0 1 2];
y1 = [0 1 0];
x2 = [2 3 4];
y2 = [1 2 1];
polyin = polyshape({x1,x2},{y1,y2});
plot(polyin)
```




```
polysort = sortboundaries(polyin,'centroid','ascend','ReferencePoint',[4 2]);
polyout = rmboundary(polysort,1);
plot(polyout)
```



Input Arguments

`polyin` — Input `polyshape`
scalar

Input `polyshape`, specified as a scalar.

Data Types: `polyshape`

`I` — Boundary index
scalar integer | vector of integers

Boundary index, specified as a scalar integer or vector of integers. Each element of `I` corresponds to a single boundary of the input `polyshape`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

TF — Simplify indicator
`true` | `false`

Simplify indicator, specified as one of these values:

- `true` — Alter polygon vertices to produce a well-defined polygon when the input vertices produce intersections or improper nesting.
- `false` — Do not alter input vertices regardless of intersections or improper nesting. Computing with ill-defined polygons can lead to inaccurate or unexpected results.

Data Types: `logical`

See Also

`addboundary` | `boundary` | `polyshape` | `sortboundaries`

Introduced in R2017b

rmholes

Remove holes in `polyshape`

Syntax

```
polyout = rmholes(polyin)
```

Description

`polyout = rmholes(polyin)` returns a `polyshape` object made up of the solid boundaries of the polygon `polyin`, with all the hole boundaries removed.

Examples

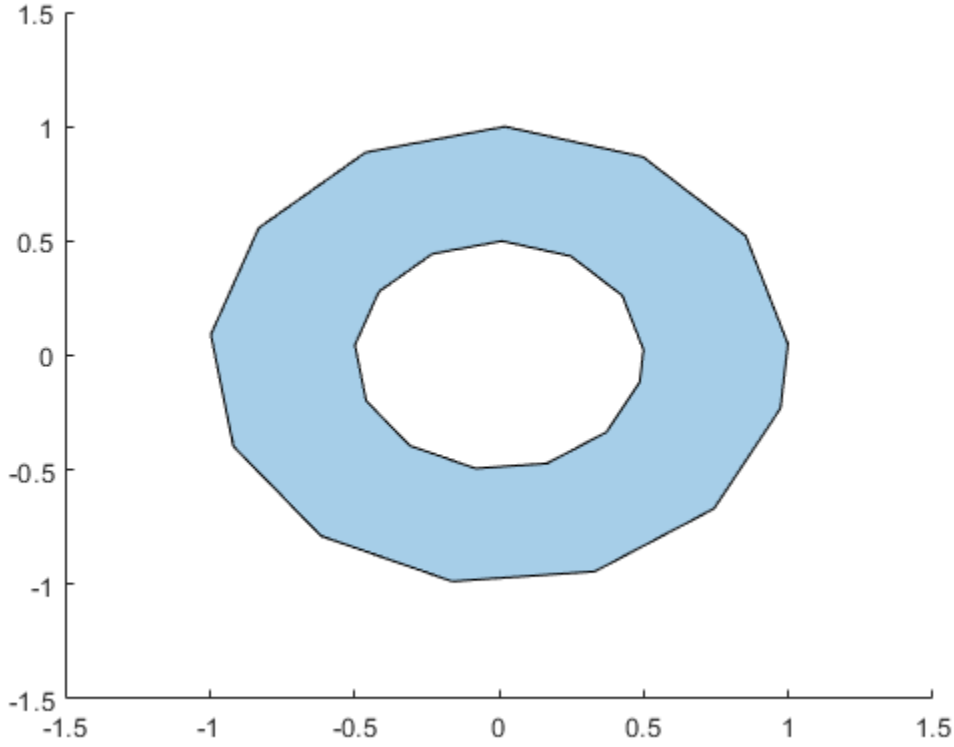
Fill Polygon Hole

Create a polygon with a hole.

```
t = 0.05:0.5:2*pi;
x1 = cos(t);
y1 = sin(t);
x2 = 0.5*cos(t);
y2 = 0.5*sin(t);
polyin = polyshape({x1,x2},{y1,y2})
```

```
polyin = 
  polyshape with properties:

      Vertices: [27x2 double]
    NumRegions: 1
      NumHoles: 1
```

```
plot(polyin)
```

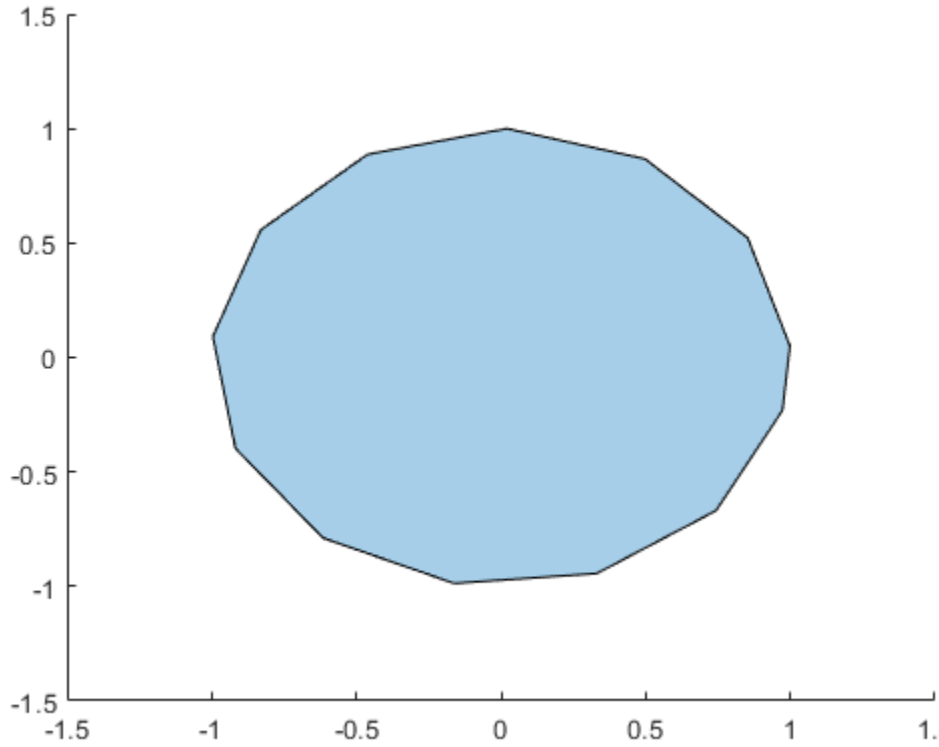


Create a new polygon with the hole removed.

```
polyout = rmholes(polyin)
```

```
polyout = 
  polyshape with properties:

      Vertices: [13x2 double]
    NumRegions: 1
      NumHoles: 0
```

```
plot(polyout)
```



Input Arguments

`polyin` — Input `polyshape`
scalar | vector | matrix | multidimensional array

Input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `polyshape`

See Also

`holes` | `ishole` | `polyshape` | `rmboundary`

Introduced in R2017b

rmslivers

Remove `polyshape` boundary outliers

Syntax

```
polyout = rmslivers(polyin,tol)
```

Description

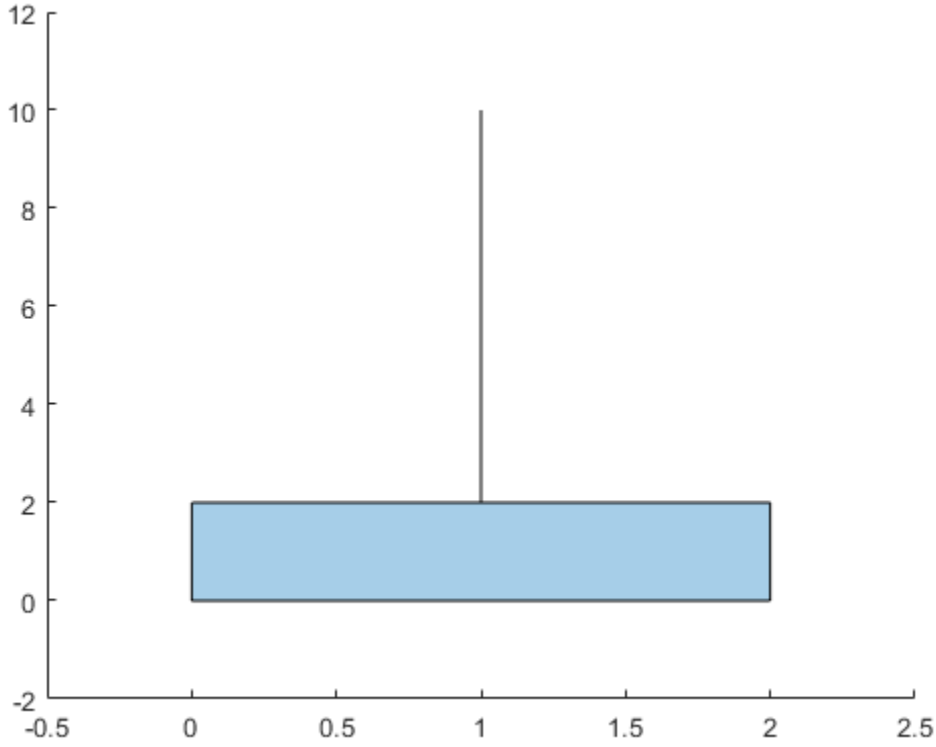
`polyout = rmslivers(polyin,tol)` returns a `polyshape` object made up of the boundaries of `polyin` with any outlier vertices removed. `tol` defines boundary outliers on page 1-11894 based on the ratio of relative distances between vertices.

Examples

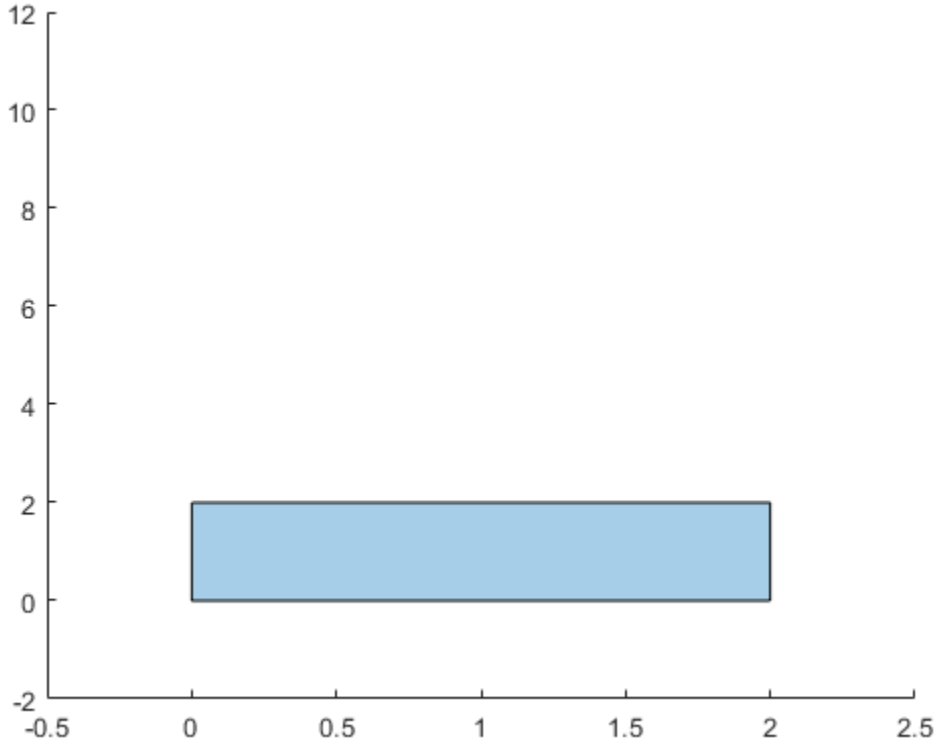
Boundary Outlier

Remove an outlier boundary point from a polygon, eliminating the extraneous sliver or antenna effect.

```
polyin = polyshape([0 0 2 2 1.0001 1 0.9999],[2 0 0 2 2 10 2]);
plot(polyin)
xlim([-0.5 2.5])
ylim([-2 12])
```




```
polyout = rmslivers(polyin,0.001);
plot(polyout)
xlim([-0.5 2.5])
ylim([-2 12])
```



Input Arguments

`polyin` — Input `polyshape`
scalar | vector | matrix | multidimensional array

Input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `polyshape`

`tol` — Outlier tolerance
scalar

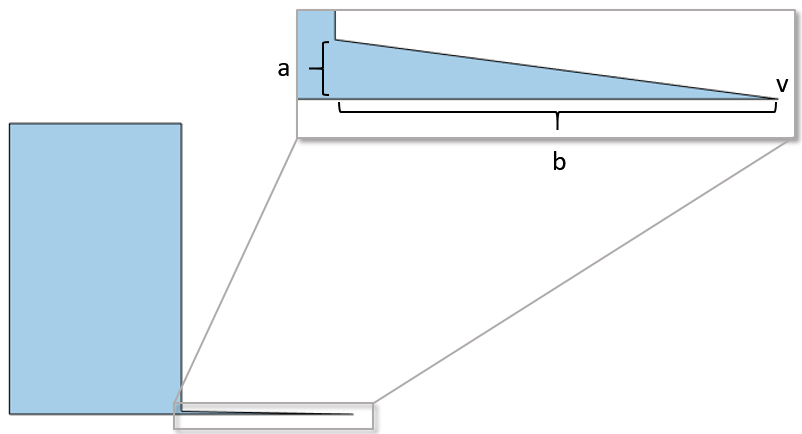
Outlier tolerance, specified as a scalar. When the input `polyshape` is an array, `rmslivers` removes outliers from each element of the array according to `tol`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

More About

Outlier Tolerance

The outlier tolerance specifies the allowable ratio between the relative distances of vertices in a polygon region. For example, in the following figure, if $\frac{a}{b} \le tol$ for a specified tolerance *tol*, then `rmslivers` does not remove the vertex *v*. If $\frac{a}{b} > tol$, then `rmslivers` removes *v*, eliminating the sliver.



See Also

`polyshape`

Introduced in R2017b

rotate

Rotate `polyshape`

Syntax

```
polyout = rotate(polyin,theta)
polyout = rotate(polyin,theta,refpoint)
```

Description

`polyout = rotate(polyin,theta)` returns a `polyshape` object created by rotating `polyin` by `theta` degrees with respect to the reference point (0,0).

`polyout = rotate(polyin,theta,refpoint)` specifies a reference point to rotate with respect to.

Examples

Rotate Square

Create a square, `polyin`, and rotate it 45 degrees counterclockwise with respect to the point (0,0).

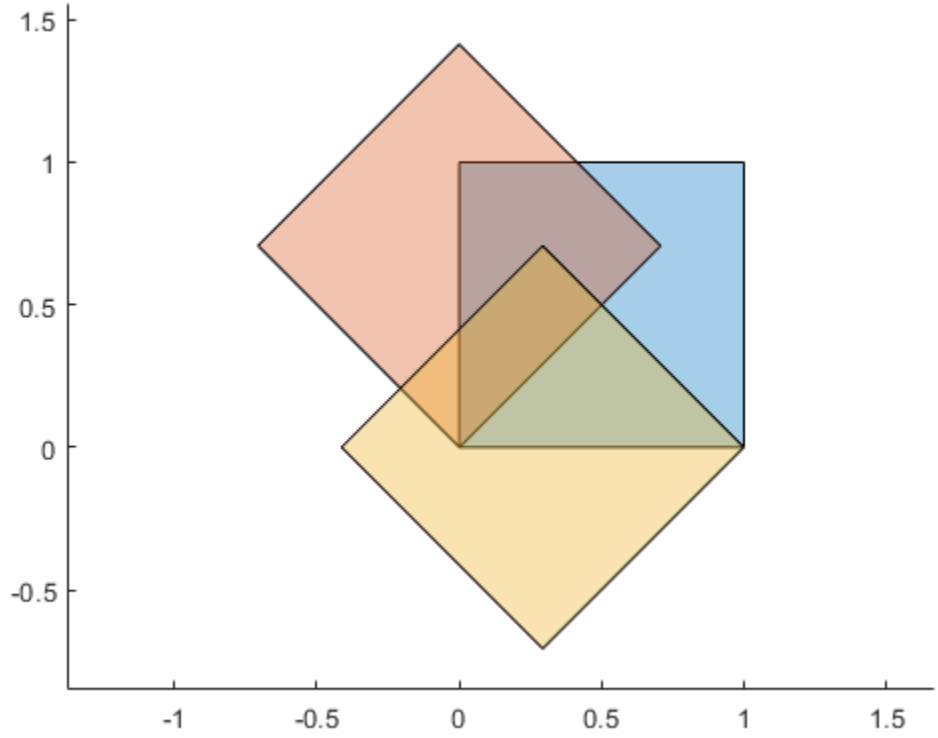
```
polyin = polyshape([0 0 1 1],[1 0 0 1]);
poly1 = rotate(polyin,45);
```

Rotate `polyin` 45 degrees with respect to the point (1,0).

```
poly2 = rotate(polyin,45,[1 0]);
```

Plot all three polygons.

```
plot([polyin poly1 poly2])
axis equal
```



Input Arguments

`polyin` — Input `polyshape`
scalar | vector | matrix | multidimensional array

Input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `polyshape`

`theta` — Rotation angle
scalar

Rotation angle, specified as a scalar number of degrees. When the input `polyshape` is an array, each element of the array is rotated by `theta` degrees. The rotation is counterclockwise when `theta` is positive, and clockwise when `theta` is negative.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`refpoint` — Reference point
`[0 0]` | two-element row vector

Reference point, specified as a two-element row vector containing the *x*-coordinate and *y*-coordinate. The `rotate` function rotates with respect to the reference point. When the input `polyshape` is an array, each element of the array is rotated with respect to `refpoint`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

See Also

`polyshape` | `scale` | `translate`

Introduced in R2017b

scale

Scale `polyshape`

Syntax

```
polyout = scale(polyin,s)
polyout = scale(polyin,s,refpoint)
```

Description

`polyout = scale(polyin,s)` returns a `polyshape` object defined by scaling `polyin` by a factor `s` with respect to the reference point (0,0). If `s` is a scalar, then `scale` applies it to both the *x* and *y* directions. If `s` is a two-element row vector, then the first element is the scale factor in the *x* direction and the second element is the scale factor in the *y* direction.

`polyout = scale(polyin,s,refpoint)` specifies a reference point to scale with respect to.

Examples

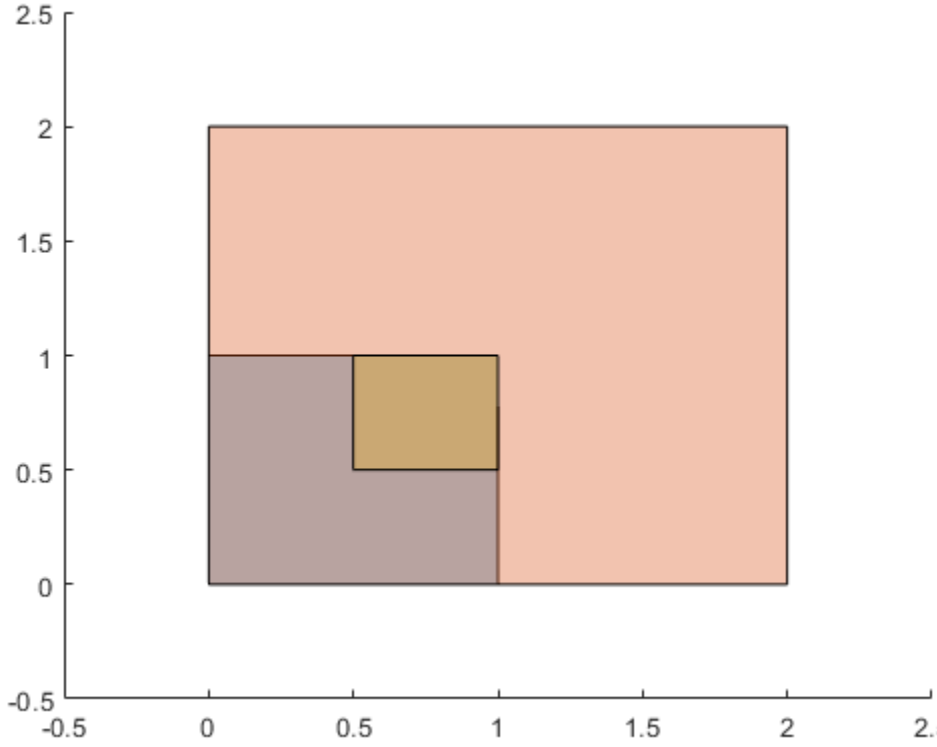
Scale Polygon

Create a polygon and scale it by a factor of 2.

```
polyin = polyshape([0 0 1 1],[1 0 0 1]);
poly1 = scale(polyin,2);
```

Scale `polyin` by a factor of 0.5 with respect to the point (1,1). Plot all three polygons.

```
poly2 = scale(polyin,0.5,[1 1]);
plot(polyin)
hold on
plot(poly1)
plot(poly2)
```



Input Arguments

`polyin` — Input `polyshape`
scalar | vector | matrix | multidimensional array

Input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `polyshape`

`s` — Scale factor
scalar | two-element row vector

Scale factor, specified as a scalar or a two-element row vector. The elements of `s` must be positive.

- If `s` is a scalar, then `scale` applies it to both the *x* and *y* directions.
- If `s` is a two-element row vector, then the first element is the scale factor in the *x* direction and the second element is the scale factor in the *y* direction.

When the input `polyshape` is an array, `scale` applies `s` to each element of the array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`refpoint` — Reference point
`[0 0]` | two-element row vector

Reference point, specified as a two-element row vector containing the *x*-coordinate and *y*-coordinate of the point to scale with respect to. When the input `polyshape` is an array, each element of the array is scaled with respect to `refpoint`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

See Also

`polyshape` | `rotate` | `translate`

Introduced in R2017b

simplify

Simplify `polyshape` boundaries

Syntax

```
polyout = simplify(polyin)
polyout = simplify(polyin,'KeepCollinearPoints',TF)
```

Description

`polyout = simplify(polyin)` returns a `polyshape` object made up of the boundaries of the polygon `polyin` with all vertex duplicates removed, and all boundary intersections and improper nesting resolved.

`polyout = simplify(polyin,'KeepCollinearPoints',TF)` specifies whether to keep or remove collinear points in `polyout`.

Examples

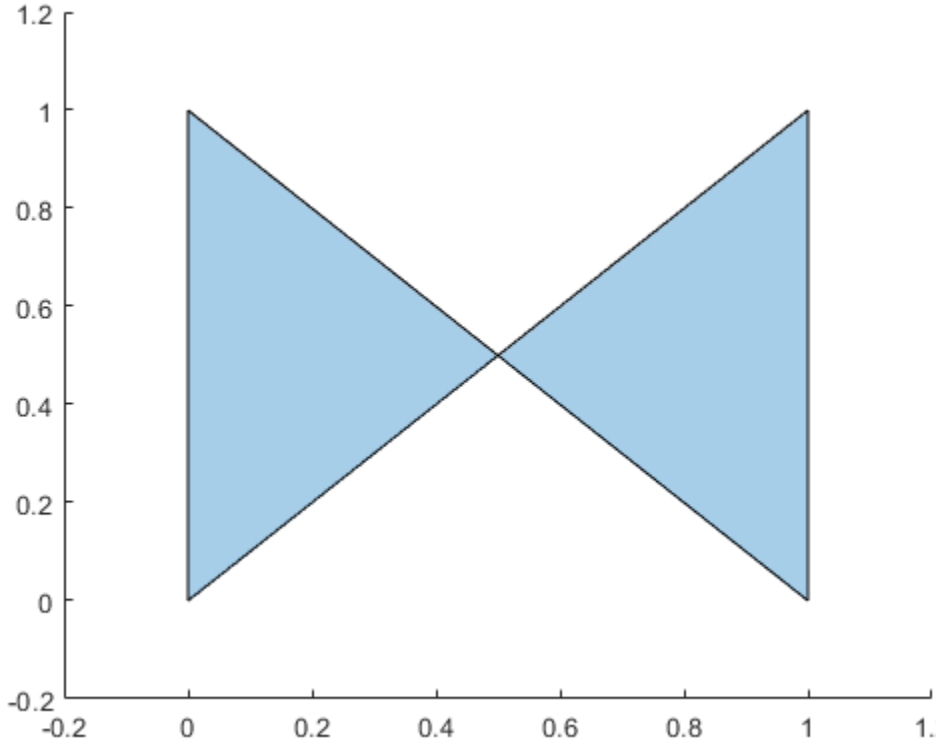
Intersecting Boundary

Create and plot a polygon that contains a boundary intersection.

```
P = [0 0; 1 1; 1 0; 0.5 0.5; 0 1; 0 0];
polyin = polyshape(P,'Simplify',false)
```

```
polyin 
  polyshape with properties:

      Vertices: [5x2 double]
    NumRegions: 1
      NumHoles: 0
```

```
plot(polyin)
```



Use the `simplify` function to remove the intersection, which produces a well-defined polygon. Simplifying the polygon maintains the boundary shape, but splits the polygon into two distinct regions.

```
polyout = simplify(polyin)
```

```
polyout = 
  polyshape with properties:

      Vertices: [7x2 double]
    NumRegions: 2
      NumHoles: 0
```

Input Arguments

polyin — Input polyshape
scalar | vector | matrix | multidimensional array

Input polyshape, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: polyshape

TF — Collinear vertices indicator
false | true

Collinear vertices indicator, specified as false or true:

- false — Remove collinear points so that the output polyshape contains the fewest vertices necessary to define the boundaries.
- true — Keep all collinear points as vertices.

When the 'KeepCollinearPoints' parameter is not specified, its value is automatically set to the value used when creating the input polyshape.

Data Types: logical

See Also

issimplified | polyshape

Introduced in R2017b

sortboundaries

Sort `polyshape` boundaries

Syntax

```
polyout = sortboundaries(polyin,criterion,direction)
polyout = sortboundaries(polyin,'centroid',
direction,'ReferencePoint',point)
```

Description

`polyout = sortboundaries(polyin,criterion,direction)` returns a `polyshape` object whose boundaries are sorted according to the specified `criterion` and `direction`. For example, `polyout = sortboundaries(polyin,'area','ascend')` returns a `polyshape` whose boundaries are the same as `polyin`. `polyout` lists the boundaries in ascending order by their area.

`polyout = sortboundaries(polyin,'centroid', direction,'ReferencePoint',point)` returns a `polyshape` whose boundaries are sorted based on the distance of the centroid of each boundary to a reference point.

Examples

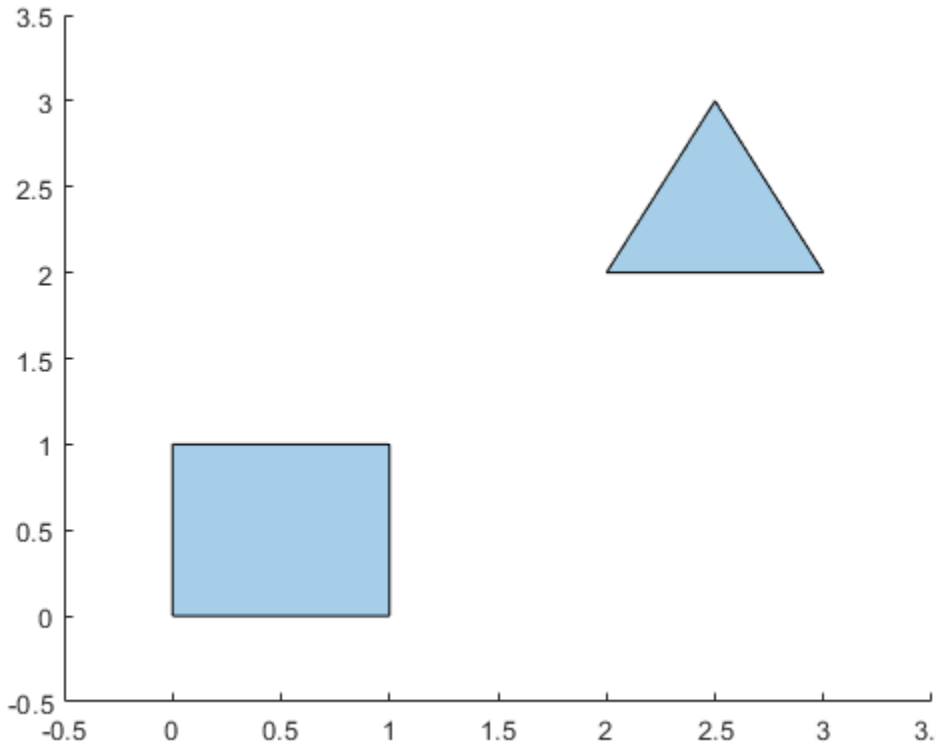
Sort by Number of Sides

Sort the boundaries of a polygon according to their number of sides in order to access a single boundary at a time.

Create a polygon containing two boundaries, one with three sides and one with four sides.

```
x1 = [0 1 1 0];
y1 = [0 0 1 1];
x2 = [2 3 2.5];
```

```
y2 = [2 2 3];
polyin = polyshape({x1,x2},{y1,y2});
plot(polyin)
```



Sort the boundaries in descending order according to their number of sides, so that the four-sided boundary is indexed first in the output `polyshape` object.

```
polyout = sortboundaries(polyin,'numsides','descend');
```

You now can reference the four-sided boundary based on its index, for example when you want to access the boundary's vertices or compute its area.

```
[x,y] = boundary(polyout,1)
```

```
x = 5×1

     0
     0
     1
     1
     0


y = 5×1

     0
     1
     1
     0
     0
```

```
area(polyout,1)
```

```
ans = 1
```

Input Arguments

`polyin` — Input `polyshape`
scalar | vector | matrix | multidimensional array

Input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `polyshape`

`criterion` — Sort criterion
`'area'` | `'perimeter'` | `'numsides'` | `'centroid'`

Sort criterion, specified as one of these values:

- `'area'` — Sort by boundary area.
- `'perimeter'` — Sort by boundary perimeter.
- `'numsides'` — Sort by the number of sides of each boundary.
- `'centroid'` — Sort by the distance from the centroid of each boundary to the reference point (0,0).

`direction` — Sort direction
`'ascend'` | `'descend'`

Sort direction, specified as `'ascend'` or `'descend'`.

`point` — Reference point
`[0 0]` | two-element row vector

Reference point, specified as a two-element row vector. The first element is the *x*-coordinate of the reference point, and the second element is the *y*-coordinate.

See Also

`boundary` | `polyshape` | `rmboundary` | `sortregions`

Introduced in R2017b

sortregions

Sort `polyshape` regions

Syntax

```
polyout = sortregions(polyin,criterion,direction)
polyout = sortregions(polyin,'centroid',direction,'ReferencePoint',
point)
```

Description

`polyout = sortregions(polyin,criterion,direction)` returns a `polyshape` object whose regions are sorted according to the specified `criterion` and `direction`. For example, `polyout = sortregions(polyin,'area','ascend')` returns a `polyshape` whose solid regions are the same as `polyin`, and listed in ascending order by their area.

`polyout = sortregions(polyin,'centroid',direction,'ReferencePoint', point)` returns a `polyshape` whose regions are sorted based on the distance of the centroid of each region to a reference point.

Examples

Sort by Perimeter

Identify and access individual regions of a polygon by sorting them first.

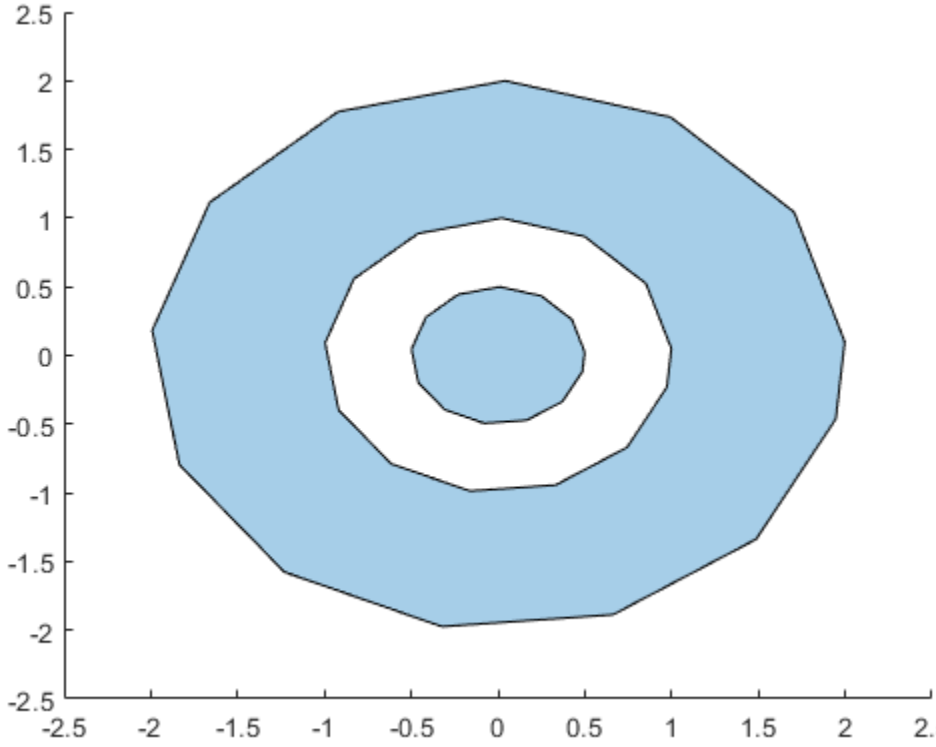
Create a polygon with two solid regions and one hole.

```
t = 0.05:0.5:2*pi;
x1 = cos(t);
y1 = sin(t);
x2 = 0.5*cos(t);
y2 = 0.5*sin(t);
```

```
x3 = 2*cos(t);
y3 = 2*sin(t);
polyin = polyshape({x1,x2,x3},{y1,y2,y3})
```

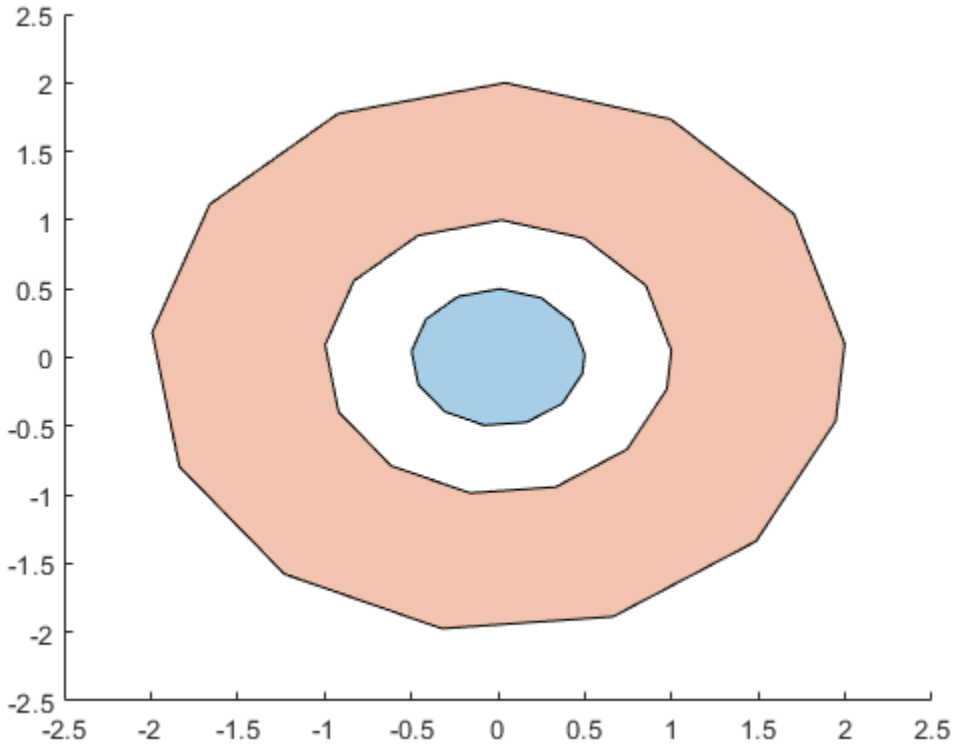
```
polyin 
  polyshape with properties:

      Vertices: [41x2 double]
    NumRegions: 2
      NumHoles: 1
```

```
plot(polyin)
```



Sort the regions of the polygon in ascending order based on their perimeter. You then can use the `regions` function to create an array of `polyshape` objects R, where each element of R corresponds to a single region of `polyout`. The elements of R are indexed in the same order as the regions of `polyout`, so that you can access and compute with each region based on their index.

```
polyout = sortregions(polyin,'perimeter','ascend');
R = regions(polyout);
plot(R)
```




```
smallregion = area(R(1))
```

```
smallregion = 0.7541
```

```
bigregion = area(R(2))
```

```
bigregion = 9.0488
```

Input Arguments

polyin — Input polyshape
scalar | vector | matrix | multidimensional array

Input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `polyshape`

criterion — Sort criterion
`'area'` | `'perimeter'` | `'numsides'` | `'centroid'`

Sort criterion, specified as one of these values:

- `'area'` — Sort by region area.
- `'perimeter'` — Sort by region perimeter.
- `'numsides'` — Sort by the number of sides of each region.
- `'centroid'` — Sort by the distance from the centroid of each region to the reference point (0,0).

direction — Sort direction
`'ascend'` | `'descend'`

Sort direction, specified as `'ascend'` or `'descend'`.

point — Reference point
`[0 0]` | two-element row vector

Reference point, specified as a two-element row vector. The first element is the *x*-coordinate of the reference point, and the second element is the *y*-coordinate.

See Also

`polyshape` | `regions` | `sortboundaries`

Introduced in R2017b

subtract

Difference of two `polyshape` objects

Syntax

```
polyout = subtract(poly1,poly2)
[polyout,shapeID,vertexID] = subtract(poly1,poly2)
___ = subtract(poly1,poly2,'KeepCollinearPoints',TF)
```

Description

`polyout = subtract(poly1,poly2)` returns a `polyshape` object whose regions are the geometric difference of two `polyshape` objects. The output `polyout` contains the regions of `poly1` minus any part of `poly2` that overlaps with `poly1`. The input arguments `poly1` and `poly2` must have compatible array sizes.

`[polyout,shapeID,vertexID] = subtract(poly1,poly2)` also returns vertex mapping information from the vertices in `polyout` to the vertices in `poly1` and `poly2`. The `subtract` function only supports this syntax when `poly1` and `poly2` are scalar `polyshape` objects.

The `shapeID` elements identify whether the corresponding vertex in `polyout` originated in `poly1`, `poly2`, or was created from the difference. `vertexID` maps the vertices of `polyout` to the vertices of `poly1`, `poly2`, or the difference.

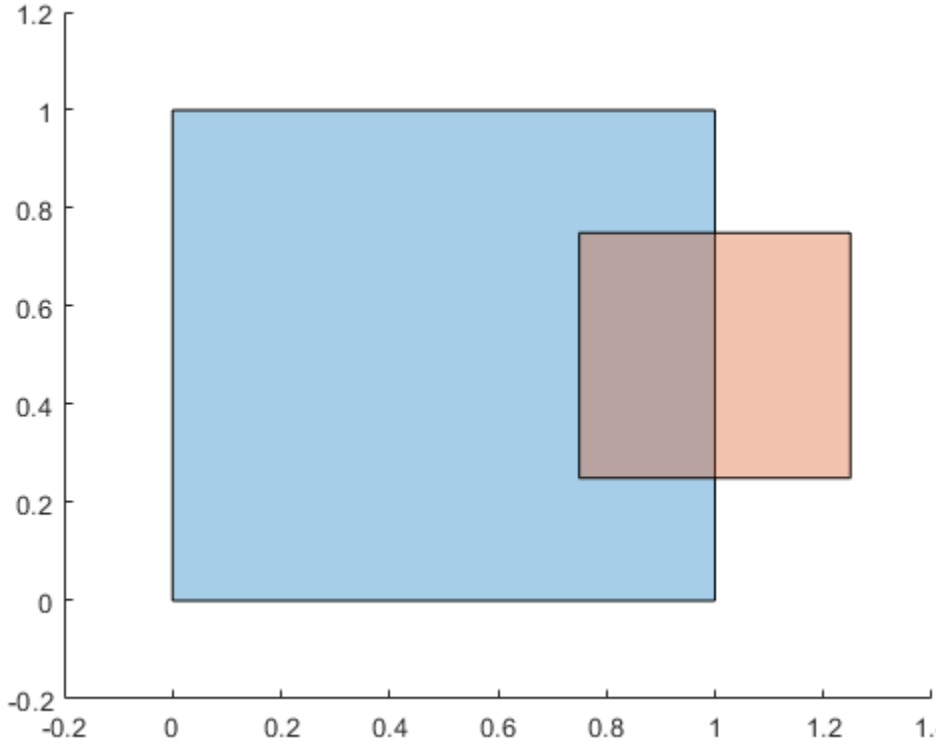
`___ = subtract(poly1,poly2,'KeepCollinearPoints',TF)` specifies whether to keep or remove collinear points in `polyout` for any of the previous syntaxes.

Examples

Difference of Two Polygons

Create and plot two polygons `poly1` and `poly2` that partially overlap.

```
poly1 = polyshape([0 0 1 1],[1 0 0 1]);
poly2 = polyshape([0.75 1.25 1.25 0.75],[0.25 0.25 0.75 0.75]);
plot(poly1)
hold on
plot(poly2)
```




```
figure
```

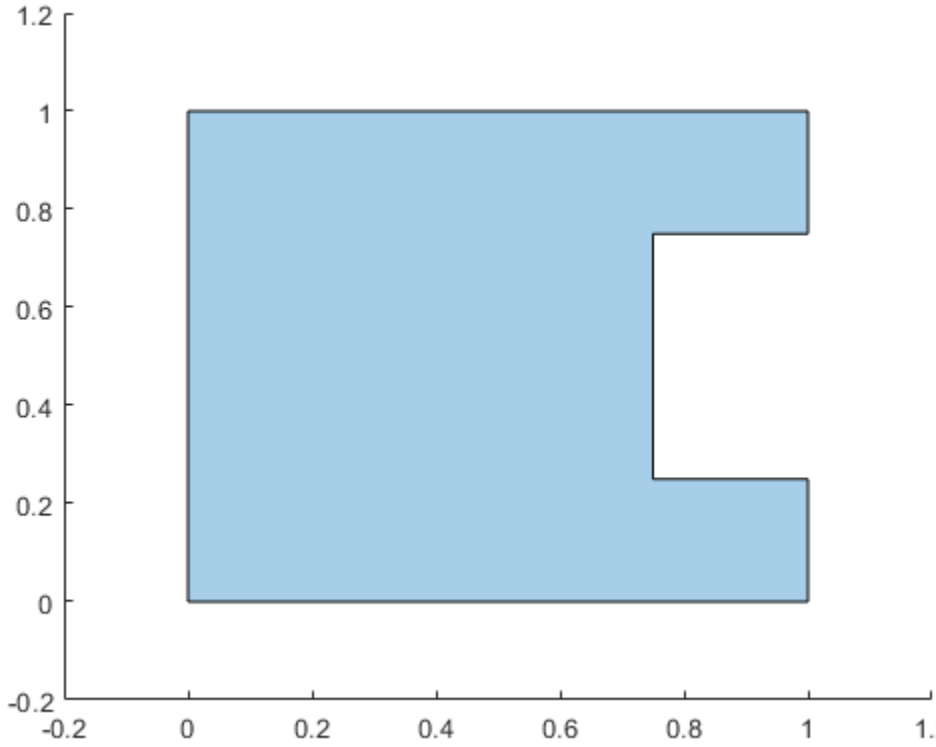
Subtract `poly2` from `poly1`. The resulting polygon is `poly1` minus any part of `poly2` that overlaps with `poly1`.

```
polyout1 = subtract(poly1,poly2)
```

```
polyout1 
  polyshape with properties:
```

```
      Vertices: [8x2 double]
    NumRegions: 1
      NumHoles: 0
```

```
plot(polyout1)
```



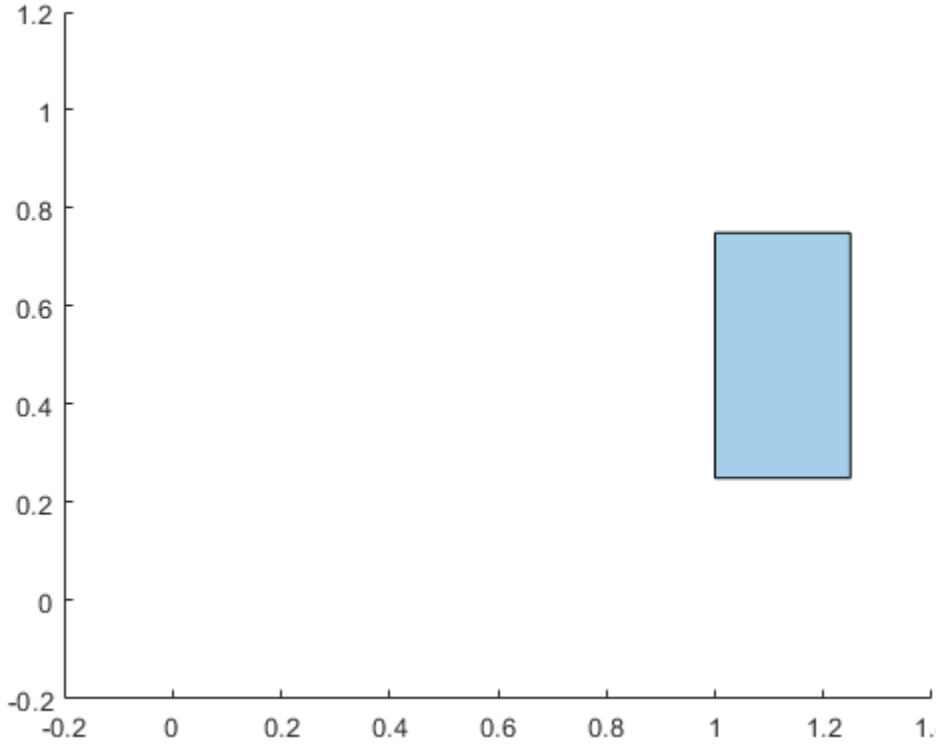
Now subtract the two polygons in the opposite order, that is, subtract `poly1` from `poly2`. The resulting polygon is `poly2` minus any part of `poly1` that overlaps `poly2`.

```
polyout2 = subtract(poly2,poly1)
```

```
polyout2 
  polyshape with properties:
```

```
      Vertices: [4x2 double]
    NumRegions: 1
      NumHoles: 0
```

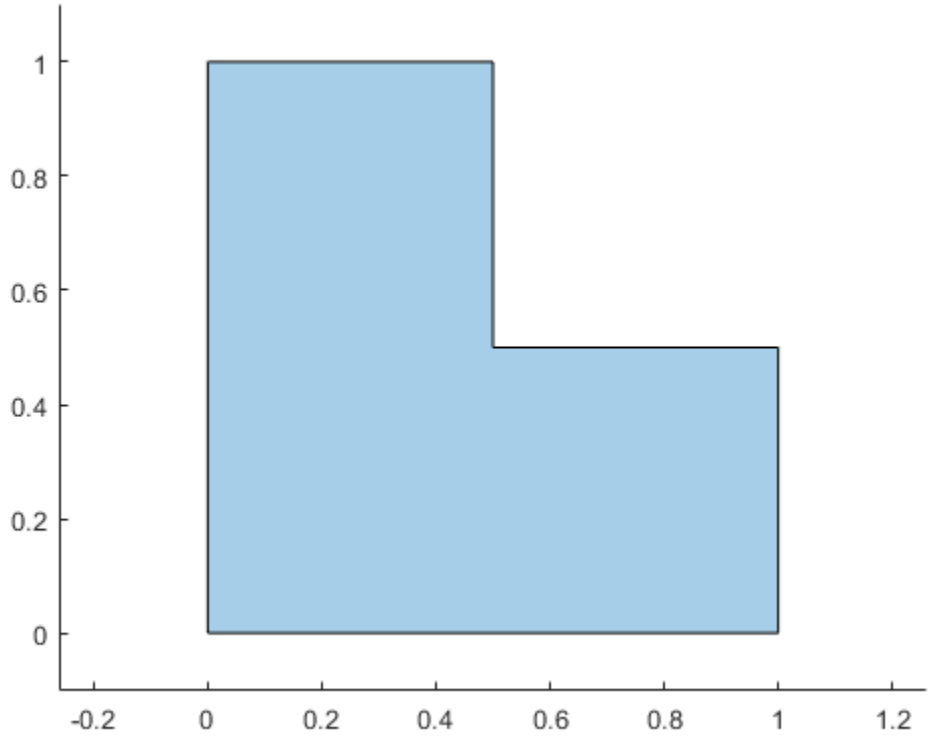
```
plot(polyout2)
xlim([-0.2 1.4]);
ylim([-0.2 1.2]);
```



Vertex Mapping

Create two polygons, and compute and plot their difference. Display the vertex coordinates of the difference and the corresponding vertex mapping information.

```
poly1 = polyshape([0 0 1 1],[1 0 0 1]);
poly2 = translate(poly1,[0.5 0.5]);
[polyout,shapeID,vertexID] = subtract(poly1,poly2);
plot(polyout)
axis equal
```




```
[polyout.Vertices shapeID vertexID]
```

```
ans = 6×4
```

```
         0    1.0000    1.0000    1.0000
    0.5000    1.0000         0         0
    0.5000    0.5000    2.0000    4.0000
    1.0000    0.5000         0         0
    1.0000         0    1.0000    3.0000
         0         0    1.0000    4.0000
```

The first, fifth, and sixth vertices of the difference originated in `poly1`, since the corresponding values in `shapeID` are 1. These vertices are the first, third, and fourth vertices in the property `poly1.Vertices`, respectively, since the corresponding values in `vertexID` are 1, 3, and 4. Similarly, the third vertex of the difference originated in `poly2`, and is the fourth vertex in the property `poly2.Vertices`. The second and fourth vertices of the difference were created from the subtraction computation because the corresponding values of `shapeID` and `vertexID` are 0.

Input Arguments

`poly1` — First input `polyshape`
scalar | vector | matrix | multidimensional array

First input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

`poly2` — Second input `polyshape`
scalar | vector | matrix | multidimensional array

Second input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

TF — Collinear vertices indicator
`false` | `true`

Collinear vertices indicator, specified as `false` or `true`:

- `false` — Remove collinear points so that the output `polyshape` contains the fewest vertices necessary to define the boundaries.
- `true` — Keep all collinear points as vertices.

When the `'KeepCollinearPoints'` parameter is not specified, its value is assigned according to the values used when creating the input `polyshape` objects:

- If the value was `true` for each input `polyshape` when they were created, then the value for the output `polyshape` is set to `true`.
- If the value was `false` for each input `polyshape` when they were created, then the value for the output `polyshape` is set to `false`.
- If the values for the input `polyshape` objects do not match, then the value for the output `polyshape` is set to `false`.

Data Types: `logical`

Output Arguments

`polyout` — Output `polyshape`
scalar | vector | matrix | multidimensional array

Output `polyshape`, returned as a scalar, vector, matrix, or multidimensional array.

The two input `polyshape` arguments must have compatible sizes. For example, if two input `polyshape` vectors have different lengths *M* and *N*, then they must have different orientations (one must be a row vector and one must be a column vector). `polyout` is then *M*-by-*N* or *N*-by-*M* depending on the orientation of each input vector. For more information on compatible array sizes, see “Compatible Array Sizes for Basic Operations”.

`shapeID` — Shape ID
column vector

Shape ID, returned as a column vector whose elements each represent the origin of the vertex in the difference. The value of an element in `shapeID` is 0 when the corresponding vertex of the output `polyshape` was created by the subtraction. An element is 1 when the corresponding vertex originated from `poly1`, and 2 when it originated from `poly2`.

The length of `shapeID` is equal to the number of rows in the `Vertices` property of the output `polyshape`. The `xor` function only supports this output argument if the input `polyshape` objects are scalar.

Data Types: `double`

`vertexID` — Vertex ID
column vector

Vertex ID, returned as a column vector whose elements map the vertices in the output `polyshape` to the vertices in the `polyshape` of origin. The elements of `vertexID`

contain the row numbers of the corresponding vertices in the `Vertices` property of the input `polyshape`. An element is 0 when the corresponding vertex of the output `polyshape` was created by the difference.

The length of `vertexID` is equal to the number of rows in the `Vertices` property of the output `polyshape`. The `subtract` function only supports this output argument when the input `polyshape` objects are scalar.

Data Types: `double`

See Also

`intersect` | `polyshape` | `union` | `xor`

Introduced in R2017b

translate

Translate `polyshape`

Syntax

```
polyout = translate(polyin,v)
polyout = translate(polyin,x,y)
```

Description

`polyout = translate(polyin,v)` returns a `polyshape` object defined by translating `polyin` by `v`. The first element of `v` specifies the distance to translate in the *x* direction, and the second element specifies the distance to translate in the *y* direction.

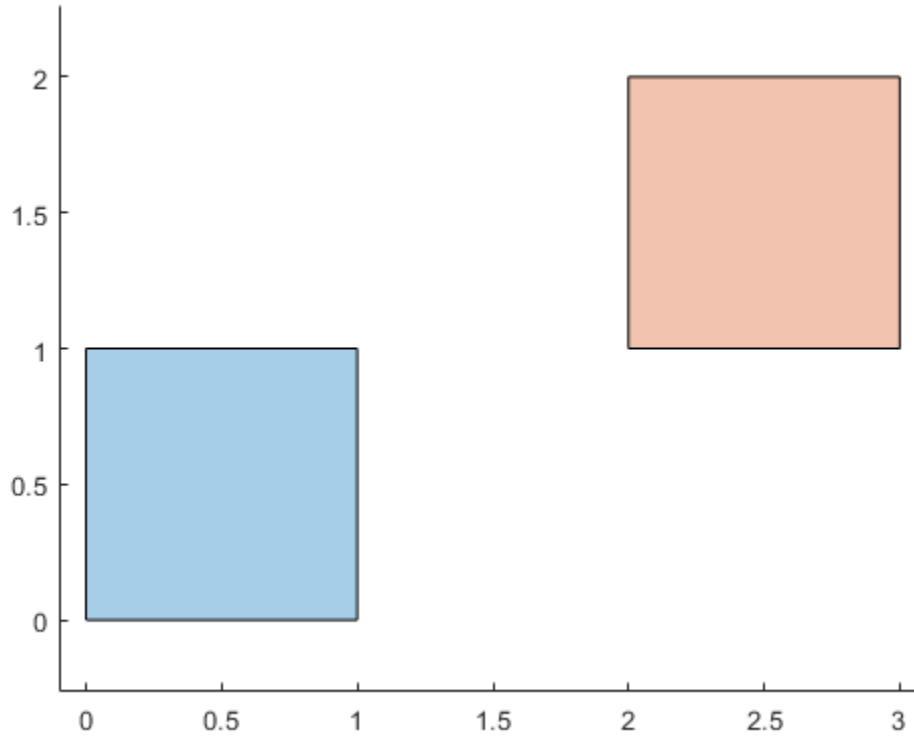
`polyout = translate(polyin,x,y)` specifies the *x* and *y* translation amounts as separate arguments.

Examples

Translate Square

Create a square, then translate the square to the right by two units, and up by one unit. Plot both squares.

```
polyin = polyshape([0 0 1 1],[1 0 0 1]);
polyout = translate(polyin,[2 1]);
plot([polyin polyout])
axis equal
```



Input Arguments

`polyin` — Input `polyshape`
scalar | vector | matrix | multidimensional array

Input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `polyshape`

`v` — Translation distance
two-element row vector

Translation distance, specified as a two-element row vector. The first element of `v` specifies the distance to translate in the *x* direction, and the second element specifies the distance to translate in the *y* direction. Positive distances translate right and up, and negative distances translate left and down. When the input `polyshape` is an array, each element of the array is translated according to `v`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

x — *x* translation distance
scalar

x translation distance, specified as a scalar. Positive distances translate right, and negative distances translate left. When the input `polyshape` is an array, each element of the array is translated in the *x* direction according to `x`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

y — *y* translation distance
scalar

y translation distance, specified as a scalar. Positive distances translate up, and negative distances translate down. When the input `polyshape` is an array, each element of the array is translated in the *y* direction according to `y`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

See Also

`polyshape` | `rotate` | `scale`

Introduced in R2017b

triangulation

Triangulate `polyshape`

Syntax

```
T = triangulation(polyin)
```

Description

`T = triangulation(polyin)` returns a 2-D `triangulation` object of a `polyshape` object. `T` has properties that describe the vertices and connectivity of triangles that make up `polyin`. For more information, see `triangulation`.

Examples

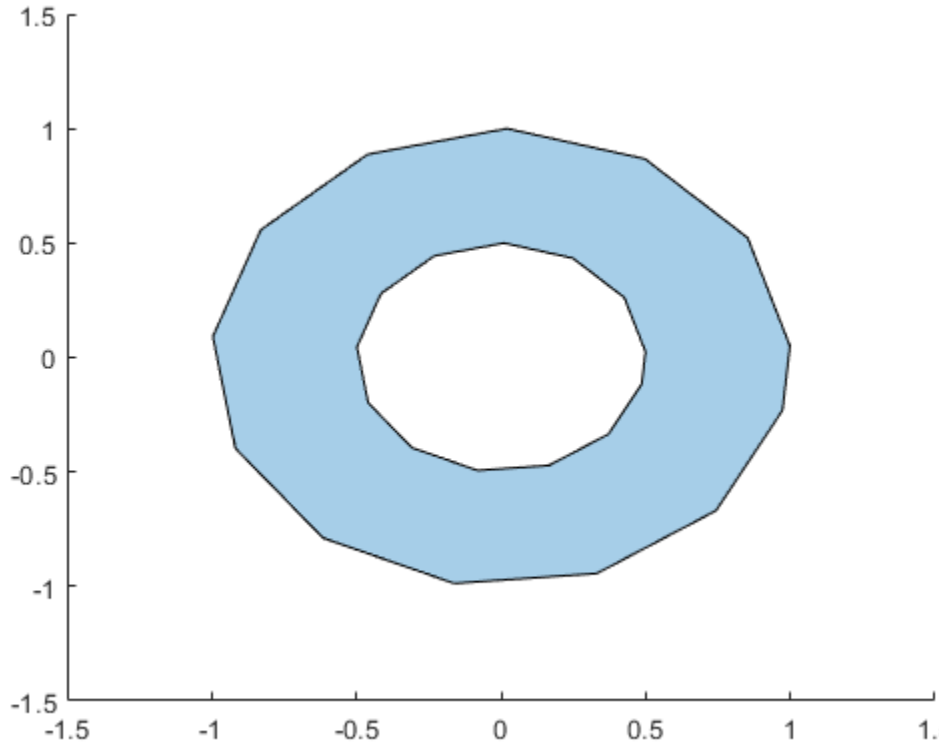
Polygon Triangulation

Create and triangulate a polygon.

```
t = 0.05:0.5:2*pi;
x1 = cos(t);
y1 = sin(t);
x2 = 0.5*cos(t);
y2 = 0.5*sin(t);
polyin = polyshape({x1,x2},{y1,y2})
```

```
polyin = 
  polyshape with properties:

      Vertices: [27x2 double]
    NumRegions: 1
      NumHoles: 1
```

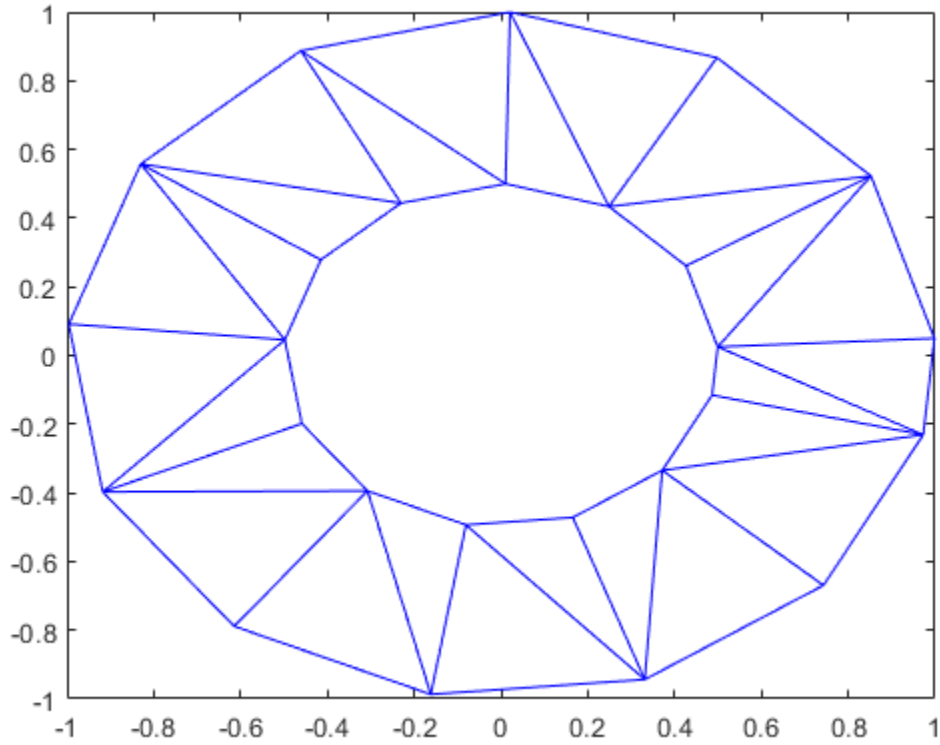
```
plot(polyin)
```




```
T = triangulation(polyin)
```

```
T = 
  triangulation with properties:

              Points: [26x2 double]
    ConnectivityList: [26x3 double]
```

```
triplot(T)
```



Input Arguments

`polyin` — Input `polyshape`
scalar

Input `polyshape`, specified as a scalar.

Data Types: `polyshape`

See Also

`polyshape` | `triangulation`

Introduced in R2017b

turningdist

Compute turning distance between `polyshape` objects

Syntax

```
td = turningdist(poly1,poly2)
td = turningdist(polyvec)
```

Description

`td = turningdist(poly1,poly2)` returns an array of turning distances between the corresponding element pairs of two `polyshape` arrays with compatible sizes. The turning distance between two `polyshape` objects is a measure of how closely their shapes match, regardless of rotation or scaling. A turning distance close to 0 indicates a near match. The larger the value, the more the two shapes differ.

TD(i,j) is the turning distance between the *i*th `polyshape` in `poly1` and the *j*th `polyshape` in `poly2`.

`td = turningdist(polyvec)` returns a matrix of turning distances between element pairs of a vector of `polyshape` objects.

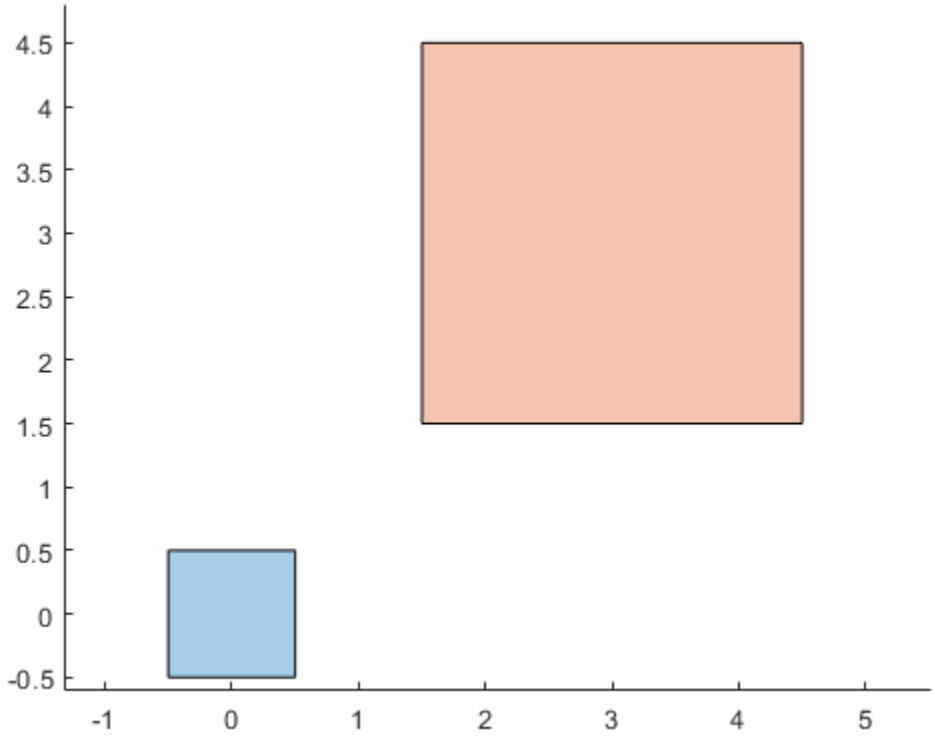
Examples

Turning Distance Between Two Squares

Create and plot two squares of different sizes and locations, represented as `polyshape` objects `poly1` and `poly2`.

```
poly1 = nsidedpoly(4,'SideLength',1);
poly2 = nsidedpoly(4,'SideLength',3,'Center',[3 3]);
plot(poly1)
hold on
plot(poly2)
```

```
axis equal
hold off
```



Because the two squares have the same shape despite their scaling, their turning distance is 0.

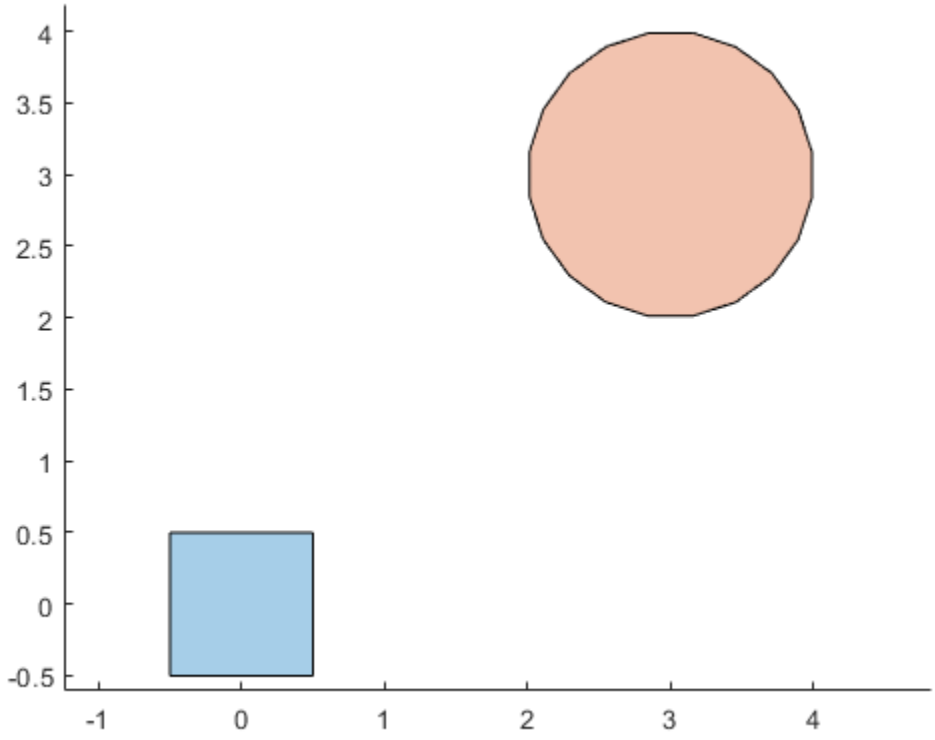
```
td = turningdist(poly1,poly2)
```

```
td = 0
```

Create and plot a third `polyshape`, and compare its turning distance to `poly1`. Since their shapes have more differences than `poly1` and `poly2`, the turning distance is larger.

```
poly3 = nsidedpoly(20,'Center',[3 3]);
plot(poly1)
```

```
hold on
plot(poly3)
axis equal
hold off
```




```
td = turningdist(poly1,poly3)
```

```
td = 0.4443
```

Input Arguments

poly1 — First input polyshape
scalar | vector | matrix | multidimensional array

First input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

poly2 — Second input polyshape
scalar | vector | matrix | multidimensional array

Second input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

polyvec — polyshape vector
vector

`polyshape` vector.

Output Arguments

td — Turning distance
scalar | vector | matrix | multidimensional array

Turning distance, returned as a scalar, vector, matrix, or multidimensional array whose elements are greater than or equal to 0.

- If you input two `polyshape` arguments, then they must have compatible sizes. For example, if two input `polyshape` vectors have different lengths *M* and *N*, then they must have different orientations (one must be a row vector and one must be a column vector). `td` is then *M*-by-*N* or *N*-by-*M* depending on the orientation of each input vector. For more information on compatible array sizes, see “Compatible Array Sizes for Basic Operations”.
- If you input a single `polyshape` vector with length *N*, then `td` is *N*-by-*N*.

Data Types: `double`

References

[1] Arkin, E.M., Chew, L.P., Huttenlocher, D.P., Kedem, K., and Mitchell, J.S.B. "An efficiently computable metric for comparing polygonal shapes." *IEEE Transactions*

on Pattern Analysis and Machine Intelligence. Vol. 13, Number 3, 1991, pp. 209-16. doi:10.1109/34.75509.

See Also

`polyshape`

Introduced in R2018a

union

Union of `polyshape` objects

Syntax

```
polyout = union(poly1,poly2)
polyout = union(polyvec)
[polyout,shapeID,vertexID] = union(poly1,poly2)
[polyout,shapeID,vertexID] = union(polyvec)
___ = union(___,'KeepCollinearPoints',TF)
```

Description

`polyout = union(poly1,poly2)` returns a `polyshape` object whose regions are the union of two `polyshape` objects. The union contains the combined regions of `poly1` and `poly2`, which must have compatible array sizes.

`polyout = union(polyvec)` returns a `polyshape` object whose regions are the geometric union of all `polyshape` objects in the vector `polyvec`. The union contains the combined regions of the `polyshape` objects in `polyvec`.

`[polyout,shapeID,vertexID] = union(poly1,poly2)` also returns vertex mapping information from the vertices in `polyout` to the vertices in `poly1` and `poly2`. The `union` function only supports this syntax when `poly1` and `poly2` are scalar `polyshape` objects.

The `shapeID` elements identify whether the corresponding vertex in `polyout` originated in `poly1`, `poly2`, or was created from the union. `vertexID` maps the vertices of `polyout` to the vertices of `poly1`, `poly2`, or the union.

`[polyout,shapeID,vertexID] = union(polyvec)` returns vertex mapping information from `polyvec` to each element of the vector of `polyshape` objects `polyvec`.

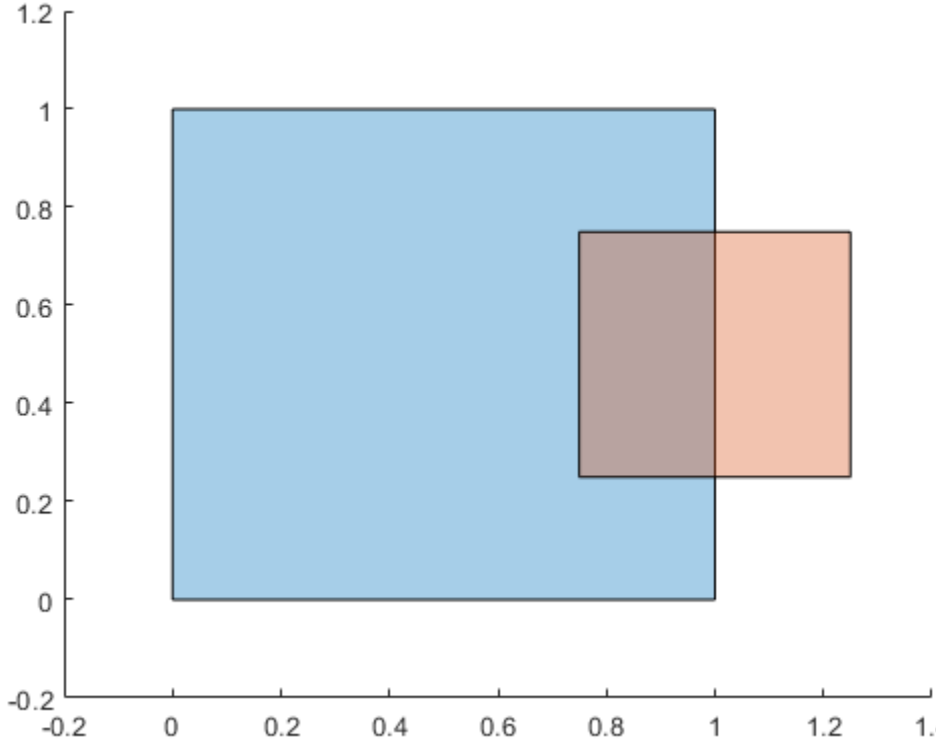
`___ = union(___,'KeepCollinearPoints',TF)` specifies whether to keep or remove collinear points in `polyout` for any of the previous syntaxes.

Examples

Union of Two Polygons

Create and plot two polygons.

```
poly1 = polyshape([0 0 1 1],[1 0 0 1]);
poly2 = polyshape([0.75 1.25 1.25 0.75],[0.25 0.25 0.75 0.75]);
plot(poly1)
hold on
plot(poly2)
```



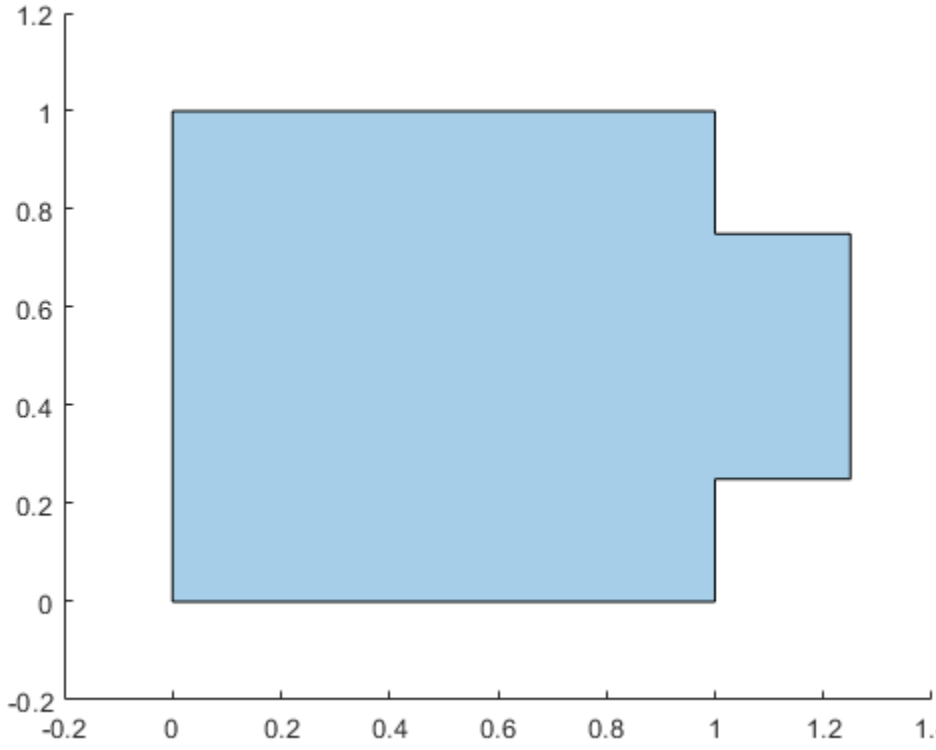

```
figure
```

Compute and plot the union of `poly1` and `poly2`.

```
polyout = union(poly1,poly2)
```

```
polyout 
  polyshape with properties:

      Vertices: [8x2 double]
    NumRegions: 1
      NumHoles: 0
```

```
plot(polyout)
xlim([-0.2 1.4]);
ylim([-0.2 1.2]);
```



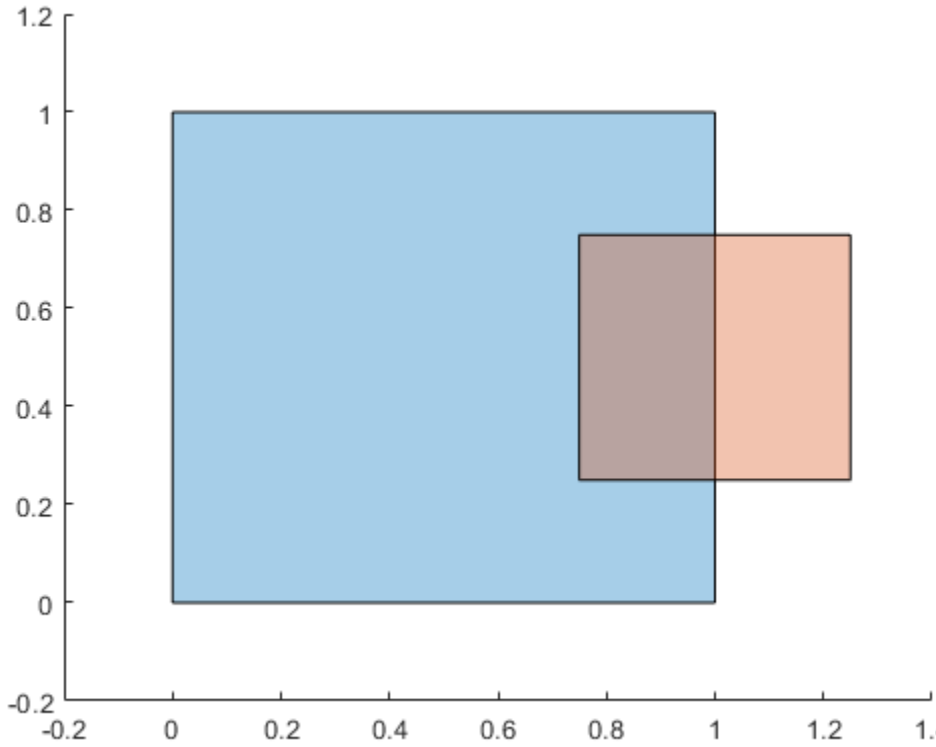
Vector of Polygons

Create a vector of polygons and plot each polygon.

```
polyarray1 = polyshape([0 0 1 1],[1 0 0 1]);
polyarray2 = polyshape([0.75 1.25 1.25 0.75],[0.25 0.25 0.75 0.75]);
poly1 = [polyarray1 polyarray2]
```

```
poly1 
  1x2 polyshape array with properties:

    Vertices
    NumRegions
    NumHoles
```

```
plot(poly1(1))
hold on
plot(poly1(2))
```



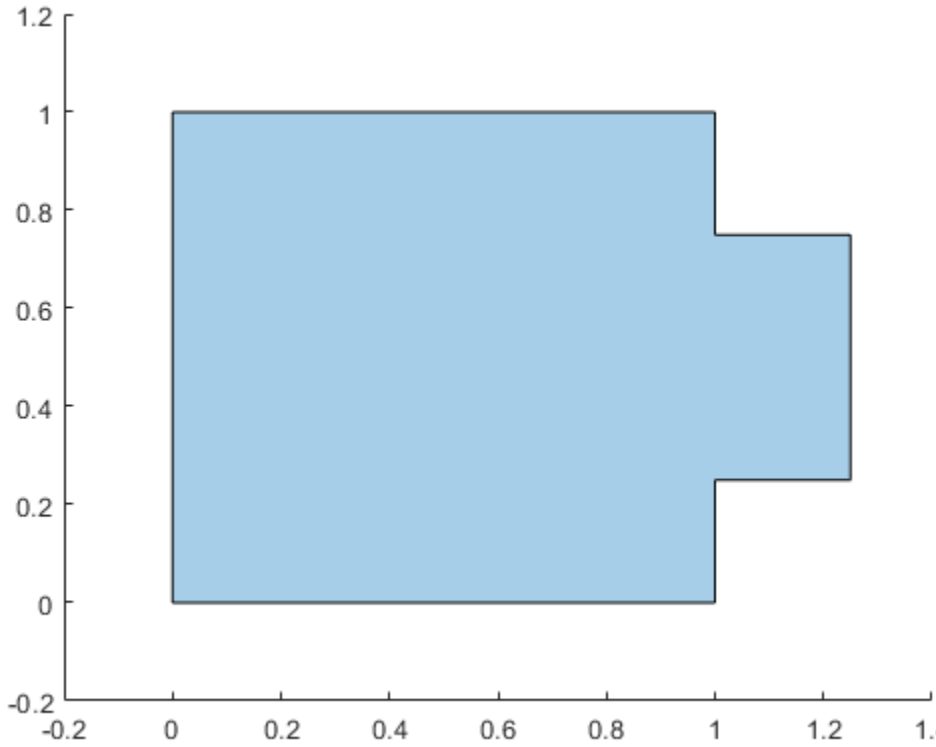

```
figure
```

Compute and plot the union of the two polygons.

```
polyout = union(poly1)
```

```
polyout 
  polyshape with properties:

      Vertices: [8x2 double]
    NumRegions: 1
      NumHoles: 0
```

```
plot(polyout)
xlim([-0.2 1.4]);
ylim([-0.2 1.2]);
```

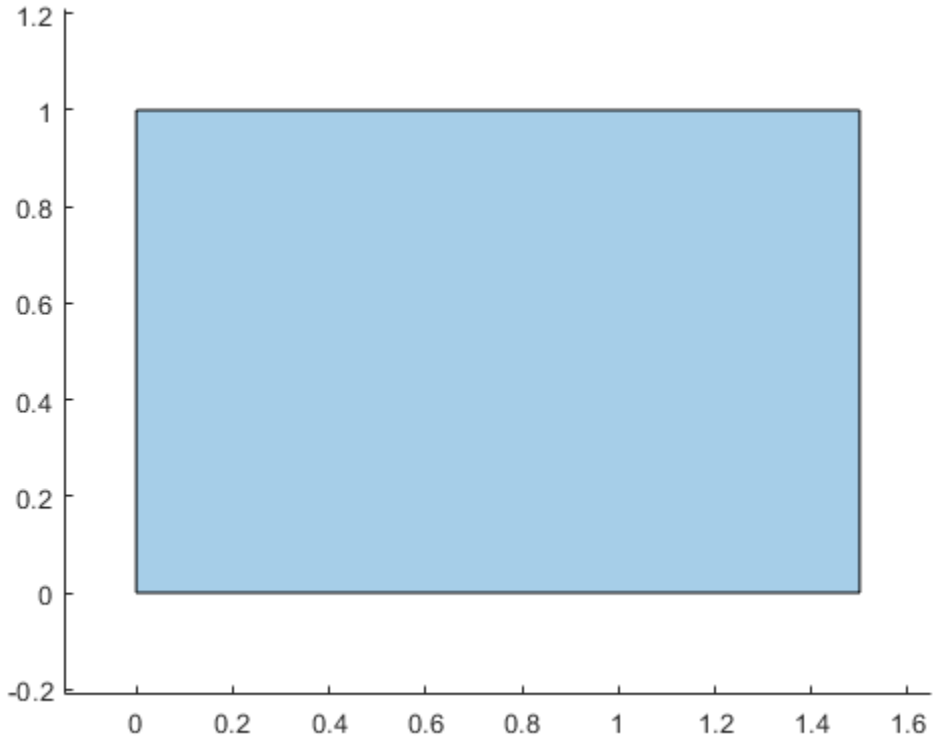


Vertex Mapping

Create two polygons, and compute and plot their union. Display the vertex coordinates of the union and the corresponding vertex mapping information.

```
poly1 = polyshape([0 0 1 1],[1 0 0 1]);
poly2 = translate(poly1,[0.5 0]);
```

```
[polyout,shapeID,vertexID] = union(poly1,poly2);
plot(polyout)
axis equal
```




```
[polyout.Vertices shapeID vertexID]
```

```
ans = 4×4

         0    1.0000    1.0000    1.0000
    1.5000    1.0000    2.0000    2.0000
    1.5000         0    2.0000    3.0000
         0         0    1.0000    4.0000
```

The first and last vertices of the union originated in `poly1`, since the corresponding values in `shapeID` are 1. These vertices are the first and fourth vertices in the property `poly1.Vertices`, respectively, since the corresponding values in `vertexID` are 1 and 4. Similarly, the second and third vertices of the union originated in `poly2`, and they are the second and third vertices in the property `poly2.Vertices`, respectively.

Input Arguments

`poly1` — First input `polyshape`
scalar | vector | matrix | multidimensional array

First input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

`poly2` — Second input `polyshape`
scalar | vector | matrix | multidimensional array

Second input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

`polyvec` — `polyshape` vector
vector

`polyshape` vector.

TF — Collinear vertices indicator
`false` | `true`

Collinear vertices indicator, specified as `false` or `true`:

- `false` — Remove collinear points so that the output `polyshape` contains the fewest vertices necessary to define the boundaries.
- `true` — Keep all collinear points as vertices.

When the `'KeepCollinearPoints'` parameter is not specified, its value is assigned according to the values used when creating the input `polyshape` objects:

- If the value was `true` for each input `polyshape` when they were created, then the value for the output `polyshape` is set to `true`.
- If the value was `false` for each input `polyshape` when they were created, then the value for the output `polyshape` is set to `false`.

- If the values for the input `polyshape` objects do not match, then the value for the output `polyshape` is set to `false`.

Data Types: `logical`

Output Arguments

`polyout` — Output `polyshape`
scalar | vector | matrix | multidimensional array

Output `polyshape`, returned as a scalar, vector, matrix, or multidimensional array.

- If you input two `polyshape` arguments, then they must have compatible sizes. For example, if two input `polyshape` vectors have different lengths *M* and *N*, then they must have different orientations (one must be a row vector and one must be a column vector). `polyout` is then *M*-by-*N* or *N*-by-*M* depending on the orientation of each input vector. For more information on compatible array sizes, see “Compatible Array Sizes for Basic Operations”.
- If you provide a single input argument `polyvec`, then `polyout` is a scalar `polyshape` object.

`shapeID` — Shape ID
column vector

Shape ID, returned as a column vector whose elements each represent the origin of the vertex in the union.

- The length of `shapeID` is equal to the number of rows in the `Vertices` property of the output `polyshape`.
- The elements of `shapeID` depend on the number of input arguments:
  - If you provide two input arguments `poly1` and `poly2`, then they must be scalar `polyshape` objects. The value of an element in `shapeID` is 0 when the corresponding vertex of the output `polyshape` was created by the union. An element is 1 when the corresponding vertex originated from `poly1`, and 2 when it originated from `poly2`.
  - If you provide one input argument `polyvec` that is a vector of `polyshape` objects, then `shapeID` contains the element index of `polyvec` from which the corresponding output vertex originated. The value of an element is 0 when the corresponding vertex was created by the union.

Data Types: `double`

`vertexID` — Vertex ID
column vector

Vertex ID, returned as a column vector whose elements map the vertices in the output `polyshape` to the vertices in the `polyshape` of origin. The elements of `vertexID` contain the row numbers of the corresponding vertices in the `Vertices` property of the input `polyshape`. An element is 0 when the corresponding vertex of the output `polyshape` was created by the union.

The length of `vertexID` is equal to the number of rows in the `Vertices` property of the output `polyshape`. If you provide two input `polyshape` objects, then `union` only supports this output argument if they are scalar.

Data Types: `double`

See Also

`intersect` | `polyshape` | `xor`

Introduced in R2017b

xor

Exclusive OR of two `polyshape` objects

Syntax

```
polyout = xor(poly1,poly2)
[polyout,shapeID,vertexID] = xor(poly1,poly2)
___ = xor(poly1,poly2,'KeepCollinearPoints',TF)
```

Description

`polyout = xor(poly1,poly2)` returns a `polyshape` object whose regions are the geometric exclusive OR of two `polyshape` objects. The geometric exclusive OR contains the regions of `poly1` and `poly2` that do not overlap. `poly1` and `poly2` must have compatible array sizes.

`[polyout,shapeID,vertexID] = xor(poly1,poly2)` also returns vertex mapping information from the vertices in `polyout` to the vertices in `poly1` and `poly2`. The `xor` function only supports this syntax when `poly1` and `poly2` are scalar `polyshape` objects.

The `shapeID` elements identify whether the corresponding vertex in `polyout` originated in `poly1`, `poly2`, or was created from the exclusive OR. `vertexID` maps the vertices of `polyout` to the vertices of `poly1`, `poly2`, or the exclusive OR.

`___ = xor(poly1,poly2,'KeepCollinearPoints',TF)` specifies whether to keep or remove collinear points in `polyout` for any of the previous syntaxes.

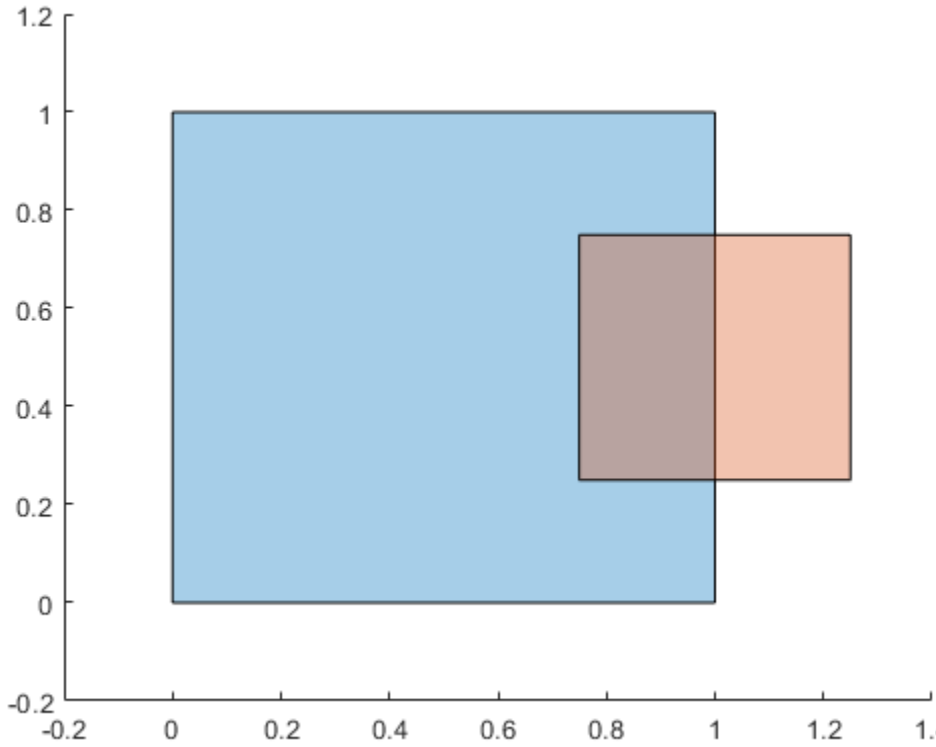
Examples

Exclusive OR

Create and plot two polygons.

```
poly1 = polyshape([0 0 1 1],[1 0 0 1]);
poly2 = polyshape([0.75 1.25 1.25 0.75],[0.25 0.25 0.75 0.75]);
```

```
plot(poly1)
hold on
plot(poly2)
```




```
figure
```

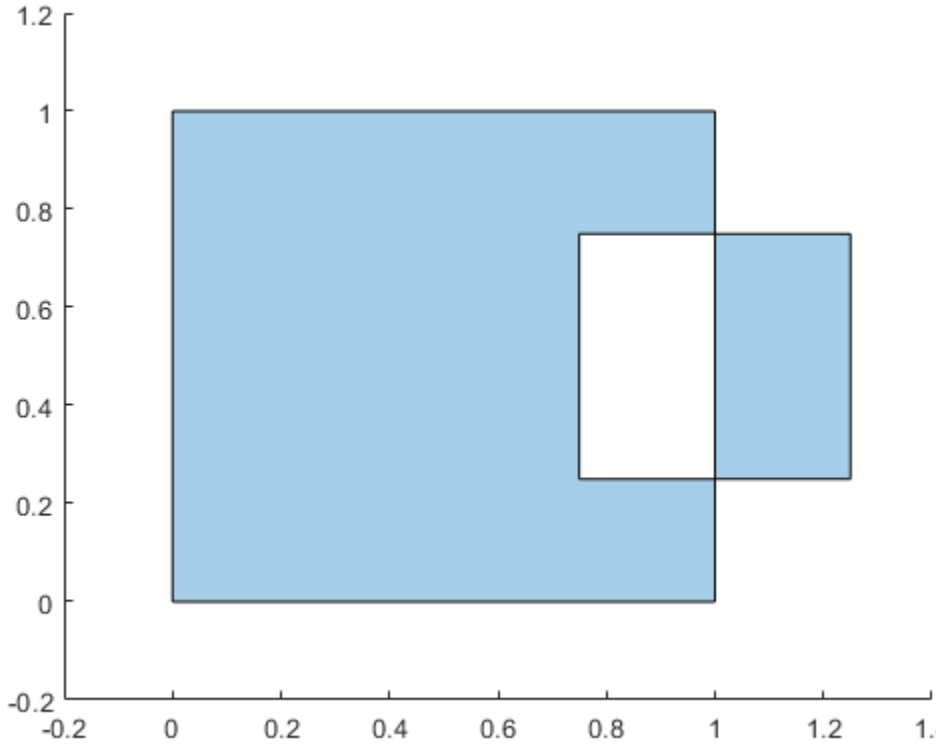
Compute and plot the exclusive OR of `poly1` and `poly2`.

```
polyout = xor(poly1,poly2)
```

```
polyout 
  polyshape with properties:

      Vertices: [13x2 double]
    NumRegions: 2
```

```
      NumHoles: 0
```

```
plot(polyout)
```

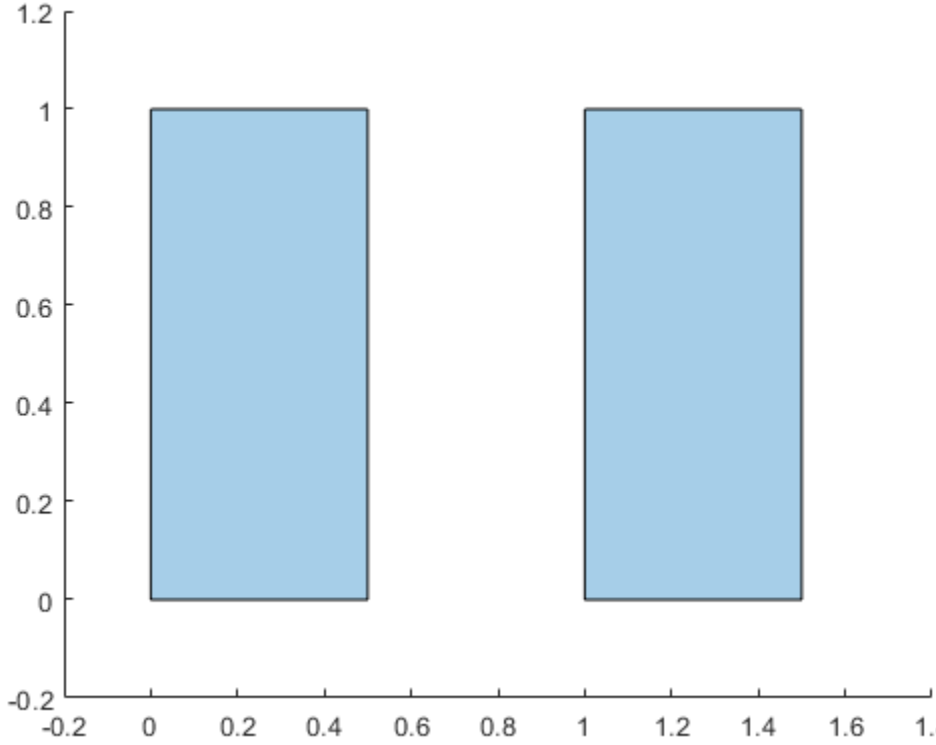


Vertex Mapping

Create two polygons, and compute and plot their exclusive OR. Display the vertex coordinates of the exclusive OR and the corresponding vertex mapping information.

```
poly1 = polyshape([0 0 1 1],[1 0 0 1]);
poly2 = translate(poly1,[0.5 0]);
```

```
[polyout,shapeID,vertexID] = xor(poly1,poly2);
plot(polyout)
```




```
[polyout.Vertices shapeID vertexID]
```

```
ans = 9×4

         0    1.0000    1.0000    1.0000
    0.5000    1.0000    2.0000    1.0000
    0.5000         0    2.0000    4.0000
         0         0    1.0000    4.0000
       NaN       NaN       NaN       NaN
    1.0000    1.0000    1.0000    2.0000
    1.5000    1.0000    2.0000    2.0000
```

```
    1.5000         0    2.0000    3.0000
    1.0000         0    1.0000    3.0000
```

There are two boundaries in the exclusive OR, separated by a row of `NaN` values in the `Vertices` property. For example, consider the first boundary in the array. The first and last vertices of the exclusive OR originated in `poly1`, since the corresponding values in `shapeID` are 1. These vertices are the first and fourth vertices in the property `poly1.Vertices`, respectively, since the corresponding values in `vertexID` are 1 and 4. Similarly, the second and third vertices of the exclusive OR originated in `poly2`, and they are also the first and fourth vertices in the property `poly2.Vertices`, respectively.

Input Arguments

`poly1` — First input `polyshape`
scalar | vector | matrix | multidimensional array

First input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

`poly2` — Second input `polyshape`
scalar | vector | matrix | multidimensional array

Second input `polyshape`, specified as a scalar, vector, matrix, or multidimensional array.

`TF` — Collinear vertices indicator
`false` | `true`

Collinear vertices indicator, specified as `false` or `true`:

- `false` — Remove collinear points so that the output `polyshape` contains the fewest vertices necessary to define the boundaries.
- `true` — Keep all collinear points as vertices.

When the `'KeepCollinearPoints'` parameter is not specified, its value is assigned according to the values used when creating the input `polyshape` objects:

- If the value was `true` for each input `polyshape` when they were created, then the value for the output `polyshape` is set to `true`.
- If the value was `false` for each input `polyshape` when they were created, then the value for the output `polyshape` is set to `false`.

- If the values for the input `polyshape` objects do not match, then the value for the output `polyshape` is set to `false`.

Data Types: `logical`

Output Arguments

`polyout` — Output `polyshape`
scalar | vector | matrix | multidimensional array

Output `polyshape`, returned as a scalar, vector, matrix, or multidimensional array.

The two input `polyshape` arguments must have compatible sizes. For example, if two input `polyshape` vectors have different lengths *M* and *N*, then they must have different orientations (one must be a row vector and one must be a column vector). `polyout` is then *M*-by-*N* or *N*-by-*M* depending on the orientation of each input vector. For more information on compatible array sizes, see “Compatible Array Sizes for Basic Operations”.

`shapeID` — Shape ID
column vector

Shape ID, returned as a column vector whose elements each represent the origin of the vertex in the exclusive OR. The value of an element in `shapeID` is 0 when the corresponding vertex of the output `polyshape` was created by the exclusive OR. An element is 1 when the corresponding vertex originated from `poly1`, and 2 when it originated from `poly2`.

The length of `shapeID` is equal to the number of rows in the `Vertices` property of the output `polyshape`. The `xor` function only supports this output argument if the input `polyshape` objects are scalar.

Data Types: `double`

`vertexID` — Vertex ID
column vector

Vertex ID, returned as a column vector whose elements map the vertices in the output `polyshape` to the vertices in the `polyshape` of origin. The elements of `vertexID` contain the row numbers of the corresponding vertices in the `Vertices` property of the input `polyshape`. An element is 0 when the corresponding vertex of the output `polyshape` was created by the exclusive OR.

The length of `vertexID` is equal to the number of rows in the `Vertices` property of the output `polyshape`. The `xor` function only supports this output argument when the input `polyshape` objects are scalar.

Data Types: `double`

See Also

`polyshape` | `subtract`

Introduced in R2017b

polyval

Polynomial evaluation

Syntax

```
y = polyval(p,x)
[y,delta] = polyval(p,x,S)
y = polyval(p,x,[],mu)
[y,delta] = polyval(p,x,S,mu)
```

Description

`y = polyval(p,x)` evaluates the polynomial `p` at each point in `x`. The argument `p` is a vector of length `n+1` whose elements are the coefficients (in descending powers) of an `n`th-degree polynomial:

$$p(x) = p_1x^n + p_2x^{n-1} + \ldots + p_nx + p_{n+1}.$$

The polynomial coefficients in `p` can be calculated for different purposes by functions like `polyint`, `polyder`, and `polyfit`, but you can specify any vector for the coefficients.

To evaluate a polynomial in a matrix sense, use `polyvalm` instead.

`[y,delta] = polyval(p,x,S)` uses the optional output structure `S` produced by `polyfit` to generate error estimates. `delta` is an estimate of the standard error in predicting a future observation at `x` by `p(x)`.

`y = polyval(p,x,[],mu)` or `[y,delta] = polyval(p,x,S,mu)` use the optional output `mu` produced by `polyfit` to center and scale the data. `mu(1)` is `mean(x)`, and `mu(2)` is `std(x)`. Using these values, `polyval` centers `x` at zero and scales it to have unit standard deviation,

$$\hat{x} = \frac{x - \bar{x}}{\sigma_x}.$$

This centering and scaling transformation improves the numerical properties of the polynomial.

Examples

Evaluate Polynomial at Several Points

Evaluate the polynomial $p(x) = 3x^2 + 2x + 1$ at the points $x = 5, 7, 9$. The polynomial coefficients can be represented by the vector `[3 2 1]`.

```
p = [3 2 1];
x = [5 7 9];
y = polyval(p,x)
```

```
y = 1×3

    86   162   262
```

Integrate Quartic Polynomial

Evaluate the definite integral

$$I = \int_{-1}^{3} \left(3x^4 - 4x^2 + 10x - 25\right) dx \, .$$

Create a vector to represent the polynomial integrand $3x^4 - 4x^2 + 10x - 25$. The x^3 term is absent and thus has a coefficient of 0.

```
p = [3 0 -4 10 -25];
```

Use `polyint` to integrate the polynomial using a constant of integration equal to `0`.

```
q = polyint(p)
```

```
q = 1×6

    0.6000         0   -1.3333    5.0000  -25.0000         0
```

Find the value of the integral by evaluating `q` at the limits of integration.

```
a = -1;
b = 3;
I = diff(polyval(q,[a b]))
```

```
I = 49.0667
```

Linear Regression With Error Estimate

Fit a linear model to a set of data points and plot the results, including an estimate of a 95% prediction interval.

Create a few vectors of sample data points *(x,y)*. Use `polyfit` to fit a first degree polynomial to the data. Specify two outputs to return the coefficients for the linear fit as well as the error estimation structure.

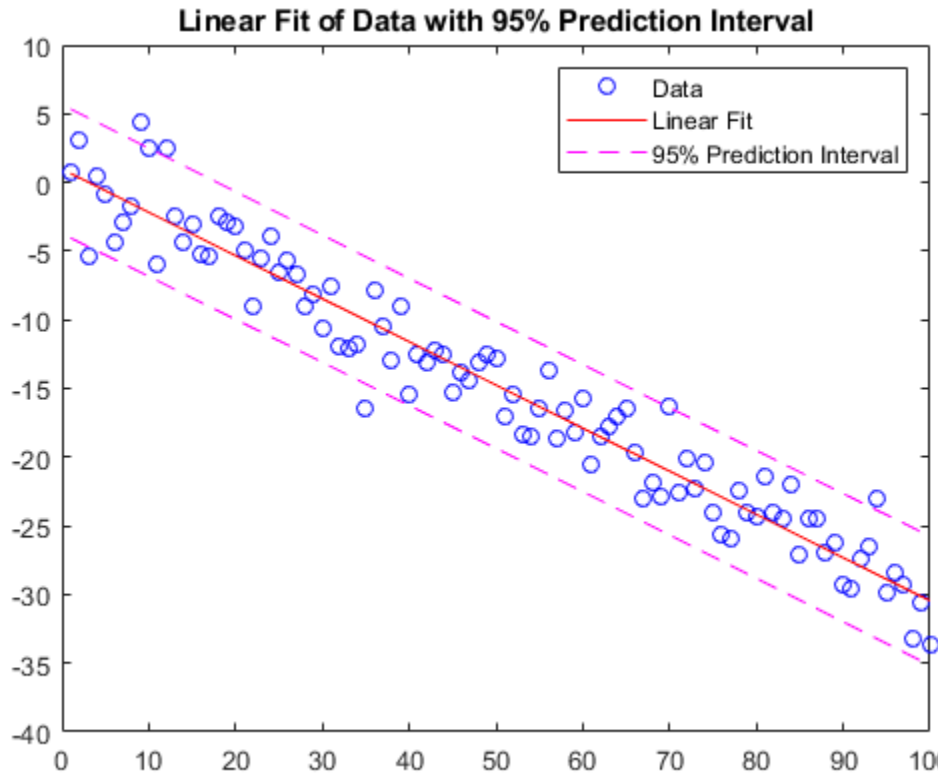
```
x = 1:100;
y = -0.3*x + 2*randn(1,100);
[p,S] = polyfit(x,y,1);
```

Evaluate the first-degree polynomial fit in `p` at the points in `x`. Specify the error estimation structure as the third input so that `polyval` calculates an estimate of the standard error. The standard error estimate is returned in `delta`.

```
[y_fit,delta] = polyval(p,x,S);
```

Plot the original data, linear fit, and 95% prediction interval $y \pm 2\Delta$.

```
plot(x,y,'bo')
hold on
plot(x,y_fit,'r-')
plot(x,y_fit+2*delta,'m--',x,y_fit-2*delta,'m--')
title('Linear Fit of Data with 95% Prediction Interval')
legend('Data','Linear Fit','95% Prediction Interval')
```



Use Centering and Scaling to Improve Numerical Properties

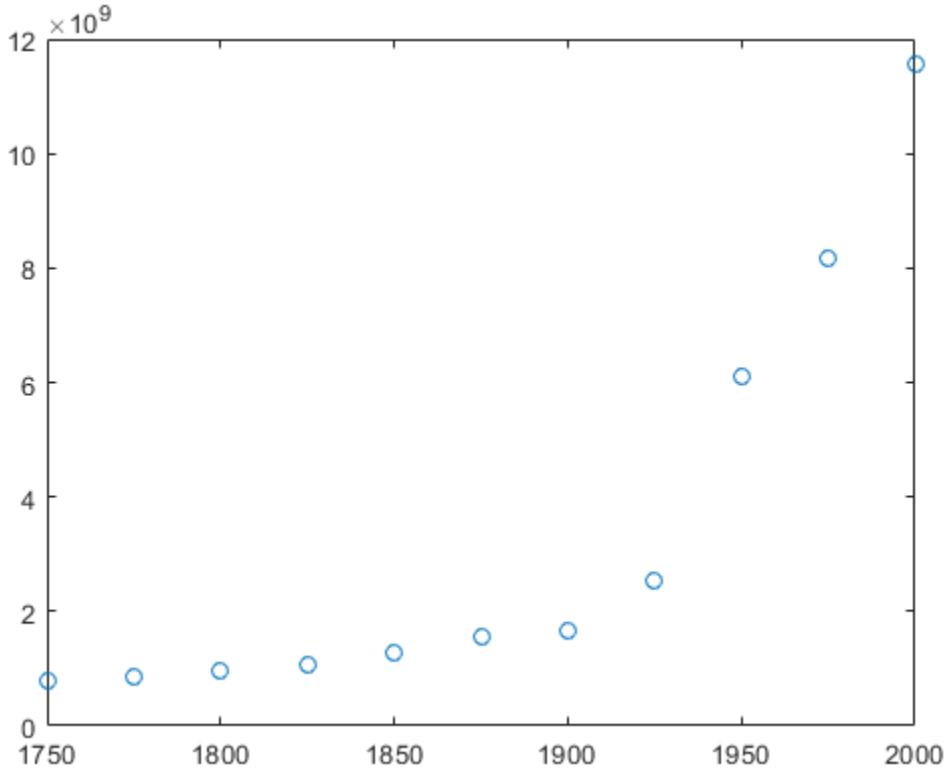
Create a table of population data for the years 1750 - 2000 and plot the data points.

```
year = (1750:25:2000)';
pop = 1e6*[791 856 978 1050 1262 1544 1650 2532 6122 8170 11560]';
T = table(year, pop)
```

```
T=11×2 table
    year       pop
    ____    _________
```

```
    1750     7.91e+08
    1775     8.56e+08
    1800     9.78e+08
    1825     1.05e+09
    1850    1.262e+09
    1875    1.544e+09
    1900     1.65e+09
    1925    2.532e+09
    1950    6.122e+09
    1975     8.17e+09
    2000    1.156e+10
```

```
plot(year,pop,'o')
```

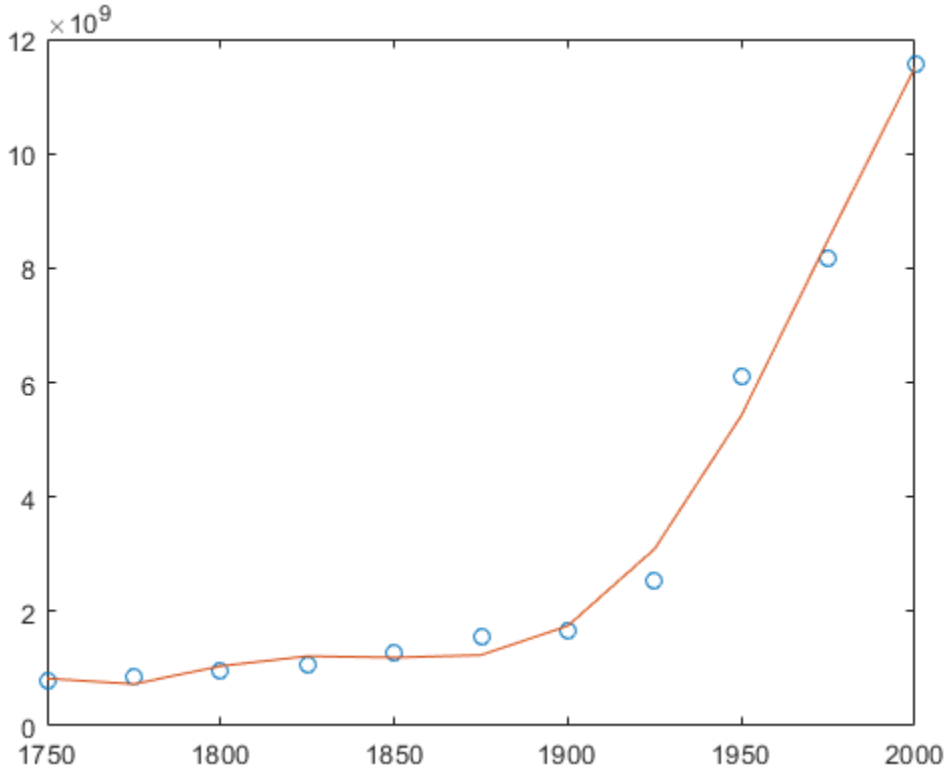


Use `polyfit` with three outputs to fit a 5th-degree polynomial using centering and scaling, which improves the numerical properties of the problem. `polyfit` centers the data in `year` at 0 and scales it to have a standard deviation of 1, which avoids an ill-conditioned Vandermonde matrix in the fit calculation.

```
[p,~,mu] = polyfit(T.year, T.pop, 5);
```

Use `polyval` with four inputs to evaluate `p` with the scaled years, `(year-mu(1))/mu(2)`. Plot the results against the original years.

```
f = polyval(p,year,[],mu);
hold on
plot(year,f)
hold off
```



Input Arguments

p — Polynomial coefficients
vector

Polynomial coefficients, specified as a vector. For example, the vector `[1 0 1]` represents the polynomial $x^2 + 1$, and the vector `[3.13 -2.21 5.99]` represents the polynomial $3.13x^2 - 2.21x + 5.99$.

For more information, see “Create and Evaluate Polynomials”.

Data Types: `single` | `double`
Complex Number Support: Yes

x — Query points
vector

Query points, specified as a vector. `polyval` evaluates the polynomial `p` at the points in `x` and returns the corresponding function values in `y`.

Data Types: `single` | `double`
Complex Number Support: Yes

S — Error estimation structure
structure

Error estimation structure. This structure is an optional output from `[p,S] = polyfit(x,y,n)` that can be used to obtain error estimates. `S` contains the following fields:

Field	Description
`R`	Triangular factor from a QR decomposition of the Vandermonde matrix of `x`
`df`	Degrees of freedom
`normr`	Norm of the residuals

If the data in `y` is random, then an estimate of the covariance matrix of `p` is `(Rinv*Rinv')*normr^2/df`, where `Rinv` is the inverse of `R`.

`mu` — Centering and scaling values

two-element vector

Centering and scaling values, specified as a two-element vector. This vector is an optional output from `[p,S,mu] = polyfit(x,y,n)` that is used to improve the numerical properties of fitting and evaluating the polynomial `p`. The value `mu(1)` is `mean(x)`, and `mu(2)` is `std(x)`. These values are used to center the query points in `x` at zero with unit standard deviation.

Specify `mu` to evaluate `p` at the scaled points, `(x - mu(1))/mu(2)`.

Output Arguments

`y` — Function values

vector

Function values, returned as a vector of the same size as the query points `x`. The vector contains the result of evaluating the polynomial `p` at each point in `x`.

`delta` — Standard error for prediction

scalar

Standard error for prediction, returned as a scalar. Generally, an interval of $y \pm \Delta$ corresponds to a roughly 68% prediction interval for future observations of large samples, and $y \pm 2\Delta$ a roughly 95% prediction interval.

If the coefficients in `p` are least-squares estimates computed by `polyfit`, and the errors in the data input to `polyfit` are independent, normal, and have constant variance, then $y \pm \Delta$ is at least a 50% prediction interval.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

If x is a tall array, then it must be a column vector.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`polyder` | `polyfit` | `polyint` | `polyvalm`

Topics

“Create and Evaluate Polynomials”
“Programmatic Fitting”

Introduced before R2006a

polyvalm

Matrix polynomial evaluation

Syntax

```
Y = polyvalm(p,X)
```

Description

`Y = polyvalm(p,X)` returns the evaluation of polynomial `p` in a matrix sense. This evaluation is the same as substituting matrix `X` in the polynomial, `p`.

Examples

Matrix Evaluation of Characteristic Polynomial

Find the characteristic polynomial of a Pascal Matrix of order 4.

```
X =  pascal(4)
```

```
X = 4×4

     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    20
```

```
p = poly(X)
```

```
p = 1×5

    1.0000  -29.0000   72.0000  -29.0000    1.0000
```

The characteristic polynomial is

$$p(x) = x^4 - 29x^3 + 72x^2 - 29x + 1$$

Pascal matrices have the property that the vector of coefficients of the characteristic polynomial is the same forward and backward (palindromic).

Substitute the matrix, `X`, into the characteristic equation, `p`. The result is very close to being a zero matrix. This example is an instance of the Cayley-Hamilton theorem, where a matrix satisfies its own characteristic equation.

```
Y = polyvalm(p,X)
```

```
Y = 4×4
10^-10 ×

   -0.0013   -0.0063   -0.0104   -0.0241
   -0.0048   -0.0217   -0.0358   -0.0795
   -0.0114   -0.0510   -0.0818   -0.1805
   -0.0228   -0.0970   -0.1553   -0.3396
```

Input Arguments

p — Polynomial coefficients
vector

Polynomial coefficients, specified as a vector. For example, the vector `[1 0 1]` represents the polynomial $x^2 + 1$, and the vector `[3.13 -2.21 5.99]` represents the polynomial $3.13x^2 - 2.21x + 5.99$.

For more information, see “Create and Evaluate Polynomials”.

Data Types: `single` | `double`
Complex Number Support: Yes

X — Input matrix
square matrix

Input matrix, specified as a square matrix.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Output polynomial coefficients
row vector

Output polynomial coefficients, returned as a row vector.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also
`polyfit` | `polyval`

Topics
“Create and Evaluate Polynomials”

Introduced before R2006a

posixtime

Convert MATLAB datetime to POSIX time

Syntax

```
p = posixtime(t)
```

Description

`p = posixtime(t)` returns POSIX times equivalent to the `datetime` values in `t`. The POSIX time is the number of seconds (including fractional seconds) elapsed since 00:00:00 1-Jan-1970 UTC (Universal Coordinated Time), ignoring leap seconds. `p` is a `double` array.

- If the time zone of `t` is not specified, then `posixtime` treats the times in `t` as UTC times. This interpretation might differ from your treatment of “unzoned” `datetime` arrays in other contexts. For example, you might think of `datetime('now')` as returning your local time. However, `posixtime` interprets it as a UTC time.
- If the time zone of `t` is specified, then `posixtime` uses the offset for the time zone to compute POSIX times with respect to UTC.

The best practice is to specify the time zone of `t` before calling `posixtime`.

Examples

Convert Datetime Array to POSIX Times

Create `datetime` values and convert them to the equivalent POSIX® times. Show the differences in POSIX times between zoned and unzoned `datetime` values. The best practice is to specify a time zone for a `datetime` array before calling `posixtime`.

Create a `datetime` array and specify its time zone.

```
t1 = datetime('2016-07-29 10:05:24') + calmonths(1:3);
t1.TimeZone = 'America/New_York'
```

```
t1 = 1x3 datetime array
   29-Aug-2016 10:05:24   29-Sep-2016 10:05:24   29-Oct-2016 10:05:24
```

Convert `t1` to the equivalent POSIX times. `posixtime` accounts for the time zone offset when it computes POSIX times.

```
format longG
p1 = posixtime(t1)
```

```
p1 = 1×3

                 1472479524                 1475157924                 1477749924
```

Create a `datetime` array with the same values as `t1`, but with no time zone. Convert it to the equivalent POSIX times. `posixtime` treats the times in `t2` as UTC times, with no time zone offset.

```
t2 = datetime('2016-07-29 10:05:24') + calmonths(1:3);
p2 = posixtime(t2)
```

```
p2 = 1×3

                 1472465124                 1475143524                 1477735524
```

Show the differences between `p2` and `p1`. The differences are equal to the time offset, in seconds, between UTC and the time zone of `t1`.

```
p2 - p1
```

```
ans = 1×3

      -14400      -14400      -14400
```

Input Arguments

t — Input date and time
`datetime` array

Input date and time, specified as a `datetime` array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`datenum` | `datetime` | `exceltime` | `juliandate` | `yyyymmdd`

Topics

“Convert Date and Time to Julian Date or POSIX Time”

Introduced in R2014b

pow2

Base 2 power and scale floating-point numbers

Syntax

```
X = pow2(Y)
X = pow2(F,E)
```

Description

`X = pow2(Y)` returns an array `X` whose elements are 2 raised to the power `Y`.

`X = pow2(F,E)` computes $x = f * 2^e$ for corresponding elements of `F` and `E`. The result is computed quickly by simply adding `E` to the floating-point exponent of `F`. Arguments `F` and `E` are real and integer arrays, respectively.

Examples

For IEEE arithmetic, the statement `X = pow2(F,E)` yields the values:

```
F         E        X
1/2       1        1
pi/4      2        pi
-3/4      2        -3
1/2       -51      eps
1-eps/2   1024     realmax
1/2       -1021    realmin
```

Tips

This function corresponds to the ANSI C function `ldexp()` and the IEEE floating-point standard function `scalbn()`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`exp` | `hex2num` | `log2` | `mpower` | `power` | `realmax` | `realmin`

Introduced before R2006a

power, .^

Element-wise power

Syntax

```
C = A.^B
C = power(A,B)
```

Description

`C = A.^B` raises each element of `A` to the corresponding powers in `B`. The sizes of `A` and `B` must be the same or be compatible.

If the sizes of `A` and `B` are compatible, then the two arrays implicitly expand to match each other. For example, if one of `A` or `B` is a scalar, then the scalar is combined with each element of the other array. Also, vectors with different orientations (one row vector and one column vector) implicitly expand to form a matrix.

`C = power(A,B)` is an alternate way to execute `A.^B`, but is rarely used. It enables operator overloading for classes.

Examples

Square Each Element of Vector

Create a vector, `A`, and square each element.

```
A = 1:5;
C = A.^2
```

```
C = 1×5

     1     4     9    16    25
```

Find Inverse of Each Matrix Element

Create a matrix, `A`, and take the inverse of each element.

```
A = [1 2 3; 4 5 6; 7 8 9];
C = A.^-1
```

```
C = 3×3

    1.0000    0.5000    0.3333
    0.2500    0.2000    0.1667
    0.1429    0.1250    0.1111
```

An inversion of the elements is not equal to the inverse of the matrix, which is instead written `A^-1` or `inv(A)`.

Row Vector to Power of Column Vector

Create a 1-by-2 row vector and a 3-by-1 column vector and raise the row vector to the power of the column vector.

```
a = [2 3];
b = (1:3)';
a.^b
```

```
ans = 3×2

     2     3
     4     9
     8    27
```

The result is a 3-by-2 matrix, where each (i,j) element in the matrix is equal to `a(j) .^ b(i)`:

$$a = [a_1\ a_2], \qquad b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}, \qquad a\ .\hat{}\ b = \begin{bmatrix} {a_1}^{b_1} & {a_2}^{b_1} \\ {a_1}^{b_2} & {a_2}^{b_2} \\ {a_1}^{b_3} & {a_2}^{b_3} \end{bmatrix}.$$

Find Roots of Number

Calculate the roots of `-1` to the `1/3` power.

```
A = -1;
B = 1/3;
C = A.^B
```

```
C = 0.5000 + 0.8660i
```

For negative base `A` and noninteger `B`, if `abs(B)` is less than `1`, the `power` function returns the complex roots of `A`.

Use the `nthroot` function to obtain the real roots.

```
C = nthroot(A,3)
```

```
C = -1
```

Input Arguments

A, B — Operands
scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. `A` and `B` must either be the same size or have sizes that are compatible (for example, `A` is an `M`-by-`N` matrix and `B` is a scalar or `1`-by-`N` row vector). For more information, see “Compatible Array Sizes for Basic Operations”.

- Operands with an integer data type cannot be complex.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`

Complex Number Support: Yes

More About

IEEE Compliance

For real inputs, `power` has a few behaviors that differ from those recommended in the IEEE-754 Standard.

	MATLAB	IEEE
`power(1,NaN)`	`NaN`	`1`
`power(NaN,0)`	`NaN`	`1`

Compatibility Considerations

Implicit expansion change affects arguments for operators

Behavior changed in R2016b

Starting in R2016b with the addition of implicit expansion, some combinations of arguments for basic operations that previously returned errors now produce results. For example, you previously could not add a row and a column vector, but those operands are now valid for addition. In other words, an expression like `[1 2] + [1; 2]` previously returned a size mismatch error, but now it executes.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a `try/catch` block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see “Compatible Array Sizes for Basic Operations”.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- When both `X` and `Y` are real, but `power(X,Y)` is complex, simulation produces an error and generated code returns `NaN`. To get the complex result, make the input value `X` complex by passing in `complex(X)`. For example, `power(complex(X),Y)`.
- When both `X` and `Y` are real, but `X .^ Y` is complex, simulation produces an error and generated code returns `NaN`. To get the complex result, make the input value `X` complex by using `complex(X)`. For example, `complex(X).^Y`.
- Code generation does not support sparse matrix inputs for this function.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- When both `X` and `Y` are real, but `power(X,Y)` is complex, simulation produces an error and generated code returns `NaN`. To get the complex result, make the input value `X` complex by passing in `complex(X)`. For example, `power(complex(X),Y)`.
- When both `X` and `Y` are real, but `X .^ Y` is complex, simulation produces an error and generated code returns `NaN`. To get the complex result, make the input value `X` complex by using `complex(X)`. For example, `complex(X).^Y`.
- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- 64-bit integers are not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`mpower` | `nthroot` | `realpow`

Topics

“Array vs. Matrix Operations”
“Operator Precedence”
“MATLAB Operators and Special Characters”

Introduced before R2006a

ppval

Evaluate piecewise polynomial

Syntax

```
v = ppval(pp,xq)
```

Description

`v = ppval(pp,xq)` evaluates the piecewise polynomial `pp` at the query points `xq`.

Examples

Create Piecewise Polynomial with Polynomials of Several Degrees

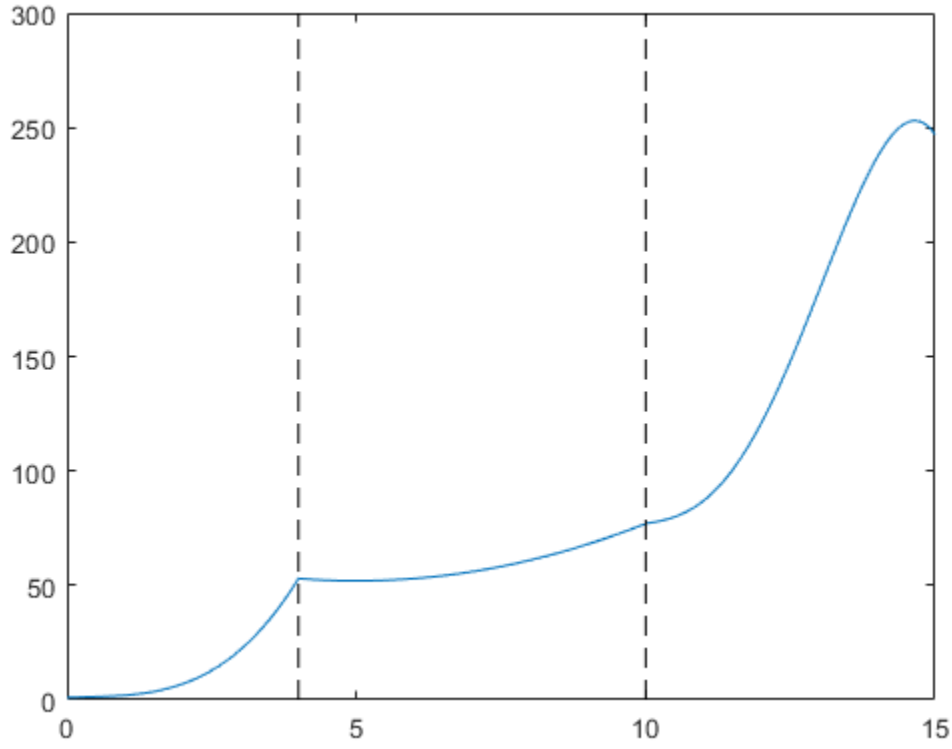
Create a piecewise polynomial that has a cubic polynomial in the interval [0,4], a quadratic polynomial in the interval [4,10], and a quartic polynomial in the interval [10,15].

```
breaks = [0 4 10 15];
coefs = [0 1 -1 1 1; 0 0 1 -2 53; -1 6 1 4 77];
pp = mkpp(breaks,coefs)
```

```
pp = struct with fields:
      form: 'pp'
    breaks: [0 4 10 15]
     coefs: [3x5 double]
    pieces: 3
     order: 5
       dim: 1
```

Evaluate the piecewise polynomial at many points in the interval [0,15] and plot the results. Plot vertical dashed lines at the break points where the polynomials meet.

```
xq = 0:0.01:15;
plot(xq,ppval(pp,xq))
line([4 4],ylim,'LineStyle','--','Color','k')
line([10 10],ylim,'LineStyle','--','Color','k')
```



Create Piecewise Polynomial with Repeated Pieces

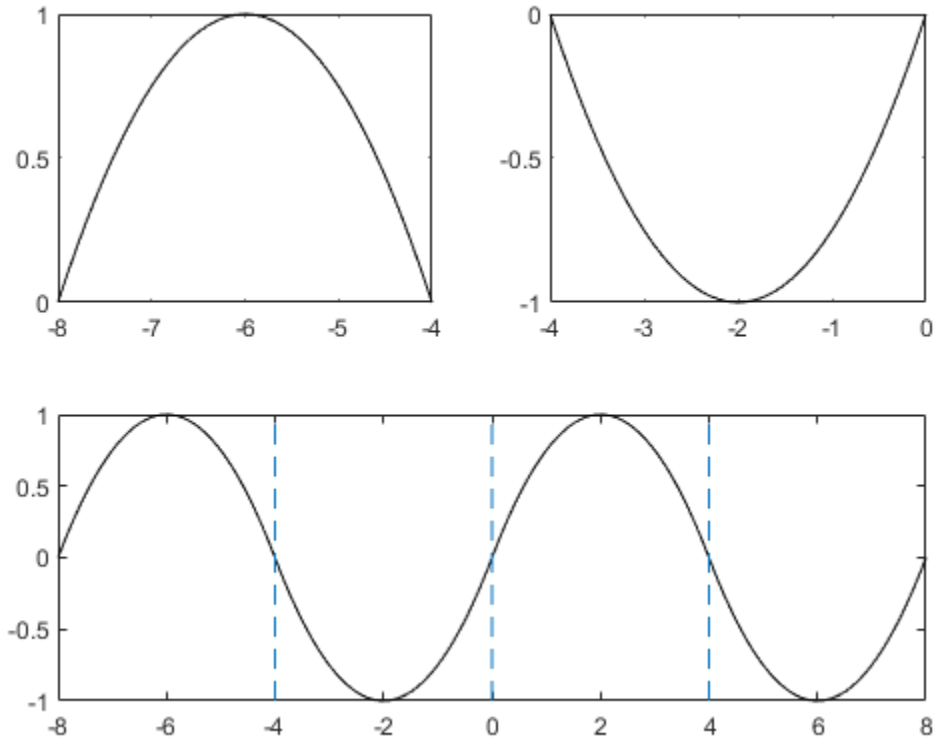
Create and plot a piecewise polynomial with four intervals that alternate between two quadratic polynomials.

The first two subplots show a quadratic polynomial and its negation shifted to the intervals [-8,-4] and [-4,0]. The polynomial is

$$1 - \left(\frac{x}{2} - 1\right)^2 = \frac{-x^2}{4} + x\,.$$

The third subplot shows a piecewise polynomial constructed by alternating these two quadratic pieces over four intervals. Vertical lines are added to show the points where the polynomials meet.

```
subplot(2,2,1)
cc = [-1/4 1 0];
pp1 = mkpp([-8 -4],cc);
xx1 = -8:0.1:-4;
plot(xx1,ppval(pp1,xx1),'k-')

subplot(2,2,2)
pp2 = mkpp([-4 0],-cc);
xx2 = -4:0.1:0;
plot(xx2,ppval(pp2,xx2),'k-')

subplot(2,1,2)
pp = mkpp([-8 -4 0 4 8],[cc;-cc;cc;-cc]);
xx = -8:0.1:8;
plot(xx,ppval(pp,xx),'k-')
hold on
line([-4 -4],ylim,'LineStyle','--')
line([0 0],ylim,'LineStyle','--')
line([4 4],ylim,'LineStyle','--')
hold off
```



Input Arguments

pp — Piecewise polynomial
structure

Piecewise polynomial, specified as a structure. You can create `pp` using `spline`, `pchip`, `makima`, `interp1`, or the spline utility function `mkpp`.

xq — Query points
vector | array

Query points, specified as a vector or array. `xq` specifies the points where `ppval` evaluates the piecewise polynomial.

Data Types: `single` | `double`

Output Arguments

v — Piecewise polynomial values at query points
vector | matrix | array

Piecewise polynomial values at query points, returned as a vector, matrix, or array.

If `pp` has `[d1,..,dr]`-valued coefficients (nonscalar coefficient values), then:

- When `xq` is a vector of length `N`, `v` has size `[d1,...,dr,N]`, and `v(:,...,:,j)` is the value at `xq(j)`.
- When `xq` has size `[N1,...,Ns]`, `v` has size `[d1,...,dr,N1,...,Ns]`, and `v(:,...,:, j1,...,js)` is the value at `xq(j1,...,js)`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The size of output `v` does not match MATLAB when both of the following statements are true:

  - The input `xx` is a variable-size array that is not a variable-length vector.
  - `xx` becomes a row vector at run time.

  In this case, the code generator does not remove the singleton dimensions. However, MATLAB might remove singleton dimensions.

  For example, suppose that `xx` is a :4-by-:5 array (the first dimension is variable size with an upper bound of 4 and the second dimension is variable size with an upper

bound of 5). Suppose that `ppval(pp,0)` returns a 2-by-3 fixed-size array. `v` has size 2-by-3-by-:4-by-:5. At run time, suppose that, size(x,1) =1 and size (x,2) = 5. In the generated code, the size(v) is [2,3,1,5]. In MATLAB, the size is [2,3,5].

See Also

`mkpp` | `pchip` | `spline` | `unmkpp`

Introduced before R2006a

prefdir

Folder containing preferences, settings, history, and layout files

Syntax

```
folder = prefdir
folder = prefdir(1)
```

Description

`folder = prefdir` returns the name of the preferences folder. The preferences folder contains preferences and settings for MATLAB and related products, the command history file, MATLAB favorites files, and MATLAB desktop layout files.

`folder = prefdir(1)` creates a folder for preferences and related files if one does not exist. If the folder does exist, the name is assigned to `folder`.

Examples

View Location of Preferences Folder

```
prefdir
```

```
ans =
    'C:\Users\username\AppData\Roaming\MathWorks\MATLAB\R2018b'
```

View Contents of Preferences Folder

Set the current folder to the preferences folder and then view the contents of the folder.

```
cd(prefdir)
dir
```

Open Preferences Folder in Windows Explorer Tool

```
winopen(prefdir)
```

See Also

preferences

Topics

"Preferences"

Introduced before R2006a

preferences

Open Preferences dialog box

Syntax

```
preferences
```

Description

`preferences` displays the Preferences dialog box. Use the Preferences dialog box to interactively access and set options for MATLAB and related products.

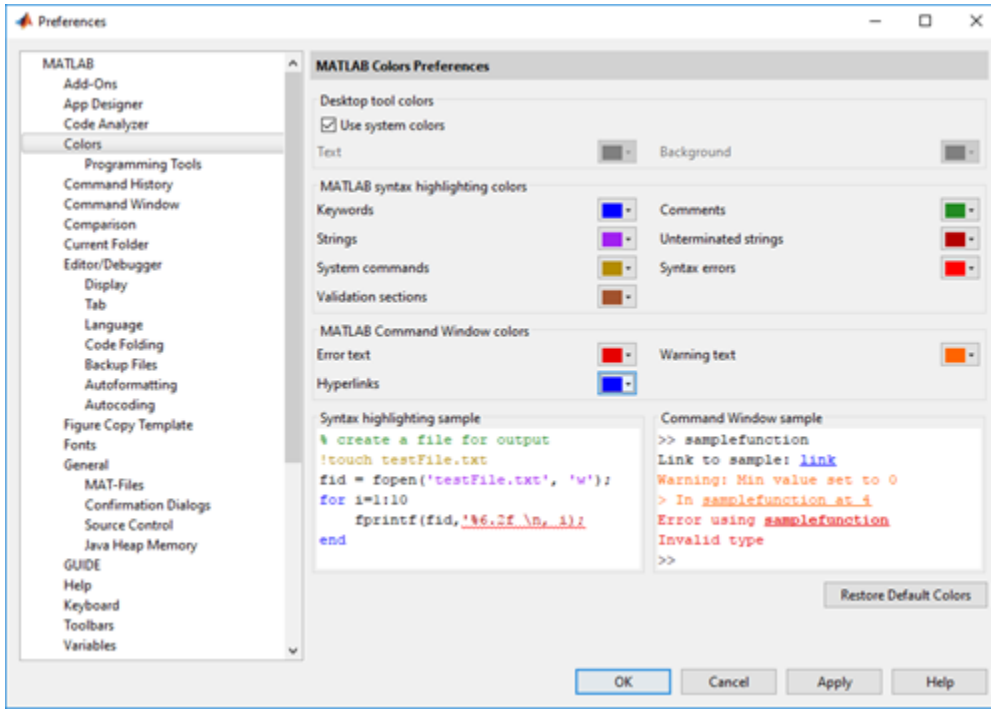
Examples

Change MATLAB Command Window Color Preferences

Open the Preferences dialog box.

```
preferences
```

Select the colors you want to use from the **Error text**, **Warning text**, and **Hyperlinks** color palettes. The new colors display in the **Command Window sample** area.



See Also

`prefdir`

Topics

"Preferences"

Introduced before R2006a

primes

Prime numbers less than or equal to input value

Syntax

```
p = primes(n)
```

Description

`p = primes(n)` returns a row vector containing all the prime numbers less than or equal to `n`. The data type of `p` is the same as that of `n`.

Examples

Primes Less Than or Equal to 25

```
p = primes(25)
```

```
p = 1×9

     2     3     5     7    11    13    17    19    23
```

Primes Less Than or Equal to an Unsigned Integer

```
n = uint16(12);
p = primes(n)
```

```
p = 1x5 uint16 row vector

    2    3    5    7   11
```

Input Arguments

n — Input value
scalar, real integer value

Input value, specified as a scalar that is a real integer value.

Example: `10`

Example: `int16(32)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The maximum double precision input is `2^32`.
- The maximum single precision input is `2^24`.
- The input `n` cannot have type `int64` or `uint64`.

See Also

`factor` | `isprime`

Introduced before R2006a

print

Print figure or save to specific file format

Syntax

```
print(filename,formattype)
print(filename,formattype,formatoptions)
```

```
print
print(printer)
print(driver)
print(printer,driver)
```

```
print('-clipboard',clipboardformat)
```

```
print(resize, ___ )
print(resolution, ___ )
print(renderer, ___ )
print('-noui', ___ )
print(fig, ___ )
```

```
cdata = print('-RGBImage');
```

Description

`print(filename,formattype)` saves the current figure on page 1-11999 to a file using the specified file format, such as `print('BarPlot','-dpng')`. If the file name does not include an extension, then `print` appends the appropriate one.

`print(filename,formattype,formatoptions)` specifies additional options that are available for some formats.

`print` prints the current figure to the default printer.

`print(printer)` specifies the printer. Specify the printer as a character vector or string containing the printer name preceded by `-P`, for example, `'-Pmy printer'`. The printer must be set up on your system.

`print(driver)` specifies the driver. Use this option if you want to ensure that the printed output is either black and white or color.

`print(printer,driver)` specifies the printer and the driver.

`print('-clipboard',clipboardformat)` copies the current figure to the clipboard using the format specified by `clipboardformat`. You can paste the copied figure into other applications.

`print(resize, ___ )` maximizes the figure size to fill the page. Specify `resize` as `'-bestfit'` to preserve the figure's aspect ratio or `'-fillpage'` to ignore the aspect ratio. These options are valid only when saving to a page format (PDF, and PS) or printing to a printer. Use this option with any of the input arguments from the previous syntaxes.

`print(resolution, ___ )` uses the specified resolution. Specify the resolution as a character vector or string containing an integer value preceded by `-r`, for example, `'-r200'`. Use this option with any of the input arguments from the previous syntaxes.

`print(renderer, ___ )` uses the specified renderer. Specify the renderer as either `'-painters'` or `'-opengl'`.

`print('-noui', ___ )` excludes user interface controls, such as push buttons and sliders, from the saved or printed output. It does not exclude user interface objects that can contain an axes, such as a uitab or uipanel.

`print(fig, ___ )` saves or prints the figure or Simulink block diagram specified by `fig`.

`cdata = print('-RGBImage');` returns the RGB image data for the current figure. This option differs from screen captures in that all printing features apply to the output. You can also specify the `resolution`, `renderer`, `'-noui'`, and `fig` options with this syntax. However, you cannot specify a Simulink block diagram.

Examples

Print Paper Copy of Figure

Create a bar chart and print it to your system default printer. If you do not specify the figure to print, then `print` uses the current figure.

```
bar(1:10)
print
```

Copy Figure to Clipboard

Create a plot and copy it to the system clipboard.

```
plot(1:10)
print('-clipboard','-dmeta')
```

You can paste the copied plot into other applications.

Save Figure as Image File

Create a plot and save it as a PNG image file.

```
bar(1:10)
print('BarPlot','-dpng')
```

`print` saves the plot as `BarPlot.png`.

Save Figure as Vector Graphics File

Create a plot and save it as an Encapsulated PostScript file.

```
bar(1:10)
print('BarPlot','-depsc')
```

`print` saves the plot as `BarPlot.eps`.

Add TIFF Preview to EPS File

Save the current figure as an Encapsulated PostScript File and add a TIFF preview.

```
surf(peaks)
print('SurfacePlot','-depsc','-tiff')
```

Specify Figure to Save

Save a specific figure by passing its object variable to `print`.

```
fig = figure;
plot(1:10)
print(fig,'MySavedPlot','-dpng')
```

Alternatively, refer to a figure using the value of its `Number` property, which is the integer value that displays in the figure window title bar. For example, save the figure with `Figure 2` displayed in the title bar. Precede the integer value by `-f`.

```
figure(2);
plot(1:10)
print('-f2','MySavedPlot','-dpng')
```

Save Figure at Screen Size and Resolution

Save a surface plot to a PNG file. Set the `PaperPositionMode` property for the figure to `'auto'` so that it saves at the size displayed on the screen. Use `'-r0'` to save it with screen resolution.

```
surf(peaks)
set(gcf,'PaperPositionMode','auto')
print('PeaksSurface','-dpng','-r0')
```

Save Figure that Fills Page

Save a figure that fills the page using the `'-fillpage'` option.

```
bar([1 10 7 8 2 2 9 3 6])
print('FillPageFigure','-dpdf','-fillpage')
```

Save Figure Without Saving UIControls

Create a figure with a push button that clears the axes. Save the figure to a JPEG file without saving the push button.

```
surf(peaks)
uicontrol('Style','pushbutton','String','Clear',...
```

```
    'Position',[20 20 50 20],'Callback','cla');
print('SurfacePlot','-djpeg','-noui')
```

Return RGB Image Data for Figure

Return the RGB image data for a figure.

```
surf(peaks)
cdata = print('-RGBImage');
```

Display the image data at full resolution using `imshow`.

```
imshow(cdata)
```

Create High Resolution Movie Frame

Create a surface plot. Return the RGB image data for the figure and specify the image resolution. Then, convert the image data to a movie frame, `F`.

```
surf(peaks)
cdata = print('-RGBImage','-r120');
F = im2frame(cdata);
```

Input Arguments

`filename` — File name
character vector | string

File name, specified as a character vector or string containing the desired file name and path.

Example: `'My Saved Chart'`

Example: `'Folder\My Saved Chart'`

Example: `"My Saved Chart"`

The maximum file name length, including the path, is operating system and file format specific. Typically, the file name should be no more than 126 characters, or if you include the path, then no more than 128 characters.

Data Types: `char` | `string`

`formattype` — File format
`'-djpeg'` | `'-dpng'` | `'-dtiff'` | `'-dpdf'` | `'-deps'` | `...`

File format, specified as one of the options in these tables.

Bitmap Image File

Bitmap images contain a pixel-based representation of the figure. The size of the generated file depends on the figure, the format, and your system resolution. Bitmap images are widely used by Web browsers and other applications that display graphics. However, they do not support transparency or scale well and you cannot modify individual graphics objects, such as lines and text, in other graphics applications.

This table lists the supported bitmap image formats.

Option	Bitmap Image Format	Corresponding File Extension
`'-djpeg'`	JPEG 24-bit	`.jpg`
`'-dpng'`	PNG 24-bit	`.png`
`'-dtiff'`	TIFF 24-bit (compressed)	`.tif`
`'-dtiffn'`	TIFF 24-bit (not compressed)	`.tif`
`'-dmeta'`	Enhanced metafile (Windows only)	`.emf`
`'-dbmpmono'`	BMP Monochrome	`.bmp`
`'-dbmp'`	BMP 24-bit	`.bmp`
`'-dbmp16m'`	BMP 24-bit	`.bmp`
`'-dbmp256'`	BMP 8-bit (256 color, uses a fixed colormap)	`.bmp`
`'-dhdf'`	HDF 24-bit	`.hdf`
`'-dpbm'`	PBM (plain format) 1-bit	`.pbm`
`'-dpbmraw'`	PBM (raw format) 1-bit	`.pbm`
`'-dpcxmono'`	PCX 1-bit	`.pcx`

Option	Bitmap Image Format	Corresponding File Extension
`'-dpcx24b'`	PCX 24-bit color (three 8-bit planes)	`.pcx`
`'-dpcx256'`	PCX 8-bit newer color (256 color)	`.pcx`
`'-dpcx16'`	PCX older color (EGA/VGA 16-color)	`.pcx`
`'-dpgm'`	PGM (plain format)	`.pgm`
`'-dpgmraw'`	PGM (raw format)	`.pgm`
`'-dppm'`	PPM (plain format)	`.ppm`
`'-dppmraw'`	PPM (raw format)	`.ppm`

Vector Graphics File

Vector graphics files store commands that redraw the figure. This type of format scales well, but can result in a large file. Also, it might not produce the correct 3-D arrangement of objects in certain cases. Some applications support extensive editing of vector graphics formats. However, some applications do not support editing beyond resizing the graphic. In general, try to make all the necessary changes while your figure is still in MATLAB.

If you set the `Renderer` property for the figure, then `print` uses that renderer when generating output. Otherwise, `print` chooses the appropriate renderer. Typically, `print` uses the Painters renderer when generating vector graphics files. For some complex figures, `print` uses the OpenGL renderer instead. If it uses the OpenGL renderer, then the vector graphics file contains an embedded bitmap image, which might limit the extent to which you can edit the image in other applications. To ensure that `print` uses the Painters renderer, set the `Renderer` property for the figure to `'painters'` or specify `'-painters'` as an input argument to `print`.

If you want output that has transparency, then create a vector graphics file using a Metafile, PDF, or SVG format. If you use a PS or EPS format, then transparency is only supported for the figure and axes backgrounds. Bitmap images do not support transparency, but will closely match what is shown on screen to give the appearance of transparency.

Note The default figure renderer is OpenGL. If the figure renderer differs from the renderer used when generating output, some details of the saved figure can differ from the figure on the display. If necessary, you can make the displayed figure and the saved figure use the same renderer. Set the `Renderer` property for the figure or specify the `renderer` input argument to the `print` function.

This table lists the supported vector graphics formats.

Option	Vector Graphics Format	Corresponding File Extension
`'-dpdf'`	Full page Portable Document Format (PDF) color	`.pdf`
`'-deps'`	Encapsulated PostScript (EPS) Level 3 black and white	`.eps`
`'-depsc'`	Encapsulated PostScript (EPS) Level 3 color	`.eps`
`'-deps2'`	Encapsulated PostScript (EPS) Level 2 black and white	`.eps`
`'-depsc2'`	Encapsulated PostScript (EPS) Level 2 color	`.eps`
`'-dmeta'`	Enhanced Metafile (Windows only)	`.emf`
`'-dsvg'`	SVG (Scalable Vector Graphics)	`.svg`
`'-dps'`	Full-page PostScript (PS) Level 3 black and white	`.ps`
`'-dpsc'`	Full-page PostScript (PS) Level 3 color	`.ps`
`'-dps2'`	Full-page PostScript (PS) Level 2 black and white	`.ps`
`'-dpsc2'`	Full-page PostScript (PS) Level 2 color	`.ps`

You cannot save Simulink block diagrams in PostScript or EPS formats.

Note Only PDF and full-page PS formats use the first two elements of the `PaperPosition` property. Other formats ignore these values.

`formatoptions` — Additional formatting options

`'-tiff'` | `'-loose'` | `'-cmyk'` | `'-append'`

Additional formatting options supported by some file formats, specified as one or more of these values:

- `'-tiff'` — Include a TIFF preview. EPS files only.
- `'-loose'` — Use a loose bounding box. EPS and PS files only.
- `'-cmyk'` — Use CMYK colors instead of RGB colors. EPS and PS files only.
- `'-append'` — Append the figure to an existing PS file. PS files only.

Example: `print('my file','-deps','-tiff','-loose')` saves the current figure to the file `my file.eps` using a loose bounding box and includes a TIFF preview.

`printer` — Printer name

character vector | string

Printer name, specified as a character vector or string containing `-P` and the printer name.

Example: `'-Pmy local printer'`

Example: `"-Pmy local printer"`

To view a list of available printers, use this command:

```
[~,printers] = findprinters
```

If you do not specify a printer, then `print` uses the system default printer. If you want to set up a new printer or select a different default printer, use the operating system printer management utilities. Restart MATLAB if you do not see a printer that is set up already.

Data Types: `char` | `string`

`driver` — Printer driver

`'-dwin'` | `'-dwinc'` | `'-dprn'` | `'-dprnc'`

Printer driver, specified as `'-dwin'`, `'-dwinc'`, `'-dprn'`, or `'-dprnc'`. If you do not specify a driver, then `print` uses the driver returned from the `printopt` function.

The option you use depends on your system, for example:

System	Driver	Output
Windows	`'-dwin'`	Black and white
	`'-dwinc'`	Color
Linux or Mac	`'-dprn'`	Black and white
	`'-dprnc'`	Color

`clipboardformat` — Format copied to clipboard
`-dmeta` | `-dbitmap` | `-dpdf`

Format copied to clipboard, specified as one of these options:

- `'-dmeta'` — Enhanced metafile (Windows only)
- `'-dbitmap'` — Bitmap image (Windows and Mac OS)
- `'-dpdf'` — PDF file (Windows and Mac OS)

`resize` — Option to expand figure to fill page
`'-fillpage'` | `'-bestfit'`

Option to expand figure to fill page, specified as one of these values:

- `'-fillpage'` — Maximize the size of the figure to fill the page. Leave a .25 inch margin on all sides of the page. The tick marks, layout, and aspect ratio of the figure might change.
- `'-bestfit'` — Maximize the size of the figure to fill the page, but preserve the aspect ratio of the figure. The figure might not fill the entire page. This option leaves a minimum page margin of .25 inches.

Both options are valid only when printing a figure to a printer or saving to a page format such as PDF and PS. They are not valid for Simulink block diagrams.

`resolution` — Resolution
character vector | string

Resolution, specified as a character vector or a string containing `-r` and an integer value indicating the resolution in dots per inch. For example, `'-r300'` sets the output resolution to 300 dots per inch. To specify screen resolution, use `'-r0'`.

In general, using a higher resolution value yields higher-quality output, but at the cost of higher memory use and larger output files. The higher the resolution setting, the longer it takes to render your figure.

Specifying the resolution is useful when creating a bitmap image or when using the OpenGL renderer with a vector graphics file format (since OpenGL produces a bitmap image even with vector formats). Specifying the resolution has no effect when using the Painters renderer with a vector graphics file format, since Painters produces a true vector graphics file that contains the commands that redraw the figure.

Note Simulink printing does not support the resolution option. For higher quality output of Simulink models, use a vector format such as SVG or PDF.

Data Types: `char` | `string`

`renderer` — Graphics renderer
`'-opengl'` | `'-painters'`

Graphics renderer, specified as `'-opengl'` or `'-painters'`.

- `'-opengl'` — OpenGL renderer. Use this renderer when saving bitmap images. OpenGL produces a bitmap image even with vector formats, which might limit the extent to which you can edit the image in other applications.
- `'-painters'` — Painters renderer. Use this renderer when saving vector graphics files. If you save to a vector graphics file and if the figure `RendererMode` property is set to `'auto'`, then `print` automatically attempts to use the Painters renderer. If you want to ensure that your output format is a true vector graphics file, then specify the Painters renderer. For example:

  ```
  print('-painters','-deps','myVectorFile')
  ```

Note Sometimes, saving a file with the `'-painters'` option can cause longer rendering times and, in rare cases, might not accurately arrange graphics objects in 3-D views. Also, the Painters renderer cannot print or save lines thinner than one pixel.

If you do not specify the renderer, then `print` automatically uses the appropriate renderer to produce the output format requested. However, if you set the `Renderer` property for the figure, then `print` uses that renderer when generating output.

`fig` — Figure, Simulink block diagram
figure object | Simulink block diagram

Figure object or Simulink block diagram. You can refer to a figure using either its object variable name or using the figure number preceded by `-f`. For example, `-f2` refers to the figure with a `Number` property value of `2`. When specifying a Simulink block diagram, precede the model name with `-s`. Specify the current model using `'-s'`.

You cannot save Simulink block diagrams in PostScript or EPS formats.

Output Arguments

`cdata` — Image data
n-by-m-by-3 array

Image data, returned as an n-by-m-by-3 array. The size of the image data array depends on the `PaperPosition` property of the figure and the output resolution.

Note Starting in R2015b, if you use `print` with the `'-r0'` option on a high-resolution system, then the size of the `cdata` output array is larger than in previous releases or on other systems. Also, the number of elements in `cdata` might not match the size of the figure in pixels based on the figure's `PaperPosition` property and the root's `ScreenPixelsPerInch` property. For more information, see "DPI-Aware Behavior in MATLAB".

Limitations

- Starting MATLAB in no display mode on Linux or using the `-noFigureWindows` startup option on any platform has these limitations for `print`:
  - Printing or saving figures with visible uicontrols errors. To print or save the figure, hide the uicontrols by setting their `Visible` properties to `'off'`, or use the `'-noui'` option with the `print` function.

  - Always uses the painters renderer, even if you specify the `'-opengl'` option.
- In MATLAB Online, `print` only prints to PDF. For additional file format options, save the figure to a file by specifying a `filename`.

More About

Current Figure

The current figure is typically the last figure that you create or click with the mouse. User interaction can change the current figure.

To print a specific figure, specify the figure as the first input argument. If you do not specify a figure, then the `print` function acts on the figure returned by `gcbf`. If `gcbf` returns empty, then `print` acts on the figure returned by `gcf`.

Compatibility Considerations

Starting in R2016a, printed and saved figures match the size of the figure on the screen by default. Previously, printed and saved figures were 8-by-6 inches by default.

Tips

- You can set properties of the figure to control some printing and saving parameters. This table lists properties of the figure related to printing and saving.

Figure Property	Description
`PaperPosition`	Size of the printed or saved figure. If printing to a printer or a full-page output format, then this property also determines the figure location on the page.
`PaperPositionMode`	Specifies whether to use the `PaperPosition` property or the size of the figure on the screen to set the size of the printed or saved figure.

Figure Property	Description
`InvertHardcopy`	Specifies whether to use the current background color of the figure or to change the background color to white when printing or saving the figure.
`PaperOrientation`	Figure orientation on printed page.
`PaperType`	Standard printer paper size.
`PaperSize`	Custom width and height of printer paper.
`PaperUnits`	Units for the `PaperSize` and `PaperPosition` properties.

- If you are using a Linux or Mac system and get an error about an invalid or unrecognized printer, then try specifying a PostScript driver for the `driver` option. If the printer supports PostScript level 3, try `'-dps'` for black and white or `'-dpsc'` for color. If the printer supports PostScript level 2, then try `'-dps2'` for black and white or `'-dps2c'` for color. For example:

  ```
  print('-dpsc','-Pmy printer')
  ```

  To use this driver as the default driver, edit the `printopt` file. Alternatively, print the figure to a PDF file. Then, print the PDF from an external PDF viewer.
- If you print a figure that has a callback defined for the `SizeChangedFcn` property and if the output size differs from the size of the figure on the screen, then the `print` function displays a warning message. To avoid the warning message, set the `PaperPositionMode` property for the figure to `'auto'`.

See Also

`getframe` | `saveas` | `savefig` | `sprintf`

Introduced before R2006a

printopt

Configure printer defaults

Syntax

```
[pcmd,dev] = printopt
```

Description

`[pcmd,dev] = printopt` returns the current system-dependent printing command and output device. `printopt` is a file used by `print` to produce the printed output. You can edit the file `printopt.m` to set your default printer type and destination.

- `pcmd` contains the command that `print` uses to send a file to the printer when not using the printer drivers directly.
- `dev` contains the printer driver or graphics format option for the `print` command.

The defaults for `pcmd` and `dev` are platform-dependent. This table lists the default values for each platform.

Platform	Print Command	Driver or Format
Mac and Linux	`lpr -r`	`-dprn` In R2014a and earlier, the driver was `-dps2`.
Windows	COPY /B %s LPT1:	`-dwin`

See Also

`print` | `printdlg`

Introduced before R2006a

printdlg

Open figure Print dialog box

Note

Use this function only with GUIDE, or with apps created using the `figure` function.

Note The `-crossplatform` and `-setup` input arguments have been removed in R2014b. They no longer have any effect.

Syntax

```
printdlg
printdlg(fig)
```

Description

`printdlg` prints the current figure.

`printdlg(fig)` creates a modal dialog box from which you can print the figure window identified by the handle `fig`. Uimenus do not print.

Tips

If you want to set up a new printer, use the operating system printer management utilities. Restart MATLAB if you do not see the printer which is already setup.

See Also

`print` | `printopt` | `printpreview`

Introduced before R2006a

printpreview

Open figure Print Preview dialog box

Note

Use this function only with GUIDE, or with apps created using the `figure` function.

Syntax

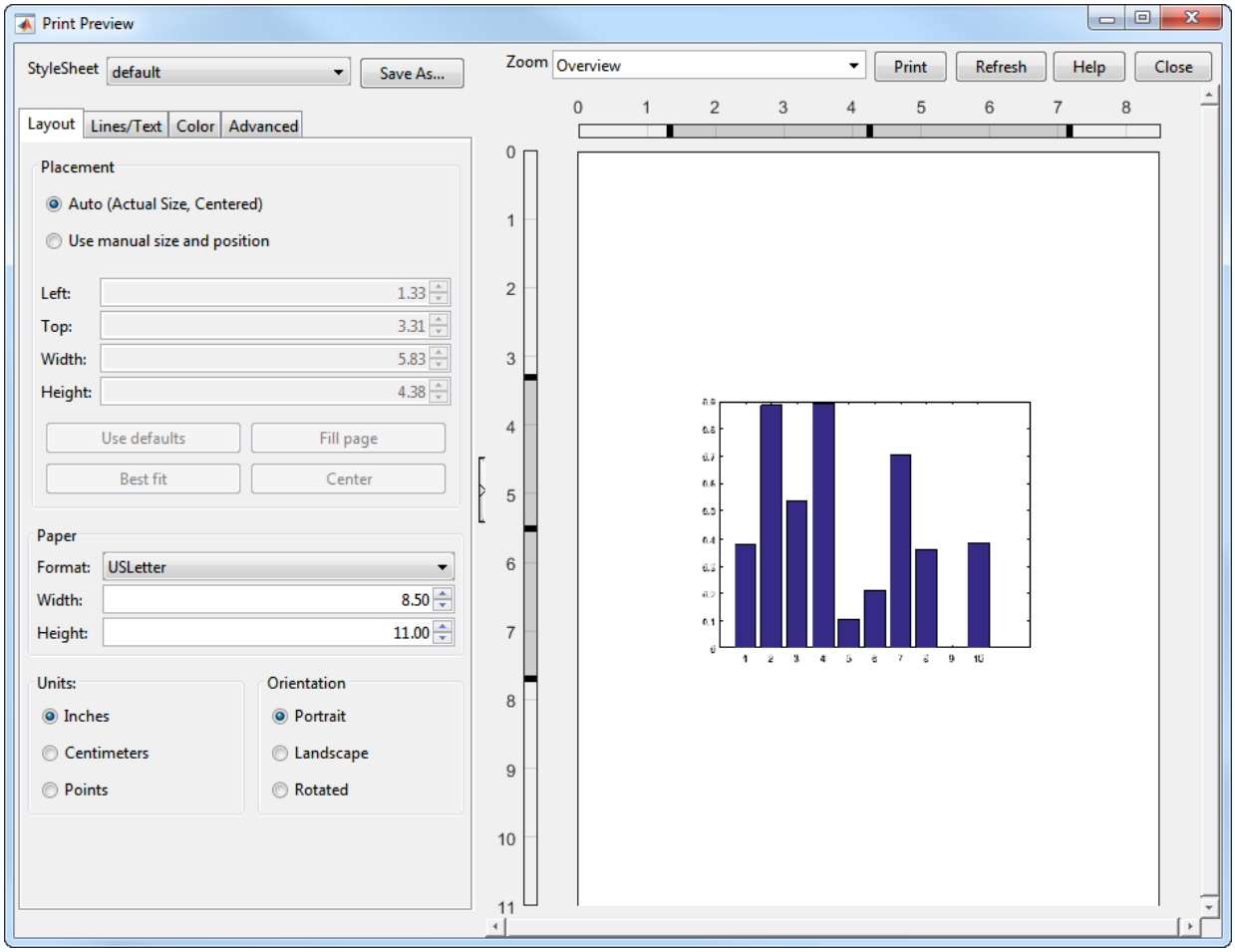
```
printpreview
printpreview(f)
```

Description

`printpreview` displays a dialog box showing the figure in the currently active figure window as it will print. A scaled version of the figure displays in the right-hand pane of the dialog box.

`printpreview(f)` displays a dialog box showing the figure having the handle `f` as it will print.

Use the Print Preview dialog box, shown below, to control the layout and appearance of figures before sending them to a printer or print file. Controls are grouped into four tabbed panes: **Layout**, **Lines/Text**, **Color**, and **Advanced**.



Right Pane Controls

You can position and scale plots on the printed page using the rulers in the right-hand pane of the Print Preview dialog. Use the outer ruler handlebars to change margins. Moving them changes plot proportions. Use the center ruler handlebars to change the position of the plot on the page. Plot proportions do not change, but you can move portions of the plot off the paper. The buttons on that pane let you refresh the plot, close the dialog (preserving all current settings), print the page immediately, or obtain context-

sensitive help. Use the **Zoom** box and scroll bars to view and position page elements more precisely.

The Layout Tab

Use the **Layout** tab, shown above, to control the paper format and placement of the plot on printed pages. The following table summarizes the **Layout** options:

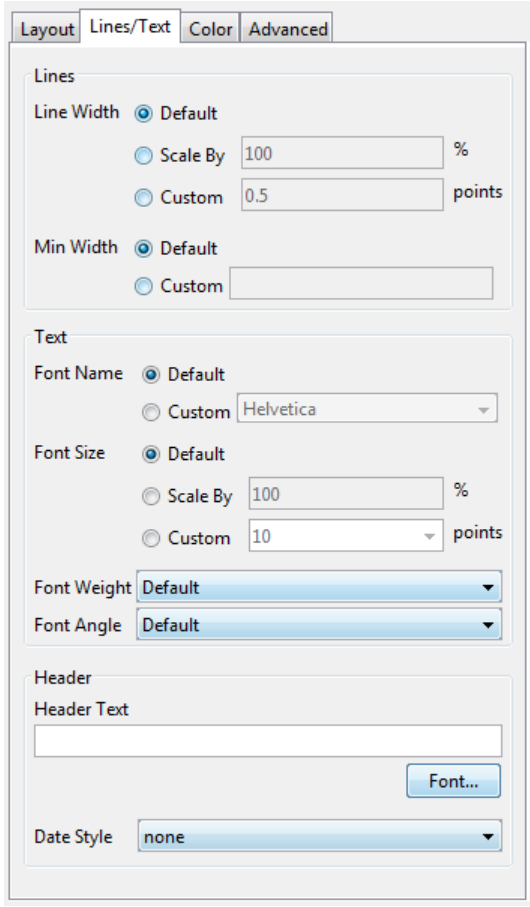
Group	Option	Description
Placement	**Auto**	Let MATLAB decide placement of plot on page*
	Use manual...	Specify position parameters for plot on page*
	Top, Left, Width, Height	Standard position parameters in current units
	Use defaults	Revert to default position
	Fill page	Expand figure to fill printable area
	Best fit	Expand figure to fill printable area, center the figure, and preserve the figure's aspect ratio
	Center	Center plot on printed page
Paper	**Format**	U.S. and ISO sheet size selector
	Width, Height	Sheet size in current units
Units	**Inches**	Use inches as units for dimensions and positions
	Centimeters	Use centimeters as units for dimensions and positions
	Points	Use points as units for dimensions and positions
Orientation	**Portrait**	Upright paper orientation
	Landscape	Sideways paper orientation
	Rotated	Currently the same as **Landscape**

* Selecting **Auto** in the Placement group sets the figure `PaperPositionMode` to `'auto'` and disables the controls in that panel. Selecting **Use manual size and position** sets the figure `PaperPositionMode` to `'manual'` and enables the controls. If you set

`PaperPositionMode` programmatically, the print preview Placement controls respond accordingly.

The Lines/Text Tab

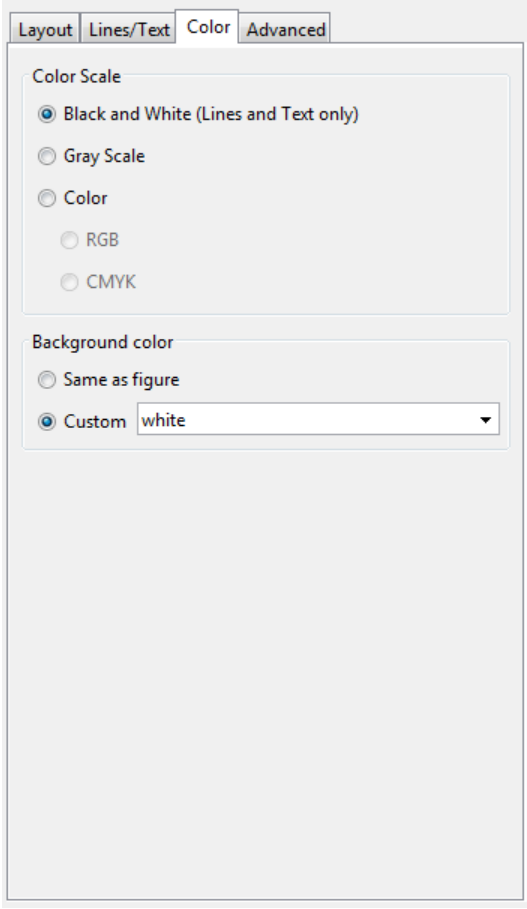
Use the **Lines/Text** tab, shown below, to control the line weights, font characteristics, and headers for printed pages. The following table summarizes the **Lines/Text** options:



Group	Option	Description
Lines	**Line Width**	Scale all lines by a percentage from 0 upward (100 being no change), print lines at a specified point size, or default line widths used on the plot
	Min Width	Smallest line width (in points) to use when printing; defaults to 0.5 point
Text	**Font Name**	Select a system font for all text on plot, or default to fonts currently used on the plot
	Font Size	Scale all text by a percentage from 0 upward (100 being no change), print text at a specified point size, or default to this used on the plot
	Font Weight	Select `Normal` ... `Bold` font styling for all text from drop-down menu or default to the font weights used on the plot
	Font Angle	Select `Normal`, `Italic` or `Oblique` font styling for all text from drop-down menu or default to the font angles used on the plot
Header	**Header Text**	Type the text to appear on the header at the upper left of printed pages, or leave blank for no header. Header text only supports plain text characters.
	Date Style	Select a date format to have today's date appear on each printed page, or `none` for no date

The Color Tab

Use the **Color** tab, shown below, to control how colors are printed for lines and backgrounds. The following table summarizes the **Color** options:

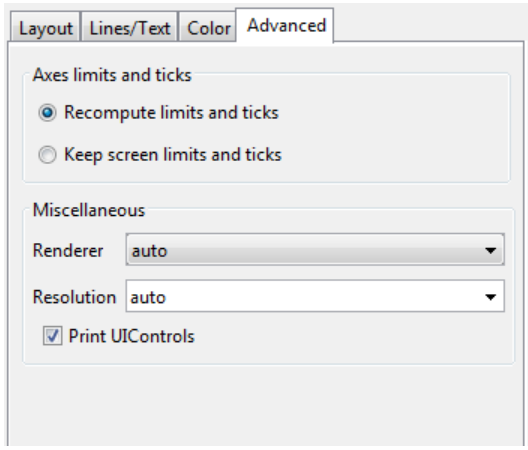


Group	Option	Description
Color Scale	**Black and White**	Select to print lines and text in black and white, but use color for patches and other objects
	Gray Scale	Convert colors to shades of gray on printed pages
	Color	Print everything in color, matching colors on plot; select `RGB` (default) or `CMYK` color model for printing

Group	Option	Description
Background Color	**Same as figure**	Print the figure's background color as it is
	Custom	Select a color name, or type a colorspec for the background; `white` (default) implies no background color, even on colored paper.

The Advanced Tab

Use the **Advanced** tab, shown below, to control finer details of printing, such as limits and ticks, renderer, resolution, and the printing of UIControls. The following table summarizes the **Advanced** options:



Group	Option	Description
Axes limits and ticks	**Recompute limits and ticks**	Redraw *x*- and *y*-axes ticks and limits based on printed plot size (default)
	Keep screen limits and ticks	Use the *x*- and *y*-axes ticks and limits shown on the plot when printing the previewed figure
Miscellaneous	**Renderer**	Select a rendering algorithm for printing: `painters`, `opengl`, or `auto` (default)

Group	Option	Description
	Resolution	Select resolution to print at in dots per inch: `150`, `300`, `600`, or `auto` (default), or type in any other positive value
	Print UIControls	Print all visible UIControls in the figure (default), or uncheck to exclude them from being printed

Alternatives

Use **File** > **Print Preview** on the figure window menu to access the Print Preview dialog box, described below. For details, see “Print Figure from File Menu”.

See Also

`print`

Topics

“Print Figure from File Menu”

Introduced before R2006a

prism

Prism colormap array

Syntax

```
c = prism
c = prism(m)
```

Description

`c = prism` returns the prism colormap as a three-column array with the same number of rows as the colormap for the current figure. If no figure exists, then the number of rows is equal to the default length of 256. Each row in the array contains the red, green, and blue intensities for a specific color. The intensities are in the range [0,1], and the color scheme is a repeated sequence of the rainbow colors.

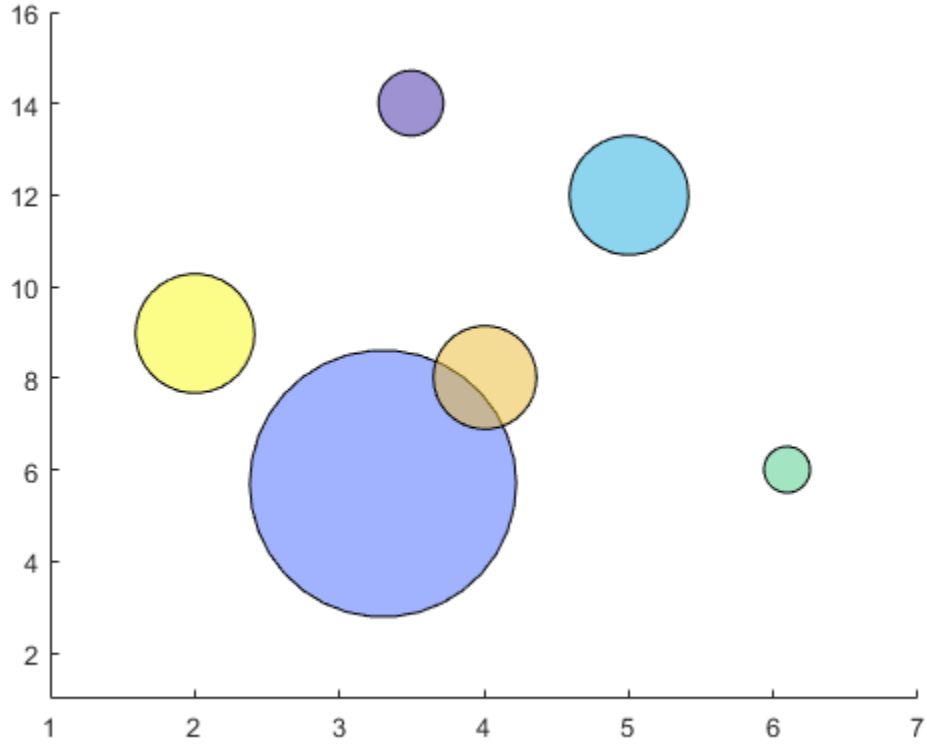
`c = prism(m)` returns the colormap with `m` colors.

Examples

Downsample the Prism Colormap

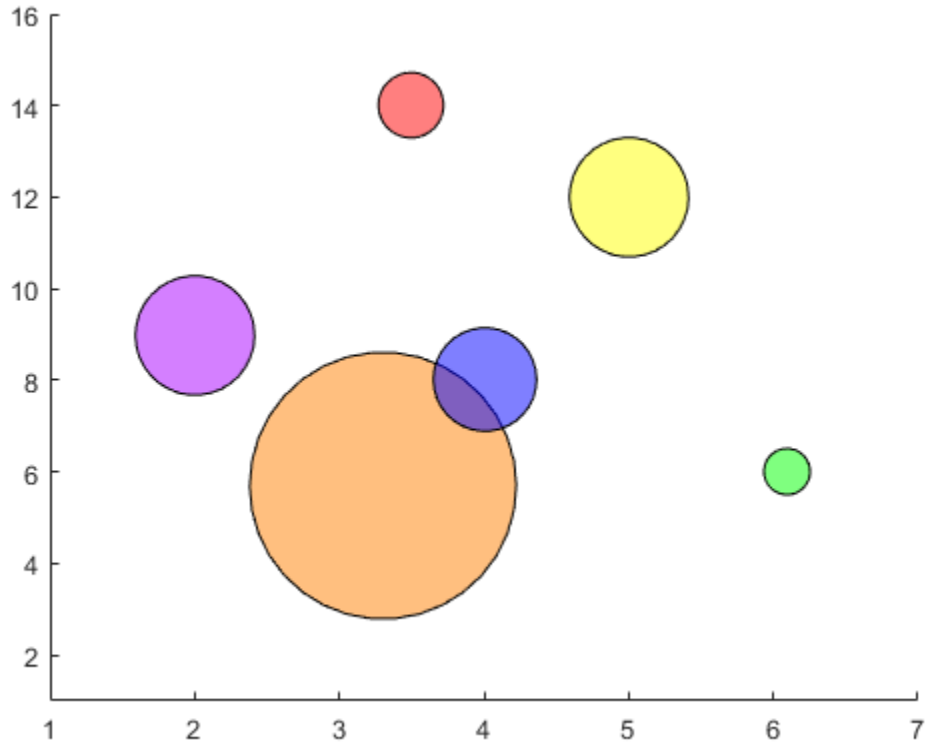
Create a scatter plot with the default colors.

```
x = [3.5 3.3 5 6.1 4 2];
y = [14 5.7 12 6 8 9];
sz = 100*[6 100 20 3 15 20];
c = [1 2 3 4 5 6];
scatter(x,y,sz,c,'filled','MarkerEdgeColor','k','MarkerFaceAlpha',.5);
xlim([1 7]);
ylim([1 16]);
```



Get the prism colormap array with six entries. Then replace the colormap in the scatter plot.

```
c = prism(6);
colormap(c);
```



Input Arguments

m — Number of colors

256 (default) | scalar integer value

Number of colors, specified as a scalar integer value. The default value of `m` is equal to the length of the colormap for the current figure. If no figure exists, the default value is 256.

Data Types: `single` | `double`

Compatibility Considerations

Colormaps Have 256 Colors by Default

Behavior changed in R2019b

Starting in R2019b, colormaps have 256 colors by default.

In R2019a and previous releases, the default size is 64. If you have code that depends on a colormap having 64 colors, specify the number of colors when you set the colormap for the figure, axes, or chart. For example, `colormap(prism(64))` sets the figure's colormap to the 64-color `prism` colormap.

Alternatively, you can change the default colormap for all figures within your MATLAB session:

```
set(groot,'defaultFigureColormap',prism(64))
```

See Also

`colormap`

Topics

"Change Color Scheme Using a Colormap"

Introduced before R2006a

prod

Product of array elements

Syntax

```
B = prod(A)
B = prod(A,'all')
B = prod(A,dim)
B = prod(A,vecdim)
B = prod( ___ ,outtype)
B = prod( ___ ,nanflag)
```

Description

`B = prod(A)` returns the product of the array elements of `A`.

- If `A` is a vector, then `prod(A)` returns the product of the elements.
- If `A` is a nonempty matrix, then `prod(A)` treats the columns of `A` as vectors and returns a row vector of the products of each column.
- If `A` is an empty 0-by-0 matrix, `prod(A)` returns `1`.
- If `A` is a multidimensional array, then `prod(A)` acts along the first nonsingleton dimension on page 1-12024 and returns an array of products. The size of this dimension reduces to `1` while the sizes of all other dimensions remain the same.

`prod` computes and returns `B` as `single` when the input, `A`, is `single`. For all other numeric and logical data types, `prod` computes and returns `B` as `double`.

`B = prod(A,'all')` computes the product of all elements of `A`. This syntax is valid for MATLAB versions R2018b and later.

`B = prod(A,dim)` returns the products along dimension `dim`. For example, if `A` is a matrix, `prod(A,2)` is a column vector containing the products of each row.

`B = prod(A,vecdim)` computes the product based on the dimensions specified in the vector `vecdim`. For example, if `A` is a matrix, then `prod(A,[1 2])` is the product of all

elements in A, since every element of a matrix is contained in the array slice defined by dimensions 1 and 2.

`B = prod( ___ ,outtype)` returns an array in the class specified by `outtype`, using any of the input arguments in the previous syntaxes. `outtype` can be `'double'`, `'native'`, or `'default'`.

`B = prod( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `prod(A,'includenan')` includes `NaN` values in the calculation while `prod(A,'omitnan')` ignores them.

Examples

Product of Elements in Each Column

Create a 3-by-3 array whose elements correspond to their linear indices.

```
A=[1:3:7;2:3:8;3:3:9]
```

```
A = 3×3

     1     4     7
     2     5     8
     3     6     9
```

Find the product of the elements in each column. The length of the first dimension is 1, and the length of the second dimension matches `size(A,2)`.

```
B = prod(A)
```

```
B = 1×3

     6   120   504
```

Product of Logical Input

Create an array of logical values.

```
A = [true false; true true]
```

```
A = 2x2 logical array

   1   0
   1   1
```

Find the product of the elements in each column.

```
B = prod(A)
```

```
B = 1×2

     1     0
```

The output has type `double`.

```
class(B)
```

```
ans =
'double'
```

Product of Elements in Each Row

Create a 3-by-3 array whose elements correspond to their linear indices.

```
A=[1:3:7;2:3:8;3:3:9]
```

```
A = 3×3

     1     4     7
     2     5     8
     3     6     9
```

Find the product of the elements in each row and reduce the length of the second dimension to 1. The length of the first dimension matches `size(A,1)`, and the length of the second dimension is 1.

```
dim = 2;
B = prod(A,dim)
```

```
B = 3×1

    28
    80
   162
```

Product of Array Page

Create a 3-D array and compute the product over each page of data (rows and columns).

```
A(:,:,1) = [2 4; -2 1];
A(:,:,2) = [1 2; -5 3];
A(:,:,3) = [4 4; 1 -3];
B1 = prod(A,[1 2])
```

```
B1 =
B1(:,:,1) =

   -16


B1(:,:,2) =

   -30


B1(:,:,3) =

   -48
```

To compute the product over all dimensions of an array, you can either specify each dimension in the vector dimension argument, or use the `'all'` option.

```
B2 = prod(A,[1 2 3])
```

```
B2 = -23040
```

```
Ball = prod(A,'all')
```

```
Ball = -23040
```

Single-Precision Input Treated as Double

Create a 3-by-3 array of single-precision values.

```
A = single([1200 1500 1800; 1300 1600 1900; 1400 1700 2000])
```

```
A = 3x3 single matrix

        1200        1500        1800
        1300        1600        1900
        1400        1700        2000
```

Find the product of the elements in each row by multiplying in double precision.

```
B = prod(A,2,'double')
```

```
B = 3×1
10^9 ×

    3.2400
    3.9520
    4.7600
```

The output is double precision.

```
class(B)
```

```
ans =
'double'
```

Integer Data Type for Input and Output

Create a 3-by-3 array of 8-bit unsigned integers.

```
A = uint8([1:3:7;2:3:8;3:3:9])
```

```
A = 3x3 uint8 matrix

   1   4   7
```

```
   2   5   8
   3   6   9
```

Find the product of the elements in each column natively in `uint8`.

```
B = prod(A,'native')
```

```
B = 1x3 uint8 row vector

     6   120   255
```

The result is an array of 8-bit unsigned integers.

```
class(B)
```

```
ans =
'uint8'
```

Product Excluding NaN

Create a vector and compute its product, excluding `NaN` values. If you do not specify `'omitnan'`, then `prod(A)` returns `NaN`.

```
A = [1 3 2 4 NaN 3 NaN 2];
P = prod(A,'omitnan')
```

```
P = 144
```

Input Arguments

A — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

`dim` — Dimension to operate along

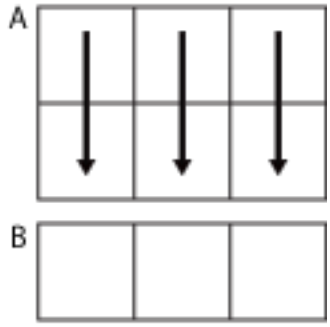
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal `1`.

Dimension `dim` indicates the dimension whose length reduces to `1`. The `size(B,dim)` is `1`, while the sizes of all other dimensions remain the same.

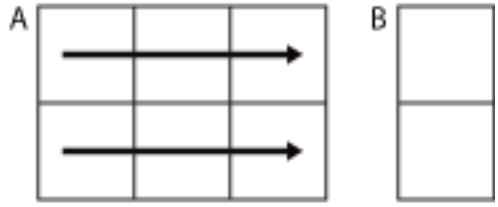
Consider a two-dimensional input array, `A`.

- If `dim = 1`, then `prod(A,1)` returns a row vector containing the product of the elements in each column.



prod(A,1)

- If `dim = 2`, then `prod(A,2)` returns a column vector containing the product of the elements in each row.



prod(A,2)

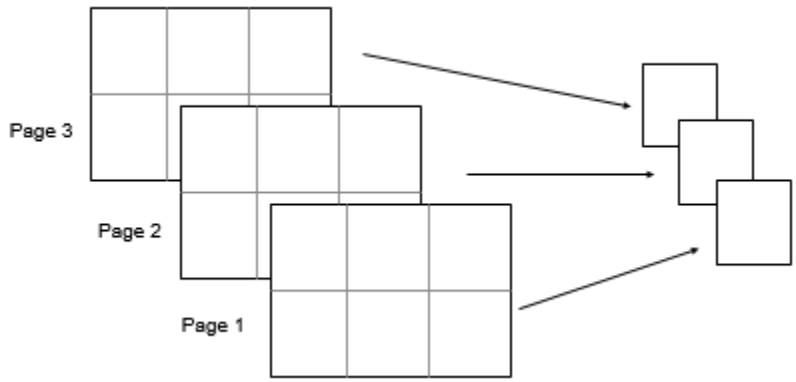
`prod` returns `A` when `dim` is greater than `ndims(A)`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`vecdim` — Vector of dimensions
vector of positive integers

Vector of dimensions, specified as a vector of positive integers. Each element represents a dimension of the input array. The lengths of the output in the specified operating dimensions are 1, while the others remain the same.

Consider a 2-by-3-by-3 input array, `A`. Then `prod(A,[1 2])` returns a 1-by-1-by-3 array whose elements are the products of each page of `A`.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`outtype` — Output class
`'default'` (default) | `'double'` | `'native'`

Output class, specified as `'default'`, `'double'`, or `'native'`, and which defines the data type of the output, `B`.

`outtype`	Output data type
`'default'`	`double`, unless the input data type is `single`. In which case, the output data type is `single`.
`'double'`	`double`

outtype	Output data type
'native'	same data type as the input array, A

nanflag — NaN condition
'includenan' (default) | 'omitnan'

NaN condition, specified as one of these values:

- 'includenan' — Include NaN values from the input when computing the product, resulting in NaN output.
- 'omitnan' — Ignore NaN values in the input. If all elements are NaN, then prod returns 1.

Output Arguments

B — Product array
scalar | vector | matrix | multidimensional array

Product array, returned as a scalar, vector, matrix, or multidimensional array.

The class of B is as follows:

- If the outtype argument specifies 'default' or is not used
  - and the input is not single, then the output is double.
  - and the input is single, then the output is single.
- If the outtype argument specifies 'double', then the output is double regardless of the input data type.
- If the outtype argument specifies 'native', then the output is the same data type as the input.

More About

First Nonsingleton Dimension

The first nonsingleton dimension is the first dimension of an array whose size is not equal to 1.

For example:

- If X is a 1-by-n row vector, then the second dimension is the first nonsingleton dimension of X.
- If X is a 1-by-0-by-n empty array, then the second dimension is the first nonsingleton dimension of X.
- If X is a 1-by-1-by-3 array, then the third dimension is the first nonsingleton dimension of X.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If you supply `dim`, it must be a constant.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- If you supply `dim`, it must be a constant.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- 64-bit integers are not supported with the `'native'` option.
- The order of the products in `prod` operation is not defined. Therefore, the `prod` operation on a GPU array might not return exactly the same answer as the `prod` operation on the corresponding MATLAB numeric array.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- The order of the products in `prod` operation is not defined. Therefore, the `prod` operation on a distributed array might not return exactly the same answer as the `prod` operation on the corresponding MATLAB numeric array. The difference might be significant when `A` is a signed integer type and its product is accumulated natively.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cumprod` | `diff` | `ndims` | `sum`

Introduced before R2006a

profile

Profile execution time for functions

Syntax

```
profile action
profile action option1 ... optionN
profile option1 ... optionN

p = profile('info')

s = profile('status')
```

Description

Use the Profiler to track execution time. Knowing the execution time of your MATLAB code helps you to debug and optimize it. For information on the user interface to the Profiler, see “Profile to Improve Performance”.

`profile action` profiles the execution time for functions. Use `action` to start, stop, and restart the Profiler, and view or clear profile statistics. For example, `profile on` starts the Profiler.

`profile action option1 ... optionN` starts or restarts the Profiler with the specified options. For example, `profile resume -history` restarts the Profiler and records the sequence of function calls.

`profile option1 ... optionN` sets the specified Profiler options. If the Profiler is on and you specify one of the options, MATLAB throws an error. To change options, first specify `profile off`, and then specify the new options.

`p = profile('info')` stops the Profiler and displays a structure containing the results. To access the data generated by `profile`, use this syntax.

`s = profile('status')` returns a structure with the Profiler status information.

Examples

Profile Function and Save Results as HTML

This example is not supported in MATLAB Online.

Turn on the Profiler, and call the `magic` function.

```
profile on
n = 100;
M = magic(n);
```

View the results in the Profiler window.

```
profile viewer
```

Save the results as HTML. By default, `profsave` saves the files to the `profile_results` subfolder in your current working folder.

```
profsave
```

Profile Function and Display Function Call History

Create the file `myFunction.m` using this main function and local function.

```
function c = myFunction(a,b)
c = sqrt(square(a)+square(b));
end

function y = square(x)
y = x.^2;
end
```

Turn on the Profiler, and enable the function call history option. Profile a call to the `myFunction` function.

```
profile on -history
a = rand(5);
b = rand(5);
c = myFunction(a,b);
```

Save the profile results.

```
p = profile('info')
```

```
p =

      FunctionTable: [2x1 struct]
    FunctionHistory: [2x6 double]
     ClockPrecision: 3.3475e-07
         ClockSpeed: 3.0600e+09
               Name: 'MATLAB'
           Overhead: 0
```

Display the function call history.

```
p.FunctionHistory
```

```
ans =

     0     0     1     0     1     1
     1     2     2     2     2     1
```

Display function entry and exit information by iterating over the function call history.

```
numEvents = size(p.FunctionHistory,2);
for n = 1:numEvents
    name = p.FunctionTable(p.FunctionHistory(2,n)).FunctionName;

    if p.FunctionHistory(1,n) == 0
        disp(['Entered ' name]);
    else
        disp(['Exited ' name]);
    end
end
```

```
Entered myFunction
Entered myFunction>square
Exited myFunction>square
Entered myFunction>square
Exited myFunction>square
Exited myFunction
```

Set the function call history to the default value.

```
profile -nohistory
```

Profile and Save Results as MAT-File

Turn on the Profiler, and call the `magic` function.

```
profile on
n = 100;
M = magic(n);
```

Save the results to a MAT-file

```
p = profile('info')
save myprofiledata p
```

```
p =

      FunctionTable: [1x1 struct]
    FunctionHistory: [2x2 double]
     ClockPrecision: 3.3475e-07
         ClockSpeed: 3.0600e+09
               Name: 'MATLAB'
           Overhead: 0
```

View Current Profiler Settings

```
s = profile('status')
```

```
s =

     ProfilerStatus: 'off'
        DetailLevel: 'mmex'
              Timer: 'performance'
    HistoryTracking: 'off'
        HistorySize: 1000000
```

Input Arguments

`action` — Control options for Profiler
`on` | `off` | `resume` | `clear` | `viewer` | `info` | `status`

Control options for the Profiler specified as one of these options.

Option	Result
`on`	Start the Profiler, clearing any previously recorded profile statistics.
`off`	Stop the Profiler.
`resume`	Restart the Profiler without clearing previously recorded statistics.
`clear`	Clear the recorded statistics.
`viewer`	Stop the Profiler and display the results in the Profiler window. For more information, see “Profile to Improve Performance”. The profiler user interface is not available in MATLAB Online.
`info`	Stop the Profiler and return a structure containing the results.
`status`	Return a structure with the Profiler status information.

`option1 ... optionN` — One or more profiling options
character vectors

One or more profiling options, specified as character vectors corresponding to valid settings from the history and clock option tables. You can specify one or more of these options with one of the following syntaxes:

- `profile on option1 ... optionN`
- `profile resume option1 ... optionN`
- `profile option1 ... optionN`

If you change the Profiler settings, the settings persist when you stop the Profiler or clear the statistics. To revert to default Profiler behavior, manually set the options to the default settings or start a new MATLAB session.

History Options

Option	Result
`-history`	Record the exact sequence of function calls, including function entry and exit events. By default, the `-history` option is not enabled.

Option	Result
`-historysize` *`integer`*	Specify the number of function entry and exit events to record. By default, `historysize` is 1,000,000. If the number of function calls exceeds the specified `historysize`, the `profile` function continues to record profiling statistics other than the sequence of calls.
`-nohistory`	Default setting. Disable recording of the exact sequence of function calls. The `profile` function collects all other profiling statistics.

Clock Options

Option	Result
`-timer 'performance'`	Default setting. Use wall-clock time from the clock that the operating system supplies to measure performance.
`-timer 'processor'`	Use the wall-clock time directly from the processor. Sometimes your power savings settings or use of multiple processors influence this measurement.
`-timer 'real'`	Use system time reported by the operating system. This option is the most computationally expensive measurement and has the most impact on the performance of profiled code. Changing the time on the operating system clock influences this measurement.
`-timer 'cpu'`	Use computer time and sums time across all threads. This measurement is different from wall-clock time. For example, the computer time for the `pause` function is typically small, but wall-clock time accounts for the actual time paused, which is larger.

Output Arguments

p — Profiler statistics
structure

Profiler statistics, returned as a structure containing these fields.

Field	Description

`FunctionTable`	Function statistics, returned as a structure array. Each structure in the array contains information about one of the functions or local functions called during profiling. Each structure contains the following fields:
	• `CompleteName` – Full path to `FunctionName`.
	• `FunctionName` – Name of function. If the function is a local function, `FunctionName` includes the main function.
	• `FileName` – Full path to `FunctionName`, with the file extension. If the function is a local function, `FileName` is the full path to the main function.
	• `Type` – Type of function. For example, MATLAB function, MEX-function, local function, or nested function.
	• `NumCalls` – Number of times the profiled code called the function.
	• `TotalTime` – Total time spent in the function and its child functions.
	• `TotalRecursiveTime` – MATLAB no longer uses this field.
	• `Children` – Information about functions that the function called. Each entry in the array contains information about one child function. The structure contains these fields:
	  • `Index` – Index to child function information structure within `FunctionTable`.
	  • `NumCalls` – Number of times the profiled code called the child function.
	  • `TotalTime` – Total time spent in the child function.
	• `Parents` – Information about the parent functions of `FunctionName`. Each structure in the array contains information about one of the parents. The structure contains these fields:
	  • `Index` – Index to parent function information structure within `FunctionTable`.
	  • `NumCalls` – Number of times the parent function called this function.
	• `ExecutedLines` – Array containing line-by-line details for the profiled function.

Field	Description
	• Column 1 – Line number for the executed line of code in `FileName`. • Column 2 – Number of times the profiled code executed the line of code. • Column 3 – Total time spent on the line of code. The sum of Column 3 entries does not necessarily add up to the `TotalTime`. • `IsRecursive` – Indicator of whether the function is recursive. If the value is `1` (`true`), the function is recursive. If the value is `0` (`false`), the function is nonrecursive. • `PartialData` – Indicator of whether the profile statistics are incomplete. If the value is logical `1` (`true`), the function was modified during profiling. For example, if you edited the function or cleared it from memory. In that event, the Profiler collects data only up until you modified the function.
`FunctionHistory`	Function call history, returned as an array. • Row 1 – Indicator of function entry or exit. The Profiler records function entry with a `0`, and function exit with a `1`. • Row 2 – Index to function information structure within `FunctionTable`.
`ClockPrecision`	Precision of the time measurement of the `profile` function, returned as a `double`.
`ClockSpeed`	Estimated CPU clock speed, returned as a `double`.
`Name`	Name of the profiler, returned as a character array.
`Overhead`	Reserved for future use.

s — Profiler status
structure

Profiler status, returned as a structure containing these fields.

Field	Values	Default Value
`ProfilerStatus`	`'on'` or `'off'`	`'off'`
`DetailLevel`	`'mmex'`	`'mmex'`

Field	Values	Default Value
`Timer`	`'performance'`, `'processor'`, `'cpu'`, or `'real'`	`'performance'`
`HistoryTracking`	`'on'` or `'off'`	`'off'`
`HistorySize`	integer	1000000

Limitations

- If the profiled code uses indirect (or mutual) recursion, the MATLAB Profiler might return inaccurate results. If the recursion is direct (a single function calling itself), then the Profiler returns the total time for the non-recursive calls to the function. To determine if a function in the profiled code is recursive (directly or indirectly), examine the value of the `IsRecursive` field in the `FunctionTable` entry.

Tips

- To open the Profiler user interface, use the `profile viewer` syntax or see “Profile to Improve Performance”. The profiler user interface is not available in MATLAB Online.
- As of MATLAB R2015b, the default timer is `'performance'`. In previous versions of MATLAB, the default profiler timer was `'cpu'`, which measures compute time instead of wall-clock time.

See Also

`checkcode` | `matlab.codetools.requiredFilesAndProducts` | `profsave`

Topics

“Profile to Improve Performance”
“Profiling Parallel Code” (Parallel Computing Toolbox)

Introduced before R2006a

profsave

Save profile report in HTML format

Syntax

```
profsave
profsave(profinfo)
profsave(profinfo,dirname)
```

Description

`profsave` executes the `profile('info')` function and saves the results in HTML format. `profsave` creates a separate HTML file for each function listed in the `FunctionTable` field of the structure returned by `profile`. By default, `profsave` stores the HTML files in a subfolder of the current folder named `profile_results`.

`profsave(profinfo)` saves the profiling results, `profinfo`, in HTML format. `profinfo` is a structure of profiling information returned by the `profile('info')` function.

`profsave(profinfo,dirname)` saves the profiling results, `profinfo`, in HTML format. `profsave` creates a separate HTML file for each function listed in the `FunctionTable` field of `profinfo` and stores them in the folder specified by `dirname`.

Examples

Run `profile` and save the results.

```
profile on
plot(magic(5))
profile off
profsave(profile('info'),'myprofile_results')
```

See Also

`profile`

Topics

"Profile to Improve Performance"

Introduced before R2006a

ProgressDialog Properties

Control appearance and behavior of progress dialog box

Description

Progress dialog boxes indicate that an operation is in progress by displaying an animated progress bar. The `uiprogressdlg` function creates a progress dialog box and sets any required properties before displaying it. By changing property values of a progress dialog box, you can modify certain aspects of its appearance and behavior. Use dot notation to refer to a specific object and property:

```
uf = uifigure;
d = uiprogressdlg(uf);
d.Value = .25;
```

Properties

Text and Styling

`Message` — message

`''` (default) | character vector | cell array of character vectors | string array

Message, specified as a character vector, cell array of character vectors, or string array. The message displays within the dialog box, above the progress bar.

To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array corresponds to a line of text. Hard breaks within each element, such as `'\n'`, create additional lines of text.

Example: `d = uiprogressdlg(uifigure,'Message','Calculating result.');`

`Title` — Title

`''` (default) | character vector | string scalar

Title, specified as a character vector or a string scalar. The title displays in the title bar of the dialog box.

Example: `d = uiprogressdlg(uifigure,'Title','Calculating');`

`Icon` — Icon

`''` (default) | predefined icon | custom icon

Icon, specified as a predefined icon or a custom icon.

Predefined Icon

This table lists the values for the predefined icons.

Value	Icon
`''` (default)	No icon displays.
`'question'`	
`'info'`	
`'success'`	
`'warning'`	
`'error'`	

Custom Icon

Specify a custom icon as one of these values:

- A character vector or string scalar that specifies the file name of an SVG, JPEG, GIF, or PNG image that is on the MATLAB path. Alternatively, you can specify a full path to the image file.
- A truecolor image array. See “Image Types” for more information.

Progress

`Value` — Fraction complete

`0` (default) | number between `0` and `1`

Fraction complete, specified as a number between `0` and `1`. The progress bar reaches its full length when the value is `1`. Change `Value` at different points in your code to provide a visual indication of progress in the running app.

Data Types: `double`

`ShowPercentage` — Show percentage
`'off'` (default) | `'on'`

Show percentage, specified as `'off'` or `'on'`. Set this property to `'on'` to display the fraction complete as a percentage in the dialog box.

`Indeterminate` — Indeterminate progress
`'off'` (default) | `'on'`

Indeterminate progress, specified as `'off'` or `'on'`. Set this property to `'on'` to provide an animated bar without any specific progress information. This animation is useful when you do not know how long a calculation will take.

To prevent indeterminate progress bars from displaying indefinitely, call the `close` function after completing your calculations.

Interactivity

`Cancelable` — Allow cancellation
`'off'` (default) | `'on'`

Allow cancellation, specified as `'off'` or `'on'`. A value of `'on'` displays a cancel button in the dialog box. You can customize the button label by specifying the `CancelText` property.

When you allow cancellation, you must check the value of the `CancelRequested` property, and call the `close` function when the value is `true`. Otherwise, the dialog box displays indefinitely.

`CancelText` — Cancel button text
`'Cancel'` (default) | character vector | string scalar

Cancel button text, specified as a character vector or string scalar. This property has an effect only when the `Cancelable` property is set to `'on'`.

Example: `d = uiprogressdlg(uifigure,'Cancelable','on','CancelText','Stop')`

`CancelRequested` — Cancel requested
`true` | `false`

Cancel requested, specified as `true` or `false`. Use this property when `Cancelable` is `'on'`, and you want to know whether the user clicked the cancel button. This property is `false` until the user clicks the cancel button, then the value changes to `true`.

See Also

`appdesigner` | `uiprogressdlg`

Introduced in R2018a

matlab.project.Project

Project object

Description

The `matlab.project.Project` object represents the currently loaded project. Use the project object to query the currently loaded project and to perform various operation on it.

Creation

To create a `matlab.project.Project` object, use the `currentProject` function to get the currently open project. For example:

```
proj = currentProject;
```

If no project is currently open, use the `openProject` function to load an existing project. For example:

```
proj = openProject("C:/projects/project1/");
```

Properties

`Name` — Project name
string

Project name, specified as a string.

`Categories` — Label categories
array of `Category` objects

Label categories for the project, specified as an array of `Category` objects.

`Shortcuts` — Shortcut files
array of `Shortcut` objects

Shortcut files for the project, specified as an array of `Shortcut` objects.

`ProjectPath` — Project path folders
array of `PathFolder` objects

Project path folders that are added to the MATLAB path, specified as an array of `PathFolder` objects.

`ProjectReferences` — Paths of referenced projects
array of `ProjectReference` objects

Paths of referenced projects, specified as an array of `ProjectReference` objects.

`Files` — Paths of project files
array of `ProjectFile` objects

Paths of project files, specified as an array of `ProjectFile` objects.

`RootFolder` — Project root folder
string

Project root folder, specified as a string. `RootFolder` includes the full path of the project root.

Object Functions

Create and Export Projects

close	Close project
export	Export project to archive
isLoaded	Determine if project is loaded
reload	Reload project

Setup Projects

addFile	Add file or folder to project
addFolderIncludingChildFiles	Add folder and child files to project
removeFile	Remove file from project
addPath	Add folder to project path
removePath	Remove folder from project path

addReference	Add referenced project to project
removeReference	Remove project reference
addStartupFile	Add startup file to project
addShutdownFile	Add shutdown file to project
removeStartupFile	Remove startup file from project startup list
removeShutdownFile	Remove shutdown file from project shutdown list
addShortcut	Add shortcut to project
removeShortcut	Remove shortcut from project

Manage Project Files

addLabel	Attach label to project file
createLabel	Create project label
removeLabel	Remove label from project
findLabel	Get project file label
createCategory	Create category of project labels
findCategory	Find project category of labels
removeCategory	Remove project category of labels
findFile	Find project file by name
listModifiedFiles	List modified files in project
listRequiredFiles	Get project file dependencies
refreshSourceControl	Update source control status of project files
updateDependencies	Update project dependencies

Examples

Get Times Table App Example Project

Open the Times Table App project and use `currentProject` to get a project object you can manipulate at the command line.

```
matlab.project.example.timesTable
proj = currentProject
```

```
proj = 

  Project with properties:

                      Name: "Times Table App"
   SourceControlIntegration: "Git"
         RepositoryLocation: "C:\myProjects\examples\repositories\TimesTableApp"
      SourceControlMessages: ["No remote tracking branch" "Branch status: Normal" ...
```

```
                 ReadOnly: 0
                 TopLevel: 1
             Dependencies: [1×1 digraph]
               Categories: [1×1 matlab.project.Category]
                    Files: [1×14 matlab.project.ProjectFile]
                Shortcuts: [1×4 matlab.project.Shortcut]
              ProjectPath: [1×3 matlab.project.PathFolder]
        ProjectReferences: [1×0 matlab.project.ProjectReference]
             StartupFiles: [1×0 string]
            ShutdownFiles: [1×0 string]
              Description: "This example project contains the source code and tests ...
               RootFolder: "C:\myProjects\examples\TimesTableApp"
      ProjectStartupFolder: "C:\myProjects\examples\TimesTableApp"
```

Find Project Commands

Open the Times Table App project and create a project object.

```
matlab.project.example.timesTable
proj = currentProject;
```

Find the functions you can execute on the project object.

```
methods(proj)
```

```
Methods for class matlab.project.Project:

addFile                        listModifiedFiles
addFolderIncludingChildFiles   listRequiredFiles
addPath                        refreshSourceControl
addReference                   reload
addShortcut                    removeCategory
addShutdownFile                removeFile
addStartupFile                 removePath
addprop                        removeReference
close                          removeShortcut
createCategory                 removeShutdownFile
export                         removeStartupFile
findCategory                   updateDependencies
findFile
isLoaded
```

Examine Project Properties

Get a project object, and examine its properties.

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Examine the project files.

```
files = proj.Files
```

```
files = 

  1×14 ProjectFile array with properties:

    Path
    Labels
    Revision
    SourceControlStatus
```

Examine the labels of the 13th file.

```
proj.Files(13).Labels
```

```
ans = 

  Label with properties:

           File: "C:\myProjects\examples\TimesTableApp\utilities\runTheseTests.m"
       DataType: 'none'
           Data: []
           Name: "Design"
    CategoryName: "Classification"
```

Get a particular file by name.

```
myfile = findFile(proj,"source/timesTableGame.m")
```

```
myfile = 

  ProjectFile with properties:

                   Path: "C:\myProjects\examples\TimesTableApp\source\timesTableGame.m"
                 Labels: [1×1 matlab.project.Label]
               Revision: ""
    SourceControlStatus: Unmodified
```

Find out what you can do with the file.

```
methods(myfile)
```

```
Methods for class matlab.project.ProjectFile:

addLabel     findLabel    removeLabel
```

See Also

`currentProject` | `openProject`

Topics

"Create and Edit Projects Programmatically"

Introduced in R2019a

addFile

Package: `matlab.project`

Add file or folder to project

Syntax

```
addFile(proj,fileOrFolder)
newfile = addFile(proj,fileOrFolder)
```

Description

`addFile(proj,fileOrFolder)` adds a file or folder to the project `proj`. When adding a folder to the project, MATLAB only adds the specified folder. To add the folder including any all subfolders and files, use `addFolderIncludingChildFiles` instead.

`newfile = addFile(proj,fileOrFolder)` returns a `ProjectFile` object for the added file.

Examples

Remove and Add Project Files

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Remove a file.

```
removeFile(proj,"source/timestable.mlapp")
```

Add the file back to the project.

```
addFile(proj,"source/timestable.mlapp");
```

Manipulate Project Files After Adding

Add a file to the project and then manipulate the file using the returned project file object.

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Save the current workspace variables to a file.

```
save("myvariables.mat");
```

Add the new file to the project and return a project file object.

```
file = addFile(proj,"myvariables.mat");
```

Use the project file object to manipulate the file, for example, by adding a label.

```
addLabel(file, "Classification", "Other")
```

Input Arguments

`proj` — Project
`matlab.project.Project` object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

`fileOrFolder` — Path of file or folder
character vector | string scalar

Path of the file or folder to add to the project, specified as a character vector or string scalar. Specify a path relative to the project root folder. The specified file or folder must be within the project root folder. File paths must include the file extension.

See Also

`addFolderIncludingChildFiles` | `currentProject` | `openProject` | `removeFile`

Topics

"Add Files to Project"

Introduced in R2019a

addFolderIncludingChildFiles

Package: `matlab.project`

Add folder and child files to project

Syntax

```
addFolderIncludingChildFiles(proj,folder)
newfile = addFolderIncludingChildFiles(proj,folder)
```

Description

`addFolderIncludingChildFiles(proj,folder)` adds to the specified project a folder and all of its subfolders and files. To add only the specified folder without any of its subfolders and files, use `addFile` instead.

`newfile = addFolderIncludingChildFiles(proj,folder)` returns a `ProjectFile` object for the added file.

Examples

Add Folders to a Project

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Create a new parent folder in the project folder.

```
newFolderPath = fullfile(proj.RootFolder, "newFolder");
mkdir(newFolderPath);
```

Create a new subfolder in the parent folder.

```
newSubFolderPath = fullfile(newFolderPath, "newSubFolder");
mkdir(newSubFolderPath);
```

Create a new file in the subfolder.

```
filepath = fullfile(newSubFolderPath, "newVariables.mat");
save(filepath)
```

Add the parent folder and its subfolders and files to the project. Check to make sure the file in the subfolder was correctly added.

```
projectFile = addFolderIncludingChildFiles(proj, newFolderPath);
findFile(proj,"newFolder/newSubFolder/newVariables.mat")
```

```
ans = 

  ProjectFile with properties:

              Path: "C:\myProjects\examples\TimesTableApp\newFolder\newSubFolder\newVariables.mat"
            Labels: [1×1 matlab.project.Label]
          Revision: ""
    SourceControlStatus: Added
```

Input Arguments

`proj` — Project
`matlab.project.Project` object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

`folder` — Path of folder
character vector | string scalar

Path of the folder to add to the project, specified as a character vector or string scalar. Specify a path relative to the project root folder. The specified folder must be within the project root folder.

See Also

`addFile` | `currentProject` | `openProject` | `removeFile`

Topics

"Add Files to Project"

Introduced in R2019a

addLabel

Package: `matlab.project`

Attach label to project file

Syntax

```
addLabel(file,categoryName,labelName)
addLabel(file,categoryName,labelName,labelData)
```

Description

`addLabel(file,categoryName,labelName)` attaches the specified label in the specified category to the specified file.

`addLabel(file,categoryName,labelName,labelData)` attaches the label with the specified text or numeric data. You cannot add label data to built-in labels as they are read-only.

Examples

Attach a Label to a Project File

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Get a file by name.

```
myfile = findFile(proj,"source/timesTableGame.m")
```

```
mmyfile =
```

```
  ProjectFile with properties:

                    Path: "C:\myProjects\examples\TimesTableApp\source\timesTableGame.m"
                  Labels: [1×1 matlab.project.Label]
                Revision: ""
    SourceControlStatus: Unmodified
```

View the existing label by getting the `Labels` property of the file.

```
myfile.Labels
```

```
ans =

  Label with properties:

            File: "C:\myProjects\examples\TimesTableApp\source\timesTableGame.m"
        DataType: 'none'
            Data: []
            Name: "Design"
    CategoryName: "Classification"
```

Attach the label `"Artifact"` to the file in the category `"Classification"`.

```
addLabel(myfile,"Classification","Artifact")
```

```
ans =

  Label with properties:

            File: "C:\myProjects\examples\TimesTableApp\source\timesTableGame.m"
        DataType: 'none'
            Data: []
            Name: "Artifact"
    CategoryName: "Classification"
```

There are now two labels, the original one and the added one. To view just the added one, index into the `Labels` property.

```
reviewlabel = myfile.Labels(1)
```

```
reviewlabel =

  Label with properties:

            File: "C:\myProjects\examples\TimesTableApp\source\timesTableGame.m"
        DataType: 'none'
            Data: []
            Name: "Artifact"
    CategoryName: "Classification"
```

Detach the new label from the file. The file now only has one label.

```
removeLabel(myfile,reviewlabel)
myfile
```

```
myfile =
```

```
  ProjectFile with properties:

                    Path: "C:\myProjects\examples\TimesTableApp\source\timesTableGame.m"
                  Labels: [1×0 matlab.project.Label]
                Revision: ""
    SourceControlStatus: Unmodified
```

Attach a Label to a Subset of Files

Attach the label `"Utility"` in the `"Classification"` category to all files in the project that have the `.m` file extension.

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Get the list of files.

```
files = proj.Files;
```

Loop through each file. To get just the file extension, use the `fileparts` function and take the last part. If a file has the extension `.m`, attach the label `"Utility"`.

```
for fileIndex = 1:numel(files)
   file = files(fileIndex);
   [~, ~, fileExtension] = fileparts(file.Path);
   if strcmp(fileExtension,".m")
       addLabel(file,"Classification","Utility");
   end
end
```

In the project **Files** view, the **Classification** column displays the label `Utility` for each `.m` file in the `utilities` folder.

Attach a Label and Label Data to a File

Create the label category `"Review"` and the label "To Review", and then attach the label and label data to a file. You cannot add label data to built-in labels as they are read-only.

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Create a new category "Review".

```
createCategory(proj,"Review","char");
```

For the new category, create a label "To Review".

```
reviewCategory = findCategory(proj,"Review");
createLabel(reviewCategory,"To Review");
```

Get a file by name.

```
myfile = findFile(proj,"source/timesTableGame.m")
```

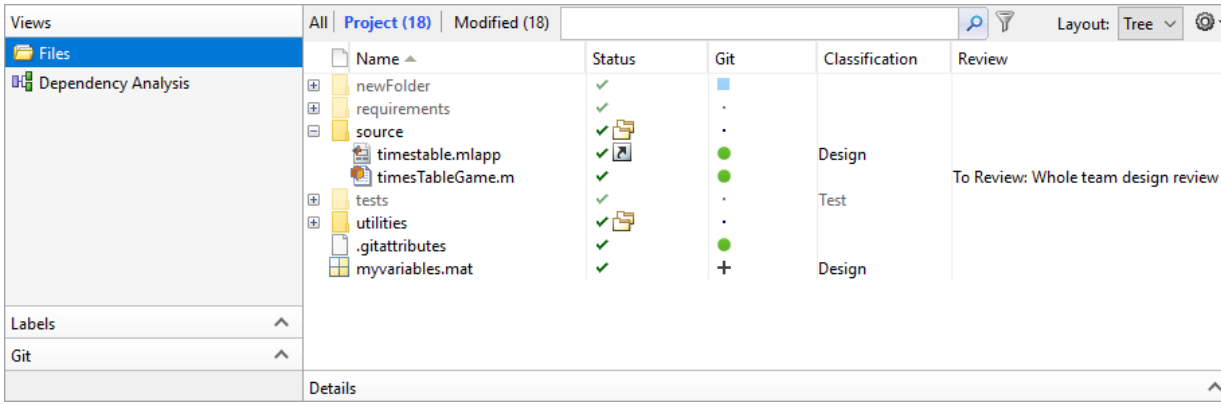
```
myfile = 

  ProjectFile with properties:

                  Path: "C:\myProjects\examples\TimesTableApp\source\timesTableGame.m"
                Labels: [1×1 matlab.project.Label]
              Revision: ""
    SourceControlStatus: Unmodified
```

Attach the label "To Review" and a character vector of label data to the file.

```
addLabel(myfile,"Review","To Review","Whole team design review")
```

```
ans = 

  Label with properties:

           File: "C:\myProjects\examples\TimesTableApp\source\timesTableGame.m"
       DataType: 'char'
           Data: 'Whole team design review'
           Name: "To Review"
    CategoryName: "Review"
```

In the project **Files** view, for the `timesTableGame.m` file, the **Review** column displays the `To Review` label with label data.



Alternatively, you can set or change label data using the `Data` property.

```
mylabel = myfile.Labels(1);
mylabel.Data = "Final review";
```

Input Arguments

`file` — File to label

`ProjectFile` object

File to label, specified as a `ProjectFile` object. You can get the `ProjectFile` object by examining the project's Files property (`proj.Files`), or by using `findFile` to find a file by name. The file must be in the project.

`categoryName` — Name of category for label

character vector | string scalar

Name of the category for the label, specified as a character vector or string scalar.

`labelName` — Name of label

character vector | string scalar | `LabelDefinition` object

Name of the label to attach, specified as a character vector, string scalar, or as a `LabelDefinition` object returned by the `file.Label` property or the `findLabel` function. You can specify a new label name that does not already exist in the project.

`labelData` — Data to attach to label
character vector | string scalar | numeric

Data to attach to the label, specified as a character vector, string scalar, or a numeric value. Data type depends on the label definition. Get a label to examine its `DataType` property using `file.Label` or `findLabel`.

See Also

`createLabel` | `currentProject` | `findFile` | `findLabel` | `openProject` | `removeLabel`

Topics

"Add Labels to Project Files"

Introduced in R2019a

addPath

Package: `matlab.project`

Add folder to project path

Syntax

```
folderonpath = addPath(proj,folder)
```

Description

`folderonpath = addPath(proj,folder)` adds a folder to the specified project path. The folder must be in the project. The project puts the folder on the MATLAB search path when it loads, and removes it from the path when it closes. To learn more, see “Specify Project Path”.

Examples

Add Folder to Project Path

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Create a new folder.

```
newfolder = fullfile(proj.RootFolder,"newfolder");
mkdir(newfolder);
```

Add the new folder to the project.

```
addFile(proj,newfolder);
```

Then, add the new folder to the project path.

```
newfolderonpath = addPath(proj,newfolder);
```

Input Arguments

`proj` — Project
`matlab.project.Project` object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

`folder` — Path of folder
character vector | string

Path of the folder to add to the project path, specified as a character vector or string. Specify the path relative to the project root folder. The folder must be within the root folder.

Output Arguments

`folderonpath` — Folder on project path
`PathFolder` object

Folder on project path, returned as a `PathFolder` object containing the added folder path. The project puts the folders on the MATLAB search path when it loads and removes them from the path when it closes.

See Also

`addFile` | `addFolderIncludingChildFiles` | `currentProject` | `openProject` | `removePath`

Topics

"Specify Project Path"

Introduced in R2019a

addReference

Package: `matlab.project`

Add referenced project to project

Syntax

```
projreference = addReference(proj,referenceFolder)
projreference = addReference(proj,referenceFolder,type)
```

Description

`projreference = addReference(proj,referenceFolder)` adds a reference to the project specified by `referencefolder`. The reference is added to the specified project, `proj`.

`projreference = addReference(proj,referenceFolder,type)` specifies the type of reference to create. Specify the type as either `"relative"` or `"absolute"`.

Examples

Add a Referenced Project

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Create a blank project.

```
projectToReference = matlab.project.createProject();
```

Reload the first project and add a reference to the new blank project. Reloading the first project is necessary because since only one project can be open at one time, MATLAB unloads the first project before creating and opening the new blank project.

```
reload(proj);
addReference(proj,projectToReference,"absolute");
```

Input Arguments

`proj` — Project
`matlab.project.Project` object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

`referenceFolder` — Path of referenced project folder
character vector | string scalar

Path of the referenced project folder, specified as a character vector or string scalar. Specify the path relative to the project root folder. The folder must be within the project root folder.

`type` — Type of reference
`"relative"` | `"absolute"`

Type of reference, specified as `"relative"` or `"absolute"`. Specify `"relative"` if your project hierarchy has a well-defined root relative to your project root, for example, a folder under source control. Specify `"absolute"` if the project you want to reference is in a location accessible to your computer, for example, a network drive.

Output Arguments

`projreference` — Referenced project
`ProjectReference` object

Referenced project, returned as a `ProjectReference` object containing information about the referenced project.

See Also

`currentProject` | `openProject` | `removeReference`

Topics

"Componentize Large Projects"

Introduced in R2019a

addShortcut

Package: `matlab.project`

Add shortcut to project

Syntax

```
shortcut = addShortcut(proj,file)
```

Description

`shortcut = addShortcut(proj,file)` adds a shortcut to the specified file in the project. In projects, a shortcut can be used to perform common project tasks such as opening important files and loading data.

To set the shortcut to run at startup or shutdown, see “Specify Startup and Shutdown Files”.

Examples

Add a Shortcut

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Create a new file.

```
filepath = fullfile(proj.RootFolder,"newvariables.mat");
save(filepath)
```

Add this new file to the project.

```
projectFile = addFile(proj,filepath)
```

Add a new shortcut to the new file.

```
shortcut = addShortcut(proj,filepath);
```

```
shortcut = 

  Shortcut with properties:

     Name: "newvariables"
    Group: "General"
     File: "C:\myProjects\examples\TimesTableApp\newvariables.mat"
```

Input Arguments

`proj` — Project
`matlab.project.Project` object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

`file` — Path of file
character vector | string scalar

Path of the shortcut file, including the file extension, specified as a character vector or string scalar. Specify the path relative to the project root folder. The file must be within the project root folder.

Output Arguments

`shortcut` — Shortcut
`Shortcut` object

`Shortcut` object containing information about the shortcut.

See Also

`currentProject` | `openProject` | `removeShortcut`

Topics

"Create Shortcuts to Frequent Tasks"

Introduced in R2019a

addShutdownFile

Package: `matlab.project`

Add shutdown file to project

Syntax

```
addShutdownFile(proj,file)
```

Description

`addShutdownFile(proj,file)` adds a shutdown file to the specified project. When you close the project, the project runs the shutdown file automatically.

Examples

Add a Shutdown File

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Specify executable MATLAB code to run as the project shuts down. Automatically run the file when the project closes, by making it a shutdown file.

```
filepath = fullfile("utilities","runTheseTests.m");
addShutdownFile(proj,filepath);
```

Input Arguments

`proj` — Project
`matlab.project.Project` object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

`file` — Path of file
character vector | string scalar

Path of the MATLAB file to add, including the file extension, specified as a character vector or string scalar. Specify the path relative to the project root folder. The file must be within the project root folder.

See Also

`addShutdownFile` | `currentProject` | `openProject` | `removeStartupFile`

Topics

"Automate Startup and Shutdown Tasks"

Introduced in R2019a

addStartupFile

Package: `matlab.project`

Add startup file to project

Syntax

```
addStartupFile(proj,file)
```

Description

`addStartupFile(proj,file)` adds a startup file to the specified project. Startup files automatically run (if they are `.m` and `.p` files), load (if they are `.mat` files), or open (if they are Simulink models) when you open the project.

Examples

Add a Startup File

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Automatically run the Times Table app when the project opens by making the `runTheseTests.m` file a startup file.

```
filepath = fullfile("utilities","runTheseTests.m");
addStartupFile(proj,filepath);
```

Input Arguments

`proj` — Project
`matlab.project.Project` object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

`file` — Path of file
character vector | string scalar

Path of the MATLAB file to add, including the file extension, specified as a character vector or string scalar. Specify the path relative to the project root folder. The file must be within the project root folder.

See Also

`addShutdownFile` | `currentProject` | `openProject` | `removeStartupFile`

Topics

"Automate Startup and Shutdown Tasks"

Introduced in R2019a

close

Package: `matlab.project`

Close project

Syntax

```
close(proj)
```

Description

`close(proj)` closes the specified project.

Examples

Open and Close a Project

Open a project. To be able to manipulate it at the command line, return a project object.

```
proj = openProject("C:/projects/project1/myproject.prj")
```

Close the project using the project object.

```
close(proj)
```

Input Arguments

`proj` — Project
`matlab.project.Project` object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

See Also

`currentProject` | `openProject`

Introduced in R2019a

createCategory

Package: `matlab.project`

Create category of project labels

Syntax

```
createCategory(proj,categoryName)
createCategory(proj,categoryName,dataType)
createCategory(proj,categoryName,dataType,"single-valued")
newcategory = createCategory( ___ )
```

Description

`createCategory(proj,categoryName)` creates a new category of labels in the specified project.

`createCategory(proj,categoryName,dataType)` also specifies the type of data to store in labels of the new category. For more information on data types, see “Fundamental MATLAB Classes”.

`createCategory(proj,categoryName,dataType,"single-valued")` specifies a single-valued category, where you can attach only one label from the category to a file. If you do not specify a single-valued category, then you can attach multiple labels from the category to a file.

`newcategory = createCategory( ___ )` returns the new category as a `Category` object. Use this syntax with any of the previous input argument combinations.

Examples

Create New Category of Labels

Create a new category of labels to indicate the owner of a file, and attach a new label from it to a file, along with label data.

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Create a new category of labels, called `Engineers`, to indicate which engineer owns a file. These labels have the `char` data type for attaching character vector data.

```
createCategory(proj,"Engineers","char");
```

Use the `findCategory` function to get the new category.

```
engineersCategory = findCategory(proj,"Engineers");
```

Create labels in the new category.

```
createLabel(engineersCategory,"Tom");
createLabel(engineersCategory,"Harry");
```

Attach one of the new labels to a file in the project.

```
myfile = findFile(proj,"source/timesTableGame.m");
addLabel(myfile,"Engineers","Tom");
```

Get the label and add data.

```
label = findLabel(myfile,"Engineers","Tom");
label.Data = "Maintenance responsibility";
disp(label)
```

```
Label with properties:

            File: [1x80 char]
            Data: "Maintenance responsibility"
        DataType: 'char'
            Name: "Tom"
    CategoryName: "Engineers"
```

Create Category for Numeric Labels

Create a category of labels with the `double` data type, used for numeric data.

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Create a new category of labels. Specify "`double`" as the data type. This is the data type MATLAB uses for numbers by default.

```
coverageCategory = createCategory(proj,"Coverage","double")
```

```
category = 

  Category with properties:

                 Name: "Coverage"
             DataType: 'double'
    LabelDefinitions: []
```

Create a label in the new category and add it to a file in the project.

```
createLabel(coverageCategory,"Test");
myfile = findFile(proj,"source/timesTableGame.m");
label = addLabel(myfile,"Coverage","Test");
```

Add numeric data to the label.

```
label.Data = 80
```

```
newLabel = 

  Label with properties:

            File: "C:\myProjects\examples\TimesTableApp\source\timesTableGame.m"
        DataType: 'double'
            Data: 80
            Name: "Test"
    CategoryName: "Coverage"
```

Create Single-Valued Category

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Create a category of labels for file ownership, and specify single-valued to restrict only one label in the category per file.

```
engineersCategory = createCategory(proj,"Engineers","char", "single-valued");
```

Create a label in the new category and add it to a file in the project.

```
createLabel(engineersCategory,"Tom");
myfile = findFile(proj,"source/timesTableGame.m");
addLabel(myfile,"Engineers","Tom")
```

```
ans = 

  Label with properties:

            File: "C:\myProjects\examples\TimesTableApp\source\timesTableGame.m"
        DataType: 'char'
            Data: []
            Name: "Tom"
    CategoryName: "Engineers"
```

Create a second label in the new category and add it to the same file in the project. MATLAB replaces the first label with the new one.

```
createLabel(engineersCategory,"Harry");
addLabel(myfile,"Engineers","Harry")
```

```
ans = 

  Label with properties:

            File: "C:\myProjects\examples\TimesTableApp\source\timesTableGame.m"
        DataType: 'char'
            Data: []
            Name: "Harry"
    CategoryName: "Engineers"
```

Check to see if the first label is still attached to the file.

```
findLabel(myfile,"Engineers","Tom")
```

```
ans = 

  0×0 Label array with properties:
```

```
File
DataType
Data
Name
CategoryName
```

Input Arguments

`proj` — Project
`matlab.project.Project` object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

`categoryName` — Name of category
character vector | string scalar

Name of the category of labels to create, specified as a character vector or string scalar.

`dataType` — Type of data to store in labels
character vector | string scalar

The type of data to store in labels in the new category, specified as a character vector or string scalar.

Tips

After you create a new category, you can create labels in the new category using the `createLabel` function.

See Also

`createLabel` | `currentProject` | `openProject`

Topics

"Add Labels to Project Files"
"Fundamental MATLAB Classes"

Introduced in R2019a

createLabel

Package: `matlab.project`

Create project label

Syntax

```
createLabel(category,newLabelName)
```

Description

`createLabel(category,newLabelName)` creates a new label, in the specified category. Use this syntax if you previously got a `Category` object by accessing a `Categories` property, for instance by using syntax like `proj.Categories(1)`.

Examples

Create a New Label

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Get the first existing category.

```
cat = proj.Categories(1)
```

```
cat = 

  Category with properties:

                 Name: "Classification"
        SingleValued: 1
```

```
            DataType: "none"
    LabelDefinitions: [1×7 matlab.project.LabelDefinition]
```

Define a new label in the category.

```
createLabel(cat,"Future");
```

Create a New Category of Labels for File Ownership

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Create a new category of labels called `"Engineers"` which can be used to denote file ownership in a project. These labels have the `char` datatype for attaching character vector data.

```
createCategory(proj,"Engineers","char");
```

Get the new category by name using the `findCategory` function

```
engineersCategory = findCategory(proj,"Engineers");
```

Create labels in the new category.

```
createLabel(engineersCategory,"Tom");
createLabel(engineersCategory,"Harry");
```

Attach one of the new labels to a file in the project.

```
myfile = findFile(proj,"source/timesTableGame.m");
addLabel(myfile,"Engineers","Tom");
```

Get the label and add data.

```
label = findLabel(myfile,"Engineers","Tom");
label.Data = "Maintenance responsibility";
disp(label)
```

```
  Label with properties:

           File: "C:\myProjects\examples\TimesTableApp\source\timesTableGame.m"
```

```
        DataType: 'char'
            Data: "Maintenance responsibility"
            Name: "Tom"
    CategoryName: "Engineers"
```

Input Arguments

`category` — Category
`Category` object

Category for the new label, specified as a `Category` object. Get the `Category` object by accessing a `Categories` property, using a syntax like `proj.Categories(1)`, or use the `findCategory` function. To create a new category, use the `createCategory` function.

`newLabelName` — The name of the new label
character vector

The name of the new label, specified as a character vector.

Tips

- To create and attach a new label in an existing category using a single step, use `addLabel` instead.
- To create a new category of labels, use `createCategory` first.

See Also

`addLabel` | `createCategory` | `currentProject` | `openProject`

Topics

"Add Labels to Project Files"

Introduced in R2019a

export

Package: `matlab.project`

Export project to archive

Syntax

```
export(proj,archiveName)
export(proj,archiveName,'ArchiveReferences',archiveReferences)
```

Description

`export(proj,archiveName)` exports the specified project to a new project archive file named `archiveName`. `archiveName` must include the `.mlproj` extension or no extension at all. If no extension is specified, MATLAB appends the `.mlproj` extension to the file.

The project archive preserves the project files, structure, labels, and shortcuts, and does not include any source control information. You can use the project archive to send the project to customers, suppliers, or coworkers who do not have access to your source control repository. Recipients can create a new project from the archive by double-clicking the project archive file.

By default, this function packages all the project references in the archive. You might not want this option if the project contains references to absolute networks paths. In this case, the function duplicates the project itself and treats it as a relative reference upon extraction.

`export(proj,archiveName,'ArchiveReferences',archiveReferences)` exports the specified project to a new project archive file and specifies whether to include the project references in the package.

Examples

Export Project to an Archived Project File

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Export the project to an archived project file.

```
export(proj,"timestableproj.mlproj")
```

You can now send someone the project by sharing the file `timestableproj.mlproj` with them.

Input Arguments

`proj` — Project
`matlab.project.Project` object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

`archiveName` — Archive file name or path
character vector | string scalar

Archive file name or path, specified as a character vector or string scalar ending in the file extension `.mlproj`. If no extension is specified, MATLAB appends the `.mlproj` extension to the file name.

If `archiveName` is a file name, MATLAB exports the file to the current folder. You also can specify a fully qualified path name.

Example: `"project.mlproj"`

`archiveReferences` — Include references in packaging
true (default) | false

Include references in the package, specified as true or false.

If true, MATLAB exports the project and includes the references in the package. Extracting the archive creates a copy of the referenced projects and links them to the top-level project. It also includes a copy of projects referenced by your referenced projects.

If false, MATLAB exports the project and preserves the existing absolute and relative link to the references. For a working project on extraction, the references must be at the same absolute and relative paths.

The argument is ignored if the project does not have references.

Data Types: `logical`

See Also

`currentProject` | `openProject`

Topics

"Share Projects"

Introduced in R2019a

findCategory

Package: `matlab.project`

Find project category of labels

Syntax

```
category = findCategory(proj,categoryName)
```

Description

`category = findCategory(proj,categoryName)` gets the category from the specified project with the specified name. You need to get a category before you can use the `addLabel` or `removeLabel` function.

Examples

Get a Category of Project Labels

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Use `findCategory` to get a category of labels by name.

```
category = findCategory(proj,"Classification")
```

```
category = 

  Category with properties:

                 Name: "Classification"
       SingleValued: 1
```

```
            DataType: "none"
    LabelDefinitions: [1×7 matlab.project.LabelDefinition]
```

Alternatively, you can get categories by index. Get the first category.

```
proj.Categories(1)
```

```
ans = 

  Category with properties:

                Name: "Classification"
        SingleValued: 1
            DataType: "none"
    LabelDefinitions: [1×7 matlab.project.LabelDefinition]
```

Find out what you can do with the category.

```
methods(category)
```

```
Methods for class matlab.project.Category:

createLabel  findLabel    removeLabel
```

Input Arguments

`proj` — Project
`matlab.project.Project` object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

`categoryName` — Name of category
character vector | string scalar

Name of the category to get, specified as a character vector or string scalar.

Output Arguments

`category` — Category of labels
`Category` object

Category of labels, returned as a `Category` object that you can query or modify. If the specified category is not found, the function returns an empty array.

See Also

`addLabel` | `currentProject` | `openProject` | `removeLabel`

Topics

"Add Labels to Project Files"

Introduced in R2019a

findFile

Package: `matlab.project`

Find project file by name

Syntax

```
file = findFile(proj,fileOrFolder)
```

Description

`file = findFile(proj,fileOrFolder)` gets the file in the specified project with the specified name. You need to get a file before you can query labels, or use the `addLabel` or `removeLabel` function.

Examples

Find File by Name

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Use `findFile` to get a file by name. If the file is in a subfolder of the project root, you need to specify the full path.

```
myfile = findFile(proj,"source/timesTableGame.m")
```

```
myfile = 

  ProjectFile with properties:

                   Path: "C:\myProjects\examples\TimesTableApp\source\timesTableGame.m"
                 Labels: [1×1 matlab.project.Label]
```

```
            Revision: ""
    SourceControlStatus: Unmodified
```

Alternatively, you can get files by index. Get the first file.

```
file = proj.Files(1);
```

Find out what you can do with the file.

```
methods(file)
```

```
Methods for class matlab.project.ProjectFile:

addLabel     findLabel    removeLabel
```

Input Arguments

`proj` — Project
`matlab.project.Project` object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

`fileOrFolder` — Path of file or folder
character vector | string scalar

Path of the file or folder to find, relative to the project root folder, specified as a character vector or string scalar. The specified value must include the full path from the project root to the file, including the names of any subfolders along the way. It must also include the file extension. The file or folder must be within the root folder.

Output Arguments

`file` — Project file
`ProjectFile` object

Project file, returned as a `ProjectFile` object that you can query or modify.

See Also

`addLabel` | `currentProject` | `findCategory` | `findLabel` | `openProject` | `removeLabel`

Topics

"Add Labels to Project Files"

Introduced in R2019a

findLabel

Package: `matlab.project`

Get project file label

Syntax

```
label = findLabel(file,categoryName,labelName)
label = findLabel(file,labelDefinition)
label = findLabel(category,labelName)
```

Description

`label = findLabel(file,categoryName,labelName)` gets the specified label, in the specified category, from the specified file. It returns the label definition and its attached data. Use this syntax when you know the label name and category. If the label is not found, `findLabel` returns an empty array.

`label = findLabel(file,labelDefinition)` gets the label defined by the specified label definition object. Use this syntax if you previously got a `labelDefinition` by accessing a `Labels` property, for instance by using an expression like `myfile.Labels(1)`.

`label = findLabel(category,labelName)` gets a label using a category object rather than a file name and category name. Use this syntax if you have a category object gotten from the `proj.Categories` property or by using the `findCategory` function.

Examples

Find Files with the Label

Find all project files with the label "`Utility`"

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Get the list of project files.

```
files = proj.Files;
```

Loop through the files. Get each file's extension by taking the last element returned by the `fileparts` function. If the file has the extension `.m`, attach the label "`Utility`".

```
for fileIndex = 1:numel(files)
   file = files(fileIndex);
   [~,~,fileExtension] = fileparts(file.Path);
   if strcmp(fileExtension,".m")
      addLabel(file,"Classification","Utility");
   end
end
```

Use the `findLabel` function to find all the files with the label "`Utility`" and add them to the array `utilityFilesToReview`.

```
utilityFilesToReview = {};
for jj=1:numel(files)
   thisFile = files(jj);
   label = findLabel(thisFile,"Classification","Utility");
   if (~isempty(label))
      % This is a file labeled "Utility". Add to the
      % list of utility files.
      utilityFilesToReview = [utilityFilesToReview; thisFile];
   end
end
```

Find Label by Name or Definition

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Get a file by name.

```
myfile = findFile(proj,"source/timesTableGame.m");
```

Get a label from that file by name.

```
label = findLabel(myfile,"Classification","Design");
```

```
label = 

  Label with properties:

            File: "C:\myProjects\examples\TimesTableApp\source\timesTableGame.m"
        DataType: 'none'
            Data: []
            Name: "Design"
    CategoryName: "Classification"
```

Examine the `Labels` property of the file to get an array of `Label` objects, one for each label attached to the file. Index into the `Labels` property to get the label definition attached to the particular file.

```
labels = myfile.Labels
labeldefinition = myfile.Labels(1)
```

Get a label from the label definition.

```
label = findLabel(myfile,labeldefinition);
```

Find Label by Category

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Get a category.

```
category = proj.Categories(1)
```

```
category = 

  Category with properties:

                Name: "Classification"
```

```
        SingleValued: 1
            DataType: "none"
    LabelDefinitions: [1×7 matlab.project.LabelDefinition]
```

Get a label definition from that category.

```
ld = findLabel(category,"Design")
```

```
ld = 

  LabelDefinition with properties:

            Name: "Design"
    CategoryName: "Classification"
```

Input Arguments

`file` — File to search in
`ProjectFile` object

File to search in, specified as a `ProjectFile` object or an array of file objects. You can get the file object by examining the project's Files property (using the syntax `proj.Files`), or use `findFile` to get a file by name. The file must be in the specified project.

`categoryName` — Name of category
character vector | string scalar

Name of category for the label, specified as a character vector or string scalar.

`labelName` — Name of label
character vector | string scalar

Name of label, specified as a character vector or string scalar.

`labelDefinition` — Label definition
`LabelDefinition` object

Label definition, specified as a `LabelDefinition` object gotten from the `file.Label` property.

`category` — Category object
`Category` object

Category object. Get a category object from the `proj.Categories` property or by using the `findCategory` function.

Output Arguments

`label` — Label
`Label` object

Label, returned as a `Label` object.

See Also

`addLabel` | `createLabel` | `currentProject` | `findFile` | `openProject`

Topics

"Add Labels to Project Files"

Introduced in R2019a

isLoaded

Package: `matlab.project`

Determine if project is loaded

Syntax

```
TF = isLoaded(proj)
```

Description

`TF = isLoaded(proj)` determines if the specified project is loaded.

Examples

Find Out if Project Is Loaded

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Find out if the project is still loaded.

```
loaded = isLoaded(proj)
```

```
loaded =

     1
```

Close the project and check again.

```
close(proj)
loaded = isLoaded(proj)
```

```
loaded =

     0
```

Input Arguments

proj — Project
`matlab.project.Project` object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

Output Arguments

TF — Load status
`1` | `0`

Load status, returned as `1` (true) if the project is loaded.

Data Types: `logical`

See Also

`currentProject` | `openProject` | `reload`

Introduced in R2019a

listModifiedFiles

Package: `matlab.project`

List modified files in project

Syntax

```
modifiedfiles = listModifiedFiles(proj)
```

Description

`modifiedfiles = listModifiedFiles(proj)` returns the modified project files in the specified project. The function refreshes the source control status in the project, and then returns an array of project files which are listed in the **Modified Files** view of the project.

Examples

List Modified Files in Project

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Modify the project name.

```
proj.Name = "Better Times Table App";
```

Get all the modified files in the project.

```
modifiedfiles = listModifiedFiles(proj)
```

```
modifiedfiles =
```

```
1x2 ProjectFile array with properties:

  Path
  Labels
  Revision
  SourceControlStatus
```

Observe that two files are modified. Compare this with the **Modified Files** view in the project, where you can see two modified project metadata files in the `resources` folder

Input Arguments

`proj` — Project
`matlab.project.Project` object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

Output Arguments

`modifiedfiles` — Modified files
array of `ProjectFile` objects

Modified files, returned as an array of `ProjectFile` objects.

See Also

`currentProject` | `openProject` | `refreshSourceControl`

Topics

"Use Source Control with Projects"

Introduced in R2019a

listRequiredFiles

Package: `matlab.project`

Get project file dependencies

Syntax

```
files = listRequiredFiles(proj,file)
```

Description

`files = listRequiredFiles(proj,file)` returns the names of the files that the specified file requires to run.

Examples

Get Required Files

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Get the files required by the Times Table app file.

```
file = "source/timestable.mlapp"
files = listRequiredFiles(proj, file);
```

```
files =

  2×1 string array
```

```
"C:\myProjects\examples\TimesTableApp\source\timestable.mlapp"
"C:\myProjects\examples\TimesTableApp\source\timesTableGame.m"
```

Input Arguments

`proj` — Project
`matlab.project.Project` object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

`file` — Path of file
character vector | string scalar | `ProjectFile` object

Path of the file, including the file extension, specified as a character vector, string scalar, or a `ProjectFile` object. Specify `file` as an absolute file path or a path relative to the project root folder. The file must be within the root folder.

Output Arguments

`file` — Project file
string array

Required files, returned as a string array.

See Also

`currentProject` | `openProject`

Topics

"Analyze Project Dependencies"

Introduced in R2019a

refreshSourceControl

Package: `matlab.project`

Update source control status of project files

Syntax

```
refreshSourceControl(proj)
```

Description

`refreshSourceControl(proj)` updates the source control status for all files in the specified project. Use `refreshSourceControl` to get the latest source control information before querying the `SourceControlStatus` property on individual files.

If you use `listModifiedFiles` to find all modified files in the project, you do not need to call `refreshSourceControl` first.

Examples

Refresh Source Control Information on Files in the Project

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Refresh source control status before querying individual files.

```
refreshSourceControl(proj)
```

Get all the project files that are Unmodified. Use the `ismember` function to get an array of `logicals` stating which files in the Times Table App project are unmodified. Use the array to get the list of unmodified files.

```
unmodifiedStatus = ismember([mainProject.Files.SourceControlStatus], ...
    matlab.sourcecontrol.Status.Unmodified);
mainProject.Files(unmodifiedStatus)
```

```
ans = 
  1×9 ProjectFile array with properties:

    Path
    Labels
    Revision
    SourceControlStatus
```

Input Arguments

proj — Project
`matlab.project.Project` object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

See Also

`currentProject` | `listModifiedFiles` | `openProject`

Topics

"Use Source Control with Projects"

Introduced in R2019a

reload

Package: `matlab.project`

Reload project

Syntax

```
reload(proj)
```

Description

`reload(proj)` reloads the project. Use `reload` when you want to re-run the project startup shortcuts.

Examples

Reload Project

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

When you want to run the startup shortcuts again, reload the project.

```
reload(proj)
```

Input Arguments

`proj` — Project
`matlab.project.Project` object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

See Also

`currentProject` | `isLoaded` | `openProject`

Introduced in R2019a

removeCategory

Package: `matlab.project`

Remove project category of labels

Syntax

```
removeCategory(proj,categoryName)
```

Description

`removeCategory(proj,categoryName)` removes a category of labels from the specified project.

Examples

Remove Category

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Create a new category of labels.

```
createCategory(proj,"Engineers","char");
```

Remove the new category of labels.

```
removeCategory(proj,"Engineers");
```

A message appears warning you that you cannot undo the operation. Click **Continue**. You can configure warnings in the MATLAB Project Preferences.

Input Arguments

`proj` — Project
`matlab.project.Project` object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

`categoryName` — Name of category
character vector | string scalar

Name of the category to remove, specified as a character vector or string scalar. The category must exist in the project.

See Also

`createCategory` | `currentProject` | `findCategory` | `openProject`

Topics

"Add Labels to Project Files"

Introduced in R2019a

removeFile

Package: `matlab.project`

Remove file from project

Syntax

```
removeFile(proj,file)
```

Description

`removeFile(proj,file)` removes a file from the specified project.

Examples

Remove File from Project

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Remove a file.

```
removeFile(proj,"source/timesTableGame.m")
```

Add the file back to the project.

```
addFile(proj,"source/timesTableGame.m")
```

Input Arguments

proj — Project
matlab.project.Project object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

file — File to remove
character vector | string scalar | `ProjectFile` object

File to remove, specified as a character vector, string scalar, or a `ProjectFile` object. If you specify `file` as a character vector or string scalar, it must contain the path of the file to remove relative to the project root folder, including the file extension. The file must be in the project.

See Also

`addFile` | `currentProject` | `findFile` | `openProject`

Topics

"Add Files to Project"

Introduced in R2019a

removeLabel

Package: `matlab.project`

Remove label from project

Syntax

```
removeLabel(category,labelName)
removeLabel(file,categoryName,labelName)
removeLabel(file,labelDefinition)
```

Description

`removeLabel(category,labelName)` removes the specified label from the specified category of labels in the currently loaded project. Use this syntax to remove a label from a category by name.

`removeLabel(file,categoryName,labelName)` removes the specified label in the specified category from the specified file. Use this syntax to remove a label from a file by name.

`removeLabel(file,labelDefinition)` removes the label specified by the label definition object `labelDefinition`. Before you can remove the label, you need to get the label definition object from the `file.Label` property or by using `findLabel`.

Examples

Remove a Label

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Examine the first existing category.

```
cat = proj.Categories(1)
```

```
cat = 

  Category with properties:

                 Name: "Classification"
        SingleValued: 1
            DataType: "none"
    LabelDefinitions: [1×7 matlab.project.LabelDefinition]
```

Define a new label in the category.

```
createLabel(cat,"Future");
```

Remove the new label.

```
removeLabel(cat,"Future");
```

Input Arguments

`category` — Category of labels
`Category` object

Category of labels, specified as a `Category` object. Get a `Category` object from the `proj.Categories` property or by using `findCategory`.

`labelName` — Name of label
character vector | string scalar

Name of the label to remove, specified as a character vector or string scalar.

`file` — File to remove label from
`ProjectFile` object

File to remove the label from, specified as a `ProjectFile` object. You can get the `ProjectFile` object by examining the project's Files property (`proj.Files`), or use `findFile` to find a file by name. The file must be within the project root folder.

`categoryName` — Name of category that contains label
character vector | string scalar

Name of the category that contains the label to remove, specified as a character vector or string scalar.

`labelDefinition` — Label to remove
`LabelDefinition` object

Name of the label to remove, specified as a `LabelDefinition` object returned by the `file.Label` property or `findLabel`.

See Also

`addLabel` | `createLabel` | `currentProject` | `findCategory` | `findFile` | `findLabel` | `openProject`

Topics

"Add Labels to Project Files"

Introduced in R2019a

removePath

Package: `matlab.project`

Remove folder from project path

Syntax

```
removePath(proj,folderpath)
```

Description

`removePath(proj,folderpath)` removes a folder from the current project path. The folder must be in the project.

Examples

Remove a Folder from the Project Path

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Create a new folder.

```
folderpath = fullfile(proj.RootFolder,"folder");
mkdir(filepath);
```

Add this new folder to the project.

```
projectFile = addFile(proj,folderpath);
```

Add this new folder to the project path.

```
folderpath = addPath(proj,folderpath);
```

Remove the new folder from the project path.

```
removePath(proj,folderpath)
```

Input Arguments

`proj` — Project
`matlab.project.Project` object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

`folderpath` — Path of folder
character vector | string scalar

Path of the folder to remove from the project path, specified as a character vector or string scalar. Specify the path relative to the project root folder. The folder must be within the root folder.

See Also

`addPath` | `currentProject` | `openProject`

Topics

"Specify Project Path"

Introduced in R2019a

removeReference

Package: `matlab.project`

Remove project reference

Syntax

```
removeReference(proj,folder)
```

Description

`removeReference(proj,folder)` removes the reference to the project specified by `folder`. The reference is removed from the specified project, `proj`.

Examples

Remove a Referenced Project

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Create a new blank project.

```
projectToReference = matlab.project.createProject();
```

Reload the first project and add a reference to the new blank project.

```
reload(proj);
addReference(proj,projectToReference,"absolute");
```

Remove the reference to the blank project.

```
removeReference(proj,projectToReference);
```

Input Arguments

`proj` — Project
`matlab.project.Project` object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

`folder` — Path of folder
character vector | string scalar

Path of the folder containing the referenced project to remove, specified as a character vector or string scalar. Specify the path relative to the project root folder. The folder must be within the root folder.

See Also

`addReference` | `currentProject` | `openProject`

Topics

"Componentize Large Projects"

Introduced in R2019a

removeShortcut

Package: `matlab.project`

Remove shortcut from project

Syntax

```
removeShortcut(proj,shortcut)
```

Description

`removeShortcut(proj,shortcut)` removes the specified shortcut from the specified project.

Examples

Remove a Shortcut

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Create a new file.

```
filepath = fullfile(proj.RootFolder,"newVariables.mat");
save(filepath)
```

Add this new file to the project.

```
projectFile = addFile(proj,filepath)
```

Add a new shortcut to the new file.

```
shortcut = addShortcut(proj,filepath);
```

Remove the shortcut.

```
removeShortcut(proj,shortcut);
```

Input Arguments

`proj` — Project
`matlab.project.Project` object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

`shortcut` — Shortcut to remove
character vector | string scalar | `Shortcut` object

Shortcut to remove, specified as a character vector, string scalar, or `Shortcut` object. If you specify `shortcut` as a character vector or string scalar, it must contain the path of the shortcut to remove relative to the project root folder, including the file extension. The shortcut must be in the project.

See Also

`addShortcut` | `currentProject` | `openProject`

Topics

"Create Shortcuts to Frequent Tasks"

Introduced in R2019a

removeShutdownFile

Package: `matlab.project`

Remove shutdown file from project shutdown list

Syntax

```
removeShutdownFile(proj,file)
```

Description

`removeShutdownFile(proj,file)` removes the shutdown file from the shutdown list in the specified project.

Examples

Remove a Shutdown File

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Specify a file of executable MATLAB code to run as the project shuts down. Automatically run the file when the project closes, by making it a shutdown file.

```
filepath = fullfile("utilities","runTheseTests.m");
addShutdownFile(proj,filepath);
```

Remove the shutdown file. The code no longer runs when the project shuts down.

```
removeShutdownFile(proj,filepath);
```

Input Arguments

`proj` — Project
`matlab.project.Project` object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

`file` — Path of file
character vector | string scalar

Path of the MATLAB file to remove, including the file extension, specified as a character vector or string scalar. Specify the path relative to the project root folder. The file must be within the root folder.

See Also

`addShutdownFile` | `currentProject` | `openProject` | `removeStartupFile`

Topics

"Automate Startup and Shutdown Tasks"

Introduced in R2019a

removeStartupFile

Package: `matlab.project`

Remove startup file from project startup list

Syntax

```
removeStartupFile(proj,file)
```

Description

`removeStartupFile(proj,file)` removes the specified startup file from the startup list in the specified project.

Examples

Remove a Startup File

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Automatically run the `TimesTable` app when the project opens by making the `runTimesTable.m` file a startup file.

```
filepath = fullfile("utilities","runTheseTests.m");
addStartupFile(proj,filepath);
```

Remove the startup file. The app no longer runs automatically when the project opens.

```
removeStartupFile(proj, filepath);
```

Input Arguments

`proj` — Project
`matlab.project.Project` object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

`file` — Path of file
character vector | string scalar

Path of the file to remove, including the file extension, specified as a character vector or string scalar. Specify the path relative to the project root folder. The folder must be within the root folder.

See Also

`addStartupFile` | `currentProject` | `openProject` | `removeShutdownFile`

Topics

"Automate Startup and Shutdown Tasks"

Introduced in R2019a

updateDependencies

Package: `matlab.project`

Update project dependencies

Syntax

```
updateDependencies(proj)
```

Description

`updateDependencies(proj)` runs a dependency analysis on the specified project to update the known dependencies between project files.

Examples

Update the Project Dependencies

Open the Times Table App project. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable
proj = currentProject;
```

Update the project dependencies

```
updateDependencies(proj);
```

Get the files required by the `timestable.mlapp` project file.

```
g = proj.Dependencies;
requiredFiles = bfsearch(g, which("source/timestable.mlapp"))
```

```
requiredFiles =

  2×1 cell array
```

```
{'C:\myProjects\examples\TimesTableApp\source\timestable.mlapp'}
{'C:\myProjects\examples\TimesTableApp\source\timesTableGame.m'}
```

Input Arguments

`proj` — Project
`matlab.project.Project` object

Project, specified as a `matlab.project.Project` object. Use `currentProject` to create a project object from the currently loaded project.

See Also

`currentProject` | `openProject`

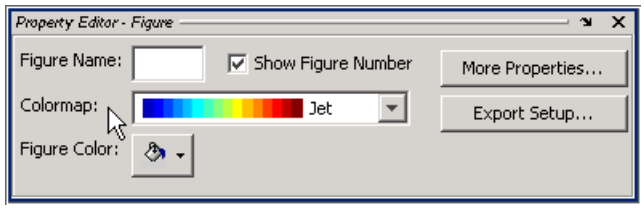
Topics

"Analyze Project Dependencies"
"Create and Edit Projects Programmatically"

Introduced in R2019a

propedit

Open Property Editor



Syntax

```
propedit
propedit(handle_list)
```

Description

`propedit` starts the Property Editor, a graphical user interface to the properties of graphics objects. If no current figure exists, `propedit` will create one.

`propedit(handle_list)` edits the properties for the object (or objects) in `handle_list`.

Starting the Property Editor enables plot editing mode for the figure.

See Also

Property Inspector | `plotedit` | `propertyeditor`

Introduced before R2006a

properties

Class property names

Syntax

```
properties(ClassName)
properties(obj)
p = properties( ___ )
```

Description

`properties(ClassName)` displays the names of the non-hidden, public properties for the MATLAB class, including inherited properties.

`properties(obj)` displays the names of the public properties for the object or object array. When `obj` is scalar, `properties` also returns dynamic properties. When `obj` is an array, `properties` returns the properties of the class of the array.

`p = properties( ___ )` returns the property names in a cell array of character vectors.

Examples

List Properties Using Class Name

List the properties of a class using the class name.

```
properties memmapfile
```

```
Properties for class memmapfile:

    Filename
    Writable
    Offset
    Format
```

```
    Repeat
    Data
```

List Properties of Object

List the public properties of an MException object.

```
me = MException('Msg:ID','MsgText');
p = properties(me)
```

```
p = 5x1 cell array
    {'identifier'}
    {'message'   }
    {'cause'     }
    {'stack'     }
    {'Correction'}
```

Input Arguments

`ClassName` — Name of the class
character vector | string scalar

Name of the class whose properties you want to query, specified as a character vector or string scalar.

Data Types: `char` | `string`

`obj` — MATLAB object
object

Object of the class whose properties you want to query, specified as an object or array of objects.

Output Arguments

`p` — Property names
cell array

Property names returned as a cell array of character vector.

More About

Public Properties

A property is public when its `GetAccess` attribute value is `public` and its `Hidden` attribute value is `false` (default values for these attributes). For a complete list of attributes, see “Property Attributes”.

Properties Keyword

The word `properties` is also a MATLAB class-definition keyword. See `classdef` for more information on class definition keywords.

Extended Capabilities

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`events` | `fieldnames` | `methods`

Topics

“Properties”
“Dynamic Properties — Adding Properties to an Instance”

Introduced in R2008a

propertyeditor

Show or hide **Property Editor**

Syntax

```
propertyeditor('on')
propertyeditor('off')
propertyeditor('toggle')
propertyeditor
propertyeditor(figure_handle,...)
```

Description

`propertyeditor('on')` displays the Property Editor tool on the current figure.

`propertyeditor('off')` hides the Property Editor on the current figure.

`propertyeditor('toggle')` toggles the visibility of the Property Editor on the current figure.

`propertyeditor` is the same as `propertyeditor('on')`.

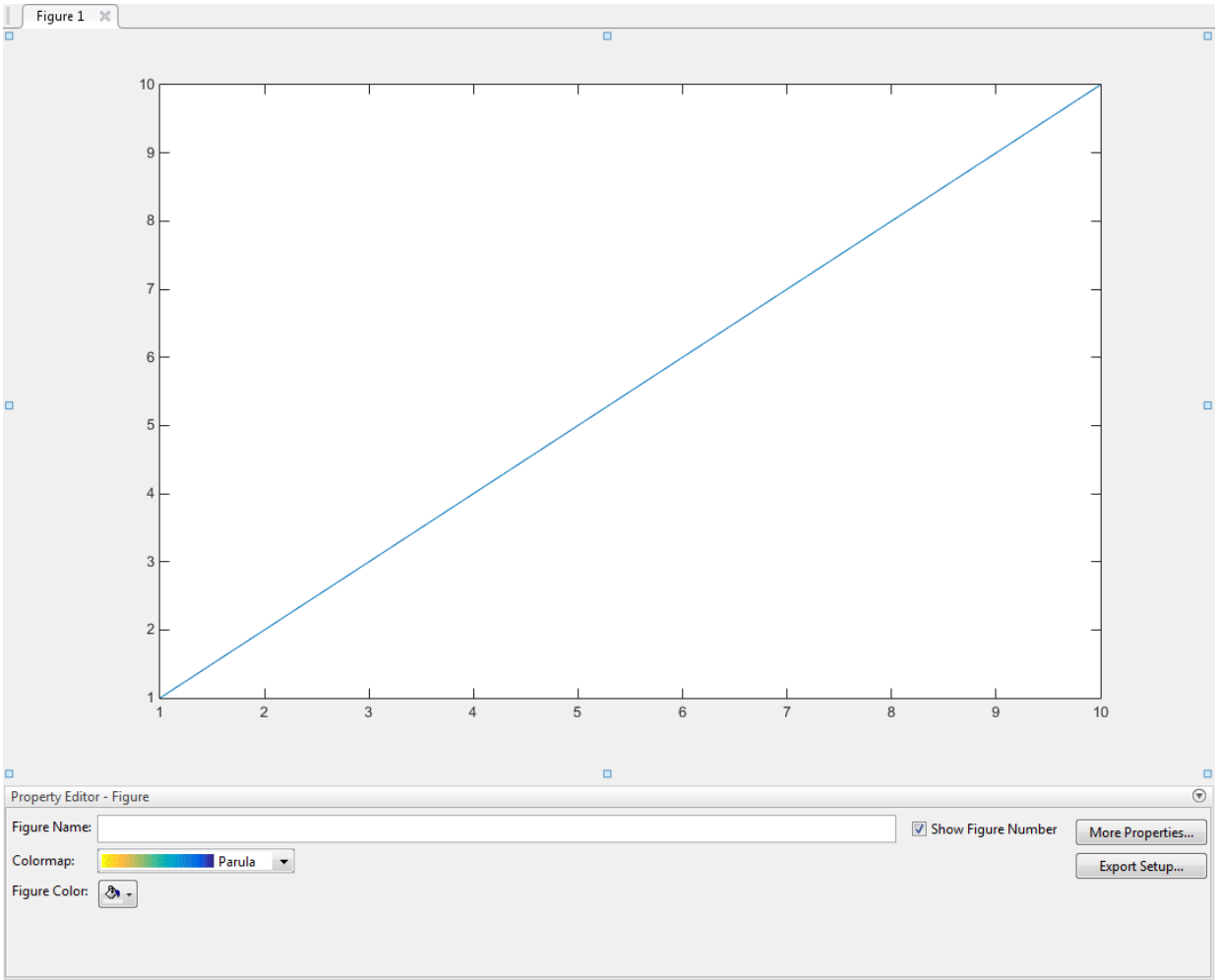
`propertyeditor(figure_handle,...)` displays or hides the Property Editor on the figure specified by `figure_handle`.

Examples

Open Property Editor

Create a simple plot and open the property editor.

```
plot(1:10)
propertyeditor
```



Tips

If you call `propertyeditor` in a MATLAB program and subsequent lines depend on the Property Editor being fully initialized, follow it by `drawnow` to ensure complete initialization.

Alternatives

Open or close the **Property Editor** tool from the figure's **View** menu.

See Also

Property Inspector | `figurepalette` | `plotbrowser` | `plottools`

Introduced before R2006a

matlab.mixin.util.PropertyGroup class

Package: `matlab.mixin.util`

Custom property list for object display

Description

Use the `PropertyGroup` class to create custom property display lists for class derived from `matlab.mixin.CustomDisplay`. You can change the order of properties displayed and specify which properties to display.

Construction

`P = matlab.mixin.util.PropertyGroup(propertyList)` constructs a property group object with the supplied `propertyList`.

`P = matlab.mixin.util.PropertyGroup(propertyList,title)` displays `title` above the list of properties.

Input Arguments

`propertyList`

The `propertyList` is either a cell array of character vector property names, a string array of property names, or a scalar `struct` with property name-value pairs. Empty character vectors and empty strings are valid values. Missing strings are not valid.

`title`

Text to display above properties, specified as a character vector or a string scalar. Empty character vectors and empty strings are valid values. Missing strings are not valid.

Properties

NumProperties

The number of properties in the `PropertyList`.

Attributes:

Dependent	true
GetAccess	public
GetObservable	true
SetAccess	private
Transient	true

PropertyList

The list of properties to display, stored as a scalar `struct` or a cell array of `char` vectors.

Attributes:

GetAccess	public
GetObservable	true
SetAccess	public
SetObservable	true

Title

An optional title for the `PropertyGroup`.

Attributes:

GetAccess	public
GetObservable	true
SetAccess	public
SetObservable	true

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

See Also

`matlab.mixin.CustomDisplay`

Topics

"Customize Property Display"
"Custom Display Interface"
Property Attributes

Introduced in R2013b

psi

Psi (polygamma) function

Syntax

```
Y = psi(X)
Y = psi(k,X)
```

Description

`Y = psi(X)` evaluates the ψ function for each element of array `X`. `X` must be real and nonnegative. The ψ function, also known as the digamma function, is the logarithmic derivative of the gamma function

$$\begin{aligned}\psi(x) &= \text{digamma}(x)\\ &= \frac{d(\log(\Gamma(x)))}{dx}\\ &= \frac{d(\Gamma(x))/dx}{\Gamma(x)}\end{aligned}$$

`Y = psi(k,X)` evaluates the `k`th derivative of ψ at the elements of `X`. `psi(0,X)` is the digamma function, `psi(1,X)` is the trigamma function, `psi(2,X)` is the tetragamma function, etc.

Examples

Example 1

Use the `psi` function to calculate Euler's constant, γ.

```
format long
-psi(1)
ans =
   0.57721566490153
```

```
-psi(0,1)
ans =
   0.57721566490153
```

Example 2

The trigamma function of 2, `psi(1,2)`, is the same as $(\pi^2/6) - 1$.

```
format long
psi(1,2)
ans =
   0.64493406684823

pi^2/6 - 1
ans =
   0.64493406684823
```

References

[1] Abramowitz, M. and I. A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, 1965, Sections 6.3 and 6.4.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitation:

For the syntax `Y = psi(k,X)`, `k` must be a non-tall scalar.

For more information, see “Tall Arrays for Out-of-Memory Data”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Strict single-precision calculations are not supported. In the generated code, single-precision inputs produce single-precision outputs. However, variables inside the function might be double-precision.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`gamma` | `gammainc` | `gammaln`

Introduced before R2006a

publish

Generate view of MATLAB file in specified format

Syntax

```
publish(file)
publish(file,format)

publish(file,Name,Value)
publish(file,options)

my_doc = publish(file, ___ )
```

Description

`publish(file)` generates a view of the specified MATLAB code file and output in an HTML format that can be used for sharing. `publish` saves the HTML file and a file for each graphic that the code creates in a subfolder named `html`. The location of the `html` subfolder is relative to the location of `file`.

For example, `publish('C:\myMATLABfiles\myfile.m')` runs the code in `myfile.m` using the base workspace, and then saves the formatted code and results in `C:\myMATLABfiles\html\myfile.html`.

`publish(file,format)` generates a view of the specified MATLAB file in the specified file format. All file formats save to the `html` subfolder.

`publish(file,Name,Value)` generates a view of the specified MATLAB file with options specified by one or more `name,value` pair arguments.

`publish(file,options)` uses the `options` structure to generate the view of the specified MATLAB file. Using a structure to specify options is useful when you want to preconfigure and save your options for repeated use. The fields and values of the `options` structure correspond to names and values of name-value pair arguments.

`my_doc = publish(file, ___ )` generates a view of the specified MATLAB file and returns the path of the resulting output file. You can use this syntax with any of the input argument combinations in the previous syntaxes.

Examples

Generate HTML View of MATLAB Script

Generate an HTML view of a MATLAB script including the code, results, and comments.

Copy the example file `fourier_demo2.m` to your current folder.

```
filename = fullfile(matlabroot,'help','techdoc','matlab_env','examples','fourier_demo2.m');
copyfile(filename,'.','f')
```

Use the `publish` function to run the example and generate an HTML view of the example. Then, use the `web` function to view the resulting file.

```
publish('fourier_demo2.m');
web('html/fourier_demo2.html')
```

Generate View of MATLAB Script in Microsoft Word Format

Generate a Microsoft Word view of a MATLAB script including the code, results, and comments.

Copy the example file `fourier_demo2.m` to your current folder.

```
filename = fullfile(matlabroot,'help','techdoc','matlab_env','examples','fourier_demo2.m');
copyfile(filename,'.','f')
```

Use the `publish` function to generate a Microsoft Word view of the example. Then, use the `winopen` function to view the resulting file.

```
publish('fourier_demo2.m','doc');
winopen('html/fourier_demo2.doc')
```

Customize View Using Name-Value Pairs

Use name-value pair arguments to change the appearance of published figure windows in the HTML view of a MATLAB script.

Copy the example file `fourier_demo2.m` to your current folder.

```
filename = fullfile(matlabroot,'help','techdoc','matlab_env','examples','fourier_demo2.m');
copyfile(filename,'.','f')
```

Use the `publish` function to generate an HTML view of the example. Specify `figureSnapMethod` as `entireFigureWindow` to include window decorations and to match the figure background color to the screen color for figures. Then, use the `web` function to view the resulting file.

```
publish('fourier_demo2.m','figureSnapMethod','entireFigureWindow')
web('html/fourier_demo2.html')
```

Customize View Using Structure

Use a structure to create a Microsoft Word view of a MATLAB script that does not show the code from the script. Specifying options as a structure is useful when you want to preconfigure and save your options for repeated use.

Copy the example file `fourier_demo2.m` to your current folder.

```
filename = fullfile(matlabroot,'help','techdoc','matlab_env','examples','fourier_demo2.m');
copyfile(filename,'.','f')
```

Create a structure `options_doc_nocode` that specifies the `format` and `showcode` options.

```
options_doc_nocode.format = 'doc';
options_doc_nocode.showCode = false;
```

Use the `publish` function to generate a Microsoft Word view of the example using the options specified in `options_doc_nocode`. Then, use the `winopen` function to view the resulting file.

```
publish('fourier_demo2.m',options_doc_nocode);
winopen('html/fourier_demo2.doc')
```

Save File Path of Published Script to Variable

Generate an HTML view of a MATLAB script, and save the path of the published HTML file to a variable.

Copy the example file `fourier_demo2.m` to your current folder.

```
filename = fullfile(matlabroot,'help','techdoc','matlab_env','examples','fourier_demo2.m');
copyfile(filename,'.','f')
```

Use the `publish` function to generate an HTML view of the example, and save the path of the published HTML file to the variable `mydoc`.

```
mydoc = publish('fourier_demo2.m')
```

```
mydoc =

C:\myMATLABfiles\html\fourier_demo2.html
```

Input Arguments

`file` — MATLAB file name
character vector | string

MATLAB file name, specified as a character vector or string. `file` can include a full or partial path.

Note When MATLAB publishes a file, it can overwrite existing files from the output folder that start with the same name as `file`.

Example: `'myfile.m'`

Example: `'C:\myMATLABfiles\myfile.m'`

`format` — Output format
`'html'` (default) | `'doc'` | `'latex'` | `'ppt'` | `'xml'` | `'pdf'`

Output format of published file, specified as one of the values listed in the table.

Value	Output Format
`'html'` (default)	Hypertext Markup Language
`'doc'`	Microsoft Word
`'latex'`	LaTeX
`'ppt'`	Microsoft PowerPoint
`'xml'`	Extensible Markup Language
`'pdf'`	Portable Document Format (PDF)

The Microsoft Word and Microsoft PowerPoint formats are only available on Windows platforms.

MATLAB does not preserve syntax highlighting when you specify the output format as Microsoft PowerPoint or LaTeX.

Example: `publish('myfile.m','ppt');`

`options` — Options for published output
structure

Options for published output, specified as a structure. Use the `options` structure instead of name-value pair arguments when you want to reuse the same configuration for publishing multiple MATLAB files.

The fields and values of the `options` structure correspond to names and values of the name-value pair arguments.

For example, this command creates the structure `options`, and specifies the PDF output format and the output folder `C:\myPublishedOutput`.

```
options = struct('format','pdf','outputDir','C:\myPublishedOutput')
```

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'format','pdf','showCode',false` specifies the PDF output file format and excludes the code from the output.

Output Options

`format` — Output format
`'html'` (default) | `'doc'` | `'latex'` | `'ppt'` | `'xml'` | `'pdf'`

Output format of published file, specified as one of the values listed in the table.

Value	Output Format
`'html'` (default)	Hypertext Markup Language (HTML)
`'doc'`	Microsoft Word
`'latex'`	LaTeX
`'ppt'`	Microsoft PowerPoint
`'xml'`	Extensible Markup Language
`'pdf'`	Portable Document Format (PDF)

The Microsoft Word and Microsoft PowerPoint formats are only available on Windows platforms.

MATLAB does not preserve syntax highlighting when you specify the output format as Microsoft PowerPoint or LaTeX.

`outputDir` — Output folder
character vector

Output folder to which the published document is saved, specified as a character vector. Specify the full path of the folder.

Example: `'C:\myPublishedOutput'`

`stylesheet` — Extensible Stylesheet Language (XSL) file
character vector

Extensible Stylesheet Language (XSL) file to use when publishing a MATLAB file to HTML, XML, or LaTeX format, specified as a character vector. Specify the full path of the XSL file.

Example: `'C:\myStylesheet\stylesheet.xsl'`

Figure Options

`createThumbnail` — Whether to create thumbnail image
`true` (default) | `false`

Whether to create a thumbnail image for the published document, specified as `true` or `false`. You can use the generated thumbnail to display a small representation of your file on HTML pages.

`figureSnapMethod` — Figure window appearance
`'entireGUIWindow'` (default) | `'print'` | `'getframe'` | `'entireFigureWindow'`

Figure window appearance for the published document, specified as one of the values listed in the table. Figure window appearance includes the background color of the plot and whether it includes window decorations (title bar, toolbar, menu bar, and window border).

This option is not available in MATLAB Online.

Value	Window Decorations		Background Color	
	GUIs	Figures	GUIs	Figures
`'entireGUIWindow'` (default)	Included	Excluded	Matches screen	White
`'print'`	Excluded	Excluded	White	White
`'getframe'`	Excluded	Excluded	Matches screen	Matches screen
`'entireFigureWindow'`	Included	Included	Matches screen	Matches screen

`imageFormat` — Image file format
`'png'` | `'epsc2'` | `'jpg'` | ...

Image file format for the images in the published document, specified as one of the values listed in the table. The list of valid image formats depends on the specified output format.

Output Format	Valid Image Formats	Default Image Format
Microsoft Word	Any image format that your installed version of Microsoft Office can import, including `'png'` , `'jpg'`, `'bmp'`, and `'tiff'`. If the `'figureSnapMethod'` option is set to `'print'`, then you also can specify `'eps'`, `'epsc'`, `'eps2'`, `'ill'`, `'meta'`, and `'pdf'`.	`'png'`
Hypertext Markup Language (HTML)	All image formats. Ensure that the tools you use to view and process the output files can display the output format you specify.	`'png'`
LaTeX	All image formats. Ensure that the tools you use to view and process the output files can display the output format you specify.	`'epsc2'` The default changes to `'png'` in these cases: • `figureSnapMethod` is `'getframe'`. • `figureSnapMethod` is `'entireFigureWindow'`. • `figureSnapMethod` is `'entireGUIWindow'` and the snapped window is a GUI window.
Portable Document Format (PDF)	`'bmp'` and `'jpg'`.	`'bmp'`
Microsoft PowerPoint	Any format that your installed version of Microsoft Office can import, including `'png'`, `'jpg'`, `'bmp'`, and `'tiff'`.	`'png'`

Output Format	Valid Image Formats	Default Image Format
Extensible Markup Language (XML)	All image formats. Ensure that the tools you use to view and process the output files can display the image format you specify.	`'png'`

`maxHeight` — Maximum image height

`[]` (default) | positive integer

Maximum image height of the published images, specified as one of these values:

- `[]` (default) — Unrestricted height. This value is always used when the output format is PDF.
- Positive integer — Height in pixels.

`maxWidth` — Maximum image width

`[]` (default) | positive integer

Maximum image width of the published images, specified as one of these values:

- `[]` (default) — Unrestricted width. This value is always used when the output format is PDF.
- Positive integer — Width in pixels.

`useNewFigure` — Whether to create new figure

`true` (default) | `false`

Whether to create new figure, specified as `true` or `false`. If `true` and the code generates a figure, then MATLAB creates a new figure window in the default size with a white background before publishing. If `false`, MATLAB does not create a figure window.

Specifying a value of `false` is useful if you want to use a figure with different properties for publishing. For example, you can open a figure window, change the size and background color, and then publish your code. Figures in your published document use the characteristics of the figure you opened before publishing.

Code Options

`evalCode` — Whether to run code

`true` (default) | `false`

Whether to run the code and include the MATLAB output in the published view, specified as `true` or `false`.

`catchError` — Whether to catch errors
`true` (default) | `false`

Whether to catch errors during publishing, specified as `true` or `false`. If `true` and an error occurs, MATLAB continues publishing and includes the error in the published file. If `false` and an error occurs, MATLAB displays the error at the command line and does not produce a published file.

`codeToEvaluate` — Additional code to run
character vector

Additional code to run during publishing, specified as a character vector. Use this option to run code that is not included in the MATLAB file. For example, you can set the value of an input argument for a function being published.

If this option is unspecified, MATLAB only runs the code in the MATLAB file you are publishing.

`maxOutputLines` — Maximum number of lines
`Inf` (default) | nonnegative integer value

Maximum number of lines of output to be included in the published document, specified as one of these values:

- `Inf` (default) — MATLAB includes all output in the published document.
- Nonnegative integer — MATLAB only includes the specified number of lines in the published document. Setting this option is useful if a smaller, representative sample of the output suffices.

`showCode` — Whether to include code
`true` (default) | `false`

Whether to include code in published file, specified as `true` or `false`.

If the output format is HTML, MATLAB includes the code at the end of the published HTML file as comments, even when you set the `'showCode'` option to `false`. Including the code as comments enables the `grabcode` function to extract the MATLAB code from an HTML file, even when the file does not display the code. The code does not display in a Web browser because MATLAB includes the code as comments.

Limitations

- In MATLAB Online, you cannot use the `publish` function on a file in your `Shared` folder. To publish a shared file, open the file in the Editor, and then click **Publish** on the toolbar. When you use the **Publish** button in MATLAB Online instead of the `publish` function, MATLAB stores the results in the `Published` subfolder of your root folder instead of the `html` subfolder.

Tips

- To enhance the readability of the published document and include additional image snapshots, external file content, and external images, see “Publishing Markup”.

See Also

`grabcode` | `snapnow` | `web` | `winopen`

Topics

“Publish and Share MATLAB Code”
“Publishing Markup”

Introduced before R2006a

PutCharArray

Character array in Automation server

Syntax

IDL Method Signature

```
PutCharArray([in] BSTR varname, [in] BSTR workspace,
    [in] BSTR string)
```

Microsoft Visual Basic Client

```
PutCharArray(varname As String, workspace As String,
    string As String)
```

MATLAB Client

`PutCharArray(h,'varname','workspace','string')`

Description

`PutCharArray(h,'varname','workspace','string')` stores the character array in `string` in the specified `workspace` of the server attached to handle `h`, assigning to it the variable `varname`. The values for *`workspace`* are `base` or `global`. The function name is case-sensitive.

Examples

Pass Character Data from Visual Basic® .NET Client to MATLAB®

This example shows how to pass character data from a Visual Basic® .NET client to MATLAB®. Create an application with the following code.

```
type putchararray.vb
```

```
Dim Matlab As Object
Try
    Matlab = GetObject(, "matlab.application")
Catch e As Exception
    Matlab = CreateObject("matlab.application")
End Try
MsgBox("MATLAB window created; now open it...")
Matlab.PutCharArray("str", "base", _
  "He jests at scars that never felt a wound.")
MsgBox("In MATLAB, type" & vbCrLf & "str")
MsgBox("closing MATLAB window...")
Matlab.Quit()
```

Open the MATLAB window, then click **Ok**.

In the MATLAB window type `str`; MATLAB displays:

```
str =

He jests at scars that never felt a wound.
```

Click **Ok** to close and terminate MATLAB.

Pass Character Data from VBA Client to MATLAB®

This example shows how to pass character data from a Visual Basic® .NET client to MATLAB®. Create an application with the following code.

```
type putchararray.vba
```

```
Dim Matlab As Object
Set Matlab = CreateObject("matlab.application")
MsgBox ("MATLAB window created; now open it...")
x = Matlab.PutCharArray("str", "base", "He jests at scars that never felt a wound.")
MsgBox ("In MATLAB, type" & vbCrLf & "str")
MsgBox ("closing MATLAB window...")
y = Matlab.Quit()
```

Open the MATLAB window, then click **Ok**.

In the MATLAB window type `str`; MATLAB displays:

```
str =

He jests at scars that never felt a wound.
```

Click **Ok** to close and terminate MATLAB.

Tips

- The character array specified in the `string` argument can have any dimensions. However, `PutCharArray` changes the dimensions to a 1-by-n column-wise representation, where n is the number of characters in the array. Executing the following commands in MATLAB illustrates this behavior:

  ```
  h = actxserver('matlab.application');
  chArr = ['abc'; 'def'; 'ghk']
  chArr =
  abc
  def
  ghk

  PutCharArray(h,'Foo','base',chArr)
  tstArr = GetCharArray(h,'Foo','base')
  tstArr =
  adgbehcfk
  ```

See Also

`Execute` | `GetCharArray` | `GetWorkspaceData` | `PutWorkspaceData`

Topics

"Conversion of COM Types to MATLAB Types"
"Conversion of MATLAB Types to COM Types"

Introduced before R2006a

PutFullMatrix

Matrix in Automation server workspace

Syntax

IDL Method Signature

```
PutFullMatrix([in] BSTR varname, [in] BSTR workspace,
[in] SAFEARRAY(double) xreal, [in] SAFEARRAY(double) ximag)
```

Microsoft Visual Basic Client

```
PutFullMatrix([in] varname As String, [in] workspace As String,
[in] xreal As Double, [in] ximag As Double)
```

MATLAB Client

`PutFullMatrix(h,'varname','workspace',xreal,ximag)`

Description

`PutFullMatrix(h,'varname','workspace',xreal,ximag)` stores a matrix in the specified *workspace* of the server attached to handle `h` and assigns it to variable `varname`. Use `xreal` and `ximag` for the real and imaginary parts of the matrix. The values for *workspace* are `base` or `global`.

The matrix cannot be a scalar, an empty array, or have more than two dimensions. To use higher dimensional matrices, reshape the matrix to a 2-D matrix before sending it to the MATLAB server. Then change the dimensions back after receiving it from MATLAB.

For VBScript clients, use the `GetWorkspaceData` and `PutWorkspaceData` functions to pass numeric data to and from the MATLAB workspace. These functions use the `variant` data type instead of `safearray`, which VBScript does not support.

Examples

Put Matrix into MATLAB® Base Workspace from Visual Basic® .NET Client

This example uses a Visual Basic® .NET client to write a matrix to the base workspace of the MATLAB server.

```
type putfullmatrixbase.vb
```

```
Dim MatLab As Object
Dim XReal(4, 4) As Double
Dim XImag(4, 4) As Double
Dim ZReal(4, 4) As Double
Dim ZImag(4, 4) As Double
Dim i, j As Integer

For i = 0 To 4
   For j = 0 To 4
      XReal(i, j) = Rnd() * 6
      XImag(i, j) = 0
   Next j
Next i

Matlab = CreateObject("matlab.application")
MatLab.PutFullMatrix("M","base",XReal,XImag)
MatLab.GetFullMatrix("M","base",ZReal,ZImag)
```

Put Matrix into MATLAB® Global Workspace from Visual Basic® .NET Client

This example uses a Visual Basic® .NET client to write a matrix to the global workspace of the MATLAB server.

```
type putfullmatrixglobal.vb
```

```
Dim MatLab As Object
Dim XReal(1,2) As Double
Dim XImag(1,2) As Double
Dim result As String
Dim i,j As Integer
```

```
For i = 0 To 1
   For j = 0 To 2
      XReal(i,j) = (j * 2 + 1) + i
      XImag(i,j) = 1
   Next j
Next i

Matlab = CreateObject("matlab.application")
MatLab.PutFullMatrix("X","global",XReal,XImag)
result = Matlab.Execute("whos global")
MsgBox(result)
```

Put Matrix into MATLAB® Base Workspace from VBA Client

This example uses a VBA client to write a matrix to the base workspace of the MATLAB server.

```
type putfullmatrixbase.vba
```

```
Dim MatLab As Object
Dim XReal(4, 4) As Double
Dim XImag(4, 4) As Double
Dim ZReal(4, 4) As Double
Dim ZImag(4, 4) As Double
Dim i, j As Integer

For i = 0 To 4
    For j = 0 To 4
        XReal(i, j) = Rnd() * 6
        XImag(i, j) = 0
    Next j
Next i

Set MatLab = CreateObject("matlab.application")
x = MatLab.PutFullMatrix("M", "base", XReal, XImag)
y = MatLab.GetFullMatrix("M", "base", ZReal, ZImag)
```

Put Matrix into MATLAB® Global Workspace from VBA Client

This example uses a VBA client to write a matrix to the global workspace of the MATLAB server.

```
type putfullmatrixglobal.vba
```

```
Dim MatLab As Object
Dim XReal(1, 2) As Double
Dim XImag(1, 2) As Double
Dim result As String
Dim i, j As Integer

For i = 0 To 1
    For j = 0 To 2
        XReal(i, j) = (j * 2 + 1) + i
        XImag(i, j) = 1
    Next j
Next i

Set MatLab = CreateObject("matlab.application")
x = MatLab.PutFullMatrix("X", "global", XReal, XImag)
result = MatLab.Execute("whos global")
MsgBox (result)
```

See Also

`Execute` | `GetFullMatrix` | `PutWorkspaceData`

Topics

"Conversion of COM Types to MATLAB Types"
"Conversion of MATLAB Types to COM Types"

Introduced before R2006a

PutWorkspaceData

Data in Automation server workspace

Syntax

IDL Method Signature

```
PutWorkspaceData([in] BSTR varname, [in] BSTR workspace,
[in] VARIANT data)
```

Microsoft Visual Basic Client

```
PutWorkspaceData(varname As String, workspace As String,
data As Object)
```

MATLAB Client

`PutWorkspaceData(h,'varname','workspace',data)`

Description

`PutWorkspaceData(h,'varname','workspace',data)` stores `data` in the *`workspace`* of the server attached to handle `h` and assigns it to `varname`. The values for *`workspace`* are `base` or `global`.

Use `PutWorkspaceData` to pass numeric and character array data respectively to the server. Do *not* use `PutWorkspaceData` on sparse arrays, structures, or function handles. Use the `Execute` method for these data types.

The `GetWorkspaceData` and `PutWorkspaceData` functions pass numeric data as a `variant` data type. These functions are especially useful for VBScript clients as VBScript does not support the `safearray` data type used by `GetFullMatrix` and `PutFullMatrix`.

Examples

Put Visual Basic® .NET Array Into MATLAB® Workspace

This example creates an array in a Visual Basic .NET application and displays it in MATLAB.

Create a Visual Basic .NET client with the following code.

```
type putworkspacedata.vb
```

```
Dim Matlab As Object
Dim data(6) As Double
Dim i As Integer
Matlab = CreateObject("matlab.application")
For i = 0 To 6
    data(i) = i * 15
Next i
Matlab.PutWorkspaceData("A", "base", data)
MsgBox("In MATLAB, type" & vbCrLf & "A")
```

Put VBA Array Into MATLAB® Workspace

This example creates an array in a VBA application and displays it in MATLAB.

Create a VBA client with the following code.

```
type putworkspacedata.vba
```

```
Dim Matlab As Object
Dim data(6) As Double
Dim i As Integer
Set Matlab = CreateObject("matlab.application")
For i = 0 To 6
    data(i) = i * 15
Next i
Matlab.PutWorkspaceData("A", "base", data)
MsgBox("In MATLAB, type" & vbCrLf & "A")
```

See Also

`Execute` | `GetWorkspaceData` | `PutCharArray` | `PutFullMatrix`

Topics

"Conversion of COM Types to MATLAB Types"
"Conversion of MATLAB Types to COM Types"

Introduced before R2006a

pwd

Identify current folder

Syntax

```
pwd
currentFolder = pwd
```

Description

`pwd` displays the MATLAB current folder.

`currentFolder = pwd` returns the path to the current folder.

Examples

Store Path to MATLAB® Current Folder

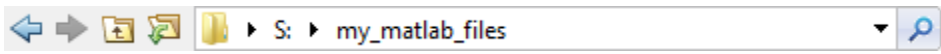
Change the current folder to a local folder and store the path.

```
cd c:\myMATLABFiles
currentFolder = pwd
```

```
currentFolder =

    'c:\myMATLABFiles'
```

Alternative Functionality

View the current folder in the Current Folder toolbar.



Output Arguments

`currentFolder` — MATLAB current folder path
character vector

MATLAB current folder path, returned as a character vector.

See Also

`cd` | `dir`

Introduced before R2006a

pyargs

Create keyword arguments for Python function

Syntax

```
kwa = pyargs(argKey,argValue)
```

Description

`kwa = pyargs(argKey,argValue)` creates one or more keyword arguments to pass to a Python function. A Python keyword argument is a value preceded by an identifier. The `pyargs` function is the only way to create keyword arguments in MATLAB.

Examples

Use Keyword Argument to Modify Calendar Month Display

The Python `calendar.TextCalendar.formatmonth` method displays a monthly calendar. The method takes two keyword arguments. The keyword `w` specifies the width of the date columns. The keyword `l` specifies the number of lines for each week.

Create a calendar.

```
cal = py.calendar.TextCalendar;
```

Display the calendar for December 2014 using the default line spacing and column width.

```
formatmonth(cal,int32(2014),int32(12))
```

```
ans =

  Python str with no properties.

       December 2014
Mo Tu We Th Fr Sa Su
```

```
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

Display the function signature from the Python documentation for the `calendar.TextCalendar.formatmonth` function.

```
py.help('calendar.TextCalendar.formatmonth')
```

```
Help on method formatmonth in calendar.TextCalendar:

calendar.TextCalendar.formatmonth = formatmonth(self, theyear, themonth, w=0, l=0) unbound
calendar.TextCalendar method
    Return a month's calendar string (multi-line).
```

Notice that arguments `w` and `l` are optional, with default values of `0`.

Now display the function signatures in MATLAB.

```
methods(cal,'-full')
```

Search the output for the `formatmonth` function signature.

```
lhs formatmonth(self, theyear, themonth, w, l, pyargs)
```

Change the value of the line spacing argument to 2 and the value of the width argument to 3 using a `pyargs` argument.

```
formatmonth(cal,int32(2014),int32(12),pyargs('l',int32(2),'w',int32(3)))
```

```
ans =

  Python str with no properties.

           December 2014

    Mon Tue Wed Thu Fri Sat Sun

      1   2   3   4   5   6   7

      8   9  10  11  12  13  14

     15  16  17  18  19  20  21

     22  23  24  25  26  27  28
```

```
29  30  31
```

Input Arguments

`argKey,argValue` — Python function arguments
keyword and value arguments

Python function keyword arguments specified as one or more comma-separated pairs of `argKey,argValue` arguments. `argKey` is the Python function key name and is a string or character vector. `argValue` is the argument value, represented by any valid Python type. Use the function argument list to identify `argKey` and `argValue`. You can specify several key and value pair arguments in any order as `argKey1,argValue1,...,argKeyN,argValueN`.

Example: `'length',int32(2)`

See Also

Topics
"Pass Keyword Arguments"
"Python Function Arguments"

External Websites
`python.org calendar.TextCalendar method`

Introduced in R2014b

matlab.exception.PyException class

Package: `matlab.exception`

Capture error information for Python exception

Description

Process information from a `matlab.exception.PyException` object to handle Python errors thrown from Python methods called from MATLAB. This class is derived from `MException`.

Creation

You do not construct a `matlab.exception.PyException` object explicitly. MATLAB automatically constructs a `PyException` object whenever Python throws an exception. The `PyException` object wraps the original Python exception.

Properties

`ExceptionObject` — Object
exception object

Result from Python `sys.exc_info` function. For information about what the function returns, type:

```
help('py.sys.exc_info')
```

```
exc_info() -> (type, value, traceback)

Return information about the most recent exception caught by an except
clause in the current stack frame or in an older stack frame.
```

Examples

Catch Python Exception

Generate a Python exception and display information. When MATLAB displays a message containing the text `Python Error`, refer to your Python documentation for more information.

```
try
  py.list('x','y',1)
catch e
  e.message
  if(isa(e,'matlab.exception.PyException'))
    e.ExceptionObject
  end
end
```

```
ans =

Python Error: TypeError: list() takes at most 1 argument (3 given)

ans =

  Python tuple with no properties.

    (<type 'exceptions.TypeError'>, TypeError('list() takes at most 1 argument (3 given)',), None)
```

See Also

Topics

"Capture Information About Exceptions"
"Throw an Exception"

Introduced in R2014b

pyenv

Change default environment of Python interpreter

Syntax

```
pe = pyenv

pe = pyenv('Version',version)
pe = pyenv('Version',executable)

pe = pyenv('ExecutionMode',executionMode)
pe = pyenv('Version',executable,'ExecutionMode',executionMode)
```

Description

Use `pyenv` to change the default version or execution mode of the Python interpreter. These changes are persistent across different MATLAB sessions.

`pe = pyenv` returns details about the current (default) Python environment as a `PythonEnvironment` object.

`pe = pyenv('Version',version)` changes the default Python version on Microsoft Windows platforms.

Note MATLAB automatically loads Python when you type `py.command`. You cannot change the interpreter after MATLAB loads Python. To change the interpreter, restart MATLAB, and then call `pyenv`.

`pe = pyenv('Version',executable)` specifies the full path to the Python executable. You can use this syntax on any platform or for repackaged CPython implementation downloads.

`pe = pyenv('ExecutionMode',executionMode)` changes the default execution mode of the Python interpreter.

`pe = pyenv('Version',executable,'ExecutionMode',executionMode)` changes the default version and execution mode of the interpreter.

Examples

Display Python Version for Your System

```
pe = pyenv;
pe.Version
```

```
ans =

    "2.7"
```

Create `list` Variable Out-of-Process

Set the execution mode to `OutOfProcess`.

```
pyenv("ExecutionMode","OutOfProcess")
```

```
ans =
  PythonEnvironment with properties:

          Version: "2.7"
       Executable: "C:\Python27\pythonw.exe"
          Library: "C:\windows\system32\python27.dll"
             Home: "C:\Python27"
           Status: NotLoaded
    ExecutionMode: OutOfProcess
```

Create the variable.

```
py.list({'Monday','Tuesday','Wednesday','Thursday','Friday'});
```

Show the process. MATLAB displays information specific to your environment.

```
pyenv
```

```
ans =
  PythonEnvironment with properties:
```

```
          Version: "2.7"
       Executable: "C:\Python27\pythonw.exe"
          Library: "C:\windows\system32\python27.dll"
             Home: "C:\Python27"
           Status: Loaded
    ExecutionMode: OutOfProcess
        ProcessID: "8196"
      ProcessName: "MATLABPyHost"
```

Use Python Version 2.7

```
pe = pyenv;
if pe.Status == 'Loaded'
    disp('To change the Python version, restart MATLAB, then call pyenv('Version','2.7
else
    pyenv('Version','2.7');
end
```

Input Arguments

`version` — Python version
string | character vector

Python version number, specified as a string or character vector (Windows platform only). The version must contain the major and minor version numbers separated by a period. For information about supported versions, see “System and Configuration Requirements”.

`pyenv` looks for the version in the Windows registry. If you download the Python application from `www.python.org/downloads`, the installation automatically adds the version to the registry. If you download the application from a different source, you must either add it to the registry or use the `pyenv(executable)` syntax to change the version.

Example: `2.7`

Data Types: `char` | `string`

`executable` — Name of Python executable file
string | character vector

Name of an existing Python executable file, specified as a string or character vector. This argument must contain the name of the Python executable file, and it can contain the full path.

Example: `/usr/bin/python`

Data Types: `char` | `string`

`executionMode` — Execution mode
`'InProcess'` (default) | `'OutOfProcess'`

Execution mode indicating whether to run Python scripts in the same process as MATLAB, specified as `'InProcess'` or `'OutOfProcess'`. The default `'InProcess'` runs the scripts in the MATLAB process and is suggested for performance-critical use cases.

`'OutOfProcess'` starts a separate process and is used for safe execution of Python scripts and libraries. Select `'OutOfProcess'` for:

- Working with Python libraries which require a different version of a 3rd party library also required by MATLAB
- Debugging workflows

When you call a Python function out-of-process, there is overhead associated with the call. This behavior might affect performance.

See Also

`PythonEnvironment`

Topics

"Out-of-Process Execution of Python Functionality"

Introduced in R2019b

PythonEnvironment

Python environment information

Description

`PythonEnvironment` objects contain information about the settings and status of the Python interpreter. MATLAB communicates with the interpreter when you call a `py.` command. Python environment information is persistent across different MATLAB sessions.

Creation

To create a `PythonEnvironment` object or change the settings, call `pyenv`. The Python interpreter is loaded when you call a Python command in the `py` package. You cannot change the interpreter after MATLAB loads Python. To change the interpreter, restart MATLAB, and then call `pyenv`.

Properties

`Version` — Python version
string

Python version, specified as a read-only string. To set the `Version` property, call `pyenv` with the `'Version'` argument set to a MATLAB-supported version number. For information about supported versions, see “System and Configuration Requirements”"".

Example: `"2.7"`

Data Types: `string`

`Executable` — Name of Python executable file
string

Name of the Python executable file, specified as a read-only string. To set the `Executable` property, call `pyenv` with the `'Version'` argument set to the name of the Python executable file and, optionally, the full path.

Example: `"/usr/local/bin/python"`

Data Types: `string`

`Library` — Shared library file
string

Shared library file, specified as a read-only string.

Example: `"libpython2.7.so.1.0"`

Data Types: `string`

`Home` — Home folder
string

Home folder, specified as a read-only string.

Example: `"/usr/local/bin/../.."`

Data Types: `string`

`Status` — Process status
`NotLoaded` (default) | `Loaded` | `Terminated`

Process status, specified as `Loaded`, `NotLoaded`, or `Terminated`.

Example: `Terminated`

`ExecutionMode` — Execution mode
`InProcess` (default) | `OutOfProcess`

Execution mode indicating whether to run Python scripts in the same process as MATLAB, specified as `InProcess` or `OutOfProcess`. To set the `ExecutionMode` property, call `pyenv` with the `ExecutionMode` argument.

Example: `OutOfProcess`

`ProcessID` — Process ID
string

Process ID that the Python interpreter is running on, specified as a read-only string. If `ExecutionMode` is `InProcess`, then `ProcessID` is the MATLAB process ID. If `Status` is not `Loaded`, then `ProcessID` is empty.

Example: `"9388"`

Data Types: `string`

`ProcessName` — Process name
string

Process name in registry, specified as a read-only string.

Example: `"MATLABPyHost"`

Data Types: `string`

Object Functions

terminate Terminate process associated with Python interpreter

Examples

Display Python Environment for Your System

```
pe = pyenv
```

```
pe = 

  PythonEnvironment with properties:

         Version: "2.7"
      Executable: "C:\Python27\pythonw.exe"
         Library: "C:\windows\system32\python27.dll"
            Home: "C:\Python27"
          Status: NotLoaded
```

```
ExecutionMode: OutOfProcess
```

Limitations

- Saving (serializing) `PythonEnvironment` objects into a MAT-file is not supported.

See Also

`pyenv`

Topics

"Out-of-Process Execution of Python Functionality"

Introduced in R2019b

terminate

Terminate process associated with Python interpreter

Syntax

```
terminate(pe)
```

Description

`terminate(pe)` terminates the process associated with the Python interpreter if the ExecutionMode property is `OutOfProcess`. Throws an error if the property value is `InProcess`.

Note Pressing **Ctrl+C** does not terminate the out-of-process command.

Examples

Terminate Process

Suppose that you created this `MATLABPyHost` process:

```
pyenv
```

```
ans = 
  PythonEnvironment with properties:

          Version: "2.7"
       Executable: "C:\Python27\pythonw.exe"
          Library: "C:\windows\system32\python27.dll"
             Home: "C:\Python27"
           Status: Loaded
    ExecutionMode: OutOfProcess
        ProcessID: "8196"
      ProcessName: "MATLABPyHost"
```

Terminate the process.

```
terminate(pyenv)
```

Display information about the current environment.

```
pyenv
```

```
ans = 
  PythonEnvironment with properties:

          Version: "2.7"
       Executable: "C:\Python27\pythonw.exe"
          Library: "C:\windows\system32\python27.dll"
             Home: "C:\Python27"
           Status: Terminated
    ExecutionMode: OutOfProcess
```

Input Arguments

pe — Python environment
`PythonEnvironment`

Python environment, specified as a `PythonEnvironment` object.

See Also

`pyenv`

Introduced in R2019b

pyversion

(Not recommended) Change default version of Python interpreter

Note `pyversion` is not recommended. Use `pyenv` instead.

Syntax

```
pyversion
[version, executable, isloaded] = pyversion

___ = pyversion version
___ = pyversion executable
```

Description

`pyversion` displays details about the current Python version.

`[version, executable, isloaded] = pyversion` returns Python version information.

`___ = pyversion version` changes the default Python version on Microsoft Windows platforms. You can request any of the outputs from previous syntaxes. The setting is persistent across MATLAB sessions.

`___ = pyversion executable` specifies the full path to the Python executable. You can use this syntax on any platform or for repackaged CPython implementation downloads.

Input Arguments

`version` — Python version
string | character vector

Python version number, specified as a string or character vector. The version must contain the major and minor version numbers separated by a period. (Windows platform only)

`executable` — Name of existing Python executable file
string | character vector

Name of an existing Python executable file, specified as a string or character vector. This argument must contain the name of the Python executable file, and it can contain the full path.

Output Arguments

`version` — Python version
character vector

Python version number, returned as a character vector.

`executable` — Name of Python executable file
character vector

Name of Python executable file, returned as a character vector.

`isloaded` — Version loaded indicator
logical

Version loaded indicator specifying if this version is loaded, returned as logical. MATLAB loads Python when you type a `py.` command.

If MATLAB cannot load Python, `isloaded` is `0` and MATLAB displays “Undefined variable "py" or function "py.command"” when you type a `py.` command.

Compatibility Considerations

`pyversion` is not recommended

Not recommended starting in R2019b

`pyversion` is not recommended. Use `pyenv` instead. There are no plans to remove `pyversion` at this time.

To execute Python functions out-of-process, MATLAB provides a new function, `pyenv`. This function configures Python environment settings, including the version. Even if you do not use the out-of-process feature, MathWorks recommends using `pyenv` for managing all variables. For more information, see “Out-of-Process Execution of Python Functionality”.

See Also

`pyenv`

Introduced in R2014b

qmr

Quasi-minimal residual method

Syntax

```
x = qmr(A,b)
qmr(A,b,tol)
qmr(A,b,tol,maxit)
qmr(A,b,tol,maxit,M)
qmr(A,b,tol,maxit,M1,M2)
qmr(A,b,tol,maxit,M1,M2,x0)
[x,flag] = qmr(A,b,...)
[x,flag,relres] = qmr(A,b,...)
[x,flag,relres,iter] = qmr(A,b,...)
[x,flag,relres,iter,resvec] = qmr(A,b,...)
```

Description

`x = qmr(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and should be large and sparse. The column vector `b` must have length `n`. You can specify `A` as a function handle, `afun`, such that `afun(x,'notransp')` returns `A*x` and `afun(x,'transp')` returns `A'*x`.

“Parameterizing Functions” explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `qmr` converges, a message to that effect is displayed. If `qmr` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`qmr(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `qmr` uses the default, `1e-6`.

`qmr(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `qmr` uses the default, `min(n,20)`.

`qmr(A,b,tol,maxit,M)` and `qmr(A,b,tol,maxit,M1,M2)` use preconditioners `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `qmr` applies no preconditioner. `M` can be a function handle `mfun` such that `mfun(x,'notransp')` returns `M\x` and `mfun(x,'transp')` returns `M'\x`.

`qmr(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is `[]`, then `qmr` uses the default, an all zero vector.

`[x,flag] = qmr(A,b,...)` also returns a convergence flag.

Flag	Convergence
`0`	`qmr` converged to the desired tolerance `tol` within `maxit` iterations.
`1`	`qmr` iterated `maxit` times but did not converge.
`2`	Preconditioner `M` was ill-conditioned.
`3`	The method stagnated. (Two consecutive iterates were the same.)
`4`	One of the scalar quantities calculated during `qmr` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = qmr(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = qmr(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = qmr(A,b,...)` also returns a vector of the residual norms at each iteration, including `norm(b-A*x0)`.

Examples

Using qmr with a Matrix Input

This example shows how to use `qmr` with a matrix input. The code:

```
n = 100;
on = ones(n,1);
```

```
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8; maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x = qmr(A,b,tol,maxit,M1,M2);
```

displays the message:

```
qmr converged at iteration 9 to a solution...
with relative residual
5.6e-009
```

Using qmr with a Function Handle

This example replaces the matrix `A` in the previous example with a handle to a matrix-vector product function `afun`. The example is contained in a file `run_qmr` that

- Calls `qmr` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_qmr` are available to `afun`.

The following shows the code for `run_qmr`:

```
function x1 = run_qmr
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x1 = qmr(@afun,b,tol,maxit,M1,M2);

    function y = afun(x,transp_flag)
       if strcmp(transp_flag,'transp')      % y = A'*x
          y = 4 * x;
          y(1:n-1) = y(1:n-1) - 2 * x(2:n);
          y(2:n) = y(2:n) - x(1:n-1);
       elseif strcmp(transp_flag,'notransp') % y = A*x
          y = 4 * x;
          y(2:n) = y(2:n) - 2 * x(1:n-1);
```

```
            y(1:n-1) = y(1:n-1) - x(2:n);
        end
    end
end
```

When you enter

```
x1=run_qmr;
```

MATLAB software displays the message

```
qmr converged at iteration 9 to a solution with relative residual
5.6e-009
```

Using qmr with a Preconditioner

This example demonstrates the use of a preconditioner.

Load `A = west0479`, a real 479-by-479 nonsymmetric sparse matrix.

```
load west0479;
A = west0479;
```

Define `b` so that the true solution is a vector of all ones.

```
b = full(sum(A,2));
```

Set the tolerance and maximum number of iterations.

```
tol = 1e-12;
maxit = 20;
```

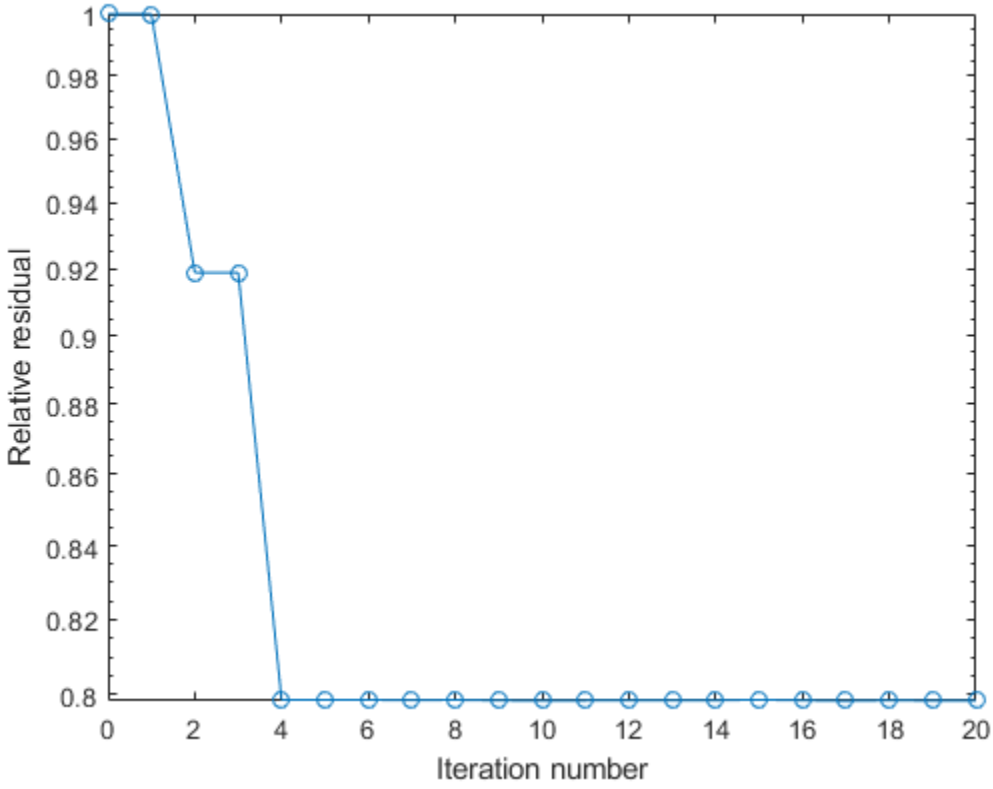
Use `qmr` to find a solution at the requested tolerance and number of iterations.

```
[x0,fl0,rr0,it0,rv0] = qmr(A,b,tol,maxit);
```

`fl0` is 1 because `qmr` does not converge to the requested tolerance `1e-12` within the requested 20 iterations. The seventeenth iterate is the best approximate solution and is the one returned as indicated by `it0 = 17`. MATLAB stores the residual history in `rv0`.

Plot the behavior of `qmr`.

```
semilogy(0:maxit,rv0/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



The plot shows that the solution does not converge. You can use a preconditioner to improve the outcome.

Create the preconditioner with `ilu`, since the matrix `A` is nonsymmetric.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-5));
```

```
Error using ilu
There is a pivot equal to zero. Consider decreasing
the drop tolerance or consider using the 'udiag' option.
```

MATLAB cannot construct the incomplete LU as it would result in a singular factor, which is useless as a preconditioner.

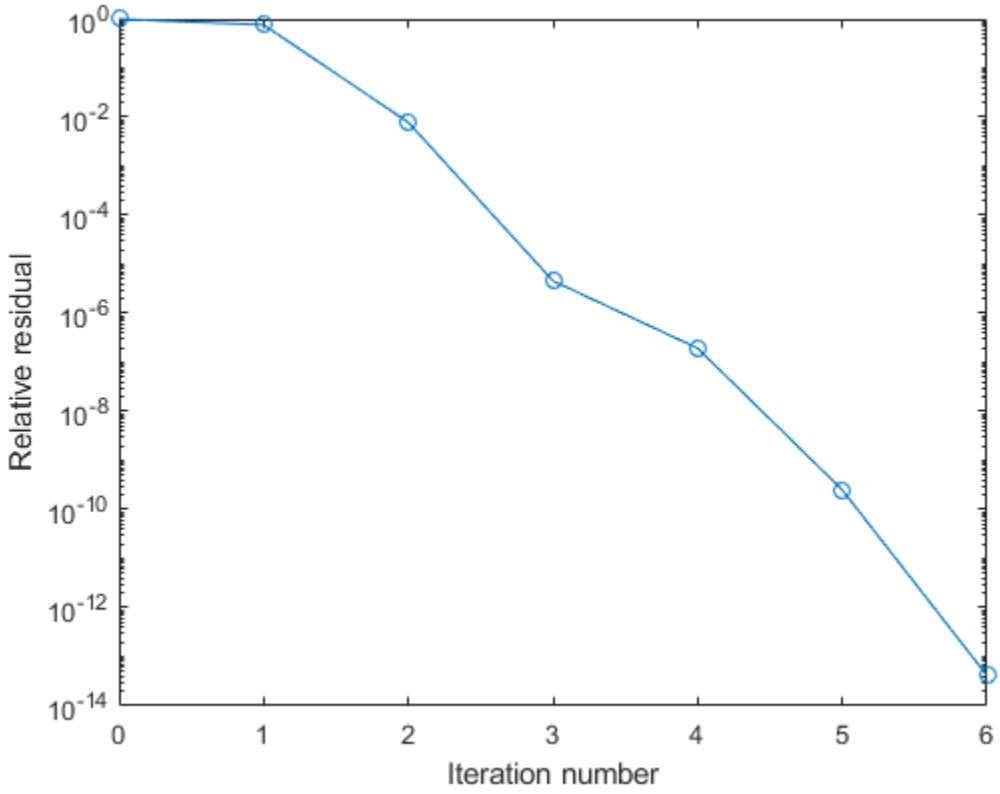
You can try again with a reduced drop tolerance, as indicated by the error message.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
[x1,fl1,rr1,it1,rv1] = qmr(A,b,tol,maxit,L,U);
```

`fl1` is 0 because `qmr` drives the relative residual to `4.1410e-014` (the value of `rr1`). The relative residual is less than the prescribed tolerance of `1e-12` at the sixth iteration (the value of `it1`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. The output `rv1(1)` is `norm(b)`, and the output `rv1(7)` is `norm(b-A*x2)`.

You can follow the progress of `qmr` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0).

```
semilogy(0:it1,rv1/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Freund, Roland W. and Nöel M. Nachtigal, "QMR: A quasi-minimal residual method for non-Hermitian linear systems," *SIAM Journal: Numer. Math.* 60, 1991, pp. 315–339.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- `b` must be a full (nonsparse) column vector.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- If `M1` is a function, then it is applied independently to each row.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bicg` | `bicgstab` | `cgs` | `gmres` | `ilu` | `lsqr` | `minres` | `mldivide` | `pcg` | `symmlq`

Topics

“Create Function Handle”

Introduced before R2006a

qr

QR decomposition

Syntax

```
X = qr(A)
[Q,R] = qr(A)
[Q,R,P] = qr(A)
[ ___ ] = qr(A,0)
[Q,R,P] = qr(A,outputForm)
```

```
[C,R] = qr(S,B)
[C,R,P] = qr(S,B)
[ ___ ] = qr(S,B,0)
[C,R,P] = qr(S,B,outputForm)
```

Description

`X = qr(A)` returns the upper-triangular `R` factor of the QR decomposition `A = Q*R`. If `A` is full, then `R = triu(X)`. If `A` is sparse, then `R = X`.

`[Q,R] = qr(A)` performs a QR decomposition on `m`-by-`n` matrix `A` such that `A = Q*R`. The factor `R` is an `m`-by-`n` upper-triangular matrix, and the factor `Q` is an `m`-by-`m` orthogonal matrix.

`[Q,R,P] = qr(A)` additionally returns a permutation matrix `P` such that `A*P = Q*R`.

`[ ___ ] = qr(A,0)` produces an economy-size decomposition using any of the previous output argument combinations. The size of the outputs depends on the size of `m`-by-`n` matrix `A`:

- If `m > n`, then `qr` computes only the first `n` columns of `Q` and the first `n` rows of `R`.
- If `m <= n`, then the economy-size decomposition is the same as the regular decomposition.
- If you specify a third output with the economy-size decomposition, then it is returned as a permutation vector such that `A(:,P) = Q*R`.

`[Q,R,P] = qr(A,outputForm)` specifies whether to return the permutation information `P` as a matrix or a vector. For example, if `outputForm` is `'vector'`, then `A(:,P) = Q*R`. The default value of `outputForm` is `'matrix'` such that `A*P = Q*R`.

`[C,R] = qr(S,B)` computes `C = Q'*B` and the upper-triangular factor `R`. You can use `C` and `R` to compute a least-squares solution to the sparse linear system `S*X = B` with `X = R\C`.

`[C,R,P] = qr(S,B)` additionally returns a permutation matrix `P`. You can use `C`, `R`, and `P` to compute a least-squares solution to the sparse linear system `S*X = B` with `X = P*(R\C)`.

`[ ____ ] = qr(S,B,0)` produces an economy-size decomposition using any of the previous output argument combinations. The size of the outputs depends on the size of `m`-by-`n` sparse matrix `S`:

- If `m > n`, then `qr` computes only the first `n` rows of `C` and `R`.
- If `m <= n`, then the economy-size decomposition is the same as the regular decomposition.
- If you specify a third output with the economy-size decomposition, then it is returned as a permutation vector such that the least-squares solution to `S*X = B` is `X(P,:) = R\C`.

`[C,R,P] = qr(S,B,outputForm)` specifies whether to return the permutation information `P` as a matrix or vector. For example, if `outputForm` is `'vector'`, then the least-squares solution to `S*X = B` is `X(P,:) = R\C`. The default value of `outputForm` is `'matrix'` such that the least-squares solution to `S*X = B` is `X = P*(R\C)`.

Examples

Q-Less QR Decomposition

Find the QR decomposition of the 5-by-5 Pascal matrix. Specify one output argument to return just the upper-triangular factor.

```
A = pascal(5);
X = qr(A)
```

```
X = 5×5
```

```
   -2.2361   -6.7082  -15.6525  -31.3050  -56.3489
    0.3090    3.1623   11.0680   26.5631   53.1263
    0.3090   -0.1744    1.8708    7.4833   19.2428
    0.3090   -0.4565    0.3548    0.6325    2.8460
    0.3090   -0.7387   -0.0281   -0.7490   -0.1195
```

Extract the upper-triangular factor R from X.

```
R = triu(X)
```

```
R = 5×5

   -2.2361   -6.7082  -15.6525  -31.3050  -56.3489
         0    3.1623   11.0680   26.5631   53.1263
         0         0    1.8708    7.4833   19.2428
         0         0         0    0.6325    2.8460
         0         0         0         0   -0.1195
```

Compare the R in the Q-less QR decomposition to the R factor in a full QR decomposition.

```
[Q1,R1] = qr(A)
```

```
Q1 = 5×5

   -0.4472   -0.6325    0.5345   -0.3162   -0.1195
   -0.4472   -0.3162   -0.2673    0.6325    0.4781
   -0.4472    0.0000   -0.5345    0.0000   -0.7171
   -0.4472    0.3162   -0.2673   -0.6325    0.4781
   -0.4472    0.6325    0.5345    0.3162   -0.1195


R1 = 5×5

   -2.2361   -6.7082  -15.6525  -31.3050  -56.3489
         0    3.1623   11.0680   26.5631   53.1263
         0         0    1.8708    7.4833   19.2428
         0         0         0    0.6325    2.8460
         0         0         0         0   -0.1195
```

Full QR Decomposition of Matrix

Compute the full QR decomposition of a magic square test matrix by specifying two output arguments.

```
A = magic(5);
[Q,R] = qr(A)
```

```
Q = 5×5

   -0.5234    0.5058    0.6735   -0.1215   -0.0441
   -0.7081   -0.6966   -0.0177    0.0815   -0.0800
   -0.1231    0.1367   -0.3558   -0.6307   -0.6646
   -0.3079    0.1911   -0.4122   -0.4247    0.7200
   -0.3387    0.4514   -0.4996    0.6328   -0.1774


R = 5×5

  -32.4808  -26.6311  -21.3973  -23.7063  -25.8615
         0   19.8943   12.3234    1.9439    4.0856
         0         0  -24.3985  -11.6316   -3.7415
         0         0         0  -20.0982   -9.9739
         0         0         0         0  -16.0005
```

Verify that $A = QR$, within machine precision.

```
norm(A-Q*R)
```

```
ans = 1.2569e-14
```

Permuted QR Decomposition

Specify three output arguments to return a permutation matrix or vector that reduces fill-in in the R factor of the QR decomposition.

Compute the QR decomposition of the `west0479` sparse matrix. Specify three outputs to return a permutation matrix that satisfies AP = QR.

```
load west0479
A = west0479;
[Q,R,P] = qr(A);
```

Verify that `A*P = Q*R` for the permutation matrix `P`, within machine precision.

```
norm(A*P-Q*R,'fro')
```

```
ans = 3.3386e-10
```

Now specify the `'vector'` option to return `p` as a permutation vector.

```
[Q,R,p] = qr(A,'vector');
```

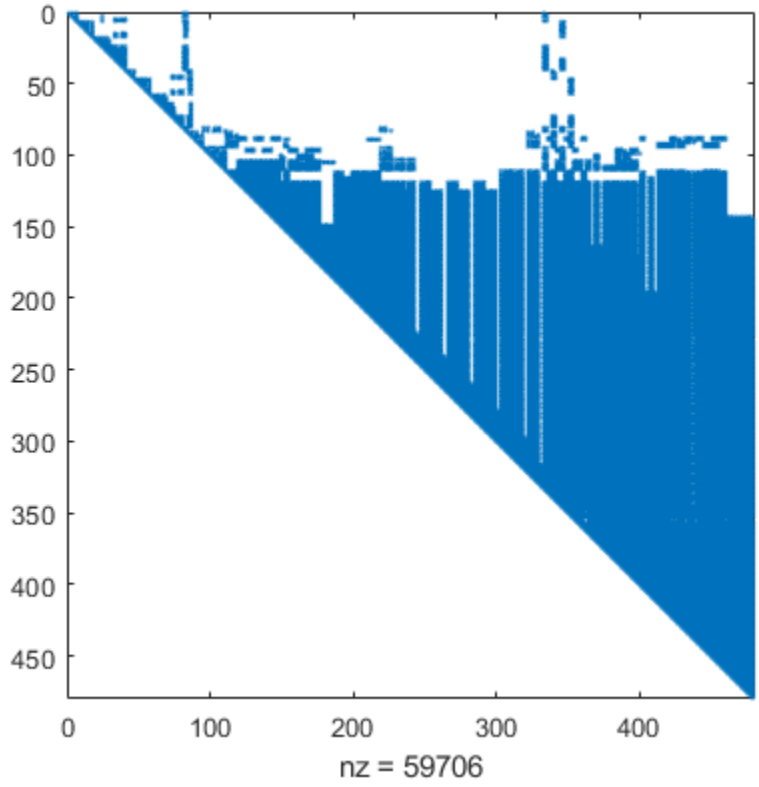
Verify that `A(:,p) = Q*R` for the permutation vector `p`, within machine precision.

```
norm(A(:,p) - Q*R,'fro')
```

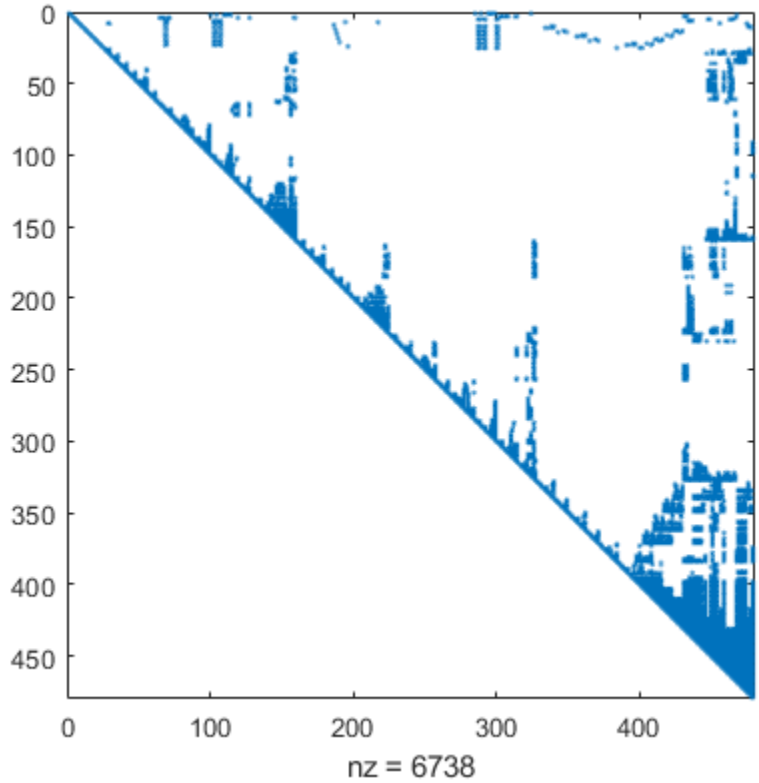
```
ans = 3.3386e-10
```

Verify that the use of a permutation matrix or permutation vector in the decomposition results in an `R` factor with fewer nonzeros for sparse inputs compared to a nonpermuted decomposition.

```
[Q1,R1] = qr(A);
spy(R1)
```




```
spy(R)
```



The results show that the permuted decomposition produces an R factor with substantially fewer nonzeros.

Solve Linear System with Economy-Size QR Factors

Use the economy-size QR decomposition of a coefficient matrix to solve the linear system Ax = *b*.

Create a 10-by-5 coefficient matrix by using the first five columns of `magic(10)`. For the right-hand side of the linear equation Ax = *b*, use the row sums of the matrix. With this setup, the solution to the equation *x* should be a vector of ones.

```
A = magic(10);
A = A(:,1:5)
```

```
A = 10×5

    92    99     1     8    15
    98    80     7    14    16
     4    81    88    20    22
    85    87    19    21     3
    86    93    25     2     9
    17    24    76    83    90
    23     5    82    89    91
    79     6    13    95    97
    10    12    94    96    78
    11    18   100    77    84
```

```
b = sum(A,2)
```

```
b = 10×1

   215
   215
   215
   215
   215
   290
   290
   290
   290
   290
```

Compute the economy-size QR decomposition of `A`. Then solve the linear system QRx = *b* with `x(p,:) = R\(Q\b)`. Because `Q` is orthogonal, this equation is the same as `x(p,:) = R\(Q'*b)`.

```
[Q,R,p] = qr(A,0)
```

```
Q = 10×5

   -0.0050   -0.4775   -0.0504    0.5193    0.0399
   -0.0349   -0.5001   -0.0990   -0.1954   -0.2006
   -0.4384    0.1059   -0.4660    0.4464    0.0628
   -0.0947   -0.4151   -0.2923   -0.2542    0.5274
```

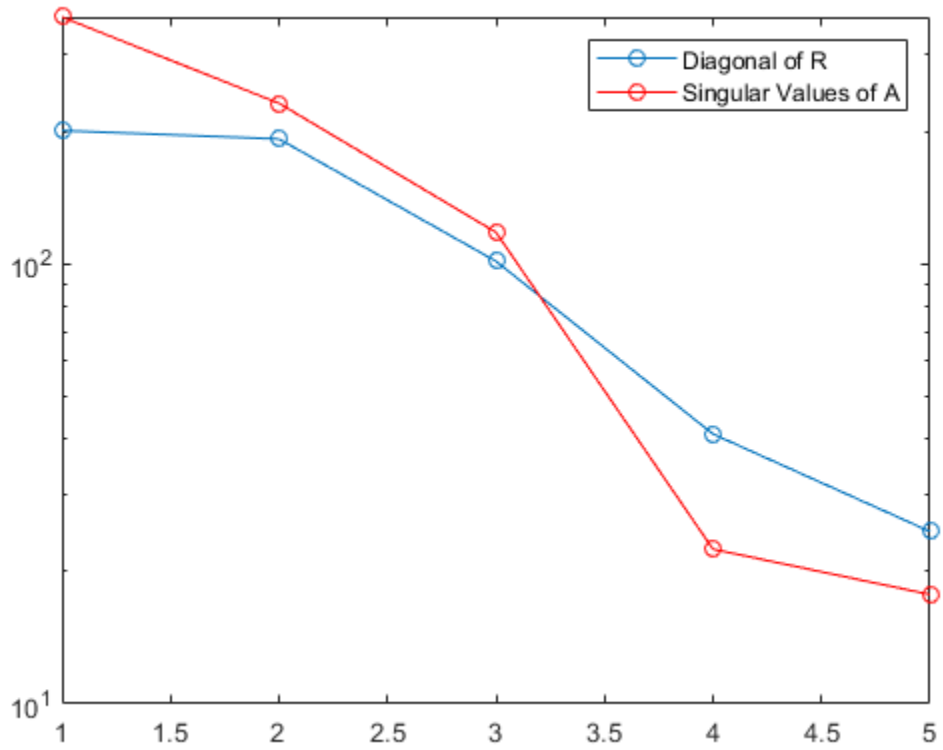
```
   -0.1246   -0.4117   -0.2812   -0.1326   -0.4130
   -0.3787    0.0209    0.2702    0.4697    0.0390
   -0.4085   -0.0017    0.2217   -0.2450   -0.2015
   -0.0648   -0.3925    0.6939    0.0669    0.1225
   -0.4683    0.0833    0.0283   -0.3038    0.5265
   -0.4982    0.0867    0.0394   -0.1822   -0.4138


R = 5×5

 -200.7112  -55.5026 -167.6040  -84.7237 -168.7997
         0 -192.1053  -40.3557 -152.4040  -39.2814
         0         0  101.3180  -89.4254   96.0172
         0         0         0   41.0248  -14.9083
         0         0         0         0   24.6386


p = 1×5

     3     1     5     2     4
```

```
x(p,:) = R\(Q\b)
```

```
x = 5×1

    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
```

Make a semilog plot of the diagonal of R to confirm that the permuted decomposition produces an R factor with `abs(diag(R))` decreasing. Plot the singular values of A in the same plot for comparison. In practice, the diagonal values of R behave in a similar way to the singular values of A. Therefore, you can use the diagonal values of R as a measure for how close to singular the matrix A is.

```
semilogy(abs(diag(R)),'-o')
hold on
semilogy(svd(A),'r-o')
legend('Diagonal of R','Singular Values of A')
```



Solve Sparse Linear System

Solve a sparse linear system and use the results to see how much of vector `b` lies in the column space of `S`.

Create a random 500-by-20 sparse matrix with 10% density and a vector of ones. Use `qr` to factorize the matrix into the factors `R` and `C = Q'*b`.

```
S = sprand(500,20,0.1);
b = ones(500,1);
[C,R] = qr(S,b,0);
```

Use the results to solve Sx = b with `x = R\C`.

```
x = R\C;
```

Consider the identity$\|b\|^2 = \|\mathrm{Sx} - b\|^2 + \|C\|^2$.

Dividing through by the norm of `b`, you get a new identity that shows how much of `b` lies in the column space of `S`:

$$\frac{\|\mathrm{Sx} - b\|^2}{\|b\|^2} + \frac{\|C\|^2}{\|b\|^2} = 1.$$

The first term tells how much of `b` *does not* lie in the column space of `S`, while the second term tells how much of `b` *does* lie in the column space of `S`.

```
t1 = norm(S*x-b)^2/norm(b)^2
```

```
t1 = 0.4000
```

```
t2 = norm(C)^2/norm(b)^2
```

```
t2 = 0.6000
```

Solve Rectangular Sparse Linear System

Use `qr` to solve the matrix equation Sx = B with a rectangular sparse coefficient matrix S.

Load the `west0479` sparse matrix and use the first 200 columns as the rectangular coefficient matrix in a linear system. For the right-hand side of the equation, use the row sums of S. With this setup, the solution to Sx = B is a vector of ones.

```
load west0479
S = west0479(:,1:200);
B = sum(S,2);
```

Solve Sx = B using `qr` with two inputs and three outputs. The solution to the linear system is `x = P*(R\C)`.

```
[C,R,P] = qr(S,B);
x = P*(R\C);
```

Verify that Sx − B = 0, within machine precision.

```
norm(S*x-B)
```

```
ans = 8.4349e-11
```

Note: To calculate the upper-triangular factor R and permutation matrix P, but avoid computing the orthogonal matrix Q (which is often the most computationally expensive part of a call to qr), you can specify B as an empty matrix:

```
emptyB = zeros(size(S,1),0);
[~,R,P] = qr(S,emptyB);
```

Input Arguments

A — Input matrix

matrix

Input matrix, specified as a full or sparse matrix.

Data Types: single | double
Complex Number Support: Yes

S — Input coefficient matrix

matrix

Input coefficient matrix, specified as a sparse matrix. With two input matrices, qr computes a least-squares solution to the linear system S*X = B.

Data Types: double
Complex Number Support: Yes

B — Right-hand side matrix

matrix

Right-hand side matrix, specified as a full or sparse matrix. With two input matrices, qr computes C = Q'*B, which you can use to solve the linear system S*X = B.

Data Types: single | double
Complex Number Support: Yes

outputForm — Shape of permutation output

'matrix' (default) | 'vector'

Shape of permutation output, specified as `'matrix'` or `'vector'`. This flag controls whether the permutation output `P` is returned as a permutation matrix or permutation vector. You must specify three output arguments to `qr` to use this option.

- If `outputForm` is `'vector'`, then `P` is a permutation vector that satisfies `A(:,P) = Q*R`.
- The default value of `outputForm` is `'matrix'` such that `A*P = Q*R`.

Example: `[Q,R,P] = qr(A,'vector')`

Output Arguments

`X` — Output matrix
matrix

Output matrix. The contents of `X` depend on whether `A` is full or sparse:

- If `A` is full, then the upper-triangular factor of the QR decomposition is `R = triu(X)`. (The lower-triangular elements are part of the data used to calculate `Q`.)
- If `A` is sparse, then the factor is `R = X`.

`Q` — Orthogonal factor
matrix

Orthogonal factor, returned as a matrix that satisfies `A = Q*R` for an `m`-by-`n` matrix `A`.

- For full decompositions, `qr(A)` returns `Q` as an `m`-by-`m` orthogonal matrix satisfying $Q^H Q = QQ^H = I_m$.
- For rectangular `A` with `m > n`, the economy-sized decomposition `qr(A,0)` computes only the first `n` columns of `Q` and first `n` rows of `R`. The columns of `Q` form an orthonormal basis for the column space of `A`.

Different machines and releases of MATLAB can produce different columns in `Q` that are still numerically accurate. Corresponding rows and columns in `Q` and `R` can flip their signs, since this does not affect the value of the expression `A = Q*R`.

`R` — Upper-triangular factor
matrix

Upper-triangular factor, returned as a matrix that satisfies `A = Q*R`.

P — Permutation information
matrix | vector

Permutation information, returned as a matrix or vector. The shape of P depends on the value of `outputForm`. Also, `qr` selects P to satisfy different criteria depending on whether the first input matrix is full or sparse:

- Full — `qr` selects P so that `abs(diag(R))` is decreasing.
- Sparse — `qr` selects P to reduce fill-in in R.

C — Linear system factor
matrix

Linear system factor, returned as a matrix that satisfies `C = Q'*B`. The least-squares solution to `S*X = B` is `X = R\C`. If the permutation output P is specified, then the solution is either `X = P*(R\C)` or `X(P,:) = R\C`, depending on the value of `outputForm`:

- If `outputForm` is `'vector'`, then the least-squares solution to `S*X = B` is `X(P,:) = R\C`.
- The default value of `outputForm` is `'matrix'` such that the least-squares solution to `S*X = B` is `X = P*(R\C)`.

Tips

- To solve multiple linear systems involving the same coefficient matrix, use `decomposition` objects.
- For the syntax `[C,R] = qr(S,B)`, the value of `X = R\C` is a least-squares solution to `S*X = B` only when S does not have low rank.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The input `A` must be nonsparse.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- The input `A` must be nonsparse.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`chol` | `decomposition` | `lsqminnorm` | `lu` | `null` | `orth` | `qrdelete` | `qrinsert` | `qrupdate` | `rank`

Topics

"Factorizations"
"Systems of Linear Equations"

Introduced before R2006a

qrdelete

Remove column or row from QR factorization

Syntax

```
[Q1,R1] = qrdelete(Q,R,j)
[Q1,R1] = qrdelete(Q,R,j,'col')
[Q1,R1] = qrdelete(Q,R,j,'row')
```

Description

`[Q1,R1] = qrdelete(Q,R,j)` returns the QR factorization of the matrix `A1`, where `A1` is `A` with the column `A(:,j)` removed and `[Q,R] = qr(A)` is the QR factorization of `A`.

`[Q1,R1] = qrdelete(Q,R,j,'col')` is the same as `qrdelete(Q,R,j)`.

`[Q1,R1] = qrdelete(Q,R,j,'row')` returns the QR factorization of the matrix `A1`, where `A1` is `A` with the row `A(j,:)` removed and `[Q,R] = qr(A)` is the QR factorization of `A`.

Examples

```
A = magic(5);
[Q,R] = qr(A);
j = 3;
[Q1,R1] = qrdelete(Q,R,j,'row');

Q1 =
    0.5274   -0.5197   -0.6697   -0.0578
    0.7135    0.6911    0.0158    0.1142
    0.3102   -0.1982    0.4675   -0.8037
    0.3413   -0.4616    0.5768    0.5811

R1 =
   32.2335   26.0908   19.9482   21.4063   23.3297
```

```
         0  -19.7045  -10.9891    0.4318   -1.4873
         0         0   22.7444    5.8357   -3.1977
         0         0         0  -14.5784    3.7796
```

returns a valid QR factorization, although possibly different from

```
A2 = A;
A2(j,:) = [];
[Q2,R2] = qr(A2)

Q2 =
   -0.5274    0.5197    0.6697   -0.0578
   -0.7135   -0.6911   -0.0158    0.1142
   -0.3102    0.1982   -0.4675   -0.8037
   -0.3413    0.4616   -0.5768    0.5811

R2 =
  -32.2335  -26.0908  -19.9482  -21.4063  -23.3297
         0   19.7045   10.9891   -0.4318    1.4873
         0         0  -22.7444   -5.8357    3.1977
         0         0         0  -14.5784    3.7796
```

Algorithms

The `qrdelete` function uses a series of Givens rotations to zero out the appropriate elements of the factorization. [1]

References

[1] Golub, Gene H., and Charles F. Van Loan. *Matrix Computations*. 4th ed. Baltimore, MD: Johns Hopkins University Press, 2013, Sections 6.5.2–6.5.3, pp. 335–338.

See Also

`planerot` | `qr` | `qrinsert`

Introduced before R2006a

qrinsert

Insert column or row into QR factorization

Syntax

```
[Q1,R1] = qrinsert(Q,R,j,x)
[Q1,R1] = qrinsert(Q,R,j,x,'col')
[Q1,R1] = qrinsert(Q,R,j,x,'row')
```

Description

`[Q1,R1] = qrinsert(Q,R,j,x)` returns the QR factorization of the matrix `A1`, where `A1` is `A = Q*R` with the column `x` inserted before `A(:,j)`. If `A` has `n` columns and `j = n +1`, then `x` is inserted after the last column of `A`.

`[Q1,R1] = qrinsert(Q,R,j,x,'col')` is the same as `qrinsert(Q,R,j,x)`.

`[Q1,R1] = qrinsert(Q,R,j,x,'row')` returns the QR factorization of the matrix `A1`, where `A1` is `A = Q*R` with an extra row, `x`, inserted before `A(j,:)`.

Examples

```
A = magic(5);
[Q,R] = qr(A);
j = 3;
x = 1:5;
[Q1,R1] = qrinsert(Q,R,j,x,'row')

Q1 =
    0.5231    0.5039   -0.6750    0.1205    0.0411    0.0225
    0.7078   -0.6966    0.0190   -0.0788    0.0833   -0.0150
    0.0308    0.0592    0.0656    0.1169    0.1527   -0.9769
    0.1231    0.1363    0.3542    0.6222    0.6398    0.2104
    0.3077    0.1902    0.4100    0.4161   -0.7264   -0.0150
    0.3385    0.4500    0.4961   -0.6366    0.1761    0.0225
```

```
R1 =
   32.4962   26.6801   21.4795   23.8182   26.0031
         0   19.9292   12.4403    2.1340    4.3271
         0         0   24.4514   11.8132    3.9931
         0         0         0   20.2382   10.3392
         0         0         0         0   16.1948
         0         0         0         0         0
```

returns a valid QR factorization, although possibly different from

```
A2 = [A(1:j-1,:); x; A(j:end,:)];
[Q2,R2] = qr(A2)

Q2 =
   -0.5231    0.5039    0.6750   -0.1205    0.0411    0.0225
   -0.7078   -0.6966   -0.0190    0.0788    0.0833   -0.0150
   -0.0308    0.0592   -0.0656   -0.1169    0.1527   -0.9769
   -0.1231    0.1363   -0.3542   -0.6222    0.6398    0.2104
   -0.3077    0.1902   -0.4100   -0.4161   -0.7264   -0.0150
   -0.3385    0.4500   -0.4961    0.6366    0.1761    0.0225

R2 =
  -32.4962  -26.6801  -21.4795  -23.8182  -26.0031
         0   19.9292   12.4403    2.1340    4.3271
         0         0  -24.4514  -11.8132   -3.9931
         0         0         0  -20.2382  -10.3392
         0         0         0         0   16.1948
         0         0         0         0         0
```

Algorithms

The `qrinsert` function inserts the values of `x` into the `j`th column (row) of `R`. It then uses a series of Givens rotations to zero out the nonzero elements of `R` on and below the diagonal in the `j`th column (row). [1]

References

[1] Golub, Gene H., and Charles F. Van Loan. *Matrix Computations*. 4th ed. Baltimore, MD: Johns Hopkins University Press, 2013, Sections 6.5.2–6.5.3, pp. 335–338.

See Also

`planerot` | `qr` | `qrdelete`

Introduced before R2006a

qrupdate

Rank 1 update to QR factorization

Syntax

```
[Q1,R1] = qrupdate(Q,R,u,v)
```

Description

`[Q1,R1] = qrupdate(Q,R,u,v)` when `[Q,R] = qr(A)` is the original QR factorization of `A`, returns the QR factorization of `A + u*v'`, where `u` and `v` are column vectors of appropriate lengths.

Examples

The matrix

```
mu = sqrt(eps)

mu =

   1.4901e-08

A = [ones(1,4); mu*eye(4)];
```

is a well-known example in least squares that indicates the dangers of forming `A'*A`. Instead, we work with the QR factorization – orthonormal Q and upper triangular R.

```
 [Q,R] = qr(A);
```

As we expect, `R` is upper triangular.

```
R =

   -1.0000   -1.0000   -1.0000   -1.0000
         0    0.0000    0.0000    0.0000
```

```
         0         0    0.0000    0.0000
         0         0         0    0.0000
         0         0         0         0
```

In this case, the upper triangular entries of R, excluding the first row, are on the order of `sqrt(eps)`.

Consider the update vectors

```
 u = [-1 0 0 0 0]'; v = ones(4,1);
```

Instead of computing the rather trivial QR factorization of this rank one update to A from scratch with

```
[QT,RT] = qr(A + u*v')

QT =

     0     0     0     0     1
    -1     0     0     0     0
     0    -1     0     0     0
     0     0    -1     0     0
     0     0     0    -1     0

RT =

  1.0e-007 *

   -0.1490         0         0         0
         0   -0.1490         0         0
         0         0   -0.1490         0
         0         0         0   -0.1490
         0         0         0         0
```

we may use `qrupdate`.

```
[Q1,R1] = qrupdate(Q,R,u,v)

Q1 =

   -0.0000   -0.0000   -0.0000   -0.0000    1.0000
    1.0000   -0.0000   -0.0000   -0.0000    0.0000
    0.0000    1.0000   -0.0000   -0.0000    0.0000
    0.0000    0.0000    1.0000   -0.0000    0.0000
   -0.0000   -0.0000   -0.0000    1.0000    0.0000
```

```
R1 =

   1.0e-007 *
    0.1490    0.0000    0.0000    0.0000
         0    0.1490    0.0000    0.0000
         0         0    0.1490    0.0000
         0         0         0    0.1490
         0         0         0         0
```

Note that both factorizations are correct, even though they are different.

Tips

`qrupdate` works only for full matrices.

Algorithms

`qrupdate` uses the algorithm in section 12.5.1 of the third edition of *Matrix Computations* by Golub and van Loan. `qrupdate` is useful since, if we take `N = max(m,n)`, then computing the new QR factorization from scratch is roughly an $O(N^3)$ algorithm, while simply updating the existing factors in this way is an $O(N^2)$ algorithm.

References

[1] Golub, Gene H. and Charles Van Loan, *Matrix Computations*, Third Edition, Johns Hopkins University Press, Baltimore, 1996

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Because `qrupdate` operates in-place on `Q` and `R`, the outputs `Q1` and `R1` must be the same type as the inputs `Q` and `R`.

- `Q` and `R` must belong to the same class as each other, either both single or both double.
- If either `u` or `v` is single, both `Q` and `R` must be single.
- If any of the inputs are complex, both `Q` and `R` must be complex.

See Also

`cholupdate` | `qr`

Introduced before R2006a

quad

(Not recommended) Numerically evaluate integral, adaptive Simpson quadrature

Note `quad` is not recommended. Use `integral` instead.

Syntax

```
q = quad(fun,a,b)
q = quad(fun,a,b,tol)
q = quad(fun,a,b,tol,trace)
[q,fcnt] = quad(...)
```

Description

Quadrature is a numerical method used to find the area under the graph of a function, that is, to compute a definite integral.

$$q = \int_a^b f(x)dx$$

`q = quad(fun,a,b)` tries to approximate the integral of function `fun` from `a` to `b` to within an error of `1e-6` using recursive adaptive Simpson quadrature. `fun` is a function handle. Limits `a` and `b` must be finite. The function `y = fun(x)` should accept a vector argument `x` and return a vector result `y`, the integrand evaluated at each element of `x`.

“Parameterizing Functions” explains how to provide additional parameters to the function `fun`, if necessary.

`q = quad(fun,a,b,tol)` uses an absolute error tolerance `tol` instead of the default which is `1.0e-6`. Larger values of `tol` result in fewer function evaluations and faster computation, but less accurate results. In MATLAB version 5.3 and earlier, the `quad` function used a less reliable algorithm and a default relative tolerance of `1.0e-3`.

`q = quad(fun,a,b,tol,trace)` with non-zero `trace` shows the values of `[fcnt a b-a Q]` during the recursion.

`[q,fcnt] = quad(...)` returns the number of function evaluations.

The function `quadl` may be more efficient with high accuracies and smooth integrands.

The list below contains information to help you determine which quadrature function in MATLAB to use:

- The `quad` function may be most efficient for low accuracies with nonsmooth integrands.
- The `quadl` function may be more efficient than `quad` at higher accuracies with smooth integrands.
- The `quadgk` function may be most efficient for high accuracies and oscillatory integrands. It supports infinite intervals and can handle moderate singularities at the endpoints. It also supports contour integration along piecewise linear paths.
- The `quadv` function vectorizes `quad` for an array-valued `fun`.
- If the interval is infinite, $[a, \infty)$, then for the integral of `fun(x)` to exist, `fun(x)` must decay as `x` approaches infinity, and `quadgk` requires it to decay rapidly. Special methods should be used for oscillatory functions on infinite intervals, but `quadgk` can be used if `fun(x)` decays fast enough.
- The `quadgk` function will integrate functions that are singular at finite endpoints if the singularities are not too strong. For example, it will integrate functions that behave at an endpoint `c` like `log|x-c|` or `|x-c|`^{p} for `p >= -1/2`. If the function is singular at points inside `(a,b)`, write the integral as a sum of integrals over subintervals with the singular points as endpoints, compute them with `quadgk`, and add the results.

Examples

To compute the integral

$$\int_0^2 \frac{1}{x^3 - 2x - 5} dx,$$

write a function `myfun` that computes the integrand:

```
function y = myfun(x)
y = 1./(x.^3-2*x-5);
```

Then pass `@myfun`, a function handle to `myfun`, to `quad`, along with the limits of integration, `0` to `2`:

```
Q = quad(@myfun,0,2)
```

```
Q =

   -0.4605
```

Alternatively, you can pass the integrand to `quad` as an anonymous function handle `F`:

```
F = @(x)1./(x.^3-2*x-5);
Q = quad(F,0,2);
```

Diagnostics

`quad` may issue one of the following warnings:

`'Minimum step size reached'` indicates that the recursive interval subdivision has produced a subinterval whose length is on the order of roundoff error in the length of the original interval. A nonintegrable singularity is possible.

`'Maximum function count exceeded'` indicates that the integrand has been evaluated more than 10,000 times. A nonintegrable singularity is likely.

`'Infinite or Not-a-Number function value encountered'` indicates a floating point overflow or division by zero during the evaluation of the integrand in the interior of the interval.

Algorithms

`quad` implements a low order method using an adaptive recursive Simpson's rule.

References

[1] Gander, W. and W. Gautschi, “Adaptive Quadrature – Revisited,” BIT, Vol. 40, 2000, pp. 84-101. This document is also available at `https://www.inf.ethz.ch/personal/gander`.

See Also

`dblquad` | `integral` | `integral2` | `integral3` | `quad2d` | `quadgk` | `quadl` | `quadv` | `trapz` | `triplequad`

Topics

"Anonymous Functions"
"Create Function Handle"

Introduced before R2006a

quad2d

Numerically evaluate double integral — tiled method

Syntax

```
q = quad2d(fun,a,b,c,d)
q = quad2d(fun,a,b,c,d,Name,Value)
[q,E] = quad2d( ___ )
```

Description

`q = quad2d(fun,a,b,c,d)` approximates the integral of `fun(x,y)` over the planar region $a \le x \le b$ and $c(x) \le y \le d(x)$. The bounds `c` and `d` can each be scalars or function handles.

`q = quad2d(fun,a,b,c,d,Name,Value)` specifies additional options with one or more `Name,Value` pair arguments. For example, you can specify `'AbsTol'` and `'RelTol'` to adjust the error thresholds that the algorithm must satisfy.

`[q,E] = quad2d( ___ )` also returns an approximate upper bound on the absolute error, $E = | q - I |$, where I is the exact value of the integral.

Examples

Evaluate Double Integral

Integrate

$$y\sin(x) + x\cos(y)$$

over $-\pi \le x \le 2\pi$ and $0 \le y \le \pi$.

```
fun = @(x,y) y.*sin(x)+x.*cos(y);
Q = quad2d(fun,pi,2*pi,0,pi)
```

```
Q = -9.8696
```

Compare the result to the true value of the integral, $-\pi^2$.

```
-pi^2
```

```
ans = -9.8696
```

Integrand with Singularity on Integration Boundary

Integrate the function

$$\left[(x+y)^{1/2}(1+x+y)^2\right]^{-1}$$

over the region $0 \le x \le 1$ and $0 \le y \le 1 - x$. This integrand is infinite at the origin (0,0), which lies on the boundary of the integration region.

```
fun = @(x,y) 1./(sqrt(x + y) .* (1 + x + y).^2 );
ymax = @(x) 1 - x;
Q = quad2d(fun,0,1,0,ymax)
```

```
Q = 0.2854
```

The true value of the integral is $\pi/4 - 1/2$.

```
pi/4 - 0.5
```

```
ans = 0.2854
```

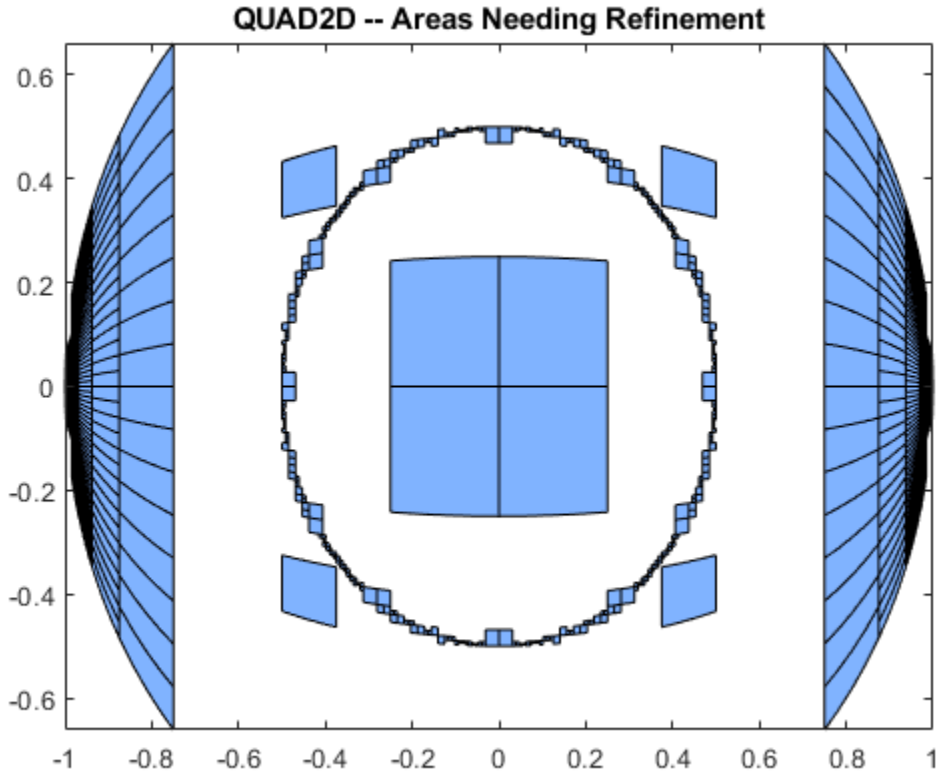
Limitations of quad2d

`quad2d` begins by mapping the region of integration to a rectangle. Consequently, it may have trouble integrating over a region that does not have four sides or has a side that cannot be mapped smoothly to a straight line. If the integration is unsuccessful, some helpful tactics are leaving `Singular` set to its default value of `true`, changing between Cartesian and polar coordinates, or breaking the region of integration into pieces and adding the results of integration over the pieces.

For instance:

```
fun = @(x,y)abs(x.^2 + y.^2 - 0.25);
c = @(x)-sqrt(1 - x.^2);
d = @(x)sqrt(1 - x.^2);
quad2d(fun,-1,1,c,d,'AbsTol',1e-8,...
    'FailurePlot',true,'Singular',false);
```

```
Warning: Reached the maximum number of function evaluations (2000). The result fails th
```

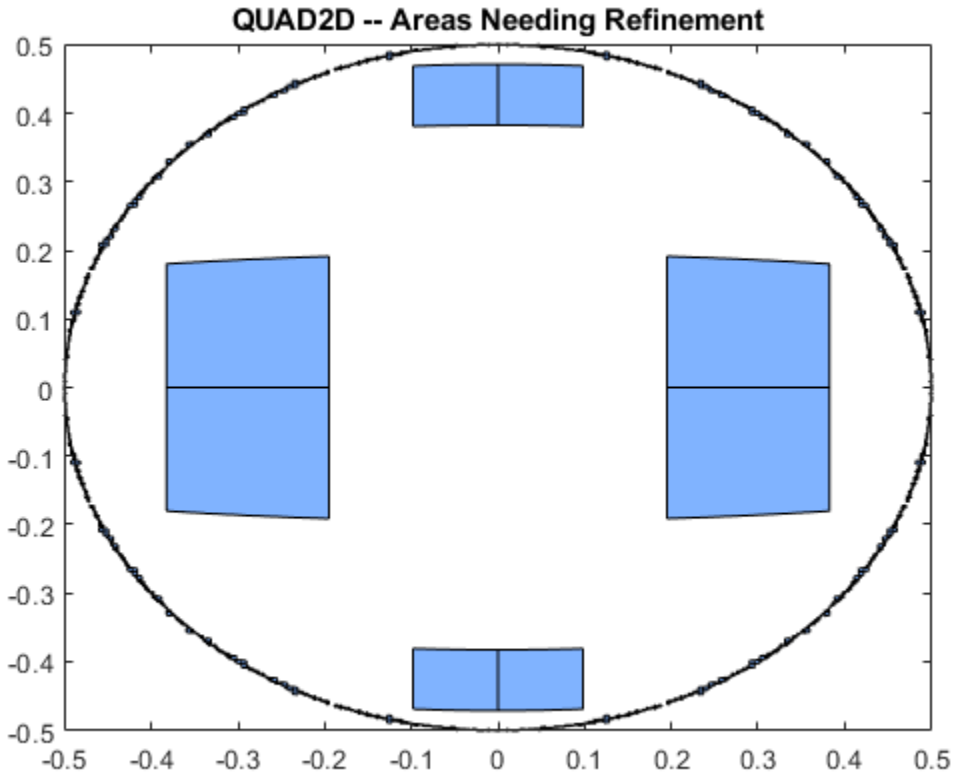


The failure plot shows two areas of difficulty, near the points `(-1,0)` and `(1,0)` and near the circle $x^2 + y^2 = 0.25$.

Changing the value of `Singular` to `true` will cope with the geometric singularities at `(-1,0)` and `(1,0)`. The larger shaded areas may need refinement but are probably not areas of difficulty.

```
Q = quad2d(fun,-1,1,c,d,'AbsTol',1e-8, ...
     'FailurePlot',true,'Singular',true);
```

```
Warning: Reached the maximum number of function evaluations (2000). The result passes t
```



From here you can take advantage of symmetry:

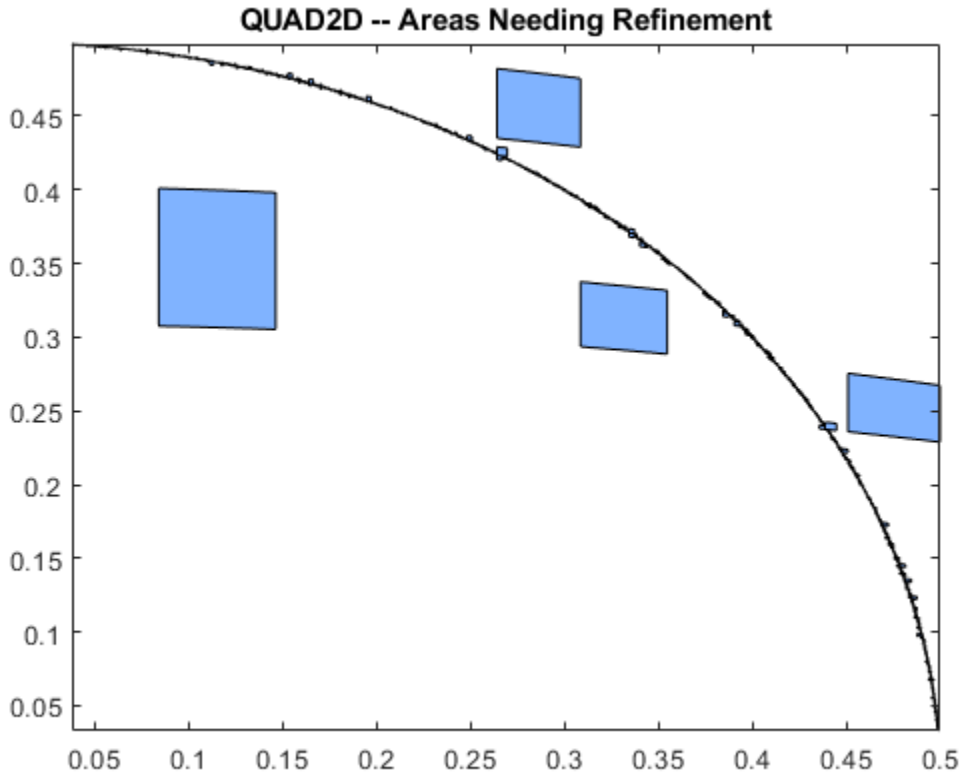
```
Q = 4*quad2d(fun,0,1,0,d,'Abstol',1e-8,...
     'Singular',true,'FailurePlot',true)
```

```
Q = 0.9817
```

However, the code is still working very hard near the singularity. It may not be able to provide higher accuracy:

```
Q = 4*quad2d(fun,0,1,0,d,'Abstol',1e-10,...
     'Singular',true,'FailurePlot',true);
```

```
Warning: Reached the maximum number of function evaluations (2000). The result passes t
```



At higher accuracy, a change in coordinates may work better.

```
polarfun = @(theta,r) fun(r.*cos(theta),r.*sin(theta)).*r;
Q = 4*quad2d(polarfun,0,pi/2,0,1,'AbsTol',1e-10);
```

It is best to put the singularity on the boundary by splitting the region of integration into two parts:

```
Q1 = 4*quad2d(polarfun,0,pi/2,0,0.5,'AbsTol',5e-11);
Q2 = 4*quad2d(polarfun,0,pi/2,0.5,1,'AbsTol',5e-11);
Q = Q1 + Q2;
```

Input Arguments

`fun` — Function to integrate

function handle

Function to integrate, specified as a function handle. The function `Z = fun(X,Y)` must accept 2-D matrices `X` and `Y` of the same size and return a matrix `Z` of corresponding values. Therefore, the function must be vectorized (that is, you must use elementwise operators such as `.^` instead of matrix operators such as `^`). The inputs and outputs of the function must be either single or double precision.

Example: `@(x,y) x.^2 - y.^2`

Data Types: `function_handle`

`a,b` — *x* limits of integration

scalars

x limits of integration, specified as scalars.

Data Types: `single` | `double`
Complex Number Support: Yes

`c,d` — *y* limits of integration

scalars | function handles

y limits of integration, specified as scalars or function handles. Each limit can be specified as a scalar or a function handle. If the limits are specified as function handles, then they are functions of the *x* limit of integration `ymin = @x c(x)` and `ymax = @(x) d(x)`. The function handles `ymin` and `ymax` must accept matrices and return matrices of the same size with the corresponding values. The inputs and outputs of the functions must be either single or double precision.

Data Types: `single` | `double` | `function_handle`
Complex Number Support: Yes

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `quad2d(@(x,y) x.*y.^2, 0, 1, 0, 2, 'AbsTol',1e-3)` specifies the absolute tolerance for the integration as `1e-3`.

`AbsTol` — Absolute error tolerance
`1e-5` (default) | scalar

Absolute error tolerance, specified as the comma-separated pair consisting of `'AbsTol'` and a scalar.

`quad2d` attempts to satisfy `ERRBND <= max(AbsTol,RelTol*|Q|)`. This is absolute error control when `|Q|` is sufficiently small and relative error control when `|Q|` is larger. A default tolerance value is used when a tolerance is not specified. The default value of `AbsTol` is 1e-5. The default value of `RelTol` is `100*eps(class(Q))`. This is also the minimum value of `RelTol`. Smaller `RelTol` values are automatically increased to the default value.

`RelTol` — Relative error tolerance
`100*eps(class(q))` (default) | scalar

Relative error tolerance, specified as the comma-separated pair consisting of `'RelTol'` and a scalar.

`quad2d` attempts to satisfy `ERRBND <= max(AbsTol,RelTol*|Q|)`. This is absolute error control when `|Q|` is sufficiently small and relative error control when `|Q|` is larger. A default tolerance value is used when a tolerance is not specified. The default value of `AbsTol` is 1e-5. The default value of `RelTol` is `100*eps(class(Q))`. This is also the minimum value of `RelTol`. Smaller `RelTol` values are automatically increased to the default value.

`MaxFunEvals` — Maximum number of evaluations of `fun`
`2000` (default) | scalar

Maximum number of evaluations of `fun`, specified as the comma-separated pair consisting of `'MaxFunEvals'` and a scalar. Use this option to limit the number of times `quad2d` evaluates the function `fun`.

`FailurePlot` — Toggle to generate failure plot
`false` or `0` (default) | `true` or `1`

Toggle to generate failure plot, specified as the comma-separated pair consisting of `'FailurePlot'` and a numeric or logical `1` (`true`) or `0` (`false`). Set `FailurePlot` to `true` or `1` to generate a graphical representation of the regions needing further refinement when `MaxFunEvals` is reached. No plot is generated if the integration succeeds before reaching `MaxFunEvals`. The failure plot contains (generally) 4-sided regions that are mapped to rectangles internally. Clusters of small regions indicate the areas of difficulty in the integration.

`Singular` — Toggle to transform boundary singularities
`true` or `1` (default) | `false` or `0`

Toggle to transform boundary singularities, specified as the comma-separated pair consisting of `'Singular'` and a numeric or logical `1` (`true`) or `0` (`false`). By default, `quad2d` employs transformations to weaken boundary singularities for better performance. Set `'Singular'` to `false` or `0` to turn these transformations off, which can provide a performance benefit on some smooth problems.

Output Arguments

q — Calculated integral
scalar

Calculated integral, returned as a scalar.

E — Error bound
scalar

Error bound, returned as a scalar. The error bound provides an upper bound on the error between the calculated integral q and the exact value of the integral I such that $E = | q - I |$.

References

[1] L.F. Shampine, "Matlab Program for Quadrature in 2D." *Applied Mathematics and Computation.* Vol. 202, Issue 1, 2008, pp. 266–274.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Generated code issues a warning if the size of the internal storage arrays is not large enough. If a warning occurs, as a workaround, you can try to divide the region of integration into pieces and sum the integrals over each piece.

See Also

`integral` | `integral2` | `integral3` | `quadgk`

Topics

"Anonymous Functions"
"Create Function Handle"

Introduced in R2009a

quadgk

Numerically evaluate integral — Gauss-Kronrod quadrature

Syntax

```
q = quadgk(fun,a,b)
[q,errbnd] = quadgk(fun,a,b)
[ ___ ] = quadgk(fun,a,b,Name,Value)
```

Description

`q = quadgk(fun,a,b)` integrates the function handle `fun` from `a` to `b` using high-order global adaptive quadrature and default error tolerances.

`[q,errbnd] = quadgk(fun,a,b)` additionally returns an approximate upper bound on the absolute error `|q - I|`, where `I` is the exact value of the integral.

`[ ___ ] = quadgk(fun,a,b,Name,Value)` specifies additional options with one or more name-value pair arguments using either of the previous output argument combinations. For example, specify `'Waypoints'` followed by a vector of real or complex numbers to indicate specific points for the integrator to use.

Examples

Integral with Singularity at Endpoint

Evaluate the integral

$$q = \int_0^1 e^x \ln(x)\, dx.$$

This integral has a singularity at the point $x = 0$ because $\ln(0)$ diverges to $-\infty$.

Create an anonymous function for the integrand. The `log` function calculates $\ln(x)$.

```
f = @(x) exp(x).*log(x);
```

Integrate `f` from 0 to 1.

```
q = quadgk(f,0,1)
```

```
q = -1.3179
```

Complex Contour Integration

Integrate a complex function around a pole by specifying a contour.

Evaluate the complex contour integral

$$q = \oint \frac{\mathrm{dz}}{2z-1}.$$

The integrand has a simple pole at $z = 1/2$, so use a rectangular contour that encloses that point. The contour starts and ends at $x = 1$ on the real number line. Use the `'Waypoints'` name-value pair to specify the piecewise segments in the contour.

```
f = @(z) 1./(2.*z-1);
contour_segments = [1+1i 0+1i 0-1i 1-1i];
q = quadgk(f,1,1,'Waypoints',contour_segments)
```

```
q = -0.0000 + 3.1416i
```

Examine Absolute Error

Use `quadgk` to evaluate an oscillatory integrand that is difficult to evaluate.

Evaluate the integral

$$Q = \int_0^{\pi} \sin(20000\pi x)\mathrm{dx}.$$

The integrand oscillates very quickly, so it is difficult to evaluate. Use `quadgk` to evaluate the integral, and specify two outputs to examine how close the error tolerances are to being met.

```
fun = @(x) sin(2e4*pi*x);
[Q,errbnd] = quadgk(fun,0,pi)
```

```
Warning: Reached the limit on the maximum number of intervals in use. Approximate bound
```

```
Q = -0.0082
```

```
errbnd = 0.5723
```

The warning message indicates how to adjust `MaxIntervalCount` to allow for another iteration in the solution process.

Solve the integral again, but specify `MaxIntervalCount` as `1e5`. With many more intervals, `quadgk` is able to meet the absolute error tolerance for the problem (`1e-10` for double precision).

```
[Q,errbnd] = quadgk(fun,0,pi,'MaxIntervalCount',1e5)
```

```
Q = 1.6656e-06
```

```
errbnd = 2.6323e-12
```

Input Arguments

`fun` — Integrand
function handle

Integrand, specified as a function handle that defines the function to be integrated from `a` to `b`.

For scalar-valued problems, the function `y = fun(x)` must accept a vector argument `x` and return a vector result `y`, where `y` is the integrand evaluated at each element of `x`. This requirement generally means that `fun` must use array operators (`.^`, `.*`, ...) instead of matrix operators (`^`, `*`, ...).

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

Example: `q = quadgk(@(x) exp(1-x.^2),a,b)` integrates an anonymous function handle.

Example: `q = quadgk(@myFun,a,b)` integrates the function `myFun`, which is saved as a file.

Data Types: `function_handle`

a, b — Integration limits (as separate arguments)
scalars

Integration limits, specified as separate arguments of real or complex scalars. The limits `a` and `b` can be `-Inf` or `Inf`. If both are finite, they can be complex. If at least one is complex, the integral is approximated over a straight line path from `a` to `b` in the complex plane.

Example: `quadgk(fun,0,1)` integrates `fun` from `0` to `1`.

Data Types: `single` | `double`
Complex Number Support: Yes

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `q = quadgk(fun,a,b,'Waypoints',[0.1 1.1 2.1])` uses the `'Waypoints'` option to specify a few points of interest where the integrand should be evaluated.

`AbsTol` — Absolute error tolerance
`1e-10` (double), `1e-5` (single) (default) | nonnegative real number

Absolute error tolerance, specified as the comma-separated pair consisting of `'AbsTol'` and a nonnegative real number. `quadgk` uses the absolute error tolerance to limit an estimate of the absolute error, `|q − I|`, where `q` is the computed value of the integral and `I` is the (unknown) exact value. `quadgk` might provide more decimal places of precision if you decrease the absolute error tolerance.

`quadgk` attempts to satisfy

```
errbnd <= max(AbsTol,RelTol*abs(q))
```

This relation is absolute error control when `|q|` is sufficiently small and relative error control when `|q|` is larger. For pure absolute error control, use `'AbsTol' > 0` and `'RelTol'= 0`. For pure relative error control use `'RelTol' > 0` and `'AbsTol' = 0`.

Except when using pure absolute error control, the minimum relative tolerance is `'RelTol' >= 100*eps(class(q))`.

Example: `quadgk(fun,a,b,'AbsTol',1e-12)` sets the absolute error tolerance to approximately 12 decimal places of accuracy.

Example: `quadgk(fun,a,b,'AbsTol',tol,'RelTol',0)` uses a purely absolute error control, requiring that `errbnd <= tol`.

Data Types: `single` | `double`

`RelTol` — Relative error tolerance
`1e-6` (double), `1e-4` (single) (default) | nonnegative real number

Relative error tolerance, specified as the comma-separated pair consisting of `'RelTol'` and a nonnegative real number. `quadgk` uses the relative error tolerance to limit an estimate of the relative error, `|q - I|/|I|`, where `q` is the computed value of the integral and `I` is the (unknown) exact value. `quadgk` might provide more significant digits of precision if you decrease the relative error tolerance.

`quadgk` attempts to satisfy

```
errbnd <= max(AbsTol,RelTol*abs(q))
```

This relation is absolute error control when `|q|` is sufficiently small and relative error control when `|q|` is larger. For pure absolute error control, use `'AbsTol' > 0` and `'RelTol'= 0`. For pure relative error control use `'RelTol' > 0` and `'AbsTol' = 0`. Except when using pure absolute error control, the minimum relative tolerance is `'RelTol' >= 100*eps(class(q))`.

Example: `quadgk(fun,a,b,'RelTol',1e-9)` sets the relative error tolerance to approximately 9 significant digits.

Example: `quadgk(fun,a,b,'AbsTol',0,'RelTol',tol)` uses a purely relative error tolerance, requiring that `errbnd <= |I|*tol`.

Data Types: `single` | `double`

`Waypoints` — Integration waypoints
vector

Integration waypoints, specified as the comma-separated pair consisting of `'Waypoints'` and a vector of real or complex numbers. Use waypoints to indicate points in the integration interval that you would like the integrator to use in the initial mesh:

- Add more evaluation points near interesting features of the function, such as a local extrema.
- Integrate efficiently across discontinuities of the integrand by specifying the locations of the discontinuities.
- Perform complex contour integrations by specifying complex numbers as waypoints. If `xmin`, `xmax`, or any entry of the waypoints vector is complex, then the integration is performed over a sequence of straight line paths in the complex plane. In this case, all of the integration limits and waypoints must be finite.

Do not use waypoints to specify singularities. Instead, split the interval and add the results of separate integrations with the singularities at the endpoints.

Example: `'Waypoints',[1+1i,1-1i]` specifies two complex waypoints along the interval of integration.

Data Types: `single` | `double`
Complex Number Support: Yes

`MaxIntervalCount` — Maximum number of intervals allowed
`650` (default) | scalar

Maximum number of intervals allowed, specified as a scalar. This option limits the number of intervals that `quadgk` uses at any one time after the first iteration. A warning is issued if `quadgk` returns early because of this limit. Routinely increasing this value is not recommended, but it may be appropriate when `errbnd` is small enough that the desired accuracy has nearly been achieved.

Example: `quadgk(fun,a,b,'MaxIntervalCount',700)`

Output Arguments

`q` — Value of integral
scalar

Value of integral, returned as a scalar.

`errbnd` — Approximate upper bound on absolute error
scalar

Approximate upper bound on absolute error, returned as a scalar. The approximate upper bound on absolute error in the integration is `errbnd = |q − I|`, where `q` is the

computed value of the integral and `I` is the (unknown) exact value. `quadgk` attempts to satisfy

```
errbnd <= max(AbsTol,RelTol*abs(q))
```

Specify this output argument to see how well the integration meets the `AbsTol` and `RelTol` error tolerances. In cases where `errbnd` is close to the desired value, you might be able to reach the desired value by increasing the value of `MaxIntervalCount`.

Tips

- `quadgk` and `integral` use essentially the same integration method. You should generally use `integral` rather than `quadgk`. However, you can use `quadgk` to:
  - Monitor solution accuracy with the `errbnd` output argument.
  - Specify a large value for `MaxIntervalCount` when `integral` warns about reaching the maximum number of intervals.
- `quadgk` can integrate functions that are singular at finite endpoints if the singularities are not too strong. For example, it can integrate functions that behave at an endpoint `c` like `log|x-c|` or $|x-c|^p$ for `p >= -1/2`. If the function is singular at points inside the integration limits `[a b]`, then write the integral as a sum of integrals over subintervals with the singular points as endpoints, compute them with `quadgk`, and add the results.
- If the interval is infinite, $[a, \infty)$, then for the integral of `fun(x)` to exist, `fun(x)` must decay as `x` approaches infinity, and `quadgk` requires it to decay rapidly.

References

[1] Shampine, L.F. "Vectorized Adaptive Quadrature in MATLAB." *Journal of Computational and Applied Mathematics*. Vol. 211, 2008, pp.131–140.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`integral` | `integral2` | `integral3` | `quad2d`

Topics

"Anonymous Functions"
"Create Function Handle"

Introduced in R2007b

quadl

(Not recommended) Numerically evaluate integral, adaptive Lobatto quadrature

Note `quadl` is not recommended. Use `integral` instead.

Syntax

```
q = quadl(fun,a,b)
q = quadl(fun,a,b,tol)
quadl(fun,a,b,tol,trace)
[q,fcnt] = quadl(...)
```

Description

`q = quadl(fun,a,b)` approximates the integral of function `fun` from `a` to `b`, to within an error of 10^{-6} using recursive adaptive Lobatto quadrature. `fun` is a function handle. It accepts a vector `x` and returns a vector `y`, the function `fun` evaluated at each element of `x`. Limits `a` and `b` must be finite.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

`q = quadl(fun,a,b,tol)` uses an absolute error tolerance of `tol` instead of the default, which is `1.0e-6`. Larger values of `tol` result in fewer function evaluations and faster computation, but less accurate results.

`quadl(fun,a,b,tol,trace)` with non-zero `trace` shows the values of `[fcnt a b-a q]` during the recursion.

`[q,fcnt] = quadl(...)` returns the number of function evaluations.

Use array operators `.*`, `./` and `.^` in the definition of `fun` so that it can be evaluated with a vector argument.

The function `quad` might be more efficient with low accuracies or nonsmooth integrands.

The list below contains information to help you determine which quadrature function in MATLAB to use:

- The `quad` function might be most efficient for low accuracies with nonsmooth integrands.
- The `quadl` function might be more efficient than `quad` at higher accuracies with smooth integrands.
- The `quadgk` function might be most efficient for high accuracies and oscillatory integrands. It supports infinite intervals and can handle moderate singularities at the endpoints. It also supports contour integration along piecewise linear paths.
- The `quadv` function vectorizes `quad` for an array-valued `fun`.
- If the interval is infinite, $[a, \infty)$, then for the integral of `fun(x)` to exist, `fun(x)` must decay as `x` approaches infinity, and `quadgk` requires it to decay rapidly. Special methods should be used for oscillatory functions on infinite intervals, but `quadgk` can be used if `fun(x)` decays fast enough.
- The `quadgk` function will integrate functions that are singular at finite endpoints if the singularities are not too strong. For example, it will integrate functions that behave at an endpoint `c` like `log|x-c|` or $|x-c|^p$ for `p >= -1/2`. If the function is singular at points inside `(a,b)`, write the integral as a sum of integrals over subintervals with the singular points as endpoints, compute them with `quadgk`, and add the results.

Examples

Pass the function handle, `@myfun`, to `quadl`:

```
Q = quadl(@myfun,0,2);
```

where the function `myfun.m` is:

```
function y = myfun(x)
y = 1./(x.^3-2*x-5);
```

Pass anonymous function handle `F` to `quadl`:

```
F = @(x) 1./(x.^3-2*x-5);
Q = quadl(F,0,2);
```

Diagnostics

`quadl` might issue one of the following warnings:

`'Minimum step size reached'` indicates that the recursive interval subdivision has produced a subinterval whose length is on the order of roundoff error in the length of the original interval. A nonintegrable singularity is possible.

`'Maximum function count exceeded'` indicates that the integrand has been evaluated more than 10,000 times. A nonintegrable singularity is likely.

`'Infinite or Not-a-Number function value encountered'` indicates a floating point overflow or division by zero during the evaluation of the integrand in the interior of the interval.

Algorithms

`quadl` implements a high order method using an adaptive Gauss/Lobatto quadrature rule.

References

[1] Gander, W. and W. Gautschi, “Adaptive Quadrature – Revisited,” BIT, Vol. 40, 2000, pp. 84-101. This document is also available at `https://www.inf.ethz.ch/personal/gander`.

See Also

`dblquad` | `integral` | `integral2` | `integral3` | `quad` | `quad2d` | `quadgk` | `triplequad`

Topics

“Anonymous Functions”
“Create Function Handle”

Introduced before R2006a

quadv

(Not recommended) Vectorized quadrature

Note `quadv` is not recommended. Use `integral` with the `'ArrayValued'` option instead.

Syntax

```
Q = quadv(fun,a,b)
Q = quadv(fun,a,b,tol)
Q = quadv(fun,a,b,tol,trace)
[Q,fcnt] = quadv(...)
```

Description

`Q = quadv(fun,a,b)` approximates the integral of the complex array-valued function `fun` from `a` to `b` to within an error of `1.e-6` using recursive adaptive Simpson quadrature. `fun` is a function handle. The function `Y = fun(x)` should accept a scalar argument `x` and return an array result `Y`, whose components are the integrands evaluated at `x`. Limits `a` and `b` must be finite.

"Parameterizing Functions" explains how to provide addition parameters to the function `fun`, if necessary.

`Q = quadv(fun,a,b,tol)` uses the absolute error tolerance `tol` for all the integrals instead of the default, which is `1.e-6`.

Note The same tolerance is used for all components, so the results obtained with `quadv` are usually not the same as those obtained with `quad` on the individual components.

`Q = quadv(fun,a,b,tol,trace)` with non-zero `trace` shows the values of `[fcnt a b-a Q(1)]` during the recursion.

`[Q,fcnt] = quadv(...)` returns the number of function evaluations.

The list below contains information to help you determine which quadrature function in MATLAB to use:

- The `quad` function might be most efficient for low accuracies with nonsmooth integrands.
- The `quadl` function might be more efficient than `quad` at higher accuracies with smooth integrands.
- The `quadgk` function might be most efficient for high accuracies and oscillatory integrands. It supports infinite intervals and can handle moderate singularities at the endpoints. It also supports contour integration along piecewise linear paths.
- The `quadv` function vectorizes `quad` for an array-valued `fun`.
- If the interval is infinite, $[a, \infty)$, then for the integral of `fun(x)` to exist, `fun(x)` must decay as `x` approaches infinity, and `quadgk` requires it to decay rapidly. Special methods should be used for oscillatory functions on infinite intervals, but `quadgk` can be used if `fun(x)` decays fast enough.
- The `quadgk` function will integrate functions that are singular at finite endpoints if the singularities are not too strong. For example, it will integrate functions that behave at an endpoint `c` like `log|x-c|` or $|x-c|^p$ for `p >= -1/2`. If the function is singular at points inside `(a,b)`, write the integral as a sum of integrals over subintervals with the singular points as endpoints, compute them with `quadgk`, and add the results.

Examples

For the parameterized array-valued function `myarrayfun`, defined by

```
function Y = myarrayfun(x,n)
Y = 1./((1:n)+x);
```

the following command integrates `myarrayfun`, for the parameter value n = 10 between a = 0 and b = 1:

```
Qv = quadv(@(x)myarrayfun(x,10),0,1);
```

The resulting array `Qv` has 10 elements estimating `Q(k) = log((k+1)./(k))`, for `k = 1:10`.

The entries in `Qv` are slightly different than if you compute the integrals using `quad` in a loop:

```
for k = 1:10
    Qs(k) = quadv(@(x)myscalarfun(x,k),0,1);
end
```

where `myscalarfun` is:

```
function y = myscalarfun(x,k)
y = 1./(k+x);
```

See Also

`dblquad` | `integral` | `integral2` | `integral3` | `quad` | `quad2d` | `quadgk` | `quadl` | `triplequad`

Topics

"Create Function Handle"

Introduced before R2006a

matlab.unittest.qualifications Package

Summary of classes in MATLAB Qualifications Interface

Description

Qualifications are methods for testing values and responding to failures. Qualification failures might or might not correspond to a test failure, and they might or might not continue execution in the test when one is encountered. To determine which qualification to use, see “Table of Verifications, Assertions, and Other Qualifications”.

Classes

Qualification Classes

matlab.unittest.qualifications.Assertable	Qualification to validate preconditions of a test
matlab.unittest.qualifications.Assumable	Qualification to filter test content
matlab.unittest.qualifications.FatalAssertable	Qualification to abort test execution
matlab.unittest.qualifications.Verifiable	Qualification to produce soft failure conditions

Event Data Classes

matlab.unittest.qualifications.ExceptionEventData	Event data for ExceptionThrown event listeners
matlab.unittest.qualifications.QualificationEventData	Event data for qualification event listeners

Exception Handling Classes

matlab.unittest.qualifications.AssertionFailedException	Exception used for assertion failures
matlab.unittest.qualifications.AssumptionFailedException	Exception used for assumption failures
matlab.unittest.qualifications.FatalAssertionFailedException	Exception used for fatal assertion failures

See Also

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

matlab.unittest.qualifications.Assertable class

Package: `matlab.unittest.qualifications`

Qualification to validate preconditions of a test

Description

The `Assertable` class provides a qualification to validate preconditions of a test. Apart from actions performed for failures, the `Assertable` class works the same as other `matlab.unittest` qualifications.

Upon an assertion failure, the `Assertable` class throws an `AssertionFailedException` to inform the testing framework of the failure. This is most useful when a failure at the assertion point renders the rest of the current test method invalid, yet does not prevent proper execution of other test methods. Often, you use assertions to ensure that preconditions of the current test are not violated or that fixtures are set up correctly. Make sure the test content is “Exception Safe” on page 1-12249. If you cannot make the fixture teardown exception safe or if you cannot recover it after failure, use fatal assertions instead.

Use assertions to allow remaining test methods to receive coverage when preconditions are violated in a test and all fixture states are restorable. Assertions also reduce the noise level of failures by not performing later verifications that fail due the precondition failures. In the event of a failure, however, the test framework marks the full content of the test method that failed as incomplete. Therefore, if the failure does not affect the preconditions of the test or cause problems with fixture setup or teardown, use verifications, which give the added information that the full test content was run.

Methods

assertClass	Assert exact class of specified value
assertEmpty	Assert value is empty
assertEqual	Assert value is equal to specified value
assertError	Assert function throws specified exception
assertFail	Produce unconditional assertion failure
assertFalse	Assert value is false
assertGreaterThan	Assert value is greater than specified value
assertGreaterThanOrEqual	Assert value is greater than or equal to specified value
assertInstanceOf	Assert value is object of specified type
assertLength	Assert value has specified length
assertLessThan	Assert value is less than specified value
assertLessThanOrEqual	Assert value is less than or equal to specified value
assertMatches	Assert string matches specified regular expression
assertNotEmpty	Assert value is not empty
assertNotEqual	Assert value is not equal to specified value
assertNotSameHandle	Assert value is not handle to specified instance
assertNumElements	Assert value has specified element count
assertReturnsTrue	Assert function returns true when evaluated
assertSameHandle	Assert two values are handles to same instance
assertSize	Assert value has specified size
assertSubstring	Assert string contains specified string
assertThat	Assert that value meets specified constraint
assertTrue	Assert value is true
assertWarning	Assert function issues specified warnings
assertWarningFree	Assert function issues no warnings

Events

`AssertionFailed`	Triggered upon failing assertion. A `QualificationEventData` object is passed to listener callback functions.
`AssertionPassed`	Triggered upon passing assertion. A `QualificationEventData` object is passed to listener callback functions.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Test for Preconditions Using Assertions

Use assertable qualifications to test for preconditions. This example will create a test case to write a polynomial to a MAT-file.

Create `DocPolynomSaveLoadTest` Test Case. Refer to the following `DocPolynomSaveLoadTest` test case in the subsequent steps in this example. The steps highlight specific code in the `testSaveLoad` function; the code statements are not intended to be executed outside the context of the class definition file.

DocPolynomSaveLoadTest Class Definition File

```
classdef DocPolynomSaveLoadTest < matlab.unittest.TestCase

    methods (TestClassSetup)
        function addDocPolynomClassToPath(testCase)
            origPath = path;
            testCase.addTeardown(@path, origPath);
            addpath(fullfile(matlabroot, ...
                'help', 'techdoc', 'matlab_oop', 'examples'));
        end
    end
```

```
methods (Test)
    function testSaveLoad(testCase)

        import matlab.unittest.diagnostics.Diagnostic;

        %% Phase 1: Setup
        % Create a temporary working folder
        tempFolder = tempname;
        [success, message] = mkdir(tempFolder);
        testCase.assertTrue(success, ...
            Diagnostic.join('Could not create temporary folder.',...
            message));
        testCase.addTeardown(@() testCase.cleanUpTemporaryFolder(...
            tempFolder));

        % Change to the temporay folder and register the
        % teardown, which restores the original folder
        origFolder = pwd;
        testCase.addTeardown(@cd, origFolder);
        cd(tempFolder);

        %% Phase 2: Exercise
        % Save the instance to a mat file.
        p = DocPolynom([1, 0, 1]);
        save('DocPolynomFile', 'p');

        % Validate Precondition. Save resulted in valid .mat file
        testCase.assertEqual(exist('DocPolynomFile.mat','file'),...
            2, Diagnostic.join(...
            'mat file was not saved correctly.',@() dir(pwd)));

        loaded = load('DocPolynomFile');


        %% Phase 3: Verify
        testCase.verifyEqual(loaded.p, p,...
            'Loaded polynom did not equal original polynom.');


        %% Phase 4: Teardown
        % Done inline via calls to addTeardown at the points
        % at which the state was changed.
```

```
        end
    end

    methods(Access=private)
        function cleanUpTemporaryFolder(testCase,tempFolder)
            % Clean up the temporary folder and fatally assert
            % that it was correctly cleaned up.

            import matlab.unittest.diagnostics.Diagnostic;

            [success, message] = rmdir(tempFolder, 's');
            testCase.fatalAssertTrue(success, ...
                Diagnostic.join('Could not remove temporary folder.',...
                message));
        end
    end

end
```

To execute the MATLAB commands in “Run DocPolynomSaveLoadTest Test Case”, add the `DocPolynomSaveLoadTest.m` file to a folder on your MATLAB path.

The `testSaveLoad` function consists of the following phases:

- Phase 1: Setup — Create and verify precondition code.
- Phase 2: Exercise — Create a `DocPolynom` object and save it to a MAT-file.
- Phase 3: Verify — Test that object was successfully saved.
- Phase 4: Teardown — Execute teardown code.

Define phase 1 precondition. For this test, use a temporary folder for creating a `DocPolynom` object. The precondition for continuing with this test is that the following commands execute successfully.

```
tempFolder = tempname;
[success, message] = mkdir(tempFolder);
```

Test the results of the `mkdir` function. Use the `assertTrue` method to test the `mkdir` `success` argument for errors. If an assertion occurs, the remainder of the `testSaveLoad` test method is invalid, and the test is marked `Incomplete`.

```
testCase.assertTrue(success, ...
    Diagnostic.join('Could not create the temporary folder.', ...
        message))
```

If the `mkdir` function fails, MATLAB displays the diagnostic message, `Could not create the temporary folder`, as well as the contents of the `mkdir` `message` argument.

Add teardown fixture code. Creating a temporary folder is setup code, which requires a corresponding call to the `rmdir` function to restore MATLAB to the original state. Use the `addTeardown` method to ensure the teardown code executes even when an exception is thrown in the middle of the test method. This makes the test Exception Safe.

```
testCase.addTeardown(@() testCase.cleanUpTemporaryFolder(tempFolder))
```

Place teardown code in the helper function. Although the `addTeardown` statement occurs in the same code block as the `mkdir` setup statement, the `cleanUpTemporaryFolder` code is executed in phase 4 of the test method.

In the `DocPolynomSaveLoadTest` test case, the helper function, `cleanUpTemporaryFolder`, executes the `rmdir` function.

Define the precondition for creating valid MAT-File. A precondition for verifying that the `DocPolynom` object was correctly saved and loaded is that the MAT-file, `DocPolynomFile.mat`, was successfully created. The following code in the `Phase 2: Exercise` block tests this condition. If an assertion occurs, the remainder of the `testSaveLoad` test method is invalid, and the test is marked `Failed` and `Incomplete`.

```
 testCase.assertEqual(exist('DocPolynomFile.mat','file'), 2, ...
     Diagnostic.join('The mat file was not saved correctly.', ...
     @() dir(pwd)))
```

If the file was not created, MATLAB displays the diagnostic message, `The mat file was not saved correctly`, as well as the contents of the temporary folder.

Run `DocPolynomSaveLoadTest` Test Case.

```
tc = DocPolynomSaveLoadTest;
run(tc);
```

```
Running DocPolynomSaveLoadTest
.
```

```
Done DocPolynomSaveLoadTest
__________
```

More About

Exception Safe

Test content is exception safe when all fixture teardown is performed with `addTeardown` or through the appropriate object destructors when a failure occurs. This ensures that the failure does not affect later testing due to stale fixtures.

This code is not exception safe. After an assertion failure, the test framework does not close the figure.

```
% Not exception safe
f = figure;
testCase.assertEqual(actual, expected);
close(f);
```

This code is exception safe because the test framework closes the figure in all cases.

```
% Exception safe
f = figure;
testCase.addTeardown(@close, f);
testCase.assertEqual(actual, expected);
```

However, tearing down a fixture using `addTeardown` does not guarantee code is exception safe. This code shows a failure in `assertEqual`.

```
% Not exception safe
f = figure;
testCase.assertEqual(actual, expected);
testCase.addTeardown(@close, f);
```

See Also

`Assumable` | `FatalAssertable` | `QualificationEventData` | `TestCase` | `Verifiable` | `matlab.unittest.qualifications`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assertClass

Class: `matlab.unittest.qualifications.Assertable`
Package: `matlab.unittest.qualifications`

Assert exact class of specified value

Syntax

```
assertClass(assertable,actual,className)
assertClass(assertable,actual,metaClass)
assertClass( ___ ,diagnostic)
```

Description

`assertClass(assertable,actual,className)` asserts that `actual` is a MATLAB value whose class is the class specified by `className`.

`assertClass(assertable,actual,metaClass)` asserts that `actual` is a MATLAB value whose class is the class specified by the `meta.class` instance `metaClass`. The instance must be an exact class match. See `assertInstanceOf` to assert inclusion in a class hierarchy.

`assertClass( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`className`

Name of class, specified as a character vector.

Default:

`metaClass`

An instance of `meta.class`.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyClass`, and replace calls to `verifyClass` with `assertClass`.

Tips

- The method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsOfClass;
  assertable.assertThat(actual, IsOfClass(className));
  assertable.assertThat(actual, IsOfClass(metaClass));
  ```

  There exists more functionality when using the `IsOfClass` constraint directly via `assertThat`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`assertInstanceOf` | `assertThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assertEmpty

Class: `matlab.unittest.qualifications.Assertable`
Package: `matlab.unittest.qualifications`

Assert value is empty

Syntax

```
assertEmpty(assertable,actual)
assertEmpty(assertable,actual,diagnostic)
```

Description

`assertEmpty(assertable,actual)` asserts that `actual` is an empty MATLAB value.

`assertEmpty(assertable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyEmpty`, and replace calls to `verifyEmpty` with `assertEmpty`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEmpty;
  assertable.assertThat(actual, IsEmpty());
  ```

  This method is a convenience method. There exists more functionality when using the `IsEmpty` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort

testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`assertNotEmpty` | `assertThat` | `isempty`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assertEqual

Class: `matlab.unittest.qualifications.Assertable`
Package: `matlab.unittest.qualifications`

Assert value is equal to specified value

Syntax

```
assertEqual(assertable,actual,expected)
assertEqual( ___ ,Name,Value)
assertEqual( ___ ,diagnostic)
```

Description

`assertEqual(assertable,actual,expected)` asserts that `actual` is strictly equal to `expected`. If `expected` is not a MATLAB or Java object, `actual` and `expected` must have the same class, size, and value for the assertion to pass. `assertEqual` compares `actual` and `expected` in the same way as the `IsEqualTo` constraint.

`assertEqual( ___ ,Name,Value)` asserts equality with additional options specified by one or more `Name,Value` pair arguments.

`assertEqual( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expected`

Expected value.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`AbsTol`

Absolute tolerance, specified as a numeric array. The tolerance is applied only to values of the same data type. The value can be a scalar or array the same size as the actual and expected values.

For an absolute tolerance to be satisfied, `abs(expected-actual) <= absTol` must be true.

Default:

`RelTol`

Relative tolerance, specified as a numeric array. The tolerance is applied only to values of the same data type. The value can be a scalar or array the same size as the actual and expected values.

For a relative tolerance to be satisfied, `abs(expected-actual) <= relTol.*abs(expected)` must be true.

Default:

Examples

Compare Numeric Values

Numeric values are equal if they are of the same class with equivalent size, complexity, and sparsity.

At the command prompt, create a test case for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Assert that a numeric value is equal to itself.

```
assertEqual(testCase,5,5)
```

```
Assertion passed.
```

Assert if actual and expected values have equal sizes.

```
assertEqual(testCase,[5 5],5)
```

```
Assertion failed.
    ---------------------
    Framework Diagnostic:
    ---------------------
    assertEqual failed.
    --> Sizes do not match.

        Actual size:
             1     2
        Expected size:
```

```
             1     1

    Actual Value:
         5     5
    Expected Value:
         5
Assertion failed.
```

Test Classes

Assert if an `int8` is equal to an `int16`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
assertEqual(testCase,int8(5),int16(5))
```

```
Assertion failed.
    ---------------------
    Framework Diagnostic:
    ---------------------
    assertEqual failed.
    --> Classes do not match.

        Actual Class:
            int8
        Expected Class:
            int16

    Actual Value:
      int8

       5
    Expected Value:
      int16

       5
Assertion failed.
```

Test Cell Arrays

Elements of equal cell arrays must match in class, size, and value.

Assert that a cell array is equal to itself.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
assertEqual(testCase,{'cell',struct,5},{'cell',struct,5})
```

```
Assertion passed.
```

Test Numeric Tolerances

Assert if `4.95` is equal to `5`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
assertEqual(testCase,4.95,5)
```

```
Assertion failed.
    --------------------
    Framework Diagnostic:
    --------------------
    assertEqual failed.
    --> The numeric values are not equal using "isequaln".
    --> Failure table:
            Actual    Expected           Error              RelativeError
            ______    ________    __________________    ___________________

             4.95        5        -0.0499999999999998    -0.00999999999999996

    Actual Value:
       4.950000000000000
    Expected Value:
         5
Assertion failed.
```

Assert that the difference between the actual value `1.5` and the expected value `2` is within `1`.

```
assertEqual(testCase,1.5,2,'AbsTol',1)
```

```
Assertion passed.
```

Assert if the difference between the actual and expected values is less than 10%. Display the diagnostic information upon failure.

```
assertEqual(testCase,1.5,2,'RelTol',0.1,...
    'Difference between actual and expected exceeds relative tolerance')
```

```
Assertion failed.
    ----------------
    Test Diagnostic:
    ----------------
    Difference between actual and expected exceeds relative tolerance
    --------------------
    Framework Diagnostic:
    --------------------
```

```
    assertEqual failed.
    --> The numeric values are not equal using "isequaln".
    --> The error was not within relative tolerance.
    --> Failure table:
            Actual    Expected    Error    RelativeError    RelativeTolerance
            ______    ________    _____    _____________    _________________

             1.5         2        -0.5         -0.25               0.1

    Actual Value:
       1.500000000000000
    Expected Value:
         2
Assertion failed.
```

Tips

- This method is functionally equivalent to any of the following:

  ```
  import matlab.unittest.constraints.IsEqualTo;
  assertable.assertThat(actual, IsEqualTo(expected));
  ```

  ```
  import matlab.unittest.constraints.IsEqualTo;
  import matlab.unittest.constraints.AbsoluteTolerance;
  assertable.assertThat(actual, IsEqualTo(expected, ...
      'Within', AbsoluteTolerance(abstol)));
  ```

  ```
  import matlab.unittest.constraints.IsEqualTo;
  import matlab.unittest.constraints.RelativeTolerance;
  assertable.assertThat(actual, IsEqualTo(expected, ...
      'Within', RelativeTolerance(reltol)));
  ```

  ```
  import matlab.unittest.constraints.IsEqualTo;
  import matlab.unittest.constraints.AbsoluteTolerance;
  import matlab.unittest.constraints.RelativeTolerance;
  assertable.assertThat(actual, IsEqualTo(expected, ...
      'Within', AbsoluteTolerance(abstol) | RelativeTolerance(reltol)));
  ```

  There exists more functionality when using the `IsEqualTo`, `AbsoluteTolerance`, and `RelativeTolerance` constraints directly via `assertThat`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to

completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`assertNotEqual` | `assertThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assertError

Class: `matlab.unittest.qualifications.Assertable`
Package: `matlab.unittest.qualifications`

Assert function throws specified exception

Syntax

```
assertError(assertable,actual,identifier)
assertError(assertable,actual,metaClass)
assertError( ___ ,diagnostic)
[output1,...,outputN] = assertError( ___ )
```

Description

`assertError(assertable,actual,identifier)` asserts that `actual` is a function handle that throws an exception with an error identifier that is equal to `identifier`.

`assertError(assertable,actual,metaClass)` asserts that `actual` is a function handle that throws an exception whose type is defined by the `meta.class` instance specified in `metaClass`. This method does not require the instance to be an exact class match, but rather it must be in the specified class hierarchy, and that hierarchy must include the `MException` class.

`assertError( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

`[output1,...,outputN] = assertError( ___ )` returns multiple output arguments from the invocation of the function handle `actual`. Use this syntax to control the number of output arguments requested from the function handle. If the function handle errors, all outputs are of type `missing`. Otherwise, `output1,...,outputN` are output values from `actual`. You can use this syntax with any of the input arguments of the previous syntaxes.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`identifier`

Error identifier, specified as a character vector.

Default:

`metaClass`

An instance of `meta.class`.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyError`, and replace calls to `verifyError` with `assertError`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.Throws;
  assertable.assertThat(actual, Throws(identifier));
  assertable.assertThat(actual, Throws(metaClass));
  ```

  There exists more functionality when using the `Throws` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`MException` | `assertThat` | `assertWarning` | `error`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assertFail

Produce unconditional assertion failure

Syntax

```
assertFail(assertable)
assertFail(assertable,diagnostic)
```

Description

`assertFail(assertable)` produces an unconditional assertion failure when encountered.

`assertFail(assertable,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyFail`, and replace calls to `verifyFail` with `assertFail`.

Tips

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

Topics

“Table of Verifications, Assertions, and Other Qualifications”

Introduced in R2013a

assertFalse

Class: `matlab.unittest.qualifications.Assertable`
Package: `matlab.unittest.qualifications`

Assert value is false

Syntax

```
assertFalse(assertable,actual)
assertFalse(assertable,actual,diagnostic)
```

Description

`assertFalse(assertable,actual)` asserts that `actual` is a scalar logical with the value of false.

`assertFalse(assertable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

For examples, see `verifyFalse`, and replace calls to `verifyFalse` with `assertFalse`.

Tips

- This method passes if and only if the actual value is a scalar logical with a value of false. Therefore, entities such as empty arrays, false valued arrays, and zero doubles produce failures when used in this method, despite these entities exhibiting "false-like" behavior such as bypassing the execution of code inside of "if" statements.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsFalse;
  assertable.assertThat(actual, IsFalse());
  ```

  There exists more functionality when using the `IsFalse` constraint directly via `assertThat`.
- Unlike `assertTrue`, this method may create a new constraint for each call. For performance critical uses, consider using `assertTrue`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or

incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`assertThat` | `assertTrue`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assertGreaterThan

Class: `matlab.unittest.qualifications.Assertable`
Package: `matlab.unittest.qualifications`

Assert value is greater than specified value

Syntax

```
assertGreaterThan(assertable,actual,floor)
assertGreaterThan(assertable,actual,floor,diagnostic)
```

Description

`assertGreaterThan(assertable,actual,floor)` asserts that all elements of `actual` are greater than all the elements of `floor`.

`assertGreaterThan(assertable,actual,floor,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test. `actual` must be the same size as `floor` unless either one is scalar, at which point scalar expansion occurs.

`floor`

Minimum value, exclusive.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyGreaterThan`, and replace calls to `verifyGreaterThan` with `assertGreaterThan`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsGreaterThan;
  assertable.assertThat(actual, IsGreaterThan(floor));
  ```

  There exists more functionality when using the `IsGreaterThan` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or

incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`assertGreaterThanOrEqual` | `assertLessThan` | `assertLessThanOrEqual` | `assertThat` | `gt` | `matlab.unittest.constraints.IsGreaterThan` | `matlab.unittest.diagnostics.Diagnostic`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assertGreaterThanOrEqual

Class: `matlab.unittest.qualifications.Assertable`
Package: `matlab.unittest.qualifications`

Assert value is greater than or equal to specified value

Syntax

```
assertGreaterThanOrEqual(assertable,actual,floor)
assertGreaterThanOrEqual(assertable,actual,floor,diagnostic)
```

Description

`assertGreaterThanOrEqual(assertable,actual,floor)` asserts that all elements of `actual` are greater than or equal to all the elements of `floor`.

`assertGreaterThanOrEqual(assertable,actual,floor,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test. `actual` must be the same size as `floor` unless either one is scalar, at which point scalar expansion occurs.

`floor`

Minimum value.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyGreaterThanOrEqual`, and replace calls to `verifyGreaterThanOrEqual` with `assertGreaterThanOrEqual`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsGreaterThanOrEqualTo;
  assertable.assertThat(actual, IsGreaterThanOrEqualTo(floor));
  ```

  There exists more functionality when using the `IsGreaterThanOrEqualTo` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or

incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`assertGreaterThan` | `assertLessThan` | `assertLessThanOrEqual` | `assertThat` | `ge` | `matlab.unittest.constraints.IsGreaterThanOrEqualTo` | `matlab.unittest.diagnostics.Diagnostic`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assertInstanceOf

Class: `matlab.unittest.qualifications.Assertable`
Package: `matlab.unittest.qualifications`

Assert value is object of specified type

Syntax

```
assertInstanceOf(assertable,actual,className)
assertInstanceOf(assertable,actual,metaClass)
assertInstanceOf( ___ ,diagnostic)
```

Description

`assertInstanceOf(assertable,actual,className)` asserts that `actual` is a MATLAB value whose class is the class specified by `className`.

`assertInstanceOf(assertable,actual,metaClass)` asserts that `actual` is a MATLAB value whose class is the class specified by the `meta.class` instance `metaClass`.

`assertInstanceOf( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`className`

Name of class, specified as a character vector.

Default:

`metaClass`

An instance of `meta.class`.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyInstanceOf`, and replace calls to `verifyInstanceOf` with `assertInstanceOf`.

Tips

- This method is functionally equivalent to:

  ```
  assertable.assertThat(actual, IsInstanceOf(className));
  assertable.assertThat(actual, IsInstanceOf(metaClass));
  ```

  There exists more functionality when using the `IsInstanceOf` constraint directly via `assertThat`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`assertClass` | `assertThat` | `isa`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assertLength

Class: `matlab.unittest.qualifications.Assertable`
Package: `matlab.unittest.qualifications`

Assert value has specified length

Syntax

```
assertLength(assertable,actual,expectedLength)
assertLength(assertable,actual,expectedLength,diagnostic)
```

Description

`assertLength(assertable,actual,expectedLength)` that `actual` is a MATLAB array whose length is `expectedLength`.

`assertLength(assertable,actual,expectedLength,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedLength`

The length of an array is defined as the largest dimension of that array.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyLength`, and replace calls to `verifyLength` with `assertLength`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasLength;
  assertable.assertThat(actual, HasLength(expectedLength));
  ```

  There exists more functionality when using the `HasLength` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or

incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`assertNumElements` | `assertSize` | `assertThat` | `length`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assertLessThan

Class: `matlab.unittest.qualifications.Assertable`
Package: `matlab.unittest.qualifications`

Assert value is less than specified value

Syntax

```
assertLessThan(assertable,actual,ceiling)
assertLessThan(assertable,actual,ceiling,diagnostic)
```

Description

`assertLessThan(assertable,actual,ceiling)` asserts that all elements of `actual` are less than all the elements of `ceiling`.

`assertLessThan(assertable,actual,ceiling,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test. `actual` must be the same size as `ceiling` unless either one is scalar, at which point scalar expansion occurs.

Default:

`ceiling`

Maximum value, exclusive.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyLessThan`, and replace calls to LessThan with `assertLessThan`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsLessThan;
  assertable.assertThat(actual, IsLessThan(ceiling));
  ```

  There exists more functionality when using the `IsLessThan` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or

incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`assertGreaterThan` | `assertGreaterThanOrEqual` | `assertLessThanOrEqual` | `assertThat` | `lt` | `matlab.unittest.constraints.IsLessThan` | `matlab.unittest.diagnostics.Diagnostic`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assertLessThanOrEqual

Class: `matlab.unittest.qualifications.Assertable`
Package: `matlab.unittest.qualifications`

Assert value is less than or equal to specified value

Syntax

```
assertLessThanOrEqual(assertable,actual,ceiling)
assertLessThanOrEqual(assertable,actual,ceiling,diagnostic)
```

Description

`assertLessThanOrEqual(assertable,actual,ceiling)` asserts that all elements of `actual` are less than or equal to all the elements of `ceiling`.

`assertLessThanOrEqual(assertable,actual,ceiling,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test. `actual` must be the same size as `ceiling` unless either one is scalar, at which point scalar expansion occurs.

Default:

`ceiling`

Maximum value.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyLessThanOrEqual`, and replace calls to `verifyLessThanOrEqual` with `assertLessThanOrEqual`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsLessThanOrEqualTo;
  assertable.assertThat(actual, IsLessThanOrEqualTo(ceiling));
  ```

  There exists more functionality when using the `IsLessThanOrEqualTo` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or

incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`assertGreaterThan` | `assertGreaterThanOrEqual` | `assertLessThan` | `assertThat` | `le` | `matlab.unittest.constraints.IsLessThanOrEqualTo` | `matlab.unittest.diagnostics.Diagnostic`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assertMatches

Class: `matlab.unittest.qualifications.Assertable`
Package: `matlab.unittest.qualifications`

Assert string matches specified regular expression

Syntax

```
assertMatches(assertable,actual,expression)
assertMatches(assertable,actual,expression,diagnostic)
```

Description

`assertMatches(assertable,actual,expression)` asserts that `actual` is a string scalar or character vector that matches the regular expression defined by `expression`.

`assertMatches(assertable,actual,expression,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expression`

The value to match, specified as a regular expression.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyMatches`, and replace calls to `verifyMatches` with `assertMatches`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.Matches;
  assertable.assertThat(actual, Matches(expression));
  ```

  There exists more functionality when using the `Matches` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or

incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`assertSubstring` | `assertThat` | `regexp`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assertNotEmpty

Class: `matlab.unittest.qualifications.Assertable`
Package: `matlab.unittest.qualifications`

Assert value is not empty

Syntax

```
assertNotEmpty(assertable,actual)
assertNotEmpty(assertable,actual,diagnostic)
```

Description

`assertNotEmpty(assertable,actual)` asserts that `actual` is a non-empty MATLAB value.

`assertNotEmpty(assertable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyNotEmpty`, and replace calls to `verifyNotEmpty` with `assertNotEmpty`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEmpty;
  assertable.assertThat(actual, ~IsEmpty());
  ```

  There exists more functionality when using the `IsEmpty` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture

teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`assertEmpty` | `assertThat` | `isempty`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assertNotEqual

Class: `matlab.unittest.qualifications.Assertable`
Package: `matlab.unittest.qualifications`

Assert value is not equal to specified value

Syntax

```
assertNotEqual(assertable,actual,notExpected)
assertNotEqual(assertable,actual,notExpected,diagnostic)
```

Description

`assertNotEqual(assertable,actual,notExpected)` asserts that `actual` is not equal to `notExpected`.

`assertNotEqual(assertable,actual,notExpected,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`notExpected`

Value to compare.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyNotEqual`, and replace calls to `verifyNotEqual` with `assertNotEqual`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEqualTo;
  assertable.assertThat(actual, ~IsEqualTo(notExpected));
  ```

  There exists more functionality when using the `IsEqualTo` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or

incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`assertEqual` | `assertThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assertNotSameHandle

Class: `matlab.unittest.TestCase`
Package: `matlab.unittest`

Assert value is not handle to specified instance

Syntax

```
assertNotSameHandle(assertable,actual,notExpectedHandle)
assertNotSameHandle(assertable,actual,notExpectedHandle,diagnostic)
```

Description

`assertNotSameHandle(assertable,actual,notExpectedHandle)` asserts that `actual` is a different size and/or does not contain the same instances as the `notExpectedHandle` handle array.

`assertNotSameHandle(assertable,actual,notExpectedHandle,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`notExpectedHandle`

The handle array to compare.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyNotSameHandle`, and replace calls to `verifyNotSameHandle` with `assertNotSameHandle`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsSameHandleAs;
  assertable.assertThat(actual, ~IsSameHandleAs(notExpectedHandle));
  ```

  There exists more functionality when using the `IsSameHandleAs` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or

incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`assertSameHandle` | `assertThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assertNumElements

Class: `matlab.unittest.qualifications.Assertable`
Package: `matlab.unittest.qualifications`

Assert value has specified element count

Syntax

```
assertNumElements(assertable,actual,expectedElementCount)
assertNumElements(assertable,actual,expectedElementCount,diagnostic)
```

Description

`assertNumElements(assertable,actual,expectedElementCount)` asserts that `actual` is a MATLAB array with `expectedElementCount` number of elements.

`assertNumElements(assertable,actual,expectedElementCount,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedElementCount`

The expected number of elements in the array.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyNumElements`, and replace calls to `verifyNumElements` with `assertNumElements`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasElementCount;
  assertable.assertThat(actual, HasElementCount(expectedElementCount));
  ```

  There exists more functionality when using the `HasElementCount` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or

incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`assertLength` | `assertSize` | `assertThat` | `numel`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assertReturnsTrue

Class: `matlab.unittest.qualifications.Assertable`
Package: `matlab.unittest.qualifications`

Assert function returns true when evaluated

Syntax

```
assertReturnsTrue(assertable,actual)
assertReturnsTrue(assertable,actual,diagnostic)
```

Description

`assertReturnsTrue(assertable,actual)` asserts that `actual` is a function handle that returns a scalar logical whose value is true.

`assertReturnsTrue(assertable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The function handle to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyReturnsTrue`, and replace calls to `verifyReturnsTrue` with `assertReturnsTrue`.

Tips

- It is a shortcut for quick custom comparison functionality that can be defined quickly, and possibly inline. It can be preferable over simply evaluating the function directly and using `assertTrue` because the function handle will be shown in the diagnostics, thus providing more insight into the failure condition which is lost when using `assertTrue`.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.ReturnsTrue;
  assertable.assertThat(actual, ReturnsTrue());
  ```

  There exists more functionality when using the `ReturnsTrue` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or

incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`assertThat` | `assertTrue`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assertSameHandle

Class: `matlab.unittest.qualifications.Assertable`
Package: `matlab.unittest.qualifications`

Assert two values are handles to same instance

Syntax

```
assertSameHandle(assertable,actual,expectedHandle)
assertSameHandle(assertable,actual,expectedHandle,diagnostic)
```

Description

`assertSameHandle(assertable,actual,expectedHandle)` asserts that `actual` is the same size and contains the same instances as the `expectedHandle` handle array.

`assertSameHandle(assertable,actual,expectedHandle,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedHandle`

The expected handle array.

diagnostic

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifySameHandle`, and replace calls to `verifySameHandle` with `assertSameHandle`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsSameHandleAs;
  assertable.assertThat(actual, IsSameHandleAs(expectedHandle));
  ```

  There exists more functionality when using the `IsSameHandleAs` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`assertNotSameHandle` | `assertThat` | `handle`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assertSize

Class: `matlab.unittest.qualifications.Assertable`
Package: `matlab.unittest.qualifications`

Assert value has specified size

Syntax

```
assertSize(assertable,actual,expectedSize)
assertSize(assertable,actual,expectedSize,diagnostic)
```

Description

`assertSize(assertable,actual,expectedSize)` asserts that `actual` is a MATLAB array whose size is `expectedSize`.

`assertSize(assertable,actual,expectedSize,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedSize`

The expected sizes of each dimension the array.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifySize`, and replace calls to `verifySize` with `assertSize`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasSize;
  assertable.assertThat(actual, HasSize(expectedSize));
  ```

  There exists more functionality when using the `HasSize` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or

incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`assertLength` | `assertNumElements` | `assertThat` | `size`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assertSubstring

Class: `matlab.unittest.qualifications.Assertable`
Package: `matlab.unittest.qualifications`

Assert string contains specified string

Syntax

```
assertSubstring(assertable,actual,substring)
assertSubstring(assertable,actual,substring,diagnostic)
```

Description

`assertSubstring(assertable,actual,substring)` asserts that `actual` is a string scalar or character vector that contains `substring`.

`assertSubstring(assertable,actual,substring,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`substring`

The value to match, specified as a string scalar or character vector.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifySubstring`, and replace calls to `verifySubstring` with `assertSubstring`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.ContainsSubstring;
  assertable.assertThat(actual, ContainsSubstring(substring));
  ```

  There exists more functionality when using the `ContainsSubstring` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or

incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`assertMatches` | `assertThat` | `strfind`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assertThat

Class: `matlab.unittest.qualifications.Assertable`
Package: `matlab.unittest.qualifications`

Assert that value meets specified constraint

Syntax

```
assertThat(assertable,actual,constraint)
assertThat(assertable,actual,constraint,diagnostic)
```

Description

`assertThat(assertable,actual,constraint)` asserts that `actual` is a value that satisfies the `constraint` provided.

If the constraint is not satisfied, an assertion failure is produced utilizing only the framework diagnostic generated by the `constraint`.

`assertThat(assertable,actual,constraint,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

When using this signature, both the diagnostic information contained within `diagnostic` is used in addition to the diagnostic information provided by the `constraint`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`constraint`

Constraint that the actual value must satisfy to pass the verification, specified as a `matlab.unittest.constraints` instance.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyThat`, and replace calls to `verifyThat` with `assertThat`.

Tips

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assertTrue

Class: `matlab.unittest.qualifications.Assertable`
Package: `matlab.unittest.qualifications`

Assert value is true

Syntax

```
assertTrue(assertable,actual)
assertTrue(assertable,actual,diagnostic)
```

Description

`assertTrue(assertable,actual)` asserts that `actual` is a scalar logical with the value of true.

`assertTrue(assertable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyTrue`, and replace calls to `verifyTrue` with `assertTrue`.

Tips

- This method passes if and only if the actual value is a scalar logical with a value of true. Therefore, entities such as true valued arrays and non-zero doubles produce qualification failures when used in this method, despite these entities exhibiting "true-like" behavior such as triggering the execution of code inside of "if" statements.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsTrue;
  assertable.assertThat(actual, IsTrue());
  ```

  There exists more functionality when using the `IsTrue` constraint directly via `assertThat`.

  Use of this method for performance benefits can come at the expense of less diagnostic information, and may not provide the same level of strictness adhered to by other constraints such as `IsEqualTo`. A similar approach that is generally less performant but can provide slightly better diagnostic information is the use of `assertReturnsTrue`, which at least shows the display of the function evaluated to generate the failing result.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`assertFalse` | `assertReturnsTrue` | `assertThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assertWarning

Class: `matlab.unittest.qualifications.Assertable`
Package: `matlab.unittest.qualifications`

Assert function issues specified warnings

Syntax

```
assertWarning(assertable,actual,warningIDs)
assertWarning(assertable,actual,warningIDs,diagnostic)
[output1,...,outputN] = assertWarning( ___ )
```

Description

`assertWarning(assertable,actual,warningIDs)` asserts that `actual` issues warnings with the identifiers `warningIDs`.

`assertWarning(assertable,actual,warningIDs,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

`[output1,...,outputN] = assertWarning( ___ )` also returns the output arguments `output1,...,outputN` that are produced when invoking `actual`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The function handle to test.

Default:

`warningIDs`

Warning identifiers, specified as a character vector, cell array of character vectors, or string array.

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Output Arguments

`output1,...,outputN`

Output arguments, 1 through n (if any), from `actual`, returned as any type. The argument type is specified by the `actual` argument list.

Examples

See examples for `verifyWarning`, and replace calls to `verifyWarning` with `assertWarning`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IssuesWarnings;
  assertable.assertThat(actual,IssuesWarnings(cellstr(warningIDs)));
  ```

  There exists more functionality when using the `IssuesWarnings` constraint directly via `assertThat`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`assertError` | `assertThat` | `assertWarningFree` | `warning`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assertWarningFree

Class: `matlab.unittest.qualifications.Assertable`
Package: `matlab.unittest.qualifications`

Assert function issues no warnings

Syntax

```
assertWarningFree(assertable,actual)
assertWarningFree(assertable,actual,diagnostic)
[output1,...,outputN] = assertWarningFree( ___ )
```

Description

`assertWarningFree(assertable,actual)` asserts that `actual` is a function handle that issues no warnings.

`assertWarningFree(assertable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

`[output1,...,outputN] = assertWarningFree( ___ )` also returns the output arguments `output1,...,outputN` that are produced when invoking `actual`.

Input Arguments

`assertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assertion in conjunction with the test running framework.

`actual`

The function handle to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Output Arguments

`output1,...,outputN`

Output arguments, 1 through n (if any), from `actual`, returned as any type. The argument type is specified by the `actual` argument list.

Examples

See examples for `verifyWarningFree`, and replace calls to `verifyWarningFree` with `assertWarningFree`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IssuesNoWarnings;
  assertable.assertThat(actual, IssuesNoWarnings());
  ```

  There exists more functionality when using the `IssuesNoWarnings` constraint directly via `assertThat`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. Alternatively,

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`assertThat` | `assertWarning` | `warning`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

matlab.unittest.qualifications.AssertionFailedException class

Package: `matlab.unittest.qualifications`

Exception used for assertion failures

Description

The `AssertionFailedException` class provides an exception used for assertion failures. This class is used exclusively by the `Assertable` qualification type.

See Also

`Assertable` | `MException`

matlab.unittest.qualifications.Assumable class

Package: `matlab.unittest.qualifications`

Qualification to filter test content

Description

The `Assumable` class provides a qualification to filter test content. Apart from actions performed in the event of failures, the `Assumable` class works the same as other `matlab.unittest` qualifications.

Upon an assumption failure, the `Assumable` class informs the testing framework of the failure by throwing an `AssumptionFailedException`. The test framework then marks the test content as filtered and continues testing. Often, assumptions are used to ensure that the test is run only when certain preconditions are met. However, running the test without satisfying the preconditions does not produce a test failure. Ensure that the test content is "Exception Safe" on page 1-12335. If the failure condition is meant to produce a test failure, use assertions or verifications instead of assumptions.

The attributes specified in the `TestCase` method definition determine which tests are filtered. The following behavior occurs when the test framework encounters an assumption failure inside of a `TestCase` method:

- If you define the `TestCase` method using the `Test` attribute, the framework marks the entire method as filtered and runs subsequent test methods.
- If you define the `TestCase` method using the `TestMethodSetup` or `TestMethodTeardown` attributes, the test framework marks the method to run for that instance as filtered.
- If you define the `TestCase` method using the `TestClassSetup` or `TestClassTeardown` attributes, the test framework filters the entire `TestCase` class.

Filtering test content using assumptions does not produce test failures. Therefore, dead test code can result. Avoid this by monitoring filtered tests.

Methods

assumeClass	Assume exact class of specified value
assumeEmpty	Assume value is empty
assumeEqual	Assume value is equal to specified value
assumeError	Assume function throws specified exception
assumeFail	Produce unconditional assumption failure
assumeFalse	Assume value is false
assumeGreaterThan	Assume value is greater than specified value
assumeGreaterThanOrEqual	Assume value is greater than or equal to specified value
assumeInstanceOf	Assume value is object of specified type
assumeLength	Assume value has specified length
assumeLessThan	Assume value is less than specified value
assumeLessThanOrEqual	Assume value is less than or equal to specified value
assumeMatches	Assume string matches specified regular expression
assumeNotEmpty	Assume value is not empty
assumeNotEqual	Assume value is not equal to specified value
assumeNotSameHandle	Assume value is not handle to specified instance
assumeNumElements	Assume value has specified element count
assumeReturnsTrue	Assume function returns true when evaluated
assumeSameHandle	Assume two values are handles to same instance
assumeSize	Assume value has specified size
assumeSubstring	Assume string contains specified string
assumeThat	Assume value meets specified constraint
assumeTrue	Assume value is true
assumeWarning	Assume function issues specified warnings
assumeWarningFree	Assume function issues no warnings

Events

`AssumptionFailed`	Triggered upon failing assumption. A `QualificationEventData` object is passed to listener callback functions.
`AssumptionPassed`	Triggered upon passing assumption. A `QualificationEventData` object is passed to listener callback functions.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Write TestClassSetup Method Using Assumptions

Assumptions assure that a test runs only when certain preconditions are satisfied and when such an event should not produce a test failure. When an assumption failure occurs, the test is marked as filtered.

Create `IsSupportedTest` test case. Refer to the following `IsSupportedTest` test case in the subsequent steps in this example, which highlight specific functions in the file.

IsSupportedTest Class Definition File

```
classdef IsSupportedTest < matlab.unittest.TestCase
    methods(TestClassSetup)
        function TestPlatform(testcase)
            testcase.assumeFalse(ispc,...
                'Do not run any of these tests on Windows.')
        end
    end
    methods(Test)
        function test1(testcase)
            % write test code here
        end
```

```
    end
end
```

To execute the MATLAB commands in this example, add the `IsSupportedTest.m` file to a folder on your MATLAB path.

Write Test to Verify Platform. All tests in this test case must run on UNIX platforms only. The `TestPlatform` function uses the `assumeFalse` method to test if MATLAB is running on a Windows platform. If it is, the test fails.

```
function TestPlatform(testcase)
    testcase.assumeFalse(ispc,...
        'Do not run any of these tests on Windows.')
end
```

Make `TestPlatform` a `TestClassSetup` Test. To make the `TestPlatform` test a precondition, add it inside the `methods (TestClassSetup)` block.

Run the test case. Create a test case object and run the tests on a Windows platform.

```
tc = IsSupportedTest;
res = tc.run;
```

```
Running IsSupportedTest

================================================================================
All tests in IsSupportedTest were filtered.
    Test Diagnostic: Do not run any of these tests on Windows.
    Details
================================================================================

Done IsSupportedTest
__________

Failure Summary:

     Name                    Failed  Incomplete  Reason(s)
    ====================================================================
     IsSupportedTest/test1              X       Filtered by assumption.
```

The test(s) were filtered, and did not run (marked `Incomplete`).

For more information, click the Details link.

```
================================================================================
An assumption was not met while setting up or tearing down IsSupportedTest.
```

```
As a result, all IsSupportedTest tests were filtered.

    ----------------
    Test Diagnostic:
    ----------------
    Do not run any of these tests on Windows.

    ---------------------
    Framework Diagnostic:
    ---------------------
    assumeFalse failed.
    --> The value must evaluate to "false".

    Actual Value:
             1

    ------------------
    Stack Information:
    ------------------
    In C:\work\IsSupportedTest.m (IsSupportedTest.TestPlatform) at 4
================================================================================
```

The link to `IsSupportedTest.TestPlatform` under `Stack Information` takes you to the failed `assumeFalse` method.

More About

Exception Safe

Test content is exception safe when all fixture teardown is performed with `addTeardown` or through the appropriate object destructors when a failure occurs. This ensures that the failure does not affect later testing due to stale fixtures.

This code is not exception safe. After an assertion failure, the test framework does not close the figure.

```
% Not exception safe
f = figure;
testCase.assumeEqual(actual, expected)
close(f)
```

This code is exception safe because the test framework closes the figure in all cases.

```
% Exception safe
f = figure;
testCase.addTeardown(@close, f)
testCase.assumeEqual(actual, expected)
```

However, tearing down a fixture using `addTeardown` does not guarantee code is exception safe. This code shows a failure in `assumeEqual`.

```
% Not exception safe
f = figure;
testCase.assumeEqual(actual, expected)
testCase.addTeardown(@close, f)
```

See Also

`Assertable` | `FatalAssertable` | `QualificationEventData` | `TestCase` | `Verifiable` | `matlab.unittest.qualifications`

Topics

"Table of Verifications, Assertions, and Other Qualifications"
"Dynamically Filtered Tests"

Introduced in R2013a

assumeClass

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Assume exact class of specified value

Syntax

```
assumeClass(assumable,actual,className)
assumeClass(assumable,actual,metaClass)
assumeClass( ___ ,diagnostic)
```

Description

`assumeClass(assumable,actual,className)` assumes that `actual` is a MATLAB value whose class is the class specified by `className`.

`assumeClass(assumable,actual,metaClass)` assumes that `actual` is a MATLAB value whose class is the class specified by the `meta.class` instance `metaClass`. The instance must be an exact class match. See `assumeInstanceOf` to assume inclusion in a class hierarchy.

`assumeClass( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`className`

Name of class, specified as a character vector.

Default:

`metaClass`

An instance of `meta.class`.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyClass`, and replace calls to `verifyClass` with `assumeClass`.

Tips

- The method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsOfClass;
  assumable.assumeThat(actual, IsOfClass(className));
  assumable.assumeThat(actual, IsOfClass(metaClass));
  ```

  There exists more functionality when using the `IsOfClass` constraint directly via `assumeThat`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`assumeInstanceOf` | `assumeThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assumeEmpty

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Assume value is empty

Syntax

```
assumeEmpty(assumable,actual)
assumeEmpty(assumable,actual,diagnostic)
```

Description

`assumeEmpty(assumable,actual)` assumes that `actual` is an empty MATLAB value.

`assumeEmpty(assumable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyEmpty`, and replace calls to `verifyEmpty` with `assumeEmpty`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEmpty;
  assumable.assumeThat(actual, IsEmpty());
  ```

  There exists more functionality when using the `IsEmpty` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`assumeNotEmpty` | `assumeThat` | `isempty`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assumeEqual

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Assume value is equal to specified value

Syntax

```
assumeEqual(assumable,actual,expected)
assumeEqual( ___ ,Name,Value)
assumeEqual( ___ ,diagnostic)
```

Description

`assumeEqual(assumable,actual,expected)` assumes that `actual` is strictly equal to `expected`. If `expected` is not a MATLAB or Java object, `actual` and `expected` must have the same class, size, and value for the assumption to be met. `assumeEqual` compares `actual` and `expected` in the same way as the `IsEqualTo` constraint.

`assumeEqual( ___ ,Name,Value)` assumes equality with additional options specified by one or more `Name,Value` pair arguments.

`assumeEqual( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expected`

Expected value.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`AbsTol`

Absolute tolerance, specified as a numeric array. The tolerance is applied only to values of the same data type. The value can be a scalar or array the same size as the actual and expected values.

For an absolute tolerance to be satisfied, `abs(expected-actual) <= absTol` must be true.

Default:

`RelTol`

Relative tolerance, specified as a numeric array. The tolerance is applied only to values of the same data type. The value can be a scalar or array the same size as the actual and expected values.

For a relative tolerance to be satisfied, `abs(expected-actual) <= relTol.*abs(expected)` must be true.

Default:

Examples

Compare Numeric Values

Numeric values are equal if they are of the same class with equivalent size, complexity, and sparsity.

At the command prompt, create a test case for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Assume that a numeric value is equal to itself.

```
assumeEqual(testCase,5,5)
```

```
An assumption was met.
```

Assume if actual and expected values have equal sizes.

```
assumeEqual(testCase,[5 5],5)
```

```
An assumption was not met.
    ---------------------
    Framework Diagnostic:
    ---------------------
    assumeEqual failed.
    --> Sizes do not match.

        Actual size:
             1     2
        Expected size:
```

```
             1     1

    Actual Value:
         5     5
    Expected Value:
         5
Assumption failed.
```

Test Classes

Assume if an `int8` is equal to an `int16`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
assumeEqual(testCase,int8(5),int16(5))
```

```
An assumption was not met.
    ---------------------
    Framework Diagnostic:
    ---------------------
    assumeEqual failed.
    --> Classes do not match.

        Actual Class:
            int8
        Expected Class:
            int16

    Actual Value:
      int8

       5
    Expected Value:
      int16

       5
Assumption failed.
```

Test Cell Arrays

Elements of equal cell arrays must match in class, size, and value.

Assume that a cell array is equal to itself.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
assumeEqual(testCase,{'cell',struct,5},{'cell',struct,5})
```

```
An assumption was met.
```

Test Numeric Tolerances

Assume if `4.95` is equal to `5`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
assumeEqual(testCase,4.95,5)
```

```
An assumption was not met.
    ---------------------
    Framework Diagnostic:
    ---------------------
    assumeEqual failed.
    --> The numeric values are not equal using "isequaln".
    --> Failure table:
            Actual    Expected           Error              RelativeError
            ______    ________    ___________________    ___________________

             4.95        5        -0.0499999999999998    -0.00999999999999996

    Actual Value:
       4.950000000000000
    Expected Value:
         5
Assumption failed.
```

Assume that the difference between the actual value `1.5` and the expected value `2` is within `1`.

```
assumeEqual(testCase,1.5,2,'AbsTol',1)
```

```
An assumption was met.
```

Assume if the difference between the actual and expected values is less than 10%. Display the diagnostic information upon failure.

```
assumeEqual(testCase,1.5,2,'RelTol',0.1,...
    'Difference between actual and expected exceeds relative tolerance')
```

```
An assumption was not met.
    ----------------
    Test Diagnostic:
    ----------------
    Difference between actual and expected exceeds relative tolerance
    ---------------------
    Framework Diagnostic:
    ---------------------
```

```
    assumeEqual failed.
    --> The numeric values are not equal using "isequaln".
    --> The error was not within relative tolerance.
    --> Failure table:
            Actual    Expected    Error    RelativeError    RelativeTolerance
            ______    ________    _____    _____________    _________________

             1.5         2        -0.5         -0.25               0.1

    Actual Value:
       1.500000000000000
    Expected Value:
         2
Assumption failed.
```

Tips

- This method is functionally equivalent to any of the following:

  ```
  import matlab.unittest.constraints.IsEqualTo;
  assumable.assumeThat(actual, IsEqualTo(expected));
  ```

  ```
  import matlab.unittest.constraints.IsEqualTo;
  import matlab.unittest.constraints.AbsoluteTolerance;
  assumable.assumeThat(actual, IsEqualTo(expected, ...
      'Within', AbsoluteTolerance(abstol)));
  ```

  ```
  import matlab.unittest.constraints.IsEqualTo;
  import matlab.unittest.constraints.RelativeTolerance;
  assumable.assumeThat(actual, IsEqualTo(expected, ...
      'Within', RelativeTolerance(reltol)));
  ```

  ```
  import matlab.unittest.constraints.IsEqualTo;
  import matlab.unittest.constraints.AbsoluteTolerance;
  import matlab.unittest.constraints.RelativeTolerance;
  assumable.assumeThat(actual, IsEqualTo(expected, ...
      'Within', AbsoluteTolerance(abstol) | RelativeTolerance(reltol)));
  ```

  There exists more functionality when using the `IsEqualTo`, `AbsoluteTolerance`, and `RelativeTolerance` constraints directly via `assumeThat`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the

primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`assumeNotEqual` | `assumeThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assumeError

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Assume function throws specified exception

Syntax

```
assumeError(assumable,actual,identifier)
assumeError(assumable,actual,metaClass)
assumeError( ___ ,diagnostic)
[output1,...,outputN] = assumeError( ___ )
```

Description

`assumeError(assumable,actual,identifier)` assumes that `actual` is a function handle that throws an exception with an error identifier that is equal to `identifier`.

`assumeError(assumable,actual,metaClass)` assumes that `actual` is a function handle that throws an exception whose type is defined by the `meta.class` instance specified in `metaClass`. This method does not require the instance to be an exact class match, but rather it must be in the specified class hierarchy, and that hierarchy must include the `MException` class.

`assumeError( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

`[output1,...,outputN] = assumeError( ___ )` returns multiple output arguments from the invocation of the function handle `actual`. Use this syntax to control the number of output arguments requested from the function handle. If the function handle errors, all outputs are of type `missing`. Otherwise, `output1,...,outputN` are output values from `actual`. You can use this syntax with any of the input arguments of the previous syntaxes.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`identifier`

Error identifier, specified as a character vector.

Default:

`metaClass`

An instance of `meta.class`.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyError`, and replace calls to `verifyError` with `assumeError`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.Throws;
  assumable.assumeThat(actual, Throws(identifier));
  assumable.assumeThat(actual, Throws(metaClass));
  ```

  There exists more functionality when using the `Throws` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`MException` | `assumeThat` | `assumeWarning` | `error`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assumeFail

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Produce unconditional assumption failure

Syntax

```
assumeFail(assumable)
assumeFail(assumable,diagnostic)
```

Description

`assumeFail(assumable)` produces an unconditional assumption failure when encountered.

`assumeFail(assumable,diagnostic)`also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle

- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyFail`, and replace calls to `verifyFail` with `assumeFail`.

Tips

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assumeFalse

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Assume value is false

Syntax

```
assumeFalse(assumable,actual)
assumeFalse(assumable,actual,diagnostic)
```

Description

`assumeFalse(assumable,actual)` assumes that `actual` is a scalar logical with the value of false.

`assumeFalse(assumable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

For examples, see `verifyFalse`, and replace calls to `verifyFalse` with `assumeFalse`.

Tips

- This method passes if and only if the actual value is a scalar logical with a value of false. Therefore, entities such as empty arrays, false valued arrays, and zero doubles produce failures when used in this method, despite these entities exhibiting "false-like" behavior such as bypassing the execution of code inside of "if" statements.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsFalse;
  assumable.assumeThat(actual, IsFalse());
  ```

  There exists more functionality when using the `IsFalse` constraint directly via `assumeThat`.
- Unlike `assumeTrue`, this method may create a new constraint for each call. For performance critical uses, consider using `assumeTrue`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or

incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`assumeThat` | `assumeTrue`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assumeGreaterThan

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Assume value is greater than specified value

Syntax

```
assumeGreaterThan(assumable,actual,floor)
assumeGreaterThan(assumable,actual,floor,diagnostic)
```

Description

`assumeGreaterThan(assumable,actual,floor)` assumes that all elements of `actual` are greater than all the elements of `floor`.

`assumeGreaterThan(assumable,actual,floor,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test. `actual` must be the same size as `floor` unless either one is scalar, at which point scalar expansion occurs.

`floor`

Minimum value, exclusive.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyGreaterThan`, and replace calls to `verifyGreaterThan` with `assumeGreaterThan`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsGreaterThan;
  assumable.assumeThat(actual, IsGreaterThan(floor));
  ```

  There exists more functionality when using the `IsGreaterThan` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or

incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`assumeGreaterThanOrEqual` | `assumeLessThan` | `assumeLessThanOrEqual` | `assumeThat` | `gt` | `matlab.unittest.constraints.IsGreaterThan` | `matlab.unittest.diagnostics.Diagnostic`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assumeGreaterThanOrEqual

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Assume value is greater than or equal to specified value

Syntax

```
assumeGreaterThanOrEqual(assumable,actual,floor)
assumeGreaterThanOrEqual(assumable,actual,floor,diagnostic)
```

Description

`assumeGreaterThanOrEqual(assumable,actual,floor)` assumes that all elements of `actual` are greater than or equal to all the elements of `floor`.

`assumeGreaterThanOrEqual(assumable,actual,floor,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test. `actual` must be the same size as `floor` unless either one is scalar, at which point scalar expansion occurs.

`floor`

Minimum value.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyGreaterThanOrEqual`, and replace calls to `verifyGreaterThanOrEqual` with `assumeGreaterThanOrEqual`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsGreaterThanOrEqualTo;
  assumable.assumeThat(actual, IsGreaterThanOrEqualTo(floor));
  ```

  There exists more functionality when using the `IsGreaterThanOrEqualTo` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or

incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`assumeGreaterThan` | `assumeLessThan` | `assumeLessThanOrEqual` | `assumeThat` | `ge` | `matlab.unittest.constraints.IsGreaterThanOrEqualTo` | `matlab.unittest.diagnostics.Diagnostic`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assumeInstanceOf

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Assume value is object of specified type

Syntax

```
assumeInstanceOf(assumable,actual,className)
assumeInstanceOf(assumable,actual,metaClass)
assumeInstanceOf( ___ ,diagnostic)
```

Description

`assumeInstanceOf(assumable,actual,className)` assumes that `actual` is a MATLAB value whose class is the class specified by `className`.

`assumeInstanceOf(assumable,actual,metaClass)` assumes that `actual` is a MATLAB value whose class is the class specified by the `meta.class` instance `metaClass`.

`assumeInstanceOf( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`className`

Name of class, specified as a character vector.

Default:

`metaClass`

An instance of `meta.class`.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyInstanceOf`, and replace calls to `verifyInstanceOf` with `assumeInstanceOf`.

Tips

- This method is functionally equivalent to:

```
import matlab.unittest.constraints.IsInstanceOf;
assumable.assumeThat(actual, IsInstanceOf(className));
assumable.assumeThat(actual, IsInstanceOf(metaClass));
```

There exists more functionality when using the `IsInstanceOf` constraint directly via `assumeThat`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`assumeClass` | `assumeThat` | `isa`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assumeLength

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Assume value has specified length

Syntax

```
assumeLength(assumable,actual,expectedLength)
assumeLength(assumable,actual,expectedLength,diagnostic)
```

Description

`assumeLength(assumable,actual,expectedLength)` assumes that `actual` is a MATLAB array whose length is `expectedLength`.

`assumeLength(assumable,actual,expectedLength,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedLength`

The length of an array is defined as the largest dimension of that array.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyLength`, and replace calls to `verifyLength` with `assumeLength`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasLength;
  assumable.assumeThat(actual, HasLength(expectedLength));
  ```

  There exists more functionality when using the `HasLength` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or

incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`assumeNumElements` | `assumeSize` | `assumeThat` | `length`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assumeLessThan

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Assume value is less than specified value

Syntax

```
assumeLessThan(assumable,actual,ceiling)
assumeLessThan(assumable,actual,ceiling,diagnostic)
```

Description

`assumeLessThan(assumable,actual,ceiling)` assumes that all elements of `actual` are less than all the elements of `ceiling`.

`assumeLessThan(assumable,actual,ceiling,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test. `actual` must be the same size as `ceiling` unless either one is scalar, at which point scalar expansion occurs.

Default:

`ceiling`

Maximum value, exclusive.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyLessThan`, and replace calls to LessThan with `assumeLessThan`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsLessThan;
  assumable.assumeThat(actual, IsLessThan(ceiling));
  ```

  There exists more functionality when using the `IsLessThan` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or

incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`assumeGreaterThan` | `assumeGreaterThanOrEqual` | `assumeLessThanOrEqual` | `assumeThat` | `lt` | `matlab.unittest.constraints.IsLessThan` | `matlab.unittest.diagnostics.Diagnostic`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assumeLessThanOrEqual

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Assume value is less than or equal to specified value

Syntax

```
assumeLessThanOrEqual(assumable,actual,ceiling)
assumeLessThanOrEqual(assumable,actual,ceiling,diagnostic)
```

Description

`assumeLessThanOrEqual(assumable,actual,ceiling)` assumes that all elements of `actual` are less than or equal to all the elements of `ceiling`.

`assumeLessThanOrEqual(assumable,actual,ceiling,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test. `actual` must be the same size as `ceiling` unless either one is scalar, at which point scalar expansion occurs.

Default:

`ceiling`

Maximum value.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyLessThanOrEqual`, and replace calls to `verifyLessThanOrEqual` with `assumeLessThanOrEqual`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsLessThanOrEqualTo;
  assumable.assumeThat(actual, IsLessThanOrEqualTo(ceiling));
  ```

  There exists more functionality when using the `IsLessThanOrEqualTo` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or

incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`assumeGreaterThan` | `assumeGreaterThanOrEqual` | `assumeLessThan` | `assumeThat` | `le` | `matlab.unittest.constraints.IsLessThanOrEqualTo` | `matlab.unittest.diagnostics.Diagnostic`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assumeMatches

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Assume string matches specified regular expression

Syntax

```
assumeMatches(assumable,actual,expression)
assumeMatches(assumable,actual,expression,diagnostic)
```

Description

`assumeMatches(assumable,actual,expression)` assumes that `actual` is a string scalar or character vector that matches the regular expression defined by `expression`.

`assumeMatches(assumable,actual,expression,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The string to test.

Default:

`expression`

The value to match, specified as a regular expression.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyMatches`, and replace calls to `verifyMatches` with `assumeMatches`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.Matches;
  assumable.assumeThat(actual, Matches(expression));
  ```

  There exists more functionality when using the `Matches` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or

incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`assumeSubstring` | `assumeThat` | `regexp`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assumeNotEmpty

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Assume value is not empty

Syntax

```
assumeNotEmpty(assumable,actual)
assumeNotEmpty(assumable,actual,diagnostic)
```

Description

`assumeNotEmpty(assumable,actual)` assumes that `actual` is a non-empty MATLAB value.

`assumeNotEmpty(assumable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyNotEmpty`, and replace calls to `verifyNotEmpty` with `assumeNotEmpty`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEmpty;
  assumable.assumeThat(actual, ~IsEmpty());
  ```

  There exists more functionality when using the `IsEmpty` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`assumeEmpty` | `assumeThat` | `isempty`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assumeNotEqual

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Assume value is not equal to specified value

Syntax

```
assumeNotEqual(assumable,actual,notExpected)
assumeNotEqual(assumable,actual,notExpected,diagnostic)
```

Description

`assumeNotEqual(assumable,actual,notExpected)` assumes that `actual` is not equal to `notExpected`.

`assumeNotEqual(assumable,actual,notExpected,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`notExpected`

Value to compare.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyNotEqual`, and replace calls to `verifyNotEqual` with `assumeNotEqual`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEqualTo;
  assumable.assumeThat(actual, ~IsEqualTo(notExpected));
  ```

  There exists more functionality when using the `IsEqualTo` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or

incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`assumeEqual` | `assumeThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assumeNotSameHandle

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Assume value is not handle to specified instance

Syntax

```
assumeNotSameHandle(assumable,actual,notExpectedHandle)
assumeNotSameHandle(assumable,actual,notExpectedHandle,diagnostic)
```

Description

`assumeNotSameHandle(assumable,actual,notExpectedHandle)` assumes that `actual` is a different size and/or does not contain the same instances as the `notExpectedHandle` handle array.

`assumeNotSameHandle(assumable,actual,notExpectedHandle,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`notExpectedHandle`

The handle array to compare.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyNotSameHandle`, and replace calls to `verifyNotSameHandle` with `assumeNotSameHandle`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsSameHandleAs;
  assumable.assumeThat(actual, ~IsSameHandleAs(notExpectedHandle));
  ```

  There exists more functionality when using the `IsSameHandleAs` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or

incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`assumeSameHandle` | `assumeThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assumeNumElements

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Assume value has specified element count

Syntax

```
assumeNumElements(assumable,actual,expectedElementCount)
assumeNumElements(assumable,actual,expectedElementCount,diagnostic)
```

Description

`assumeNumElements(assumable,actual,expectedElementCount)` assumes that `actual` is a MATLAB array with `expectedElementCount` number of elements.

`assumeNumElements(assumable,actual,expectedElementCount,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedElementCount`

The expected number of elements in the array.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyNumElements`, and replace calls to `verifyNumElements` with `assumeNumElements`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasElementCount;
  assumable.assumeThat(actual, HasElementCount(expectedElementCount));
  ```

  There exists more functionality when using the `HasElementCount` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or

incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`assumeLength` | `assumeSize` | `assumeThat` | `numel`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assumeReturnsTrue

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Assume function returns true when evaluated

Syntax

```
assumeReturnsTrue(assumable,actual)
assumeReturnsTrue(assumable,actual,diagnostic)
```

Description

`assumeReturnsTrue(assumable,actual)` assumes that `actual` is a function handle that returns a scalar logical whose value is true.

`assumeReturnsTrue(assumable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The function handle to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyReturnsTrue`, and replace calls to `verifyReturnsTrue` with `assumeReturnsTrue`.

Tips

- It is a shortcut for quick custom comparison functionality that can be defined quickly, and possibly inline. It can be preferable over simply evaluating the function directly and using `assumeTrue` because the function handle will be shown in the diagnostics, thus providing more insight into the failure condition which is lost when using `assumeTrue`.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.ReturnsTrue;
  assumable.assumeThat(actual, ReturnsTrue());
  ```

  There exists more functionality when using the `ReturnsTrue` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`assumeThat` | `assumeTrue`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assumeSameHandle

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Assume two values are handles to same instance

Syntax

```
assumeSameHandle(assumable,actual,expectedHandle)
assumeSameHandle(assumable,actual,expectedHandle,diagnostic)
```

Description

`assumeSameHandle(assumable,actual,expectedHandle)` assumes that `actual` is the same size and contains the same instances as the `expectedHandle` handle array.

`assumeSameHandle(assumable,actual,expectedHandle,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedHandle`

The expected handle array.

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifySameHandle`, and replace calls to `verifySameHandle` with `assumeSameHandle`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsSameHandleAs;
  assumable.assumeThat(actual, IsSameHandleAs(expectedHandle));
  ```

  There exists more functionality when using the `IsSameHandleAs` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`assumeNotSameHandle` | `assumeThat` | `handle`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assumeSize

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Assume value has specified size

Syntax

```
assumeSize(assumable,actual,expectedSize)
assumeSize(assumable,actual,expectedSize,diagnostic)
```

Description

`assumeSize(assumable,actual,expectedSize)` assumes that `actual` is a MATLAB array whose size is `expectedSize`.

`assumeSize(assumable,actual,expectedSize,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedSize`

The expected sizes of each dimension the array.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifySize`, and replace calls to `verifySize` with `assumeSize`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasSize;
  assumable.assumeThat(actual, HasSize(expectedSize));
  ```

  There exists more functionality when using the `HasSize` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`assumeLength` | `assumeNumElements` | `assumeThat` | `size`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assumeSubstring

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Assume string contains specified string

Syntax

```
assumeSubstring(assumable,actual,substring)
assumeSubstring(assumable,actual,substring,diagnostic)
```

Description

`assumeSubstring(assumable,actual,substring)` assumes that `actual` is a string scalar or character vector that contains `substring`.

`assumeSubstring(assumable,actual,substring,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`substring`

The value to match, specified as a string scalar or character vector.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifySubstring`, and replace calls to `verifySubstring` with `assumeSubstring`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.ContainsSubstring;
  assumable.assumeThat(actual, ContainsSubstring(substring));
  ```

  There exists more functionality when using the `ContainsSubstring` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or

incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`assumeMatches` | `assumeThat` | `strfind`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assumeThat

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Assume value meets specified constraint

Syntax

```
assumeThat(assumable,actual,constraint)
assumeThat(assumable,actual,constraint,diagnostic)
```

Description

`assumeThat(assumable,actual,constraint)` assumes that `actual` is a value that satisfies the `constraint` provided.

If the constraint is not satisfied, an assumption failure is produced utilizing only the framework diagnostic generated by the `constraint`.

`assumeThat(assumable,actual,constraint,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

When using this signature, both the diagnostic information contained within `diagnostic` is used in addition to the diagnostic information provided by the `constraint`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`constraint`

Constraint that the actual value must satisfy to pass the verification, specified as a `matlab.unittest.constraints` instance.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyThat`, and replace calls to `verifyThat` with `assumeThat`.

Tips

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assumeTrue

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Assume value is true

Syntax

```
assumeTrue(assumable,actual)
assumeTrue(assumable,actual,diagnostic)
```

Description

`assumeTrue(assumable,actual)` assumes that `actual` is a scalar logical with the value of true.

`assumeTrue(assumable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyTrue`, and replace calls to `verifyTrue` with `assumeTrue`.

Tips

- This method passes if and only if the actual value is a scalar logical with a value of true. Therefore, entities such as true valued arrays and nonzero doubles produce qualification failures when used in this method, despite these entities exhibiting "true-like" behavior such as triggering the execution of code inside of "if" statements.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsTrue;
  assumable.assumeThat(actual, IsTrue());
  ```

  There exists more functionality when using the `IsTrue` constraint directly via `assumeThat`.

  Use of this method for performance benefits can come at the expense of less diagnostic information, and may not provide the same level of strictness adhered to by other constraints such as `IsEqualTo`. A similar approach that is generally less performant but can provide slightly better diagnostic information is the use of `assumeReturnsTrue`, which at least shows the display of the function evaluated to generate the failing result.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to

completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`assumeFalse` | `assumeReturnsTrue` | `assumeThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assumeWarning

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Assume function issues specified warnings

Syntax

```
assumeWarning(assumable,actual,warningIDs)
assumeWarning(assumable,actual,warningIDs,diagnostic)
[output1,...,outputN] = assumeWarning( ___ )
```

Description

`assumeWarning(assumable,actual,warningIDs)` assumes that `actual` issues warnings with the identifiers `warningIDs`.

`assumeWarning(assumable,actual,warningIDs,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

`[output1,...,outputN] = assumeWarning( ___ )` also return the output arguments `output1,...,outputN` that are produced when invoking `actual`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The function handle to test.

Default:

`warningIDs`

Warning identifiers, specified as a character vector, cell array of character vectors, or string array.

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Output Arguments

`output1,...,outputN`

Output arguments, 1 through n (if any), from `actual`, returned as any type. The argument type is specified by the `actual` argument list.

Examples

See examples for `verifyWarning`, and replace calls to `verifyWarning` with `assumeWarning`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IssuesWarnings;
  assumable.assumeThat(actual,IssuesWarnings(cellstr(warningIDs)));
  ```

  There exists more functionality when using the `IssuesWarnings` constraint directly via `assumeThat`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`assumeError` | `assumeThat` | `assumeWarningFree` | `warning`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

assumeWarningFree

Class: `matlab.unittest.qualifications.Assumable`
Package: `matlab.unittest.qualifications`

Assume function issues no warnings

Syntax

```
assumeWarningFree(assumable,actual)
assumeWarningFree(assumable,actual,diagnostic)
[output1,...,outputN] = assumeWarningFree( ___ )
```

Description

`assumeWarningFree(assumable,actual)` assumes that `actual` is a function handle that issues no warnings.

`assumeWarningFree(assumable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

`[output1,...,outputN] = assumeWarningFree( ___ )` also returns the output arguments `output1,...,outputN` that are produced when invoking `actual`.

Input Arguments

`assumable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the assumption in conjunction with the test running framework.

`actual`

The function handle to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Output Arguments

`output1,...,outputN`

Output arguments, 1 through n (if any), from `actual`, returned as any type. The argument type is specified by the `actual` argument list.

Examples

See examples for `verifyWarningFree`, and replace calls to `verifyWarningFree` with `assumeWarningFree`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IssuesNoWarnings;
  assumable.assumeThat(actual, IssuesNoWarnings());
  ```

  There exists more functionality when using the `IssuesNoWarnings` constraint directly via `assumeThat`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. Alternatively,

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`assumeThat` | `assumeWarning` | `warning`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

matlab.unittest.qualifications.AssumptionFailedException class

Package: `matlab.unittest.qualifications`

Exception used for assumption failures

Description

The `AssumptionFailedException` class provides an exception used for assumption failures. This class is used exclusively by the `Assumable` qualification type.

See Also

`Assumable` | `MException`

matlab.unittest.qualifications.ExceptionEventData class

Package: `matlab.unittest.qualifications`

Event data for ExceptionThrown event listeners

Description

The `ExceptionEventData` class holds event data for `ExceptionThrown` event listeners. `ExceptionThrown` event listeners are callback functions that you register with the testing framework to listen for the `TestRunner` to encounter an error during execution of test content. Typically, authors of custom plugins use this class. Only the test framework constructs this class directly.

Properties

`Exception`

Unexpected exception caught by `TestRunner` during its execution of test content

See Also

`MException` | `matlab.unittest.TestRunner` | `matlab.unittest.plugins.TestRunnerPlugin`

Introduced in R2014a

matlab.unittest.qualifications.FatalAssertable class

Package: `matlab.unittest.qualifications`

Qualification to abort test execution

Description

The `FatalAssertable` class provides a qualification to abort test execution. Apart from actions performed for failures, the `FatalAssertable` class works the same as `matlab.unittest` qualifications.

Upon a fatal assertion failure, the `FatalAssertable` class informs the testing framework of the failure by throwing a `FatalAssertionFailedException`. The test running framework then displays diagnostic information for the failure and aborts the entire test session. This is useful when the software under test contains so many errors that it does not make sense to continue the test session. Also, you can use fatal assertions in fixture teardown to guarantee the fixture state is restored correctly. If it is not restored, the full testing session will abort and indicate to restart MATLAB before you resume testing. This allows later tests to run in a consistent MATLAB state. If you can recover the fixture teardown and make it “Exception Safe” on page 1-12424 for failures, use assertions instead.

Fatal assertions prevent false test failures due to the failure of a fundamental test. They also prevent false test failures when a prior test failed to restore test fixtures. If the test framework cannot properly tear down fixtures, restart MATLAB to ensure testing can resume in a clean state.

Methods

fatalAssertClass	Fatally assert exact class of specified value
fatalAssertEmpty	Fatally assert value is empty
fatalAssertEqual	Fatally assert value is equal to specified value
fatalAssertError	Fatally assert function throws specified exception
fatalAssertFail	Produce unconditional fatal assertion failure
fatalAssertFalse	Fatally assert value is false
fatalAssertGreaterThanOrEqual	Fatally assert value is greater than or equal to specified value
fatalAssertInstanceOf	Fatally assert value is object of specified type
fatalAssertLength	Fatally assert value has specified length
fatalAssertLessThan	Fatally assert value is less than specified value
fatalAssertLessThanOrEqual	Fatally assert value is less than or equal to specified value
fatalAssertMatches	Fatally assert string matches specified regular expression
fatalAssertNotEmpty	Fatally assert value is not empty
fatalAssertNotEqual	Fatally assert value is not equal to specified value
fatalAssertNotSameHandle	Fatally assert value is not handle to specified instance
fatalAssertNumElements	Fatally assert value has specified element count
fatalAssertReturnsTrue	Fatally assert function returns true when evaluated
fatalAssertSameHandle	Fatally assert two values are handles to same instance
fatalAssertSize	Fatally assert value has specified size
fatalAssertSubstring	Fatally assert string contains specified string
fatalAssertThat	Fatally assert value meets specified constraint
fatalAssertTrue	Fatally assert value is true
fatalAssertWarning	Fatally assert function issues specified warnings
fatalAssertWarningFree	Fatally assert function issues no warnings

Events

`FatalAssertionFailed`	Triggered upon failing fatal assertion. A `QualificationEventData` object is passed to listener callback functions.
`FatalAssertionPassed`	Triggered upon passing fatal assertion. A `QualificationEventData` object is passed to listener callback functions.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Write Helper Function Using Fatal Assertions

A fatal assertion renders the remainder of the current test method invalid because the state is unrecoverable. A helper function is a function in the `TestCase` class but not located within any of the `methods` block statement. Execution of these functions is not controlled by the `matlab.unittest` framework.

Add the `DocPolynomSaveLoadTest.m` file to a folder on your MATLAB path. Refer to the helper function, `cleanUpTemporaryFolder`, in the `DocPolynomSaveLoadTest` test case.

DocPolynomSaveLoadTest Class Definition File

```
classdef DocPolynomSaveLoadTest < matlab.unittest.TestCase

    methods (TestClassSetup)
        function addDocPolynomClassToPath(testCase)
            origPath = path;
            testCase.addTeardown(@path, origPath);
            addpath(fullfile(matlabroot, ...
                'help', 'techdoc', 'matlab_oop', 'examples'));
        end
```

```
end

methods (Test)
    function testSaveLoad(testCase)

        import matlab.unittest.diagnostics.Diagnostic;

        %% Phase 1: Setup
        % Create a temporary working folder
        tempFolder = tempname;
        [success, message] = mkdir(tempFolder);
        testCase.assertTrue(success, ...
            Diagnostic.join('Could not create temporary folder.',...
            message));
        testCase.addTeardown(@() testCase.cleanUpTemporaryFolder(...
            tempFolder));

        % Change to the temporay folder and register the
        % teardown, which restores the original folder
        origFolder = pwd;
        testCase.addTeardown(@cd, origFolder);
        cd(tempFolder);

        %% Phase 2: Exercise
        % Save the instance to a mat file.
        p = DocPolynom([1, 0, 1]);
        save('DocPolynomFile', 'p');

        % Validate Precondition. Save resulted in valid .mat file
        testCase.assertEqual(exist('DocPolynomFile.mat','file'),...
            2, Diagnostic.join(...
            'mat file was not saved correctly.',@() dir(pwd)));

        loaded = load('DocPolynomFile');


        %% Phase 3: Verify
        testCase.verifyEqual(loaded.p, p,...
            'Loaded polynom did not equal original polynom.');


        %% Phase 4: Teardown
        % Done inline via calls to addTeardown at the points
        % at which the state was changed.
```

```
        end
    end

    methods(Access=private)
        function cleanUpTemporaryFolder(testCase,tempFolder)
            % Clean up the temporary folder and fatally assert
            % that it was correctly cleaned up.

            import matlab.unittest.diagnostics.Diagnostic;

            [success, message] = rmdir(tempFolder, 's');
            testCase.fatalAssertTrue(success, ...
                Diagnostic.join('Could not remove temporary folder.',...
                message));
        end
    end

end
```

Make the `cleanUpTemporaryFolder` function a helper function by placing it inside a separate `methods` block.

```
methods(Access=private)
    function cleanUpTemporaryFolder(testCase, tempFolder)
        % code
    end
end
```

Use the `fatalAssertTrue` method to test the `rmdir` `success` argument for errors. If a fatal assertion occurs, the test run is aborted.

```
function cleanUpTemporaryFolder(testCase, tempFolder)

    import matlab.unittest.diagnostics.Diagnostic

    [success, message] = rmdir(tempFolder, 's');
    testCase.fatalAssertTrue(success, ...
        Diagnostic.join('Could not remove the temporary folder.',...
        message))
end
```

If the `rmdir` function fails, then this test has failed to restore the state of MATLAB and the machine at initial startup. Aborting prevents subsequent tests to fail because MATLAB is left in an unexpected state by this test.

More About

Exception Safe

Test content is exception safe when all fixture teardown is performed with `addTeardown` or through the appropriate object destructors when a failure occurs. This ensures that the failure does not affect later testing due to stale fixtures.

This code is not exception safe. After an assertion failure, the test framework does not close the figure.

```
% Not exception safe
f = figure;
testCase.fatalAssertEqual(actual, expected)
close(f)
```

This code is exception safe because the test framework closes the figure in all cases.

```
% Exception safe
f = figure;
testCase.addTeardown(@close, f)
testCase.fatalAssertEqual(actual, expected)
```

However, tearing down a fixture using `addTeardown` does not guarantee code is exception safe. This code shows a failure in `fatalAssertEqual`.

```
% Not exception safe
f = figure;
testCase.fatalAssertEqual(actual, expected);
testCase.addTeardown(@close, f)
```

See Also

`Assertable` | `Assumable` | `QualificationEventData` | `TestCase` | `Verifiable` | `matlab.unittest.qualifications`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertClass

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Fatally assert exact class of specified value

Syntax

```
fatalAssertClass(fatalAssertable,actual,className)
fatalAssertClass(fatalAssertable,actual,metaClass)
fatalAssertClass( ___ ,diagnostic)
```

Description

`fatalAssertClass(fatalAssertable,actual,className)` fatally asserts that `actual` is a MATLAB value whose class is the class specified by `className`.

`fatalAssertClass(fatalAssertable,actual,metaClass)` fatally asserts that `actual` is a MATLAB value whose class is the class specified by the `meta.class` instance `metaClass`.

`fatalAssertClass( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`className`

Name of class, specified as a character vector.

Default:

`metaClass`

An instance of `meta.class`.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyClass`, and replace calls to `verifyClass` with `fatalAssertClass`.

Tips

- The method is functionally equivalent to:

```
import matlab.unittest.constraints.IsOfClass;
fatalAssertable.fatalAssertThat(actual, IsOfClass(className));
fatalAssertable.fatalAssertThat(actual, IsOfClass(metaClass));
```

There exists more functionality when using the `IsOfClass` constraint directly via `fatalAssertThat`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`fatalAssertInstanceOf` | `fatalAssertThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertEmpty

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Fatally assert value is empty

Syntax

```
fatalAssertEmpty(fatalAssertable,actual)
fatalAssertEmpty(fatalAssertable,actual,diagnostic)
```

Description

`fatalAssertEmpty(fatalAssertable,actual)` fatally asserts that `actual` is an empty MATLAB value.

`fatalAssertEmpty(fatalAssertable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyEmpty`, and replace calls to `verifyEmpty` with `fatalAssertEmpty`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEmpty;
  fatalAssertable.fatalAssertThat(actual, IsEmpty());
  ```

  There exists more functionality when using the `IsEmpty` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit

from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`fatalAssertNotEmpty` | `fatalAssertThat` | `isempty`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertEqual

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Fatally assert value is equal to specified value

Syntax

```
fatalAssertEqual(fatalAssertable,actual,expected)
fatalAssertEqual( ___ ,Name,Value)
fatalAssertEqual( ___ ,diagnostic)
```

Description

`fatalAssertEqual(fatalAssertable,actual,expected)` fatally asserts that `actual` is strictly equal to `expected`. If `expected` is not a MATLAB or Java object, `actual` and `expected` must have the same class, size, and value for the fatal assertion to pass. `fatalAssertEqual` compares `actual` and `expected` in the same way as the `IsEqualTo` constraint.

`fatalAssertEqual( ___ ,Name,Value)` fatally asserts equality with additional options specified by one or more `Name,Value` pair arguments.

`fatalAssertEqual( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expected`

Expected value.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`AbsTol`

Absolute tolerance, specified as a numeric array. The tolerance is applied only to values of the same data type. The value can be a scalar or array the same size as the actual and expected values.

For an absolute tolerance to be satisfied, `abs(expected-actual) <= absTol` must be true.

Default:

`RelTol`

Relative tolerance, specified as a numeric array. The tolerance is applied only to values of the same data type. The value can be a scalar or array the same size as the actual and expected values.

For a relative tolerance to be satisfied, `abs(expected-actual) <= relTol.*abs(expected)` must be true.

Default:

Examples

Compare Numeric Values

Numeric values are equal if they are of the same class with equivalent size, complexity, and sparsity.

At the command prompt, create a test case for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Fatally assert that a numeric value is equal to itself.

```
fatalAssertEqual(testCase,5,5)
```

```
Fatal assertion passed.
```

Fatally assert if actual and expected values have equal sizes.

```
fatalAssertEqual(testCase,[5 5],5)
```

```
Fatal assertion failed.
    ---------------------
    Framework Diagnostic:
    ---------------------
    fatalAssertEqual failed.
    --> Sizes do not match.

        Actual size:
             1     2
        Expected size:
```

```
             1     1

    Actual Value:
         5     5
    Expected Value:
         5
Fatal assertion failed.
```

Test Classes

Fatally assert if an `int8` is equal to an `int16`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
fatalAssertEqual(testCase,int8(5),int16(5))
```

```
Fatal assertion failed.
    ---------------------
    Framework Diagnostic:
    ---------------------
    fatalAssertEqual failed.
    --> Classes do not match.

        Actual Class:
            int8
        Expected Class:
            int16

    Actual Value:
      int8

       5
    Expected Value:
      int16

       5
Fatal assertion failed.
```

Test Cell Arrays

Elements of equal cell arrays must match in class, size, and value.

Fatally assert that a cell array is equal to itself.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
fatalAssertEqual(testCase,{'cell',struct,5},{'cell',struct,5})
```

```
Fatal assertion passed.
```

Test Numeric Tolerances

Fatally assert if `4.95` is equal to `5`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
fatalAssertEqual(testCase,4.95,5)
```

```
Fatal assertion failed.
    --------------------
    Framework Diagnostic:
    --------------------
    fatalAssertEqual failed.
    --> The numeric values are not equal using "isequaln".
    --> Failure table:
            Actual    Expected           Error              RelativeError
            ______    ________    __________________    ___________________

             4.95        5        -0.0499999999999998    -0.00999999999999996

    Actual Value:
       4.950000000000000
    Expected Value:
         5
Fatal assertion failed.
```

Fatally assert that the difference between the actual value `1.5` and the expected value `2` is within `1`.

```
fatalAssertEqual(testCase,1.5,2,'AbsTol',1)
```

```
Fatal assertion passed.
```

Fatally assert if the difference between the actual and expected values is less than 10%. Display the diagnostic information upon failure.

```
fatalAssertEqual(testCase,1.5,2,'RelTol',0.1,...
    'Difference between actual and expected exceeds relative tolerance')
```

```
Fatal assertion failed.
    ----------------
    Test Diagnostic:
    ----------------
    Difference between actual and expected exceeds relative tolerance
    --------------------
    Framework Diagnostic:
    --------------------
```

```
    fatalAssertEqual failed.
    --> The numeric values are not equal using "isequaln".
    --> The error was not within relative tolerance.
    --> Failure table:
            Actual    Expected    Error    RelativeError    RelativeTolerance
            ______    ________    _____    _____________    _________________

             1.5         2        -0.5         -0.25               0.1

    Actual Value:
       1.500000000000000
    Expected Value:
        2
Fatal assertion failed.
```

Tips

- This method is functionally equivalent to any of the following:

  ```
  import matlab.unittest.constraints.IsEqualTo;
  fatalAssertable.fatalAssertThat(actual, IsEqualTo(expected));

  import matlab.unittest.constraints.IsEqualTo;
  import matlab.unittest.constraints.AbsoluteTolerance;
  fatalAssertable.fatalAssertThat(actual, IsEqualTo(expected, ...
      'Within', AbsoluteTolerance(abstol)));

  import matlab.unittest.constraints.IsEqualTo;
  import matlab.unittest.constraints.RelativeTolerance;
  fatalAassertable.fatalAssertThat(actual, IsEqualTo(expected, ...
      'Within', RelativeTolerance(reltol)));

  import matlab.unittest.constraints.IsEqualTo;
  import matlab.unittest.constraints.AbsoluteTolerance;
  import matlab.unittest.constraints.RelativeTolerance;
  fatalAssertable.fatalAssertThat(actual, IsEqualTo(expected, ...
      'Within', AbsoluteTolerance(abstol) | RelativeTolerance(reltol)));
  ```

  There exists more functionality when using the `IsEqualTo`, `AbsoluteTolerance`, and `RelativeTolerance` constraints directly via `fatalAssertThat`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`fatalAssertNotEqual` | `fatalAssertThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertError

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Fatally assert function throws specified exception

Syntax

```
fatalAssertError(fatalAssertable,actual,identifier)
fatalAssertError(fatalAssertable,actual,metaClass)
fatalAssertError( ___ ,diagnostic)
[output1,...,outputN] = fatalAssertError( ___ )
```

Description

`fatalAssertError(fatalAssertable,actual,identifier)` fatally asserts that `actual` is a function handle that throws an exception with an error identifier that is equal to `identifier`.

`fatalAssertError(fatalAssertable,actual,metaClass)` fatally asserts that `actual` is a function handle that throws an exception whose type is defined by the `meta.class` instance specified in `metaClass`. This method does not require the instance to be an exact class match, but rather it must be in the specified class hierarchy, and that hierarchy must include the `MException` class..

`fatalAssertError( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

`[output1,...,outputN] = fatalAssertError( ___ )` returns multiple output arguments from the invocation of the function handle `actual`. Use this syntax to control the number of output arguments requested from the function handle. If the function handle errors, all outputs are of type `missing`. Otherwise, `output1,...,outputN` are output values from `actual`. You can use this syntax with any of the input arguments of the previous syntaxes.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`identifier`

Error identifier, specified as a character vector.

Default:

`metaClass`

An instance of `meta.class`.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyError`, and replace calls to `verifyError` with `fatalAssertError`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.Throws;
  fatalAssertable.fatalAssertThat(actual, Throws(identifier));
  fatalAssertable.fatalAssertThat(actual, Throws(metaClass));
  ```

  There exists more functionality when using the `Throws` constraint directly via `fatalAssertThat`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`MException` | `error` | `fatalAssertThat` | `fatalAssertWarning`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertFail

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Produce unconditional fatal assertion failure

Syntax

```
fatalAssertFail(fatalAssertable)
fatalAssertFail(fatalAssertable,diagnostic)
```

Description

`fatalAssertFail(fatalAssertable)` produces an unconditional fatal assertion failure when encountered.

`fatalAssertFail(fatalAssertable,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyFail`, and replace calls to `verifyFail` with `fatalAssertFail`.

Tips

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertFalse

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Fatally assert value is false

Syntax

```
fatalAssertFalse(fatalAssertable,actual)
fatalAssertFalse(fatalAssertable,actual,diagnostic)
```

Description

`fatalAssertFalse(fatalAssertable,actual)` fatally asserts that `actual` is a scalar logical with the value of false.

`fatalAssertFalse(fatalAssertable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyFalse`, and replace calls to `verifyFalse` with `fatalAssertFalse`.

Tips

- This method passes if and only if the actual value is a scalar logical with a value of false. Therefore, entities such as empty arrays, false valued arrays, and zero doubles produce failures when used in this method, despite these entities exhibiting "false-like" behavior such as bypassing the execution of code inside of "if" statements.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsFalse;
  fatalAssertable.fatalAssertThat(actual, IsFalse());
  ```

  There exists more functionality when using the `IsFalse` constraint directly via `fatalAssertThat`.
- Unlike `fatalAssertTrue`, this method may create a new constraint for each call. For performance critical uses, consider using `fatalAssertTrue`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed

and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`fatalAssertThat` | `fatalAssertTrue`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertGreaterThan

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Fatally assert value is greater than specified value

Syntax

```
fatalAssertGreaterThan(fatalAssertable,actual,floor)
fatalAssertGreaterThan(fatalAssertable,actual,floor,diagnostic)
```

Description

`fatalAssertGreaterThan(fatalAssertable,actual,floor)` fatally asserts that all elements of `actual` are greater than all the elements of `floor`.

`fatalAssertGreaterThan(fatalAssertable,actual,floor,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test. `actual` must be the same size as `floor` unless either one is scalar, at which point scalar expansion occurs.

`floor`

Minimum value, exclusive.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyGreaterThan`, and replace calls to `verifyGreaterThan` with `fatalAssertGreaterThan`.

Tips

- This method is functionally equivalent to:

  ```
  matlab.unittest.constraints.IsGreaterThan;
  fatalAssertable.fatalAssertThat(actual, IsGreaterThan(floor));
  ```

  There exists more functionality when using the `IsGreaterThan` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed

and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`fatalAssertGreaterThanOrEqual` | `fatalAssertLessThan` | `fatalAssertLessThanOrEqual` | `fatalAssertThat` | `gt` | `matlab.unittest.constraints.IsGreaterThan` | `matlab.unittest.diagnostics.Diagnostic`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertGreaterThanOrEqual

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Fatally assert value is greater than or equal to specified value

Syntax

```
fatalAssertGreaterThanOrEqual(fatalAssertable,actual,floor)
fatalAssertGreaterThanOrEqual(fatalAssertable,actual,floor,
diagnostic)
```

Description

`fatalAssertGreaterThanOrEqual(fatalAssertable,actual,floor)` fatally asserts that all elements of `actual` are greater than or equal to all the elements of `floor`.

`fatalAssertGreaterThanOrEqual(fatalAssertable,actual,floor, diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test. `actual` must be the same size as `floor` unless either one is scalar, at which point scalar expansion occurs.

`floor`

Minimum value.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyGreaterThanOrEqual`, and replace calls to `verifyGreaterThanOrEqual` with `fatalAssertGreaterThanOrEqual`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsGreaterThanOrEqualTo;
  fatalAssertable.fatalAssertThat(actual, IsGreaterThanOrEqualTo(floor));
  ```

  There exists more functionality when using the `IsGreaterThanOrEqualTo` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`fatalAssertGreaterThan` | `fatalAssertLessThan` | `fatalAssertLessThanOrEqual` | `fatalAssertThat` | `ge` | `matlab.unittest.constraints.IsGreaterThanOrEqualTo` | `matlab.unittest.diagnostics.Diagnostic`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertInstanceOf

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Fatally assert value is object of specified type

Syntax

```
fatalAssertInstanceOf(fatalAssertable,actual,className)
fatalAssertInstanceOf(fatalAssertable,actual,metaClass)
fatalAssertInstanceOf( ___ ,diagnostic)
```

Description

`fatalAssertInstanceOf(fatalAssertable,actual,className)` fatally asserts that `actual` is a MATLAB value whose class is the class specified by `className`.

`fatalAssertInstanceOf(fatalAssertable,actual,metaClass)` fatally asserts that `actual` is a MATLAB value whose class is the class specified by the `meta.class` instance `metaClass`.

`fatalAssertInstanceOf( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`className`

Name of class, specified as a character vector.

Default:

`metaClass`

An instance of `meta.class`.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyInstanceOf`, and replace calls to `verifyInstanceOf` with `fatalAssertInstanceOf`.

Tips

- This method is functionally equivalent to:

```
import matlab.unittest.constraints.IsInstanceOf;
fatalAssertable.fatalAssertThat(actual, IsInstanceOf(className));
fatalAssertable.fatalAssertThat(actual, IsInstanceOf(metaClass));
```

There exists more functionality when using the `IsInstanceOf` constraint directly via `fatalAssertThat`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`fatalAssertClass` | `fatalAssertThat` | `isa`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertLength

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Fatally assert value has specified length

Syntax

```
fatalAssertLength(fatalAssertable,actual,expectedLength)
fatalAssertLength(fatalAssertable,actual,expectedLength,diagnostic)
```

Description

`fatalAssertLength(fatalAssertable,actual,expectedLength)` fatally asserts that `actual` is a MATLAB array whose length is `expectedLength`.

`fatalAssertLength(fatalAssertable,actual,expectedLength,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedLength`

The length of an array is defined as the largest dimension of that array.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyLength`, and replace calls to `verifyLength` with `fatalAssertLength`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasLength;
  fatalAssertable.fatalAssertThat(actual, HasLength(expectedLength));
  ```

  There exists more functionality when using the `HasLength` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed

and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`fatalAssertNumElements` | `fatalAssertSize` | `fatalAssertThat` | `length`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertLessThan

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Fatally assert value is less than specified value

Syntax

```
fatalAssertLessThan(fatalAssertable,actual,ceiling)
fatalAssertLessThan(fatalAssertable,actual,ceiling,diagnostic)
```

Description

`fatalAssertLessThan(fatalAssertable,actual,ceiling)` fatally asserts that all elements of `actual` are less than all the elements of `ceiling`.

`fatalAssertLessThan(fatalAssertable,actual,ceiling,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test. `actual` must be the same size as `ceiling` unless either one is scalar, at which point scalar expansion occurs.

Default:

`ceiling`

Maximum value, exclusive.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyLessThan`, and replace calls to `verifyLessThan` with `fatalAssertLessThan`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsLessThan;
  assertable.assertThat(actual, IsLessThan(ceiling));
  ```

  There exists more functionality when using the `IsLessThan` constraint directly via `assertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed

and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`fatalAssertGreaterThan` | `fatalAssertGreaterThanOrEqual` | `fatalAssertLessThanOrEqual` | `fatalAssertThat` | `lt` | `matlab.unittest.constraints.IsLessThan` | `matlab.unittest.diagnostics.Diagnostic`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertLessThanOrEqual

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Fatally assert value is less than or equal to specified value

Syntax

```
fatalAssertLessThanOrEqual(fatalAssertable,actual,ceiling)
fatalAssertLessThanOrEqual(fatalAssertable,actual,ceiling,
diagnostic)
```

Description

`fatalAssertLessThanOrEqual(fatalAssertable,actual,ceiling)` fatally asserts that all elements of `actual` are less than or equal to all the elements of `ceiling`.

`fatalAssertLessThanOrEqual(fatalAssertable,actual,ceiling, diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test. `actual` must be the same size as `ceiling` unless either one is scalar, at which point scalar expansion occurs.

Default:

`actual`

The value to test.

Default:

`ceiling`

Maximum value.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyLessThanOrEqual`, and replace calls to `verifyLessThanOrEqual` with `fatalAssertLessThanOrEqual`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsLessThanOrEqualTo;
  fatalAssertable.fatalAssertThat(actual, IsLessThanOrEqualTo(ceiling));
  ```

  There exists more functionality when using the `IsLessThanOrEqualTo` constraint directly via `fatalAssertThat`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`fatalAssertGreaterThan` | `fatalAssertGreaterThanOrEqual` | `fatalAssertLessThan` | `fatalAssertThat` | `le` | `matlab.unittest.constraints.IsLessThanOrEqualTo` | `matlab.unittest.diagnostics.Diagnostic`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertMatches

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Fatally assert string matches specified regular expression

Syntax

```
fatalAssertMatches(fatalAssertable,actual,expression)
fatalAssertMatches(fatalAssertable,actual,expression,diagnostic)
```

Description

`fatalAssertMatches(fatalAssertable,actual,expression)` fatally asserts that `actual` is a string scalar or character vector that matches the regular expression defined by `expression`.

`fatalAssertMatches(fatalAssertable,actual,expression,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expression`

The value to match, specified as a regular expression.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyMatches`, and replace calls to `verifyMatches` with `fatalAssertMatches`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.Matches;
  fatalAssertable.fatalAssertThat(actual, Matches(expression));
  ```

  There exists more functionality when using the `Matches` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed

and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`fatalAssertSubstring` | `fatalAssertThat` | `regexp`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertNotEmpty

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Fatally assert value is not empty

Syntax

```
fatalAssertNotEmpty(fatalAssertable,actual)
fatalAssertNotEmpty(fatalAssertable,actual,diagnostic)
```

Description

`fatalAssertNotEmpty(fatalAssertable,actual)` fatally asserts that `actual` is a non-empty MATLAB value.

`fatalAssertNotEmpty(fatalAssertable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyNotEmpty`, and replace calls to `verifyNotEmpty` with `fatalAssertNotEmpty`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEmpty;
  fatalAssertable.fatalAssertThat(actual, ~IsEmpty());
  ```

  There exists more functionality when using the `IsEmpty` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit

from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`fatalAssertEmpty` | `fatalAssertThat` | `isempty`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertNotEqual

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Fatally assert value is not equal to specified value

Syntax

```
fatalAssertNotEqual(fatalAssertable,actual,notExpected)
fatalAssertNotEqual(fatalAssertable,actual,notExpected,diagnostic)
```

Description

`fatalAssertNotEqual(fatalAssertable,actual,notExpected)` fatally asserts that `actual` is not equal to `notExpected`.

`fatalAssertNotEqual(fatalAssertable,actual,notExpected,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`notExpected`

Value to compare.

Default:

Examples

See examples for `verifyNotEqual`, and replace calls to `verifyNotEqual` with `fatalAssertNotEqual`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEqualTo;
  fatalAssertable.fatalAssertThat(actual, ~IsEqualTo(notExpected));
  ```

  There exists more functionality when using the `IsEqualTo` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`fatalAssertEqual` | `fatalAssertThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertNotSameHandle

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Fatally assert value is not handle to specified instance

Syntax

```
fatalAssertNotSameHandle(fatalAssertable,actual,notExpectedHandle)
fatalAssertNotSameHandle(fatalAssertable,actual,notExpectedHandle,
diagnostic)
```

Description

`fatalAssertNotSameHandle(fatalAssertable,actual,notExpectedHandle)` fatally asserts that `actual` is a different size and/or does not contain the same instances as the `notExpectedHandle` handle array.

`fatalAssertNotSameHandle(fatalAssertable,actual,notExpectedHandle, diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`notExpectedHandle`

The handle array to compare.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyNotSameHandle`, and replace calls to `verifyNotSameHandle` with `fatalAssertNotSameHandle`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsSameHandleAs;
  fatalAssertable.fatalAssertThat(actual, ~IsSameHandleAs(notExpectedHandle));
  ```

  There exists more functionality when using the `IsSameHandleAs` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`fatalAssertSameHandle` | `fatalAssertThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertNumElements

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Fatally assert value has specified element count

Syntax

```
fatalAssertNumElements(fatalAssertable,actual,expectedElementCount)
fatalAssertNumElements(fatalAssertable,actual,expectedElementCount,
diagnostic)
```

Description

`fatalAssertNumElements(fatalAssertable,actual,expectedElementCount)` fatally asserts that `actual` is a MATLAB array with `expectedElementCount` number of elements.

`fatalAssertNumElements(fatalAssertable,actual,expectedElementCount, diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedElementCount`

The expected number of elements in the array.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyNumElements`, and replace calls to `verifyNumElements` with `fatalAssertNumElements`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasElementCount;
  fatalAssertable.fatalAssertThat(actual, HasElementCount(expectedElementCount));
  ```

  There exists more functionality when using the `HasElementCount` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`fatalAssertLength` | `fatalAssertSize` | `fatalAssertThat` | `numel`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertReturnsTrue

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Fatally assert function returns true when evaluated

Syntax

```
fatalAssertReturnsTrue(fatalAssertable,actual)
fatalAssertReturnsTrue(fatalAssertable,actual,diagnostic)
```

Description

`fatalAssertReturnsTrue(fatalAssertable,actual)` fatally asserts that `actual` is a function handle that returns a scalar logical whose value is true.

`fatalAssertReturnsTrue(fatalAssertable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The function handle to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyReturnsTrue`, and replace calls to `verifyReturnsTrue` with `fatalAssertReturnsTrue`.

Tips

- It is a shortcut for quick custom comparison functionality that can be defined quickly, and possibly inline. It can be preferable over simply evaluating the function directly and using `fatalAssertTrue` because the function handle will be shown in the diagnostics, thus providing more insight into the failure condition which is lost when using `fatalAssertTrue`.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.ReturnsTrue;
  fatalAssertable.fatalAssertThat(actual, ReturnsTrue());
  ```

  There exists more functionality when using the `ReturnsTrue` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`fatalAssertThat` | `fatalAssertTrue`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertSameHandle

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Fatally assert two values are handles to same instance

Syntax

```
fatalAssertSameHandle(fatalAssertable,actual,expectedHandle)
fatalAssertSameHandle(fatalAssertable,actual,expectedHandle,
diagnostic)
```

Description

`fatalAssertSameHandle(fatalAssertable,actual,expectedHandle)` fatally asserts that `actual` is the same size and contains the same instances as the `expectedHandle` handle array.

`fatalAssertSameHandle(fatalAssertable,actual,expectedHandle, diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

expectedHandle

The expected handle array.

diagnostic

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifySameHandle`, and replace calls to `verifySameHandle` with `fatalAssertSameHandle`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsSameHandleAs;
  fatalAssertable.fatalAssertThat(actual, IsSameHandleAs(expectedHandle));
  ```

  There exists more functionality when using the `IsSameHandleAs` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test

methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`fatalAssertNotSameHandle` | `fatalAssertThat` | `handle`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertSize

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Fatally assert value has specified size

Syntax

```
fatalAssertSize(fatalAssertable,actual,expectedSize)
fatalAssertSize(fatalAssertable,actual,expectedSize,diagnostic)
```

Description

`fatalAssertSize(fatalAssertable,actual,expectedSize)` fatally asserts that `actual` is a MATLAB array whose size is `expectedSize`.

`fatalAssertSize(fatalAssertable,actual,expectedSize,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedSize`

The expected sizes of each dimension the array.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifySize`, and replace calls to `verifySize` with `fatalAssertSize`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasSize;
  fatalAssertable.fatalAssertThat(actual, HasSize(expectedSize));
  ```

  There exists more functionality when using the `HasSize` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed

and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`fatalAssertLength` | `fatalAssertNumElements` | `fatalAssertThat` | `size`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertSubstring

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Fatally assert string contains specified string

Syntax

```
fatalAssertSubstring(fatalAssertable,actual,substring)
fatalAssertSubstring(fatalAssertable,actual,substring,diagnostic)
```

Description

`fatalAssertSubstring(fatalAssertable,actual,substring)` fatally asserts that `actual` is a string scalar or character vector that contains `substring`.

`fatalAssertSubstring(fatalAssertable,actual,substring,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`substring`

The value to match, specified as a string scalar or character vector.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifySubstring`, and replace calls to `verifySubstring` with `fatalAssertSubstring`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.ContainsSubstring;
  fatalAssertable.fatalAssertThat(actual,...
  ContainsSubstring(substring));
  ```

  There exists more functionality when using the `ContainsSubstring` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed

and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`fatalAssertMatches` | `fatalAssertThat` | `strfind`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertThat

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Fatally assert value meets specified constraint

Syntax

```
fatalAssertThat(fatalAssertable,actual,constraint)
fatalAssertThat(fatalAssertable,actual,constraint,diagnostic)
```

Description

`fatalAssertThat(fatalAssertable,actual,constraint)` fatally asserts that `actual` is a value that satisfies the `constraint` provided.

If the constraint is not satisfied, a fatal assertion failure is produced utilizing only the framework diagnostic generated by the `constraint`.

`fatalAssertThat(fatalAssertable,actual,constraint,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

When using this signature, both the diagnostic information contained within `diagnostic` is used in addition to the diagnostic information provided by the `constraint`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`constraint`

Constraint that the actual value must satisfy to pass the verification, specified as a `matlab.unittest.constraints` instance.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyThat`, and replace calls to `verifyThat` with `fatalAssertThat`.

Tips

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertTrue

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Fatally assert value is true

Syntax

```
fatalAssertTrue(fatalAssertable,actual)
fatalAssertTrue(fatalAssertable,actual,diagnostic)
```

Description

`fatalAssertTrue(fatalAssertable,actual)` fatally asserts that `actual` is a scalar logical with the value of true.

`fatalAssertTrue(fatalAssertable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

See examples for `verifyTrue`, and replace calls to `verifyTrue` with `fatalAssertTrue`.

Tips

- This method passes if and only if the actual value is a scalar logical with a value of true. Therefore, entities such as true valued arrays and nonzero doubles produce qualification failures when used in this method, despite these entities exhibiting "true-like" behavior such as triggering the execution of code inside of "if" statements.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsTrue;
  fatalAssertable.fatalAssertThat(actual, IsTrue());
  ```

  There exists more functionality when using the `IsTrue` constraint directly via `fatalAssertThat`.

  However, this method is optimized for performance and does not construct a new `IsTrue` constraint for each call. Sometimes such use can come at the expense of less diagnostic information. Use the `fatalAssertReturnsTrue` method for a similar approach which may provide better diagnostic information.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`fatalAssertFalse` | `fatalAssertReturnsTrue` | `fatalAssertThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertWarning

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Fatally assert function issues specified warnings

Syntax

```
fatalAssertWarning(fatalAssertable,actual,warningIDs)
fatalAssertWarning(fatalAssertable,actual,warningIDs,diagnostic)
[output1,...,outputN] = fatalAssertWarning( ___ )
```

Description

`fatalAssertWarning(fatalAssertable,actual,warningIDs)` fatally asserts that `actual` issues warnings with the identifiers `warningIDs`.

`fatalAssertWarning(fatalAssertable,actual,warningIDs,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

`[output1,...,outputN] = fatalAssertWarning( ___ )` also returns the output arguments `output1,...,outputN` that are produced when invoking `actual`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The function handle to test.

Default:

`warningIDs`

Warning identifiers, specified as a character vector, cell array of character vectors, or string array.

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Output Arguments

`output1,...,outputN`

Output arguments, 1 through n (if any), from `actual`, returned as any type. The argument type is specified by the `actual` argument list.

Examples

See examples for `verifyWarning`, and replace calls to `verifyWarning` with `fatalAssertWarning`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IssuesWarnings;
  fatalAssertable.fatalAssertThat(actual,IssuesWarnings(cellstr(warningIDs)));
  ```

  There exists more functionality when using the `IssuesWarnings` constraint directly via `fatalAssertThat`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`fatalAssertError` | `fatalAssertThat` | `fatalAssertWarningFree` | `warning`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

fatalAssertWarningFree

Class: `matlab.unittest.qualifications.FatalAssertable`
Package: `matlab.unittest.qualifications`

Fatally assert function issues no warnings

Syntax

```
fatalAssertWarningFree(fatalAssertable,actual)
fatalAssertWarningFree(fatalAssertable,actual,diagnostic)
[output1,...,outputN] = fatalAssertWarningFree( ___ )
```

Description

`fatalAssertWarningFree(fatalAssertable,actual)` fatally asserts that `actual` is a function handle that issues no warnings.

`fatalAssertWarningFree(fatalAssertable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

`[output1,...,outputN] = fatalAssertWarningFree( ___ )` also returns the output arguments `output1,...,outputN` that are produced when invoking `actual`.

Input Arguments

`fatalAssertable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

`actual`

The function handle to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Output Arguments

`output1,...,outputN`

Output arguments, 1 through n (if any), from `actual`, returned as any type. The argument type is specified by the `actual` argument list.

Examples

See examples for `verifyWarningFree`, and replace calls to `verifyWarningFree` with `fatalAssertWarningFree`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IssuesNoWarnings;
  fatalAssertable.fatalAssertThat(actual, IssuesNoWarnings());
  ```

  There exists more functionality when using the `IssuesNoWarnings` constraint directly via `fatalAssertThat`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does

not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. Alternatively,

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. For more information, see `matlab.unittest.qualifications.Verifiable`.
- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

See Also

`fatalAssertThat` | `fatalAssertWarning` | `warning`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

matlab.unittest.qualifications.FatalAssertionFailedException class

Package: `matlab.unittest.qualifications`

Exception used for fatal assertion failures

Description

The `FatalAssertionFailedException` class provides an exception used for fatal assertion failures. This class is used exclusively by the `FatalAssertable` qualification type.

See Also

`FatalAssertable` | `MException`

matlab.unittest.qualifications.QualificationEventData class

Package: `matlab.unittest.qualifications`

Event data for qualification event listeners

Description

The `QualificationEventData` class holds event data for qualification event listeners. Qualification event listeners are callback functions that you register with the testing framework to listen for passing and/or failing qualifications. Qualifications can be assertions, fatal assertions, assumptions, or verifications performed on test content. The corresponding qualification classes define these events. Typically, authors of custom plugins use this class. Only the test framework constructs this class directly.

Properties

`ActualValue`

Value tested to satisfy the qualification logic of the `Constraint`.

`Constraint`

Instance of `matlab.unittest.constraints.Constraint` used for the qualification

When you use a qualification method on a `TestCase` or `Fixture` object, the `Constraint` property contains the underlying constraint used for the qualification. For example, if you use the `verifyEqual` method, the underlying constraint is the `IsEqualTo` constraint. Therefore, if you invoke the constraint's `getDiagnosticFor` method, the diagnostic result can appear different than what the test framework displays.

`TestDiagnostic`

Diagnostic specified in the qualification, represented as a character vector, string, function handle, or instance of the `Diagnostic` class.

`TestDiagnosticResults`

Result of diagnostic specified in the qualification, represented as an array of `DiagnosticResult` instances.

`FrameworkDiagnosticResults`

Result of diagnostic from constraint used for the qualification, represented as an array of `DiagnosticResult` instances.

`Stack`

Function call stack leading up to the qualification event, represented as a structure array.

See Also

`matlab.unittest.diagnostics.DiagnosticResult` | `matlab.unittest.fixtures.Fixture` | `matlab.unittest.qualifications.Assertable` | `matlab.unittest.qualifications.Assumable` | `matlab.unittest.qualifications.FatalAssertable` | `matlab.unittest.qualifications.Verifiable`

Introduced in R2014a

matlab.unittest.qualifications.Verifiable class

Package: `matlab.unittest.qualifications`

Qualification to produce soft failure conditions

Description

The `Verifiable` class provides a qualification to produce soft-failure conditions. Apart from actions performed for failures, the `Verifiable` class works the same as other `matlab.unittest` qualifications.

Upon a verification failure, the `Verifiable` class informs the testing framework of the failure, including all diagnostic information associated with the failure, but continues to execute the currently running test without throwing an `MException`. This is most useful when a failure at the verification point is not fatal to the remaining test content. Often, you use verifications as the primary verification of a Four-Phase Test. Use other qualification types, such as assertions, fatal assertions, and assumptions, to test for violation of preconditions or incorrect test setup.

Since verifications do not throw `MException`s, all test content runs to completion even when verification failures occur. This helps you understand how close a piece of software is to meeting the test suite requirements. Qualification types that throw exceptions do not provide this insight, since once an exception is thrown an arbitrary amount of code remains that is not reached or exercised. Verifications also provide more testing coverage in failure conditions. However, when you overuse verifications, they can produce excess noise for a single failure condition. If a failure condition will cause later qualification points to also fail, use assertions or fatal assertions instead.

Methods

verifyClass	Verify exact class of specified value
verifyEmpty	Verify value is empty
verifyEqual	Verify value is equal to specified value
verifyError	Verify function throws specified exception
verifyFail	Produce unconditional verification failure
verifyFalse	Verify value is false
verifyGreaterThan	Verify value is greater than specified value
verifyGreaterThanOrEqual	Verify value is greater than or equal to specified value
verifyInstanceOf	Verify value is object of specified type
verifyLength	Verify value has specified length
verifyLessThan	Verify value is less than specified value
verifyLessThanOrEqual	Verify value is less than or equal to specified value
verifyMatches	Verify string matches specified regular expression
verifyNotEmpty	Verify value is not empty
verifyNotEqual	Verify value is not equal to specified value
verifyNotSameHandle	Verify value is not handle to specified instance
verifyNumElements	Verify value has specified element count
verifyReturnsTrue	Verify function returns true when evaluated
verifySameHandle	Verify two values are handles to same instance
verifySize	Verify value has specified size
verifySubstring	Verify string contains specified string
verifyThat	Verify value meets given constraint
verifyTrue	Verify value is true
verifyWarning	Verify function issues specified warnings
verifyWarningFree	Verify function issues no warnings

Events

`VerificationFailed`	Triggered upon failing verification. A `QualificationEventData` object is passed to listener callback functions.
`VerificationPassed`	Triggered upon passing verification. A `QualificationEventData` object is passed to listener callback functions.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Write Test Methods Using Verifications

Verifications produce and record failures without throwing an exception, meaning the currently running test runs to completion. This example creates a test case to verify arithmetic operations on objects of the `DocPolynom` example class.

Create the `DocPolynomTest` Test Case. Refer to the following `DocPolynomTest` test case in the subsequent steps in this example, which highlight specific functions in the file.

DocPolynomTest Class Definition File

```
classdef DocPolynomTest < matlab.unittest.TestCase
    % Tests the DocPolynom class.

   properties
      msgEqn = 'Equation under test: ';
   end

   methods (TestClassSetup)
        function addDocPolynomClassToPath(testCase)
            testCase.addTeardown(@path,addpath(fullfile(matlabroot,...
                'help', 'techdoc', 'matlab_oop', 'examples')))
```

```
        end
    end

    methods (Test)
        function testConstructor(testCase)
            p = DocPolynom([1, 0, 1]);
            testCase.verifyClass(p, ?DocPolynom)
        end

        function testAddition(testCase)
            p1 = DocPolynom([1, 0, 1]);
            p2 = DocPolynom([5, 2]);

            actual = p1 + p2;
            expected = DocPolynom([1, 5, 3]);

            msg = [testCase.msgEqn,...
                '(x^2 + 1) + (5*x + 2) = x^2 + 5*x + 3'];
            testCase.verifyEqual(actual, expected, msg)
        end

        function testMultiplication(testCase)
            p1 = DocPolynom([1, 0, 3]);
            p2 = DocPolynom([5, 2]);

            actual = p1 * p2;
            expected = DocPolynom([5, 2, 15, 6]);

            msg = [testCase.msgEqn,...
                '(x^2 + 3) * (5*x + 2) = 5*x^3 + 2*x^2 + 15*x + 6'];
            testCase.verifyEqual(actual, expected, msg)
        end

    end
end
```

To execute the MATLAB commands in this example, add the `DocPolynomTest.m` file to a folder on your MATLAB path.

Write Test to Verify Constructor. Create a function, `testConstructor`, using the `verifyClass` method to test the `DocPolynom` class constructor.

```
function testConstructor(testCase)
    p = DocPolynom([1, 0, 1]);
```

```
    testCase.verifyClass(p, ?DocPolynom)
end
```

Write Tests to Verify Operations. In the `testAddition` function, use the `verifyEqual` method to test the equation `(x^2 + 1) + (5*x + 2) = x^2 + 5*x + 3`. The `verifyEqual` method includes this equation in the `diagnostic` argument.

```
function testAddition(testCase)
    p1 = DocPolynom([1, 0, 1]);
    p2 = DocPolynom([5, 2]);

    actual = p1 + p2;
    expected = DocPolynom([1, 5, 3]);

    msg = [testCase.msgEqn,...
        '(x^2 + 1) + (5*x + 2) = x^2 + 5*x + 3'];
    testCase.verifyEqual(actual, expected, msg)
end
```

The function, `testMultiplication`, tests multiplication operations.

Run the tests in the `DocPolynomTest` test case.

```
tc = DocPolynomTest;
ts = matlab.unittest.TestSuite.fromClass(?DocPolynomTest);
res = run(ts);
```

```
Running DocPolynomTest
...
Done DocPolynomTest
__________
```

All tests passed.

See Also

`Assertable` | `Assumable` | `FatalAssertable` | `QualificationEventData` | `TestCase` | `matlab.unittest.qualifications`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

External Websites

Four-Phase Test

Introduced in R2013a

verifyClass

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Verify exact class of specified value

Syntax

```
verifyClass(verifiable,actual,className)
verifyClass(verifiable,actual,metaClass)
verifyClass( ___ ,diagnostic)
```

Description

`verifyClass(verifiable,actual,className)` verifies that `actual` is a MATLAB value whose class is the class specified by `className`.

`verifyClass(verifiable,actual,metaClass)` verifies that `actual` is a MATLAB value whose class is the class specified by the `meta.class` instance `metaClass`. The instance must be an exact class match. Use `verifyInstanceOf` to verify inclusion in a class hierarchy.

`verifyClass( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`className`

Name of class, specified as a character vector.

Default:

`metaClass`

An instance of `meta.class`.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test a Class

These interactive tests verify the class of the number, 5.

Create a `TestCase` object and the value to test.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
actvalue = 5;
```

Verify class of `actvalue` is double.

```
verifyClass(testCase, actvalue, 'double');
```

```
Interactive verification passed.
```

Verify class of `actvalue` is char.

```
verifyClass(testCase, actvalue, 'char');
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyClass failed.
--> The value's class is incorrect.

    Actual Class:
        double
    Expected Class:
        char

Actual double:
         5
```

Test fails.

Test a Function Handle

These interactive tests verify function handles, specified as a meta.class instance, `?function_handle`.

Create a `TestCase` object.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Create a function handle.

```
fh = @sin;
verifyClass(testCase, fh, ?function_handle);
```

```
Interactive verification passed.
```

Test the function name.

```
fh = 'sin';
verifyClass(testCase, fh, ?function_handle);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyClass failed.
--> The value's class is incorrect.

    Actual Class:
        char
    Expected Class:
        function_handle

Actual char:
    sin
```

Test fails.

Test a Derived Class

Verify that a derived class is not the same class as its base class.

Create a class, `BaseExample`.

```
classdef BaseExample
end
```

Create a derived class, `DerivedExample`.

```
classdef DerivedExample < BaseExample
end
```

Verify the classes are not equal.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyClass(testCase, DerivedExample(), ?BaseExample);
```

```
Interactive verification failed.
```

```
----------------------
Framework Diagnostic:
----------------------
verifyClass failed.
--> The value's class is incorrect.

    Actual Class:
        DerivedExample
    Expected Class:
        BaseExample

Actual DerivedExample:
      DerivedExample with no properties.
```

Test fails.

Test Class of Output Value

Use `verifyClass` to test the `add5` function returns a `double` value.

Function for unit testing:

```
type add5.m
```

```
function res = add5(x)
% ADD5 Increment input by 5.
if ~isa(x,'numeric')
    error('add5:InputMustBeNumeric','Input must be numeric.')
end
res = x + 5;
end
```

`TestCase` class containing test methods:

```
type Add5Test.m
```

```
classdef Add5Test < matlab.unittest.TestCase
    methods (Test)
        function testDoubleOut(testCase)
            actOutput = add5(1);
            testCase.verifyClass(actOutput,'double')
        end
        function testNonNumericInput(testCase)
```

```
            testCase.verifyError(@()add5('0'),'add5:InputMustBeNumeric')
        end
    end
end
```

Create a test suite from the `Add5Test` class file.

```
suite = matlab.unittest.TestSuite.fromFile('Add5Test.m')
```

```
suite 
  1x2 Test array with properties:

    Name
    ProcedureName
    TestClass
    BaseFolder
    Parameterization
    SharedTestFixtures
    Tags

Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

Run the tests.

```
result = run(suite);
```

```
Running Add5Test
..
Done Add5Test
__________
```

Tips

- The method is functionally equivalent to the following methods:

  ```
  import matlab.unittest.constraints.IsOfClass;
  verifiable.verifyThat(actual, IsOfClass(className));
  verifiable.verifyThat(actual, IsOfClass(metaClass));
  ```

  There exists more functionality when using the `IsOfClass` constraint directly via `verifyThat`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,
  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`matlab.unittest.constraints` | `matlab.unittest.constraints.IsOfClass` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications` | `verifyInstanceOf` | `verifyThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

verifyEmpty

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Verify value is empty

Syntax

```
verifyEmpty(verifiable,actual)
verifyEmpty( ___ ,diagnostic)
```

Description

`verifyEmpty(verifiable,actual)` verifies that `actual` is an empty MATLAB value.

`verifyEmpty( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test for Empty Character Vectors

Create a `TestCase` object.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Verify the empty character.

```
verifyEmpty(testCase, '');
```

```
Verification passed.
```

Test for Empty Arrays

An array with any zero dimension is empty.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyEmpty(testCase, ones(2, 5, 0, 3));
```

```
Interactive verification passed.
```

```
verifyEmpty(testCase, [2 3], 'Array is not empty.');
```

```
Interactive verification failed.

----------------
Test Diagnostic:
----------------
```

```
Array is not empty.

----------------------
Framework Diagnostic:
----------------------
verifyEmpty failed.
--> The value must be empty.
--> The value has a size of [1  2].

Actual double:
         2     3
```

Test failed.

Test for Empty Cell Arrays

Test empty cell array.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyEmpty(testCase,{},'Cell array is not empty.');
```

```
Interactive verification passed.
```

A cell array of empty arrays is not empty.

```
verifyEmpty(testCase,{[],[],[]},'Cell array is not empty.');
```

```
Interactive verification failed.

----------------
Test Diagnostic:
----------------
Cell array is not empty.

----------------------
Framework Diagnostic:
----------------------
verifyEmpty failed.
--> The value must be empty.
--> The value has a size of [1  3].
```

```
Actual cell:
        []    []    []
```

Test for Empty Test Suite

Test for empty object, `emptyTestSuite`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
emptyTestSuite = matlab.unittest.TestSuite.empty;
verifyEmpty(testCase, emptyTestSuite)
```

```
Verification passed.
```

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEmpty;
  verifiable.verifyThat(actual, IsEmpty());
  ```

  There exists more functionality when using the `IsEmpty` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`isempty` | `matlab.unittest.constraints` | `matlab.unittest.constraints.IsEmpty` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications` | `verifyNotEmpty` | `verifyThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

verifyEqual

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Verify value is equal to specified value

Syntax

```
verifyEqual(verifiable,actual,expected)
verifyEqual( ___ ,Name,Value)
verifyEqual( ___ ,diagnostic)
```

Description

`verifyEqual(verifiable,actual,expected)` verifies that `actual` is strictly equal to `expected`. If `expected` is not a MATLAB or Java object, `actual` and `expected` must have the same class, size, and value for the verification to pass. `verifyEqual` compares `actual` and `expected` in the same way as the `IsEqualTo` constraint.

`verifyEqual( ___ ,Name,Value)` verifies equality with additional options specified by one or more `Name,Value` pair arguments.

`verifyEqual( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expected`

Expected value.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`AbsTol`

Absolute tolerance, specified as a numeric array. The tolerance is applied only to values of the same data type. The value can be a scalar or array the same size as the actual and expected values.

For an absolute tolerance to be satisfied, `abs(expected-actual) <= absTol` must be true.

Default:

`RelTol`

Relative tolerance, specified as a numeric array. The tolerance is applied only to values of the same data type. The value can be a scalar or array the same size as the actual and expected values.

For a relative tolerance to be satisfied, `abs(expected-actual) <= relTol.*abs(expected)` must be true.

Default:

Examples

Compare Numeric Values

Numeric values are equal if they are of the same class with equivalent size, complexity, and sparsity.

At the command prompt, create a test case for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Verify that a numeric value is equal to itself.

```
verifyEqual(testCase,5,5)
```

```
Verification passed.
```

Verify if actual and expected values have equal sizes.

```
verifyEqual(testCase,[5 5],5)
```

```
Verification failed.
    ----------------------
    Framework Diagnostic:
    ----------------------
    verifyEqual failed.
    --> Sizes do not match.

        Actual size:
             1     2
        Expected size:
```

```
             1     1

    Actual Value:
         5     5
    Expected Value:
         5
```

Test Classes

Verify if an `int8` is equal to an `int16`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyEqual(testCase,int8(5),int16(5))
```

```
Verification failed.
    ----------------------
    Framework Diagnostic:
    ----------------------
    verifyEqual failed.
    --> Classes do not match.

        Actual Class:
            int8
        Expected Class:
            int16

    Actual Value:
      int8

       5
    Expected Value:
      int16

       5
```

Test Cell Arrays

Elements of equal cell arrays must match in class, size, and value.

Verify that a cell array is equal to itself.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyEqual(testCase,{'cell',struct,5},{'cell',struct,5})
```

```
Verification passed.
```

Test Numeric Tolerances

Verify if `4.95` is equal to `5`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyEqual(testCase,4.95,5)
```

```
Verification failed.
    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyEqual failed.
    --> The numeric values are not equal using "isequaln".
    --> Failure table:
            Actual    Expected           Error              RelativeError
            ______    ________    __________________    ___________________

             4.95        5        -0.0499999999999998    -0.00999999999999996

    Actual Value:
       4.950000000000000
    Expected Value:
         5
```

Verify that the difference between the actual value `1.5` and the expected value `2` is within `1`.

```
verifyEqual(testCase,1.5,2,'AbsTol',1)
```

```
Verification passed.
```

Verify if the difference between the actual and expected values is less than 10%. Display the diagnostic information upon failure.

```
verifyEqual(testCase,1.5,2,'RelTol',0.1,...
    'Difference between actual and expected exceeds relative tolerance')
```

```
Verification failed.
    ----------------
    Test Diagnostic:
    ----------------
    Difference between actual and expected exceeds relative tolerance
    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyEqual failed.
```

```
--> The numeric values are not equal using "isequaln".
--> The error was not within relative tolerance.
--> Failure table:
        Actual    Expected    Error    RelativeError    RelativeTolerance
        ______    ________    _____    _____________    _________________

         1.5         2        -0.5         -0.25               0.1

Actual Value:
   1.500000000000000
Expected Value:
     2
```

Tips

- This method is functionally equivalent to any of the following:

  ```
  import matlab.unittest.constraints.IsEqualTo;
  verifiable.verifyThat(actual, IsEqualTo(expected));

  import matlab.unittest.constraints.IsEqualTo;
  import matlab.unittest.constraints.AbsoluteTolerance;
  verifiable.verifyThat(actual, IsEqualTo(expected, ...
      'Within', AbsoluteTolerance(abstol)));

  import matlab.unittest.constraints.IsEqualTo;
  import matlab.unittest.constraints.RelativeTolerance;
  verifiable.verifyThat(actual, IsEqualTo(expected, ...
      'Within', RelativeTolerance(reltol)));

  import matlab.unittest.constraints.IsEqualTo;
  import matlab.unittest.constraints.AbsoluteTolerance;
  import matlab.unittest.constraints.RelativeTolerance;
  verifiable.verifyThat(actual, IsEqualTo(expected, ...
      'Within', AbsoluteTolerance(abstol) | RelativeTolerance(reltol)));
  ```

  There exists more functionality when using the `IsEqualTo`, `AbsoluteTolerance`, and `RelativeTolerance` constraints directly via `verifyThat`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`matlab.unittest.constraints` | `matlab.unittest.constraints.AbsoluteTolerance` | `matlab.unittest.constraints.IsEqualTo` | `matlab.unittest.constraints.RelativeTolerance` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications` | `verifyNotEqual` | `verifyThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

verifyError

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Verify function throws specified exception

Syntax

```
verifyError(verifiable,actual,identifier)
verifyError(verifiable,actual,metaClass)
verifyError( ___ ,diagnostic)
[output1,...,outputN] = verifyError( ___ )
```

Description

`verifyError(verifiable,actual,identifier)` verifies that `actual` is a function handle that throws an exception with an error identifier that is equal to `identifier`.

`verifyError(verifiable,actual,metaClass)` verifies that `actual` is a function handle that throws an exception whose type is defined by the `meta.class` instance specified in `metaClass`. This method does not require the instance to be an exact class match, but rather it must be in the specified class hierarchy, and that hierarchy must include the `MException` class.

`verifyError( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

`[output1,...,outputN] = verifyError( ___ )` returns multiple output arguments from the invocation of the function handle `actual`. Use this syntax to control the number of output arguments requested from the function handle. If the function handle errors, all outputs are of type `missing`. Otherwise, `output1,...,outputN` are output values from `actual`. You can use this syntax with any of the input arguments of the previous syntaxes.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`identifier`

Error identifier, specified as a character vector.

Default:

`metaClass`

An instance of `meta.class`.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test for Error IDs

```
testCase = matlab.unittest.TestCase.forInteractiveUse;

% Passing scenarios
%%%%%%%%%%%%%%%%%%%%%%
verifyError(testCase, @() error('SOME:error:id','Error!'), 'SOME:error:id');
verifyError(testCase, @testCase.assertFail, ...
    ?matlab.unittest.qualifications.AssertionFailedException);

% Failing scenarios
%%%%%%%%%%%%%%%%%%%%%%
verifyError(testCase, 5, 'some:id', '5 is not a function handle');
verifyError(testCase, @testCase.verifyFail, ...
    ?matlab.unittest.qualifications.AssertionFailedException, ...
    'Verifications dont throw exceptions.');
verifyError(testCase, @() error('SOME:id'), 'OTHER:id', 'Wrong id');
verifyError(testCase, @() error('whoops'), ...
    ?matlab.unittest.qualifications.AssertionFailedException, ...
    'Wrong type of exception thrown');
```

Test Error Condition

Create `testNonNumericInput` to test if function throws expected error message, `add5:InputMustBeNumeric`, for unexpected condition, input is `char`.

Function for unit testing:

```
function res = add5(x)
% ADD5 Increment input by 5.
if ~isa(x,'numeric')
    error('add5:InputMustBeNumeric','Input must be numeric.')
end
res = x + 5;
end
```

`TestCase` class containing test methods:

```
classdef Add5Test < matlab.unittest.TestCase
    methods (Test)
        function testDoubleOut(testCase)
            actOutput = add5(1);
            testCase.verifyClass(actOutput,'double')
```

```
        end
        function testNonNumericInput(testCase)
            testCase.verifyError(@()add5('0'),'add5:InputMustBeNumeric')
        end
    end
end
```

Create a test suite from the `Add5Test` class file.

```
suite = matlab.unittest.TestSuite.fromFile('Add5Test.m')
```

```
result = run(suite);
```

```
Running Add5Test
..
Done Add5Test
__________
```

Verify Function Errors with Too Many Requested Outputs

In your current working folder, create the following function to test. The function returns a single output.

```
function y = plusOne(x)
    y = x+1;
end
```

At the command prompt, call `plusOne` with too many output arguments.

```
[~,~] = plusOne(41);
```

```
Error using plusOne
Too many output arguments.
```

View the error message and identifier.

```
MException.last
```

```
ans =

  MException with properties:

    identifier: 'MATLAB:TooManyOutputs'
       message: 'Too many output arguments.'
         cause: {0×1 cell}
         stack: [0×1 struct]
```

Verify that if the `plusOne` function is called with too many requested output arguments, it errors and the error has the identifier `MATLAB:TooManyOutputs`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
[~,~] = verifyError(testCase,@() plusOne(41),'MATLAB:TooManyOutputs');
```

```
Interactive verification passed.
```

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.Throws;
  verifiable.verifyThat(actual, Throws(identifier));
  verifiable.verifyThat(actual, Throws(metaClass));
  ```

  There exists more functionality when using the `Throws` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`MException` | `error` | `matlab.unittest.constraints` | `matlab.unittest.constraints.Throws` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications` | `verifyThat` | `verifyWarning`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

verifyFail

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Produce unconditional verification failure

Syntax

```
verifyFail(verifiable)
verifyFail(verifiable,diagnostic)
```

Description

`verifyFail(verifiable)` produces an unconditional verification failure when encountered.

`verifyFail(verifiable,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle

- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test for Failure Condition

An example of where this method may be used is in a callback function that should not be executed in a given scenario. A test can confirm this does not occur by unconditionally performing a failure if the code path is reached.

Create a handle class, `MyHandle`, with a `SomethingHappened` event.

```
classdef MyHandle < handle
    events
        SomethingHappened
    end
end
```

Create a file, `ListenerTest`, on your MATLAB path that contains the following `TestCase` class.

```
classdef ListenerTest < matlab.unittest.TestCase
    methods(Test)
        function testDisabledListeners(testCase)
            h = MyHandle;

            % Add a listener to a test helper method
            listener = h.addlistener('SomethingHappened', ...
                @testCase.shouldNotGetCalled);

            % Passing scenario (code path is not reached)
            %%%%%%%%%%%%%%%%%%%%
            % Disabled listener should not invoke callbacks
            listener.Enabled = false;
            h.notify('SomethingHappened');

            % Failing scenario (code path is reached)
            %%%%%%%%%%%%%%%%%%%%
```

```
            % Enabled listener invoke callback and fail
            listener.Enabled = true;
            h.notify('SomethingHappened');
        end
    end

    methods
        function shouldNotGetCalled(testCase, ~, ~)
            % A test helper callback method that should not execute
            testCase.verifyFail('This listener callback should not have executed');
        end
    end

end
```

From the command prompt, run the test.

```
run(ListenerTest);
```

```
Running ListenerTest

================================================================================
Verification failed in ListenerTest/testDisabledListeners.

    ----------------
    Test Diagnostic:
    ----------------
    This listener callback should not have executed

    ------------------
    Stack Information:
    ------------------
    In C:\Desktop\ListenerTest.m (ListenerTest.shouldNotGetCalled) at 27
    In C:\\Desktop\ListenerTest.m (@(varargin)testCase.shouldNotGetCalled(varargin{:}))
    In C:\Desktop\ListenerTest.m (ListenerTest.testDisabledListeners) at 20
================================================================================
.
Done ListenerTest
_________

Failure Summary:

     Name                                Failed  Incomplete  Reason(s)
```

```
================================================================================
 ListenerTest/testDisabledListeners    X                Failed by verification.
```

Tips

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,
  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`matlab.unittest.constraints` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

verifyFalse

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Verify value is false

Syntax

```
verifyFalse(verifiable,actual)
verifyFalse( ___ ,diagnostic)
```

Description

`verifyFalse(verifiable,actual)` verifies that `actual` is a scalar logical with the value of false.

`verifyFalse( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test MATLAB Logical Functions

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Test `true`.

```
verifyFalse(testCase, true);
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
verifyFalse failed.
--> The value must evaluate to "false".

Actual Value:
         1
```

Test failed.

Test `false`.

```
verifyFalse(testCase, false);
```

```
Interactive verification passed.
```

Test the Value 0

The number `0` is a `double` value, not a logical value.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyFalse(testCase, 0);
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
verifyFalse failed.
--> The value must be logical. It is of type "double".

Actual Value:
         0
```

Test failed.

Test Array of Logical Values

To be false, the value must be scalar.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyFalse(testCase, [false false false]);
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
verifyFalse failed.
--> The value must be scalar. It has a size of [1  3].

Actual Value:
         0     0     0
```

Test failed.

Test an array of mixed logical values.

```
verifyFalse(testCase, [false true false], ...
    'A mixed array of logicals is not the one false value');
```

```
Interactive verification failed.

----------------
Test Diagnostic:
----------------
A mixed array of logicals is not the one false value

---------------------
Framework Diagnostic:
---------------------
verifyFalse failed.
--> The value must be scalar. It has a size of [1  3].

Actual Value:
         0     1     0
```

Test failed.

Tips

- This method passes if and only if the actual value is a scalar logical with a value of false. Therefore, entities such as empty arrays, false valued arrays, and zero doubles produce failures when used in this method, despite these entities exhibiting "false-like" behavior such as bypassing the execution of code inside of "if" statements.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsFalse;
  verifiable.verifyThat(actual, IsFalse());
  ```

  There exists more functionality when using the `IsFalse` constraint directly via `verifyThat`.
- Unlike `verifyTrue`, this method may create a new constraint for each call. For performance critical uses, consider using `verifyTrue`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`matlab.unittest.constraints` | `matlab.unittest.constraints.IsFalse` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications` | `verifyThat` | `verifyTrue`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

verifyGreaterThan

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Verify value is greater than specified value

Syntax

```
verifyGreaterThan(verifiable,actual,floor)
verifyGreaterThan( ___ ,diagnostic)
```

Description

`verifyGreaterThan(verifiable,actual,floor)` verifies that all elements of `actual` are greater than all the elements of `floor`.

`verifyGreaterThan( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test. `actual` must be the same size as `floor` unless either one is scalar, at which point scalar expansion occurs.

`floor`

Minimum value, exclusive.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Compare Two Numbers

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Verify 3 is greater than 2.

```
verifyGreaterThan(testCase, 3, 2);
```

```
Interactive verification passed.
```

Test if 5 is greater than 9.

```
verifyGreaterThan(testCase, 5, 9);
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
verifyGreaterThan failed.
--> The value must be greater than the minimum value.

Actual Value:
```

```
         5
Minimum Value (Exclusive):
         9
```

Test failed.

Compare an Array to a Scalar

Test if each element is greater than the `FLOOR` value, `2`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyGreaterThan(testCase, [5 6 7], 2);
```

```
Interactive verification passed.
```

Test if value `5` is greater than each element in the `FLOOR` array, `[1 2 3]`.

```
verifyGreaterThan(testCase, 5, [1 2 3]);
```

```
Interactive verification passed.
```

Test if each element in the matrix is greater than the `FLOOR` value, `4`.

```
verifyGreaterThan(testCase, [1 2 3; 4 5 6], 4);
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
verifyGreaterThan failed.
--> Each element must be greater than the minimum value.

    Failing Indices:
             1     2     3     5

Actual Value:
         1     2     3
         4     5     6
Minimum Value (Exclusive):
         4
```

Test failed.

Compare Arrays

Test if each element is greater than each corresponding element of the `FLOOR` array, `[4 -9 0]`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyGreaterThan(testCase, [5 -3 2], [4 -9 0]);
```

```
Interactive verification passed.
```

Compare an array to itself.

```
verifyGreaterThan(testCase, eye(2), eye(2));
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyGreaterThan failed.
--> Each element must be greater than each corresponding element of the minimum value a

    Failing Indices:
             1     2     3     4

Actual Value:
         1     0
         0     1
Minimum Value (Exclusive):
         1     0
         0     1
```

Test failed.

Tips

- This method is functionally equivalent to:

  ```
  matlab.unittest.constraints.IsGreaterThan;
  verifiable.verifyThat(actual, IsGreaterThan(floor));
  ```

There exists more functionality when using the `IsGreaterThan` constraint directly via `verifyThat`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`gt` | `matlab.unittest.constraints` | `matlab.unittest.constraints.IsGreaterThan` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications` | `verifyGreaterThanOrEqual` | `verifyLessThan` | `verifyLessThanOrEqual` | `verifyThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

verifyGreaterThanOrEqual

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Verify value is greater than or equal to specified value

Syntax

```
verifyGreaterThanOrEqual(verifiable,actual,floor)
verifyGreaterThanOrEqual( ___ ,diagnostic)
```

Description

`verifyGreaterThanOrEqual(verifiable,actual,floor)` verifies that all elements of `actual` are greater than or equal to all the elements of `floor`.

`verifyGreaterThanOrEqual( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test. `actual` must be the same size as `floor` unless either one is scalar, at which point scalar expansion occurs.

`floor`

Minimum value.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Compare Two Numbers

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Verify 3 is greater than 2.

```
verifyGreaterThanOrEqual(testCase, 3, 2);
```

```
Interactive verification passed.
```

Verify 3 is greater than or equal to 3.

```
verifyGreaterThanOrEqual(testCase, 3, 3);
```

```
Interactive verification passed.
```

Test if 5 is greater than 9.

```
verifyGreaterThanOrEqual(testCase, 5, 9);
```

```
Interactive verification failed.

---------------------
```

```
Framework Diagnostic:
---------------------
verifyGreaterThanOrEqual failed.
--> The value must be greater than or equal to the minimum value.

Actual Value:
         5
Minimum Value (Inclusive):
         9
```

Test failed.

Compare an Array to a Scalar

Test if each element is greater than or equal to the `FLOOR` value, `2`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyGreaterThanOrEqual(testCase, [5 2 7], 2);
```

```
Interactive verification passed.
```

Test if each element in the matrix is greater than or equal to the `FLOOR` value, `4`.

```
verifyGreaterThanOrEqual(testCase, [1 2 3; 4 5 6], 4);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyGreaterThanOrEqual failed.
--> Each element must be greater than or equal to the minimum value.

    Failing Indices:
             1     3     5

Actual Value:
         1     2     3
         4     5     6
Minimum Value (Inclusive):
         4
```

Compare Arrays

Test if each element is greater than or equal to each corresponding element of the `FLOOR` array, `[4 -3 0]`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyGreaterThanOrEqual(testCase, [5 -3 2], [4 -3 0]);
```

```
Interactive verification passed.
```

Compare an array to itself.

```
verifyGreaterThanOrEqual(testCase, eye(2), eye(2));
```

```
Interactive verification passed.
```

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsGreaterThanOrEqualTo;
  verifiable.verifyThat(actual, IsGreaterThanOrEqualTo(floor));
  ```

  There exists more functionality when using the `IsGreaterThanOrEqualTo` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`ge` | `matlab.unittest.constraints` | `matlab.unittest.constraints.IsGreaterThanOrEqualTo` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications` | `verifyGreaterThan` | `verifyLessThan` | `verifyLessThanOrEqual` | `verifyThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

verifyInstanceOf

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Verify value is object of specified type

Syntax

```
verifyInstanceOf(verifiable,actual,className)
verifyInstanceOf(verifiable,actual,metaClass)
verifyInstanceOf( ___ ,diagnostic)
```

Description

`verifyInstanceOf(verifiable,actual,className)` verifies that `actual` is a MATLAB value whose class is the class specified by `className`.

`verifyInstanceOf(verifiable,actual,metaClass)` verifies that `actual` is a MATLAB value whose class is the class specified by the `meta.class` instance `metaClass`.

`verifyInstanceOf( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`className`

Name of class, specified as a character vector.

Default:

`metaClass`

An instance of `meta.class`.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test a Class

These interactive tests verify the class of the number, 5.

Create a `TestCase` object and the value to test.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
actvalue = 5;
```

Verify `actvalue` is an instance of class `double`.

```
verifyInstanceOf(testCase, actvalue, 'double');
```

```
Interactive verification passed.
```

Verify if `actvalue` is an instance of `char`.

```
verifyInstanceOf(testCase, 5, 'char');
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
verifyInstanceOf failed.
--> The value must be an instance of the expected type.

    Actual Class:
        double
    Expected Type:
        char

Actual Value:
         5
```

Test failed.

Test a Function Handle

These tests verify function handles, specified as a meta.class instance, `?function_handle`.

Create a function handle.

```
fh = @sin;
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyInstanceOf(testCase, fh, ?function_handle);
```

```
Interactive verification passed.
```

Test the function name.

```
fh = 'sin';
verifyInstanceOf(testCase, fh, ?function_handle);
```

```
Interactive verification failed.
```

```
----------------------
Framework Diagnostic:
----------------------
verifyInstanceOf failed.
--> The value must be an instance of the expected type.

    Actual Class:
        char
    Expected Type:
        function_handle

Actual Value:
    sin
```

Test failed.

Test a Derived Class

Verify that a derived class is not the same class as its base class.

Create a class, `BaseExample`.

```
classdef BaseExample
end
```

Create a derived class, `DerivedExample`.

```
classdef DerivedExample < BaseExample
end
```

Verify `DerivedExample` is an instance of `BaseExample`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
testCase.verifyInstanceOf(DerivedExample(), ?BaseExample);
```

```
Interactive verification passed.
```

Verify `BaseExample` is not an instance of `DerivedExample`.

```
testCase.verifyInstanceOf(BaseExample(), ?DerivedExample);
```

```
Interactive verification failed.
```

```
---------------------
Framework Diagnostic:
---------------------
verifyInstanceOf failed.
--> The value must be an instance of the expected type.

    Actual Class:
        BaseExample
    Expected Type:
        DerivedExample

Actual Value:
      BaseExample with no properties.
```

Test failed.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsInstanceOf;
  verifiable.verifyThat(actual, IsInstanceOf(className));
  verifiable.verifyThat(actual, IsInstanceOf(metaClass));
  ```

  There exists more functionality when using the `IsInstanceOf` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed

and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`isa` | `matlab.unittest.constraints` | `matlab.unittest.constraints.IsInstanceOf` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications` | `verifyClass` | `verifyThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

verifyLength

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Verify value has specified length

Syntax

```
verifyLength(verifiable,actual,expectedLength)
verifyLength( ___ ,diagnostic)
```

Description

`verifyLength(verifiable,actual,expectedLength)` verifies that `actual` is a MATLAB array whose length is `expectedLength`.

`verifyLength( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedLength`

The length of an array is defined as the largest dimension of that array.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test Array Lengths

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Verify length of array is the expected value, 5.

```
verifyLength(testCase, ones(2, 5, 3), 5, 'User diagnostic');
```

```
Interactive verification passed.
```

Length of array is not the expected value, 3.

```
verifyLength(testCase, [2 3], 3);
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
verifyLength failed.
--> The array has an incorrect length.

    Actual Length:
```

```
             2
    Expected Length:
             3

Actual Array:
         2     3
```

Test failed.

The length of a 2x3 array is 3.

```
verifyLength(testCase, [1 2 3; 4 5 6], 3);
```

```
Interactive verification passed.
```

Verify the length of a 2x3 array is not the number of elements, 6.

```
verifyLength(testCase, [1 2 3; 4 5 6], 6);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyLength failed.
--> The array has an incorrect length.

    Actual Length:
             3
    Expected Length:
             6

Actual Array:
         1     2     3
         4     5     6
```

Test failed.

```
verifyLength(testCase, eye(2), 4);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyLength failed.
```

```
--> The array has an incorrect length.

    Actual Length:
             2
    Expected Length:
             4

Actual Array:
         1     0
         0     1
```

Test Cell Array Lengths

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyLength(testCase, {'someText', 'someotherText'}, 2);
```

```
Interactive verification passed.
```

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasLength;
  verifiable.verifyThat(actual, HasLength(expectedLength));
  ```

  There exists more functionality when using the `HasLength` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test

methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`length` | `matlab.unittest.constraints` | `matlab.unittest.constraints.HasLength` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications` | `verifyNumElements` | `verifySize` | `verifyThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

verifyLessThan

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Verify value is less than specified value

Syntax

```
verifyLessThan(verifiable,actual,ceiling)
verifyLessThan( ___ ,diagnostic)
```

Description

`verifyLessThan(verifiable,actual,ceiling)` verifies that all elements of `actual` are less than all the elements of `ceiling`.

`verifyLessThan( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test. `actual` must be the same size as `ceiling` unless either one is scalar, at which point scalar expansion occurs.

Default:

`ceiling`

Maximum value, exclusive.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Compare Two Numbers

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Verify 2 is less than 3.

```
verifyLessThan(testCase, 2, 3);
```

```
Interactive verification passed.
```

Test if 9 is less than 5.

```
verifyLessThan(testCase, 9, 5);
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
verifyLessThan failed.
--> The value must be less than the maximum value.

Actual Value:
```

```
         9
Maximum Value (Exclusive):
         5
```

Test failed.

Compare an Array to a Scalar

Test if each element is less than the `CEILING` value, `9`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyLessThan(testCase, [5 6 7], 9);
```

```
Interactive verification passed.
```

Test if each element in the matrix is less than the `CEILING` value, `4`.

```
verifyLessThan(testCase, [1 2 3; 4 5 6], 4);
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
verifyLessThan failed.
--> Each element must be less than the maximum value.

    Failing Indices:
             2     4     6

Actual Value:
         1     2     3
         4     5     6
Maximum Value (Exclusive):
         4
```

Test failed.

Compare Arrays

Test if each element is less than each corresponding element of the `CEILING` array, `[7 -1 8]`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyLessThan(testCase, [5 -3 2], [7 -1 8]);
```

```
Interactive verification passed.
```

Compare an array to itself.

```
verifyLessThan(testCase, eye(2), eye(2));
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
verifyLessThan failed.
--> Each element must be less than each corresponding element of the maximum value arra

    Failing Indices:
             1     2     3     4

Actual Value:
         1     0
         0     1
Maximum Value (Exclusive):
         1     0
         0     1
```

Test failed.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsLessThan;
  verifiable.verifyThat(actual, IsLessThan(ceiling));
  ```

  There exists more functionality when using the `IsLessThan` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit

from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

- Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`lt` | `matlab.unittest.constraints` | `matlab.unittest.constraints.IsLessThan` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications` | `verifyGreaterThan` | `verifyGreaterThanOrEqual` | `verifyLessThanOrEqual` | `verifyThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

verifyLessThanOrEqual

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Verify value is less than or equal to specified value

Syntax

```
verifyLessThanOrEqual(verifiable,actual,ceiling)
verifyLessThanOrEqual( ___ ,diagnostic)
```

Description

`verifyLessThanOrEqual(verifiable,actual,ceiling)` verifies that all elements of `actual` are less than or equal to all the elements of `ceiling`.

`verifyLessThanOrEqual( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`ceiling`

Maximum value.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Compare Two Numbers

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Verify 2 is less than 3.

```
verifyLessThanOrEqual(testCase, 2, 3);
```

```
Interactive verification passed.
```

Verify 3 is less than or equal to 3.

```
verifyLessThanOrEqual(testCase, 3, 3);
```

```
Interactive verification passed.
```

Test if 9 is less than 5.

```
verifyLessThanOrEqual(testCase, 9, 5);
```

```
Interactive verification failed.

----------------------
```

```
Framework Diagnostic:
---------------------
verifyLessThanOrEqual failed.
--> The value must be less than or equal to the maximum value.

Actual Value:
         9
Maximum Value (Inclusive):
         5
```

Test failed.

Compare an Array to a Scalar

Test if each element is less than or equal to the `ceiling` value, 7.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyLessThanOrEqual(testCase, [5 2 7], 7);
```

```
Interactive verification passed.
```

Test if each element in the matrix is less than or equal to the `ceiling` value, 4.

```
verifyLessThanOrEqual(testCase, [1 2 3; 4 5 6], 4);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyLessThanOrEqual failed.
--> Each element must be less than or equal to the maximum value.

    Failing Indices:
              4     6

Actual Value:
         1     2     3
         4     5     6
Maximum Value (Inclusive):
         4
```

Test failed.

Compare Arrays

Test if each element is less than or equal to each corresponding element of the `ceiling` array, `[5 -3 8]`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyLessThanOrEqual(testCase, [5 -3 2], [5 -3 8]);
```

```
Interactive verification passed.
```

Compare an array to itself.

```
verifyLessThanOrEqual(testCase, eye(2), eye(2));
```

```
Interactive verification passed.
```

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsLessThanOrEqualTo;
  verifiable.verifyThat(actual, IsLessThanOrEqualTo(ceiling));
  ```

  There exists more functionality when using the `IsLessThanOrEqualTo` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,
  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test

methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`le` | `matlab.unittest.constraints` | `matlab.unittest.constraints.IsLessThanOrEqualTo` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications` | `verifyGreaterThan` | `verifyGreaterThanOrEqual` | `verifyLessThan` | `verifyThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

verifyMatches

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Verify string matches specified regular expression

Syntax

```
verifyMatches(verifiable,actual,expression)
verifyMatches( ___ ,diagnostic)
```

Description

`verifyMatches(verifiable,actual,expression)` that `actual` is a string scalar or character vector that matches the regular expression defined by `expression`.

`verifyMatches( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The string to test.

Default:

`expression`

The value to match, specified as a regular expression.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test for String Matches

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Verify that text matches a regular expression.

```
verifyMatches(testCase, 'Some Text', 'Some [Tt]ext', ...
    'My result should have matched the expression');
```

```
Interactive verification passed.
```

```
verifyMatches(testCase, 'Another message', '(Some |An)other');
```

```
Interactive verification passed.
```

```
verifyMatches(testCase, 'Another 3 messages', '^Another \d+ messages?$');
```

```
Interactive verification passed.
```

```
verifyMatches(testCase, '3 more messages', '\d+ messages?');
```

```
Interactive verification failed.
```

```
----------------------
Framework Diagnostic:
----------------------
verifyMatches failed.
--> The value does not match the regular expression.

Actual char:
    3 more messages
Regular Expression:
    \d+ messages?
```

Test failed.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.Matches;
  verifiable.verifyThat(actual, Matches(expression));
  ```

  There exists more functionality when using the `Matches` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no

point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`matlab.unittest.constraints` | `matlab.unittest.constraints.Matches` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications` | `regexp` | `verifySubstring` | `verifyThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

verifyNotEmpty

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Verify value is not empty

Syntax

```
verifyNotEmpty(verifiable,actual)
verifyNotEmpty( ___ ,diagnostic)
```

Description

`verifyNotEmpty(verifiable,actual)` verifies that `actual` is a non-empty MATLAB value.

`verifyNotEmpty( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test for Non-Empty Character Vectors

Create a `TestCase` object.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

```
verifyNotEmpty(testCase, '');
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyNotEmpty failed.
--> The value must not be empty.
--> The value has a size of [0  0].

Actual char:
      0×0 empty char array
```

Test failed.

Test for Non-Empty Arrays

An array with any zero dimension is empty.

Test array `[2 3]`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyNotEmpty(testCase, [2 3]);
```

```
Interactive verification passed.
```

Test array with a zero dimension.

```
verifyNotEmpty(testCase, ones(2, 5, 0, 3));
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
verifyNotEmpty failed.
--> The value must not be empty.
--> The value has a size of [2  5  0  3].

Actual Value:
       Empty array: 2-by-5-by-0-by-3
```

Test failed.

Test for Non-Empty Cell Arrays

A cell array of empty arrays is not empty.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyNotEmpty(testCase, {[], [], []}, '');
```

```
Interactive verification passed.
```

Test for Non-Empty Test Suite

Test an empty object, `emptyTestSuite`.

```
emptyTestSuite = matlab.unittest.TestSuite.empty;
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyNotEmpty(testCase, emptyTestSuite);
```

```
Interactive verification failed.
```

```
----------------------
Framework Diagnostic:
----------------------
verifyNotEmpty failed.
--> The value must not be empty.
--> The value has a size of [0  0].

Actual Value:
      0x0 TestCase array with no properties.
```

Test failed.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEmpty;
  verifiable.verifyThat(actual, ~IsEmpty());
  ```

  There exists more functionality when using the `IsEmpty` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture

teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`isempty` | `matlab.unittest.constraints` | `matlab.unittest.constraints.IsEmpty` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications` | `verifyEmpty` | `verifyThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

verifyNotEqual

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Verify value is not equal to specified value

Syntax

```
verifyNotEqual(verifiable,actual,notExpected)
verifyNotEqual( ___ ,diagnostic)
```

Description

`verifyNotEqual(verifiable,actual,notExpected)` verifies that `actual` is not equal to `notExpected`.

`verifyNotEqual( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`notExpected`

Value to compare.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Compare Numeric Values

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Compare a value to itself.

```
verifyNotEqual(testCase, 5, 5);
```

```
Verification failed.
    ----------------------
    Framework Diagnostic:
    ----------------------
    verifyNotEqual failed.
    --> The numeric values are equal using "isequaln".

    Actual Value:
         5
    Prohibited Value:
         5
```

Test failed.

Compare different number values.

```
verifyNotEqual(testCase, 4.95, 5, '4.95 should be different from 5');
```

```
Verification passed.
```

Values `4.95` and `5` are not equal.

Compare values of different sizes.

```
verifyNotEqual(testCase, [5 5], 5, '[5 5] is not equal to 5');
```

```
Verification passed.
```

Values are not equal.

Compare Classes

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyNotEqual(testCase, int8(5), int16(5), 'Classes dont match');
```

```
Verification passed.
```

Compare Cell Arrays

Test a cell array by comparing each element.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyNotEqual(testCase, {'cell', struct, 5}, {'cell', struct, 5});
```

```
Verification failed.
    ----------------------
    Framework Diagnostic:
    ----------------------
    verifyNotEqual failed.

    Actual Value:
      1×3 cell array
        {'cell'}    {1×1 struct}    {[5]}
    Prohibited Value:
      1×3 cell array
        {'cell'}    {1×1 struct}    {[5]}
```

Test failed.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEqualTo;
  verifiable.verifyThat(actual, ~IsEqualTo(notExpected));
  ```

  There exists more functionality when using the `IsEqualTo` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`matlab.unittest.constraints` | `matlab.unittest.constraints.IsEqualTo` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications` | `verifyEqual` | `verifyThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

verifyNotSameHandle

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Verify value is not handle to specified instance

Syntax

```
verifyNotSameHandle(verifiable,actual,notExpectedHandle)
verifyNotSameHandle( ___ ,diagnostic)
```

Description

`verifyNotSameHandle(verifiable,actual,notExpectedHandle)` verifies that `actual` is a different size and/or does not contain the same instances as the `notExpectedHandle` handle array.

`verifyNotSameHandle( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`notExpectedHandle`

The handle array to compare.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test Handles from Same Class

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Create a handle class, `ExampleHandle`.

```
classdef ExampleHandle < handle
end
```

Create two handle variables.

```
h1 = ExampleHandle;
h2 = ExampleHandle;
```

Handles point to different objects.

```
verifyNotSameHandle(testCase, h1, h2);
```

```
Interactive verification passed.
```

Show matching handle combinations.

```
verifyNotSameHandle(testCase, [h1 h2 h1], [h1 h2 h1]);
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
verifyNotSameHandle failed.
--> The two handles must not refer to the same handle, or should have
different sizes.

Actual Value:
      1x3 ExampleHandle array with no properties.
Handle Object:
      1x3 ExampleHandle array with no properties.
```

Test failed.

The order of the handle arguments matters.

```
verifyNotSameHandle(testCase, [h1 h2], [h2 h1]);
```

```
Interactive verification passed.
```

Test a handle with itself.

```
verifyNotSameHandle(testCase, h1, h1);
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
verifyNotSameHandle failed.
--> The two handles must not refer to the same handle, or should have
different sizes.

Actual Value:
      ExampleHandle with no properties.
Handle Object:
      ExampleHandle with no properties.
```

Test failed.

Variables are not same size.

```
verifyNotSameHandle(testCase, h2, [h2 h2]);
```

```
Interactive verification passed.
```

Variables are the same size.

```
verifyNotSameHandle(testCase, [h1 h1], [h1 h1]);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyNotSameHandle failed.
--> The two handles must not refer to the same handle, or should have
different sizes.

Actual Value:
      1x2 ExampleHandle array with no properties.
Handle Object:
      1x2 ExampleHandle array with no properties.
```

Test failed.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsSameHandleAs;
  verifiable.verifyThat(actual, ~IsSameHandleAs(notExpectedHandle));
  ```

  There exists more functionality when using the `IsSameHandleAs` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.

- Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`matlab.unittest.constraints` | `matlab.unittest.constraints.IsSameHandleAs` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications` | `verifySameHandle` | `verifyThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

verifyNumElements

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Verify value has specified element count

Syntax

```
verifyNumElements(verifiable,actual,expectedElementCount)
verifyNumElements( ___ ,diagnostic)
```

Description

`verifyNumElements(verifiable,actual,expectedElementCount)` verifies that `actual` is a MATLAB array with `expectedElementCount` number of elements.

`verifyNumElements( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedElementCount`

The expected number of elements in the array.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test Matrices

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

```
n = 7;
verifyNumElements(testCase, eye(n), n^2);
```

```
Interactive verification passed.
```

```
verifyNumElements(testCase, 3, 1);
```

```
Interactive verification passed.
```

```
verifyNumElements(testCase, [1 2 3; 4 5 6], 5);
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
verifyNumElements failed.
--> The value did not have the correct number of elements.
```

```
    Actual Number of Elements:
             6
    Expected Number of Elements:
             5

Actual Value:
         1     2     3
         4     5     6
```

Test failed.

Test Cell Array

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyNumElements(testCase, {'SomeString', 'SomeOtherString'}, 2);
```

```
Interactive verification passed.
```

Test Structure

```
s.Field1 = 1;
s.Field2 = 2;
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyNumElements(testCase, s, 2);
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
verifyNumElements failed.
--> The value did not have the correct number of elements.

    Actual Number of Elements:
             1
    Expected Number of Elements:
             2

Actual Value:
        Field1: 1
        Field2: 2
```

Test failed.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasElementCount;
  verifiable.verifyThat(actual, HasElementCount(expectedElementCount));
  ```

  There exists more functionality when using the `HasElementCount` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`matlab.unittest.constraints` | `matlab.unittest.constraints.HasElementCount` |

`matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications` | `numel` | `verifyLength` | `verifySize` | `verifyThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

verifyReturnsTrue

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Verify function returns true when evaluated

Syntax

```
verifyReturnsTrue(verifiable,actual)
verifyReturnsTrue( ___ ,diagnostic)
```

Description

`verifyReturnsTrue(verifiable,actual)` verifies that `actual` is a function handle that returns a scalar logical whose value is true.

`verifyReturnsTrue( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The function handle to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test if Condition is True

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyReturnsTrue(testCase, @true);
```

```
Interactive verification passed.
```

Verify that it is true that two numbers are equal.

```
verifyReturnsTrue(testCase, @() isequal(1,1));
```

```
Interactive verification passed.
```

Verify that it is true that two letters are not the same.

```
verifyReturnsTrue(testCase, @() ~strcmp('a','b'));
```

```
Interactive verification passed.
```

Cause verification to fail by trying to verify that “false” evaluates to “true”.

```
verifyReturnsTrue(testCase, @false);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyReturnsTrue failed.
```

```
--> The function handle should have evaluated to "true".
--> Returned value:
             0

Actual Function Handle:
        @false
```

Test failed.

Cause verification to fail by having the test specified in the function handle return a vector of logical values not a scalar logical value.

```
verifyReturnsTrue(testCase, @() strcmp('a',{'a','a'}));
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyReturnsTrue failed.
--> The function handle should have returned a scalar. The return value had a size of
--> Returned value:
             1     1

Actual Function Handle:
        @()strcmp('a',{'a','a'})
```

Test failed.

Cause verification to fail by having the test specified in the function handle return a double not a logical.

```
verifyReturnsTrue(testCase, @() exist('exist'));
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyReturnsTrue failed.
--> The function handle should have returned a logical value. It was of type "double".
--> Returned value:
             5
```

```
Actual Function Handle:
        @()exist('exist')
```

Test failed.

Tips

- It is a shortcut for quick custom comparison functionality that can be defined quickly, and possibly inline. It can be preferable over simply evaluating the function directly and using `verifyTrue` because the function handle will be shown in the diagnostics, thus providing more insight into the failure condition which is lost when using `verifyTrue`.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.ReturnsTrue;
  verifiable.verifyThat(actual, ReturnsTrue());
  ```

  There exists more functionality when using the `ReturnsTrue` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort

testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`matlab.unittest.constraints` | `matlab.unittest.constraints.ReturnsTrue` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications` | `verifyThat` | `verifyTrue`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

verifySameHandle

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Verify two values are handles to same instance

Syntax

```
verifySameHandle(verifiable,actual,expectedHandle)
verifySameHandle( ___ ,diagnostic)
```

Description

`verifySameHandle(verifiable,actual,expectedHandle)` verifies that `actual` is the same size and contains the same instances as the `expectedHandle` handle array.

`verifySameHandle( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedHandle`

The expected handle array.

diagnostic

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test Handles from Same Class

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Create a handle class, `ExampleHandle`.

```
classdef ExampleHandle < handle
end
```

Create two handle variables.

```
h1 = ExampleHandle;
h2 = ExampleHandle;
```

Show matching handle combinations.

```
verifySameHandle(testCase, h1, h1);
```

```
Interactive verification passed.
```

```
verifySameHandle(testCase, [h1 h1], [h1 h1]);
```

```
Interactive verification passed.
```

```
verifySameHandle(testCase, [h1 h2 h1], [h1 h2 h1]);
```

```
Interactive verification passed.
```

Handles must point to same object.

```
verifySameHandle(testCase, h1, h2);
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
verifySameHandle failed.
--> Values do not refer to the same handle.

Actual Value:
      ExampleHandle with no properties.
Expected Handle Object:
      ExampleHandle with no properties.
```

Test failed.

Size of handle objects must match.

```
verifySameHandle(testCase, [h1 h1], h1);
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
verifySameHandle failed.
--> Sizes do not match.
        Actual Value Size           : [1  2]
        Expected Handle Object Size : [1  1]

Actual Value:
      1x2 ExampleHandle array with no properties.
Expected Handle Object:
      ExampleHandle with no properties.
```

Test failed.

Order of arguments is important.

```
verifySameHandle(testCase, [h1 h2], [h2 h1]);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifySameHandle failed.
--> Some elements in the handle array refer to the wrong handle.

Actual Value:
      1x2 ExampleHandle array with no properties.
Expected Handle Object:
      1x2 ExampleHandle array with no properties.
```

Test failed.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsSameHandleAs;
  verifiable.verifyThat(actual, IsSameHandleAs(expectedHandle));
  ```

  There exists more functionality when using the `IsSameHandleAs` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`handle` | `matlab.unittest.constraints` | `matlab.unittest.constraints.IsSameHandleAs` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications` | `verifyNotSameHandle` | `verifyThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

verifySize

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Verify value has specified size

Syntax

```
verifySize(verifiable,actual,expectedSize)
verifySize( ___ ,diagnostic)
```

Description

`verifySize(verifiable,actual,expectedSize)` verifies that `actual` is a MATLAB array whose size is `expectedSize`.

`verifySize( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`expectedSize`

The expected sizes of each dimension the array.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test Arrays

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

```
verifySize(testCase, ones(2, 5, 3), [2 5 3]);
```

```
Interactive verification passed.
```

```
verifySize(testCase, [1 2 3; 4 5 6], [2 3]);
```

```
Interactive verification passed.
```

```
verifySize(testCase, [2 3], [3 2]);
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
verifySize failed.
--> The value had an incorrect size.

    Actual Size:
```

```
             1     2
    Expected Size:
             3     2

Actual Value:
         2     3
```

Test failed.

Number of elements is not the same as size.

```
verifySize(testCase, [1 2 3; 4 5 6], [6 1]);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifySize failed.
--> The value had an incorrect size.

    Actual Size:
             2     3
    Expected Size:
             6     1

Actual Value:
         1     2     3
         4     5     6
```

Test failed.

```
verifySize(testCase, eye(2), [4 1]);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifySize failed.
--> The value had an incorrect size.

    Actual Size:
             2     2
    Expected Size:
```

```
             4     1

Actual Value:
         1     0
         0     1
```

Test failed.

Test Cell Array

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifySize(testCase, {'SomeText', 'SomeOtherText'}, [1 2]);
```

```
Interactive verification passed.
```

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasSize;
  verifiable.verifyThat(actual, HasSize(expectedSize));
  ```

  There exists more functionality when using the `HasSize` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`matlab.unittest.constraints` | `matlab.unittest.constraints.HasSize` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications` | `size` | `verifyLength` | `verifyNumElements` | `verifyThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

verifySubstring

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Verify string contains specified string

Syntax

```
verifySubstring(verifiable,actual,substring)
verifySubstring( ___ ,diagnostic)
```

Description

`verifySubstring(verifiable,actual,substring)` verifies that `actual` is a string scalar or character vector that contains `substring`.

`verifySubstring( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The string to test.

Default:

`substring`

The value to match, specified as a string scalar or character vector.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test for Substrings

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Test that a substring is contained in a string.

```
verifySubstring(testCase, 'SomeLongMessage', 'Long');
```

```
Interactive verification passed.
```

Show that case matters.

```
verifySubstring(testCase, 'SomeLongMessage', 'lonG');
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifySubstring failed.
--> The value does not contain the substring.

Actual char:
```

```
    SomeLongMessage
Expected Substring:
    lonG
```

Test failed.

Cause the verification to fail by testing a substring that isn’t contained in the actual string.

```
verifySubstring(testCase, 'SomeLongMessage', 'OtherMessage');
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifySubstring failed.
--> The value does not contain the substring.

Actual char:
    SomeLongMessage
Expected Substring:
    OtherMessage
```

Test failed.

Show that the verification will fail if the substring is longer than the actual string.

```
verifySubstring(testCase, 'SomeLongMessage', 'SomeLongMessageThatIsLonger');
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifySubstring failed.
--> The value does not contain the substring.

Actual char:
    SomeLongMessage
Expected Substring:
    SomeLongMessageThatIsLonger
```

Test failed.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.ContainsSubstring;
  verifiable.verifyThat(actual, ContainsSubstring(substring));
  ```

  There exists more functionality when using the `ContainsSubstring` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`matlab.unittest.constraints` | `matlab.unittest.constraints.ContainsSubstring` |

`matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications` | `strfind` | `verifyMatches` | `verifyThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

verifyThat

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Verify value meets given constraint

Syntax

```
verifyThat(verifiable,actual,constraint)
verifyThat( ___ ,diagnostic)
```

Description

`verifyThat(verifiable,actual,constraint)` verifies that `actual` is a value that satisfies the `constraint` provided.

If the constraint is not satisfied, a verification failure is produced utilizing only the framework diagnostic generated by the `constraint`.

`verifyThat( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

When using this signature, both the diagnostic information contained within `diagnostic` is used in addition to the diagnostic information provided by the `constraint`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`constraint`

Constraint that the actual value must satisfy to pass the verification, specified as a `matlab.unittest.constraints` instance.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test Conditions Using Constraints

```
testCase = matlab.unittest.TestCase.forInteractiveUse;

% Passing scenarios
%%%%%%%%%%%%%%%%%%%%%
import matlab.unittest.constraints.IsTrue
verifyThat(testCase, true, IsTrue)

import matlab.unittest.constraints.IsEqualTo
verifyThat(testCase, 5, IsEqualTo(5), '5 should be equal to 5')

import matlab.unittest.constraints.IsGreaterThan
import matlab.unittest.constraints.HasNaN
verifyThat(testCase, [5 NaN], IsGreaterThan(10) | HasNaN, ...
    'The value was not greater than 10 or NaN')
```

```
% Failing scenarios
%%%%%%%%%%%%%%%%%%%%%%
import matlab.unittest.constraints.AnyCellOf
import matlab.unittest.constraints.ContainsSubstring
verifyThat(testCase, AnyCellOf({'cell','of','strings'}), ...
    ContainsSubstring('char'),'Test description')

import matlab.unittest.constraints.HasSize
verifyThat(testCase, zeros(10,4,2), HasSize([10,5,2]), ...
    @() disp('A function handle diagnostic.'))

import matlab.unittest.constraints.IsEmpty
verifyThat(testCase, 5, IsEmpty)
```

Tips

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`matlab.unittest.constraints` | `matlab.unittest.constraints.Constraint` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

verifyTrue

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Verify value is true

Syntax

```
verifyTrue(verifiable,actual)
verifyTrue( ___ ,diagnostic)
```

Description

`verifyTrue(verifiable,actual)` verifies that `actual` is a scalar logical with the value of true.

`verifyTrue( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The value to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Examples

Test MATLAB Logical Functions

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Test `true`.

```
verifyTrue(testCase, true);
```

```
Interactive verification passed.
```

Test `false`.

```
verifyTrue(testCase, false);
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
verifyTrue failed.
--> The value must evaluate to "true".

Actual Value:
         0
```

Test failed.

Test the Value 1

The number `1` is a `double` value, not a logical value.

A double value of `1` is not `true`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyTrue(testCase, 1);
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
verifyTrue failed.
--> The value must be logical. It is of type "double".

Actual Value:
         1
```

Test failed.

Test Array of Logical Values

To be true, the value must be scalar.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyTrue(testCase, [true true true]);
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
verifyTrue failed.
--> The value must be scalar. It has a size of [1  3].

Actual Value:
         1     1     1
```

Test failed.

Tips

- This method passes if and only if the actual value is a scalar logical with a value of true. Therefore, entities such as true valued arrays and nonzero doubles produce qualification failures when used in this method, despite these entities exhibiting "true-like" behavior such as triggering the execution of code inside of "if" statements.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsTrue;
  verifiable.verifyThat(actual, IsTrue());
  ```

  There exists more functionality when using the `IsTrue` constraint directly via `verifyThat`.

  Use of this method for performance benefits can come at the expense of less diagnostic information, and may not provide the same level of strictness adhered to by other constraints such as `IsEqualTo`. A similar approach that is generally less performant but can provide slightly better diagnostic information is the use of `verifyReturnsTrue`, which at least shows the display of the function evaluated to generate the failing result.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`matlab.unittest.constraints` | `matlab.unittest.constraints.IsTrue` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications` | `verifyFalse` | `verifyReturnsTrue` | `verifyThat`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

verifyWarning

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Verify function issues specified warnings

Syntax

```
verifyWarning(verifiable,actual,warningIDs)
verifyWarning(verifiable,actual,warningIDs,diagnostic)
[output1,...,outputN] = verifyWarning( ___ )
```

Description

`verifyWarning(verifiable,actual,warningIDs)` verifies that `actual` issues warnings with the identifiers `warningIDs`.

`verifyWarning(verifiable,actual,warningIDs,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

`[output1,...,outputN] = verifyWarning( ___ )` also returns the output arguments `output1,...,outputN` that are produced when invoking `actual`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The function handle to test.

Default:

warningIDs

Warning identifiers, specified as a character vector, cell array of character vectors, or string array.

diagnostic

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Output Arguments

output1,...,outputN

Output arguments, 1 through n (if any), from `actual`, returned as any type. The argument type is specified by the `actual` argument list.

Examples

Test warning Function

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Verify that actual warning identifier is the same as input warning identifier.

```
verifyWarning(testCase, @() warning('SOME:warning:id', 'Warning!'), ...
    'SOME:warning:id');
```

```
Verification passed.
```

```
verifyWarning(testCase, @() warning('SOME:other:id', 'Warning message'), ...
    'SOME:warning:id', 'Did not issue specified warning');
```

```
Warning: Warning message
> In @()warning('SOME:other:id','Warning message')
  In matlab.unittest.internal.constraints/FunctionHandleConstraint/invoke (line 35)
  In matlab.unittest.internal.constraints/WarningQualificationConstraint/invoke (line 41)
  In matlab.unittest.constraints/IssuesWarnings/invoke (line 429)
  In matlab.unittest.constraints.IssuesWarnings>@()constraint.invoke(fcn) (line 540)
  In matlab.unittest.internal.fevalcRespectingHotlinks/hotwrap (line 18)
  In matlab.unittest.internal.fevalcRespectingHotlinks (line 14)
  In matlab.unittest.constraints/IssuesWarnings/invokeCapturingOutput (line 540)
  In matlab.unittest.constraints/IssuesWarnings/issuesExpectedWarnings (line 549)
  In matlab.unittest.constraints/IssuesWarnings/satisfiedBy (line 239)
  In matlab.unittest.internal.constraints/CasualDiagnosticDecorator/satisfiedBy (line 42)
  In matlab.unittest.internal.qualifications/QualificationDelegate/qualifyThat (line 79)
  In matlab.unittest.internal.qualifications/QualificationDelegate/qualifyWarning (line 200)
  In matlab.unittest.qualifications/Verifiable/verifyWarning (line 757)
Verification failed.
    ----------------
    Test Diagnostic:
    ----------------
    Did not issue specified warning
    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyWarning failed.
    --> The function handle did not issue the expected warning(s).

        Actual Warning(s):
            --> 'SOME:other:id'
        Expected Warning(s):
            --> 'SOME:warning:id'

    Evaluated Function:
      function_handle with value:

        @()warning('SOME:other:id','Warning message')
```

Test a Function Without Warnings

Test the `true` function, which does not issue warnings.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
verifyWarning(testCase, @true, 'SOME:warning:id', ...
'@true did not issue any warning');
```

```
Verification failed.
    ----------------
    Test Diagnostic:
    ----------------
    @true did not issue any warning
    ---------------------
    Framework Diagnostic:
```

```
    ----------------------
    verifyWarning failed.
    --> The function handle did not issue any warnings.

        Expected Warning(s):
            --> 'SOME:warning:id'

    Evaluated Function:
      function_handle with value:

        @true
```

Test failed.

Test Function With Output Arguments

Create a `helper` function that generates a warning and returns output.

```
function varargout = helper()
    warning('SOME:warning:id', 'Warning!');
    varargout = {123, 'abc'};
end
```

Call `helper`.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
[actualOut1, actualOut2] = verifyWarning(testCase, @helper, ...
    'SOME:warning:id');
```

```
Verification passed.
```

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IssuesWarnings;
  verifiable.verifyThat(actual,IssuesWarnings(cellstr(warningIDs)));
  ```

  There exists more functionality when using the `IssuesWarnings` constraint directly via `verifyThat`.

- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,
  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.
  - Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`matlab.unittest.constraints` | `matlab.unittest.constraints.IssuesWarnings` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications` | `verifyError` | `verifyThat` | `verifyWarningFree` | `warning`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

verifyWarningFree

Class: `matlab.unittest.qualifications.Verifiable`
Package: `matlab.unittest.qualifications`

Verify function issues no warnings

Syntax

```
verifyWarningFree(verifiable,actual)
verifyWarningFree( ___ ,diagnostic)
output1,...,outputN = verifyWarningFree( ___ )
```

Description

`verifyWarningFree(verifiable,actual)` verifies that `actual` is a function handle that issues no warnings.

`verifyWarningFree( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

`output1,...,outputN = verifyWarningFree( ___ )` also returns the output arguments `output1,...,outputN` that are produced when invoking `actual`.

Input Arguments

`verifiable`

The `matlab.unittest.TestCase` instance which is used to pass or fail the verification in conjunction with the test running framework.

`actual`

The function handle to test.

Default:

`diagnostic`

Diagnostic information related to the qualification, specified as one of the following:

- string array
- character array
- function handle
- `matlab.unittest.diagnostics.Diagnostic` object

Diagnostic values can be nonscalar. For more information, see `matlab.unittest.diagnostics.Diagnostic`.

Output Arguments

`output1,...,outputN`

Output arguments, 1 through n (if any), from `actual`, returned as any type. The argument type is specified by the `actual` argument list.

Examples

Test for Warnings from MATLAB Functions

Create a `TestCase` object for interactive testing.

```
testCase = matlab.unittest.TestCase.forInteractiveUse;
```

Test the `why` function.

```
verifyWarningFree(testCase, @why);
```

```
The bald and not excessively bald and not excessively smart hamster obeyed a terrified
Interactive verification passed.
```

This is a randomly-generated message.

Test the `true` function.

```
verifyWarningFree(testCase, @true);
```

```
Interactive verification passed.
```

Test the `false` function.

```
actualOutputFromFalse = verifyWarningFree(testCase, @false);
```

```
Interactive verification passed.
```

Test a value that is not a function handle.

```
verifyWarningFree(testCase, 5,'diagnostic');
```

```
Interactive verification failed.

----------------
Test Diagnostic:
----------------
diagnostic

---------------------
Framework Diagnostic:
---------------------
verifyWarningFree failed.
--> The value must be an instance of the expected type.

    Actual Class:
        double
    Expected Type:
        function_handle

Actual Value:
         5
```

Test failed.

Test a function that generates warning.

```
verifyWarningFree(testCase, @() warning('some:id', 'Message'));
```

```
Warning: Message
> In @()warning('some:id','Message')
  In FunctionHandleConstraint>FunctionHandleConstraint.invoke at 43
  In WarningQualificationConstraint>WarningQualificationConstraint.invoke at 58
  In IssuesNoWarnings>IssuesNoWarnings.issuesNoWarnings at 131
  In IssuesNoWarnings>IssuesNoWarnings.satisfiedBy at 82
  In QualificationDelegate>QualificationDelegate.qualifyThat at 90
```

```
  In QualificationDelegate>QualificationDelegate.qualifyWarningFree at 204
  In Verifiable>Verifiable.verifyWarningFree at 757
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyWarningFree failed.
--> The function issued warnings.

    Warnings Issued:
            some:id

Evaluated Function:
        @()warning('some:id','Message')
```

Test failed.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IssuesNoWarnings;
  verifiable.verifyThat(actual, IssuesNoWarnings());
  ```

  There exists more functionality when using the `IssuesNoWarnings` constraint directly via `verifyThat`.
- Use verification qualifications to produce and record failures without throwing an exception. Since verifications do not throw exceptions, all test content runs to completion even when verification failures occur. Typically verifications are the primary qualification for a unit test since they typically do not require an early exit from the test. Use other qualification types to test for violation of preconditions or incorrect test setup. Alternatively,

  - Use assumption qualifications to ensure that the test environment meets preconditions that otherwise do not result in a test failure. Assumption failures result in filtered tests, and the testing framework marks the tests as `Incomplete`. For more information, see `matlab.unittest.qualifications.Assumable`.
  - Use assertion qualifications when the failure condition invalidates the remainder of the current test content, but does not prevent proper execution of subsequent test methods. A failure at the assertion point renders the current test method as failed

and incomplete. For more information, see `matlab.unittest.qualifications.Assertable`.

- Use fatal assertion qualifications to abort the test session upon failure. These qualifications are useful when the failure mode is so fundamental that there is no point in continuing testing. These qualifications are also useful when fixture teardown does not restore the MATLAB state correctly and it is preferable to abort testing and start a fresh session. For more information, see `matlab.unittest.qualifications.FatalAssertable`.

See Also

`matlab.unittest.constraints` | `matlab.unittest.constraints.IssuesNoWarnings` | `matlab.unittest.diagnostics.Diagnostic` | `matlab.unittest.qualifications` | `verifyThat` | `verifyWarning` | `warning`

Topics

"Table of Verifications, Assertions, and Other Qualifications"

Introduced in R2013a

quarter

Quarter number

Syntax

```
q = quarter(t)
```

Description

`q = quarter(t)` returns the quarter numbers for the datetime values in `t`. The `q` output is a `double` array containing integer values from 1 to 4, and is the same size as `t`.

Examples

Find Quarter Number of Dates

```
t = datetime(2013,05,31):calmonths(3):datetime(2014,05,31)
```

```
t = 1x5 datetime array
   31-May-2013   31-Aug-2013   30-Nov-2013   28-Feb-2014   31-May-2014
```

```
q = quarter(t)
```

```
q = 1×5

     2     3     4     1     2
```

Input Arguments

t — Input date and time
`datetime` array

Input date and time, specified as a `datetime` array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`day` | `month` | `week` | `year`

Introduced in R2014b

questdlg

Create question dialog box

Note If you are using App Designer or creating apps with the `uifigure` function, then use `uiconfirm` instead. For more information, see “GUIDE Migration Strategies”.

Syntax

```
answer = questdlg(quest)
answer = questdlg(quest,dlgtitle)
answer = questdlg(quest,dlgtitle,defbtn)
answer = questdlg(quest,dlgtitle,btn1,btn2,defbtn)
answer = questdlg(quest,dlgtitle,btn1,btn2,btn3,defbtn)

answer = questdlg(quest,dlgtitle,opts)
answer = questdlg(quest,dlgtitle,btn1,btn2,opts)
answer = questdlg(quest,dlgtitle,btn1,btn2,btn3,opts)
```

Description

`answer = questdlg(quest)` creates a modal on page 1-16280 dialog box that presents a question and returns the user's response -- `'Yes'`, `'No'`, `'Cancel'`, or `''`.

By default, the dialog box has three standard buttons, labeled **Yes**, **No**, and **Cancel**.

- If the user clicks one of these buttons, then the `answer` value is the same as the label of the pressed button.
- If the user clicks the close button (X) on the dialog box title bar or presses the **Esc** key, then the `answer` value is an empty character vector (`''`).
- If the user presses the **Return** key, then the `answer` value is the same as the label of the default button selection. In this case, `'Yes'`.

`answer = questdlg(quest,dlgtitle)` specifies a dialog box title.

`answer = questdlg(quest,dlgtitle,defbtn)` specifies which button is the default if the user presses the keyboard **Return** key. The `defbtn` value must match one of the button labels.

`answer = questdlg(quest,dlgtitle,btn1,btn2,defbtn)` customizes two of the standard buttons by labeling them with the values of `btn1` and `btn2`. The third standard button is removed. The `defbtn` value must match the value of `btn1` or `btn2`.

If the user presses the keyboard **Return** key, and the `defbtn` value does not match one of the button labels, then the dialog box remains open.

`answer = questdlg(quest,dlgtitle,btn1,btn2,btn3,defbtn)` customizes the third standard button with a label that matches the value of `btn3`.

`answer = questdlg(quest,dlgtitle,opts)` specifies an options structure to specify the default button selection and whether to use TeX to interpret the question text.

`answer = questdlg(quest,dlgtitle,btn1,btn2,opts)` customizes two standard buttons with labels that match the values of `btn1` and `btn2`. The third standard button is removed.

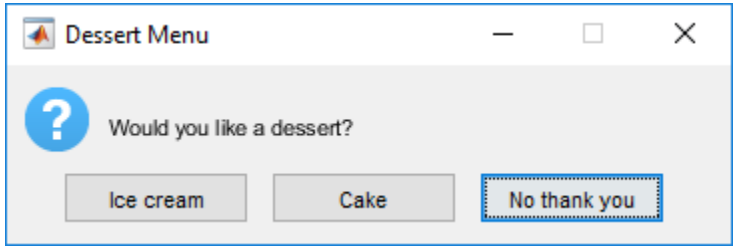
`answer = questdlg(quest,dlgtitle,btn1,btn2,btn3,opts)` customizes the third standard button with a label that matches the value of `btn3`.

Examples

Encode User's Choice as an Integer in Question Dialog Box

```
answer = questdlg('Would you like a dessert?', ...
    'Dessert Menu', ...
    'Ice cream','Cake','No thank you','No thank you');
% Handle response
switch answer
    case 'Ice cream'
        disp([answer ' coming right up.'])
        dessert = 1;
    case 'Cake'
        disp([answer ' coming right up.'])
        dessert = 2;
    case 'No thank you'
        disp('I''ll bring you your check.')
```

```
        dessert = 0;
end
```

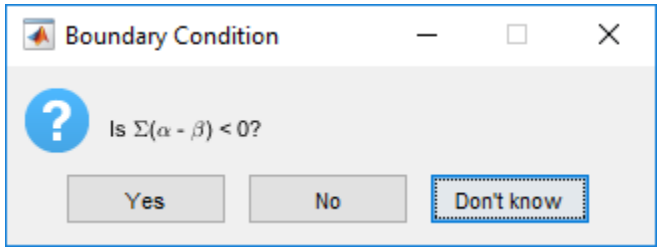


To access the return value assigned to `dessert`, save the example as a function. For example, create function `choosedessert` by making this the first line of code.

```
function dessert = choosedessert
```

Format Question in Dialog Box Using TeX

```
opts.Interpreter = 'tex';
% Include the desired Default answer
opts.Default = 'Don''t know';
% Use the TeX interpreter to format the question
quest = 'Is \Sigma(\alpha - \beta) < 0?';
answer = questdlg(quest,'Boundary Condition',...
                  'Yes','No','Don''t know',opts)
```



Input Arguments

quest — Dialog box question

character vector | cell array of character vectors | string array

Dialog box question, specified as a character vector, cell array of character vectors, or string array. The question automatically wraps to fit within the dialog box.

Example: `'What is the velocity?'`

`dlgtitle` — Dialog box title
`''` (default) | character vector | string scalar

Dialog box title, specified as a character vector or string scalar.

Example: `'Configuration'`

`defbtn` — Default button selection
character vector | string scalar

The default button selection, specified as a character vector or string scalar. The default button selection is the value that MATLAB returns if the user presses the keyboard **Return** key instead of clicking a button in the dialog box. The default selection must be the same as one of the dialog box button labels. If the `defbtn` argument value does not match a button label, then the dialog box remains open if the user presses the keyboard **Enter** key.

Example: `'Cancel'`

`btn1` — First customized button label
character vector | string scalar

First customized button label, specified as a character vector or string scalar.

Example: `'Start'`

`btn2` — Second customized button label
character vector | string scalar

Second customized button label, specified as a character vector or string scalar.

Example: `'Reset'`

`btn3` — Third customized button label
character vector | string scalar

Third customized button label, specified as a character vector or string scalar.

Example: `'Test'`

`opts` — Dialog box settings

structure

Dialog box settings specified as a structure. The structure specifies which button is the default button selection, and whether to use TeX to interpret the question text.

The `opts` structure, must include both of the fields presented in this table. The structure can include additional fields, but `questdlg` does not use them.

Field	Values
`Default`	The default button selection, specified as a character vector or string scalar. The default selection must be the same as one of the dialog box button labels. The default button is the button value that MATLAB returns if the user presses the keyboard **Return** key instead of clicking a dialog box button. If the `Default` field value does not match one of the button labels, then the dialog box does not respond to the user pressing the keyboard **Enter** key.
`Interpreter`	`'none'` (default) or `'tex'`. If set to `'tex'`, then the prompt is rendered using TeX. Button labels and the dialog box title are not affected. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the `question` text.

Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they modify only the next character or the characters within the curly braces. When you set the `Interpreter` property to `'tex'`, the supported modifiers are as follows.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`

Modifier	Description	Example
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{`*`specifier`*`}`	Font name — Replace *`specifier`* with the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{`*`specifier`*`}`	Font size —Replace *`specifier`* with a numeric scalar value in point units.	`'\fontsize{15} text'`
`\color{`*`specifier`*`}`	Font color — Replace *`specifier`* with one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Replace *`specifier`* with a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	β	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\epsilon	ϵ	\Delta	Δ	\spadesuit	♠
\zeta	ζ	\Theta	Θ	\leftrightarrow	↔
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	μ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

Example: `opts.Default = 'Cancel'`

Example: `opts.Interpreter = 'tex'`

More About

Modal Dialog Box

A modal dialog box prevents the user from interacting with other MATLAB windows before responding.

See Also

`inputdlg` | `listdlg`

Introduced before R2006a

quit

Terminate MATLAB program

Syntax

```
quit
quit cancel
quit force
quit(code)
quit(code,"force")
```

Description

`quit` terminates the MATLAB program. The `quit` function does not automatically save the workspace.

To interrupt a MATLAB command, see “Stop Execution”.

`quit cancel` is for use in a `finish.m` script and cancels quitting. It has no effect anywhere else.

`quit force` bypasses `finish.m` and terminates MATLAB. Use this syntax to override the `finish` script if the script does not let you quit.

`quit(code)` returns the specified value as the MATLAB exit code.

`quit(code,"force")` bypasses `finish.m` and terminates MATLAB with the exit code.

Example: `quit(0,"force")`

Input Arguments

code — Exit code
0 (default) | signed integer

Exit code, specified as a signed integer. On Windowsplatforms, MATLAB returns values in the range of `INT_MIN` to `INT_MAX` (-2147483647 to 2147483647). On Linux and macOS platforms, MATLAB returns values between 0 and 255.

The exit code is returned by default on macOS and Linux. On Windows, start MATLAB with the `-wait` option.

Example: `quit(1)`

Limitations

- MATLAB Online does not support displaying a confirmation dialog box or using a `finish.m` script. Before logging out, MATLAB Online saves open files and session data.

Alternatives

Click the close button on the MATLAB desktop.

See Also

`exit` | `finish` | `save`

Topics

"Exit MATLAB"

Introduced before R2006a

Quit (COM)

Terminate MATLAB Automation server

Syntax

IDL Method Signature

```
void Quit(void)
```

Microsoft Visual Basic Client

```
Quit
```

MATLAB Client

```
Quit(h)
```

Description

`Quit(h)` terminates the MATLAB server session attached to handle `h`. The MATLAB object is active until all references have been released, such as when the variable in a function call goes out of scope, or by calling the MATLAB `clear h` command.

The function name is case-sensitive.

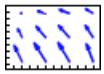
To release the MATLAB object, type:

```
clear h
```

Introduced before R2006a

quiver

Quiver or velocity plot



Syntax

```
quiver(x,y,u,v)
quiver(u,v)
quiver(...,scale)
quiver(...,LineSpec)
quiver(...,LineSpec,'filled')
quiver(...,'PropertyName',PropertyValue,...)
quiver(ax,...)
h = quiver(...)
```

Description

A quiver plot displays velocity vectors as arrows with components `(u,v)` at the points `(x,y)`.

For example, the first vector is defined by components `u(1)`,`v(1)` and is displayed at the point `x(1)`,`y(1)`.

`quiver(x,y,u,v)` plots vectors as arrows at the coordinates specified in each corresponding pair of elements in `x` and `y`. The matrices `x`, `y`, `u`, and `v` must all be the same size and contain corresponding position and velocity components. However, `x` and `y` can also be vectors, as explained in the next section. By default, the arrows are scaled to just not overlap, but you can scale them to be longer or shorter if you want.

`quiver(u,v)` draws vectors specified by `u` and `v` at equally spaced points in the *x*-*y* plane.

`quiver(...,scale)` automatically scales the arrows to fit within the grid and then stretches them by the factor `scale`. `scale = 2` doubles their relative length, and `scale = 0.5` halves the length. Use `scale = 0` to plot the velocity vectors without automatic scaling. You can also tune the length of arrows after they have been drawn by choosing the Plot Edit tool, selecting the quiver object, opening the Property Editor, and adjusting the **Length** slider.

`quiver(...,LineSpec)` specifies line style, marker symbol, and color using any valid `LineSpec`. `quiver` draws the markers at the origin of the vectors.

`quiver(...,LineSpec,'filled')` fills markers specified by `LineSpec`.

`quiver(...,'`*`PropertyName`*`',PropertyValue,...)` specifies property name and property value pairs for the quiver objects the function creates.

`quiver(ax,...)` plots into the axes `ax` instead of into the current axes (`gca`).

`h = quiver(...)` returns the `Quiver` object.

Expanding x- and y-Coordinates

MATLAB expands `x` and `y` if they are not matrices. This expansion is equivalent to calling `meshgrid` to generate matrices from vectors:

```
[x,y] = meshgrid(x,y);
quiver(x,y,u,v)
```

In this case, the following must be true:

`length(x) = n` and `length(y) = m`, where `[m,n] = size(u) = size(v)`.

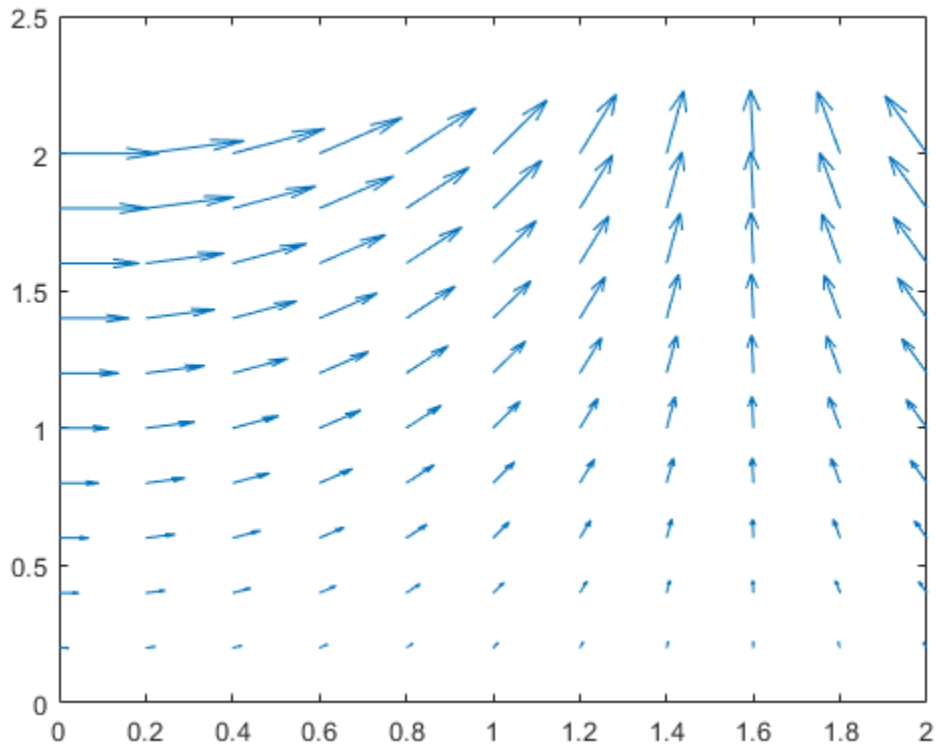
The vector `x` corresponds to the columns of `u` and `v`, and vector `y` corresponds to the rows of `u` and `v`.

Examples

Plot Vector Velocities

Use `quiver` to display an arrow at each data point in `x` and `y` such that the arrow direction and length represent the corresponding values in `u` and `v`.

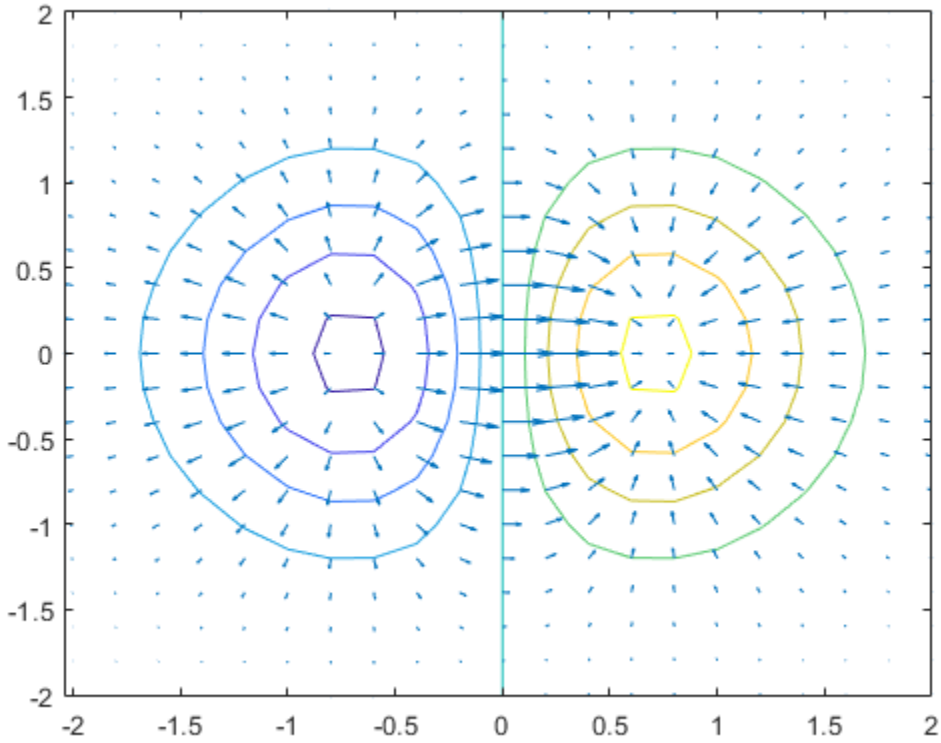
```
[x,y] = meshgrid(0:0.2:2,0:0.2:2);
u = cos(x).*y;
v = sin(x).*y;

figure
quiver(x,y,u,v)
```



Show Gradient with Quiver Plot

Plot the gradient of the function $z = xe^{-x^2 - y^2}$.

```
[X,Y] = meshgrid(-2:.2:2);
Z = X.*exp(-X.^2 - Y.^2);
[DX,DY] = gradient(Z,.2,.2);

figure
contour(X,Y,Z)
hold on
quiver(X,Y,DX,DY)
hold off
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions

`LineSpec` | `contour` | `plot` | `quiver3`

Properties

Quiver

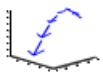
Topics

“Combine Contour Plot and Quiver Plot”

Introduced before R2006a

quiver3

3-D quiver or velocity plot



Syntax

```
quiver3(x,y,z,u,v,w)
quiver3(z,u,v,w)
quiver3(...,scale)
quiver3(...,LineSpec)
quiver3(...,LineSpec,'filled')
quiver3(...,'PropertyName',PropertyValue,...)
quiver3(ax,...)
h = quiver3(...)
```

Description

A three-dimensional quiver plot displays vectors with components `(u,v,w)` at the points `(x,y,z)`, where `u`, `v`, `w`, `x`, `y`, and `z` all have real (non-complex) values.

`quiver3(x,y,z,u,v,w)` plots vectors with directions determined by components `(u,v,w)` at points determined by `(x,y,z)`. The matrices `x`,`y`,`z`,`u`,`v`, and `w` must all be the same size and contain the corresponding position and vector components.

`quiver3(z,u,v,w)` plots vectors with directions determined by components `(u,v,w)` at equally spaced points along the surface `z`. For each vector `(u(i,j),v(i,j),w(i,j))`, the column index `j` determines the *x*-value of the point on the surface, the row index `i` determines the *y*-value, and `z(i,j)` determines the *z*-value. That is, `quiver3` locates the vector at the point on the surface `(j,i,z(i,j))`. The `quiver3` function automatically scales the vectors to prevent overlapping based on the distance between them.

`quiver3(...,scale)` automatically scales the vectors to prevent them from overlapping, and then multiplies them by `scale`. `scale = 2` doubles their relative length, and `scale = 0.5` halves them. Use `scale = 0` to plot the vectors without the automatic scaling.

`quiver3(...,LineSpec)` specifies line style, marker symbol, and color using any valid `LineSpec`. `quiver3` draws the markers at the origin of the vectors.

`quiver3(...,LineSpec,'filled')` fills markers specified by `LineSpec`.

`quiver3(...,'PropertyName',PropertyValue,...)` specifies property name and property value pairs for the quiver chart that the function creates.

`quiver3(ax,...)` plots into the axes `ax` instead of into the current axes (`gca`).

`h = quiver3(...)` returns the `Quiver` object.

Examples

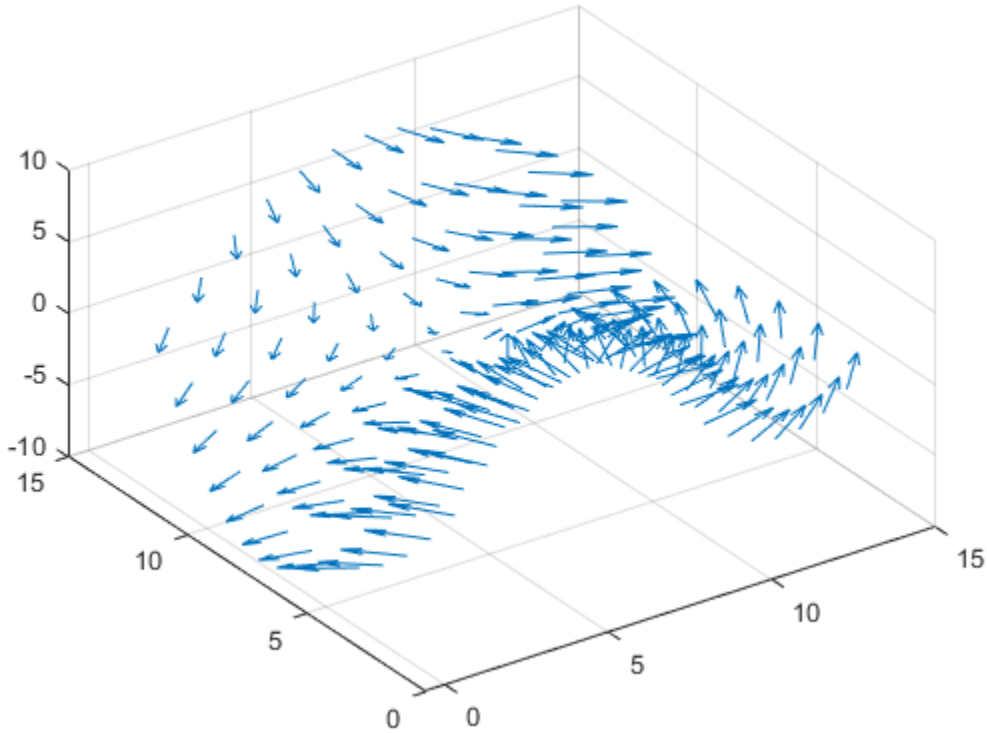
Create 3-D Quiver Plot

Define the data.

```
x = -3:0.5:3;
y = -3:0.5:3;
[X,Y] = meshgrid(x, y);
Z = Y.^2 - X.^2;
[U,V,W] = surfnorm(Z);
```

Plot vectors with components `(U,V,W)` at points that are equally spaced in the *x*-direction and *y*-direction with heights determined by `Z`.

```
figure
quiver3(Z,U,V,W)
view(-35,45)
```

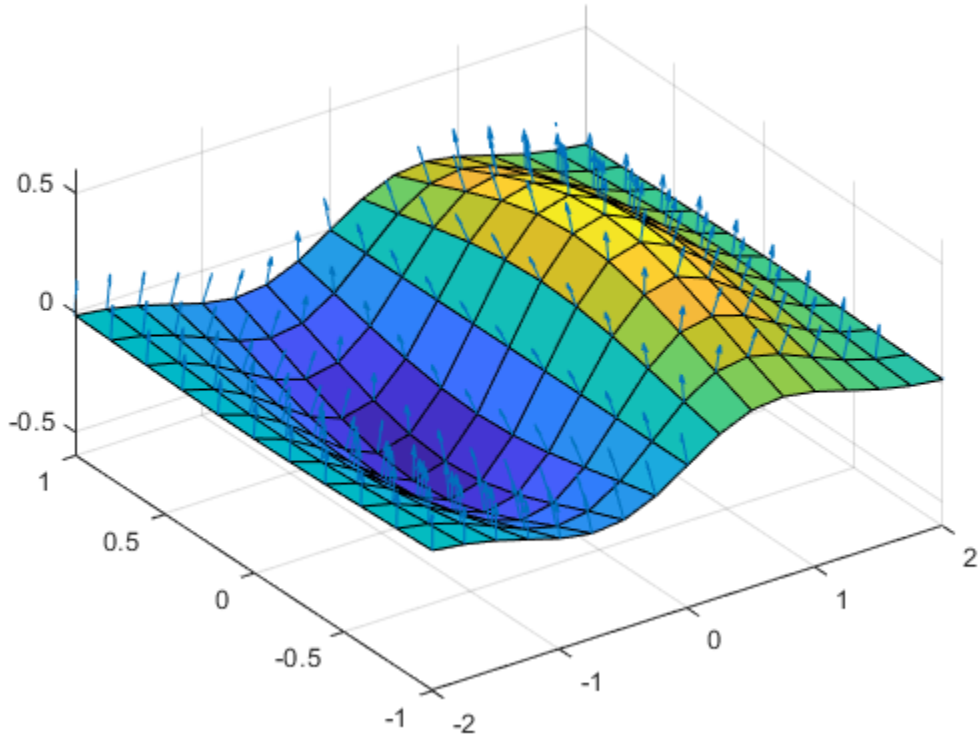


Plot Surface Normals

Plot the surface normals of the function $z = xe^{-x^2 - y^2}$.

```
[X,Y] = meshgrid(-2:0.25:2,-1:0.2:1);
Z = X.* exp(-X.^2 - Y.^2);
[U,V,W] = surfnorm(X,Y,Z);

figure
quiver3(X,Y,Z,U,V,W,0.5)
```

```
hold on
surf(X,Y,Z)
view(-35,45)
axis([-2 2 -1 1 -.6 .6])
hold off
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions

`LineSpec` | `axis` | `contour` | `plot` | `plot3` | `quiver` | `surfnorm` | `view`

Properties

Quiver

Topics

“Projectile Path Over Time”

Introduced before R2006a

Quiver Properties

Quiver chart appearance and behavior

Description

`Quiver` properties control the appearance and behavior of a `Quiver` object. By changing property values, you can modify certain aspects of the quiver chart.

Starting in R2014b, you can use dot notation to query and set properties.

```
q = quiver(1:10,1:10);
c = q.Color;
q.Color = 'red';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Arrows

`Color` — Arrow color

`[0 0 1]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Arrow color, specified as a three-element RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'blue'`

Example: `[0 0 1]`

`ColorMode` — Control how `Color` is set

`'auto'` (default) | `'manual'`

Control how the `Color` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the `Color` property by selecting a color from the `ColorOrder` property of the axes.
- `'manual'` — You control the value of the `Color` property manually, either by setting the value of the `Color` property directly on the object, or by including a color in the `LineSpec` argument when you call a plotting function.

If you change the value of the `Color` property manually, MATLAB changes the value of the `ColorMode` property to `'manual'`.

`LineStyle` — Style of arrow stem
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Style of arrow stem, specified as one of the line styles listed in this table.

Line Style	Description	Result
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No stem	No stem

`LineStyleMode` — Control how `LineStyle` is set
`'auto'` (default) | `'manual'`

Control how the `LineStyle` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the `LineStyle` property by selecting a line style from the `LineStyleOrder` property of the axes.
- `'manual'` — You control the value of the `LineStyle` property manually, either by setting the value of the `LineStyle` property directly on the object, or by specifying the `LineSpec` argument when you call a plotting function.

If you change the value of the `LineStyle` property manually, MATLAB changes the value of the `LineStyleMode` property to `'manual'`.

`LineWidth` — Width of arrow stem
`0.5` (default) | scalar numeric value

Width of arrow stem, specified as a scalar numeric value greater than zero in point units. One point equals `1/72` inch. The default value is `0.5` point.

Example: `0.75`

`ShowArrowHead` — Arrowhead display
`'on'` (default) | `'off'`

Arrowhead display, specified as one of these values:

- `'on'` — Display the vectors with arrowheads.
- `'off'` — Display the vectors without arrowheads.

`MaxHeadSize` — Maximum size of arrowhead
`0.2` (default) | scalar

Maximum size of arrowhead, specified as a scalar value in units relative to the length of the arrow.

Example: `0.1`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`AutoScale` — Automatic scaling of arrow length
`'on'` (default) | `'off'`

Automatic scaling of arrow length, specified as one of these values:

- `'on'` — Scale the arrow length to fit within the grid-defined coordinate data and scale arrows so that they do not overlap. The `quiver` function then applies the `AutoScaleFactor` to the arrow length.
- `'off'` — Do not scale the arrow lengths.

`AutoScaleFactor` — Scale factor
`0.9` (default) | scalar

Scale factor, specified as a scalar. A value of `2` doubles the length of the arrows. A value of `0.5` halves the arrow lengths.

This property has an effect only if the `AutoScale` property is set to `'on'`.

Example: 2

AlignVertexCenters — Sharp vertical and horizontal lines

'off' (default) | 'on'

Sharp vertical and horizontal lines, specified as 'off' or 'on'.

If the associated figure has a GraphicsSmoothing property set to 'on' and a Renderer property set to 'opengl', then the figure applies a smoothing technique to plots. In some cases, this smoothing technique can cause vertical and horizontal lines to appear uneven in thickness or color. Use the AlignVertexCenters property to eliminate the uneven appearance.

- 'off' — Do not sharpen vertical or horizontal lines. The lines might appear uneven in thickness or color.
- 'on' — Sharpen vertical and horizontal lines to eliminate an uneven appearance.

Note You must have a graphics card that supports this feature. To see if the feature is supported, call the rendererinfo function. If it is supported, rendererinfo returns value of 1 for info.Details.SupportsAlignVertexCenters.

Markers

Marker — Marker symbol

'none' (default) | 'o' | '+' | '*' | '.' | ...

Marker symbol, specified as one of the values listed in this table. By default, the object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

Value	Description
'o'	Circle
'+'	Plus sign
'*'	Asterisk
'.'	Point
'x'	Cross
'square' or 's'	Square

Value	Description
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerMode` — Control how `Marker` is set
`'auto'` (default) | `'manual'`

Control how the `Marker` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the object's `Marker` property.
- `'manual'` — You control the value of the `Marker` property manually, either by setting the value of the `Marker` property directly on the object, or by including a marker in the `LineSpec` argument when you call a plotting function.

If you change the value of the `Marker` property manually, MATLAB changes the value of the `MarkerMode` property to `'manual'`.

`MarkerSize` — Marker size
6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

`MarkerEdgeColor` — Marker outline color
`'auto'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'auto'` uses the same color as the `Color` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

MarkerFaceColor — Marker fill color

'none' (default) | 'auto' | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...

Marker fill color, specified as 'auto', an RGB triplet, a hexadecimal color code, a color name, or a short name. The 'auto' option uses the same color as the Color property of the parent axes. If you specify 'auto' and the axes plot box is invisible, the marker fill color is the color of the figure.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Data

`UData` — Vector lengths in x-direction

vector | matrix

Vector lengths in *x*-direction, specified as a vector or a matrix. The `UData`, `VData`, and `WData` properties together specify the components of the vectors displayed as arrows in the quiver chart.

Example: `1:10`

`UDataSource` — Variable linked to `UData`

`''` (default) | character vector | string

Variable linked to `UData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `UData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, MATLAB does not update the `UData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`VData` — Vector lengths in y-direction

vector | matrix

Vector lengths in *y*-direction, specified as a vector or a matrix. The `UData`, `VData`, and `WData` properties together specify the components of the vectors displayed as arrows in the quiver chart.

Example: `1:10`

`VDataSource` — Variable linked to `VData`

`''` (default) | character vector | string

Variable linked to `VData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `VData`.

By default, there is no linked variable so the value is an character vector, `''`. If you link a variable, MATLAB does not update the `VData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`WData` — Vector lengths in z-direction

vector | matrix

Vector lengths in *z*-direction, specified as a vector or a matrix. The `UData`, `VData`, and `WData` properties together specify the components of the vectors displayed as arrows in the quiver chart. For 2-D quiver charts, `WData` is an empty array.

Example: `1:10`

`WDataSource` — Variable linked to `WData`

`''` (default) | character vector | string

Variable linked to `WData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `WData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, MATLAB does not update the `WData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`XData` — x-coordinates

vector | matrix

x-coordinates, specified as a vector or matrix. The input argument `X` to the `quiver` function determines the *x*-coordinates. If you do not specify `X`, then `quiver` uses the indices of `UData` as the *x*-coordinates. `XData` must be equal in size to `YData`.

Setting this property sets the associated mode property to manual mode.

Example: `1:10`

`XDataMode` — Selection mode for `XData`

`'auto'` | `'manual'`

Selection mode for `XData`, specified as one of these values:

- `'auto'` — Automatically select the values.
- `'manual'` — Use manually specified values. To specify the values, set the `XData` property or use the input argument `X` to the function.

`XDataSource` — Variable linked to `XData`

`''` (default) | character vector | string

Variable linked to `XData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `XData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `XData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'x'`

`YData` — y-coordinates

vector | matrix

y-coordinates, specified as a vector or matrix. The input argument `Y` to the `quiver` function determines the *y*-coordinates. If you do not specify `Y`, then `quiver` uses the indices of `VData` as the *y*-coordinates. `YData` must be equal in size to `XData`.

Setting this property sets the associated mode property to manual mode.

Example: `1:10`

`YDataMode` — Selection mode for `YData`

`'auto'` | `'manual'`

Selection mode for `YData`, specified as one of these values:

- `'auto'` — Automatically select the values.
- `'manual'` — Use manually specified values. To specify the values, set the `YData` property or use the input argument `Y` to the function.

`YDataSource` — Variable linked to `YData`

`''` (default) | character vector | string

Variable linked to `YData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `YData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `YData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'y'`

`ZData` — z-coordinates
vector | matrix

z-coordinates, specified as a vector or matrix. The input argument `Z` to the `quiver3` function determines the *z*-coordinates. For 2-D quiver charts, `ZData` is an empty array. For 3-D quiver charts, `ZData` must be equal in size to `XData` and `YData`.

Example: `1:10`

`ZDataSource` — Variable linked to `ZData`
`''` (default) | character vector | string

Variable linked to `ZData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `ZData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `ZData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'z'`

Legend

`DisplayName` — Legend label
`''` (default) | character vector | string scalar

Legend label, specified as a character vector or string scalar. The legend does not display until you call the `legend` command. If you do not specify the text, then `legend` sets the label using the form `'dataN'`.

`Annotation` — Control for including or excluding object from legend
`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`DataTipTemplate` — Data tip content

`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

UIContextMenu — Context menu

empty GraphicsPlaceholder array (default) | ContextMenu object

Context menu, specified as a ContextMenu object. Use this property to display a context menu when you right-click the object. Create the context menu using the uicontextmenu function.

Note If the PickableParts property is set to 'none' or if the HitTest property is set to 'off', then the context menu does not appear.

Selected — Selection state

'off' (default) | 'on'

Selection state, specified as one of these values:

- 'on' — Selected. If you click the object when in plot edit mode, then MATLAB sets its Selected property to 'on'. If the SelectionHighlight property also is set to 'on', then MATLAB displays selection handles around the object.
- 'off' — Not selected.

SelectionHighlight — Display of selection handles

'on' (default) | 'off'

Display of selection handles when selected, specified as one of these values:

- 'on' — Display selection handles when the Selected property is set to 'on'.
- 'off' — Never display selection handles, even when the Selected property is set to 'on'.

Clipping — Clipping of object to axes limits

'on' (default) | 'off'

Clipping of the object to the axes limits, specified as one of these values:

- 'on' — Do not display parts of the object that are outside the axes limits.
- 'off' — Display the entire object, even if parts of it appear outside the axes limits. Parts of the object might appear outside the axes limits if you create a plot, set hold on, freeze the axis scaling, and then create the object so that it is larger than the original plot.

The `Clipping` property of the axes that contains the object must be set to `'on'`. Otherwise, this property has no effect. For more information about the clipping behavior, see the `Clipping` property of the axes.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.

- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks when visible. The `Visible` property must be set to `'on'` and you must click a part of the `Quiver` object that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The `HitTest` property determines if the `Quiver` object responds to the click or if an ancestor does.

- `'none'` — Cannot capture mouse clicks. Clicking the `Quiver` object passes the click to the object below it in the current view of the figure window. The `HitTest` property of the `Quiver` object has no effect.

`HitTest` — Response to captured mouse clicks

`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `Quiver` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `Quiver` object that has one of these:
  - `HitTest` property set to `'on'`
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

---

Note The `PickableParts` property determines if the `Quiver` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent

`Axes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `Group`, or `Transform` object.

Children — Children

empty GraphicsPlaceholder array | DataTip object array

Children, returned as an empty GraphicsPlaceholder array or a DataTip object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the Children property. To add a child to this list, set the Parent property of the DataTip object to the chart object.

HandleVisibility — Visibility of object handle

'on' (default) | 'off' | 'callback'

Visibility of the object handle in the Children property of the parent, specified as one of these values:

- 'on' — Object handle is always visible.
- 'off' — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the HandleVisibility to 'off' to temporarily hide the handle during the execution of that function.
- 'callback' — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the Children property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the get, findobj, gca, gcf, gco, newplot, cla, clf, and close functions.

Hidden object handles are still valid. Set the root ShowHiddenHandles property to 'on' to list all object handles regardless of their HandleVisibility property setting.

Identifiers

Type — Type of graphics object

'quiver'

This property is read-only.

Type of graphics object, returned as 'quiver'. Use this property to find all objects of a given type within a plotting hierarchy, such as searching for the type using findobj.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also
`quiver` | `quiver3`

Topics
“Access Property Values”
“Graphics Object Properties”

Introduced before R2006a

qz

QZ factorization for generalized eigenvalues

Syntax

```
[AA,BB,Q,Z] = qz(A,B)
[AA,BB,Q,Z,V,W] = qz(A,B)
qz(A,B,flag)
```

Description

The `qz` function gives access to intermediate results in the computation of generalized eigenvalues.

`[AA,BB,Q,Z] = qz(A,B)` for square matrices `A` and `B`, produces upper quasitriangular matrices `AA` and `BB`, and unitary matrices `Q` and `Z` such that `Q*A*Z = AA`, and `Q*B*Z = BB`. For complex matrices, `AA` and `BB` are triangular.

`[AA,BB,Q,Z,V,W] = qz(A,B)` also produces matrices `V` and `W` whose columns are generalized eigenvectors.

`qz(A,B,flag)` for real matrices `A` and `B`, produces one of two decompositions depending on the value of `flag`:

`'complex'`	Produces a possibly complex decomposition with a triangular `AA`. For compatibility with earlier versions, `'complex'` is the default.
`'real'`	Produces a real decomposition with a quasitriangular `AA`, containing 1-by-1 and 2-by-2 blocks on its diagonal.

If `AA` is triangular, then the diagonal elements `a = diag(AA)` and `b = diag(BB)` are the generalized eigenvalues that satisfy

```
A*V*b = B*V*a
b'*W'*A = a'*W'*B
```

The eigenvalues produced by `lambda = eig(A,B)` are the ratios of the diagonal elements `a` and `b`, such that `lambda = a./b`.

If `AA` is not triangular, it is necessary to further reduce the 2-by-2 blocks to obtain the eigenvalues of the full system.

See Also

`eig`

Introduced before R2006a

rad2deg

Convert angle from radians to degrees

Syntax

```
D = rad2deg(R)
```

Description

`D = rad2deg(R)` converts angle units from radians to degrees for each element of `R`.

Examples

`pi` in Degrees

Convert `pi` into degrees.

```
D = rad2deg(pi)
```

```
D = 180
```

Spherical Distance

Specify the mean radius of Earth and the distance from Munich to Bangalore measured along the Earth's surface (in kilometers). Compute the spherical distance between Munich and Bangalore in degrees.

```
dist = 7194;
radEarth = 6371;
R = dist/radEarth;
D = rad2deg(R)
```

```
D = 64.6972
```

Input Arguments

R — Angle in radians
scalar | vector | matrix | multidimensional array

Angle in radians, specified as a scalar, vector, matrix, or multidimensional array. If R contains complex elements, `rad2deg` converts the real and imaginary parts separately.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

D — Angle in degrees
scalar | vector | matrix | multidimensional array

Angles in degrees, returned as a scalar, vector, matrix, or multidimensional array. D is the same size as R.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`deg2rad`

Introduced in R2015b

rand

Uniformly distributed random numbers

Syntax

```
X = rand
X = rand(n)
X = rand(sz1,...,szN)
X = rand(sz)
```

```
X = rand( ___ ,typename)
X = rand( ___ ,'like',p)
```

Description

`X = rand` returns a single uniformly distributed random number in the interval (0,1).

`X = rand(n)` returns an `n`-by-`n` matrix of random numbers.

`X = rand(sz1,...,szN)` returns an `sz1`-by-...-by-`szN` array of random numbers where `sz1,...,szN` indicate the size of each dimension. For example, `rand(3,4)` returns a 3-by-4 matrix.

`X = rand(sz)` returns an array of random numbers where size vector `sz` specifies `size(X)`. For example, `rand([3 4])` returns a 3-by-4 matrix.

`X = rand( ___ ,typename)` returns an array of random numbers of data type `typename`. The `typename` input can be either `'single'` or `'double'`. You can use any of the input arguments in the previous syntaxes.

`X = rand( ___ ,'like',p)` returns an array of random numbers like `p`; that is, of the same object type as `p`. You can specify either `typename` or `'like'`, but not both.

Note The `'seed'`, `'state'`, and `'twister'` inputs to the `rand` function are not recommended. Use the `rng` function instead. For more information, see “Replace Discouraged Syntaxes of rand and randn”.

Examples

Matrix of Random Numbers

Generate a 5-by-5 matrix of uniformly distributed random numbers between 0 and 1.

```
r = rand(5)
```

```
r = 5×5

    0.8147    0.0975    0.1576    0.1419    0.6557
    0.9058    0.2785    0.9706    0.4218    0.0357
    0.1270    0.5469    0.9572    0.9157    0.8491
    0.9134    0.9575    0.4854    0.7922    0.9340
    0.6324    0.9649    0.8003    0.9595    0.6787
```

Random Numbers Within Specified Interval

Generate a 10-by-1 column vector of uniformly distributed numbers in the interval (-5,5).

```
r = -5 + (5+5)*rand(10,1)
```

```
r = 10×1

    3.1472
    4.0579
   -3.7301
    4.1338
    1.3236
   -4.0246
   -2.2150
    0.4688
    4.5751
    4.6489
```

In general, you can generate `N` random numbers in the interval (a,b) with the formula `r = a + (b-a).*rand(N,1)`.

Random Integers

Use the `randi` function (instead of `rand`) to generate 5 random integers from the uniform distribution between 10 and 50.

```
r = randi([10 50],1,5)
```

```
r = 1×5

    43    47    15    47    35
```

Random Complex Numbers

Generate a single random complex number with real and imaginary parts in the interval (0,1).

```
a = rand + 1i*rand
```

```
a = 0.8147 + 0.9058i
```

Reset Random Number Generator

Save the current state of the random number generator and create a 1-by-5 vector of random numbers.

```
s = rng;
r = rand(1,5)
```

```
r = 1×5

    0.8147    0.9058    0.1270    0.9134    0.6324
```

Restore the state of the random number generator to `s`, and then create a new 1-by-5 vector of random numbers. The values are the same as before.

```
rng(s);
r1 = rand(1,5)
```

```
r1 = 1×5

    0.8147    0.9058    0.1270    0.9134    0.6324
```

Always use the `rng` function (rather than the `rand` or `randn` functions) to specify the settings of the random number generator. For more information, see “Replace Discouraged Syntaxes of rand and randn”.

3-D Array of Random Numbers

Create a 3-by-2-by-3 array of random numbers.

```
X = rand([3,2,3])
```

```
X =
X(:,:,1) =

    0.8147    0.9134
    0.9058    0.6324
    0.1270    0.0975


X(:,:,2) =

    0.2785    0.9649
    0.5469    0.1576
    0.9575    0.9706


X(:,:,3) =

    0.9572    0.1419
    0.4854    0.4218
    0.8003    0.9157
```

Specify Data Type of Random Numbers

Create a 1-by-4 vector of random numbers whose elements are single precision.

```
r = rand(1,4,'single')
```

```
r = 1x4 single row vector

    0.8147    0.9058    0.1270    0.9134
```

```
class(r)
```

```
ans =
'single'
```

Clone Size from Existing Array

Create a matrix of random numbers with the same size as an existing array.

```
A = [3 2; -2 1];
sz = size(A);
X = rand(sz)
```

```
X = 2×2

    0.8147    0.1270
    0.9058    0.9134
```

It is a common pattern to combine the previous two lines of code into a single line:

```
X = rand(size(A));
```

Clone Size and Data Type from Existing Array

Create a 2-by-2 matrix of single precision random numbers.

```
p = single([3 2; -2 1]);
```

Create an array of random numbers that is the same size and data type as `p`.

```
X = rand(size(p),'like',p)
```

```
X = 2x2 single matrix

    0.8147    0.1270
    0.9058    0.9134
```

```
class(X)
```

```
ans =
'single'
```

Clone Distributed Array

If you have Parallel Computing Toolbox, create a 1000-by-1000 distributed array of random numbers with underlying data type `single`. For the `distributed` data type, the `'like'` syntax clones the underlying data type in addition to the primary data type.

```
p = rand(1000,'single','distributed');
```

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 6 workers.
```

Create an array of random numbers that is the same size, primary data type, and underlying data type as `p`.

```
X = rand(size(p),'like',p);
```

```
class(X)
```

```
ans =

distributed
```

```
classUnderlying(X)
```

```
ans =
single
```

Input Arguments

n — Size of square matrix

integer value

Size of square matrix, specified as an integer value.

- If `n` is `0`, then `X` is an empty matrix.
- If `n` is negative, then it is treated as `0`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

sz1,...,szN — Size of each dimension (as separate arguments)

integer values

Size of each dimension, specified as separate arguments of integer values.

- If the size of any dimension is `0`, then `X` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- Beyond the second dimension, `rand` ignores trailing dimensions with a size of 1. For example, `rand(3,1,1,1)` produces a 3-by-1 vector of random numbers.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

sz — Size of each dimension (as a row vector)

integer values

Size of each dimension, specified as a row vector of integer values. Each element of this vector indicates the size of the corresponding dimension:

- If the size of any dimension is `0`, then `X` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- Beyond the second dimension, `rand` ignores trailing dimensions with a size of 1. For example, `rand([3,1,1,1])` produces a 3-by-1 vector of random numbers.

Example: `sz = [2,3,4]` creates a 2-by-3-by-4 array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`typename` — Data type (class) to create
`'double'` (default) | `'single'`

Data type (class) to create, specified as `'double'`, `'single'`, or the name of another class that provides `rand` support.

Example: `rand(5,'single')`

`p` — Prototype of array to create
numeric array

Prototype of array to create, specified as a numeric array.

Example: `rand(5,'like',p)`

Data Types: `single` | `double`
Complex Number Support: Yes

Tips

- The sequence of numbers produced by `rand` is determined by the internal settings of the uniform pseudorandom number generator that underlies `rand`, `randi`, and `randn`. You can control that shared random number generator using `rng`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The data type (class) must be a built-in MATLAB numeric type. For other classes, the static `rand` method is not invoked. For example, `rand(sz,'myclass')` does not invoke `myclass.rand(sz)`.

- Size arguments must have a fixed size.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `rand` in the Parallel Computing Toolbox documentation.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `rand` in the Parallel Computing Toolbox documentation.

See Also

`RandStream` | `randi` | `randn` | `randperm` | `rng` | `sprand` | `sprandn`

Topics

“Create Arrays of Random Numbers”
“Random Numbers Within a Specific Range”
“Random Numbers Within a Sphere”
“Creating and Controlling a Random Number Stream”
“Class Support for Array-Creation Functions”
“Replace Discouraged Syntaxes of rand and randn”
“Why Do Random Numbers Repeat After Startup?”

Introduced before R2006a

rand (RandStream)

Uniformly distributed random numbers

Class

`RandStream`

`parallel.gpu.RandStream`

Syntax

```
r = rand(s,n)
r = rand(s,m,n)
r = rand(s,[m,n])
r = rand(s,m,n,p,...)
r = rand(s,[m,n,p,...])
r = rand(s)
r = rand(s,size(A))
r = rand(..., 'double')
r = rand(..., 'single')
```

Description

`r = rand(s,n)` returns an `n`-by-`n` matrix containing pseudorandom values drawn from the standard uniform distribution on the open interval (0,1). The values are drawn from the random stream `s`.

`r = rand(s,m,n)` or `r = rand(s,[m,n])` returns an `m`-by-`n` matrix.

`r = rand(s,m,n,p,...)` or `r = rand(s,[m,n,p,...])` returns an `m`-by-`n`-by-`p`-by-... array.

`r = rand(s)` returns a scalar.

`r = rand(s,size(A))` returns an array the same size as `A`.

`r = rand(..., 'double')` or `r = rand(..., 'single')` returns an array of uniform values of the specified class.

Note The size inputs `m`, `n`, `p`, ... should be nonnegative integers. Negative integers are treated as 0.

The sequence of numbers produced by `rand` is determined by the internal state of the random number stream `s`. Resetting that stream to the same fixed state allows computations to be repeated. Setting the stream to different states leads to unique computations, however, it does not improve any statistical properties.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When you provide `s` as a `parallel.gpu.RandStream` object, the computation is performed on the GPU. The result is returned as a `gpuArray`.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`@RandStream` | `parallel.gpu.RandStream` | `rand` | `randi (RandStream)` | `randn (RandStream)` | `randperm (RandStream)`

randi

Uniformly distributed pseudorandom integers

Syntax

```
X = randi(imax)
X = randi(imax,n)
X = randi(imax,sz1,...,szN)
X = randi(imax,sz)

X = randi(imax,classname)
X = randi(imax,n,classname)
X = randi(imax,sz1,...,szN,classname)
X = randi(imax,sz,classname)

X = randi(imax,'like',p)
X = randi(imax,n,'like',p)
X = randi(imax,sz1,...,szN,'like',p)
X = randi(imax,sz,'like',p)

X = randi([imin,imax], ___ )
```

Description

`X = randi(imax)` returns a pseudorandom scalar integer between `1` and `imax`.

`X = randi(imax,n)` returns an `n`-by-`n` matrix of pseudorandom integers drawn from the discrete uniform distribution on the interval [`1`,`imax`].

`X = randi(imax,sz1,...,szN)` returns an `sz1`-by-...-by-`szN` array where `sz1,...,szN` indicates the size of each dimension. For example, `randi(10,3,4)` returns a 3-by-4 array of pseudorandom integers between 1 and 10.

`X = randi(imax,sz)` returns an array where size vector `sz` defines `size(X)`. For example, `randi(10,[3,4])` returns a 3-by-4 array of pseudorandom integers between 1 and 10.

`X = randi(imax,classname)` returns a pseudorandom integer where `classname` specifies the data type. `classname` can be `'single'`, `'double'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, or `'uint32'`.

`X = randi(imax,n,classname)` returns an `n`-by-`n` array of data type `classname`.

`X = randi(imax,sz1,...,szN,classname)` returns an `sz1`-by-...-by-`szN` array of data type `classname`.

`X = randi(imax,sz,classname)` returns an array where size vector `sz` defines `size(X)` and `classname` defines `class(X)`.

`X = randi(imax,'like',p)` returns a pseudorandom integer like `p`; that is, with the same data type (class).

`X = randi(imax,n,'like',p)` returns an `n`-by-`n` array like `p`.

`X = randi(imax,sz1,...,szN,'like',p)` returns an `sz1`-by-...-by-`szN` array like `p`.

`X = randi(imax,sz,'like',p)` returns an array like `p` where size vector `sz` defines `size(X)`.

`X = randi([imin,imax], ___ )` returns an array containing integers drawn from the discrete uniform distribution on the interval [`imin`,`imax`], using any of the above syntaxes.

Examples

Square Matrix of Random Integers

Generate a 5-by-5 matrix of random integers between 1 and 10. The first input to `randi` indicates the largest integer in the sampling interval (the smallest integer in the interval is 1).

```
r = randi(10,5)
```

```
r = 5×5

     9     1     2     2     7
    10     3    10     5     1
```

```
     2     6    10    10     9
    10    10     5     8    10
     7    10     9    10     7
```

Random Integers Within Specified Interval

Generate a 10-by-1 column vector of uniformly distributed random integers from the sample interval `[-5,5]`.

```
r = randi([-5,5],10,1)
```

```
r = 10×1

     3
     4
    -4
     5
     1
    -4
    -2
     1
     5
     5
```

Control Random Number Generation

Save the current state of the random number generator and create a 1-by-5 vector of random integers.

```
s = rng;
r = randi(10,1,5)
```

```
r = 1×5

     9    10     2    10     7
```

Restore the state of the random number generator to `s`, and then create a new 1-by-5 vector of random integers. The values are the same as before.

```
rng(s);
r1 = randi(10,1,5)
```

```
r1 = 1×5

     9    10     2    10     7
```

Always use the `rng` function (rather than the `rand` or `randn` functions) to specify the settings of the random number generator. For more information, see “Replace Discouraged Syntaxes of rand and randn”.

3-D Array of Random Integers

Create a 3-by-2-by-3 array of uniformly distributed random integers between 1 and 500.

```
X = randi(500,[3,2,3])
```

```
X =
X(:,:,1) =

   408   457
   453   317
    64    49


X(:,:,2) =

   140   483
   274    79
   479   486


X(:,:,3) =

   479    71
   243   211
   401   458
```

Random Integers of Other Data Types

Create a 1-by-4 vector of random numbers whose elements are of type `int16`.

```
r = randi(100,1,4,'int16')
```

```
r = 1x4 int16 row vector

   82   91   13   92
```

```
class(r)
```

```
ans =
'int16'
```

Size Defined by Existing Array

Create a matrix of uniformly distributed random integers between 1 and 10 with the same size as an existing array.

```
A = [3 2; -2 1];
sz = size(A);
X = randi(10,sz)
```

```
X = 2×2

     9     2
    10    10
```

It is a common pattern to combine the previous two lines of code into a single line:

```
X = randi(10,size(A));
```

Size and Numeric Data Type Defined by Existing Array

Create a 2-by-2 matrix of 8-bit signed integers.

```
p = int8([3 2; -2 1]);
```

Create an array of random integers that is the same size and data type as `p`.

```
X = randi(10,size(p),'like',p)
```

```
X = 2x2 int8 matrix

    9    2
   10   10
```

```
class(X)
```

```
ans =
'int8'
```

Input Arguments

`imax` — Largest integer in sample interval
positive integer

Largest integer in sample interval, specified as a positive integer. `randi` draws values from the uniform distribution in the sample interval `[1,imax]`.

Example: `randi(10,5)`

`imin` — Smallest integer in sample interval
1 (default) | scalar integer

Smallest integer in sample interval, specified as a scalar integer.

Both `imin` and `imax` must be integers that satisfy `imin` ≤ `imax`.

For example, `randi([50,100],5)` returns a 5-by-5 matrix of random integers between (and including) 50 and 100.

`n` — Size of square matrix
integer value

Size of square matrix, specified as an integer value.

- If `n` is `0`, then `X` is an empty matrix.
- If `n` is negative, then it is treated as `0`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz1,...,szN` — Size of each dimension (as separate arguments)
two or more integer values

Size of each dimension, specified as separate arguments of integer values.

- If the size of any dimension is `0`, then `X` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- Beyond the second dimension, `randi` ignores trailing dimensions with a size of 1. For example, `randi([5,10],3,1,1,1)` produces a 3-by-1 vector of random integers between 5 and 10.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz` — Size of each dimension (as a row vector)
integer values

Size of each dimension, specified as a row vector of integer values. Each element of this vector indicates the size of the corresponding dimension:

- If the size of any dimension is `0`, then `X` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- Beyond the second dimension, `randi` ignores trailing dimensions with a size of 1. For example, `randi([5,10],[3,1,1,1])` produces a 3-by-1 vector of random integers between 5 and 10.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`classname` — Data type (class) to create
`'double'` (default) | `'single'` | `'int8'` | `'uint8'` | ...

Output class, specified as `'double'`, `'single'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, or the name of another class that provides `randi` support.

Example: `randi(5,5,'int8')`

Data Types: `char`

p — Prototype of array to create
numeric array

Prototype of array to create, specified as a numeric array.

Example: `randi(5,5,'like',p)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Tips

- The sequence of numbers produced by `randi` is determined by the internal settings of the uniform pseudorandom number generator that underlies `rand`, `randi`, and `randn`. You can control that shared random number generator using `rng`.
- The arrays returned by `randi` might contain repeated integer values. This behavior is sometimes referred to as sampling with replacement. Use `randperm` if you require all unique values.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The data type (class) must be a built-in MATLAB numeric type. Does not invoke the static `randi` method for other classes. For example, `randi(imax,sz,'myclass')` does not invoke `myclass.randi(imax,sz)`.
- Size arguments must have a fixed size.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `randi` in the Parallel Computing Toolbox documentation.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `randi` in the Parallel Computing Toolbox documentation.

See Also

`RandStream` | `rand` | `randn` | `randperm` | `rng`

Topics

"Random Integers"
"Create Arrays of Random Numbers"
"Generate Random Numbers That Are Repeatable"
"Generate Random Numbers That Are Different"
"Creating and Controlling a Random Number Stream"
"Class Support for Array-Creation Functions"
"Why Do Random Numbers Repeat After Startup?"

Introduced in R2008b

randi (RandStream)

Uniformly distributed pseudorandom integers

Class

`RandStream`

`parallel.gpu.RandStream`

Syntax

```
r = randi(s,imax,n)
r = randi(s,imax,m,n)
r = randi(s,imax,[m,n])
r = randi(s,imax,m,n,p,...)
r = randi(s,imax,[m,n,p,...])
r = randi(s,imax)
r = randi(s,imax,size(A))
r = randi(s,[imin,imax],...)
r = randi(...,classname)
```

Description

`r = randi(s,imax,n)` returns an `n`-by-`n` matrix containing pseudorandom integer values drawn from the discrete uniform distribution on `1:imax`. `randi` draws those values from the random stream `s`.

`r = randi(s,imax,m,n)` or `r = randi(s,imax,[m,n])` returns an `m`-by-`n` matrix.

`r = randi(s,imax,m,n,p,...)` or `r = randi(s,imax,[m,n,p,...])` returns an `m`-by-`n`-by-`p`-by-... array.

`r = randi(s,imax)` returns a scalar.

`r = randi(s,imax,size(A))` returns an array the same size as `A`.

`r = randi(s,[imin,imax],...)` returns an array containing integer values drawn from the discrete uniform distribution on `imin:imax`.

`r = randi(...,classname)` returns an array of integer values of class `classname`. `classname` does not support 64-bit integers.

Note The size inputs `m`, `n`, `p`, ... should be nonnegative integers. Negative integers are treated as 0.

The arrays returned by `randi` might contain repeated integer values. This is sometimes referred to as sampling with replacement. To get unique integer values, sometimes referred to as sampling without replacement, use `randperm (RandStream)`.

The sequence of numbers produced by `randi` is determined by the internal state of the random stream `s`. `randi` uses one uniform value from `s` to generate each integer value. Resetting `s` to the same fixed state allows computations to be repeated. Setting the stream to different states leads to unique computations, however, it does not improve any statistical properties.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When you provide `s` as a `parallel.gpu.RandStream` object, the computation is performed on the GPU. The result is returned as a `gpuArray`.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`RandStream` | `parallel.gpu.RandStream` | `rand (RandStream)` | `randi` | `randn (RandStream)` | `randperm (RandStream)`

randn

Normally distributed random numbers

Syntax

```
X = randn
X = randn(n)
X = randn(sz1,...,szN)
X = randn(sz)

X = randn( ___ ,typename)
X = randn( ___ ,'like',p)
```

Description

`X = randn` returns a random scalar drawn from the standard normal distribution.

`X = randn(n)` returns an `n`-by-`n` matrix of normally distributed random numbers.

`X = randn(sz1,...,szN)` returns an `sz1`-by-...-by-`szN` array of random numbers where `sz1,...,szN` indicate the size of each dimension. For example, `randn(3,4)` returns a 3-by-4 matrix.

`X = randn(sz)` returns an array of random numbers where size vector `sz` defines `size(X)`. For example, `randn([3 4])` returns a 3-by-4 matrix.

`X = randn( ___ ,typename)` returns an array of random numbers of data type `typename`. The `typename` input can be either `'single'` or `'double'`. You can use any of the input arguments in the previous syntaxes.

`X = randn( ___ ,'like',p)` returns an array of random numbers like `p`; that is, of the same object type as `p`. You can specify either `typename` or `'like'`, but not both.

Note The `'seed'`, `'state'`, and `'twister'` inputs to the `randn` function are not recommended. Use the `rng` function instead. For more information, see “Replace Discouraged Syntaxes of rand and randn”.

Examples

Matrix of Random Numbers

Generate a 5-by-5 matrix of normally distributed random numbers.

```
r = randn(5)
```

```
r = 5×5

    0.5377   -1.3077   -1.3499   -0.2050    0.6715
    1.8339   -0.4336    3.0349   -0.1241   -1.2075
   -2.2588    0.3426    0.7254    1.4897    0.7172
    0.8622    3.5784   -0.0631    1.4090    1.6302
    0.3188    2.7694    0.7147    1.4172    0.4889
```

Bivariate Normal Random Numbers

Generate values from a bivariate normal distribution with specified mean vector and covariance matrix.

```
mu = [1 2];
sigma = [1 0.5; 0.5 2];
R = chol(sigma);
z = repmat(mu,10,1) + randn(10,2)*R
```

```
z = 10×2

    1.5377    0.4831
    2.8339    6.9318
   -1.2588    1.8302
    1.8622    2.3477
    1.3188    3.1049
   -0.3077    1.0750
    0.5664    1.6190
    1.3426    4.1420
    4.5784    5.6532
    3.7694    5.2595
```

Random Complex Numbers

Generate a single random complex number with normally distributed real and imaginary parts.

```
a = randn + 1i*randn
```

```
a = 0.5377 + 1.8339i
```

Reset Random Number Generator

Save the current state of the random number generator and create a 1-by-5 vector of random numbers.

```
s = rng;
r = randn(1,5)
```

```
r = 1×5

    0.5377    1.8339   -2.2588    0.8622    0.3188
```

Restore the state of the random number generator to `s`, and then create a new 1-by-5 vector of random numbers. The values are the same as before.

```
rng(s);
r1 = randn(1,5)
```

```
r1 = 1×5

    0.5377    1.8339   -2.2588    0.8622    0.3188
```

Always use the `rng` function (rather than the `rand` or `randn` functions) to specify the settings of the random number generator. For more information, see “Replace Discouraged Syntaxes of rand and randn”.

3-D Array of Random Numbers

Create a 3-by-2-by-3 array of random numbers.

```
X = randn([3,2,3])
```

```
X =
X(:,:,1) =

    0.5377    0.8622
    1.8339    0.3188
   -2.2588   -1.3077


X(:,:,2) =

   -0.4336    2.7694
    0.3426   -1.3499
    3.5784    3.0349


X(:,:,3) =

    0.7254   -0.2050
   -0.0631   -0.1241
    0.7147    1.4897
```

Specify Data Type of Random Numbers

Create a 1-by-4 vector of random numbers whose elements are single precision.

```
r = randn(1,4,'single')
```

```
r = 1x4 single row vector

    0.5377    1.8339   -2.2588    0.8622
```

```
class(r)
```

```
ans =
'single'
```

Clone Size from Existing Array

Create a matrix of normally distributed random numbers with the same size as an existing array.

```
A = [3 2; -2 1];
sz = size(A);
X = randn(sz)
```

```
X = 2×2

    0.5377   -2.2588
    1.8339    0.8622
```

It is a common pattern to combine the previous two lines of code into a single line:

```
X = randn(size(A));
```

Clone Size and Data Type from Existing Array

Create a 2-by-2 matrix of single precision random numbers.

```
p = single([3 2; -2 1]);
```

Create an array of random numbers that is the same size and data type as `p`.

```
X = randn(size(p),'like',p)
```

```
X = 2x2 single matrix

    0.5377   -2.2588
    1.8339    0.8622
```

```
class(X)
```

```
ans =
'single'
```

Clone Distributed Array

If you have Parallel Computing Toolbox, create a 1000-by-1000 distributed array of random numbers with underlying data type `single`. For the `distributed` data type, the `'like'` syntax clones the underlying data type in addition to the primary data type.

```
p = randn(1000,'single','distributed');
```

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 6 workers.
```

Create an array of random numbers that is the same size, primary data type, and underlying data type as `p`.

```
X = randn(size(p),'like',p);
```

```
class(X)
```

```
ans =

distributed
```

```
classUnderlying(X)
```

```
ans =
single
```

Input Arguments

n — Size of square matrix
integer value

Size of square matrix, specified as an integer value.

- If `n` is `0`, then `X` is an empty matrix.
- If `n` is negative, then it is treated as `0`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz1,...,szN` — Size of each dimension (as separate arguments)
integer values

Size of each dimension, specified as separate arguments of integer values.

- If the size of any dimension is `0`, then `X` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- Beyond the second dimension, `randn` ignores trailing dimensions with a size of 1. For example, `randn(3,1,1,1)` produces a 3-by-1 vector of random numbers.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz` — Size of each dimension (as a row vector)
integer values

Size of each dimension, specified as a row vector of integer values. Each element of this vector indicates the size of the corresponding dimension:

- If the size of any dimension is `0`, then `X` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- Beyond the second dimension, `randn` ignores trailing dimensions with a size of 1. For example, `randn([3,1,1,1])` produces a 3-by-1 vector of random numbers.

Example: `sz = [2,3,4]` creates a 2-by-3-by-4 array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`typename` — Data type (class) to create
`'double'` (default) | `'single'`

Data type (class) to create, specified as `'double'`, `'single'`, or the name of another class that provides `randn` support.

Example: `randn(5,'single')`

`p` — Prototype of array to create
numeric array

Prototype of array to create, specified as a numeric array.

Example: `randn(5,'like',p)`

Data Types: `single` | `double`

Tips

- The sequence of numbers produced by `randn` is determined by the internal settings of the uniform pseudorandom number generator that underlies `rand`, `randi`, and `randn`. You can control that shared random number generator using `rng`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The data type (class) must be a built-in MATLAB numeric type. For other classes, the static `randn` method is not invoked. For example, `randn(sz,'myclass')` does not invoke `myclass.randn(sz)`.
- Size arguments must have a fixed size.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `randn` in the Parallel Computing Toolbox documentation.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `randn` in the Parallel Computing Toolbox documentation.

See Also

`RandStream` | `rand` | `randi` | `randperm` | `rng` | `sprand` | `sprandn`

Topics

"Create Arrays of Random Numbers"
"Random Numbers Within a Specific Range"
"Random Numbers Within a Sphere"
"Random Numbers from Normal Distribution with Specific Mean and Variance"
"Creating and Controlling a Random Number Stream"
"Class Support for Array-Creation Functions"
"Replace Discouraged Syntaxes of rand and randn"
"Why Do Random Numbers Repeat After Startup?"

Introduced before R2006a

randn (RandStream)

Normally distributed pseudorandom numbers

Class

`RandStream`

`parallel.gpu.RandStream`

Syntax

```
r = randn(s,m,n)
r = randn(s,[m,n])
r = randn(s,m,n,p,...)
r = randn(s,[m,n,p,...])
r = randn(s)
r = randn(s,size(A))
r = randn(...,'double')
r = randn(...,'single')
```

Description

`r = randn(s,n)` returns an `n`-by-`n` matrix containing pseudorandom values drawn from the standard normal distribution. `randn` draws those values from the random stream `s`.

`r = randn(s,m,n)` or `r = randn(s,[m,n])` returns an `m`-by-`n` matrix.

`r = randn(s,m,n,p,...)` or `r = randn(s,[m,n,p,...])` returns an `m`-by-`n`-by-`p`-by-... array.

`r = randn(s)` returns a scalar.

`r = randn(s,size(A))` returns an array the same size as `A`.

`r = randn(...,'double')` or `r = randn(...,'single')` returns an array of uniform values of the specified class.

Note The size inputs `m`, `n`, `p`, ... should be nonnegative integers. Negative integers are treated as 0.

The sequence of numbers produced by `randn` is determined by the internal state of the random stream `s`. `randn` uses one or more uniform values from `s` to generate each normal value. Resetting that stream to the same fixed state allows computations to be repeated. Setting the stream to different states leads to unique computations, however, it does not improve any statistical properties.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When you provide `s` as a `parallel.gpu.RandStream` object, the computation is performed on the GPU. The result is returned as a `gpuArray`.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`RandStream` | `parallel.gpu.RandStream` | `rand (RandStream)` | `randi (RandStream)` | `randn`

randperm

Random permutation of integers

Syntax

```
p = randperm(n)
p = randperm(n,k)
```

Description

`p = randperm(n)` returns a row vector containing a random permutation of the integers from 1 to `n` without repeating elements.

`p = randperm(n,k)` returns a row vector containing `k` unique integers selected randomly from 1 to `n`.

Examples

Random Permutation of Integers

Generate a random permutation of the integers from 1 to 6. The input to `randperm` indicates the largest integer in the sampling interval (the smallest integer in the interval is 1).

```
r = randperm(6)
```

```
r = 1×6

     6     3     5     1     2     4
```

Permute a Selection of Integers

Generate a random permutation of four unique integers (without repeating elements) selected randomly from the integers 1 to 8.

```
r1 = randperm(8,4)
```

```
r1 = 1×4

     6     4     7     3
```

Generate another random permutation of four unique integers.

```
r2 = randperm(8,4)
```

```
r2 = 1×4

     8     7     5     4
```

Control Random Number Generation

Save the current state of the random number generator and create a random permutation of the integers from 1 to 8.

```
s = rng;
r = randperm(8)
```

```
r = 1×8

     6     3     7     8     5     1     2     4
```

Restore the state of the random number generator to `s`, and then create a new random permutation of the integers from 1 to 8. The permutation is the same as before.

```
rng(s)
r1 = randperm(8)
```

```
r1 = 1×8
```

```
     6     3     7     8     5     1     2     4
```

You can use the `rng` function to specify the settings of the random number generator.

Input Arguments

n — Number of integers in sample interval
positive integer

Number of integers in sample interval, specified as a positive integer. `randperm` permutes integer values from 1 to `n` inclusive.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

k — Number of selected integers
positive integer

Number of selected integers, specified as a positive integer. `k` must also be less than or equal to `n`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Tips

- The sequence of numbers produced by `randperm` is determined by the internal settings of the uniform pseudorandom number generator that underlies `rand`, `randi`, `randn`, and `randperm`. To control that shared random number generator, use the `rng` function.
- The arrays returned by `randperm` contain permutation of integers without repeating integer values. This behavior is sometimes referred to as sampling without replacement. If you require repeating values, use the `randi` function.
- `randperm(n)` and `randperm(n,n)` both generate permutations of the integers 1 through `n`, but they can give different random orderings in the permutations. For large `n`, `randperm(n,n)` is faster than `randperm(n)`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The syntax `p = randperm(n,k)` is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`nchoosek` | `perms` | `permute` | `randi` | `randperm(RandStream)` | `rng`

Introduced before R2006a

randperm (RandStream)

Random permutation

Class

`RandStream`

`parallel.gpu.RandStream`

Syntax

```
p = randperm(s,n)
p = randperm(s,n,k)
```

Description

`p = randperm(s,n)` returns a row vector containing a random permutation of integers from 1 to `n` inclusive. `randperm(s,n)` uses random values drawn from the random stream `s`.

`p = randperm(s,n,k)` returns a row vector containing `k` unique integers selected randomly from 1 to `n` inclusive.

Examples

Create a random stream `s` and generate a random permutation of the integers from 1 to 6 based on `s`:

```
s = RandStream('mt19937ar','Seed',0);
randperm(s,6)
```

MATLAB returns the vector

```
[6     3     5     1     2     4]
```

Use the random stream `s` to generate three integers between 1 and 10:

```
randperm(s,10,3)
ans =
      1     8     9
```

Tips

For `p = randperm(s,n,k)`, `p` contains `k` *unique* values. `randperm` performs k-permutations (sampling without replacement). To allow repeated values in the output (sampling with replacement), use `randi(s,n,1,k)`.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- When you provide `s` as a `parallel.gpu.RandStream` object, the computation is performed on the GPU. The result is returned as a `gpuArray`.
- The syntax `randperm(s,n,k)` is not supported.

For more information, see "Run MATLAB Functions on a GPU" (Parallel Computing Toolbox).

See Also

`nchoosek` | `parallel.gpu.RandStream` | `perms` | `permute` | `rand` | `randi (RandStream)` | `randperm`

RandStream

Random number stream

Constructor

`RandStream`

Description

Pseudorandom numbers in MATLAB come from one or more random number streams. The simplest way to generate arrays of random numbers is to use `rand`, `randn`, or `randi`. These functions all rely on the same stream of uniform random numbers, known as the global stream. You can create other streams that act separately from the global stream, and you can use their `rand`, `randi`, or `randn` methods to generate arrays of random numbers. You can also create a random number stream and make it the global stream.

To create a single random number stream, use the `RandStream` constructor. To create multiple independent random number streams, use `RandStream.create`. The `rng` function provides a simple interface to create a new global stream.

`stream = RandStream.getGlobalStream` returns the global random number stream, that is, the one currently used by the `rand`, `randi`, and `randn` functions.

`prevstream = RandStream.setGlobalStream(stream)` designates the random number stream `stream` as the new global stream to be used by the `rand`, `randi`, and `randn` functions, and returns the previous global stream.

A random number stream `s` has properties that control its behavior. Access or assign to a property using `p = s.Property` or `s.Property = p`. The following table lists defined properties:

Properties

Property	Description
`Type`	(Read-only) Generator algorithm used by the stream. The list of possible generators is given by `RandStream.list`.
`Seed`	(Read-only) Seed value used to create the stream.
`NumStreams`	(Read-only) Number of streams in the group in which the current stream was created.
`StreamIndex`	(Read-only) Index of the current stream from among the group of streams with which it was created.
`State`	Internal state of the generator. You should not depend on the format of this property. The value you assign to `S.State` must be a value read from `S.State` previously. Use `reset` to return a stream to a predictable state without having previously read from the `State` property. The sequence of random numbers produced by a random number stream `s` is determined by the internal state of its random number generator. Saving and restoring the generator's internal state with the `State` property allows you to reproduce a sequence of random numbers.
`Substream`	Index of the substream to which the stream is currently set. The default is 1. Multiple substreams are not supported by all generator types; the multiplicative lagged Fibonacci generator (`mlfg6331_64`) and combined multiple recursive generator (`mrg32k3a`) support substreams.
`NormalTransform`	Transformation algorithm used by `randn(s, ...)` to generate normal pseudorandom values. Possible values are `'Ziggurat'`, `'Polar'`, or `'Inversion'`.

Property	Description
`Antithetic`	Logical value indicating whether `S` generates antithetic pseudorandom values, that is, the usual values subtracted from 1. The default is false.
`FullPrecision`	Logical value indicating whether `S` generates values using its full precision. Some generators can create pseudorandom values faster, but with fewer random bits, if `FullPrecision` is false. The default is true.

Methods

Method	Description
`RandStream`	Create a random number stream.
`RandStream.create`	Create multiple independent random number streams.
`get`	Get the properties of a random stream object.
`list`	List available random number generator algorithms.
`set`	Set random stream property.
`RandStream.getGlobalStream`	Get the global random number stream.
`RandStream.setGlobalStream`	Set global random number stream.
`reset`	Reset a stream to its initial internal state
`rand`	Pseudorandom numbers from a uniform distribution
`randn`	Pseudorandom numbers from a standard normal distribution
`randi`	Pseudorandom integers from a uniform discrete distribution
`randperm`	Random permutation of a set of values

Examples

Example 1

Create a single stream and designate it as the current global stream:

```
s = RandStream('mt19937ar','Seed',1);
RandStream.setGlobalStream(s);
```

Example 2

Create three independent streams:

```
[s1,s2,s3] = RandStream.create('mrg32k3a','NumStreams',3);
r1 = rand(s1,100000,1);
r2 = rand(s2,100000,1);
r3 = rand(s3,100000,1);
corrcoef([r1,r2,r3])
```

Example 3

Create only one stream from a set of three independent streams, and designate it as the current global stream:

```
s2 = RandStream.create('mrg32k3a','NumStreams',3,...
     'StreamIndices',2);
RandStream.setGlobalStream(s2);
```

Example 4

Reset the global random number stream that underlies `rand`, `randi`, and `randn` back to its beginning, to reproduce previous results:

```
stream = RandStream.getGlobalStream;
reset(stream);
```

Example 5

Save and restore the current global stream's state to reproduce the output of `rand`:

```
stream = RandStream.getGlobalStream;
savedState = stream.State;
u1 = rand(1,5)
u1 =
    0.8147    0.9058    0.1270    0.9134    0.6324

stream.State = savedState;
u2 = rand(1,5)
u2 =
    0.8147    0.9058    0.1270    0.9134    0.6324
```

`u2` contains exactly the same values as `u1`.

Example 6

Reset the global random number stream to its initial settings. This causes `rand`, `randi`, and `randn` to start over, as if in a new MATLAB session:

```
s = RandStream('mt19937ar','Seed',0);
RandStream.setGlobalStream(s);
```

Example 7

Reinitialize the global random number stream using a seed based on the current time. This causes `rand`, `randi`, and `randn` to return different values in different MATLAB sessions. It is usually not desirable to do this more than once per MATLAB session as it may affect the statistical properties of the random numbers MATLAB produces:

```
s = RandStream('mt19937ar','Seed','shuffle');
RandStream.setGlobalStream(s);
```

Example 8

Change the transformation algorithm that `randn` uses to create normal pseudorandom values from uniform values. This does not replace or reset the global stream.

```
stream = RandStream.getGlobalStream;
stream.NormalTransform = 'inversion'
```

See Also

`rand` | `randi` | `randn` | `rng`

RandStream constructor

Random number stream

Class

`RandStream`

Syntax

```
s = RandStream('gentype')
s = RandStream('gentype',Name,Value)
```

Description

`s = RandStream('gentype')` creates a random number stream that uses the uniform pseudorandom number generator algorithm specified by `gentype`. `RandStream.list` returns all possible values for `gentype`, or see “Choosing a Random Number Generator” for details on generator algorithms.

`s = RandStream('gentype',Name,Value)` allows you to specify one or more optional `Name,Value` pairs to control creation of the stream.

Once you have created a random, you can use `RandStream.setGlobalStream` to make it the global stream, so that the functions `rand`, `randi`, and `randn` draw values from it.

Parameters for `RandStream` are:

Parameter	Description
`Seed`	Nonnegative scalar integer with which to initialize all streams. Seeds must be an integer between 0 and $2^{32} - 1$ or `'shuffle'` to create a seed based on the current time. Default is 0.

Parameter	Description
`NormalTransform`	Transformation algorithm used by `randn(s, ...)` to generate normal pseudorandom values. Possible values are `'Ziggurat'`, `'Polar'`, or `'Inversion'`.

Examples

Example 1

Create a random number stream, make it the global stream, and save and restore its state to reproduce the output of `randn`:

```
s = RandStream('mrg32k3a');
RandStream.setGlobalStream(s);
savedState = s.State;
z1 = randn(1,5)
z1 =
   -0.1894   -1.4426   -0.3592    0.8883   -0.4337
s.State = savedState;
z2 = randn(1,5)
z2 =
   -0.1894   -1.4426   -0.3592    0.8883   -0.4337
```

`z2` contains exactly the same values as `z1`.

Example 2

Return `rand`, `randi`, and `randn` to their default startup settings:

```
s = RandStream('mt19937ar','Seed',0)
RandStream.setGlobalStream(s);
```

Example 3

Replace the current global random number stream with a stream whose seed is based on the current time, so `rand`, `randi`, and `randn` will return different values in different MATLAB sessions. It is usually not desirable to do this more than once per MATLAB

session as it may affect the statistical properties of the random numbers MATLAB produces:

```
s = RandStream('mt19937ar','Seed','shuffle');
RandStream.setGlobalStream(s);
```

Tips

- Streams created using `RandStream` might not be independent from each other. Use `RandStream.create` to create multiple streams that are independent.

See Also

`RandStream` | `RandStream.create` | `RandStream.getGlobalStream` | `RandStream.list` | `RandStream.rand` | `RandStream.randi` | `RandStream.randn` | `RandStream.setGlobalStream` | `rng`

RandStream.getGlobalStream

Current global random number stream

Class

`@RandStream`

Syntax

```
stream = RandStream.getGlobalStream
```

Description

`stream = RandStream.getGlobalStream` returns the current global random number stream.

`rand`, `randi`, and `randn` all rely on the same stream of uniform pseudorandom numbers, known as the global stream. `rand` draws one value from that stream to generate each uniform value it returns. `randi` draws one uniform value from that stream to generate each integer value it returns. And `randn` draws one or more uniform values to generate each normal value it returns. Note that there are also `rand`, `randi`, and `randn` methods for which you specify a specific random stream from which to draw values.

Note The `rng` function is a shorter alternative for many common uses of `RandStream.getGlobalStream`.

See Also

`RandStream` | `RandStream.setGlobalStream` | `rand` | `randi` | `randn` | `rng`

Topics

"Creating and Controlling a Random Number Stream"

“Managing the Global Stream”

RandStream.setGlobalStream

Set global random number stream

Syntax

```
prevstream = RandStream.setGlobalStream(stream)
```

Description

`prevstream = RandStream.setGlobalStream(stream)` designates the random number stream, specified as `stream`, to be the global stream that the `rand`, `randi`, and `randn` functions draw values from. It returns the previous global random number stream as `prevstream`.

`rand`, `randi`, and `randn` all rely on the same stream of uniform pseudorandom numbers, known as the global stream. `rand` draws one value from that stream to generate each uniform value it returns. `randi` draws one uniform value from that stream to generate each integer value it returns. And `randn` draws one or more uniform values to generate each normal value it returns. Note that there are also `rand`, `randi`, and `randn` methods for which you specify a specific random stream from which to draw values.

Note The `rng` function is a shorter alternative for many common uses of `RandStream.setGlobalStream`.

See Also

`RandStream` | `RandStream.getGlobalStream` | `rand` | `randi` | `randn` | `rng`

Topics

"Creating and Controlling a Random Number Stream"
"Managing the Global Stream"

rank

Rank of matrix

Syntax

```
k = rank(A)
k = rank(A,tol)
```

Description

`k = rank(A)` returns the rank on page 1-12749 of matrix `A`.

Use `sprank` to determine the structural rank of a sparse matrix.

`k = rank(A,tol)` specifies a different tolerance to use in the rank computation. The rank is computed as the number of singular values of `A` that are larger than `tol`.

Examples

Rank of Matrix

Determine whether a matrix is full rank.

Create a 3-by-3 matrix. The values in the third column are twice as large as those in the second column.

```
A = [3 2 4; -1 1 2; 9 5 10]
```

```
A = 3×3

     3     2     4
    -1     1     2
     9     5    10
```

Calculate the rank of the matrix. If the matrix is full rank, then the rank is equal to the number of columns, `size(A,2)`.

```
rank(A)
```

```
ans = 2
```

```
size(A,2)
```

```
ans = 3
```

Since the columns are linearly dependent, the matrix is rank deficient.

Specify Rank Tolerance

Calculate the rank of a matrix using a tolerance.

Create a 4-by-4 diagonal matrix. The diagonal has one small value equal to `1e-15`.

```
A = [10 0 0 0; 0 25 0 0; 0 0 34 0; 0 0 0 1e-15]
```

```
A = 4×4

   10.0000         0         0         0
         0   25.0000         0         0
         0         0   34.0000         0
         0         0         0    0.0000
```

Calculate the rank of the matrix.

```
rank(A)
```

```
ans = 3
```

The matrix is not considered to be full rank, since the default algorithm calculates the number of singular values larger than `max(size(A))*eps(norm(A))`. For this matrix, the small value on the diagonal is excluded since it is smaller than the tolerance.

Calculate the rank of the matrix again, but specify a tolerance of `1e-16`.

```
rank(A,1e-16)
```

```
ans = 4
```

Input Arguments

A — Input matrix
matrix

Input matrix.

Data Types: `single` | `double`
Complex Number Support: Yes

`tol` — Tolerance
`max(size(A))*eps(norm(A))` | scalar

Tolerance, specified as a scalar. See the "Algorithms" on page 1-12749 section for more information.

Example: `rank(A,1e-5)`

More About

Rank

The number of linearly independent columns in a matrix is the *rank* of the matrix. The row and column rank of a matrix are always equal.

A matrix is *full rank* if its rank is the highest possible for a matrix of the same size, and *rank deficient* if it does not have full rank. The rank gives a measure of the dimension of the *range* or *column space* of the matrix, which is the collection of all linear combinations of the columns.

Algorithms

`rank` uses a method based on the singular value decomposition, or SVD. The SVD algorithm is more time consuming than some alternatives, but it is also the most reliable.

The rank of a matrix `A` is computed as the number of singular values that are larger than a tolerance. By default, the tolerance is `max(size(A))*eps(norm(A))`. However, you can specify a different tolerance with the command `rank(A,tol)`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`null` | `orth` | `sprank` | `svd`

Introduced before R2006a

rat

Rational fraction approximation

Syntax

```
R = rat(X)
R = rat(X,tol)
```

```
[N,D] = rat( ___ )
```

Description

`R = rat(X)` returns the rational fraction approximation on page 1-12754 of `X` to within the default tolerance, `1e-6*norm(X(:),1)`. The approximation is a character array containing the truncated continued fractional expansion.

`R = rat(X,tol)` approximates `X` to within the tolerance, `tol`.

`[N,D] = rat( ___ )` returns two arrays, `N` and `D`, such that `N./D` approximates `X`, using any of the above syntaxes.

Examples

Approximate Value of π

Approximate the value of *π* using a rational representation of the quantity `pi`.

The mathematical quantity *π* is not a rational number, but the quantity `pi` that approximates it *is* a rational number since all floating-point numbers are rational.

Find the rational representation of `pi`.

```
format rat
pi
```

```
ans =
     355/113
```

The resulting expression is a character vector. You also can use `rats(pi)` to get the same answer.

Use `rat` to see the continued fractional expansion of `pi`.

```
R = rat(pi)
```

```
R =
'3 + 1/(7 + 1/(16))'
```

The result is an approximation by continued fractional expansion. If you consider the first two terms of the expansion, you get the approximation $3 + \frac{1}{7} = \frac{22}{7}$, which only agrees with `pi` to 2 decimals.

However, if you consider all three terms printed by `rat`, you can recover the value `355/113`, which agrees with `pi` to 6 decimals.

$$3 + \frac{1}{7 + \frac{1}{16}} = \frac{355}{113}$$

Specify a tolerance for additional accuracy in the approximation.

```
R = rat(pi,1e-7)
```

```
R =
'3 + 1/(7 + 1/(16 + 1/(-294)))'
```

The resulting approximation, `104348/33215`, agrees with `pi` to 9 decimals.

Express Array Elements as Ratios

Create a 4-by-4 matrix.

```
format short;
X = hilb(4)
```

```
X = 4×4

    1.0000    0.5000    0.3333    0.2500
    0.5000    0.3333    0.2500    0.2000
    0.3333    0.2500    0.2000    0.1667
    0.2500    0.2000    0.1667    0.1429
```

Express the elements of X as ratios of small integers using `rat`.

```
[N,D] = rat(X)
```

```
N = 4×4

     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1


D = 4×4

     1     2     3     4
     2     3     4     5
     3     4     5     6
     4     5     6     7
```

The two matrices, N and D, approximate X with `N./D`.

View the elements of X as ratios using `format rat`.

```
format rat
X
```

```
X =
       1              1/2            1/3            1/4
       1/2            1/3            1/4            1/5
       1/3            1/4            1/5            1/6
       1/4            1/5            1/6            1/7
```

In this form, it is clear that N contains the numerators of each fraction and D contains the denominators.

Input Arguments

X — Input array
numeric array

Input array, specified as a numeric array of class `single` or `double`.

Data Types: `single` | `double`
Complex Number Support: Yes

tol — Tolerance
scalar

Tolerance, specified as a scalar. `N` and `D` approximate `X`, such that `N./D - X < tol`. The default tolerance is `1e-6*norm(X(:),1)`.

Output Arguments

R — Continued fraction
character array

Continued fraction, returned as a character array with `m` rows, where `m` is the number of elements in `X`. The accuracy of the rational approximation via continued fractions increases with the number of terms.

N — Numerator
numeric array

Numerator, returned as a numeric array. `N./D` approximates `X`.

D — Denominator
numeric array

Denominator, returned as a numeric array. `N./D` approximates `X`.

Algorithms

Even though all floating-point numbers are rational numbers, it is sometimes desirable to approximate them by simple rational numbers, which are fractions whose numerator and

denominator are small integers. Rational approximations are generated by truncating continued fraction expansions.

The `rat` function approximates each element of `X` by a continued fraction of the form

$$\frac{N}{D} = D_1 + \frac{1}{D_2 + \frac{1}{\left(D_3 + \ldots + \frac{1}{D_k}\right)}} \ .$$

The *D*s are obtained by repeatedly picking off the integer part and then taking the reciprocal of the fractional part. The accuracy of the approximation increases exponentially with the number of terms and is worst when `X = sqrt(2)`. For `X = sqrt(2)` , the error with `k` terms is about `2.68*(.173)^k`, so each additional term increases the accuracy by less than one decimal digit. It takes 21 terms to get full floating-point accuracy.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For code generation, only the two output syntax is supported.

See Also

`format` | `rats`

Introduced before R2006a

rats

Rational output

Syntax

```
S = rats(X)
S = rats(X,strlen)
```

Description

`S = rats(X)` returns a character vector containing the rational approximations to the elements of `X` using the default length of `13`.

`rats` returns asterisks for elements that cannot be printed in the allotted space, but which are not negligible compared to the other elements in `X`.

`S = rats(X,strlen)` returns a character vector of length `strlen`. The rational approximation uses a tolerance that is inversely proportional to the length.

Examples

Rational Representation of Matrix

Create a 4-by-4 matrix.

```
format short
X = hilb(4)
```

```
X = 4×4

    1.0000    0.5000    0.3333    0.2500
    0.5000    0.3333    0.2500    0.2000
    0.3333    0.2500    0.2000    0.1667
    0.2500    0.2000    0.1667    0.1429
```

View the rational representation of the matrix using `rats`. The result is the same as using `format rat`.

```
R = rats(X)
```

```
R = 4x56 char array
    '       1            1/2           1/3           1/4     '
    '      1/2           1/3           1/4           1/5     '
    '      1/3           1/4           1/5           1/6     '
    '      1/4           1/5           1/6           1/7     '
```

Adjust Output Vector Length

Find the rational representation of `pi` with the default character vector length and approximation tolerance. The result is the same as using `format rat`.

```
rats(pi)
```

```
ans =
'    355/113   '
```

Adjust the length of the output, which also adjusts the approximation tolerance.

```
rats(pi,20)
```

```
ans =
'    104348/33215     '
```

The resulting rational approximation has greater accuracy. As the output length increases, the tolerance decreases.

Adjust the output length again to achieve greater accuracy.

```
rats(pi,25)
```

```
ans =
'      1146408/364913      '
```

The resulting approximation agrees with `pi` to 10 decimal places.

Input Arguments

X — Input array
numeric array

Input array, specified as a numeric array of class `single` or `double`.

Data Types: `single` | `double`
Complex Number Support: Yes

`strlen` — Length of character vector
positive integer

Length of character vector, specified as a positive integer. The default length is 13, which allows for 6 elements in 78 spaces.

Output Arguments

S — Rational output
character vector

Rational output, returned as a character vector.

Algorithms

`rats` obtains rational approximations with `[N,D] = rat(X,tol)`, where `tol` is `min(10^(-(strlen-1)/2)*norm(X(isfinite(X)),1),.1)`. Thus, the tolerance is inversely proportional to the output length, `strlen`.

See Also

`format` | `rat`

Introduced before R2006a

rbbox

Create rubberband box for area selection

Syntax

```
rbbox
rbbox(initialRect)
rbbox(initialRect,fixedPoint)
rbbox(initialRect,fixedPoint,stepSize)
finalRect = rbbox(...)
```

Description

`rbbox` initializes and tracks a rubberband box in the current figure. It sets the initial rectangular size of the box to 0, anchors the box at the figure's `CurrentPoint`, and begins tracking from this point.

`rbbox(initialRect)` specifies the initial location and size of the rubberband box as `[x y width height]`, where `x` and `y` define the lower left corner, and `width` and `height` define the size. `initialRect` is in the units specified by the current figure's `Units` property, and measured from the lower left corner of the figure window. The corner of the box closest to the pointer position follows the pointer until `rbbox` receives a button-up event.

`rbbox(initialRect,fixedPoint)` specifies the corner of the box that remains fixed. All arguments are in the units specified by the current figure's `Units` property, and measured from the lower left corner of the figure window. `fixedPoint` is a two-element vector, `[x y]`. The tracking point is the corner diametrically opposite the anchored corner defined by `fixedPoint`.

`rbbox(initialRect,fixedPoint,stepSize)` specifies how frequently the rubberband box is updated. When the tracking point exceeds `stepSize` figure units, `rbbox` redraws the rubberband box. The default stepsize is `1`.

`finalRect = rbbox(...)` returns a four-element vector, `[x y width height]`, where `x` and `y` are the *x* and *y* components of the lower left corner of the box, and `width` and `height` are the dimensions of the box.

Examples

Create an annotation rectangle by rubber banding the rectangle size in the figure.

```
set(gcf,'Units','normalized')
k = waitforbuttonpress;
rect_pos = rbbox;
annotation('rectangle',rect_pos,'Color','red')
```

Execute the code, click down and drag a rectangle within the figure. Releasing the mouse button draws a rectangle in the figure.

Tips

`rbbox` is useful for defining and resizing a rectangular region:

- For box definition, `initialRect` is `[x y 0 0]`, where `(x,y)` is the figure's `CurrentPoint`.
- For box resizing, `initialRect` defines the rectangular region that you resize (e.g., a legend). `fixedPoint` is the corner diametrically opposite the tracking point.

`rbbox` returns immediately if a button is not currently pressed. Therefore, you use `rbbox` with `waitforbuttonpress` so that the mouse button is down when `rbbox` is called. `rbbox` returns when you release the mouse button.

See Also

`axis` | `dragrect` | `waitforbuttonpress`

Introduced before R2006a

rcond

Reciprocal condition number

Syntax

```
C = rcond(A)
```

Description

`C = rcond(A)` returns an estimate for the reciprocal condition of `A` in 1-norm. If `A` is well conditioned, `rcond(A)` is near 1.0. If `A` is badly conditioned, `rcond(A)` is near 0.

Examples

Sensitivity of Badly Conditioned Matrix

Examine the sensitivity of a badly conditioned matrix.

A notable matrix that is symmetric and positive definite, but badly conditioned, is the Hilbert matrix. The elements of the Hilbert matrix are $H(i, j) = 1/(i + j - 1)$.

Create a 10-by-10 Hilbert matrix.

```
A = hilb(10);
```

Find the reciprocal condition number of the matrix.

```
C = rcond(A)
```

```
C = 2.8286e-14
```

The reciprocal condition number is small, so `A` is badly conditioned.

The condition of `A` has an effect on the solutions of similar linear systems of equations. To see this, compare the solution of $Ax = b$ to that of the perturbed system, $Ax = b + 0.01$.

Create a column vector of ones and solve $Ax = b$.

```
b = ones(10,1);
x = A\b;
```

Now change b by `0.01` and solve the perturbed system.

```
b1 = b + 0.01;
x1 = A\b1;
```

Compare the solutions, `x` and `x1`.

```
norm(x-x1)
```

```
ans = 1.1250e+05
```

Since `A` is badly conditioned, a small change in `b` produces a very large change (on the order of 1e5) in the solution to `x = A\b`. The system is sensitive to perturbations.

Find Condition of Identity Matrix

Examine why the reciprocal condition number is a more accurate measure of singularity than the determinant.

Create a 5-by-5 multiple of the identity matrix.

```
A = eye(5)*0.01;
```

This matrix is full rank and has five equal singular values, which you can confirm by calculating `svd(A)`.

Calculate the determinant of `A`.

```
det(A)
```

```
ans = 1.0000e-10
```

Although the determinant of the matrix is close to zero, `A` is actually very well conditioned and *not* close to being singular.

Calculate the reciprocal condition number of `A`.

```
rcond(A)
```

```
ans = 1
```

The matrix has a reciprocal condition number of `1` and is, therefore, very well conditioned. Use `rcond(A)` or `cond(A)` rather than `det(A)` to confirm singularity of a matrix.

Input Arguments

`A` — Input matrix
square numeric matrix

Input matrix, specified as a square numeric matrix.

Data Types: `single` | `double`

Output Arguments

`C` — Reciprocal condition number
scalar

Reciprocal condition number, returned as a scalar. The data type of `C` is the same as `A`.

The reciprocal condition number is a scale-invariant measure of how close a given matrix is to the set of singular matrices.

- If `C` is near 0, the matrix is nearly singular and badly conditioned.
- If `C` is near 1.0, the matrix is well conditioned.

Tips

- `rcond` is a more efficient but less reliable method of estimating the condition of a matrix compared to the condition number, `cond`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

See Also

`cond` | `condest` | `norm` | `normest` | `rank` | `svd`

Introduced before R2006a

rdivide, ./

Right array division

Syntax

```
x = A./B
x = rdivide(A,B)
```

Description

`x = A./B` divides each element of `A` by the corresponding element of `B`. The sizes of `A` and `B` must be the same or be compatible.

If the sizes of `A` and `B` are compatible, then the two arrays implicitly expand to match each other. For example, if one of `A` or `B` is a scalar, then the scalar is combined with each element of the other array. Also, vectors with different orientations (one row vector and one column vector) implicitly expand to form a matrix.

`x = rdivide(A,B)` is an alternative way to divide `A` by `B`, but is rarely used. It enables operator overloading for classes.

Examples

Divide Two Numeric Arrays

Create two numeric arrays, `A` and `B`, and divide the second array, `B`, into the first, `A`.

```
A = [2 4 6 8; 3 5 7 9];
B = 10*ones(2,4);
x = A./B
```

```
x = 2×4

    0.2000    0.4000    0.6000    0.8000
```

```
    0.3000    0.5000    0.7000    0.9000
```

Integer Division

Divide an `int16` scalar value by each element of an `int16` vector.

```
a = int16(10);
b = int16([3 4 6]);
x = a./b
```

```
x = 1x3 int16 row vector

   3   3   2
```

MATLAB® rounds the results when dividing integer data types.

Divide Scalar by Array

Create an array and divide it into a scalar.

```
C = 5;
D = magic(3);
x = C./D
```

```
x = 3×3

    0.6250    5.0000    0.8333
    1.6667    1.0000    0.7143
    1.2500    0.5556    2.5000
```

When you specify a scalar value to be divided by an array, the scalar value expands into an array of the same size, then element-by-element division is performed.

Divide Row and Column Vectors

Create a 1-by-2 row vector and 3-by-1 column vector and divide them.

```
a = 1:2;
b = (1:3)';
a ./ b
```

```
ans = 3×2

    1.0000    2.0000
    0.5000    1.0000
    0.3333    0.6667
```

The result is a 3-by-2 matrix, where each (i,j) element in the matrix is equal to `a(j) ./ b(i)`:

$$a = [a_1\ a_2], \qquad b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}, \qquad a\ ./\ b = \begin{bmatrix} a_1\ ./\ b_1 & a_2\ ./\ b_1 \\ a_1\ ./\ b_2 & a_2\ ./\ b_2 \\ a_1\ ./\ b_3 & a_2\ ./\ b_3 \end{bmatrix}.$$

Input Arguments

A, B — Operands
scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Numeric inputs `A` and `B` must either be the same size or have sizes that are compatible (for example, `A` is an `M`-by-`N` matrix and `B` is a scalar or `1`-by-`N` row vector). For more information, see “Compatible Array Sizes for Basic Operations”.

- If `A` or `B` is an integer data type, then the other input must be the same integer type or be a scalar double. Operands with an integer data type cannot be complex.
- If `A` and `B` are duration arrays, then they must be the same size unless one is a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `duration` | `char`
Complex Number Support: Yes

Tips

- The element-wise operators `./` and `.\` are related to each other by the equation `A./B = B.\A`.
- When dividing integers, use `idivide` for more rounding options.
- MATLAB does not support complex integer division.

Compatibility Considerations

Implicit expansion change affects arguments for operators

Behavior changed in R2016b

Starting in R2016b with the addition of implicit expansion, some combinations of arguments for basic operations that previously returned errors now produce results. For example, you previously could not add a row and a column vector, but those operands are now valid for addition. In other words, an expression like `[1 2] + [1; 2]` previously returned a size mismatch error, but now it executes.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a `try/catch` block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see “Compatible Array Sizes for Basic Operations”.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- 64-bit integers are not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`idivide` | `ldivide` | `mldivide` | `mrdivide`

Topics

“Array vs. Matrix Operations”
“Operator Precedence”
“MATLAB Operators and Special Characters”

Introduced before R2006a

matlab.settings.reloadFactoryFile

Package: `matlab.settings`

Load or reload factory settings

Syntax

```
matlab.settings.reloadFactoryFile(toolboxName)
```

Description

`matlab.settings.reloadFactoryFile(toolboxName)` loads or reloads the factory settings tree into MATLAB. Use `matlab.settings.reloadFactoryFile` to inspect the factory settings tree while creating it, without having to restart MATLAB. This function is meant for debugging purposes only and should not be included in shipping toolbox code.

Note

- `matlab.settings.reloadFactoryFile` does not reload the `settingsInfo.json` file. You must restart MATLAB after changing the file.
- You must recreate any variables that reference the specified toolbox after calling `matlab.settings.reloadFactoryFile`. For example, if you create the variable `a = s.mytoolbox` and then call `matlab.settings.reloadFactoryFile`, you must recreate `a` to access the updated settings for `mytoolbox`.

Examples

Test Factory Settings Tree

Create a toolbox factory tree and then test that the tree is created correctly.

Create the function `createMyToolboxFactoryTree` that creates the factory settings tree for the toolbox `mytoolbox`.

```
function myToolboxFactoryTree = createMyToolboxFactoryTree()
    myToolboxFactoryTree = matlab.settings.FactoryGroup.createToolboxGroup('mytoolbox', ...
        'Hidden',false);

    toolboxFontGroup = addGroup(myToolboxFactoryTree,'font','Hidden',false)
    addSetting(toolboxFontGroup,'FontSize','FactoryValue',11,'Hidden',false, ...
        'ValidationFcn',@matlab.settings.mustBeNumericScalar)
    addSetting(toolboxFontGroup,'FontColor','FactoryValue','Black', ...
        'Hidden',false,'ValidationFcn',@matlab.settings.mustBeStringScalar);
end
```

Create the `settingsInfo.json` file for the toolbox. Specify `mytoolbox` as the root settings group name and `createMyToolboxFactoryTree` as the settings tree creation function. Place `settingsInfo.json` in the toolbox `resources` folder.

```
{
"ToolboxGroupName" : "mytoolbox",
"Hidden" : "false",
"CreateTreeFcn" : "createMyToolboxFactoryTree"
}
```

Add the folder that contains the settings tree creation function and the toolbox resources folder to the MATLAB path. Then, load the factory settings tree for `mytoolbox`.

```
matlab.settings.reloadFactoryFile('mytoolbox');
```

Use the `settings` function to access the root of the settings tree and display the value of the `FontSize` setting.

```
s = settings;
s.mytoolbox.font.FontSize
```

```
ans =
  Setting 'mytoolbox.font.FontSize' with properties:
       ActiveValue: 11
    TemporaryValue: <no value>
     PersonalValue: <no value>
      FactoryValue: 11
```

Input Arguments

`toolboxName` — Name of toolbox

string scalar | character vector

Name of toolbox to reload the factory settings for, specified as a string scalar or character vector.

Example: `matlab.settings.reloadFactoryFile('mytoolbox');`

See Also

`addGroup` | `addSetting` | `matlab.settings.FactoryGroup.createToolboxGroup` | `matlab.settings.loadSettingsCompatibilityResults`

Topics

"Create Factory Settings for Toolboxes"
"Access and Modify Settings"

Introduced in R2019b

rendererinfo

Graphics renderer information

Syntax

```
info = rendererinfo(target)
```

Description

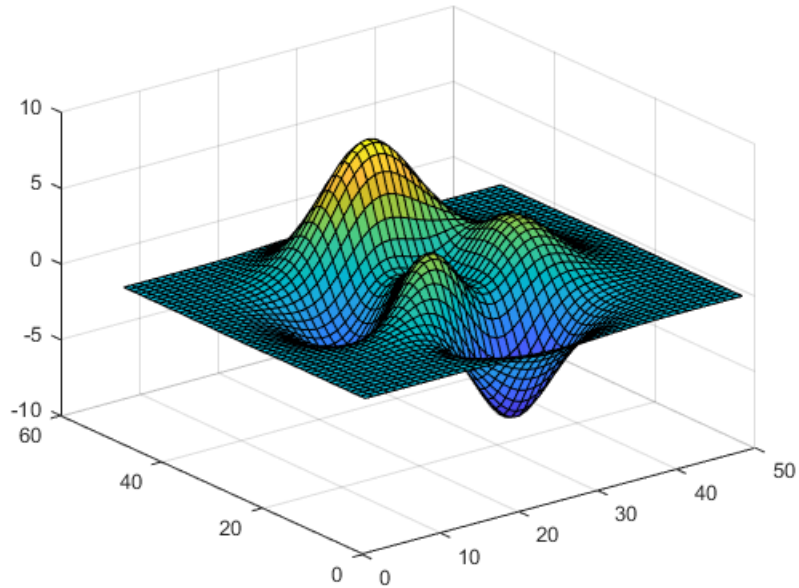
`info = rendererinfo(target)` returns a structure containing the renderer information for the target graphics object. Specify `target` as any type of axes or a chart that can be a child of a figure. You can also specify an array of *n* axes or charts, in which case `info` is returned as a 1-by-n structure array.

Examples

Information About Axes

Create a surface plot of the `peaks` function.

```
surf(peaks)
```



Get the current axes, and then get renderer information for the axes. Your system might return different information.

```
ax = gca;
info = rendererinfo(ax)
```

```
info =

  struct with fields:

    GraphicsRenderer: 'OpenGL Hardware'
              Vendor: 'NVIDIA Corporation'
             Version: '4.6.0 NVIDIA 391.58'
      RendererDevice: 'Quadro P600/PCIe/SSE2'
             Details: [1×1 struct]
```

Get the driver details.

```
info.Details
```

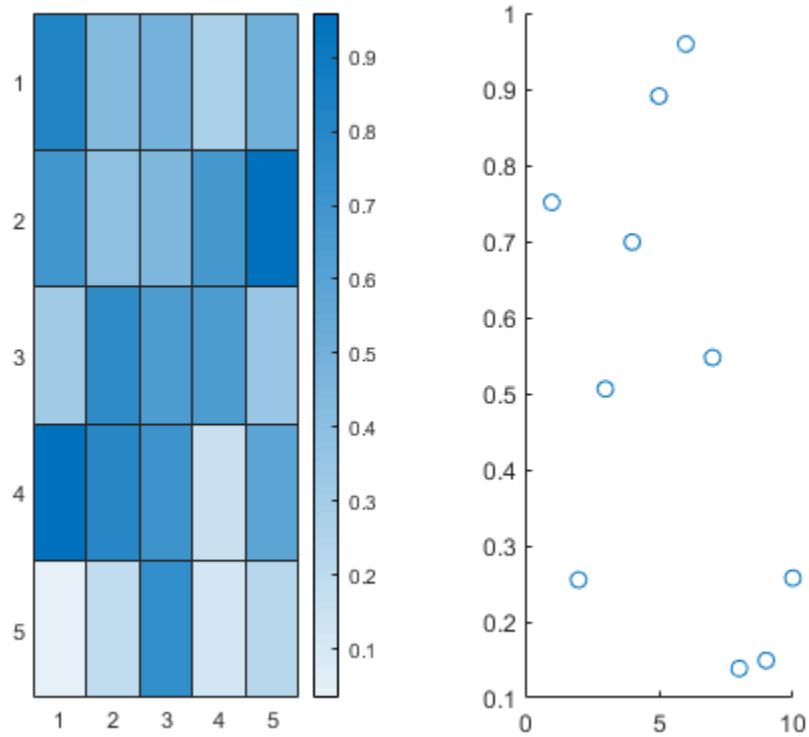
```
ans =
```

```
  struct with fields:

           RendererDriverVersion: '23.21.13.9174'
       RendererDriverReleaseDate: '2018-6-4'
            HardwareSupportLevel: 'Full'
    SupportsDepthPeelTransparency: 1
       SupportsAlignVertexCenters: 1
       SupportsGraphicsSmoothing: 1
                  MaxTextureSize: 32768
               MaxFrameBufferSize: 32768
```

Information About Multiple Target Objects

Create a subplot containing a heatmap chart and a scatter plot.

```
subplot(1,2,1)
h = heatmap(rand(5));
ax1 = subplot(1,2,2);
scatter(1:10,rand(1,10))
```



Get the renderer information for the heatmap chart and the parent axes of the scatter plot. In this case, `info` is an array that contains two structures.

```
info = rendererinfo([h ax1])
```

```
info =

  1×2 struct array with fields:

    GraphicsRenderer
    Vendor
    Version
    RendererDevice
    Details
```

Index into the array to get the renderer version for the heatmap chart. Your system might return different version information.

```
info(1).Version
```

```
ans =

    '4.6.0 NVIDIA 391.74'
```

Input Arguments

`target` — Target object

axes | chart | array of axes or charts

Target object, specified as one of the following:

- Any type of axes, such as an `Axes`, `PolarAxes`, or `GeographicAxes` object.
- A chart that can be a child of a figure. For example, a `HeatmapChart` object can be the target object because it can be a child of a figure. By contrast, a `Stem` object cannot be the target object because it is a child of an `Axes` object.
- An array of axes, charts, or a combination of them.

Output Arguments

`info` — Renderer information

structure

Renderer information, returned as a structure that contains information such as the name of the graphics renderer, vendor, and version. The `Details` field is a nested structure that contains additional details. Both the `info` structure and the `info.Details` structure are described in the tables below.

If you specify `target` as an array of *n* axes or charts, `info` is returned as a 1-by-n structure array. Each structure in the array corresponds to an element of `target`.

Info Structure

All systems return these fields.

Field	Description
`GraphicsRenderer`	Graphics renderer, returned as one of these values: • `'OpenGL Hardware'` • `'OpenGL Software'` • `'MathWorks Painters'` • `'WebGL'`
`Vendor`	Manufacturer of the graphics renderer implementation.
`Version`	Version of the graphics renderer implementation.
`RendererDevice`	Device that supports the graphics renderer. If you are using hardware-accelerated graphics, this field is the graphics card model name.
`Details`	Nested structure that contains additional details, such as the renderer's driver version. For the Painters renderer, this structure is empty.

Details Structure

Some systems return a subset of these fields, depending on the graphics renderer. For the Painters renderer, the `info.Details` structure is empty.

Field	Description
`RendererDriverVersion`	Version of the OpenGL driver. This field displays only on Windows systems running hardware-accelerated OpenGL. This field does not display on virtual machines.
`RendererDriverReleaseDate`	Release date of the OpenGL driver. This field displays only on Windows systems running hardware-accelerated OpenGL. This field does not display on virtual machines.

Field	Description
`HardwareSupportLevel`	Hardware support level, returned as one of these values: • `'Full'` — MATLAB uses graphics hardware as much as possible to provide advanced graphics features on page 1-12780. Most systems running hardware-accelerated OpenGL and WebGL™ return this value. • `'Basic'` — MATLAB uses graphics hardware, but some features are disabled. Disabling graphics features is sometimes necessary to avoid graphics display issues. • `'None'` — MATLAB does not use any graphics hardware. All systems running software OpenGL return this value. On some systems, software implementations of certain features might be supported. If MATLAB detects an unsupported driver, this field also contains `'known graphics driver issues'`.
`SupportsDepthPeelTransparency`	Depth peel transparency feature support, returned as `logical(1)` if supported and `logical(0)` otherwise.
`SupportsAlignVertexCenters`	Align vertex centers feature support, returned as `logical(1)` if supported and `logical(0)` otherwise.
`SupportsGraphicsSmoothing`	Graphics smoothing feature support, returned as `logical(1)` if supported and `logical(0)` otherwise.
`MaxTextureSize`	Maximum texture size that the renderer supports (in pixels).

Field	Description
`MaxFrameBufferSize`	Maximum frame buffer size that the renderer supports (in pixels).

Limitations

The `rendererinfo` function is not supported in the Live Editor.

More About

Advanced Graphics Features

Advanced graphics features are features that require certain renderer implementations. These features are graphics smoothing on page 1-12781, depth peel transparency on page 1-12782, align vertex centers on page 1-12782, and hardware-accelerated markers on page 1-12783. You can tell whether your system supports some or all of these features by getting the value of the `HardwareSupportLevel` field of the `info.Details` structure.

This table lists the advanced graphics features and the circumstances under which they are supported. In some cases, certain features are supported, but they are disabled to avoid graphics display issues.

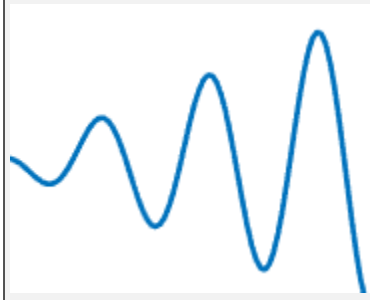
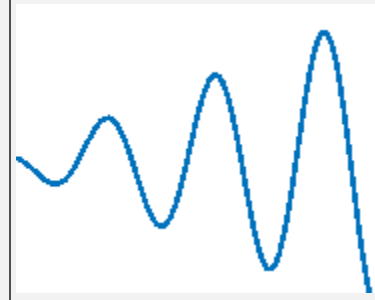
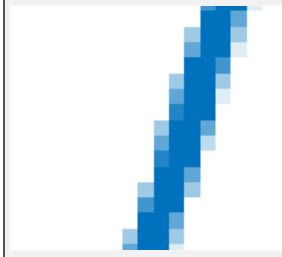
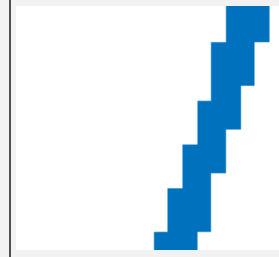
Graphics Feature	Hardware OpenGL	Basic Hardware OpenGL	Software OpenGL on Windows	Software OpenGL on Linux	WebGL
Graphics Smoothing	Supported for OpenGL 2.1 or higher	Supported for OpenGL 2.1 or higher	Not supported	Not supported	Supported
Depth Peel Transparency	Supported for OpenGL 2.1 or higher	Disabled	Not supported	Supported	Supported
Align Vertex Centers	Supported for OpenGL 2.1 or higher	Disabled	Not supported	Not supported	Supported

Graphics Feature	Hardware OpenGL	Basic Hardware OpenGL	Software OpenGL on Windows	Software OpenGL on Linux	WebGL
Hardware-accelerated markers	Supported for OpenGL 4.0 or higher	Disabled	Not supported	Not supported	Supported

Graphics Smoothing

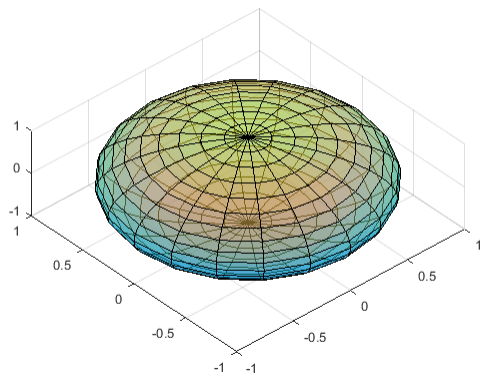
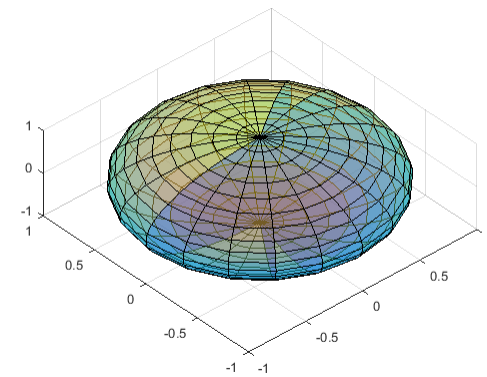
Graphics smoothing improves the appearance of plots by reducing jagged lines. By default, this feature is enabled if your system supports it. To turn off this feature for a particular figure, set the `GraphicsSmoothing` property of the figure to `'off'`.

This table shows the difference when the feature is enabled or disabled.

When Supported and Enabled	When Not Supported or Disabled
	
(Zoomed-in view) 	(Zoomed-in view) 

Depth Peel Transparency

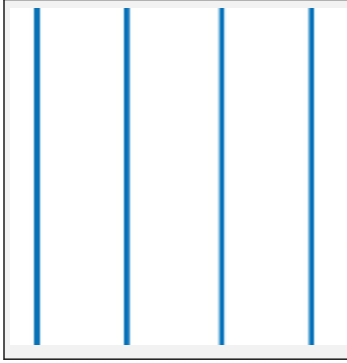
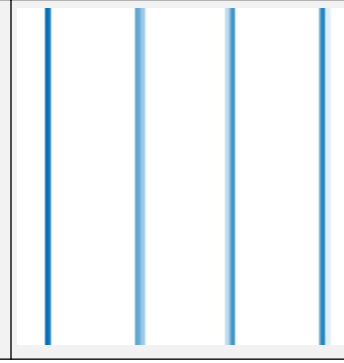
Depth peel transparency is a feature for correctly drawing semitransparent 3-D objects or plots that contain intersecting semitransparent objects. In the table, the left image shows the result of using transparency on a sphere when the depth peel transparency feature is supported. The right image shows the same sphere with unexpected shaded areas that occur when the feature is not supported.

When Supported	When Not Supported
	

Align Vertex Centers

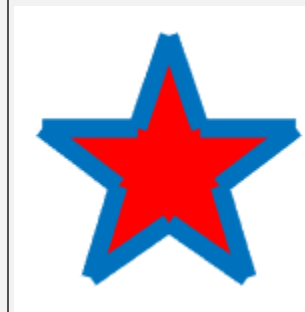
Align vertex centers is a feature for sharp vertical and horizontal lines. If graphics smoothing is enabled, horizontal and vertical lines might be uneven in thickness or color. The align vertex centers feature eliminates the uneven appearance. By default, the align vertex centers feature is not enabled. However, if your system supports this feature, then you can turn it on for objects that have an `AlignVertexCenters` property by setting the property to `'on'`.

This table shows the difference when the feature is enabled or disabled.

When Supported and Enabled	When Not Supported or Disabled
	

Hardware-Accelerated Markers

Hardware-accelerated markers take advantage of your graphics hardware for improved performance and quality. This table shows the difference when the feature is supported or not supported.

When Supported	When Not Supported
	

Tips

- Painters is an alternate rendering method for screen display and printing. For more information, see the `Renderer` property of the figure.
- By default, MATLAB tries to use hardware-accelerated graphics if your graphics hardware supports it. However, in some cases, MATLAB automatically switches to a software implementation if it detects one of these situations:

- You are using a graphics driver with known issues or graphics virtualization.
- A previous MATLAB session crashed due to a graphics issue. If the previous session was using software OpenGL and crashed, then subsequent sessions use a more stable version of software OpenGL that has fewer capabilities.
- Your system does not have the necessary graphics hardware.

See Also

`matlab (Linux)` | `matlab (Windows)` | `matlab (macOS)` | `opengl`

Topics

"System Requirements for Graphics"
"Resolving Low-Level Graphics Issues"

Introduced in R2019a

remove

Package: `matlab.settings`

Record removal of factory setting or group

Syntax

```
remove(path)
```

Description

`remove(path)` records the removal of a factory setting or settings group. Recording changes to the factory settings tree for each toolbox version ensures that users upgrading to a new version of a toolbox do not have backward incompatibility issues with their toolbox settings.

Examples

Record Removal of Setting

Create a settings file upgrader object and record the removal of a setting in the second version of a toolbox.

Create a settings file upgrader object for version 2 of `mytoolbox`.

```
upgrader = SettingsFileUpgrader('version2');
```

Record the removal of the setting `MySetting` from the `mytoolbox` factory settings group.

```
upgrader.remove('mytoolbox.MySetting');
```

Record Removal of Settings Group

Create a settings file upgrader object and record the removal of a settings group in the second version of a toolbox.

Create a settings file upgrader object for version 2 of `mytoolbox`.

```
upgrader = SettingsFileUpgrader('version2');
```

Record the removal of the settings group `mysettings` from the `mytoolbox` factory settings group.

```
upgrader.remove('mytoolbox.mysettings');
```

Input Arguments

`path` — Setting or group path
character vector | string scalar

Setting or group path, specified as a character vector or string scalar. Specify the full path of the setting or group, starting from the toolbox factory settings group and including the setting or group name.

Example: `upgrader.remove('mytoolbox.mysettings');`

See Also

`SettingsFileUpgrader` | `move`

Topics

"Create Factory Settings for Toolboxes"
"Access and Modify Settings"

Introduced in R2019b

read

Read entire TIFF image

Syntax

```
imageData = read(t)
[Y,Cb,Cr] = read(t)
```

Description

`imageData = read(t)` reads the image data from the current image file directory (IFD) in the TIFF file associated with the `Tiff` object `t`.

`[Y,Cb,Cr] = read(t)` reads the YCbCr component data from the current image file directory in the TIFF file. Use this syntax only with images that have a YCbCr photometric interpretation.

Depending upon the values of the `YCbCrSubSampling` tag, the size of the `Cb` and `Cr` components can differ from the `Y` component.

Examples

Read TIFF Image Data

Create a `Tiff` object and read data from the TIFF file.

```
t = Tiff('peppers_RGB_tiled.tif','r');
imageData = read(t);
```

Display the image.

```
imshow(imageData);
title('Peppers Image (RGB)')
```

Close the `Tiff` object.

```
close(t);
```

Read YCbCr TIFF Image Data

Create a `Tiff` object for a file, get image data, and display the image.

```
t = Tiff('peppers_YCbCr_tiled.tif','r');
[Y,Cb,Cr] = read(t);
```

Display the Y component of the image.

```
imshow(Y);
title('Peppers Image (Y Component)');
```

Close the `Tiff` object.

```
close(t);
```

Input Arguments

`t` — Tiff object
`Tiff` object

`Tiff` object representing a TIFF file. Use the `Tiff` function to create the object.

Output Arguments

`imageData` — Image data
numeric array

Image data, returned as a numeric array. For example, for an RGB image `imageData` is an `M`-by-`N`-by-`3` array. Where `M` and `N` are the height and width of the image, respectively.

`Y` — Luma component
numeric array

Luma component of the image data, returned as a two-dimensional numeric array.

`Cb` — Blue-difference chroma component
numeric array

Blue-difference chroma component of the image data, returned as a two-dimensional numeric array.

`Cr` — Red-difference chroma component
numeric array

Red-difference chroma component of the image data, returned as a two-dimensional numeric array.

See Also

`imread` | `write`

Topics

"Importing Images"
"Exporting to Images"

Introduced in R2009b

read

Read data from remote host over TCP/IP

Syntax

```
read(t)
read(t,size)
read(t,size,datatype)
```

Description

`read(t)` reads all available bytes of data from `tcpclient` object `t` connected to the remote host and returns the data. The number of values read is determined by the `BytesAvailable` property.

For any read or write operation, the data type is converted to `uint8` for the data transfer. It is then converted back to whatever data type you set if you specified another data type.

`read(t,size)` reads the specified number of values, `size`, from `tcpclient` object `t` connected to the remote host and returns the data. If `size` is greater than the object's `BytesAvailable` property, then the function waits until the specified amount of data is read or the `timeout` is reached.

`read(t,size,datatype)` reads the specified number of values, `size`, with the specified precision, `datatype`, from `tcpclient` object `t` connected to the remote host and returns the data. The `datatype` argument is a character vector of a standard MATLAB data type.

For any read or write operation, the data type is converted to `uint8` for the data transfer. It is then converted back to whatever data type you set if you specified another data type.

Examples

Read All Available Data from Host

Create a TCP/IP object called `t`, using the IP address shown, and Port of `4012`.

```
t = tcpclient('172.28.154.231', 4012)
```

```
t = 

   tcpclient with properties:

          Address: '172.28.154.231'
             Port: 4012
          Timeout: 10
   BytesAvailable: 0
```

Read all the bytes of data available.

```
read(t)
```

The `read` function used with no arguments reads all available bytes of data from `tcpclient` object `t` connected to the remote host and returns the data. The number of values read is determined by the `BytesAvailable` property, which is equal to the numbers of bytes available in the input buffer.

Close the connection between the TCP/IP client object and the remote host by clearing the object.

```
clear t
```

Specify the Size and Data Type to Read

Create a TCP/IP object called `t`, connecting to a TCP/IP echo server, with Port of `7`.

```
t = tcpclient('localhost', 7)
```

```
t = 

   tcpclient with properties:

          Address: 'local host'
             Port: 7
          Timeout: 10
   BytesAvailable: 0
```

Assign 10 bytes of data to the variable `data`.

```
data = (1:10)
```

Check the data.

```
whos data
```

```
Name     Size     Bytes     Class     Attributes

data     1x10        10     double
```

Write data to the echo server.

```
write(t, data)
```

Check that the data was written using the `BytesAvailable` property.

```
t.BytesAvailable
```

```
ans =

    80
```

For any read or write operation, the data type is converted to `uint8` for the data transfer. It is then converted back to whatever data type you set if you specified another data type. Since 1 `double` equals 8 `uint8` bytes, there are 80 bytes available.

Read 10 doubles from the server. The object name is always the first argument. The `size` argument must be the second argument, and `datatype` must be the third argument.

```
read(t, 10, 'double')
```

```
ans =

  1    2    3    4    5    6    7    8    9    10
```

Close the connection between the TCP/IP client object and the remote host by clearing the object.

```
clear t
```

Input Arguments

size — Number of bytes to read
numeric scalar

Number of bytes to read, specified as a numeric scalar. Size cannot be set to `inf`. If `size` is greater than the object's `BytesAvailable` property, the function waits until the specified amount of data is read. The first argument must be the object name, and the second argument is the size. The `size` argument is optional.

Example: `read(t, 5)`

Data Types: `double`

datatype — MATLAB data type
`'uint8'` (default) | `'int8'` | `'uint16'` | `'int16'` | `'uint32'` | `'int32'` | `'uint64'` | `'int64'` | `'single'` | `'double'`

MATLAB data type, specified as a character vector. Size cannot be set to `inf`. The `datatype` must be set to one of the 10 values shown above. The first argument must be the object name, the second argument is the size, and the third argument is the data type. The `size` and `datatype` arguments are optional.

For any read or write operation, the data type is converted to `uint8` for the data transfer. It is then converted back to whatever data type you set if you specified another data type.

Example: `read(t, 10, 'double')`

Data Types: `char`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Topics

“Create a TCP/IP Connection”
“Configure Properties for TCP/IP Communication”
“Write and Read Data over TCP/IP Interface”

Introduced in R2014b

readasync

(Not recommended) Read data asynchronously from device

Note This `serial` object function is not recommended. Use `serialport` object functions instead. See “Compatibility Considerations”.

Syntax

```
readasync(obj)
readasync(obj,size)
```

Description

`readasync(obj)` initiates an asynchronous read operation on the serial port object, `obj`.

`readasync(obj,size)` asynchronously reads, at most, the number of bytes given by `size`. If `size` is greater than the difference between the `InputBufferSize` property value and the `BytesAvailable` property value, an error is returned.

Examples

This example creates the serial port object `s` on a Windows platform. It connects `s` to a Tektronix TDS 210 oscilloscope, configures `s` to read data asynchronously only if `readasync` is issued, and configures the instrument to return the peak-to-peak value of the signal on channel 1.

```
s = serial('COM1');
fopen(s)
s.ReadAsyncMode = 'manual';
fprintf(s,'Measurement:Meas1:Source CH1')
fprintf(s,'Measurement:Meas1:Type Pk2Pk')
fprintf(s,'Measurement:Meas1:Value?')
```

Begin reading data asynchronously from the instrument using `readasync`. When the read operation is complete, return the data to the MATLAB workspace using `fscanf`.

```
readasync(s)
s.BytesAvailable
```

```
ans =
     15
```

```
out = fscanf(s)
```

```
out =
    2.0399999619E0
```

```
fclose(s)
```

Tips

Before you can read data, you must connect `obj` to the device with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

Only use `readasync` to configure the `ReadAsyncMode` property to `manual`. `readasync` is ignored if used when `ReadAsyncMode` is `continuous`.

The `TransferStatus` property indicates if an asynchronous read or write operation is in progress. You can write data while an asynchronous read is in progress because serial ports have separate read and write pins. You can stop asynchronous read and write operations with the `stopasync` function.

You can monitor the amount of data stored in the input buffer with the `BytesAvailable` property. Additionally, you can use the `BytesAvailableFcn` property to execute a callback function when the terminator or the specified amount of data is read.

Rules for Completing an Asynchronous Read Operation

An asynchronous read operation with `readasync` completes when one of these conditions is met:

- The terminator specified by the `Terminator` property is read.
- The time specified by the `Timeout` property passes.
- The specified number of bytes is read.
- The input buffer is filled (if `size` is not specified).

Because `readasync` checks for the terminator, this function can be slow. To increase speed, you might want to configure `ReadAsyncMode` to `continuous` and continuously return data to the input buffer as soon as it is available from the device.

Compatibility Considerations

`serial` object interface is not recommended

Not recommended starting in R2019b

Use of this function with a `serial` object is not recommended. To access a serial port device, use a `serialport` object with its functions and properties instead.

See “Transition Your Code to serialport Interface” for more information about using the recommended functionality.

See Also

Functions

`fopen` | `stopasync`

Introduced before R2006a

Tree Properties

Control appearance and behavior of tree

Description

Trees are UI components for presenting lists of items in a hierarchy within an app. The `uitree` function creates a tree and sets any required properties before displaying it. By changing tree property values, you can modify certain aspects of its appearance and behavior. Use dot notation to refer to a specific object and property:

```
uf = uifigure;
t = uitree(uf);
t.Multiselect = 'on';
```

Properties

Nodes

`SelectedNodes` — Selected nodes

`[]` (default) | `TreeNode` object | array of `TreeNode` objects

Selected nodes, specified as a `TreeNode` object or an array of `TreeNode` objects. Use this property to get or set the selected nodes in a tree.

To allow users to select multiple nodes, set the `Multiselect` property to `'on'`. MATLAB always returns `SelectedNodes` as a column vector when the tree has multiple selected nodes.

Font and Color

`FontName` — Font name

system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size
positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

`FontWeight` — Font weight
`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

`FontAngle` — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

`FontColor` — Font color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`BackgroundColor` — Background color

`[1 1 1]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Interactivity

`Visible` — Tree visibility

`'on'` (default) | `'off'`

Tree visibility, specified as `'on'` or `'off'`. When the `Visible` property is set to `'off'`, the tree is not visible, but you can query and set its properties.

`Multiselect` — Multiple node selection

`'off'` (default) | `'on'`

Multiple node selection, specified as `'off'` or `'on'`. Set this property to `'on'` to allow users to select multiple nodes simultaneously.

`Editable` — Node text editability

`'off'` (default) | `'on'`

Node text editability, specified as `'off'` or `'on'`. Set this property to `'on'` to allow the user to edit the node text at run time. The `Enable` property must also be set to `'on'` to make the text editable.

`Enable` — Operational state of tree

`'on'` (default) | `'off'`

Operational state of tree, specified as `'on'` or `'off'`. Set this property to `'off'` to make the tree and its nodes appear dim, indicating that the user cannot interact with the tree or its nodes.

`Tooltip` — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size
`[20 20 100 300]` (default) | `[left bottom width height]`

Location and size, specified as a four-element vector of the form `[left bottom width height]`. This table describes each element in the vector. All measurements are in pixel units.

Element	Description
`left`	Distance from the inner left edge of the parent container to the left edge of the bounding box that encloses the tree
`bottom`	Distance from the inner bottom edge of the parent container to the bottom edge of the bounding box that encloses the tree
`width`	Distance between the right and left edges of the bounding box
`height`	Distance between the top and bottom edges of the bounding box

`InnerPosition` — Location and size
`[20 20 100 300]` (default) | `[left bottom width height]`

Location and size, specified as a four-element vector of the form, `[left bottom width height]`. The values in the vector are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`OuterPosition` — Location and size
`[20 20 100 300]` (default) | `[left bottom width height]`

This property is read-only.

Location and size, returned as a four-element vector of the form, `[left bottom width height]`. The values in the vector are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is

not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a tree in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
t = uitree(g);
t.Layout.Row = 3;
t.Layout.Column = 2;
```

To make the tree span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this tree spans columns `2` through `3`:

```
t.Layout.Column = [2 3];
```

Callbacks

`SelectionChangedFcn` — Selection changed callback

`' '` (default) | function handle | cell array | character vector

Selection changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

Use this callback function to execute commands when the user selects a different node in the tree.

This callback function can access specific information about the user's interaction with the tree, such as the selected nodes. MATLAB passes this information in a `SelectedNodesChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.SelectedNodes` returns the selected `TreeNode` object or objects. The `SelectedNodesChangedData` object is not available to callback functions specified as character vectors.

The following table describes properties of the `SelectedNodesChangedData` object.

Property	Description
`SelectedNodes`	Most recently selected `TreeNode` object or objects
`PreviousSelectedNodes`	Previously selected `TreeNode` object or objects
`Source`	Component that executes the callback
`EventName`	`'SelectionChanged'`

For more information about writing callbacks, see “Write Callbacks in App Designer”.

`NodeExpandedFcn` — Node expanded callback
`''` (default) | function handle | cell array | character vector

Node expanded callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

Use this callback function to execute commands when the user expands a node in the tree.

This callback function can access specific information about the user’s interaction with the node. MATLAB passes this information in a `NodeExpandedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.Node` returns the `TreeNode` object that the user collapsed. The `NodeExpandedData` object is not available to callback functions specified as character vectors.

The following table describes properties of the `NodeExpandedData` object.

Property	Description
`Node`	`TreeNode` object that the user expanded
`Source`	Component that executes the callback.
`EventName`	`'NodeExpanded'`

For more information about writing callbacks, see “Write Callbacks in App Designer”.

`NodeCollapsedFcn` — Node collapsed callback

`''` (default) | function handle | cell array | character vector

Node collapsed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

Use this callback function to execute commands when the user collapses a node in the tree.

This callback function can access specific information about the user's interaction with the node. MATLAB passes this information in a `NodeCollapsedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.Node` returns the `TreeNode` object that the user collapsed. The `NodeCollapsedData` object is not available to callback functions specified as character vectors.

The following table describes properties of the `NodeCollapsedData` object.

Property	Description
`Node`	`TreeNode` object that the user collapsed
`Source`	Component that executes the callback.
`EventName`	`'NodeCollapsed'`

For more information about writing callbacks, see "Write Callbacks in App Designer".

`NodeTextChangedFcn` — Node text changed callback

`''` (default) | function handle | cell array | character vector

Node text changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

Use this callback function to execute commands when the user changes the text for a node in the tree.

This callback function can access specific information about the user's interaction with the tree node. MATLAB passes this information in a `NodeTextChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.PreviousText` returns the previous node text. The `NodeTextChangedData` object is not available to callback functions specified as character vectors.

The following table describes the properties of the `NodeTextChangedData` object.

Property	Description
`Node`	`TreeNode` object that has changed text
`Text`	New node text
`PreviousText`	Previous node text
`Source`	Component that executes the callback
`EventName`	`'NodeTextChanged'`

For more information about writing callbacks, see "Write Callbacks in App Designer".

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see "Write Callbacks in App Designer".

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`Children` — Children
empty `GraphicsPlaceholder` array (default) | array of `TreeNode` objects

Children, returned as an array of `TreeNode` objects.

You cannot add or remove children using the `Children` property, but you can use the property to view the list of children. The order of the children reflects the order of the

child nodes displayed on the screen. To add a child to this list, set the `Parent` property of the child component to be the `Tree` object.

To reorder the children, use the `move` function.

Objects with the `HandleVisibility` property set to `'off'` do not list in the `Children` property.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object
`'uitree'`

This property is read-only.

Type of graphics object, returned as `'uitree'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`appdesigner` | `scroll` | `uitree` | `uitreenode`

Introduced in R2017b

TreeNode Properties

Control appearance and behavior of tree node

Description

Tree nodes are the items listed in a tree hierarchy. The `uitreenode` function creates a tree node and sets any required properties before displaying it. By changing tree node property values, you can modify certain aspects of its appearance and behavior. Use dot notation to refer to a specific object and property.

```
uf = uifigure;
tree = uitree(uf);
node = uitreenode(tree);
node.Text = 'Item 1';
```

Properties

Node

`Text` — Node text
`'Tree Node'` (default) | character vector | string scalar

Node text, specified as a character vector or string scalar.

`NodeData` — Node data
`[]` (default) | array

Node data, specified as an array of any type. Specify `NodeData` to share node-relevant data within your app code.

`Icon` — Icon image file
`''` (default) | character vector | string scalar

Icon image file, specified as a character vector or string scalar. Specify a file name that is on the MATLAB path when the user runs the app. Alternatively, specify a full path to the image file.

Callbacks

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent object

`Tree` object | `TreeNode` object

Parent object, specified as a `Tree` or `TreeNode` object.

`Children` — Children

empty `GraphicsPlaceholder` array (default) | array of `TreeNode` objects

Children, returned as an array of `TreeNode` objects.

You cannot add or remove children using the `Children` property, but you can use the property to view the list of children. The order of the children reflects the order of the child nodes displayed on the screen. To add a child to this list, set the `Parent` property of the child component to be the `TreeNode` object.

To reorder the children, use the `move` function.

Objects with the `HandleVisibility` property set to `'off'` do not list in the `Children` property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uitreenode'`

This property is read-only.

Type of graphics object, returned as `'uitreenode'`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`appdesigner` | `uitree` | `uitreenode`

Introduced in R2017b

collapse

Package: `matlab.ui.container`

Collapse tree node

Syntax

```
collapse(parent)
collapse(parent,'all')
```

Description

`collapse(parent)` collapses the nodes of the parent tree or tree node. If `parent` is a `Tree` object, then the top-level nodes in the tree display in a collapsed state. Any children beyond the top level do not collapse. If parent is a `TreeNode` object, then the specified node displays in a collapsed state, but the children inside that node do not collapse.

`collapse(parent,'all')` collapses all child nodes of the parent tree or tree node.

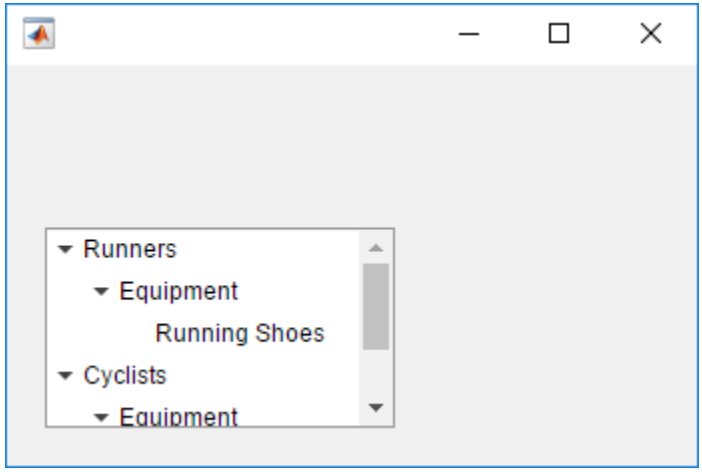
Examples

Collapse Top-Level Nodes

Create a tree that has three levels of nodes, and expand all of the nodes.

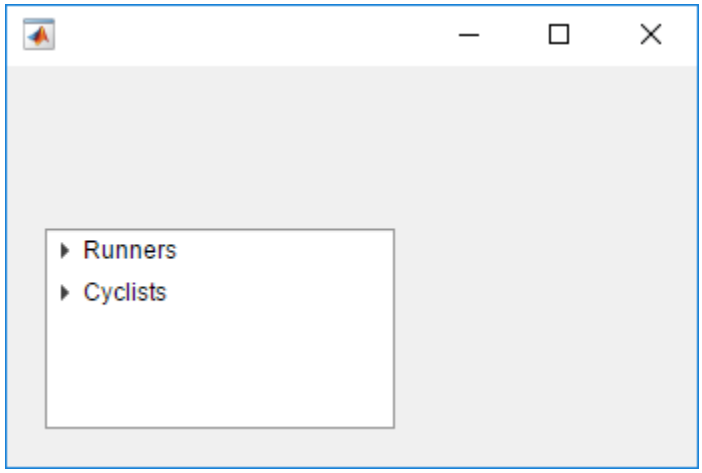
```
f = uifigure;
tree = uitree(f,'Position',[20 20 175 100]);

% First level nodes
category1 = uitreenode(tree,'Text','Runners');
category2 = uitreenode(tree,'Text','Cyclists');

% Second level nodes
equip1 = uitreenode(category1,'Text','Equipment');
```

```
equip2 = uitreenode(category2,'Text','Equipment');

% Third level nodes
shoes = uitreenode(equip1,'Text','Running Shoes');
bike = uitreenode(equip2,'Text','Bicycle');
helmet = uitreenode(equip2,'Text','Helmet');

% Expand all nodes
expand(tree','all');
```



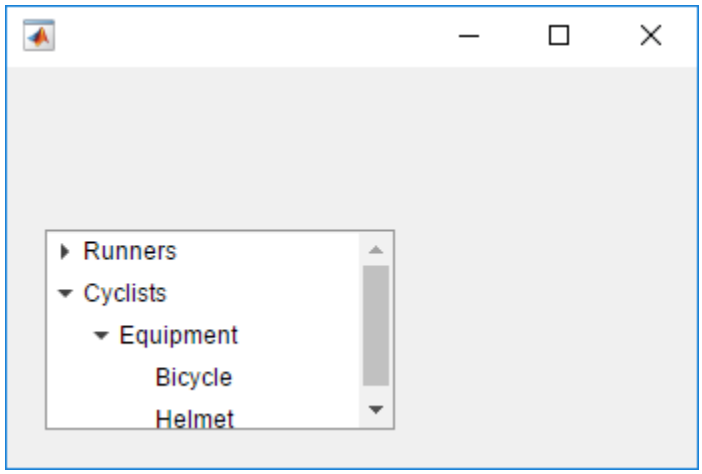
Collapse the top-level nodes.

```
collapse(tree)
```



Expand the `Cyclists` node.

```
expand(category2)
```



The `Equipment` child node is already expanded because the previous `collapse` command collapsed only the `Cyclists` node. If you want to collapse all the child nodes, specify the `'all'` option when you call the `collapse` function.

Input Arguments

`parent` — Parent tree or node
`Tree` object | `TreeNode` object

Parent tree or node, specified as a `Tree` object or `TreeNode` object.

See Also

`appdesigner` | `expand` | `move` | `scroll` | `uitree` | `uitreenode`

Introduced in R2017b

expand

Package: `matlab.ui.container`

Expand tree node

Syntax

```
expand(parent)
expand(parent,'all')
```

Description

`expand(parent)` expands the nodes of a tree or tree node. If `parent` is a `Tree` object, then the top-level nodes display in an expanded state. If `parent` is a `TreeNode` object, then that node displays in an expanded state.

`expand(parent,'all')` expands all nodes of a tree or tree node.

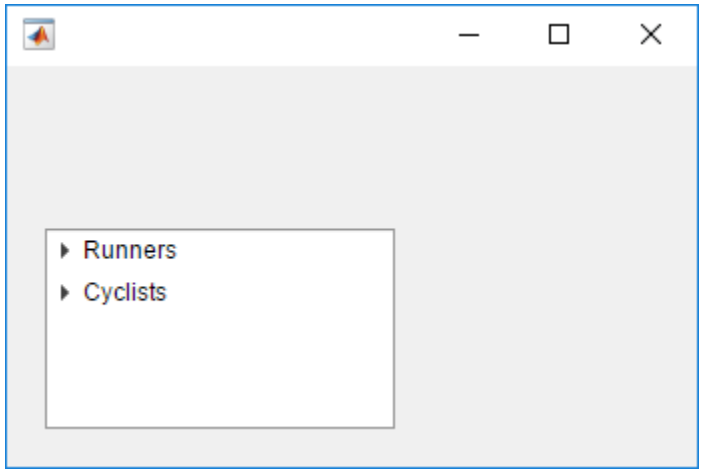
Examples

Expand Nodes in a Tree

Create a tree that has three levels of nodes. By default, the nodes display in a collapsed state.

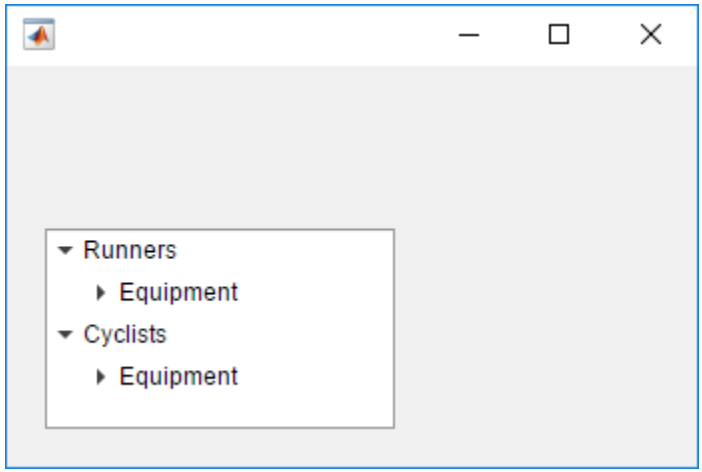
```
f = uifigure;
tree = uitree(f,'Position',[20 20 175 100]);

% First level nodes
category1 = uitreenode(tree,'Text','Runners');
category2 = uitreenode(tree,'Text','Cyclists');

% Second level nodes
equip1 = uitreenode(category1,'Text','Equipment');
```

```
equip2 = uitreenode(category2,'Text','Equipment');

% Third level nodes
shoes = uitreenode(equip1,'Text','Running Shoes');
bike = uitreenode(equip2,'Text','Bicycle');
helmet = uitreenode(equip2,'Text','Helmet');
```



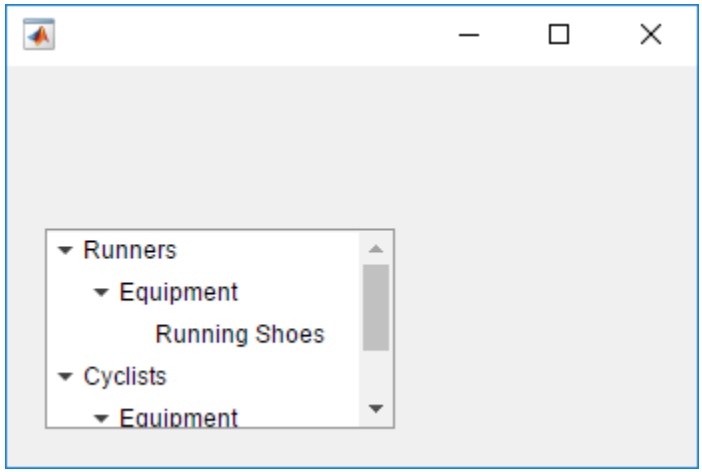
Expand the top-level nodes.

```
expand(tree)
```



Expand all the nodes in the tree.

```
expand(tree,'all')
```



Input Arguments

parent — Parent tree or node
Tree object | TreeNode object

Parent tree or node, specified as a Tree object or TreeNode object.

See Also

appdesigner | collapse | move | scroll | uitree | uitreenode

Introduced in R2017b

move

Package: `matlab.ui.container`

Move tree node

Syntax

```
move(targetnode,siblingnode)
move(targetnode,siblingnode,location)
```

Description

`move(targetnode,siblingnode)` moves the target node after the specified sibling node.

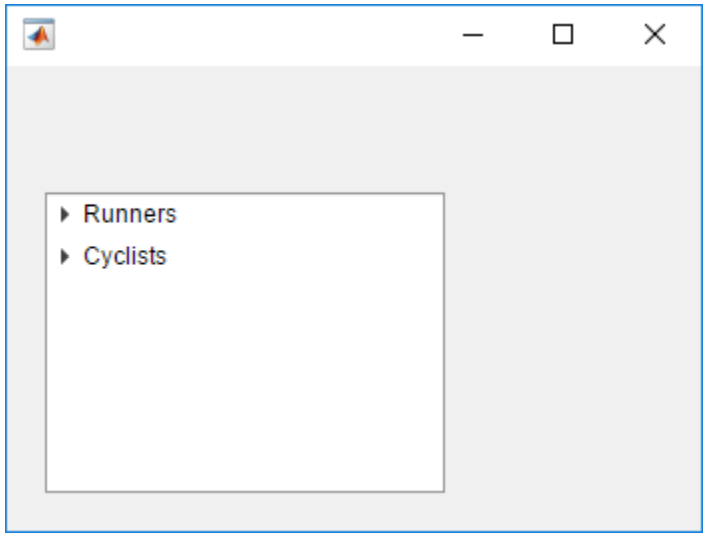
`move(targetnode,siblingnode,location)` moves the target node after or before the specified sibling node. Specify location as `'after'` or `'before'`.

Examples

Change the Order of Two Nodes

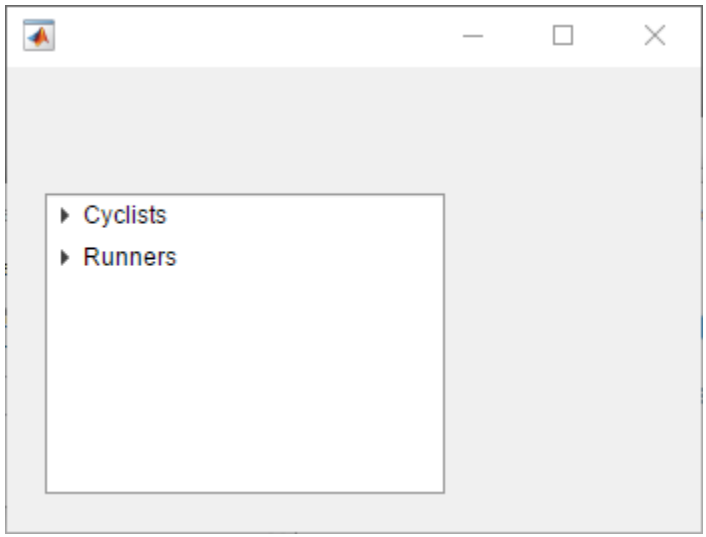
Create a tree containing two top-level nodes that each contain a child node.

```
f = uifigure;
tree = uitree(f,'Position',[20 20 200 150]);
node1 = uitreenode(tree,'Text','Runners');
node2 = uitreenode(tree,'Text','Cyclists');
runner = uitreenode(node1,'Text','Joe');
cyclist = uitreenode(node2,'Text','Rajeev');
```



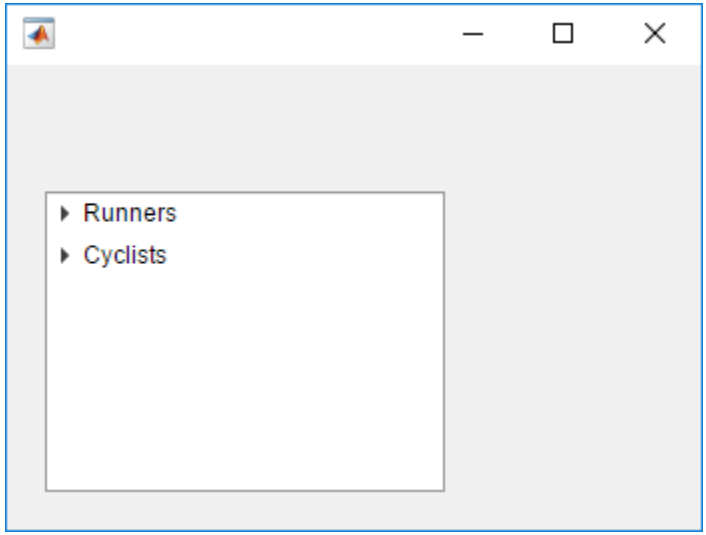
Move `Cyclists` before `Runners`.

```
move(node2,node1,'before');
```



Move `Cyclists` after `Runners`. Since the default value of `location` is `'after'`, you can omit that argument.

```
move(node2,node1);
```



Input Arguments

`targetnode` — Target node
`TreeNode` object

Target node, specified as a `TreeNode` object.

`siblingnode` — Sibling node
`TreeNode` object

Sibling node, specified as a `TreeNode` object.

`location` — Target location
`'after'` (default) | `'before'`

Target location, specified as `'after'` or `'before'`.

See Also

appdesigner | collapse | expand | scroll | uitree | uitreenode

Introduced in R2017b

uiconfirm

Create confirmation dialog box

Syntax

```
uiconfirm(f,message,title)
uiconfirm(f,message,title,Name,Value)
selection = uiconfirm( ___ )
```

Description

`uiconfirm(f,message,title)` displays a modal in-app confirmation dialog box in target figure `f`. The target figure must be created with the `uifigure` function. This syntax displays two options for the user to select, **OK** and **Cancel**. The figure behind the dialog box is not accessible while the dialog box is displaying, but the MATLAB command prompt is accessible.

`uiconfirm(f,message,title,Name,Value)` displays the confirmation dialog box with one or more `Name,Value` pair arguments that customize the appearance and behavior of the dialog box. For example, you can specify a custom set of options in the dialog box instead of the default, **OK** and **Cancel**.

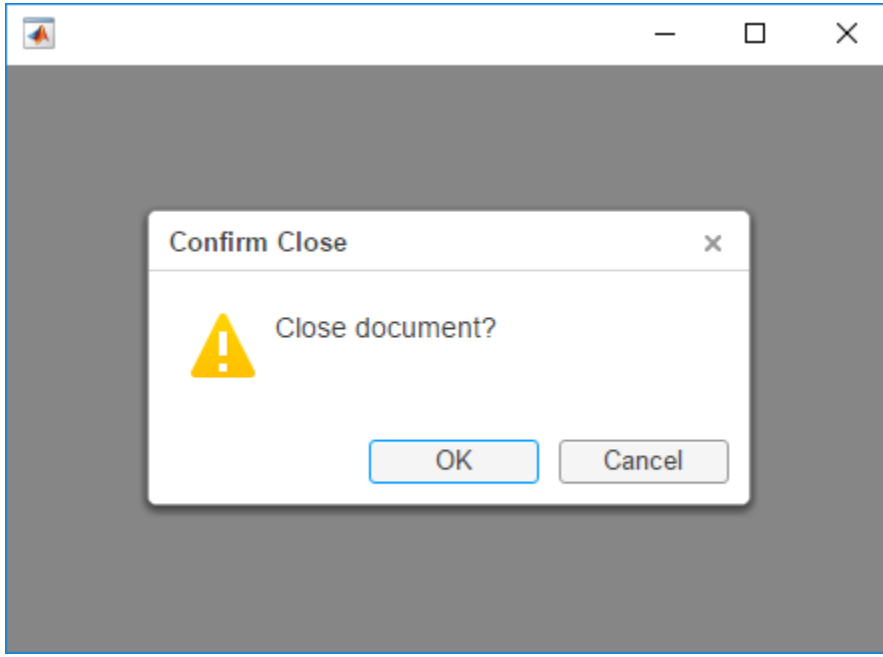
`selection = uiconfirm( ___ )` returns the user's selection as a character vector. Specify the `selection` output argument with any of the previous syntaxes. When you use this syntax, the MATLAB command prompt is not accessible while the dialog box is displaying.

Examples

Specify Different Icon

Create a dialog box that displays the warning icon instead of the default question icon.

```
f = uifigure;
selection = uiconfirm(f,'Close document?','Confirm Close',...
                        'Icon','warning');
```

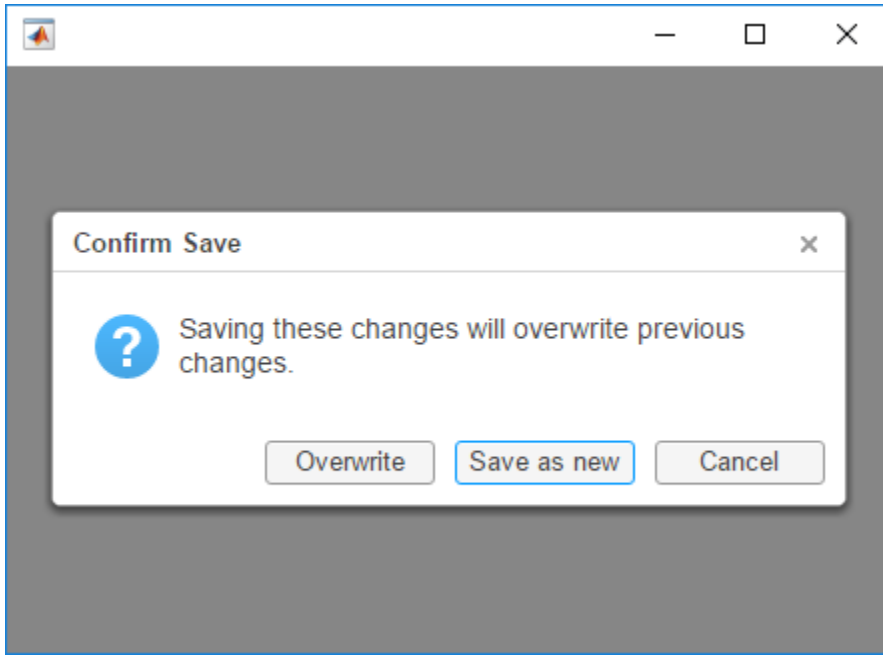


When the user selects an option, `uiconfirm` returns that choice as a character vector.

Specify Custom Options

Create a confirmation dialog containing three options: **Overwrite**, **Save as new**, and **Cancel**. Specify **Save as new** as the default option, and specify **Cancel** as the option that maps to the cancel behavior.

```
f = uifigure;
msg = 'Saving these changes will overwrite previous changes.';
title = 'Confirm Save';
selection = uiconfirm(f,msg,title,...
           'Options',{'Overwrite','Save as new','Cancel'},...
           'DefaultOption',2,'CancelOption',3);
```



When the user selects an option, `uiconfirm` returns their selection as a character vector.

Define `CloseFcn` Callback

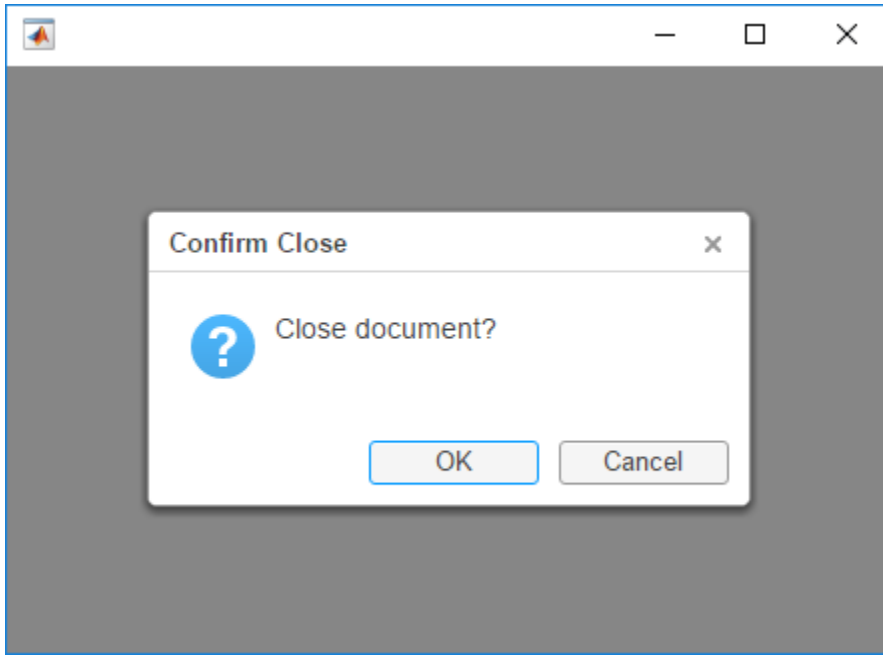
The `CloseFcn` name-value pair argument is useful for executing specific tasks when the dialog box closes.

In the MATLAB Editor, create a new function called `mycallback.m` that contains the following code. This callback function displays the `SelectedOption` field in a `struct` called `event`. MATLAB automatically passes this `struct` as the second argument to the callback function.

```
function mycallback(src,event)
   display(event.SelectedOption);
end
```

In the MATLAB Command Window, execute the following code to create a confirmation dialog box that specifies `mycallback` as the value for `CloseFcn`.

```
f = uifigure;
uiconfirm(f,'Close document?','Confirm Close',...
            'CloseFcn',@mycallback);
```



When the user selects an option, the value of `SelectedOption` displays in the Command Window.

Define `CloseFcn` Callback in App Designer

To create a confirmation dialog box in App Designer that has a `CloseFcn` callback, write the callback as a private function in App Designer.

Start by selecting **Code View**. Then create a private function by selecting **Function** > **Private Function**.

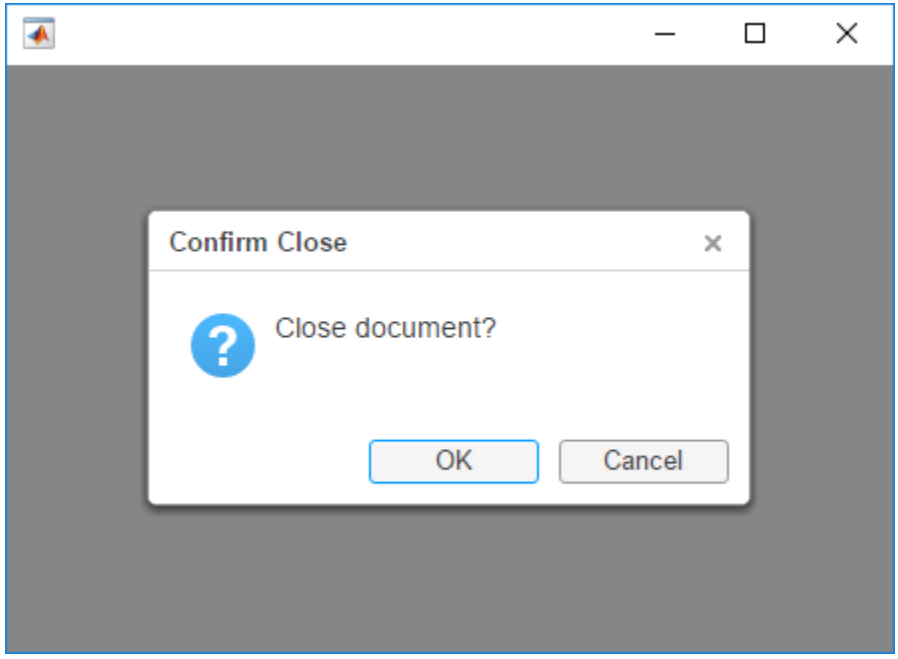
Next, write the private function so that it matches this code:

```
function mycallback(app,src,event)
   display(event.SelectedOption);
end
```

Add this command to the callback function that you want to display the dialog box. In this case, the target figure is `app.UIFigure`, which is the default name for the figure in App Designer.

```
uiconfirm(app.UIFigure,'Close document?','Confirm Close',...
            'CloseFcn',@(src,event)mycallback(app,src,event));
```

Save and run your app. When the user triggers the callback that creates the dialog box, the dialog box displays in the app.



Input Arguments

f — Target figure
`Figure` object

Target figure, specified as a `Figure` object. The figure must be created with the `uifigure` function.

`message` — Message to display
character vector | cell array of character vectors | string array

Message to display, specified as a character vector, cell array of character vectors, or string array. Specify a cell array or string array when your message has multiple lines of text. Each element in the array corresponds to a different line of text.

`title` — Dialog box title
character vector | string scalar

Dialog box title, specified as a character vector or string scalar.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `selection = uiconfirm(f,message,title,'Options',{'Save','Delete','Quit'})` specifies three custom options for the dialog box.

`Options` — Custom options
`{'OK','Cancel'}` (default) | cell array of character vectors | string array

Custom options, specified as a cell array of character vectors or a string array.

`Icon` — Icon
`'question'` (default) | predefined icon | custom icon

Icon, specified as a predefined icon or a custom icon.

Predefined Icon

This table lists the values for the predefined icons. For example, to show the check mark icon, specify the name-value pair `'Icon','success'`.

Value	Icon
`'question'` (default)	

Value	Icon
`'info'`	
`'success'`	
`'warning'`	
`'error'`	
`''`	No icon displays.

Custom Icon

Specify a custom icon as one of these values:

- A character vector that specifies the file name of an SVG, JPEG, GIF, or PNG image that is on the MATLAB path. Alternatively, you can specify a full path to the image file.
- A truecolor image array. See “Image Types” for more information.

`DefaultOption` — Default option

1 (default) | character vector | string scalar | whole number

Default option, specified as a character vector, string scalar, or a whole number. The default option corresponds to the button in the dialog box that has focus by default.

When you specify a character vector or string scalar, it must match an element in the `Options` array. However, if you are calling `uiconfirm` without the `Options` argument, then `DefaultOption` must be `'OK'` or `'Cancel'`.

When you specify a whole number, it must be in the range [1, n], where n is the length of the `Options` array. If you are calling `uiconfirm` without the `Options` argument, then `DefaultOption` must be `1` or `2`.

`CancelOption` — Cancel option

2 (default) | character vector | string scalar | whole number

Cancel option, specified as a character vector, string scalar, or a whole number. The cancel option specifies which option maps to cancel actions in the dialog box.

When you specify a character vector or string scalar, it must match an element in the `Options` array. However, if you are calling `uiconfirm` without the `Options` argument, then `CancelOption` must be `'OK'` or `'Cancel'`.

When you specify a whole number, it must be in the range [1, n], where n is the length of the `Options` array. If you are calling `uiconfirm` without the `Options` argument, then `CancelOption` must be `1` or `2`.

`CloseFcn` — Close callback function
`''` (default) | function handle | cell array | character vector

Close callback function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback is useful for executing specific tasks when the dialog box closes.

When you specify `CloseFcn` as a function handle (or cell array containing a function handle), MATLAB passes a `struct` containing event data as an input argument to the callback function. This `struct` contains the fields described in the following table.

Structure Field	Value
`Source`	`Figure` object associated with the dialog box.
`EventName`	`'ConfirmDialogClosed'`
`DialogTitle`	Title of the dialog box.
`SelectedOptionIndex`	Index of the selected option. For `n` options, the index can be any whole number from `1` to `n`.
`SelectedOption`	Button label for the selected option, returned as a character vector.

See Also

`questdlg` | `uialert` | `uifigure` | `uiprogressdlg`

Introduced in R2017b

uitree

Create tree component

Syntax

```
t = uitree
t = uitree(Name,Value)
t = uitree(parent)
t = uitree(parent,Name,Value)
```

Description

`t = uitree` creates a tree in a new figure window and returns the `Tree` object. MATLAB calls the `uifigure` function to create the figure.

`t = uitree(Name,Value)` specifies `Tree` property values using one or more `Name,Value` pair arguments.

`t = uitree(parent)` creates a tree in the specified parent container. The parent can be a `Figure` created using the `uifigure` function, or one of its child containers.

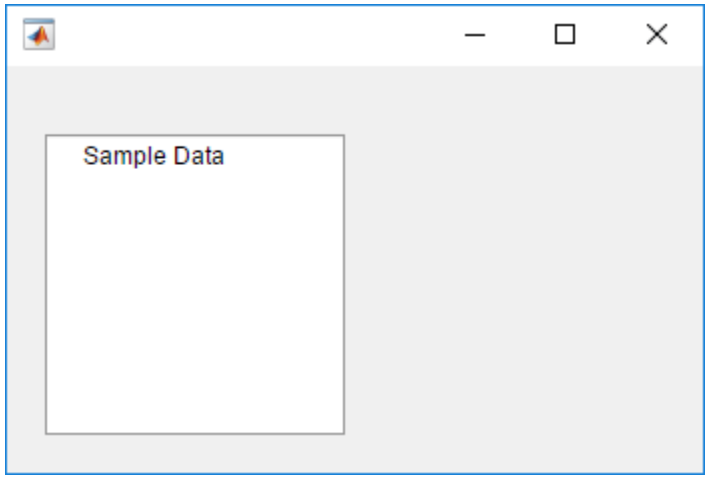
`t = uitree(parent,Name,Value)` creates the tree in the specified container and sets one or more `Tree` property values.

Examples

Tree with One Node

Create a tree that contains one node called `Sample Data`.

```
t = uitree('Position',[20 20 150 150]);
node = uitreenode(t,'Text','Sample Data');
```



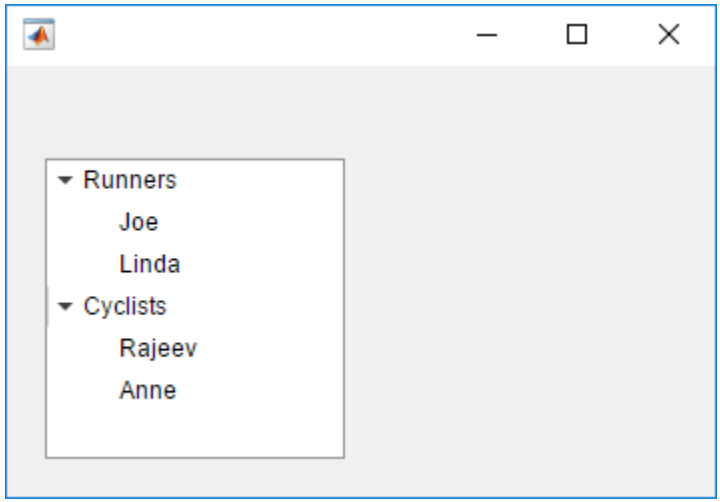
Tree with Nested Nodes and Callback Function

Create a program file called `mytreeapp.m` that contains the following commands to create a tree, a set of nested tree nodes, and a callback function for the tree. The `SelectionChangedFcn` property specifies the function to execute when the user clicks a node in the tree.

```
function mytreeapp
    f = uifigure;
    t = uitree(f,'Position',[20 20 150 150]);

    % Assign Tree callback in response to node selection
    t.SelectionChangedFcn = @nodechange;

    % First level nodes
    category1 = uitreenode(t,'Text','Runners','NodeData',[]);
    category2 = uitreenode(t,'Text','Cyclists','NodeData',[]);

    % Second level nodes.
    % Node data is age (y), height (m), weight (kg)
    p1 = uitreenode(category1,'Text','Joe','NodeData',[40 1.67 58] );
    p2 = uitreenode(category1,'Text','Linda','NodeData',[49 1.83 90]);
    p3 = uitreenode(category2,'Text','Rajeev','NodeData',[25 1.47 53]);
    p4 = uitreenode(category2,'Text','Anne','NodeData',[88 1.92 100]);
```

```
    % Expand the tree
    expand(t);

    function nodechange(src,event)
        node = event.SelectedNodes;
        display(node.NodeData);
    end
end
```

When the user runs `mytreeapp` and clicks a node in the tree, MATLAB displays the `NodeData` for that node.



Input Arguments

`parent` — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If you do not specify a parent container, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments as `Name1,Value1,...,NameN,ValueN`.

The properties listed here are a subset. For a full list, see Tree.

Example: `t = uitree('Position',[100 100 150 150])` creates a tree with a specific location and size.

`SelectedNodes` — Selected nodes
`[]` (default) | `TreeNode` object | array of `TreeNode` objects

Selected nodes, specified as a `TreeNode` object or an array of `TreeNode` objects. Use this property to get or set the selected nodes in a tree.

To allow users to select multiple nodes, set the `Multiselect` property to `'on'`. MATLAB always returns `SelectedNodes` as a column vector when the tree has multiple selected nodes.

`Multiselect` — Multiple node selection
`'off'` (default) | `'on'`

Multiple node selection, specified as `'off'` or `'on'`. Set this property to `'on'` to allow users to select multiple nodes simultaneously.

`SelectionChangedFcn` — Selection changed callback
`''` (default) | function handle | cell array | character vector

Selection changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

Use this callback function to execute commands when the user selects a different node in the tree.

This callback function can access specific information about the user's interaction with the tree, such as the selected nodes. MATLAB passes this information in a `SelectedNodesChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.SelectedNodes` returns the selected `TreeNode` object or objects. The `SelectedNodesChangedData` object is not available to callback functions specified as character vectors.

The following table describes properties of the `SelectedNodesChangedData` object.

Property	Description
`SelectedNodes`	Most recently selected `TreeNode` object or objects
`PreviousSelectedNodes`	Previously selected `TreeNode` object or objects
`Source`	Component that executes the callback
`EventName`	`'SelectionChanged'`

For more information about writing callbacks, see "Write Callbacks in App Designer".

`Position` — Location and size

`[20 20 100 300]` (default) | `[left bottom width height]`

Location and size, specified as a four-element vector of the form `[left bottom width height]`. This table describes each element in the vector. All measurements are in pixel units.

Element	Description
`left`	Distance from the inner left edge of the parent container to the left edge of the bounding box that encloses the tree
`bottom`	Distance from the inner bottom edge of the parent container to the bottom edge of the bounding box that encloses the tree
`width`	Distance between the right and left edges of the bounding box
`height`	Distance between the top and bottom edges of the bounding box

See Also

Functions
`appdesigner` | `scroll` | `uitreenode`

Properties
Tree

Topics

"App That Displays Data in a Hierarchy Using Tree"

Introduced in R2017b

uitreenode

Create tree node component

Syntax

```
node = uitreenode
node = uitreenode(parent)
node = uitreenode(parent,sibling)
node = uitreenode(parent,sibling,location)
node = uitreenode( ___ ,Name,Value)
```

Description

`node = uitreenode` creates a tree node UI component inside a tree within a new figure window and returns the `TreeNode` object. MATLAB calls the `uifigure` function to create the figure.

`node = uitreenode(parent)` creates a tree node in the specified parent container. The parent container can be a `Tree` or `TreeNode` object.

`node = uitreenode(parent,sibling)` creates a tree node in the specified parent container, after the specified sibling node.

`node = uitreenode(parent,sibling,location)` creates a tree node, and places it after or before the sibling node. Specify `location` as `'after'` or `'before'`.

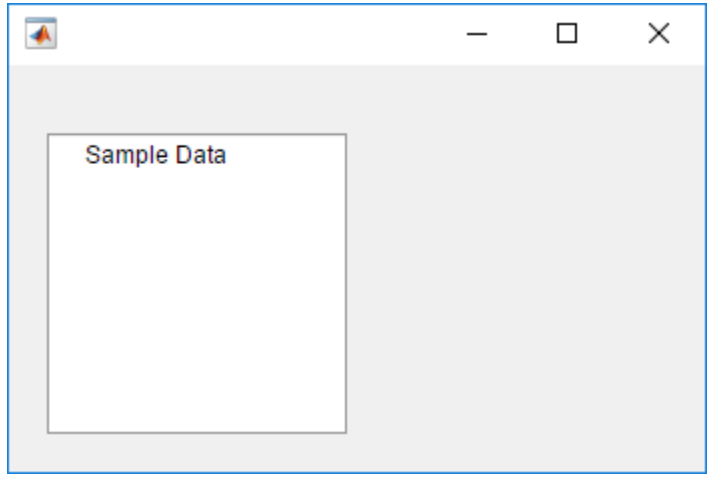
`node = uitreenode( ___ ,Name,Value)` specifies `TreeNode` property values using one or more `Name,Value` pair arguments. Specify `Name,Value` as the last set of arguments when you use any of the previous syntaxes.

Examples

One Node in a Tree

Create a tree that contains one node called `Sample Data`.

```
t = uitree('Position',[20 20 150 150]);
node = uitreenode(t,'Text','Sample Data');
```



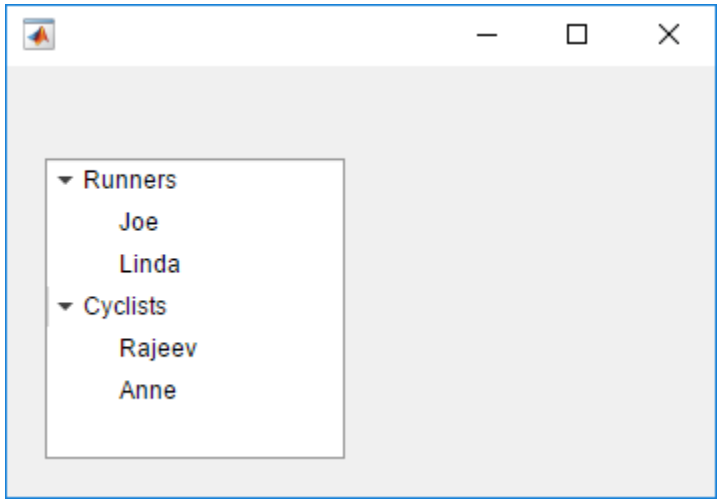
Tree with Nested Nodes

Create a program file called `mytreeapp.m` that contains the following commands to create a tree, a set of nested tree nodes, and a callback function for the tree. The `SelectionChangedFcn` property specifies the function to execute when the user clicks a node in the tree.

```
function mytreeapp
    f = uifigure;
    t = uitree(f,'Position',[20 20 150 150]);

    % Assign Tree callback in response to node selection
    t.SelectionChangedFcn = @nodechange;

    % First level nodes
    category1 = uitreenode(t,'Text','Runners','NodeData',[]);
    category2 = uitreenode(t,'Text','Cyclists','NodeData',[]);
```

```
    % Second level nodes.
    % Node data is age (y), height (m), weight (kg)
    p1 = uitreenode(category1,'Text','Joe','NodeData',[40 1.67 58] );
    p2 = uitreenode(category1,'Text','Linda','NodeData',[49 1.83 90]);
    p3 = uitreenode(category2,'Text','Rajeev','NodeData',[25 1.47 53]);
    p4 = uitreenode(category2,'Text','Anne','NodeData',[88 1.92 100]);

    % Expand the tree
    expand(t);

    function nodechange(src,event)
        node = event.SelectedNodes;
        display(node.NodeData);
    end
end
```

When the user runs `mytreeapp` and clicks a node in the tree, MATLAB displays the `NodeData` for that node.



Input Arguments

`parent` — Parent object

`Tree` object | `TreeNode` object

Parent object, specified as a `Tree` or `TreeNode` object.

`sibling` — Sibling node
`TreeNode`

Sibling node, specified as a `TreeNode` object.

`location` — Location relative to sibling
`'after'` (default) | `'before'`

Location of the node relative to its sibling, specified as a `'after'` or `'before'`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments as `Name1,Value1,...,NameN,ValueN`.

The properties listed here are a subset. For a full list, see TreeNode.

Example: `node = uitreenode(t,'Text','Measurements')` creates a tree node with the label, `'Measurements'`.

`Text` — Node text
`'Tree Node'` (default) | character vector | string scalar

Node text, specified as a character vector or string scalar.

`Icon` — Icon image file
`''` (default) | character vector | string scalar

Icon image file, specified as a character vector or string scalar. Specify a file name that is on the MATLAB path when the user runs the app. Alternatively, specify a full path to the image file.

`NodeData` — Node data
`[]` (default) | array

Node data, specified as an array of any type. Specify `NodeData` to share node-relevant data within your app code.

See Also

Functions
`collapse` | `expand` | `move` | `scroll` | `uitree`

Properties
TreeNode

Topics
"App That Displays Data in a Hierarchy Using Tree"

Introduced in R2017b

readEncodedStrip

Read data from specified strip

Syntax

```
stripData = readEncodedStrip(t,stripNumber)
[Y,Cb,Cr] = readEncodedStrip(t,stripNumber)
```

Description

`stripData = readEncodedStrip(t,stripNumber)` returns the image data from the strip specified by `stripNumber` from the TIFF file represented by the `Tiff` object `t`.

The `readEncodedStrip` function trims the area in a strip that falls outside of the `ImageLength` boundary. Therefore, image data from a strip at the bottom edge of the image can have different dimensions.

`[Y,Cb,Cr] = readEncodedStrip(t,stripNumber)` returns the YCbCr components of the strip data specified by `stripNumber` from the TIFF file represented by the `Tiff` object `t`. Use this syntax only with images that have a YCbCr photometric interpretation.

Depending upon the values of the `YCbCrSubSampling` tag, the size of the `Cb` component can differ from the `Y` component.

Examples

Read Strip of Data from TIFF Image

Read a strip of image data from a TIFF file that contains an image with a stripped layout.

```
t = Tiff('peppers_RGB_stripped.tif','r');
```

Determine the number of strips and the length of the strip in the image.

```
numberOfStrips(t)
```

```
ans = 6
```

```
getTag(t,'RowsPerStrip')
```

```
ans = 35
```

Read and display the third strip of the image. The `readEncodedStrip` function trims the area in a strip that falls outside of the `ImageLength` boundary. Therefore, image data from a strip at the bottom edge of the image can have different dimensions.

```
strip = readEncodedStrip(t,3);
imshow(strip);
title('3^{rd} Strip Peppers Image');
```

Close the `Tiff` object.

```
close(t);
```

Read Strip of Data from YCbCr TIFF Image

Read a strip of image data from a YCbCr TIFF file that contains an image with a stripped layout.

```
t = Tiff('peppers_YCbCr_stripped.tif','r');
```

Determine the number of strips and the length of each strip in the image.

```
numberOfStrips(t)
```

```
ans = 6
```

```
getTag(t,'RowsPerStrip')
```

```
ans = 35
```

Read and display the Y component of the third strip of the image. The `readEncodedStrip` function trims the area in a strip that falls outside of the `ImageLength` boundary. Therefore, image data from a strip at the bottom edge of the image can have different dimensions.

```
[Y,Cb,Cr] = readEncodedStrip(t,3);
imshow(Y);
title('3^{rd} Strip of Peppers Image (YCbCr)')
```

Close the `Tiff` object.

```
close(t);
```

Input Arguments

`t` — Tiff object
`Tiff` object

`Tiff` object representing a TIFF file. Use the `Tiff` function to create the object.

`stripNumber` — Strip number
positive integer

Strip number, specified as a positive integer. Strip numbers are one-based numbers.

Example: 15

Data Types: `double`

Output Arguments

`stripData` — Strip data
numeric array

Strip data, returned as a numeric array.

`Y` — Luma component
numeric array

Luma component of the strip data, returned as a two-dimensional numeric array.

`Cb` — Blue-difference chroma component
numeric array

Blue-difference chroma component of the strip data, returned as a two-dimensional numeric array.

`Cr` — Red-difference chroma component
numeric array

Red-difference chroma component of the strip data, returned as a two-dimensional numeric array.

Algorithms

References

This function corresponds to the `TIFFReadEncodedStrip` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff` | `isTiled` | `readEncodedTile`

Topics

"Importing Images"

“Exporting to Images”

Introduced in R2009b

readEncodedTile

Read data from specified tile

Syntax

```
tileData = readEncodedTile(t,tileNumber)
[Y,Cb,Cr] = readEncodedTile(t,tileNumber)
```

Description

`tileData = readEncodedTile(t,tileNumber)` returns image data contained in the tile specified by `tileNumber` from the TIFF file represented by the `Tiff` object `t`.

The `readEncodedTile` function trims the area in a tile that is outside of the `ImageLength` and `ImageWidth` boundaries. Therefore, image data from tiles that occur on the right edge or the bottom edge of the image can have different dimensions.

`[Y,Cb,Cr] = readEncodedTile(t,tileNumber)` returns the YCbCr components of the image data contained in the tile specified by `tileNumber` from the TIFF file represented by the `Tiff` object `t`. Use this syntax only with images that have a YCbCr photometric interpretation.

Depending upon the values of the `YCbCrSubSampling` tag, the size of the `Cb` component can differ from the `Y` component.

Examples

Read Tile of Data from TIFF Image

Read a tile of image data from a TIFF file that contains an image with a tiled layout.

```
t = Tiff('peppers_RGB_tiled.tif','r');
```

Determine the number of tiles and the tile size in the image.

```
nTiles = numberOfTiles(t)
```

```
nTiles = 36
```

```
tileSize = [getTag(t,'TileLength') getTag(t,'TileWidth')]
```

```
tileSize = 1×2

    32    48
```

Read and display the 19th tile of the image. The `readEncodedTile` function trims the area in a tile that is outside of the `ImageLength` and `ImageWidth` boundaries. Therefore, image data from tiles that occur on the right edge or the bottom edge of the image can have different dimensions.

```
tile = readEncodedTile(t,19);
imshow(tile,'InitialMagnification','fit'); % Magnify for display
title('19^{th} Tile of Peppers Image (RGB)');
```

Close the `Tiff` object.

```
close(t);
```

Read Tile from YCbCr TIFF Image

Read a tile of image data from a YCbCr TIFF file that contains an image with a tiled layout.

```
t = Tiff('peppers_YCbCr_tiled.tif','r');
```

Determine the number of tiles and the tile size in the image.

```
nTiles = numberOfTiles(t)
```

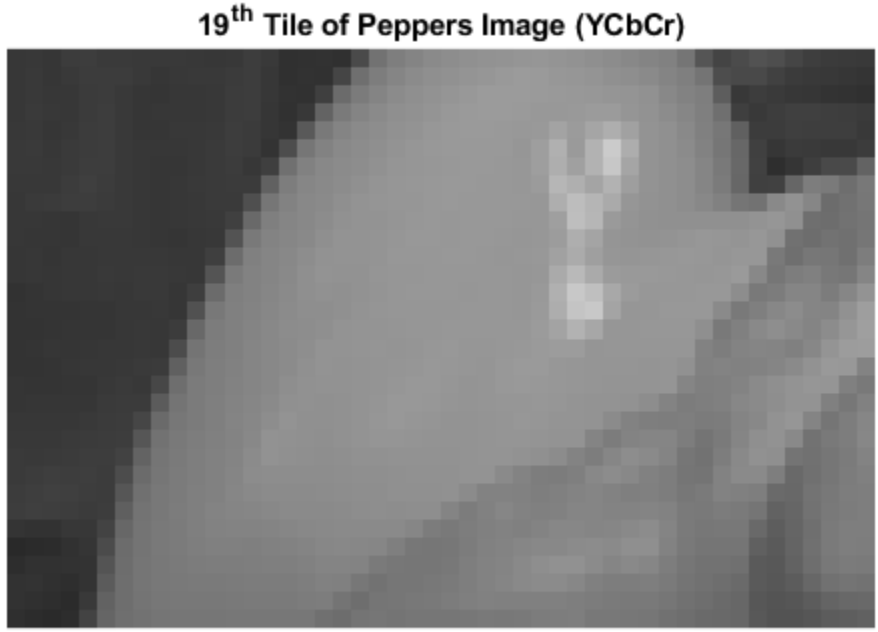
```
nTiles = 36
```

```
tileSize = [getTag(t,'TileLength') getTag(t,'TileWidth')]
```

```
tileSize = 1×2

    32    48
```

Read and display the Y component of the 19th tile of the image. The `readEncodedTile` function trims the area in a tile that is outside of the `ImageLength` and `ImageWidth` boundaries. Therefore, image data from tiles that occur on the right edge or the bottom edge of the image can have different dimensions.

```
[Y,Cb,Cr] =  readEncodedTile(t,19);
imshow(Y,'InitialMagnification','fit') % Magnify for display
title('19^{th} Tile of Peppers Image (YCbCr)')
```



Close the `Tiff` object.

```
close(t);
```

Input Arguments

`t` — Tiff object
`Tiff` object

`Tiff` object representing a TIFF file. Use the `Tiff` function to create the object.

`tileNumber` — Tile number
positive integer

Tile number, specified as a positive integer. Tile numbers are one-based numbers.

Example: 15

Data Types: `double`

Output Arguments

`tileData` — Tile data
numeric array

Tile data, returned as a numeric array.

`Y` — Luma component
numeric array

Luma component of the tile data, returned as a two-dimensional numeric array.

`Cb` — Blue-difference chroma component
numeric array

Blue-difference chroma component of the tile data, returned as a two-dimensional numeric array.

`Cr` — Red-difference chroma component
numeric array

Red-difference chroma component of the tile data, returned as a two-dimensional numeric array.

Algorithms

References

This function corresponds to the `TIFFReadEncodedTile` function in the LibTIFF C API. To use this function, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff` | `isTiled` | `readEncodedStrip`

Topics

"Importing Images"
"Exporting to Images"

Introduced in R2009b

readRGBAImage

Read image using RGBA interface

Syntax

```
[RGB,alpha] = readRGBAImage(t)
```

Description

`[RGB,alpha] = readRGBAImage(t)` reads the image and alpha matting data from the TIFF file associated with the `Tiff` object `t` using the RGBA interface.

Examples

Get Image Data and Associated Alpha Matting

Get all image data and its associated alpha matting from a TIFF file.

Create a `Tiff` object for an image file.

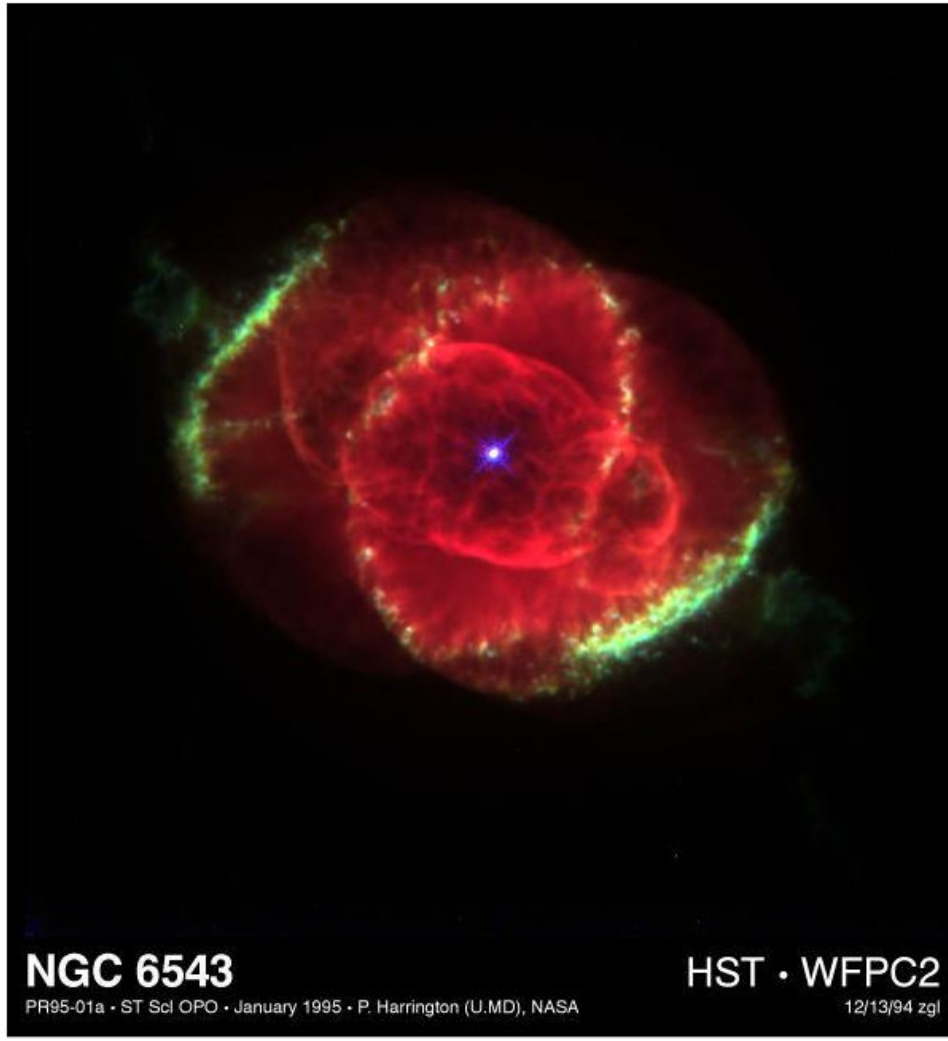
```
t = Tiff('example.tif','r');
```

Read the image data and the associated alpha matting.

```
[RGB,A] = readRGBAImage(t);
```

Display the image.

```
imshow(RGB)
```



Close the `Tiff` object.

```
close(t);
```

Input Arguments

t — Tiff object
Tiff object

Tiff object representing a TIFF file. Use the Tiff function to create the object.

Output Arguments

RGB — Image data
numeric array

Image data, returned as a m-by-n-by-3 numeric array. Where m and n are the height and width of the image, respectively.

The readRGBAImage function can transform the pixel values based on specifications in these tags:
PhotometricInterpretation
BitsPerSample
SamplesPerPixel
Orientation
ExtraSamples
ColorMap

alpha — Alpha matting
numeric array

Alpha matting associated with the image, returned as an m-by-n numeric array. Where m and n are the height and width of the image, respectively.

If the image does not have associated alpha matting, then alpha is a matrix with all values set to 255 (transparent).

Algorithms

References

This function corresponds to the `TIFFReadRGBAImage` function in the LibTIFF C API. To use this function, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`read` | `readRGBAStrip` | `readRGBATile`

Topics

"Importing Images"
"Exporting to Images"

Introduced in R2009b

readRGBAStrip

Read strip data using RGBA interface

Syntax

```
[RGB,alpha] = readRGBAStrip(t,row)
```

Description

`[RGB,alpha] = readRGBAStrip(t,row)` reads a strip of image data specified by `row` from the TIFF file associated with the `Tiff` object `t` using the RGBA interface.

The `readRGBAStrip` function trims the area in a strip that falls outside of the `ImageLength` boundary. Therefore, image data from a strip at the bottom edge of the image can have different dimensions.

Examples

Get Strip from RGBA TIFF Image

Read a strip of data from a TIFF file using the RGBA interface.

Create a `Tiff` object for the image file.

```
t = Tiff('peppers_RGB_stripped.tif','r');
```

Get the number of strips and strip size. The `readRGBAStrip` function trims the area in a strip that falls outside of the `ImageLength` boundary. Therefore, image data from a strip at the bottom edge of the image can have different dimensions.

```
numberOfStrips(t)
```

```
ans = 6
```

```
getTag(t,'RowsPerStrip')
```

```
ans = 35
```

Read the strip containing the fourth row of the image. Since each strip contains 35 rows, specifying any row between 1 and 35 returns the first strip of data.

```
[RGB,A] = readRGBAStrip(t,4);
```

Display the image.

```
imshow(RGB)
```

Close the `Tiff` object.

```
close(t);
```

Input Arguments

`t` — Tiff object
`Tiff` object

`Tiff` object representing a TIFF file. Use the `Tiff` function to create the object.

`row` — Row number
positive integer

Row number, specified as positive integer. Specify `row` as a one-based number of any row contained by the strip.

Example: 25

Data Types: `double`

Output Arguments

RGB — Image strip data
numeric array

Image strip data, returned as a `m`-by-`n`-by-`3` numeric array. Where `m` and `n` are the height and width of the strip, respectively.

The value in the `RowsPerStrip` tag determines the height the strip and the `ImageWidth` tag determines the width of the strip.

The `readRGBAStrip` function can transform the pixel values based on specifications in these tags:
`PhotometricInterpretation`
`BitsPerSample`
`SamplesPerPixel`
`Orientation`
`ExtraSamples`
`ColorMap`

`alpha` — Alpha matting
numeric array

Alpha matting associated with the image strip, returned as a numeric array. The number of rows and columns in the alpha matting data are the same as the strip data.

If the image does not have associated alpha matting, then `alpha` is a matrix with all values set to `255` (transparent).

Algorithms

References

This function corresponds to the `TIFFReadRGBAStrip` function in the LibTIFF C API. To use this function, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff` | `readRGBAImage` | `readRGBATile`

Topics

"Importing Images"
"Exporting to Images"

Introduced in R2009b

readRGBATile

Read tile data using RGBA interface

Syntax

```
[RGB,alpha] = readRGBATile(t,row,col)
```

Description

`[RGB,alpha] = readRGBATile(t,row,col)` uses the RGBA interface to read a tile from the TIFF file associated with the `Tiff` object `t`. The tile is specified by `row` and `col` which are the one-based row and column numbers of any pixel in the requested tile.

The `readRGBATile` function trims the area in a tile that falls outside of the `ImageLength` and `ImageWidth` boundaries. Therefore, image data from tiles that occur on the bottom edge or the right edge of the image can have different dimensions.

Examples

Get Tile from RGBA TIFF Image

Read a tile of data from a TIFF image file using the RGBA interface.

Create a `Tiff` object for the image file.

```
t = Tiff('peppers_RGB_tiled.tif','r');
```

Get the number of tiles and tile size.

```
numTiles = numberOfTiles(t)
```

```
numTiles = 36
```

```
tileSize = [getTag(t,'TileLength') getTag(t,'TileWidth')]
```

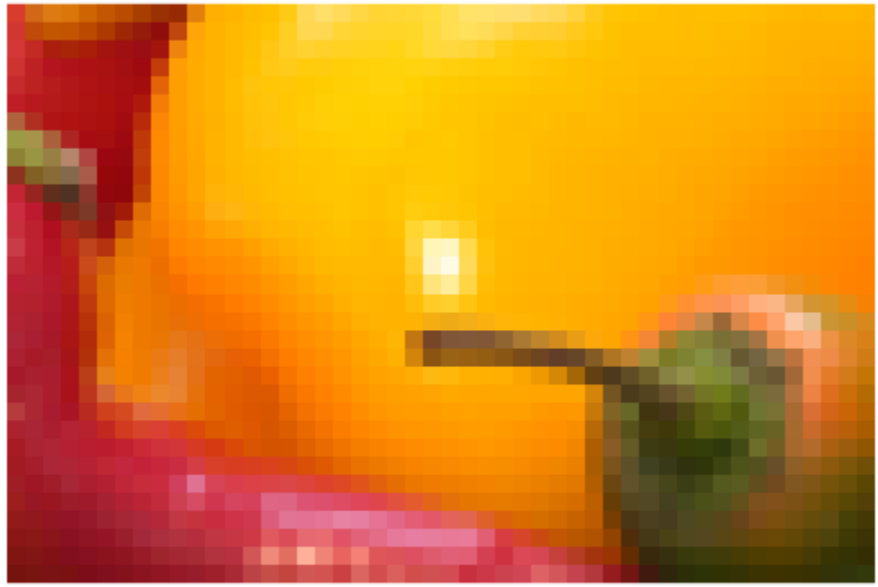
```
tileSize = 1×2

    32    48
```

Read the tile containing the `100`th row and `100`th column from the image. The `readRGBATile` function trims the area in a tile that is outside of the `ImageLength` and `ImageWidth` boundaries. Therefore, image data from tiles that occur on the right edge or the bottom edge of the image can have different dimensions.

```
row = 100;
col = 100;
[RGB,A] = readRGBATile(t,row,col);
```

Display the image.

```
imshow(RGB,'InitialMagnification','fit') % magnify for display
```



Close the `Tiff` object.

```
close(t);
```

Input Arguments

t — Tiff object
`Tiff` object

`Tiff` object representing a TIFF file. Use the `Tiff` function to create the object.

`row` — Row number
positive integer

Row number, specified as positive integer. Specify `row` as a one-based row index of any pixel in the tile.

Example: `100`

Data Types: `double`

`col` — Column number
positive integer

Column number, specified as positive integer. Specify `col` as a one-based column index of any pixel in the tile.

Example: `57`

Data Types: `double`

Output Arguments

`RGB` — Image tile data
numeric array

Image tile data, returned as an `m`-by-`n`-by-`3` numeric array. Where `m` and `n` are the height and width of the tile, respectively.

The value in the `TileLength` tag determines the number of rows and the `TileWidth` tag determines the number of columns in the tile data.

The `readRGBATile` function can transform the pixel values based on specifications in these tags:
`PhotometricInterpretation`
`BitsPerSample`
`SamplesPerPixel`
`Orientation`
`ExtraSamples`
`ColorMap`

`alpha` — Alpha matting
numeric array

Alpha matting associated with the image tile, returned as a numeric array. The number of rows and columns in the alpha matting data are the same as the tile data.

If the image does not have associated alpha matting, then `alpha` is a matrix with all values set to `255` (transparent).

Algorithms

References

This function corresponds to the `TIFFReadRGBATile` function in the LibTIFF C API. To use this function, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`readRGBAImage` | `readRGBAStrip`

Topics

"Importing Images"
"Exporting to Images"

Introduced in R2009b

Remove

Convenience function for static .NET System.Delegate Remove method

Syntax

```
result = Remove(combinedDelegate,removedDelegate)
```

Description

`result = Remove(combinedDelegate,removedDelegate)` removes last instance of the `removedDelegate` delegate from the `combinedDelegate` delegate.

Input Arguments

`combinedDelegate`

.NET `System.Delegate` object. The combined delegate from which to remove the `removedDelegate` delegate.

Default:

`removedDelegate`

.NET `System.Delegate` object. The delegate to remove from the `combinedDelegate` delegate.

Default:

Output Arguments

`result`

.NET `System.Delegate` object. A new delegate which is the same as the `combinedDelegate` delegate except without the last instance of the `removedDelegate` delegate.

Alternatives

Use the static `Remove` method of the `System.Delegate` class.

See Also

`Combine` | `RemoveAll`

Topics

"Combine and Remove .NET Delegates"

External Websites

MSDN System.Delegate.Remove Method reference page

Introduced in R2011a

RemoveAll

Convenience function for static .NET System.Delegate RemoveAll method

Syntax

```
result = RemoveAll(combinedDelegate,removedDelegate)
```

Description

`result = RemoveAll(combinedDelegate,removedDelegate)` removes all instances of `removedDelegate` from `combinedDelegate`.

Input Arguments

`combinedDelegate`

.NET `System.Delegate` object. The combined delegate from which to remove all instances of the `removedDelegate` delegate.

Default:

`removedDelegate`

.NET `System.Delegate` object. The delegate to remove from the `combinedDelegate` delegate.

Default:

Output Arguments

`result`

.NET `System.Delegate` object. A new delegate which is the same as the `combinedDelegate` delegate except without all instances of the `removedDelegate` delegate.

Alternatives

Use the static `RemoveAll` method of the `System.Delegate` class.

See Also

`Combine` | `Remove`

Topics

"Combine and Remove .NET Delegates"

External Websites

MSDN System.Delegate.RemoveAll Method reference page

Introduced in R2011a

timeseries

Create `timeseries` object

Description

Time series represent the time-evolution of a dynamic population or process. They are used to identify, model, and forecast patterns and behaviors in data that is sampled over discrete time intervals.

Creation

To create a `timeseries` object, use the `timeseries` function with input arguments that describe the data samples.

Syntax

```
ts = timeseries(datavals)
ts = timeseries(datavals,timevals)
ts = timeseries(datavals,timevals,quality)
ts = timeseries( ___ ,'Name',tsname)
```

```
ts = timeseries()
ts = timeseries(tsname)
```

Description

`ts = timeseries(datavals)` returns a `timeseries` object containing the data in `datavals`.

`ts = timeseries(datavals,timevals)` returns a `timeseries` object containing the data in `datavals` corresponding to the times in the vector `timevals`.

`ts = timeseries(datavals,timevals,quality)` specifies quality descriptions in terms of the codes defined by `QualityInfo.Code`.

`ts = timeseries( ___ ,'Name',tsname)` specifies a name `tsname` for the `timeseries` object.

`ts = timeseries()` returns an empty `timeseries` object.

`ts = timeseries(tsname)` creates an empty `timeseries` object with name `tsname`.

Input Arguments

`datavals` — Sample data

scalar | vector | multidimensional array

Sample data, specified as a numeric or `logical` scalar, vector, or multidimensional array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`timevals` — Sample times

scalar | vector

Sample times, specified as a numeric scalar or vector, or a cell array of date character vectors. Valid date character vectors can have the following forms:

Format	Example
dd-mmm-yyyy HH:MM:SS	01-Mar-2000 15:45:17
dd-mmm-yyyy	01-Mar-2000
mm/dd/yy	03/01/00
mm/dd	03/01
HH:MM:SS	15:45:17
HH:MM:SS PM	3:45:17 PM
HH:MM	15:45
HH:MM PM	3:45 PM
mmm.dd,yyyy HH:MM:SS	Mar.01,2000 15:45:17
mmm.dd,yyyy	Mar.01,2000
mm/dd/yyyy	03/01/2000

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `cell`

`quality` — Quality codes
`[]` (default) | scalar | vector | multidimensional array

Quality codes, specified as `[]` or a scalar, vector, or multidimensional array of integers ranging from -128 to 127.

- When the quality code value is a vector, it must have the same length as the time vector. Each element applies to the corresponding data sample.
- When the quality code value is an array, it must have the same size as the data array. Each element applies to the corresponding element of the data array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`tsname` — `timeseries` name
character vector

`timeseries` name, specified as a character vector.

Properties

`Data` — Sample data
scalar | vector | multidimensional array

Sample data, represented as a numeric or `logical` scalar, vector, or multidimensional array. Either the first or the last dimension of the data must align with the orientation of the time vector.

`Data` has the following attributes:

`Dependent`	`true`

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`DataInfo` — Data information
collection of fields

Data information, represented as a collection of the following fields:

- `Units` — Character vector specifying data units.
- `Interpolation` — A `tsdata.interpolation` object that specifies the interpolation method.

  Fields of the `tsdata.interpolation` object include the following:

  - `Fhandle` — Function handle to a user-defined interpolation function.
  - `Name` — Character vector specifying the name of the interpolation method. Methods include `'linear'` (default) for linear interpolation and `'zoh'` for zero-order hold.
- `UserData` — Additional user-defined information entered as a character vector.

`Events` — Event information
array of `tsdata.event` objects

Event information, represented as an array of `tsdata.event` objects containing event information.

Fields of the `tsdata.event` object include the following:

- `EventData` — Additional user-defined information about the event.
- `Name` — Character vector specifying the name of the event
- `Time` — Time for which the event occurs, specified as a real number or a date character vector.
- `Units` — Time units.
- `StartDate` — A reference date specified as a date character vector. `StartDate` is empty when the time vector is numeric.

`IsTimeFirst` — Time vector alignment
`true` | `false`

Time vector alignment, represented as one of the following options:

- `true` — The first dimension of the data array is aligned with the time vector. For example, `ts = timeseries(rand(3,3),1:3);`
- `false` — The last dimension of the data array is aligned with the time vector. For example, `ts = timeseries(rand(3,4,5),1:5);`

`IsTimeFirst` has the following attributes:

`Dependent`	`true`
`SetAccess`	`'protected'`

`Length` — Time vector length
scalar

Time vector length, represented as a scalar.

`Length` has the following attributes:

`Dependent`	`true`
`SetAccess`	`'protected'`

`Name` — `timeseries` name
character vector

`timeseries` name, represented as a character vector.

`Quality` — Quality codes
`[]` (default) | scalar | vector | multidimensional array

Quality codes, represented as `[]` or a scalar, vector, or multidimensional array of integers ranging from -128 to 127.

- When the quality code value is a vector, it must have the same length as the time vector. Each element applies to the corresponding data sample.
- When the quality code value is an array, it must have the same size as the data array. Each element applies to the corresponding element of the data array.

`Quality` has the following attributes:

`Dependent`	`true`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`QualityInfo` — Quality information
collection of fields

Quality information to describe `Quality` codes, represented as a collection of the following fields:

- `Code` — Integer vector containing values `-128` to `127` that define the quality codes. You can assign one of these integer values to each `Data` value by using the `Quality` property.
- `Description` — Cell array of character vectors, where each element provides a readable description of the associated quality `Code`.
- `UserData` — Additional user-defined information.

The lengths of `Code` and `Description` must match.

`Time` — Time values
vector

Time values, represented as a vector.

When `TimeInfo.StartDate` is empty, values are measured relative to `0`. When `TimeInfo.StartDate` is defined, values represent date character vectors measured relative to the `StartDate`.

The length of `Time` must be the same as the first or the last dimension of `Data`.

`Time` has the following attributes:

`Dependent`	`true`

`TimeInfo` — Time information
collection of fields

Time information, represented as a collection of the following fields describing the time vector:

- `Units` — Time units with the value `'weeks'`, `'days'`, `'hours'`, `'minutes'`, `'seconds'`, `'milliseconds'`, `'microseconds'`, or `'nanoseconds'`.
- `Start` — Start time.
- `End` — End time (read only).
- `Increment` — Interval between subsequent time values (`NaN` when times are not uniformly sampled).
- `Length` — Length of time vector (read only).
- `Format` — Character vector defining the date display (see `datestr`).
- `StartDate` — Date character vector defining the reference date (see `setabstime`).

- `UserData` — Additional user-defined information.

To access the value of a field, use the form `ts.TimeInfo.field` for a `timeseries` object `ts`.

`TreatNaNasMissing` — Missing value indicator
`true` (default) | `false`

Missing value indicator, represented as one of the following options:

- `true` — Treat all `NaN` values as missing data except for descriptive statistics functions.
- `false` — Include `NaN` values in descriptive statistics functions, propagating `NaN` to the result.

`UserData` — User data
`[ ]` (default)

User data, represented as any additional data to add to the `timeseries` object.

Object Functions

Modify and Plot

addevent	Add event to timeseries
addsample	Add data sample to timeseries object
append	Concatenate timeseries objects in time
delevent	Remove event from timeseries
delsample	Remove sample from timeseries object
detrend	Subtract mean or best-fit line from timeseries object
filter	Modify frequency content of timeseries objects
idealfilter	timeseries ideal filter
plot	Plot timeseries
resample	Resample timeseries time vector
set	Set timeseries properties
setabstime	Set timeseries times as date character vectors
setinterpmethod	Set default interpolation method for timeseries object
setuniformtime	Modify uniform timeseries time vector
synchronize	Synchronize and resample two timeseries objects using common time vector

Query

get	Query timeseries properties
getabstime	Convert timeseries time vector to cell array
getdatasamples	Access timeseries data samples
getdatasamplesize	timeseries data sample size
getinterpmethod	timeseries interpolation method
getqualitydesc	timeseries data quality
getsamples	Subset of timeseries
getsampleusingtime	Subset of timeseries data
gettsafteratevent	Create timeseries at or after event
gettsafterevent	Create timeseries after event
gettsatevent	Create timeseries at event
gettsbeforeatevent	Create timeseries at or before event
gettsbeforeevent	Create timeseries before event
gettsbetweenevents	Create timeseries between events

Descriptive Statistics

iqr	Interquartile range of timeseries data
max	Maximum of timeseries data
mean	Mean of timeseries data
median	Median of timeseries data
min	Minimum of timeseries data
std	Standard deviation of timeseries data
sum	Sum of timeseries data
var	Variance of timeseries data

Examples

Create `timeseries`

Create a `timeseries` object with 5 scalar data samples, specifying a name for the `timeseries`.

```
ts1 = timeseries((1:5)','Name','MyTimeSeries');
```

Create a `timeseries` with 5 data samples, where each sample is a column vector of length 2.

```
ts2 = timeseries(rand(2,5));
```

Create a `timeseries` with 5 data samples that were sampled in intervals of 10 time units.

```
ts3 = timeseries((1:5)',[0 10 20 30 40]);
```

See Also

`timetable` | `tscollection` | `tsdata.event`

Topics

"What Are Time Series?"
"Time Series Objects and Collections"

Introduced before R2006a

addsample

Add data sample to `timeseries` object

Syntax

```
tsout = addsample(tsin,'Data',datavals,'Time',timevals)
tsout = addsample(tsin,'Data',datavals,'Time',timevals,Name,Value)
tsout = addsample(tsin,s)
```

Description

`tsout = addsample(tsin,'Data',datavals,'Time',timevals)` adds one or more data samples to a `timeseries` object `tsin` using an array of data `datavals` and a corresponding time vector `timevals`.

`tsout = addsample(tsin,'Data',datavals,'Time',timevals, Name,Value)`specifies additional information for adding data samples using one or more name-value pairs.

`tsout = addsample(tsin,s)` adds data whose information is contained in the structure `s`.

Examples

Add Data Samples

Add and change data in a `timeseries` object.

Create a `timeseries` object `ts1` and display the data and time samples.

```
ts1 = timeseries((0:10:50)',(0:5)');
ts1.Data
```

```
ans = 6×1
```

```
     0
    10
    20
    30
    40
    50
```

```
ts1.Time
```

```
ans = 6×1

     0
     1
     2
     3
     4
     5
```

Add a data sample to the end of `ts1`.

```
ts2 = addsample(ts1,'Data',60,'Time',6);
ts2.Data
```

```
ans = 7×1

     0
    10
    20
    30
    40
    50
    60
```

```
ts2.Time
```

```
ans = 7×1

     0
     1
     2
     3
     4
     5
```

```
     6
```

Change the last data sample in `ts2` from 60 to 100.

```
ts3 = addsample(ts2,'Data',100,'Time',6,'OverwriteFlag',true);
ts3.Data
```

```
ans = 7×1

     0
    10
    20
    30
    40
    50
   100
```

Input Arguments

`tsin` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

`datavals` — Sample data
scalar | vector | multidimensional array

Sample data, specified as a numeric or `logical` scalar, vector, or multidimensional array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`timevals` — Sample times
scalar | vector

Sample times, specified as a numeric scalar or vector, or a cell array of date character vectors. Valid date character vectors and strings can have the following forms:

Format	Example
dd-mmm-yyyy HH:MM:SS	01-Mar-2000 15:45:17
dd-mmm-yyyy	01-Mar-2000
mm/dd/yy	03/01/00
mm/dd	03/01
HH:MM:SS	15:45:17
HH:MM:SS PM	3:45:17 PM
HH:MM	15:45
HH:MM PM	3:45 PM
mmm.dd,yyyy HH:MM:SS	Mar.01,2000 15:45:17
mmm.dd,yyyy	Mar.01,2000
mm/dd/yyyy	03/01/2000

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `cell`

s — Sample structure
scalar

Sample structure, specified as a scalar of type `struct` with the following optional fields:

- `s.data` — Numeric array of data
- `s.time` — Numeric time vector or cell array of valid date character vectors or strings
- `s.quality` — Integer array of quality codes from -128 to 127
- `s.overwriteflag` — Overwrite indicator specified as `true` to overwrite existing data samples with the added samples

Data Types: `struct`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `tsout = addsample(tsin,'Data',5,'Time',3,'OverwriteFlag',true)`

`Quality` — Quality codes
`[]` (default) | scalar | vector | multidimensional array

Quality codes, specified as `[]` or a scalar, vector, or multidimensional array of integers ranging from -128 to 127.

- When the quality code value is a vector, it must have the same length as the time vector. Each element applies to the corresponding data sample.
- When the quality code value is an array, it must have the same size as the data array. Each element applies to the corresponding element of the data array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`OverwriteFlag` — Overwrite indicator
`false` (default) | `true`

Overwrite indicator, specified as a logical value that controls whether to overwrite data samples that have the same associated time as the added samples. `true` indicates the added data overwrites the previous data, while `false` adds the sample resulting in a repeated time value.

Data Types: `logical`

See Also

`delsample` | `getdatasamples` | `timeseries`

Introduced before R2006a

append

Concatenate `timeseries` objects in time

Syntax

```
ts = append(ts1,ts2,…,tsN)
```

Description

`ts = append(ts1,ts2,…,tsN)` creates a new `timeseries` object by concatenating `timeseries` objects `ts1`, `ts2`, and so on, along the time dimension.

Examples

Append Two `timeseries` Objects

Create two `timeseries` objects, and append them by time. Display the time samples of the resulting `timeseries`.

```
ts1 = timeseries(rand(5,1),[1 2 3 4 5]);
ts2 = timeseries(rand(5,1),[6 7 8 9 10]);
ts = append(ts1,ts2);
ts.Time
```

```
ans = 10×1

     1
     2
     3
     4
     5
     6
     7
     8
     9
```

```
10
```

Input Arguments

`ts1,ts2,…,tsN` — Input `timeseries`
list of scalar `timeseries` objects

Input `timeseries`, specified as a list of scalar `timeseries` objects to concatenate.

- The last time sample of an input `timeseries` must be earlier than or equal to the first time of the following `timeseries` in the list.
- The sample size of each input `timeseries` must be the same.

Data Types: `timeseries`

See Also

`timeseries`

Introduced before R2006a

delsample

Remove sample from `timeseries` object

Syntax

```
tsout = delsample(tsin,'Index',timeind)
tsout = delsample(tsin,'Value',timevals)
```

Description

`tsout = delsample(tsin,'Index',timeind)` deletes the samples in `tsin` corresponding to the time indices in `timeind`.

`tsout = delsample(tsin,'Value',timevals)` deletes the samples corresponding to the time values in `timevals`.

Examples

Delete Sample

Delete a sample in a `timeseries` object.

Create a `timeseries` and display its time samples and corresponding data.

```
tsin = timeseries((0:10:50)',(0:5)');
tsin.Time
```

```
ans = 6×1

     0
     1
     2
     3
     4
```

```
     5
```

```
tsin.Data
```

```
ans = 6×1

     0
    10
    20
    30
    40
    50
```

Delete the last sample by referencing its index, which is 6.

```
tsout = delsample(tsin,'Index',6);
tsout.Time
```

```
ans = 5×1

     0
     1
     2
     3
     4
```

Alternatively, you can specify the time value 5 to remove the last sample of `tsin`.

```
tsout = delsample(tsin,'Value',5);
```

Input Arguments

`tsin` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

`timeind` — Time indices

scalar | vector

Time indices, specified as a numeric scalar or vector.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`timevals` — Sample times

scalar | vector

Sample times, specified as a numeric scalar or vector, or a cell array of date character vectors. Valid date character vectors can have the following forms:

Format	Example
`dd-mmm-yyyy HH:MM:SS`	`01-Mar-2000 15:45:17`
`dd-mmm-yyyy`	`01-Mar-2000`
`mm/dd/yy`	`03/01/00`
`mm/dd`	`03/01`
`HH:MM:SS`	`15:45:17`
`HH:MM:SS PM`	`3:45:17 PM`
`HH:MM`	`15:45`
`HH:MM PM`	`3:45 PM`
`mmm.dd,yyyy HH:MM:SS`	`Mar.01,2000 15:45:17`
`mmm.dd,yyyy`	`Mar.01,2000`
`mm/dd/yyyy`	`03/01/2000`

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `cell`

See Also

`set` | `timeseries`

Introduced before R2006a

detrend

Subtract mean or best-fit line from `timeseries` object

Syntax

```
tsout = detrend(tsin,method)
tsout = detrend(tsin,method,ind)
```

Description

`tsout = detrend(tsin,method)` subtracts either a mean or a best-fit line from 2-D data in a `timeseries` using the specified `method`, and also removes all `NaN` values.

`tsout = detrend(tsin,method,ind)` specifies the indices of the columns or rows to detrend. `ind` is a vector of integers representing column indices for column-oriented data (`tsin.IsTimeFirst` is `true`) and representing row indices for row-oriented data (`tsin.IsTimeFirst` is `false`).

Examples

Subtract Mean from Data

Create a `timeseries` object with 2-D data, and subtract the mean of each column from the data.

```
A = magic(3)
```

```
A = 3×3

     8     1     6
     3     5     7
     4     9     2
```

```
tsin = timeseries(A,[1 2 3]);
tsout = detrend(tsin,'constant');
tsout.Data
```

```
ans = 3×3

     3    -4     1
    -2     0     2
    -1     4    -3
```

Subtract the mean of each column for only the second and third columns of data.

```
tscol = detrend(tsin,'constant',[2 3]);
tscol.Data
```

```
ans = 3×3

     8    -4     1
     3     0     2
     4     4    -3
```

Input Arguments

`tsin` — Input `timeseries`
scalar

Input `timeseries` containing 2-D data, specified as a scalar.

Data Types: `timeseries`

`method` — Detrend method
`'constant'` | `'linear'`

Detrend method, specified as one of the following options:

- `'constant'` — Subtract the mean from the data.
- `'linear'` — Subtract the best-fit line from the data.

`ind` — Row or column indices
scalar | vector

Row or column indices, specified as a positive integer numeric scalar or vector. `ind` represents column indices for column-oriented data (`tsin.IsTimeFirst` is `true`) and represents row indices for row-oriented data (`tsin.IsTimeFirst` is `false`).

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

See Also

`timeseries`

Introduced before R2006a

filter

Modify frequency content of `timeseries` objects

Syntax

```
tsout = filter(tsin,b,a)
tsout = filter(tsin,b,a,ind)
```

Description

`tsout = filter(tsin,b,a)` applies the rational transfer function on page 1-12903 filter $b(z^{-1})/a(z^{-1})$ to the uniformly-spaced data in the `timeseries` object `tsin`. The numerator `b` and denominator `a` are vectors containing the transfer function coefficients.

`tsout = filter(tsin,b,a,ind)` specifies the indices of the columns or rows to filter. `ind` is a vector of integers representing column indices for column-oriented data (`tsin.IsTimeFirst` is `true`) and representing row indices for row-oriented data (`tsin.IsTimeFirst` is `false`).

Examples

Apply Transfer Function

This example applies the following transfer function to a set of data:

$$H(z^{-1}) = \frac{b(z^{-1})}{a(z^{-1})} = \frac{2 + 3z^{-1}}{1 + 0.2z^{-1}}$$

Create a `timeseries` object from the matrix `count` in `count.dat`.

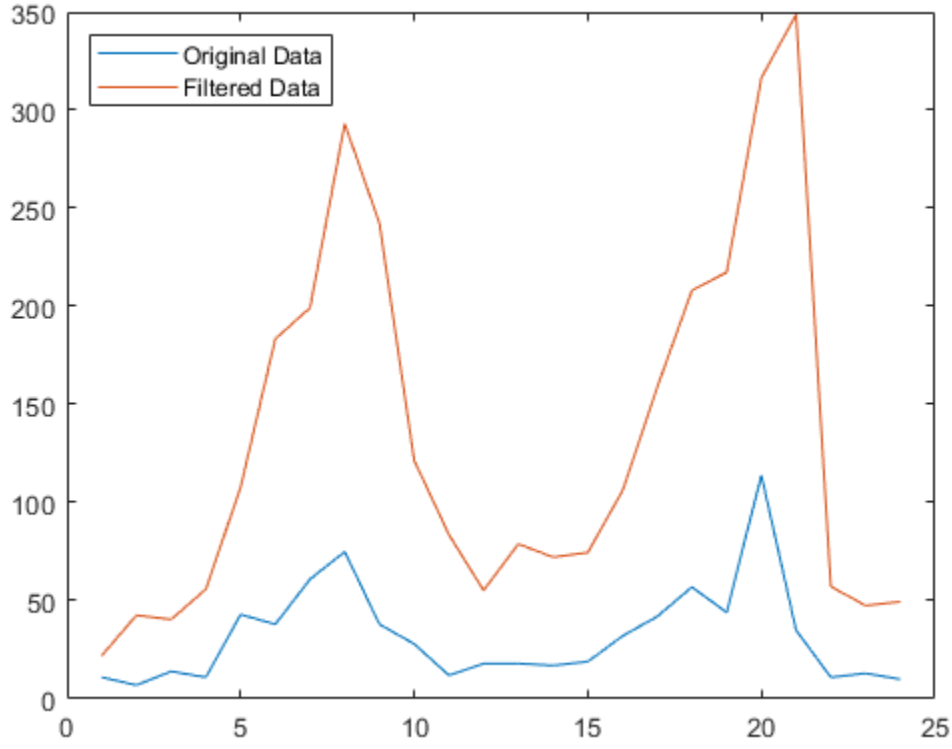
```
load count.dat
tsin = timeseries(count(:,1),[1:24]);
```

Enter the coefficients for the denominator and numerator of the transfer function. Order the coefficients in ascending powers of z^{-1} to represent $1 + 0.2x$ and $2 - 3z^{-1}$.

```
a = [1 0.2];
b = [2 3];
```

Apply the transfer function using `filter`, and compare the original data to the filtered data.

```
tsout = filter(tsin,b,a);
plot(tsin)
hold on
plot(tsout)
legend('Original Data','Filtered Data','Location','NorthWest')
```



Input Arguments

`tsin` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar. `tsin` must be uniformly sampled.

Data Types: `timeseries`

`b` — Numerator coefficients
scalar | vector

Numerator coefficients of the transfer function, specified as a scalar or vector.

`a` — Denominator coefficients
scalar | vector

Denominator coefficients of the transfer function, specified as a scalar or vector.

`ind` — Row or column indices
scalar | vector

Row or column indices, specified as a positive integer numeric scalar or vector. `ind` represents column indices for column-oriented data (`tsin.IsTimeFirst` is `true`) and represents row indices for row-oriented data (`tsin.IsTimeFirst` is `false`).

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

More About

Rational Transfer Function

The input-output description of the `filter` operation on a vector in the Z-transform domain is a rational transfer function. A rational transfer function is of the form,

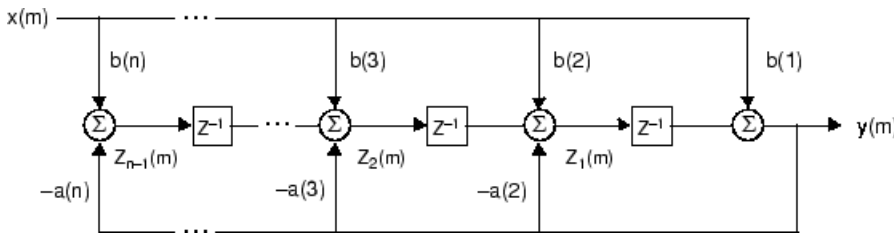
$$Y(z) = \frac{b(1) + b(2)z^{-1} + \ldots + b(n_b + 1)z^{-n_b}}{1 + a(2)z^{-1} + \ldots + a(n_a + 1)z^{-n_a}} X(z),$$

which handles both FIR and IIR filters [1]. n_a is the feedback filter order, and n_b is the feedforward filter order.

You also can express the rational transfer function as the following difference equation,

$$a(1)y(n) = b(1)x(n) + b(2)x(n-1) + \ldots + b(n_b+1)x(n-n_b)$$
$$-a(2)y(n-1) - \ldots - a(n_a+1)y(n-n_a)\,.$$

Furthermore, you can represent the rational transfer function using its direct form II transposed implementation, as in the following diagram. Due to normalization, assume `a(1) = 1`.



The operation of `filter` at sample m is given by the time domain difference equations

$$y(m) = b(1)x(m) + w_1(m-1)$$
$$w_1(m) = b(2)x(m) + w_2(m-1) - a(2)y(m)$$
$$\vdots \quad = \quad \vdots \qquad \vdots$$
$$w_{n-2}(m) = b(n-1)x(m) + w_{n-1}(m-1) - a(n-1)y(m)$$
$$w_{n-1}(m) = b(n)x(m) - a(n)y(m)\,.$$

See Also

`timeseries`

Introduced before R2006a

get

Query `timeseries` properties

Syntax

```
tsinfo = get(ts)
tsinfo = get(ts,propname)
```

Description

`tsinfo = get(ts)` returns all properties of the `timeseries` object `ts`.

`tsinfo = get(ts,propname)` returns the specified property `propname` of `ts`, and is equivalent to `tsinfo = ts.propname`. For a full list of `timeseries` property names, see `timeseries`.

Examples

Display Properties

Create a `timeseries` object, and display its properties.

```
ts = timeseries(rand(5,1));
tsinfo = get(ts)
```

```
tsinfo = struct with fields:
               Events: []
                 Name: 'unnamed'
             UserData: []
                 Data: [5x1 double]
             DataInfo: [1x1 tsdata.datametadata]
                 Time: [5x1 double]
             TimeInfo: [1x1 tsdata.timemetadata]
              Quality: []
          QualityInfo: [1x1 tsdata.qualmetadata]
```

```
          IsTimeFirst: 1
    TreatNaNasMissing: 1
               Length: 5
```

Display only the `Length` property.

```
tslength = get(ts,'Length')
```

```
tslength = 5
```

Input Arguments

`ts` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

`propname` — Property name
character vector

Property name, specified as a character vector. For a full list of `timeseries` property names, see `timeseries`.

Data Types: `char`

See Also

`set` | `timeseries`

Introduced before R2006a

getabstime

Convert `timeseries` time vector to cell array

Syntax

```
tstime = getabstime(ts)
```

Description

`tstime = getabstime(ts)` returns a cell array whose elements are the dates in the time vector associated with a `timeseries` object `ts`. The property `ts.TimeInfo.StartDate` must represent a date as a character vector.

Examples

Cell Array of Time Vector Elements

Extract time values in a `timeseries` time vector as a cell array of dates and times.

Create a `timeseries` object with the default time vector `[0 1 2 3 4]`.

```
ts = timeseries([3 6 8 0 10]);
```

Set the `TimeInfo.StartDate` property to a date character vector and display the new times associated with `ts`, which are in increments of 1 second.

```
ts.TimeInfo.StartDate = '10/27/2005';
tstime = getabstime(ts)
```

```
tstime = 5x1 cell array
    {'27-Oct-2005 00:00:00'}
    {'27-Oct-2005 00:00:01'}
    {'27-Oct-2005 00:00:02'}
    {'27-Oct-2005 00:00:03'}
```

```
{'27-Oct-2005 00:00:04'}
```

Input Arguments

ts — Input timeseries
scalar

Input `timeseries`, specified as a scalar.

Tips

- To define the `timeseries` time vector relative to a calendar date, set the `TimeInfo.StartDate` property to a valid date character vector. Valid dates can have the following formats:

Format	Example
dd-mmm-yyyy HH:MM:SS	01-Mar-2000 15:45:17
dd-mmm-yyyy	01-Mar-2000
mm/dd/yy	03/01/00
mm/dd	03/01
HH:MM:SS	15:45:17
HH:MM:SS PM	3:45:17 PM
HH:MM	15:45
HH:MM PM	3:45 PM
mmm.dd,yyyy HH:MM:SS	Mar.01,2000 15:45:17
mmm.dd,yyyy	Mar.01,2000
mm/dd/yyyy	03/01/2000

See Also

`setabstime` | `timeseries`

Introduced before R2006a

getdatasamples

Access `timeseries` data samples

Syntax

```
tsdata = getdatasamples(ts,ind)
```

Description

`tsdata = getdatasamples(ts,ind)` returns the data of a `timeseries` object `ts` corresponding to the indices in `ind`. The data in `tsdata` corresponds to the time samples in `ts.time(ind)`.

Examples

Access Data Samples

Create a timeseries object, and extract the second and third data samples.

```
ts = timeseries((1:5)');
tsdata = getdatasamples(ts,[2,3])
```

```
tsdata = 2×1

     2
     3
```

Input Arguments

`ts` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

`ind` — Row or column indices
scalar | vector

Row or column indices, specified as a positive integer numeric scalar or vector. `ind` represents column indices for column-oriented data (`ts.IsTimeFirst` is `true`) and represents row indices for row-oriented data (`ts.IsTimeFirst` is `false`).

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

See Also

`getsamples` | `resample` | `timeseries`

Introduced before R2006a

getdatasamplesize

`timeseries` data sample size

Syntax

```
tsdatasize = getdatasamplesize(ts)
```

Description

`tsdatasize = getdatasamplesize(ts)` returns the size of each data sample in a `timeseries` object `ts`.

Examples

Display Data Sample Size

Create a `timeseries` object from a 24-by-3 array of data and display the size of each data sample.

```
load count.dat
ts = timeseries(count,[1:24]);
tsdatasize = getdatasamplesize(ts)
```

```
tsdatasize = 1×2

     1     3
```

Input Arguments

`ts` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

See Also

`timeseries`

Introduced before R2006a

getinterpmethod

`timeseries` interpolation method

Syntax

```
tsinterp = getinterpmethod(ts)
```

Description

`tsinterp = getinterpmethod(ts)` returns the interpolation method used by the `timeseries` object. `tsinterp` can be `'linear'` for linear interpolation (default) or `'zoh'` for zero-order hold.

Examples

Display Interpolation Method

Create a `timeseries` object and display its interpolation method.

```
ts = timeseries(rand(5,1));
tsinterp = getinterpmethod(ts)
```

```
tsinterp =
'linear'
```

Input Arguments

`ts` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

See Also

`setinterpmethod` | `timeseries`

Introduced before R2006a

getqualitydesc

`timeseries` data quality

Syntax

```
tsquality = getqualitydesc(ts)
```

Description

`tsquality = getqualitydesc(ts)` returns a cell array of data quality descriptions based on the `Quality` values assigned to a `timeseries` object `ts`.

Examples

Quality Descriptions

Create a timeseries object `ts`, specifying a quality vector made up of zeros and ones. Assign values to the `QualityInfo.Code` and `QualityInfo.Description` properties, and display the quality descriptions.

```
ts = timeseries([3 4.2 5 6.1 8]',1:5,[1 0 1 0 1]');
ts.QualityInfo.Code = [0 1];
ts.QualityInfo.Description = {'Good' 'Bad'};
tsquality = getqualitydesc(ts)
```

```
tsquality = 5x1 cell array
    {'Bad' }
    {'Good'}
    {'Bad' }
    {'Good'}
    {'Bad' }
```

Input Arguments

`ts` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

See Also

`timeseries`

Introduced before R2006a

getsamples

Subset of `timeseries`

Syntax

```
tsout = getsamples(tsin,ind)
```

Description

`tsout = getsamples(tsin,ind)` returns a `timeseries` object that is a subset of the input `timeseries`. The samples in `tsout` are the samples of `tsin` corresponding to `tsin.Time(ind)`.

Examples

Subset of timeseries

Create a `timeseries` object `tsout` that is a subset of the samples in the `timeseries` object `tsin`.

```
tsin = timeseries([10 20 30 40 50]',[1 2 3 4 5]);
tsout = getsamples(tsin, tsin.time([2 3]));
tsout.Data
```

```
ans = 2×1

    20
    30
```

Input Arguments

`tsin` — Input `timeseries`

scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

`ind` — Row or column indices

scalar | vector

Row or column indices, specified as a positive integer numeric scalar or vector. `ind` represents column indices for column-oriented data (`ts.IsTimeFirst` is `true`) and represents row indices for row-oriented data (`ts.IsTimeFirst` is `false`).

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

See Also

`getdatasamples` | `resample` | `timeseries`

Introduced before R2006a

getsampleusingtime

Subset of `timeseries` data

Syntax

```
tsout = getsampleusingtime(tsin,timeval)
tsout = getsampleusingtime(tsin,timeval,'AllowDuplicateTimes',true)
tsout = getsampleusingtime(tsin,starttime,endtime)
```

Description

`tsout = getsampleusingtime(tsin,timeval)` returns a `timeseries` object that contains the single data sample of an input `timeseries` corresponding to the time `timeval`.

`tsout = getsampleusingtime(tsin,timeval,'AllowDuplicateTimes',true)` includes multiple data samples with the same time value specified in `timeval`.

`tsout = getsampleusingtime(tsin,starttime,endtime)` includes samples of `tsin` between the times `starttime` and `endtime`.

Examples

Extract Subset of `timeseries`

Create a `timeseries` object and extract the data samples corresponding to the 2nd through 4th time values.

```
tsin = timeseries((1:5)',1:5);
tsout = getsampleusingtime(tsin,2,4);
tsout.Data
```

```
ans = 3×1
```

```
2
3
4
```

Input Arguments

tsin — Input timeseries

scalar

Input `timeseries`, specified as a scalar.

timeval — Sample time

scalar | vector

Sample time, specified as a numeric scalar, date character vector, or `datenum` scalar. Valid date character vectors can have the following forms:

Format	Example
dd-mmm-yyyy HH:MM:SS	01-Mar-2000 15:45:17
dd-mmm-yyyy	01-Mar-2000
mm/dd/yy	03/01/00
mm/dd	03/01
HH:MM:SS	15:45:17
HH:MM:SS PM	3:45:17 PM
HH:MM	15:45
HH:MM PM	3:45 PM
mmm.dd,yyyy HH:MM:SS	Mar.01,2000 15:45:17
mmm.dd,yyyy	Mar.01,2000
mm/dd/yyyy	03/01/2000

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `cell`

starttime — Start time

scalar | vector

Start time, specified as a numeric scalar, date character vector, or `datenum` scalar.

`endtime` — End time
scalar | vector

End time, specified as a numeric scalar, date character vector, or `datenum` scalar.

Tips

- If the time vector in `tsin` is not relative to a calendar date, then `starttime` and `endtime` must be numeric.
- If the time vector in `tsin` is relative to a calendar date, then `starttime` and `endtime` values must be date character vectors or `datenum` values.

See Also

`timeseries`

Introduced before R2006a

idealfilter

`timeseries` ideal filter

Syntax

```
tsout = idealfilter(tsin,interval,filtertype)
tsout = idealfilter(tsin,interval,filtertype,ind)
```

Description

`tsout = idealfilter(tsin,interval,filtertype)` applies an ideal (noncausal) filter of type `filtertype` to the frequency intervals specified by `interval` for a `timeseries` object `tsin`.

Ideal filters are noncausal, and the ends of the filter amplitude are flat in the frequency domain. The data in `ts` must have zero mean.

`tsout = idealfilter(tsin,interval,filtertype,ind)` optionally specifies the row or column indices of `tsin` to apply the filter to.

Examples

Apply Ideal Notch and Pass Filters

First apply an ideal notch filter to a `timeseries` object, then apply a pass filter.

Load the data in the file `count.dat`, and create a `timeseries` object from the matrix `count`.

```
load count.dat
tsin = timeseries(count(:,1),1:24);
```

Compute the mean of the data in `tsin`.

```
tsinmean = mean(tsin);
```

Define the frequency interval, in hertz, for filtering the data.

```
interval = [0.08 0.2];
```

Invoke an ideal notch filter.

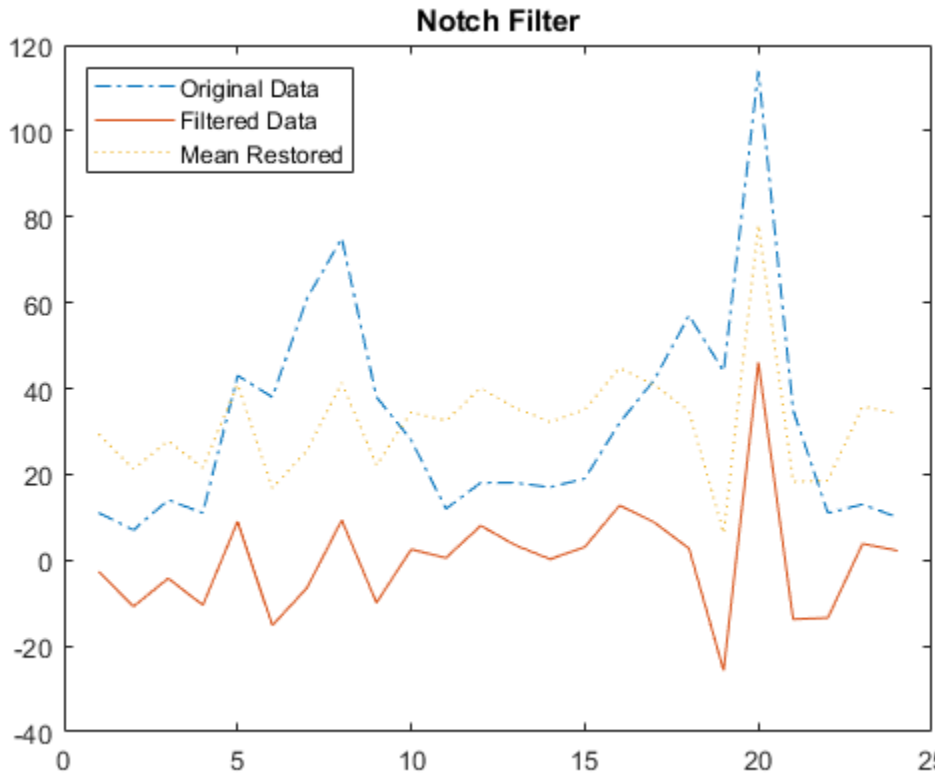
```
tsoutnotch = idealfilter(tsin,interval,'notch');
```

Compare the original data and the filtered data.

```
plot(tsin,'-.')
hold on
plot(tsoutnotch,'-')
```

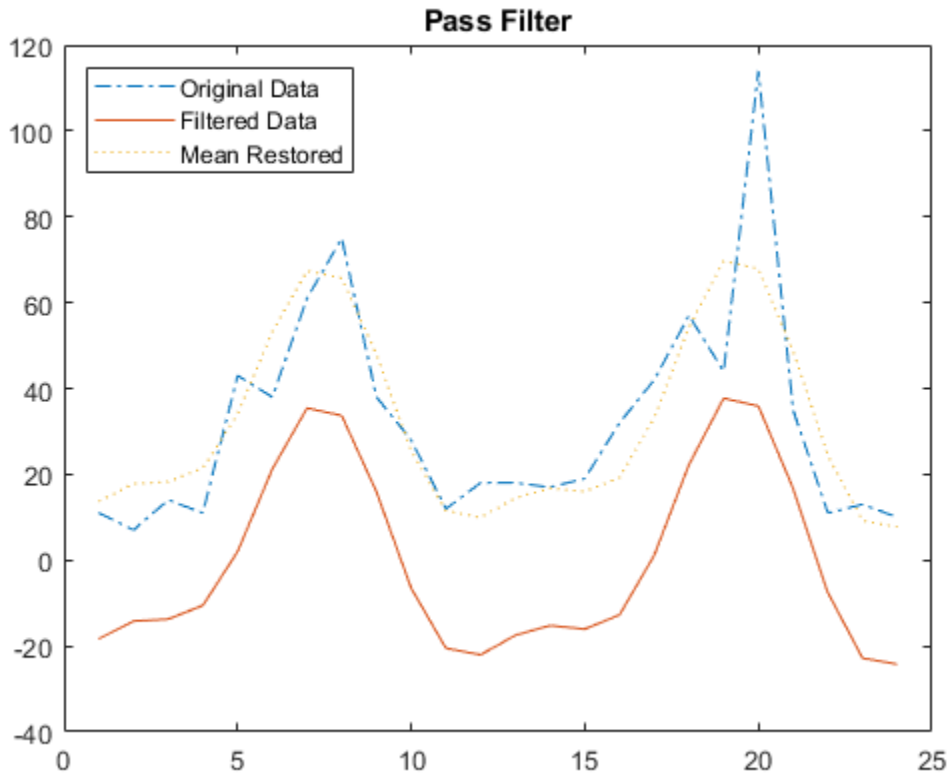
Restore the mean to the filtered data.

```
tsoutnotchmean = tsoutnotch + tsinmean;
plot(tsoutnotchmean,':')
title('Notch Filter')
legend('Original Data','Filtered Data','Mean Restored',...
       'Location','NorthWest')
hold off
```



Repeat the filtering process using a `pass` filter.

```
plot(tsin,'-.')
hold on
tsoutpass = idealfilter(tsin,interval,'pass');
plot(tsoutpass,'-')

tsoutpassmean = tsoutpass + tsinmean;
plot(tsoutpassmean,':')
title('Pass Filter')
legend('Original Data','Filtered Data','Mean Restored',...
       'Location','NorthWest')
```



Input Arguments

`tsin` — Input `timeseries`
scalar

Input `timeseries` with zero mean, specified as a scalar.

- If `tsin` is nonuniformly sampled, then `idealfilter` resamples the data on a uniform time vector before applying the filter.
- `idealfilter` replaces any `NaN` elements of `tsin` using the interpolation method associated with `tsin` prior to applying the filter.

Data Types: `timeseries`

`interval` — Frequency interval
two-column matrix

Frequency interval, specified as a two-column matrix where each row represents the start and end frequencies for each interval.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`filtertype` — Filter type
`'pass'` | `'notch'`

Filter type, specified as one of the following options:

- `'pass'` — Allow variations in a specific frequency range
- `'notch'` — Remove variations in a specific frequency range

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ind` — Row or column indices
scalar | vector

Row or column indices, specified as a positive integer numeric scalar or vector. `ind` represents column indices for column-oriented data (`tsin.IsTimeFirst` is `true`) and represents row indices for row-oriented data (`tsin.IsTimeFirst` is `false`).

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

See Also

`filter` | `timeseries`

Introduced before R2006a

iqr

Interquartile range of `timeseries` data

Syntax

```
tsiqr = iqr(ts)
tsiqr = iqr(ts,Name,Value)
```

Description

`tsiqr = iqr(ts)` returns the interquartile range of the data samples in a `timeseries` object.

`tsiqr = iqr(ts,Name,Value)` specifies additional options when computing the interquartile range using one or more name-value pair arguments. For example, `tsiqr = iqr(ts,'Quality',-99,'MissingData','remove')` defines -99 as the missing sample quality code, and removes the missing samples before computing the interquartile range.

Examples

Interquartile Range of Sample Data

Create a `timeseries` object and compute the interquartile range of the sample data.

```
ts = timeseries((1:10)');
tsiqr = iqr(ts)
```

```
tsiqr = 5
```

Input Arguments

`ts` — Input `timeseries`

scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `tsiqr = iqr(ts,'Quality',-99,'MissingData','remove')`

`Quality` — Missing value indicator

scalar | vector | matrix | multidimensional array

Missing value indicator, specified a scalar, vector, matrix, or multidimensional array of integers ranging from -128 to 127. Each element is a quality code to treat as missing data.

By default, `iqr` removes any missing data before computing the interquartile range. To interpolate the data instead of removing it, specify the name-value pair `'MissingData','interpolation'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`MissingData` — Missing data method

`'remove'` (default) | `'interpolate'`

Missing data method, specified as either `'remove'` to remove missing values before computing the interquartile range or `'interpolate'` to fill missing values by interpolating the data. Specify the `'Quality'` name-value pair to indicate which data samples are considered missing.

See Also

`mean` | `std` | `timeseries` | `var`

Introduced before R2006a

max

Maximum of `timeseries` data

Syntax

```
tsmax = max(ts)
tsmax = max(ts,Name,Value)
```

Description

`tsmax = max(ts)` returns the maximum value of the data samples in a `timeseries` object.

`tsmax = max(ts,Name,Value)` specifies additional options when computing the maximum using one or more name-value pair arguments. For example, `tsmax = max(ts,'Quality',-99,'MissingData','remove')` defines -99 as the missing sample quality code, and removes the missing samples before computing the maximum.

Examples

Maximum Data Sample

Create a `timeseries` object and compute the maximum data sample.

```
ts = timeseries((1:5)');
tsmax = max(ts)
```

```
tsmax = 5
```

Input Arguments

ts — Input timeseries
scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `tsmax = max(ts,'Quality',-99,'MissingData','remove')`

Quality — Missing value indicator
scalar | vector | matrix | multidimensional array

Missing value indicator, specified a scalar, vector, matrix, or multidimensional array of integers ranging from -128 to 127. Each element is a quality code to treat as missing data.

By default, `max` removes any missing data before computing the maximum. To interpolate the data instead of removing it, specify the name-value pair `'MissingData','interpolation'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

MissingData — Missing data method
`'remove'` (default) | `'interpolate'`

Missing data method, specified as either `'remove'` to remove missing values before computing the maximum or `'interpolate'` to fill missing values by interpolating the data. Specify the `'Quality'` name-value pair to indicate which data samples are considered missing.

See Also

`mean` | `min` | `timeseries`

Introduced before R2006a

mean

Mean of `timeseries` data

Syntax

```
tsmean = mean(ts)
tsmean = mean(ts,Name,Value)
```

Description

`tsmean = mean(ts)` returns the mean of the data samples in a `timeseries` object.

`tsmean = mean(ts,Name,Value)` specifies additional options when computing the mean using one or more name-value pair arguments. For example, `tsmean = mean(ts,'Quality',-99,'MissingData','remove')` defines -99 as the missing sample quality code, and removes the missing samples before computing the mean.

Examples

Mean of Data Samples

Create a `timeseries` object and compute the mean of the data samples.

```
ts = timeseries((1:5)');
tsmean = mean(ts)
```

```
tsmean = 3
```

Input Arguments

`ts` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `tsmean = mean(ts,'Quality',-99,'MissingData','remove')`

`Quality` — Missing value indicator
scalar | vector | matrix | multidimensional array

Missing value indicator, specified a scalar, vector, matrix, or multidimensional array of integers ranging from -128 to 127. Each element is a quality code to treat as missing data.

By default, missing data is removed before computing. To interpolate the data instead of removing it, specify the name-value pair `'MissingData','interpolation'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`MissingData` — Missing data method
`'remove'` (default) | `'interpolate'`

Missing data method, specified as either `'remove'` to remove missing values or `'interpolate'` to fill missing values by interpolating the data. Specify the `'Quality'` name-value pair to indicate which data samples are considered missing.

`Weighting` — Weights
`'none'` (default) | `'time'`

Weights, specified as `'none'` or `'time'`.
When you specify `'time'`, larger time values correspond to larger weights.

Algorithms

MATLAB determines weighting by:

1. Attaching a weighting to each time value, depending on its order, as follows:
   - First time point — The duration of the first time interval `(t(2) - t(1))`.
   - Time point that is neither the first nor last time point — The duration between the midpoint of the previous time interval to the midpoint of the subsequent time interval `((t(k + 1) - t(k))/2 + (t(k) - t(k - 1))/2)`.
   - Last time point — The duration of the last time interval `(t(end) - t(end - 1))`.
2. Normalizing the weighting for each time by dividing each weighting by the mean of all weightings.

   Note If the `timeseries` object is uniformly sampled, then the normalized weighting for each time is 1.0. Therefore, time weighting has no effect.
3. Multiplying the data for each time by its normalized weighting.

See Also

`median` | `std` | `sum` | `timeseries` | `var`

Introduced before R2006a

median

Median of `timeseries` data

Syntax

```
tsmedian = median(ts)
tsmedian = median(ts,Name,Value)
```

Description

`tsmedian = median(ts)` returns the median of the data samples in a `timeseries` object.

`tsmedian = median(ts,Name,Value)` specifies additional options when computing the median using one or more name-value pair arguments. For example, `tsmedian = median(ts,'Quality',-99,'MissingData','remove')` defines -99 as the missing sample quality code, and removes the missing samples before computing the median.

Examples

Median of Data Samples

Create a `timeseries` object and compute the median of the data samples.

```
ts = timeseries((1:5)');
tsmedian = mean(ts)
```

```
tsmedian = 3
```

Input Arguments

`ts` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `tsmedian = median(ts,'Quality',-99,'MissingData','remove')`

`Quality` — Missing value indicator
scalar | vector | matrix | multidimensional array

Missing value indicator, specified a scalar, vector, matrix, or multidimensional array of integers ranging from -128 to 127. Each element is a quality code to treat as missing data.

By default, missing data is removed before computing. To interpolate the data instead of removing it, specify the name-value pair `'MissingData','interpolation'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`MissingData` — Missing data method
`'remove'` (default) | `'interpolate'`

Missing data method, specified as either `'remove'` to remove missing values or `'interpolate'` to fill missing values by interpolating the data. Specify the `'Quality'` name-value pair to indicate which data samples are considered missing.

`Weighting` — Weights
`'none'` (default) | `'time'`

Weights, specified as `'none'` or `'time'`.
When you specify `'time'`, larger time values correspond to larger weights.

Algorithms

MATLAB determines weighting by:

1. Attaching a weighting to each time value, depending on its order, as follows:
   - First time point — The duration of the first time interval `(t(2) - t(1))`.
   - Time point that is neither the first nor last time point — The duration between the midpoint of the previous time interval to the midpoint of the subsequent time interval `((t(k + 1) - t(k))/2 + (t(k) - t(k - 1))/2)`.
   - Last time point — The duration of the last time interval `(t(end) - t(end - 1))`.
2. Normalizing the weighting for each time by dividing each weighting by the mean of all weightings.

   Note If the `timeseries` object is uniformly sampled, then the normalized weighting for each time is 1.0. Therefore, time weighting has no effect.
3. Multiplying the data for each time by its normalized weighting.

See Also

`mean` | `std` | `timeseries` | `var`

Introduced before R2006a

min

Minimum of `timeseries` data

Syntax

```
tsmin = min(ts)
tsmin = min(ts,Name,Value)
```

Description

`tsmin = min(ts)` returns the minimum value of the data samples in a `timeseries` object.

`tsmin = min(ts,Name,Value)` specifies additional options when computing the minimum using one or more name-value pair arguments. For example, `tsmin = min(ts,'Quality',-99,'MissingData','remove')` defines -99 as the missing sample quality code, and removes the missing samples before computing the minimum.

Examples

Minimum Data Sample

Create a `timeseries` object and compute the minimum data sample.

```
ts = timeseries((1:5)');
tsmin = min(ts)
```

```
tsmin = 1
```

Input Arguments

`ts` — Input `timeseries`

scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `tsmin = min(ts,'Quality',-99,'MissingData','remove')`

`Quality` — Missing value indicator

scalar | vector | matrix | multidimensional array

Missing value indicator, specified a scalar, vector, matrix, or multidimensional array of integers ranging from -128 to 127. Each element is a quality code to treat as missing data.

By default, `min` removes any missing data before computing the minimum. To interpolate the data instead of removing it, specify the name-value pair `'MissingData','interpolation'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`MissingData` — Missing data method

`'remove'` (default) | `'interpolate'`

Missing data method, specified as either `'remove'` to remove missing values before computing the minimum or `'interpolate'` to fill missing values by interpolating the data. Specify the `'Quality'` name-value pair to indicate which data samples are considered missing.

See Also

`max` | `mean` | `timeseries`

Introduced before R2006a

plot

Plot `timeseries`

Syntax

```
plot(ts)
plot(ts,specs)
```

Description

`plot(ts)` plots the `timeseries` data in `ts` against time, interpolating values between samples.

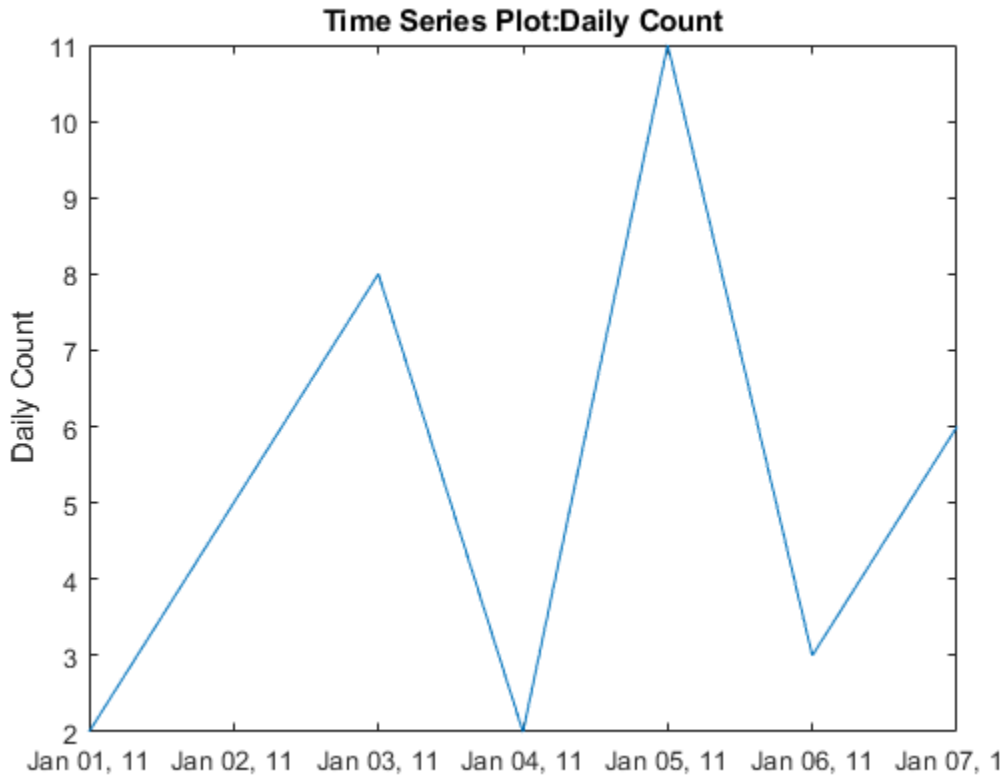
`plot(ts,specs)` plots the `timeseries` data using a line graph and applies the specified `specs` to lines, markers, or both. You can also specify name-value pairs to define Line Properties.

Examples

Plot Time Series Object with Specified Start Date

Create a time series object, set the start date, and then plot the time vector relative to the start date.

```
x = [2 5 8 2 11 3 6];
ts1 = timeseries(x,1:7);

ts1.Name = 'Daily Count';
ts1.TimeInfo.Units = 'days';
ts1.TimeInfo.StartDate = '01-Jan-2011';     % Set start date.
ts1.TimeInfo.Format = 'mmm dd, yy';       % Set format for display on x-axis.

ts1.Time = ts1.Time - ts1.Time(1);        % Express time relative to the start date.

plot(ts1)
```

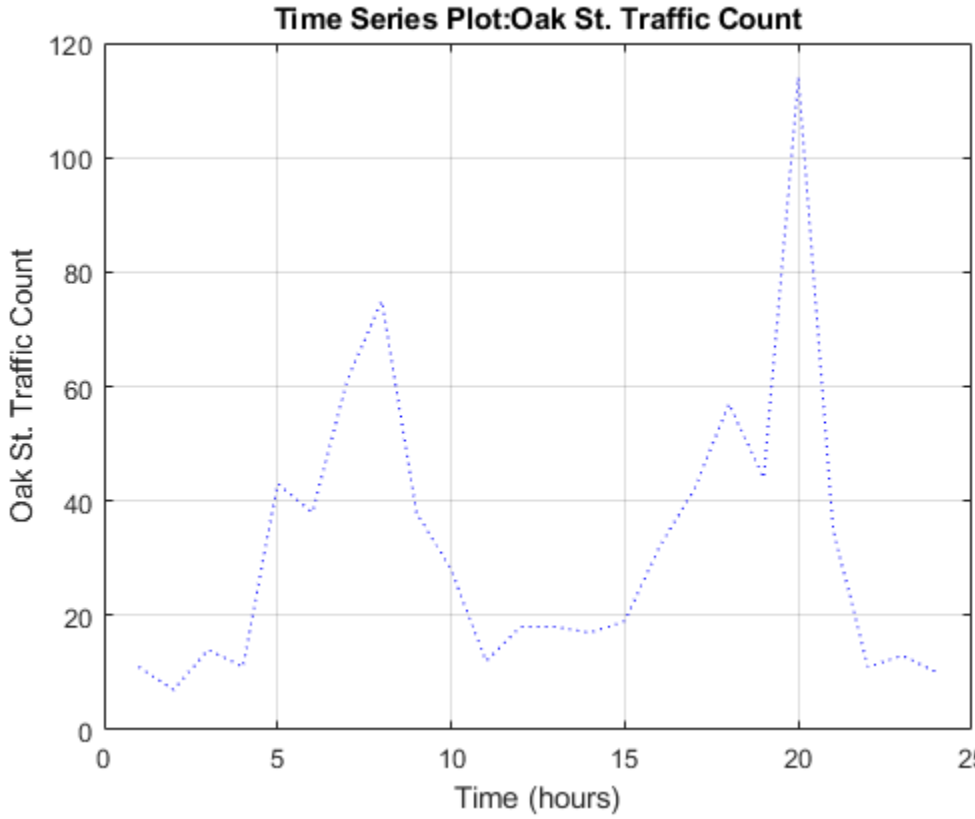


Plot Two Time Series Objects on the Same Axes

Create two time series objects from traffic count data, and then plot them in sequence on the same axes. Add an event to one series, which is automatically displayed with a red marker.

```
load count.dat;
count1 = timeseries(count(:,1),1:24);
count1.Name = 'Oak St. Traffic Count';
count1.TimeInfo.Units = 'hours';
```

```
plot(count1,':b')
grid on
```



Obtain time of maximum value and add it as an event:

```
[~,index] = max(count1.Data);
max_event = tsdata.event('peak',count1.Time(index));
max_event.Units = 'hours';
```

Add the event to the time series:

```
count1 = addevent(count1,max_event);
```

Replace plot with new one showing the event:

```
plot(count1,'.-b')
grid on
```

Make a new time series object from column 2 of the same data source:

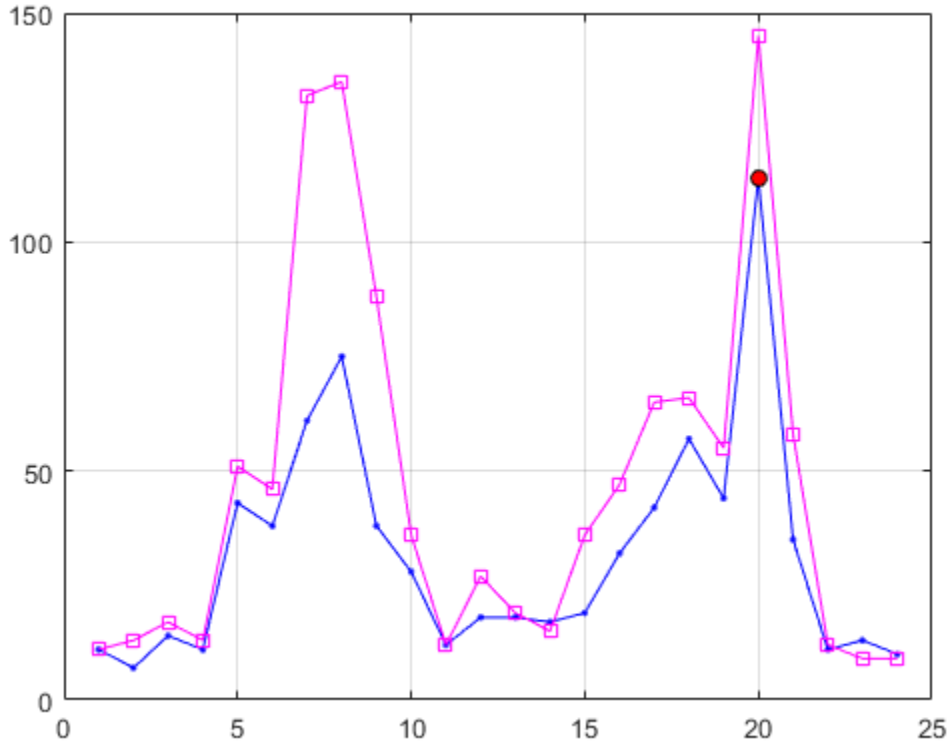
```
count2 = timeseries(count(:,2),1:24);
count2.Name = 'Maple St. Traffic Count';
count2.TimeInfo.Units = 'Hours';
```

Turn hold on to add the new data to the plot:

```
hold on
```

The plot method does not add labels to a held plot. Use property/value pairs to customize markers:

```
plot(count2,'s-m','MarkerSize',6),
```

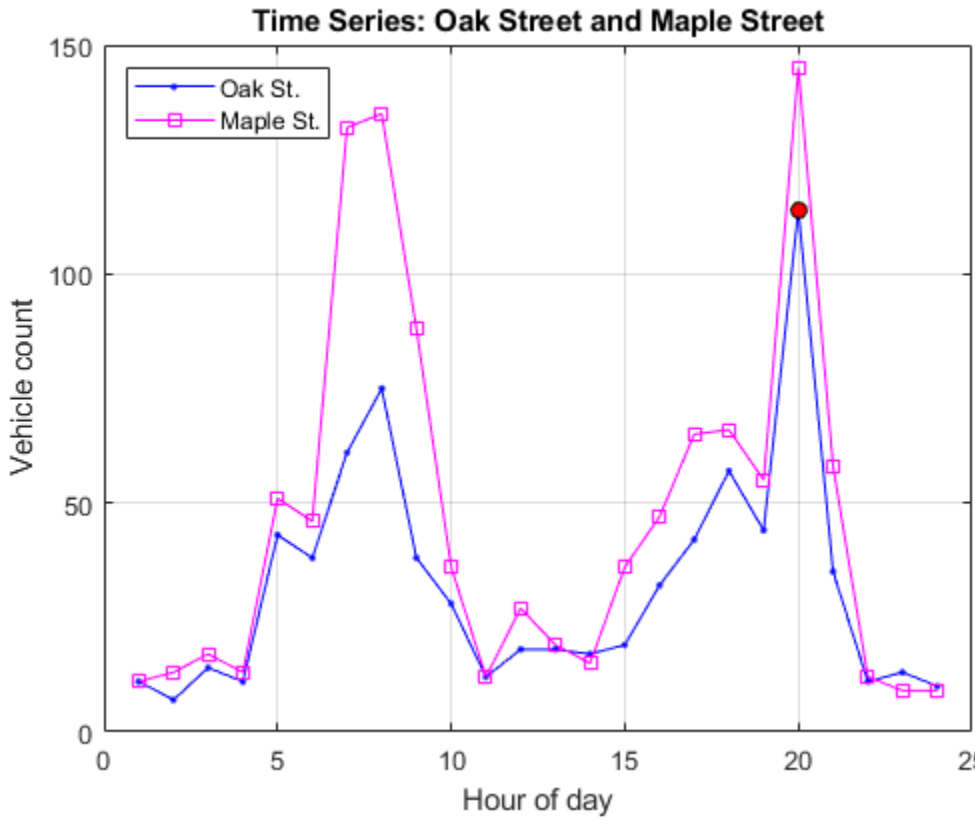


Labels are erased, so generate them manually:

```
title('Time Series: Oak Street and Maple Street')
xlabel('Hour of day')
ylabel('Vehicle count')
```

Add a legend in the upper left:

```
legend('Oak St.','Maple St.','Location','northwest')
```



Input Arguments

`ts` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

`specs` — Line specifications
character vector

Line specifications, specified as a character vector defining the appearance of lines, markers, or both. See `LineSpec` for more information. You can also specify name-value pairs to define Line Properties.

Data Types: `char`

Tips

- The `plot` function generates titles and axis labels automatically. These labels are:
  - Plot Title — `'Time Series Plot: <name>'`

    where `<name>` is the string assigned to `ts.Name`, or by default, `'unnamed'`
  - X-Axis Label — `'Time (<units>)'`

    where `<units>` is the value of the `ts.TimeInfo.Units` field, which defaults to `'seconds'`
  - Y-Axis Label — `'<name>'`

    where `<name>` is the string assigned to `ts.Name`, or by default, `'unnamed'`
- You can place new time series data on a time series plot (by setting `hold on`, for example, and issuing another `timeseries/plot` command). When you add data to a plot, the title and axis labels become blank strings to avoid labeling confusion. You can add your own labels after plotting using the `title`, `xlabel`, and `ylabel` commands.
- Time series events, when defined, are marked in the plot with a circular marker with red fill. You can also specify markers for all data points using a `linespec` or name/value syntax in addition to any event markers your data defines. The event markers plot on top of the markers you define.
- The value assigned to `ts.DataInfo.Interpolation.Name` controls the type of interpolation the `plot` method uses when plotting and resampling time series data. Invoke the `timeseries` method `setinterpmethod` to change default linear interpolation to zero-order hold interpolation (staircase). This method creates a new `timeseries` object, with which you can overwrite the original one if you want. For example, to cause time series `ts` to use zero-order hold interpolation, type the following:

  ```
  ts = ts.setinterpmethod('zoh');
  ```

See Also

`plot` | `setinterpmethod` | `timeseries`

Introduced before R2006a

resample

Resample `timeseries` time vector

Syntax

```
tsout = resample(tsin,timevec)
tsout = resample(tsin,timevec,interpmethod)
tsout = resample(tsin,timevec,interpmethod,code)
```

Description

`tsout = resample(tsin,timevec)` resamples a `timeseries` object `tsin` using a new time vector `timevec`. The `resample` function uses the interpolation method associated with `tsin`, which you can display using the command `getinterpmethod(tsin)`.

`tsout = resample(tsin,timevec,interpmethod)` resamples `tsin` using a specified interpolation method. `interpmethod` can be `'linear'` for linear interpolation or `'zoh'` for zero-order hold.

`tsout = resample(tsin,timevec,interpmethod,code)` applies the quality codes in `code` to all samples.

Examples

Resample `timeseries`

Create a `timeseries` object, and resample it using linear interpolation according to the times in `timevec`. Compare the original data to the resampled data.

```
tsin = timeseries([1.1 2.9 3.7 4.0 3.0]',1:5);
tsout = resample(tsin,[1 1.5 3.5 4.5 4.9]);
tsindata = tsin.Data
```

```
tsindata = 5×1

    1.1000
    2.9000
    3.7000
    4.0000
    3.0000
```

```
tsoutdata = tsout.Data
```

```
tsoutdata = 5×1

    1.1000
    2.0000
    3.8500
    3.5000
    3.1000
```

Input Arguments

`tsin` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

`timevec` — New time vector
vector

New time vector, specified as a vector of times for resampling.

When the original time vector contains dates and times but `timevec` is numeric, `resample` defines `timevec` relative to the `tsin.TimeInfo.StartDate` property using the existing units.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`interpmethod` — Interpolation method
`'linear'` (default) | `'zoh'`

Interpolation method, specified as one of the following options:

- `'linear'` — Linear interpolation
- `'zoh'` — Zero-order hold

code — Quality codes
[ ] (default) | scalar | vector | multidimensional array

Quality codes, specified as `[]` or a scalar, vector, or multidimensional array of integers ranging from -128 to 127.

- When the quality code value is a vector, it must have the same length as the time vector. Each element applies to the corresponding data sample.
- When the quality code value is an array, it must have the same size as the data array. Each element applies to the corresponding data element of the data array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

See Also

`getinterpmethod` | `retime` | `synchronize` | `timeseries`

Introduced before R2006a

set

Set `timeseries` properties

Syntax

```
tsprop = set(ts)
tsprop = set(ts,propname)
tsprop = set(ts,propname,propval)
```

Description

`tsprop = set(ts)` returns all property values of a `timeseries` object `ts`.

`tsprop = set(ts,propname)` returns the property specified in `propname`. This syntax is equivalent to the command `tsprop = ts.propname`. For a full list of `timeseries` property names, see `timeseries`.

`tsprop = set(ts,propname,propval)` sets the property `propname` of `ts` to the value `propval`. This syntax is equivalent to the command `ts.propname = propval`. For a full list of `timeseries` property names and corresponding valid values, see `timeseries`.

Examples

Name of timeseries

Create an unnamed `timeseries` object, and use the `set` function to assign a name to the `Name` property.

```
ts = timeseries((1:5));
tsprop = set(ts,'Name','newname')
```

```
  timeseries

  Common Properties:
```

```
    Name: 'newname'
    Time: [5x1 double]
TimeInfo: tsdata.timemetadata
    Data: [1x1x5 double]
DataInfo: tsdata.datametadata
```

Input Arguments

`ts` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

`propname` — Property name
character vector

Property name, specified as a character vector. For a full list of `timeseries` property names, see `timeseries`.

Data Types: `char`

`propval` — Property value
scalar

Property value, specified as a scalar. For a full list of `timeseries` property names and valid values, see `timeseries`.

See Also

`get` | `timeseries`

Introduced before R2006a

setabstime

Set `timeseries` times as date character vectors

Syntax

```
tsout = setabstime(tsin,timevals)
tsout = setabstime(tsin,timevals,timeformat)
```

Description

`tsout = setabstime(tsin,timevals)` sets the times in `timeseries` object `tsin` to the date character vectors specified in `timevals`.

`tsout = setabstime(tsin,timevals,timeformat)` explicitly specifies the format of `timeformat` used in `timevals`.

Examples

`timeseries` Dates

Set date character vectors as the time values in a `timeseries` object.

```
tsin = timeseries((1:3)');
tsout = setabstime(tsin,{'12-DEC-2005 12:34:56','12-DEC-2005 13:34:56','12-DEC-2005 14
```

Input Arguments

`tsin` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

`timevals` — Sample times
array

Sample times, specified as a cell array of date character vectors. Valid date character vectors can have the following forms:

Format	Example
`dd-mmm-yyyy HH:MM:SS`	`01-Mar-2000 15:45:17`
`dd-mmm-yyyy`	`01-Mar-2000`
`mm/dd/yy`	`03/01/00`
`mm/dd`	`03/01`
`HH:MM:SS`	`15:45:17`
`HH:MM:SS PM`	`3:45:17 PM`
`HH:MM`	`15:45`
`HH:MM PM`	`3:45 PM`
`mmm.dd,yyyy HH:MM:SS`	`Mar.01,2000 15:45:17`
`mmm.dd,yyyy`	`Mar.01,2000`
`mm/dd/yyyy`	`03/01/2000`

Data Types: `cell`

`timeformat` — Time format
scalar

Time format, specified as a scalar date format used for the time values.

See Also

`getabstime` | `timeseries`

Introduced before R2006a

setinterpmethod

Set default interpolation method for `timeseries` object

Syntax

```
ts = setinterpmethod(ts,interpmethod)
ts = setinterpmethod(ts,fun)
ts = setinterpmethod(ts,interpobj)
```

Description

`ts = setinterpmethod(ts,interpmethod)` sets the default interpolation method of a `timeseries` object. `interpmethod` can be `'linear'` for linear interpolation or `'zoh'` for zero-order hold.

`ts = setinterpmethod(ts,fun)` uses a function handle `fun` to set the default interpolation method of `ts`.

`ts = setinterpmethod(ts,interpobj)` uses a `tsdata.interpolation` object `interpobj` to replace the interpolation object stored in `ts`.

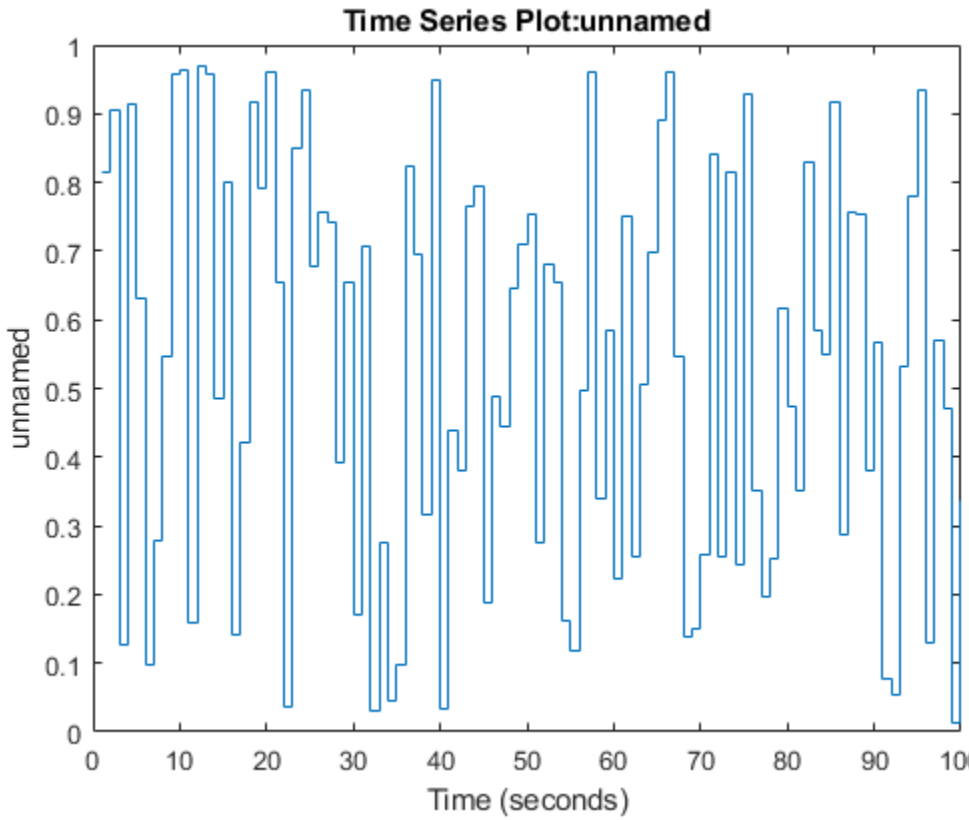
Examples

Set `timeseries` Interpolation Method

Set the interpolation method of a `timeseries` object in three ways.

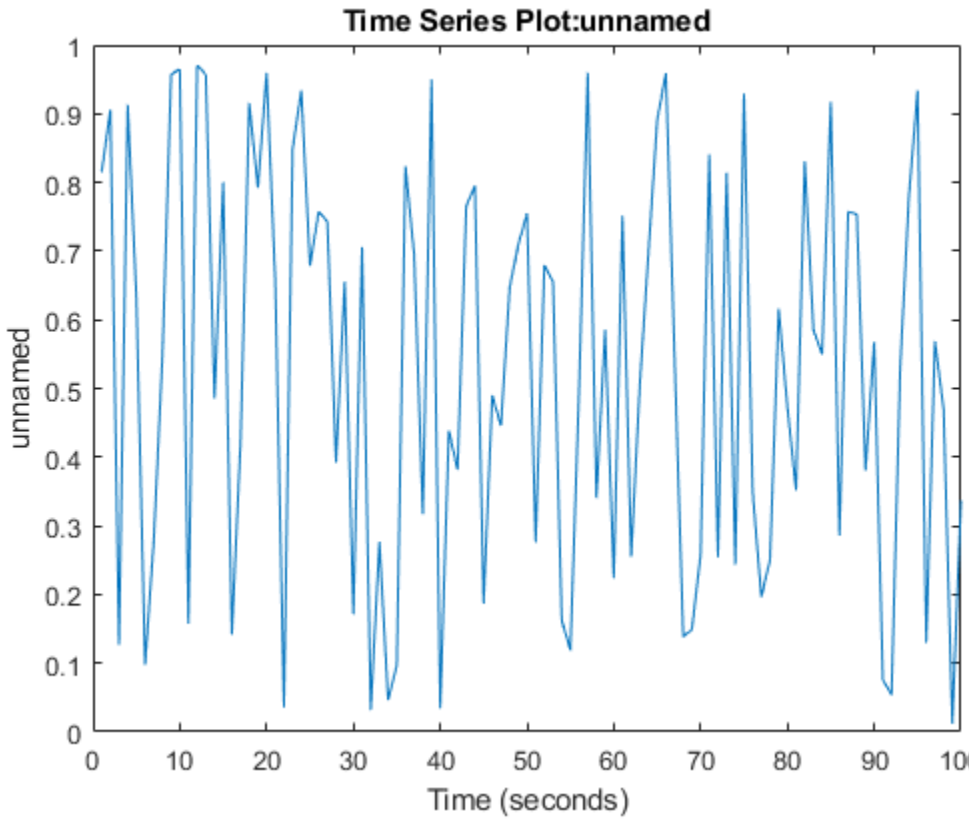
Create a `timeseries` object and set the interpolation method to zero-order hold.

```
ts = timeseries(rand(100,1),1:100);
ts = setinterpmethod(ts,'zoh');
plot(ts)
```



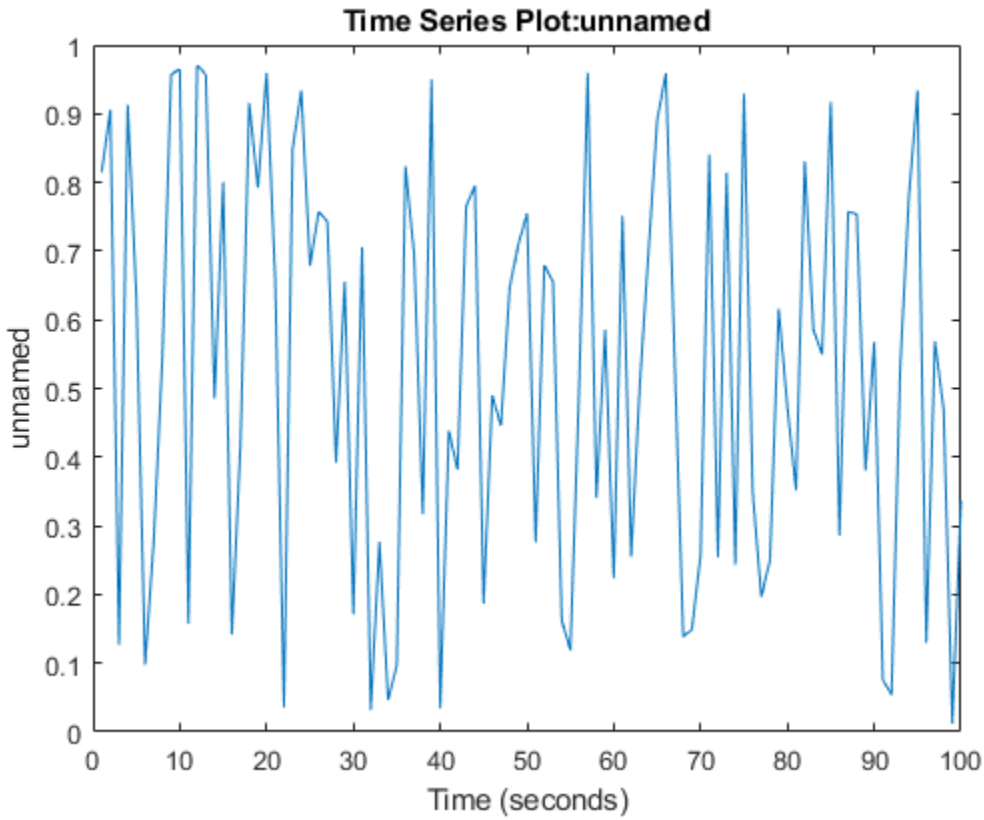
Set the default interpolation method using a function handle.

```
fun = @(newtime,oldtime,olddata)...
               interp1(oldtime,olddata,newtime,...
                       'linear','extrap');
ts = setinterpmethod(ts,fun);
plot(ts)
```



Set the default interpolation method to a `tsdata.interpolation` object.

```
interpobj = tsdata.interpolation(fun);
ts = setinterpmethod(ts,interpobj);
plot(ts)
```



Input Arguments

`ts` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

`interpmethod` — Interpolation method
`'linear'` (default) | `'zoh'`

Interpolation method, specified as one of the following options:

- `'linear'` — Linear interpolation
- `'zoh'` — Zero-order hold

`fun` — Interpolation method handle
scalar

Interpolation method handle, specified as a scalar function handle defining the interpolation method. The order of input arguments for the function handle must be `newtime`, `oldtime`, and `olddata`. A single output argument must contain the interpolated data.

Data Types: `function_handle`

`interpobj` — Interpolation object
scalar

Interpolation object, specified as a `tsdata.interpolation` object that directly replaces the interpolation object stored in `ts`.

See Also

`getinterpmethod` | `timeseries`

Introduced before R2006a

setuniformtime

Modify uniform `timeseries` time vector

Syntax

```
tsout = setuniformtime(tsin,'StartTime',starttime)
tsout = setuniformtime(tsin,'Interval',interval)
tsout = setuniformtime(tsin,'EndTime',endtime)
tsout = setuniformtime(tsin,'StartTime',starttime,'Interval',
interval)
tsout = setuniformtime(tsin,'StartTime',starttime,'EndTime',endtime)
tsout = setuniformtime(tsin,'Interval',interval,'EndTime',endtime)
```

Description

`tsout = setuniformtime(tsin,'StartTime',starttime)` returns a `timeseries` object with a modified uniform time vector determined from `starttime`. The end time is given by `starttime+(length(tsin)-1)`. The unit of time is unchanged.

`tsout = setuniformtime(tsin,'Interval',interval)` sets the start time to 0 and defines the end time as `(length(tsin)-1)*interval`.

`tsout = setuniformtime(tsin,'EndTime',endtime)` sets the start time to 0 and defines the interval as `endtime/(length(tsin)-1)`.

`tsout = setuniformtime(tsin,'StartTime',starttime,'Interval', interval)` defines the end time as `starttime+(length(tsin)-1)*interval`.

`tsout = setuniformtime(tsin,'StartTime',starttime,'EndTime', endtime)`defines the interval as `(endtime-starttime)/(length(tsin)-1)`.

`tsout = setuniformtime(tsin,'Interval',interval,'EndTime',endtime)` defines the start time as `endtime-(length(tsin)-1)*interval`.

Examples

Modify Uniform Time Vector

Create a `timeseries` object with a uniform time vector.

```
tsin = timeseries((1:5)',1:5);
tsin.Time
```

```
ans = 5×1

     1
     2
     3
     4
     5
```

Modify the time vector by specifying a new start time and end time.

```
tsout = setuniformtime(tsin,'StartTime',10,'EndTime',20);
tsout.Time
```

```
ans = 5×1

   10.0000
   12.5000
   15.0000
   17.5000
   20.0000
```

Input Arguments

`tsin` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

`starttime` — Start time
numeric scalar

Start time of the uniform time vector, specified as a numeric scalar.

`interval` — Interval
numeric scalar

Interval of the uniform time vector, specified as a numeric scalar.

`endtime` — End time
numeric scalar

End time of the uniform time vector, specified as a numeric scalar.

See Also

`timeseries`

Introduced before R2006a

synchronize

Synchronize and resample two `timeseries` objects using common time vector

Syntax

```
[ts1,ts2] = synchronize(ts1,ts2,synchronizemethod)
[ts1,ts2] = synchronize( ___ ,Name,Value)
```

Description

`[ts1,ts2] = synchronize(ts1,ts2,synchronizemethod)` creates two new `timeseries` objects by synchronizing `ts1` and `ts2` using a common time vector and the specified method.

`[ts1,ts2] = synchronize( ___ ,Name,Value)` creates the two new `timeseries` objects with additional options specified by one or more `Name,Value` pair arguments for the previous syntax.

Examples

Synchronize `timeseries` Objects

Create two `timeseries` objects such that `ts1.timeinfo.StartDate` is one day after `ts2.timeinfo.StartDate`.

```
ts1 = timeseries([1 2],[datestr(now); datestr(now+1)]);
ts2 = timeseries([1 2],[datestr(now-1); datestr(now)]);
```

Change `ts1.timeinfo.StartDate` to match `ts2.timeinfo.StartDate` and change `ts1.Time` to 1.

```
[ts1 ts2] = synchronize(ts1,ts2,'union');
```

The following command preserves `ts1.timeinfo.StartDate` and keeps `ts1.Time` as 0.

```
[ts1 ts2] = synchronize(ts1,ts2,'union','KeepOriginalTimes',true);
```

Input Arguments

ts1 — First input timeseries
scalar

First input `timeseries`, specified as a scalar.

Data Types: `timeseries`

ts2 — Second input timeseries
scalar

Second input `timeseries`, specified as a scalar.

Data Types: `timeseries`

synchronizemethod — Synchronize method
`'Union'` | `'Intersection'` | `'Uniform'`

Synchronize method, specified as one of the following options:

- `'Union'` — Resample `timeseries` objects using a time vector that is a union of the time vectors of `ts1` and `ts2` on the time range where the two time vectors overlap.
- `'Intersection'` — Resample `timeseries` objects on a time vector that is the intersection of the time vectors of `ts1` and `ts2`.
- `'Uniform'` — Requires an additional argument as follows:

  ```
  [ts1,ts2] = synchronize(ts1,ts2,'Uniform','Interval',value)
  ```

  This method resamples time series on a uniform time vector, where `value` specifies the time interval between two consecutive samples. The uniform time vector is the overlap of the time vectors of `ts1` and `ts2`. The interval units are the smaller units of `ts1` and `ts2`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[ts1,ts2] = synchronize(ts1,ts2,'tolerance',1e-13)`

`InterpMethod` — Interpolation method
`'linear'` (default) | `'zoh'`

Interpolation method for the synchronize operation, specified as `'linear'` for linear interpolation or `'zoh'` for zero-order hold. The interpolation method can also be specified as `tsdata.interpolation` object containing a user-defined interpolation method.

`QualityCode` — Quality codes
`[]` (default) | scalar | vector | multidimensional array

Quality codes, specified as `[]` or a scalar, vector, or multidimensional array of integers ranging from -128 to 127.

- When the quality code value is a vector, it must have the same length as the time vector. Each element applies to the corresponding data sample.
- When the quality code value is an array, it must have the same size as the data array. Each element applies to the corresponding data element of the data array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`KeepOriginalTimes` — Original time indicator
`true` | `false`

Original time vector, specified as either `true` to keep the original time values or `false` to use different time values.

Data Types: `logical`

`tolerance` — Tolerance
`1e-10` (default) | real numeric scalar

Tolerance, specified as a real numeric scalar defining the tolerance for differentiating between two time values of `ts1` and `ts2`. For example, when the sixth time value in `ts1` is `5+(1e-12)` and the sixth time value in `ts2` is `5-(1e-13)`, both values are treated as 5 by default. To differentiate the two times, you can set `'tolerance'` to a smaller value such as `1e-15`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`ts1` — First output `timeseries`
scalar

First output `timeseries` that was synchronized and resampled, specified as a scalar.

Data Types: `timeseries`

`ts2` — Second output `timeseries`
scalar

Second output `timeseries` that was synchronized and resampled, specified as a scalar.

Data Types: `timeseries`

See Also

`set` | `synchronize` | `timeseries`

Introduced before R2006a

std

Standard deviation of `timeseries` data

Syntax

```
tsstd = std(ts)
tsstd = std(ts,Name,Value)
```

Description

`tsstd = std(ts)` returns the standard deviation of the data in a `timeseries` object.

`tsstd = std(ts,Name,Value)` specifies additional options when computing the standard deviation using one or more name-value pair arguments. For example, `tsstd = std(ts,'Quality',-99,'MissingData','remove')` defines -99 as the missing sample quality code, and removes the missing samples before computing the standard deviation.

Examples

Standard Deviation of Sample Data

Create a `timeseries` object and compute the standard deviation of the sample data.

```
ts = timeseries((1:10)');
tsstd = std(ts)
```

```
tsstd = 3.0277
```

Input Arguments

`ts` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `tsstd = std(ts,'Quality',-99,'MissingData','remove')`

`Quality` — Missing value indicator
scalar | vector | matrix | multidimensional array

Missing value indicator, specified a scalar, vector, matrix, or multidimensional array of integers ranging from -128 to 127. Each element is a quality code to treat as missing data.

By default, missing data is removed before computing. To interpolate the data instead of removing it, specify the name-value pair `'MissingData','interpolation'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`MissingData` — Missing data method
`'remove'` (default) | `'interpolate'`

Missing data method, specified as either `'remove'` to remove missing values or `'interpolate'` to fill missing values by interpolating the data. Specify the `'Quality'` name-value pair to indicate which data samples are considered missing.

`Weighting` — Weights
`'none'` (default) | `'time'`

Weights, specified as `'none'` or `'time'`.

When you specify `'time'`, larger time values correspond to larger weights.

Algorithms

MATLAB determines weighting by:

1. Attaching a weighting to each time value, depending on its order, as follows:
   - First time point — The duration of the first time interval `(t(2) - t(1))`.
   - Time point that is neither the first nor last time point — The duration between the midpoint of the previous time interval to the midpoint of the subsequent time interval `((t(k + 1) - t(k))/2 + (t(k) - t(k - 1))/2)`.
   - Last time point — The duration of the last time interval `(t(end) - t(end - 1))`.
2. Normalizing the weighting for each time by dividing each weighting by the mean of all weightings.

   Note If the `timeseries` object is uniformly sampled, then the normalized weighting for each time is 1.0. Therefore, time weighting has no effect.
3. Multiplying the data for each time by its normalized weighting.

See Also

`iqr` | `mean` | `timeseries` | `var`

Introduced before R2006a

sum

Sum of `timeseries` data

Syntax

```
tssum = sum(ts)
tssum = sum(ts,Name,Value)
```

Description

`tssum = sum(ts)` returns the sum of the data samples in a `timeseries` object.

`tssum = sum(ts,Name,Value)` specifies additional options when computing the sum using one or more name-value pair arguments. For example, `tssum = sum(ts,'Quality',-99,'MissingData','remove')` defines -99 as the missing sample quality code, and removes the missing samples before computing the sum.

Examples

Sum of Sample Data

Create a `timeseries` object and compute the sum of the sample data.

```
ts = timeseries((1:5)');
tssum = sum(ts)
```

```
tssum = 15
```

Input Arguments

`ts` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `tssum = sum(ts,'Quality',-99,'MissingData','remove')`

`Quality` — Missing value indicator
scalar | vector | matrix | multidimensional array

Missing value indicator, specified a scalar, vector, matrix, or multidimensional array of integers ranging from -128 to 127. Each element is a quality code to treat as missing data.

By default, missing data is removed before computing. To interpolate the data instead of removing it, specify the name-value pair `'MissingData','interpolation'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`MissingData` — Missing data method
`'remove'` (default) | `'interpolate'`

Missing data method, specified as either `'remove'` to remove missing values or `'interpolate'` to fill missing values by interpolating the data. Specify the `'Quality'` name-value pair to indicate which data samples are considered missing.

`Weighting` — Weights
`'none'` (default) | `'time'`

Weights, specified as `'none'` or `'time'`.
When you specify `'time'`, larger time values correspond to larger weights.

Algorithms

MATLAB determines weighting by:

1. Attaching a weighting to each time value, depending on its order, as follows:
   - First time point — The duration of the first time interval `(t(2) - t(1))`.
   - Time point that is neither the first nor last time point — The duration between the midpoint of the previous time interval to the midpoint of the subsequent time interval `((t(k + 1) - t(k))/2 + (t(k) - t(k - 1))/2)`.
   - Last time point — The duration of the last time interval `(t(end) - t(end - 1))`.
2. Normalizing the weighting for each time by dividing each weighting by the mean of all weightings.

   Note If the `timeseries` object is uniformly sampled, then the normalized weighting for each time is 1.0. Therefore, time weighting has no effect.
3. Multiplying the data for each time by its normalized weighting.

See Also

`mean` | `timeseries`

Introduced before R2006a

var

Variance of `timeseries` data

Syntax

```
tsvar = var(ts)
tsvar = var(ts,Name,Value)
```

Description

`tsvar = var(ts)` returns the variance of the data samples in a `timeseries` object.

`tsvar = var(ts,Name,Value)` specifies additional options when computing the variance using one or more name-value pair arguments. For example, `tsvar = var(ts,'Quality',-99,'MissingData','remove')` defines -99 as the missing sample quality code, and removes the missing samples before computing the variance.

Examples

Variance of Sample Data

Create a `timeseries` object and compute the variance of the sample data.

```
ts = timeseries((1:10)');
tsvar = var(ts)
```

```
tsvar = 9.1667
```

Input Arguments

`ts` — Input `timeseries`
scalar

Input `timeseries`, specified as a scalar.

Data Types: `timeseries`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `tsvar = var(ts,'Quality',-99,'MissingData','remove')`

`Quality` — Missing value indicator
scalar | vector | matrix | multidimensional array

Missing value indicator, specified a scalar, vector, matrix, or multidimensional array of integers ranging from -128 to 127. Each element is a quality code to treat as missing data.

By default, missing data is removed before computing. To interpolate the data instead of removing it, specify the name-value pair `'MissingData','interpolation'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`MissingData` — Missing data method
`'remove'` (default) | `'interpolate'`

Missing data method, specified as either `'remove'` to remove missing values or `'interpolate'` to fill missing values by interpolating the data. Specify the `'Quality'` name-value pair to indicate which data samples are considered missing.

`Weighting` — Weights
`'none'` (default) | `'time'`

Weights, specified as `'none'` or `'time'`.
When you specify `'time'`, larger time values correspond to larger weights.

Algorithms

MATLAB determines weighting by:

1. Attaching a weighting to each time value, depending on its order, as follows:
   - First time point — The duration of the first time interval `(t(2) - t(1))`.
   - Time point that is neither the first nor last time point — The duration between the midpoint of the previous time interval to the midpoint of the subsequent time interval `((t(k + 1) - t(k))/2 + (t(k) - t(k - 1))/2)`.
   - Last time point — The duration of the last time interval `(t(end) - t(end - 1))`.
2. Normalizing the weighting for each time by dividing each weighting by the mean of all weightings.

   Note If the `timeseries` object is uniformly sampled, then the normalized weighting for each time is 1.0. Therefore, time weighting has no effect.
3. Multiplying the data for each time by its normalized weighting.

See Also

`iqr` | `mean` | `std` | `timeseries`

Introduced before R2006a

triangulation

Triangulation in 2-D or 3-D

Description

Use `triangulation` to create an in-memory representation of any 2-D or 3-D triangulation data that is in matrix format, such as the matrix output from the `delaunay` function or other software tools. When your data is represented using `triangulation`, you can perform topological and geometric queries, which you can use to develop geometric algorithms. For example, you can find the triangles or tetrahedra attached to a vertex, those that share an edge, their circumcenters, and other features.

Creation

To create a `triangulation` object, use the `triangulation` function with input arguments that define the triangulation's points and connectivity.

Syntax

```
TR = triangulation(T,P)
TR = triangulation(T,x,y)
TR = triangulation(T,x,y,z)
```

Description

`TR = triangulation(T,P)` creates a 2-D or 3-D triangulation representation using the triangulation connectivity list `T` and the points in matrix `P`.

`TR = triangulation(T,x,y)` creates a 2-D triangulation representation with the point coordinates specified as column vectors `x` and `y`.

`TR = triangulation(T,x,y,z)` creates a 3-D triangulation representation with the point coordinates specified as column vectors `x`, `y`, and `z`.

Input Arguments

T — Triangulation connectivity list
m-by-n matrix

Triangulation connectivity list, specified as an m-by-n matrix, where m is the number of triangles or tetrahedra, and n is the number of vertices per triangle or tetrahedron. Each row of T contains the vertex IDs that define a triangle or tetrahedron. The vertex IDs are the row numbers of the input points. The ID of a triangle or tetrahedron in the triangulation is the corresponding row number in T.

P — Points
matrix

Points, specified as a matrix whose columns are the *x*-coordinates, *y*-coordinates, and (possibly) z-coordinates of the triangulation points. The row numbers of P are the vertex IDs in the triangulation.

x — *x*-coordinates
column vector

x-coordinates of triangulation points, specified as a column vector.

y — *y*-coordinates
column vector

y-coordinates of triangulation points, specified as a column vector.

z — *z*-coordinates
column vector

z-coordinates of triangulation points, specified as a column vector.

Properties

`Points` — Triangulation points
matrix

Triangulation points, represented as a matrix with the following characteristics:

- Each row in `TR.Points` contains the coordinates of a vertex.
- Each row number of `TR.Points` is a vertex ID.

`ConnectivityList` — Triangulation connectivity list
matrix

Triangulation connectivity list, represented as a matrix with the following characteristics:

- Each element in `TR.ConnectivityList` is a vertex ID.
- Each row represents a triangle or tetrahedron in the triangulation.
- Each row number of `TR.ConnectivityList` is a triangle or tetrahedron ID.

Object Functions

barycentricToCartesian	Convert coordinates from barycentric to Cartesian
cartesianToBarycentric	Convert coordinates from Cartesian to barycentric
circumcenter	Circumcenter of triangle or tetrahedron
edgeAttachments	Triangles or tetrahedra attached to specified edge
edges	Triangulation edges
faceNormal	Triangulation unit normal vectors
featureEdges	Handle sharp edges of triangulation
freeBoundary	Free boundary facets
incenter	Incenter of triangulation elements
isConnected	Test if two vertices are connected by an edge
nearestNeighbor	Closest vertex
neighbors	Triangle or tetrahedron neighbors
pointLocation	Triangle or tetrahedron enclosing point
size	Size of triangulation connectivity list
vertexAttachments	Triangles or tetrahedra attached to vertex
vertexNormal	Triangulation vertex normal

Examples

2-D Triangulation

Define and plot the points in a 2-D triangulation.

```
P = [ 2.5    8.0
      6.5    8.0
      2.5    5.0
      6.5    5.0
      1.0    6.5
      8.0    6.5];
```

Define the triangulation connectivity list.

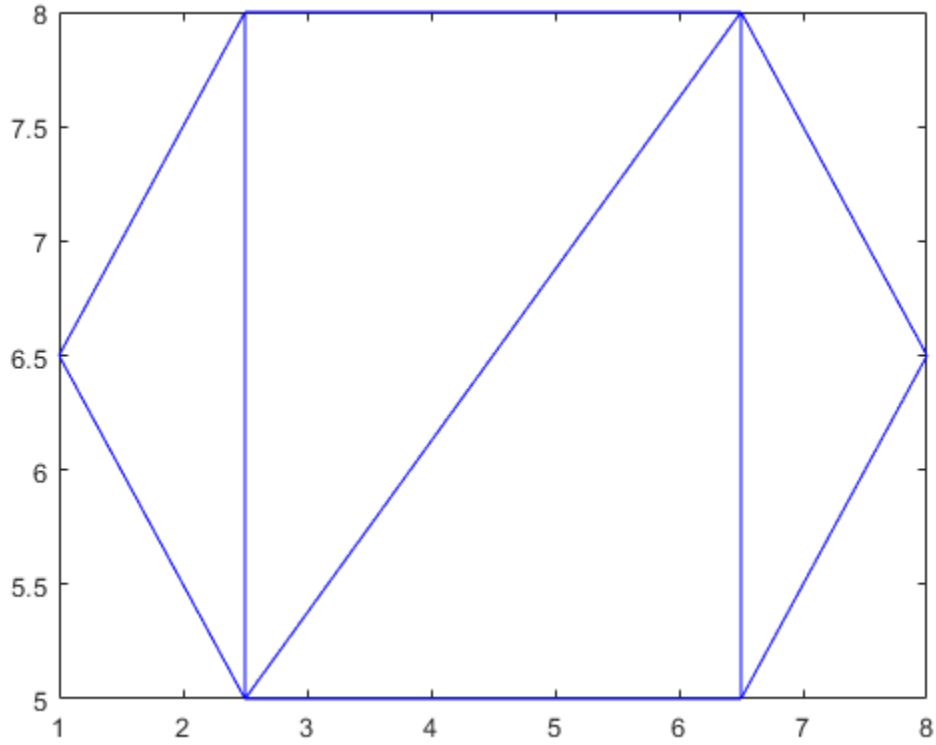
```
T = [5  3  1;
     3  2  1;
     3  4  2;
     4  6  2];
```

Create and plot the triangulation representation.

```
TR = triangulation(T,P)
```

```
TR = 
  triangulation with properties:

              Points: [6x2 double]
    ConnectivityList: [4x3 double]
```

```
triplot(TR)
```



Examine the coordinates of the vertices of the first triangle.

```
TR.Points(TR.ConnectivityList(1,:),:)
```

```
ans = 3×2

    1.0000    6.5000
    2.5000    5.0000
    2.5000    8.0000
```

See Also

`delaunayTriangulation`

Topics

"Triangulation Representations"

Introduced in R2013a

barycentricToCartesian

Convert coordinates from barycentric to Cartesian

Syntax

```
C = barycentricToCartesian(TR,ID,B)
```

Description

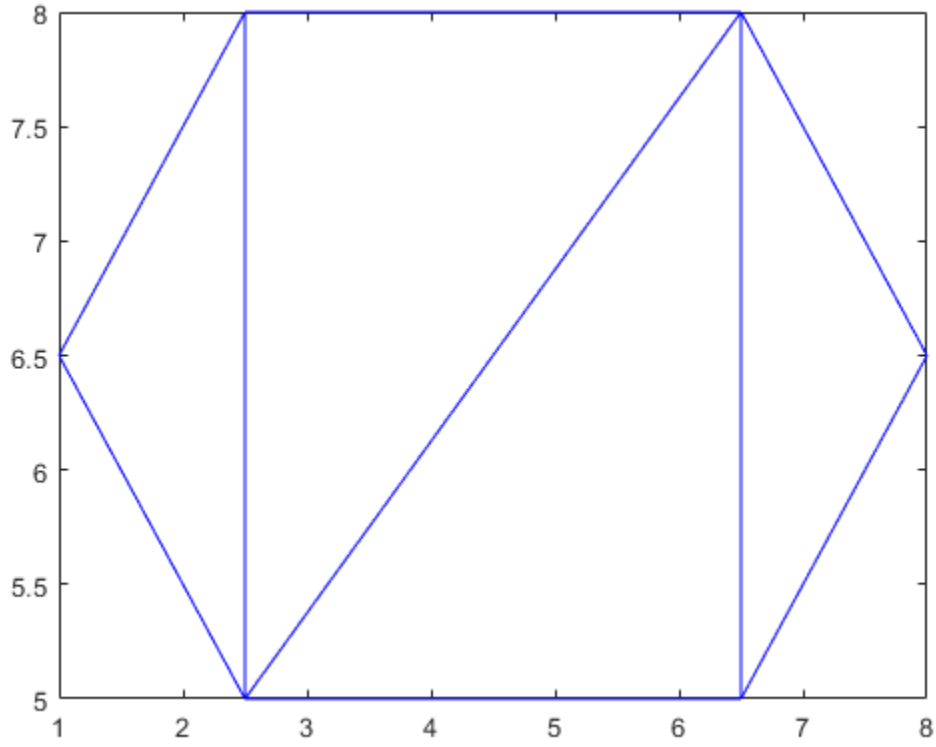
`C = barycentricToCartesian(TR,ID,B)` returns the Cartesian coordinates of the points in `B` relative to the triangulation object `TR`. Each row of `B` contains the barycentric coordinates of a point with respect to the triangle or tetrahedron indexed by `ID`. The identification numbers of the triangles or tetrahedra in `TR` are the corresponding row numbers of the property `TR.ConnectivityList`.

Examples

Convert Barycentric Coordinates to Cartesian Coordinates

Create a triangulation from a set of points `P` and a triangulation connectivity list `T`, and plot the triangulation.

```
P = [2.5 8.0; 6.5 8.0; 2.5 5.0; 6.5 5.0; 1.0 6.5; 8.0 6.5];
T = [5 3 1; 3 2 1; 3 4 2; 4 6 2];
TR = triangulation(T,P);
triplot(TR)
```



Specify the identification number of the first (leftmost) triangle in TR, and the barycentric coordinates of the triangle's second point.

```
ID = 1;
B = [0 1 0];
```

Convert the barycentric coordinates to Cartesian coordinates with respect to the first triangle in TR.

```
C = barycentricToCartesian(TR,ID,B)
```

```
C = 1×2

    2.5000    5.0000
```

Input Arguments

TR — Triangulation representation
scalar triangulation object

Triangulation representation, specified as a scalar `triangulation` or `delaunayTriangulation` object.

Data Types: `triangulation` | `delaunayTriangulation`

ID — Triangle or tetrahedron identification
scalar | column vector

Triangle or tetrahedron identification, specified as a scalar or a column vector whose elements each correspond to a single triangle or tetrahedron in the triangulation object. The identification number of each triangle or tetrahedron is the corresponding row number of the `ConnectivityList` property.

Data Types: `double`

B — Barycentric coordinates
matrix

Barycentric coordinates, specified as a three-column matrix for 2-D coordinates or a four-column matrix for 3-D coordinates.

Data Types: `double`

See Also

`cartesianToBarycentric` | `delaunayTriangulation` | `triangulation`

Introduced in R2013a

cartesianToBarycentric

Convert coordinates from Cartesian to barycentric

Syntax

```
B = cartesianToBarycentric(TR,ID,C)
```

Description

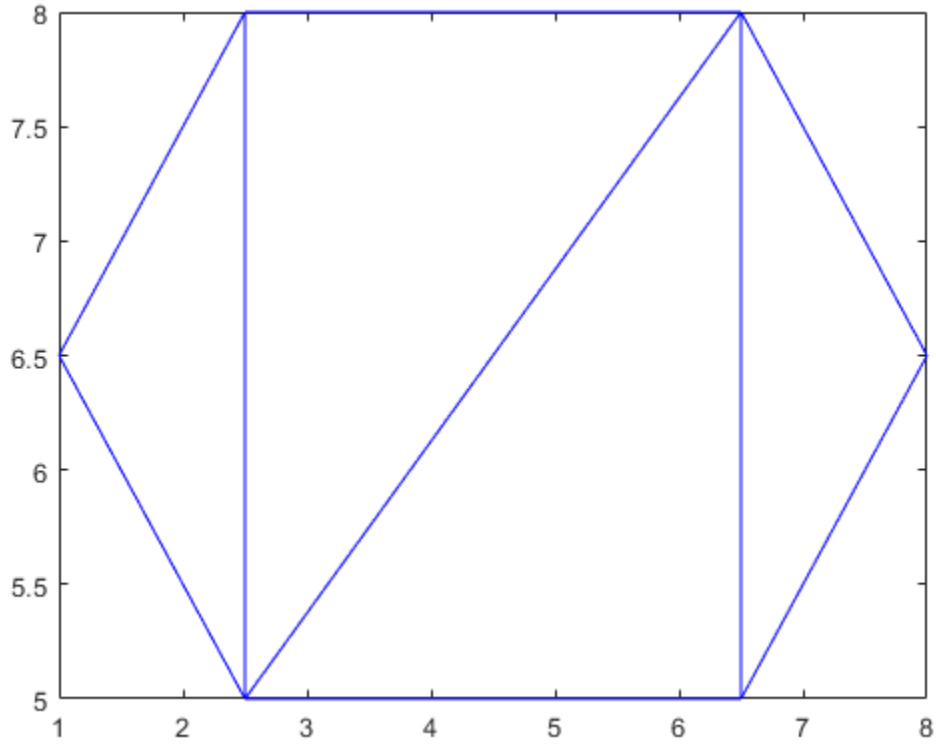
`B = cartesianToBarycentric(TR,ID,C)` returns the barycentric coordinates of the points in `C` relative to the triangulation object `TR`. Each row of `C` contains the Cartesian coordinates of a point with respect to the triangle or tetrahedron indexed by `ID`. The identification numbers of the triangles or tetrahedra in `TR` are the corresponding row numbers of the property `TR.ConnectivityList`.

Examples

Convert Cartesian Coordinates to Barycentric Coordinates

Create a triangulation from a set of points `P` and triangulation connectivity list `T`, and plot the triangulation.

```
P = [2.5 8.0; 6.5 8.0; 2.5 5.0; 6.5 5.0; 1.0 6.5; 8.0 6.5];
T = [5 3 1; 3 2 1; 3 4 2; 4 6 2];
TR = triangulation(T,P);
triplot(TR)
```



Find the Cartesian coordinates of the third vertex in the first (leftmost) triangle in TR.

```
L = TR.ConnectivityList(1,3);
C = TR.Points(L,:)
```

```
C = 1×2

    2.5000    8.0000
```

Convert the point C to barycentric coordinates with respect to the first triangle.

```
B = cartesianToBarycentric(TR,1,C)
```

```
B = 1×3

     0     0     1
```

Input Arguments

TR — Triangulation representation
scalar triangulation object

Triangulation representation, specified as a scalar `triangulation` or `delaunayTriangulation` object.

Data Types: `triangulation` | `delaunayTriangulation`

ID — Triangle or tetrahedron identification
scalar | column vector

Triangle or tetrahedron identification, specified as a scalar or a column vector whose elements each correspond to a single triangle or tetrahedron in the triangulation object. The identification number of each triangle or tetrahedron is the corresponding row number of the `ConnectivityList` property.

Data Types: `double`

C — Cartesian coordinates
matrix

Cartesian coordinates, specified as a two-column matrix for 2-D coordinates or a three-column matrix for 3-D coordinates.

Data Types: `double`

See Also

`barycentricToCartesian` | `delaunayTriangulation` | `triangulation`

Introduced in R2013a

circumcenter

Circumcenter of triangle or tetrahedron

Syntax

```
C = circumcenter(TR)
C = circumcenter(TR,ID)
[C,r] = circumcenter( ___ )
```

Description

`C = circumcenter(TR)` returns the coordinates of the circumcenters for each triangle or tetrahedron in the triangulation `TR`.

`C = circumcenter(TR,ID)` returns the coordinates of the circumcenters for the triangles or tetrahedra indexed by `ID`. The identification numbers of the triangles or tetrahedra in `TR` are the corresponding row numbers of the property `TR.ConnectivityList`.

`[C,r] = circumcenter( ___ )` also returns the radii of the circumscribed circles or spheres in `TR` for either of the previous syntaxes.

Examples

Compute Circumcenters of 2-D Triangulation

Load 2-D triangulation data and create a triangulation representation.

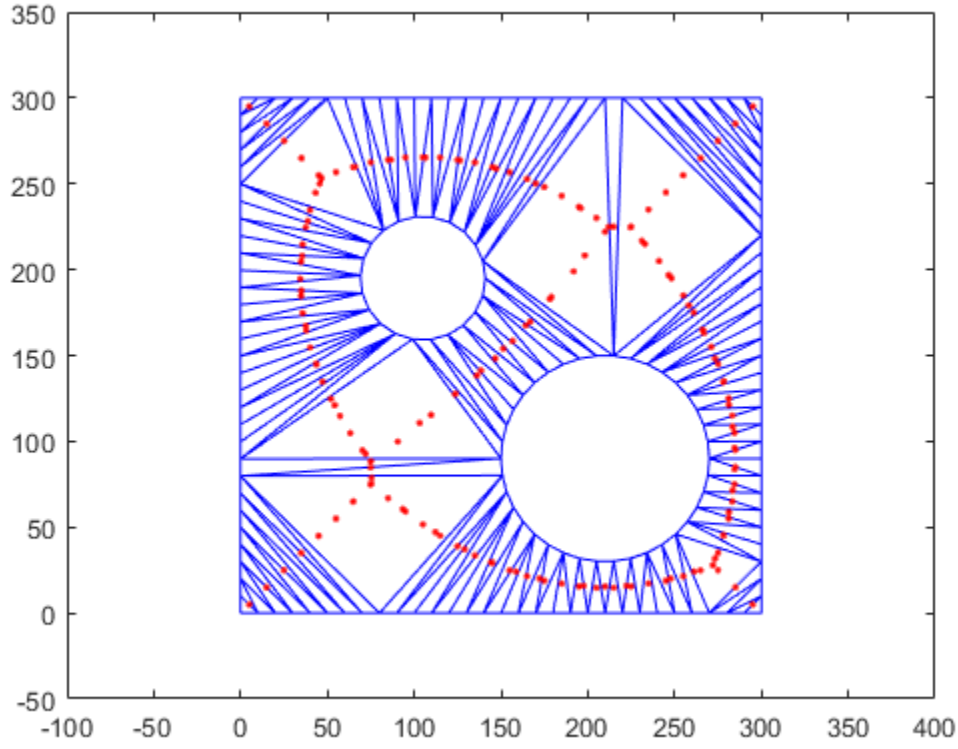
```
load trimesh2d
TR = triangulation(tri,x,y);
```

Compute the circumcenters of each triangle in `TR`.

```
C = circumcenter(TR);
```

Plot the triangulation along with the circumcenters in red. The *x*-coordinates of the circumcenters are contained in the first column of `C` and the corresponding *y*-coordinates are contained in the second column.

```
triplot(TR)
axis([-100 400 -50 350])
hold on
plot(C(:,1),C(:,2),'r.')
hold off
```



Radii of Circumscribed Spheres

Create a Delaunay triangulation for a set of points.

```
P = gallery('uniformdata',10,3,0);
TR = delaunayTriangulation(P);
```

Compute the circumcenters of the first five tetrahedra in TR, and the radii of their circumscribed spheres.

```
[C,r] = circumcenter(TR,[1:5]')
```

```
C = 5×3

    0.9626    0.3892    0.0928
    6.3458    0.2377    3.1814
    0.4820    0.9064    0.5176
   -1.2993    1.8384   -1.2185
   -0.1595    1.0852   -0.2536


r = 5×1

    0.2292
    6.2460
    0.3212
    2.4303
    0.7787
```

Input Arguments

TR — Triangulation representation
scalar triangulation object

Triangulation representation, specified as a scalar `triangulation` or `delaunayTriangulation` object.

Data Types: `triangulation` | `delaunayTriangulation`

ID — Triangle or tetrahedron identification
scalar | column vector

Triangle or tetrahedron identification, specified as a scalar or a column vector whose elements each correspond to a single triangle or tetrahedron in the triangulation object. The identification number of each triangle or tetrahedron is the corresponding row number of the `ConnectivityList` property.

Data Types: `double`

Output Arguments

`C` — Circumcenters
matrix

Circumcenters of triangles or tetrahedra, returned as a two-column matrix for 2-D coordinates or a three-column matrix for 3-D coordinates.

Data Types: `double`

`r` — Radii
scalar | vector

Radii of the circumscribed circles or spheres, returned as a scalar or vector.

Data Types: `double`

See Also

`delaunayTriangulation` | `incenter` | `triangulation`

Introduced in R2013a

edgeAttachments

Triangles or tetrahedra attached to specified edge

Syntax

```
ID = edgeAttachments(TR,startID,endID)
ID = edgeAttachments(TR,E)
```

Description

`ID = edgeAttachments(TR,startID,endID)` identifies the triangles or tetrahedra attached to the specified edges. `startID` and `endID` specify edges by their start and end points, where each point is represented by its row number in `TR.Points`. The return value `ID` identifies triangles by their identification numbers.

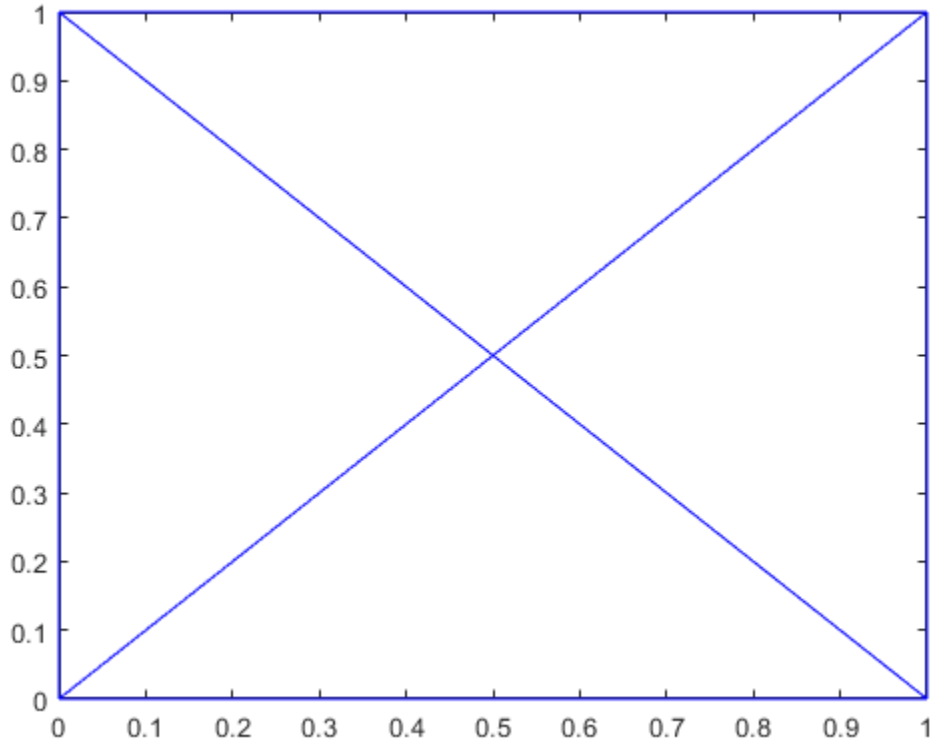
`ID = edgeAttachments(TR,E)` specifies the starting and ending vertices of each edge in a two-column matrix `E`.

Examples

2-D Delaunay Triangulation

Create and plot a Delaunay triangulation.

```
x = [0 1 1 0 0.5]';
y = [0 0 1 1 0.5]';
TR = delaunayTriangulation(x,y);
triplot(TR)
```



Compute the identifications of the triangles attached to the edge defined by the starting point (0,0) and ending point (0.5,0.5). The respective vertex identifications of these points are 1 and 5.

```
ID = edgeAttachments(TR,1,5);
ID{:}
```

```
ans = 1×2

     4     1
```

The `ConnectivityList` property indicates which vertices belong to each triangle. The fourth triangle is defined by vertices (0.5,0.5), (0,0), and (1,0), and the first triangle is defined by vertices (0,1), (0,0), and (0.5,0.5).

```
TR.ConnectivityList
```

```
ans = 4×3

     4     1     5
     4     5     3
     5     2     3
     5     1     2
```

```
TR.Points
```

```
ans = 5×2

         0         0
    1.0000         0
    1.0000    1.0000
         0    1.0000
    0.5000    0.5000
```

3-D Triangulation

Load 2-D triangulation data and create a triangulation representation.

```
load tetmesh
TR = triangulation(tet,X);
```

Select two edges by their starting and ending vertex identifications.

```
startID = [15; 21];
endID = [936; 716];
```

Find the edge attachments, and examine the identifications of the triangles attached to each edge.

```
ID = edgeAttachments(TR,startID,endID);
ID{1}
```

```
ans = 1×6

         927        2060        3438        3423        2583        4690
```

```
ID{2}
```

```
ans = 1×5

        2652        3946        3953        4665        4218
```

Input Arguments

TR — Triangulation representation
scalar triangulation object

Triangulation representation, specified as a scalar `triangulation` or `delaunayTriangulation` object.

Data Types: `triangulation` | `delaunayTriangulation`

`startID` — Starting vertex identification
column vector

Starting vertex identification for each edge, specified as a column vector. A vertex identification number is the number of the row in the `Points` property that corresponds to the vertex.

Data Types: `double`

`endID` — Ending vertex identification
column vector

Ending vertex identification for each edge, specified as a column vector. A vertex identification number is the number of the row in the `Points` property that corresponds to the vertex.

Data Types: `double`

E — Edge matrix
two-column matrix

Edge matrix, specified as a two-column matrix. Each row contains the starting and ending vertex identifications for an edge. A vertex identification number is the number of the row in the `Points` property that corresponds to the vertex.

Data Types: `double`

See Also

`delaunayTriangulation` | `edges` | `triangulation` | `vertexAttachments`

Introduced in R2013a

edges

Triangulation edges

Syntax

```
E = edges(TR)
```

Description

`E = edges(TR)` returns the triangulation edges as a two-column matrix of vertex identification numbers. Vertex identifications are the row numbers of the triangulation vertices in `TR.Points`. The first column of `E` contains the starting vertex identification of each edge, and the second column contains the ending vertex identification.

Examples

Find Edges of 2-D Triangulation

Load 2-D triangulation data and create a triangulation representation.

```
P = [2.5 8.0; 6.5 8.0; 2.5 5.0; 6.5 5.0; 1.0 6.5; 8.0 6.5];
T = [5 3 1; 3 2 1; 3 4 2; 4 6 2];
TR = triangulation(T,P);
```

Find the starting and ending vertex identification numbers of the edges in the triangulation.

```
E = edges(TR)
```

```
E = 9×2

     1     2
     1     3
     1     5
     2     3
```

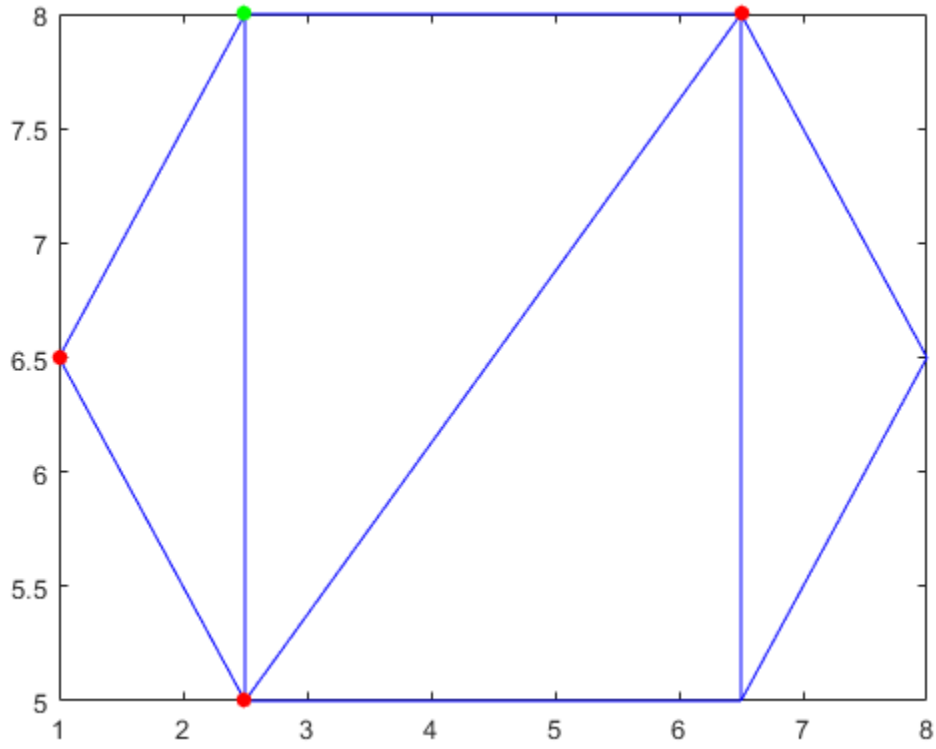
```
     2     4
     2     6
     3     4
     3     5
     4     6
```

Define the coordinates of the first point in the triangulation. Then define the coordinates of the three vertices sharing an edge with the first point (the second, third, and fifth).

```
startVert = TR.Points(1,:);
endVert = TR.Points([2 3 5],:);
```

Plot the triangulation. Plot the first vertex in green, and plot the connecting points along an edge in red.

```
triplot(TR)
hold on
plot(startVert(1,1),startVert(1,2),'g.','MarkerSize',20)
plot(endVert(:,1),endVert(:,2),'r.','MarkerSize',20)
hold off
```



Input Arguments

TR — Triangulation representation
scalar triangulation object

Triangulation representation, specified as a scalar `triangulation` or `delaunayTriangulation` object.

Data Types: `triangulation` | `delaunayTriangulation`

See Also

`delaunayTriangulation` | `edgeAttachments` | `triangulation`

Introduced in R2013a

faceNormal

Triangulation unit normal vectors

Syntax

```
F = faceNormal(TR)
F = faceNormal(TR,ID)
```

Description

`F = faceNormal(TR)` returns the unit normal vectors to all triangles in a 2-D triangulation. The `faceNormal` function supports 2-D triangulations only. `F` is a three-column matrix where each row contains the unit normal coordinates corresponding to a triangle in `TR.ConnectivityList`.

`F = faceNormal(TR,ID)` returns the unit normal vector to each triangle indexed by `ID`. The identification numbers of the triangles in `TR` are the corresponding row numbers of the property `TR.ConnectivityList`.

Examples

Unit Normals on a Surface

Compute and plot the unit normal vectors to the facets of a triangulation on a spherical surface.

Create a set of points on a spherical surface.

```
theta = gallery('uniformdata',[100,1],0)*2*pi;
phi = gallery('uniformdata',[100,1],1)*pi;
x = cos(theta).*sin(phi);
y = sin(theta).*sin(phi);
z = cos(phi);
```

Triangulate the sphere using the `delaunayTriangulation` function.

```
DT = delaunayTriangulation(x,y,z);
```

Find the free boundary facets of the triangulation, and use them to create a 2-D triangulation on the surface.

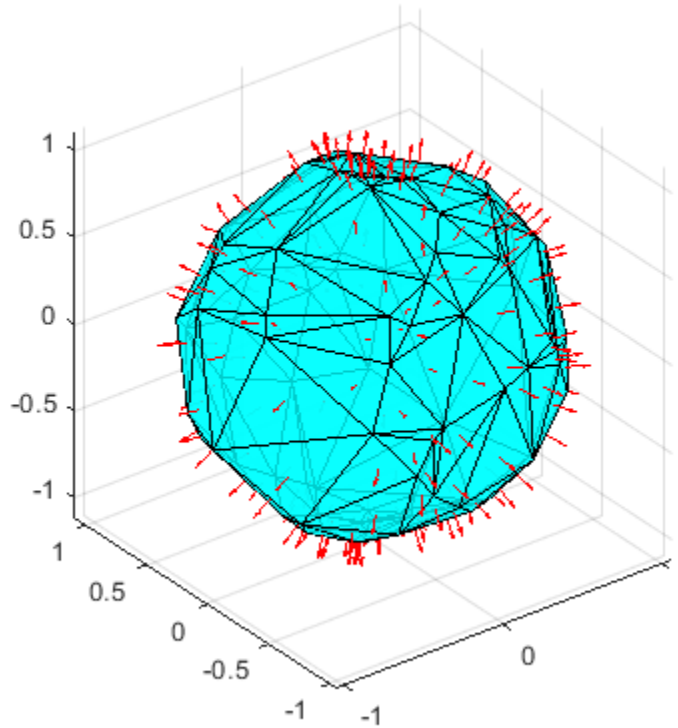
```
[T,Xb] = freeBoundary(DT);
TR = triangulation(T,Xb);
```

Compute the centers and face normals of each triangular facet in TR.

```
P = incenter(TR);
F = faceNormal(TR);
```

Plot the triangulation along with the centers and face normals.

```
trisurf(T,Xb(:,1),Xb(:,2),Xb(:,3), ...
     'FaceColor','cyan','FaceAlpha',0.8);
axis equal
hold on
quiver3(P(:,1),P(:,2),P(:,3), ...
     F(:,1),F(:,2),F(:,3),0.5,'color','r');
```



Input Arguments

TR — Triangulation representation
scalar triangulation object

Triangulation representation for 2-D triangulations only, specified as a scalar `triangulation` or `delaunayTriangulation` object.

Data Types: `triangulation` | `delaunayTriangulation`

ID — Triangle identification
scalar | column vector

Triangle identification, specified as a scalar or a column vector whose elements each correspond to a single triangle in the triangulation object. The identification number of each triangle is the corresponding row number of the `ConnectivityList` property.

Data Types: `double`

See Also

`delaunayTriangulation` | `freeBoundary` | `triangulation`

Introduced in R2013a

featureEdges

Handle sharp edges of triangulation

Syntax

```
F = featureEdges(TR,theta)
```

Description

`F = featureEdges(TR,theta)` returns the feature edges on page 1-13013 in a 2-D triangulation according to an angle `theta`. This function is typically used to extract sharp edges in a surface mesh plot.

Examples

Surface Feature Edges

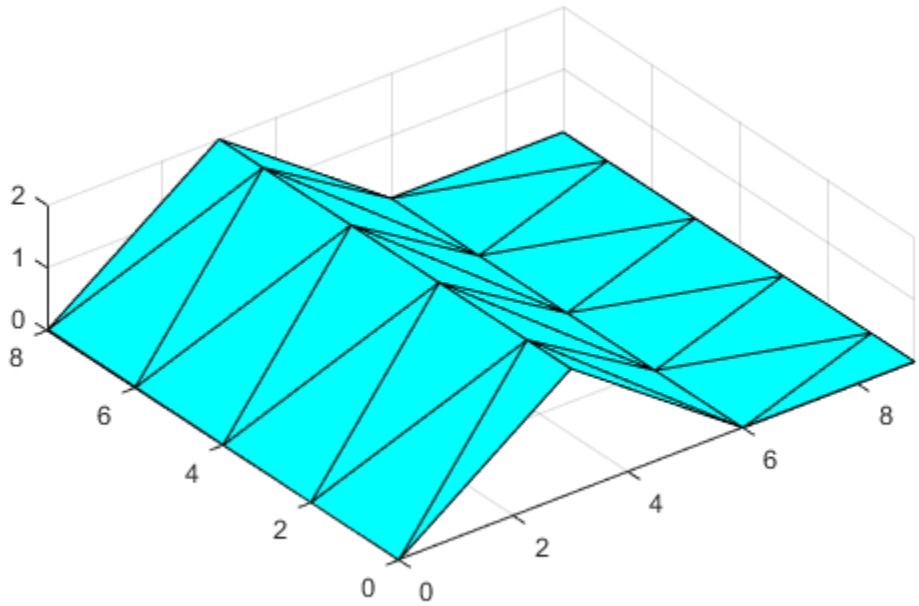
Find and plot the feature edges of a surface.

Create a 2-D Delaunay triangulation.

```
x = [0 0 0 0 0 3 3 3 3 3 3 6 6 6 6 6 9 9 9 9 9 9]';
y = [0 2 4 6 8 0 1 3 5 7 8 0 2 4 6 8 0 1 3 5 7 8]';
DT = delaunayTriangulation(x,y);
T = DT.ConnectivityList;
```

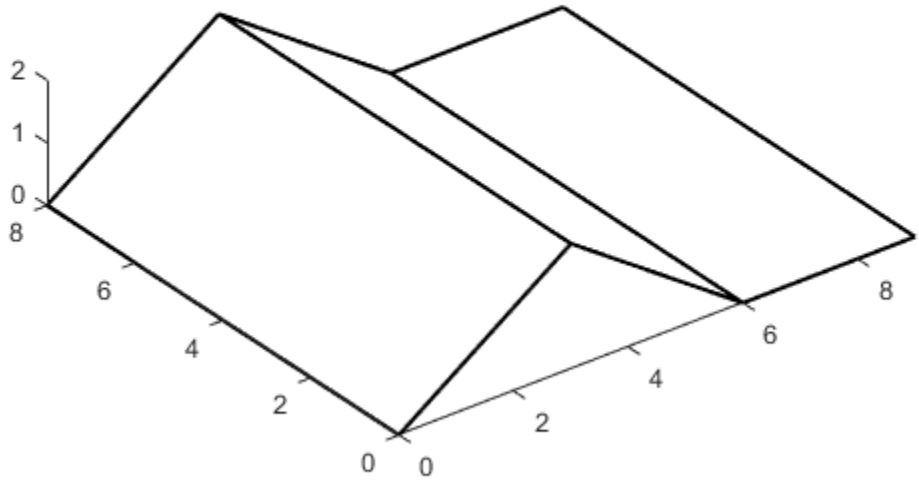
Add elevations to the triangulation to create a surface and plot the surface.

```
z = [0 0 0 0 0 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0]';
trisurf(T,x,y,z,'FaceColor','cyan')
axis equal
```



Create a new triangulation on the elevated surface, and compute and plot its feature edges using a filter angle of $\frac{\pi}{6}$.

```
TR = triangulation(T,x,y,z);
F = featureEdges(TR,pi/6)';
plot3(x(F),y(F),z(F),'k','LineWidth',1.5);
axis equal
```



Input Arguments

TR — Triangulation representation
scalar triangulation object

Triangulation representation for 2-D triangulations only, specified as a scalar `triangulation` or `delaunayTriangulation` object.

Data Types: `triangulation` | `delaunayTriangulation`

`theta` — Filter angle
scalar

Filter angle in radians, specified as a scalar in the range [0,π]. `featureEdges` returns adjacent triangles that have a dihedral angle that deviates from π by an angle greater than `theta`.

Data Types: `double`

More About

Feature Edge

A feature edge is a triangulation edge that has any of the following attributes:

- The edge belongs to only one triangle.
- The edge is shared by more than two triangles.
- The edge is shared by a pair of triangles with angular deviation greater than the angle `theta`.

`F` is a two-column matrix whose rows correspond to a feature edge. The first element of each row is the identification number of the starting edge vertex, and the second element is the identification number of the ending edge vertex. A vertex identification is the row number of the corresponding vertex in the `Points` property.

See Also

`delaunayTriangulation` | `edges` | `triangulation`

Introduced in R2013a

freeBoundary

Free boundary facets

Syntax

```
F = freeBoundary(TR)
[F,P] = freeBoundary(TR)
```

Description

`F = freeBoundary(TR)` returns the free boundary facets of the triangles or tetrahedra in TR. A facet in TR is on the free boundary if it is referenced by only one triangle or tetrahedron.

`[F,P] = freeBoundary(TR)` also returns a matrix containing the vertices of the free boundary facets.

Examples

2-D Delaunay Triangulation

You can use the `freeBoundary` function to highlight the outer edges of a 2-D Delaunay triangulation.

Create a triangulation from a 2-D set of points.

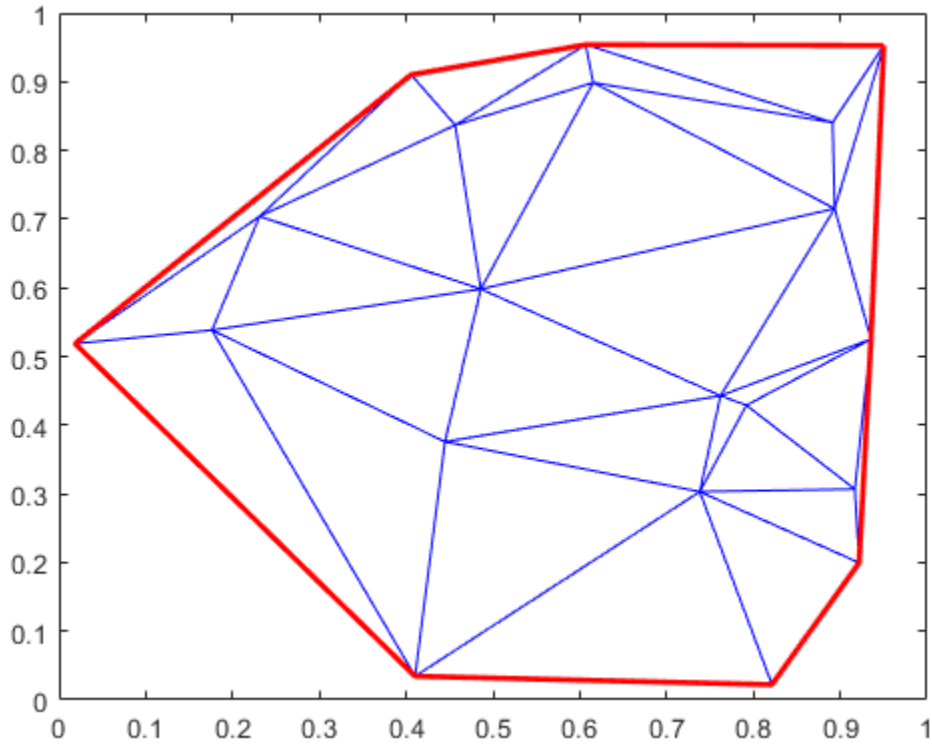
```
x = gallery('uniformdata',[20,1],0);
y = gallery('uniformdata',[20,1],1);
TR = delaunayTriangulation(x,y);
```

Find the free boundary edges.

```
F = freeBoundary(TR);
```

Plot the triangulation and highlight the free boundary edges in red.

```
triplot(TR)
hold on
plot(x(F),y(F),'-r','LineWidth',2)
```



Surface of 3-D Triangulation

You can use the `freeBoundary` function to extract the facets of a 3-D triangulation that cover the surface of an object.

Load a 3-D triangulation.

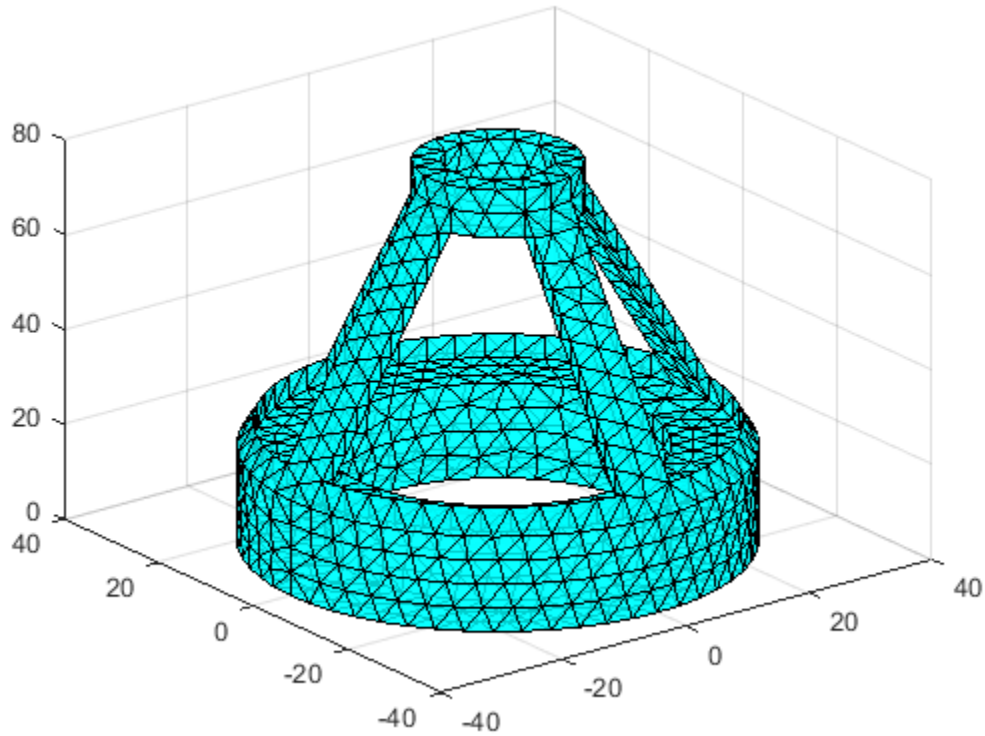
```
load tetmesh
TR = triangulation(tet,X);
```

Compute the boundary triangulation.

```
[F,P] = freeBoundary(TR);
```

Plot the boundary triangulation.

```
trisurf(F,P(:,1),P(:,2),P(:,3), ...
       'FaceColor','cyan','FaceAlpha',0.8);
```



Input Arguments

TR — Triangulation representation
scalar triangulation object

Triangulation representation, specified as a scalar `triangulation` or `delaunayTriangulation` object.

Data Types: `triangulation` | `delaunayTriangulation`

Output Arguments

F — Triangulation connectivity list
matrix

Triangulation connectivity list, returned as a matrix.

If you call `freeBoundary` with one output argument, then the elements of `F` are row numbers in the `Points` property of the input triangulation. If you call `freeBoundary` with two output arguments, then the elements of `F` are row numbers in `P`.

Data Types: `double`

P — Free boundary points
matrix

Free boundary points, returned as a matrix containing the coordinates of the vertices of the free boundary facets. Each row of `P` contains the coordinates of a vertex.

Data Types: `double`

See Also

`delaunayTriangulation` | `faceNormal` | `featureEdges` | `triangulation`

Introduced in R2013a

incenter

Incenter of triangulation elements

Syntax

```
C = incenter(TR)
C = incenter(TR,ID)
[C,r] = incenter( ___ )
```

Description

`C = incenter(TR)` returns the coordinates of the incenters of each triangle or tetrahedron in the triangulation `TR`.

`C = incenter(TR,ID)` returns the coordinates of the incenter of each triangle or tetrahedron specified by `ID`. The identification numbers of the triangles or tetrahedra in `TR` are the corresponding row numbers of the property `TR.ConnectivityList`.

`[C,r] = incenter( ___ )` also returns the radii of the inscribed circles or spheres.

Examples

Compute Incenters of 2-D Delaunay Triangulation

Create a 2-D Delaunay triangulation.

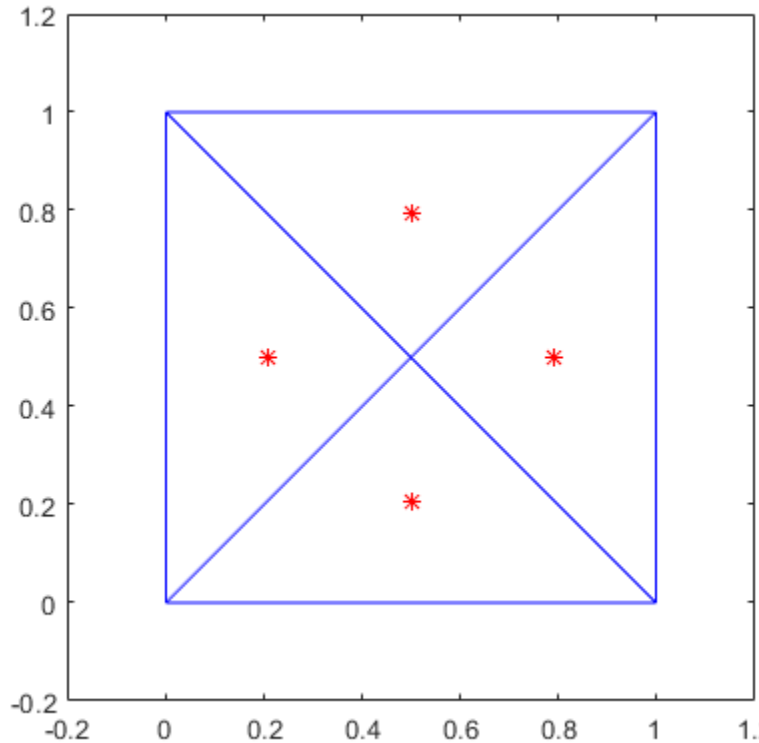
```
x = [0 1 1 0 0.5]';
y = [0 0 1 1 0.5]';
TR = delaunayTriangulation(x,y);
```

Compute the incenters of the triangles.

```
C = incenter(TR);
```

Plot the triangles and incenters.

```
triplot(TR)
axis equal
axis([-0.2 1.2 -0.2 1.2])
hold on
plot(C(:,1),C(:,2),'*r')
```



3-D Triangulation

Load a 3-D triangulation.

```
load tetmesh
```

Calculate the incenter coordinates of the first five tetrahedra in the triangulation, in addition to the radii of their inscribed spheres.

```
TR = triangulation(tet,X);
[C,r] = incenter(TR,[1:5]')
```

```
C = 5×3

   -6.1083  -31.0234    8.1439
   -2.1439  -31.0283    5.8742
   -1.9555  -31.9463    7.4112
   -4.3019  -30.8460   10.5169
   -3.1596  -29.3642    6.1851


r = 5×1

    0.7528
    0.9125
    0.8430
    0.6997
    0.7558
```

Input Arguments

TR — Triangulation representation
scalar triangulation object

Triangulation representation, specified as a scalar `triangulation` or `delaunayTriangulation` object.

Data Types: `triangulation` | `delaunayTriangulation`

`ID` — Triangle or tetrahedron IDs
scalar | column vector

Triangle or tetrahedron IDs, specified as a scalar or a column vector whose elements each correspond to a single triangle or tetrahedron in the triangulation object. The identification number of each triangle or tetrahedron is the corresponding row number of the `ConnectivityList` property.

Data Types: `double`

Output Arguments

`C` — Incenters
matrix

Incenters, returned as a matrix whose rows contain the coordinates of an incenter.

Data Types: `double`

`r` — Radii
vector

Radii of the inscribed circles or spheres, returned as a vector.

Data Types: `double`

See Also

`circumcenter` | `delaunayTriangulation` | `triangulation`

Introduced in R2013a

isConnected

Test if two vertices are connected by an edge

Syntax

```
TF = isConnected(TR,startID,endID)
TF = isConnected(TR,E)
```

Description

`TF = isConnected(TR,startID,endID)` returns a `logical` column vector whose elements are 1 (`true`) when the specified starting and ending pairs of vertices are connected by an edge. `startID` and `endID` specify edges by their starting and ending points, where each point is represented by its row number in `TR.Points`. The return value `ID` identifies triangles by their identification numbers.

`TF = isConnected(TR,E)` specifies the edge start and end vertex IDs in a two-column matrix `E`.

Examples

Determine Connecting Points in 2-D Triangulation

Load a 2-D triangulation.

```
load trimesh2d
TR = triangulation(tri,x,y);
```

Determine whether vertices 3 and 117 are connected by an edge.

```
TF = isConnected(TR,3,117)
```

```
TF = logical
   1
```

Determine whether vertices 3 and 164 are connected by an edge.

```
TF = isConnected(TR,3,164)
```

```
TF = logical
   0
```

Represent Vertices in a Matrix

Create a 3-D Delaunay triangulation.

```
X = gallery('uniformdata',[10,3],0);
DT = delaunayTriangulation(X);
```

Determine whether vertices 2 and 7 are connected by an edge, and whether vertices 4 and 9 are connected.

```
E = [2 7; 4 9];
TF = isConnected(DT,E)
```

```
TF = 2x1 logical array

   1
   0
```

Input Arguments

TR — Triangulation representation
scalar triangulation object

Triangulation representation, specified as a scalar `triangulation` or `delaunayTriangulation` object.

Data Types: `triangulation` | `delaunayTriangulation`

`startID` — Starting vertex identification
column vector

Starting vertex identification for each edge, specified as a column vector. A vertex identification number is the number of the row in the `Points` property that corresponds to the vertex.

Data Types: `double`

`endID` — Ending vertex identification
column vector

Ending vertex identification for each edge, specified as a column vector. A vertex identification number is the number of the row in the `Points` property that corresponds to the vertex.

Data Types: `double`

`E` — Edge matrix
two-column matrix

Edge matrix, specified as a two-column matrix. Each row contains the starting and ending vertex identifications for an edge. A vertex identification number is the number of the row in the `Points` property that corresponds to the vertex.

Data Types: `double`

See Also

`delaunayTriangulation` | `edgeAttachments` | `edges` | `triangulation`

Introduced in R2013a

nearestNeighbor

Closest vertex

Syntax

```
ID = nearestNeighbor(TR,P)
ID = nearestNeighbor(TR,x,y)
ID = nearestNeighbor(TR,x,y,z)
[ID,d] = nearestNeighbor( ___ )
```

Description

`ID = nearestNeighbor(TR,P)` returns the IDs of the vertices closest to the query points in `P`. Each row in the matrix `P` contains the coordinates of a query point.

`ID = nearestNeighbor(TR,x,y)` specifies the `x`-coordinates and `y`-coordinates of 2-D query points as separate column vectors.

`ID = nearestNeighbor(TR,x,y,z)` specifies the *x*-coordinates, *y*-coordinates, and *z*-coordinates of 3-D query points as separate column vectors.

`[ID,d] = nearestNeighbor( ___ )` also returns the Euclidean distance between each query point and its nearest neighbor for any of the previous syntaxes.

Examples

3-D Query

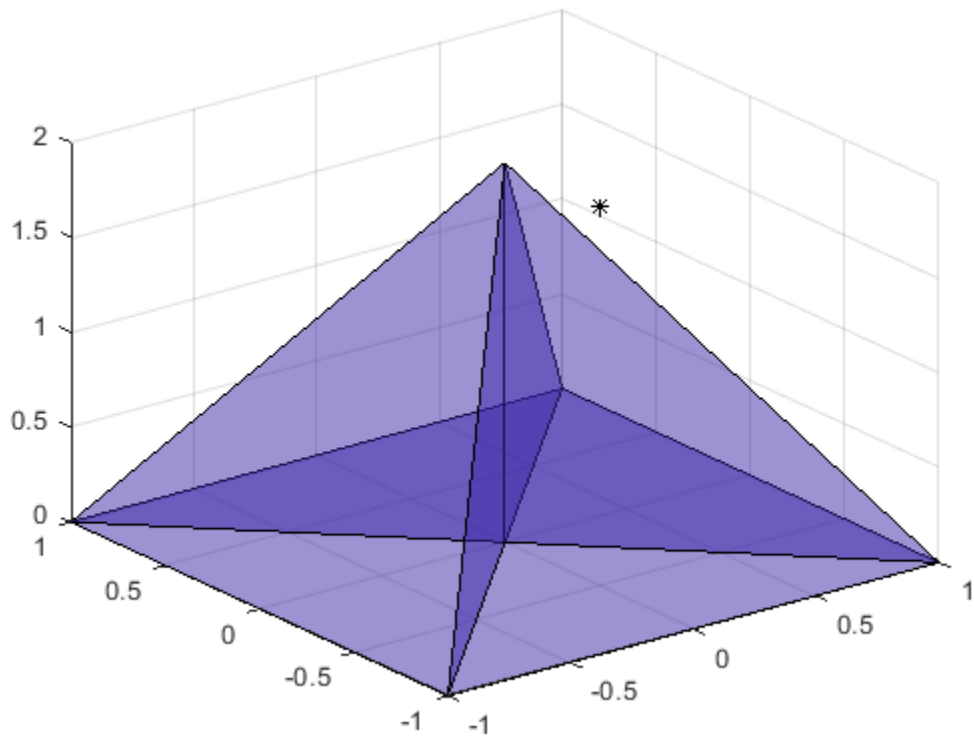
Compute the nearest neighbors in a 3-D triangulation.

Create a 3-D Delaunay triangulation.

```
P = [1 1 0; -1 1 0; -1 -1 0; 1 -1 0; 0 0 2; 0 0 0];
TR = delaunayTriangulation(P);
```

Plot the triangulation and a query point.

```
tri = TR(:,:);
trisurf(tri,P(:,1),P(:,2),P(:,3),'FaceAlpha',0.5)
hold on
x = 0;
y = -0.5;
z = 2;
plot3(x,y,z,'k*')
```



Find the coordinates of the nearest neighbor to the query point.

```
ID = nearestNeighbor(TR,x,y,z);
C = TR.Points(ID,:)
```

```
C = 1×3

     0     0     2
```

2-D Query

Compute the nearest neighbors in a 2-D triangulation.

Create a 2-D triangulation.

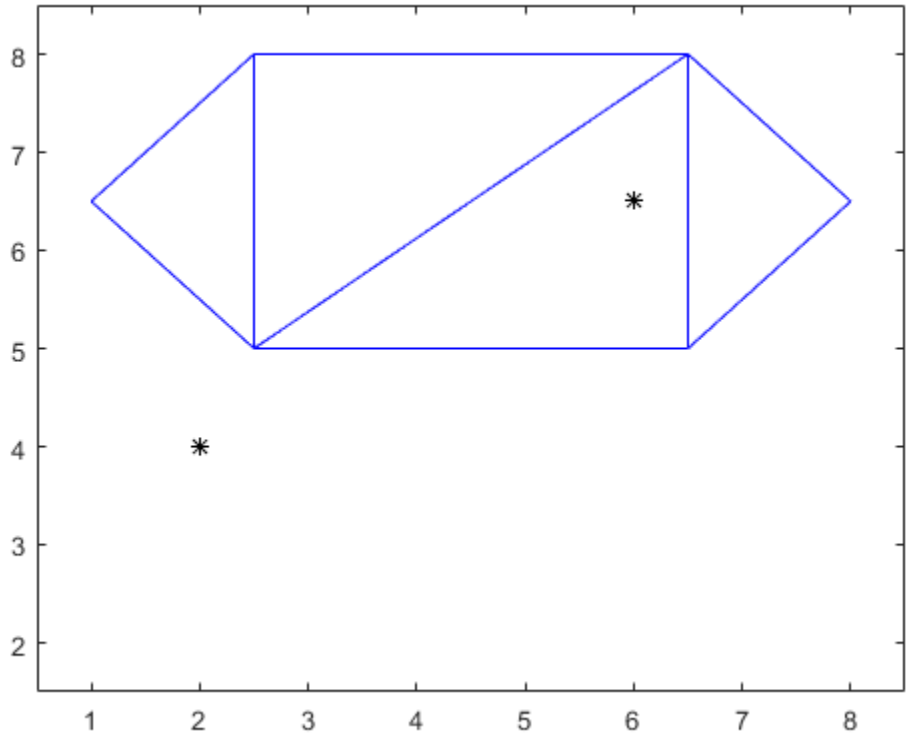
```
C = [5 3 1; 3 2 1; 3 4 2; 4 6 2];
TP = [2.5 8.0; 6.5 8.0; 2.5 5.0; 6.5 5.0; 1.0 6.5; 8.0 6.5];
TR = triangulation(C,TP);
```

Define two query points.

```
P = [2 4; 6 6.5];
```

Plot the triangulation and the query points.

```
triplot(TR)
hold on
plot(P(:,1),P(:,2),'k*')
ylim([1.5 8.5])
xlim([0.5 8.5])
```

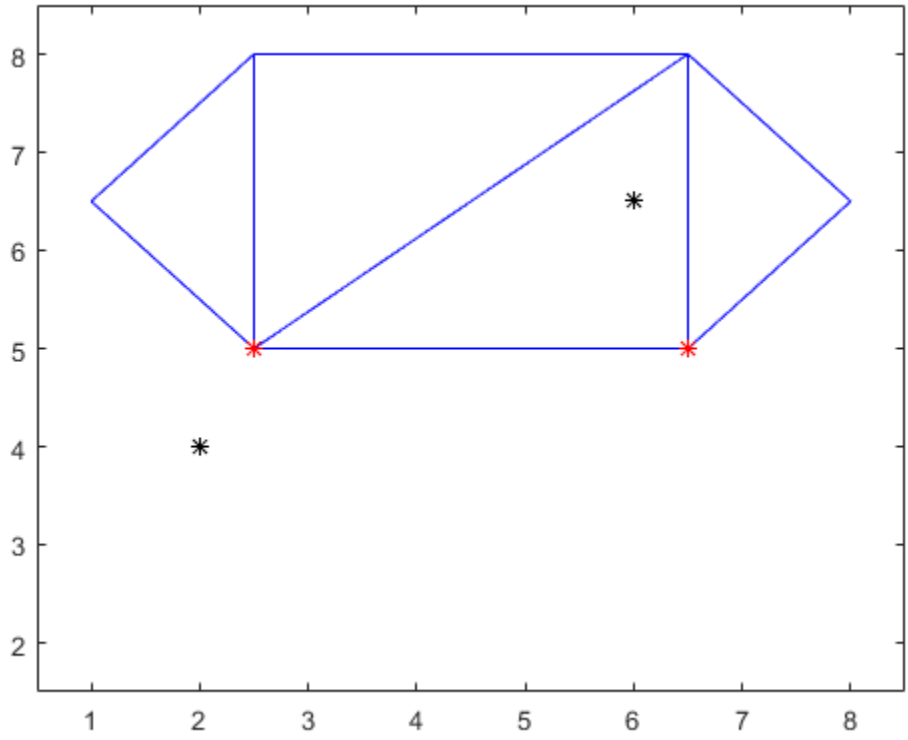


Find the nearest neighbors to the query points and the distances between them.

```
[ID,d] = nearestNeighbor(TR,P);
```

Highlight in red the points in the triangulation that are the nearest neighbors to the query points.

```
N = TP(ID,:);
plot(N(:,1),N(:,2),'*r')
```



Display the distance between each query point and its nearest neighbor.

```
d
```

```
d = 2×1

    1.1180
    1.5811
```

Input Arguments

TR — Triangulation representation
scalar triangulation object

Triangulation representation, specified as a scalar `triangulation` or `delaunayTriangulation` object. `nearestNeighbor` does not support `delaunayTriangulation` objects with constrained edges.

Data Types: `triangulation` | `delaunayTriangulation`

P — Query points
2-column matrix (2-D) | 3-column matrix (3-D)

Query points, specified as a matrix with 2 or 3 columns. `P` contains the *x*-coordinates, *y*-coordinates, and (possibly) *z*-coordinates of the query points.

Data Types: `double`

x — *x*-coordinates
column vector

x-coordinates of query points, specified as a column vector.

Data Types: `double`

y — *y*-coordinates
column vector

y-coordinates of query points, specified as a column vector.

Data Types: `double`

z — *z*-coordinates
column vector

z-coordinates of query points, specified as a column vector.

Data Types: `double`

Output Arguments

`ID` — Vertex IDs
column vector

Vertex IDs of the nearest neighbors to the query points, returned as a column vector. A vertex ID is the row number of the corresponding vertex in the `Points` property.

Data Types: `double`

d — Euclidean distances
column vector

Euclidean distances from the query points to their nearest neighbors, returned as a column vector the same length as `ID`.

Data Types: `double`

See Also

`delaunayTriangulation` | `triangulation`

Introduced in R2013a

neighbors

Triangle or tetrahedron neighbors

Syntax

```
N = neighbors(TR)
N = neighbors(TR,ID)
```

Description

`N = neighbors(TR)` returns the IDs of the neighbors to all triangles or tetrahedra in `TR`. `N` is a 3-column (2-D) or 4-column (3-D) matrix whose rows contain the IDs of the neighboring triangles or tetrahedra to that element in the triangulation.

`N = neighbors(TR,ID)` returns the neighbors of the triangles or tetrahedra specified in `ID`.

By convention, the (*i*,*j*) element of `N` is the neighbor opposite to the *j*th vertex of the *i*th element of `ID`.

Examples

3-D Triangulation

Find neighboring tetrahedra in a 3-D triangulation.

Create a 3-D triangulation.

```
load tetmesh
TR = triangulation(tet,X);
```

Compute the IDs of the neighboring tetrahedra to each element of the triangulation.

```
N = neighbors(TR);
```

Display the IDs of the neighbors to the fifth tetrahedron.

```
N(5,:)
```

```
ans = 1×4

        2360        1539           2        1851
```

Examine the vertex IDs of the first neighbor.

```
TR.ConnectivityList(N(1),:)
```

```
ans = 1×4

        1093         891         893         858
```

2-D Delaunay Triangulation

Find neighboring triangles in a 2-D Delaunay triangulation.

Create a 2-D Delaunay triangulation.

```
x = gallery('uniformdata',[10,1],0);
y = gallery('uniformdata',[10,1],1);
TR = delaunayTriangulation(x,y);
```

Find the neighbors of the first triangle. `NaN` indicates that the triangle is on the boundary of the triangulation and only has two neighbors.

```
N = neighbors(TR,1)
```

```
N = 1×3

   NaN     4     5
```

Examine the vertex IDs of the third neighbor.

```
TR.ConnectivityList(N(3),:)
```

```
ans = 1×3
```

```
4     3     7
```

Input Arguments

TR — Triangulation representation
scalar triangulation object

Triangulation representation, specified as a scalar `triangulation` or `delaunayTriangulation` object.

Data Types: `triangulation` | `delaunayTriangulation`

ID — Triangle or tetrahedron IDs
scalar | column vector

Triangle or tetrahedron IDs, specified as a scalar or a column vector whose elements correspond to a single triangle or tetrahedron in the triangulation object. The ID of each triangle or tetrahedron is the corresponding row number of the `ConnectivityList` property.

If a triangle or tetrahedron has one or more boundary facets, the nonexistent neighbors are represented as `NaN` values in `N`.

Data Types: `double`

See Also

`delaunayTriangulation` | `edgeAttachments` | `triangulation`

Introduced in R2013a

pointLocation

Triangle or tetrahedron enclosing point

Syntax

```
ID = pointLocation(TR,P)
ID = pointLocation(TR,x,y)
ID = pointLocation(TR,x,y,z)
[ID,B] = pointLocation( ___ )
```

Description

`ID = pointLocation(TR,P)` returns the IDs of the triangles or tetrahedra enclosing the query points in `P`. Each row in the matrix `P` contains the coordinates of a query point.

`ID = pointLocation(TR,x,y)` specifies the `x`-coordinates and `y`-coordinates of 2-D query points as separate column vectors.

`ID = pointLocation(TR,x,y,z)` specifies the *x*-coordinates, *y*-coordinates, and *z*-coordinates of 3-D query points as separate column vectors.

`[ID,B] = pointLocation( ___ )` also returns the barycentric coordinates of each query point with respect to its enclosing triangle or tetrahedron for any of the previous syntaxes.

Examples

Enclosing Triangles

Find the triangles of a triangulation that enclose a set of query points.

Define the points and connectivity of a triangulation.

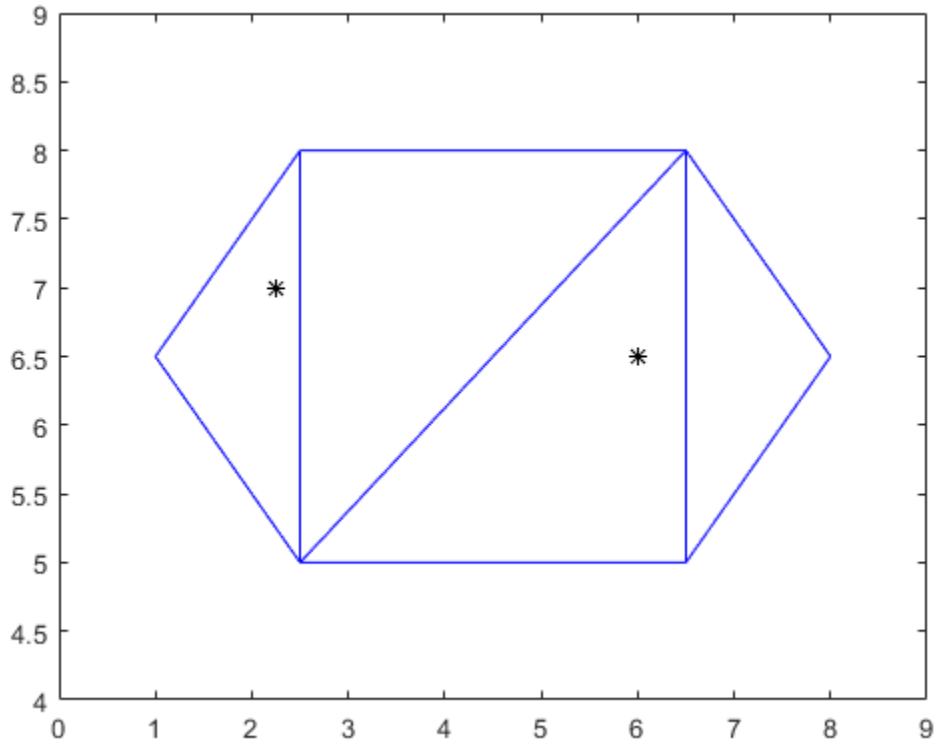
```
TP = [2.5 8.0; 6.5 8.0; 2.5 5.0; 6.5 5.0; 1.0 6.5; 8.0 6.5];
C = [5 3 1; 3 2 1; 3 4 2; 4 6 2];
TR = triangulation(C,TP);
```

Define two query points.

```
P = [2.25 7; 6 6.5];
```

Plot the triangulation and the query points.

```
triplot(TR)
hold on
plot(P(:,1),P(:,2),'k*')
ylim([4 9])
xlim([0 9])
```



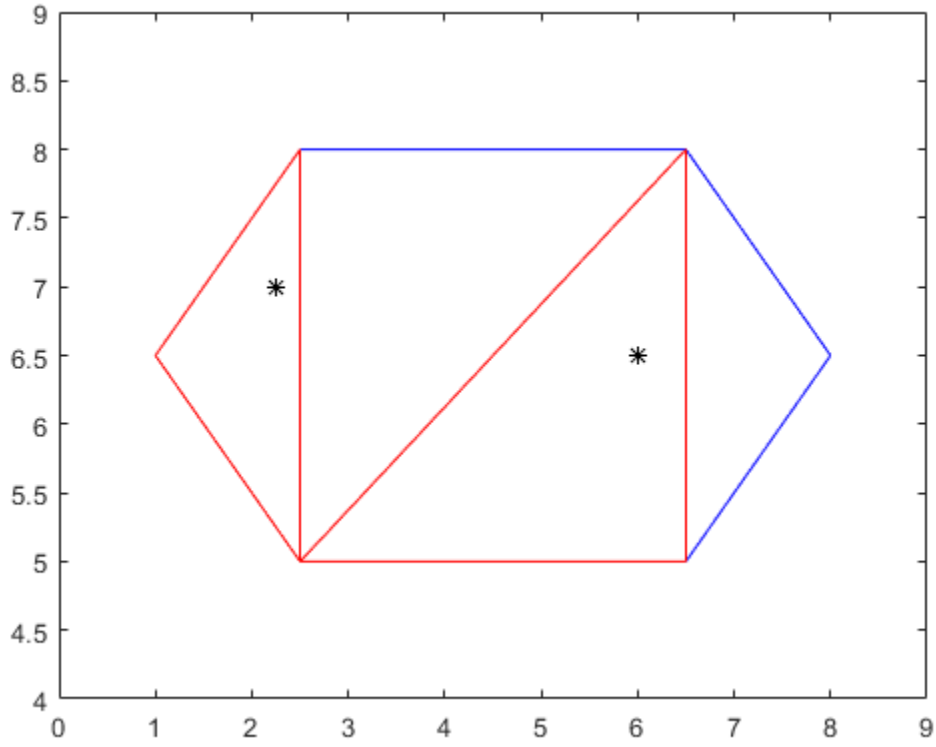
Determine the IDs of the triangles that enclose each query point.

```
ID = pointLocation(TR,P)
```

```
ID = 2×1

     1
     3
```

Highlight the triangles that enclose the query points in red.

```
triplot(TR.ConnectivityList(ID,:),TP(:,1),TP(:,2),'r')
```



Enclosing Tetrahedra

Find the tetrahedra of a 3-D triangulation that enclose a set of query points.

Create a Delaunay triangulation from a set of 3-D points.

```
x = gallery('uniformdata',[20 1],0);
y = gallery('uniformdata',[20 1],1);
z = gallery('uniformdata',[20 1],2);
TR = delaunayTriangulation(x,y,z);
```

Find the IDs of the tetrahedra that enclose the query points, and compute the barycentric coordinates of the query points.

```
P = [0.7 0.6 0.3; 0.5 0.5 0.5];
[ID,B] = pointLocation(TR,P)
```

```
ID = 2×1

    57
    56


B = 2×4

    0.2796    0.0184    0.5286    0.1734
    0.3687    0.0149    0.5343    0.0821
```

Input Arguments

TR — Triangulation representation
scalar triangulation object

Triangulation representation, specified as a scalar `triangulation` or `delaunayTriangulation` object.

Data Types: `triangulation` | `delaunayTriangulation`

P — Query points
2-column matrix (2-D) | 3-column matrix (3-D)

Query points, specified as a 2-column matrix (2-D) or a 3-column matrix (3-D). `P` contains the *x*-coordinates, *y*-coordinates, and (possibly) *z*-coordinates of the query points.

Data Types: `double`

x — *x*-coordinates
column vector

x-coordinates of query points, specified as a column vector.

Data Types: `double`

y — *y*-coordinates
column vector

y-coordinates of query points, specified as a column vector.

Data Types: `double`

z — *z*-coordinates
column vector

z-coordinates of query points, specified as a column vector.

Data Types: `double`

Output Arguments

`ID` — Triangle or tetrahedra IDs
column vector

Triangle or tetrahedra IDs of the triangles or tetrahedra enclosing the query points, returned as a column vector. A triangle or tetrahedron ID is the row number of the corresponding triangle or tetrahedron in the `ConnectivityList` property.

If a query point lies on the boundary of two or more triangles or tetrahedra, then the largest ID is returned.

`ID` contains `NaN` values for points that are not located in a triangle or tetrahedron of the triangulation.

Data Types: `double`

B — Barycentric coordinates
3-column matrix (2-D) | 4-column matrix (3-D)

Barycentric coordinates of each query point with respect to its enclosing triangle or tetrahedron, returned as a 3-column matrix (2-D) or a 4-column matrix (3-D).

Data Types: `double`

See Also

`delaunayTriangulation` | `triangulation`

Introduced in R2013a

size

Size of triangulation connectivity list

Syntax

```
sz = size(TR)
```

Description

`sz = size(TR)` returns the size of the triangulation connectivity list in the property `TR.ConnectivityList`. The output `sz` is a two-element row vector. The first element is the number of triangles or tetrahedra in the triangulation, and the second element is the number of vertices per triangle (three) or tetrahedron (four).

Examples

Size of 2-D Triangulation

Create a 2-D triangulation.

```
P = [ 2.5    8.0
      6.5    8.0
      2.5    5.0
      6.5    5.0
      1.0    6.5
      8.0    6.5];

C = [5  3  1;
     3  2  1;
     3  4  2;
     4  6  2];

TR = triangulation(C,P)
```

```
TR 
  triangulation with properties:
```

```
              Points: [6x2 double]
    ConnectivityList: [4x3 double]
```

Get the size of the connectivity list. The triangulation has 4 triangles, and each triangle has 3 vertices.

```
size(TR)
```

```
ans = 1×2

     4     3
```

Input Arguments

TR — Triangulation representation
scalar triangulation object

Triangulation representation, specified as a scalar `triangulation` or `delaunayTriangulation` object.

Data Types: `triangulation` | `delaunayTriangulation`

See Also

`delaunayTriangulation` | `triangulation`

Introduced in R2013a

vertexAttachments

Triangles or tetrahedra attached to vertex

Syntax

```
V = vertexAttachments(TR)
V = vertexAttachments(TR,ID)
```

Description

`V = vertexAttachments(TR)` returns the IDs of the triangles or tetrahedra attached to every vertex in the triangulation `TR`. A triangle or tetrahedron ID is the row number of the corresponding triangle or tetrahedron in the property `TR.ConnectivityList`.

The output `v` is an *m*-by-1 cell array where *m* is the number of vertices in `TR`. Each element of `V` contains the IDs of the attached triangles or tetrahedra corresponding to that vertex. The vertices can have a different number of attached triangles or tetrahedra depending on where they are located in the triangulation.

`V = vertexAttachments(TR,ID)` returns the IDs of the triangles or tetrahedra attached to the vertices specified in `ID`. The vertex IDs in `ID` are the row numbers of the corresponding vertices in the property `TR.Points`.

Examples

2-D Delaunay Triangulation

Locate and plot the attachments to a specified vertex in a 2-D Delaunay triangulation.

Create a Delaunay triangulation from a set of 2-D points.

```
x = gallery('uniformdata',[20,1],0);
y = gallery('uniformdata',[20,1],1);
TR = delaunayTriangulation(x,y);
```

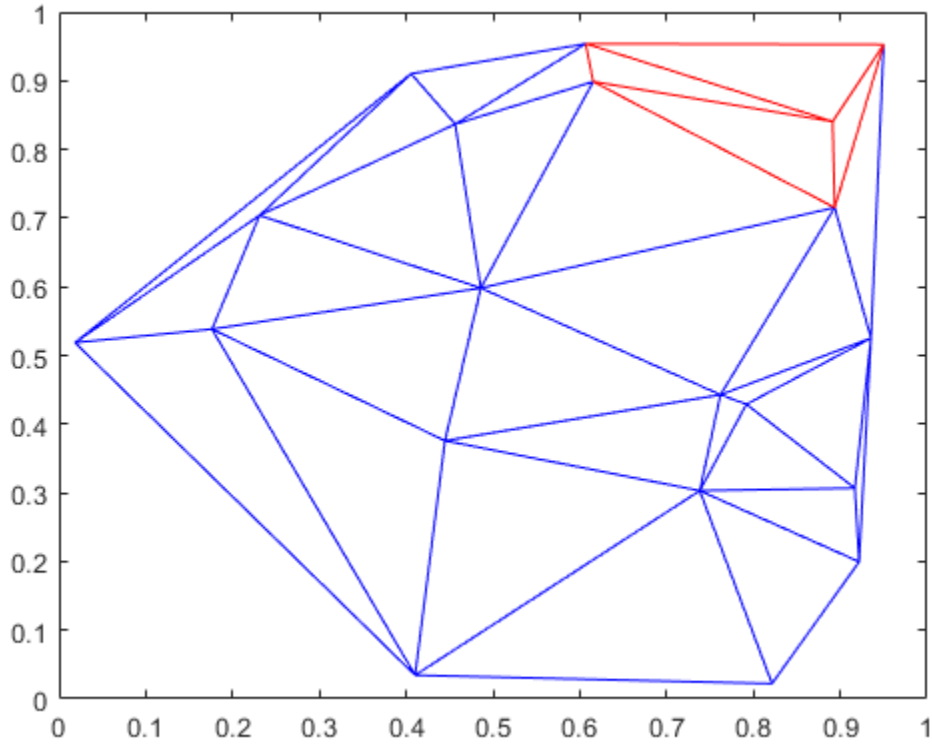
Find the triangles attached to the 5th vertex.

```
V = vertexAttachments(TR,5);
V{:}
```

```
ans = 1×4

    18    23    21    22
```

Plot the triangulation, and plot the triangles attached to vertex 5 in red.

```
triplot(TR)
hold on
triplot(TR(V{:},:),x,y,'Color','r')
```



Input Arguments

TR — Triangulation representation
scalar triangulation object

Triangulation representation, specified as a scalar `triangulation` or `delaunayTriangulation` object.

Data Types: `triangulation` | `delaunayTriangulation`

ID — Vertex IDs
column vector

Vertex IDs of the query points, returned as a column vector. A vertex ID is the row number of the corresponding vertex in the `Points` property.

Data Types: `double`

See Also

`delaunayTriangulation` | `edgeAttachments` | `triangulation`

Introduced in R2013a

vertexNormal

Triangulation vertex normal

Syntax

```
V = vertexNormal(TR)
V = vertexNormal(TR,ID)
```

Description

`V = vertexNormal(TR)` returns the unit normal vectors to all vertices in a 3-D surface triangulation. `V` is a three-column matrix with each row containing the unit normal coordinates corresponding to the vertices in `TR.Points`.

`V = vertexNormal(TR,ID)` returns the unit normal vector to each vertex indexed by `ID`. A vertex ID is the row number corresponding to the vertices in the property `TR.Points`.

Examples

Surface of a Cube

Compute and plot the unit normal vectors to the vertices of a triangulation.

Create a 3-D triangulation representing the volume of a cube.

```
[X,Y,Z] = meshgrid(1:4);
x = X(:);
y = Y(:);
z = Z(:);
DT = delaunayTriangulation(x,y,z);
```

Triangulate the boundary of the cube.

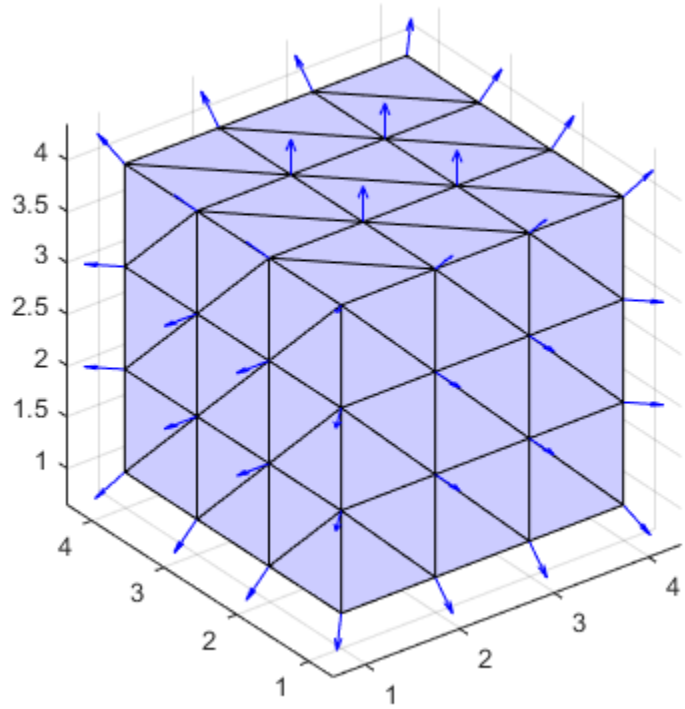
```
[Tfb,Xfb] = freeBoundary(DT);
TR = triangulation(Tfb,Xfb);
```

Find the unit normal vectors to the triangle vertices.

```
V = vertexNormal(TR);
```

Plot the triangulated surface and the unit normal vectors.

```
trisurf(TR,'FaceColor',[0.8 0.8 1.0]);
axis equal
hold on
quiver3(Xfb(:,1),Xfb(:,2),Xfb(:,3), ...
     V(:,1),V(:,2),V(:,3),0.5,'Color','b');
```



Input Arguments

TR — Triangulation representation
scalar triangulation object

Triangulation representation for 3-D surface triangulations only, specified as a scalar `triangulation` or `delaunayTriangulation` object.

Data Types: `triangulation` | `delaunayTriangulation`

ID — Vertex IDs
scalar | column vector

Vertex IDs, specified as a scalar or a column vector whose elements correspond to a single vertex in the triangulation object. The ID of each vertex is the corresponding row number of the vertices in the `Points` property.

Data Types: `double`

See Also

`delaunayTriangulation` | `faceNormal` | `triangulation`

Introduced in R2013a

readcell

Read cell array from file

Syntax

```
C = readcell(filename)
C = readcell(filename,opts)
C = readcell( ___ ,Name,Value)
```

Description

`C = readcell(filename)` creates a cell array by reading column-oriented data from a file.

`readcell` determines the file format from the file extension:

- `.txt`, `.dat`, or `.csv` for delimited text files
- `.xls`, `.xlsb`, `.xlsm`, `.xlsx`, `.xltm`, `.xltx`, or `.ods` for spreadsheet files

`C = readcell(filename,opts)` additionally uses the import options `opts`.

`C = readcell( ___ ,Name,Value)` creates cell array from a file with additional options specified by one or more name-value pair arguments. Use any of the input arguments from the previous syntaxes before specifying the name-value pairs.

To set specific import options for your data, you can either use the `opts` object or you can specify name-value pairs. When you specify name-value pairs in addition to `opts`, then `readcell` supports only these name-value pairs:

- Text Files — `DateLocale`, `Encoding`
- Spreadsheet Files — `Sheet`, `UseExcel`

Examples

Read Tabular Data into Cell Array from Text File

Display the contents of `basic_cell.txt`, and then import the mixed data into a cell array.

```
type basic_cell.txt
```

```
1,2,3
hello,world,NaN
10-Oct-2018 10:27:56,1,
```

```
C = readcell('basic_cell.txt')
```

```
C=3×3 cell
    {[                    1]}    {[    2]}    {[        3]}
    {'hello'                }    {'world'}    {[      NaN]}
    {[10-Oct-2018 10:27:56]}    {[    1]}    {1x1 missing}
```

Read Tabular Data into Cell Array from Spreadsheet

Import mixed tabular data from `basic_cell.xls` into a cell array.

```
C = readcell('basic_cell.xls')
```

```
C=3×3 cell
    {[                    1]}    {[    2]}    {[        3]}
    {'hello'                }    {'world'}    {1x1 missing}
    {[10-Oct-2018 10:27:56]}    {[    1]}    {1x1 missing}
```

Read Cell Array from Specified Sheet and Range

Preview the data from a spreadsheet file and import the mixed data into a cell array from a specified sheet and range.

The spreadsheet file `airlinesmall_subset.xlsx` contains data in multiple worksheets for years between 1996 and 2008. Each worksheet has data for a given year. Preview the data from file `airlinesmall_subset.xlsx`. The `preview` function shows data from the from the first worksheet by default.

```
opts = detectImportOptions('airlinesmall_subset.xlsx');
preview('airlinesmall_subset.xlsx',opts)
```

```
ans=8×29 table
    Year    Month    DayofMonth    DayOfWeek    DepTime    CRSDepTime    ArrTime    CRS
    ____    _____    __________    _________    _______    __________    _______    ___

    1996      1          18            4         2117         2120        2305
    1996      1          12            5         1252         1245        1511
    1996      1          16            2         1441         1445        1708
    1996      1           1            1         2258         2300        2336
    1996      1           4            4         1814         1814        1901
    1996      1          31            3         1822         1820        1934
    1996      1          18            4          729          730         841
    1996      1          26            5         1704         1705        1829
```

Import ten rows of data for variables 7, 8, and 9 from the worksheet named `'2007'`. The Excel range `'G2:I11'` represents columns 7 through 9, and ten rows starting at row 2 after the variable names.

```
M = readcell('airlinesmall_subset.xlsx','Sheet','2007','Range','G2:I11')
```

```
M=10×3 cell
    {[ 935]}    {[ 935]}    {'WN'}
    {[1041]}    {[1040]}    {'WN'}
    {[1430]}    {[1500]}    {'WN'}
    {[ 940]}    {[ 950]}    {'WN'}
    {[1515]}    {[1515]}    {'WN'}
    {[2042]}    {[2035]}    {'WN'}
    {[2116]}    {[2130]}    {'WN'}
    {[1604]}    {[1605]}    {'WN'}
    {[1258]}    {[1230]}    {'WN'}
    {[1134]}    {[1145]}    {'WN'}
```

Input Arguments

`filename` — Name of file to read

character vector | string scalar

Name of the file to read, specified as a character vector or a string scalar.

Depending on the location of your file, `filename` can take on one of these forms.

Location	Form
Current folder or folder on the MATLAB path	Specify the name of the file in `filename`. **Example:** `'myFile.txt'`
File in a folder	If the file is not in the current folder or in a folder on the MATLAB path, then specify the full or relative path name in `filename`. **Example:** `'C:\myFolder\myFile.xlsx'` **Example:** `'dataDir\myFile.txt'`
Remote Location	If the file is stored at a remote location, then `filename` must contain the full path of the file specified as a uniform resource locator (URL) of the form: *`scheme_name`*`://path_to_file/my_file.ext` Based on your remote location, *`scheme_name`* can be one of the values in this table. **Remote Location** / **`scheme_name`** Amazon S3 / `s3` Windows Azure Blob Storage / `wasb`, `wasbs` HDFS / `hdfs` For more information, see “Work with Remote Data”. **Example:** `'s3://bucketname/path_to_file/my_file.csv'`

- If `filename` includes the file extension, then the importing function determines the file format from the extension. Otherwise, you must specify the `'FileType'` name-value pair arguments to indicate the type of file.
- On Windows systems with Microsoft Excel software, the importing function reads any Excel spreadsheet file format recognized by your version of Excel.

- If your system does not have Excel for Windows or if you are using MATLAB Online, the importing function operates with the `UseExcel` property set to `false`, and reads only `.xls, .xlsx, .xlsm, .xltx, and .xltm` files.
- For delimited text files, the importing function converts empty fields in the file to either `NaN` (for a numeric variable) or an empty character vector (for a text variable). All lines in the text file must have the same number of delimiters. The importing function ignores insignificant white space in the file.

Data Types: `char` | `string`

`opts` — File import options
`SpreadsheetImportOptions` | `DelimitedtextImportOptions` | `FixedWidthImportOptions`

File import options, specified as an `SpreadsheetImportOptions`, `DelimitedTextImportOptions`, or `FixedWidthImportOptions` object created by the `detectImportOptions` function. The `opts` object contains properties that control the data import process. For more information on the properties of each object, see the appropriate object page.

Type of Files	Output
Spreadsheet files	`SpreadsheetImportOptions` object
Text files	`DelimitedTextImportOptions` object
Fixed-width text files	`FixedWidthImportOptions` object

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'NumHeaderLines',5` indicates that the first five lines that precede the tabular data are header lines.

Text and Spreadsheet Files

`FileType` — Type of file
`'text'` | `'spreadsheet'`

Type of file, specified as the comma-separated pair consisting of `'FileType'` and `'text'` or `'spreadsheet'`.

Specify the `'FileType'` name-value pair argument when the `filename` does not include the file extension or if the extension is other than one of the following:

- `.txt`, `.dat`, or `.csv` for delimited text files
- `.xls`, `.xlsb`, `.xlsm`, `.xlsx`, `.xltm`, `.xltx`, or `.ods` for spreadsheet files

Example: `'FileType','text'`

Data Types: `char` | `string`

`NumHeaderLines` — Number of header lines
positive integer

Number of header lines in the file, specified as the comma-separated pair consisting of `'NumHeaderLines'` and a positive integer. If unspecified, the importing function automatically detects the number of header lines in the file.

Example: `'NumHeaderLines',7`

Data Types: `single` | `double`

`ExpectedNumVariables` — Expected number of variables
positive integer

Expected number of variables, specified as the comma-separated pair consisting of `'ExpectedNumVariables'` and a positive integer. If unspecified, the importing function automatically detects the number of variables.

Data Types: `single` | `double`

`Range` — Portion of data to read
character vector | string scalar | numeric vector

Portion of the data to read from text or spreadsheet files, specified as the comma separated pair consisting of `'Range'` and a character vector, string scalar, or numeric vector in one of these forms.

Ways to specify Range	Description
Starting Cell `'Cell'` or `[row col]`	Specify the starting cell for the data as a character vector or string scalar or a two element numeric vector. • Character vector or string scalar containing a column letter and row number using Excel `A1` notation. For example, `A5` is the identifier for the cell at the intersection of column `A` and row `5`. • Two element numeric vector of the form `[row col]` indicating the starting row and column. Using the starting cell, the importing function automatically detects the extent of the data by beginning the import at the start cell and ending at the last empty row or footer range. **Example:** `'A5'` or `[5 1]`
Rectangular Range `'Corner1:Corner2'` or `[r1 c1 r2 c2]`	Specify the exact range to read using the rectangular range in one of these forms. • `'Corner1:Corner2'` — Specify the range using `Corner1` and `Corner2` which are the two opposing corners that define the region to read in Excel `A1` notation. For example, `'C2:N15'`. • `[r1 c1 r2 c2]` — Specify the range using a four element numeric vector containing start-row, start-column, end-row, and end-column. For example, `[2 3 15 13]`. The importing function only reads the data contained in the specified range. Any empty fields within the specified range are imported as missing cells.

Ways to specify Range	Description
Row Range or Column Range `'Row1:Row2'` or `'Column1:Column2'`	Specify the range by identifying the beginning and ending rows using Excel row numbers. Using the specified row range, the importing function automatically detects the column extent by reading from the first nonempty column to the end of the data, and creates one variable per column. **Example:** `'5:500'` Alternatively, specify the range by identifying the beginning and ending columns using Excel column letters or numbers. Using the specified column range, the import function automatically detects the row extent by reading from the first nonempty row to the end of the data or the footer range. The number of columns in the specified range must match the number specified in the `ExpectedNumVariables` property. **Example:** `'A:K'`
Starting Row Number `n`	Specify the first row containing the data using the positive scalar row index. Using the specified row index, the importing function automatically detects the extent of the data by reading from the specified first row to the end of the data or the footer range. **Example:**5

Ways to specify Range	Description
Excel's Named Range `'NamedRange'`	In Excel, you can create names to identify ranges in the spreadsheet. For instance, you can select a rectangular portion of the spreadsheet and call it `'myTable'`. If such named ranges exist in a spreadsheet, then the importing function can read that range using its name. **Example:** `'Range','myTable'`
Unspecified or Empty `''`	If unspecified, the importing function automatically detects the used range. **Example:** `'Range',''` **Note:** *Used Range* refers to the rectangular portion of the spreadsheet that actually contains data. The importing function automatically detects the used range by trimming any leading and trailing rows and columns that do not contain data. Text that is only white space is considered data and is captured within the used range.

Data Types: `char` | `string` | `double`

`TextType` — Type for imported text data
`'char'` (default) | `'string'`

Type for imported text data, specified as the comma-separated pair consisting of `'TextType'` and either `'char'` or `'string'`.

- `'char'` — Import text data into MATLAB as character vectors.
- `'string'` — Import text data into MATLAB as string arrays.

Example: `'TextType','char'`

`DatetimeType` — Type for imported date and time data
`'datetime'` (default) | `'text'` | `'exceldatenum'` (spreadsheet files only)

Type for imported date and time data, specified as the comma-separated pair consisting of `'DatetimeType'` and one of these values: `'datetime'`, `'text'`, or `'exceldatenum'`. The value `'exceldatenum'` is applicable only for spreadsheet files, and is not valid for text files.

Value	Type for Imported Date and Time Data
`'datetime'`	MATLAB `datetime` data type For more information, see `datetime`.
`'text'`	If `'DatetimeType'` is specified as `'text'`, then the type for imported date and time data depends on the value specified in the `'TextType'` parameter: • If `'TextType'` is set to `'char'`, then the importing function returns dates as a cell array of character vectors. • If `'TextType'` is set to `'string'`, then the importing function returns dates as an array of strings.
`'exceldatenum'`	Excel serial date numbers A serial date number is a single number equal to the number of days from a given reference date. Excel serial date numbers use a different reference date than MATLAB serial date numbers. For more information on Excel dates, see `https://support.microsoft.com/en-us/kb/214330`.

Data Types: `char` | `string`

Text Files Only

`Delimiter` — Field delimiter characters

character vector | string scalar | cell array of character vectors | string array

Field delimiter characters in a delimited text file, specified as a character vector, string scalar, cell array of character vectors, or string array.

Example: `'Delimiter','|'`

Example: `'Delimiter',{';','*'}`

Data Types: `char` | `string` | `cell`

`Whitespace` — Characters to treat as white space

character vector | string scalar

Characters to treat as white space, specified as a character vector or string scalar containing one or more characters.

Example: `'Whitespace',' _'`

Example: `'Whitespace','?!.,'`

`LineEnding` — End-of-line characters
`{'\n','\r','\r\n'}` (default) | character vector | string scalar | cell array of character vectors | string array

End-of-line characters, specified as a character vector, string scalar, cell array of character vectors, or string array.

Example: `'LineEnding','\n'`

Example: `'LineEnding','\r\n'`

Example: `'LineEnding',{'\b',':'}`

Data Types: `char` | `string` | `cell`

`CommentStyle` — Style of comments
character vector | string scalar | cell array of character vectors | string array

Style of comments, specified as a character vector, string scalar, cell array of character vectors, or string array.

For example, to ignore the text following a percent sign on the same line, specify `CommentStyle` as `'%'`.

Example: `'CommentStyle',{'/*'}`

Data Types: `char` | `string` | `cell`

`Encoding` — Character encoding scheme
`'system'` | `'UTF-8'` | `'ISO-8859-1'` | `'windows-1251'` | `'windows-1252'` | ...

Character encoding scheme associated with the file, specified as the comma-separated pair consisting of `'Encoding'` and `'system'` or a standard character encoding scheme name, such as one of the values in this table.

`'Big5'`	`'ISO-8859-1'`	`'windows-847'`
`'Big5-HKSCS'`	`'ISO-8859-2'`	`'windows-949'`

'CP949'	'ISO-8859-3'	'windows-1250'
'EUC-KR'	'ISO-8859-4'	'windows-1251'
'EUC-JP'	'ISO-8859-5'	'windows-1252'
'EUC-TW'	'ISO-8859-6'	'windows-1253'
'GB18030'	'ISO-8859-7'	'windows-1254'
'GB2312'	'ISO-8859-8'	'windows-1255'
'GBK'	'ISO-8859-9'	'windows-1256'
'IBM866'	'ISO-8859-11'	'windows-1257'
'KOI8-R'	'ISO-8859-13'	'windows-1258'
'KOI8-U'	'ISO-8859-15'	'US-ASCII'
	'Macintosh'	'UTF-8'
	'Shift_JIS'	

Example: `'Encoding','system'` uses the system default encoding.

Data Types: `char` | `string`

`DurationType` — Output data type of duration data
`'duration'` (default) | `'text'`

Output data type of duration data from text files, specified as the comma-separated pair consisting of `'DurationType'` and either `'duration'` or `'text'`.

Value	Type for Imported Duration Data
`'duration'`	MATLAB `duration` data type For more information, see `duration`.

Value	Type for Imported Duration Data
`'text'`	If `'DurationType'` is specified as `'text'`, then the type for imported duration data depends on the value specified in the `'TextType'` parameter: • If `'TextType'` is set to `'char'`, then the importing function returns duration data as a cell array of character vectors. • If `'TextType'` is set to `'string'`, then the importing function returns duration data as an array of strings.

Data Types: `char` | `string`

`DateLocale` — Locale for reading dates
character vector | string scalar

Locale for reading dates, specified as the comma-separated pair consisting of `'DateLocale'` and a character vector or a string scalar of the form *xx_YY*, where:

- *YY* is an uppercase ISO 3166-1 alpha-2 code indicating a country.
- *xx* is a lowercase ISO 639-1 two-letter code indicating a language.

For a list of common values for the locale, see the `Locale` name-value pair argument for the `datetime` function.

When using the `%D` format specifier to read text as `datetime` values, use `DateLocale` to specify the locale in which the importing function should interpret month and day-of-week names and abbreviations.

If you specify the `DateLocale` argument in addition to `opts` the import options, then the importing function uses the specified value for the `DateLocale` argument, overriding the locale defined in the import options.

Example: `'DateLocale','ja_JP'`

`ConsecutiveDelimitersRule` — Procedure to handle consecutive delimiters
`'split'` | `'join'` | `'error'`

Procedure to handle consecutive delimiters in a delimited text file, specified as one of the values in this table.

Consecutive Delimiters Rule	Behavior
`'split'`	Split the consecutive delimiters into multiple fields.
`'join'`	Join the delimiters into one delimiter.
`'error'`	Return an error and abort the import operation.

Data Types: `char` | `string`

`LeadingDelimitersRule` — Procedure to manage leading delimiters
`'keep'` | `'ignore'` | `'error'`

Procedure to manage leading delimiters in a delimited text file, specified as one of the values in this table.

Leading Delimiters Rule	Behavior
`'keep'`	Keep the delimiter.
`'ignore'`	Ignore the delimiter.
`'error'`	Return an error and abort the import operation.

Spreadsheet Files Only

`Sheet` — Sheet to read from
`''` empty character array (default) | character vector | string scalar | positive scalar integer

Sheet to read from, specified as an empty character array, a character vector or string scalar containing the sheet name, or a positive scalar integer denoting the sheet index. Based on the value specified for the `Sheet` property, the import function behaves as described in the table.

Specification	Behavior
`''` (default)	Import data from the first sheet.
Name	Import data from the matching sheet name, regardless of order of sheets in the spreadsheet file.
Integer	Import data from sheet in the position denoted by the integer, regardless of the sheet names in the spreadsheet file.

Data Types: `char` | `string` | `single` | `double`

`UseExcel` — Flag to start instance of Microsoft Excel for Windows
`false` (default) | `true`

Flag to start an instance of Microsoft Excel for Windows when reading spreadsheet data, specified as the comma-separated pair consisting of `'UseExcel'` and either `true`, or `false`.

You can set the `'UseExcel'` parameter to one of these values:

- `true` — The importing function starts an instance of Microsoft Excel when reading the file.
- `false` — The importing function does not start an instance of Microsoft Excel when reading the file. When operating in this mode, the importing function functionality differs in the support of file formats and interactive features, such as formulas and macros.

UseExcel	**true**	**false**
Supported file formats	`.xls, .xlsx, .xlsm, .xltx, .xltm, .xlsb, .ods`	`.xls, .xlsx, .xlsm, .xltx, .xltm`
Support for interactive features, such as formulas and macros	Yes	No

When reading from spreadsheet files on Windows platforms, if you want to start an instance of Microsoft Excel, then set the `'UseExcel'` parameter to `true`.

See Also

`readmatrix` | `readtable` | `readtimetable` | `readvars` | `writecell`

Introduced in R2019a

readmatrix

Read matrix from file

Syntax

```
A = readmatrix(filename)
A = readmatrix(filename,opts)
A = readmatrix( ___ ,Name,Value)
```

Description

`A = readmatrix(filename)` creates an array by reading column-oriented data from a file. The `readmatrix` function performs automatic detection of import parameters for your file.

`readmatrix` determines the file format from the file extension:

- `.txt`, `.dat`, or `.csv` for delimited text files
- `.xls`, `.xlsb`, `.xlsm`, `.xlsx`, `.xltm`, `.xltx`, or `.ods` for spreadsheet files

For files containing mixed numeric and text data, `readmatrix` imports the data as a numeric array by default.

`A = readmatrix(filename,opts)` additionally uses the import options `opts`.

`A = readmatrix( ___ ,Name,Value)` creates an array from a file with additional options specified by one or more name-value pair arguments. Use any of the input arguments from the previous syntaxes before specifying the name-value pairs.

To set specific import options for your data, you can either use the `opts` object or you can specify name-value pairs. When you specify name-value pairs in addition to `opts`, then `readmatrix` supports only these name-value pairs:

- Text Files — `DateLocale`, `Encoding`
- Spreadsheet Files — `Sheet`, `UseExcel`

Examples

Read Matrix from Text File

Display the contents of `basic_matrix.txt` and then import the data into a matrix.

```
type basic_matrix.txt
```

```
6,8,3,1
5,4,7,3
1,6,7,10
4,2,8,2
2,7,5,9
```

```
M = readmatrix('basic_matrix.txt')
```

```
M = 5×4

     6     8     3     1
     5     4     7     3
     1     6     7    10
     4     2     8     2
     2     7     5     9
```

Read Matrix from Spreadsheet File

Import numeric data from `basic_matrix.xls` into a matrix.

```
M = readmatrix('basic_matrix.xls')
```

```
M = 5×4

     6     8     3     1
     5     4     7     3
     1     6     7    10
     4     2     8     2
     2     7     5     9
```

Read Matrix from Specified Sheet and Range Using Import Options

Preview the data from a spreadsheet file and import numerical data as a matrix from a specified sheet and range.

The spreadsheet file `airlinesmall_subset.xlsx` contains data in multiple worksheets for years between 1996 and 2008. Each worksheet has data for a given year. Preview the data from file `airlinesmall_subset.xlsx`. The `preview` function shows data from the first worksheet by default. The first eight variables in the file contain numerical data.

```
opts = detectImportOptions('airlinesmall_subset.xlsx');
preview('airlinesmall_subset.xlsx',opts)
```

```
ans=8×29 table
    Year    Month    DayofMonth    DayOfWeek    DepTime    CRSDepTime    ArrTime    CRS
    ____    _____    __________    _________    _______    __________    _______    ___

    1996      1          18            4         2117         2120        2305
    1996      1          12            5         1252         1245        1511
    1996      1          16            2         1441         1445        1708
    1996      1           1            1         2258         2300        2336
    1996      1           4            4         1814         1814        1901
    1996      1          31            3         1822         1820        1934
    1996      1          18            4          729          730         841
    1996      1          26            5         1704         1705        1829
```

Configure the values in the opts object to import `10` rows for the first five variables from the worksheet named `'2007'`.

```
opts.Sheet = '2007';
opts.SelectedVariableNames = [1:5];
opts.DataRange = '2:11';
M = readmatrix('airlinesmall_subset.xlsx',opts)
```

```
M = 10×5

        2007           1           2           2         711
        2007           1           3           3         652
        2007           1           4           4        1116
        2007           1           5           5         825
        2007           1           7           7        1411
        2007           1           8           1        1935
```

```
        2007           1           9           2        2005
        2007           1          11           4        1525
        2007           1          12           5        1133
        2007           1          13           6         922
```

Read Matrix from Specified Sheet and Range

Preview the data from a spreadsheet file and import numerical data, as a matrix, from a specified sheet and range.

The spreadsheet file `airlinesmall_subset.xlsx` contains data in multiple worksheets for years between 1996 and 2008. Each worksheet has data for a given year. Preview the data from file `airlinesmall_subset.xlsx`. The `preview` function shows data from the first worksheet by default. The first eight variables in the file contain numerical data.

```
opts = detectImportOptions('airlinesmall_subset.xlsx');
preview('airlinesmall_subset.xlsx',opts)
```

```
ans=8×29 table
    Year    Month    DayofMonth    DayOfWeek    DepTime    CRSDepTime    ArrTime    CRS
    ____    _____    __________    _________    _______    __________    _______    ___

    1996      1          18            4         2117         2120        2305
    1996      1          12            5         1252         1245        1511
    1996      1          16            2         1441         1445        1708
    1996      1           1            1         2258         2300        2336
    1996      1           4            4         1814         1814        1901
    1996      1          31            3         1822         1820        1934
    1996      1          18            4          729          730         841
    1996      1          26            5         1704         1705        1829
```

Import `10` rows of the first `5` variables from the worksheet named `'2007'`.

```
M = readmatrix('airlinesmall_subset.xlsx','Sheet','2007','Range','A2:E11')
```

```
M = 10×5

        2007           1           2           2         711
        2007           1           3           3         652
```

```
    2007           1           4           4        1116
    2007           1           5           5         825
    2007           1           7           7        1411
    2007           1           8           1        1935
    2007           1           9           2        2005
    2007           1          11           4        1525
    2007           1          12           5        1133
    2007           1          13           6         922
```

Input Arguments

`filename` — Name of file to read
character vector | string scalar

Name of the file to read, specified as a character vector or a string scalar.

Depending on the location of your file, `filename` can take on one of these forms.

Location	Form
Current folder or folder on the MATLAB path	Specify the name of the file in `filename`. **Example:** `'myFile.txt'`
File in a folder	If the file is not in the current folder or in a folder on the MATLAB path, then specify the full or relative path name in `filename`. **Example:** `'C:\myFolder\myFile.xlsx'` **Example:** `'dataDir\myFile.txt'`

<table>
<tr><th>Location</th><th>Form</th></tr>
<tr><td>Remote Location</td><td>If the file is stored at a remote location, then <code>filename</code> must contain the full path of the file specified as a uniform resource locator (URL) of the form:

<code>scheme_name://path_to_file/my_file.ext</code>

Based on your remote location, <code>scheme_name</code> can be one of the values in this table.
<table>
<tr><th>Remote Location</th><th><code>scheme_name</code></th></tr>
<tr><td>Amazon S3</td><td><code>s3</code></td></tr>
<tr><td>Windows Azure Blob Storage</td><td><code>wasb</code>, <code>wasbs</code></td></tr>
<tr><td>HDFS</td><td><code>hdfs</code></td></tr>
</table>
For more information, see “Work with Remote Data”.

Example: <code>'s3://bucketname/path_to_file/my_file.csv'</code></td></tr>
</table>

- If `filename` includes the file extension, then the importing function determines the file format from the extension. Otherwise, you must specify the `'FileType'` name-value pair arguments to indicate the type of file.
- On Windows systems with Microsoft Excel software, the importing function reads any Excel spreadsheet file format recognized by your version of Excel.
- If your system does not have Excel for Windows or if you are using MATLAB Online, the importing function operates with the `UseExcel` property set to `false`, and reads only `.xls, .xlsx, .xlsm, .xltx, and .xltm` files.
- For delimited text files, the importing function converts empty fields in the file to either `NaN` (for a numeric variable) or an empty character vector (for a text variable). All lines in the text file must have the same number of delimiters. The importing function ignores insignificant white space in the file.

Data Types: `char` | `string`

`opts` — File import options
`SpreadsheetImportOptions` | `DelimitedtextImportOptions` | `FixedWidthImportOptions`

File import options, specified as an `SpreadsheetImportOptions`, `DelimitedTextImportOptions`, or `FixedWidthImportOptions` object created by the `detectImportOptions` function. The `opts` object contains properties that control the data import process. For more information on the properties of each object, see the appropriate object page.

Type of Files	Output
Spreadsheet files	`SpreadsheetImportOptions` object
Text files	`DelimitedTextImportOptions` object
Fixed-width text files	`FixedWidthImportOptions` object

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'NumHeaderLines',5` indicates that the first five lines that precede the tabular data are header lines.

Text and Spreadsheet Files

`FileType` — Type of file
`'text'` | `'spreadsheet'`

Type of file, specified as the comma-separated pair consisting of `'FileType'` and `'text'` or `'spreadsheet'`.

Specify the `'FileType'` name-value pair argument when the `filename` does not include the file extension or if the extension is other than one of the following:

- `.txt`, `.dat`, or `.csv` for delimited text files
- `.xls`, `.xlsb`, `.xlsm`, `.xlsx`, `.xltm`, `.xltx`, or `.ods` for spreadsheet files

Example: `'FileType','text'`

Data Types: `char` | `string`

`NumHeaderLines` — Number of header lines
positive integer

Number of header lines in the file, specified as the comma-separated pair consisting of `'NumHeaderLines'` and a positive integer. If unspecified, the importing function automatically detects the number of header lines in the file.

Example: `'NumHeaderLines',7`

Data Types: `single` | `double`

`ExpectedNumVariables` — Expected number of variables
positive integer

Expected number of variables, specified as the comma-separated pair consisting of `'ExpectedNumVariables'` and a positive integer. If unspecified, the importing function automatically detects the number of variables.

Data Types: `single` | `double`

Range — Portion of data to read
character vector | string scalar | numeric vector

Portion of the data to read from text or spreadsheet files, specified as the comma separated pair consisting of `'Range'` and a character vector, string scalar, or numeric vector in one of these forms.

Ways to specify Range	Description
Starting Cell `'Cell'` or `[row col]`	Specify the starting cell for the data as a character vector or string scalar or a two element numeric vector. • Character vector or string scalar containing a column letter and row number using Excel `A1` notation. For example, `A5` is the identifier for the cell at the intersection of column `A` and row `5`. • Two element numeric vector of the form `[row col]` indicating the starting row and column. Using the starting cell, the importing function automatically detects the extent of the data by beginning the import at the start cell and ending at the last empty row or footer range. **Example:** `'A5'` or `[5 1]`

Ways to specify Range	Description
Rectangular Range `'Corner1:Corner2'` or `[r1 c1 r2 c2]`	Specify the exact range to read using the rectangular range in one of these forms. • `'Corner1:Corner2'` — Specify the range using `Corner1` and `Corner2` which are the two opposing corners that define the region to read in Excel `A1` notation. For example, `'C2:N15'`. • `[r1 c1 r2 c2]` — Specify the range using a four element numeric vector containing start-row, start-column, end-row, and end-column. For example, `[2 3 15 13]`. The importing function only reads the data contained in the specified range. Any empty fields within the specified range are imported as missing cells.
Row Range or Column Range `'Row1:Row2'` or `'Column1:Column2'`	Specify the range by identifying the beginning and ending rows using Excel row numbers. Using the specified row range, the importing function automatically detects the column extent by reading from the first nonempty column to the end of the data, and creates one variable per column. **Example:** `'5:500'` Alternatively, specify the range by identifying the beginning and ending columns using Excel column letters or numbers. Using the specified column range, the import function automatically detects the row extent by reading from the first nonempty row to the end of the data or the footer range. The number of columns in the specified range must match the number specified in the `ExpectedNumVariables` property. **Example:** `'A:K'`

Ways to specify Range	Description
Starting Row Number `n`	Specify the first row containing the data using the positive scalar row index. Using the specified row index, the importing function automatically detects the extent of the data by reading from the specified first row to the end of the data or the footer range. **Example:**5
Excel's Named Range `'NamedRange'`	In Excel, you can create names to identify ranges in the spreadsheet. For instance, you can select a rectangular portion of the spreadsheet and call it `'myTable'`. If such named ranges exist in a spreadsheet, then the importing function can read that range using its name. **Example:** `'Range','myTable'`
Unspecified or Empty `''`	If unspecified, the importing function automatically detects the used range. **Example:** `'Range',''` **Note:** *Used Range* refers to the rectangular portion of the spreadsheet that actually contains data. The importing function automatically detects the used range by trimming any leading and trailing rows and columns that do not contain data. Text that is only white space is considered data and is captured within the used range.

Data Types: `char` | `string` | `double`

`TreatAsMissing` — Text to interpret as missing data
character vector | string scalar | cell array of character vectors | string array

Text to interpret as missing data, specified as a character vector, string scalar, cell array of character vectors, or string array.

Example: `'TreatAsMissing',{'NA','TBD'}` instructs the importing function to treat any occurrence of `NA` or `TBD` as a missing fields.

Data Types: `char` | `string` | `cell`

`OutputType` — Output data type
character vector | string scalar

Output data type, specified as the comma-separated pair consisting of `'OutputType'` and a character vector or string scalar containing name of any of the data types in this table.

Type of Data	Output data type
Numeric	`'uint8'`, `'int8'`, `'int16'`, `'int32'`, `'int64'`, `'uint16'`, `'uint32'`, `'uint64'`, `'single'`, or `'double'`
Text	`'char'` or `'string'`
Other types	`'datetime'`, `'duration'`, or `'categorical'`

Example: `'OutputType','uint8'`

Data Types: `char` | `string`

Text Files Only

`Delimiter` — Field delimiter characters
character vector | string scalar | cell array of character vectors | string array

Field delimiter characters in a delimited text file, specified as a character vector, string scalar, cell array of character vectors, or string array.

Example: `'Delimiter','|'`

Example: `'Delimiter',{';','*'}`

Data Types: `char` | `string` | `cell`

`Whitespace` — Characters to treat as white space
character vector | string scalar

Characters to treat as white space, specified as a character vector or string scalar containing one or more characters.

Example: `'Whitespace',' _'`

Example: `'Whitespace','?!.,'`

`LineEnding` — End-of-line characters
`{'\n','\r','\r\n'}` (default) | character vector | string scalar | cell array of character vectors | string array

End-of-line characters, specified as a character vector, string scalar, cell array of character vectors, or string array.

Example: `'LineEnding','\n'`

Example: `'LineEnding','\r\n'`

Example: `'LineEnding',{'\b',':'}`

Data Types: `char` | `string` | `cell`

`CommentStyle` — Style of comments
character vector | string scalar | cell array of character vectors | string array

Style of comments, specified as a character vector, string scalar, cell array of character vectors, or string array.

For example, to ignore the text following a percent sign on the same line, specify `CommentStyle` as `'%'`.

Example: `'CommentStyle',{'/*'}`

Data Types: `char` | `string` | `cell`

`Encoding` — Character encoding scheme
`'system'` | `'UTF-8'` | `'ISO-8859-1'` | `'windows-1251'` | `'windows-1252'` | ...

Character encoding scheme associated with the file, specified as the comma-separated pair consisting of `'Encoding'` and `'system'` or a standard character encoding scheme name, such as one of the values in this table.

'Big5'	'ISO-8859-1'	'windows-847'
'Big5-HKSCS'	'ISO-8859-2'	'windows-949'
'CP949'	'ISO-8859-3'	'windows-1250'
'EUC-KR'	'ISO-8859-4'	'windows-1251'
'EUC-JP'	'ISO-8859-5'	'windows-1252'
'EUC-TW'	'ISO-8859-6'	'windows-1253'

'GB18030'	'ISO-8859-7'	'windows-1254'
'GB2312'	'ISO-8859-8'	'windows-1255'
'GBK'	'ISO-8859-9'	'windows-1256'
'IBM866'	'ISO-8859-11'	'windows-1257'
'KOI8-R'	'ISO-8859-13'	'windows-1258'
'KOI8-U'	'ISO-8859-15'	'US-ASCII'
	'Macintosh'	'UTF-8'
	'Shift_JIS'	

Example: `'Encoding','system'` uses the system default encoding.

Data Types: `char` | `string`

`DateLocale` — Locale for reading dates
character vector | string scalar

Locale for reading dates, specified as the comma-separated pair consisting of `'DateLocale'` and a character vector or a string scalar of the form *xx_YY*, where:

- *YY* is an uppercase ISO 3166-1 alpha-2 code indicating a country.
- *xx* is a lowercase ISO 639-1 two-letter code indicating a language.

For a list of common values for the locale, see the `Locale` name-value pair argument for the `datetime` function.

When using the `%D` format specifier to read text as `datetime` values, use `DateLocale` to specify the locale in which the importing function should interpret month and day-of-week names and abbreviations.

If you specify the `DateLocale` argument in addition to `opts` the import options, then the importing function uses the specified value for the `DateLocale` argument, overriding the locale defined in the import options.

Example: `'DateLocale','ja_JP'`

`DecimalSeparator` — Characters indicating decimal separator
character vector | string scalar

Characters indicating the decimal separator in numeric variables, specified as a character vector or string scalar. The importing function uses the characters specified in the

`DecimalSeparator` name-value pair to distinguish the integer part of a number from the decimal part.

When converting to integer data types, numbers with a decimal part are rounded to the nearest integer.

Example: If name-value pair is specified as `'DecimalSeparator',','`, then the importing function imports the text `"3,14159"` as the number `3.14159`.

Data Types: `char` | `string`

`ThousandsSeparator` — Characters that indicate thousands grouping
character vector | string scalar

Characters that indicate the thousands grouping in numeric variables, specified as a character vector or string scalar. The thousands grouping characters act as visual separators, grouping the number at every three place values. The importing function uses the characters specified in the `ThousandsSeparator` name-value pair to interpret the numbers being imported.

Example: If name-value pair is specified as `'ThousandsSeparator',','`, then the importing function imports the text `"1,234,000"` as `1234000`.

Data Types: `char` | `string`

`TrimNonNumeric` — Remove nonnumeric characters
`false` (default) | `true`

Remove nonnumeric characters from a numeric variable, specified as a logical `true` or `false`.

Example: If name-value pair is specified as `'TrimNonNumeric',true`, then the importing function reads `'$500/-'` as `500`.

Data Types: `logical`

`ConsecutiveDelimitersRule` — Procedure to handle consecutive delimiters
`'split'` | `'join'` | `'error'`

Procedure to handle consecutive delimiters in a delimited text file, specified as one of the values in this table.

Consecutive Delimiters Rule	Behavior
`'split'`	Split the consecutive delimiters into multiple fields.
`'join'`	Join the delimiters into one delimiter.
`'error'`	Return an error and abort the import operation.

Data Types: `char` | `string`

`LeadingDelimitersRule` — Procedure to manage leading delimiters
`'keep'` | `'ignore'` | `'error'`

Procedure to manage leading delimiters in a delimited text file, specified as one of the values in this table.

Leading Delimiters Rule	Behavior
`'keep'`	Keep the delimiter.
`'ignore'`	Ignore the delimiter.
`'error'`	Return an error and abort the import operation.

Spreadsheet Files Only

`Sheet` — Sheet to read from
`''` empty character array (default) | character vector | string scalar | positive scalar integer

Sheet to read from, specified as an empty character array, a character vector or string scalar containing the sheet name, or a positive scalar integer denoting the sheet index. Based on the value specified for the `Sheet` property, the import function behaves as described in the table.

Specification	Behavior
`''` (default)	Import data from the first sheet.
Name	Import data from the matching sheet name, regardless of order of sheets in the spreadsheet file.
Integer	Import data from sheet in the position denoted by the integer, regardless of the sheet names in the spreadsheet file.

Data Types: `char` | `string` | `single` | `double`

`UseExcel` — Flag to start instance of Microsoft Excel for Windows
`false` (default) | `true`

Flag to start an instance of Microsoft Excel for Windows when reading spreadsheet data, specified as the comma-separated pair consisting of `'UseExcel'` and either `true`, or `false`.

You can set the `'UseExcel'` parameter to one of these values:

- `true` — The importing function starts an instance of Microsoft Excel when reading the file.
- `false` — The importing function does not start an instance of Microsoft Excel when reading the file. When operating in this mode, the importing function functionality differs in the support of file formats and interactive features, such as formulas and macros.

UseExcel	**true**	**false**
Supported file formats	`.xls, .xlsx, .xlsm, .xltx, .xltm, .xlsb, .ods`	`.xls, .xlsx, .xlsm, .xltx, .xltm`
Support for interactive features, such as formulas and macros	Yes	No

When reading from spreadsheet files on Windows platforms, if you want to start an instance of Microsoft Excel, then set the `'UseExcel'` parameter to `true`.

See Also

`readcell` | `readtable` | `readtimetable` | `readvars` | `writematrix`

Introduced in R2019a

readtable

Create table from file

Syntax

```
T = readtable(filename)
T = readtable(filename,opts)
T = readtable( ___ ,Name,Value)
```

Description

`T = readtable(filename)` creates a table by reading column oriented data from a file.

`readtable` determines the file format from the file extension:

- `.txt`, `.dat`, or `.csv` for delimited text files
- `.xls`, `.xlsb`, `.xlsm`, `.xlsx`, `.xltm`, `.xltx`, or `.ods` for spreadsheet files

`readtable` creates one variable in `T` for each column in the file and reads variable names from the first row of the file. By default, the variables created are `double` when the entire column is numeric, or cell arrays of character vectors when any element in a column is not numeric.

`T = readtable(filename,opts)` creates a table using the import options `opts`.

`T = readtable( ___ ,Name,Value)` creates a table from a file with additional options specified by one or more name-value pair arguments. For example, you can specify whether `readtable` reads the first row of the file as variable names or as data.

To set specific import options for your data, you can either use the `opts` object or you can specify name-value pairs. When you specify name-value pairs in addition to `opts`, then `readtable` supports only these name-value pairs:

- Text and spreadsheet parameters — `ReadVariableNames`, `ReadRowNames`
- Text only parameters — `DateLocale`, `Encoding`

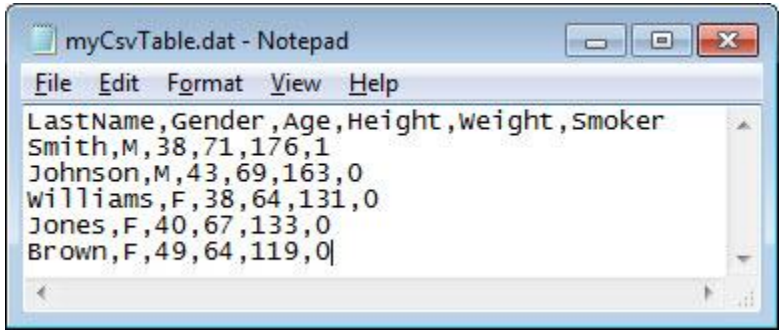
- Spreadsheet only parameters — `Sheet`, `UseExcel`

Examples

Create Table from Text File

Load the file `myCsvTable.dat` and preview its contents in a text editor. A screen shot is shown below. Notice that the file contains comma-separated column oriented data.

```
filename = fullfile(matlabroot,'examples','matlab','myCsvTable.dat');
```



Create a table from the comma-separated text file. The resulting table `T` contains one variable for each column in the file and `readtable` treats the entries in the first line of the file as variable names.

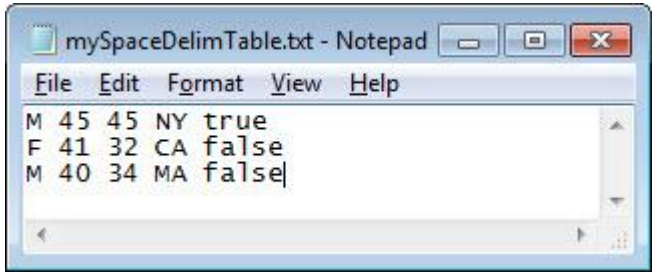
```
T = readtable(filename)
```

```
T=5×6 table
      LastName       Gender    Age    Height    Weight    Smoker
    ____________     ______    ___    ______    ______    ______

    {'Smith'   }     {'M'}     38       71        176        1
    {'Johnson' }     {'M'}     43       69        163        0
    {'Williams'}     {'F'}     38       64        131        0
    {'Jones'   }     {'F'}     40       67        133        0
    {'Brown'   }     {'F'}     49       64        119        0
```

Create Table from Text File without Column Headings

Load the file `mySpaceDelimTable.txt` and preview its contents in a text editor. A screen shot is shown below. Notice that the file contains space delimited, column oriented data.

```
filename = fullfile(matlabroot,'examples','matlab','mySpaceDelimTable.txt');
```



Create a table from the space delimited text file that does not contain variable names as column headings.

```
T = readtable(filename,...
    'Delimiter',' ','ReadVariableNames',false)
```

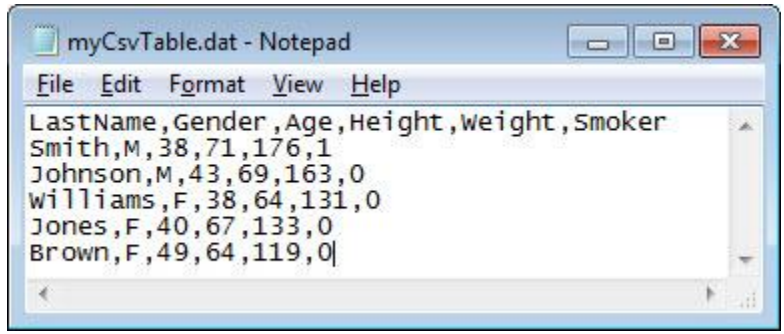
```
T=3×5 table
    Var1     Var2    Var3     Var4       Var5
    _____    ____    ____    ______    _________

    {'M'}     45      45     {'NY'}    {'true' }
    {'F'}     41      32     {'CA'}    {'false'}
    {'M'}     40      34     {'MA'}    {'false'}
```

T contains default variable names.

Create and Format Table from Text File

Load the file `myCsvTable.dat` and preview its contents in a text editor. A screen shot is shown below. Notice that the file contains comma-separated column oriented data.

```
filename = fullfile(matlabroot,'examples','matlab','myCsvTable.dat');
```



Create a table from the comma-separated text file. Import the first two columns as character vectors, the third column as `uint32`, and the next two columns as double-precision, floating-point numbers. Import the entries of the last column as character vectors.

```
T = readtable(filename,'Format','%s%s%u%f%f%s')
```

```
T=5×6 table
      LastName       Gender    Age    Height    Weight    Smoker
    ____________     ______    ___    ______    ______    ______

    {'Smith'   }     {'M'}     38       71       176      {'1'}
    {'Johnson' }     {'M'}     43       69       163      {'0'}
    {'Williams'}     {'F'}     38       64       131      {'0'}
    {'Jones'   }     {'F'}     40       67       133      {'0'}
    {'Brown'   }     {'F'}     49       64       119      {'0'}
```

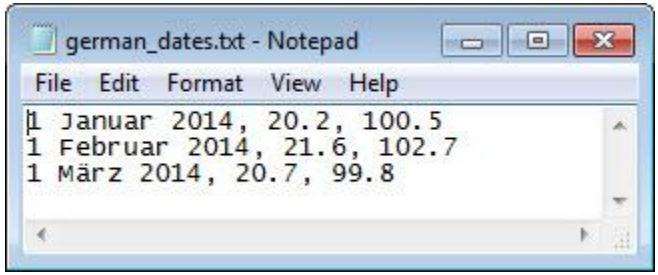
The conversion specifiers are `%s` for a cell array of character vectors, `%f` for `double`, and `%u` for `uint32`.

Read Foreign-Language Dates from Text File

Read German dates from a file and add them to a table as English dates.

Load the file `german_dates.txt` and preview its contents in a text editor. A screen shot is shown below. Notice that the first column of values contains dates in German and the second and third columns are numeric values.

```
filename = fullfile(matlabroot,'examples','matlab','german_dates.txt');
```



Read the sample file using `readtable`. The conversion specifiers is `%D` dates and `%f` for floating-point values. Specify the file encoding using the `FileEncoding` name-value pair argument. Specify the format and locale of the dates using the `DateLocale` name-value pair argument.

```
T = readtable(filename,'ReadVariableNames',false,...
    'Format','%{dd MMMM yyyy}D %f %f',...
    'FileEncoding','ISO-8859-15',...
    'DateLocale','de_DE')
```

```
T=3×3 table
          Var1          Var2    Var3
    ________________    ____    _____

     01 January 2014    20.2    100.5
    01 February 2014    21.6    102.7
       01 March 2014    20.7     99.8
```

Create Table from Spreadsheet Including Row Names

Create a table from a spreadsheet that contains variable names in the first row and row names in the first column.

```
T = readtable('patients.xls','ReadRowNames',true);
```

Display the first five rows and first four variables of the table.

```
T(1:5,1:4)
```

```
ans=5×4 table
                  Gender      Age              Location                Height
                 __________   ___   ______________________________    ______

    Smith       {'Male'  }    38    {'County General Hospital'  }      71
    Johnson     {'Male'  }    43    {'VA Hospital'              }      69
    Williams    {'Female'}    38    {'St. Mary's Medical Center'}      64
    Jones       {'Female'}    40    {'VA Hospital'              }      67
    Brown       {'Female'}    49    {'County General Hospital'  }      64
```

View the `DimensionNames` property of the table.

```
T.Properties.DimensionNames
```

```
ans = 1x2 cell array
    {'LastName'}    {'Variables'}
```

`'LastName'` is the name in the first column of the first row of the spreadsheet.

Read Specific Range of Data from Spreadsheet

Create a table using data from a specified region of the spreadsheet `patients.xls`. Use the data from the 5-by-3 rectangular region between the corners `C2` and `E6`. Do not use the first row of this region as variable names.

```
T = readtable('patients.xls',...
    'Range','C2:E6',...
    'ReadVariableNames',false)
```

```
T =

    Var1                Var2                Var3
    ____    ____________________________    ____

    38      'County General Hospital'       71
    43      'VA Hospital'                   69
    38      'St. Mary's Medical Center'     64
    40      'VA Hospital'                   67
    49      'County General Hospital'       64
```

T contains default variable names.

Detect and Use Import Options for Text Files

Create import options, tailor the data types for multiple variables, and then read the data.

Create an import options object from a text file.

```
opts = detectImportOptions('airlinesmall.csv')
```

```
opts 
  DelimitedTextImportOptions with properties:

   Format Properties:
                   Delimiter: {','}
                  Whitespace: '\b\t '
                  LineEnding: {'\n'  '\r'  '\r\n'}
                CommentStyle: {}
    ConsecutiveDelimitersRule: 'split'
        LeadingDelimitersRule: 'keep'
                EmptyLineRule: 'skip'
                     Encoding: 'windows-1252'

   Replacement Properties:
                 MissingRule: 'fill'
             ImportErrorRule: 'fill'
            ExtraColumnsRule: 'addvars'

   Variable Import Properties: Set types by name using setvartype
               VariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
               VariableTypes: {'double', 'double', 'double' ... and 26 more}
        SelectedVariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
             VariableOptions: Show all 29 VariableOptions
    Access VariableOptions sub-properties using setvaropts/getvaropts
        PreserveVariableNames: false

   Location Properties:
                   DataLines: [2 Inf]
           VariableNamesLine: 1
              RowNamesColumn: 0
           VariableUnitsLine: 0
     VariableDescriptionsLine: 0
```

```
    To display a preview of the table, use preview
```

Examine the `Type` property of variables `TaxiIn` and `TaxiOut`.

```
getvaropts(opts,{'TaxiIn','TaxiOut'})
```

```
ans = 
  1x2 TextVariableImportOptions array with properties:

    Name
    Type
    FillValue
    TreatAsMissing
    QuoteRule
    Prefixes
    Suffixes
    EmptyFieldRule
    WhitespaceRule
```

Change the type of the variables `TaxiIn` and `TaxiOut` to `double`.

```
 opts = setvartype(opts,{'TaxiIn','TaxiOut'},'double');
```

Specify the subset of variables to import and examine.

```
opts.SelectedVariableNames = {'TaxiIn','TaxiOut'};
```

Use the `readtable` function along with the options object to import the selected variables. Display a summary of the table.

```
T = readtable('airlinesmall.csv',opts);
summary(T)
```

```
Variables:

    TaxiIn: 123523x1 double

        Values:

            Min             0
            Median          5
            Max             1451
            NumMissing      37383
```

```
    TaxiOut: 123523x1 double

        Values:

            Min             0
            Median          13
            Max             755
            NumMissing      37364
```

Detect and Use Import Options for Spreadsheet Files

Detect import options for a spreadsheet file, specify the variables to import, and then read the data.

Create an import options object from a file.

```
opts = detectImportOptions('patients.xls')
```

```
opts = 
  SpreadsheetImportOptions with properties:

   Sheet Properties:
                        Sheet: ''

   Replacement Properties:
                  MissingRule: 'fill'
              ImportErrorRule: 'fill'

   Variable Import Properties: Set types by name using setvartype
                VariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
                VariableTypes: {'char', 'char', 'double' ... and 7 more}
           SelectedVariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
              VariableOptions: Show all 10 VariableOptions
    Access VariableOptions sub-properties using setvaropts/getvaropts
           PreserveVariableNames: false

   Range Properties:
                    DataRange: 'A2' (Start Cell)
            VariableNamesRange: 'A1'
                RowNamesRange: ''
            VariableUnitsRange: ''
```

```
    VariableDescriptionsRange: ''
    To display a preview of the table, use preview
```

Modify the options object to specify which variables to import.

```
opts.SelectedVariableNames = {'Systolic','Diastolic'};
```

Use `readtable` along with the options object to import the specified variables.

```
T = readtable('patients.xls',opts);
summary(T)
```

```
Variables:

    Systolic: 100x1 double

        Values:

            Min          109
            Median       122
            Max          138

    Diastolic: 100x1 double

        Values:

            Min            68
            Median       81.5
            Max            99
```

Read Tabular Data Containing Arbitrary Variable Names

You can read tabular data while preserving variable names that include any characters, including spaces and non-ASCII characters. First, create a table with arbitrary variable names and write the table to a text file. Then, read the tabular data back while preserving the original variable names.

Create a table containing arbitrary variable names and write the tabular data to the file `'sampletable.txt'`.

```
LastName = {'Sanchez';'Johnson';'Li';'Diaz';'Brown'};
Age = [38;43;38;40;49];
```

```
Smoker = logical([1;0;1;0;1]);
varNames = {'Last Name','Age','Smoker (1 or 0)'};
T = table(LastName,Age,Smoker,'VariableNames',varNames)
```

```
T=5×3 table
     Last Name     Age    Smoker (1 or 0)
    ___________    ___    _______________

    {'Sanchez'}    38          true
    {'Johnson'}    43          false
    {'Li'     }    38          true
    {'Diaz'   }    40          false
    {'Brown'  }    49          true
```

```
writetable(T,'sampletable.txt')
```

Read the tabular data back using `readtable`. By default, MATLAB® converts any variable names that include spaces and non-ASCII characters into valid MATLAB® identifiers. For example, MATLAB® converts the variable names `'Last Name'` to `'LastName'`, and `'Smoker (1 or 0)'` to `'Smoker_1or0_'`. To read the tabular data while preserving variable names, set the `'PreserveVariableNames'` parameter to `true`.

```
T_preserve = readtable('sampletable.txt','PreserveVariableNames',true)
```

```
T_preserve=5×3 table
     Last Name     Age    Smoker (1 or 0)
    ___________    ___    _______________

    {'Sanchez'}    38            1
    {'Johnson'}    43            0
    {'Li'     }    38            1
    {'Diaz'   }    40            0
    {'Brown'  }    49            1
```

Input Arguments

`filename` — Name of file to read

character vector | string scalar

Name of the file to read, specified as a character vector or a string scalar.

Depending on the location of your file, `filename` can take on one of these forms.

Location	Form
Current folder or folder on the MATLAB path	Specify the name of the file in `filename`. **Example:** `'myFile.txt'`
File in a folder	If the file is not in the current folder or in a folder on the MATLAB path, then specify the full or relative path name in `filename`. **Example:** `'C:\myFolder\myFile.xlsx'` **Example:** `'dataDir\myFile.txt'`
Remote Location	If the file is stored at a remote location, then `filename` must contain the full path of the file specified as a uniform resource locator (URL) of the form: *`scheme_name`*`://path_to_file/my_file.ext` Based on your remote location, *`scheme_name`* can be one of the values in this table. **Remote Location** / **`scheme_name`**: Amazon S3 — `s3` Windows Azure Blob Storage — `wasb`, `wasbs` HDFS — `hdfs` For more information, see “Work with Remote Data”. **Example:** `'s3://bucketname/path_to_file/my_file.csv'`

- If `filename` includes the file extension, then the importing function determines the file format from the extension. Otherwise, you must specify the `'FileType'` name-value pair arguments to indicate the type of file.
- On Windows systems with Microsoft Excel software, the importing function reads any Excel spreadsheet file format recognized by your version of Excel.

- If your system does not have Excel for Windows or if you are using MATLAB Online, the importing function operates with the `UseExcel` property set to `false`, and reads only `.xls, .xlsx, .xlsm, .xltx, and .xltm` files.
- For delimited text files, the importing function converts empty fields in the file to either `NaN` (for a numeric variable) or an empty character vector (for a text variable). All lines in the text file must have the same number of delimiters. The importing function ignores insignificant white space in the file.

Data Types: `char` | `string`

`opts` — File import options
`SpreadsheetImportOptions` | `DelimitedtextImportOptions` | `FixedWidthImportOptions`

File import options, specified as an `SpreadsheetImportOptions`, `DelimitedTextImportOptions`, or `FixedWidthImportOptions` object created by the `detectImportOptions` function. The `opts` object contains properties that control the data import process. For more information on the properties of each object, see the appropriate object page.

Type of Files	Output
Spreadsheet files	`SpreadsheetImportOptions` object
Text files	`DelimitedTextImportOptions` object
Fixed-width text files	`FixedWidthImportOptions` object

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'ReadVariableNames',false` indicates that the first row of the file does not correspond to variable names.

When reading:

- Text files, only these parameter names apply: `FileType`, `ReadVariableNames`, `ReadRowNames`, `TreatAsEmpty`, `DatetimeType`, `Delimiter`, `HeaderLines`,

Format, EmptyValue, MultipleDelimsAsOne, CollectOutput, CommentStyle, ExpChars, EndOfLine, DateLocale, and Encoding.

- Spreadsheet files, only these parameter names apply: FileType, ReadVariableNames, ReadRowNames, TreatAsEmpty, DatetimeType, Sheet, Range, UseExcel, and TextType.
- Text or Spreadsheet files with the opts import options, only these parameter names apply: ReadVariableNames, ReadRowNames, DateLocale, Encoding, Sheet, and UseExcel.

Text and Spreadsheet Files

FileType — Type of file

'text' | 'spreadsheet'

Type of file, specified as the comma-separated pair consisting of 'FileType' and 'text' or 'spreadsheet'.

Specify the 'FileType' name-value pair argument when the filename does not include the file extension or if the extension is other than one of the following:

- .txt, .dat, or .csv for delimited text files
- .xls, .xlsb, .xlsm, .xlsx, .xltm, .xltx, or .ods for spreadsheet files

Example: 'FileType','text'

Data Types: char | string

ReadVariableNames — Read first row as variable names

true | false | 1 | 0

Indicator for reading the first row as variable names, specified as the comma-separated pair consisting of 'ReadVariableNames' and either true, false, 1, or 0. If unspecified, readtable automatically detects the presence of variable names.

Indicator	Description
true	Use when the first row of the region to read contains the variable names for the table. readtable creates a variable, with the detected variable name, for each column in T.

Indicator	Description
`false`	Use when the first row of the region to read contains data in the table. `readtable` creates default variable names of the form `'Var1',...,'VarN'`, where `N` is the number of variables.
unspecified	When left unspecified, `readtable` automatically detects `true` or `false` and proceeds accordingly.

Note: If both the `'ReadVariableNames'` and `'ReadRowNames'` logical indicators are `true`, then `readtable` saves the name in the first column of the first row of the region to read as the first dimension name in the property, `T.Properties.DimensionNames`.

If you specify the `ReadVariableNames` argument in addition to `opts` the import options, then the `readtable` behavior changes based on the specification:

- If `ReadVariableNames` is `true`, then read the variable names from the specified file by using the `VariableNamesRange` or the `VariableNamesLine` property of the import options object.
- If `ReadVariableNames` is `false`, then read the variable names from the `VariableNames` property of the import options object.

`ReadRowNames` — Indicator for reading the first column as row names
`false` (default) | `true` | `0` | `1`

Indicator for reading first column as row names, specified as the comma-separated pair consisting of `'ReadRowNames'` and either `false`, `true`, `0`, or `1`.

Indicator	Description
`false`	Use when the first column of the region to read contains data, and not the row names for the table.
`true`	Use when the first column of the region to read contains the row names for the table.
unspecified	When left unspecified, `readtable` assumes `false`.

Note: If both the `'ReadVariableNames'` and `'ReadRowNames'` logical indicators are `true`, then `readtable` saves the name in the first column of the first row of the region to read as the first dimension name in the property, `T.Properties.DimensionNames`.

If you specify the `ReadRowNames` argument in addition to `opts` the import options , then the `readtable` behavior changes based on the specification:

- If `ReadRowNames` is `true`, then read the row names from the specified file by using the `RowNamesRange` or the `RowNameColumn` property of the import options object.
- If `ReadRowNames` is `false`, then do not import row names.

`TreatAsEmpty` — Placeholder text to treat as empty value
character vector | cell array of character vectors | string | string array

Placeholder text to treat as an empty value, specified as the comma-separated pair consisting of `'TreatAsEmpty'` and a character vector, cell array of character vectors, string, or string array. Table elements corresponding to these characters are set to `NaN`.

`'TreatAsEmpty'` only applies to numeric columns in the file, and `readtable` does not accept numeric literals, such as `'-99'`.

Example: `'TreatAsEmpty','N/A'` or `'TreatAsEmpty',"N/A"` sets `N/A` within numeric columns to `NaN`.

Example: `'TreatAsEmpty',{'.','NA','N/A'}` or `'TreatAsEmpty',["."," NA","N/A"]` sets `.`, `NA` and `N/A` within numeric columns to `NaN`.

Data Types: `char` | `string`

`TextType` — Type for imported text data
`'char'` (default) | `'string'`

Type for imported text data, specified as the comma-separated pair consisting of `'TextType'` and either `'char'` or `'string'`.

- `'char'` — Import text data into MATLAB as character vectors.
- `'string'` — Import text data into MATLAB as string arrays.

Example: `'TextType','char'`

`DatetimeType` — Type for imported date and time data
`'datetime'` (default) | `'text'` | `'exceldatenum'` (spreadsheet files only)

Type for imported date and time data, specified as the comma-separated pair consisting of `'DatetimeType'` and one of these values: `'datetime'`, `'text'`, or `'exceldatenum'`. The value `'exceldatenum'` is applicable only for spreadsheet files, and is not valid for text files.

Value	Type for Imported Date and Time Data
`'datetime'`	MATLAB `datetime` data type For more information, see `datetime`.
`'text'`	If `'DatetimeType'` is specified as `'text'`, then the type for imported date and time data depends on the value specified in the `'TextType'` parameter: • If `'TextType'` is set to `'char'`, then the importing function returns dates as a cell array of character vectors. • If `'TextType'` is set to `'string'`, then the importing function returns dates as an array of strings.
`'exceldatenum'`	Excel serial date numbers A serial date number is a single number equal to the number of days from a given reference date. Excel serial date numbers use a different reference date than MATLAB serial date numbers. For more information on Excel dates, see `https://support.microsoft.com/en-us/kb/214330`.

Data Types: `char` | `string`

`PreserveVariableNames` — Flag to preserve variable names
`false` (default) | `true`

Flag to preserve variable names, specified as the comma-separated pair consisting of `PreserveVariableNames` and either `true`, or `false`.

- `true` — Preserve variable names that are not valid MATLAB identifiers such as variable names that include spaces and non-ASCII characters.
- `false` — Convert invalid variable names (as determined by the `isvarname` function) to valid MATLAB identifiers.

Starting in R2019b, variable names and row names can include any characters, including spaces and non-ASCII characters. Also, they can start with any characters, not just letters. Variable and row names do not have to be valid MATLAB identifiers (as determined by the `isvarname` function). To preserve these variable names and row names, set `PreserveVariableNames` to `true`.

Text Files Only

`Delimiter` — Field delimiter character

character vector | cell array of character vectors | string

Field delimiter character, specified as the comma-separated pair consisting of `'Delimiter'` and a character vector, a cell array of character vectors, or a string. Specify `Delimiter` using any valid character such as a comma `','` or a period `'.'`.

This table lists some commonly used field delimiter characters.

Specifier	Field Delimiter
`','` `'comma'`	Comma
`' '` `'space'`	Space
`'\t'` `'tab'`	Tab
`';'` `'semi'`	Semicolon
`'|'` `'bar'`	Vertical bar
unspecified	If unspecified, `readtable` automatically detects the delimiter.

To treat consecutive delimiters as a single delimiter, specify `Delimiter` as a cell array of character vectors. In addition, you must also specify the `MultipleDelimsAsOne` option.

Example: `'Delimiter',','` or `'Delimiter','comma'`

Data Types: `char` | `string` | `cell`

`HeaderLines` — Lines to skip

positive integer

Lines to skip at beginning of the file, specified as the comma-separated pair consisting of `'HeaderLines'` and a positive integer. If unspecified, `readtable` automatically detects the number of lines to skip.

Data Types: `single` | `double`

`Format` — Column format
character vector | string

Column format of the file, specified as the comma-separated pair consisting of `'Format'` and a character vector or a string scalar of one or more conversion specifiers. The conversion specifiers are the same as the specifiers accepted by the `textscan` function.

Specifying the format can significantly improve speed for some large files. If you do not specify a value for `Format`, then `readtable` uses `%q` to interpret nonnumeric columns. The `%q` specifier reads the text and omits double quotation marks (`"`) if appropriate.

By default, the variables created are either `double` or cell array of character vectors, depending on the data. If the entire column is numeric, variables are imported as `double`. If any element in a column is not numeric, the variables are imported as cell arrays of character vectors.

Data Types: `char` | `string`

`EmptyValue` — Returned value for empty numeric fields
`NaN` (default) | scalar

Returned value for empty numeric fields in delimited text files, specified as the comma-separated pair consisting of `'EmptyValue'` and a scalar.

`MultipleDelimsAsOne` — Multiple delimiter handling
`0 (false)` (default) | `1 (true)`

Multiple delimiter handling, specified as the comma-separated pair consisting of `'MultipleDelimsAsOne'` and either `true` or `false`. If `true`, then the importing function treats consecutive delimiters as a single delimiter. Repeated delimiters separated by white-space are also treated as a single delimiter. You must also specify the `Delimiter` option.

Example: `'MultipleDelimsAsOne',1`

`CollectOutput` — Logical indicator determining data concatenation
`false` (default) | `true`

Logical indicator determining data concatenation, specified as the comma-separated pair consisting of `'CollectOutput'` and either `true` or `false`. If `true`, then the importing function concatenates consecutive output cells of the same fundamental MATLAB class into a single array.

`CommentStyle` — Symbols designating text to ignore
character vector | cell array of character vectors | string | string array

Symbols designating text to ignore, specified as the comma-separated pair consisting of `'CommentStyle'` and a character vector, cell array of character vectors, string, or string array.

For example, specify a character such as `'%'` to ignore text following the symbol on the same line. Specify a cell array of two character vectors, such as `{'/*','*/'}`, to ignore any text between those sequences.

MATLAB checks for comments only at the start of each field, not within a field.

Example: `'CommentStyle',{'/*','*/'}`

Data Types: `char` | `string`

`ExpChars` — Exponent characters
`'eEdD'` (default) | character vector | string

Exponent characters, specified as the comma-separated pair consisting of `'ExpChars'` and a character vector or string. The default exponent characters are `e`, `E`, `d`, and `D`.

Data Types: `char` | `string`

`EndOfLine` — End-of-line characters
character vector | string

End-of-line characters, specified as the comma-separated pair consisting of `'EndOfLine'` and a character vector or string. The character vector must be `'\r\n'` or it must specify a single character. Common end-of-line characters are a newline character (`'\n'`) or a carriage return (`'\r'`). If you specify `'\r\n'`, then the importing function treats any of `\r`, `\n`, and the combination of the two (`\r\n`) as end-of-line characters.

The default end-of-line sequence is `\n`, `\r`, or `\r\n`, depending on the contents of your file.

If there are missing values and an end-of-line sequence at the end of the last line in a file, then the importing function returns empty values for those fields. This ensures that individual cells in output cell array, `C`, are the same size.

Example: `'EndOfLine',':'`

Data Types: `char` | `string`

`DateLocale` — Locale for reading dates
character vector | string scalar

Locale for reading dates, specified as the comma-separated pair consisting of `'DateLocale'` and a character vector or a string scalar of the form *xx_YY*, where:

- *YY* is an uppercase ISO 3166-1 alpha-2 code indicating a country.
- *xx* is a lowercase ISO 639-1 two-letter code indicating a language.

For a list of common values for the locale, see the `Locale` name-value pair argument for the `datetime` function.

When using the `%D` format specifier to read text as `datetime` values, use `DateLocale` to specify the locale in which the importing function should interpret month and day-of-week names and abbreviations.

If you specify the `DateLocale` argument in addition to `opts` the import options, then the importing function uses the specified value for the `DateLocale` argument, overriding the locale defined in the import options.

Example: `'DateLocale','ja_JP'`

`Encoding` — Character encoding scheme
`'UTF-8'` | `'ISO-8859-1'` | `'windows-1251'` | `'windows-1252'` | ...

Character encoding scheme associated with the file, specified as the comma-separated pair consisting of `'Encoding'` and `'system'` or a standard character encoding scheme name like one of the values in this table. When you do not specify any encoding or specify encoding as `'system'`, the `readtable` function uses your system default encoding to read the file.

`'Big5'`	`'ISO-8859-1'`	`'windows-874'`
`'Big5-HKSCS'`	`'ISO-8859-2'`	`'windows-949'`

'CP949'	'ISO-8859-3'	'windows-1250'
'EUC-KR'	'ISO-8859-4'	'windows-1251'
'EUC-JP'	'ISO-8859-5'	'windows-1252'
'EUC-TW'	'ISO-8859-6'	'windows-1253'
'GB18030'	'ISO-8859-7'	'windows-1254'
'GB2312'	'ISO-8859-8'	'windows-1255'
'GBK'	'ISO-8859-9'	'windows-1256'
'IBM866'	'ISO-8859-11'	'windows-1257'
'KOI8-R'	'ISO-8859-13'	'windows-1258'
'KOI8-U'	'ISO-8859-15'	'US-ASCII'
	'Macintosh'	'UTF-8'
	'Shift_JIS'	

If you specify the `Encoding` argument in addition to `opts` the import options, then the `readtable` function uses the specified value for `Encoding` argument, overriding the encoding defined in the import options.

Example: `'Encoding'`, `'UTF-8'`

Example: `'Encoding'`, `'system'` uses the system default encoding.

Data Types: `char` | `string`

`DurationType` — Output data type of duration data
`'duration'` (default) | `'text'`

Output data type of duration data from text files, specified as the comma-separated pair consisting of `'DurationType'` and either `'duration'` or `'text'`.

Value	Type for Imported Duration Data
`'duration'`	MATLAB `duration` data type For more information, see `duration`.

Value	Type for Imported Duration Data
`'text'`	If `'DurationType'` is specified as `'text'`, then the type for imported duration data depends on the value specified in the `'TextType'` parameter: • If `'TextType'` is `'char'`, then the importing function returns duration data as a cell array of character vectors. • If `'TextType'` is `'string'`, then the importing function returns duration data as an array of strings.

Data Types: `char` | `string`

Spreadsheet Files Only

Sheet — Worksheet to read

1 (default) | positive integer | character vector | string

Worksheet to read, specified as the comma-separated pair consisting of `'Sheet'` and a positive integer indicating the worksheet index or a character vector or string containing the worksheet name. The worksheet name cannot contain a colon (`:`). To determine the names of sheets in a spreadsheet file, use `sheets = sheetnames(filename)`. For more information, see `sheetnames`.

If you specify the `Sheet` argument in addition to `opts` the import options, then the `readtable` function uses the specified value for `Sheet` argument, overriding the sheet name defined in the import options.

Example: `'Sheet'`, 2

Example: `'Sheet'`, `'MySheetName'`

Example: `'Sheet'`, `"MySheetName"`

Data Types: `char` | `string` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Range — Portion of worksheet to read

character vector | string scalar

Portion of the worksheet to read, indicated as a rectangular area specified by a comma separated pair consisting of `'Range'` and a character vector or string scalar in one of the following forms.

Ways to specify Range	Description
`'Corner1:Corner2'` Rectangular Range	Specify the range using the syntax `'Corner1:Corner2'`, where *`Corner1`* and *`Corner2`* are two opposing corners that define the region. For example, `'D2:H4'` represents the 3-by-5 rectangular region between the two corners `D2` and `H4` on the worksheet. The `'Range'` name-value pair argument is not case-sensitive, and uses Excel A1 reference style (see Excel help). **Example:** `'Range','Corner1:Corner2'`
`''` Unspecified or Empty	If unspecified, `readtable` automatically detects the used range. **Example:** `'Range',''` **Note:** *Used Range* refers to the rectangular portion of the spreadsheet that actually contains data. `readtable` automatically detects the used range by trimming any leading and trailing rows and columns that do not contain data. Text that is only white space is considered data and is captured within the used range.
`'Row1:Row2'` Row Range	You can identify range by specifying the beginning and ending rows using Excel row designators. Then `readtable` automatically detects the used column range within the designated rows. For instance, `readtable` interprets the range specification `'1:7'` as an instruction to read all columns in the used range in rows 1 through 7 (inclusive). **Example:** `'Range','1:7'`

Ways to specify Range	Description
'*Column1:Column2*' Column Range	You can identify range by specifying the beginning and ending columns using Excel column designators. Then `readtable` automatically detects the used row range within the designated columns. For instance, `readtable` interprets the range specification `'A:F'` as an instruction to read all rows in the used range in columns A through F (inclusive). **Example:** `'Range','A:F'`
'*NamedRange*' Excel's Named Range	In Excel, you can create names to identify ranges in the spreadsheet. For instance, you can select a rectangular portion of the spreadsheet and call it `'myTable'`. If such named ranges exist in a spreadsheet, then `readtable` can read that range using its name. **Example:** `'Range','myTable'`

Example: `'Range'`, `'A1:F10'`

Example: `'Range'`, `"A1:F10"`

Data Types: `char` | `string`

`UseExcel` — Flag to start instance of Microsoft Excel for Windows
`false` (default) | `true`

Flag to start an instance of Microsoft Excel for Windows when reading spreadsheet data, specified as the comma-separated pair consisting of `'UseExcel'` and either `true`, or `false`.

You can set the `'UseExcel'` parameter to one of these values:

- `true` — The importing function starts an instance of Microsoft Excel when reading the file.
- `false` — The importing function does not start an instance of Microsoft Excel when reading the file. When operating in this mode, the importing function functionality differs in the support of file formats and interactive features, such as formulas and macros.

UseExcel	true	false
Supported file formats	.xls, .xlsx, .xlsm, .xltx, .xltm, .xlsb, .ods	.xls, .xlsx, .xlsm, .xltx, .xltm
Support for interactive features, such as formulas and macros	Yes	No

When reading from spreadsheet files on Windows platforms, if you want to start an instance of Microsoft Excel, then set the `'UseExcel'` parameter to `true`.

Output Arguments

T — Output table
table

Output table, returned as a table. The table can store metadata such as descriptions, variable units, variable names, and row names. For more information, see the Properties section of `table`.

Tips

- Large files in XLSX format sometimes load slowly. For better import and export performance, Microsoft recommends that you use the XLSB format.

See Also

Import Tool | `detectImportOptions` | `preview` | `readcell` | `readmatrix` | `readtimetable` | `readvars` | `table` | `textscan` | `writetable`

Topics

"Access Data in Tables"
"Import Text Files"
"Import Spreadsheets"

Introduced in R2013b

readtimetable

Create timetable from file

Syntax

```
TT = readtimetable(filename)
TT = readtimetable(filename,opts)
TT = readtimetable( ___ ,Name,Value)
```

Description

`TT = readtimetable(filename)` creates a timetable by reading column-oriented data from a file.

`readtimetable` determines the file format from the file extension:

- `.txt`, `.dat`, or `.csv` for delimited text files
- `.xls`, `.xlsb`, `.xlsm`, `.xlsx`, `.xltm`, `.xltx`, or `.ods` for spreadsheet files

`readtimetable` creates one variable in `TT` for each column in the file and reads variable names from the first row of the file.

`readtimetable` sets the first column of type `datetime` or `duration` in the tabular data to be the row times of the timetable. The remaining columns become variables of the timetable.

`TT = readtimetable(filename,opts)` additionally uses the import options `opts`.

`TT = readtimetable( ___ ,Name,Value)` creates a timetable from a file with additional options specified by one or more name-value pair arguments. Use any of the input arguments from the previous syntaxes before specifying the name-value pairs.

To set specific import options for your data, you can either use the `opts` object or you can specify name-value pairs. When you specify name-value pairs in addition to `opts`, then `readtimetable` supports only these name-value pairs:

- Text and spreadsheet parameters — `ReadVariableNames`, `RowTimes`, `SampleRate`, `TimeStep`, `StartTime`
- Text only parameters — `DateLocale`, `Encoding`
- Spreadsheet only parameters — `Sheet`, `UseExcel`

Examples

Create Timetable from Text File

Create a table from the comma-separated text file.

```
TT = readtimetable('outages.csv');
```

Display a summary of the table. When creating a timetable, if you do not specify any parameters for row times, then the `readtimetable` function detects and designates the first datetime or duration variable in the data, `OutageTime`, as the row times variable. The remaining variables become the variables of the timetable.

```
summary(TT)
```

```
RowTimes:

    OutageTime: 1468x1 datetime
        Values:
            Min         01-Feb-2002 12:18:00
            Median      18-Mar-2010 21:05:30
            Max         15-Jan-2014 02:41:00

Variables:

    Region: 1468x1 cell array of character vectors

    Loss: 1468x1 double

        Values:

            Min             0
            Median          180.26
            Max             23418
            NumMissing      604
```

```
    Customers: 1468x1 double

        Values:

            Min             0
            Median          75765
            Max             5.9689e+06
            NumMissing      328

    RestorationTime: 1468x1 datetime

        Values:

            Min             07-Feb-2002 16:50:00
            Median          31-Mar-2010 10:54:00
            Max             18-Sep-2042 23:31:00
            NumMissing      29

    Cause: 1468x1 cell array of character vectors
```

Create Timetable from File using Import Options

Detect import options for a text file, specify the variable types, and then create a timetable from the data.

Create an import options object from a file and examine the variable options.

```
opts = detectImportOptions('outages.csv');
opts.VariableOptions
```

```
ans =
  1x6 heterogeneous VariableImportOptions (TextVariableImportOptions, DatetimeVariable

    Name
    Type
    FillValue
    TreatAsMissing
    QuoteRule
    Prefixes
    Suffixes
    EmptyFieldRule
```

Modify the options object to specify the desired datatypes for the variables in the data. Change the datatypes for the variables `Region` and `Cause` to categorical.

```
opts = setvartype(opts,{'Region','Cause'},{'categorical','categorical'});
```

Use `readtimetable` along with the options object to import the timetable. Then display a summary of the timetable.

```
TT = readtimetable('outages.csv',opts);
summary(TT)
```

```
RowTimes:

    OutageTime: 1468x1 datetime
        Values:
            Min         01-Feb-2002 12:18:00
            Median      18-Mar-2010 21:05:30
            Max         15-Jan-2014 02:41:00

Variables:

    Region: 1468x1 categorical

        Values:

            MidWest        142
            NorthEast      557
            SouthEast      389
            SouthWest       26
            West           354

    Loss: 1468x1 double

        Values:

            Min             0
            Median          180.26
            Max             23418
            NumMissing      604

    Customers: 1468x1 double

        Values:

            Min             0
```

```
            Median          75765
            Max             5.9689e+06
            NumMissing      328

    RestorationTime: 1468x1 datetime

        Values:

            Min             07-Feb-2002 16:50:00
            Median          31-Mar-2010 10:54:00
            Max             18-Sep-2042 23:31:00
            NumMissing      29

    Cause: 1468x1 categorical

        Values:

            attack                294
            earthquake              2
            energy emergency      188
            equipment fault       156
            fire                   25
            severe storm          338
            thunder storm         201
            unknown                24
            wind                   95
            winter storm          145
```

Create Timetable from File and Specify Row Times

Read a table from the comma-separated text file and create a timetable with a row times variable of your choice.

Create an import options object and preview the tabular data.

```
opts = detectImportOptions('outages.csv');
preview('outages.csv',opts)
```

```
ans=8×6 table
       Region             OutageTime          Loss     Customers       RestorationTime
    _____________    ____________________    ______    __________    ___________________
```

```
    {'SouthWest'}    01-Feb-2002 12:18:00    458.98    1.8202e+06    07-Feb-2002 16:50
    {'SouthEast'}    23-Jan-2003 00:49:00    530.14    2.1204e+05                      N
    {'SouthEast'}    07-Feb-2003 21:15:00     289.4    1.4294e+05    17-Feb-2003 08:14
    {'West'     }    06-Apr-2004 05:44:00    434.81    3.4037e+05    06-Apr-2004 06:10
    {'MidWest'  }    16-Mar-2002 06:18:00    186.44    2.1275e+05    18-Mar-2002 23:23
    {'West'     }    18-Jun-2003 02:49:00         0             0    18-Jun-2003 10:54
    {'West'     }    20-Jun-2004 14:39:00    231.29           NaN    20-Jun-2004 19:16
    {'West'     }    06-Jun-2002 19:28:00    311.86           NaN    07-Jun-2002 00:51
```

Create a timetable by specifying the `RestorationTime` variable to be the row times variable for the timetable. Then, display a summary of the timetable.

```
TT = readtimetable('outages.csv','RowTimes','RestorationTime');
summary(TT)
```

```
RowTimes:

    RestorationTime: 1468x1 datetime
        Values:
            Min             07-Feb-2002 16:50:00
            Median          31-Mar-2010 10:54:00
            Max             18-Sep-2042 23:31:00
            NumMissing      29

Variables:

    Region: 1468x1 cell array of character vectors

    OutageTime: 1468x1 datetime

        Values:

            Min       01-Feb-2002 12:18:00
            Median    18-Mar-2010 21:05:30
            Max       15-Jan-2014 02:41:00

    Loss: 1468x1 double

        Values:

            Min             0
            Median          180.26
            Max             23418
            NumMissing      604
```

```
    Customers: 1468x1 double

        Values:

            Min             0
            Median          75765
            Max             5.9689e+06
            NumMissing      328

    Cause: 1468x1 cell array of character vectors
```

Input Arguments

`filename` — Name of file to read
character vector | string scalar

Name of the file to read, specified as a character vector or a string scalar.

Depending on the location of your file, `filename` can take on one of these forms.

Location	Form
Current folder or folder on the MATLAB path	Specify the name of the file in `filename`. **Example:** `'myFile.txt'`
File in a folder	If the file is not in the current folder or in a folder on the MATLAB path, then specify the full or relative path name in `filename`. **Example:** `'C:\myFolder\myFile.xlsx'` **Example:** `'dataDir\myFile.txt'`

Location	Form
Remote Location	If the file is stored at a remote location, then `filename` must contain the full path of the file specified as a uniform resource locator (URL) of the form: *`scheme_name`*`://path_to_file/my_file.ext` Based on your remote location, *`scheme_name`* can be one of the values in this table. **Remote Location** — **`scheme_name`** Amazon S3 — `s3` Windows Azure Blob Storage — `wasb`, `wasbs` HDFS — `hdfs` For more information, see “Work with Remote Data”. **Example:** `'s3://bucketname/path_to_file/my_file.csv'`

- If `filename` includes the file extension, then the importing function determines the file format from the extension. Otherwise, you must specify the `'FileType'` name-value pair arguments to indicate the type of file.
- On Windows systems with Microsoft Excel software, the importing function reads any Excel spreadsheet file format recognized by your version of Excel.
- If your system does not have Excel for Windows or if you are using MATLAB Online, the importing function operates with the `UseExcel` property set to `false`, and reads only `.xls, .xlsx, .xlsm, .xltx, and .xltm` files.
- For delimited text files, the importing function converts empty fields in the file to either `NaN` (for a numeric variable) or an empty character vector (for a text variable). All lines in the text file must have the same number of delimiters. The importing function ignores insignificant white space in the file.

Data Types: `char` | `string`

`opts` — File import options
`SpreadsheetImportOptions` | `DelimitedtextImportOptions` | `FixedWidthImportOptions`

File import options, specified as an `SpreadsheetImportOptions`, `DelimitedTextImportOptions`, or `FixedWidthImportOptions` object created by the `detectImportOptions` function. The `opts` object contains properties that control the data import process. For more information on the properties of each object, see the appropriate object page.

Type of Files	Output
Spreadsheet files	`SpreadsheetImportOptions` object
Text files	`DelimitedTextImportOptions` object
Fixed-width text files	`FixedWidthImportOptions` object

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'NumHeaderLines',5` indicates that the first five lines that precede the tabular data are header lines.

Text and Spreadsheet Files

`FileType` — Type of file
`'text'` | `'spreadsheet'`

Type of file, specified as the comma-separated pair consisting of `'FileType'` and `'text'` or `'spreadsheet'`.

Specify the `'FileType'` name-value pair argument when the `filename` does not include the file extension or if the extension is other than one of the following:

- `.txt`, `.dat`, or `.csv` for delimited text files
- `.xls`, `.xlsb`, `.xlsm`, `.xlsx`, `.xltm`, `.xltx`, or `.ods` for spreadsheet files

Example: `'FileType','text'`

Data Types: `char` | `string`

`NumHeaderLines` — Number of header lines
positive integer

Number of header lines in the file, specified as the comma-separated pair consisting of `'NumHeaderLines'` and a positive integer. If unspecified, the importing function automatically detects the number of header lines in the file.

Example: `'NumHeaderLines',7`

Data Types: `single` | `double`

`ExpectedNumVariables` — Expected number of variables
positive integer

Expected number of variables, specified as the comma-separated pair consisting of `'ExpectedNumVariables'` and a positive integer. If unspecified, the importing function automatically detects the number of variables.

Data Types: `single` | `double`

Range — Portion of data to read
character vector | string scalar | numeric vector

Portion of the data to read from text or spreadsheet files, specified as the comma separated pair consisting of `'Range'` and a character vector, string scalar, or numeric vector in one of these forms.

Ways to specify Range	Description
Starting Cell `'Cell'` or `[row col]`	Specify the starting cell for the data as a character vector or string scalar or a two element numeric vector. • Character vector or string scalar containing a column letter and row number using Excel `A1` notation. For example, `A5` is the identifier for the cell at the intersection of column `A` and row `5`. • Two element numeric vector of the form `[row col]` indicating the starting row and column. Using the starting cell, the importing function automatically detects the extent of the data by beginning the import at the start cell and ending at the last empty row or footer range. **Example:** `'A5'` or `[5 1]`

Ways to specify Range	Description
Rectangular Range `'Corner1:Corner2'` or `[r1 c1 r2 c2]`	Specify the exact range to read using the rectangular range in one of these forms. • `'Corner1:Corner2'` — Specify the range using `Corner1` and `Corner2` which are the two opposing corners that define the region to read in Excel `A1` notation. For example, `'C2:N15'`. • `[r1 c1 r2 c2]` — Specify the range using a four element numeric vector containing start-row, start-column, end-row, and end-column. For example, `[2 3 15 13]`. The importing function only reads the data contained in the specified range. Any empty fields within the specified range are imported as missing cells.
Row Range or Column Range `'Row1:Row2'` or `'Column1:Column2'`	Specify the range by identifying the beginning and ending rows using Excel row numbers. Using the specified row range, the importing function automatically detects the column extent by reading from the first nonempty column to the end of the data, and creates one variable per column. **Example:** `'5:500'` Alternatively, specify the range by identifying the beginning and ending columns using Excel column letters or numbers. Using the specified column range, the import function automatically detects the row extent by reading from the first nonempty row to the end of the data or the footer range. The number of columns in the specified range must match the number specified in the `ExpectedNumVariables` property. **Example:** `'A:K'`

Ways to specify Range	Description
Starting Row Number `n`	Specify the first row containing the data using the positive scalar row index. Using the specified row index, the importing function automatically detects the extent of the data by reading from the specified first row to the end of the data or the footer range. **Example:**5
Excel's Named Range `'NamedRange'`	In Excel, you can create names to identify ranges in the spreadsheet. For instance, you can select a rectangular portion of the spreadsheet and call it `'myTable'`. If such named ranges exist in a spreadsheet, then the importing function can read that range using its name. **Example:** `'Range','myTable'`
Unspecified or Empty `''`	If unspecified, the importing function automatically detects the used range. **Example:** `'Range',''` **Note:** *Used Range* refers to the rectangular portion of the spreadsheet that actually contains data. The importing function automatically detects the used range by trimming any leading and trailing rows and columns that do not contain data. Text that is only white space is considered data and is captured within the used range.

Data Types: `char` | `string` | `double`

`TextType` — Type for imported text data
`'char'` (default) | `'string'`

Type for imported text data, specified as the comma-separated pair consisting of `'TextType'` and either `'char'` or `'string'`.

- `'char'` — Import text data into MATLAB as character vectors.
- `'string'` — Import text data into MATLAB as string arrays.

Example: `'TextType','char'`

`DatetimeType` — Type for imported date and time data
`'datetime'` (default) | `'text'` | `'exceldatenum'` (spreadsheet files only)

Type for imported date and time data, specified as the comma-separated pair consisting of `'DatetimeType'` and one of these values: `'datetime'`, `'text'`, or `'exceldatenum'`. The value `'exceldatenum'` is applicable only for spreadsheet files, and is not valid for text files.

Value	Type for Imported Date and Time Data
`'datetime'`	MATLAB `datetime` data type For more information, see `datetime`.
`'text'`	If `'DatetimeType'` is specified as `'text'`, then the type for imported date and time data depends on the value specified in the `'TextType'` parameter: • If `'TextType'` is set to `'char'`, then the importing function returns dates as a cell array of character vectors. • If `'TextType'` is set to `'string'`, then the importing function returns dates as an array of strings.
`'exceldatenum'`	Excel serial date numbers A serial date number is a single number equal to the number of days from a given reference date. Excel serial date numbers use a different reference date than MATLAB serial date numbers. For more information on Excel dates, see `https://support.microsoft.com/en-us/kb/214330`.

Data Types: `char` | `string`

`TreatAsMissing` — Text to interpret as missing data
character vector | string scalar | cell array of character vectors | string array

Text to interpret as missing data, specified as a character vector, string scalar, cell array of character vectors, or string array.

Example: `'TreatAsMissing',{'NA','TBD'}` instructs the importing function to treat any occurrence of `NA` or `TBD` as a missing fields.

Data Types: `char` | `string` | `cell`

`ReadVariableNames` — Read first row as variable names
`true` | `false`

Read the first row as variable names, specified as the comma-separated pair consisting of `'ReadVariableNames'` and either `true` or `false`. If unspecified, the importing function automatically detects the presence of variable names.

Indicator	Description
`true`	Use when the first row of the region to read contains the variable names for the table. The importing function creates a variable, with the detected variable name, for each column in `T`.
`false`	Use when the first row of the region to read contains data in the table. The importing function creates default variable names of the form `'Var1',...,'VarN'`, where `N` is the number of variables.
Unspecified	When left unspecified, the importing function automatically detects `true` or `false` and proceeds accordingly.

When you specify `ReadVariableNames` name-value pair in addition to `opts`, then the importing function proceeds as follows.

- If `ReadVariableNames` is set to `true`, then the importing function reads the variable names from the specified file by using the `VariableNamesRange` or the `VariableNamesLine` property of the import options object.
- If `ReadVariableNames` is set to `false`, then the importing function reads the variable names from the `VariableNames` property of the import options object.

Data Types: `logical`

`RowTimes` — Row times variable
variable name | time vector

Row times variable, specified as the comma-separated pair consisting of `'RowTimes'` and a variable name or a time vector.

- Variable name must be a character vector or string scalar containing the name of any variable in the input table that contains `datetime` or `duration` values. The variable

specified by the variable name provides row time labels for the rows. The remaining variables of the input table become the variables of the timetable.

- Time vector must be a `datetime` vector or a `duration` vector. The number of elements of time vector must equal the number of rows of the input table. The time values in the time vector do not need to be unique, sorted, or regular. All the variables of the input table become variables of the timetable.

Data Types: `char` | `string` | `datetime` | `duration`

`SampleRate` — Sample rate for row times
positive numeric scalar

Sample rate for row times, specified as the comma-separated pair consisting of `'SampleRate'` and a positive numeric scalar. The sample rate is the number of samples per second (Hz) of the time vector of the output timetable.

When you use `'SampleRate'` to specify the row time vector of the timetable, the default first row time (start time) is zero second. To set a start time other than zero, specify the`'StartTime'` name-value pair.

Data Types: `double`

`TimeStep` — Time step between row times
duration scalar | calendarDuration scalar

Time step between row times, specified as the comma-separated pair consisting of `'TimeStep'` and a duration scalar or calendarDuration scalar. The value of the `'TimeStep'` parameter specifies the length of time between consecutive row times. The importing function uses the time step value to calculate regularly spaced row times.

When you use `'TimeStep'` to specify the row time vector of the timetable, the default first row time (start time) is zero second. To set a start time other than zero, specify the `'StartTime'` name-value pair.

If the `'TimeStep'` is a calendar duration value, then the `'StartTime'` must be a datetime value.

Data Types: `duration` | `calendarDuration`

`StartTime` — Start time of row times
datetime scalar | duration scalar

Start time of the row times, specified as the comma-separated pair consisting of `StartTime` and a datetime scalar or duration scalar.

To define the time vector for the timetable, use `'StartTime'` with either the `'SampleRate'` or the `'TimeStep'` name-value pair arguments.

The data type of the start time, dictates the data type of the row time vector.

- If the start time is a datetime value, then the row times of the timetable are datetime values.
- If the start time is a duration value, then the row times are durations.

Data Types: `datetime` | `duration`

`PreserveVariableNames` — Flag to preserve variable names
`false` (default) | `true`

Flag to preserve variable names, specified as the comma-separated pair consisting of `PreserveVariableNames` and either `true`, or `false`.

- `true` — Preserve variable names that are not valid MATLAB identifiers such as variable names that include spaces and non-ASCII characters.
- `false` — Convert invalid variable names (as determined by the `isvarname` function) to valid MATLAB identifiers.

Starting in R2019b, variable names and row names can include any characters, including spaces and non-ASCII characters. Also, they can start with any characters, not just letters. Variable and row names do not have to be valid MATLAB identifiers (as determined by the `isvarname` function). To preserve these variable names and row names, set `PreserveVariableNames` to `true`.

Text Files Only

`Delimiter` — Field delimiter characters
character vector | string scalar | cell array of character vectors | string array

Field delimiter characters in a delimited text file, specified as a character vector, string scalar, cell array of character vectors, or string array.

Example: `'Delimiter','|'`

Example: `'Delimiter',{';','*'}`

Data Types: `char` | `string` | `cell`

`Whitespace` — Characters to treat as white space
character vector | string scalar

Characters to treat as white space, specified as a character vector or string scalar containing one or more characters.

Example: `'Whitespace',' _'`

Example: `'Whitespace','?!.,'`

`LineEnding` — End-of-line characters
`{'\n','\r','\r\n'}` (default) | character vector | string scalar | cell array of character vectors | string array

End-of-line characters, specified as a character vector, string scalar, cell array of character vectors, or string array.

Example: `'LineEnding','\n'`

Example: `'LineEnding','\r\n'`

Example: `'LineEnding',{'\b',':'}`

Data Types: `char` | `string` | `cell`

`CommentStyle` — Style of comments
character vector | string scalar | cell array of character vectors | string array

Style of comments, specified as a character vector, string scalar, cell array of character vectors, or string array.

For example, to ignore the text following a percent sign on the same line, specify `CommentStyle` as `'%'`.

Example: `'CommentStyle',{'/*'}`

Data Types: `char` | `string` | `cell`

`Encoding` — Character encoding scheme
`'system'` | `'UTF-8'` | `'ISO-8859-1'` | `'windows-1251'` | `'windows-1252'` | ...

Character encoding scheme associated with the file, specified as the comma-separated pair consisting of `'Encoding'` and `'system'` or a standard character encoding scheme name, such as one of the values in this table.

'Big5'	'ISO-8859-1'	'windows-847'
'Big5-HKSCS'	'ISO-8859-2'	'windows-949'
'CP949'	'ISO-8859-3'	'windows-1250'
'EUC-KR'	'ISO-8859-4'	'windows-1251'
'EUC-JP'	'ISO-8859-5'	'windows-1252'
'EUC-TW'	'ISO-8859-6'	'windows-1253'
'GB18030'	'ISO-8859-7'	'windows-1254'
'GB2312'	'ISO-8859-8'	'windows-1255'
'GBK'	'ISO-8859-9'	'windows-1256'
'IBM866'	'ISO-8859-11'	'windows-1257'
'KOI8-R'	'ISO-8859-13'	'windows-1258'
'KOI8-U'	'ISO-8859-15'	'US-ASCII'
	'Macintosh'	'UTF-8'
	'Shift_JIS'	

Example: `'Encoding','system'` uses the system default encoding.

Data Types: `char` | `string`

`DurationType` — Output data type of duration data
`'duration'` (default) | `'text'`

Output data type of duration data from text files, specified as the comma-separated pair consisting of `'DurationType'` and either `'duration'` or `'text'`.

Value	Type for Imported Duration Data
`'duration'`	MATLAB `duration` data type For more information, see `duration`.

Value	Type for Imported Duration Data
`'text'`	If `'DurationType'` is specified as `'text'`, then the type for imported duration data depends on the value specified in the `'TextType'` parameter: • If `'TextType'` is set to `'char'`, then the importing function returns duration data as a cell array of character vectors. • If `'TextType'` is set to `'string'`, then the importing function returns duration data as an array of strings.

Data Types: `char` | `string`

`DateLocale` — Locale for reading dates
character vector | string scalar

Locale for reading dates, specified as the comma-separated pair consisting of `'DateLocale'` and a character vector or a string scalar of the form *xx_YY*, where:

- *YY* is an uppercase ISO 3166-1 alpha-2 code indicating a country.
- *xx* is a lowercase ISO 639-1 two-letter code indicating a language.

For a list of common values for the locale, see the `Locale` name-value pair argument for the `datetime` function.

When using the `%D` format specifier to read text as `datetime` values, use `DateLocale` to specify the locale in which the importing function should interpret month and day-of-week names and abbreviations.

If you specify the `DateLocale` argument in addition to `opts` the import options, then the importing function uses the specified value for the `DateLocale` argument, overriding the locale defined in the import options.

Example: `'DateLocale','ja_JP'`

`DecimalSeparator` — Characters indicating decimal separator
character vector | string scalar

Characters indicating the decimal separator in numeric variables, specified as a character vector or string scalar. The importing function uses the characters specified in the `DecimalSeparator` name-value pair to distinguish the integer part of a number from the decimal part.

When converting to integer data types, numbers with a decimal part are rounded to the nearest integer.

Example: If name-value pair is specified as `'DecimalSeparator',','`, then the importing function imports the text `"3,14159"` as the number `3.14159`.

Data Types: `char` | `string`

`ThousandsSeparator` — Characters that indicate thousands grouping
character vector | string scalar

Characters that indicate the thousands grouping in numeric variables, specified as a character vector or string scalar. The thousands grouping characters act as visual separators, grouping the number at every three place values. The importing function uses the characters specified in the `ThousandsSeparator` name-value pair to interpret the numbers being imported.

Example: If name-value pair is specified as `'ThousandsSeparator',','`, then the importing function imports the text `"1,234,000"` as `1234000`.

Data Types: `char` | `string`

`TrimNonNumeric` — Remove nonnumeric characters
`false` (default) | `true`

Remove nonnumeric characters from a numeric variable, specified as a logical `true` or `false`.

Example: If name-value pair is specified as `'TrimNonNumeric',true`, then the importing function reads `'$500/-'` as `500`.

Data Types: `logical`

`ConsecutiveDelimitersRule` — Procedure to handle consecutive delimiters
`'split'` | `'join'` | `'error'`

Procedure to handle consecutive delimiters in a delimited text file, specified as one of the values in this table.

Consecutive Delimiters Rule	Behavior
`'split'`	Split the consecutive delimiters into multiple fields.
`'join'`	Join the delimiters into one delimiter.

Consecutive Delimiters Rule	Behavior
`'error'`	Return an error and abort the import operation.

Data Types: `char` | `string`

`LeadingDelimitersRule` — Procedure to manage leading delimiters
`'keep'` | `'ignore'` | `'error'`

Procedure to manage leading delimiters in a delimited text file, specified as one of the values in this table.

Leading Delimiters Rule	Behavior
`'keep'`	Keep the delimiter.
`'ignore'`	Ignore the delimiter.
`'error'`	Return an error and abort the import operation.

Spreadsheet Files Only

`Sheet` — Sheet to read from
`''` empty character array (default) | character vector | string scalar | positive scalar integer

Sheet to read from, specified as an empty character array, a character vector or string scalar containing the sheet name, or a positive scalar integer denoting the sheet index. Based on the value specified for the `Sheet` property, the import function behaves as described in the table.

Specification	Behavior
`''` (default)	Import data from the first sheet.
Name	Import data from the matching sheet name, regardless of order of sheets in the spreadsheet file.
Integer	Import data from sheet in the position denoted by the integer, regardless of the sheet names in the spreadsheet file.

Data Types: `char` | `string` | `single` | `double`

UseExcel — Flag to start instance of Microsoft Excel for Windows
`false` (default) | `true`

Flag to start an instance of Microsoft Excel for Windows when reading spreadsheet data, specified as the comma-separated pair consisting of `'UseExcel'` and either `true`, or `false`.

You can set the `'UseExcel'` parameter to one of these values:

- `true` — The importing function starts an instance of Microsoft Excel when reading the file.
- `false` — The importing function does not start an instance of Microsoft Excel when reading the file. When operating in this mode, the importing function functionality differs in the support of file formats and interactive features, such as formulas and macros.

UseExcel	true	false
Supported file formats	`.xls, .xlsx, .xlsm, .xltx, .xltm, .xlsb, .ods`	`.xls, .xlsx, .xlsm, .xltx, .xltm`
Support for interactive features, such as formulas and macros	Yes	No

When reading from spreadsheet files on Windows platforms, if you want to start an instance of Microsoft Excel, then set the `'UseExcel'` parameter to `true`.

Output Arguments

TT — Output timetable
timetable

Output timetable. The timetable can store metadata such as descriptions, variable units, variable names, and row times. For more information, see the Properties sections of `timetable`.

See Also

`readcell` | `readmatrix` | `readtable` | `readvars` | `timetable` | `timetable2table` | `writetimetable`

Introduced in R2019a

readvars

Read variables from file

Syntax

```
[Var1,Var2,...,VarN] = readvars(filename)
[Var1,Var2,...,VarN] = readvars(filename,opts)
[Var1,Var2,...,VarN] = readvars( ___ ,Name,Value)
```

Description

`[Var1,Var2,...,VarN] = readvars(filename)` creates variables by reading column-oriented data from a file. If the file contains `N` columns of data, then `readvars` returns `N` variables `Var1,Var2,...,VarN`.

`readvars` determines the file format from the file extension:

- `.txt`, `.dat`, or `.csv` for delimited text files
- `.xls`, `.xlsb`, `.xlsm`, `.xlsx`, `.xltm`, `.xltx`, or `.ods` for spreadsheet files

`[Var1,Var2,...,VarN] = readvars(filename,opts)` additionally uses the import options `opts`.

`[Var1,Var2,...,VarN] = readvars( ___ ,Name,Value)` creates variables from a file with additional options specified by one or more name-value pair arguments. Use any of the input arguments from the previous syntaxes before specifying the name-value pairs.

To set specific import options for your data, you can either use the `opts` object or you can specify name-value pairs. If you specify name-value pairs in addition to `opts`, then `readvars` supports only these name-value pairs:

- Text Files — `DateLocale`, `Encoding`
- Spreadsheet Files — `Sheet`, `UseExcel`

Examples

Read Variables from Text File

Import columns from tabular data in a text file as separate variables. First, preview the contents of the text file `outages.csv` and then read columns.

Preview the data in `outages.csv`. The file has 6 variables.

```
opts = detectImportOptions('outages.csv');
preview('outages.csv',opts)
```

```
ans=8×6 table
       Region             OutageTime          Loss     Customers       RestorationTime
    _____________    ____________________    ______    __________    ___________________

    {'SouthWest'}    01-Feb-2002 12:18:00    458.98    1.8202e+06    07-Feb-2002 16:50
    {'SouthEast'}    23-Jan-2003 00:49:00    530.14    2.1204e+05                    N
    {'SouthEast'}    07-Feb-2003 21:15:00     289.4    1.4294e+05    17-Feb-2003 08:14
    {'West'     }    06-Apr-2004 05:44:00    434.81    3.4037e+05    06-Apr-2004 06:10
    {'MidWest'  }    16-Mar-2002 06:18:00    186.44    2.1275e+05    18-Mar-2002 23:23
    {'West'     }    18-Jun-2003 02:49:00         0             0    18-Jun-2003 10:54
    {'West'     }    20-Jun-2004 14:39:00    231.29           NaN    20-Jun-2004 19:16
    {'West'     }    06-Jun-2002 19:28:00    311.86           NaN    07-Jun-2002 00:51
```

Import the first three columns as separate variables in the workspace.

```
[Region,OutageTime,Loss] = readvars('outages.csv');
whos Region OutageTime Loss
```

```
  Name               Size             Bytes  Class       Attributes

  Loss            1468x1              11744  double
  OutageTime      1468x1              23489  datetime
  Region          1468x1             186732  cell
```

Read Variables from Spreadsheet File

Import columns from a spreadsheet file as separate variables in the workspace. First, preview the contents of the text file `outages.csv` and then read columns as separate variables.

Preview the data in `patients.xls`. The file has `10` variables.

```
opts = detectImportOptions('patients.xls');
preview('patients.xls',opts)
```

```
ans=8×10 table
      LastName        Gender      Age              Location                Height    We
    ____________    __________    ___    ______________________________    ______    __

    {'Smith'   }    {'Male'  }    38     {'County General Hospital'  }      71       1
    {'Johnson' }    {'Male'  }    43     {'VA Hospital'              }      69       1
    {'Williams'}    {'Female'}    38     {'St. Mary's Medical Center'}      64       1
    {'Jones'   }    {'Female'}    40     {'VA Hospital'              }      67       1
    {'Brown'   }    {'Female'}    49     {'County General Hospital'  }      64       1
    {'Davis'   }    {'Female'}    46     {'St. Mary's Medical Center'}      68       1
    {'Miller'  }    {'Female'}    33     {'VA Hospital'              }      64       1
    {'Wilson'  }    {'Male'  }    40     {'VA Hospital'              }      68       1
```

Import the first three columns as separate variables in the workspace.

```
[LastName,Gender,Age] = readvars('patients.xls');
whos  LastName Gender Age
```

```
  Name            Size            Bytes  Class     Attributes

  Age           100x1               800  double
  Gender        100x1             12212  cell
  LastName      100x1             12416  cell
```

Read Variables from Specified Sheet and Range using Import Options

Preview the data from a spreadsheet file and import columns of data from a specified sheet and range as separate variables.

The spreadsheet file `airlinesmall_subset.xlsx` contains data in multiple worksheets for years between 1996 and 2008. Each worksheet has data for a given year.

Preview the data from file `airlinesmall_subset.xlsx`. The `preview` function shows data from the first worksheet by default.

```
opts = detectImportOptions('airlinesmall_subset.xlsx');
preview('airlinesmall_subset.xlsx',opts)
```

```
ans=8×29 table
    Year    Month    DayofMonth    DayOfWeek    DepTime    CRSDepTime    ArrTime    CRS
    ____    _____    __________    _________    _______    __________    _______    ___

    1996      1          18            4         2117         2120        2305
    1996      1          12            5         1252         1245        1511
    1996      1          16            2         1441         1445        1708
    1996      1           1            1         2258         2300        2336
    1996      1           4            4         1814         1814        1901
    1996      1          31            3         1822         1820        1934
    1996      1          18            4          729          730         841
    1996      1          26            5         1704         1705        1829
```

Configure the values in the `opts` object to import `10` rows for the columns `5` and `6` of the worksheet named `'2007'`.

```
opts.Sheet = '2007';
opts.SelectedVariableNames = [5 6];
opts.DataRange = '2:11';
[DepTime,CRSDepTime] = readvars('airlinesmall_subset.xlsx',opts);
```

Display the variables.

```
[DepTime, CRSDepTime]
```

```
ans = 10×2

         711         710
         652         655
        1116        1120
         825         825
        1411        1400
        1935        1935
        2005        2005
        1525        1525
        1133        1105
         922         925
```

Read Variables from Specified Sheet and Range

Preview the data from a spreadsheet file and import columns as separate variables from a specified sheet and range.

The spreadsheet file `airlinesmall_subset.xlsx` contains data in multiple worksheets for years between 1996 and 2008. Each worksheet has data for a given year. Preview the data from file `airlinesmall_subset.xlsx`. The `preview` function shows data from the first worksheet by default.

```
opts = detectImportOptions('airlinesmall_subset.xlsx');
preview('airlinesmall_subset.xlsx',opts)
```

```
ans=8×29 table
    Year    Month    DayofMonth    DayOfWeek    DepTime    CRSDepTime    ArrTime    CRS
    ____    _____    __________    _________    _______    __________    _______    ___

    1996      1          18            4         2117         2120        2305
    1996      1          12            5         1252         1245        1511
    1996      1          16            2         1441         1445        1708
    1996      1           1            1         2258         2300        2336
    1996      1           4            4         1814         1814        1901
    1996      1          31            3         1822         1820        1934
    1996      1          18            4          729          730         841
    1996      1          26            5         1704         1705        1829
```

Import `10` rows of the first three variables from the worksheet named `'2007'`.

```
[Year, Month, DayOfMonth] = readvars('airlinesmall_subset.xlsx','Sheet','2007','Range',
whos Year Month DayOfMonth
```

```
  Name             Size            Bytes  Class     Attributes

  DayOfMonth      10x1                80  double
  Month           10x1                80  double
  Year            10x1                80  double
```

```
disp([Year Month DayOfMonth])
```

```
        2007           1           2
        2007           1           3
```

```
        2007           1           4
        2007           1           5
        2007           1           7
        2007           1           8
        2007           1           9
        2007           1          11
        2007           1          12
        2007           1          13
```

Input Arguments

`filename` — Name of file to read
character vector | string scalar

Name of the file to read, specified as a character vector or a string scalar.

Depending on the location of your file, `filename` can take on one of these forms.

Location	Form
Current folder or folder on the MATLAB path	Specify the name of the file in `filename`. **Example:** `'myFile.txt'`
File in a folder	If the file is not in the current folder or in a folder on the MATLAB path, then specify the full or relative path name in `filename`. **Example:** `'C:\myFolder\myFile.xlsx'` **Example:** `'dataDir\myFile.txt'`

Location	Form
Remote Location	If the file is stored at a remote location, then `filename` must contain the full path of the file specified as a uniform resource locator (URL) of the form: *`scheme_name`*`://path_to_file/my_file.ext` Based on your remote location, *`scheme_name`* can be one of the values in this table. **Remote Location** — **`scheme_name`** Amazon S3 — `s3` Windows Azure Blob Storage — `wasb`, `wasbs` HDFS — `hdfs` For more information, see “Work with Remote Data”. **Example:** `'s3://bucketname/path_to_file/my_file.csv'`

- If `filename` includes the file extension, then the importing function determines the file format from the extension. Otherwise, you must specify the `'FileType'` name-value pair arguments to indicate the type of file.
- On Windows systems with Microsoft Excel software, the importing function reads any Excel spreadsheet file format recognized by your version of Excel.
- If your system does not have Excel for Windows or if you are using MATLAB Online, the importing function operates with the `UseExcel` property set to `false`, and reads only `.xls, .xlsx, .xlsm, .xltx, and .xltm` files.
- For delimited text files, the importing function converts empty fields in the file to either `NaN` (for a numeric variable) or an empty character vector (for a text variable). All lines in the text file must have the same number of delimiters. The importing function ignores insignificant white space in the file.

Data Types: `char` | `string`

`opts` — File import options
`SpreadsheetImportOptions` | `DelimitedtextImportOptions` | `FixedWidthImportOptions`

File import options, specified as an `SpreadsheetImportOptions`, `DelimitedTextImportOptions`, or `FixedWidthImportOptions` object created by the `detectImportOptions` function. The `opts` object contains properties that control the data import process. For more information on the properties of each object, see the appropriate object page.

Type of Files	Output
Spreadsheet files	`SpreadsheetImportOptions` object
Text files	`DelimitedTextImportOptions` object
Fixed-width text files	`FixedWidthImportOptions` object

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'NumHeaderLines',5` indicates that the first five lines that precede the tabular data are header lines.

Text and Spreadsheet Files

`FileType` — Type of file
`'text'` | `'spreadsheet'`

Type of file, specified as the comma-separated pair consisting of `'FileType'` and `'text'` or `'spreadsheet'`.

Specify the `'FileType'` name-value pair argument when the `filename` does not include the file extension or if the extension is other than one of the following:

- `.txt`, `.dat`, or `.csv` for delimited text files
- `.xls`, `.xlsb`, `.xlsm`, `.xlsx`, `.xltm`, `.xltx`, or `.ods` for spreadsheet files

Example: `'FileType','text'`

Data Types: `char` | `string`

`NumHeaderLines` — Number of header lines
positive integer

Number of header lines in the file, specified as the comma-separated pair consisting of `'NumHeaderLines'` and a positive integer. If unspecified, the importing function automatically detects the number of header lines in the file.

Example: `'NumHeaderLines',7`

Data Types: `single` | `double`

`ExpectedNumVariables` — Expected number of variables
positive integer

Expected number of variables, specified as the comma-separated pair consisting of `'ExpectedNumVariables'` and a positive integer. If unspecified, the importing function automatically detects the number of variables.

Data Types: `single` | `double`

Range — Portion of data to read
character vector | string scalar | numeric vector

Portion of the data to read from text or spreadsheet files, specified as the comma separated pair consisting of `'Range'` and a character vector, string scalar, or numeric vector in one of these forms.

Ways to specify Range	Description
Starting Cell `'Cell'` or `[row col]`	Specify the starting cell for the data as a character vector or string scalar or a two element numeric vector. • Character vector or string scalar containing a column letter and row number using Excel `A1` notation. For example, `A5` is the identifier for the cell at the intersection of column `A` and row `5`. • Two element numeric vector of the form `[row col]` indicating the starting row and column. Using the starting cell, the importing function automatically detects the extent of the data by beginning the import at the start cell and ending at the last empty row or footer range. **Example:** `'A5'` or `[5 1]`

Ways to specify Range	Description
Rectangular Range `'Corner1:Corner2'` or `[r1 c1 r2 c2]`	Specify the exact range to read using the rectangular range in one of these forms. • `'Corner1:Corner2'` — Specify the range using `Corner1` and `Corner2` which are the two opposing corners that define the region to read in Excel `A1` notation. For example, `'C2:N15'`. • `[r1 c1 r2 c2]` — Specify the range using a four element numeric vector containing start-row, start-column, end-row, and end-column. For example, `[2 3 15 13]`. The importing function only reads the data contained in the specified range. Any empty fields within the specified range are imported as missing cells.
Row Range or Column Range `'Row1:Row2'` or `'Column1:Column2'`	Specify the range by identifying the beginning and ending rows using Excel row numbers. Using the specified row range, the importing function automatically detects the column extent by reading from the first nonempty column to the end of the data, and creates one variable per column. **Example:** `'5:500'` Alternatively, specify the range by identifying the beginning and ending columns using Excel column letters or numbers. Using the specified column range, the import function automatically detects the row extent by reading from the first nonempty row to the end of the data or the footer range. The number of columns in the specified range must match the number specified in the `ExpectedNumVariables` property. **Example:** `'A:K'`

Ways to specify Range	Description
Starting Row Number `n`	Specify the first row containing the data using the positive scalar row index. Using the specified row index, the importing function automatically detects the extent of the data by reading from the specified first row to the end of the data or the footer range. **Example:**5
Excel's Named Range `'NamedRange'`	In Excel, you can create names to identify ranges in the spreadsheet. For instance, you can select a rectangular portion of the spreadsheet and call it `'myTable'`. If such named ranges exist in a spreadsheet, then the importing function can read that range using its name. **Example:** `'Range','myTable'`
Unspecified or Empty `''`	If unspecified, the importing function automatically detects the used range. **Example:** `'Range',''` **Note:** *Used Range* refers to the rectangular portion of the spreadsheet that actually contains data. The importing function automatically detects the used range by trimming any leading and trailing rows and columns that do not contain data. Text that is only white space is considered data and is captured within the used range.

Data Types: `char` | `string` | `double`

`TextType` — Type for imported text data
`'char'` (default) | `'string'`

Type for imported text data, specified as the comma-separated pair consisting of `'TextType'` and either `'char'` or `'string'`.

- `'char'` — Import text data into MATLAB as character vectors.
- `'string'` — Import text data into MATLAB as string arrays.

Example: `'TextType','char'`

`DatetimeType` — Type for imported date and time data
`'datetime'` (default) | `'text'` | `'exceldatenum'` (spreadsheet files only)

Type for imported date and time data, specified as the comma-separated pair consisting of `'DatetimeType'` and one of these values: `'datetime'`, `'text'`, or `'exceldatenum'`. The value `'exceldatenum'` is applicable only for spreadsheet files, and is not valid for text files.

Value	Type for Imported Date and Time Data
`'datetime'`	MATLAB `datetime` data type For more information, see `datetime`.
`'text'`	If `'DatetimeType'` is specified as `'text'`, then the type for imported date and time data depends on the value specified in the `'TextType'` parameter: • If `'TextType'` is set to `'char'`, then the importing function returns dates as a cell array of character vectors. • If `'TextType'` is set to `'string'`, then the importing function returns dates as an array of strings.
`'exceldatenum'`	Excel serial date numbers A serial date number is a single number equal to the number of days from a given reference date. Excel serial date numbers use a different reference date than MATLAB serial date numbers. For more information on Excel dates, see `https://support.microsoft.com/en-us/kb/214330`.

Data Types: `char` | `string`

`TreatAsMissing` — Text to interpret as missing data
character vector | string scalar | cell array of character vectors | string array

Text to interpret as missing data, specified as a character vector, string scalar, cell array of character vectors, or string array.

Example: `'TreatAsMissing',{'NA','TBD'}` instructs the importing function to treat any occurrence of `NA` or `TBD` as a missing fields.

Data Types: `char` | `string` | `cell`

Text Files Only

`Delimiter` — Field delimiter characters
character vector | string scalar | cell array of character vectors | string array

Field delimiter characters in a delimited text file, specified as a character vector, string scalar, cell array of character vectors, or string array.

Example: `'Delimiter','|'`

Example: `'Delimiter',{';','*'}`

Data Types: `char` | `string` | `cell`

`Whitespace` — Characters to treat as white space
character vector | string scalar

Characters to treat as white space, specified as a character vector or string scalar containing one or more characters.

Example: `'Whitespace',' _'`

Example: `'Whitespace','?!.,'`

`LineEnding` — End-of-line characters
`{'\n','\r','\r\n'}` (default) | character vector | string scalar | cell array of character vectors | string array

End-of-line characters, specified as a character vector, string scalar, cell array of character vectors, or string array.

Example: `'LineEnding','\n'`

Example: `'LineEnding','\r\n'`

Example: `'LineEnding',{'\b',':'}`

Data Types: `char` | `string` | `cell`

`CommentStyle` — Style of comments
character vector | string scalar | cell array of character vectors | string array

Style of comments, specified as a character vector, string scalar, cell array of character vectors, or string array.

For example, to ignore the text following a percent sign on the same line, specify `CommentStyle` as `'%'`.

Example: `'CommentStyle',{'/*'}`

Data Types: `char` | `string` | `cell`

`Encoding` — Character encoding scheme
`'system'` | `'UTF-8'` | `'ISO-8859-1'` | `'windows-1251'` | `'windows-1252'` | ...

Character encoding scheme associated with the file, specified as the comma-separated pair consisting of `'Encoding'` and `'system'` or a standard character encoding scheme name, such as one of the values in this table.

`'Big5'`	`'ISO-8859-1'`	`'windows-847'`
`'Big5-HKSCS'`	`'ISO-8859-2'`	`'windows-949'`
`'CP949'`	`'ISO-8859-3'`	`'windows-1250'`
`'EUC-KR'`	`'ISO-8859-4'`	`'windows-1251'`
`'EUC-JP'`	`'ISO-8859-5'`	`'windows-1252'`
`'EUC-TW'`	`'ISO-8859-6'`	`'windows-1253'`
`'GB18030'`	`'ISO-8859-7'`	`'windows-1254'`
`'GB2312'`	`'ISO-8859-8'`	`'windows-1255'`
`'GBK'`	`'ISO-8859-9'`	`'windows-1256'`
`'IBM866'`	`'ISO-8859-11'`	`'windows-1257'`
`'KOI8-R'`	`'ISO-8859-13'`	`'windows-1258'`
`'KOI8-U'`	`'ISO-8859-15'`	`'US-ASCII'`
	`'Macintosh'`	`'UTF-8'`
	`'Shift_JIS'`	

Example: `'Encoding','system'` uses the system default encoding.

Data Types: `char` | `string`

`DurationType` — Output data type of duration data
`'duration'` (default) | `'text'`

Output data type of duration data from text files, specified as the comma-separated pair consisting of `'DurationType'` and either `'duration'` or `'text'`.

Value	Type for Imported Duration Data
`'duration'`	MATLAB `duration` data type For more information, see `duration`.
`'text'`	If `'DurationType'` is specified as `'text'`, then the type for imported duration data depends on the value specified in the `'TextType'` parameter: • If `'TextType'` is set to `'char'`, then the importing function returns duration data as a cell array of character vectors. • If `'TextType'` is set to `'string'`, then the importing function returns duration data as an array of strings.

Data Types: `char` | `string`

`DateLocale` — Locale for reading dates
character vector | string scalar

Locale for reading dates, specified as the comma-separated pair consisting of `'DateLocale'` and a character vector or a string scalar of the form *xx_YY*, where:

- *YY* is an uppercase ISO 3166-1 alpha-2 code indicating a country.
- *xx* is a lowercase ISO 639-1 two-letter code indicating a language.

For a list of common values for the locale, see the `Locale` name-value pair argument for the `datetime` function.

When using the `%D` format specifier to read text as `datetime` values, use `DateLocale` to specify the locale in which the importing function should interpret month and day-of-week names and abbreviations.

If you specify the `DateLocale` argument in addition to `opts` the import options, then the importing function uses the specified value for the `DateLocale` argument, overriding the locale defined in the import options.

Example: `'DateLocale','ja_JP'`

`DecimalSeparator` — Characters indicating decimal separator
character vector | string scalar

Characters indicating the decimal separator in numeric variables, specified as a character vector or string scalar. The importing function uses the characters specified in the `DecimalSeparator` name-value pair to distinguish the integer part of a number from the decimal part.

When converting to integer data types, numbers with a decimal part are rounded to the nearest integer.

Example: If name-value pair is specified as `'DecimalSeparator',','`, then the importing function imports the text `"3,14159"` as the number `3.14159`.

Data Types: `char` | `string`

`ThousandsSeparator` — Characters that indicate thousands grouping
character vector | string scalar

Characters that indicate the thousands grouping in numeric variables, specified as a character vector or string scalar. The thousands grouping characters act as visual separators, grouping the number at every three place values. The importing function uses the characters specified in the `ThousandsSeparator` name-value pair to interpret the numbers being imported.

Example: If name-value pair is specified as `'ThousandsSeparator',','`, then the importing function imports the text `"1,234,000"` as `1234000`.

Data Types: `char` | `string`

`TrimNonNumeric` — Remove nonnumeric characters
`false` (default) | `true`

Remove nonnumeric characters from a numeric variable, specified as a logical `true` or `false`.

Example: If name-value pair is specified as `'TrimNonNumeric',true`, then the importing function reads `'$500/-'` as `500`.

Data Types: `logical`

`ConsecutiveDelimitersRule` — Procedure to handle consecutive delimiters
`'split'` | `'join'` | `'error'`

Procedure to handle consecutive delimiters in a delimited text file, specified as one of the values in this table.

Consecutive Delimiters Rule	Behavior
`'split'`	Split the consecutive delimiters into multiple fields.
`'join'`	Join the delimiters into one delimiter.
`'error'`	Return an error and abort the import operation.

Data Types: `char` | `string`

`LeadingDelimitersRule` — Procedure to manage leading delimiters
`'keep'` | `'ignore'` | `'error'`

Procedure to manage leading delimiters in a delimited text file, specified as one of the values in this table.

Leading Delimiters Rule	Behavior
`'keep'`	Keep the delimiter.
`'ignore'`	Ignore the delimiter.
`'error'`	Return an error and abort the import operation.

Spreadsheet Files Only

`Sheet` — Sheet to read from
`''` empty character array (default) | character vector | string scalar | positive scalar integer

Sheet to read from, specified as an empty character array, a character vector or string scalar containing the sheet name, or a positive scalar integer denoting the sheet index. Based on the value specified for the `Sheet` property, the import function behaves as described in the table.

Specification	Behavior
`''` (default)	Import data from the first sheet.
Name	Import data from the matching sheet name, regardless of order of sheets in the spreadsheet file.

Specification	Behavior
Integer	Import data from sheet in the position denoted by the integer, regardless of the sheet names in the spreadsheet file.

Data Types: `char` | `string` | `single` | `double`

`UseExcel` — Flag to start instance of Microsoft Excel for Windows
`false` (default) | `true`

Flag to start an instance of Microsoft Excel for Windows when reading spreadsheet data, specified as the comma-separated pair consisting of `'UseExcel'` and either `true`, or `false`.

You can set the `'UseExcel'` parameter to one of these values:

- `true` — The importing function starts an instance of Microsoft Excel when reading the file.
- `false` — The importing function does not start an instance of Microsoft Excel when reading the file. When operating in this mode, the importing function functionality differs in the support of file formats and interactive features, such as formulas and macros.

UseExcel	true	false
Supported file formats	`.xls, .xlsx, .xlsm, .xltx, .xltm, .xlsb, .ods`	`.xls, .xlsx, .xlsm, .xltx, .xltm`
Support for interactive features, such as formulas and macros	Yes	No

When reading from spreadsheet files on Windows platforms, if you want to start an instance of Microsoft Excel, then set the `'UseExcel'` parameter to `true`.

See Also

`readcell` | `readmatrix` | `readtable` | `readtimetable`

Introduced in R2019a

real

Real part of complex number

Syntax

```
X = real(Z)
```

Description

`X = real(Z)` returns the real part of each element in array `Z`.

Examples

Real Part of Complex Number

Find the real part of the complex number `Z`.

```
Z = 2+3i;
X = real(Z)
```

```
X = 2
```

Real Part of Vector of Complex Values

Find the real part of each element in vector `Z`. The `real` function acts on `Z` element-wise.

```
Z = [0.5i 1+3i -2.2];
X = real(Z)
```

```
X = 1×3

         0    1.0000   -2.2000
```

Input Arguments

Z — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. `real` operates element-wise when `Z` is nonscalar.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`abs` | `complex` | `conj` | `i` | `imag` | `j`

Introduced before R2006a

reallog

Natural logarithm for nonnegative real arrays

Syntax

```
Y = reallog(X)
```

Description

`Y = reallog(X)` returns the natural logarithm of each element in array `X`. Array `X` must contain only nonnegative real numbers. The size of `Y` is the same as the size of `X`.

Examples

Natural Log of Matrix

Create a 4-by-4 matrix and compute the natural log of each element.

```
X = magic(4)
```

```
X = 4×4

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

```
Y = reallog(X)
```

```
Y = 4×4

    2.7726    0.6931    1.0986    2.5649
    1.6094    2.3979    2.3026    2.0794
    2.1972    1.9459    1.7918    2.4849
```

```
    1.3863    2.6391    2.7081         0
```

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

If the output of the function running on the GPU can be complex, then you must explicitly specify its input arguments as complex. For more information, see “Work with Complex Numbers on a GPU” (Parallel Computing Toolbox).

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`log` | `realpow` | `realsqrt`

Introduced before R2006a

realmax

Largest positive floating-point number

Syntax

```
f = realmax
f = realmax(precision)
```

Description

`f = realmax` returns the largest finite floating-point number in IEEE double precision. This is equal to `(2-2^(-52))*2^1023`.

`f = realmax(precision)` returns the largest finite floating-point number in IEEE single or double precision. This is equal to `realmax` for double precision, and to `single((2-2^(-23))*2^127)` for single precision.

Examples

Double Precision

Set the output format to long scientific notation. Return the largest finite floating-point number in IEEE® double precision.

```
format long e
f = realmax
```

```
f =
    1.797693134862316e+308
```

Single Precision

Return the largest finite floating-point number in IEEE® single precision.

```
f = realmax('single')
```

```
f = single
    3.4028e+38
```

Check the class of `f`.

```
class(f)
```

```
ans =
'single'
```

Input Arguments

`precision` — Floating-point precision type
`'double'` (default) | `'single'`

Floating-point precision type, specified as `'double'` or `'single'`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`flintmax` | `format` | `intmax` | `realmin`

Topics

"Floating-Point Numbers"

Introduced before R2006a

realmin

Smallest normalized floating-point number

Syntax

```
f = realmin
f = realmin(precision)
```

Description

`f = realmin` returns the smallest positive normalized floating-point number in IEEE double precision. This is equal to `2^(-1022)`.

`f = realmin(precision)` returns the smallest positive normalized floating-point number in IEEE single or double precision. This is equal to `realmin` for double precision, and to `single(2^(-126))` for single precision.

Examples

Double Precision

Set the output format to long scientific notation. Return the smallest positive normalized floating-point number in IEEE® double precision.

```
format long e
f = realmin
```

```
f =
    2.225073858507201e-308
```

Single Precision

Return the smallest positive normalized floating-point number in IEEE® single precision.

```
f = realmin('single')
```

```
f = single
    1.1755e-38
```

Check the class of `f`.

```
class(f)
```

```
ans =
'single'
```

Input Arguments

`precision` — Floating-point precision type
`'double'` (default) | `'single'`

Floating-point precision type, specified as `'double'` or `'single'`.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`eps` | `format` | `intmin` | `realmax`

Topics
"Floating-Point Numbers"

Introduced before R2006a

realpow

Array power for real-only output

Syntax

```
Z = realpow(X,Y)
```

Description

`Z = realpow(X,Y)` raises each element of array `X` to the power of its corresponding element in array `Y`. Arrays `X` and `Y` must be the same size. The range of `realpow` is the set of all real numbers, i.e., all elements of the output array `Z` must be real.

Examples

Powers of Matrix Elements

Create a 3-by-3 matrix with the same value in each element.

```
X = -2*ones(3,3)
```

```
X = 3×3

    -2    -2    -2
    -2    -2    -2
    -2    -2    -2
```

Compute a 3-by-3 matrix of integer values.

```
Y = pascal(3)
```

```
Y = 3×3

     1     1     1
```

```
     1     2     3
     1     3     6
```

Compute the element-wise powers of the elements in X corresponding to the exponents defined in Y.

```
Z = realpow(X,Y)
```

```
Z = 3×3

    -2    -2    -2
    -2     4    -8
    -2    -8    64
```

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`mpower` | `power` | `reallog` | `realsqrt`

Introduced before R2006a

realsqrt

Square root for nonnegative real arrays

Syntax

```
Y = realsqrt(X)
```

Description

`Y = realsqrt(X)` returns the square root of each element of array `X`. The size of `Y` is the same as the size of `X`.

If you want negative and complex numbers to return complex results instead of error messages, then use `sqrt` instead.

Examples

Square Root of Matrix Elements

Create a 4-by-4 matrix of real, positive values.

```
X = magic(4)
```

```
X = 4×4

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Compute the square root of each element in `X`.

```
Y = realsqrt(X)
```

```
Y = 4×4

    4.0000    1.4142    1.7321    3.6056
    2.2361    3.3166    3.1623    2.8284
    3.0000    2.6458    2.4495    3.4641
    2.0000    3.7417    3.8730    1.0000
```

Input Arguments

X — Input matrix
scalar | vector | matrix | multidimensional array

Input matrix, specified as a scalar, vector, matrix, or multidimensional array. The values in X must be nonnegative and real.

Data Types: `single` | `double`

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

If the output of the function running on the GPU can be complex, then you must explicitly specify its input arguments as complex. For more information, see “Work with Complex Numbers on a GPU” (Parallel Computing Toolbox).

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`reallog` | `realpow` | `sqrt` | `sqrtm`

Introduced before R2006a

record

(Not recommended) Record data and event information to file

Note This `serial` object function is not recommended. Use `serialport` object functions instead. See “Compatibility Considerations”.

Syntax

```
record(obj)
record(obj,'switch')
```

Description

`record(obj)` toggles the recording state for the serial port object, `obj`.

`record(obj,'switch')` initiates or terminates recording for `obj`. *`switch`* can be `on` or `off`. If *`switch`* is `on`, recording is initiated. If *`switch`* is `off`, recording is terminated.

Examples

This example creates the serial port object `s` on a Windows platform. It connects `s` to the device, configures `s` to record information to a file, writes and reads text data, and then disconnects `s` from the device.

```
s = serial('COM1');
fopen(s)
s.RecordDetail = 'verbose';
s.RecordName = 'MySerialFile.txt';
record(s,'on')
fprintf(s,'*IDN?')
out = fscanf(s);
record(s,'off')
fclose(s)
```

Tips

Before you can record information to disk, `obj` must be connected to the device with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to record information while `obj` is not connected to the device. Each serial port object must record information to a separate file. Recording is automatically terminated when `obj` is disconnected from the device with `fclose`.

The `RecordName` and `RecordMode` properties are read-only while `obj` is recording, and must be configured before using `record`.

Compatibility Considerations

`serial` object interface is not recommended

Not recommended starting in R2019b

Use of this function with a `serial` object is not recommended. To access a serial port device, use a `serialport` object with its functions and properties instead.

See “Transition Your Code to serialport Interface” for more information about using the recommended functionality.

See Also

Functions

`fclose` | `fopen`

Introduced before R2006a

record

Record audio to `audiorecorder` object

Syntax

```
record(recorderObj)
record(recorderObj, length)
```

Description

`record(recorderObj)` records audio from an input device, such as a microphone connected to your system. *`recorderObj`* is an `audiorecorder` object that defines the sample rate, bit depth, and other properties of the recording.

`record(recorderObj, length)` records for the number of seconds specified by *`length`*.

Examples

Record 5 seconds of your speech with a microphone:

```
myVoice = audiorecorder;

% Define callbacks to show when
% recording starts and completes.
myVoice.StartFcn = 'disp(''Start speaking.'')';
myVoice.StopFcn = 'disp(''End of recording.'')';

record(myVoice, 5);
```

To listen to the recording, call the `play` method:

```
play(myVoice);
```

See Also

`audiorecorder` | `getaudiodata` | `recordblocking`

Topics

"Record Audio"
"Record or Play Audio within a Function"

recordblocking

Record audio to `audiorecorder` object, hold control until recording completes

Syntax

`recordblocking(recorderObj, length)`

Description

`recordblocking(recorderObj, length)` records audio from an input device, such as a microphone connected to your system, for the number of seconds specified by *`length`*. The `recordblocking` method does not return control until recording completes. *`recorderObj`* is an `audiorecorder` object that defines the sample rate, bit depth, and other properties of the recording.

Examples

Record 5 seconds of your speech with a microphone, and play it back:

```
myVoice = audiorecorder;

disp('Start speaking.');
recordblocking(myVoice, 5);
disp('End of recording. Playing back ...');

play(myVoice);
```

See Also

`audiorecorder` | `getaudiodata` | `record`

Topics

"Record Audio"

rectangle

Create rectangle with sharp or curved corners

Syntax

```
rectangle('Position',pos)
rectangle('Position',pos,'Curvature',cur)
rectangle( ___ ,Name,Value)
rectangle(ax, ___ )
r = rectangle( ___ )
```

Description

`rectangle('Position',pos)` creates a rectangle in 2-D coordinates. Specify `pos` as a four-element vector of the form `[x y w h]` in data units. The `x` and `y` elements determine the location and the `w` and `h` elements determine the size. The function plots into the current axes without clearing existing content from the axes.

`rectangle('Position',pos,'Curvature',cur)` adds curvature to the sides of the rectangle. For different curvatures along the horizontal and vertical sides, specify `cur` as a two-element vector of the form `[horizontal vertical]`. For the same length of curvature along all sides, specify `cur` as a scalar value. Specify values between 0 (no curvature) and 1 (maximum curvature). Use `[1 1]` to create an ellipse or circle.

`rectangle( ___ ,Name,Value)` specifies rectangle properties using one or more name-value pair arguments. For example, `'FaceColor','red'` specifies a red fill color. You can specify rectangle properties with any of the input argument combinations in the previous syntaxes.

`rectangle(ax, ___ )` creates the rectangle in the axes specified by `ax` instead of in the current axes (`gca`). The option `ax` can precede any of the input argument combinations in the previous syntaxes.

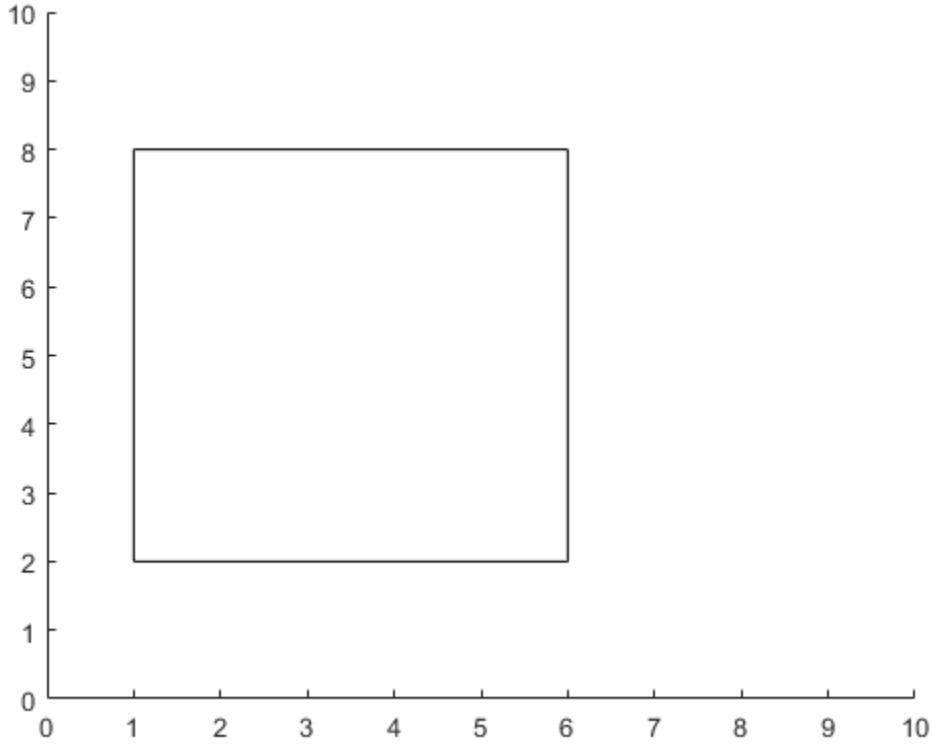
`r = rectangle( ___ )` returns the rectangle object. Use `r` to modify properties of the rectangle after it is created. For a list of properties, see Rectangle. You can return an output argument with any of the previous syntaxes.

Examples

Draw Rectangle

Draw a rectangle that has a lower left corner at the point (1,2). Make the width of the rectangle 5 units and the height 6 units. Then, change the axis limits.

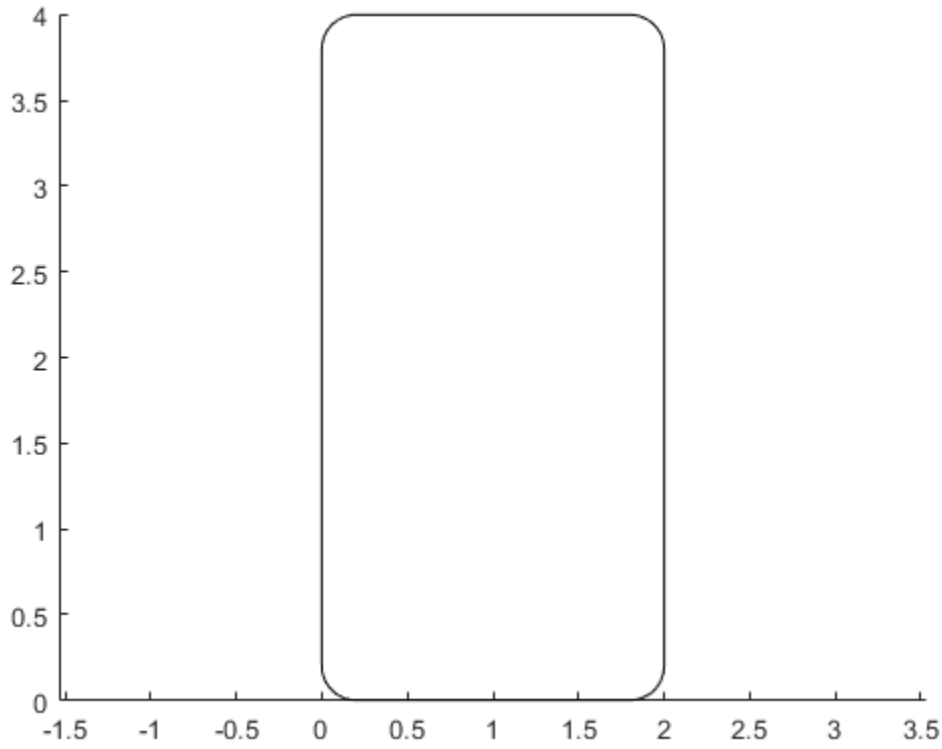
```
rectangle('Position',[1 2 5 6])
axis([0 10 0 10])
```



Draw Rectangle with Curved Edges

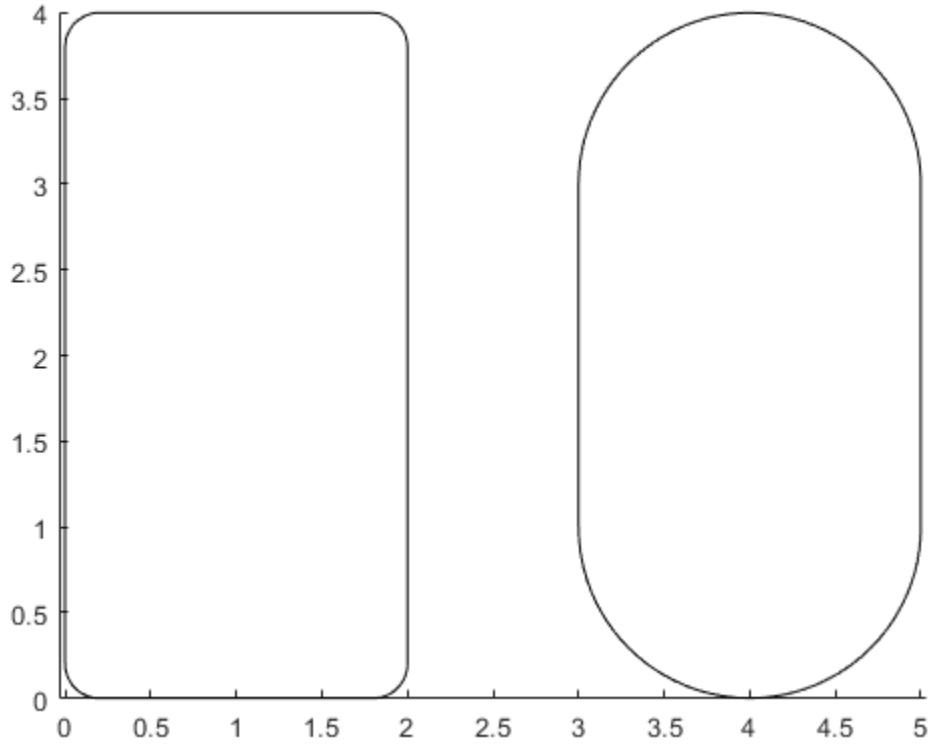
Draw a rectangle that has a lower left corner at (0,0) and an upper right corner at (2,4). Create the rectangle with curved corners by specifying the curvature as the scalar value `0.2`. For data units of equal length along both the *x*-axis and *y*-axis, use `axis equal`.

```
figure
rectangle('Position',[0 0 2 4],'Curvature',0.2)
axis equal
```



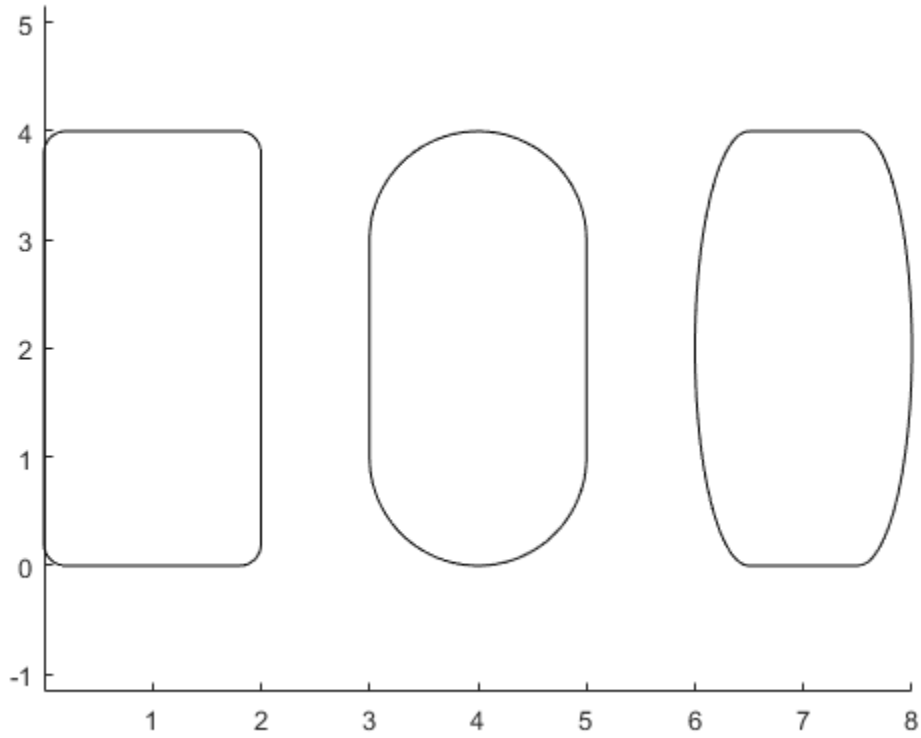
Add a second rectangle that has the shortest side completely curved by specifying the curvature as the scalar value `1`.

```
rectangle('Position',[3 0 2 4],'Curvature',1)
```



Add a third rectangle and use different curvatures along the horizontal and vertical sides. Use a horizontal curvature of 0.5 to make 50 percent of each horizontal side curved. Use a vertical curvature of 1 to make both vertical sides completely curved.

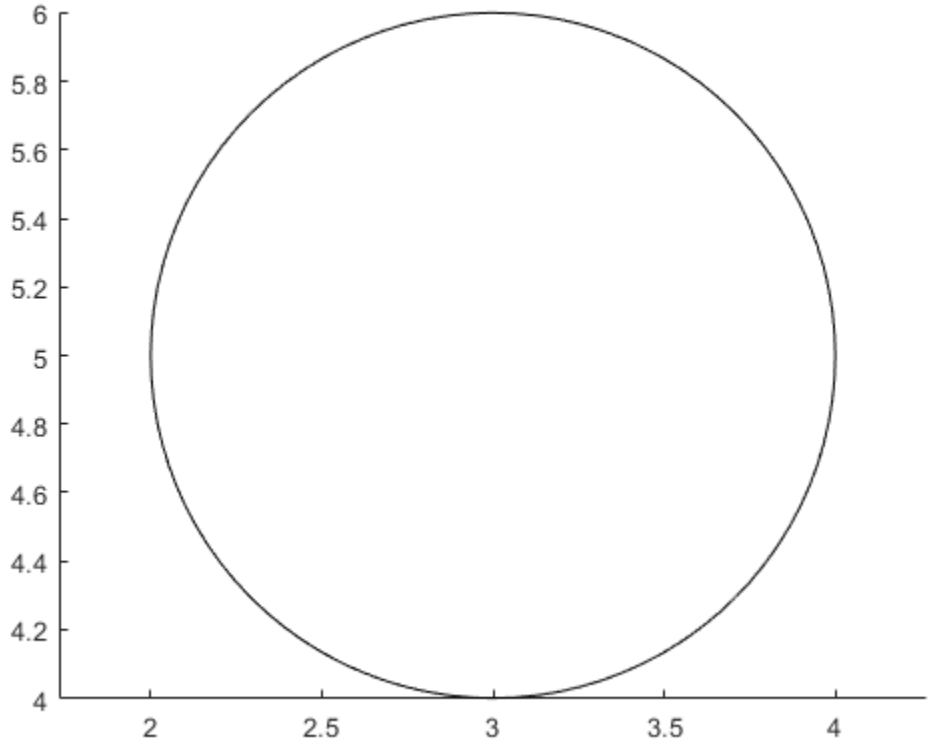
```
rectangle('Position',[6 0 2 4],'Curvature',[0.5,1])
```



Draw Circle

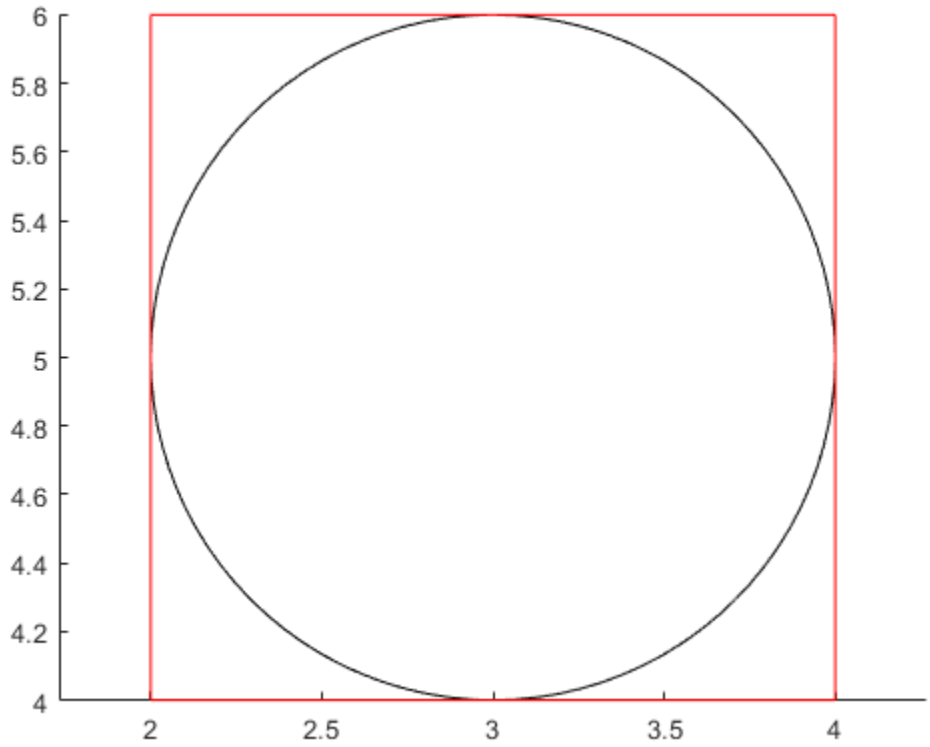
Draw a circle by setting the `Curvature` property to `[1 1]`. Draw the circle so that it fills the rectangular area between the points (2,4) and (4,6). The `Position` property defines the smallest rectangle that contains the circle.

```
pos = [2 4 2 2];
rectangle('Position',pos,'Curvature',[1 1])
axis equal
```



Draw a red rectangle using the same position values to show how the circle fills the rectangular area.

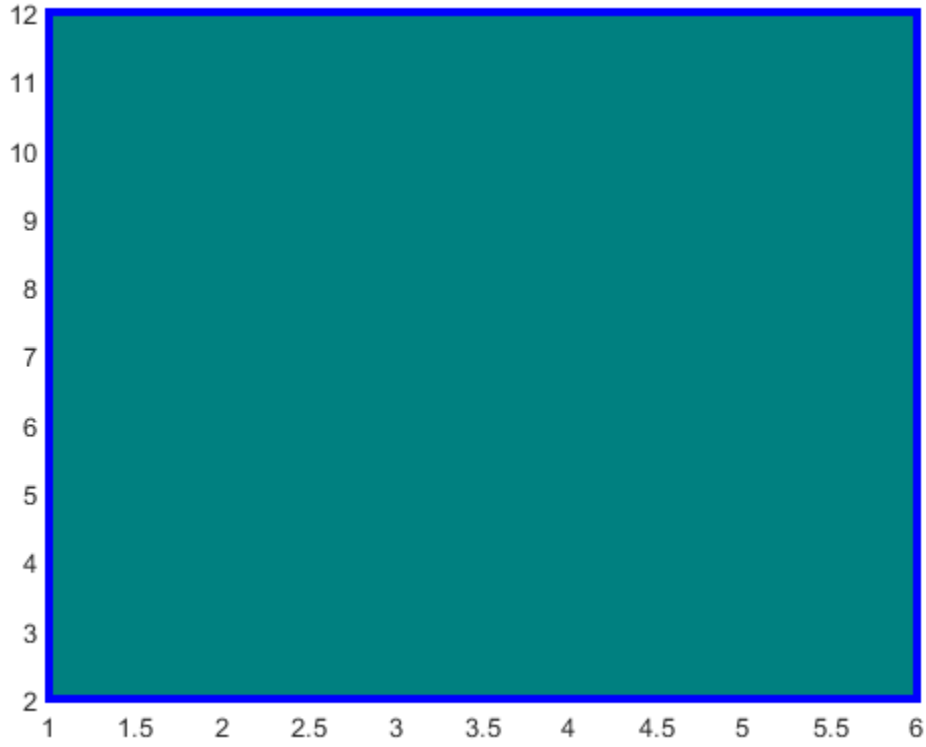
```
rectangle('Position',pos,'EdgeColor','r')
```



Specify Rectangle Outline and Fill Color

Draw a rectangle and specify the fill color, outline color, and outline width.

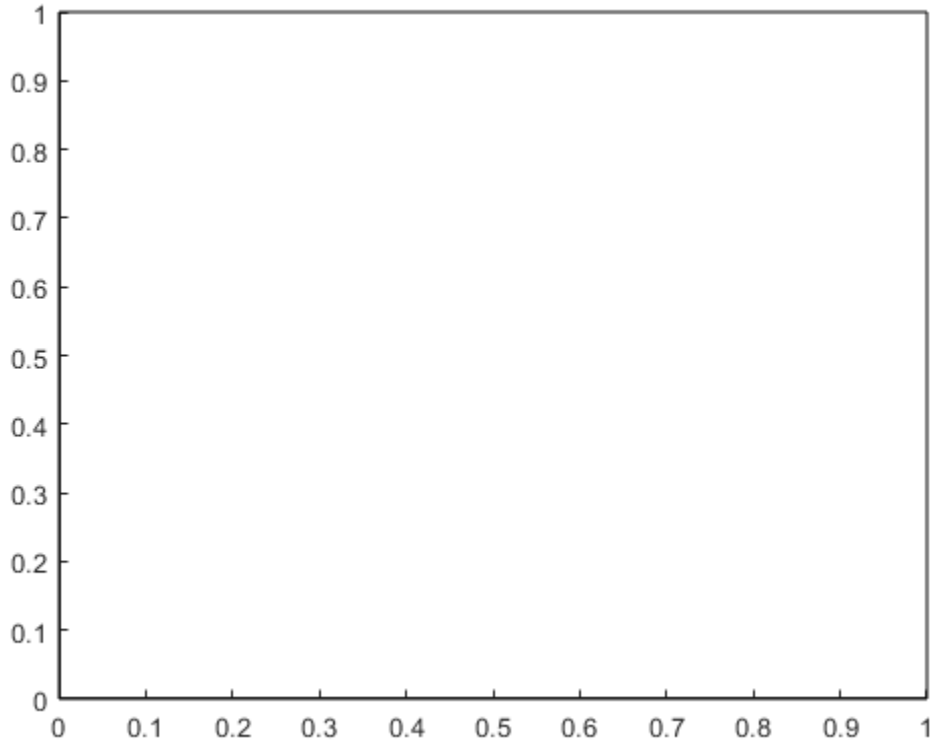
```
rectangle('Position',[1,2,5,10],'FaceColor',[0 .5 .5],'EdgeColor','b',...
    'LineWidth',3)
```



Modify Rectangle After Creation

Add a rectangle to the current axes and return the rectangle object, `r`.

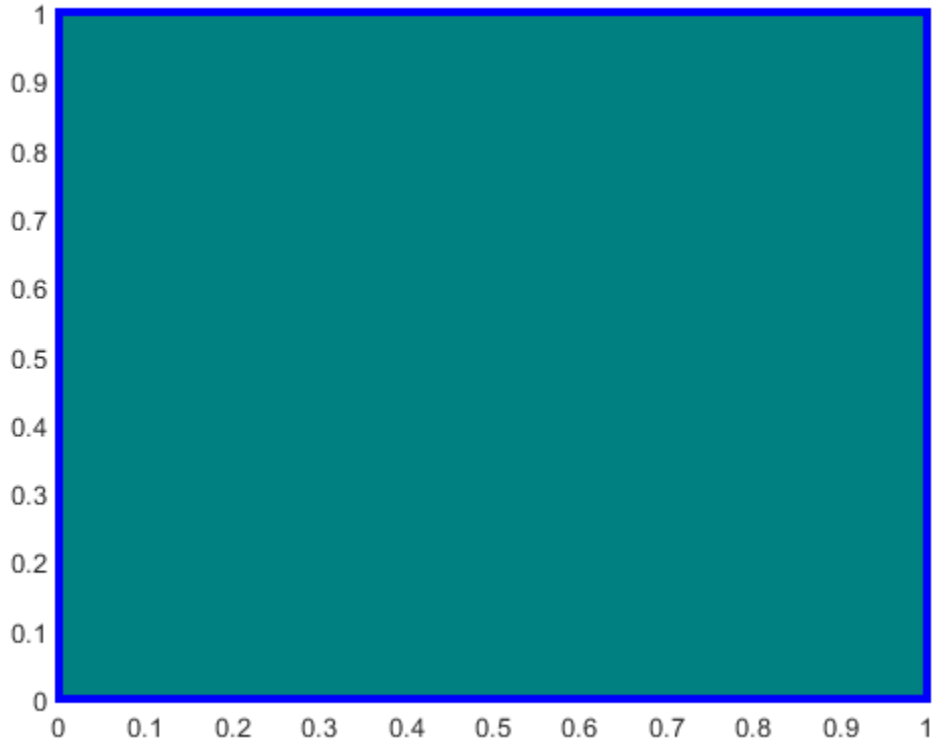
```
r = rectangle('Position',[0 0 1 1]')
```




```
r =
  Rectangle with properties:

    FaceColor: 'none'
    EdgeColor: [0 0 0]
    LineWidth: 0.5000
    LineStyle: '-'
    Curvature: [0 0]
     Position: [0 0 1 1]

  Show all properties
```

Modify the rectangle after creation using `r`. Change the fill color, outline color, and outline width.

```
r.FaceColor = [0 .5 .5];
r.EdgeColor = 'b';
r.LineWidth = 3;
```



Input Arguments

pos — Size and location of rectangle
four-element vector of the form `[x y w h]`

Size and location of the rectangle, specified as a four-element vector of the form `[x y w h]`. The `x` and `y` elements define the coordinate for the lower left corner of the rectangle. The `w` and `h` elements define the dimensions of the rectangle.

All values are in data units.

Example: `rectangle('Position',[0.5 0.5 0.3 0.4])`

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`cur` — Amount of horizontal and vertical curvature
`[0 0]` (default) | two-element vector | scalar

Amount of horizontal and vertical curvature, specified as a two-element vector or a scalar value. Use this property to vary the shape of the rectangle from rectangular to ellipsoidal.

The horizontal curvature is the fraction of the width that is curved along the top and bottom edges. The vertical curvature is the fraction of the height that is curved along the left and right edges.

- To use different horizontal and vertical curvatures, specify a two-element vector of the form `[horizontal vertical]`. The `horizontal` element determines the horizontal curvature and the `vertical` element determines the vertical curvature. Specify the values between 0 (no curvature) and 1 (maximum curvature). For example, a value of `[0 0]` creates a rectangle with square edges and value of `[1 1]` creates an ellipse.
- To use the same curvature for the horizontal and vertical edges, specify a scalar value between 0 and 1, inclusive. The shorter dimension determines the length of the curvature.

Example: `[0.5 0.6]`

Example: `0.75`

Data Types: `double`

`ax` — Axes object
axes object

Axes object. If you do not specify an axes object, then `rectangle` uses the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `rectangle('Position',[0 0 1 1],'EdgeColor','k','FaceColor',[0 .5 .5])`

`FaceColor` — Fill color

`'none'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Fill color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`EdgeColor` — Outline color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Outline color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default RGB triplet value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`LineWidth` — Line width

`0.5` (default) | positive value

Line width, specified as a positive value in point units.

Example: `0.75`

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

See Also

Functions
`annotation` | `line` | `patch`

Properties
Rectangle

Introduced before R2006a

Rectangle Properties

Rectangle appearance and behavior

Description

Rectangle properties control the appearance and behavior of a rectangle object. By changing property values, you can modify certain aspects of the rectangle.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = rectangle;
w = h.LineWidth;
h.LineWidth = 3;
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Color and Styling

`FaceColor` — Fill color

`'none'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Fill color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`EdgeColor` — Outline color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Outline color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in point units.

Example: `0.75`

`Curvature` — Amount of horizontal and vertical curvature
`[0 0]` (default) | two-element vector | scalar

Amount of horizontal and vertical curvature, specified as a two element vector of the form `[x y]` or a scalar value. Use this property to vary the shape of the rectangle from

rectangular to ellipsoidal. The horizontal curvature is the fraction of the width that is curved along the top and bottom edges. The vertical curvature is the fraction of the height that is curved along the left and right edges.

- To use different horizontal and vertical curvatures, specify a two-element vector of the form `[x y]`. The `x` element determines the horizontal curvature and the `y` element determines the vertical curvature. Specify `x` and `y` as values between 0 (no curvature) and 1 (maximum curvature). For example, a value of `[0 0]` creates a rectangle with square edges and value of `[1 1]` creates an ellipse.
- To use the same curvature for the horizontal and vertical edges, specify a scalar value in the range `[0,1]`. The shorter dimension determines the length of the curvature.

Example: `[0.5 0.6]`

Example: `0.75`

`AlignVertexCenters` — Sharp vertical and horizontal lines

`'off'` (default) | `'on'`

Sharp vertical and horizontal lines, specified as `'off'` or `'on'`.

If the associated figure has a `GraphicsSmoothing` property set to `'on'` and a `Renderer` property set to `'opengl'`, then the figure applies a smoothing technique to plots. In some cases, this smoothing technique can cause vertical and horizontal lines to appear uneven in thickness or color. Use the `AlignVertexCenters` property to eliminate the uneven appearance.

- `'off'` — Do not sharpen vertical or horizontal lines. The lines might appear uneven in thickness or color.
- `'on'` — Sharpen vertical and horizontal lines to eliminate an uneven appearance.

Note You must have a graphics card that supports this feature. To see if the feature is supported, call the `rendererinfo` function. If it is supported, `rendererinfo` returns value of `1` for `info.Details.SupportsAlignVertexCenters`.

Position

`Position` — Size and location of rectangle

`[0 0 1 1]` (default) | four-element vector

Size and location of the rectangle, specified as a four-element vector of the form `[x y width height]`. Specify the values in data units. The `x` and `y` elements define the coordinate for the lower-left corner of the rectangle. The `width` and `height` elements define the dimensions of the rectangle.

Example: `[0.5 0.5 0.3 0.4]`

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

`Clipping` — Clipping of object to axes limits

`'on'` (default) | `'off'`

Clipping of the object to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the object that are outside the axes limits.
- `'off'` — Display the entire object, even if parts of it appear outside the axes limits. Parts of the object might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the object so that it is larger than the original plot.

The `Clipping` property of the axes that contains the object must be set to `'on'`. Otherwise, this property has no effect. For more information about the clipping behavior, see the `Clipping` property of the axes.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see "Callback Definition".

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.

  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.

- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks when visible. The `Visible` property must be set to `'on'` and you must click a part of the `Rectangle` object that has a defined color. You cannot click a part that has an associated color property set to `'none'`. The `HitTest` property determines if the `Rectangle` object responds to the click or if an ancestor does.
- `'all'` — Capture mouse clicks regardless of visibility. The `Visible` property can be set to `'on'` or `'off'` and you can click a part of the `Rectangle` object that has no color. The `HitTest` property determines if the `Rectangle` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `Rectangle` object passes the click through it to the object below it in the current view of the figure window. The `HitTest` property has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `Rectangle` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `Rectangle` object that has one of these:
  - `HitTest` property set to `'on'`
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

---

Note The `PickableParts` property determines if the `Rectangle` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent
`Axes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `Group`, or `Transform` object.

`Children` — Children
empty `GraphicsPlaceholder` array

The object has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'rectangle'`

This property is read-only.

Type of graphics object, returned as `'rectangle'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[ ]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`rectangle`

Topics

“Access Property Values”
“Graphics Object Properties”

Introduced before R2006a

rectint

Rectangle intersection area

Syntax

```
area = rectint(A,B)
```

Description

`area = rectint(A,B)` returns the area of intersection of the rectangles specified by position vectors `A` and `B`.

If `A` and `B` each specify one rectangle, the output `area` is a scalar.

`A` and `B` can also be matrices, where each row is a position vector. `area` is then a matrix giving the intersection of all rectangles specified by `A` with all the rectangles specified by `B`. That is, if `A` is `n`-by-4 and `B` is `m`-by-4, then `area` is an `n`-by-`m` matrix where `area(i,j)` is the intersection area of the rectangles specified by the `i`th row of `A` and the `j`th row of `B`.

Note A position vector is a four-element vector `[x,y,width,height]`, where the point defined by `x` and `y` specifies one corner of the rectangle, and `width` and `height` define the size in units along the x and y axes respectively.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`polyarea`

Introduced before R2006a

recycle

Set option to move deleted files to recycle folder

Syntax

```
status = recycle
previousState = recycle(state)
```

Description

`status = recycle` returns the current state for recycling files you remove using the `delete` function. When `status` is `off`, the `delete` function permanently removes the files. When `status` is `on`, deleted files move to a different location. For details, see the Tips on page 1-13204 section.

Note On Mac platforms, the `recycle` option is not applied to files deleted from network drives.

`previousState = recycle(state)` sets the recycle option for MATLAB to the specified state, either `on` or `off`. The returned `previousState` value is the recycle state before running the statement.

Examples

View Current Recycling State

Start from a state where file recycling is off. Verify the current recycle state.

```
state = recycle
```

```
state =

off
```

Turn File Recycling On

Turn on file recycling. Then, delete an existing file and move it to the recycle bin or temporary folder.

```
recycle('on');
delete('myfile.txt')
```

Input Arguments

`state` — State of recycle option
`'on'` | `'off'`

State of the recycle option, specified as `'on'` or `'off'`.

Tips

- The location for storing recycled files varies by platform, as follows:
  - Microsoft Windows platforms — Recycle bin.
  - Mac platforms — Trash.
  - Linux platforms — Subfolder with the prefix `MATLAB_Files_` in the system temporary folder, as returned by the `tempdir` function.
- The general preference for **Deleting files** sets the state of the `recycle` function at startup. When you change the preference, MATLAB changes the state of `recycle`. When you change the state of `recycle`, MATLAB also updates the **Deleting files** preference.

See Also

`delete` | `dir` | `ls` | `rmdir`

Topics

"General Preferences"

Introduced before R2006a

reducepatch

Reduce number of patch faces

Syntax

```
reducepatch(p,r)
nfv = reducepatch(p,r)
nfv = reducepatch(fv,r)
nfv = reducepatch(p)
nfv = reducepatch(fv)
reducepatch(...,'fast')
reducepatch(...,'verbose')
nfv = reducepatch(f,v,r)
[nf,nv] = reducepatch(...)
```

Description

`reducepatch(p,r)` reduces the number of faces of the patch identified by handle `p`, while attempting to preserve the overall shape of the original object. The MATLAB software interprets the reduction factor `r` in one of two ways depending on its value:

- If `r` is less than 1, `r` is interpreted as a fraction of the original number of faces. For example, if you specify `r` as 0.2, then the number of faces is reduced to 20% of the number in the original patch.
- If `r` is greater than or equal to 1, then `r` is the target number of faces. For example, if you specify `r` as 400, then the number of faces is reduced until there are 400 faces remaining.

`nfv = reducepatch(p,r)` returns the reduced set of faces and vertices but does not set the `Faces` and `Vertices` properties of patch `p`. The struct `nfv` contains the faces and vertices after reduction.

`nfv = reducepatch(fv,r)` performs the reduction on the faces and vertices in the struct `fv`.

`nfv = reducepatch(p)` and `nfv = reducepatch(fv)`uses a reduction value of `0.5`.

`reducepatch(...,'fast')` assumes the vertices are unique and does not compute shared vertices.

`reducepatch(...,'verbose')` prints progress messages to the command window as the computation progresses.

`nfv = reducepatch(f,v,r)` performs the reduction on the faces in `f` and the vertices in `v`.

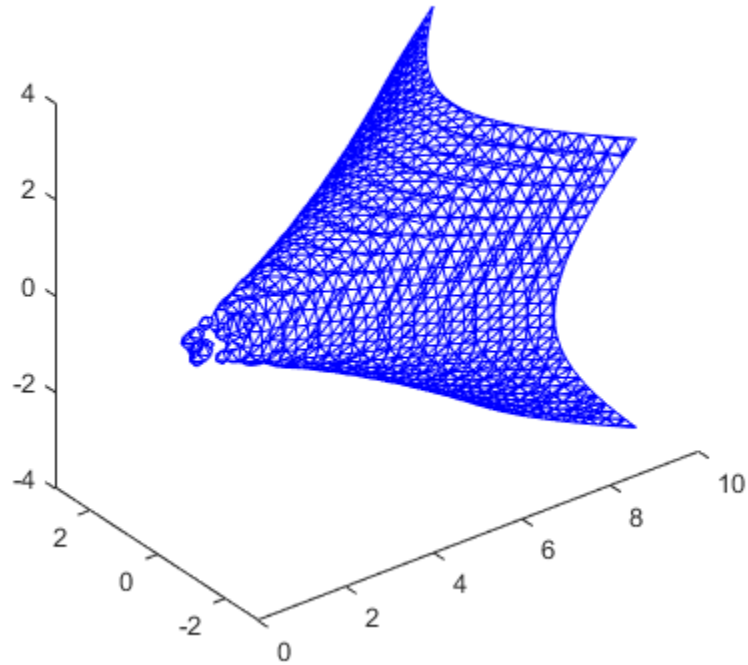
`[nf,nv] = reducepatch(...)` returns the faces and vertices in the arrays `nf` and `nv`.

Examples

Reduce Number of Patch Faces

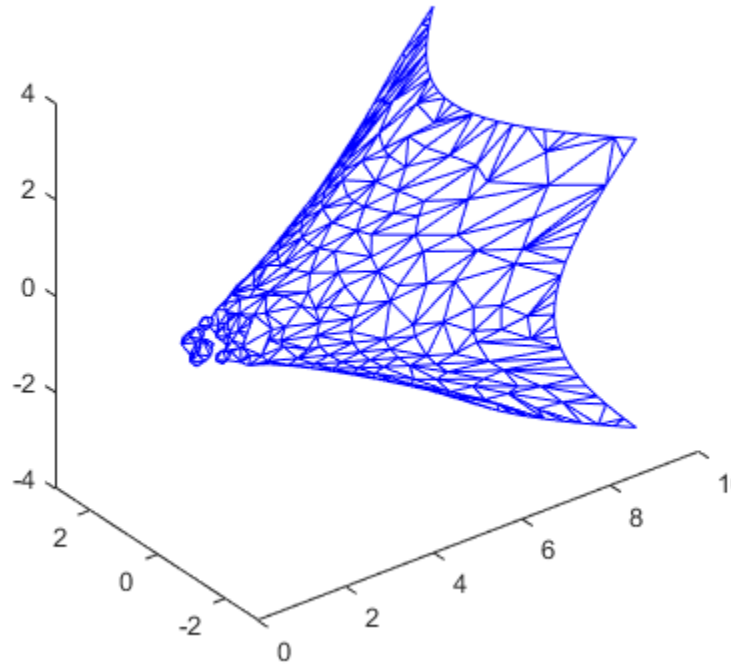
This example illustrates the effect of reducing the number of faces to only 15% of the original value.

```
figure
[x,y,z,v] = flow;
p = patch(isosurface(x,y,z,v,-3));
p.FaceColor = 'w';
p.EdgeColor = 'b';
daspect([1,1,1])
view(3)
```



Reduce the number of faces.

```
reducepatch(p,0.15)
```



Tips

If the patch contains nonshared vertices, MATLAB computes shared vertices before reducing the number of faces. If the faces of the patch are not triangles, MATLAB triangulates the faces before reduction. The faces returned are always defined as triangles.

The number of output triangles may not be exactly the number specified with the reduction factor argument (`r`), particularly if the faces of the original patch are not triangles.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`isocaps` | `isonormals` | `isosurface` | `reducevolume` | `smooth3` | `subvolume`

Topics

“Vector Field Displayed with Cone Plots”

Introduced before R2006a

reducevolume

Reduce number of elements in volume data set

Syntax

```
[nx,ny,nz,nv] = reducevolume(X,Y,Z,V,[Rx,Ry,Rz])
[nx,ny,nz,nv] = reducevolume(V,[Rx,Ry,Rz])
nv = reducevolume(...)
```

Description

`[nx,ny,nz,nv] = reducevolume(X,Y,Z,V,[Rx,Ry,Rz])` reduces the number of elements in the volume by retaining every Rxth element in the *x* direction, every Ryth element in the *y* direction, and every Rzth element in the *z* direction. If a scalar R is used to indicate the amount or reduction instead of a three-element vector, the MATLAB software assumes the reduction to be `[R R R]`.

The arrays X, Y, and Z define the coordinates for the volume V. The reduced volume is returned in nv, and the coordinates of the reduced volume are returned in nx, ny, and nz.

`[nx,ny,nz,nv] = reducevolume(V,[Rx,Ry,Rz])` assumes the arrays X, Y, and Z are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)`, where `[m,n,p] = size(V)`.

`nv = reducevolume(...)` returns only the reduced volume.

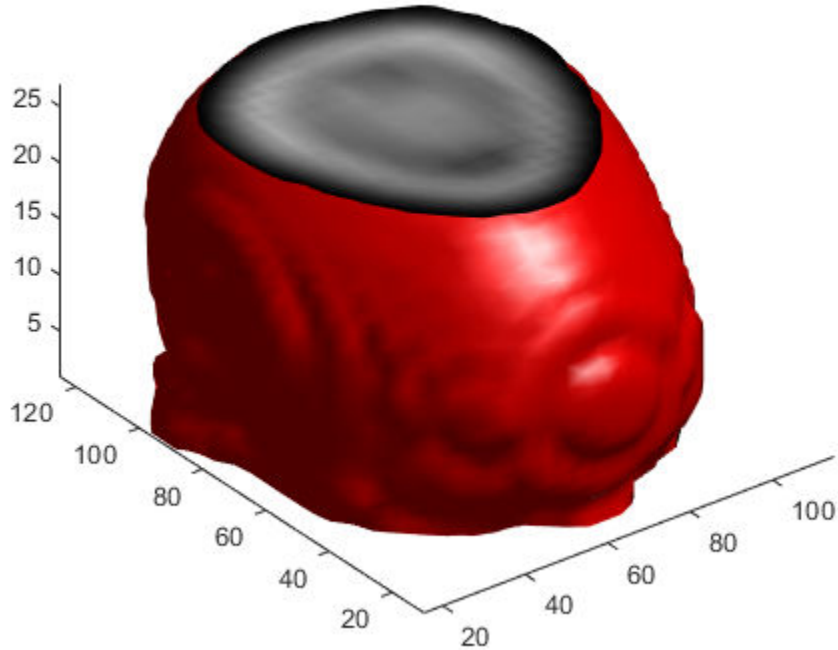
Examples

Reduce Volume Data Set

This example uses a data set that is a collection of MRI slices of a human skull. This data is processed in a variety of ways:

- The 4-D array is squeezed (`squeeze`) into three dimensions and then reduced (`reducevolume`) so that what remains is every fourth element in the x and y directions and every element in the z direction.
- The reduced data is smoothed (`smooth3`).
- The outline of the skull is an isosurface generated as a patch (`p1`) whose vertex normals are recalculated to improve the appearance when lighting is applied (`patch`, `isosurface`, `isonormals`).
- A second patch (`p2`) with an interpolated face color draws the end caps (`FaceColor`) isocaps).
- The view of the object is set (`view`, `axis`, `daspect`).
- A 100-element grayscale colormap provides coloring for the end caps (`colormap`).
- Adding a light to the right of the camera illuminates the object (`camlight`, `lighting`).

```
load mri
D = squeeze(D);
[x,y,z,D] = reducevolume(D,[4,4,1]);
D = smooth3(D);
p1 = patch(isosurface(x,y,z,D,5),...
    'FaceColor','red','EdgeColor','none');
isonormals(x,y,z,D,p1)
p2 = patch(isocaps(x,y,z,D,5),...
    'FaceColor','interp','EdgeColor','none');
view(3)
axis tight
daspect([1,1,.4])
colormap(gray(100))
camlight
lighting gouraud
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`isocaps` | `isonormals` | `isosurface` | `reducepatch` | `smooth3` | `subvolume`

Introduced before R2006a

refresh

Redraw current figure

Syntax

```
refresh
refresh(h)
```

Description

`refresh` erases and redraws the current figure.

`refresh(h)` redraws the figure identified by `h`.

See Also

Introduced before R2006a

refreshdata

Refresh charted data

Syntax

```
refreshdata
refreshdata(target)
refreshdata(target,workspace)
```

Description

`refreshdata` updates charted data in the current figure to reflect changes in workspace variables. Before using `refreshdata`, you must first set the data source properties (for example, `XDataSource`) of the charts. The function evaluates these data sources before updating the charts.

`refreshdata(target)` updates charted data in the specified figure, axes, or chart with data source properties.

`refreshdata(target,workspace)` specifies the workspace where the data source properties are evaluated.

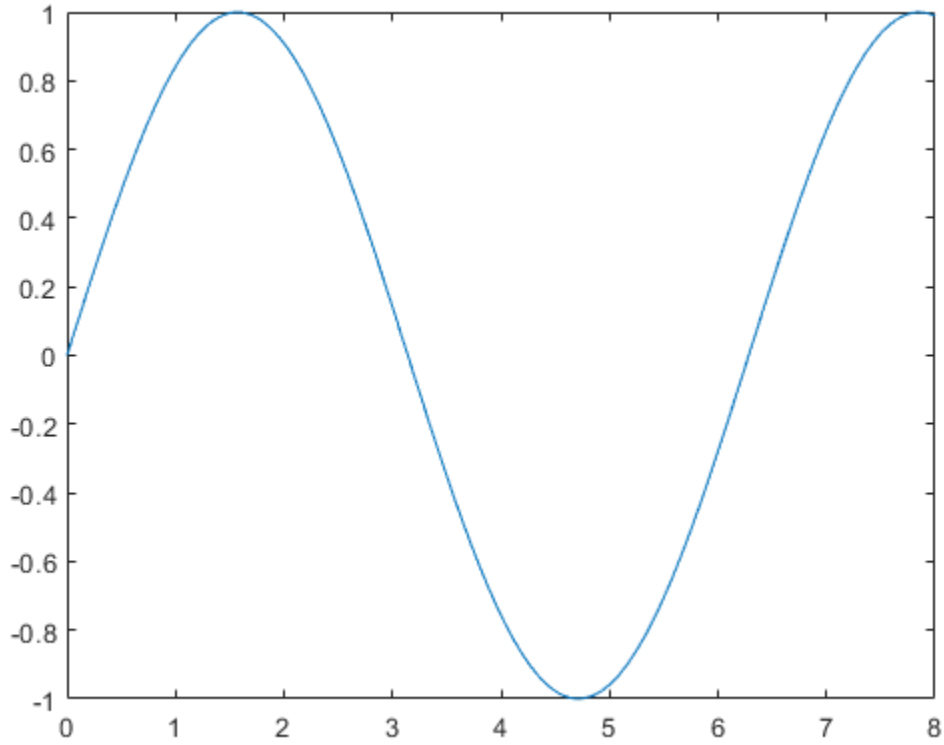
Examples

Refresh Plotted Data

Refresh charted data to reflect changes in workspace variables.

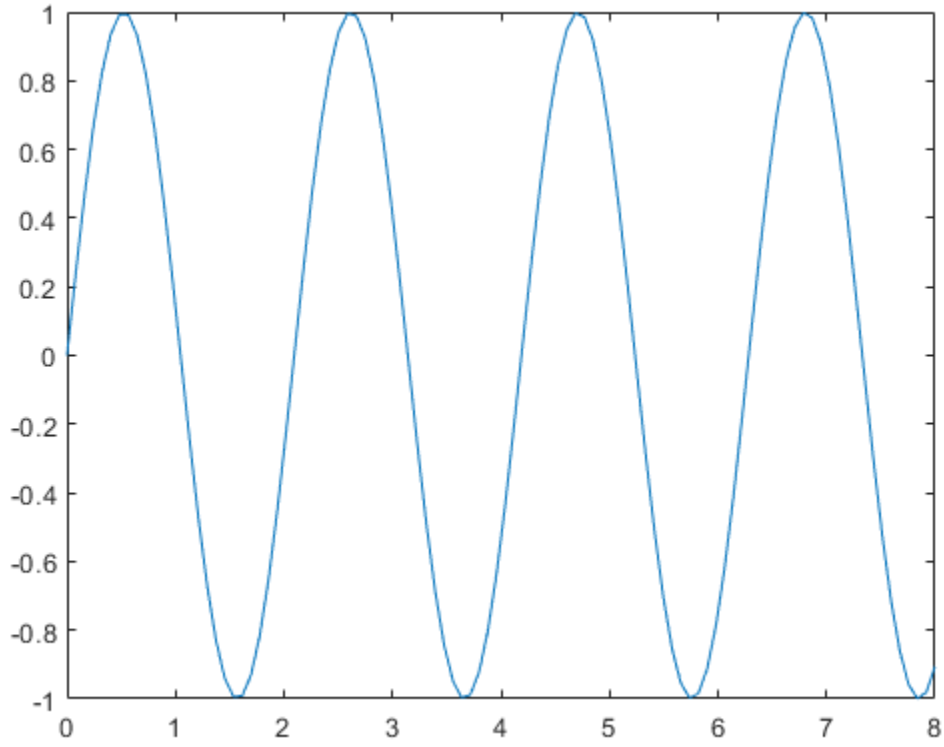
First, plot some data and return a `Line` object.

```
x = linspace(0,8);
y = sin(x);
ln = plot(x,y);
```



Identify the data sources for the chart by setting the `XDataSource` and `YDataSource` properties of the `Line` object. Then, modify `y`. Update the chart by calling `refreshdata`.

```
ln.XDataSource = 'x';
ln.YDataSource = 'y';
y = sin(3.*x);

refreshdata
```



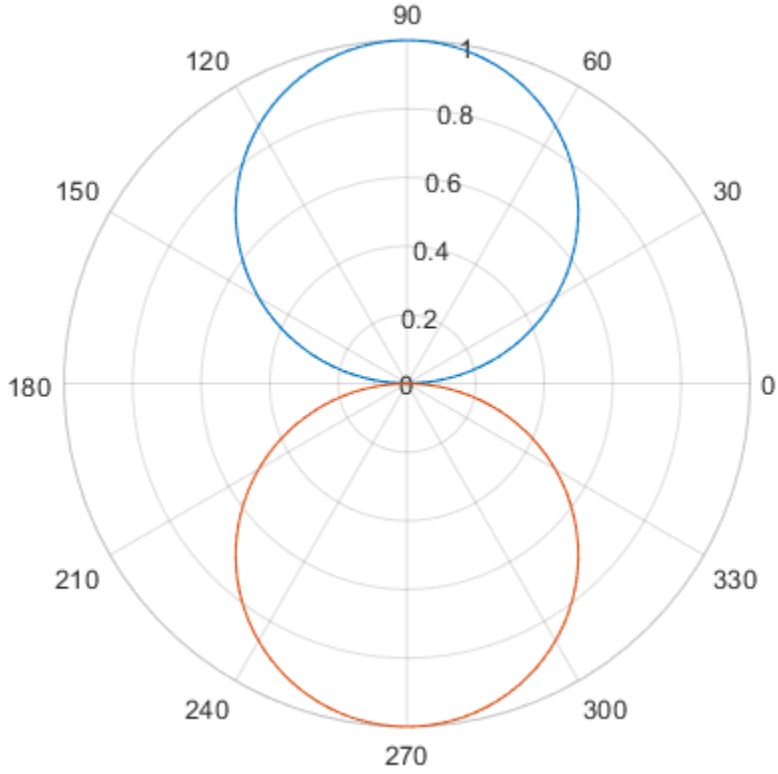
Refresh Plotted Data for One Line

Refresh charted data for a specific line and leave other lines unchanged.

First, plot two sets of polar data and return the `Line` objects.

```
theta = linspace(0,2*pi);
rho1 = sin(theta);
rho2 = -sin(theta);

px = polaraxes;
```

```
hold on;
ln1 = polarplot(theta,rho1);
ln2 = polarplot(theta,rho2);
```

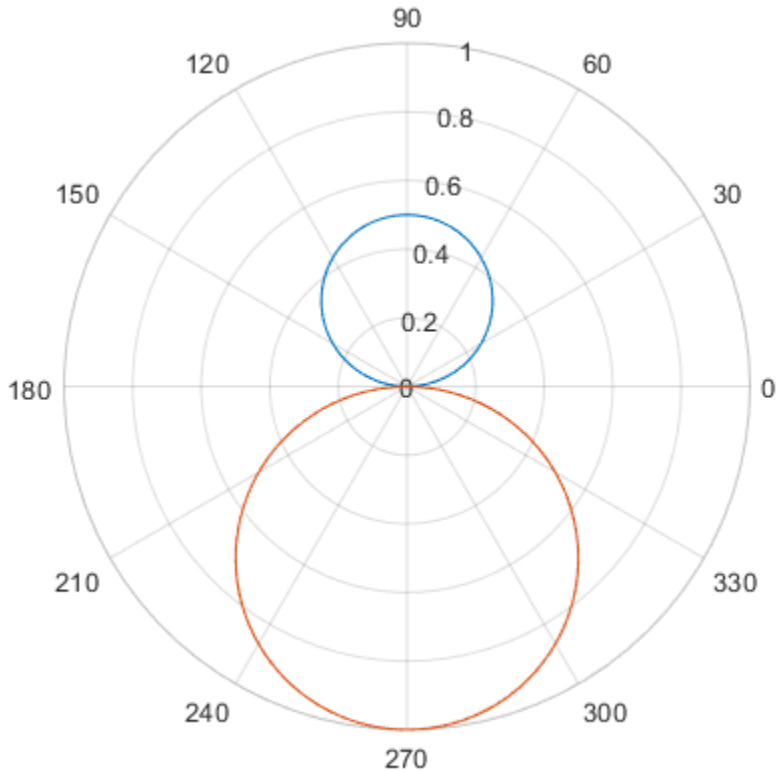


Set the `RDataSource` property for each line. Then, modify `rho1` and `rho2`.

```
ln1.RDataSource = 'rho1';
ln2.RDataSource = 'rho2';

rho1 = 0.5*sin(theta);
rho2 = 0.5*cos(theta);
```

Update only the first line by using `ln1` as the input argument to `refreshdata`. Note that the second line remains unchanged.

```
refreshdata(ln1)
```



Input Arguments

target — Target object

Figure object | Axes object | PolarAxes object | GeographicAxes object | chart object

Target object, specified as a Figure object, Axes object, PolarAxes object, GeographicAxes object, or chart object with data source properties. For example, Line objects have an XDataSource and YDataSource property.

`workspace` — Workspace
`'base'` (default) | `'caller'`

Workspace where data source properties are evaluated, specified as one of these values:

- `'base'` — Evaluate data source properties in the base workspace.
- `'caller'` — Evaluate data source properties in the workspace of the function that called `refreshdata`.

If you set data source properties of variables that are not in the base workspace, then you must specify the workspace option as `'caller'`.

See Also

Functions
`linkdata` | `linkprop`

Properties
Line Properties

Topics
"Automatically Refresh Plot After Changing Data"

Introduced before R2006a

regexp

Match regular expression (case sensitive)

Syntax

```
startIndex = regexp(str,expression)
[startIndex,endIndex] = regexp(str,expression)

out = regexp(str,expression,outkey)
[out1,...,outN] = regexp(str,expression,outkey1,...,outkeyN)

___ = regexp(___,option1,...,optionM)

___ = regexp(___,'forceCellOutput')
```

Description

`startIndex = regexp(str,expression)` returns the starting index of each substring of `str` that matches the character patterns specified by the regular expression. If there are no matches, `startIndex` is an empty array.

`[startIndex,endIndex] = regexp(str,expression)` returns the starting and ending indices of all matches.

`out = regexp(str,expression,outkey)` returns the output specified by `outkey`. For example, if `outkey` is `'match'`, then `regexp` returns the substrings that match the expression rather than their starting indices.

`[out1,...,outN] = regexp(str,expression,outkey1,...,outkeyN)` returns the outputs specified by multiple output keywords, in the specified order. For example, if you specify `'match'`,`'tokens'`, then `regexp` returns substrings that match the entire expression and tokens on page 1-13249 that match parts of the expression.

`___ = regexp(___,option1,...,optionM)` modifies the search using the specified option flags. For example, specify `'ignorecase'` to perform a case-insensitive match. You can include any of the inputs and request any of the outputs from previous syntaxes.

`___ = regexp( ___ ,'forceCellOutput')` returns each output argument as a scalar cell. The cells contain the numeric arrays or substrings that are described as the outputs of the previous syntaxes. You can include any of the inputs and request any of the outputs from previous syntaxes.

Examples

Find Patterns in Text

Find words that start with `c`, end with `t`, and contain one or more vowels between them.

```
str = 'bat cat can car coat court CUT ct CAT-scan';
expression = 'c[aeiou]+t';
startIndex = regexp(str,expression)
```

```
startIndex = 1×2

     5    17
```

The regular expression `'c[aeiou]+t'` specifies this pattern:

- `c` must be the first character.
- `c` must be followed by one of the characters inside the brackets, `[aeiou]`.
- The bracketed pattern must occur one or more times, as indicated by the `+` operator.
- `t` must be the last character, with no characters between the bracketed pattern and the `t`.

Values in `startIndex` indicate the index of the first character of each word that matches the regular expression. The matching word `cat` starts at index 5, and `coat` starts at index 17. The words `CUT` and `CAT` do not match because they are uppercase.

Find Patterns in Multiple Pieces of Text

Find the location of capital letters and spaces within character vectors in a cell array.

```
str = {'Madrid, Spain','Romeo and Juliet','MATLAB is great'};
capExpr = '[A-Z]';
```

```
spaceExpr = '\s';

capStartIndex = regexp(str,capExpr);
spaceStartIndex = regexp(str,spaceExpr);
```

`capStartIndex` and `spaceStartIndex` are cell arrays because the input `str` is a cell array.

View the indices for the capital letters.

```
celldisp(capStartIndex)
```

```
capStartIndex{1} =

     1     9


capStartIndex{2} =

     1    11


capStartIndex{3} =

     1     2     3     4     5     6
```

View the indices for the spaces.

```
celldisp(spaceStartIndex)
```

```
spaceStartIndex{1} =

     8


spaceStartIndex{2} =

     6    10
```

```
spaceStartIndex{3} =

     7    10
```

Return Substrings Using match Keyword

Capture words within a character vector that contain the letter `x`.

```
str = 'EXTRA! The regexp function helps you relax.';
expression = '\w*x\w*';
matchStr = regexp(str,expression,'match')
```

```
matchStr = 1x2 cell array
    {'regexp'}    {'relax'}
```

The regular expression `'\w*x\w*'` specifies that the character vector:

- Begins with any number of alphanumeric or underscore characters, `\w*`.
- Contains the lowercase letter `x`.
- Ends with any number of alphanumeric or underscore characters after the `x`, including none, as indicated by `\w*`.

Split Text at Delimiter Using split Keyword

Split a character vector into several substrings, where each substring is delimited by a `^` character.

```
str = ['Split ^this text into ^several pieces'];
expression = '\^';
splitStr = regexp(str,expression,'split')
```

```
splitStr = 1x3 cell array
    {'Split '}    {'this text into '}    {'several pieces'}
```

Because the caret symbol has special meaning in regular expressions, precede it with the escape character, a backslash (\). To split a character vector at other delimiters, such as a semicolon, you do not need to include the backslash.

Return Both Matching and Nonmatching Substrings

Capture parts of a character vector that match a regular expression using the `'match'` keyword, and the remaining parts that do not match using the `'split'` keyword.

```
str = 'She sells sea shells by the seashore.';
expression = '[Ss]h.';
[match,noMatch] = regexp(str,expression,'match','split')
```

```
match = 1x3 cell array
    {'She'}    {'she'}    {'sho'}


noMatch = 1x4 cell array
    {0x0 char}    {' sells sea '}    {'lls by the sea'}    {'re.'}
```

The regular expression `'[Ss]h.'` specifies that:

- `S` or `s` is the first character.
- `h` is the second character.
- The third character can be anything, including a space, as indicated by the dot (`.`).

When the first (or last) character in a character vector matches a regular expression, the first (or last) return value from the `'split'` keyword is an empty character vector.

Optionally, reassemble the original character vector from the substrings.

```
combinedStr = strjoin(noMatch,match)
```

```
combinedStr =
'She sells sea shells by the seashore.'
```

Capture Substrings of Matches Using Ordinal Tokens

Find the names of HTML tags by defining a token within a regular expression. Tokens are indicated with parentheses, `()`.

```
str = '<title>My Title</title><p>Here is some text.</p>';
expression = '<(\w+).*>.*</\1>';
[tokens,matches] = regexp(str,expression,'tokens','match');
```

The regular expression `<(\w+).*>.*</\1>` specifies this pattern:

- `<(\w+)` finds an opening angle bracket followed by one or more alphanumeric or underscore characters. Enclosing `\w+` in parentheses captures the name of the HTML tag in a token.
- `.*>` finds any number of additional characters, such as HTML attributes, and a closing angle bracket.
- `</\1>` finds the end tag corresponding to the first token (indicated by `\1`). The end tag has the form `</`*`tagname`*`>`.

View the tokens and matching substrings.

```
celldisp(tokens)
```

```
tokens{1}{1} =

title


tokens{2}{1} =

p
```

```
celldisp(matches)
```

```
matches{1} =

<title>My Title</title>
```

```
matches{2} =

<p>Here is some text.</p>
```

Capture Substrings of Matches Using Named Tokens

Parse dates that can appear with either the day or the month first, in these forms: `mm/dd/yyyy` or `dd-mm-yyyy`. Use named tokens to identify each part of the date.

```
str = '01/11/2000  20-02-2020  03/30/2000  16-04-2020';
expression = ['(?<month>\d+)/(?<day>\d+)/(?<year>\d+)|'...
              '(?<day>\d+)-(?<month>\d+)-(?<year>\d+)'];
tokenNames = regexp(str,expression,'names');
```

The regular expression specifies this pattern:

- `(?<name>\d+)` finds one or more numeric digits and assigns the result to the token indicated by name.
- `|` is the logical `or` operator, which indicates that there are two possible patterns for dates. In the first pattern, slashes (`/`) separate the tokens. In the second pattern, hyphens (`-`) separate the tokens.

View the named tokens.

```
for k = 1:length(tokenNames)
   disp(tokenNames(k))
end
```

```
    month: '01'
      day: '11'
     year: '2000'

    month: '02'
      day: '20'
     year: '2020'

    month: '03'
      day: '30'
     year: '2000'
```

```
    month: '04'
      day: '16'
     year: '2020'
```

Perform Case-Insensitive Matches

Find both uppercase and lowercase instances of a word.

By default, `regexp` performs case-sensitive matching.

```
str = 'A character vector with UPPERCASE and lowercase text.';
expression = '\w*case';
matchStr = regexp(str,expression,'match')
```

```
matchStr = 1x1 cell array
    {'lowercase'}
```

The regular expression specifies that the character vector:

- Begins with any number of alphanumeric or underscore characters, `\w*`.
- Ends with the literal text `case`.

The `regexpi` function uses the same syntax as `regexp`, but performs case-insensitive matching.

```
matchWithRegexpi = regexpi(str,expression,'match')
```

```
matchWithRegexpi = 1x2 cell array
    {'UPPERCASE'}    {'lowercase'}
```

Alternatively, disable case-sensitive matching for `regexp` using the `'ignorecase'` option.

```
matchWithIgnorecase = regexp(str,expression,'match','ignorecase')
```

```
matchWithIgnorecase = 1x2 cell array
    {'UPPERCASE'}    {'lowercase'}
```

For multiple expressions, disable case-sensitive matching for selected expressions using the `(?i)` search flag.

```
expression = {'(?-i)\w*case';...
              '(?i)\w*case'};
matchStr = regexp(str,expression,'match');
celldisp(matchStr)
```

```
matchStr{1}{1} =

lowercase


matchStr{2}{1} =

UPPERCASE


matchStr{2}{2} =

lowercase
```

Parse Text with Newline Characters

Create a character vector that contains a newline, `\n`, and parse it using a regular expression. Since `regexp` returns `matchStr` as a cell array containing text that has multiple lines, you can take the text out of the cell array to display all lines.

```
str = sprintf('abc\n de');
expression = '.*';
matchStr = regexp(str,expression,'match');
matchStr{:}
```

```
ans =
    'abc
      de'
```

By default, the dot (`.`) matches every character, including the newline, and returns a single match that is equivalent to the original character vector.

Exclude newline characters from the match using the `'dotexceptnewline'` option. This returns separate matches for each line of text.

```
matchStrNoNewline = regexp(str,expression,'match','dotexceptnewline')
```

```
matchStrNoNewline = 1x2 cell array
    {'abc'}    {' de'}
```

Find the first or last character of each line using the `^` or `$` metacharacters and the `'lineanchors'` option.

```
expression = '.$';
lastInLine = regexp(str,expression,'match','lineanchors')
```

```
lastInLine = 1x2 cell array
    {'c'}    {'e'}
```

Return Matches in Cell

Find matches within a piece of text and return the output in a scalar cell.

Find words that start with c, end with t, and contain one or more vowels between them. Return the starting indices in a scalar cell.

```
str = 'bat cat can car coat court CUT ct CAT-scan';
expression = 'c[aeiou]+t';
startIndex = regexp(str,expression,'forceCellOutput')
```

```
startIndex = 1x1 cell array
    {1x2 double}
```

To access the starting indices as a numeric array, index into the cell.

```
startIndex{1}
```

```
ans = 1×2

     5    17
```

Return the matching and nonmatching substrings. Each output is in its own scalar cell.

```
[match,noMatch] = regexp(str,expression,'match','split','forceCellOutput')
```

```
match = 1x1 cell array
    {1x2 cell}


noMatch = 1x1 cell array
    {1x3 cell}
```

To access the array of matches, index into `match`.

```
match{1}
```

```
ans = 1x2 cell array
    {'cat'}    {'coat'}
```

To access the substrings that do not match, index into `noMatch`.

```
noMatch{1}
```

```
ans = 1x3 cell array
    {'bat '}    {' can car '}    {' court CUT ct C...'}
```

Input Arguments

`str` — Input text

character vector | cell array of character vectors | string array

Input text, specified as a character vector, a cell array of character vectors, or a string array. Each character vector in a cell array, or each string in a string array, can be of any length and contain any characters.

If `str` and `expression` are string arrays or cell arrays, they must have the same dimensions.

Data Types: `string` | `char` | `cell`

`expression` — Regular expression
character vector | cell array of character vectors | string array

Regular expression, specified as a character vector, a cell array of character vectors, or a string array. Each expression can contain characters, metacharacters, operators, tokens, and flags that specify patterns to match in `str`.

The following tables describe the elements of regular expressions.

Metacharacters

Metacharacters represent letters, letter ranges, digits, and space characters. Use them to construct a generalized pattern of characters.

Metacharacter	Description	Example
`.`	Any single character, including white space	`'..ain'` matches sequences of five consecutive characters that end with `'ain'`.
`[`$c_1c_2c_3$`]`	Any character contained within the square brackets. The following characters are treated literally: `$ | . * + ?` and `-` when not used to indicate a range.	`'[rp.]ain'` matches `'rain'` or `'pain'` or `'.ain'`.
`[^`$c_1c_2c_3$`]`	Any character not contained within the square brackets. The following characters are treated literally: `$ | . * + ?` and `-` when not used to indicate a range.	`'[^*rp]ain'` matches all four-letter sequences that end in `'ain'`, except `'rain'` and `'pain'` and `'*ain'`. For example, it matches `'gain'`, `'lain'`, or `'vain'`.
`[`c_1`-`c_2`]`	Any character in the range of c_1 through c_2	`'[A-G]'` matches a single character in the range of `A` through `G`.
`\w`	Any alphabetic, numeric, or underscore character. For English character sets, `\w` is equivalent to `[a-zA-Z_0-9]`	`'\w*'` identifies a word.
`\W`	Any character that is not alphabetic, numeric, or underscore. For English character sets, `\W` is equivalent to `[^a-zA-Z_0-9]`	`'\W*'` identifies a term that is not a word.

Metacharacter	Description	Example
`\s`	Any white-space character; equivalent to `[ \f\n\r\t\v]`	`'\w*n\s'` matches words that end with the letter `n`, followed by a white-space character.
`\S`	Any non-white-space character; equivalent to `[^ \f\n\r\t\v]`	`'\d\S'` matches a numeric digit followed by any non-white-space character.
`\d`	Any numeric digit; equivalent to `[0-9]`	`'\d*'` matches any number of consecutive digits.
`\D`	Any nondigit character; equivalent to `[^0-9]`	`'\w*\D\>'` matches words that do not end with a numeric digit.
`\oN` or `\o{N}`	Character of octal value `N`	`'\o{40}'` matches the space character, defined by octal `40`.
`\xN` or `\x{N}`	Character of hexadecimal value `N`	`'\x2C'` matches the comma character, defined by hex `2C`.

Character Representation

Operator	Description
`\a`	Alarm (beep)
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\`*`char`*	Any character with special meaning in regular expressions that you want to match literally (for example, use `\\` to match a single backslash)

Quantifiers

Quantifiers specify the number of times a pattern must occur in the matching text.

Quantifier	Matches the expression when it occurs...	Example
`expr*`	0 or more times consecutively.	`'\w*'` matches a word of any length.
`expr?`	0 times or 1 time.	`'\w*(\.m)?'` matches words that optionally end with the extension `.m`.
`expr+`	1 or more times consecutively.	`'<img src="\w+\.gif">'` matches an `<img>` HTML tag when the file name contains one or more characters.
`expr{m,n}`	At least `m` times, but no more than `n` times consecutively. `{0,1}` is equivalent to `?`.	`'\S{4,8}'` matches between four and eight non-white-space characters.
`expr{m,}`	At least `m` times consecutively. `{0,}` and `{1,}` are equivalent to `*` and `+`, respectively.	`'<a href="\w{1,}\.html">'` matches an `<a>` HTML tag when the file name contains one or more characters.
`expr{n}`	Exactly `n` times consecutively. Equivalent to `{n,n}`.	`'\d{4}'` matches four consecutive digits.

Quantifiers can appear in three modes, described in the following table. *q* represents any of the quantifiers in the previous table.

Mode	Description	Example
`expr`*q*	Greedy expression: match as many characters as possible.	Given the text `'<tr><td><p>text</p></td>'`, the expression `'</?t.*>'` matches all characters between `<tr` and `/td>`: `'<tr><td><p>text</p></td>'`
`expr`*q*`?`	Lazy expression: match as few characters as necessary.	Given the text`'<tr><td><p>text</p></td>'`, the expression `'</?t.*?>'` ends each match at the first occurrence of the closing angle bracket (`>`): `'<tr>'  '<td>'  '</td>'`

Mode	Description	Example
`expr`*q*`+`	Possessive expression: match as much as possible, but do not rescan any portions of the text.	Given the text`'<tr><td><p>text</p></td>'`, the expression `'</?t.*+>'` does not return any matches, because the closing angle bracket is captured using `.*`, and is not rescanned.

Grouping Operators

Grouping operators allow you to capture tokens, apply one operator to multiple elements, or disable backtracking in a specific group.

Grouping Operator	Description	Example
`(expr)`	Group elements of the expression and capture tokens.	`'Joh?n\s(\w*)'` captures a token that contains the last name of any person with the first name `John` or `Jon`.
`(?:expr)`	Group, but do not capture tokens.	`'(?:[aeiou][^aeiou]){2}'` matches two consecutive patterns of a vowel followed by a nonvowel, such as `'anon'`. Without grouping, `'[aeiou][^aeiou]{2}'`matches a vowel followed by two nonvowels.
`(?>expr)`	Group atomically. Do not backtrack within the group to complete the match, and do not capture tokens.	`'A(?>.*)Z'` does not match `'AtoZ'`, although `'A(?:.*)Z'` does. Using the atomic group, `Z` is captured using `.*` and is not rescanned.
`(expr1|expr2)`	Match expression `expr1` or expression `expr2`. If there is a match with `expr1`, then `expr2` is ignored. You can include `?:` or `?>` after the opening parenthesis to suppress tokens or group atomically.	`'(let|tel)\w+'` matches words that start with `let` or `tel`.

Anchors

Anchors in the expression match the beginning or end of the input text or word.

Anchor	Matches the...	Example
`^expr`	Beginning of the input text.	`'^M\w*'` matches a word starting with `M` at the beginning of the text.
`expr$`	End of the input text.	`'\w*m$'` matches words ending with `m` at the end of the text.
`\<expr`	Beginning of a word.	`'\<n\w*'` matches any words starting with `n`.
`expr\>`	End of a word.	`'\w*e\>'` matches any words ending with `e`.

Lookaround Assertions

Lookaround assertions look for patterns that immediately precede or follow the intended match, but are not part of the match.

The pointer remains at the current location, and characters that correspond to the `test` expression are not captured or discarded. Therefore, lookahead assertions can match overlapping character groups.

Lookaround Assertion	Description	Example
`expr(?=test)`	Look ahead for characters that match `test`.	`'\w*(?=ing)'` matches terms that are followed by `ing`, such as `'Fly'` and `'fall'` in the input text `'Flying, not falling.'`
`expr(?!test)`	Look ahead for characters that do not match `test`.	`'i(?!ng)'` matches instances of the letter `i` that are not followed by `ng`.
`(?<=test)expr`	Look behind for characters that match `test`.	`'(?<=re)\w*'` matches terms that follow `'re'`, such as `'new'`, `'use'`, and `'cycle'` in the input text `'renew, reuse, recycle'`

Lookaround Assertion	Description	Example
`(?<!test)expr`	Look behind for characters that do not match `test`.	`'(?<!\d)(\d)(?!\d)'` matches single-digit numbers (digits that do not precede or follow other digits).

If you specify a lookahead assertion *before* an expression, the operation is equivalent to a logical `AND`.

Operation	Description	Example
`(?=test)expr`	Match both `test` and `expr`.	`'(?=[a-z])[^aeiou]'` matches consonants.
`(?!test)expr`	Match `expr` and do not match `test`.	`'(?![aeiou])[a-z]'` matches consonants.

Logical and Conditional Operators

Logical and conditional operators allow you to test the state of a given condition, and then use the outcome to determine which pattern, if any, to match next. These operators support logical `OR`, and `if` or `if/else` conditions.

Conditions can be tokens, lookaround operators, or dynamic expressions of the form `(?@cmd)`. Dynamic expressions must return a logical or numeric value.

Conditional Operator	Description	Example
`expr1|expr2`	Match expression `expr1` or expression `expr2`. If there is a match with `expr1`, then `expr2` is ignored.	`'(let|tel)\w+'` matches words that start with `let` or `tel`.
`(?(cond)expr)`	If condition `cond` is `true`, then match `expr`.	`'(?(?@ispc)[A-Z]:\\)'` matches a drive name, such as `C:\`, when run on a Windows system.
`(?(cond)expr1|expr2)`	If condition `cond` is `true`, then match `expr1`. Otherwise, match `expr2`.	`'Mr(s?)\..*?(?(1)her|his)\w*'` matches text that includes `her` when the text begins with `Mrs`, or that includes `his` when the text begins with `Mr`.

Token Operators

Tokens are portions of the matched text that you define by enclosing part of the regular expression in parentheses. You can refer to a token by its sequence in the text (an ordinal token), or assign names to tokens for easier code maintenance and readable output.

Ordinal Token Operator	Description	Example
`(expr)`	Capture in a token the characters that match the enclosed expression.	`'Joh?n\s(\w*)'` captures a token that contains the last name of any person with the first name `John` or `Jon`.
`\N`	Match the `N`th token.	`'<(\w+).*>.*</\1>'` captures tokens for HTML tags, such as `'title'` from the text `'<title>Some text</title>'`.
`(?(N)expr1|expr2)`	If the `N`th token is found, then match `expr1`. Otherwise, match `expr2`.	`'Mr(s?)\..*?(?(1)her|his) \w*'` matches text that includes `her` when the text begins with `Mrs`, or that includes `his` when the text begins with `Mr`.

Named Token Operator	Description	Example
`(?<name>expr)`	Capture in a named token the characters that match the enclosed expression.	`'(?<month>\d+)-(?<day>\d+)-(?<yr>\d+)'` creates named tokens for the month, day, and year in an input date of the form `mm-dd-yy`.
`\k<name>`	Match the token referred to by `name`.	`'<(?<tag>\w+).*>.*</\k<tag>>'` captures tokens for HTML tags, such as `'title'` from the text `'<title>Some text</title>'`.
`(?(name)expr1|expr2)`	If the named token is found, then match `expr1`. Otherwise, match `expr2`.	`'Mr(?<sex>s?)\..*?(?(sex)her|his) \w*'` matches text that includes `her` when the text begins with `Mrs`, or that includes `his` when the text begins with `Mr`.

Note If an expression has nested parentheses, MATLAB captures tokens that correspond to the outermost set of parentheses. For example, given the search pattern `'(and(y|rew))'`, MATLAB creates a token for `'andrew'` but not for `'y'` or `'rew'`.

Dynamic Regular Expressions

Dynamic expressions allow you to execute a MATLAB command or a regular expression to determine the text to match.

The parentheses that enclose dynamic expressions do *not* create a capturing group.

Operator	Description	Example
`(??expr)`	Parse `expr` and include the resulting term in the match expression. When parsed, `expr` must correspond to a complete, valid regular expression. Dynamic expressions that use the backslash escape character (`\`) require two backslashes: one for the initial parsing of `expr`, and one for the complete match.	`'^(\d+)((??\\w{$1}))'` determines how many characters to match by reading a digit at the beginning of the match. The dynamic expression is enclosed in a second set of parentheses so that the resulting match is captured in a token. For instance, matching `'5XXXXX'` captures tokens for `'5'` and `'XXXXX'`.
`(??@cmd)`	Execute the MATLAB command represented by `cmd`, and include the output returned by the command in the match expression.	`'(.{2,}).?(??@fliplr($1))'` finds palindromes that are at least four characters long, such as `'abba'`.
`(?@cmd)`	Execute the MATLAB command represented by `cmd`, but discard any output the command returns. (Helpful for diagnosing regular expressions.)	`'\w*?(\w)(?@disp($1))\1\w*'` matches words that include double letters (such as `pp`), and displays intermediate results.

Within dynamic expressions, use the following operators to define replacement text.

Replacement Operator	Description
`$&` or `$0`	Portion of the input text that is currently a match
`$` `	Portion of the input text that precedes the current match

Replacement Operator	Description
`$'`	Portion of the input text that follows the current match (use `$''` to represent `$'`)
`$N`	`N`th token
`$<name>`	Named token
`${cmd}`	Output returned when MATLAB executes the command, `cmd`

Comments

Characters	Description	Example
`(?#comment)`	Insert a comment in the regular expression. The comment text is ignored when matching the input.	`'(?# Initial digit)\<\d\w+'` includes a comment, and matches words that begin with a number.

Search Flags

Search flags modify the behavior for matching expressions. An alternative to using a search flag within an expression is to pass an `option` input argument.

Flag	Description
`(?-i)`	Match letter case (default for `regexp` and `regexprep`).
`(?i)`	Do not match letter case (default for `regexpi`).
`(?s)`	Match dot (`.`) in the pattern with any character (default).
`(?-s)`	Match dot in the pattern with any character that is not a newline character.
`(?-m)`	Match the `^` and `$` metacharacters at the beginning and end of text (default).
`(?m)`	Match the `^` and `$` metacharacters at the beginning and end of a line.
`(?-x)`	Include space characters and comments when matching (default).
`(?x)`	Ignore space characters and comments when matching. Use `'\ '` and `'\#'` to match space and `#` characters.

The expression that the flag modifies can appear either after the parentheses, such as

```
(?i)\w*
```

or inside the parentheses and separated from the flag with a colon (`:`), such as

```
(?i:\w*)
```

The latter syntax allows you to change the behavior for part of a larger expression.

Data Types: `char` | `cell` | `string`

`outkey` — Keyword that indicates which outputs to return
`'start'` (default) | `'end'` | `'tokenExtents'` | `'match'` | `'tokens'` | `'names'` | `'split'`

Keyword that indicates which outputs to return, specified as one of the following character vectors.

Output Keyword	Returns
`'start'` (default)	Starting indices of all matches, `startIndex`
`'end'`	Ending indices of all matches, `endIndex`
`'tokenExtents'`	Starting and ending indices of all tokens
`'match'`	Text of each substring that matches the pattern in `expression`
`'tokens'`	Text of each captured token in `str`
`'names'`	Name and text of each named token
`'split'`	Text of nonmatching substrings of `str`

Data Types: `char` | `string`

`option` — Search option
`'once'` | `'warnings'` | `'ignorecase'` | `'emptymatch'` | `'dotexceptnewline'` | `'lineanchors'` | ...

Search option, specified as a character vector. Options come in pairs: one option that corresponds to the default behavior, and one option that allows you to override the default. Specify only one option from a pair. Options can appear in any order.

Default	Override	Description
`'all'`	`'once'`	Match the expression as many times as possible (default), or only once.
`'nowarnings'`	`'warnings'`	Suppress warnings (default), or display them.

Default	Override	Description
`'matchcase'`	`'ignorecase'`	Match letter case (default), or ignore case.
`'noemptymatch'`	`'emptymatch'`	Ignore zero length matches (default), or include them.
`'dotall'`	`'dotexceptnewline'`	Match dot with any character (default), or all except newline (`\n`).
`'stringanchors'`	`'lineanchors'`	Apply `^` and `$` metacharacters to the beginning and end of a character vector (default), or to the beginning and end of a line. The newline character (`\n`) specifies the end of a line. The beginning of a line is specified as the first character, or any character that immediately follows a newline character.
`'literalspacing'`	`'freespacing'`	Include space characters and comments when matching (default), or ignore them. With `freespacing`, use `'\ '` and `'\#'` to match space and `#` characters.

Data Types: `char` | `string`

Output Arguments

`startIndex` — Starting index of each match
row vector | cell array of row vectors

Starting indices of each match, returned as a row vector or cell array, as follows:

- If `str` and `expression` are both character vectors or string scalars, the output is a row vector (or, if there are no matches, an empty array).
- If either `str` or `expression` is a cell array of character vectors or a string array, and the other is a character vector or a string scalar, the output is a cell array of row vectors. The output cell array has the same dimensions as the input array.
- If `str` and `expression` are both cell arrays or string arrays, they must have the same dimensions. The output is a cell array with the same dimensions.

`endIndex` — Ending index of each match
row vector | cell array of row vectors

Ending index of each match, returned as a row vector or cell array, as follows:

- If `str` and `expression` are both character vectors or string scalars, the output is a row vector (or, if there are no matches, an empty array).
- If either `str` or `expression` is a cell array of character vectors or a string array, and the other is a character vector or a string scalar, the output is a cell array of row vectors. The output cell array has the same dimensions as the input array.
- If `str` and `expression` are both cell arrays or string arrays, they must have the same dimensions. The output is a cell array with the same dimensions.

`out` — Information about matches

numeric array | cell array | string array | structure array

Information about matches, returned as a numeric, cell, string, or structure array. The information in the output depends upon the value you specify for `outkey`, as follows.

Output Keyword	Output Description	Output Type and Dimensions
`'start'`	Starting indices of matches	For both `'start'` and `'end'`: • If `str` and `expression` are both character vectors or string scalars, the output is a row vector (or, if there are no matches, an empty array). • If either `str` or `expression` is a cell array of character vectors or a string array, and the other is a character vector or a string scalar, the output is a cell array of row vectors. The output cell array has the same dimensions as the input array. • If `str` and `expression` are both cell arrays or string arrays, they must have the same dimensions. The output is a cell array with the same dimensions.
`'end'`	Ending indices of matches	

Output Keyword	Output Description	Output Type and Dimensions
`'tokenExtents'`	Starting and ending indices of all tokens	By default, when returning all matches: • If `str` and `expression` are both character vectors or string scalars, the output is a 1-by-`n` cell array, where `n` is the number of matches. Each cell contains an `m`-by-2 numeric array of indices, where `m` is the number of tokens in the match. • If either `str` or `expression` is a cell array of character vectors or a string array, the output is a cell array with the same dimensions as the input array. Each cell contains a 1-by-`n` cell array, where each inner cell contains an `m`-by-2 numeric array. • If `str` and `expression` are both cell arrays or string arrays, they must have the same dimensions. The output is a cell array with the same dimensions. When you specify the `'once'` option to return only one match, the output is either an `m`-by-2 numeric array or a cell array with the same dimensions as `str` and/or `expression`. If a token is expected at a particular index `N`, but is not found, then MATLAB returns extents for that token of `[N,N-1]`.

Output Keyword	Output Description	Output Type and Dimensions
`'match'`	Text of each substring that matches the pattern in `expression`	By default, when returning all matches: • If `str` and `expression` are both character vectors or string scalars, the output is a 1-by-`n` array, where `n` is the number of matches. If `str` is a character vector, then the output is a cell array of character vectors. If `str` is a string scalar, then the output is a string array. • If either `str` or `expression` is a cell array of character vectors or a string array, the output is a cell array with the same dimensions as the input array. If `str` is a cell array of character vectors, then so is the output. If `str` is a string array, then the output is a cell array in which each cell contains a string array. • If `str` and `expression` are both cell arrays or string arrays, they must have the same dimensions. The output is a cell array with the same dimensions. If `str` is a cell array of character vectors, then so is the output. If `str` is a string array, then the output is a cell array in which each cell contains a string array. When you specify the `'once'` option to return only one match, the output is either a character vector, a string array, or a cell array with the same dimensions as `str` and `expression`.

Output Keyword	Output Description	Output Type and Dimensions
`'tokens'`	Text of each captured token in `str`	By default, when returning all matches: • If `str` and `expression` are both character vectors or string scalars, the output is a 1-by-`n` cell array, where `n` is the number of matches. Each cell contains a 1-by-`m` cell array of matches, where `m` is the number of tokens in the match. If `str` is a character vector, then the output is a cell array of character vectors. If `str` is a string array, then the output is a cell array in which each cell contains a string array. • If either `str` or `expression` is a cell array of character vectors or a string array, the output is a cell array with the same dimensions as the input array. Each cell contains a 1-by-`n` cell array, where each inner cell contains a 1-by-`m` array. If `str` is a cell array of character vectors, then each inner cell contains a 1-by-`m` cell array. If `str` is a string array, then each inner cell contains a 1-by-`m` string array. • If `str` and `expression` are both cell arrays or string arrays, they must have the same dimensions. The output is a cell array with the same dimensions. If `str` is a cell array of character vectors, then so is the output. If `str` is a string array, then the output is a cell array in which the innermost cells contain string arrays. When you specify the `'once'` option to return only one match, the output is a 1-by-`m` string array, cell array of character vectors, or a cell array that has the same dimensions as `str` and/or `expression`.

Output Keyword	Output Description	Output Type and Dimensions
		If a token is expected at a particular index, but is not found, then MATLAB returns an empty value for the token, `''` for character vectors, or `""` for strings.
`'names'`	Name and text of each named token	For all matches: • If `str` and `expression` are both character vectors or string scalars, the output is a 1-by-`n` structure array, where `n` is the number of matches. The structure field names correspond to the token names. • If `str` or `expression` is a cell array of character vectors or a string array, the output is a cell array with the same dimensions as the input array. Each cell contains a 1-by-`n` structure array. • If `str` and `expression` are both cell arrays or string arrays, they must have the same dimensions. The output is a cell array with the same dimensions.

Output Keyword	Output Description	Output Type and Dimensions
`'split'`	Text of nonmatching substrings of `str`	For all matches: • If `str` and `expression` are both character vectors or string scalars, the output is a 1-by-`n` array, where `n` is the number of nonmatches. If `str` is a character vector, then the output is a cell array of character vectors. If `str` is a string scalar, then the output is a string array. • If either `str` or `expression` is a cell array of character vectors or a string array, the output is a cell array with the same dimensions as the input array. Each cell contains a 1-by-`n` cell array of character vectors. If `str` is a cell array of character vectors, then so is the output. If `str` is a string array, then the output is a cell array in which each cell contains a string array. • If `str` and `expression` are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions. If `str` is a cell array of character vectors, then so is the output. If `str` is a string array, then the output is a cell array in which each cell contains a string array.

More About

Tokens

Tokens are portions of the matched text that correspond to portions of the regular expression. To create tokens, enclose part of the regular expression in parentheses.

For example, this expression finds a date of the form `dd-mmm-yyyy`, including tokens for the day, month, and year.

```
str = 'Here is a date: 01-Apr-2020';
expression = '(\d+)-(\w+)-(\d+)';

mydate = regexp(str,expression,'tokens');
mydate{:}
```

```
ans =
    '01'    'April'    '2020'
```

You can associate names with tokens so that they are more easily identifiable:

```
str = 'Here is a date: 01-Apr-2020';
expression = '(?<day>\d+)-(?<month>\w+)-(?<year>\d+)';

mydate = regexp(str,expression,'names')
```

```
mydate =
      day: '01'
    month: 'Apr'
     year: '2020'
```

For more information, see “Tokens in Regular Expressions”.

Tips

- Use `contains` or `strfind` to find an exact character match within text. Use `regexp` to look for a pattern of characters.

Algorithms

MATLAB parses each input character vector or string from left to right, attempting to match the text in the character vector or string with the first element of the regular expression. During this process, MATLAB skips over any text that does not match.

When MATLAB finds the first match, it continues parsing to match the second piece of the expression, and so on.

See Also

`contains` | `regexpi` | `regexprep` | `regexptranslate` | `replace` | `strfind` | `strjoin` | `strrep` | `strsplit`

Topics

"Regular Expressions"
"Lookahead Assertions in Regular Expressions"
"Dynamic Regular Expressions"

Introduced before R2006a

regexpi

Match regular expression (case insensitive)

Syntax

```
startIndex = regexpi(str,expression)
[startIndex,endIndex] = regexpi(str,expression)
```

```
out = regexpi(str,expression,outkey)
[out1,...,outN] = regexpi(str,expression,outkey1,...,outkeyN)
```

```
___ = regexpi(___,option1,...,optionM)
```

```
___ = regexpi(___,'forceCellOutput')
```

Description

`startIndex = regexpi(str,expression)` returns the starting index of each substring of `str` that matches the character patterns specified by the regular expression, without regard to letter case. If there are no matches, `startIndex` is an empty array.

`[startIndex,endIndex] = regexpi(str,expression)` returns the starting and ending indices of all matches.

`out = regexpi(str,expression,outkey)` returns the output specified by `outkey`. For example, if `outkey` is `'match'`, then `regexpi` returns the substrings that match the expression rather than their starting indices.

`[out1,...,outN] = regexpi(str,expression,outkey1,...,outkeyN)` returns the outputs specified by multiple output keywords, in the specified order. For example, if you specify `'match'`,`'tokens'`, then `regexpi` returns substrings that match the entire expression and tokens on page 1-13272 that match parts of the expression.

`___ = regexpi(___,option1,...,optionM)` modifies the search using the specified option flags. For example, specify `'matchcase'` to perform a case-sensitive match. You can include any of the inputs and request any of the outputs from previous syntaxes.

`___ = regexpi(___,'forceCellOutput')` returns each output argument as a scalar cell. The cells contain the numeric arrays or substrings that are described as the outputs of the previous syntaxes. You can include any of the inputs and request any of the outputs from previous syntaxes.

Examples

Pattern Matching

Find words that start with `c`, end with `t`, and contain one or more vowels between them.

```
str = 'bat cat can car COAT court cut ct CAT-scan';
expression = 'c[aeiou]+t';
startIndex = regexpi(str,expression)
```

```
startIndex = 1×4

     5    17    28    35
```

Values in `startIndex` indicate the index of the first character of each word that matches the regular expression.

The regular expression `'c[aeiou]+t'` specifies this pattern:

- `c` must be the first character.
- `c` must be followed by one of the characters inside the brackets, `[aeiou]`.
- The bracketed pattern must occur one or more times, as indicated by the `+` operator.
- `t` must be the last character, with no characters between the bracketed pattern and the `t`.

Case-Sensitive Match

Match letter case in all or part of an expression.

By default, `regexpi` performs case-insensitive matching.

```
str = 'A character vector with UPPERCASE and lowercase text.';
expression = '\w*case';
matchStr = regexpi(str,expression,'match')
```

```
matchStr = 1x2 cell array
    {'UPPERCASE'}    {'lowercase'}
```

Use the `regexp` function with the same syntax as `regexpi` to perform case-sensitive matching.

```
matchWithRegexp = regexp(str,expression,'match')
```

```
matchWithRegexp = 1x1 cell array
    {'lowercase'}
```

To disable case-sensitive matching for `regexp`, use the `'ignorecase'` option.

```
matchWithIgnorecase = regexp(str,expression,'match','ignorecase')
```

```
matchWithIgnorecase = 1x2 cell array
    {'UPPERCASE'}    {'lowercase'}
```

For multiple expressions, enable and disable case-insensitive matching for selected expressions using the `(?i)` and `(?-i)` search flags.

```
expression = {'(?-i)\w*case';...
              '(?i)\w*case'};
matchStr = regexp(str,expression,'match');
celldisp(matchStr)
```

```
matchStr{1}{1} =

lowercase


matchStr{2}{1} =

UPPERCASE


matchStr{2}{2} =
```

lowercase

Input Arguments

str — Input text

character vector | cell array of character vectors | string array

Input text, specified as a character vector, a cell array of character vectors, or a string array. Each character vector in a cell array, or each string in a string array, can be of any length and contain any characters.

If `str` and `expression` are string arrays or cell arrays, they must have the same dimensions.

Data Types: `string` | `char` | `cell`

expression — Regular expression

character vector | cell array of character vectors | string array

Regular expression, specified as a character vector, a cell array of character vectors, or a string array. Each expression can contain characters, metacharacters, operators, tokens, and flags that specify patterns to match in `str`.

The following tables describe the elements of regular expressions.

Metacharacters

Metacharacters represent letters, letter ranges, digits, and space characters. Use them to construct a generalized pattern of characters.

Metacharacter	Description	Example
`.`	Any single character, including white space	`'..ain'` matches sequences of five consecutive characters that end with `'ain'`.

Metacharacter	Description	Example
[$c_1c_2c_3$]	Any character contained within the square brackets. The following characters are treated literally: `$ | . * + ?` and `-` when not used to indicate a range.	`'[rp.]ain'` matches `'rain'` or `'pain'` or `'.ain'`.
[^$c_1c_2c_3$]	Any character not contained within the square brackets. The following characters are treated literally: `$ | . * + ?` and `-` when not used to indicate a range.	`'[^*rp]ain'` matches all four-letter sequences that end in `'ain'`, except `'rain'` and `'pain'` and `'*ain'`. For example, it matches `'gain'`, `'lain'`, or `'vain'`.
[c_1-c_2]	Any character in the range of c_1 through c_2	`'[A-G]'` matches a single character in the range of `A` through `G`.
`\w`	Any alphabetic, numeric, or underscore character. For English character sets, `\w` is equivalent to `[a-zA-Z_0-9]`	`'\w*'` identifies a word.
`\W`	Any character that is not alphabetic, numeric, or underscore. For English character sets, `\W` is equivalent to `[^a-zA-Z_0-9]`	`'\W*'` identifies a term that is not a word.
`\s`	Any white-space character; equivalent to `[ \f\n\r\t\v]`	`'\w*n\s'` matches words that end with the letter `n`, followed by a white-space character.
`\S`	Any non-white-space character; equivalent to `[^ \f\n\r\t\v]`	`'\d\S'` matches a numeric digit followed by any non-white-space character.
`\d`	Any numeric digit; equivalent to `[0-9]`	`'\d*'` matches any number of consecutive digits.
`\D`	Any nondigit character; equivalent to `[^0-9]`	`'\w*\D\>'` matches words that do not end with a numeric digit.
`\oN` or `\o{N}`	Character of octal value `N`	`'\o{40}'` matches the space character, defined by octal `40`.
`\xN` or `\x{N}`	Character of hexadecimal value `N`	`'\x2C'` matches the comma character, defined by hex `2C`.

Character Representation

Operator	Description
`\a`	Alarm (beep)
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\`*`char`*	Any character with special meaning in regular expressions that you want to match literally (for example, use `\\` to match a single backslash)

Quantifiers

Quantifiers specify the number of times a pattern must occur in the matching text.

Quantifier	Matches the expression when it occurs...	Example
`expr*`	0 or more times consecutively.	`'\w*'` matches a word of any length.
`expr?`	0 times or 1 time.	`'\w*(\.m)?'` matches words that optionally end with the extension `.m`.
`expr+`	1 or more times consecutively.	`'<img src="\w+\.gif">'` matches an `<img>` HTML tag when the file name contains one or more characters.
`expr{m,n}`	At least `m` times, but no more than `n` times consecutively. `{0,1}` is equivalent to `?`.	`'\S{4,8}'` matches between four and eight non-white-space characters.
`expr{m,}`	At least `m` times consecutively. `{0,}` and `{1,}` are equivalent to `*` and `+`, respectively.	`'<a href="\w{1,}\.html">'` matches an `<a>` HTML tag when the file name contains one or more characters.
`expr{n}`	Exactly `n` times consecutively. Equivalent to `{n,n}`.	`'\d{4}'` matches four consecutive digits.

Quantifiers can appear in three modes, described in the following table. *q* represents any of the quantifiers in the previous table.

Mode	Description	Example
`expr`*q*	Greedy expression: match as many characters as possible.	Given the text `'<tr><td><p>text</p></td>'`, the expression `'</?t.*>'` matches all characters between `<tr` and `/td>`: `'<tr><td><p>text</p></td>'`
`expr`*q*`?`	Lazy expression: match as few characters as necessary.	Given the text`'<tr><td><p>text</p></td>'`, the expression `'</?t.*?>'` ends each match at the first occurrence of the closing angle bracket (`>`): `'<tr>'  '<td>'  '</td>'`
`expr`*q*`+`	Possessive expression: match as much as possible, but do not rescan any portions of the text.	Given the text`'<tr><td><p>text</p></td>'`, the expression `'</?t.*+>'` does not return any matches, because the closing angle bracket is captured using `.*`, and is not rescanned.

Grouping Operators

Grouping operators allow you to capture tokens, apply one operator to multiple elements, or disable backtracking in a specific group.

Grouping Operator	Description	Example
`(expr)`	Group elements of the expression and capture tokens.	`'Joh?n\s(\w*)'` captures a token that contains the last name of any person with the first name `John` or `Jon`.

Grouping Operator	Description	Example
`(?:expr)`	Group, but do not capture tokens.	`'(?:[aeiou][^aeiou]){2}'` matches two consecutive patterns of a vowel followed by a nonvowel, such as `'anon'`. Without grouping, `'[aeiou][^aeiou]{2}'`matches a vowel followed by two nonvowels.
`(?>expr)`	Group atomically. Do not backtrack within the group to complete the match, and do not capture tokens.	`'A(?>.*)Z'` does not match `'AtoZ'`, although `'A(?:.*)Z'` does. Using the atomic group, `Z` is captured using `.*` and is not rescanned.
`(expr1|expr2)`	Match expression `expr1` or expression `expr2`. If there is a match with `expr1`, then `expr2` is ignored. You can include `?:` or `?>` after the opening parenthesis to suppress tokens or group atomically.	`'(let|tel)\w+'` matches words that start with `let` or `tel`.

Anchors

Anchors in the expression match the beginning or end of the input text or word.

Anchor	Matches the...	Example
`^expr`	Beginning of the input text.	`'^M\w*'` matches a word starting with `M` at the beginning of the text.
`expr$`	End of the input text.	`'\w*m$'` matches words ending with `m` at the end of the text.
`\<expr`	Beginning of a word.	`'\<n\w*'` matches any words starting with `n`.
`expr\>`	End of a word.	`'\w*e\>'` matches any words ending with `e`.

Lookaround Assertions

Lookaround assertions look for patterns that immediately precede or follow the intended match, but are not part of the match.

The pointer remains at the current location, and characters that correspond to the `test` expression are not captured or discarded. Therefore, lookahead assertions can match overlapping character groups.

Lookaround Assertion	Description	Example
`expr(?=test)`	Look ahead for characters that match `test`.	`'\w*(?=ing)'` matches terms that are followed by `ing`, such as `'Fly'` and `'fall'` in the input text `'Flying, not falling.'`
`expr(?!test)`	Look ahead for characters that do not match `test`.	`'i(?!ng)'` matches instances of the letter `i` that are not followed by `ng`.
`(?<=test)expr`	Look behind for characters that match `test`.	`'(?<=re)\w*'` matches terms that follow `'re'`, such as `'new'`, `'use'`, and `'cycle'` in the input text `'renew, reuse, recycle'`
`(?<!test)expr`	Look behind for characters that do not match `test`.	`'(?<!\d)(\d)(?!\d)'` matches single-digit numbers (digits that do not precede or follow other digits).

If you specify a lookahead assertion *before* an expression, the operation is equivalent to a logical `AND`.

Operation	Description	Example
`(?=test)expr`	Match both `test` and `expr`.	`'(?=[a-z])[^aeiou]'` matches consonants.
`(?!test)expr`	Match `expr` and do not match `test`.	`'(?![aeiou])[a-z]'` matches consonants.

Logical and Conditional Operators

Logical and conditional operators allow you to test the state of a given condition, and then use the outcome to determine which pattern, if any, to match next. These operators support logical `OR`, and `if` or `if/else` conditions.

Conditions can be tokens, lookaround operators, or dynamic expressions of the form `(?@cmd)`. Dynamic expressions must return a logical or numeric value.

Conditional Operator	Description	Example
`expr1|expr2`	Match expression `expr1` or expression `expr2`. If there is a match with `expr1`, then `expr2` is ignored.	`'(let|tel)\w+'` matches words that start with `let` or `tel`.
`(?(cond)expr)`	If condition `cond` is `true`, then match `expr`.	`'(?(?@ispc)[A-Z]:\\)'` matches a drive name, such as `C:\`, when run on a Windows system.
`(?(cond)expr1|expr2)`	If condition `cond` is `true`, then match `expr1`. Otherwise, match `expr2`.	`'Mr(s?)\..*?(?(1)her|his)\w*'` matches text that includes `her` when the text begins with `Mrs`, or that includes `his` when the text begins with `Mr`.

Token Operators

Tokens are portions of the matched text that you define by enclosing part of the regular expression in parentheses. You can refer to a token by its sequence in the text (an ordinal token), or assign names to tokens for easier code maintenance and readable output.

Ordinal Token Operator	Description	Example
`(expr)`	Capture in a token the characters that match the enclosed expression.	`'Joh?n\s(\w*)'` captures a token that contains the last name of any person with the first name `John` or `Jon`.
`\N`	Match the `N`th token.	`'<(\w+).*>.*</\1>'` captures tokens for HTML tags, such as `'title'` from the text `'<title>Some text</title>'`.
`(?(N)expr1|expr2)`	If the `N`th token is found, then match `expr1`. Otherwise, match `expr2`.	`'Mr(s?)\..*?(?(1)her|his)\w*'` matches text that includes `her` when the text begins with `Mrs`, or that includes `his` when the text begins with `Mr`.

Named Token Operator	Description	Example
`(?<name>expr)`	Capture in a named token the characters that match the enclosed expression.	`'(?<month>\d+)-(?<day>\d+)-(?<yr>\d+)'` creates named tokens for the month, day, and year in an input date of the form `mm-dd-yy`.
`\k<name>`	Match the token referred to by `name`.	`'<(?<tag>\w+).*>.*</\k<tag>>'` captures tokens for HTML tags, such as `'title'` from the text `'<title>Some text</title>'`.
`(?(name)expr1|expr2)`	If the named token is found, then match `expr1`. Otherwise, match `expr2`.	`'Mr(?<sex>s?)\..*?(?(sex)her|his) \w*'` matches text that includes `her` when the text begins with `Mrs`, or that includes `his` when the text begins with `Mr`.

Note If an expression has nested parentheses, MATLAB captures tokens that correspond to the outermost set of parentheses. For example, given the search pattern `'(and(y|rew))'`, MATLAB creates a token for `'andrew'` but not for `'y'` or `'rew'`.

Dynamic Regular Expressions

Dynamic expressions allow you to execute a MATLAB command or a regular expression to determine the text to match.

The parentheses that enclose dynamic expressions do *not* create a capturing group.

Operator	Description	Example
`(??expr)`	Parse `expr` and include the resulting term in the match expression. When parsed, `expr` must correspond to a complete, valid regular expression. Dynamic expressions that use the backslash escape character (`\`) require two backslashes: one for the initial parsing of `expr`, and one for the complete match.	`'^(\d+)((??\\w{$1}))'` determines how many characters to match by reading a digit at the beginning of the match. The dynamic expression is enclosed in a second set of parentheses so that the resulting match is captured in a token. For instance, matching `'5XXXXX'` captures tokens for `'5'` and `'XXXXX'`.
`(??@cmd)`	Execute the MATLAB command represented by `cmd`, and include the output returned by the command in the match expression.	`'(.{2,}).?(??@fliplr($1))'` finds palindromes that are at least four characters long, such as `'abba'`.
`(?@cmd)`	Execute the MATLAB command represented by `cmd`, but discard any output the command returns. (Helpful for diagnosing regular expressions.)	`'\w*?(\w)(?@disp($1))\1\w*'` matches words that include double letters (such as `pp`), and displays intermediate results.

Within dynamic expressions, use the following operators to define replacement text.

Replacement Operator	Description
`$&` or `$0`	Portion of the input text that is currently a match
`$\``	Portion of the input text that precedes the current match
`$'`	Portion of the input text that follows the current match (use `$''` to represent `$'`)
`$N`	`N`th token
`$<name>`	Named token
`${cmd}`	Output returned when MATLAB executes the command, `cmd`

Comments

Characters	Description	Example
`(?#comment)`	Insert a comment in the regular expression. The comment text is ignored when matching the input.	`'(?# Initial digit)\<\d\w+'` includes a comment, and matches words that begin with a number.

Search Flags

Search flags modify the behavior for matching expressions. An alternative to using a search flag within an expression is to pass an `option` input argument.

Flag	Description
`(?-i)`	Match letter case (default for `regexp` and `regexprep`).
`(?i)`	Do not match letter case (default for `regexpi`).
`(?s)`	Match dot (`.`) in the pattern with any character (default).
`(?-s)`	Match dot in the pattern with any character that is not a newline character.
`(?-m)`	Match the `^` and `$` metacharacters at the beginning and end of text (default).
`(?m)`	Match the `^` and `$` metacharacters at the beginning and end of a line.
`(?-x)`	Include space characters and comments when matching (default).
`(?x)`	Ignore space characters and comments when matching. Use `'\ '` and `'\#'` to match space and # characters.

The expression that the flag modifies can appear either after the parentheses, such as

```
(?i)\w*
```

or inside the parentheses and separated from the flag with a colon (`:`), such as

```
(?i:\w*)
```

The latter syntax allows you to change the behavior for part of a larger expression.

Data Types: `char` | `cell` | `string`

outkey — Keyword that indicates which outputs to return
`'start'` (default) | `'end'` | `'tokenExtents'` | `'match'` | `'tokens'` | `'names'` | `'split'`

Keyword that indicates which outputs to return, specified as one of the following character vectors.

Output Keyword	Returns
`'start'` (default)	Starting indices of all matches, `startIndex`
`'end'`	Ending indices of all matches, `endIndex`
`'tokenExtents'`	Starting and ending indices of all tokens
`'match'`	Text of each substring that matches the pattern in `expression`
`'tokens'`	Text of each captured token in `str`
`'names'`	Name and text of each named token
`'split'`	Text of nonmatching substrings of `str`

Data Types: `char` | `string`

`option` — Search option
`'once'` | `'warnings'` | `'matchcase'` | `'emptymatch'` | `'dotexceptnewline'` | `'lineanchors'` | ...

Search option, specified as a character vector. Options come in pairs: one option that corresponds to the default behavior, and one option that allows you to override the default. Specify only one option from a pair. Options can appear in any order.

Default	Override	Description
`'all'`	`'once'`	Match the expression as many times as possible (default), or only once.
`'nowarnings'`	`'warnings'`	Suppress warnings (default), or display them.
`'ignorecase'`	`'matchcase'`	Ignore letter case (default), or match case.
`'noemptymatch'`	`'emptymatch'`	Ignore zero length matches (default), or include them.
`'dotall'`	`'dotexceptnewline'`	Match dot with any character (default), or all except newline (`\n`).

Default	Override	Description
`'stringanchors'`	`'lineanchors'`	Apply ^ and $ metacharacters to the beginning and end of a character vector (default), or to the beginning and end of a line. The newline character (`\n`) specifies the end of a line. The beginning of a line is specified as the first character, or any character that immediately follows a newline character.
`'literalspacing'`	`'freespacing'`	Include space characters and comments when matching (default), or ignore them. With `freespacing`, use `'\ '` and `'\#'` to match space and # characters.

Data Types: `char`

Output Arguments

`startIndex` — Starting index of each match

row vector | cell array of row vectors

Starting indices of each match, returned as a row vector or cell array, as follows:

- If `str` and `expression` are both character vectors or string scalars, the output is a row vector (or, if there are no matches, an empty array).
- If either `str` or `expression` is a cell array of character vectors or a string array, and the other is a character vector or a string scalar, the output is a cell array of row vectors. The output cell array has the same dimensions as the input array.
- If `str` and `expression` are both cell arrays or string arrays, they must have the same dimensions. The output is a cell array with the same dimensions.

`endIndex` — Ending index of each match

row vector | cell array of row vectors

Ending index of each match, returned as a row vector or cell array, as follows:

- If `str` and `expression` are both character vectors or string scalars, the output is a row vector (or, if there are no matches, an empty array).

- If either `str` or `expression` is a cell array of character vectors or a string array, and the other is a character vector or a string scalar, the output is a cell array of row vectors. The output cell array has the same dimensions as the input array.
- If `str` and `expression` are both cell arrays or string arrays, they must have the same dimensions. The output is a cell array with the same dimensions.

`out` — Information about matches

numeric array | cell array | string array | structure array

Information about matches, returned as a numeric, cell, string, or structure array. The information in the output depends upon the value you specify for `outkey`, as follows.

Output Keyword	Output Description	Output Type and Dimensions
`'start'`	Starting indices of matches	For both `'start'` and `'end'`: • If `str` and `expression` are both character vectors or string scalars, the output is a row vector (or, if there are no matches, an empty array). • If either `str` or `expression` is a cell array of character vectors or a string array, and the other is a character vector or a string scalar, the output is a cell array of row vectors. The output cell array has the same dimensions as the input array. • If `str` and `expression` are both cell arrays or string arrays, they must have the same dimensions. The output is a cell array with the same dimensions.
`'end'`	Ending indices of matches	

Output Keyword	Output Description	Output Type and Dimensions
`'tokenExtents'`	Starting and ending indices of all tokens	By default, when returning all matches: • If `str` and `expression` are both character vectors or string scalars, the output is a 1-by-`n` cell array, where `n` is the number of matches. Each cell contains an `m`-by-2 numeric array of indices, where `m` is the number of tokens in the match. • If either `str` or `expression` is a cell array of character vectors or a string array, the output is a cell array with the same dimensions as the input array. Each cell contains a 1-by-`n` cell array, where each inner cell contains an `m`-by-2 numeric array. • If `str` and `expression` are both cell arrays or string arrays, they must have the same dimensions. The output is a cell array with the same dimensions. When you specify the `'once'` option to return only one match, the output is either an `m`-by-2 numeric array or a cell array with the same dimensions as `str` and/or `expression`. If a token is expected at a particular index `N`, but is not found, then MATLAB returns extents for that token of `[N,N-1]`.

Output Keyword	Output Description	Output Type and Dimensions
`'match'`	Text of each substring that matches the pattern in `expression`	By default, when returning all matches: • If `str` and `expression` are both character vectors or string scalars, the output is a 1-by-`n` array, where `n` is the number of matches. If `str` is a character vector, then the output is a cell array of character vectors. If `str` is a string scalar, then the output is a string array. • If either `str` or `expression` is a cell array of character vectors or a string array, the output is a cell array with the same dimensions as the input array. If `str` is a cell array of character vectors, then so is the output. If `str` is a string array, then the output is a cell array in which each cell contains a string array. • If `str` and `expression` are both cell arrays or string arrays, they must have the same dimensions. The output is a cell array with the same dimensions. If `str` is a cell array of character vectors, then so is the output. If `str` is a string array, then the output is a cell array in which each cell contains a string array. When you specify the `'once'` option to return only one match, the output is either a character vector, a string array, or a cell array with the same dimensions as `str` and `expression`.

Output Keyword	Output Description	Output Type and Dimensions
`'tokens'`	Text of each captured token in `str`	By default, when returning all matches: • If `str` and `expression` are both character vectors or string scalars, the output is a 1-by-`n` cell array, where `n` is the number of matches. Each cell contains a 1-by-`m` cell array of matches, where `m` is the number of tokens in the match. If `str` is a character vector, then the output is a cell array of character vectors. If `str` is a string array, then the output is a cell array in which each cell contains a string array. • If either `str` or `expression` is a cell array of character vectors or a string array, the output is a cell array with the same dimensions as the input array. Each cell contains a 1-by-`n` cell array, where each inner cell contains a 1-by-`m` array. If `str` is a cell array of character vectors, then each inner cell contains a 1-by-`m` cell array. If `str` is a string array, then each inner cell contains a 1-by-`m` string array. • If `str` and `expression` are both cell arrays or string arrays, they must have the same dimensions. The output is a cell array with the same dimensions. If `str` is a cell array of character vectors, then so is the output. If `str` is a string array, then the output is a cell array in which the innermost cells contain string arrays. When you specify the `'once'` option to return only one match, the output is a 1-by-`m` string array, cell array of character vectors, or a cell array that has the same dimensions as `str` and/or `expression`.

Output Keyword	Output Description	Output Type and Dimensions
		If a token is expected at a particular index, but is not found, then MATLAB returns an empty value for the token, `''` for character vectors, or `""` for strings.
`'names'`	Name and text of each named token	For all matches: • If `str` and `expression` are both character vectors or string scalars, the output is a 1-by-`n` structure array, where `n` is the number of matches. The structure field names correspond to the token names. • If `str` or `expression` is a cell array of character vectors or a string array, the output is a cell array with the same dimensions as the input array. Each cell contains a 1-by-`n` structure array. • If `str` and `expression` are both cell arrays or string arrays, they must have the same dimensions. The output is a cell array with the same dimensions.

Output Keyword	Output Description	Output Type and Dimensions
`'split'`	Text of nonmatching substrings of `str`	For all matches: • If `str` and `expression` are both character vectors or string scalars, the output is a 1-by-`n` array, where `n` is the number of nonmatches. If `str` is a character vector, then the output is a cell array of character vectors. If `str` is a string scalar, then the output is a string array. • If either `str` or `expression` is a cell array of character vectors or a string array, the output is a cell array with the same dimensions as the input array. Each cell contains a 1-by-`n` cell array of character vectors. If `str` is a cell array of character vectors, then so is the output. If `str` is a string array, then the output is a cell array in which each cell contains a string array. • If `str` and `expression` are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions. If `str` is a cell array of character vectors, then so is the output. If `str` is a string array, then the output is a cell array in which each cell contains a string array.

More About

Tokens

Tokens are portions of the matched text that correspond to portions of the regular expression. To create tokens, enclose part of the regular expression in parentheses.

For example, this expression finds a date of the form `dd-mmm-yyyy`, including tokens for the day, month, and year.

```
str = 'Here is a date: 01-Apr-2020';
expression = '(\d+)-(\w+)-(\d+)';

mydate = regexp(str,expression,'tokens');
mydate{:}
```

```
ans =
    '01'    'April'    '2020'
```

You can associate names with tokens so that they are more easily identifiable:

```
str = 'Here is a date: 01-Apr-2020';
expression = '(?<day>\d+)-(?<month>\w+)-(?<year>\d+)';

mydate = regexp(str,expression,'names')
```

```
mydate =
      day: '01'
    month: 'Apr'
     year: '2020'
```

For more information, see “Tokens in Regular Expressions”.

See Also

`contains` | `regexp` | `regexprep` | `regexptranslate` | `replace` | `strfind` | `strjoin` | `strrep` | `strsplit`

Topics

“Lookahead Assertions in Regular Expressions”
“Dynamic Regular Expressions”

Introduced before R2006a

regexprep

Replace text using regular expression

Syntax

```
newStr = regexprep(str,expression,replace)
newStr = regexprep(str,expression,replace,option1,...optionM)
```

Description

`newStr = regexprep(str,expression,replace)` replaces the text in `str` that matches `expression` with the text described by `replace`. The `regexprep` function returns the updated text in `newStr`.

- If `str` is a single piece of text (either a character vector or a string scalar), then `newStr` is also a single piece of text of the same type. `newStr` is a single piece of text even when `expression` or `replace` is a cell array of character vectors or a string array. When `expression` is a cell array or a string array, `regexprep` applies the first expression to `str`, and then applies each subsequent expression to the preceding result.
- If `str` is a cell array or a string array, then `newStr` is a cell array or string array with the same dimensions as `str`. For each element of `str`, the `regexprep` function applies each expression in sequence.
- If there are no matches to `expression`, then `newStr` is equivalent to `str`.

`newStr = regexprep(str,expression,replace,option1,...optionM)` modifies the search using the specified options. For example, specify `'ignorecase'` to perform a case-insensitive match.

Examples

Update Text

Replace words that begin with `M`, end with `y`, and have at least one character between them.

```
str = 'My flowers may bloom in May';
expression = 'M(\w+)y';
replace = 'April';

newStr = regexprep(str,expression,replace)
```

```
newStr =
'My flowers may bloom in April'
```

Include Tokens in Replacement Text

Replace variations of the phrase `'walk up'` by capturing the letters that follow `'walk'` in a token.

```
str = 'I walk up, they walked up, we are walking up.';
expression = 'walk(\w*) up';
replace = 'ascend$1';

newStr = regexprep(str,expression,replace)
```

```
newStr =
'I ascend, they ascended, we are ascending.'
```

Include Dynamic Expression in Replacement Text

Replace lowercase letters at the beginning of sentences with their uppercase equivalents using the `upper` function.

```
str = 'here are two sentences. neither is capitalized.';
expression = '(^|\.)\s*.';
replace = '${upper($0)}';

newStr = regexprep(str,expression,replace)
```

```
newStr =
'Here are two sentences. Neither is capitalized.'
```

The regular expression matches single characters (`.`) that follow the beginning of the character vector (`^`) or a period (`\.`) and any whitespace (`\s*`). The `replace` expression calls the `upper` function for the currently matching character (`$0`).

Update Multiple Pieces of Text

Replace each occurrence of a double letter in a set of character vectors with the symbols `'--'`.

```
str = {                                   ...
'Whose woods these are I think I know.' ; ...
'His house is in the village though;'   ; ...
'He will not see me stopping here'      ; ...
'To watch his woods fill up with snow.'};

expression = '(.)\1';
replace = '--';
newStr = regexprep(str,expression,replace)
```

```
newStr = 4x1 cell array
    {'Whose w--ds these are I think I know.'}
    {'His house is in the vi--age though;'  }
    {'He wi-- not s-- me sto--ing here'     }
    {'To watch his w--ds fi-- up with snow.'}
```

Preserve Case in Original Text

Ignore letter case in the regular expression when finding matches, but mimic the letter case of the original text when updating.

```
str = 'My flowers may bloom in May';
expression = 'M(\w+)y';
replace = 'April';

newStr = regexprep(str,expression,replace,'preservecase')
```

```
newStr =
'My flowers april bloom in April'
```

Replace Zero-Length Matches

Insert text at the beginning of a character vector using the `'^'` operator, which returns a zero-length match, and the `'emptymatch'` keyword.

```
str = 'abc';
expression = '^';
replace = '__';

newStr = regexprep(str,expression,replace,'emptymatch')
```

```
newStr =
'__abc'
```

Input Arguments

`str` — Text to update
character vector | cell array of character vectors | string array

Text to update, specified as a character vector, a cell array of character vectors, or a string array.

Data Types: `char` | `cell` | `string`

`expression` — Regular expression
character vector | cell array of character vectors | string array

Regular expression, specified as a character vector, a cell array of character vectors, or a string array. Each expression can contain characters, metacharacters, operators, tokens, and flags that specify patterns to match in `str`.

The following tables describe the elements of regular expressions.

Metacharacters

Metacharacters represent letters, letter ranges, digits, and space characters. Use them to construct a generalized pattern of characters.

Metacharacter	Description	Example
`.`	Any single character, including white space	`'..ain'` matches sequences of five consecutive characters that end with `'ain'`.
`[`$c_1c_2c_3$`]`	Any character contained within the square brackets. The following characters are treated literally: `$ | . * + ?` and `-` when not used to indicate a range.	`'[rp.]ain'` matches `'rain'` or `'pain'` or `'.ain'`.
`[^`$c_1c_2c_3$`]`	Any character not contained within the square brackets. The following characters are treated literally: `$ | . * + ?` and `-` when not used to indicate a range.	`'[^*rp]ain'` matches all four-letter sequences that end in `'ain'`, except `'rain'` and `'pain'` and `'*ain'`. For example, it matches `'gain'`, `'lain'`, or `'vain'`.
`[`c_1`-`c_2`]`	Any character in the range of c_1 through c_2	`'[A-G]'` matches a single character in the range of `A` through `G`.
`\w`	Any alphabetic, numeric, or underscore character. For English character sets, `\w` is equivalent to `[a-zA-Z_0-9]`	`'\w*'` identifies a word.
`\W`	Any character that is not alphabetic, numeric, or underscore. For English character sets, `\W` is equivalent to `[^a-zA-Z_0-9]`	`'\W*'` identifies a term that is not a word.
`\s`	Any white-space character; equivalent to `[ \f\n\r\t\v]`	`'\w*n\s'` matches words that end with the letter `n`, followed by a white-space character.
`\S`	Any non-white-space character; equivalent to `[^ \f\n\r\t\v]`	`'\d\S'` matches a numeric digit followed by any non-white-space character.
`\d`	Any numeric digit; equivalent to `[0-9]`	`'\d*'` matches any number of consecutive digits.
`\D`	Any nondigit character; equivalent to `[^0-9]`	`'\w*\D\>'` matches words that do not end with a numeric digit.

Metacharacter	Description	Example
`\oN` or `\o{N}`	Character of octal value `N`	`'\o{40}'` matches the space character, defined by octal `40`.
`\xN` or `\x{N}`	Character of hexadecimal value `N`	`'\x2C'` matches the comma character, defined by hex `2C`.

Character Representation

Operator	Description
`\a`	Alarm (beep)
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\`*`char`*	Any character with special meaning in regular expressions that you want to match literally (for example, use `\\` to match a single backslash)

Quantifiers

Quantifiers specify the number of times a pattern must occur in the matching text.

Quantifier	Matches the expression when it occurs...	Example
`expr*`	0 or more times consecutively.	`'\w*'` matches a word of any length.
`expr?`	0 times or 1 time.	`'\w*(\.m)?'` matches words that optionally end with the extension `.m`.
`expr+`	1 or more times consecutively.	`'<img src="\w+\.gif">'` matches an `<img>` HTML tag when the file name contains one or more characters.

Quantifier	Matches the expression when it occurs...	Example
`expr{m,n}`	At least `m` times, but no more than `n` times consecutively. `{0,1}` is equivalent to `?`.	`'\S{4,8}'` matches between four and eight non-white-space characters.
`expr{m,}`	At least `m` times consecutively. `{0,}` and `{1,}` are equivalent to `*` and `+`, respectively.	`'<a href="\w{1,}\.html">'` matches an `<a>` HTML tag when the file name contains one or more characters.
`expr{n}`	Exactly `n` times consecutively. Equivalent to `{n,n}`.	`'\d{4}'` matches four consecutive digits.

Quantifiers can appear in three modes, described in the following table. *q* represents any of the quantifiers in the previous table.

Mode	Description	Example
`expr`*q*	Greedy expression: match as many characters as possible.	Given the text `'<tr><td><p>text</p></td>'`, the expression `'</?t.*>'` matches all characters between `<tr` and `/td>`: `'<tr><td><p>text</p></td>'`
`expr`*q*`?`	Lazy expression: match as few characters as necessary.	Given the text`'<tr><td><p>text</p></td>'`, the expression `'</?t.*?>'` ends each match at the first occurrence of the closing angle bracket (`>`): `'<tr>'  '<td>'  '</td>'`
`expr`*q*`+`	Possessive expression: match as much as possible, but do not rescan any portions of the text.	Given the text`'<tr><td><p>text</p></td>'`, the expression `'</?t.*+>'` does not return any matches, because the closing angle bracket is captured using `.*`, and is not rescanned.

Grouping Operators

Grouping operators allow you to capture tokens, apply one operator to multiple elements, or disable backtracking in a specific group.

Grouping Operator	Description	Example
`(expr)`	Group elements of the expression and capture tokens.	`'Joh?n\s(\w*)'` captures a token that contains the last name of any person with the first name `John` or `Jon`.
`(?:expr)`	Group, but do not capture tokens.	`'(?:[aeiou][^aeiou]){2}'` matches two consecutive patterns of a vowel followed by a nonvowel, such as `'anon'`. Without grouping, `'[aeiou][^aeiou]{2}'`matches a vowel followed by two nonvowels.
`(?>expr)`	Group atomically. Do not backtrack within the group to complete the match, and do not capture tokens.	`'A(?>.*)Z'` does not match `'AtoZ'`, although `'A(?:.*)Z'` does. Using the atomic group, `Z` is captured using `.*` and is not rescanned.
`(expr1|expr2)`	Match expression `expr1` or expression `expr2`. If there is a match with `expr1`, then `expr2` is ignored. You can include `?:` or `?>` after the opening parenthesis to suppress tokens or group atomically.	`'(let|tel)\w+'` matches words that start with `let` or `tel`.

Anchors

Anchors in the expression match the beginning or end of the input text or word.

Anchor	Matches the...	Example
`^expr`	Beginning of the input text.	`'^M\w*'` matches a word starting with `M` at the beginning of the text.
`expr$`	End of the input text.	`'\w*m$'` matches words ending with `m` at the end of the text.

Anchor	Matches the...	Example
`\<expr`	Beginning of a word.	`'\<n\w*'` matches any words starting with `n`.
`expr\>`	End of a word.	`'\w*e\>'` matches any words ending with `e`.

Lookaround Assertions

Lookaround assertions look for patterns that immediately precede or follow the intended match, but are not part of the match.

The pointer remains at the current location, and characters that correspond to the `test` expression are not captured or discarded. Therefore, lookahead assertions can match overlapping character groups.

Lookaround Assertion	Description	Example
`expr(?=test)`	Look ahead for characters that match `test`.	`'\w*(?=ing)'` matches terms that are followed by `ing`, such as `'Fly'` and `'fall'` in the input text `'Flying, not falling.'`
`expr(?!test)`	Look ahead for characters that do not match `test`.	`'i(?!ng)'` matches instances of the letter `i` that are not followed by `ng`.
`(?<=test)expr`	Look behind for characters that match `test`.	`'(?<=re)\w*'` matches terms that follow `'re'`, such as `'new'`, `'use'`, and `'cycle'` in the input text `'renew, reuse, recycle'`
`(?<!test)expr`	Look behind for characters that do not match `test`.	`'(?<!\d)(\d)(?!\d)'` matches single-digit numbers (digits that do not precede or follow other digits).

If you specify a lookahead assertion *before* an expression, the operation is equivalent to a logical `AND`.

Operation	Description	Example
`(?=test)expr`	Match both `test` and `expr`.	`'(?=[a-z])[^aeiou]'` matches consonants.

Operation	Description	Example
`(?!test)expr`	Match `expr` and do not match `test`.	`'(?![aeiou])[a-z]'` matches consonants.

Logical and Conditional Operators

Logical and conditional operators allow you to test the state of a given condition, and then use the outcome to determine which pattern, if any, to match next. These operators support logical `OR`, and `if` or `if/else` conditions.

Conditions can be tokens, lookaround operators, or dynamic expressions of the form `(?@cmd)`. Dynamic expressions must return a logical or numeric value.

Conditional Operator	Description	Example
`expr1|expr2`	Match expression `expr1` or expression `expr2`. If there is a match with `expr1`, then `expr2` is ignored.	`'(let|tel)\w+'` matches words that start with `let` or `tel`.
`(?(cond)expr)`	If condition `cond` is `true`, then match `expr`.	`'(?(?@ispc)[A-Z]:\\)'` matches a drive name, such as `C:\`, when run on a Windows system.
`(?(cond)expr1|expr2)`	If condition `cond` is `true`, then match `expr1`. Otherwise, match `expr2`.	`'Mr(s?)\..*?(?(1)her|his)\w*'` matches text that includes `her` when the text begins with `Mrs`, or that includes `his` when the text begins with `Mr`.

Token Operators

Tokens are portions of the matched text that you define by enclosing part of the regular expression in parentheses. You can refer to a token by its sequence in the text (an ordinal token), or assign names to tokens for easier code maintenance and readable output.

Ordinal Token Operator	Description	Example
`(expr)`	Capture in a token the characters that match the enclosed expression.	`'Joh?n\s(\w*)'` captures a token that contains the last name of any person with the first name `John` or `Jon`.
`\N`	Match the `N`th token.	`'<(\w+).*>.*</\1>'` captures tokens for HTML tags, such as `'title'` from the text `'<title>Some text</title>'`.
`(?(N)expr1|expr2)`	If the `N`th token is found, then match `expr1`. Otherwise, match `expr2`.	`'Mr(s?)\..*?(?(1)her|his) \w*'` matches text that includes `her` when the text begins with `Mrs`, or that includes `his` when the text begins with `Mr`.

Named Token Operator	Description	Example
`(?<name>expr)`	Capture in a named token the characters that match the enclosed expression.	`'(?<month>\d+)-(?<day>\d+)-(?<yr>\d+)'` creates named tokens for the month, day, and year in an input date of the form `mm-dd-yy`.
`\k<name>`	Match the token referred to by `name`.	`'<(?<tag>\w+).*>.*</\k<tag>>'` captures tokens for HTML tags, such as `'title'` from the text `'<title>Some text</title>'`.
`(?(name)expr1|expr2)`	If the named token is found, then match `expr1`. Otherwise, match `expr2`.	`'Mr(?<sex>s?)\..*?(?(sex)her|his) \w*'` matches text that includes `her` when the text begins with `Mrs`, or that includes `his` when the text begins with `Mr`.

Note If an expression has nested parentheses, MATLAB captures tokens that correspond to the outermost set of parentheses. For example, given the search pattern `'(and(y|rew))'`, MATLAB creates a token for `'andrew'` but not for `'y'` or `'rew'`.

Dynamic Regular Expressions

Dynamic expressions allow you to execute a MATLAB command or a regular expression to determine the text to match.

The parentheses that enclose dynamic expressions do *not* create a capturing group.

Operator	Description	Example
`(??expr)`	Parse `expr` and include the resulting term in the match expression. When parsed, `expr` must correspond to a complete, valid regular expression. Dynamic expressions that use the backslash escape character (`\`) require two backslashes: one for the initial parsing of `expr`, and one for the complete match.	`'^(\d+)((??\\w{$1}))'` determines how many characters to match by reading a digit at the beginning of the match. The dynamic expression is enclosed in a second set of parentheses so that the resulting match is captured in a token. For instance, matching `'5XXXXX'` captures tokens for `'5'` and `'XXXXX'`.
`(??@cmd)`	Execute the MATLAB command represented by `cmd`, and include the output returned by the command in the match expression.	`'(.{2,}).?(??@fliplr($1))'` finds palindromes that are at least four characters long, such as `'abba'`.
`(?@cmd)`	Execute the MATLAB command represented by `cmd`, but discard any output the command returns. (Helpful for diagnosing regular expressions.)	`'\w*?(\w)(?@disp($1))\1\w*'` matches words that include double letters (such as `pp`), and displays intermediate results.

Within dynamic expressions, use the following operators to define replacement text.

Replacement Operator	Description
`$&` or `$0`	Portion of the input text that is currently a match
`$` `	Portion of the input text that precedes the current match
`$'`	Portion of the input text that follows the current match (use `$''` to represent `$'`)
`$N`	`N`th token
`$<name>`	Named token
`${cmd}`	Output returned when MATLAB executes the command, `cmd`

Comments

Characters	Description	Example
`(?#comment)`	Insert a comment in the regular expression. The comment text is ignored when matching the input.	`'(?# Initial digit)\<\d\w+'` includes a comment, and matches words that begin with a number.

Search Flags

Search flags modify the behavior for matching expressions. An alternative to using a search flag within an expression is to pass an `option` input argument.

Flag	Description
`(?-i)`	Match letter case (default for `regexp` and `regexprep`).
`(?i)`	Do not match letter case (default for `regexpi`).
`(?s)`	Match dot (`.`) in the pattern with any character (default).
`(?-s)`	Match dot in the pattern with any character that is not a newline character.
`(?-m)`	Match the `^` and `$` metacharacters at the beginning and end of text (default).
`(?m)`	Match the `^` and `$` metacharacters at the beginning and end of a line.
`(?-x)`	Include space characters and comments when matching (default).
`(?x)`	Ignore space characters and comments when matching. Use `'\ '` and `'\#'` to match space and # characters.

The expression that the flag modifies can appear either after the parentheses, such as

```
(?i)\w*
```

or inside the parentheses and separated from the flag with a colon (`:`), such as

```
(?i:\w*)
```

The latter syntax allows you to change the behavior for part of a larger expression.

Data Types: `char` | `cell` | `string`

`replace` — Replacement text
character vector | cell array of character vectors | string array

Replacement text, specified as a character vector, a cell array of character vectors, or a string array, as follows:

- If `replace` is a single character vector and `expression` is a cell array of character vectors, then `regexprep` uses the same replacement text for each expression.
- If `replace` is a cell array of `N` character vectors and `expression` is a single character vector, then `regexprep` attempts `N` matches and replacements.
- If both `replace` and `expression` are cell arrays of character vectors, then they must contain the same number of elements. `regexprep` pairs each `replace` element with its matching element in `expression`.

The replacement text can include regular characters, special characters (such as tabs or new lines), or replacement operators, as shown in the following tables.

Replacement Operator	Description
`$&` or `$0`	Portion of the input text that is currently a match
`$` `	Portion of the input text that precedes the current match
`$'`	Portion of the input text that follows the current match (use `$''` to represent `$'`)
`$N`	`N`th token
`$<name>`	Named token
`${cmd}`	Output returned when MATLAB executes the command, `cmd`

Operator	Description
`\a`	Alarm (beep)
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\`*`char`*	Any character with special meaning in regular expressions that you want to match literally (for example, use `\\` to match a single backslash)

Data Types: `char` | `cell` | `string`

option — Search or replacement option
`'once'` | `N` | `'warnings'` | `'ignorecase'` | `'preservecase'` | `'emptymatch'` | `'dotexceptnewline'` | `'lineanchors'` | ...

Search or replacement option, specified as a character vector or an integer value, as shown in the following table.

Options come in sets: one option that corresponds to the default behavior, and one or two options that allow you to override the default. Specify only one option from a set. Options can appear in any order.

Default	Override	Description
`'all'`	`'once'`	Match and replace the expression as many times as possible (default), or only once.
	`N`	Replace only the `N`th occurrence of the match, where `N` is an integer value.
`'nowarnings'`	`'warnings'`	Suppress warnings (default), or display them.
`'matchcase'`	`'ignorecase'`	Match letter case (default), or ignore case while matching and replacing.
	`'preservecase'`	Ignore case while matching, but preserve the case of corresponding characters in the original text while replacing.
`'noemptymatch'`	`'emptymatch'`	Ignore zero length matches (default), or include them.
`'dotall'`	`'dotexceptnewline'`	Match dot with any character (default), or all except newline (`\n`).
`'stringanchors'`	`'lineanchors'`	Apply `^` and `$` metacharacters to the beginning and end of a character vector (default), or to the beginning and end of a line. The newline character (`\n`) specifies the end of a line. The beginning of a line is specified as the first character, or any character that immediately follows a newline character.
`'literalspacing'`	`'freespacing'`	Include space characters and comments when matching (default), or ignore them. With `freespacing`, use `'\ '` and `'\#'` to match space and `#` characters.

Data Types: `char` | `string`

Output Arguments

`newStr` — Updated text
character vector | cell array of character vectors | string array

Updated text, returned as a character vector, a cell array of character vectors, or a string array. The data type of `newStr` is the same as the data type of `str`.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

See Also
`contains` | `regexp` | `replace` | `strcmp` | `strfind` | `strrep`

Topics
“Lookahead Assertions in Regular Expressions”
“Tokens in Regular Expressions”
“Dynamic Regular Expressions”

Introduced before R2006a

regexptranslate

Translate text into regular expression

Syntax

```
newStr = regexptranslate(op,str)
```

Description

`newStr = regexptranslate(op,str)` translates `str` into a regular expression and returns the result in `newStr`. You can use `newStr` as a regular expression in the `regexp`, `regexpi`, and `regexprep` functions. The input argument `op` specifies the type of translation that `regexptranslate` performs. For example, if you specify `op` as `'escape'`, then `regexptranslate` translates special characters in `str` so that they are literal characters in the output. `newStr` has the same data type as `str`.

Examples

Translate Special Character

Translate a special character in a character vector using the `regexptranslate` function. Then use the result as a regular expression in `regexp`.

Create a character vector that contains the characters `'\n'`.

```
chr = 'The sequence \n generates a new line.'
```

```
chr =
'The sequence \n generates a new line.'
```

Create a regular expression that finds `'\n'` as a sequence of the two consecutive characters `'\'` and `'n'`. Since the `regexp` function interprets `'\n'` as a newline character, use `regexptranslate` to create a regular expression to escape `'\n'`.

```
pattern = regexptranslate('escape','\n')
```

```
pattern =
'\\n'
```

Find the starting index of `'\n'` in `chr`. To prevent `regexp` from interpreting `'\n'` as a newline, use `pattern` as the regular expression.

```
idx = regexp(chr,pattern)
```

```
idx = 14
```

Call `regexp` without escaping `'\n'`. Since `regexp` interprets `'\n'` as a newline, it does not find the literal characters in `chr`. The `regexp` function returns an empty array when it does not find a match.

```
idx = regexp(chr,'\n')
```

```
idx =

     []
```

Escape Special Characters in Replacement Text

Create a string. Starting in R2017a, you can create strings using double quotes.

```
str = "Put your money in."
```

```
str =
"Put your money in."
```

Specify the text, `'$0.02'`, as the text to replace the word `'money'`. To escape the `'$'` and `'.'` characters, use `regexptranslate`.

```
r = regexptranslate('escape','$0.02')
```

```
r =
'\$0\.02'
```

Replace `'money'` using the `regexprep` function.

```
newStr = regexprep(str,'money',r)
```

```
newStr =
"Put your $0.02 in."
```

Translate Wildcard Characters

Create a string array that contains file names. Then find only the file names that end with `'.mat'`.

```
str = ["test1.mat","myfile.mat","my-matlab-script.m", ...
       "jan30.mat","table3.xls"]
```

```
str = 1x5 string array
  Columns 1 through 4

    "test1.mat"    "myfile.mat"    "my-matlab-script.m"    "jan30.mat"

  Column 5

    "table3.xls"
```

To match strings with a regular expression, specify `'*.mat'` as the regular expression. Then translate the wildcard character, `'*'`, using the `regexptranslate` function.

```
pattern = regexptranslate('wildcard','*.mat')
```

```
pattern =
'.*\.mat'
```

Find matching elements in `str` using the regular expression specified by `pattern`.

```
matches = regexp(str,pattern)
```

```
matches=1×5 cell
    {[1]}    {[1]}    {0x0 double}    {[1]}    {0x0 double}
```

Create a logical array, `TF`, that contains `1` where corresponding elements of `str` matched `pattern`. Then index into `str` using `TF` to display the file names that end with `'.mat'`.

```
tf = ~cellfun('isempty',matches);
newStr = str(tf)
```

```
newStr = 1x3 string array
    "test1.mat"    "myfile.mat"    "jan30.mat"
```

Replace Text with Regular Expression

Create a character vector that contains words separated by whitespace characters, such as spaces and newline characters.

```
chr = 'Whose woods these are I think I know.';
chr = [chr newline 'His house is in the village though']
```

```
chr =
    'Whose woods these are I think I know.
     His house is in the village though'
```

Specify `'\s'` as a regular expression that matches whitespace characters. Then replace those characters in `chr`.

```
expression = '\s';
newChr = regexptranslate('flexible',chr,expression)
```

```
newChr =
'Whose\swoods\sthese\sare\sI\sthink\sI\sknow.\sHis\shouse\sis\sin\sthe\svillage\sthough
```

Input Arguments

op — Translation type
`'escape'` | `'wildcard'` | `'flexible'`

Type of translation, specified as a character vector or string scalar. You can translate special characters or wildcard characters, or replace text with a matching regular expression, using the options in the table.

Type of Translation	Description
`'escape'`	Translate all special characters in `str`, such as `'$'`, `'.'`, `'?'`,`'['`, so that they are treated as literal characters when used in `regexp`, `regexpi`, and `regexprep`. The translation inserts a backslash, or escape, character, `'\'`, before each special character in `str`.
`'wildcard'`	Translate all wildcard and `'.'` characters in `str` so that they are treated as literal wildcard characters and periods when used in `regexp`, `regexpi`, and `regexprep`. The translation replaces all instances of `'*'` with `'.*'`, all instances of `'?'` with `'.'`, and all instances of `'.'` with `'\.'`.
`'flexible'`	Replace text in `str` with a regular expression that matches the text. If you specify `'flexible'`, then also specify a regular expression to use as a replacement: `newStr = regexptranslate('flexible',str,expression)`. The `expression` input can be a character vector or string scalar. This syntax is equivalent to `newStr = regexprep(str,expression,regexptranslate('escape',expression))`.

`str` — Input text

character vector | cell array of character vectors | string array

Input text, specified as a character vector, a cell array of character vectors, or a string array.

See Also

`contains` | `regexp` | `regexpi` | `regexprep` | `replace` | `strfind` | `strjoin` | `strrep` | `strsplit`

Topics

"Regular Expressions"

Introduced before R2006a

registerevent

Associate event handler for COM object event at run time

Syntax

```
registerevent(c,eventhandler)
```

Description

`registerevent(c,eventhandler)` registers event handler routines with their corresponding events.

Input Arguments

c — COM object
function handle

COM object, specified as a function handle.

eventhandler — Function to call when event occurs
string | character vector | function handle

Function to call when an event occurs, specified as a string or a character vector naming the function or a function handle that maps to that function. Strings and character vectors are not case-sensitive.

Limitations

- COM functions are available on Microsoft Windows systems only.

See Also

`eventlisteners` | `isevent` | `unregisterallevents` | `unregisterevent`

Topics

"COM Event Handlers"

Introduced before R2006a

regionZoomInteraction

Region-zoom interaction

Description

A region-zoom interaction allows you to zoom into a rectangular region of a 2-D chart without having to select any buttons in the axes toolbar. To enable region zooming, set the `Interactions` property of the axes to a `regionZoomInteraction` object. When this interaction is enabled, you can zoom into a region by dragging within the chart.

To enable multiple interactions, set the `Interactions` property to an array of objects.

Creation

Syntax

```
rz = regionZoomInteraction
rz = regionZoomInteraction('Dimensions',d)
```

Description

`rz = regionZoomInteraction` creates a region-zoom interaction object.

`rz = regionZoomInteraction('Dimensions',d)` sets the `Dimensions` property. Use this property to constrain zooming to specific dimensions. For example, `rz = regionZoomInteraction('Dimensions','x')` constrains zooming to the *x*-dimension.

Properties

`Dimensions` — Dimensions
`'xyz'` (default) | `'xy'` | `'yz'` | `'xz'` | `'x'` | `'y'` | `'z'`

Dimensions to allow zooming, specified as one of these values:

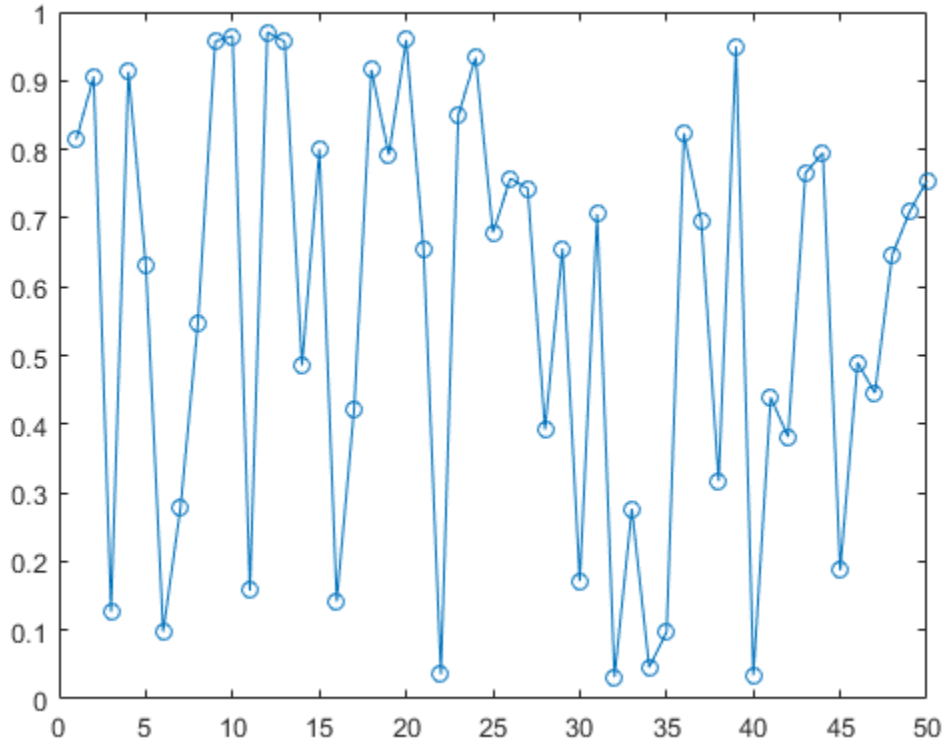
- `'xyz'` — Allows zooming in all dimensions.
- `'xy'` — Allows zooming in the *x* and *y* dimensions only.
- `'yz'` — Allows zooming in the *y* and *z* dimensions only.
- `'xz'` — Allows zooming in the *x* and *z* dimensions only.
- `'x'` — Allows zooming in the *x* dimension only.
- `'y'` — Allows zooming in the *y* dimension only.
- `'z'` — Allows zooming in the *z* dimension only.

Examples

Axes with Region Zoom and Data Tip Interactions

Create a plot of fifty random numbers. Get the current axes, and replace the default interactions with the region-zoom and data tip interactions. Then hover over the plotted points to display data tips. Drag to zoom into a region of the plot.

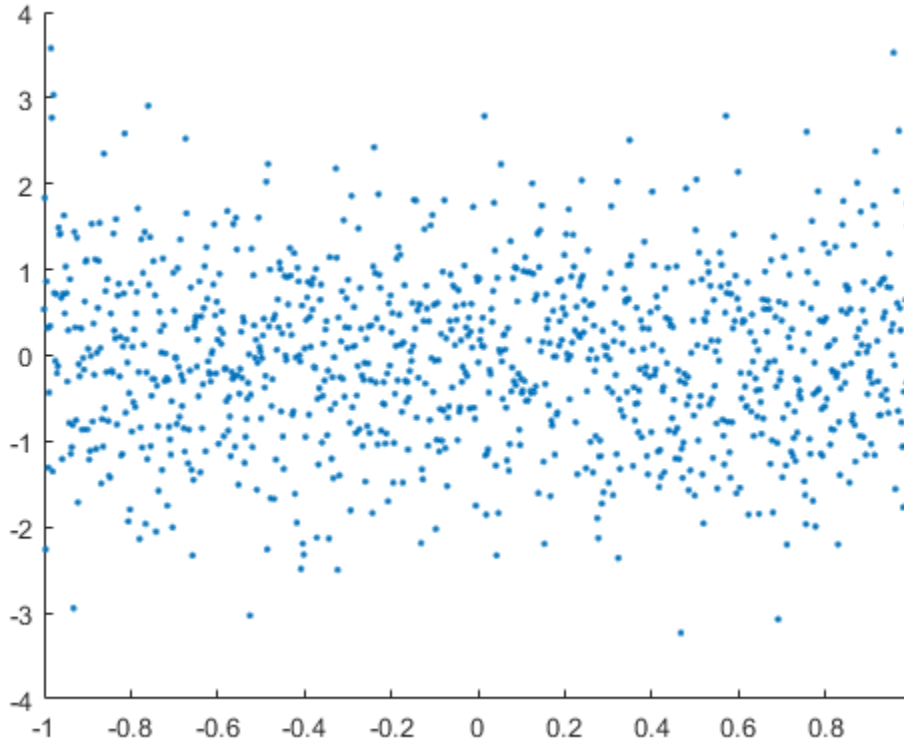
```
plot(rand(1,50),'-o')
ax = gca;
ax.Interactions = [regionZoomInteraction dataTipInteraction];
```



Restrict Region Zooming to X-Dimension

Create a scatter plot of normally distributed random data. Replace the default set of interactions with a region-zoom interaction that operates only in the *x*-dimension. Then drag within the plot to zoom into a region of interest.

```
x = linspace(-1,1,1000);
y = randn(1,1000);
scatter(x,y,'.')
ax = gca;
ax.Interactions = regionZoomInteraction('Dimensions','x');
```



Limitations

Region-zoom interactions are not supported in 3-D axes or in the Live Editor.

Tips

In most cases, the axes have a default set of interactions which depend on the type of chart you are displaying. You can replace the default set with a new set of interactions, but you cannot access or modify any of the interactions in the default set.

See Also

`dataTipInteraction` | `disableDefaultInteractivity` | `enableDefaultInteractivity` | `panInteraction` | `rotateInteraction` | `rulerPanInteraction` | `zoomInteraction`

Topics

"Control Chart Interactivity"

Introduced in R2019a

regmatlabserver

Register current MATLAB as Automation server

Syntax

```
regmatlabserver
[status,message] = regmatlabserver
```

Description

`regmatlabserver` registers the current MATLAB executable as an Automation server. MATLAB displays a minimized command window.

`[status,message] = regmatlabserver` returns the status and error message, if any, reported by the operating system.

Examples

Register MATLAB as Automation Server

Start MATLAB with the **Run as administrator** option.

At the MATLAB prompt, type:

```
[s,msg] = regmatlabserver
```

```
s =

     0


msg =

     ''
```

MATLAB successfully registered.

Output Arguments

`status` — MATLAB registration status
`0` | nonzero

MATLAB registration status returned as `0` or nonzero. If the attempt to register MATLAB is successful, `status` is `0`. Otherwise, `status` is nonzero.

`message` — System error message
character vector

System error message returned as a character vector if registration is unsuccessful.

See Also

Topics
"Register MATLAB as Automation Server"

Introduced before R2006a

rehash

Refresh function and file system path caches

Syntax

```
rehash
rehash path
rehash toolbox
rehash toolboxcache
```

Description

`rehash` updates the MATLAB list of known files and classes for folders on the search path that are not in *`matlabroot`*. It compares the timestamps for loaded functions against their timestamps on disk. It clears loaded functions if the files on disk are newer. All of this normally happens each time MATLAB displays the Command Window prompt. Use `rehash` with no arguments only when you run a program file that updates another program file, and the calling file needs to use the updated version of the second file before the calling file has finished running.

`rehash path` performs the same updates as `rehash`, except that it unconditionally updates the list of known files and classes for all folders on the search path that are not in *`matlabroot`*. Run `rehash path` only if you receive a warning during MATLAB startup notifying you that MATLAB could not tell if a folder has changed, and you encounter problems with MATLAB not using the most current versions of your program files.

`rehash toolbox` performs the same updates as `rehash path`, except it updates the list of known files and classes for *all* folders on the search path, including those in *`matlabroot`*. Run `rehash toolbox` when you change, add, or remove files in *`matlabroot`* during a session. Typically, you should not make changes to files and folders in *`matlabroot`*.

`rehash toolboxcache` performs the same updates as `rehash toolbox`, and also updates the cache file. This is equivalent to clicking the **Update Toolbox Path Cache** button in the General Preferences dialog box.

See Also

`addpath` | `matlabroot` | `path` | `rmpath`

Topics

"Toolbox Path Caching in MATLAB"
"What Is the MATLAB Search Path?"

Introduced before R2006a

release

Release COM interface

Syntax

```
release(c)
```

Description

`release(c)` releases the interface and all resources used by the interface. Other interfaces on that object might still be active.

You must release the handle when you are done with the interface. A released interface is no longer valid. MATLAB generates an error if you try to use an object that represents that interface.

To release the interface and delete the object, use the `delete` function.

Input Arguments

c — COM object
function handle

COM object, specified as a function handle.

Limitations

- COM functions are available on Microsoft Windows systems only.

See Also

`actxserver`

Topics

"Release COM Interfaces and Objects"

Introduced before R2006a

ReleaseCompatibilityResults

Release compatibility results object

Description

A `ReleaseCompatibilityResults` object represents the results of upgrading the personal settings of a toolbox for a specific version number.

Creation

Create a `ReleaseCompatibilityResults` object for a specific toolbox version number by using the `matlab.settings.loadSettingsCompatibilityResults` function. For example, this command gets the `ReleaseCompatibilityResults` object for version 2 of the toolbox `mytoolbox`.

```
myCompatibilityResults = matlab.settings.loadSettingsCompatibilityResults('mytoolbox','Version2')
```

```
ans = 
  ReleaseCompatibilityResults with properties:
                VersionLabel: "Version2"
     PreValidationExceptions: [0×0 matlab.settings.ReleaseCompatibilityException]
                     Results: [1×1 matlab.settings.VersionResults]
```

Properties

`VersionLabel` — Toolbox version of results

string scalar

Toolbox version of results, specified as a string scalar.

Example: `'Version2'`

`PreValidationExceptions` — List of validation exceptions

array of release compatibility exception objects

List of validation exceptions, specified as an array of release compatibility exception objects. The list contains exceptions that occur while validating the personal and factory settings trees before upgrading the personal settings.

`Results` — Upgrade Results
version results object

Upgrade results, specified as a version results object.

Examples

Get Results of Upgrading Personal Settings

Create functions to create and then upgrade a toolbox factory tree and then test that the upgrade completes successfully.

Create the function `createMyToolboxFactoryTree` that creates the factory settings tree for the toolbox `mytoolbox`.

```
function myToolboxFactoryTree = createMyToolboxFactoryTree()
    myToolboxFactoryTree = matlab.settings.FactoryGroup.createToolboxGroup('mytoolbox', ...
        'Hidden',false);

    toolboxFontGroup = addGroup(myToolboxFactoryTree,'font','Hidden',false)
    addSetting(toolboxFontGroup,'MyFontSize','FactoryValue',11,'Hidden',false, ...
        'ValidationFcn',@matlab.settings.mustBeNumericScalar)
    addSetting(toolboxFontGroup,'MyFontColor','FactoryValue','Black', ...
        'Hidden',false,'ValidationFcn',@matlab.settings.mustBeStringScalar);
end
```

Create the function `createMyToolboxSettingsFileUpgraders` with an empty settings file upgrader object.

```
function upgraders = createMyToolboxSettingsFileUpgraders()
    upgraders = matlab.settings.SettingsFileUpgrader.empty;
end
```

Create the `settingsInfo.json` file for the toolbox. Specify `mytoolbox` as the root settings group name, `createMyToolboxFactoryTree` as the settings tree creation function, and `createMyToolboxSettingsFileUpgraders` as the settings tree upgrade function. Place `settingsInfo.json` in the toolbox `resources` folder.

```
{
"ToolboxGroupName" : "mytoolbox",
"Hidden" : "false",
"CreateTreeFcn" : "createMyToolboxFactoryTree",
"CreateUpgradersFcn" : "createMyToolboxSettingsFileUpgraders"
}
```

Add the folder that contains the settings tree creation function and the toolbox resources folder to the MATLAB path. Then, load the factory settings tree for `mytoolbox`.

```
matlab.settings.reloadFactoryFile('mytoolbox');
```

Use the `settings` function to access the root of the settings tree and set the personal value for the `MyFontSize` setting.

```
s = settings;
s.mytoolbox.font.MyFontSize.PersonalValue = 15;
```

Change the settings names in `createMyToolboxFactoryTree` to be `FontSize` and `FontColor`.

```
function myToolboxFactoryTree = createMyToolboxFactoryTree()
    myToolboxFactoryTree = matlab.settings.FactoryGroup.createToolboxGroup('mytoolbox', ...
        'Hidden',false);

    toolboxFontGroup = addGroup(myToolboxFactoryTree,'font','Hidden',false)
    addSetting(toolboxFontGroup,'FontSize','FactoryValue',11,'Hidden',false, ...
        'ValidationFcn',@matlab.settings.mustBeNumericScalar)
    addSetting(toolboxFontGroup,'FontColor','FactoryValue','Black', ...
        'Hidden',false,'ValidationFcn',@matlab.settings.mustBeStringScalar);
end
```

Record the rename of the two settings in the `createMyToolboxSettingsFileUpgraders` function as changes to the settings tree for version 2 of `mytoolbox`.

```
function upgraders = createMyToolboxSettingsFileUpgraders()
    upgraders = matlab.settings.SettingsFileUpgrader('Version2');
    upgraders.move('mytoolbox.font.MyFontSize','mytoolbox.font.FontSize');
    upgraders.move('mytoolbox.font.MyFontColor','mytoolbox.font.FontColor');
end
```

Reload the factory settings tree for `mytoolbox`.

```
matlab.settings.reloadFactoryFile('mytoolbox');
```

Use the `settings` function to access the root of the settings tree and verify that the personal value for the `FontSize` setting was correctly moved from the `MyFontSize` setting.

```
s = settings;
s.mytoolbox.font.FontSize
```

```
ans = 
  Setting 'mytoolbox.font.FontSize' with properties:
       ActiveValue: 15
    TemporaryValue: <no value>
```

```
     PersonalValue: 15
      FactoryValue: 11
```

Get the upgrade results for version 2 of `mytoolbox`.

```
matlab.settings.loadSettingsCompatibilityResults('mytoolbox','Version2')
```

```
ans = 
  ReleaseCompatibilityResults with properties:
              VersionLabel: "Version2"
    PreValidationExceptions: [0×0 matlab.settings.ReleaseCompatibilityException]
                    Results: [1×1 matlab.settings.VersionResults]
```

See Also

`ReleaseCompatibilityException` | `VersionResults` | `matlab.settings.loadSettingsCompatibilityResults`

Topics

"Create Factory Settings for Toolboxes"
"Access and Modify Settings"

Introduced in R2019b

ReleaseCompatibilityException

Release compatibility exception object

Description

A `ReleaseCompatibilityException` object represents an exception that occurs when upgrading the personal settings of a toolbox.

Creation

Create a `ReleaseCompatibilityResults` object for a specific toolbox version number by using the `matlab.settings.loadSettingsCompatibilityResults` function. Access the `PreValidationExceptions` property to get an array of `ReleaseCompatibilityException` objects.

For example, this code gets the prevalidation exceptions for version 2 of the toolbox `mytoolbox` as an array of `ReleaseCompatibilityException` objects.

```
myCompatibilityResults = matlab.settings.loadSettingsCompatibilityResults('mytoolbox','Version2');
myCompatibilityResults.PreValidationExceptions
```

```
ans =
  0×0 ReleaseCompatibilityException array with properties:
    ExceptionString
    ExceptionID
```

Properties

`ExceptionString` — Exception message
string scalar

Exception message, specified as a string scalar. If an exception occurs, `ExceptionString` contains the exception message.

`ExceptionID` — Exception identifier
string scalar

Exception identifier, specified as a string scalar. If an exception occurs, `ExceptionID` contains the identifier of the exception.

Examples

Get Results of Upgrading Personal Settings

Create functions to create and then upgrade a toolbox factory tree and then test that the upgrade completes successfully.

Create the function `createMyToolboxFactoryTree` that creates the factory settings tree for the toolbox `mytoolbox`.

```
function myToolboxFactoryTree = createMyToolboxFactoryTree()
    myToolboxFactoryTree = matlab.settings.FactoryGroup.createToolboxGroup('mytoolbox', ...
        'Hidden',false);

    toolboxFontGroup = addGroup(myToolboxFactoryTree,'font','Hidden',false)
    addSetting(toolboxFontGroup,'MyFontSize','FactoryValue',11,'Hidden',false, ...
        'ValidationFcn',@matlab.settings.mustBeNumericScalar)
    addSetting(toolboxFontGroup,'MyFontColor','FactoryValue','Black', ...
        'Hidden',false,'ValidationFcn',@matlab.settings.mustBeStringScalar);
end
```

Create the function `createMyToolboxSettingsFileUpgraders` with an empty settings file upgrader object.

```
function upgraders = createMyToolboxSettingsFileUpgraders()
    upgraders = matlab.settings.SettingsFileUpgrader.empty;
end
```

Create the `settingsInfo.json` file for the toolbox. Specify `mytoolbox` as the root settings group name, `createMyToolboxFactoryTree` as the settings tree creation function, and `createMyToolboxSettingsFileUpgraders` as the settings tree upgrade function. Place `settingsInfo.json` in the toolbox `resources` folder.

```
{
"ToolboxGroupName" : "mytoolbox",
"Hidden" : "false",
"CreateTreeFcn" : "createMyToolboxFactoryTree",
"CreateUpgradersFcn" : "createMyToolboxSettingsFileUpgraders"
}
```

Add the folder that contains the settings tree creation function and the toolbox resources folder to the MATLAB path. Then, load the factory settings tree for `mytoolbox`.

```
matlab.settings.reloadFactoryFile('mytoolbox');
```

Use the `settings` function to access the root of the settings tree and set the personal value for the `MyFontSize` setting.

```
s = settings;
s.mytoolbox.font.MyFontSize.PersonalValue = 15;
```

Change the settings names in `createMyToolboxFactoryTree` to be `FontSize` and `FontColor`.

```
function myToolboxFactoryTree = createMyToolboxFactoryTree()
    myToolboxFactoryTree = matlab.settings.FactoryGroup.createToolboxGroup('mytoolbox', ...
        'Hidden',false);

    toolboxFontGroup = addGroup(myToolboxFactoryTree,'font','Hidden',false)
    addSetting(toolboxFontGroup,'FontSize','FactoryValue',11,'Hidden',false, ...
        'ValidationFcn',@matlab.settings.mustBeNumericScalar)
    addSetting(toolboxFontGroup,'FontColor','FactoryValue','Black', ...
        'Hidden',false,'ValidationFcn',@matlab.settings.mustBeStringScalar);
end
```

Record the rename of the two settings in the `createMyToolboxSettingsFileUpgraders` function as changes to the settings tree for version 2 of `mytoolbox`. When recording the rename of the two settings, introduce an error in one of the paths. For example, remove `mytoolbox` from the destination path of the first recorded change.

```
function upgraders = createMyToolboxSettingsFileUpgraders()
    upgraders = matlab.settings.SettingsFileUpgrader('Version2');
    upgraders.move('mytoolbox.font.MyFontSize','font.FontSize');
    upgraders.move('mytoolbox.font.MyFontColor','mytoolbox.font.FontColor');
end
```

Reload the factory settings tree for `mytoolbox`.

```
matlab.settings.reloadFactoryFile('mytoolbox');
```

Use the `settings` function to access the root of the settings tree and verify the personal value for the `FontSize` setting. Notice that the personal value was not moved.

```
s = settings;
s.mytoolbox.font.FontSize
```

```
ans = 
  Setting 'mytoolbox.font.FontSize' with properties:
       ActiveValue: 11
    TemporaryValue: <no value>
     PersonalValue: <no value>
      FactoryValue: 11
```

Get the upgrade results for version 2 of `mytoolbox` and check the exception log for the first operation.

```
upgradeResults = matlab.settings.loadSettingsCompatibilityResults('mytoolbox','Version2');
upgradeResults.Results.VersionChanges(1).ExceptionLog
```

```
ans 
  ReleaseCompatibilityException with properties:
    ExceptionString: ''font.FontSize' is an invalid path for adding a new group.
                     The path needs to be relative to the master settings file
                     and cannot contain ".." between folder names.'
        ExceptionID: "MATLAB:settings:config:PathNotRelativeToMasterFile"
```

See Also

`OperationResult` | `matlab.settings.loadSettingsCompatibilityResults`

Topics

"Create Factory Settings for Toolboxes"
"Access and Modify Settings"

Introduced in R2019b

rem

Remainder after division

Syntax

```
r = rem(a,b)
```

Description

`r = rem(a,b)` returns the remainder after division of `a` by `b`, where `a` is the dividend and `b` is the divisor. This function is often called the remainder operation, which can be expressed as `r = a - b.*fix(a./b)`. The `rem` function follows the convention that `rem(a,0)` is `NaN`.

Examples

Remainder After Division of Scalar

Compute the remainder after dividing 5 into 23.

```
a = 23;
b = 5;
r = rem(a,b)
```

```
r = 3
```

Remainder After Division of Vector

Find the remainder after division for a vector of integers and the divisor `3`.

```
a = 1:5;
b = 3;
r = rem(a,b)
```

```
r = 1×5

     1     2     0     1     2
```

Remainder After Division for Positive and Negative Values

Find the remainder after division for a set of integers including both positive and negative values. Note that nonzero results have the same sign as the dividend.

```
a = [-4 -1 7 9];
b = 3;
r = rem(a,b)
```

```
r = 1×4

    -1    -1     1     0
```

Remainder After Division for Floating-Point Values

Find the remainder after division for several angles using a divisor of `2*pi`. When possible, `rem` attempts to produce exact integer results by compensating for floating-point round-off effects.

```
theta = [0.0 3.5 5.9 6.2 9.0 4*pi];
b = 2*pi;
r = rem(theta,b)
```

```
r = 1×6

         0    3.5000    5.9000    6.2000    2.7168         0
```

Input Arguments

a — Dividend

scalar | vector | matrix | multidimensional array

Dividend, specified as a scalar, vector, matrix, or multidimensional array. `a` must be a real-valued array of any numerical type. Numeric inputs `a` and `b` must either be the same size or have sizes that are compatible (for example, `a` is an `M`-by-`N` matrix and `b` is a scalar or `1`-by-`N` row vector). For more information, see "Compatible Array Sizes for Basic Operations".

If `a` and `b` are duration arrays, then they must be the same size unless one is a scalar. If one input is a duration array, the other input can be a duration array or a numeric array. In this context, `rem` treats numeric values as a number of standard 24-hour days.

If one input has an integer data type, then the other input must be of the same integer data type or be a scalar `double`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `duration` | `char`

b — Divisor

scalar | vector | matrix | multidimensional array

Divisor, specified as a scalar, vector, matrix, or multidimensional array. `b` must be a real-valued array of any numerical type. Numeric inputs `a` and `b` must either be the same size or have sizes that are compatible (for example, `a` is an `M`-by-`N` matrix and `b` is a scalar or `1`-by-`N` row vector). For more information, see "Compatible Array Sizes for Basic Operations".

If `a` and `b` are duration arrays, then they must be the same size unless one is a scalar. If one input is a duration array, the other input can be a duration array or a numeric array. In this context, `rem` treats numeric values as a number of standard 24-hour days.

If one input has an integer data type, then the other input must be of the same integer data type or be a scalar `double`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `duration` | `char`

More About

Differences Between mod and rem

The concept of remainder after division is not uniquely defined, and the two functions `mod` and `rem` each compute a different variation. The `mod` function produces a result that is either zero or has the same sign as the divisor. The `rem` function produces a result that is either zero or has the same sign as the dividend.

Another difference is the convention when the divisor is zero. The `mod` function follows the convention that `mod(a,0)` returns `a`, whereas the `rem` function follows the convention that `rem(a,0)` returns `NaN`.

Both variants have their uses. For example, in signal processing, the `mod` function is useful in the context of periodic signals because its output is periodic (with period equal to the divisor).

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Generated code performs the arithmetic using the output class. Results might not match MATLAB due to differences in rounding errors.
- If one of the inputs has type `int64` or `uint64`, then both inputs must have the same type.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- 64-bit integers are not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`mod`

Introduced before R2006a

remove

Package: `containers`

Delete key-value pairs from `Map` object

Syntax

```
remove(M,keySet)
```

Description

`remove(M,keySet)` deletes the specified keys, and the values associated with them, from the input `Map` object.

Examples

Remove Key-Value Pair

Create a `Map` object. Display its keys and values.

```
ids = [437 1089 2362];
names = {'Li, N.','Jones, R.','Sanchez, C.'};
M = containers.Map(ids,names)
```

```
M = 
  Map with properties:

        Count: 3
      KeyType: double
    ValueType: char
```

```
keys(M)
```

```
ans=1×3 cell
    {[437]}    {[1089]}    {[2362]}
```

```
values(M)
```

```
ans = 1x3 cell array
    {'Li, N.'}    {'Jones, R.'}    {'Sanchez, C.'}
```

Remove a key-value pair. Display the updated keys and values.

```
remove(M,2362);
keys(M)
```

```
ans=1×2 cell
    {[437]}    {[1089]}
```

```
values(M)
```

```
ans = 1x2 cell array
    {'Li, N.'}    {'Jones, R.'}
```

Remove Multiple Key-Value Pairs

Create a `Map` object.

```
months = {'Jan','Feb','Mar','Apr'};
rainfall = [327.2 368.2 197.6 178.4];
M = containers.Map(months,rainfall);
keys(M)
```

```
ans = 1x4 cell array
    {'Apr'}    {'Feb'}    {'Jan'}    {'Mar'}
```

```
values(M)
```

```
ans=1×4 cell
    {[178.4000]}    {[368.2000]}    {[327.2000]}    {[197.6000]}
```

To remove multiple key-value pairs, specify the keys as a cell array.

```
keySet = {'Feb','Mar','Apr'};
remove(M,keySet);
keys(M)
```

```
ans = 1x1 cell array
    {'Jan'}
```

```
values(M)
```

```
ans = 1x1 cell array
    {[327.2000]}
```

Input Arguments

M — Input Map object
Map object

Input Map object.

keySet — Keys of key-value pairs to remove
numeric scalar | character vector | string scalar | cell array

Keys of the key-value pairs to remove from the Map object, specified as a numeric scalar, character vector, string scalar, or cell array. To remove multiple key-value pairs, specify keySet as a cell array—even when you specify the keys as numeric scalars or strings.

See Also

containers.Map | isKey | keys | values

Topics

"Overview of Map Data Structure"
"Examine Contents of Map"
"Read and Write Using Key Index"
"Modify Keys and Values in Map"

Introduced in R2008b

removecats

Remove categories from categorical array

Syntax

```
B = removecats(A)
B = removecats(A,oldcats)
```

Description

`B = removecats(A)` removes unused categories from the categorical array, `A`. The output categorical array, `B`, has the same size and values as `A`. However, `B` possibly has fewer categories.

`B = removecats(A,oldcats)` removes the categories specified by `oldcats`. The function `removecats` removes categories, but does not remove any elements of the array. Therefore, elements of `B`, whose values correspond to `oldcats`, are undefined.

Examples

Remove All Unused Categories

Create a categorical array representing political parties of four people.

```
A = categorical({'republican' 'democrat' 'democrat' 'republican'},...
    {'democrat' 'republican' 'independent'})
```

```
A = 1x4 categorical array
     republican      democrat      democrat      republican
```

`A` is a 1-by-4 categorical array.

Summarize the categorical array, `A`.

```
summary(A)
```

```
     democrat      republican      independent
            2               2                0
```

A has three categories. `democrat` appears twice in the array, `republican` appears twice in the array, and `independent` is unused.

Remove the unused category, `independent`.

```
B = removecats(A)
```

```
B = 1x4 categorical array
     republican      democrat      democrat      republican
```

B has the same values as A.

Display the categories of B.

```
categories(B)
```

```
ans = 2x1 cell array
    {'democrat'  }
    {'republican'}
```

B has fewer categories than A.

Remove Categories and Make Corresponding Values Undefined

Create a categorical array, A, containing modes of transportation.

```
A = categorical({'plane' 'car'; 'train' 'car'; 'plane' 'car'})
```

```
A = 3x2 categorical array
     plane      car
     train      car
     plane      car
```

A is a 3-by-2 categorical array.

Display the categories of A.

```
categories(A)
```

```
ans = 3x1 cell array
    {'car'  }
    {'plane'}
    {'train'}
```

A has three categories, `car`, `plane`, and `train`.

Remove the category, `train`.

```
B = removecats(A,'train')
```

```
B = 3x2 categorical array
     plane            car
     <undefined>      car
     plane            car
```

The element that was from the category `train` is now undefined.

Display the categories of B.

```
categories(B)
```

```
ans = 2x1 cell array
    {'car'  }
    {'plane'}
```

B has one fewer category than A.

Input Arguments

A — Categorical array
vector | matrix | multidimensional array

Categorical array, specified as a vector, matrix, or multidimensional array.

`oldcats` — Categories to remove
character vector | cell array of character vectors | string array

Categories to remove, specified as a character vector, a cell array of character vectors, or a string array. The default is all the unused categories from `A`.

Tips

- `~ismember(categories(A),unique(A))` returns logical `true` (`1`) for any unused category of `A`.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also
`addcats` | `categories` | `iscategory` | `mergecats` | `renamecats` | `reordercats` | `setcats` | `summary`

Introduced in R2013b

removeGroup

Package: `matlab.settings`

Remove settings group

Syntax

```
removeGroup(parentgroup,name)
```

Description

`removeGroup(parentgroup,name)` removes a settings group called `name` from the specified parent settings group. If `name` contains settings or settings groups, `removeGroup` removes them as well.

Examples

Remove Settings Group

Create and then remove a settings group.

Use the `settings` function to access the root of the settings tree and then create the settings group `mysettings`.

```
s = settings;
addGroup(s,'mysettings');
s
```

```
s = 
  SettingsGroup with properties:
              mysettings: [1×1 SettingsGroup]
                  matlab: [1×1 SettingsGroup]
```

Remove the settings group `mysettings`.

```
removeGroup(s,'mysettings');
s
```

```
s = 
  SettingsGroup with properties:
                   matlab: [1×1 SettingsGroup]
```

Input Arguments

parentgroup — Parent settings group
SettingsGroup object

Parent settings group to remove the group from, specified as a SettingsGroup object. Use the settings function to access the root settings group object and all the available settings groups.

name — Name of settings group to remove
character vector | string scalar

Name of the settings group to remove, specified as a character vector or string scalar. If a group called name does not exist in the specified settings group, MATLAB throws an error.

See Also

addGroup | hasGroup | removeSetting | settings

Topics

"Access and Modify Settings"
"Create Custom Settings"

Introduced in R2019b

removeSetting

Package: `matlab.settings`

Remove setting

Syntax

```
removeSetting(parentgroup,name)
```

Description

`removeSetting(parentgroup,name)` removes a setting called `name` from the specified parent settings group.

Examples

Remove Setting

Create a settings group and add a new setting to the group. Then, remove the setting.

Use the `settings` function to access the root of the settings tree and then create the settings group `mysettings`.

```
s = settings;
addGroup(s,'mysettings');
```

Add the setting `MyWorkAddress` to `mysettings`.

```
addSetting(s.mysettings,'MyWorkAddress','PersonalValue','3 Apple Hill Drive');
s.mysettings.MyWorkAddress
```

```
ans = 
  Setting 'mysettings.MyWorkAddress' with properties:
       ActiveValue: '3 Apple Hill Drive'
    TemporaryValue: <no value>
```

```
     PersonalValue: '3 Apple Hill Drive'
      FactoryValue: <no value>
```

Remove the setting `MyWorkAddress` from `mysettings`.

```
removeSetting(s.mysettings,'MyWorkAddress');
s.mysettings
```

```
ans = 
  SettingsGroup 'mysettings' with no properties.
```

Input Arguments

`parentgroup` — Parent settings group
`SettingsGroup` object

Parent settings group to remove the setting from, specified as a `SettingsGroup` object. Use the `settings` function to access the root settings group object and all the available settings groups.

`name` — Name of setting to remove
character vector | string scalar

Name of the setting to remove, specified as a character vector or string scalar. If `name` does not exist in the specified settings group, MATLAB throws an error.

See Also

`addSetting` | `hasSetting` | `removeGroup` | `settings`

Topics

"Access and Modify Settings"
"Create Custom Settings"

Introduced in R2019b

removeStyle

Package: `matlab.ui.control`

Remove style from table UI component

Syntax

```
removeStyle(uit)
removeStyle(uit,ordernum)
```

Description

`removeStyle(uit)` removes all styles created with the `uistyle` function from the specified table UI component. To determine which styles are on `uit` and available to remove, query the value of `uit.StyleConfigurations`.

`removeStyle(uit,ordernum)` specifies which style to remove. Specify a style based on the order in which it was added. The property `uit.StyleConfigurations` lists styles in the order that they were added.

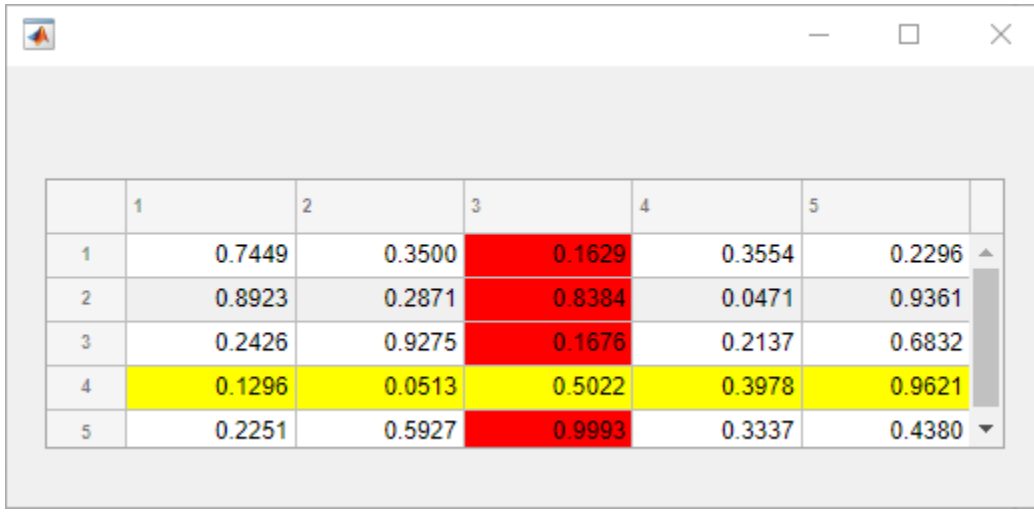
Examples

Remove All Styles from Table

First, add two styles to a table UI component.

```
fig = uifigure;
fig.Position = [500 500 520 220];

uit = uitable(fig);
uit.Data = rand(5);
uit.Position = [20 30 480 135];

s1 = uistyle('BackgroundColor','red');
```

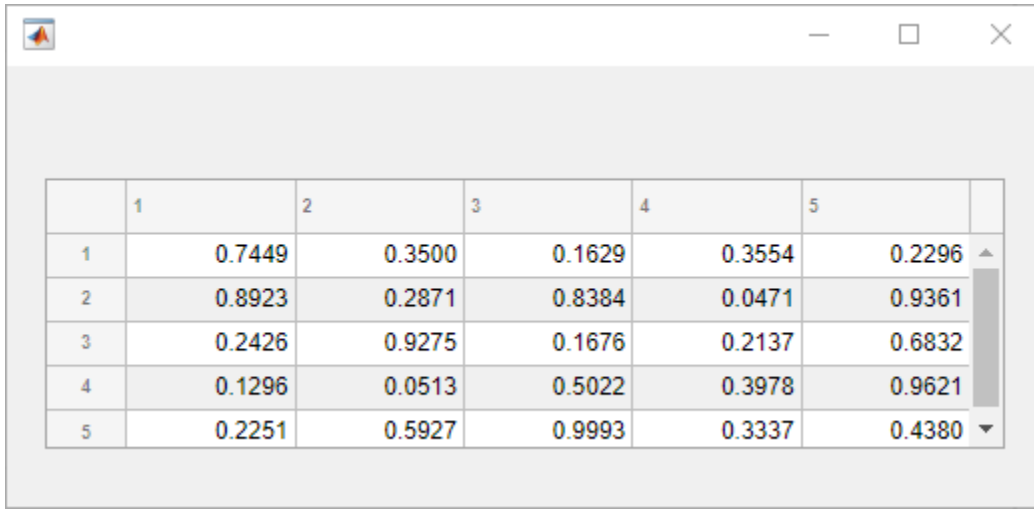
```
addStyle(uit,s1,'column',3)

s2 = uistyle('BackgroundColor','yellow');
addStyle(uit,s2,'row',4)
```



	1	2	3	4	5
1	0.7449	0.3500	0.1629	0.3554	0.2296
2	0.8923	0.2871	0.8384	0.0471	0.9361
3	0.2426	0.9275	0.1676	0.2137	0.6832
4	0.1296	0.0513	0.5022	0.3978	0.9621
5	0.2251	0.5927	0.9993	0.3337	0.4380

Then, remove both styles to revert the table UI component back to its default appearance.

```
removeStyle(uit)
```



	1	2	3	4	5
1	0.7449	0.3500	0.1629	0.3554	0.2296
2	0.8923	0.2871	0.8384	0.0471	0.9361
3	0.2426	0.9275	0.1676	0.2137	0.6832
4	0.1296	0.0513	0.5022	0.3978	0.9621
5	0.2251	0.5927	0.9993	0.3337	0.4380

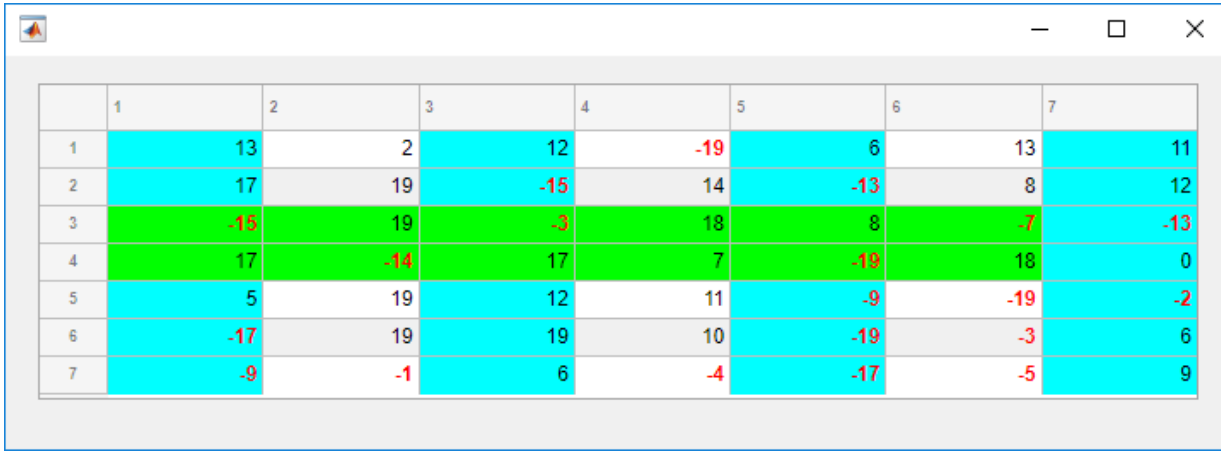
Remove Subset of Styles from Table

Add multiple styles to a table UI component, and then remove some of them.

First, create a table UI component and add styles to different parts of it.

```
fig = uifigure;
fig.Position = [500 500 720 230];
uit = uitable(fig);
uit.Data = randi([-20,20],7);
uit.Position = [20 30 680 185];

[row,col] = find(uit.Data<0);

s1 = uistyle;
s1.BackgroundColor = 'cyan';
addStyle(uit,s1,'column',[1 3 5])

s2 = uistyle;
s2.FontColor = 'red';
s2.FontWeight = 'bold';
addStyle(uit,s2,'cell',[row,col])
```

```
s3 = uistyle('BackgroundColor','green');
addStyle(uit,s3,'row',[3 4])

addStyle(uit,s1,'column',7)
```



	1	2	3	4	5	6	7
1	13	2	12	-19	6	13	11
2	17	19	-15	14	-13	8	12
3	-15	19	-3	18	8	-7	-13
4	17	-14	17	7	-19	18	0
5	5	19	12	11	-9	-19	-2
6	-17	19	19	10	-19	-3	6
7	-9	-1	6	-4	-17	-5	9

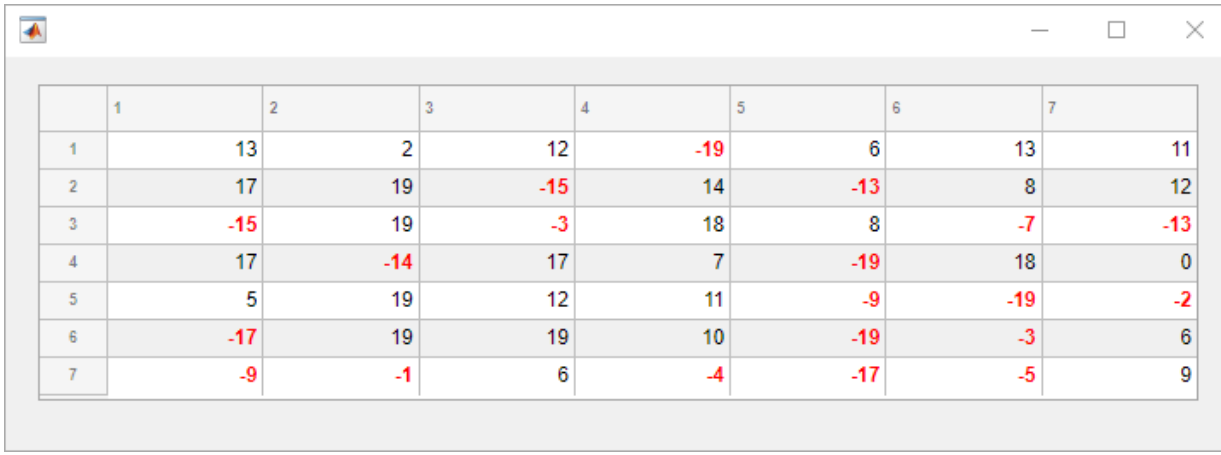
Now, remove the row and column styles. First, query the value of the `StyleConfigurations` property for the table.

```
uit.StyleConfigurations
```

```
ans=4×3 table
         Target      TargetIndex                 Style
         ______     _____________    ____________________________

    1    column     { 1x3 double}    [1x1 matlab.ui.style.Style]
    2    cell       {20x2 double}    [1x1 matlab.ui.style.Style]
    3    row        { 1x2 double}    [1x1 matlab.ui.style.Style]
    4    column     {[         7]}    [1x1 matlab.ui.style.Style]
```

The `StyleConfigurations` property value shows that style order numbers 1 and 4 affect columns, and that the row style was the third style added to the table. Remove the styles by specifying style order numbers 1, 3, and 4.

```
removeStyle(uit,[1 3 4])
```



	1	2	3	4	5	6	7
1	13	2	12	-19	6	13	11
2	17	19	-15	14	-13	8	12
3	-15	19	-3	18	8	-7	-13
4	17	-14	17	7	-19	18	0
5	5	19	12	11	-9	-19	-2
6	-17	19	19	10	-19	-3	6
7	-9	-1	6	-4	-17	-5	9

Input Arguments

`uit` — Table component
`Table` object

Table component, specified as a `Table` object created with the `uitable` function. The `Table` object must be parented to a figure created with the `uifigure` function, or one of its child containers.

`ordernum` — Style order number
positive integer | vector of positive integers

Style order number, specified as a positive integer or a vector of positive integers. To determine the styles currently applied to the table, and the order in which they were added, query the value of the `StyleConfigurations` property.

When you remove a style other than the last one that was added, the remaining styles move up in the order to close the gaps. If no style order number is specified, all styles are removed from the table.

Example: `removeStyle(uit,2)` removes the second style in the list returned by `uit.StyleConfigurations`.

Example: `removeStyle(uit,[1 3 5])` removes the first, third, and fifth styles in the list returned by `uit.StyleConfigurations`.

Example: `removeStyle(uit)` removes all styles from the table UI component.

See Also

Functions
`addStyle` | `uistyle` | `uitable`

Properties
Style Properties | Table Properties

Introduced in R2019b

removeToolbarExplorationButtons

Remove data exploration buttons from figure toolbar

Syntax

```
removeToolbarExplorationButtons(fig)
```

Description

`removeToolbarExplorationButtons(fig)` removes any data exploration buttons on page 1-13341 from the figure toolbar of the specified figure.

In R2018b, the data exploration buttons were moved from the figure toolbar to the axes toolbar. In most cases, you do not need to use this function. However, if you have code that relies on the buttons appearing in the figure toolbar, you can use the `addToolbarExplorationButtons` and `removeToolbarExplorationButtons` functions to control the appearance of the buttons in the figure toolbar.

Examples

Add and Remove Data Exploration Buttons

Create a figure with a surface plot. Then add the data exploration buttons to the figure toolbar.

```
fig = figure;
surf(peaks);
addToolbarExplorationButtons(fig)
```

Notice that the figure toolbar now includes buttons to zoom in, zoom out, and so on.

Remove the buttons from the figure toolbar.

```
removeToolbarExplorationButtons(fig)
```

Input Arguments

`fig` — Target figures
single `Figure` object | vector of `Figure` objects

Target figures, specified as a single `Figure` object or a vector of `Figure` objects.

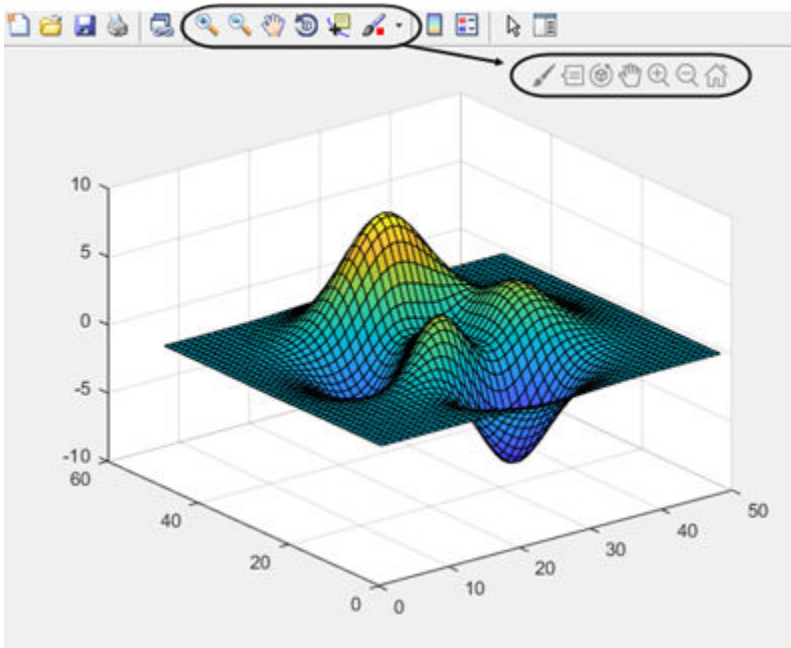
More About

Data Exploration Buttons

The standard data exploration buttons include options to:

- Zoom in or out of the axes view
- Pan the axes view
- Rotate the axes view
- Show data tips
- Brush data

In R2018b, the data exploration buttons were moved from the figure toolbar to the axes toolbar. This figure illustrates the relocation of the buttons.



See Also

addToolbarExplorationButtons | axtoolbar

Introduced in R2018a

removets

Remove `timeseries` from `tscollection`

Syntax

```
tscout = removets(tscin,tsname)
```

Description

`tscout = removets(tscin,tsname)` removes a `timeseries` object with name `tsname` from a `tscollection` object. `tsname` can be a single character vector or a cell array of character vectors.

Examples

Remove timeseries

Create a `tscollection` object from two `timeseries` objects. Then, remove the `timeseries` named `Speed` from the `tscollection`.

```
ts1 = timeseries([1.1 2.9 3.7 4.0 3.0]',1:5,...
                 'Name','Acceleration');
ts2 = timeseries([3.2 4.2 6.2 8.5 1.1]',1:5,...
                 'Name','Speed');
tscin = tscollection({ts1;ts2});
tscout = removets(tscin,'Speed')
```

```
Time Series Collection Object: unnamed

Time vector characteristics

      Start time            1 seconds
      End time              5 seconds

Member Time Series Objects:
```

```
Acceleration
```

Input Arguments

`tscin` — Input `tscollection`
scalar

Input `tscollection`, specified as a scalar.

`tsname` — `timeseries` name
character vector | cell array of character vectors

`timeseries` name, specified as a character vector or a cell array of character vectors.

See Also

`addts` | `timeseries` | `tscollection`

Introduced before R2006a

removevars

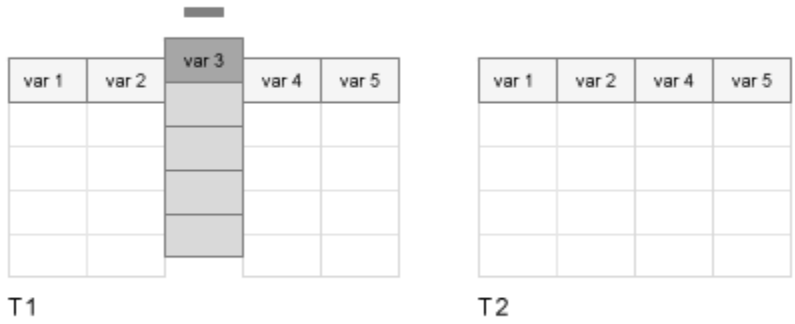
Delete variables from table or timetable

Syntax

```
T2 = removevars(T1,vars)
```

Description

`T2 = removevars(T1,vars)` deletes the table variables specified by `vars` and copies the remaining variables to `T2` (see diagram). You can specify variables by name, by position, or using logical indices.



Examples

Remove Variable

Create a table and remove variables one at a time. You can specify variables by name or by position in the table.

Read data from a spreadsheet into a table. Display the first three rows.

```
T1 = readtable('outages.csv');
head(T1,3)
```

```
ans=3×6 table
       Region           OutageTime        Loss     Customers     RestorationTime
    _____________    ________________    ______    __________    ________________    _

    {'SouthWest'}    2002-02-01 12:18    458.98    1.8202e+06    2002-02-07 16:50    {
    {'SouthEast'}    2003-01-23 00:49    530.14    2.1204e+05                NaT    {
    {'SouthEast'}    2003-02-07 21:15     289.4    1.4294e+05    2003-02-17 08:14    {
```

Remove the variable that is named `Region`.

```
T2 = removevars(T1,'Region');
head(T2,3)
```

```
ans=3×5 table
       OutageTime        Loss     Customers     RestorationTime          Cause
    ________________    ______    __________    ________________    ________________

    2002-02-01 12:18    458.98    1.8202e+06    2002-02-07 16:50    {'winter storm'}
    2003-01-23 00:49    530.14    2.1204e+05                NaT    {'winter storm'}
    2003-02-07 21:15     289.4    1.4294e+05    2003-02-17 08:14    {'winter storm'}
```

Remove the fourth variable from `T2`.

```
T3 = removevars(T2,4);
head(T3,3)
```

```
ans=3×4 table
       OutageTime        Loss     Customers          Cause
    ________________    ______    __________    ________________

    2002-02-01 12:18    458.98    1.8202e+06    {'winter storm'}
    2003-01-23 00:49    530.14    2.1204e+05    {'winter storm'}
    2003-02-07 21:15     289.4    1.4294e+05    {'winter storm'}
```

Specify Multiple Variables

Remove multiple table variables using the `removevars` function. You can specify variables by name or by position.

Read data from a spreadsheet into a table.

```
T1 = readtable('outages.csv');
head(T1,3)
```

```
ans=3×6 table
       Region           OutageTime        Loss     Customers     RestorationTime
    _____________    ________________    ______    __________    _________________

    {'SouthWest'}    2002-02-01 12:18    458.98    1.8202e+06    2002-02-07 16:50    {
    {'SouthEast'}    2003-01-23 00:49    530.14    2.1204e+05                 NaT    {
    {'SouthEast'}    2003-02-07 21:15     289.4    1.4294e+05    2003-02-17 08:14    {
```

Remove the variables named `Loss` and `Customers`. Specify names using a cell array of character vectors.

```
T2 = removevars(T1,{'Loss','Customers'});
head(T2,3)
```

```
ans=3×4 table
       Region           OutageTime       RestorationTime          Cause
    _____________    ________________    _________________    _________________

    {'SouthWest'}    2002-02-01 12:18    2002-02-07 16:50    {'winter storm'}
    {'SouthEast'}    2003-01-23 00:49                 NaT    {'winter storm'}
    {'SouthEast'}    2003-02-07 21:15    2003-02-17 08:14    {'winter storm'}
```

Remove the first and fourth variables, using a numeric array to indicate their positions in `T2`.

```
T3 = removevars(T2,[1 4]);
head(T3,3)
```

```
ans=3×2 table
       OutageTime       RestorationTime
```

```
_________________    _________________

2002-02-01 12:18    2002-02-07 16:50
2003-01-23 00:49                 NaT
2003-02-07 21:15    2003-02-17 08:14
```

Input Arguments

T1 — Input table
table | timetable

Input table, specified as a table or timetable.

vars — Variables in input table
character vector | cell array of character vectors | string array | numeric array | logical array

Variables in the input table, specified as a character vector, cell array of character vectors, string array, numeric array, or logical array.

Example: `T2 = removevars(T1,2)` removes the second table variable.

Example: `T2 = removevars(T1,'Date')` removes the table variable named `Date`.

Example: `T2 = removevars(T1,{'Latitude','Longitude','Elevation'})` removes the table variables named `Latitude`, `Longitude`, and `Elevation`.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`addvars` | `mergevars` | `movevars` | `splitvars`

Topics

“Add, Delete, and Rearrange Table Variables”
“Add and Delete Table Rows”
“Access Data in Tables”
“Modify Units, Descriptions, and Table Variable Names”
“Clean Messy and Missing Data in Tables”

Introduced in R2018a

renamecats

Rename categories in categorical array

Syntax

```
B = renamecats(A,newnames)
B = renamecats(A,oldnames,newnames)
```

Description

`B = renamecats(A,newnames)` renames all the categories in the categorical array, `A`. Elements of `B` use the new category names.

`B = renamecats(A,oldnames,newnames)` renames only the categories specified by `oldnames`.

Examples

Rename All Categories

Create a categorical array containing states from New England.

```
A = categorical({'MA';'ME';'CT';'VT';'ME';'NH';'VT';'MA';'NH';'CT';'RI'})
```

```
A = 11x1 categorical array
     MA
     ME
     CT
     VT
     ME
     NH
     VT
     MA
     NH
     CT
```

```
     RI
```

A is an 11-by-1 categorical array.

Display the categories of A.

```
categories(A)
```

```
ans = 6x1 cell array
    {'CT'}
    {'MA'}
    {'ME'}
    {'NH'}
    {'RI'}
    {'VT'}
```

A has six categories.

Rename the categories to use the full state name instead of the abbreviation.

```
B = renamecats(A,{'Connecticut','Massachusetts',...
    'Maine','New Hampshire','Rhode Island' 'Vermont'})
```

```
B = 11x1 categorical array
     Massachusetts
     Maine
     Connecticut
     Vermont
     Maine
     New Hampshire
     Vermont
     Massachusetts
     New Hampshire
     Connecticut
     Rhode Island
```

Elements of B use the new category names.

Display the categories of B.

```
categories(B)
```

```
ans = 6x1 cell array
    {'Connecticut'  }
    {'Massachusetts'}
    {'Maine'        }
    {'New Hampshire'}
    {'Rhode Island' }
    {'Vermont'      }
```

Rename One Category

Create a categorical array containing colors.

```
A = categorical({'red' 'blue'; 'purple' 'white'; 'green' 'red'})
```

```
A = 3x2 categorical array
     red         blue
     purple      white
     green       red
```

A is a 3-by-2 categorical array.

Display the categories of A.

```
categories(A)
```

```
ans = 5x1 cell array
    {'blue'  }
    {'green' }
    {'purple'}
    {'red'   }
    {'white' }
```

A has five categories that are listed in alphabetical order.

Change the category name from `purple` to `violet`.

```
B = renamecats(A,'purple','violet')
```

```
B = 3x2 categorical array
     red         blue
```

```
     violet      white
     green       red
```

The element `B(2,1)` is `violet` instead of `purple`.

Display the categories of `B`.

```
categories(B)
```

```
ans = 5x1 cell array
    {'blue'  }
    {'green' }
    {'violet'}
    {'red'   }
    {'white' }
```

`violet` replaces `purple` and the categories are no longer in alphabetical order. Note that the category has not changed its position.

Input Arguments

`A` — Categorical array
vector | matrix | multidimensional array

Categorical array, specified as a vector, matrix, or multidimensional array.

`newnames` — New category names for `B`
character vector | cell array of character vectors | string array

New category names for `B`, specified as a character vector, a cell array of character vectors, or a string array. The new category names must be unique, and must not duplicate any existing names.

`oldnames` — Old category names from `A`
character vector | cell array of character vectors | string array

Old category names from `A`, specified as a character vector, a cell array of character vectors, or a string array.

Tips

- Renaming categories does not change their position in `categories(B)`. Use `reordercats` to change the category ordering. For example, you can use `B = reordercats(B,sort(categories(B)))` to put the categories in alphabetical order.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`addcats` | `categories` | `iscategory` | `mergecats` | `removecats` | `reordercats` | `setcats`

Introduced in R2013b

reordercats

Reorder categories in categorical array

Syntax

```
B = reordercats(A)
B = reordercats(A,neworder)
```

Description

`B = reordercats(A)` reorders the categories in the categorical array, `A`, to be in alphanumeric order.

The order of the categories is used by functions such as `summary` and `histogram`. If the categorical array is ordinal, the order of the categories defines their mathematical ordering. The first category specified is the smallest and the last category is the largest.

`B = reordercats(A,neworder)` puts the categories in the order specified by `neworder`.

Examples

Alphabetize Categories of Nonordinal Categorical Array

Create two categorical arrays, `X` and `Y`.

```
X = categorical({'frog';'cat';'cat';'ant';'frog'})
```

```
X = 5x1 categorical array
     frog
     cat
     cat
     ant
     frog
```

```
Y = categorical({'deer';'bear';'eagle';'deer'})
```

```
Y = 4x1 categorical array
     deer
     bear
     eagle
     deer
```

X is a 5-by-1 categorical array. The categories of X are the sorted unique values from the array: `{'ant';'cat';'frog'}`.

Y is a 4-by-1 categorical array. The categories of Y are the sorted unique values from the array: `{'bear';'deer';'eagle'}`.

Concatenate X and Y into a single categorical array, A.

```
A = [X;Y]
```

```
A = 9x1 categorical array
     frog
     cat
     cat
     ant
     frog
     deer
     bear
     eagle
     deer
```

`vertcat` appends the values from Y to the values from X.

List the categories of the categorical array, A.

```
acats = categories(A)
```

```
acats = 6x1 cell array
    {'ant'  }
    {'cat'  }
    {'frog' }
    {'bear' }
    {'deer' }
    {'eagle'}
```

`vertcat` appends the categories of `Y` to the categories from `X`. The categories of `A` are *not* in alphabetical order.

Reorder the categories of `A` into alphabetical order.

```
B = reordercats(A)
```

```
B = 9x1 categorical array
     frog
     cat
     cat
     ant
     frog
     deer
     bear
     eagle
     deer
```

The output categorical array, `B`, has the same elements in the same order as the input categorical array, `A`.

List the categories of the categorical array, `B`.

```
bcats = categories(B)
```

```
bcats = 6x1 cell array
    {'ant'  }
    {'bear' }
    {'cat'  }
    {'deer' }
    {'eagle'}
    {'frog' }
```

The categories of `B` are in alphabetical order.

Reorder Categories in Nonordinal Categorical Array

Create a categorical array containing the color of various items.

```
A = categorical({'red';'green';'blue';'red';'green';'red';'blue';'blue'})
```

```
A = 8x1 categorical array
     red
     green
     blue
     red
     green
     red
     blue
     blue
```

A is an 8-by-1 categorical array.

Display the categories of A.

```
categories(A)
```

```
ans = 3x1 cell array
    {'blue' }
    {'green'}
    {'red'  }
```

The categories of A are in alphabetical order and have no mathematical meaning.

Reorder the categories to match the order commonly used for colors.

```
B = reordercats(A,{'red','green','blue'})
```

```
B = 8x1 categorical array
     red
     green
     blue
     red
     green
     red
     blue
     blue
```

B contains the same values as A.

Display the categories of B.

```
categories(B)
```

```
ans = 3x1 cell array
    {'red'  }
    {'green'}
    {'blue' }
```

B is not ordinal and the order of the categories has no mathematical meaning. Although the categories appear in the order of the color spectrum, relational operations, such as greater than and less than, have no meaning.

Reorder Categories in Ordinal Categorical Array

Create an ordinal categorical array, A, containing modes of transportation. Order the categories based on the average price of travel.

```
A = categorical({'plane';'car'; 'train';'car';'plane';'car'},...
    {'car','train','plane'},'Ordinal',true)
```

```
A = 6x1 categorical array
     plane
     car
     train
     car
     plane
     car
```

A is a 6-by-1 ordinal categorical array.

Display the categories of A.

```
categories(A)
```

```
ans = 3x1 cell array
    {'car'  }
    {'train'}
    {'plane'}
```

Since A is ordinal, `car < train < plane`.

Reorder the categories to reflect a decrease in the cost of train travel.

```
B = reordercats(A,{'train','car','plane'})
```

```
B = 6x1 categorical array
     plane
     car
     train
     car
     plane
     car
```

B contains the same values as A.

Display the categories of B.

```
categories(B)
```

```
ans = 3x1 cell array
    {'train'}
    {'car'  }
    {'plane'}
```

The mathematical ordering of the categories is now `train < car < plane`. The results from relational operations, `min`, and `max` reflect the new category ordering.

Reorder Categories with Numeric Vector

Create a categorical array, A, containing modes of transportation.

```
A = categorical({'plane';'car';'train';'car';'car';'plane';'car'})
```

```
A = 7x1 categorical array
     plane
     car
     train
     car
     car
     plane
     car
```

Display the categories of A.

```
categories(A)
```

```
ans = 3x1 cell array
    {'car'  }
    {'plane'}
    {'train'}
```

Reorder categories from least to most frequent occurrence in A.

```
B = countcats(A);
[C,neworder] = sort(B);
neworder
```

```
neworder = 3×1

     3
     2
     1
```

```
D = reordercats(A,neworder);
categories(D)
```

```
ans = 3x1 cell array
    {'train'}
    {'plane'}
    {'car'  }
```

Because `countcats` counts the occurrences of each category, `neworder` describes how to reorder the categories—not the elements—of A.

Input Arguments

A — Categorical array

vector | matrix | multidimensional array

Categorical array, specified as a vector, matrix, or multidimensional array. If A is an ordinal categorical array, a reordering of the categories changes the mathematical meaning. Consequently, the relational operators, such as greater than and less than, might return different results.

`neworder` — New category order for `B`
cell array of character vectors | string array | numeric vector

New category order for `B`, specified as a cell array of character vectors, a string array, or a numeric vector. `neworder` must be a permutation of `categories(A)`.

Tips

- To convert the categorical array, `B`, to an ordinal categorical array, use `B = categorical(B,'Ordinal',true)`. You can specify the order of the categories with `B = categorical(B,`*`valueset`*`,'Ordinal',true)`, where the order of the values in *`valueset`* defines the category order.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also
`addcats` | `categories` | `iscategory` | `mergecats` | `removecats` | `renamecats` | `setcats`

Introduced in R2013b

rename

Rename file on FTP server

Syntax

```
rename(ftpobj,oldname,newname)
```

Description

`rename(ftpobj,oldname,newname)` changes the name of the file `oldname` to `newname` in the current folder on the FTP server associated with `ftpobj`.

Examples

Rename File

Connect to an FTP server and rename a file on the server. This example shows a hypothetical FTP session on `ftp.example.com`, a machine that does not exist. If you have an account on an FTP server that grants you permission to upload or change files on that server, then you can use the `rename` function as shown in this example.

First, connect to the server.

```
ftpobj = ftp('ftp.example.com')
```

```
ftpobj = 

  FTP Object
     host: ftp.example.com
     user: anonymous
      dir: /
     mode: binary
```

Display the contents of the current folder on the FTP server.

```
dir(ftpobj)
```

```
myscript.m                  README.txt                pub
```

Rename a file on the FTP server.

```
rename(ftpobj,'README.txt','INTRO.txt')
```

Display the updated contents on the FTP server, including the renamed file.

```
dir(ftpobj)
```

```
INTRO.txt                  myscript.m                pub
```

Input Arguments

`ftpobj` — Connection to FTP server
FTP object

Connection to an FTP server, specified as an FTP object.

`oldname` — Original name of file
character vector | string scalar

Original name of the file on the FTP server, specified as a character vector or string scalar.

`newname` — New name of file
character vector | string scalar

New name of the file on the FTP server, specified as a character vector or string scalar.

Tips

- The `rename` function only renames files on an FTP server. To rename files on your local machine, use the `movefile` function.

See Also

`delete` | `dir` | `ftp` | `mget` | `mput`

Introduced before R2006a

repelem

Repeat copies of array elements

Syntax

```
u = repelem(v,n)
B = repelem(A,r1,...,rN)
```

Description

`u = repelem(v,n)`, where `v` is a scalar or vector, returns a vector of repeated elements of `v`.

- If `n` is a scalar, then each element of `v` is repeated `n` times. The length of `u` is `length(v)*n`.
- If `n` is a vector, then it must be the same length as `v`. Each element of `n` specifies the number of times to repeat the corresponding element of `v`.

This syntax is not supported for `table` input.

`B = repelem(A,r1,...,rN)` returns an array with each element of `A` repeated according to `r1,...,rN`. Each `r1,...,rN` must either be a scalar or a vector with the same length as `A` in the corresponding dimension. For example, if `A` is a matrix, `repelem(A,2,3)` returns a matrix containing a 2-by-3 block of each element of `A`.

Examples

Repeat Vector Elements

Create a vector and repeat each of its elements three times into a new vector.

```
v = [1 2 3 4];
u = repelem(v,3)
```

```
u = 1×12

     1     1     1     2     2     2     3     3     3     4     4     4
```

Repeat the first two elements of v twice and the last two elements three times.

```
u = repelem(v,[2 2 3 3])
```

```
u = 1×10

     1     1     2     2     3     3     3     4     4     4
```

Repeat Matrix Elements

Create a matrix and repeat each element into a 3-by-2 block of a new matrix.

```
A = [1 2; 3 4]
```

```
A = 2×2

     1     2
     3     4
```

```
B = repelem(A,3,2)
```

```
B = 6×4

     1     1     2     2
     1     1     2     2
     1     1     2     2
     3     3     4     4
     3     3     4     4
     3     3     4     4
```

Repeat Matrix Columns

Create a matrix and copy its columns into a new array, repeating the first column twice and second column three times.

```
A = [1 2; 3 4]
```

```
A = 2×2

     1     2
     3     4
```

```
B = repelem(A,1,[2 3])
```

```
B = 2×5

     1     1     2     2     2
     3     3     4     4     4
```

Input Arguments

v — Input element
scalar | vector

Input element, specified as a scalar or a vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `datetime` | `duration`
Complex Number Support: Yes

n — Number of times to repeat each element
scalar | vector

Number of times to repeat each element, specified as a scalar or a vector. If `n` is a scalar, then all elements of `v` are repeated `n` times. If `n` is a vector, then each element of `n` specifies the number of times to repeat the corresponding element of `v`. In either case, `n` must be integer-valued.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

A — Input array
matrix | multidimensional array

Input array, specified as a matrix or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell` | `datetime` | `duration`
Complex Number Support: Yes

`r1,...,rN` — Repetition factors for each dimension (as separate arguments)
scalars | vectors

Repetition factors for each dimension, specified as separate arguments of integer-valued scalars or vectors. If `A` is a table, each repetition factor must be a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`u` — Output vector
vector

Output vector. If `v` is a row vector or scalar, `u` is a row vector. If `v` is a column vector, `u` is also a column vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `datetime` | `duration`

`B` — Output array
matrix | multidimensional array

Output array, returned as a matrix or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `datetime` | `duration`

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

- The two-input syntax is not supported.
- The replication factor in the first dimension must be 1. For example, `repelem(TA,1,n,p,...)`.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The input must be a vector or matrix. The input cannot be a multidimensional array.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- The input must be a vector or matrix. The input cannot be a multidimensional array.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`kron` | `repmat`

Introduced in R2015a

replace

Find and replace one or more substrings

Syntax

```
newStr = replace(str,old,new)
```

Description

`newStr = replace(str,old,new)` replaces all occurrences of the substring `old` with `new`. If `old` contains multiple substrings, then `new` either must be the same size as `old`, or must be a single substring.

Examples

Replace Substring in String Array

Replace placeholder text in a list of file names.

Starting in R2017a, you can create strings using double quotes.

```
str = ["<ROOT_DIR>\MyData\data.tar.gz";
       "<ROOT_DIR>\MyScripts\cleandata.m";
       "<ROOT_DIR>\MyScripts\preprocess.m";
       "<ROOT_DIR>\MyScripts\publishResults.m"]
```

```
str = 4x1 string array
    "<ROOT_DIR>\MyData\data.tar.gz"
    "<ROOT_DIR>\MyScripts\cleandata.m"
    "<ROOT_DIR>\MyScripts\preprocess.m"
    "<ROOT_DIR>\MyScripts\publishResults.m"
```

Replace `<ROOT_DIR>` with a string that is the name of a file path.

```
old = "<ROOT_DIR>";
new = "C:\MyProject";
newStr = replace(str,old,new)
```

```
newStr = 4x1 string array
    "C:\MyProject\MyData\data.tar.gz"
    "C:\MyProject\MyScripts\cleandata.m"
    "C:\MyProject\MyScripts\preprocess.m"
    "C:\MyProject\MyScripts\publishResults.m"
```

Replace Multiple Substrings

Replace carriage returns with newline characters.

Starting in R2017a, you can create strings using double quotes.

```
str = ["Submission Date: 11/29/15\r";
       "Acceptance Date: 1/20/16\r";
       "Contact: john.smith@example.com\r\n"]
```

```
str = 3x1 string array
    "Submission Date: 11/29/15\r"
    "Acceptance Date: 1/20/16\r"
    "Contact: john.smith@example.com\r\n"
```

Replace the carriage returns.

```
old = {'\r\n','\r'};
new = '\n';
newStr = replace(str,old,new)
```

```
newStr = 3x1 string array
    "Submission Date: 11/29/15\n"
    "Acceptance Date: 1/20/16\n"
    "Contact: john.smith@example.com\n"
```

Input Arguments

`str` — Input text
string array | character vector | cell array of character vectors

Input text, specified as a string array, a character vector, or a cell array of character vectors.

Data Types: `string` | `char` | `cell`

`old` — Substring to replace
string array | character vector | cell array of character vectors

Substring to replace, specified as a string array, character vector, or cell array of character vectors.

Data Types: `string` | `char` | `cell`

`new` — New substring
string array | character vector | cell array of character vectors

New substring, specified as a string array, character vector, or cell array of character vectors.

Data Types: `string` | `char` | `cell`

Tips

- To perform multiple replacements for overlapping patterns, use the `strrep` function.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `str`, `old`, and `new` must be a string scalar, a character vector, or a cell array containing not more than one character vector.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`count` | `erase` | `eraseBetween` | `extractAfter` | `extractBefore` | `extractBetween` | `insertAfter` | `insertBefore` | `join` | `regexp` | `regexprep` | `replaceBetween` | `size` | `split` | `strlength` | `strrep`

Topics

“Create String Arrays”
“Search and Replace Text”
“Test for Empty Strings and Missing Values”

Introduced in R2016b

replaceBetween

Replace substrings between start and end points

Syntax

```
newStr = replaceBetween(str,startStr,endStr,newText)
newStr = replaceBetween(str,startPos,endPos,newText)
newStr = replaceBetween( ___ ,'Boundaries',bounds)
```

Description

`newStr = replaceBetween(str,startStr,endStr,newText)` replaces substrings in `str` with the text in `newText`. The substrings that are replaced occur between the substrings `startStr` and `endStr`. However, `replaceBetween` does not replace `startStr` and `endStr` themselves. `replaceBetween` returns the result as `newStr`. The `newText` argument can have a different number of characters than the substring it replaces.

If `str` is a string array or a cell array of character vectors, then `replaceBetween` replaces substrings in each element of `str`. The output argument `newStr` has the same data type as `str`.

`newStr = replaceBetween(str,startPos,endPos,newText)` replaces substrings in `str`. The substrings that are replaced occur between the positions `startPos` and `endPos` in `str`, including the characters at those positions.

`newStr = replaceBetween( ___ ,'Boundaries',bounds)` forces the starts and ends specified in any of the previous syntaxes to be either inclusive or exclusive. They are inclusive when `bounds` is `'inclusive'`, and exclusive when `bounds` is `'exclusive'`. For example, `replaceBetween(str,startStr,endStr,newText,'Boundaries','inclusive')` replaces `startStr`, `endStr`, and all the text between them with the text specified by `newText`.

Examples

Replace Text Between Substrings

Create string arrays and replace text that occurs between substrings.

Starting in R2017a, you can create strings using double quotes.

```
str = "The quick brown fox"
```

```
str =
"The quick brown fox"
```

Replace the text that occurs between the substrings `quick` and `fox`. The `replaceBetween` function replaces the text but does not replace `quick` or `fox` in the output.

```
newStr = replaceBetween(str,"quick "," fox","red")
```

```
newStr =
"The quick red fox"
```

Replace substrings from each element of a string array. When you specify different substrings as start and end indicators, they must be contained in a string array or a cell array that is the same size as `str`. The replacement text also must be in a string array or a cell array of the same size.

```
str = ["The quick brown fox jumps";"over the lazy dog"]
```

```
str = 2x1 string array
    "The quick brown fox jumps"
    "over the lazy dog"
```

```
newText = ["red";"sleeping"];
newStr = replaceBetween(str,["quick ";"the "],[" fox";" dog"],newText)
```

```
newStr = 2x1 string array
    "The quick red fox jumps"
    "over the sleeping dog"
```

Replace Substrings Between Start and End Positions

Create string arrays and replace substrings between start and end positions that are specified as numbers.

Starting in R2017a, you can create strings using double quotes. Create a string that contains a name. To replace the middle name, specify the seventh and 11th positions in the string.

```
str = "Edgar Allen Poe"
```

```
str =
"Edgar Allen Poe"
```

```
newStr = replaceBetween(str,7,11,'A.')
```

```
newStr =
"Edgar A. Poe"
```

Replace substrings from each element of a string array. When you specify different start and end positions with numeric arrays, they must be the same size as the input string array. The replacement text also must be in a string array or a cell array of the same size.

```
str = ["Edgar Allen Poe";"Louisa May Alcott"]
```

```
str = 2x1 string array
    "Edgar Allen Poe"
    "Louisa May Alcott"
```

```
newText = ["A.";"M."];
newStr = replaceBetween(str,[7;8],[11;10],newText)
```

```
newStr = 2x1 string array
    "Edgar A. Poe"
    "Louisa M. Alcott"
```

Replace Text Within Inclusive and Exclusive Boundaries

Replace text from string arrays within boundaries that are forced to be inclusive or exclusive. `replaceBetween` replaces the boundaries along with the text when the

boundaries are inclusive. `replaceBetween` does not replace the boundaries when the boundaries are exclusive.

Starting in R2017a, you can create strings using double quotes.

```
str = "small|medium|large"
```

```
str =
"small|medium|large"
```

Replace the text between sixth and 13th positions, but do not replace the characters at those positions.

```
newText = "regular";
newStr = replaceBetween(str,6,13,newText,'Boundaries','exclusive')
```

```
newStr =
"small|regular|large"
```

Replace the text between two substrings, and also the substrings themselves.

```
str = "The quick brown fox jumps over the lazy dog"
```

```
str =
"The quick brown fox jumps over the lazy dog"
```

```
newText = "red bird flies";
newStr = replaceBetween(str,'brown','jumps',newText,'Boundaries','inclusive')
```

```
newStr =
"The quick red bird flies over the lazy dog"
```

Replace Text Between Positions in Character Vector

Create a character vector and replace text between start and end positions.

```
chr = 'mushrooms, peppers, and onions'
```

```
chr =
'mushrooms, peppers, and onions'
```

```
newChr = replaceBetween(chr,12,18,'pineapple')
```

```
newChr =
'mushrooms, pineapple, and onions'
```

Replace text between substrings.

```
newChr = replaceBetween(chr,'mushrooms,',' and',' extra cheese,')
```

```
newChr =
'mushrooms, extra cheese, and onions'
```

Input Arguments

`str` — Input text

string array | character vector | cell array of character vectors

Input text, specified as a string array, a character vector, or a cell array of character vectors.

Data Types: `string` | `char` | `cell`

`startStr` — String that indicates start of substring to replace

string array | character vector | cell array of character vectors

String that indicates the start of the substring to replace, specified as a string array, a character vector, or a cell array of character vectors.

If `str` is a string array or a cell array of character vectors, then `startStr` can be a character vector, a string scalar, or a string array or a cell array of the same size as `str`.

Example: `replaceBetween(str,"AB","YZ",newText)` replaces the substring between `AB` and `YZ` in each element of `str` with the text specified by `newText`.

Example: If `str` is a 2-by-1 string array, then `replaceBetween(str,["AB";"FG"],["YZ";"ST"],newText)` replaces the substrings between `AB` and `YZ` in `str(1)`, and between `FG` and `ST` in `str(2)`.

Data Types: `string` | `char` | `cell`

`endStr` — String that indicates end of substring to replace

string array | character vector | cell array of character vectors

Substring that indicates the end of the text to replace, specified as a string array, a character vector, or a cell array of character vectors.

If `str` is a string array or a cell array of character vectors, then `endStr` can be a character vector, a string scalar, or a string array or a cell array of the same size as `str`.

Example: `replaceBetween(str,"AB","YZ",newText)` replaces the substring between AB and YZ in each element of `str` with the text specified by `newText`.

Example: If `str` is a 2-by-1 string array, then `replaceBetween(str,["AB";"FG"],["YZ";"ST"],newText)` replaces the substrings between AB and YZ in `str(1)`, and between FG and ST in `str(2)`.

Data Types: `string` | `char` | `cell`

`startPos` — Start position of substring to replace
numeric array

Start position of substring to replace, specified as a numeric array.

If `str` is a string array or a cell array of character vectors, then `startPos` can be a numeric scalar or a numeric array of the same size as `str`.

Example: `replaceBetween(str,5,9,newText)` replaces the substring from the fifth to the ninth positions in each element of `str` with the text specified by `newText`.

Example: If `str` is a 2-by-1 string array, then `replaceBetween(str,[5;10],[9;21],newText)` replaces the substring from the fifth through the ninth positions in `str(1)`, and the 10th through the 21st positions in `str(2)`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`endPos` — End position of substring to replace
numeric array

End position of substring to replace, specified as a numeric array.

If `str` is a string array or a cell array of character vectors, then `endPos` can be a numeric scalar or a numeric array of the same size as `str`.

Example: `replaceBetween(str,5,9,newText)` replaces the substring from the fifth to the ninth positions in each element of `str` with the text specified by `newText`.

Example: If `str` is a 2-by-1 string array, then `replaceBetween(str,[5;10],[9;21],newText)` replaces the substring from the fifth through the ninth positions in `str(1)`, and the 10th through the 21st positions in `str(2)`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`newText` — Replacement text
string array | character vector | cell array of character vectors

Replacement text, specified as a string array, a character vector, or a cell array of character vectors.

If `str` is a string array or a cell array of character vectors, then `newText` can be a character vector, a string scalar, or a string array or a cell array of the same size as `str`.

Example: `replaceBetween(str,"AB","YZ","efg")` replaces the substring between `AB` and `YZ` in each element of `str` with `efg`.

Example: If `str` is a 2-by-1 string array, then `replaceBetween(str,["AB";"FG"],["YZ";"ST"],["efg";"lmnop"])` replaces the substrings between `AB` and `YZ` in `str(1)` with `efg`, and between `FG` and `ST` in `str(2)` with `lmnop`.

Data Types: `string` | `char` | `cell`

Output Arguments

`newStr` — Output text
string array | character vector | cell array of character vectors

Output text, returned as a string array, a character vector, or a cell array of character vectors. `str` and `newStr` have the same data type.

Data Types: `string` | `char` | `cell`

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `str`, `startStr`, `endStr`, and `newText` must be a string scalar, a character vector, or a cell array containing not more than one character vector.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`join` | `replace` | `split`

Topics

“Create String Arrays”
“Search and Replace Text”
“Test for Empty Strings and Missing Values”

Introduced in R2016b

repmat

Repeat copies of array

Syntax

```
B = repmat(A,n)
B = repmat(A,r1,...,rN)
B = repmat(A,r)
```

Description

`B = repmat(A,n)` returns an array containing `n` copies of `A` in the row and column dimensions. The size of `B` is `size(A)*n` when `A` is a matrix.

`B = repmat(A,r1,...,rN)` specifies a list of scalars, `r1,..,rN`, that describes how copies of `A` are arranged in each dimension. When `A` has `N` dimensions, the size of `B` is `size(A).*[r1...rN]`. For example, `repmat([1 2; 3 4],2,3)` returns a 4-by-6 matrix.

`B = repmat(A,r)` specifies the repetition scheme with row vector `r`. For example, `repmat(A,[2 3])` returns the same result as `repmat(A,2,3)`.

Examples

Initialize Matrix with Same Element Value

Create a 3-by-2 matrix whose elements contain the value 10.

```
A = repmat(10,3,2)
```

```
A = 3×2

    10    10
    10    10
```

```
    10    10
```

Square Block Format

Repeat copies of a matrix into a 2-by-2 block arrangement.

```
A = diag([100 200 300])
```

```
A = 3×3

   100     0     0
     0   200     0
     0     0   300
```

```
B = repmat(A,2)
```

```
B = 6×6

   100     0     0   100     0     0
     0   200     0     0   200     0
     0     0   300     0     0   300
   100     0     0   100     0     0
     0   200     0     0   200     0
     0     0   300     0     0   300
```

Rectangular Block Format

Repeat copies of a matrix into a 2-by-3 block arrangement.

```
A = diag([100 200 300])
```

```
A = 3×3

   100     0     0
     0   200     0
     0     0   300
```

```
B = repmat(A,2,3)
```

```
B = 6×9

   100     0     0   100     0     0   100     0     0
     0   200     0     0   200     0     0   200     0
     0     0   300     0     0   300     0     0   300
   100     0     0   100     0     0   100     0     0
     0   200     0     0   200     0     0   200     0
     0     0   300     0     0   300     0     0   300
```

3-D Block Array

Repeat copies of a matrix into a 2-by-3-by-2 block arrangement.

```
A = [1 2; 3 4]
```

```
A = 2×2

     1     2
     3     4
```

```
B = repmat(A,[2 3 2])
```

```
B =
B(:,:,1) =

     1     2     1     2     1     2
     3     4     3     4     3     4
     1     2     1     2     1     2
     3     4     3     4     3     4


B(:,:,2) =

     1     2     1     2     1     2
     3     4     3     4     3     4
     1     2     1     2     1     2
     3     4     3     4     3     4
```

Vertical Stack of Row Vectors

Vertically stack a row vector four times.

```
A = 1:4;
B = repmat(A,4,1)
```

```
B = 4×4

     1     2     3     4
     1     2     3     4
     1     2     3     4
     1     2     3     4
```

Horizontal Stack of Column Vectors

Horizontally stack a column vector four times.

```
A = (1:3)';
B = repmat(A,1,4)
```

```
B = 3×4

     1     1     1     1
     2     2     2     2
     3     3     3     3
```

Tabular Block Format

Create a table with variables `Age` and `Height`.

```
A = table([39; 26],[70; 63],'VariableNames',{'Age' 'Height'})
```

```
A=2×2 table
    Age    Height
    ___    ______
```

```
    39        70
    26        63
```

Repeat copies of the table into a 2-by-3 block format.

```
B = repmat(A,2,3)
```

```
B=4×6 table
    Age    Height    Age_1    Height_1    Age_2    Height_2
    ___    ______    _____    ________    _____    ________

    39       70       39         70        39         70
    26       63       26         63        26         63
    39       70       39         70        39         70
    26       63       26         63        26         63
```

`repmat` repeats the entries of the table and appends a number to the new variable names.

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell`
Complex Number Support: Yes

n — Number of times to repeat input array in row and column dimensions
integer value

Number of times to repeat the input array in the row and column dimensions, specified as an integer value. If `n` is `0` or negative, the result is an empty array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`r1,...,rN` — Repetition factors for each dimension (as separate arguments)
integer values

Repetition factors for each dimension, specified as separate arguments of integer values. If any repetition factor is `0` or negative, the result is an empty array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`r` — Vector of repetition factors for each dimension (as a row vector)
integer values

Vector of repetition factors for each dimension, specified as a row vector of integer values. If any value in `r` is `0` or negative, the result is an empty array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Tips

- To build block arrays by forming the tensor product of the input with an array of ones, use `kron`. For example, to stack the row vector `A = 1:3` four times vertically, you can use `B = kron(A,ones(4,1))`.
- To create block arrays and perform a binary operation in a single pass, use `bsxfun`. In some cases, `bsxfun` provides a simpler and more memory efficient solution. For example, to add the vectors `A = 1:5` and `B = (1:10)'` to produce a 10-by-5 array, use `bsxfun(@plus,A,B)` instead of `repmat(A,10,1) + repmat(B,1,5)`.
- When `A` is a scalar of a certain type, you can use other functions to get the same result as `repmat`.

repmat Syntax	Equivalent Alternative
`repmat(NaN,m,n)`	`NaN(m,n)`
`repmat(single(inf),m,n)`	`inf(m,n,'single')`
`repmat(int8(0),m,n)`	`zeros(m,n,'int8')`
`repmat(uint32(1),m,n)`	`ones(m,n,'uint32')`
`repmat(eps,m,n)`	`eps(ones(m,n))`

Compatibility Considerations

Some repetition arguments produce error

Behavior changed in R2019b

Starting in R2019b, some syntaxes involving nonscalar or empty repetition arguments will produce an error. The following table describes how to update these syntaxes.

Errors	Use Instead
`repmat(A,r1,r2)`, where `r1` and `r2` are row vectors	`repmat(A,[r1 r2])`
`repmat(A,empt)`, where `empt` is an empty array	`repmat(A,1)`
`repmat(A,empt1,empt2)`, where `empt1` and `empt2` are empty arrays	`repmat(A,1)`
`repmat(A,n,empt)`, where `n` is an integer and `empt` is an empty array	`repmat(A,[n 1])`

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

- The replication factor in the first dimension must be 1. For example, `repmat(TA,1,n,p,...)`.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Size arguments must have a fixed size.
- For sparse matrices, the `repmat` function does not support trailing ones as inputs after the first two dimensions.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Size arguments must have a fixed size.
- For sparse matrices, the `repmat` function does not support trailing ones as inputs after the first two dimensions.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bsxfun` | `kron` | `meshgrid` | `ndgrid` | `repelem` | `reshape`

Introduced before R2006a

resample

Resample `tscollection` time vector

Syntax

```
tscout = resample(tscin,timevec)
tscout = resample(tscin,timevec,interpmethod)
tscout = resample(tscin,timevec,interpmethod,code)
```

Description

`tscout = resample(tscin,timevec)` resamples a `tscollection` object `tscin` using a new time vector `timevec`. The `resample` function uses the interpolation method associated with each `timeseries` in `tscin`.

`tscout = resample(tscin,timevec,interpmethod)` resamples `tscin` using a specified interpolation method. `interpmethod` can be `'linear'` for linear interpolation or `'zoh'` for zero-order hold.

`tscout = resample(tscin,timevec,interpmethod,code)` applies the quality codes in `code` to all samples.

Examples

Resample `tscollection`

Resample a `tscollection` object made up of two `timeseries` objects.

Create a `tscollection` object from two `timeseries` objects.

```
ts1=timeseries([1.1 2.9 3.7 4.0 3.0]',1:5,'Name','Acceleration');
ts2=timeseries([3.2 4.2 6.2 8.5 1.1]',1:5,'Name','Speed');
tscin = tscollection({ts1,ts2})
```

```
Time Series Collection Object: unnamed

Time vector characteristics

      Start time            1 seconds
      End time              5 seconds

Member Time Series Objects:

      Acceleration
      Speed
```

Resample the data in `tscin` according to a new time vector using zero-order hold interpolation.

```
tsout = resample(tscin,[1 1.5 3.5 4.5 4.9],'zoh')
```

```
Time Series Collection Object: unnamed

Time vector characteristics

      Start time            1 seconds
      End time              4.900000e+00 seconds

Member Time Series Objects:

      Acceleration
      Speed
```

Input Arguments

`tscin` — Input `tscollection`
scalar

Input `tscollection`, specified as a scalar.

`timevec` — New time vector
vector

New time vector, specified as a vector of times for resampling.

When the original time vector contains dates and times but `timevec` is numeric, `resample` defines `timevec` relative to the `tscin.TimeInfo.StartDate` property using the existing units.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`interpmethod` — Interpolation method
`'linear'` (default) | `'zoh'`

Interpolation method, specified as one of the following options:

- `'linear'` — Linear interpolation
- `'zoh'` — Zero-order hold

`code` — Quality codes
`[]` (default) | scalar | vector | multidimensional array

Quality codes, specified as `[]` or a scalar, vector, or multidimensional array of integers ranging from -128 to 127.

- When the quality code value is a vector, it must have the same length as the time vector. Each element applies to the corresponding data sample.
- When the quality code value is an array, it must have the same size as the data array. Each element applies to the corresponding data element of the data array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

See Also

`getinterpmethod` | `setinterpmethod` | `synchronize` | `timeseries` | `tscollection`

Introduced before R2006a

reset

Reset graphics object properties to their defaults

Syntax

```
reset(h)
```

Description

`reset(h)` resets all properties on the object identified by `h` to their default values. Properties that do not have default values are not affected.

If `h` is a figure, then MATLAB software does not reset `Position`, `Units`, `WindowStyle`, or `PaperUnits`. If `h` is an `Axes` object or a `GeographicAxes` object, then MATLAB does not reset `Position` or `Units`.

Examples

`reset(gca)` resets the properties of the current axes.

`reset(gcf)` resets the properties of the current figure.

See Also

`cla` | `clf` | `gca` | `gcf` | `hold`

Topics

"Default Property Values"

Introduced before R2006a

reset (RandStream)

Reset random number stream

Class

`RandStream`

`parallel.gpu.RandStream`

Syntax

```
reset(s)
reset(s,seed)
```

Description

`reset(s)` resets the generator for the random stream, `s`, to the internal state corresponding to its seed. This is similar to clearing `s` and recreating it using `RandStream(Type,...)`, except that `reset` does not set the stream's `NormalTransform`, `Antithetic`, and `FullPrecision` properties to their original values.

`reset(s,seed)` resets the generator for the random stream, `s`, to the internal state corresponding to `seed` (the seed value), and it updates the `seed` property of `s`. The value of `seed` must be an integer between 0 and $2^{32} - 1$. Resetting a stream's seed can invalidate independence with other streams.

Note Resetting a stream should be used primarily for reproducing results.

Examples

Example 1

Reset a random number stream to its initial state. This does not create a random number stream, it simply resets the stream:

```
stream = RandStream('twister','Seed',0)

stream =

mt19937ar random stream
             Seed: 0
  NormalTransform: Ziggurat

reset(stream);
stream.Seed

ans =

           0
```

Example 2

Reset a random number stream using a specific seed:

```
stream = RandStream('twister','Seed',0)

stream =

mt19937ar random stream
             Seed: 0
  NormalTransform: Ziggurat

reset(stream,1);
stream.Seed

ans =

           1
```

See Also

`RandStream` | `RandStream.getGlobalStream` | `parallel.gpu.RandStream`

rescale

Scale range of array elements

Syntax

```
B = rescale(A)
B = rescale(A,l,u)
B = rescale( ___ ,Name,Value)
```

Description

`B = rescale(A)` scales the entries of an array to the interval [0,1]. The output array `B` is the same size as `A`.

`B = rescale(A,l,u)` scales the entries of an array to the interval [l,u].

`B = rescale( ___ ,Name,Value)` specifies additional parameters for scaling an array for either of the previous syntaxes. For example, `rescale(A,'InputMin',5)` sets all elements in `A` that are less than 5 equal to 5 before scaling to the range [0,1].

Examples

Scale to Unit Interval

Scale the entries of a vector to the interval [0,1].

```
A = 1:5;
B = rescale(A)
```

```
B = 1×5

         0    0.2500    0.5000    0.7500    1.0000
```

Scale to Specified Range

Scale the elements of a vector to the interval [-1,1].

```
A = 1:5;
B = rescale(A,-1,1)
```

```
B = 1×5

   -1.0000   -0.5000         0    0.5000    1.0000
```

Scale Matrix Columns and Rows

Scale each column of a matrix to the interval [0,1] by specifying the minimum and maximum of each column. `rescale` scales along the dimension of the input array that corresponds with the shape of the `'InputMin'` and `'InputMax'` parameter values.

```
A = magic(3)
```

```
A = 3×3

     8     1     6
     3     5     7
     4     9     2
```

```
colmin = min(A)
```

```
colmin = 1×3

     3     1     2
```

```
colmax = max(A)
```

```
colmax = 1×3

     8     9     7
```

```
Bcol = rescale(A,'InputMin',colmin,'InputMax',colmax)
```

```
Bcol = 3×3

    1.0000         0    0.8000
         0    0.5000    1.0000
    0.2000    1.0000         0
```

Scale each row of A to the interval [0,1].

```
rowmin = min(A,[],2)
```

```
rowmin = 3×1

     1
     3
     2
```

```
rowmax = max(A,[],2)
```

```
rowmax = 3×1

     8
     7
     9
```

```
Brow = rescale(A,'InputMin',rowmin,'InputMax',rowmax)
```

```
Brow = 3×3

    1.0000         0    0.7143
         0    0.5000    1.0000
    0.2857    1.0000         0
```

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

- If `A` has type `single`, then the output also has type `single`. Otherwise, the output has type `double`.
- If `A` is a scalar, then `rescale` returns the lower bound of the interval (0 by default) or `NaN` (when the output range contains `Inf`).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`l` — Lower bound
0 (default) | scalar | vector | matrix | multidimensional array

Lower bound, specified as a scalar, vector, matrix, or multidimensional array. `l` must have a size that is compatible with the input array. For example, if `A` is an *M*-by-*N* matrix, then `rescale` operates along the dimension dictated by the shape of `l`:

- If `l` is a scalar, then `rescale` uses it as the lower bound for all elements of `A`.
- If `l` is a 1-by-*N* row vector, then `rescale` uses each element as the lower bound for the corresponding column of `A`.
- If `l` is an *M*-by-1 column vector, then `rescale` uses each element as the lower bound for the corresponding row of `A`.

For more information on compatible array sizes, see “Compatible Array Sizes for Basic Operations”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`u` — Upper bound
1 (default) | scalar | vector | matrix | multidimensional array

Upper bound, specified as a scalar, vector, matrix, or multidimensional array. `u` must have a size that is compatible with the input array. For example, if `A` is an *M*-by-*N* matrix, then `rescale` operates along the dimension dictated by the shape of `u`:

- If `u` is a scalar, then `rescale` uses it as the upper bound for all elements of `A`.
- If `u` is a 1-by-*N* row vector, then `rescale` uses each element as the upper bound for the corresponding column of `A`.
- If `u` is an *M*-by-1 column vector, then `rescale` uses each element as the upper bound for the corresponding row of `A`.

For more information on compatible array sizes, see “Compatible Array Sizes for Basic Operations”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `B = rescale(A,'InputMin',5,'InputMax',10)`

`InputMin` — Minimum of input range
scalar | vector | matrix | multidimensional array

Minimum of input range, specified as a scalar, vector, matrix, or multidimensional array. The default value for an input array `A` is `min(A(:))`. Specifying an input range either expands or shrinks the range of the input data. For instance, `rescale` sets all elements that are less than the specified input minimum to the `'InputMin'` value before scaling.

The `'InputMin'` value must have a size that is compatible with the input array. For example, if `A` is an *M*-by-*N* matrix, then `rescale` operates along the dimension dictated by the shape of the input minimum:

- If the input minimum is a scalar, then `rescale` uses that minimum value for all elements of `A`.
- If the input minimum is a 1-by-*N* row vector, then `rescale` uses each element as the minimum for the corresponding column of `A`.
- If the input minimum is an *M*-by-1 column vector, then `rescale` uses each element as the minimum for the corresponding row of `A`.

For more information on compatible array sizes, see “Compatible Array Sizes for Basic Operations”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`InputMax` — Maximum of input range
scalar | vector | matrix | multidimensional array

Maximum of input range, specified as a scalar, vector, matrix, or multidimensional array. The default value for an input array `A` is `max(A(:))`. Specifying an input range either expands or shrinks the range of the input data. For instance, `rescale` sets all elements that are greater than the specified input maximum to the `'InputMax'` value before scaling.

The `'InputMax'` value must have a size that is compatible with the input array. For example, if `A` is an *M*-by-*N* matrix, then `rescale` operates along the dimension dictated by the shape of the input maximum:

- If the input maximum is a scalar, then `rescale` uses that maximum value for all elements of `A`.
- If the input maximum is a 1-by-*N* row vector, then `rescale` uses each element as the maximum for the corresponding column of `A`.
- If the input maximum is an *M*-by-1 column vector, then `rescale` uses each element as the maximum for the corresponding row of `A`.

For more information on compatible array sizes, see “Compatible Array Sizes for Basic Operations”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Algorithms

`B = rescale(A,l,u,'InputMin',inmin,'InputMax',inmax)` uses the formula

```
l + [(A-inmin)./(inmax-inmin)].*(u-l)
```

to scale the elements of an array `A`.

- If `l` and `u` are not specified, then `rescale` uses the default values 0 and 1, respectively.
- If the `'InputMin'` name-value pair is not specified, then `rescale` sets its value to the default `min(A(:))`.
- If the `'InputMax'` name-value pair is not specified, then `rescale` sets its value to the default `max(A(:))`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- The inputs `l` and `u`, and the value of the name-value pairs `'InputMin'` and `'InputMax'`, cannot have more than one row.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bounds` | `max` | `min`

Introduced in R2017b

reshape

Reshape array

Syntax

```
B = reshape(A,sz)
B = reshape(A,sz1,...,szN)
```

Description

`B = reshape(A,sz)` reshapes `A` using the size vector, `sz`, to define `size(B)`. For example, `reshape(A,[2,3])` reshapes `A` into a 2-by-3 matrix. `sz` must contain at least 2 elements, and `prod(sz)` must be the same as `numel(A)`.

`B = reshape(A,sz1,...,szN)` reshapes `A` into a `sz1`-by-`...`-by-`szN` array where `sz1,...,szN` indicates the size of each dimension. You can specify a single dimension size of `[]` to have the dimension size automatically calculated, such that the number of elements in `B` matches the number of elements in `A`. For example, if `A` is a 10-by-10 matrix, then `reshape(A,2,2,[])` reshapes the 100 elements of `A` into a 2-by-2-by-25 array.

Examples

Reshape Vector into Matrix

Reshape a 1-by-10 vector into a 5-by-2 matrix.

```
A = 1:10;
B = reshape(A,[5,2])
```

```
B = 5×2

     1     6
     2     7
```

```
     3     8
     4     9
     5    10
```

Reshape Matrix to Have Specified Number of Columns

Reshape a 4-by-4 square matrix into a matrix that has 2 columns. Specify `[]` for the first dimension to let `reshape` automatically calculate the appropriate number of rows.

```
A = magic(4)
```

```
A = 4×4

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

```
B = reshape(A,[],2)
```

```
B = 8×2

    16     3
     5    10
     9     6
     4    15
     2    13
    11     8
     7    12
    14     1
```

The result is an 8-by-2 matrix, which maintains the same number of elements as the original matrix. The elements in `B` also maintain their columnwise order from `A`.

Reshape Multidimensional Array into Matrix

Reshape a 3-by-2-by-3 array of zeros into a 9-by-2 matrix.

```
A = zeros(3,2,3);
B = reshape(A,9,2)
```

```
B = 9×2

     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
```

Input Arguments

A — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `cell` | `datetime` | `duration` | `calendarDuration`
Complex Number Support: Yes

sz — Output size

row vector of integers

Output size, specified as a row vector of integers. Each element of `sz` indicates the size of the corresponding dimension in `B`. You must specify `sz` so that the number of elements in `A` and `B` are the same. That is, `prod(sz)` must be the same as `numel(A)`.

Beyond the second dimension, the output, `B`, does not reflect trailing dimensions with a size of `1`. For example, `reshape(A,[3,2,1,1])` produces a 3-by-2 matrix.

Example: `reshape(A,[3,2])`

Example: `reshape(A,[6,4,10])`

Example: `reshape(A,[5,5,5,5])`

`sz1,...,szN` — Size of each dimension
two or more integers | `[]` (optional)

Size of each dimension, specified as two or more integers with at most one `[]` (optional). You must specify at least 2 dimension sizes, and at most one dimension size can be specified as `[]`, which automatically calculates the size of that dimension to ensure that `numel(B)` matches `numel(A)`. When you use `[]` to automatically calculate a dimension size, the dimensions that you *do* explicitly specify must divide evenly into the number of elements in the input matrix, `numel(A)`.

Beyond the second dimension, the output, `B`, does not reflect trailing dimensions with a size of `1`. For example, `reshape(A,3,2,1,1)` produces a 3-by-2 matrix.

Example: `reshape(A,3,2)`

Example: `reshape(A,6,[],10)`

Example: `reshape(A,2,5,3,[])`

Example: `reshape(A,5,5,5,5)`

Output Arguments

`B` — Reshaped array
vector | matrix | multidimensional array | cell array

Reshaped array, returned as a vector, matrix, multidimensional array, or cell array. The data type and number of elements in `B` are the same as the data type and number of elements in `A`. The elements in `B` preserve their columnwise ordering from `A`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `cell` | `datetime` | `duration` | `calendarDuration`

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

- Reshaping the tall dimension (dimension one) is not supported. The first dimension input should always be empty, such as `reshape(X,[],M,N,...)`.

For more information, see “Tall Arrays for Out-of-Memory Data”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If the input is a compile-time empty cell array, then the size arguments must be constants.
- Size arguments must have a fixed size.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).
- For sparse matrices, the `reshape` function does not support trailing ones as inputs after the first two dimensions.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- If the input is a compile-time empty cell array, then the size arguments must be constants.
- Size arguments must have a fixed size.
- For sparse matrices, the `reshape` function does not support trailing ones as inputs after the first two dimensions.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`colon` | `permute` | `repmat` | `shiftdim` | `squeeze`

Introduced before R2006a

residue

Partial fraction expansion (partial fraction decomposition)

Syntax

```
[r,p,k] = residue(b,a)
[b,a] = residue(r,p,k)
```

Description

`[r,p,k] = residue(b,a)` finds the residues, poles, and direct term of a "Partial Fraction Expansion" on page 1-13417 of the ratio of two polynomials, where the expansion is of the form

$$\frac{b(s)}{a(s)} = \frac{b_m s^m + b_{m-1} s^{m-1} + \ldots + b_1 s + b_0}{a_n s^n + a_{n-1} s^{n-1} + \ldots + a_1 s + a_0} = \frac{r_n}{s - p_n} + \ldots + \frac{r_2}{s - p_2} + \frac{r_1}{s - p_1} + k(s).$$

The inputs to `residue` are vectors of coefficients of the polynomials `b = [bm ... b1 b0]` and `a = [an ... a1 a0]`. The outputs are the residues `r = [rn ... r2 r1]`, the poles `p = [pn ... p2 p1]`, and the polynomial `k`. For most textbook problems, `k` is `0` or a constant.

`[b,a] = residue(r,p,k)` converts the partial fraction expansion back to the ratio of two polynomials and returns the coefficients in `b` and `a`.

Examples

Find Partial Fraction Expansion with Real Roots

Find the partial fraction expansion of the following ratio of polynomials $F(s)$ using `residue`

$$F(s) = \frac{b(s)}{a(s)} = \frac{-4s + 8}{s^2 + 6s + 8}.$$

```
b = [-4 8];
a = [1 6 8];
[r,p,k] = residue(b,a)
```

```
r = 2×1

   -12
     8


p = 2×1

    -4
    -2


k =

     []
```

This represents the partial fraction expansion

$$\frac{-4s+8}{s^2+6s+8} = \frac{-12}{s+4} + \frac{8}{s+2}.$$

Convert the partial fraction expansion back to polynomial coefficients using `residue`.

```
[b,a] = residue(r,p,k)
```

```
b = 1×2

    -4     8


a = 1×3

     1     6     8
```

This result represents the original fraction $F(s)$.

Expansion with Complex Roots and Equal Degree of Numerator and Denominator

If the degree of the numerator is equal to the degree of the denominator, the output k can be nonzero.

Find the partial fraction expansion of a ratio of two polynomials $F(s)$ with complex roots and equal degree of numerator and denominator, where $F(s)$ is

$$F(s) = \frac{b(s)}{a(s)} = \frac{2s^3 + s^2}{s^3 + s + 1}.$$

```
b = [2 1 0 0];
a = [1 0 1 1];
[r,p,k] = residue(b,a)
```

```
r = 3×1 complex

   0.5354 + 1.0390i
   0.5354 - 1.0390i
  -0.0708 + 0.0000i


p = 3×1 complex

   0.3412 + 1.1615i
   0.3412 - 1.1615i
  -0.6823 + 0.0000i


k = 2
```

`residue` returns the complex roots and poles, and a constant term in k, representing the partial fraction expansion

$$F(s) = \frac{b(s)}{a(s)} = \frac{2s^3 + s^2}{s^3 + s^2 + 1} = \frac{0.5354 + 1.0390i}{s - (0.3412 + 1.1615i)} + \frac{0.5354 - 1.0390i}{s - (0.3412 - 1.1615i)} + \frac{-0.0708}{s + 0.6823} + 2.$$

Expansion with Numerator Degree Greater Than Denominator Degree

When the degree of the numerator is greater than the degree of the denominator, the output k is a vector that represents the coefficients of a polynomial in s.

Perform the following partial fraction expansion of $F(s)$ using `residue`.

$$F(s) = \frac{b(s)}{a(s)} = \frac{2s^4 + s}{s^2 + 1} = \frac{0.5 - 1i}{s - 1i} + \frac{0.5 + 1i}{s + 1i} + 2s^2 - 2\,.$$

```
b = [2 0 0 1 0];
a = [1 0 1];
[r,p,k] = residue(b,a)
```

```
r = 2×1 complex

   0.5000 - 1.0000i
   0.5000 + 1.0000i


p = 2×1 complex

   0.0000 + 1.0000i
   0.0000 - 1.0000i


k = 1×3

     2     0    -2
```

k represents the polynomial $2s^2 - 2$.

Input Arguments

b — Coefficients of numerator polynomial
vector of numbers

Coefficients of the polynomial in the numerator, specified as a vector of numbers representing the coefficients of the polynomial in descending powers of s.

Data Types: `single` | `double`

Complex Number Support: Yes

a — Coefficients of denominator polynomial
vector of numbers

Coefficients of the polynomial in the denominator, specified as a vector of numbers representing the coefficients of the polynomial in descending powers of s.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

r — Residues of partial fraction expansion
column vector of numbers

Residues of partial fraction expansion, returned as a column vector of numbers.

p — Poles of partial fraction expansion
column vector of numbers

Poles of partial fraction expansion, returned as a column vector of numbers.

k — Direct term
row vector of numbers

Direct term, returned as a row vector of numbers that specify the coefficients of the polynomial in descending powers of s.

More About

Partial Fraction Expansion

Consider the fraction $F(s)$ of two polynomials b and a of degree n and m, respectively

$$F(s) = \frac{b(s)}{a(s)} = \frac{b_n s^n + \ldots + b_2 s^2 + b_1 s + b_0}{a_m s^m + \ldots + a_2 s^2 + a_1 s + a_0}.$$

The fraction $F(s)$ can be represented as a sum of simple fractions

$$\frac{b(s)}{a(s)} = \frac{r_m}{s - p_m} + \frac{r_{m-1}}{s - p_{m-1}} + \ldots + \frac{r_0}{s - p_0} + k(s)$$

This sum is called the partial fraction expansion of F. The values $r_m,\ldots,r_1$ are the residues, the values $p_m,\ldots,p_1$ are the poles, and $k(s)$ is a polynomial in s. For most textbook problems, $k(s)$ is 0 or a constant.

The number of poles `n` is

```
n = length(a)-1 = length(r) = length(p)
```

The direct term vector is empty if `length(b) < length(a)`; otherwise

```
length(k) = length(b)-length(a)+1
```

If `p(j) = ... = p(j+m-1)` is a pole of multiplicity `m`, then the expansion includes terms of the form

$$\frac{r_j}{s - p_j} + \frac{r_{j+1}}{(s - p_j)^2} + \ldots + \frac{r_{j+m-1}}{(s - p_j)^m}.$$

Algorithms

`residue` first obtains the poles using `roots`. Next, if the fraction is nonproper, the direct term `k` is found using `deconv`, which performs polynomial long division. Finally, `residue` determines the residues by evaluating the polynomial with individual roots removed. For repeated roots, `resi2` computes the residues at the repeated root locations.

Numerically, the partial fraction expansion of a ratio of polynomials represents an ill-posed problem. If the denominator polynomial, $a(s)$, is near a polynomial with multiple roots, then small changes in the data, including roundoff errors, can result in arbitrarily large changes in the resulting poles and residues. Problem formulations making use of state-space or zero-pole representations are preferable.

References

[1] Oppenheim, A.V. and R.W. Schafer. *Digital Signal Processing*. Prentice-Hall, 1975, p. 56.

See Also

`deconv` | `poly` | `roots`

Topics

"Create and Evaluate Polynomials"

Introduced before R2006a

restoredefaultpath

Restore search path to factory-installed state

Syntax

```
restoredefaultpath
```

Description

`restoredefaultpath` resets the MATLAB search path to the factory-installed state. By default, the search path includes the MATLAB *`userpath`* folder, the folders defined as part of the `MATLABPATH` environment variable, and the folders provided with MATLAB and other MathWorks products.

Note `restoredefaultpath` is intended only for situations where MATLAB is experiencing startup problems due to a corrupt search path. For general search path cleanup, see "Alternative Functionality" on page 1-13420.

If MATLAB fails to initialize properly on startup, then call both `restoredefaultpath` and `matlabrc`. For more details, see "Path Unsuccessfully Set at Startup".

MATLAB does not support issuing `restoredefaultpath` from a UNC path name. Doing so might result in MATLAB being unable to find files on the search path. If you do use `restoredefaultpath` from a UNC path name, restore the expected behavior by changing the current folder to an absolute path and then reissuing the `restoredefaultpath` command.

Alternative Functionality

For general search path cleanup, use the `rmpath` function or the Set Path dialog box. To open the Set Path dialog box, go to the **Home** tab and in the **Environment** section, click **Set Path**. Alternatively, use the `pathtool` function.

See Also

`addpath` | `matlabrc` | `pathtool` | `rmpath` | `userpath`

Topics

"Path Unsuccessfully Set at Startup"
"What Is the MATLAB Search Path?"

Introduced before R2006a

rethrow

Reissue error

Note As of version 7.5, MATLAB supports error handling that is based on the `MException` class. Calling `rethrow` with a structure argument, as described on this page, is now replaced by calling rethrow with an `MException` object, as described on the reference page for `rethrow`. `rethrow` called with a structure input will be removed in a future version.

Syntax

```
rethrow(errorStruct)
```

Description

`rethrow(errorStruct)` reissues the error specified by `errorStruct`. The currently running function terminates and control returns to the keyboard (or to any enclosing `catch` block). The `errorStruct` argument must be a MATLAB structure containing at least the `message` and `identifier` fields:

Fieldname	Description
`message`	Text of the error message
`identifier`	Message identifier of the error message
`stack`	Information about the error from the program stack

See "Message Identifiers" in the MATLAB documentation for more information on the syntax and usage of message identifiers.

Examples

`rethrow` is usually used in conjunction with `try, catch` statements to reissue an error from a `catch` block after performing `catch`-related operations. For example,

```
try
   do_something
catch
   do_cleanup
   rethrow(previous_error)
end
```

Tips

The `errorStruct` input can contain the field `stack`, identical in format to the output of the `dbstack` command. If the `stack` field is present, the stack of the rethrown error will be set to that value. Otherwise, the stack will be set to the line at which the rethrow occurs.

See Also

`MException` | `assert` | `error` | `rethrow` | `throw` | `throwAsCaller` | `try, catch`

Introduced before R2006a

retime

Resample or aggregate data in timetable, and resolve duplicate or irregular times

Syntax

```
TT2 = retime(TT1,newTimeStep,method)
TT2 = retime(TT1,'regular',method,'TimeStep',dt)
TT2 = retime(TT1,'regular',method,'SampleRate',Fs)
TT2 = retime(TT1,newTimes,method)

TT2 = retime(TT1,newTimeStep)
TT2 = retime(TT1,'regular','TimeStep',dt)
TT2 = retime(TT1,'regular','SampleRate',Fs)
TT2 = retime(TT1,newTimes)

TT2 = retime( ___ ,Name,Value)
```

Description

`TT2 = retime(TT1,newTimeStep,method)` returns a timetable that contains the variables from `TT1` and row times that are regularly spaced by the time step `newTimeStep`. The `retime` function resamples or aggregates data in the variables of `TT1` using the function specified by `method`. You can use `retime` to:

- Interpolate data values from `TT1` at different times.
- Aggregate data into time bins (for example, to create a timetable containing quarterly means from monthly data).
- Remove rows from `TT1` that have duplicate row times.
- Make an irregular timetable into a regular timetable, since `newTimeStep` specifies regular row times.

The `newTimeStep` input argument is a character vector or string that specifies a predefined time step. For example, when `newTimeStep` is `'daily'`, and `method` is `'mean'`, then `TT2` contains the daily means of the data from `TT1`.

The first row time of TT2 is on the time step before the earliest row time from TT1. The row times in TT2 cover the range of row times from TT1. However, TT2 might not include any of the actual row times from TT1, since TT1 might not have any row times that fall on any of the regular row times of TT2.

To interpolate or fill in values in TT2 using different methods for different variables, specify the `VariableContinuity` property of TT1. For more information, see “Retime and Synchronize Timetable Variables Using Different Methods”.

To resample or aggregate data from multiple timetables, see `synchronize`.

`TT2 = retime(TT1,'regular',method,'TimeStep',dt)` calculates regularly spaced row times using the time step `dt`. The `dt` input argument is a scalar duration or calendar duration, specifying a time step of any size. The row times of TT2 span the range of row times of TT1.

Use this syntax when the time step is not one of the predefined time steps you can specify as a character vector or string.

`TT2 = retime(TT1,'regular',method,'SampleRate',Fs)` calculates regularly spaced row times using the sample rate `Fs`. The `Fs` input argument is a positive numeric scalar that specifies the number of samples per second (Hz).

`TT2 = retime(TT1,newTimes,method)` adjusts the timetable variables data to the time vector `newTimes`, using the method specified by `method`. The `newTimes` time vector can be irregular, but it must be a sorted datetime or duration vector and contain unique values. The times in `newTimes` become the row times of TT2.

`TT2 = retime(TT1,newTimeStep)` adjusts timetable data using the `'fillwithmissing'` method. TT2 has missing data indicators wherever TT2 has a row time that does not match any row time in TT1.

If TT1 has rows with duplicate row times and TT2 has row times that match the duplicates, then TT2 contains the first row from each group of rows in TT1 with duplicate row times that match.

`TT2 = retime(TT1,'regular','TimeStep',dt)` calculates regularly spaced row times using the time step `dt` and, where needed, inserts missing data indicators.

`TT2 = retime(TT1,'regular','SampleRate',Fs)` calculates regularly spaced row times using the sample rate `Fs` and, where needed, inserts missing data indicators.

`TT2 = retime(TT1,newTimes)` returns a timetable containing missing data indicators wherever `newTimes` does not match row times in `TT1`.

`TT2 = retime( ___ ,Name,Value)` adjusts timetable data using additional options specified by one or more `Name,Value` pairs. You can use this syntax with the input arguments of any of the previous syntaxes.

Examples

Interpolate Irregular Timetable Data at Hourly Times

Create timetable data that are approximately hourly, but with some irregularity in the times. Interpolate the data so that the output timetable has regular hourly row times.

```
Time = datetime({'2015-12-18 07:02:12';'2015-12-18 08:00:47';...
                 '2015-12-18 09:01:37';'2015-12-18 10:03:10';...
                 '2015-12-18 10:59:34'});
Temp = [37.3;41.9;45.7;42.3;39.8];
Pressure = [30.1;29.9;30.03;29.9;29.8];
TT = timetable(Time,Temp,Pressure)
```

```
TT=5×3 timetable
            Time            Temp    Pressure
    ____________________    ____    ________

    18-Dec-2015 07:02:12    37.3      30.1
    18-Dec-2015 08:00:47    41.9      29.9
    18-Dec-2015 09:01:37    45.7     30.03
    18-Dec-2015 10:03:10    42.3      29.9
    18-Dec-2015 10:59:34    39.8      29.8
```

To resample with cubic spline interpolation, specify `'spline'`.

```
TT2 = retime(TT,'hourly','spline')
```

```
TT2=5×3 timetable
            Time             Temp     Pressure
    ____________________    ______    ________

    18-Dec-2015 07:00:00    37.228     30.124
    18-Dec-2015 08:00:00    41.824     29.899
```

```
    18-Dec-2015 09:00:00    45.694     30.029
    18-Dec-2015 10:00:00    42.552      29.91
    18-Dec-2015 11:00:00    39.808       29.8
```

Aggregate Timetable Data and Calculate Mean Values

Create a timetable with temperature and pulse readings taken every fifteen minutes.

```
Time = [minutes(0):minutes(15):minutes(105)]';
Temp = [98;97.5;97.9;98.1;97.9;98;98.3;97.8];
Pulse = [80;75;73;68;69;65;72;71];
TT = timetable(Time,Temp,Pulse)
```

```
TT=8×3 timetable
     Time      Temp    Pulse
    _______    ____    _____

    0 min        98     80
    15 min     97.5     75
    30 min     97.9     73
    45 min     98.1     68
    60 min     97.9     69
    75 min       98     65
    90 min     98.3     72
    105 min    97.8     71
```

Calculate the mean for each reading over hourly time bins. When you aggregate data over time bins, the row times of the output timetable are the left edges of the time bins.

```
TT2 = retime(TT,'hourly','mean')
```

```
TT2=2×3 timetable
     Time      Temp     Pulse
    ______    ______    _____

    0 min     97.875       74
    60 min        98    69.25
```

Specify Time Step of Your Own

Create a timetable that contains times, temperature, and pressure readings taken approximately at the half-hour mark, but with one measurement from 9:00 AM missing.

```
Time = datetime({'2015-12-18 07:29:53';'2015-12-18 08:00:00';...
                 '2015-12-18 08:31:02';'2015-12-18 09:30:00'});
Temp = [37.3;41.9;45.7;39.8];
Pressure = [30.1;29.9;30.03;29.8];
TT1 = timetable(Time,Temp,Pressure)
```

```
TT1=4×3 timetable
            Time            Temp    Pressure
    ____________________    ____    ________

    18-Dec-2015 07:29:53    37.3      30.1
    18-Dec-2015 08:00:00    41.9      29.9
    18-Dec-2015 08:31:02    45.7     30.03
    18-Dec-2015 09:30:00    39.8      29.8
```

Specify a 30 minute time step. Since 30 minutes is not a predefined time step, you must specify it as a duration value, using the `'TimeStep'` name-value pair argument. Resample the data from `TT1` using linear interpolation.

```
dt = minutes(30);
TT2 = retime(TT1,'regular','linear','TimeStep',dt)
```

```
TT2=6×3 timetable
            Time             Temp     Pressure
    ____________________    ______    ________

    18-Dec-2015 07:00:00    32.736     30.298
    18-Dec-2015 07:30:00    37.318     30.099
    18-Dec-2015 08:00:00      41.9       29.9
    18-Dec-2015 08:30:00    45.573     30.026
    18-Dec-2015 09:00:00    42.802     29.917
    18-Dec-2015 09:30:00      39.8       29.8
```

Specify Sample Rate

Create a timetable using column vectors of data. The row times are between 10 and 50 milliseconds.

```
Intensity = [100 98.7 95.2 101.4 99.1]';
Time = milliseconds([11 20 34 40.3 49.9])';
TT1 = timetable(Time,Intensity)
```

```
TT1=5×2 timetable
       Time       Intensity
    __________    _________

    0.011 sec         100
    0.02 sec         98.7
    0.034 sec        95.2
    0.0403 sec      101.4
    0.0499 sec       99.1
```

Resample the data in TT1 using a sample rate of 100 Hz.

```
TT2 = retime(TT1,'regular','linear','SampleRate',100)
```

```
TT2=5×2 timetable
      Time      Intensity
    ________    _________

    0.01 sec     100.14
    0.02 sec       98.7
    0.03 sec       96.2
    0.04 sec      101.1
    0.05 sec     99.076
```

Interpolate Timetable Data to Time Vector

Create a timetable that contains times, temperature, and pressure readings.

```
Time = datetime({'2015-12-18 07:29:53';'2015-12-18 08:00:00';...
                 '2015-12-18 08:31:02';'2015-12-18 09:30:00'});
Temp = [37.3;41.9;45.7;39.8];
```

```
Pressure = [30.1;29.9;30.03;29.8];
TT1 = timetable(Time,Temp,Pressure)
```

```
TT1=4×3 timetable
            Time             Temp    Pressure
    ____________________    ____    ________

    18-Dec-2015 07:29:53    37.3      30.1
    18-Dec-2015 08:00:00    41.9      29.9
    18-Dec-2015 08:31:02    45.7     30.03
    18-Dec-2015 09:30:00    39.8      29.8
```

Create a time vector and interpolate the timetable data at the times in the vector. This time vector starts at 8:00 AM. If you instead use the `'TimeStep'` name-value pair argument, then the output timetable would start at 7:00 AM. One reason to use a time vector is to ensure that the output timetable starts, and ends, with times you specify.

```
newTimes = [datetime('2015-12-18 08:00:00'):minutes(30):datetime('2015-12-18 09:30:00')
TT2 = retime(TT1,newTimes,'linear')
```

```
TT2=4×3 timetable
            Time             Temp     Pressure
    ____________________    ______    ________

    18-Dec-2015 08:00:00      41.9       29.9
    18-Dec-2015 08:30:00    45.573     30.026
    18-Dec-2015 09:00:00    42.802     29.917
    18-Dec-2015 09:30:00      39.8       29.8
```

Adjust Timetable by Inserting Missing Data Indicators

Create a timetable that contains times and measurements of wind speed and direction.

```
Time = datetime({'2015-12-18 07:00:00';'2015-12-18 08:03:47';...
                 '2015-12-18 09:00:00';'2015-12-18 10:00:00';...
                 '2015-12-18 10:59:34'});
WindSpeed = [13.4;6.5;7.3;8.5;2.3];
WindDirection = categorical({'NE';'N';'NE';'NW';'W'});
TT1 = timetable(Time,WindSpeed,WindDirection)
```

```
TT1=5×3 timetable
            Time            WindSpeed    WindDirection
    ____________________    _________    _____________

    18-Dec-2015 07:00:00      13.4            NE
    18-Dec-2015 08:03:47       6.5            N
    18-Dec-2015 09:00:00       7.3            NE
    18-Dec-2015 10:00:00       8.5            NW
    18-Dec-2015 10:59:34       2.3            W
```

Adjust the data to an hourly time vector. Insert missing data indicators in TT2 where TT1 does not have data on the hourly time vector.

```
TT2 = retime(TT1,'hourly')
```

```
TT2=5×3 timetable
            Time            WindSpeed    WindDirection
    ____________________    _________    _____________

    18-Dec-2015 07:00:00      13.4        NE
    18-Dec-2015 08:00:00       NaN        <undefined>
    18-Dec-2015 09:00:00       7.3        NE
    18-Dec-2015 10:00:00       8.5        NW
    18-Dec-2015 11:00:00       NaN        <undefined>
```

Apply Multiple Methods to Timetable

Load a timetable. Adjust the timetable variables using the `retime` function and different methods for different variables.

Load a timetable with temperature, wind speed, and rainfall measurements for Boston.

```
load bostonTT
Boston
```

```
Boston=6×4 timetable
           Time            Temp    WindSpeed    Rain
    ____________________    ____    _________    ____

    2016-06-09 06:03:00    59.5       0.1       0.05
```

```
    2016-06-09 12:00:23      63       2.3       0.08
    2016-06-09 18:02:57    61.7       3.1       0.13
    2016-06-10 06:01:47    55.4       5.7       0.15
    2016-06-10 12:06:00    62.3       2.6       0.87
    2016-06-10 18:02:57    58.8       6.2       0.33
```

Adjust the data to produce daily mean temperatures and wind speeds, and daily sums of the rainfall. `retime` applies the same method to all timetable variables. To apply different methods, index into the timetable to select variables, and call `retime` for each method you use.

```
BOS = Boston(:,{'Temp','WindSpeed'});
TT1 = retime(BOS,'daily','mean')
```

```
TT1=2×3 timetable
           Time             Temp     WindSpeed
    ___________________    ______    _________

    2016-06-09 00:00:00      61.4     1.8333
    2016-06-10 00:00:00    58.833     4.8333
```

```
BOS = Boston(:,'Rain');
TT2 = retime(BOS,'daily','sum')
```

```
TT2=2×2 timetable
           Time            Rain
    ___________________    ____

    2016-06-09 00:00:00    0.26
    2016-06-10 00:00:00    1.35
```

To combine all results in one timetable, concatenate `TT1` and `TT2`.

```
TT = [TT1 TT2]
```

```
TT=2×4 timetable
           Time             Temp     WindSpeed    Rain
    ___________________    ______    _________    ____

    2016-06-09 00:00:00      61.4     1.8333      0.26
    2016-06-10 00:00:00    58.833     4.8333      1.35
```

Include Right Edges of Time Bins

Bin timetable data into hourly bins. Specify the right edges, or end times, of the time bins as row times. By default, the left edges, or start times of the time bins, are row times.

Create a timetable with temperature and pulse readings taken at fifteen minutes intervals.

```
Time = [minutes(15):minutes(15):minutes(105)]';
Temp = [97.5;97.9;98.1;97.9;98;98.3;97.8];
Pulse = [75;73;68;69;65;72;71];
TT = timetable(Time,Temp,Pulse)
```

```
TT=7×3 timetable
     Time      Temp    Pulse
    _______    ____    _____

    15 min     97.5     75
    30 min     97.9     73
    45 min     98.1     68
    60 min     97.9     69
    75 min       98     65
    90 min     98.3     72
    105 min    97.8     71
```

Calculate the mean for each reading over hourly time bins. Specify that the row times of the output timetable are the right edges of the time bins. Since the right edges are included, the reading at 60 minutes is included in the first time bin.

```
TT2 = retime(TT,'hourly','mean','IncludedEdge','right')
```

```
TT2=2×3 timetable
     Time       Temp     Pulse
    _______    ______    ______

    60 min      97.85     71.25
    120 min    98.033    69.333
```

Calculate the means, with the left edges as the row times. The mean values are different from those in TT2, because the reading at 60 minutes is now in the second time bin.

```
TT3 = retime(TT,'hourly','mean')
```

```
TT3=2×3 timetable
     Time       Temp      Pulse
    ______     ______     _____

    0 min      97.833        72
    60 min         98     69.25
```

Input Arguments

TT1 — Input timetable
timetable

Input timetable.

newTimeStep — Time step for spacing times in output timetable
character vector

Time step for spacing times in the output timetable, specified as a character vector. `newTimeStep` can be any of the predefined time steps in the table.

Time Step	Description
`'yearly'`	One year
`'quarterly'`	One quarter
`'monthly'`	One month
`'weekly'`	One week
`'daily'`	One day
`'hourly'`	One hour
`'minutely'`	One minute
`'secondly'`	One second

dt — Time step of any size
datetime scalar | duration scalar

Time step of any size, specified as a datetime scalar or duration scalar.

Data Types: `datetime` | `duration` | `calendarDuration`

`Fs` — Sample rate
positive numeric scalar

Sample rate, specified as a positive numeric scalar. `Fs` specifies the number of samples per second (Hz).

`newTimes` — New time vector
datetime vector | duration vector

New time vector, specified as a datetime vector or a duration vector. The new time vector must be a column vector. `newTimes` can have a different number of rows than `TT1`.

`method` — Method for adjusting timetable data
character vector | string scalar | function handle

Method for adjusting timetable data, specified as a character vector, string scalar, or function handle. You can use any of the listed methods listed to adjust the data from `TT1`.

Fill Methods

Copy data from the rows of `TT1` when row times of `TT2` match row times of `TT1`. Then, fill the remaining rows of `TT2` with missing data indicators.

To fill the remaining rows with a constant instead of a missing data indicator, specify `method` as `'fillwithconstant'` and use the `'Constant'` name-value pair argument.

Method	Description
`'fillwithmissing'`	Fill gaps with missing data indicators (for example, `NaN` for numeric variables).
`'fillwithconstant'`	Fill gaps with the value of the `'Constant'` name-value pair argument. The default value is `0`.

Nearest Neighbor Methods

Copy data from the rows of `TT1` to the rows of `TT2` whose row times are the nearest match, according to the specified method. `TT1` must be sorted by its row times.

Method	Description
`'previous'`	Copy data from the nearest preceding neighbor in the input timetable, proceeding from the end of the vector of row times. If there are duplicate row times, then `'previous'` indicates the last of the duplicates.
`'next'`	Copy data from the nearest following neighbor in the input timetable, proceeding from the beginning of the vector of row times. If there are duplicate row times, then `'next'` indicates the first of the duplicates.
`'nearest'`	Copy data from the nearest neighbor in the input timetable.

Interpolation Methods

Interpolate data values in `TT2` from data values in neighboring rows of `TT1`. The input timetable must have row times that are sorted and unique. To control how the data are extrapolated beyond the first and last row times of `TT1`, use the `'EndValues'` name-value pair argument.

Method	Description
`'linear'`	Use linear interpolation.
`'spline'`	Use piecewise cubic spline interpolation.
`'pchip'`	Use shape-preserving piecewise cubic interpolation.
`'makima'`	Use modified Akima cubic Hermite interpolation.

Aggregation Methods

Aggregate data from the rows of `TT1` over time bins specified by the row times of `TT2`. Each row time of `TT2` is the left edge of a time bin, with the next consecutive row time being the right edge. By default, the left edges are included in the time bins. To control whether the left or the right bin edges are included in the time bins, use the `'IncludedEdge'` name-value pair argument.

If you specify the time vector `newTimes`, then `newTimes` must be sorted in ascending order.

All the listed methods omit `NaN`s, `NaT`s, and other missing data indicators, except for `func`. To include missing data indicators, specify `func` as a function handle to a function that includes them when aggregating data.

Method	Description
`'sum'`	Sum the values in each time bin.
`'mean'`	Calculate the mean of the values in each time bin.
`'prod'`	Calculate the product of the values in each time bin.
`'min'`	Calculate the minimum of the values in each time bin.
`'max'`	Calculate the maximum of the values in each time bin.
`'count'`	Count the number of values in each time bin.
`'firstvalue'`	Use the first value in each time bin.
`'lastvalue'`	Use the last value in each time bin.
`@func`	Use the function specified by the function handle (for example, `@std` to calculate the standard deviation for the values in each time bin). `func` must return an output argument that is a scalar or a row vector, and must accept empty inputs.

Default Method

The default method is equivalent to leaving `method` unspecified.

Method	Description
`'default'` (default)	Either fill gaps with missing data indicators, or use per-variable methods if they are specified by the `VariableContinuity` property of the input timetable.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `TT2 = retime(TT1,newTimes,'Constant',-1)` creates the timetable `TT2` and assigns the value `-1` to elements in rows of `TT2` with row times that do not match row times from `TT1`.

`Constant` — Value for filling gaps when method is `'fillwithconstant'`
array

Value for filling gaps when the method is `'fillwithconstant'`, specified as the comma-separated pair consisting of `'Constant'` and an array. The default value is `0`. The data type of the value specified by `'Constant'` must be compatible with the data types of the timetable variables.

Example: `TT2 = retime(TT1,'hourly','fillwithconstant','Constant','NONE')` fills gaps in `TT2` with the character vector `'NONE'` when all the variables in `TT2` contain text.

`EndValues` — Method for extrapolation when using interpolation method
`'extrap'` (default) | array

Method for extrapolation when using an interpolation method, specified as the comma-separated pair consisting of `'EndValues'` and either `'extrap'`, or an array. If you specify an array, then its data type must be compatible with all the timetable variables.

Method	Description
`'extrap'` (default)	Extrapolate using the method specified by the `method` input argument

Method	Description
array	Extrapolate by filling gaps outside the range of input row times with an array

Example: `TT2 = retime(TT1,'daily','previous','EndValues',1000)` fills gaps in `TT2` with previous row values where `TT2` has row times within the range of row times from `TT1`, and with the value `1000` where `TT2` has row times outside that range.

`IncludedEdge` — Edges to include in each time bin
`'left'` (default) | `'right'`

Edges to include in each time bin, specified as the comma-separated pair consisting of `'IncludedEdge'` and either `'left'` or `'right'`. Each row time of `TT2` is the left edge of a time bin, with the next consecutive row time being the right edge.

Edges to Include	Description
`'left'` (default)	All bins include the left bin edge, except for the last bin, which includes both edges
`'right'`	All bins include the right bin edge, except for the first bin, which includes both edges

If you specify `'left'`, then the time bins include the left edges except for the last bin, which includes both edges. If you specify `'right'`, then the time bins include the right edges except for the first bin, which includes both edges.

Example: `TT2 = retime(TT1,'hourly','mean','IncludedEdge','right')` includes the right bin edge of each time bin.

Compatibility Considerations

`'SamplingRate'` is not recommended

Not recommended starting in R2018b

The `'SamplingRate'` name-value pair argument is not recommended. Use `'SampleRate'` instead. The corresponding timetable property is also named `SampleRate`.

For backward compatibility, you still can specify `'SamplingRate'` as the name of the name-value pair. However, the value is assigned to the `SampleRate` property.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- Interpolation method `'spline'` is not supported.
- The `'EndValues'` name-value pair is not supported.
- Retiming to `'regular'` spaced row times is not supported.
- All variables in input timetables must support missing values (floating point, categorical, datetime, duration, or text) for the methods `'nearest'`, `'next'`, and `'previous'`.

For more information, see “Tall Arrays”.

See Also

`horzcat` | `innerjoin` | `lag` | `outerjoin` | `resample` | `synchronize` | `vertcat`

Topics

“Create Timetables”
“Clean Timetable with Missing, Duplicate, or Nonuniform Times”
“Resample and Aggregate Data in Timetable”
“Combine Timetables and Synchronize Their Data”
“Retime and Synchronize Timetable Variables Using Different Methods”

Introduced in R2016b

return

Return control to invoking script or function

Syntax

```
return
```

Description

`return` forces MATLAB to return control to the invoking program before it reaches the end of the script or function. The invoking program is a script or function that calls the script or function containing the call to `return`. If you call the script or function that contains `return` directly, there is no invoking program and MATLAB returns control to the command prompt.

Note Be careful when you use `return` within conditional blocks, such as `if` or `switch`, or within loop control statements, such as `for` or `while`. When MATLAB reaches a `return` statement, it does not just exit the loop; it exits the script or function and returns control to the invoking program or command prompt.

Examples

Return Control to Keyboard

In your current working folder, create a function, `findSqrRootIndex`, to find the index of the first occurrence of the square root of a value within an array. If the square root is not found, the function returns `NaN`.

```
function idx = findSqrRootIndex(target,arrayToSearch)

idx = NaN;
if target < 0
```

```
    return
end

for idx = 1:length(arrayToSearch)
    if arrayToSearch(idx) == sqrt(target)
        return
    end
end
```

At the command prompt, call the function.

```
A = [3 7 28 14 42 9 0];
b = 81;
findSqrRootIndex(b,A)
```

```
ans =

     6
```

When MATLAB encounters the return statement, it returns control to the keyboard because there is no invoking script or function.

Return Control to Invoking Function

In a file, `returnControlExample.m`, in your current working folder, create the following function to find the index of the first occurrence of the square root of a value within an array. This function calls the `findSqrRootIndex` function you created in the previous example.

```
function returnControlExample(target)
    arrayToSearch = [3 7 28 14 42 9 0];
    idx = findSqrRootIndex(target,arrayToSearch);

    if isnan(idx)
        disp('Square root not found.')
    else
        disp(['Square root found at index ' num2str(idx)])
    end
```

```
end
```

At the command prompt, call the function.

```
returnControlExample(49)
```

```
Square root found at index 2
```

When MATLAB encounters the `return` statement within `findSqrRootIndex`, it returns control to the invoking function, `returnControlExample`, and displays the relevant message.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`break` | `continue` | `disp` | `end` | `error` | `for` | `if` | `keyboard` | `switch` | `while`

Introduced before R2006a

reverse

Reverse order of characters in strings

Syntax

```
newStr = reverse(str)
```

Description

`newStr = reverse(str)` reverses the order of the characters in `str`.

Examples

Reverse Strings

Reverse the strings in a string array and find strings that read the same when reversed. Starting in R2017a, you can create strings using double quotes.

```
str = ["airport","control tower","radar","runway"]
```

```
str = 1x4 string array
    "airport"    "control tower"    "radar"    "runway"
```

```
newStr = reverse(str)
```

```
newStr = 1x4 string array
    "tropria"    "rewot lortnoc"    "radar"    "yawnur"
```

```
tf = (newStr == str)
```

```
tf = 1x4 logical array

   0   0   1   0
```

```
str(tf)
```

```
ans =
"radar"
```

Input Arguments

`str` — Input text
string array | character vector | cell array of character vectors

Input text, specified as a string array, character vector, or cell array of character vectors.

Output Arguments

`newStr` — Output text
string array | character vector | cell array of character vectors

Output text, returned as a string array, a character vector, or a cell array of character vectors. `str` and `newStr` are the same data type.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `str` must be a string scalar, a character vector, or a cell array containing not more than one character vector.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`eq` | `flip` | `fliplr` | `lower` | `replace` | `reshape` | `upper`

Introduced in R2016b

rewriteDirectory

Write modified metadata to existing IFD

Syntax

```
rewriteDirectory(t)
```

Description

`rewriteDirectory(t)` writes modified metadata (tag) data to an existing directory. Use this function when you want to change the value of a tag in an existing image file directory.

Examples

Modify Value of Tag

Create image data, write it to a TIFF file, and modify the value of a tag.

Write a sample TIFF file, `mytif.tif` and create a `Tiff` object associated with this file.

```
imdata = peaks(256);
imwrite(imdata,'mytif.tif');
t = Tiff('mytif.tif','r+');
```

Modify the value of a tag.

```
setTag(t,'Software','MATLAB');
rewriteDirectory(t);
close(t);
```

Input Arguments

t — Tiff object
`Tiff` object

`Tiff` object representing a TIFF file. Use the `Tiff` function to create the object.

Algorithms

References

This function corresponds to the `TIFFRewriteDirectory` function in the LibTIFF C API. To use this function, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`writeDirectory`

Topics

"Importing Images"
"Exporting to Images"

Introduced in R2009b

rgb2gray

Convert RGB image or colormap to grayscale

Syntax

```
I = rgb2gray(RGB)
newmap = rgb2gray(map)
```

Description

`I = rgb2gray(RGB)` converts the truecolor image `RGB` to the grayscale image `I`. The `rgb2gray` function converts RGB images to grayscale by eliminating the hue and saturation information while retaining the luminance. If you have Parallel Computing Toolbox installed, `rgb2gray` can perform this conversion on a GPU.

`newmap = rgb2gray(map)` returns a grayscale colormap equivalent to `map`.

Examples

Convert RGB Image to Grayscale Image

Read and display an RGB image, and then convert it to grayscale.

Read the sample file, `peppers.png`, and display the RGB image.

```
RGB = imread('peppers.png');
imshow(RGB)
```

Convert the RGB image to a grayscale image and display it.

```
I = rgb2gray(RGB);
figure
imshow(I)
```

Convert RGB Colormap to Grayscale Colormap

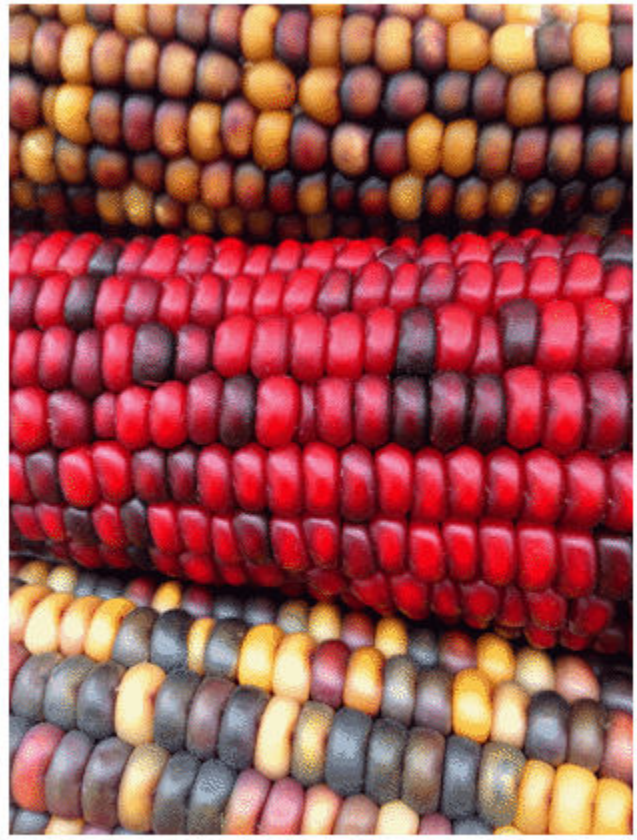
Read an indexed image with an RGB colormap. Then, convert the colormap to grayscale.

Read the sample file, `corn.tif`, which is an indexed image with an RGB colormap.

```
[X,map] = imread('corn.tif');
```

Display the image.

```
imshow(X,map)
```



Convert the RGB colormap to a grayscale colormap and redisplay the image.

```
newmap = rgb2gray(map);
imshow(X,newmap)
```

Input Arguments

RGB — Truecolor image
m-by-*n*-by-3 numeric array

Truecolor image, specified as an *m*-by-*n*-by-3 numeric array.

If you have Parallel Computing Toolbox installed, `RGB` can also be a gpuArray.

Data Types: `single` | `double` | `uint8` | `uint16`

`map` — Colormap
c-by-3 numeric matrix

Colormap, specified as a *c*-by-3 numeric matrix with values in the range [0, 1]. Each row of `map` is a three-element RGB triplet that specifies the red, green, and blue components of a single color of the colormap.

If you have Parallel Computing Toolbox installed, `map` can also be a gpuArray.

Data Types: `double`

Output Arguments

`I` — Grayscale image
m-by-*n* numeric array

Grayscale image, returned as an *m*-by-*n* numeric array.

If you have Parallel Computing Toolbox installed, then `I` can also be a gpuArray.

`newmap` — Grayscale colormap
c-by-3 numeric matrix

Grayscale colormap, returned as an *c*-by-3 numeric matrix with values in the range [0, 1]. The three columns of `newmap` are identical, so that each row of `map` specifies a single intensity value.

If you have Parallel Computing Toolbox installed, then `newmap` can also be a gpuArray.

Data Types: `double`

Tips

- `rgb2gray` supports the generation of C code using MATLAB Coder.

Algorithms

`rgb2gray` converts RGB values to grayscale values by forming a weighted sum of the *R*, *G*, and *B* components:

```
0.2989 * R + 0.5870 * G + 0.1140 * B
```

These are the same weights used by the `rgb2ntsc` function to compute the *Y* component.

The coefficients used to calculate grayscale values in `rgb2gray` are identical to those used to calculate luminance (E'y) in Rec.ITU-R BT.601-7 after rounding to 3 decimal places.

Rec.ITU-R BT.601-7 calculates E'y using the following formula:

```
0.299 * R + 0.587 * G + 0.114 * B
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

When generating code, if you choose the generic `MATLAB Host Computer` target platform, `rgb2gray` generates code that uses a precompiled, platform-specific shared library. Use of a shared library preserves performance optimizations but limits the target platforms for which code can be generated.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`ind2gray` | `mat2gray` | `ntsc2rgb` | `rgb2ind` | `rgb2ntsc`

Topics

“Image Types”

rgb2hsv

Convert RGB colors to HSV

Syntax

```
HSV = rgb2hsv(RGB)
hsvmap = rgb2hsv(rgbmap)
```

Description

`HSV = rgb2hsv(RGB)` converts the red, green, and blue values of an RGB image to hue, saturation, and value (HSV) values of an HSV image.

`hsvmap = rgb2hsv(rgbmap)` converts an RGB colormap to an HSV colormap.

Examples

Convert Colormap to HSV

Get a down-sampled version of the `prism` colormap.

```
rgb = prism(6)
```

```
rgb = 6×3

    1.0000         0         0
    1.0000    0.5000         0
    1.0000    1.0000         0
         0    1.0000         0
         0         0    1.0000
    0.6667         0    1.0000
```

Convert the RGB values in the colormap to HSV.

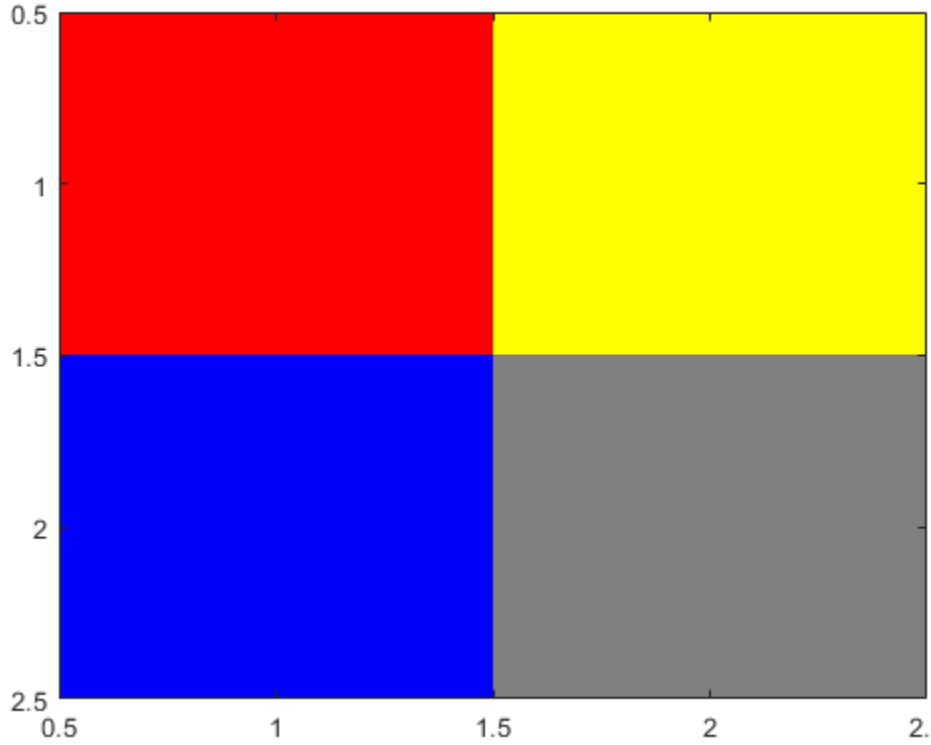
```
hsv = rgb2hsv(rgb)
```

```
hsv = 6×3

         0    1.0000    1.0000
    0.0833    1.0000    1.0000
    0.1667    1.0000    1.0000
    0.3333    1.0000    1.0000
    0.6667    1.0000    1.0000
    0.7778    1.0000    1.0000
```

Convert Truecolor Image to HSV

Create a 2-by-2 truecolor image.

```
rgb(:,:,1) = [1 1; 0 .5];
rgb(:,:,2) = [0 1; 0 .5];
rgb(:,:,3) = [0 0; 1 .5];
image(rgb);
```



Convert the image to an HSV array.

```
hsv = rgb2hsv(rgb)
```

```
hsv =
hsv(:,:,1) =

         0    0.1667
    0.6667         0


hsv(:,:,2) =

     1     1
```

```
     1     0


hsv(:,:,3) =

    1.0000    1.0000
    1.0000    0.5000
```

Input Arguments

RGB — RGB image
m-by-*n*-by-3 numeric array

RGB image to convert, specified as an *m*-by-*n*-by-3 numeric array. The third dimension of RGB defines the red, green, and blue intensity of each pixel, respectively.

Data Types: `single` | `double` | `uint8` | `uint16`

`rgbmap` — RGB colormap
c-by-3 numeric matrix

RGB colormap, specified as a *c*-by-3 numeric matrix with values in the range [0, 1]. Each row of `rgbmap` is a three-element RGB triplet that specifies the red, green, and blue components of a single color of the colormap.

Data Types: `double`

Output Arguments

HSV — HSV image
m-by-*n*-by-3 numeric array

HSV image, returned as an *m*-by-*n*-by-3 numeric array with values in the range [0, 1]. The third dimension of HSV defines the hue, saturation, and value for each pixel, respectively, as described in the table.

Attribute	Description
Hue	Value from `0` to `1` that corresponds to the color's position on a color wheel. As hue increases from `0` to `1`, the color transitions from red to orange, yellow, green, cyan, blue, magenta, and finally back to red.
Saturation	Amount of hue or departure from neutral. `0` indicates a neutral shade, whereas `1` indicates maximum saturation.
Value	Maximum value among the red, green, and blue components of a specific color.

The data type of `HSV` depends on the type of `RGB`:

- If `RGB` is of type `single`, then `HSV` is of type `single`.
- Otherwise, `HSV` is of type `double`.

Data Types: `double` | `single`

`hsvmap` — HSV colormap
c-by-3 numeric matrix

HSV colormap, returned as a *c*-by-3 numeric matrix with values in the range [0, 1]. Each row of `hsvmap` is a three-element HSV triplet that specifies the hue, saturation, and value components of a single color of the colormap.

The data type depends on the type of `rgbmap`:

- If `rgbmap` is of type `single`, then `hsvmap` is of type `single`.
- Otherwise, `hsvmap` is of type `double`.

Data Types: `single` | `double`

References

[1] Smith, A. R. "Color Gamut Transform Pairs". *SIGGRAPH 78 Conference Proceedings*. 1978, pp. 12–19.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`hsv` | `hsv2rgb`

Introduced before R2006a

rgb2ind

Convert RGB image to indexed image

Syntax

```
[X,cmap] = rgb2ind(RGB,Q)
[X,cmap] = rgb2ind(RGB,tol)
X = rgb2ind(RGB,inmap)
___ = rgb2ind(___,dithering)
```

Description

`[X,cmap] = rgb2ind(RGB,Q)` converts the RGB image to an indexed image `X` with associated colormap `cmap` using minimum variance quantization with `Q` quantized colors and dithering.

`[X,cmap] = rgb2ind(RGB,tol)` converts the RGB image to an indexed image using uniform quantization with tolerance `tol` and dithering.

`X = rgb2ind(RGB,inmap)` converts the RGB image to an indexed image using the inverse colormap algorithm with specified colormap `inmap` and dithering.

`___ = rgb2ind(___,dithering)` enables or disables dithering.

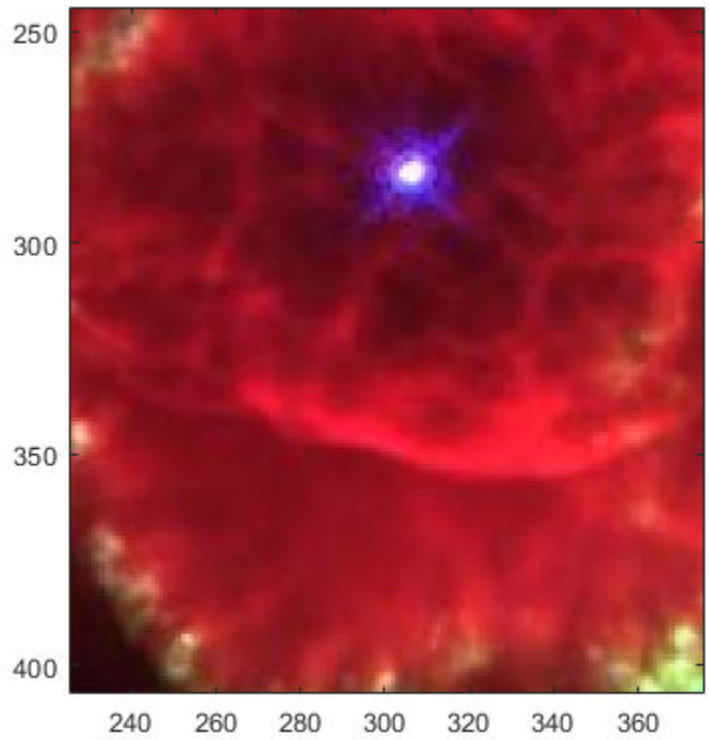
Examples

Convert RGB Image to Indexed Image

Read and display a truecolor `uint8` JPEG image of a nebula.

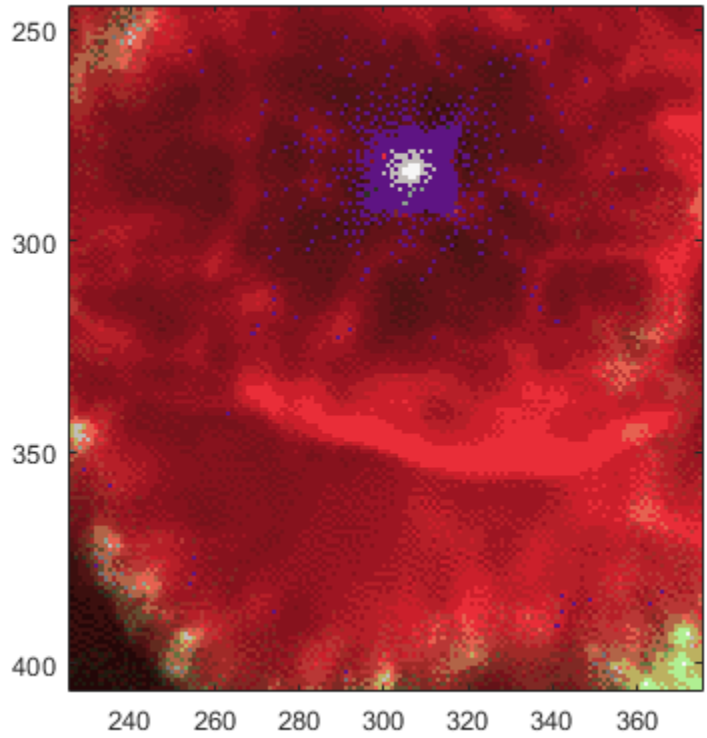
```
RGB = imread('ngc6543a.jpg');
figure
imagesc(RGB)
```

```
axis image
zoom(4)
```



Convert RGB to an indexed image with 32 colors.

```
[IND,map] = rgb2ind(RGB,32);
figure
imagesc(IND)
colormap(map)
axis image
zoom(4)
```



Input Arguments

RGB — RGB image
m-by-*n*-by-3 array

RGB image, specified as an *m*-by-*n*-by-3 array.

Data Types: `single` | `double` | `uint8` | `uint16`

Q — Number of quantized colors
positive integer

Number of quantized colors used for minimum variance quantization, specified as a positive integer that is less than or equal to 65,536. The returned colormap `cmap` has `Q` or fewer colors.

`tol` — Tolerance
number in the range [0, 1]

Tolerance used for uniform quantization, specified as a number in the range [0, 1]. The returned colormap `cmap` has `(floor(1/tol)+1)^3` or fewer colors.

`inmap` — Input colormap
c-by-3 matrix

Input colormap, specified as a *c*-by-3 matrix with values in the range [0, 1]. Each row of `inmap` is a three-element RGB triplet that specifies the red, green, and blue components of a single color of the colormap. The colormap has a maximum of 65,536 colors.

Data Types: `double`

`dithering` — Perform dithering
`'dither'` (default) | `'nodither'`

Perform dithering, specified as `'dither'` or `'nodither'`. Dithering increases the color resolution at the expense of spatial resolution. For more information, see `dither`.

If you select `'nodither'`, then `rgb2ind` does not perform dithering. Instead, the function maps each color in the original image to the closest color in the new colormap.

Output Arguments

`X` — Indexed image
m-by-*n* matrix of nonnegative integers

Indexed image, returned as an *m*-by-*n* matrix of nonnegative integers. If the length of `map` is less than or equal to 256, then the output image is of class `uint8`. Otherwise, the output image is of class `uint16`. The value `0` in the output array `X` corresponds to the first color in the colormap.

Note The values in image `X` are indexes into the colormap `map` and should not be used in mathematical processing, such as filtering operations.

Data Types: `uint8` | `uint16`

`cmap` — Colormap
c-by-3 matrix

Colormap, returned as a *c*-by-3 matrix with values in the range [0, 1]. Each row of `cmap` is a three-element RGB triplet that specifies the red, green, and blue components of a single color of the colormap. The colormap has a maximum of 65,536 colors.

Data Types: `double`

Algorithms

- Uniform Quantization — If you specify `tol`, then `rgb2ind` uses uniform quantization to convert the image. Uniform quantization cuts the RGB color cube into smaller cubes of length `tol`. For example, if you specify a `tol` of 0.1, then the edges of the cubes are one-tenth the length of the RGB cube. The total number of small cubes is:

  ```
  t = (floor(1/tol)+1)^3
  ```

  Each cube represents a single color in the output image. Therefore, `t` is the maximum length of the colormap . `rgb2ind` removes any colors that don't appear in the input image, so the actual colormap can be smaller than `t`.
- Minimum Variance Quantization — If you specify `Q`, then `rgb2ind` uses minimum variance quantization. Minimum variance quantization cuts the RGB color cube into smaller boxes (not necessarily cubes) of different sizes, depending on how the colors are distributed in the image. If the input image actually uses fewer colors than the number specified, then the output colormap is also smaller.
- Inverse Colormap — If you specify an input colormap `inmap`, then `rgb2ind` uses colormap mapping. The inverse colormap algorithm quantizes the specified colormap into 32 distinct levels per color component. Then, for each pixel in the input image, the closest color in the quantized colormap is found.

References

[1] Spencer W. Thomas, "Efficient Inverse Color Map Computation", *Graphics Gems II*, (ed. James Arvo), Academic Press: Boston. 1991. (includes source code)

See Also

`cmunique` | `dither` | `imapprox` | `ind2rgb`

Topics

"Image Types"
"Reduce the Number of Colors in an Image" (Image Processing Toolbox)

Introduced before R2006a

rgbplot

Plot colormap

Syntax

```
rgbplot(map)
```

Description

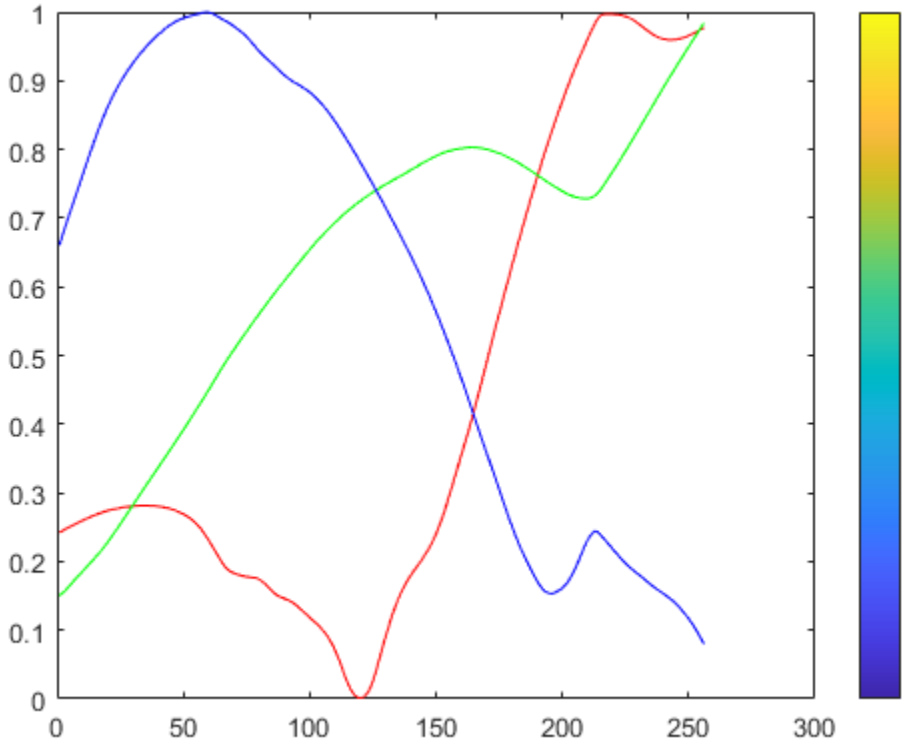
`rgbplot(map)` plots the red, green, and blue intensities of the specified colormap.

Examples

Plot a Predefined Colormap

Plot the `parula` colormap, and display a colorbar as a visual reference.

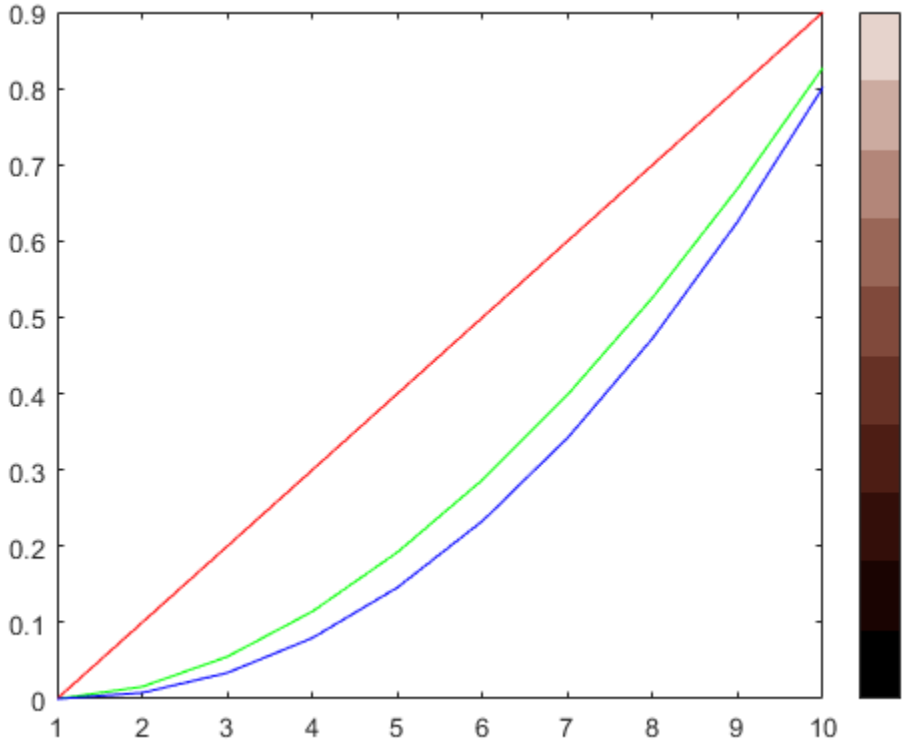
```
rgbplot(parula)
hold on
colormap(parula)
colorbar('Ticks',[])
```



Plot a Custom Colormap

Create a custom colormap and plot it. Then display a colorbar as a visual reference.

```
r = (0:.1:.9)';
g = r.^1.8;
b = r.^2.1;
mymap = [r g b];
rgbplot(mymap)
hold on
colormap(mymap)
colorbar('Ticks',[])
```



Input Arguments

`map` — Colormap

three-column matrix of RGB triplets

Colormap to plot, specified as a three-column matrix of RGB triplets. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of a color. The intensities must be in the range [0, 1]. For example, here is a colormap that contains five colors:

```
map = [0.2 0.1 0.5
    0.1 0.5 0.8
    0.2 0.7 0.6
    0.8 0.7 0.3
    0.9 1 0];
```

This table lists the RGB triplet values for common colors.

Color	RGB Triplet
yellow	`[1 1 0]`
magenta	`[1 0 1]`
cyan	`[0 1 1]`
red	`[1 0 0]`
green	`[0 1 0]`
blue	`[0 0 1]`
white	`[1 1 1]`
black	`[0 0 0]`

Alternatively, you can create the matrix by calling one of the predefined colormap functions. Call the function as the input argument to the `rgbplot` function. For example, this command plots the `parula` colormap.

```
rgbplot(parula)
```

Data Types: `double` | `single`

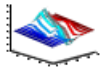
See Also

`colormap`

Introduced before R2006a

ribbon

Ribbon plot



Syntax

```
ribbon(Y)
ribbon(X,Y)
ribbon(X,Y,width)
ribbon(axes_handle,...)
h = ribbon(...)
```

Description

`ribbon(Y)` plots the columns of `Y` as three-dimensional ribbons of uniform width using `X = 1:size(Y,1)`. Ribbons advance along the *x*-axis centered on tick marks at unit intervals, three-quarters of a unit in width. Ribbon maps values in `X` to colors in colormap linearly. To change ribbon colors in the graph, change the `colormap`.

`ribbon(X,Y)` plots three dimensional ribbons for data in `Y`, centered at locations specified in `X`. `X` and `Y` are vectors or matrices of the same size. Additionally, `X` can be a row or a column vector, and `Y` a matrix with `length(X)` rows. When `Y` is a matrix, `ribbon` plots each column in `Y` as a ribbon at the corresponding `X` location.

`ribbon(X,Y,width)` specifies the width of the ribbons. The default is `0.75`. If `width = 1`, the ribbons touch, leaving no space between them when viewed down the *z*-axis. If `width > 1`, ribbons overlap and can intersect.

`ribbon(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

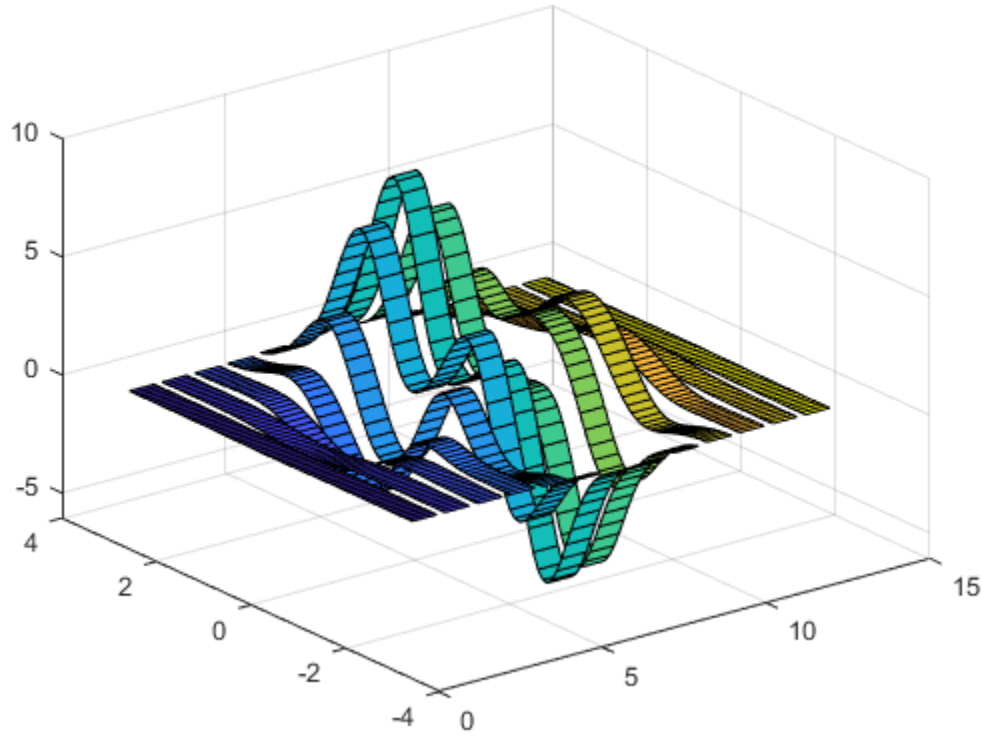
`h = ribbon(...)` returns a vector of handles to surface graphics objects. `ribbon` returns one handle per strip.

Examples

Ribbon Plot

Create a ribbon plot of the `peaks` function.

```
[x,y] = meshgrid(-3:.5:3,-3:.1:3);
z = peaks(x,y);

figure
ribbon(y,z)
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`plot` | `plot3` | `surface` | `waterfall`

Introduced before R2006a

rlim

Set or query *r*-axis limits for polar axes

Syntax

```
rlim(limits)

rlim('auto')
rlim('manual')

rl = rlim
m = rlim('mode')

___ = rlim(pax,___)
```

Description

`rlim(limits)` specifies the *r*-axis limits for the current polar axes. Specify `limits` as a two-element vector of the form `[rmin rmax]`, where `rmax` is a numeric value greater than `rmin`.

`rlim('auto')` lets MATLAB choose the *r*-axis limits. This command sets the `RLimMode` property for the polar axes object to `'auto'`.

`rlim('manual')` prevents the limits from changing automatically. Use this option if you want to retain the current limits when adding new data to the polar axes using the `hold on` command. This command sets the `RLimMode` property for the polar axes object to `'manual'`.

`rl = rlim` returns a two-element vector containing the limits for the current polar axes.

`m = rlim('mode')` returns the current value of the limits mode, which is either `'auto'` or `'manual'`. By default, the mode is automatic unless you specify limits or set the mode to manual.

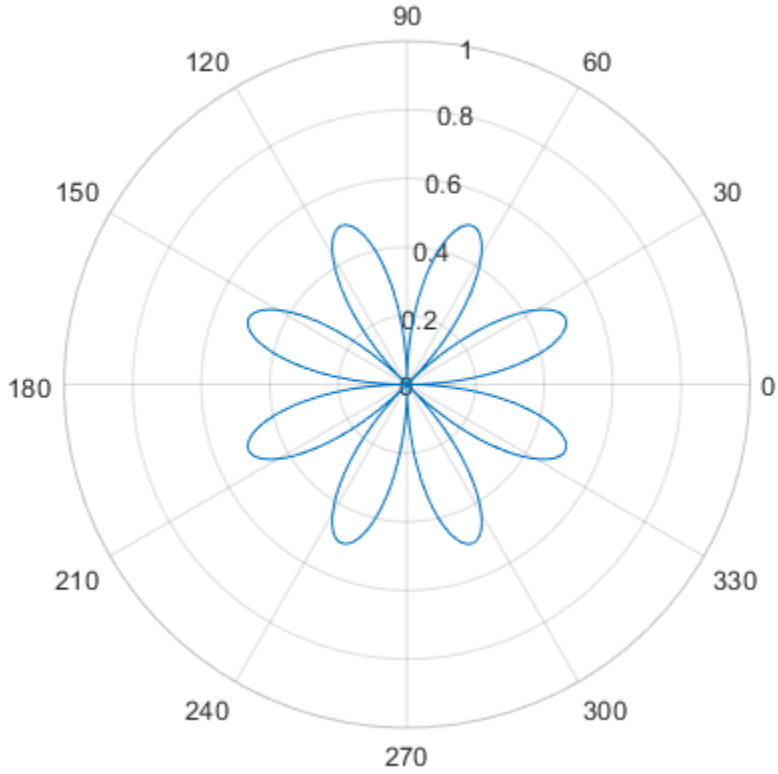
`____ = rlim(pax, ____)` uses the polar axes specified by `pax` instead of the current polar axes. Specify `pax` as the first input argument. Include additional input or output arguments only if the original syntax supported them.

Examples

Specify r-Axis Limits

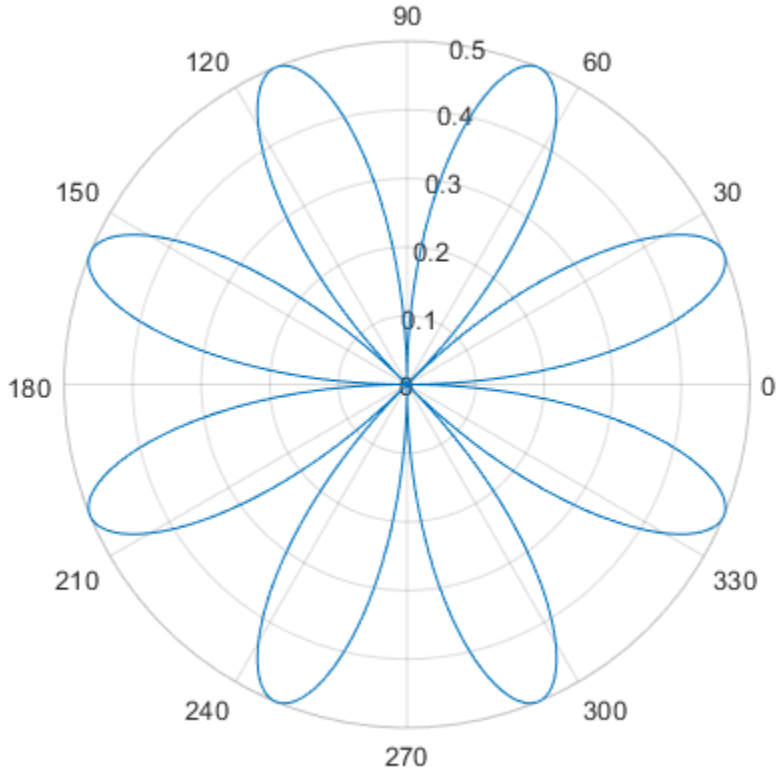
Create a polar plot and change the *r*-axis limits.

```
theta = 0:0.01:2*pi;
rho = sin(2*theta).*cos(2*theta);
polarplot(theta,rho)
rlim([0 1])
```



Set the limits back to the original values.

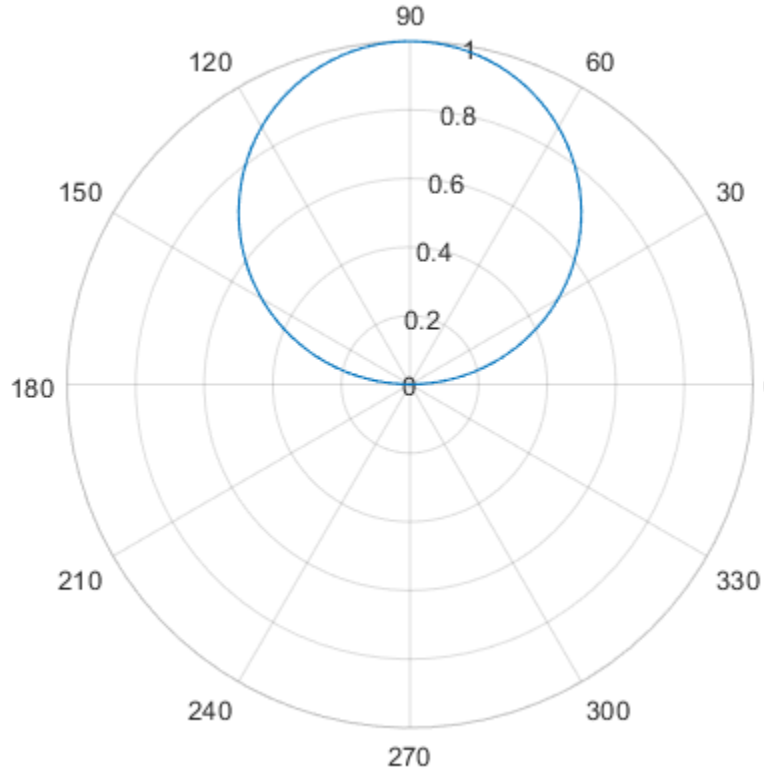
```
rlim('auto')
```



Plot Negative Radius Values

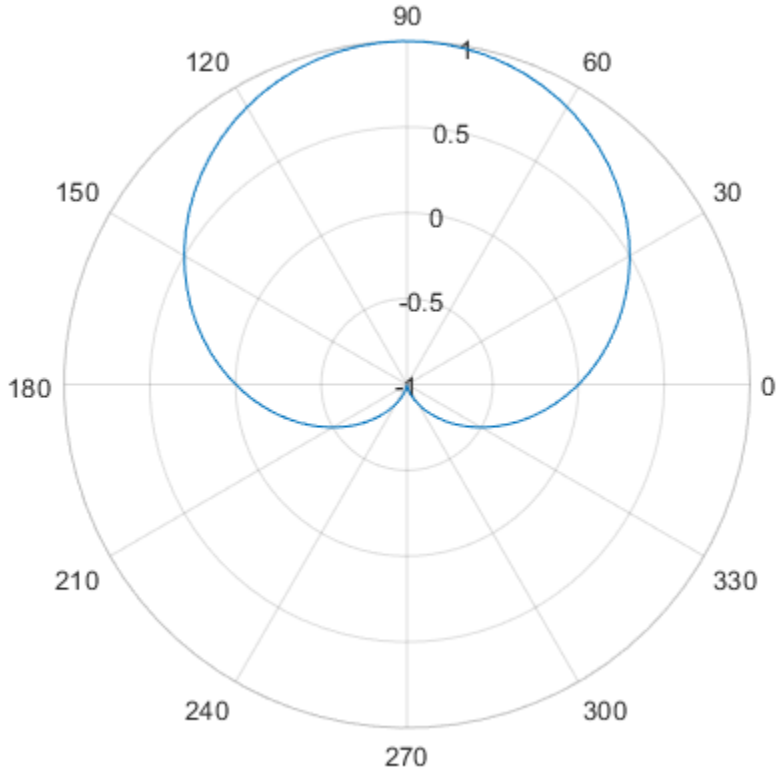
Create a polar plot using negative radius values. By default, `polarplot` reflects negative values through the origin.

```
theta = linspace(0,2*pi);
rho = sin(theta);
polarplot(theta,rho)
```



Change the limits of the *r*-axis so it ranges from -1 to 1.

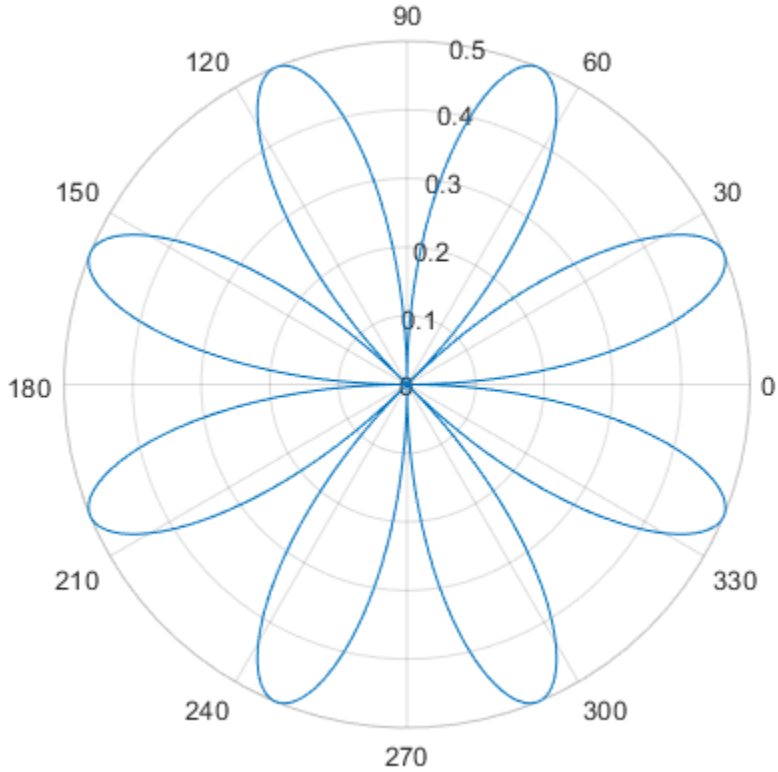
```
rlim([-1 1])
```



Return r-Axis Limits

Create a polar plot and return the *r*-axis limits.

```
theta = 0:0.01:2*pi;
rho = sin(2*theta).*cos(2*theta);
polarplot(theta,rho)
```



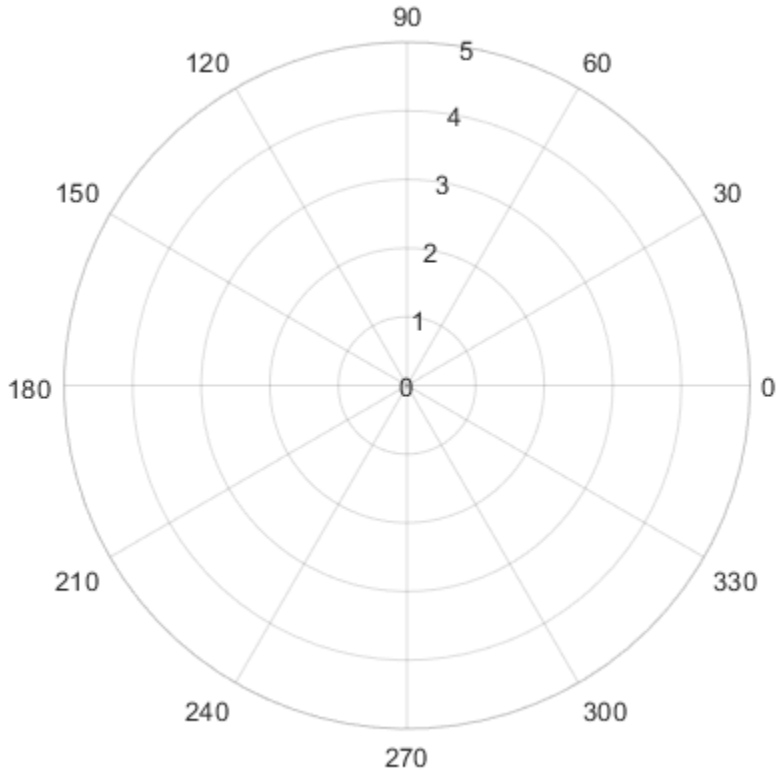

```
rl = rlim
```

```
rl = 1×2

         0    0.5000
```

Specify *r*-Axis Limits for Specific Polar Axes

Set the limits for a specific polar axes by specifying the polar axes object as the first input to `rlim`. Otherwise, `rlim` sets the limits for the current axes.

```
pax = polaraxes;
rlim(pax,[0 5])
```



Input Arguments

`limits` — Minimum and maximum limits
two-element vector

Minimum and maximum limits, specified as a two-element vector of the form `[rmin rmax]`, where `rmax` is a numeric value greater than `rmin`. You can specify both limits, or specify one limit and let MATLAB automatically calculate the other.

- To automatically set the minimum limit to the minimum data value, specify the first element as `-inf`, for example, `rlim([-inf 0])`.
- To automatically set the maximum limit to the maximum data value, specify the second element as `inf`, for example, `rlim([0 inf])`.

When you specify the limits, the `RLim` property for the polar axes object updates to the specified values and the `RLimMode` property changes to `'manual'`.

Example: `rlim([0 1])`

pax — Polar axes object
polar axes object

Polar axes object. If you do not specify a polar axes object, then `rlim` sets the limits for the current polar axes.

Output Arguments

`rl` — Current limits
two-element vector

Current limits, returned as a two-element vector of the form `[rmin rmax]`. Querying the limits returns the value of the `RLim` property for the polar axes object.

`m` — Current limits mode
`'auto'` | `'manual'`

Current limits mode, returned as one of these values:

- `'auto'` — The limits automatically update to reflect changes in the data.
- `'manual'` — The limits do not automatically update to reflect changes in the data.

Querying the *r*-axis limits mode returns the value of the `RLimMode` property for the polar axes object.

See Also

Functions
`deg2rad` | `polarplot` | `rad2deg` | `thetalim` | `title`

Properties
PolarAxes

Introduced in R2016a

rmappdata

Remove application-defined data

Syntax

```
rmappdata(h,name)
```

Description

`rmappdata(h,name)` removes the application-defined data, `name`, from its association with the UI component, `h`.

See Also

`getappdata` | `isappdata` | `setappdata`

Introduced before R2006a

rmdir

Remove folder

Syntax

```
rmdir folderName
rmdir folderName s
```

```
status = rmdir( ___ )
[status,msg] = rmdir( ___ )
[status,msg,msgID] = rmdir( ___ )
```

Description

`rmdir folderName` removes the folder `folderName` from the current folder. `folderName` must be empty. If the operation is not successful, MATLAB throws an error to the Command Window.

`rmdir folderName s` also attempts to remove all subfolders and files in `folderName`, regardless of their write permissions. The result for read-only files follows the practices of the operating system.

`status = rmdir( ___ )` removes the specified folder and returns a status of `1` if the operation is successful. Otherwise, `rmdir` returns `0`. Warnings and errors are not thrown to the Command Window. You can use this syntax with any of the input argument combinations in the previous syntaxes.

`[status,msg] = rmdir( ___ )` also returns the message text for any warning or error that occurs.

`[status,msg,msgID] = rmdir( ___ )` also returns the message ID for any warning or error that occurs.

Examples

Remove Folders from Current Folder

Create the folders `myproject` and `myproject/myfiles` in the current folder, and then remove them.

```
mkdir myproject
mkdir myproject/myfiles

rmdir myproject/myfiles
rmdir myproject
```

Remove Nonempty Folder

Remove the folder `myfiles`, which contains the files `myfile1.m` and `myfile2.m`.

Create the folder `myfiles` and move the files `myfile1.m` and `myfile2.m` from the current folder into the new folder.

```
mkdir myfiles
movefile myfile1.m myfiles
movefile myfile2.m myfiles
```

Try to remove the folder `myfiles` using `rmdir`. Because the `myfiles` folder is not empty, the operation fails and returns a status of 0 and an error message detailing why the operation failed.

```
[status, message, messageid] = rmdir('myfiles')
```

```
status = logical
   0


message =
'No directories were removed.'

messageid =
'MATLAB:RMDIR:NoDirectoriesRemoved'
```

Now, use the 's' flag to remove the folder `myfiles`. A status of 1 and an empty `message` and `messageid` indicate that the operation is successful.

```
[status, message, messageid] = rmdir('myfiles', 's')
```

```
status = logical
   1


message =

  0x0 empty char array


messageid =

  0x0 empty char array
```

Input Arguments

`folderName` — Folder name
character vector | string scalar

Folder name to remove, specified as a character vector or string scalar. Specify `folderName` as an absolute or relative path.

Data Types: `char` | `string`

Output Arguments

`status` — Folder removal status
0 | 1

Status of folder indicating if the attempt to remove the folder is successful, specified as `0` or `1`. If the attempt is successful, `status` is 1. Otherwise, `status` is 0.

Data Types: `logical`

`msg` — Error message
character vector

Error message, specified as a character vector. If an error or warning occurs, `msg` contains the message text of the error or warning. Otherwise, `msg` is empty, `''`.

`msgID` — Error message identifier
character vector

Error message identifier, specified as a character vector. If an error or warning occurs, `msgID` contains the message identifier of the error or warning. Otherwise, `msgID` is empty, `''`.

Alternative Functionality

In the Current Folder browser, right-click the folder name and select **Delete** from the context menu. To open the Current Folder browser, use the **Current Folder Browser** command.

See Also

Current Folder Browser | `copyfile` | `delete` | `dir` | `mkdir` | `movefile`

Introduced before R2006a

rmdir

Remove folder on FTP server

Syntax

```
rmdir(ftpobj,folder)
```

Description

`rmdir(ftpobj,folder)` removes the specified folder from the current folder on the FTP server associated with `ftpobj`.

Examples

Remove Folder

Connect to an FTP server and remove a folder on the server. This example shows a hypothetical FTP session on `ftp.example.com`, a machine that does not exist. If you have an account on an FTP server that grants you permission to upload or change files on that server, then you can use the `rmdir` function as shown in this example.

First, connect to the server.

```
ftpobj = ftp('ftp.example.com')
```

```
ftpobj = 

  FTP Object
     host: ftp.example.com
     user: anonymous
      dir: /
     mode: binary
```

Display the contents of the current folder on the FTP server.

```
dir(ftpobj)
```

```
myscript.m                  README.txt                pub
```

Remove the pub folder from the FTP server.

```
rmdir(ftpobj,'pub')
```

Display the updated contents on the FTP server.

```
dir(ftpobj)
```

```
myscript.m                  README.txt
```

Input Arguments

`ftpobj` — Connection to FTP server
FTP object

Connection to an FTP server, specified as an FTP object.

`folder` — Remote folder
character vector | string scalar

Remote folder to delete, specified as a character vector or string scalar.

See Also

`cd` | `delete` | `dir` | `ftp` | `mkdir`

Introduced before R2006a

rmfield

Remove fields from structure

Syntax

```
s = rmfield(s,field)
```

Description

`s = rmfield(s,field)` removes the specified field or fields from structure array `s`. Specify multiple fields using a cell array of character vectors or a string array. The dimensions of `s` remain the same.

Examples

Remove Single Field

Define a scalar structure with fields named `a`, `b`, and `c`.

```
s.a = 1;
s.b = 2;
s.c = 3;
```

Remove field `b`.

```
field = 'b';
s = rmfield(s,field)
```

```
s = struct with fields:
    a: 1
    c: 3
```

Remove Multiple Fields

Define a scalar structure with fields `first`, `second`, `third`, and `fourth`.

```
S.first = 1;
S.second = 2;
S.third = 3;
S.fourth = 4;
```

Remove fields `first` and `fourth`.

```
fields = {'first','fourth'};
S = rmfield(S,fields)
```

```
S = struct with fields:
    second: 2
     third: 3
```

Input Arguments

`s` — Input structure
structure array

Input structure, specified as a structure array.

Data Types: `struct`

`field` — Field name or names
character array | cell array of character vectors | string array

Field name or names, specified as a character array, cell array of character vectors, or string array.

Example: `'f1'`

Example: `{'f1';'f2'}`

Data Types: `char` | `cell` | `string`

Extended Capabilities

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`fieldnames` | `isfield` | `orderfields`

Topics

“Generate Field Names from Variables”

Introduced before R2006a

rmmissing

Remove missing entries

Syntax

```
R = rmmissing(A)
R = rmmissing(A,dim)
R = rmmissing( ___ ,Name,Value)
[R,TF] = rmmissing( ___ )
```

Description

`R = rmmissing(A)` removes missing entries from an array or table. If `A` is a vector, then `rmmissing` removes any entry that contains missing data. If `A` is a matrix or table, then `rmmissing` removes any row that contains missing data. Missing values are defined according to the data type of `A`:

- `NaN` — `double`, `single`, `duration`, and `calendarDuration`
- `NaT` — `datetime`
- `<missing>` — `string`
- `<undefined>` — `categorical`
- `' '` — `char`
- `{''}` — `cell` of character arrays

`R = rmmissing(A,dim)` specifies the dimension of `A` to operate along. By default, `rmmissing` operates along the first dimension whose size does not equal 1.

`R = rmmissing( ___ ,Name,Value)` specifies additional parameters for removing missing entries using one or more name-value pair arguments. For example, you can use `rmmissing('MinNumMissing',n)` to remove rows that contain at least `n` missing values.

`[R,TF] = rmmissing( ___ )` also returns a logical vector corresponding to the rows or columns of `A` that were removed.

Examples

Vector with NaN Values

Create a vector with NaN values and remove each NaN.

```
A = [1 3 NaN 6 NaN];
R = rmmissing(A)
```

```
R = 1×3

     1     3     6
```

Table with Multiple Data Types

Remove incomplete rows from a table with multiple data types.

First, create a table whose variables include `categorical`, `double`, and `char` data types.

```
A = table(categorical({'';'F';'M'}),[45;32;NaN],{'';'CA';'MA'},[6051;7234;NaN],...
    'VariableNames',{'Gender' 'Age' 'State' 'ID'})
```

```
A=3×4 table
      Gender       Age      State        ID
    ___________    ___    __________    ____

    <undefined>     45    {0x0 char}    6051
    F               32    {'CA'    }    7234
    M              NaN    {'MA'    }     NaN
```

Remove any row of the table that contains missing data.

```
R = rmmissing(A)
```

```
R=1×4 table
    Gender    Age    State      ID
    ______    ___    ______    ____
```

```
      F        32     {'CA'}    7234
```

Only remove rows with missing values in the `Age` or `ID` table variables.

```
R = rmmissing(A,'DataVariables',{'Age','ID'})
```

```
R=2×4 table
      Gender       Age      State        ID
    ___________    ___    __________    ____

    <undefined>    45     {0x0 char}    6051
    F              32     {'CA'    }    7234
```

Alternatively, use the `isnumeric` function to identify the numeric variables to operate on.

```
R = rmmissing(A,'DataVariables',@isnumeric)
```

```
R=2×4 table
      Gender       Age      State        ID
    ___________    ___    __________    ____

    <undefined>    45     {0x0 char}    6051
    F              32     {'CA'    }    7234
```

Matrix with Missing Data

Create a matrix with missing data and remove any column (second dimension) containing two or more missing values. Return the new matrix and the logical row vector that indicates which columns of `A` were removed.

```
A = [NaN NaN 5 3 NaN 5 7 NaN 9 2;
     8 9 NaN 1 4 5 6 5 NaN 5;
     NaN 4 9 8 7 2 4 1 NaN 3]
```

```
A = 3×10

   NaN   NaN     5     3   NaN     5     7   NaN     9     2
     8     9   NaN     1     4     5     6     5   NaN     5
   NaN     4     9     8     7     2     4     1   NaN     3
```

```
[R,TF] = rmmissing(A,2,'MinNumMissing',2)
```

```
R = 3×8

   NaN     5     3   NaN     5     7   NaN     2
     9   NaN     1     4     5     6     5     5
     4     9     8     7     2     4     1     3


TF = 1x10 logical array

   1   0   0   0   0   0   0   0   1   0
```

Input Arguments

A — Input data
vector | matrix | cell array of character vectors | table | timetable

Input data, specified as a vector, matrix, table, or timetable. If `A` is a `timetable`, then `rmmissing(A)` removes any row of `A` containing missing data and also removes the corresponding time vector element. If the time vector contains a `NaT` or `NaN`, then `rmmissing(A)` removes it from the time vector and also removes the corresponding row of `A`.

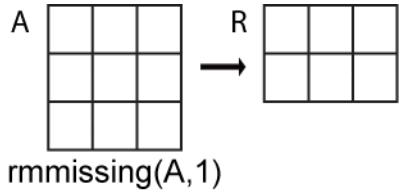
Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `cell` | `table` | `timetable` | `categorical` | `datetime` | `duration` | `calendarDuration`

`dim` — Dimension to operate along
1 | 2

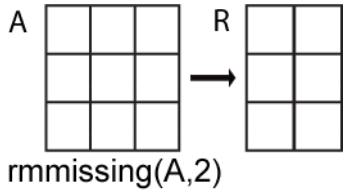
Dimension to operate along, specified as 1 or 2. By default, `rmmissing` operates along the first dimension whose size does not equal 1.

Consider a two-dimensional input array `A`.

- If `dim=1`, then `rmmissing` removes rows of `A`.



- If `dim=2`, then `rmmissing` removes columns of `A`.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `rmmissing(A,'DataVariables',{'Temperature','Altitude'})` removes rows of `A` that contain missing data in the `Temperature` or `Altitude` variables

`MinNumMissing` — Minimum number missing
non-negative scalar

Minimum number of missing entries required to remove a row or column, specified as the comma-separated pair consisting of `'MinNumMissing'` and a non-negative scalar, which is 1 by default.

Example: `'MinNumMissing',6`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`DataVariables` — Table variables
variable name | cell array of variable names | numeric vector | logical vector | function handle

Table variables, specified as the comma-separated pair consisting of `'DataVariables'` and a variable name, a cell array of variable names, a numeric vector, a logical vector, or a function handle. When operating on the rows of `A`, `rmmissing` removes any row that has missing data in the column corresponding to the variables specified. When operating on the columns of `A`, `rmmissing` removes the specified variables from the table. The value for `'DataVariables'` can be one of the following:

- A character vector specifying a single table variable name
- A cell array of character vectors where each element is a table variable name
- A vector of table variable indices
- A logical vector whose elements each correspond to a table variable, where `true` includes the corresponding variable and `false` excludes it
- A function handle that returns a logical scalar, such as `@isnumeric`

Example: `'Age'`

Example: `{'Height','Weight'}`

Example: `@iscategorical`

Data Types: `char` | `cell` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `function_handle`

Output Arguments

`R` — Data with missing entries removed
vector | matrix | table | timetable

Data with missing entries removed, returned as a vector, matrix, table, or timetable. The size of `R` depends on the number of removed rows or columns.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `cell` | `table` | `timetable` | `categorical` | `datetime` | `duration` | `calendarDuration`

`TF` — Removed entry indicator
vector

Removed entry indicator, returned as a logical vector. The value 1 (`true`) corresponds to rows or columns in `R` that were removed. The value 0 (`false`) corresponds to unchanged rows and columns. The orientation and size of `TF` depends on `A` and the dimension of operation.

Data Types: `logical`

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- The `'DataVariables'` name-value pair cannot specify a function handle.
- `rmmissing(A,2)` is not supported for tall tables.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Tables are not supported.

See Also

Clean Missing Data | `fillmissing` | `ismissing` | `isnan` | `isnat` | `standardizeMissing`

Topics

Clean Messy Data and Locate Extrema Using Live Editor Tasks
“Missing Data in MATLAB”

Introduced in R2016b

rmoutliers

Detect and remove outliers in data

Syntax

```
B = rmoutliers(A)
B = rmoutliers(A,method)
B = rmoutliers(A,'percentiles',threshold)
B = rmoutliers(A,movmethod,window)
B = rmoutliers( ___ ,dim)
B = rmoutliers( ___ ,Name,Value)
[B,TF] = rmoutliers( ___ )
```

Description

`B = rmoutliers(A)` detects and removes outliers from the data in a vector, matrix, table, or timetable.

- If `A` is a row or column vector, `rmoutliers` detects outliers and removes them.
- If `A` is a matrix, table, or timetable, `rmoutliers` detects outliers in each column or variable of `A` separately and removes the entire row.

By default, an outlier is a value that is more than three scaled median absolute deviations (MAD).

`B = rmoutliers(A,method)` specifies a method for determining outliers. For example, `rmoutliers(A,'mean')` defines an outlier as an element of `A` more than three standard deviations from the mean.

`B = rmoutliers(A,'percentiles',threshold)` defines outliers as points outside of the percentiles specified in `threshold`. The `threshold` argument is a two-element row vector containing the lower and upper percentile thresholds, such as `[10 90]`.

`B = rmoutliers(A,movmethod,window)` specifies a moving method for detecting local outliers according to a specified window. For example,

`rmoutliers(A,'movmean',5)` defines outliers as elements more than three local standard deviations away from the local mean within a five-element window.

`B = rmoutliers( ___ ,dim)` removes outliers along dimension `dim` of `A` for any of the previous syntaxes. For example, `rmoutliers(A,2)` removes columns instead of rows for a matrix `A`.

`B = rmoutliers( ___ ,Name,Value)` specifies additional parameters for detecting and removing outliers using one or more name-value pair arguments. For example, `rmoutliers(A,'SamplePoints',t)` detects outliers in `A` relative to the corresponding elements of a time vector `t`.

`[B,TF] = rmoutliers( ___ )` also returns a logical vector corresponding to the rows or columns of `A` that were removed.

Examples

Remove Outliers in Vector

Create a vector containing two outliers, and remove them. `TF` allows you to identify which elements of the input vector were detected as outliers and removed.

```
A = [57 59 60 100 59 58 57 58 300 61 62 60 62 58 57];
[B,TF] = rmoutliers(A)
```

```
B = 1×13

    57    59    60    59    58    57    58    61    62    60    62    58    57


TF = 1x15 logical array

   0   0   0   1   0   0   0   0   1   0   0   0   0   0   0
```

```
A(TF)
```

```
ans = 1×2

   100   300
```

Detect Outliers using Mean

Remove outliers of a vector where an outlier is defined as a point more than three standard deviations from the mean of the data.

```
A = [57 59 60 100 59 58 57 58 300 61 62 60 62 58 57];
[B,TF] = rmoutliers(A,'mean')
```

```
B = 1×14

    57    59    60   100    59    58    57    58    61    62    60    62    58    57


TF = 1x15 logical array

   0   0   0   0   0   0   0   0   1   0   0   0   0   0   0
```

```
A(TF)
```

```
ans = 300
```

Detect Outliers with Sliding Window

Create a vector of data containing a local outlier.

```
x = -2*pi:0.1:2*pi;
A = sin(x);
A(47) = 0;
```

Create a time vector that corresponds to the data in `A`.

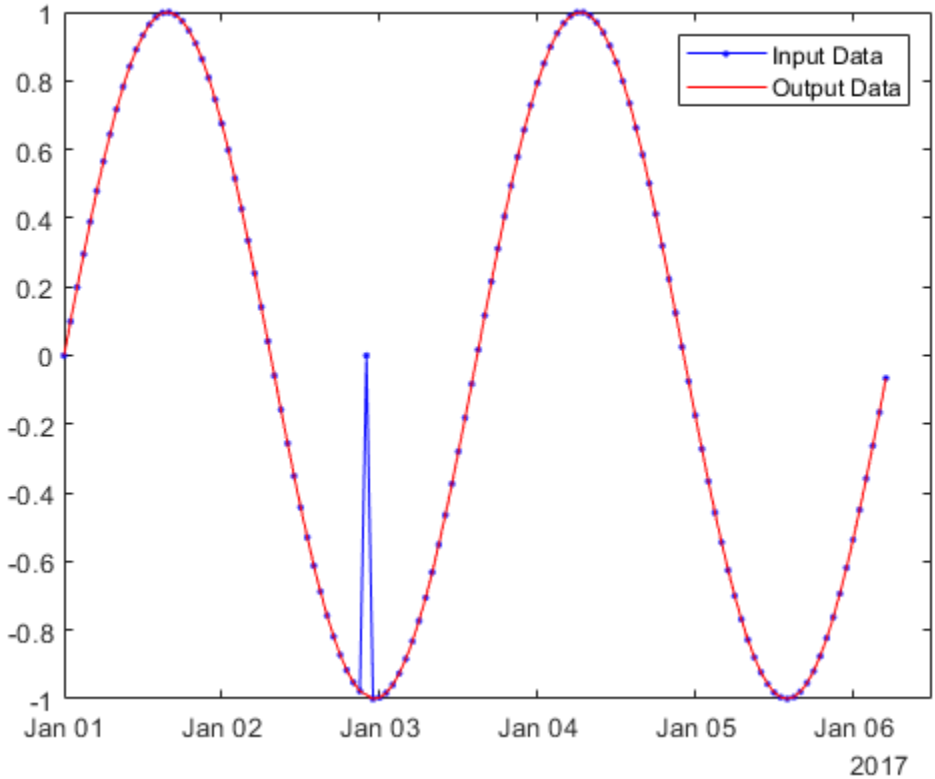
```
t = datetime(2017,1,1,0,0,0) + hours(0:length(x)-1);
```

Define outliers as points more than three local scaled MAD away from the local median within a sliding window. Find the locations of the outliers in `A` relative to the points in `t` with a window size of 5 hours, and remove them.

```
[B,TF] = rmoutliers(A,'movmedian',hours(5),'SamplePoints',t);
```

Plot the input data and the data with the outlier removed.

```
plot(t,A,'b.-',t(~TF),B,'r-')
legend('Input Data','Output Data')
```



Remove Columns Containing Outliers

Create a matrix containing two outliers, and remove the columns containing them.

```
A = magic(5);
A(4,4) = 500;
```

```
A(5,5) = 500;
A
```

```
A = 5×5

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19   500     3
    11    18    25     2   500
```

```
B = rmoutliers(A,2)
```

```
B = 5×3

    17    24     1
    23     5     7
     4     6    13
    10    12    19
    11    18    25
```

Input Arguments

A — Input data
vector | matrix | table | timetable

Input data, specified as a vector, matrix, table, or timetable.

Data Types: `double` | `single`

method — Method for detecting outliers
`'median'` (default) | `'mean'` | `'quartiles'` | `'grubbs'` | `'gesd'`

Method for detecting outliers, specified as one of the following:

Method	Description
`'median'`	Outliers are defined as elements more than three scaled MAD from the median. The scaled MAD is defined as `c*median(abs(A-median(A)))`, where `c=-1/(sqrt(2)*erfcinv(3/2))`.
`'mean'`	Outliers are defined as elements more than three standard deviations from the mean. This method is faster but less robust than `'median'`.
`'quartiles'`	Outliers are defined as elements more than 1.5 interquartile ranges above the upper quartile (75 percent) or below the lower quartile (25 percent). This method is useful when the data in `A` is not normally distributed.
`'grubbs'`	Outliers are detected using Grubbs's test for outliers, which removes one outlier per iteration based on hypothesis testing. This method assumes that the data in `A` is normally distributed.
`'gesd'`	Outliers are detected using the generalized extreme Studentized deviate test for outliers. This iterative method is similar to `'grubbs'`, but can perform better when there are multiple outliers masking each other.

`threshold` — Percentile thresholds

two-element row vector

Percentile thresholds, specified as a two-element row vector whose elements are in the interval [0,100]. The first element indicates the lower percentile threshold and the second element indicates the upper percentile threshold. For example, a threshold of `[10 90]` defines outliers as points below the 10th percentile and above the 90th percentile. The first element of `threshold` must be less than the second element.

`movmethod` — Moving method
`'movmedian'` | `'movmean'`

Moving method for determining outliers, specified as one of the following:

Method	Description
`'movmedian'`	Outliers are defined as elements more than three local scaled MAD from the local median over a window length specified by `window`.
`'movmean'`	Outliers are defined as elements more than three local standard deviations from the local mean over a window length specified by `window`.

`window` — Window length
scalar | two-element vector

Window length, specified as a scalar or two-element vector.

When `window` is a positive integer scalar, the window is centered about the current element and contains `window-1` neighboring elements. If `window` is even, then the window is centered about the current and previous elements.

When `window` is a two-element vector of positive integers `[b f]`, the window contains the current element, `b` elements backward, and `f` elements forward.

When `A` is a timetable or `'SamplePoints'` is specified as a `datetime` or `duration` vector, `window` must be of type `duration`, and the windows are computed relative to the sample points.

`dim` — Operating dimension
1 (default) | 2

Operating dimension, specified as 1 or 2. By default, `rmoutliers` operates along the first dimension whose size does not equal 1.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `rmoutliers(A,'ThresholdFactor',4)`

`ThresholdFactor` — Detection threshold factor
nonnegative scalar

Detection threshold factor, specified as the comma-separated pair consisting of `'ThresholdFactor'` and a nonnegative scalar.

For methods `'median'` and `'movmedian'`, the detection threshold factor replaces the number of scaled MAD, which is 3 by default.

For methods `'mean'` and `'movmean'`, the detection threshold factor replaces the number of standard deviations from the mean, which is 3 by default.

For methods `'grubbs'` and `'gesd'`, the detection threshold factor is a scalar ranging from 0 to 1. Values close to 0 result in a smaller number of outliers and values close to 1 result in a larger number of outliers. The default detection threshold factor is 0.5.

For the `'quartile'` method, the detection threshold factor replaces the number of interquartile ranges, which is 1.5 by default.

This name-value pair is not supported when the specified method is `'percentiles'`.

`SamplePoints` — Sample points
vector

Sample points, specified as the comma-separated pair consisting of `'SamplePoints'` and a vector. The sample points represent the location of the data in `A`, and must be sorted and contain unique elements. Sample points do not need to be uniformly sampled. If `A` is a timetable, then the default sample points vector is the vector of row times. Otherwise, the default vector is `[1 2 3 ...]`.

Moving windows are defined relative to the sample points. For example, if `t` is a vector of times corresponding to the input data, then `rmoutliers(rand(1,10),'movmean',3,'SamplePoints',t)` has a window that represents the time interval between `t(i)-1.5` and `t(i)+1.5`.

When the sample points vector has data type `datetime` or `duration`, then the moving window length must have type `duration`.

Data Types: `single` | `double` | `datetime` | `duration`

`DataVariables` — Table variables
variable name | cell array of variable names | numeric vector | logical vector | function handle

Table variables, specified as the comma-separated pair consisting of `'DataVariables'` and a variable name, a cell array of variable names, a numeric vector, a logical vector, or a function handle. The `'DataVariables'` value indicates which columns of the input table to detect outliers in, and can be one of the following:

- A character vector specifying a single table variable name
- A cell array of character vectors where each element is a table variable name
- A vector of table variable indices
- A logical vector whose elements each correspond to a table variable, where `true` includes the corresponding variable and `false` excludes it
- A function handle that takes the table as input and returns a logical scalar

Example: `'Age'`

Example: `{'Height','Weight'}`

Example: `@isnumeric`

Data Types: `char` | `cell` | `double` | `single` | `logical` | `function_handle`

`MinNumOutliers` — Minimum outlier count
1 (default) | positive integer scalar

Minimum outlier count, specified as the comma-separated pair consisting of `'MinNumOutliers'` and a positive scalar. The `'MinNumOutliers'` value specifies the minimum number of outliers required to remove a row or column. For example, `rmoutliers(A,'MinNumOutliers',3)` removes a row of a matrix `A` when there are 3 or more outliers detected in that column.

`MaxNumOutliers` — Maximum outlier count
positive scalar

Maximum outlier count, for the `'gesd'` method only, specified as the comma-separated pair consisting of `'MaxNumOutliers'` and a positive scalar. The `'MaxNumOutliers'` value specifies the maximum number of outliers returned by the `'gesd'` method. For example, `rmoutliers(A,'MaxNumOutliers',5)` returns no more than five outliers.

The default value for `'MaxNumOutliers'` is the integer nearest to 10 percent of the number of elements in `A`. Setting a larger value for the maximum number of outliers can ensure that all outliers are detected, but at the cost of reduced computational efficiency.

Output Arguments

B — Data with outliers removed
vector | matrix | table | timetable

Data with outliers removed, returned as a vector, matrix, table, or timetable. The size of `B` depends on the number of removed rows or columns.

TF — Removed data indicator
logical vector

Removed data indicator, returned as a logical vector. The value 1 (`true`) corresponds to rows or columns in `A` that were removed. The value 0 (`false`) corresponds to unchanged rows or columns. The orientation and size of `TF` depends on `A` and the dimension of operation.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- The `'percentiles'`, `'grubbs'`, and `'gesd'` methods are not supported.
- The `'movmedian'` and `'movmean'` methods do not support tall timetables.
- The `'SamplePoints'` and `'MaxNumOutliers'` name-value pairs are not supported.
- The value of `'DataVariables'` cannot be a function handle.
- Computation of `rmoutliers(A)`, `rmoutliers(A,'median',...)`, or `rmoutliers(A,'quartiles',...)` along the first dimension is only supported for tall column vectors `A`.

- `rmoutliers(A,2)` is not supported for tall tables.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The `'movmean'` and `'movmedian'` methods do not support the `'SamplePoints'` name-value pair argument.
- Tables are not supported.

See Also

Clean Outlier Data | `fillmissing` | `filloutliers` | `ismissing` | `isoutlier` | `rmmissing`

Topics

Clean Messy Data and Locate Extrema Using Live Editor Tasks
“Data Smoothing and Outlier Detection”

Introduced in R2018b

rmpath

Remove folders from search path

Syntax

```
rmpath(folderName)
```

Description

`rmpath(folderName)` removes the specified folder from the search path.

Examples

Remove Folder from Search Path

Remove `/usr/local/matlab/mytools` from the search path.

```
rmpath('/usr/local/matlab/mytools')
```

Input Arguments

`folderName` — Name of folder
character vector | string scalar

Name of folder to remove from the search path, specified as a character vector or string scalar. Use the full path name for `folderName`.

Example: `'c:\matlab\work'`

Example: `'/home/user/matlab'`

MATLAB resolves all path names containing `'.'`, `'..'`, and symbolic links to their target location before removing them from the path. For example, if you specify `c:\matlab\..\work`, MATLAB removes the folder `c:\work` from the path.

Data Types: `char` | `string`

See Also

`addpath` | `path` | `savepath`

Topics

"What Is the MATLAB Search Path?"

Introduced before R2006a

rmpref

Remove custom preference

Syntax

```
rmpref(group,pref)
rmpref(group)
```

Description

`rmpref(group,pref)` removes the specified preference in the specified group. If `pref` specifies multiple preferences, `rmpref` removes each one. If a specified preference does not exist, MATLAB returns an error.

`rmpref(group)` removes the specified group and all the preferences in that group. If the group does not exist, MATLAB returns an error.

Examples

Remove a Preference

Add a preference called `version` to the `mytoolbox` group of preferences and then remove it.

```
addpref('mytoolbox','version','1.0')
rmpref('mytoolbox','version')
```

Input Arguments

group — Custom preference group name
character vector | string scalar

Custom preference group name, specified as a character vector or a string scalar.

Example: `'mytoolbox'`

Data Types: `char` | `string`

`pref` — Custom preference name
character vector | cell array of character vectors | string array

Custom preference name, specified as a character vector, a cell array of character vectors, or a string array.

Example: `'version'`

Example: `{'version','modifieddate','docpath'}`

Data Types: `char` | `string`

See Also

`addpref` | `getpref` | `ispref` | `setpref` | `uigetpref` | `uisetpref`

Topics

"Preferences"

Introduced before R2006a

rmprop

Remove custom properties from table or timetable

Syntax

```
T = rmprop(T,propertyNames)
```

Description

`T = rmprop(T,propertyNames)` removes properties that contain custom metadata from the table or timetable `T`. The input argument `propertyNames` specifies the names of the properties.

Examples

Remove Custom Properties

Add properties that contain custom metadata to a table. Then remove some of the properties.

First, read measurements of humidity and air quality into a table. Display the first three rows.

```
T = readtable('indoors.csv');
head(T,3)
```

```
ans=3×3 table
           Time            Humidity    AirQuality
    ____________________    ________    __________

    2015-11-15 00:00:24       36           80
    2015-11-15 01:13:35       36           80
    2015-11-15 02:26:47       37           79
```

Add properties for custom metadata using the `addprop` function. Then assign metadata to those properties.

```
T = addprop(T,{'Instrument','Precision','SourceFile'},{'variable','variable','table'});
T.Properties.CustomProperties.Instrument = ["clock" "hygrometer" "air quality meter"];
T.Properties.CustomProperties.Precision = [NaN 0.5 0.1];
T.Properties
```

```
ans = 
  TableProperties with properties:

             Description: ''
                UserData: []
          DimensionNames: {'Row'   'Variables'}
           VariableNames: {'Time'   'Humidity'   'AirQuality'}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                RowNames: {}

   Custom Properties (access using t.Properties.CustomProperties.<name>):
              SourceFile: []
              Instrument: ["clock"    "hygrometer"    "air quality meter"]
               Precision: [NaN 0.5000 0.1000]
```

To remove properties, use the `rmprop` function. The only properties that you can remove are the custom properties that you previously added using `addprop`. You cannot remove the other properties in `T.Properties`, although you can delete the values they contain.

Remove the `Instrument` and `SourceFile` properties from `T.Properties.CustomProperties`.

```
T = rmprop(T,{'Instrument','SourceFile'});
T.Properties
```

```
ans = 
  TableProperties with properties:

             Description: ''
                UserData: []
          DimensionNames: {'Row'   'Variables'}
           VariableNames: {'Time'   'Humidity'   'AirQuality'}
    VariableDescriptions: {}
           VariableUnits: {}
```

```
    VariableContinuity: []
               RowNames: {}

Custom Properties (access using t.Properties.CustomProperties.<name>):
            Precision: [NaN 0.5000 0.1000]
```

Input Arguments

T — Input table
table | timetable

Input table, specified as a table or timetable.

propertyNames — Names of custom properties
character vector | cell array of character vectors | string array

Names of the custom properties, specified as a character vector, cell array of character vectors, or string array.

See Also

`addprop` | `summary` | `table` | `timetable`

Topics

"Add Custom Properties to Tables and Timetables"
"Access Data in Tables"
"Modify Units, Descriptions, and Table Variable Names"

Introduced in R2018b

rng

Control random number generation

Syntax

```
rng(seed)
rng('shuffle')
rng(seed, generator)
rng('shuffle', generator)
rng('default')
scurr = rng
rng(s)
sprev = rng(...)
```

Description

Note To use the `rng` function instead of `rand` or `randn` with the `'seed'`, `'state'`, or `'twister'` inputs, see the documentation on “Replace Discouraged Syntaxes of rand and randn”.

`rng(seed)` seeds the random number generator using the nonnegative integer `seed` so that `rand`, `randi`, and `randn` produce a predictable sequence of numbers.

`rng('shuffle')` seeds the random number generator based on the current time. Thus, `rand`, `randi`, and `randn` produce a different sequence of numbers after each time you call `rng`.

`rng(seed, generator)` and `rng('shuffle', generator)` additionally specify the type of the random number generator used by `rand`, `randi`, and `randn`. The `generator` input is one of:

- `'twister'`: Mersenne Twister
- `'simdTwister'`: SIMD-oriented Fast Mersenne Twister

- `'combRecursive'`: Combined Multiple Recursive
- `'philox'`: Philox 4x32 generator with 10 rounds
- `'threefry'`: Threefry 4x64 generator with 20 rounds
- `'multFibonacci'`: Multiplicative Lagged Fibonacci
- `'v5uniform'`: Legacy MATLAB 5.0 uniform generator
- `'v5normal'`: Legacy MATLAB 5.0 normal generator
- `'v4'`: Legacy MATLAB 4.0 generator

`rng('default')` puts the settings of the random number generator used by `rand`, `randi`, and `randn` to their default values. This way, the same random numbers are produced as if you restarted MATLAB. The default settings are the Mersenne Twister with seed 0.

`scurr = rng` returns the current settings of the random number generator used by `rand`, `randi`, and `randn`. The settings are returned in a structure `scurr` with fields 'Type', 'Seed', and '`State`'.

`rng(s)` restores the settings of the random number generator used by `rand`, `randi`, and `randn` back to the values captured previously with a command such as `s = rng`.

`sprev = rng(...)` returns the previous settings of the random number generator used by `rand`, `randi`, and `randn` before changing the settings.

Examples

Example 1 — Retrieve and Restore Generator Settings

Save the current generator settings in `s`:

```
s = rng;
```

Call `rand` to generate a vector of random values:

```
x = rand(1,5)

x =

    0.8147    0.9058    0.1270    0.9134    0.6324
```

Restore the original generator settings by calling `rng`. Generate a new set of random values and verify that `x` and `y` are equal:

```
rng(s);
y = rand(1,5)

y =

    0.8147    0.9058    0.1270    0.9134    0.6324
```

Example 2 — Restore Settings for Legacy Generator

Use the legacy generator.

```
sprev = rng(0,'v5uniform')

sprev =
     Type: 'twister'
     Seed: 0
    State: [625x1 uint32]

x = rand

x =

    0.9501
```

Restore the previous settings by calling `rng`:

```
rng(sprev)
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `'twister'`, `'v5normal'`, and `'v4'` generators are supported.

- The generated code for `rng('shuffle')` might produce different seeds than MATLAB produces.
- For a MEX target:
  - If extrinsic calls are disabled or `rng` is called inside a `parfor` loop, the output of `rng` in the MEX function is not compatible with the `rng` function in MATLAB. You cannot pass the output of `s = rng` from the MEX function to `rng` in MATLAB.
  - If extrinsic calls are enabled and `rng` is not called from inside a `parfor` loop, only `rng` can access the data in the structure that `rng` returns.

See Also

`RandStream` | `now` | `rand` | `randi` | `randn`

Topics

"Generate Random Numbers That Are Repeatable"
"Generate Random Numbers That Are Different"
"Creating and Controlling a Random Number Stream"
"Why Do Random Numbers Repeat After Startup?"

Root Properties

Graphics environment and state information

Description

The `Root` object is the root of the graphics object tree. `Root` properties contain information about the graphics environment and the current state of the graphics system. Starting in R2014b, you can use dot notation to refer to a particular object and property:

```
r = groot;
fig = r.Children;
```

If you are using an earlier release, use the `get` function to query property values.

Properties

Display Information

`MonitorPositions` — Width and height of displays
n-by-4 matrix

This property is read-only.

Width and height of displays, returned as an n-by-4 matrix, where n is the number of displays. Each row corresponds to one display and is a four-element vector of the form `[x y width height]`. For example, if there are two displays, then the matrix has this form:

```
[x1 y1 width1 height1
 x2 y2 width2 height2]
```

The first two elements in each row indicate the display location with respect to the origin point. The last two elements in each row indicate the display size. The origin point is the lower-left corner of the primary display. If the units are pixels, then the origin point is `(1,1)`. For all other units, the origin point is `(0,0)`. The `Units` property determines the units of this measurement.

Note MATLAB sets the display information values for this property at startup. The values are static. If your system display settings change, for example, if you plug in a new monitor, then the values do not update. To refresh the values, restart MATLAB.

`PointerLocation` — Current location of pointer

two-element vector

Current location of pointer, specified as a two-element vector of the form `[x y]`. The `x` and `y` values are the coordinates of the pointer position measured from the origin point. The origin point is the lower-left corner of the primary display. If the units are pixels, then the origin point is `(1,1)`. For all other units, the origin point is `(0,0)`. The `Units` property determines the units of this measurement.

This property contains the current pointer location, even if the pointer is outside a MATLAB window. Move the pointer by changing the values of this property. On Macintosh systems, you cannot change the pointer location by setting this property.

Querying the `PointerLocation` property in a callback routine might return a value that is different from the location of the pointer when the callback was triggered. This difference results from delays in callback execution caused by competition for system resources.

Example: `[500 400]`

`ScreenDepth` — Number of bits that define each pixel color

scalar

Number of bits that define each pixel color, specified as a scalar. The default value depends on the computer. The maximum number of simultaneously displayed colors on the current graphics device equals 2 raised to the value of this property.

`ScreenPixelsPerInch` — Display resolution

scalar

This property is read-only.

Display resolution, returned as a scalar in pixels per inch. The value depends on the system.

- On Windows systems, the value is 96 DPI.
- On Macintosh systems, the value is 72 DPI.

- On Linux system, the value is determined by your system resolution.

Note The `ScreenPixelsPerInch` property became a read-only property in R2015b. To change the size of text and other elements on the screen, adjust the display scaling for your operating system.

`ScreenSize` — Size of primary display

four-element vector

This property is read-only.

Size of primary display, returned as a four-element vector of the form `[left bottom width height]`.

- The `left` and `bottom` values are both `1` when the units are pixels, and `0` for all other units.
- The `width` and `height` values are the width and height of the display, respectively.

Note Starting in R2015b on Windows systems, if the `Units` property is set to `'pixels'`, then the width and height values might differ from the screen size reported by the operating system. The values MATLAB reports are based on a pixel size of 1/96th of an inch. On Macintosh and Linux systems, the values match the size reported by the operating system.

Some important information to consider when using this property:

- The values might not represent the usable display size due to the presence of UIs, such as the Microsoft Windows task bar.
- MATLAB sets the display size values for this property at startup. The values are static. If your system display settings change, the display size values do not update. To refresh the values, restart MATLAB.

`FixedWidthFontName` — Font name for fixed-width font

character vector | string

Font name for fixed-width font, specified as a character vector or string giving the name of a system supported font. This property determines the font for axes, text, and uicontrols that have a `FontName` property set to `'FixedWidth'`. The default value

depends on the system. `'Courier New'` is the default in systems that use Latin-based characters.

Specifying the `FixedWidthFontName` property eliminates the need to hardcode font names in MATLAB applications. MATLAB attempts to set `FixedWidthFontName` property to the correct value for the system.

If you are a MATLAB application developer and want to use a fixed-width font, set the `FontName` property for axes, text, and uicontrol objects to `'FixedWidth'` instead of setting this root property. Users of the application can set the root property if they do not want to use the preselected value.

Example: `'Courier New'`

`Units` — Units for `MonitorPositions`, `ScreenSize`, and `PointerLocation`
`'pixels'` (default) | `'inches'` | `'centimeters'` | `'points'` | `'characters'` | `'normalized'`

Units for the `MonitorPositions`, `ScreenSize`, and `PointerLocation` properties, specified as one of the values shown in this table.

Units	Description
`'pixels'` (default)	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'points'`	Points. One point equals 1/72 inch.

Units	Description
`'normalized'`	Normalized with respect to the display. The lower left corner of the display maps to `(0,0)` and the upper right corner maps to `(1,1)`.
`'characters'`	Based on the default system font character size. • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.

All units are measured from the lower-left corner of the primary display. If the units are pixels, then the lower-left corner maps to `(1,1)`. For all other units, the lower-left corner maps to `(0,0)`.

If you change the units, it is good practice to return it to its default value after completing your operation to prevent affecting other functions that assume the `Units` property is set to the default value.

Identifiers

`CallbackObject` — Object whose callback is executing

`[]` (default) | graphics object

This property is read-only.

Object whose callback is executing, returned as a graphics object. For more information, see the `gcbo` command.

`CurrentFigure` — Current figure

empty `GraphicsPlaceholder` (default) | figure object

Current figure, specified as a figure object. The current figure is typically the one most recently created, clicked on, or made current by calling the `figure` function. Setting this property makes a figure the current figure without sorting it to the front of other figures on the display. However, using the `figure` function to make a figure the current figure sorts that figure to the front of the display. To become the current figure, the `HandleVisibility` property of the figure must be set to `'on'`.

This property returns an empty `GraphicsPlaceholder` array if there are no figures. However, the `gcf` command always returns a figure object. If there are no figure objects, then `gcf` creates one.

Type — Type of graphics object
`'root'`

This property is read-only.

Type of graphics object, returned as `'root'`. The `Root` object handle is always visible using the `groot` function.

Tag — Tag to associate with root
`''` (default) | character vector | string

Tag to associate with root, specified as a character vector or string. There is only one `Root` object, which you can always access using the `groot` function.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

Parent/Child

`Parent` — Parent
empty `GraphicsPlaceholder`

The `Root` object has no parent. This property is always an empty `GraphicsPlaceholder`.

`Children` — Children
empty `GraphicsPlaceholder` | array of figure objects

Children, specified as an array of figure objects that have visible handles. The `HandleVisibility` property of the figure determines if the handle is visible or hidden. This property does not contain figures with hidden handles.

Change the order of the children to change the sorting order of the figures on the display.

`HandleVisibility` — Visibility of `Root` object handle
`'on'` (default) | `'callback'` | `'off'`

This property has no effect. The `Root` object handle is always visible using the `groot` function.

`ShowHiddenHandles` — Hidden handle display
`'off'` (default) | `'on'`

Hidden handle display, specified as one of these values:

- `'off'` — Do not display hidden object handles. The `HandleVisibility` property of the object determines if the handle is visible or hidden.
- `'on'` — Expose all object handles regardless of the `HandleVisibility` property.

See Also
`groot`

Topics
"Access Property Values"
"Graphics Object Properties"

Introduced before R2006a

matlab.project.rootProject

Package: `matlab.project`

Get root project

Syntax

```
proj = matlab.project.rootProject
```

Description

`proj = matlab.project.rootProject` gets the root project of the currently loaded project and returns a project object that you can use to manipulate the root project programmatically. If no project is open, MATLAB returns an empty array. Use `matlab.project.rootProject` when working with referenced projects.

To get the project which currently has context, for example, if the project is running a shortcut, startup file, or shutdown file for a referenced project, use `currentProject` instead.

For more information about referenced projects, see “Componentize Large Projects”

Examples

Get Root Project

Open the Times Table App example project and get the root project object to manipulate the project at the command line.

```
matlab.project.example.timesTable
proj = matlab.project.rootProject
```

```
proj 

  Project with properties:
```

```
                     Name: "Times Table App"
SourceControlIntegration: "Git"
      RepositoryLocation: "C:\myProjects\examples\repositories\TimesTableApp"
   SourceControlMessages: [1×3 string]
                 ReadOnly: 0
                 TopLevel: 1
            Dependencies: [1×1 digraph]
              Categories: [1×1 matlab.project.Category]
                   Files: [1×14 matlab.project.ProjectFile]
               Shortcuts: [1×4 matlab.project.Shortcut]
             ProjectPath: [1×3 matlab.project.PathFolder]
        ProjectReferences: [1×0 matlab.project.ProjectReference]
             StartupFiles: [1×0 string]
            ShutdownFiles: [1×0 string]
              Description: "This example project contains the source code and tests for a...
               RootFolder: "C:\Users\myProjects\examples\TimesTableApp"
     ProjectStartupFolder: "C:\myProjects\examples\TimesTableApp"
```

Examine the project files.

```
files = proj.Files
```

```
files = 

  1×14 ProjectFile array with properties:

    Path
    Labels
    Revision
    SourceControlStatus
```

Output Arguments

`proj` — Project
`matlab.project.Project` object

Project, returned as a `matlab.project.Project` object. Use the `matlab.project.Project` object to programmatically manipulate the currently open project.

See Also

`currentProject` | `openProject`

Topics

"Create and Edit Projects Programmatically"

"Componentize Large Projects"

Introduced in R2019a

roots

Polynomial roots

Syntax

```
r = roots(p)
```

Description

`r = roots(p)` returns the roots of the polynomial represented by `p` as a column vector. Input `p` is a vector containing `n+1` polynomial coefficients, starting with the coefficient of x^n. A coefficient of `0` indicates an intermediate power that is not present in the equation. For example, `p = [3 2 -2]` represents the polynomial $3x^2 + 2x - 2$.

The `roots` function solves polynomial equations of the form $p_1x^n + ... + p_nx + p_{n+1} = 0$. Polynomial equations contain a single variable with nonnegative exponents.

Examples

Roots of Quadratic Polynomial

Solve the equation $3x^2 - 2x - 4 = 0$.

Create a vector to represent the polynomial, then find the roots.

```
p = [3 -2 -4];
r = roots(p)
```

```
r = 2×1

    1.5352
   -0.8685
```

Roots of Quartic Polynomial

Solve the equation $x^4 - 1 = 0$.

Create a vector to represent the polynomial, then find the roots.

```
p = [1 0 0 0 -1];
r = roots(p)
```

```
r = 4×1 complex

  -1.0000 + 0.0000i
   0.0000 + 1.0000i
   0.0000 - 1.0000i
   1.0000 + 0.0000i
```

Input Arguments

p — Polynomial coefficients
vector

Polynomial coefficients, specified as a vector. For example, the vector `[1 0 1]` represents the polynomial $x^2 + 1$, and the vector `[3.13 -2.21 5.99]` represents the polynomial $3.13x^2 - 2.21x + 5.99$.

For more information, see “Create and Evaluate Polynomials”.

Data Types: `single` | `double`
Complex Number Support: Yes

Tips

- Use the `poly` function to obtain a polynomial from its roots: `p = poly(r)`. The `poly` function is the inverse of the `roots` function.

- Use the `fzero` function to find the roots of nonlinear equations. While the `roots` function works only with polynomials, the `fzero` function is more broadly applicable to different types of equations.

Algorithms

The `roots` function considers `p` to be a vector with `n+1` elements representing the `n`th degree characteristic polynomial of an `n`-by-`n` matrix, `A`. The roots of the polynomial are calculated by computing the eigenvalues of the companion matrix, `A`.

```
A = diag(ones(n-1,1),-1);
A(1,:) = -p(2:n+1)./p(1);
r = eig(A)
```

The results produced are the exact eigenvalues of a matrix within roundoff error of the companion matrix, `A`. However, this does not mean that they are the exact roots of a polynomial whose coefficients are within roundoff error of those in `p`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Output is variable-size and always complex.
- Roots are not always in the same order as in MATLAB.
- Roots of poorly conditioned polynomials do not always match MATLAB.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The output `r` is always complex even if all the imaginary parts are zero.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`fzero` | `poly` | `polyval` | `residue`

Topics

“Roots of Polynomials”
“Create and Evaluate Polynomials”

Introduced before R2006a

rose

Angle histogram plot

Note `rose` is not recommended. Use `polarhistogram` instead.

Syntax

```
rose(theta)
rose(theta,x)
rose(theta,nbins)
rose(ax,...)
h = rose(...)
[tout,rout] = rose(...)
```

Description

`rose(theta)` creates an angle histogram, which is a polar plot showing the distribution of values grouped according to their numeric range, showing the distribution of `theta` in 20 angle bins or less. The vector `theta`, expressed in radians, determines the angle of each bin from the origin. The length of each bin reflects the number of elements in `theta` that fall within a group, which ranges from 0 to the greatest number of elements deposited in any one bin.

`rose(theta,x)` uses the vector `x` to specify the number and the locations of bins. `length(x)` is the number of bins and the values of `x` specify the center angle of each bin. For example, if `x` is a five-element vector, `rose` distributes the elements of `theta` in five bins centered at the specified `x` values.

`rose(theta,nbins)` plots `nbins` equally spaced bins in the range `[0,2*pi]`. The default is `20`.

`rose(ax,...)` plots into the axes `ax` instead of the current axes (`gca`).

`h = rose(...)` returns the handle of the line object used to create the graph.

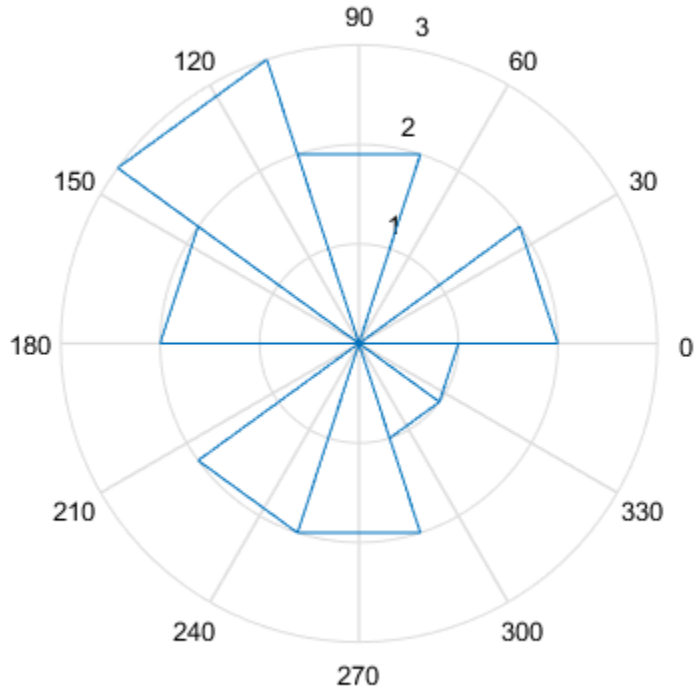
`[tout,rout] = rose(...)` returns the vectors `tout` and `rout` so `polar(tout,rout)` generates the histogram for the data. This syntax does not generate a plot.

Examples

Create Angle Histogram

Create an angle histogram of values between 0 and 2π. Distribute the data among 10 bins.

```
theta = [0.4 1.4 3.1 2.3 0.4 2.5 3.9 2.8 2.3 1.6 4.6 4.5 6.1 3.9 5.1];
rose(theta,10)
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`compass` | `histogram` | `polarhistogram` | `polarplot`

Introduced before R2006a

rosser

Classic symmetric eigenvalue test problem

Syntax

```
A = rosser
A = rosser(classname)
```

Description

`A = rosser` returns the Rosser matrix on page 1-13545 in double precision.

`A = rosser(classname)` returns the Rosser matrix with a class specified by `classname`. Specify `classname` as `'single'` to return the Rosser matrix in single precision.

Examples

Generate the Rosser matrix

`rosser` returns the Rosser matrix.

```
rosser
```

```
ans = 8×8

   611   196  -192   407    -8   -52   -49    29
   196   899   113  -192   -71   -43    -8   -44
  -192   113   899   196    61    49     8    52
   407  -192   196   611     8    44    59   -23
    -8   -71    61     8   411  -599   208   208
   -52   -43    49    44  -599   411   208   208
   -49    -8     8    59   208   208    99  -911
    29   -44    52   -23   208   208  -911    99
```

Generate matrix of class ‘single’

Specify `classname` as `single` to return a Rosser matrix of that class.

```
Y = rosser('single')
```

```
Y = 8x8 single matrix

   611   196  -192   407    -8   -52   -49    29
   196   899   113  -192   -71   -43    -8   -44
  -192   113   899   196    61    49     8    52
   407  -192   196   611     8    44    59   -23
    -8   -71    61     8   411  -599   208   208
   -52   -43    49    44  -599   411   208   208
   -49    -8     8    59   208   208    99  -911
    29   -44    52   -23   208   208  -911    99
```

```
whos('Y')
```

```
  Name      Size            Bytes  Class     Attributes

  Y         8x8               256  single
```

Input Arguments

`classname` — Input class
`'double'` (default) | `'single'`

Input class, specified as `'double'` (default) or `'single'`. `rosser(C)` produces a matrix of the specified class.

More About

Rosser Matrix

The Rosser matrix is a well known matrix used, for example, to evaluate eigenvalue algorithms. The matrix is 8-by-8 with integer elements. It has:

- A double eigenvalue
- Three nearly equal eigenvalues
- Dominant eigenvalues of the opposite sign
- A zero eigenvalue
- A small, nonzero eigenvalue

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`eig`

Introduced before R2006a

rot90

Rotate array 90 degrees

Syntax

```
B = rot90(A)
B = rot90(A,k)
```

Description

`B = rot90(A)` rotates array `A` counterclockwise by 90 degrees. For multidimensional arrays, `rot90` rotates in the plane formed by the first and second dimensions.

`B = rot90(A,k)` rotates array `A` counterclockwise by `k*90` degrees, where `k` is an integer.

Examples

Rotate Column Vector

Create a column vector of sequential elements.

```
A = (1:5)'
```

```
A = 5×1

     1
     2
     3
     4
     5
```

Rotate `A` counterclockwise by 90 degrees using `rot90`.

```
B = rot90(A)
```

```
B = 1×5

     1     2     3     4     5
```

The result, B, has the same elements as A but a different orientation.

Rotate Multidimensional Array

Create a 3-by-3-by-2 cell array of characters.

```
A = cat(3,{'a' 'b' 'c';'d' 'e' 'f';'g' 'h' 'i'},{'j' 'k' 'l';'m' 'n' 'o';'p' 'q' 'r'})
```

```
A = 3x3x2 cell array
A(:,:,1) =

    {'a'}    {'b'}    {'c'}
    {'d'}    {'e'}    {'f'}
    {'g'}    {'h'}    {'i'}


A(:,:,2) =

    {'j'}    {'k'}    {'l'}
    {'m'}    {'n'}    {'o'}
    {'p'}    {'q'}    {'r'}
```

Rotate the cell array by 270 degrees.

```
B = rot90(A,3)
```

```
B = 3x3x2 cell array
B(:,:,1) =

    {'g'}    {'d'}    {'a'}
    {'h'}    {'e'}    {'b'}
    {'i'}    {'f'}    {'c'}


B(:,:,2) =
```

```
    {'p'}    {'m'}    {'j'}
    {'q'}    {'n'}    {'k'}
    {'r'}    {'o'}    {'l'}
```

The function rotates each page of the array independently. Since a full 360 degree rotation (`k = 4`) leaves the array unchanged, `rot90(A,3)` is equivalent to `rot90(A,-1)`.

Input Arguments

A — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `cell` | `categorical` | `datetime` | `duration` | `calendarDuration`
Complex Number Support: Yes

k — Rotation constant
integer

Rotation constant, specified as an integer. Specify `k` to rotate by `k*90` degrees rather than nesting calls to `rot90`.

Example: `rot90(A,-2)` rotates `A` by -180 degrees and is equivalent to `rot90(A,2)`, which rotates by 180 degrees.

Tips

- Use the `flip` function to flip arrays in any dimension.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Does not support cell arrays for the first argument.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Does not support cell arrays for the first argument.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`flip` | `fliplr` | `flipud`

Introduced before R2006a

rotate

Rotate object about specified origin and direction

Syntax

```
rotate(h,direction,alpha)
rotate(...,origin)
```

Description

The `rotate` function rotates a graphics object in three-dimensional space.

`rotate(h,direction,alpha)` rotates the graphics object `h` by `alpha` degrees. Specify `h` as a surface, patch, line, text, or image object. `direction` is a two- or three-element vector that describes the axis of rotation in conjunction with the origin of the axis of rotation. The default origin of the axis of rotation is the center of the plot box. This point is not necessarily the origin of the axes.

Positive `alpha` is defined as the righthand-rule angle about the direction vector as it extends from the origin of rotation.

If `h` is an array of handles, all objects must be children of the same axes.

`rotate(...,origin)` specifies the origin of the axis of rotation as a three-element vector $[x_0,y_0,z_0]$.

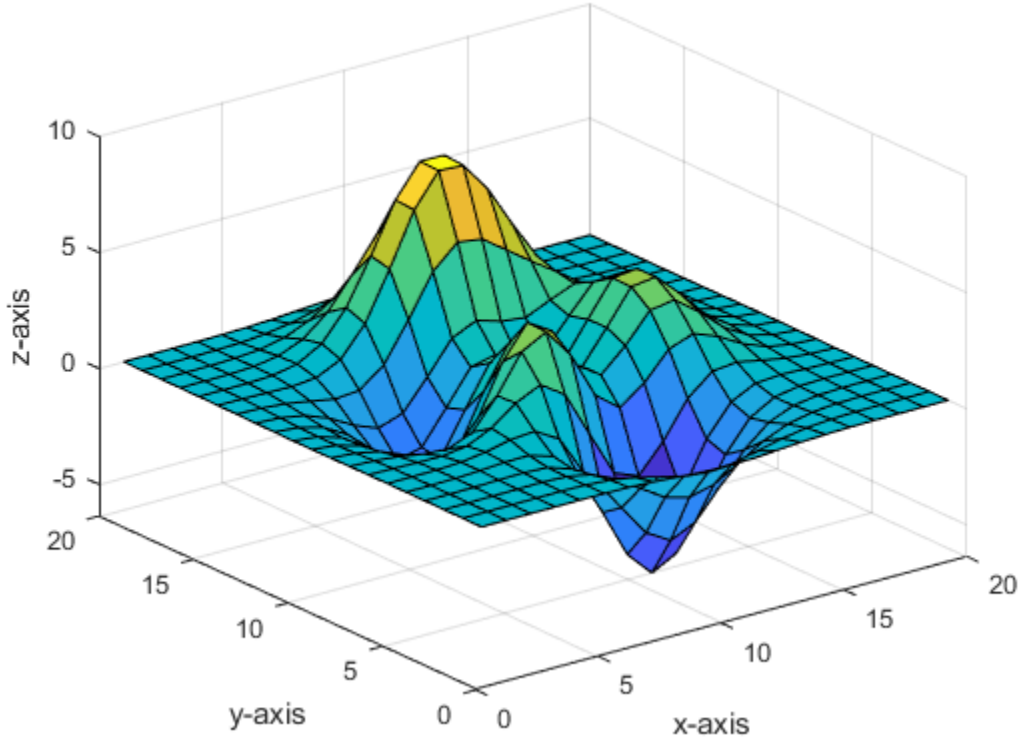
Examples

Rotate Plot Around *x*-Axis

Create a surface plot of the `peaks` function and return the surface object.

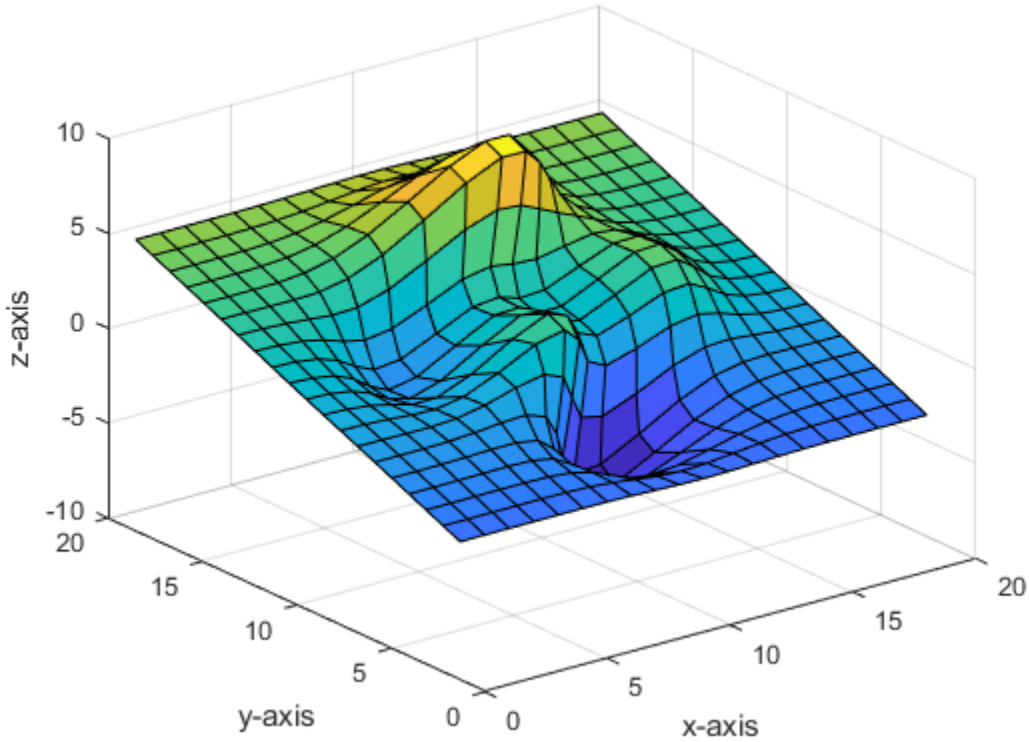
```
s = surf(peaks(20));
xlabel('x-axis')
```

```
ylabel('y-axis')
zlabel('z-axis')
```



Rotate the surface plot 25 degrees around its *x*-axis.

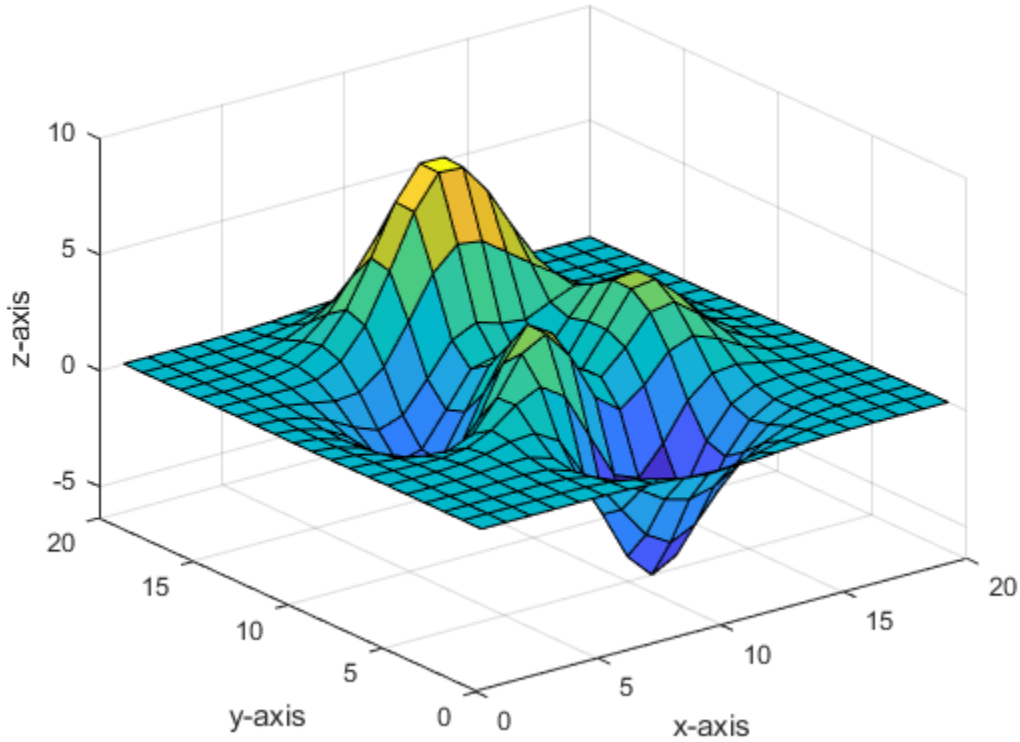
```
direction = [1 0 0];
rotate(s,direction,25)
```



Rotate Plot Around *y*-Axis

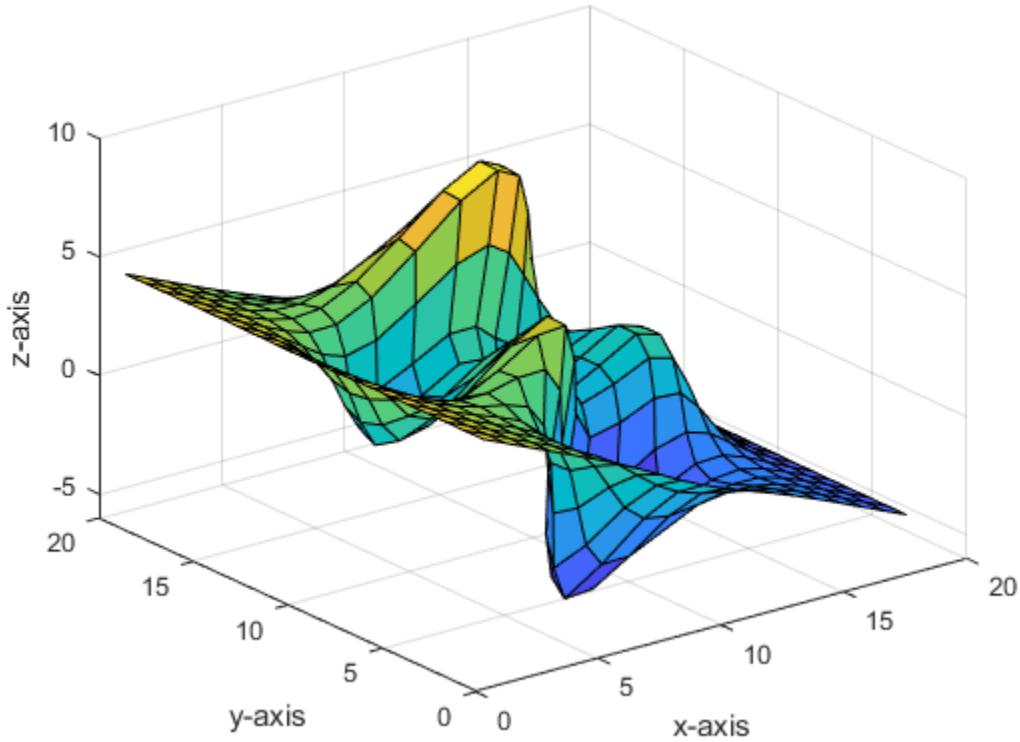
Create a surface plot of the `peaks` function and return the surface object.

```
s = surf(peaks(20));
xlabel('x-axis')
ylabel('y-axis')
zlabel('z-axis')
```



Rotate the surface plot 25 degrees around its *y*-axis.

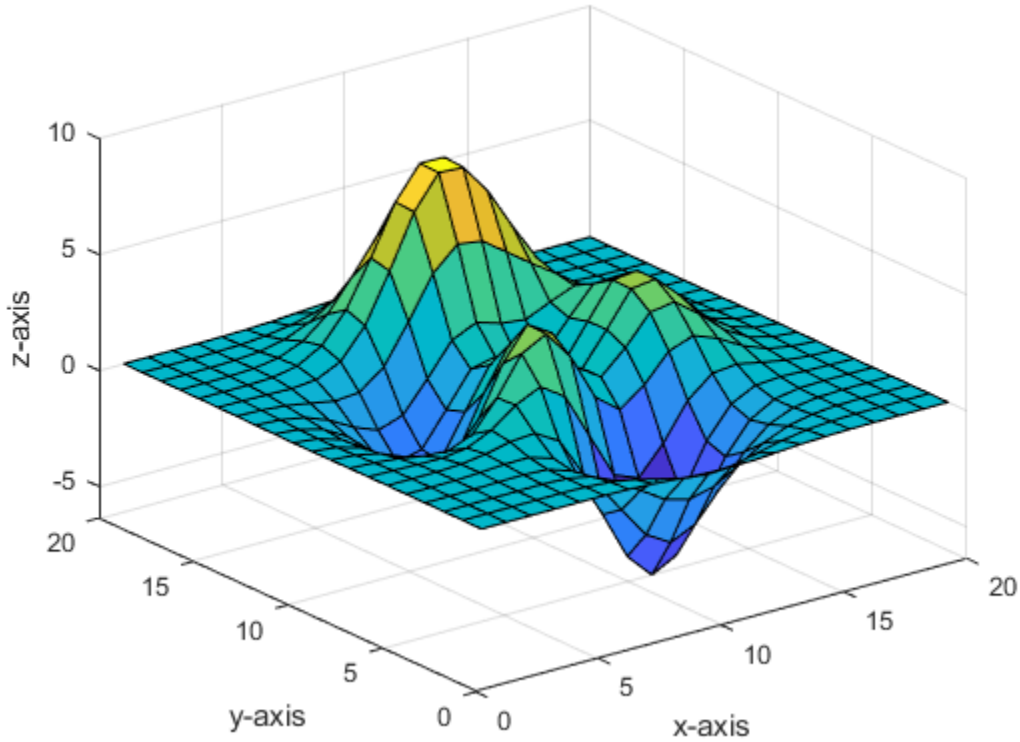
```
direction = [0 1 0];
rotate(s,direction,25)
```



Rotate Plot Around *x*-Axis and *y*-Axis

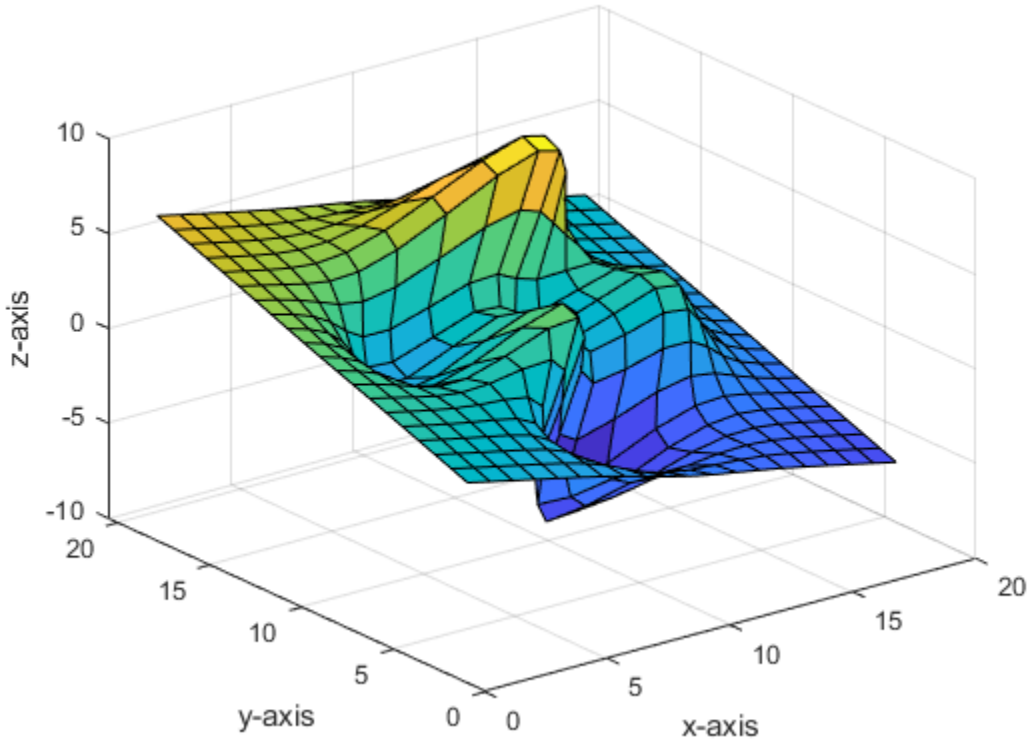
Create a surface plot of the `peaks` function and return the surface object.

```
s = surf(peaks(20));
xlabel('x-axis')
ylabel('y-axis')
zlabel('z-axis')
```



Rotate the surface plot 25 degrees around its *x*-axis and *y*-axis.

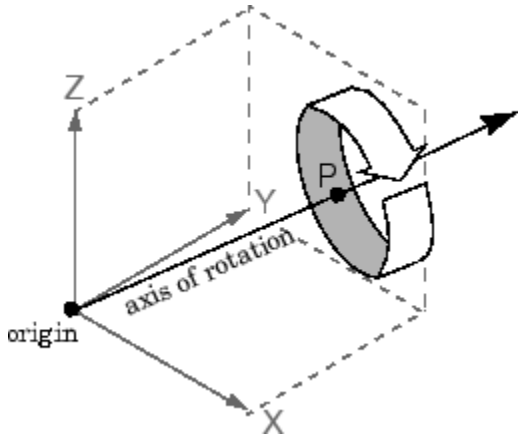
```
direction = [1 1 0];
rotate(s,direction,25)
```



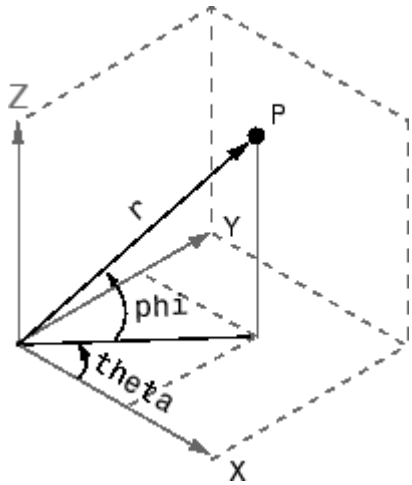
Tips

The rotation transformation modifies the object's data. This technique is different from that used by `view` and `rotate3d`, which modify only the viewpoint.

The axis of rotation is defined by an origin of rotation and a point `P`. Specify `P` as the spherical coordinates `[theta phi]` or as the Cartesian coordinates $[x_p, y_p, z_p]$.



In the two-element form for `direction`, `theta` is the angle in the *x*-*y* plane counterclockwise from the positive *x*-axis. `phi` is the elevation of the direction vector from the *x*-*y* plane.



The three-element form for `direction` specifies the axis direction using Cartesian coordinates. The direction vector is the vector from the origin of rotation to P.

Tips

`rotate` changes the values of the `Xdata`, `Ydata`, and `Zdata` properties to rotate graphics objects.

See Also

`rotate3d` | `sph2cart` | `view`

Introduced before R2006a

rotate3d

Rotate 3-D view using mouse

Syntax

```
rotate3d on
rotate3d off
rotate3d
rotate3d(figure_handle,...)
rotate3d(axes_handle,...)
h = rotate3d(figure_handle)
```

Description

`rotate3d on` turns on rotate mode and enables rotation on all axes within the current figure.

`rotate3d off` turns off rotate mode and disables interactive axes rotation in the current figure. Starting in R2018b, some rotate interactions are enabled by default, regardless of the rotate mode. If you want to disable these default interactions, then use the `disableDefaultInteractivity` function.

`rotate3d` toggles interactive axes rotation in the current figure.

`rotate3d(figure_handle,...)` enables rotation within the specified figure instead of the current figure.

`rotate3d(axes_handle,...)` turns on rotate mode only in the specified axes.

`h = rotate3d(figure_handle)` returns a rotate3d *mode object* for figure figure_handle for you to customize the mode's behavior.

Using Rotate Mode Objects

You access the following properties of rotate mode objects.

- `FigureHandle <handle>` — The associated figure handle, a read-only property that cannot be set
- *`Enable`* `'on'|'off'` — Specifies whether this figure mode is currently enabled on the figure
- *`RotateStyle`* `'orbit'|'box'` — Sets the method of rotation

  `'orbit'` rotates the entire axes; `'box'` rotates a plot-box outline of the axes.

Rotate3D Mode Callbacks

You can program the following callbacks for rotate3d mode operations.

- `ButtonDownFilter <function_handle>` — Function to intercept `ButtonDown` events

  The application can inhibit the rotate operation under circumstances the programmer defines, depending on what the callback returns. The input function handle should reference a function with two implicit arguments (similar to handle callbacks):

  ```
  function [res] = myfunction(obj,event_obj)
  % obj          handle to the object that has been clicked on
  % event_obj    handle to event data object (empty in this release)
  % res [output] logical flag to determine whether the rotate
  %              operation should take place or the 'ButtonDownFcn'
  %              property of the object should take precedence
  ```

- `ActionPreCallback <function_handle>` — Function to execute before rotating

  Set this callback to listen to when a rotate operation will start. The input function handle should reference a function with two implicit arguments (similar to graphics object callbacks):

  ```
  function myfunction(obj,event_obj)
  % obj         handle to the figure that has been clicked on
  % event_obj   object containing struct of event data
  ```

  The event data has the following field:

`Axes`	The handle of the axes that is being panned

- `ActionPostCallback <function_handle>` — Function to execute after rotating

Set this callback to listen to when a rotate operation has finished. The input function handle should reference a function with two implicit arguments (similar to graphics object callbacks):

```
function myfunction(obj,event_obj)
% obj         handle to the figure that has been clicked on
% event_obj   object containing struct of event data (same as the
%             event data of the 'ActionPreCallback' callback)
```

Rotate3D Mode Utility Functions

The following functions in pan mode query and set certain of its properties.

- `flags = isAllowAxesRotate(h,axes)` — Function querying permission to rotate axes

  Calling the function `isAllowAxesRotate` on the rotate3d object, `h`, with a vector of axes handles, `axes`, as input will return a logical array of the same dimension as the axes handle vector which indicate whether a rotate operation is permitted on the axes objects.
- `setAllowAxesRotate(h,axes,flag)` — Function to set permission to pan axes

  Calling the function `setAllowAxesRotate` on the rotate3d object, `h`, with a vector of axes handles, `axes`, and a logical scalar, `flag`, will either allow or disallow a rotate operation on the axes objects.

Examples

Example 1

Rotate the plot using the mouse:

```
surf(peaks);
rotate3d on;
```

Example 2

Rotate the plot using the "Plot Box" rotate style:

```
surf(peaks);
h = rotate3d;
h.RotateStyle = 'box';
h.Enable = 'on';
```

Example 3

Create two axes as subplots and then prevent one from rotating:

```
ax1 = subplot(1,2,1);
surf(peaks);
h = rotate3d;
h.Enable = 'on';
ax2 = subplot(1,2,2);
surf(membrane);
setAllowAxesRotate(h,ax2,false); % disable rotating for second plot
```

Example 4

Create a buttonDown callback for rotate mode objects to trigger. Copy the following code to a new file, execute it, and observe rotation behavior:

```
function demo_mbd
% Allow a line to have its own 'ButtonDownFcn' callback
hLine = plot(rand(1,10),'ButtonDownFcn','disp(''This executes'')');
hLine.Tag = 'DoNotIgnore';
h = rotate3d;
h.ButtonDownFilter = @mycallback;
h.Enable = 'on';
% mouse-click on the line
function [flag] = mycallback(obj,event_obj)
% If the tag of the object is 'DoNotIgnore', then return true
objTag = obj.Tag;
if strcmpi(objTag,'DoNotIgnore')
   flag = true;
else
   flag = false;
end
```

Example 5

Create callbacks for pre- and post-buttonDown events for rotate3D mode objects to trigger. Copy the following code to a new file, execute it, and observe rotation behavior:

```
function demo_mbd2
% Listen to rotate events
surf(peaks);
h = rotate3d;
h.ActionPreCallback = @myprecallback;
h.ActionPostCallback = @mypostcallback;
h.Enable = 'on';

function myprecallback(obj,evd)
disp('A rotation is about to occur.');

function mypostcallback(obj,evd)
newView = round(evd.Axes.View);
msgbox(sprintf('The new view is [%d %d].',newView));
```

Tips

When enabled, `rotate3d` provides continuous rotation of axes and the objects it contains through mouse movement. A numeric readout appears in the lower left corner of the figure during rotation, showing the current azimuth and elevation of the axes. Releasing the mouse button removes the animated box and the readout. This differs from the `camorbit` function in that while the `rotate3d` tool modifies the `View` property of the axes, the `camorbit` function fixes the aspect ratio and modifies the `CameraTarget`, `CameraPosition` and `CameraUpVector` properties of the axes. See Axes for more information.

You can also enable 3-D rotation from the figure **Tools** menu or the figure toolbar.

You can create a `rotate3d` mode object once and use it to customize the behavior of different axes, as example 3 illustrates. You can also change its callback functions on the fly.

Note **Do not change figure callbacks within an interactive mode.** While a mode is active (when panning, zooming, etc.), you will receive a warning if you attempt to change any of the figure's callbacks and the operation will not succeed. The one exception to this rule is the figure `WindowButtonMotionFcn` callback, which can be changed from within a mode. Therefore, if you are creating a UI that updates a figure's callbacks, the UI should some keep track of which interactive mode is active, if any, before attempting to do this.

When you assign different 3-D rotation behaviors to different `subplot` axes via a mode object and then link them using the `linkaxes` function, the behavior of the axes you manipulate with the mouse will carry over to the linked axes, regardless of the behavior you previously set for the other axes.

Alternatives

Use the Rotate3D tool on the toolbar to enable and disable rotate3D mode on a plot, or select **Rotate 3D** from the figure's **Tools** menu.

See Also

`camorbit` | `pan` | `rotate` | `view` | `zoom`

Topics

"Interactively Explore Plotted Data"

Introduced before R2006a

rotateInteraction

Rotate interaction

Description

A rotate interaction allows you to rotate a chart without having to select any buttons in the axes toolbar. To enable chart rotation, set the `Interactions` property of the axes to a `rotateInteraction` object. When this interaction is enabled, you can rotate a chart by dragging it.

To enable multiple interactions, set the `Interactions` property to an array of objects.

Creation

Syntax

```
r = rotateInteraction
```

Description

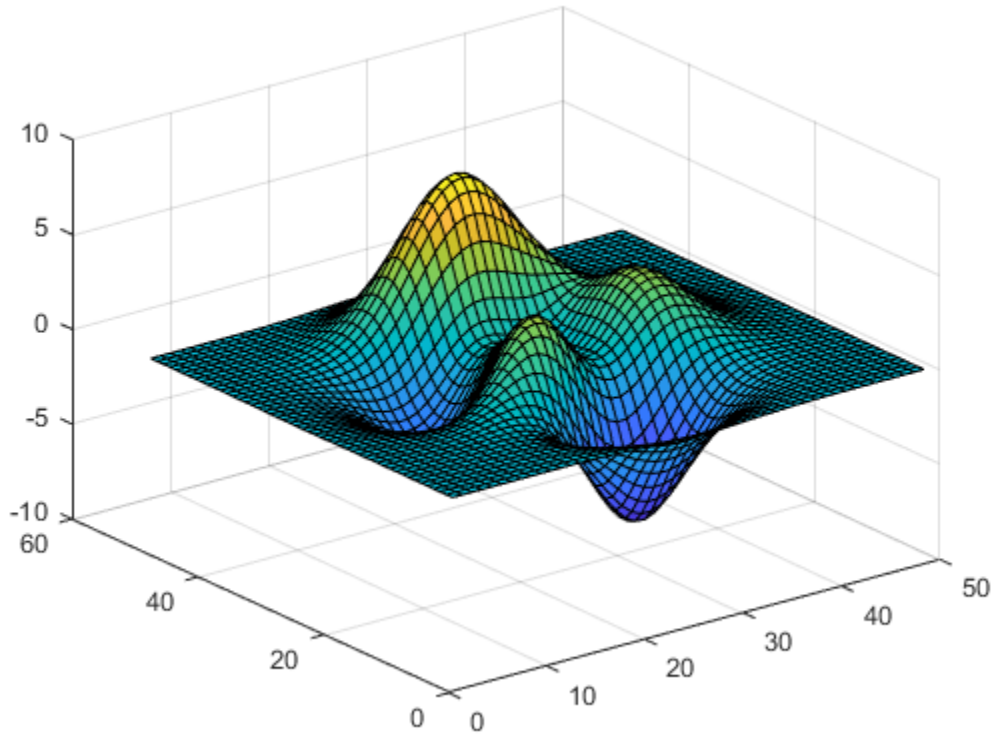
`r = rotateInteraction` creates a rotate interaction object.

Examples

Axes with Rotate and Data Tip Interactions

Create a surface plot. Get the current axes and replace the default interactions with the rotate and data tip interactions. Then hover over the surface to display data tips. Drag to rotate the plot.

```
surf(peaks)
ax = gca;
ax.Interactions = [rotateInteraction dataTipInteraction];
```



Limitations

The `rotateInteraction` function is not supported in the Live Editor.

Tips

In most cases, the axes have a default set of interactions which depend on the type of chart you are displaying. You can replace the default set with a new set of interactions, but you cannot access or modify any of the interactions in the default set.

See Also

`dataTipInteraction` | `disableDefaultInteractivity` | `enableDefaultInteractivity` | `panInteraction` | `regionZoomInteraction` | `rulerPanInteraction` | `zoomInteraction`

Topics

"Control Chart Interactivity"

Introduced in R2019a

round

Round to nearest decimal or integer

Syntax

```
Y = round(X)
Y = round(X,N)
Y = round(X,N,type)

Y = round(t)
Y = round(t,unit)
```

Description

`Y = round(X)` rounds each element of `X` to the nearest integer. In the case of a tie, where an element has a fractional part of exactly `0.5`, the `round` function rounds away from zero to the integer with larger magnitude.

`Y = round(X,N)` rounds to `N` digits:

- `N > 0`: round to `N` digits to the *right* of the decimal point.
- `N = 0`: round to the nearest integer.
- `N < 0`: round to `N` digits to the *left* of the decimal point.

`Y = round(X,N,type)` specifies the type of rounding. Specify `'significant'` to round to `N` significant digits (counted from the leftmost digit). In this case, `N` must be a positive integer.

`Y = round(t)` rounds each element of the `duration` array `t` to the nearest number of seconds.

`Y = round(t,unit)` rounds each element of `t` to the nearest number of the specified unit of time.

Examples

Round Matrix Elements

Round the elements of a 2-by-2 matrix to the nearest integer.

```
X = [2.11 3.5; -3.5 0.78];
Y = round(X)
```

```
Y = 2×2

     2     4
    -4     1
```

Round to Specified Number of Decimal Digits

Round `pi` to the nearest 3 decimal digits.

```
Y = round(pi,3)
```

```
Y = 3.1420
```

Round to Nearest Multiple of 100

Round the number `863178137` to the nearest multiple of 100.

```
round(863178137,-2)
```

```
ans = 863178100
```

Round Elements to Specified Number of Significant Digits

Round the elements of a vector to retain 2 significant digits.

```
format shortg
x = [1253 1.345 120.44]
```

```
x = 1×3

         1253        1.345       120.44
```

```
y = round(x,2,'significant')
```

```
y = 1×3

         1300          1.3          120
```

Controlling Number Display While Rounding

The `format` command controls how MATLAB® displays numbers at the command line. If a number has extra digits that cannot be displayed in the current format, then MATLAB automatically rounds the number for display purposes. This can lead to unexpected results when combined with the `round` function.

Consider the result of the following subtraction operation, which displays 5 digits.

```
format short
x = 112.05 - 110
```

```
x = 2.0500
```

Based on the displayed value of `x`, rounding `x` to 1 decimal should return `2.1`.

```
round(x,1)
```

```
ans = 2
```

In fact, the problem here is that MATLAB is rounding `x` to 5 digits for display purposes. The `round` function returns the correct answer. Confirm the answer by viewing `x` with `format long`, which displays `x` rounded to 15 digits.

```
format long
x
```

```
x =
   2.049999999999997
```

Round Duration Values

Round each value in a duration array to the nearest number of seconds.

```
t = hours(8) + minutes(29:31) + seconds(1.3:0.5:2.3);
t.Format = 'hh:mm:ss.SS'
```

```
t = 1x3 duration array
   08:29:01.30   08:30:01.80   08:31:02.30
```

```
Y1 = round(t)
```

```
Y1 = 1x3 duration array
   08:29:01.00   08:30:02.00   08:31:02.00
```

Round each value in t to the nearest number of hours.

```
Y2 = round(t,'hours')
```

```
Y2 = 1x3 duration array
   08:00:00.00   09:00:00.00   09:00:00.00
```

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. For complex X, `round` treats the real and imaginary parts independently.

X must be `single` or `double` when you use `round` with more than one input.

`round` converts logical and `char` elements of X into `double` values.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `logical`
Complex Number Support: Yes

`N` — Number of digits
scalar integer

Number of digits, specified as a scalar integer. When you specify `N`, the `round` function rounds `X` to the nearest multiple of 10^{-N}.

If you specify the `'significant'` rounding type, then `N` must be a positive integer.

`type` — Rounding type
`'decimals'` (default) | `'significant'`

Rounding type, specified as `'decimals'` or `'significant'`. The rounding type determines whether `round` considers digits in relation to the decimal point or the overall number of significant digits. `N` must be a positive integer when you specify `'significant'`. In that case, the `round` function rounds to the nearest number with `N` significant digits.

The default value is `'decimals'`, so that `round(X,N,'decimals')` is equivalent to `round(X,N)`.

Example: `round(3132,2,'significant')` returns `3100`, which is the closest number to `3132` that has `2` significant digits.

Data Types: `char` | `string`

`t` — Input duration
`duration` array

Input duration, specified as a `duration` array.

`unit` — Unit of time
`'seconds'` (default) | `'minutes'` | `'hours'` | `'days'` | `'years'`

Unit of time, specified as `'seconds'`, `'minutes'`, `'hours'`, `'days'`, or `'years'`. A duration of 1 year is equal to exactly 365.2425 24-hour days.

Data Types: `char` | `string`

Tips

- `format short` and `format long` both display rounded numbers. This can cause unexpected results when combined with the `round` function.
- For display purposes, use `sprintf` to control the exact display of a number as a string. For example, to display exactly 2 decimal digits of `pi` (and no trailing zeros), use `sprintf('%.2f',pi)`.

Compatibility Considerations

Rounding to specified number of digits

Behavior changed in R2014b

In R2014b, these syntaxes were added to round to any number of decimal or significant digits and to round duration values:

```
Y = round(X,N)
Y = round(X,N,type)
Y = round(t)
Y = round(t,unit)
```

Older versions of MATLAB support only this syntax, which rounds to the nearest integer:

```
Y = round(X)
```

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation supports only the syntax `Y = round(X)`.
- Code generation does not support `char` or `logical` data types for `X`.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Code generation supports only the syntax `Y = round(X)`.
- Code generation does not support `char` or `logical` data types for `X`.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- These syntaxes are not supported:

  ```
  Y = round(X,N)
  ```

  ```
  Y = round(X,N,type)
  ```

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`ceil` | `fix` | `floor`

Topics

"Integers"
"Floating-Point Numbers"

Introduced before R2006a

rowfun

Apply function to table or timetable rows

Syntax

```
B = rowfun(func,A)
B = rowfun(func,A,Name,Value)
```

Description

`B = rowfun(func,A)` applies the function `func` to each row of the table or timetable `A` and returns the results in the table or timetable `B`.

`func` accepts `size(A,2)` inputs.

If `A` is a timetable and `func` aggregates data over groups of rows, then `rowfun` assigns the first row time from each group of rows in `A` as the corresponding row time in `B`. To return `B` as a table without row times, specify `'OutputFormat'` as `'table'`.

`B = rowfun(func,A,Name,Value)` applies the function `func` to each row of the table `A` with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify which variables to pass to the function `func` and how to call `func`.

Examples

Apply Function with Single Output to Rows

Apply the function `hypot` to each row of the 5-by-2 table `A` to find the shortest distance between the variables `x` and `y`.

Create a table, `A`, with two variables of numeric data.

```
x = gallery('integerdata',10,[5,1],2);
y = gallery('integerdata',10,[5,1],8);

A = table(x,y)
```

```
A=5×2 table
    x    y
    _    __

    9     1
    4     5
    3     2
    7     3
    1    10
```

Apply the function, `hypot`, to each row of `A`. The function `hypot` takes two inputs and returns one output.

```
B = rowfun(@hypot,A,'OutputVariableNames','z')
```

```
B=5×1 table
       z
    ______

    9.0554
    6.4031
    3.6056
    7.6158
     10.05
```

`B` is a table.

Append the function output, `B`, to the input table, `A`.

```
[A B]
```

```
ans=5×3 table
    x    y       z
    _    __    ______

    9     1    9.0554
    4     5    6.4031
    3     2    3.6056
```

```
     7     3    7.6158
     1    10     10.05
```

Apply Function with Multiple Outputs to Rows

Define and apply a geometric Brownian motion model to a range of parameters.

Create a function in a file named `gbmSim.m` that contains the following code.

```
% Copyright 2015 The MathWorks, Inc.

function [m,mtrue,s,strue] = gbmSim(mu,sigma)
% Discrete approximation to geometric Brownian motion
%
% [m,mtrue,s,strue] = gbmSim(mu,sigma) computes the
% simulated mean, true mean, simulated standard deviation,
% and true standard deviation based on the parameters mu and sigma.
numReplicates = 1000; numSteps = 100;
y0 = 1;
t1 = 1;
dt = t1 / numSteps;
y1 = y0*prod(1 + mu*dt + sigma*sqrt(dt)*randn(numSteps,numReplicates));
m = mean(y1); s = std(y1);

% Theoretical values
mtrue = y0 * exp(mu*t1); strue = mtrue * sqrt(exp(sigma^2*t1) - 1);
end
```

`gbmSim` accepts two inputs, `mu` and `sigma`, and returns four outputs, `m`, `mtrue`, `s`, and `strue`.

Define the table, `params`, containing the parameters to input to the Brownian Motion Model.

```
mu = [-.5; -.25; 0; .25; .5];
sigma = [.1; .2; .3; .2; .1];

params = table(mu,sigma)
```

```
params =

  5x2 table

     mu      sigma
    _____    _____

     -0.5     0.1
    -0.25     0.2
        0     0.3
     0.25     0.2
      0.5     0.1
```

Apply the function, `gbmSim`, to the rows of the table, `params`.

```
stats = rowfun(@gbmSim,params,...
    'OutputVariableNames',...
    {'simulatedMean' 'trueMean' 'simulatedStd' 'trueStd'})
```

```
stats =

  5x4 table

    simulatedMean    trueMean    simulatedStd    trueStd
    _____________    ________    ____________    ________

       0.60501       0.60653       0.05808       0.060805
       0.77916        0.7788         0.161        0.15733
        1.0024             1        0.3048        0.30688
        1.2795         1.284       0.25851        0.25939
        1.6498        1.6487       0.16285        0.16529
```

The four variable names specified by the `'OutputVariableNames'` name-value pair argument indicate that `rowfun` should obtain four outputs from `gbmSim`. You can specify fewer output variable names to return fewer outputs from `gbmSim`.

Append the function output, `stats`, to the input, `params`.

```
[params stats]
```

```
ans =
```

```
5x6 table

    mu      sigma    simulatedMean    trueMean    simulatedStd    trueStd
   _____    _____    _____________    ________    ____________    ________

   -0.5      0.1        0.60501       0.60653       0.05808       0.060805
  -0.25      0.2        0.77916        0.7788         0.161        0.15733
      0      0.3         1.0024             1        0.3048        0.30688
   0.25      0.2         1.2795         1.284       0.25851        0.25939
    0.5      0.1         1.6498        1.6487       0.16285        0.16529
```

Apply Function to Groups of Rows

Create a table, A, where g is a grouping variable.

```
g = gallery('integerdata',3,[15,1],1);
x = gallery('uniformdata',[15,1],9);
y = gallery('uniformdata',[15,1],2);

A = table(g,x,y)
```

```
A=15×3 table
    g       x          y
    _    _______    ________

    3    0.24756     0.87516
    3     0.4358      0.3179
    3    0.97755     0.27323
    2    0.85995      0.6765
    3    0.30063    0.071171
    2    0.26589     0.19659
    3    0.13338     0.52908
    2     0.7425     0.17176
    1    0.85692     0.86996
    2    0.24286     0.24369
    3    0.19492     0.84291
    2    0.39076     0.55766
    1    0.29683     0.35681
    1    0.39031      0.2324
    2    0.18726      0.6476
```

Define the anonymous function, `func`, to compute the average difference between `x` and `y`.

```
func = @(x,y) mean(x-y);
```

Find the average difference between variables in groups `1`, `2`, and `3` defined by the grouping variable, `g`.

```
B = rowfun(func,A,...
    'GroupingVariable','g',...
    'OutputVariableName','MeanDiff')
```

```
B=3×3 table
    g    GroupCount    MeanDiff
    _    __________    ________

    1        3         0.028298
    2        6         0.032569
    3        6         -0.10327
```

The variable `GroupCount` indicates the number of rows in `A` for each group.

Input Arguments

`func` — Function
function handle

Function, specified as a function handle. You can define the function in a file or as an anonymous function. If `func` corresponds to more than one function file (that is, if `func` represents a set of overloaded functions), MATLAB determines which function to call based on the class of the input arguments.

`func` can accept no more than `size(A,2)` inputs. By default, `rowfun` returns the first output of `func`. To return more than one output from `func`, use the `'NumOutputs'` or `'OutputVariableNames'` name-value pair arguments.

Example: `func = @(x,y) x.^2+y.^2;` takes two inputs and finds the sum of the squares.

`A` — Input table
table | timetable

Input table, specified as a table or a timetable.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'InputVariables',2` uses only the second variable in `A` as an input to `func`.

`InputVariables` — Specifiers for selecting variables of `A` to pass to `func`
positive integer | vector of positive integers | character vector | cell array of character vectors | string array | logical vector | function handle

Specifiers for selecting variables of `A` to pass to `func`, specified as the comma-separated pair consisting of `'InputVariables'` and a positive integer, vector of positive integers, character vector, cell array of character vectors, string array, logical vector, or a function handle.

If you specify `'InputVariables'` as a function handle, then it must return a logical scalar, and `rowfun` passes only the variables in `A` where the function returns `1` (`true`).

`GroupingVariables` — One or more variables in `A` that define groups of rows
positive integer | vector of positive integers | character vector | cell array of character vectors | string array | logical vector

One or more variables in `A` that define groups of rows, specified as the comma-separated pair consisting of `'GroupingVariables'` and a positive integer, vector of positive integers, character vector, cell array of character vectors, string array, or logical vector.

The value of `'GroupingVariables'` specifies which table variables are the grouping variables, not their data types. A grouping variable can be numeric, or have data type `categorical`, `calendarDuration`, `datetime`, `duration`, `logical`, or `string`.

Rows in `A` that have the same grouping variable values belong to the same group. `rowfun` applies `func` to each group of rows, rather than separately to each row of `A`. The output, `B`, contains one row for each group.

If any grouping variable contains `NaN`s or missing values (such as `NaT`s, undefined categorical values, or missing strings), then the corresponding rows do not belong to any group, and are excluded from the output.

Row labels can be grouping variables. You can group on row labels alone, on one or more variables in A, or on row labels and variables together.

- If A is a table, then the labels are row names.
- If A is a timetable, then the labels are row times.

SeparateInputs — Indicator for calling func with separate inputs
true (default) | false | 1 | 0

Indicator for calling func with separate inputs, specified as the comma-separated pair consisting of 'SeparateInputs' and either true, false, 1, or 0.

true	func expects separate inputs. rowfun calls func with size(A,2) inputs, one argument for each data variable. This is the default behavior.
false	func expects one vector containing all inputs. rowfun creates the input vector to func by concatenating the values in each row of A.

ExtractCellContents — Indicator to pass values from cell variables to func
false (default) | true | 0 | 1

Indicator to pass values from cell variables to func, specified as the comma-separated pair consisting of 'ExtractCellContents' and either false, true, 0, or 1.

true	rowfun extracts the contents of a variable in A whose data type is cell and passes the values, rather than the cells, to func For grouped computation, the values within each group in a cell variable must allow vertical concatenation.
false	rowfun passes the cells of a variable in A whose data type is cell to func. This is the default behavior.

OutputVariableNames — Variable names for outputs of func
character vector | cell array of character vectors | string array

Variable names for outputs of func, specified as the comma-separated pair consisting of 'OutputVariableNames' and a character vector, cell array of character vectors, or

string array, with names that are nonempty and distinct. The number of names must equal the number of outputs desired from `func`.

Furthermore, the variable names must be valid MATLAB identifiers. If valid MATLAB identifiers are not available for use as variable names, MATLAB uses a cell array of *N* character vectors of the form `{'Var1' ... 'VarN'}` where *N* is the number of variables. You can determine valid MATLAB variable names using the function `isvarname`.

`NumOutputs` — Number of outputs from `func`
`0` | positive integer

Number of outputs from `func`, specified as the comma-separated pair consisting of `'NumOutputs'` and `0` or a positive integer. The integer must be less than or equal to the possible number of outputs from `func`.

Example: `'NumOutputs',2` causes `rowfun` to call `func` with two outputs.

`OutputFormat` — Format of `B`
`'table'` (default) | `'timetable'` | `'uniform'` | `'cell'`

Format of `B`, specified as the comma-separated pair consisting of `'OutputFormat'` and either the value `'table'`, `'uniform'`, or `'cell'`.

`'table'`	`rowfun` returns a table with one variable for each output of `func`. For grouped computation, `B`, also contains the grouping variables. `'table'` allows you to use a function that returns values of different sizes or data types. However, for ungrouped computation, all of the outputs from `func` must have one row each time it is called. For grouped computation, all of the outputs from `func` must have the same number of rows. This is the default output format.

`'timetable'`	`rowfun` returns a timetable with one variable for each variable in `A` (or each variable specified with `'InputVariables'`). For grouped computation, `B` also contains the grouping variables. `rowfun` creates the row times of `B` from the row times of `A`. If the row times assigned to `B` do not make sense in the context of the calculations performed using `func`, then specify the output format as `'OutputFormat','table'`. If `A` is a timetable, then this is the default output format.
`'uniform'`	`rowfun` concatenates the values returned by `func` into a vector. All of the outputs from `func` must be scalars with the same data type.
`'cell'`	`rowfun` returns `B` as a cell array. `'cell'` allows you to use a function that returns values of different sizes or data types.

`ErrorHandler` — Function to call if `func` fails
function handle

Function to call if `func` fails, specified as the comma-separated pair consisting of `'ErrorHandler'` and a function handle. Define this function so that it rethrows the error or returns valid outputs for function `func`.

MATLAB calls the specified error-handling function with two input arguments:

- A structure with these fields:

`identifier`	Error identifier.
`message`	Error message text.
`index`	Row or group index at which the error occurred.

- The set of input arguments to function `func` at the time of the error.

For example,

```
function [A, B] = errorFunc(S, varargin)
warning(S.identifier, S.message);
A = NaN; B = NaN;
```

Output Arguments

B — Output table
table | timetable

Output table, returned as a table or a timetable. `B` can store metadata such as descriptions, variable units, variable names, and row names. For more information, see the Properties sections of `table` or `timetable`.

See Also

`arrayfun` | `cellfun` | `findgroups` | `groupsummary` | `isvarname` | `splitapply` | `structfun` | `varfun`

Topics

"Calculations on Tables"
"Split Table Data Variables and Apply Functions"
"Anonymous Functions"

Introduced in R2013b

rows2vars

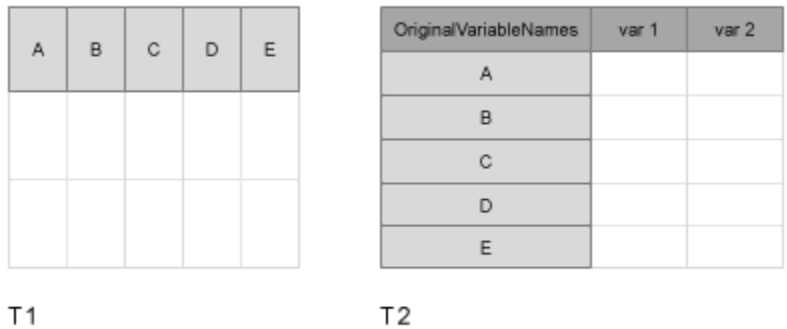
Reorient table or timetable so that rows become variables

Syntax

```
T2 = rows2vars(T1)
T2 = rows2vars(T1,'VariableNamesSource',varnames)
T2 = rows2vars( ___ ,'DataVariables',datavars)
```

Description

`T2 = rows2vars(T1)` reorients the rows of `T1`, so that they become variables in the output table `T2`, as shown in the diagram. If `rows2vars` can concatenate the contents of the rows of `T1`, then the corresponding variables of `T2` are arrays. Otherwise, the variables of `T2` are cell arrays. `rows2vars` always returns a table, though `T1` can be either a table or a timetable.



The `rows2vars` function copies the names of the variables of `T1` to a new variable of `T2`. If `T1` has row names or row times, then those names or times become the variable names

of T2. Otherwise, rows2vars generates names Var1,…,VarN as the variable names of T2.

T2 = rows2vars(T1,'VariableNamesSource',varnames) specifies a variable in T1 as the source of the names given to the variables of T2. You can specify varnames as a variable name or a numeric or logical index.

T2 = rows2vars(___ ,'DataVariables',datavars) selects the variables of T1 specified by datavars and reorients only those variables to become the rows of T2. The remaining variables of T1 are discarded. datavars can be one or more variable names, an array of numeric or logical indices, or a subscripting object (for example, the output of the vartype function). You can use this syntax with the input arguments of either of the previous syntaxes.

If you specify a variable using the 'VariableNamesSource' name-value pair argument from the previous syntax, then you cannot specify that variable in datavars.

Examples

Reorient Rows as Variables

Create tables, and then reorient their rows to be variables in new tables.

Load arrays of data from the patients.mat file. Create a table that contains the LastName, Gender, Age, Height, and Weight variables.

```
load patients
T1 = table(LastName,Gender,Age,Height,Weight);
head(T1,3)
```

```
ans=3×5 table
      LastName        Gender      Age    Height    Weight
    ____________    __________    ___    ______    ______

    {'Smith'   }    {'Male'  }    38       71       176
    {'Johnson' }    {'Male'  }    43       69       163
    {'Williams'}    {'Female'}    38       64       131
```

Reorient the rows of T1 to be the variables of the output table.

```
T2 = rows2vars(T1);
```

Display the first four variables of T2. The first variable of T2 contains the names of the variables of T1. The remaining variables of T2 correspond to rows of T1. Since T1 did not have any row labels, the variables of T2 have default names, Var1 to VarN for N variables.

```
T2(:,1:4)
```

```
ans=5×4 table
    OriginalVariableNames      Var1          Var2            Var3
    _____________________    _________    ___________    ___________

        {'LastName'}         {'Smith'}    {'Johnson'}    {'Williams'}
        {'Gender'  }         {'Male' }    {'Male'   }    {'Female'  }
        {'Age'     }         {[   38]}    {[     43]}    {[      38]}
        {'Height'  }         {[   71]}    {[     69]}    {[      64]}
        {'Weight'  }         {[  176]}    {[    163]}    {[     131]}
```

Create a table with row names. If a table has row names, then rows2vars turns the row names into the names of variables.

```
T3 = table(Gender,Age,Height,Weight,'RowNames',LastName);
head(T3,3)
```

```
ans=3×4 table
                  Gender      Age    Height    Weight
                __________    ___    ______    ______

    Smith       {'Male'  }    38       71       176
    Johnson     {'Male'  }    43       69       163
    Williams    {'Female'}    38       64       131
```

Reorient the rows of T3.

```
T4 = rows2vars(T3);
T4(:,1:4)
```

```
ans=4×4 table
    OriginalVariableNames     Smith      Johnson      Williams
    _____________________    ________    ________    __________

         {'Gender'}          {'Male'}    {'Male'}    {'Female'}
```

```
    {'Age'   }          {[  38]}    {[  43]}    {[    38]}
    {'Height'}          {[  71]}    {[  69]}    {[    64]}
    {'Weight'}          {[ 176]}    {[ 163]}    {[   131]}
```

Reorient Timetable Rows

Load a timetable and display it.

```
load bostonTT
Boston
```

```
Boston=6×3 timetable
           Time            Temp    WindSpeed    Rain
    ___________________    ____    _________    ____

    2016-06-09 06:03:00    59.5       0.1       0.05
    2016-06-09 12:00:23      63       2.3       0.08
    2016-06-09 18:02:57    61.7       3.1       0.13
    2016-06-10 06:01:47    55.4       5.7       0.15
    2016-06-10 12:06:00    62.3       2.6       0.87
    2016-06-10 18:02:57    58.8       6.2       0.33
```

Reorient it so that its rows become variables in the output. The rows2vars function turns the row times into names, but must modify them so that they are valid variable names. Also, the output argument returned by rows2vars is always a table, even when the input argument is a timetable.

```
T = rows2vars(Boston)
```

```
Warning: Variable names were modified to make them valid MATLAB identifiers.
```

```
T=3×7 table
    OriginalVariableNames    x2016_06_0906_03_00    x2016_06_0912_00_23    x2016_06_091
    _____________________    ___________________    ___________________    ____________

         'Temp'                     59.5                     63                   61.7
         'WindSpeed'                 0.1                    2.3                    3.1
         'Rain'                     0.05                   0.08                   0.13
```

Names of Output Table Variables

Read data from a spreadsheet into a table. Display the first three rows.

```
T1 = readtable('patients.xls');
head(T1,3)
```

```
ans=3×10 table
      LastName        Gender      Age              Location                Height    We
    ____________    __________    ___    ______________________________    ______    __

    {'Smith'   }    {'Male'  }    38     {'County General Hospital'  }      71       1
    {'Johnson' }    {'Male'  }    43     {'VA Hospital'              }      69       1
    {'Williams'}    {'Female'}    38     {'St. Mary's Medical Center'}      64       1
```

Reorient the rows of `T1` to be variables of a new table, `T2`. Specify that the `LastName` variable from `T1` is the source of the names of the variables of `T2`.

```
T2 = rows2vars(T1,'VariableNamesSource','LastName');
```

Display the first four variables of `T2`. The first variable of `T2` contains the names of the variables of `T1`. The remaining variables of `T2` correspond to rows of `T1`.

```
T2(:,1:4)
```

```
ans=9×4 table
       OriginalVariableNames                    Smith                    Johnson
    ____________________________    ____________________________    ________________    

    {'Gender'                  }    {'Male'                    }    {'Male'       }    
    {'Age'                     }    {[                      38]}    {[         43]}    
    {'Location'                }    {'County General Hospital'}    {'VA Hospital'}    
    {'Height'                  }    {[                      71]}    {[         69]}    
    {'Weight'                  }    {[                     176]}    {[        163]}    
    {'Smoker'                  }    {[                       1]}    {[          0]}    
    {'Systolic'                }    {[                     124]}    {[        109]}    
    {'Diastolic'               }    {[                      93]}    {[         77]}    
    {'SelfAssessedHealthStatus'}    {'Excellent'               }    {'Fair'       }    
```

Display the data in `T2.Smith`. In this example, every variable of `T2` is a 9-by-1 cell array, because the values in the rows of `T1` cannot be concatenated into arrays.

```
T2.Smith
```

```
ans=9×1 cell
    {'Male'                    }
    {[                      38]}
    {'County General Hospital'}
    {[                      71]}
    {[                     176]}
    {[                       1]}
    {[                     124]}
    {[                      93]}
    {'Excellent'               }
```

Specify Data Variables

Read data from a spreadsheet into a table. Use the first column of the spreadsheet as the row names of the table. Display the first three rows.

```
T1 = readtable('patients.xls','ReadRowNames',true);
head(T1,3)
```

```
ans=3×9 table
                  Gender      Age              Location                Height    Weight
                 __________   ___    ______________________________    ______    ______

    Smith       {'Male'  }    38     {'County General Hospital'  }      71        176
    Johnson     {'Male'  }    43     {'VA Hospital'              }      69        163
    Williams    {'Female'}    38     {'St. Mary's Medical Center'}      64        131
```

Reorient specified variables from T1 and discard the rest. To specify data variables by name, use a cell array of character vectors.

```
T2 = rows2vars(T1,'DataVariables',{'Gender','Age','Height','Weight'});
T2(:,1:4)
```

```
ans=4×4 table
    OriginalVariableNames     Smith      Johnson      Williams
    _____________________    ________    ________    __________

         {'Gender'}          {'Male'}    {'Male'}    {'Female'}
```

```
    {'Age'   }          {[  38]}    {[  43]}    {[    38]}
    {'Height'}          {[  71]}    {[  69]}    {[    64]}
    {'Weight'}          {[ 176]}    {[ 163]}    {[   131]}
```

You also can specify data variables by position in the input table. To specify positions of variables, use a numeric array.

```
T3 = rows2vars(T1,'DataVariables',[1 2 6:9]);
T3(:,1:4)
```

```
ans=6×4 table
       OriginalVariableNames            Smith        Johnson      Williams
    ____________________________    _____________    ________    __________

    {'Gender'                  }    {'Male'     }    {'Male'}    {'Female'}
    {'Age'                     }    {[       38]}    {[  43]}    {[    38]}
    {'Smoker'                  }    {[        1]}    {[   0]}    {[     0]}
    {'Systolic'                }    {[      124]}    {[ 109]}    {[   125]}
    {'Diastolic'               }    {[       93]}    {[  77]}    {[    83]}
    {'SelfAssessedHealthStatus'}    {'Excellent'}    {'Fair'}    {'Good'  }
```

Input Arguments

T1 — Input table
table | timetable

Input table, specified as a table or timetable.

`varnames` — Source of names for output table variables
character vector | string scalar | integer | logical array

Source of names for the output table variables, specified as a character vector, string scalar, integer, or logical array.

- If `varnames` is a character vector or string scalar, then it is the name of a variable in the input table `T1`.
- If `varnames` is the integer `n`, then it indicates the `n`th variable in `T1`.
- If `varnames` is a logical array, then it indicates the `n`th variable by specifying its `n`th element as `1` (`true`). All other elements must be `0` (`false`).

While `varnames` must be a name, number, or logical array that specifies a table variable, the variable itself can have any data type, with these limitations.

- The values contained in the specified table variable must have a data type that allows the values to be converted to strings. For example, `varnames` can be the name of a table variable that contains a datetime array, because datetime values can be converted to strings.
- The number of names taken from the specified table variable must match the number of rows of the input table.

`datavars` — Variables in input table

character vector | cell array of character vectors | string array | numeric array | logical array | subscripting object

Variables in the input table, specified as a character vector, cell array of character vectors, string array, numeric array, logical array, or subscripting object.

See Also

`inner2outer` | `innerjoin` | `join` | `outerjoin` | `stack` | `unstack` | `vartype`

Introduced in R2018a

rref

Reduced row echelon form (Gauss-Jordan elimination)

Syntax

```
R = rref(A)
R = rref(A,tol)
[R,p] = rref(A)
```

Description

`R = rref(A)` returns the reduced row echelon form on page 1-13604 of `A` using Gauss-Jordan elimination with partial pivoting on page 1-13604.

`R = rref(A,tol)` specifies a pivot tolerance that the algorithm uses to determine negligible columns.

`[R,p] = rref(A)` also returns the nonzero pivots `p`.

Examples

Reduced Row Echelon Form of Matrix

Create a matrix and calculate the reduced row echelon form. In this form, the matrix has leading 1s in the pivot position of each column.

```
A = magic(3)
```

```
A = 3×3

     8     1     6
     3     5     7
     4     9     2
```

```
RA = rref(A)
```

```
RA = 3×3

     1     0     0
     0     1     0
     0     0     1
```

The 3-by-3 magic square matrix is full rank, so the reduced row echelon form is an identity matrix.

Now, calculate the reduced row echelon form of the 4-by-4 magic square matrix. Specify two outputs to return the nonzero pivot columns. Since this matrix is rank deficient, the result is not an identity matrix.

```
B = magic(4)
```

```
B = 4×4

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

```
[RB,p] = rref(B)
```

```
RB = 4×4

     1     0     0     1
     0     1     0     3
     0     0     1    -3
     0     0     0     0


p = 1×3

     1     2     3
```

Row Reduction of Augmented Matrices

Use Gauss-Jordan elimination on augmented matrices to solve a linear system and calculate the matrix inverse. These techniques are mainly of academic interest, since there are more efficient and numerically stable ways to calculate these values.

Create a 3-by-3 magic square matrix. Add an additional column to the end of the matrix. This augmented matrix represents a linear system Ax = *b*, with the extra column corresponding to *b*.

```
A = magic(3);
A(:,4) = [1; 1; 1]
```

```
A = 3×4

     8     1     6     1
     3     5     7     1
     4     9     2     1
```

Calculate the reduced row echelon form of `A`. Index into `R` to extract the entries in the extra (augmented) column, which contains the solution to the linear system.

```
R = rref(A)
```

```
R = 3×4

    1.0000         0         0    0.0667
         0    1.0000         0    0.0667
         0         0    1.0000    0.0667
```

```
x = R(:,end)
```

```
x = 3×1

    0.0667
    0.0667
    0.0667
```

A more efficient way to solve this linear system is with the backslash operator, `x = A\b`.

Create a similar magic square matrix, but this time append an identity matrix of the same size to the end columns.

```
A = [magic(3) eye(3)]
```

```
A = 3×6

     8     1     6     1     0     0
     3     5     7     0     1     0
     4     9     2     0     0     1
```

Calculate the reduced row echelon form of `A`. In this form the extra columns contain the inverse matrix for the 3-by-3 magic square matrix.

```
R = rref(A)
```

```
R = 3×6

    1.0000         0         0    0.1472   -0.1444    0.0639
         0    1.0000         0   -0.0611    0.0222    0.1056
         0         0    1.0000   -0.0194    0.1889   -0.1028
```

```
inv_A = R(:,4:end)
```

```
inv_A = 3×3

    0.1472   -0.1444    0.0639
   -0.0611    0.0222    0.1056
   -0.0194    0.1889   -0.1028
```

A more efficient way to calculate the inverse matrix is with `inv(A)`.

Solve System of Equations

Consider a linear system of equations with four equations and three unknowns.

$$x_1 + x_2 + 5x_3 = 6$$
$$2x_1 + x_2 + 8x_3 = 8$$
$$x_1 + 2x_2 + 7x_3 = 10$$
$$-x_1 + x_2 - x_3 = 2\,.$$

Create an augmented matrix that represents the system of equations.

```
A = [1  1  5;
     2  1  8;
     1  2  7;
    -1  1 -1];
b = [6 8 10 2]';
M = [A b];
```

Use `rref` to express the system in reduced row echelon form.

```
R = rref(M)
```

```
R = 4×4

     1     0     3     2
     0     1     2     4
     0     0     0     0
     0     0     0     0
```

The first two rows of `R` contain equations that express x_1 and x_2 in terms of x_3. The second two rows imply that there exists at least one solution that fits the right-hand side vector (otherwise one of the equations would read 1 = 0). The third column does not contain a pivot, so x_3 is an independent variable. Therefore, there are infinitely many solutions for x_1 and x_2, and x_3 can be chosen freely.

$$x_1 = 2 - 3x_3$$
$$x_2 = 4 - 2x_3 .$$

For example, if $x_3 = 1$, then $x_1 = -1$ and $x_2 = 2$.

From a numerical standpoint, a more efficient way to solve this system of equations is with `x0 = A\b`, which (for a rectangular matrix `A`) calculates the least-squares solution. In that case, you can check the accuracy of the solution with `norm(A*x0-b)/norm(b)` and the uniqueness of the solution by checking if `rank(A)` is equal to the number of unknowns. If more than one solution exists, then they all have the form of $x = x_0 + nt$, where n is the null space `null(A)` and t can be chosen freely.

Input Arguments

A — Input matrix
matrix

Input matrix.

Data Types: `single` | `double`
Complex Number Support: Yes

tol — Pivot tolerance
`max(size(A))*eps*norm(A,inf)` (default) | scalar

Pivot tolerance, specified as a scalar. If the largest element (by absolute value) in a pivot column is below the tolerance, then the column is zeroed out. This prevents division and multiplication with nonzero pivot elements smaller than the tolerance.

Data Types: `single` | `double`

Output Arguments

R — Reduced row echelon form of A
matrix

Reduced row echelon form on page 1-13604 of `A`, returned as a matrix.

p — Nonzero pivot columns
vector

Nonzero pivot columns, returned as a vector. Each element in `p` is a column index of `A`. You can use `p` to estimate several quantities:

- `length(p)` is an estimate of the rank of `A`.
- `x(p)` contains the pivot variables in a linear system `Ax = b`.
- `A(:,p)` is a basis for the range of `A`.
- `R(1:r,p)` is the `r`-by-`r` identity matrix, where `r = length(p)`.

Limitations

- `rank`, `orth`, and `null` are typically faster and more accurate for computing the rank and basis vectors of a matrix.
- `mldivide` is recommended to solve linear systems.

More About

Partial Pivoting

Partial pivoting is the practice of selecting the column element with largest absolute value in the pivot column, and then interchanging the rows of the matrix so that this element is in the pivot position (the leftmost nonzero element in the row).

For example, in the matrix below the algorithm starts by identifying the largest value in the first column (the value in the (2,1) position equal to `1.1`), and then interchanges the complete first and second rows so that this value appears in the (1,1) position.

$$\begin{pmatrix} 0.5 & 1.4 & 2 \\ 1.1 & 0.3 & 0.1 \\ 0.2 & 5 & 0.5 \end{pmatrix}$$

The use of partial pivoting in Gaussian elimination reduces (but does not eliminate) roundoff errors in the calculation.

Reduced Row Echelon Form

A matrix is in *row echelon form* when these conditions are met:

- All nonzero rows are above rows of all zeros.
- The leading coefficient of each row is strictly to the right of the one in the row above it.

An example of a matrix in row echelon form is

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 1 \\ 0 & 0 & 2 \end{pmatrix}.$$

An additional requirement for *reduced row echelon form* is:

- Every leading coefficient must be 1, and must be the only nonzero in its column.

While the identity matrix is most commonly associated with reduced row echelon form, other forms are possible. Another example of a matrix in reduced row echelon form is

$$A = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

Algorithms

`rref` implements Gauss-Jordan elimination with partial pivoting. A default tolerance of `max(size(A))*eps*norm(A,inf)` tests for negligible column elements that are zeroed-out to reduce roundoff error.

See Also

`inv` | `lu` | `mldivide` | `rank`

Introduced before R2006a

rsf2csf

Convert real Schur form to complex Schur form

Syntax

```
[U,T] = rsf2csf(U,T)
```

Description

The *complex Schur form* of a matrix is upper triangular with the eigenvalues of the matrix on the diagonal. The *real Schur form* has the real eigenvalues on the diagonal and the complex eigenvalues in 2-by-2 blocks on the diagonal.

`[U,T] = rsf2csf(U,T)` converts the real Schur form to the complex form.

Arguments `U` and `T` represent the unitary and Schur forms of a matrix `A`, respectively, that satisfy the relationships: `A = U*T*U'` and `U'*U = eye(size(A))`. See `schur` for details.

Examples

Given matrix `A`,

```
 1     1     1     3
 1     2     1     1
 1     1     3     1
-2     1     1     4
```

with the eigenvalues

```
4.8121    1.9202 + 1.4742i    1.9202 + 1.4742i    1.3474
```

Generating the Schur form of `A` and converting to the complex Schur form

```
[u,t] = schur(A);
[U,T] = rsf2csf(u,t)
```

yields a triangular matrix T whose diagonal (underlined here for readability) consists of the eigenvalues of A.

```
U =

-0.4916       -0.2756 - 0.4411i    0.2133 + 0.5699i   -0.3428
-0.4980       -0.1012 + 0.2163i   -0.1046 + 0.2093i    0.8001
-0.6751        0.1842 + 0.3860i   -0.1867 - 0.3808i   -0.4260
-0.2337        0.2635 - 0.6481i    0.3134 - 0.5448i    0.2466

T =

4.8121        -0.9697 + 1.0778i   -0.5212 + 2.0051i   -1.0067
     0         1.9202 + 1.4742i    2.3355              0.1117 + 1.6547i
     0              0              1.9202 - 1.4742i    0.8002 + 0.2310i
     0              0                   0              1.3474
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

See Also

`schur`

Introduced before R2006a

rtickangle

Rotate *r*-axis tick labels

Syntax

```
rtickangle(angle)
rtickangle(pax,angle)

ang = rtickangle
ang = rtickangle(pax)
```

Description

`rtickangle(angle)` rotates the *r*-axis tick labels for the current axes to the specified angle in degrees, where `0` is horizontal. Specify a positive value for counterclockwise rotation or a negative value for clockwise rotation.

`rtickangle(pax,angle)` rotates the tick labels for the axes specified by `pax` instead of the current axes.

`ang = rtickangle` returns the rotation angle for the *r*-axis tick labels of the current axes as a scalar value in degrees. Positive values indicate counterclockwise rotation. Negative values indicate clockwise rotation.

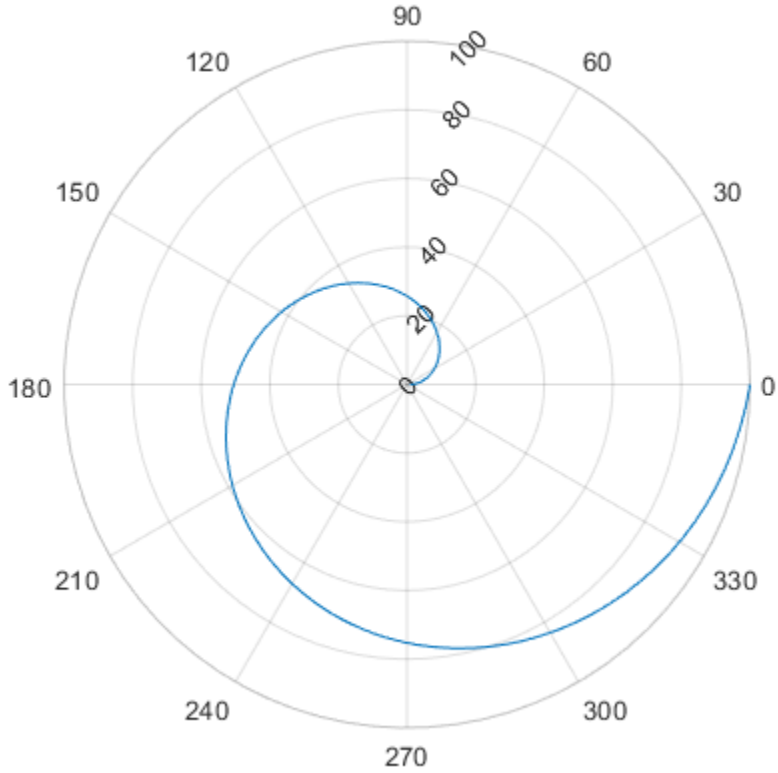
`ang = rtickangle(pax)` uses the polar axes specified by `pax` instead of the current axes.

Examples

Rotate r-Axis Tick Labels

Create a polar plot and rotate the *r*-axis tick labels 45 degrees counterclockwise.

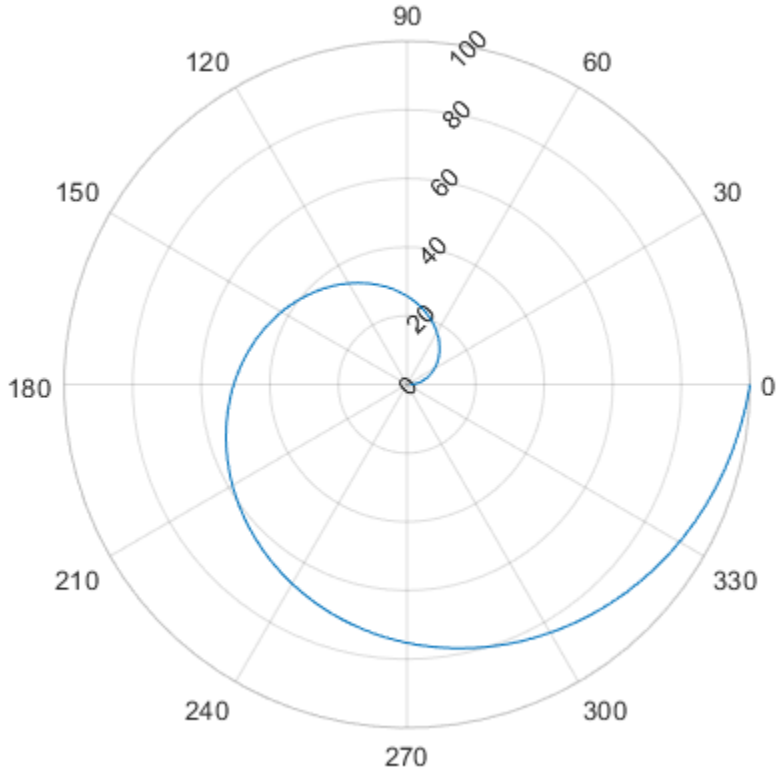
```
polarplot(1:100)
rtickangle(45)
```



Rotate r-Axis Tick Labels for Specific Polar Axes

Create a polar plot and assign the polar axes object to the variable `pax`. Then, rotate the *r*-axis tick labels for the polar axes. Ensure that `rtickangle` affects `pax` by specifying the polar axes object as the first input argument.

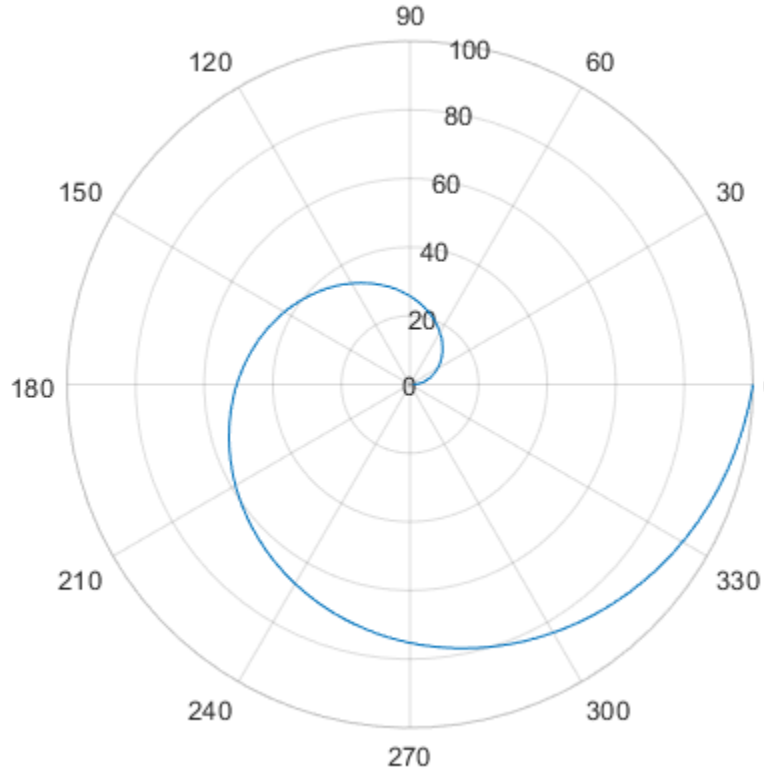
```
polarplot(1:100)
pax = gca;
rtickangle(pax,45)
```



Query Current Angle for r-Axis Tick Labels

Create a polar plot. Then, query the rotation angle for the *r*-axis tick labels. By default, the labels are not rotated.

```
polarplot(1:100)
```




```
ang = rtickangle
```

```
ang = 0
```

Input Arguments

angle — Rotation of tick labels
`0` (default) | scalar value in degrees

Rotation of tick labels, specified as a scalar value in degrees, where 0 is horizontal.

Example: `rtickangle(90)`

pax — One or more polar axes
current axes (default) | single object | vector of objects

One or more polar axes, specified as a single object or a vector of objects. If you do not specify the polar axes, then `rtickangle` uses the current axes.

Algorithms

The `rtickangle` function sets and queries the `RTickLabelRotation` property of the axes object.

See Also

Functions
`rlim` | `rtickformat` | `rticklabels` | `rticks` | `thetalim`

Properties
PolarAxes

Introduced in R2016b

rtickformat

Specify *r*-axis tick label format

Syntax

```
rtickformat(fmt)
rtickformat(pax, ___ )
```

```
rfmt = rtickformat
rfmt = rtickformat(pax)
```

Description

`rtickformat(fmt)` sets the format for the *r*-axis tick labels. For example, specify `fmt` as `'usd'` to display the labels in U.S. dollars.

`rtickformat(pax, ___ )` uses the axes specified by `pax` instead of the current axes. Specify `ax` as the first input argument.

`rfmt = rtickformat` returns the format style used for *r*-axis tick labels of the current axes. Depending on the type of labels along the *r*-axis, `rfmt` is a character vector or string containing a numeric format.

`rfmt = rtickformat(pax)` returns the format style used for the axes specified by `pax` instead of the current axes.

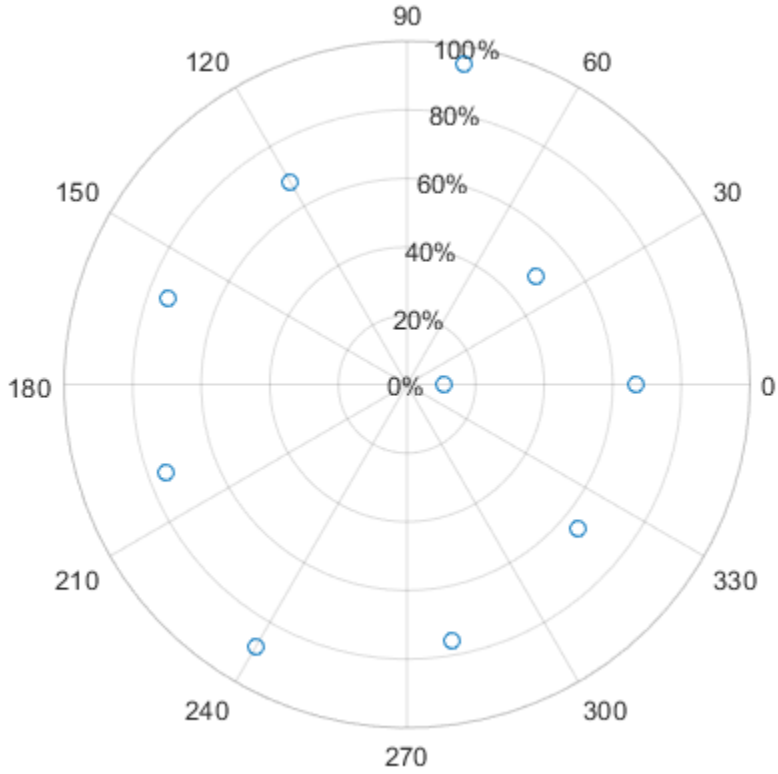
Examples

Display *r*-Axis Tick Labels as Percentages

Create a polar plot. Display the tick labels along the *r*-axis as percentages.

```
th = linspace(0,2*pi,10);
r = [11 49 95 68 74 75 88 76 65 67];
```

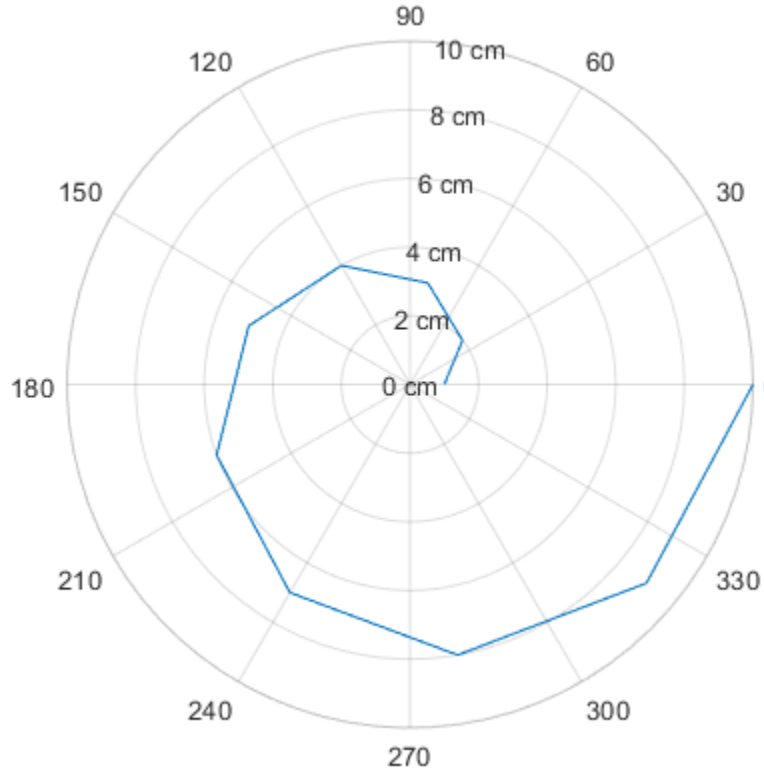
```
polarplot(th,r,'o')
rtickformat('percentage')
```



Display Text After Each *r*-Axis Tick Label

Display the tick labels along the *r*-axis with the text "cm" after each value.

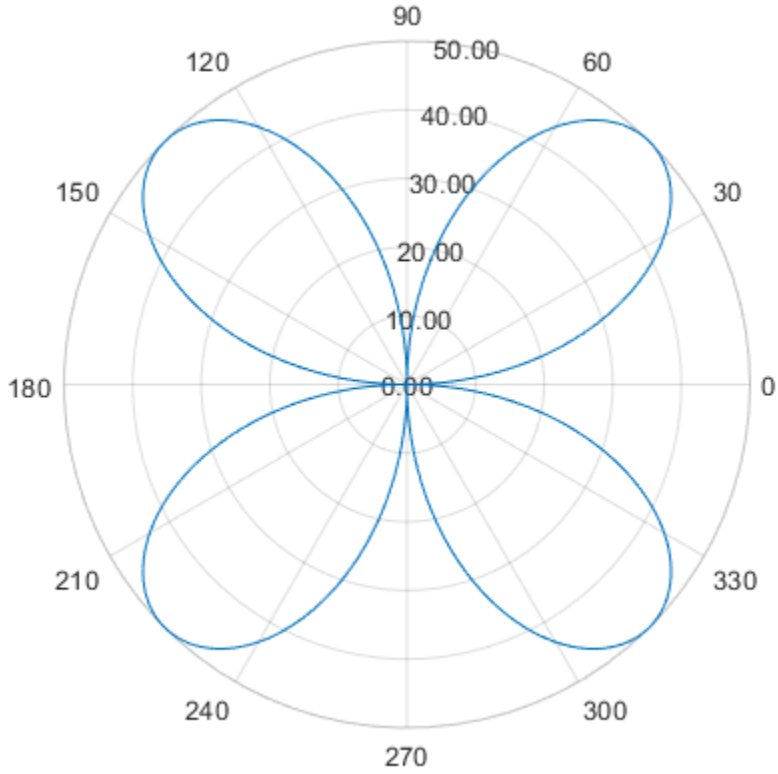
```
polarplot(1:10)
rtickformat('%g cm')
```



Control Number of Decimals for *r*-Axis Tick Labels

Display the *r*-axis tick labels with two decimal places. Control the decimal places by passing `rtickformat` a character vector of a numeric format that uses fixed-point notation for the conversion character and a precision value of 2.

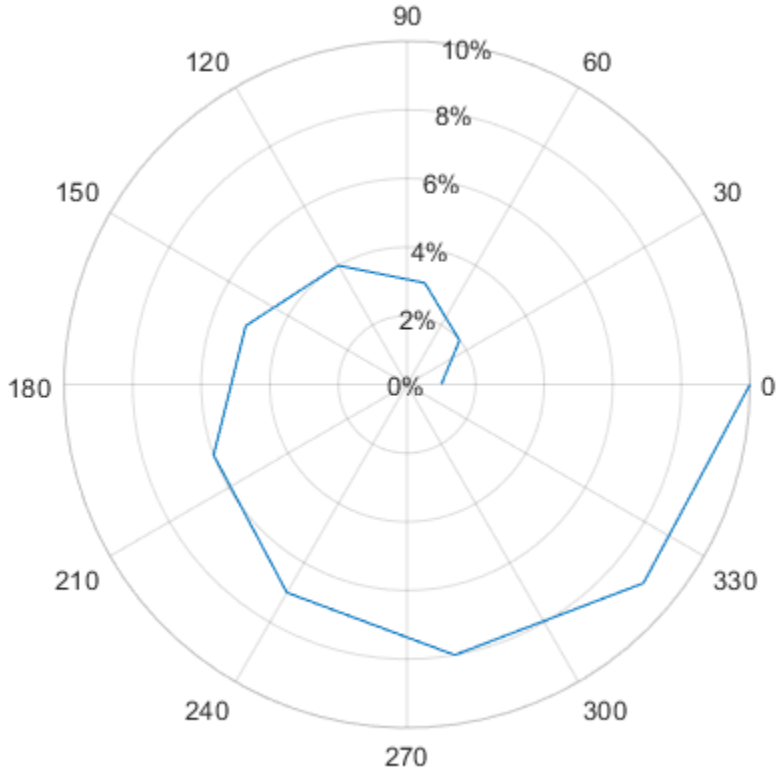
```
theta = 0:0.01:2*pi;
rho = 50*sin(2*theta);
polarplot(theta,rho)
rtickformat('%.2f')
```



Specify *r*-Axis Tick Label Format for Specific Polar Axes

Create a polar plot and assign the polar axes object to the variable `pax`. Ensure that `rtickformat` affects the polar axes you just created by passing `pax` as the first input argument to the function.

```
polarplot(1:10)
pax = gca;
rtickformat(pax,'percentage')
```



Input Arguments

`fmt` — Format for numeric tick labels
`'%g'` (default) | character vector | string

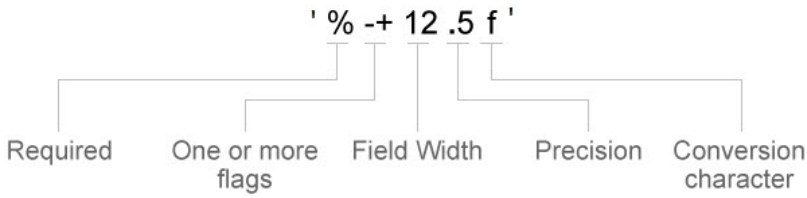
Format for numeric tick labels, specified as a character vector or string. You can specify one of the formats listed in this table. Alternatively, you can specify a custom format.

Predefined Format	Description
`'usd'`	U.S. dollars. This option is equivalent using `'$%,.2f'`. If the labels use scientific notation, this option sets the exponent value to 0.
`'eur'`	Euro. This option is equivalent to using `'\x20AC%,.2f'` with an exponent value of 0.
`'gbp'`	British pound. This option is equivalent to using `'\x00A3%,.2f'` with an exponent value of 0.
`'jpy'`	Japanese yen. This option is equivalent to using `'\x00A5%,d'` with an exponent value of 0.
`'degrees'`	Display degree symbol after values. This option is equivalent to using `'%g\x00B0'` with the default exponent value.
`'percentage'`	Display percent sign after values. This option is equivalent to using `'%g%%'` with the default exponent value.
`'auto'`	Default format of `'%g'` with the default exponent value.

Example: `rtickformat('usd')`

Custom Numeric Format

You can specify a custom numeric format by creating a character vector or string containing identifiers.



Identifiers are optional, except the percent sign and conversion character. Construct the format in this order:

- One or more flags — Options such as adding a plus sign before positive values. For a full list of options, see the table of Optional Flags.
- Field width — Minimum number of characters to print in the tick label. Specify the field width as an integer value. If the number of significant digits in the tick value is smaller than the field width, then the label is padded with spaces.
- Precision — Number of digits to the right of the decimal point or the number of significant digits, depending on the conversion character. Specify the precision as an integer value.
- Conversion character — Value type. For a full list of options, see the table of Conversion Characters. If you specify a conversion that does not fit the data, then MATLAB overrides the specified conversion, and uses `%e`.

Also, you can specify literal text at the beginning or end of the format. To print a single quotation mark, use `''`. To print a percent character, use `%%`.

Example: `rtickformat('%.2f')` displays the values using fixed-point notation with two decimal places.

Example: `rtickformat('$%.2f')` displays a dollar sign before each value.

Example: `rtickformat('%.2f million')` displays `million` after each value.

Optional Flags

Identifier	Description	Example of Numeric Format
`,`	Display commas every three digits, such as `'1,000'`.	`'%,4.4g'`
`+`	Print the sign character (`+`) for positive values, such as `'+100'`.	`'%+4.4g'`
`0`	Pad the field width with leading zeros instead of spaces, such as `'0100'`.	`'%04.4g'`
`-`	Left-justify, which pads the end of the value with spaces instead of the beginning. For example, if the field width is `4`, then this flag formats the label as `'100 '` instead of `' 100'`.	`'%-4.4g'`
`#`	For the `%f`, `%e`, and `%g` conversion characters, print the decimal point even when the precision is 0, such as `'100.'`. For `%g`, do not remove trailing zeros.	`'%#4.4g'`

Conversion Characters

Identifier	Description	Example
`d` or `i`	Signed integer with base 10. The precision value indicates the number of significant digits.	`'%.4d'` displays π as `0003`.
`f`	Fixed-point notation. The precision value indicates the number of decimal places.	`'%.4f'` displays π as `3.1416`.
`e`	Exponential notation. The precision value indicates the number of decimal places.	`'%.4e'` displays π as 3.1416×10^0.
`g`	The more compact version of `e` or `f`, with no trailing zeros. The precision value indicates the maximum number of decimal places.	`'%.4g'` displays π as `3.1416`.

pax — One or more polar axes
current axes (default) | single object | vector of objects

One or more polar axes, specified as a single object or a vector of objects. If you do not specify a polar axes object, then `rtickformat` uses the current axes.

Algorithms

The `rtickformat` function sets and queries the `TickLabelFormat` property of the ruler object associated with the *r*-axis.

See Also

`rlim` | `rtickangle` | `rticklabels` | `rticks` | `thetatickformat`

Introduced in R2016b

rticklabels

Set or query *r*-axis tick labels

Syntax

```
rticklabels(labels)
rl = rticklabels
```

```
rticklabels('auto')
rticklabels('manual')
m = rticklabels('mode')
```

```
___ = rticklabels(pax,___)
```

Description

`rticklabels(labels)` sets the *r*-axis tick labels for the current axes. Specify `labels` as a string array or a cell array of character vectors; for example, `{'January','February','March'}`. If you specify the labels, then the *r*-axis tick values and tick labels no longer update automatically based on changes to the axes.

`rl = rticklabels` returns the *r*-axis tick labels for the current axes.

`rticklabels('auto')` sets an automatic mode, enabling the axes to determine the *r*-axis tick labels. Use this option if you set the labels and then want to set them back to the default values.

`rticklabels('manual')` sets a manual mode, freezing the *r*-axis tick labels at the current values.

`m = rticklabels('mode')` returns the current value of the *r*-axis tick labels mode, which is either `'auto'` or `'manual'`. By default, the mode is automatic unless you specify the tick labels or set the mode to manual.

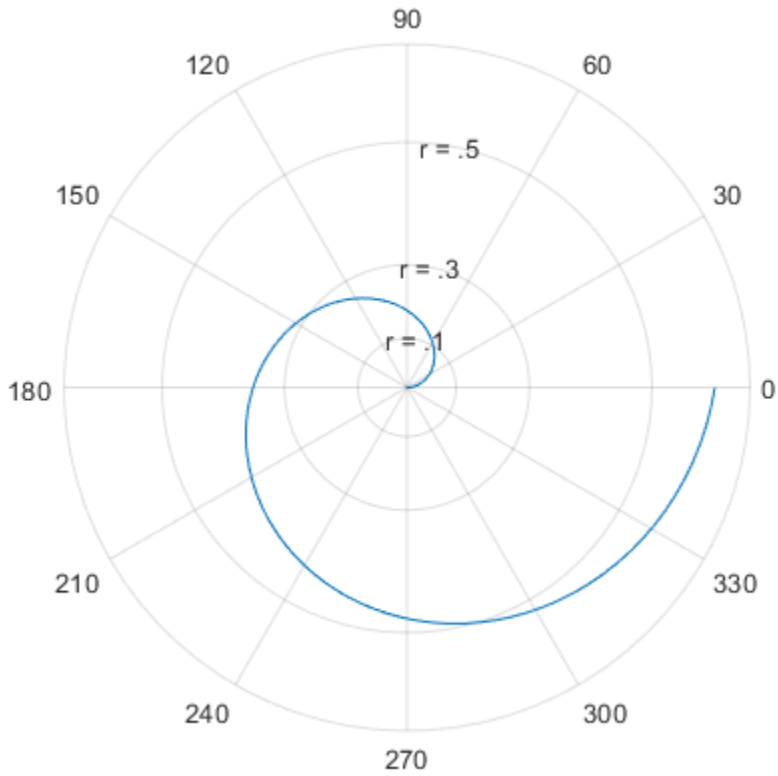
`___ = rticklabels(pax,___)` uses the axes specified by `pax` instead of the current axes. Specify `ax` as the first input argument for any of the previous syntaxes.

Examples

Specify *r*-Axis Tick Values and Labels

Create a polar plot. Display tick marks and grid lines along the *r*-axis at the values 0.1, 0.3, and 0.5. Then, specify a label for each tick mark.

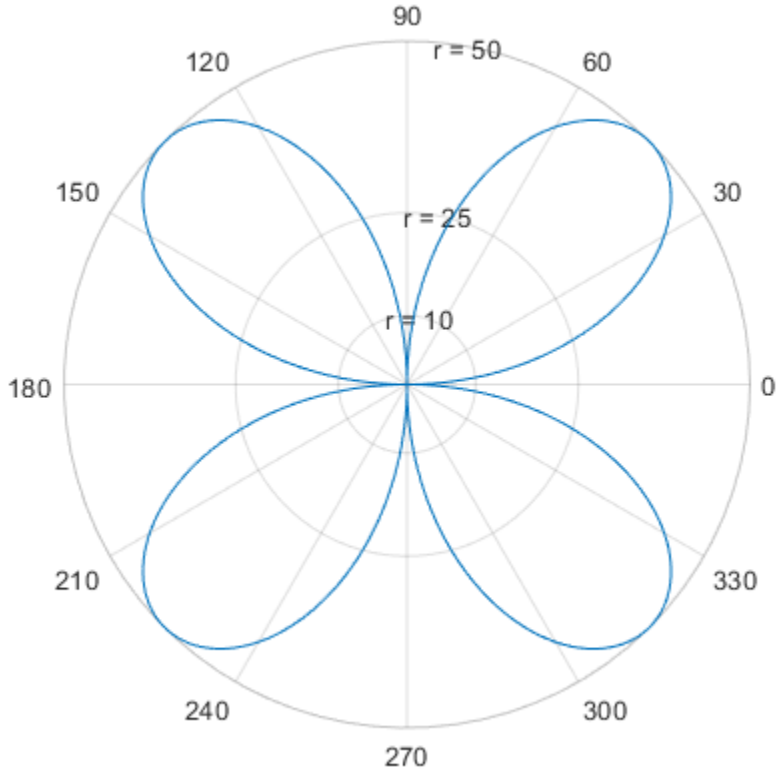
```
theta = linspace(0,2*pi);
rho = theta/10;
polarplot(theta,rho)
rticks([0.1 0.25 0.5])
rticklabels({'r = .1','r = .3','r = .5'})
```



Specify r-Axis Tick Labels for Specific Polar Axes

Create a polar plot and assign the polar axes object to the variable `pax`. Ensure that `rticks` and `rticklabels` affect the polar axes you just created by specifying `pax` as the first input argument to the functions.

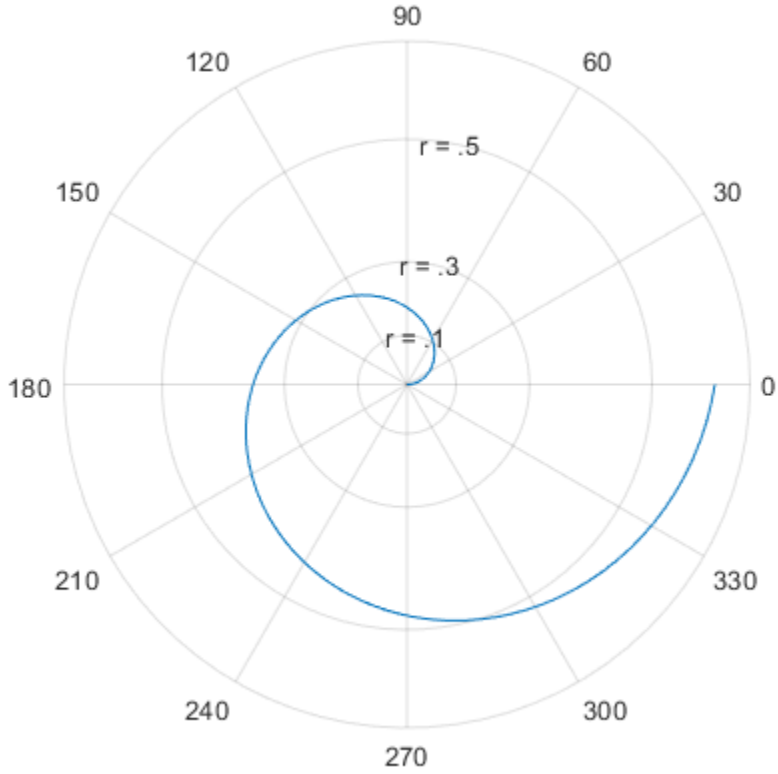
```
theta = 0:0.01:2*pi;
rho = 50*sin(2*theta);
polarplot(theta,rho)
pax = gca;

rticks(pax,[10 25 50])
rticklabels(pax,{'r = 10','r = 25','r = 50'})
```



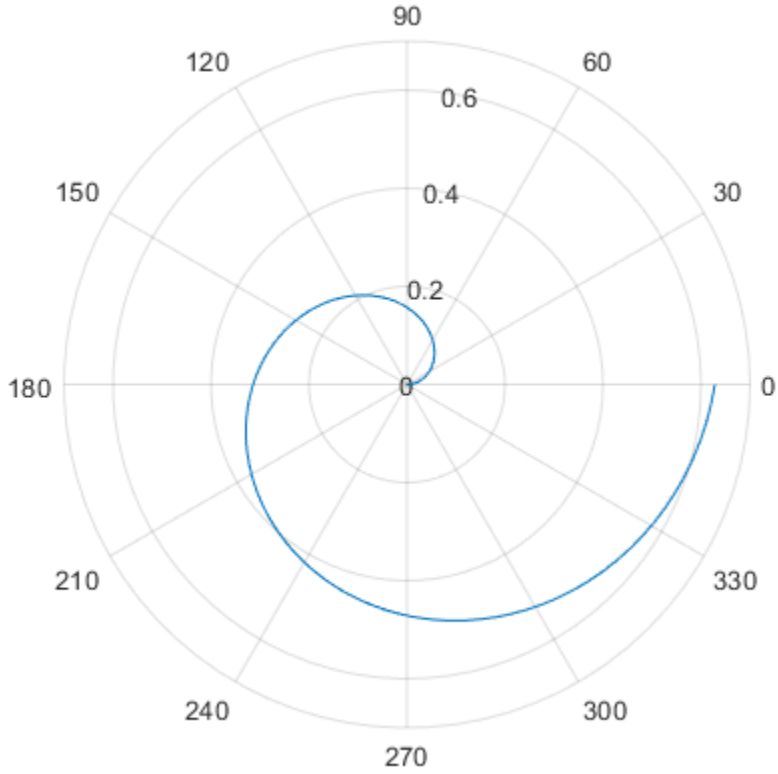
Set r-Axis Tick Labels Back to Default Labels

Create a polar plot and specify the *r*-axis tick values and corresponding labels. Then, set the *r*-axis tick values and labels back to the default values.

```
theta = linspace(0,2*pi);
rho = theta/10;
polarplot(theta,rho)
rticks([0.1 0.25 0.5])
rticklabels({'r = .1','r = .3','r = .5'})
```



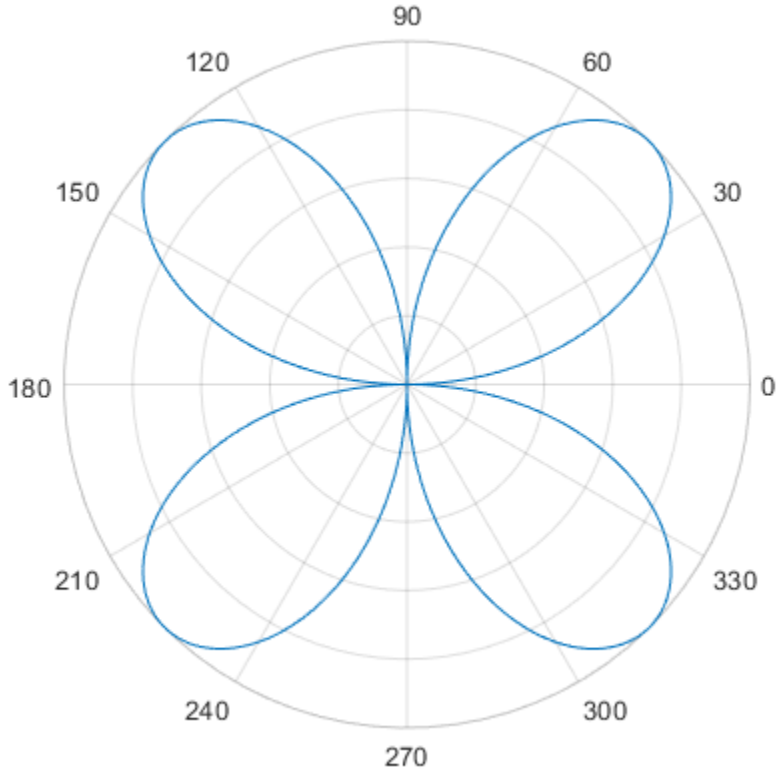

```
rticks('auto')
rticklabels('auto')
```



Remove r-Axis Tick Labels

Remove the tick labels along the *r*-axis by specifying the tick labels as an empty array.

```
theta = 0:0.01:2*pi;
rho = 50*sin(2*theta);
polarplot(theta,rho)
rticklabels({})
```



Input Arguments

`labels` — Tick labels

cell array of character vectors | string array | categorical array

Tick labels on page 1-13630, specified as a cell array of character vectors, string array, or categorical array. If you do not want tick labels to show, then specify an empty cell array `{}`. Tick labels support TeX and LaTeX markup. See the `TickLabelInterpreter` property of the axes object for more information.

Example: `rticklabels({'0','\pi','2\pi'})`

Example: `rticklabels({'January','Febrary','March'})`

Example: `rticklabels({})`

Note

- If you specify the tick labels as a categorical array, MATLAB uses the values in the array, not the categories.
- To specify the tick values, use the `rticks` function.

pax — One or more polar axes
current axes (default) | single object | vector of objects

One or more polar axes, specified as a single object or a vector of objects. If you do not specify the polar axes, then `rticklabels` uses the current axes.

Output Arguments

`rl` — Current tick labels
cell array of character vectors | character array

Current tick labels, returned as a cell array of character vectors or a character array.

`m` — Current tick labels mode
`'auto'` | `'manual'`

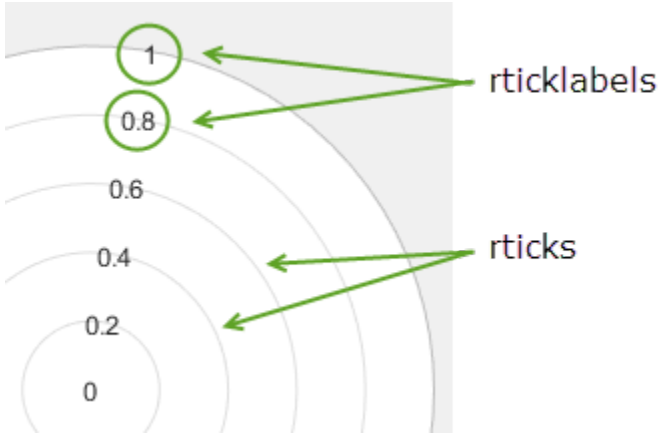
Current tick labels mode, returned as one of these values:

- `'auto'` — Axes automatically determines the *r*-axis tick labels.
- `'manual'` — Axes uses manually specified *r*-axis tick labels.

More About

Tick Labels

The tick labels are the labels that you see next to each tick mark. The tick values are the locations along the *r*-axis where the tick marks appear. Set the tick values using the `rticks` function. Set the corresponding tick labels using the `rticklabels` function.



Algorithms

The `rticklabels` function sets and queries several polar axes properties related to the *r*-axis tick labels.

- `RTickLabel` — Property that stores the text for the *r*-axis tick labels.
- `RTickLabelMode` — Property that stores the *r*-axis tick label mode. When you set the *r*-axis tick labels using `rticklabels`, this property changes to `'manual'`.
- `RTickMode` — Property that stores the *r*-axis tick value mode. When you set the *r*-axis tick labels using `rticklabels`, this property changes to `'manual'`.

See Also

Functions
`rlim` | `rtickangle` | `rtickformat` | `rticks` | `thetaticklabels`

Properties
PolarAxes

Introduced in R2016b

rticks

Set or query *r*-axis tick values

Syntax

```
rticks(ticks)
rt = rticks

rticks('auto')
rticks('manual')
m = rticks('mode')

___ = rticks(pax, ___ )
```

Description

`rticks(ticks)` sets the *r*-axis tick values on page 1-13640, which are the locations along the *r*-axis where the tick marks and grid lines appear. Specify `ticks` as a vector of increasing values; for example, `[0 2 4 6]`. This command affects the current axes.

`rt = rticks` returns the current *r*-axis tick values as a vector.

`rticks('auto')` sets an automatic mode, enabling the axes to determine the *r*-axis tick values. Use this option if you change the tick values and then want to set them back to the default values.

`rticks('manual')` sets a manual mode, freezing the *r*-axis tick values at the current values. Use this option if you want to retain the current tick values when resizing the axes or adding new data to the axes.

`m = rticks('mode')` returns the current *r*-axis tick labels mode, which is either `'auto'` or `'manual'`. By default, the mode is automatic unless you specify tick values or change the mode to manual.

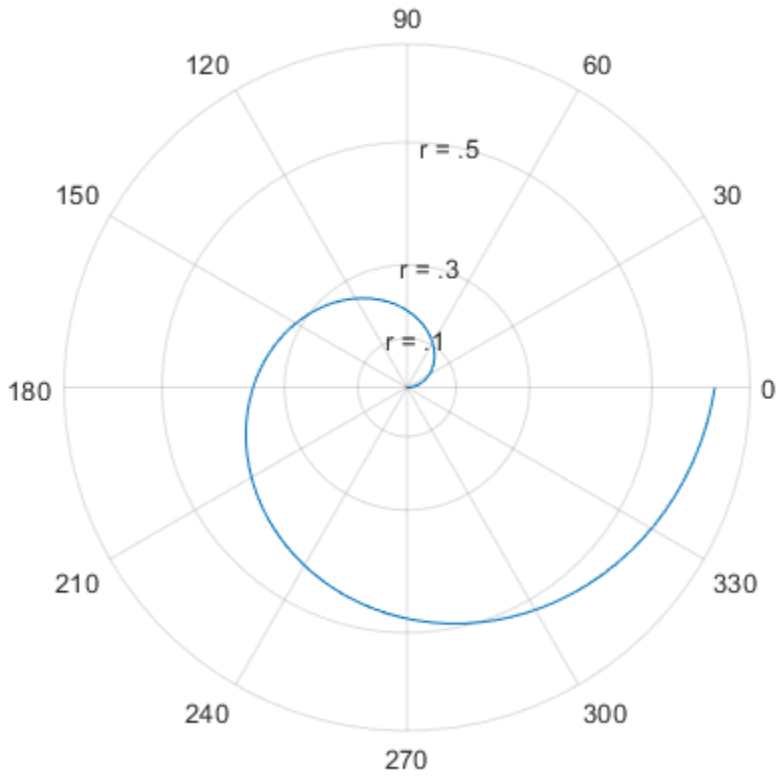
`___ = rticks(pax, ___ )` uses the axes object `pax` instead of the current axes. Specify `pax` as the first input argument for any of the previous syntaxes.

Examples

Specify *r*-Axis Tick Values and Labels

Create a polar plot. Display tick marks and grid lines along the *r*-axis at the values 0.1, 0.3, and 0.5. Then, specify a label for each tick mark.

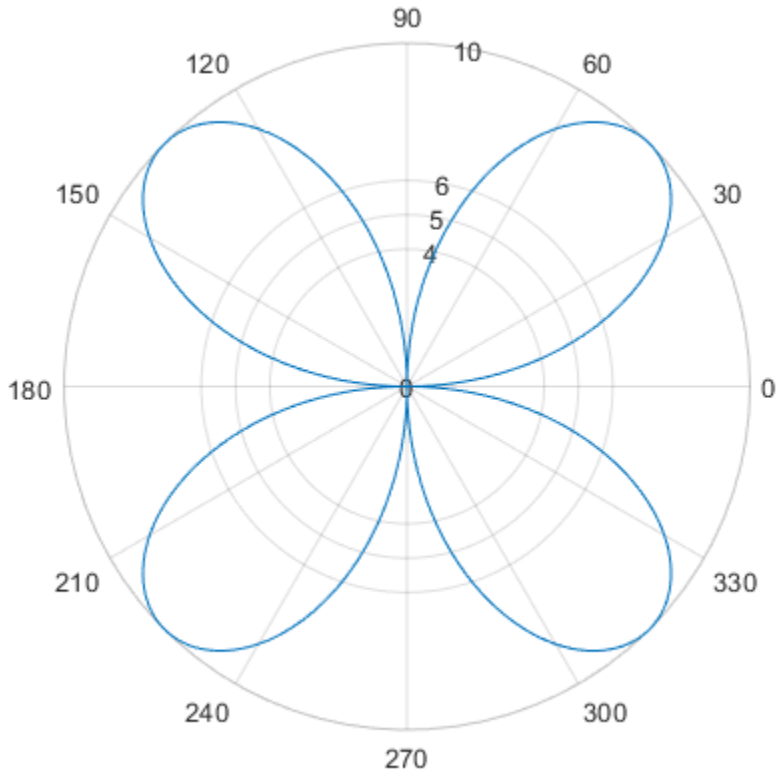
```
theta = linspace(0,2*pi);
rho = theta/10;
polarplot(theta,rho)
rticks([0.1 0.25 0.5])
rticklabels({'r = .1','r = .3','r = .5'})
```



Specify Nonuniform *r*-Axis Tick Values

Display tick marks and grid lines along the *r*-axis at nonuniform values between 0 and 10. MATLAB® labels the tick marks with the numeric values.

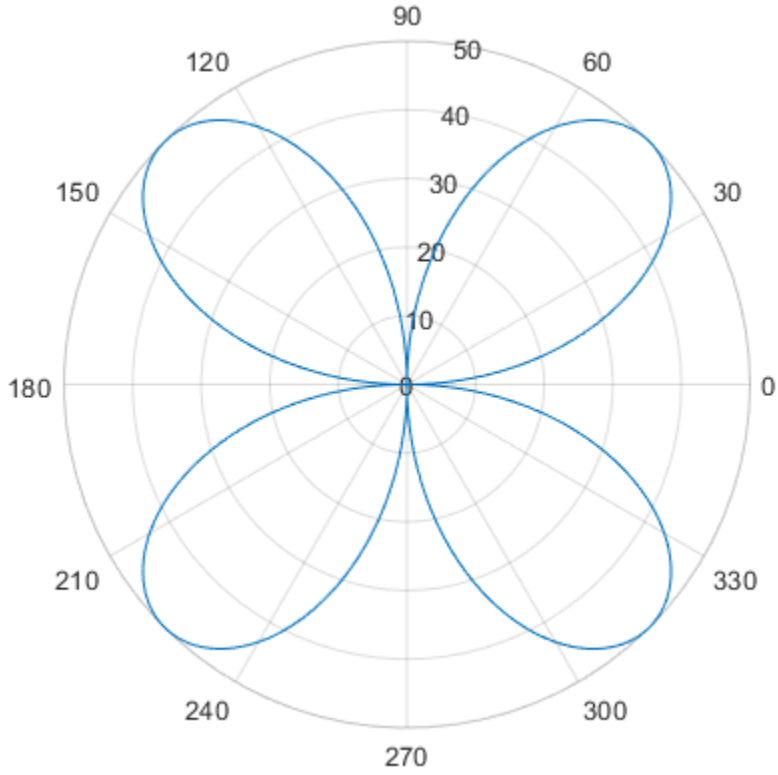
```
theta = 0:0.01:2*pi;
rho = 10*sin(2*theta);
polarplot(theta,rho)
rticks([0 4 5 6 10])
```



Increment *r*-Axis Tick Values by 10

Display tick marks and grid lines along the *r*-axis at increments of 10, starting from 0 and ending at 50.

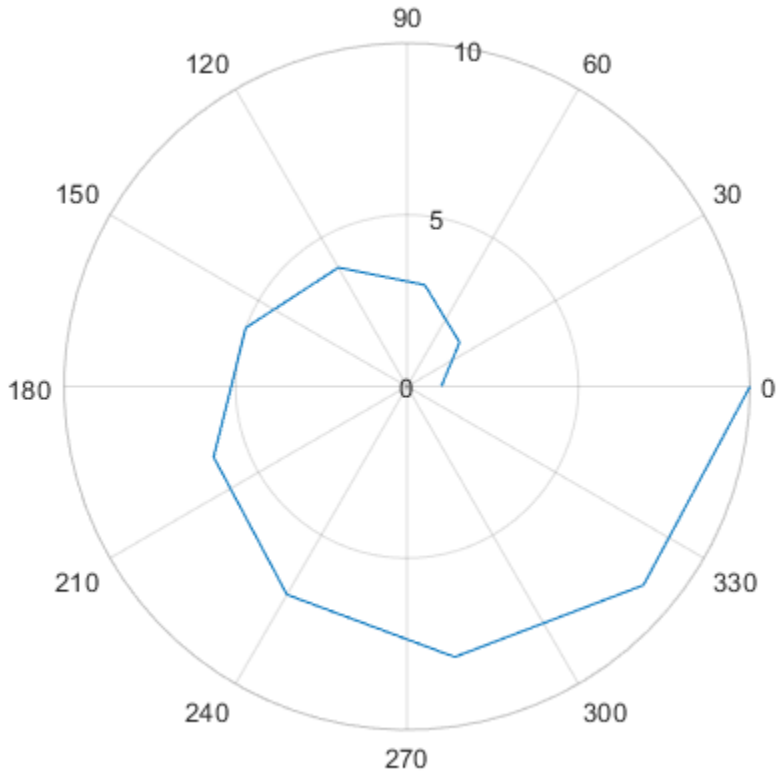
```
theta = 0:0.01:2*pi;
rho = 50*sin(2*theta);
polarplot(theta,rho)
rticks(0:10:50)
```



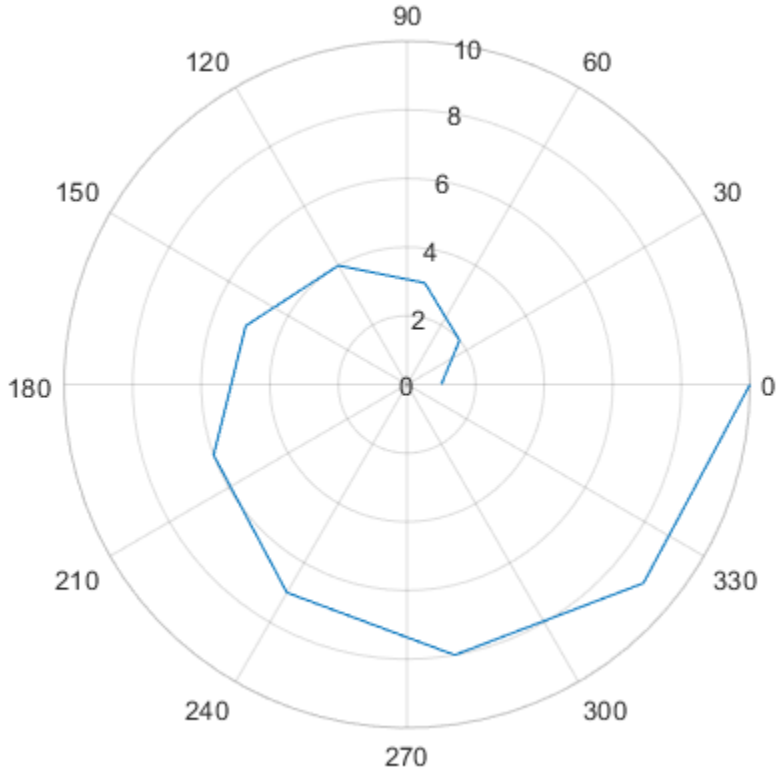
Set *r*-Axis Tick Values Back to Default Values

Create a polar plot and specify the *r*-axis tick values. Then, set the *r*-axis tick values back to the default values.

```
polarplot(1:10)
rticks([0 5 10])
```



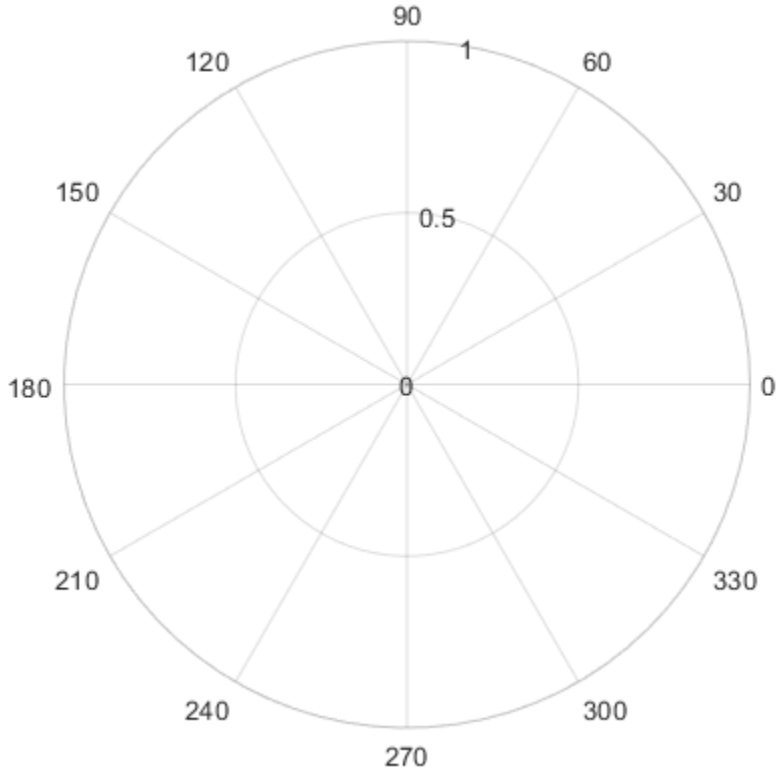

```
rticks('auto')
```



Specify *r*-Axis Tick Values for Specific Polar Axes

Create polar axes and return the polar axes object `pax`. Ensure that `rticks` affects the polar axes you just created by passing `pax` as the first input argument to the function.

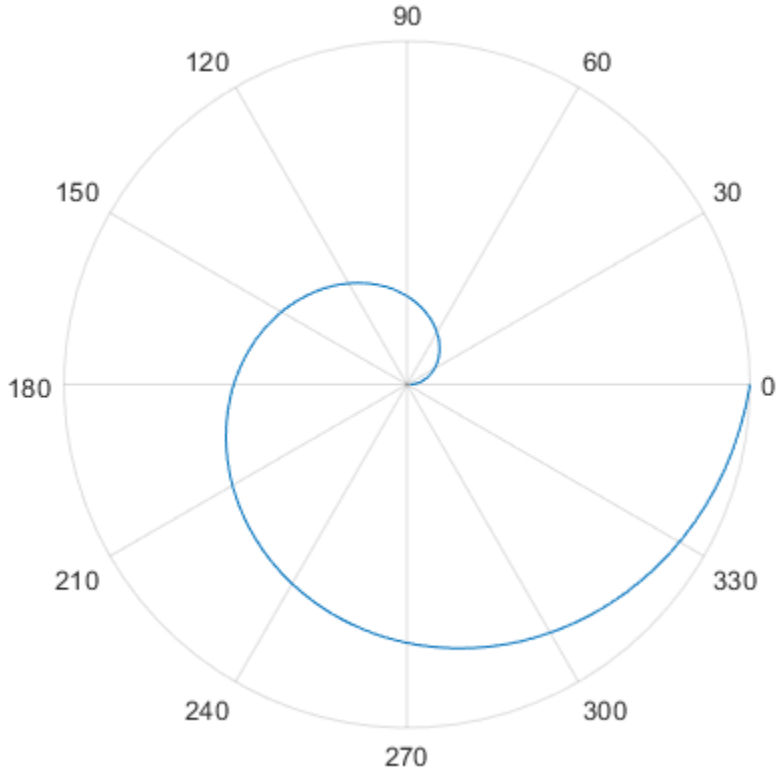
```
pax = polaraxes;
rticks(pax,[0 .5 1])
```



Remove *r*-Axis Tick Marks and Grid Lines

Remove the tick marks and grid lines along the *r*-axis by specifying the tick values as an empty array.

```
polarplot(1:100)
rticks([])
```



Input Arguments

`ticks` — Tick values
vector of increasing values

Tick values on page 1-13640, specified as a vector of increasing values. If you do not want tick marks along the *r*-axis, specify an empty vector `[]`.

Example: `rticks([0 1 2 3 4])`

Example: `rticks(0:10:100)`

Example: `rticks([])`

Note To specify the tick labels, use the `rticklabels` function.

pax — One or more polar axes
current axes (default) | single object | vector of objects

One or more polar axes, specified as a single object or a vector of objects. If you do not specify the polar axes, then `rticks` uses the current axes.

Output Arguments

`rt` — Current tick values
vector

Current tick values, returned as a vector.

`m` — Current mode
`'auto'` | `'manual'`

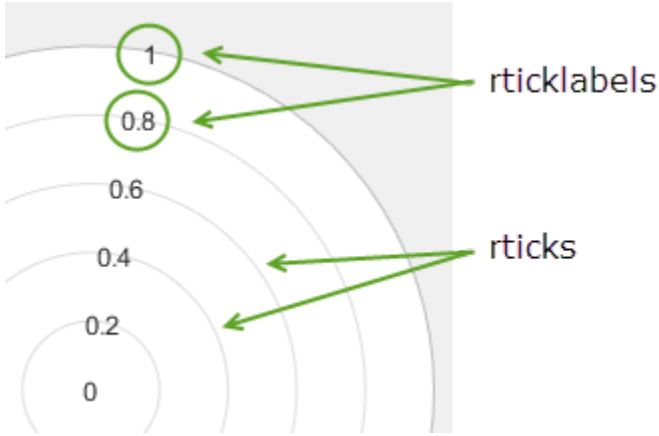
Current mode, returned as one of these values:

- `'auto'` — Axes automatically determines the *r*-axis tick values.
- `'manual'` — Axes uses manually specified *r*-axis tick values.

More About

Tick Values

The tick values are the locations along the *r*-axis where the tick marks appear. The tick labels are the labels that you see next to each tick mark. Set the tick values using the `rticks` function. Set the corresponding tick labels using the `rticklabels` function.



Algorithms

The `rticks` function sets and queries several axes properties related to the *r*-axis tick values.

- `RTick` — Property that stores the *r*-axis tick values.
- `RTickMode` — Property that stores the *r*-axis tick value mode. When you set the *r*-axis tick values, this property changes to `'manual'`.

See Also

Functions
`rlim` | `rtickangle` | `rtickformat` | `rticklabels` | `thetaticks`

Properties
PolarAxes

Introduced in R2016b

ruler2num

Convert data from specific ruler to numeric data

Syntax

```
num = ruler2num(data,ruler)
```

Description

`num = ruler2num(data,ruler)` converts the values in `data` to numeric values. The `ruler` input indicates the ruler associated with `data`, which determines how to map the values to the corresponding numeric values. The output `num` is the same size as the input `data`.

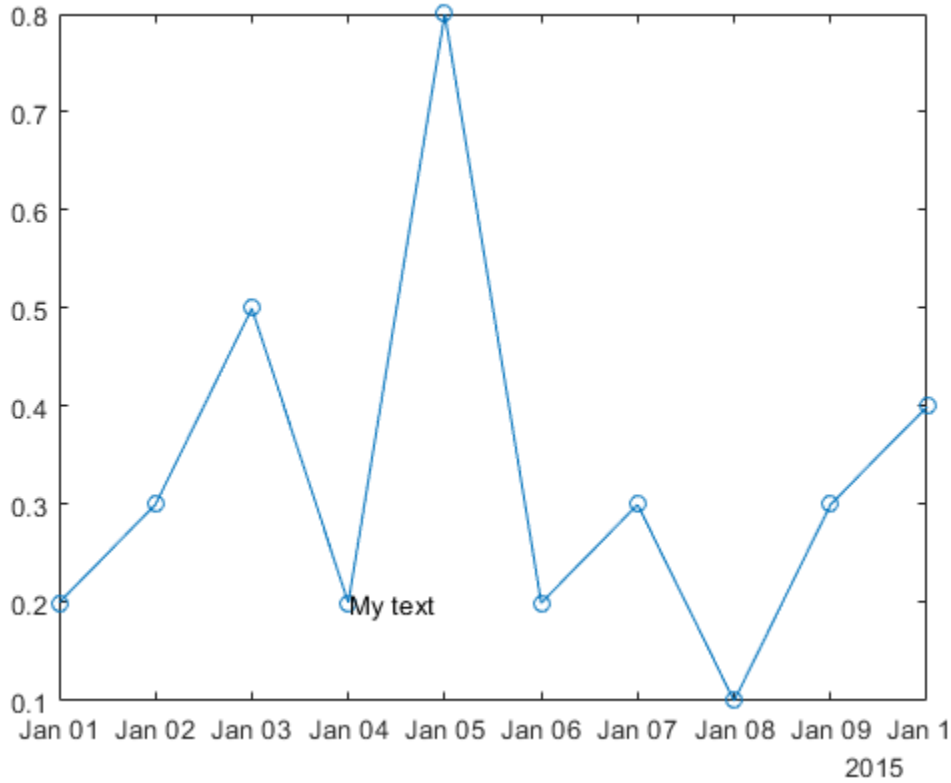
Examples

Convert Date Value to Equivalent Numeric Value

Convert a date value to the equivalent numeric value in order to set the `Position` property of a text object.

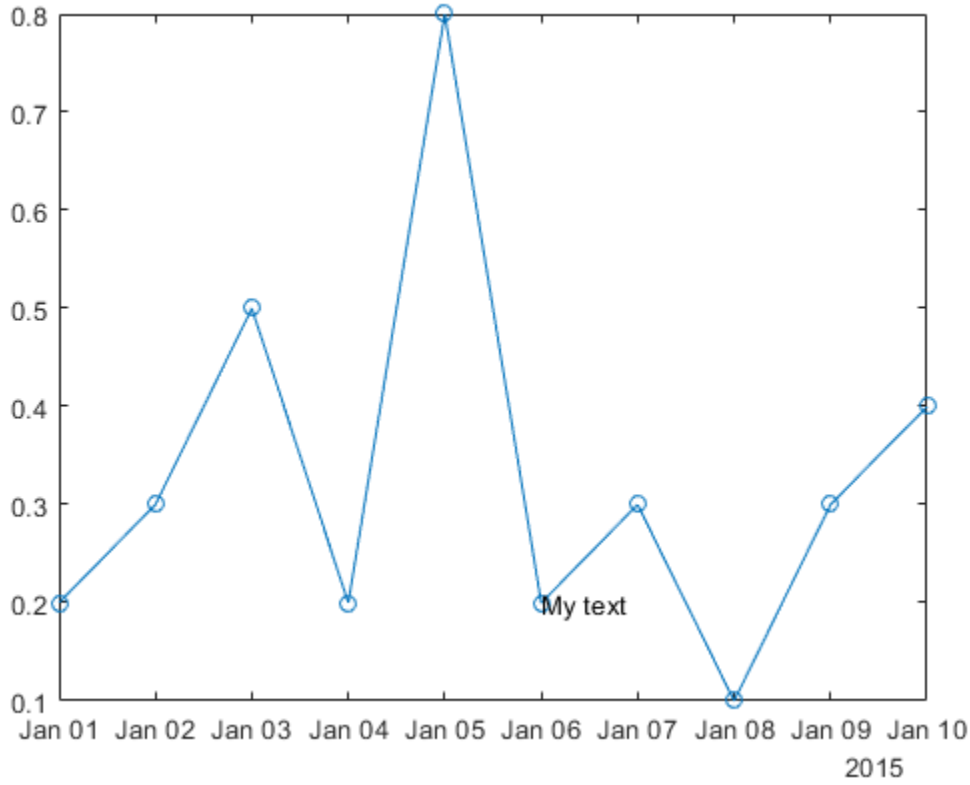
Create a plot with dates along the *x*-axis. Add a text description to the fourth data point and return the text object.

```
t = datetime(2015,1,1:10);
y = [.2 .3 .5 .2 .8 .2 .3 .1 .3 .4];
plot(t,y,'-o')
txt = text(t(4),y(4),'My text');
```



Change the position of the text to the sixth data point by setting the `Position` property of the text object. Since the `Position` property accepts only numeric values, convert the datetime value at the sixth data point to the equivalent numeric value.

```
ax = gca;
x6 = ruler2num(t(6),ax.XAxis);
txt.Position = [x6 y(6)];
```



Input Arguments

`data` — Input array
scalar | vector | matrix

Input array, specified as a scalar, vector, or matrix. If `data` is already numeric, then the output `num` is equal to the input `data`.

`ruler` — Ruler
`DatetimeRuler` object | `DurationRuler` object | `NumericRuler` object | `GeographicRuler` object

Ruler associated with the input data, specified as a `DatetimeRuler`, `DurationRuler`, `NumericRuler`, or `GeographicRuler` object. Specify the ruler object using the `XAxis`, `YAxis`, or `ZAxis` property of the `Axes` object. For example:

```
ax = gca;
num = ruler2num(data,ax.XAxis)
```

If you specify a `NumericRuler` object and the input `data` is not numeric, then the output `num` is determined by:

```
num = full(double(data))
```

If you specify a `GeographicRuler` object, then the output `num` is equal to the input `data`.

See Also

`axes` | `datetime` | `duration` | `num2ruler`

Introduced in R2016b

rulerPanInteraction

Ruler-pan interaction

Description

A ruler-pan interaction allows you to pan an axis without having to select any buttons in the axes toolbar. To enable ruler panning, set the `Interactions` property of the axes to a `rulerPanInteraction` object. When this interaction is enabled, you can pan an axis by dragging it.

To enable multiple interactions, set the `Interactions` property to an array of objects.

Creation

Syntax

```
rp = rulerPanInteraction
rp = rulerPanInteraction('Dimensions',d)
```

Description

`rp = rulerPanInteraction` creates a ruler-pan interaction object.

`rp = rulerPanInteraction('Dimensions',d)` sets the `Dimensions` property. Use this property to constrain panning to a single axis or a subset of axes. For example, `rp = rulerPanInteraction('Dimensions','x')` constrains panning to the *x*-axis.

Properties

`Dimensions` — Dimensions
`'xyz'` (default) | `'xy'` | `'yz'` | `'xz'` | `'x'` | `'y'` | `'z'`

Dimensions to allow panning, specified as one of these values:

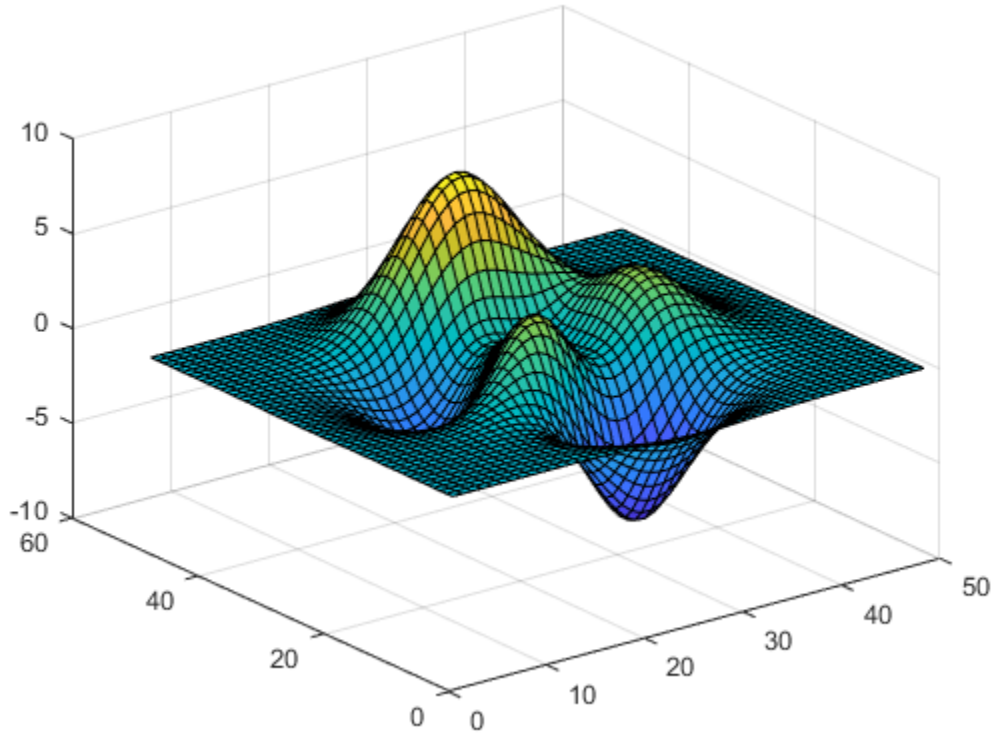
- `'xyz'` — Allows panning on all three axes.
- `'xy'` — Allows panning on the *x*- and *y*-axes only.
- `'yz'` — Allows panning on the *y*- and *z*-axes only.
- `'xz'` — Allows panning on the *x*- and *z*-axes only.
- `'x'` — Allows panning on the *x*-axis only.
- `'y'` — Allows panning on the *y*-axis only.
- `'z'` — Allows panning on the *z*-axis only.

Examples

Axes with Ruler Pan and Data Tip Interactions

Create a surface plot. Get the current axes and replace the default interactions with the ruler-pan and data tip interactions. Then hover over the surface to display data tips. Drag any axis to pan the limits.

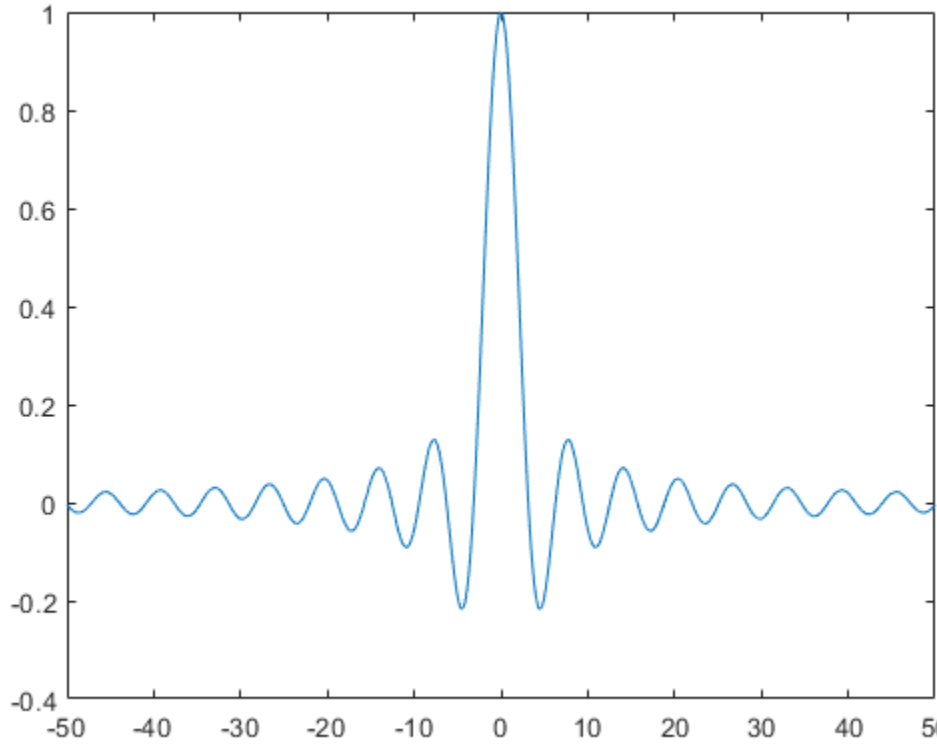
```
surf(peaks)
ax = gca;
ax.Interactions = [rulerPanInteraction dataTipInteraction];
```



Restrict Ruler Panning to X-Dimension

Create *x* and *y* values, and plot them using custom *x*-axis limits. Replace the default set of interactions with a ruler-pan interaction that is restricted to the *x*-dimension. Then drag the *x*-axis to pan. Notice that you cannot pan the *y*-axis.

```
x = linspace(-500,500,5000);
y = sin(x)./x;
plot(x,y)
xlim([-50 50])
ax = gca;
ax.Interactions = rulerPanInteraction('Dimensions','x');
```



Limitations

The `rulerPanInteraction` function is not supported in the Live Editor.

Tips

In most cases, the axes have a default set of interactions which depend on the type of chart you are displaying. You can replace the default set with a new set of interactions, but you cannot access or modify any of the interactions in the default set.

See Also

`dataTipInteraction` | `disableDefaultInteractivity` | `enableDefaultInteractivity` | `panInteraction` | `regionZoomInteraction` | `rotateInteraction` | `zoomInteraction`

Topics

"Control Chart Interactivity"

Introduced in R2019a

run

Run MATLAB script

Syntax

```
run(scriptname)
```

Description

`run(scriptname)` runs the MATLAB script specified by `scriptname`.

Examples

Run Script Not on Current Path

Create a temporary folder that is not on your current path.

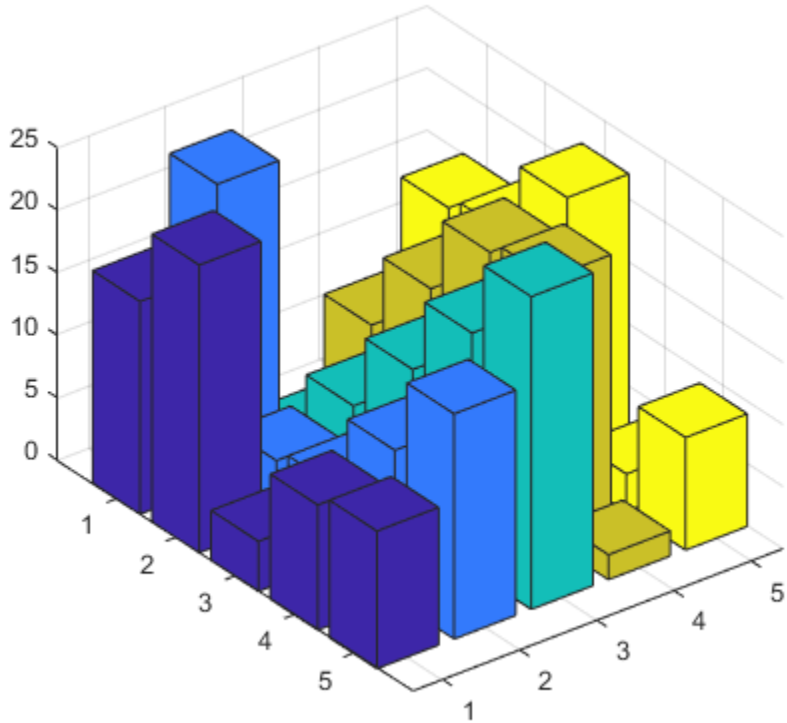
```
tmp = tempname;
mkdir(tmp)
```

Write MATLAB code to a file in the folder.

```
newFile = fullfile(tmp,'ANewFile.m');
fid = fopen(newFile,'w');
fprintf(fid,'Z = magic(5);\n');
fprintf(fid,'b = bar3(Z);\n');
fclose(fid);
```

Run the script.

```
run(newFile)
```



Input Arguments

`scriptname` — Full or relative script path
character vector | string scalar

Full or relative script path to a MATLAB script, specified as a character vector or string scalar. `scriptname` can specify any file type that MATLAB can execute, such as MATLAB script files, Simulink models, or MEX-files.

Example: `scriptname = 'myScript'`

Example: `scriptname = 'anotherScript.m'`

Example: `scriptname = 'oneMoreScript.mlx'`

Tips

- `run` can execute a script not on the MATLAB path if its input argument specifies the path to the script. To run a script by simply entering its name, you should use `cd` to navigate to the appropriate folder or `addpath` to add the folder to the MATLAB search path.
- `scriptname` can access any variables in the current workspace.
- `run` changes to the folder that contains the script, executes it, and resets back to the original folder. If the script itself changes folders, then `run` does not revert to the original folder, unless `scriptname` changes to the folder in which this script resides.
- If `scriptname` corresponds to both a `.m` file and a P-file residing in the same folder, then `run` executes the P-file. This occurs even if you specify `scriptname` with a `.m` extension.
- If a script is not on the MATLAB path, executing the `run` command caches the script. In the same session and after calling `run`, you can edit the script using an external editor. Call `clear scriptname` before calling `run` again to use the changed version of the script rather than the cached version. If you edit the script with the MATLAB editor, `run` executes the changed version and there is no need to call `clear scriptname`.

See Also

`addpath` | `cd` | `path` | `pwd`

Topics

"Files and Folders that MATLAB Accesses"

Introduced before R2006a

runperf

Run set of tests for performance measurement

Syntax

```
results = runperf
results = runperf(tests)
results = runperf(tests,Name,Value)
```

Description

`results = runperf` runs all the tests in your current folder for performance measurements and returns an array of `matlab.perftest.TimeResult` objects. Each element in `results` corresponds to an element in the test suite.

The performance test framework runs the tests using a variable number of measurements to reach a sample mean with a 0.05 relative margin of error within a 0.95 confidence level. It runs the tests four times to warm up the code, and then between 4 and 256 times to collect measurements that meet the statistical objectives. If the sample mean does not meet the 0.05 relative margin of error within a 0.95 confidence level after 256 test runs, the performance test framework stops running the test and displays a warning. In this case, the `matlab.perftest.TimeResult` object contains information for the 4 warm-up runs and 256 measurement runs.

The `runperf` function provides a simple way to run a collection of tests as a performance experiment.

`results = runperf(tests)` runs a specified set of tests.

`results = runperf(tests,Name,Value)` runs a set of tests with additional options specified by one or more `Name,Value` pair arguments.

Examples

Run Script as Performance Test

In your current working folder, create a script-based test, `onesTest.m`, that uses three different methods to initialize a `1000`-by-`1500` matrix of ones.

```
rows = 1000;
cols = 1500;

%% Ones Function
X = ones(rows,cols);

%% Loop Assignment Without Preallocation
for r = 1:rows
    for c = 1:cols
        X(r,c) = 1;
    end
end

%% Loop Assignment With Preallocation
X = zeros(rows,cols);
for r = 1:rows
    for c = 1:cols
        X(r,c) = 1;
    end
end
```

Run the script as a performance test. Your results might vary.

```
results = runperf('onesTest');
```

```
Running onesTest
......... ......... ......... ......... .........
......... ......... ......... ......... ....
Done onesTest
__________
```

Display the results. The `results` variable is a `1`-by-`3` `TimeResult` array. Each element in the array corresponds to one of the tests defined in the code section in `onesTest.m`.

```
results
```

```
results =

  1×3 TimeResult array with properties:
```

```
    Name
    Valid
    Samples
    TestActivity

Totals:
   3 Valid, 0 Invalid.
   4.4113 seconds testing time.
```

Display the measurement results for the second test, which loops the assignment without preallocation.

```
results(2)
```

```
ans = 

  TimeResult with properties:

            Name: 'onesTest/LoopAssignmentWithoutPreallocation'
           Valid: 1
         Samples: [4×7 table]
    TestActivity: [8×12 table]

Totals:
   1 Valid, 0 Invalid.
   3.1462 seconds testing time.
```

Display the complete table of test measurements.

```
results(2).TestActivity
```

```
results(2).TestActivity

ans = 

  8×12 table

                      Name                        Passed    Failed    Incomplete    MeasuredTime    Objective         Tim
    ______________________________________________ ______ ______ __________ ____________ _________ ________
    onesTest/LoopAssignmentWithoutPreallocation    true      false       false         0.39586        warmup      24-Jun-2
    onesTest/LoopAssignmentWithoutPreallocation    true      false       false         0.38351        warmup      24-Jun-2
    onesTest/LoopAssignmentWithoutPreallocation    true      false       false         0.37995        warmup      24-Jun-2
    onesTest/LoopAssignmentWithoutPreallocation    true      false       false         0.38603        warmup      24-Jun-2
    onesTest/LoopAssignmentWithoutPreallocation    true      false       false         0.38388        sample      24-Jun-2
    onesTest/LoopAssignmentWithoutPreallocation    true      false       false         0.39803        sample      24-Jun-2
    onesTest/LoopAssignmentWithoutPreallocation    true      false       false         0.39742        sample      24-Jun-2
    onesTest/LoopAssignmentWithoutPreallocation    true      false       false         0.37702        sample      24-Jun-2
```

The performance testing framework ran four warm-up runs, followed by four measurements runs (indicated as `sample` in the `Objective` column).

Display the mean measured time for the second test. To exclude data collected in the warm-up runs, use the values in the `Samples` field.

```
mean(results(2).Samples.MeasuredTime)
```

```
ans =

    0.3891
```

To compare the different initialization methods in the script, display the mean measured time for all the tests. Concatenate the values from the `Samples` field across the three elements in the `results` array. Then use `varfun` to group the table entries by name and compute the mean.

```
fullTable = vertcat(results.Samples);
varfun(@mean,fullTable,'InputVariables','MeasuredTime','GroupingVariables','Name')
```

```
ans =

  3×3 table

                      Name                        GroupCount    mean_MeasuredTime
    ____________________________________________    __________    _________________

    onesTest/OnesFunction                              65            0.0063079
    onesTest/LoopAssignmentWithoutPreallocation         4              0.38909
    onesTest/LoopAssignmentWithPreallocation           13             0.018792
```

In the example output, the `ones` function was the fastest way to initialize the matrix to ones. The performance testing framework made `65` measurement runs for this test. Your results might vary.

Performance Tests from Select Unit Tests

In your current working folder, create a class-based test, `preallocationTest.m`, that compares different methods of preallocation.

```
classdef preallocationTest < matlab.perftest.TestCase
    methods(Test)
        function testOnes(testCase)
            x = ones(1,1e7);
        end

        function testIndexingWithVariable(testCase)
            id = 1:1e7;
```

```
            x(id) = 1;
        end

        function testIndexingOnLHS(testCase)
            x(1:1e7) = 1;
        end

        function testForLoop(testCase)
            for i=1:1e7
                x(i) = 1;
            end
        end

    end
end
```

The measurement boundary for the `preallocationTest` class is the test method. The time measurement for each test method includes all the code in method. For information on designating measurement boundaries, see the `startMeasuring` and `stopMeasuring` methods of `matlab.perftest.TestCase`.

Run performance tests for all the elements that contain `'Indexing'` in the name. Your results might vary, and you might see a warning if `runperf` doesn't meet statistical objectives.

```
results = runperf('preallocationTest','Name','*Indexing*')
```

```
Running preallocationTest
.......... .......... .......
Done preallocationTest
__________


results =

  1×2 TimeResult array with properties:

    Name
    Valid
    Samples
    TestActivity

Totals:
```

```
   2 Valid, 0 Invalid.
   2.4858 seconds testing time.
```

Display the mean measured time for each of the tests. Concatenate the values from the `Samples` field across the two elements in the `results` array. Then use `varfun` to group the table entries by name and compute the mean.

```
fullTable = vertcat(results.Samples);
varfun(@mean,fullTable,'InputVariables','MeasuredTime','GroupingVariables','Name')
```

```
ans =

  2×3 table

                        Name                        GroupCount    mean_MeasuredTime
    ____________________________________________    __________    _________________

    preallocationTest/testIndexingWithVariable          6             0.16337
    preallocationTest/testIndexingOnLHS                13            0.049936
```

Input Arguments

`tests` — Suite of tests

character vector | cell array of character vectors

Suite of tests specified as a character vector or cell array of character vectors. Each character vector in the cell array can contain the name of a test file, a test class, a test suite element name, a package containing your test classes, a folder containing your test files, or a project folder containing test files.

Example: `runperf('ATestFile.m')`

Example: `runperf('ATestFile/aTest')`

Example: `runperf('mypackage.MyTestClass')`

Example: `runperf(pwd)`

Example: `runperf({'mypackage.MyTestClass','ATestFile.m',pwd,'mypackage.subpackage'})`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `runperf(tests,'Name','productA_*')` runs test elements with a name that starts with `'productA_'`.

`BaseFolder` — Name of base folder
character vector

Name of the base folder that contains the file defining the test class, function, or script, specified as a character vector. This argument filters `TestSuite` array elements. For a test element to be included in the suite, the test element must be contained in the specified base folder. Use the wildcard character `*` to match any number of characters. Use the question mark character `?` to match a single character. For test files defined in packages, the base folder is the parent of the top-level package folder.

`IncludeSubfolders` — Indicator to run tests in subfolders
`false` (default) | `true` | `0` | `1`

Indicator to run tests in subfolders, specified as `false` or `true` (`0` or `1`). By default the framework runs tests in the specified folders, but not in their subfolders.

Data Types: `logical`

`IncludeSubpackages` — Indicator to run tests in subpackages
`false` (default) | `true` | `0` | `1`

Indicator to run tests in subpackages, specified as `false` or `true` (`0` or `1`). By default the framework runs tests in the specified packages, but not in their subpackages.

Data Types: `logical`

`Name` — Name of suite element
character vector

Name of the suite element, specified as a character vector. This argument filters `TestSuite` array elements. For the testing framework to run a test, the `Name` property of the test element must match the specified name. Use the wildcard character, `*`, to match

any number of characters. Use the question mark character, `?`, to match a single character.

`ParameterName` — Name of parameter
character vector

Name of a parameter used by the test suite element, specified as a character vector. This argument filters `TestSuite` array elements. Use the wildcard character * to match any number of characters. Use the question mark character ? to match a single character.

`ParameterProperty` — Name of parameterization property
character vector

Name of a parameterization property used by the test suite element, specified as a character vector. This argument filters `TestSuite` array elements. Use the wildcard character * to match any number of characters. Use the question mark character ? to match to a single character.

`ProcedureName` — Name of the test procedure in the test
character vector

Name of test procedure, specified as a character vector. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

In a class-based test, the `ProcedureName` is the name of the test method. In a function-based, test it is the name of the local function that contains the test. In a script-based test, it is a name generated from the test section title. Unlike `Name`, the name of the test procedure does not include any class or package name or information about parameterization.

`Superclass` — Name of a class that the test class derives from
character vector | string scalar

Name of a class that the test class derives from, specified as a character vector or string scalar. This argument filters `TestSuite` array elements.

`Tag` — Name of test element tag
character vector | string scalar

Name of test element tag, specified as a character vector or string scalar. This argument filters `TestSuite` array elements. Use the wildcard character * to match any number of characters. Use the question mark character ? to match a single character.

Tips

- To customize the statistical objectives of the performance test, use the `TimeExperiment` class to construct and run the performance test.

Alternatives

To create a test suite explicitly, you can use the `testsuite` function or the `matlab.unittest.TestSuite` methods to create a suite. Then, you can run your performance test with the `run` method of your specified `TimeExperiment`.

See Also

`comparisonPlot` | `matlab.perftest.FrequentistTimeExperiment` | `matlab.unittest.measurement.MeasurementResult` | `runtests`

Topics

"Overview of Performance Testing Framework"

Introduced in R2016a

runtests

Run set of tests

Syntax

```
results = runtests
results = runtests(tests)
results = runtests(tests,Name,Value)
```

Description

`results = runtests` runs all the tests in your current folder, and returns the results as a `TestResult` object.

`results = runtests(tests)` runs a specified set of tests.

`results = runtests(tests,Name,Value)` runs a set of tests with additional options specified by one or more `Name,Value` pair arguments.

Examples

Run Tests in Working Folder

Create a folder `myExample` in your current working folder, and change into that folder.

In the `myExample` folder, create a test script, `typeTest.m`.

```
%% Test double class
exp = 'double';
act = ones;
assert(isa(act,exp))

%% Test single class
```

```
exp = 'single';
act = ones('single');
assert(isa(act,exp))

%% Test uint16 class
exp = 'uint16';
act = ones('uint16');
assert(isa(act,exp))
```

In the `myExample` folder, create a test script, `sizeValueTest.m`.

```
%% Test size
exp = [7 13];
act = ones([7 13]);
assert(isequal(size(act),exp))

%% Test values
act = ones(42);
assert(unique(act) == 1)
```

Run all tests in the current folder.

```
runtests
```

```
Running sizeValueTest
..
Done sizeValueTest
__________

Running typeTest
...
Done typeTest
__________


ans =

  1x5 TestResult array with properties:

    Name
    Passed
    Failed
```

```
    Incomplete
    Duration
    Details

Totals:
   5 Passed, 0 Failed, 0 Incomplete.
   0.038077 seconds testing time.
```

MATLAB® ran 5 tests. There are 2 passing tests from `sizeValueTest` and 3 passing tests from `typeTest`.

Run Tests Using File Name

Create the test file shown below, and save it as `runtestsExampleTest.m` on your MATLAB path.

```
function tests = runtestsExampleTest
tests = functiontests(localfunctions);

function testFunctionOne(testCase)
```

Run the tests.

```
results = runtests('runtestsExampleTest.m');
```

```
Running runtestsExampleTest
.
Done runtestsExampleTest
__________
```

Run Tests in Subfolder

If it does not exist, create the test file, `runtestsExampleTest.m`, in the example above.

Create a subfolder, `tmpTest`, and, in that folder, create the following `runtestsExampleSubFolderTest.m` file.

```
function tests = runtestsExampleSubFolderTest
tests = functiontests(localfunctions);

function testFunctionTwo(testCase)
```

Run the tests from the folder above `tmpTest` by setting `'IncludeSubfolders'` to true.

```
results = runtests(pwd,'IncludeSubfolders',true);
```

```
Running runtestsExampleTest
.
Done runtestsExampleTest
__________

Running runtestsExampleSubFolderTest
.
Done runtestsExampleSubFolderTest
__________
```

`runtests` ran the tests in both the current folder and the subfolder.

If you do not specify the `'IncludeSubfolders'` property for the `runtests` function, it does not run the test in the subfolder.

```
results = runtests(pwd);
```

```
Running runtestsExampleTest
.
Done runtestsExampleTest
__________
```

Run Tests in Parallel

Create the following test file, and save it as `runInParallelTest.m` on your MATLAB path.

```
function tests = runInParallelTest
tests = functiontests(localfunctions);

function testA(testCase)
verifyEqual(testCase,5,5);

function testB(testCase)
```

```
verifyTrue(testCase,logical(1));

function testC(testCase)
verifySubstring(testCase,'SomeLongText','Long');

function testD(testCase)
verifySize(testCase,ones(2,5,3),[2 5 3]);

function testE(testCase)
verifyGreaterThan(testCase,3,2);

function testF(testCase)
verifyEmpty(testCase,{},'Cell array is not empty.');

function testG(testCase)
verifyMatches(testCase,'Some Text','Some [Tt]ext');
```

Run the tests in parallel. Running tests in parallel requires the Parallel Computing Toolbox. The testing framework might vary the order and number of groups or which tests it includes in each group.

```
results = runtests('runInParallelTest','UseParallel',true);
```

```
Split tests into 7 groups and running them on 4 workers.
----------------
Finished Group 2
----------------
Running runInParallelTest
.
Done runInParallelTest
__________


----------------
Finished Group 3
----------------
Running runInParallelTest
.
Done runInParallelTest
__________


----------------
Finished Group 1
----------------
```

```
Running runInParallelTest
.
Done runInParallelTest
__________

----------------
Finished Group 4
----------------
Running runInParallelTest
.
Done runInParallelTest
__________

----------------
Finished Group 6
----------------
Running runInParallelTest
.
Done runInParallelTest
__________

----------------
Finished Group 5
----------------
Running runInParallelTest
.
Done runInParallelTest
__________

----------------
Finished Group 7
----------------
Running runInParallelTest
.
Done runInParallelTest
__________
```

Run Select Parameterized Tests

In your working folder, create `testZeros.m`. This class contains four test methods.

```
classdef testZeros < matlab.unittest.TestCase
    properties (TestParameter)
        type = {'single','double','uint16'};
        outSize = struct('s2d',[3 3], 's3d',[2 5 4]);
    end

    methods (Test)
        function testClass(testCase, type, outSize)
            testCase.verifyClass(zeros(outSize,type), type);
        end

        function testSize(testCase, outSize)
            testCase.verifySize(zeros(outSize), outSize);
        end

        function testDefaultClass(testCase)
            testCase.verifyClass(zeros, 'double');
        end
        function testDefaultSize(testCase)
            testCase.verifySize(zeros, [1 1]);
        end

        function testDefaultValue(testCase)
            testCase.verifyEqual(zeros,0);
        end
    end
end
```

The full test suite has 11 test elements: 6 from the `testClass` method, 2 from the `testSize` method, and 1 each from the `testDefaultClass`, `testDefaultSize`, and `testDefaultValue` methods.

At the command prompt, run all the parameterizations for the `testSize` method.

```
runtests('testZeros/testSize')
```

```
Running testZeros
..
Done testZeros
```

```
__________

ans =

  1x2 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
   2 Passed, 0 Failed, 0 Incomplete.
   0.05129 seconds testing time.
```

The `runtests` function executed the two parameterized tests from the `testSize` method. Alternatively, you can specify the test procedure name with `runtests('testZeros','ProcedureName','testSize')`.

Run the test elements that use the `outSize` parameter property.

```
runtests('testZeros','ParameterProperty','outSize')
```

```
Running testZeros
........
Done testZeros
__________

ans =

  1x8 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
```

```
   8 Passed, 0 Failed, 0 Incomplete.
   0.07314 seconds testing time.
```

The `runtests` function executed eight tests that use the `outSize` parameter property: six from the `testClass` method and two from the `testSize` method.

Run the test elements that use the `single` parameter name.

```
runtests('testZeros','ParameterName','single')
```

```
Running testZeros
..
Done testZeros
__________


ans =

  1x2 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
   2 Passed, 0 Failed, 0 Incomplete.
   0.0079192 seconds testing time.
```

The `runtests` function executed the two tests from the `testClass` method that use the `outSize` parameter name.

Input Arguments

`tests` — Array of tests

character vector | cell array of character vectors

Suite of tests specified as a character vector or cell array of character vectors. Each character vector in the cell array can contain the name of a test file, a test class, a test

suite element name, a package containing your test classes, a folder containing your test files, or a project folder containing test files.

Example: `runtests('ATestFile.m')`

Example: `runtests('ATestFile/aTest')`

Example: `runtests('mypackage.MyTestClass')`

Example: `runtests(pwd)`

Example: `runtests({'mypackage.MyTestClass','ATestFile.m',pwd,'mypackage.subpackage'})`

Example: `runtests('C:/projects/project1/')`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `runtests(tests,'Name','productA_*')` runs test elements with a name that starts with `'productA_'`.

`BaseFolder` — Name of base folder
character vector

Name of the base folder that contains the file defining the test class, function, or script, specified as a character vector. This argument filters `TestSuite` array elements. For a test element to be included in the suite, the test element must be contained in the specified base folder. Use the wildcard character `*` to match any number of characters. Use the question mark character `?` to match a single character. For test files defined in packages, the base folder is the parent of the top-level package folder.

`Debug` — Indicator to apply debugging capabilities
`false` (default) | `true` | `0` | `1`

Indicator to apply debugging capabilities when running `tests`, specified as `false` or `true` (`0` or `1`). For example, if a test failure is encountered, the framework pauses test execution to enter debug mode.

Data Types: `logical`

`IncludeSubfolders` — Indicator to run tests in subfolders
`false` (default) | `true` | `0` | `1`

Indicator to run tests in subfolders, specified as `false` or `true` (`0` or `1`). By default the framework runs tests in the specified folders, but not in their subfolders.

Data Types: `logical`

`IncludeSubpackages` — Indicator to run tests in subpackages
`false` (default) | `true` | `0` | `1`

Indicator to run tests in subpackages, specified as `false` or `true` (`0` or `1`). By default the framework runs tests in the specified packages, but not in their subpackages.

Data Types: `logical`

`IncludeReferencedProjects` — Indicator to include tests from referenced projects
`false` (default) | `true` | `0` | `1`

Indicator to include tests from referenced projects, specified as logical `false` or `true`. For more information on referenced projects, see “Componentize Large Projects”.

Data Types: `logical`

`LoggingLevel` — Maximum verbosity level for logged diagnostics
1 (default) | 0 | 2 | 3 | 4 | `matlab.unittest.Verbosity` enumeration

Maximum verbosity level for logged diagnostics included for the test run, specified as an integer value from 0 through 4, or as a `matlab.unittest.Verbosity` enumeration object. The `runtests` function includes diagnostics that are logged at this level and below. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration.

By default `runtests` includes diagnostics logged at the `matlab.unittest.Verbosity.Terse` level (level 1). To exclude logged diagnostics, specify `LoggingLevel` as `Verbosity.None` (level 0).

Logged diagnostics are diagnostics that you supply to the testing framework with a call to the `log` or `log` method.

Numeric Representation	Enumeration Member Name	Verbosity Description
0	None	No information
1	Terse	Minimal information
2	Concise	Moderate amount of information
3	Detailed	Some supplemental information
4	Verbose	Lots of supplemental information

`OutputDetail` — Display level for event details

2 (default) | 0 | 1 | 3 | 4 | `matlab.unittest.Verbosity` enumeration

Display level for event details, specified as an integer value from 0 through 4, or as a `matlab.unittest.Verbosity` enumeration object. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration.

The `runtests` function displays failing and logged events with the amount of detail specified by `OutputDetail`. By default the `runtests` displays events at the `matlab.unittest.Verbosity.Concise` level (level 2).

Numeric Representation	Enumeration Member Name	Verbosity Description
0	None	No information
1	Terse	Minimal information
2	Concise	Moderate amount of information
3	Detailed	Some supplemental information
4	Verbose	Lots of supplemental information

`Name` — Name of suite element

character vector

Name of the suite element, specified as a character vector. This argument filters `TestSuite` array elements. For the testing framework to run a test, the `Name` property of the test element must match the specified name. Use the wildcard character, *, to match

any number of characters. Use the question mark character, `?`, to match a single character.

`ParameterName` — Name of parameter
character vector

Name of a parameter used by the test suite element, specified as a character vector. This argument filters `TestSuite` array elements. Use the wildcard character `*` to match any number of characters. Use the question mark character `?` to match a single character.

`ParameterProperty` — Name of parameterization property
character vector

Name of a parameterization property used by the test suite element, specified as a character vector. This argument filters `TestSuite` array elements. Use the wildcard character `*` to match any number of characters. Use the question mark character `?` to match to a single character.

`ProcedureName` — Name of the test procedure in the test
character vector

Name of test procedure, specified as a character vector. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

In a class-based test, the `ProcedureName` is the name of the test method. In a function-based, test it is the name of the local function that contains the test. In a script-based test, it is a name generated from the test section title. Unlike `Name`, the name of the test procedure does not include any class or package name or information about parameterization.

`ReportCoverageFor` — Path to source code to include in code coverage report
path to file or folder

Path to source code to include in code coverage report, specified as a character vector, string array, or cell array of character vectors. Using this option with `runtests` runs the specified tests and produces a code coverage report for the specified code files. The report shows which lines in the source code were executed by the tests.

The source code can be an absolute or relative path to one or more folders or to files that have a `.m` or `.mlx` extension.

Example: `runtests(tests,'ReportCoverageFor','mySource.m')`

Data Types: `char` | `string` | `cell`

`Strict` — Indicator to apply strict checks
`false` (default) | `true` | `0` | `1`

Indicator to apply strict checks when running `tests`, specified as `false` or `true` (`0` or `1`). For example, the framework generates a qualification failure if a test issues a warning.

Data Types: `logical`

`Superclass` — Name of a class that the test class derives from
character vector | string scalar

Name of a class that the test class derives from, specified as a character vector or string scalar. This argument filters `TestSuite` array elements.

`Tag` — Name of test element tag
character vector | string scalar

Name of test element tag, specified as a character vector or string scalar. This argument filters `TestSuite` array elements. Use the wildcard character `*` to match any number of characters. Use the question mark character `?` to match a single character.

`UseParallel` — Indicator to run tests in parallel
`false` (default) | `true` | `0` | `1`

Indicator to run tests in parallel, specified as `false` or `true` (`0` or `1`). By default `runtests` runs tests in serial. If you set `UseParallel` to `true`, the `runtests` function divides the test suite into separate groups and runs the groups in parallel if:

- The Parallel Computing Toolbox is installed.
- An open parallel pool exists or automatic pool creation is enabled in the Parallel Preferences.

Otherwise, `runtests` runs tests in serial regardless of the value for `UseParallel`.

Note Running tests in parallel requires the Parallel Computing Toolbox. The testing framework might vary the order and number of groups or which tests it includes in each group.

Testing in parallel might not be compatible with other options. For example, testing occurs in serial if `'UseParallel'` and `'Debug'` are both set to true.

Data Types: `logical`

See Also

`functiontests` | `matlab.unittest.TestResult` | `matlab.unittest.TestRunner` | `matlab.unittest.TestSuite`

Topics

"Write Function-Based Unit Tests"
"Write Simple Test Case Using Functions"
"Write Test Using Setup and Teardown Functions"

Introduced in R2013b

Surface Properties

Chart surface appearance and behavior

Description

`Surface` properties control the appearance and behavior of `Surface` objects. By changing property values, you can modify certain aspects of the surface chart.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = surf(...);
c = h.CData;
h.CDataMapping = 'direct';
```

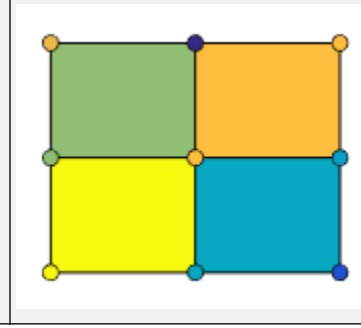
If you are using an earlier release, use the `get` and `set` functions instead.

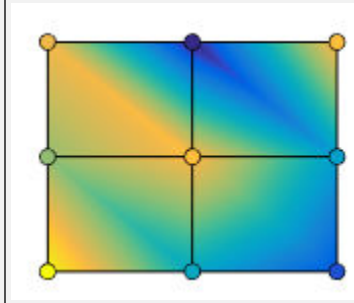
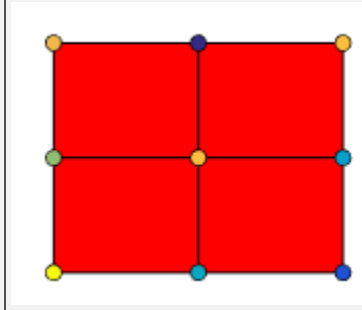
Properties

Faces

`FaceColor` — Face color
`'flat'` (default) | `'interp'` | `'none'` | `'texturemap'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Face color, specified as one of the values in this table.

Value	Description
`'flat'`	Use a different color for each face based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color value at the first vertex of each face (in the positive *x* and *y* directions) determines the color for the entire face. You cannot use this value when the `FaceAlpha` property is set to `'interp'`. 

Value	Description
`'interp'`	Use interpolated coloring for each face based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color varies across each face by interpolating the color values at the vertices. You cannot use this value when the `FaceAlpha` property is set to `'flat'`. 
RGB triplet, hexadecimal color code, or color name	Use the specified color for all the faces. This option does not use the color values in the `CData` property. 
`'texturemap'`	Transform the color data in `CData` so that it conforms to the surface.
`'none'`	Do not draw the faces.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

FaceAlpha — Face transparency
1 (default) | scalar in range `[0,1]` | `'flat'` | `'interp'` | `'texturemap'`

Face transparency, specified as one of these values:

- Scalar in range `[0,1]` — Use uniform transparency across all the faces. A value of `1` is fully opaque and `0` is completely transparent. Values between `0` and `1` are semitransparent. This option does not use the transparency values in the `AlphaData` property.
- `'flat'` — Use a different transparency for each face based on the values in the `AlphaData` property. The transparency value at the first vertex determines the transparency for the entire face. First you must specify the `AlphaData` property as a matrix the same size as the `ZData` property. The `FaceColor` property also must be set to `'flat'`.
- `'interp'` — Use interpolated transparency for each face based on the values in `AlphaData` property. The transparency varies across each face by interpolating the values at the vertices. First you must specify the `AlphaData` property as a matrix the same size as the `ZData` property. The `FaceColor` property also must be set to `'interp'`.
- `'texturemap'` — Transform the data in `AlphaData` so that it conforms to the surface.

FaceLighting — Effect of light objects on faces
`'flat'` (default) | `'gouraud'` | `'none'`

Effect of light objects on faces, specified as one of these values:

- `'flat'` — Apply light uniformly across each face. Use this value to view faceted objects.
- `'gouraud'` — Vary the light across the faces. Calculate the light at the vertices and then linearly interpolate the light across the faces. Use this value to view curved surfaces.
- `'none'` — Do not apply light from light objects to the faces.

To add a light object to the axes, use the `light` function.

Note The `'phong'` value has been removed. Use `'gouraud'` instead.

BackFaceLighting — Face lighting when normals point away from camera

`'reverselit'` (default) | `'unlit'` | `'lit'`

Face lighting when the vertex normals point away from camera, specified as one of these values:

- `'reverselit'` — Light the face as if the vertex normal pointed towards the camera.
- `'unlit'` — Do not light the face.
- `'lit'` — Light the face according to the vertex normal.

Use this property to discriminate between the internal and external surfaces of an object. For an example, see “Back Face Lighting”.

Edges

MeshStyle — Edges to display

`'both'` (default) | `'row'` | `'column'`

Edges to display, specified as `'both'`, `'row'`, or `'column'`.

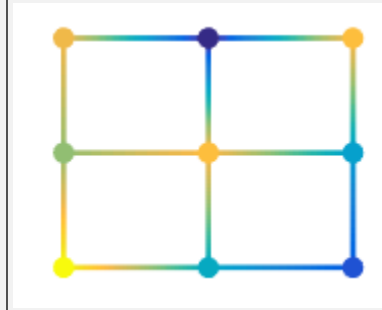
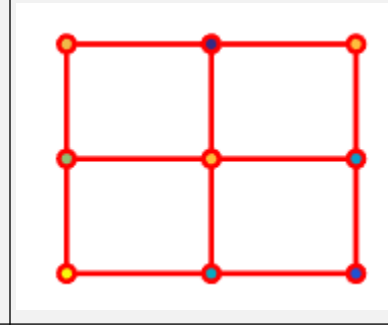
EdgeColor — Edge line color

`[0 0 0]` (default) | `'none'` | `'flat'` | `'interp'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Edge line color, specified as one of the values listed here. The default color of `[0 0 0]` corresponds to black edges.

Value	Description
`'none'`	Do not draw the edges.

Value	Description
`'flat'`	Use a different color for each edge based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color value at the first vertex of each face (in the positive *x* and *y* directions) determines the color for the adjacent edges. You cannot use this value when the `EdgeAlpha` property is set to `'interp'`.

Value	Description
`'interp'`	Use interpolated coloring for each edge based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color varies across each edge by linearly interpolating the color values at the vertices. You cannot use this value when the `EdgeAlpha` property is set to `'flat'`. 
RGB triplet, hexadecimal color code, or color name	Use the specified color for all the edges. This option does not use the color values in the `CData` property. 

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`EdgeAlpha` — Edge transparency

1 (default) | scalar value in range `[0,1]` | `'flat'` | `'interp'`

Edge transparency, specified as one of these values:

- Scalar in range `[0,1]` — Use uniform transparency across all of the edges. A value of `1` is fully opaque and `0` is completely transparent. Values between `0` and `1` are semitransparent. This option does not use the transparency values in the `AlphaData` property.
- `'flat'` — Use a different transparency for each edge based on the values in the `AlphaData` property. First you must specify the `AlphaData` property as a matrix the same size as the `ZData` property. The transparency value at the first vertex determines the transparency for the entire edge. The `EdgeColor` property also must be set to `'flat'`.
- `'interp'` — Use interpolated transparency for each edge based on the values in `AlphaData` property. First you must specify the `AlphaData` property as a matrix the same size as the `ZData` property. The transparency varies across each edge by interpolating the values at the vertices. The `EdgeColor` property also must be set to `'interp'`.

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

`AlignVertexCenters` — Sharp vertical and horizontal lines
`'off'` (default) | `'on'`

Sharp vertical and horizontal lines, specified as `'off'` or `'on'`.

If the associated figure has a `GraphicsSmoothing` property set to `'on'` and a `Renderer` property set to `'opengl'`, then the figure applies a smoothing technique to plots. In some cases, this smoothing technique can cause vertical and horizontal lines to appear uneven in thickness or color. Use the `AlignVertexCenters` property to eliminate the uneven appearance.

- `'off'` — Do not sharpen vertical or horizontal lines. The lines might appear uneven in thickness or color.
- `'on'` — Sharpen vertical and horizontal lines to eliminate an uneven appearance.

Note You must have a graphics card that supports this feature. To see if the feature is supported, call the `rendererinfo` function. If it is supported, `rendererinfo` returns value of `1` for `info.Details.SupportsAlignVertexCenters`.

`EdgeLighting` — Effect of light objects on edges

`'none'` (default) | `'flat'` | `'gouraud'`

Effect of light objects on edges, specified as one of these values:

- `'flat'` — Apply light uniformly across the each edges.
- `'none'` — Do not apply lights from light objects to the edges.
- `'gouraud'` — Calculate the light at the vertices, and then linearly interpolate across the edges.

Note The `'phong'` value has been removed. Use `'gouraud'` instead.

Markers

`Marker` — Marker symbol

`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | ...

Marker symbol, specified as one of the values listed in this table. By default, the object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerSize` — Marker size

6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

`MarkerEdgeColor` — Marker outline color

`'auto'` (default) | `'flat'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'`

Marker outline color, specified as `'auto'`, `'flat'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` option uses the same color as the `EdgeColor` property. The `'flat'` option uses the `CData` value at the vertex to set the color.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to

F. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | `'flat'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, `'flat'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` option uses the same color as the `Color` property for the axes. The `'flat'` option uses the `CData` value of the vertex to set the color.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

This property affects only the circle, square, diamond, pentagram, hexagram, and the four triangle marker types.

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Example: `'#D2F9A7'`

Coordinate Data

XData — *x*-coordinate data

matrix | vector

x-coordinate data specified as a matrix that is the same size as `ZData` or as a vector of length `n`, where `[m,n] = size(ZData)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

XDataMode — Selection mode for XData

`'auto'` | `'manual'`

Selection mode for `XData`, specified as one of these values:

- `'auto'` — Use the column indices of `ZData`..
- `'manual'` — Use manually specified values. To specify the values, use an input argument to the plotting function or directly set the `XData` property.

XDataSource — Variable linked to XData

`''` (default) | character vector | string

Variable linked to `XData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `XData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `XData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'x'`

`YData` — *y*-coordinate data
matrix | vector

y-coordinate data, specified as a matrix that is the same size as `ZData` or a vector of length `m`, where `[m,n] = size(ZData)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`YDataMode` — Selection mode for `YData`
`'auto'` | `'manual'`

Selection mode for `YData`, specified as one of these values:

- `'auto'` — Use the row indices of `ZData`.
- `'manual'` — Use manually specified values. To specify the values, use an input argument to the plotting function or directly set the `YData` property.

`YDataSource` — Variable linked to `YData`
`''` (default) | character vector | string

Variable linked to `YData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `YData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `YData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'y'`

`ZData` — *z*-coordinate data
matrix

z-coordinate data, specified as a matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`ZDataSource` — Variable linked to `ZData`
`''` (default) | character vector | string

Variable linked to `ZData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `ZData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `ZData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'z'`

Color and Transparency Data

`CData` — Vertex colors
2-D or 3-D array

Vertex colors, specified in one of these forms:

- 2-D array — Use colormap colors. Specify a color for each vertex by setting `CData` to an array the same size as `ZData`. The `CDataMapping` property determines how these values map into the current colormap. If the `FaceColor` property is set to `'texturemap'`, then `CData` does not need to be the same size as `ZData`. However, it must be of type `double` or `uint8`. The `CData` values map to conform to the surface defined by `ZData`.
- 3-D array — Use true colors. Specify an RGB triplet color for each vertex by setting `CData` to an m-by-n-by-3 array where `[m,n] = size(ZData)`. An RGB triplet is a three-element vector that specifies the intensities of the red, green, and blue components of a color. The first page of the array contains the red components, the second the green components, and the third the blue components of the colors. Since the surface uses true colors instead of colormap colors, the `CDataMapping` property has no effect.
  - If `CData` is of type `double` or `single`, then an RGB triplet value of `[0 0 0]` corresponds to black and `[1 1 1]` corresponds to white.
  - If `CData` is an integer type, then the surface uses the full range of data to determine the color. For example, if `CData` is of type `uint8`, then `[0 0 0]` corresponds to black and `[255 255 255]` corresponds to white. If `CData` is of type `int8`, then `[-128 -128 -128]` corresponds to black and `[127 127 127]` corresponds to white.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`CDataMode` — Selection mode for `CData`
`'auto'` (default) | `'manual'`

Selection mode for `CData`, specified as one of these values:

- `'auto'` — Use the `ZData` values to set the colors.
- `'manual'` — Use manually specified values. To specify the values, set the `CData` property.

`CDataSource` — Variable linked to `CData`
`''` | character vector | string

Variable linked to `CData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `CData`.

By default, there is no linked variable, so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `CData` values immediately. To force an update of the values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, then you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`CDataMapping` — Color mapping method
`'scaled'` (default) | `'direct'`

Color mapping method, specified as `'scaled'` or `'direct'`. Use this property to control the mapping of color data values in `CData` into the colormap.

The methods have these effects:

- `'direct'` — Interpret the values as indices into the current colormap. Values with a decimal portion are fixed to the nearest lower integer.
  - If the values are of type `double` or `single`, then values of `1` or less map to the first color in the colormap. Values equal to or greater than the length of the colormap map to the last color in the colormap.
  - If the values are of type `uint8`, `uint16`, `uint32`, `uint64` , `int8`, `int16`, `int32`, or `int64`, then values of `0` or less map to the first color in the colormap. Values equal to or greater than the length of the colormap map to the last color in the colormap (or up to the range limits of the type).
  - If the values are of type `logical`, then values of `0` map to the first color in the colormap and values of `1` map to the second color in the colormap.
- `'scaled'` — Scale the values to range between the minimum and maximum color limits. The `CLim` property of the axes contains the color limits.

`AlphaData` — Transparency data
1 (default) | array same size as `ZData`

Transparency data for each vertex, specified as an array the same size as the `ZData` property. After specifying the values, set the `FaceAlpha` and `EdgeAlpha` properties to

control the type of transparency. If the `FaceAlpha` and `EdgeAlpha` properties are both set to scalar values, then the surface does not use the `AlphaData` values.

The `AlphaDataMapping` property determines how the surface interprets the `AlphaData` property values.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`AlphaDataMapping` — Interpretation of `AlphaData` values

`'scaled'` (default) | `'direct'` | `'none'`

Interpretation of `AlphaData` values, specified as one of these values:

- `'none'` — Interpret the values as transparency values. A value of 1 or greater is completely opaque, a value of 0 or less is completely transparent, and a value between 0 and 1 is semitransparent.
- `'scaled'` — Map the values into the figure's alphamap. The minimum and maximum alpha limits of the axes determine the `AlphaData` values that map to the first and last elements in the alphamap, respectively. For example, if the alpha limits are `[3 5]`, then values of `3` or less map to the first element in the alphamap. Values of `5` or greater map to the last element in the alphamap. The `ALim` property of the axes contains the alpha limits. The `Alphamap` property of the figure contains the alphamap.
- `'direct'` — Interpret the values as indices into the figure's alphamap. Values with a decimal portion are fixed to the nearest lower integer.
  - If the values are of type `double` or `single`, then values of 1 or less map to the first element in the alphamap. Values equal to or greater than the length of the alphamap map to the last element in the alphamap.
  - If the values are of integer type, then values of 0 or less map to the first element in the alphamap. Values equal to or greater than the length of the alphamap map to the last element in the alphamap (or up to the range limits of the type). The integer types are `uint8`, `uint16`, `uint32`, `uint64` , `int8`, `int16`, `int32`, and `int64`.
  - If the values are of type `logical`, then values of 0 map to the first element in the alphamap and values of 1 map to the second element in the alphamap.

Normals

`VertexNormals` — Normal vectors for each surface vertex

`[]` (default) | m-by-n-by-3 array

Normal vectors for each surface vertex, specified as a m-by-n-by-3 array, where `[m,n] = size(ZData)`. Specify one normal vector per vertex.

Specifying values for this property sets the associated mode to manual. If you do not specify normal vectors, then the surface generates this data for lighting calculations.

Data Types: `single` | `double`

`VertexNormalsMode` — Selection mode for `VertexNormals`
`'auto'` (default) | `'manual'`

Selection mode for `VertexNormals`, specified as one of these values:

- `'auto'` — Calculate the normal vectors based on the coordinate data.
- `'manual'` — Use manually specified values. To specify the values, set the `VertexNormals` property.

`FaceNormals` — Normal vectors for each surface face
`[]` (default) | (m-1)-by-(n-1)-by-3 array

Normal vectors for each surface face, specified as a (m-1)-by-(n-1)-by-3 array, where `[m,n] = size(ZData)`. Specify one normal vector per face.

Specifying values for this property sets the associated mode to manual. If you do not specify normal vectors, then the surface generates this data for lighting calculations.

Data Types: `single` | `double`

`FaceNormalsMode` — Selection mode for `FaceNormals`
`'auto'` (default) | `'manual'`

Selection mode for `FaceNormals`, specified as one of these values:

- `'auto'` — Calculate the normal vectors based on the coordinate data.
- `'manual'` — Use manually specified values. To specify the values, set the `FaceNormals` property.

Lighting

`AmbientStrength` — Strength of ambient light
`0.3` (default) | scalar in range `[0,1]`

Strength of ambient light, specified as a scalar value in the range `[0,1]`. Ambient light is a nondirectional light that illuminates the entire scene. There must be at least one visible light object in the axes for the ambient light to be visible.

The `AmbientLightColor` property for the axes sets the color of the ambient light. The color is the same for all objects in the axes.

Example: `0.5`

Data Types: `double`

`DiffuseStrength` — Strength of diffuse light
`0.6` (default) | scalar in range `[0,1]`

Strength of diffuse light, specified as a scalar value in the range `[0,1]`. Diffuse light is the nonspecular reflectance from light objects in the axes.

Example: `0.3`

Data Types: `double`

`SpecularStrength` — Strength of specular reflection
`0.9` (default) | scalar in range `[0,1]`

Strength of specular reflection, specified as a scalar value in the range `[0,1]`. Specular reflections are the bright spots on the surface from light objects in the axes.

Example: `0.3`

Data Types: `double`

`SpecularExponent` — Size of specular spot
10 (default) | scalar greater than or equal to 1

Size of specular spot, specified as a scalar value greater than or equal to 1. Most materials have exponents in the range `[5 20]`.

Example: 7

Data Types: `double`

`SpecularColorReflectance` — Color of specular reflections
1 (default) | scalar in range `[0,1]`

Color of specular reflections, specified as a scalar value in the range `[0,1]`. A value of `1` sets the color using only the color of the light source. A value of `0` sets the color using

both the color of the object from which it reflects and the color of the light source. The `Color` property of the light contains the color of the light source. The proportions vary linearly for values in between.

Example: `0.5`

Data Types: `double`

Legend

`DisplayName` — Legend label

`''` (default) | character vector | string scalar

Legend label, specified as a character vector or string scalar. The legend does not display until you call the `legend` command. If you do not specify the text, then `legend` sets the label using the form `'dataN'`.

`Annotation` — Control for including or excluding object from legend

`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`DataTipTemplate` — Data tip content

`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

`Clipping` — Clipping of object to axes limits

`'on'` (default) | `'off'`

Clipping of the object to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the object that are outside the axes limits.
- `'off'` — Display the entire object, even if parts of it appear outside the axes limits. Parts of the object might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the object so that it is larger than the original plot.

The `Clipping` property of the axes that contains the object must be set to `'on'`. Otherwise, this property has no effect. For more information about the clipping behavior, see the `Clipping` property of the axes.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.

- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.

- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
    - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback.

MATLAB resumes executing the running callback when the interrupting callback completes.

- If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.

- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks when visible. The `Visible` property must be set to `'on'` and you must click a part of the `Surface` object that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The `HitTest` property determines if the `Surface` object responds to the click or if an ancestor does.
- `'all'` — Capture mouse clicks regardless of visibility. The `Visible` property can be set to `'on'` or `'off'` and you can click a part of the `Surface` object that has no color. The `HitTest` property determines if the `Surface` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `Surface` object passes the click through it to the object below it in the current view of the figure window. The `HitTest` property has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `Surface` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `Surface` object that has one of these:
  - `HitTest` property set to `'on'`
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

---

Note The `PickableParts` property determines if the `Surface` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent

`Axes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `Group`, or `Transform` object.

`Children` — Children

empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the chart object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object
`'surface'`

This property is read-only.

Type of graphics object, returned as `'surface'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`mesh` | `meshc` | `meshz` | `surf` | `surfc`

Topics

"Access Property Values"
"Graphics Object Properties"

Introduced before R2006a

Surface Properties

Primitive surface appearance and behavior

Description

`Surface` properties control the appearance and behavior of `Surface` objects. By changing property values, you can modify certain aspects of the surface chart.

Starting in R2014b, you can use dot notation to query and set properties.

```
s = surface;
c = s.CData;
s.CDataMapping = 'direct';
```

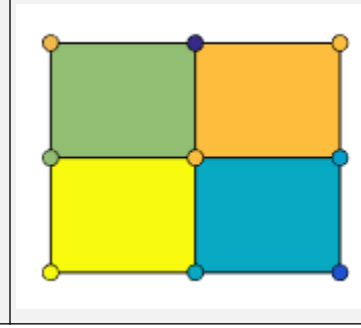
If you are using an earlier release, use the `get` and `set` functions instead.

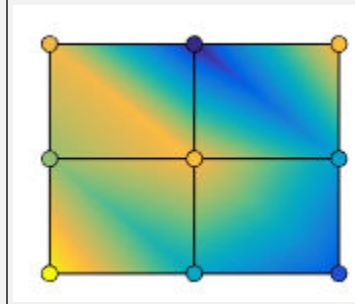
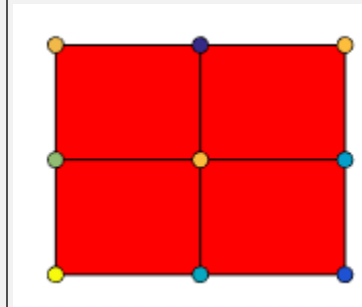
Properties

Faces

`FaceColor` — Face color
`'flat'` (default) | `'interp'` | `'none'` | `'texturemap'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Face color, specified as one of the values in this table.

Value	Description
`'flat'`	Use a different color for each face based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color value at the first vertex of each face (in the positive *x* and *y* directions) determines the color for the entire face. You cannot use this value when the `FaceAlpha` property is set to `'interp'`. 

Value	Description
`'interp'`	Use interpolated coloring for each face based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color varies across each face by interpolating the color values at the vertices. You cannot use this value when the `FaceAlpha` property is set to `'flat'`. 
RGB triplet, hexadecimal color code, or color name	Use the specified color for all the faces. This option does not use the color values in the `CData` property. 
`'texturemap'`	Transform the color data in `CData` so that it conforms to the surface.
`'none'`	Do not draw the faces.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

FaceAlpha — Face transparency
1 (default) | scalar in range `[0,1]` | `'flat'` | `'interp'` | `'texturemap'`

Face transparency, specified as one of these values:

- Scalar in range `[0,1]` — Use uniform transparency across all the faces. A value of `1` is fully opaque and `0` is completely transparent. Values between `0` and `1` are semitransparent. This option does not use the transparency values in the `AlphaData` property.
- `'flat'` — Use a different transparency for each face based on the values in the `AlphaData` property. The transparency value at the first vertex determines the transparency for the entire face. First you must specify the `AlphaData` property as a matrix the same size as the `ZData` property. The `FaceColor` property also must be set to `'flat'`.
- `'interp'` — Use interpolated transparency for each face based on the values in `AlphaData` property. The transparency varies across each face by interpolating the values at the vertices. First you must specify the `AlphaData` property as a matrix the same size as the `ZData` property. The `FaceColor` property also must be set to `'interp'`.
- `'texturemap'` — Transform the data in `AlphaData` so that it conforms to the surface.

FaceLighting — Effect of light objects on faces
`'flat'` (default) | `'gouraud'` | `'none'`

Effect of light objects on faces, specified as one of these values:

- `'flat'` — Apply light uniformly across each face. Use this value to view faceted objects.
- `'gouraud'` — Vary the light across the faces. Calculate the light at the vertices and then linearly interpolate the light across the faces. Use this value to view curved surfaces.
- `'none'` — Do not apply light from light objects to the faces.

To add a light object to the axes, use the `light` function.

Note The `'phong'` value has been removed. Use `'gouraud'` instead.

BackFaceLighting — Face lighting when normals point away from camera

`'reverselit'` (default) | `'unlit'` | `'lit'`

Face lighting when the vertex normals point away from camera, specified as one of these values:

- `'reverselit'` — Light the face as if the vertex normal pointed towards the camera.
- `'unlit'` — Do not light the face.
- `'lit'` — Light the face according to the vertex normal.

Use this property to discriminate between the internal and external surfaces of an object. For an example, see “Back Face Lighting”.

Edges

MeshStyle — Edges to display

`'both'` (default) | `'row'` | `'column'`

Edges to display, specified as `'both'`, `'row'`, or `'column'`.

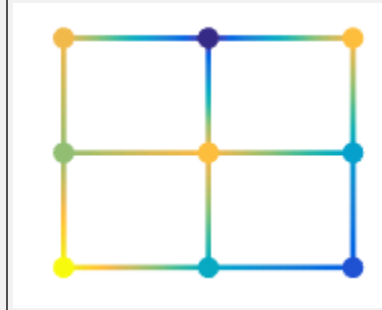
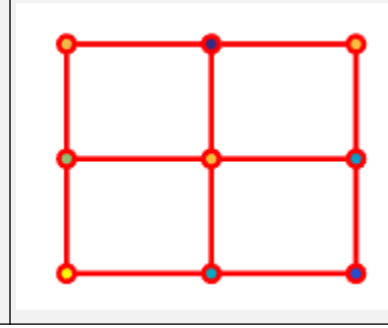
EdgeColor — Edge line color

`[0 0 0]` (default) | `'none'` | `'flat'` | `'interp'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Edge line color, specified as one of the values listed here. The default color of `[0 0 0]` corresponds to black edges.

Value	Description
`'none'`	Do not draw the edges.

Value	Description
`'flat'`	Use a different color for each edge based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color value at the first vertex of each face (in the positive *x* and *y* directions) determines the color for the adjacent edges. You cannot use this value when the `EdgeAlpha` property is set to `'interp'`.

Value	Description
`'interp'`	Use interpolated coloring for each edge based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color varies across each edge by linearly interpolating the color values at the vertices. You cannot use this value when the `EdgeAlpha` property is set to `'flat'`. 
RGB triplet, hexadecimal color code, or color name	Use the specified color for all the edges. This option does not use the color values in the `CData` property. 

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`EdgeAlpha` — Edge transparency

1 (default) | scalar value in range `[0,1]` | `'flat'` | `'interp'`

Edge transparency, specified as one of these values:

- Scalar in range `[0,1]` — Use uniform transparency across all of the edges. A value of `1` is fully opaque and `0` is completely transparent. Values between `0` and `1` are semitransparent. This option does not use the transparency values in the `AlphaData` property.
- `'flat'` — Use a different transparency for each edge based on the values in the `AlphaData` property. First you must specify the `AlphaData` property as a matrix the same size as the `ZData` property. The transparency value at the first vertex determines the transparency for the entire edge. The `EdgeColor` property also must be set to `'flat'`.
- `'interp'` — Use interpolated transparency for each edge based on the values in `AlphaData` property. First you must specify the `AlphaData` property as a matrix the same size as the `ZData` property. The transparency varies across each edge by interpolating the values at the vertices. The `EdgeColor` property also must be set to `'interp'`.

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

`AlignVertexCenters` — Sharp vertical and horizontal lines
`'off'` (default) | `'on'`

Sharp vertical and horizontal lines, specified as `'off'` or `'on'`.

If the associated figure has a `GraphicsSmoothing` property set to `'on'` and a `Renderer` property set to `'opengl'`, then the figure applies a smoothing technique to plots. In some cases, this smoothing technique can cause vertical and horizontal lines to appear uneven in thickness or color. Use the `AlignVertexCenters` property to eliminate the uneven appearance.

- `'off'` — Do not sharpen vertical or horizontal lines. The lines might appear uneven in thickness or color.
- `'on'` — Sharpen vertical and horizontal lines to eliminate an uneven appearance.

Note You must have a graphics card that supports this feature. To see if the feature is supported, call the `rendererinfo` function. If it is supported, `rendererinfo` returns value of `1` for `info.Details.SupportsAlignVertexCenters`.

`EdgeLighting` — Effect of light objects on edges

`'none'` (default) | `'flat'` | `'gouraud'`

Effect of light objects on edges, specified as one of these values:

- `'flat'` — Apply light uniformly across the each edges.
- `'none'` — Do not apply lights from light objects to the edges.
- `'gouraud'` — Calculate the light at the vertices, and then linearly interpolate across the edges.

Note The `'phong'` value has been removed. Use `'gouraud'` instead.

Markers

`Marker` — Marker symbol

`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | ...

Marker symbol, specified as one of the values listed in this table. By default, the object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerSize` — Marker size
6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

`MarkerEdgeColor` — Marker outline color
`'auto'` (default) | `'flat'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'`

Marker outline color, specified as `'auto'`, `'flat'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` option uses the same color as the `EdgeColor` property. The `'flat'` option uses the `CData` value at the vertex to set the color.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to

F. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | `'flat'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, `'flat'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` option uses the same color as the `Color` property for the axes. The `'flat'` option uses the `CData` value of the vertex to set the color.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

This property affects only the circle, square, diamond, pentagram, hexagram, and the four triangle marker types.

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Example: `'#D2F9A7'`

Coordinate Data

XData — *x*-coordinate data

vector or matrix

x-coordinate data specified as a matrix that is the same size as `ZData` or a vector of `length(n)`, where `[m,n] = size(ZData)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

XDataMode — Selection mode for XData

`'auto'` | `'manual'`

Selection mode for `XData`, specified as one of these values:

- `'auto'` — Use the column indices of `ZData`.
- `'manual'` — Use manually specified value. To specify the value, pass an input argument to the plotting function or directly set the `XData` property.

YData — *y*-coordinate data

vector or matrix

y-coordinate data specified as a matrix that is the same size as `ZData` or a vector of `length(m)`, where `[m,n] = size(ZData)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`YDataMode` — Selection mode for `YData`
`'auto'` | `'manual'`

Selection mode for `YData`, specified as one of these values:

- `'auto'` — Use the row indices of `ZData`.
- `'manual'` — Use manually specified value. To specify the value, pass an input argument to the plotting function or directly set the `YData` property.

`ZData` — *z*-coordinate data
matrix

z-coordinate data specified as a matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

Color and Transparency Data

`CData` — Vertex colors
2-D or 3-D array

Vertex colors, specified in one of these forms:

- 2-D array — Use colormap colors. Specify a color for each vertex by setting `CData` to an array the same size as `ZData`. The `CDataMapping` property determines how these values map into the current colormap. If the `FaceColor` property is set to `'texturemap'`, then `CData` does not need to be the same size as `ZData`. However, it must be of type `double` or `uint8`. The `CData` values map to conform to the surface defined by `ZData`.
- 3-D array — Use true colors. Specify an RGB triplet color for each vertex by setting `CData` to an m-by-n-by-3 array where `[m,n] = size(ZData)`. An RGB triplet is a three-element vector that specifies the intensities of the red, green, and blue components of a color. The first page of the array contains the red components, the second the green components, and the third the blue components of the colors. Since the surface uses true colors instead of colormap colors, the `CDataMapping` property has no effect.

- If `CData` is of type `double` or `single`, then an RGB triplet value of `[0 0 0]` corresponds to black and `[1 1 1]` corresponds to white.
- If `CData` is an integer type, then the surface uses the full range of data to determine the color. For example, if `CData` is of type `uint8`, then `[0 0 0]` corresponds to black and `[255 255 255]` corresponds to white. If `CData` is of type `int8`, then `[-128 -128 -128]` corresponds to black and `[127 127 127]` corresponds to white.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`CDataMode` — Selection mode for `CData`
`'auto'` (default) | `'manual'`

Selection mode for `CData`, specified as one of these values:

- `'auto'` — Use the `ZData` values to set the colors.
- `'manual'` — Use manually specified values. To specify the values, set the `CData` property.

`CDataMapping` — Direct or scaled colormapping
`'scaled'` (default) | `'direct'`

Direct or scaled colormapping, specified as one of these values:

- `scaled` — Transform the color data to span the portion of the colormap indicated by the axes `CLim` property, linearly mapping data values to colors. See the `caxis` reference page for more information on this mapping.
- `direct` — Use the color data as indices directly into the colormap. The color data should then be integer values ranging from 1 to `length(colormap)`. MATLAB maps values less than 1 to the first color in the colormap, and values greater than `length(colormap)` to the last color in the colormap. Values with a decimal portion are fixed to the nearest lower integer.

`AlphaData` — Transparency data
1 (default) | array same size as `ZData`

Transparency data for each vertex, specified as an array the same size as the `ZData` property. After specifying the values, set the `FaceAlpha` and `EdgeAlpha` properties to control the type of transparency. If the `FaceAlpha` and `EdgeAlpha` properties are both set to scalar values, then the surface does not use the `AlphaData` values.

The `AlphaDataMapping` property determines how the surface interprets the `AlphaData` property values.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`AlphaDataMapping` — Interpretation of `AlphaData` values

`'scaled'` (default) | `'direct'` | `'none'`

Interpretation of `AlphaData` values, specified as one of these values:

- `'none'` — Interpret the values as transparency values. A value of 1 or greater is completely opaque, a value of 0 or less is completely transparent, and a value between 0 and 1 is semitransparent.
- `'scaled'` — Map the values into the figure's alphamap. The minimum and maximum alpha limits of the axes determine the `AlphaData` values that map to the first and last elements in the alphamap, respectively. For example, if the alpha limits are `[3 5]`, then values of `3` or less map to the first element in the alphamap. Values of `5` or greater map to the last element in the alphamap. The `ALim` property of the axes contains the alpha limits. The `Alphamap` property of the figure contains the alphamap.
- `'direct'` — Interpret the values as indices into the figure's alphamap. Values with a decimal portion are fixed to the nearest lower integer.
  - If the values are of type `double` or `single`, then values of 1 or less map to the first element in the alphamap. Values equal to or greater than the length of the alphamap map to the last element in the alphamap.
  - If the values are of integer type, then values of 0 or less map to the first element in the alphamap. Values equal to or greater than the length of the alphamap map to the last element in the alphamap (or up to the range limits of the type). The integer types are `uint8`, `uint16`, `uint32`, `uint64` , `int8`, `int16`, `int32`, and `int64`.
  - If the values are of type `logical`, then values of 0 map to the first element in the alphamap and values of 1 map to the second element in the alphamap.

Normals

`VertexNormals` — Normal vectors for each surface vertex

`[]` (default) | m-by-n-by-3 array

Normal vectors for each surface vertex, specified as a m-by-n-by-3 array, where `[m,n] = size(ZData)`. Specify one normal vector per vertex.

Specifying values for this property sets the associated mode to manual. If you do not specify normal vectors, then the surface generates this data for lighting calculations.

Data Types: `single` | `double`

`VertexNormalsMode` — Selection mode for `VertexNormals`
`'auto'` (default) | `'manual'`

Selection mode for `VertexNormals`, specified as one of these values:

- `'auto'` — Calculate the normal vectors based on the coordinate data.
- `'manual'` — Use manually specified values. To specify the values, set the `VertexNormals` property.

`FaceNormals` — Normal vectors for each surface face
`[]` (default) | (m-1)-by-(n-1)-by-3 array

Normal vectors for each surface face, specified as a (m-1)-by-(n-1)-by-3 array, where `[m,n] = size(ZData)`. Specify one normal vector per face.

Specifying values for this property sets the associated mode to manual. If you do not specify normal vectors, then the surface generates this data for lighting calculations.

Data Types: `single` | `double`

`FaceNormalsMode` — Selection mode for `FaceNormals`
`'auto'` (default) | `'manual'`

Selection mode for `FaceNormals`, specified as one of these values:

- `'auto'` — Calculate the normal vectors based on the coordinate data.
- `'manual'` — Use manually specified values. To specify the values, set the `FaceNormals` property.

Lighting

`AmbientStrength` — Strength of ambient light
`0.3` (default) | scalar in range `[0,1]`

Strength of ambient light, specified as a scalar value in the range `[0,1]`. Ambient light is a nondirectional light that illuminates the entire scene. There must be at least one visible light object in the axes for the ambient light to be visible.

The `AmbientLightColor` property for the axes sets the color of the ambient light. The color is the same for all objects in the axes.

Example: `0.5`

Data Types: `double`

`DiffuseStrength` — Strength of diffuse light
`0.6` (default) | scalar in range `[0,1]`

Strength of diffuse light, specified as a scalar value in the range `[0,1]`. Diffuse light is the nonspecular reflectance from light objects in the axes.

Example: `0.3`

Data Types: `double`

`SpecularStrength` — Strength of specular reflection
`0.9` (default) | scalar in range `[0,1]`

Strength of specular reflection, specified as a scalar value in the range `[0,1]`. Specular reflections are the bright spots on the surface from light objects in the axes.

Example: `0.3`

Data Types: `double`

`SpecularExponent` — Size of specular spot
10 (default) | scalar greater than or equal to 1

Size of specular spot, specified as a scalar value greater than or equal to 1. Most materials have exponents in the range `[5 20]`.

Example: 7

Data Types: `double`

`SpecularColorReflectance` — Color of specular reflections
1 (default) | scalar in range `[0,1]`

Color of specular reflections, specified as a scalar value in the range `[0,1]`. A value of `1` sets the color using only the color of the light source. A value of `0` sets the color using both the color of the object from which it reflects and the color of the light source. The `Color` property of the light contains the color of the light source. The proportions vary linearly for values in between.

Example: `0.5`

Data Types: `double`

Legend

`DisplayName` — Legend label

`''` (default) | character vector | string scalar

Legend label, specified as a character vector or string scalar. The legend does not display until you call the `legend` command. If you do not specify the text, then `legend` sets the label using the form `'dataN'`.

`Annotation` — Control for including or excluding object from legend

`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`DataTipTemplate` — Data tip content

`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

This property applies only to surfaces with pinned data tips.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

`Clipping` — Clipping of object to axes limits

`'on'` (default) | `'off'`

Clipping of the object to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the object that are outside the axes limits.
- `'off'` — Display the entire object, even if parts of it appear outside the axes limits. Parts of the object might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the object so that it is larger than the original plot.

The `Clipping` property of the axes that contains the object must be set to `'on'`. Otherwise, this property has no effect. For more information about the clipping behavior, see the `Clipping` property of the axes.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.

- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.

- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback.

MATLAB resumes executing the running callback when the interrupting callback completes.

- If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.

- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks when visible. The `Visible` property must be set to `'on'` and you must click a part of the `Surface` object that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The `HitTest` property determines if the `Surface` object responds to the click or if an ancestor does.
- `'all'` — Capture mouse clicks regardless of visibility. The `Visible` property can be set to `'on'` or `'off'` and you can click a part of the `Surface` object that has no color. The `HitTest` property determines if the `Surface` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `Surface` object passes the click through it to the object below it in the current view of the figure window. The `HitTest` property has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `Surface` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `Surface` object that has one of these:
  - `HitTest` property set to `'on'`
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

---

Note The `PickableParts` property determines if the `Surface` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent

`Axes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `Group`, or `Transform` object.

`Children` — Children

empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the chart object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object
`'surface'`

This property is read-only.

Type of graphics object, returned as `'surface'`

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`pcolor` | `surface`

Topics

"Access Property Values"
"Graphics Object Properties"

Introduced before R2006a

save

Save workspace variables to file

Syntax

```
save(filename)
save(filename,variables)
save(filename,variables,fmt)

save(filename,variables,version)
save(filename,variables,version,'-nocompression')
save(filename,variables,'-append')
save(filename,variables,'-append','-nocompression')

save filename
```

Description

`save(filename)` saves all variables from the current workspace in a MATLAB formatted binary file (MAT-file) called `filename`. If `filename` exists, `save` overwrites the file.

`save(filename,variables)` saves only the variables or fields of a structure array specified by `variables`.

`save(filename,variables,fmt)` saves in the file format specified by `fmt`. The `variables` argument is optional. If you do not specify `variables`, the `save` function saves all variables in the workspace.

`save(filename,variables,version)` saves to the MAT-file version specified by `version`. The `variables` argument is optional.

`save(filename,variables,version,'-nocompression')` saves the variables to the MAT-file without compression. The `'-nocompression'` flag only supports MAT-file Version 7 (default) and Version 7.3. Therefore, you must specify `version` as `'-v7'` or `'-v7.3'`. The `variables` argument is optional.

`save(filename,variables,'-append')` adds new variables to an existing file. If a variable already exists in a MAT-file, then `save` overwrites it with the value in the workspace.

For ASCII files, `'-append'` adds data to the end of the file.

To append to a Version 6 MAT-file, you must also include `'-v6'` as an input argument.

`save(filename,variables,'-append','-nocompression')` adds new variables to an existing file without compression. The existing file must be a MAT-file Version 7 (default) or 7.3.

`save filename` is the command form of the syntax. Command form requires fewer special characters. You do not need to type parentheses or enclose the input in single or double quotes. Separate inputs with spaces instead of commas.

For example, to save a file named `test.mat`, these statements are equivalent:

```
save test.mat      % command form
save('test.mat')   % function form
```

You can include any of the inputs described in previous syntaxes. For example, to save the variable named `X`:

```
save test.mat X       % command form
save('test.mat','X')  % function form
```

Do not use command form when any of the inputs, such as `filename`, are variables or strings.

Examples

Save All Workspace Variables to MAT-File

Save all variables from the workspace in a binary MAT-file, `test.mat`. If `filename` is a variable, use function syntax.

```
filename = 'test.mat';
save(filename)
```

Otherwise, you also can use command syntax.

```
save test.mat
```

Remove the variables from the workspace, and then retrieve the data with the `load` function.

```
clear
load('test.mat')
```

Save Specific Variables to MAT-File

Create and save two variables, `p` and `q`, to a file called `pqfile.mat`.

```
p = rand(1,10);
q = ones(10);
save('pqfile.mat','p','q')
```

MATLAB® saves the variables to the file, `pqfile.mat`, in the current folder.

You also can use command syntax to save the variables, `p` and `q`.

```
save pqfile.mat p q
```

Save Data to ASCII File

Create two variables, save them to an ASCII file, and then view the contents of the file.

```
p = rand(1,10);
q = ones(10);
save('pqfile.txt','p','q','-ascii')
type('pqfile.txt')
```

The `type` function displays the contents of the file.

Alternatively, use command syntax for the `save` operation.

```
save pqfile.txt p q -ascii
```

Save Structure Fields as Individual Variables

Create a structure, `s1`, that contains three fields, `a`, `b`, and `c`.

```
s1.a = 12.7;
s1.b = {'abc',[4 5; 6 7]};
s1.c = 'Hello!';
```

Save the fields of structure `s1` as individual variables in a file called `newstruct.mat`.

```
save('newstruct.mat','-struct','s1');
```

Check the contents of the file using the `whos` function.

```
disp('Contents of newstruct.mat:')
```

```
Contents of newstruct.mat:
```

```
whos('-file','newstruct.mat')
```

```
  Name      Size            Bytes  Class     Attributes

  a         1x1                 8  double
  b         1x2               262  cell
  c         1x6                12  char
```

Save Variables to Version 7.3 MAT-File

Create two variables and save them to a Version 7.3 MAT-file called `example.mat`.

```
A = rand(5);
B = magic(10);
save('example.mat','A','B','-v7.3')
```

You also can use command syntax for the `save` operation.

```
save example.mat A B -v7.3
```

Save Variables to MAT-File Without Compression

Create two variables and save them, without compression, to a Version 7 or 7.3 MAT-file called myFile.mat.

```
A = rand(5);
B = magic(10);
save('myFile.mat','A','B','-v7.3','-nocompression')
```

Alternatively, use the command syntax for the save operation.

```
save myFile.mat A B -v7.3 -nocompression
```

The '-nocompression' flag facilitates a faster save for those variables that are larger than 2 GB or those that do not benefit from compression.

Append Variable to MAT-File

Save two variables to a MAT-file. Then, append a third variable to the same file.

```
p = rand(1,10);
q = ones(10);
save('test.mat','p','q')
```

View the contents of the MAT-file.

```
whos('-file','test.mat')
```

```
  Name       Size            Bytes  Class     Attributes

  p          1x10               80  double
  q         10x10              800  double
```

Create a new variable, a, and append it to the MAT-file.

```
a = 50;
save('test.mat','a','-append')
```

View the contents of the MAT-file.

```
whos('-file','test.mat')
```

```
  Name       Size            Bytes  Class     Attributes
```

```
  a          1x1                 8  double
  p          1x10               80  double
  q         10x10              800  double
```

The variable, `a`, is appended to `test.mat`, without overwriting the previous variables, `p` and `q`.

Note To append to a Version 6 MAT-file, specify both `'-v6'` and `'-append'`. For example, to save variable `a` to the file, `test.mat`, call:

```
save('test.mat','a','-v6','-append')
```

Append Variable to MAT-File Without Compression

Save two variables to a MAT-file. Then, append a third variable, without compression, to the same file.

Create two variables `A` and `B` and save them to a MAT-file Version `7` or `7.3`. By default, the `save` function compresses variables `A` and `B` before saving them to `myFile.mat`.

```
A = rand(5);
B = magic(10);
save('myFile.mat','A','B','-v7.3')
```

View the contents of the MAT-file.

```
whos('-file','myFile.mat')
```

```
  Name       Size            Bytes  Class     Attributes

  A          5x5               200  double
  B         10x10              800  double
```

Create a new variable `C` and append it, without compression, to `myFile.mat`.

```
C = 5;
save('myFile.mat','C','-append','-nocompression')
```

View the contents of the MAT-file.

```
whos('-file','myFile.mat')

  Name       Size            Bytes  Class     Attributes

  A          5x5               200  double
  B         10x10              800  double
  C          1x1                 8  double
```

Input Arguments

filename — Name of file
'matlab.mat' (default) | character vector | string scalar

Name of file, specified as a character vector or string scalar. If you do not specify `filename`, the `save` function saves to a file named `matlab.mat`.

If `filename` has no extension (that is, no period followed by text), and the value of `format` is not specified, then MATLAB appends `.mat`. If `filename` does not include a full path, MATLAB saves to the current folder. You must have permission to write to the file.

When using the command form of `save`, it is unnecessary to enclose the input in single quotes. However, if `filename` contains a space, you must enclose the argument in single quotes. For example, `save 'filename withspace.mat'`.

Note Do not use command form when `filename` is a string.

Example: `'myFile.mat'`

variables — Names of variables to save
character vector | string scalar

Names of variables to save, specified as character vectors or string scalars. When using the command form of `save`, you do not need to enclose the input in single quotes.

Note Do not use command form when `variables` is a string.

`variables` can be in one of the following forms.

Form of variables Input	Variables to Save
var1,...,varN	Save the listed variables, specified as individual character vectors or strings. Use the `'*'` wildcard to match patterns. For example, `save('filename.mat','A*')` saves all variables in the file that start with `A`.
`'-regexp'`,*expr1,...,exprN*	Save only the variables whose names match the regular expressions, specified as character vectors or strings. For example, `save('filename.mat','-regexp','^Mon','^Tues')` saves only the variables in the file whose names begin with `Mon` or `Tues`.
`'-struct'`,*structName*	Store the fields of the scalar structure specified by *structName* as individual variables in the file. For example, `save('filename.mat','-struct','S')` saves the scalar structure, `S`.
`'-struct'`,*structName*,*field1,...,fieldN*	Store the specified fields of the specified scalar structure as individual variables in the file. For example, `save('filename.mat','-struct','S','a','b')` saves the fields `S.a` and `S.b`.
`'-struct'`,*structName*,`'-regexp'`,*expr1,...,exprN*	Store only the fields whose names match the regular expressions, specified as character vectors or strings.

`fmt` — File format

`'-mat'` (default) | `'-ascii'` | `'-ascii','-tabs'` | `'-ascii','-double'` | `'-ascii','-double','-tabs'`

File format, specified as one of the following. When using the command form of `save`, you do not need to enclose the input in single or double quotes, for example, `save myFile.txt -ascii -tabs`.

Value of `fmt`	File Format
`'-mat'`	Binary MAT-file format.
`'-ascii'`	Text format with 8 digits of precision.

Value of `fmt`	File Format
`'-ascii','-tabs'`	Tab-delimited text format with 8 digits of precision.
`'-ascii','-double'`	Text format with 16 digits of precision.
`'-ascii','-double','-tabs'`	Tab-delimited text format with 16 digits of precision.

For MAT-files, data saved on one machine and loaded on another machine retains as much accuracy and range as the different machine floating-point formats allow.

Use one of the text formats to save MATLAB numeric values to text files. In this case:

- Each variable must be a two-dimensional `double` array.
- The output includes only the real component of complex numbers.
- MATLAB writes data from each variable sequentially to the file. If you plan to use the `load` function to read the file, all variables must have the same number of columns. The `load` function creates a single variable from the file.

If you specify a text format and any variable is a two-dimensional character array, then MATLAB translates characters to their corresponding internal ASCII codes. For example, `'abc'` appears in a text file as:

```
  9.7000000e+001  9.8000000e+001  9.9000000e+001
```

Data Types: `char` | `string`

`version` — MAT-file version
`'-v7.3'` | `'-v7'` | `'-v6'` | `'-v4'`

MAT-file version, specified as one of the following. When using the command form of `save`, you do not need to enclose the input in single or double quotes.

Value of `version`	Loads in MATLAB Versions	Supported Features	Compression	Maximum Size of Each Variable
`'-v7.3'`	7.3 (R2006b) or later	Saving and loading parts of variables, and all Version 7 features. Version 7.3 also supports saving variables without compression using the `'-nocompression'` option.	Yes (default)	≥ 2 GB on 64-bit computers
`'-v7'`	7.0 (R14) or later	Unicode character encoding, which enables file sharing between systems that use different default character encoding schemes, and all Version 6 features. Version 7 also supports saving variables without compression using the `'-nocompression'` option.	Yes	2^31 bytes per variable
`'-v6'`	5 (R8) or later	N-dimensional arrays, cell arrays, structure arrays, variable names longer than 19 characters, and all Version 4 features.	No	2^31 bytes per variable
`'-v4'`	All	Two-dimensional `double`, character, and sparse arrays.	No	100,000,000 elements per array, and 2^31 bytes per variable

If any data items require features that the specified version does not support, MATLAB does not save those items and issues a warning. You cannot specify a version later than your current version of MATLAB software.

Note Version 7.3 MAT-files use an HDF5 based format that requires some overhead storage to describe the contents of the file. For cell arrays, structure arrays, or other containers that can store heterogeneous data types, Version 7.3 MAT-files are sometimes larger than Version 7 MAT-files.

To view or set the default version for MAT-files, go to the **Home** tab and in the **Environment** section, click **Preferences**. Select **MATLAB** > **General** > **MAT-Files** and then choose a **MAT-file save format** option.

Data Types: `char` | `string`

Tips

- For more flexibility in creating ASCII files, use `dlmwrite` or `fprintf`.
- Saving graphics objects with the `save` function can result in a large file since the file contains all the information required to regenerate the object.
- Avoid saving figures with the `save` function. Use the `savefig` function instead. Using `save` to save a figure in R2014b or later makes MAT-file inaccessible in earlier versions of MATLAB. If you use `save` to save a figure, then the function displays a warning message. Delete any figures before using `save`. Keep in mind that the figures might not be directly in your workspace. For example, they might be stored in a structure or in the workspace of a callback function.

See Also

`clear` | `hgsave` | `load` | `matfile` | `regexp` | `saveas` | `whos`

Topics

"Create and Edit Variables"
"Save and Load Workspace Variables"
"Write Data to Text Files"
"Regular Expressions"

Introduced before R2006a

saveas

Save figure to specific file format

Syntax

```
saveas(fig,filename)
saveas(fig,filename,formattype)
```

Description

`saveas(fig,filename)` saves the figure or Simulink block diagram specified by `fig` to file `filename`. Specify the file name as a character vector or string that includes a file extension, for example, `'myplot.jpg'`. The file extension defines the file format. If you do not specify an extension, then `saveas` saves the figure to a FIG-file. To save the current figure, specify `fig` as `gcf`.

`saveas(fig,filename,formattype)` creates the file using the specified file format, `formattype`. If you do not specify a file extension in the file name, for example, `'myplot'`, then the standard extension corresponding to the specified format automatically appends to the file name. If you specify a file extension, it does not have to match the format. `saveas` uses `formattype` for the format, but saves the file with the specified extension. Thus, the file extension might not match the actual format used.

Examples

Save Figure as PNG File

Create a bar chart and save it as a PNG file.

```
x = [2 4 7 2 4 5 2 5 1 4];
bar(x);
saveas(gcf,'Barchart.png')
```

Save Figure as EPS File

Create a bar chart and save it as an EPS file. Specify the `'epsc'` driver to save it in color.

```
x = [2 4 7 2 4 5 2 5 1 4];
bar(x);
saveas(gcf,'Barchart','epsc')
```

`saveas` saves the bar chart as `Barchart.eps`.

Save Simulink Block Diagram as BMP File

Save a Simulink block diagram named `'sldemo_tank'` as a BMP file. Use `get_param` to get the handle of the diagram. You must have Simulink installed to run this code.

```
sldemo_tank
fig = get_param('sldemo_tank','Handle');
saveas(fig,'MySimulinkDiagram.bmp');
```

Input Arguments

`fig` — Figure to save
figure object | Simulink block diagram

Figure to save, specified as a figure object or a Simulink block diagram. If you specify other types of graphics objects, such as an axes, then `saveas` saves the parent figure to the object. This means that `saveas` cannot save a subplot without also saving all subplots in the parent figure.

Example: `saveas(gcf,'MyFigure.png')`

To save a Simulink block diagram, use `get_param` to get the handle of the diagram. For example, save a block diagram named `'sldemo_tank'`.

```
sldemo_tank
saveas(get_param('sldemo_tank','Handle'),'MySimulinkDiagram.bmp');
```

`filename` — File name
character vector | string

File name, specified as a character vector or string with or without a file extension.

Example: `'Bar Chart'`

Example: `'Bar Chart.png'`

If you specify a file extension, then `saveas` uses the associated format. If you specify a file extension and additionally specify the `formattype` input argument, then `saveas` uses `formattype` for the format and saves the file with the specified file name. Thus, the file extension might not match the actual format used.

You can specify any extension corresponding to a file format. This table lists some common file extensions.

Extension	Resulting Format
`.fig`	MATLAB FIG-file (invalid for Simulink block diagrams)
`.m`	MATLAB FIG-file and MATLAB code that opens figure (invalid for Simulink block diagrams)
`.jpg`	JPEG image
`.png`	Portable Network Graphics
`.eps`	EPS Level 3 Black and White
`.pdf`	Portable Document Format
`.bmp`	Windows bitmap
`.emf`	Enhanced metafile
`.pbm`	Portable bitmap
`.pcx`	Paintbrush 24-bit
`.pgm`	Portable Graymap
`.ppm`	Portable Pixmap
`.tif`	TIFF image, compressed

Data Types: `char` | `string`

`formattype` — File format

`'fig'` | `'m'` | `'mfig'` | bitmap image file format | vector graphics file format

File format, specified as one of these options:

- `'fig'` — Save the figure as a MATLAB figure file with the `.fig` extension. To open figures saved with the `.fig` extension, use the `openfig` function. This format is not valid for Simulink block diagrams.
- `'m'` or `'mfig'` — Save the figure as a MATLAB figure file and additionally create a MATLAB file that opens the figure. To open the figure, run the MATLAB file. This option is not valid for Simulink block diagrams.
- Bitmap image file format — Specify the format as one of the bitmap image options in the table, Bitmap Image Formats.
- Vector graphics file format — Specify the format as one of the vector graphics options in the table, Vector Graphics Formats.

Bitmap Image File

Bitmap images contain a pixel-based representation of the figure. The size of the generated file depends on the figure, the format, and your system resolution. Bitmap images are widely used by web browsers and other applications that display graphics. However, they do not support transparency or scale well and you cannot modify individual graphics objects (such as lines and text) in other graphics applications.

Bitmap Image Formats

Option	Format	Default File Extension
`'jpeg'`	JPEG 24-bit	`.jpg`
`'png'`	PNG 24-bit	`.png`
`'tiff'`	TIFF 24-bit (compressed)	`.tif`
`'tiffn'`	TIFF 24-bit (not compressed)	`.tif`
`'meta'`	Enhanced metafile (Windows only)	`.emf`
`'bmpmono'`	BMP Monochrome	`.bmp`
`'bmp'`	BMP 24-bit	`.bmp`
`'bmp16m'`	BMP 24-bit	`.bmp`
`'bmp256'`	BMP 8-bit (256 color, uses a fixed colormap)	`.bmp`
`'hdf'`	HDF 24-bit	`.hdf`
`'pbm'`	PBM (plain format) 1-bit	`.pbm`
`'pbmraw'`	PBM (raw format) 1-bit	`.pbm`
`'pcxmono'`	PCX 1-bit	`.pcx`
`'pcx24b'`	PCX 24-bit color (three 8-bit planes)	`.pcx`
`'pcx256'`	PCX 8-bit newer color (256 color)	`.pcx`
`'pcx16'`	PCX older color (EGA/VGA 16-color)	`.pcx`
`'pgm'`	PGM (plain format)	`.pgm`
`'pgmraw'`	PGM (raw format)	`.pgm`
`'ppm'`	PPM (plain format)	`.ppm`
`'ppmraw'`	PPM (raw format)	`.ppm`

Vector Graphics File

Vector graphics files store commands that redraw the figure. This type of format scales well, but can result in a large file. In some cases, a vector graphics format might not produce the correct 3-D arrangement of objects. Some applications support extensive editing of vector graphics formats, but others do not support editing beyond resizing the graphic. The best practice is to make all the necessary changes while your figure is still in MATLAB.

Typically, `saveas` uses the Painters renderer when generating vector graphics files. For some complex figures, `saveas` uses the OpenGL renderer instead. If it uses the OpenGL renderer, then the vector graphics file contains an embedded image, which might limit the extent to which you can edit the image in other applications. Also, if `saveas` uses the OpenGL renderer to generate the file, then transparency is not supported. To ensure that `saveas` uses the Painters renderer, set the `Renderer` property for the figure to `'painters'`.

If you set the `Renderer` property for the figure, then `saveas` uses that renderer. Otherwise, it chooses the appropriate renderer. However, if `saveas` chooses a renderer that differs from the renderer used for the figure on the display, then some details of the saved figure can differ from the displayed figure. If necessary, you can make the displayed figure and the saved figure use the same renderer by setting the `Renderer` property for the figure.

Vector Graphics Formats

Option	Format	Default File Extension
`'pdf'`	Full page Portable Document Format (PDF) color	`.pdf`
`'eps'`	Encapsulated PostScript (EPS) Level 3 black and white	`.eps`
`'epsc'`	Encapsulated PostScript (EPS) Level 3 color	`.eps`
`'eps2'`	Encapsulated PostScript (EPS) Level 2 black and white	`.eps`
`'epsc2'`	Encapsulated PostScript (EPS) Level 2 color	`.eps`
`'meta'`	Enhanced Metafile (Windows only)	`.emf`
`'svg'`	SVG (scalable vector graphics)	`.svg`
`'ps'`	Full-page PostScript (PS) Level 3 black and white	`.ps`
`'psc'`	Full-page PostScript (PS) Level 3 color	`.ps`
`'ps2'`	Full-page PostScript (PS) Level 2 black and white	`.ps`
`'psc2'`	Full-page PostScript (PS) Level 2 color	`.ps`

Note Only PDF and PS formats use the `PaperOrientation` property of the figure and the `left` and `bottom` elements of the `PaperPosition` property. Other formats ignore these values.

More About

Compatibility Considerations

Starting in R2016a, saved figures match the size of the figure on the screen by default. Previously, saved figures were 8-by-6 inches by default.

Tips

- To control the size or resolution when you save a figure, use the `print` function instead.
- The `saveas` function and the **Save As** dialog box (accessed from the **File** menu) do not produce identical results. The **Save As** dialog box produces images at screen resolution and at screen size. The `saveas` function uses a resolution of 150 DPI and uses the `PaperPosition` and `PaperPositionMode` properties of the figure to determine the size of the image.
- Details of saved and printed figures can differ from the figure on the display. To get output that is more consistent with the display, see “Save Figure Preserving Background Color” and “Save Figure at Specific Size and Resolution”.

See Also

`open` | `print` | `savefig`

Topics

“Save Plot as Image or Vector Graphics File”

Introduced before R2006a

savefig

Save figure and contents to FIG-file

Syntax

```
savefig(filename)
savefig(H,filename)
savefig(H,filename,'compact')
```

Description

`savefig(filename)` saves the current figure to a FIG-file named *`filename`*`.fig`.

`savefig(H,filename)` saves the figures identified by the graphics array `H` to a FIG-file named *`filename`*`.fig`.

`savefig(H,filename,'compact')` saves the specified figures in a FIG-file that can be opened only in MATLAB R2014b or later releases. The `'compact'` option reduces the size of the `.fig` file and the time required to create the file.

Examples

Save Current Figure to FIG-File

Create a surface plot of the `peaks` function. Save the figure to the file `PeaksFile.fig`.

```
figure
surf(peaks)
savefig('PeaksFile.fig')
```

To open the saved figure, use the command:

```
openfig('PeaksFile.fig');
```

MATLAB creates a new figure using the saved `.fig` file.

Save Multiple Figures to FIG-File

Create two plots and store the figure handles in array `h`. Save the figures to the file `TwoFiguresFile.fig`. Close the figures after saving them.

```
h(1) = figure;
z = peaks;
surf(z)

h(2) = figure;
plot(z)

savefig(h,'TwoFiguresFile.fig')
close(h)
```

To open the two figures, use the command:

```
figs = openfig('TwoFiguresFile.fig');
```

`figs` contains the handles of the two figures created.

Save Figure Using `'compact'` Option

Save a figure using the compact option:

```
h = figure
surf(peaks)
savefig(h,'PeaksFile.fig','compact')
```

To open the figure, use the command:

```
openfig('PeaksFile.fig');
```

Input Arguments

`H` — One or more figures
single figure | array of figures

One or more figures, specified as a single figure or an array of figures.

`filename` — File name
`'Untitled.fig'` (default) | character vector | string

File name, specified as a character vector or string. If you do not specify a file name, then MATLAB saves the file as `Untitled.fig`, which is the default.

If the specified file name does not include a `.fig` file extension, then MATLAB appends the extension. `savefig` does not accept other file extensions.

Example: `'ExampleFile.fig'`

Data Types: `char` | `string`

`'compact'` — File format for R2014b or later releases
`'compact'`

File format for R2014b or later releases of MATLAB, specified as `'compact'`. This option results in smaller `.fig` files. However, do not use the `'compact'` option if you want to open the `.fig` file in versions of MATLAB before R2014b.

Tips

- You must use MATLAB to open files saved using `savefig`. To open the file, pass the file name to the function `openfig` or `open`. For example,

  ```
  openfig('ExampleFile.fig')
  ```

  opens the file, `ExampleFile.fig`, in MATLAB.
- `savefig` saves the full MATLAB figure. To save only part of a figure, such as an axes, or to save handles in addition to the data, use the `save` function to create a MAT-file.

See Also

`findobj` | `load` | `open` | `openfig` | `save`

Topics

"Save Plot as Image or Vector Graphics File"

Introduced in R2013b

saveobj

Modify save process for object

Syntax

```
b = saveobj(a)
```

Description

`b = saveobj(a)` is called by the `save` function if the class of `a` defines a `saveobj` method. `save` writes the returned value, `b`, to the MAT-file.

Define a `loadobj` method to take the appropriate action when loading the object.

If `A` is an array of objects, MATLAB invokes `saveobj` separately for each object saved.

Examples

Call the superclass `saveobj` method from the subclass implementation of `saveobj` with the following syntax:

```
classdef mySub < super
   methods
      function sobj = saveobj(obj)
         % Call superclass saveobj method
         sobj = saveobj@super(obj);
         % Perform subclass save operations
         ...
      end
   ...
   end
...
end
```

Update object when saved:

```
function b = saveobj(a)
   % If the object does not have an account number,
   % Add account number to AccountNumber property
   if isempty(a.AccountNumber)
      a.AccountNumber = getAccountNumber(a);
   end
   b = a;
end
```

See Also

`load` | `loadobj` | `save`

Topics

"Save and Load Process for Objects"
"Object Save and Load"

Introduced before R2006a

savepath

Save current search path

Syntax

```
savepath
savepath folderName/pathdef.m
```

```
status = savepath( ___ )
```

Description

`savepath` saves the current MATLAB search path to an existing `pathdef.m` file in the current folder. If there is no `pathdef.m` file in the current folder, then `savepath` saves the search path to the first `pathdef.m` file on the current path. If there is no such file on the current path, then `savepath` saves the search path to the `pathdef.m` file that MATLAB located at startup.

On a Windows system with User Account Control (UAC) enabled, you might be prompted to allow the update operation because it requires administrator-level permission.

In MATLAB Online, changes to the path are automatically saved. Therefore, calling `savepath` is not necessary.

`savepath folderName/pathdef.m` saves the current search path to `pathdef.m` located in the folder specified by `folderName`. If you do not specify `folderName`, then `savepath` saves `pathdef.m` in the current folder.

Use this syntax if you do not have write access to the current `pathdef.m` file.

To automatically use the saved search path in a future session, specify `folderName` as the MATLAB startup folder.

`status = savepath( ___ )` additionally indicates if the operation is successful, using any of the input arguments in the previous syntaxes. The `status` output is `0` when `savepath` is successful, and `1` otherwise.

Examples

Save Search Path to Specific Folder

Save the current search path to `pathdef.m` located in the folder, `I:/my_matlab_files`.

```
savepath I:/my_matlab_files/pathdef.m
```

Input Arguments

`folderName` — Folder name
string array | character vector

Folder name, specified as a string array or character vector. `folderName` can be a relative or absolute path.

Example: `C:\myFolder`

Tips

- To display the paths to all `pathdef.m` files in the current folder and on the current search path, use `which`.

  ```
  which pathdef.m -all
  ```

  The `savepath` command updates the first `pathdef.m` file in this list.
- To save the search path programmatically each time you exit MATLAB, use `savepath` in a `finish.m` file.

See Also

`addpath` | `finish` | `path` | `rmpath` | `userpath`

Topics

"What Is the MATLAB Search Path?"
"Run Script When Exiting"

Introduced before R2006a

scatter

Scatter plot

Syntax

```
scatter(x,y)
scatter(x,y,sz)
scatter(x,y,sz,c)
scatter( ___ ,'filled')
scatter( ___ ,mkr)
scatter( ___ ,Name,Value)

scatter(ax, ___ )

s = scatter( ___ )
```

Description

`scatter(x,y)` creates a scatter plot with circles at the locations specified by the vectors `x` and `y`. This type of graph is also known as a bubble plot.

`scatter(x,y,sz)` specifies the circle sizes. To plot each circle with equal size, specify `sz` as a scalar. To plot each circle with a different size, specify `sz` as a vector with length equal to the length of `x` and `y`.

`scatter(x,y,sz,c)` specifies the circle colors. To plot all circles with the same color, specify `c` as a color name or an RGB triplet. To use varying color, specify `c` as a vector or a three-column matrix of RGB triplets.

`scatter( ___ ,'filled')` fills in the circles. Use the `'filled'` option with any of the input argument combinations in the previous syntaxes.

`scatter( ___ ,mkr)` specifies the marker type.

`scatter( ___ ,Name,Value)` modifies the scatter chart using one or more name-value pair arguments. For example, `'LineWidth',2` sets the marker outline width to 2 points.

`scatter(ax, ___ )` plots into the axes specified by `ax` instead of into the current axes. The option `ax` can precede any of the input argument combinations in the previous syntaxes.

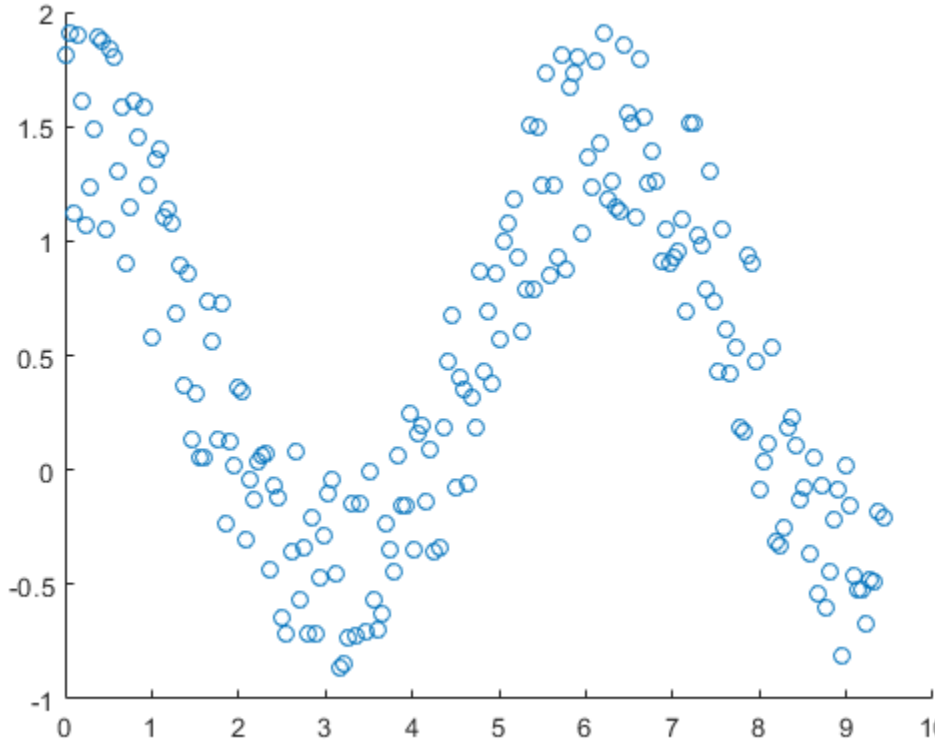
`s = scatter( ___ )` returns the `Scatter` object. Use `s` to make future modifications to the scatter chart after it is created.

Examples

Create Scatter Plot

Create `x` as 200 equally spaced values between 0 and 3π. Create `y` as cosine values with random noise. Then, create a scatter plot.

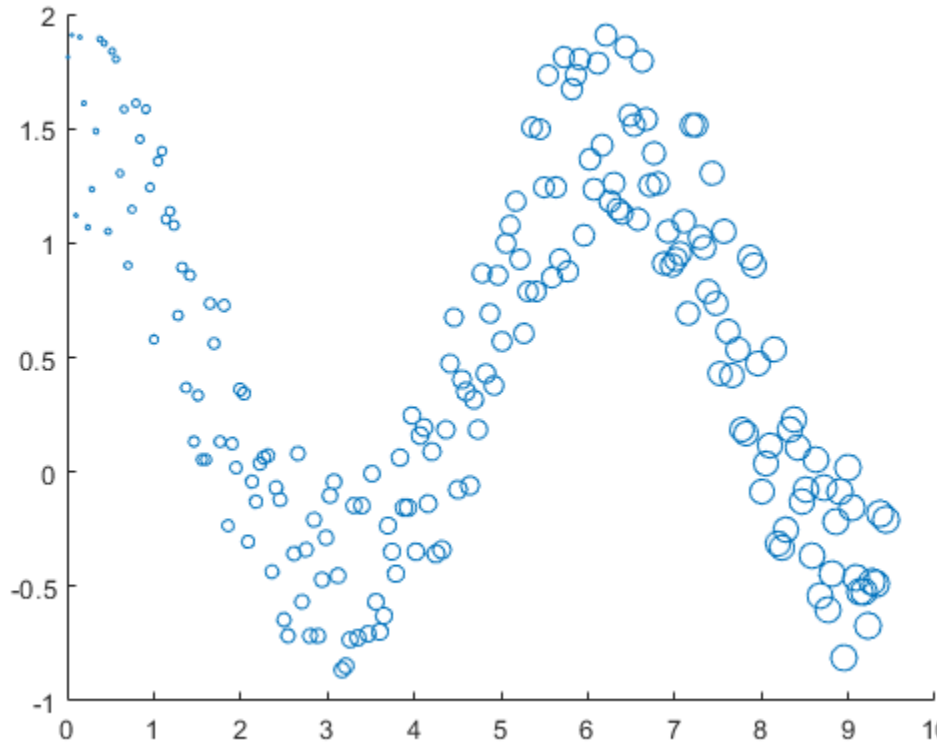
```
x = linspace(0,3*pi,200);
y = cos(x) + rand(1,200);
scatter(x,y)
```



Vary Circle Size

Create a scatter plot using circles with different sizes. Specify the size in points squared

```
x = linspace(0,3*pi,200);
y = cos(x) + rand(1,200);
sz = linspace(1,100,200);
scatter(x,y,sz)
```

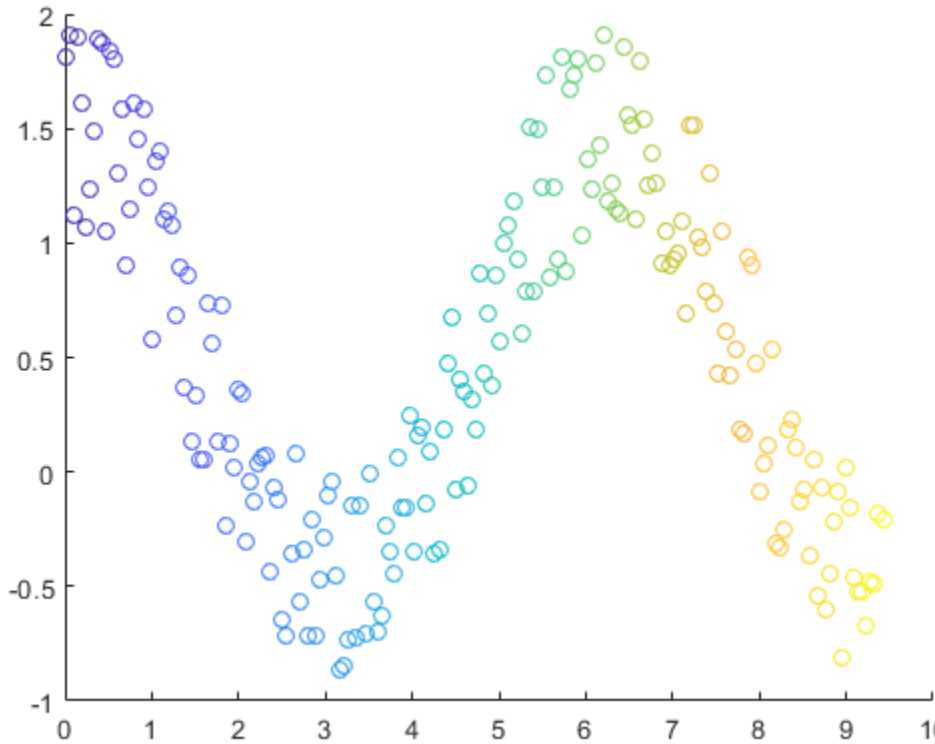


Corresponding elements in `x`, `y`, and `sz` determine the location and size of each circle. To plot all circles with the equal area, specify `sz` as a numeric scalar.

Vary Circle Color

Create a scatter plot and vary the circle color.

```
x = linspace(0,3*pi,200);
y = cos(x) + rand(1,200);
c = linspace(1,10,length(x));
scatter(x,y,[],c)
```



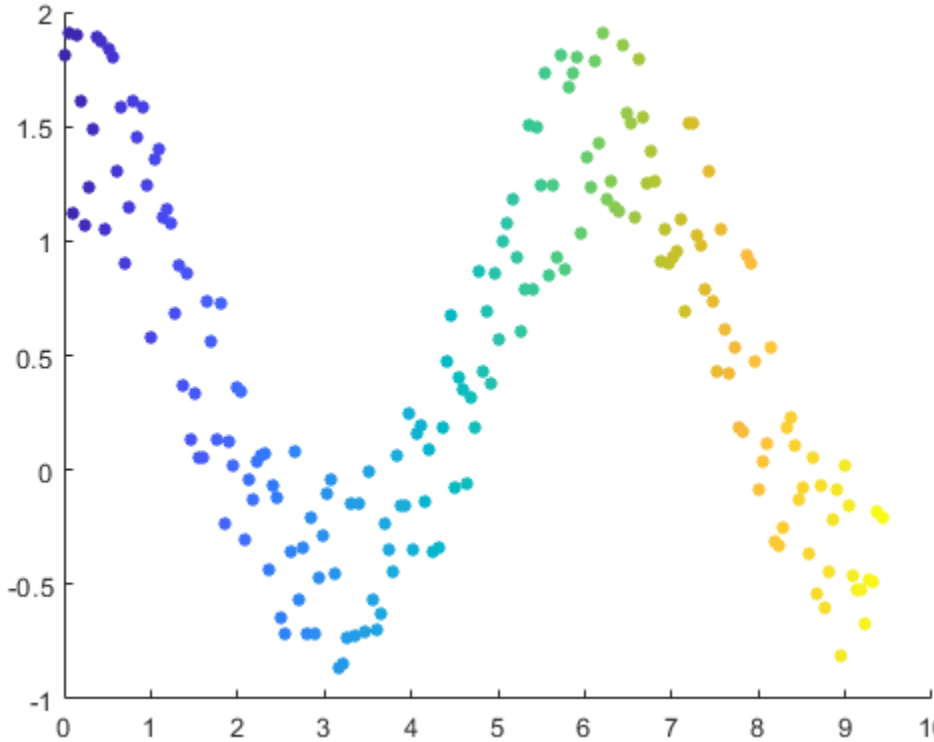
Corresponding elements in `x`, `y`, and `c` determine the location and color of each circle. The `scatter` function maps the elements in `c` to colors in the current colormap.

Fill the Markers

Create a scatter plot and fill in the markers. `scatter` fills each marker using the color of the marker edge.

```
x = linspace(0,3*pi,200);
y = cos(x) + rand(1,200);
sz = 25;
```

```
c = linspace(1,10,length(x));
scatter(x,y,sz,c,'filled')
```

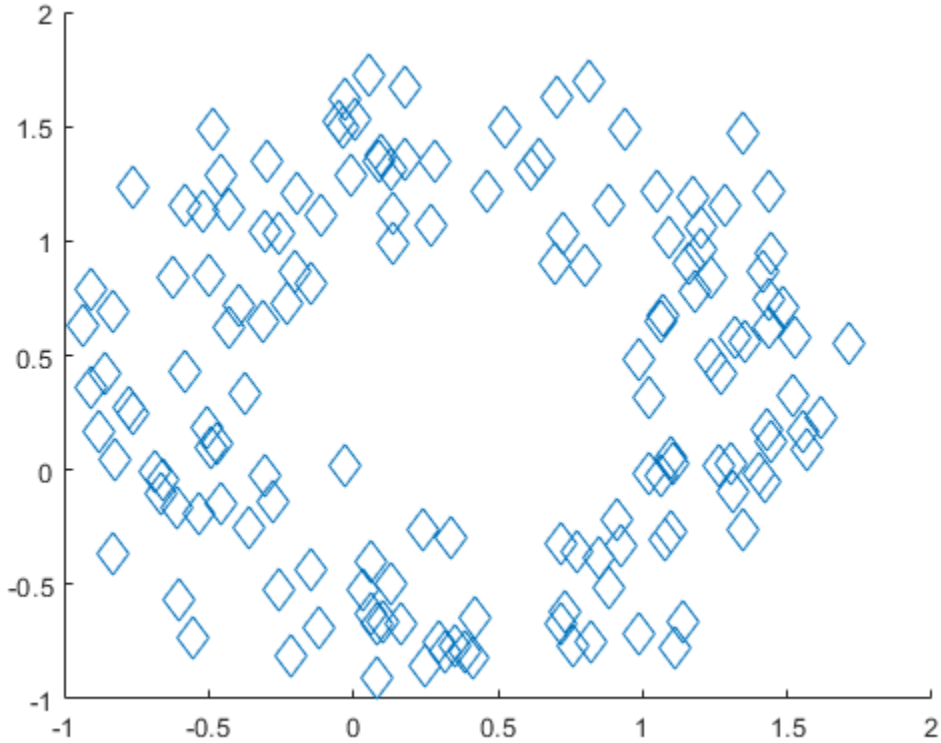


Specify Marker Symbol

Create vectors x and y as sine and cosine values with random noise. Then, create a scatter plot and use diamond markers with an area of 140 points squared.

```
theta = linspace(0,2*pi,150);
x = sin(theta) + 0.75*rand(1,150);
y = cos(theta) + 0.75*rand(1,150);
```

```
sz = 140;
scatter(x,y,sz,'d')
```

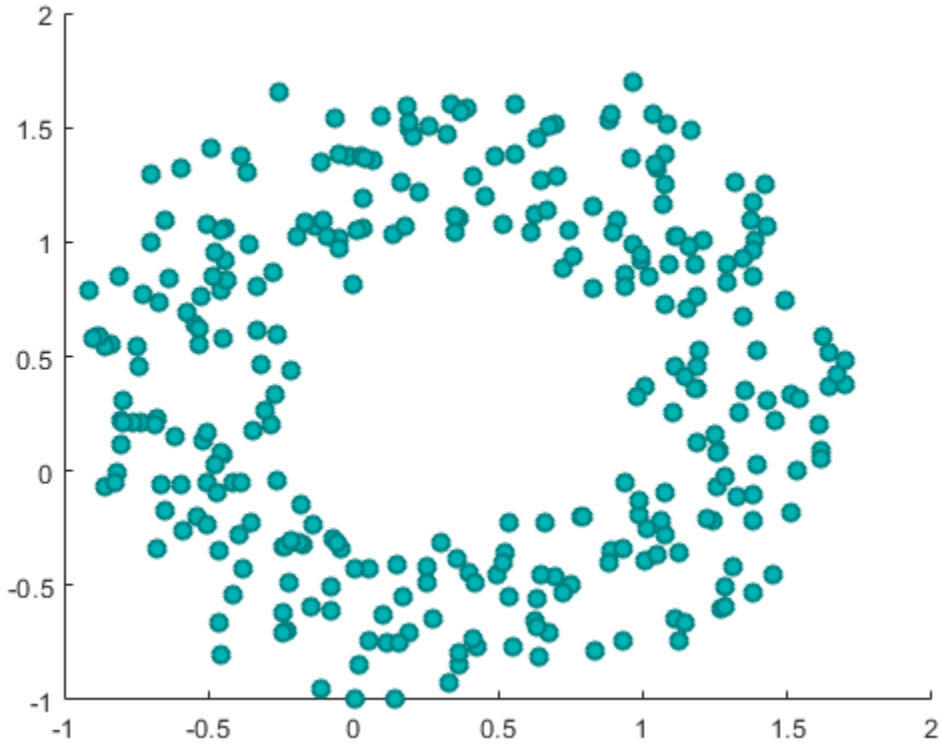


Change Marker Color and Line Width

Create vectors x and y as sine and cosine values with random noise. Create a scatter plot and set the marker edge color, marker face color, and line width.

```
theta = linspace(0,2*pi,300);
x = sin(theta) + 0.75*rand(1,300);
y = cos(theta) + 0.75*rand(1,300);
```

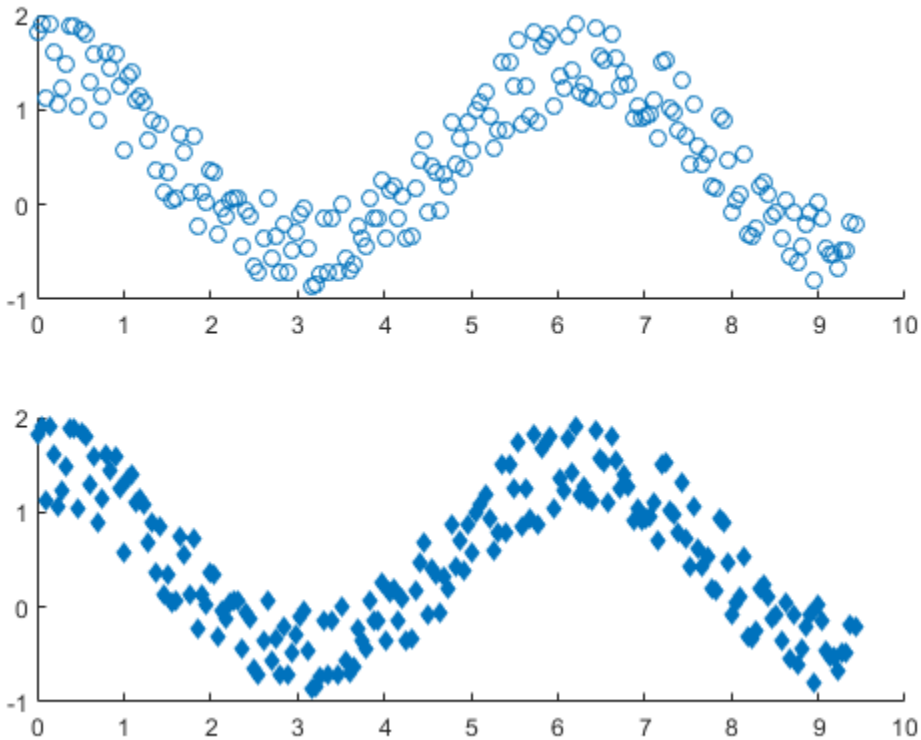
```
sz = 40;
scatter(x,y,sz,'MarkerEdgeColor',[0 .5 .5],...
               'MarkerFaceColor',[0 .7 .7],...
               'LineWidth',1.5)
```



Specify Target Axes and Marker Type

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Plot scattered data into each axes. In the bottom scatter plot, specify diamond filled diamond markers.

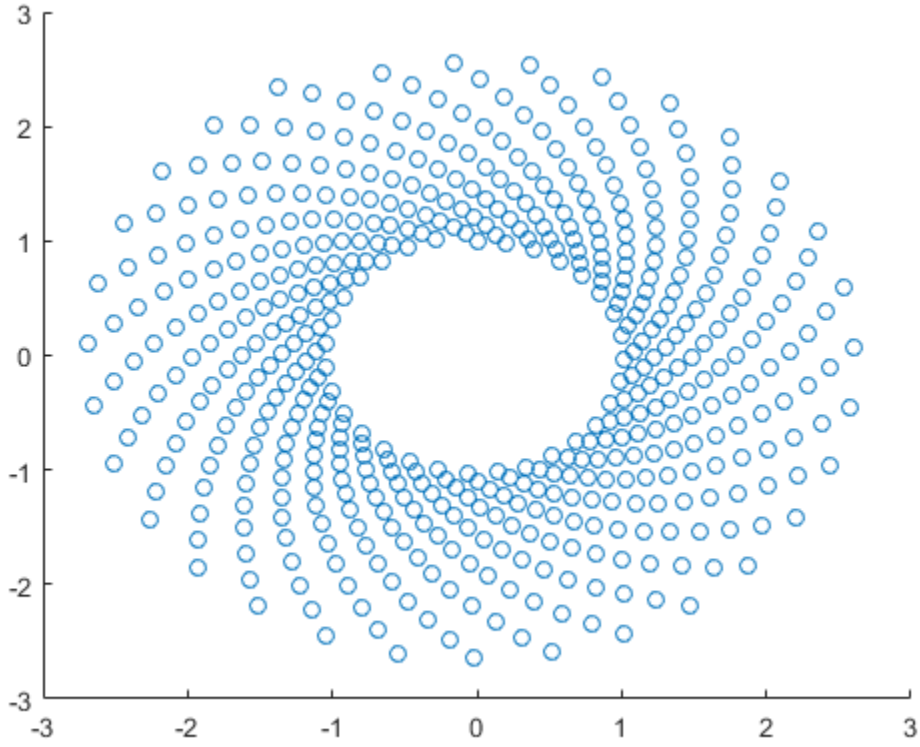
```
x = linspace(0,3*pi,200);
y = cos(x) + rand(1,200);
tiledlayout(2,1)

% Top plot
ax1 = nexttile;
scatter(ax1,x,y)

% Bottom plot
ax2 = nexttile;
scatter(ax2,x,y,'filled','d')
```



Modify Scatter Series After Creation

Create a scatter plot and return the scatter series object, `s`.

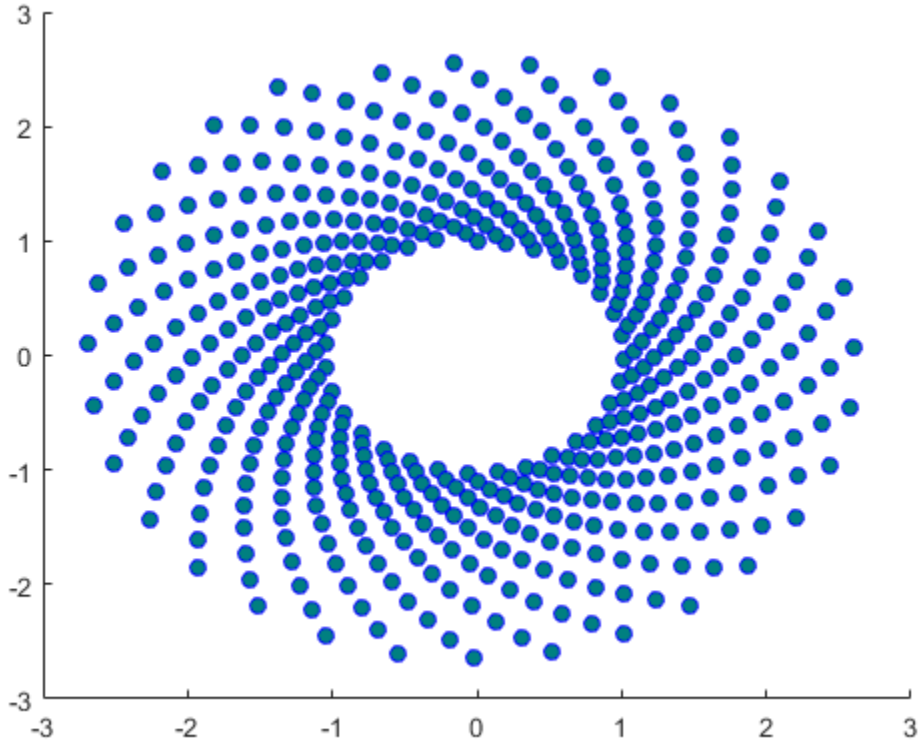
```
theta = linspace(0,1,500);
x = exp(theta).*sin(100*theta);
y = exp(theta).*cos(100*theta);
s = scatter(x,y);
```



Use `s` to query and set properties of the scatter series after it has been created. Set the line width to `0.6` point. Set the marker edge color to blue. Set the marker face color using an RGB triplet color.

Note: Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead, such as `set(s,'LineWidth',0.6)`.

```
s.LineWidth = 0.6;
s.MarkerEdgeColor = 'b';
s.MarkerFaceColor = [0 0.5 0.5];
```



Input Arguments

x — x values
vector

x values, specified as a vector. `x` and `y` must be vectors of equal length.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

y — y values
vector

y values, specified as a vector. `x` and `y` must be vectors of equal length.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

sz — Marker area
36 (default) | numeric scalar | row or column vector | `[]`

Marker area in points squared, specified in one of these forms:

- Numeric scalar — Plot all markers with equal size.
- Row or column vector — Use different sizes for each marker. Corresponding elements in `x`, `y`, and `sz` determine the location and area of each marker. The length of `sz` must equal the length of `x` and `y`.
- `[]` — Use the default area of 36 points squared.

The units for the marker area is points squared.

Example: `50`

Example: `[36 25 25 17 46]`

c — Marker color
`[0 0 1]` (default) | RGB triplet | three-column matrix of RGB triplets | vector | `'r'` | `'g'` | `'b'` | ...

Marker color, specified in one of these forms:

- RGB triplet or color name — Plot all markers with the same color.
- Three column matrix of RGB triplets — Use different colors for each marker. Each row of the matrix specifies an RGB triplet color for the corresponding marker. The number of rows must equal the length of `x` and `y`.
- Vector — Use different colors for each marker and linearly map values in `c` to the colors in the current colormap. The length of `c` must equal the length of `x` and `y`. To change the colormap for the axes, use the `colormap` function.

If you have three points in the scatter plot and want the colors to be indices into the colormap, specify `c` as a three-element column vector.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`

Example: `'k'`

Example: `[1 2 3 4]`

`mkr` — Marker type

`'o'` (default) | `'+'` | `'*'` | `'.'` | `'x'` | ...

Marker type, specified as one of the values listed in this table.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square

Value	Description
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`'filled'` — Option to fill interior of markers
`'filled'`

Option to fill the interior of the markers, specified as `'filled'`. Use this option with markers that have a face, for example, `'o'` or `'square'`. Markers that do not have a face and contain only edges do not draw (`'+'`, `'*'`, `'.'`, and `'x'`).

The `'filled'` option sets the `MarkerFaceColor` property of the `Scatter` object to `'flat'` and the `MarkerEdgeColor` property to `'none'`, so the marker faces draw, but the edges do not.

ax — Target axes
`Axes` object | `PolarAxes` object | `GeographicAxes` object

Target axes, specified as an `Axes` object, a `PolarAxes` object, or a `GeographicAxes` object. If you do not specify the axes and if the current axes are Cartesian axes, then the `scatter` function uses the current axes. To plot into polar axes, specify the `PolarAxes` object as the first input argument or use the `polarscatter` function. To plot into geographic axes, specify the `GeographicAxes` object as the first input argument or use the `geoscatter` function.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'MarkerFaceColor','red'` sets the marker face color to red.

The `Scatter` object properties listed here are only a subset. For a complete list, see Scatter.

`MarkerEdgeColor` — Marker outline color

`'flat'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified `'flat'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'flat'` uses colors from the `CData` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

Example: `'#D2F9A7'`

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'flat'` | `'auto'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'flat'`, `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'flat'` option uses the `CData` values. The `'auto'` option uses the same color as the `Color` property for the axes.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Example: `'#D2F9A7'`

`LineWidth` — Width of marker edge

`0.5` (default) | positive value

Width of marker edge, specified as a positive value in point units.

Example: `0.75`

Output Arguments

`s` — `Scatter` object
`Scatter` object

`Scatter` object. Use `s` to access and modify properties of the scatter chart after it has been created.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- Supported syntaxes for tall arrays `X` and `Y` are:
  - `scatter(X,Y)`
  - `scatter(X,Y,sz)`
  - `scatter(X,Y,sz,c)`
  - `scatter(___,'filled')`
  - `scatter(___,mkr)`
  - `scatter(___,Name,Value)`
  - `scatter(ax,___)`
- `sz` must be scalar or empty `[]`.
- `c` must be scalar or an RGB triplet.
- Categorical inputs are not supported.
- With tall arrays, the `scatter` function plots in iterations, progressively adding to the plot as more data is read. During the updates, a progress indicator shows the proportion of data that has been plotted. Zooming and panning is supported during

the updating process, before the plot is complete. To stop the update process, press the pause button in the progress indicator.

For more information, see “Visualization of Tall Arrays”.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions
`hold` | `plot` | `scatter3`

Properties
Scatter

Topics

“Plot Dates and Durations”

“Plot Categorical Data”

External Websites

MATLAB Plot Gallery

Introduced before R2006a

scatter3

3-D scatter plot

Syntax

```
scatter3(X,Y,Z)
scatter3(X,Y,Z,S)
scatter3(X,Y,Z,S,C)
scatter3( ___ ,'filled')
scatter3( ___ ,markertype)
scatter3( ___ ,Name,Value)

scatter3(ax, ___ )

h = scatter3( ___ )
```

Description

`scatter3(X,Y,Z)` displays circles at the locations specified by the vectors `X`, `Y`, and `Z`.

`scatter3(X,Y,Z,S)` draws each circle with the size specified by `S`. To plot each circle with equal size, specify `S` as a scalar. To plot each circle with a specific size, specify `S` as a vector.

`scatter3(X,Y,Z,S,C)` draws each circle with the color specified by `C`.

- If `C` is a RGB triplet or character vector or string containing a color name, then all circles are plotted with the specified color.
- If `C` is a three column matrix with the number of rows in `C` equal to the length of `X`, `Y`, and `Z`, then each row of `C` specifies an RGB color value for the corresponding circle.
- If `C` is a vector with length equal to the length of `X`, `Y`, and `Z`, then the values in `C` are linearly mapped to the colors in the current colormap.

`scatter3( ___ ,'filled')` fills in the circles, using any of the input argument combinations in the previous syntaxes.

`scatter3( ___ ,markertype)` specifies the marker type.

`scatter3( ___ ,Name,Value)` modifies the scatter chart using one or more name-value pair arguments.

`scatter3(ax, ___ )` plots into the axes specified by `ax` instead of into the current axes (`gca`). The `ax` option can precede any of the input argument combinations in the previous syntaxes.

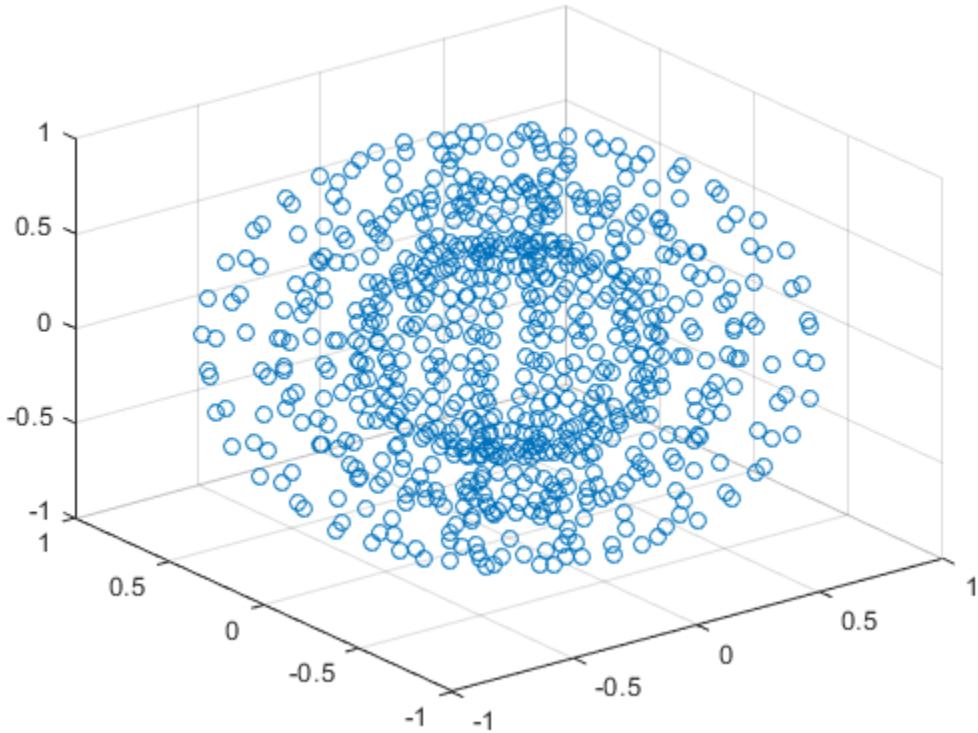
`h = scatter3( ___ )` returns the `Scatter` object. Use `h` to modify properties of the scatter chart after it is created.

Examples

Create 3-D Scatter Plot

Create a 3-D scatter plot. Use `sphere` to define vectors `x`, `y`, and `z`.

```
figure
[X,Y,Z] = sphere(16);
x = [0.5*X(:); 0.75*X(:); X(:)];
y = [0.5*Y(:); 0.75*Y(:); Y(:)];
z = [0.5*Z(:); 0.75*Z(:); Z(:)];
scatter3(x,y,z)
```



Vary Marker Size

Use `sphere` to define vectors `x`, `y`, and `z`.

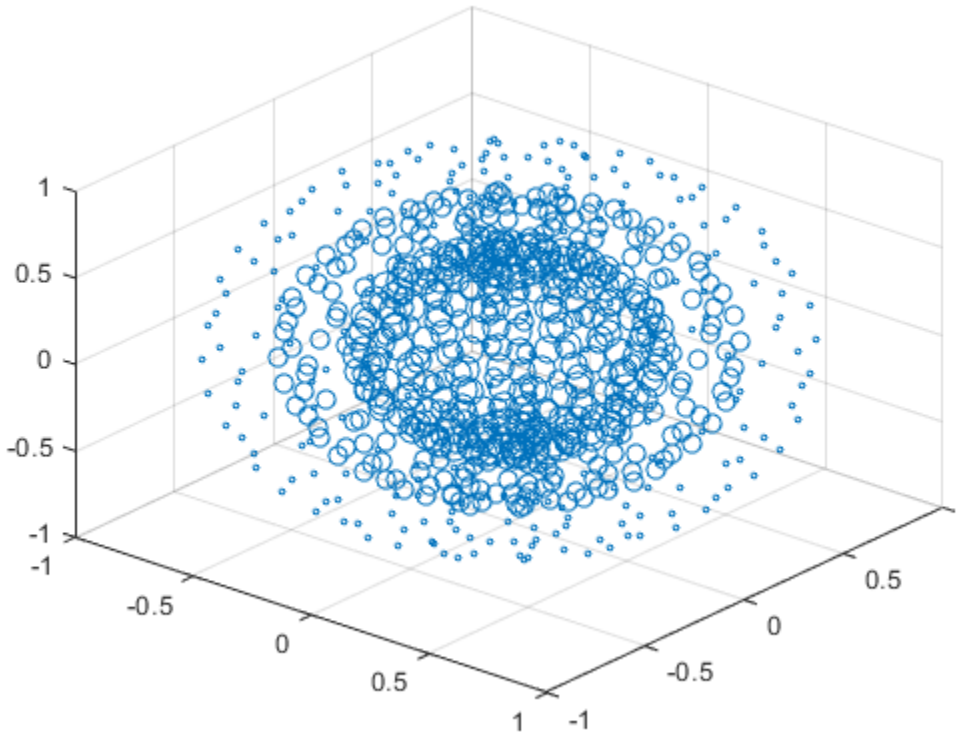
```
[X,Y,Z] = sphere(16);
x = [0.5*X(:); 0.75*X(:); X(:)];
y = [0.5*Y(:); 0.75*Y(:); Y(:)];
z = [0.5*Z(:); 0.75*Z(:); Z(:)];
```

Define vector `s` to specify the marker sizes.

```
S = repmat([100,50,5],numel(X),1);
s = S(:);
```

Create a 3-D scatter plot and use `view` to change the angle of the axes in the figure.

```
figure
scatter3(x,y,z,s)
view(40,35)
```



Corresponding entries in `x`, `y`, `z`, and `s` determine the location and size of each marker.

Vary Marker Color

Use `sphere` to define vectors `x`, `y`, and `z`.

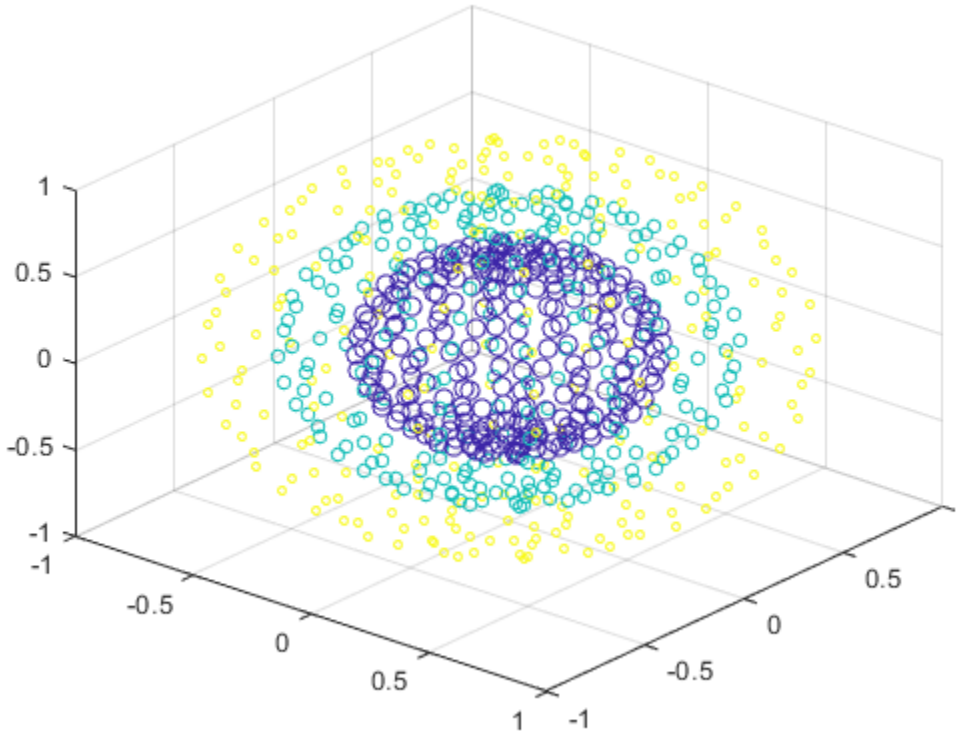
```
[X,Y,Z] = sphere(16);
x = [0.5*X(:); 0.75*X(:); X(:)];
y = [0.5*Y(:); 0.75*Y(:); Y(:)];
z = [0.5*Z(:); 0.75*Z(:); Z(:)];
```

Define vectors `s` and `c` to specify the size and color of each marker.

```
S = repmat([50,25,10],numel(X),1);
C = repmat([1,2,3],numel(X),1);
s = S(:);
c = C(:);
```

Create a 3-D scatter plot and use `view` to change the angle of the axes in the figure.

```
figure
scatter3(x,y,z,s,c)
view(40,35)
```



Corresponding entries in `x`, `y`, `z`, and `c` determine the location and color of each marker.

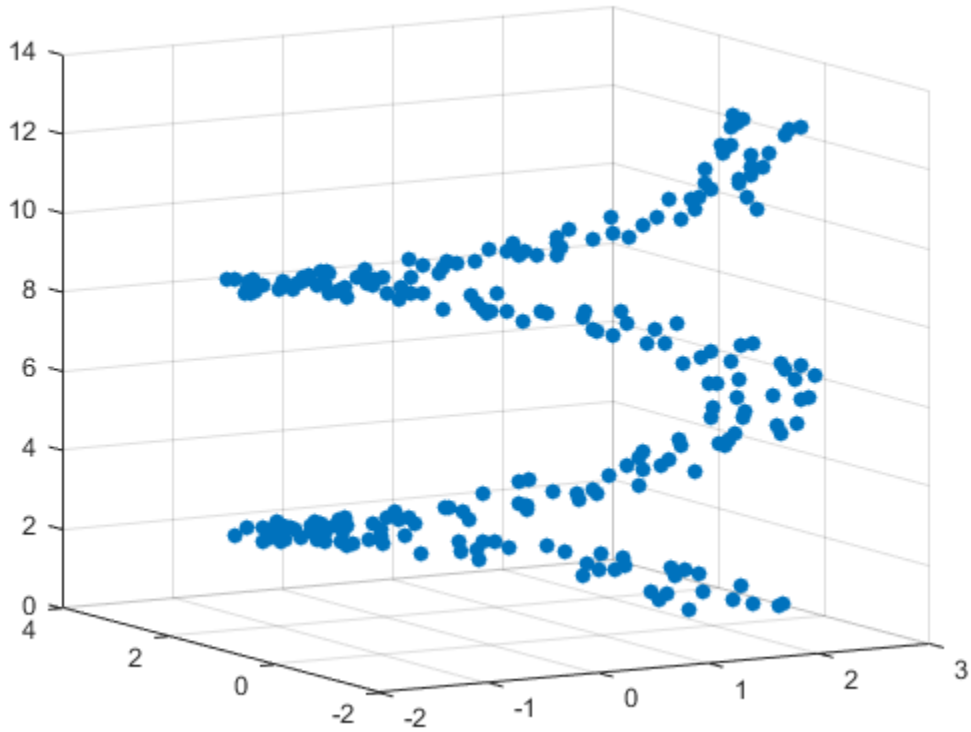
Fill in Markers

Create vectors `x` and `y` as cosine and sine values with random noise.

```
z = linspace(0,4*pi,250);
x = 2*cos(z) + rand(1,250);
y = 2*sin(z) + rand(1,250);
```

Create a 3-D scatter plot and fill in the markers. Use `view` to change the angle of the axes in the figure.

```
scatter3(x,y,z,'filled')
view(-30,10)
```



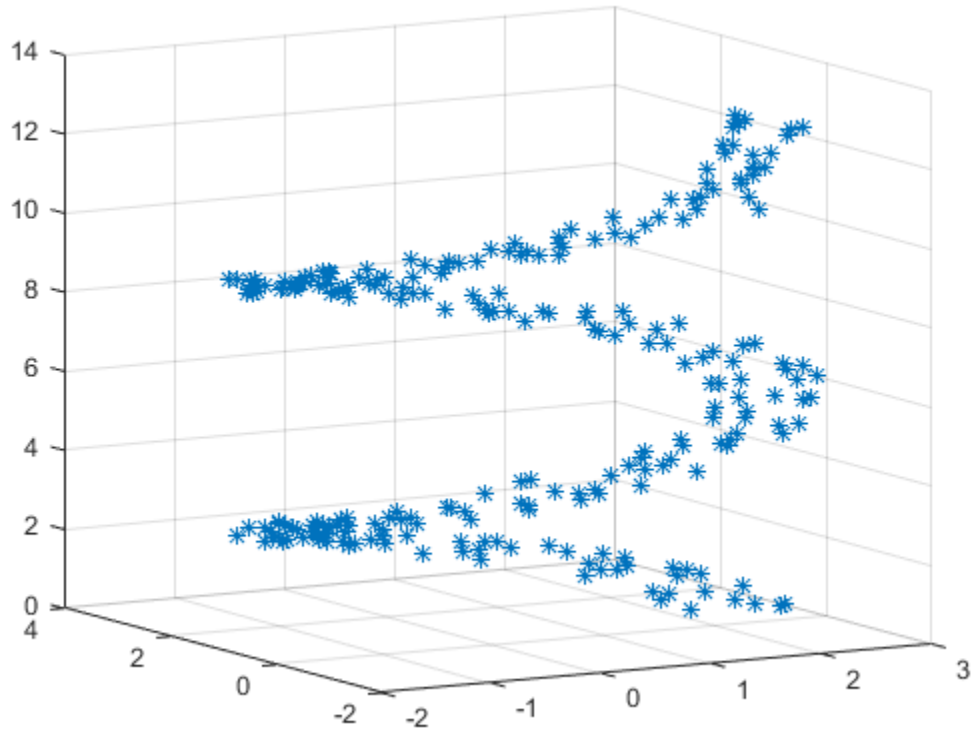
Set Marker Type

Initialize the random-number generator to make the output of `rand` repeatable. Define vectors `x` and `y` as cosine and sine values with random noise.

```
rng default
z = linspace(0,4*pi,250);
x = 2*cos(z) + rand(1,250);
y = 2*sin(z) + rand(1,250);
```

Create a 3-D scatter plot and set the marker type. Use `view` to change the angle of the axes in the figure.

```
figure
scatter3(x,y,z,'*')
view(-30,10)
```



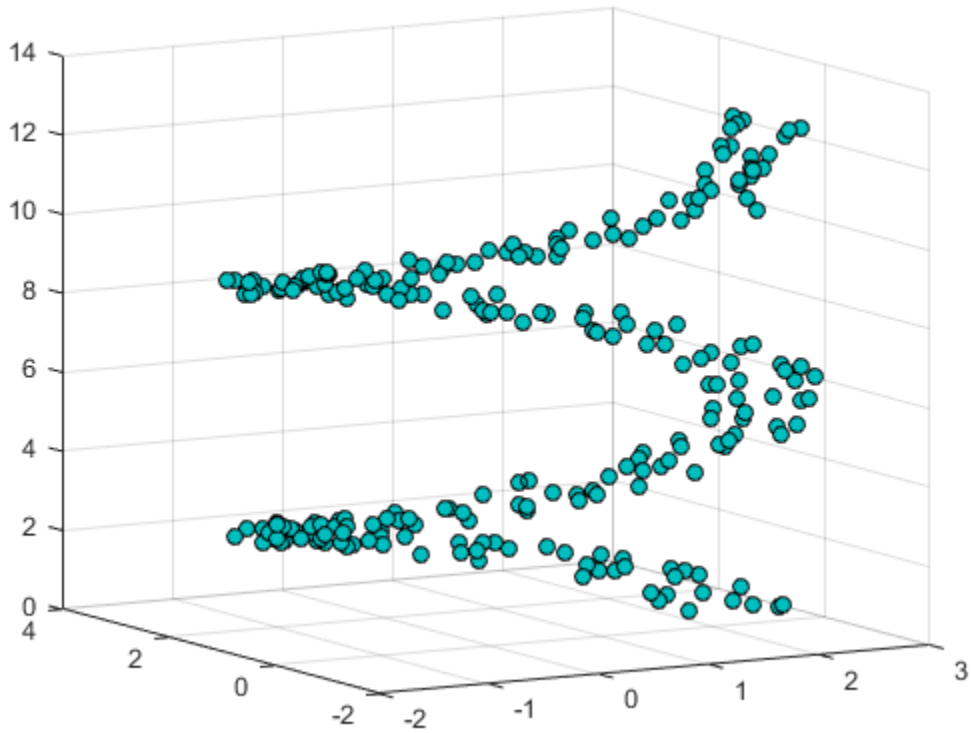
Set Marker Properties

Initialize the random-number generator to make the output of `rand` repeatable. Define vectors `x` and `y` as cosine and sine values with random noise.

```
rng default
z = linspace(0,4*pi,250);
x = 2*cos(z) + rand(1,250);
y = 2*sin(z) + rand(1,250);
```

Create a 3-D scatter plot and set the marker edge color and the marker face color. Use `view` to change the angle of the axes in the figure.

```
figure
scatter3(x,y,z,...
        'MarkerEdgeColor','k',...
        'MarkerFaceColor',[0 .75 .75])
view(-30,10)
```

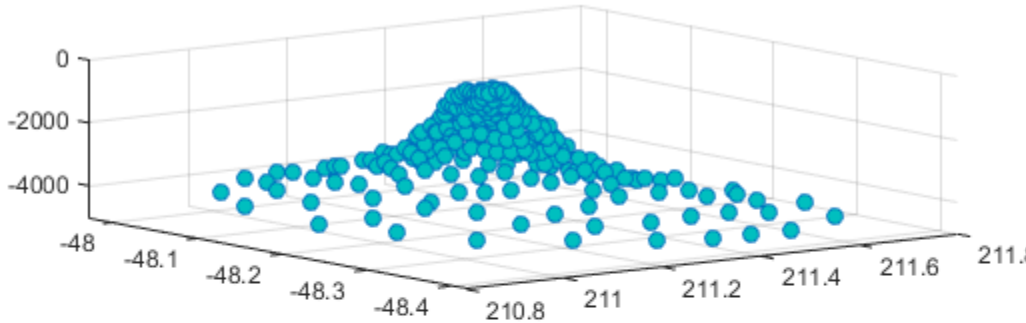


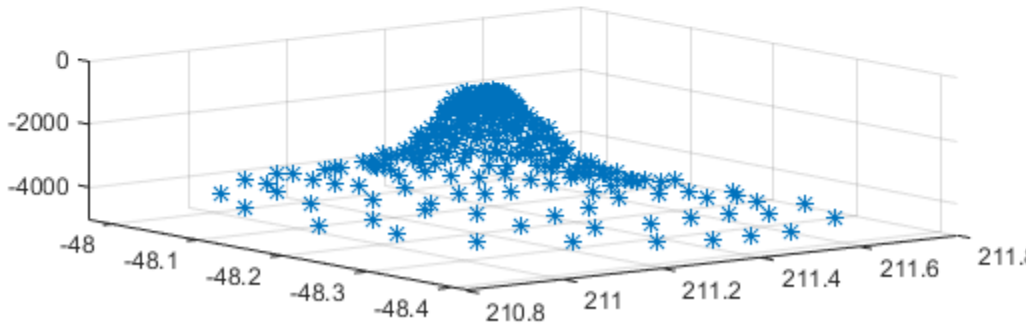
Specify Axes for 3-D Scatter Plot

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions.

Load the `seamount` data set to get vectors `x`, `y`, and `z`. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Then create separate scatter plots in the axes by specifying the axes object as the first argument to `scatter3`.

```
load seamount
tiledlayout(2,1)
ax1 = nexttile;
ax2 = nexttile;
scatter3(ax1,x,y,z,'MarkerFaceColor',[0 .75 .75])
scatter3(ax2,x,y,z,'*')
```





Set Scatter Series Properties Using Handle

Use the `sphere` function to create vectors `x`, `y`, and `z`.

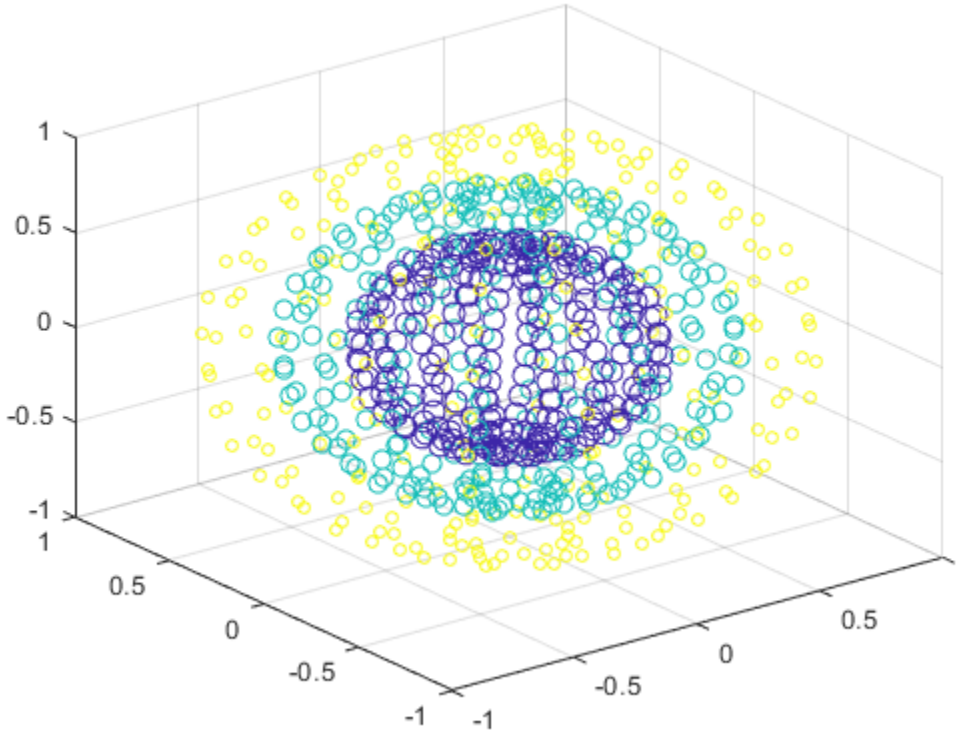
```
[X,Y,Z] = sphere(16);
x = [0.5*X(:); 0.75*X(:); X(:)];
y = [0.5*Y(:); 0.75*Y(:); Y(:)];
z = [0.5*Z(:); 0.75*Z(:); Z(:)];
```

Create vectors `s` and `c` to specify the size and color for each marker.

```
S = repmat([70,50,20],numel(X),1);
C = repmat([1,2,3],numel(X),1);
s = S(:);
c = C(:);
```

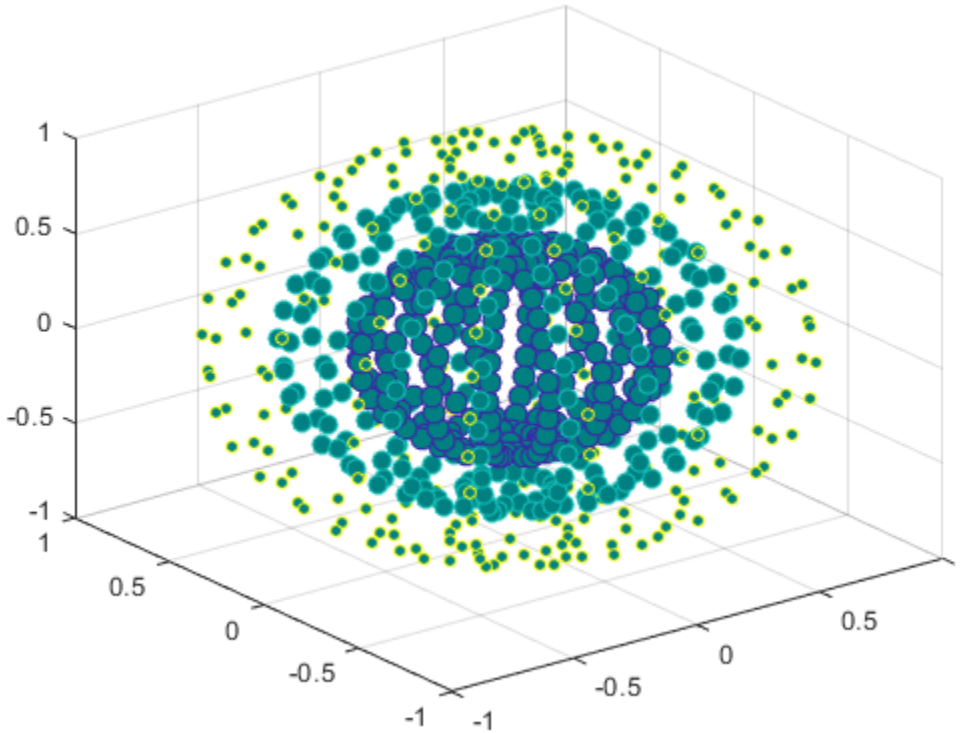
Create a 3-D scatter plot and return the scatter series object.

```
h = scatter3(x,y,z,s,c);
```



Use an RGB triplet color value to set the marker face color. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead.

```
h.MarkerFaceColor = [0 0.5 0.5];
```



Input Arguments

X — x values
vector

x values, specified as a vector. X, Y, and Z must be vectors of equal length.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

Y — y values
vector

y values, specified as a vector. X, Y, and Z must be vectors of equal length.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

Z — z values
vector

z values, specified as a vector. X, Y, and Z must be vectors of equal length.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

S — Marker area
36 (default) | scalar | vector | `[]`

Marker area, specified as a scalar, a vector, or `[]`. The values in S must be positive. The units for area are points squared.

- If S is a scalar, then `scatter3` plots all markers with the specified area.
- If S is a row or column vector, then each entry in S specifies the area for the corresponding marker. The length of S must equal the length of X, Y and Z. Corresponding entries in X, Y, Z and S determine the location and area of each marker.
- If S is empty, then the default size of 36 points squared is used.

Example: `50`

Example: `[36,25,25,17,46]`

C — Marker color
`[0 0 1]` (blue) (default) | RGB triplet | three-column matrix of RGB triplets | vector | `'r'` | `'g'` | `'b'` | ...

Marker color, specified as a an RGB triplet, a three-column matrix of RGB triplet, a vector, or one of the color options in the table.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`

If you have three points in the scatter plot and want the colors to be indices into the colormap, specify `C` as a three-element column vector.

Example: `'y'`

Example: `[1,2,3,4]`

`markertype` — Marker
`'o'` (default) | `'+'` | `'*'` | `'.'` | `'x'` | ...

Marker, specified as one of the markers in this table.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle

Value	Description
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

ax — Axes object
axes object

Axes object. If you do not specify an axes, then `scatter3` plots into the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'MarkerFaceColor','red'` sets the marker face color to red.

The properties listed here are only a subset. For a complete list, see Scatter.

`LineWidth` — Width of marker edge
`0.5` (default) | positive value

Width of marker edge, specified as a positive value in point units.

Example: `0.75`

`MarkerEdgeColor` — Marker outline color
`'flat'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified `'flat'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'flat'` uses colors from the `CData` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to

F. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

Example: `'#D2F9A7'`

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'flat'` | `'auto'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'flat'`, `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'flat'` option uses the `CData` values. The `'auto'` option uses the same color as the `Color` property for the axes.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Example: `'#D2F9A7'`

Output Arguments

h — Scatter object
`Scatter` object

`Scatter` object. This is a unique identifier, which you can use to query and modify the properties of the `Scatter` object after it is created.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions
`LineSpec` | `plot3` | `scatter`

Properties
Scatter

Topics

“Plot Dates and Durations”
“Plot Categorical Data”

Introduced before R2006a

Scatter Properties

Scatter chart appearance and behavior

Description

`Scatter` properties control the appearance and behavior of `Scatter` object. By changing property values, you can modify certain aspects of the scatter chart.

Starting in R2014b, you can use dot notation to query and set properties.

```
s = scatter(1:10,1:10);
m = s.Marker;
s.Marker = '*';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Markers

`Marker` — Marker symbol

`'o'` (default) | `'+'` | `'*'` | `'.'` | `'x'` | ...

Marker symbol, specified as one of the options listed in this table:

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond

Value	Description
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'	Six-pointed star (hexagram)
'none'	No markers

`LineWidth` — Width of marker edge

`0.5` (default) | positive value

Width of marker edge, specified as a positive value in point units.

Example: `0.75`

`MarkerEdgeColor` — Marker outline color

`'flat'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified `'flat'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'flat'` uses colors from the `CData` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: [0.5 0.5 0.5]

Example: 'blue'

Example: '#D2F9A7'

MarkerFaceColor — Marker fill color

'none' (default) | 'flat' | 'auto' | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...

Marker fill color, specified as `'flat'`, `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'flat'` option uses the `CData` values. The `'auto'` option uses the same color as the `Color` property for the axes.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Example: `'#D2F9A7'`

`MarkerEdgeAlpha` — Edge transparency

1 (default) | scalar in range `[0,1]`

Marker edge transparency, specified as a scalar in the range `[0,1]`. A value of 1 is opaque and 0 is completely transparent. Values between 0 and 1 are semitransparent.

Example: `s.MarkerEdgeAlpha = 0.5;`

`MarkerFaceAlpha` — Marker face transparency

1 (default) | scalar in range `[0,1]`

Marker face transparency, specified as a scalar in the range `[0,1]`. A value of 1 is opaque and 0 is completely transparent. Values between 0 and 1 are semitransparent.

Example: `s.MarkerFaceAlpha = 0.5;`

Color and Size Data

`CData` — Marker colors

`[]` (default) | RGB triplet | matrix of RGB triplets | vector

Marker colors, specified as one of these values:

- RGB triplet — Use the same color for all the markers in the plot. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.5 0.6 0.7]`.

- Three-column matrix of RGB triplets — Use a different color for each marker in the plot. Each row of the matrix defines one color. The number of rows must equal the number of markers.
- Vector — Use a different color for each marker in the plot. Specify `CData` as a vector the same length as `XData`. Linearly map the values in the vector to the colors in the current colormap.

Example: `[1 0 0; 0 1 0; 0 0 1]`

`CDataMode` — Control how `CData` is set
`'auto'` (default) | `'manual'`

Control how the `CData` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the `CData` property by selecting a single color from the `ColorOrder` property of the axes.
- `'manual'` — You control the value of the `CData` property manually, either by specifying a color when you call `scatter` or `scatter3`, or by setting the `CData` property on the `Scatter` object after creating it.

If you change the value of the `CData` property manually, MATLAB changes the value of the `CDataMode` property to `'manual'`.

`CDataSource` — Variable linked to `CData`
`''` | character vector or string containing MATLAB workspace variable

Variable linked to `CData`, specified as a character vector or string containing a MATLAB workspace variable. MATLAB evaluates the variable in the base workspace to generate the `CData`.

By default, there is no linked variable so the value is an empty character vector. If you link a variable, then MATLAB does not update the `CData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`SizeData` — Marker sizes
`[]` (default) | scalar | vector

Marker sizes, specified in one of these forms:

- Scalar — Use the same size for all of the markers.
- Vector — Use a different size for each marker. Specify `SizeData` as a vector the same length as `XData`.

Specify the values in point units, where one point equals 1/72 inch. To specify a marker that has an area of one square inch, use a value of 72^2.

Example: `50`

`SizeDataSource` — Variable linked to `SizeData`

`''` | character vector or string containing MATLAB workspace variable

Variable linked to `SizeData`, specified as a character vector or string containing a MATLAB workspace variable. MATLAB evaluates the variable in the base workspace to generate the `SizeData`.

By default, there is no linked variable so the value is an empty character vector. If you link a variable, then MATLAB does not update the `SizeData` values. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Cartesian Coordinate Data

`XData` — x values

`[]` (default) | scalar | vector

x values, specified as a scalar or a vector. The scatter plot displays an individual marker for each value in `XData`.

The input argument `X` to the `scatter` and `scatter3` functions set the *x* values. `XData` and `YData` must have equal lengths.

Example: `[1 2 4 2 6]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

XDataSource — Variable linked to XData

`''` (default) | character vector | string

Variable linked to `XData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `XData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `XData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'x'`

YData — y values

`[]` (default) | scalar | vector

y values, specified as a scalar or a vector. The scatter plot displays an individual marker for each value in `YData`.

The input argument `Y` to the `scatter` and `scatter3` functions set the *y* values. `XData` and `YData` must have equal lengths.

Example: `[1 3 3 4 6]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

YDataSource — Variable linked to YData

`''` (default) | character vector | string

Variable linked to `YData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `YData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `YData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'y'`

`ZData` — z values
`[]` (default) | scalar | vector

z values, specified as a scalar or a vector.

- For 2-D scatter plots, `ZData` is empty by default.
- For 3-D scatter plots, the input argument `Z` to the `scatter3` function sets the *z* values. `XData`, `YData`, and `ZData` must have equal lengths.

Example: `[1 2 2 1 0]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`ZDataSource` — Variable linked to `ZData`
`''` (default) | character vector | string

Variable linked to `ZData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `ZData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `ZData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'z'`

Polar Coordinate Data

`RData` — Radius values
vector

Radius values, specified as a vector. `ThetaData` and `RData` must be vectors of equal length.

This property applies only to polar axes.

`RDataSource` — Variable linked to `RData`
`''` (default) | character vector or string containing MATLAB workspace variable name

Variable linked to `RData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `RData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `RData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

This property applies only to polar axes.

`ThetaData` — Angle values
vector

Angle values, specified as a vector. `ThetaData` and `RData` must be vectors of equal length.

This property applies only to polar axes.

`ThetaDataSource` — Variable linked to `ThetaData`
`''` (default) | character vector or string containing MATLAB workspace variable name

Variable linked to `ThetaData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `RData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `ThetaData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

This property applies only to polar axes.

Geographic Coordinate Data

`LatitudeData` — Latitude values
vector

Latitude values, specified as a vector. `LatitudeData` and `LongitudeData` must be vectors of equal length.

This property applies only to geographic axes.

`LatitudeDataSource` — Variable linked to `LatitudeData`
`''` (default) | character vector or string containing MATLAB workspace variable name

Variable linked to `LatitudeData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `RData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, MATLAB does not update the `LatitudeData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

This property applies only to geographic axes.

`LongitudeData` — Longitude values
vector

Longitude values, specified as a vector. `LongitudeData` and `LatitudeData` must be vectors of equal length.

This property applies only to geographic axes.

`LongitudeDataSource` — Variable linked to `LongitudeData`

`''` (default) | character vector or string containing MATLAB workspace variable name

Variable linked to `LongitudeData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `RData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, MATLAB does not update the `LatitudeData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

This property applies only to geographic axes.

Legend

`DisplayName` — Legend label

`''` (default) | character vector | string scalar

Legend label, specified as a character vector or string scalar. The legend does not display until you call the `legend` command. If you do not specify the text, then `legend` sets the label using the form `'dataN'`.

`Annotation` — Control for including or excluding object from legend

`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`DataTipTemplate` — Data tip content

`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

`Clipping` — Clipping of object to axes limits

`'on'` (default) | `'off'`

Clipping of the object to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the object that are outside the axes limits.
- `'off'` — Display the entire object, even if parts of it appear outside the axes limits. Parts of the object might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the object so that it is larger than the original plot.

The `Clipping` property of the axes that contains the object must be set to `'on'`. Otherwise, this property has no effect. For more information about the clipping behavior, see the `Clipping` property of the axes.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see "Callback Definition".

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks when visible. The `Visible` property must be set to `'on'` and you must click a part of the `Scatter` object that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The `HitTest` property determines if the `Scatter` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `Scatter` object passes the click to the object below it in the current view of the figure window. The `HitTest` property of the `Scatter` object has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `Scatter` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `Scatter` object that has one of these:
  - `HitTest` property set to `'on'`
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

Note The `PickableParts` property determines if the `Scatter` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent

`Axes` object | `PolarAxes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `PolarAxes`, `Group`, or `Transform` object.

`Children` — Children

empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the chart object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.

- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'scatter'`

This property is read-only.

Type of graphics object, returned as `'scatter'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`scatter` | `scatter3`

Topics

"Access Property Values"
"Graphics Object Properties"

Introduced before R2006a

scatterhistogram

Create scatter plot with histograms

Syntax

```
s = scatterhistogram(tbl,xvar,yvar)
s = scatterhistogram(tbl,xvar,yvar,'GroupVariable',grpvar)

s = scatterhistogram(xvalues,yvalues)
s = scatterhistogram(xvalues,yvalues,'GroupData',grpvalues)

s = scatterhistogram( ___ ,Name,Value)
s = scatterhistogram(parent, ___ )
```

Description

`s = scatterhistogram(tbl,xvar,yvar)` creates a scatter plot with marginal histograms from the table `tbl` and returns the `ScatterHistogramChart` object. The `xvar` input indicates the table variable to display along the *x*-axis. The `yvar` input indicates the table variable to display along the *y*-axis. Use `s` to modify the object after it is created. For a list of properties, see ScatterHistogramChart.

`s = scatterhistogram(tbl,xvar,yvar,'GroupVariable',grpvar)` uses the table variable specified by `grpvar` to group observations specified by `xvar` and `yvar`.

`s = scatterhistogram(xvalues,yvalues)` creates a scatter plot of the data in `xvalues` and `yvalues` and displays the marginal histograms for the `xvalues` and `yvalues` data along the *x*-axis and *y*-axis, respectively.

`s = scatterhistogram(xvalues,yvalues,'GroupData',grpvalues)` uses the data in `grpvalues` to group the data in `xvalues` and `yvalues`.

`s = scatterhistogram( ___ ,Name,Value)` specifies additional options for the scatter plot with marginal histograms using one or more name-value pair arguments. Specify the options after all other input arguments. For a list of properties, see ScatterHistogramChart.

`s = scatterhistogram(parent, ___ )` creates the scatter plot with marginal histograms in the figure, panel, or tab specified by `parent`.

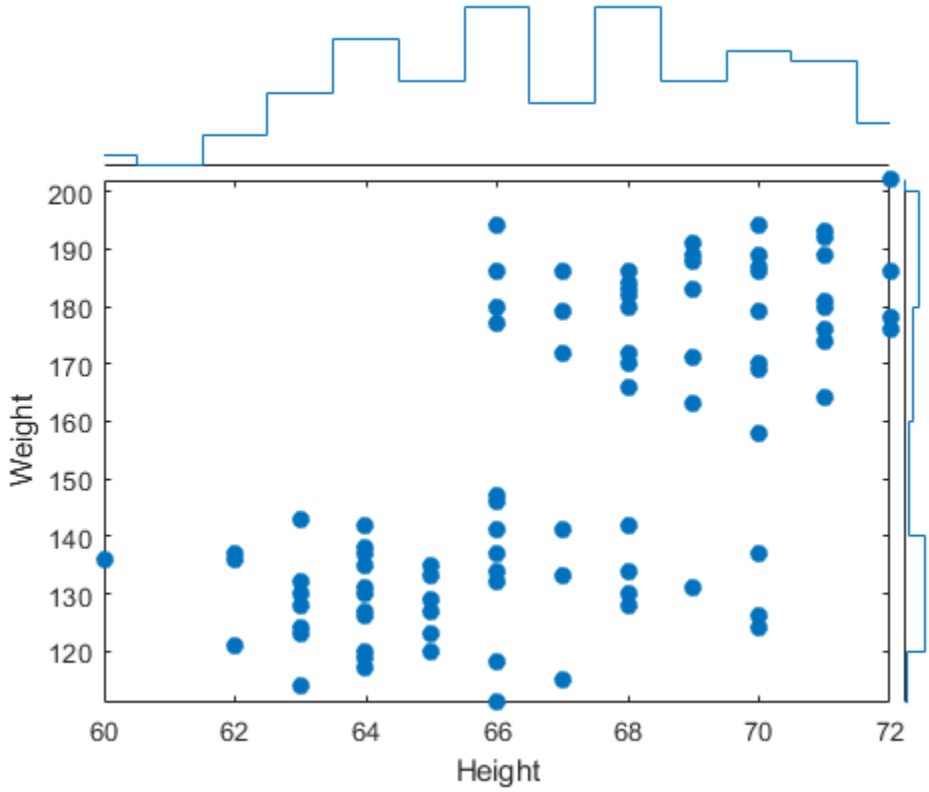
Examples

Scatter Histogram Chart with Tabular Data

Create a scatter plot with marginal histograms from a table of data for medical patients.

Load the `patients` data set and create a table from a subset of the variables loaded into the workspace. Then, create a scatter histogram chart comparing the `Height` values to the `Weight` values.

```
load patients
tbl = table(LastName,Age,Gender,Height,Weight);
s = scatterhistogram(tbl,'Height','Weight');
```

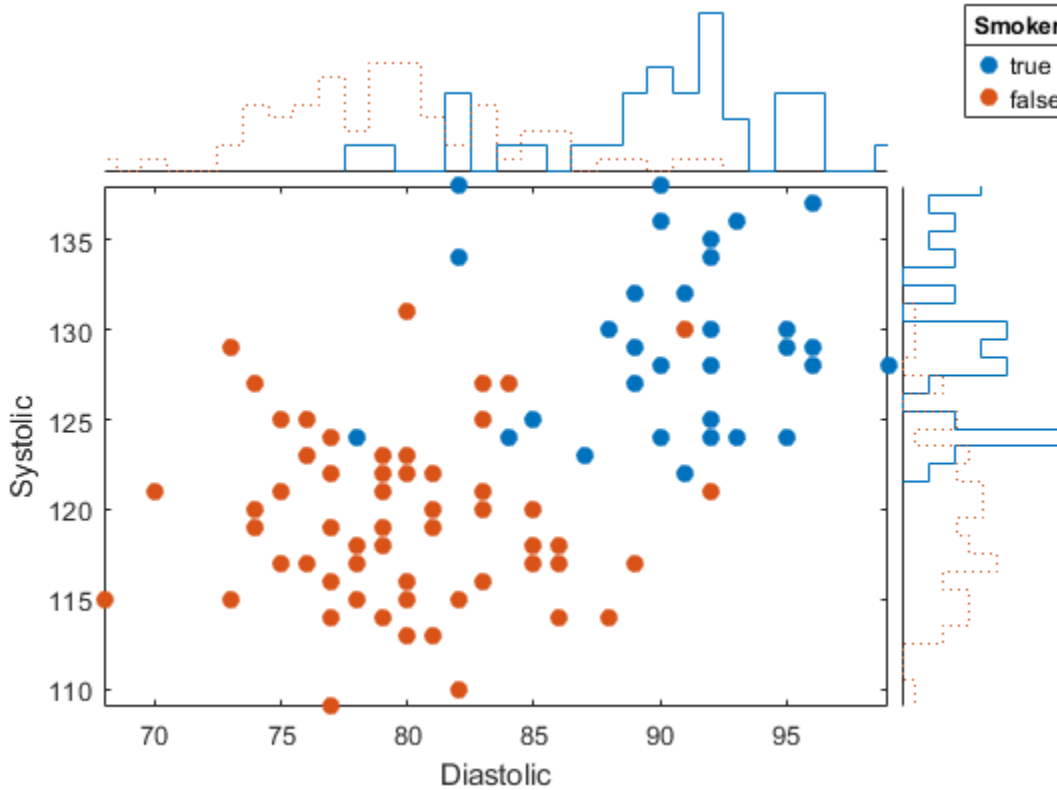


Specify Table Variable for Grouping Data

Using the `patients` data set, create a scatter plot with marginal histograms and specify the table variable to use for grouping the data.

Load the `patients` data set and create a scatter histogram chart from the data. Compare the patients' `Systolic` and `Diastolic` values. Group the data according to the patients' smoker status by setting the `'GroupVariable'` name-value pair argument to `'Smoker'`.

```
load patients
tbl = table(LastName,Diastolic,Systolic,Smoker);
s = scatterhistogram(tbl,'Diastolic','Systolic','GroupVariable','Smoker');
```



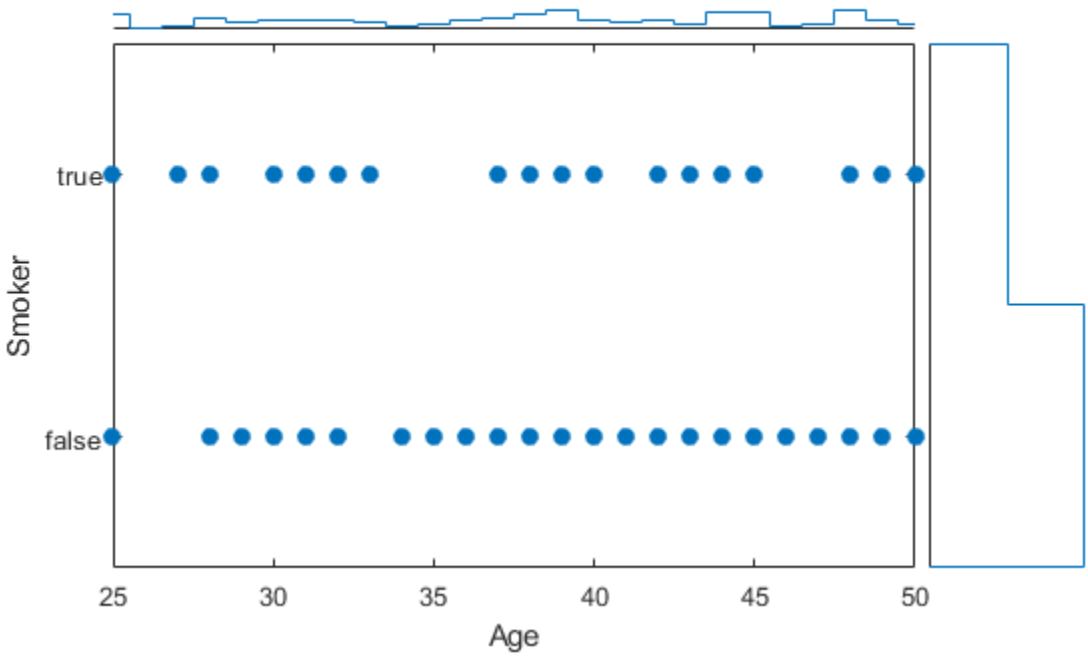
Visualize Categorical and Numeric Data

Use a scatter plot with marginal histograms to visualize categorical and numeric medical data.

Load the `patients` data set, and convert the `Smoker` data to a categorical array. Then, create a scatter histogram chart that compares patients' `Age` values to their smoker

status. The resulting scatter plot contains overlapping data points. However, the *y*-axis marginal histogram indicates that there are far more nonsmokers than smokers in the data set.

```
load patients
Smoker = categorical(Smoker);
s = scatterhistogram(Age,Smoker);
xlabel('Age')
ylabel('Smoker')
```



Specify Group Data and Customize Chart Properties

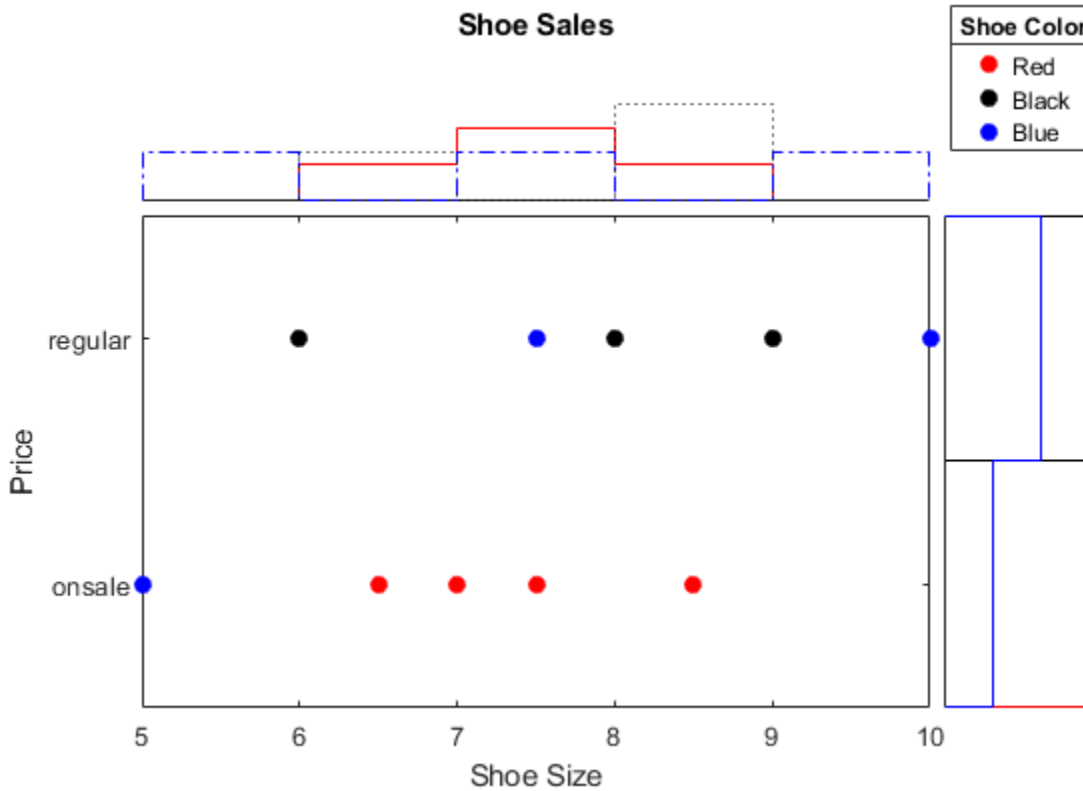
Create a scatter plot with marginal histograms using arrays of shoe data. Group the data according to shoe color, and customize properties of the scatter histogram chart.

Create arrays of data. Then, create a scatter histogram chart to visualize the data. Use custom labels along the *x*-axis and *y*-axis to specify the variable names of the first two input arguments. You can specify the title, axis labels, and legend title by setting properties of the `ScatterHistogramChart` object.

```
xvalues = [7 6 5 6.5 9 7.5 8.5 7.5 10 8];
yvalues = categorical({'onsale','regular','onsale','onsale', ...
    'regular','regular','onsale','onsale','regular','regular'});
grpvalues = {'Red','Black','Blue','Red','Black','Blue','Red', ...
    'Red','Blue','Black'};
s = scatterhistogram(xvalues,yvalues,'GroupData',grpvalues);

s.Title = 'Shoe Sales';
s.XLabel = 'Shoe Size';
s.YLabel = 'Price';
s.LegendTitle = 'Shoe Color';
```

Change the colors in the scatter histogram chart to match the group labels. Change the histogram bin widths to be the same for all groups.

```
s.Color = {'Red','Black','Blue'};
s.BinWidths = 1;
```

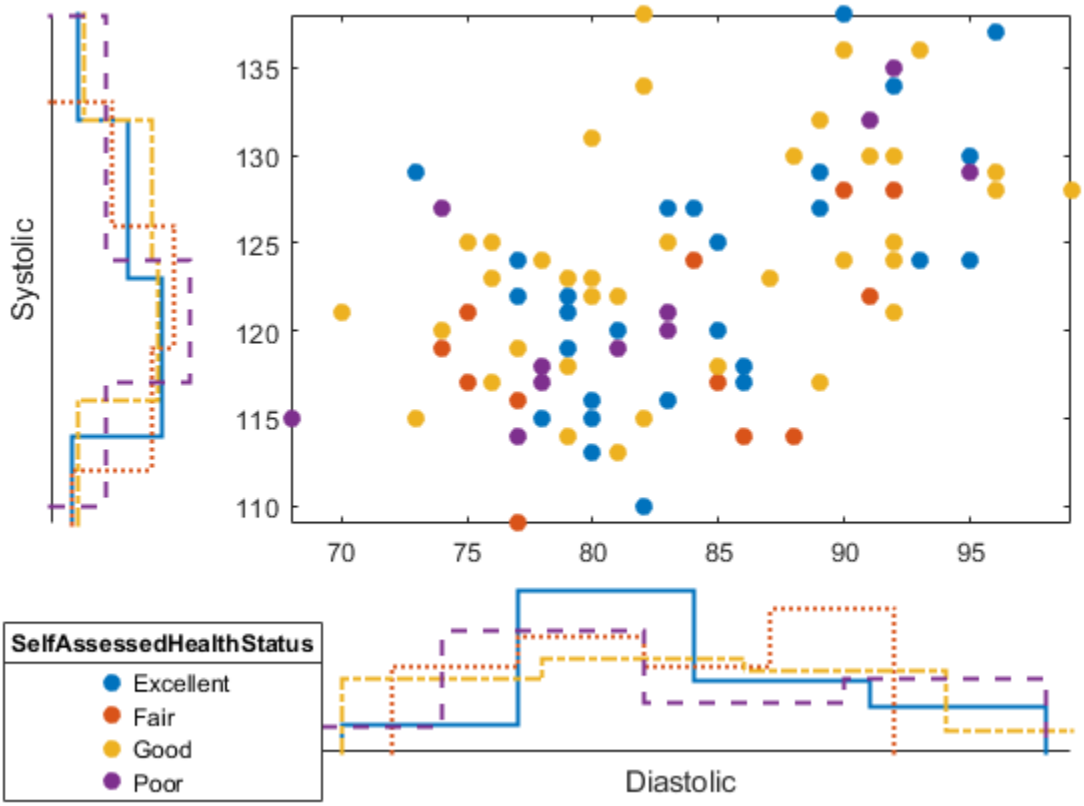


Specify Scatter Histogram Chart Appearance

Create a scatter plot with marginal histograms. Specify the number of bins and line widths of the histograms, the location of the scatter plot, and the legend visibility.

Load the `patients` data set and create a scatter histogram chart from the data. Compare the patients' `Diastolic` and `Systolic` values, and group the data according to the patients' `SelfAssessedHealthStatus` values. Adjust the histograms by specifying the `NumBins` and `LineWidth` options. Place the scatter plot in the `'NorthEast'` location of the figure by using the `ScatterPlotLocation` option. Ensure the legend is visible by specifying the `LegendVisible` option as `'on'`.

```
load patients
tbl = table(LastName,Diastolic,Systolic,SelfAssessedHealthStatus);
s = scatterhistogram(tbl,'Diastolic','Systolic','GroupVariable','SelfAssessedHealthStat
    'NumBins',4,'LineWidth',1.5,'ScatterPlotLocation','NorthEast','LegendVisible','on')
```

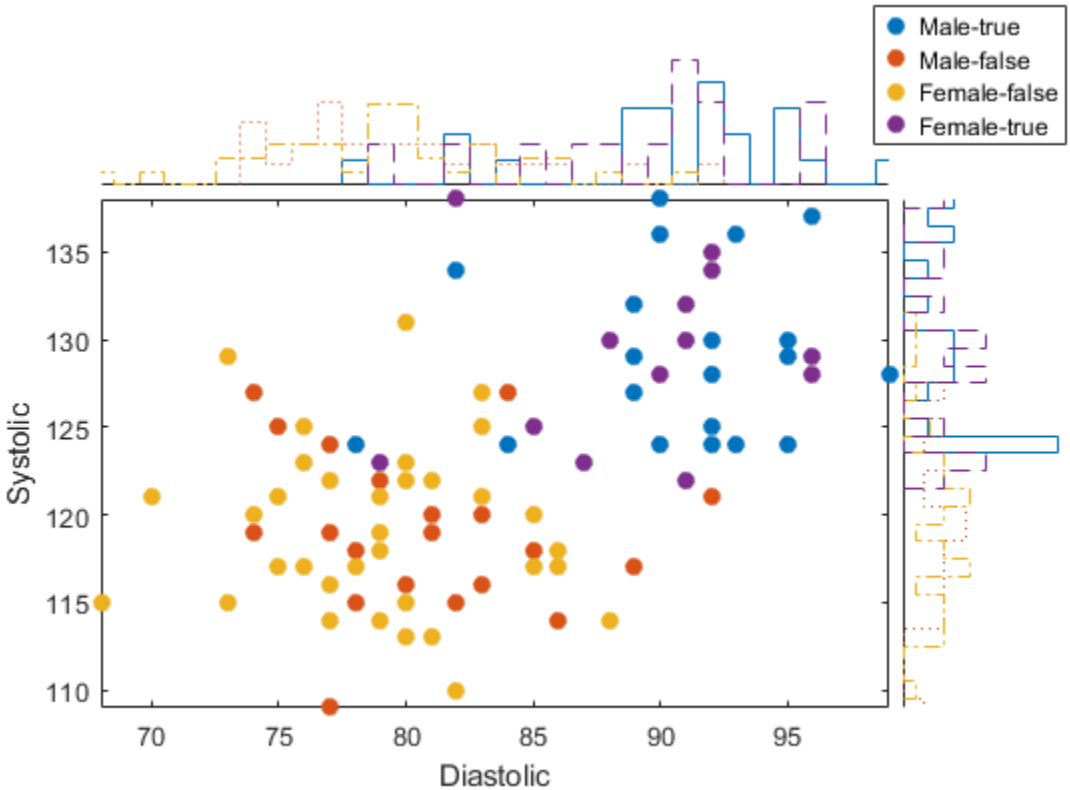


Group Data Using Two Variables

Create a scatter plot with marginal histograms. Group the data by using a combination of two different variables.

Load the `patients` data set. Combine the `Smoker` and `Gender` data to create a new variable. Create a scatter histogram chart that compares the `Diastolic` and `Systolic` values of the patients. Use the new variable `SmokerGender` to group the data in the scatter histogram chart.

```
load patients
[idx,genderStatus,smokerStatus] = findgroups(string(Gender),string(Smoker));
SmokerGender = strcat(genderStatus(idx),"-",smokerStatus(idx));
s = scatterhistogram(Diastolic,Systolic,'GroupData',SmokerGender,'LegendVisible','on');
xlabel('Diastolic')
ylabel('Systolic')
```

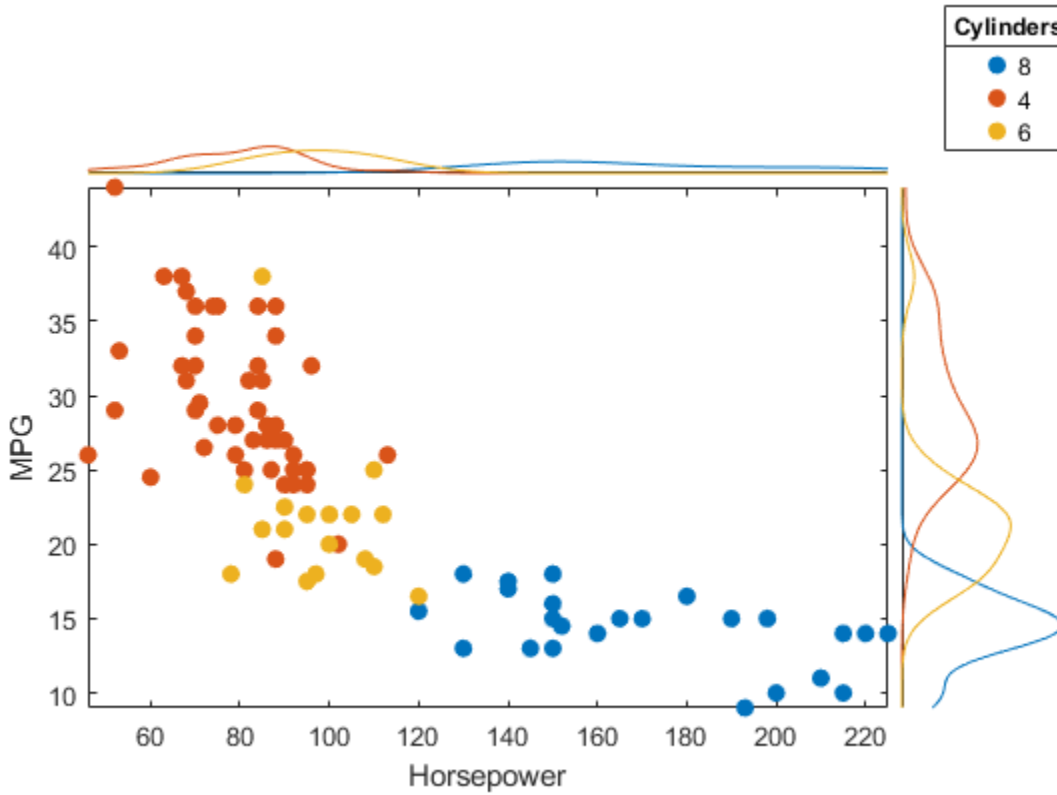


Specify Kernel Density Histograms

Create a scatter plot with kernel density marginal histograms. This example requires a Statistics and Machine Learning Toolbox license.

Load the `carsmall` data set and create a scatter histogram chart from the data. Compare the `Horsepower` and `MPG` values. Use the number of cylinders to group the data by setting the `GroupVariable` option to `Cylinders`. Specify kernel density histograms by setting the `HistogramDisplayStyle` option to `'smooth'`. Specify a solid line for all the histograms by setting the `LineStyle` option to `'-'`.

```
load carsmall
tbl = table(Horsepower,MPG,Cylinders);
s = scatterhistogram(tbl,'Horsepower','MPG', ...
    'GroupVariable','Cylinders','HistogramDisplayStyle','smooth', ...
    'LineStyle','-');
```



Input Arguments

`tbl` — Source table
table

Source table, specified as a table.

You can create a table from workspace variables using the `table` function, or you can import data as a table using the `readtable` function.

The `SourceTable` property of the `ScatterHistogramChart` object stores the source table.

xvar — Table variable for *x*-axis
character vector | string scalar | numeric scalar | logical vector

Table variable for *x*-axis, specified in one of these forms:

- Character vector or string scalar — Indicating one of the variable names. For example, `scatterhistogram(tbl,'Acceleration','Horsepower')` selects the variable named `'Acceleration'` for the *x*-axis.
- Numeric scalar — Indicating the table variable index. For example, `scatterhistogram(tbl,5,3)` selects the fifth variable in the table for the *x*-axis.
- Logical vector — Containing one `true` element.

The values associated with your table variable must be of a numeric type or `categorical`.

The `XVariable` property of the `ScatterHistogramChart` object stores the selected variable name.

yvar — Table variable for *y*-axis
character vector | string scalar | numeric scalar | logical vector

Table variable for *y*-axis, specified in one of these forms:

- Character vector or string scalar — Indicating one of the variable names. For example, `scatterhistogram(tbl,'Acceleration','Horsepower')` selects the variable named `'Horsepower'` for the *y*-axis.
- Numeric scalar — Indicating the table variable index. For example, `scatterhistogram(tbl,5,3)` selects the third variable in the table for the *y*-axis.
- Logical vector — Containing one `true` element.

The values associated with your table variable must be of a numeric type or `categorical`.

The `YVariable` property of the `ScatterHistogramChart` object stores the selected variable name.

grpvar — Table variable for grouping data
character vector | string scalar | numeric scalar | logical vector

Table variable for grouping data, specified in one of these forms:

- Character vector or string scalar — Indicating one of the variable names
- Numeric scalar — Indicating the table variable index
- Logical vector — Containing one `true` element

The values associated with your table variable must form a numeric vector, logical vector, categorical array, string array, or cell array of character vectors.

`grpvar` splits the data in `xvar` and `yvar` into unique groups. Each group has a default color and an independent histogram in each axis. In the legend, `scatterhistogram` displays the group names in order of their first appearance in `GroupData`.

Example: `'Model_Year'`

Example: 2

`xvalues` — Values appearing along *x*-axis
numeric vector | categorical array

Values appearing along the *x*-axis, specified as a numeric vector or categorical array.

The `XData` property of the `ScatterHistogramChart` object stores the `xvalues` data.

Example: `[0.5 4.3 2.4 5.6 3.4]`

Example:
`categorical({'small','medium','small','large','medium','small'})`

`yvalues` — Values appearing along *y*-axis
numeric vector | categorical array

Values appearing along the *y*-axis, specified as a numeric vector or categorical array.

The `YData` property of the `ScatterHistogramChart` object stores the `yvalues` data.

Example: `[0.5 4.3 2.4 5.6 3.4]`

Example:
`categorical({'small','medium','small','large','medium','small'})`

`grpvalues` — Group values
numeric vector | logical vector | categorical array | string array | cell array of character vectors

Group values for the scatter plot and the corresponding marginal histograms, specified as a numeric vector, logical vector, categorical array, string array, or cell array of character vectors.

`grpvalues` splits the data in `xvalues` and `yvalues` into unique groups. Each group has a default color and an independent histogram in each axis. In the legend, `scatterhistogram` displays the group names in order of their first appearance in `GroupData`.

Example: `[1 2 1 3 2 1 3]`

Example: `categorical({'blue','green','green','blue','green'})`

`parent` — Parent container
`Figure` object | `Panel` object | `Tab` object | `TiledChartLayout` object

Parent container, specified as a `Figure`, `Panel`, `Tab`, or `TiledChartLayout` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `scatterhistogram(tbl,xvar,yvar,'GroupVariable',grpvar,'HistogramDisplayStyle','stairs')` specifies `grpvar` as the grouping variable and displays stairstep plots next to the scatter plot.

Note The properties listed here are only a subset. For a complete list, see ScatterHistogramChart.

`Title` — Chart title
`''` (default) | character vector | string array | cell array of character vectors | categorical array

Chart title, specified as a character vector, string array, cell array of character vectors, or categorical array. The default chart has no title.

To create a multiline title, specify a string array or cell array of character vectors. Each element in the array corresponds to a line of text.

If you specify the title as a categorical array, MATLAB uses the values in the array, not the categories.

Example: `s = scatterhistogram(__,'Title','My Title Text')`

Example: `s.Title = 'My Title Text'`

Example: `s.Title = {'My','Title'}`

`NumBins` — Number of histogram bins
positive integer scalar | positive integer column vector | positive integer matrix

Number of histogram bins, specified as a positive integer scalar, 2-by-1 positive integer vector, or 2-by-*n* positive integer matrix, where *n* is the number of groups in `GroupData`.

Specified Value	Description
scalar	The value is the number of bins for the *x* and *y* histograms.
2-by-1 vector	The first value is the number of bins for the *x* data, and the second value is the number of bins for the *y* data.
2-by-*n* matrix	The `(1,j)` value is the number of bins for the histogram of the *x* data that is in the `j`th group. Similarly, the `(2,j)` value is the number of bins for the histogram of the *y* data that is in the `j`th group.

`scatterhistogram` uses the `'BinMethod','auto'` name-value pair argument of `histogram` to determine the default `NumBins` and `BinWidths` values.

You cannot change `NumBins` for categorical data.

Example: `s = scatterhistogram(__,'NumBins',20)`

Example: `s.NumBins = [10; 15]`

`HistogramDisplayStyle` — Histogram display style
`'stairs'` (default) | `'bar'` | `'smooth'`

Histogram display style, specified as one of these options.

Display Style	Description
`'stairs'`	Display a stairstep plot that shows the outline of the histogram without filling the bars.
`'bar'`	Display a histogram bar plot.
`'smooth'`	Display a smooth plot generated through kernel density estimates. This option requires a Statistics and Machine Learning Toolbox license.

`scatterhistogram` uses the `'pdf'` type of normalization to generate the histograms. For more information, see the `'Normalization'` name-value pair argument of `histogram`.

Example: `s = scatterhistogram(__,'HistogramDisplayStyle','smooth')`

Example: `s.HistogramDisplayStyle = 'bar'`

`LineWidth` — Histogram line width
positive scalar | positive vector

Histogram line width, specified as a positive scalar or positive vector in points. By default, `scatterhistogram` assigns a line width of `0.5` to each histogram plot line.

When the total number of groups exceeds the number of specified line widths, `scatterhistogram` cycles through the specified line widths.

Example: `s = scatterhistogram(__,'LineWidth',0.75)`

Example: `s.LineWidth = [0.5 0.75 0.5]`

`MarkerStyle` — Marker symbol
character vector | string array | cell array of character vectors

Marker symbol for each scatter plot group, specified in one of these forms:

- Character vector designating a marker style
- String array or cell array of character vectors designating one or more marker styles

Choose among these marker options.

Value	Description
`'o'`	Circle

Value	Description
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

By default, `scatterhistogram` assigns the marker symbol `'o'` to each group in the scatter plot. When the total number of groups exceeds the number of specified symbols, `scatterhistogram` cycles through the specified symbols.

Example: `s = scatterhistogram(__,'MarkerStyle','x')`

Example: `s.MarkerStyle = {'x','o'}`

`ScatterPlotLocation` — Location of scatter plot
`'SouthWest'` (default) | `'SouthEast'` | `'NorthEast'` | `'NorthWest'`

Location of the scatter plot, specified as one of these options.

Location	Description
`'SouthWest'`	Plot the histograms above and to the right of the scatter plot.
`'SouthEast'`	Plot the histograms above and to the left of the scatter plot.
`'NorthEast'`	Plot the histograms below and to the left of the scatter plot.

Location	Description
`'NorthWest'`	Plot the histograms below and to the right of the scatter plot.

Example: `s = scatterhistogram(__,'ScatterPlotLocation','NorthEast')`

Example: `s.ScatterPlotLocation = 'SouthEast'`

`LegendVisible` — State of legend visibility
`'on'` | `'off'`

State of legend visibility, specified as `'on'` or `'off'`. Set `LegendVisible` to `'on'` to display the legend or `'off'` to hide the legend.

If `GroupData` is empty (`[]`) or contains a single group, then `scatterhistogram` does not display a legend. Otherwise, `scatterhistogram` displays a legend by default, unless the legend overlaps the scatter plot or marginal histograms.

In the legend, `scatterhistogram` displays the group names in order of their first appearance in `GroupData`.

Example: `s = scatterhistogram(__,'LegendVisible','on')`

Example: `s.LegendVisible = 'off'`

Tips

- To interactively explore the data in your `ScatterHistogramChart` object, use these options. Some of these options are not available in the Live Editor.
  - Zoom/pan — Use the scroll wheel or the **+** and **-** buttons to zoom. Click and drag the scatter plot to pan. `scatterhistogram` updates the marginal histograms based on the data within the current scatter plot limits.
  - Data tips — Hover over the scatter plot or marginal histograms to display a data tip.

See Also

Functions
`categorical` | `histogram` | `scatter` | `table`

Properties
ScatterHistogramChart

Topics
"Access Data in Tables"

Introduced in R2018b

ScatterHistogramChart Properties

Control scatter histogram chart appearance and behavior

Description

`ScatterHistogramChart` properties control the appearance and behavior of a `ScatterHistogramChart` object. By changing property values, you can modify certain aspects of the chart display. For example, you can add a title:

```
s = scatterhistogram(rand(10,1),rand(10,1));
s.Title = 'My Title';
```

Properties

Labels

`Title` — Chart title

`''` (default) | character vector | string array | cell array of character vectors | categorical array

Chart title, specified as a character vector, string array, cell array of character vectors, or categorical array. The default chart has no title.

To create a multiline title, specify a string array or cell array of character vectors. Each element in the array corresponds to a line of text.

If you specify the title as a categorical array, MATLAB uses the values in the array, not the categories.

Example: `s = scatterhistogram(__,'Title','My Title Text')`

Example: `s.Title = 'My Title Text'`

Example: `s.Title = {'My','Title'}`

`XLabel` — Label for *x*-axis

character vector | string array | cell array of character vectors | categorical array

Label for the *x*-axis, specified as a character vector, string array, cell array of character vectors, or categorical array. Use `''` for no label.

To create a multiline label, specify a string array or cell array of character vectors. Each element in the array corresponds to a line of text.

If you specify the label as a categorical array, MATLAB uses the values in the array, not the categories.

Example: `s = scatterhistogram(__,'XLabel','My Label')`

Example: `s.XLabel = 'My Label'`

Example: `s.XLabel = {'My','Label'}`

`YLabel` — Label for *y*-axis
character vector | string array | cell array of character vectors | categorical array

Label for the *y*-axis, specified as a character vector, string array, cell array of character vectors, or categorical array. Use `''` for no label.

To create a multiline label, specify a string array or cell array of character vectors. Each element in the array corresponds to a line of text.

If you specify the label as a categorical array, MATLAB uses the values in the array, not the categories.

Example: `s = scatterhistogram(__,'YLabel','My Label')`

Example: `s.YLabel = 'My Label'`

Example: `s.YLabel = {'My','Label'}`

`LegendTitle` — Legend title
character vector | string array | cell array of character vectors | categorical array

Legend title, specified as a character vector, string array, cell array of character vectors, or categorical array. Use `''` for no title.

To create a multiline title, specify a string array or cell array of character vectors. Each element in the array corresponds to a line of text.

If you specify the title as a categorical array, MATLAB uses the values in the array, not the categories.

Example: `s = scatterhistogram(__,'LegendTitle','My Title Text')`

Example: `s.LegendTitle = 'My Title Text'`

Example: `s.LegendTitle = {'My','Title'}`

Histograms

`NumBins` — Number of histogram bins

positive integer scalar | positive integer column vector | positive integer matrix

Number of histogram bins, specified as a positive integer scalar, 2-by-1 positive integer vector, or 2-by-*n* positive integer matrix, where *n* is the number of groups in `GroupData`.

Specified Value	Description
scalar	The value is the number of bins for the *x* and *y* histograms.
2-by-1 vector	The first value is the number of bins for the *x* data, and the second value is the number of bins for the *y* data.
2-by-*n* matrix	The `(1,j)` value is the number of bins for the histogram of the *x* data that is in the `j`th group. Similarly, the `(2,j)` value is the number of bins for the histogram of the *y* data that is in the `j`th group.

`scatterhistogram` uses the `'BinMethod','auto'` name-value pair argument of `histogram` to determine the default `NumBins` and `BinWidths` values.

You cannot change `NumBins` for categorical data.

Example: `s = scatterhistogram(__,'NumBins',20)`

Example: `s.NumBins = [10; 15]`

`BinWidths` — Histogram bin widths

positive scalar | positive column vector | positive matrix

Histogram bin widths, specified as a positive scalar, 2-by-1 positive vector, or 2-by-*n* positive matrix, where *n* is the number of groups in `GroupData`.

Specified Value	Description
scalar	The value is the bin width for the *x* and *y* histograms.
2-by-1 vector	The first value is the bin width for the *x* data, and the second value is the bin width for the *y* data.

Specified Value	Description
2-by-*n* matrix	The `(1,j)` value is the bin width for the histogram of the *x* data that is in the `j`th group. Similarly, the `(2,j)` value is the bin width for the histogram of the *y* data that is in the `j`th group.

`scatterhistogram` uses the `'BinMethod','auto'` name-value pair argument of `histogram` to determine the default `NumBins` and `BinWidths` values. The `BinWidths` values for categorical data are always `0`.

If you set `BinWidths`, then `scatterhistogram` ignores the `NumBins` value.

Example: `s = scatterhistogram(__,'BinWidths',0.5)`

Example: `s.BinWidths = [1.5; 2]`

`XHistogramDirection` — Direction of *x* data histograms
`'up'` (default) | `'down'`

Direction of the *x* data histograms, specified as `'up'` or `'down'`. If the `XHistogramDirection` value is `'up'`, then the *x* data histograms have bars directed upwards. If the `XHistogramDirection` value is `'down'`, then the *x* data histograms have bars directed downwards.

Example: `s = scatterhistogram(__,'XHistogramDirection','down')`

Example: `s.XHistogramDirection = 'down'`

`YHistogramDirection` — Direction of *y* data histograms
`'right'` (default) | `'left'`

Direction of the *y* data histograms, specified as `'right'` or `'left'`. If the `YHistogramDirection` value is `'right'`, then the *y* data histograms have bars directed rightwards. If the `YHistogramDirection` value is `'left'`, then the *y* data histograms have bars directed leftwards.

Example: `s = scatterhistogram(__,'YHistogramDirection','left')`

Example: `s.YHistogramDirection = 'left'`

`HistogramDisplayStyle` — Histogram display style
`'stairs'` (default) | `'bar'` | `'smooth'`

Histogram display style, specified as one of these options.

Display Style	Description
`'stairs'`	Display a stairstep plot that shows the outline of the histogram without filling the bars.
`'bar'`	Display a histogram bar plot.
`'smooth'`	Display a smooth plot generated through kernel density estimates. This option requires a Statistics and Machine Learning Toolbox license.

`scatterhistogram` uses the `'pdf'` type of normalization to generate the histograms. For more information, see the `'Normalization'` name-value pair argument of `histogram`.

Example: `s = scatterhistogram(__,'HistogramDisplayStyle','smooth')`

Example: `s.HistogramDisplayStyle = 'bar'`

`LineStyle` — Histogram line style

character vector | string array | cell array of character vectors

Histogram line style, specified in one of these forms:

- Character vector designating one line style
- String array or cell array of character vectors designating one or more line styles

Choose among these line style options.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

When the total number of groups exceeds the number of specified line styles, `scatterhistogram` cycles through the specified line styles.

Example: `s = scatterhistogram(__,'LineStyle',':')`

Example: `s.LineStyle = {':','-','-.'}`

`LineWidth` — Histogram line width

positive scalar | positive vector

Histogram line width, specified as a positive scalar or positive vector in points. By default, `scatterhistogram` assigns a line width of `0.5` to each histogram plot line.

When the total number of groups exceeds the number of specified line widths, `scatterhistogram` cycles through the specified line widths.

Example: `s = scatterhistogram(__,'LineWidth',0.75)`

Example: `s.LineWidth = [0.5 0.75 0.5]`

Color and Font

`Color` — Group color

character vector | string array | cell array of character vectors | matrix of RGB values

Group color, specified in one of these forms:

- Character vector designating a color name.
- String array or cell array of character vectors designating one or more color names.
- Three-column matrix of RGB values in the range [0,1]. The three columns represent the R value, G value, and B value, respectively.

Choose among these predefined colors and their equivalent RGB triplets.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`

By default, `scatterhistogram` assigns a maximum of seven unique group colors. When the total number of groups exceeds the number of specified colors, `scatterhistogram` cycles through the specified colors.

Example: `s = scatterhistogram(__,'Color',{'blue','green',red'})`

Example: `s.Color = [0 0 1; 0 0.5 0.5; 0.5 0.5 0.5]`

`FontName` — Font name
system-supported font name

Font name, specified as a system-supported font name. The same font is used for the title, axis labels, legend title, and group names. The default font depends on the specific operating system and locale.

Example: `s = scatterhistogram(__,'FontName','Cambria')`

Example: `s.FontName = 'Cambria'`

`FontSize` — Font size
scalar numeric value

Font size, specified as a scalar value. `FontSize` is the same for the title, axis labels, legend title, and group names. The default font size depends on the specific operating system and locale.

As you adjust the size of plot elements, the software automatically updates the font size. However, changing the `FontSize` property disables this automatic resizing.

Example: `s = scatterhistogram(__,'FontSize',12)`

Example: `s.FontSize = 12`

Markers

`MarkerStyle` — Marker symbol
character vector | string array | cell array of character vectors

Marker symbol for each scatter plot group, specified in one of these forms:

- Character vector designating a marker style
- String array or cell array of character vectors designating one or more marker styles

Choose among these marker options.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

By default, `scatterhistogram` assigns the marker symbol `'o'` to each group in the scatter plot. When the total number of groups exceeds the number of specified symbols, `scatterhistogram` cycles through the specified symbols.

Example: `s = scatterhistogram(__,'MarkerStyle','x')`

Example: `s.MarkerStyle = {'x','o'}`

`MarkerSize` — Marker size

nonnegative scalar | nonnegative vector

Marker size for each scatter plot group, specified as a nonnegative scalar or nonnegative vector, with values measured in points. By default, `scatterhistogram` assigns `36` as the marker size for each group in the scatter plot. When the total number of groups exceeds the number of specified values, `scatterhistogram` cycles through the specified values.

Example: `s = scatterhistogram(__,'MarkerSize',30)`

Example: `s.MarkerSize = 40`

MarkerFilled — State of marker face fill

'on' (default) | 'off'

State of marker face fill, specified as 'on' or 'off'. If MarkerFilled is set to 'on', then scatterhistogram fills the interior of the markers in the scatter plot. If MarkerFilled is set to 'off', then scatterhistogram leaves the interior of the scatter plot markers empty.

Example: s = scatterhistogram(__,'MarkerFilled','off')

Example: s.MarkerFilled = 'off'

MarkerAlpha — Marker transparency

numeric scalar | numeric vector

Marker transparency for each scatter plot group, specified as a numeric scalar or numeric vector with values between 0 and 1. Values closer to 0 specify more transparent markers, and values closer to 1 specify more opaque markers. By default, scatterhistogram assigns a MarkerAlpha value of 1 to all markers in the scatter plot.

Example: s = scatterhistogram(__,'MarkerAlpha',0.75)

Example: s.MarkerAlpha = [0.2 0.7 0.4]

Layout

ScatterPlotLocation — Location of scatter plot

'SouthWest' (default) | 'SouthEast' | 'NorthEast' | 'NorthWest'

Location of the scatter plot, specified as one of these options.

Location	Description
'SouthWest'	Plot the histograms above and to the right of the scatter plot.
'SouthEast'	Plot the histograms above and to the left of the scatter plot.
'NorthEast'	Plot the histograms below and to the left of the scatter plot.
'NorthWest'	Plot the histograms below and to the right of the scatter plot.

Example: s = scatterhistogram(__,'ScatterPlotLocation','NorthEast')

Example: `s.ScatterPlotLocation = 'SouthEast'`

`ScatterPlotProportion` — Ratio of scatter plot length to chart length
`0.75` (default) | numeric scalar between 0 and 1

Ratio of the scatter plot length to the overall chart length, specified as a numeric scalar between 0 and 1. The `ScatterPlotProportion` value applies to both `x` and `y` axes.

Example: `s = scatterhistogram(__,'ScatterPlotProportion',0.7)`

Example: `s.ScatterPlotProportion = 0.6`

`LegendVisible` — State of legend visibility
`'on'` | `'off'`

State of legend visibility, specified as `'on'` or `'off'`. Set `LegendVisible` to `'on'` to display the legend or `'off'` to hide the legend.

If `GroupData` is empty (`[]`) or contains a single group, then `scatterhistogram` does not display a legend. Otherwise, `scatterhistogram` displays a legend by default, unless the legend overlaps the scatter plot or marginal histograms.

In the legend, `scatterhistogram` displays the group names in order of their first appearance in `GroupData`.

Example: `s = scatterhistogram(__,'LegendVisible','on')`

Example: `s.LegendVisible = 'off'`

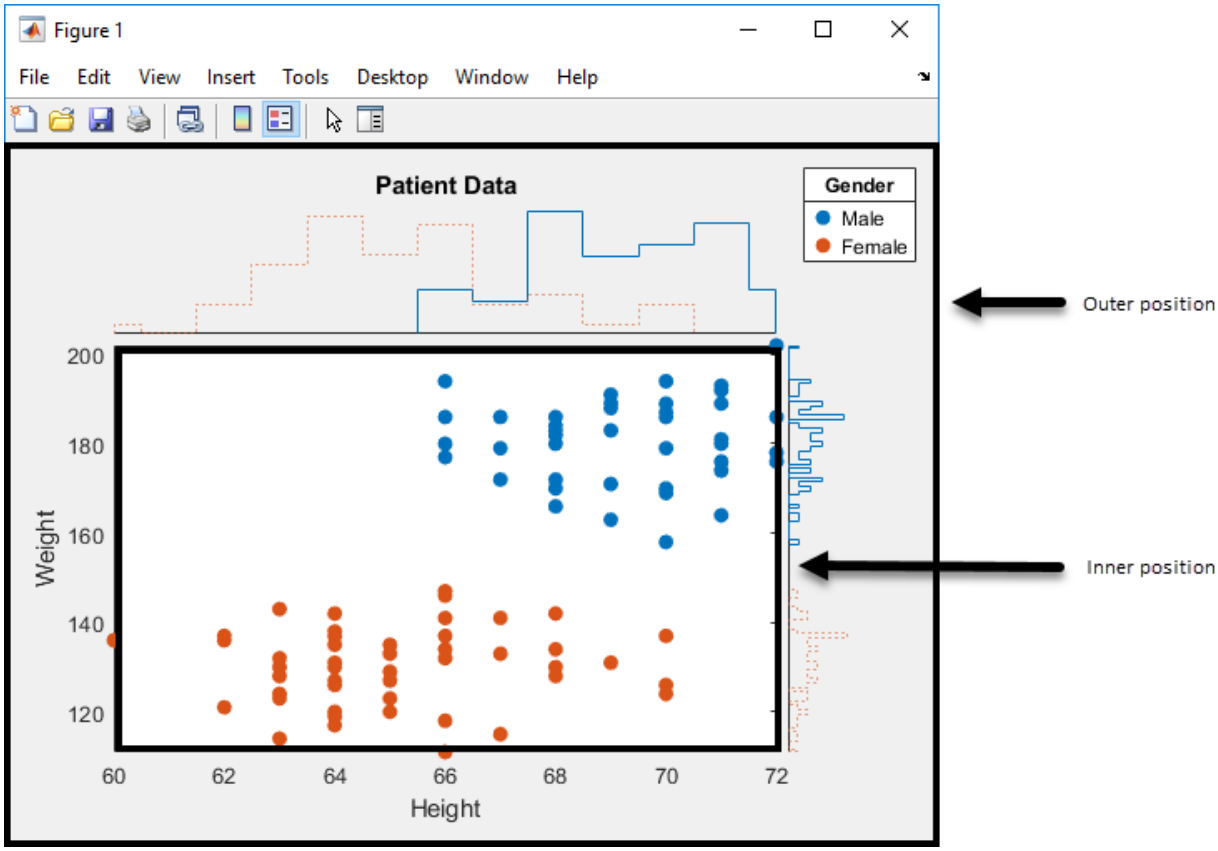
Position

`ActivePositionProperty` — Position property to hold constant
`'outerposition'` | `'innerposition'`

Position property to hold constant when adding, removing, or changing decorations, specified as one of the following values:

- `'outerposition'` — The `OuterPosition` property remains constant when you add, remove, or change decorations such as a title or an axis label. If any positional adjustments are needed, MATLAB adjusts the `InnerPosition` property.
- `'innerposition'` — The `InnerPosition` property remains constant when you add, remove, or change decorations such as a title or an axis label. If any positional adjustments are needed, MATLAB adjusts the `OuterPosition` property.

This figure shows the `innerposition` and `outerposition` definitions for `ScatterHistogramChart`.



Example: `s.ActivePositionProperty = 'outerposition'`

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`InnerPosition` — Inner size and position

four-element numeric vector

Inner size and position of the chart within the parent container (typically a figure, panel, or tab), specified as a four-element numeric vector of the form `[left bottom width height]`. The inner position includes only the scatter plot.

- The `left` and `bottom` elements define the distance from the lower left corner of the container to the lower left corner of the scatter plot.
- The `width` and `height` elements are the dimensions of the scatter plot.

For an illustration, see “ActivePositionProperty” on page 1-0 .

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`OuterPosition` — Outer size and position

`[0 0 1 1]` (default) | four-element numeric vector

Outer size and position of the full scatter histogram chart within the parent container (typically a figure, panel, or tab), specified as a four-element numeric vector of the form `[left bottom width height]`. The default value of `[0 0 1 1]` includes the whole interior of the container.

For an illustration, see “ActivePositionProperty” on page 1-0 .

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`Position` — Inner size and position

four-element numeric vector

Inner size and position of the chart within the parent container (typically a figure, panel, or tab), specified as a four-element numeric vector of the form `[left bottom width height]`. This property is equivalent to the “InnerPosition” on page 1-0 property.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

Units — Position units
`'normalized'` (default) | `'inches'` | `'centimeters'` | `'points'` | `'pixels'` | `'characters'`

Position units, specified as one of these values.

Units	Description
`'normalized'` (default)	Normalized with respect to the container, which is typically the figure or a panel. The lower left corner of the container maps to `(0,0)`, and the upper right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the default uicontrol font of the graphics root object: • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.
`'points'`	Typography points. One point equals 1/72 inch.
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96 inch. • On Macintosh systems, a pixel is 1/72 inch. On Linux systems, the size of a pixel is determined by your system resolution.

When specifying the units as a name-value pair during object creation, you must set the `Units` property before specifying the properties that you want to use these units, such as `OuterPosition`.

`Layout` — Layout options

empty `LayoutOptions` array (default) | `TiledChartLayoutOptions` object

Layout options, specified as a `TiledChartLayoutOptions` object. This property specifies options when the chart is a child of a tiled chart layout. Use this property to position the chart within the layout by setting the `Tile` and `TileSpan` properties on the `TiledChartLayoutOptions` object.

For example, this code places chart `c` in the third tile of a tiled chart layout.

```
c.Layout.Tile = 3;
```

To make the chart span multiple tiles, specify the `TileSpan` property as a two-element vector. For example, this chart spans 2 rows and 3 columns of tiles.

```
c.Layout.TileSpan = [2 3];
```

If the chart is not a child of a tiled chart layout (for example, if it is a child of a figure or panel), then this property is empty and has no effect.

`Visible` — State of object visibility

`'on'` (default) | `'off'`

State of object visibility, specified as one of these values:

- `'on'` — Display the `ScatterHistogramChart` object.
- `'off'` — Hide the `ScatterHistogramChart` object without deleting it. You can still access the properties of an invisible `ScatterHistogramChart` object.

Data and Limits

`SourceTable` — Source table

table

Source table, specified as a table.

You can create a table from workspace variables using the `table` function, or you can import data as a table using the `readtable` function.

Note The property is ignored and read-only when you use arrays instead of tabular data.

`XVariable` — Table variable for *x*-axis

character vector | string scalar | numeric scalar | logical vector

Table variable for *x*-axis, specified in one of these forms:

- Character vector or string scalar indicating one of the variable names
- Numeric scalar indicating the table variable index
- Logical vector containing one `true` element

The values associated with your table variable must be of a numeric type or `categorical`.

If you set the `XVariable` property value, then the `XData` property automatically updates to appropriate values.

Note The property is ignored and read-only when you use arrays instead of tabular data.

Example: `s.XVariable = 'Acceleration'` specifies the variable named `'Acceleration'`.

`YVariable` — Table variable for *y*-axis

character vector | string scalar | numeric scalar | logical vector

Table variable for *y*-axis, specified in one of these forms:

- Character vector or string scalar indicating one of the variable names
- Numeric scalar indicating the table variable index
- Logical vector containing one `true` element

The values associated with your table variable must be of a numeric type or `categorical`.

If you set the `YVariable` property value, then the `YData` property automatically updates to appropriate values.

Note The property is ignored and read-only when you use arrays instead of tabular data.

Example: `s.YVariable = 'Horsepower'` specifies the variable named `'Horsepower'`.

`GroupVariable` — Table variable for grouping data
character vector | string scalar | numeric scalar | logical vector

Table variable for grouping data, specified in one of these forms:

- Character vector or string scalar indicating one of the variable names
- Numeric scalar indicating the table variable index
- Logical vector containing one `true` element

The values associated with your table variable must form a numeric vector, logical vector, categorical array, string array, or cell array of character vectors.

`GroupVariable` splits the data in `XVariable` and `YVariable` into unique groups. Each group has a default color and an independent histogram in each axis. In the legend, `scatterhistogram` displays the group names in order of their first appearance in `GroupData`.

When you specify the group variable, MATLAB updates the `GroupData` property values.

Note This property is ignored and read-only when you use arrays instead of tabular data.

Example: `s.GroupVariable = 'Origin'`

`XData` — *x* values
numeric vector | categorical array

Values appearing along the *x*-axis, specified as a numeric vector or categorical array.

If you are using tabular data, you cannot set this property. The `XData` values automatically populate based on the table variable you select with the `XVariable` property.

Example: `s.XData = [0.5 4.3 2.4 5.6 3.4]`

`YData` — *y* values
numeric vector | categorical array

Values appearing along the *y*-axis, specified as a numeric vector or categorical array.

If you are using tabular data, you cannot set this property. The `YData` values automatically populate based on the table variable you select with the `YVariable` property.

Example: `s.YData = [0.5 4.3 2.4 5.6 3.4]`

`GroupData` — Group values
numeric vector | logical vector | categorical array | string array | cell array of character vectors

Group values for the scatter plot and the corresponding marginal histograms, specified as a numeric vector, logical vector, categorical array, string array, or cell array of character vectors.

`GroupData` splits the data in `XData` and `YData` into unique groups. Each group has a default color and an independent histogram in each axis. In the legend, `scatterhistogram` displays the group names in order of their first appearance in `GroupData`.

If you are using tabular data, you cannot set this property. The `GroupData` values automatically populate based on the table variable you select with the `GroupVariable` property.

Example: `s.GroupData = [1 2 1 3 2 1 3]`

Example: `s.GroupData = {'blue','green','green','blue','green'}`

`XLimits` — *x*-axis limits
two-element numeric vector | two-element categorical vector

x-axis limits, specified as a two-element numeric vector or two-element categorical vector. By default, the values are derived from the `XData` values.

Example: `s.XLimits = categorical({'blue','green'})`

Example: `s.XLimits = [10 50]`

`YLimits` — *y*-axis limits
two-element numeric vector | two-element categorical vector

y-axis limits, specified as a two-element numeric vector or two-element categorical vector. By default, the values are derived from the `YData` values.

Example: `s.YLimits = categorical({'blue','green'})`

Example: `s.YLimits = [10 50]`

Parent/Child

`Parent` — Parent container
`Figure` object | `Panel` object | `Tab` object | `TiledChartLayout` object

Parent container, specified as a `Figure`, `Panel`, `Tab`, or `TiledChartLayout` object.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle for `ScatterHistogramChart` in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is always invisible. This option is useful for preventing unintended changes to the UI by another function. To temporarily hide the handle during the execution of that function, set the `HandleVisibility` to `'off'`.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but allows callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return the object. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles, regardless of their `HandleVisibility` property setting.

See Also
`scatterhistogram`

Topics
"Access Property Values"

Introduced in R2018b

schur

Schur decomposition

Syntax

```
T = schur(A)
T = schur(A,flag)
[U,T] = schur(A,...)
```

Description

The `schur` function computes the Schur form of a matrix.

`T = schur(A)` returns the Schur matrix `T`.

`T = schur(A,flag)` for real matrix A, returns a Schur matrix `T` in one of two forms depending on the value of `flag`:

`'complex'`	`T` is triangular and is complex if `A` is real and has complex eigenvalues.
`'real'`	`T` has the real eigenvalues on the diagonal and the complex eigenvalues in 2-by-2 blocks on the diagonal. `'real'` is the default when `A` is real.

If `A` is complex, `schur` returns the complex Schur form in matrix `T` and `flag` is ignored. The complex Schur form is upper triangular with the eigenvalues of `A` on the diagonal.

The function `rsf2csf` converts the real Schur form to the complex Schur form.

`[U,T] = schur(A,...)` also returns a unitary matrix `U` so that `A = U*T*U'` and `U'*U = eye(size(A))`.

Examples

`H` is a 3-by-3 eigenvalue test matrix:

```
H = [ -149    -50   -154
       537    180    546
       -27     -9    -25 ]
```

Its Schur form is

```
schur(H)

ans =
     1.0000   -7.1119 -815.8706
          0    2.0000  -55.0236
          0         0    3.0000
```

The eigenvalues, which in this case are 1, 2, and 3, are on the diagonal. The fact that the off-diagonal elements are so large indicates that this matrix has poorly conditioned eigenvalues; small changes in the matrix elements produce relatively large changes in its eigenvalues.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Generated code can return a different Schur decomposition than MATLAB returns.
- When the input matrix contains a nonfinite value, the generated code does not issue an error. Instead, the output contains `NaN` values.
- Code generation does not support sparse matrix inputs for this function.

See Also

`eig` | `hess` | `qz` | `rsf2csf`

Introduced before R2006a

scatteredInterpolant

Interpolate 2-D or 3-D scattered data

Description

Use `scatteredInterpolant` to perform interpolation on a 2-D or 3-D data set of scattered data on page 1-13882. `scatteredInterpolant` returns the interpolant on page 1-13882 `F` for the given data set. You can evaluate `F` at a set of query points, such as `(xq,yq)` in 2-D, to produce interpolated values `vq = F(xq,yq)`.

Use `griddedInterpolant` to perform interpolation with gridded data on page 1-13882.

Creation

Syntax

```
F = scatteredInterpolant
F = scatteredInterpolant(x,y,v)
F = scatteredInterpolant(x,y,z,v)
F = scatteredInterpolant(P,v)
F = scatteredInterpolant( ___ ,Method)
F = scatteredInterpolant( ___ ,Method,ExtrapolationMethod)
```

Description

`F = scatteredInterpolant` creates an empty scattered data interpolant object.

`F = scatteredInterpolant(x,y,v)` creates an interpolant that fits a surface of the form $v = F(x,y)$. Vectors `x` and `y` specify the `(x,y)` coordinates of the sample points. `v` is a vector that contains the sample values associated with the points `(x,y)`.

`F = scatteredInterpolant(x,y,z,v)` creates a 3-D interpolant of the form $v = F(x,y,z)$.

`F = scatteredInterpolant(P,v)` specifies the coordinates of the sample points as an array. The rows of `P` contain the (x, y) or (x, y, z) coordinates for the values in `v`.

`F = scatteredInterpolant( ___ ,Method)` specifies an interpolation method: `'nearest'`, `'linear'`, or `'natural'`. Specify `Method` as the last input argument in any of the first three syntaxes.

`F = scatteredInterpolant( ___ ,Method,ExtrapolationMethod)` specifies both the interpolation and extrapolation methods. Pass `Method` and `ExtrapolationMethod` together as the last two input arguments in any of the first three syntaxes.

- `Method` can be: `'nearest'`, `'linear'`, or `'natural'`.
- `ExtrapolationMethod` can be: `'nearest'`, `'linear'`, or `'none'`.

Input Arguments

x, y, z — Sample points

vector

Sample points, specified as vectors of the same size as `v`. The sample points should be unique. However, if the sample points contain duplicates, `scatteredInterpolant` displays a warning and merges the duplicates into a single point.

Data Types: `double`

P — Sample points array

matrix

Sample points array, specified as an `m`-by-`n` matrix, where `m` is the number of points and `n` is the dimension of the space where the points reside. Each row of `P` contains the (x, y) or (x, y, z) coordinates of a sample point. The sample points should be unique. However, if the sample points contain duplicates, `scatteredInterpolant` displays a warning and merges the duplicates into a single point.

Data Types: `double`

v — Sample values

vector

Sample values, specified as a vector that defines the function values at the sample points, $v = F(x,y,z)$.

Data Types: `double`

`Method` — Interpolation method
`'linear'` (default) | `'nearest'` | `'natural'`

Interpolation method, specified as one of these options.

Method	Description	Continuity
`'linear'` (default)	Linear interpolation	C^0
`'nearest'`	Nearest neighbor interpolation	Discontinuous
`'natural'`	Natural neighbor interpolation	C^1 (except at sample points)

`ExtrapolationMethod` — Extrapolation method
`'linear'` | `'nearest'` | `'none'`

Extrapolation method, specified as one of these options.

ExtrapolationMethod	Description
`'linear'`	Linear extrapolation based on boundary gradients. Default when `Method` is `'linear'` or `'natural'`.
`'nearest'`	Nearest neighbor extrapolation. This method evaluates to the value of the nearest neighbor on the boundary. Default when `Method` is `'nearest'`.
`'none'`	No extrapolation. Any queries outside the convex hull of `Points` return `NaN`.

Properties

`Points` — Sample points
matrix

Sample points, specified as a matrix. The size of the matrix is `m`-by-`2` or `m`-by-`3` to represent `m` points in 2-D or 3-D space. Each row of `Points` contains the (x, y) or (x, y, z) coordinates of a unique sample point. The rows in `Points` correspond to the function values in `Values`.

Data Types: `double`

`Values` — Function values at sample points
vector

Function values at sample points, specified as a vector of values associated with each point in `Points`.

Data Types: `double`

`Method` — Interpolation method
`'linear'` (default) | `'nearest'` | `'natural'`

Interpolation method, specified as `'linear'`,`'nearest'` , or `'natural'`. See `Method` for descriptions of these methods.

`ExtrapolationMethod` — Extrapolation method
`'linear'` | `'nearest'` | `'none'`

Extrapolation method, specified as `'nearest'`, `'linear'`, or `'none'`. See `ExtrapolationMethod` for descriptions of these methods.

Data Types: `double`

Evaluate Interpolant at Specific Points

Use `scatteredInterpolant` to create the interpolant on page 1-13882, `F`. Then you can evaluate `F` at specific points using any of the following syntaxes:

```
F = scatteredInterpolant
F = scatteredInterpolant(x,y,v)
F = scatteredInterpolant(x,y,z,v)
F = scatteredInterpolant(P,v)
F = scatteredInterpolant( ___ ,Method)
F = scatteredInterpolant( ___ ,Method,ExtrapolationMethod)
Vq = F(Pq)
Vq = F(Xq,Yq)
Vq = F(Xq,Yq,Zq)
Vq = F({xq,yq})
Vq = F({xq,yq,zq})
```

- `Vq = F(Pq)` specifies the query points in the matrix `Pq`. Each row in `Pq` contains the coordinates of a query point.

- `Vq = F(Xq,Yq)` and `Vq = F(Xq,Yq,Zq)` specify the query points as two or three matrices of equal size.
- `Vq = F({xq,yq})` and `Vq = F({xq,yq,zq})` specify the query points as grid vectors on page 1-13883. Use this syntax to conserve memory when you want to query a large grid of points.

Examples

2-D Interpolation

Define some sample points and calculate the value of a trigonometric function at those locations. These points are the sample values for the interpolant.

```
t = linspace(3/4*pi,2*pi,50)';
x = [3*cos(t); 2*cos(t); 0.7*cos(t)];
y = [3*sin(t); 2*sin(t); 0.7*sin(t)];
v = repelem([-0.5; 1.5; 2],length(t));
```

Create the interpolant.

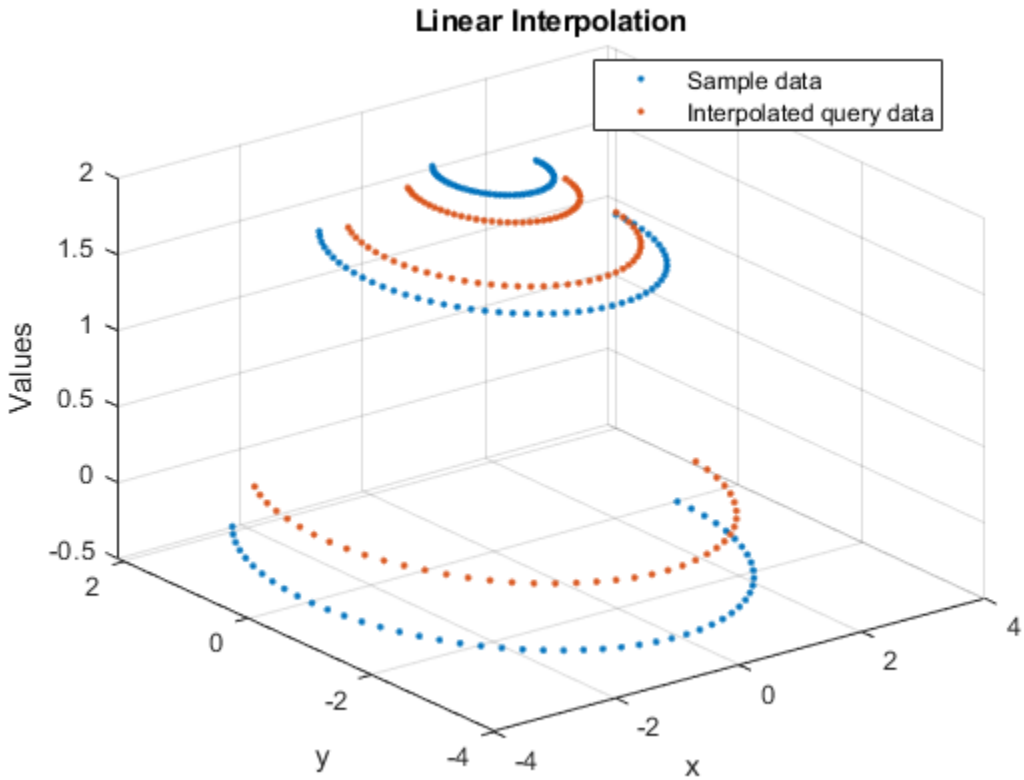
```
F = scatteredInterpolant(x,y,v);
```

Evaluate the interpolant at query locations (`xq`,`yq`).

```
tq = linspace(3/4*pi+0.2,2*pi-0.2,40)';
xq = [2.8*cos(tq); 1.7*cos(tq); cos(tq)];
yq = [2.8*sin(tq); 1.7*sin(tq); sin(tq)];
vq = F(xq,yq);
```

Plot the result.

```
plot3(x,y,v,'.',xq,yq,vq,'.'), grid on
title('Linear Interpolation')
xlabel('x'), ylabel('y'), zlabel('Values')
legend('Sample data','Interpolated query data','Location','Best')
```



3-D Interpolation

Create an interpolant for a set of scattered sample points, then evaluate the interpolant at a set of 3-D query points.

Define 200 random points and sample a trigonometric function. These points are the sample values for the interpolant.

```
P = -2.5 + 5*gallery('uniformdata',[200 3],0);
v = sin(P(:,1).^2 + P(:,2).^2 + P(:,3).^2)./(P(:,1).^2+P(:,2).^2+P(:,3).^2);
```

Create the interpolant.

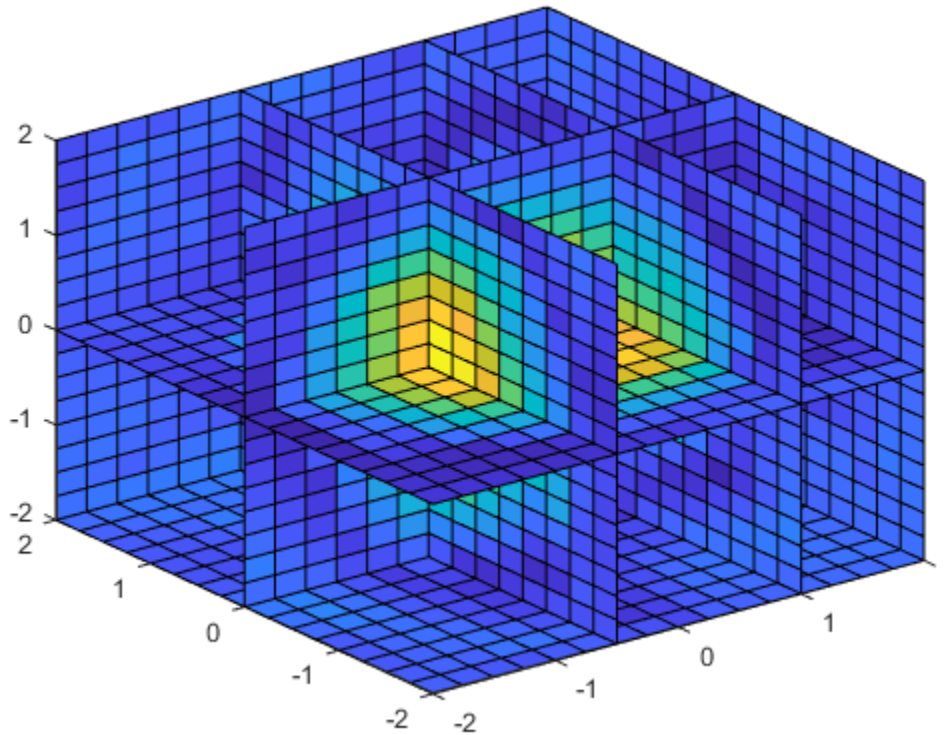
```
F = scatteredInterpolant(P,v);
```

Evaluate the interpolant at query locations (`xq`,`yq`,`zq`).

```
[xq,yq,zq] = meshgrid(-2:0.25:2);
vq = F(xq,yq,zq);
```

Plot slices of the result.

```
xslice = [-.5,1,2];
yslice = [0,2];
zslice = [-2,0];
slice(xq,yq,zq,vq,xslice,yslice,zslice)
```



Replacement of Sample Values

Replace the elements in the `Values` property when you want to change the values at the sample points. You get immediate results when you evaluate the new interpolant because the original triangulation does not change.

Create 50 random points and sample an exponential function. These points are the sample values for the interpolant.

```
x = -2.5 + 5*gallery('uniformdata',[50 1],0);
y = -2.5 + 5*gallery('uniformdata',[50 1],1);
v = x.*exp(-x.^2-y.^2);
```

Create the interpolant.

```
F = scatteredInterpolant(x,y,v)
```

```
F = 
  scatteredInterpolant with properties:

                 Points: [50x2 double]
                 Values: [50x1 double]
                 Method: 'linear'
    ExtrapolationMethod: 'linear'
```

Evaluate the interpolant at `(1.40,1.90)`.

```
F(1.40,1.90)
```

```
ans = 0.0029
```

Change the interpolant sample values and reevaluate the interpolant at the same point.

```
vnew = x.^2 + y.^2;
F.Values = vnew;
F(1.40,1.90)
```

```
ans = 6.1109
```

Compare Scattered Data Interpolation Methods

Compare the results of several different interpolation algorithms offered by `scatteredInterpolant`.

Create a sample data set of 50 scattered points. The number of points is artificially small to highlight the differences between the interpolation methods.

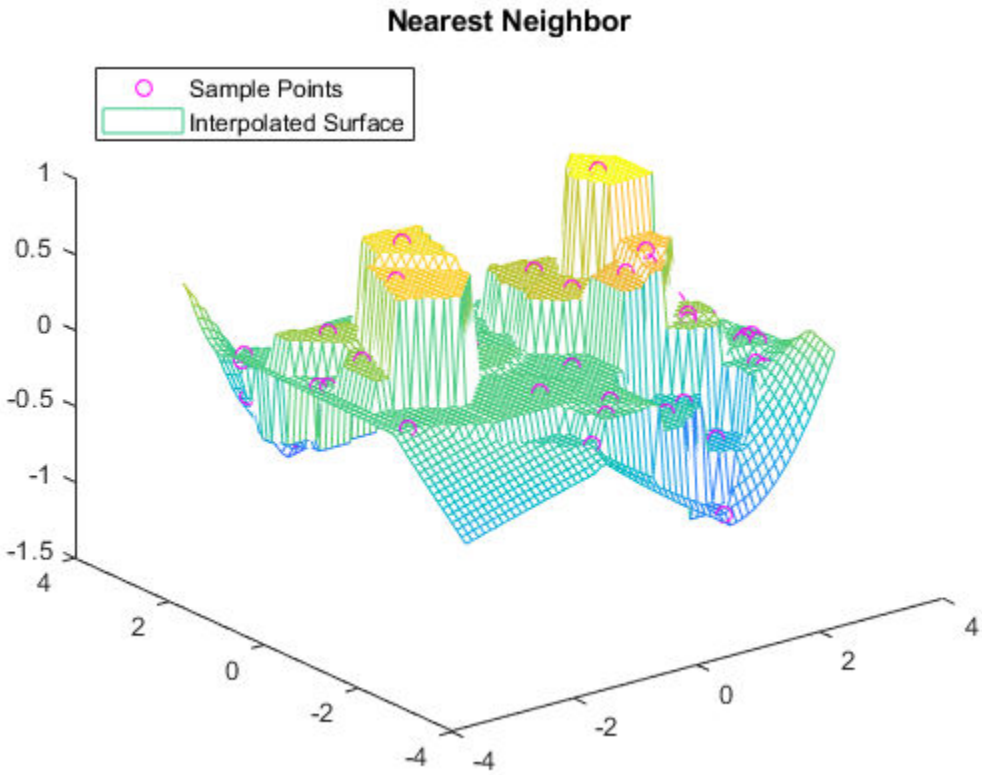
```
x = -3 + 6*rand(50,1);
y = -3 + 6*rand(50,1);
v = sin(x).^4 .* cos(y);
```

Create the interpolant and a grid of query points.

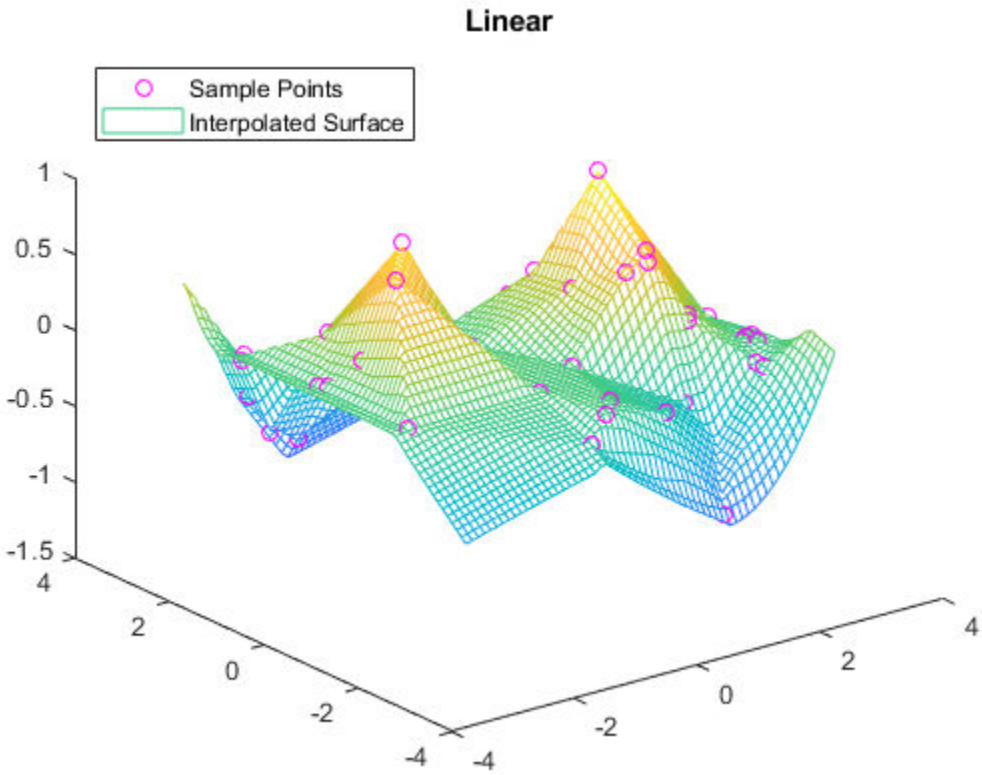
```
F = scatteredInterpolant(x,y,v);
[xq,yq] = meshgrid(-3:0.1:3);
```

Plot the results using the `'nearest'`, `'linear'`, and `'natural'` methods. Each time the interpolation method changes, you need to requery the interpolant to get the updated results.

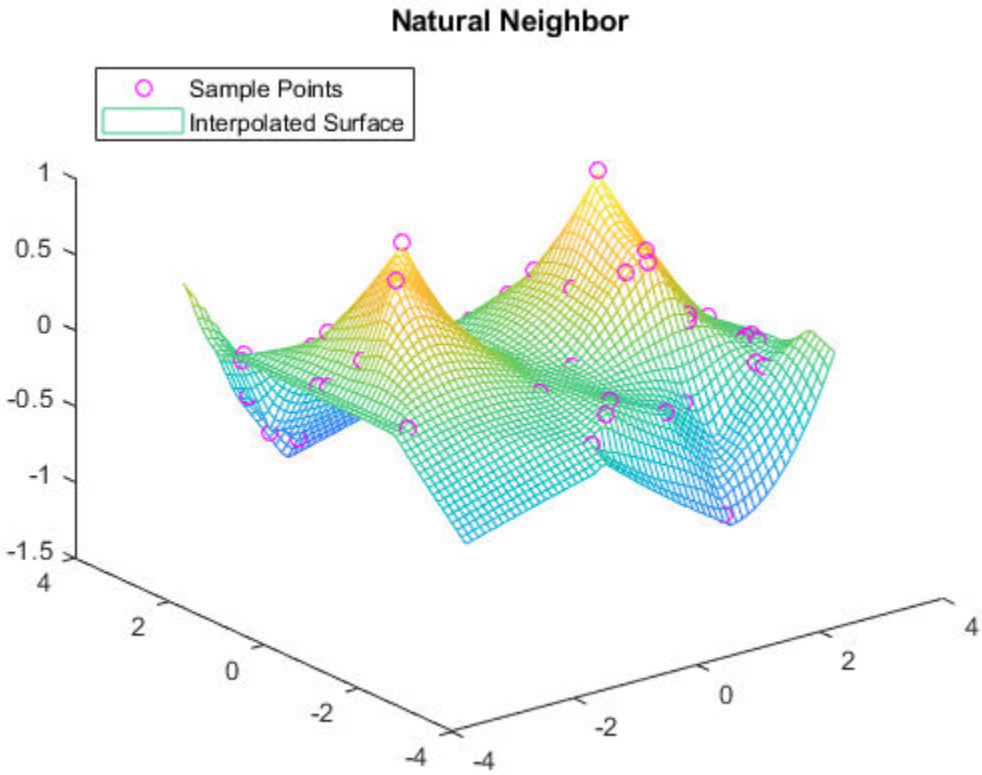
```
F.Method = 'nearest';
vq1 = F(xq,yq);
plot3(x,y,v,'mo')
hold on
mesh(xq,yq,vq1)
title('Nearest Neighbor')
legend('Sample Points','Interpolated Surface','Location','NorthWest')
```




```
F.Method = 'linear';
vq2 = F(xq,yq);
figure
plot3(x,y,v,'mo')
hold on
mesh(xq,yq,vq2)
title('Linear')
legend('Sample Points','Interpolated Surface','Location','NorthWest')
```

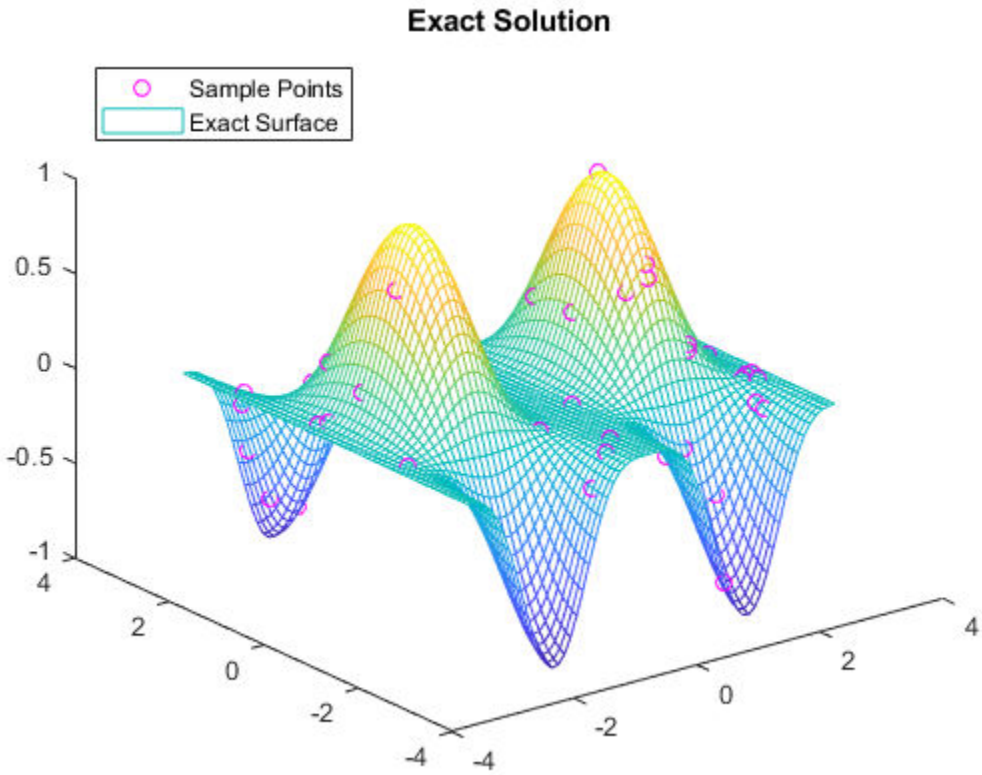



```
F.Method = 'natural';
vq3 = F(xq,yq);
figure
plot3(x,y,v,'mo')
hold on
mesh(xq,yq,vq3)
title('Natural Neighbor')
legend('Sample Points','Interpolated Surface','Location','NorthWest')
```



Plot the exact solution.

```
figure
plot3(x,y,v,'mo')
hold on
mesh(xq,yq,sin(xq).^4 .* cos(yq))
title('Exact Solution')
legend('Sample Points','Exact Surface','Location','NorthWest')
```



2-D Extrapolation

Query an interpolant at a single point outside the convex hull using nearest neighbor extrapolation.

Define a matrix of 200 random points and sample an exponential function. These points are the sample values for the interpolant.

```
P = -2.5 + 5*gallery('uniformdata',[200 2],0);
x = P(:,1);
y = P(:,2);
v = x.*exp(-x.^2-y.^2);
```

Create the interpolant, specifying linear interpolation and nearest neighbor extrapolation.

```
F = scatteredInterpolant(P,v,'linear','nearest')
```

```
F = 
  scatteredInterpolant with properties:

                 Points: [200x2 double]
                 Values: [200x1 double]
                 Method: 'linear'
    ExtrapolationMethod: 'nearest'
```

Evaluate the interpolant outside the convex hull.

```
vq = F(3.0,-1.5)
```

```
vq = 0.0031
```

Disable extrapolation and evaluate F at the same point.

```
F.ExtrapolationMethod = 'none';
vq = F(3.0,-1.5)
```

```
vq = NaN
```

More About

Interpolant

Interpolating function that you can evaluate at query points.

Gridded Data

A set of points that are axis-aligned and ordered.

Scattered Data

A set of points that have no structure among their relative locations.

Full Grid

A grid represented as a set of arrays. For example, you can create a full grid using `ndgrid`.

Grid Vectors

A set of vectors that serve as a compact representation of a grid in `ndgrid` format.

For example, `[X,Y] = ndgrid(xg,yg)` returns a full grid in the matrices `X` and `Y`. You can represent the same grid using the grid vectors `xg` and `yg`.

Tips

- It is quicker to evaluate a `scatteredInterpolant` object `F` at many different sets of query points than it is to compute the interpolations separately using the functions `griddata` or `griddatan`. For example:

  ```
  % Fast to create interpolant F and evaluate multiple times
  F = scatteredInterpolant(X,Y,V)
  v1 = F(Xq1,Yq1)
  v2 = F(Xq2,Yq2)

  % Slower to compute interpolations separately using griddata
  v1 = griddata(X,Y,V,Xq1,Yq1)
  v2 = griddata(X,Y,V,Xq2,Yq2)
  ```

- To change the interpolation sample values or interpolation method, it is more efficient to update the properties of the interpolant object `F` than it is to create a new `scatteredInterpolant` object. When you update `Values` or `Method`, the underlying Delaunay triangulation of the input data does not change, so you can compute new results quickly.
- Scattered data interpolation with `scatteredInterpolant` uses a Delaunay triangulation of the data, so can be sensitive to scaling issues in the sample points `x`, `y`, `z`, or `P`. When this occurs, you can use `normalize` to rescale the data and improve the results. See “Normalize Data with Differing Magnitudes” for more information.

Algorithms

`scatteredInterpolant` uses a Delaunay triangulation of the scattered sample points to perform interpolation [1].

References

[1] Amidror, Isaac. “Scattered data interpolation methods for electronic imaging systems: a survey.” *Journal of Electronic Imaging*. Vol. 11, No. 2, April 2002, pp. 157–176.

See Also

`griddata` | `griddatan` | `griddedInterpolant` | `meshgrid` | `ndgrid`

Introduced in R2013a

matlab.unittest.Scope class

Package: `matlab.unittest`

Test scope enumeration class

Description

The `matlab.unittest.Scope` enumeration class provides a means to specify the scope of test execution. The enumeration class contains the following members.

Numeric Representation	Enumeration Member Name	Scope Description
1	`TestMethod`	`TestMethodSetup`, `Test`, and `TestMethodTeardown` methods
2	`TestClass`	`TestClassSetup` and `TestClassTeardown` methods
3	`SharedTestFixture`	shared test fixture `setup` and `teardown` methods

Construction

`matlab.unittest.Scope.`*`MemberName`* creates an instance of the test scope enumeration class.

Examples

Create Instance of Enumeration Class

```
n = matlab.unittest.Scope.TestClass
```

```
n =

    TestClass
```

Display information about the variables.

```
whos n
```

```
  Name      Size            Bytes  Class                    Attributes

  n         1x1                 0  matlab.unittest.Scope
```

See Also

`matlab.unittest.plugins.diagnosticrecord`

Introduced in R2017a

sec

Secant of angle in radians

Syntax

```
Y = sec(X)
```

Description

`Y = sec(X)` returns the secant of the elements of `X`. The `sec` function operates element-wise on arrays. The function accepts both real and complex inputs.

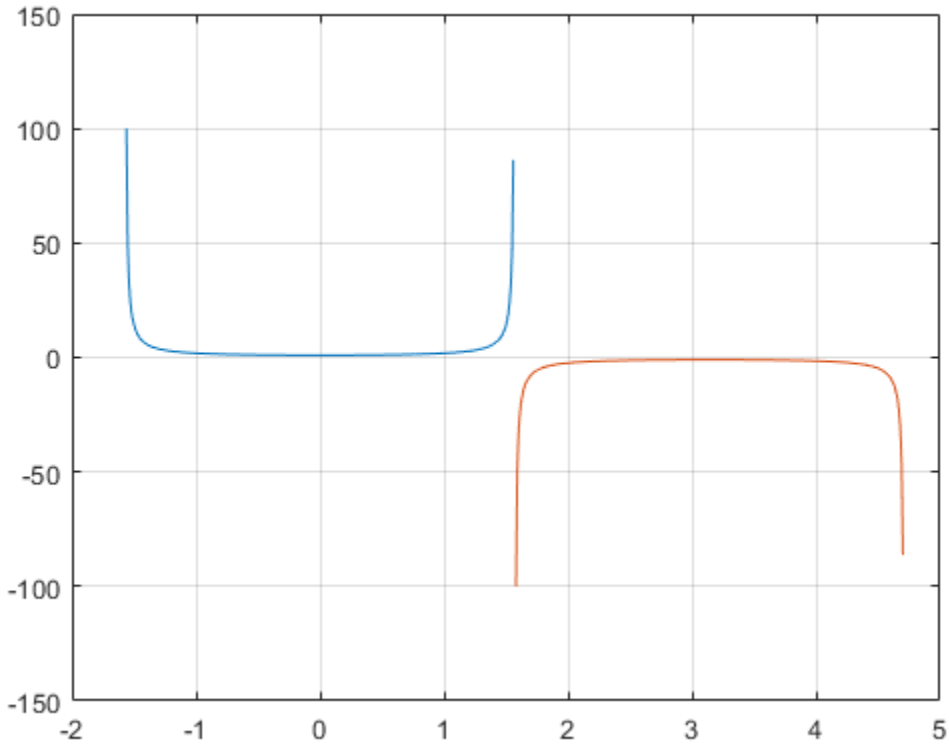
- For real values of `X`, `sec(X)` returns real values in the interval [-∞, -1] and [1, ∞].
- For complex values of `X`, `sec(X)` returns complex values.

Examples

Plot Secant Function

Plot the secant over the domain $-\pi/2 < x < \pi/2$ and $\pi/2 < x < 3\pi/2$.

```
x1 = -pi/2+0.01:0.01:pi/2-0.01;
x2 = pi/2+0.01:0.01:(3*pi/2)-0.01;
plot(x1,sec(x1),x2,sec(x2)), grid on
```



Secant of Vector of Complex Angles

Calculate the secant of the complex angles in vector x.

```
x = [-i pi+i*pi/2 -1+i*4];
y = sec(x)
```

```
y = 1×3 complex

   0.6481 + 0.0000i  -0.3985 + 0.0000i   0.0198 - 0.0308i
```

Input Arguments

X — Input angle in radians
scalar | vector | matrix | multidimensional array

Input angle in radians, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Secant of input angle
scalar | vector | matrix | multidimensional array

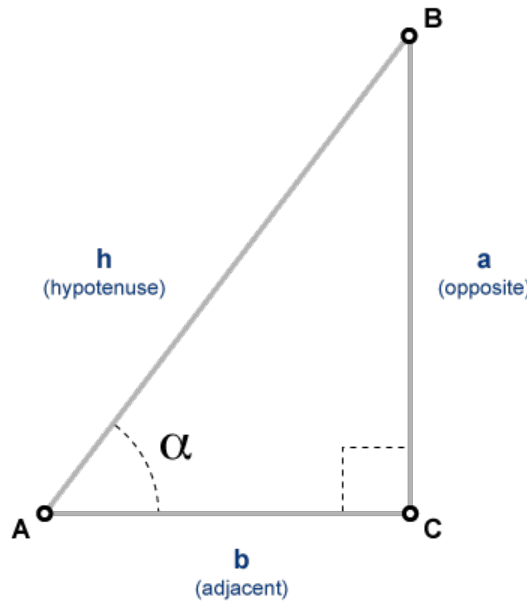
Secant of input angle, returned as real-valued or complex-valued scalar, vector, matrix or multidimensional array.

More About

Secant Function

The secant of an angle, α, defined with reference to a right angled triangle is

$$\sec(\alpha) = \frac{1}{\cos(\alpha)} = \frac{\text{hypotenuse}}{\text{adjacent side}} = \frac{h}{b} .$$



The secant of a complex argument, α, is

$$\sec(\alpha) = \frac{2}{e^{i\alpha} + e^{-i\alpha}} .$$

Tips

- In floating-point arithmetic, `sec` is a bounded function. That is, `sec` does not return values of `Inf` or `-Inf` at points of divergence that are multiples of `pi`, but a large magnitude number instead. This stems from the inaccuracy of the floating-point representation of π.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`asec` | `asecd` | `asech` | `secd` | `sech`

Introduced before R2006a

secd

Secant of argument in degrees

Syntax

```
Y = secd(X)
```

Description

`Y = secd(X)` returns the secant of the elements of `X`, which are expressed in degrees.

Examples

Secant of 90 degrees compared to secant of π/2 radians

```
secd(90)
```

```
ans = Inf
```

```
sec(pi/2)
```

```
ans = 1.6331e+16
```

`secd(90)` is infinite, whereas `sec(pi/2)` is large but finite.

Secant of vector of complex angles, specified in degrees

```
z = [35+i 15+2i 10+3i];
y = secd(z)
```

```
y = 1×3 complex
```

```
1.2204 + 0.0149i   1.0346 + 0.0097i   1.0140 + 0.0094i
```

Input Arguments

X — Angle in degrees
scalar value | vector | matrix | N-D array

Angle in degrees, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `secd` operation is element-wise when X is nonscalar.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Secant of angle
scalar value | vector | matrix | N-D array

Secant of angle, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as X.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- In some cases, generated code returns `-Inf` when MATLAB returns `Inf`.
- In some cases, generated code returns `Inf` when MATLAB returns `-Inf`.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`asec` | `asecd` | `sec`

Introduced before R2006a

sech

Hyperbolic secant

Syntax

```
Y = sech(X)
```

Description

`Y = sech(X)` returns the hyperbolic secant of the elements of `X`. The `sech` function operates element-wise on arrays. The function accepts both real and complex inputs. All angles are in radians.

Examples

Hyperbolic Secant of Vector

Create a vector and calculate the hyperbolic secant of each value.

```
X = [0 pi 2*pi 3*pi];
Y = sech(X)
```

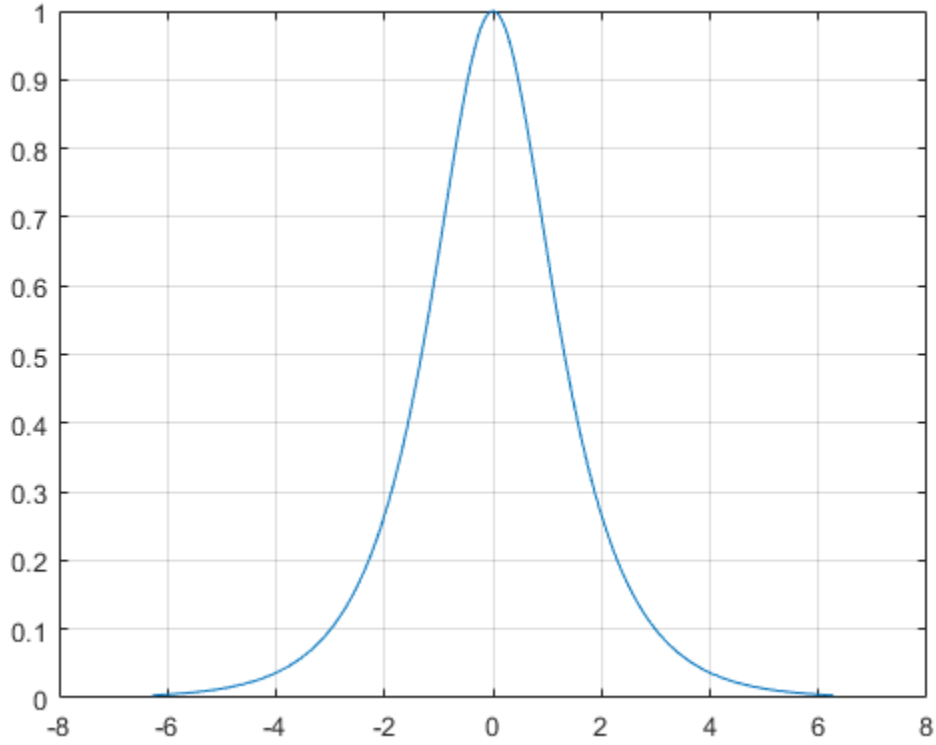
```
Y = 1×4

    1.0000    0.0863    0.0037    0.0002
```

Graph of Hyperbolic Secant Function

Plot the hyperbolic secant over the domain $-2\pi \le x \le 2\pi$.

```
x = -2*pi:0.01:2*pi;
y = sech(x);
```

```
plot(x,y)
grid on
```



Input Arguments

X — Input angles in radians
scalar | vector | matrix | multidimensional array

Input angles in radians, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Hyperbolic Secant

The hyperbolic secant of x is equal to the inverse of the hyperbolic cosine

$$\operatorname{sech}(x) = \frac{1}{\cosh(x)} = \frac{2}{e^x + e^{-x}}.$$

In terms of the traditional secant function with a complex argument, the identity is

$$\operatorname{sech}(x) = \sec(ix) .$$

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`asech` | `cosh` | `sec` | `sinh`

Introduced before R2006a

second

Second number

Syntax

```
s = second(t)
s = second(t,secondType)
```

Description

`s = second(t)` returns the second values, including a fractional part, for the datetime values in `t`. The `s` output is a `double` array the same size as `t` and contains values from 0 to less than 60.

For datetime values whose time zone is `UTCLeapSeconds`, the `s` output can contain a value between 60 and 61 for times that fall during a leap second occurrence.

`s = second(t,secondType)` returns the type of second number specified by `secondType`.

The `second` function returns the second numbers of datetime values. To assign second values to datetime array `t`, use `t.Second` and modify the `Second` property.

Examples

Find Second Number of Datetime Values

```
t1 = datetime('now','Format','dd-MMM-yyyy HH:mm:ss.SSS');
t = t1 + seconds(30:15:60)
```

```
t = 1x3 datetime array
Columns 1 through 2

   27-Aug-2019 03:06:52.823   27-Aug-2019 03:07:07.823
```

```
Column 3

   27-Aug-2019 03:07:22.823
```

```
s = second(t)
```

```
s = 1×3

   52.8230    7.8230   22.8230
```

Input Arguments

t — Input date and time
`datetime` array

Input date and time, specified as a `datetime` array.

secondType — Type of second values
`'secondofminute'` (default) | `'secondofday'`

Type of second values, specified as either `'secondofminute'` or `'secondofday'`.

- If `secondType` is `'secondofminute'`, then `second` returns the second of the minute.
- If `secondType` is `'secondofday'`, then `second` returns the second of the day, which (except for leap seconds) is in the range [0, 86400).

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`datetime` | `hms` | `hour` | `minute` | `timeofday`

Introduced in R2014b

seconds

Duration in seconds

Syntax

```
S = seconds(X)
```

Description

`S = seconds(X)` returns an array of seconds equivalent to the values in `X`.

- If `X` is a numeric array, then `S` is a `duration` array in units of seconds.
- If `X` is a `duration` array, then `S` is a `double` array with each element equal to the number of seconds in the corresponding element of `X`.

The `seconds` function converts between `duration` and `double` values. To display a duration in units of seconds, set its `Format` property to `'s'`.

Examples

Create Duration Array of Seconds

```
X =  magic(4);
S = seconds(X)
```

```
S = 4x4 duration array
   16 sec    2 sec    3 sec   13 sec
    5 sec   11 sec   10 sec    8 sec
    9 sec    7 sec    6 sec   12 sec
    4 sec   14 sec   15 sec    1 sec
```

Convert Durations to Numeric Array of Seconds

Create a `duration` array.

```
X = hours(1) + minutes(1:4)
```

```
X = 1x4 duration array
   1.0167 hr   1.0333 hr     1.05 hr   1.0667 hr
```

Convert each duration in `X` to a number of seconds.

```
S = seconds(X)
```

```
S = 1×4

        3660        3720        3780        3840
```

`S` is a `double` array.

Find the natural logarithm of `S`. The `log` function accepts inputs of type `double`.

```
Y = log(S)
```

```
Y = 1×4

    8.2052    8.2215    8.2375    8.2532
```

Input Arguments

X — Input array
numeric array | duration array | logical array

Input array, specified as a numeric array, duration array, or logical array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This function fully supports code generation. For more information, see “Code Generation for Duration Arrays” (MATLAB Coder).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`duration`

Introduced in R2014b

selectmoveresize

Select, move, resize, or copy axes and uicontrol graphics objects

Note The `selectmoveresize` function is not recommended. Use the `plotedit` function to copy, move, resize, and edit objects instead.

Syntax

```
A = selectmoveresize
set(gca,'ButtonDownFcn','selectmoveresize')
```

Description

In releases before R2014b, `selectmoveresize` was useful as the callback routine for axes and uicontrol button down functions. It allowed users to move, resize, and copy axes and uicontrol objects.

`A = selectmoveresize` returns a structure array containing

- `A.Type`: Action type, which can be `'Select'`, `'Move'`, `'Resize'`, or `'Copy'`
- `A.Handles`: a list of the selected handles, or, for a `Copy`, an m-by-2 matrix containing the original handles in the first column and the new handles in the second column

`set(gca,'ButtonDownFcn','selectmoveresize')` sets the `ButtonDownFcn` property of the current axes to `'selectmoveresize'`.

See Also

Axes | Uicontrol | `plotedit`

Introduced before R2006a

matlab.unittest.selectors Package

Summary of classes in MATLAB Selectors Interface

Description

Use selectors to filter or select elements of a test suite based on their attributes. The `matlab.unittest.selectors` package consists of the following selectors.

Classes

matlab.unittest.selectors.AndSelector	Boolean conjunction of two selectors
matlab.unittest.selectors.HasBaseFolder	Selector for TestSuite elements determined by folder
matlab.unittest.selectors.HasName	Selector for TestSuite elements determined by name
matlab.unittest.selectors.HasParameter	Selector for TestSuite elements determined by parameterization
matlab.unittest.selectors.HasProcedureName	Selector for TestSuite elements determined by procedure name
matlab.unittest.selectors.HasSharedTestFixture	Selector for TestSuite elements that use shared test fixture
matlab.unittest.selectors.HasSuperclass	Selector for TestSuite elements determined by test class hierarchy
matlab.unittest.selectors.HasTag	Selector for TestSuite elements determined by tag
matlab.unittest.selectors.NotSelector	Boolean complement of selector
matlab.unittest.selectors.OrSelector	Boolean disjunction of two selectors

See Also

`selectIf`

Introduced in R2014a

matlab.unittest.selectors.AndSelector class

Package: `matlab.unittest.selectors`

Boolean conjunction of two selectors

Description

The testing framework constructs an `AndSelector` when you denote the conjunction of two selectors with the logical AND operator (&). The framework constructs instances of the class, so there is no need to construct this class directly.

Properties

`FirstSelector` — Left side selector
instance of class in `matlab.unittest.selectors` package

This property is read-only.

Left side selector in the Boolean conjunction, specified as an instance of a class in the `matlab.unittest.selectors` package.

`SecondSelector` — Right side selector
instance of class in `matlab.unittest.selectors` package

This property is read-only.

Right side selector in the Boolean conjunction, specified as an instance of a class in the `matlab.unittest.selectors` package.

See Also

`matlab.unittest.selectors.NotSelector` | `matlab.unittest.selectors.OrSelector`

Introduced in R2018a

matlab.unittest.selectors.HasParameter class

Package: `matlab.unittest.selectors`

Selector for TestSuite elements determined by parameterization

Description

The `HasParameter` selector filters `TestSuite` array elements determined by parameterization.

Construction

`matlab.unittest.selectors.HasParameter` constructs a selector for `TestSuite` elements determined by their parameterization. When you instantiate `HasParameter` without input arguments, the resulting `TestSuite` array only contains elements that have parameterized test methods.

`matlab.unittest.selectors.HasParameter(Name,Value)` constructs a selector with additional options specified by one or more `Name,Value` pair arguments. The selector filters based on the name of the property that defines a parameter, the name of the parameter, and the value of the parameter. For an element to be selected for the `TestSuite` array, it must have at least one parameter that satisfies all the conditions.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`Property`

Name of the property that defines the parameter, specified as a character vector, string scalar, or `matlab.unittest.constraints.Constraint` instance. If the specified property name is a character vector or string scalar, the testing framework creates an `IsEqualTo` constraint with `Property` as the expected value.

`Name`

Name of the parameter, specified as a character vector, string scalar, or `matlab.unittest.constraints.Constraint` instance. If the specified name is a character vector or string scalar, the testing framework creates an `IsEqualTo` constraint with `Name` as the expected value.

`Value`

Value of the parameter, specified as any MATLAB data type or as an instance of the `matlab.unittest.constraints.Constraint` class. If the specified property name is not a constraint, the testing framework creates an `IsEqualTo` constraint with the input data, `Value`, as the expected value.

Properties

`PropertyConstraint`

Condition that the test element's parameter property name must satisfy to be included in the test suite, specified as an instance of the `Constraint` in the `Property` input argument.

`NameConstraint`

Condition that the test element's parameter name must satisfy to be included in the test suite, specified as an instance of the `Constraint` in the `Name` input argument.

`ValueConstraint`

Condition that the test element's parameter property value must satisfy to be included in the test suite, specified as an instance of the `Constraint` in the `Value` input argument.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Filter Test Suite by Parameterization

In your current folder, create the test class named `testZeros`. This class contains five test methods.

```
classdef testZeros < matlab.unittest.TestCase
    properties (TestParameter)
        type = {'single','double','uint16'};
        outSize = struct('s2d',[3 3],'s3d',[2 5 4]);
    end

    methods (Test)
        function testClass(testCase, type, outSize)
            testCase.verifyClass(zeros(outSize,type), type);
        end

        function testSize(testCase, outSize)
            testCase.verifySize(zeros(outSize), outSize);
        end

        function testDefaultClass(testCase)
            testCase.verifyClass(zeros, 'double');
        end
        function testDefaultSize(testCase)
            testCase.verifySize(zeros, [1 1]);
        end

        function testDefaultValue(testCase)
            testCase.verifyEqual(zeros,0);
        end
    end
end
```

The test class contains two parameterized test methods, `testClass` and `testSize`.

At the command prompt, create a test suite from the file.

```
s = matlab.unittest.TestSuite.fromFile('testZeros.m');
{s.Name}'
```

```
ans =

  11×1 cell array

    {'testZeros/testClass(type=single,outSize=s2d)'}
    {'testZeros/testClass(type=single,outSize=s3d)'}
    {'testZeros/testClass(type=double,outSize=s2d)'}
    {'testZeros/testClass(type=double,outSize=s3d)'}
    {'testZeros/testClass(type=uint16,outSize=s2d)'}
    {'testZeros/testClass(type=uint16,outSize=s3d)'}
    {'testZeros/testSize(outSize=s2d)'             }
    {'testZeros/testSize(outSize=s3d)'             }
    {'testZeros/testDefaultClass'                  }
    {'testZeros/testDefaultSize'                   }
    {'testZeros/testDefaultValue'                  }
```

The suite contains 11 test elements: 6 from the parameterized `testClass` method, 2 from the parameterized `testSize` method, and 1 from each of the `testDefaultClass`, `testDefaultSize`, and `testDefaultValue` methods.

Select all of the test elements from parameterized test methods.

```
import matlab.unittest.selectors.HasParameter

s1 = s.selectIf(HasParameter);
{s1.Name}'
```

```
ans =

  8×1 cell array

    {'testZeros/testClass(type=single,outSize=s2d)'}
    {'testZeros/testClass(type=single,outSize=s3d)'}
    {'testZeros/testClass(type=double,outSize=s2d)'}
    {'testZeros/testClass(type=double,outSize=s3d)'}
    {'testZeros/testClass(type=uint16,outSize=s2d)'}
    {'testZeros/testClass(type=uint16,outSize=s3d)'}
    {'testZeros/testSize(outSize=s2d)'             }
    {'testZeros/testSize(outSize=s3d)'             }
```

The suite contains the eight test elements from the two parameterized test methods.

Select all of the test elements from nonparameterized test methods.

```
s2 = s.selectIf(~HasParameter);
{s2.Name}'
```

```
ans =

  3×1 cell array

    {'testZeros/testDefaultClass'}
    {'testZeros/testDefaultSize' }
    {'testZeros/testDefaultValue'}
```

Select all test elements that are parameterized and have a property named `'type'` with a parameter name `'double'`.

```
s3 = s.selectIf(HasParameter('Property','type','Name','double'));
{s3.Name}'
```

```
ans =

  2×1 cell array

    {'testZeros/testClass(type=double,outSize=s2d)'}
    {'testZeros/testClass(type=double,outSize=s3d)'}
```

The resulting suite contains two elements. The `testClass` method is the only method in `testZeros` that uses the `'type'` property. Selecting only `'double'` from the parameters results in two test elements — one for each value of `'outSize'`.

Select all test elements that a have a parameter defined by a property starting with `'t'`.

```
import matlab.unittest.constraints.StartsWithSubstring

s4 = s.selectIf(HasParameter('Property',StartsWithSubstring('t')));
{s4.Name}'
```

```
ans =

  6×1 cell array

    {'testZeros/testClass(type=single,outSize=s2d)'}
    {'testZeros/testClass(type=single,outSize=s3d)'}
```

```
    {'testZeros/testClass(type=double,outSize=s2d)'}
    {'testZeros/testClass(type=double,outSize=s3d)'}
    {'testZeros/testClass(type=uint16,outSize=s2d)'}
    {'testZeros/testClass(type=uint16,outSize=s3d)'}
```

The resulting suite contains the six parameterized test elements from the `testClass` method. The `testSize` method is parameterized, but the elements from the method are not included in the suite because the method does not use a property that starts with `'t'`.

Select all test elements that are parameterized and test the `zeros` function with a `2-D` array. A parameter value representing a `2-D` array has a length of `1` (for example `zeros(3)`) or `2` (for example `zeros(2,3)`).

```
import matlab.unittest.constraints.HasLength

s5 = s.selectIf(HasParameter('Property','outSize',...
    'Value',HasLength(1)|HasLength(2)));
{s5.Name}'
```

```
ans =

  4×1 cell array

    {'testZeros/testClass(type=single,outSize=s2d)'}
    {'testZeros/testClass(type=double,outSize=s2d)'}
    {'testZeros/testClass(type=uint16,outSize=s2d)'}
    {'testZeros/testSize(outSize=s2d)'             }
```

Select only the test element that tests that the output is a `double` data type and that it has the correct size for a `2-D` array.

```
s6 = s.selectIf(HasParameter('Property','type','Name','double')...
    & HasParameter('Property','outSize','Name','s2d'))
```

```
s6 =

  Test with properties:

                 Name: 'testZeros/testClass(type=double,outSize=s2d)'
        ProcedureName: 'testClass'
            TestClass: "testZeros"
           BaseFolder: 'C:\work'
      Parameterization: [1×2 matlab.unittest.parameters.TestParameter]
    SharedTestFixtures: [0×0 matlab.unittest.fixtures.EmptyFixture]
                 Tags: {1×0 cell}
```

```
Tests Include:
   2 Unique Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

See Also

`fromClass` | `fromFile` | `fromFolder` | `fromMethod` | `fromPackage` | `matlab.unittest.parameters` | `matlab.unittest.selectors` | `selectIf`

Topics

"Create Basic Parameterized Test"
"Create Advanced Parameterized Test"

Introduced in R2014a

matlab.unittest.selectors.HasSharedTestFixture class

Package: `matlab.unittest.selectors`

Selector for TestSuite elements that use shared test fixture

Description

The `HasSharedTestFixture` selector filters `TestSuite` array elements based on shared test fixtures.

Construction

`matlab.unittest.selectors.HasSharedTestFixture(f)` constructs a selector for `TestSuite` elements based on their required shared test fixtures. For an element to be selected for the `TestSuite` array, it must use a fixture that is compatible with the specified fixture, `f`.

Input Arguments

`f` — Shared test fixture
`Fixture`

Shared test fixture, specified as a `matlab.unittest.fixtures.Fixture` instance. The `TestSuite` array element must use a fixture compatible with `f` to be selected for the `TestSuite`.

Properties

ExpectedFixture

Shared test fixture, specified as a `matlab.unittest.fixtures.Fixture` instance in the input argument, `f`. The `TestSuite` array element must use a fixture compatible with the `ExpectedFixture` property to be selected for the `TestSuite`.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Filter Test Suite by Shared Test Fixtures

Create a package folder, `+mytestpackage`, in your current folder. This package contains two test classes.

In the `+mytestpackage` folder, create a test class named `AExampleTest`. This class contains two tests that use a suppressed warnings fixture.

```
classdef (SharedTestFixtures={ ...
        matlab.unittest.fixtures.SuppressedWarningsFixture( ...
        'MATLAB:rmpath:DirNotFound')}) ...
        AExampleTest < matlab.unittest.TestCase
    methods (Test)
        function testOne(testCase)
            % test code
        end
        function testTwo(testCase)
            % test code
        end
    end
end
```

In the `+mytestpackage` folder, create a test class named `BExampleTest`. This class contains one test that uses a shared path fixture and a suppressed warnings fixture.

```
classdef (SharedTestFixtures={ ...
        matlab.unittest.fixtures.PathFixture( ...
        fullfile(matlabroot,'help','techdoc','matlab_oop','examples')),...
        matlab.unittest.fixtures.SuppressedWarningsFixture( ...
        'MATLAB:rmpath:DirNotFound')}) ...
        BExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testPathAdd(testCase)
            % test code
        end
    end
end
```

At the command prompt, define the following fixtures.

```
pf = matlab.unittest.fixtures.PathFixture(...
    fullfile(matlabroot,'help','techdoc','matlab_oop','examples'));
swf = matlab.unittest.fixtures.SuppressedWarningsFixture(...
    'MATLAB:rmpath:DirNotFound');
```

Create a test suite from the package.

```
import matlab.unittest.TestSuite
import matlab.unittest.selectors.HasSharedTestFixture

suite = TestSuite.fromPackage('mytestpackage')
```

```
suite = 

  1×3 Test array with properties:

    Name
    ProcedureName
    TestClass
    BaseFolder
    Parameterization
    SharedTestFixtures
    Tags

Tests Include:
   0 Parameterizations, 2 Unique Shared Test Fixture Classes, 0 Tags.
```

The test suite has three test elements.

Create a filtered suite that only contains tests that use the path fixture, `pf`.

```
s1 = suite.selectIf(HasSharedTestFixture(pf));
```

The resulting suite, `s1`, contains the test element from `BExampleTest`, since the test in that class uses the shared test fixture, `pf`.

Alternatively, pass the selector to the `TestSuite.fromPackage` method instead of generating a full test suite, and then using the `TestSuite.selectIf` method to filter the suite.

```
s1 = TestSuite.fromPackage('mytestpackage',HasSharedTestFixture(pf));
```

Create a filtered test suite that contains tests that use the suppressed warnings fixture, `swf`, but not the path fixture, `pf`.

```
s2 = suite.selectIf(~HasSharedTestFixture(pf) & HasSharedTestFixture(swf));
```

The test suite, `s2`, only contains the two test elements from `AExampleTest`. Tests in `BExampleTest` are excluded because, in addition to the suppressed warnings fixture, they use the path fixture.

Create a filtered suite that only contains tests that use the path fixture to a different location.

```
pf2 = matlab.unittest.fixtures.PathFixture(fullfile(matlabroot));
s3 =  TestSuite.fromPackage('mytestpackage', HasSharedTestFixture(pf2))
```

```
s3 =

  1×0 Test array with properties:

    Name
    ProcedureName
    TestClass
    BaseFolder
    Parameterization
    SharedTestFixtures
    Tags

Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

The test suite does not contain any test elements. The tests in `BExampleTest` use a shared path fixture, but the selected path fixture, `pf2`, adds a different folder to the path so its tests are not included in the suite.

See Also

`fromClass` | `fromFile` | `fromFolder` | `fromMethod` | `fromPackage` | `matlab.unittest.selectors` | `selectIf`

Introduced in R2014a

matlab.unittest.selectors.HasBaseFolder class

Package: `matlab.unittest.selectors`

Selector for TestSuite elements determined by folder

Description

The `HasBaseFolder` selector filters `TestSuite` array elements determined by the name of the folder that contains the file that defines the test class or function.

Construction

`matlab.unittest.selectors.HasBaseFolder(f)` constructs a selector for `TestSuite` elements determined by the folder, `f`, which contains the file that defines the test class or function. You can specify the base folder as a character vector, a string scalar, or an instance of the `matlab.unittest.constraints.Constraint` class. If the specified base folder, `f`, is a character vector or string scalar instead of a `Constraint`, the testing framework creates an `IsEqualTo` constraint to select test elements from the base folder `f`.

For a test element to be included in the suite, the file that defines it must be contained in the specified base folder. For test classes defined in packages, the base folder is the parent of the top-level package folder. The base folder never contains any folders that start with `'+'` or `'@'`.

Input Arguments

`f` — Base folder
character vector | string scalar | `Constraint`

Base folder, specified as a character vector, string scalar, or `matlab.unittest.constraints.Constraint` instance. The following conditions must be satisfied for the test element to be selected for the `TestSuite`:

- If `f` is a character vector or string scalar, the test element's base folder must exactly match the specified folder.
- If `f` is a constraint, the test element's base folder must satisfy the specified constraint.

Properties

`Constraint`

Condition the base folder must satisfy to be included in the test suite, specified as an instance of the `matlab.unittest.constraints.Constraint` class.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Filter Test Suite by Base Folder

Create a folder named `MyTests` in your current folder. In this folder, create two subfolders, `Feature1` and `Feature2`.

In a new file, `Feature1_Test.m`, in the `Feature1` subfolder, create the following test class.

```
classdef Feature1_Test < matlab.unittest.TestCase
    methods (Test)
        function testA1(testCase)
            % test code
        end
        function testB1(testCase)
            % test code
        end
    end
end
```

In a new file, `Feature2_Test.m`, in the `Feature2` subfolder, create the following test class.

```
classdef Feature2_Test < matlab.unittest.TestCase
    methods (Test)
        function testA2(testCase)
            % test code
        end
        function testB2(testCase)
            % test code
        end
    end
end
```

If necessary, set your current folder to the folder above `MyTests`. At the command prompt, create a test suite from the `MyTests` folder and examine the contents.

```
import matlab.unittest.TestSuite
import matlab.unittest.selectors.HasBaseFolder
import matlab.unittest.constraints.ContainsSubstring

suite = TestSuite.fromFolder('MyTests','IncludingSubfolders',true);
{suite.Name}
```

```
ans =

  1×4 cell array

    {'Feature1_Test/testA1'}    {'Feature1_Test/testB1'}    {'Feature2_Test/testA2'}    {'Feature2_Test/testB2'}
```

The suite contains the four tests from the two test files.

Select all test suite elements for classes that are defined in the `'Feature1'` folder.

```
s1 = suite.selectIf(HasBaseFolder(...
    fullfile(pwd,'MyTests','Feature1')));
{s1.Name}
```

```
ans =

  1×2 cell array

    {'Feature1_Test/testA1'}    {'Feature1_Test/testB1'}
```

The filtered test suite contains only test elements from the `Feature1` folder.

Select all the test suite elements for classes that are defined in folders that do not contain `'Feature1'`, and then examine the contents.

```
s2 = suite.selectIf(~HasBaseFolder(...
    fullfile(pwd,'MyTests','Feature1')));
{s2.Name}
```

```
ans =

  1×2 cell array

    {'Feature2_Test/testA2'}    {'Feature2_Test/testB2'}
```

The filtered test suite only contains test elements from the `Feature2` folder.

Alternatively, to generate a filtered suite directly, pass the selector to the `TestSuite.fromFolder` method.

```
s2 = TestSuite.fromFolder('MyTests',...
    ~HasBaseFolder(ContainsSubstring('Feature1')),...
    'IncludingSubfolders',true);
{s2.Name}
```

```
ans =

  1×2 cell array

    {'Feature2_Test/testA2'}    {'Feature2_Test/testB2'}
```

See Also

`fromClass` | `fromFile` | `fromFolder` | `fromMethod` | `fromPackage` | `matlab.unittest.selectors` | `selectIf`

Introduced in R2014a

matlab.unittest.selectors.HasName class

Package: `matlab.unittest.selectors`

Selector for TestSuite elements determined by name

Description

The `HasName` selector filters `TestSuite` array elements determined by the test element name.

Construction

`matlab.unittest.selectors.HasName(n)` constructs a selector for `TestSuite` elements determined by the test element name, `n`. You can specify the name as a character vector, a string scalar, or an instance of the `matlab.unittest.constraints.Constraint` class. If the specified name, `n`, is a character vector or string scalar, the testing framework creates an `IsEqualTo` constraint with `n` as the expected value.

For a test element to be included in the suite, the test element must have the same name as the specified name.

Input Arguments

`n` — Test element name
character vector | string scalar | `Constraint`

Test element name, specified as a character vector, string scalar, or `matlab.unittest.constraints.Constraint` instance. The following conditions must be satisfied for the test element to be selected for the `TestSuite`:

- If `n` is a character vector or string scalar, the test element's name must exactly match the specified name.
- If `n` is a constraint, the test element's name must satisfy the specified constraint.

Properties

Constraint

Condition the test element name must satisfy to be included in the test suite, specified as an instance of the `matlab.unittest.constraints.Constraint` class.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Filter Test Suite by Name

Create the following test class in a file, `ExampleTest.m`, in your current folder.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testPathAdd(testCase)
            % test code
        end
        function testOne(testCase)
            % test code
        end
         function testTwo(testCase)
            % test code
        end
    end
end
```

At the command prompt, create a test suite from the `ExampleTest.m` file and examine the contents.

```
import matlab.unittest.TestSuite
import matlab.unittest.selectors.HasName
import matlab.unittest.constraints.EndsWithSubstring
```

```
suite = TestSuite.fromFile('ExampleTest.m');
{suite.Name}
```

```
ans =

  1×3 cell array

    {'ExampleTest/testPathAdd'}    {'ExampleTest/testOne'}    {'ExampleTest/testTwo'}
```

The suite contains three tests.

Select all the test suite elements that have the name `'ExampleTest/testPathAdd'`, and examine the contents.

```
s1 = suite.selectIf(HasName('ExampleTest/testPathAdd'));
{s1.Name}
```

```
ans =

  1×1 cell array

    {'ExampleTest/testPathAdd'}
```

The filtered test suite only contains one test element.

Select all the test suite elements that end in either `'One'` or `'Two'`, and examine the contents.

```
s1 =  suite.selectIf(HasName(EndsWithSubstring('One')) | ...
    HasName(EndsWithSubstring('Two')));
{s1.Name}
```

```
ans =

  1×2 cell array

    {'ExampleTest/testOne'}    {'ExampleTest/testTwo'}
```

At the time of the test suite construction, create a test suite that only contains tests with the substring `'One'`.

```
import matlab.unittest.constraints.ContainsSubstring
s2 = TestSuite.fromFile('ExampleTest.m',...
    HasName(ContainsSubstring('One')))
```

```
s2 =
```

```
  Test with properties:

                 Name: 'ExampleTest/testOne'
        ProcedureName: 'testOne'
            TestClass: "ExampleTest"
           BaseFolder: 'C:\work'
      Parameterization: [0×0 matlab.unittest.parameters.EmptyParameter]
    SharedTestFixtures: [0×0 matlab.unittest.fixtures.EmptyFixture]
                 Tags: {1×0 cell}

Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

See Also

fromClass | fromFile | fromFolder | fromMethod | fromPackage | matlab.unittest.selectors | selectIf

Introduced in R2014a

matlab.unittest.selectors.HasProcedureName class

Package: `matlab.unittest.selectors`

Selector for `TestSuite` elements determined by procedure name

Description

The `HasProcedureName` selector filters `TestSuite` array elements determined by the procedure name. The name of the test procedure does not include any class or package name or information about parameterization. In a class-based test, the procedure name is the name of the test method. In a function-based test, it is the name of the local function that contains the test. In a script-based test, it is a name generated from the test section title.

Construction

`matlab.unittest.selectors.HasProcedureName(n)` constructs a selector for `TestSuite` elements determined by the procedure name, `n`.

For a test element to be included in the suite, the procedure name of the test element must match the specified name or satisfy the specified constraint.

Input Arguments

n — Procedure name
character vector | string scalar | instance of `matlab.unittest.constraints.Constraint`

Procedure name specified as a character vector, string scalar, or `matlab.unittest.constraints.Constraint` instance. The name of the test procedure does not include any class or package name or information about parameterization. In a class-based test, the procedure name is the name of the test method. In a function-based test, it is the name of the local function that contains the test. In a script-based test, it is the name of a test section.

Example: `"Test1"`

Example: `ContainsSubstring('Test')`

Properties

`Constraint` — Condition procedure name must satisfy
instance of `matlab.unittest.constraints.Constraint` class

Condition the procedure name must satisfy to be included in the test suite, specified as an instance of the `matlab.unittest.constraints.Constraint` class.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Filter Test Suite by Procedure Name

Create the following test class in a file, `ExampleTest.m`, in your current folder.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testPathAdd(testCase)
            % test code
        end
        function testOne(testCase)
            % test code
        end
         function testTwo(testCase)
            % test code
        end
    end
end
```

At the command prompt, create a test suite from the `ExampleTest.m` file and examine the contents.

```
import matlab.unittest.TestSuite;
import matlab.unittest.selectors.HasProcedureName;
import matlab.unittest.constraints.EndsWithSubstring;

suite = TestSuite.fromFile('ExampleTest.m');
{suite.Name}
```

```
ans =

  1×3 cell array

    {'ExampleTest/testPathAdd'}    {'ExampleTest/testOne'}    {'ExampleTest/testTwo'}
```

The suite contains three tests.

Select all the test suite elements that have the procedure name `testPathAdd`, and examine the contents.

```
s1 = suite.selectIf(HasProcedureName("testPathAdd"))
```

```
s1 =

  Test with properties:

                 Name: 'ExampleTest/testPathAdd'
        ProcedureName: 'testPathAdd'
            TestClass: "ExampleTest"
           BaseFolder: 'C:\work'
      Parameterization: [0×0 matlab.unittest.parameters.EmptyParameter]
    SharedTestFixtures: [0×0 matlab.unittest.fixtures.EmptyFixture]
                 Tags: {1×0 cell}

Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

The filtered test suite only contains one test element.

Select all the test suite elements with a procedure name that ends in either `'One'` or `'Two'`, and examine the contents.

```
s1 =  suite.selectIf(HasProcedureName(EndsWithSubstring('One')) | ...
    HasProcedureName(EndsWithSubstring('Two')));
{s1.Name}
```

```
ans =
```

```
  1×2 cell array

    {'ExampleTest/testOne'}    {'ExampleTest/testTwo'}
```

At the time of the test suite construction, create a test suite that only contains tests with the substring `'One'`.

```
import matlab.unittest.constraints.ContainsSubstring;
s2 = TestSuite.fromFile('ExampleTest.m',...
    HasProcedureName(ContainsSubstring('One')))
```

```
s2 =

  Test with properties:

                 Name: 'ExampleTest/testOne'
        ProcedureName: 'testOne'
            TestClass: "ExampleTest"
           BaseFolder: 'C:\work'
      Parameterization: [0×0 matlab.unittest.parameters.EmptyParameter]
    SharedTestFixtures: [0×0 matlab.unittest.fixtures.EmptyFixture]
                 Tags: {1×0 cell}

Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

Alternatives

Use the `HasProcedureName` selector for maximum flexibility to create test suites from procedure names. Alternatively, at the time of test suite construction, you can filter the test suite using the `'ProcedureName'` name-value pair. For example, the following lines of code are functionally equivalent.

```
s = TestSuite.fromClass(?ExampleTest,'ProcedureName','Test1');
s = testsuite('ExampleTest.m','ProcedureName','Test1');
```

See Also

`fromClass` | `fromFile` | `fromFolder` | `fromMethod` | `fromPackage` | `matlab.unittest.selectors` | `selectIf`

Introduced in R2017a

matlab.unittest.selectors.HasSuperclass class

Package: `matlab.unittest.selectors`

Selector for `TestSuite` elements determined by test class hierarchy

Construction

`matlab.unittest.selectors.HasSuperclass(n)` creates a selector that filters `TestSuite` array elements by retaining only those elements whose test class derives from `n`.

Input Arguments

n — Superclass name
character vector | string scalar

Superclass name, specified as a character vector or string scalar.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Filter Test Suite by Test Class

Create the following handle class in your current folder.

```
classdef MyClass < handle
    properties
        % properties of class
    end
    methods
```

```
        % methods of class
    end
end
```

Create the following test class in your current folder.

```
classdef Feature1_Test < matlab.unittest.TestCase
    methods (Test)
        function testA1(testCase)
            % test code
        end
        function testB1(testCase)
            % test code
        end
    end
end
```

Create the following test class in your current folder. This test class subclasses `MyClass`.

```
classdef Feature2_Test < matlab.unittest.TestCase & MyClass
    methods (Test)
        function testA2(testCase)
            % test code
        end
        function testB2(testCase)
            % test code
        end
    end
end
```

Create a test suite from all the files in your current folder. Depending on what files are in your folder, your test suite might differ.

```
import matlab.unittest.TestSuite;
suite = TestSuite.fromFolder(pwd)
```

```
suite =

  1×4 Test array with properties:

    Name
    ProcedureName
    TestClass
    BaseFolder
    Parameterization
```

```
    SharedTestFixtures
    Tags

Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

View the names of the test elements in the suite.

```
{suite.Name}'
```

```
ans =

  4×1 cell array

    {'Feature1_Test/testA1'}
    {'Feature1_Test/testB1'}
    {'Feature2_Test/testA2'}
    {'Feature2_Test/testB2'}
```

Create another test suite that contains only the suite elements that have `MyClass` in the class hierarchy. View the test element names. Since the test class `Feature2_Test` is the only test class to subclass `MyClass`, only test elements from that class are included in the suite.

```
newSuite = suite.selectIf(HasSuperclass('MyClass'));
{newSuite.Name}'
```

```
ans =

  2×1 cell array

    {'Feature2_Test/testA2'}
    {'Feature2_Test/testB2'}
```

Alternatively, use the `testsuite` function to create the filtered suite directly.

```
newSuite = testsuite(pwd,'Superclass','MyClass');
```

See Also

`fromClass` | `fromFile` | `fromFolder` | `fromMethod` | `fromPackage` | `matlab.unittest.selectors` | `selectIf`

Introduced in R2018a

matlab.unittest.selectors.HasTag class

Package: `matlab.unittest.selectors`

Selector for TestSuite elements determined by tag

Description

The `HasTag` selector filters `TestSuite` array elements determined by the test element tag.

Construction

`matlab.unittest.selectors.HasTag` constructs a selector for `TestSuite` elements determined by the test element tag. When you instantiate `HasTag` without input arguments, the resulting `TestSuite` array contains only elements with one or more tags.

`matlab.unittest.selectors.HasTag(t)` constructs a selector for `TestSuite` elements determined by the test element tag, `t`. You can specify the tag as a character vector, a string scalar, or an instance of the `matlab.unittest.constraints.Constraint` class. If the specified tag, `t`, is a character vector or string scalar, the testing framework creates an `IsEqualTo` constraint to select test elements with the tag `t`.

For a test element to be included in the suite, the test element must be tagged with the specified character vector or string scalar or with a value that satisfies the specified constraint.

Input Arguments

`t` — Test element tag
character vector | string scalar | `Constraint`

Test element tag, specified as a character vector, string scalar, or `matlab.unittest.constraints.Constraint` instance. If a test element tag meets the following conditions, the `TestSuite` contains the test:

- If `t` is a character vector or string scalar, the test element tag is the specified value.
- If `t` is a constraint, the test element tag is a value that satisfies the specified constraint.

Properties

`Constraint` — Condition test element tag must satisfy
instance of `matlab.unittest.constraints.Constraint` class

Condition the test element tag must satisfy to be included in the test suite, specified as an instance of the `matlab.unittest.constraints.Constraint` class.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Filter Test Suite by Tag

Create the following test class in a file, `ExampleTest.m`, in your current folder.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods (Test)
        function testA (testCase)
            % test code
        end
    end
    methods (Test, TestTags = {'Unit'})
        function testB (testCase)
            % test code
        end
        function testC (testCase)
            % test code
        end
    end
    methods (Test, TestTags = {'Unit','FeatureA'})
```

```
        function testD (testCase)
            % test code
        end
    end
    methods (Test, TestTags = {'System','FeatureA'})
        function testE (testCase)
            % test code
        end
    end
end
```

At the command prompt, create a test suite from the `ExampleTest` class and examine the contents.

```
import matlab.unittest.TestSuite
import matlab.unittest.selectors.HasTag

suite = TestSuite.fromClass(?ExampleTest)
```

```
suite =

  1×5 Test array with properties:

    Name
    ProcedureName
    TestClass
    BaseFolder
    Parameterization
    SharedTestFixtures
    Tags

Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 3 Unique Tags.
```

Click the hyperlink for `3 Unique Tags` to display all the tags in the suite.

```
        Tag
    ____________

    {'FeatureA'}
    {'System'  }
    {'Unit'    }
```

Select all the test suite elements that have the tag `'Unit'`.

```
s1 = suite.selectIf(HasTag('Unit'))
```

```
s1 =

  1×3 Test array with properties:

    Name
    ProcedureName
    TestClass
    BaseFolder
    Parameterization
    SharedTestFixtures
    Tags

Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 2 Unique Tags.
```

Select all the test suite elements that do not contain the tag `'FeatureA'`.

```
s2 =  suite.selectIf(~HasTag('FeatureA'));
{s2.Name}
```

```
ans =

  1×3 cell array

    {'ExampleTest/testB'}    {'ExampleTest/testC'}    {'ExampleTest/testA'}
```

Select all the test suite elements that have no tags.

```
s3 =  suite.selectIf(~HasTag)
```

```
s3 =

  Test with properties:

                 Name: 'ExampleTest/testA'
        ProcedureName: 'testA'
            TestClass: "ExampleTest"
           BaseFolder: 'C:\work'
      Parameterization: [0×0 matlab.unittest.parameters.EmptyParameter]
    SharedTestFixtures: [0×0 matlab.unittest.fixtures.EmptyFixture]
                 Tags: {1×0 cell}
```

```
Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

Alternatives

Use the `HasTag` selector for maximum flexibility to create test suites from tags. Alternatively, at the time of test suite construction, you can filter the test suite using the `'Tag'` name-value pair. For example:

```
s = TestSuite.fromClass(?ExampleTest,'Tag','Unit');
```

You can also select and run tagged tests using the `'Tag'` name-value pair with the `runtests` function. For example:

```
runtests('ExampleTest.m','Tag','Unit')
```

See Also

`fromClass` | `fromFile` | `fromFolder` | `fromMethod` | `fromPackage` | `matlab.unittest.selectors` | `selectIf`

Topics

"Tag Unit Tests"

Introduced in R2015a

matlab.unittest.selectors.NotSelector class

Package: `matlab.unittest.selectors`

Boolean complement of selector

Description

The testing framework constructs a `NotSelector` when you denote the complement of a selector with the logical NOT operator (~). The framework constructs instances of the class, so there is no need to construct this class directly.

Properties

`Selector` — Selector being complemented
instance of class in `matlab.unittest.selectors` package

This property is read-only.

Selector being complemented, specified as an instance of a class in the `matlab.unittest.selectors` package.

See Also

`matlab.unittest.selectors.AndSelector` | `matlab.unittest.selectors.OrSelector`

Introduced in R2018a

matlab.unittest.selectors.OrSelector class

Package: `matlab.unittest.selectors`

Boolean disjunction of two selectors

Description

The testing framework constructs an `OrSelector` when you denote the disjunction of two selectors with the logical OR operator (`|`). The framework constructs instances of the class, so there is no need to construct this class directly.

Properties

`FirstSelector` — Left side selector
instance of class in `matlab.unittest.selectors` package

This property is read-only.

Left side selector in the Boolean disjunction, specified as an instance of a class in the `matlab.unittest.selectors` package.

`SecondSelector` — Right side selector
instance of class in `matlab.unittest.selectors` package

This property is read-only.

Right side selector in the Boolean disjunction, specified as an instance of a class in the `matlab.unittest.selectors` package.

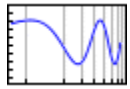
See Also

`matlab.unittest.selectors.AndSelector` | `matlab.unittest.selectors.NotSelector`

Introduced in R2018a

semilogx

Semilogarithmic plot



Syntax

```
semilogx(Y)
semilogx(X1,Y1,...)
semilogx(X1,Y1,LineSpec,...)
semilogx(...,'PropertyName',PropertyValue,...)
semilogx(ax,...)
h = semilogx(...)
```

Description

`semilogx` plot data as logarithmic scales for the *x*-axis.

`semilogx(Y)` creates a plot using a base 10 logarithmic scale for the *x*-axis and a linear scale for the *y*-axis. It plots the columns of `Y` versus their index. The values in `Y` can be numeric, datetime, duration, or categorical values. If `Y` contains complex values, then `semilogx(Y)` is equivalent to `semilogx(real(Y),imag(Y))`. The `semilogx` function ignores the imaginary component in all other uses of this function.

`semilogx(X1,Y1,...)` plots all `Yn` versus `Xn` pairs. If only one of `Xn` or `Yn` is a matrix, `semilogx` plots the vector argument versus the rows or columns of the matrix, along the dimension of the matrix whose length matches the length of the vector. If the matrix is square, its columns plot against the vector if their lengths match. The values in `Yn` can be numeric, datetime, duration, or categorical values. The values in `Xn` must be numeric.

`semilogx(X1,Y1,LineSpec,...)` plots all lines defined by the `Xn,Yn,LineSpec` triples. `LineSpec` determines line style, marker symbol, and color of the plotted lines.

`semilogx(...,'PropertyName',PropertyValue,...)` sets property values for all charting lines created by `semilogx`. For a list of properties, see Chart Line.

`semilogx(ax,...)` creates the line in the axes specified by `ax` instead of in the current axes (`gca`). The option `ax` can precede any of the input argument combinations in the previous syntaxes.

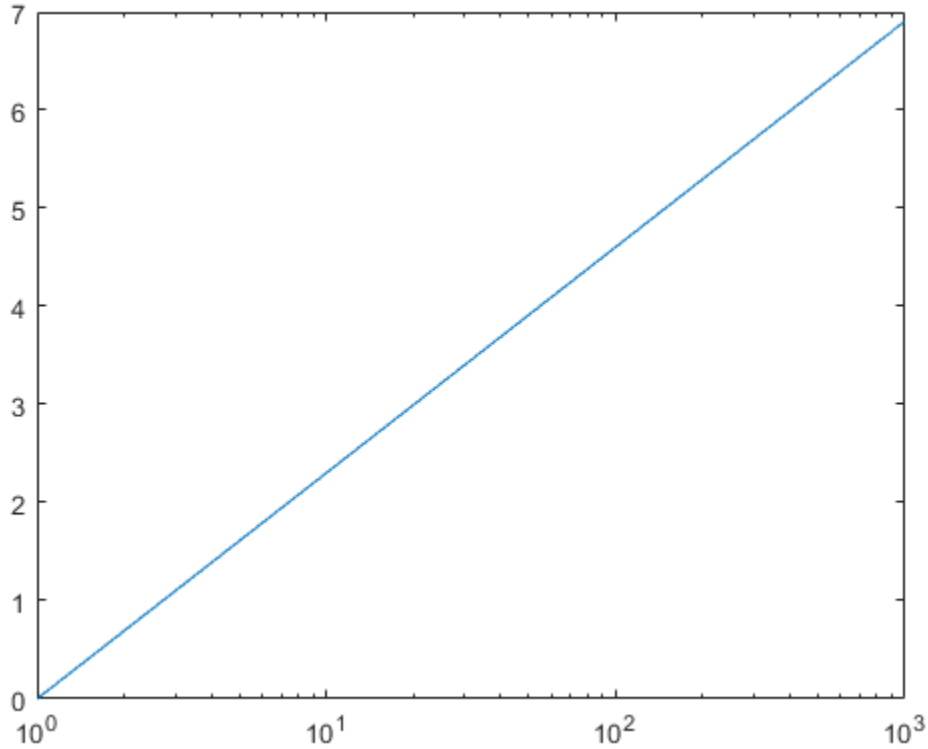
`h = semilogx(...)` return a vector of chart line objects.

Examples

Logarithmic Scale for x-Axis

Create a plot with a logarithmic scale for the x-axis and a linear scale for the y-axis.

```
x = 0:1000;
y = log(x);

figure
semilogx(x,y)
```



Tips

If you do not specify a color when plotting more than one line, `semilogx` automatically cycle through the colors and line styles in the order specified by the current axes `ColorOrder` and `LineStyleOrder` properties.

You can mix `Xn,Yn` pairs with `Xn,Yn,`*`LineSpec`* triples; for example,

```
semilogx(X1,Y1,X2,Y2,LineSpec,X3,Y3)
```

If you attempt to add a `loglog`, `semilogx`, or `semilogy` plot to a linear axis mode graph with `hold on`, the axis mode remains as it is and the new data plots as linear.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions

`LineSpec` | `loglog` | `plot` | `semilogy`

Properties

Chart Line

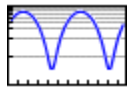
Topics

“Plot Dates and Durations”
“Plot Categorical Data”

Introduced before R2006a

semilogy

Semilogarithmic plot



Syntax

```
semilogy(Y)
semilogy(X1,Y1,...)
semilogy(X1,Y1,LineSpec,...)
semilogy(...,'PropertyName',PropertyValue,...)
semilogy(ax,...)
h = semilogy(...)
```

Description

`semilogy` plots data with logarithmic scale for the *y*-axis.

`semilogy(Y)` creates a plot using a base 10 logarithmic scale for the *y*-axis and a linear scale for the *x*-axis. It plots the columns of `Y` versus their index. If `Y` contains complex values, then `semilogy(Y)` is equivalent to `semilogy(real(Y),imag(Y))`. The `semilogy` function ignores the imaginary component in all other uses of this function.

`semilogy(X1,Y1,...)` plots all `Yn` versus `Xn` pairs. If only one of `Xn` or `Yn` is a matrix, `semilogy` plots the vector argument versus the rows or columns of the matrix, along the dimension of the matrix whose length matches the length of the vector. If the matrix is square, its columns plot against the vector if their lengths match. The values in `Xn` can be numeric, datetime, duration, or categorical values. The values in `Yn` must be numeric.

`semilogy(X1,Y1,LineSpec,...)` plots all lines defined by the `Xn,Yn,LineSpec` triples. `LineSpec` determines line style, marker symbol, and color of the plotted lines.

`semilogy(...,'PropertyName',PropertyValue,...)` sets property values for all the charting lines created by `semilogy`. For a list of properties, see Chart Line.

`semilogy(ax,...)` creates the line in the axes specified by `ax` instead of in the current axes (`gca`). The option `ax` can precede any of the input argument combinations in the previous syntaxes.

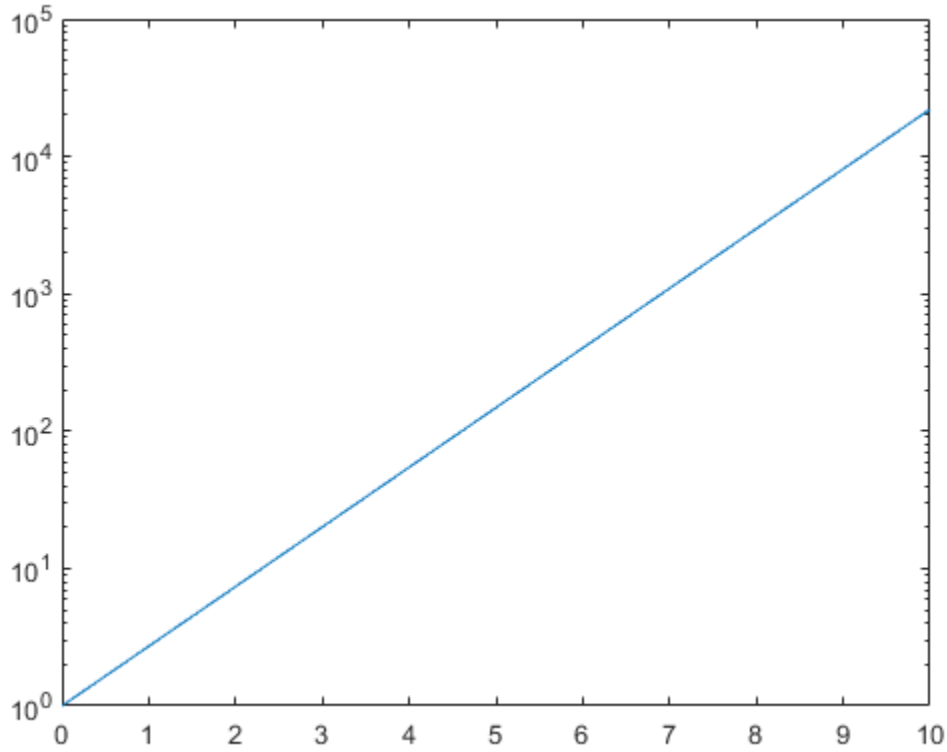
`h = semilogy(...)` returns a vector of chart line objects.

Examples

Logarithmic Scale for y-Axis

Create a plot with a logarithmic scale for the y-axis and a linear scale for the x-axis.

```
x = 0:0.1:10;
y = exp(x);

figure
semilogy(x,y)
```



Tips

If you do not specify a color when plotting more than one line, `semilogy` automatically cycle through the colors and line styles in the order specified by the current axes `ColorOrder` and `LineStyleOrder` properties.

You can mix `Xn,Yn` pairs with `Xn,Yn,`*`LineSpec`* triples; for example,

```
semilogy(X1,Y1,X2,Y2,LineSpec,X3,Y3)
```

If you attempt to add a `loglog`, `semilogx`, or `semilogy` plot to a linear axis mode graph with `hold on`, the axis mode remains as it is and the new data plots as linear.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions

`LineSpec` | `loglog` | `plot` | `semilogx`

Properties

Chart Line

Topics

“Plot Dates and Durations”
“Plot Categorical Data”

Introduced before R2006a

sendmail

Send email message to address list

Syntax

```
sendmail(recipients,subject)
sendmail(recipients,subject,message)
sendmail(recipients,subject,message,attachments)
```

Description

`sendmail(recipients,subject)` sends email to `recipients` with the specified `subject`. For a single recipient, specify `recipients` as a character vector or a string. For multiple recipients, specify `recipients` as a cell array of character vectors or a string array. `subject` must be a character vector or string.

`sendmail(recipients,subject,message)` includes the specified `message`. If `message` is a character vector or a string, `sendmail` automatically wraps text at 75 characters. To force a line break in the message text, use `10`, as shown in the Examples. If `message` is a cell array of character vectors or a string array, then each element represents a new line of text.

`sendmail(recipients,subject,message,attachments)` attaches the files listed in the `attachments` input argument. `attachments` can be a character vector, cell array of character vectors, or string array.

Examples

Send a message with two attachments to a hypothetical email address:

```
sendmail('user@otherdomain.com',...
         'Test subject','Test message',...
         {'folder/attach1.html','attach2.doc'});
```

Send a message with forced line breaks (using `10`) to a hypothetical email address:

```
sendmail('user@otherdomain.com','New subject', ...
        ['Line1 of message' 10 'Line2 of message' 10 ...
         'Line3 of message' 10 'Line4 of message']);
```

The resulting message is:

```
Line1 of message
Line2 of message
Line3 of message
Line4 of message
```

Tips

- The `sendmail` function does not support HTML-formatted messages. However, you can send HTML files as attachments.
- If `sendmail` cannot determine your email address or outgoing SMTP mail server from your system registry, specify those settings using the `setpref` function. For example:

  ```
  setpref('Internet','SMTP_Server','my_server.example.com');
  setpref('Internet','E_mail','my_email@example.com');
  ```

  To identify the SMTP server for the call to `setpref`, check the preferences for your electronic mail application, or consult your email system administrator. If you cannot easily determine the server name, try `'mail'`, which is a common default, such as:

  ```
  setpref('Internet','SMTP_Server','mail');
  ```
- By default, the `sendmail` function does not support email servers that require authentication. To support these servers, change your system settings and set preferences for the SMTP user name and password, with commands in the following form:

  ```
  props = java.lang.System.getProperties;
  props.setProperty('mail.smtp.auth','true');

  setpref('Internet','SMTP_Username','myaddress@example.com');
  setpref('Internet','SMTP_Password','mypassword');
  ```
- To override the default character encoding, set the preference for email character encoding as follows:

  ```
  setpref('Internet','E_mail_Charset',encoding);
  ```

where *encoding* is a character vector specifying the character encoding, such as `'SJIS'`.

Alternatives

On Windows systems with Microsoft Outlook®, you can send email directly through Outlook by accessing the COM server with `actxserver`. For an example, see Solution 1-RTY6J.

See Also

`getpref` | `setpref`

Topics

"Specify Proxy Server Settings for Connecting to the Internet"

serial

(Not recommended) Create serial port object

Note `serial` is not recommended. Use `serialport` instead. See “Compatibility Considerations”.

Syntax

```
s = serial('port')
s = serial('port',Name,Value)
```

Description

`s = serial('port')` creates a serial port object `s` associated with the serial port specified by `'port'`. If `'port'` does not exist, or if it is in use, you cannot connect the serial port object to the device.

`s = serial('port',Name,Value)` creates a serial port object with the specified property names and property values. If an invalid property name or property value is specified, an error is returned, and the serial port object is not created.

Examples

Create Serial Port Object

This example shows how to create a serial port object.

Find available serial ports.

Use the seriallist function to find your available serial ports.

```
seriallist
```

Create a serial port object and assign port.

This example creates the serial port object s and associates it with the serial port COM1. You must specify the port as the first argument to create a serial port object.

```
s = serial('COM1');
```

Create a serial port object and specify properties.

This example creates the serial port object s2, associated with the serial port COM3, and sets properties. You can optionally set communication properties by specifying name-value pairs during object creation, after the port argument. This example sets the baud rate to 4800 and the terminator to CR. You can see these values in the object output.

```
s2 = serial('COM3','BaudRate',4800,'Terminator','CR')
```

Input Arguments

`'port'` — Serial port name
character vector | string

Serial port name, specified as a character vector or string. The `seriallist` function provides a list of available serial ports. You must specify the port to create a serial port object.

The port name depends on the platform that the serial port is on. This list is an example of serial constructors on different platforms:

Platform	Serial Port Constructor
Linux 64	`s = serial('/dev/ttyS0')`
macOS 64	`s = serial('/dev/tty.KeySerial1')`
Windows 64	`s = serial('COM1')`

Example: `s = serial('COM1')`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `s = serial('COM2','BaudRate',1200,'DataBits',7);`

For a list of serial port object properties that you can use with `serial`, refer to “Property Reference”.

Note Port must be the first argument used to create the serial object. You can then follow port with any number of supported name-value pairs.

`BaudRate` — Rate at which bits are transmitted

9600 (default) | double

Rate at which bits are transmitted, specified as the comma-separated pair consisting of `'BaudRate'` and a double. You configure baud rate as bits per second. The transferred bits include the start bit, the data bits, the parity bit (if used), and the stop bits. However, only the data bits are stored.

The baud rate is the rate at which information is transferred in a communication channel. In the serial port context, 9600 baud means that the serial port is capable of transferring a maximum of 9600 bits per second. If the information unit is one baud (one bit), the bit rate and the baud rate are identical. If one baud is given as 10 bits, (for example, eight data bits plus two framing bits), the bit rate is still 9600 but the baud rate is 9600/10, or 960. You always configure `BaudRate` as bits per second.

Note Both the computer and the peripheral device must be configured to the same baud rate before you can successfully read or write data.

Standard baud rates include 110, 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200, 128000, and 256000 bits per second.

You can also set the `BaudRate` property after creating the serial object using this syntax:

```
s.BaudRate = 4800;
```

Example: `s = serial('COM1','BaudRate',4800);`

Data Types: `double`

`ByteOrder` — Byte order of the device
littleEndian (default) | bigEndian

Byte order of the device, specified as the comma-separated pair consisting of `'ByteOrder'` and `littleEndian` or `bigEndian`. If `ByteOrder` is `littleEndian`, the device stores the first byte in the first memory address. If `ByteOrder` is `bigEndian`, the device stores the last byte in the first memory address.

For example, suppose the hexadecimal value 4F52 is to be stored in device memory. Because this value consists of two bytes, 4F and 52, two memory locations are used. Using big-endian format, 4F is stored first in the lower storage address. Using little-endian format, 52 is stored first in the lower storage address.

The byte order of `littleEndian` is the default and is used in read and write operations if you do not specify the property. You need to specify the property only to change the byte order to `bigEndian`.

You can also set the `ByteOrder` property after creating the serial object using this syntax:

```
s.ByteOrder = 'bigEndian';
```

Note Configure `ByteOrder` to the appropriate value for your device before performing a read or write operation. Refer to your device documentation for information about the order in which it stores bytes.

Example: `s = serial('COM1','ByteOrder','bigEndian');`

Data Types: `char` | `string`

`DataBits` — Number of data bits to transmit
8 (default) | 5 | 6 | 7

Number of data bits to transmit, specified as the comma-separated pair consisting of `'DataBits'` and `5`, `6`, `7`, or `8`, which is the default. Data is transmitted as a series of five, six, seven, or eight bits with the least significant bit sent first. At least seven data bits are required to transmit ASCII characters. Eight bits are required to transmit binary data. Five-bit and six-bit data formats are used for specialized communications equipment.

Note Both the computer and the peripheral device must be configured to transmit the same number of data bits.

In addition to the data bits, the serial data format consists of a start bit, one or two stop bits, and possibly a parity bit. You specify the number of stop bits with the `StopBits` property, and the type of parity checking with the `Parity` property.

You can also set the `DataBits` property after creating the serial object using this syntax:

```
s.DataBits = 7;
```

Example: `s = serial('COM1','DataBits',7);`

Data Types: `double`

`Parity` — Type of parity checking
none (default) | odd | even | mark | space

Type of parity checking, specified as the comma-separated pair consisting of `'Parity'` and `none`, `odd`, `even`, `mark`, or `space`.

`{none}`	Default. No parity checking. Parity checking is not performed and the parity bit is not transmitted.
`odd`	Odd parity checking. The number of mark bits (1s) in the data is counted, and the parity bit is asserted or unasserted to obtain an odd number of mark bits.
`even`	Even parity checking. The number of mark bits in the data is counted, and the parity bit is asserted or unasserted to obtain an even number of mark bits.
`mark`	Mark parity checking. The parity bit is asserted.
`space`	Space parity checking. The parity bit is unasserted.

Parity checking can detect errors of one bit only. An error in two bits might cause the data to have a seemingly valid parity, when in fact it is incorrect.

In addition to the parity bit, the serial data format consists of a start bit, between five and eight data bits, and one or two stop bits. You specify the number of data bits with the `DataBits` property, and the number of stop bits with the `StopBits` property.

You can also set the `Parity` property after creating the serial object using this syntax:

```
s.Parity = 'even';
```

Example: `s = serial('COM1','Parity','even');`

Data Types: `char` | `string`

`StopBits` — Number of bits used to indicate the end of a byte
1 (default) | 1.5 | 2

Number of bits used to indicate the end of a byte, specified as the comma-separated pair consisting of `'StopBits'` and `1`, `1.5`, or `2`. If `StopBits` is `1`, one stop bit is used to indicate the end of data transmission. If `StopBits` is `2`, two stop bits are used to indicate the end of data transmission. If `StopBits` is `1.5`, the stop bit is transferred for 150% of the normal time used to transfer one bit.

Note Both the computer and the peripheral device must be configured to transmit the same number of stop bits.

Summary of the possible values:

`{1}`	Default. One stop bit is transmitted to indicate the end of a byte.
`1.5`	The stop bit is transferred for 150% of the normal time used to transfer one bit.
`2`	Two stop bits are transmitted to indicate the end of a byte.

In addition to the stop bits, the serial data format consists of a start bit, between five and eight data bits, and possibly a parity bit. You specify the number of data bits with the `DataBits` property, and the type of parity checking with the `Parity` property.

You can also set the `StopBits` property after creating the serial object using this syntax:

```
s.StopBits = 2;
```

Example: `s = serial('COM1','StopBits',2);`

Data Types: `double`

`Terminator` — Terminator character
string

Terminator character, specified as the comma-separated pair consisting of `'Terminator'` and a string. You can configure `Terminator` to an integer value ranging

from 0 to 127, which represents the ASCII code for the character, or you can configure `Terminator` to the ASCII character. For example, to configure `Terminator` to a carriage return, specify the value to be `CR` or `13`. To configure `Terminator` to a linefeed, specify the value to be `LF` or `10`. You can also set `Terminator` to `CR/LF` or `LF/CR`. If `Terminator` is `CR/LF`, the terminator is a carriage return followed by a line feed. If Terminator is `LF/CR`, the terminator is a linefeed followed by a carriage return. Note that there are no integer equivalents for these two values.

Additionally, you can set `Terminator` to a 1-by-2 cell array. The first element of the cell is the read terminator and the second element of the cell array is the write terminator.

When performing a write operation using the `fprintf` function, all occurrences of `\n` are replaced with the `Terminator` property value. Note that `%s\n` is the default format for `fprintf`. A read operation with `fgetl`, `fgets`, or `fscanf` completes when the `Terminator` value is read. The terminator is ignored for binary operations.

You can also use the terminator to generate a bytes-available event when the `BytesAvailableFcnMode` is set to `terminator`.

You can also set the `Terminator` property after creating the serial object, using this syntax:

```
s.Terminator = 'CR';
```

Example: `s = serial('COM1','Terminator','CR');`

Data Types: `char` | `string`

Tips

Refer to “Property Reference” for a list of serial port object properties that you can use with `serial`.

Before you can communicate with the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a Status property value of `open`. An error is returned if you attempt a read or write operation while the object is not connected to the device. You can connect only one serial port object to a given serial port.

Compatibility Considerations

`serial` function is not recommended

Not recommended starting in R2019b

`serial` and its object properties are not recommended. Use `serialport` and its properties instead.

This example shows how to connect to a serial port device using the recommended functionality.

Functionality	Use This Instead

`s = serial("COM1");`
`s.BaudRate = 115200;`
`fopen(s)`
	`s = serialport("COM1",115200);`

See “Transition Your Code to serialport Interface” for more information about using the recommended functionality.

See Also

Functions

`serialport`

Topics

“Create a Serial Port Object”
“Property Reference”

Introduced before R2006a

Serial Port Properties

Configure serial port communications

Description

Serial port properties are used to configure communications using the `serial` object and to configure read and write behavior.

Some properties can be set during object creation with the `serial` function. See the `serial` reference page for information about those properties, which include `Port`, `BaudRate`, `ByteOrder`, `DataBits`, `Parity`, `StopBits`, and `Terminator`.

Read and write properties are used to configure factors involved in the reading and writing of data, such as setting a timeout for completing the operation. Use control pin properties to configure the state of control pins and data flow control. Use recording properties to set up and control recording information to disk.

Note This sample syntax for all the properties assumes you have created a serial object, `s`. Many of the properties can only be set before calling `fopen` on the object. Some can be changed while the object is open.

Properties can be set after you create the `serial` object. A read-only property is called as follows:

```
s = serial('COM1');
s.TransferStatus
ans =
     idle
```

A property you can configure is set as follows:

```
s = serial('COM1');
s.Timeout = 30;
```

Properties

Read and Write Properties

`BytesAvailable` — Number of bytes available in the input buffer

0 (default) | double

Number of bytes available in the input buffer, specified as a double. This read-only property indicates the number of bytes currently available to be read from the input buffer. The property value is continuously updated as the input buffer is filled, and is set to `0` after the `fopen` function is issued.

You can make use of `BytesAvailable` only when reading data asynchronously. This is because when reading data synchronously, control is returned to the MATLAB command line only after the input buffer is empty. Therefore, the `BytesAvailable` value is always 0.

The `BytesAvailable` value can range from zero to the size of the input buffer. Use the `InputBufferSize` property to specify the size of the input buffer. Use the `ValuesReceived` property to return the total number of values read.

Example: `s.BytesAvailable`

Data Types: `double`

`InputBufferSize` — Size of the input buffer in bytes

512 (default) | double

Size of the input buffer in bytes, specified as a double. You configure `InputBufferSize` as the total number of bytes that can be stored in the input buffer during a read operation.

A read operation is terminated if the amount of data stored in the input buffer equals the `InputBufferSize` value. You can read text data with the `fgetl`, `fget`, or `fscanf` functions. You can read binary data with the `fread` function.

You can configure `InputBufferSize` only when the serial port object is disconnected from the device. You can configure it before calling the `fopen` function. You disconnect an object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

If you configure `InputBufferSize` while there is data in the input buffer, that data is purged.

Example: `s.InputBufferSize = 768;`

Data Types: `double`

`ReadAsyncMode` — Specify whether an asynchronous read operation is continuous or manual

continuous (default) | manual

Specify whether an asynchronous read operation is continuous or manual, specified as `manual` or `continuous`. If `ReadAsyncMode` is `continuous`, the serial port object continuously queries the device to determine if data is available to be read. If data is available, it is automatically read and stored in the input buffer. If issued, the `readasync` function is ignored.

If `ReadAsyncMode` is `manual`, the object does not query the device to determine if data is available to be read. Instead, you must manually issue the `readasync` function to perform an asynchronous read operation. Because `readasync` checks for the terminator, this function can be slow. To increase speed, configure `ReadAsyncMode` to `continuous`.

Note If the device is ready to transmit data, it will do so regardless of the `ReadAsyncMode` value. Therefore, if `ReadAsyncMode` is `manual` and a read operation is not in progress, data might be lost. To guarantee that all transmitted data is stored in the input buffer, you should configure `ReadAsyncMode` to `continuous`.

To determine the amount of data available in the input buffer, use the `BytesAvailable` property. For either `ReadAsyncMode` value, you can bring data into the MATLAB workspace with one of the synchronous read functions such as `fscanf`, `fgetl`, `fgets`, or `fread`.

Example: `s.ReadAsyncMode = 'manual';`

Data Types: `char` | `string`

`Timeout` — Waiting time to complete a read or write operation

10 (default) | double

Waiting time to complete a read or write operation, specified as a double. You configure `Timeout` to be the maximum time (in seconds) to wait to complete a read or write operation. The default value of `10` seconds is used if you do not specify a different value. Timeouts are rounded upwards to full seconds.

If a timeout occurs, the read or write operation aborts. Additionally, if a timeout occurs during an asynchronous read or write operation, then:

- An error event is generated.
- The callback function specified for `ErrorFcn` is executed.

Example: `s.Timeout = 30;`

Data Types: `double`

`TransferStatus` — Status of asynchronous read or write operation
idle (default) | read | write | read&write

Status of asynchronous read or write operation, specified as `idle`, `read`, `write`, or `read&write`. This read-only property indicates if an asynchronous read or write operation is in progress. If `TransferStatus` is `idle`, no asynchronous read or write operations are in progress. If it is `read`, an asynchronous read operation is in progress. If it is `write`, an asynchronous write operation is in progress. If `TransferStatus` is `read&write`, both an asynchronous read and an asynchronous write operation are in progress.

You can write data asynchronously using the `fprintf` or `fwrite` functions. You can read data asynchronously using the `readasync` function, or by configuring the `ReadAsyncMode` property to `continuous`. While `readasync` is executing, `TransferStatus` might indicate that data is being read even though data is not filling the input buffer. If `ReadAsyncMode` is `continuous`, `TransferStatus` indicates that data is being read only when data is filling the input buffer.

You can execute an asynchronous read and an asynchronous write operation simultaneously because serial ports have separate read and write pins.

Summary the possible values:

`{idle}`	No asynchronous operations are in progress.
`read`	An asynchronous read operation is in progress.
`write`	An asynchronous write operation is in progress.

`read&write`	Asynchronous read and write operations are in progress.

Example: `s.TransferStatus`

Data Types: `char` | `string`

`ValuesReceived` — Total number of values read from the device
0 (default) | double

Total number of values read from the device, specified as a double. This is a read-only property, and the value is updated after each successful read operation and set to `0` after the `fopen` function is issued. If the terminator is read from the device, then this value is reflected by `ValuesReceived`.

If you are reading data asynchronously, use the `BytesAvailable` property to return the number of bytes currently available in the input buffer.

When performing a read operation, the received data is represented by values rather than bytes. A value consists of one or more bytes. For example, one `uint32` value consists of four bytes.

For example, create a serial port object associated with the serial port COM1, and open the connection.

```
s = serial('COM1');
fopen(s)
```

If you write the `RS232?` command, and read back the response using `fscanf`, `ValuesReceived` is `17` because the instrument is configured to send the `LF` terminator.

```
fprintf(s,'RS232?')
out = fscanf(s)
out =
9600;0;0;NONE;LF
s.ValuesReceived
ans =
    17
```

Example: `s.ValuesReceived`

Data Types: `double`

`BytesToOutput` — Number of bytes currently in the output buffer

0 (default) | double

Number of bytes currently in the output buffer, specified as a double. This read-only property indicates the number of bytes currently in the output buffer waiting to be written to the device. The property value is continuously updated as the output buffer is filled and emptied, and is set to `0` after the `fopen` function is issued.

You can make use of `BytesToOutput` only when writing data asynchronously. This is because when writing data synchronously, control is returned to the MATLAB command line only after the output buffer is empty. Therefore, the `BytesToOutput` value is always `0`.

Use the `ValuesSent` property to return the total number of values written to the device.

Note If you attempt to write out more data than can fit in the output buffer, an error is returned and `BytesToOutput` is `0`. Specify the size of the output buffer with the `OutputBufferSize` property.

Example: `s.BytesToOutput`

Data Types: `double`

`OutputBufferSize` — Size of the output buffer in bytes

512 (default) | double

Size of the output buffer in bytes, specified as a double. You configure `OutputBufferSize` as the total number of bytes that can be stored in the output buffer during a write operation.

You can configure `OutputBufferSize` only when the serial port object is disconnected from the device. You can configure it before calling the `fopen` function. You disconnect an object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

An error occurs if the output buffer cannot hold all the data to be written. You write text data with the `fprintf` function. You write binary data with the `fwrite` function.

Example: `s.OutputBufferSize = 256;`

Data Types: `double`

`ValuesSent` — Total number of values written to the device

0 (default) | double

Total number of values written to the device, specified as a double. This is a read-only property, and the value is updated after each successful write operation and set to `0` after the `fopen` function is issued. If you are writing the terminator, `ValuesSent` reflects this value.

If you are writing data asynchronously, use the `BytesToOutput` property to return the number of bytes currently in the output buffer.

When performing a write operation, the transmitted data is represented by values rather than bytes. A value consists of one or more bytes. For example, one `uint32` value consists of four bytes.

For example, create a serial port object associated with the serial port COM1 and open the connection.

```
s = serial('COM1');
fopen(s)
```

If you write the `*IDN?` command using the `fprintf` function, `ValuesSent` is `6` because the default data format is `%s\n`, and the terminator was written.

```
fprintf(s,'*IDN?')
s.ValuesSent
ans =
    6
```

Example: `s.ValuesSent`

Data Types: `double`

`Status` — Status of serial port device connection

closed (default) | open

Status of serial port device connection, returned as `closed` or `open`. This read-only property indicates whether the serial port object is connected to the device. If `Status` is `closed`, the serial port object is not connected to the device. If `Status` is `open`, the serial port object is connected to the device.

Before you can write or read data, you must connect the serial port object to the device with the `fopen` function. Use the `fclose` function to disconnect a serial port object from the device.

Example: `s.Status`

Data Types: `char` | `string`

Control Pin Properties

`DataTerminalReady` — State of the DTR pin

on (default) | off

State of the DTR pin, specified as `on` or `off`. If `DataTerminalReady` is `on`, the Data Terminal Ready (DTR) pin is asserted. If `DataTerminalReady` is `off`, the DTR pin is unasserted.

In normal usage, the DTR and Data Set Ready (DSR) pins work together, and are used to signal if devices are connected and powered. However, there is nothing in the RS-232 standard that states the DTR pin must be used in any specific way. For example, DTR and DSR might be used for handshaking. You should refer to your device documentation to determine its specific pin behavior.

You can return the value of the DSR pin with the `PinStatus` property.

Example: `s.DataTerminalReady = 'off';`

Data Types: `char` | `string`

`FlowControl` — Data flow control method

none (default) | hardware | software

Data flow control method, specified as `none`, `hardware`, or `software`. If `FlowControl` is `none`, data flow control (handshaking) is not used. If `FlowControl` is `hardware`, hardware handshaking is used to control data flow. If `FlowControl` is `software`, software handshaking is used to control data flow.

Hardware handshaking typically utilizes the Request to Send (RTS) and Clear to Send (CTS) pins to control data flow. Software handshaking uses control characters (Xon and Xoff) to control data flow.

You can return the value of the CTS pin with the `PinStatus` property. You can specify the value of the RTS pin with the `RequestToSend` property. However, if `FlowControl` is `hardware`, and you specify a value for `RequestToSend`, that value might not be honored.

Note Although you might be able to configure your device for both hardware handshaking and software handshaking at the same time, MATLAB does not support this behavior.

Example: `s.FlowControl = 'hardware';`

Data Types: `char` | `string`

`PinStatus` — State of the CD, CTS, DSR, and RI pins
structure

State of the CD, CTS, DSR, and RI pins, returned as a structure. This read-only property returns a structure array that contains the fields `CarrierDetect`, `ClearToSend`, `DataSetReady` and `RingIndicator`. These fields indicate the state of the Carrier Detect (CD), Clear to Send (CTS), Data Set Ready (DSR) and Ring Indicator (RI) pins, respectively.

`PinStatus` can be `on` or `off` for any of these fields. A value of `on` indicates the associated pin is asserted. A value of `off` indicates the associated pin is unasserted. A pin status event occurs when any of these pins changes its state. A pin status event executes the call back function specified by `PinStatusFcn`.

In normal usage, the Data Terminal Ready (DTR) and DSR pins work together, while the Request to Send (RTS) and CTS pins work together. You can specify the state of the DTR pin with the `DataTerminalReady` property. You can specify the state of the RTS pin with the `RequestToSend` property.

Example: `s.PinStatus`

Data Types: `struct`

`RequestToSend` — State of the RTS pin
on (default) | off

State of the RTS pin, specified as `on` or `off`. If `RequestToSend` is `on`, the Request to Send (RTS) pin is asserted. If `RequestToSend` is `off`, the RTS pin is unasserted.

In normal usage, the RTS and Clear to Send (CTS) pins work together, and are used as standard handshaking pins for data transfer. In this case, RTS and CTS are automatically managed by the DTE and DCE. However, there is nothing in the RS-232 standard that requires the RTS pin must be used in any specific way. Therefore, if you manually configure the `RequestToSend` value, it is probably for nonstandard operations.

If your device does not use hardware handshaking in the standard way, and you need to manually configure `RequestToSend`, configure the `FlowControl` property to `none`. Otherwise, the `RequestToSend` value that you specify might not be honored. Refer to your device documentation to determine its specific pin behavior.

You can return the value of the CTS pin with the `PinStatus` property.

Example: `s.RequestToSend = 'off';`

Data Types: `char` | `string`

Recording Properties

`RecordDetail` — Detail level of information saved to a record file

compact (default) | verbose

Detail level of information saved to a record file, specified as `compact` or `verbose`. If `RecordDetail` is `compact`, the number of values written to the device, the number of values read from the device, the data type of the values, and event information are saved to the record file. If `RecordDetail` is `verbose`, the data written to the device and the data read from the device are also saved to the record file.

Summary of the possible values:

`{compact}`	The number of values written to the device, the number of values read from the device, the data type of the values, and event information are saved to the record file.
`verbose`	The data written to the device, and the data read from the device are also saved to the record file.

Example: `s.RecordDetail = 'verbose';`

Data Types: `char` | `string`

`RecordMode` — Method for saving data and event information in record files

overwrite (default) | append | index

Method for saving data and event information in record files, specified as `overwrite`, `append`, or `index`. If `RecordMode` is `overwrite`, the record file is overwritten each time recording is initiated. If `RecordMode` is `append`, data is appended to the record file each time recording is initiated. If `RecordMode` is `index`, a different record file is created each time recording is initiated, each with an indexed filename.

You can configure `RecordMode` only when the object is not recording. You terminate recording with the `record` function. An object that is not recording has a `RecordStatus` property value of `off`.

You specify the record filename with the `RecordName` property. The indexed filename follows a prescribed set of rules.

Summary of the possible values:

`{overwrite}`	The record file is overwritten.
`append`	Data is appended to an existing record file.
`index`	A different record file is created, each with an indexed filename.

For example, record serial data using the record properties. Create the serial port object and open the connection.

```
s = serial('COM1');
fopen(s)
```

Specify the record filename with the `RecordName` property, configure `RecordMode` to `index`, and initiate recording.

```
s.RecordName = 'MyRecord.txt';
s.RecordMode = 'index';
record(s)
```

The record filename is automatically updated with an indexed filename after recording is turned off.

```
record(s,'off')
s.RecordName
ans =
MyRecord01.txt
```

Disconnect `s` from the peripheral device, remove `s` from memory, and remove `s` from the MATLAB workspace.

```
fclose(s)
delete(s)
clear s
```

Example: `s.RecordMode = 'index';`

Data Types: `char` | `string`

RecordName — Name of the record file

string

Name of the record file, specified as a string. You can specify any value for `RecordName` - including a directory path - provided the file name is supported by your operating system.

The default record filename is `record.txt`, which is used if you record a data file and do not specify a different name.

MATLAB supports any file name supported by your operating system. You can access the file using the `type` function. For example, if you name the record file `MyRecord.txt`, to type this file at the MATLAB command line, enter:

```
type('MyRecord.txt')
```

You can specify whether data and event information are saved to one disk file or to multiple disk files with the `RecordMode` property. If `RecordMode` is `index`, the filename follows a prescribed set of rules.

You can configure `RecordName` only when the object is not recording. You terminate recording with the `record` function. An object that is not recording has a `RecordStatus` property value of `off`.

Example: `s.RecordName = 'MonthlyDataFile_April';`

Data Types: `char` | `string`

RecordStatus — Status of recording serial data and event information

off (default) | on

Status of recording serial data and event information, returned as `on` or `off`. This read-only property indicates whether recording is on or off, which is controlled by the `record` function. If `RecordStatus` is `off`, then data and event information are not saved to a record file. If `RecordStatus` is `on`, then data and event information are saved to the record file specified by `RecordName`.

Use the `record` function to initiate or complete recording. `RecordStatus` is automatically configured to reflect the recording state.

Example: `s.RecordStatus`

Data Types: `char` | `string`

See Also

Introduced before R2006a

serialbreak

(Not recommended) Send break to device connected to serial port

Note This `serial` object function is not recommended. Use `serialport` object functions instead. See “Compatibility Considerations”.

Syntax

```
serialbreak(obj)
serialbreak(obj,time)
```

Description

`serialbreak(obj)` sends a break of 10 milliseconds to the device connected to the serial port object, `obj`.

`serialbreak(obj,time)` sends a break to the device with a duration, in milliseconds, specified by `time`. Note that the duration of the break might be inaccurate under some operating systems.

Tips

For some devices, the break signal provides a way to clear the hardware buffer.

Before you can send a break to the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to send a break while `obj` is not connected to the device.

`serialbreak` is a synchronous function, and blocks the command line until execution is complete.

If you issue `serialbreak` while data is being asynchronously written, an error is returned. In this case, you must call the `stopasync` function or wait for the write operation to complete.

Compatibility Considerations

`serial` object interface is not recommended

Not recommended starting in R2019b

Use of this function with a `serial` object is not recommended. To access a serial port device, use a `serialport` object with its functions and properties instead.

See “Transition Your Code to serialport Interface” for more information about using the recommended functionality.

See Also

Functions

`fopen` | `stopasync`

Introduced before R2006a

seriallist

(Not recommended) List of serial ports connected to your system

Note `seriallist` is not recommended. Use `serialportlist` instead. See "Compatibility Considerations".

Syntax

```
seriallist
```

Description

`seriallist` returns a list of all serial ports on a system. The list includes virtual serial ports provided by USB-to-serial devices and Bluetooth Serial Port Profile devices. This provides a list of the serial ports that you have access to on your computer and could use for serial port communication.

Examples

Display List of Serial Ports

The `seriallist` function displays a list of all serial ports on a system.

Display available serial ports.

```
seriallist
```

```
ans =

  1x2 string array

  "COM1"    "COM3"
```

You can now create a `serial` object and communicate over the serial port.

Compatibility Considerations

`seriallist` function is not recommended

Not recommended starting in R2019b

`seriallist` is not recommended. Use `serialportlist` instead.

This example shows how to view serial port devices using the recommended functionality.

Functionality	Use This Instead

`s = seriallist`
 	
`ans =`
 	
`"COM3"`
	`s = serialportlist`
	
	`ans =`
	
	`"COM3"`

See “Transition Your Code to serialport Interface” for more information about using the recommended functionality.

See Also

Functions

`serialportlist`

Topics

“Create a Serial Port Object”
“Overview of the Serial Port”

Introduced in R2017a

serialport

Connection to serial port

Description

A `serialport` object represents a serial client for communication with the serial port. After creating the object, use dot notation to set its properties.

Creation

Syntax

```
s = serialport(port,baudrate)
s = serialport(port,Name,Value)
s = serialport
```

Description

`s = serialport(port,baudrate)` connects to the serial port specified by `port` with a baud rate of `baudrate`.

`s = serialport(port,Name,Value)` connects to the serial port and sets additional properties using optional name-value pair arguments.

`s = serialport`, without arguments, connects to the serial port using the property settings of your last cleared `serialport` object instance. The retained properties are `Port`, `BaudRate`, `ByteOrder`, `FlowControl`, `StopBits`, `DataBits`, `Parity`, `Timeout`, and `Terminator`. See “Properties” on page 1-13984.

Input Arguments

`port` — Serial port name
string | char

Serial port name, specified as a string or character vector. Use `serialportlist` to get a list of connected ports.

Example: `"COM2"`

`baudrate` — Baud rate
double

Baud rate for serial communication, specified as a double.

Example: `9600`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

You can use Name-Value pairs to set the `DataBits`, `Parity`, `StopBits`, `FlowControl`, `ByteOrder`, and `Timeout` object properties. See "Properties" on page 1-13984 for their data types and allowed values.

Example: `"Timeout",30`

Properties

`Port` — Serial port for connection
string

This property is read-only.

Serial port for connection, returned as a string or character vector.

Example: `"COM1"`

Data Types: `string`

`BaudRate` — Communication speed
double

Communication speed in bits per second, returned as a positive integer double.

Example: `14400`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Parity` — Parity
`"none"` (default) | `"even"` | `"odd"`

Parity to check whether data has been lost or written, returned as `"none"`, `"even"`, or `"odd"`.

Example: `"odd"`

Data Types: `char` | `string`

`DataBits` — Number of bits to represent one character of data
8 (default) | 7 | 6 | 5

Number of bits to represent one character of data, returned as 8, 7, 6, or 5.

Example: 8

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`StopBits` — Pattern of bits that indicates the end of a character
1 (default) | `1.5` | 2

Pattern of bits that indicates the end of a character or of the whole transmission, returned as 1, `1.5`, or 2.

Example: 1

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`FlowControl` — Mode for managing data transmission rate
`"none"` (default) | `"hardware"` | `"software"`

Mode for managing data transmission rate, returned as `"none"`, `"hardware"`, or `"software"`.

Example: `"software"`

Data Types: `char` | `string`

`ByteOrder` — Sequential order of bytes
`"little-endian"` (default) | `"big-endian"`

Sequential order in which bytes are arranged into larger numerical values, returned as `"little-endian"` or `"big-endian"`.

Example: `"little-endian"`

Data Types: `char` | `string`

`Timeout` — Allowed time to complete operations
10 (default) | numeric

Allowed time in seconds to complete read and write operations, returned as a numeric value.

Example: `60`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`NumBytesAvailable` — Number of bytes available to read
numeric

This property is read-only.

Number of bytes available to read, returned as a numeric value.

Example: `1024`

Data Types: `double`

`NumBytesWritten` — Number of bytes written to the serial port
0 (default) | numeric

This property is read-only.

Number of bytes written to the serial port, returned as a numeric value.

Example: `512`

Data Types: `double`

`Terminator` — Terminator character for line
`"LF"` (default) | `"CR"` | `"CR/LF"`

Terminator character for reading and writing lines, returned as `"LF"`, `"CR"`, or `"CR/LF"`. Set this property with the `configureTerminator` function.

Example: `"CR"`

Data Types: `char` | `string`

`BytesAvailableFcn` — Bytes available callback function
function handle

Bytes available callback function, returned as a function handle. The callback is triggered by receiving a certain number of bytes or a terminator. Set this property with the `configureCallback` function.

Example: `@myFun`

Data Types: `function_handle`

`BytesAvailableFcnCount` — Number of bytes of data to trigger callback
64 (default) | numeric value

Number of bytes of data to trigger callback, returned as a double. This value is used only when the `BytesAvailableFcnMode` property is `"byte"`. Set these properties with the `configureCallback` function.

Example: 128

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`BytesAvailableFcnMode` — Bytes available callback trigger mode
`"off"` (default) | `"byte"` | `"terminator"`

Bytes available callback trigger mode, returned as `"off"`, `"byte"`, or `"terminator"`. This setting determines if the callback is off, triggered by a number of bytes, or by a terminator. Set this property with the `configureCallback` function.

Example: `"off"`

Data Types: `char` | `string`

`ErrorOccurredFcn` — Callback function triggered by error
function handle

Callback function triggered by error, returned as a function handle.

Example: `@myErrFun`

Data Types: `function_handle`

`UserData` — General purpose property for user data
any type

General purpose property for user data, returned as any type.

Example: `datetime()`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`
Complex Number Support: Yes

Object Functions

read	Read data from serial port
readline	Read line of ASCII string data from serial port
write	Write data to serial port
writeline	Write line of ASCII data to serial port
flush	Flush serial port buffers
configureTerminator	Set terminator for ASCII string communication on serial port
configureCallback	Set serial port callback function and trigger
getpinstatus	Get serial pin status
setRTS	Set serial RTS pin
setDTR	Set serial DTR pin

Examples

Read Data from Serial Port

Read 16 values of uint32 data from the `COM3` serial port.

```
s = serialport("COM3",9600,"Timeout",5);
data = read(s,16,"uint32");
```

See Also

Functions
`serialportlist`

Introduced in R2019b

serialportlist

List of serial ports connected to your system

Syntax

```
serialportlist
serialportlist("all")
serialportlist("available")
```

Description

`serialportlist` or `serialportlist("all")` returns a list of all serial ports on a system. The list includes virtual serial ports provided by USB-to-serial devices and Bluetooth Serial Port Profile devices. The list shows all the serial ports you can access on your computer and can use for serial port communication.

`serialportlist("available")` returns a list of only those serial ports on your system that are available at this time.

Examples

Get List of Serial Ports

Identify serial ports on your system.

```
ports = serialportlist
```

```
ports =

  1×4 string array

    "COM1"    "COM3"    "COM11"    "COM12"
```

List only those ports that are available.

```
freeports = serailportlist("available")
```

```
freeports =

  1×2 string array

    "COM1"    "COM12"
```

See Also

Functions
`serialport`

Introduced in R2019b

configureCallback

Set serial port callback function and trigger

Syntax

```
configureCallback(device,"terminator",callbackFcn)
configureCallback(device,"byte",count,callbackFcn)
configureCallback(device,"off")
```

Description

`configureCallback(device,"terminator",callbackFcn)` sets `callbackFcn` to trigger whenever a terminator is available to read.

`configureCallback(device,"byte",count,callbackFcn)` sets `callbackFcn` to trigger whenever `count` number of bytes are available to read.

`configureCallback(device,"off")` turns off callbacks.

The `serialport` object `BytesAvailableFcnMode` property indicates the mode as `"terminator"`, `"byte"`, or `"off"`. The `BytesAvailableFcn` property indicates callback function specified by `callbackFcn`. The `BytesAvailableFcnCount` property indicates the `count` setting.

Examples

Configure Serial Port Callback

Configure serial port interface callbacks, then clear callbacks.

Set the callback to trigger when 50 bytes of data are available to be read.

```
s = serialport("COM3",9600);
configureCallback(s,"byte",50,@callbackFcn)
```

Set the callback to trigger when a terminator is available to be read.

```
configureCallback(s,"terminator",@callbackFcn)
```

Turn the callback off.

```
configureCallback(s,"off")
```

Input Arguments

device — Serial port
serialport object

Serial port, specified as a `serialport` object.

Example: `serialport()`

count — Number of bytes of data to trigger callback
positive integer value

Number of bytes of available data to trigger callback, specified as a positive integer value.

Example: 128

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

callbackFcn — Callback function to run when triggered
function handle

Callback function to run when triggered, specified as a function handle.

Example: `@myFunction`

Data Types: `function_handle`

See Also

Functions
`configureTerminator` | `serialport`

Introduced in R2019b

configureTerminator

Set terminator for ASCII string communication on serial port

Syntax

```
configureTerminator(device,term)
configureTerminator(device,readterm,writeterm)
```

Description

`configureTerminator(device,term)` defines the terminator for both read and write communications with the specified serial port device. Allowed terminator values are `"LF"` (default), `"CR"`, `"CR/LF"`, and integer values from `0` to `255`.

`configureTerminator(device,readterm,writeterm)` defines separate terminators for read and write communications with the specified serial port device.

Examples

Set Same Terminator for Read and Write

Set both the read and write terminators to `"CR/LF"`, and then confirm the setting.

```
s = serialport("COM3",9600);
configureTerminator(s,"CR/LF")
s.Terminator
```

```
ans =

    "CR/LF"
```

Set Different Terminators for Read and Write

Set the read terminator to "CR" and the write terminator to 10, then confirm the settings.

```
s = serialport("COM3",9600);
configureTerminator(s,"CR",10)
s.Terminator
```

```
ans =

  1×2 cell array

    {["CR"]}    {[10]}
```

Input Arguments

device — Serial port
serialport object

Serial port, specified as a serialport object.

Example: serialport()

term — ASCII terminator
"LF" (default) | "CR" | "CR/LF" | 0 to 255

ASCII terminator for serial port read and write communication, specified as a string, character vector, or numeric integer value 0 to 255. Use this form when setting the same terminator for both read and write. When reading from the port with a terminator value of "CR/LF", the read terminates on the occurrence of CR and LF together. When writing to the port with a terminator value of "CR/LF", the write terminates by adding both.

Example: "CR"

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | char | string

readterm,writeterm — ASCII terminators for read and write
"LF" (default) | "CR" | "CR/LF" | 0 to 255

ASCII terminators for serial port read or write communication, specified as a string, character vector, or numeric integer value 0 to 255. Use this form when setting different terminators for read and write. When reading from the port with a terminator value of

`"CR/LF"`, the read terminates on the occurrence of CR and LF together. When writing to the port with a terminator value of `"CR/LF"`, the write terminates by adding both.

Example: `"CR"`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

See Also

Functions
`configureCallback` | `readline` | `serialport` | `writeline`

Introduced in R2019b

flush

Flush serial port buffers

Syntax

```
flush(device)
flush(device,"input")
flush(device,"output")
```

Description

`flush(device)` flushes all data from both the input and output buffers of the specified serial port.

`flush(device,"input")` flushes only the input buffer of the specified serial port.

`flush(device,"output")` flushes only the output buffer of the specified serial port.

Examples

Flush Serial Port Inputs and Outputs

Use the default to flush both serial port buffers.

```
s = serialport("COM3",9600);
  ⋮
flush(s);
```

Input Arguments

device — Serial port
`serialport` object

Serial port, specified as a `serialport` object.

Example: `serialport()`

See Also

Functions
`getpinstatus` | `read` | `serialport` | `write`

Introduced in R2019b

getpinstatus

Get serial pin status

Syntax

```
status = getpinstatus(device)
```

Description

`status = getpinstatus(device)` gets the serial pin status and returns it as a structure to `status`.

Examples

Get Serial Pin Status

Get the serial pin status for the specified port.

```
s = serialport("COM3",9600);
 ⋮
status = getpinstatus(s)
```

```
status =

  struct with fields:

      ClearToSend: 1
     DataSetReady: 1
```

```
    CarrierDetect: 1
    RingIndicator: 0
```

Input Arguments

device — Serial port
serialport object

Serial port, specified as a `serialport` object.

Example: `serialport()`

Output Arguments

status — Pin status
structure

Pin status, returned as a structure with the logical type fields named `ClearToSend`, `DataSetReady`, `CarrierDetect`, and `RingIndicator`.

See Also

Functions
`flush` | `serialport`

Introduced in R2019b

read

Read data from serial port

Syntax

```
data = read(device,count,precision)
```

Description

`data = read(device,count,precision)` reads the the number of values specified by `count`, with the data type specified by `precision`, from the device connected to the serial port `device`. For all numeric `precision` types, `data` is a row vector of double values. For the text type `precision` values of `'char'` or `'string'`, `data` is of the specified type.

The function blocks MATLAB and waits until the specified number of values are read from the serial port, or a timeout occurs.

Examples

Read Data from Serial Port

Read five values of uint32 data from the `COM3` serial port. These five values at 4 bytes each are a total of 20 bytes.

```
s = serialport("COM3",9600);
data = read(s,5,"uint32");
```

Input Arguments

device — Serial port
`serialport` object

Serial port, specified as a `serialport` object.

Example: `serialport()`

`count` — Number of items to read
numeric

Number of items to read, specified as a scalar positive integer numeric value. You cannot specify a `count` of `inf` or `NaN`. If `count` is greater than the `NumBytesAvailable` property of `device`, the function blocks MATLAB and waits until the specified amount of data is read or a timeout occurs.

Example: `20`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`precision` — Size and format of each value
`'uint8'` | `'int8'` | `'uint16'` | `'int16'` | `'uint32'` | `'int32'` | `'uint64'` | `'int64'` | `'single'` | `'double'` | `'char'` | `'string'`

Size and format of each value, specified as a character vector or string. `precision` determines the number of bits to read for each value and the interpretation of those bits as a MATLAB data type.

Example: `'int16'`

Data Types: `char` | `string`

Output Arguments

`data` — Numeric or ASCII data
1-by-N double | string | char

Numeric or ASCII data read, returned as a 1-by-N vector of doubles or text. For all numeric `precision` types, `data` is a row vector of double values. For the text type `precision` values of `'char'` or `'string'`, `data` is of the specified type.

See Also

Functions
`readline` | `serialport` | `serialportlist`

Introduced in R2019b

readline

Read line of ASCII string data from serial port

Syntax

```
data = readline(device)
```

Description

`data = readline(device)` reads data until the first occurrence of the terminator and returns `data` as a string. The function blocks MATLAB waits until the terminator is reached or a timeout occurs.

Examples

Read Line of ASCII Data from Device

Read a string of ASCII data from the serial port.

```
s = serialport("COM3",9600);
configureTerminator(s,"CR");
data = readline(s);
```

Input Arguments

device — Serial port
`serialport` object

Serial port, specified as a `serialport` object.

Example: `serialport()`

Output Arguments

data — ASCII data
string

ASCII data read, returned as a string. The terminator is not included.

See Also

Functions
`configureTerminator` | `read` | `serialport` | `serialportlist`

Introduced in R2019b

setDTR

Set serial DTR pin

Syntax

```
setDTR(device,true)
setDTR(device,false)
```

Description

`setDTR(device,true)` sets the specified serial data terminal ready (DTR) pin.

`setDTR(device,false)` resets the specified serial DTR pin.

Examples

Set and Reset DTR Pin

Set and then reset the serial DTR pin.

```
s = serialport("COM3",9600);
  ⋮
setDTR(s,true)
  ⋮
setDTR(s,false)
```

Input Arguments

device — Serial port
`serialport` object

Serial port, specified as a `serialport` object.

Example: `serialport()`

See Also

Functions
`serialport` | `setRTS`

Introduced in R2019b

setRTS

Set serial RTS pin

Syntax

```
setRTS(device,true)
setRTS(device,false)
```

Description

`setRTS(device,true)` sets the specified serial request to send (RTS) pin.

`setRTS(device,false)` resets the specified RTS pin.

Examples

Set and Reset RTS Pin

Set and then reset the serial RTS pin.

```
s = serialport("COM3",9600);
  ⋮
setRTS(s,true)
  ⋮
setRTS(s,false)
```

Input Arguments

device — Serial port
`serialport` object

Serial port, specified as a `serialport` object.

Example: `serialport()`

See Also

Functions
`serialport` | `setDTR`

Introduced in R2019b

write

Write data to serial port

Syntax

```
write(device,data,precision)
```

Description

`write(device,data,precision)` writes the 1-by-N matrix of `data` to the specified serial port. The written data has the specified precision regardless of the format in the matrix.

The `write` function blocks MATLAB and waits until finished writing data to the serial port.

Examples

Write Numeric Data to Serial Port

Write the values `[1,2,3,4,5]` in uint8 format to the serial port device. These five values at 1 byte each are a total of 5 bytes.

```
s = serialport("COM3",9600);
  ⋮
write(s,1:5,"uint8")
```

Input Arguments

`device` — Serial port
`serialport` object

Serial port, specified as a `serialport` object.

Example: `serialport()`

`data` — Numeric or ASCII data
1-by-N numeric array | string | char

Numeric or ASCII data to write to serial port, specified as a 1-by-N vector of numeric values or text. For all numeric `precision` types, `data` is a row vector of values.

Example: `[20:24]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`precision` — Size and format of each value
`'uint8'` | `'int8'` | `'uint16'` | `'int16'` | `'uint32'` | `'int32'` | `'uint64'` | `'int64'` | `'single'` | `'double'` | `'char'` | `'string'`

Size and format of each value, specified as a character vector or string. `precision` determines the number of bits to write for each value and its format as a MATLAB data type. For ASCII text, you can specify `precision` as either `'char'` or `'string'`.

Example: `'int16'`

Data Types: `char` | `string`

See Also

Functions
`read` | `serialport` | `writeline`

Introduced in R2019b

writeline

Write line of ASCII data to serial port

Syntax

```
writeline(device,data)
```

Description

`writeline(device,data)` writes the ASCII text `data`, followed by the terminator, to the specified serial port device. The function blocks MATLAB and waits until the data and terminator are written to the port.

Examples

Write ID Query to Serial Device

Write an instrument identification query string to a serial instrument. `writeline` automatically includes the defined terminator.

```
s = serialport("COM3",9600);
configureTerminator(s,"CR")
 ⋮
writeline(s,"*IDN?")
```

Input Arguments

`device` — Serial port
`serialport` object

Serial port, specified as a `serialport` object.

Example: `serialport()`

data — ASCII data
string | char

ASCII data to write to serial port, specified as a string or character vector of text.

Example: `"IDN?"`

Data Types: `char` | `string`

See Also

Functions
`configureTerminator` | `readline` | `serialport` | `write`

Introduced in R2019b

set

Set graphics object properties

Syntax

```
set(H,Name,Value)
set(H,NameArray,ValueArray)
set(H,S)
s = set(H)
values = set(H,Name)
```

Description

Note Do not use the `set` function on Java objects as it will cause a memory leak. For more information, see "Access Public and Private Data".

`set(H,Name,Value)` specifies a value for the property `Name` on the object identified by `H`. Use single quotes around the property name, for example, `set(H,'Color','red')`. If `H` is a vector of objects, then `set` sets the property for all the objects. If `H` is empty (that is, `[]`), `set` does nothing, but does not return an error or warning.

`set(H,NameArray,ValueArray)` specifies multiple property values using the cell arrays `NameArray` and `ValueArray`. To set `n` property values on each of `m` graphics objects, specify `ValueArray` as an `m`-by-`n` cell array, where `m = length(H)` and `n` is equal to the number of property names contained in `NameArray`.

`set(H,S)` specifies multiple property values using `S`, where `S` is a structure whose field names are the object property names and whose field values are the corresponding property values. MATLAB ignores empty structures.

`s = set(H)` returns the user-settable properties and possible values for the object identified by `H`. `s` is a structure whose field names are the object's property names and whose field values are the possible values of the corresponding properties. If you do not

specify an output argument, the MATLAB software displays the information on the screen. H must be a single object.

`values = set(H,Name)` returns the possible values for the specified property. If the possible values are character vectors, `set` returns each in a cell of the cell array `values`. For other properties, `set` returns a statement indicating that `Name` does not have a fixed set of property values. If you do not specify an output argument, MATLAB displays the information on the screen. H must be a single object.

Note For more information about properties you can set, see the property pages for each object, for example, Figure, Axes, Chart Line, Text, and so on.

Examples

Change Color of Specific Line

Plot a line and return the chart line object as `p`. Set the `Color` property of the line to `'red'`.

```
p = plot(1:10);
set(p,'Color','red')
```

Change Color for Multiple Lines

Create a plot with four lines using random data and return the four chart line objects as `P`. Set the `Color` property for all of the lines to `'red'`.

```
P = plot(rand(4));
set(P,'Color','red')
```

Set Line Style to Different Value for Multiple Lines

Set the value of the `LineStyle` property for four chart line objects each to a different value. Transpose the value of the cell array so that it has the proper shape.

```
P = plot(rand(4));
NameArray = {'LineStyle'};
ValueArray = {'-','--',':','-.'}';
set(P,NameArray,ValueArray)
```

Set Different Values for Multiple Properties on Multiple Objects

Set the values of the `Marker` and `Tag` properties on three different stem series objects to different values. Each row of the value cell array corresponds to an object in `h` and contains two values, one for the `Marker` property and one for the `Tag` property.

```
x = 0:30;
y = [1.5*cos(x); 4*exp(-.1*x).*cos(x); exp(.05*x).*cos(x)]';
S = stem(x,y);
NameArray = {'Marker','Tag'};
ValueArray = {'o','Decaying Exponential';...
   'square','Growing Exponential';...
   '*','Steady State'};
set(S,NameArray,ValueArray)
```

Tips

You can use any combination of property name/property value pairs, structure arrays, and cell arrays in one call to `set`.

Setting Property Units

Note that if you are setting both the `FontSize` and the `FontUnits` properties in one function call, you must set the `FontUnits` property first so that the MATLAB software can correctly interpret the specified `FontSize`. The same applies to figure and axes units — always set the `Units` property before setting properties whose values you want to be interpreted in those units. For example,

```
f = figure('Units','characters','Position',[30 30 120 35]);
```

See Also

`findobj` | `gca` | `gcbo` | `gcf` | `gco` | `get`

Topics

"Graphics Object Properties"

Introduced before R2006a

set

Set property values for `audioplayer` object

Syntax

```
set(obj,Name,Value)
set(obj,cellOfNames,cellOfValues)
set(obj,structOfProperties)
settableProperties = set(obj)
```

Description

`set(obj,Name,Value)` sets the named property to the specified value for the object *`obj`*.

`set(obj,cellOfNames,cellOfValues)` sets the properties listed in the cell array *`cellOfNames`* to the corresponding values in the cell array *`cellOfValues`*. Each cell array must contain the same number of elements.

`set(obj,structOfProperties)` sets the properties identified by each field of the structure array *`structOfProperties`* to the values of the associated fields.

`settableProperties` `= set(obj)` returns the names of the properties that you can set in a structure array. The field names of *`settableProperties`* are the property names.

Examples

View the list of properties that you can set for an `audioplayer` object:

```
load handel.mat;
handelObj = audioplayer(y, Fs);
set(handelObj)
```

Set the `Tag` and `UserData` properties of an `audioplayer` object using a structure array:

```
newValues.Tag = 'My Tag';
newValues.UserData = {'My User Data', pi, [1 2 3 4]};

load handel.mat;
handelObj = audioplayer(y, Fs);
set(handelObj, newValues)

% View the values all properties.
get(handelObj)
```

Tips

The `set` function allows combinations of property name/value pairs, cell array pairs, and structure arrays in the same function call.

Alternatives

To set the value of a single property, you can use dot notation. Reference each property as though it is a field of a structure array. For example, set the `Tag` property for an object called `handelObj` (as created in the Examples):

```
handelObj.Tag = 'This is my tag.';
```

This command is exactly equivalent to:

```
set(handelObj, 'Tag', 'This is my tag.');
```

See Also

`audioplayer` | `get`

set

Set property values for `audiorecorder` object

Syntax

```
set(obj,Name,Value)
set(obj,cellOfNames,cellOfValues)
set(obj,structOfProperties)
settableProperties = set(obj)
```

Description

`set(obj,Name,Value)` sets the named property to the specified value for the object *`obj`*.

`set(obj,cellOfNames,cellOfValues)` sets the properties listed in the cell array *`cellOfNames`* to the corresponding values in the cell array *`cellOfValues`*. Each cell array must contain the same number of elements.

`set(obj,structOfProperties)` sets the properties identified by each field of the structure array *`structOfProperties`* to the values of the associated fields.

`settableProperties` `= set(`*`obj`*`)` returns the names of the properties that you can set in a structure array. The field names of *`settableProperties`* are the property names.

Examples

View the list of properties that you can set for an `audiorecorder` object:

```
recorderObj = audiorecorder;
set(recorderObj)
```

Set the `Tag` and `UserData` properties of an `audiorecorder` object using a structure array:

```
newValues.Tag = 'My Tag';
newValues.UserData = {'My User Data', pi, [1 2 3 4]};

recorderObj = audiorecorder;
set(recorderObj, newValues)

% View the values all properties.
get(recorderObj)
```

Tips

The `set` function allows combinations of property name/value pairs, cell array pairs, and structure arrays in the same function call.

Alternatives

To set the value of a single property, you can use dot notation. Reference each property as though it is a field of a structure array. For example, set the `Tag` property for an object called `recorderObj` (as created in the Examples):

```
recorderObj.Tag = 'This is my tag.';
```

This command is exactly equivalent to:

```
set(recorderObj, 'Tag', 'This is my tag.');
```

See Also

`audiorecorder` | `get`

matlab.io.datastore.Shuffleable class

Package: `matlab.io.datastore`

Add shuffling support to datastore

Description

`matlab.io.datastore.Shuffleable` is an abstract mixin class that adds support for shuffling samples in a datastore in random order.

To use this mixin class, you must inherit from the `matlab.io.datastore.Shuffleable` class, in addition to inheriting from the `matlab.io.Datastore` base class. Type the following syntax as the first line of your class definition file:

```
classdef MyDatastore < matlab.io.Datastore & ...
                       matlab.io.datastore.Shuffleable
    ...
end
```

To add support for shuffling to your custom datastore, you must:

- Inherit from the additional class `matlab.io.datastore.Shuffleable`.
- Define this additional method: `shuffle`.

For more details and steps to create your custom datastore, see “Develop Custom Datastore”.

Methods

shuffle Return a shuffled version of a datastore

Attributes

`Abstract`	`true`

`Sealed`	`false`

For information on class attributes, see “Class Attributes”.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

See Also

`datastore` | `matlab.io.Datastore` | `matlab.io.datastore.HadoopFileBased` | `matlab.io.datastore.Partitionable`

Topics

“Develop Custom Datastore”

Introduced in R2018a

shuffle

Class: `matlab.io.datastore.Shuffleable`
Package: `matlab.io.datastore`

Return a shuffled version of a datastore

Syntax

```
ds2 = shuffle(ds)
```

Description

`ds2 = shuffle(ds)` shuffles samples of the datastore `ds` in random order, and returns the shuffled datastore, `ds2`.

Input Arguments

ds — Input datastore
`matlab.io.Datastore` object

Input datastore, specified as a `matlab.io.Datastore` object.

Output Arguments

ds2 — Shuffled datastore
`matlab.io.Datastore` object

Shuffled datastore, returned as a `matlab.io.Datastore` object.

Attributes

`Abstract`	`true`

To learn about attributes of methods, see Method Attributes.

Tips

- You must implement the `shuffle` method by deriving a subclass from the `matlab.io.datastore.Shuffleable` class. For more information, see “Develop Custom Datastore”.

See Also

`matlab.io.Datastore` | `matlab.io.datastore.Shuffleable`

Introduced in R2018a

matlab.mixin.SetGet class

Package: `matlab.mixin`

Provide handle classes with set and get methods

Description

Use the `matlab.mixin.SetGet` class to derive classes that inherit `set` and `get` methods that behave like the MATLAB graphics `set` and `get` functions.

The `set` and `get` methods inherited from `matlab.mixin.SetGet` support inexact property name matching. You can use a combination of partial and case-insensitive names as long as the inexact name is not ambiguous.

In contrast, dot-based property references require exact property name matches. To implement exact name matching with `set` and `get` methods, use the `matlab.mixin.SetGetExactNames` class.

The `matlab.mixin.SetGet` class is a `handle` class.

Class Attributes

`Abstract`	`true`
`HandleCompatible`	`true`

For information on class attributes, see “Class Attributes”.

Methods

Public Methods

set	Assign specified property-name/property-value pairs
get	Query specified property values
setdisp	Customize set method display
getdisp	Customize get method display

Examples

Subclass SetGet

Define a class that derives from `matlab.mixin.SetGet` and has two properties

```
classdef PropSetGet < matlab.mixin.SetGet
   properties
      Number {mustBeNumeric}
      Str string
   end
end
```

Create an instance of the class and use the inherited set method to set a property value.

```
p = PropSetGet;
set(p,'Number',8)
p.Number
```

```
ans =

     8
```

See Also

`matlab.mixin.SetGetExactNames`

Topics

"Implement Set/Get Interface for Properties"
Class Attributes

Introduced in R2014b

matlab.mixin.SetGetExactNames class

Package: `matlab.mixin`
Superclasses: `matlab.mixin.SetGet`

Require exact name match for set and get methods

Description

Use the `matlab.mixin.SetGetExactNames` class to derive classes that inherit methods named `set` and `get` that set and get property values. For information on how to use the `set` and `get` methods, see the MATLAB `set` and `get` functions.

Classes derived from `matlab.mixin.SetGetExactNames` require case-sensitive, exact property name matches. To support inexact name matches, derive from the `matlab.mixin.SetGet` class.

The `matlab.mixin.SetGetExactNames` class is a `handle` class.

Class Attributes

`Abstract`	`true`
`HandleCompatible`	`true`

For information on class attributes, see “Class Attributes”.

Methods

Public Methods

set	Assign specified property-name/property-value pairs
get	Query specified property values
setdisp	Customize set method display
getdisp	Customize get method display

Examples

Subclass SetGetExactNames

Define a class that has two properties.

```
classdef PropSetGet < matlab.mixin.SetGetExactNames
   properties
      Number {mustBeNumeric}
      Str string
   end
end
```

Create an instance of the class and try to use the inherited set method to set a property value. The method fails because the property name must be a case-sensitive match for the name defined in the class.

```
p = PropSetGet;
set(p,'number',8)
```

```
Error using PropSetGetExact/set
The name 'number' is not an accessible property for an instance of class 'PropSetGetExact'.
```

See Also

`matlab.mixin.SetGet`

Topics

"Implement Set/Get Interface for Properties"

Introduced in R2016b

set

Class: `matlab.mixin.SetGet`
Package: `matlab.mixin`

Assign specified property-name/property-value pairs

Syntax

```
set(H,Name,Value,...)
set(H,pn,pv)
set(H,s)
S = set(h)
```

Description

`set(H,Name,Value,...)` sets the named property to the specified value for the objects in the handle array `H`. `set` matches partial and case-insensitive names that are not ambiguous. Inexact name matching applies only to class properties. Dynamic properties require exact name matches.

`set(H,pn,pv)` sets the named properties specified in the cell array of strings `pn` to the corresponding values in the cell array `pv` for all objects specified in `H`. The cell array `pn` must be `1`-by-`n` (where `n` is the number of property names), but the cell array `pv` can be `m`-by-`n` where `m` is equal to `length(H)`. `set` updates each object with the associated set of values for the list of property names contained in.

`set(H,s)` sets the properties identified by each field name of `struct` `s` with the values contained in `s`. `s` is a `struct` whose field names are object property names.

`S = set(h)` returns the user-settable properties of scalar `h`. `S` is a `struct` whose field names are the object's property names and values that are either empty cell arrays or cell arrays of possible values for properties that have a finite set of predefined possible values.

Input Arguments

`H` — Input handle array
handle array

Input handle array, specified as a single handle or an array of handles

`Name` — Property name
character vector | string scalar

Property name, specified as a combination of partial and case-insensitive names. The inexact name must not be ambiguous. Dynamic properties must use exact names.

Data Types: `char` | `string`

`Value` — Property value to assign to the named property
property value

Property value to assign to the named property, specified as appropriate for that property.

`pn` — Property names
cell array of character vectors or strings

Property names, specified as a cell array of character vectors or scalar strings. The cell array `pn` must be `1`-by-`n` (where `n` is the number of property names).

Data Types: `cell`

`pv` — Property values
cell array

Property values, specified as a cell array. The cell array `pv` can be `m`-by-`n` where `m` is equal to `length(H)` and `n` is the number of property names in `pn`.

Data Types: `cell`

`s` — Property name and value structure
`struct`

Property name and value structure. The fields of `S` correspond to property names and the values of the fields are the property values to set.

Data Types: `struct`

h — Scalar object handle
handle

Scalar object handle. To obtain user-settable properties, input argument must be a scalar object handle.

Output Arguments

S — Settable properties
struct

Settable properties, returned as a structure with fields corresponding to property names and values that are either empty cell arrays or cell arrays of possible values for properties that have a finite set of predefined possible values.

Attributes

Access	public

To learn about attributes of methods, see Method Attributes.

Examples

Set the value of a property named `Data` on an instance of a class that derives from `matlab.mixin.SetGet`.

```
set(obj,'Data',[1 2 3 4])
```

Tips

- Override the `matlab.mixin.SetGet` class `setdisp` method to change how MATLAB displays information returned by `set`.

See Also

`matlab.mixin.SetGet` | `matlab.mixin.SetGet.get` | `matlab.mixin.SetGet.setdisp` | `set`

Topics

"Implement Set/Get Interface for Properties"

setdisp

Class: `matlab.mixin.SetGet`
Package: `matlab.mixin`

Customize `set` method display

Syntax

```
setdisp(h)
```

Description

`setdisp(h)` called by `set` when `set` is called with no output arguments and a single input argument that is a scalar handle. Override this `matlab.mixin.SetGet` class method in a subclass to change how property information is displayed in the command window.

Input Arguments

h — Object handle
handle

Object handle whose settable properties and their possible values are to be displayed.

Attributes

Access	public

To learn about attributes of methods, see Method Attributes.

See Also

`matlab.mixin.SetGet` | `matlab.mixin.SetGet.getdisp` | `matlab.mixin.SetGet.set` | `set`

Topics

"Implement Set/Get Interface for Properties"

get

Class: `matlab.mixin.SetGet`
Package: `matlab.mixin`

Query specified property values

Syntax

```
CV = get(H,Name)
SV = get(H)
get(H)
```

Description

`CV = get(H,Name)` returns the value of the named property from the objects in the handle array `H`. If `H` is scalar, `get` returns a single value; if `H` is an array, `get` returns a cell array of property values. `get` matches partial and case-insensitive names that are not ambiguous. Inexact name matching applies only to class properties. Dynamic properties require exact name matches.

If you specify a cell array of property names as the second argument, then `get` returns a cell array of values, where each row in the cell corresponds to an element in `H` and each column in the cell corresponds to an element in the property name cell array.

If `H` is nonscalar and `Name` is the name of a dynamic property, `get` returns a value only if the property exists in all objects referenced in `H`.

`SV = get(H)` returns a structure in which the field names are the object's property names and the values are the current values of the corresponding properties. If `H` is nonscalar, then `SV` is a `numel(H)`-by-`1` array of structures.

`get(H)` displays all property names and their current values for the MATLAB objects with handle `H`.

Your subclass can override the `matlab.mixin.SetGet` `getdisp` method to control how MATLAB displays this information.

Input Arguments

`H` — Input handle array
object array

Input handle array, specified as a single handle or an array of handles

`Name` — Property to query
character vector | string scalar

Property to query, specified as a combination of partial and case-insensitive names. The inexact name must not be ambiguous. Dynamic properties must use exact names.

Data Types: `char` | `string`

Output Arguments

`CV` — Value of queried property
property value

Value of queried property, returned as a single value or a cell array of values.

`SV` — Structure of property names and values
`struct`

Structure of property names and values, in which the field names are the object's property names and the values are the current values of the corresponding properties.

Data Types: `struct`

Attributes

`Access`	`public`

To learn about attributes of methods, see Method Attributes.

Examples

Get the value of a property named `Data` from an instance of a class derived from `matlab.mixin.SetGet`.

```
propValue = get(obj,'Data');
```

Tips

- Override the `matlab.mixin.SetGet` class `matlab.mixin.SetGet.getdisp` method to change how MATLAB displays information returned by `get`.

See Also

`get` | `matlab.mixin.SetGet` | `matlab.mixin.SetGet.getdisp`

Topics

"Implement Set/Get Interface for Properties"

getdisp

Class: `matlab.mixin.SetGet`
Package: `matlab.mixin`

Customize `get` method display

Syntax

```
getdisp(h)
```

Description

`getdisp(h)` called by `get` when `get` is called with no output arguments and a single input argument that is a scalar handle. Override this `matlab.mixin.SetGet` class method in a subclass to change how property information is displayed in the command window.

Input Arguments

h — Object handle
object

Object handle whose gettable properties and their current values are to be displayed.

Attributes

Access	public

To learn about attributes of methods, see Method Attributes.

See Also

`get` | `get(SetGet)`

Topics

"Implement Set/Get Interface for Properties"

set (RandStream)

Set random number stream property

Class

`RandStream`

`parallel.gpu.RandStream`

Syntax

```
set(stream,'PropertyName',Value)
set(stream,'Property1',Value1,'Property2',Value2,...)
set(stream,A)
A = set(stream,'Property')
set(stream,'Property')
A = set(stream)
set(stream)
```

Description

`set(stream,'PropertyName',Value)` sets the property `'PropertyName'` of the random stream `stream` to the value `Value`.

`set(stream,'Property1',Value1,'Property2',Value2,...)` sets multiple random stream property values with a single statement.

`set(stream,A)` where `A` is a structure whose field names are property names of the random stream `stream` sets the properties of `stream` named by each field with the values contained in those fields.

`A = set(stream,'Property')` or `set(stream,'Property')` displays possible values for the specified property of `stream`.

`A = set(stream)` or `set(stream)` displays or returns all writable properties of `stream` and their possible values.

See Also

`RandStream` | `get (RandStream)` | `parallel.gpu.RandStream` | `rand` | `randi` | `randn`

set

Set `tscollection` properties

Syntax

```
tscprop = set(tsc)
tscprop = set(tsc,propname)
set(tsc,propname,propval)
```

Description

`tscprop = set(tsc)` returns all property values of a `tscollection` object `tsc`.

`tscprop = set(tsc,propname)` displays the property specified in `propname`. This syntax is equivalent to the command `tscprop = tsc.propname`. For a full list of `tscollection` property names, see `tscollection`.

`set(tsc,propname,propval)` sets the property `propname` of `tsc` to the value `propval`. This syntax is equivalent to the command `tsc.propname = propval`. You can set multiple properties and their values at a time by using comma-separated pairs. For a full list of `tscollection` property names and corresponding valid values, see `tscollection`.

Examples

Name of `tscollection`

Create a `tscollection` object from two `timeseries` objects and set the `Name` property.

```
ts1 = timeseries(rand(5,1),'Name','ts1');
ts2 = timeseries(rand(5,1),'Name','ts2');
tsc = tscollection({ts1,ts2});
tscprop = set(tsc,'Name','newname')
```

```
Time Series Collection Object: newname

Time vector characteristics

      Start time            0 seconds
      End time              4 seconds

Member Time Series Objects:

      ts1
      ts2
```

Input Arguments

`tsc` — Input `tscollection`
scalar

Input `tscollection`, specified as a scalar.

`propname` — Property name
character vector

Property name, specified as a character vector. For a full list of `tscollection` property names, see `tscollection`.

Data Types: `char`

`propval` — Property value
scalar

Property value, specified as a scalar. For a full list of `tscollection` property names and valid values, see `tscollection`.

See Also

`get` | `timeseries` | `tscollection`

Introduced before R2006a

setabstime

Set `tscollection` times as date character vectors

Syntax

```
tscout = setabstime(tscin,timevals)
tscout = setabstime(tscin,timevals,timeformat)
```

Description

`tscout = setabstime(tscin,timevals)` sets the times in `tscollection` object `tscin` to the date character vectors specified in `timevals`.

`tscout = setabstime(tscin,timevals,timeformat)` explicitly specifies the format of `timeformat` used in `timevals`.

Examples

Absolute Time Vector

Create a `tscollection` object and define the absolute time vector.

```
tscin = tscollection(timeseries(rand(3,1)));
tscout = setabstime(tscin,{'12-DEC-2005 12:34:56','12-DEC-2005 13:34:56','12-DEC-2005 1
```

```
Time Series Collection Object: unnamed

Time vector characteristics

      Start date            12-Dec-2005 12:34:56
      End date              12-Dec-2005 14:34:56

Member Time Series Objects:

      unnamed
```

Input Arguments

`tscin` — Input `tscollection`

scalar

Input `tscollection`, specified as a scalar.

`timevals` — Sample times

array

Sample times, specified as a cell array of date character vectors. Valid date character vectors can have the following forms:

Format	Example
dd-mmm-yyyy HH:MM:SS	01-Mar-2000 15:45:17
dd-mmm-yyyy	01-Mar-2000
mm/dd/yy	03/01/00
mm/dd	03/01
HH:MM:SS	15:45:17
HH:MM:SS PM	3:45:17 PM
HH:MM	15:45
HH:MM PM	3:45 PM
mmm.dd,yyyy HH:MM:SS	Mar.01,2000 15:45:17
mmm.dd,yyyy	Mar.01,2000
mm/dd/yyyy	03/01/2000

Data Types: `cell`

`timeformat` — Time format

scalar

Time format, specified as a scalar date format used for the time values.

See Also

`getabstime` | `timeseries` | `tscollection`

Introduced before R2006a

setappdata

Store application-defined data

Syntax

```
setappdata(obj,name,val)
```

Description

Use this function to store data in a UI. You can retrieve the data elsewhere in your code using the `getappdata` function. Both of these functions provide a convenient way to share data between callbacks or between separate UIs.

`setappdata(obj,name,val)` stores the contents of `val`. The graphics object, `obj`, and the name identifier, `name`, uniquely identify the data for later retrieval.

Examples

Store and Retrieve Date Information

Create a figure window. Then, get the current time using the `date` function.

```
f = figure;
val = date
```

```
val =

23-Dec-2014
```

Store the contents of `val` using the `setappdata` function. In this case, `val` is stored in the figure object using the name identifier, `'todaysdate'`.

```
setappdata(f,'todaysdate',val);
```

Retrieve the data and display it.

```
getappdata(f,'todaysdate')
```

```
ans =

23-Dec-2014
```

Input Arguments

obj — Graphics object in which to store the value
figure | uipanel | uibuttongroup | uicontrol | ...

Graphics object in which to store the value, specified as any graphics object. The graphics object must be accessible from within the functions you plan to store and retrieve the data.

name — Name identifier
character vector

Name identifier, specified as a character vector. Select a unique name identifier that is easy to remember so that you can easily recall it when you want to retrieve the data.

Example: `setappdata(h,'mydata',5)` stores the value 5 using the name `'mydata'`.

Data Types: `char`

val — Value to store
any MATLAB data type

Value to store, specified as any MATLAB data type.

See Also

`getappdata` | `guidata` | `isappdata` | `rmappdata`

Topics

"Share Data Among Callbacks"

Introduced before R2006a

setcats

Set categories in categorical array

Syntax

```
B = setcats(A,newcats)
```

Description

`B = setcats(A,newcats)` sets categories in output categorical array `B` using categories defined by `newcats` and elements defined by `A`.

- If an element of `A` is in a category listed in `newcats`, then the corresponding element of `B` has the same value as in `A`.
- If `A` has a category not listed in `newcats`, then `B` does not have that category. The corresponding elements in `B` are undefined.
- If `newcats` lists a category that is not a category of `A`, then `B` has no elements equal to that category.

Examples

Set New Categories

Create a categorical array containing various colors.

```
A = categorical({'blue','black','red';'red','blue','black';'black','red','blue'})
```

```
A = 3x3 categorical array
     blue       black      red
     red        blue       black
     black      red        blue
```

Set new categories.

```
B = setcats(A,{'red','black'})
```

```
B = 3x3 categorical array
     <undefined>      black            red
     red              <undefined>      black
     black            red              <undefined>
```

Elements that were `blue` in `A` are undefined in `B`.

Set New Categories and Assign Elements

Create a categorical array containing various colors.

```
A = categorical({'blue','black','red';'red','blue','black';'black','red','blue'})
```

```
A = 3x3 categorical array
     blue       black      red
     red        blue       black
     black      red        blue
```

Set new categories. Include a category that is not a category of `A`.

```
B = setcats(A,{'red','pink','blue'})
```

```
B = 3x3 categorical array
     blue             <undefined>      red
     red              blue             <undefined>
     <undefined>      red              blue
```

No element of `B` is `pink`, because `pink` is not a category of `A`. Assign an element of `B` to be `pink`.

```
B(1,2) = 'pink';
B
```

```
B = 3x3 categorical array
     blue             pink      red
     red              blue      <undefined>
     <undefined>      red       blue
```

Input Arguments

`A` — Categorical array
vector | matrix | multidimensional array

Categorical array, specified as a vector, matrix, or multidimensional array.

`newcats` — New categories
character vector | cell array of character vectors | string array

New categories, specified as a character vector, a cell array of character vectors, or a string array.

Tips

- To change category names in a categorical array, use `renamecats`.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`addcats` | `categories` | `iscategory` | `mergecats` | `removecats` | `renamecats` | `reordercats`

Introduced in R2014b

setdiff

Set difference of two arrays

Syntax

```
C = setdiff(A,B)
C = setdiff(A,B,setOrder)
C = setdiff(A,B, ___ ,'rows')
C = setdiff(A,B,'rows', ___ )
[C,ia] = setdiff( ___ )

[C,ia] = setdiff(A,B,'legacy')
[C,ia] = setdiff(A,B,'rows','legacy')
```

Description

`C = setdiff(A,B)` returns the data in `A` that is not in `B`, with no repetitions. `C` is in sorted order.

- If `A` and `B` are tables or timetables, then `setdiff` returns the rows from `A` that are not in `B`. For timetables, `setdiff` takes row times into account to determine equality, and sorts the output timetable `C` by row times.

`C = setdiff(A,B,setOrder)` returns `C` in a specific order. `setOrder` can be `'sorted'` or `'stable'`.

`C = setdiff(A,B, ___ ,'rows')` and `C = setdiff(A,B,'rows', ___ )` treat each row of `A` and each row of `B` as single entities and return the rows from `A` that are not in `B`, with no repetitions. You must specify `A` and `B` and optionally can specify `setOrder`.

The `'rows'` option does not support cell arrays, unless one of the inputs is either a categorical array or a datetime array.

`[C,ia] = setdiff( ___ )` also returns the index vector `ia` using any of the previous syntaxes.

- Generally, `C = A(ia)`.
- If the `'rows'` option is specified, then `C = A(ia,:)`.
- If `A` and `B` are tables or timetables, then `C = A(ia,:)`.

`[C,ia] = setdiff(A,B,'legacy')` and `[C,ia] = setdiff(A,B,'rows','legacy')` preserve the behavior of the `setdiff` function from R2012b and prior releases.

The `'legacy'` option does not support categorical arrays, datetime arrays, duration arrays, tables, or timetables.

Examples

Difference of Two Vectors

Define two vectors with values in common.

```
A = [3 6 2 1 5 1 1];
B = [2 4 6];
```

Find the values in `A` that are not in `B`.

```
C = setdiff(A,B)
```

```
C = 1×3

     1     3     5
```

Difference of Two Tables

Define two tables with rows in common.

```
A = table([1:5]',['A';'B';'C';'D';'E'],logical([0;1;0;1;0]))
```

```
A=5×3 table
    Var1    Var2    Var3
    ____    ____    _____
```

```
     1       A      false
     2       B      true
     3       C      false
     4       D      true
     5       E      false
```

```
B = table([1:2:10]',['A';'C';'E';'G';'I'],logical(zeros(5,1)))
```

```
B=5×3 table
    Var1    Var2    Var3
    ____    ____    _____

     1       A      false
     3       C      false
     5       E      false
     7       G      false
     9       I      false
```

Find the rows in A that are not in B.

```
C = setdiff(A,B)
```

```
C=2×3 table
    Var1    Var2    Var3
    ____    ____    _____

     2       B      true
     4       D      true
```

Difference of Two Vectors and Indices to Different Values

Define two vectors with values in common.

```
A = [3 6 2 1 5 1 1];
B = [2 4 6];
```

Find the values in A that are not in B as well as the index vector `ia`, such that `C = A(ia)`.

```
[C,ia] = setdiff(A,B)
```

```
C = 1×3

     1     3     5


ia = 3×1

     4
     1
     5
```

Difference of Two Tables and Indices to Different Rows

Define a table, A, of gender, age, and height for five people.

```
A = table(['M';'M';'F';'M';'F'],[27;52;31;46;35],[74;68;64;61;64],...
'VariableNames',{'Gender' 'Age' 'Height'},...
'RowNames',{'Ted' 'Fred' 'Betty' 'Bob' 'Judy'})
```

```
A=5×3 table
             Gender    Age    Height
             ______    ___    ______

    Ted        M       27       74
    Fred       M       52       68
    Betty      F       31       64
    Bob        M       46       61
    Judy       F       35       64
```

Define a table, B, with the same variables as A.

```
B = table(['F';'M';'F';'F'],[64;68;62;58],[31;47;35;23],...
'VariableNames',{'Gender' 'Height' 'Age'},...
'RowNames',{'Meg' 'Joe' 'Beth' 'Amy'})
```

```
B=4×3 table
            Gender    Height    Age
            ______    ______    ___
```

```
    Meg       F         64      31
    Joe       M         68      47
    Beth      F         62      35
    Amy       F         58      23
```

Find the rows in `A` that are not in `B`, as well as the index vector `ia`, such that `C = A(ia,:)`.

```
[C,ia] = setdiff(A,B)
```

```
C=4×3 table
            Gender    Age    Height
            ______    ___    ______

    Judy      F       35       64
    Ted       M       27       74
    Bob       M       46       61
    Fred      M       52       68


ia = 4×1

     5
     1
     4
     2
```

The rows of `C` are in sorted order first by `Gender` and next by `Age`.

Difference of Rows in Two Matrices

Define two matrices with rows in common.

```
A = [7 9 7; 0 0 0; 7 9 7; 5 5 5; 1 4 5];
B = [0 0 0; 5 5 5];
```

Find the rows from `A` that are not in `B` as well as the index vector `ia`, such that `C = A(ia,:)`.

```
[C,ia] = setdiff(A,B,'rows')
```

```
C = 2×3

     1     4     5
     7     9     7


ia = 2×1

     5
     1
```

Difference of Two Vectors with Specified Output Order

Use the `setOrder` argument to specify the ordering of the values in `C`.

Specify `'stable'` or `'sorted'` when the order of the values in `C` are important.

```
A = [3 6 2 1 5 1 1];
B = [2 4 6];
[C,ia] = setdiff(A,B,'stable')
```

```
C = 1×3

     3     1     5


ia = 3×1

     1
     4
     5
```

Alternatively, you can specify `'sorted'` order.

```
 [C,ia] = setdiff(A,B,'sorted')
```

```
C = 1×3

     1     3     5
```

```
ia = 3×1

     4
     1
     5
```

Difference of Vectors Containing NaNs

Define two vectors containing `NaN`.

```
A = [5 NaN NaN];
B = [5 NaN];
```

Find the set difference of `A` and `B`.

```
C = setdiff(A,B)
```

```
C = 1×2

   NaN   NaN
```

`setdiff` treats `NaN` values as distinct.

Cell Array of Character Vectors with Trailing White Space

Create a cell array of character vectors, `A`.

```
A = {'dog','cat','fish','horse'};
```

Create a cell array of character vectors, `B`, where some of the vectors have trailing white space.

```
B = {'dog ','cat','fish ','horse'};
```

Find the character vectors in `A` that are not in `B`.

```
[C,ia] = setdiff(A,B)
```

```
C = 1x2 cell array
    {'dog'}    {'fish'}


ia = 2×1

     1
     3
```

`setdiff` treats trailing white space in cell arrays of character vectors as distinct characters.

Difference of Char and Cell Array of Character Vectors

Create a character vector, A.

```
A = ['cat';'dog';'fox';'pig'];
class(A)
```

```
ans =
'char'
```

Create a cell array of character vectors, B.

```
B={'dog','cat','fish','horse'};
class(B)
```

```
ans =
'cell'
```

Find the character vectors in A that are not in B.

```
C = setdiff(A,B)
```

```
C = 2x1 cell array
    {'fox'}
    {'pig'}
```

The result, C, is a cell array of character vectors.

```
class(C)
```

```
ans =
'cell'
```

Preserve Legacy Behavior of setdiff

Use the `'legacy'` flag to preserve the behavior of `setdiff` from R2012b and prior releases in your code.

Find the difference of A and B with the current behavior.

```
A = [3 6 2 1 5 1 1];
B = [2 4 6];
[C1,ia1] = setdiff(A,B)
```

```
C1 = 1×3

     1     3     5


ia1 = 3×1

     4
     1
     5
```

Find the difference of A and B, and preserve the legacy behavior.

```
[C2,ia2] = setdiff(A,B,'legacy')
```

```
C2 = 1×3

     1     3     5


ia2 = 1×3

     7     1     5
```

Input Arguments

A,B — Input arrays

numeric arrays | logical arrays | character arrays | string arrays | categorical arrays | datetime arrays | duration arrays | cell arrays of character vectors | tables | timetables

Input arrays, specified as numeric arrays, logical arrays, character arrays, string arrays, categorical arrays, datetime arrays, duration arrays, cell arrays of character vectors, tables, or timetables. If you specify the `'rows'` option, `A` and `B` must have the same number of columns.

`A` and `B` must belong to the same class with the following exceptions:

- `logical`, `char`, and all numeric classes can combine with `double` arrays.
- Cell arrays of character vectors can combine with character arrays or string arrays.
- Categorical arrays can combine with character arrays, cell arrays of character vectors, or string arrays.
- Datetime arrays can combine with cell arrays of date character vectors or single date character vectors.

There are additional requirements for `A` and `B` based on data type:

- If `A` and `B` are both ordinal categorical arrays, they must have the same sets of categories, including their order. If neither `A` nor `B` are ordinal, they need not have the same sets of categories, and the comparison is performed using the category names. In this case, the categories of `C` are the sorted union of the categories from `A` and `B`.
- If `A` and `B` are tables or timetables, they must have the same variable names (except for order). For tables, row names are ignored, so that two rows that have the same values, but different names, are considered equal. For timetables, row times are taken into account, so that two rows that have the same values, but different times, are not considered equal.
- If `A` and `B` are datetime arrays, they must be consistent with each other in whether they specify a time zone.

`A` and `B` also can be objects with the following class methods:

- `sort` (or `sortrows` for the `'rows'` option)
- `eq`

- `ne`

The object class methods must be consistent with each other. These objects include heterogeneous arrays derived from the same root class. For example, `A` and `B` can be arrays of handles to graphics objects.

`setOrder` — Order flag
`'sorted'` (default) | `'stable'`

Order flag, specified as `'sorted'` or `'stable'`, indicates the order of the values (or rows) in `C`.

Flag	Description

`'sorted'`	The values (or rows) in `C` return in sorted order as returned by `sort`.
	Example
	`C = setdiff([4 1 3 2 5],[2 1],'sorted')`
	`C =`
	`  3  4  5`

`'stable'`	The values (or rows) in `C` return in the same order as in `A`.
	Example
	`C = setdiff([4 1 3 2 5],[2 1],'stable')`
	`C =`
	`  4  3  5`

Data Types: `char` | `string`

Output Arguments

`C` — Difference of `A` and `B`
vector | matrix | table | timetable

Difference of A and B, returned as a vector, matrix, table, or timetable. If the inputs A and B are tables or timetables, then the order of the variables in C is the same as the order of the variables in A.

The following describes the shape of C when the inputs are vectors or matrices and when the `'legacy'` flag is not specified:

- If the `'rows'` flag is not specified and A is a row vector, then C is a row vector.
- If the `'rows'` flag is not specified and A is not a row vector, then C is a column vector.
- If the `'rows'` flag is specified, then C is a matrix containing the rows of A that are not in B.
- If all the values (or rows) of A are also in B, then C is an empty matrix.

The class of C is the same as the class of A, unless:

- A is a character array and B is a cell array of character vectors, in which case C is a cell array of character vectors.
- A is a character vector, cell array of character vectors, or string, and B is a categorical array, in which case C is a categorical array.
- A is a cell array of character vectors or single character vector and B is a datetime array, in which case C is a datetime array.
- A is a character vector or cell array of character vectors and B is a string array, in which case C is a string array.

`ia` — Index to A
column vector

Index to A, returned as a column vector when the `'legacy'` flag is not specified. `ia` identifies the values (or rows) in A that are not in B. If there is a repeated value (or row) appearing exclusively in A, then `ia` contains the index to the first occurrence of the value (or row).

Tips

- To find the set difference with respect to a subset of variables from a table or timetable, you can use column subscripting. For example, you can use `setdiff(A(:,vars),B(:,vars))`, where *`vars`* is a positive integer, a vector of positive integers, a variable name, a cell array of variable names, or a logical vector.

Alternatively, you can use `vartype` to create a subscript that selects variables of a specified type.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- The `'stable'` and `'legacy'` options are not supported.
- Inputs of type `'char'` are not supported.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support cell arrays for the first or second arguments.
- When you do not specify the `'rows'` option:
  - Inputs `A` and `B` must be vectors. If you specify the `'legacy'` option, then inputs `A` and `B` must be row vectors.
  - The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1.
  - Do not use `[]` to represent the empty set. Use a 1-by-0 or 0-by-1 input, for example, `zeros(1,0)`, to represent the empty set.
  - If you specify the `'legacy'` option, then empty outputs are row vectors, 1-by-0. They are never 0-by-0.
- When you specify both the `'legacy'` and `'rows'` options, the output `ia` is a column vector. If `ia` is empty, then it is 0-by-1. It is never 0-by-0, even if the output `C` is 0-by-0.

- When the `setOrder` is not `'stable'` or when you specify the `'legacy'` option, the inputs must already be sorted in ascending order. The first output, `C`, is sorted in ascending order.
- Complex inputs must be `single` or `double`.
- When one input is complex and the other input is real, do one of the following:
  - Set `setOrder` to `'stable'`.
  - Sort the real input in complex ascending order (by absolute value). Suppose the real input is `x`. Use `sort(complex(x))`or `sortrows(complex(x))`.
- See "Code Generation for Complex Data with Zero-Valued Imaginary Parts" (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `'legacy'` flag is not supported.
- 64-bit integers are not supported.

For more information, see "Run MATLAB Functions on a GPU" (Parallel Computing Toolbox).

See Also

`intersect` | `ismember` | `issorted` | `setxor` | `sort` | `union` | `unique`

Topics

"Combine Categorical Arrays"

Introduced before R2006a

setDirectory

Make specified IFD the current IFD

Syntax

```
setDirectory(t,dirNum)
```

Description

`setDirectory(t,dirNum)` sets the image file directory (IFD) specified by `dirNum` as the current IFD. `Tiff` object functions operate on the current IFD.

Examples

Navigate Through Multiple Images in TIFF File

Navigate the image file directories (IFDs) in a TIFF file and get tag or image data from the different IFDs.

Create a `Tiff` object for the file `example.tif` which contains three IFDs and display the directory index number for the current IFD in the file.

```
t = Tiff('example.tif','r');
currentDirectory(t)
```

```
ans = 1
```

Change the current IFD to 2 and get the value of the `ImageDescription` tag.

```
setDirectory(t,2);
getTag(t,'ImageDescription')
```

```
ans =
'First image converted to grayscale.'
```

Change the current IFD to 3 and get the value of the `ImageDescription` tag.

```
setDirectory(t,3);
getTag(t,'ImageDescription')
```

```
ans =
'Credit:  Jeff Mather and Alex Taylor.'
```

Close the `Tiff` object.

```
close(t);
```

Input Arguments

`t` — Tiff object
`Tiff` object

`Tiff` object representing a TIFF file. Use the `Tiff` function to create the object.

`dirNum` — Directory index number
positive integer

Directory index number, specified as a one-based positive integer.

Example: 2

Data Types: `double`

Algorithms

References

This function corresponds to the `TIFFSetDirectory` function in the LibTIFF C API. To use this function, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`currentDirectory` | `nextDirectory`

Topics

"Importing Images"
"Exporting to Images"

Introduced in R2009b

setenv

Set environment variable

Syntax

```
setenv(name,value)
setenv(name)
```

Description

`setenv(name,value)` sets the value of an operating system environment variable. If `name` exists as an environment variable, then `setenv` replaces its current value with `value`. If `name` does not exist, then `setenv` creates an environment variable called `name` and assigns `value` to it.

`setenv` passes `name` and `value` to the operating system unchanged. Special characters, such as `;`, `/`, `:`, `$`, and `%`, are unexpanded and intact in `value`.

A process spawned using the MATLAB `system`, `unix`, `dos`, or `!` function reads the values assigned to variables using the `setenv` function. You can retrieve any value set with `setenv` by calling `getenv(`*`name`*`)`.

`setenv(name)` assigns a null value to `name`. This syntax is equivalent to `setenv(name,'')`. On the Microsoft Windows platform, this syntax is equivalent to undefining the variable. On most UNIX platforms, it is possible to have an environment variable defined as empty.

Examples

Create Environment Variable TEMP

```
setenv('TEMP','C:\TEMP');
getenv('TEMP')
```

```
ans = 'C:\TEMP'
```

Append Folder to System Path

```
setenv('PATH', [getenv('PATH') ';D:\mypath']);
```

Input Arguments

name — Environment variable name
string | character vector

Environment variable name, specified as a string or a character vector.

The maximum number of characters in `name` is 2^{15} - 2 (or 32766). If `name` contains the = character, then `setenv` throws an error. The behavior of environment variables with = in the name is not well-defined.

Example: `'PATH'`

value — Environment variable value
string | character vector

Environment variable value, specified as a string or a character vector.

Example: `'C:\TEMP'`

See Also

`dos` | `getenv` | `system` | `unix`

Topics

"MATLAB Operators and Special Characters"

Introduced before R2006a

setfield

Assign value to structure array field

Syntax

```
S = setfield(S,field,value)
S = setfield(S,field1,...,fieldN,value)
S = setfield(S,idx,field1,...,fieldN,value)
S = setfield(S,idx,field1,idx1,...,fieldN,idxN,value)
```

Description

`S = setfield(S,field,value)` assigns a value to the specified field of the structure `S`. For example, `S = setfield(S,'a',1)` makes the assignment `S.a = 1`.

As an alternative to `setfield`, use dot notation: `S.field = value`. Dot notation is typically more efficient.

If `S` does not have the specified field, then `setfield` creates it and assigns `value` to it.

`S = setfield(S,field1,...,fieldN,value)` assigns a value to the specified field of a nested structure. For example, `S = setfield(S,'a','b','c',1)` makes the assignment `S.a.b.c = 1`, where the fields `S.a` and `S.a.b` are also structures.

`S = setfield(S,idx,field1,...,fieldN,value)` specifies an element of `S` and assigns a value to one of its fields. For example, `S = setfield(S,{3,4},'a',1)` makes the assignment `S(3,4).a = 1`.

`S = setfield(S,idx,field1,idx1,...,fieldN,idxN,value)` specifies elements of fields. For example, `S = setfield(S,'a',{2},1)` makes the assignment `S.a(2) = 1`. Similarly, `S = setfield(S,{3,4},'a',{2},'b',1)` makes the assignment `S(3,4).a(2).b = 1`.

Examples

Assign Values to Fields

Create a scalar structure.

```
S.x = linspace(0,2*pi);
S.y = sin(S.x);
S.title = ''
```

```
S = struct with fields:
        x: [1x100 double]
        y: [1x100 double]
    title: ''
```

Assign a value to a field using the `setfield` function.

```
S = setfield(S,'title','y = sin(x)')
```

```
S = struct with fields:
        x: [1x100 double]
        y: [1x100 double]
    title: 'y = sin(x)'
```

Assign a value to another field. If you specify a field that does not exist, then `setfield` creates it.

```
e = sqrt(abs(S.y));
S = setfield(S,'sqrty',e)
```

```
S = struct with fields:
        x: [1x100 double]
        y: [1x100 double]
    title: 'y = sin(x)'
    sqrty: [1x100 double]
```

You also can assign a value to a field using dot notation.

```
S.title = 'y = sin(x), with error bar values'
```

```
S = struct with fields:
        x: [1x100 double]
        y: [1x100 double]
    title: 'y = sin(x), with error bar values'
```

```
    sqrty: [1x100 double]
```

Field of Nested Structure

Create a nested structure. In a nested structure, a structure at any level can have fields that are structures, and other fields that are not structures.

```
S.a.b.c = 1;
S.a.b.d = 2;
S.a.b.e = struct('f',[3 4],'g',5);
S.h = 50
```

```
S = struct with fields:
    a: [1x1 struct]
    h: 50
```

While `S` is a structure, the fields `S.a`, `S.a.b`, and `S.a.b.e` are also structures.

```
S.a
```

```
ans = struct with fields:
    b: [1x1 struct]
```

```
S.a.b
```

```
ans = struct with fields:
    c: 1
    d: 2
    e: [1x1 struct]
```

```
S.a.b.e
```

```
ans = struct with fields:
    f: [3 4]
    g: 5
```

Assign a value to `S.a.b.d` using the `setfield` function. When you specify a comma-separated list of nested structure names, include the structure names at every level

between the top and the field name you specify. In this case, the comma-separated list of structure names is `'a','b'` and the field name is `'d'`.

```
S = setfield(S,'a','b','d',1024);
S.a.b
```

```
ans = struct with fields:
    c: 1
    d: 1024
    e: [1x1 struct]
```

You also can use dot notation to assign a value.

```
S.a.b.d = 2048;
S.a.b
```

```
ans = struct with fields:
    c: 1
    d: 2048
    e: [1x1 struct]
```

Fields of Elements of Structure Array

Assign values to fields of elements of a structure array.

First, create a structure array. As in all structure arrays, each element is a structure with the same fields.

```
S.x = linspace(0,2*pi);
S.y = sin(S.x);
S(2).x = S.x;
S(2).y = cos(S(2).x)
```

```
S=2×2 struct
    x
    y
```

You also can assign values using `setfield`. If a field does not exist, `setfield` creates it. Create a field named `title`.

```
S = setfield(S,{1},'title','y = sin(x)')
```

```
S=2×3 struct
    x
    y
    title
```

The `setfield` function assigns a value to a field of an individual element, but the output argument is the entire structure array.

Display the first element of `S`.

```
S(1)
```

```
ans = struct with fields:
        x: [1x100 double]
        y: [1x100 double]
    title: 'y = sin(x)'
```

As an alternative, index into the structure array, and then use dot notation to assign a value to a field of an element.

```
S(2).title = 'y = cos(x)';
S(2)
```

```
ans = struct with fields:
        x: [1x100 double]
        y: [1x100 double]
    title: 'y = cos(x)'
```

Indices of Nested Structure Array

Assign a value to a field of a nested structure, in which the structures at some levels are structure arrays. In this example, `S` is a 1-by-2 structure array. The second element, `S(2)`, has a nested structure `a.b`, where `b` is a 1-by-3 structure array.

First, create a nested structure. After creating the structure using dot notation, create another nonscalar structure array using the `struct` function and add it as a field.

```
S.a = 1;
S(2).a.b = struct('d',{5,10,20});
S
```

```
S=2×1 struct
    a
```

```
S(2).a.b
```

```
ans=3×1 struct
    d
```

Display the third element of `S(2).a.b`.

```
S(2).a.b(3)
```

```
ans = struct with fields:
    d: 20
```

Assign a new value to the field `d` of `S(2).a.b(3)` using the `setfield` function. Display the structure with the updated field.

```
S = setfield(S,{2},'a','b',{3},'d',3.1416);
S(2).a.b(3)
```

```
ans = struct with fields:
    d: 3.1416
```

Elements of Field

Create a structure with a field whose value is an array.

```
S.a = [5 10 15 20 25]
```

```
S = struct with fields:
    a: [5 10 15 20 25]
```

Assign values to elements of `S.a` using the `setfield` function. To assign values to particular elements, specify indices after the name of the field. You must specify the

indices within a cell array. However, specify the new values in an array whose data type matches the data type of the field.

```
S = setfield(S,'a',{3:5},[0 -50 -100])
```

```
S = struct with fields:
    a: [5 10 0 -50 -100]
```

You also can use dot notation and array indexing to assign values to the same elements.

```
S.a(3:5) = [20 40 80]
```

```
S = struct with fields:
    a: [5 10 20 40 80]
```

Input Arguments

S — Structure array
structure array

Structure array. If S is nonscalar, then each element of S is a structure, and all elements have the same fields with the same names.

field — Field name
character vector | string scalar

Field name, specified as a character vector or string scalar.

idx — Indices
cell array of numeric or logical values

Indices, specified as a cell array of numeric or logical values. Indices for S and fields 1 through N-1 specify individual elements of structure arrays. Indices for field N specify one or more elements of the array in that field, which can be of any type.

Example: `S = setfield(S,{1,2},'a',1)` is equivalent to `S(1,2).a = 1`.

Example: If `S.a = [5 10 20]`, then `S = setfield(S,'a',{[2,3]},[50 100])` is equivalent to `S.a(2:3) = [50 100]`.

`value` — Values
array

Values, specified as any type of array having any size.

See Also

`fieldnames` | `getfield` | `isfield` | `orderfields` | `rmfield` | `struct`

Topics

"Access Data in Structure Array"
"Generate Field Names from Variables"

Introduced before R2006a

setpixelposition

Set component position in pixels

Note

Use this function only with GUIDE, or with apps created using the `figure` function.

Syntax

```
setpixelposition(handle,position)
setpixelposition(handle,position,recursive)
```

Description

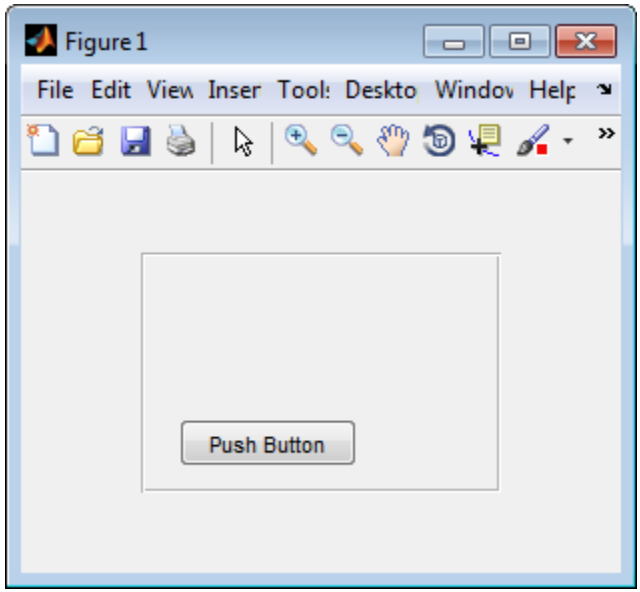
`setpixelposition(handle,position)` sets the position, in pixels on page 1-14084, of the component specified by `handle`. The `position` argument is a four-element vector that specifies the location and size of the component: [pixels from left, pixels from bottom, pixels across, pixels high]. The first two values in this vector are the number of pixels from the left and bottom edge of the parent container. The last two values specify the width and height of the component.

`setpixelposition(handle,position,recursive)` sets the position as above. If Boolean `recursive` is true, the position is set relative to the parent figure of `handle`.

Examples

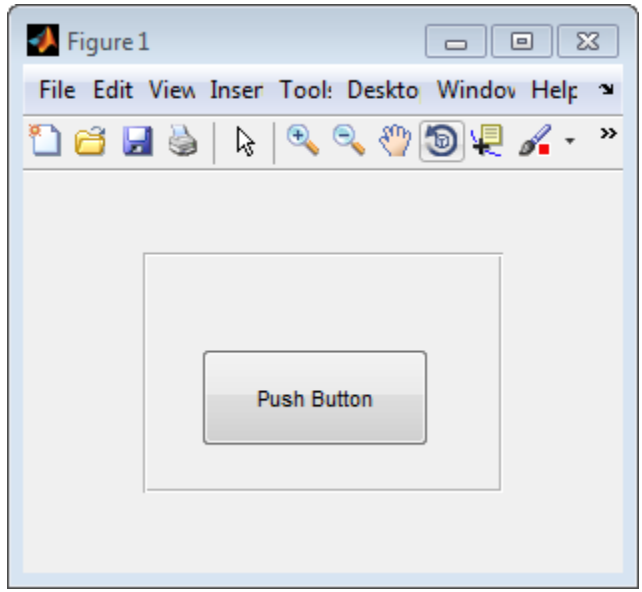
This example first creates a push button within a panel.

```
f = figure('Position',[300 300 300 200]);
p = uipanel('Position',[.2 .2 .6 .6]);
h1 = uicontrol(p,'Style','PushButton',...
               'Units','normalized',...
               'String','Push Button',...
               'Position',[.1 .1 .5 .2]);
```



The example then retrieves the position of the push button and changes its position with respect to the panel.

```
pos1 = getpixelposition(h1);
setpixelposition(h1,pos1 + [10 10 25 25]);
```



More About

Pixels

Distances in pixels are independent of your system resolution on Windows and Macintosh systems:

- On Windows systems, a pixel is 1/96th of an inch.
- On Macintosh systems, a pixel is 1/72nd of an inch.

On Linux systems, the size of a pixel is determined by your system resolution.

See Also

`figure` | `getpixelposition` | `uicontrol` | `uipanel`

setpref

Set custom preference value

Syntax

```
setpref(group,pref,value)
```

Description

`setpref(group,pref,value)` sets the specified preference in the specified group to the specified value. If the preference or group does not exist, MATLAB creates it.

Preferences are persistent and maintain their values between MATLAB sessions.

Examples

Modify Preference Value

Create a preference group and preference and then change the preference value.

Use `addpref` to create a preference group called `mytoolbox` and add a preference within it called `version`.

```
addpref('mytoolbox','version','1.0')
```

Change the value of the `version` preference and then get the preference value.

```
setpref('mytoolbox','version',{'1.0','beta'})
getpref('mytoolbox','version')
```

```
ans =
  1×2 cell array
    {'1.0'}    {'beta'}
```

Input Arguments

group — Custom preference group name
character vector | string scalar

Custom preference group name, specified as a character vector or a string scalar. `group` must be a valid variable name. For more information, see “Variable Names”.

Example: `'mytoolbox'`

Data Types: `char` | `string`

pref — Custom preference name
character vector | cell array of character vectors | string array

Custom preference name, specified as a character vector, a cell array of character vectors, or a string array. `pref` must be a valid variable name. For more information, see “Variable Names”.

If `pref` is a cell array of character vectors or a non-scalar string, `value` must specify a value for each preference specified in `pref`.

Example: `'version'`

Example: `{'version','modifieddate','docpath'}`

Data Types: `char` | `string`

value — Custom preference value
any MATLAB data type

Custom preference value, specified as any MATLAB data type, including numeric types, character vectors, cell arrays, structures, and objects. If `pref` is a cell array of character vectors or a nonscalar string array, `value` must specify a value for each preference specified in `pref`.

Example: `1.1`

Example: `{{1.1,'beta'},datetime(2018,1,9),'C:\mytoolbox\documentation'}`

See Also

`addpref` | `getpref` | `ispref` | `rmpref` | `uigetpref` | `uisetpref`

Topics

"Preferences"

Introduced before R2006a

setstr

(Not recommended) Convert numeric values or strings into character representation

Note `setstr` is not recommended. Use `char` instead.

Description

This MATLAB 4 function has been renamed `char` in MATLAB 5.

Introduced before R2006a

setSubDirectory

Make subIFD the current IFD

Syntax

```
setSubDirectory(t,offset)
```

Description

`setSubDirectory(t,offset)` sets the current IFD to the subimage file directory (subIFD) specified by `offset`. Use this function when you want to access subIFDs linked through the `SubIFD` tag.

Examples

Set Subimage File Directory

Open a TIFF file and read the value of the `SubIFD` tag in the current IFD. The `SubIFD` tag contains byte offsets that specify the location of subIFDs in the IFD.

```
t = Tiff('example.tif','r');
```

Read the value of the `SubIFD` tag to get the subdirectory offsets. Multiple values in the `offsets` array indicate multiple subdirectories.

```
offsets = getTag(t,'SubIFD')
```

```
offsets = uint64
    90114
```

Navigate to the first subIFD. If you have multiple subdirectories, then you can set any one of the subdirectories as the current directory.

```
setSubDirectory(t,offsets(1))
close(t);
```

Input Arguments

`t` — Tiff object
`Tiff` object

`Tiff` object representing a TIFF file. Use the `Tiff` function to create the object.

`offset` — Offset from current directory
positive integer

Offset from the current directory, specified as a positive integer. Specify offset value in bytes from the start of the file.

Get value of offset from the `SubIFD` tag.

Data Types: `double`

Algorithms

References

This function corresponds to the `TIFFSetSubDirectory` function in the LibTIFF C API. To use this function, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`setDirectory`

Topics

"Importing Images"
"Exporting to Images"

Introduced in R2009b

setTag

Set value of tag

Syntax

```
setTag(t,tagID,tagValue)
setTag(t,tagStruct)
```

Description

`setTag(t,tagID,tagValue)` sets the value of the TIFF tag specified by `tagID` to the value specified by `tagValue` in the TIFF file associated with the `tiff` object `t`.

`setTag(t,tagStruct)` sets all the tags specified in `tagStruct`. The `tagStruct` structure can contain multiple tag names and their corresponding values.

Examples

Set Tag Values

You can set tags for a TIFF file by specifying the tag name, the tag numeric identifier, or by specifying a structure of multiple tag names and values.

Create a new TIFF file, write image data, and then set tag values in different ways.

Read sample data into an array, `imdata`. Create a `Tiff` object associated with a new file, `myfile.tif`, and open the file for writing.

```
imdata = imread('example.tif');
t = Tiff('myfile.tif','w');
```

Set tag values by specifying the tag name.

```
setTag(t,'Photometric',Tiff.Photometric.RGB)
setTag(t,'PlanarConfiguration',Tiff.PlanarConfiguration.Chunky)
```

Set tag values by specifying the numeric tag identifier. Use the fields of the `Tiff.TagID` structure to obtain the tag IDs. For instance, the `Tiff.TagID.ImageLength` contains the numeric identifier for the `ImageLength` tag.

```
setTag(t,Tiff.TagID.ImageLength,size(imdata,1))
setTag(t,Tiff.TagID.ImageWidth,size(imdata,2))
```

Create a structure with fields named after TIFF tags and assign values to the fields. Pass this structure to the `setTag` method to set the values of these tags.

```
tagStruct.BitsPerSample = 8;
tagStruct.SamplesPerPixel = 3;
tagStruct.TileWidth = 128;
tagStruct.TileLength = 128;
tagStruct.Compression = Tiff.Compression.JPEG;
tagStruct.Software = 'MATLAB';
setTag(t,tagStruct)
```

Write the image data to the TIFF file and close the `Tiff` object.

```
write(t,imdata);
close(t);
```

Input Arguments

`t` — Tiff object
`Tiff` object

`Tiff` object representing a TIFF file. Use the `Tiff` function to create the object.

`tagID` — Tag ID
character vector | string scalar | numeric identifier

Tag ID of a `Tiff` object, specified as a character vector or string scalar, or a numeric identifier.

For example, you can specify `tagId` for the `ImageWidth` tag as any of these:

- Character vector or string scalar containing the tag name `'ImageWidth'`
- Numeric identifier `256` for the `ImageWidth` tag defined by the TIFF specification

- Field of the `Tiff.TagID` structure `Tiff.TagID.ImageWidth`.

The names of the fields of the `Tiff.TagID` structure are valid tag names that contain the corresponding tag numeric identifiers. For instance, the field `Tiff.TagID.ImageWidth` contains the value `256`. To see a list of all the tags along with their numeric identifiers, type `Tiff.TagID` in the command window.

Example: `'ImageWidth'`

Example: `256`

Example: `Tiff.TagID.ImageWidth`

Data Types: `double` | `char` | `string`

`tagValue` — Tag value
numeric identifier

Tag value, specified as a valid numeric identifier.

For example, you can specify `tagValue` for the `PlanarConfiguration` tag in one of these ways:

- Numeric identifier `2` that sets the value of the `PlanarConfiguration` tag indicating a separate configuration
- Field of the `Tiff.PlanarConfiguration` structure `Tiff.PlanarConfiguration.Separate`

The `Tiff.PlanarConfiguration` structure contains the numeric identifiers for all the valid values for the `PlanarConfiguration` tag. To see all the valid tag values for any given tag, type `Tiff.TagName` in the command window. For example, to see all the valid values for the `Photometric` tag, type `Tiff.Photometric` in the command window.

Data Types: `double`

`tagStruct` — Multiple tags
structure

Multiple tags, specified as a structure containing tag names and their corresponding values. The names of fields in `tagstruct` must be the name of supported TIFF tags, and their corresponding values must be valid TIFF tag values.

Data Types: `struct`

Tips

- If you are modifying a tag rather than creating it, then you must use the `rewriteDirectory` method after using the `setTag` method.

Algorithms

References

This function corresponds to the `TIFFSetField` function in the LibTIFF C API. To use this function, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff` | `Tiff.getTagNames` | `getTag`

Topics

"Importing Images"
"Exporting to Images"

Introduced in R2009b

settimeseriesnames

Rename `timeseries` in `tscollection`

Syntax

```
tscout = settimeseriesnames(tscin,oldname,newname)
```

Description

`tscout = settimeseriesnames(tscin,oldname,newname)` renames a `timeseries` object in a `tscollection` object from `oldname` to `newname`.

Examples

Rename timeseries

Create a `tscollection` object from a `timeseries` object. Then, change the `timeseries` name from `Speed` to `Velocity`.

```
ts = timeseries([3.2 4.2 6.2 8.5 1.1]',1:5,...
                 'Name','Speed');
tscin = tscollection(ts);
tscout = settimeseriesnames(tscin,'Speed','Velocity')
```

```
Time Series Collection Object: unnamed

Time vector characteristics

      Start time            1 seconds
      End time              5 seconds

Member Time Series Objects:

      Velocity
```

Input Arguments

`tscin` — Input `tscollection`
scalar

Input `tscollection`, specified as a scalar.

`oldname` — Existing `timeseries` name
character vector

Existing `timeseries` name, specified as a character vector.

`newname` — New `timeseries` name
character vector

New `timeseries` name, specified as a character vector.

See Also

`timeseries` | `tscollection`

Introduced before R2006a

Setting

Setting object

Description

A `Setting` object represents an individual setting within the settings hierarchical tree.

Creation

Access individual `Setting` objects using the root `SettingsGroup` object returned by the `settings` function. For example, this code accesses the `MaxWidth` setting.

```
s = settings
s.matlab.editor.language.matlab.comments.MaxWidth
```

Properties

`ActiveValue` — Current or active setting value
numeric array | logical array | character array

Current or active value, specified as a numeric array, logical array, or character array, depending on the setting.

The active value is determined as follows:

- If the setting has a temporary value, then the active value is the temporary value.
- If the setting has no temporary value, but it has a personal value, then the active value is the personal value.
- If the setting has no temporary value or personal value, then the active value is the factory value.

For example, suppose you have a setting `MySetting` with these values:

- Temporary value: 12

- Personal value: no value
- Factory value: 10

In this case, the active value for `MySetting` is the temporary value, `12`.

`TemporaryValue` — Temporary setting value
MATLAB data

Temporary setting value, specified as MATLAB data of any type except for handle types. Data containers such as cell arrays, structs, and objects that include handles are also not supported.

The temporary value is available only for the current MATLAB session and is cleared at the end of the session.

Some settings are linked to a preference. If a setting is linked to a preference, changing the temporary value for a setting temporarily changes the corresponding preference. At the end of the MATLAB session, the preference regains its original value. For more information about preferences, see “Preferences”.

`PersonalValue` — Personal setting value
MATLAB data

Personal setting value, specified as MATLAB data of any type except for handle types. Data containers such as cell arrays, structs, and objects that include handles are also not supported.

The personal value is available across MATLAB sessions for an individual user. When modified, the value is saved to the preferences folder.

Some settings are linked to a preference. If a setting is linked to a preference, changing the personal value for the setting changes the corresponding preference as well. For more information about preferences, see “Preferences”.

`FactoryValue` — Factory setting value
MATLAB data

This property is read-only.

Factory setting value, specified as MATLAB data of any type except for handle types. Data containers such as cell arrays, structs, and objects that include handles are also not supported.

The factory value is the default product setting. It is not modifiable. In addition, you cannot specify the factory value for a custom setting.

Object Functions

clearTemporaryValue	Clear the temporary value for a setting
clearPersonalValue	Clear the personal value for a setting
hasTemporaryValue	Determine whether the setting has a temporary value set
hasPersonalValue	Determine whether the setting has a personal value set
hasFactoryValue	Determine whether the setting has a factory value set

Examples

View All the Values of a Setting

View the current values for the maximum column width for comments in the Editor.

```
s = settings;
s.matlab.editor.language.matlab.comments.MaxWidth
```

```
ans = 
 Setting 'matlab.editor.language.matlab.comments.MaxWidth' with properties.

       ActiveValue: 80
    TemporaryValue: 80
     PersonalValue: <no value>
      FactoryValue: 75
```

Access and Modify a Setting

Get and set the value for the maximum column width for comments in MATLAB.

Get the root `SettingsGroup` object and view the active value for the maximum column width for comments in MATLAB.

```
s = settings
s.matlab.editor.language.matlab.comments.MaxWidth.ActiveValue
```

```
ans = 
  int32
   75
```

Set the temporary value for the maximum column width for comments in MATLAB.

```
s.matlab.editor.language.matlab.comments.MaxWidth.TemporaryValue = 80
s.matlab.editor.language.matlab.comments.MaxWidth.ActiveValue
```

```
ans =
  int32
   80
```

See Also

`SettingsGroup` | `settings`

Topics

"Access and Modify Settings"

Introduced in R2018a

settings

Access the `SettingsGroup` root object

Syntax

```
s = settings
```

Description

`s = settings` returns the root `SettingsGroup` object in the settings hierarchical tree. Use the returned object to access and modify settings.

For more information about settings, see “Access and Modify Settings”. For documentation on individual settings, go to “System Commands” and select a link in the **Settings** category.

Examples

Access and Modify a Setting

Get and set the value for the maximum column width for comments in MATLAB.

Get the root `SettingsGroup` object and access the value for the maximum column width for comments in MATLAB.

```
s = settings
s.matlab.editor.language.matlab.comments.MaxWidth
```

```
ans = 

 Setting 'matlab.editor.language.matlab.comments.MaxWidth' with properties.

       ActiveValue: 75
    TemporaryValue: <no value>
     PersonalValue: <no value>
      FactoryValue: 75
```

Set the temporary value for the maximum column width for comments in MATLAB. The temporary value is available only for the current MATLAB session and is cleared at the end of the session. For more information about the different types of settings values, see “Access and Modify Settings”.

```
s.matlab.editor.language.matlab.comments.MaxWidth.TemporaryValue = 80
s.matlab.editor.language.matlab.comments.MaxWidth
```

```
ans =

 Setting 'matlab.editor.language.matlab.comments.MaxWidth' with properties.

       ActiveValue: 80
    TemporaryValue: 80
     PersonalValue: <no value>
      FactoryValue: 75
```

See Also

`SettingsGroup`

Topics

“Access and Modify Settings”

Introduced in R2018a

SettingsFileUpgrader

Settings file upgrader object

Description

A `SettingsFileUpgrader` object represents the changes in the factory settings tree for a specific version of a toolbox. Create a `SettingsFileUpgrader` for each toolbox version that includes a change to the factory settings tree and record the changes to the tree. Recording changes ensures that toolbox users upgrading to a new version do not have any backward incompatibility issues with their toolbox settings.

Creation

Syntax

```
upgrader = SettingsFileUpgrader(version)
```

Description

`upgrader = SettingsFileUpgrader(version)` creates a `SettingsFileUpgrader` object for the specified toolbox version.

Input Arguments

`version` — Toolbox version
character vector | string

Toolbox version to create a `SettingsFileUpgrader` object for, specified as a character vector or string.

Example: "mytoolbox_Version1"

Properties

`Version` — Toolbox version
string

Toolbox version of the `SettingsFileUpgrader` object, specified as a string.

Object Functions

move	Record move or rename of factory setting or group
remove	Record removal of factory setting or group

Examples

Record Changes to Factory Settings Tree

Record changes to the factory settings tree that occurred in two different versions of a toolbox.

Create a settings file upgrader object for version 2 of `mytoolbox`.

```
upgraders = matlab.settings.SettingsFileUpgrader('Version2');
```

Record the move of the settings group font from the `mytoolbox.mysettings` factory settings group to the `mytoolbox` group.

```
upgraders.move('mytoolbox.mysettings.font','mytoolbox.font');
```

Create a settings file upgrader object for version 3 of `mytoolbox`.

```
upgraders(2) = matlab.settings.SettingsFileUpgrader('Version3');
```

Record the rename of the two font settings `FontSize` and `FontColor`, previously named `MyFontName` and `MyFontColor`.

```
upgraders(2).move('mytoolbox.font.MyFontSize','mytoolbox.font.FontSize');
upgraders(2).move('mytoolbox.font.MyFontColor','mytoolbox.font.FontColor');
```

See Also

`move` | `remove`

Topics

"Create Factory Settings for Toolboxes"
"Access and Modify Settings"

Introduced in R2019b

SettingsGroup

Group of settings and subgroup objects

Description

A `SettingsGroup` is an object within the settings hierarchical tree. At the top of the tree is the root `SettingsGroup`. Each `SettingsGroup` can contain a collection of other `SettingsGroup` and `Setting` objects.

Creation

Access the root `SettingsGroup` object using the `settings` function. For example:

```
s = settings
```

Object Functions

addSetting	Add new setting
addGroup	Add new settings group
removeGroup	Remove settings group
removeSetting	Remove setting
hasGroup	Determine if settings group exists
hasSetting	Determine if setting exists in settings group

Examples

Access and Modify a Setting

Get and set the value for the maximum column width for comments in MATLAB.

Get the root `SettingsGroup` object and view the active value for the maximum column width for comments in MATLAB.

```
s = settings
s.matlab.editor.language.matlab.comments.MaxWidth.ActiveValue
```

```
ans =
  int32
   75
```

Set the temporary value for the maximum column width for comments in MATLAB.

```
s.matlab.editor.language.matlab.comments.MaxWidth.TemporaryValue = 80
s.matlab.editor.language.matlab.comments.MaxWidth.ActiveValue
```

```
ans =
  int32
   80
```

See Also

`settings`

Topics

"Access and Modify Settings"
"Create Custom Settings"

Introduced in R2018a

setxor

Set exclusive OR of two arrays

Syntax

```
C = setxor(A,B)
C = setxor(A,B,setOrder)
C = setxor(A,B, ___ ,'rows')
C = setxor(A,B,'rows', ___ )
[C,ia,ib] = setxor( ___ )
```

```
[C,ia,ib] = setxor(A,B,'legacy')
[C,ia,ib] = setxor(A,B,'rows','legacy')
```

Description

`C = setxor(A,B)` returns the data of `A` and `B` that are not in their intersection (the symmetric difference), with no repetitions. That is, `setxor` returns the data that occurs in `A` or `B`, but not both. `C` is in sorted order.

- If `A` and `B` are tables or timetables, then `setxor` returns the rows that occur in one or the other of the two tables, but not both. For timetables, `setxor` takes row times into account to determine equality, and sorts the output timetable `C` by row times.

`C = setxor(A,B,setOrder)` returns `C` in a specific order. `setOrder` can be `'sorted'` or `'stable'`.

`C = setxor(A,B, ___ ,'rows')` and `C = setxor(A,B,'rows', ___ )` treat each row of `A` and each row of `B` as single entities and returns the rows of `A` and `B` that are not in their intersection, with no repetitions. You must specify `A` and `B` and optionally can specify `setOrder`.

The `'rows'` option does not support cell arrays, unless one of the inputs is either a categorical array or a datetime array.

`[C,ia,ib] = setxor( ___ )` also returns index vectors `ia` and `ib` using any of the previous syntaxes.

- Generally, the values in `C` are a sorted combination of the elements of `A(ia)` and `B(ib)`.
- If the `'rows'` option is specified, then `C` is a sorted combination of the rows of `A(ia,:)` and `B(ib,:)`.
- If `A` and `B` are tables or timetables, then `C` is a sorted combination of the rows of `A(ia,:)` and `B(ib,:)`.

`[C,ia,ib] = setxor(A,B,'legacy')` and `[C,ia,ib] = setxor(A,B,'rows','legacy')` preserve the behavior of the `setxor` function from R2012b and prior releases.

The `'legacy'` option does not support categorical arrays, datetime arrays, duration arrays, tables, or timetables.

Examples

Symmetric Difference of Two Vectors

Define two vectors with a value in common.

```
A = [5 1 3 3 3]; B = [4 1 2];
```

Find the values of `A` and `B` that are not in their intersection.

```
C = setxor(A,B)
```

```
C = 1×4

     2     3     4     5
```

Symmetric Difference of Two Tables

Define two tables with rows in common.

```
A = table([1:5]',['A';'B';'C';'D';'E'],logical([0;1;0;1;0]))
```

```
A=5×3 table
    Var1    Var2    Var3
```

```
    ____    ____    _____

     1       A      false
     2       B      true
     3       C      false
     4       D      true
     5       E      false
```

```
B = table([1:2:10]',['A';'C';'E';'G';'I'],logical(zeros(5,1)))
```

```
B=5×3 table
    Var1    Var2    Var3
    ____    ____    _____

     1       A      false
     3       C      false
     5       E      false
     7       G      false
     9       I      false
```

Find the rows of A and B that are not in their intersection.

```
C = setxor(A,B)
```

```
C=4×3 table
    Var1    Var2    Var3
    ____    ____    _____

     2       B      true
     4       D      true
     7       G      false
     9       I      false
```

Symmetric Difference of Two Vectors and Indices to Different Values

Define two vectors with a value in common.

```
A = [5 1 3 3 3]; B = [4 1 2];
```

Find the values of A and B that are not in their intersection as well as the index vectors ia and ib.

```
[C,ia,ib] = setxor(A,B)
```

```
C = 1×4

     2     3     4     5


ia = 2×1

     3
     1


ib = 2×1

     3
     1
```

C is a sorted combination of the elements A(ia) and B(ib).

Symmetric Difference of Two Tables and Indices to Different Rows

Define a table, A, of gender, age, and height for five people.

```
A = table(['M';'M';'F'],[27;52;31],[74;68;64],...
'VariableNames',{'Gender' 'Age' 'Height'},...
'RowNames',{'Ted' 'Fred' 'Betty'})
```

```
A=3×3 table
             Gender    Age    Height
             ______    ___    ______

    Ted        M       27       74
    Fred       M       52       68
    Betty      F       31       64
```

Define a table, B, with the same variables as A.

```
B = table(['F';'M'],[64;68],[31;47],...
'VariableNames',{'Gender' 'Height' 'Age'},...
'RowNames',{'Meg' 'Joe'})
```

```
B=2×3 table
           Gender    Height    Age
           ______    ______    ___

    Meg      F         64      31
    Joe      M         68      47
```

Find the rows of `A` and `B` that are not in their intersection, as well as the index vectors `ia` and `ib`.

```
[C,ia,ib] = setxor(A,B)
```

```
C=3×3 table
            Gender    Age    Height
            ______    ___    ______

    Ted       M       27       74
    Joe       M       47       68
    Fred      M       52       68


ia = 2×1

     1
     2


ib = 2
```

`C` is a sorted combination of the elements `A(ia,:)` and `B(ib,:)`.

Symmetric Difference of Rows in Two Matrices

Define two matrices with rows in common.

```
A = [7 8 9; 7 7 1; 7 7 1; 1 2 3; 4 5 6];
B = [1 2 3; 4 5 6; 7 7 2];
```

Find the rows of A and B that are not in their intersection as well as the index vectors ia and ib.

```
[C,ia,ib] = setxor(A,B,'rows')
```

```
C = 3×3

     7     7     1
     7     7     2
     7     8     9


ia = 2×1

     2
     1


ib = 3
```

C is a sorted combination of the rows of A(ia,:) and B(ib,:).

Symmetric Difference of Two Vectors in Specified Order

Use the setOrder argument to specify the ordering of the values in C.

Specify 'stable' if you want the values in C to have the same order as A and B.

```
A = [5 1 3 3 3]; B = [4 1 2];
[C,ia,ib] = setxor(A,B,'stable')
```

```
C = 1×4

     5     3     4     2


ia = 2×1

     1
     3


ib = 2×1
```

```
     1
     3
```

Alternatively, you can specify `'sorted'` order.

```
[C,ia,ib] = setxor(A,B,'sorted')
```

```
C = 1×4

     2     3     4     5


ia = 2×1

     3
     1


ib = 2×1

     3
     1
```

Symmetric Difference of Vectors Containing NaNs

Define two vectors containing `NaN`.

```
A = [5 NaN NaN]; B = [5 NaN NaN];
```

Find the symmetric difference of vectors `A` and `B`.

```
C = setxor(A,B)
```

```
C = 1×4

   NaN   NaN   NaN   NaN
```

The `setxor` function treats `NaN` values as distinct.

Cell Array of Character Vectors with Trailing White Space

Create a cell array of character vectors, A.

```
A = {'dog','cat','fish','horse'};
```

Create a cell array of character vectors, B, where some of the vectors have trailing white space.

```
B = {'dog ','cat','fish ','horse'};
```

Find the character vectors that are not in the intersection of A and B.

```
[C,ia,ib] = setxor(A,B)
```

```
C = 1x4 cell array
    {'dog'}    {'dog '}    {'fish'}    {'fish '}


ia = 2×1

     1
     3


ib = 2×1

     1
     3
```

`setxor` treats trailing white space in cell arrays of character vectors as distinct characters.

Symmetric Difference of Vectors of Different Classes and Shapes

Create a column vector character array.

```
A = ['A';'B';'C'], class(A)
```

```
A = 3x1 char array
    'A'
    'B'
```

```
    'C'


ans =
'char'
```

Create a row vector containing elements of numeric type `double`.

```
B = [66 67 68], class(B)
```

```
B = 1×3

    66    67    68


ans =
'double'
```

Find the symmetric difference of `A` and `B`.

```
C = setxor(A,B)
```

```
C = 2x1 char array
    'A'
    'D'
```

The result is a column vector character array.

```
class(C)
```

```
ans =
'char'
```

Symmetric Difference of Char and Cell Array of Character Vectors

Create a character vector, `A`.

```
A = ['cat';'dog';'fox';'pig'];
class(A)
```

```
ans =
'char'
```

Create a cell array of character vectors, B.

```
B={'dog','cat','fish','horse'};
class(B)
```

```
ans =
'cell'
```

Find the character vectors that are not in the intersection of A and B.

```
C = setxor(A,B)
```

```
C = 4x1 cell array
    {'fish' }
    {'fox'  }
    {'horse'}
    {'pig'  }
```

The result, C, is a cell array of character vectors.

```
class(C)
```

```
ans =
'cell'
```

Preserve Legacy Behavior of setxor

Use the `'legacy'` flag to preserve the behavior of `setxor` from R2012b and prior releases in your code.

Find the symmetric difference of A and B with the current behavior.

```
A = [5 1 3 3 3]; B = [4 1 2 2];
[C1,ia1,ib1] = setxor(A,B)
```

```
C1 = 1×4

     2     3     4     5


ia1 = 2×1
```

```
     3
     1


ib1 = 2×1

     3
     1
```

Find the symmetric difference and preserve the legacy behavior.

```
[C2,ia2,ib2] = setxor(A,B,'legacy')
```

```
C2 = 1×4

     2     3     4     5


ia2 = 1×2

     5     1


ib2 = 1×2

     4     1
```

Input Arguments

A,B — Input arrays

numeric arrays | logical arrays | character arrays | string arrays | categorical arrays | datetime arrays | duration arrays | cell arrays of character vectors | tables | timetables

Input arrays, specified as numeric arrays, logical arrays, character arrays, string arrays, categorical arrays, datetime arrays, duration arrays, cell arrays of character vectors, tables, or timetables. If you specify the `'rows'` option, `A` and `B` must have the same number of columns.

`A` and `B` must belong to the same class with the following exceptions:

- `logical`, `char`, and all numeric classes can combine with `double` arrays.
- Cell arrays of character vectors can combine with character arrays or string arrays.
- Categorical arrays can combine with character arrays, cell arrays of character vectors, or string arrays.
- Datetime arrays can combine with cell arrays of date character vectors or single date character vectors.

There are additional requirements for `A` and `B` based on data type:

- If `A` and `B` are both ordinal categorical arrays, they must have the same sets of categories, including their order. If neither `A` nor `B` are ordinal, they need not have the same sets of categories, and the comparison is performed using the category names. In this case, the categories of `C` are the sorted union of the categories from `A` and `B`.
- If `A` and `B` are tables or timetables, they must have the same variable names (except for order). For tables, row names are ignored, so that two rows that have the same values, but different names, are considered equal. For timetables, row times are taken into account, so that two rows that have the same values, but different times, are not considered equal.
- If `A` and `B` are datetime arrays, they must be consistent with each other in whether they specify a time zone.

`A` and `B` also can be objects with the following class methods:

- `sort` (or `sortrows` for the `'rows'` option)
- `eq`
- `ne`

The object class methods must be consistent with each other. These objects include heterogeneous arrays derived from the same root class. For example, `A` and `B` can be arrays of handles to graphics objects.

`setOrder` — Order flag
`'sorted'` (default) | `'stable'`

Order flag, specified as `'sorted'` or `'stable'`, indicates the order of the values (or rows) in `C`.

Flag	Description

`'sorted'`	The values (or rows) in `C` return in sorted order as returned by `sort`.
	
	Example
	
	`C = setxor([5 1 3],[4 1 2],'sorted')`
	
	`C =`
	
	`  2  3  4  5`

`'stable'`	The values (or rows) in `C` return in the same order as they appear in `A`, then `B`.
	
	Example
	
	`C = setxor([5 1 3],[4 1 2],'stable')`
	
	`C =`
	
	`  5  3  4  2`

Data Types: `char` | `string`

Output Arguments

`C` — Symmetric difference array
vector | matrix | table | timetable

Symmetric difference array, returned as a vector, matrix, table, or timetable. If the inputs `A` and `B` are tables or timetables, then the order of the variables in `C` is the same as the order of the variables in `A`.

The following describes the shape of `C` when the inputs are vectors or matrices and when the `'legacy'` flag is not specified:

- If the `'rows'` flag is not specified, then `C` is a column vector unless both `A` and `B` are row vectors, in which case `C` is a row vector. For example, `setxor([],[1 2])` returns a column vector.
- If the `'rows'` flag is specified, then `C` is a matrix containing the rows of `A` and `B` that are not in the intersection.

- If all the values (or rows) of `A` are also in `B`, then `C` is empty.

The class of the inputs `A` and `B` determines the class of `C`:

- If the class of `A` and `B` are the same, then `C` is the same class.
- If you combine a `char` or nondouble numeric class with `double`, then `C` is the same class as the nondouble input.
- If you combine a `logical` class with `double`, then `C` is `double`.
- If you combine a cell array of character vectors with `char`, then `C` is a cell array of character vectors.
- If you combine a categorical array with a character vector, cell array of character vectors, or string, then `C` is a categorical array.
- If you combine a datetime array with a cell array of date character vectors or single date character vector, then `C` is a datetime array.
- If you combine a string array with a character vector or cell array of character vectors, then `C` is a string array.

`ia` — Index to `A`
column vector

Index to `A`, returned as a column vector when the `'legacy'` flag is not specified. `ia` identifies the values (or rows) in `A` that contribute to the symmetric difference. If there is a repeated value (or row) appearing exclusively in `A`, then `ia` contains the index to the first occurrence of the value (or row).

`ib` — Index to `B`
column vector

Index to `B`, returned as a column vector when the `'legacy'` flag is not specified. `ib` identifies the values (or rows) in `B` that contribute to the symmetric difference. If there is a repeated value (or row) appearing exclusively in `B`, then `ib` contains the index to the first occurrence of the value (or row).

Tips

- To find the symmetric difference with respect to a subset of variables from a table or timetable, you can use column subscripting. For example, you can use `setxor(A(:,vars),B(:,vars))`, where *`vars`* is a positive integer, a vector of

positive integers, a variable name, a cell array of variable names, or a logical vector. Alternatively, you can use `vartype` to create a subscript that selects variables of a specified type.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- The `'stable'` and `'legacy'` options are not supported.
- Inputs of type `'char'` are not supported.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support cell arrays for the first or second arguments.
- When you do not specify the `'rows'` option:
  - Inputs `A` and `B` must be vectors with the same orientation. If you specify the `'legacy'` option, then inputs `A` and `B` must be row vectors.
  - The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1.
  - The input `[]` is not supported. Use a 1-by-0 or 0-by-1 input, for example , `zeros(1,0)`, to represent the empty set.
  - If you specify the `'legacy'` option, then empty outputs are row vectors, 1-by-0, never 0-by-0.
- When you specify both the `'legacy'` option and the `'rows'` option, the outputs `ia` and `ib` are column vectors. If these outputs are empty, then they are 0-by-1. They are never 0-by-0, even if the output `C` is 0-by-0.

- When the `setOrder` is not `'stable'` or when you specify the `'legacy'` flag, the inputs must already be sorted in ascending order. The first output, `C`, is sorted in ascending order.
- Complex inputs must be `single` or `double`.
- When one input is complex and the other input is real, do one of the following:
  - Set `setOrder` to `'stable'`.
  - Sort the real input in complex ascending order (by absolute value). Suppose the real input is `x`. Use `sort(complex(x))`or `sortrows(complex(x))`.
- See “Code Generation for Complex Data with Zero-Valued Imaginary Parts” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `'legacy'` flag is not supported.
- 64-bit integers are not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`intersect` | `ismember` | `issorted` | `setdiff` | `sort` | `union` | `unique`

Topics

“Combine Categorical Arrays”

Introduced before R2006a

sgtitle

Add title to subplot grid

Syntax

```
sgtitle(txt)
sgtitle(target,txt)
sgtitle( ___ ,Name,Value)
sgt = sgtitle( ___ )
```

Description

`sgtitle(txt)` adds a title above the grid of subplots in the current figure. If a figure does not it exist, then this command creates one.

`sgtitle(target,txt)` adds the title to the subplot grid in the specified figure, panel, or tab, instead of the current figure.

`sgtitle( ___ ,Name,Value)` modifies text properties using one or more name-value pair arguments. For example, `'FontSize',12` specifies 12-point font. For a list of properties, see Text. Specify name-value pair arguments after all the other input arguments.

`sgt = sgtitle( ___ )` returns the subplot `Text` object used to create the title.

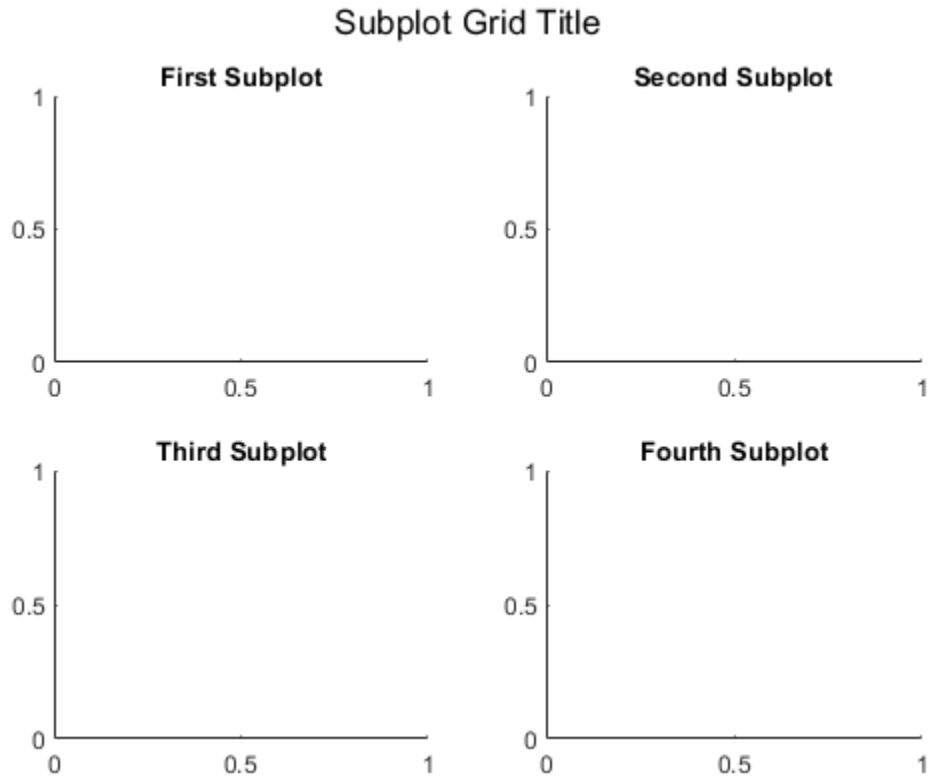
Examples

Add Title to Subplot Grid

Create a figure with four subplots. Add a title to each subplot, and then add an overall title to the subplot grid.

```
subplot(2,2,1)
title('First Subplot')
```

```
subplot(2,2,2)
title('Second Subplot')
subplot(2,2,3)
title('Third Subplot')
subplot(2,2,4)
title('Fourth Subplot')

sgtitle('Subplot Grid Title')
```

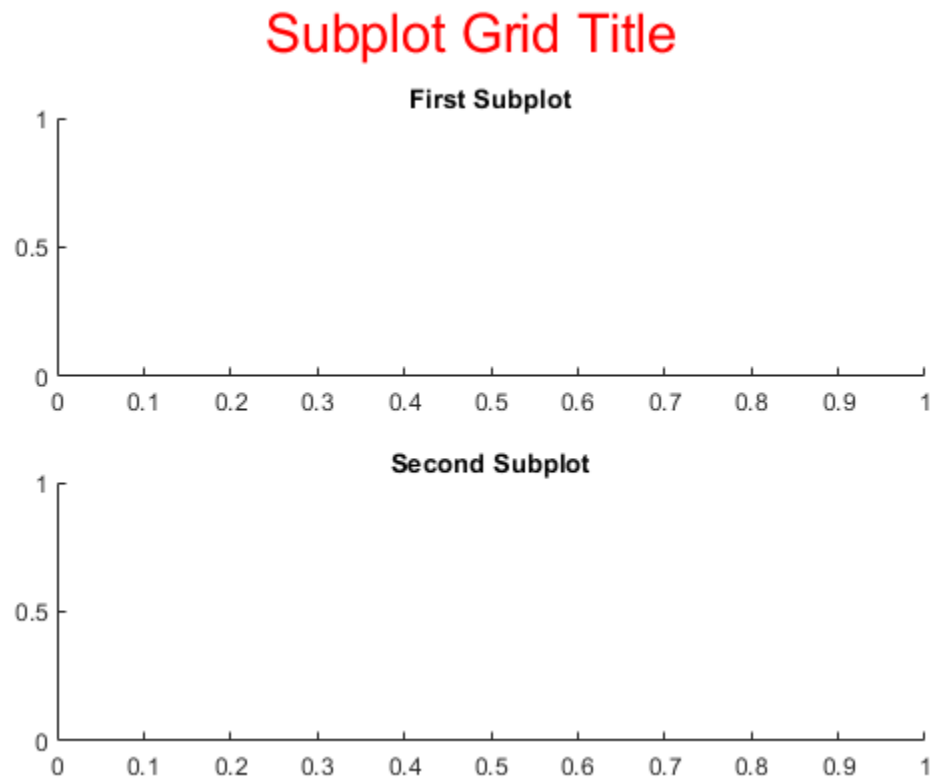


Modify Title Appearance

Modify a title appearance by setting properties, first by using name-value pair arguments, and then by returning the `Text` object created and using dot notation.

Change the `Color` property using a name-value pair argument. Next, change the `FontSize` property using dot notation.

```
subplot(2,1,1)
title('First Subplot')
subplot(2,1,2)
title('Second Subplot')

sgt = sgtitle('Subplot Grid Title','Color','red');
sgt.FontSize = 20;
```



Input Arguments

`txt` — Text to display

character vector | cell array of character vectors | string array | numeric value

Text to display, specified as a character vector, cell array of character vectors, string array, or numeric value.

Example: `'my label'`

Example: `{'first line','second line'}`

Example: `string('my label')`

Example: 123

To include numeric variables with text, use the `num2str` function. For example:

```
x = 42;
txt = ['The value is ',num2str(x)];
```

To include special characters, such as superscripts, subscripts, Greek letters, or mathematical symbols use TeX markup. For a list of supported markup, see the `Interpreter` property of the `Text` object.

To create multiline text, use these conventions:

- Use a cell array, where each cell contains a line of text, such as `{'first line','second line'}`.
- Use a string array, where each element contains a line of text, such as `string({'line one','line two'})`.
- Use a character array, where each row contains the same number of characters, such as `['abc'; 'ab ']`.
- Use `sprintf` to create text with a new line character, such as `sprintf('first line \n second line')`. This property converts text with new line characters to cell arrays.

If the text contains only a numeric value, then it is converted using `sprintf('%g',value)`. For example, `12345678` displays as `1.23457e+07`.

`target` — Subplot grid container
`Figure` object | `Panel` object | `Tab` object

Subplot grid container, specified as a `Figure`, `Panel`, or `Tab` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `sgtitle('My Title','FontSize',14)`

Note The properties listed here are only a subset. For a full list, see Text.

`FontSize` — Font size
scalar value greater than zero

Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals `1/72` inch. To change the font units, use the `FontUnits` property.

Example: 12

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`FontName` — Font name
supported font name | `'FixedWidth'`

Font name, specified as a supported font name or `'FixedWidth'`. To display and print text properly, you must choose a font that your system supports. The default font depends on your operating system and locale.

To use a fixed-width font that looks good in any locale, use `'FixedWidth'`. The fixed-width font relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

See Also

Functions
`subplot` | `title` | `xlabel` | `ylabel`

Properties
Text

Introduced in R2018b

shading

Set color shading properties

Syntax

```
shading flat
shading faceted
shading interp
shading(axes_handle,...)
```

Description

The `shading` function controls the color shading of surface and patch graphics objects.

`shading flat` each mesh line segment and face has a constant color determined by the color value at the endpoint of the segment or the corner of the face that has the smallest index or indices.

`shading faceted` flat shading with superimposed black mesh lines. This is the default shading mode.

`shading interp` varies the color in each line segment and face by interpolating the colormap index or true color value across the line or face.

`shading(axes_handle,...)` applies the shading type to the objects in the axes specified by `axes_handle`, instead of the current axes. Use single quotes when using a function form. For example:

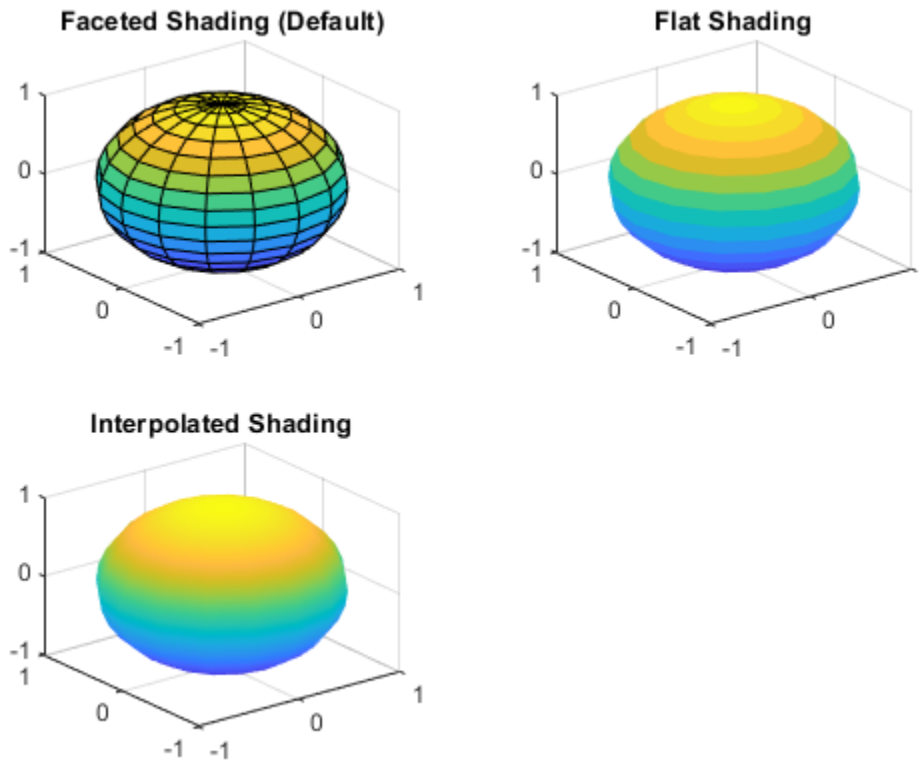
```
shading(gca,'interp')
```

Examples

Display Sphere with Different Types of Shading

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-2 tiled chart layout. Call the `nexttile` function to create the axes. Then display three different spheres using different types of shading.

```
tiledlayout(2,2)
nexttile
sphere(16)
title('Faceted Shading (Default)')

nexttile
sphere(16)
shading flat
title('Flat Shading')

nexttile
sphere(16)
shading interp
title('Interpolated Shading')
```



Algorithms

`shading` sets the `EdgeColor` and `FaceColor` properties of all surface and patch graphics objects in the current axes. `shading` sets the appropriate values, depending on whether the surface or patch objects represent meshes or solid surfaces.

See Also

`fill` | `fill3` | `hidden` | `light` | `lighting` | `mesh` | `patch` | `pcolor` | `surf`

Introduced before R2006a

sheetnames

Get sheet names from spreadsheet file

Syntax

```
sheets = sheetnames(filename)
```

Description

`sheets = sheetnames(filename)` returns a string array of sheet names from the specified spreadsheet file.

Examples

Query Sheet Names from Spreadsheet File

Get the name of all worksheets from the spreadsheet file `airlinesmall_subset.xlsx`.

```
sheets = sheetnames('airlinesmall_subset.xlsx')
```

```
sheets = 13x1 string array
    "1996"
    "1997"
    "1998"
    "1999"
    "2000"
    "2001"
    "2002"
    "2003"
    "2004"
    "2005"
    "2006"
    "2007"
    "2008"
```

Input Arguments

`filename` — Name of file to read

character vector | string scalar

Name of the file to read, specified as a character vector or a string scalar.

Depending on the location of your file, `filename` can take one of these forms.

<table>
<tr><th>Location</th><th>Form</th></tr>
<tr><td>Current folder or folder on the MATLAB path</td><td>Name of the file.

Example: <code>'myFile.xls'</code></td></tr>
<tr><td>Other local folder</td><td>Full or relative path to the file.

Example: <code>'C:\myFolder\myFile.xlsx'</code>

Example: <code>'myDir\myFile.xlsx'</code></td></tr>
<tr><td>Remote Location</td><td>Full path to the file, specified as a uniform resource locator (URL) of the form:

scheme_name://path_to_file/my_file.ext

Based on your remote location, scheme_name can be one of the values in this table.
<table>
<tr><th>Remote Location</th><th>scheme_name</th></tr>
<tr><td>Amazon S3</td><td>s3</td></tr>
<tr><td>Windows Azure Blob Storage</td><td>wasb, wasbs</td></tr>
<tr><td>HDFS</td><td>hdfs</td></tr>
</table>
For more information, see “Work with Remote Data”.

Example: <code>'s3://bucketname/path_to_file/my_file.xlsx'</code></td></tr>
</table>

- On Windows systems with Microsoft Excel software, the `sheetnames` function supports: `.xls, .xlsm, .xlsx, .xltm, .xltx, .xlsb, and .ods`.

- If your system does not have Excel for Windows, then `sheetnames` supports only: `.xls, .xlsx, .xlsm, .xltx, and .xltm` files.

Data Types: `char` | `string`

See Also

`readcell` | `readmatrix` | `readtable` | `readtimetable` | `readvars`

Topics

"Read Spreadsheet Data into Table"
"Read Collection or Sequence of Spreadsheet Files"
"Work with Remote Data"

Introduced in R2019b

shg

Show most recent graph window

Syntax

```
shg
```

Description

`shg` makes the current figure visible and raises it above all other figures on the screen. This is identical to using the command `figure(gcf)`.

See Also

`figure` | `gca` | `gcf`

shiftdim

Shift array dimensions

Syntax

```
B = shiftdim(A,n)
B = shiftdim(A)
[B,m] = shiftdim(A)
```

Description

`B = shiftdim(A,n)` shifts the dimensions of an array `A` by `n` positions. `shiftdim` shifts the dimensions to the left when `n` is a positive integer and to the right when `n` is a negative integer. For example, if `A` is a 2-by-3-by-4 array, then `shiftdim(A,2)` returns a 4-by-2-by-3 array.

`B = shiftdim(A)` returns an array with the same elements as `A` but with leading dimensions of length 1 removed.

`[B,m] = shiftdim(A)` also returns the number of dimensions of length 1 that were removed.

Examples

4-D Array

Create a 4-by-2-by-3-by-5 array. Shift the dimensions 2 positions to the left, wrapping the first 2 lengths to the last 2 dimensions.

```
A = rand(4,2,3,5);
B = shiftdim(A,2);
size(B)
```

```
ans = 1×4
```

```
     3     5     4     2
```

Shift the dimensions 2 positions to the right, resulting in 2 leading dimensions of length 1. Shifting to the right does not wrap the dimension lengths.

```
C = shiftdim(A,-2);
size(C)
```

```
ans = 1×6

     1     1     4     2     3     5
```

Shift Array Dimensions

Shift the dimensions of an array.

Compute a 5-D array `A`, and remove the leading dimensions of length 1. The `shiftdim` function shifts 2 dimensions and returns the 3-D array `B`.

```
A = rand(1,1,3,2,4);
[B,nshifts] = shiftdim(A);
nshifts
```

```
nshifts = 2
```

```
size(B)
```

```
ans = 1×3

     3     2     4
```

Shift the dimensions of `B` twice to the left.

```
C = shiftdim(B,2);
size(C)
```

```
ans = 1×3

     4     3     2
```

Shift the dimensions of `C` once to the right.

```
D = shiftdim(C,-1);
size(D)
```

```
ans = 1×4

     1     4     3     2
```

Input Arguments

`A` — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

`n` — Number of positions
integer

Number of dimension positions to shift, specified as an integer. When `n` is positive, `shiftdim` shifts the dimensions to the left, wrapping the leading dimensions to the end. When `n` is negative, `shiftdim` shifts the dimensions to the right, padding additional leading dimensions with length 1.

Output Arguments

`B` — Output array
vector | matrix | multidimensional array

Output array, specified as a vector, matrix, or multidimensional array.

`m` — Number of dimensions removed
non-negative integer

Number of dimensions removed, specified as a non-negative integer. `shiftdim` removes only leading dimensions of length 1.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Does not support cell arrays for the first argument.
- Second argument must be a constant.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`circshift` | `permute` | `reshape` | `squeeze`

Introduced before R2006a

showplottool

Show or hide figure plot tool

Syntax

```
showplottool('tool')
showplottool('on','tool')
showplottool('off','tool')
showplottool('toggle','tool')
showplottool(figure_handle,...)
```

Description

`showplottool('tool')` shows the specified plot tool on the current figure. `'tool'` can be one of these values:

- `'figurepalette'`
- `'plotbrowser'`
- `'propertyeditor'`

`showplottool('on','tool')` shows the specified plot tool on the current figure.

`showplottool('off','tool')` hides the specified plot tool on the current figure.

`showplottool('toggle','tool')` toggles the visibility of the specified plot tool on the current figure.

`showplottool(figure_handle,...)` operates on the specified figure instead of the current figure.

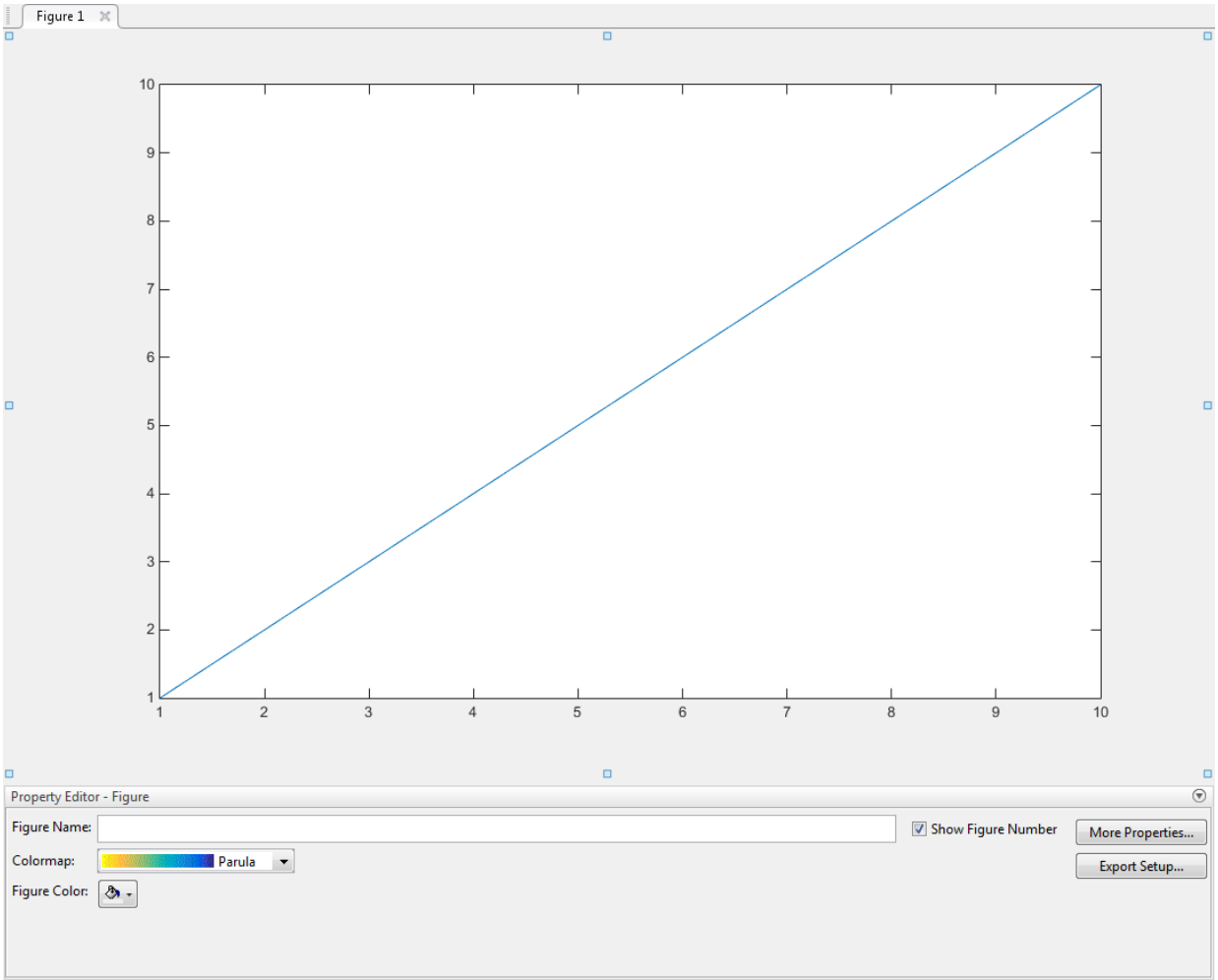
Note When you dock, undock, resize, or reposition a plotting tool and then close it, it will still be configured as you left it the next time you open it. There is no command to reset plotting tools to their original, default locations.

Examples

Open One of the Plot Tools

Create a simple plot and open the property editor.

```
plot(1:10);
showplottool('propertyeditor')
```



Alternatives

Select the **Figure Palette**, **Plot Browser**, and **Property Editor** tools from the figure's **View** menu.

See Also

`figurepalette` | `plotbrowser` | `plottools` | `propertyeditor`

Topics

"Interactively Explore Plotted Data"

Introduced before R2006a

shrinkfaces

Reduce size of patch faces

Syntax

```
shrinkfaces(p,sf)
nfv = shrinkfaces(p,sf)
nfv = shrinkfaces(fv,sf)
shrinkfaces(p)
nfv = shrinkfaces(f,v,sf)
[nf,nv] = shrinkfaces(...)
```

Description

`shrinkfaces(p,sf)` shrinks the area of the faces in patch `p` to shrink factor `sf`. A shrink factor of 0.6 shrinks each face to 60% of its original area. If the patch contains shared vertices, the MATLAB software creates nonshared vertices before performing the face-area reduction.

`nfv = shrinkfaces(p,sf)` returns the face and vertex data in the struct `nfv`, but does not set the `Faces` and `Vertices` properties of patch `p`.

`nfv = shrinkfaces(fv,sf)` uses the face and vertex data from the struct `fv`.

`shrinkfaces(p)` and `shrinkfaces(fv)` (without specifying a shrink factor) assume a shrink factor of 0.3.

`nfv = shrinkfaces(f,v,sf)` uses the face and vertex data from the arrays `f` and `v`.

`[nf,nv] = shrinkfaces(...)` returns the face and vertex data in two separate arrays instead of a struct.

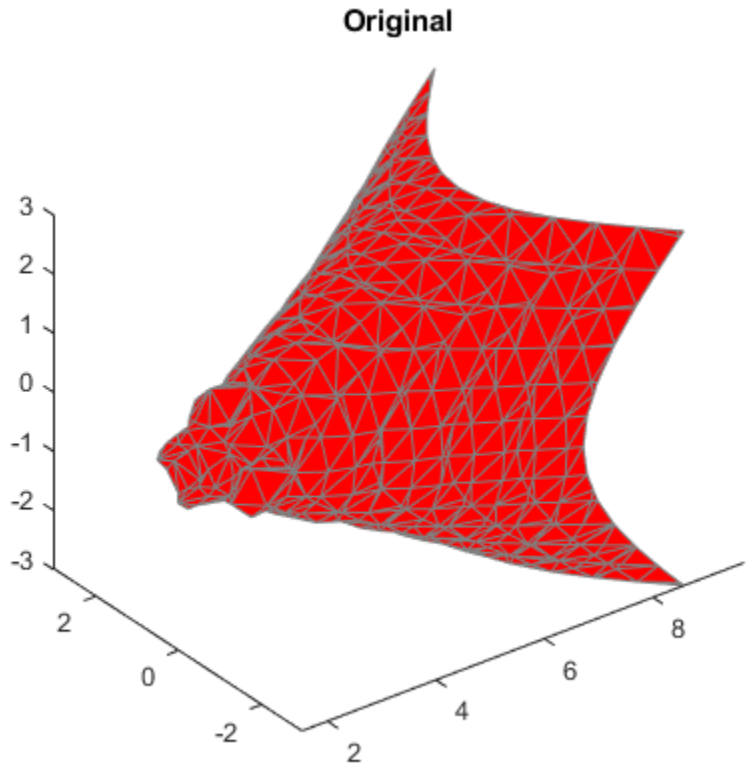
Examples

Reduce Size of Patch Faces

Load the flow data set, which represents the speed profile of a submerged jet within an infinite tank. Create two isosurfaces that provide a before and after view of the effects of shrinking the face size.

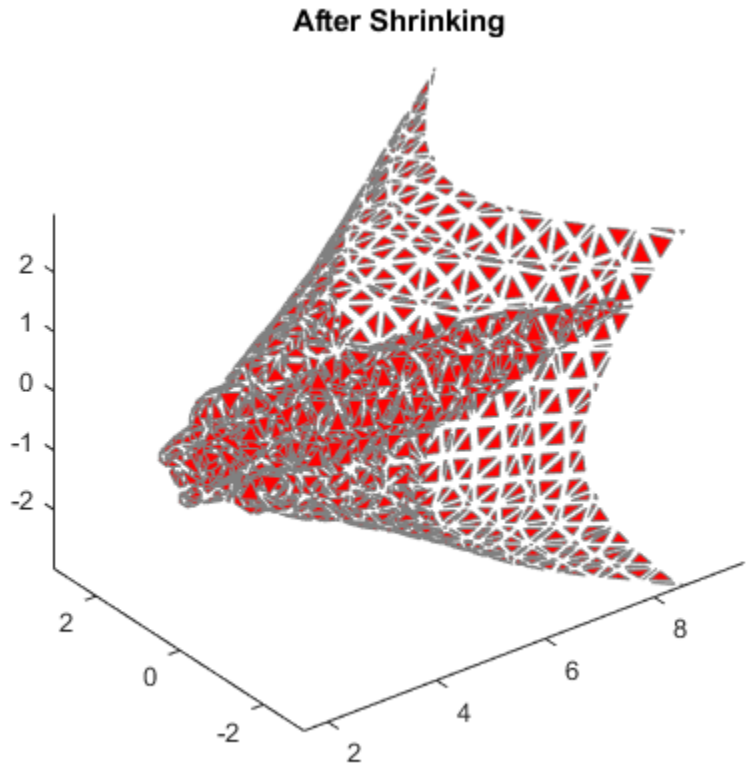
Sample the flow data at every other point using `reducevolume` and then use `isosurface` to generate the faces and vertices data. Draw the first isosurface. Then set up the view and add a title.

```
[x,y,z,v] = flow;
[x,y,z,v] = reducevolume(x,y,z,v,2);
fv = isosurface(x,y,z,v,-3);
p1 = patch(fv);
p1.FaceColor = 'red';
p1.EdgeColor = [0.5 0.5 0.5];
daspect([1 1 1]);
view(3);
axis tight
title('Original')
```



Use `shrinkfaces` to modify the face and vertex data and draw a second patch.

```
figure
p2 = patch(shrinkfaces(fv,.3));
p2.FaceColor = 'red';
p2.EdgeColor = [0.5 0.5 0.5];
daspect([1 1 1]);
view(3);
axis tight
title('After Shrinking')
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`axis` | `daspect` | `isosurface` | `patch` | `reducevolume` | `view`

Introduced before R2006a

sign

Sign function (signum function)

Syntax

```
sign(x)
```

Description

Y = `sign(x)` returns an array `Y` the same size as `x`, where each element of `Y` is:

- `1` if the corresponding element of `x` is greater than `0`.
- `0` if the corresponding element of `x` equals `0`.
- `-1` if the corresponding element of `x` is less than `0`.
- `x./abs(x)` if `x` is complex.

Examples

Find Sign Function

Find the sign function of a number.

```
sign(2)
```

```
ans = 1
```

Find the sign function of the values of a vector.

```
V = [-11 0 1.5 Inf NaN];
sign(V)
```

```
ans = 1×5
```

```
    -1     0     1     1   NaN
```

Find the sign function of the values of a matrix.

```
M = magic(3) - 5;
sign(M)
```

```
ans = 3×3

     1    -1     1
    -1     0     1
    -1     1    -1
```

Find the sign function of a complex number.

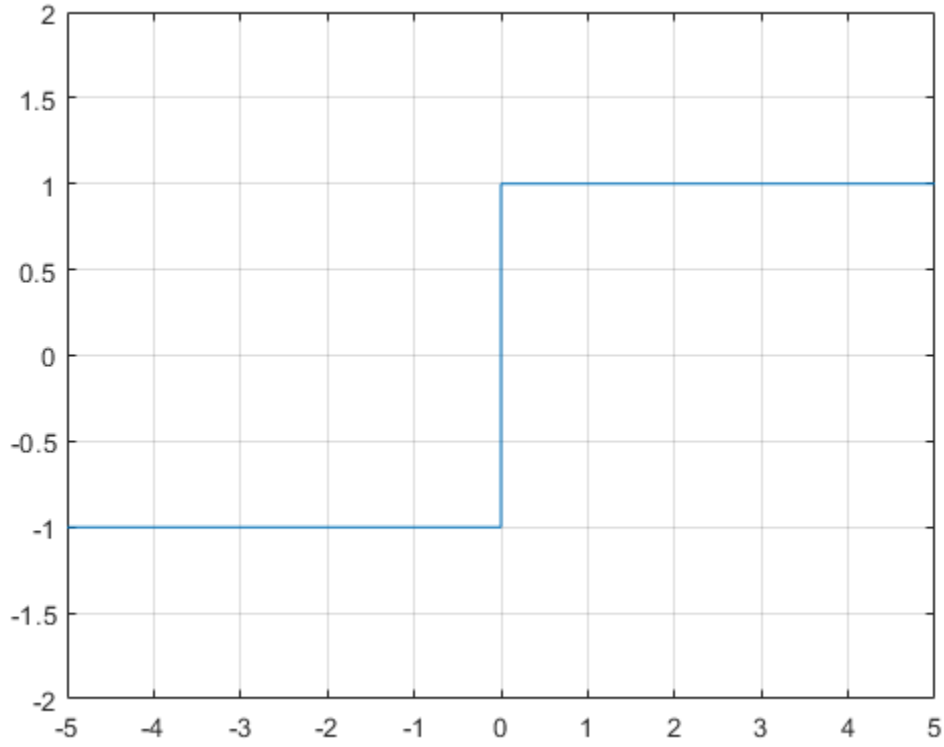
```
z = 4 - 3*i;
sign(z)
```

```
ans = 0.8000 - 0.6000i
```

Plot Sign Function

Plot the sign function and show its behavior at the zero-crossing. Use `eps` to represent values just above and below `0`.

```
x = [-5 -eps(1) 0 eps(1) 5];
y = sign(x);
plot(x,y)
ylim([-2 2])
grid on
```



Plot Real and Imaginary Parts of Sign Function

Plot real and imaginary parts of the sign function over $-3 < x < -3$ and $-3 < y < 3$.

First, create a mesh of values over `-3 < x < 3` and `-3 < y < 3` using `meshgrid`. Then create complex numbers from these values using `z = x + 1i*y`.

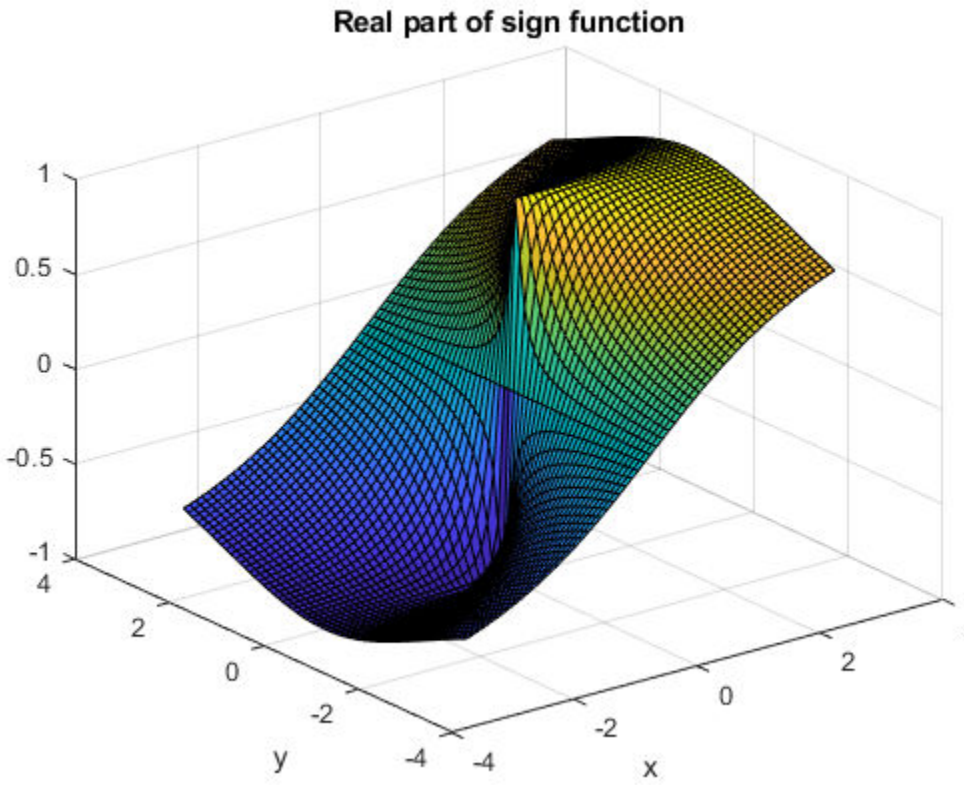
```
v = -3:0.1:3;
[x, y] = meshgrid(v);
z = x + 1i*y;
```

Find the real and imaginary parts of the sign function of `z`.

```
s = sign(z);
re = real(s);
im = imag(s);
```

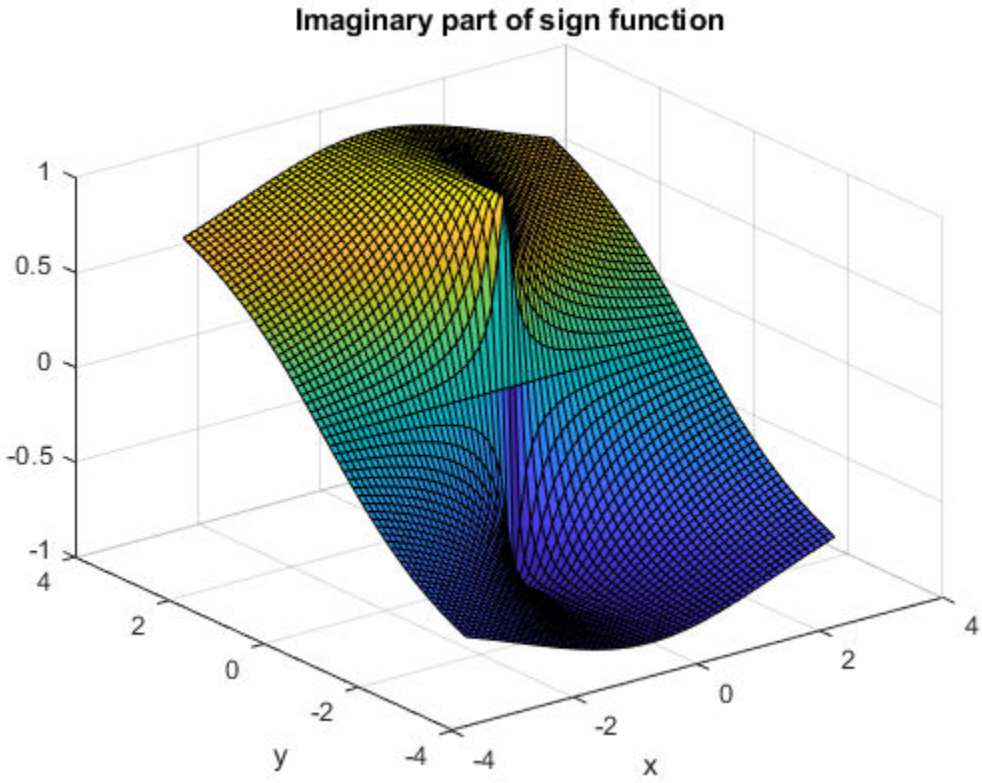
Plot the real and imaginary parts.

```
surf(x,y,re)
title('Real part of sign function')
xlabel('x')
ylabel('y')
```




```
figure(2)
surf(x,y,im)
title('Imaginary part of sign function')
```

```
xlabel('x')
ylabel('y')
```



Input Arguments

x — Input
scalar | vector | matrix | multidimensional array

Input, specified as a scalar, vector, matrix, or multidimensional array.

If an element of `x` is `NaN`, then `sign` returns `NaN` in the corresponding element of the output.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `duration`
Complex Number Support: Yes

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`abs` | `conj` | `imag` | `real`

Introduced before R2006a

sin

Sine of argument in radians

Syntax

```
Y = sin(X)
```

Description

`Y = sin(X)` returns the sine of the elements of `X`. The `sin` function operates element-wise on arrays. The function accepts both real and complex inputs.

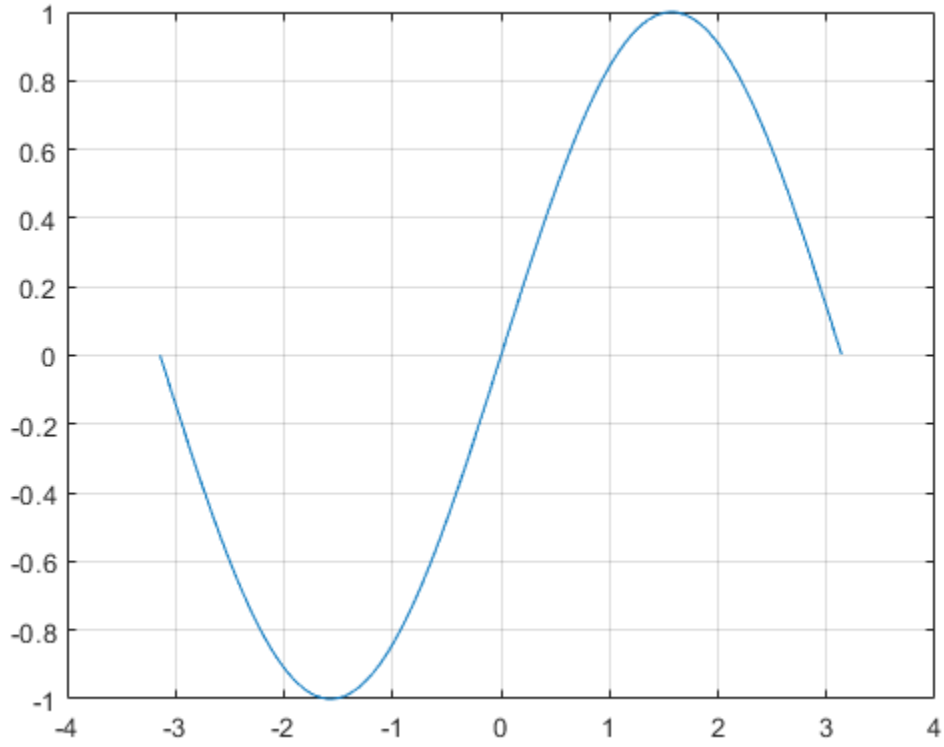
- For real values of `X`, `sin(X)` returns real values in the interval [-1, 1].
- For complex values of `X`, `sin(X)` returns complex values.

Examples

Plot Sine Function

Plot the sine function over the domain $-\pi \le x \le \pi$.

```
x = -pi:0.01:pi;
plot(x,sin(x)), grid on
```



Sine of Vector of Complex Angles

Calculate the sine of the complex angles in vector x.

```
x = [-i pi+i*pi/2 -1+i*4];
y = sin(x)
```

```
y = 1×3 complex

   0.0000 - 1.1752i   0.0000 - 2.3013i -22.9791 +14.7448i
```

Input Arguments

X — Input angle in radians
scalar | vector | matrix | multidimensional array

Input angle in radians, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Sine of input angle
scalar | vector | matrix | multidimensional array

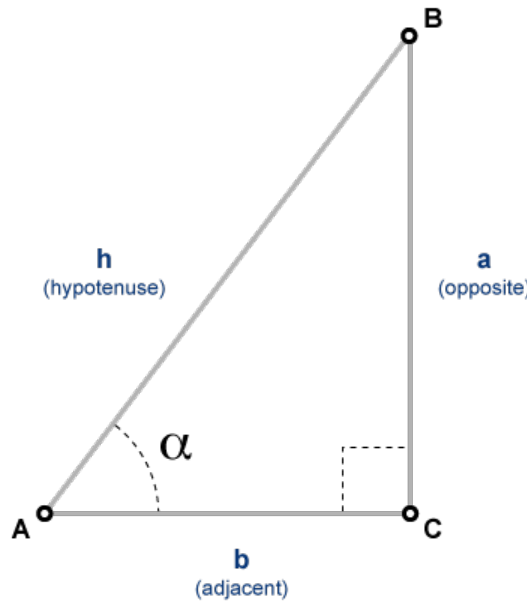
Sine of input angle, returned as a real-valued or complex-valued scalar, vector, matrix or multidimensional array.

More About

Sine Function

The sine of an angle, α, defined with reference to a right angled triangle is

$$\sin(\alpha) = \frac{\text{opposite side}}{\text{hypotenuse}} = \frac{a}{h} .$$



The sine of a complex argument, α, is

$$\sin(\alpha) = \frac{e^{i\alpha} - e^{-i\alpha}}{2i} .$$

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`asin` | `asind` | `sind` | `sinh` | `sinpi`

Introduced before R2006a

sind

Sine of argument in degrees

Syntax

```
Y = sind(X)
```

Description

`Y = sind(X)` returns the sine of the elements in `X`, which are expressed in degrees.

Examples

Sine of 180 degrees compared to sine of π radians

```
sind(180)
```

```
ans = 0
```

```
sin(pi)
```

```
ans = 1.2246e-16
```

Sine of vector of complex angles, specified in degrees

```
z = [90+i 15+2i 10+3i];
y = sind(z)
```

```
y = 1×3 complex

   1.0002 + 0.0000i   0.2590 + 0.0337i   0.1739 + 0.0516i
```

Input Arguments

X — Angle in degrees
scalar value | vector | matrix | N-D array

Angle in degrees, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `sind` operation is element-wise when X is nonscalar.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Sine of angle
scalar value | vector | matrix | N-D array

Sine of angle, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as X.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`asin` | `asind` | `sin`

Introduced before R2006a

sinpi

Compute sin(X*pi) accurately

Syntax

```
Y = sinpi(X)
```

Description

`Y = sinpi(X)` computes `sin(X*pi)` without explicitly computing `X*pi`. This calculation is more accurate than `sin(X*pi)` because the floating-point value of `pi` is an approximation of π. In particular:

- For integers, `sinpi(n)` is exactly zero.
- For odd integers, `sinpi(n/2)` is +1 or -1.

Examples

Calculate Sine of Multiples of π

Compare the accuracy of `sinpi(X)` vs. `sin(X*pi)`.

Create a vector of values.

```
X = [0 1/2 1 3/2 2];
```

Calculate the sine of `X*pi` using the normal `sin` function.

```
Y = sin(X*pi)
```

```
Y = 1×5

         0    1.0000    0.0000   -1.0000   -0.0000
```

The results contain small numerical errors due to the fact that `pi` is a floating-point approximation of the true value of π. For instance, `Y(3)` is not *exactly* zero even though $\sin(\pi) = 0$.

```
Y(3)
```

```
ans = 1.2246e-16
```

Use `sinpi` to calculate the same values. In this case, the results are exact.

```
Z = sinpi(X)
```

```
Z = 1×5

     0     1     0    -1     0
```

```
Z(3)
```

```
ans = 0
```

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cospi` | `sin` | `sind`

Introduced in R2018b

single

Single-precision arrays

Description

Single-precision variables in MATLAB are stored as 4-byte (32-bit) floating-point values of data type (class) `single`. For example:

```
y = single(10);
whos y
```

```
Name      Size            Bytes  Class     Attributes

  y         1x1                 4  single
```

For more information on floating-point values, see “Floating-Point Numbers”.

Creation

If you have an array of a different type, such as `double` or `int8`, then you can convert that array to single precision using the `single` function.

Syntax

```
Y = single(X)
```

Description

`Y = single(X)` converts the values in `X` to single precision.

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`

Examples

Convert to Single-Precision Variable

Convert a double-precision variable to single precision with the `single` function.

```
x = 100;
xtype = class(x)
```

```
xtype =
'double'
```

```
y = single(x)
```

```
y = single
    100
```

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cast` | `double` | `typecast`

Topics

“Floating-Point Numbers”
“Identifying Numeric Classes”

Introduced before R2006a

sinh

Hyperbolic sine

Syntax

```
Y = sinh(X)
```

Description

`Y = sinh(X)` returns the hyperbolic sine of the elements of `X`. The `sinh` function operates element-wise on arrays. The function accepts both real and complex inputs. All angles are in radians.

Examples

Hyperbolic Sine of Vector

Create a vector and calculate the hyperbolic sine of each value.

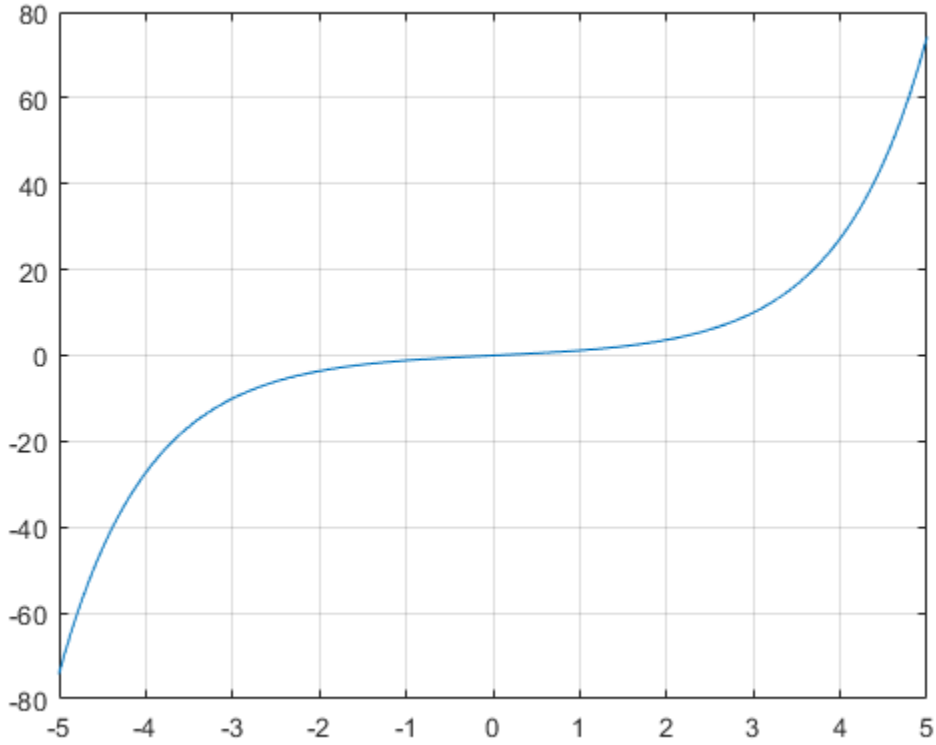
```
X = [0 pi 2*pi 3*pi];
Y = sinh(X)
```

```
Y = 1×4
10^3 ×

         0    0.0115    0.2677    6.1958
```

Graph of Hyperbolic Sine Function

Plot the hyperbolic sine over the domain $-5 \le x \le 5$.

```
x = -5:0.01:5;
y = sinh(x);
plot(x,y)
grid on
```

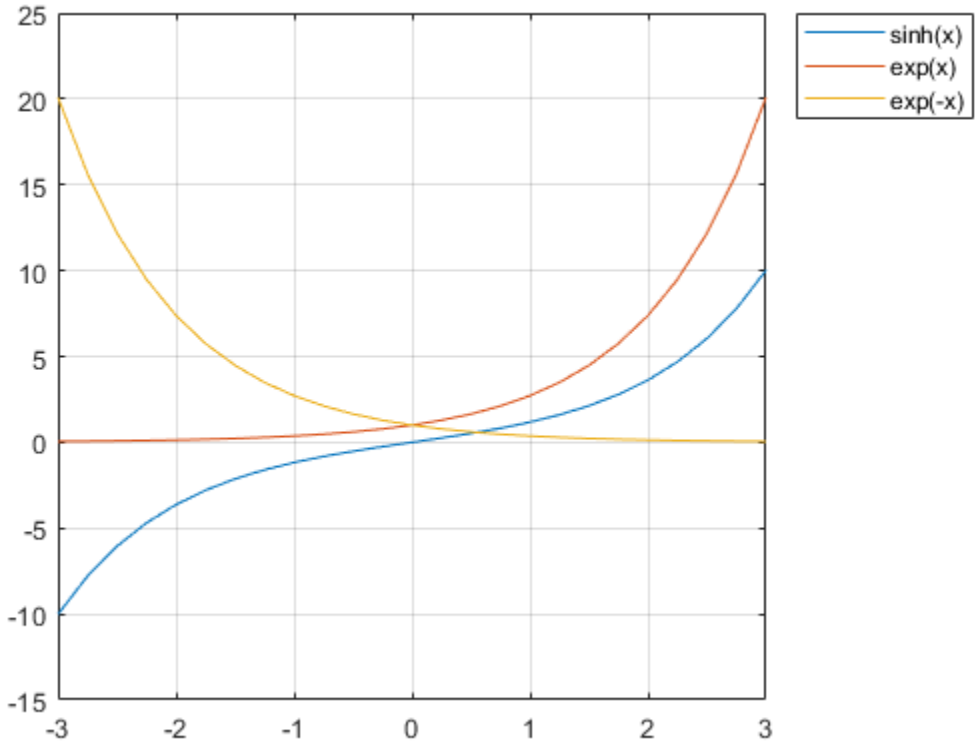


Plot Hyperbolic Sine and Exponential Functions

The hyperbolic sine satisfies the identity $\sinh(x) = \frac{e^x - e^{-x}}{2}$. In other words, $\sinh(x)$ is half the difference of the functions e^x and e^{-x}. Verify this by plotting the functions.

Create a vector of values between -3 and 3 with a step of 0.25. Calculate and plot the values of `sinh(x)`, `exp(x)`, and `exp(-x)`. As expected, the `sinh` curve is positive where `exp(x)` is large, and negative where `exp(-x)` is large.

```
x = -3:0.25:3;
y1 = sinh(x);
y2 = exp(x);
y3 = exp(-x);
plot(x,y1,x,y2,x,y3)
grid on
legend('sinh(x)','exp(x)','exp(-x)','Location','bestoutside')
```



Input Arguments

X — Input angles in radians
scalar | vector | matrix | multidimensional array

Input angles in radians, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Hyperbolic Sine

The hyperbolic sine of an angle x can be expressed in terms of exponential functions as

$$\sinh(x) = \frac{e^x - e^{-x}}{2}.$$

In terms of the traditional sine function with a complex argument, the identity is

$$\sinh(x) = -i\sin(ix).$$

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`asinh` | `cosh` | `sin`

Introduced before R2006a

size

Array size

Syntax

```
sz = size(A)
szdim = size(A,dim)
szdim = size(A,dim1,dim2,…,dimN)
[sz1,...,szN] = size( ___ )
```

Description

`sz = size(A)` returns a row vector whose elements are the lengths of the corresponding dimensions of `A`. For example, if `A` is a 3-by-4 matrix, then `size(A)` returns the vector `[3 4]`.

If `A` is a table or timetable, then `size(A)` returns a two-element row vector consisting of the number of rows and the number of table variables.

`szdim = size(A,dim)` returns the length of dimension `dim` when `dim` is a positive integer scalar. Starting in R2019b, you can also specify `dim` as a vector of positive integers to query multiple dimension lengths at a time. For example, `size(A,[2 3])` returns the lengths of the second and third dimensions of `A` in the 1-by-2 row vector `szdim`.

`szdim = size(A,dim1,dim2,…,dimN)` returns the lengths of dimensions `dim1,dim2,…,dimN` in the row vector `szdim` (starting in R2019b).

`[sz1,...,szN] = size( ___ )` returns the lengths of the queried dimensions of `A` separately.

Examples

Size of 4-D Array

Create a random 4-D array and return its size.

```
A = rand(2,3,4,5);
sz = size(A)
```

```
sz = 1×4

     2     3     4     5
```

Query only the length of the second dimension of A.

```
szdim2 = size(A,2)
```

```
szdim2 = 3
```

Starting in R2019b, you can query multiple dimension lengths at a time by specifying a vector dimension argument. For example, find the lengths of the first and third dimensions of A.

```
szdim13 = size(A,[1 3])
```

```
szdim13 = 1×2

     2     4
```

Find the lengths of the second through fourth dimensions of A.

```
szdim23 = size(A,2:4)
```

```
szdim23 = 1×3

     3     4     5
```

Alternatively, you can list the queried dimensions as separate input arguments.

```
szdim23 = size(A,2,3,4);
```

Size of Table

Create a table with 5 rows and 4 variables.

```
LastName = {'Smith';'Johnson';'Williams';'Jones';'Brown'};
Age = [38;43;38;40;49];
Height = [71;69;64;67;64];
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];

A = table(Age,Height,Weight,BloodPressure,'RowNames',LastName)
```

```
A=5×4 table
                Age    Height    Weight    BloodPressure
                ___    ______    ______    _____________

    Smith       38       71       176       124     93
    Johnson     43       69       163       109     77
    Williams    38       64       131       125     83
    Jones       40       67       133       117     75
    Brown       49       64       119       122     80
```

Find the size of the table. Although the `BloodPressure` variable contains two columns, `size` only counts the number of variables.

```
sz = size(A)
```

```
sz = 1×2

     5     4
```

Dimension Lengths as Separate Arguments

Create a random matrix and return the number of rows and columns separately.

```
A = rand(4,3);
[numRows,numCols] = size(A)
```

```
numRows = 4

numCols = 3
```

Input Arguments

A — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, a vector, a matrix, or a multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `function_handle` | `cell` | `categorical` | `datetime` | `duration` | `calendarDuration` | `table` | `timetable`
Complex Number Support: Yes

dim — Queried dimensions

positive integer scalar | vector of positive integer scalars

Queried dimensions, specified as a positive integer scalar or vector of positive integer scalars. If an element of `dim` is larger than `ndims(A)`, then `size` returns `1` in the corresponding element of the output.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

dim1,dim2,…,dimN — List of queried dimensions

positive integer scalars

List of queried dimensions, specified as positive integer scalars separated by commas. If an element of the list is larger than `ndims(A)`, then `size` returns `1` in the corresponding element of the output.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

sz — Array size

row vector of nonnegative integers

Array size, returned as a row vector of nonnegative integers.

- Each element of `sz` represents the length of the corresponding dimension of `A`. If any element of `sz` is equal to `0`, then `A` is an empty array.
- If `A` is a scalar, then `sz` is the row vector `[1 1]`.
- If `A` is a table or timetable, then `sz` is a two-element row vector containing the number of rows and the number of variables. Multiple columns within a single variable are not counted.
- If `A` is a character vector of type `char`, then `size` returns the row vector `[1 M]` where `M` is the number of characters. However, if `A` is a string scalar, `size` returns `[1 1]` because it is a single element of a string array. For example, compare the output of `size` for a character vector and string:

  ```
  szchar = size('mytext')
  ```

  ```
  szchar =

       1     6
  ```

  ```
  szstr = size("mytext")
  ```

  ```
  szstr =

       1     1
  ```

  To find the number of characters in a string, use the `strlength` function.

Data Types: `double`

`szdim` — Dimension lengths
nonnegative integer scalar | vector of nonnegative integer scalars

Dimension lengths, returned as a nonnegative integer scalar when `dim` is a scalar and a row vector of nonnegative integer scalars when `dim` is a vector. If an element of the specified dimension argument is larger than `ndims(A)`, then `size` returns `1` in the corresponding element of `szdim`.

Data Types: `double`

`sz1,...,szN` — Dimension lengths listed separately
nonnegative integer scalars

Dimension lengths listed separately, returned as nonnegative integer scalars separated by commas.

- When `dim` is not specified and fewer than `ndims(A)` output arguments are listed, then all remaining dimension lengths are collapsed into the last argument in the list. For example, if `A` is a 3-D array with size `[3 4 5]`, then `[sz1,sz2] = size(A)` returns `sz1 = 3` and `sz2 = 20`.
- When `dim` is specified, the number of output arguments must equal the number of queried dimensions.
- If you specify more than `ndims(A)` output arguments, then the extra trailing arguments are returned as `1`.

Data Types: `double`

Tips

- To determine if an array is empty, a scalar, or a matrix, use the functions `isempty`, `isscalar`, and `ismatrix`. You can also determine the orientation of a vector with the `isrow` and `iscolumn` functions.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`height` | `length` | `ndims` | `numel` | `strlength` | `width`

Introduced before R2006a

size

Package: `containers`

Size of `Map` object

Syntax

```
n = size(M,1)
szdim = size(M,dim)
sz = size(M)
[sz1,sz2,...,szL] = size(M)
```

Description

`n = size(M,1)` returns the number of key-value pairs in the input `Map` object. `size(M,1)` is equivalent to `length(M)` and to the property `M.Count`.

`szdim = size(M,dim)` returns the length of dimension `dim`. Since a `Map` object stores data as an `n`-by-`1` set of key-value pairs, `size` returns `1` when `dim > 1`.

`sz = size(M)` returns a two-element vector `[n 1]`, where `n` is the number of key-value pairs in `M`.

`[sz1,sz2,...,szL] = size(M)` returns `[n,1,...,1]`.

Examples

Size of Map

Create a `Map` object.

```
months = {'Jan','Feb','Mar','Apr'};
rainfall = [327.2 368.2 197.6 178.4];
M = containers.Map(months,rainfall)
```

```
M = 
  Map with properties:

        Count: 4
      KeyType: char
    ValueType: double
```

Return the number of key-value pairs.

```
n = size(M,1)
```

```
n = 4
```

Return the number of pairs as a vector.

```
sz = size(M)
```

```
sz = 1×2

     4     1
```

Input Arguments

M — Input Map object
Map object

Input Map object.

dim — Queried dimension
positive numeric scalar

Queried dimension, specified as a positive numeric scalar.

See Also

containers.Map | isKey | keys | length | values

Topics

"Overview of Map Data Structure"

“Create Map Object”
“Examine Contents of Map”

Introduced in R2008b

size

Class: `TriRep`

(Not recommended) Size of triangulation matrix

Note `size(TriRep)` is not recommended. Use `size(triangulation)` instead.

`TriRep` is not recommended. Use `triangulation` instead.

Syntax

```
size(TR)
```

Description

`size(TR)` provides size information for a triangulation matrix. The matrix is of size `mtri`-by-`nv`, where `mtri` is the number of simplices and `nv` is the number of vertices per simplex (triangle/tetrahedron, etc).

Input Arguments

TR	Triangulation matrix

More About

Simplex

A simplex is a triangle/tetrahedron or higher-dimensional equivalent.

See Also

`delaunayTriangulation` | `size` | `triangulation`

size

Size of `tscollection`

Syntax

```
tscsize = size(tsc)
```

Description

`tscsize = size(tsc)` returns a vector of the form `[n m]` where `n` is the length of the time vector of a `tscollection` object, and `m` is the number of `tscollection` members.

Examples

Size of `tscollection`

Create a `tscollection` object and display its size.

```
ts = timeseries([1.1 2.9 3.7 4.0 3.0]',1:5);
tsc = tscollection(ts);
tscsize = size(tsc)
```

```
tscsize = 1×2

     5     1
```

Input Arguments

`tsc` — Input `tscollection`
scalar

Input `tscollection`, specified as a scalar.

See Also

`length` | `timeseries` | `tscollection`

Introduced before R2006a

slice

Volume slice planes

Syntax

```
slice(X,Y,Z,V,xslice,yslice,zslice)
slice(V,xslice,yslice,zslice)
slice( ___ ,method)
slice(ax, ___ )
s = slice( ___ )
```

Description

`slice(X,Y,Z,V,xslice,yslice,zslice)` draws slices for the volumetric data `V`. Specify `X`,`Y`, and `Z` as the coordinate data. Specify `xslice`, `yslice`, and `zslice` as the slice locations using one of these forms:

- To draw one or more slice planes that are orthogonal to a particular axis, specify the slice arguments as a scalar or vector.
- To draw a single slice along a surface, specify all the slice arguments as matrices that define a surface.

`slice(V,xslice,yslice,zslice)` uses the default coordinate data for `V`. The (*x*,*y*,*z*) location for each element in `V` is based on the column, row, and page index, respectively.

`slice( ___ ,method)` specifies the interpolation method, where `method` can be `'linear'` (the default), `'cubic'`, or `'nearest'`. Use this option with any of the input arguments in previous syntaxes.

`slice(ax, ___ )` plots into the specified axes, instead of the current axes (`gca`).

`s = slice( ___ )` returns the `Surface` objects created. `slice` returns one `Surface` object for each slice.

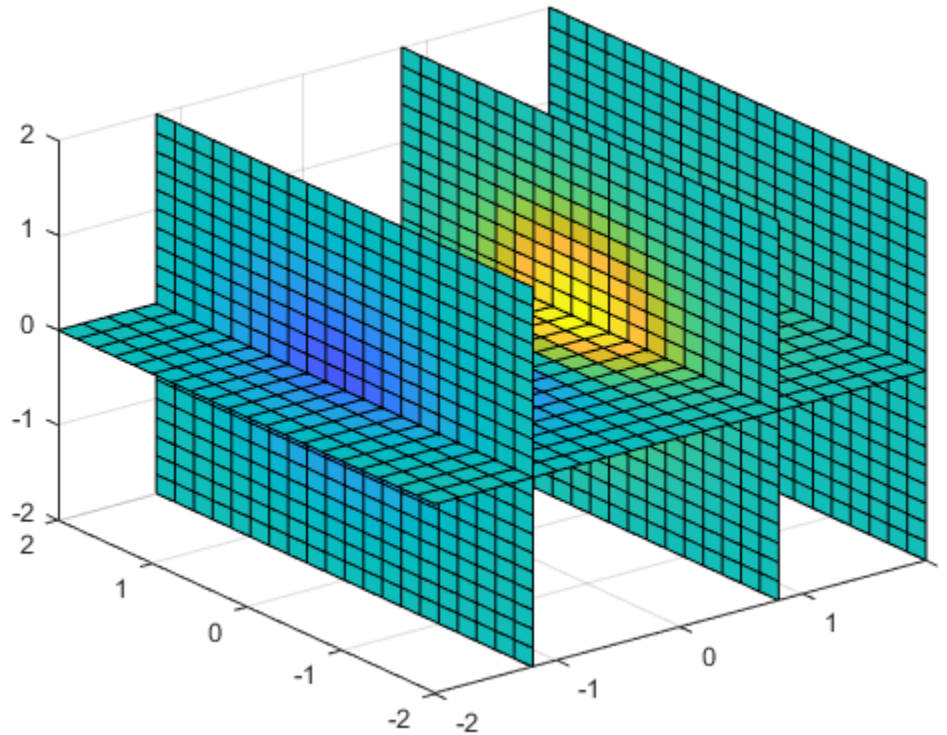
Examples

Volume Data Along Slices

Show volumetric data along slice planes that are orthogonal to each axis.

Create slice planes through the volume defined by $v = xe^{-x^2 - y^2 - z^2}$, where x, y, and z range from `[-2,2]`. Create slice planes orthogonal to the x-axis at the values `-1.2`, `0.8`, and `2` and orthogonal to the z-axis at the value `0`. Do not create any slice planes that are orthogonal to the y-axis by specifying an empty array.

```
[X,Y,Z] = meshgrid(-2:.2:2);
V = X.*exp(-X.^2-Y.^2-Z.^2);

xslice = [-1.2,0.8,2];
yslice = [];
zslice = 0;
slice(X,Y,Z,V,xslice,yslice,zslice)
```



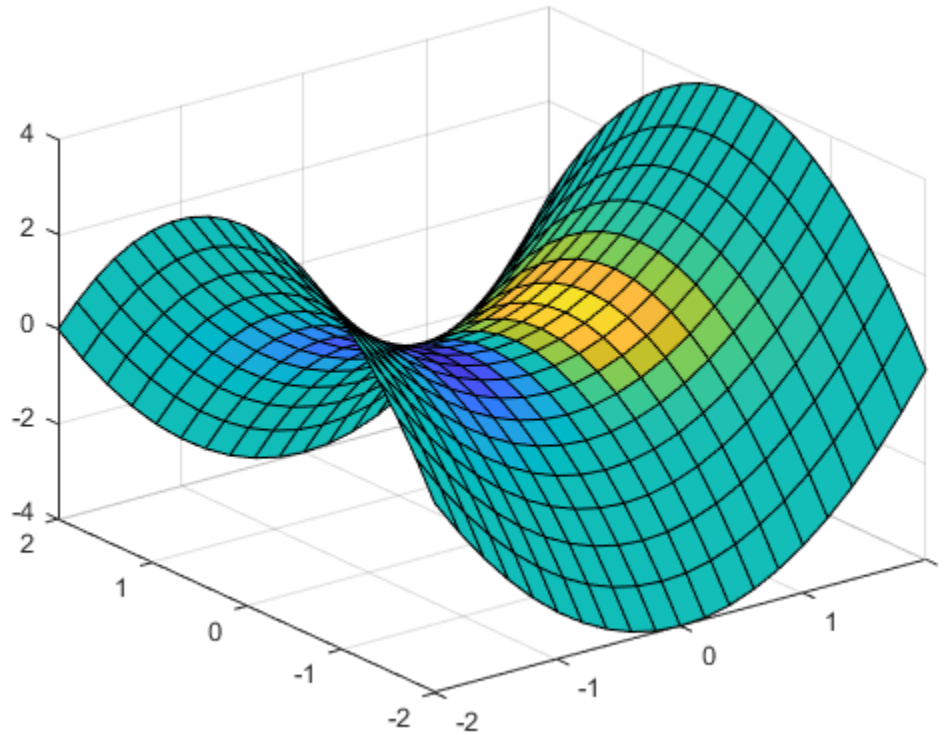
Volume Data Along Surface

Show volumetric data along a nonplanar slice. Define the surface where you want to show the volumetric data.

Create volume array `V` as the volume defined by $v = xe^{-x^2 - y^2 - z^2}$, where x, y, and z range from `[-5,5]`. Then, show a slice of the volume data along the surface defined by $z = x^2 - y^2$.

```
[X,Y,Z] = meshgrid(-5:0.2:5);
V = X.*exp(-X.^2-Y.^2-Z.^2);
```

```
[xsurf,ysurf] = meshgrid(-2:0.2:2);
zsurf = xsurf.^2-ysurf.^2;
slice(X,Y,Z,V,xsurf,ysurf,zsurf)
```

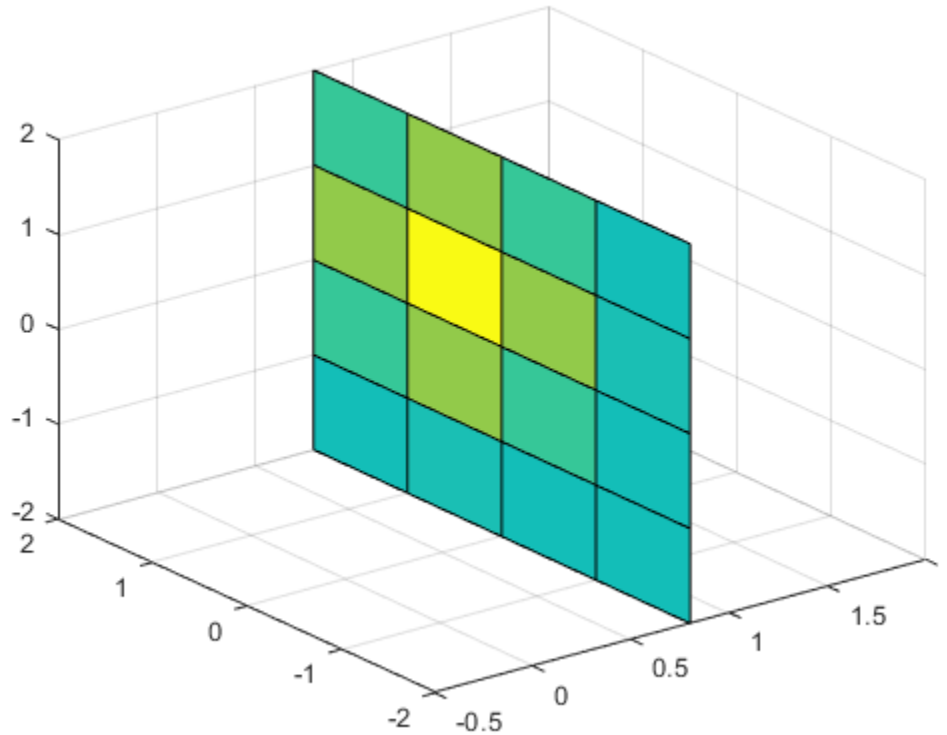


Specify Interpolation Method

Create a slice plane through volumetric data. Specify the interpolation method for the data values.

Create a slice plane orthogonal to the *x*-axis at the value 0.8. Since the volume data is not defined for *x* values of 0.8, the `slice` function interpolates the nearby values. To use the nearest data point value, specify the interpolation method as `'nearest'`.

```
[X,Y,Z] = meshgrid(-2:2);
V = X.*exp(-X.^2-Y.^2-Z.^2);
xslice = 0.8;
yslice = [];
zslice = [];
slice(X,Y,Z,V,xslice,yslice,zslice,'nearest')
```



Input Arguments

X — *x*-axis coordinate data
vector | 3-D array

x-axis coordinate data, specified as a vector or a 3-D array the same size as `V`. If you specify a 3-D array, it must be monotonic and orthogonally spaced, as if produced by the `meshgrid` function.

Data Types: `single` | `double`

Y — *y*-axis coordinate data
vector | 3-D array

y-axis coordinate data, specified as a vector or a 3-D array the same size as `V`. If you specify a 3-D array, it must be monotonic and orthogonally spaced, as if produced by the `meshgrid` function.

Data Types: `single` | `double`

Z — *z*-axis coordinate data
vector | 3-D array

z-axis coordinate data, specified as a vector or a 3-D array the same size as `V`. If you specify a 3-D array, it must be monotonic and orthogonally spaced, as if produced by the `meshgrid` function.

Data Types: `single` | `double`

V — Volume data
3-D array

Volume data, specified as a 3-D array.

Data Types: `single` | `double`

`xslice` — *x*-axis slice values
scalar | vector | `[]` | matrix

x-axis slice values, specified in one of these forms:

- Scalar — Draw one slice plane orthogonal to the *x*-axis at the specified location.
- Vector — Draw multiple slice planes orthogonal to the *x*-axis at the specified locations.

- `[]` — Do not draw any slice planes.
- Matrix — Draw the slice along a surface, instead of a plane. If you use this option, then `yslice` and `zslice` also must be matrices that are the same size.

Example: `[2 4]` draws slice planes orthogonal to the x-axis at the values 2 and 4.

Data Types: `single` | `double`

`yslice` — *y*-axis slice values
scalar | vector | `[]` | matrix

y-axis slice values, specified in one of these forms:

- Scalar — Draw one slice plane orthogonal to the *y*-axis at the specified location.
- Vector — Draw multiple slice planes orthogonal to the *y*-axis at the specified locations.
- `[]` — Do not draw any slice planes.
- Matrix — Draw the slice along a surface, instead of a plane. If you use this option, then `xslice` and `zslice` also must be matrices that are the same size.

Example: `[2 4]` draws slice planes orthogonal to the y-axis at the values 2 and 4.

Data Types: `single` | `double`

`zslice` — *z*-axis slice values
scalar | vector | `[]` | matrix

z-axis slice values, specified in one of these forms:

- Scalar — Draw one slice plane orthogonal to the *z*-axis at the specified location.
- Vector — Draw multiple slice planes orthogonal to the *z*-axis at the specified locations.
- `[]` — Do not draw any slice planes.
- Matrix — Draw the slice along a surface, instead of a plane. If you use this option, then `xslice` and `yslice` also must be matrices that are the same size.

Example: `[2 4]` draws slice planes orthogonal to the z-axis at the values 2 and 4.

Data Types: `single` | `double`

`method` — Interpolation method
`'linear'` (default) | `'cubic'` | `'nearest'`

Interpolation method, specified as one of these values:

- `'linear'` — Linear interpolation of the values at neighboring grid points in each respective dimension
- `'cubic'` — Cubic interpolation of the values at neighboring grid points in each respective dimension
- `'nearest'` — Nearest grid point value

ax — Target axes
current axes (default) | `Axes` object

Target axes, specified as an `Axes` object. If you do not specify the axes, then `slice` plots into the current axes.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`contourslice` | `interp3` | `isosurface` | `meshgrid`

Introduced before R2006a

smooth3

Smooth 3-D data

Syntax

Description

`W = smooth3(V)` smooths the input data `V` and returns the smoothed data in `W`.

`W = smooth3(V,'filter')` *`filter`* determines the convolution kernel and can be one of these values:

- `'gaussian'`
- `'box'` (default)

`W = smooth3(V,'filter',size)` sets the size of the convolution kernel (default is `[3 3 3]`). If `size` is scalar, then `size` is interpreted as `[size, size, size]`.

`W = smooth3(V,'filter',size,sd)` sets an attribute of the convolution kernel. When *`filter`* is `gaussian`, `sd` is the standard deviation (default is .65).

Examples

Smooth Isosurface Data

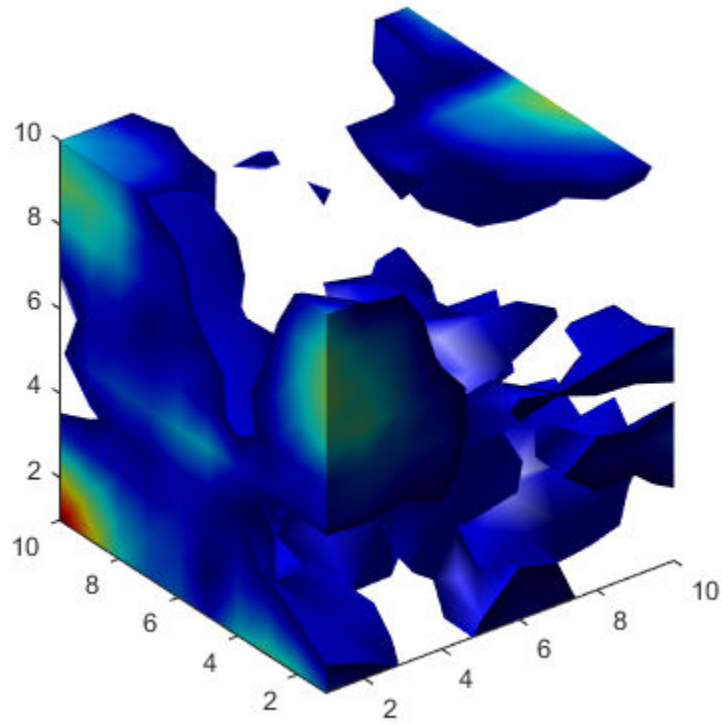
Create a 10-by-10-by-10 array of random data and smooth it.

```
data = rand(10,10,10);
data = smooth3(data,'box',5);
```

Display the data as an isosurface with end caps.

```
patch(isocaps(data,.5),...
   'FaceColor','interp','EdgeColor','none');
```

```
p1 = patch(isosurface(data,.5),...
   'FaceColor','blue','EdgeColor','none');
isonormals(data,p1);
view(3);
axis vis3d tight
camlight left
colormap('jet');
lighting gouraud
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`isocaps` | `isonormals` | `isosurface` | `patch`

Topics

“Techniques for Visualizing Scalar Volume Data”

Introduced before R2006a

smoothdata

Smooth noisy data

Syntax

```
B = smoothdata(A)
B = smoothdata(A,dim)

B = smoothdata(___,method)
B = smoothdata(___,method,window)

B = smoothdata(___,nanflag)

B = smoothdata(___,Name,Value)

[B,window] = smoothdata(___)
```

Description

`B = smoothdata(A)` returns a moving average of the elements of a vector using a fixed window length that is determined heuristically. The window slides down the length of the vector, computing an average over the elements within each window.

- If `A` is a matrix, then `smoothdata` computes the moving average down each column.
- If `A` is a multidimensional array, then `smoothdata` operates along the first dimension whose size does not equal 1.
- If `A` is a table or timetable with numeric variables, then `smoothdata` operates on each variable separately.

`B = smoothdata(A,dim)` operates along the dimension `dim` of `A`. For example, if `A` is a matrix, then `smoothdata(A,2)` smooths the data in each row of `A`.

`B = smoothdata(___,method)` specifies the smoothing method for either of the previous syntaxes. For example, `B = smoothdata(A,'sgolay')` uses a Savitzky-Golay filter to smooth the data in `A`.

`B = smoothdata(___,method,window)` specifies the length of the window used by the smoothing method. For example, `smoothdata(A,'movmedian',5)` smooths the data in `A` by taking the median over a five-element sliding window.

`B = smoothdata(___,nanflag)` specifies how `NaN` values are treated for any of the previous syntaxes. `'omitnan'` ignores `NaN` values and `'includenan'` includes them when computing within each window.

`B = smoothdata(___,Name,Value)` specifies additional parameters for smoothing using one or more name-value pair arguments. For example, if `t` is a vector of time values, then `smoothdata(A,'SamplePoints',t)` smooths the data in `A` relative to the times in `t`.

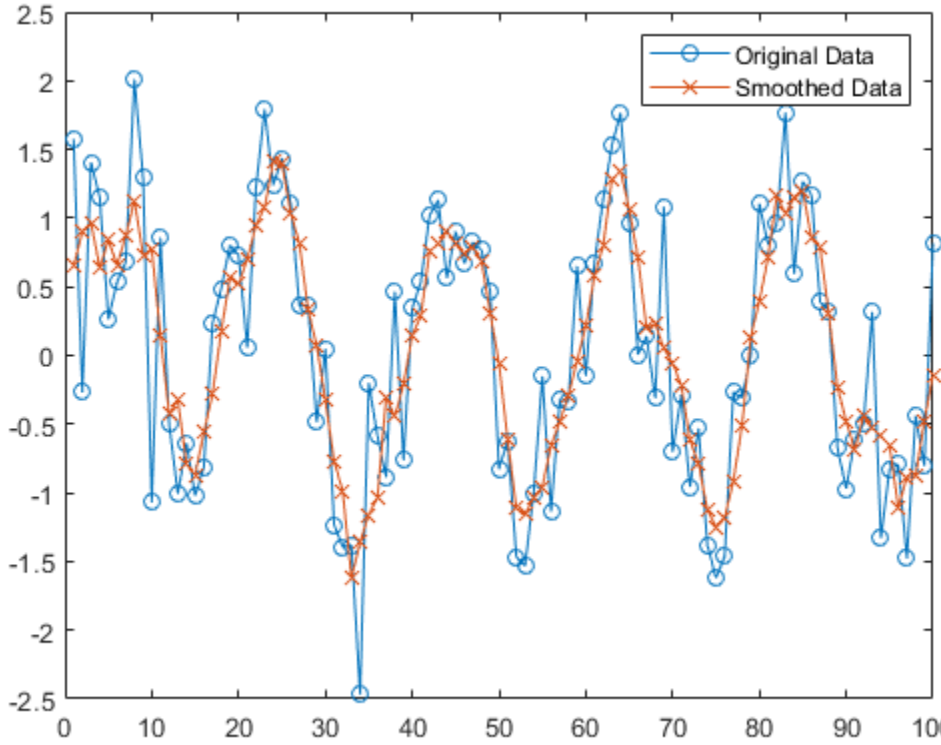
`[B,window] = smoothdata(___)` also returns the moving window length.

Examples

Smooth Data with Moving Average

Create a vector containing noisy data, and smooth the data with a moving average. Plot the original and smoothed data.

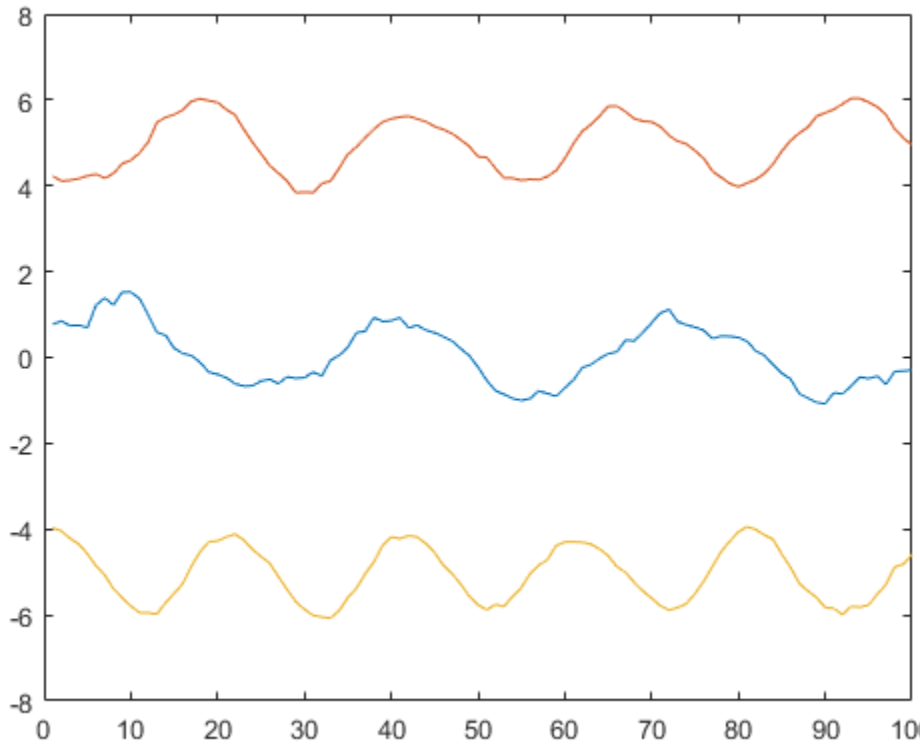
```
x = 1:100;
A = cos(2*pi*0.05*x+2*pi*rand) + 0.5*randn(1,100);
B = smoothdata(A);
plot(x,A,'-o',x,B,'-x')
legend('Original Data','Smoothed Data')
```



Matrix of Noisy Data

Create a matrix whose rows represent three noisy signals. Smooth the three signals using a moving average, and plot the smoothed data.

```
x = 1:100;
s1 = cos(2*pi*0.03*x+2*pi*rand) + 0.5*randn(1,100);
s2 = cos(2*pi*0.04*x+2*pi*rand) + 0.4*randn(1,100) + 5;
s3 = cos(2*pi*0.05*x+2*pi*rand) + 0.3*randn(1,100) - 5;
A = [s1; s2; s3];
B = smoothdata(A,2);
plot(x,B(1,:),x,B(2,:),x,B(3,:))
```



Gaussian Filter

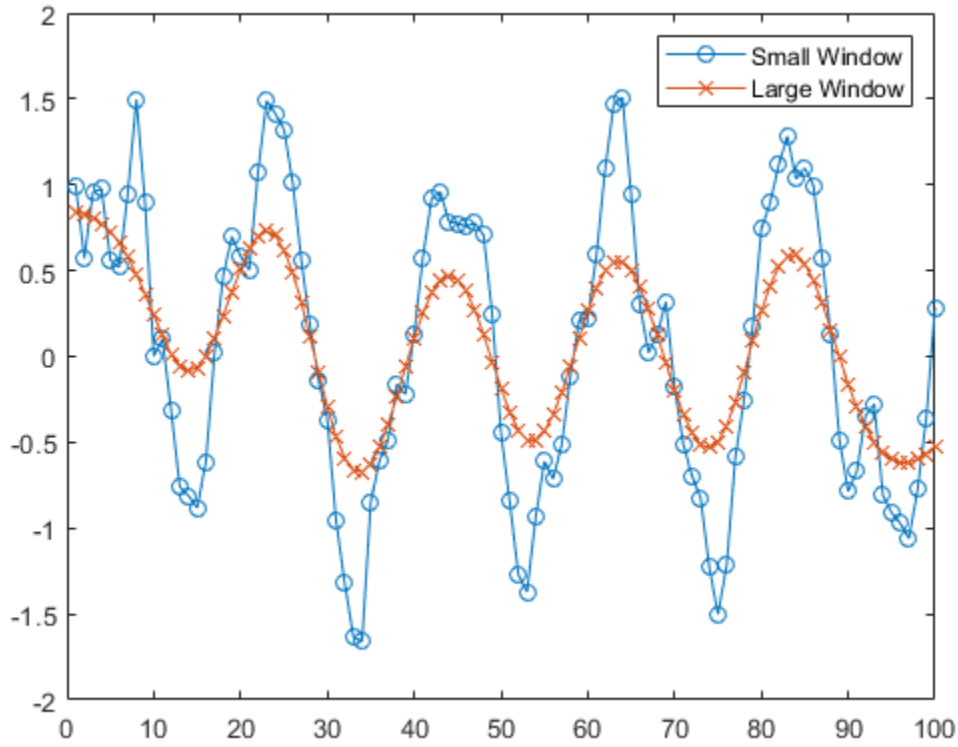
Smooth a vector of noisy data with a Gaussian-weighted moving average filter. Display the window length used by the filter.

```
x = 1:100;
A = cos(2*pi*0.05*x+2*pi*rand) + 0.5*randn(1,100);
[B, window] = smoothdata(A,'gaussian');
window
```

```
window = 4
```

Smooth the original data with a larger window of length 20. Plot the smoothed data for both window lengths.

```
C = smoothdata(A,'gaussian',20);
plot(x,B,'-o',x,C,'-x')
legend('Small Window','Large Window')
```



Vector with NaN

Create a noisy vector containing NaN values, and smooth the data ignoring NaN, which is the default.

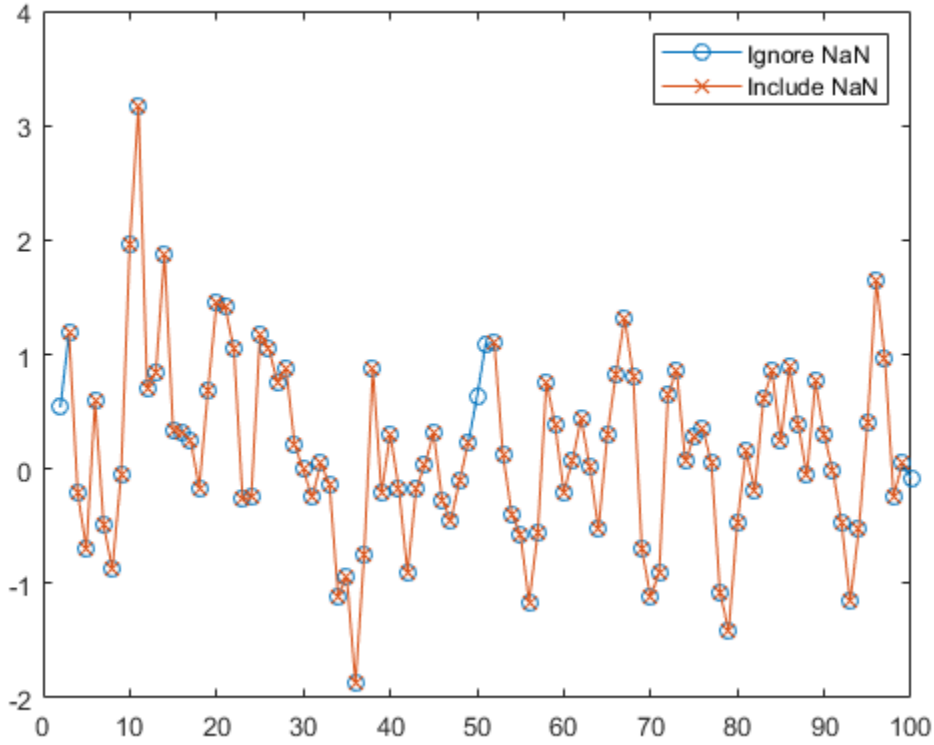
```
A = [NaN randn(1,48) NaN randn(1,49) NaN];
B = smoothdata(A);
```

Smooth the data including NaN values. The average in a window containing NaN is NaN.

```
C = smoothdata(A,'includenan');
```

Plot the smoothed data in B and C.

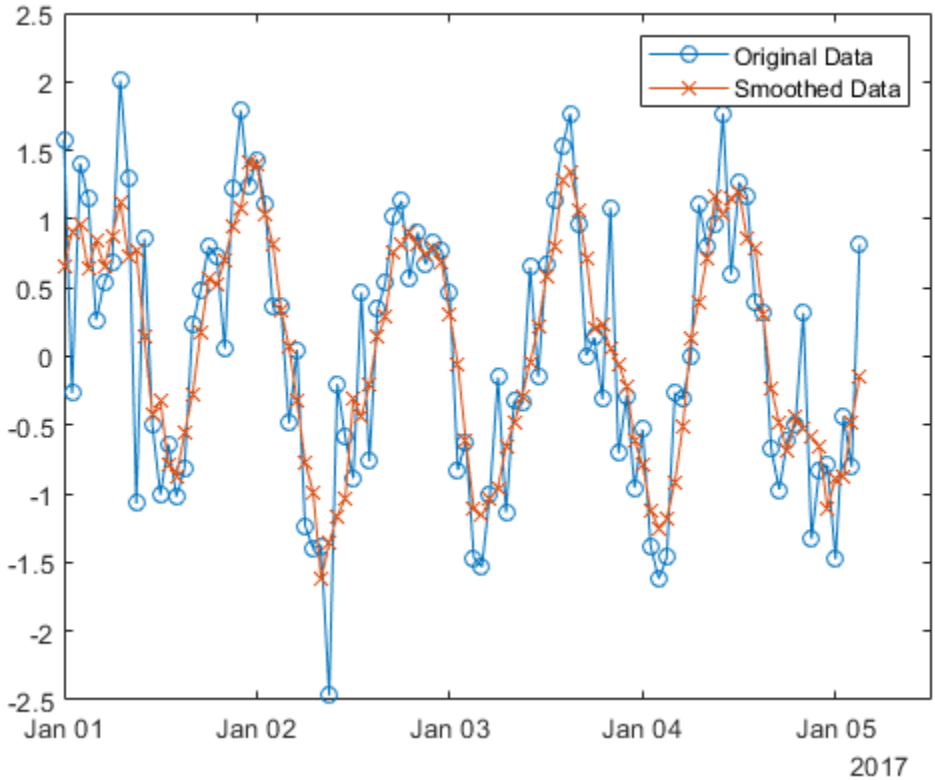
```
plot(1:100,B,'-o',1:100,C,'-x')
legend('Ignore NaN','Include NaN')
```



Smooth Data with Sample Points

Create a vector of noisy data that corresponds to a time vector `t`. Smooth the data relative to the times in `t`, and plot the original data and the smoothed data.

```
x = 1:100;
A = cos(2*pi*0.05*x+2*pi*rand) + 0.5*randn(1,100);
t = datetime(2017,1,1,0,0,0) + hours(0:99);
B = smoothdata(A,'SamplePoints',t);
plot(t,A,'-o',t,B,'-x')
legend('Original Data','Smoothed Data')
```



Input Arguments

A — Input array
vector | matrix | multidimensional array | table | timetable

Input array, specified as a vector, matrix, multidimensional array, table, or timetable. If A is a table or timetable, then either the variables must be numeric, or you must use the `'DataVariables'` name-value pair to list numeric variables explicitly. Specifying variables is useful when you are working with a table that also contains non-numeric variables.

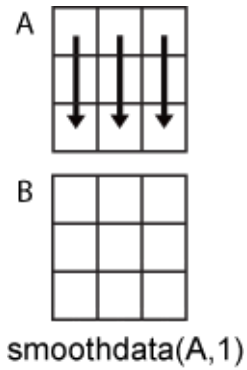
Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `table` | `timetable`
Complex Number Support: Yes

dim — Dimension to operate along
positive integer scalar

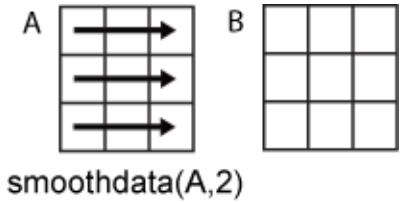
Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a matrix A.

`B = smoothdata(A,1)` smooths the data in each column of A.



`B = smoothdata(A,2)` smooths the data in each row of A.



When `A` is a table or timetable, `dim` is not supported. `smoothdata` operates along each table or timetable variable separately.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`method` — Smoothing method
`'movmean'` (default) | `'movmedian'` | `'gaussian'` | `'lowess'` | `'loess'` | `'rlowess'` | `'rloess'` | `'sgolay'`

Smoothing method, specified as one of the following:

- `'movmean'` — Moving average over each window of `A`. This method is useful for reducing periodic trends in data.
- `'movmedian'` — Moving median over each window of `A`. This method is useful for reducing periodic trends in data when outliers are present.
- `'gaussian'` — Gaussian-weighted moving average over each window of `A`.
- `'lowess'` — Linear regression over each window of `A`. This method can be computationally expensive, but results in fewer discontinuities.
- `'loess'` — Quadratic regression over each window of `A`. This method is slightly more computationally expensive than `'lowess'`.
- `'rlowess'` — Robust linear regression over each window of `A`. This method is a more computationally expensive version of the method `'lowess'`, but it is more robust to outliers.
- `'rloess'` — Robust quadratic regression over each window of `A`. This method is a more computationally expensive version of the method `'loess'`, but it is more robust to outliers.
- `'sgolay'` — Savitzky-Golay filter, which smooths according to a quadratic polynomial that is fitted over each window of `A`. This method can be more effective than other methods when the data varies rapidly.

`window` — Window length

positive integer scalar | two-element vector of positive integers | positive duration scalar | two-element vector of positive durations

Window length, specified as a positive integer scalar, a two-element vector of positive integers, a positive duration scalar, or a two-element vector of positive durations.

When `window` is a positive integer scalar, then the window is centered about the current element and contains `window-1` neighboring elements. If `window` is even, then the window is centered about the current and previous elements. If `window` is a two-element vector of positive integers `[b f]`, then the window contains the current element, `b` elements backward, and `f` elements forward.

When `A` is a timetable or when `'SamplePoints'` is specified as a `datetime` or `duration` vector, `window` must be of type `duration`, and the window is computed relative to the sample points.

When the window length is also specified as an output argument, the output value matches the input value.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`nanflag` — `NaN` condition

`'omitnan'` (default) | `'includenan'`

`NaN` condition, specified as one of the following values:

- `'omitnan'` — Ignore `NaN` values in the input. If a window contains all `NaN` values, then `smoothdata` returns `NaN`.
- `'includenan'` — Include `NaN` values when computing within each window, resulting in `NaN`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `smoothdata(A,'SmoothingFactor',0.5)`

`SmoothingFactor` — Window size factor
scalar ranging from 0 to 1

Window size factor, specified as the comma-separated pair consisting of `'SmoothingFactor'` and a scalar ranging from 0 to 1. The value of `'SmoothingFactor'` adjusts the level of smoothing by scaling the heuristic window size. Values near 0 produce smaller moving window lengths, resulting in less smoothing. Values near 1 produce larger moving window lengths, resulting in more smoothing.

`'SmoothingFactor'` is 0.25 by default and can only be specified when `window` is not specified.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`SamplePoints` — Sample points
vector

Sample points, specified as the comma-separated pair consisting of `'SamplePoints'` and a vector. The sample points represent the location of the data in `A`. Sample points do not need to be uniformly sampled. By default, the sample points vector is `[1 2 3 ...]`.

Moving windows are defined relative to the sample points, which must be sorted and contain unique elements. For example, if `t` is a vector of times corresponding to the input data, then `smoothdata(rand(1,10),3,'SamplePoints',t)` has a window that represents the time interval between `t(i)-1.5` and `t(i)+1.5`.

When the sample points vector has data type `datetime` or `duration`, then the moving window length must have type `duration`.

This name-value pair is not supported when the input data is a timetable.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `datetime` | `duration`

`DataVariables` — Table variables
variable name | cell array of variable names | numeric vector | logical vector | function handle

Table variables, specified as the comma-separated pair consisting of `'DataVariables'` and a variable name, a cell array of variable names, a numeric vector, a logical vector, or a function handle. The `'DataVariables'` value indicates which columns of the input table to smooth, and can be one of the following:

- A character vector specifying a single table variable name
- A cell array of character vectors where each element is a table variable name
- A vector of table variable indices
- A logical vector whose elements each correspond to a table variable, where `true` includes the corresponding variable and `false` excludes it
- A function handle that takes the table as input and returns a logical scalar

Example: `'Age'`

Example: `{'Height','Weight'}`

Example: `@isnumeric`

Data Types: `char` | `cell` | `double` | `single` | `logical` | `function_handle`

`Degree` — Savitzky-Golay degree
nonnegative integer

Savitzky-Golay degree, specified as the comma-separated pair consisting of `'Degree'` and a nonnegative integer. This name-value pair can only be specified when `'sgolay'` is the specified smoothing method. The value of `'Degree'` corresponds to the degree of the polynomial in the Savitzky-Golay filter that fits the data within each window, which is 2 by default.

The value of `'Degree'` must be less than the window length for uniform sample points. For nonuniform sample points, the value must be less than the maximum number of points in any window.

Output Arguments

`B` — Output array
vector | matrix | multidimensional array

Output array, returned as a vector, matrix, or multidimensional array. `B` is the same size and type as `A`.

`window` — Window length
positive integer scalar | two-element vector of positive integers | positive duration scalar | two-element vector of positive durations

Window length, returned as a positive integer scalar, a two-element vector of positive integers, a positive duration scalar, or a two-element vector of positive durations.

When `window` is specified as an input argument, the output value matches the input value. When `window` is not specified as an input argument, then its value is the scalar heuristically determined by `smoothdata` based on the input data.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

Algorithms

When the window size for the smoothing method is not specified, `smoothdata` computes a default window size based on a heuristic. For a smoothing factor τ, the heuristic estimates a moving average window size that attenuates approximately 100*τ percent of the energy of the input data.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- Tall timetables are not supported.
- The `'rlowess'` and `'rloess'` methods are not supported.
- Multiple outputs are not supported.
- You must specify the window size. Automatic selection of the window size is not supported.
- The `'SamplePoints'` and `'SmoothingFactor'` name-value pairs are not supported.
- The value of `'DataVariables'` cannot be a function handle.

For more information, see “Tall Arrays”.

See Also

Smooth Data | `fillmissing` | `filter` | `movmad` | `movmean` | `movmedian`

Topics

Clean Messy Data and Locate Extrema Using Live Editor Tasks
"Data Smoothing and Outlier Detection"

Introduced in R2017a

Smooth Data

Smooth noisy data in the Live Editor

Description

The **Smooth Data** task lets you interactively smooth noisy data. The task automatically generates MATLAB code for your live script.

Using this task, you can:

- Customize the method for smoothing data in a workspace variable.
- Adjust parameters to generate less or more smoothing.
- Automatically visualize the smoothed data.

Open the Task

To add the **Smooth Data** task to a live script in the MATLAB Editor:

- On the **Live Editor** tab, select **Task > Smooth Data**.
- In a code block in the script, type a relevant keyword, such as `smooth` or `noisy`. Select **Smooth Data** from the suggested command completions.

Parameters

`Smoothing method` — Method for smoothing data

`Moving mean` (default) | `Moving median` | `Gaussian filter` | `Local linear regression` | `Local quadratic regression` | `Robust local linear regression` | `Robust local quadratic regression` | `Savitzky-Golay polynomial filter` | ...

Specify the smoothing method as one of the following options, which operate over local windows of data.

Method	Description
`Moving mean`	Moving average. This method is useful for reducing periodic trends in data.
`Moving median`	Moving median. This method is useful for reducing periodic trends in data when outliers are present.
`Gaussian filter`	Gaussian-weighted moving average.
`Local linear regression`	Linear regression. This method can be computationally expensive, but it results in fewer discontinuities.
`Local quadratic regression`	Quadratic regression. This method is slightly more computationally expensive than local linear regression.
`Robust local linear regression`	Robust linear regression. This method is a more computationally expensive version of local linear regression, but it is more robust to outliers.
`Robust local quadratic regression`	Robust quadratic regression. This method is a more computationally expensive version of local quadratic regression, but it is more robust to outliers.
`Savitzky-Golay polynomial filter`	Savitzky-Golay polynomial filter, which smooths according to a polynomial of specified degree, and is fitted over each window. This method can be more effective than other methods when the data varies rapidly.

`Moving window` — Window for smoothing methods
`Centered` (default) | `Asymmetric`

Specify the window type and size for the smoothing method instead of specifying a general smoothing factor.

Window	Description
`Centered`	Specified window length centered about the current point.
`Asymmetric`	Specified window containing the number of elements before the current point and the number of elements after the current point.

Window sizes are relative to the **X-axis** variable units.

See Also

Functions
`smoothdata`

Live Editor Tasks
Clean Missing Data | Clean Outlier Data | Find Change Points | Find Local Extrema | Remove Trends

Topics

"Add Interactive Tasks to a Live Script"
Clean Messy Data and Locate Extrema Using Live Editor Tasks
"Data Smoothing and Outlier Detection"

Introduced in R2019b

snapnow

Take snapshot of image for inclusion in published document

Syntax

```
snapnow
```

Description

`snapnow`, in a file being published, takes a snapshot of the most recently generated image or plot. The snapshot appears at the end of the code section that contains the `snapnow` command. For more information about publishing, see “Publish and Share MATLAB Code”.

Outside of the context of publishing a file, MATLAB interprets a `snapnow` command as a `drawnow` command.

Examples

Capture Snapshot of Figure in Loop

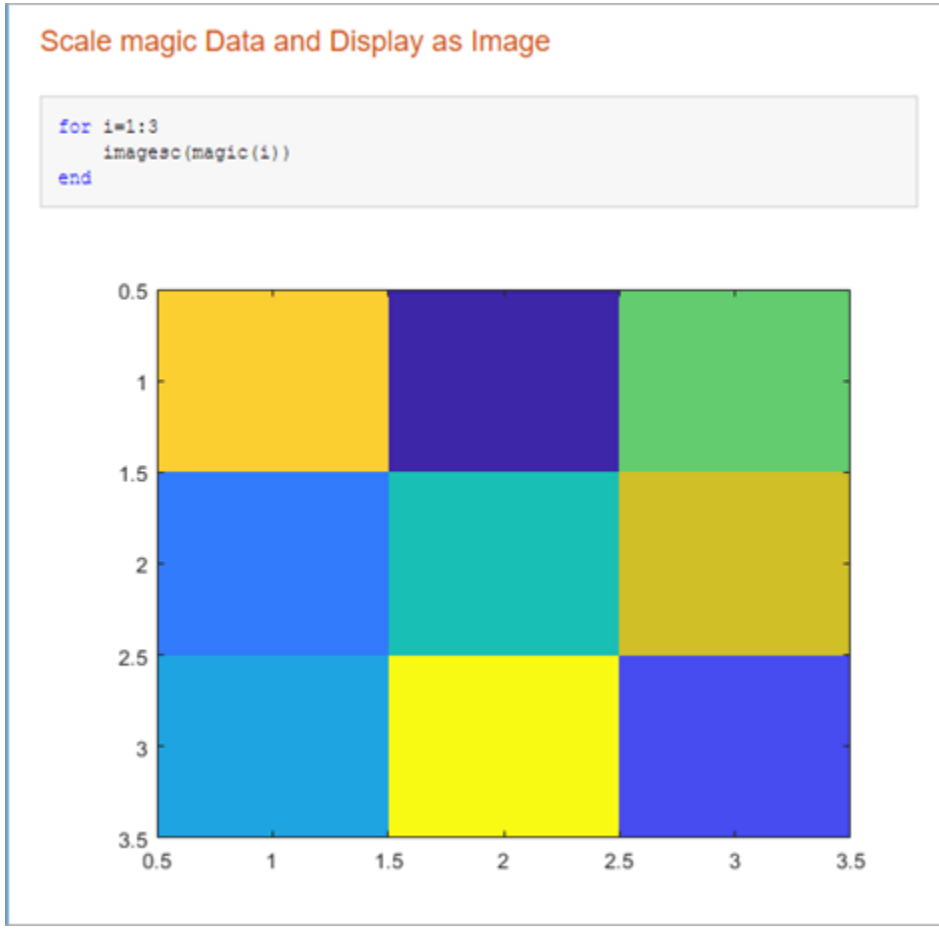
Use `snapnow` to capture a snapshot of an image after each iteration of a loop.

Create a file `loopIterations.m` with this code that runs a `for` loop three times and produces graphics after every iteration.

```
%% Scale magic Data and Display as Image
for i=1:3
    imagesc(magic(i))
end
```

Save and then publish the file. MATLAB displays the published file with only a snapshot of the final image.

```
publish('loopIterations.m')
web('html/loopIterations.html')
```

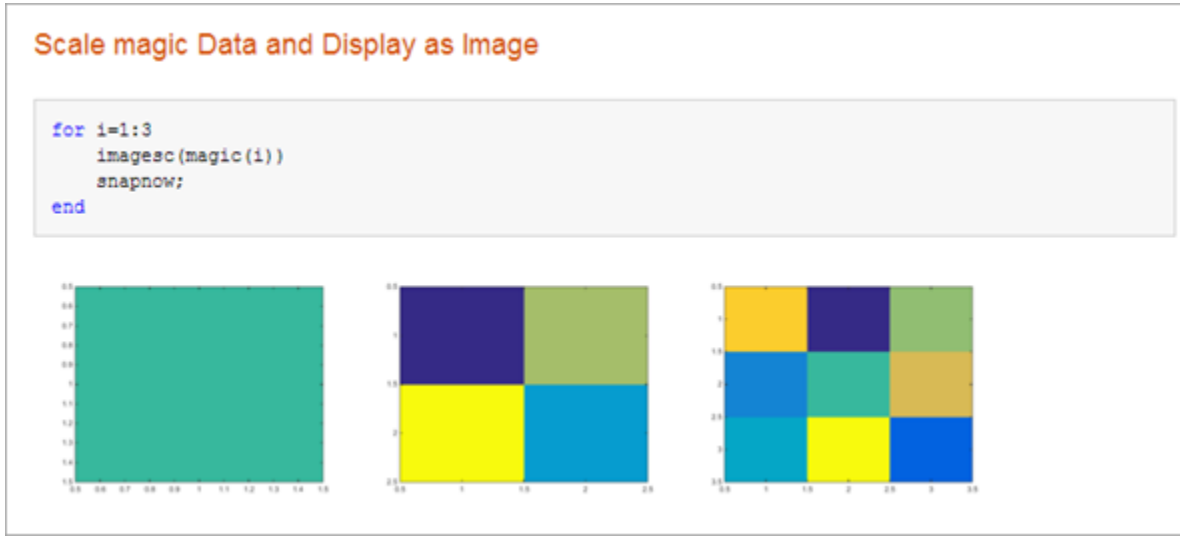


Add a call to the `snapnow` function inside the `for` loop.

```
%% Scale magic Data and Display as Image
for i=1:3
    imagesc(magic(i))
    snapnow;
end
```

Save and then publish the file. MATLAB displays the published file with a snapshot for each loop iteration.

```
publish('loopIterations.m')
web('html/loopIterations.html')
```



See Also

`drawnow` | `publish`

Topics

"Publishing Markup"

Introduced in R2008b

sort

Sort array elements

Syntax

```
B = sort(A)
B = sort(A,dim)
B = sort( ___ ,direction)
B = sort( ___ ,Name,Value)
[B,I] = sort( ___ )
```

Description

`B = sort(A)` sorts the elements of `A` in ascending order.

- If `A` is a vector, then `sort(A)` sorts the vector elements.
- If `A` is a matrix, then `sort(A)` treats the columns of `A` as vectors and sorts each column.
- If `A` is a multidimensional array, then `sort(A)` operates along the first array dimension whose size does not equal 1, treating the elements as vectors.

`B = sort(A,dim)` returns the sorted elements of `A` along dimension `dim`. For example, if `A` is a matrix, then `sort(A,2)` sorts the elements of each row.

`B = sort( ___ ,direction)` returns sorted elements of `A` in the order specified by `direction` using any of the previous syntaxes. `'ascend'` indicates ascending order (the default) and `'descend'` indicates descending order.

`B = sort( ___ ,Name,Value)` specifies additional parameters for sorting. For example, `sort(A,'ComparisonMethod','abs')` sorts the elements of `A` by magnitude.

`[B,I] = sort( ___ )` also returns a collection of index vectors for any of the previous syntaxes. `I` is the same size as `A` and describes the arrangement of the elements of `A` into `B` along the sorted dimension. For example, if `A` is a vector, then `B = A(I)`.

Examples

Sort Vector in Ascending Order

Create a row vector and sort its elements in ascending order.

```
A = [9 0 -7 5 3 8 -10 4 2];
B = sort(A)
```

```
B = 1×9

   -10    -7     0     2     3     4     5     8     9
```

Sort Matrix Rows in Ascending Order

Create a matrix and sort each of its rows in ascending order.

```
A = [3 6 5; 7 -2 4; 1 0 -9]
```

```
A = 3×3

     3     6     5
     7    -2     4
     1     0    -9
```

```
B = sort(A,2)
```

```
B = 3×3

     3     5     6
    -2     4     7
    -9     0     1
```

Sort Matrix Columns in Descending Order

Create a matrix and sort its columns in descending order.

```
A = [10 -12 4 8; 6 -9 8 0; 2 3 11 -2; 1 1 9 3]
```

```
A = 4×4

    10   -12     4     8
     6    -9     8     0
     2     3    11    -2
     1     1     9     3
```

```
B = sort(A,'descend')
```

```
B = 4×4

    10     3    11     8
     6     1     9     3
     2    -9     8     0
     1   -12     4    -2
```

Sort String Array

Starting in R2017a, you can create string arrays using double quotes, and sort them using the `sort` function. Sort strings in each column of a string array according to Unicode® dictionary order.

```
A = ["Santos","Burns"; ...
     "Jones","Morita"; ...
     "Petrov","Adams"];
B = sort(A)
```

```
B = 3x2 string array
    "Jones"     "Adams"
    "Petrov"    "Burns"
    "Santos"    "Morita"
```

Sort the strings in each row.

```
B = sort(A,2)
```

```
B = 3x2 string array
    "Burns"    "Santos"
```

```
    "Jones"    "Morita"
    "Adams"    "Petrov"
```

Sort and Index `datetime` Array

Create an array of `datetime` values and sort them in ascending order, that is, from the earliest to the latest calendar date.

```
ds = {'2012-12-22';'2063-04-05';'1992-01-12'};
A = datetime(ds,'Format','yyyy-MM-dd')
```

```
A = 3x1 datetime array
   2012-12-22
   2063-04-05
   1992-01-12
```

```
[B,I] = sort(A)
```

```
B = 3x1 datetime array
   1992-01-12
   2012-12-22
   2063-04-05


I = 3×1

     3
     1
     2
```

`B` lists the sorted dates and `I` contains the corresponding indices of `A`.

Access the sorted elements from the original array directly by using the index array `I`.

```
A(I)
```

```
ans = 3x1 datetime array
   1992-01-12
   2012-12-22
```

```
2063-04-05
```

Sort 3-D Array

Create a 2-by-2-by-2 array and sort its elements in ascending order along the third dimension.

```
A(:,:,1) = [2 3; 1 6];
A(:,:,2) = [-1 9; 0 12];
A
```

```
A =
A(:,:,1) =

     2     3
     1     6


A(:,:,2) =

    -1     9
     0    12
```

```
B = sort(A,3)
```

```
B =
B(:,:,1) =

    -1     3
     0     6


B(:,:,2) =

     2     9
     1    12
```

Use `A(:)`, the column representation of `A`, to sort all of the elements of `A`.

```
B = sort(A(:))
```

```
B = 8×1

    -1
     0
     1
     2
     3
     6
     9
    12
```

Complex Vector

Sort the elements of a complex vector by their real parts. For elements with equal real parts, `sort` breaks the tie based on their imaginary parts.

```
A = [1+2i 3+i i 0 -i];
B = sort(A,'ComparisonMethod','real')
```

```
B = 1×5 complex

   0.0000 - 1.0000i   0.0000 + 0.0000i   0.0000 + 1.0000i   1.0000 + 2.0000i   3.0000 -
```

Input Arguments

A — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

- If `A` is a scalar, then `sort(A)` returns `A`.
- If `A` is complex, then by default, `sort` sorts the elements by magnitude. If more than one element has equal magnitude, then the elements are sorted by phase angle on the interval $(-\pi, \pi]$.
- If `A` is a cell array of character vectors or a string array, then `sort(A)` sorts the elements according to the code order for the UTF-16 character encoding scheme. The

sort is case-sensitive. For more information on sorting character and string arrays, see “Sort Order for Character and String Arrays” on page 1-14231.

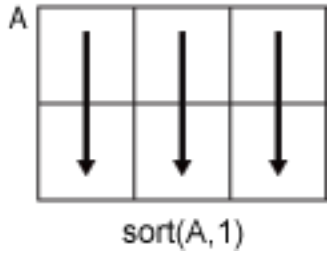
- If `A` is a string array, then `sort` reorders the elements of the array, but does not reorder characters within the strings.
- If `A` is a categorical array, then the sorting order is based on the category order returned by `categories(A)`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `cell` | `categorical` | `datetime` | `duration`
Complex Number Support: Yes

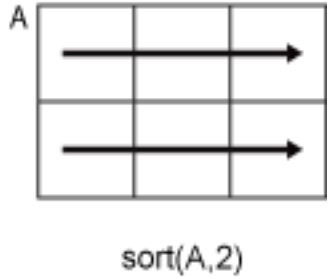
`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

- Consider a matrix `A`. `sort(A,1)` sorts the elements in the columns of `A`.



- `sort(A,2)` sorts the elements in the rows of `A`.



`sort` returns `A` if `dim` is greater than `ndims(A)`. `dim` is not supported when `A` is a cell array, that is, `sort` only operates along the first array dimension whose size does not equal 1.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`direction` — Sorting direction
`'ascend'` (default) | `'descend'`

Sorting direction, specified as `'ascend'` or `'descend'`. `direction` is not supported when `A` is a cell array, that is, `sort` only sorts in ascending order.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `sort(A,'MissingPlacement','last')`

`MissingPlacement` — Placement of missing values
`'auto'` (default) | `'first'` | `'last'`

Placement of missing values (`NaN`, `NaT`, `<undefined>`, and `missing`) specified as the comma-separated pair consisting of `'MissingPlacement'` and one of the following:

- `'auto'` — Missing elements are placed last for ascending order and first for descending order.
- `'first'` — Missing elements are placed first.
- `'last'` — Missing elements are placed last.

`ComparisonMethod` — Element comparison method
`'auto'` (default) | `'real'` | `'abs'`

Element comparison method, specified as the comma-separated pair consisting of `'ComparisonMethod'` and one of the following:

- `'auto'` — Sort `A` by `real(A)` when `A` is real, and sort by `abs(A)` when `A` is complex.
- `'real'` — Sort `A` by `real(A)` when `A` is real or complex. If `A` has elements with equal real parts, then use `imag(A)` to break ties.

- `'abs'` — Sort `A` by `abs(A)` when `A` is real or complex. If `A` has elements with equal magnitude, then use `angle(A)` in the interval (-π,π] to break ties.

Output Arguments

`B` — Sorted array
vector | matrix | multidimensional array

Sorted array, returned as a vector, matrix, or multidimensional array. `B` is the same size and type as `A`. The order of the elements in `B` preserves the order of any equal elements in `A`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `cell` | `categorical` | `datetime` | `duration`

`I` — Sort index
vector | matrix | multidimensional array

Sort index, returned as a vector, matrix, or multidimensional array. `I` is the same size as `A`. The index vectors are oriented along the same dimension that `sort` operates on. For example, if `A` is a 2-by-3 matrix, then `[B,I] = sort(A,2)` sorts the elements in each row of `A`. The output `I` is a collection of 1-by-3 row index vectors describing the rearrangement of each row of `A`.

More About

Sort Order for Character and String Arrays

MATLAB stores characters as Unicode using the UTF-16 character encoding scheme. Character and string arrays are sorted according to the UTF-16 code point order. For the characters that are also the ASCII characters, this order means that uppercase letters come before lowercase letters. Digits and some punctuation also come before letters.

Tips

- The `sortrows` function provides additional flexibility for subsorting over multiple columns of matrix or table inputs.
- The `sort` function and the relational operators use different orderings for complex numbers. For more information, see “Relational Operations”.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

- You must specify the dimension to sort, as in `sort(X,dim)`.
- Sorting the tall dimension, as in `sort(X,1)`, is only supported for column vectors.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The first input argument must not be a cell array.
- If `A` is complex with all zero imaginary parts, then MATLAB might convert `A` to `real(A)` before calling `sort(A)`. In this case, MATLAB sorts `A` by `real(A)`, but the generated code sorts `A` by `abs(A)`. To make the generated code match MATLAB, use `sort(real(A))` or `sort(A,'ComparisonMethod','real')`. See “Code Generation for Complex Data with Zero-Valued Imaginary Parts” (MATLAB Coder).
- If you supply `dim`, then it must be constant.
- For limitations related to variable-size inputs, see “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`issorted` | `max` | `min` | `sortrows` | `topkrows` | `unique`

Topics

“Reshaping and Rearranging Arrays”

Introduced before R2006a

sortrows

Sort rows of matrix or table

Syntax

```
B = sortrows(A)
B = sortrows(A,column)
B = sortrows( ___ ,direction)
B = sortrows( ___ ,Name,Value)
[B,index] = sortrows( ___ )
```

```
tblB = sortrows(tblA)
```

```
tblB = sortrows(tblA,'RowNames')
tblB = sortrows(tblA,rowDimName)
tblB = sortrows(tblA,vars)
tblB = sortrows( ___ ,direction)
tblB = sortrows( ___ ,Name,Value)
[tblB,index] = sortrows( ___ )
```

Description

`B = sortrows(A)` sorts the rows of a matrix in ascending order based on the elements in the first column. When the first column contains repeated elements, `sortrows` sorts according to the values in the next column and repeats this behavior for succeeding equal values.

`B = sortrows(A,column)` sorts `A` based on the columns specified in the vector `column`. For example, `sortrows(A,4)` sorts the rows of `A` in ascending order based on the elements in the fourth column. `sortrows(A,[4 6])` first sorts the rows of `A` based on the elements in the fourth column, then based on the elements in the sixth column to break ties.

`B = sortrows( ___ ,direction)` sorts the rows of `A` in the order specified by `direction` for any of the previous syntaxes. `direction` can be `'ascend'` (default) for ascending order or `'descend'` for descending order. `direction` can also be a cell array

whose elements are `'ascend'` and `'descend'`, where each element corresponds to a column that `sortrows` operates on. For example, `sortrows(A,[4 6],{'ascend' 'descend'})` sorts the rows of `A` in ascending order based on the fourth column, then in descending order based on the sixth column to break ties.

`B = sortrows( ___ ,Name,Value)` specifies additional parameters for sorting rows. For example, `sortrows(A,'ComparisonMethod','abs')` sorts the elements of `A` by magnitude.

`[B,index] = sortrows( ___ )` also returns an index vector that describes the rearrangement of rows such that `B = A(index,:)`.

`tblB = sortrows(tblA)` sorts the rows of a table in ascending order based on the values in the first variable. If elements in the first variable are repeated, then `sortrows` sorts by the elements in the second variable, and so on.

If `tblA` is a timetable, then `sortrows` sorts the rows of `tblA` in ascending order based on its row times. Row times of a timetable label the rows along the first dimension of the timetable.

`tblB = sortrows(tblA,'RowNames')` sorts a table based on its row names. Row names of a table label the rows along the first dimension of the table. If `tblA` does not have row names, that is, if `tblA.Properties.RowNames` is empty, then `sortrows` returns `tblA`.

This syntax is not supported when `tblA` is a timetable.

`tblB = sortrows(tblA,rowDimName)` sorts `tblA` by row labels `rowDimName` along the first dimension.

- If `tblA` is a table, then row labels are row names.
- If `tblA` is a timetable, then row labels are row times.

`tblB = sortrows(tblA,vars)` sorts a table by the elements in the variables specified by `vars`. For example, `sortrows(tblA,{'Var1','Var2'})` first sorts the rows of `tblA` based on the elements in `Var1`, then by the elements in `Var2`.

- If `tblA` is a table and it has row names, then `vars` can include the row names.
- If `tblA` is a timetable, then `vars` can include the row times.

`tblB = sortrows( ___ ,direction)` sorts `tblA` in the order specified by `direction` for any of the previous table syntaxes. `direction` can be `'ascend'` or `'descend'`,

which is applied to all specified variables, row names, or row times that `sortrows` operates on. `direction` can also be a cell array whose elements are `'ascend'` and `'descend'`, where each element corresponds to the specified variables, row names, or row times being sorted on.

`tblB = sortrows( ___ ,Name,Value)` specifies additional parameters for sorting rows of a table or timetable. For example, `sortrows(tblA,'Var1','MissingPlacement','first')` sorts based on the elements in `Var1`, ordering missing elements such as `NaN` at the beginning of the table.

`[tblB,index] = sortrows( ___ )` also returns an index vector such that `tblB = tblA(index,:)`.

Examples

Sort Rows of Matrix

Create a matrix and sort its rows in ascending order based on the elements in the first column. When the first column contains repeated elements, `sortrows` looks to the elements in the second column to break the tie. For repeated elements in the second column, `sortrows` looks to the third column, and so on.

```
A = floor(gallery('uniformdata',[6 7],0)*100);
A(1:4,1) = 95;  A(5:6,1) = 76;  A(2:4,2) = 7;  A(3,3) = 73
```

```
A = 6×7

    95    45    92    41    13     1    84
    95     7    73    89    20    74    52
    95     7    73     5    19    44    20
    95     7    40    35    60    93    67
    76    61    93    81    27    46    83
    76    79    91     0    19    41     1
```

```
B = sortrows(A)
```

```
B = 6×7

    76    61    93    81    27    46    83
    76    79    91     0    19    41     1
```

```
    95     7    40    35    60    93    67
    95     7    73     5    19    44    20
    95     7    73    89    20    74    52
    95    45    92    41    13     1    84
```

Sort the rows of A based on the values in the second column. When the specified column has repeated elements, the corresponding rows maintain their original order.

```
C = sortrows(A,2)
```

```
C = 6×7

    95     7    73    89    20    74    52
    95     7    73     5    19    44    20
    95     7    40    35    60    93    67
    95    45    92    41    13     1    84
    76    61    93    81    27    46    83
    76    79    91     0    19    41     1
```

Sort the rows of A based on the elements in the first column, and look to the seventh column to break any ties.

```
D = sortrows(A,[1 7])
```

```
D = 6×7

    76    79    91     0    19    41     1
    76    61    93    81    27    46    83
    95     7    73     5    19    44    20
    95     7    73    89    20    74    52
    95     7    40    35    60    93    67
    95    45    92    41    13     1    84
```

Sort the rows of A in descending order based on the elements in the fourth column, and display the output vector `index` to see how the rows were rearranged.

```
[E,index] = sortrows(A,4,'descend')
```

```
E = 6×7

    95     7    73    89    20    74    52
    76    61    93    81    27    46    83
```

```
    95    45    92    41    13     1    84
    95     7    40    35    60    93    67
    95     7    73     5    19    44    20
    76    79    91     0    19    41     1


index = 6×1

     2
     5
     1
     4
     3
     6
```

Complex Matrix

Create a matrix containing complex numbers, and sort the rows of the matrix in ascending order based on the elements in the first column. Since the magnitudes of `A(1,1)` and `A(3,1)` are equal, `sortrows` computes their angles to break the tie.

```
A = [1+2i 3+i i; 2+10i 6i 2+5i; 2+i 4 3+3i]
```

```
A = 3×3 complex

   1.0000 + 2.0000i   3.0000 + 1.0000i   0.0000 + 1.0000i
   2.0000 +10.0000i   0.0000 + 6.0000i   2.0000 + 5.0000i
   2.0000 + 1.0000i   4.0000 + 0.0000i   3.0000 + 3.0000i
```

```
B = sortrows(A)
```

```
B = 3×3 complex

   2.0000 + 1.0000i   4.0000 + 0.0000i   3.0000 + 3.0000i
   1.0000 + 2.0000i   3.0000 + 1.0000i   0.0000 + 1.0000i
   2.0000 +10.0000i   0.0000 + 6.0000i   2.0000 + 5.0000i
```

```
angle(A(1,1))
```

```
ans = 1.1071
```

```
angle(A(3,1))
```

```
ans = 0.4636
```

Use the `'real'` option to sort the rows of `A` by their real part. Since `A(2,1)` and `A(3,1)` have equal real parts, `sortrows` uses the imaginary part to break the tie.

```
C = sortrows(A,'ComparisonMethod','real')
```

```
C = 3×3 complex

   1.0000 + 2.0000i   3.0000 + 1.0000i   0.0000 + 1.0000i
   2.0000 + 1.0000i   4.0000 + 0.0000i   3.0000 + 3.0000i
   2.0000 +10.0000i   0.0000 + 6.0000i   2.0000 + 5.0000i
```

```
imag(A(2,1))
```

```
ans = 10
```

```
imag(A(3,1))
```

```
ans = 1
```

Sort Rows of Cell Array

Create a 6-by-2 cell array of character vectors, and sort its rows. The result is an alphabetized list sorted by both country and name.

```
A = {'Germany' 'Lukas'; 'USA' 'William'; 'USA' 'Andrew'; ...
'Germany' 'Andreas'; 'USA' 'Olivia'; 'Germany' 'Julia'}
```

```
A = 6x2 cell array
    {'Germany'}    {'Lukas'  }
    {'USA'    }    {'William'}
    {'USA'    }    {'Andrew' }
    {'Germany'}    {'Andreas'}
    {'USA'    }    {'Olivia' }
    {'Germany'}    {'Julia'  }
```

```
B = sortrows(A)
```

```
B = 6x2 cell array
    {'Germany'}    {'Andreas'}
    {'Germany'}    {'Julia'  }
    {'Germany'}    {'Lukas'  }
    {'USA'    }    {'Andrew' }
    {'USA'    }    {'Olivia' }
    {'USA'    }    {'William'}
```

Sort the countries first, then sort the names in descending order.

```
C = sortrows(A,[1 2],{'ascend' 'descend'})
```

```
C = 6x2 cell array
    {'Germany'}    {'Lukas'  }
    {'Germany'}    {'Julia'  }
    {'Germany'}    {'Andreas'}
    {'USA'    }    {'William'}
    {'USA'    }    {'Olivia' }
    {'USA'    }    {'Andrew' }
```

Sort Rows of Table

Sort the rows of a table by variable values.

Create a table with four variables listing patient information for five people.

```
LastName = {'Smith';'Johnson';'Williams';'Jones';'Brown'};
Age = [38;43;38;40;49];
Height = [71;69;64;67;64];
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];

tblA = table(Age,Height,Weight,BloodPressure,'RowNames',LastName)
```

```
tblA=5×4 table
                Age    Height    Weight    BloodPressure
                ___    ______    ______    _____________

    Smith       38       71       176       124     93
    Johnson     43       69       163       109     77
```

```
    Williams    38       64       131       125     83
    Jones       40       67       133       117     75
    Brown       49       64       119       122     80
```

Sort the rows of the table. The `sortrows` function sorts the rows in ascending order first by the variable `Age`, and then by the variable `Height` to break the tie between the two rows with equal ages.

```
tblB = sortrows(tblA)
```

```
tblB=5×4 table
                Age    Height    Weight    BloodPressure
                ___    ______    ______    _____________

    Williams    38       64       131       125     83
    Smith       38       71       176       124     93
    Jones       40       67       133       117     75
    Johnson     43       69       163       109     77
    Brown       49       64       119       122     80
```

Sort Rows of Table by Row Names

Create a table with four variables listing patient information for five people.

```
LastName = {'Smith';'Johnson';'Williams';'Jones';'Brown'};
Age = [38;43;38;40;49];
Height = [71;69;64;67;64];
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];

tblA = table(Age,Height,Weight,BloodPressure,'RowNames',LastName)
```

```
tblA=5×4 table
                Age    Height    Weight    BloodPressure
                ___    ______    ______    _____________

    Smith       38       71       176       124     93
    Johnson     43       69       163       109     77
    Williams    38       64       131       125     83
    Jones       40       67       133       117     75
```

```
    Brown       49       64       119       122     80
```

Sort the rows of the table in ascending order based on the row names, and return the index vector that describes how the rows were rearranged.

```
[tblB,index] = sortrows(tblA,'RowNames')
```

```
tblB=5×4 table
                Age    Height    Weight    BloodPressure
                ___    ______    ______    _____________

    Brown       49       64       119       122     80
    Johnson     43       69       163       109     77
    Jones       40       67       133       117     75
    Smith       38       71       176       124     93
    Williams    38       64       131       125     83


index = 5×1

     5
     2
     4
     1
     3
```

Sort Rows of Table by Variables

Create a table with four variables listing patient information for five people.

```
LastName = {'Sweet';'Jacobson';'Wang';'Joiner';'Berger'};
Age = [38;43;38;40;49];
Height = [71;69;64;67;64];
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];

tblA = table(Age,Height,Weight,BloodPressure,'RowNames',LastName)
```

```
tblA=5×4 table
                Age    Height    Weight    BloodPressure
                ___    ______    ______    _____________
```

```
    Sweet       38       71       176       124     93
    Jacobson    43       69       163       109     77
    Wang        38       64       131       125     83
    Joiner      40       67       133       117     75
    Berger      49       64       119       122     80
```

Sort the rows of the table in ascending order by `Height`, and then in descending order by `Weight`.

```
tblB = sortrows(tblA,{'Height','Weight'},{'ascend','descend'})
```

```
tblB=5×4 table
                Age    Height    Weight    BloodPressure
                ___    ______    ______    _____________

    Wang        38       64       131       125     83
    Berger      49       64       119       122     80
    Joiner      40       67       133       117     75
    Jacobson    43       69       163       109     77
    Sweet       38       71       176       124     93
```

Table with Missing Elements

Create a table with four variables listing patient information for five people. The `Weight` variable contains missing values.

```
LastName = {'Sweet';'Jacobson';'Wang';'Joiner';'Berger'};
Age = [38;43;38;40;49];
Height = [71;69;64;67;64];
Weight = [176;NaN;131;133;NaN];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];
tblA = table(Age,Height,Weight,BloodPressure,'RowNames',LastName)
```

```
tblA=5×4 table
                Age    Height    Weight    BloodPressure
                ___    ______    ______    _____________

    Sweet       38       71       176       124     93
    Jacobson    43       69       NaN       109     77
```

```
    Wang        38       64       131       125     83
    Joiner      40       67       133       117     75
    Berger      49       64       NaN       122     80
```

Sort the rows of the table in ascending order by `Weight`, placing the rows containing `NaN` first.

```
tblB = sortrows(tblA,'Weight','MissingPlacement','first')
```

```
tblB=5×4 table
                Age    Height    Weight    BloodPressure
                ___    ______    ______    _____________

    Jacobson    43       69       NaN       109     77
    Berger      49       64       NaN       122     80
    Wang        38       64       131       125     83
    Joiner      40       67       133       117     75
    Sweet       38       71       176       124     93
```

Sort Rows of Timetable

Create a timetable, and sort the rows by row times.

```
TimeDuration = [hours(3) hours(2) hours(1) hours(5) hours(6)]';
TT = timetable(TimeDuration,[98;97.5;97.9;98.1;101],[120;111;119;117;118]);

B = sortrows(TT,'TimeDuration')
```

```
B=5×3 timetable
    TimeDuration    Var1    Var2
    ____________    ____    ____

    1 hr            97.9    119
    2 hr            97.5    111
    3 hr              98    120
    5 hr            98.1    117
    6 hr             101    118
```

Input Arguments

A — Input array
column vector | matrix

Input array, specified as a column vector or matrix.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `cell` | `categorical` | `datetime` | `duration`
Complex Number Support: Yes

`column` — Column sorting vector
nonzero integer scalar | vector of nonzero integers

Column sorting vector, specified as a nonzero integer scalar or a vector of nonzero integers. Each specified integer value indicates a column to sort by. Negative integers indicate that the sort order is descending.

`direction` — Sorting direction
character vector | cell array of character vectors

Sorting direction, specified as a character vector or cell array of character vectors containing `'ascend'` (default) or `'descend'`. If `direction` is a cell array of character vectors, then the number of entries must match the number of columns or variables being sorted on.

If the `column` argument and the `direction` argument are specified together, then `sortrows` sorts according to `direction`, ignoring the signs of the elements in `column`.

Data Types: `char` | `cell`

`tblA` — Input table
table | timetable

Input table, specified as a table or a timetable. Each variable in `tblA` must be a valid input to `sort` or `sortrows`.

Data Types: `table` | `timetable`

`rowDimName` — Name of first dimension of input table or timetable
character vector

Name of the first dimension of the input table or timetable, specified as a character vector.

- If `tblA` is a table with row names, then `rowDimName` is the name of the first dimension of the table. By default, the name of the first dimension is `'Row'`. Dimension names are a property of tables. You can access the dimension names of `tblA` using `tblA.Properties.DimensionNames`.
- If `tblA` is a timetable, then `rowDimName` is the name of the vector of row times. You can specify its name when you create a timetable, such as `Time` or `Date`. You can also access the dimension names using `tblA.Properties.DimensionNames`.

Example: If a table `T` has row names, and you changed the name of the first dimension using `T.Properties.DimensionName{1} = 'Name'`, then `sortrows(T,'Name')` sorts the table by row name.

Example: If a timetable `TT` has a time vector named `Date`, then `sortrows(TT,'Date')` sorts the timetable on the dates and times that `Date` contains.

Data Types: `char`

`vars` — Sorting variables

scalar integer | vector of integers | variable name | cell array of variable names | logical vector

Sorting variables, specified as a scalar integer, a vector of integers, a variable name, a cell array of variable names, or a logical vector. `vars` indicates the table variables to sort by.

If an element of `vars` is a positive integer, then `sortrows` sorts the corresponding variable in `tblA` in ascending order. If an element of `vars` is a negative integer, then `sortrows` sorts the corresponding variable in `tblA` in descending order.

Example: `sortrows(tblA,{'Height','Weight'})` sorts the rows of `tblA` in ascending order, first by the variable `Height`, then by the variable `Weight` to break ties.

Example: `sortrows(tblA,[1 4])` sorts by the first variable of `tblA` in ascending order, then sorts by the fourth variable to break ties.

Example: `sortrows(TT,{'Time','X'})` sorts the row times of timetable `TT` in ascending order first, then sorts by the table variable `'X'` to break ties.

Data Types: `double` | `single` | `char` | `cell` | `logical`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `sortrows(A,'MissingPlacement','last')`

`MissingPlacement` — Placement of missing values
`'auto'` (default) | `'first'` | `'last'`

Placement of missing values (`NaN`, `NaT`, `<undefined>`, and `missing`) specified as the comma-separated pair consisting of `'MissingPlacement'` and one of the following:

- `'auto'` — Missing elements are placed last for ascending order and first for descending order.
- `'first'` — Missing elements are placed first.
- `'last'` — Missing elements are placed last.

`ComparisonMethod` — Element comparison method
`'auto'` (default) | `'real'` | `'abs'`

Element comparison method, specified as the comma-separated pair consisting of `'ComparisonMethod'` and one of the following:

- `'auto'` — Sort rows of `A` by `real(A)` when `A` is real, and sort by `abs(A)` when `A` is complex.
- `'real'` — Sort rows of `A` by `real(A)` when `A` is real or complex. If a column of `A` has elements with equal real parts, then use `imag(A)` to break ties.
- `'abs'` — Sort rows of `A` by `abs(A)` when `A` is real or complex. If a column of `A` has elements with equal magnitude, then use `angle(A)` in the interval (-π,π] to break ties.

Output Arguments

`B` — Sorted array
vector | matrix | multidimensional array

Sorted array, returned as a vector, matrix, or multidimensional array. `B` is the same size as `A`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `cell` | `categorical` | `datetime` | `duration`

`tblB` — Sorted table
table | timetable

Sorted table, returned as a table or timetable with the same variables as `tblA`.

Data Types: `table` | `timetable`

`index` — Sort index
index vector

Sort index, returned as an index vector. The sort index describes the rearrangement of the rows in the input such that `B = A(index,:)`.

Data Types: `double`

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

- Sorting by row names is not supported.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The first input argument must not be a cell array.

- If `A` is complex with all zero imaginary parts, then MATLAB might convert `A` to `real(A)` before calling `sortrows(A)`. In this case, MATLAB sorts the rows of `A` by `real(A)`, but the generated code sorts the rows of `A` by `abs(A)`. To make the generated code match MATLAB, use `sortrows(real(A))` or `sortrows(A,'ComparisonMethod','real')`. See "Code Generation for Complex Data with Zero-Valued Imaginary Parts" (MATLAB Coder).

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- The first input argument must not be a cell array.
- If `A` is complex with all zero imaginary parts, then MATLAB might convert `A` to `real(A)` before calling `sortrows(A)`. In this case, MATLAB sorts the rows of `A` by `real(A)`, but the generated code sorts the rows of `A` by `abs(A)`. To make the generated code match MATLAB, use `sortrows(real(A))` or `sortrows(A,'ComparisonMethod','real')`.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Sorting cell arrays is not supported.
- Sparse inputs are not supported.

For more information, see "Run MATLAB Functions on a GPU" (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- Sorting cell arrays is not supported.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`issorted` | `issortedrows` | `sort` | `topkrows`

Introduced before R2006a

sortx

Sort elements in heatmap row

Syntax

```
sortx(h,row)
sortx(h,row,direction)
sortx( ___ ,'MissingPlacement',lcn)
sortx(h)

C = sortx( ___ )
[C,x] = sortx( ___ )
```

Description

`sortx(h,row)` displays the elements in `row` in ascending order (from left to right). This function sorts the row elements by rearranging the columns. Specify `row` as one element or as an array of elements from `YData`. If the first row that you specify has repeating values, then `sortx` rearranges the remaining columns according to the second row, and so on.

`sortx(h,row,direction)`, where `direction` is `'descend'`, sorts the values in descending order instead of ascending order (default). To sort in a unique direction for each row in `row`, specify `direction` as an array whose elements are `'ascend'` or `'descend'`.

`sortx( ___ ,'MissingPlacement',lcn)` specifies whether to put `NaN` elements first or last in the sort order. Specify `lcn` as `'first'`, `'last'`, or `'auto'`. The default setting is `'auto'`, which lists `NaN` elements last for ascending order and first for descending order. Use this option after any of the previous input argument combinations.

`sortx(h)` displays the elements in the top row in ascending order. If the first row has repeating elements, then `sortx` rearranges the remaining columns according to the second row, and so on.

`C = sortx( ___ )` returns the sorted color data, which is a matrix of values as they appear in the heatmap chart.

`[C,x] = sortx( ___ )` also returns the sorted *x* values in the order they are displayed.

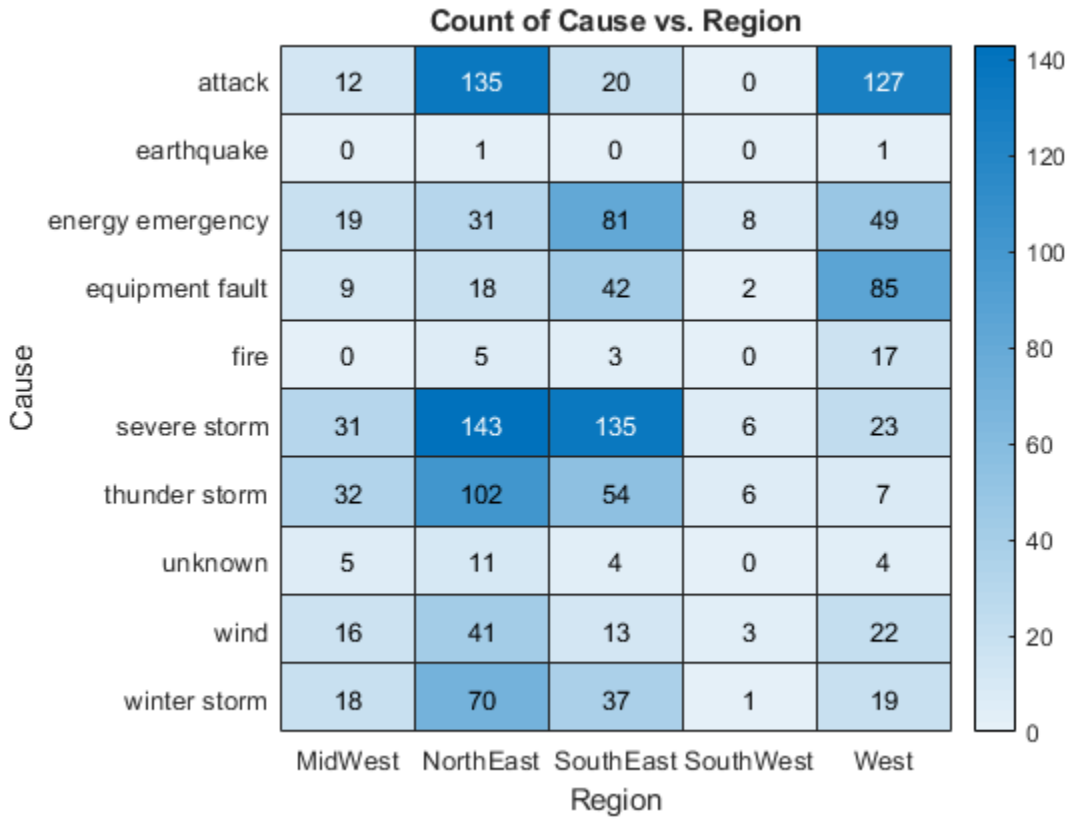
Examples

Display Heatmap Row in Ascending Order

Create a heatmap of utility outages and sort the values in a particular row so that they appear in ascending order, from left to right.

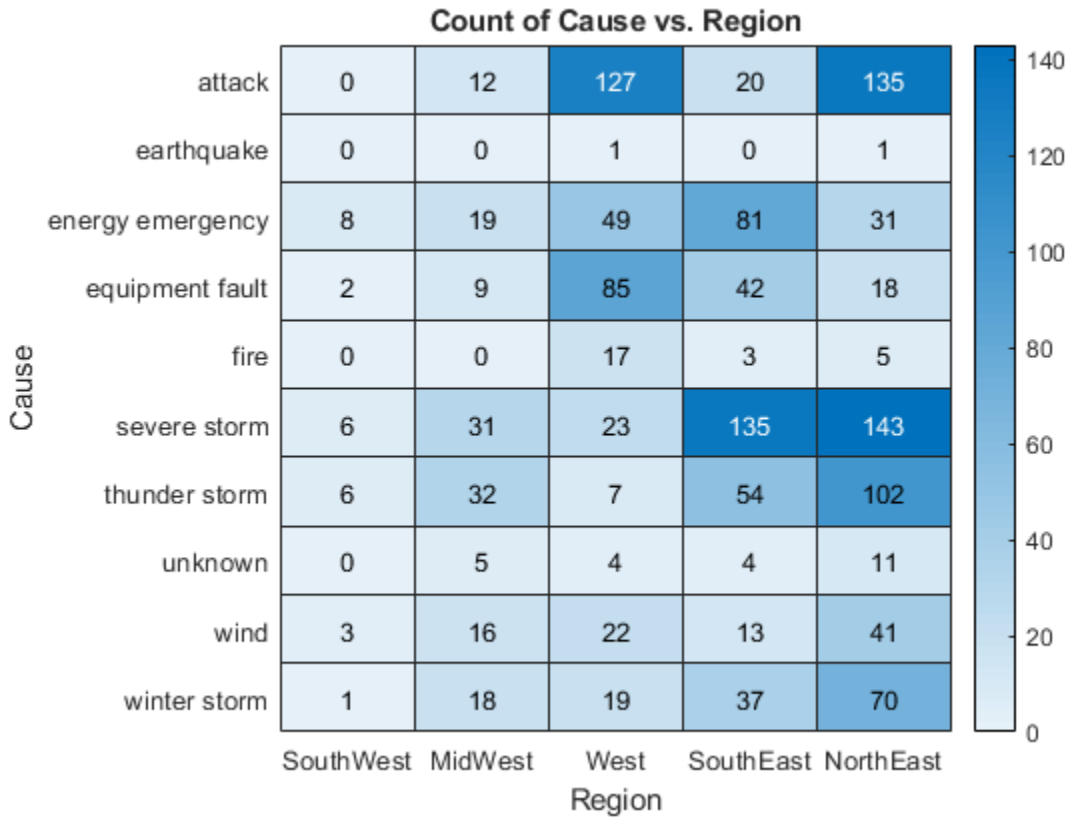
First, create a heatmap. Read the sample file `outages.csv` into a table. The sample file contains data representing electric utility outages in the United States. The table contains six columns: `Region`, `OutageTime`, `Loss`, `Customers`, `RestorationTime`, and `Cause`. Create a heatmap that shows the different regions along the *x*-axis and the different outage causes along the *y*-axis. In each cell, show how many times each region experienced a power outage due to a particular cause.

```
T = readtable('outages.csv');
h = heatmap(T,'Region','Cause');
```



Sort the values in the `'winter storm'` row so that they appear in ascending order from left to right.

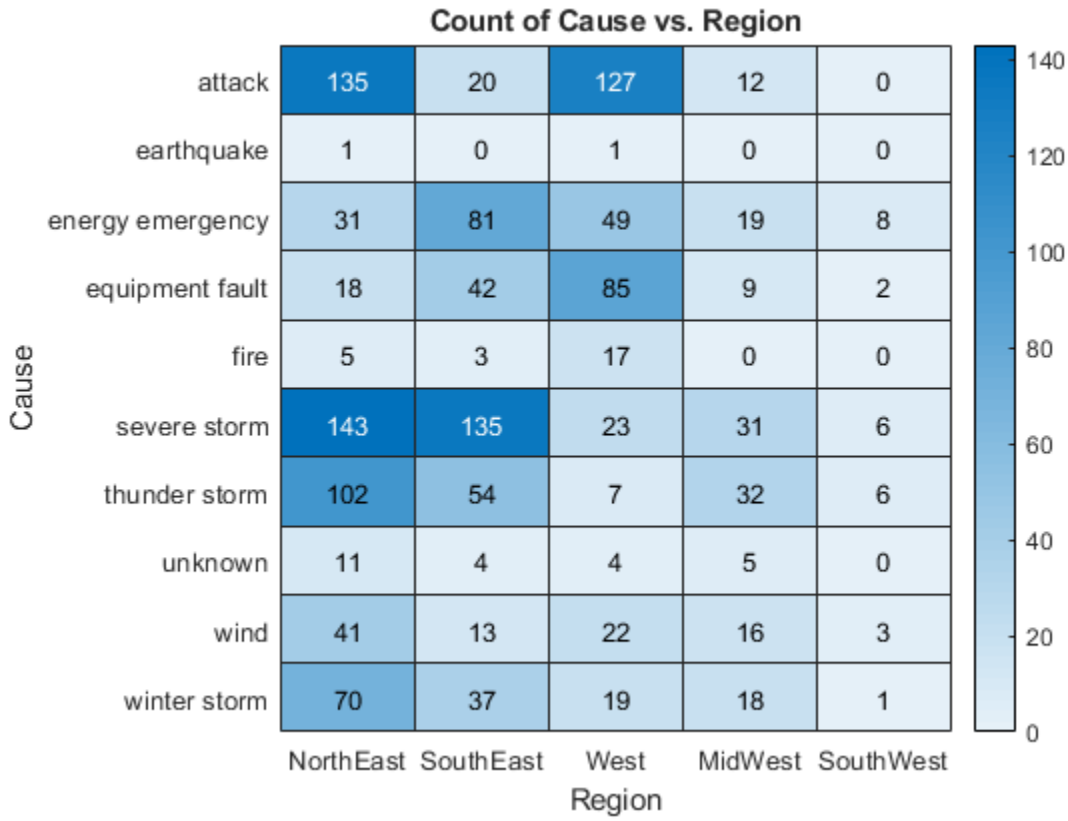
```
sortx(h,'winter storm')
```



Display Heatmap Row in Descending Order

Create a heatmap of utility outages and display the values in the `'winter storm'` row in descending order by rearranging the columns.

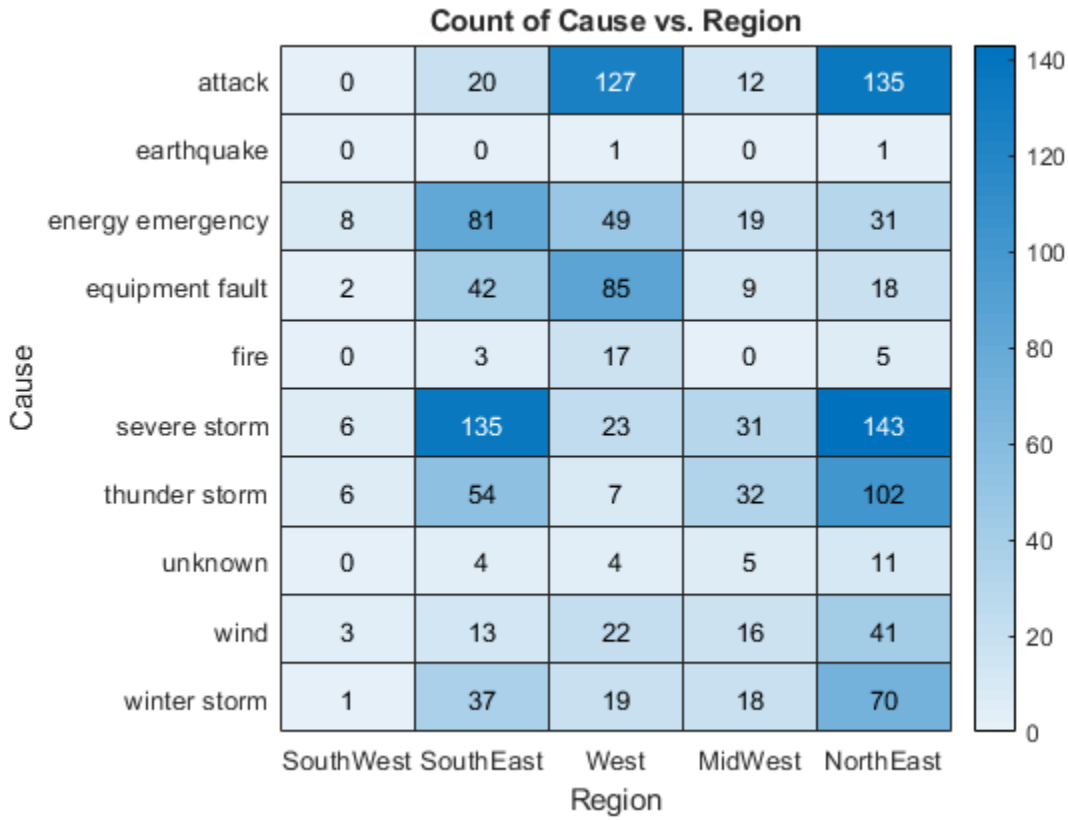
```
T = readtable('outages.csv');
h = heatmap(T,'Region','Cause');
sortx(h,'winter storm','descend')
```



Rearrange Heatmap Columns Based on Multiple Rows

Create a heatmap of utility outages and rearrange the columns using values from multiple rows. Sort the columns so that the row `'unknown'` appears in ascending order. Since that row contains duplicate values, use the row `'earthquake'` to sort the remaining columns.

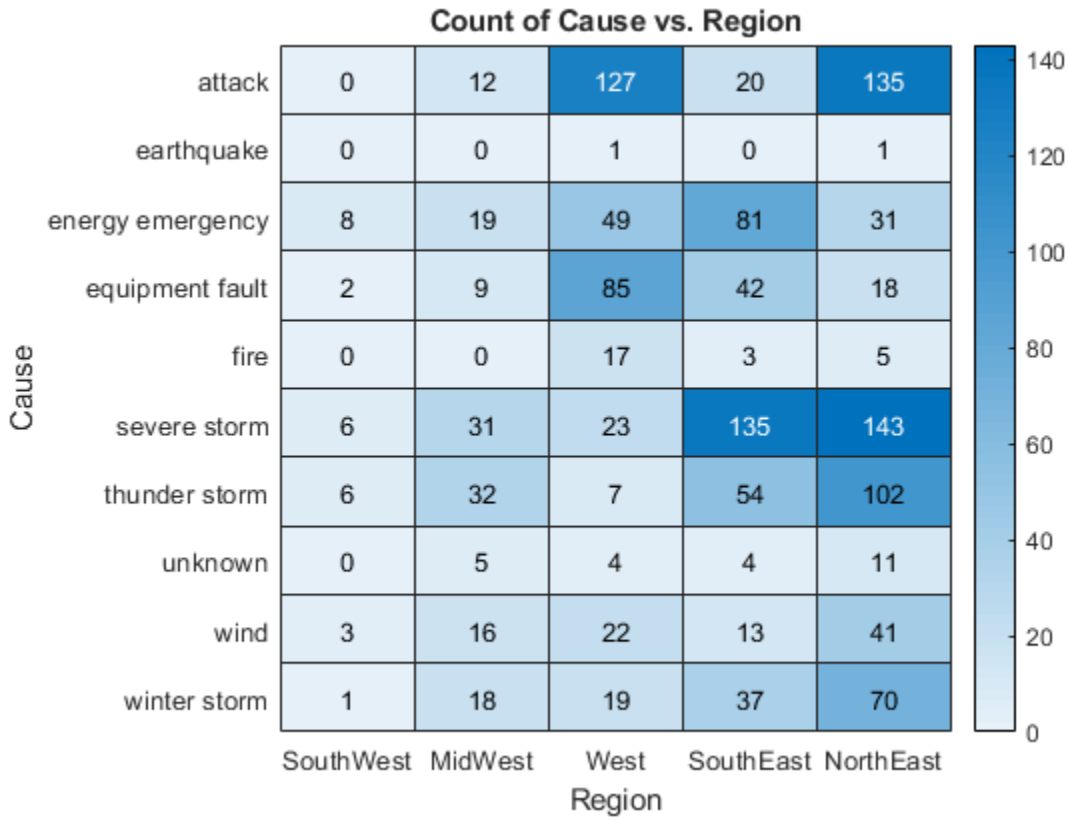
```
T = readtable('outages.csv');
h = heatmap(T,'Region','Cause');
sortx(h,{'unknown','earthquake'})
```



Return Rearranged Heatmap Column Values

Create a heatmap of utility outages and sort the values in the `'winter storm'` row so that they appear in ascending order. Return the sorted color data and the column values (*x* values) in the order that they appear along the *x*-axis.

```
T = readtable('outages.csv');
h = heatmap(T,'Region','Cause');
[C,x] = sortx(h,'winter storm')
```



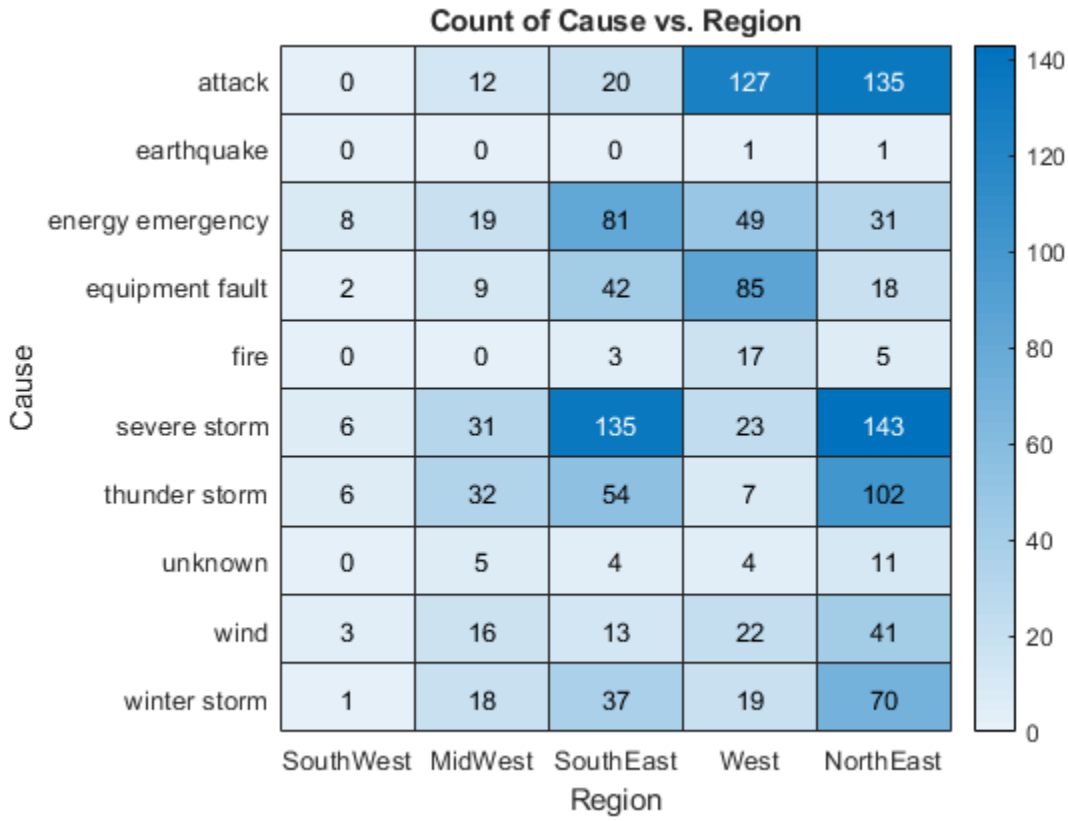

```
C = 10×5

     0    12   127    20   135
     0     0     1     0     1
     8    19    49    81    31
     2     9    85    42    18
     0     0    17     3     5
     6    31    23   135   143
     6    32     7    54   102
     0     5     4     4    11
     3    16    22    13    41
     1    18    19    37    70
```

```
x = 5x1 cell array
    {'SouthWest'}
    {'MidWest'  }
    {'West'     }
    {'SouthEast'}
    {'NorthEast'}
```

Restore Original Heatmap Column Order

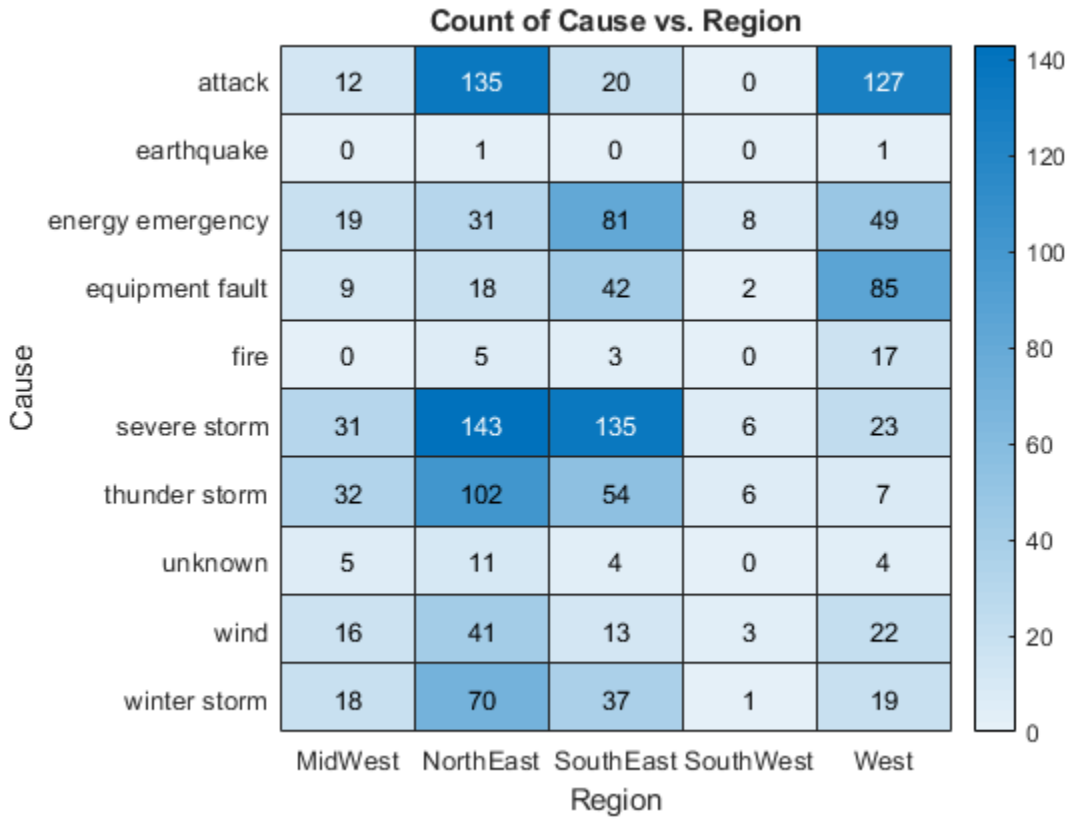
Create a heatmap of utility outages and sort the values in the top row in ascending order by rearranging the columns. Then, restore the original order.

```
T = readtable('outages.csv');
h = heatmap(T,'Region','Cause');
sortx(h)
```



Restore the original column order by setting the `XDisplayData` property of the `HeatmapChart` object equal to the `XData` property.

```
h.XDisplayData = h.XData;
```



Input Arguments

h — Heatmap
`HeatmapChart` object

Heatmap to update, specified as a `HeatmapChart` object.

`row` — Row to sort
character vector | cell array of character vectors | string array

Row to sort, specified as a character vector, cell array of character vectors, or string array indicating one or more elements from the `YData` or `YDisplayData` property of the `HeatmapChart` object.

Example: `sortx(h,'MyRowName')` sorts the elements in the row called `'MyRowName'` in ascending order.

`direction` — Direction to sort
`'ascend'` (default) | `'descend'` | cell array of character vectors | string array

Direction to sort, specified as one of these values:

- `'ascend'` — Sort all rows in `row` in ascending order.
- `'descend'` — Sort all rows in `row` in descending order.
- Array — Sort in a unique direction for each row in `row`. Specify `direction` as a cell array of character vectors or a string array containing the elements `'ascend'` or `'descend'`. The array must be the same length as `row`.

Example: `sortx(h,row,{'ascend','descend','ascend'})` sorts the elements in the first row in `row` in ascending order. If there are any repeating values in the first row, then it breaks ties using the elements of the second row in descending order, and so on.

`lcn` — `NaN` placement
`'auto'` (default) | `'first'` | `'last'`

`NaN` placement, specified as one of these values:

- `'auto'` — List `NaN` elements last for ascending order and first for descending order.
- `'first'` — List `NaN` elements first.
- `'last'` — List `NaN` elements last.

Example: `sortx(h,row,'MissingPlacement','last')`

Output Arguments

`C` — Sorted color data
matrix

Sorted color data, returned as a matrix. The values in the matrix appear in the same order as they appear on the heatmap.

The `ColorDisplayData` property of the `HeatmapChart` object stores the sorted color data. The `ColorData` property stores the unsorted color data.

x — Rearranged column order
cell array of character vectors

Rearranged column order, returned as a cell array of character vectors.

The `XDisplayData` property of the `HeatmapChart` object stores the sorted column order. The `XData` property stores the unsorted column order.

Algorithms

The `sortx` function sets and queries these `HeatmapChart` object properties:

- `XDisplayData` — Property that stores the *x*-axis data.
- `ColorDisplayData` — Property that stores the color display data.

The `sortx` function also resets the `XLimits` property.

See Also

Functions
`heatmap` | `sorty`

Properties
HeatmapChart

Introduced in R2017b

sorty

Sort elements in heatmap column

Syntax

```
sorty(h,column)
sorty(h,column,direction)
sorty( ___ ,'MissingPlacement',lcn)
sorty(h)

C = sorty( ___ )
[C,y] = sorty( ___ )
```

Description

`sorty(h,column)` displays the elements in `column` in ascending order (from top to bottom). This function sorts the column elements by rearranging the rows. Specify `column` as one element or as an array of elements from `XData`. If the first column that you specify has repeating values, then `sorty` rearranges the remaining rows according to the second column, and so on.

`sorty(h,column,direction)`, where `direction` is `'descend'`, sorts the values in descending order instead of ascending order (default). To sort in a unique direction for each column in `column`, specify `direction` as an array whose elements are `'ascend'` or `'descend'`.

`sorty( ___ ,'MissingPlacement',lcn)` specifies whether to put `NaN` elements first or last in the sort order. Specify `lcn` as `'first'`, `'last'`, or `'auto'`. The default setting is `'auto'`, which lists `NaN` elements last for ascending order and first for descending order. Use this option after any of the previous input argument combinations.

`sorty(h)` displays the elements in the first column in ascending order. If the first column has repeating elements, then `sorty` rearranges the remaining rows according to the second column, and so on.

`C = sorty( ___ )` returns the sorted color data, which is a matrix of values as they appear in the heatmap chart.

`[C,y] = sorty( ___ )` also returns the sorted *y* values in the order they are displayed.

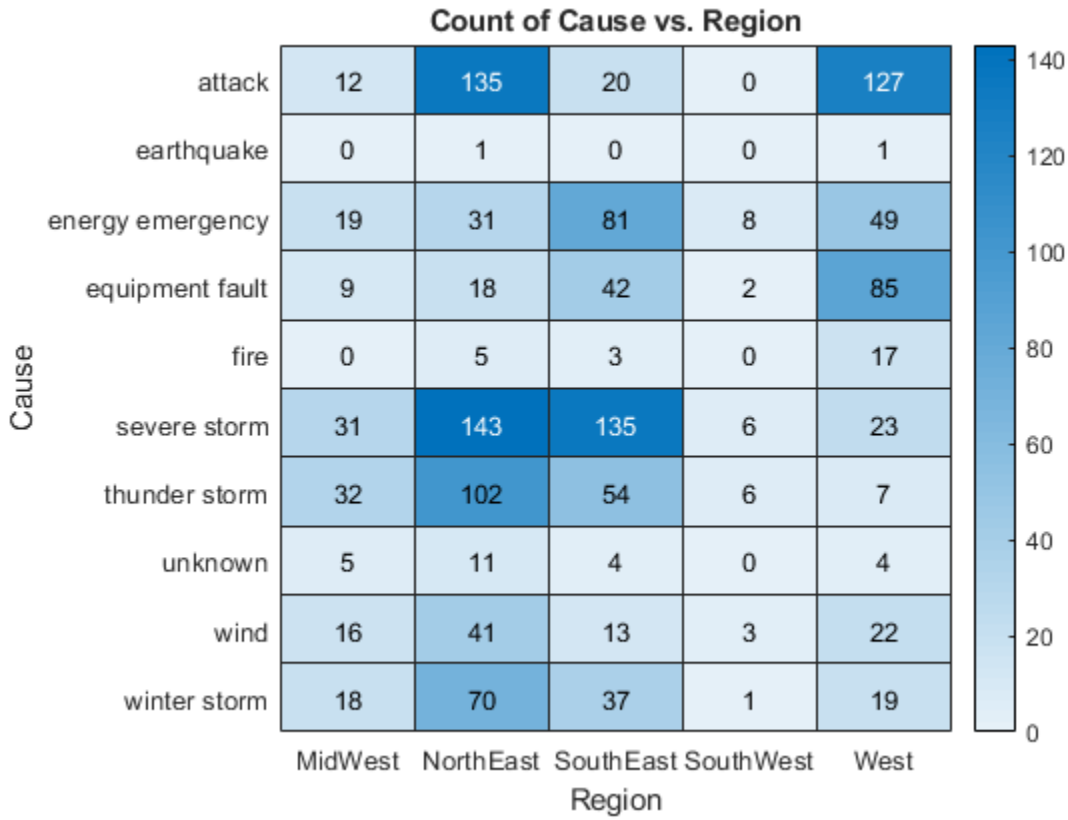
Examples

Display Heatmap Column in Ascending Order

Create a heatmap of utility outages and sort the values in a particular column so that they appear in ascending order, from top to bottom.

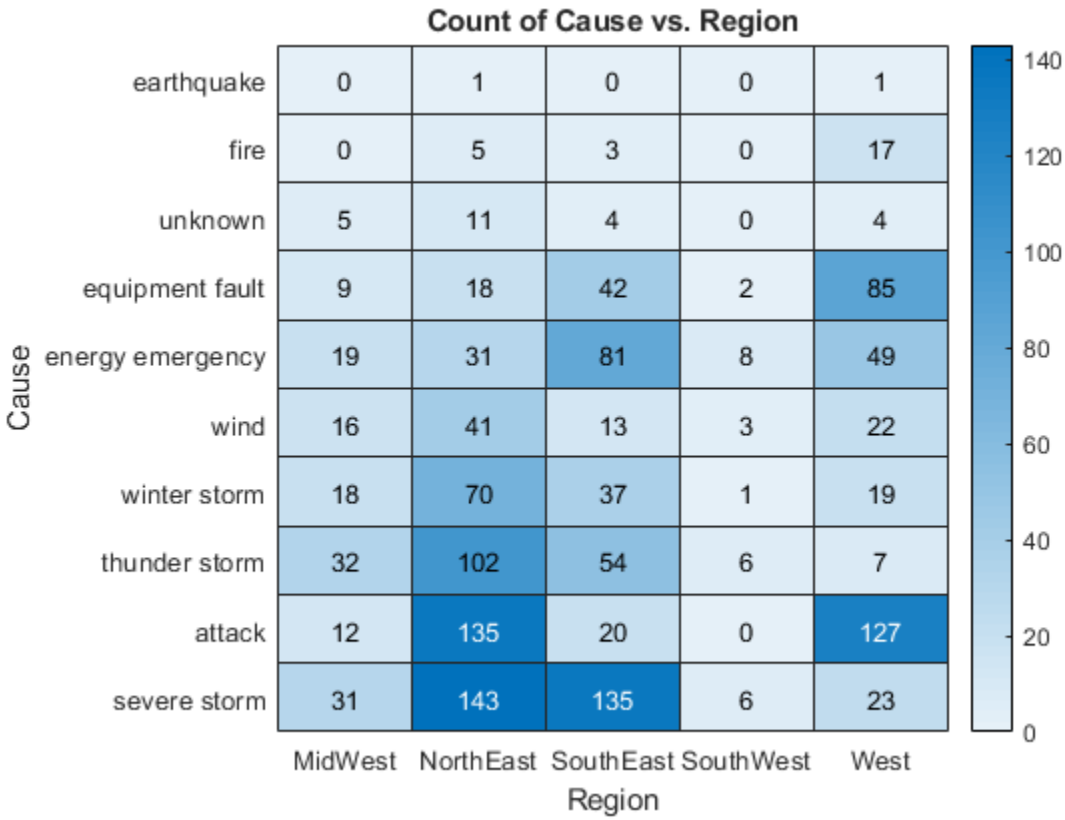
First, create a heatmap. Read the sample file `outages.csv` into a table. The sample file contains data representing electric utility outages in the United States. The table contains six columns: `Region`, `OutageTime`, `Loss`, `Customers`, `RestorationTime`, and `Cause`. Create a heatmap that shows the different regions along the *x*-axis and the different outage causes along the *y*-axis. In each cell, show how many times each region experienced a power outage due to a particular cause.

```
T = readtable('outages.csv');
h = heatmap(T,'Region','Cause');
```



Sort the values in the `'NorthEast'` column so that they appear in ascending order from top to bottom.

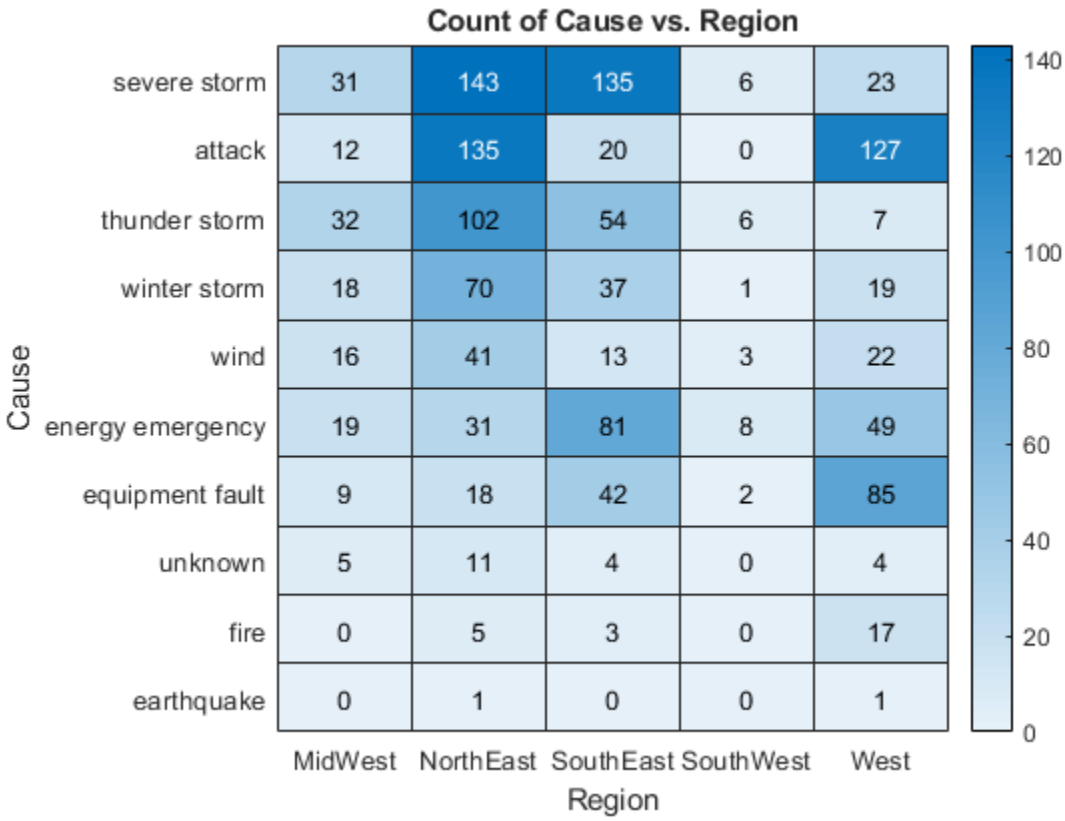
```
sorty(h,'NorthEast')
```



Display Heatmap Column in Descending Order

Create a heatmap of utility outages and display the values in the `'NorthEast'` column in descending order by rearranging the rows.

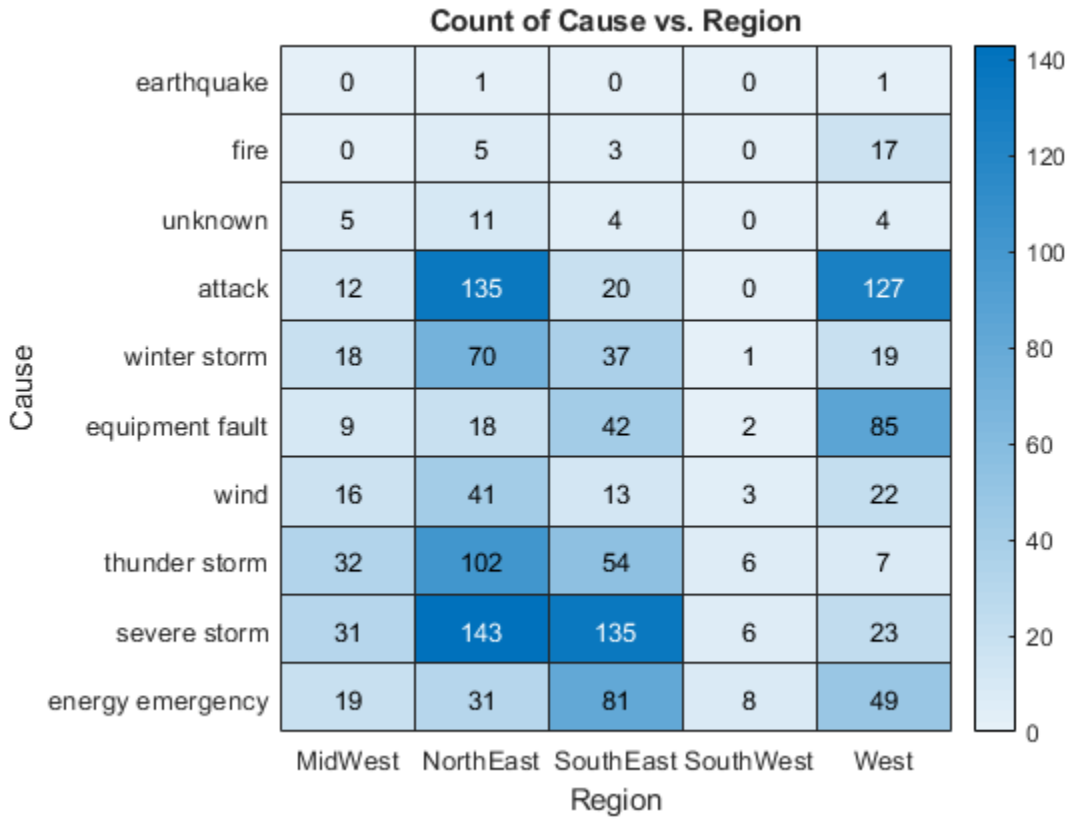
```
T = readtable('outages.csv');
h = heatmap(T,'Region','Cause');
sorty(h,'NorthEast','descend')
```



Rearrange Heatmap Rows Based on Multiple Columns

Create a heatmap of utility outages and rearrange the rows using values from multiple columns. Sort the rows so that the column `'SouthWest'` appears in ascending order. Since that column contains duplicate values, use the column `'NorthEast'` to sort the remaining rows.

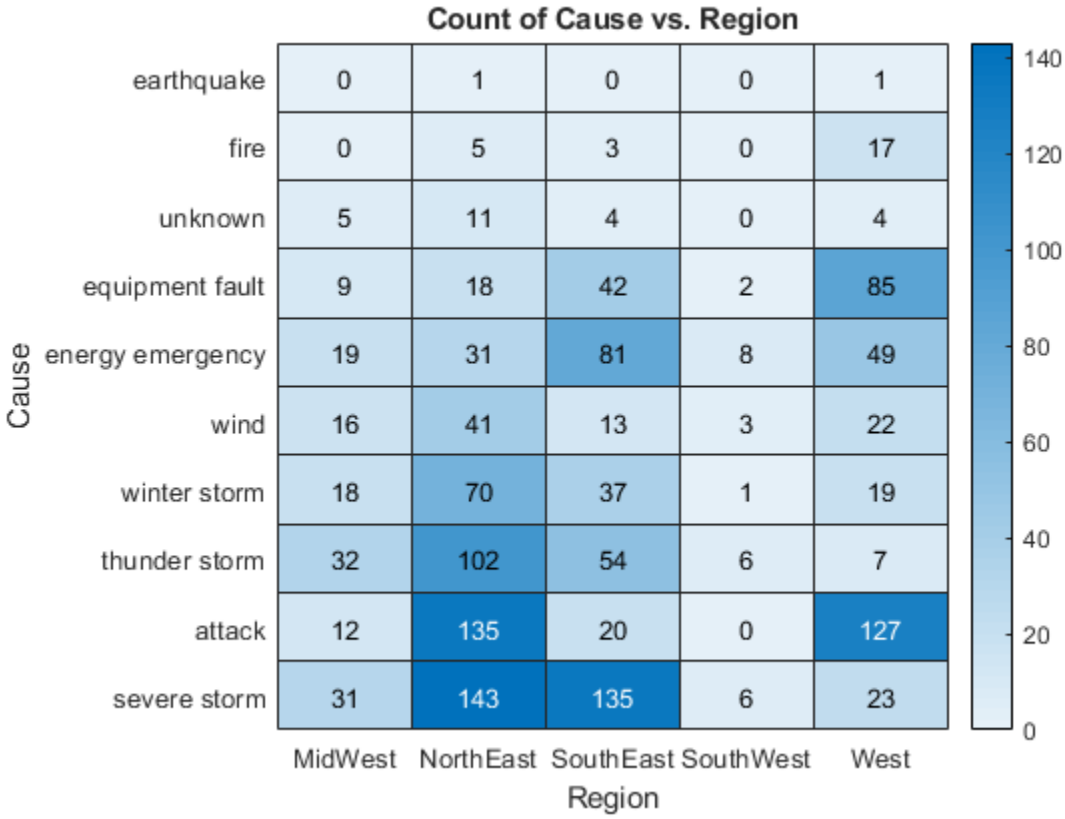
```
T = readtable('outages.csv');
h = heatmap(T,'Region','Cause');
sorty(h,{'SouthWest','NorthEast'})
```



Return Rearranged Heatmap Row Values

Create a heatmap of utility outages and sort the values in the `'NorthEast'` column so that they appear in ascending order, from top to bottom. Return the sorted color data and the row values (y values) in the order they appear along the y-axis.

```
T = readtable('outages.csv');
h = heatmap(T,'Region','Cause');
[C,y] = sorty(h,'NorthEast')
```



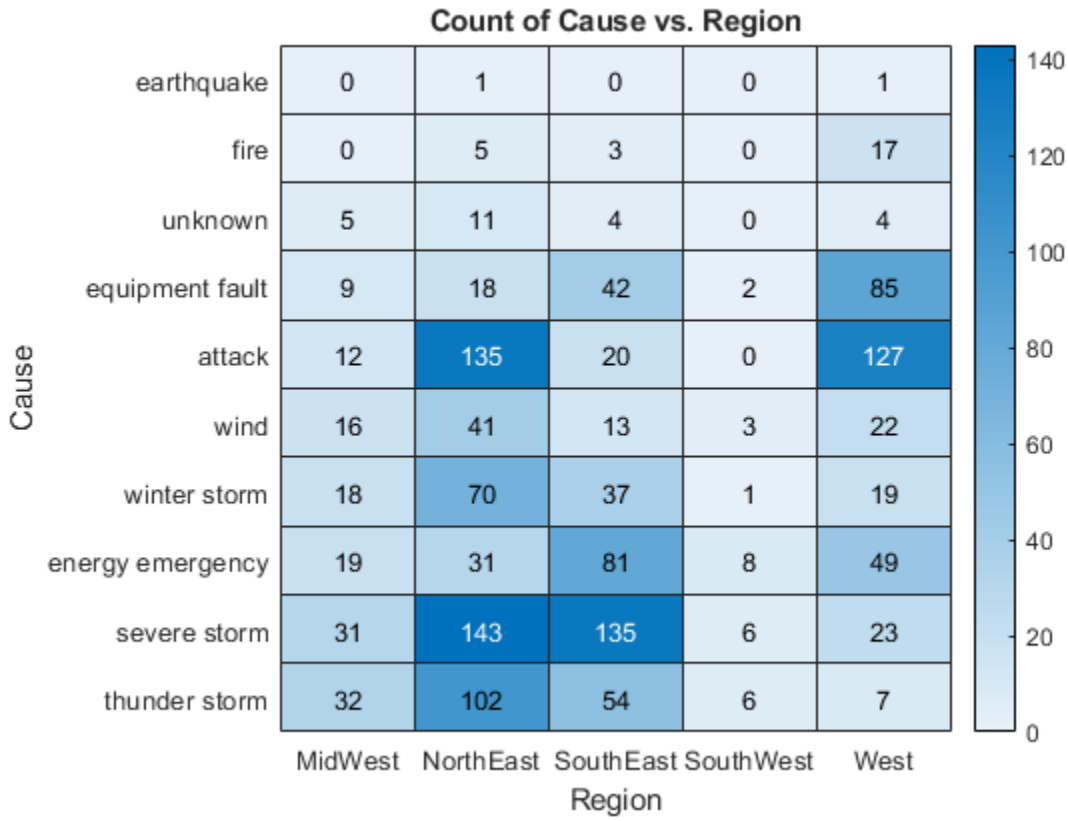

```
C = 10×5

     0     1     0     0     1
     0     5     3     0    17
     5    11     4     0     4
     9    18    42     2    85
    19    31    81     8    49
    16    41    13     3    22
    18    70    37     1    19
    32   102    54     6     7
    12   135    20     0   127
    31   143   135     6    23
```

```
y = 10x1 cell array
    {'earthquake'      }
    {'fire'            }
    {'unknown'         }
    {'equipment fault' }
    {'energy emergency'}
    {'wind'            }
    {'winter storm'    }
    {'thunder storm'   }
    {'attack'          }
    {'severe storm'    }
```

Restore Original Heatmap Row Order

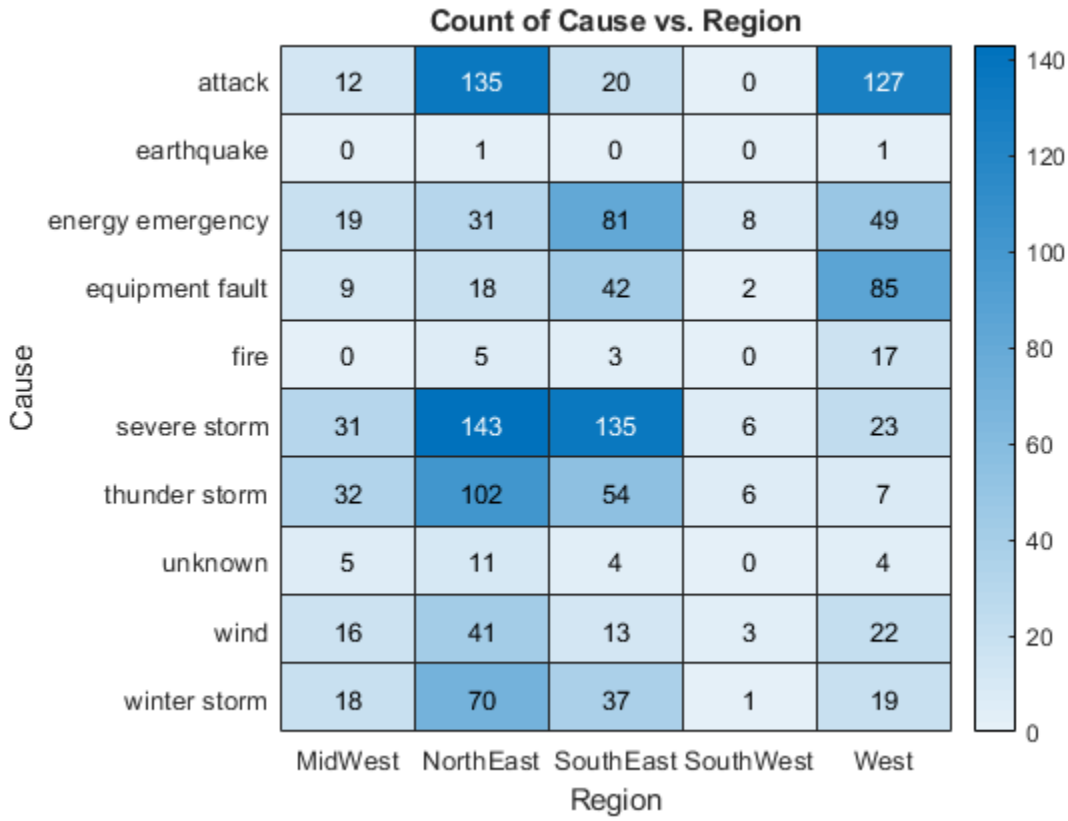
Create a heatmap of utility outages and sort the values in the left column in ascending order (from top to bottom) by rearranging the columns. Then, restore the original order.

```
T = readtable('outages.csv');
h = heatmap(T,'Region','Cause');
sorty(h)
```



Restore the original row order by setting the `YDisplayData` property of the `HeatmapChart` object equal to the `YData` property.

```
h.YDisplayData = h.YData;
```



Input Arguments

h — Heatmap
`HeatmapChart` object

Heatmap to update, specified as a `HeatmapChart` object.

`column` — Column to sort
character vector | cell array of character vectors | string array

Column to sort, specified as a character vector, cell array of character vectors, or string array indicating of one or more elements from the `XData` or `XDisplayData` property of the `HeatmapChart` object.

Example: `sorty(h,'MyColumnName')` sorts the elements in the column called `'MyColumnName'` in ascending order.

`direction` — Direction to sort
`'ascend'` (default) | `'descend'` | cell array of character vectors | string array

Direction to sort, specified as `'ascend'`, `'descend'`, a cell array of character vectors, or a string array.

- `'ascend'` — Sort all columns listed in `column` in ascending order.
- `'descend'` — Sort all columns listed in `column` in descending order.
- Array — Sort in a unique direction for each column in `column`. Specify `direction` as a cell array of character vectors or a string array containing the elements `'ascend'` or `'descend'`. The array must be the same length as `column`.

Example: `{'ascend','descend','ascend'}` sorts the elements in the first column in ascending order. If there are any repeating values in the first column, then `sorty` breaks ties using the elements of the second column in descending order, and so on.

`lcn` — `NaN` placement
`'auto'` (default) | `'first'` | `'last'`

`NaN` placement, specified as one of these values:

- `'auto'` — List `NaN` elements last for ascending order and first for descending order.
- `'first'` — List `NaN` elements first.
- `'last'` — List `NaN` elements last.

Example: `sortx(h,column,'MissingPlacement','last')`

Output Arguments

`C` — Sorted color data
matrix

Sorted color data, returned as a matrix. The values in the matrix appear in the same order as they appear on the heatmap.

The `ColorDisplayData` property of the `HeatmapChart` object stores the sorted color data. The `ColorData` property stores the unsorted color data.

y — Rearranged row order
cell array of character vectors

Rearranged row order, returned as a cell array of character vectors.

The `YDisplayData` property of the `HeatmapChart` object stores the sorted row order. The `YData` property stores the unsorted row order.

Algorithms

The `sorty` function sets these `HeatmapChart` object properties:

- `YDisplayData` — Property that stores the *y*-axis data.
- `ColorDisplayData` — Property that stores the color display data.

The `sorty` function also resets the `YLimits` property.

See Also

Functions
`heatmap` | `sortx`

Properties
HeatmapChart

Introduced in R2017b

sound

Convert matrix of signal data to sound

Syntax

```
sound(y)
sound(y,Fs)
sound(y,Fs,nBits)
```

Description

`sound(y)` sends audio signal `y` to the speaker at the default sample rate of 8192 hertz.

`sound(y,Fs)` sends audio signal `y` to the speaker at sample rate `Fs`.

`sound(y,Fs,nBits)` uses `nBits` bits per sample for audio signal `y`.

Examples

Play Sample Data at Default Sample Rate

Load the example file `gong.mat`, which contains sample data `y` and rate `Fs`, and listen to the audio.

```
load gong.mat;
sound(y);
```

Play Sample Data at Specific Sample Rate

Play an excerpt from Handel's “Hallelujah Chorus” at twice the recorded sample rate.

```
load handel.mat;
sound(y, 2*Fs);
```

Play Sample Data with Specific Bit Depth

```
load handel.mat;
nBits = 16;
sound(y,Fs,nBits);
```

MATLAB® plays the audio with a bit depth of 16 bits per sample, if this is supported on your system.

Input Arguments

y — Audio data
column vector | m-by-2 matrix

Audio data, specified as an m-by-1 column vector for single-channel (mono) audio, or an m-by-2 matrix for stereo playback, where m is the number of audio samples. If y is an m-by-2 matrix, then the first column corresponds to the left channel, and the second column corresponds to the right channel. Stereo playback is available only if your system supports it.

Data Types: double

Fs — Sample rate
8192 (default) | positive number

Sample rate, in hertz, of audio data y, is specified as a positive number from 1000 through 384000. Valid values depend on both the sample rates permitted by MATLAB and the specific audio hardware on your system. MATLAB has a hard restriction of 1000 Hz <= Fs <= 384000 Hz, although further hardware-dependent restrictions apply.

Data Types: single | double

nBits — Bit depth of sample values
16 (default) | 8 | 24

Bit depth of the sample values, specified as an integer. Valid values depend on the audio hardware installed. Most platforms support bit depths of 8 bits or 16 bits.

Tips

- The `sound` function supports sound devices on all Windows and most UNIX platforms.
- Most sound cards support sample rates between 5 and 48 kilohertz. Specifying a sample rate outside this range might produce unexpected results.

See Also

`audioplayer` | `audioread` | `audiowrite` | `soundsc`

Topics

"Play Audio"

Introduced before R2006a

soundsc

Scale data and play as sound

Syntax

```
soundsc(y)
soundsc(y,Fs)
soundsc(y,Fs,nBits)

soundsc( ___ ,yRange)
```

Description

`soundsc(y)` scales the values of audio signal `y` to fit in the range from –1.0 to 1.0, and then sends the data to the speaker at the default sample rate of 8192 hertz. By first scaling the data, `soundsc` plays the audio as loudly as possible without clipping. The mean of the dynamic range of the data is set to zero.

`soundsc(y,Fs)` sends audio signal `y` to the speaker at sample rate `Fs`.

`soundsc(y,Fs,nBits)` uses `nBits` bits per sample for audio signal `y`.

`soundsc( ___ ,yRange)`, where `yRange` is a vector of the form `[low,high]`, linearly scales the values in `y` between `low` and `high` to the full sound range `[-1.0,1.0]`. Values outside `[low,high]` scale beyond `[-1.0,1.0]`. You can use `yRange` with any of the input arguments in the previous syntaxes.

Examples

Play Sample Data at Default Sample Rate

Load the example file `gong.mat`, which contains sample data `y` and rate `Fs`, and listen to the audio.

```
load gong.mat;
soundsc(y);
```

Play Sample Data at Specific Sample Rate

Play an excerpt from Handel's "Hallelujah Chorus" at twice the recorded sample rate.

```
load handel.mat;
soundsc(y, 2*Fs);
```

Play Sample Data with Specific Bit Depth

```
load handel.mat;
nBits = 16;
soundsc(y,Fs,nBits);
```

MATLAB® plays the scaled audio with a bit depth of 16 bits per sample.

Scale Selected Audio Data

```
load handel.mat;
yRange = [-0.7,0.7];
soundsc(y,yRange);
```

Input Arguments

y — Audio data
column vector | m-by-2 matrix

Audio data, specified as an m-by-1 column vector for single-channel (mono) audio, or an m-by-2 matrix for stereo playback, where m is the number of audio samples. If y is an m-by-2 matrix, then the first column corresponds to the left channel, and the second column corresponds to the right channel. Stereo playback is available only if your system supports it.

Data Types: double

Fs — Sample rate
8192 (default) | positive number

Sample rate, in hertz, of audio data `y`, is specified as a positive number from 1000 through 384000. Valid values depend on both the sample rates permitted by MATLAB and the specific audio hardware on your system. MATLAB has a hard restriction of 1000 Hz <= Fs <= 384000 Hz, although further hardware-dependent restrictions apply.

Data Types: `single` | `double`

`nBits` — Bit depth of sample values
16 (default) | 8 | 24

Bit depth of the sample values, specified as an integer. Valid values depend on the audio hardware installed. Most platforms support bit depths of 8 bits or 16 bits.

yRange — Range of audio data to scale
`[-max(abs(y)),max(abs(y))]` (default) | two-element vector

Range of audio data to scale, specified as a two-element vector of the form `[low,high]`, where `low` and `high` are the lower and upper limits of the range. Values in `y` that are scaled beyond `[-1.0, 1.0]` are clipped when played back on a sound device.

Example: `[-0.8,0.8]`

Data Types: `double`

Tips

- The `sound` function supports sound devices on all Windows and most UNIX platforms.
- Most sound cards support sample rates between 5 and 48 kilohertz. Specifying a sample rate outside this range might produce unexpected results.

See Also

`audioplayer` | `audioread` | `audiowrite` | `sound`

Introduced before R2006a

spalloc

Allocate space for sparse matrix

Syntax

```
S = spalloc(m,n,nz)
```

Description

`S = spalloc(m,n,nz)` creates an all zero sparse matrix `S` of size `m`-by-`n` with room to hold `nz` nonzeros, where `nz >= 1`. The matrix can then be generated column by column without requiring repeated storage allocation as the number of nonzeros grows.

`spalloc(m,n,nz)` is shorthand for

```
sparse([],[],[],m,n,nz)
```

If you specify a value of `0` for `nz`, then `spalloc` instead sets the value of `nz` to `1`.

Examples

Use `spalloc` to initialize an `n`-by-`n` empty sparse matrix with space for `3*n` nonzeros. Then use a `for`-loop to fill in the columns of `S` one at a time. The result is a tridiagonal matrix with about 50% band density.

```
S = spalloc(n,n,3*n);
for j = 1:n
    ind = [max(j-1,1) j min(j+1,n)];
    S(:,j) = sparse(ind,1,round(rand(3,1)),n,1,3);
end
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `distributed.spalloc` in the Parallel Computing Toolbox documentation.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`nnz` | `nzmax` | `sparse`

Introduced before R2006a

sparse

Create sparse matrix

Syntax

```
S = sparse(A)

S = sparse(m,n)

S = sparse(i,j,v)
S = sparse(i,j,v,m,n)
S = sparse(i,j,v,m,n,nz)
```

Description

`S = sparse(A)` converts a full matrix into sparse form by squeezing out any zero elements. If a matrix contains many zeros, converting the matrix to sparse storage saves memory.

`S = sparse(m,n)` generates an `m`-by-`n` all zero sparse matrix.

`S = sparse(i,j,v)` generates a sparse matrix `S` from the triplets `i`, `j`, and `v` such that `S(i(k),j(k)) = v(k)`. The `max(i)`-by-`max(j)` output matrix has space allotted for `length(v)` nonzero elements. `sparse` adds together elements in `v` that have duplicate subscripts in `i` and `j`.

If the inputs `i`, `j`, and `v` are vectors or matrices, they must have the same number of elements. Alternatively, the argument `v` and/or one of the arguments `i` or `j` can be scalars.

`S = sparse(i,j,v,m,n)` specifies the size of `S` as `m`-by-`n`.

`S = sparse(i,j,v,m,n,nz)` allocates space for `nz` nonzero elements. Use this syntax to allocate extra space for nonzero values to be filled in after construction.

Examples

Save Memory Using Sparse Storage

Create a 10,000-by-10,000 full storage identity matrix.

```
A = eye(10000);
whos A
```

```
  Name          Size                   Bytes  Class     Attributes

  A         10000x10000            800000000  double
```

This matrix uses 800-megabytes of memory.

Convert the matrix to sparse storage.

```
S = sparse(A);
whos S
```

```
  Name          Size                 Bytes  Class     Attributes

  S         10000x10000            240008  double    sparse
```

In sparse form, the same matrix uses roughly 0.25-megabytes of memory. In this case, you can avoid full storage completely by using the `speye` function, which creates sparse identity matrices directly.

Sparse Matrix of All Zeros

```
S = sparse(10000,5000)
```

```
S =
   All zero sparse: 10000x5000
```

Sparse Matrix of Nonzeros with Specified Size

Create a 1500-by-1500 sparse matrix from the triplets `i`, `j`, and `v`.

```
i = [900 1000];
j = [900 1000];
v = [10 100];
S = sparse(i,j,v,1500,1500)
```

```
S =
 (900,900)     10
(1000,1000)   100
```

When you specify a size larger than `max(i)` -by- `max(j)`, the `sparse` function pads the output with extra rows and columns of zeros.

```
size(S)
```

```
ans = 1×2

        1500        1500
```

Preallocate Storage in Sparse Matrix

Create a sparse matrix with `10` nonzero values, but which has space allocated for `100` nonzero values.

```
S = sparse(1:10,1:10,5,20,20,100);
N = nnz(S)
```

```
N = 10
```

```
N_alloc = nzmax(S)
```

```
N_alloc = 100
```

The `spalloc` function is a shorthand way to create a sparse matrix with *no* nonzero elements but which has space allotted for some number of nonzeros.

Accumulate Values into Sparse Matrix

Use repeated subscripts to accumulate values into a single sparse matrix that would otherwise require one or more loops.

Create a column vector of data and two column vectors of subscripts.

```
i = [6 6 6 5 10 10 9 9]';
j = [1 1 1 2 3 3 10 10]';
v = [100 202 173 305 410 550 323 121]';
```

Visualize the subscripts and values side-by-side.

```
[i,j,v]
```

```
ans = 8×3

     6     1   100
     6     1   202
     6     1   173
     5     2   305
    10     3   410
    10     3   550
     9    10   323
     9    10   121
```

Use the `sparse` function to accumulate the values that have identical subscripts.

```
S = sparse(i,j,v)
```

```
S =
   (6,1)      475
   (5,2)      305
  (10,3)      960
   (9,10)     444
```

Input Arguments

A — Input matrix
full matrix | sparse matrix

Input matrix, specified as a full or sparse matrix. If `A` is already sparse, then `sparse(A)` returns `A`.

Data Types: `double` | `logical`
Complex Number Support: Yes

`i,j` — Subscript pairs (as separate arguments)
scalars | vectors | matrices

Subscript pairs, specified as separate arguments of scalars, vectors, or matrices. Corresponding elements in `i` and `j` specify `S(i,j)` subscript pairs, which determine the placement of the values in `v` into the output. If either `i` or `j` is a vector or matrix, then the other input can be a scalar or can be a vector or matrix with the same number of elements. In that case, `sparse` uses `i(:)` and `j(:)` as the subscripts. If `i` and `j` have identical values for several elements in `v`, then those elements are added together.

Note If any value in `i` or `j` is larger than `2^31-1` for 32-bit platforms, or `2^48-1` on 64-bit platforms, then the sparse matrix cannot be constructed.

Data Types: `double` | `logical`

`v` — Values
scalar | vector | matrix

Values, specified as a scalar, vector, or matrix. If `v` is a vector or matrix, then one of the inputs `i` or `j` must also be a vector or matrix with the same number of elements.

Any elements in `v` that are zero are ignored, as are the corresponding subscripts in `i` and `j`. However, if you do not specify the dimension sizes of the output, `m` and `n`, then `sparse` calculates the maxima `m = max(i)` and `n = max(j)` before ignoring any zero elements in `v`.

Data Types: `double` | `logical`
Complex Number Support: Yes

`m,n` — Size of each dimension (as separate arguments)
integer values

Size of each dimension, specified as separate arguments of integer values. If you specify `m` (the row size), you also must specify `n` (the column size).

If you do not specify `m` and `n`, then `sparse` uses the default values `m = max(i)` and `n = max(j)`. These maxima are computed before any zeros in `v` are removed.

Data Types: `double`

`nz` — Storage allocation for nonzero elements
nonnegative integer

Storage allocation for nonzero elements, specified as a nonnegative integer. `nz` generally must be greater than or equal to `max([numel(i), numel(j), numel(v), 1])`. However, if the sizes of `i`, `j`, and `v` allow you to specify a value of `0` for `nz`, then `sparse` instead sets the value to `1`.

For a sparse matrix, `S`, the `nnz` function returns the number of nonzero elements in the matrix, and the `nzmax` function returns the amount of storage allocated for nonzero matrix elements. If `nnz(S)` and `nzmax(S)` return different results, then more storage might be allocated than is actually required. For this reason, set `nz` only in anticipation of later fill-in.

If you do not specify `nz`, then `sparse` uses a default value of `max([numel(i), numel(j), numel(v), 1])`.

Data Types: `double`

Tips

- MATLAB stores sparse matrices in compressed sparse column format. For more information, see John R. Gilbert, Cleve Moler, and Robert Schreiber's Sparse Matrices In MATLAB: Design and Implementation.
- The `accumarray` function has similar accumulation behavior to that of `sparse`.
  - `accumarray` groups data into bins using *n*-dimensional subscripts, but `sparse` groups data into bins using 2-D subscripts.
  - `accumarray` adds elements that have identical subscripts into the output by default, but can optionally apply any function to the bins. `sparse` only adds elements that have identical subscripts into the output.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The number of rows, columns, and nonzero elements must each have a value less than `intmax`.
- In MATLAB, you can construct a sparse matrix using scalar expansion. For example, `sparse([1 2],[3 4], 2)`. For code generation, you can only use scalar expansion for compile-time scalar inputs. Variable-size arrays that are scalar at run time are not expanded.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`accumarray` | `diag` | `find` | `full` | `issparse` | `nnz` | `nonzeros` | `nzmax` | `spalloc` | `speye` | `spones` | `sprandn` | `sprandsym` | `spy`

Topics

“Constructing Sparse Matrices”

"Sparse Matrix Operations"
"Accessing Sparse Matrices"

Introduced before R2006a

spaugment

Form least-squares augmented system

Syntax

```
S = spaugment(A,c)
S = spaugment(A)
```

Description

`S = spaugment(A,c)` creates the sparse, square, symmetric indefinite matrix `S = [c*I A; A' 0]`. The matrix `S` is related to the least-squares problem

```
min norm(b - A*x)
```

by

```
r = b - A*x
S * [r/c; x] = [b; 0]
```

The optimum value of the residual scaling factor `c`, involves `min(svd(A))` and `norm(r)`, which are usually too expensive to compute.

`S = spaugment(A)` without a specified value of `c`, uses `max(max(abs(A)))/1000`.

Note In previous versions of MATLAB, the augmented matrix was used by sparse linear equation solvers, \ and /, for nonsquare problems. Now, MATLAB calculates a least-squares solution using the `qr` factorization of `A`.

See Also

`spparms`

Introduced before R2006a

spconvert

Import from sparse matrix external format

Syntax

```
S = spconvert(D)
```

Description

`S = spconvert(D)` constructs sparse matrix `S` from the columns of `D` in a manner similar to the `sparse` function.

- If `D` is of size `N`-by-`3`, then `spconvert` uses the columns `[i,j,re]` of `D` to construct `S`, such that `S(i(k), j(k)) = re(k)`.
- If `D` is of size `N`-by-`4`, then `spconvert` uses the columns `[i,j,re,im]` of `D` to construct `S`, such that `S(i(k), j(k)) = re(k) + 1i*im(k)`.

Examples

Convert Data File to Sparse Matrix

Create an ASCII file, `uphill.dat`, which contains the following values. Save the file in your current directory.

```
1    1    1.000000000000000
1    2    0.500000000000000
2    2    0.333333333333333
1    3    0.333333333333333
2    3    0.250000000000000
3    3    0.200000000000000
1    4    0.250000000000000
2    4    0.200000000000000
3    4    0.166666666666667
4    4    0.142857142857143
4    4    0.000000000000000
```

It is common to purposefully make the last line of the file include the desired size of the matrix with a value of 0. This practice ensures that the converted sparse matrix has that size.

Load the data into MATLAB and convert it into a sparse matrix.

```
load uphill.dat
H = spconvert(uphill)
```

```
H =

   (1,1)        1.0000
   (1,2)        0.5000
   (2,2)        0.3333
   (1,3)        0.3333
   (2,3)        0.2500
   (3,3)        0.2000
   (1,4)        0.2500
   (2,4)        0.2000
   (3,4)        0.1667
   (4,4)        0.1429
```

In this case, the last line in the file is not necessary because the earlier lines already specify that the matrix is at least 4-by-4.

Input Arguments

D — Input matrix
matrix

Input matrix, specified as a matrix with either three or four columns. In both cases, the first two columns of `D` are subscripts and the third column is composed of data values. A four column matrix specifies the real (third column) and imaginary (fourth column) parts of complex numbers.

If `D` is already a sparse matrix, then `spconvert` returns `D`.

Data Types: `single` | `double`

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`full` | `sparse`

Introduced before R2006a

spdiags

Extract nonzero diagonals and create sparse band and diagonal matrices

Syntax

```
Bout = spdiags(A)
[Bout,id] = spdiags(A)

Bout = spdiags(A,d)

S = spdiags(Bin,d,m,n)
S = spdiags(Bin,d,A)
```

Description

`Bout = spdiags(A)` extracts the nonzero diagonals from `m`-by-`n` matrix `A` and returns them as the columns in `min(m,n)`-by-`p` matrix `Bout`, where `p` is the number of nonzero diagonals.

`[Bout,id] = spdiags(A)` also returns the diagonal numbers `id` for the nonzero diagonals in `A`. The size of `Bout` is `min(m,n)`-by-`length(id)`.

`Bout = spdiags(A,d)` extracts the diagonals in `A` specified by `d` and returns them as the columns of `min(m,n)`-by-`length(d)` matrix `Bout`.

`S = spdiags(Bin,d,m,n)` creates an `m`-by-`n` sparse matrix `S` by taking the columns of `Bin` and placing them along the diagonals specified by `d`.

`S = spdiags(Bin,d,A)` replaces the diagonals in `A` specified by `d` with the columns of `Bin`.

Examples

Create Tridiagonal Matrix

Create a tridiagonal matrix using three vectors, change some of the matrix diagonals, and then extract the diagonals.

Create a 9-by-1 vector of ones, and then create a tridiagonal matrix using the vector. View the matrix elements.

```
n = 9;
e = ones(n,1);
A = spdiags([e -2*e e],-1:1,n,n);
full(A)
```

```
ans = 9×9

    -2     1     0     0     0     0     0     0     0
     1    -2     1     0     0     0     0     0     0
     0     1    -2     1     0     0     0     0     0
     0     0     1    -2     1     0     0     0     0
     0     0     0     1    -2     1     0     0     0
     0     0     0     0     1    -2     1     0     0
     0     0     0     0     0     1    -2     1     0
     0     0     0     0     0     0     1    -2     1
     0     0     0     0     0     0     0     1    -2
```

Change the values on the main (`d = 0`) diagonal of `A`.

```
Bin = abs(-(n-1)/2:(n-1)/2)';
d = 0;
A = spdiags(Bin,d,A);
full(A)
```

```
ans = 9×9

     4     1     0     0     0     0     0     0     0
     1     3     1     0     0     0     0     0     0
     0     1     2     1     0     0     0     0     0
     0     0     1     1     1     0     0     0     0
     0     0     0     1     0     1     0     0     0
     0     0     0     0     1     1     1     0     0
     0     0     0     0     0     1     2     1     0
     0     0     0     0     0     0     1     3     1
     0     0     0     0     0     0     0     1     4
```

Finally, recover the diagonals of A as the columns in a matrix.

```
Bout = spdiags(A);
full(Bout)
```

```
ans = 9×3

     1     4     0
     1     3     1
     1     2     1
     1     1     1
     1     0     1
     1     1     1
     1     2     1
     1     3     1
     0     4     1
```

Extract Nonzero Diagonals from Matrix

Extract the nonzero diagonals of a matrix and examine the output format of `spdiags`.

Create a matrix containing a mix of nonzero and zero diagonals.

```
A = [0     5     0    10     0     0
     0     0     6     0    11     0
     3     0     0     7     0    12
     1     4     0     0     8     0
     0     2     5     0     0     9];
```

Extract the nonzero diagonals from the matrix. Specify two outputs to return the diagonal numbers.

```
[Bout,d] = spdiags(A)
```

```
Bout = 5×4

     0     0     5    10
     0     0     6    11
     0     3     7    12
     1     4     8     0
     2     5     9     0
```

```
d = 4×1

    -3
    -2
     1
     3
```

The columns of the first output `Bout` contain the nonzero diagonals of `A`. The second output `d` lists the indices of the nonzero diagonals of `A`. The longest nonzero diagonal in `A` is in column 3 of `Bout`. To give all columns of `Bout` the same length, the other nonzero diagonals of `A` have extra zeros added to their corresponding columns in `Bout`. For `m`-by-`n` matrices with `m < n`, the rules are:

- For nonzero diagonals *below* the main diagonal of `A`, extra zeros are added at the *tops* of columns (as in the first two columns of `Bout`).
- For nonzero diagonals *above* the main diagonal of `A`, extra zeros are added at the *bottoms* of columns (as in the last column of `Bout`).

`spdiags` pads `Bout` with zeros in this manner even if the longest diagonal is not returned in `Bout`.

Extract Specific Diagonals from Matrix

Create a 5-by-5 random matrix.

```
A = randi(10,5,5)
```

```
A = 5×5

     9     1     2     2     7
    10     3    10     5     1
     2     6    10    10     9
    10    10     5     8    10
     7    10     9    10     7
```

Extract the main diagonal, and the first diagonals above and below it.

```
d = [-1 0 1];
Bout = spdiags(A,d)
```

```
Bout = 5×3

    10     9     0
     6     3     1
     5    10    10
    10     8    10
     0     7    10
```

Try to extract the fifth super-diagonal (`d = 5`). Because `A` has only four super-diagonals, `spdiags` returns the diagonal as all zeros of the same length as the main (`d = 0`) diagonal.

```
B5 = spdiags(A,5)
```

```
B5 = 5×1

     0
     0
     0
     0
     0
```

Columns and Diagonals of Different Sizes

Examine how `spdiags` creates diagonals when the columns of the input matrix are longer than the diagonals they are replacing.

Create a 6-by-7 matrix of the numbers 1 through 6.

```
Bin = repmat((1:6)',[1 7])
```

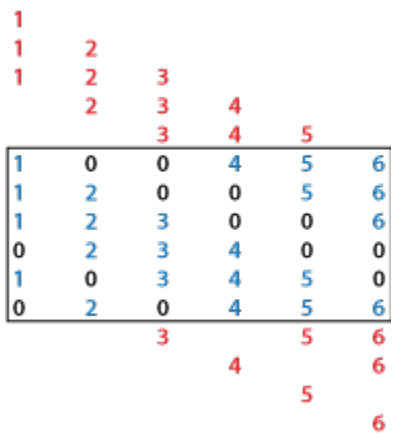
```
Bin = 6×7

     1     1     1     1     1     1     1
     2     2     2     2     2     2     2
     3     3     3     3     3     3     3
     4     4     4     4     4     4     4
     5     5     5     5     5     5     5
     6     6     6     6     6     6     6
```

Use `spdiags` to create a square 6-by-6 matrix with several of the columns of `Bin` as diagonals. Because some of the diagonals only have one or two elements, there is a mismatch in sizes between the columns in `Bin` and diagonals in `A`.

```
d = [-4 -2 -1 0 3 4 5];
A = spdiags(Bin,d,6,6);
full(A)
```

```
ans = 6×6

     1     0     0     4     5     6
     1     2     0     0     5     6
     1     2     3     0     0     6
     0     2     3     4     0     0
     1     0     3     4     5     0
     0     2     0     4     5     6
```

Each of the columns in `Bin` has six elements, but only the main diagonal in `A` has six elements. Therefore, all the other diagonals in `A` truncate the elements in the columns of `Bin` so that they fit on the selected diagonals:



The way `spdiags` truncates the diagonals depends on the size of `m`-by-`n` matrix `A`. When $m \geq n$, the behavior is as pictured above:

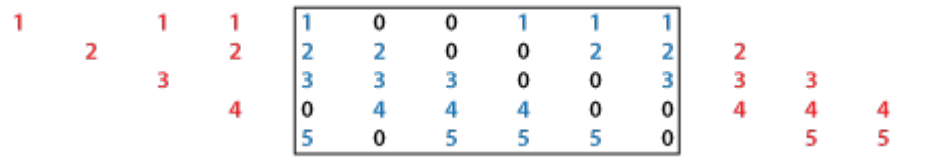
- Diagonals *below* the main diagonal take elements from the *tops* of the columns first.

- Diagonals *above* the main diagonal take elements from the *bottoms* of columns first.

This behavior reverses when $m < n$:

```
A = spdiags(Bin,d,5,6);
full(A)
```

```
ans = 5×6

     1     0     0     1     1     1
     2     2     0     0     2     2
     3     3     3     0     0     3
     0     4     4     4     0     0
     5     0     5     5     5     0
```



- Diagonals *above* the main diagonal take elements from the *tops* of the columns first.
- Diagonals *below* the main diagonal take elements from the *bottoms* of columns first.

Input Arguments

A — Input matrix
matrix

Input matrix. This matrix is typically (but not necessarily) sparse.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

d — Diagonal numbers
scalar | vector

Diagonal numbers, specified as a scalar or vector of positive integers. The diagonal numbers follow the same conventions as `diag`:

- `d < 0` is below the main diagonal, and satisfies `d >= -(m-1)`.
- `d = 0` is the main diagonal.
- `d > 0` is above the main diagonal, and satisfies `d <= (n-1)`.

An `m`-by-`n` matrix `A` has `(m + n - 1)` diagonals. These diagonals are specified in the vector `d` using indices from `-(m-1)` to `(n-1)`. For example, if `A` is 5-by-6, it has 10 diagonals, which are specified in the vector `d` using the indices -4, -3 , ... 4, 5. The following diagram illustrates this diagonal numbering.

0	1	2	3	4	5
-1	0	1	2	3	4
-2	-1	0	1	2	3
-3	-2	-1	0	1	2
-4	-3	-2	-1	0	1

If you specify a diagonal that lies outside of `A` (such as `d = 7` in the example above), then `spdiags` returns that diagonal as all zeros.

Example: `spdiags(A,[3 5])` extracts the third and fifth diagonals from `A`.

`Bin` — Diagonal elements
matrix

Diagonal elements, specified as a matrix. This matrix is typically (but not necessarily) full. `spdiags` uses the columns of `Bin` to replace specified diagonals in `A`. If the requested size of the output is `m`-by-`n`, then `Bin` must have `min(m,n)` columns.

With the syntax `S = spdiags(Bin,d,m,n)`, if a column of `Bin` has more elements than the diagonal it is replacing, and `m >= n`, then `spdiags` takes elements of super-diagonals from the *lower* part of the column of `Bin`, and elements of sub-diagonals from the *upper* part of the column of `Bin`. However, if `m < n` , then super-diagonals are from the *upper* part of the column of `Bin`, and sub-diagonals from the *lower* part. For an example of this behavior, see “Columns and Diagonals of Different Sizes” on page 1-14299.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

`m`, `n` — Dimension sizes
nonnegative scalar integers

Dimension sizes, specified as nonnegative scalar integers. `spdiags` uses these inputs to determine how large a matrix to create.

Example: `spdiags(Bin,d,300,400)` creates a 300-by-400 matrix with the columns of `B` placed along the specified diagonals `d`.

Output Arguments

`Bout` — Diagonal elements
matrix

Diagonal elements, returned as a full matrix. The columns of `Bout` contain diagonals extracted from `A`. Any elements of `Bout` corresponding to positions outside of `A` are set to zero.

`id` — Diagonal numbers
column vector

Diagonal numbers, returned as a column vector. See `d` for a description of the diagonal numbering.

`S` — Output matrix
matrix

Output matrix. `S` takes one of two forms:

- With `S = spdiags(Bin,d,A)`, the specified diagonals in `A` are replaced with the columns in `Bin` to create `S`.
- With `S = spdiags(Bin,d,m,n)`, the `m`-by-`n` sparse matrix `S` is formed by taking the columns of `Bin` and placing them along the diagonals specified by `d`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The first input cannot be sparse.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`diag` | `speye`

Introduced before R2006a

specular

Calculate specular reflectance

Syntax

```
R = specular(Nx,Ny,Nz,S,V)
```

Description

`R = specular(Nx,Ny,Nz,S,V)` returns the reflectance of a surface with normal vector components `[Nx,Ny,Nz]`. `S` and `V` specify the direction to the light source and to the viewer, respectively. You can specify these directions as three vectors `[x,y,z]` or two vectors `[Theta Phi` (in spherical coordinates).

The specular highlight is strongest when the normal vector is in the direction of `(S+V)/2` where `S` is the source direction, and `V` is the view direction.

The surface spread exponent can be specified by including a sixth argument as in `specular(Nx,Ny,Nz,S,V,spread)`.

speye

Sparse identity matrix

Syntax

```
S = speye
S = speye(n)
S = speye(n,m)
S = speye(sz)
```

Description

`S = speye` returns a sparse scalar `1`.

`S = speye(n)` returns a sparse `n`-by-`n` identity matrix, with ones on the main diagonal and zeros elsewhere.

`S = speye(n,m)` returns a sparse `n`-by-`m` matrix, with ones on the main diagonal and zeros elsewhere.

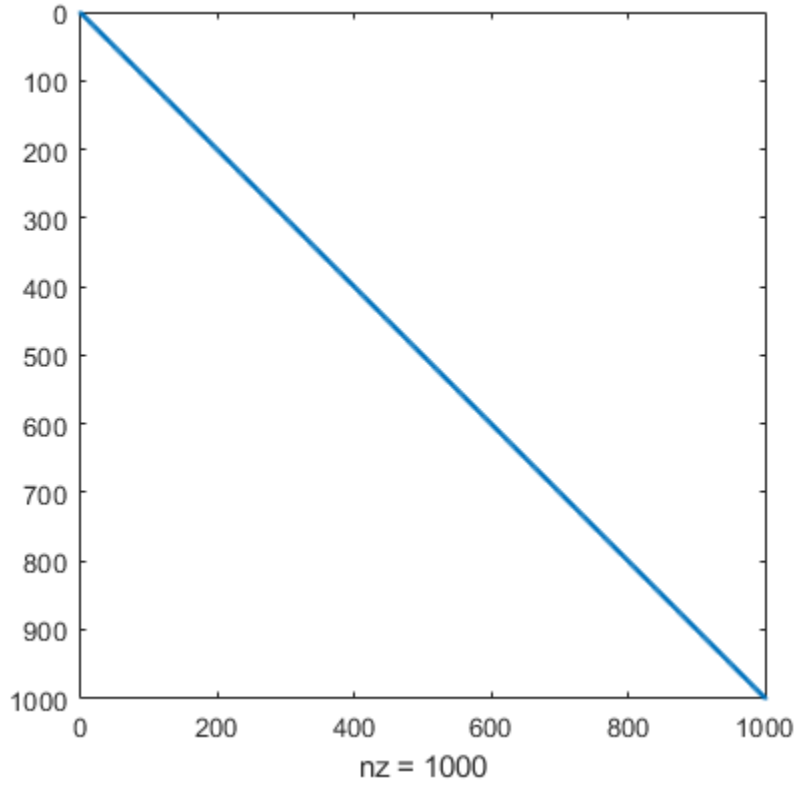
`S = speye(sz)` returns a matrix with ones on the main diagonal and zeros elsewhere. The size vector `sz` defines `size(S)`. For example, `speye([2 3])` returns a 2-by-3 matrix.

Examples

Square Sparse Identity Matrix

Create a 1000-by-1000 square sparse identity matrix and view the sparsity pattern.

```
I = speye(1000);
spy(I)
```

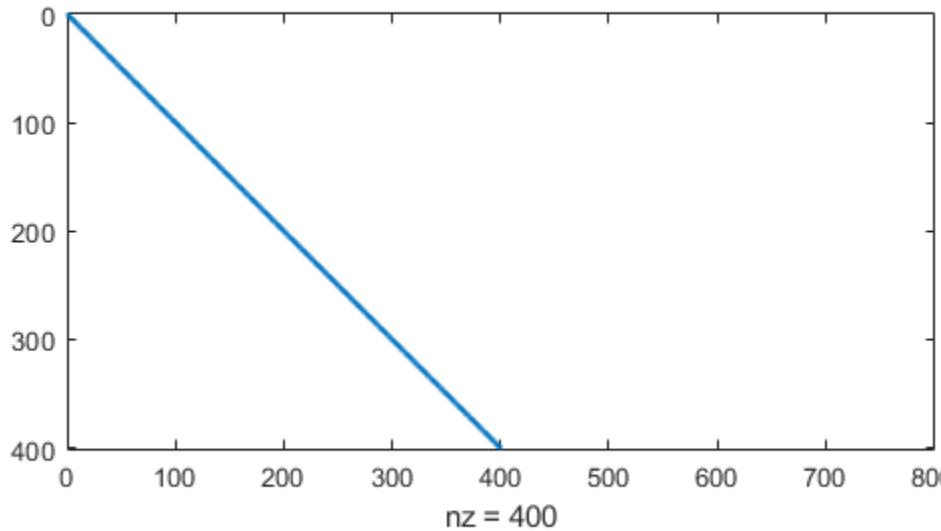


The result is the same as `sparse(eye(1000))`, but this latter version requires temporary storage for the full representation before it is converted to sparse storage.

Rectangular Sparse Identity Matrix

Create a 400-by-800 sparse identity matrix and view the pattern of nonzeros.

```
S = speye(400,800);
spy(S)
```



This command is equivalent to `speye([400 800])`.

Input Arguments

`n`, `m` — Dimension sizes
nonnegative integer scalars

Dimension sizes, specified as nonnegative integer scalars.

- If `n` is the only input argument, then `S` is an `n`-by-`n` identity matrix.

- If `n` or `m` is `0`, then `S` is an empty matrix.
- If `n` or `m` is negative, then it is treated as `0`.

Example: `speye(4)` creates a 4-by-4 identity matrix.

Example: `speye(3,6)` creates a 3-by-6 identity matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz` — Dimension sizes
two-element row vector

Dimension sizes, specified as a two-element row vector. The row vector is of the form `[numRows numCols]`.

- If an element of `sz` is `0`, then `S` is an empty matrix.
- If an element of `sz` is negative, then the element is treated as `0`.

Example: `speye([4 5])` creates a 4-by-5 matrix and is equivalent to `speye(4,5)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`eye` | `spalloc` | `spdiags` | `spones` | `sprand` | `sprandn`

Introduced before R2006a

spfun

Apply function to nonzero sparse matrix elements

Syntax

```
f = spfun(fun,S)
```

Description

The `spfun` function selectively applies a function to only the *nonzero* elements of a sparse matrix `S`, preserving the sparsity pattern of the original matrix (except for underflow or if `fun` returns zero for some nonzero elements of `S`).

`f = spfun(fun,S)` evaluates `fun(S)` on the elements of `S` that are nonzero. `fun` is a function handle.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

Examples

Given the 4-by-4 sparse diagonal matrix

```
S = spdiags([1:4]',0,4,4)
```

```
S =
   (1,1)        1
   (2,2)        2
   (3,3)        3
   (4,4)        4
```

Because `fun` returns nonzero values for all nonzero element of `S`, `f = spfun(@exp,S)` has the same sparsity pattern as `S`.

```
f =
   (1,1)       2.7183
```

```
   (2,2)       7.3891
   (3,3)      20.0855
   (4,4)      54.5982
```

whereas `exp(S)` has `1`s where `S` has `0`s.

```
full(exp(S))

ans =
    2.7183    1.0000    1.0000    1.0000
    1.0000    7.3891    1.0000    1.0000
    1.0000    1.0000   20.0855    1.0000
    1.0000    1.0000    1.0000   54.5982
```

Tips

Functions that operate element-by-element, like those in the `elfun` directory, are the most appropriate functions to use with `spfun`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- On each lab, `fun` receives as input only the nonzero elements of `S` that are local to the lab. Therefore, functions that operate element by element are the most appropriate for use with `spfun`.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`sparse`

Topics

“Create Function Handle”

Introduced before R2006a

sph2cart

Transform spherical coordinates to Cartesian

Syntax

```
[x,y,z] = sph2cart(azimuth,elevation,r)
```

Description

`[x,y,z] = sph2cart(azimuth,elevation,r)` transforms corresponding elements of the spherical coordinate arrays `azimuth`, `elevation`, and `r` to Cartesian, or *xyz*, coordinates.

Examples

Spherical to Cartesian Coordinates

Convert the spherical coordinates defined by corresponding entries in the matrices `az`, `el`, and `r` to Cartesian coordinates `x`, `y`, and `z`. These points correspond to the eight vertices of a cube.

```
az = [0.7854 0.7854 -0.7854 -0.7854; 2.3562 2.3562 -2.3562 -2.3562]
```

```
az = 2×4

    0.7854    0.7854   -0.7854   -0.7854
    2.3562    2.3562   -2.3562   -2.3562
```

```
el = [0.6155 -0.6155 0.6155 -0.6155; 0.6155 -0.6155 0.6155 -0.6155]
```

```
el = 2×4

    0.6155   -0.6155    0.6155   -0.6155
```

```
    0.6155   -0.6155    0.6155   -0.6155
```

```
r = 1.7321*ones(2,4)
```

```
r = 2×4

    1.7321    1.7321    1.7321    1.7321
    1.7321    1.7321    1.7321    1.7321
```

```
[x,y,z] = sph2cart(az,el,r)
```

```
x = 2×4

    1.0000    1.0000    1.0000    1.0000
   -1.0000   -1.0000   -1.0000   -1.0000


y = 2×4

    1.0000    1.0000   -1.0000   -1.0000
    1.0000    1.0000   -1.0000   -1.0000


z = 2×4

    1.0001   -1.0001    1.0001   -1.0001
    1.0001   -1.0001    1.0001   -1.0001
```

Input Arguments

`azimuth` — Azimuth angle

scalar | vector | matrix | multidimensional array

Azimuth angle, specified as a scalar, vector, matrix, or multidimensional array. `azimuth`, `elevation`, and `r` must be the same size, or any of them can be scalar.

`azimuth` is the counterclockwise angle in the *x*-*y* plane measured in radians from the positive *x*-axis.

Data Types: `single` | `double`

Complex Number Support: Yes

`elevation` — Elevation angle
scalar | vector | matrix | multidimensional array

Elevation angle, specified as a scalar, vector, matrix, or multidimensional array. `azimuth`, `elevation`, and `r` must be the same size, or any of them can be scalar.

`elevation` is the elevation angle in radians from the *x*-*y* plane.

Data Types: `single` | `double`
Complex Number Support: Yes

`r` — Radius
scalar | vector | matrix | multidimensional array

Radius, specified as a scalar, vector, matrix, or multidimensional array. `azimuth`, `elevation`, and `r` must be the same size, or any of them can be scalar.

The length units of `r` can be arbitrary, and the output arrays `x`, `y`, and `z` use the same units.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

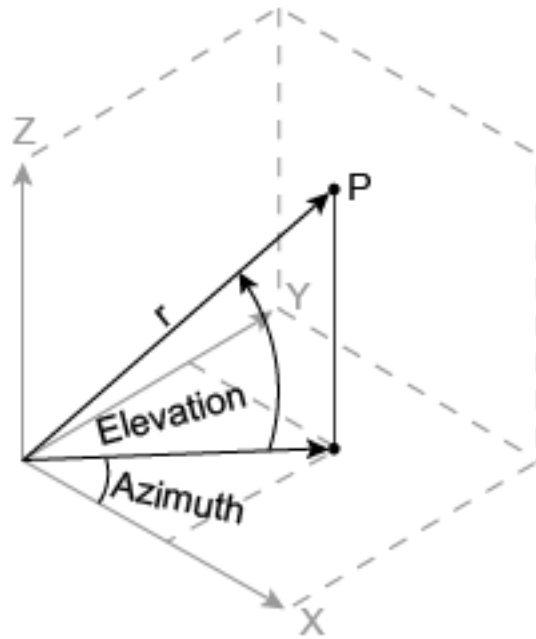
`x,y,z` — Cartesian coordinates
arrays

Cartesian coordinates, returned as arrays.

Algorithms

The mapping from spherical coordinates to three-dimensional Cartesian coordinates is

```
x = r .* cos(elevation) .* cos(azimuth)
y = r .* cos(elevation) .* sin(azimuth)
z = r .* sin(elevation)
```



Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cart2pol` | `cart2sph` | `pol2cart`

Introduced before R2006a

sphere

Generate sphere

Syntax

```
sphere
sphere(n)
sphere(ax,...)
[X,Y,Z] = sphere(...)
```

Description

The `sphere` function generates the *x*-, *y*-, and *z*-coordinates of a unit sphere for use with `surf` and `mesh`.

`sphere` generates a sphere consisting of 20-by-20 faces.

`sphere(n)` draws a `surf` plot of an `n`-by-`n` sphere in the current figure.

`sphere(ax,...)` creates the sphere in the axes specified by `ax` instead of in the current axes. Specify `ax` as the first input argument.

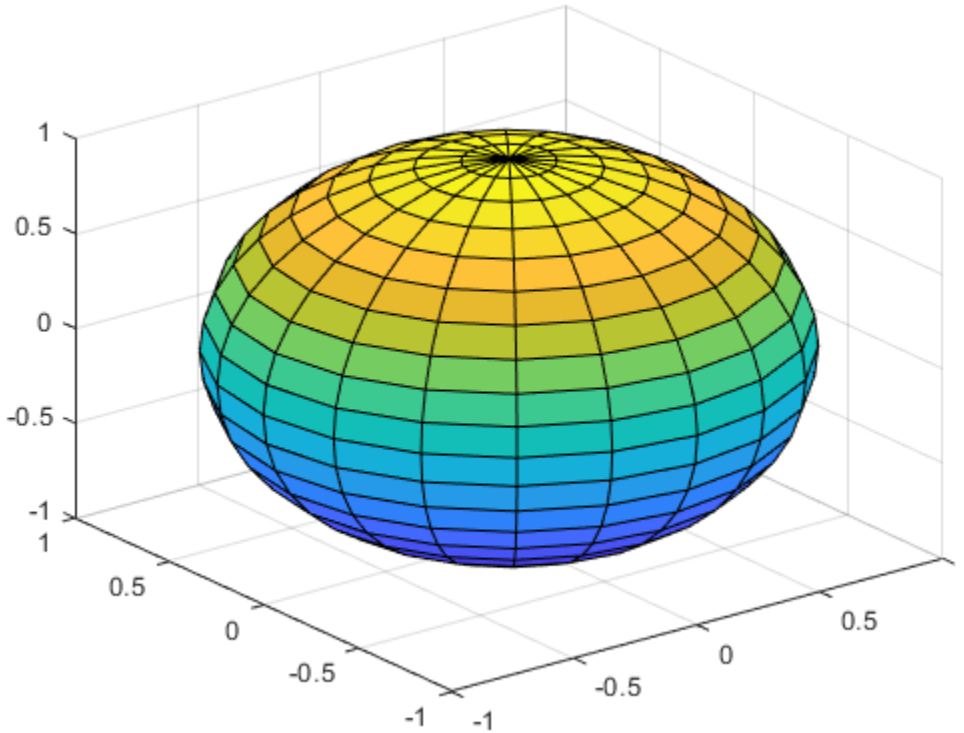
`[X,Y,Z] = sphere(...)` returns the coordinates of the `n`-by-`n` sphere in three matrices that are `(n+1)`-by-`(n+1)` in size. You draw the sphere with `surf(X,Y,Z)` or `mesh(X,Y,Z)`.

Examples

Plot Sphere

Generate and plot a sphere.

```
figure
sphere
```



Plot Multiple Spheres

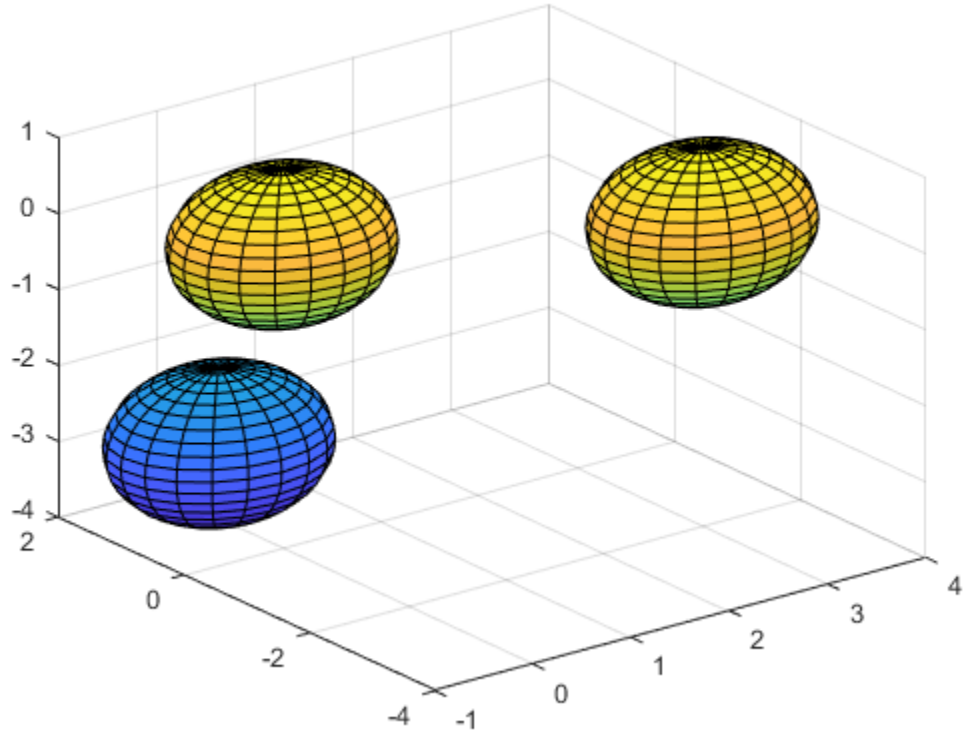
Define `x`, `y`, and `z` as coordinates of a sphere.

```
[x,y,z] = sphere;
```

Plot a sphere centered at the origin. Plot two more spheres centered at `(3,-2,0)` and `(0,1,-3)`.

```
figure
surf(x,y,z)
```

```
hold on
surf(x+3,y-2,z) % centered at (3,-2,0)
surf(x,y+1,z-3) % centered at (0,1,-3)
```



See Also

axis | cylinder

Introduced before R2006a

spinmap

Rotate colormap colors

Syntax

```
spinmap
spinmap(t)
spinmap(t,n)
```

Description

`spinmap` rotates the current colormap by two rows repeatedly for approximately three seconds.

`spinmap(t)` rotates the current colormap by two rows repeatedly for `t` seconds.

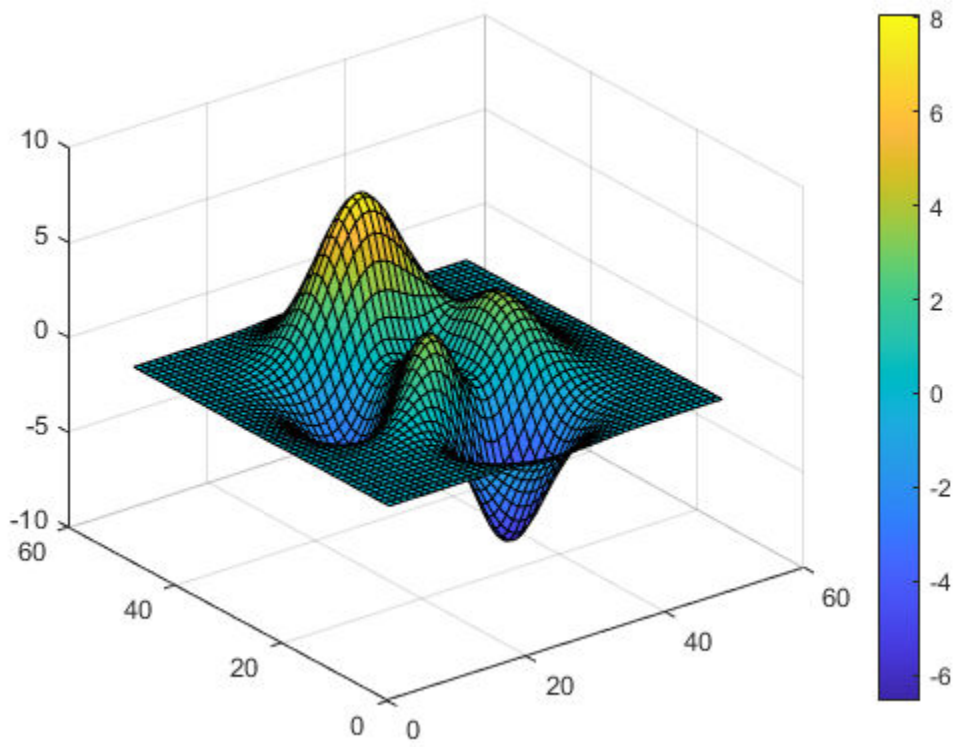
`spinmap(t,n)` rotates the current colormap by `n` rows for `t` seconds.

Examples

Rotate Colormap with Specific Duration and Increment

Create a surface plot that uses the default colormap. Then rotate the colormap by one row for five seconds.

```
surf(peaks);
colorbar
spinmap(5,1)
```



Input Arguments

t — Rotation duration
3 (default) | scalar numeric value | `'inf'`

Rotation duration, specified as a scalar numeric value or `'inf'`. A numeric value specifies the number of seconds to rotate the colormap. A value of `'inf'` makes the colormap rotate indefinitely.

n — Rotation amount
2 (default) | scalar numeric value

Rotation amount, specified as a numeric scalar value. The first rotation shifts each color in the colormap by `n` rows. Subsequent rotations shift the colormap by an additional `n` rows.

Positive values increment the rows, where the last color wraps to the first row. A negative value decrements the rows, where the first color wraps to the last row.

Tips

Press **Ctrl+C** to interrupt the rotation before `t` has elapsed.

See Also

Colormap Editor | `colormap`

Introduced before R2006a

spline

Cubic spline data interpolation

Syntax

```
s = spline(x,y,xq)
pp = spline(x,y)
```

Description

`s = spline(x,y,xq)` returns a vector of interpolated values `s` corresponding to the query points in `xq`. The values of `s` are determined by cubic spline interpolation of `x` and `y`.

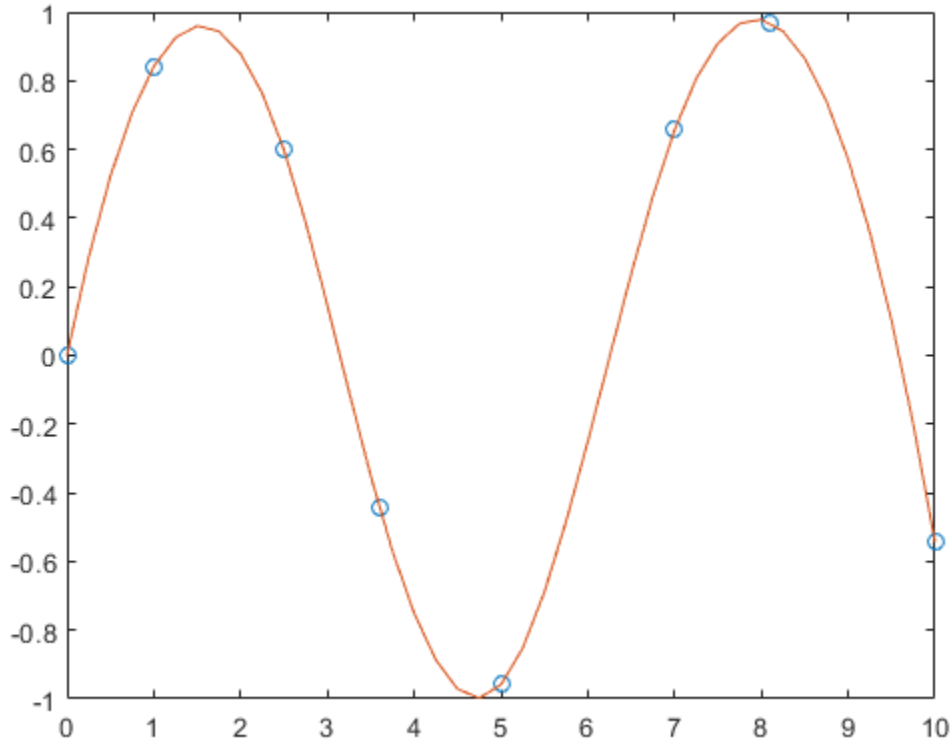
`pp = spline(x,y)` returns a piecewise polynomial structure for use by `ppval` and the spline utility `unmkpp`.

Examples

Spline Interpolation of Sine Data

Use `spline` to interpolate a sine curve over unevenly-spaced sample points.

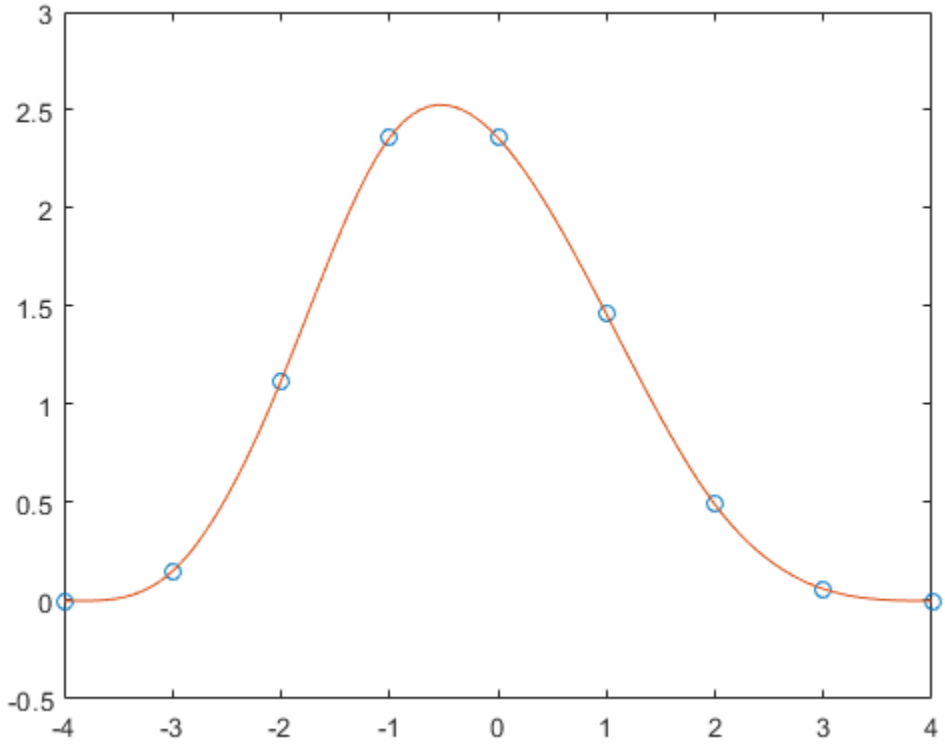
```
x = [0 1 2.5 3.6 5 7 8.1 10];
y = sin(x);
xx = 0:.25:10;
yy = spline(x,y,xx);
plot(x,y,'o',xx,yy)
```



Spline Interpolation of Distribution with Specified Endpoint Slopes

Use clamped or complete spline interpolation when endpoint slopes are known. This example enforces zero slopes at the end points of the interpolation.

```
x = -4:4;
y = [0 .15 1.12 2.36 2.36 1.46 .49 .06 0];
cs = spline(x,[0 y 0]);
xx = linspace(-4,4,101);
plot(x,y,'o',xx,ppval(cs,xx),'-');
```



Extrapolation Using Cubic Spline

Extrapolate a data set to predict population growth.

Create two vectors to represent the census years from 1900 to 1990 (`t`) and the corresponding United States population in millions of people (`p`).

```
t = 1900:10:1990;
p = [ 75.995  91.972  105.711  123.203  131.669 ...
     150.697 179.323  203.212  226.505  249.633 ];
```

Extrapolate and predict the population in the year 2000 using a cubic spline.

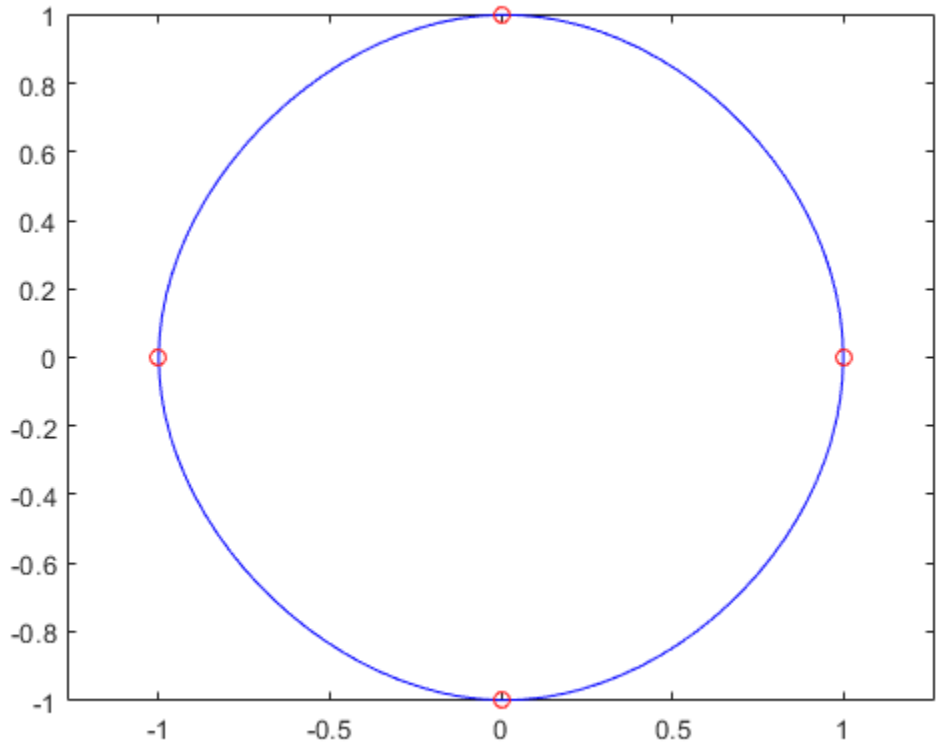
```
spline(t,p,2000)
```

```
ans = 270.6060
```

Spline Interpolation of Angular Data

Generate the plot of a circle, with the five data points `y(:,2),...,y(:,6)` marked with o's. The matrix `y` contains two more columns than does `x`. Therefore, `spline` uses `y(:,1)` and `y(:,end)` as the endslopes. The circle starts and ends at the point (1,0), so that point is plotted twice.

```
x = pi*[0:.5:2];
y = [0  1  0 -1  0  1  0;
     1  0  1  0 -1  0  1];
pp = spline(x,y);
yy = ppval(pp, linspace(0,2*pi,101));
plot(yy(1,:),yy(2,:),'-b',y(1,2:5),y(2,2:5),'or')
axis equal
```



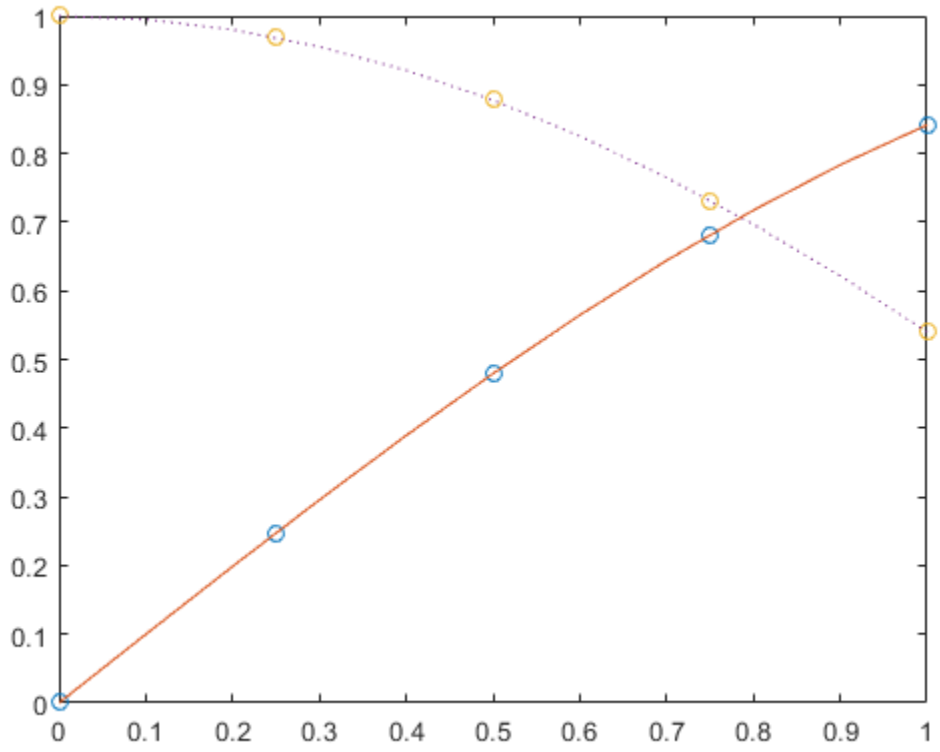
Spline Interpolation of Sine and Cosine Data

Use spline to sample a function over a finer mesh.

Generate sine and cosine curves for a few values between 0 and 1. Use spline interpolation to sample the functions over a finer mesh.

```
x = 0:.25:1;
Y = [sin(x); cos(x)];
xx = 0:.1:1;
YY = spline(x,Y,xx);
```

```
plot(x,Y(1,:),'o',xx,YY(1,:),'-')
hold on
plot(x,Y(2,:),'o',xx,YY(2,:),':')
hold off
```

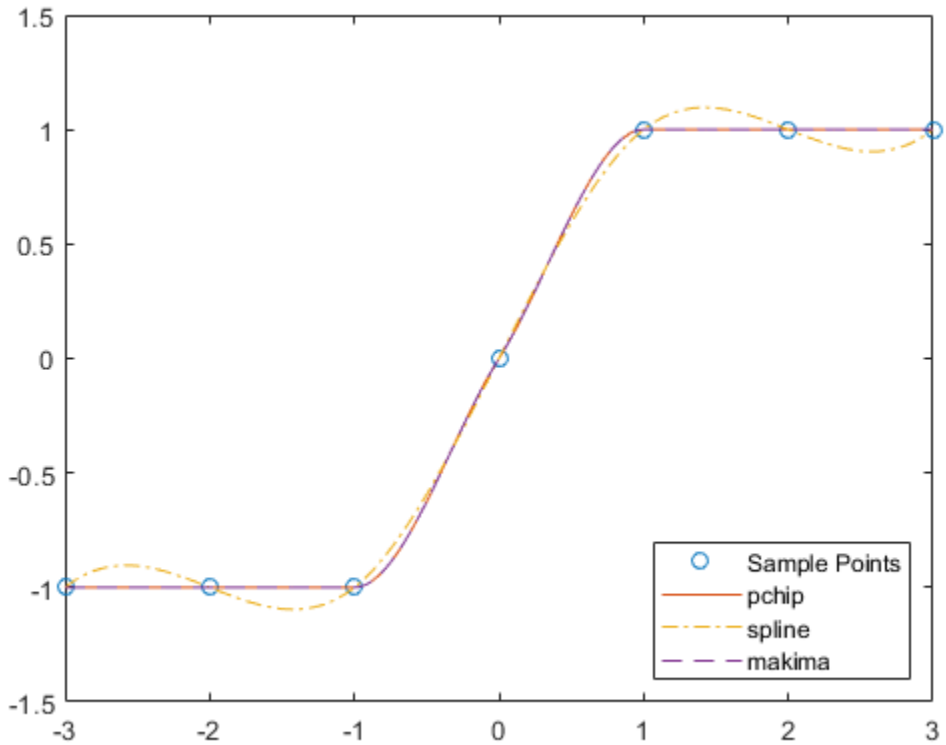


Data Interpolation with `spline`, `pchip`, and `makima`

Compare the interpolation results produced by `spline`, `pchip`, and `makima` for two different data sets. These functions all perform different forms of piecewise cubic Hermite interpolation. Each function differs in how it computes the slopes of the interpolant, leading to different behaviors when the underlying data has flat areas or undulations.

Compare the interpolation results on sample data that connects flat regions. Create vectors of `x` values, function values at those points `y`, and query points `xq`. Compute interpolations at the query points using `spline`, `pchip`, and `makima`. Plot the interpolated function values at the query points for comparison.

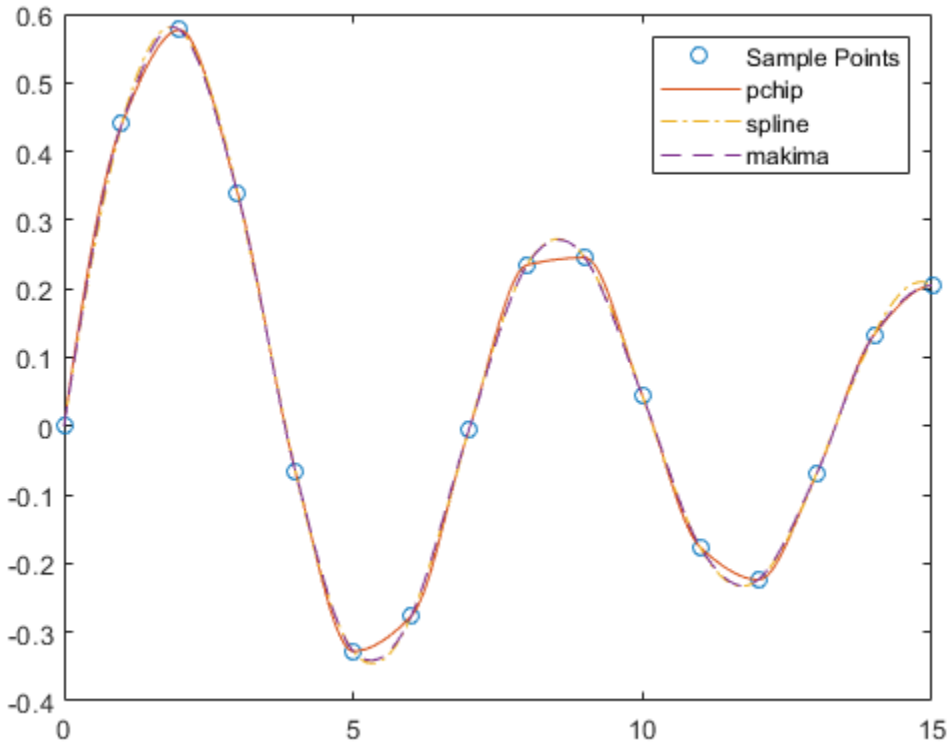
```
x = -3:3;
y = [-1 -1 -1 0 1 1 1];
xq1 = -3:.01:3;
p = pchip(x,y,xq1);
s = spline(x,y,xq1);
m = makima(x,y,xq1);
plot(x,y,'o',xq1,p,'-',xq1,s,'-.',xq1,m,'--')
legend('Sample Points','pchip','spline','makima','Location','SouthEast')
```



In this case, `pchip` and `makima` have similar behavior in that they avoid overshoots and can accurately connect the flat regions.

Perform a second comparison using an oscillatory sample function.

```
x = 0:15;
y = besselj(1,x);
xq2 = 0:0.01:15;
p = pchip(x,y,xq2);
s = spline(x,y,xq2);
m = makima(x,y,xq2);
plot(x,y,'o',xq2,p,'-',xq2,s,'-.',xq2,m,'--')
legend('Sample Points','pchip','spline','makima')
```



When the underlying function is oscillatory, `spline` and `makima` capture the movement between points better than `pchip`, which is aggressively flattened near local extrema.

Input Arguments

x — *x*-coordinates

vector

x-coordinates, specified as a vector. The vector `x` specifies the points at which the data `y` is given. The elements of `x` must be unique.

Data Types: `single` | `double`

y — Function values at *x*-coordinates

vector | matrix | array

Function values at *x*-coordinates, specified as a numeric vector, matrix, or array. `x` and `y` typically have the same length, but `y` also can have exactly two more elements than `x` to specify endslopes.

If `y` is a matrix or array, then the values in the last dimension, `y(:,...,:,j)`, are taken as the values to match with `x`. In that case, the last dimension of `y` must be the same length as `x` or have exactly two more elements.

The endslopes of the cubic spline follow these rules:

- If `x` and `y` are vectors of the same size, then the not-a-knot end conditions are used.
- If `x` or `y` is a scalar, then it is expanded to have the same length as the other and the not-a-knot end conditions are used.
- If `y` is a vector that contains two more values than `x` has entries, then `spline` uses the first and last values in `y` as the endslopes for the cubic spline. For example, if `y` is a vector, then:
  - `y(2:end-1)` gives the function values at each point in `x`
  - `y(1)` gives the slope at the beginning of the interval located at `min(x)`
  - `y(end)` gives the slope at the end of the interval located at `max(x)`
- Similarly, if `y` is a matrix or an `N`-dimensional array with `size(y,N)` equal to `length(x)+2`, then:

- `y(:,...,:,j+1)` gives the function values at each point in `x` for `j = 1:length(x)`
- `y(:,:,...:,1)` gives the slopes at the beginning of the intervals located at `min(x)`
- `y(:,:,...:,end)` gives the slopes at the end of the intervals located at `max(x)`

Data Types: `single` | `double`

xq — Query points
scalar | vector | matrix | array

Query points, specified as a scalar, vector, matrix, or array. The points specified in `xq` are the *x*-coordinates for the interpolated function values `yq` computed by `spline`.

Data Types: `single` | `double`

Output Arguments

s — Interpolated values at query points
scalar | vector | matrix | array

Interpolated values at query points, returned as a scalar, vector, matrix, or array.

The size of `s` is related to the sizes of `y` and `xq`:

- If `y` is a vector, then `s` has the same size as `xq`.
- If `y` is an array of size `Ny = size(y)`, then these conditions apply:
  - If `xq` is a scalar or vector, then `size(s)` returns `[Ny(1:end-1) length(xq)]`.
  - If `xq` is an array, then `size(s)` returns `[Ny(1:end-1) size(xq)]`.

pp — Piecewise polynomial
structure

Piecewise polynomial, returned as a structure. Use this structure with the `ppval` function to evaluate the piecewise polynomial at one or more query points. The structure has these fields.

Field	Description
`form`	`'pp'` for *piecewise polynomial*
`breaks`	Vector of length `L+1` with strictly increasing elements that represent the start and end of each of `L` intervals
`coefs`	`L`-by-`k` matrix with each row `coefs(i,:)` containing the local coefficients of an order `k` polynomial on the `i`th interval, `[breaks(i),breaks(i+1)]`
`pieces`	Number of pieces, `L`
`order`	Order of the polynomials
`dim`	Dimensionality of target

Since the polynomial coefficients in `coefs` are local coefficients for each interval, you must subtract the lower endpoint of the corresponding knot interval to use the coefficients in a conventional polynomial equation. In other words, for the coefficients `[a,b,c,d]` on the interval `[x1,x2]`, the corresponding polynomial is

$$f(x) = a(x - x_1)^3 + b(x - x_1)^2 + c(x - x_1) + d \ .$$

Tips

- You also can perform spline interpolation using the `interp1` function with the command `interp1(x,y,xq,'spline')`. While `spline` performs interpolation on rows of an input matrix, `interp1` performs interpolation on columns of an input matrix.

Algorithms

A tridiagonal linear system (possibly with several right-hand sides) is solved for the information needed to describe the coefficients of the various cubic polynomials that make up the interpolating spline. `spline` uses the functions `ppval`, `mkpp`, and `unmkpp`. These routines form a small suite of functions for working with piecewise polynomials. For access to more advanced features, see `interp1` or the Curve Fitting Toolbox™ spline functions.

References

[1] de Boor, Carl. *A Practical Guide to Splines*. Springer-Verlag, New York: 1978.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Input `x` must be strictly increasing.
- Code generation does not remove `y` entries with `NaN` values.
- Code generation does not report an error for infinite endslopes in `y`.
- If you generate code for the `pp = spline(x,y)` syntax, then you cannot input `pp` to the `ppval` function in MATLAB. To create a MATLAB `pp` structure from a `pp` structure created by the code generator:
  - In code generation, use `unmkpp` to return the piecewise polynomial details to MATLAB.
  - In MATLAB, use `mkpp` to create the `pp` structure.
- If you supply `xq`, and if `y` has a variable-size and is not a variable-length vector, then the orientation of vector outputs in the generated code might not match the orientation in MATLAB.

See Also

`interp1` | `makima` | `pchip` | `ppval`

Introduced before R2006a

split

Split strings at delimiters

Syntax

```
newStr = split(str)
newStr = split(str,delimiter)
newStr = split(str,delimiter,dim)
[newStr,matches] = split( ___ )
```

Description

`newStr = split(str)` divides `str` at whitespace characters and returns the result as the output array `newStr`. The input array `str` can be a string array, a character vector, or a cell array of character vectors. If `str` is a string array, then so is `newStr`. Otherwise, `newStr` is a cell array of character vectors. `newStr` does not include the whitespace characters from `str`.

If `str` is a string array or cell array of character vectors, and has multiple elements, then each element must be divisible into the same number of substrings.

- If `str` is a string scalar or character vector, then `newStr` is an `N`-by-`1` string array or cell array of character vectors, where `N` is the number of substrings.
- If `str` is an `M`-by-`1` string array or cell array, then `newStr` is an `M`-by-`N` array.
- If `str` is a `1`-by-`M` string array or cell array, then `newStr` is an `1`-by-`M`-by-`N` array.

For a string array or cell array of any size, `split` orients the `N` substrings along the first trailing dimension with a size of `1`.

If the number of substrings is not the same for every element of `str`, then call `split` in a `for`-loop to divide the elements of `str` one at a time.

`newStr = split(str,delimiter)` divides each element of `str` at the delimiters specified by `delimiter`. The output `newStr` does not include the delimiters.

`newStr = split(str,delimiter,dim)` divides each element of `str` into a vector oriented along the dimension specified by `dim`.

`[newStr,matches] = split( ___ )` additionally returns an array, `matches`, that contains all occurrences of delimiters at which the `split` function splits `str`. You can use this syntax with any of the input arguments of the previous syntaxes.

Examples

Split Strings at Whitespace and Rejoin Them

Split names in a string array at whitespace characters. Then reorder the strings and join them so that the last names precede the first names.

Create a 3-by-1 string array containing names. Starting in R2017a, you can create strings using double quotes.

```
names = ["Mary Butler";
         "Santiago Marquez";
         "Diana Lee"]
```

```
names = 3x1 string array
    "Mary Butler"
    "Santiago Marquez"
    "Diana Lee"
```

Split `names` at whitespace characters, making it a 3-by-2 string array.

```
names = split(names)
```

```
names = 3x2 string array
    "Mary"        "Butler"
    "Santiago"    "Marquez"
    "Diana"       "Lee"
```

Switch the columns of `names` so that the last names are in the first column. Add a comma after each last name.

```
names = [names(:,2) names(:,1)];
names(:,1) = names(:,1) + ','
```

```
names = 3x2 string array
    "Butler,"     "Mary"
    "Marquez,"    "Santiago"
    "Lee,"        "Diana"
```

Join the last and first names. The `join` function places a space character between the strings it joins. After the join, `names` is a 3-by-1 string array.

```
names = join(names)
```

```
names = 3x1 string array
    "Butler, Mary"
    "Marquez, Santiago"
    "Lee, Diana"
```

Split String at Delimiter and Join with New Delimiter

Create a string that contains the path to a folder. Starting in R2017a, you can create strings using double quotes.

```
myPath = "/Users/jdoe/My Documents/Examples"
```

```
myPath =
"/Users/jdoe/My Documents/Examples"
```

Split the path at the `/` character. `split` returns `myFolders` as a 5-by-1 string array. The first string is `""` because `myPath` starts with the `/` character.

```
myFolders = split(myPath,"/")
```

```
myFolders = 5x1 string array
    ""
    "Users"
    "jdoe"
    "My Documents"
    "Examples"
```

Join `myFolders` into a new path with `\` as the delimiter. Add `C:` as the beginning of the path.

```
myNewPath = join(myFolders,"\");
myNewPath = 'C:' + myNewPath
```

```
myNewPath =
"C:\Users\jdoe\My Documents\Examples"
```

Split String at Multiple Delimiters

Create a string. Starting in R2017a, you can create strings using double quotes.

```
str = "A horse! A horse! My kingdom for a horse!"
```

```
str =
"A horse! A horse! My kingdom for a horse!"
```

Split `str` at exclamation points and at whitespace characters. `newStr` is a 10-by-1 string array. The last string is an empty string, `""`, because the last character in `str` is a delimiter.

```
newStr = split(str,[" ","!"])
```

```
newStr = 12x1 string array
    "A"
    "horse"
    ""
    "A"
    "horse"
    ""
    "My"
    "kingdom"
    "for"
    "a"
    "horse"
    ""
```

Split String Array with Missing Data Between Delimiters

Create a string array in which each element contains comma-delimited data about a patient. Starting in R2017a, you can create strings using double quotes.

```
patients = ["LastName,Age,Gender,Height,Weight";
            "Adams,47,F,64,123";
            "Jones,,,68,175";
            "King,,M,66,180";
            "Smith,38,F,63,118"]
```

```
patients = 5x1 string array
    "LastName,Age,Gender,Height,Weight"
    "Adams,47,F,64,123"
    "Jones,,,68,175"
    "King,,M,66,180"
    "Smith,38,F,63,118"
```

Split the string array. A pair of commas with nothing between them indicates missing data. When `split` divides on repeated delimiters, it returns empty strings as corresponding elements of the output array.

```
patients = split(patients,",")
```

```
patients = 5x5 string array
    "LastName"    "Age"    "Gender"    "Height"    "Weight"
    "Adams"       "47"     "F"         "64"        "123"
    "Jones"       ""       ""          "68"        "175"
    "King"        ""       "M"         "66"        "180"
    "Smith"       "38"     "F"         "63"        "118"
```

Orient Strings Along Specified Dimension

Create a 3-by-1 string array containing names. Starting in R2017a, you can create strings using double quotes.

```
names = ["Mary Butler";
         "Santiago Marquez";
         "Diana Lee"]
```

```
names = 3x1 string array
    "Mary Butler"
    "Santiago Marquez"
    "Diana Lee"
```

Split the array at whitespace characters. By default, `split` orients the output substrings along the first trailing dimension with a size of 1. Because `names` is a 3-by-1 string array, `split` orients the substrings along the second dimension of `splitNames`, that is, the columns.

```
splitNames = split(names)
```

```
splitNames = 3x2 string array
    "Mary"        "Butler"
    "Santiago"    "Marquez"
    "Diana"       "Lee"
```

To orient the substrings along the rows, or first dimension, specify the dimension after you specify the delimiter. `splitNames` is now a 2-by-3 string array, with the first names in the first row and the last names in the second row.

```
splitNames = split(names," ",1)
```

```
splitNames = 2x3 string array
    "Mary"      "Santiago"    "Diana"
    "Butler"    "Marquez"     "Lee"
```

Split String and Return Delimiters

Create a string. Starting in R2017a, you can create strings using double quotes.

```
str = "bacon, lettuce, and tomato"
```

```
str =
"bacon, lettuce, and tomato"
```

Split `str` on delimiters. Return the results of the split in a string array, and the delimiters in a second string array. When there is no text between consecutive delimiters, `split` returns an empty string.

```
[newStr,matches] = split(str,["and",","," "])
```

```
newStr = 7x1 string array
    "bacon"
    ""
```

```
    "lettuce"
    ""
    ""
    ""
    "tomato"


matches = 6x1 string array
    ","
    " "
    ","
    " "
    "and"
    " "
```

Join `newStr` and `matches` back together with the `join` function.

```
originalStr = join(newStr,matches)
```

```
originalStr =
"bacon, lettuce, and tomato"
```

Input Arguments

`str` — Input text
string array | character vector | cell array of character vectors

Input text, specified as a string array, a character vector, or a cell array of character vectors.

Data Types: `string` | `char` | `cell`

`delimiter` — Delimiting substrings
string array | character vector | cell array of character vectors

Delimiting substrings, specified as a string array, a character vector, or a cell array of character vectors. The substrings specified in `delimiter` do not appear in the output `newStr`.

Specify multiple delimiters in a string array or a cell array of character vectors. The `split` function splits `str` on the elements of `delimiter`. The order in which delimiters

appear in `delimiter` does not matter unless multiple delimiters begin a match at the same character in `str`. In that case, the `split` function splits on the first matching delimiter in `delimiter`.

Example: `split(str,{' ',',','--'})` splits `str` on spaces, commas, and pairs of consecutive dashes.

Data Types: `string` | `char` | `cell`

`dim` — Dimension along which to split strings
positive integer

Dimension along which to split strings, specified as a positive integer. If `dim` is not specified, then the default is the last array dimension with a size that does not equal `1`.

Output Arguments

`newStr` — Substrings split out of original array
string array | cell array of character vectors

Substrings split out of original array, returned as a string array or a cell array of character vectors. If the input array `str` is a string array, then so is `newStr`. Otherwise, `newStr` is a cell array of character vectors.

Data Types: `string`

`matches` — Identified delimiters
string array | cell array of character vectors

Identified delimiters, returned as a string array or a cell array of character vectors. If the input array `str` is a string array, then so is `matches`. Otherwise, `matches` is a cell array of character vectors.

`matches` always contains one fewer element than output `newStr` contains.

Data Types: `string`

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

See Also

`compose` | `join` | `newline` | `regexp` | `splitlines` | `sprintf` | `string`

Topics

“Create String Arrays”
“Analyze Text Data with String Arrays”
“Regular Expressions”

Introduced in R2016b

splitlines

Split strings at newline characters

Syntax

```
newStr = splitlines(str)
```

Description

`newStr = splitlines(str)` splits `str` at newline characters and returns the result as the output array `newStr`.

`splitlines` splits at actual newline characters, not at the literal `\n`. To split a string that contains `\n`, first use `compose` and then use `splitlines`.

Examples

Split String at Newline

Split a string at a newline character. When the literal `\n` represents a newline character, convert it to an actual newline using the `compose` function. Then use `splitlines` to split the string at the newline character.

Create a string in which two lines of text are separated by `\n`. You can use `+` to concatenate text onto the end of a string. Starting in R2017a, you can create strings using double quotes.

```
str = "In Xanadu did Kubla Khan";
str = str + "\n" + "A stately pleasure-dome decree"
```

```
str =
"In Xanadu did Kubla Khan\nA stately pleasure-dome decree"
```

Convert `\n` into an actual newline character. Although `str` displays on two lines, `str` is a 1-by-1 string containing both lines of text.

```
str = compose(str)
```

```
str = 
    "In Xanadu did Kubla Khan
     A stately pleasure-dome decree"
```

Split `str` at the newline character. `newStr` is a 1-by-2 string array. Each element contains one line of the text.

```
newStr = splitlines(str)
```

```
newStr = 2x1 string array
    "In Xanadu did Kubla Khan"
    "A stately pleasure-dome decree"
```

Split Character Vector at Newlines

Create a character vector and split it at newline characters. The `newline` function returns the newline character, `char(10)`.

```
chr = 'Whose woods these are I think I know.';
chr = [chr newline 'His house is in the village though;']
```

```
chr = 
    'Whose woods these are I think I know.
     His house is in the village though;'
```

```
C = splitlines(chr)
```

```
C = 2x1 cell array
    {'Whose woods these are I think I know.'}
    {'His house is in the village though;'  }
```

Input Arguments

`str` — Input text

string array | character vector | cell array of character vectors

Input text, specified as a string array, a character vector, or a cell array of character vectors. If `str` is a string array or cell array of character vectors, then each element of `str` must contain the same number of newlines.

Output Arguments

`newStr` — Output text
string array | cell array of character vectors

Output text, returned as a string array or a cell array of character vectors. `newStr` has one more dimension than `str`. The size of the new dimension is one more than the number of newlines in a string element. `splitlines` assigns the results of the split along the new dimension. For example, if `str` is a 2-by-3 string array, and each string has three newline characters, then `newStr` is a 2-by-3-by-4 array.

If the input array `str` is a string array, then so is `newStr`. Otherwise, `newStr` is a cell array of character vectors.

Tips

If the elements of a string array have different numbers of newline characters, use a `for`-loop to access the string elements individually and split them.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

See Also

`compose` | `join` | `newline` | `regexp` | `split` | `sprintf` | `string`

Topics

"Create String Arrays"
"Analyze Text Data with String Arrays"

Introduced in R2016b

splitapply

Split data into groups and apply function

Syntax

```
Y = splitapply(func,X,G)
Y = splitapply(func,X1,...,XN,G)
Y = splitapply(func,T,G)
[Y1,...,YM] = splitapply( ___ )
```

Description

`Y = splitapply(func,X,G)` splits `X` into groups specified by `G` and applies the function `func` to each group. `splitapply` returns `Y` as an array that contains the concatenated outputs from `func` for the groups split out of `X`. The input argument `G` is a vector of positive integers that specifies the groups to which corresponding elements of `X` belong. If `G` contains `NaN` values, `splitapply` omits the corresponding values in `X` when it splits `X` into groups. To create `G`, you can use the `findgroups` function.

`splitapply` combines two steps in the "Split-Apply-Combine Workflow" on page 1-14357.

`Y = splitapply(func,X1,...,XN,G)` splits `X1,...,XN` into groups and applies `func`. The `splitapply` function calls `func` once per group, with corresponding elements from `X1,...,XN` as the `N` input arguments to `func`.

`Y = splitapply(func,T,G)` splits variables of table `T` into groups and applies `func`. The `splitapply` function treats the variables of `T` as vectors, matrices, or cell arrays, depending on the data types of the table variables. If `T` has `N` variables, then `func` must accept `N` input arguments.

`[Y1,...,YM] = splitapply( ___ )` splits variables into groups and applies `func` to each group. `func` returns multiple output arguments. `Y1,...,YM` contains the concatenated outputs from `func` for the groups split out of the input data variables. `func` can return output arguments that belong to different classes, but the class of each output

must be the same each time `func` is called. You can use this syntax with any of the input arguments of the previous syntaxes.

The number of output arguments from `func` need not be the same as the number of input arguments specified by `X1,...,XN`.

Examples

Calculate Mean Values of Groups of Data

Calculate the mean heights by gender for groups of patients and display the results.

Load patient heights and genders from the data file `patients.mat`.

```
load patients
whos Gender Height
```

```
  Name          Size            Bytes  Class     Attributes

  Gender      100x1             12212  cell
  Height      100x1               800  double
```

Specify groups by gender with `findgroups`.

```
G = findgroups(Gender);
```

Split `Height` into groups specified by `G`. Calculate the mean height by gender. The first row of the output argument is the mean height of the female patients, and the second row is the mean height of the male patients.

```
splitapply(@mean,Height,G)
```

```
ans = 2×1

   65.1509
   69.2340
```

Split Two Data Variables and Apply Function

Calculate the variances of the differences in blood pressure readings for groups of patients, and display the results. The blood pressure readings are contained in two data variables. To calculate the differences, use a function that takes two input arguments.

Load blood pressure readings and smoking data for 100 patients from the data file `patients.mat`.

```
load patients
whos Systolic Diastolic Smoker
```

```
  Name             Size            Bytes  Class      Attributes

  Diastolic      100x1               800  double
  Smoker         100x1               100  logical
  Systolic       100x1               800  double
```

Define `func` as a function that calculates the variances of the differences between systolic and diastolic blood-pressure readings for smokers and nonsmokers. `func` requires two input arguments.

```
func = @(x,y) var(x-y);
```

Use `findgroups` and `splitapply` to split the patient data into groups and calculate the variances of the differences. `findgroups` also returns group identifiers in `smokers`. The `splitapply` function calls `func` once per group, with `Systolic` and `Diastolic` as the two input arguments.

```
[G,smokers] = findgroups(Smoker);
varBP = splitapply(func,Systolic,Diastolic,G)
```

```
varBP = 2×1

   44.4459
   48.6783
```

Create a table that contains the variances of the differences, with the number of patients in each group.

```
numPatients = splitapply(@numel,Smoker,G);
T = table(smokers,numPatients,varBP)
```

```
T=2×3 table
    smokers    numPatients    varBP
```

```
_______    ___________    ______

 false         66         44.446
 true          34         48.678
```

Return Nonscalar Output for Groups

Calculate the minimum, median, and maximum weights for groups of patients and return these results as arrays for each group. `splitapply` concatenates the output arguments so that you can distinguish output for each group from output for the other groups.

Define a function that returns the minimum, median, and maximum as a row vector.

```
mystats = @(x)[min(x) median(x) max(x)];
```

Load patient weights, genders, and status as smokers from `patients.mat`.

```
load patients
whos Weight Gender Smoker
```

```
  Name          Size            Bytes  Class      Attributes

  Gender      100x1             12212  cell
  Smoker      100x1               100  logical
  Weight      100x1               800  double
```

Use `findgroups` and `splitapply` to split the patient weights into groups and calculate statistics for each group.

```
G = findgroups(Gender,Smoker);
Y = splitapply(mystats,Weight,G)
```

```
Y = 4×3

  111.0000  131.0000  147.0000
  115.0000  131.0000  146.0000
  158.0000  181.5000  194.0000
  164.0000  181.0000  202.0000
```

In this example, you can return nonscalar output as row vectors because the data and grouping variables are column vectors. Each row of Y contains statistics for a different group of patients.

Split Table Data Variables and Apply Function

Calculate the mean body-mass-index (BMI) from tables of patient data. Group the patients by gender and status as smokers or nonsmokers.

Load patient data and grouping variables into tables.

```
load patients
DT = table(Height,Weight);
GT = table(Gender,Smoker);
```

Define a function that calculates mean BMI from the weights and heights of groups or patients.

```
meanBMIFcn = @(h,w)mean((w ./ (h.^2)) * 703);
```

Create a table that contains the mean BMI for each group.

```
[G,results] = findgroups(GT);
meanBMI = splitapply(meanBMIFcn,DT,G);
results.meanBMI = meanBMI
```

```
results=4×3 table
      Gender      Smoker    meanBMI
    __________    ______    _______

    {'Female'}    false     21.672
    {'Female'}    true      21.669
    {'Male'  }    false     26.578
    {'Male'  }    true      26.458
```

Return Multiple Statistics for Groups

Calculate the minimum, mean, and maximum heights for groups of patients and return results in a table.

Define a function in a file named `multiStats.m` that accepts an input vector and returns the minimum, mean, and maximum values of the vector.

```
% Copyright 2015 The MathWorks, Inc.

function [lo,avg,hi] = multiStats(x)
lo = min(x);
avg = mean(x);
hi = max(x);
end
```

Load patient data into a table.

```
load patients
T = table(Gender,Height);
summary(T)
```

```
Variables:

    Gender: 100x1 cell array of character vectors

    Height: 100x1 double

        Values:

            Min          60
            Median       67
            Max          72
```

Group patient heights by gender. Create a table that contains the outputs from `multiStats` for each group.

```
[G,gender] = findgroups(T.Gender);
[minHeight,meanHeight,maxHeight] = splitapply(@multiStats,T.Height,G);
result = table(gender,minHeight,meanHeight,maxHeight)
```

```
result =

  2x4 table

      gender      minHeight    meanHeight    maxHeight
    __________    _________    __________    _________

    {'Female'}       60          65.151         70
    {'Male'  }       66          69.234         72
```

Input Arguments

func — Function to apply to groups of data
function handle

Function to apply to groups of data, specified as a function handle.

If `func` returns a nonscalar output argument, then the argument must be oriented so that `splitapply` can concatenate the output arguments from successive calls to `func`. For example, if the input data variables are column vectors, then `func` must return either a scalar or a row vector as an output argument.

Example: `Y = splitapply(@sum,X,G)` returns the sums of the groups of data in `X`.

X — Data variable
vector | matrix | cell array

Data variable, specified as a vector, matrix, or cell array. The elements of `X` belong to groups specified by the corresponding elements of `G`.

If `X` is a matrix, `splitapply` treats each column or row as a separate data variable. The orientation of `G` determines whether `splitapply` treats the columns or rows of `X` as data variables.

G — Group numbers
vector of positive integers

Group numbers, specified as a vector of positive integers.

- If X is a vector or cell array, then G must be the same length as X.
- If X is a matrix, then the length of G must be equal to the number of columns or rows of X, depending on the orientation of G.
- If the input argument is table T, then G must be a column vector. The length of G must be equal to the number of rows of T.

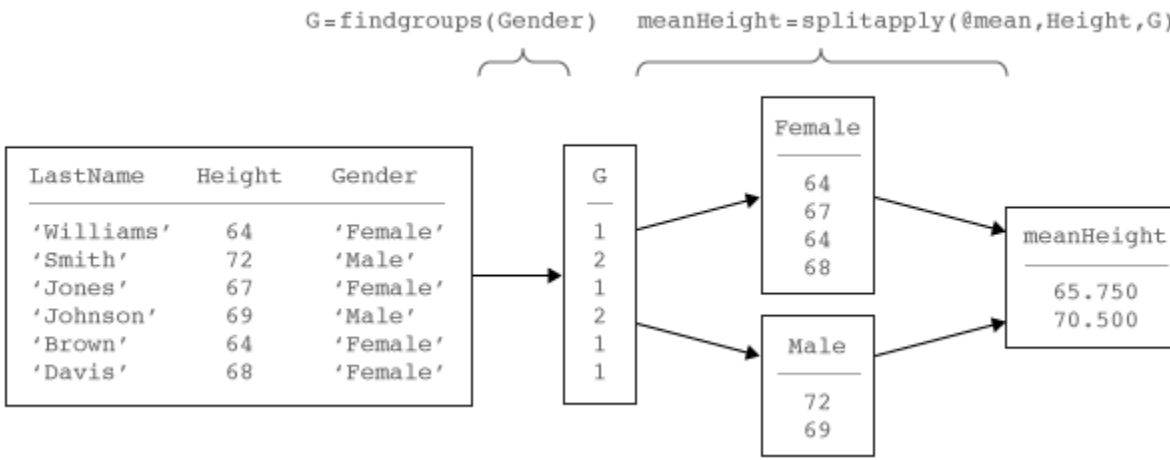
T — Data variables
table

Data variables, specified as a table. `splitapply` treats each table variable as a separate data variable.

More About

Split-Apply-Combine Workflow

The *Split-Apply-Combine* workflow is common in data analysis. In this workflow, the analyst splits the data into groups, applies a function to each group, and combines the results. The diagram shows a typical example of the workflow and the parts of the workflow implemented by `findgroups` and `splitapply`.



Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

The specified function must not rely on any state, such as `persistent` variables or random number functions like `rand`.

For more information, see “Tall Arrays”.

See Also

`accumarray` | `arrayfun` | `discretize` | `findgroups` | `groupsummary` | `histcounts` | `rowfun` | `unique` | `varfun`

Topics

“Split Table Data Variables and Apply Functions”
“Split Data into Groups and Calculate Statistics”
“Calculations on Tables”
“Grouping Variables To Split Data”

Introduced in R2015b

splitvars

Split multicolumn variables in table or timetable

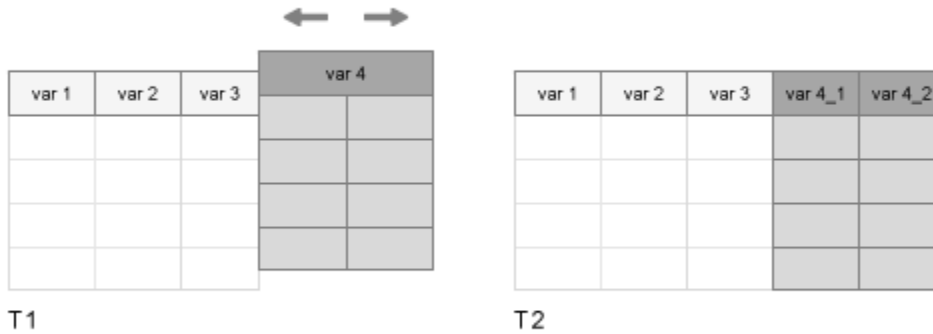
Syntax

```
T2 = splitvars(T1)
T2 = splitvars(T1,vars)
T2 = splitvars( ___ ,'NewVariableNames',newNames)
```

Description

`T2 = splitvars(T1)` splits all multicolumn variables in `T1` so that they are single-column variables in `T2` (see diagram). All single-column variables from `T1` are unaltered.

- If a variable in `T1` has multiple columns, then `splitvars` makes unique names for the new variables in `T2` from the name of the original variable in `T1`.
- If a variable in `T1` is a table itself, then `splitvars` uses the names of its variables (and, if necessary, the name of that table) to make unique names for the new variables in `T2`.



To merge variables into one multicolumn variable, use the `mergevars` function.

`T2 = splitvars(T1,vars)` splits only the table variables specified by `vars`. You can specify variables by name, by position, or using logical indices.

`T2 = splitvars( ___ ,'NewVariableNames',newNames)` specifies `newNames` as the names of the variables that are split and copied to `T2`.

Examples

Split Variables

Create a table from workspace variables. Some of the variables are matrices with multiple columns.

```
A = (1:3)';
B = [5 11 12; 20 30 50; 0.1 3.4 5.9]';
C = {'a','XX';'b','YY';'c','ZZ'};
D = [128 256 512]';
T1 = table(A,B,C,D)
```

```
T1=3×4 table
    A           B                    C            D
    _    _________________    _________________    ___

    1     5     20    0.1    {'a'}    {'XX'}    128
    2    11     30    3.4    {'b'}    {'YY'}    256
    3    12     50    5.9    {'c'}    {'ZZ'}    512
```

Split the variables `B` and `C`. All variables in the output table have one column.

```
T2 = splitvars(T1)
```

```
T2=3×7 table
    A    B_1    B_2    B_3     C_1      C_2       D
    _    ___    ___    ___    _____    ______    ___

    1     5     20     0.1    {'a'}    {'XX'}    128
    2    11     30     3.4    {'b'}    {'YY'}    256
    3    12     50     5.9    {'c'}    {'ZZ'}    512
```

Specify Variable

Create a table that contains tables, using arrays of data from the `patients.mat` file. Display the first three rows.

```
load patients
Personal_Data = table(Gender,Age);
BMI_Data = table(Height,Weight);
BloodPressure = table(Systolic,Diastolic);
T1 = table(LastName,Personal_Data,BMI_Data,BloodPressure);
head(T1,3)
```

```
ans=3×4 table
      LastName        Personal_Data          BMI_Data            BloodPressure
                        Gender      Age    Height    Weight    Systolic    Diastolic
    ____________    ________________    ________________    ______________________

    {'Smith'   }    {'Male'  }    38       71       176        124          93
    {'Johnson' }    {'Male'  }    43       69       163        109          77
    {'Williams'}    {'Female'}    38       64       131        125          83
```

Specify `BloodPressure` as the variable to split.

```
T2 = splitvars(T1,'BloodPressure');
head(T2,3)
```

```
ans=3×5 table
      LastName        Personal_Data          BMI_Data        Systolic    Diastolic
                        Gender      Age    Height    Weight
    ____________    ________________    ________________    ________    _________

    {'Smith'   }    {'Male'  }    38       71       176        124          93
    {'Johnson' }    {'Male'  }    43       69       163        109          77
    {'Williams'}    {'Female'}    38       64       131        125          83
```

To specify multiple variables by name, use a cell array of character vectors.

```
T3 = splitvars(T1,{'BMI_Data','BloodPressure'});
head(T3,3)
```

```
ans=3×6 table
      LastName        Personal_Data      Height    Weight    Systolic    Diastolic
                      Gender      Age
    ____________    ________________    ______    ______    ________    _________

    {'Smith'   }    {'Male'  }    38       71       176        124          93
    {'Johnson' }    {'Male'  }    43       69       163        109          77
    {'Williams'}    {'Female'}    38       64       131        125          83
```

To specify variables by position, use a numeric array.

```
T4 = splitvars(T1,[2 4]);
head(T4,3)
```

```
ans=3×6 table
      LastName        Gender      Age        BMI_Data        Systolic    Diastolic
                                         Height    Weight
    ____________    __________    ___    ________________    ________    _________

    {'Smith'   }    {'Male'  }    38       71       176        124          93
    {'Johnson' }    {'Male'  }    43       69       163        109          77
    {'Williams'}    {'Female'}    38       64       131        125          83
```

New Names for Split Variables

Create a table that contains multi-column variables, using data from the `patients.mat` file. Display the first three rows.

```
load patients
Personal_Data = [Age,Height,Weight];
BloodPressure = [Systolic,Diastolic];
T1 = table(LastName,Gender,Personal_Data,BloodPressure);
head(T1,3)
```

```
ans=3×4 table
      LastName        Gender       Personal_Data      BloodPressure
    ____________    __________    ________________    _____________

    {'Smith'   }    {'Male'  }    38     71    176     124     93
    {'Johnson' }    {'Male'  }    43     69    163     109     77
```

```
    {'Williams'}    {'Female'}    38     64    131     125     83
```

Split `BloodPressure` and specify new names for the new variables in the output table.

```
T2 = splitvars(T1,'BloodPressure','NewVariableNames',{'Systolic','Diastolic'});
head(T2,3)
```

```
ans=3×5 table
      LastName        Gender       Personal_Data      Systolic    Diastolic
    ____________    __________    _______________    ________    _________

    {'Smith'   }    {'Male'  }    38     71    176      124          93
    {'Johnson' }    {'Male'  }    43     69    163      109          77
    {'Williams'}    {'Female'}    38     64    131      125          83
```

Split both `BMI_Data` and `BloodPressure`. For each variable being split, you must provide a cell array with the correct number of new names.

```
T3 = splitvars(T1,{'Personal_Data','BloodPressure'},...
     'NewVariableNames',{{'Age','Height','Weight'},{'Systolic','Diastolic'}});
head(T3,3)
```

```
ans=3×7 table
      LastName        Gender      Age    Height    Weight    Systolic    Diastolic
    ____________    __________    ___    ______    ______    ________    _________

    {'Smith'   }    {'Male'  }    38       71       176        124          93
    {'Johnson' }    {'Male'  }    43       69       163        109          77
    {'Williams'}    {'Female'}    38       64       131        125          83
```

Input Arguments

T1 — Input table

table | timetable

Input table, specified as a table or timetable.

vars — Variables in input table

character vector | cell array of character vectors | string array | numeric array | logical array

Variables in the input table, specified as a character vector, cell array of character vectors, string array, numeric array, or logical array.

newNames — Names of split variables
cell array of character vectors | string array

Names of the split variables, specified as a cell array of character vectors or string array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

See Also

`addvars` | `mergevars` | `movevars` | `removevars`

Topics

“Add, Delete, and Rearrange Table Variables”
“Add and Delete Table Rows”
“Access Data in Tables”
“Modify Units, Descriptions, and Table Variable Names”
“Clean Messy and Missing Data in Tables”

Introduced in R2018a

spones

Replace nonzero sparse matrix elements with ones

Syntax

```
R = spones(S)
```

Description

`R = spones(S)` generates a matrix `R` with the same sparsity structure as `S`, but with `1`'s in the nonzero positions.

Examples

`c = sum(spones(S))` is the number of nonzeros in each column.

`r = sum(spones(S'))'` is the number of nonzeros in each row.

`sum(c)` and `sum(r)` are equal, and are equal to `nnz(S)`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`nnz` | `spalloc` | `spfun`

Introduced before R2006a

spparms

Set parameters for sparse matrix routines

Syntax

```
spparms('key',value)
spparms
values = spparms
[keys,values] = spparms
spparms(values)
value = spparms('key')
spparms('default')
spparms('tight')
```

Description

`spparms('key',value)` sets one or more of the *tunable* parameters used in the sparse routines. In ordinary use, you should never need to deal with this function.

The meanings of the `key` parameters are

`'spumoni'`	Sparse Monitor flag:
`0`	Produces no diagnostic output, the default
`1`	Produces information about choice of algorithm based on matrix structure, and about storage allocation
`2`	Also produces very detailed information about the sparse matrix algorithms
`'thr_rel'`, `'thr_abs'`	Minimum degree threshold is `thr_rel*mindegree + thr_abs`.
`'exact_d'`	Nonzero to use exact degrees in minimum degree. Zero to use approximate degrees.

`'supernd'`	If positive, minimum degree amalgamates the supernodes every `supernd` stages.
`'rreduce'`	If positive, minimum degree does row reduction every `rreduce` stages.
`'wh_frac'`	Rows with `density > wh_frac` are ignored in `colmmd`.
`'autommd'`	Nonzero to use minimum degree (MMD) orderings with QR-based \ and /.
`'autoamd'`	Nonzero to use `colamd` ordering with the LU-based \ and /, and to use `amd` with Cholesky-based \ and /.
`'piv_tol'`	Pivot tolerance used by the LU-based \ and /.
`'bandden'`	Band density used by \ and / for banded matrices. Band density is defined as (# nonzeros in the band)/(# nonzeros in a full band). If `bandden = 1.0`, never use band solver. If `bandden = 0.0`, always use band solver. Default is `0.5`.
`'umfpack'`	Nonzero to use UMFPACK instead of the v4 LU-based solver in \ and /.
`'sym_tol'`	Symmetric pivot tolerance. See `lu` for more information about the role of the symmetric pivot tolerance.

`spparms`, by itself, prints a description of the current settings.

`values = spparms` returns a vector whose components give the current settings.

`[keys,values] = spparms` returns that vector, and also returns a character matrix whose rows are the keywords for the parameters.

`spparms(values)`, with no output argument, sets all the parameters to the values specified by the argument vector.

`value = spparms('key')` returns the current setting of one parameter.

`spparms('default')` sets all the parameters to their default settings.

`spparms('tight')` sets the minimum degree ordering parameters to their *tight* settings, which can lead to orderings with less fill-in, but which make the ordering functions themselves use more execution time.

The key parameters for `default` and `tight` settings are

	Keyword	Default	Tight
values(1)	'spumoni'	0.0	
values(2)	'thr_rel'	1.1	1.0
values(3)	'thr_abs'	1.0	0.0
values(4)	'exact_d'	0.0	1.0
values(5)	'supernd'	3.0	1.0
values(6)	'rreduce'	3.0	1.0
values(7)	'wh_frac'	0.5	0.5
values(8)	'autommd'	1.0	
values(9)	'autoamd'	1.0	
values(10)	'piv_tol'	0.1	
values(11)	'bandden'	0.5	
values(12)	'umfpack'	1.0	
values(13)	'sym_tol'	0.001	

See Also

chol | colamd | lu | qr | symamd

Introduced before R2006a

sprand

Sparse uniformly distributed random matrix

Syntax

```
R = sprand(S)
R = sprand(m,n,density)
R = sprand(m,n,density,rc)
```

Description

`R = sprand(S)` has the same sparsity structure as `S`, but uniformly distributed random entries.

`R = sprand(m,n,density)` is a random, `m`-by-`n`, sparse matrix with approximately `density*m*n` uniformly distributed nonzero entries (`0 <= density <= 1`).

`R = sprand(m,n,density,rc)` also has reciprocal condition number approximately equal to `rc`. `R` is constructed from a sum of matrices of rank one.

If `rc` is a vector of length `lr`, where `lr <= min(m,n)`, then `R` has `rc` as its first `lr` singular values, all others are zero. In this case, `R` is generated by random plane rotations applied to a diagonal matrix with the given singular values. It has a great deal of topological and algebraic structure.

Tips

- `sprand` uses the same random number generator as `rand`, `randi`, and `randn`. You control this generator with `rng`.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `rc` argument is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `distributed.sprand` in the Parallel Computing Toolbox documentation.

See Also

`sprandn` | `sprandsym`

Introduced before R2006a

sprandn

Sparse normally distributed random matrix

Syntax

```
R = sprandn(S)
R = sprandn(m,n,density)
R = sprandn(m,n,density,rc)
```

Description

`R = sprandn(S)` has the same sparsity structure as `S`, but normally distributed random entries with mean `0` and variance `1`.

`R = sprandn(m,n,density)` is a random, `m`-by-`n`, sparse matrix with approximately `density*m*n` normally distributed nonzero entries (`0 <= density <= 1`).

`R = sprandn(m,n,density,rc)` also has reciprocal condition number approximately equal to `rc`. `R` is constructed from a sum of matrices of rank one.

If `rc` is a vector of length `lr`, where `lr <= min(m,n)`, then `R` has `rc` as its first `lr` singular values, all others are zero. In this case, `R` is generated by random plane rotations applied to a diagonal matrix with the given singular values. It has a great deal of topological and algebraic structure.

Tips

- `sprandn` uses the same random number generator as `rand`, `randi`, and `randn`. You control this generator with `rng`.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `rc` argument is not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `distributed.sprandn` in the Parallel Computing Toolbox documentation.

See Also

`sprand` | `sprandsym`

Introduced before R2006a

sprandsym

Sparse symmetric random matrix

Syntax

```
R = sprandsym(S)
R = sprandsym(n,density)
R = sprandsym(n,density,rc)
R = sprandsym(n,density,rc,kind)
R = sprandsym(S,[],rc,3)
```

Description

`R = sprandsym(S)` returns a symmetric random matrix whose lower triangle and diagonal have the same structure as `S`. Its elements are normally distributed, with mean `0` and variance `1`.

`R = sprandsym(n,density)` returns a symmetric random, `n`-by-`n`, sparse matrix with approximately `density*n*n` nonzeros; each entry is the sum of one or more normally distributed random samples, and (`0 <= density <= 1`).

`R = sprandsym(n,density,rc)` returns a matrix with a reciprocal condition number equal to `rc`. The distribution of entries is nonuniform; it is roughly symmetric about 0; all are in [−1,1].

If `rc` is a vector of length `n`, then `R` has eigenvalues `rc`. Thus, if `rc` is a positive (nonnegative) vector then `R` is a positive (nonnegative) definite matrix. In either case, `R` is generated by random Jacobi rotations applied to a diagonal matrix with the given eigenvalues or condition number. It has a great deal of topological and algebraic structure.

`R = sprandsym(n,density,rc,kind)` is positive definite.

- If kind = 1, `R` is generated by random Jacobi rotation of a positive definite diagonal matrix. `R` has the desired condition number exactly.

- If kind = 2, R is a shifted sum of outer products. R has the desired condition number only approximately, but has less structure.

`R = sprandsym(S,[],rc,3)` has the same structure as the matrix `S` and approximate condition number `1/rc`.

Tips

`sprandsym` uses the same random number generator as `rand`, `randi`, and `randn`. You control this generator with `rng`.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Supports these syntaxes only:

  ```
  R = sprandsym(S)
  ```

  ```
  R = sprandsym(n,density)
  ```

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- Supports these syntaxes only:

```
R = sprandsym(S)
```

```
R = sprandsym(n,density)
```

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`sprand` | `sprandn`

Introduced before R2006a

sprank

Structural rank

Syntax

```
r = sprank(A)
```

Description

`r = sprank(A)` calculates the structural rank on page 1-14378 of sparse matrix `A`.

Examples

Calculate Structural Rank of Matrix

Calculate the structural rank of a 2-by-4 matrix.

```
A = [1 0 2 0
     2 0 4 0];
A = sparse(A);
rs = sprank(A)
```

```
rs = 2
```

Compare the structural rank to the regular rank calculation.

```
rf = rank(full(A))
```

```
rf = 1
```

For this matrix, the structural rank is 2 since two of the columns are nonzero. But the actual rank of the matrix is 1 since the columns are multiples of each other.

Input Arguments

A — Input matrix
sparse matrix

Input matrix, specified as a sparse matrix.

Data Types: `double`
Complex Number Support: Yes

More About

Structural Rank

The structural rank of a matrix is the maximum rank of all matrices with the same nonzero pattern. A matrix has full structural rank if it can be permuted so that the diagonal has no zero entries.

The structural rank is an upper bound on the rank of a matrix, so it satisfies `sprank(A) >= rank(full(A))`.

Here are some definitions of the structural rank in terms of other functions:

- The structural rank is a "maximum matching" and is related to the Dulmage-Mendelsohn decomposition by `sprank(A) = sum(dmperm(A)>0)`.
- Unlike `dmperm`, the `matchpairs` function also takes weights into account when it calculates matches. You can calculate a maximum matching by converting the matrix to 1s and 0s and maximizing the weight of the matches with `matchpairs(double(A~=0),0,'max')`. The structural rank is then equal to the number of matches.

See Also

`dmperm` | `matchpairs`

Introduced before R2006a

spring

Spring colormap array

Syntax

```
c = spring
c = spring(m)
```

Description

`c = spring` returns the spring colormap as a three-column array with the same number of rows as the colormap for the current figure. If no figure exists, then the number of rows is equal to the default length of 256. Each row in the array contains the red, green, and blue intensities for a specific color. The intensities are in the range [0,1], and the color scheme looks like this image.

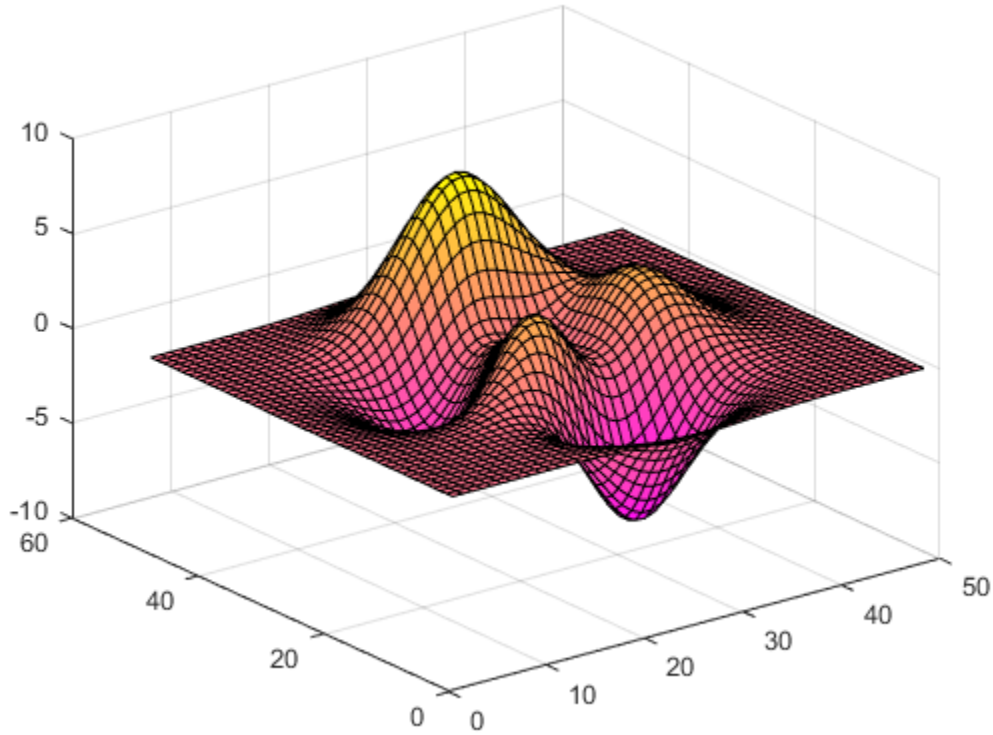
`c = spring(m)` returns the colormap with `m` colors.

Examples

Reverse the Colormap

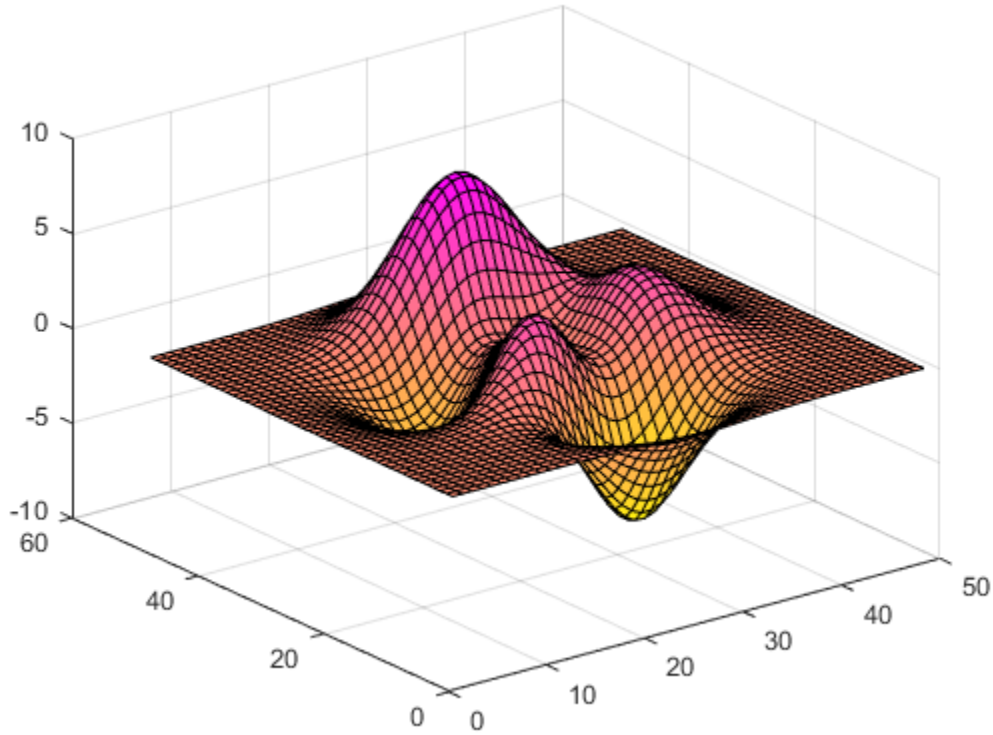
Plot a surface and assign the spring colormap.

```
surf(peaks);
colormap('spring');
```



Get the spring colormap array and reverse the order. Then apply the modified colormap to the surface.

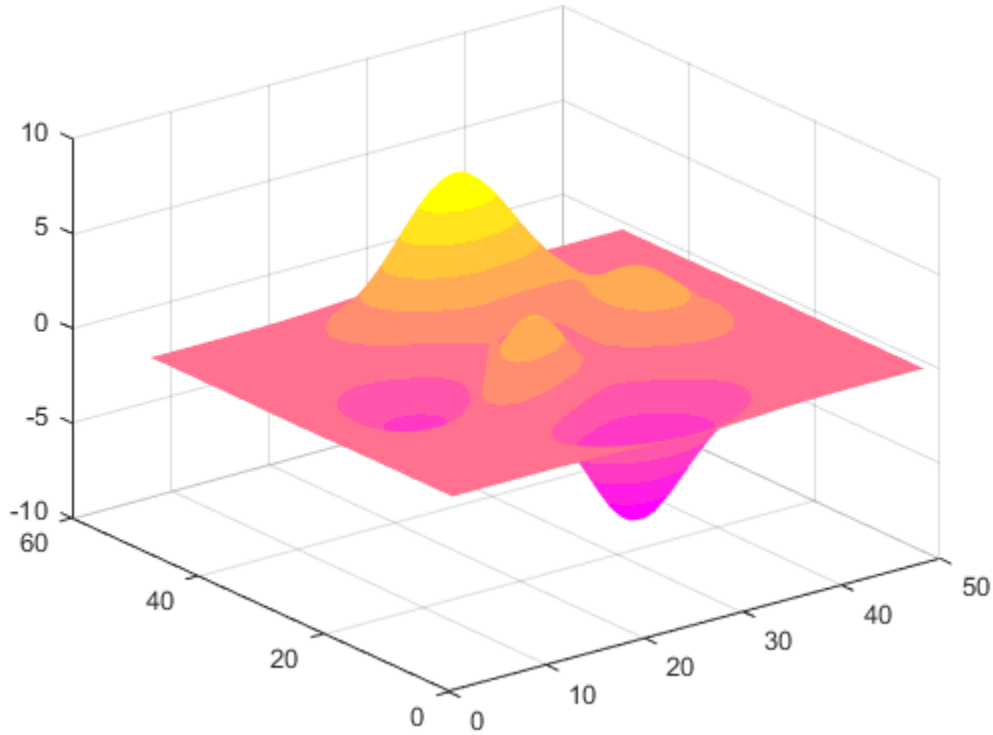
```
c = spring;
c = flipud(c);
colormap(c);
```



Downsample the Spring Colormap

Get a downsampled version of the spring colormap containing only ten colors. Then display the contours of the peaks function by applying the colormap and interpolated shading.

```
c = spring(10);
surf(peaks);
colormap(c);
shading interp;
```



Input Arguments

m — Number of colors
256 (default) | scalar integer value

Number of colors, specified as a scalar integer value. The default value of `m` is equal to the length of the colormap for the current figure. If no figure exists, the default value is 256.

Data Types: `single` | `double`

Compatibility Considerations

Colormaps Have 256 Colors by Default

Behavior changed in R2019b

Starting in R2019b, colormaps have 256 colors by default.

In R2019a and previous releases, the default size is 64. If you have code that depends on a colormap having 64 colors, specify the number of colors when you set the colormap for the figure, axes, or chart. For example, `colormap(spring(64))` sets the figure's colormap to the 64-color `spring` colormap.

Alternatively, you can change the default colormap for all figures within your MATLAB session:

```
set(groot,'defaultFigureColormap',spring(64))
```

See Also

`colormap`

Topics

"Change Color Scheme Using a Colormap"

Introduced before R2006a

sprintf

Format data into string or character vector

Syntax

```
str = sprintf(formatSpec,A1,...,An)
[str,errmsg] = sprintf(formatSpec,A1,...,An)
str = sprintf(literalText)
```

Description

`str = sprintf(formatSpec,A1,...,An)` formats the data in arrays `A1,...,An` using the formatting operators specified by `formatSpec` and returns the resulting text in `str`. The `sprintf` function formats the values in `A1,...,An` in column order. If `formatSpec` is a string, then so is the output `str`. Otherwise, `str` is a character vector.

To return multiple pieces of formatted text as a string array or a cell array of character vectors, use the `compose` function.

`[str,errmsg] = sprintf(formatSpec,A1,...,An)` returns an error message as a character vector when the operation is unsuccessful. Otherwise, `errmsg` is empty.

`str = sprintf(literalText)` translates escape-character sequences in `literalText`, such as `\n` and `\t`. It returns all other characters unaltered. If `literalText` contain a formatting operator (such as `%f`), then `str` discards it and all characters after.

Examples

Floating-Point Formats

Format a floating-point number using `%e`, `%f`, and `%g` specifiers.

```
A = 1/eps;
str_e = sprintf('%0.5e',A)
```

```
str_e =
'4.50360e+15'
```

```
str_f = sprintf('%0.5f',A)
```

```
str_f =
'4503599627370496.00000'
```

```
str_g = sprintf('%0.5g',A)
```

```
str_g =
'4.5036e+15'
```

Literal Text and Array Inputs

Combine literal text with array values to create a character vector.

```
formatSpec = 'The array is %dx%d.';
A1 = 2;
A2 = 3;
str = sprintf(formatSpec,A1,A2)
```

```
str =
'The array is 2x3.'
```

Specify Formatted Text as String Array

To return formatted text as a string, specify `formatSpec` as a string instead of a character vector when you call the `sprintf` function.

Starting in R2017a, you can create strings using double quotes. Convert data and return the result as a string.

```
formatSpec = "The current time is: %d:%d %s";
A1 = 11;
A2 = 20;
A3 = 'a.m.';
str = sprintf(formatSpec,A1,A2,A3)
```

```
str =
"The current time is: 11:20 a.m."
```

Convert input string. Input arrays that contain text either can be character vectors or strings.

```
A1 = 2;
A2 = 35;
A3 = "p.m.";
str = sprintf(formatSpec,A1,A2,A3)
```

```
str =
"The current time is: 2:35 p.m."
```

Integer Format with Floating-Point Inputs

Explicitly convert double-precision values to integers.

```
str = sprintf('%d',round(pi))
```

```
str =
'3'
```

Specify Field Width of a Printed Value

Specify the minimum width of the printed value.

```
str = sprintf('%025d',[123456])
```

```
str =
'0000000000000000000123456'
```

The `0` flag in the `%025d` format specifier requests leading zeros in the output.

Reorder Inputs Using Position Identifier (n$)

Reorder the input values using the `n$` position identifier.

```
A1 = 'X';
A2 = 'Y';
A3 = 'Z';
formatSpec = ' %3$s %2$s %1$s';
str = sprintf(formatSpec,A1,A2,A3)
```

```
str =
' Z Y X'
```

Create Character Vector from Values in Cell Array

```
C = { 1,   2,   3 ;
     'AA','BB','CC'};

str = sprintf(' %d %s',C{:})
```

```
str =
' 1 AA 2 BB 3 CC'
```

The syntax `C{:}` creates a comma-separated list of arrays that contain the contents of each cell from `C` in column order. For example, `C{1}==1` and `C{2}=='AA'`.

Input Arguments

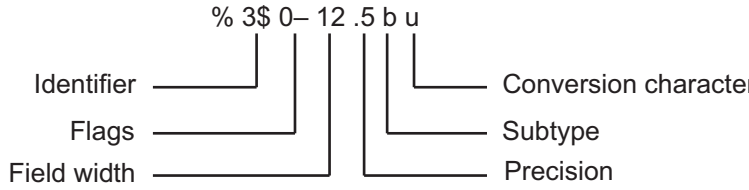
`formatSpec` — Format of output fields
formatting operators

Format of the output fields, specified using formatting operators. `formatSpec` also can include ordinary text and special characters.

`formatSpec` can be a character vector in single quotes, or, starting in R2016b, a string scalar.

Formatting Operator

A formatting operator starts with a percent sign, `%`, and ends with a conversion character. The conversion character is required. Optionally, you can specify identifier, flags, field width, precision, and subtype operators between `%` and the conversion character. (Spaces are invalid between operators and are shown here only for readability).



Conversion Character

This table shows conversion characters to format numeric and character data as text.

Value Type	Conversion	Details
Integer, signed	`%d` or `%i`	Base 10
Integer, unsigned	`%u`	Base 10
	`%o`	Base 8 (octal)
	`%x`	Base 16 (hexadecimal), lowercase letters `a-f`
	`%X`	Same as `%x`, uppercase letters `A-F`
Floating-point number	`%f`	Fixed-point notation (Use a precision operator to specify the number of digits after the decimal point.)
	`%e`	Exponential notation, such as `3.141593e+00` (Use a precision operator to specify the number of digits after the decimal point.)
	`%E`	Same as `%e`, but uppercase, such as `3.141593E+00` (Use a precision operator to specify the number of digits after the decimal point.)
	`%g`	The more compact of `%e` or `%f`, with no trailing zeros (Use a precision operator to specify the number of significant digits.)

Value Type	Conversion	Details
	`%G`	The more compact of `%E` or `%f`, with no trailing zeros (Use a precision operator to specify the number of significant digits.)
Characters or strings	`%c`	Single character
	`%s`	Character vector or string array. The type of the output text is the same as the type of `formatSpec`.

Optional Operators

The optional identifier, flags, field width, precision, and subtype operators further define the format of the output text.

- **Identifier**

  Order for processing the function input arguments. Use the syntax *n*`$`, where *n* represents the positions of the other input arguments in the function call.

  Example: `('%3$s %2$s %1$s %2$s','A','B','C')` prints input arguments `'A'`, `'B'`, `'C'` as follows: `C B A B`.

  Note: If an input argument is an array, you cannot use identifiers to specify particular array elements from that input argument.

- **Flags**

`'–'`	Left-justify. **Example:** `%-5.2f` **Example:** `%-10s`
`'+'`	Always print a sign character (+ or –) for any numeric value. **Example:** `%+5.2f` Right-justify text. **Example:** `%+10s`
`' '`	Insert a space before the value. **Example:** `% 5.2f`
`'0'`	Pad to field width with zeros before the value. **Example:** `%05.2f`

`'#'`	Modify selected numeric conversions: • For `%o`, `%x`, or `%X`, print `0`, `0x`, or `0X` prefix. • For `%f`, `%e`, or `%E`, print decimal point even when precision is 0. • For `%g` or `%G`, do not remove trailing zeros or decimal point. **Example:** `%#5.0f`

- **Field Width**

  Minimum number of characters to print. The field width operator can be a number, or an asterisk (*) to refer to an input argument.

  When you specify * as the field width operator, the other input arguments must provide both a width and a value to be printed. Widths and values can be pairs of arguments or pairs within a numeric array. With * as the field width operator, you can print different values with different widths.

  Example: The input arguments `('%12d',intmax)` are equivalent to `('%*d',12,intmax)`.

  Example: The input arguments `('%*d',[2 10 5 100])` return `'10 100'`, with two spaces allocated for `10` and five spaces for `100`. As an alternative, you also can specify the field widths and values as multiple arguments, as in `('%*d',2,10,5,100)`.

  The function pads to field width with spaces before the value unless otherwise specified by flags.

- **Precision**

For `%f`, `%e`, or `%E`	Number of digits to the right of the decimal point **Example:** `'%.4f'` prints `pi` as `'3.1416'`
For `%g` or `%G`	Number of significant digits **Example:** `'%.4g'` prints `pi` as `'3.142'`

  The precision operator can be a number, or an asterisk (*) to refer to an argument.

  When you specify * as the field precision operator, the other input arguments must provide both a precision and a value to be printed. Precisions and values can be pairs of arguments, or pairs within a numeric array. With * as the precision operator, you can print different values to different precisions.

When you specify `*.*` as field width and precision operators, you must specify field widths, precisions, and values as triplets.

Example: The input arguments `('%.4f',pi)` are equivalent to `('%.*f',4,pi)`.

Example: The input arguments `('%6.4f',pi)` are equivalent to `('%.*f',6,4,pi)`.

Example: The input arguments `('%*.*f',6,4,pi,9,6,exp(1))` return `'3.1416 2.718282'`, with `9` and `6` as the field width and precision for the output of `exp(1)`.

Note If you specify a precision operator for floating-point values that exceeds the precision of the input numeric data type, the results might not match the input values to the precision you specified. The result depends on your computer hardware and operating system.

- **Subtypes**

  You can use a subtype operator to print a floating-point value as its octal, decimal, or hexadecimal value. The subtype operator immediately precedes the conversion character. This table shows the conversions that can use subtypes.

Input Value Type	Subtype and Conversion Character	Output Value Type

Floating-point number	`%bx` or `%bX`
	`%bo`
	`%bu`
		Double-precision hexadecimal, octal, or decimal value
		Example: `%bx` prints `pi` as `400921fb54442d18`

	`%tx` or `%tX`
	`%to`
	`%tu`
		Single-precision hexadecimal, octal, or decimal value
		Example: `%tx` prints `pi` as `40490fdb`

Text Before or After Formatting Operators

`formatSpec` can also include additional text before a percent sign, `%`, or after a conversion character. The text can be:

- Ordinary text to print.

- Special characters that you cannot enter as ordinary text. This table shows how to represent special characters in `formatSpec`.

Special Character	Representation
Single quotation mark	`''`
Percent character	`%%`
Backslash	`\\`
Alarm	`\a`
Backspace	`\b`
Form feed	`\f`
New line	`\n`
Carriage return	`\r`
Horizontal tab	`\t`
Vertical tab	`\v`
Character whose Unicode numeric value can be represented by the hexadecimal number, `N`	`\xN` **Example:** `sprintf('\x5A')` returns `'Z'`
Character whose Unicode numeric value can be represented by the octal number, `N`	`\N` **Example:** `sprintf('\132')` returns `'Z'`

Notable Behavior of Conversions with Formatting Operators

- Numeric conversions print only the real component of complex numbers.
- If you specify a conversion that does not fit the data, such as a text conversion for a numeric value, MATLAB overrides the specified conversion, and uses `%e`.

  Example: `'%s'` converts `pi` to `3.141593e+00`.
- If you apply a text conversion (either `%c` or `%s`) to integer values, MATLAB converts values that correspond to valid character codes to characters.

  Example: `'%s'` converts `[65 66 67]` to `ABC`.

Data Types: `char` | `string`

`A1,...,An` — Numeric, character, or string arrays
arrays

Numeric, character, or string arrays.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string`

`literalText` — Input text without formatting operators
character vector | string scalar

Input text without formatting operators, specified as a character vector or string scalar. `sprintf` translates any escape-character sequences in `literalText`.

Data Types: `char` | `string`

Output Arguments

`str` — Formatted text
character vector | string scalar

Formatted text, returned as a character vector or a string scalar. The type of output matches the type of `formatSpec`.

`errmsg` — Error message
character vector

Error message, returned as a character vector, when the operation is unsuccessful. Otherwise, `errmsg` is empty.

Tips

- The `sprintf` function is similar to `fprintf`, but `fprintf` prints to a file or to the Command Window.
- Format specifiers for the reading functions `sscanf` and `fscanf` differ from the formats for the writing functions `sprintf` and `fprintf`. The reading functions do not support a precision field. The width field specifies a minimum for writing, but a maximum for reading.

References

[1] Kernighan, B. W., and D. M. Ritchie, *The C Programming Language*, Second Edition, Prentice-Hall, Inc., 1988.

[2] ANSI specification X3.159-1989: “Programming Language C,” ANSI, 1430 Broadway, New York, NY 10018.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The `formatSpec` parameter must be constant.
- In `formatSpec`, hexadecimal numbers must be in the range [0 7F] and octal numbers must be in the range [0 177].
- If all the input arrays are constant, the code generator evaluates the `sprintf` call in MATLAB at compile time. In this case, the code generation restrictions for `sprintf` do not apply and the behavior of `sprintf` in the generated code is the same as the behavior in MATLAB.
- If extrinsic calls are not possible, the code generator produces C code for `sprintf`. Extrinsic calls are not possible when extrinsic calls are disabled or when `sprintf` is called inside a `parfor` loop.
- The behavior of `sprintf` in the generated code matches the C compiler behavior instead of the MATLAB behavior in these cases:
  - The format specifier has a corresponding C format specifier, for example, `%e` or `%E`.
  - The `sprintf` call is inside a `parfor` loop.
  - Extrinsic calls are disabled.
- These options and capabilities are not supported:
  - The `n$` position identifier for reordering input values

  - Printing arrays
  - Using subtypes to print a floating-point number as its octal, decimal, or hexadecimal value
- When you call `sprintf` with the format specifier `%s`, you cannot put a null character in the middle of the input character vector. To write a null character, use `sprintf(fid, '%c', char(0))`.
- Input argument types must match their format types. For example, if `n` is a double, code generation does not allow the following code:

  ```
  str = sprintf('%d',n)
  ```

  For code generation, first cast `n` to a signed integer type such as `int8`.

  ```
  str = sprintf('%d',int8(n))
  ```
- When you call `sprintf` with an integer format specifier, the type of the integer argument must be a type that the target hardware can represent as a native C type. For example, if you call `sprintf('%d', int64(n))`, then the target hardware must have a native C type that supports a 64-bit integer.
- Dynamic memory allocation must be enabled.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`char` | `compose` | `fprintf` | `fscanf` | `int2str` | `num2str` | `sscanf`

Topics

“Formatting Text”

Introduced before R2006a

spy

Visualize sparsity pattern of matrix

Syntax

```
spy(S)
spy(S,LineSpec)
spy( ___ ,MarkerSize)
```

Description

`spy(S)` plots the sparsity pattern of matrix `S`. Nonzero values are colored while zero values are white. The plot displays the number of nonzeros in the matrix, `nz = nnz(S)`.

`spy(S,LineSpec)` additionally specifies `LineSpec` to give the marker symbol and color to use in the plot. For example, `spy(A,'r*')` uses red asterisks for nonzeros.

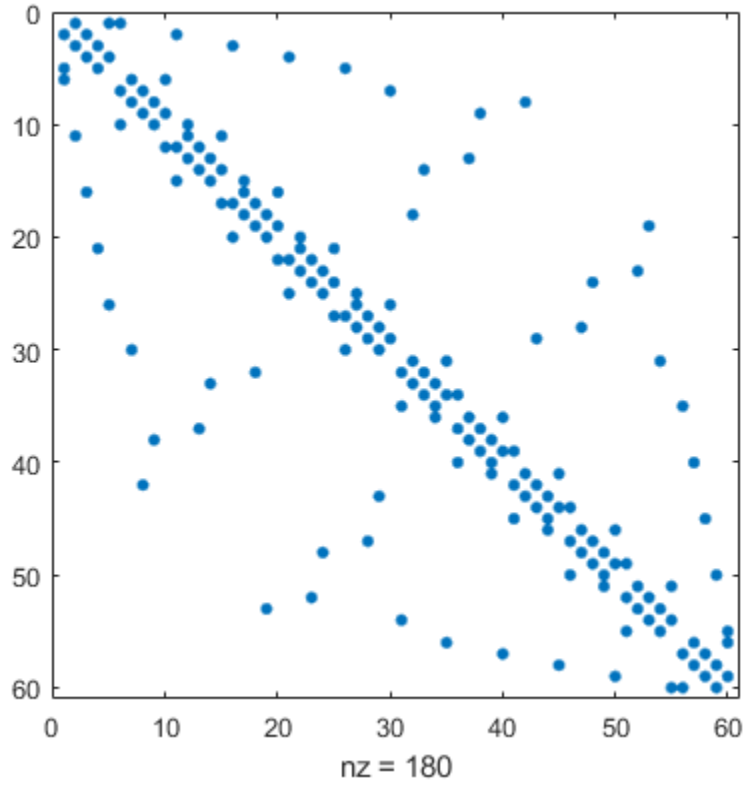
`spy( ___ ,MarkerSize)` specifies `MarkerSize` to give the size of the markers using either of the previous input argument combinations.

Examples

Plot Sparsity Pattern

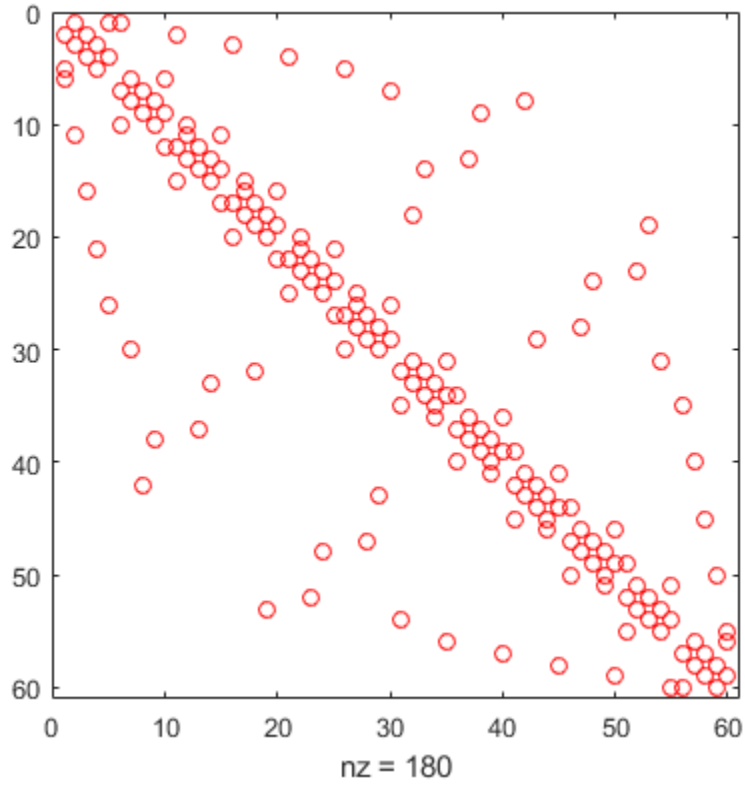
Plot the 60-by-60 sparse adjacency matrix of the connectivity graph of the Buckminster Fuller geodesic dome. This matrix also represents the carbon-60 molecule and a soccer ball.

```
B = bucky;
spy(B)
```



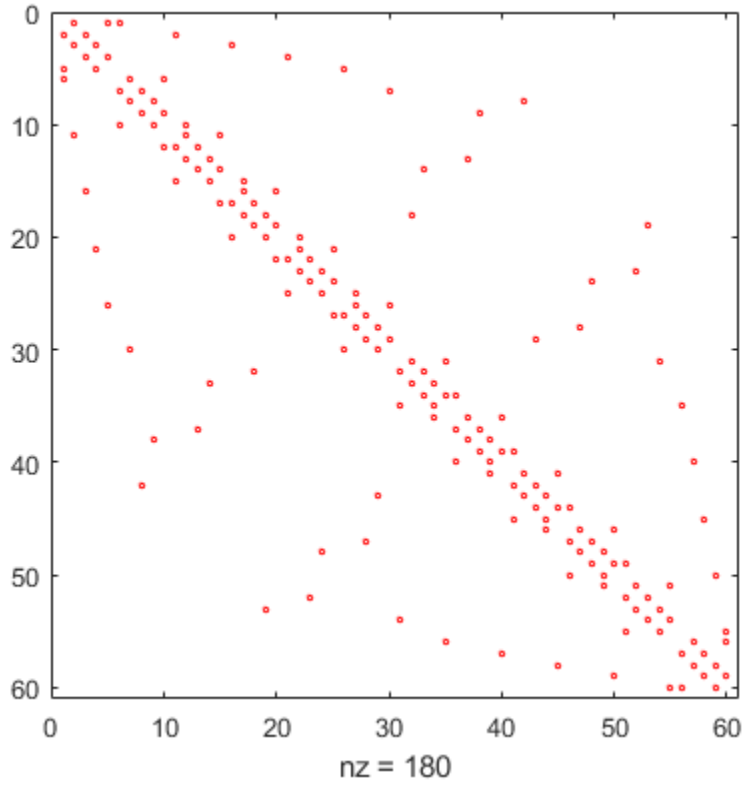
Specify the color and marker.

```
spy(B,'ro')
```



Also specify the marker size.

```
spy(B,'ro',2)
```



Input Arguments

S — Input matrix
matrix

Input matrix. S is typically a sparse matrix, but can be either full or sparse.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

`MarkerSize` — Size of markers
positive integer scalar

Size of markers, specified as a positive integer scalar.

Example: `spy(A,3)` uses markers of size 3.

`LineSpec` — Marker symbol and color
character vector | string

Marker symbol and color, specified as a character vector or string containing symbols. The symbols can appear in any order. You do not need to specify both characteristics (marker and color). For example, if you omit the color and specify the marker, then the plot uses the default colors but uses the specified markers.

Example: `'or'` uses red circle markers.

Marker	Description
o	Circle
+	Plus sign
*	Asterisk
.	Point
x	Cross
s	Square
d	Diamond
^	Upward-pointing triangle
v	Downward-pointing triangle
>	Right-pointing triangle
<	Left-pointing triangle
p	Pentagram
h	Hexagram

Color	Description
y	Yellow
m	Magenta

Color	Description
c	Cyan
r	Red
g	Green
b	Blue
w	White
k	Black

Tips

- `format +` is a text-based alternative for displaying the nonzero structure of a small matrix:

```
format +
eye(4)
```

```
ans =

+
 +
  +
   +
```

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`LineSpec` | `find` | `nonzeros` | `sparse`

Topics

“Sparse Matrix Reordering”

Introduced before R2006a

sqrt

Square root

Syntax

```
B = sqrt(X)
```

Description

`B = sqrt(X)` returns the square root of each element of the array `X`. For the elements of `X` that are negative or complex, `sqrt(X)` produces complex results.

The `sqrt` function's domain includes negative and complex numbers, which can lead to unexpected results if used unintentionally. For negative and complex numbers `z = u + i*w`, the complex square root `sqrt(z)` returns

```
sqrt(r)*(cos(phi/2) + 1i*sin(phi/2))
```

where `r = abs(z)` is the radius and `phi = angle(z)` is the phase angle on the closed interval `-pi <= phi <= pi`.

If you want negative and complex numbers to return error messages rather than return complex results, use `realsqrt` instead.

Examples

Square Root of Vector Elements

Create a row vector containing both negative and positive values.

```
X = -2:2
```

```
X = 1×5
```

```
    -2    -1     0     1     2
```

Compute the square root of each element of X.

```
Y = sqrt(X)
```

```
Y = 1×5 complex

   0.0000 + 1.4142i   0.0000 + 1.0000i   0.0000 + 0.0000i   1.0000 + 0.0000i   1.4142 -
```

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a numeric scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

IEEE Compliance

For real inputs, `sqrt` has a few behaviors that differ from those recommended in the IEEE-754 Standard. In particular, negative inputs produce complex results instead of `NaN`.

	MATLAB	IEEE
`sqrt(-0)`	`0`	`-0`
`sqrt(X)` for `X < 0`	`0+sqrt(-X)*i`	`NaN`

Tips

- See `sqrtm` for the matrix square root.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Simulation produces an error. Generated standalone code returns `NaN` when the input value `x` is real, but the output should be complex. To get the complex result, make the input value complex by passing in `complex(x)`.
- If you generate code for standalone targets and the input to `sqrt` in your MATLAB code is not a constant, the value that the generated code returns for `sqrt(-0)` is identical to the value that the standard library function of your C/C++ compiler returns. If the standard library function complies with the IEEE-754 Standard, the generated code returns `-0`.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Simulation produces an error. Generated standalone code returns `NaN` when the input value `x` is real, but the output should be complex. To get the complex result, make the input value complex by passing in `complex(x)`.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

If the output of the function running on the GPU can be complex, then you must explicitly specify its input arguments as complex. For more information, see “Work with Complex Numbers on a GPU” (Parallel Computing Toolbox).

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`nthroot` | `realsqrt` | `sqrtm`

Introduced before R2006a

sqrtm

Matrix square root

Syntax

```
X = sqrtm(A)
[X,residual] = sqrtm(A)
[X,alpha,condx] = sqrtm(A)
```

Description

`X = sqrtm(A)` returns the principal square root of the matrix `A`, that is, `X*X = A`.

`X` is the unique square root for which every eigenvalue has nonnegative real part. If `A` has any eigenvalues with negative real parts, then a complex result is produced. If `A` is singular, then `A` might not have a square root. If exact singularity is detected, a warning is printed.

`[X,residual] = sqrtm(A)` also returns the residual, `residual = norm(A-X^2,1)/norm(A,1)`. This syntax does not print warnings if exact singularity is detected.

`[X,alpha,condx] = sqrtm(A)` returns stability factor `alpha` and an estimate of the matrix square root condition number of `X` in 1-norm, `condx`. The residual `norm(A-X^2,1)/norm(A,1)` is bounded approximately by `n*alpha*eps` and the 1-norm relative error in `X` is bounded approximately by `n*alpha*condx*eps`, where `n = max(size(A))`.

Examples

Matrix Squareroot of Difference Operator

Create a matrix representation of the fourth difference operator, `A`. This matrix is symmetric and positive definite.

```
A = [5 -4 1 0 0; -4 6 -4 1 0; 1 -4 6 -4 1; 0 1 -4 6 -4; 0 0 1 -4 6]
```

```
A = 5×5

     5    -4     1     0     0
    -4     6    -4     1     0
     1    -4     6    -4     1
     0     1    -4     6    -4
     0     0     1    -4     6
```

Calculate the unique positive definite square root of `A` using `sqrtm`. `X` is the matrix representation of the second difference operator.

```
X = round(sqrtm(A))
```

```
X = 5×5

     2    -1     0     0     0
    -1     2    -1     0     0
     0    -1     2    -1     0
     0     0    -1     2    -1
     0     0     0    -1     2
```

Matrix with Multiple Squareroots

Consider a matrix that has four squareroots, `A`.

$$A = \begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix}$$

Two of the squareroots of `A` are given by `Y1` and `Y2`:

$$Y_1 = \begin{bmatrix} 1.5667 & 1.7408 \\ 2.6112 & 4.1779 \end{bmatrix}$$

$$Y_2 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Confirm that `Y1` and `Y2` are squareroots of matrix `A`.

```
A = [7 10; 15 22];
Y1 = [1.5667 1.7408; 2.6112 4.1779];
A - Y1*Y1
```

```
ans = 2×2
10^-3 ×

   -0.1258   -0.1997
   -0.2995   -0.4254
```

```
Y2 = [1 2; 3 4];
A - Y2*Y2
```

```
ans = 2×2

     0     0
     0     0
```

The other two squareroots of `A` are `-Y1` and `-Y2`. All four of these roots can be obtained from the eigenvalues and eigenvectors of `A`. If `[V,D] = eig(A)`, then the squareroots have the general form `Y = V*S/V`, where `D = S*S` and `S` has four choices of sign to produce four different values of `Y`:

$$S = \begin{bmatrix} \pm 0.3723 & 0 \\ 0 & \pm 5.3723 \end{bmatrix}$$

Calculate the squareroot of `A` with `sqrtm`. The `sqrtm` function chooses the positive square roots and produces `Y1`, even though `Y2` seems to be a more natural result.

```
Y = sqrtm(A)
```

```
Y = 2×2

    1.5667    1.7408
    2.6112    4.1779
```

Input Arguments

A — Input matrix
square matrix

Input matrix, specified as a square matrix.

Data Types: `single` | `double`
Complex Number Support: Yes

Tips

- Some matrices, like `A = [0 1; 0 0]`, do not have any square roots, real or complex, and `sqrtm` cannot be expected to produce one.

Algorithms

The algorithm `sqrtm` uses is described in [3].

References

[1] N.J. Higham, "Computing real square roots of a real matrix," *Linear Algebra and Appl.*, 88/89, pp. 405–430, 1987

[2] Bjorck, A. and S. Hammerling, "A Schur method for the square root of a matrix," *Linear Algebra and Appl.*, 52/53, pp. 127–140, 1983

[3] Deadman, E., Higham, N. J. and R. Ralha, "Blocked Schur algorithms for computing the matrix square root," *Lecture Notes in Comput. Sci.*, 7782, Springer-Verlag, pp. 171–182, 2013

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- When the input matrix contains a nonfinite value, the generated code does not issue an error. Instead, the output contains `NaN` values.
- Code generation does not support sparse matrix inputs for this function.

See Also

`expm` | `funm` | `logm`

Introduced before R2006a

squeeze

Remove dimensions of length 1

Syntax

```
B = squeeze(A)
```

Description

`B = squeeze(A)` returns an array with the same elements as the input array `A`, but with dimensions of length 1 removed. For example, if `A` is a 3-by-1-by-1-by-2 array, then `squeeze(A)` returns a 3-by-2 matrix.

If `A` is a row vector, column vector, scalar, or an array with no dimensions of length 1, then `squeeze` returns the input `A`.

Examples

3-D Array to Matrix

Create a 2-by-1-by-2 array and remove the dimension of length 1, resulting in a 2-by-2 matrix.

```
A = zeros(2,1,2);
A(:,:,1) = [1 2]';
A(:,:,2) = [3 4]'
```

```
A =
A(:,:,1) =

     1
     2


A(:,:,2) =
```

```
     3
     4


B = squeeze(A)

B = 2×2

     1     3
     2     4
```

3-D Array to Vector

Create a 1-by-1-by-3 array and remove the dimensions of length 1, resulting in a 3-by-1 column vector.

```
A = zeros(1,1,3);
A(:,:,1:3) = [1 2 3]

A =
A(:,:,1) =

     1


A(:,:,2) =

     2


A(:,:,3) =

     3


B = squeeze(A)

B = 3×1

     1
     2
```

3

Input Arguments

A — Input array
multidimensional array

Input array, specified as a multidimensional array.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Does not support cell arrays.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Does not support cell arrays.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`permute` | `reshape` | `shiftdim`

Introduced before R2006a

ss2tf

Convert state-space representation to transfer function

Syntax

```
[b,a] = ss2tf(A,B,C,D)
[b,a] = ss2tf(A,B,C,D,ni)
```

Description

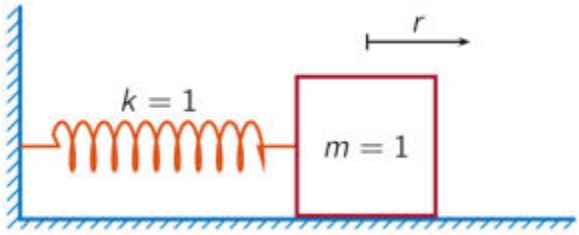
`[b,a] = ss2tf(A,B,C,D)` converts a state-space representation of a system into an equivalent transfer function. `ss2tf` returns the Laplace-transform transfer function for continuous-time systems and the Z-transform transfer function for discrete-time systems.

`[b,a] = ss2tf(A,B,C,D,ni)` returns the transfer function that results when the `ni`th input of a system with multiple inputs is excited by a unit impulse.

Examples

Mass-Spring System

A one-dimensional discrete-time oscillating system consists of a unit mass, *m*, attached to a wall by a spring of unit elastic constant. A sensor samples the acceleration, *a*, of the mass at F_s = 5 Hz.



Generate 50 time samples. Define the sampling interval $\Delta t = 1/F_s$.

```
Fs = 5;
dt = 1/Fs;
N = 50;
t = dt*(0:N-1);
```

The oscillator can be described by the state-space equations

$$\begin{aligned} x(k+1) &= Ax(k) + Bu(k), \\ y(k) &= Cx(k) + Du(k), \end{aligned}$$

where $x = (r\ \ v)^T$ is the state vector, r and v are respectively the position and velocity of the mass, and the matrices

$$A = \begin{pmatrix} \cos\Delta t & \sin\Delta t \\ -\sin\Delta t & \cos\Delta t \end{pmatrix}, \quad B = \begin{pmatrix} 1-\cos\Delta t \\ \sin\Delta t \end{pmatrix}, \quad C = (-1\ \ 0), \quad D = (1).$$

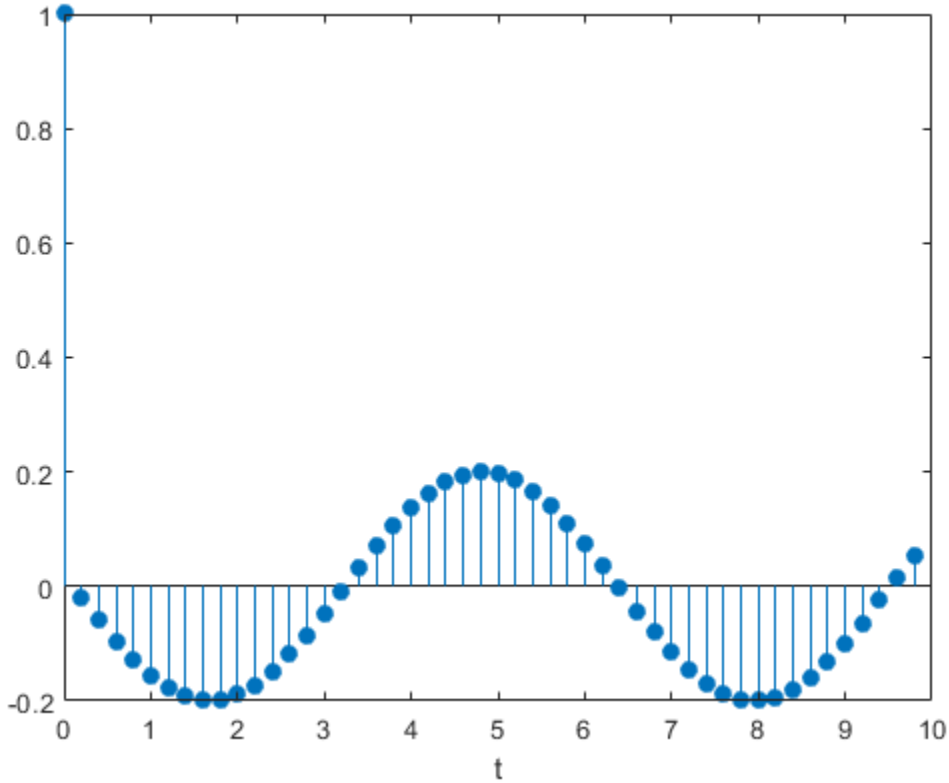
```
A = [cos(dt) sin(dt);-sin(dt) cos(dt)];
B = [1-cos(dt);sin(dt)];
C = [-1 0];
D = 1;
```

The system is excited with a unit impulse in the positive direction. Use the state-space model to compute the time evolution of the system starting from an all-zero initial state.

```
u = [1 zeros(1,N-1)];

x = [0;0];
for k = 1:N
    y(k) = C*x + D*u(k);
    x = A*x + B*u(k);
end
```

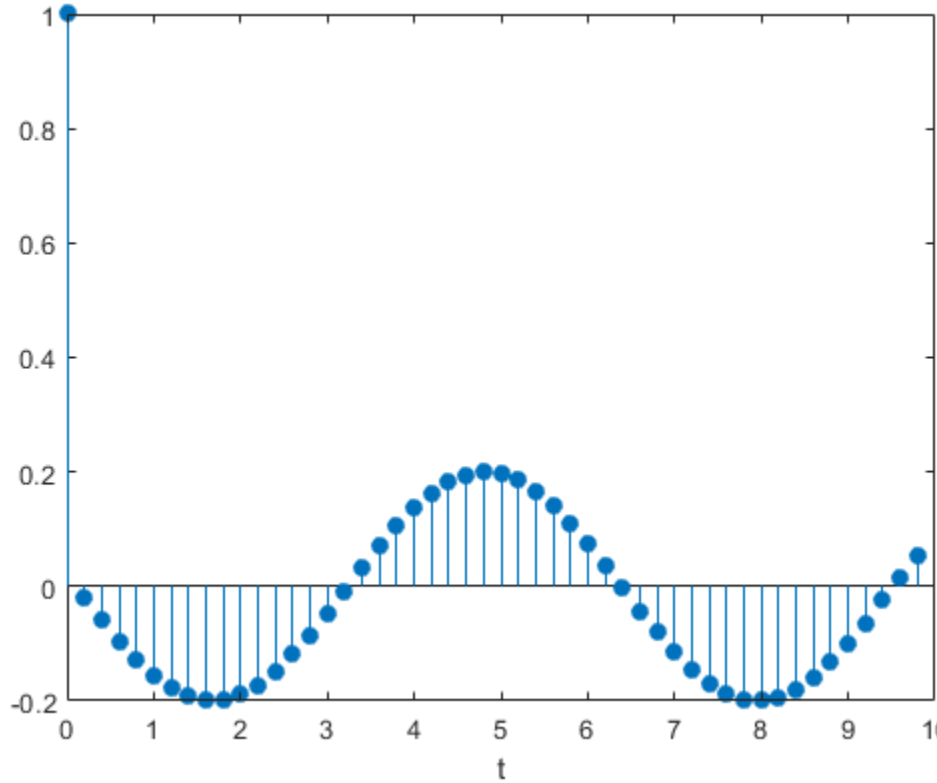
Plot the acceleration of the mass as a function of time.

```
stem(t,y,'filled')
xlabel('t')
```



Compute the time-dependent acceleration using the transfer function $H(z)$ to filter the input. Plot the result.

```
[b,a] = ss2tf(A,B,C,D);
yt = filter(b,a,u);
stem(t,yt,'filled')
xlabel('t')
```

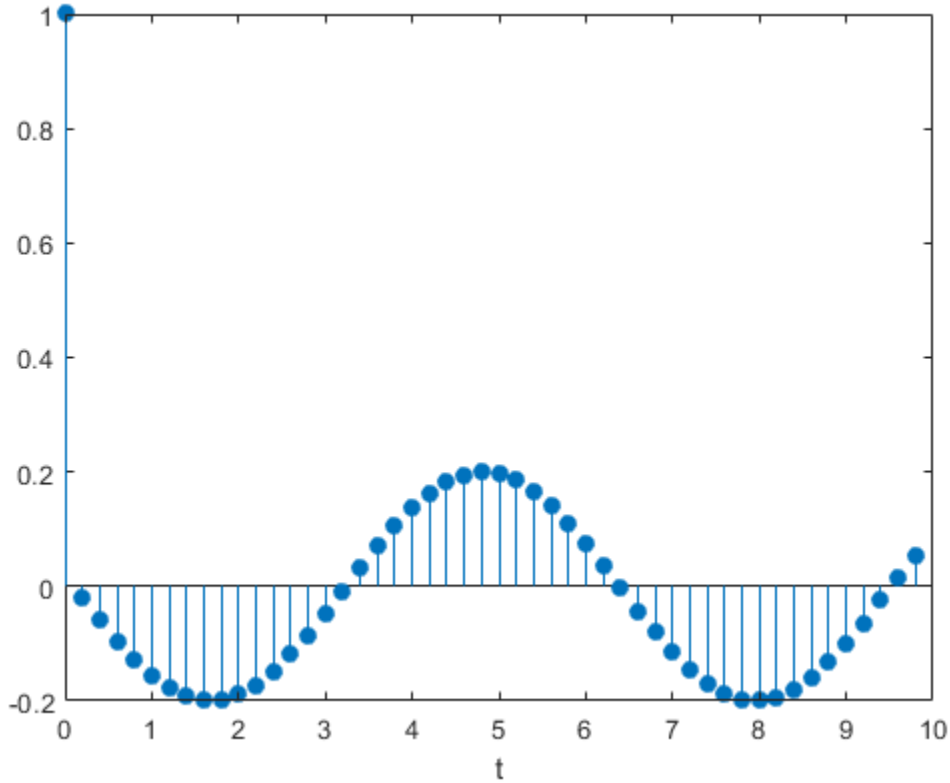


The transfer function of the system has an analytic expression:

$$H(z) = \frac{1 - z^{-1}(1 + \cos\Delta t) + z^{-2}\cos\Delta t}{1 - 2z^{-1}\cos\Delta t + z^{-2}}.$$

Use the expression to filter the input. Plot the response.

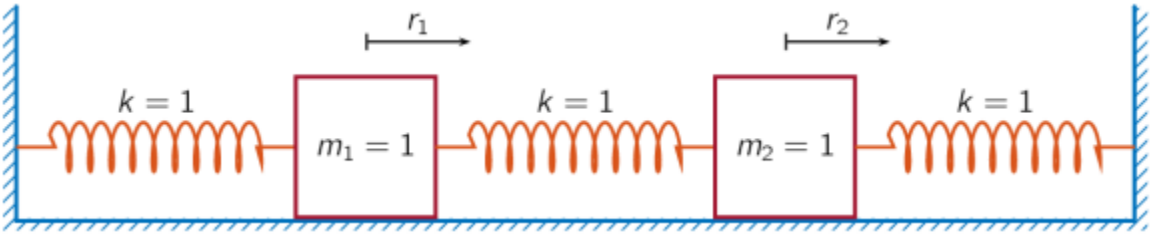
```
bf = [1 -(1+cos(dt)) cos(dt)];
af = [1 -2*cos(dt) 1];
yf = filter(bf,af,u);
stem(t,yf,'filled')
xlabel('t')
```



The result is the same in all three cases.

Two-Body Oscillator

An ideal one-dimensional oscillating system consists of two unit masses, m_1 and m_2, confined between two walls. Each mass is attached to the nearest wall by a spring of unit elastic constant. Another such spring connects the two masses. Sensors sample a_1 and a_2, the accelerations of the masses, at F_s = 16 Hz.



Specify a total measurement time of 16 s. Define the sampling interval $\Delta t = 1/F_s$.

```
Fs = 16;
dt = 1/Fs;
N = 257;
t = dt*(0:N-1);
```

The system can be described by the state-space model

$$x(n+1) = Ax(n) + Bu(n),$$
$$y(n) = Cx(n) + Du(n),$$

where $x = (r_1\ v_1\ r_2\ v_2)^T$ is the state vector and r_i and v_i are respectively the location and the velocity of the i-th mass. The input vector $u = (u_1\ u_2)^T$ and the output vector $y = (a_1\ a_2)^T$. The state-space matrices are

$$A = \exp(A_c \Delta t), \quad B = A_c^{-1}(A - I)B_c, \quad C = \begin{pmatrix} -2 & 0 & 1 & 0 \\ 1 & 0 & -2 & 0 \end{pmatrix}, \quad D = I,$$

the continuous-time state-space matrices are

$$A_c = \begin{pmatrix} 0 & 1 & 0 & 0 \\ -2 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & -2 & 0 \end{pmatrix}, \quad B_c = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix},$$

and I denotes an identity matrix of the appropriate size.

```
Ac = [0 1 0 0;-2 0 1 0;0 0 0 1;1 0 -2 0];
A = expm(Ac*dt);
Bc = [0 0;1 0;0 0;0 1];
B = Ac\(A-eye(4))*Bc;
```

```
C = [-2 0 1 0;1 0 -2 0];
D = eye(2);
```

The first mass, m_1, receives a unit impulse in the positive direction.

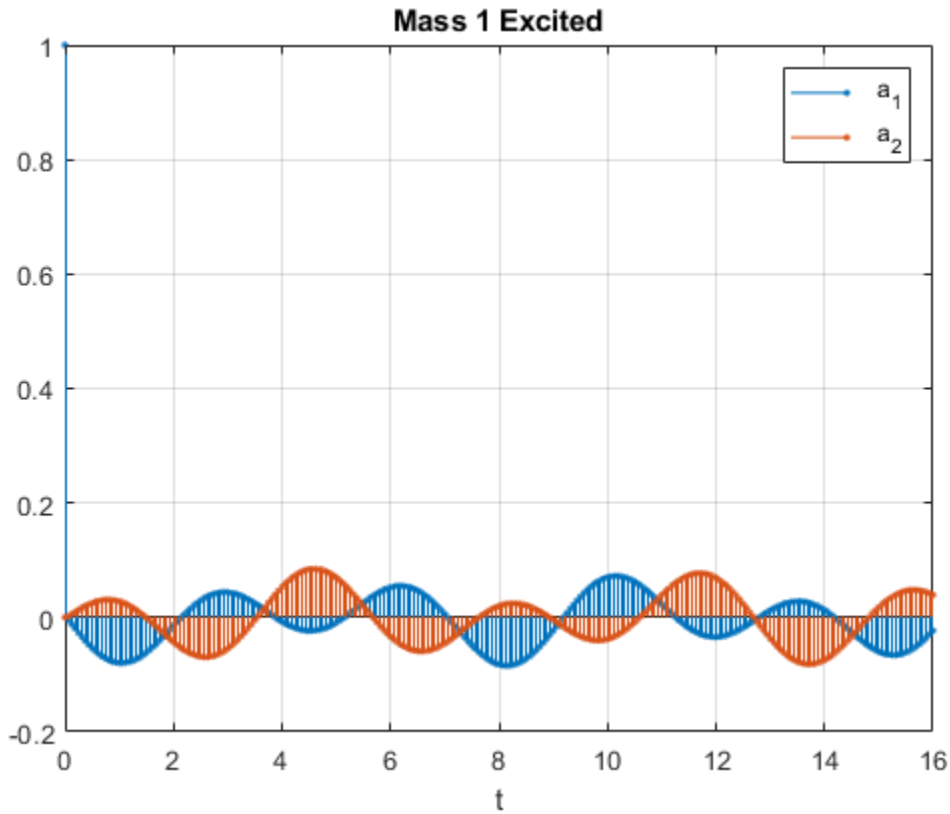
```
ux = [1 zeros(1,N-1)];
u0 = zeros(1,N);
u = [ux;u0];
```

Use the model to compute the time evolution of the system starting from an all-zero initial state.

```
x = [0;0;0;0];
for k = 1:N
    y(:,k) = C*x + D*u(:,k);
    x = A*x + B*u(:,k);
end
```

Plot the accelerations of the two masses as functions of time.

```
stem(t,y','.')
xlabel('t')
legend('a_1','a_2')
title('Mass 1 Excited')
grid
```

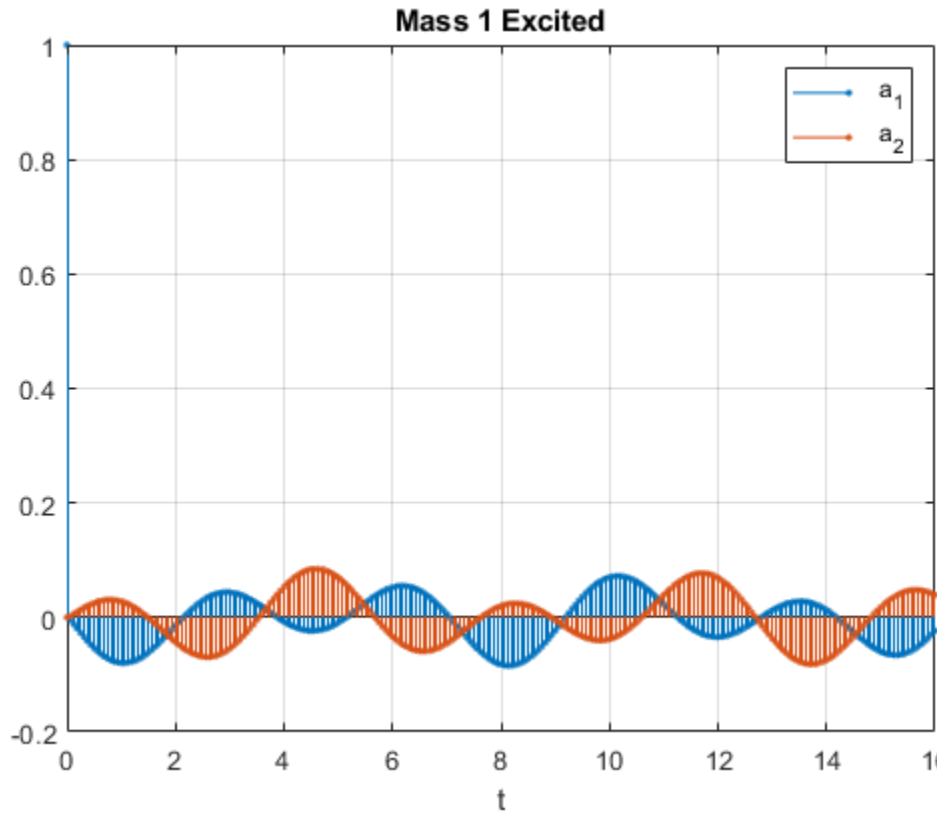


Convert the system to its transfer function representation. Find the response of the system to a positive unit impulse excitation on the first mass.

```
[b1,a1] = ss2tf(A,B,C,D,1);
y1u1 = filter(b1(1,:),a1,ux);
y1u2 = filter(b1(2,:),a1,ux);
```

Plot the result. The transfer function gives the same response as the state-space model.

```
stem(t,[y1u1;y1u2]','.')
xlabel('t')
legend('a_1','a_2')
title('Mass 1 Excited')
grid
```

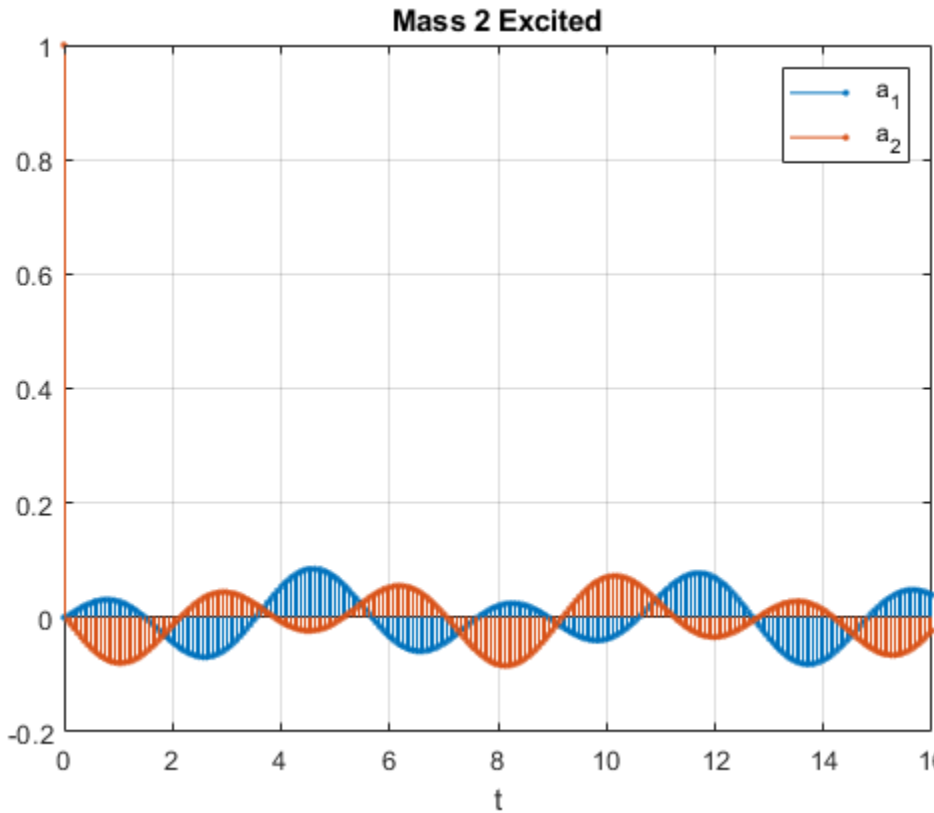


The system is reset to its initial configuration. Now the other mass, m_2, receives a unit impulse in the positive direction. Compute the time evolution of the system.

```
u = [u0;ux];

x = [0;0;0;0];
for k = 1:N
    y(:,k) = C*x + D*u(:,k);
    x = A*x + B*u(:,k);
end
```

Plot the accelerations. The responses of the individual masses are switched.

```
stem(t,y','.')
xlabel('t')
legend('a_1','a_2')
title('Mass 2 Excited')
grid
```

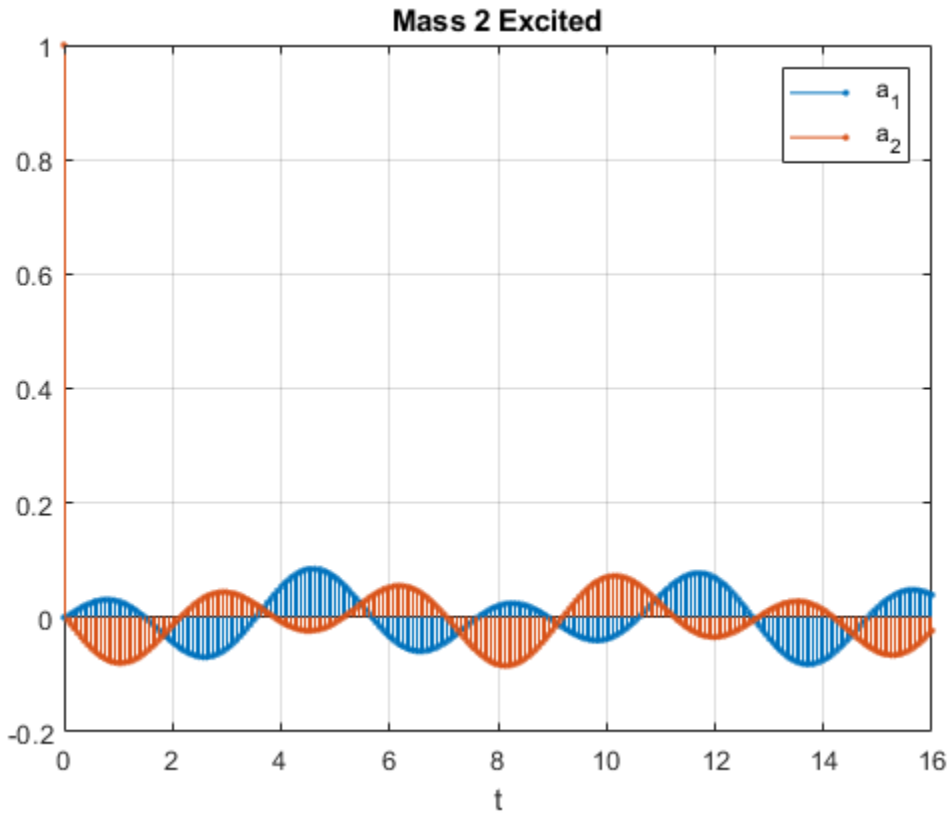


Find the response of the system to a positive unit impulse excitation on the second mass.

```
[b2,a2] = ss2tf(A,B,C,D,2);
y2u1 = filter(b2(1,:),a2,ux);
y2u2 = filter(b2(2,:),a2,ux);
```

Plot the result. The transfer function gives the same response as the state-space model.

```
stem(t,[y2u1;y2u2]','.')
xlabel('t')
legend('a_1','a_2')
title('Mass 2 Excited')
grid
```



Input Arguments

A — State matrix
matrix

State matrix, specified as a matrix. If the system has *p* inputs and *q* outputs and is described by *n* state variables, then `A` is *n*-by-*n*.

Data Types: `single` | `double`

`B` — Input-to-state matrix
matrix

Input-to-state matrix, specified as a matrix. If the system has *p* inputs and *q* outputs and is described by *n* state variables, then `B` is *n*-by-*p*.

Data Types: `single` | `double`

`C` — State-to-output matrix
matrix

State-to-output matrix, specified as a matrix. If the system has *p* inputs and *q* outputs and is described by *n* state variables, then `C` is *q*-by-*n*.

Data Types: `single` | `double`

`D` — Feedthrough matrix
matrix

Feedthrough matrix, specified as a matrix. If the system has *p* inputs and *q* outputs and is described by *n* state variables, then `D` is *q*-by-*p*.

Data Types: `single` | `double`

`ni` — Input index
1 (default) | integer scalar

Input index, specified as an integer scalar. If the system has *p* inputs, use `ss2tf` with a trailing argument `ni` = 1, ..., *p* to compute the response to a unit impulse applied to the `ni`th input.

Data Types: `single` | `double`

Output Arguments

`b` — Transfer function numerator coefficients
vector | matrix

Transfer function numerator coefficients, returned as a vector or matrix. If the system has p inputs and q outputs and is described by n state variables, then b is q-by-(n + 1) for each input. The coefficients are returned in descending powers of s or z.

a — Transfer function denominator coefficients
vector

Transfer function denominator coefficients, returned as a vector. If the system has p inputs and q outputs and is described by n state variables, then a is 1-by-(n + 1) for each input. The coefficients are returned in descending powers of s or z.

More About

Transfer Function

- For discrete-time systems, the state-space matrices relate the state vector x, the input u, and the output y through

  $$\begin{aligned} x(k+1) &= \mathrm{A}\,x(k) + \mathrm{B}\,u(k) \\ y(k) &= \mathrm{C}\,x(k) + \mathrm{D}\,u(k)\,. \end{aligned}$$

  The transfer function is the Z-transform of the system's impulse response. It can be expressed in terms of the state-space matrices as

  $$\mathrm{H}(z)=\mathrm{C}(z\mathrm{I}-\mathrm{A})^{-1}\mathrm{B}+\mathrm{D}\,.$$

- For continuous-time systems, the state-space matrices relate the state vector x, the input u, and the output y through

  $$\begin{aligned} \dot{x} &= \mathrm{A}\,x + \mathrm{B}\,u \\ y &= \mathrm{C}\,x + \mathrm{D}\,u\,. \end{aligned}$$

  The transfer function is the Laplace transform of the system's impulse response. It can be expressed in terms of the state-space matrices as

  $$\mathrm{H}(s) = \mathrm{C}(s\mathrm{I}-\mathrm{A})^{-1}\mathrm{B}+\mathrm{D}\,.$$

See Also

`latc2tf` | `sos2tf` | `ss2sos` | `ss2zp` | `tf2ss` | `zp2tf`

Introduced before R2006a

sscanf

Read formatted data from strings

Syntax

```
A = sscanf(str,formatSpec)
A = sscanf(str,formatSpec,sizeA)
[A,n] = sscanf( ___ )
[A,n,errmsg] = sscanf( ___ )
[A,n,errmsg,nextindex] = sscanf( ___ )
```

Description

`A = sscanf(str,formatSpec)` reads data from `str`, converts it according to the format specified by `formatSpec`, and returns the results in an array. `str` is either a character array or a string scalar. The `sscanf` function repeatedly applies `formatSpec` to sequences of characters in `str` until it either reaches the end of `str` or fails to match `formatSpec` to a sequence of characters. If `str` is a character array with more than one row, `sscanf` reads the characters in column order.

`A = sscanf(str,formatSpec,sizeA)` sets the size of the output array to be `sizeA` and then reads data from `str` into the output array. `sizeA` must be a positive integer or have the form `[m n]`, where `m` and `n` are positive integers.

`[A,n] = sscanf( ___ )` also returns the number of elements that `sscanf` successfully reads into `A`.

`[A,n,errmsg] = sscanf( ___ )` also returns a character vector containing an error message when `sscanf` fails to read all the data into `A`. If `sscanf` succeeds, then `errmsg` is an empty character vector.

`[A,n,errmsg,nextindex] = sscanf( ___ )` also returns the index of the position in `str` that immediately follows the last character scanned by `sscanf`.

Examples

Convert Character Vector to Numbers

Create a character vector that represents several numbers separated by whitespace characters. Convert the character vector to a column vector of numbers. `sscanf` treats whitespace characters as separators between numbers.

```
chr = '2.7183  3.1416  0.0073'
```

```
chr =
'2.7183  3.1416  0.0073'
```

```
A = sscanf(chr,'%f')
```

```
A = 3×1

    2.7183
    3.1416
    0.0073
```

Convert Text and Resize Output Array

Create a string that represents several numbers and convert it using `sscanf`. Specify the size of the output array.

Starting in R2017a, you can create strings using double quotes.

```
str = "2.7183  3.1416  0.0073"
```

```
str =
"2.7183  3.1416  0.0073"
```

```
A = sscanf(str,'%f',[1 3])
```

```
A = 1×3

    2.7183    3.1416    0.0073
```

Convert `str` to a 2-by-2 matrix. Because `str` represents only three numbers, `sscanf` pads `A` with enough zeroes to fill in the matrix.

```
A = sscanf(str,'%f',[2 2])
```

```
A = 2×2

    2.7183    0.0073
    3.1416         0
```

Count Elements Found in Text

Create a string that contains numbers separated by whitespace characters. Count the elements that `sscanf` puts into the output array when it converts the string to numbers.

Starting in R2017a, you can create strings using double quotes.

```
str = "78 72 64 66 49"
```

```
str =
"78 72 64 66 49"
```

Count the elements in the output array `A`. Convert the numbers in the string using the `%d` operator. `%d` matches integers separated by whitespace. To return the number of elements in `A`, specify a second output argument.

```
[A,n] = sscanf(str,'%d')
```

```
A = 5×1

    78
    72
    64
    66
    49


n = 5
```

Display Error Message

Create a string and read data from it. When `sscanf` fails to convert all of the input string, display the error message.

Starting in R2017a, you can create strings using double quotes.

```
str = "3.14159 are the first 6 digits of pi"
```

```
str =
"3.14159 are the first 6 digits of pi"
```

Convert the number in `str`. Since `str` also contains characters that `%f` cannot match, `sscanf` returns an error message. `sscanf` stops processing as soon as it encounters the word `'are'` because it cannot be converted to a number.

```
[A,n,errmsg] = sscanf(str,'%f')
```

```
A = 3.1416

n = 1

errmsg =
'Matching failure in format.'
```

Return Last Position Scanned

Create a character vector and read data from it. When `sscanf` fails to convert all of the input, return the index that immediately follows the position at which `sscanf` stopped. Use this index to display the unscanned input.

```
chr = '3.14159 are the first 6 digits of pi'
```

```
chr =
'3.14159 are the first 6 digits of pi'
```

Convert the data in `chr`. Return the index.

```
[A,~,~,nextindex] = sscanf(chr,'%f')
```

```
A = 3.1416

nextindex = 9
```

Display the characters from `chr` that `sscanf` did not scan.

```
chr(nextindex:end)
```

```
ans =
'are the first 6 digits of pi'
```

Match Specified Characters

Create a string that contains several temperatures, indicated by the degree symbol and the letter `F`. Convert the temperatures to a numeric array.

Starting in R2017a, you can create strings using double quotes. To insert the degree symbol (`char(176)`), use the `insertBefore` function.

```
T = "78F 72F 64F 66F 49F";
degreeSymbol = char(176);
T = insertBefore(T,'F',degreeSymbol)
```

```
T =
"78°F 72°F 64°F 66°F 49°F"
```

Return the temperatures as a numeric array.

```
A = sscanf(T,strcat("%d",degreeSymbol,"F"))
```

```
A = 5×1

    78
    72
    64
    66
    49
```

Input Arguments

`str` — Input text to scan
character array | string scalar

Input text to scan, specified as a character array or string scalar. If `str` is a character array, then it can have multiple rows, and `sscanf` reads the characters in column order.

Data Types: `char` | `string`

`formatSpec` — Format of input fields
formatting operators

Format of the input fields, specified using formatting operators. `formatSpec` can be a character vector in single quotes, or a string scalar.

Numeric Fields

This table lists available conversion specifiers to convert text to numeric outputs. `sscanf` converts values to their decimal (base 10) representation.

Numeric Field Type	Conversion Specifier	Description
Integer, signed	`%d`	Base 10.
	`%i`	The values determine the base: • The default is base 10. • If the initial digits are `0x` or `0X`, then the values are base 16 (hexadecimal). • If the initial digit is `0`, then values are base 8 (octal).
	`%ld` or `%li`	64-bit values, base 10, 8, or 16.
Integer, unsigned	`%u`	Base 10.
	`%o`	Base 8 (octal).
	`%x`	Base 16 (hexadecimal).
	`%lu`, `%lo`, `%lx`	64-bit values, base 10, 8, or 16.

Numeric Field Type	Conversion Specifier	Description
Floating-point number	`%f`, `%e`, or `%g`	Floating-point values. Input fields can contain any of the following (not case sensitive): `Inf`, `-Inf`, `NaN`, or `-NaN`. Input fields that represent floating-point numbers can include leading `+` or `-` symbols and exponential notation using `e` or `E`. The conversion specifiers `%f`, `%e`, and `%g` all treat input fields the same way.

Character Fields

This table lists available conversion specifiers to convert text so that the output is a character array.

Character Field Type	Conversion Specifier	Description
Character vector or string scalar	`%s`	Read the text until `sscanf` encounters whitespace.
	`%c`	Read any single character, including whitespace. To read multiple characters at a time, specify field width. For example, `%10c` reads 10 characters at a time.
Pattern-matching	`%[...]`	Read only characters in the brackets up to the first nonmatching character or whitespace. Example: `%[mus]` reads `'summer '` as `'summ'`.

If `formatSpec` contains a combination of numeric and character specifiers, then `sscanf` converts each character to its numeric equivalent.

Optional Operators

- Fields and Characters to Ignore

`sscanf` reads all numeric values and characters in sequence, unless you tell it to ignore a particular field or a portion of a field. To skip fields, insert an asterisk (*) after the percent sign (%). For example, to skip integers, specify `%*d`.

- Field Width

  To specify the maximum number of digits or text characters to read at a time, insert a number after the percent character. For example, `%10c` reads up to 10 characters at a time, including whitespace. `%4f` reads up to four digits at a time, including the decimal point.

- Literal Text to Ignore

  `sscanf` ignores specified text immediately before or after the conversion specifier.

  Example: `Level%u` reads `'Level1'` as `1`.

  Example: `%uStep` reads `'2Step'` as `2`.

Data Types: `char` | `string`

`sizeA` — Dimensions of output array
`Inf` (default) | integer | two-element row vector

Dimensions of the output array, `A`, specified as `Inf`, a positive integer, or a two-element row vector.

Form of `sizeA` Input	Description
`Inf`	Read input to the end. For numeric data, `A` is a column vector. For text data, `A` is a character vector.
`n`	Read at most `n` numeric values or space-delimited character fields. For numeric data, `A` is a column vector. For text data, `A` is a character vector.
`[m n]`	Read at most `m*n` numeric values or character fields. `n` can be `Inf`, but `m` cannot. `A` is `m`-by-`n`, filled in column order.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

A — Output data
column vector | matrix | character array

Output data, returned as a column vector, matrix, or character array. The class and size of `A` depend on the conversions specified by `formatSpec` and the size of the output array specified by `sizeA`:

- If `formatSpec` contains only numeric specifiers, then `A` is a numeric column vector. If you also specify the `sizeA` argument, then `A` is a matrix of the specified size, and is padded with zeroes if necessary. If the input contains fewer than `sizeA` values, then the size of `A` is smaller than `sizeA`. Instead, it is the size required to store the values scanned from the input.
  - If `formatSpec` contains only 64-bit signed integer specifiers, then `A` is of class `int64`.
  - If `formatSpec` contains only 64-bit unsigned integer specifiers, then `A` is of class `uint64`.
  - Otherwise, `A` is of class `double`.
- If `formatSpec` contains only `%c` or `%s` specifiers, then `A` is a character vector. If you also specify `sizeA`, then `A` is a character array, and is padded as necessary with null characters. (The null character is a control character with the value zero.) If the input contains fewer than `sizeA` characters, then the size of `A` is smaller than `sizeA`. Instead, it is the size required to store the characters scanned from the input.
- If `formatSpec` contains a combination of numeric and character specifiers, then `A` is numeric, of class `double`, and `sscanf` converts each character to its numeric equivalent. This conversion occurs even when `formatSpec` explicitly skips all numeric fields (for example, `formatSpec` is `'%*d %s'`).
- If `sscanf` cannot match all of the input to `formatSpec`, then `A` can be numeric or a character array. The class of `A` depends on the values that `sscanf` reads before it stops processing.

Data Types: `double` | `int64` | `uint64` | `char`

n — Number of elements read into output array
integer

Number of elements read into the output array, returned as an integer.

Data Types: `double`

`errmsg` — Error message
character vector

Error message, returned as a character vector. If `str` contains any data that `sscanf` cannot convert, then `errmsg` contains an error message. If `sscanf` converts all the data successfully, then `errmsg` is an empty character vector.

Data Types: `char`

`nextindex` — Position after last character scanned
integer

Position after the last character scanned, returned as an integer.

Data Types: `double`

Tips

- Format specifiers for the reading functions `sscanf` and `fscanf` differ from the formats for the writing functions `sprintf` and `fprintf`. The reading functions do not support a precision field. The width field specifies a minimum for writing, but a maximum for reading.

See Also

`compose` | `fprintf` | `fscanf` | `sprintf` | `textscan`

Topics

"Formatting Text"

Introduced before R2006a

stack

Stack data from multiple variables into single variable

Syntax

```
S = stack(U,vars)
S = stack(U,vars,Name,Value)
[S,iu] = stack( ___ )
```

Description

`S = stack(U,vars)` converts the table or timetable, `U`, into an equivalent table or timetable, `S`, that is stacked. The `stack` function stacks up multiple variables from `U`, specified by `vars`, into a single variable in `S`. In general, `S` contains fewer variables, but more rows, than `U`.

The output argument, `S`, contains a new categorical variable to indicate which variable in `U` the stacked data in each row came from. `stack` replicates data from the variables in `U` that are not stacked.

- If `U` is a table, then you cannot stack row names.
- If `U` is a timetable, then you cannot stack row times.

`S = stack(U,vars,Name,Value)` converts the table, `U`, with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify variable names for the new and stacked variables in `U`.

`[S,iu] = stack( ___ )` also returns an index vector, `iu`, indicating the correspondence between rows in `S` and rows in `U`. You can use any of the previous input arguments.

Examples

Stack Three Variables into One

Create a table containing test scores from three separate tests. The table is in unstacked format.

```
Test1 = [93;57;87;89];
Test2 = [89;77;92;86];
Test3 = [95;62;89;91];

U = table(Test1,Test2,Test3)
```

```
U=4×3 table
    Test1    Test2    Test3
    _____    _____    _____

     93       89       95
     57       77       62
     87       92       89
     89       86       91
```

The table contains four rows and three variables.

Stack the test scores into a single variable.

```
S = stack(U,1:3)
```

```
S=12×2 table
    Test1_Test2_Test3_Indicator    Test1_Test2_Test3
    ___________________________    _________________

                Test1                      93
                Test2                      89
                Test3                      95
                Test1                      57
                Test2                      77
                Test3                      62
                Test1                      87
                Test2                      92
                Test3                      89
                Test1                      89
                Test2                      86
                Test3                      91
```

S contains twelve rows and two variables. S is in stacked format.

The categorical variable, `Test1_Test2_Test3_Indicator`, identifies which test corresponds to the score in the stacked data variable, `Test1_Test2_Test3`.

Stack Variables and Specify Variable Names

Create a table indicating the amount of snowfall at three locations from five separate storms.

```
Storm = [1;2;3;4;5];
Date = {'12/25/11';'1/2/12';'1/23/12';'2/7/12';'2/15/12'};
Natick = [20;5;13;0;17];
Boston = [18;9;21;5;12];
Worcester = [26;10;16;3;15];

U = table(Storm,Date,Natick,Boston,Worcester)
```

```
U=5×5 table
    Storm        Date        Natick    Boston    Worcester
    _____    ____________    ______    ______    _________

      1      {'12/25/11'}      20        18         26
      2      {'1/2/12'  }       5         9         10
      3      {'1/23/12' }      13        21         16
      4      {'2/7/12'  }       0         5          3
      5      {'2/15/12' }      17        12         15
```

The variables `Storm` and `Date` contain data that is constant at each location.

Stack the variables `Natick`, `Boston`, and `Worcester` into a single variable. Name the variable containing the stacked data, `Snowfall`, and name the new indicator variable, `Town`.

```
S = stack(U,{'Natick','Boston','Worcester'},...
    'NewDataVariableName','Snowfall',...
    'IndexVariableName','Town')
```

```
S=15×4 table
    Storm        Date          Town       Snowfall
    _____    ____________    _________    ________

      1      {'12/25/11'}    Natick          20
```

```
    1      {'12/25/11'}    Boston          18
    1      {'12/25/11'}    Worcester       26
    2      {'1/2/12'  }    Natick           5
    2      {'1/2/12'  }    Boston           9
    2      {'1/2/12'  }    Worcester       10
    3      {'1/23/12' }    Natick          13
    3      {'1/23/12' }    Boston          21
    3      {'1/23/12' }    Worcester       16
    4      {'2/7/12'  }    Natick           0
    4      {'2/7/12'  }    Boston           5
    4      {'2/7/12'  }    Worcester        3
    5      {'2/15/12' }    Natick          17
    5      {'2/15/12' }    Boston          12
    5      {'2/15/12' }    Worcester       15
```

`S` contains three rows for each storm, and `stack` repeats the data in the constant variables, `Storm` and `Date`, accordingly.

The categorical variable, `Town`, identifies which variable in `U` contains the corresponding `Snowfall` data.

Stack Variables and Output Index Vector

Create a table containing estimated influenza rates along the east coast of the United States. Create a different variable for the Northeast, Mid Atlantic, and South Atlantic. Data Source: Google Flu Trends (https://www.google.org/flutrends).

```
Month = {'October';'November';'December';...
    'January';'February';'March'};
Year = [2005*ones(3,1); 2006*ones(3,1)];
NE = [1.1902; 1.3610; 1.5003; 1.7772; 2.1350; 2.2345];
MidAtl = [1.1865; 1.4120; 1.6043; 1.8830; 2.1227; 1.9920];
SAtl = [1.2730; 1.5820; 1.8625; 1.9540; 2.4803; 2.0203];

fluU = table(Month,Year,NE,MidAtl,SAtl)
```

```
fluU=6×5 table
       Month         Year      NE      MidAtl     SAtl
    ____________     ____    ______    ______    ______

    {'October' }     2005    1.1902    1.1865     1.273
```

```
    {'November'}    2005     1.361     1.412     1.582
    {'December'}    2005    1.5003    1.6043    1.8625
    {'January' }    2006    1.7772     1.883     1.954
    {'February'}    2006     2.135    2.1227    2.4803
    {'March'   }    2006    2.2345     1.992    2.0203
```

The variables `Month` and `Year` contain data that is constant across the row.

Stack the variables `NE`, `MidAtl`, and `SAtl` into a single variable called `FluRate`. Name the new indicator variable `Region` and output an index vector, `ifluU`, to indicate the correspondence between rows in the input unstacked table, `fluU`, and the output stacked table, `fluS`.

```
[fluS,ifluU] = stack(fluU,3:5,...
    'NewDataVariableName','FluRate',...
    'IndexVariableName','Region')
```

```
fluS=18×4 table
       Month         Year    Region    FluRate
    ____________     ____    ______    _______

    {'October' }     2005    NE        1.1902
    {'October' }     2005    MidAtl    1.1865
    {'October' }     2005    SAtl       1.273
    {'November'}     2005    NE         1.361
    {'November'}     2005    MidAtl     1.412
    {'November'}     2005    SAtl       1.582
    {'December'}     2005    NE        1.5003
    {'December'}     2005    MidAtl    1.6043
    {'December'}     2005    SAtl      1.8625
    {'January' }     2006    NE        1.7772
    {'January' }     2006    MidAtl     1.883
    {'January' }     2006    SAtl       1.954
    {'February'}     2006    NE         2.135
    {'February'}     2006    MidAtl    2.1227
    {'February'}     2006    SAtl      2.4803
    {'March'   }     2006    NE        2.2345
      ⋮


ifluU = 18×1

     1
     1
```

```
     1
     2
     2
     2
     3
     3
     3
     4
       ⋮
```

`ifluU(5)` is 2. The fifth row in the output table, `fluS`, contains data from the second row in the input table `fluU`.

Input Arguments

`U` — Input table
table | timetable

Input table, specified as a table or a timetable.

`vars` — Variables in `U` to stack
positive integer | vector of positive integers | character vector | cell array of character vectors | string array | logical vector

Variables in `U` to stack, specified as a positive integer, vector of positive integers, character vector, cell array of character vectors, string array, or logical vector.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'NewDataVariableName','StackedData'` names the new data variable `StackedData`.

`ConstantVariables` — Variables other than `vars` to include in the output
positive integer | vector of positive integers | character vector | cell array of character vectors | string array | logical vector

Variables other than `vars` to include in the output, specified as the comma-separated pair consisting of `'ConstantVariables'` and a positive integer, vector of positive integers, character vector, cell array of character vectors, string array, or logical vector. `stack` replicates the data from the constant variables for each stacked entry from a row.

The default is all the variables in `U` not specified by `vars`. You can specify the `'ConstantVariables'` name-value pair argument to exclude variables not specified by `vars` or `'ConstantVariables'` from the output table, `S`.

`U` can have row labels along its first dimension. If `U` is a table, then it can have row names as the labels. If `U` is a timetable, then it must have row times as the labels.

- You can include the row names or row times when you specify the value of `'ConstantVariables'`.
- `stack` replicates the row names or row times even when you do not include them in `'ConstantVariables'`.

`NewDataVariableName` — Name for the new data variable in `S`
character vector | string scalar

Name for the new data variable in `S`, specified as the comma-separated pair consisting of `'NewDataVariableName'` and a character vector or string scalar. The default is a concatenation of the names of the variables from `U` that are stacked up.

`IndexVariableName` — Name for the new indicator variable in `S`
character vector | string scalar

Name for the new indicator variable in `S`, specified as the comma-separated pair consisting of `'IndexVariableName'` and a character vector or string scalar. The default is a name based on `NewDataVariableName`.

Output Arguments

`S` — Output table
table | timetable

Stacked table, returned as a table or a timetable. `S` contains a stacked data variable, a categorical indicator variable, and any constant variables.

You can store additional metadata such as descriptions, variable units, variable names, and row names in `S`. For more information, see the Properties sections of `table` or `timetable`.

`stack` assigns the variable units and variable description property values from the first variable listed in `vars` to the corresponding `S.Properties.VariableUnits` and `S.Properties.VariableDescrisciptions` values for the new data variable.

`iu` — Index to `U`
column vector

Index to `U`, returned as a column vector. The index vector, `iu`, identifies the row in the input table, `U`, containing the corresponding data. `stack` creates the `j`th row in the output table, `S`, using `U(iu(j),vars)`.

Tips

- You can specify more than one group of data variables in `U`, and each group becomes a stacked data variable in `S`. Use a cell array to contain multiple values for `vars`, and a cell array of character vectors or string array to contain multiple values for the `'NewDataVariableName'` name-value pair argument. All groups must contain the same number of variables.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

See Also

`join` | `unstack`

Introduced in R2013b

StackedAxesProperties Properties

Appearance and behavior of individual axes in stacked plot

Description

`StackedAxesProperties` properties control the appearance and behavior of individual *y*-axes in a *stacked plot*. In a stacked plot, you can plot the variables of a table or timetable, or the columns of a matrix, in separate *y*-axes, stacked vertically. By changing property values, you can modify certain aspects of the stacked plot.

The `stackedplot` function returns a `StackedLineChart` object. In turn, this object has an array of `StackedAxesProperties` objects, named `AxesProperties`. To set properties of an individual *y*-axis, index into `AxesProperties` and access properties of the corresponding object.

You can use dot notation to query and set properties. For example, change *y*-limits of the third plot using its `YLimits` property.

```
T = readtable('outages.csv');
s = stackedplot(T);
c = s.AxesProperties(3).YLimits
s.AxesProperties(3).YLimits = [0 1e6];
```

Properties

`YLimits` — Minimum and maximum *y*-axis limits
two-element vector of the form `[min max]`

Minimum and maximum *y*-axis limits, specified as a two-element vector of the form `[min max]`, where `max` is greater than `min`. You can specify the limits as numeric, categorical, datetime, or duration values. However, the type of values that you specify must match the type of values along the axis.

You can specify both limits or you can specify one limit and let the axes automatically calculate the other. For an automatically calculated minimum or maximum limit, use `-inf` or `inf`, respectively.

Example: `s.AxesProperties(3).YLimits = [0 1e6]`

Example: `s.AxesProperties(3).YLimits = [-inf 1e6]`

Example: `s.AxesProperties(3).YLimits = [0 inf]`

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`LegendLabels` — Legend labels
cell array of character vectors | string array

Legend labels, specified as a cell array of character vectors or string array. If you do not specify labels, then the labels are variable names (for tables and timetables) or `'Column N'` (for the *N*th column of a matrix).

Legends do not display unless the `LegendVisible` property is set to `'on'`.

Example: `s.AxesProperties(1).LegendLabels = {'Label Text'}`

`LegendLocation` — Location with respect to axes
`'north'` | `'south'` | `'east'` | `'west'` | `'northeast'` | ...

Location with respect to the axes, specified as one of the location values listed in this table.

Value	Description
`'north'`	Inside top of axes
`'south'`	Inside bottom of axes
`'east'`	Inside right of axes
`'west'`	Inside left of axes
`'northeast'`	Inside top-right of axes (default for 2-D axes)
`'northwest'`	Inside top-left of axes
`'southeast'`	Inside bottom-right of axes
`'southwest'`	Inside bottom-left of axes

Example: `s.AxesProperties(2).LegendLocation = 'southeast'`

`LegendVisible` — State of legend visibility
`'on'` | `'off'`

State of legend visibility, specified as `'on'` or `'off'`. Set `LegendVisible` to `'on'` to display the legend or `'off'` to hide the legend.

Example: `s.AxesProperties(3).LegendVisible = 'on'`

See Also

Functions
`stackedplot`

Properties
StackedLineChart Properties | StackedLineProperties

Topics

"Access Property Values"
"Graphics Object Properties"

Introduced in R2018b

StackedLineChart Properties

Stacked plot appearance and behavior

Description

`StackedLineChart` properties control the appearance and behavior of a *stacked plot*. In a stacked plot, you can plot the variables of a table or timetable, or the columns of a matrix, in separate *y*-axes stacked vertically. By changing property values, you can modify certain aspects of the stacked plot.

You can use dot notation to query and set properties.

```
T = readtable('outages.csv');
s = stackedplot(T);
c = s.Color;
s.Color = 'red';
```

Properties

Table Data

`SourceTable` — Source table
table | timetable

Source table, specified as a table or a timetable.

You can create a table from workspace variables using the `table` function, or you can import data as a table using the `readtable` function. You can create a timetable from workspace variables using the `timetable` function.

Note The property is ignored and read-only when the plotted data comes from an array.

Array Data

`XData` — *x*-values
vector

x-values, specified as a vector. The number of elements in the vector must equal the number of rows in the value of the `YData` property.

Note The property is ignored and read-only when the plotted data comes from a table or timetable.

Example: `[1:10]`

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`YData` — *y*-values
array

y-values, specified as an array. The `stackedplot` function plots each column of the value of `YData` in its own *y*-axis.

Note The property is ignored and read-only when the plotted data comes from a table or timetable.

Example: `[1:10;5:5:50]`

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

Stacked Plot Display

`DisplayVariables` — Displayed variables
cell array of character vectors | string array | numeric array | logical array

Displayed variables, specified as a character vector, cell array of character vectors, string array, numeric array, or logical array. By setting this property you can plot a subset or a permutation of the variables in the input table or timetable.

If `DisplayVariables` is a cell array, it also can be a nested cell array. The `stackedplot` function plots all the variables specified in a nested cell in the same *y*-axis.

Example: `s.DisplayVariables = [1 3 4]` specifies the first, third, and fourth variables.

Example: `s.DisplayVariables = {{'Temp1','Temp2'},'Pressure'}` uses a nested cell array to specify that `Temp1` and `Temp2` are plotted together.

Example: `s.DisplayVariables = {{1,2},5}` specifies variables by number and plots the first and second variables together.

`DisplayLabels` — Labels for *y*-axes
cell array of character vectors | string array

Labels for the stacked *y*-axes values, specified as a cell array of character vectors or a string array. Specify one label for each value in `DisplayVariables`. By default, the values are the same as the values in `DisplayVariables`.

If you add a value, delete a value, or rearrange the values in `DisplayVariables`, then this property updates accordingly to maintain the pairings of values and labels.

Example: `s.DisplayLabels = {'Temperature','Humidity','Pressure'}`

`XVariable` — Table variable for *x*-axis
character vector | string | numeric scalar | logical vector

Table variable for the *x*-axis, specified in one of these forms:

- Character vector or string indicating the name of one table variable
- Numeric scalar indicating the index of one table variable
- Logical vector containing one `true` element

Note The property is ignored and read-only when the plotted data come from an array or a timetable.

`XLimits` — Minimum and maximum *x*-axis limits
two-element vector of the form `[min max]`

Minimum and maximum *x*-axis limits, specified as a two-element vector of the form `[min max]`, where `max` is greater than `min`. You can specify the limits as numeric, categorical, datetime, or duration values. However, the type of values that you specify must match the type of values along the axis.

You can specify both limits or you can specify one limit and let the axes automatically calculate the other. For an automatically calculated minimum or maximum limit, use `-inf` or `inf`, respectively.

Example: `s.XLimits = [0 100]`

Example: `s.XLimits = [-inf 100]`

Example: `s.XLimits = [0 inf]`

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `datetime` | `duration`

`AxesProperties` — Properties of *y*-axes
array of `StackedAxesProperties` objects

Properties of the *y*-axes, specified as an array of `StackedAxesProperties` objects. You can index into `AxesProperties` to change the *y*-axis properties for one variable in the stacked plot. For more information, see StackedAxesProperties.

Example: `s.AxesProperties(2).YLimits = [0 10]` specifies limits for the second plot in the stacked plot. The other plots are unaltered.

`LineProperties` — Properties of lines
array of `StackedLineProperties` objects

Properties of lines, specified as an array of `StackedLineProperties` objects. You can index into `LineProperties` to change the properties of a line for one variable in the stacked plot. For more information, see StackedLineProperties.

Example: `s.LineProperties(3).Color = 'red'` changes the line color of the third plot in the stacked plot. The other plots are unaltered.

`GridVisible` — Display of grid lines
`'on'` | `'off'`

Display of grid lines, specified as `'on'` or `'off'`.

Line

`Color` — Line color
`[0 0.4470 0.7410]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the first table.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: 'blue'

Example: [0 0 1]

Example: '#0000FF'

LineStyle — Line style

'-' (default) | '--' | ':' | '-.' | 'none'

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
'-'	Solid line	
'--'	Dashed line	
':'	Dotted line	
'-.'	Dash-dotted line	
'none'	No line	No line

LineWidth — Line width

0.5 (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

Markers

Marker — Marker symbol

'none' (default) | 'o' | '+' | '*' | '.' | ...

Marker symbol, specified as one of the values listed in this table. By default, the object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerSize` — Marker size
6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

`MarkerEdgeColor` — Marker outline color
`'none'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'none'`, an RGB triplet, a hexadecimal color code, or one of the color options listed in the first table.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

Example: `'#D2F9A7'`

`MarkerFaceColor` — Marker fill color

`'none'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'none'`, an RGB triplet, a hexadecimal color code, or one of the color options listed in the first table.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `[0.3 0.2 0.1]`

Example: `'green'`

Example: `'#D2F9A7'`

Font

`FontName` — Font name

supported font name | `'FixedWidth'`

Font name, specified as a supported font name or `'FixedWidth'`. To display and print text properly, you must choose a font that your system supports. The default font depends on your operating system and locale.

To use a fixed-width font that looks good in any locale, use `'FixedWidth'`. The fixed-width font relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`FontSize` — Font size

8 (default) | scalar numeric value

Font size, specified as a scalar numeric value. The font size affects the title, axis labels, and tick labels. It also affects any legends associated with the axes. The font size is measured in points.

Example: `s.FontSize = 12`

Labels

`Title` — Title of stacked plot

character vector | string scalar | numeric scalar | string array | numeric array | cell array | categorical array

Title of the stacked plot, specified as a character vector, string scalar, numeric scalar, string array, numeric array, cell array, or categorical array.

If the value of `Title` is an array, then the elements of the array are stacked vertically and displayed as the title. If the value is a cell array, then its elements can be character vectors, strings, or numbers.

Alternatively, use the `title` function to add a title.

```
title('My Title')
```

Example: `s.Title = 'A Title';`

Example: `s.Title = 137`

Example: `s.Title = {'Stacked plot',char(datetime('now'))};`

Example: `s.Title = {'Stacked plot',137};`

`XLabel` — Text for *x*-axis label

character vector | string scalar | numeric scalar | string array | numeric array | cell array | categorical array

Text for *x*-axis label, specified as a character vector, string scalar, numeric scalar, string array, numeric array, cell array, or categorical array.

If the value of `XLabel` is an array, then the elements of the array are stacked vertically and displayed as the *x*-axis label. If the value is a cell array, then its elements can be character vectors, strings, or numbers.

Alternatively, use the `xlabel` function to add an *x*-axis label.

```
xlabel('My x-Axis Label')
```

Example: `s.XLabel = 'An x-axis Label';`

Example: `s.XLabel = 137`

Example: `s.XLabel = {'An x-axis Label',char(datetime('now'))};`

Example: `s.XLabel = {'An x-axis Label',137};`

Position

`OuterPosition` — Outer size and location

`[0 0 1 1]` (default) | four-element vector

Outer size and location of the plot within the parent container (typically a figure, panel, or tab), specified as a four-element vector of the form `[left bottom width height]`.

- The `left` and `bottom` elements define the distance from the lower left corner of the container to the lower left corner of the plot.
- The `width` and `height` elements are the width and height of the rectangle that encloses the plot and the axes, including decorations such as the title and axis labels.

The default value of `[0 0 1 1]` covers the whole interior of the container. The units are normalized relative to the size of the container. To change the units, set the `Units` property.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`InnerPosition` — Inner size and location

four-element vector

Inner size and location of the plot within the parent container (typically a figure, panel, or tab) specified as a four-element vector of the form `[left bottom width height]`.

- The `left` and `bottom` elements define the distance from the lower left corner of the container to the lower left corner of the plot.
- The `width` and `height` elements are the width and height of the rectangle that encloses the plot and the axes, excluding decorations such as the title and axis labels.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`Position` — Inner size and location

four-element vector

Inner size and location of the plot within the parent container (typically a figure, panel, or tab) returned as a four-element vector of the form `[left bottom width height]`. This property is equivalent to the `InnerPosition` property.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`ActivePositionProperty` — Position property to hold constant

`'outerposition'` | `'innerposition'`

Position property to hold constant when adding, removing, or changing decorations, specified as one of the following values:

- `'outerposition'` — The `OuterPosition` property remains constant when you add, remove, or change decorations such as a title or an axis label. If any positional adjustments are needed, MATLAB adjusts the `InnerPosition` property.
- `'innerposition'` — The `InnerPosition` property remains constant when you add, remove, or change decorations such as a title or an axis label. If any positional adjustments are needed, MATLAB adjusts the `OuterPosition` property.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`Units` — Position units

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'points'` | `'pixels'` | `'characters'`

Position units, specified as one of these values.

Units	Description
`'normalized'` (default)	Normalized with respect to the container, which is typically the figure or a panel. The lower left corner of the container maps to `(0,0)` and the upper right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.

Units	Description
`'characters'`	Based on the default uicontrol font of the graphics root object: • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.
`'points'`	Typography points. One point equals 1/72 inch.
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems. • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. • On Linux systems, the size of a pixel is determined by your system resolution.

When specifying the units as a `Name,Value` pair during object creation, you must set the `Units` property before specifying the properties that you want to use these units, such as `Position`.

`Layout` — Layout options

empty `LayoutOptions` array (default) | `TiledChartLayoutOptions` object

Layout options, specified as a `TiledChartLayoutOptions` object. This property specifies options when the chart is a child of a tiled chart layout. Use this property to position the chart within the layout by setting the `Tile` and `TileSpan` properties on the `TiledChartLayoutOptions` object.

For example, this code places chart `c` in the third tile of a tiled chart layout.

```
c.Layout.Tile = 3;
```

To make the chart span multiple tiles, specify the `TileSpan` property as a two-element vector. For example, this chart spans `2` rows and `3` columns of tiles.

```
c.Layout.TileSpan = [2 3];
```

If the chart is not a child of a tiled chart layout (for example, if it is a child of a figure or panel), then this property is empty and has no effect.

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

Parent/Child

`Parent` — Parent container

`Figure` object | `Panel` object | `Tab` object | `TiledChartLayout` object

Parent container, specified as a `Figure`, `Panel`, `Tab`, or `TiledChartLayout` object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

See Also

Functions
`stackedplot`

Properties
StackedAxesProperties | StackedLineProperties

Topics

"Access Property Values"
"Graphics Object Properties"

Introduced in R2018b

StackedLineProperties Properties

Appearance and behavior of individual lines in stacked plot

Description

`StackedLineProperties` properties control the appearance and behavior of individual lines in a *stacked plot*. In a stacked plot, you can plot the variables of a table or timetable, or the columns of a matrix, in separate *y*-axes, stacked vertically. By changing property values, you can modify certain aspects of the stacked plot.

The `stackedplot` function returns a `StackedLineChart` object. In turn, this object has an array of `StackedLineProperties` objects, named `LineProperties`. To set properties of an individual line, index into `LineProperties` and access properties of the corresponding object.

You can use dot notation to query and set properties. For example, change the third plot to a scatter plot using its `PlotType` property.

```
T = readtable('outages.csv');
s = stackedplot(T);
c = s.LineProperties(3).PlotType
s.LineProperties(3).PlotType = 'scatter';
```

Properties

`Color` — Line color

`[0 0.4470 0.7410]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the first table.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

`Marker` — Marker symbol
`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | ...

Marker symbol, specified as one of the values listed in this table. By default, the object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross

Value	Description
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerEdgeColor` — Marker outline color

`'none'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'none'`, an RGB triplet, a hexadecimal color code, or one of the color options listed in the first table.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `[0.5 0.5 0.5]`

Example: `'blue'`

Example: `'#D2F9A7'`

`MarkerFaceColor` — Marker fill color

`'none'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'none'`, an RGB triplet, a hexadecimal color code, or one of the color options listed in the first table.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: [0.3 0.2 0.1]

Example: 'green'

Example: '#D2F9A7'

MarkerSize — Marker size
6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

PlotType — Plot type
'plot' (default) | 'stairs' | 'scatter'

Plot type, specified as 'plot', 'stairs', or 'scatter'.

See Also

Functions
stackedplot

Properties
StackedAxesProperties | StackedLineChart Properties

Topics

"Access Property Values"
"Graphics Object Properties"

Introduced in R2018b

stackedplot

Stacked plot of several variables with common x-axis

Syntax

```
s = stackedplot(tbl)
s = stackedplot(tbl,vars)
s = stackedplot( ___ ,'XVariable',xvar)

s = stackedplot(X,Y)
s = stackedplot(Y)

s = stackedplot( ___ ,LineSpec)
s = stackedplot( ___ ,Name,Value)
s = stackedplot(parent, ___ )
```

Description

`s = stackedplot(tbl)` plots the variables of a table or timetable in a stacked plot and returns a `StackedLineChart` object. You can use `s` to change properties of the stacked plot after you have created it. For a list of properties, see StackedLineChart.

The function plots the variables in separate *y*-axes, stacked vertically. The variables share a common *x*-axis.

- If `tbl` is a table, then the function plots the variables against row numbers.
- If `tbl` is a timetable, then the function plots the variables against row times.

The `stackedplot` function plots all the numeric, logical, categorical, datetime, and duration variables of `tbl`, and ignores table variables having any other data type.

`s = stackedplot(tbl,vars)` plots only the table or timetable variables specified by `vars`.

`s = stackedplot( ___ ,'XVariable',xvar)` specifies the table variable that provides the *x*-values for the stacked plot. This syntax supports only tables, and not timetables.

`s = stackedplot(X,Y)` plots the columns of `Y` versus the vector `X`.

`s = stackedplot(Y)` plots the columns of `Y` versus their row number. The *x*-axis scale ranges from 1 to the number of rows in `Y`.

`s = stackedplot( ___ ,LineSpec)` sets the line style, marker symbol, and color. You can use this syntax with the input arguments of any of the previous syntaxes.

`s = stackedplot( ___ ,Name,Value)` sets properties for the stacked plot using one or more `Name,Value` pair arguments. For a list of properties, see Stacked Line Chart. Use this option with any of the input argument combinations in the previous syntaxes. Name-value pair settings apply to all the plots in the stacked plot. Enclose each property name in quotes.

`s = stackedplot(parent, ___ )` creates the stacked plot in the figure, panel, or tab specified by `parent`. The option `parent` can precede any of the input argument combinations in the previous syntaxes.

Examples

Plot Timetable Variables

Read data from a spreadsheet to a table. Then convert the table to a timetable. The first variable that contains dates and times, `OutageTime`, provides the row times for the timetable. Display the first five rows.

```
tbl = readtable('outages.csv');
tbl = table2timetable(tbl);
head(tbl,5)
```

```
ans=5×6 timetable
       OutageTime          Region         Loss     Customers     RestorationTime      _

    ________________    _____________    ______    __________    ________________    __

    2002-02-01 12:18    {'SouthWest'}    458.98    1.8202e+06    2002-02-07 16:50    {
    2003-01-23 00:49    {'SouthEast'}    530.14    2.1204e+05                 NaT    {
```

```
    2003-02-07 21:15    {'SouthEast'}     289.4    1.4294e+05    2003-02-17 08:14    {
    2004-04-06 05:44    {'West'     }    434.81    3.4037e+05    2004-04-06 06:10    {
    2002-03-16 06:18    {'MidWest'  }    186.44    2.1275e+05    2002-03-18 23:23    {
```

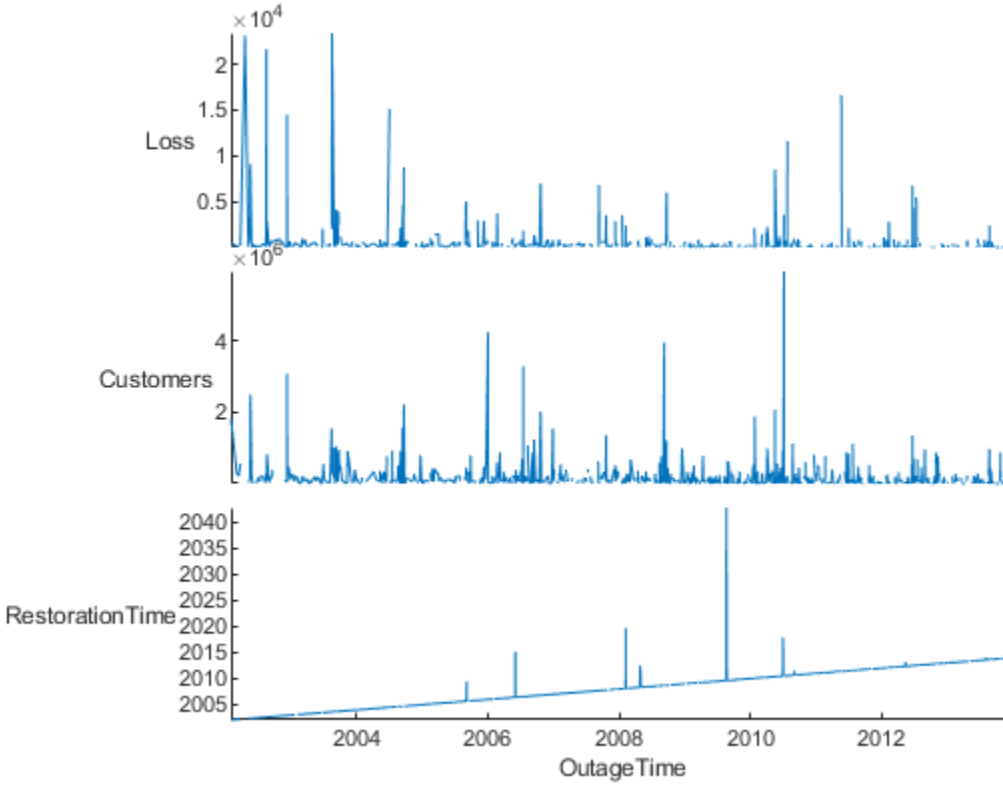
Sort the timetable so that its row times are in order. The row times of a timetable do not need to be in order. However, if you use the row times as the *x*-axis of a plot, then it is better to ensure the timetable is sorted by its row times.

```
tbl = sortrows(tbl);
head(tbl,5)
```

```
ans=5×6 timetable
       OutageTime          Region         Loss     Customers     RestorationTime
    ________________    _____________    ______    __________    _________________    _

    2002-02-01 12:18    {'SouthWest'}    458.98    1.8202e+06    2002-02-07 16:50    {
    2002-03-05 17:53    {'MidWest'  }    96.563    2.8666e+05    2002-03-10 14:41    {
    2002-03-16 06:18    {'MidWest'  }    186.44    2.1275e+05    2002-03-18 23:23    {
    2002-03-26 01:59    {'MidWest'  }    388.04    5.6422e+05    2002-03-28 19:55    {
    2002-04-20 16:46    {'MidWest'  }     23141           NaN                 NaT    {
```

Create a stacked plot of data from `tbl`. The row times, `OutageTime`, provide the values along the *x*-axis. The `stackedplot` function plots the values from the `Loss`, `Customers`, and `RestorationTime` variables, with each variable plotted along its own y-axis. However, the plot does not include the `Region` and `Cause` variables because they contain data that cannot be plotted.

```
s = stackedplot(tbl);
```



Specify Variables

Create a table from patient data. Display the first three rows.

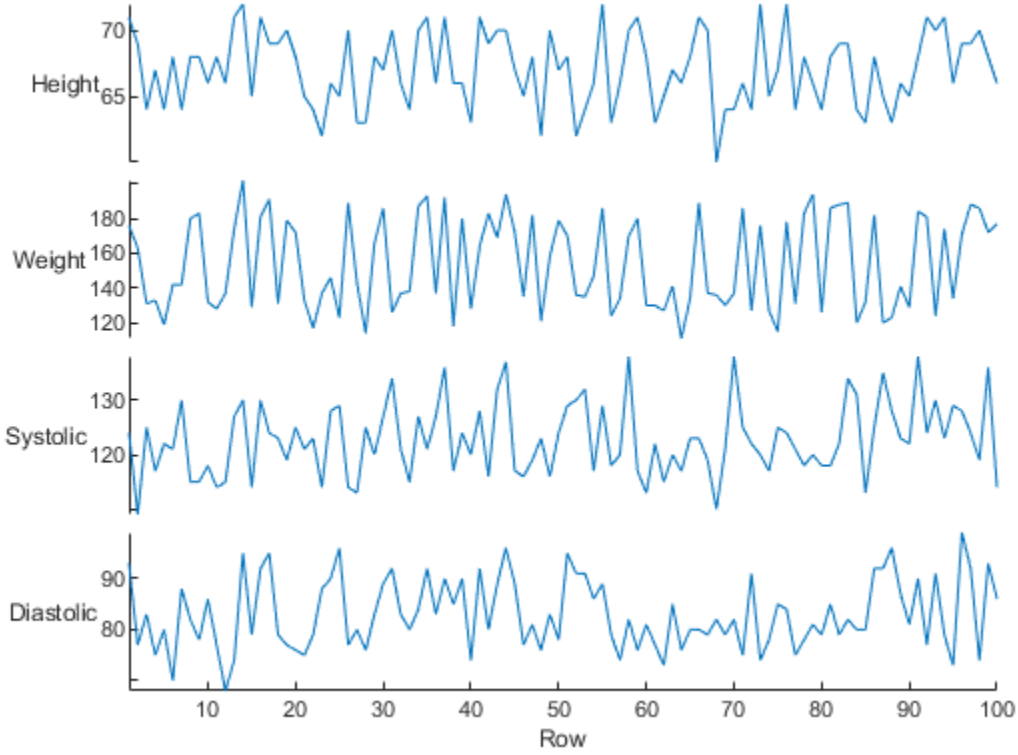
```
tbl = readtable('patients.xls');
head(tbl,3)
```

```
ans=3×10 table
      LastName         Gender      Age              Location                Height    We
    ____________    __________    ___    ________________________________    ______    __

    {'Smith'   }    {'Male'  }    38     {'County General Hospital'  }       71       1
```

```
    {'Johnson' }    {'Male'  }    43     {'VA Hospital'              }      69       16
    {'Williams'}    {'Female'}    38     {'St. Mary's Medical Center'}      64       13
```

Plot only four of the variables from the table.

```
stackedplot(tbl,{'Height','Weight','Systolic','Diastolic'});
```



Reorder Variables

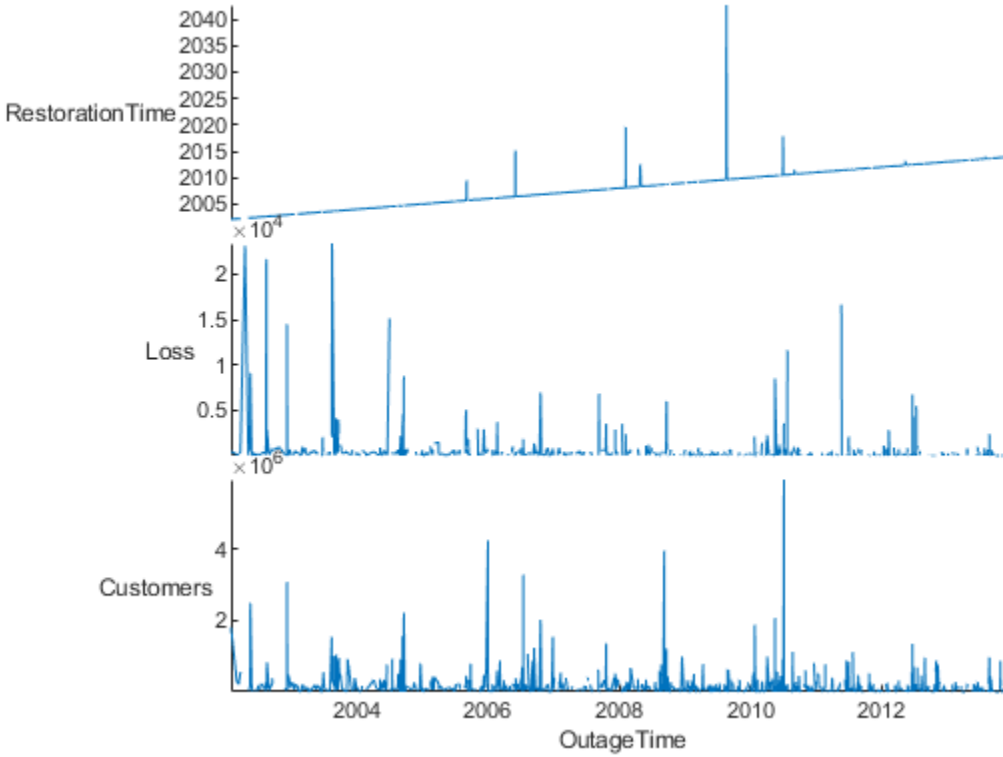
Create a timetable and display its first three rows.

```
tbl = readtable('outages.csv');
tbl = table2timetable(tbl);
tbl = sortrows(tbl);
head(tbl,3)
```

```
ans=3×6 timetable
       OutageTime          Region         Loss     Customers     RestorationTime

    ________________    _____________    ______    __________    ________________    __

    2002-02-01 12:18    {'SouthWest'}    458.98    1.8202e+06    2002-02-07 16:50    {
    2002-03-05 17:53    {'MidWest'  }    96.563    2.8666e+05    2002-03-10 14:41    {
    2002-03-16 06:18    {'MidWest'  }    186.44    2.1275e+05    2002-03-18 23:23    {
```

Reorder the variables by specifying them in an order that differs from their order in the table. For example, `RestorationTime` is the last variable in the timetable that can be plotted. By default, `stackedplot` places it at the bottom of the plot. But you can reorder the variables to put `RestorationTime` at the top.

```
stackedplot(tbl,{'RestorationTime','Loss','Customers'});
```



There are also other ways to reorder the variables.

- Specify them by their numeric order in the table: `stackedplot(tbl,[4 2 3]);`
- Reorder the values in the `DisplayVariables` property of the `StackedLineChart` object: `s = stackedplot(tbl); s.DisplayVariables = {'RestorationTime','Loss','Customers'}`

Plot Multiple Variables Using One Y-Axis

Create a table from a subset of patient data, using the `Weight`, `Systolic`, and `Diastolic` variables.

```
load patients
tbl = table(Weight,Systolic,Diastolic);
head(tbl,3)
```

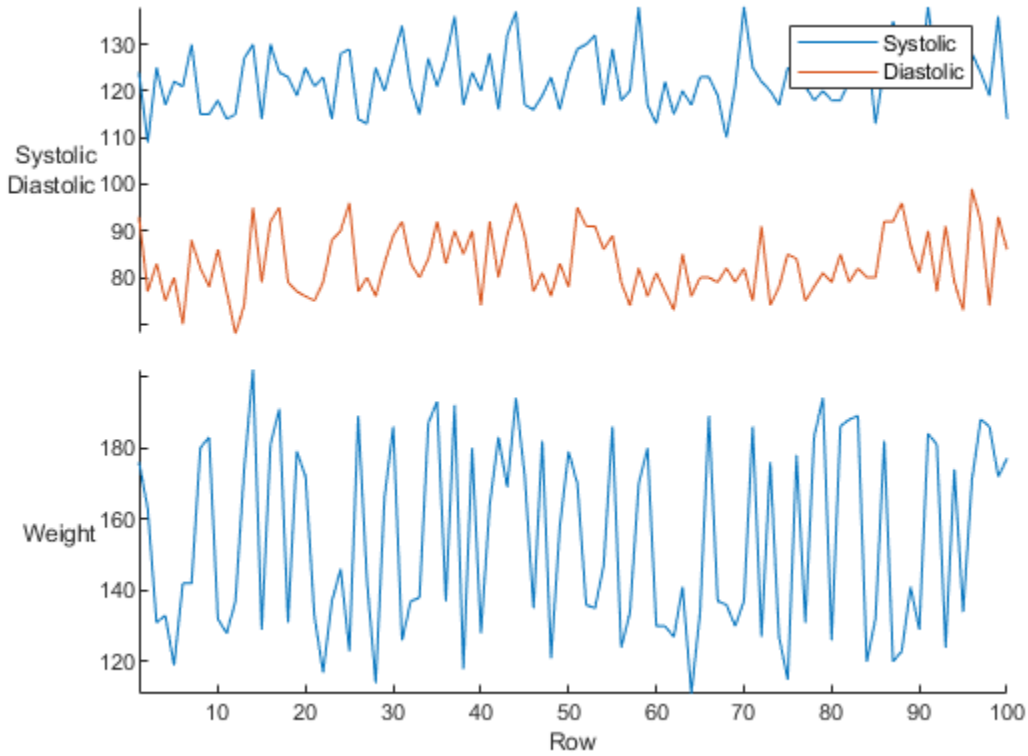
```
ans=3×3 table
    Weight    Systolic    Diastolic
    ______    ________    _________

     176        124          93
     163        109          77
     131        125          83
```

Create a stacked plot, with `Systolic` and `Diastolic` plotted using the same *y*-axis. To plot variables together, specify them within a nested cell array.

```
vars = {{'Systolic','Diastolic'},'Weight'}
```

```
vars=1×2 cell
    {1x2 cell}    {'Weight'}
```

```
stackedplot(tbl,vars);
```



Plot Columns of Matrix

Create a numeric matrix and a numeric vector.

```
X = [0:4:20]
```

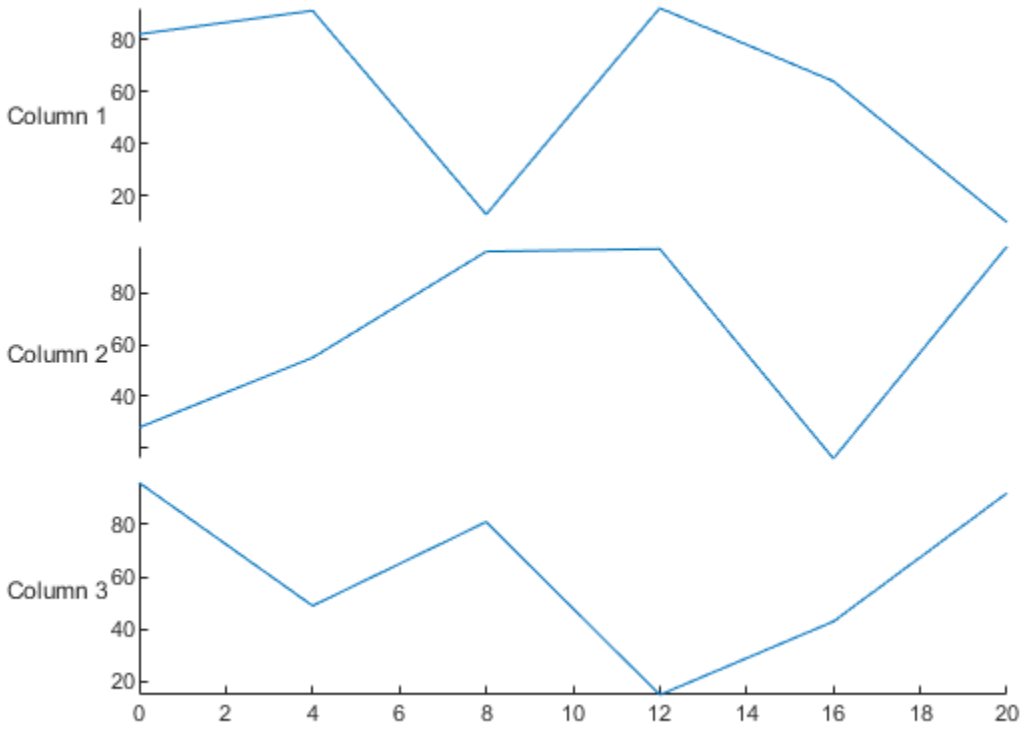
```
X = 1×6

     0     4     8    12    16    20
```

```
Y = randi(100,6,3)
```

```
Y = 6×3

    82    28    96
    91    55    49
    13    96    81
    92    97    15
    64    16    43
    10    98    92
```

Create a stacked plot using X and Y.

```
stackedplot(X,Y);
```



Specify Title and Labels Using Name-Value Pairs

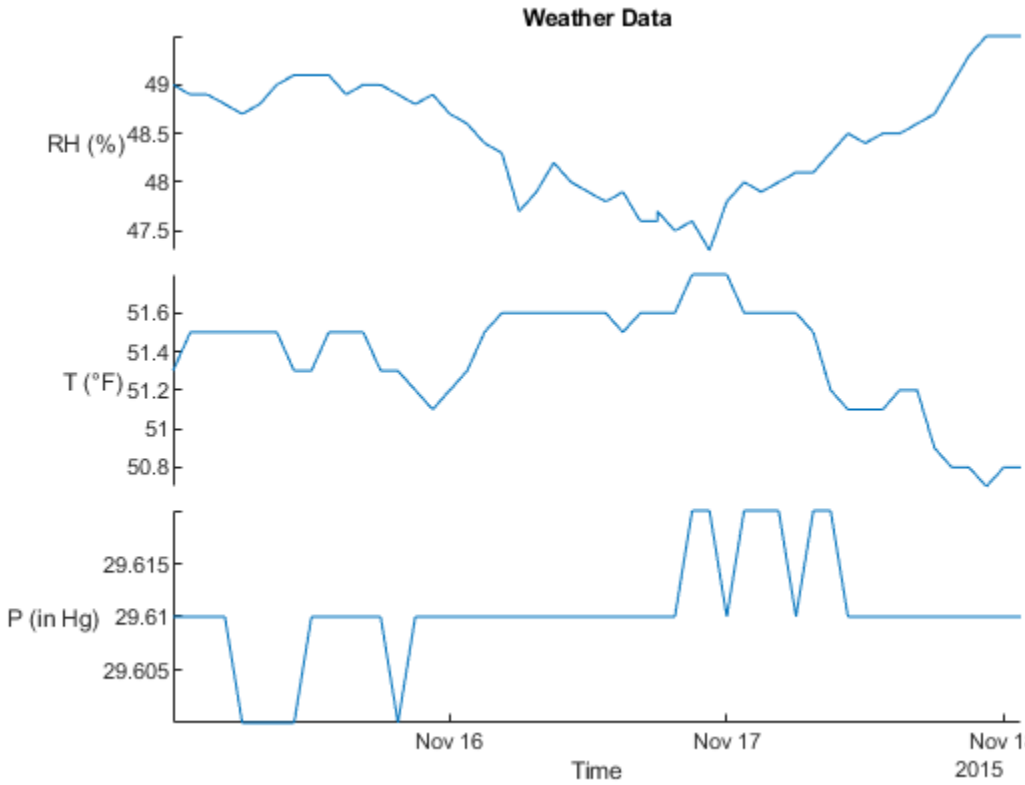
Load a timetable that has a set of weather measurements. Display its first three rows.

```
load outdoors
outdoors(1:3,:)
```

```
ans=3×4 timetable
           Time            Humidity    TemperatureF    PressureHg
    ___________________    ________    ____________    __________

    2015-11-15 00:00:24        49          51.3          29.61
    2015-11-15 01:30:24      48.9          51.5          29.61
    2015-11-15 03:00:24      48.9          51.5          29.61
```

Create a stacked plot. Specify the title and labels for the *y*-axes using name-value pair arguments. You can use name-values pairs to change any properties from their defaults values. (Also note that you can specify the degree symbol using `char(176)`.)

```
degreeSymbol = char(176);
newYlabels = {'RH (%)',['T (' degreeSymbol 'F)'],'P (in Hg)'};
stackedplot(outdoors,'Title','Weather Data','DisplayLabels',newYlabels);
```



Change Individual Plots to Scatter and Stair Plots

The `stackedplot` function returns a `StackedLineChart` object. You can use it to set the same property value for all plots, or to set different property values for individual plots. In this example, first change the line widths for all plots in a stacked plot. Then, use the `PlotType` property of individual plots, so that the stacked plot has a line plot, scatter plot, and stair plot.

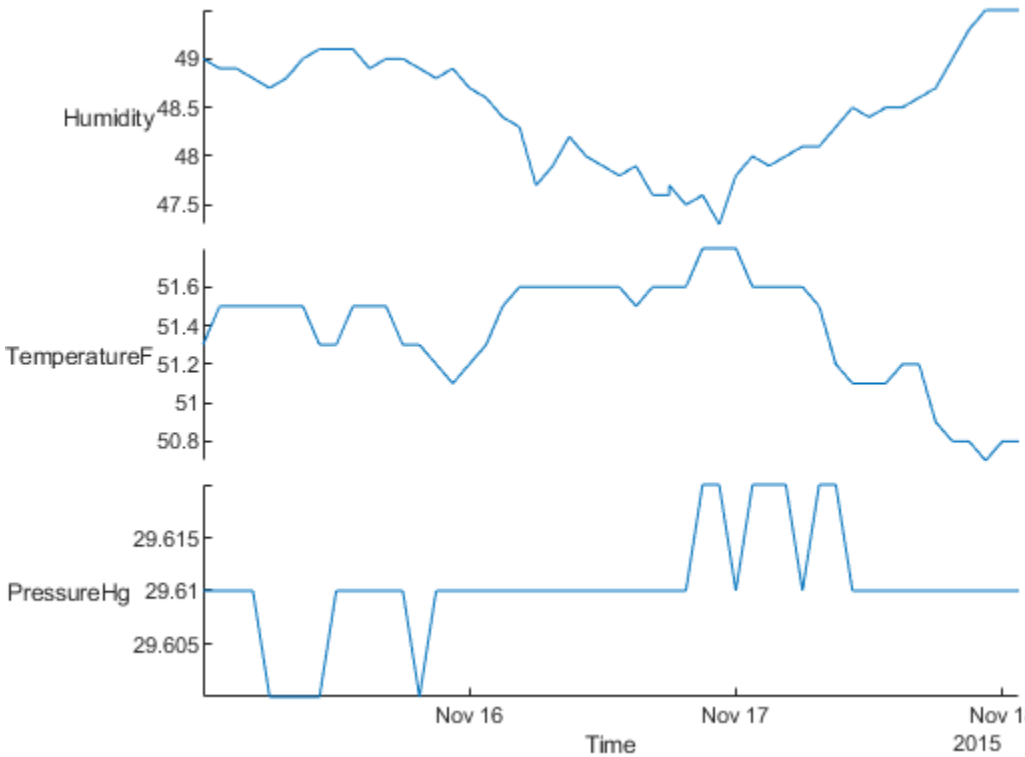
Load a timetable that has a set of weather measurements.

```
load outdoors
outdoors(1:3,:)
```

```
ans=3×4 timetable
           Time            Humidity    TemperatureF    PressureHg
    ___________________    ________    ____________    __________

    2015-11-15 00:00:24        49          51.3          29.61
    2015-11-15 01:30:24      48.9          51.5          29.61
    2015-11-15 03:00:24      48.9          51.5          29.61
```

Create a stacked plot and return a `StackedLineChart` object.

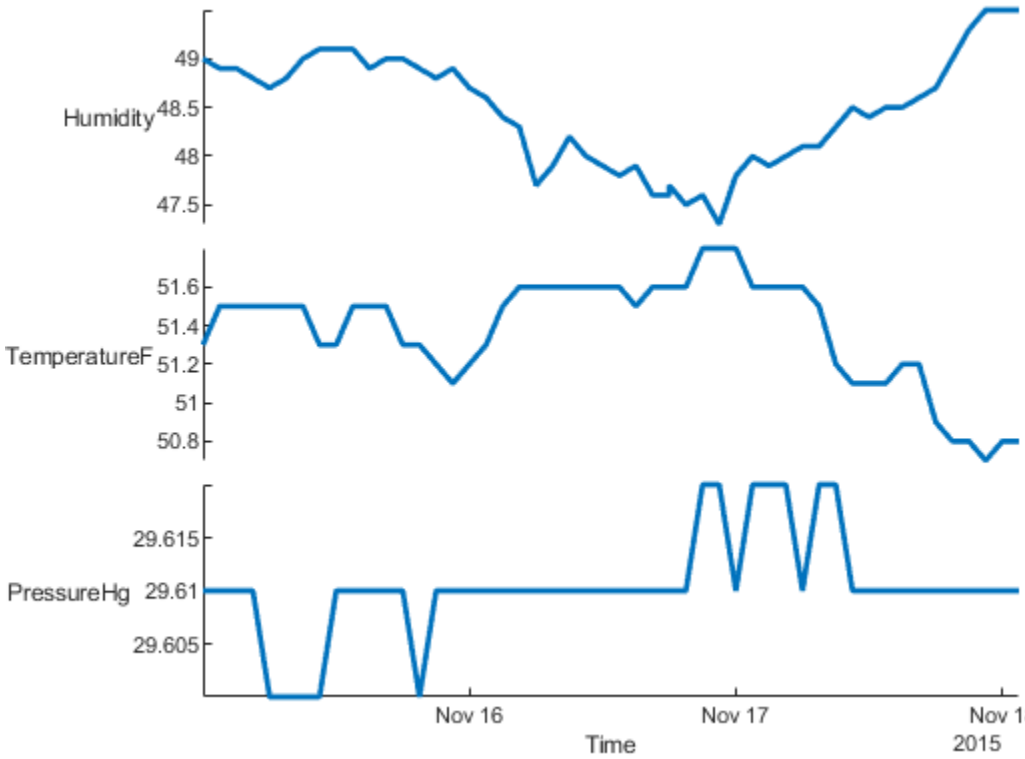
```
s = stackedplot(outdoors)
```



```
s =
  StackedLineChart with properties:

         SourceTable: [51x3 timetable]
    DisplayVariables: {'Humidity'  'TemperatureF'  'PressureHg'}
               Color: [0 0.4470 0.7410]
           LineStyle: '-'
           LineWidth: 0.5000
              Marker: 'none'
          MarkerSize: 6

  Show all properties
```

The object provides access to many properties that apply to all of the plots. For example, you can use `s.LineWidth` to make the lines wider.

```
s.LineWidth = 2;
```



The object also provides access to arrays of objects that you can use to modify the lines and *y*-axes for individual plots. To access properties of individual lines, use `s.LineProperties`. For each plot, you can specify a different line style, marker, plot type, and so on.

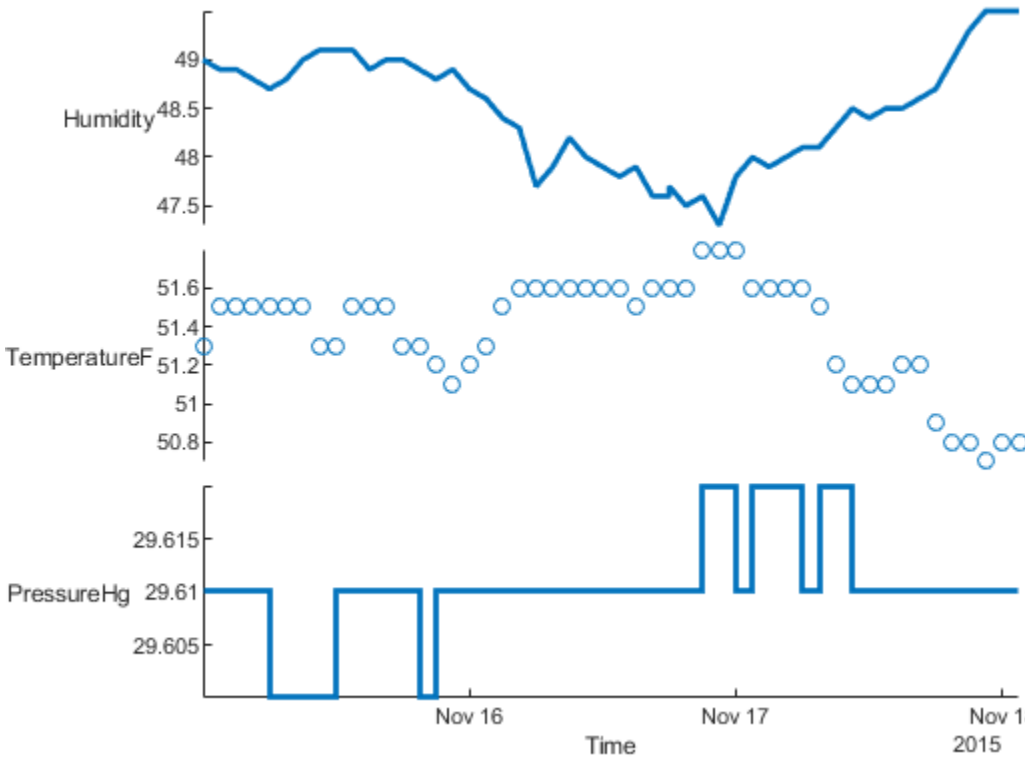
```
s.LineProperties
```

```
ans=3×1 object
  3x1 StackedLineProperties array with properties:

    Color
    MarkerFaceColor
    MarkerEdgeColor
    LineStyle
```

```
LineWidth
Marker
MarkerSize
PlotType
```

Change the second plot to a scatter plot, and the third plot to a stair plot, using the `PlotType` property.

```
s.LineProperties(2).PlotType = 'scatter';
s.LineProperties(3).PlotType = 'stairs';
```



You also can access individual *y*-axes through the `s.AxesProperties` property.

```
s.AxesProperties
```

```
ans=3×1 object
  3x1 StackedAxesProperties array with properties:

    YLimits
    LegendLabels
    LegendLocation
    LegendVisible
```

Input Arguments

`tbl` — Input table or timetable
table | timetable

Input table or timetable.

`vars` — Variables in input table or timetable
cell array of character vectors | string array | numeric array | logical array

Variables in the input table, specified as a cell array of character vectors, string array, numeric array, or logical array.

If `vars` is a cell array, it also can be a nested cell array. The `stackedplot` function plots all the variables specified in a nested cell in the same *y*-axis.

`xvar` — Table variable that contains *x*-values
character vector | string scalar | integer | logical array

Table variable that contains *x*-values, specified as a character vector, string scalar, integer, or logical array.

You can specify `xvar` only when the input argument `tbl` is a table, not a timetable.

`X` — *x*-values
numeric vector | datetime vector | duration vector | logical vector

x-values, specified as a numeric, datetime, duration, or logical vector. The length of `X` must equal the number of rows of `Y`.

`Y` — *y*-values
numeric array | datetime array | duration array | categorical array | logical array

y-values, specified as a numeric, datetime, duration, categorical, or logical array. The `stackedplot` function plots each column in a separate *y*-axis.

`LineSpec` — Line style, marker, and color

character vector | string scalar

Line style, marker, and color, specified as a character vector or string scalar containing symbols. The symbols can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the marker and no line.

Example: `'--or'` is a red dashed line with circle markers

Line Style	Description
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Marker	Description
`o`	Circle
`+`	Plus sign
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Pentagram
`h`	Hexagram

Color	Description
y	yellow
m	magenta
c	cyan
r	red
g	green
b	blue
w	white
k	black

parent — Parent container
Figure object | Panel object | Tab object | TiledChartLayout object

Parent container, specified as a Figure, Panel, Tab, or TiledChartLayout object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

The stacked chart line properties listed here are only a subset common to all stacked plots, whether the data source is a table or array. For a complete list, see StackedLineChart Properties.

Example: 'Marker','o','MarkerSize',10

Color — Line color
[0 0.4470 0.7410] (default) | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...

Line color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the first table.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: 'blue'

Example: [0 0 1]

Example: '#0000FF'

LineStyle — Line style
'-' (default) | '--' | ':' | '-.' | 'none'

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
'-'	Solid line	
'--'	Dashed line	
':'	Dotted line	
'-.'	Dash-dotted line	
'none'	No line	No line

LineWidth — Line width
0.5 (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

Marker — Marker symbol
'none' (default) | 'o' | '+' | '*' | '.' | 'x' | ...

Marker symbol, specified as one of the markers in this table. By default, a stacked chart line does not have markers. Add markers at each data point along the line by specifying a marker symbol.

Value	Description
'o'	Circle

Value	Description
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'Marker','+'`

Example: `'Marker','diamond'`

`MarkerSize` — Marker size
6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

Tips

- To interactively explore the data in your stacked plot, use these features.
  - Zoom — Use the scroll wheel to zoom.
  - Pan — Click and drag the stacked plot to pan across the *x*-values.
  - Data cursor — Hover over a location to display *y*-values for each plot.

See Also

Functions
`plot` | `scatter` | `stairs` | `table` | `timetable`

Properties
StackedAxesProperties | StackedLineChart | StackedLineProperties

Topics
"Access Property Values"
"Graphics Object Properties"

Introduced in R2018b

stairs

Stairstep graph

Syntax

```
stairs(Y)
stairs(X,Y)
stairs( ___ ,LineSpec)
stairs( ___ ,Name,Value)

stairs(ax, ___ )

h = stairs( ___ )

[xb,yb] = stairs( ___ )
```

Description

`stairs(Y)` draws a stairstep graph of the elements in `Y`.

- If `Y` is a vector, then `stairs` draws one line.
- If `Y` is a matrix, then `stairs` draws one line per matrix column.

`stairs(X,Y)` plots the elements in `Y` at the locations specified by `X`. The inputs `X` and `Y` must be vectors or matrices of the same size. Additionally, `X` can be a row or column vector and `Y` must be a matrix with `length(X)` rows.

`stairs( ___ ,LineSpec)` specifies a line style, marker symbol, and color. For example, `':*r'` specifies a dotted red line with asterisk markers. Use this option with any of the input argument combinations in the previous syntaxes.

`stairs( ___ ,Name,Value)` modifies the stairstep chart using one or more name-value pair arguments. For example, `'Marker','o','MarkerSize',8` specifies 8 point circle markers.

`stairs(ax, ___ )` plots into the axes specified by `ax` instead of into the current axes (`gca`). The option, `ax`, can precede any of the input argument combinations in the previous syntaxes.

`h = stairs( ___ )` returns one or more `Stair` objects. Use `h` to make changes to properties of a specific `Stair` object after it is created.

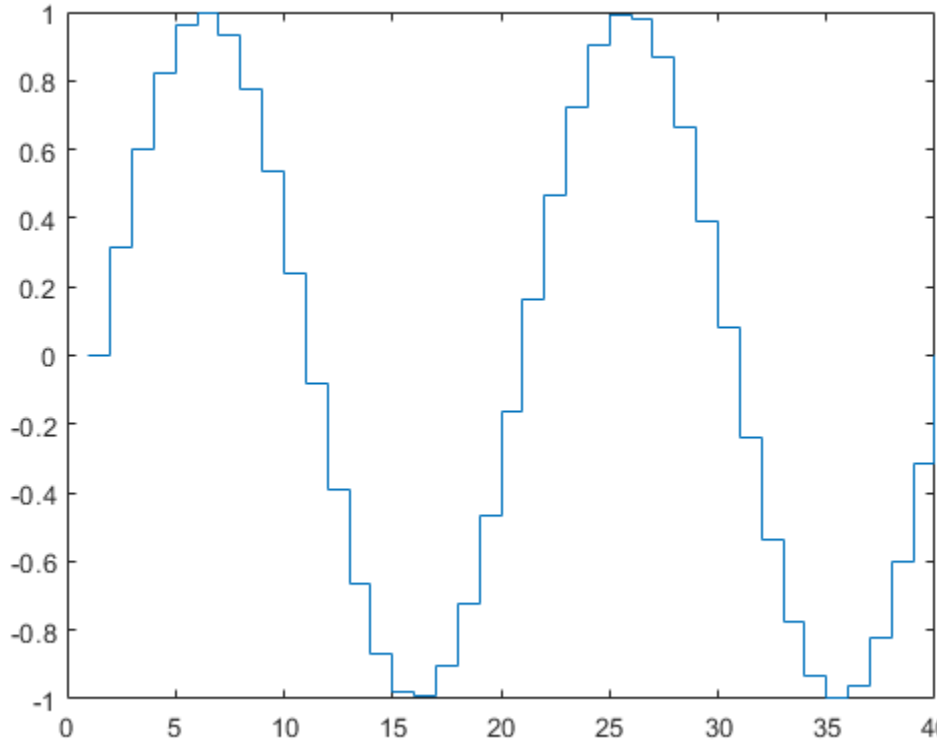
`[xb,yb] = stairs( ___ )` does not create a plot, but returns matrices `xb` and `yb` of the same size, such that `plot(xb,yb)` plots the stairstep graph.

Examples

Plot Single Data Series

Create a stairstep plot of sine evaluated at 40 equally spaced values between 0 and 4π.

```
X = linspace(0,4*pi,40);
Y = sin(X);

figure
stairs(Y)
```

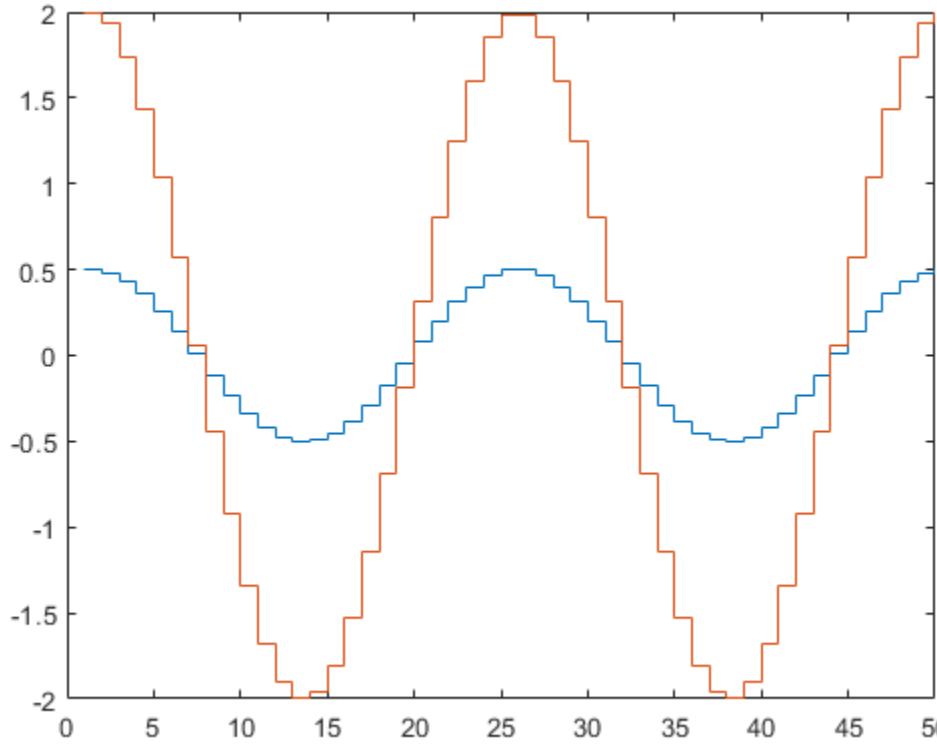


The length of Y automatically determines and generates the *x*-axis scale.

Plot Multiple Data Series

Create a stairstep plot of two cosine functions evaluated at 50 equally spaced values between 0 and 4π.

```
X = linspace(0,4*pi,50)';
Y = [0.5*cos(X), 2*cos(X)];

figure
stairs(Y)
```

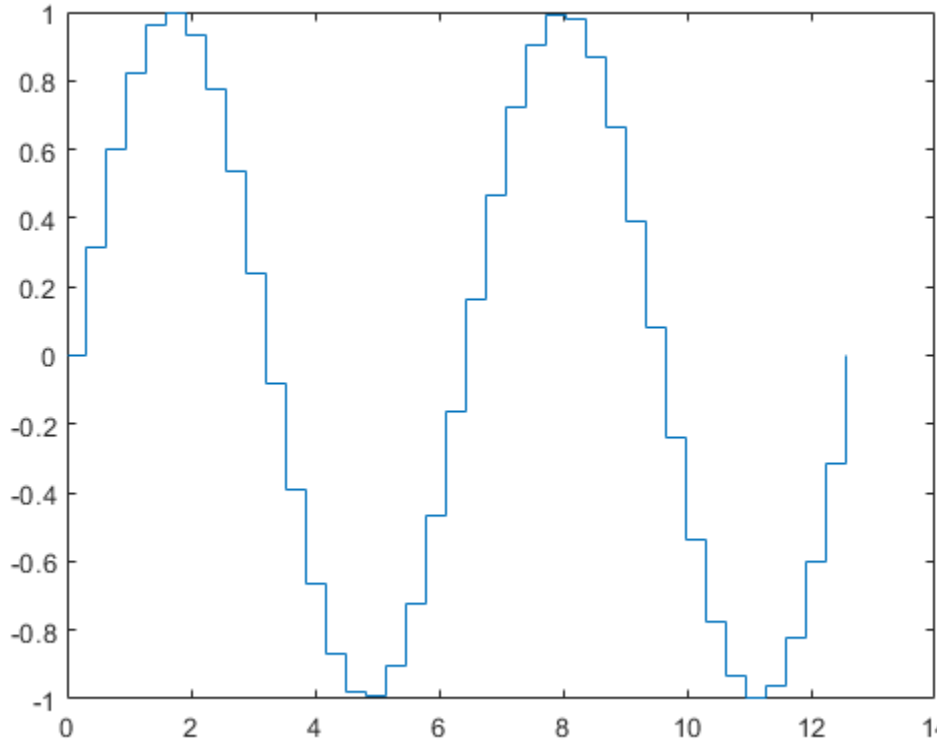


The number of rows in Y automatically determines and generates the *x*-axis scale.

Plot Single Data Series at Specified x-Values

Create a stairstep plot of a sine wave evaluated at equally spaced values between 0 and 4π. Specify the set of *x*-values for the plot.

```
X = linspace(0,4*pi,40);
Y = sin(X);

figure
stairs(X,Y)
```

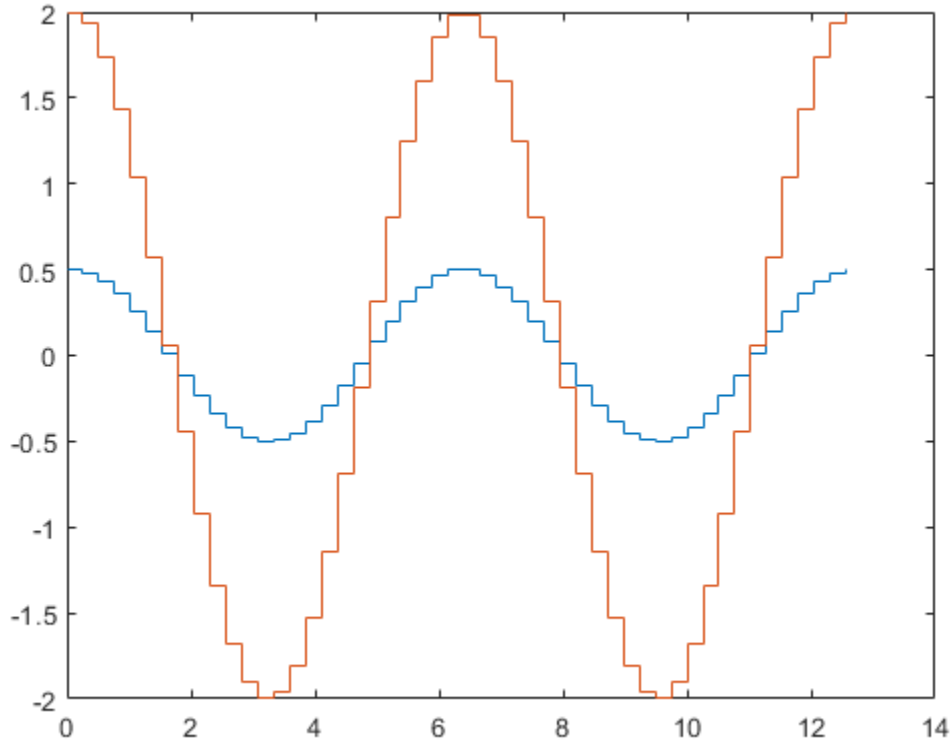


The entries in Y are plotted against the corresponding entries in X.

Plot Multiple Data Series at Specified x-Values

Create a stairstep plot of two cosine waves evaluated at equally spaced values between 0 and 4π. Specify the set of x-values for the plot.

```
X = linspace(0,4*pi,50)';
Y = [0.5*cos(X), 2*cos(X)];

figure
stairs(X,Y)
```



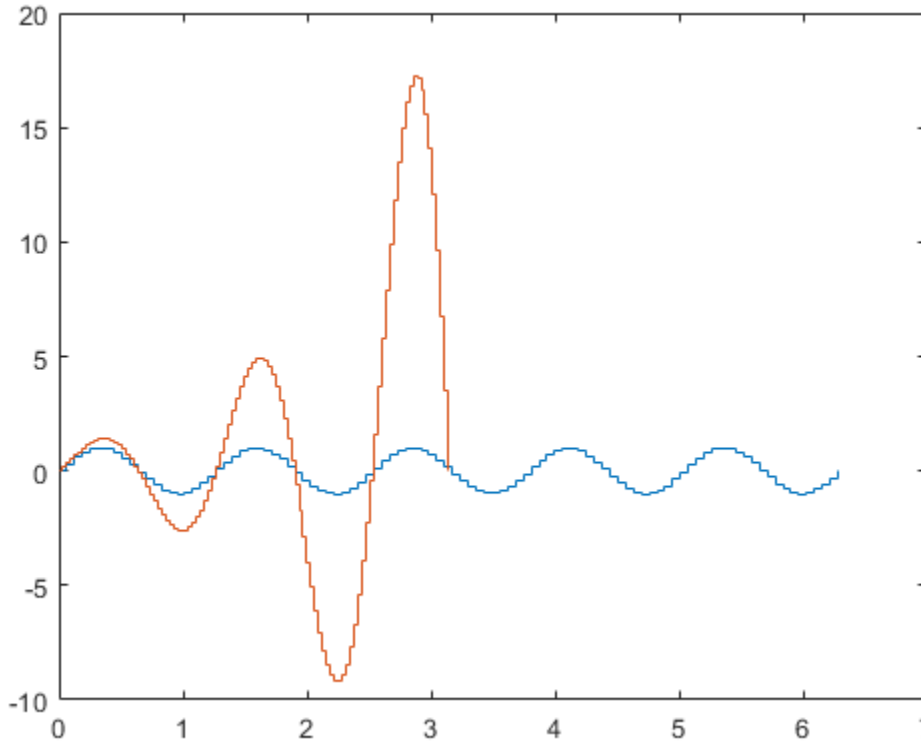
The first vector input, X, determines the *x*-axis positions for both data series.

Plot Multiple Data Series at Unique Sets of x-Values

Create a stairstep plot of two sine waves evaluated at different values. Specify a unique set of *x*-values for plotting each data series.

```
x1 = linspace(0,2*pi)';
x2 = linspace(0,pi)';
X = [x1,x2];
Y = [sin(5*x1),exp(x2).*sin(5*x2)];
```

```
figure
stairs(X,Y)
```

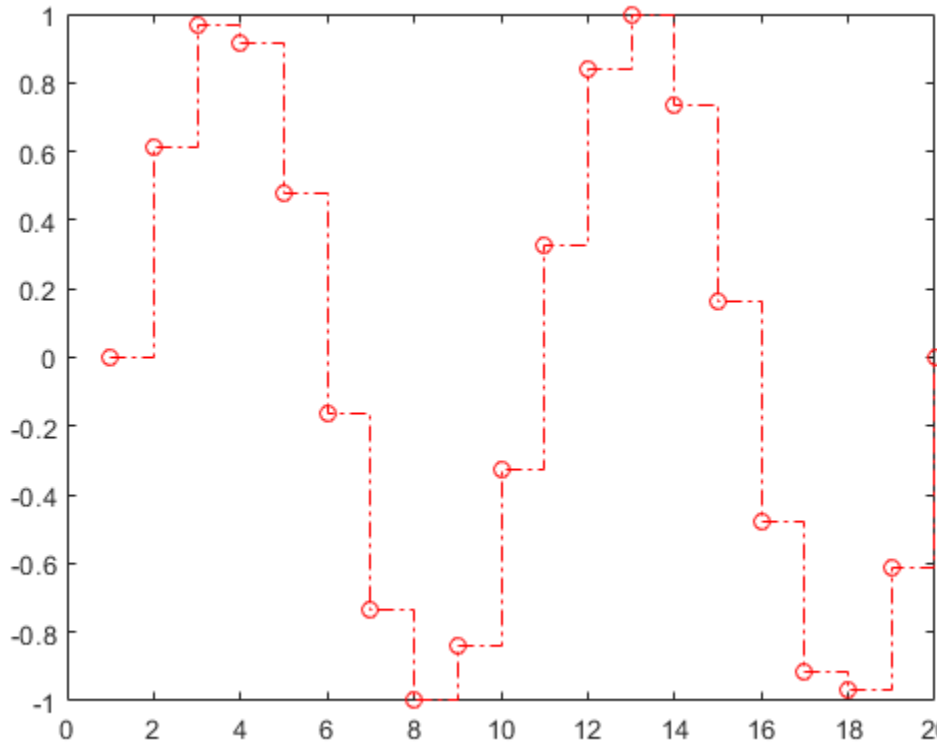


Each column of X is plotted against the corresponding column of Y.

Specify Line Style, Marker Symbol and Color

Create a stairstep plot and set the line style to a dot-dashed line, the marker symbol to circles, and the color to red.

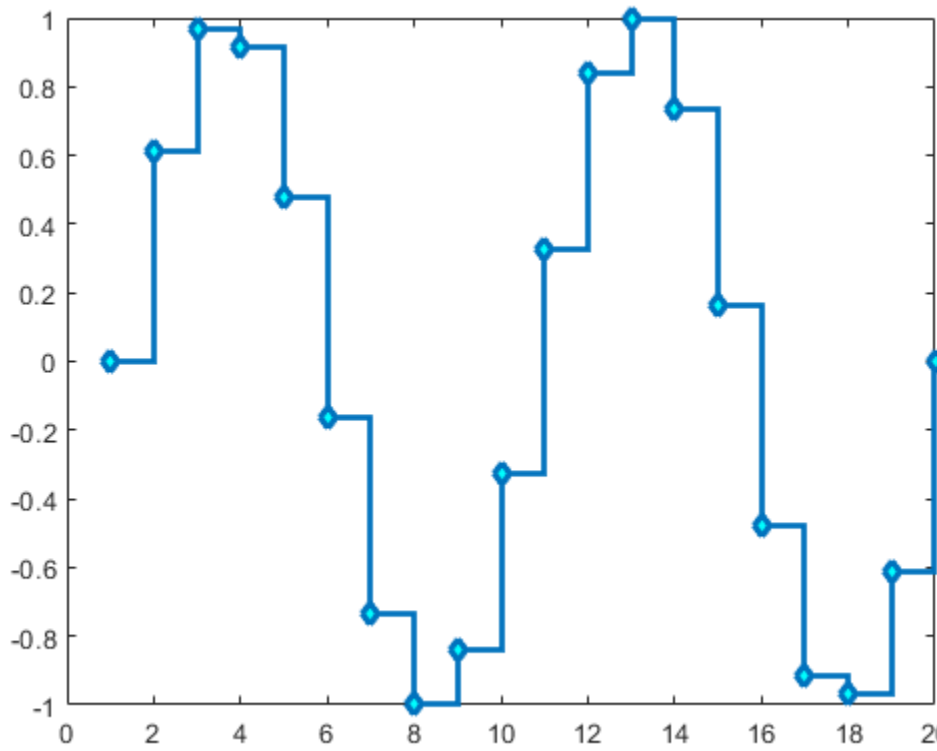
```
X = linspace(0,4*pi,20);
Y = sin(X);

figure
stairs(Y, '-.or')
```



Specify Additional Style Options

Create a stairstep plot and set the line width to 2, the marker symbols to diamonds, and the marker face color to cyan using `Name,Value` pair arguments.

```
X = linspace(0,4*pi,20);
Y = sin(X);

figure
stairs(Y,'LineWidth',2,'Marker','d','MarkerFaceColor','c')
```

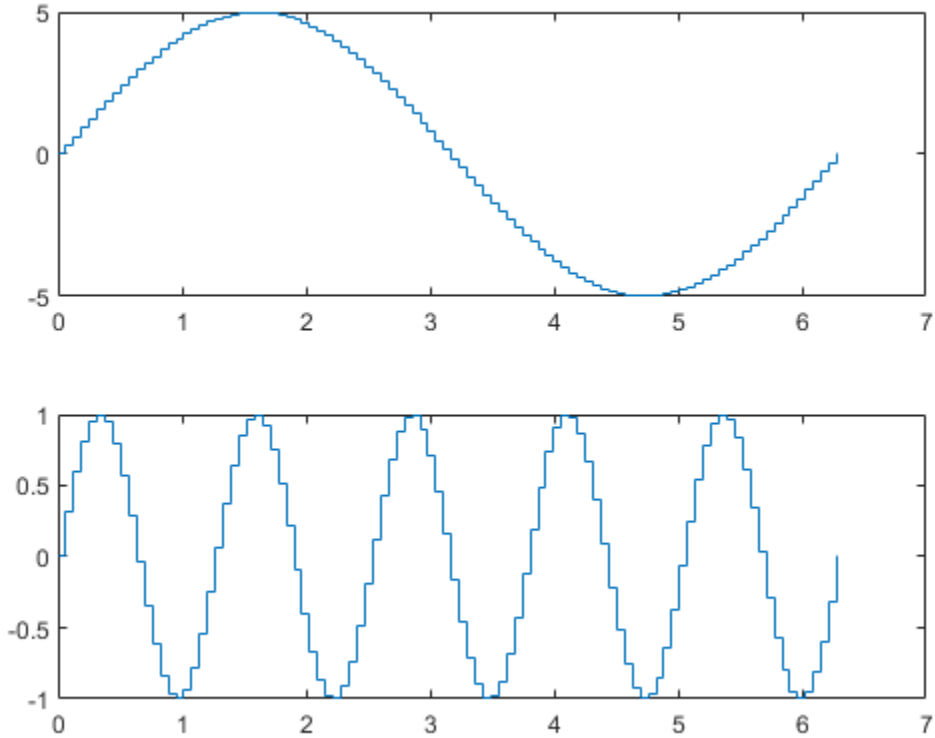


Specify Axes for Stairstep Plots

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout.

Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Create separate stairstep plots in the axes by specifying the axes object as the first argument to `stairs`.

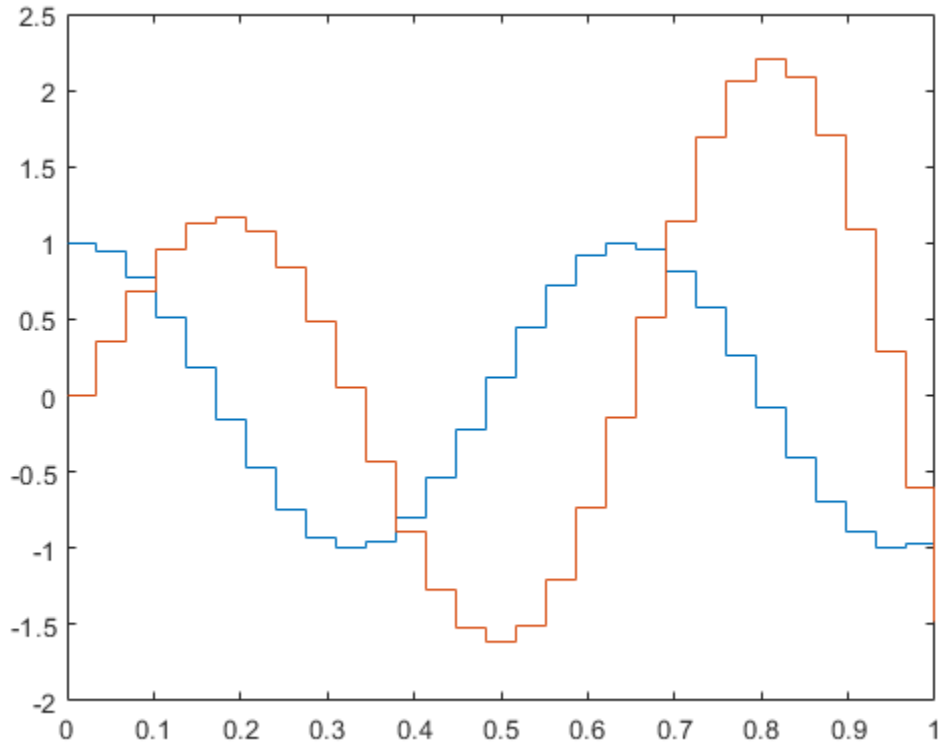
```
x = linspace(0,2*pi);
y1 = 5*sin(x);
y2 = sin(5*x);
tiledlayout(2,1)

% Top plot
ax1 = nexttile;
stairs(ax1,x,y1)

% Bottom plot
ax2 = nexttile;
stairs(ax2,x,y2)
```



Modify Stairstep Plot After Creation

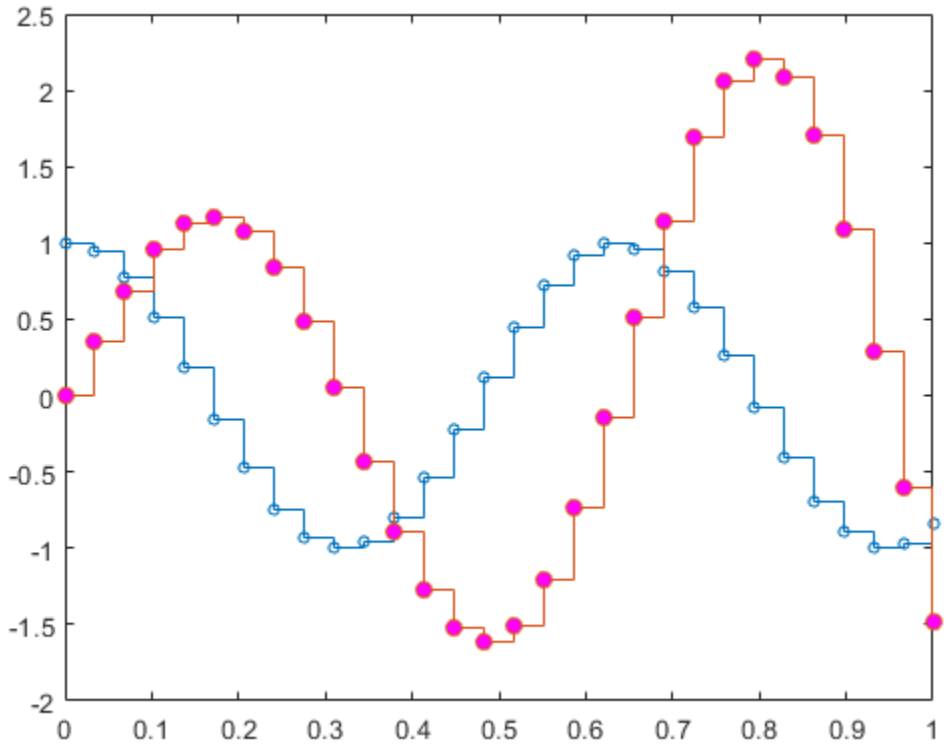
Create a stairstep plot of two data series and return the two stair objects.

```
X = linspace(0,1,30)';
Y = [cos(10*X), exp(X).*sin(10*X)];
h = stairs(X,Y);
```



Use small circle markers for the first data series. Use magenta filled circles for the second series. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead.

```
h(1).Marker = 'o';
h(1).MarkerSize = 4;
h(2).Marker = 'o';
h(2).MarkerFaceColor = 'm';
```



Create a Stairstep Plot using plot Function

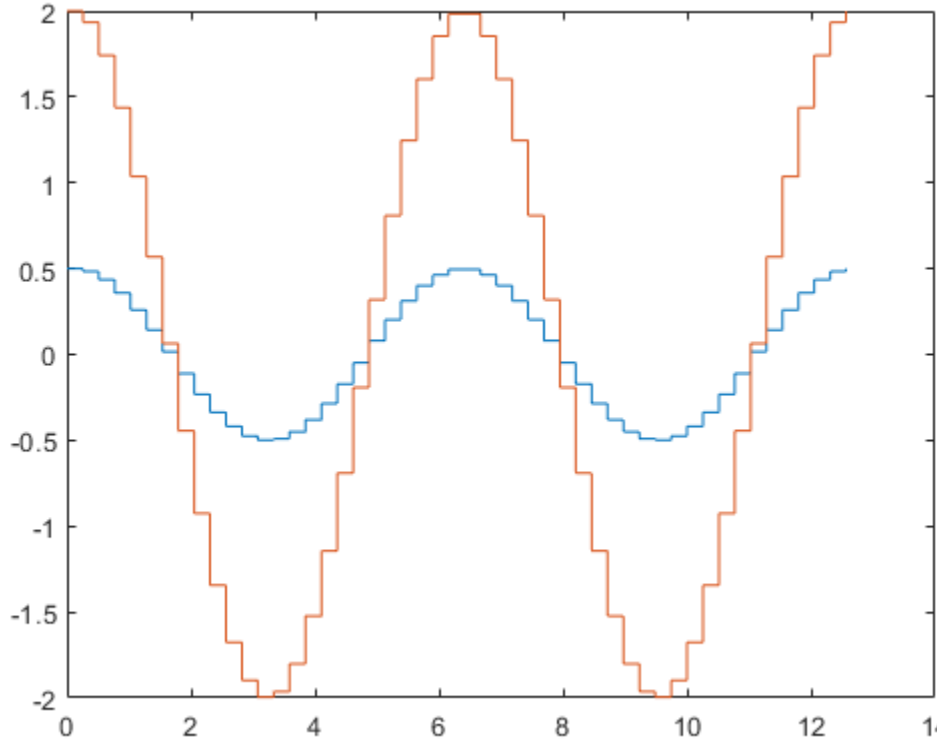
Evaluate two cosine functions at 50 equally spaced values between 0 and 4π and create a stairstep plot using `plot`.

```
X = linspace(0,4*pi,50)';
Y = [0.5*cos(X), 2*cos(X)];
[xb,yb] = stairs(X,Y);
```

`stairs` returns two matrices of the same size, `xb` and `yb`, but no plot.

Use `plot` to create the stairstep plot with `xb` and `yb`.

```
figure
plot(xb,yb)
```



Input Arguments

Y — y values
vector or matrix

y values, specified as a vector or matrix. When Y is a vector, `stairs` creates one stair object. When Y is a matrix, `stairs` draws one line per matrix column and creates a separate stair object for each column.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

X — x values
vector or matrix

x values, specified as a vector or matrix. When `Y` is a vector, `X` must be a vector of the same size. When `Y` is a matrix, `X` must be a matrix of the same size, or a vector whose length equals the number of rows in `Y`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`LineSpec` — Line style, marker symbol, and color
character vector | string

Line style, marker symbol, and color, specified as a character vector or string. For more information on line style, marker symbol, and color options see `LineSpec`.

Example: `':*r'`

ax — `Axes` object
`Axes` object

`Axes` object. If you do not specify the axes, then `stairs` plots into the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Marker','s','MarkerFaceColor','red'` plots the stairstep graph with red square markers.

The properties listed here are only a subset. For a complete list, see Stair.

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

`Color` — Line color
`[0 0.4470 0.7410]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: 'blue'

Example: [0 0 1]

Example: '#0000FF'

Marker — Marker symbol

'none' (default) | 'o' | '+' | '*' | '.' | ...

Marker symbol, specified as one of the values listed in this table. By default, the object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerSize` — Marker size
6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

`MarkerEdgeColor` — Marker outline color
`'auto'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'auto'` uses the same color as the `Color` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

MarkerFaceColor — Marker fill color

'none' (default) | 'auto' | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...

Marker fill color, specified as 'auto', an RGB triplet, a hexadecimal color code, a color name, or a short name. The 'auto' option uses the same color as the Color property of the parent axes. If you specify 'auto' and the axes plot box is invisible, the marker fill color is the color of the figure.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Output Arguments

h — `Stair` objects
`Stair` objects

`Stair` objects. These are unique identifiers, which you can use to query and modify the properties of a specific `Stair` object after it is created.

xb — x values for use with `plot`
vector or matrix

x values for use with `plot`, returned as a vector or matrix. `xb` contains the appropriate values such that `plot(xb,yb)` creates the stairstep graph.

yb — y values for use with `plot`
vector or matrix

y values for use with `plot`, returned as a vector or matrix. `yb` contains the appropriate values such that `plot(xb,yb)` creates the stairstep graph.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions

`LineSpec` | `bar` | `histogram` | `stem`

Properties

Stair

Topics

“Plot Dates and Durations”
“Plot Categorical Data”

Introduced before R2006a

Stair Properties

Stair chart appearance and behavior

Description

`Stair` properties control the appearance and behavior of a `Stair` object. By changing property values, you can modify certain aspects of the stair chart.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = stairs(1:10);
c = h.Color;
h.Color = 'red';
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Color and Styling

`Color` — Line color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`ColorMode` — Control how `Color` is set
`'auto'` (default) | `'manual'`

Control how the `Color` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the `Color` property by selecting a color from the `ColorOrder` property of the axes.
- `'manual'` — You control the value of the `Color` property manually, either by setting the value of the `Color` property directly on the object, or by including a color in the `LineSpec` argument when you call a plotting function.

If you change the value of the `Color` property manually, MATLAB changes the value of the `ColorMode` property to `'manual'`.

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineStyleMode` — Control how `LineStyle` is set
`'auto'` (default) | `'manual'`

Control how the `LineStyle` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the `LineStyle` property by selecting a line style from the `LineStyleOrder` property of the axes.
- `'manual'` — You control the value of the `LineStyle` property manually, either by setting the value of the `LineStyle` property directly on the object, or by specifying the `LineSpec` argument when you call a plotting function.

If you change the value of the `LineStyle` property manually, MATLAB changes the value of the `LineStyleMode` property to `'manual'`.

`LineWidth` — Line width

`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

Markers

`Marker` — Marker symbol

`'none'` (default) | `'o'` | `'+'` | `'*'` | `'.'` | ...

Marker symbol, specified as one of the values listed in this table. By default, the object does not display markers. Specifying a marker symbol adds markers at each data point or vertex.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

`MarkerMode` — Control how `Marker` is set

`'auto'` (default) | `'manual'`

Control how the `Marker` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the object's `Marker` property.
- `'manual'` — You control the value of the `Marker` property manually, either by setting the value of the `Marker` property directly on the object, or by including a marker in the `LineSpec` argument when you call a plotting function.

If you change the value of the `Marker` property manually, MATLAB changes the value of the `MarkerMode` property to `'manual'`.

`MarkerSize` — Marker size
6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

`MarkerEdgeColor` — Marker outline color
`'auto'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'auto'` uses the same color as the `Color` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` option uses the same color as the `Color` property of the parent axes. If you specify `'auto'` and the axes plot box is invisible, the marker fill color is the color of the figure.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to

F. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Data

`XData` — x values

`[ ]` (default) | vector

x values, specified as a vector. The input argument `X` to the `stairs` function sets the *x* values. If you do not specify the *x* values, then `stairs` uses the indices of `YData`. `XData` and `YData` must have equal lengths.

Example: `1:10`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`XDataMode` — Selection mode for `XData`
`'auto'` (default) | `'manual'`

Selection mode for `XData`, specified as one of these values:

- `'auto'` — Use the indices of the values in `YData` (or `ZData` for 3-D plots).
- `'manual'` — Use manually specified values. To specify the values, set the `XData` property or specify the input argument `X` to the plotting function.

`XDataSource` — Variable linked to `XData`
`''` (default) | character vector | string

Variable linked to `XData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `XData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `XData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'x'`

`YData` — y values
`[]` (default) | vector

y values, specified as a vector. The input argument `Y` to the `stairs` function sets the *y* values. `XData` and `YData` must have equal lengths.

Example: `1:10`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`YDataSource` — Variable linked to `YData`
`''` (default) | character vector | string

Variable linked to `YData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `YData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `YData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'y'`

Legend

`DisplayName` — Legend label
`''` (default) | character vector | string scalar

Legend label, specified as a character vector or string scalar. The legend does not display until you call the `legend` command. If you do not specify the text, then `legend` sets the label using the form `'dataN'`.

`Annotation` — Control for including or excluding object from legend
`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`DataTipTemplate` — Data tip content

`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

`Clipping` — Clipping of object to axes limits

`'on'` (default) | `'off'`

Clipping of the object to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the object that are outside the axes limits.
- `'off'` — Display the entire object, even if parts of it appear outside the axes limits. Parts of the object might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the object so that it is larger than the original plot.

The `Clipping` property of the axes that contains the object must be set to `'on'`. Otherwise, this property has no effect. For more information about the clipping behavior, see the `Clipping` property of the axes.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see "Callback Definition".

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see "Callback Definition".

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you

do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks when visible. The `Visible` property must be set to `'on'` and you must click a part of the `Stair` object that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The `HitTest` property determines if the `Stair` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `Stair` object passes the click to the object below it in the current view of the figure window. The `HitTest` property of the `Stair` object has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `Stair` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `Stair` object that has one of these:

- `HitTest` property set to `'on'`
- `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

Note The `PickableParts` property determines if the `Stair` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent

`Axes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `Group`, or `Transform` object.

`Children` — Children

empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the chart object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'stair'`

This property is read-only.

Type of graphics object, returned as `'stair'`. Use this property to find all objects of a given type within a plotting hierarchy, such as searching for the type using `findobj`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`stairs`

Topics

“Access Property Values”
“Graphics Object Properties”

Introduced before R2006a

standardizeMissing

Insert standard missing values

Syntax

```
B = standardizeMissing(A,indicator)
B = standardizeMissing(A,indicator,'DataVariables',vars)
```

Description

`B = standardizeMissing(A,indicator)` replaces values specified in `indicator` with standard missing values in an array or table.

Standard missing values depend on the data type:

- `NaN` for `double`, `single`, `duration`, and `calendarDuration`
- `NaT` for `datetime`
- `<missing>` for `string`
- `<undefined>` for `categorical`
- `' '` for `char`
- `{''}` for `cell` of character arrays

`B = standardizeMissing(A,indicator,'DataVariables',vars)` standardizes missing values in the variables specified by `vars` when `A` is a table or timetable.

Examples

Nonstandard Missing Numbers

Create a row vector and replace all instances of `-99` with the standard missing value for `double` data types, `NaN`.

```
A = [0 1 5 -99 8 3 4 -99 16];
B = standardizeMissing(A,-99)
```

```
B = 1×9

     0     1     5   NaN     8     3     4   NaN    16
```

Replace All Instances of Specified Values

Create a table containing `Inf` and `'N/A'` to represent missing values.

```
dblVar = [NaN;3;Inf;7;9];
cellstrVar = {'one';'three';'';'N/A';'nine'};
charVar = ['A';'C';'E';' ';'I'];
categoryVar = categorical({'red';'yellow';'blue';'violet';''});

A = table(dblVar,cellstrVar,charVar,categoryVar)
```

```
A=5×4 table
    dblVar    cellstrVar    charVar    categoryVar
    ______    __________    _______    ___________

     NaN      {'one'   }       A       red
       3      {'three' }       C       yellow
     Inf      {0x0 char}       E       blue
       7      {'N/A'   }               violet
       9      {'nine'  }       I       <undefined>
```

Replace all instances of `Inf` with `NaN` and replace all instances of `'N/A'` with the empty character vector, `''`.

```
B = standardizeMissing(A,{Inf,'N/A'})
```

```
B=5×4 table
    dblVar    cellstrVar    charVar    categoryVar
    ______    __________    _______    ___________

     NaN      {'one'   }       A       red
       3      {'three' }       C       yellow
     NaN      {0x0 char}       E       blue
       7      {0x0 char}               violet
```

```
    9      {'nine'  }       I       <undefined>
```

Replace Only Values in Specified Variables

Replace instances of `Inf` and `'N/A'` occurring in specified variables of a table with the standard missing value indicators.

Create a table containing `Inf` and `'N/A'` to represent missing values.

```
a = {'alpha';'bravo';'charlie';'';'N/A'};
x = [1;NaN;3;Inf;5];
y = [57;732;93;1398;Inf];

A = table(a,x,y)
```

```
A=5×3 table
         a           x      y
    ___________    ___    ____

    {'alpha'  }      1      57
    {'bravo'  }    NaN     732
    {'charlie'}      3      93
    {0x0 char }    Inf    1398
    {'N/A'    }      5     Inf
```

For the variables `a` and `x`, replace instances of `Inf` with `NaN` and `'N/A'` with the empty character vector, `''`.

```
B = standardizeMissing(A,{Inf,'N/A'},'DataVariables',{'a','x'})
```

```
B=5×3 table
         a           x      y
    ___________    ___    ____

    {'alpha'  }      1      57
    {'bravo'  }    NaN     732
    {'charlie'}      3      93
    {0x0 char }    NaN    1398
    {0x0 char }      5     Inf
```

`Inf` in the variable `y` remains unchanged because `y` is not included in the `'DataVariables'` name-value pair argument.

Input Arguments

`A` — Input data
vector | matrix | multidimensional array | table | timetable

Input data, specified as a vector, matrix, multidimensional array, table, or timetable. If `A` is a timetable, then `ismissing` operates on the table data only and ignores `NaT` and `NaN` values in the vector of row times.

Data Types: `double` | `single` | `char` | `string` | `cell` | `table` | `timetable` | `categorical` | `datetime` | `duration`

`indicator` — Nonstandard missing value indicator
scalar | vector | cell array

Nonstandard missing-value indicator, specified as a scalar, vector, or cell array. The elements of `indicator` define the values that `standardizeMissing` treats as missing. If `A` is an array, then `indicator` must be a vector. If `A` is a table or timetable, then `indicator` can also be a cell array with entries of multiple data types.

The data types specified in `indicator` match data types in the corresponding entries of `A`. The following are additional data type matches between the elements of `indicator` and elements of `A`:

- `double` indicators match `double`, `single`, integer, and `logical` entries of `A`.
- `string` and `char` indicators match `categorical` entries of `A`.

Example: `B = standardizeMissing(A,'N/A')` replaces the character vector `'N/A'` with the empty character vector, `''`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `cell` | `datetime` | `duration`

`vars` — Table variables to standardize
variable name | cell array of variable names | numeric vector | logical vector | function handle

Table variables to standardize, specified as a variable name, a cell array of variable names, a numeric vector, a logical vector, or a function handle. `vars` can be one of the following:

- A character vector specifying a single table variable name
- A cell array of character vectors where each element is a table variable name
- A vector of table variable indices
- A logical vector whose elements each correspond to a table variable, where `true` includes the corresponding variable and `false` excludes it
- A function handle that returns a logical scalar, such as `@isnumeric`

Example: `'Age'`

Example: `{'Height','Weight'}`

Example: `@iscategorical`

Data Types: `char` | `cell` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `function_handle`

Output Arguments

B — Standardized array or table
vector | matrix | multidimensional array | table | timetable

Standardized array or table, specified as a vector, matrix, multidimensional array, table, or timetable. `B` has the same size as `A`.

Data Types: `double` | `single` | `char` | `string` | `cell` | `table` | `timetable` | `categorical` | `datetime` | `duration` | `calendarDuration`

Algorithms

`standardizeMissing` treats leading and trailing white space differently for cell arrays of character vectors, character arrays, and categorical arrays.

- For cell arrays of character vectors, `standardizeMissing` does not ignore white space. All character vectors must match exactly a character vector specified in `indicator`.

- For character arrays, `standardizeMissing` ignores trailing white space.
- For categorical arrays, `standardizeMissing` ignores leading and trailing white space.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Tables are not supported.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`fillmissing` | `ismissing` | `rmmissing`

Topics

“Missing Data in MATLAB”

Introduced in R2013b

startsWith

Determine if strings start with pattern

Syntax

```
TF = startsWith(str,pattern)
TF = startsWith(str,pattern,'IgnoreCase',true)
```

Description

`TF = startsWith(str,pattern)` returns `1` (`true`) if `str` starts with the specified pattern, and returns `0` (`false`) otherwise.

If `pattern` is an array containing multiple patterns, then `startsWith` returns `1` if it finds that `str` starts with any element of `pattern`.

`TF = startsWith(str,pattern,'IgnoreCase',true)` ignores case when determining if `str` starts with `pattern`.

Examples

Determine If String Starts with Pattern

Create a string array that contains file names. Determine which file names start with the word `data`.

Starting in R2017a, you can create strings using double quotes.

```
str = ["abstract.docx","data.tar","code.m"; ...
       "data-analysis.ppt","results.ptx","summary.ppt"]
```

```
str = 2x3 string array
    "abstract.docx"        "data.tar"       "code.m"
    "data-analysis.ppt"    "results.ptx"    "summary.ppt"
```

Return a logical array where the position of each element equal to `1` corresponds to the position of a string in `str` that starts with `data`.

```
pattern = "data";
TF = startsWith(str,pattern)
```

```
TF = 2x3 logical array

   0   1   0
   1   0   0
```

Display the file names that start with `data`. Index back into `str` using `TF`.

```
str(TF)
```

```
ans = 2x1 string array
    "data-analysis.ppt"
    "data.tar"
```

Test Start of String Against Multiple Patterns

Create a string array that contains file names. Determine which file names start with either `abstract` or `data`.

Starting in R2017a, you can create strings using double quotes.

```
str = ["abstract.docx","data.tar.gz","mycode.m","results.ptx"]
```

```
str = 1x4 string array
    "abstract.docx"    "data.tar.gz"    "mycode.m"    "results.ptx"
```

```
pattern = ["abstract","data"];
TF = startsWith(str,pattern)
```

```
TF = 1x4 logical array

   1   1   0   0
```

Display the strings that start with either `abstract` or `data`. Index back into `str` using `TF`.

```
str(TF)
```

```
ans = 1x2 string array
    "abstract.docx"    "data.tar.gz"
```

Ignore Case When Testing Start of String

Create a string array that contains file names. Determine which file names start with `data`, ignoring case.

Starting in R2017a, you can create strings using double quotes.

```
str = ["DATA.TAR.GZ","data.xlsx","SUMMARY.PPT","tmp.gz"]
```

```
str = 1x4 string array
    "DATA.TAR.GZ"    "data.xlsx"    "SUMMARY.PPT"    "tmp.gz"
```

```
pattern = "data";
TF = startsWith(str,pattern,'IgnoreCase',true)
```

```
TF = 1x4 logical array

   1   1   0   0
```

Display the strings that start with `data`. Index back into `str` using `TF`.

```
str(TF)
```

```
ans = 1x2 string array
    "DATA.TAR.GZ"    "data.xlsx"
```

Determine if Character Vector Starts with Pattern

Create a character vector that contains the name of a file. Determine if the name starts with different patterns.

```
chr = 'data-analysis.ppt'
```

```
chr =
'data-analysis.ppt'
```

```
TF = startsWith(chr,'data')
```

```
TF = logical
   1
```

```
TF = startsWith(chr,'test')
```

```
TF = logical
   0
```

Input Arguments

`str` — Input text
string array | character vector | cell array of character vectors

Input text, specified as a string array, a character vector, or a cell array of character vectors.

`pattern` — Search pattern
string array | character vector | cell array of character vectors

Search pattern, specified as a string array, a character vector, or a cell array of character vectors.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `str` and `pattern` must be a string scalar, a character vector, or a cell array containing not more than one character vector.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`contains` | `endsWith` | `find` | `regexp` | `strcmp` | `strfind`

Topics

“Create String Arrays”
“Search and Replace Text”
“Compare Text”
“Test for Empty Strings and Missing Values”

Introduced in R2016b

startup

User-defined startup script for MATLAB

Syntax

```
startup
```

Description

`startup` executes user-specified commands when starting.

Create a `startup.m` file in the `userpath` folder, which is on the MATLAB search path. Add commands you want executed at startup. For example, your `startup` function might include physical constants, defaults for graphics properties, engineering conversion factors, or anything else you want predefined in your workspace.

Tips

- To specify the current folder in MATLAB when it starts, set the **Initial working folder** preference, described in “General Preferences”.

Algorithms

At startup, MATLAB automatically executes the `matlabrc` function and, if it exists on the MATLAB search path, `startup`. The `matlabrc` file, which is in the *`matlabroot`*`/toolbox/local` folder, is reserved for use by MathWorks and by system administrators on multiuser systems.

See Also

`finish` | `matlabrc` | `matlabroot` | `path` | `quit` | `userpath`

Topics

"Startup Options in MATLAB Startup File"
"Preferences"

Introduced before R2006a

stats

Return cached values and statistics for `MemoizedFunction` object

Syntax

```
s = stats(mfcn)
```

Description

`s = stats(mfcn)` returns cached values and statistics on the use of a `MemoizedFunction` object.

Examples

Obtain Statistics for Memoized Function

Create a `MemoizedFunction` object by memozing the `plus` function.

```
mf = memoize(@plus)
```

```
mf 
  MemoizedFunction with properties:

     Function: @plus
      Enabled: 1
    CacheSize: 10
```

Call the memoized function several times. The first time you call the function with a particular set of inputs, MATLAB caches the results.

```
a = mf(13,42);  % calls plus; caches results
b = mf(7,33);   % calls plus; caches results
c = mf(13,42);  % does not call plus; returns cached results
d = mf(5,120);  % calls plus; caches results
```

```
e = mf(13,42);  % does not call plus; returns cached results
f = mf(7,33);   % does not call plus; returns cached results
```

Call the `stats` function.

```
s = stats(mf)
```

```
s = struct with fields:
                    Cache: [1x1 struct]
       MostHitCachedInput: [1x1 struct]
      CacheHitRatePercent: 50
    CacheOccupancyPercent: 30
```

Determine which sets of inputs you used the most.

```
s.MostHitCachedInput
```

```
ans = struct with fields:
     Hits: 2
    Input: {[13]  [42]}
```

Obtain the cached results.

```
c = s.Cache
```

```
c = struct with fields:
         Inputs: {{1x2 cell}  {1x2 cell}  {1x2 cell}}
        Nargout: [1 1 1]
        Outputs: {{1x1 cell}  {1x1 cell}  {1x1 cell}}
       HitCount: [2 1 0]
      TotalHits: 3
    TotalMisses: 3
```

Display all sets of cached inputs and outputs alongside how many times MATLAB accessed the cached values.

```
T = cell2table([c.Inputs' c.Outputs'],'VariableNames',{'Inputs','Outputs'});
T.Times_Cache_Accessed = c.HitCount'
```

```
T=3×3 table
         Inputs          Outputs    Times_Cache_Accessed
    ________________     _______    ____________________
```

```
    {[13]}    {[ 42]}    {[ 55]}             2
    {[ 7]}    {[ 33]}    {[ 40]}             1
    {[ 5]}    {[120]}    {[125]}             0
```

Input Arguments

`mfcn` — Function with memoization semantics
`MemoizedFunction` object

Function with memoization semantics, specified as a `MemoizedFunction` object.

Output Arguments

`s` — `MemoizedFunction` statistics
structure

`MemoizedFunction` statistics, returned as a structure containing these fields.

Field	Description
`Cache`	Cached results, returned as a structure. If `m` is the number of function calls with cached results, then the structure contains the following fields: • `Inputs` – Cached input values, returned as a `1`-by-`m` cell array. Each element in the `Inputs` cell array is a `1`-by-`n` cell array, where `n` is the number of inputs for a particular function call. • `Outputs` – Cached output values, returned as a `1`-by-`m` cell array. Each element in the `Outputs` cell array is a `1`-by-`p` cell array, where `p` is the number of outputs for a particular function call. • `HitCount` – Number of times each cached set of input values was accessed, returned as `1`-by-`m` `double` array. • `TotalHits` – Total number of times a set of input values was found in the cache, returned as a `double`. • `TotalMisses` – Total number of times a set of input values was not found in the cache, returned as a `double`.

Field	Description
`MostHitCachedInput`	Inputs for the most accessed cached values, returned as a structure. The structure contains the following fields: • `Hits` – Number of times the cached values were accessed, returned as an integer • `Input` – Input values for the most accessed cached values, returned as a cell array of input values
`CacheHitRatePercent`	Percentage of times an input was found in the cache, returned as a double. This property is computed by `(TotalHits/(TotalHits+TotalMisses))*100`.
`CacheOccupancyPercent`	How full the cache is, returned as a double representing a percentage. This property is computed by `(Number of cache entries/cache size)*100`.

See Also

Functions
`memoize`

Objects
`MemoizedFunction`

Introduced in R2017a

std

Standard deviation

Syntax

```
S = std(A)
S = std(A,w)
S = std(A,w,'all')
S = std(A,w,dim)
S = std(A,w,vecdim)
S = std( ___ ,nanflag)
```

Description

`S = std(A)` returns the standard deviation on page 1-14563 of the elements of `A` along the first array dimension whose size does not equal 1.

- If `A` is a vector of observations, then the standard deviation is a scalar.
- If `A` is a matrix whose columns are random variables and whose rows are observations, then `S` is a row vector containing the standard deviations corresponding to each column.
- If `A` is a multidimensional array, then `std(A)` operates along the first array dimension whose size does not equal 1, treating the elements as vectors. The size of this dimension becomes `1` while the sizes of all other dimensions remain the same.
- By default, the standard deviation is normalized by `N-1`, where `N` is the number of observations.

`S = std(A,w)` specifies a weighting scheme for any of the previous syntaxes. When `w = 0` (default), `S` is normalized by `N-1`. When `w = 1`, `S` is normalized by the number of observations, `N`. `w` also can be a weight vector containing nonnegative elements. In this case, the length of `w` must equal the length of the dimension over which `std` is operating.

`S = std(A,w,'all')` computes the standard deviation over all elements of `A` when `w` is either 0 or 1. This syntax is valid for MATLAB versions R2018b and later.

`S = std(A,w,dim)` returns the standard deviation along dimension `dim` for any of the previous syntaxes. To maintain the default normalization while specifying the dimension of operation, set `w = 0` in the second argument.

`S = std(A,w,vecdim)` computes the standard deviation over the dimensions specified in the vector `vecdim` when `w` is 0 or 1. For example, if `A` is a matrix, then `std(A,0,[1 2])` computes the standard deviation over all elements in `A`, since every element of a matrix is contained in the array slice defined by dimensions 1 and 2.

`S = std( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. For example, `std(A,'includenan')` includes all `NaN` values in `A` while `std(A,'omitnan')` ignores them.

Examples

Standard Deviation of Matrix Columns

Create a matrix and compute the standard deviation of each column.

```
A = [4 -5 1; 2 3 5; -9 1 7];
S = std(A)
```

```
S = 1×3

    7.0000    4.1633    3.0551
```

Standard Deviation of 3-D Array

Create a 3-D array and compute the standard deviation along the first dimension.

```
A(:,:,1) = [2 4; -2 1];
A(:,:,2) = [9 13; -5 7];
A(:,:,3) = [4 4; 8 -3];
S = std(A)
```

```
S =
S(:,:,1) =
```

```
    2.8284    2.1213


S(:,:,2) =

    9.8995    4.2426


S(:,:,3) =

    2.8284    4.9497
```

Specify Standard Deviation Weights

Create a matrix and compute the standard deviation of each column according to a weight vector `w`.

```
A = [1 5; 3 7; -9 2];
w = [1 1 0.5];
S = std(A,w)
```

```
S = 1×2

    4.4900    1.8330
```

Standard Deviation Along Matrix Rows

Create a matrix and calculate the standard deviation along each row.

```
A = [6 4 23 -3; 9 -10 4 11; 2 8 -5 1];
S = std(A,0,2)
```

```
S = 3×1

   11.0303
    9.4692
```

```
    5.3229
```

Standard Deviation of Array Page

Create a 3-D array and compute the standard deviation over each page of data (rows and columns).

```
A(:,:,1) = [2 4; -2 1];
A(:,:,2) = [9 13; -5 7];
A(:,:,3) = [4 4; 8 -3];
S = std(A,0,[1 2])
```

```
S =
S(:,:,1) =

    2.5000


S(:,:,2) =

    7.7460


S(:,:,3) =

    4.5735
```

Standard Deviation Excluding NaN

Create a vector and compute its standard deviation, excluding NaN values.

```
A = [1.77 -0.005 3.98 -2.95 NaN 0.34 NaN 0.19];
S = std(A,'omitnan')
```

```
S = 2.2797
```

Input Arguments

A — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array. If A is a scalar, then std(A) returns 0. If A is a 0-by-0 empty array, then std(A) returns NaN.

Data Types: single | double | datetime | duration
Complex Number Support: Yes

w — Weight

0 (default) | 1 | vector

Weight, specified as one of these values:

- 0 — Normalize by N-1, where N is the number of observations. If there is only one observation, then the weight is 1.
- 1 — Normalize by N.
- Vector made up of nonnegative scalar weights corresponding to the dimension of A along which the standard deviation is calculated.

Data Types: single | double

dim — Dimension to operate along

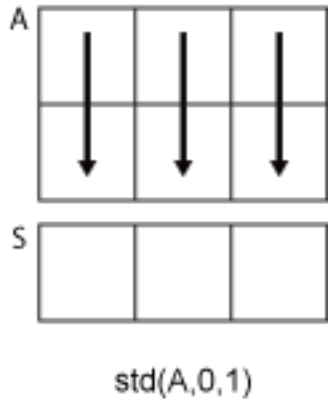
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

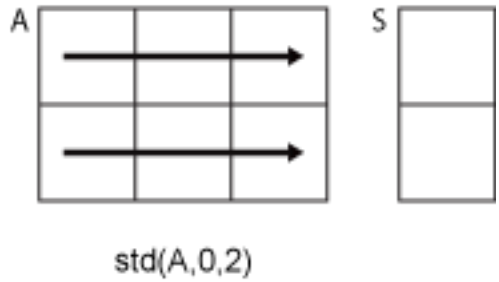
Dimension dim indicates the dimension whose length reduces to 1. The size(S,dim) is 1, while the sizes of all other dimensions remain the same.

Consider a two-dimensional input array, A.

- If dim = 1, then std(A,0,1) returns a row vector containing the standard deviation of the elements in each column.



- If `dim = 2`, then `std(A,0,2)` returns a column vector containing the standard deviation of the elements in each row.



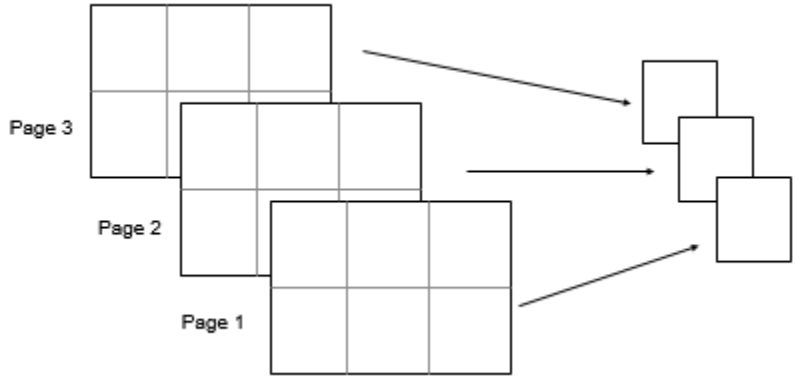
If `dim` is greater than `ndims(A)`, then `std(A)` returns an array of zeros the same size as `A`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`vecdim` — Vector of dimensions
vector of positive integers

Vector of dimensions, specified as a vector of positive integers. Each element represents a dimension of the input array. The lengths of the output in the specified operating dimensions are 1, while the others remain the same.

Consider a 2-by-3-by-3 input array, `A`. Then `std(A,0,[1 2])` returns a 1-by-1-by-3 array whose elements are the standard deviations computed over each page of `A`.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`nanflag` — `NaN` condition
`'includenan'` (default) | `'omitnan'`

`NaN` condition, specified as one of these values:

- `'includenan'` — Include `NaN` values when computing the standard deviation, resulting in `NaN`.
- `'omitnan'` — Ignore `NaN` values appearing in either the input array or weight vector.

For `datetime` arrays, you can also use `'omitnat'` or `'includenat'` to omit and include `NaT` values, respectively.

Data Types: `char`

More About

Standard Deviation

For a random variable vector A made up of N scalar observations, the standard deviation is defined as

$$S = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}|A_i - \mu|^2},$$

where μ is the mean of A:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} A_i .$$

The standard deviation is the square root of the variance. Some definitions of standard deviation use a normalization factor of N instead of $N-1$, which you can specify by setting `w` to `1`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitation:

- The weighting scheme cannot be a vector.

For more information, see “Tall Arrays for Out-of-Memory Data”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If you specify `dim`, then it must be a constant.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- If you specify `dim`, then it must be a constant.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`corrcoef` | `cov` | `mean` | `median` | `var`

Introduced before R2006a

stem

Plot discrete sequence data

Syntax

```
stem(Y)
stem(X,Y)
stem( ___ ,'filled')
stem( ___ ,LineSpec)
stem( ___ ,Name,Value)

stem(ax, ___ )

h = stem( ___ )
```

Description

`stem(Y)` plots the data sequence, `Y`, as stems that extend from a baseline along the *x*-axis. The data values are indicated by circles terminating each stem.

- If `Y` is a vector, then the *x*-axis scale ranges from 1 to `length(Y)`.
- If `Y` is a matrix, then `stem` plots all elements in a row against the same *x* value, and the *x*-axis scale ranges from 1 to the number of rows in `Y`.

`stem(X,Y)` plots the data sequence, `Y`, at values specified by `X`. The `X` and `Y` inputs must be vectors or matrices of the same size. Additionally, `X` can be a row or column vector and `Y` must be a matrix with `length(X)` rows.

- If `X` and `Y` are both vectors, then `stem` plots entries in `Y` against corresponding entries in `X`.
- If `X` is a vector and `Y` is a matrix, then `stem` plots each column of `Y` against the set of values specified by `X`, such that all elements in a row of `Y` are plotted against the same value.
- If `X` and `Y` are both matrices, then `stem` plots columns of `Y` against corresponding columns of `X`.

`stem( ___ ,'filled')` fills the circles. Use this option with any of the input argument combinations in the previous syntaxes.

`stem( ___ ,LineSpec)` specifies the line style, marker symbol, and color.

`stem( ___ ,Name,Value)` modifies the stem chart using one or more `Name,Value` pair arguments.

`stem(ax, ___ )` plots into the axes specified by `ax` instead of into the current axes (`gca`). The option, `ax`, can precede any of the input argument combinations in the previous syntaxes.

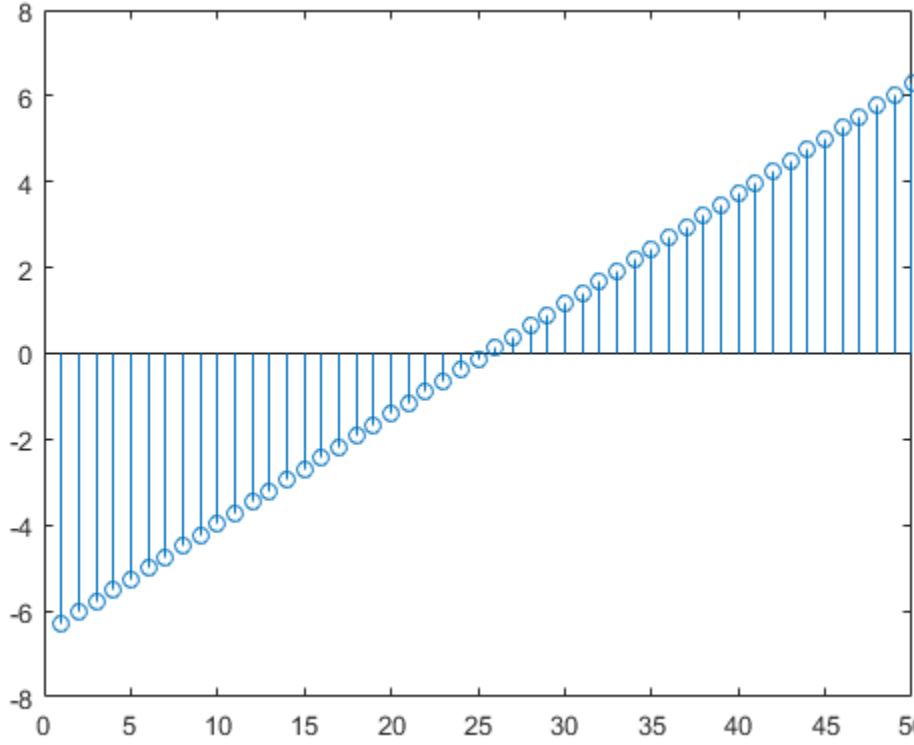
`h = stem( ___ )` returns a vector of `Stem` objects in `h`. Use `h` to modify the stem chart after it is created.

Examples

Plot Single Data Series

Create a stem plot of 50 data values between -2π and 2π.

```
figure
Y = linspace(-2*pi,2*pi,50);
stem(Y)
```

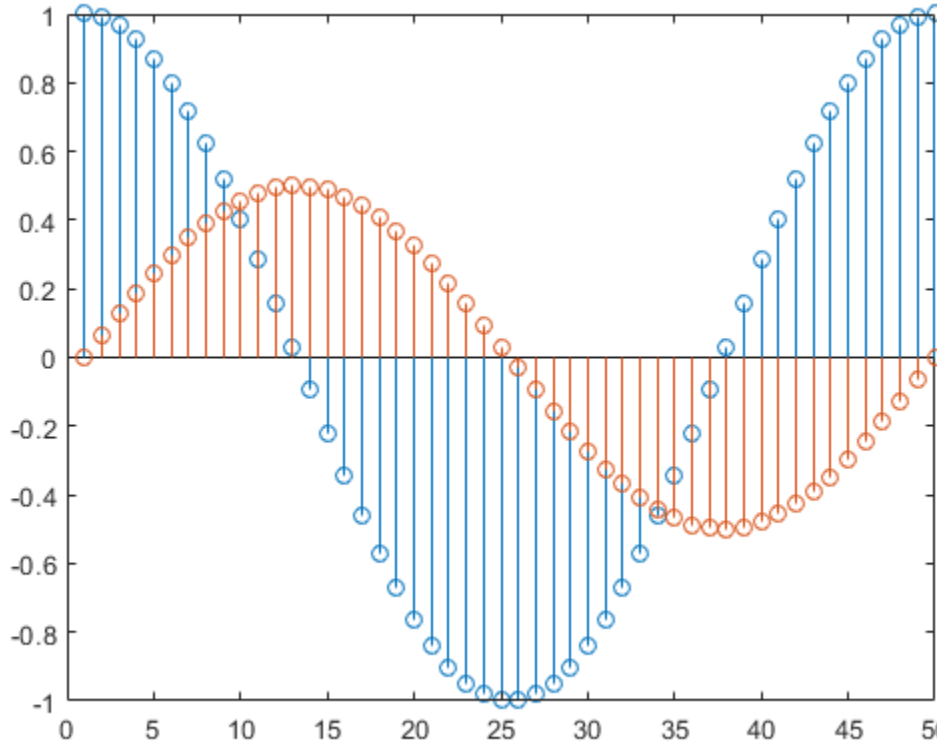


Data values are plotted as stems extending from the baseline and terminating at the data value. The length of Y automatically determines the position of each stem on the *x*-axis.

Plot Multiple Data Series

Plot two data series using a two-column matrix.

```
figure
X = linspace(0,2*pi,50)';
Y = [cos(X), 0.5*sin(X)];
stem(Y)
```



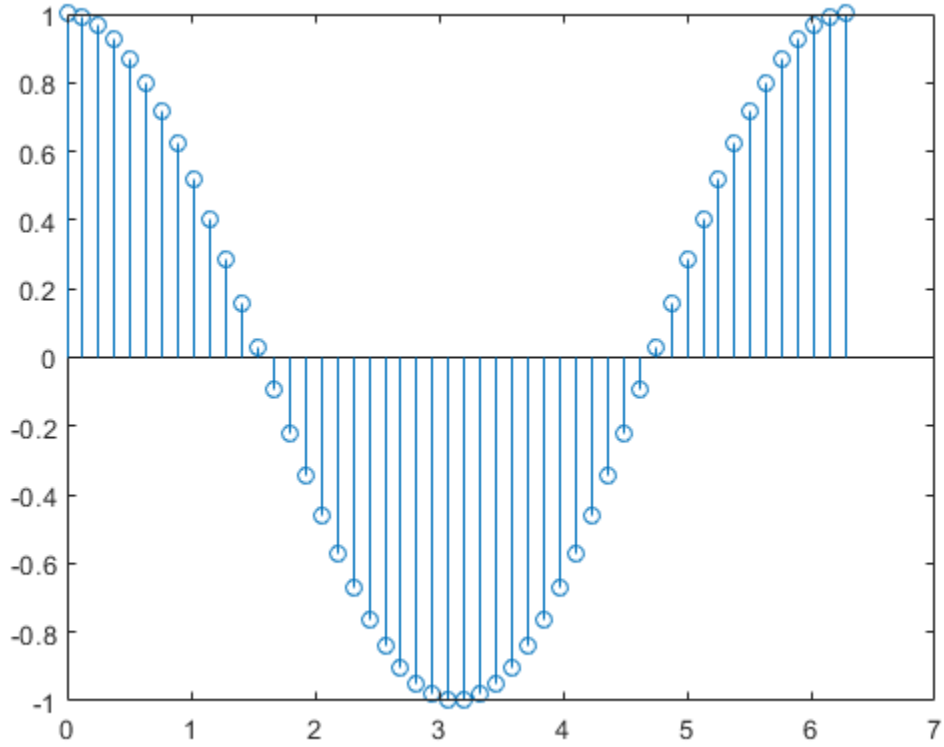
Each column of Y is plotted as a separate series, and entries in the same row of Y are plotted against the same *x* value. The number of rows in Y automatically generates the position of each stem on the *x*-axis.

Plot Single Data Series at Specified x values

Plot 50 data values of cosine evaluated between 0 and 2π and specify the set of *x* values for the stem plot.

```
figure
X = linspace(0,2*pi,50)';
```

```
Y = cos(X);
stem(X,Y)
```



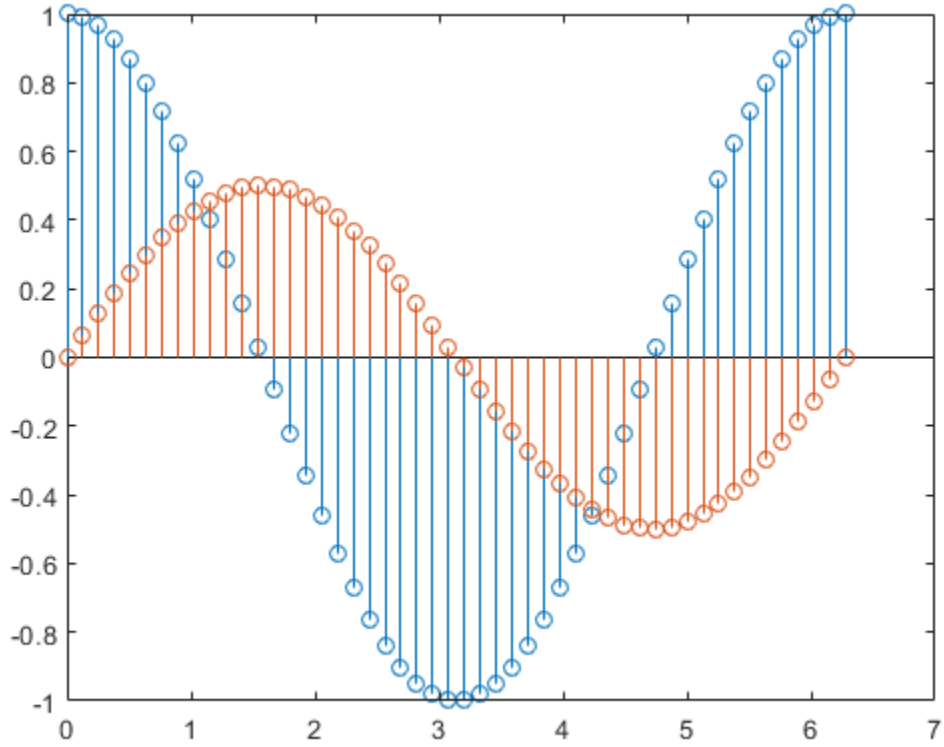
The first vector input determines the position of each stem on the *x*-axis.

Plot Multiple Data Series at Specified x values

Plot 50 data values of sine and cosine evaluated between 0 and 2π and specify the set of *x* values for the stem plot.

```
figure
X = linspace(0,2*pi,50)';
```

```
Y = [cos(X), 0.5*sin(X)];
stem(X,Y)
```



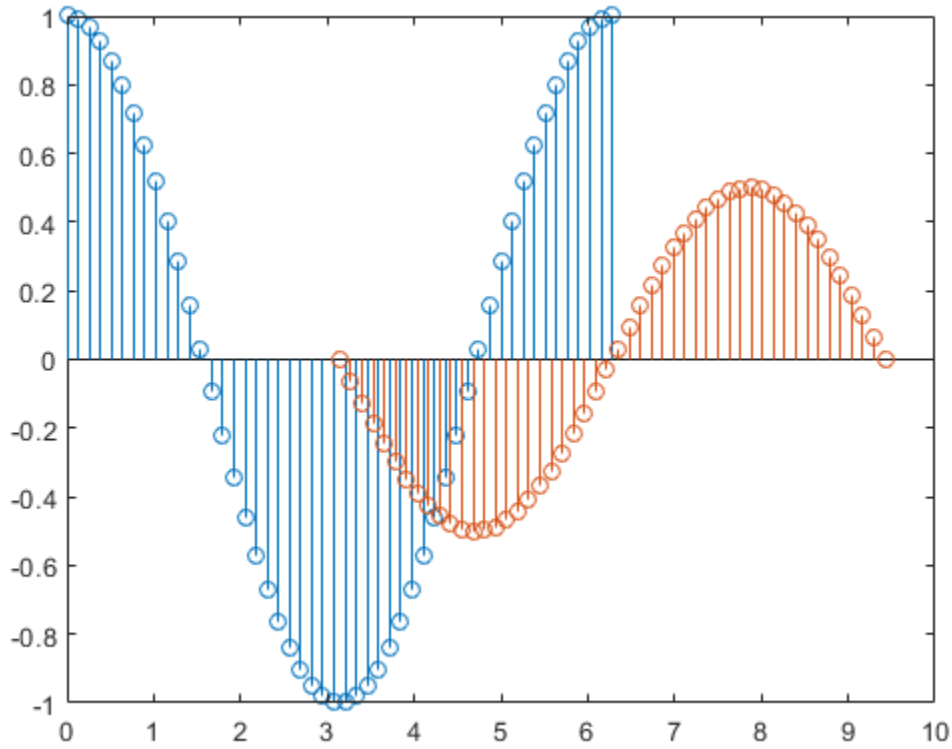
The vector input determines the *x*-axis positions for both data series.

Plot Multiple Data Series at Unique Sets of x values

Plot 50 data values of sine and cosine evaluated at different sets of *x* values. Specify the corresponding sets of *x* values for each series.

```
figure
x1 = linspace(0,2*pi,50)';
```

```
x2 = linspace(pi,3*pi,50)';
X = [x1, x2];
Y = [cos(x1), 0.5*sin(x2)];
stem(X,Y)
```

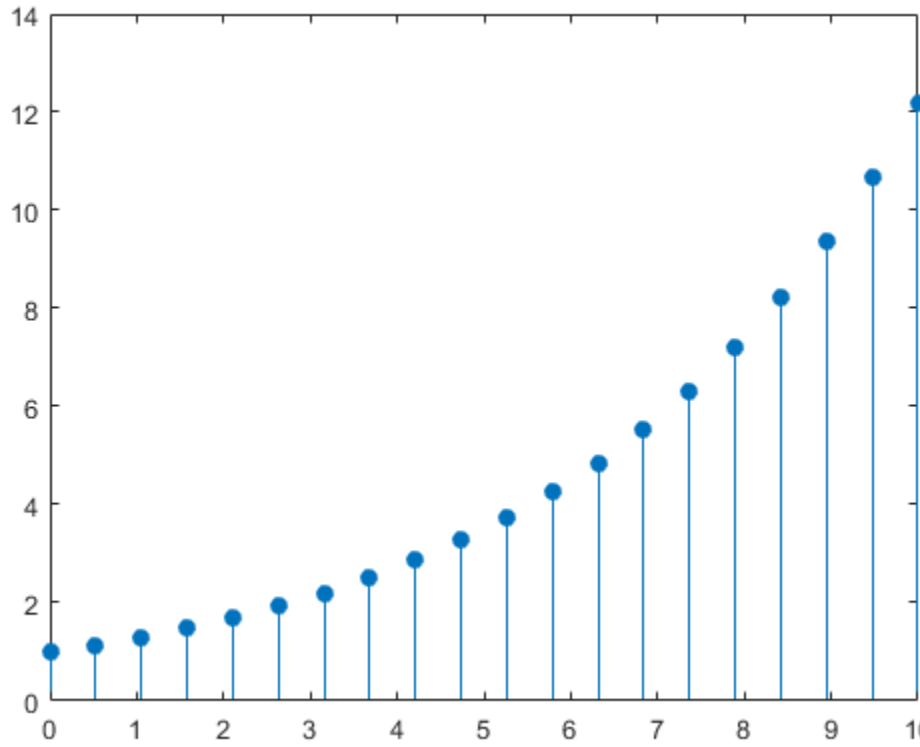


Each column of X is plotted against the corresponding column of Y.

Fill in Plot Markers

Create a stem plot and fill in the circles that terminate each stem.

```
X = linspace(0,10,20)';
Y = (exp(0.25*X));
stem(X,Y,'filled')
```

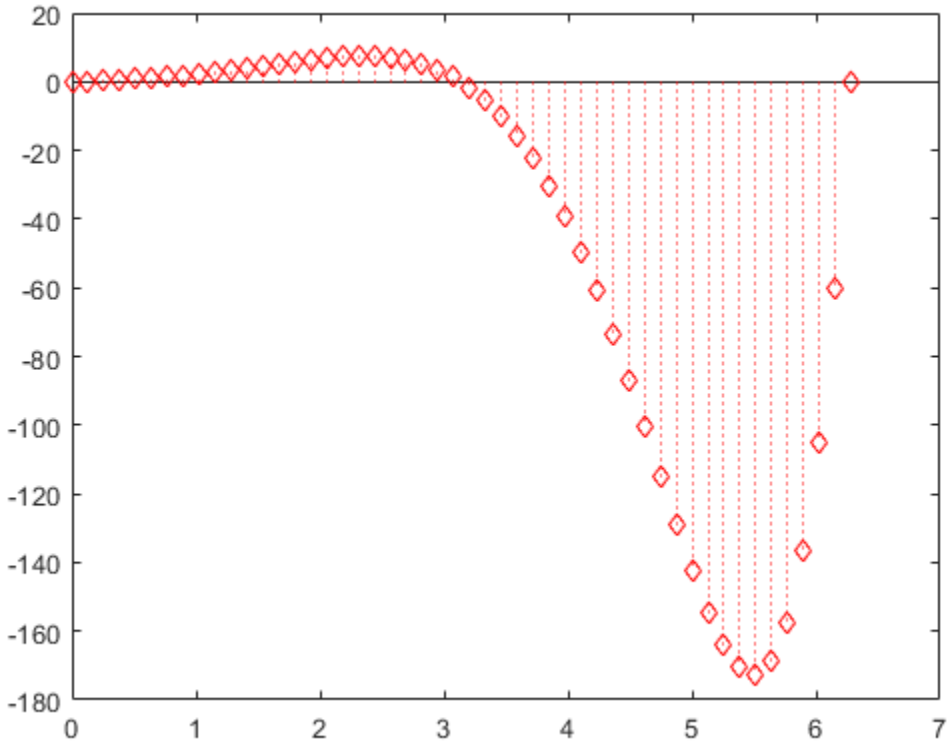


Specify Stem and Marker Options

Create a stem plot and set the line style to a dotted line, the marker symbols to diamonds, and the color to red using the `LineSpec` option.

```
figure
X = linspace(0,2*pi,50)';
```

```
Y = (exp(X).*sin(X));
stem(X,Y,':diamondr')
```



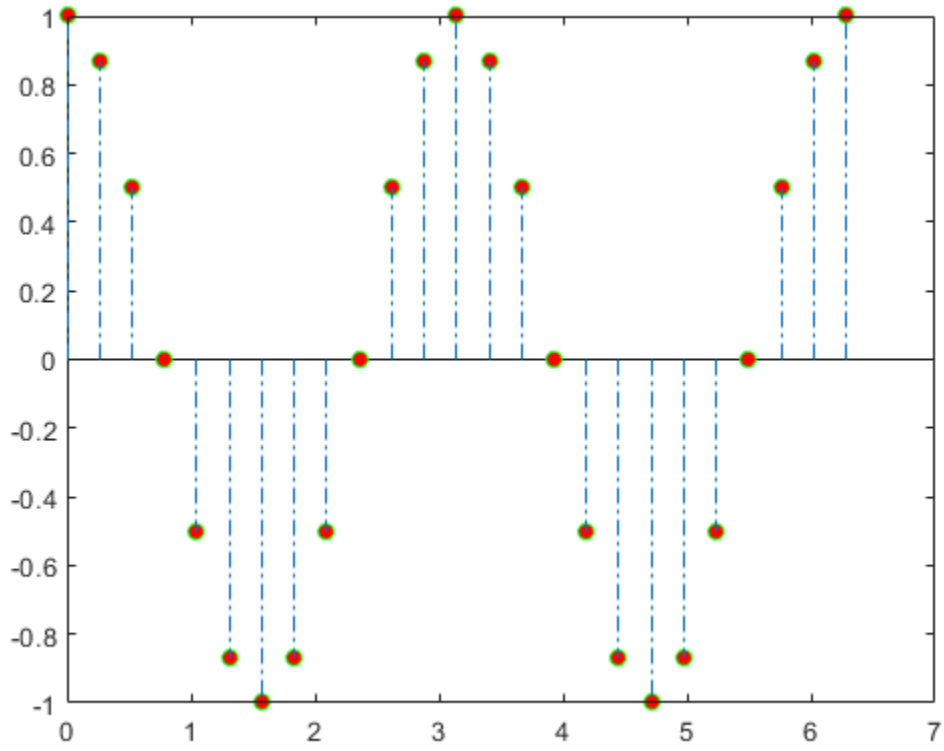
To color the inside of the diamonds, use the `'fill'` option.

Specify Additional Stem and Marker Options

Create a stem plot and set the line style to a dot-dashed line, the marker face color to red, and the marker edge color to green using `Name,Value` pair arguments.

```
figure
X = linspace(0,2*pi,25)';
```

```
Y = (cos(2*X));
stem(X,Y,'LineStyle','-.',...
     'MarkerFaceColor','red',...
     'MarkerEdgeColor','green')
```



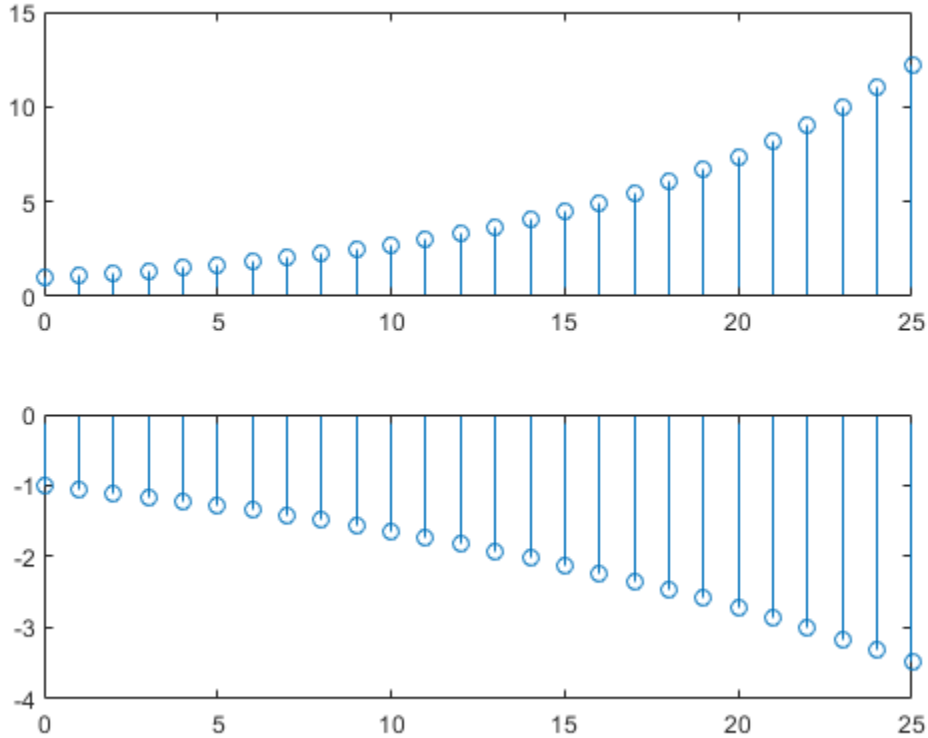
The stem remains the default color.

Specify Axes for Stem Plot

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout.

Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Create separate stem plots in the axes by specifying the axes object as the first argument to `stem`.

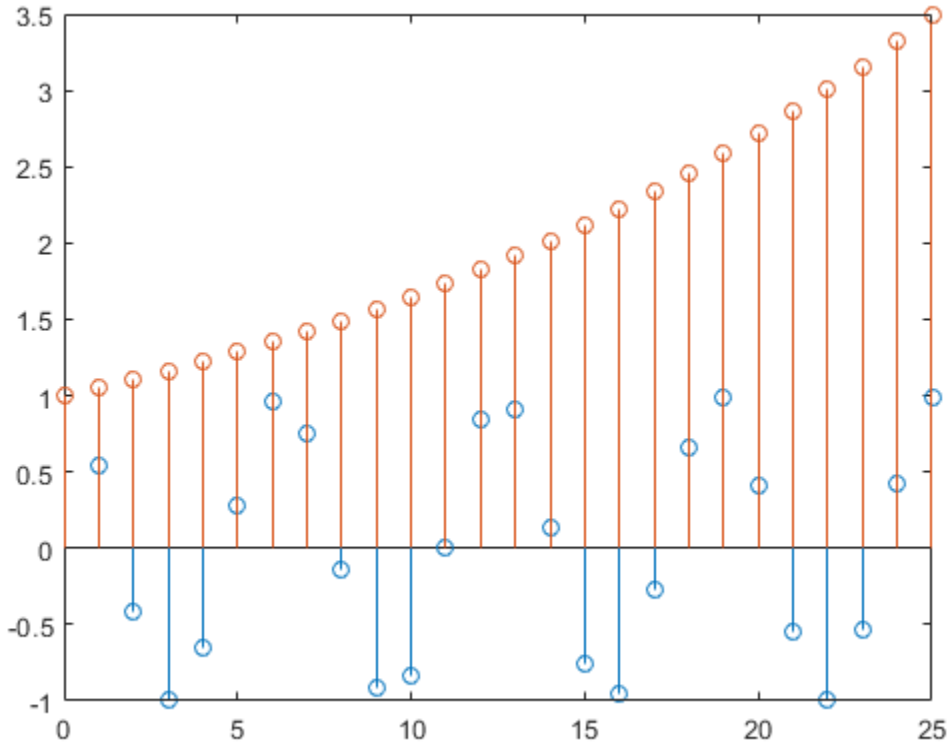
```
x = 0:25;
y1 = exp(0.1*x);
y2 = -exp(.05*x);
tiledlayout(2,1)

% Top plot
ax1 = nexttile;
stem(ax1,x,y1)

% Bottom plot
ax2 = nexttile;
stem(ax2,x,y2)
```



Modify Stem Series After Creation

Create a stem plot.

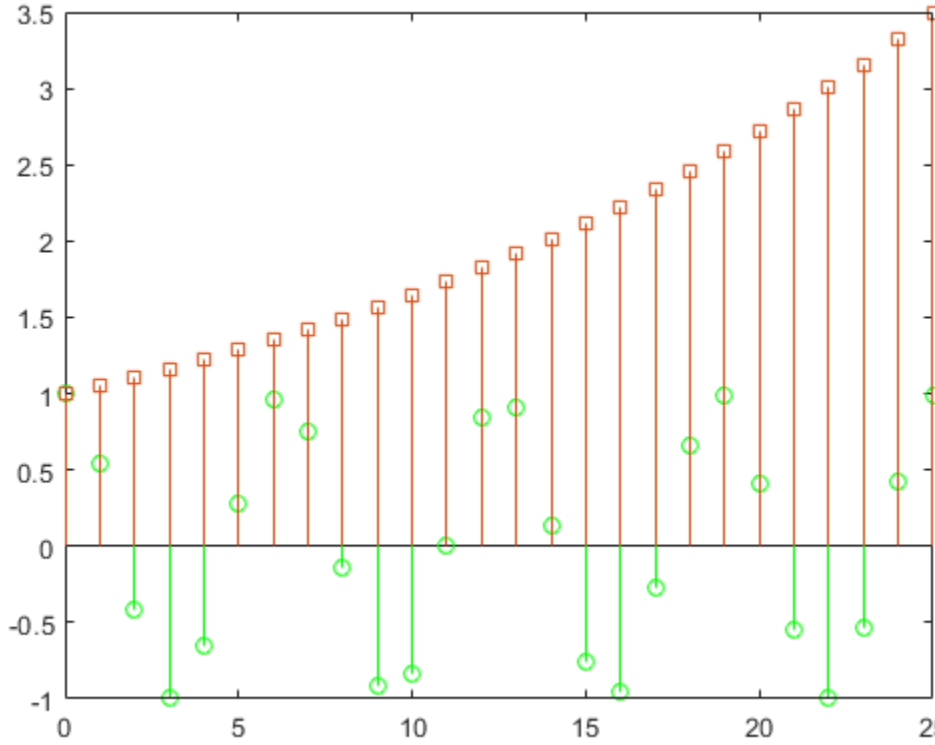
```
X = 0:25;
Y = [cos(X); exp(0.05*X)]';
h = stem(X,Y);
```



The `stem` function creates a stem series object for each column of data. The output argument, `h`, contains the two stem series objects.

Set the first stem series color to green. Change the markers of the second stem series to squares. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead.

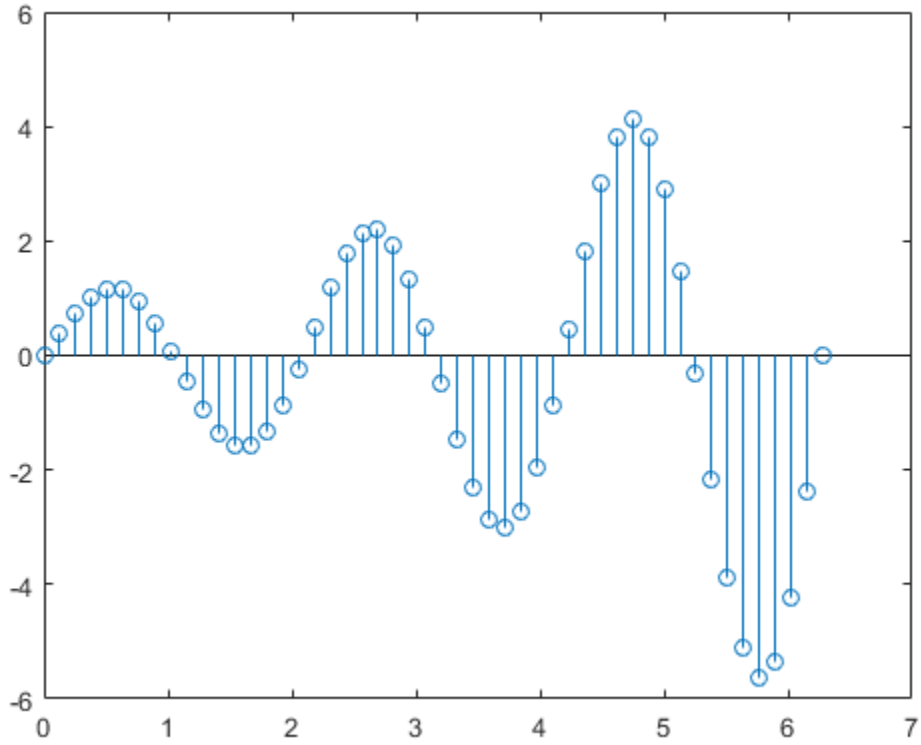
```
h(1).Color = 'green';
h(2).Marker = 'square';
```



Adjust Baseline Properties

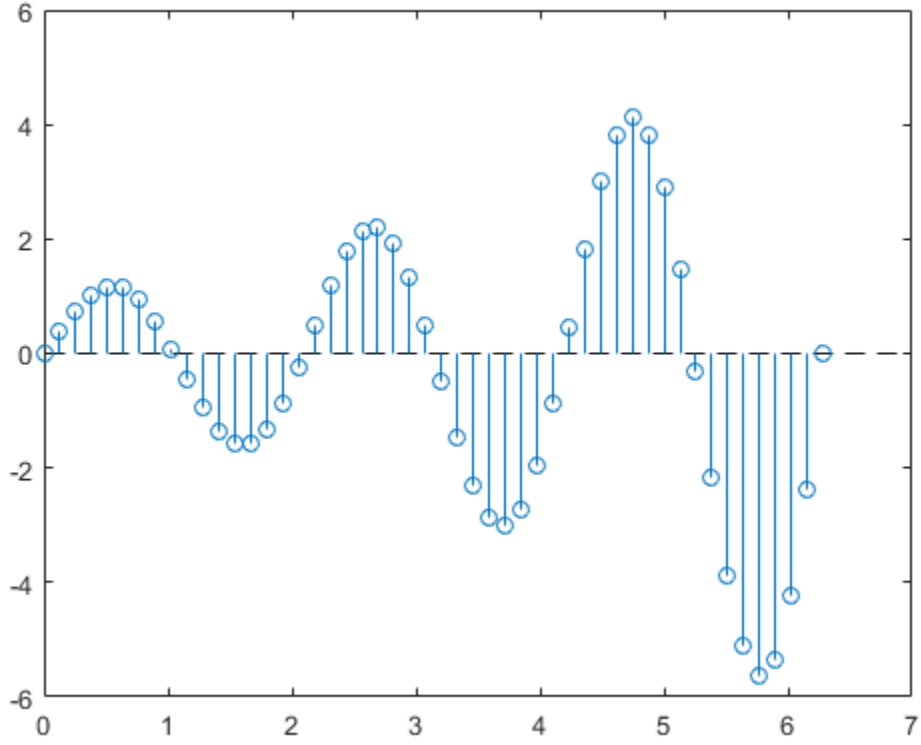
Create a stem plot and change properties of the baseline.

```
X = linspace(0,2*pi,50);
Y = exp(0.3*X).*sin(3*X);
h = stem(X,Y);
```



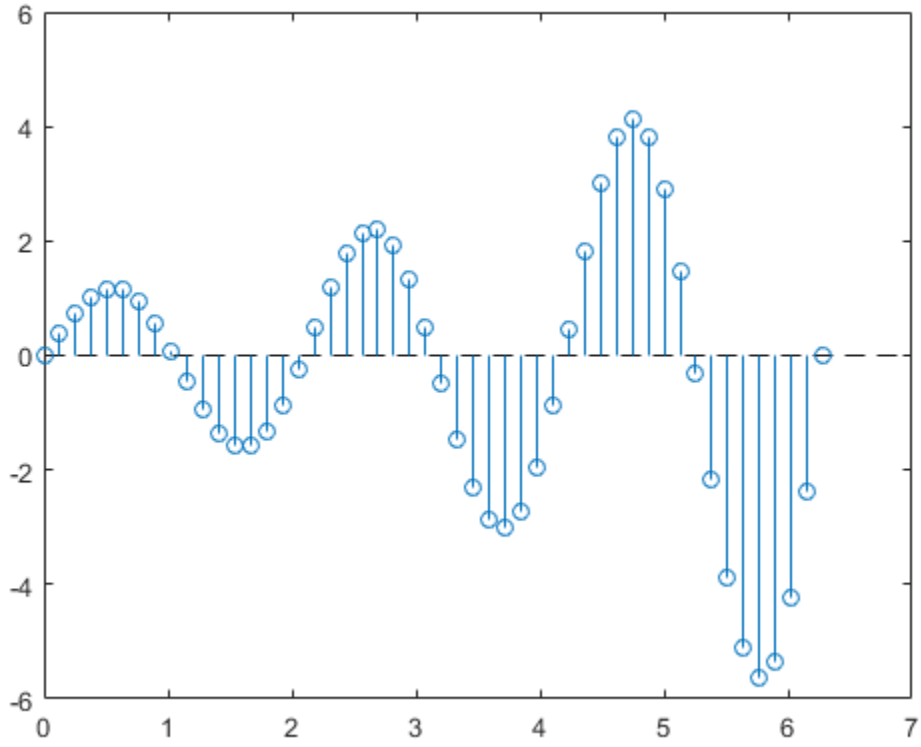
Change the line style of the baseline. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead.

```
hbase = h.BaseLine;
hbase.LineStyle = '--';
```



Hide the baseline by setting its `Visible` property to `'off'` .

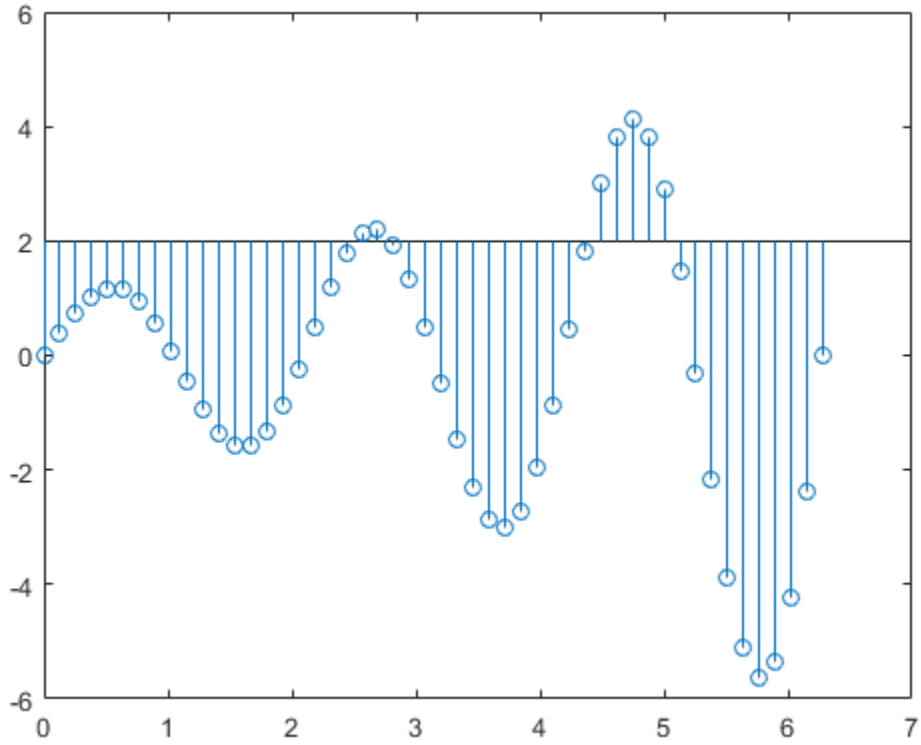
```
hbase.Visible = 'off';
```



Change Baseline Level

Create a stem plot with a baseline level at 2.

```
X = linspace(0,2*pi,50)';
Y = (exp(0.3*X).*sin(3*X));
stem(X,Y,'BaseValue',2);
```



Input Arguments

Y — Data sequence to display
vector or matrix

Data sequence to display, specified as a vector or matrix. When `Y` is a vector, `stem` creates one `Stem` object. When `Y` is a matrix, `stem` creates a separate `Stem` object for each column.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

X — Locations to plot data values in Y
vector or matrix

Locations to plot data values in Y, specified as a vector or matrix. When Y is a vector, X must be a vector of the same size. When Y is a matrix, X must be a matrix of the same size, or a vector whose length equals the number of rows in Y.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`LineSpec` — Line style, marker, and color
character vector | string

Line style, marker, and color, specified as a character vector or string. For more information, see `LineSpec`.

Example: `':*r'`

ax — `Axes` object
`Axes` object

`Axes` object. If you do not specify the axes, then `stem` plots into the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'LineStyle',':','MarkerFaceColor','red'` plots the stem as a dotted line and colors the marker face red.

The `Stem` properties listed here are only a subset. For a complete list, see Stem.

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

`Color` — Stem color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Stem color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`Marker` — Marker symbol

`'o'` (default) | `'+'` | `'*'` | `'.'` | `'x'` | ...

Marker symbol, specified as one of the markers listed in this table.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'+'`

Example: `'diamond'`

`MarkerSize` — Marker size

6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

`MarkerEdgeColor` — Marker outline color

`'auto'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'auto'` uses the same color as the `Color` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` option uses the same color as the `Color` property of the parent axes. If you specify `'auto'` and the axes plot box is invisible, the marker fill color is the color of the figure.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Output Arguments

h — Stem objects
Stem objects

Stem objects. These are unique identifiers, which you can use to modify the properties of a specific Stem object after it is created.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions
`LineSpec` | `bar` | `plot` | `stairs`

Properties
Stem

Topics

“Combine Line and Stem Plots”
“Plot Dates and Durations”
“Plot Categorical Data”

Introduced before R2006a

stem3

Plot 3-D discrete sequence data

Syntax

```
stem3(Z)
stem3(X,Y,Z)
stem3( ___ ,'filled')
stem3( ___ ,LineSpec)
stem3( ___ ,Name,Value)

stem3(ax, ___ )

h = stem3( ___ )
```

Description

`stem3(Z)` plots entries in `Z` as stems extending from the *xy*-plane and terminating with circles at the entry values. The stem locations in the *xy*-plane are automatically generated.

`stem3(X,Y,Z)` plots entries in `Z` as stems extending from the *xy*-plane where `X` and `Y` specify the stem locations in the *xy*-plane. The inputs `X`, `Y`, and `Z` must be vectors or matrices of the same size.

`stem3( ___ ,'filled')` fills the circles. Use this option with any of the input argument combinations in the previous syntaxes.

`stem3( ___ ,LineSpec)` specifies the line style, marker symbol, and color.

`stem3( ___ ,Name,Value)` modifies the stem chart using one or more name-value pair arguments.

`stem3(ax, ___ )` plots into the axes specified by `ax` instead of into the current axes (`gca`). The option, `ax`, can precede any of the input argument combinations in the previous syntaxes.

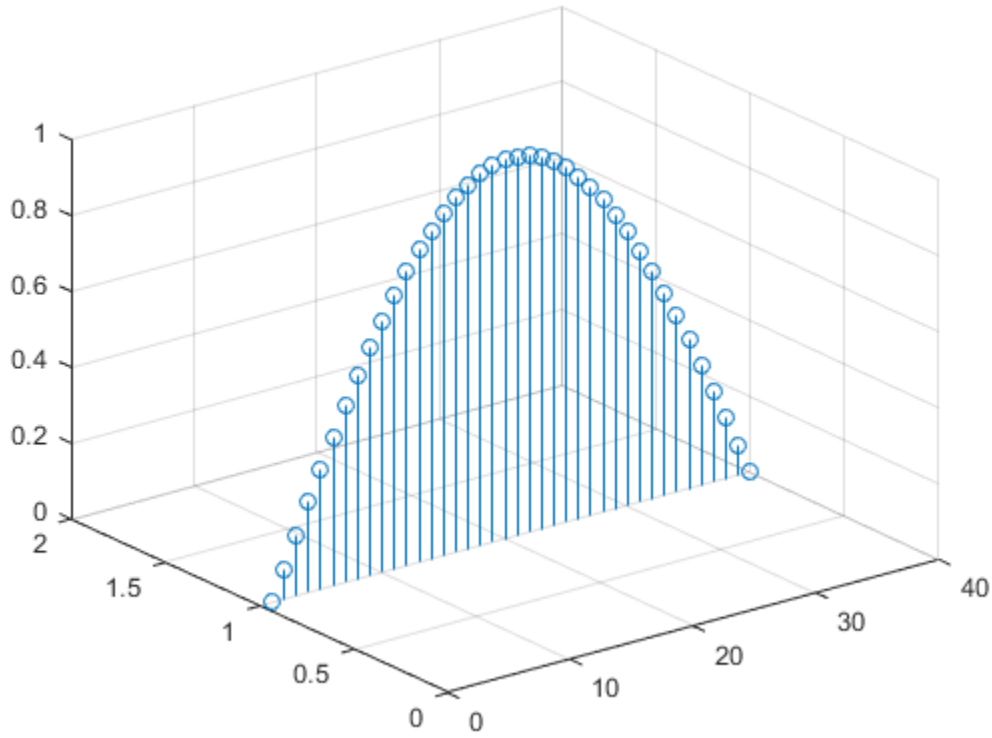
`h = stem3( ___ )` returns the `Stem` object `h`.

Examples

Row Vector Input

Create a 3-D stem plot of cosine values between $-\pi/2$ and $\pi/2$ with a row vector input.

```
figure
X = linspace(-pi/2,pi/2,40);
Z = cos(X);
stem3(Z)
```

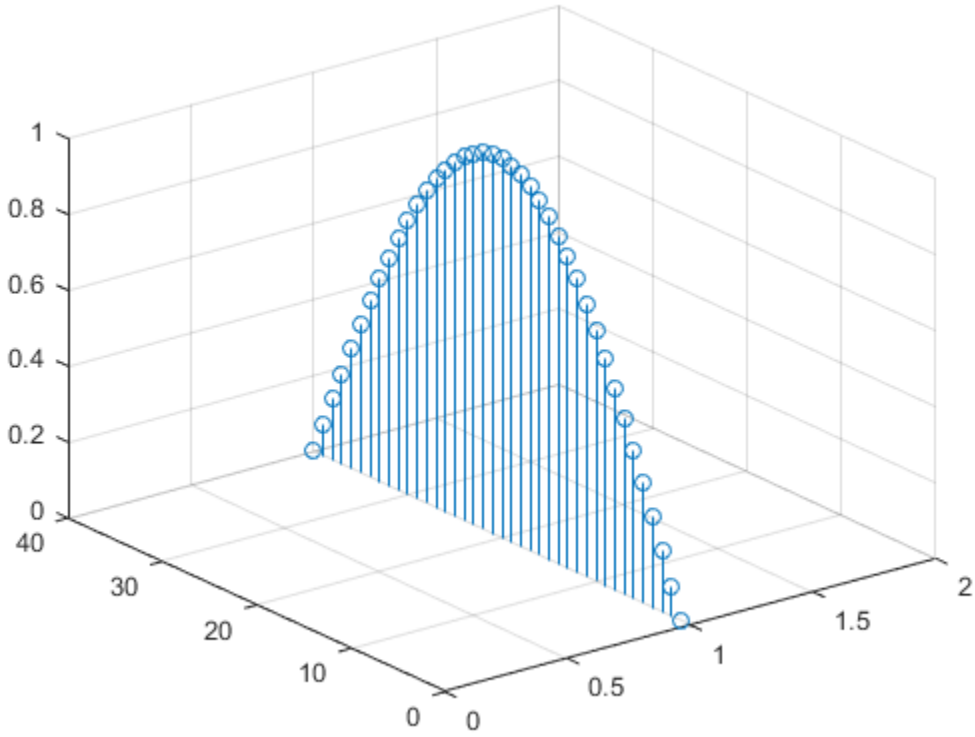


`stem3` plots elements of `Z` against the same *y* value at equally space *x* values.

Column Vector Input

Create a 3-D stem plot of cosine values between $-\pi/2$ and $\pi/2$ with a column vector input.

```
figure
X = linspace(-pi/2,pi/2,40)';
Z = cos(X);
stem3(Z)
```

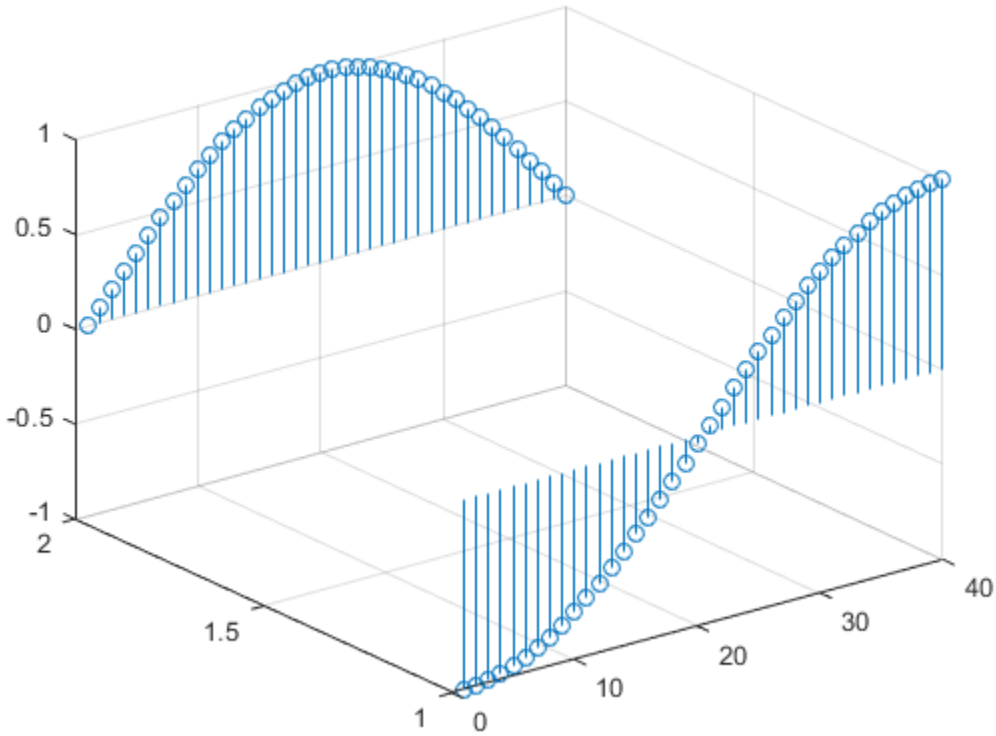


`stem3` plots elements of `Z` against the same x value at equally space y values.

Matrix Input

Create a 3-D stem plot of sine and cosine values between $-\pi/2$ and $\pi/2$ with a matrix input.

```
figure
X = linspace(-pi/2,pi/2,40);
Z = [sin(X); cos(X)];
stem3(Z)
```

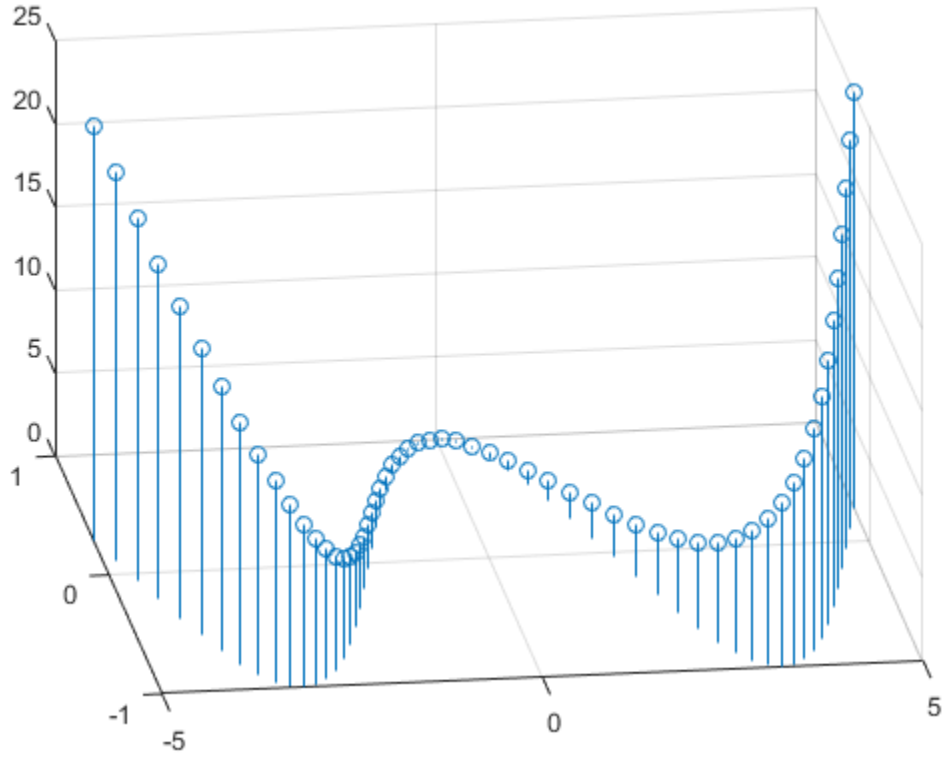


`stem3` plots each row of `Z` against the same *y* value at equally space *x* values.

Specify Stem Locations with Vector Inputs

Create a 3-D stem plot and specify the stem locations along a curve. Use `view` to adjust the angle of the axes in the figure.

```
figure
X = linspace(-5,5,60);
Y = cos(X);
Z = X.^2;
stem3(X,Y,Z)
view(-8,30)
```

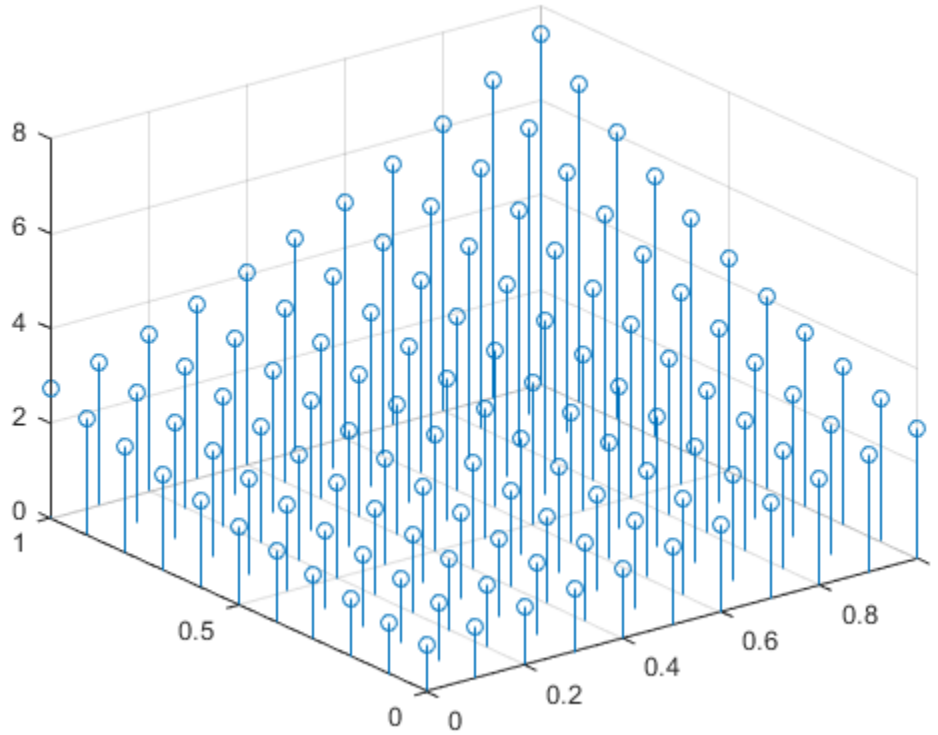


X and Y determine the stem locations. Z determines the marker heights.

Specify Stem Locations with Matrix Inputs

Create a 3-D stem plot with matrix data and specify the stem locations in the *xy*-plane.

```
figure
[X,Y] = meshgrid(0:.1:1);
Z = exp(X+Y);
stem3(X,Y,Z)
```

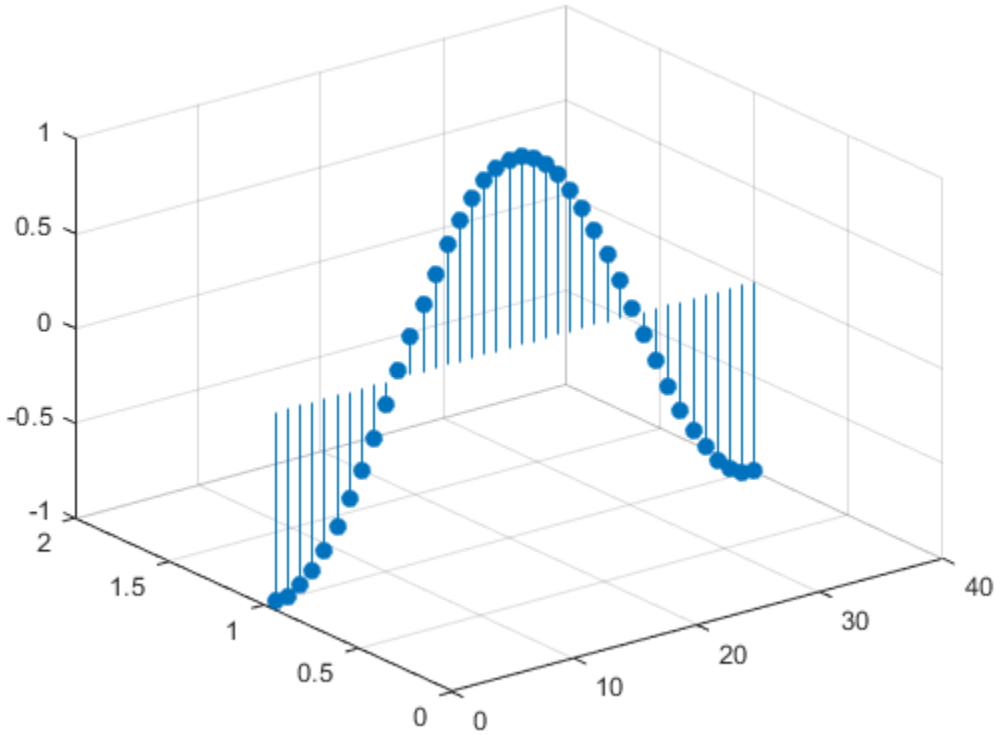


X and Y determine the stem locations. Z determines the marker heights.

Fill in Markers

Create a 3-D stem plot of cosine values between $-\pi$ and π and fill in the markers.

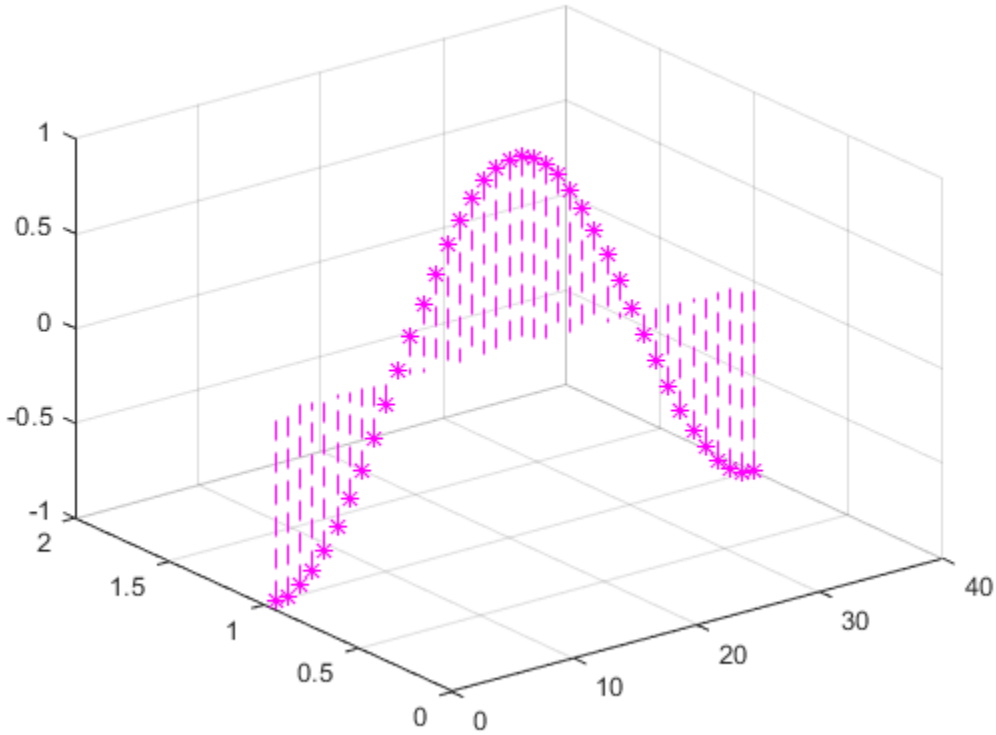
```
X = linspace(-pi,pi,40);
Z = cos(X);
stem3(Z,'filled')
```



Line Style, Marker Symbol, and Color Options

Create a 3-D stem plot of cosine values between $-\pi$ and π. Use a dashed line style for the stem, set the marker symbols to stars, and set the color to magenta.

```
figure
X = linspace(-pi,pi,40);
Z = cos(X);
stem3(Z,'--*m')
```



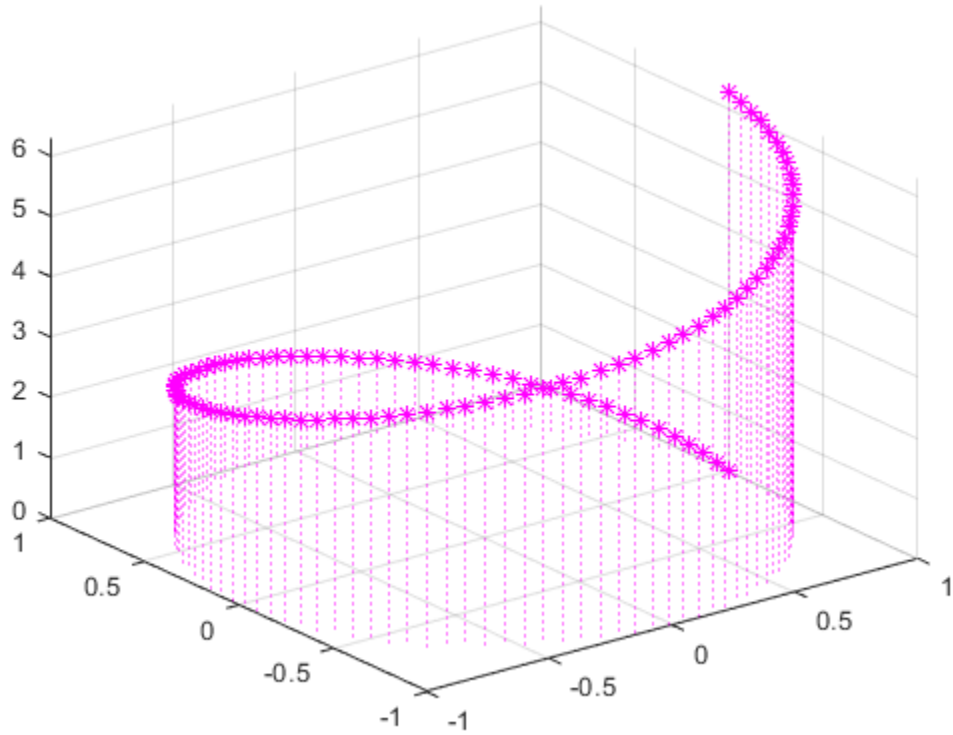
To specify only two of the three `LineSpec` options, omit the third option from the character vector. For example, `'*m'` sets the marker symbol and the color and uses the default line style.

Line Style, Marker Symbol, and Color Options

Create a 3-D stem plot and specify the stem locations along a circle. Set the stem to a dotted line style, the marker symbols to stars, and the color to magenta.

```
figure
theta = linspace(0,2*pi);
```

```
X = cos(theta);
Y = sin(theta);
Z = theta;
stem3(X,Y,Z,':*m')
```

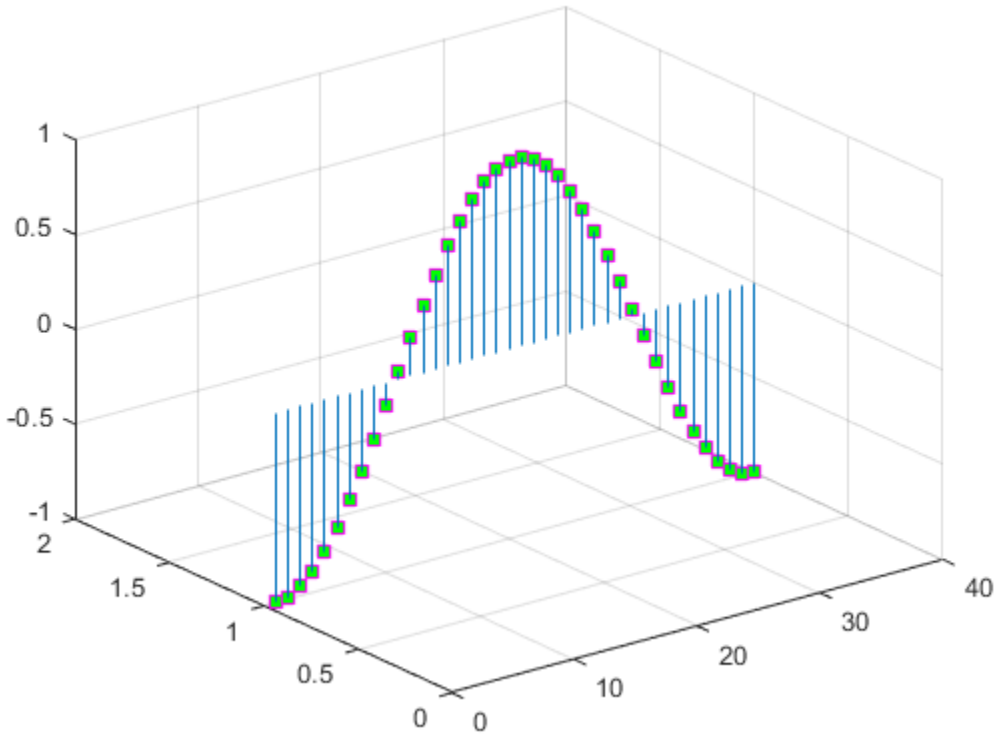


X and Y determine the stem locations. Z determines the marker heights.

Additional Style Options

Create a 3-D stem plot of cosine values between $-\pi$ and π. Set the marker symbols to squares with green faces and magenta edges.

```
figure
X = linspace(-pi,pi,40);
Z = cos(X);
stem3(Z,'Marker','s',...
     'MarkerEdgeColor','m',...
     'MarkerFaceColor','g')
```

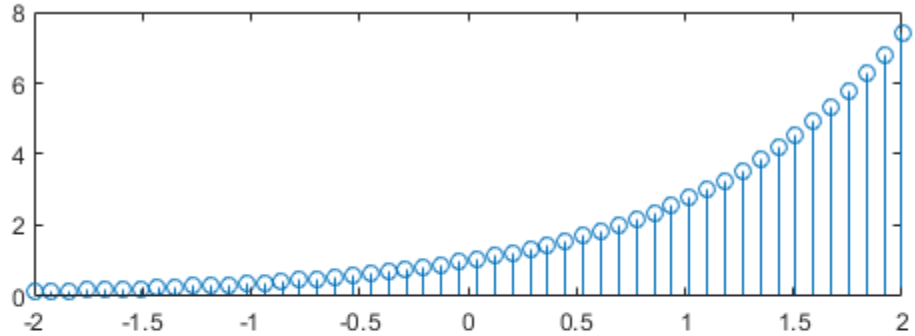


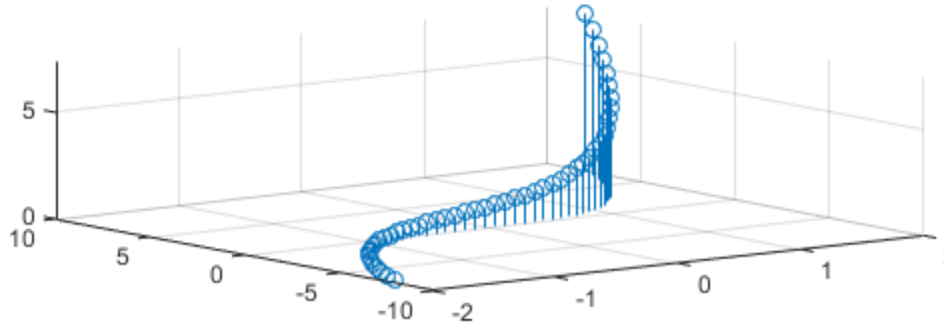
Stem Plot in Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout.

Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Create separate stem plots in the axes by specifying the axes object as the first argument to `stem3`.

```
X = linspace(-2,2,50);
Y = X.^3;
Z = exp(X);
tiledlayout(2,1)

% Top plot
ax1 = nexttile;
stem(ax1,X,Z)

% Bottom plot
ax2 = nexttile;
stem3(ax2,X,Y,Z)
```

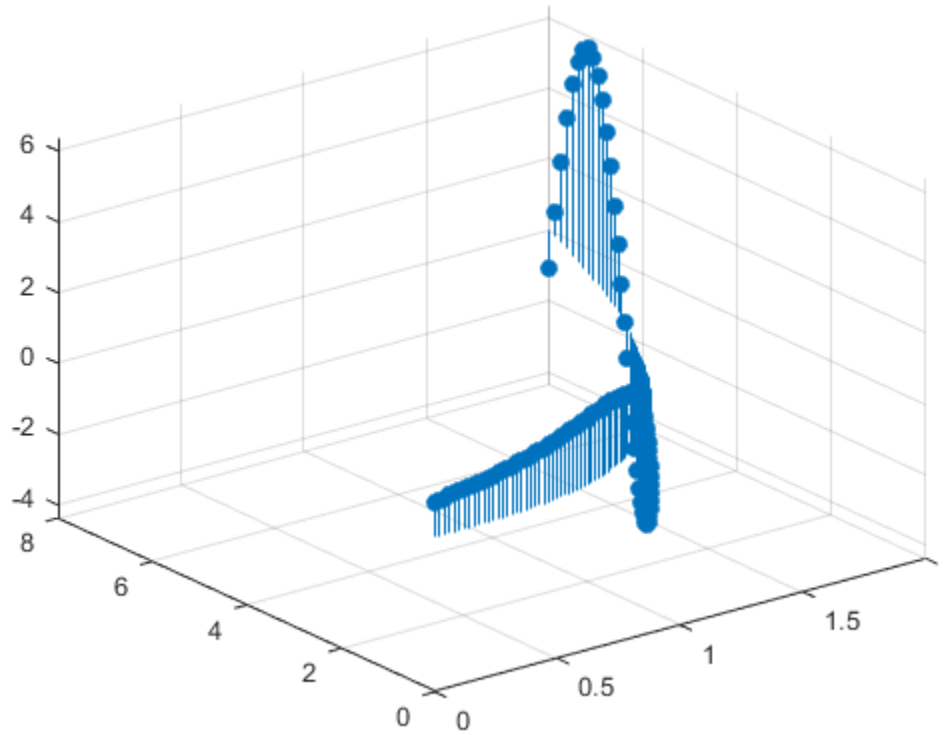




Modify Stem Series After Creation

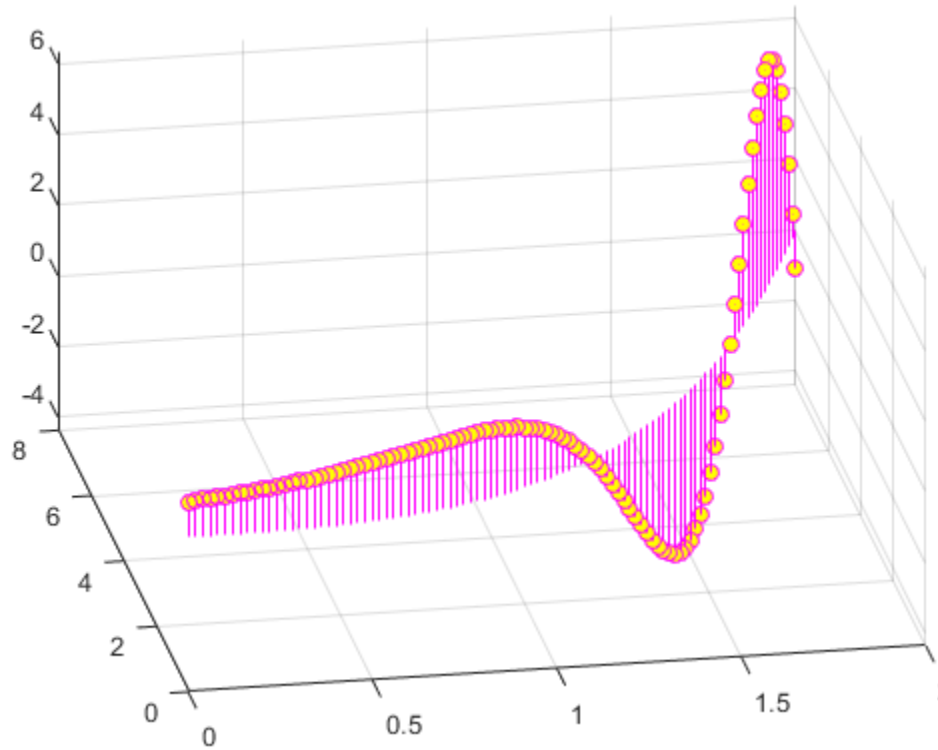
Create a 3-D stem plot and return the stem series object.

```
X = linspace(0,2);
Y = X.^3;
Z = exp(X).*cos(Y);
h = stem3(X,Y,Z,'filled');
```



Change the color to magenta and set the marker face color to yellow. Use `view` to adjust the angle of the axes in the figure. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead.

```
h.Color = 'm';
h.MarkerFaceColor = 'y';
view(-10,35)
```



Input Arguments

Z — Data sequence to display
vector or matrix

Data sequence to display, specified as a vector or matrix. `stem3` plots each element in `Z` as a stem extending from the *xy*-plane and terminating at the data value.

- If `Z` is a row vector, `stem3` plots all elements against the same *y* value at equally spaced *x* values.

- If Z is a column vector, `stem3` plots all elements against the same *x* value at equally spaced *y* values.
- If Z is a matrix, `stem3` plots each row of Z against the same *y* value at equally spaced *x* values.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

X — Locations to plot values of Z
vector or matrix

Locations to plot values ofZ, specified as a vector or a matrix. Inputs X, Y and Z must be vectors or matrices of the same size.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Y — Locations to plot values of Z
vector or matrix

Locations to plot values of Z, specified as a vector or a matrix. Inputs X, Y and Z must be vectors or matrices of the same size.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`LineSpec` — Line style, marker, and color
character vector | string

Line style, marker, and color, specified as a character vector or string. For more information, see `LineSpec`.

Example: `':*r'`

Data Types: `char`

`ax` — `Axes` object
`Axes` object

`Axes` object. If you do not specify the axes, then `stem` plots into the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'LineStyle',':','MarkerFaceColor','red'` plots the stem as a dotted line and sets the marker face color to red.

The properties listed here are only a subset. For a complete list, see Stem.

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width of stem and marker edge
`0.5` (default) | positive value

Line width of stem and marker edge, specified as a positive value in point units.

Example: `0.75`

`Color` — Stem color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Stem color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: 'blue'

Example: [0 0 1]

Example: '#0000FF'

Marker — Marker symbol
'o' (default) | '+' | '*' | '.' | 'x' | ...

Marker symbol, specified as one of the markers listed in this table.

Value	Description
'o'	Circle
'+'	Plus sign
'*'	Asterisk
'.'	Point
'x'	Cross
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'	Six-pointed star (hexagram)
'none'	No markers

Example: '+'

Example: 'diamond'

MarkerSize — Marker size
6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

`MarkerEdgeColor` — Marker outline color

`'auto'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'auto'` uses the same color as the `Color` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` option uses the same color as the `Color` property of the parent axes. If you specify `'auto'` and the axes plot box is invisible, the marker fill color is the color of the figure.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Output Arguments

h — Stem object
Stem

Stem object. This is a unique identifier, which you can use to modify the properties of the Stem object after it is created.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions
`bar` | `plot` | `stairs` | `stem`

Properties
Stem

Topics

“Combine Line and Stem Plots”

Introduced before R2006a

Stem Properties

Stem chart appearance and behavior

Description

`Stem` properties control the appearance and behavior of a `Stem` object. By changing property values, you can modify certain aspects of the stem chart.

```
h = stem(1:10);
h.Color = 'red';
```

Properties

Color and Styling

`Color` — Stem color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Stem color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`ColorMode` — Control how `Color` is set
`'auto'` (default) | `'manual'`

Control how the `Color` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the `Color` property by selecting a color from the `ColorOrder` property of the axes.
- `'manual'` — You control the value of the `Color` property manually, either by setting the value of the `Color` property directly on the object, or by including a color in the `LineSpec` argument when you call a plotting function.

If you change the value of the `Color` property manually, MATLAB changes the value of the `ColorMode` property to `'manual'`.

`LineStyle` — Stem line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Stem line style, specified as one of the line styles listed in this table.

Line Style	Description
`'-'`	Solid line
`'--'`	Dashed line
`':'`	Dotted line
`'-.'`	Dash-dotted line
`'none'`	No stems

Example: `'--'`

`LineStyleMode` — Control how `LineStyle` is set
`'auto'` (default) | `'manual'`

Control how the `LineStyle` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the `LineStyle` property by selecting a line style from the `LineStyleOrder` property of the axes.
- `'manual'` — You control the value of the `LineStyle` property manually, either by setting the value of the `LineStyle` property directly on the object, or by specifying the `LineSpec` argument when you call a plotting function.

If you change the value of the `LineStyle` property manually, MATLAB changes the value of the `LineStyleMode` property to `'manual'`.

`LineWidth` — Line width of stem and marker edge
`0.5` (default) | scalar numeric value greater than 0

Line width of stem and marker edge, specified as a scalar numeric value greater than 0 in point units. The default line width is `0.5` points.

Example: `0.75`

Markers

`Marker` — Marker symbol

`'o'` (default) | `'+'` | `'*'` | `'.'` | `'x'` | ...

Marker symbol, specified as one of the markers listed in this table.

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Example: `'+'`

Example: `'diamond'`

`MarkerMode` — Control how `Marker` is set

`'auto'` (default) | `'manual'`

Control how the `Marker` property is set, specified as one of these values:

- `'auto'` — MATLAB controls the value of the object's `Marker` property.
- `'manual'` — You control the value of the `Marker` property manually, either by setting the value of the `Marker` property directly on the object, or by including a marker in the `LineSpec` argument when you call a plotting function.

If you change the value of the `Marker` property manually, MATLAB changes the value of the `MarkerMode` property to `'manual'`.

`MarkerSize` — Marker size
6 (default) | positive value

Marker size, specified as a positive value in points, where 1 point = 1/72 of an inch.

`MarkerEdgeColor` — Marker outline color
`'auto'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'auto'` uses the same color as the `Color` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`MarkerFaceColor` — Marker fill color

`'none'` (default) | `'auto'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker fill color, specified as `'auto'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The `'auto'` option uses the same color as the `Color` property of the parent axes. If you specify `'auto'` and the axes plot box is invisible, the marker fill color is the color of the figure.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to

F. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Baseline

`BaseValue` — Baseline value

`0` (default) | numeric scalar value

Baseline value, specified as a numeric scalar value.

The baseline value that you specify applies to either the *x*-axis or the *y*-axis depending on the bar chart orientation. If you change the orientation of the bar chart from vertical to horizontal, or vice versa, the baseline value might change. Set the `BaseValue` property after setting the `Horizontal` property.

`ShowBaseLine` — Baseline visibility
`'on'` (default) | `'off'`

Baseline visibility, specified as one of these values:

- `'on'` — Show the baseline.
- `'off'` — Hide the baseline.

`BaseLine` — Baseline
baseline object

This property is read-only.

Baseline object. For a list of baseline properties, see Baseline.

Data

`XData` — Values along x-axis
`[ ]` (default) | vector

Values along the *x*-axis, specified as a vector.

- For 2-D stem charts, the input argument `X` to the `stem` function determines the *x*-values. If you do not specify `X`, then `stem` uses the indices of `YData` as the *x*-values. `XData` and `YData` must have equal lengths.
- For 3-D stem charts, the input argument `X` to the `stem3` function determines the *x*-values. If you do not specify `X`, then `stem3` uses the indices of `ZData` as the *x*-values. `XData`, `YData`, and `ZData` must have equal lengths.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`XDataMode` — Selection mode for `XData`
`'auto'` (default) | `'manual'`

Selection mode for `XData`, specified as one of these values:

- `'auto'` — Use the indices of the values in `YData` (or `ZData` for 3-D plots).
- `'manual'` — Use manually specified values. To specify the values, set the `XData` property or specify the input argument `X` to the plotting function.

`XDataSource` — Variable linked to `XData`
`''` (default) | character vector | string

Variable linked to `XData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `XData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `XData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'x'`

`YData` — Values along y-axis
`[]` (default) | vector

Values along the *y*-axis, specified as a vector.

- For 2-D stem charts, the input argument `Y` to the `stem` function determines the *y*-values. `YData` defines the stem heights. `XData` and `YData` must have equal lengths.
- For 3-D stem charts, the input argument `Y` to the `stem3` function determines the *y*-values. `YData` defines the locations of the stems along the *y*-axis. `XData`, `YData`, and `ZData` must have equal lengths.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`YDataSource` — Variable linked to `YData`
`''` (default) | character vector | string

Variable linked to `YData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `YData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `YData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'y'`

`ZData` — Values along z-axis
`[]` (default) | vector

Values along the *z*-axis, specified as a vector.

- For 2-D stem charts, `ZData` is empty by default.
- For 3-D stem charts, the input argument `Z` to the `stem3` function determines the *z*-values. `XData`, `YData`, and `ZData` must have equal lengths.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ZDataSource` — Variable linked to `ZData`
`''` (default) | character vector | string

Variable linked to `ZData`, specified as a character vector or string containing a MATLAB workspace variable name. MATLAB evaluates the variable in the base workspace to generate the `ZData`.

By default, there is no linked variable so the value is an empty character vector, `''`. If you link a variable, then MATLAB does not update the `ZData` values immediately. To force an update of the data values, use the `refreshdata` function.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Example: `'z'`

Legend

`DisplayName` — Legend label

`''` (default) | character vector | string scalar

Legend label, specified as a character vector or string scalar. The legend does not display until you call the `legend` command. If you do not specify the text, then `legend` sets the label using the form `'dataN'`.

`Annotation` — Control for including or excluding object from legend

`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend (default).
- `'off'` — Do not include the object in the legend.

For example, to exclude a graphics object, `go`, from the legend set the `IconDisplayStyle` property to `'off'`.

```
go.Annotation.LegendInformation.IconDisplayStyle = 'off';
```

Alternatively, you can control the items in a legend using the `legend` function. Specify the first input argument as a vector of the graphics objects to include. If you do not specify an existing graphics object in the first input argument, then it does not appear in the legend. However, graphics objects added to the axes after the legend is created do appear in the legend. Consider creating the legend after creating all the plots to avoid extra items.

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.

- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`DataTipTemplate` — Data tip content
`DataTipTemplate` object

Data tip content, specified as a `DataTipTemplate` object. You can control the content that appears in a data tip by modifying the properties of the underlying `DataTipTemplate` object. For a list of properties, see DataTipTemplate.

For an example of modifying data tips, see “Create Custom Data Tips”.

Note The `DataTipTemplate` object is not returned by `findobj` or `findall`, and it is not copied by `copyobj`.

`UIContextMenu` — Context menu
empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state
`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles
`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

`Clipping` — Clipping of object to axes limits
`'on'` (default) | `'off'`

Clipping of the object to the axes limits, specified as one of these values:

- `'on'` — Do not display parts of the object that are outside the axes limits.
- `'off'` — Display the entire object, even if parts of it appear outside the axes limits. Parts of the object might appear outside the axes limits if you create a plot, set `hold on`, freeze the axis scaling, and then create the object so that it is larger than the original plot.

The `Clipping` property of the axes that contains the object must be set to `'on'`. Otherwise, this property has no effect. For more information about the clipping behavior, see the `Clipping` property of the axes.

Callbacks

`ButtonDownFcn` — Mouse-click callback
`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.

  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.

- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks when visible. The `Visible` property must be set to `'on'` and you must click a part of the `Stem` object that has a defined color. You cannot click a part that has an associated color property set to `'none'`. If the plot contains markers, then the entire marker is clickable if either the edge or the fill has a defined color. The `HitTest` property determines if the `Stem` object responds to the click or if an ancestor does.
- `'none'` — Cannot capture mouse clicks. Clicking the `Stem` object passes the click to the object below it in the current view of the figure window. The `HitTest` property of the `Stem` object has no effect.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `Stem` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `Stem` object that has one of these:
  - `HitTest` property set to `'on'`
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

---

Note The `PickableParts` property determines if the `Stem` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent

`Axes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `Group`, or `Transform` object.

`Children` — Children

empty `GraphicsPlaceholder` array | `DataTip` object array

Children, returned as an empty `GraphicsPlaceholder` array or a `DataTip` object array. Use this property to view a list of data tips that are plotted on the chart.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the `DataTip` object to the chart object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object
`'stem'` (default)

This property is read-only.

Type of graphics object, returned as `'stem'`. Use this property to find all objects of a given type within a plotting hierarchy, such as searching for the type using `findobj`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`stem` | `stem3`

Topics

“Access Property Values”
“Graphics Object Properties”

Introduced before R2006a

stlread

Create triangulation from STL file

Syntax

```
TR = stlread(filename)
[TR,fileformat,attributes,solidID] = stlread(filename)
```

Description

`TR = stlread(filename)` returns a `triangulation` object `TR` containing the triangles defined in an STL file.

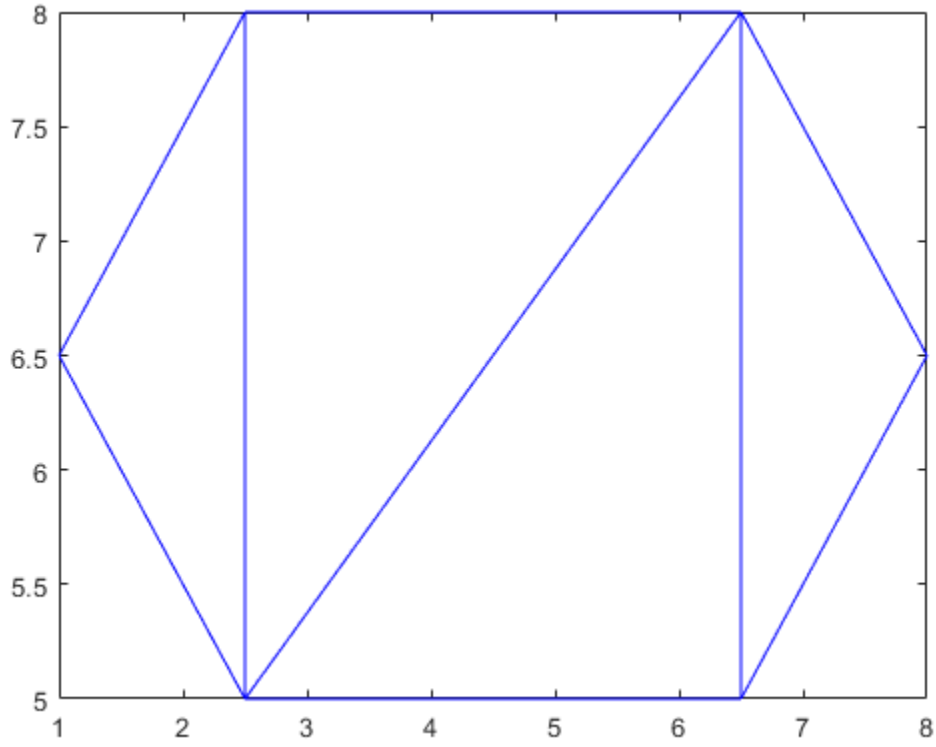
`[TR,fileformat,attributes,solidID] = stlread(filename)` also returns the file format of `filename`, attributes when the file is binary, and solid identification numbers when the file is text.

Examples

Read Triangulation from STL Text File

Create and plot a `triangulation` object from the data contained in the file `tristltext.stl`.

```
TR = stlread('tristltext.stl');
triplot(TR)
```



Input Arguments

`filename` — STL file name
character vector | scalar `string`

STL file name, specified as a character vector or scalar `string`. The file name must end with the `.stl` or `.STL` extension. If the file contains degenerate triangles, then `stlread` automatically removes them.

Example: `'stltextfile.stl'`

Output Arguments

TR — Triangulation
`triangulation` object

Triangulation, returned as a `triangulation` object.

`fileformat` — File format
`'binary'` | `'text'`

File format, returned as either `'binary'` or `'text'`.

`attributes` — Binary attributes
`uint16` vector

Binary attributes, returned as a `uint16` vector. When the input file is binary, `attributes` contains coded information about the triangles. Its length is equal to the number of triangles in the triangulation. If the input file is a text file, then `attributes` is an empty array.

`solidID` — Solid grouping index
vector

Solid grouping index, returned as a vector of identification numbers. When the input file is a text file, the identification numbers assign each triangle to a grouping of triangles in the triangulation. When the input file is binary, `solidID` contains all ones. The length of `solidID` is equal to the number of triangles in the triangulation.

Data Types: `double`

See Also

`triangulation`

Introduced in R2018b

stlwrite

Create STL file from triangulation

Syntax

```
stlwrite(TR,filename)
stlwrite(TR,filename,fileformat)
stlwrite( ___ ,Name,Value)
```

Description

`stlwrite(TR,filename)` writes a triangulation `TR` to a binary STL file `filename`. The triangulation can be either a `triangulation` object or a 2-D `delaunayTriangulation` object.

`stlwrite(TR,filename,fileformat)` also specifies a file format for the written file. `fileformat` can be either `'binary'` (default) or `'text'`.

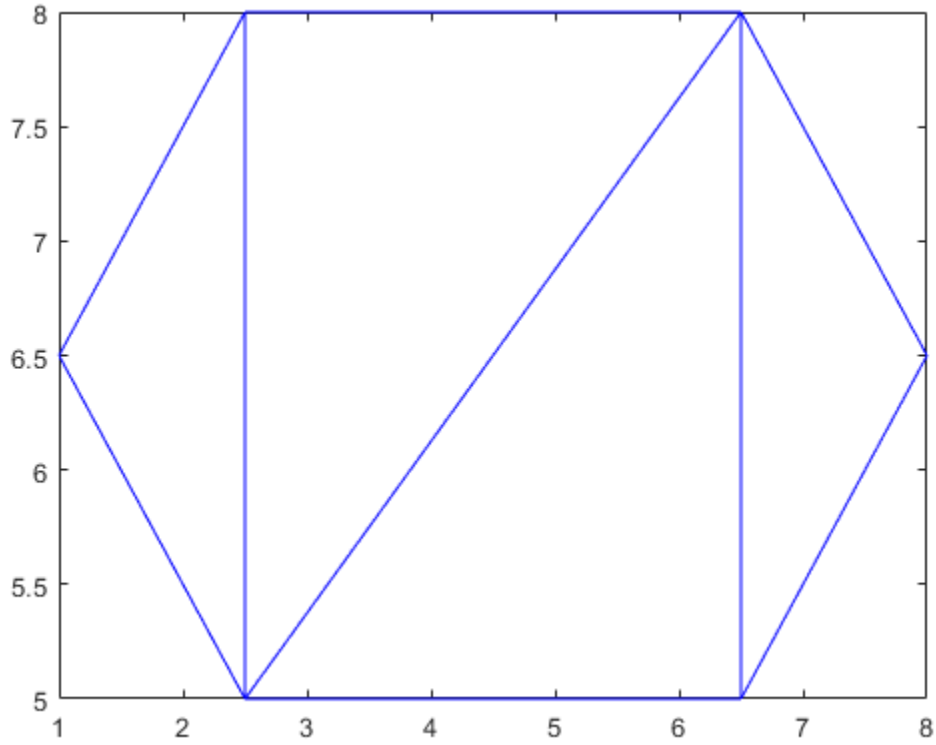
`stlwrite( ___ ,Name,Value)` specifies additional options for writing to the STL file using one or more `Name,Value` pair arguments for either of the previous syntaxes. For example, `stlwrite(TR,'stlbinary','Attribute',attributes)` also writes a `uint16` vector of attributes for each triangle in `TR`.

Examples

Write Triangulation to STL Text File

Create and plot a 2-D `triangulation` object.

```
P = [2.5 8.0; 6.5 8.0; 2.5 5.0; 6.5 5.0; 1.0 6.5; 8.0 6.5];
T = [5 3 1; 3 2 1; 3 4 2; 4 6 2];
TR = triangulation(T,P);
triplot(TR)
```



Write the triangulation to a text file named `tritext.stl`.

```
stlwrite(TR,'tritext.stl','text')
```

Input Arguments

TR — Triangulation
`triangulation` object | 2-D `delaunayTriangulation` object

Triangulation, specified as a `triangulation` object or a 2-D `delaunayTriangulation` object.

`filename` — STL file name
character vector | scalar `string`

STL file name, specified as a character vector or scalar `string`. The file name must end with the `.stl` or `.STL` extension.

Example: `'stltextfile.stl'`

`fileformat` — File format
`'binary'` | `'text'`

File format, specified as either `'binary'` or `'text'`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify the name and value pair arguments in any order as `Name1,Value1,Name2,Value2`.

Example: `stlwrite(TR,'stltext','SolidIndex',solidIDs)`

`Attribute` — Binary attributes
`uint16` vector

Binary attributes, specified as a `uint16` vector. When the input file is a binary file, `attributes` can contain coded information about the triangles. Its length must be equal to the number of triangles in the triangulation. This parameter is not supported when the input file is a text file.

`SolidIndex` — Solid grouping index
vector

Solid grouping index, specified as a vector of identification numbers. When the input file is a text file, the identification numbers must assign each triangle to a grouping of triangles in the triangulation. The length of the vector must be equal to the number of triangles in the triangulation. This parameter is not supported when the input file is binary.

See Also

`delaunayTriangulation` | `triangulation`

Introduced in R2018b

stopasync

(Not recommended) Stop asynchronous read and write operations

Note This `serial` object function is not recommended. Use `serialport` object functions instead. See “Compatibility Considerations”.

Syntax

```
stopasync(obj)
```

Description

`stopasync(obj)` stops any asynchronous read or write operation that is in progress for the serial port object, `obj`.

Tips

You can write data asynchronously using the `fprintf` or `fwrite` function. You can read data asynchronously using the `readasync` function, or by configuring the `ReadAsyncMode` property to `continuous`. In-progress asynchronous operations are indicated by the `TransferStatus` property.

If `obj` is an array of serial port objects and one of the objects cannot be stopped, the remaining objects in the array are stopped and a warning is returned. After an object stops:

- Its `TransferStatus` property is configured to `idle`.
- Its `ReadAsyncMode` property is configured to `manual`.
- The data in its output buffer is flushed.

Data in the input buffer is not flushed. You can return this data to the MATLAB workspace using any of the synchronous read functions. If you execute the `readasync` function, or

configure the `ReadAsyncMode` property to `continuous`, then the new data is appended to the existing data in the input buffer.

Compatibility Considerations

`serial` object interface is not recommended

Not recommended starting in R2019b

Use of this function with a `serial` object is not recommended. To access a serial port device, use a `serialport` object with its functions and properties instead.

See “Transition Your Code to serialport Interface” for more information about using the recommended functionality.

See Also

Functions

`fprintf` | `fwrite` | `readasync`

Introduced before R2006a

str2double

Convert strings to double precision values

Syntax

```
X = str2double(str)
```

Description

`X = str2double(str)` converts the text in `str` to double precision values. `str` contains text that represents real or complex numeric values. `str` can be a character vector, a cell array of character vectors, or a string array. If `str` is a character vector or string scalar, then `X` is a numeric scalar. If `str` is a cell array of character vectors or a string array, then `X` is a numeric array that is the same size as `str`.

Text that represents a number can contain digits, a comma (thousands separator), a decimal point, a leading `+` or `-` sign, an `e` preceding a power of 10 scale factor, and an `i` or a `j` for a complex unit. You cannot use a period as a thousands separator, or a comma as a decimal point.

If `str2double` cannot convert text to a number, then it returns a `NaN` value.

Examples

Convert Character Vectors to Numbers

Convert character vectors that represent numbers to double precision values. You can also convert a cell array of character vectors to a numeric array of the same size.

Convert a character vector to a real number.

```
X = str2double('3.1416')
```

```
X = 3.1416
```

Convert a character vector when it represents a number using exponential notation.

```
X = str2double('2.998e8')
```

```
X = 299800000
```

Convert a character vector that uses a comma as the thousands separator.

```
X = str2double('1,200.34')
```

```
X = 1.2003e+03
```

Convert a character vector to a complex number.

```
X = str2double('3.14 + 2.7i')
```

```
X = 3.1400 + 2.7000i
```

Convert a cell array of character vectors to a numeric array.

```
str = {'2.718','3.1416';
       '137','0.015'};
X = str2double(str)
```

```
X = 2×2

    2.7180    3.1416
  137.0000    0.0150
```

Convert String Arrays to Numeric Arrays

Starting in R2016b, you can create string arrays using the `string` function. You can convert strings to numbers using the `str2double` function.

Create a string that represents a number. Convert it to a numeric scalar.

```
str = string('81470.5')
```

```
str =
"81470.5"
```

```
X = str2double(str)
```

```
X = 8.1471e+04
```

Create a string array representing numbers. Convert it to a numeric array that is the same size.

```
str = string({'91.57','95.95','3.57';
              '79.22','65.57','84.91'})
```

```
str = 2x3 string array
    "91.57"    "95.95"    "3.57"
    "79.22"    "65.57"    "84.91"
```

```
X = str2double(str)
```

```
X = 2×3

   91.5700   95.9500    3.5700
   79.2200   65.5700   84.9100
```

Input Arguments

`str` — Text that represents numbers
character vector | cell array of character vectors | string array

Text that represents numbers, specified as a character vector, a cell array of character vectors, or a string array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Input text must be a string scalar or a character vector.
- Generated code always returns a complex result.
- Generated code results can differ from MATLAB results for misplaced commas (commas that are not used as thousands separators).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- `str` must be a string array or a cell array of character vectors.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cast` | `char` | `hex2num` | `num2str` | `str2num`

Introduced before R2006a

str2func

Construct function handle from character vector

Syntax

```
fh = str2func(str)
```

Description

`fh = str2func(str)` constructs a function handle, `fh`, from a function name or text representation of an anonymous function.

Function handles created using `str2func` do not have access to variables outside of their local workspace or to nested functions. If your function handle contains these variables or functions, MATLAB throws an error when you invoke the handle. Also, if you use a text representation of an anonymous function, the resulting function handle does not have access to private or local functions.

Examples

Convert Character Vector to Function Handle

Convert the character vector `'ones'` to a function handle, and call the `ones` function using the handle.

```
c = 'ones';
fh = str2func(c)
```

```
fh = function_handle with value:
    @ones
```

```
fh(1,5)
```

```
ans = 1×5

     1     1     1     1     1
```

Convert Character Vector to Handle to Anonymous Function

Convert a character vector that represents an anonymous function to a function handle. Workspace variables are not available to the `str2func` function. Therefore, include values in the character vector that are necessary to evaluate the expression and that are not defined as function inputs.

Define a character vector that represents the anonymous function $7x - 13$. Convert the character vector to a function handle.

```
str = '@(x)7*x-13';
fh = str2func(str)
```

```
fh =

    @(x)7*x-13
```

Call the anonymous function using the handle.

```
fh(3)
```

```
ans =

     8
```

If you include workspace variables in your character vector, `str2func` creates the function handle, but MATLAB throws an error when you invoke the function handle.

```
a = 13;
str = '@(x)7*x-a';
fh = str2func(str);

fh(3)
```

```
Undefined function or variable 'a'.

Error in @(x)7*x-a
```

Examine Differences Between `str2func` and `eval`

Create a function that returns two function handles used to simulate the roll of dice. The first die (`d1`) returns a number from 1 through 6, but the second die (`d2`) always returns the number 1.

Create the following function in a folder on your MATLAB path. When `str2func` is used with a character vector representing an anonymous function, it does not have access to the local function. Therefore, MATLAB calls the built-in `randi` function, and returns a number from 1 through 6. The `eval` function does have access to the local function, so `d2` uses the overloaded `randi` and always returns 1.

```
function [d1,d2] = diceRoll
str = '@()randi([1 6],1)';
d1 = str2func(str);
d2 = eval(str);
end

function r = randi(~,~)
r = 1;
end
```

At the command prompt, call the `diceRoll` function.

```
[p1,p2] = diceRoll
```

```
p1 =

  function_handle with value:

    @()randi([1,6],1)


p2 =
```

```
  function_handle with value:

    @()randi([1,6],1)
```

Both `p1` and `p2` appear to be associated with the same anonymous function.

Invoke the function handles. The result from `p1` varies from 1 through 6. The result from `p2` is always 1.

```
p1()
p2()
```

```
ans =

     5


ans =

     1
```

Input Arguments

`str` — Text to convert to function handle
function name | character vector representation of anonymous function | string scalar representation of anonymous function

Text to convert to a function handle, specified as a function name or a character vector or string scalar representation of an anonymous function.

Example: `str = 'cos'`

Example: `str = '@(x) x.^2'`

Tips

- A function handle that stores variable values does not retain its original value when you use `func2str` to convert it to a character vector, and then convert it back to a handle with `str2func`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The input argument must be constant/known at compile time.
- Code generation does not support an input argument that represents an anonymous function.

See Also

`func2str` | `functions`

Topics

"Create Function Handle"

Introduced before R2006a

str2mat

(Not recommended) Form blank-padded character array from strings

Note `str2mat` is not recommended. Use `char` instead.

Syntax

```
S = str2mat(T1, T2, T3, ...)
```

Description

`S = str2mat(T1, T2, T3, ...)` forms the character array `S` containing the character arrays `T1, T2, T3, ...` as rows. The function automatically pads each row with blanks so that every row of `S` has the same number of characters. Each input argument, `Ti`, can itself be a character array with one or more rows. This function allows for the creation of arbitrarily large character arrays. Empty character arrays are significant.

Examples

```
x = str2mat('36842', '39751', '38453', '90307');

whos x
  Name      Size         Bytes  Class

  x         4x5             40  char array

x(2,3)

ans =

    7
```

Tips

`str2mat` differs from `strvcat` in that empty character arrays produce blank rows in the output. In `strvcat`, empty character arrays are ignored.

See Also

`char`

Introduced before R2006a

str2num

Convert character array or string to numeric array

Syntax

```
X = str2num(chr)
[X,tf] = str2num(chr)
```

Description

`X = str2num(chr)` converts a character array or string scalar to a numeric matrix. The input can include spaces, commas, and semicolons to indicate separate elements. If `str2num` cannot parse the input as numeric values, then it returns an empty matrix.

The `str2num` function does not convert cell arrays or nonscalar string arrays, and is sensitive to spacing around `+` and `-` operators. In addition, `str2num` uses the `eval` function, which can cause unintended side effects when the input includes a function name. To avoid these issues, use `str2double`.

`[X,tf] = str2num(chr)` additionally returns a second output argument that is `1` (`true`) if `str2num` successfully converts `chr`. Otherwise, `str2num` returns `0` (`false`).

Examples

Convert to Numeric Matrix

Convert character vectors that represent numbers.

```
X = str2num('100')
```

```
X = 100
```

```
X = str2num('100 200 300 400')
```

```
X = 1×4

   100   200   300   400
```

`str2num` interprets exponential notation.

```
X = str2num('12e-3 5.9e-3 -8.1e-3 2.56e-3; 5 11.2 17.9 33')
```

```
X = 2×4

    0.0120    0.0059   -0.0081    0.0026
    5.0000   11.2000   17.9000   33.0000
```

Convert to Integer

Convert a character vector to an unsigned 16-bit integer using `str2num` and `uint16`.

```
X = str2num('256');
X = uint16(X)
```

```
X = uint16
    256
```

Convert to Logical

Convert a character vector containing `true` and `false` to a logical array.

```
X = str2num('false true true false')
```

```
X = 1x4 logical array

   0   1   1   0
```

Check Conversion Status

Return the status of a conversion that fails. `tf` is `0`, and `X` is an empty matrix.

```
[X,tf] = str2num('12e-3 m/s, 5.9e-3 m/s')
```

```
X =

     []

tf = logical
   0
```

If you remove the extra text (`m/s`), then conversion succeeds.

```
[X,tf] = str2num('12e-3 5.9e-3')
```

```
X = 1×2

    0.0120    0.0059


tf = logical
   1
```

Input Arguments

`chr` — Representation of numeric matrix
character array | string scalar

Representation of a numeric matrix, specified as a character array or string scalar.

Text that represents a numeric matrix can contain spaces, commas, or semicolons, such as `'5'`, `'10,11,12'`, or `'5,10;15,20'`. In addition to numeric values and delimiters, input text also can include any of the following items:

- A decimal point
- Leading `+` or `-` signs
- The letter `e` or `d` preceding a power of 10 scale factor

- The letter `i` or `j` indicating a complex or imaginary number
- `true` or `false` indicating logical values

Space characters, or the lack of them, can be significant. For instance, `str2num('1+2i')` and `str2num('1 + 2i')` both return the complex number `1.0000 + 2.0000i`, while `str2num('1 +2i')` returns the 1-by-2 vector `[1.0000 + 0.0000i 0.0000 + 2.0000i]`. To avoid this problem, use the `str2double` function.

`str2num` converts character arrays and string scalars only. To convert nonscalar string arrays or cell arrays to numeric arrays, use the `str2double` function.

Output Arguments

X — Output array
numeric matrix

Output array, returned as a numeric matrix.

tf — True or false
`1` | `0`

True or false result, returned as a `1` or `0` of data type `logical`.

See Also

`cast` | `char` | `hex2num` | `num2str` | `sscanf` | `str2double`

Topics

"Convert from Character Arrays to Numeric Values"
"Convert from Numeric Values to Character Array"

Introduced before R2006a

strcat

Concatenate strings horizontally

Syntax

```
s = strcat(s1,...,sN)
```

Description

`s = strcat(s1,...,sN)` horizontally concatenates `s1,...,sN`. Each input argument can be a character array, a cell array of character vectors, or a string array.

- If any input is a string array, then the result is a string array.
- If any input is a cell array, and none are string arrays, then the result is a cell array of character vectors.
- If all inputs are character arrays, then the result is a character array.

For character array inputs, `strcat` removes trailing ASCII white-space characters: space, tab, vertical tab, newline, carriage return, and form feed. For cell and string array inputs, `strcat` does not remove trailing white space.

Examples

Concatenate Two Character Vectors

```
s1 = 'Good';
s2 = 'morning';
s = strcat(s1,s2)
```

```
s =
'Goodmorning'
```

Concatenate Two Cell Arrays

```
s1 = {'abcde','fghi'};
s2 = {'jkl','mn'};
s = strcat(s1,s2)
```

```
s = 1x2 cell array
    {'abcdejkl'}    {'fghimn'}
```

Concatenate Two Cell Arrays with Scalar Cell Array

```
firstnames = {'Abraham'; 'George'};
lastnames = {'Lincoln'; 'Washington'};
names = strcat(lastnames, {', '}, firstnames)
```

```
names = 2x1 cell array
    {'Lincoln, Abraham'  }
    {'Washington, George'}
```

Concatenate Two String Arrays

Starting in R2017a, you can create string arrays using double quotes. Concatenate them with the `strcat` function.

```
str1 = ["John ","Mary "];
str2 = ["Smith","Jones"];
str = strcat(str1,str2)
```

```
str = 1x2 string array
    "John Smith"    "Mary Jones"
```

Concatenate a character vector onto each element of the string array.

```
str = strcat(str,', M.D.')
```

```
str = 1x2 string array
    "John Smith, M.D."    "Mary Jones, M.D."
```

Input Arguments

s1,...,sN — Input text
character arrays | cell array of character vectors | string arrays

Input text, specified as character arrays, cell arrays of character vectors, or string arrays. When combining string or cell arrays with character arrays, the string or cell arrays must be either scalars or column vectors with the same number of rows as the character arrays.

Data Types: `char` | `cell` | `string`

Tips

- Character arrays also can be concatenated using left and right square brackets.

```
s1 = 'Good ';
s2 = 'Morning';
s = [s1 s2]
```

```
s =

Good Morning
```

See Also

`cat` | `cellstr` | `horzcat` | `join` | `strjoin` | `vertcat`

Introduced before R2006a

strcmp

Compare strings

Syntax

```
tf = strcmp(s1,s2)
```

Description

`tf = strcmp(s1,s2)` compares `s1` and `s2` and returns `1` (`true`) if the two are identical and `0` (`false`) otherwise. Text is considered identical if the size and content of each are the same. The return result `tf` is of data type `logical`.

The input arguments can be any combination of string arrays, character vectors, and cell arrays of character vectors.

Examples

Compare Two Character Vectors

Compare two different character vectors.

```
s1 = 'Yes';
s2 = 'No';
tf = strcmp(s1,s2)
```

```
tf = logical
   0
```

`strcmp` returns `0` because `s1` and `s2` are not equal.

Compare two equal character vectors.

```
s1 = 'Yes';
s2 = 'Yes';
tf = strcmp(s1,s2)
```

```
tf = logical
   1
```

`strcmp` returns `1` because `s1` and `s2` are equal.

Find Text in Cell Array

Find the word `'upon'` in a cell array of character vectors.

```
s1 = 'upon';
s2 = {'Once','upon';
      'a','time'};
tf = strcmp(s1,s2)
```

```
tf = 2x2 logical array

   0   1
   0   0
```

There is only one occurrence of `s1` in array `s2`, and it occurs at element `s2(1,2)`.

Compare Two Cell Arrays of Character Vectors

Compare each element in two cell arrays of character vectors.

```
s1 = {'Time','flies','when';
      'you''re','having','fun.'};
s2 = {'Time','drags','when';
      'you''re','anxiously','waiting.'};
tf = strcmp(s1,s2)
```

```
tf = 2x3 logical array

   1   0   1
```

```
   1   0   0
```

There are three instances of equal elements in `s1` and `s2`. These are `'Time'` at indices `(1,1)`, `'when'` at indices `(1,3)`, and `'you''re'` at indices `(2,1)`.

Compare String Arrays

Starting in R2017a, you can create strings using double quotes. Compare string arrays using `strcmp`.

```
s1 = ["A","bc";
      "def","G"];
s2 = ["B","c";
      "def","G"];

tf = strcmp(s1,s2)
```

```
tf = 2x2 logical array

   0   0
   1   1
```

You can compare and sort string arrays with relational operators, just as you can with numeric arrays.

Use `==` to determine which elements of two string arrays are equal.

```
s1 == s2
```

```
ans = 2x2 logical array

   0   0
   1   1
```

Use `<` to determine which elements of `s1` are less than the corresponding elements of `s2` according to ASCII dictionary order.

```
s1 < s2
```

```
ans = 2x2 logical array

   1   1
   0   0
```

Input Arguments

s1,s2 — Input text

character vector | character array | cell array of character vectors | string array

Input text, with each input specified as a character vector, a character array, a cell array of character vectors, or a string array. The order of the inputs does not affect the comparison results.

- If both `s1` and `s2` are string arrays or cell arrays of character vectors, then `s1` and `s2` must be the same size, unless one of them is scalar.
- If both `s1` and `s2` are character arrays with multiple rows, then `s1` and `s2` can have different numbers of rows.
- When comparing a nonscalar cell array of character vectors or a string array to a multirow character array, the cell array or string array must be a column vector with the same number of rows as the character array.

Data Types: `char` | `cell` | `string`

Output Arguments

tf — True or false result

`1` | `0` | logical array

True or false result, returned as a `1` or `0` of data type `logical`.

- If each input is either a string scalar, scalar cell, or a character vector, then `tf` is a scalar.
- If at least one input is either a string array or a cell array of character vectors, then `tf` is an array the same size as the input array.

- If one input is a character array with multiple rows, and the other input is either a scalar cell or a string scalar, then `tf` is an `n`-by-`1` array, where `n` is the number of rows in the character array.
- If both inputs are character arrays, `tf` is a scalar.

Tips

- The `strcmp` function is intended for comparison of text. If used on unsupported data types, `strcmp` always returns `0`.
- For case-insensitive text comparison, use `strcmpi` instead of `strcmp`.
- Although `strcmp` shares a name with a C function, it does not follow the C language convention of returning `0` when the text inputs match.
- With string arrays, you can use relational operators (==, ~=, <, >, <=, >=) instead of `strcmp`. You can compare and sort string arrays just as you can with numeric arrays.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Enumeration inputs are not supported.
- When one input is a cell array and the other input is a character array, the character array must be a compile-time row vector.
- When both inputs are empty character arrays that have different sizes, the generated code returns `true`.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- Inputs must be string arrays or cell arrays of character vectors.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`eq` | `ne` | `regexp` | `regexpi` | `sort` | `strcmpi` | `strfind` | `strncmp` | `strncmpi`

Topics

“Compare Text”

Introduced before R2006a

strcmpi

Compare strings (case insensitive)

Syntax

```
tf = strcmpi(s1,s2)
```

Description

`tf = strcmpi(s1,s2)` compares `s1` and `s2`, ignoring any differences in letter case. The function returns `1` (`true`) if the two are identical and `0` (`false`) otherwise. Text is considered identical if the size and content of each are the same, aside from case. The return result `tf` is of data type `logical`.

The input arguments can be any combination of string arrays, character vectors, and cell arrays of character vectors.

Examples

Compare Two Character Vectors While Ignoring Case

Compare two different character vectors, ignoring any differences in letter case.

```
s1 = 'Yes';
s2 = 'No';
tf = strcmpi(s1,s2)
```

```
tf = logical
   0
```

`strcmpi` returns `0` because `s1` and `s2` are not equal, even when ignoring case.

Compare two equal character vectors.

```
s1 = 'Yes';
s2 = 'yes';
tf = strcmpi(s1,s2)
```

```
tf = logical
   1
```

`strcmpi` returns `1` because `s1` and `s2` are equal when ignoring case.

Find Case-Insensitive Match in Cell Array

Find text that matches the word `'once'` in a cell array of character vectors. Ignore case.

```
s1 = 'once';
s2 = {'Once','upon';
      'a','time'};
tf = strcmpi(s1,s2)
```

```
tf = 2x2 logical array

   1   0
   0   0
```

When you ignore case, there is one occurrence of `s1` in the array `s2`, and it occurs at the element `s2(1,1)`.

Compare Cell Arrays While Ignoring Case

Create two cell arrays of character vectors. To compare them while ignoring case, use the `strcmpi` function.

```
s1 = {'Tinker', 'Tailor';
      '  Soldier', 'Spy'};
s2 = {'Tinker', 'Baker';
      'Soldier', 'SPY'};

tf = strcmpi(s1,s2)
```

```
tf = 2x2 logical array

   1   0
   0   1
```

`tf(1,1)` is `1` because `'Tinker'` is in the first cell of both arrays. `tf(2,2)` is `1` because `'Spy'` and `'SPY'` differ only in case. `tf(2,1)` is `0` because `' Soldier'` in `s1(2,1)` has whitespace characters, and `'Soldier'` in `s2(2,1)` does not.

Compare String Arrays While Ignoring Case

Starting in R2017a, you can create strings using double quotes. Compare two string arrays, ignoring case, using `strcmpi`.

```
s1 = ["A","bc";
      "def","G"];
s2 = ["B","c";
      "DEF","G"];

tf = strcmpi(s1,s2)
```

```
tf = 2x2 logical array

   0   0
   1   1
```

Input Arguments

`s1,s2` — Input text
character vector | character array | cell array of character vectors | string array

Input text, with each input specified as a character vector, a character array, a cell array of character vectors, or a string array. The order of the inputs does not affect the comparison results.

- If both `s1` and `s2` are string arrays or cell arrays of character vectors, then `s1` and `s2` must be the same size, unless one of them is scalar.

- If both `s1` and `s2` are character arrays with multiple rows, then `s1` and `s2` can have different numbers of rows.
- When comparing a nonscalar cell array of character vectors or a string array to a multirow character array, the cell array or string array must be a column vector with the same number of rows as the character array.

Data Types: `char` | `cell` | `string`

Output Arguments

`tf` — True or false result
`1` | `0` | logical array

True or false result, returned as a `1` or `0` of data type `logical`.

- If each input is either a string scalar or a character vector, then `tf` is a scalar.
- If at least one input is either a string array or a cell array of character vectors, then `tf` is an array the same size as the input array.
- If one input is a character array with multiple rows, and the other input is either a scalar cell or a string scalar, then `tf` is an `n`-by-`1` array, where `n` is the number of rows in the character array.
- If both inputs are character arrays, `tf` is a scalar.

Tips

- The `strcmpi` function is intended for comparison of text. If used on an unsupported data type, `strcmpi` always returns `0`.
- For case-sensitive text comparison, use `strcmp` instead of `strcmpi`.
- Although `strcmpi` shares a name with a C function, it does not follow the C language convention of returning `0` when the text inputs match.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Enumeration inputs are not supported.
- Input values from the `char` class must be in the range 0-127.
- When one input is a cell array and the other input is a character array, the character array must be a compile-time row vector.
- When both inputs are empty character arrays that have different sizes, the generated code returns `true`.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- Inputs must be string arrays or cell arrays of character vectors.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`eq` | `ne` | `regexp` | `regexpi` | `sort` | `strcmp` | `strfind` | `strncmp` | `strncmpi`

Topics

"Compare Text"

Introduced before R2006a

stream2

Compute 2-D streamline data

Syntax

```
XY = stream2(x,y,u,v,startx,starty)
XY = stream2(u,v,startx,starty)
XY = stream2(...,options)
```

Description

`XY = stream2(x,y,u,v,startx,starty)` computes streamlines from vector data `u` and `v`.

The arrays `x` and `y`, which define the coordinates for `u` and `v`, must be monotonic, but do not need to be uniformly spaced. `x` and `y` must have the same number of elements, as if produced by `meshgrid`.

`startx` and `starty` define the starting positions of the streamlines.

The returned value `XY` contains a cell array of vertex arrays.

`XY = stream2(u,v,startx,starty)` assumes the arrays `x` and `y` are defined as `[x,y] = meshgrid(1:n,1:m)` where `[m,n] = size(u)`.

`XY = stream2(...,options)` specifies the options used when creating the streamlines. Define `options` as a one- or two-element vector containing the step size or the step size and the maximum number of vertices in a streamline:

```
[stepsize]
```

or

```
[stepsize, max_number_vertices]
```

If you do not specify a value, MATLAB software uses the default:

- Step size = 0.1 (one tenth of a cell)
- Maximum number of vertices = 10000

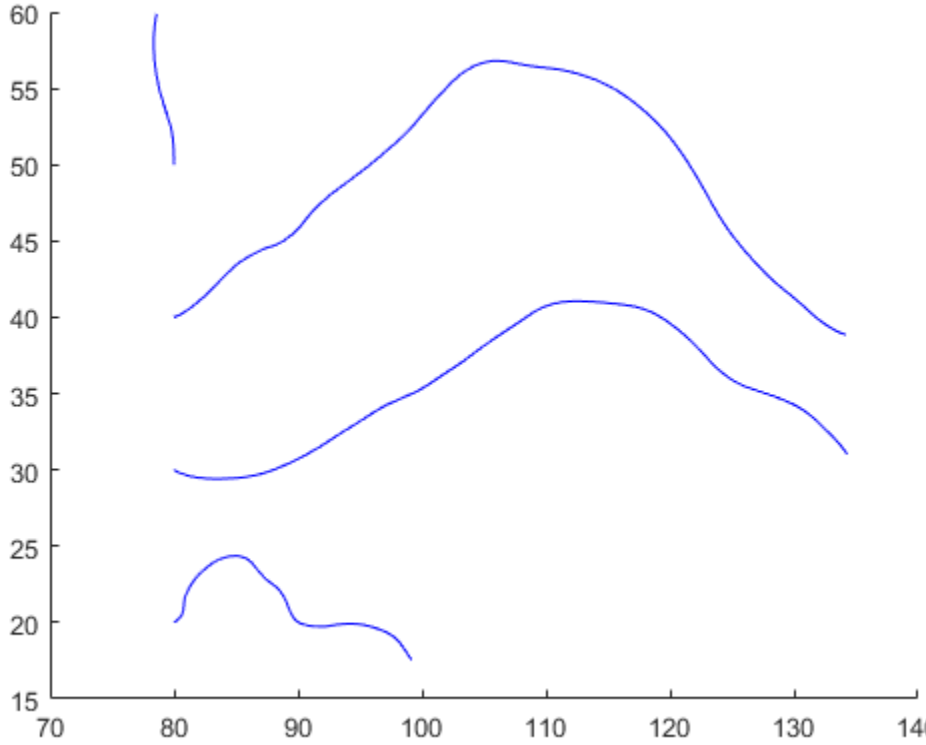
Use the `streamline` command to plot the data returned by `stream2`.

Examples

Plot 2-D Streamlines

Plot 2-D streamlines from data representing air currents over regions of North America.

```
load wind
[sx,sy] = meshgrid(80,20:10:50);
streamline(stream2(x(:,:,5),y(:,:,5),u(:,:,5),v(:,:,5),sx,sy));
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`coneplot` | `stream3` | `streamline`

Topics

“Visualizing Vector Volume Data”

Introduced before R2006a

stream3

Compute 3-D streamline data

Syntax

```
XYZ = stream3(X,Y,Z,U,V,W,startx,starty,startz)
XYZ = stream3(U,V,W,startx,starty,startz)
XYZ = stream3(...,options)
```

Description

`XYZ = stream3(X,Y,Z,U,V,W,startx,starty,startz)` computes streamlines from vector data `U`, `V`, `W`.

The arrays `X`, `Y`, and `Z`, which define the coordinates for `U`, `V`, and `W`, must be monotonic, but do not need to be uniformly spaced. `X`, `Y`, and `Z` must have the same number of elements, as if produced by `meshgrid`.

`startx`, `starty`, and `startz` define the starting positions of the streamlines.

The returned value `XYZ` contains a cell array of vertex arrays.

`XYZ = stream3(U,V,W,startx,starty,startz)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:N,1:M,1:P)` where `[M,N,P] = size(U)`.

`XYZ = stream3(...,options)` specifies the options used when creating the streamlines. Define `options` as a one- or two-element vector containing the step size or the step size and the maximum number of vertices in a streamline:

```
[stepsize]
```

or

```
[stepsize, max_number_vertices]
```

If you do not specify values, MATLAB software uses the default:

- Step size = 0.1 (one tenth of a cell)
- Maximum number of vertices = 10000

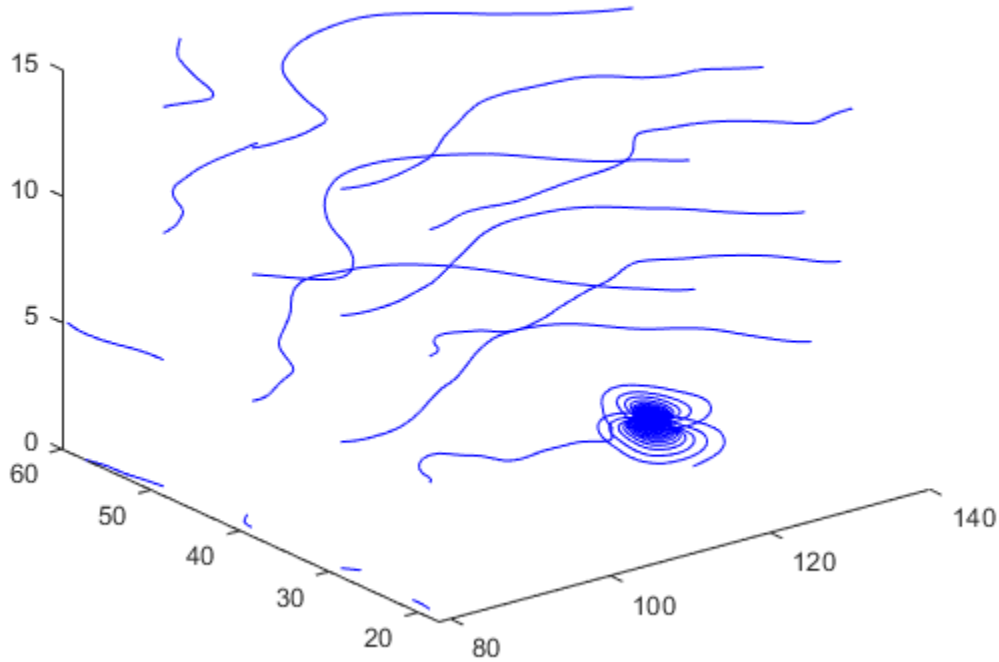
Use the `streamline` command to plot the data returned by `stream3`.

Examples

Plot 3-D Streamlines

Plot 3-D streamlines from data representing air currents over regions of North America.

```
load wind
[sx,sy,sz] = meshgrid(80,20:10:50,0:5:15);
streamline(stream3(x,y,z,u,v,w,sx,sy,sz))
view(3);
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`coneplot` | `stream2` | `streamline`

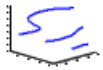
Topics

“Visualizing Vector Volume Data”

Introduced before R2006a

streamline

Plot streamlines from 2-D or 3-D vector data



Syntax

```
streamline(X,Y,Z,U,V,W,startx,starty,startz)
streamline(U,V,W,startx,starty,startz)
streamline(XYZ)
streamline(X,Y,U,V,startx,starty)
streamline(U,V,startx,starty)
streamline(XY)
streamline(...,options)
streamline(axes_handle,...)
h = streamline(...)
```

Description

`streamline(X,Y,Z,U,V,W,startx,starty,startz)` draws streamlines from 3-D vector data `U`, `V`, `W`.

The arrays `X`, `Y`, and `Z`, which define the coordinates for `U`, `V`, and `W`, must be monotonic, but do not need to be uniformly spaced. `X`, `Y`, and `Z` must have the same number of elements, as if produced by `meshgrid`.

`startx`, `starty`, `startz` define the starting positions of the streamlines.

`streamline(U,V,W,startx,starty,startz)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:N,1:M,1:P)`, where `[M,N,P] = size(U)`.

`streamline(XYZ)` assumes `XYZ` is a precomputed cell array of vertex arrays (as produced by `stream3`).

`streamline(X,Y,U,V,startx,starty)` draws streamlines from 2-D vector data `U`, `V`.

The arrays `X` and `Y`, which define the coordinates for `U` and `V`, must be monotonic, but do not need to be uniformly spaced. `X` and `Y` must have the same number of elements, as if produced by `meshgrid`.

`startx` and `starty` define the starting positions of the streamlines. The output argument `h` contains a vector of line handles, one handle for each streamline.

`streamline(U,V,startx,starty)` assumes the arrays `X` and `Y` are defined as `[X,Y] = meshgrid(1:N,1:M)`, where `[M,N] = size(U)`.

`streamline(XY)` assumes `XY` is a precomputed cell array of vertex arrays (as produced by `stream2`).

`streamline(...,options)` specifies the options used when creating the streamlines. Define `options` as a one- or two-element vector containing the step size or the step size and the maximum number of vertices in a streamline:

```
[stepsize]
```

or

```
[stepsize, max_number_vertices]
```

If you do not specify values, MATLAB uses the default:

- Step size = 0.1 (one tenth of a cell)
- Maximum number of vertices = 1000

`streamline(axes_handle,...)` plots into the axes object with the handle `axes_handle` instead of the into current axes object (`gca`).

`h = streamline(...)` returns a vector of line handles, one handle for each streamline.

Examples

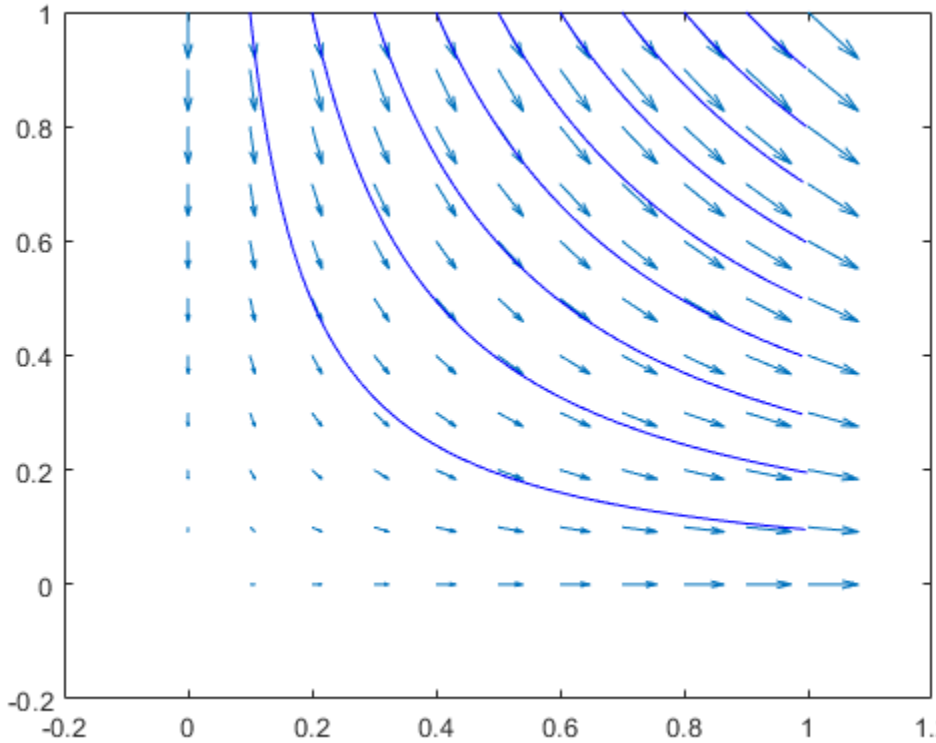
Draw Streamlines

Define arrays `x`, `y`, `u`, and `v`.

```
[x,y] = meshgrid(0:0.1:1,0:0.1:1);
u = x;
v = -y;
```

Create a quiver plot of the data. Plot streamlines that start at different points along the line $y = 1$.

```
figure
quiver(x,y,u,v)

startx = 0.1:0.1:1;
starty = ones(size(startx));
streamline(x,y,u,v,startx,starty)
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`coneplot` | `meshgrid` | `stream2` | `stream3` | `streamparticles`

Topics

“Visualizing Vector Volume Data”
“Stream Line Plots of Vector Data”

Introduced before R2006a

streamparticles

Plot stream particles

Syntax

```
streamparticles(vertices)
streamparticles(vertices,n)
streamparticles(...,'PropertyName',PropertyValue,...)
streamparticles(ax,...)
streamparticles(line_handle,...)
h = streamparticles(...)
```

Description

`streamparticles(vertices)` draws stream particles of a vector field. Stream particles are usually represented by markers and can show the position and velocity of a streamline. `vertices` is a cell array of 2-D or 3-D vertices (as if produced by `stream2` or `stream3`).

`streamparticles(vertices,n)` uses `n` to determine how many stream particles to draw. The `ParticleAlignment` property controls how `n` is interpreted.

- If `ParticleAlignment` is set to `off` (the default) and `n` is greater than `1`, approximately `n` particles are drawn evenly spaced over the streamline vertices.

  If `n` is less than or equal to `1`, `n` is interpreted as a fraction of the original stream vertices; for example, if `n` is `0.2`, approximately 20% of the vertices are used.

  `n` determines the upper bound for the number of particles drawn. The actual number of particles can deviate from `n` by as much as a factor of 2.
- If `ParticleAlignment` is `on`, `n` determines the number of particles on the streamline having the most vertices and sets the spacing on the other streamlines to this value. The default value is `n = 1`.

`streamparticles(...,'PropertyName',PropertyValue,...)` controls the stream particles using named properties and specified values. Any unspecified properties have default values. MATLAB ignores the case of property names.

`streamparticles(ax,...)` creates the stream particles in the axes specified by `ax` instead of in the current axes (`gca`). The option `ax` can precede any of the input argument combinations in the previous syntaxes.

Stream Particle Properties

`Animate` — Stream particle motion [nonnegative integer]

The number of times to animate the stream particles. The default is `0`, which does not animate. `Inf` animates until you enter **Ctrl+C**.

`FrameRate` — Animation frames per second [nonnegative integer]

This property specifies the number of frames per second for the animation. `Inf`, the default, draws the animation as fast as possible. Note that the speed of the animation might be limited by the speed of the computer. In such cases, the value of `FrameRate` cannot necessarily be achieved.

`ParticleAlignment` — Align particles with streamlines [`on` | `{off}`]

Set this property to `on` to draw particles at the beginning of each streamline. This property controls how `streamparticles` interprets the argument `n` (number of stream particles).

Stream particles are primitive line objects. In addition to stream particle properties, you can specify any line property, such as `Marker`. `streamparticles` sets the following line properties when called.

Line Property	Value Set by streamparticles
`LineStyle`	`'none'`
`Marker`	`'o'`
`MarkerEdgeColor`	`'none'`
`MarkerFaceColor`	`'red'`

You can override any of these properties by specifying a property name and value as arguments to `streamparticles`. For example, this statement uses RGB values to set the `MarkerFaceColor` to medium gray:

```
streamparticles(vertices,'MarkerFaceColor',[.5 .5 .5])
```

`streamparticles(line_handle,...)` uses the line object identified by `line_handle` to draw the stream particles.

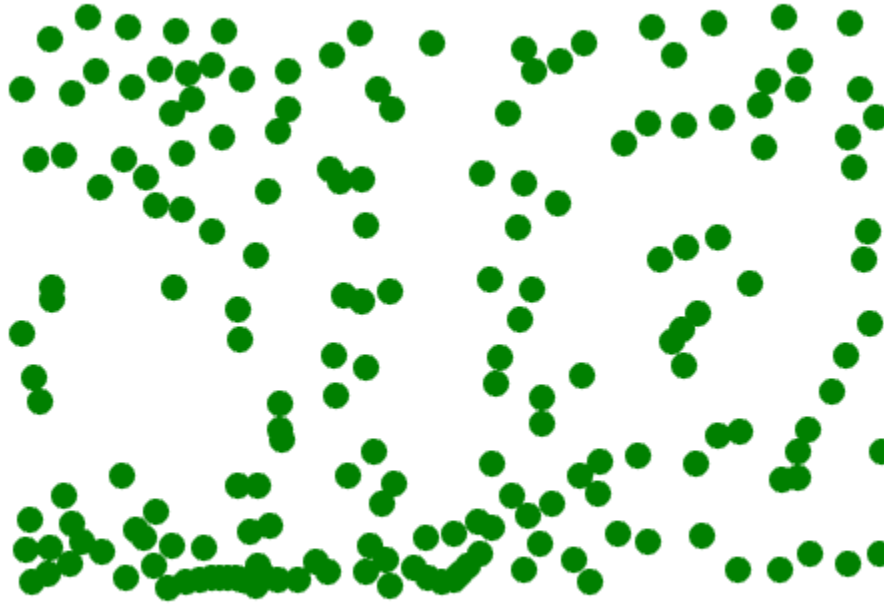
`h = streamparticles(...)` returns a vector of handles to the primitive line objects it creates. For a list of properties, see Primitive Line.

Examples

Animate Flow Without Displaying Streamlines

This example uses streamlines in the `z = 5` plane to animate the flow along these lines with stream particles.

```
load wind
figure
daspect([1,1,1]);
view(2)
[verts,averts] = streamslice(x,y,z,u,v,w,[],[],[5]);
sl = streamline([verts averts]);
axis tight manual off;
set(sl,'Visible','off')
iverts = interpstreamspeed(x,y,z,u,v,w,verts,.05);
zlim([4.9,5.1]);
streamparticles(iverts, 200, ...
    'Animate',15,'FrameRate',40, ...
    'MarkerSize',10,'MarkerFaceColor',[0 .5 0])
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`interpstreamspeed` | `stream2` | `stream3` | `streamline`

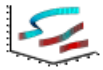
Topics

“Creating Stream Particle Animations”
“Visualizing Vector Volume Data”

Introduced before R2006a

streamribbon

3-D stream ribbon plot from vector volume data



Syntax

```
streamribbon(X,Y,Z,U,V,W,startx,starty,startz)
streamribbon(U,V,W,startx,starty,startz)
streamribbon(vertices,X,Y,Z,cav,speed)
streamribbon(vertices,cav,speed)
streamribbon(vertices,twistangle)
streamribbon(...,width)
streamribbon(axes_handle,...)
h = streamribbon(...)
```

Description

`streamribbon(X,Y,Z,U,V,W,startx,starty,startz)` draws stream ribbons from vector volume data `U`, `V`, `W`.

The arrays `X`, `Y`, and `Z`, which define the coordinates for `U`, `V`, and `W`, must be monotonic, but do not need to be uniformly spaced. `X`, `Y`, and `Z` must have the same number of elements, as if produced by `meshgrid`.

`startx`, `starty`, and `startz` define the starting positions of the stream ribbons at the center of the ribbons.

The twist of the ribbons is proportional to the curl of the vector field. The width of the ribbons is calculated automatically.

`streamribbon(U,V,W,startx,starty,startz)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`streamribbon(vertices,X,Y,Z,cav,speed)` assumes precomputed streamline vertices, curl angular velocity, and flow speed. `vertices` is a cell array of streamline vertices (as produced by `stream3`). `X`, `Y`, `Z`, `cav`, and `speed` are 3-D arrays.

`streamribbon(vertices,cav,speed)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(cav)`.

`streamribbon(vertices,twistangle)` uses the cell array of vectors `twistangle` for the twist of the ribbons (in radians). The size of each corresponding element of `vertices` and `twistangle` must be equal.

`streamribbon(...,width)` sets the width of the ribbons to `width`.

`streamribbon(axes_handle,...)` plots into the axes object with the handle `axes_handle` instead of into the current axes object (`gca`).

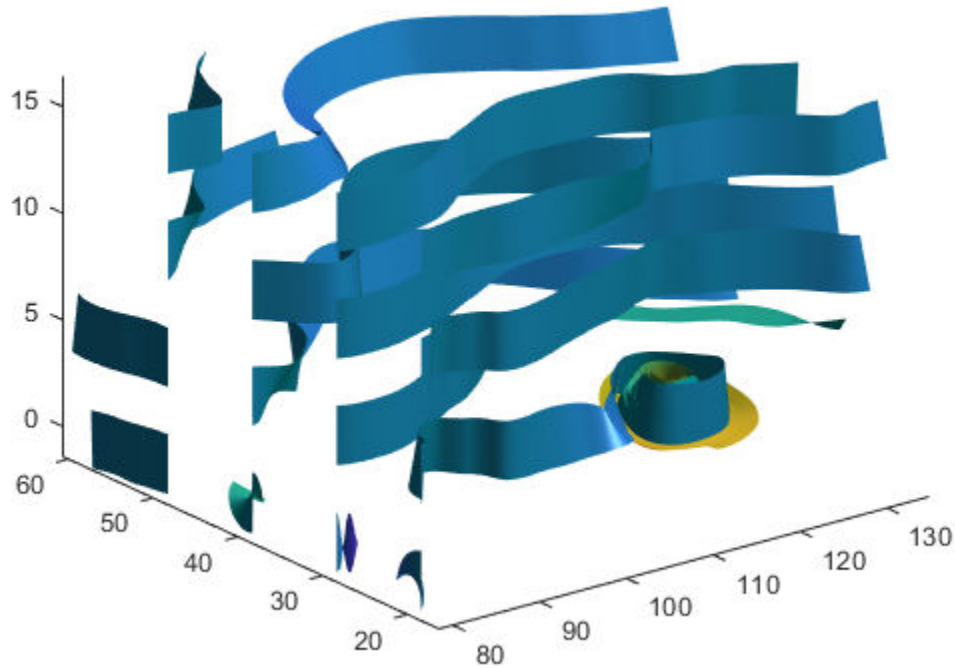
`h = streamribbon(...)` returns a vector of handles (one per start point) to `surface` objects.

Examples

Use Stream Ribbons to Indicate Flow

Use stream ribbons to indicate the flow in a data set.

```
load wind
[sx,sy,sz] = meshgrid(80,20:10:50,0:5:15);
streamribbon(x,y,z,u,v,w,sx,sy,sz);
axis tight
shading interp
view(3);
camlight
lighting gouraud
```

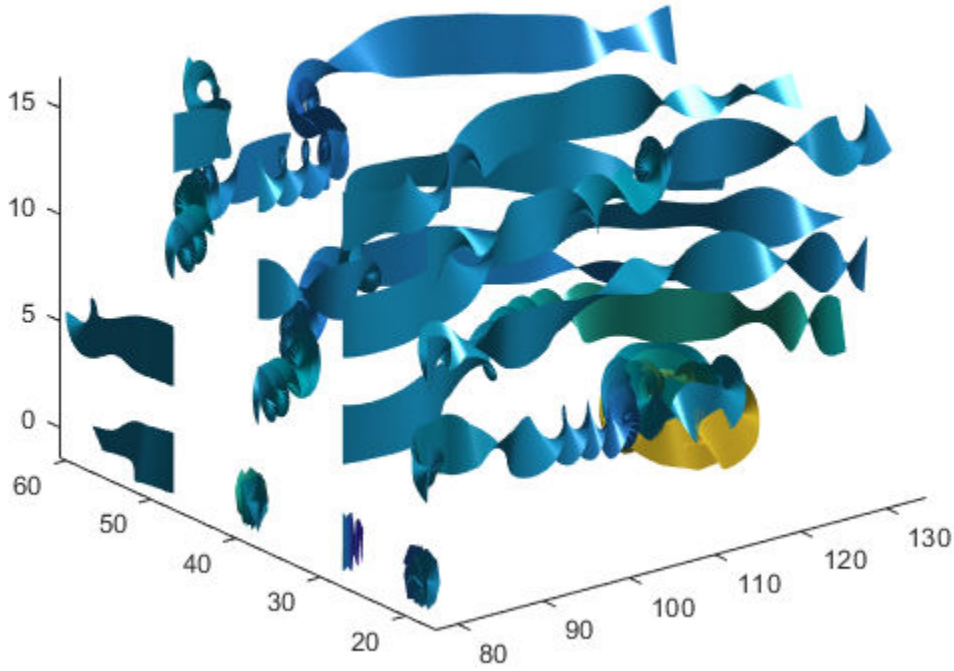


Indicate Flow Using Precalculated Data

Use precalculated vertex data, curl average velocity, and speed to indicate flow.

```
load wind
[sx,sy,sz] = meshgrid(80,20:10:50,0:5:15);
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
cav = curl(x,y,z,u,v,w);
spd = sqrt(u.^2 + v.^2 + w.^2).*.1;
streamribbon(verts,x,y,z,cav,spd);
axis tight
```

```
shading interp
view(3);
camlight;
lighting gouraud
```

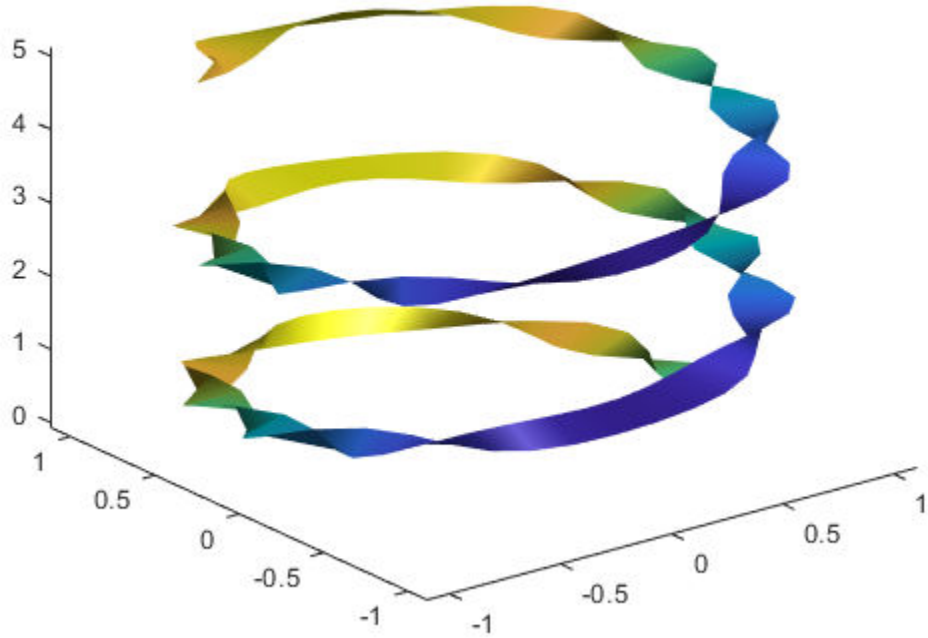


Using precalculated data enables you to use values other than those calculated from the single data source. In this case, the speed is reduced by a factor of 10 compared to the previous example.

Stream Ribbon With Twist Angle

Specify a twist angle for the stream ribbon

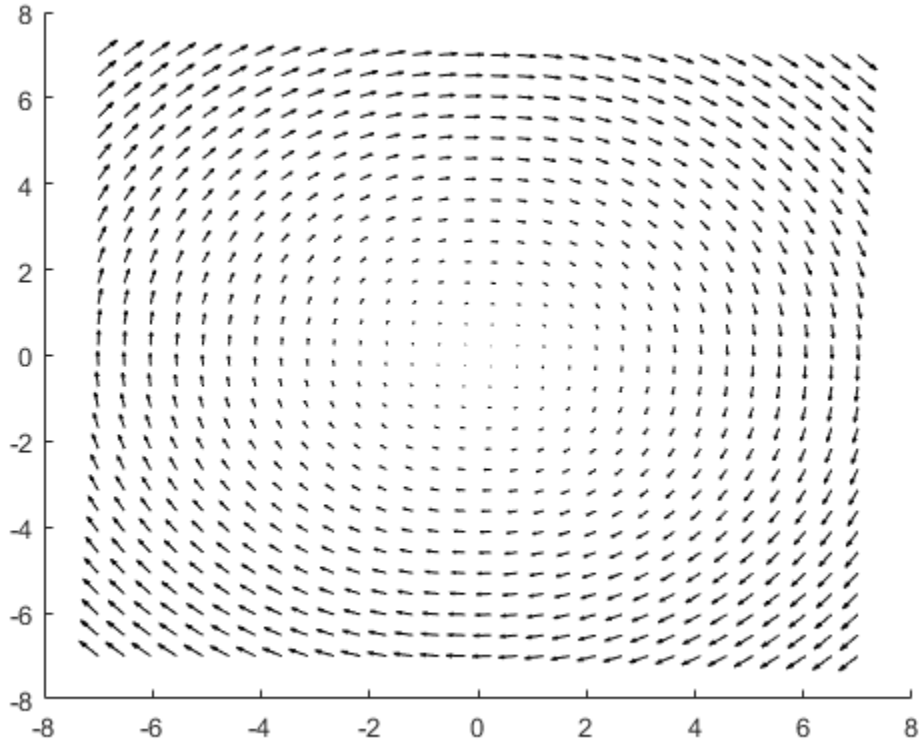
```
t = 0:.15:15;
verts = {[cos(t)' sin(t)' (t/3)']};
twistangle = {cos(t)'};
streamribbon(verts,twistangle);
axis tight
shading interp
view(3)
camlight
lighting gouraud
```



Combination of Cone and Stream Ribbon Plots

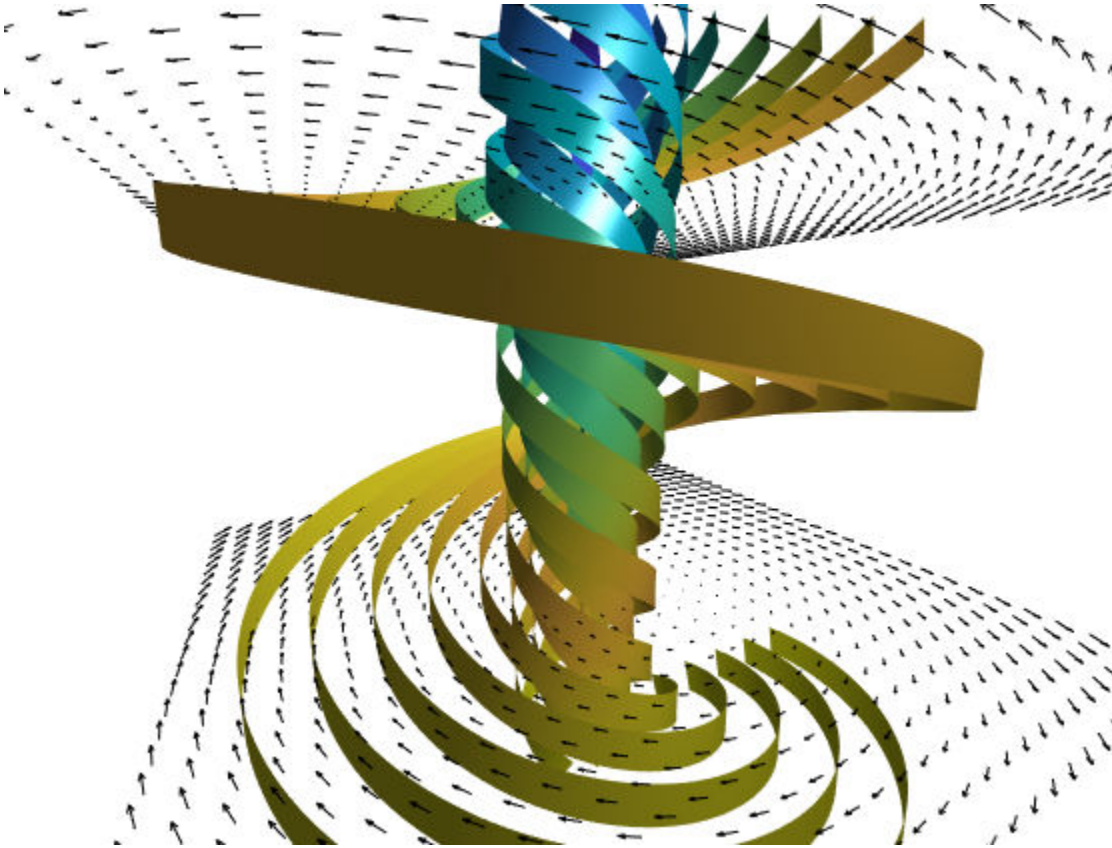
Create 3-D arrays and a cone plot.

```
xmin = -7; xmax = 7;
ymin = -7; ymax = 7;
zmin = -7; zmax = 7;
x = linspace(xmin,xmax,30);
y = linspace(ymin,ymax,20);
z = linspace(zmin,zmax,20);
[x,y,z] = meshgrid(x,y,z);
u = y;
v = -x;
w = 0*x+1;
[cx,cy,cz] = meshgrid(linspace(xmin,xmax,30),...
   linspace(ymin,ymax,30),[-3 4]);
h = coneplot(x,y,z,u,v,w,cx,cy,cz,'quiver');
set(h,'Color','k');
```



Plot two sets of stream ribbons. Then define the view and lighting.

```
[sx,sy,sz] = meshgrid([-1 0 1],[-1 0 1],-6);
streamribbon(x,y,z,u,v,w,sx,sy,sz);
[sx,sy,sz] = meshgrid([1:6],[0],-6);
streamribbon(x,y,z,u,v,w,sx,sy,sz);
shading interp
view(-30,10)
axis off tight
camproj perspective
camva(66)
camlookat
camdolly(0,0,.5,'fixtarget')
camlight
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`coneplot` | `curl` | `meshgrid` | `stream3` | `streamline` | `streamtube`

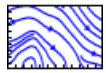
Topics

“Volume Visualization”
“Displaying Curl with Stream Ribbons”
“Visualizing Vector Volume Data”

Introduced before R2006a

streamslice

Plot streamlines in slice planes



Syntax

```
streamslice(X,Y,Z,U,V,W,startx,starty,startz)
streamslice(U,V,W,startx,starty,startz)
streamslice(X,Y,U,V)
streamslice(U,V)
streamslice(...,density)
streamslice(...,'arrowsmode')
streamslice(...,'method')
streamslice(axes_handle,...)
h = streamslice(...)
[vertices arrowvertices] = streamslice(...)
```

Description

`streamslice(X,Y,Z,U,V,W,startx,starty,startz)` draws well-spaced streamlines (with direction arrows) from vector data `U`, `V`, `W` in axis aligned *x*-, *y*-, *z*-planes at the points in the vectors `startx`, `starty`, `startz`.

The arrays `X`, `Y`, and `Z`, which define the coordinates for `U`, `V`, and `W`, must be monotonic, but do not need to be uniformly spaced. `X`, `Y`, and `Z` must have the same number of elements, as if produced by `meshgrid`. `U`, `V`, `W` must be m-by-n-by-p volume arrays.

Do not assume that the flow is parallel to the slice plane. For example, in a stream slice at a constant `z`, the `z` component of the vector field `W` is ignored when you are calculating the streamlines for that plane.

Stream slices are useful for determining where to start streamlines, stream tubes, and stream ribbons.

`streamslice(U,V,W,startx,starty,startz)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`streamslice(X,Y,U,V)` draws well-spaced streamlines (with direction arrows) from vector volume data `U`, `V`.

The arrays `X` and `Y`, which define the coordinates for `U` and `V`, must be monotonic, but do not need to be uniformly spaced. `X` and `Y` must have the same number of elements, as if produced by `meshgrid`.

`streamslice(U,V)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`streamslice(...,density)` modifies the automatic spacing of the streamlines. `density` must be greater than `0`. The default value is `1`; higher values produce more streamlines on each plane. For example, `2` produces approximately twice as many streamlines, while `0.5` produces approximately half as many.

`streamslice(...,'arrowsmode')` determines if direction arrows are present or not. `arrowmode` can be

- `arrows` — Draw direction arrows on the streamlines (default).
- `noarrows` — Do not draw direction arrows.

`streamslice(...,'method')` specifies the interpolation method to use. `method` can be

- `linear` — Linear interpolation (default)
- `cubic` — Cubic interpolation
- `nearest` — Nearest-neighbor interpolation

See `interp3` for more information on interpolation methods.

`streamslice(axes_handle,...)` plots into the axes object with the handle `axes_handle` instead of into the current axes object (`gca`).

`h = streamslice(...)` returns a vector of handles to the line objects created.

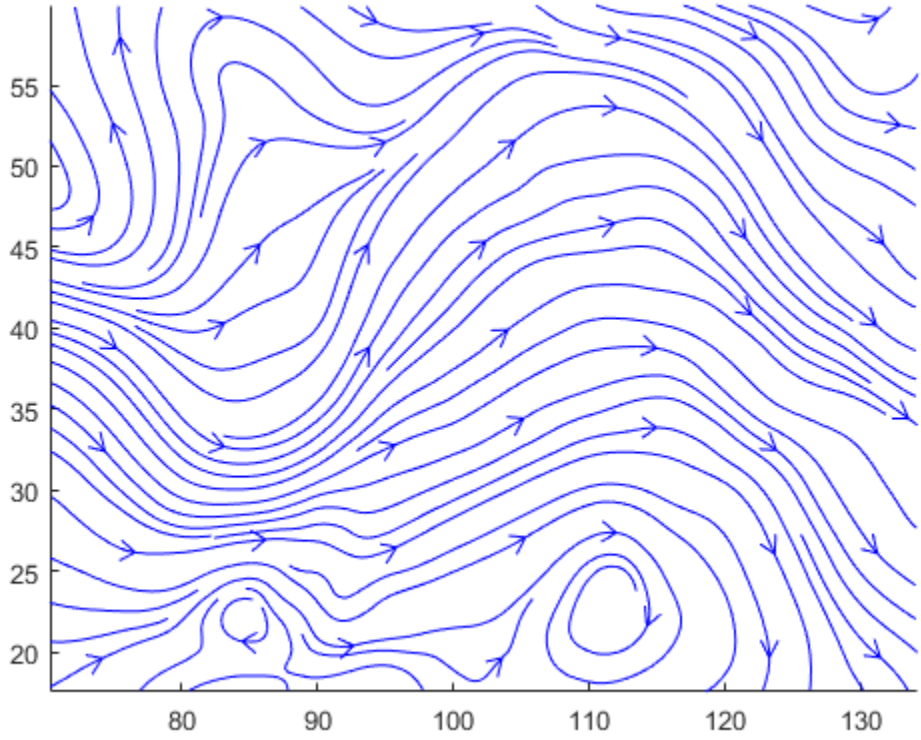
`[vertices arrowvertices] = streamslice(...)` returns two cell arrays of vertices for drawing the streamlines and the arrows. You can pass these values to any of the streamline drawing functions (`streamline`, `streamribbon`, `streamtube`).

Examples

Plot Streamlines in Slice Plane

Load the `wind` data set, which loads the three-dimensional arrays `u`, `v`, `w`, `x`, `y`, and `z`. Draw streamlines along the slice plane where `z = 5`.

```
load wind
streamslice(x,y,z,u,v,w,[],[],5)
axis tight
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`contourslice` | `interp2` | `interp3` | `meshgrid` | `slice` | `streamline` | `streamribbon` | `streamtube` | `volumebounds`

Topics

“Visualizing Vector Volume Data”

Introduced before R2006a

streamtube

Create 3-D stream tube plot

Syntax

```
streamtube(X,Y,Z,U,V,W,startx,starty,startz)
streamtube(U,V,W,startx,starty,startz)
streamtube(vertices,X,Y,Z,divergence)
streamtube(vertices,divergence)
streamtube(vertices,width)
streamtube(vertices)
streamtube(...,[scale n])
streamtube(ax,...)
h = streamtube(...)
```

Description

`streamtube(X,Y,Z,U,V,W,startx,starty,startz)` draws stream tubes from vector volume data `U`, `V`, `W`.

The arrays `X`, `Y`, and `Z`, which define the coordinates for `U`, `V`, and `W`, must be monotonic, but do not need to be uniformly spaced. `X`, `Y`, and `Z` must have the same number of elements, as if produced by `meshgrid`.

`startx`, `starty`, and `startz` define the starting positions of the streamlines at the center of the tubes.

The width of the tubes is proportional to the normalized divergence of the vector field.

`streamtube(U,V,W,startx,starty,startz)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`streamtube(vertices,X,Y,Z,divergence)` assumes precomputed streamline vertices and divergence. `vertices` is a cell array of streamline vertices (as produced by `stream3`). `X`, `Y`, `Z`, and `divergence` are 3-D arrays.

`streamtube(vertices,divergence)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(divergence)`.

`streamtube(vertices,width)` specifies the width of the tubes in the cell array of vectors, `width`. The size of each corresponding element of `vertices` and `width` must be equal. `width` can also be a scalar, specifying a single value for the width of all stream tubes.

`streamtube(vertices)` selects the width automatically.

`streamtube(...,[scale n])` scales the width of the tubes by `scale`. The default is `scale = 1`. When the stream tubes are created, using start points or divergence, specifying `scale = 0` suppresses automatic scaling. `n` is the number of points along the circumference of the tube. The default is `n = 20`.

`streamtube(ax,...)` plots into the axes object `ax` instead of into the current axes object (`gca`).

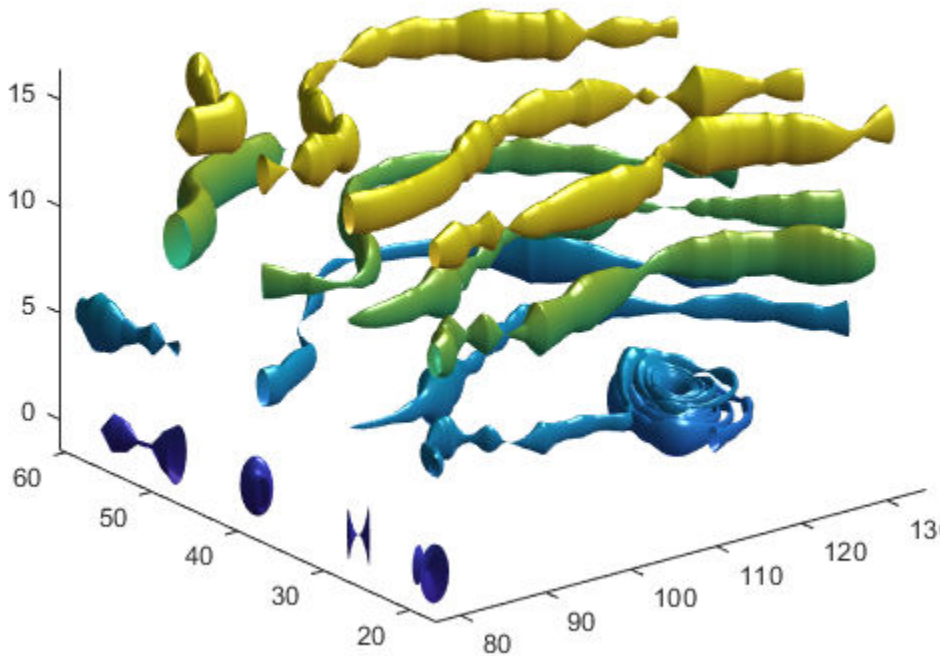
`h = streamtube(...)` returns a vector of `surface` objects (one per start point) used to draw the stream tubes.

Examples

Visualize flow

Use the `streamtube` function to indicate flow in the `wind` data set. The inputs include the coordinates, vector field components, and starting location for the stream tubes.

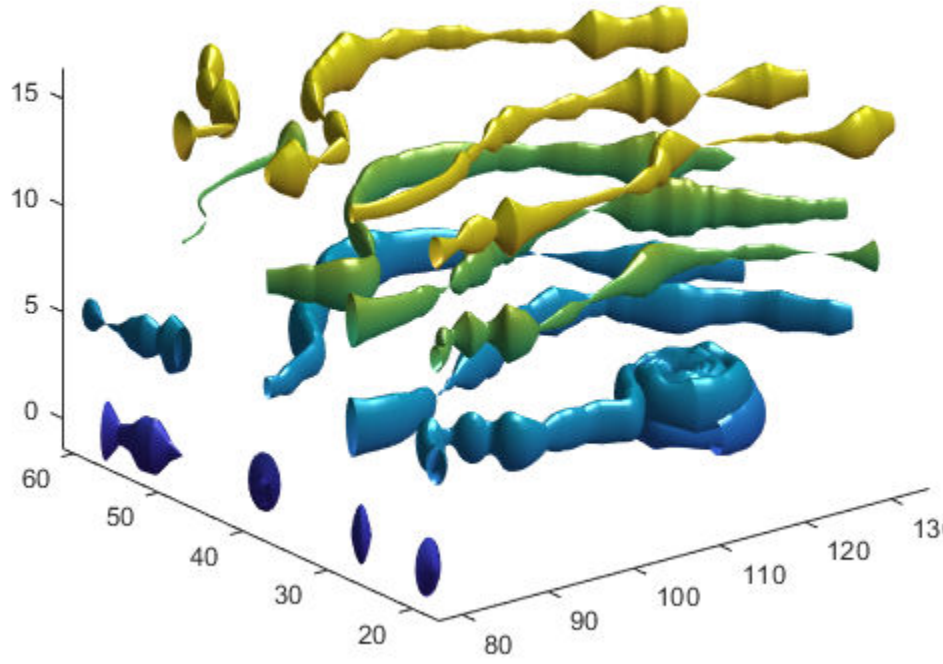
```
load wind
[sx,sy,sz] = meshgrid(80,20:10:50,0:5:15);
streamtube(x,y,z,u,v,w,sx,sy,sz);
view(3);
axis tight
shading interp;
camlight;
lighting gouraud
```



Visualize Flow Using Vertex Data and Divergence

Use vertex data returned by the `stream3` function and divergence data to visualize flow.

```
load wind
[sx,sy,sz] = meshgrid(80,20:10:50,0:5:15);
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
div = divergence(x,y,z,u,v,w);
streamtube(verts,x,y,z,-div);
view(3);
axis tight
shading interp
camlight
lighting gouraud
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`divergence` | `meshgrid` | `stream3` | `stream3` | `streamline` | `streamribbon`

Topics

“Displaying Divergence with Stream Tubes”
“Visualizing Vector Volume Data”

Introduced before R2006a

strfind

Find strings within other strings

Syntax

```
k = strfind(str,pattern)
k = strfind(str,pattern,'ForceCellOutput',cellOutput)
```

Description

`k = strfind(str,pattern)` searches `str` for occurrences of `pattern`. The output, `k`, indicates the starting index of each occurrence of `pattern` in `str`. If `pattern` is not found, then `strfind` returns an empty array, `[]`. The `strfind` function executes a case-sensitive search.

- If `str` is a character vector or a string scalar, then `strfind` returns a vector of type `double`.
- If `str` is a cell array of character vectors or a string array, then `strfind` returns a cell array of vectors of type `double`.

`k = strfind(str,pattern,'ForceCellOutput',cellOutput)` forces `strfind` to return `k` as a cell array when `cellOutput` is true, even when `str` is a character vector.

Examples

Find Pattern in Character Vector

Find the starting indices for occurrences of patterns in a character vector.

First, create a character vector.

```
str = 'Find the starting indices of a pattern in a character vector';
```

Find the pattern `in`.

```
k = strfind(str,'in')
```

```
k = 1×4

     2    15    19    40
```

There are four instances of the pattern in `str`.

Find the pattern `In`.

```
k = strfind(str,'In')
```

```
k =

     []
```

Since `strfind` is case sensitive, the pattern is not found. `k` is an empty array.

Find the blank spaces in `str`.

```
k = strfind(str,' ')
```

```
k = 1×10

     5     9    18    26    29    31    39    42    44    54
```

There are ten blank spaces in `str`.

Find Pattern in Cell Array

Find the starting indices for occurrences of a pattern in a cell array of character vectors.

Create a cell array of character vectors.

```
str = {'How much wood would a woodchuck chuck';
       'if a woodchuck could chuck wood?'};
```

Find the pattern `wood` in `str`.

```
idx = strfind(str,'wood')
```

```
idx=2×1 cell
    {1x2 double}
    {1x2 double}
```

Examine the output cell array to find the instances of the pattern.

```
idx{:,:}
```

```
ans = 1×2

    10    23


ans = 1×2

     6    28
```

The pattern `wood` occurs at indices 10 and 23 in the first character vector and at indices 6 and 28 in the second character vector.

Return Indices in Cell Array

Find the occurrences of a pattern in a character vector. Force `strfind` to return the indices of those occurrences in a cell array. Then display the indices.

Create a character vector and find the occurrences of the pattern `ain`.

```
str = 'The rain in Spain.';
k = strfind(str,'ain','ForceCellOutput',true)
```

```
k = 1x1 cell array
    {1x2 double}
```

`strfind` returns a scalar cell that contains a numeric array, which contains indices of occurrences of the pattern `ain` in `str`. To access the numeric array within the cell, use curly braces.

```
k{1}
```

```
ans = 1×2

     6    15
```

Input Arguments

str — Data to be searched
character vector | cell array of character vectors | string array

Data to be searched, specified as a character vector, a cell array of character vectors, or a string array.

Data Types: `char` | `cell` | `string`

pattern — Search pattern
character vector | string scalar

Search pattern, specified as a character vector or a string scalar.

Data Types: `char` | `string`

cellOutput — Indicator for forcing output to be returned as cell array
`false` (default) | `true` | `0` | `1`

Indicator for forcing output to be returned as a cell array, specified as `false`, `true`, `0`, or `1`.

Data Types: `logical`

Output Arguments

k — Indices of occurrences of pattern
array

Indices of occurrences of `pattern`, returned as an array. If `pattern` is not found, then `k` is an empty array, `[]`.

- If `str` is a character vector or a string scalar, `k` is a vector of doubles indicating the index of each occurrence of `pattern`.

- If `str` is a cell array of character vectors or a string array, `k` is a cell array. For each piece of text in `str`, the corresponding cell of `k` contains a vector of doubles indicating the index of each occurrence of `pattern`.

Tips

- If `pattern` is a character vector or string scalar with no characters (`''` or `""`), then `strfind` returns an empty array.
- Starting in R2016b, the `contains` function is recommended for finding patterns within string arrays.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

- The text input must be a tall array of strings or a tall cell array of character vectors.
- The text pattern must be a non-tall single string.
- The output is a cell array of index vectors, with one element per input string.

For more information, see “Tall Arrays for Out-of-Memory Data”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `str` must be a string scalar or a character vector.
- Generated code returns an empty output as a 1-by-0 character array.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- `str` must be a string array or a cell array of character vectors.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`contains` | `regexp` | `regexpi` | `regexprep` | `strcmp` | `strcmpi` | `strncmp` | `strncmpi` | `strrep` | `strsplit` | `strtok`

Introduced before R2006a

string

String array

Description

Starting in R2016b, you can represent text using *string arrays* instead of character arrays. Each element of a string array stores a sequence of characters. The sequences can have different lengths without padding, such as `"yes"` and `"no"`. A string array that has only one element is also called a *string scalar*.

You can index into, reshape, and concatenate string arrays using standard array operations, and you can append text to strings using the + operator. If a string array represents numbers, then you can convert it to a numeric array using the `double` function.

Creation

You can create a string by enclosing a piece of text in double quotes. Double quotes were introduced in R2017a.

```
str = "Hello, world"
```

```
str =
"Hello, world"
```

One way to create a string array is to concatenate strings into an array using square brackets, just as you can concatenate numbers into a numeric array.

```
str = ["Mercury","Gemini","Apollo";
       "Skylab","Skylab B","ISS"]
```

```
str = 2x3 string array
    "Mercury"    "Gemini"      "Apollo"
    "Skylab"     "Skylab B"    "ISS"
```

You also can convert variables of different data types into string arrays using the `string` function, described below.

Syntax

```
str = string(A)
```

```
str = string(D)
str = string(D,fmt)
str = string(D,fmt,locale)
```

Description

`str = string(A)` converts the input array to a string array.

`str = string(D)` converts a datetime, duration, or calendar duration array into a string array in the format specified by the `Format` property of `D`. The output contains one date or duration in each row.

`str = string(D,fmt)` represents dates or durations in the specified format, such as `'HH:mm:ss'`.

`str = string(D,fmt,locale)` represents dates or durations in the specified locale, such as `'en_US'`. The locale affects the language used to represent strings such as month and day names.

Input Arguments

`A` — Input array
array

Input array. The data type of `A` determines how `string` converts `A` to a string array.

- If `A` is a character vector, then `string` converts `A` to a string scalar. The output `str` and input `A` have the same characters in the same order.
- If `A` is a cell array or a categorical array, then `string` converts each element in `A` to a string element in `str`.
- If `A` is a numeric array, then `string` converts each number to a string element in `str`.

  Unlike the `char` function, `string` does not treat numbers as ASCII or Unicode code points.
- If `A` is a logical array, then `string` converts each value to either `"false"` or `"true"`.

- If `A` is `[]`, then `string` returns a `0`-by-`0` string array.

`D` — Input date and time
datetime or duration array

Input date and time, specified as a datetime or duration array.

Data Types: `datetime` | `duration` | `calendarDuration`

`fmt` — Date and time format
`[]` (default) | character vector | string scalar

Date and time format, specified as `[]`, a character vector, or a string scalar. If you specify `[]`, then `string` represents input `D` in the format specified by the `Format` property of `D`.

The supported formats depend on the data type of `D`.

- `datetime` formats can include combinations of units and delimiters, such as `'yyyy-MMM-dd HH:mm:ss.SSS'`. For details, see the `Format` property for datetime arrays.
- `duration` formats are either single characters (`'y'`, `'d'`, `'h'`, `'m'`, or `'s'`) or one of these combinations:
  - `'dd:hh:mm:ss'`
  - `'hh:mm:ss'`
  - `'mm:ss'`
  - `'hh:mm'`
  - Any of the above, with up to nine `S` characters to indicate fractional second digits, such as `'hh:mm:ss.SSSS'`
- `calendarDuration` formats can include combinations of the characters `'y'`, `'q'`, `'m'`, `'w'`, `'d'`, and `'t'` in order from largest to smallest unit of time, such as `'ym'`.

For more information on the `duration` and `calendarDuration` formats, see “Set Date and Time Display Format”.

`locale` — Locale represented in output
character vector | string scalar

Locale represented in the output, specified as a character vector or a string scalar. The locale affects the language used to represent certain components of dates and times, such as month names.

`locale` can be:

- `'system'`, to specify your system locale.
- A character vector in the form *xx_YY*, where *xx* is a lowercase ISO 639-1 two-letter code that specifies a language, and *YY* is an uppercase ISO 3166-1 alpha-2 code that specifies a country.

The `locale` input argument can be any of the values accepted by the `'Locale'` name-value pair argument for the `datetime` function.

Example: `'en_US'`

Example: `'ja_JP'`

Output Arguments

`str` — Output array
string array

Output array, returned as a string array.

MATLAB stores all characters as Unicode characters using the UTF-16 encoding. For more information on Unicode, see Unicode.

Examples

Split String and Find Unique Words

To find the unique words in a string, split it on space characters and call the `unique` function.

First, create a string scalar.

```
str = "A horse! A horse! My kingdom for a horse!"
```

```
str =
"A horse! A horse! My kingdom for a horse!"
```

Remove the exclamation point.

```
str = erase(str,"!")
```

```
str =
"A horse A horse My kingdom for a horse"
```

Convert all letters in `str` to lowercase characters.

```
str = lower(str)
```

```
str =
"a horse a horse my kingdom for a horse"
```

Split `str` on space characters using the `split` function. `split` discards the space characters and returns the result as a string array.

```
str = split(str)
```

```
str = 9x1 string array
    "a"
    "horse"
    "a"
    "horse"
    "my"
    "kingdom"
    "for"
    "a"
    "horse"
```

Find the unique words in `str` using the `unique` function.

```
str = unique(str)
```

```
str = 5x1 string array
    "a"
    "for"
    "horse"
    "kingdom"
    "my"
```

Convert Character Vector

```
A = 'Four score and seven years ago'
```

```
A =
'Four score and seven years ago'
```

```
str = string(A)
```

```
str =
"Four score and seven years ago"
```

`str` contains the same characters as `A`. But while `A` is a character vector, `str` is a string scalar.

```
c = size(A)
```

```
c = 1×2

     1    30
```

```
s = size(str)
```

```
s = 1×2

     1     1
```

To return the number of characters in `str`, use the `strlength` function.

```
n = strlength(str)
```

```
n = 30
```

Convert Cell Array

Convert a cell array of character vectors to a string array.

```
A = {'Mercury','Gemini','Apollo';...
     'Skylab','Skylab B','ISS'}
```

```
A = 2x3 cell array
    {'Mercury'}    {'Gemini'  }    {'Apollo'}
    {'Skylab' }    {'Skylab B'}    {'ISS'   }
```

```
str = string(A)
```

```
str = 2x3 string array
    "Mercury"    "Gemini"      "Apollo"
    "Skylab"     "Skylab B"    "ISS"
```

To access the second element in the first row of `str`, index using smooth parentheses. You can access strings in a string array with matrix indexing, just as you would access elements of a numeric array.

```
str(1,2)
```

```
ans =
"Gemini"
```

Access the third column.

```
str(:,3)
```

```
ans = 2x1 string array
    "Apollo"
    "ISS"
```

Convert Numeric Array

```
A = [77 65 84 76 65 66]
```

```
A = 1×6

    77    65    84    76    65    66
```

```
str = string(A)
```

```
str = 1x6 string array
    "77"    "65"    "84"    "76"    "65"    "66"
```

`str` is a string array in which each element represents a number from `A`. Note that `string` does not treat numbers as ASCII or Unicode® values the way that the `char` function does.

Convert Strings That Represent Numbers

Create a string array in which each element represents a number. To convert the string array to a numeric array, use the `double` function.

```
str = ["256","3.1416","8.9e-3"]
```

```
str = 1x3 string array
    "256"    "3.1416"    "8.9e-3"
```

```
X = double(str)
```

```
X = 1×3

  256.0000    3.1416    0.0089
```

When the input argument is a string array, the `double` function treats each element as the representation of a floating-point value. However, when the input is a character array, `double` instead converts each character to a number representing its Unicode® value.

As an alternative, use the `str2double` function. `str2double` is suitable when the input argument might be a string array, character vector, or cell array of character vectors.

```
Y = str2double(str)
```

```
Y = 1×3

  256.0000    3.1416    0.0089
```

```
C = '2.7183';
Z = str2double(C)
```

```
Z = 2.7183
```

Convert Duration Array

Create a `duration` array.

```
D = hours(23:25) + minutes(8) + seconds(1.2345)
```

```
D = 1x3 duration array
   23.134 hr   24.134 hr   25.134 hr
```

Convert `D` to a string array.

```
str = string(D)
```

```
str = 1x3 string array
    "23.134 hr"    "24.134 hr"    "25.134 hr"
```

`str` is a string array with one duration value per element. `str` is the same size as `D`.

Specify the format of the duration values in `str`.

```
str = string(D,'hh:mm')
```

```
str = 1x3 string array
    "23:08"    "24:08"    "25:08"
```

Tips

- For a list of functions to create and manipulate text in string arrays, see “Characters and Strings”.
- If the input argument is an object, then it must belong to a class that implements a `string` method to represent the object as a string.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- A cell array input must be scalar at compile time.
- An enumeration input must be scalar at compile time. Arrays of enumerations are not supported.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cellstr` | `char` | `compose` | `isstring` | `isstrprop` | `sprintf` | `strings` | `strlength`

Topics

“Create String Arrays”
“Analyze Text Data with String Arrays”
“Frequently Asked Questions About String Arrays”
“Locale Settings for MATLAB Process”

Introduced in R2016b

strings

Create string array with no characters

Syntax

```
str = strings
str = strings(n)
str = strings(sz1,...,szN)
str = strings(sz)
```

Description

`str = strings` returns a string with no characters. For more information on string arrays, see `string`.

Note: Starting in R2017a, you also can use double quotes. For example, `str = ""` creates a string scalar that contains no characters.

`str = strings(n)` returns an `n`-by-`n` string array. Each element is a string with no characters.

`str = strings(sz1,...,szN)` returns a `sz1`-by-...-by-`szN` string array, where `sz1,...,szN` indicate the size of each dimension. For example, `strings(2,3)` returns a 2-by-3 string array where each element is `""`.

`str = strings(sz)` returns a string array where size vector `sz` defines `size(str)`. For example, `strings([2,3])` returns a 2-by-3 string array.

Examples

Square Array of Empty Strings

```
str = strings(4)
```

```
str = 4x4 string array
    ""    ""    ""    ""
    ""    ""    ""    ""
    ""    ""    ""    ""
    ""    ""    ""    ""
```

Rectangular Array of Empty Strings

```
str = strings([2,6])
```

```
str = 2x6 string array
    ""    ""    ""    ""    ""    ""
    ""    ""    ""    ""    ""    ""
```

Clone String Array Size from Existing Array

Create an array of empty strings that is the same size as an existing array.

```
A = [1 2 3; 4 5 6];
sz = size(A);
str = strings(sz)
```

```
str = 2x3 string array
    ""    ""    ""
    ""    ""    ""
```

It is a common pattern to combine the previous two lines of code into a single line:

```
str = strings(size(A));
```

You can use `strings` to preallocate the space required for a large string array.

Input Arguments

`n` — Size of square array

nonnegative integer

Size of a square array, specified as a nonnegative integer.

- If `n` is `0`, then `str` is an empty array.
- If `n` is negative, then `strings` treats `n` as `0`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz1,...,szN` — Size of each dimension (as separate arguments)

nonnegative integers

Size of each dimension, specified as separate nonnegative integer arguments.

- If the size of any dimension is `0`, then `str` is an empty array.
- If the size of any dimension is negative, then `strings` treats it as `0`.
- Beyond the second dimension, `strings` ignores trailing dimensions with a size of `1`. For example, `strings(3,1,1,1)` produces a 3-by-1 vector of strings with no characters.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz` — Size of each dimension (as a row vector)

nonnegative integers

Size of each dimension, specified as a row vector of nonnegative integers. Each element of this vector indicates the size of the corresponding dimension:

- If the size of any dimension is `0`, then `str` is an empty array.
- If the size of any dimension is negative, then `strings` treats it as `0`.
- Beyond the second dimension, `strings` ignores trailing dimensions with a size of `1`. For example, `strings([3,1,1,1])` produces a 3-by-1 vector of strings with no characters.

Example: `sz = [2,3,4]` creates a 2-by-3-by-4 array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

See Also

`cellstr` | `char` | `empty` | `isempty` | `isstring` | `size` | `string` | `strlength` | `zeros`

Topics

"Create String Arrays"
"Analyze Text Data with String Arrays"
"Test for Empty Strings and Missing Values"
"Preallocation"

Introduced in R2016b

strip

Remove leading and trailing characters from stringS

Syntax

```
newStr = strip(str)
newStr = strip(str,side)
newStr = strip( ___ ,stripCharacter)
```

Description

`newStr = strip(str)` removes all consecutive whitespace characters from the beginning and end of `str`, and returns the result as `newStr`.

`newStr = strip(str,side)` removes all consecutive whitespace characters from the side specified by `side`. The `side` argument can be `'left'`, `'right'`, or `'both'`.

`newStr = strip( ___ ,stripCharacter)` strips the character specified by `stripCharacter`, instead of the space character. You can use any of the input arguments in the previous syntaxes.

Examples

Strip Leading and Trailing Spaces from Strings

Create a string array. Starting in R2017a, you can create strings using double quotes.

```
str = ["Ann Marie      ";
       "          James";
       "Pauline        "]
```

```
str = 3x1 string array
    "Ann Marie      "
    "          James"
```

```
    "Pauline        "
```

Delete the leading and trailing space characters in each string.

```
newStr = strip(str)
```

```
newStr = 3x1 string array
    "Ann Marie"
    "James"
    "Pauline"
```

Strip Spaces from Side of String

Create a string array. Starting in R2017a, you can create strings using double quotes.

```
str = ["   Ann Marie   ";
       "   James       ";
       "   Pauline     "]
```

```
str = 3x1 string array
    "   Ann Marie   "
    "   James       "
    "   Pauline     "
```

Delete space characters from the right side only.

```
newStr = strip(str,'right')
```

```
newStr = 3x1 string array
    "   Ann Marie"
    "   James"
    "   Pauline"
```

Strip Different Character from String Array

Create a string array with elements that represent numbers. Starting in R2017a, you can create strings using double quotes. The strings include leading zeroes that make them all the same length.

```
str = ["0095.36";
       "0003.44";
       "0007.82"]
```

```
str = 3x1 string array
    "0095.36"
    "0003.44"
    "0007.82"
```

Delete the leading zeroes.

```
newStr = strip(str,'left','0')
```

```
newStr = 3x1 string array
    "95.36"
    "3.44"
    "7.82"
```

Input Arguments

`str` — Input text
string array | character vector | cell array of character vectors

Input text, specified as a string array, a character vector, or a cell array of character vectors.

Data Types: `string` | `char` | `cell`

`side` — Side of string to strip
`'both'` (default) | `'left'` | `'right'`

Side of string to strip, specified as `'left'`, `'right'`, or `'both'`. The default behavior of `strip` is to strip characters from both the left and the right side of the input text.

Data Types: `char` | `string`

`stripCharacter` — Character to strip from input text
`' '` (default) | character | string

Character to strip from input text, specified as a character or as a string that contains one character.

Data Types: `char` | `string`

Output Arguments

`newStr` — Output text
string array | character vector | cell array of character vectors

Output text, returned as a string array, a character vector, or a cell array of character vectors. `str` and `newStr` are the same data type.

Data Types: `string` | `char` | `cell`

Algorithms

`strip` does not remove significant whitespace characters.

This table shows the most common characters that are significant whitespace characters and their descriptions. For more information, see Whitespace character.

Significant Whitespace Character	Description
`char(133)`	Next line
`char(160)`	Nonbreaking space
`char(8199)`	Figure space
`char(8239)`	Narrow no-break space

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `str` must be a string scalar, a character vector, or a cell array containing not more than one character vector.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`erase` | `join` | `pad` | `replace` | `split` | `strsplit`

Topics

“Create String Arrays”
“Search and Replace Text”
“Analyze Text Data with String Arrays”

Introduced in R2016b

strjoin

Join strings in array

Syntax

```
str = strjoin(C)
str = strjoin(C,delimiter)
```

Description

`str = strjoin(C)` constructs `str` by linking the elements of `C` with a space between consecutive elements. `C` can be a cell array of character vectors or a string array.

`str = strjoin(C,delimiter)` constructs `str` by linking each element of `C` with the elements in `delimiter`.

Examples

Join List of Words with Whitespace

Join individual character vectors in a cell array of character vectors, `C`, with a single space.

```
C = {'one','two','three'};
str = strjoin(C)
```

```
str =
'one two three'
```

Join Cell Array of Character Vectors with Delimiter

Join the character vectors in a cell array into one character vector. Specify a comma followed by a space character as the delimiter.

```
C = {'Newton','Gauss','Euclid','Lagrange'}
```

```
C = 1x4 cell array
    {'Newton'}    {'Gauss'}    {'Euclid'}    {'Lagrange'}
```

```
str = strjoin(C,', ')
```

```
str =
'Newton, Gauss, Euclid, Lagrange'
```

Join Character Vectors with Multiple Different Delimiters

Specify multiple different delimiters in a cell array of character vectors. The `delimiter` cell array must have one fewer element than `C`.

```
C = {'one','two','three'};
str = strjoin(C,{' + ',' = '})
```

```
str =
'one + two = three'
```

Input Arguments

`C` — Input text
1-by-n cell array of character vectors | 1-by-n string array

Input text, specified as a 1-by-n cell array of character vectors or string array.

Example: `{'The','rain','in','Spain'}`

Example: `["Four","score","and","seven"]`

Data Types: `cell` | `string`

`delimiter` — Delimiting characters

character vector | 1-by-n cell array of character vectors | 1-by-n string array

Delimiting characters, specified as a character vector, a 1-by-n cell array of character vectors, or a 1-by-n string array.

- If `delimiter` is a character vector, then `strjoin` forms `str` by inserting `delimiter` between each element of `C`. The `delimiter` input can include any of these escape sequences:

`\\`	Backslash
`\0`	Null
`\a`	Alarm
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab

- If `delimiter` is a cell array of character vectors, then it must contain one fewer element than `C`. Each element in the cell array must contain a character vector. `strjoin` forms `str` by interleaving the elements of `delimiter` and `C`. All characters in `delimiter` are inserted as literal text, and escape sequences are not supported.

Example: `', '`

Example: `{',',' '}`

Data Types: `char` | `cell` | `string`

Tips

- Starting in R2016b, the `join` function is recommended to join elements of a string array.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Input text must be a string scalar or a cell array of character vectors.
- Generated code returns an empty output as a 1-by-0 character array.

See Also

`cellstr` | `join` | `regexp` | `strcat` | `strsplit`

Introduced in R2013a

strlength

Lengths of strings

Syntax

```
L = strlength(str)
```

Description

`L = strlength(str)` returns the number of characters in `str`.

Examples

Number of Characters in String Scalar

Starting in R2017a, you can create a string using double quotes. The result is a 1-by-1 string array, or *string scalar*.

```
str = "Hello, World"
```

```
str =
"Hello, World"
```

Return the number of characters in `str`.

```
L = strlength(str)
```

```
L = 12
```

Length of Each String in String Array

Create a string array using the `[]` operator. `str` is a 2-by-3 string array that contains six strings.

```
str = ["Amis","Chekhov","Joyce";"Stein","","Proust"]
```

```
str = 2x3 string array
    "Amis"     "Chekhov"    "Joyce"
    "Stein"    ""           "Proust"
```

Find the length of each string in `str`. Use `strlength`, not `length`, to determine the number of characters in each element of a string array.

```
L = strlength(str)
```

```
L = 2×3

     4     7     5
     5     0     6
```

Number of Characters in Character Vector

Create a character vector. To return the number of characters in the character vector, use the `strlength` function.

```
chr = 'The rain in Spain.'
```

```
chr =
'The rain in Spain.'
```

```
L = strlength(chr)
```

```
L = 18
```

Input Arguments

`str` — Input text
string array | character vector | cell array of character vectors

Input text, specified as a string array, a character vector, or a cell array of character vectors.

Tips

- To find the length of the largest array dimension of `str`, use the `length` function, not `strlength`.

Algorithms

`strlength` counts the number of *code units* in text. Code units are bit sequences for encoding characters of a character encoding system. In some character encodings, such as UTF-16, there are some characters that are encoded with multiple code units.

If you have a string or a character vector that contains such characters, then the number of code units is greater than the number of characters.

`length(C)` also returns the number of code units when `C` is a character vector.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`isStringScalar` | `isstring` | `length` | `ndims` | `numel` | `size` | `string`

Topics

"Text in String and Character Arrays"
"Create String Arrays"
"Analyze Text Data with String Arrays"
"Search and Replace Text"
"Compare Text"

Introduced in R2016b

strjust

Justify strings

Syntax

```
newStr = strjust(str)
newStr = strjust(str,side)
```

Description

`newStr = strjust(str)` returns a right-justified version of the text in `str`.

- If `str` has trailing whitespace characters, then they become leading whitespace characters in `newStr`.
- If `str` does not have trailing whitespace characters, then `strjust` returns `str` unaltered.

`newStr = strjust(str,side)` returns a version of the text that is justified on the side specified by `side`. The text in `str` can be justified on the left, right, or center.

Examples

Right-Justify Text

Create a string array in which some elements have trailing whitespace characters. Starting in R2017a, you can create strings using double quotes.

```
str1 = ["Skylab";
        "Mir   ";
        "ISS   "]
```

```
str1 = 3x1 string array
    "Skylab"
    "Mir   "
```

```
    "ISS   "
```

Justify the text on the right. `strjust` does not alter the first element because `"Skylab"` has no whitespace.

```
str2 = strjust(str1)
```

```
str2 = 3x1 string array
    "Skylab"
    "   Mir"
    "   ISS"
```

Center-Justify Text

Create a cell array of character vectors. Some of the character vectors have leading and trailing whitespace characters.

```
C1 = {'Euler    ';
      'Fibonacci';
      '    Gauss'}
```

```
C1 = 3x1 cell array
    {'Euler    '}
    {'Fibonacci'}
    {'    Gauss'}
```

Center-justify the text. If a piece of text has leading or trailing whitespace, or both, then `strjust` adjusts the text to have an equal number of leading and trailing whitespace characters. `strjust` does not alter a piece of text when it has neither leading nor trailing whitespace.

```
C2 = strjust(C1,'center')
```

```
C2 = 3x1 cell array
    {'  Euler  '}
    {'Fibonacci'}
    {'  Gauss  '}
```

Input Arguments

`str` — Input text
character array | cell array of character vectors | string array

Input text, specified as a character array, a cell array of character vectors, or a string array.

`side` — Side to justify text on
`'right'` (default) | `'left'` | `'center'`

Side to justify text on, specified as `'left'`, `'right'`, or `'center'`. The default behavior is to justify the text on the right.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Input text must be a string scalar or a character vector.
- Generated code returns an empty output as a 1-by-0 character array.

See Also
`deblank` | `pad` | `strip` | `strtrim`

Topics
"Formatting Text"
"Search and Replace Text"

Introduced before R2006a

strmatch

(Not recommended) Find possible matches for string

Note `strmatch` is not recommended. Use `startsWith`, `strncmp`, or `validatestring`, depending on your requirements, instead. `startsWith` and `strncmp` return a logical array indicating which array elements begin with the specified string, whereas `validatestring` returns a single string that represents the best match to the specified string. See Example 2, below.

To find an exact match for a string, use `strcmp`.

Syntax

```
x = strmatch(str, strarray)
x = strmatch(str, strarray, 'exact')
```

Description

`x = strmatch(str, strarray)` looks through the rows of the character array or cell array of character vectors `strarray` to find character vectors that begin with the text contained in `str`, and returns the matching row indices. If `strmatch` does not find `str` in `strarray`, `x` is an empty matrix (`[]`). Any trailing space characters in `str` or `strarray` are ignored when matching. `strmatch` is fastest when `strarray` is a character array.

`x = strmatch(str, strarray, 'exact')` compares `str` with each row of `strarray`, looking for an exact match of the entire character vector. Any trailing space characters in `str` or `strarray` are ignored when matching.

Examples

Example 1

The statement

```
x = strmatch('max', char('max', 'minimax', 'maximum'))
```

returns `x = [1; 3]` since rows 1 and 3 begin with `'max'`. The statement

```
x = strmatch('max', char('max', 'minimax', 'maximum'),'exact')
```

returns `x = 1`, since only row 1 matches `'max'` exactly.

Example 2

This example shows how to replace use of the `strmatch` function with `validatestring` or `strncmp`.

To start with, use `strmatch` to return the index of those elements for which there is a match:

```
list = {'max', 'minimax', 'maximum', 'max'}
x = strmatch('max',list)
x =
     1
     3
     4
```

`validatestring` returns the string representing the best match. If multiple or no matches exist, this statement would return an error:

```
list = {'max', 'minimax', 'maximum', 'max'};
x = validatestring('max', list)
x =
   max
```

`strncmp` returns a logical array indicating which strings match the specified string:

```
list = {'max', 'minimax', 'maximum', 'max'};
x = strncmp('max', list, 3)
x =
     1     0     1     1
```

If you prefer that MATLAB return the numeric indices of `list`, use `find` as follows:

```
list = {'max', 'minimax', 'maximum', 'max'}
x = find(strncmp(list, 'max', 3))
```

If your input to `strmatch` is a character array with multiple rows, then first convert the character array to a cell array using `cellstr`. Then, pass the output from `cellstr` to `strncmp` or `validatestring`

See Also

`regexp` | `regexpi` | `regexprep` | `startsWith` | `strcmp` | `strcmpi` | `strfind` | `strncmp` | `strncmpi`

Introduced before R2006a

strncmp

Compare first `n` characters of strings (case sensitive)

Syntax

```
tf = strncmp(s1,s2,n)
```

Description

`tf = strncmp(s1,s2,n)` compares up to `n` characters of `s1` and `s2`. The function returns `1` (`true`) if the two are identical and `0` (`false`) otherwise. Text is considered identical if the content of each is the same up to the end or the first `n` characters, whichever comes first. The return result `tf` is of data type `logical`.

The first two input arguments can be any combination of string arrays, character vectors, and cell arrays of character vectors.

Examples

Compare First `N` Characters of Character Vectors

Create two different character vectors. Compare the first 11 characters of them.

```
s1 = 'Kansas City, KS';
s2 = 'Kansas City, MO';
tf = strncmp(s1,s2,11)
```

```
tf = logical
   1
```

`tf` is `1` because both character vectors start with `'Kansas City'`.

Compare the two character vectors using `strcmp`.

```
tf = strcmp(s1,s2)
```

```
tf = logical
   0
```

`tf` is `0` because `s1` and `s2` end with different characters.

Find Text That Starts with Same Characters

Starting in R2017a, you can create strings using double quotes. Create a string array that contains names. Find the names that start with `'Jean'`.

```
s1 = ["Jacques";
      "Jean";
      "Jeanne";
      "Jean-Luc";
      "Julie"];
s2 = "Jean";
```

```
tf = strncmp(s1,s2,4)
```

```
tf = 5x1 logical array

   0
   1
   1
   1
   0
```

`tf` is `1` for all names whose first four characters are `'Jean'`.

Alternatively, you can use the `startsWith` function.

```
tf = startsWith(s1,s2)
```

```
tf = 5x1 logical array

   0
   1
   1
```

```
1
0
```

Input Arguments

s1,s2 — Input text

character vector | character array | cell array of character vectors | string array

Input text, with each input specified as a character vector, a character array, a cell array of character vectors, or a string array. The order of the inputs does not affect the comparison results.

- If both `s1` and `s2` are string arrays or cell arrays of character vectors, then `s1` and `s2` must be the same size, unless one of them is scalar.
- If both `s1` and `s2` are character arrays with multiple rows, then `s1` and `s2` can have different numbers of rows.
- When comparing a nonscalar cell array of character vectors or string array to a multirow character array, the cell array or string array must be a column vector with the same number of rows as the character array.

Data Types: `char` | `cell` | `string`

n — Maximum number of characters to compare

integer

Maximum number of characters to compare, specified as an integer.

- If `n` is `0`, then `strncmp` always returns `1`. By convention, the zeroth character of a character vector or a string scalar is always `''`, a `0`-by-`0` character array.
- If `n` is less than `0`, then `strncmp` treats it as `0`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`tf` — True or false result
`1` | `0` | logical array

True or false result, returned as a `1` or `0` of data type `logical`.

- If each input is either a string scalar or a character vector, then `tf` is a scalar.
- If at least one input is either a string array or a cell array of character vectors, then `tf` is an array the same size as the input array.
- If one input is a character array with multiple rows, and the other input is either a scalar cell or a string scalar, then `tf` is an `n`-by-`1` array, where `n` is the number of rows in the character array.
- If both inputs are character arrays, then `tf` is a scalar.

Tips

- The `strncmp` function is intended for comparison of text. If used on numeric arrays, `strncmp` always returns `0`.
- For case-insensitive text comparison, use `strncmpi` instead of `strncmp`.
- Although `strncmp` shares a name with a C function, it does not follow the C language convention of returning `0` when the text inputs match.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Enumeration inputs are not supported.
- When one input is a cell array and the other input is a character array, the character array must be a compile-time row vector.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- `s1` and `s2` must be string arrays or cell arrays of character vectors.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`endsWith` | `eq` | `ne` | `regexp` | `regexpi` | `sort` | `startsWith` | `strcmp` | `strcmpi` | `strfind` | `strncmpi`

Topics

“Compare Text”

Introduced before R2006a

strncmpi

Compare first `n` characters of strings (case insensitive)

Syntax

```
tf = strncmpi(s1,s2,n)
```

Description

`tf = strncmpi(s1,s2,n)` compares up to `n` characters of `s1` and `s2`, ignoring any differences in letter case. The function returns `1` (`true`) if the two are identical and `0` (`false`) otherwise. Text is considered identical if the content of each is the same up to the end or the first `n` characters, whichever comes first, ignoring case. The return result `tf` is of data type `logical`.

The first two input arguments can be any combination of string arrays, character vectors, and cell arrays of character vectors.

Examples

Compare First `N` Characters While Ignoring Case

Create two character vectors. Compare the first four characters of each, ignoring case.

```
s1 = 'DATA.TAR.GZ';
s2 = 'data-samples.xls';

tf = strncmpi(s1,s2,4)
```

```
tf = logical
   1
```

`tf` is `1` because `s1` starts with `'DATA'`, and `s2` starts with `'data'`.

Find Starting Text While Ignoring Case

Starting in R2017a, you can create strings using double quotes. Create a string array that contains names. Find the names that start with `'JEAN'`, ignoring case.

```
s1 = ["Jacques";
      "Jean";
      "Jeanne";
      "Jean-Luc";
      "Julie"];
s2 = "JEAN";
```

```
tf = strncmpi(s1,s2,4)
```

```
tf = 5x1 logical array

   0
   1
   1
   1
   0
```

`tf` is `1` for all names whose first four characters match `'JEAN'` when you ignore case.

Alternatively, you can use the `startsWith` function.

```
tf = startsWith(s1,s2,'IgnoreCase',true)
```

```
tf = 5x1 logical array

   0
   1
   1
   1
   0
```

Input Arguments

`s1,s2` — Input text

character vector | character array | cell array of character vectors | string array

Input text, with each input specified as a character vector, a character array, a cell array of character vectors, or a string array. The order of the inputs does not affect the comparison results.

- If both `s1` and `s2` are string arrays or cell arrays of character vectors, then `s1` and `s2` must be the same size, unless one of them is scalar.
- If both `s1` and `s2` are character arrays with multiple rows, then `s1` and `s2` can have different numbers of rows.
- When comparing a nonscalar cell array of character vectors or string array to a multirow character array, the cell array or string array must be a column vector with the same number of rows as the character array.

Data Types: `char` | `cell` | `string`

`n` — Maximum number of characters to compare

integer

Maximum number of characters to compare, specified as an integer.

- If `n` is `0`, then `strncmpi` always returns `1`. By convention, the zeroth character of a character vector or a string scalar is always `''`, a `0`-by-`0` character array.
- If `n` is less than `0`, then `strncmpi` treats it as `0`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`tf` — True or false result

`1` | `0` | logical array

True or false result, returned as a `1` or `0` of data type `logical`.

- If each input is either a string scalar or a character vector, then `tf` is a scalar.

- If at least one input is either a string array or a cell array of character vectors, then `tf` is an array the same size as the input array.
- If one input is a character array with multiple rows, and the other input is either a scalar cell or a string scalar, then `tf` is an `n`-by-`1` array, where `n` is the number of rows in the character array.
- If both inputs are character arrays, `tf` is a scalar.

Tips

- The `strncmpi` function is intended for comparison of text. If used on numeric arrays, `strncmpi` always returns `0`.
- For case-sensitive text comparison, use `strncmp` instead of `strncmpi`.
- Although `strncmpi` shares a name with a C function, it does not follow the C language convention of returning `0` when the text inputs match.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Enumeration inputs are not supported.
- Input values from the `char` class must be in the range 0-127.
- When one input is a cell array and the other input is a character array, the character array must be a compile-time row vector.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- `s1` and `s2` must be string arrays or cell arrays of character vectors.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`endsWith` | `eq` | `ne` | `regexp` | `regexpi` | `sort` | `startsWith` | `strcmp` | `strcmpi` | `strfind` | `strncmp`

Topics

“Compare Text”

Introduced before R2006a

strread

(Not recommended) Read formatted data from string

Note `strread` is not recommended. Use `textscan` instead.

Syntax

```
A = strread('str')
[A, B, ...] = strread('str')
[A, B, ...] = strread('str', 'format')
[A, B, ...] = strread('str', 'format', N)
[A, B, ...] = strread('str', 'format', N, param, value, ...)
```

Description

`A = strread('str')` reads numeric data from input character vector `str` into a 1-by-`N` vector `A`, where `N` equals the number of whitespace-separated numbers in `str`. Use this form only with character vectors containing numeric data. See “Example 1” on page 1-14763 below.

`[A, B, ...] = strread('str')` reads numeric data from the character vector input `str` into scalar output variables `A`, `B`, and so on. The number of output variables must equal the number of whitespace-separated numbers in `str`. Use this form only with character vectors containing numeric data. See “Example 2” on page 1-14764 below.

`[A, B, ...] = strread('str', 'format')` reads data from `str` into variables `A`, `B`, and so on using the specified `format`. The number of output variables `A`, `B`, etc. must be equal to the number of format specifiers (e.g., `%s` or `%d`) in the `format` argument. You can read all of the data in `str` to a single output variable as long as you use only one format specifier in the command. See “Example 4” on page 1-14764 and “Example 5” on page 1-14765 below.

The table Formats for strread lists the valid format specifiers. More information on using formats is available under “Formats” on page 1-14767 in the “Tips” on page 1-14766 section below.

`[A, B, ...] = strread('str', 'format', N)` reads data from `str` reusing the `format` character vector `N` times, where `N` is an integer greater than zero. If `N` is -1, `strread` reads the entire character vector. When `str` contains only numeric data, you can set `format` to the empty character vector (`''`). See “Example 3” on page 1-14764 below.

`[A, B, ...] = strread('str', 'format', N, param, value, ...)` customizes `strread` using `param`/`value` pairs, as listed in the table Parameters and Values for strread below. When `str` contains only numeric data, you can set `format` to the empty character vector (`''`). The `N` argument is optional and may be omitted entirely. See “Example 7” on page 1-14765 below.

Formats for strread

Format	Action	Output
Literals (ordinary characters)	Ignore the matching characters. For example, in a character vector that has `Dept` followed by a number (for department number), to skip the `Dept` and read only the number, use `'Dept'` in the `format` character vector.	None
`%d`	Read a signed integer value.	Double array
`%u`	Read an integer value.	Double array
`%f`	Read a floating-point value.	Double array
`%s`	Read a white-space separated character vector.	Cell array of character vectors
`%q`	Read a double quoted character vector, ignoring the quotes.	Cell array of character vectors
`%c`	Read characters, including white space.	Character array
`%[...]`	Read the longest character vector containing characters specified in the brackets.	Cell array of character vectors
`%[^...]`	Read the longest nonempty character vector containing characters that are not specified in the brackets.	Cell array of character vectors
`%*...`	Ignore the characters following `*`. See “Example 8” on page 1-14766 below.	No output
`%w...`	Read field width specified by `w`. The `%f` format supports `%w.pf`, where `w` is the field width and `p` is the precision.	

Parameters and Values for strread

param	value		Action
`whitespace`	Any from the list below:		Treats vector of characters, `*`, as white space. Default is `\b\r\n\t`.
	`\b`	Backspace	
	`\n`	New line	
	`\r`	Carriage return	
	`\t`	Horizontal tab	
	`\\`	Backslash	
	`%%`	Percent sign	
	`''`	Single quotation mark	
`delimiter`	Delimiter character		Specifies delimiter character. Default is one or more whitespace characters.
`expchars`	Exponent characters		Default is `eEdD`.
`bufsize`	Positive integer		Specifies the maximum character vector length, in bytes. Default is `4095`.
`commentstyle`	`matlab`		Ignores characters after `%`.
`commentstyle`	`shell`		Ignores characters after `#`.
`commentstyle`	`c`		Ignores characters between `/*` and `*/`.
`commentstyle`	`c++`		Ignores characters after `//`.
`emptyvalue`	Value to return for empty numeric fields in delimited files		Default is `0`.

Examples

Example 1

Read numeric data into a 1-by-5 vector:

```
a = strread('0.41 8.24 3.57 6.24 9.27')
a =
    0.4100    8.2400    3.5700    6.2400    9.2700
```

Example 2

Read numeric data into separate scalar variables:

```
[a b c d e] = strread('0.41 8.24 3.57 6.24 9.27')
a =
    0.4100
b =
    8.2400
c =
    3.5700
d =
    6.2400
e =
    9.2700
```

Example 3

Read the only first three numbers in the character vector, also formatting as floating point:

```
a = strread('0.41 8.24 3.57 6.24 9.27', '%4.2f', 3)

a =
    0.4100
    8.2400
    3.5700
```

Example 4

Truncate the data to one decimal digit by specifying format `%3.1f`. The second specifier, `%*1d`, tells `strread` not to read in the remaining decimal digit:

```
a = strread('0.41 8.24 3.57 6.24 9.27', '%3.1f %*1d')

a =
    0.4000
    8.2000
    3.5000
    6.2000
    9.2000
```

Example 5

Read six numbers into two variables, reusing the format specifiers:

```
[a b] = strread('0.41 8.24 3.57 6.24 9.27 3.29', '%f %f')

a =
    0.4100
    3.5700
    9.2700
b =
    8.2400
    6.2400
    3.2900
```

Example 6

Read character vector and numeric data to two output variables. Ignore commas in the input character vector:

```
str = 'Section 4, Page 7, Line 26';

[name value] = strread(str, '%s %d,')
name =
    'Section'
    'Page'
    'Line'
value =
     4
     7
    26
```

Example 7

Read the character vector used in the last example, but this time delimiting with commas instead of spaces:

```
str = 'Section 4, Page 7, Line 26';

[a b c] = strread(str, '%s %s %s', 'delimiter', ',')
a =
    'Section 4'
b =
```

```
    'Page 7'
c =
    'Line 26'
```

Example 8

Read selected portions of the input character vector:

```
str = '<table border=5 width="100%" cellspacing=0>';

[border width space] = strread(str, ...
   '%*s%*s %c %*s "%4s" %*s %c', 'delimiter', '= ')
border =
     5
width =
    '100%'
space =
     0
```

Example 9

Read the character vector into two vectors, restricting the `Answer` values to `T` and `F`. Also note that two delimiters (comma and space) are used here:

```
str = 'Answer_1: T, Answer_2: F, Answer_3: F';

[a b] = strread(str, '%s %[TF]', 'delimiter', ', ')
a =
    'Answer_1:'
    'Answer_2:'
    'Answer_3:'
b =
    'T'
    'F'
    'F'
```

Tips

If you terminate the input character vector with a newline character (`\n`), `strread` returns arrays of equal size by padding arrays of lesser size with the `emptyvalue` character:

```
[A,B,C] = strread(sprintf('5,7,1,9\n'),'%d%d%d', ...
             'delimiter', ',', 'emptyvalue',NaN)
A =
     5
     9
B =
     7
   NaN
C =
     1
   NaN
```

If you remove the `\n` from the input character vector of this example, array `A` continues to be a 2-by-1 array, but `B` and `C` are now 1-by-1.

Delimiters

If your data uses a character other than a space as a delimiter, you must use the `strread` parameter `'delimiter'` to specify the delimiter. For example, if the character vector `str` used a semicolon as a delimiter, you would use this command:

```
[names, types, x, y, answer] = strread(str,'%s %s %f ...
    %d %s','delimiter',';')
```

Formats

The `format` character vector determines the number and types of return arguments. The number of return arguments must match the number of conversion specifiers in the `format` character vector.

The `strread` function continues reading `str` until the entire character vector is read. If there are fewer format specifiers than there are entities in `str`, `strread` reapplies the format specifiers, starting over at the beginning. See “Example 5” on page 1-14765 below.

The `format` character vector supports a subset of the conversion specifiers and conventions of the C language `fscanf` routine. White-space characters in the `format` character vector are ignored.

Preserving White-Space

If you want to preserve leading and trailing spaces in a character vector, use the `whitespace` parameter as shown here:

```
str = '   An  example      of preserving     spaces      ';

strread(str, '%s', 'whitespace', '')
ans =
    '   An  example      of preserving     spaces      '
```

See Also

`sscanf` | `textscan`

Introduced before R2006a

strrep

Find and replace substrings

Syntax

```
newStr = strrep(str,old,new)
```

Description

`newStr = strrep(str,old,new)` replaces all occurrences of `old` in `str` with `new`.

If any input argument is a nonscalar string array or cell array of character vectors, then the other input arguments must have compatible sizes.

Examples

Replace Text in Character Vector

Create a character vector and replace a substring within it.

```
chr = 'The quick brown fox'
```

```
chr =
'The quick brown fox'
```

```
newChr = strrep(chr,'quick','sly')
```

```
newChr =
'The sly brown fox'
```

Replace Text in String Array

Create a string array. Starting in R2017a, you can create strings using double quotes.

```
str = ["the quick brown fox";
       "and the lazy dog"]
```

```
str = 2x1 string array
    "the quick brown fox"
    "and the lazy dog"
```

Replace a substring in each element of the array.

```
newStr = strrep(str,'the','a')
```

```
newStr = 2x1 string array
    "a quick brown fox"
    "and a lazy dog"
```

Replace Substrings with Multiple Values

Replace placeholder content in a cell array, `'___'`, with different values in a second cell array.

```
C1 = {'Date Received: ___';
      'Date Accepted: ___'};
old = '___';
new = {'2016-09-06';
       '2016-10-11'};
C2 = strrep(C1,old,new)
```

```
C2 = 2x1 cell array
    {'Date Received: 2016-09-06'}
    {'Date Accepted: 2016-10-11'}
```

Replace Repeated Pattern

Create a character vector with a repeated, overlapping pattern. Compare the results of using the `strrep`, `replace`, and `regexprep` functions to replace the pattern.

```
repeats = 'abc 2 def 22 ghi 222 jkl 2222'
```

```
repeats =
'abc 2 def 22 ghi 222 jkl 2222'
```

Find the indices of the repeating pattern `'22'` using the `strfind` function. `strfind` finds all instances of the pattern, including instances that overlap.

```
indices = strfind(repeats, '22')
```

```
indices = 1×6

    11    18    19    26    27    28
```

Replace `'22'` using `strrep`. When you use `strrep`, it replaces every instance identified by `strfind`.

```
using_strrep = strrep(repeats, '22', '*')
```

```
using_strrep =
'abc 2 def * ghi ** jkl ***'
```

Replace `'22'` using `replace`. It does not replace every instance that `strrep` replaces.

```
using_replace = replace(repeats, '22', '*')
```

```
using_replace =
'abc 2 def * ghi *2 jkl **'
```

Replace `'22'` using `regexprep`. The results are identical to the results using the `replace` function.

```
using_regexprep = regexprep(repeats, '22', '*')
```

```
using_regexprep =
'abc 2 def * ghi *2 jkl **'
```

`strrep` finds all instances of a pattern before replacing any instance. However, the `replace` and `regexprep` functions replace an instance of a pattern as soon as they find it within the text.

Input Arguments

`str` — Input text
string array | character vector | cell array of character vectors

Input text, specified as a string array, character vector, or cell array of character vectors.

Data Types: `string` | `char` | `cell`

`old` — Substring to replace
string array | character vector | cell array of character vectors

Substring to replace, specified as a string array, character vector, or cell array of character vectors.

Data Types: `string` | `char` | `cell`

`new` — New substring
string array | character vector | cell array of character vectors

New substring, specified as a string array, character vector, or cell array of character vectors.

Data Types: `string` | `char` | `cell`

Algorithms

- The `strrep` function does not find empty character vectors or empty strings for replacement. That is, when `str` and `old` both contain the empty character vector (`''`) or the empty string(`""`), `strrep` does not replace empty character vectors or strings with the contents of `new`.
- Before replacing text, `strrep` finds all instances of `old` in `str`, like the `strfind` function. For overlapping patterns, `strrep` performs multiple replacements.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

- The original string must be a tall array of strings or a tall cell array of character vectors.
- The old string and new string inputs can be single strings or tall arrays of strings with the same size.

For more information, see “Tall Arrays for Out-of-Memory Data”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `str`, `old`, and `new` must be a string scalar or a character vector.
- Generated code returns an empty output as a 1-by-0 character array.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- Inputs must be string arrays or cell arrays of character vectors.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`erase` | `regexp` | `regexprep` | `replace` | `strfind`

Topics

"Create String Arrays"
"Search and Replace Text"
"Test for Empty Strings and Missing Values"

Introduced before R2006a

strsplit

Split string or character vector at specified delimiter

Syntax

```
C = strsplit(str)
C = strsplit(str,delimiter)
C = strsplit(str,delimiter,Name,Value)

[C,matches] = strsplit( ___ )
```

Description

`C = strsplit(str)` splits `str` at whitespace into `C`. A whitespace character is equivalent to any sequence in the set `{' ','\f','\n','\r','\t','\v'}`.

If `str` has consecutive whitespace characters, then `strsplit` treats them as one whitespace.

`C = strsplit(str,delimiter)` splits `str` at the delimiters specified by `delimiter`.

If `str` has consecutive delimiters, with no other characters between them, then `strsplit` treats them as one delimiter. For example, both `strsplit('Hello,world',',')` and `strsplit('Hello,,,world',',')` return the same output.

`C = strsplit(str,delimiter,Name,Value)` specifies additional delimiter options using one or more name-value pair arguments. For example, to treat consecutive delimiters as separate delimiters, you can specify `'CollapseDelimiters',false`.

`[C,matches] = strsplit( ___ )` additionally returns the array, `matches`. The `matches` output argument contains all occurrences of delimiters upon which `strsplit` splits `str`. You can use this syntax with any of the input arguments of the previous syntaxes.

Examples

Split Character Vector on Whitespace

```
str = 'The rain in Spain.';
C = strsplit(str)
```

```
C = 1x4 cell array
    {'The'}    {'rain'}    {'in'}    {'Spain.'}
```

`C` is a cell array containing four character vectors.

Split Character Vector of Values on Specific Delimiter

Split a character vector that contains comma-separated values.

```
data = '1.21, 1.985, 1.955, 2.015, 1.885';
C = strsplit(data,', ')
```

```
C = 1x5 cell array
    {'1.21'}    {'1.985'}    {'1.955'}    {'2.015'}    {'1.885'}
```

Split a character vector, `data`, which contains the units `m/s` with an arbitrary number of whitespace on either side of the text. The regular expression, `\s*`, matches any whitespace character appearing zero or more times.

```
data = '1.21m/s1.985m/s 1.955 m/s2.015 m/s 1.885m/s';
[C,matches] = strsplit(data,'\s*m/s\s*',...
    'DelimiterType','RegularExpression')
```

```
C = 1x6 cell array
    {'1.21'}    {'1.985'}    {'1.955'}    {'2.015'}    {'1.885'}    {0x0 char}


matches = 1x5 cell array
    {'m/s'}    {'m/s '}    {' m/s'}    {' m/s '}    {'m/s'}
```

In this case, the last character vector in C is empty. This empty character vector follows the last matched delimiter.

Split Path on File Separator

```
myPath = 'C:\work\matlab';
C = strsplit(myPath,'\')
```

```
C = 1x3 cell array
    {'C:'}    {'work'}    {'matlab'}
```

Split Character Vector with Multiple Delimiters

Split a character vector on `' '` and `'ain'`, treating multiple delimiters as one. Specify multiple delimiters in a cell array of character vectors.

```
str = 'The rain in Spain stays mainly in the plain.';
[C,matches] = strsplit(str,{' ','ain'},'CollapseDelimiters',true)
```

```
C = 1x11 cell array
  Columns 1 through 7

    {'The'}    {'r'}    {'in'}    {'Sp'}    {'stays'}    {'m'}    {'ly'}

  Columns 8 through 11

    {'in'}    {'the'}    {'pl'}    {'.'}


matches = 1x10 cell array
  Columns 1 through 7

    {' '}    {'ain '}    {' '}    {'ain '}    {' '}    {'ain'}    {' '}

  Columns 8 through 10

    {' '}    {' '}    {'ain'}
```

Split the same character vector on whitespace and on `'ain'`, using regular expressions and treating multiple delimiters separately.

```
[C,matches] = strsplit(str,{'\s','ain'},'CollapseDelimiters',...
    false, 'DelimiterType','RegularExpression')
```

```
C = 1x13 cell array
  Columns 1 through 6

    {'The'}    {'r'}    {0x0 char}    {'in'}    {'Sp'}    {0x0 char}

  Columns 7 through 13

    {'stays'}    {'m'}    {'ly'}    {'in'}    {'the'}    {'pl'}    {'.'}


matches = 1x12 cell array
  Columns 1 through 8

    {' '}    {'ain'}    {' '}    {' '}    {'ain'}    {' '}    {' '}    {'ain'}

  Columns 9 through 12

    {' '}    {' '}    {' '}    {'ain'}
```

In this case, `strsplit` treats the two delimiters separately, so empty character vectors appear in output `C` between the consecutively matched delimiters.

Split Text with Multiple, Overlapping Delimiters

Split text on the character vectors `', '` and `', and '`.

```
str = 'bacon, lettuce, and tomato';
[C,matches] = strsplit(str,{', ',', and '})
```

```
C = 1x3 cell array
    {'bacon'}    {'lettuce'}    {'and tomato'}


matches = 1x2 cell array
    {', '}    {', '}
```

Because the command lists `', '` first and `', and '` contains `', '`, the `strsplit` function splits `str` on the first delimiter and never proceeds to the second delimiter.

If you reverse the order of delimiters, `', and '` takes priority.

```
str = 'bacon, lettuce, and tomato';
[C,matches] = strsplit(str,{', and ',', '})
```

```
C = 1x3 cell array
    {'bacon'}    {'lettuce'}    {'tomato'}


matches = 1x2 cell array
    {', '}    {', and '}
```

Input Arguments

`str` — Input text

character vector | string scalar

Input text, specified as a character vector or a string scalar.

Data Types: `char` | `string`

`delimiter` — Delimiting characters

character vector | 1-by-n cell array of character vectors | 1-by-n string array

Delimiting characters, specified as a character vector, a 1-by-n cell array of character vectors, or a 1-by-n string array. Text specified in `delimiter` does not appear in the output `C`.

Specify multiple delimiters in a cell array or a string array. The `strsplit` function splits `str` on the elements of `delimiter`. The order in which delimiters appear in `delimiter` does not matter unless multiple delimiters begin a match at the same character in `str`. In that case `strsplit` splits on the first matching delimiter in `delimiter`.

`delimiter` can include the following escape sequences:

\\	Backslash

`\0`	Null
`\a`	Alarm
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab

Example: `','`

Example: `{'-',','}`

Data Types: `char` | `cell` | `string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'DelimiterType','RegularExpression'` instructs `strsplit` to treat `delimiter` as a regular expression.

`CollapseDelimiters` — Multiple delimiter handling
`1 (true)` (default) | `0 (false)`

Multiple delimiter handling, specified as the comma-separated pair consisting of `'CollapseDelimiters'` and either `true` or `false`. If `true`, then consecutive delimiters in `str` are treated as one. If `false`, then consecutive delimiters are treated as separate delimiters, resulting in empty character vector `''` elements between matched delimiters.

Example: `'CollapseDelimiters',true`

`DelimiterType` — Delimiter type
`'Simple'` (default) | `'RegularExpression'`

Delimiter type, specified as the comma-separated pair consisting of `'DelimiterType'` and one of the following character vectors.

`'Simple'`	Except for escape sequences, `strsplit` treats `delimiter` as literal text.
`'RegularExpression'`	`strsplit` treats `delimiter` as a regular expression.

In both cases, `delimiter` can include escape sequences.

Output Arguments

`C` — Parts of original text
cell array of character vectors | string array

Parts of the original character vector, returned as a cell array of character vectors or as a string array. `C` always contains one more element than `matches` contains. Therefore, if `str` begins with a delimiter, then the first element of `C` contains no characters. If `str` ends with a delimiter, then the last cell in `C` contains no characters.

`matches` — Identified delimiters
cell array of character vectors | string array

Identified delimiters, returned as a cell array of character vectors or as a string array. `matches` always contains one less element than output `C` contains. If `str` is a character vector or a cell array of character vectors, then `matches` is a cell array. If `str` is a string array, then `matches` is a string array.

Tips

- Starting in R2016b, the `split` function is recommended to split elements of a string array.

See Also

`regexp` | `split` | `strfind` | `strjoin`

Topics

"Regular Expressions"

Introduced in R2013a

strtok

Selected parts of strings

Syntax

```
token = strtok(str)
token = strtok(str,delimiters)
[token,remain] = strtok( ___ )
```

Description

`token = strtok(str)` parses `str` from left to right, using whitespace characters as delimiters, and returns part or all of the text in `token`. First, `strtok` ignores any leading whitespace in `str`. Then, `strtok` starts at the first character that is not whitespace, and includes all characters up to, but not including, the next whitespace character. `strtok` returns that part of the text in `token`. If `strtok` does not find any whitespace to use as a delimiter, then `token` includes all characters up to, and including, the end of `str`.

`token = strtok(str,delimiters)` parses `str` using the characters in `delimiters`. If `delimiters` includes more than one character, then `strtok` treats each character in `delimiters` as a separate delimiter. Because the delimiters are individual characters, `delimiters` can be any size, and the characters within `delimiters` can be in any order.

In this syntax, whitespace characters are not delimiters unless you include them within `delimiters`.

`[token,remain] = strtok( ___ )` returns the remaining text, if any, in `remain`. If `strtok` finds a delimiter, then it is included at the start of `remain`. If `strtok` finds no delimiters in `str`, then it returns the whole of `str`, except for leading delimiters, in `token`, and `remain` has no characters. You can use this syntax with any of the input arguments of the previous syntaxes.

Examples

Return Token from Character Vector

Create a character vector. Return all text up to the first whitespace character that is a delimiter. `strtok` discards leading whitespace characters.

```
chr = '     Four score and seven years ago'
```

```
chr =
'     Four score and seven years ago'
```

```
token = strtok(chr)
```

```
token =
'Four'
```

Return Token Using Delimiter

Create a string. Starting in R2017a, you can create strings using double quotes.

```
str = "A horse! A horse! My kingdom for a horse!"
```

```
str =
"A horse! A horse! My kingdom for a horse!"
```

Return the first part of the string using the `'!'` character as a delimiter.

```
token = strtok(str,'!')
```

```
token =
"A horse"
```

Return Tokens and Remaining Text

Create a string array. Starting in R2017a, you can create strings using double quotes.

```
str = ["all in good time";
       "my dog has fleas";
       "leave no stone unturned"]
```

```
str = 3x1 string array
    "all in good time"
```

```
    "my dog has fleas"
    "leave no stone unturned"
```

Return tokens in a string array, and return the remaining text in a second string array.

```
[token,remain] = strtok(str)
```

```
token = 3x1 string array
    "all"
    "my"
    "leave"


remain = 3x1 string array
    " in good time"
    " dog has fleas"
    " no stone unturned"
```

Parse Delimited Text into String Array

Create a string that contains sample HTML code. Break it down into segments delimited by the < and > characters. Store the segments in a string array.

Starting in R2017a, you can create strings using double quotes. To concatenate several strings into one string, use the `plus` operator, +.

```
str = "<ul class=continued><li class=continued>" + ...
      "<pre><a name=""13474""></a>token = strtok" + ...
      "(str,delimiter)<a name=""13475""></a>" + ...
      "token = strtok(str)"
```

```
str =
"<ul class=continued><li class=continued><pre><a name="13474"></a>token = strtok(str,de
```

Create an empty string array to contain the code segments.

```
segments = strings(0)
```

```
segments =

  0x0 empty string array
```

Break `str` into segments. Write a `while` loop that repeatedly calls `strtok` on the remaining HTML text. The `while` loop exits when there is no more text to parse.

```
remain = str;
while (remain ~= "")
   [token,remain] = strtok(remain, '<>');
   segments = [segments ; token];
end
```

Display the code segments.

```
segments
```

```
segments = 9x1 string array
    "ul class=continued"
    "li class=continued"
    "pre"
    "a name="13474""
    "/a"
    "token = strtok(str,delimiter)"
    "a name="13475""
    "/a"
    "token = strtok(str)"
```

Input Arguments

`str` — Input text

string array | character vector | cell array of character vectors

Input text, specified as a string array, a character vector, or a cell array of character vectors.

Data Types: `string` | `char` | `cell`

`delimiters` — Delimiting characters

string array | character vector | cell array of character vectors

Delimiting characters, specified as a string array, a character vector, or a cell array of character vectors. Because `strtok` treats individual characters as delimiters, `delimiters` can be any size and can contain characters in any order.

Example: `strtok(str,{'YZ','X'})` treats `'X'`, `'Y'`, and `'Z'` as separate delimiters. It does not treat `'YZ'` as a delimiter.

Data Types: `string` | `char` | `cell`

Output Arguments

`token` — Selected part of text

string array | character vector | cell array of character vectors

Selected part of the text, returned as a string array, a character vector, or a cell array of character vectors. `token` includes all text starting at the first character that is not a delimiter and ending at, but not including, the next delimiter. `str` and `token` are the same data type.

Data Types: `string` | `char` | `cell`

`remain` — Remainder of text

string array | character vector | cell array of character vectors

Remainder of the text, returned as a string array, a character vector, or a cell array of character vectors. If `strtok` finds a delimiter in `str`, then `remain` includes all text starting at, and including, that delimiter and ending at the end of the text. `str` and `remain` are the same data type.

Data Types: `string` | `char` | `cell`

Tips

Do not specify an escape-character sequence as a delimiter. `strtok` does not translate escape character sequences. Instead, you can use the `char` function to specify such characters. For example, to specify a tab as a delimiter use `char(9)` instead of `'\t'`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `str` and `delimiters` must be a string scalar or a character vector.
- Generated code returns an empty output as a 1-by-0 character array.

See Also

`regexp` | `split` | `splitlines` | `sprintf` | `strcmp` | `strfind` | `strncmp` | `strsplit` | `textscan`

Topics

"Formatting Text"
"Regular Expressions"

Introduced before R2006a

strtrim

Remove leading and trailing whitespace from strings

Syntax

```
newStr = strtrim(str)
```

Description

`newStr = strtrim(str)` removes leading and trailing whitespace characters from `str` and returns the result as `newStr`. However, `strtrim` does not remove significant whitespace characters. For example, `strtrim` removes leading and trailing space and tab characters, but does not remove the nonbreaking space character, `char(160)`.

Examples

Remove Leading Whitespace from Character Vector

Create a character vector with spaces and a tab character as leading whitespace.

```
chr = sprintf('  \t   Remove    leading whitespace')
```

```
chr =
'         Remove    leading whitespace'
```

Remove the leading tab and spaces.

```
newChr = strtrim(chr)
```

```
newChr =
'Remove    leading whitespace'
```

`strtrim` removes the leading whitespace characters, but not the whitespace between other characters.

Remove Leading and Trailing Spaces from String Array

Starting in R2017a, you can create strings using double quotes. Create a string array, and remove leading and trailing whitespace with the `strtrim` function.

```
str = ["   Gemini    ","   Apollo    ";
       "   ISS       ","   Skylab    "]
```

```
str = 2x2 string array
    "   Gemini    "    "   Apollo    "
    "   ISS       "    "   Skylab    "
```

```
newStr = strtrim(str)
```

```
newStr = 2x2 string array
    "Gemini"    "Apollo"
    "ISS"       "Skylab"
```

Remove Leading and Trailing Whitespace from Cell Array

Remove the leading and trailing whitespace from all the character vectors in a cell array and display them.

```
chr = {'     Trim leading whitespace';
       'Trim trailing whitespace     '}
```

```
chr = 2x1 cell array
    {'     Trim leading whitespace' }
    {'Trim trailing whitespace     '}
```

```
newChr = strtrim(chr)
```

```
newChr = 2x1 cell array
    {'Trim leading whitespace' }
    {'Trim trailing whitespace'}
```

Keep Nonbreaking Space Character

Create a character vector that includes the nonbreaking space character, `char(160)`, as a trailing whitespace character.

```
chr = '     Keep nonbreaking space';
chr = [chr char(160) '     '];
```

Display `chr` between | symbols to show the leading and trailing whitespace.

```
['|' chr '|']
```

```
ans =
'|     Keep nonbreaking space      |'
```

Remove the leading and trailing whitespace characters.

```
newChr = strtrim(chr);
```

Display `newChr` between | symbols. `strtrim` removes the space characters but leaves the nonbreaking space at the end of `newChr`.

```
['|' newChr '|']
```

```
ans =
'|Keep nonbreaking space |'
```

Input Arguments

`str` — Input text
character array | cell array of character arrays | string array

Input text, specified as a character array or as a cell array of character arrays, or a string array.

Algorithms

`strtrim` does not remove significant whitespace characters.

This table shows the most common characters that are significant whitespace characters and their descriptions. For more information, see Whitespace character.

Significant Whitespace Character	Description
`char(133)`	Next line
`char(160)`	Nonbreaking space
`char(8199)`	Figure space
`char(8239)`	Narrow no-break space

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Input text must be string scalar or a character array.
- Input values must be in the range 0–127.
- Generated code returns an empty output as a 1-by-0 character array.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- `str` must be a string array or cell array of character vectors.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cellstr` | `deblank` | `isspace` | `strjust`

Introduced before R2006a

struct

Structure array

Description

A *structure array* is a data type that groups related data using data containers called *fields*. Each field can contain any type of data. Access data in a field using dot notation of the form `structName.fieldName`.

Creation

When you have data to put into a new structure, create the structure using dot notation to name its fields one at a time:

```
s.a = 1;
s.b = {'A','B','C'}
```

```
s = struct with fields:
    a: 1
    b: {'A'  'B'  'C'}
```

You also can create a structure array using the `struct` function, described below. You can specify many fields simultaneously, or create a nonscalar structure array.

Syntax

```
s = struct
s = struct(field,value)
s = struct(field1,value1,...,fieldN,valueN)
s = struct([])

s = struct(obj)
```

Description

`s = struct` creates a scalar (1-by-1) structure with no fields.

`s = struct(field,value)` creates a structure array with the specified field and value. The `value` input argument can be any data type, such as a numeric, logical, character, or cell array.

- If `value` is *not* a cell array, or if `value` is a scalar cell array, then `s` is a scalar structure. For instance, `s = struct('a',[1 2 3])` creates a 1-by-1 structure, where `s.a = [1 2 3]`.
- If `value` is a nonscalar cell array, then `s` is a structure array with the same dimensions as `value`. Each element of `s` contains the corresponding element of `value`. For example, `s = struct('x',{'a','b'})` returns `s(1).x = 'a'` and `s(2).x = 'b'`.
- If `value` is an empty cell array `{}`, then `s` is an empty (0-by-0) structure.

`s = struct(field1,value1,...,fieldN,valueN)` creates a structure array with multiple fields.

- If none of the `value` inputs are cell arrays, or if all `value` inputs that are cell arrays are scalars, then `s` is a scalar structure.
- If any of the `value` inputs is a nonscalar cell array, then `s` has the same dimensions as that cell array. Also, if two or more `value` inputs are nonscalar cell arrays, then they all must have the same dimensions.

  For any `value` that is a scalar cell array or an array of any other data type, `struct` inserts the contents of `value` in the relevant field for all elements of `s`. For example, `s = struct('x',{'a','b'},'y','c')` returns `s(1).x = 'a'`, `s(2).x = 'b'`, `s(1).y = 'c'`, and `s(2).y = 'c'`.
- If any `value` input is an empty cell array, `{}`, then output `s` is an empty (0-by-0) structure. To specify an empty field and keep the values of the other fields, use `[]` as a `value` input instead.

`s = struct([])` creates an empty (0-by-0) structure with no fields.

`s = struct(obj)` creates a scalar structure with field names and values that correspond to properties of `obj`. The `struct` function does not convert `obj`, but rather creates `s` as a new structure. This structure does not retain the class information, so private, protected, and hidden properties become public fields in `s`. The `struct` function issues a warning when you use this syntax.

Input Arguments

`field` — Field name

character vector | string scalar

Field name, specified as a character vector or string scalar. Valid field names begin with a letter, and can contain letters, digits, and underscores. The maximum length of a field name is the value that the `namelengthmax` function returns.

`value` — Values

array

Values, specified as any type of array. If any `value` input is a nonscalar cell array, then all nonscalar cell array inputs must have the same dimensions.

If any `value` input is an empty cell array, `{}`, then the output is an empty structure array. To specify a single empty field, use `[]`.

`obj` — Object

scalar | array

Object, specified as a scalar or array of objects. The `struct` function copies the properties of `obj` to the fields of a new scalar structure.

The `struct` function does not create a structure from most of the fundamental data types. For example, if `obj` has the `double` or `char` data type, then `struct` issues an error message. However, `struct` does return the properties of a table or timetable as a structure. See “Fundamental MATLAB Classes” for the list of fundamental data types.

Examples

Store Related Data Variables in Structure

Store related pieces of data in the fields of a structure. You can give the fields human-readable names that describe the data.

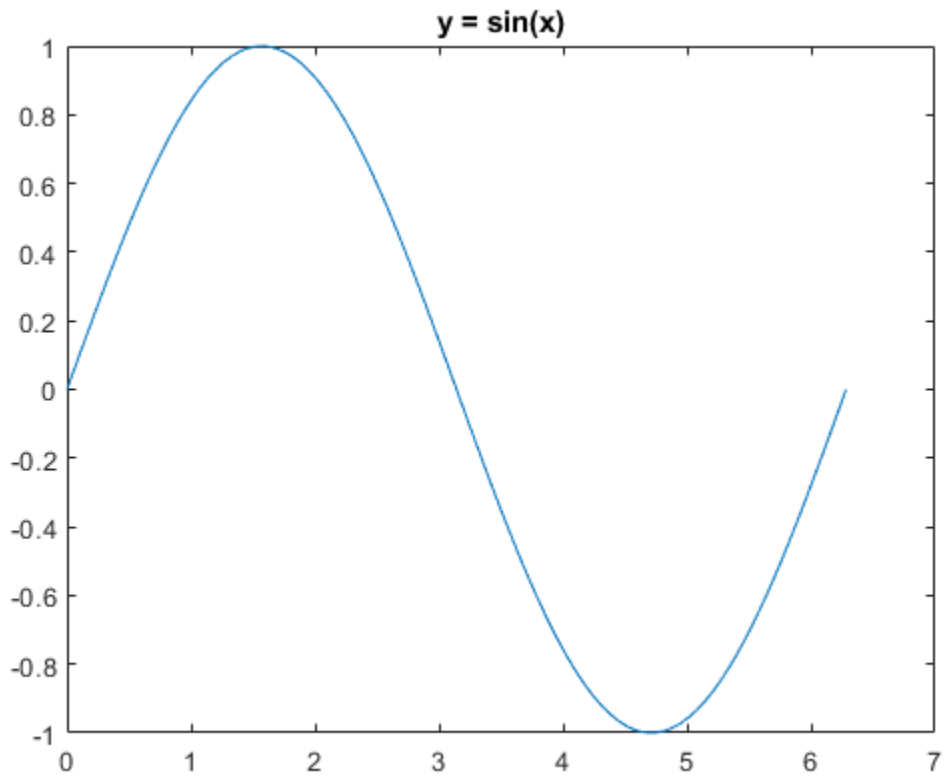
Create a structure by adding fields to it using dot notation. The fields contain x- and y-values for a sine wave, and text that describes the data.

```
data.x = linspace(0,2*pi);
data.y = sin(data.x);
data.title = 'y = sin(x)'
```

```
data = struct with fields:
        x: [1x100 double]
        y: [1x100 double]
    title: 'y = sin(x)'
```

Plot the sine wave. You can refer to the arrays of x- and y-values by their field names. Then add the title.

```
plot(data.x,data.y)
title(data.title)
```



Structure with One Field

Create a nonscalar structure that contains a single field.

```
field = 'f';
value = {'some text';
         [10, 20, 30];
         magic(5)};
s = struct(field,value)
```

```
s=3×1 struct
    f
```

View the contents of each element.

```
s.f
```

```
ans =
'some text'

ans = 1×3

    10    20    30


ans = 5×5

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

When you access a field of a nonscalar structure, such as `s.f`, MATLAB® returns a comma-separated list. In this case, `s.f` is equivalent to `s(1).f, s(2).f, s(3).f`.

Structure with Multiple Fields

Create a nonscalar structure that contains several fields.

```
field1 = 'f1';  value1 = zeros(1,10);
field2 = 'f2';  value2 = {'a', 'b'};
field3 = 'f3';  value3 = {pi, pi.^2};
field4 = 'f4';  value4 = {'fourth'};

s = struct(field1,value1,field2,value2,field3,value3,field4,value4)
```

```
s=2×4 struct
    f1
    f2
    f3
    f4
```

The cell arrays for `value2` and `value3` are 1-by-2, so `s` is also 1-by-2. Because `value1` is a numeric array and not a cell array, both `s(1).f1` and `s(2).f1` have the same contents. Similarly, because the cell array for `value4` has a single element, `s(1).f4` and `s(2).f4` have the same contents.

```
s(1)
```

```
ans = struct with fields:
    f1: [0 0 0 0 0 0 0 0 0 0]
    f2: 'a'
    f3: 3.1416
    f4: 'fourth'
```

```
s(2)
```

```
ans = struct with fields:
    f1: [0 0 0 0 0 0 0 0 0 0]
    f2: 'b'
    f3: 9.8696
    f4: 'fourth'
```

Structure with Empty Field

Create a structure that contains an empty field. Use `[]` to specify the value of the empty field.

```
s = struct('f1','a','f2',[])
```

```
s = struct with fields:
    f1: 'a'
    f2: []
```

Fields with Cell Arrays

Create a structure with a field that contains a cell array.

```
field = 'mycell';
value = {{'a','b','c'}};
s = struct(field,value)
```

```
s = struct with fields:
    mycell: {'a'  'b'  'c'}
```

Empty Structure

Create an empty structure that contains several fields.

```
s = struct('a',{},'b',{},'c',{})
```

```
s = 

  0x0 empty struct array with fields:

    a
    b
    c
```

Assign a value to a field in an empty structure.

```
s(1).a = 'a'
```

```
s = struct with fields:
    a: 'a'
    b: []
    c: []
```

Nested Structure

Create a nested structure, where `a` is a structure with a field that contains another structure.

```
a.b = struct('c',{},'d',{})
```

```
a = struct with fields:
    b: [0x0 struct]
```

View the names of the fields of `a.b`.

```
fieldnames(a.b)
```

```
ans = 2x1 cell array
    {'c'}
    {'d'}
```

Compatibility Considerations

Structure array expansion is consistent with general array expansion

Behavior changed in R2019a

Starting in R2019a, the dimensions of an expanded structure array are consistent whether you assign a value to a single field using dot notation or assign an entire structure to the array. Previously, the output dimensions were different when you did not specify indices for all dimensions. Assigning to a field using dot notation now matches the

previous behavior of assigning a structure, which is consistent with general array expansion. For example:

- If `S` is 1-by-2-by-3-by-4, then `S(5,2)=struct('a',3)` and `S(5,2).a=3` both expand `S` to 5-by-2-by-3-by-4. Previously, `S(5,2).a=3` resulted in a 5-by-24 array.
- If `S` is 0-by-5 or 5-by-0, then `S(3)=struct('a',2)` and `S(3).a=2` both expand `S` to 1-by-3. Previously, `S(3).a=1` resulted in a 1-by-5 array when `S` was 0-by-5 and in a 3-by-1 array when `S` was 5-by-0.
- If `S` is 0-by-2-by-3, then `S(3) = struct('a',2)` and`S(3).a=2` produce errors because it is ambiguous which dimension to expand. Previously, `S(3).a=2` resulted in a 1-by-6 array.
- If `S` is 1-by-1-by-3 with two fields, then `S(5)=struct('a',7,'b',[])` and `S(5).a=7` both expand `S` to 1-by-1-by-5. Previously, `S(5).a=7` resulted in a 1-by-5 array.
- If `S` is 0-by-1 with two fields, then `S(5)=struct('a',7,'b',[])` and `S(5).a=7` both expand `S` to 5-by-1. Previously, `S(5).a=7` resulted in a 1-by-5 array.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If the `value` argument is a cell array, all elements must have the same type.

See Also

`cell2struct` | `fieldnames` | `isfield` | `isstruct` | `orderfields` | `rmfield` | `struct2cell` | `struct2table` | `substruct` | `table` | `table2struct`

Topics

"Create Structure Array"
"Access Data in Structure Array"
"Access Data in Nested Structures"

“Generate Field Names from Variables”
“Cell vs. Structure Arrays”

Introduced before R2006a

struct2cell

Convert structure to cell array

Syntax

```
C = struct2cell(S)
```

Description

`C = struct2cell(S)` converts a structure into a cell array. The cell array `C` contains values copied from the fields of `S`.

The `struct2cell` function does not return field names. To return the field names in a cell array, use the `fieldnames` function.

Examples

Return Values and Field Names

Create a structure.

```
S.x = linspace(0,2*pi);
S.y = sin(S.x);
S.title = 'y = sin(x)'
```

```
S = struct with fields:
        x: [1x100 double]
        y: [1x100 double]
    title: 'y = sin(x)'
```

Convert `S` to a cell array.

```
C = struct2cell(S)
```

```
C=3×1 cell
    {1x100 double}
    {1x100 double}
    {'y = sin(x)'}
```

The cell array does not include field names. To return the field names in a cell array, use the `fieldnames` function. `fieldnames` and `struct2cell` return the field names and the values in the same order.

```
fields = fieldnames(S)
```

```
fields = 3x1 cell array
    {'x'    }
    {'y'    }
    {'title'}
```

Input Arguments

S — Input structure array
structure array

Input structure array. `S` can be a structure array of any size. If `S` is an `m`-by-`n` structure array with `p` fields, then `C` is a `p`-by-`m`-by-`n` cell array.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If `S` is a variable-size array of structures, then each field of every element must have the same type. Heterogeneous cell arrays cannot be variable-size.

- For a fixed-size array of structures `S`, if `struct2cell` cannot convert `S` to a homogeneous cell array, then the output cell array is heterogeneous. A heterogeneous output cell array can have a maximum of 1024 elements.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see "Run MATLAB Functions with Distributed Arrays" (Parallel Computing Toolbox).

See Also

`cell` | `cell2struct` | `fieldnames` | `iscell` | `isstruct` | `struct` | `struct2table` | `table2cell`

Topics

"Generate Field Names from Variables"

Introduced before R2006a

struct2table

Convert structure array to table

Syntax

```
T = struct2table(S)
T = struct2table(S,Name,Value)
```

Description

`T = struct2table(S)` converts the structure array, `S`, to a table, `T`. Each field of `S` becomes a variable in `T`.

`T = struct2table(S,Name,Value)` creates a table from a structure array, `S`, with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify row names to include in the table.

Examples

Convert Scalar Structure to Table

Convert a scalar structure to a table using the default options.

Create a structure array, `S`.

```
S.Name = {'CLARK';'BROWN';'MARTIN'};
S.Gender = {'M';'F';'M'};
S.SystolicBP = [124;122;130];
S.DiastolicBP = [93;80;92];
```

```
S
```

```
S = struct with fields:
           Name: {3x1 cell}
```

```
         Gender: {3x1 cell}
     SystolicBP: [3x1 double]
    DiastolicBP: [3x1 double]
```

The scalar structure, S, has four fields, each with three rows.

Convert the structure array to a table.

```
T = struct2table(S)
```

```
T=3×4 table
       Name       Gender    SystolicBP    DiastolicBP
    __________    ______    __________    ___________

    {'CLARK' }    {'M'}        124            93
    {'BROWN' }    {'F'}        122            80
    {'MARTIN'}    {'M'}        130            92
```

The structure field names in S become the variable names in the output table. The size of T is 3-by-4.

Change Name from a variable to row names by modifying the table property, T.Properties.RowNames, and then deleting the variable Name.

```
T.Properties.RowNames = T.Name;
T.Name = [];
```

```
T
```

```
T=3×3 table
              Gender    SystolicBP    DiastolicBP
              ______    __________    ___________

    CLARK     {'M'}        124            93
    BROWN     {'F'}        122            80
    MARTIN    {'M'}        130            92
```

Convert Nonscalar Structure Array to Table

Create a nonscalar structure array, S.

```
S(1,1).Name = 'CLARK';
S(1,1).Gender = 'M';
S(1,1).SystolicBP = 124;
S(1,1).DiastolicBP = 93;

S(2,1).Name = 'BROWN';
S(2,1).Gender = 'F';
S(2,1).SystolicBP = 122;
S(2,1).DiastolicBP = 80;

S(3,1).Name = 'MARTIN';
S(3,1).Gender = 'M';
S(3,1).SystolicBP = 130;
S(3,1).DiastolicBP = 92;

S
```

```
S=3×4 struct
    Name
    Gender
    SystolicBP
    DiastolicBP
```

S is a 3-by-1 structure array with four fields.

Convert the structure array to a table.

```
T = struct2table(S)
```

```
T=3×4 table
       Name       Gender    SystolicBP    DiastolicBP
    __________    ______    __________    ___________

    {'CLARK' }    {'M'}        124            93
    {'BROWN' }    {'F'}        122            80
    {'MARTIN'}    {'M'}        130            92
```

The structure field names in S become the variable names in the output table. The size of T is 3-by-4.

Treat Scalar Structure As Array

Use `'AsArray',true` to create a table from a scalar structure whose fields have different numbers of rows.

Create a scalar structure, `S`, with fields `name`, `billing`, and `test`.

```
S.name = 'John Doe';
S.billing = 127.00;
S.test = [79, 75, 73; 180, 178, 177.5; 220, 210, 205];
S
```

```
S = struct with fields:
       name: 'John Doe'
    billing: 127
       test: [3x3 double]
```

The fields have different numbers of rows. Therefore, you cannot use `struct2table(S)`, which uses `'AsArray',false` by default.

Treat the scalar structure as an array and convert it to a table.

```
T = struct2table(S,'AsArray',true)
```

```
T=1×3 table
        name         billing        test
    ____________     _______    ____________

    {'John Doe'}       127      {3x3 double}
```

`T` contains one row.

Input Arguments

S — Structure array

structure array

Structure array, specified as a scalar structure array.

- If `S` is a scalar structure with `n` fields, all of which have `m` rows, then `T` is an `m`-by-`n` table.

- If S is a nonscalar m-by-1 structure array with n fields, then T is an m-by-n table.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'RowNames',{'row1','row2','row3'}` uses the row names, `row1`, `row2`, and `row3` for the table, `T`.

`RowNames` — Row names for `T`
`{}` (default) | cell array of character vectors | string array

Row names for `T`, specified as the comma-separated pair consisting of `'RowNames'` and a cell array of character vectors or a string array, whose elements are nonempty and distinct.

`AsArray` — Indicator for how to treat scalar structure
`false` (default) | `true` | `0` | `1`

Indicator for how to treat scalar structure, specified as the comma-separated pair consisting of `'AsArray'` and either `false`, `true`, `0`, or `1`.

`true`	`struct2table` converts S to a table with one row and n variables. The variables can be different sizes.
`false`	`struct2table` converts a scalar structure array with n fields into an m-by-n table. Each field must have m rows. This is the default behavior

Output Arguments

`T` — Output table
table

Output table, returned as a table. The table can store metadata such as descriptions, variable units, variable names, and row names. For more information, see the Properties section of `table`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- In generated code, you must specify the `'VariableNames'` name-value pair argument when using this function. For more information, see “Code Generation for Tables” (MATLAB Coder) and “Table Limitations for Code Generation” (MATLAB Coder).

See Also

`array2table` | `cell2table` | `table` | `table2struct`

Topics

“Access Data in Tables”

Introduced in R2013b

structfun

Apply function to each field of scalar structure

Syntax

```
A = structfun(func,S)
A = structfun(func,S,Name,Value)
[A1,...,Am] = structfun( ___ )
```

Description

`A = structfun(func,S)` applies the function `func` to each field of scalar structure `S`, one field at a time. `structfun` then concatenates the outputs from `func` into the column vector `A`. The input argument `func` is a function handle to a function that takes one input argument and returns a scalar. The output from `func` can have any data type, so long as objects of that type can be concatenated. The number of elements in `A` equals the number of fields in `S`.

You cannot specify the order in which `structfun` calculates the elements of `A` or rely on them being done in any particular order.

`A = structfun(func,S,Name,Value)` applies `func` with additional options specified by one or more `Name,Value` pair arguments. For example, to return output values in a structure, specify `'UniformOutput',false`. You can return `A` as a structure when `func` returns values that cannot be concatenated into an array. The returned structure has the same fields as `S`.

`[A1,...,Am] = structfun( ___ )` returns multiple output arrays `A1,...,Am` when `func` returns `m` output values. `func` can return output arguments that have different data types, but the data type of each output must be the same each time `func` is called. You can use this syntax with any of the input arguments of the previous syntaxes.

Examples

Apply Function to Fields of Structure

Create a scalar structure with fields that contain numeric arrays of different sizes.

```
S.f1 = 1:10;
S.f2 = [2; 4; 6];
S.f3 = []
```

```
S = struct with fields:
    f1: [1 2 3 4 5 6 7 8 9 10]
    f2: [3x1 double]
    f3: []
```

Calculate the mean of each numeric array, and return the means in an array.

```
A = structfun(@mean,S)
```

```
A = 3×1

    5.5000
    4.0000
       NaN
```

Return Object Array

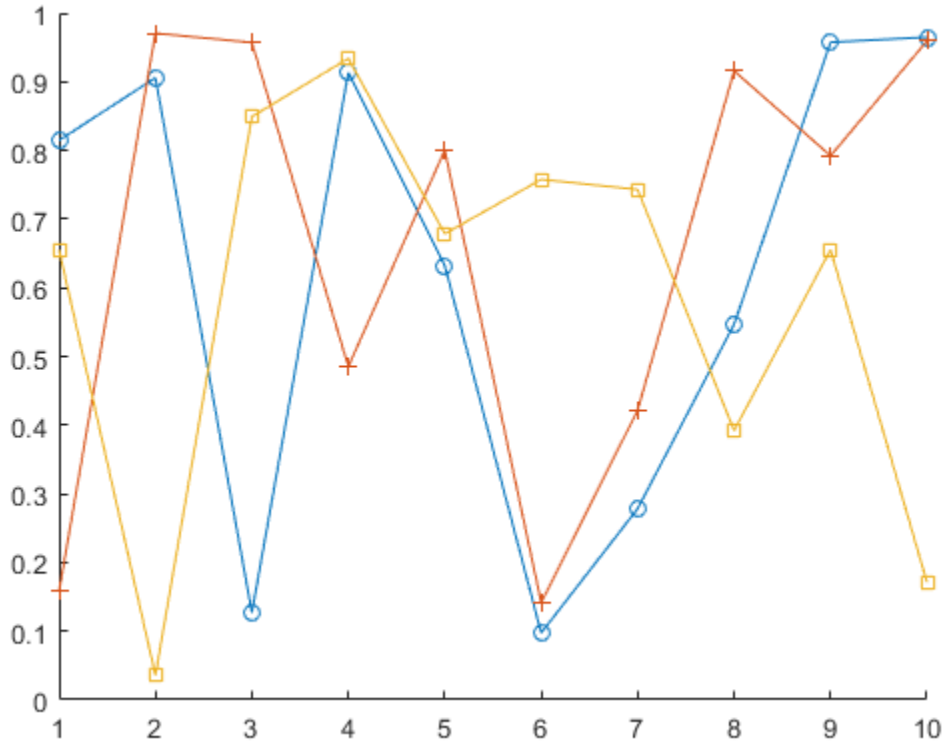
Create a scalar structure in which each field contains an array of random numbers.

```
S.X = rand(1,10);
S.Y = rand(1,10);
S.Z = rand(1,10)
```

```
S = struct with fields:
    X: [0.8147 0.9058 0.1270 0.9134 0.6324 0.0975 0.2785 0.5469 0.9575 0.9649]
    Y: [0.1576 0.9706 0.9572 0.4854 0.8003 0.1419 0.4218 0.9157 0.7922 0.9595]
    Z: [0.6557 0.0357 0.8491 0.9340 0.6787 0.7577 0.7431 0.3922 0.6555 0.1712]
```

Plot the arrays. Return an array of chart line objects from the `plot` function and use them to add different markers to each set of data points. `structfun` can return arrays of any data type, so long as objects of that data type can be concatenated.

```
figure
hold on
p = structfun(@plot,S);
p(1).Marker = 'o';
p(2).Marker = '+';
p(3).Marker = 's';
hold off
```



Return Scalar Structure

Create a scalar structure with fields that contain matrices.

```
S.f1 = 1:10;
S.f2 = [2 3; 4 5; 6 7];
S.f3 = rand(4,4)
```

```
S = struct with fields:
    f1: [1 2 3 4 5 6 7 8 9 10]
    f2: [3x2 double]
    f3: [4x4 double]
```

Calculate the means of each matrix. `mean` returns vectors containing the mean of each column, so the means cannot be returned as an array. To return the means in a structure, specify the `'UniformOutput',false` name-value pair.

```
A = structfun(@mean,S,'UniformOutput',false)
```

```
A = struct with fields:
    f1: 5.5000
    f2: [4 5]
    f3: [0.6902 0.3888 0.7627 0.5962]
```

Return Multiple Output Arrays

Create a scalar structure.

```
S.f1 = 1:10;
S.f2 = [2 3; 4 5; 6 7];
S.f3 = rand(4,4)
```

```
S = struct with fields:
    f1: [1 2 3 4 5 6 7 8 9 10]
    f2: [3x2 double]
    f3: [4x4 double]
```

Calculate the sizes of each array in `S`. The number of rows and columns are each in 3-by-1 numeric arrays.

```
[nrows,ncols] = structfun(@size,S)
```

```
nrows = 3×1
```

```
     1
     3
     4


ncols = 3×1

    10
     2
     4
```

Input Arguments

`func` — Function to apply
function handle

Function to apply to the fields of the input scalar structure, specified as a function handle.

`func` can correspond to more than one function file and therefore can represent a set of overloaded functions. In these cases, MATLAB determines which function to call based on the class of the input arguments.

Example: `A = structfun(@max,S)` returns the maximum of each field of `S`.

`S` — Input structure
scalar structure

Input structure, specified as a scalar structure.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `A = structfun(@mean,S,'UniformOutput',false)` returns the outputs from `mean` in a structure with the same fields as `S`.

`UniformOutput` — True or false
`true` (default) | `false`

True or false, specified as the comma-separated pair consisting of `'UniformOuput'` and either `true` (`1`) or `false` (`0`).

Value of `'UniformOutput'`	Description
`true` (`1`)	`func` must return scalars that `structfun` concatenates into a column vector.
`false` (`0`)	`structfun` returns the outputs of `func` in one or more scalar structures. The output scalar structures have the same fields as the input scalar structure. The outputs of `func` can have any data type.

`ErrorHandler` — Function to catch errors
function handle

Function to catch errors, specified as the comma-separated pair consisting of `'ErrorHandler'` and a function handle. If `func` throws an error, then the error handler specified by `'ErrorHandler'` catches the error and takes the action specified in the function. The error handler either must throw an error or return the same number of outputs as `func`. If the value of `'UniformOutput'` is true, then the output arguments of the error handler must be scalars and have the same data type as the outputs of `func`.

The first input argument of the error handler is a structure with these fields:

- `identifier` — Error identifier
- `message` — Error message text
- `index` — Linear index into the input arrays at which `func` threw the error

The remaining input arguments to the error handler are the input arguments for the call to `func` that made `func` throw the error.

Suppose `func` returns two doubles as output arguments. You can specify the error handler as `'ErrorHandler',@errorFunc`, where `errorFunc` is a function that raises a warning and returns two output arguments.

```
function [A,B] = errorFunc(S,varargin)
    warning(S.identifier, S.message);
```

```
    A = NaN;
    B = NaN;
end
```

If you do not specify `'ErrorHandler'`, then `structfun` rethrows the error thrown by `func`.

Output Arguments

A — Output array
column vector of any data type | scalar structure

Output array, returned as a column vector of any data type or as a scalar structure.

By default, `structfun` concatenates the outputs from `func` into a column vector. `func` must return scalars. If `func` returns objects, then the class that the objects belong to must meet these requirements.

- Support assignment by linear indexing into the object array
- Have a `reshape` method that returns an array that has the same size as the input

If the value of the `'UniformOutput'` name-value pair argument is `false` (`0`), then `structfun` returns outputs as fields of a scalar structure. In that case, the outputs from `func` can have any sizes and different data types.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The `ErrorHandler` option is not supported.
- The number of outputs must be less than or equal to three.

See Also

`arrayfun` | `cell2mat` | `cellfun` | `spfun` | `splitapply`

Topics

"Anonymous Functions"
"Create Function Handle"

Introduced before R2006a

strvcat

(Not recommended) Concatenate strings vertically

Note `strvcat` is not recommended. Use `char` instead. Unlike `strvcat`, the `char` function does not ignore empty character vectors.

Syntax

```
S = strvcat(t1, t2, t3, ...)
S = strvcat(c)
```

Description

`S = strvcat(t1, t2, t3, ...)` forms the character array `S` containing the character arrays `t1,t2,t3,...` as rows. Spaces are appended to each input argument as necessary so that the rows of `S` have the same number of characters. Empty arguments are ignored.

`S = strvcat(c)` when `c` is a cell array of character vectors, passes each element of `c` as an input to `strvcat`. Empty character vectors in the input are ignored.

Examples

The command `strvcat('Hello','Yes')` is the same as `['Hello';'Yes ']`, except that `strvcat` performs the padding automatically.

```
t1 = 'first'; t2 = 'character'; t3 = 'array'; t4 = 'second';

S1 = strvcat(t1, t2, t3)

S1 =

  3×9 char array
```

```
    'first    '
    'character'
    'array    '

S2 = strvcat(t4, t2, t3)

S2 =

  3×9 char array

    'second   '
    'character'
    'array    '

S3 = strvcat(S1, S2)

S3 =

  6×9 char array

    'first    '
    'character'
    'array    '
    'second   '
    'character'
    'array    '
```

Tips

If each text parameter, `ti`, is itself a character array, `strvcat` appends them vertically to create arbitrarily large character arrays.

See Also

`cat` | `char` | `horzcat` | `int2str` | `mat2str` | `num2str` | `strcat` | `vertcat`

Introduced before R2006a

Style Properties

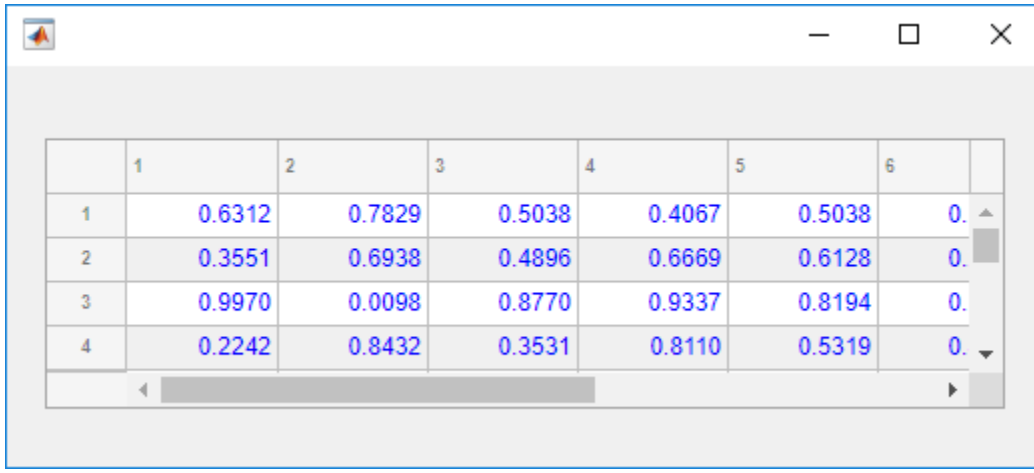
Style for table UI component

Description

Use `Style` objects to create color, font, and alignment styles for cells in table UI components. The table UI component must be one that is parented to a figure created with the `uifigure` function, or to one of its child containers. You can create a `Style` object using the `uistyle` function, specify its properties, and then apply it to the table using the `addStyle` function. You can remove styles from a table using the `removeStyle` function.

`Style` objects that you add to the table UI component take visual precedence over properties you set on the `Table` object, no matter the order in which you created them. For example, in this code the blue font color displays in the table even though red foreground color was set on the `Table` object last.

```
uit = uitable(uifigure,'Data',rand(100,10));

s = uistyle;
s.FontColor = 'blue';
addStyle(uit,s);

uit.ForegroundColor = 'red';
```



	1	2	3	4	5	6
1	0.6312	0.7829	0.5038	0.4067	0.5038	0.
2	0.3551	0.6938	0.4896	0.6669	0.6128	0.
3	0.9970	0.0098	0.8770	0.9337	0.8194	0.
4	0.2242	0.8432	0.3531	0.8110	0.5319	0.

Properties

Color

`BackgroundColor` — Background color

`[]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Therefore, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`FontColor` — Font color

[ ] (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Font and Alignment

`FontName` — Font name

`''` (default) | system-supported font name

Font name, specified as a system-supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

`FontWeight` — Font weight
`''` (default) | `'normal'` | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

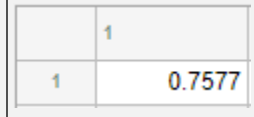
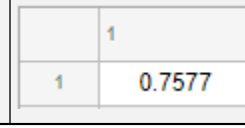
Not all fonts have a bold font weight. For fonts that do not, specifying `'bold'` results in the normal font weight.

`FontAngle` — Font angle
`''` (default) | `'normal'` | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Not all fonts have an italic font angle. For fonts that do not, specifying `'italic'` results in the normal font angle.

`HorizontalAlignment` — Horizontal alignment of cell text
`''` (default) | `'left'` | `'right'` | `'center'`

Horizontal alignment of cell text, specified as one of the values in the table.

Value	Example
`'left'`	
`'right'`	
`'center'`	

If you add a `Style` object to the table UI component that specifies horizontal alignment, it takes precedence over the justifications associated with cell format values that you specify for the `ColumnFormat` property on the `Table` object. The `ColumnFormat` property still converts values as it normally would.

See Also

Functions
`addStyle` | `removeStyle` | `uistyle` | `uitable`

Properties
Table Properties

Introduced in R2019b

sub2ind

Convert subscripts to linear indices

Syntax

```
ind = sub2ind(sz,row,col)
ind = sub2ind(sz,I1,I2,...,In)
```

Description

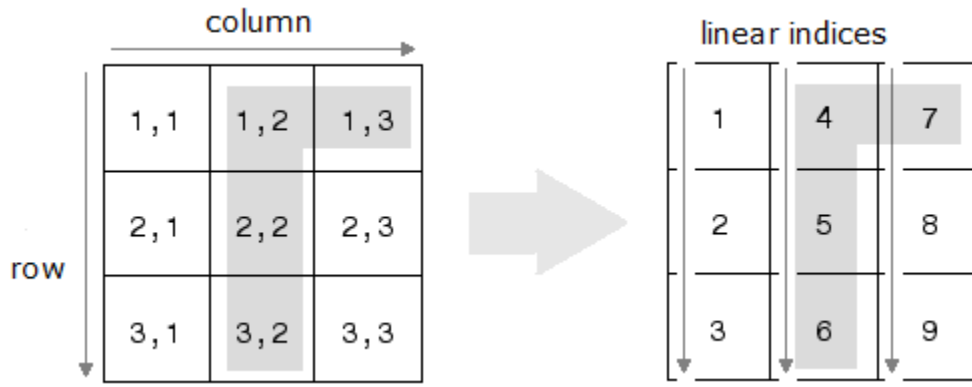
`ind = sub2ind(sz,row,col)` returns the linear indices `ind` corresponding to the row and column subscripts in `row` and `col` for a matrix of size `sz`. Here `sz` is a vector with two elements, where `sz(1)` specifies the number of rows and `sz(2)` specifies the number of columns.

`ind = sub2ind(sz,I1,I2,...,In)` returns the linear indices `ind` corresponding to multidimensional subscripts in `n` arrays `I1,I2,...,In` for a multidimensional array of size `sz`. Here `sz` is a vector with `n` elements that specifies the size of each array dimension.

Examples

Convert Subscripts to Linear Indices for Matrix

The mapping from subscripts (indexing by position) to linear indices for a 3-by-3 matrix can be illustrated as in the following.



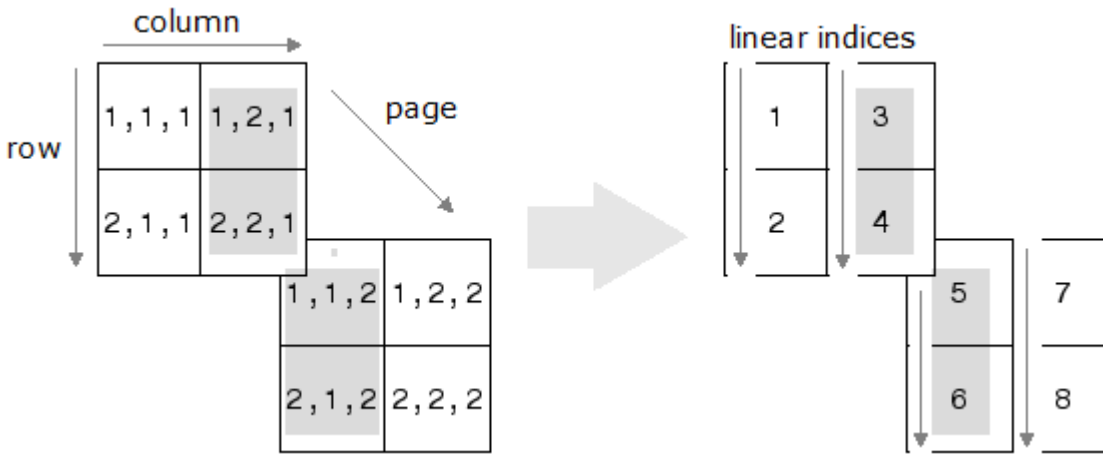
Specify the row and column subscripts in a 3-by-3 matrix. Convert the subscripts to linear indices.

```
row = [1 2 3 1];
col = [2 2 2 3];
sz = [3 3];
ind = sub2ind(sz,row,col)
```

```
ind = 1×4

     4     5     6     7
```

Convert Subscripts to Linear Indices for Three-Dimensional Array

The mapping from subscripts to linear indices for a 2-by-2-by-2 array can be illustrated as in the following.



Specify the row, column, and page subscripts in a 2-by-2-by-2 array. Convert the subscripts to linear indices.

```
I1 = [1 2 1 2];
I2 = [2 2 1 1];
I3 = [1 1 2 2];
sz = [2 2 2];
ind = sub2ind(sz,I1,I2,I3)
```

```
ind = 1×4

     3     4     5     6
```

Access Element of 3-D Array

Convert a subscript index of a 3-D array to a single linear index.

Create an array, and find the linear index corresponding to the element in the (2,1,2) position.

```
A = rand(3,4,2);
linearInd = sub2ind(size(A),2,1,2)
```

```
linearInd = 14
```

Check that both index versions refer to the same element.

```
A(2,1,2)
```

```
ans = 0.4854
```

```
A(14)
```

```
ans = 0.4854
```

Input Arguments

`sz` — Size of array

vector of positive integers

Size of array, specified as a vector of positive integers. Each element of this vector indicates the size of the corresponding dimension. For example, `[2 3 4]` defines a 2-by-3-by-4 array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`row` — Row subscripts

scalar | vector | matrix | multidimensional array

Row subscripts, specified as a scalar, vector, matrix, or multidimensional array. The size of `row` must be the same as the size of `col`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`col` — Column subscripts

scalar | vector | matrix | multidimensional array

Column subscripts, specified as a scalar, vector, matrix, or multidimensional array. The size of `col` must be the same as the size of `row`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`I1,I2,...,In` — Multidimensional subscripts

scalar | vector | matrix | multidimensional array

Multidimensional subscripts, specified as a scalar, vector, matrix, or multidimensional array. Each array `I1,I2,…,In` must have the same size.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`ind` — Linear indices
scalar | vector | matrix | multidimensional array

Linear indices, specified as a scalar, vector, matrix, or multidimensional array. The size of `ind` is the same as the size of the input `row`, `col`, or `I1,I2,…,In`.

Data Types: `double`

Algorithms

For an array `A`, if `ind = sub2ind(size(A),I1,…,In)`, then `A(ind(k)) = A(I1(k),…,In(k))` for all `k`.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The first argument must be a valid size vector. Code generation does not support size vectors for arrays with more than `intmax` elements.

- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- The first argument must be a valid size vector. Code generation does not support size vectors for arrays with more than `intmax` elements.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`find` | `ind2sub` | `size`

Topics

“Array Indexing”
“Multidimensional Arrays”

Introduced before R2006a

subplot

Create axes in tiled positions

Syntax

```
subplot(m,n,p)
subplot(m,n,p,'replace')
subplot(m,n,p,'align')
subplot(m,n,p,ax)

subplot('Position',pos)

subplot( ___ ,Name,Value)
ax = subplot( ___ )

subplot(ax)
```

Description

`subplot(m,n,p)` divides the current figure into an `m`-by-`n` grid and creates axes in the position specified by `p`. MATLAB numbers subplot positions by row. The first subplot is the first column of the first row, the second subplot is the second column of the first row, and so on. If axes exist in the specified position, then this command makes the axes the current axes.

`subplot(m,n,p,'replace')` deletes existing axes in position `p` and creates new axes.

`subplot(m,n,p,'align')` creates new axes so that the plot boxes are aligned. This option is the default behavior.

`subplot(m,n,p,ax)` converts the existing axes, `ax`, into a subplot in the same figure.

`subplot('Position',pos)` creates axes in the custom position specified by `pos`. Use this option to position a subplot that does not align with grid positions. Specify `pos` as a four-element vector of the form `[left bottom width height]`. If the new axes overlap existing axes, then the new axes replace the existing axes.

`subplot( ___ ,Name,Value)` modifies axes properties using one or more name-value pair arguments. Set axes properties after all other input arguments.

`ax = subplot( ___ )` creates an `Axes` object, `PolarAxes` object, or `GeographicAxes` object. Use `ax` to make future modifications to the axes.

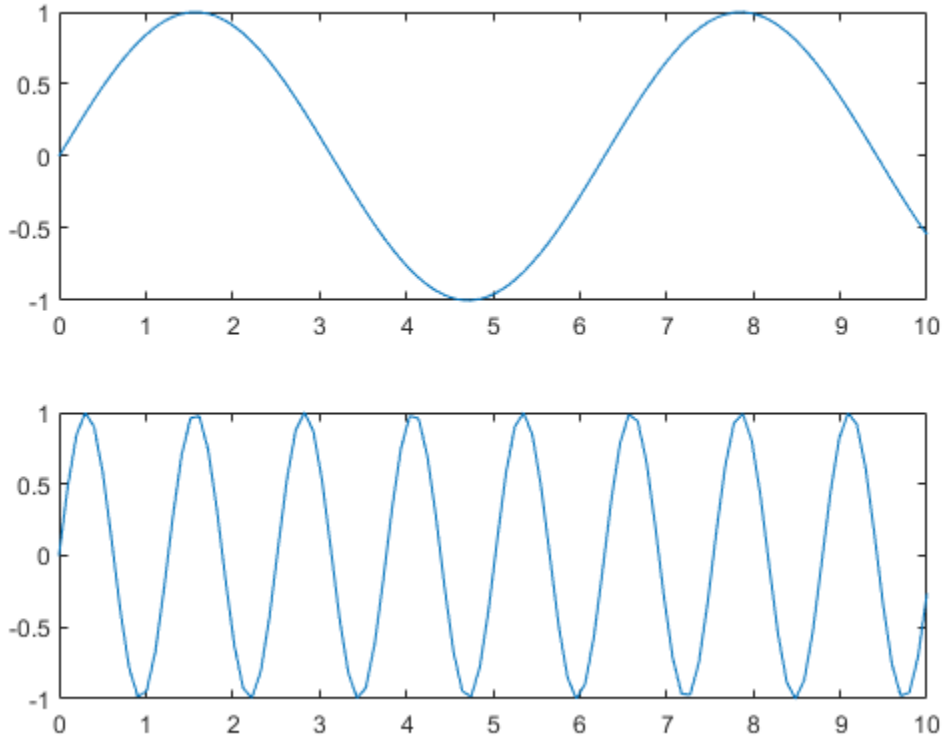
`subplot(ax)` makes the axes specified by `ax` the current axes for the parent figure. This option does not make the parent figure the current figure if it is not already the current figure.

Examples

Upper and Lower Subplots

Create a figure with two stacked subplots. Plot a sine wave in each one.

```
subplot(2,1,1);
x = linspace(0,10);
y1 = sin(x);
plot(x,y1)

subplot(2,1,2);
y2 = sin(5*x);
plot(x,y2)
```

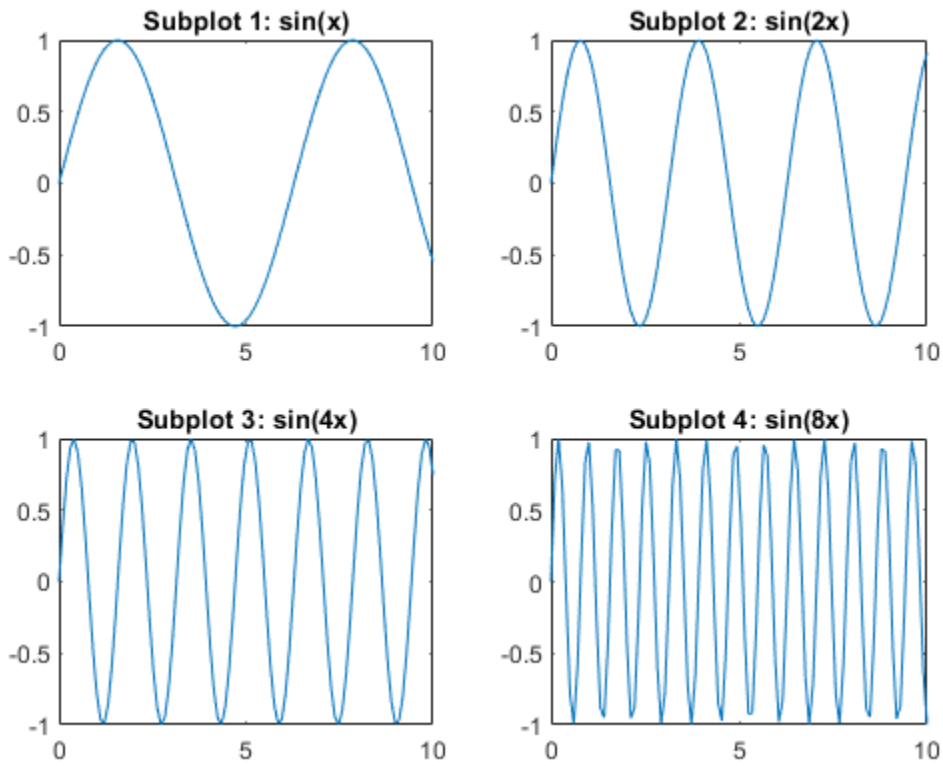


Quadrant of Subplots

Create a figure divided into four subplots. Plot a sine wave in each one and title each subplot.

```
subplot(2,2,1)
x = linspace(0,10);
y1 = sin(x);
plot(x,y1)
title('Subplot 1: sin(x)')
```

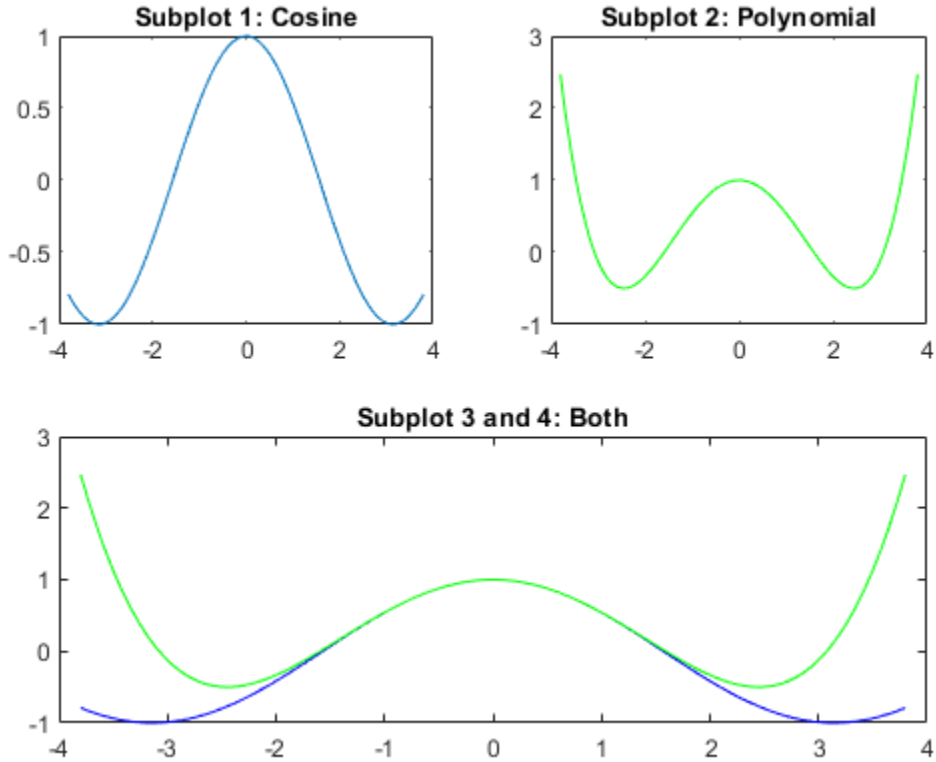
```
subplot(2,2,2)
y2 = sin(2*x);
plot(x,y2)
title('Subplot 2: sin(2x)')

subplot(2,2,3)
y3 = sin(4*x);
plot(x,y3)
title('Subplot 3: sin(4x)')

subplot(2,2,4)
y4 = sin(8*x);
plot(x,y4)
title('Subplot 4: sin(8x)')
```



Subplots with Different Sizes

Create a figure containing with three subplots. Create two subplots across the upper half of the figure and a third subplot that spans the lower half of the figure. Add titles to each subplot.

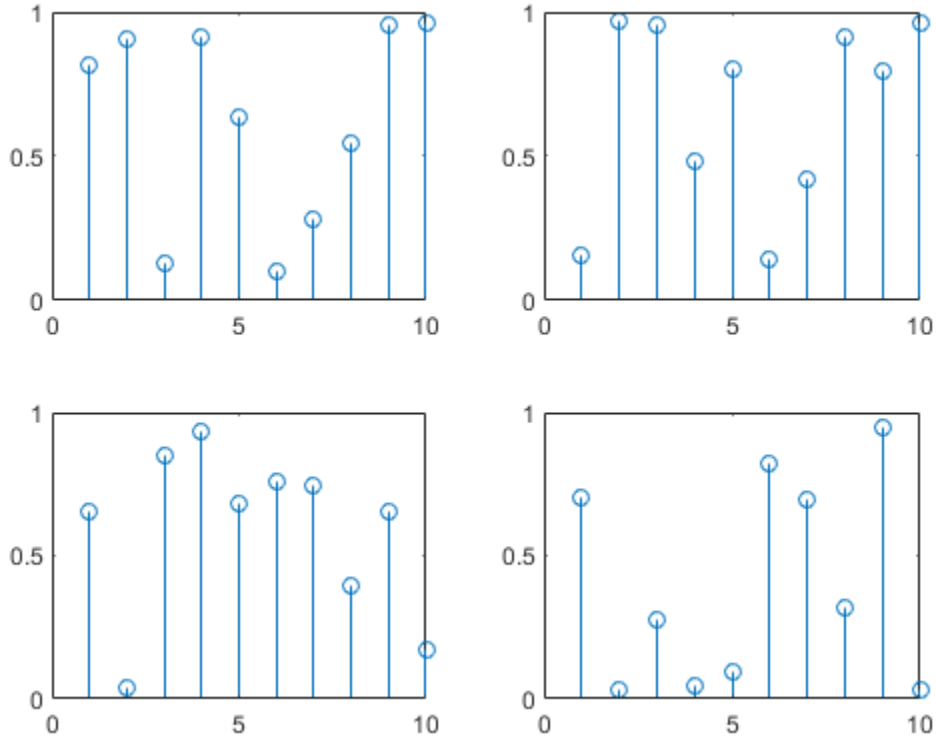
```
subplot(2,2,1);
x = linspace(-3.8,3.8);
y_cos = cos(x);
plot(x,y_cos);
title('Subplot 1: Cosine')

subplot(2,2,2);
y_poly = 1 - x.^2./2 + x.^4./24;
plot(x,y_poly,'g');
title('Subplot 2: Polynomial')

subplot(2,2,[3,4]);
plot(x,y_cos,'b',x,y_poly,'g');
title('Subplot 3 and 4: Both')
```



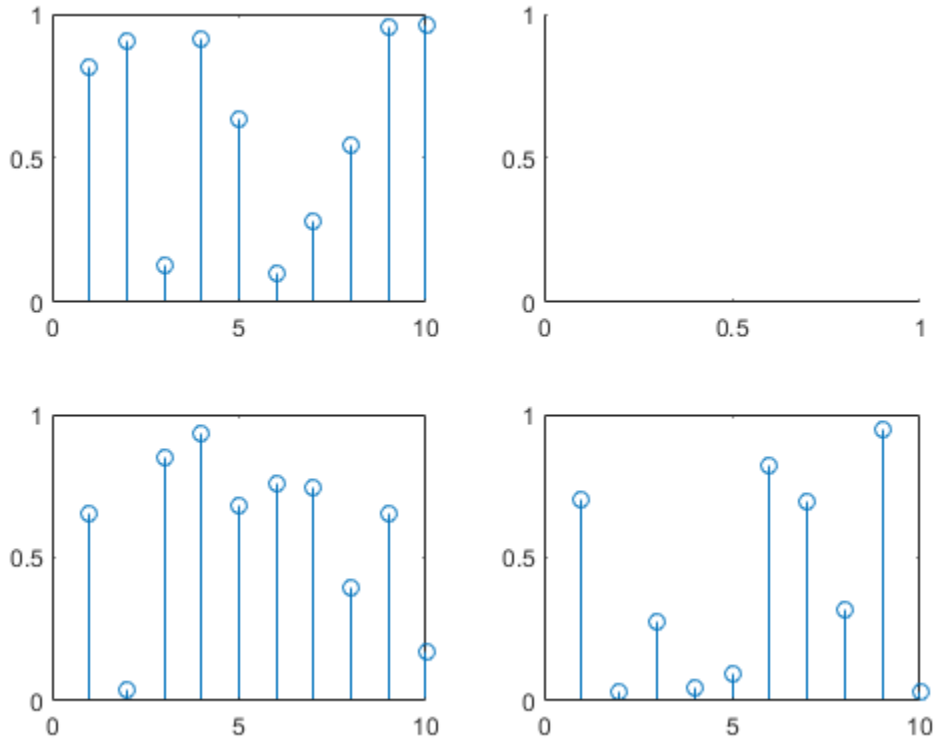
Replace Subplot with Empty Axes

Create a figure with four stem plots of random data. Then replace the second subplot with empty axes.

```
for k = 1:4
    data = rand(1,10);
    subplot(2,2,k)
    stem(data)
end
```




```
subplot(2,2,2,'replace')
```

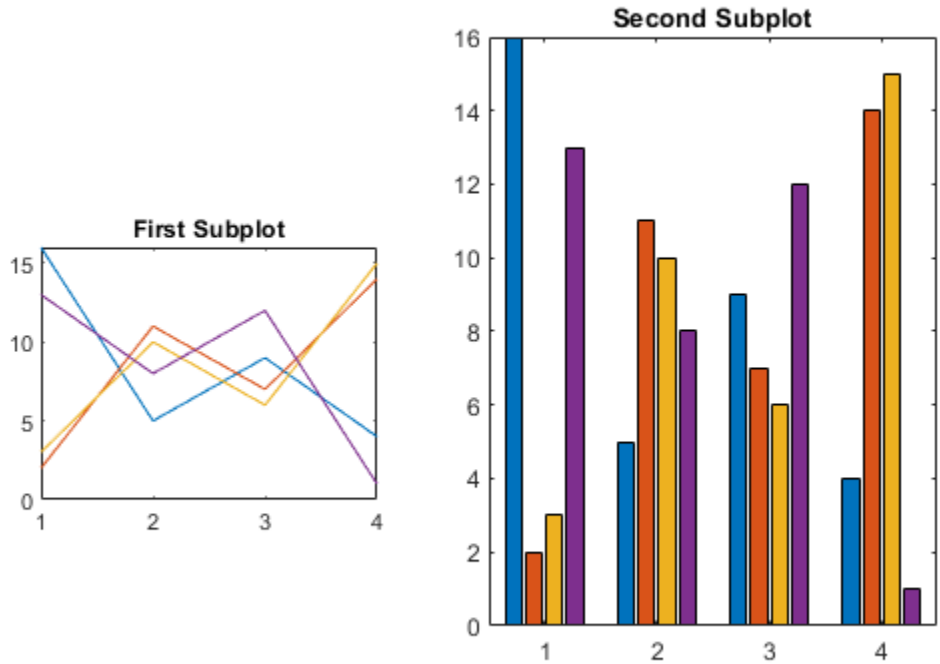


Subplots at Custom Positions

Create a figure with two subplots that are not aligned with grid positions. Specify a custom position for each subplot.

```
pos1 = [0.1 0.3 0.3 0.3];
subplot('Position',pos1)
y = magic(4);
plot(y)
title('First Subplot')
```

```
pos2 = [0.5 0.15 0.4 0.7];
subplot('Position',pos2)
bar(y)
title('Second Subplot')
```

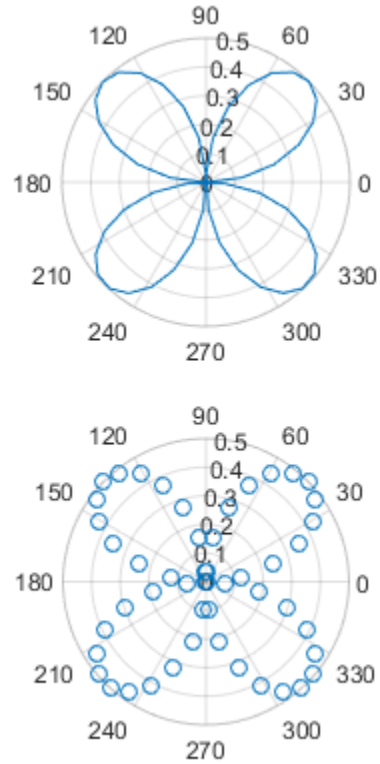


Create Subplots with Polar Axes

Create a figure with two polar axes. Create a polar line chart in the upper subplot and a polar scatter chart in the lower subplot.

```
figure
ax1 = subplot(2,1,1,polaraxes);
```

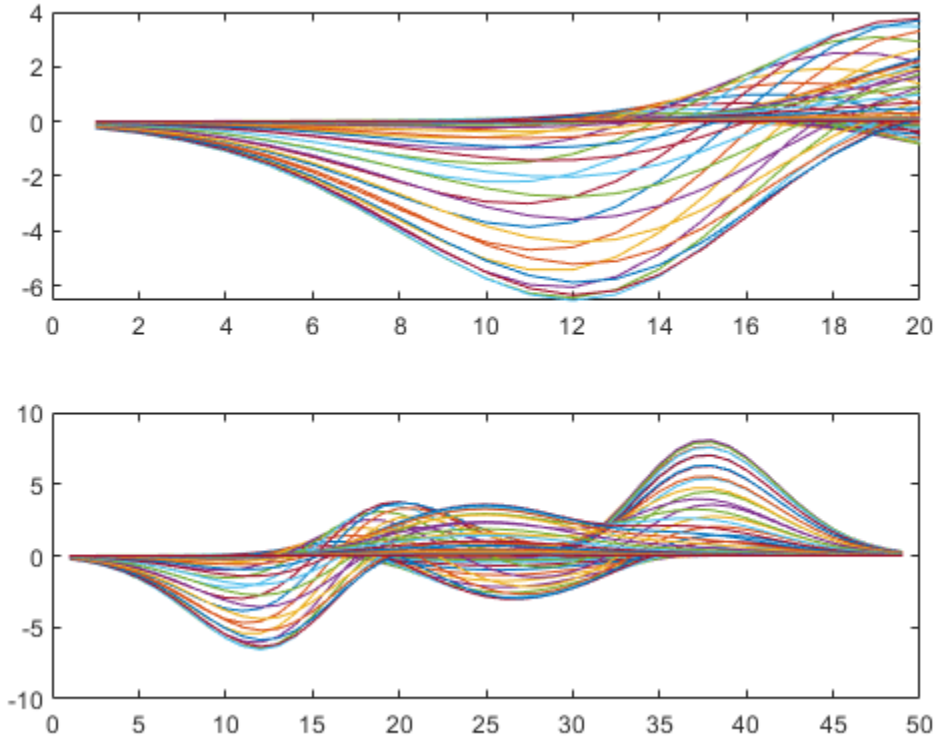
```
theta = linspace(0,2*pi,50);
rho = sin(theta).*cos(theta);
polarplot(ax1,theta,rho)

ax2 = subplot(2,1,2,polaraxes);
polarscatter(ax2,theta,rho)
```



Modify Axes Properties After Creation

Create a figure with two subplots. Assign the `Axes` objects to the variables `ax1` and `ax2`. Specify the `Axes` objects as inputs to the plotting functions to ensure that the functions plot into a specific subplot.

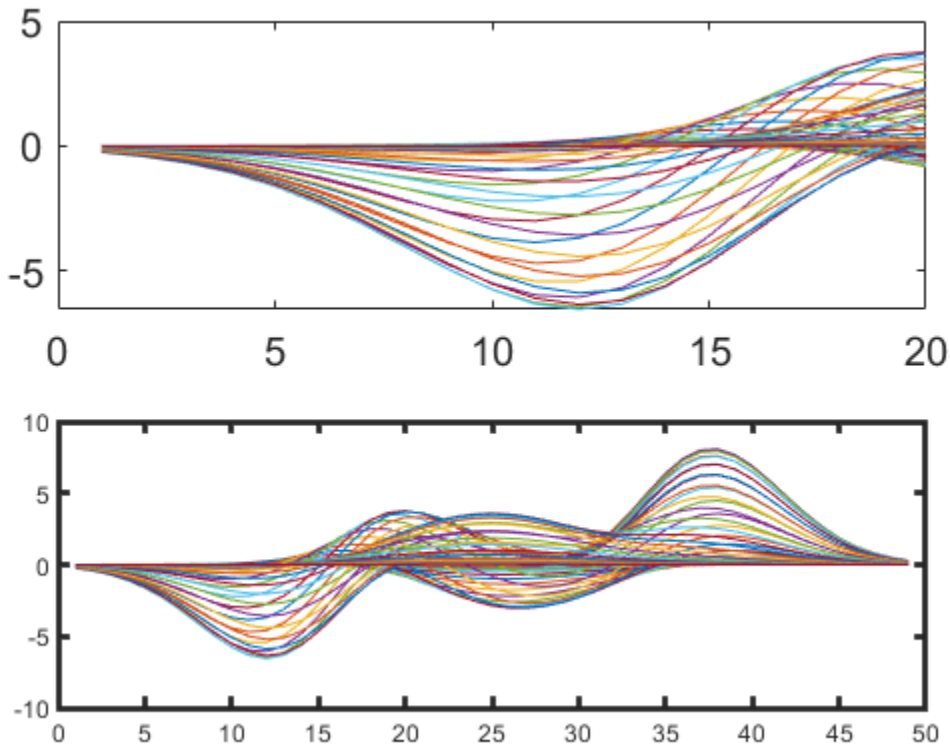
```
ax1 = subplot(2,1,1);
Z = peaks;
plot(ax1,Z(1:20,:))

ax2 = subplot(2,1,2);
plot(ax2,Z)
```



Modify the axes by setting properties of the `Axes` objects. Change the font size for the upper subplot and the line width for the lower subplot. Some plotting functions set axes properties. Execute plotting functions before specifying axes properties to avoid overriding existing axes property settings.

Note: Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead.

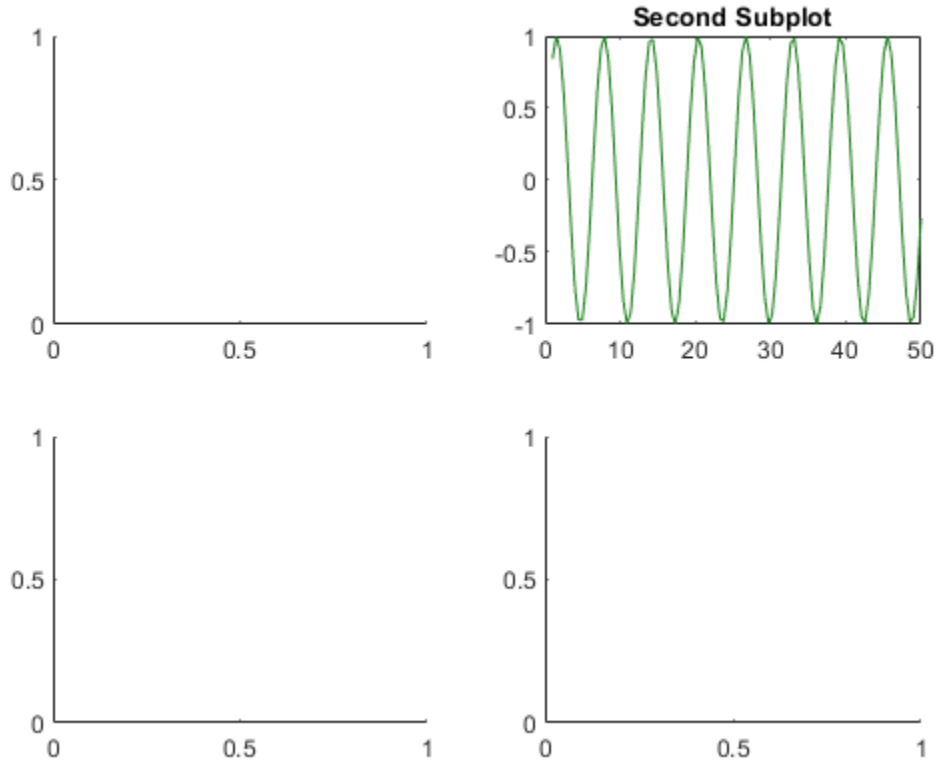
```
ax1.FontSize = 15;
ax2.LineWidth = 2;
```



Make Subplot the Current Axes

Create a figure with multiple subplots. Store the `Axes` objects in vector `ax`. Then make the second subplot the current axes. Create a line chart and change the axis limits for the second subplot. By default, graphics functions target the current axes.

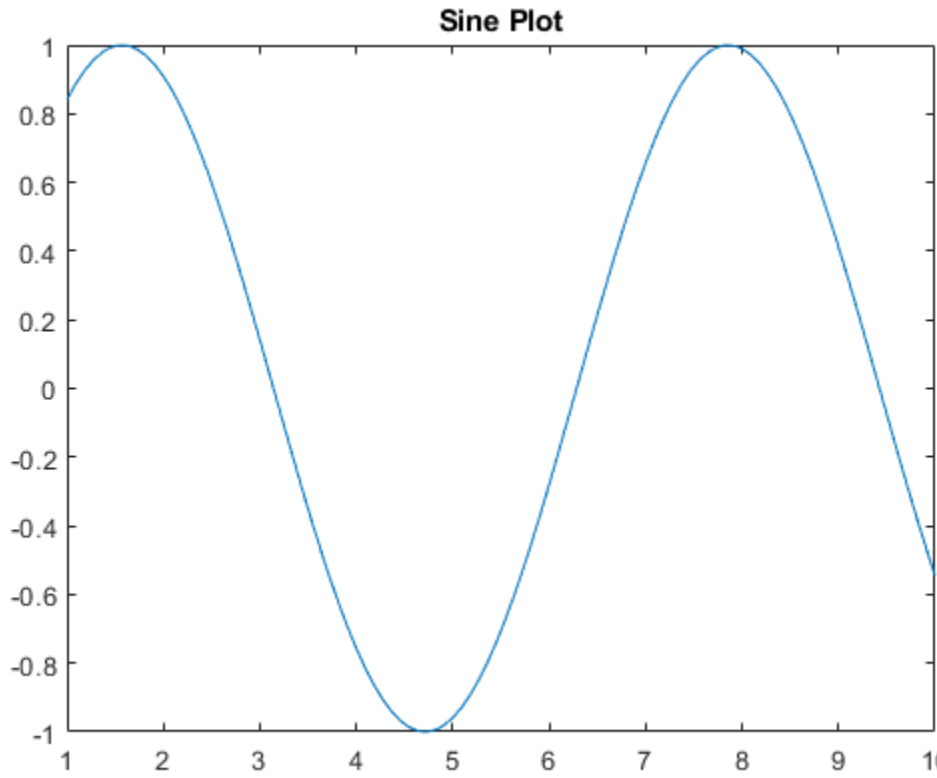
```
for k = 1:4
    ax(k) = subplot(2,2,k);
end

subplot(ax(2))
x = linspace(1,50);
y = sin(x);
plot(x,y,'Color',[0.1, 0.5, 0.1])
title('Second Subplot')
axis([0 50 -1 1])
```



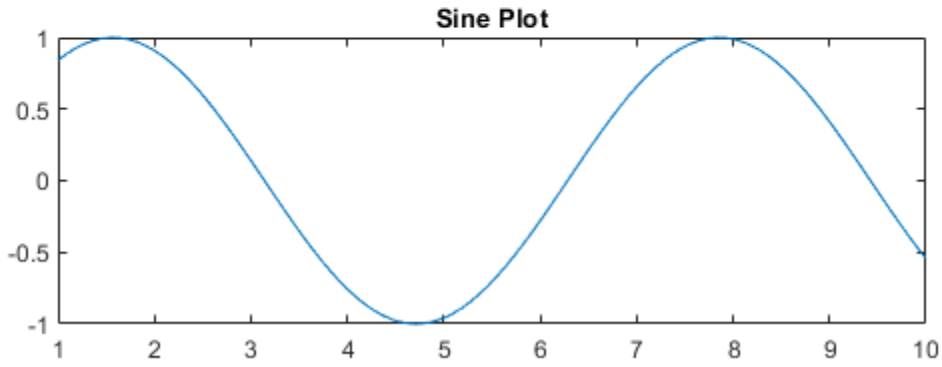
Convert Existing Axes to Subplot

Create a line chart. Then convert the axes so that it is the lower subplot of the figure. The subplot function uses the figure in which the original axes existed.

```
x = linspace(1,10);
y = sin(x);
plot(x,y)
title('Sine Plot')
```




```
ax = gca;
subplot(2,1,2,ax)
```

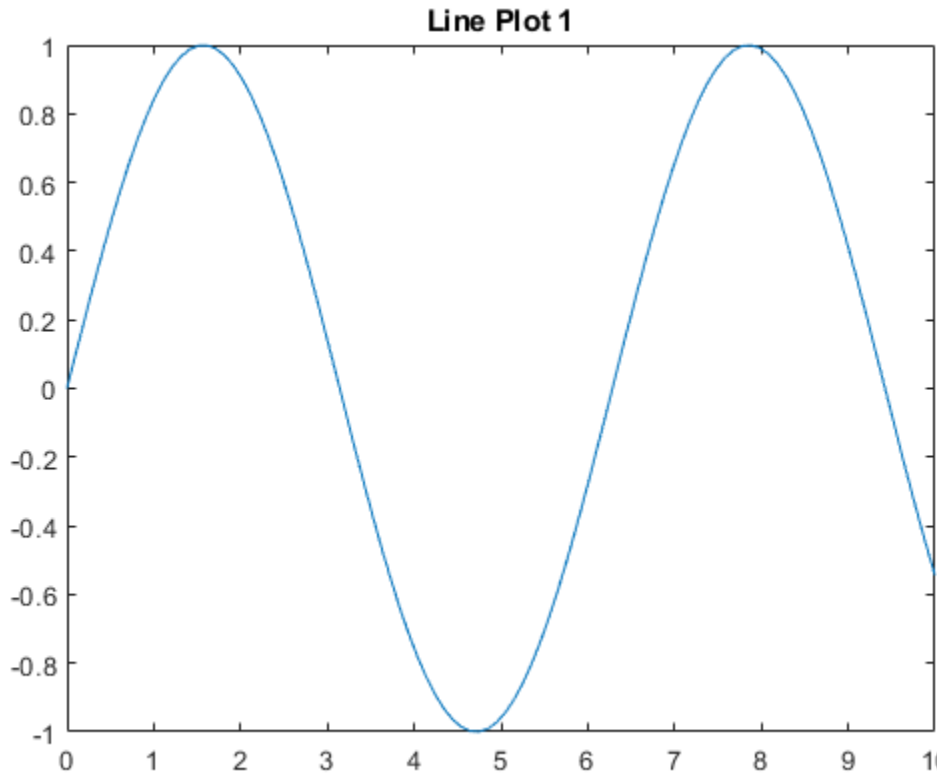


Convert Axes in Separate Figures to Subplots

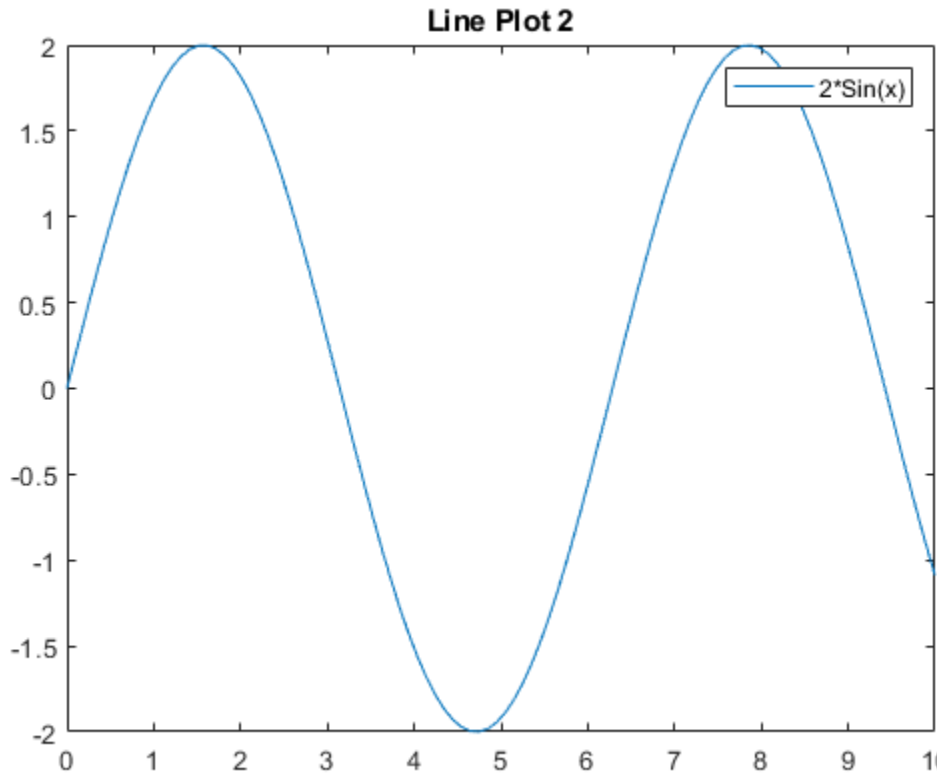
Combine axes that exist in separate figures in a single figure with subplots.

Create two plots in two different figures. Assign the `Axes` objects to the variables `ax1` and `ax2`. Assign the `Legend` object to the variable `lgd`.

```
figure
x = linspace(0,10);
y1 = sin(x);
plot(x,y1)
title('Line Plot 1')
```



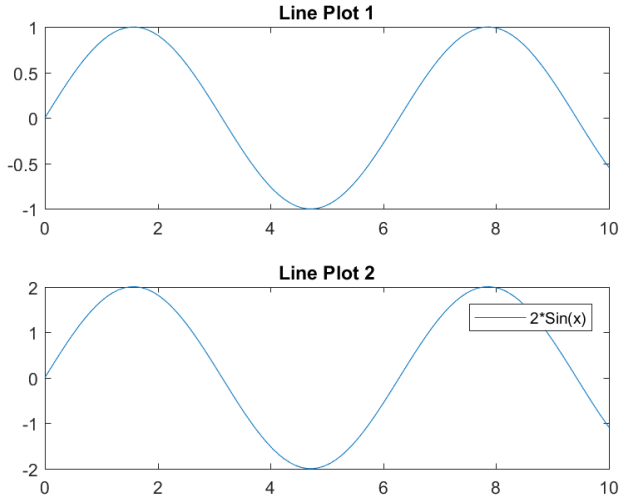

```
ax1 = gca;

figure
y2 = 2*sin(x);
plot(x,y2)
title('Line Plot 2')
lgd = legend('2*Sin(x)');
```




```
ax2 = gca;
```

Create copies of the two `Axes` objects using `copyobj`. Specify the parents of the copied axes as a new figure. Since legends and colorbars do not get copied with the associated axes, copy the legend with the axes.

```
fnew = figure;
ax1_copy = copyobj(ax1,fnew);
subplot(2,1,1,ax1_copy)

copies = copyobj([ax2,lgd],fnew);
ax2_copy = copies(1);
subplot(2,1,2,ax2_copy)
```



Input Arguments

`m` — Number of grid rows
1 (default) | positive integer

Number of grid rows, specified as a positive integer.

Data Types: `single` | `double`

`n` — Number of grid columns
1 (default) | positive integer

Number of grid columns, specified as a positive integer.

Data Types: `single` | `double`

`p` — Grid position for new axes
scalar | vector

Grid position for the new axes, specified as a scalar or vector of positive integers.

- If `p` is a scalar positive integer, then `subplot` creates a subplot in grid position `p`.

- If `p` is a vector of positive integers, then `subplot` creates a subplot that spans the grid positions listed in `p`.

Example: `subplot(2,3,1)` creates a subplot in position 1.

Example: `subplot(2,3,[2,5])` creates a subplot spanning positions 2 and 5.

Example: `subplot(2,3,[2,6])` creates a subplot spanning positions 2, 3, 5, and 6.

Data Types: `single` | `double`

`pos` — Custom position for new axes
four-element vector

Custom position for the new axes, specified as a four-element vector of the form `[left bottom width height]`.

- The `left` and `bottom` elements specify the position of the bottom-left corner of the subplot in relation to the bottom-left corner of the figure.
- The `width` and `height` elements specify the subplot dimensions.

Specify values between `0` and `1` that are normalized with respect to the interior of the figure.

Note When using a script to create subplots, MATLAB does not finalize the `Position` property value until either a `drawnow` command is issued or MATLAB returns to await a user command. The `Position` property value for a subplot is subject to change until the script either refreshes the plot or exits.

Example: `subplot('Position',[0.1 0.1 0.45 0.45])`

Data Types: `single` | `double`

`ax` — Existing axes to make current or convert to subplot
`Axes` object | `PolarAxes` object | `GeographicAxes` object | graphics object

Existing axes to make current or convert to a subplot, specified as an `Axes` object, a `PolarAxes` object, a `GeographicAxes` object, or a graphics object with an `ActivePositionProperty` property, such as a `HeatmapChart` object.

To create empty polar or geographic axes in a subplot position, specify `ax` as the `polaraxes` or `geoaxes` function. For example, `subplot(2,1,2,polaraxes)`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `subplot(m,n,p,'XGrid','on')`

Some plotting functions override property settings. Consider setting axes properties after plotting. The properties you can set depend on the type of axes:

- For Cartesian axes, see Axes.
- For polar axes, see PolarAxes Properties.
- For geographic axes, see GeographicAxes Properties.

.

Tips

- To clear the contents of the figure, use `clf`. For example, you might clear the existing subplot layout from the figure before creating a new subplot layout.
- To overlay axes, use the `axes` command instead. The `subplot` function deletes existing axes that overlap new axes. For example, `subplot('Position', [.35 .35 .3 .3])` deletes any underlying axes, but `axes('Position', [.35 .35 .3 .3])` positions new axes in the middle of the figure without deleting underlying axes.
- `subplot(111)` is an exception and not identical in behavior to `subplot(1,1,1)`. For reasons of backwards compatibility, `subplot(111)` is a special case of subplot that does not immediately create axes, but sets up the figure so that the next graphics command executes `clf reset`. The next graphics command deletes all the figure children and creates new axes in the default position. `subplot(111)` does not return an `Axes` object and an error occurs if code specifies a return argument.

Alternative Functionality

Use the `tiledlayout` and `nexttile` functions to create a configurable tiling of plots. The configuration options include:

- Control over the spacing between the plots and around the edges of the layout
- An option for a shared title at the top of the layout
- Options for shared *x*- and *y*-axis labels
- An option to control whether the tiling has a fixed size or variable size that can reflow

For more information, see “Combine Multiple Plots”.

See Also

Functions
`axes` | `cla` | `clf` | `figure` | `gca` | `nexttile` | `tiledlayout`

Properties
Axes | GeographicAxes Properties | PolarAxes Properties

Topics

“Combine Multiple Plots”

Introduced before R2006a

subsasgn

Redefine subscripted assignment

Syntax

```
A = subsasgn(A,S,B)
```

Description

`A = subsasgn(A,S,B)` called by MATLAB for the syntax `A(i) = B`, `A{i} = B`, or `A.i = B` when `A` is an object.

MATLAB uses the built-in `subsasgn` function to interpret indexed assignment statements. Modify the indexed assignment behavior of classes by overloading `subsasgn` in the class.

Note You must call `subsasgn` with an output argument. `subsasgn` does not modify the object used in the indexing operation (the first input argument). You must assign the output to obtain a modified object.

Input Arguments

A

Object used in indexing operation

S

Structure with two fields, `type` and `subs`. For compound indexing expressions, `S` is an array of structures, one for each level of indexing.

- `type` is a `char` vector or string containing `()`, `{}`, or `.`, indicating the type of indexing used.
- `subs` is a cell array, character array, or string array containing the actual subscripts.

B

Value being assigned (right side of assignment statement)

Output Arguments

A

Result of the assignment statement, which is the modified object passed in as the first argument.

If your implementation of a `subsasgn` method assigns more than one value, use `varargin` for the third input argument. For more information, see “Syntax for subsref, and subsasgn Methods”.

Examples

Argument values for the `subsasgn` for the expression shown:

```
A(1:2,:) = B;
```

The syntax `A(1:2,:) = B` calls `A = subsasgn(A,S,B)` where `S` is a structure with `S.type = '()'` and `S.subs = {1:2,':'}`. The colon character (`':'`) indicates a colon used as a subscript.

For the expression:

```
A{1:2} = B;
```

The syntax `A{1:2} = B` calls `A = subsasgn(A,S,B)` where `S.type = '{}'` and `S.subs = {[1 2]}`.

For the expression:

```
A.field = B;
```

The syntax `A.field = B` calls `A = subsasgn(A,S,B)` where `S.type = '.'` and `S.subs = 'field'`.

For the expression:

```
A(1,2).name(3:5) = B;
```

Simple calls combine in a straightforward way for more complicated indexing expressions. In such cases, `length(S)` is the number of subscripting levels. For instance, `A(1,2).name(3:5) = B` calls `A = subsasgn(A,S,B)` where `S` is a 3-by-1 array of structures with the following values:

```
S(1).type = '()'       S(2).type = '.'        S(3).type = '()'
S(1).subs = {1,2}      S(2).subs = 'name'     S(3).subs = {[3 4 5]}
```

Tips

Within the `subsasgn` method defined by a class, MATLAB calls the built-in `subsasgn`. Calling the built-in enables you to use the default indexing behavior when defining specialized indexing. For more information, see “Built-In subsref and subsasgn Called in Methods”.

Algorithms

In the assignment `A(J,K,...) = B(M,N,...)`, subscripts `J`, `K`, `M`, `N`, and so on, can be scalar, vector, or arrays, when all the following are true:

- The number of subscripts specified for `B`, excluding trailing subscripts equal to 1, does not exceed the value returned by `ndims(B)`.
- The number of nonscalar subscripts specified for `A` equals the number of nonscalar subscripts specified for `B`. For example, `A(5,1:4,1,2) = B(5:8)` is valid because both sides of the equation use one nonscalar subscript.
- The order and length of all nonscalar subscripts specified for `A` matches the order and length of nonscalar subscripts specified for `B`. For example, `A(1:4,3,3:9) = B(5:8,1:7)` is valid because both sides of the equation (ignoring the one scalar subscript `3`) use a 4-element subscript followed by a 7-element subscript.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Tall arrays support a limited subset of indexing operations. For details, see “Index and View Tall Array Elements”.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Subscripts must not contain repeated values.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- Expansion by using linear indexing is not supported.
- Curly brace indexing for cell arrays and dot indexing for structures are not supported.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`numArgumentsFromSubscript` | `subsref` | `substruct`

Topics

"Number of Arguments for subsref and subsasgn"
"Modify nargout and nargin for Indexing Methods"
"Indexed Assignment"
"Code Patterns for subsref and subsasgn Methods"
"Comma-Separated Lists"

Introduced before R2006a

subsindex

Convert object to array index

Syntax

```
ind = subsindex(A)
```

Description

MATLAB calls `subsindex` to convert an object into an integer index. Define a `subsindex` method for your class if you want to use objects of the class as array indices.

`ind = subsindex(A)` called by MATLAB for the expression `X(A)` when `A` is an object. MATLAB does not call `subsindex` if the indexing expression results in a call to an overloaded `subsref` or `subsasgn` method for the class of `X`. `subsindex` must return the value of the object as a zero-based integer index. `ind` must contain integer values in the range `0` to `prod(size(X))-1`.

MATLAB invokes `subsindex` separately on all the subscripts in an expression, such as `X(A,B)`.

Extended Capabilities

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`subsasgn` | `subsref`

Topics

“Objects in Index Expressions”

Introduced before R2006a

subspace

Angle between two subspaces

Syntax

```
theta = subspace(A,B)
```

Description

`theta = subspace(A,B)` finds the angle between two subspaces specified by the columns of `A` and `B`. If `A` and `B` are column vectors of unit length, this is the same as `acos(abs(A'*B))`.

Examples

Consider two subspaces of a Hadamard matrix, whose columns are orthogonal.

```
H = hadamard(8);
A = H(:,2:4);
B = H(:,5:8);
```

Note that matrices `A` and `B` are different sizes — `A` has three columns and `B` four. It is not necessary that two subspaces be the same size in order to find the angle between them. Geometrically, this is the angle between two hyperplanes embedded in a higher dimensional space.

```
theta = subspace(A,B)
theta =
    1.5708
```

That `A` and `B` are orthogonal is shown by the fact that `theta` is equal to $\pi/2$.

```
theta - pi/2
ans =
     0
```

Tips

If the angle between the two subspaces is small, the two spaces are nearly linearly dependent. In a physical experiment described by some observations `A`, and a second realization of the experiment described by `B`, `subspace(A,B)` gives a measure of the amount of new information afforded by the second experiment not associated with statistical errors of fluctuations.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Introduced before R2006a

subsref

Subscripted reference

Syntax

```
B = subsref(A,S)
```

Description

Classes overload `subsref` to implement custom indexing behavior for objects of the class. For more information, see “Object Indexing”.

`B = subsref(A,S)` is called by MATLAB for the syntax `A(i)`, `A{i}`, or `A.i` when `A` is an object.

Examples

Parentheses Indexing

This example shows how MATLAB® calls `subsref` for the following indexing expression.

```
A = magic(5);
A(1:2,:)
```

```
ans = 2×5

    17    24     1     8    15
    23     5     7    14    16
```

The syntax, `A(1:2,:)`, results in a call to `B = subsref(A,S)` where `S` is a 1-by-1 structure where `S.type is '()'` and `S.subs is {1:2,':'}`. The colon character indicates a colon used as a subscript.

Brace Indexing

This example shows how MATLAB® calls `subsref` for indexing expression that use braces.

```
C = {"one", 2, 'three'};
C{1:2}
```

```
ans =
"one"

ans = 2
```

The syntax, C{1:2}, results in a call to `[c1,c2] = subsref(C,S)` where `S.type` is `'{}'` and `S.subs` is `{[1 2]}`.

Dot Indexing

This example shows how MATLAB® calls `subsref` for indexing expression that use dot notation.

```
A = struct('number',10);
A.number
```

```
ans = 10
```

The syntax `A.number` results in a call to `B = subsref(A,S)` where `S.Type` is `'.'` and `S.subs` is `'number'`.

Input Arguments

A — Indexed object array
any object

Indexed object array, passed by MATLAB as the object array that is part of the indexing expression.

S — Indexing structure
specialized indexing structure

Indexing structure, passed by MATLAB as the indexing `substruct` for the indexing expression that caused the call to subsref. This structure has these fields:

- `type` – Character vector or string scalar containing `()`, `{}`, or `.`, specifying the subscript type.
- `subs` – Cell array, character vector, or string scalar containing the actual subscripts.

Index expressions can use more than one level to form more complicated expressions. For example `A{1}.field(3:5)` has three levels of indexing. For this expression, `S` is a 3-by-1 structure array with these fields:

```
disp(S(1))
    type: '{}'
    subs: {[1]}
disp(S(2))
    type: '.'
    subs: 'field'
disp(S(3))
    type: '()'
    subs: {[3 4 5]}
```

Data Types: `struct`

Output Arguments

B — Result of indexing expression
any type of value

Result of indexing expression.

More About

Understanding Indexing Expressions

`A(I)` is an array formed from the elements of `A` specified by the subscript vector `I`. The resulting array is the same size as `I` except for the special case where `A` and `I` are both

vectors. In this case, `A(I)` has the same number of elements as `I` but has the orientation of `A`.

`A(I,J)` is an array formed from the elements of the rectangular submatrix of `A`, specified by the subscript vectors `I` and `J`. The resulting array has `length(I)` rows and `length(J)` columns. A colon used as a subscript indicates all elements in that dimension. For example, `A(I,:)` means all columns of those rows specified by vector `I`. Similarly, `A(:,J)` means all rows of columns specified by `J`.

`A(I,J,K,...)` is the array specified by the subscripts. The result is `length(I)`-by-`length(J)`-by-`length(K)...`.

`A{I}` where `A` is a cell array and `I` is a scalar forms a copy of the array in the specified cell of `A`. If `I` has more than one element, this expression is a comma-separated list. You can also use multiple subscripts that specify a scalar element, as in `A{3,4}`.

`A(I).field` when `A` is a structure array and `I` is a scalar forms a copy of the array in the field with the name `field`. If `I` has more than one element, this expression is a comma-separated list. If `A` is a 1-by-1 structure array, then the subscript can be dropped. In this case, `A.field` is the same as `A(1).field`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Tall arrays support a limited subset of indexing operations. For details, see “Index and View Tall Array Elements”.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- Curly brace indexing for cell arrays and dot indexing for structures are not supported (GPU arrays do not support cell arrays or structures).

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- When using a codistributor2dbc scheme, subscripts must be scalar positive integers or ':'.
- Curly brace indexing for cell arrays and dot indexing for structures are not supported.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`numArgumentsFromSubscript` | `subsasgn` | `substruct`

Topics

“Number of Arguments for subsref and subsasgn”
“Modify nargout and nargin for Indexing Methods”
“Indexed Reference”
“Code Patterns for subsref and subsasgn Methods”
“Comma-Separated Lists”
“Assigning Output from a Comma-Separated List”

Introduced before R2006a

substruct

Create structure argument for `subsasgn` or `subsref`

Syntax

```
S = substruct(type1, subs1, type2, subs2, ...)
```

Description

`S = substruct(type1, subs1, type2, subs2, ...)` creates a structure with the fields required by an overloaded `subsref` or `subsasgn` method. Each `type` `char` vector must be one of `'.'`, `'()'`, or `'{}'`. The corresponding `subs` argument must be either a field name (for the `'.'` type) or a cell array containing the index vectors (for the `'()'` or `'{}'` types).

Output Arguments

S

`struct` with these fields:

- `type`: one of `'.'`, `'()'`, or `'{}'`
- `subs`: subscript values (field name or cell array of index vectors)

Examples

Call `subsref` with arguments equivalent to the syntax:

```
B = A(3,5).field;
```

where `A` is an object of a class that implements a `subsref` method

Use `substruct` to form the input `struct`, `S`:

```
S = substruct('()',{3,5},'.','field');
```

Call the class method:

```
B = subsref(A,S);
```

The `struct` created by `substruct` in this example contains:

```
disp(S(1))
    type: '()'
    subs: {[3]  [5]}

disp(S(2))
    type: '.'
    subs: 'field'
```

See Also

`subsasgn` | `subsref`

Topics

"Object Array Indexing"

Introduced before R2006a

subvolume

Extract subset of volume data set

Syntax

```
[Nx,Ny,Nz,Nv] = subvolume(X,Y,Z,V,limits)
[Nx,Ny,Nz,Nv] = subvolume(V,limits)
Nv = subvolume(...)
```

Description

`[Nx,Ny,Nz,Nv] = subvolume(X,Y,Z,V,limits)` extracts a subset of the volume data set `V` using the specified axis-aligned `limits`. `limits = [xmin,xmax,ymin, ymax,zmin,zmax]` (Any `NaN`s in the limits indicate that the volume should not be cropped along that axis.)

The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`. The subvolume is returned in `NV` and the coordinates of the subvolume are given in `NX`, `NY`, and `NZ`.

`[Nx,Ny,Nz,Nv] = subvolume(V,limits)` assumes the arrays `X`, `Y`, and `Z` are defined as

```
[X,Y,Z] = meshgrid(1:N,1:M,1:P)
```

where `[M,N,P] = size(V)`.

`Nv = subvolume(...)` returns only the subvolume.

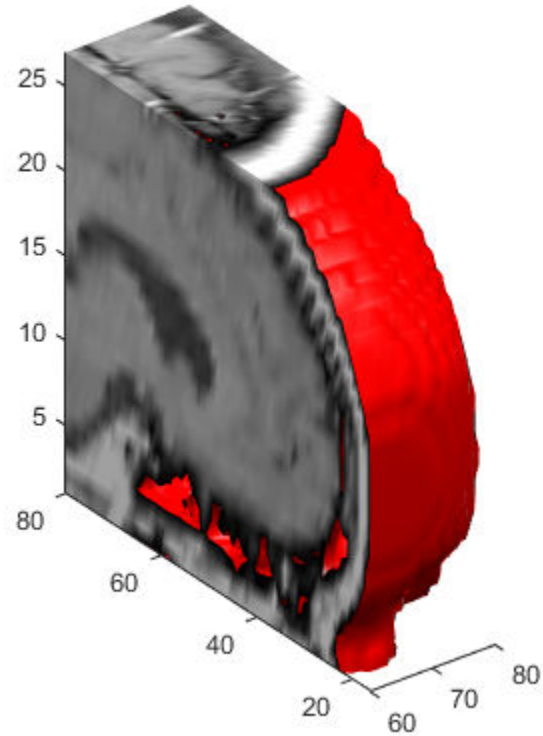
Examples

Extract Subset of Volume Data Set

Load the `mri` data set, which is a collection of MRI slices of a human skull. Then:

- Squeeze the four-dimensional array into three dimensions and extract a subset using `subvolume`.
- Use a patch to display an isosurface of the skull outline.
- Add a second patch with interpolated face colors.
- Set the view of the object.
- Change the colormap to a grayscale colormap.
- Add lights to the right and left of the camera to illuminate the object.

```
load mri
D = squeeze(D);
[x,y,z,D] = subvolume(D,[60,80,nan,80,nan,nan]);
p1 = patch(isosurface(x,y,z,D, 5),...
     'FaceColor','red','EdgeColor','none');
isonormals(x,y,z,D,p1);
p2 = patch(isocaps(x,y,z,D, 5),...
     'FaceColor','interp','EdgeColor','none');
view(3);
axis tight;
daspect([1 1 0.4])
colormap(gray(100))
camlight right;
camlight left;
lighting gouraud
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`isocaps` | `isonormals` | `isosurface` | `reducepatch` | `reducevolume` | `smooth3`

Introduced before R2006a

sum

Sum of array elements

Syntax

```
S = sum(A)
S = sum(A,'all')
S = sum(A,dim)
S = sum(A,vecdim)
S = sum( ___ ,outtype)
S = sum( ___ ,nanflag)
```

Description

`S = sum(A)` returns the sum of the elements of A along the first array dimension whose size does not equal 1.

- If `A` is a vector, then `sum(A)` returns the sum of the elements.
- If `A` is a matrix, then `sum(A)` returns a row vector containing the sum of each column.
- If `A` is a multidimensional array, then `sum(A)` operates along the first array dimension whose size does not equal 1, treating the elements as vectors. This dimension becomes `1` while the sizes of all other dimensions remain the same.

`S = sum(A,'all')` computes the sum of all elements of `A`. This syntax is valid for MATLAB versions R2018b and later.

`S = sum(A,dim)` returns the sum along dimension `dim`. For example, if `A` is a matrix, then `sum(A,2)` is a column vector containing the sum of each row.

`S = sum(A,vecdim)` sums the elements of `A` based on the dimensions specified in the vector `vecdim`. For example, if `A` is a matrix, then `sum(A,[1 2])` is the sum of all elements in `A`, since every element of a matrix is contained in the array slice defined by dimensions 1 and 2.

`S = sum( ___ ,outtype)` returns the sum with a specified data type, using any of the input arguments in the previous syntaxes. `outtype` can be `'default'`, `'double'`, or `'native'`.

`S = sum( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `sum(A,'includenan')` includes all `NaN` values in the calculation while `sum(A,'omitnan')` ignores them.

Examples

Sum of Vector Elements

Create a vector and compute the sum of its elements.

```
A = 1:10;
S = sum(A)
```

```
S = 55
```

Sum of Matrix Columns

Create a matrix and compute the sum of the elements in each column.

```
A = [1 3 2; 4 2 5; 6 1 4]
```

```
A = 3×3

     1     3     2
     4     2     5
     6     1     4
```

```
S = sum(A)
```

```
S = 1×3

    11     6    11
```

Sum of Matrix Rows

Create a matrix and compute the sum of the elements in each row.

```
A = [1 3 2; 4 2 5; 6 1 4]
```

```
A = 3×3

     1     3     2
     4     2     5
     6     1     4
```

```
S = sum(A,2)
```

```
S = 3×1

     6
    11
    11
```

Sum of Array Slices

Use a vector dimension argument to operate on specific slices of an array.

Create a 3-D array whose elements are 1.

```
A = ones(4,3,2);
```

To sum all elements in each page of A, specify the dimensions in which to sum (row and column) using a vector dimension argument. Since both pages are a 4-by-3 matrix of ones, the sum of each page is 12.

```
S1 = sum(A,[1 2])
```

```
S1 =
S1(:,:,1) =

    12
```

```
S1(:,:,2) =

    12
```

If you slice A along the first dimension, you can sum the elements of the resulting 4 pages, which are each 3-by-2 matrices.

```
S2 = sum(A,[2 3])
```

```
S2 = 4×1

     6
     6
     6
     6
```

Slicing along the second dimension, each page sum is over a 4-by-2 matrix.

```
S3 = sum(A,[1 3])
```

```
S3 = 1×3

     8     8     8
```

Starting in R2018b, to sum over all dimensions of an array, you can either specify each dimension in the vector dimension argument, or use the `'all'` option.

```
S4 = sum(A,[1 2 3])
```

```
S4 = 24
```

```
Sall = sum(A,'all')
```

```
Sall = 24
```

Sum of 3-D Array

Create a 4-by-2-by-3 array of ones and compute the sum along the third dimension.

```
A = ones(4,2,3);
S = sum(A,3)
```

```
S = 4×2

     3     3
     3     3
     3     3
     3     3
```

Sum of 32-bit Integers

Create a vector of 32-bit integers and compute the `int32` sum of its elements by specifying the output type as `native`.

```
A = int32(1:10);
S = sum(A,'native')
```

```
S = int32
    55
```

Sum Excluding NaN

Create a vector and compute its sum, excluding `NaN` values.

```
A = [1.77 -0.005 3.98 -2.95 NaN 0.34 NaN 0.19];
S = sum(A,'omitnan')
```

```
S = 3.3250
```

If you do not specify `'omitnan'`, then `sum(A)` returns `NaN`.

Input Arguments

A — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

- If `A` is a scalar, then `sum(A)` returns `A`.
- If `A` is an empty 0-by-0 matrix, then `sum(A)` returns `0`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `duration`
Complex Number Support: Yes

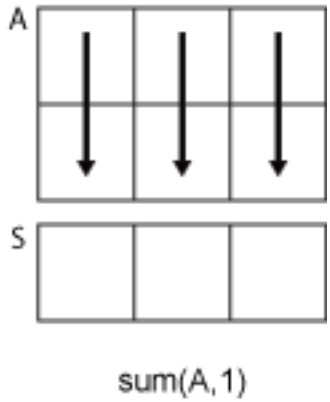
`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

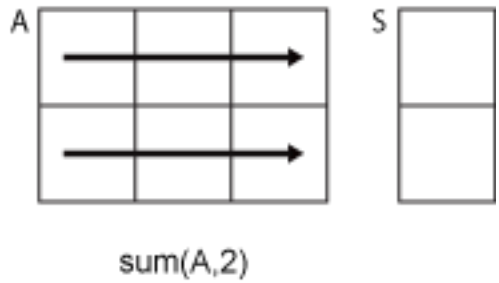
Dimension `dim` indicates the dimension whose length reduces to 1. The `size(S,dim)` is `1`, while the sizes of all other dimensions remain the same.

Consider a two-dimensional input array, `A`:

- `sum(A,1)` operates on successive elements in the columns of `A` and returns a row vector of the sums of each column.



- `sum(A,2)` operates on successive elements in the rows of `A` and returns a column vector of the sums of each row.



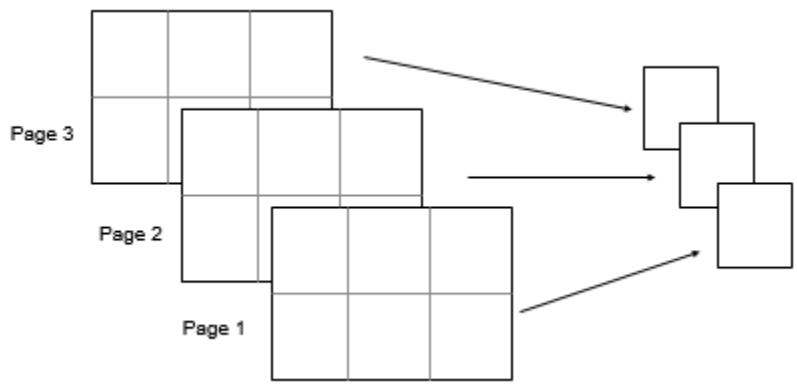
`sum` returns `A` when `dim` is greater than `ndims(A)` or when `size(A,dim)` is `1`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`vecdim` — Vector of dimensions
vector of positive integers

Vector of dimensions, specified as a vector of positive integers. Each element represents a dimension of the input array. The lengths of the output in the specified operating dimensions are 1, while the others remain the same.

Consider a 2-by-3-by-3 input array, `A`. Then `sum(A,[1 2])` returns a 1-by-1-by-3 array whose elements are the sums of each page of `A`.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

outtype — Output data type
'default' (default) | 'double' | 'native'

Output data type, specified as 'default', 'double', or 'native'. These options also specify the data type in which the operation is performed.

outtype	Output data type
'default'	double, unless the input data type is single or duration, in which case, the output is 'native'
'double'	double, unless the data type is duration, in which case, 'double' is not supported
'native'	same data type as the input, unless the input data type is char, in which case, 'native' is not supported

Data Types: char

nanflag — NaN condition
'includenan' (default) | 'omitnan'

NaN condition, specified as one of these values:

- 'includenan' — Include NaN values when computing the sum, resulting in NaN.
- 'omitnan' — Ignore all NaN values in the input.

Data Types: char

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see "Tall Arrays".

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If you specify `dim`, then it must be a constant.
- The `outtype` and `nanflag` options must be constant character vectors.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- If you specify `dim`, then it must be a constant.
- The `outtype` and `nanflag` options must be constant character vectors.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The order of the additions in the `sum` operation is not defined. Therefore, the `sum` operation on a GPU array might not return exactly the same answer as the `sum` operation on the corresponding MATLAB numeric array. The difference might be significant when `A` is a signed integer type and its product is accumulated natively.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- The order of the additions in `sum` operation is not defined. Therefore, the `sum` operation on a distributed array might not return exactly the same answer as the `sum`

operation on the corresponding MATLAB numeric array. The difference might be significant when `A` is a signed integer type and its product is accumulated natively.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cumsum` | `diff` | `mean` | `prod`

Introduced before R2006a

summary

Print summary of table, timetable, or categorical array

Syntax

```
summary(T)
s = summary(T)

summary(A)
summary(A,dim)
```

Description

`summary(T)` prints a summary of the table or timetable `T`.

- If `T` is a table, then the table summary on page 1-14903 displays the description from `T.Properties.Description` followed by a summary of the table variables.
- If `T` is a timetable, then the timetable summary displays the description from `T.Properties.Description`, a summary of the row times, and then a summary of the timetable variables.

`s = summary(T)` returns a structure, `s`, that contains a summary of the input table or timetable. Each field of `s` is itself a structure that summarizes the values in the corresponding variable of `T`. If `T` is a timetable, then `s` also has a field that summarizes the row times of `T`.

`summary(A)` prints a summary of the categorical array `A`.

- If `A` is a vector, then `summary(A)` displays the category names along with the number of elements in each category (the category counts). It also displays the number of elements that are undefined.
- If `A` is a matrix, then `summary` treats the columns of `A` as vectors and displays the category counts for each column of `A`.
- If `A` is a multidimensional array, then `summary` acts along the first array dimension whose size does not equal `1`.

`summary(A,dim)` prints the category counts of the categorical array `A` along dimension `dim`.

For example, you can display the counts of each row in a categorical array using `summary(A,2)`.

Examples

Summary of Table

Create a table.

```
load patients
BloodPressure = [Systolic Diastolic];
T = table(Gender,Age,Smoker,BloodPressure,'RowNames',LastName);
```

Add descriptions and units to table `T`. You can add a description for the table as a whole, and also for individual variables.

```
T.Properties.Description = 'Simulated patient data';
T.Properties.VariableUnits =  {''  'Yrs' ''  'mm Hg'};
T.Properties.VariableDescriptions{4} = 'Systolic/Diastolic';
```

Print a summary of table `T`.

```
format compact
```

```
summary(T)
```

```
Description:  Simulated patient data
Variables:
    Gender: 100x1 cell array of character vectors
    Age: 100x1 double
        Properties:
            Units:  Yrs
        Values:
            Min        25
            Median     39
            Max        50
    Smoker: 100x1 logical
        Values:
            True        34
```

```
            False       66
    BloodPressure: 100x2 double
        Properties:
            Units:  mm Hg
            Description:  Systolic/Diastolic
        Values:
                        BloodPressure_1    BloodPressure_2
                        _______________    _______________
            Min               109                68
            Median            122               81.5
            Max               138                99
```

`summary` displays the minimum, median, and maximum values for each column of the variable `BloodPressure`.

Summary of Timetable

Create a small timetable.

```
Time = [seconds(1:5)]';
TT = timetable(Time,[98;97.5;97.9;98.1;97.9],[120;111;119;117;116],...
               'VariableNames',{'Reading1','Reading2'})
```

```
TT=5×3 timetable
    Time     Reading1    Reading2
    _____    ________    ________

    1 sec        98        120
    2 sec      97.5        111
    3 sec      97.9        119
    4 sec      98.1        117
    5 sec      97.9        116
```

Print a summary of the timetable. `summary` prints a summary of the row times, followed by a summary of the variables. If the timetable is regular, then `summary` also prints the size of the time step between row times.

```
summary(TT)
```

```
RowTimes:
```

```
    Time: 5x1 duration
        Values:
            Min           1 sec
            Median        3 sec
            Max           5 sec
            TimeStep      1 sec

Variables:

    Reading1: 5x1 double

        Values:

            Min          97.5
            Median       97.9
            Max          98.1

    Reading2: 5x1 double

        Values:

            Min          111
            Median       117
            Max          120
```

Return Summary of Table as Structure

Create a table. Add units to the table variables. Then display the first few rows.

```
load patients
BloodPressure = [Systolic Diastolic];
T = table(Gender,Age,Smoker,BloodPressure,'RowNames',LastName);
T.Properties.VariableUnits =  {''  'Years' ''  'mm Hg'};
head(T,3)
```

```
ans=3×4 table
                  Gender      Age    Smoker    BloodPressure
                _________     ___    ______    _____________

    Smith       {'Male'  }    38     true       124     93
    Johnson     {'Male'  }    43     false      109     77
    Williams    {'Female'}    38     false      125     83
```

Return a summary of the table. To return a summary as a structure, specify an output argument when using the `summary` function.

```
s = summary(T)
```

```
s = struct with fields:
           Gender: [1x1 struct]
              Age: [1x1 struct]
           Smoker: [1x1 struct]
    BloodPressure: [1x1 struct]
```

Display the summary of the table variable `Age`. For each variable of `T`, the output argument `s` has a field that contains its summary.

```
s.Age
```

```
ans = struct with fields:
           Size: [100 1]
           Type: 'double'
    Description: ''
          Units: 'Years'
     Continuity: []
            Min: 25
         Median: 39
            Max: 50
     NumMissing: 0
```

The `NumMissing` field shows the number of elements that are the missing value. In this case, `Age` does not contain any `NaN` values, so `NumMissing` is zero. `summary` includes the `NumMissing` field for numeric, `duration`, `datetime`, and `categorical` variables.

Display the minimum age contained in the table. You can access any field of the summary by name.

```
s.Age.Min
```

```
ans = 25
```

Display the summary of the table variable `Smoker`. You can determine the numbers of smokers and nonsmokers from the `True` and `False` fields. The information contained in the summary of a table variable depends on the data type of the variable.

```
s.Smoker
```

```
ans = struct with fields:
           Size: [100 1]
           Type: 'logical'
    Description: ''
          Units: ''
     Continuity: []
           True: 34
          False: 66
```

Return Summary of Timetable as Structure

Create a timetable.

```
Time = datetime({'2015-12-18 08:00:00';'2015-12-18 10:00:00';'2015-12-18 12:00:00'});
Temp = [37.3;39.1;42.3];
Pressure = [30.1;30.03;29.9];
TT = timetable(Time,Temp,Pressure)
```

```
TT=3×3 timetable
            Time            Temp    Pressure
    ____________________    ____    ________

    18-Dec-2015 08:00:00    37.3      30.1
    18-Dec-2015 10:00:00    39.1     30.03
    18-Dec-2015 12:00:00    42.3      29.9
```

Return a summary of the timetable as a structure.

```
s = summary(TT)
```

```
s = struct with fields:
        Time: [1x1 struct]
        Temp: [1x1 struct]
    Pressure: [1x1 struct]
```

Display the summary of the row times. The `TimeStep` field shows that the time interval between consecutive row times is two hours. The `NumMissing` field shows there are no missing values (`NaT`) in the vector of row times.

```
s.Time
```

```
ans = struct with fields:
          Size: [3 1]
          Type: 'datetime'
           Min: 18-Dec-2015 08:00:00
        Median: 18-Dec-2015 10:00:00
           Max: 18-Dec-2015 12:00:00
    NumMissing: 0
      TimeStep: 02:00:00
```

Change the last row time so that the row times have different intervals between them.

```
TT.Time(3) = '2015-12-18 11:00:00';
TT
```

```
TT=3×3 timetable
            Time            Temp    Pressure
    ____________________    ____    ________

    18-Dec-2015 08:00:00    37.3      30.1
    18-Dec-2015 10:00:00    39.1     30.03
    18-Dec-2015 11:00:00    42.3      29.9
```

Return a summary of the updated timetable. Since the time steps between row times are different, the `TimeStep` field has a `NaN`.

```
s = summary(TT);
s.Time
```

```
ans = struct with fields:
          Size: [3 1]
          Type: 'datetime'
           Min: 18-Dec-2015 08:00:00
        Median: 18-Dec-2015 10:00:00
           Max: 18-Dec-2015 11:00:00
    NumMissing: 0
      TimeStep: NaN
```

Table Summary Including Custom Properties

Starting in R2018b, you can add custom properties to tables and timetables. If you add custom properties, then the summary of a table or timetable includes those properties.

First, create a table and add values to some of its predefined properties.

```
load patients
BloodPressure = [Systolic Diastolic];
T = table(Gender,Age,Smoker,BloodPressure,'RowNames',LastName);
T.Properties.Description = 'Simulated patient data';
T.Properties.VariableUnits =  {''  'Yrs' ''  'mm Hg'};
T.Properties.VariableDescriptions{4} = 'Systolic/Diastolic';
```

Add custom properties using the `addprop` function. For each custom property, specify a name. Also, specify whether the value of each custom property stores metadata that applies to the table or to individual table variables.

```
T = addprop(T,{'SourceFile','DataOrigin'},{'table','variable'});
```

Store metadata values in the custom properties.

```
T.Properties.CustomProperties.SourceFile = 'patients.mat';
T.Properties.CustomProperties.DataOrigin = {'census','census','self report','blood pres
```

Print a summary of the table. Aside from `T.Properties.Description`, the `summary` function does not display properties that apply to the table as a whole. So, it does not display the value of `T.Properties.CustomProperties.SourceFile`. However, `summary` does display properties that apply to table variables. For each variable, `summary` displays the corresponding value from `T.Properties.CustomProperties.DataOrigin`.

```
summary(T)
```

```
Description:  Simulated patient data

Variables:

    Gender: 100x1 cell array of character vectors

        Custom Properties:
            DataOrigin:  census
    Age: 100x1 double
```

```
        Properties:
            Units:  Yrs
        Custom Properties:
            DataOrigin:  census
        Values:

            Min        25
            Median     39
            Max        50

    Smoker: 100x1 logical

        Custom Properties:
            DataOrigin:  self report
        Values:

            True        34
            False       66

    BloodPressure: 100x2 double

        Properties:
            Units:  mm Hg
            Description:  Systolic/Diastolic
        Custom Properties:
            DataOrigin:  blood pressure reading
        Values:
                      BloodPressure_1    BloodPressure_2
                      _______________    _______________

            Min             109                 68
            Median          122               81.5
            Max             138                 99
```

Return the summary as a structure. Each field has a structure corresponding to one of the table variables.

```
s = summary(T)
```

```
s = struct with fields:
           Gender: [1x1 struct]
              Age: [1x1 struct]
           Smoker: [1x1 struct]
    BloodPressure: [1x1 struct]
```

The structure `s.Age` stores the summary for the `Age` variable.

```
s.Age
```

```
ans = struct with fields:
                Size: [100 1]
                Type: 'double'
         Description: ''
               Units: 'Yrs'
          Continuity: []
                 Min: 25
              Median: 39
                 Max: 50
          NumMissing: 0
    CustomProperties: [1x1 struct]
```

The `s.Age.CustomProperties` structure stores the corresponding value from the `T.Properties.CustomProperties.DataOrigin` property.

```
s.Age.CustomProperties
```

```
ans = struct with fields:
    DataOrigin: {'census'}
```

Summary of Categorical Vector

Create a 1-by-5 categorical vector.

```
A = categorical({'plane' 'car' 'train' 'car' 'plane'})
```

```
A = 1x5 categorical array
     plane      car      train      car      plane
```

`A` has three categories, `car`, `plane`, and `train`.

Print a summary of `A`.

```
summary(A)
```

```
     car      plane      train
       2          2          1
```

car appears in two elements of A, plane appears in two elements, and train appears in one element.

Since A is a row vector, summary lists the occurrences of each category horizontally.

Summary of Each Column in Categorical Array

Create a 4-by-2 categorical array, A, from a numeric array.

```
X = [1 3; 2 1; 3 1; 4 2];
valueset = 1:3;
catnames = {'red','green','blue'};

A = categorical(X,valueset,catnames)
```

```
A = 4x2 categorical array
     red              blue
     green            red
     blue             red
     <undefined>      green
```

A has three categories, red, green, and blue. The value, 4, was not included in the valueset input to the categorical function. Therefore, the corresponding element, A(4,1), does not have a corresponding category and is undefined.

Print a summary of A.

```
summary(A)
```

```
     red              1      2
     green            1      1
     blue             1      1
     <undefined>      1      0
```

red appears once in the first column of A and twice in the second column.

green appears once in the first column of A and once in the second column.

`blue` appears once in the first column of `A` and once in the second column.

`A` contains only one undefined element. It occurs in the first column.

Category Counts of Each Row in Categorical Array

Create a 3-by-2 categorical array, `A`, from a numeric array.

```
A = categorical([1 3; 2 1; 3 1],1:3,{'red','green','blue'})
```

```
A = 3x2 categorical array
     red        blue
     green      red
     blue       red
```

`A` has three categories, `red`, `green`, and `blue`.

Print a summary of `A` along the second dimension.

```
summary(A,2)
```

```
     red      green      blue
       1          0         1
       1          1         0
       1          0         1
```

`red` appears once in the first row of `A`, once in the second row, and once in the third row.

`green` appears in only one element. It occurs in the second row of `A`.

`blue` appears once in the first row of `A` and once in the third row.

Input Arguments

T — Input table
table | timetable

Input table, specified as a table or a timetable.

`A` — Categorical array
vector | matrix | multidimensional array

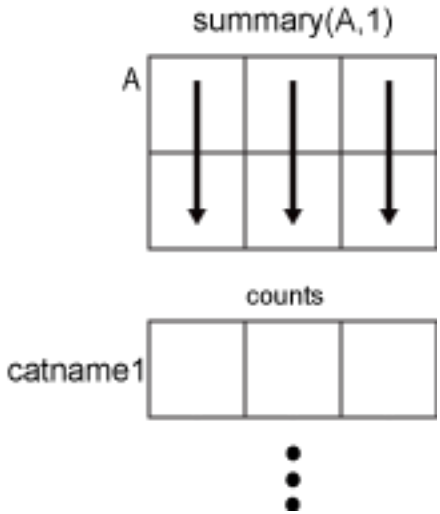
Categorical array, specified as a vector, matrix, or multidimensional array.

`dim` — Dimension of `A` to operate along
positive integer scalar

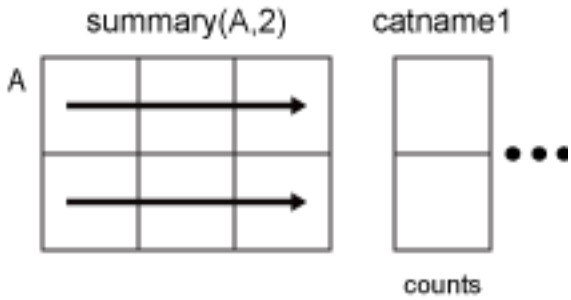
Dimension of `A` to operate to along, specified as a positive integer scalar. If no value is specified, the default is the first array dimension whose size does not equal 1.

Consider a two-dimensional categorical array `A`:

If `dim = 1`, then `summary(A,dim)` displays the category counts for each column of `A`.



If `dim = 2`, then `summary(A,dim)` returns the category counts of each row of `A`.



If `dim` is greater than `ndims(A)`, then `summary(A)` returns an array the same size as `A` for each category. `summary` returns `1` for elements in the corresponding category and `0` otherwise.

Output Arguments

`s` — Summary of table or timetable variables
scalar structure

Summary of the table or timetable variables, returned as a scalar structure. For each variable `T.`*`VarName`* in the input `T`, the output structure `s` contains a field `s.`*`VarName`* with the summary for that variable.

If `T` has variables whose names are not valid MATLAB identifiers, then `summary` modifies them to create valid field names, primarily by removing spaces and replacing non-ASCII characters with underscores.

For each data type, `s.`*`VarName`* contains the fields shown below. You can access the fields with dot indexing. For example, `s.`*`VarName`*`.Size` returns the size of the table variable named *`VarName`*.

Type of Table or Timetable Variable	Fields for Summary of Variable	Description
Numeric, `datetime`, or `duration`	`Size`	Size of variable, stored as a numeric array
	`Type`	Type of variable, stored as a character vector

Type of Table or Timetable Variable	Fields for Summary of Variable	Description
	`Description`	Description of variable, stored as a character vector
	`Units`	Units of variable, stored as a character vector
	`Min`	Minimum value
	`Median`	Median value
	`Max`	Maximum value
	`NumMissing`	Number of missing values (`NaN` or `NaT`)
	`CustomProperties` (omitted if there are no custom properties)	Names and values for custom properties associated with variable, stored as a structure
logical	`Size`	Size of variable, stored as a numeric array
	`Type`	Type of variable, stored as a character vector
	`Description`	Description of variable, stored as a character vector
	`Units`	Units of variable, stored as a character vector
	`True`	Number of `true` values
	`False`	Number of `false` values
	`CustomProperties` (omitted if there are no custom properties)	Names and values for custom properties associated with variable, stored as a structure
`categorical`	`Size`	Size of variable, stored as a numeric array
	`Type`	Type of variable, stored as a character vector

Type of Table or Timetable Variable	Fields for Summary of Variable	Description
	`Description`	Description of variable, stored as a character vector
	`Units`	Units of variable, stored as a character vector
	`Categories`	Categories, stored as a cell array of character vectors
	`Counts`	Number of elements in each category, stored as a numeric array
	`NumMissing`	Number of missing values (`<undefined>`)
	`CustomProperties` (omitted if there are no custom properties)	Names and values for custom properties associated with variable, stored as a structure
Other	`Size`	Size of variable, stored as a numeric array
	`Type`	Type of variable, stored as a character vector
	`Description`	Description of variable, stored as a character vector
	`Units`	Units of variable, stored as a character vector
	`CustomProperties` (omitted if there are no custom properties)	Names and values for custom properties associated with variable, stored as a structure

If `T` is a timetable, then `s` also has a field with a summary of the row times. For timetable row times only, the summary includes the `TimeStep` field. If the row times increase or decrease monotonically by a fixed time step, then `TimeStep` has a numeric value. If the row times are irregular, then `TimeStep` is `NaN`.

Fields for Summary of Timetable Row Times	Description of Fields
`Size`	Size of vector of row times, stored as a numeric array
`Type`	Data type, stored as a character vector
`Min`	Minimum value
`Median`	Median value
`Max`	Maximum value
`NumMissing`	Number of missing values (`NaT` or `NaN`)
`TimeStep`	Time step between consecutive row times (`NaN` if irregular)

More About

Table Summary

The table summary displays the table description from `T.Properties.Description` followed by information on the variables of `T`.

The summary contains the following information on the variables:

- Name: Size and Data Type — Variable name from `T.Properties.VariableNames`, the size of the variable, and the data type of the variable.
- Units — Variable units from `T.Properties.VariableUnits`.
- Description — Variable description from `T.Properties.VariableDescriptions`.
- Custom Properties: — Names of the custom properties that apply to variables, and their corresponding values, from `T.Properties.CustomProperties`. If there are no custom properties, then this section is omitted.
- Values — Only included for numeric, logical, `categorical`, `datetime`, or `duration` variables.
    - Numeric, `datetime`, or `duration` variables — minimum, median, and maximum values. Also, the number of missing values (`NaN`s or `NaT`s) is included when that number is greater than zero.

- Logical variables — number of values that are `true` and the number of values that are `false`.
- `categorical` variables — number of elements from each category. Also, the number of undefined elements is included when that number is greater than zero.

If `T` is a timetable, then the summary contains the same information on the vector of row times.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitation:

Some fields in the summary can be impossible to calculate in a reasonable amount of time, such as the median.

For more information, see “Tall Arrays for Out-of-Memory Data”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`categorical` | `categories` | `countcats` | `head` | `table` | `tail` | `timetable`

Topics

“Create and Work with Tables”
“Modify Units, Descriptions, and Table Variable Names”
“Access Data in Tables”

“Create Categorical Arrays”
“Access Data Using Categorical Arrays”

Introduced in R2013b

summer

Summer colormap array

Syntax

```
c = summer
c = summer(m)
```

Description

`c = summer` returns the summer colormap as a three-column array with the same number of rows as the colormap for the current figure. If no figure exists, then the number of rows is equal to the default length of 256. Each row in the array contains the red, green, and blue intensities for a specific color. The intensities are in the range [0,1], and the color scheme looks like this image.

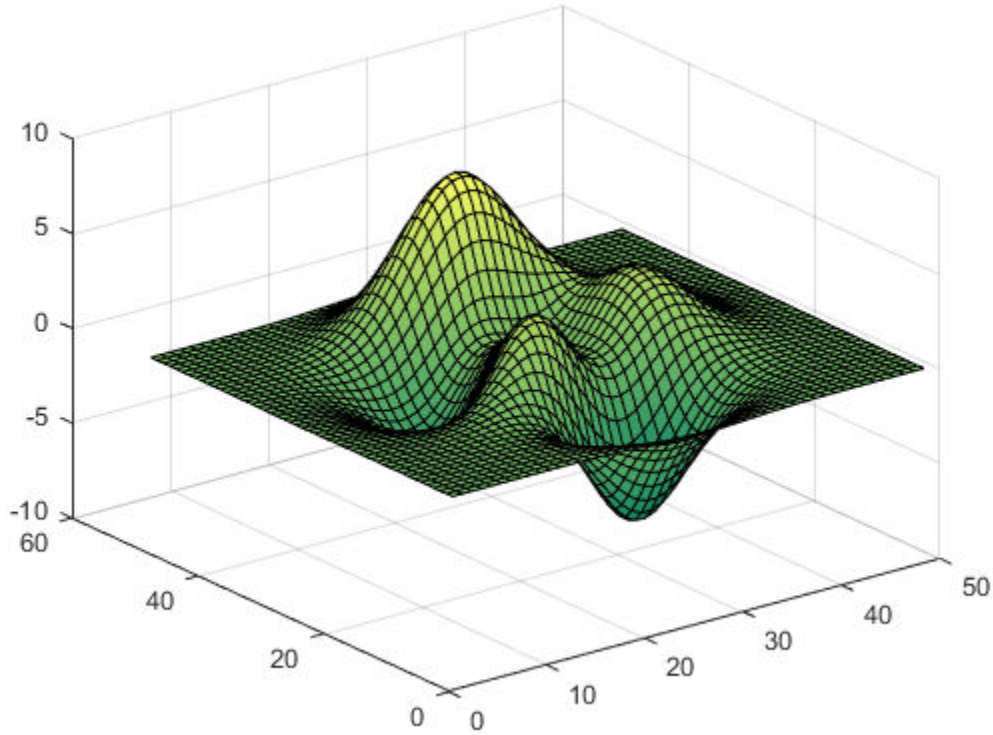
`c = summer(m)` returns the colormap with `m` colors.

Examples

Reverse the Colormap

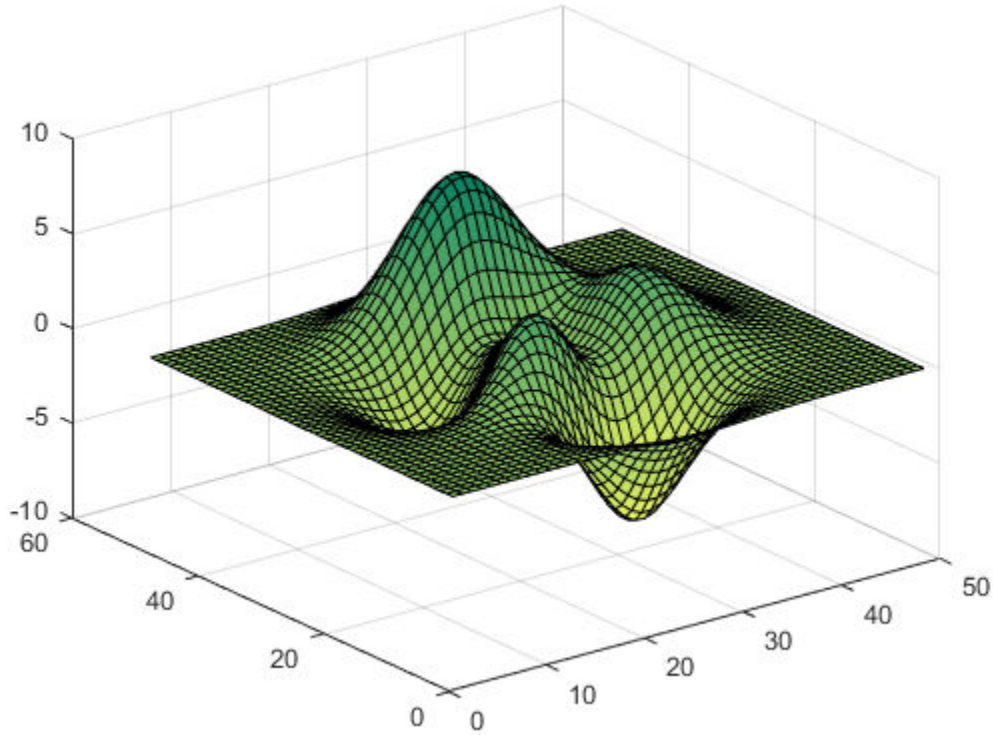
Plot a surface and assign the summer colormap.

```
surf(peaks);
colormap('summer');
```



Get the summer colormap array and reverse the order. Then apply the modified colormap to the surface.

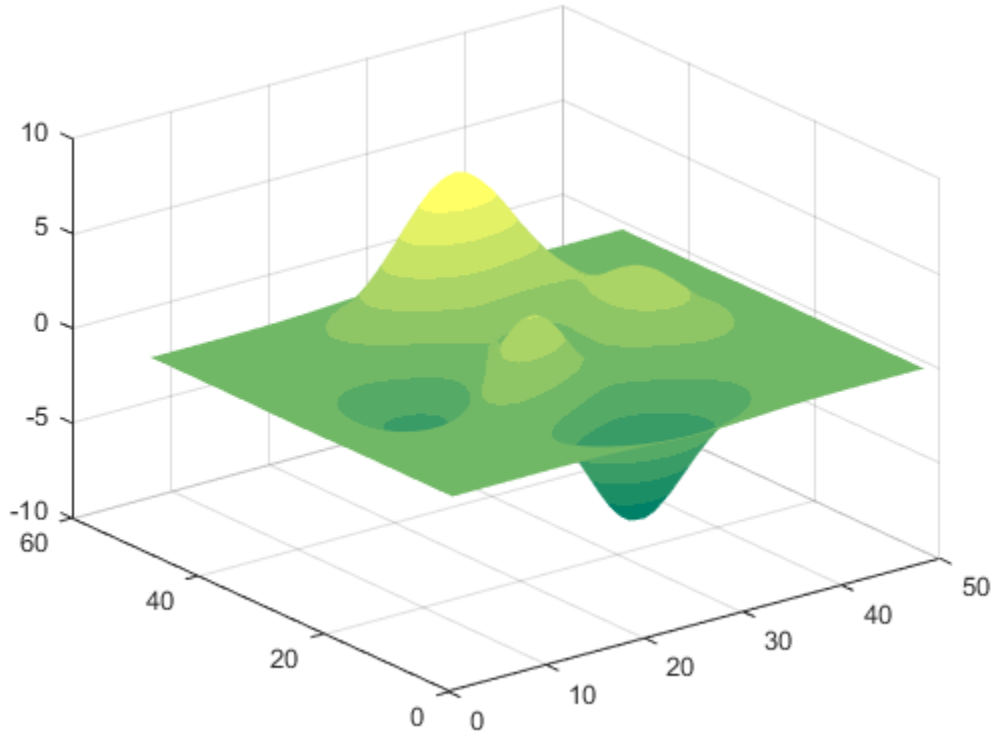
```
c = summer;
c = flipud(c);
colormap(c);
```



Downsample the Summer Colormap

Get a downsampled version of the summer colormap containing only ten colors. Then display the contours of the peaks function by applying the colormap and interpolated shading.

```
c = summer(10);
surf(peaks);
colormap(c);
shading interp;
```



Input Arguments

`m` — Number of colors
256 (default) | scalar integer value

Number of colors, specified as a scalar integer value. The default value of `m` is equal to the length of the colormap for the current figure. If no figure exists, the default value is 256.

Data Types: `single` | `double`

Compatibility Considerations

Colormaps Have 256 Colors by Default

Behavior changed in R2019b

Starting in R2019b, colormaps have 256 colors by default.

In R2019a and previous releases, the default size is 64. If you have code that depends on a colormap having 64 colors, specify the number of colors when you set the colormap for the figure, axes, or chart. For example, `colormap(summer(64))` sets the figure's colormap to the 64-color `summer` colormap.

Alternatively, you can change the default colormap for all figures within your MATLAB session:

```
set(groot,'defaultFigureColormap',summer(64))
```

See Also

`colormap`

Topics

"Change Color Scheme Using a Colormap"

Introduced before R2006a

superclasses

Names of superclasses

Syntax

```
superclasses(ClassName)
superclasses(obj)
s = superclasses( ___ )
```

Description

`superclasses(ClassName)` displays the names of all visible superclasses of the MATLAB class with the name `ClassName`. Visible classes have a `Hidden` attribute value of `false` (the default).

`superclasses(obj)` displays the names of all visible superclasses of object `obj`, where `obj` is an instance of a MATLAB class. `obj` can be either a scalar object or an array of objects.

`s = superclasses( ___ )` returns the superclass names in a cell array of character vectors.

Examples

Superclass from Class Name

Find the superclass of the matlab.lang.OnOffSwitchState class

```
superclasses('matlab.lang.OnOffSwitchState')
```

```
Superclasses for class matlab.lang.OnOffSwitchState:

    logical
```

Superclass Names from Object

Find the superclasses from an instance of the containers.Map class.

```
m = containers.Map('May',70);
superclasses(m)
```

```
Superclasses for class containers.Map:

    handle
```

Input Arguments

`ClassName` — Class name
character vector | string

Class name specified as a character vector or string scalar.

Data Types: `char` | `string`

`obj` — Object
object

Object of a MATLAB class, specified as a single object or an object array.

Output Arguments

`s` — Superclass names
cell array

Superclass names, returned as a cell array of character vectors.

See Also

`events` | `methods` | `properties`

Topics

"Class Hierarchies"

Introduced in R2008b

support

Open MathWorks Technical Support Web page

Note `support` will be removed in a future release.

Syntax

```
support
```

Description

`support` opens the MathWorks Technical Support Web page, `https://www.mathworks.com/support`, in a Web browser.

This Web page contains resources including

- A search engine, including an option for solutions to common problems
- Information about installation and licensing
- A patch archive for bug fixes you can download
- Other useful resources

See Also

`doc` | `web`

Introduced before R2006a

Support Package Installer

Install support for third-party hardware or software

Description

Use Support Package Installer to install support packages. The support packages add support for specific third-party hardware or software to specific MathWorks products.

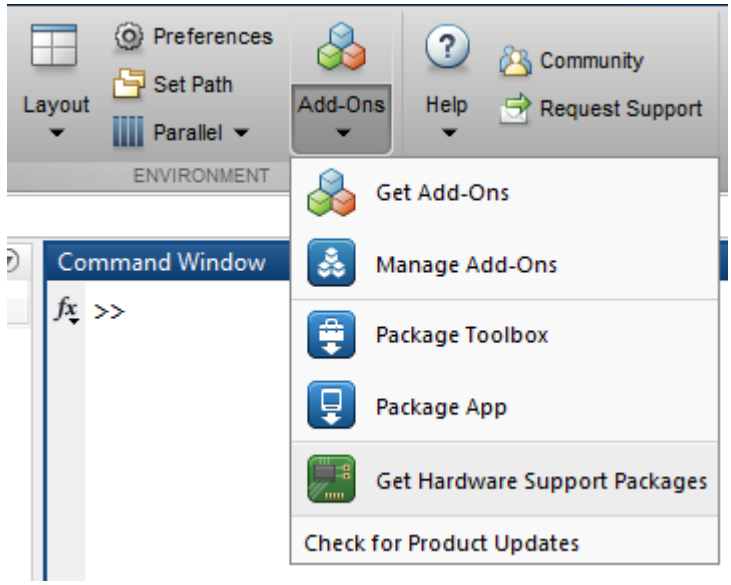
Support Package Installer can:

- Display a list of available, installable, installed, or updatable support packages.
- Install, update, download, or uninstall a support package.
- Update the firmware on specific third-party hardware.

If the support package installs third-party software, Support Package Installer displays a list of the software and licenses for you to review before continuing with the installation.

Open the Support Package Installer App

- On the MATLAB toolstrip, click **Add-Ons** > **Get Hardware Support Packages**.



- In the MATLAB Command Window, enter `supportPackageInstaller`.
- Double-click a support package installation file (`*.mlpkginstall`).

See Also

Functions

`matlabshared.supportpkg.checkForUpdate` | `matlabshared.supportpkg.getInstalled` | `supportPackageInstaller`

supportPackageInstaller

Install support for third-party hardware or software (Not recommended)

Note `supportPackageInstaller` is not recommended. To install support packages, on the MATLAB® **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**

To uninstall support packages, click **Add-Ons > Manage Add-ons**.

See “Get and Manage Add-Ons” for more details.

Syntax

```
supportPackageInstaller
```

Description

The `supportPackageInstaller` function opens Add-Ons Explorer.

Use the Add-Ons Explorer to install support packages. The support packages add support for specific third-party hardware or software to specific MathWorks products.

Examples

Use the supportPackageInstaller Function

Enter the function in the MATLAB Command Window:

```
supportPackageInstaller
```

This action starts the Add-Ons Explorer.

See Also

Topics

"Get and Manage Add-Ons"

External Websites

Hardware Support Catalog

surf

Surface plot

Syntax

```
surf(X,Y,Z)
surf(X,Y,Z,C)

surf(Z)
surf(Z,C)

surf(ax, ___ )
surf( ___ ,Name,Value)
s = surf( ___ )
```

Description

`surf(X,Y,Z)` creates a three-dimensional surface plot, which is a three-dimensional surface that has solid edge colors and solid face colors. The function plots the values in matrix `Z` as heights above a grid in the *x*-*y* plane defined by `X` and `Y`. The color of the surface varies according to the heights specified by `Z`.

`surf(X,Y,Z,C)` additionally specifies the surface color.

`surf(Z)` creates a surface plot and uses the column and row indices of the elements in `Z` as the *x*- and *y*-coordinates.

`surf(Z,C)` additionally specifies the surface color.

`surf(ax, ___ )` plots into the axes specified by `ax` instead of the current axes. Specify the axes as the first input argument.

`surf( ___ ,Name,Value)` specifies surface properties using one or more name-value pair arguments. For example, `'FaceAlpha',0.5` creates a semitransparent surface.

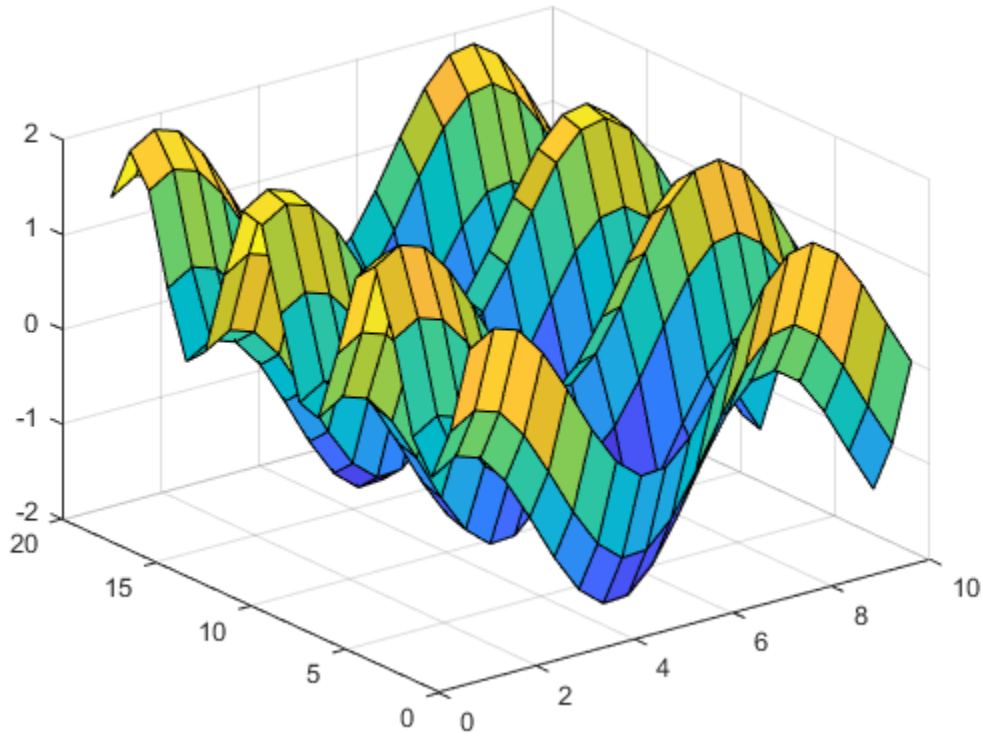
`s = surf( ___ )` returns the chart surface object. Use `s` to modify the surface after it is created. For a list of properties, see Chart Surface.

Examples

Create Surface Plot

Create three matrices of the same size. Then plot them as a surface. The surface plot uses Z for both height and color.

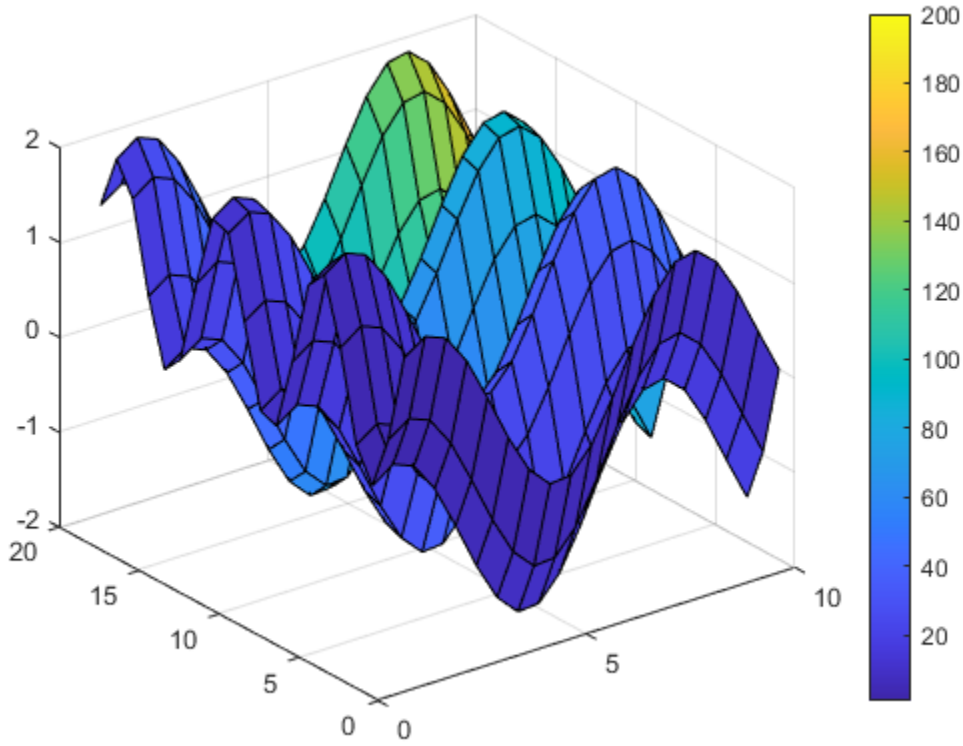
```
[X,Y] = meshgrid(1:0.5:10,1:20);
Z = sin(X) + cos(Y);
surf(X,Y,Z)
```



Specify Colormap Colors for Surface Plot

Specify the colors for a surface plot by including a fourth matrix input, `C`. The surface plot uses `Z` for height and `C` for color. Specify the colors using a *colormap*, which uses single numbers to stand for colors on a spectrum. When you use a colormap, `C` is the same size as `Z`. Add a color bar to the graph to show how the data values in `C` correspond to the colors in the colormap.

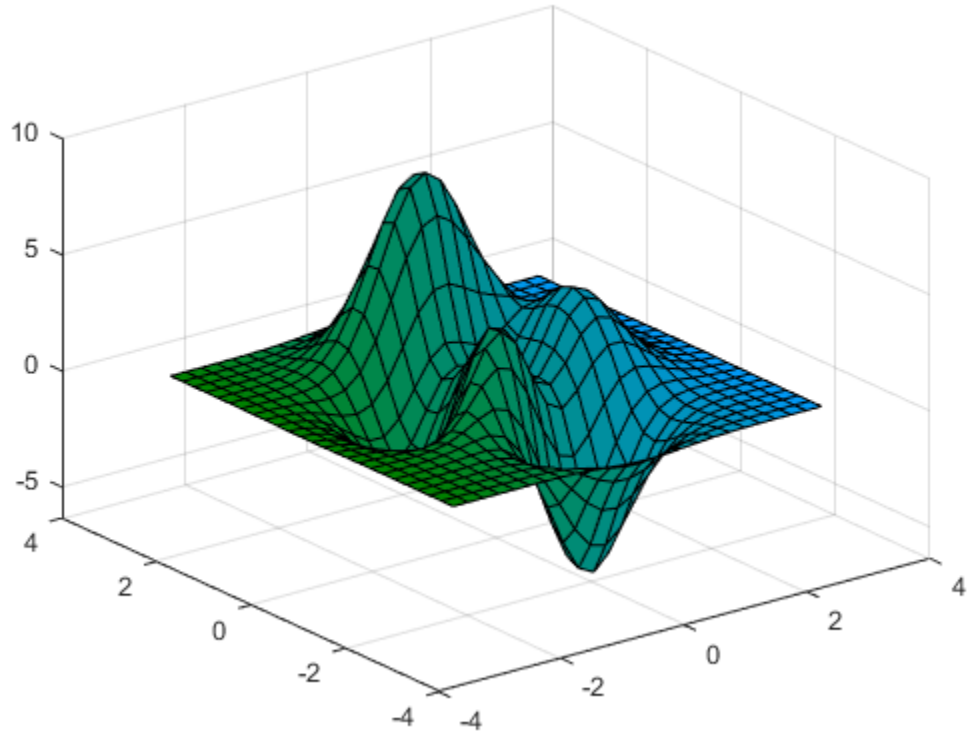
```
[X,Y] = meshgrid(1:0.5:10,1:20);
Z = sin(X) + cos(Y);
C = X.*Y;
surf(X,Y,Z,C)
colorbar
```



Specify True Colors for Surface Plot

Specify the colors for a surface plot by including a fourth matrix input, `C0`. The surface plot uses `Z` for height and `C0` for color. Specify the colors using *truecolor*, which uses triplets of numbers to stand for all possible colors. When you use truecolor, if `Z` is `m`-by-`n`, then `C0` is `m`-by-`n`-by-3. The first page of the array indicates the red component for each color, the second page indicates the green component, and the third page indicates the blue component.

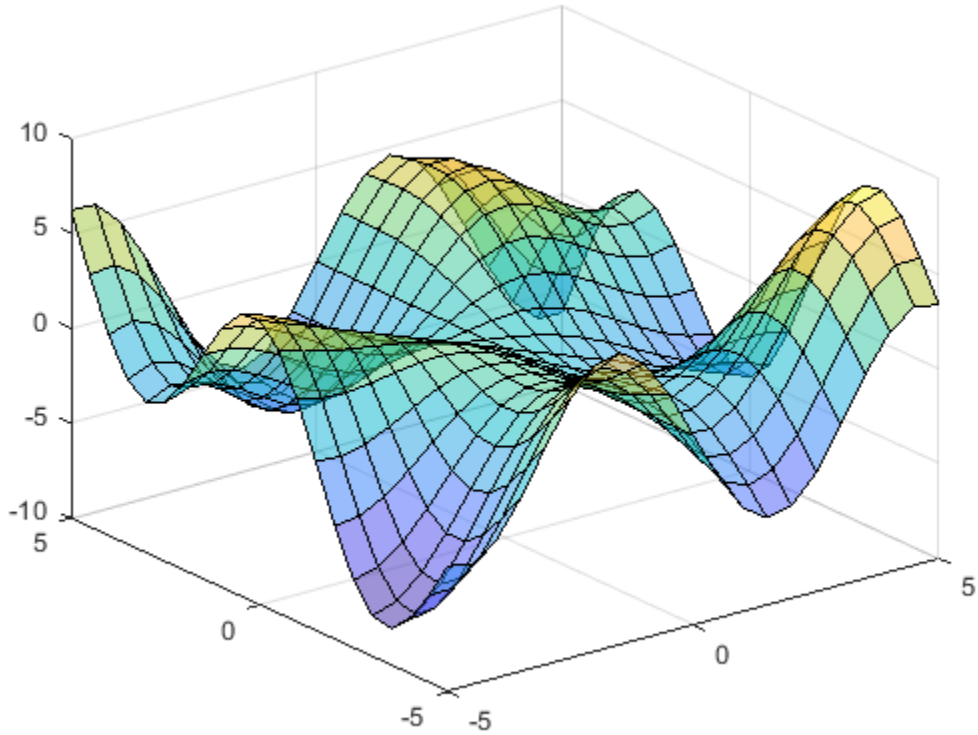
```
[X,Y,Z] = peaks(25);
C0(:,:,1) = zeros(25); % red
C0(:,:,2) = ones(25).*linspace(0.5,0.6,25); % green
C0(:,:,3) = ones(25).*linspace(0,1,25); % blue
surf(X,Y,Z,C0)
```



Modify Surface Plot Appearance

Create a semitransparent surface by specifying the `FaceAlpha` name-value pair with `0.5` as the value. To allow further modifications, assign the surface object to the variable `s`.

```
[X,Y] = meshgrid(-5:.5:5);
Z = Y.*sin(X) - X.*cos(Y);
s = surf(X,Y,Z,'FaceAlpha',0.5)
```

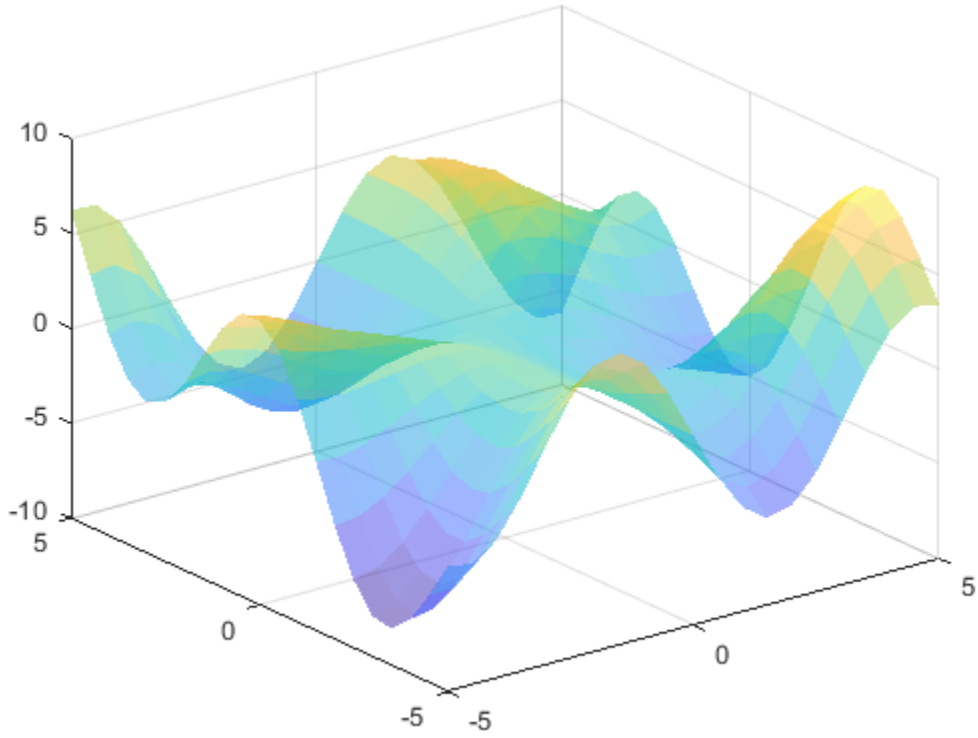



```
s =
  Surface with properties:

       EdgeColor: [0 0 0]
       LineStyle: '-'
       FaceColor: 'flat'
   FaceLighting: 'flat'
       FaceAlpha: 0.5000
           XData: [21x21 double]
           YData: [21x21 double]
           ZData: [21x21 double]
           CData: [21x21 double]
```

```
Show all properties
```

Use `s` to access and modify properties of the surface object after it is created. For example, hide the edges by setting the `EdgeColor` property.

```
s.EdgeColor = 'none';
```



Input Arguments

X — *x*-coordinates
matrix | vector

x-coordinates, specified as a matrix the same size as `Z`, or as a vector with length `n`, where `[m,n] = size(Z)`. If you do not specify values for `X` and `Y`, `surf` uses the vectors `(1:n)` and `(1:m)`.

You can use the `meshgrid` function to create `X` and `Y` matrices.

The `XData` property of the `Surface` object stores the *x*-coordinates.

Example: `X = 1:10`

Example: `X = [1 2 3; 1 2 3; 1 2 3]`

Example: `[X,Y] = meshgrid(-5:0.5:5)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`Y` — *y*-coordinates
matrix | vector

y-coordinates, specified as a matrix the same size as `Z` or as a vector with length `m`, where `[m,n] = size(Z)`. If you do not specify values for `X` and `Y`, `surf` uses the vectors `(1:n)` and `(1:m)`.

You can use the `meshgrid` function to create the `X` and `Y` matrices.

The `YData` property of the surface object stores the *y* -coordinates.

Example: `Y = 1:10`

Example: `Y = [1 1 1; 2 2 2; 3 3 3]`

Example: `[X,Y] = meshgrid(-5:0.5:5)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`Z` — *z*-coordinates
matrix

z-coordinates, specified as a matrix. `Z` must have at least two rows and two columns.

`Z` specifies the height of the surface plot at each *x*-*y* coordinate. If you do not specify the colors, then `Z` also specifies the surface colors.

The `ZData` property of the surface object stores the *z* -coordinates.

Example: `Z = [1 2 3; 4 5 6]`

Example: `Z = sin(x) + cos(y)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`C` — Color array
matrix | `m`-by-`n`-by-`3` array of RGB triplets

Color array, specified as an `m`-by-`n` matrix of colormap indices or as an `m`-by-`n`-by-`3` array of RGB triplets, where `Z` is `m`-by-`n`.

- To use colormap colors, specify `C` as a matrix. For each grid point on the surface, `C` indicates a color in the colormap. The `CDataMapping` property of the surface object controls how the values in `C` correspond to colors in the colormap.
- To use truecolor colors, specify `C` as an array of RGB triplets.

For more information, see “Differences Between Colormaps and Truecolor”.

The `CData` property of the surface object stores the color array. For additional control over the surface coloring, use the `FaceColor` and `EdgeColor` properties.

`ax` — Axes to plot in
axes object

Axes to plot in, specified as an `axes` object. If you do not specify the axes, then `surf` plots into the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

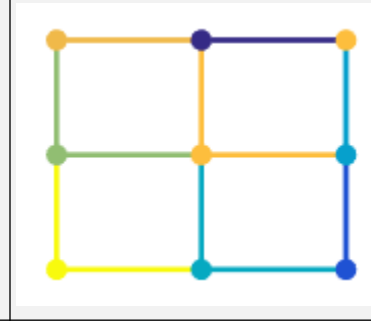
Note The properties listed here are only a subset. For a full list, see Chart Surface.

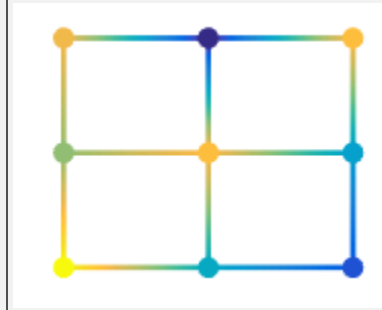
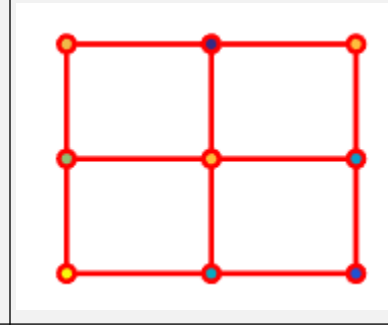
Example: `surf(X,Y,Z,'FaceAlpha',0.5,'EdgeColor','none')` creates a semitransparent surface with no edges drawn.

EdgeColor — Edge line color

`[0 0 0]` (default) | `'none'` | `'flat'` | `'interp'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Edge line color, specified as one of the values listed here. The default color of `[0 0 0]` corresponds to black edges.

Value	Description
`'none'`	Do not draw the edges.
`'flat'`	Use a different color for each edge based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color value at the first vertex of each face (in the positive *x* and *y* directions) determines the color for the adjacent edges. You cannot use this value when the `EdgeAlpha` property is set to `'interp'`. 

Value	Description
`'interp'`	Use interpolated coloring for each edge based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color varies across each edge by linearly interpolating the color values at the vertices. You cannot use this value when the `EdgeAlpha` property is set to `'flat'`. 
RGB triplet, hexadecimal color code, or color name	Use the specified color for all the edges. This option does not use the color values in the `CData` property. 

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

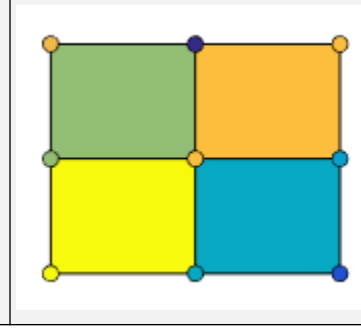
Line style, specified as one of the options listed in this table.

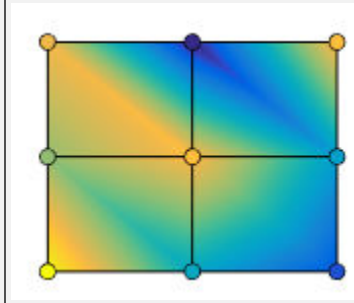
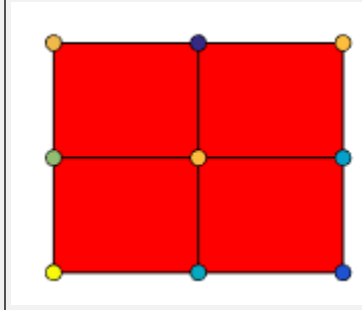
Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`FaceColor` — Face color

`'flat'` (default) | `'interp'` | `'none'` | `'texturemap'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Face color, specified as one of the values in this table.

Value	Description
`'flat'`	Use a different color for each face based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color value at the first vertex of each face (in the positive *x* and *y* directions) determines the color for the entire face. You cannot use this value when the `FaceAlpha` property is set to `'interp'`. 

Value	Description
`'interp'`	Use interpolated coloring for each face based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color varies across each face by interpolating the color values at the vertices. You cannot use this value when the `FaceAlpha` property is set to `'flat'`. 
RGB triplet, hexadecimal color code, or color name	Use the specified color for all the faces. This option does not use the color values in the `CData` property. 
`'texturemap'`	Transform the color data in `CData` so that it conforms to the surface.
`'none'`	Do not draw the faces.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

FaceAlpha — Face transparency
1 (default) | scalar in range `[0,1]` | `'flat'` | `'interp'` | `'texturemap'`

Face transparency, specified as one of these values:

- Scalar in range `[0,1]` — Use uniform transparency across all the faces. A value of `1` is fully opaque and `0` is completely transparent. Values between `0` and `1` are semitransparent. This option does not use the transparency values in the `AlphaData` property.
- `'flat'` — Use a different transparency for each face based on the values in the `AlphaData` property. The transparency value at the first vertex determines the transparency for the entire face. First you must specify the `AlphaData` property as a matrix the same size as the `ZData` property. The `FaceColor` property also must be set to `'flat'`.
- `'interp'` — Use interpolated transparency for each face based on the values in `AlphaData` property. The transparency varies across each face by interpolating the values at the vertices. First you must specify the `AlphaData` property as a matrix the same size as the `ZData` property. The `FaceColor` property also must be set to `'interp'`.
- `'texturemap'` — Transform the data in `AlphaData` so that it conforms to the surface.

FaceLighting — Effect of light objects on faces
`'flat'` (default) | `'gouraud'` | `'none'`

Effect of light objects on faces, specified as one of these values:

- `'flat'` — Apply light uniformly across each face. Use this value to view faceted objects.
- `'gouraud'` — Vary the light across the faces. Calculate the light at the vertices and then linearly interpolate the light across the faces. Use this value to view curved surfaces.
- `'none'` — Do not apply light from light objects to the faces.

To add a light object to the axes, use the `light` function.

Note The `'phong'` value has been removed. Use `'gouraud'` instead.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions

`colormap` | `imagesc` | `mesh` | `meshgrid` | `pcolor` | `shading` | `view`

Properties

Chart Surface

Topics

“Representing Data as a Surface”
“How Surface Plot Data Relates to a Colormap”

Introduced before R2006a

surfc

Contour plot under surface plot

Syntax

```
surfc(X,Y,Z)
surfc(X,Y,Z,C)

surfc(Z)
surfc(Z,C)

surfc(ax, ___ )
surfc( ___ ,Name,Value)
sc = surfc( ___ )
```

Description

`surfc(X,Y,Z)` creates a three-dimensional surface plot with a contour plot underneath. A surface plot is a three-dimensional surface that has solid edge colors and solid face colors. The function plots the values in matrix `Z` as heights above a grid in the *x*-*y* plane defined by `X` and `Y`. The color of the surface varies according to the heights specified by `Z`.

`surfc(X,Y,Z,C)` additionally specifies the surface color.

`surfc(Z)` creates a surface and contour plot and uses the column and row indices of the elements in `Z` as the *x*- and *y* -coordinates.

`surfc(Z,C)` additionally specifies the surface color.

`surfc(ax, ___ )` plots into the axes specified by `ax` instead of the current axes. Specify the axes as the first input argument.

`surfc( ___ ,Name,Value)` specifies surface properties using one or more name-value pair arguments. For example, `'FaceAlpha',0.5` creates a semitransparent surface.

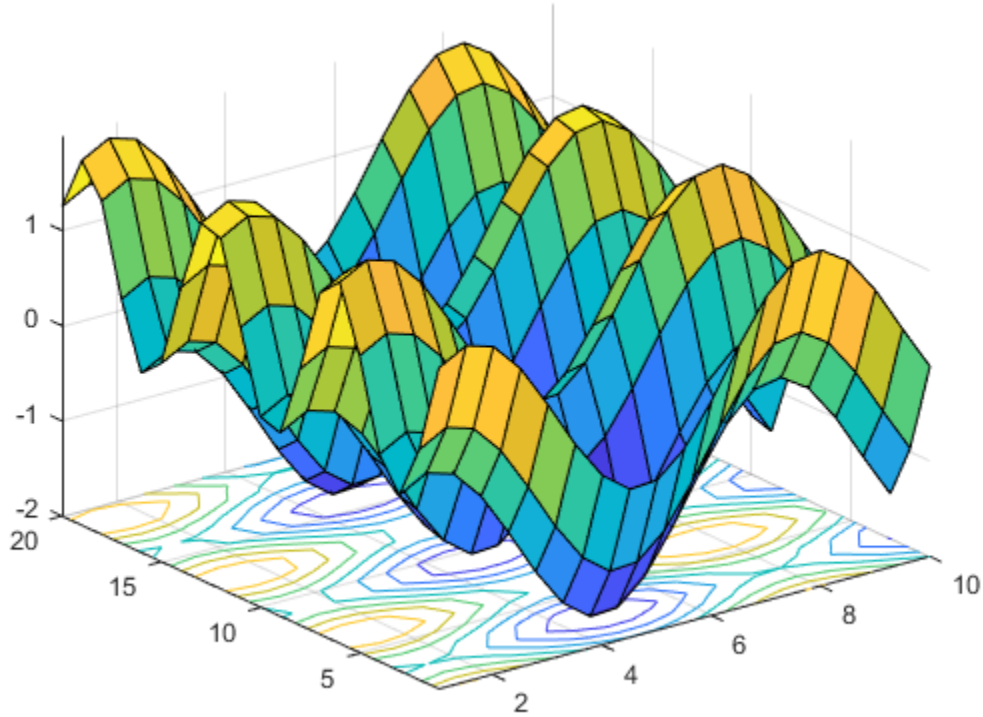
`sc = surfc( ___ )` returns a graphics array that includes the chart surface object and the contour object. Use `sc` to modify the surface and contour plots after they are created. For a list of properties, see Chart Surface and Contour Properties.

Examples

Display Contour Plot Under Surface Plot

Create three matrices of the same size. Then plot them as a surface and display a contour plot under the surface plot. The surface uses `Z` for both height and color.

```
[X,Y] = meshgrid(1:0.5:10,1:20);
Z = sin(X) + cos(Y);
surfc(X,Y,Z)
```

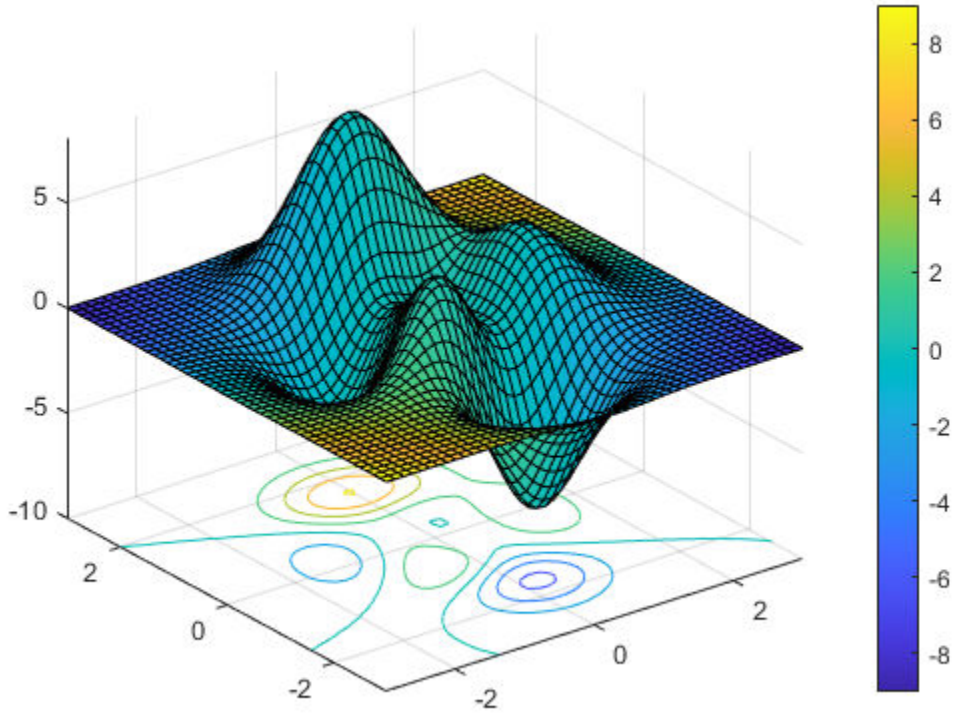


Specify Colormap Colors for Surface and Contour Plot

Specify the colors for a surface and a contour plot by including a fourth matrix input, `C`. The surface plot uses `Z` for height and `C` for color. Specify the colors using a *colormap*, which uses single numbers to stand for colors on a spectrum. When you use a colormap, `C` is the same size as `Z`. Add a color bar to the graph to show how the data values in `C` correspond to the colors in the colormap.

```
[X,Y] = meshgrid(-3:.125:3);
Z = peaks(X,Y);
C = X.*Y;
```

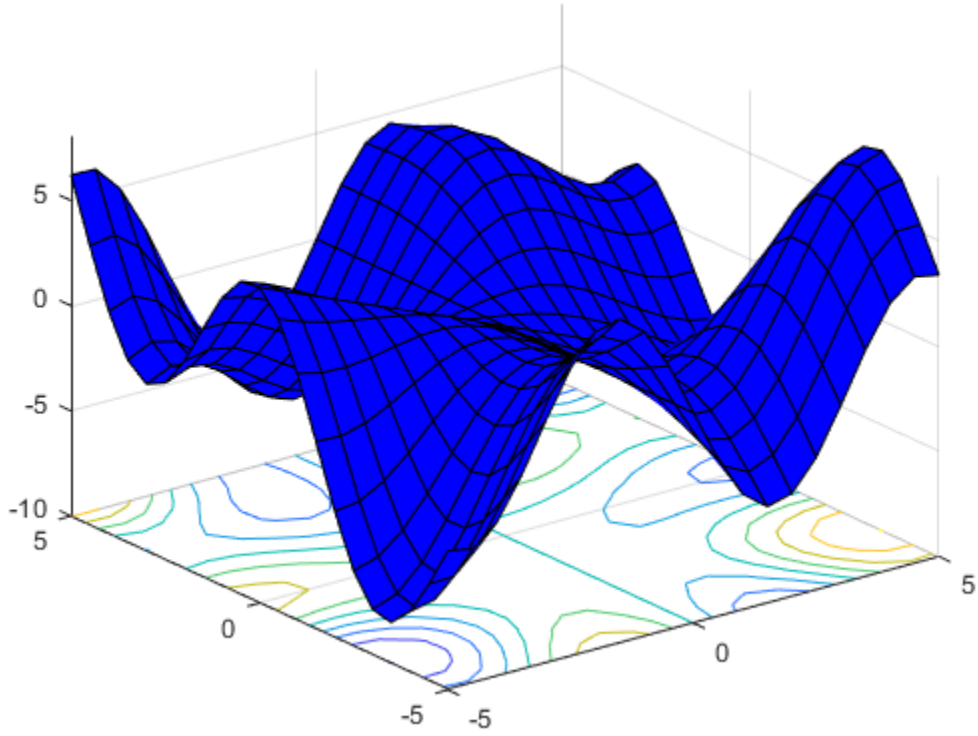
```
surfc(X,Y,Z,C)
colorbar
```



Modify Appearance of Surface and Contour Plots

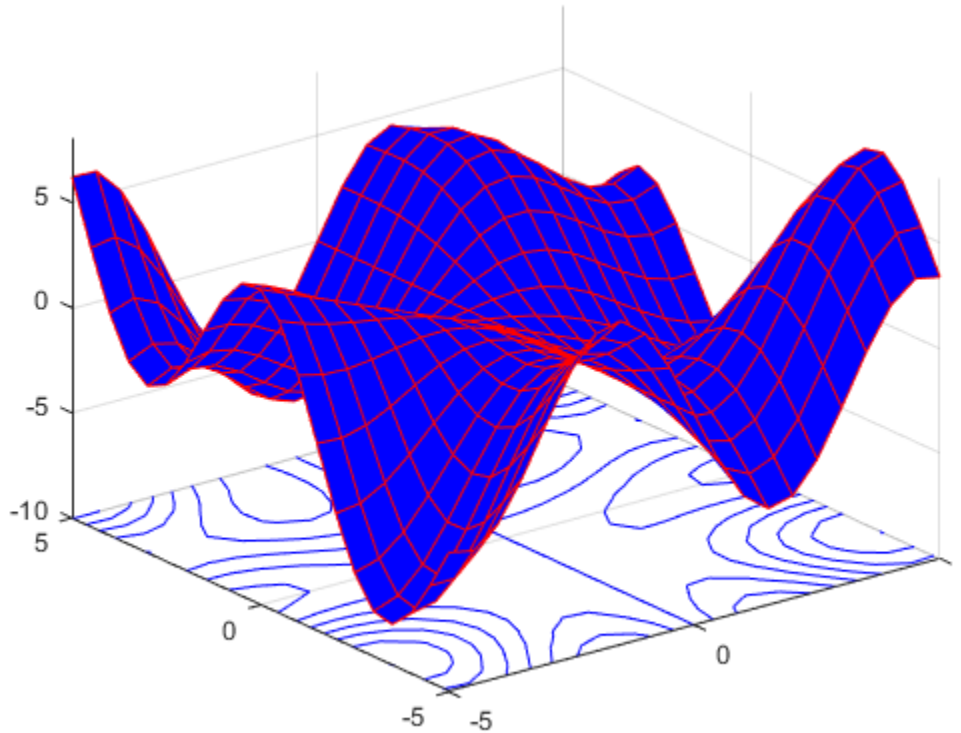
Create a blue surface plot with a contour plot underneath it by specifying the `FaceColor` name-value pair with `'b'` as the value. To allow further modifications, assign the graphics array containing the surface and contour objects to the variable `sc`.

```
[X,Y] = meshgrid(-5:.5:5);
Z = Y.*sin(X) - X.*cos(Y);
sc = surfc(X,Y,Z,'FaceColor','b');
```



Index into `sc` to access and modify properties of the surface and contour plots after they are created. The surface plot is accessible as `sc(1)` and the contour plot as `sc(2)`. For example, change the edge colors of the two plots by setting the `EdgeColor` properties.

```
sc(1).EdgeColor = 'r';
sc(2).EdgeColor = 'b';
```



Input Arguments

X — *x*-coordinates
matrix | vector

x-coordinates, specified as a matrix the same size as `Z`, or as a vector with length `n`, where `[m,n] = size(Z)`. If you do not specify values for `X` and `Y`, `surfc` uses the vectors `(1:n)` and `(1:m)`.

When `X` is a matrix, the values must be strictly increasing or decreasing along one dimension and remain constant along the other dimension. The dimension that varies

must be the opposite of the dimension that varies in `Y`. You can use the `meshgrid` function to create `X` and `Y` matrices.

When `X` is a vector, the values must be strictly increasing or decreasing.

The `XData` properties of the surface and contour objects store the *x*-coordinates.

Example: `X = 1:10`

Example: `X = [1 2 3; 1 2 3; 1 2 3]`

Example: `[X,Y] = meshgrid(-5:0.5:5)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical`

Y — *y*-coordinates
matrix | vector

y-coordinates, specified as a matrix the same size as `Z` or as a vector with length `m`, where `[m,n] = size(Z)`. If you do not specify values for `X` and `Y`, `surfc` uses the vectors `(1:n)` and `(1:m)`.

When `Y` is a matrix, the values must be strictly increasing or decreasing along one dimension and remain constant along the other dimension. The dimension that varies must be the opposite of the dimension that varies in `X`. You can use the `meshgrid` function to create `X` and `Y` matrices.

When `Y` is a vector, the values must be strictly increasing or decreasing.

The `YData` properties of the surface and contour objects store the *y*-coordinates.

Example: `Y = 1:10`

Example: `Y = [1 1 1; 2 2 2; 3 3 3]`

Example: `[X,Y] = meshgrid(-5:0.5:5)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical`

Z — *z*-coordinates
matrix

z-coordinates, specified as a matrix. `Z` must have at least two rows and two columns.

`Z` specifies the height of the surface plot at each *x*-*y* coordinate. If you do not specify the colors, then `Z` also specifies the surface colors.

The `ZData` properties of the surface and contour objects store the *z*-coordinates.

Example: `Z = [1 2 3; 4 5 6]`

Example: `Z = sin(x) + cos(y)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical`

`C` — Color array
matrix | `m`-by-`n`-by-`3` array of RGB triplets

Color array, specified as an `m`-by-`n` matrix of colormap indices or as an `m`-by-`n`-by-`3` array of RGB triplets, where `Z` is `m`-by-`n`.

- To use colormap colors, specify `C` as a matrix. For each grid point on the surface, `C` indicates a color in the colormap. The `CDataMapping` property of the surface object controls how the values in `C` correspond to colors in the colormap.
- To use truecolor colors, specify `C` as an array of RGB triplets.

For more information, see “Differences Between Colormaps and Truecolor”.

The `CData` property of the surface object stores the color array. For additional control over the surface coloring, use the `FaceColor` and `EdgeColor` properties.

`ax` — Axes to plot in
axes object

Axes to plot in, specified as an `axes` object. If you do not specify the axes, then `surfc` plots into the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

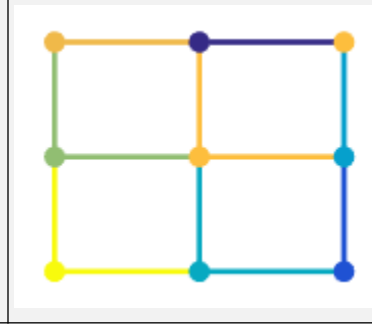
Note The properties listed here are only a subset. For a full list, see Chart Surface.

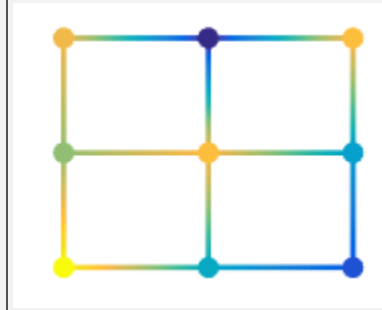
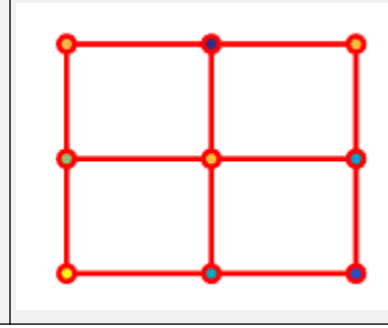
Example: `surfc(X,Y,Z,'FaceAlpha',0.5,'EdgeColor','none')` creates a semitransparent surface with no edges drawn.

`EdgeColor` — Edge line color

`[0 0 0]` (default) | `'none'` | `'flat'` | `'interp'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Edge line color, specified as one of the values listed here. The default color of `[0 0 0]` corresponds to black edges.

Value	Description
`'none'`	Do not draw the edges.
`'flat'`	Use a different color for each edge based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color value at the first vertex of each face (in the positive *x* and *y* directions) determines the color for the adjacent edges. You cannot use this value when the `EdgeAlpha` property is set to `'interp'`. 

Value	Description
`'interp'`	Use interpolated coloring for each edge based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color varies across each edge by linearly interpolating the color values at the vertices. You cannot use this value when the `EdgeAlpha` property is set to `'flat'`. 
RGB triplet, hexadecimal color code, or color name	Use the specified color for all the edges. This option does not use the color values in the `CData` property. 

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

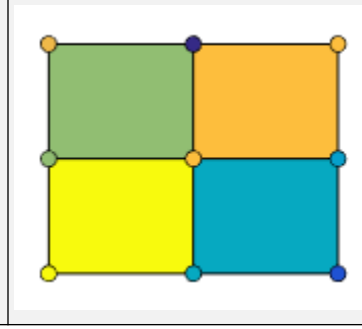
Line style, specified as one of the options listed in this table.

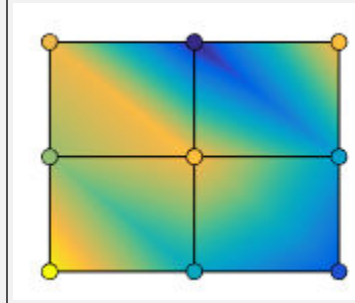
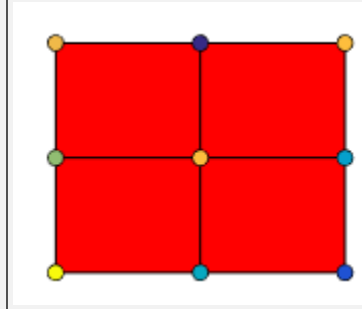
Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`FaceColor` — Face color

`'flat'` (default) | `'interp'` | `'none'` | `'texturemap'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Face color, specified as one of the values in this table.

Value	Description
`'flat'`	Use a different color for each face based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color value at the first vertex of each face (in the positive *x* and *y* directions) determines the color for the entire face. You cannot use this value when the `FaceAlpha` property is set to `'interp'`. 

Value	Description
`'interp'`	Use interpolated coloring for each face based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color varies across each face by interpolating the color values at the vertices. You cannot use this value when the `FaceAlpha` property is set to `'flat'`. 
RGB triplet, hexadecimal color code, or color name	Use the specified color for all the faces. This option does not use the color values in the `CData` property. 
`'texturemap'`	Transform the color data in `CData` so that it conforms to the surface.
`'none'`	Do not draw the faces.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

FaceAlpha — Face transparency
1 (default) | scalar in range `[0,1]` | `'flat'` | `'interp'` | `'texturemap'`

Face transparency, specified as one of these values:

- Scalar in range `[0,1]` — Use uniform transparency across all the faces. A value of `1` is fully opaque and `0` is completely transparent. Values between `0` and `1` are semitransparent. This option does not use the transparency values in the `AlphaData` property.
- `'flat'` — Use a different transparency for each face based on the values in the `AlphaData` property. The transparency value at the first vertex determines the transparency for the entire face. First you must specify the `AlphaData` property as a matrix the same size as the `ZData` property. The `FaceColor` property also must be set to `'flat'`.
- `'interp'` — Use interpolated transparency for each face based on the values in `AlphaData` property. The transparency varies across each face by interpolating the values at the vertices. First you must specify the `AlphaData` property as a matrix the same size as the `ZData` property. The `FaceColor` property also must be set to `'interp'`.
- `'texturemap'` — Transform the data in `AlphaData` so that it conforms to the surface.

FaceLighting — Effect of light objects on faces
`'flat'` (default) | `'gouraud'` | `'none'`

Effect of light objects on faces, specified as one of these values:

- `'flat'` — Apply light uniformly across each face. Use this value to view faceted objects.
- `'gouraud'` — Vary the light across the faces. Calculate the light at the vertices and then linearly interpolate the light across the faces. Use this value to view curved surfaces.
- `'none'` — Do not apply light from light objects to the faces.

To add a light object to the axes, use the `light` function.

Note The `'phong'` value has been removed. Use `'gouraud'` instead.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions

`colormap` | `contour` | `imagesc` | `meshc` | `meshgrid` | `pcolor` | `shading` | `surf` | `view`

Properties

Chart Surface | Contour

Topics

“Representing Data as a Surface”
“How Surface Plot Data Relates to a Colormap”

Introduced before R2006a

surf2patch

Convert surface data to patch data

Syntax

```
fvc = surf2patch(h)
fvc = surf2patch(Z)
fvc = surf2patch(Z,C)
fvc = surf2patch(X,Y,Z)
fvc = surf2patch(X,Y,Z,C)
fvc = surf2patch(...,'triangles')
[f,v,c] = surf2patch(...)
```

Description

`fvc = surf2patch(h)` converts the geometry and color data from the `surface` object, `h`, into patch format. The output, `fvc`, is a structure containing the face, vertex, and color data. You can pass this structure directly to the `patch` command.

`fvc = surf2patch(Z)` calculates the patch data from the surface's `ZData` matrix `Z`.

`fvc = surf2patch(Z,C)` calculates the patch data from the surface's `ZData` and `CData` matrices `Z` and `C`.

`fvc = surf2patch(X,Y,Z)` calculates the patch data from the surface's `XData`, `YData`, and `ZData` matrices `X`, `Y`, and `Z`.

`fvc = surf2patch(X,Y,Z,C)` calculates the patch data from the surface's `XData`, `YData`, `ZData`, and `CData` matrices `X`, `Y`, `Z`, and `C`.

`fvc = surf2patch(...,'triangles')` creates triangular faces instead of the quadrilaterals that compose surfaces.

`[f,v,c] = surf2patch(...)` returns the face, vertex, and color data in the three arrays `f`, `v`, and `c` instead of a struct.

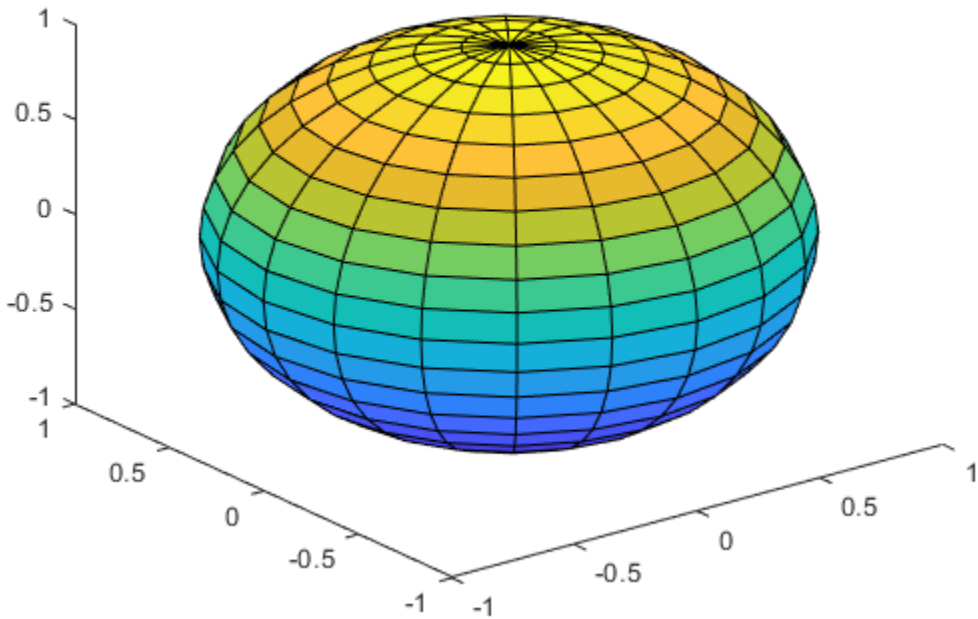
Examples

Calculate Patch Data from Surface Data

Use the `sphere` command to generate the `XData`, `YData`, and `ZData` of a surface. Then, calculate the patch data. Pass the `ZData` (`z`) to `surf2patch` as both the third and fourth arguments - the third argument is the `ZData` and the fourth argument is taken as the `CData`. You must do this since the `patch` command does not automatically use the z-coordinate data for the color data, as does the surface command.

Since patch is a low-level command, you must set the view and shading to produce the same results produced by the `surf` command.

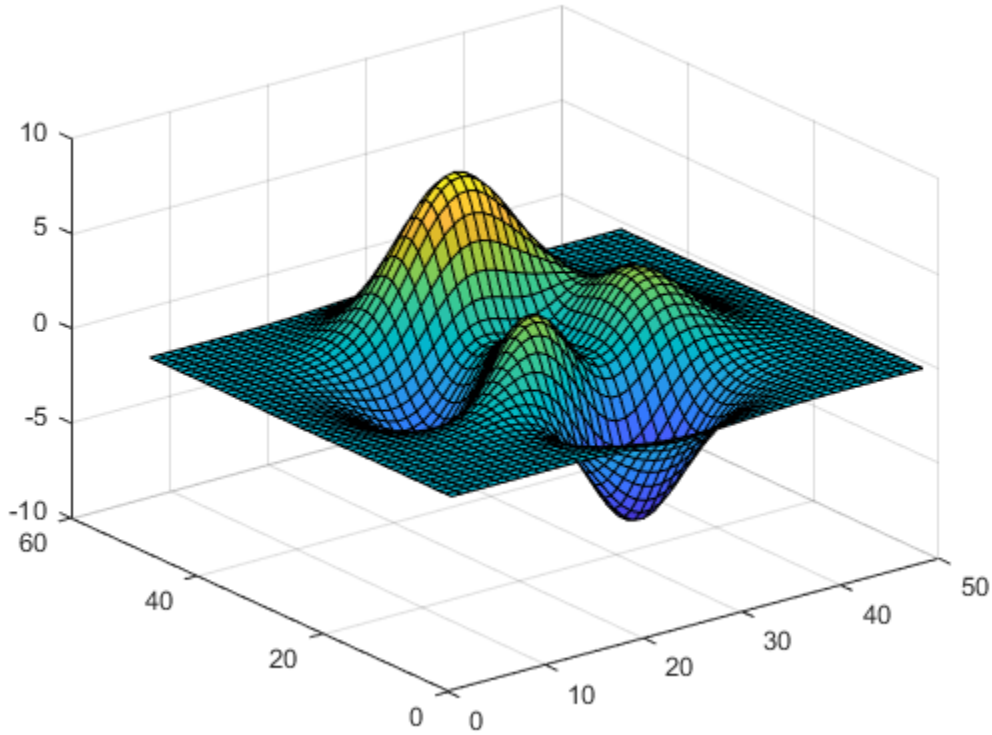
```
[x,y,z] = sphere;
figure
patch(surf2patch(x,y,z,z));
shading faceted;
view(3)
```



Calculate Patch Data Using Surface Object

Calculate face, vertex, and color data from a surface whose handle has been passed as an argument.

```
figure
s = surf(peaks);
patch(surf2patch(s));
delete(s)
shading faceted;
view(3)
```



See Also

patch | reducepatch | shrinkfaces | surf | surface

Introduced before R2006a

surface

Primitive surface plot

Syntax

```
surface(X,Y,Z)
surface(X,Y,Z,C)

surface(Z)
surface(Z,C)

surface(ax, ___ )
surface( ___ ,Name,Value)
s = surface( ___ )
```

Description

`surface(X,Y,Z)` creates a primitive, three-dimensional surface plot. The function plots the values in matrix `Z` as heights above a grid in the *x*-*y* plane defined by `X` and `Y`. The color of the surface varies according to the heights specified by `Z`.

Unlike the `surf` function, the primitive `surface` function does not call `newplot` before plotting and does not respect the value of the `NextPlot` property for the figure or axes. Instead, it adds the surface plot to the current axes without deleting other graphics objects or resetting axes properties.

`surface(X,Y,Z,C)` additionally specifies the surface color.

`surface(Z)` creates a primitive surface plot and uses the column and row indices of the elements in `Z` as the *x*- and *y*-coordinates.

`surface(Z,C)` additionally specifies the surface color.

`surface(ax, ___ )` plots into the axes specified by `ax` instead of the current axes. Specify the axes as the first input argument.

`surface( ___ ,Name,Value)` specifies surface properties using one or more name-value pair arguments. For example, `'FaceAlpha',0.5` creates a semitransparent surface.

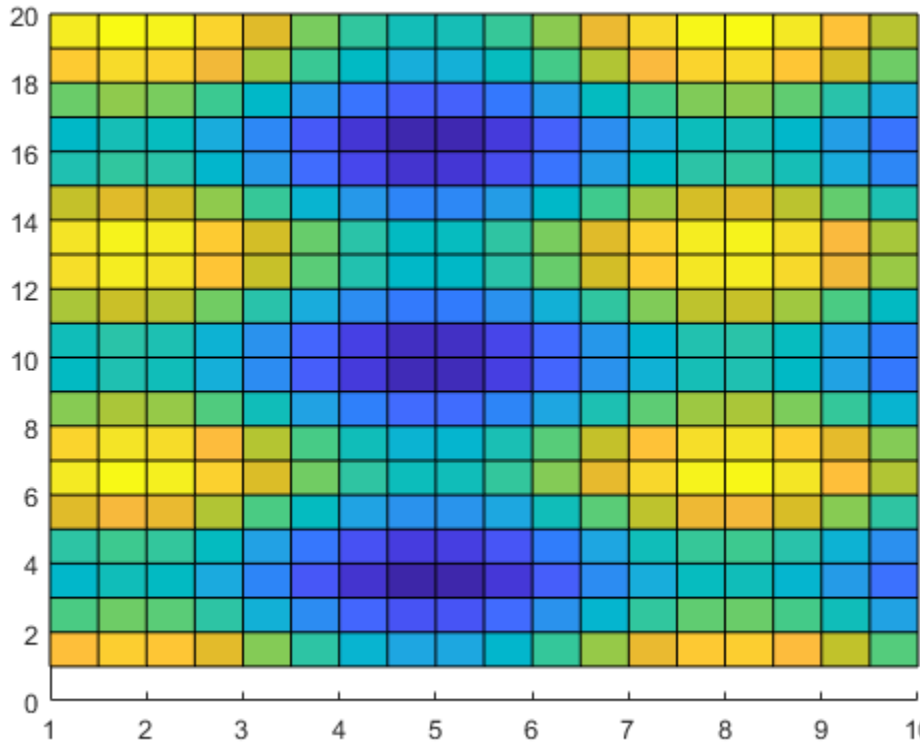
`s = surface( ___ )` returns the chart primitive surface object. Use `s` to modify the surface after it is created. For a list of properties, see Surface Properties.

Examples

Create Surface Plot

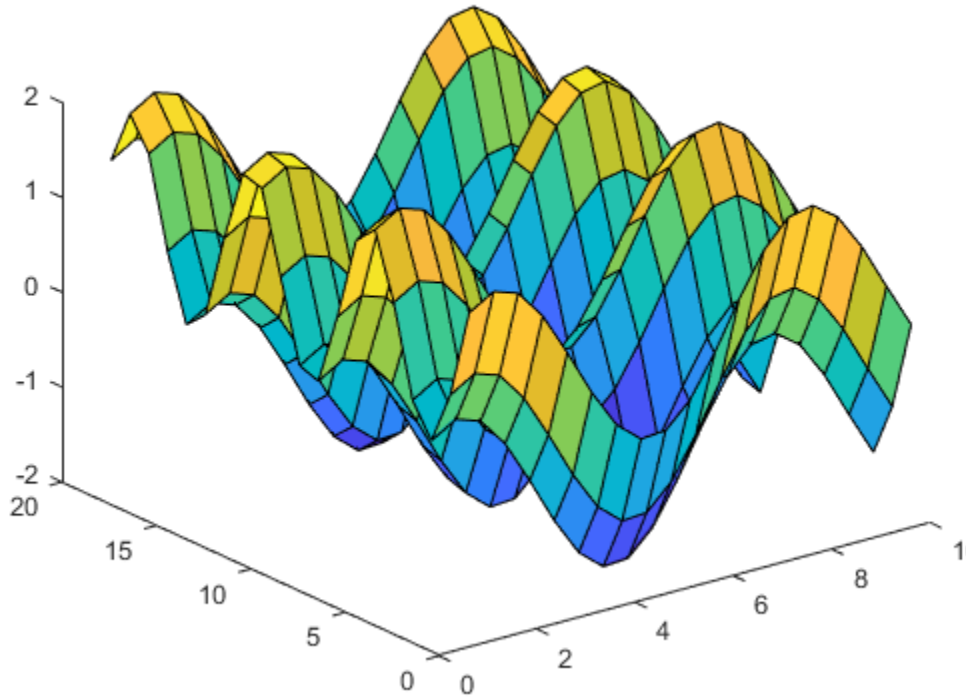
Create three matrices of the same size. Then plot them as a surface. The surface uses `Z` for both height and color.

```
[X,Y] = meshgrid(1:0.5:10,1:20);
Z = sin(X) + cos(Y);
surface(X,Y,Z)
```



By default, `surface` displays the surface plot using the default two-dimensional view. Set the view of the plot to the default three-dimensional view.

```
view(3)
```

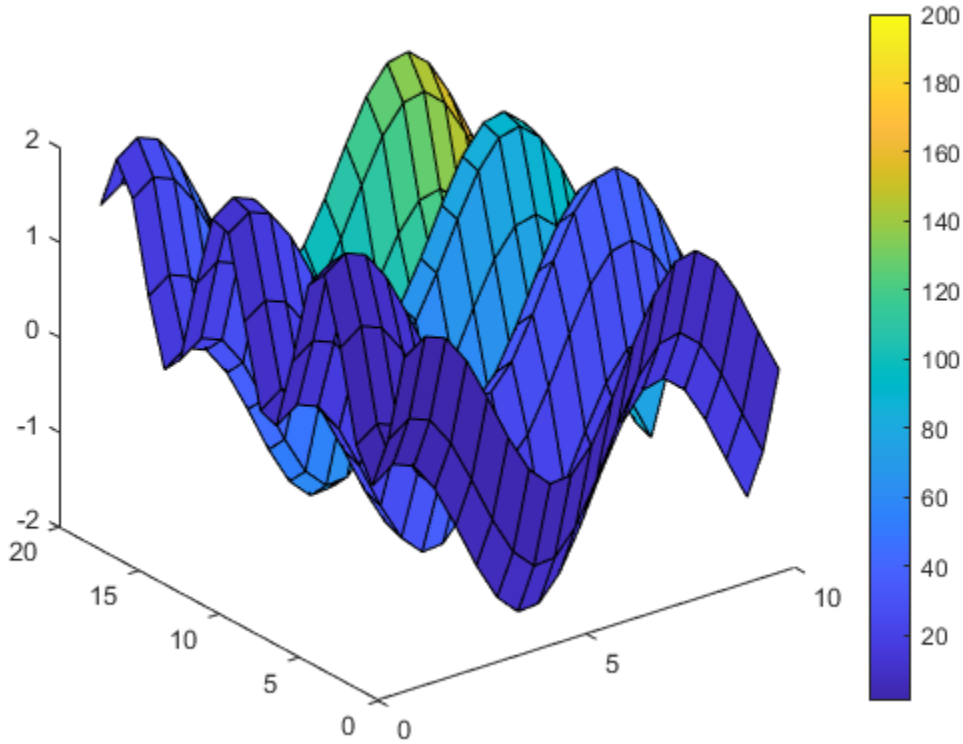


Specify Colormap Colors for Surface Plot

Specify the colors for a surface plot by including a fourth matrix input, `C`. The mesh plot uses `Z` for height and `C` for color. Specify the colors using a *colormap*, which uses single numbers to stand for colors on a spectrum. When you use a colormap, `C` is the same size as `Z`. Add a color bar to the graph to show how the data values in `C` correspond to the colors in the colormap, and set the view of the plot to the default three-dimensional view.

```
[X,Y] = meshgrid(1:0.5:10,1:20);
Z = sin(X) + cos(Y);
C = X.*Y;
```

```
surface(X,Y,Z,C)
colorbar
view(3)
```

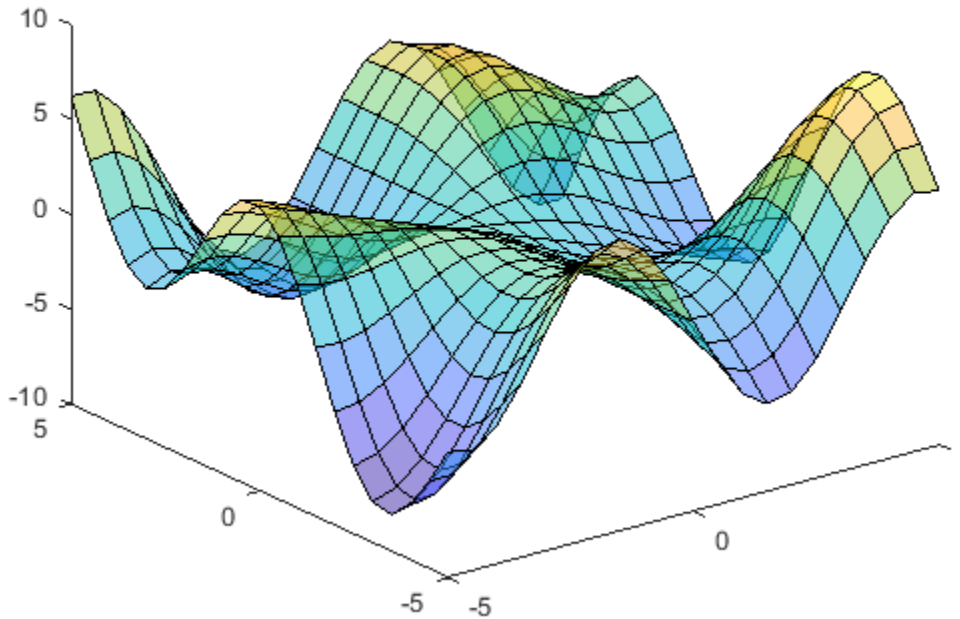


Modify Surface Plot Appearance

Create a semitransparent surface by specifying the `FaceAlpha` name-value pair with `0.5` as the value. To allow further modifications, assign the surface object to the variable `s`.

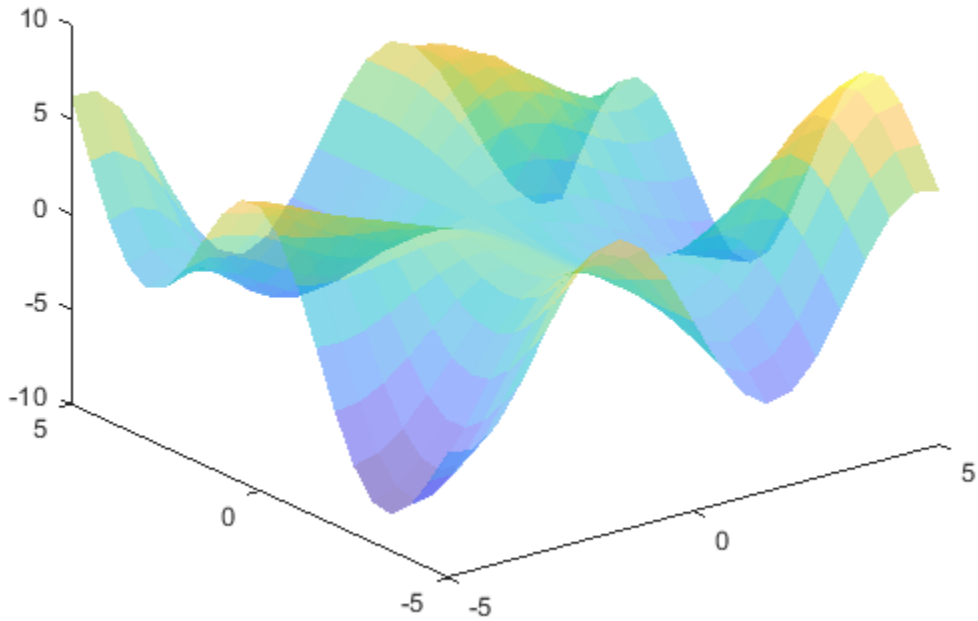
```
[X,Y] = meshgrid(-5:.5:5);
Z = Y.*sin(X) - X.*cos(Y);
```

```
s = surface(X,Y,Z,'FaceAlpha',0.5);
view(3)
```



Use `s` to access and modify properties of the surface object after it is created. For example, hide the edges by setting the `EdgeColor` property.

```
s.EdgeColor = 'none';
```



Display Image Along Surface Plot

Create a surface and display an image along it.

Create three matrices of the same size.

```
[pX,pY,pZ] = peaks(25);
```

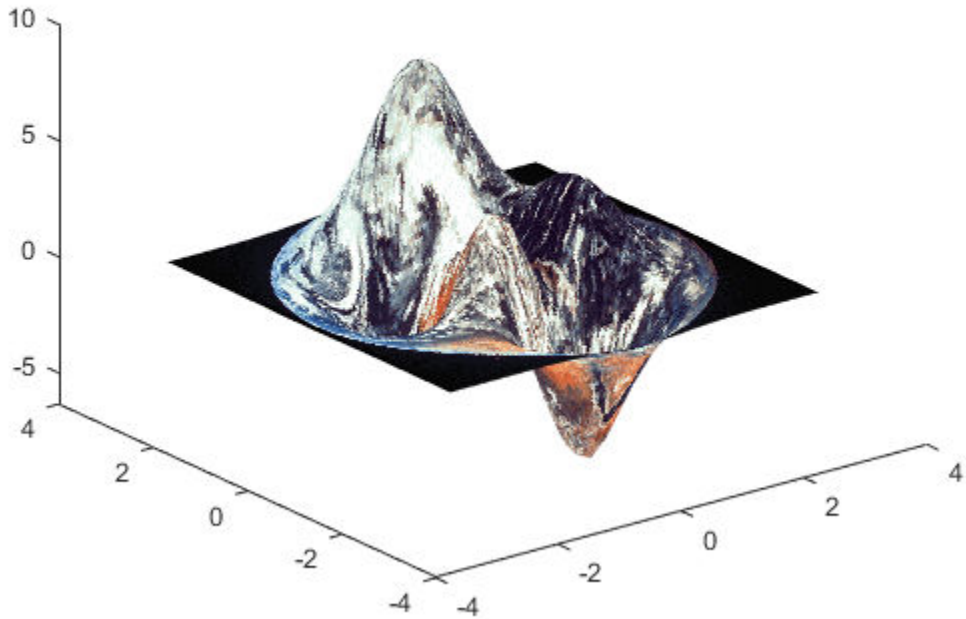
Load a data set containing an image of the Earth. The image data appears in a workspace variable `X`, and the associated colormap appears in `map`.

```
load earth
who
```

```
Your variables are:

X    map  pX   pY   pZ
```

Create a surface plot and display the image along the surface. Since the surface data `pZ` and the color data `X` have different sizes, set the surface `FaceColor` to `'texturemap'`. Set the view of the plot to the default three-dimensional view.

```
surface(pX,pY,pZ,X,'FaceColor','texturemap', ...
    'EdgeColor','none','CDataMapping','direct')
colormap(map)
view(3)
```



Input Arguments

X — *x*-coordinates
matrix | vector

x-coordinates, specified as a matrix the same size as `Z`, or as a vector with length `n`, where `[m,n] = size(Z)`. If you do not specify values for `X` and `Y`, `surface` uses the vectors `(1:n)` and `(1:m)`.

You can use the `meshgrid` function to create `X` and `Y` matrices.

The `XData` property of the `Surface` object stores the *x*-coordinates.

Example: `X = 1:10`

Example: `X = [1 2 3; 1 2 3; 1 2 3]`

Example: `[X,Y] = meshgrid(-5:0.5:5)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Y — *y*-coordinates
matrix | vector

y-coordinates, specified as a matrix the same size as `Z` or as a vector with length `m`, where `[m,n] = size(Z)`. If you do not specify values for `X` and `Y`, `surface` uses the vectors `(1:n)` and `(1:m)`.

You can use the `meshgrid` function to create the `X` and `Y` matrices.

The `YData` property of the surface object stores the *y*-coordinates.

Example: `Y = 1:10`

Example: `Y = [1 1 1; 2 2 2; 3 3 3]`

Example: `[X,Y] = meshgrid(-5:0.5:5)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Z — *z*-coordinates
matrix

z-coordinates, specified as a matrix. `Z` must have at least two rows and two columns.

`Z` specifies the height of the surface plot at each *x*-*y* coordinate. If you do not specify the colors, then `Z` also specifies the surface colors.

The `ZData` property of the surface object stores the *z*-coordinates.

Example: `Z = [1 2 3; 4 5 6]`

Example: `Z = sin(x) + cos(y)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

C — Color array
matrix | m-by-n-by-3 array of RGB triplets

Color array, specified as an m-by-n matrix of colormap indices or as an m-by-n-by-3 array of RGB triplets, where Z is m-by-n.

- To use colormap colors, specify C as a matrix. For each grid point on the surface, C indicates a color in the colormap. The CDataMapping property of the surface object controls how the values in C correspond to colors in the colormap.
- To use truecolor colors, specify C as an array of RGB triplets.

For more information, see “Differences Between Colormaps and Truecolor”.

The CData property of the surface object stores the color array. For additional control over the surface coloring, use the FaceColor and EdgeColor properties.

ax — Axes to plot in
axes object

Axes to plot in, specified as an axes object. If you do not specify the axes, then surface plots into the current axes.

Name-Value Pair Arguments

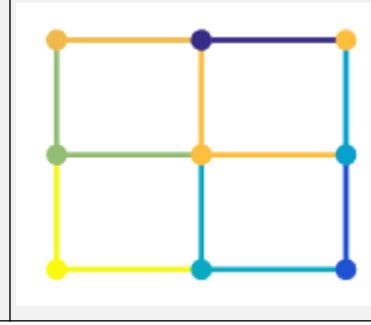
Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

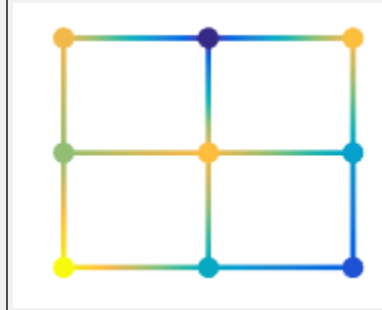
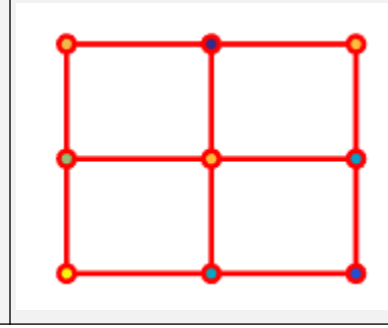
Note The properties listed here are only a subset. For a full list, see Surface Properties.

Example: surface(X,Y,Z,'FaceAlpha',0.5,'EdgeColor','none') creates a semitransparent surface with no edges drawn.

EdgeColor — Edge line color
[0 0 0] (default) | 'none' | 'flat' | 'interp' | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...

Edge line color, specified as one of the values listed here. The default color of [0 0 0] corresponds to black edges.

Value	Description
`'none'`	Do not draw the edges.
`'flat'`	Use a different color for each edge based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color value at the first vertex of each face (in the positive *x* and *y* directions) determines the color for the adjacent edges. You cannot use this value when the `EdgeAlpha` property is set to `'interp'`. 

Value	Description
`'interp'`	Use interpolated coloring for each edge based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color varies across each edge by linearly interpolating the color values at the vertices. You cannot use this value when the `EdgeAlpha` property is set to `'flat'`. 
RGB triplet, hexadecimal color code, or color name	Use the specified color for all the edges. This option does not use the color values in the `CData` property. 

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

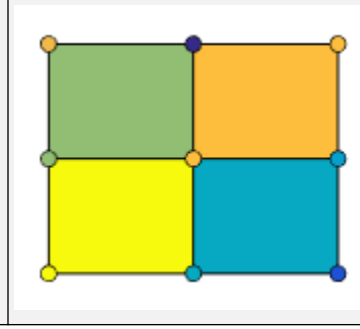
Line style, specified as one of the options listed in this table.

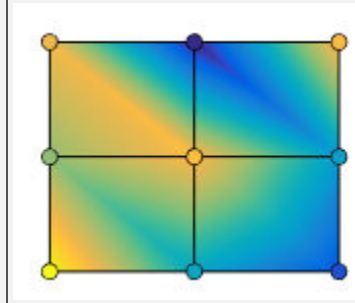
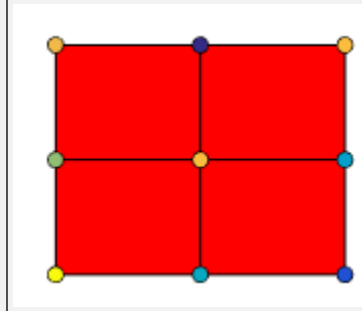
Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`FaceColor` — Face color

`'flat'` (default) | `'interp'` | `'none'` | `'texturemap'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Face color, specified as one of the values in this table.

Value	Description
`'flat'`	Use a different color for each face based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color value at the first vertex of each face (in the positive *x* and *y* directions) determines the color for the entire face. You cannot use this value when the `FaceAlpha` property is set to `'interp'`. 

Value	Description
`'interp'`	Use interpolated coloring for each face based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color varies across each face by interpolating the color values at the vertices. You cannot use this value when the `FaceAlpha` property is set to `'flat'`. 
RGB triplet, hexadecimal color code, or color name	Use the specified color for all the faces. This option does not use the color values in the `CData` property. 
`'texturemap'`	Transform the color data in `CData` so that it conforms to the surface.
`'none'`	Do not draw the faces.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

FaceAlpha — Face transparency
1 (default) | scalar in range `[0,1]` | `'flat'` | `'interp'` | `'texturemap'`

Face transparency, specified as one of these values:

- Scalar in range `[0,1]` — Use uniform transparency across all the faces. A value of `1` is fully opaque and `0` is completely transparent. Values between `0` and `1` are semitransparent. This option does not use the transparency values in the `AlphaData` property.
- `'flat'` — Use a different transparency for each face based on the values in the `AlphaData` property. The transparency value at the first vertex determines the transparency for the entire face. First you must specify the `AlphaData` property as a matrix the same size as the `ZData` property. The `FaceColor` property also must be set to `'flat'`.
- `'interp'` — Use interpolated transparency for each face based on the values in `AlphaData` property. The transparency varies across each face by interpolating the values at the vertices. First you must specify the `AlphaData` property as a matrix the same size as the `ZData` property. The `FaceColor` property also must be set to `'interp'`.
- `'texturemap'` — Transform the data in `AlphaData` so that it conforms to the surface.

FaceLighting — Effect of light objects on faces
`'flat'` (default) | `'gouraud'` | `'none'`

Effect of light objects on faces, specified as one of these values:

- `'flat'` — Apply light uniformly across each face. Use this value to view faceted objects.
- `'gouraud'` — Vary the light across the faces. Calculate the light at the vertices and then linearly interpolate the light across the faces. Use this value to view curved surfaces.
- `'none'` — Do not apply light from light objects to the faces.

To add a light object to the axes, use the `light` function.

Note The `'phong'` value has been removed. Use `'gouraud'` instead.

See Also

Functions
`colormap` | `imagesc` | `mesh` | `meshgrid` | `pcolor` | `shading` | `surf` | `view`

Properties
Primitive Surface

Topics

"Representing Data as a Surface"
"How Surface Plot Data Relates to a Colormap"

Introduced before R2006a

surfl

Surface plot with colormap-based lighting

Syntax

```
surfl(X,Y,Z)
surfl(Z)
surfl( ___ ,'light')
surfl( ___ ,s)
surfl(X,Y,Z,s,k)

surfl(ax, ___ )
s = surfl( ___ )
```

Description

`surfl(X,Y,Z)` creates a three-dimensional surface plot with highlights from a light source. The function plots the values in matrix `Z` as heights above a grid in the *x*-*y* plane defined by `X` and `Y`. The function uses the default direction for the light source and the default lighting coefficients for the shading model. This sets the color data for the surface to be the reflectance of the surface.

Because of the way surface-normal vectors are computed, `surfl` requires matrices that are at least 3-by-3.

`surfl(Z)` creates a surface and uses the column and row indices of the elements in `Z` as the *x*- and *y*-coordinates.

`surfl( ___ ,'light')` creates a surface with highlights from a MATLAB light object. This produces different results from the default colormap-based lighting method. Specify the `'light'` object as the last input argument.

`surfl( ___ ,s)` additionally specifies the direction of the light source.

`surfl(X,Y,Z,s,k)` additionally specifies the reflectance constant.

`surfl(ax, ___ )` plots into the axes specified by `ax` instead of the current axes. Specify the axes as the first input argument.

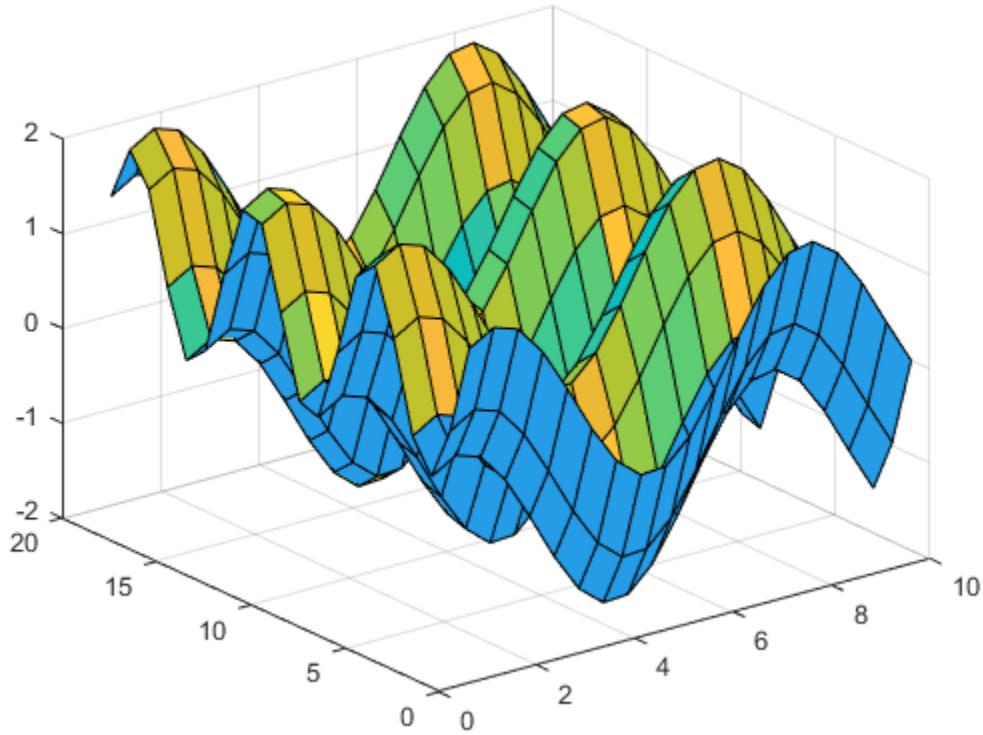
`s = surfl( ___ )` returns the chart surface object. If the light source is specified as a light object using the `'light'` option, then `s` is returned as a graphics array that includes the chart surface object and the light object. Use `s` to modify the surface and light object after it is created. For a list of properties, see Chart Surface and Light Properties.

Examples

Create Surface Plot With Colormap-Based Lighting

Create three matrices of the same size. Then plot them as a surface using colormap-based lighting. The surface uses `Z` for height and both `Z` and the light source for color.

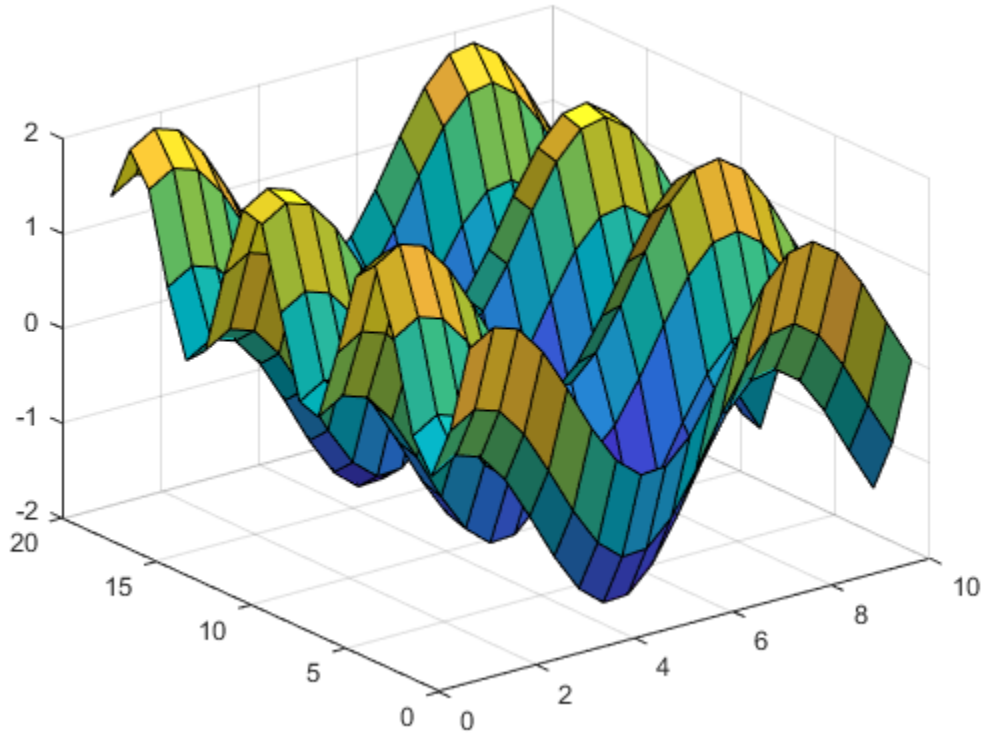
```
[X,Y] = meshgrid(1:0.5:10,1:20);
Z = sin(X) + cos(Y);
surfl(X,Y,Z)
```



Create Surface Plot With Light Object

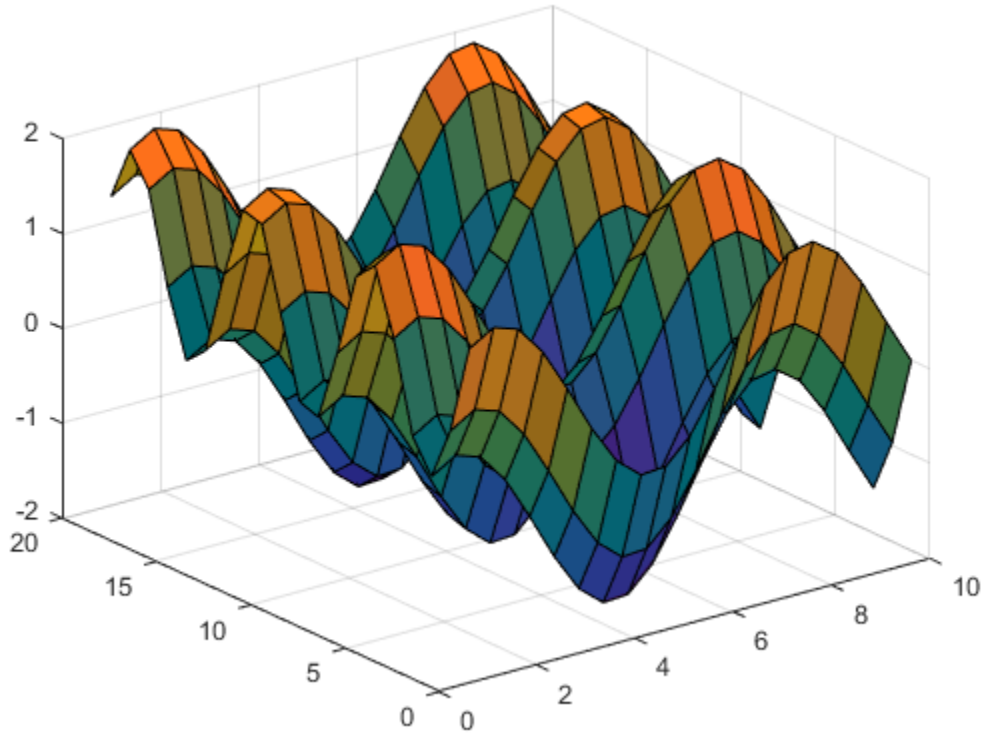
Create three matrices of the same size. Then plot them as a surface with highlights from a MATLAB® light object. The surface uses Z for height and both Z and the light object for color. The function returns an array containing a surface object and a lighting object. Assign it to the variable `sl`.

```
[X,Y] = meshgrid(1:0.5:10,1:20);
Z = sin(X) + cos(Y);
sl = surfl(X,Y,Z,'light');
```



Index into `sl` to access and modify properties of the surface object and the light object after they are created. The surface plot is accessible as `sl(1)` and the light object as `sl(2)`. For example, change the color of the light by setting the `Color` property of the light object.

```
sl(2).Color = 'r';
```



Specify Light Direction and Reflectance for Surface Plot

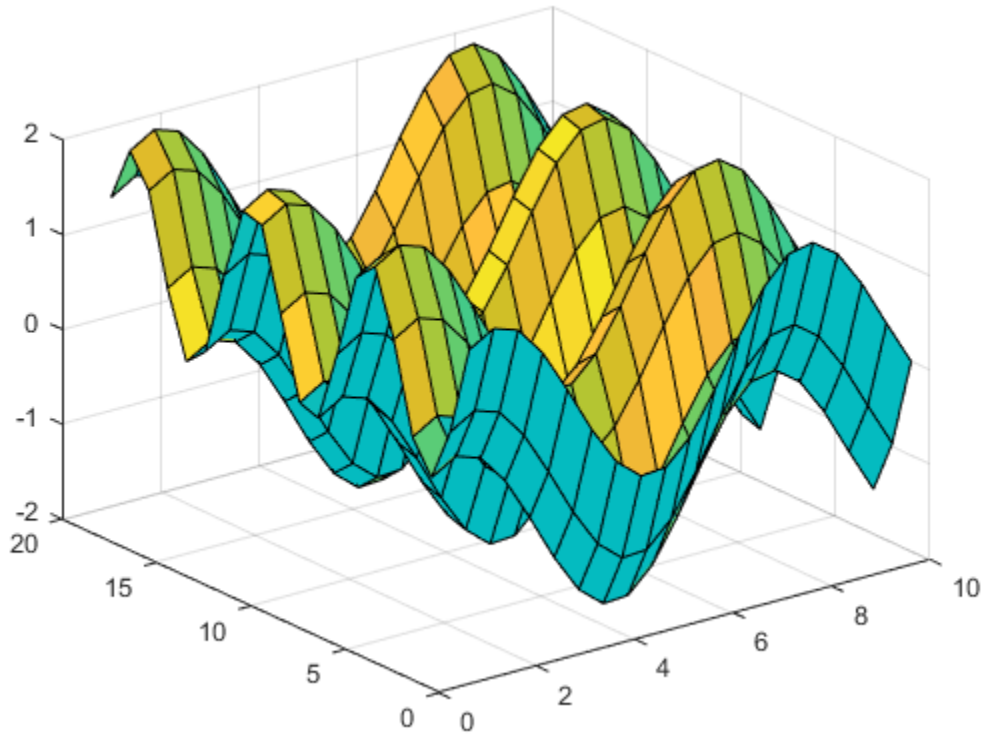
Create three matrices of the same size to plot as a surface. Specify the direction of the light source to have an azimuth of 45 degrees and an elevation of 20 degrees. Increase the reflectance of the surface by increasing the contribution of ambient light and decreasing the contibutions of diffused and specular reflection. Assign the surface object to the variable `sl`.

```
[X,Y] = meshgrid(1:0.5:10,1:20);
Z = sin(X) + cos(Y);
```

```
s = [-45 20];
k = [.65 .4 .3 10];
```

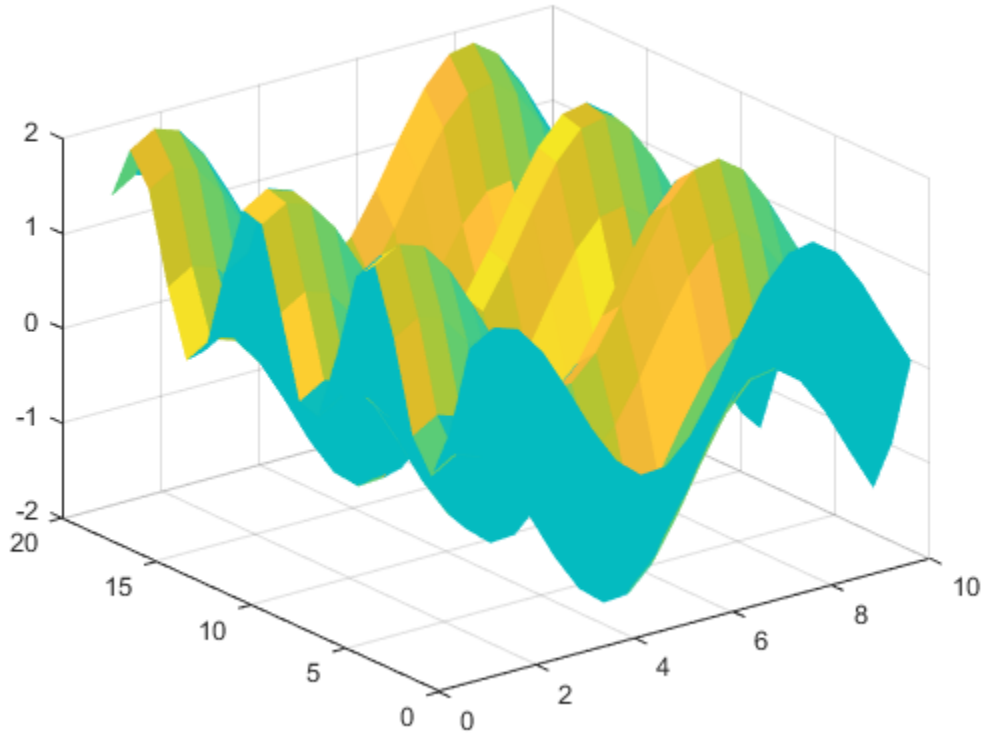
Plot the data using the source and reflectance vectors.

```
sl = surfl(X,Y,Z,s,k);
```



Use `sl` to access and modify properties of the surface object after it is created. For example, hide the edges by setting the `EdgeColor` property.

```
sl.EdgeColor = 'none';
```



Input Arguments

X — *x*-coordinates
matrix | vector

x-coordinates, specified as a matrix the same size as `Z`, or as a vector with length `n`, where `[m,n] = size(Z)`. If you do not specify values for `X` and `Y`, `surfl` uses the vectors `(1:n)` and `(1:m)`.

You can use the `meshgrid` function to create `X` and `Y` matrices.

The `XData` property of the `Surface` object stores the *x*-coordinates.

Example: `X = 1:10`

Example: `X = [1 2 3; 1 2 3; 1 2 3]`

Example: `[X,Y] = meshgrid(-5:0.5:5)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Y — *y*-coordinates
matrix | vector

y-coordinates, specified as a matrix the same size as `Z` or as a vector with length `m`, where `[m,n] = size(Z)`. If you do not specify values for `X` and `Y`, `surfl` uses the vectors `(1:n)` and `(1:m)`.

You can use the `meshgrid` function to create the `X` and `Y` matrices.

The `YData` property of the surface object stores the *y*-coordinates.

Example: `Y = 1:10`

Example: `Y = [1 1 1; 2 2 2; 3 3 3]`

Example: `[X,Y] = meshgrid(-5:0.5:5)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Z — *z*-coordinates
matrix

z-coordinates, specified as a matrix. `Z` must have at least two rows and two columns.

The `ZData` property of the surface object stores the *z*-coordinates.

Example: `Z = [1 2 3; 4 5 6]`

Example: `Z = sin(x) + cos(y)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

s — Direction to light source
two-element vector | three-element vector

Direction from the surface to the light source, specified as a two- or three-element vector. The vector has the form `[sx sy sz]` or `[azimuth elevation]`. The default direction is 45° counterclockwise from the current view direction.

k — Reflectance constant
`[.55 .6 .4 10]` (default) | four-element vector

Reflectance constant, specified as a four-element vector. The vector defines the relative contributions of ambient light, diffused reflection, specular reflection, and the specular shine coefficient using the form `[ka kd ks shine]`. By default, `k` is `[.55 .6 .4 10]`.

ax — Axes to plot in
axes object

Axes to plot in, specified as an `axes` object. If you do not specify the axes, then `surfl` plots into the current axes.

Tips

- The ordering of points in the `X`, `Y`, and `Z` matrices defines the inside and outside of parametric surfaces. To have the opposite side of the surface reflect the light source, use `surfl(X',Y',Z')`.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

Functions
`colormap` | `imagesc` | `light` | `meshgrid` | `pcolor` | `shading` | `surf` | `view`

Properties
Chart Surface | Light Properties

Topics

“Representing Data as a Surface”
“How Surface Plot Data Relates to a Colormap”

Introduced before R2006a

surfnorm

Surface normals

Syntax

```
surfnorm(X,Y,Z)
surfnorm(Z)

surfnorm(ax, ___ )
surfnorm( ___ ,Name,Value)
[Nx,Ny,Nz] = surfnorm( ___ )
```

Description

`surfnorm(X,Y,Z)` creates a three-dimensional surface plot and displays its surface normals. A surface normal is the imaginary line perpendicular to a flat surface, or perpendicular to the tangent plane at a point on a non-flat surface.

The function plots the values in matrix `Z` as heights above a grid in the *x*-*y* plane defined by `X` and `Y`. The color of the surface varies according to the heights specified by `Z`. The matrices `X`, `Y`, and `Z` must be the same size.

`surfnorm(Z)` creates a surface with normals and uses the column and row indices of the elements in `Z` as the *x* and *y*-coordinates, respectively.

`surfnorm(ax, ___ )` plots into the axes specified by `ax` instead of the current axes. Specify the axes as the first input argument.

`surfnorm( ___ ,Name,Value)` specifies surface properties using one or more name-value pair arguments. For example, `'FaceAlpha',0.5` creates a semitransparent surface.

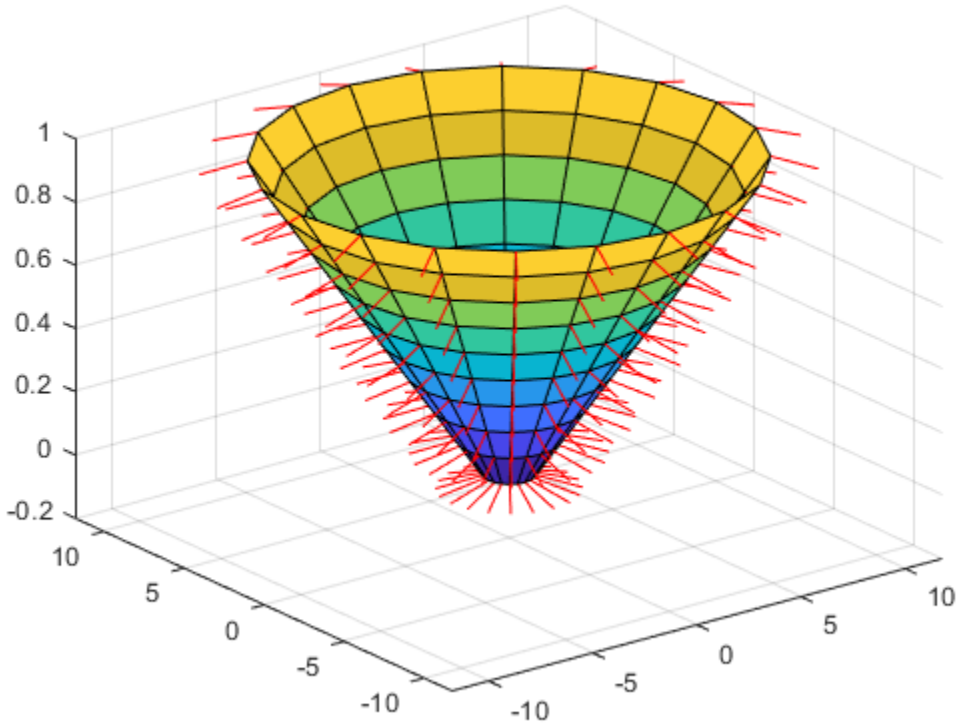
`[Nx,Ny,Nz] = surfnorm( ___ )` returns the *x*, *y*, and *z* components of the three-dimensional surface normals for the surface without plotting anything.

Examples

Create Surface Plot With Surface Normals

Create a cone. Then plot the data as a surface and display the surface normals. The surface uses Z for both height and color.

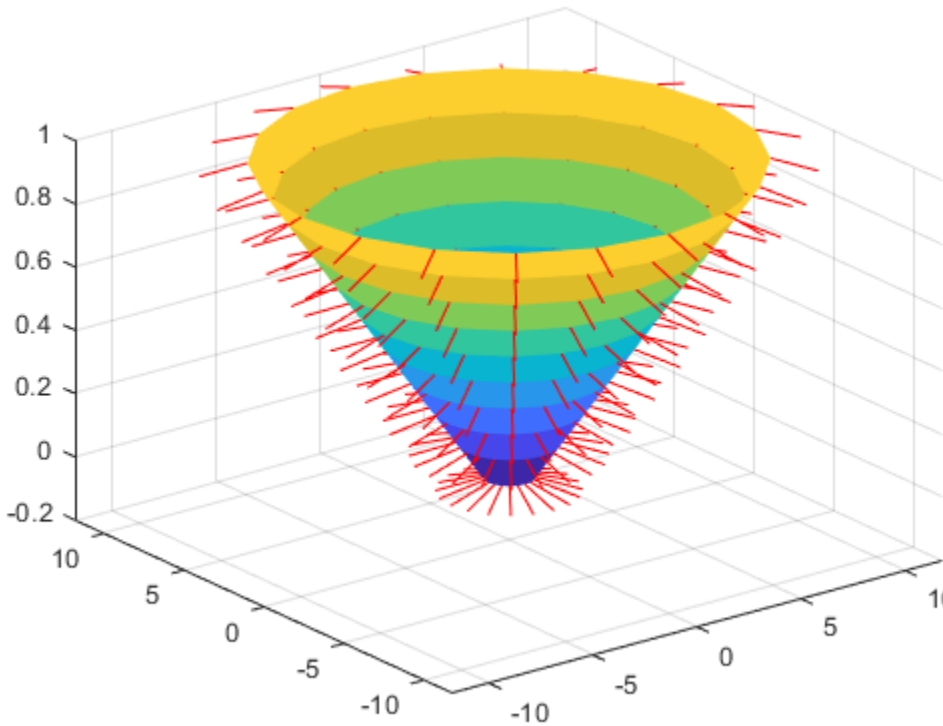
```
[X,Y,Z] = cylinder(1:10);
surfnorm(X,Y,Z)
```



Modify Surface Plot Appearance

Create a surface with no edges by specifying the `EdgeColor` name-value pair with `'none'` as the value.

```
[X,Y,Z] = cylinder(1:10);
surfnorm(X,Y,Z,'EdgeColor','none')
```

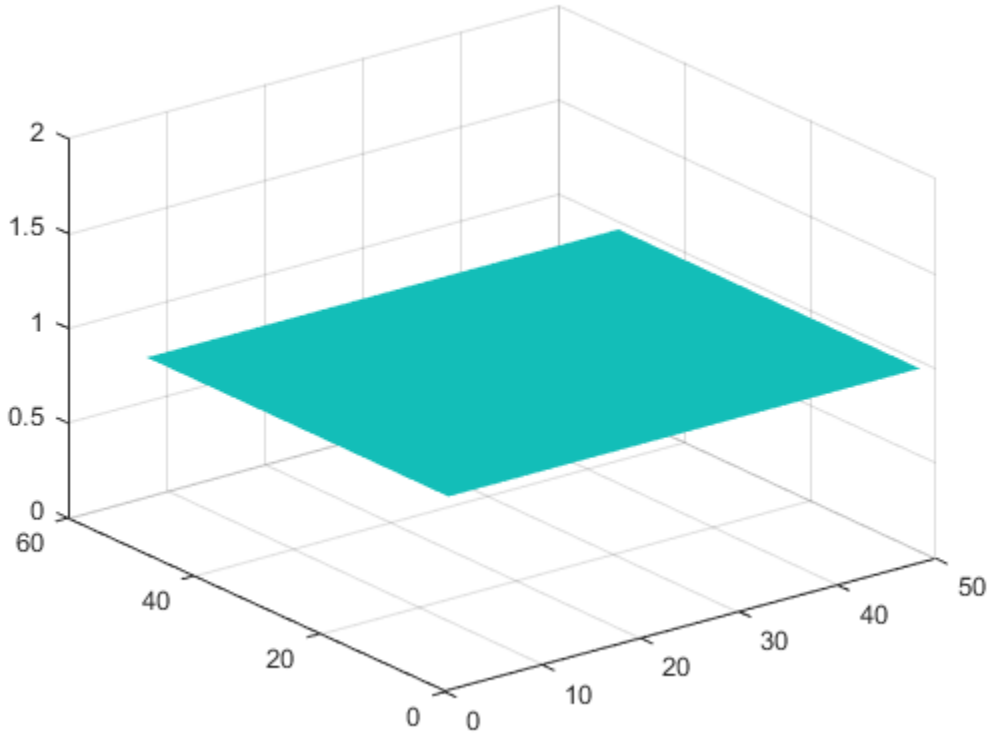


Light a Surface Using Surface Normals

Use the surface normals of a curved surface to light a flat surface.

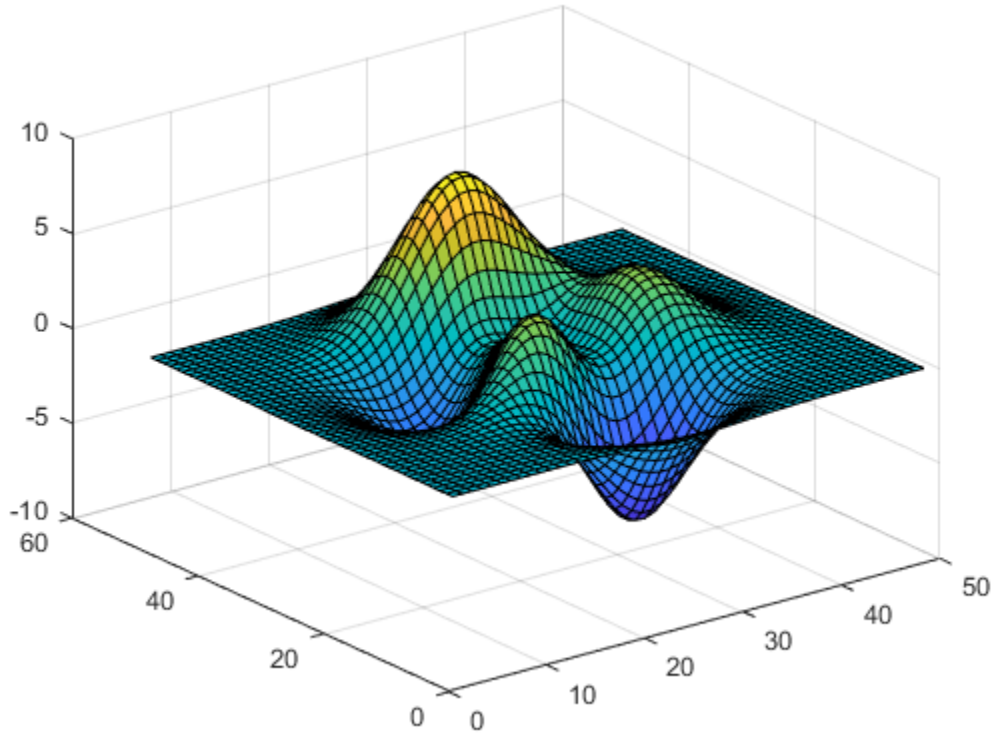
First, display a flat surface.

```
surf(ones(49),'EdgeColor','none');
```



Display the curved surface to use as a lighting source.

```
surf(peaks);
```



Now, draw the flat surface again, this time with lighting from the curved surface. To do this, first calculate the surface normals of the curved surface.

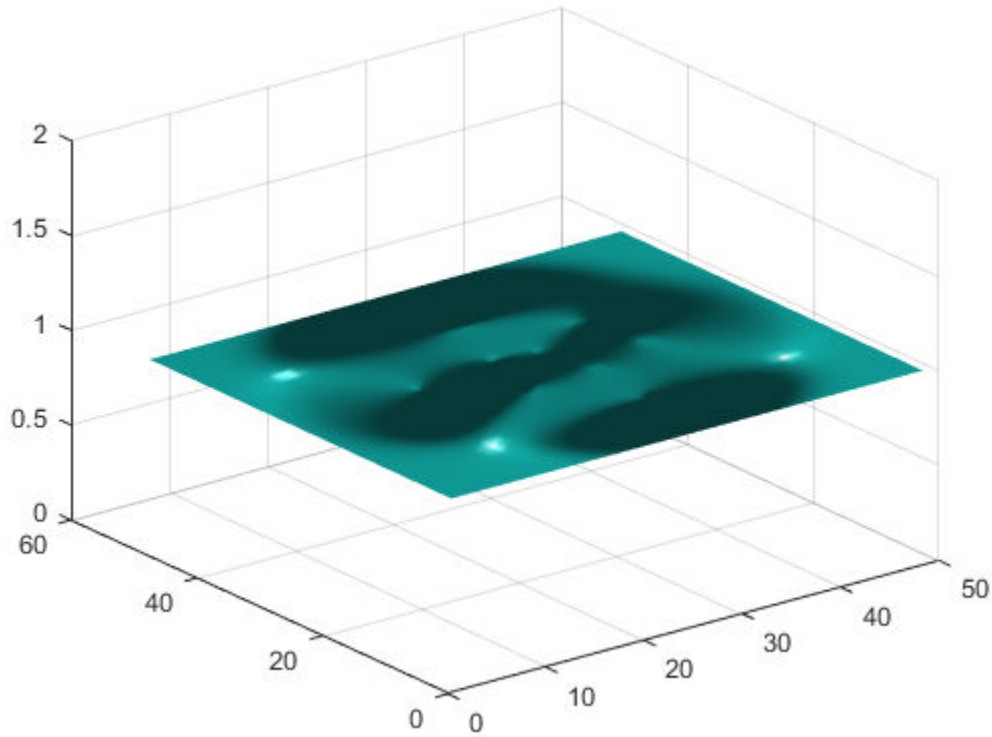
```
[nx, ny, nz] = surfnorm(peaks);
```

Combine the *x*, *y*, and *z* surface normal components into a single 49-by-49-by-3 array.

```
b = reshape([nx ny nz], 49,49,3);
```

Create a flat surface again, this time supplying this array as a value for the `VertexNormals` property. MATLAB® uses the `VertexNormals` property to calculate the surface lighting. Set the lighting algorithm to `gouraud` and add a light using `camlight`.

```
surf(ones(49),'VertexNormals',b,'EdgeColor','none');
lighting gouraud
camlight
```



Input Arguments

X — *x*-coordinates
matrix

x-coordinates, specified as a matrix the same size as Y and Z.

You can use the `meshgrid` function to create the X and Y matrices.

The `XData` property of the `Surface` object stores the *x*-coordinates.

Example: `X = [1 2 3; 1 2 3; 1 2 3]`

Example: `[X,Y] = meshgrid(-5:0.5:5)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Y — *y*-coordinates
matrix

y-coordinates, specified as a matrix the same size as `X` and `Z`.

You can use the `meshgrid` function to create the `X` and `Y` matrices.

The `YData` property of the surface object stores the *y*-coordinates.

Example: `Y = [1 1 1; 2 2 2; 3 3 3]`

Example: `[X,Y] = meshgrid(-5:0.5:5)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Z — *z*-coordinates
matrix

z-coordinates, specified as a matrix. `Z` must have at least three rows and three columns. `Z` also sets the surface colors.

The `ZData` property of the surface object stores the *z*-coordinates.

Example: `Z = [1 2 3; 4 5 6; 7 8 9]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

ax — Axes to plot in
axes object

Axes to plot in, specified as an `axes` object. If you do not specify the axes, then `surfnorm` plots into the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Note The properties listed here are only a subset. For a full list, see Surface Properties.

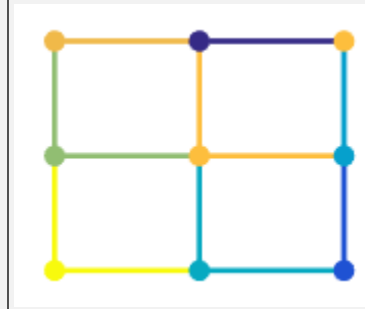
Example: `surfnorm(X,Y,Z,'FaceAlpha',0.5,'EdgeColor','none')` creates a semitransparent surface with no edges drawn.

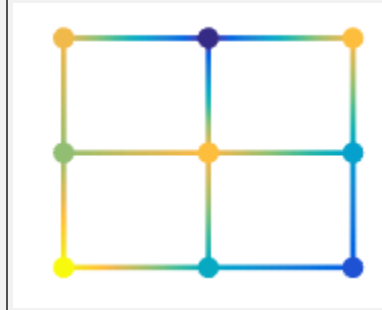
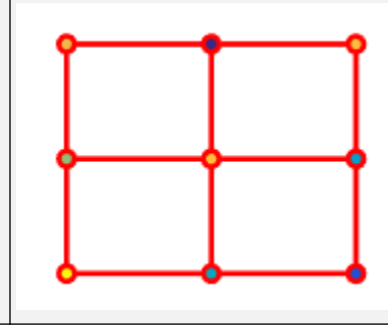
`EdgeColor` — Edge line color

`[0 0 0]` (default) | `'none'` | `'flat'` | `'interp'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Edge line color, specified as one of the values listed here. The default color of `[0 0 0]` corresponds to black edges.

Value	Description
`'none'`	Do not draw the edges.

Value	Description
`'flat'`	Use a different color for each edge based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color value at the first vertex of each face (in the positive *x* and *y* directions) determines the color for the adjacent edges. You cannot use this value when the `EdgeAlpha` property is set to `'interp'`. 

Value	Description
`'interp'`	Use interpolated coloring for each edge based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color varies across each edge by linearly interpolating the color values at the vertices. You cannot use this value when the `EdgeAlpha` property is set to `'flat'`. 
RGB triplet, hexadecimal color code, or color name	Use the specified color for all the edges. This option does not use the color values in the `CData` property. 

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`LineStyle` — Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

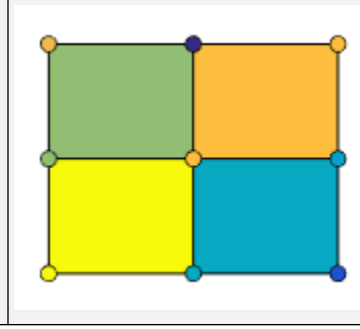
Line style, specified as one of the options listed in this table.

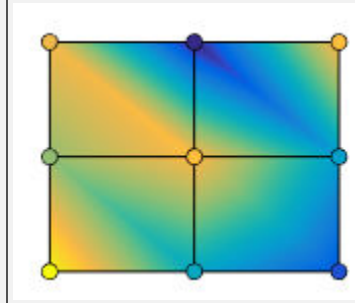
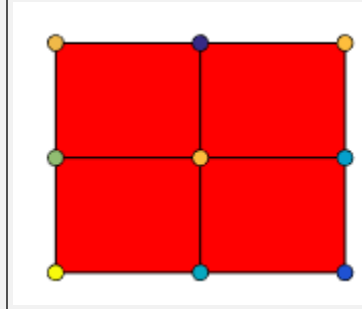
Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`FaceColor` — Face color

`'flat'` (default) | `'interp'` | `'none'` | `'texturemap'` | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Face color, specified as one of the values in this table.

Value	Description
`'flat'`	Use a different color for each face based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color value at the first vertex of each face (in the positive *x* and *y* directions) determines the color for the entire face. You cannot use this value when the `FaceAlpha` property is set to `'interp'`. 

Value	Description
`'interp'`	Use interpolated coloring for each face based on the values in the `CData` property. First you must specify the `CData` property as a matrix the same size as `ZData`. The color varies across each face by interpolating the color values at the vertices. You cannot use this value when the `FaceAlpha` property is set to `'flat'`. 
RGB triplet, hexadecimal color code, or color name	Use the specified color for all the faces. This option does not use the color values in the `CData` property. 
`'texturemap'`	Transform the color data in `CData` so that it conforms to the surface.
`'none'`	Do not draw the faces.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

FaceAlpha — Face transparency
1 (default) | scalar in range `[0,1]` | `'flat'` | `'interp'` | `'texturemap'`

Face transparency, specified as one of these values:

- Scalar in range `[0,1]` — Use uniform transparency across all the faces. A value of `1` is fully opaque and `0` is completely transparent. Values between `0` and `1` are semitransparent. This option does not use the transparency values in the `AlphaData` property.
- `'flat'` — Use a different transparency for each face based on the values in the `AlphaData` property. The transparency value at the first vertex determines the transparency for the entire face. First you must specify the `AlphaData` property as a matrix the same size as the `ZData` property. The `FaceColor` property also must be set to `'flat'`.
- `'interp'` — Use interpolated transparency for each face based on the values in `AlphaData` property. The transparency varies across each face by interpolating the values at the vertices. First you must specify the `AlphaData` property as a matrix the same size as the `ZData` property. The `FaceColor` property also must be set to `'interp'`.
- `'texturemap'` — Transform the data in `AlphaData` so that it conforms to the surface.

FaceLighting — Effect of light objects on faces
`'flat'` (default) | `'gouraud'` | `'none'`

Effect of light objects on faces, specified as one of these values:

- `'flat'` — Apply light uniformly across each face. Use this value to view faceted objects.
- `'gouraud'` — Vary the light across the faces. Calculate the light at the vertices and then linearly interpolate the light across the faces. Use this value to view curved surfaces.
- `'none'` — Do not apply light from light objects to the faces.

To add a light object to the axes, use the `light` function.

Note The `'phong'` value has been removed. Use `'gouraud'` instead.

Output Arguments

Nx — Surface normal *x*-component
matrix

Surface normal *x*-component, returned as a matrix. For more information about how the surface normals are computed, see “Algorithms” on page 1-15002.

Ny — Surface normal *y*-component
matrix

Surface normal *y*-component, returned as a matrix. For more information about how the surface normals are computed, see “Algorithms” on page 1-15002.

Nz — Surface normal *z*-component
matrix

Surface normal *z*-component, returned as a matrix. For more information about how the surface normals are computed, see “Algorithms” on page 1-15002.

Tips

- To reverse the direction of the normals, call `surfnorm` with transposed arguments:

  ```
  surfnorm(X',Y',Z')
  ```

- To show the direction of the normals on a surface, use the `surfnorm` function to calculate the surface normals and then the `quiver3` function to display them.

  ```
  [Nx,Ny,Nz] = surfnorm(X,Y,Z);
  quiver3(X,Y,Z,Nx,Ny,Nz)
  ```

- The surface normals represent conditions at vertices and are not normalized. Normals for surface elements that face away from the viewer are not visible.

Algorithms

`surfnorm` uses bicubic interpolation in the *x*, *y*, and *z* directions to calculate the surface normals of the data. To allow for interpolation at the boundaries, the function uses quadratic extrapolation to expand the data. After performing the bicubic fit of the data, diagonal vectors are computed and crossed to form the normal at each vertex.

See Also

Functions
`isonormals` | `meshgrid` | `quiver3` | `surf` | `surfl`

Properties
Chart Surface

Topics

"Representing Data as a Surface"
"How Surface Plot Data Relates to a Colormap"

Introduced before R2006a

svd

Singular value decomposition

Syntax

```
s = svd(A)
```

```
[U,S,V] = svd(A)
[U,S,V] = svd(A,'econ')
[U,S,V] = svd(A,0)
```

Description

`s = svd(A)` returns the singular values of matrix `A` in descending order.

`[U,S,V] = svd(A)` performs a singular value decomposition of matrix `A`, such that `A = U*S*V'`.

`[U,S,V] = svd(A,'econ')` produces an economy-size decomposition of `m`-by-`n` matrix `A`:

- `m > n` — Only the first `n` columns of `U` are computed, and `S` is `n`-by-`n`.
- `m = n` — `svd(A,'econ')` is equivalent to `svd(A)`.
- `m < n` — Only the first `m` columns of `V` are computed, and `S` is `m`-by-`m`.

The economy-size decomposition removes extra rows or columns of zeros from the diagonal matrix of singular values, `S`, along with the columns in either `U` or `V` that multiply those zeros in the expression `A = U*S*V'`. Removing these zeros and columns can improve execution time and reduce storage requirements without compromising the accuracy of the decomposition.

`[U,S,V] = svd(A,0)` produces a different economy-size decomposition of `m`-by-`n` matrix `A`:

- `m > n` — `svd(A,0)` is equivalent to `svd(A,'econ')`.

- `m <= n` — `svd(A,0)` is equivalent to `svd(A)`.

Examples

Singular Values of Matrix

Compute the singular values of a full rank matrix.

```
A = [1 0 1; -1 -2 0; 0 1 -1]
```

```
A = 3×3

     1     0     1
    -1    -2     0
     0     1    -1
```

```
s = svd(A)
```

```
s = 3×1

    2.4605
    1.6996
    0.2391
```

Singular Value Decomposition

Find the singular value decomposition of a rectangular matrix `A`.

```
A = [1 2; 3 4; 5 6; 7 8]
```

```
A = 4×2

     1     2
     3     4
     5     6
     7     8
```

```
[U,S,V] = svd(A)
```

```
U = 4×4

   -0.1525   -0.8226   -0.3945   -0.3800
   -0.3499   -0.4214    0.2428    0.8007
   -0.5474   -0.0201    0.6979   -0.4614
   -0.7448    0.3812   -0.5462    0.0407


S = 4×2

   14.2691         0
         0    0.6268
         0         0
         0         0


V = 2×2

   -0.6414    0.7672
   -0.7672   -0.6414
```

Confirm the relation `A = U*S*V'`, within machine precision.

```
U*S*V'
```

```
ans = 4×2

    1.0000    2.0000
    3.0000    4.0000
    5.0000    6.0000
    7.0000    8.0000
```

Economy-Size Decomposition

Calculate the full and economy-size decompositions of a rectangular matrix.

```
A = [1 2; 3 4; 5 6; 7 8]
```

```
A = 4×2
```

```
     1     2
     3     4
     5     6
     7     8
```

```
[U,S,V] = svd(A)
```

```
U = 4×4

   -0.1525   -0.8226   -0.3945   -0.3800
   -0.3499   -0.4214    0.2428    0.8007
   -0.5474   -0.0201    0.6979   -0.4614
   -0.7448    0.3812   -0.5462    0.0407


S = 4×2

   14.2691         0
         0    0.6268
         0         0
         0         0


V = 2×2

   -0.6414    0.7672
   -0.7672   -0.6414
```

```
[U,S,V] = svd(A,'econ')
```

```
U = 4×2

   -0.1525   -0.8226
   -0.3499   -0.4214
   -0.5474   -0.0201
   -0.7448    0.3812


S = 2×2

   14.2691         0
         0    0.6268
```

```
V = 2×2

   -0.6414    0.7672
   -0.7672   -0.6414
```

Since `A` is 4-by-2, `svd(A,'econ')` returns fewer columns in `U` and fewer rows in `S` compared to a full decomposition. Extra rows of zeros in `S` are excluded, along with the corresponding columns in `U` that would multiply with those zeros in the expression `A = U*S*V'`.

Rank, Column Space, and Null Space of Matrix

Use the results of the singular value decomposition to determine the rank, column space, and null space of a matrix.

```
A = [2 0 2; 0 1 0; 0 0 0]
```

```
A = 3×3

     2     0     2
     0     1     0
     0     0     0
```

```
[U,S,V] = svd(A)
```

```
U = 3×3

     1     0     0
     0     1     0
     0     0     1


S = 3×3

    2.8284         0         0
         0    1.0000         0
         0         0         0


V = 3×3
```

```
    0.7071         0   -0.7071
         0    1.0000         0
    0.7071         0    0.7071
```

Calculate the rank using the number of nonzero singular values.

```
s = diag(S);
rank_A = nnz(s)
```

```
rank_A = 2
```

Compute an orthonormal basis for the column space of `A` using the columns of `U` that correspond to nonzero singular values.

```
column_basis = U(:,logical(s))
```

```
column_basis = 3×2

     1     0
     0     1
     0     0
```

Compute an orthonormal basis for the null space of `A` using the columns of `V` that correspond to singular values equal to zero.

```
null_basis = V(:,~s)
```

```
null_basis = 3×1

   -0.7071
         0
    0.7071
```

The functions `rank`, `orth`, and `null` provide convenient ways to calculate these quantities.

Input Arguments

`A` — Input matrix
matrix

Input matrix. A can be either square or rectangular in size.

Data Types: single | double
Complex Number Support: Yes

Output Arguments

s — Singular values
column vector

Singular values, returned as a column vector. The singular values are nonnegative real numbers listed in decreasing order.

U — Left singular vectors
matrix

Left singular vectors, returned as the columns of a matrix.

- For an m-by-n matrix A with m > n, the economy-sized decompositions svd(A,'econ') and svd(A,0) compute only the first n columns of U. In this case, the columns of U are orthogonal and U is an m-by-n matrix that satisfies $U^H U = I_n$.
- For full decompositions, svd(A) returns U as an m-by-m unitary matrix satisfying $UU^H = U^H U = I_m$. The columns of U that correspond to nonzero singular values form a set of orthonormal basis vectors for the range of A.

Different machines and releases of MATLAB can produce different singular vectors that are still numerically accurate. Corresponding columns in U and V can flip their signs, since this does not affect the value of the expression A = U*S*V'.

S — Singular values
diagonal matrix

Singular values, returned as a diagonal matrix. The diagonal elements of S are nonnegative singular values in decreasing order. The size of S is as follows:

- For an m-by-n matrix A, the economy-sized decomposition svd(A,'econ') returns S as a square matrix of order min([m,n]).
- For full decompositions, svd(A) returns S with the same size as A.
- If m > n, then svd(A,0) returns S as a square matrix of order min([m,n]).

- If `m < n`, then `svd(A,0)` returns `S` with the same size as `A`.

`V` — Right singular vectors
matrix

Right singular vectors, returned as the columns of a matrix.

- For an `m`-by-`n` matrix `A` with `m < n`, the economy decomposition `svd(A,'econ')` computes only the first `m` columns of `V`. In this case, the columns of `V` are orthogonal and `V` is an `n`-by-`m` matrix that satisfies $V^H V = I_m$.
- For full decompositions, `svd(A)` returns `V` as an `n`-by-`n` unitary matrix satisfying $VV^H = V^H V = I_n$. The columns of `V` that do *not* correspond to nonzero singular values form a set of orthonormal basis vectors for the null space of `A`.

Different machines and releases of MATLAB can produce different singular vectors that are still numerically accurate. Corresponding columns in `U` and `V` can flip their signs, since this does not affect the value of the expression `A = U*S*V'`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- The three-output syntax `[U,S,V] = svd(X)` is not supported. For three outputs, you must specify `svd(X,'econ')` or `svd(X,0)`.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation uses a different `SVD` implementation than MATLAB uses. Because the singular value decomposition is not unique, left and right singular vectors might differ from those computed by MATLAB.
- When the input matrix contains a nonfinite value, the generated code does not issue an error. Instead, the output contains `NaN` values.
- Code generation does not support sparse matrix inputs for this function.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Code generation uses a different `SVD` implementation than MATLAB uses. Because the singular value decomposition is not unique, left and right singular vectors might differ from those computed by MATLAB.
- When the input matrix contains a nonfinite value, the generated code does not issue an error. Instead, the output contains `NaN` values.
- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`gsvd` | `null` | `orth` | `rank` | `svds`

Topics

"Singular Values"

Introduced before R2006a

svds

Subset of singular values and vectors

Syntax

```
s = svds(A)
s = svds(A,k)
s = svds(A,k,sigma)
s = svds(A,k,sigma,Name,Value)
s = svds(A,k,sigma,opts)

s = svds(Afun,n, ___ )

[U,S,V] = svds( ___ )
[U,S,V,flag] = svds( ___ )
```

Description

`s = svds(A)` returns a vector of the six largest singular values of matrix `A`. This is useful when computing all of the singular values with `svd` is computationally expensive, such as with large sparse matrices.

`s = svds(A,k)` returns the `k` largest singular values.

`s = svds(A,k,sigma)` returns `k` singular values based on the value of `sigma`. For example, `svds(A,k,'smallest')` returns the `k` smallest singular values.

`s = svds(A,k,sigma,Name,Value)` specifies additional options with one or more name-value pair arguments. For example, `svds(A,k,sigma,'Tolerance',1e-3)` adjusts the convergence tolerance for the algorithm.

`s = svds(A,k,sigma,opts)` specifies options using a structure.

`s = svds(Afun,n, ___ )` specifies a function handle `Afun` instead of a matrix. The second input `n` gives the size of matrix `A` used in `Afun`. You can optionally specify `k`, `sigma`, `opts`, or name-value pairs as additional input arguments.

`[U,S,V] = svds( ___ )` returns the left singular vectors `U`, diagonal matrix `S` of singular values, and right singular vectors `V`. You can use any of the input argument combinations in previous syntaxes.

`[U,S,V,flag] = svds( ___ )` also returns a convergence flag. If `flag` is `0`, then all the singular values converged.

Examples

Largest Singular Values

The matrix `A = delsq(numgrid('C',15))` is a symmetric positive definite matrix with singular values reasonably well-distributed in the interval (0 8). Compute the six largest singular values.

```
A = delsq(numgrid('C',15));
s = svds(A)
```

```
s = 6×1

    7.8666
    7.7324
    7.6531
    7.5213
    7.4480
    7.3517
```

Specify a second input to compute a specific number of the largest singular values.

```
s = svds(A,3)
```

```
s = 3×1

    7.8666
    7.7324
    7.6531
```

Smallest Singular Values

The matrix `A = delsq(numgrid('C',15))` is a symmetric positive definite matrix with singular values reasonably well-distributed in the interval (0 8). Compute the five smallest singular values.

```
A = delsq(numgrid('C',15));
s = svds(A,5,'smallest')
```

```
s = 5×1

    0.5520
    0.4787
    0.3469
    0.2676
    0.1334
```

Smallest Nonzero Singular Values

Create a sparse 100-by-100 Neumann matrix.

```
C = gallery('neumann',100);
```

Compute the ten smallest singular values.

```
ss = svds(C,10,'smallest')
```

```
ss = 10×1

    0.9828
    0.9049
    0.5625
    0.5625
    0.4541
    0.4506
    0.2256
    0.1139
    0.1139
         0
```

Compute the 10 smallest nonzero singular values. Since the matrix has a singular value that is equal to zero, the `'smallestnz'` option omits it.

```
snz = svds(C,10,'smallestnz')
```

```
snz = 10×1

    0.9828
    0.9828
    0.9049
    0.5625
    0.5625
    0.4541
    0.4506
    0.2256
    0.1139
    0.1139
```

Largest Singular Values Using Function Handle

Create two matrices representing the upper-right and lower-left nonzero blocks in a sparse matrix.

```
n = 500;
B = rand(500);
C = rand(500);
```

Save `Afun` in your current directory so that it is available for use with `svds`.

```
function y = Afun(x,tflag,B,C,n)
if strcmp(tflag,'notransp')
    y = [B*x(n+1:end); C*x(1:n)];
else
    y = [C'*x(n+1:end); B'*x(1:n)];
end
```

The function `Afun` uses `B` and `C` to compute either `A*x` or `A'*x` (depending on the specified flag) without actually forming the entire sparse matrix `A = [zeros(n) B; C`

`zeros(n)]`. This exploits the sparsity pattern of the matrix to save memory in the computation of `A*x` and `A'*x`.

Use `Afun` to calculate the 10 largest singular values of `A`. Pass `B`, `C`, and `n` as additional inputs to `Afun`.

```
s = svds(@(x,tflag) Afun(x,tflag,B,C,n),[1000 1000],10)
```

```
s =

  250.3248
  249.9914
   12.7627
   12.7232
   12.6988
   12.6608
   12.6166
   12.5643
   12.5419
   12.4512
```

Directly compute the 10 largest singular values of `A` to compare the results.

```
A = [zeros(n) B; C zeros(n)];
s = svds(A,10)
```

```
s =

  250.3248
  249.9914
   12.7627
   12.7232
   12.6988
   12.6608
   12.6166
   12.5643
   12.5419
   12.4512
```

Singular Value Decomposition of Sparse Matrix

`west0479` is a real-valued 479-by-479 sparse matrix. The matrix has a few large singular values, and many small singular values.

Load `west0479` and store it as `A`.

```
load west0479
A = west0479;
```

Compute the singular value decomposition of `A`, returning the six largest singular values and the corresponding singular vectors. Specify a fourth output argument to check convergence of the singular values.

```
[U,S,V,cflag] = svds(A);
cflag
```

```
cflag = 0
```

`cflag` indicates that all of the singular values converged. The singular values are on the diagonal of the output matrix `S`.

```
s = diag(S)
```

```
s = 6×1
10^5 ×

    3.1895
    3.1725
    3.1695
    3.1685
    3.1669
    0.3038
```

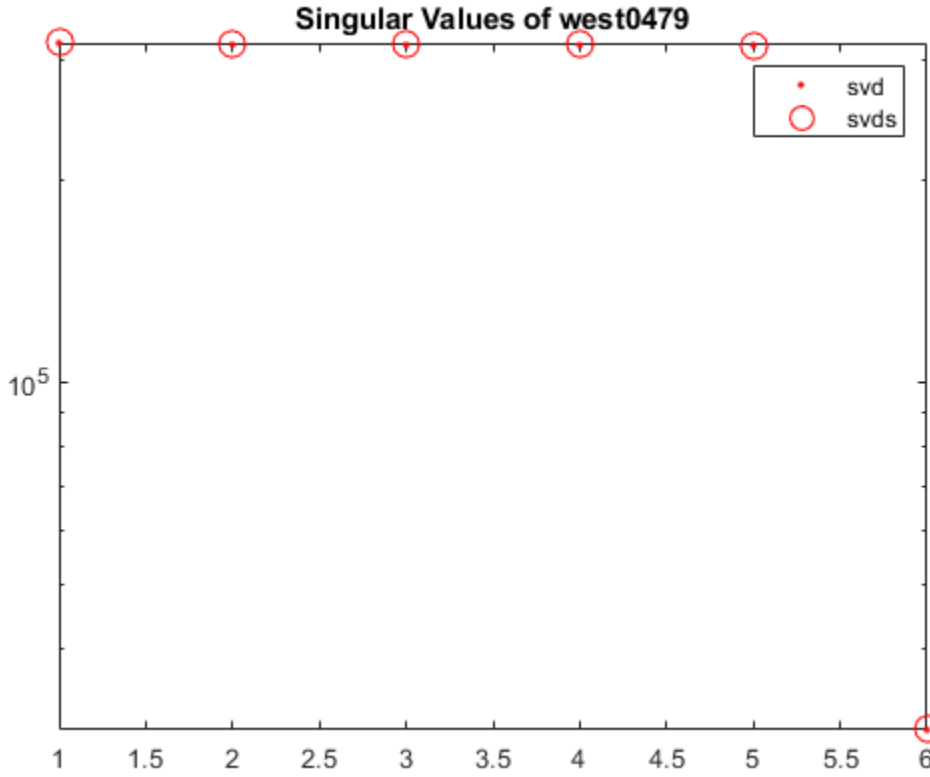
Check the results by computing the full singular value decomposition of `A`. Convert `A` to a full matrix and use `svd`.

```
[U1,S1,V1] = svd(full(A));
```

Plot the six largest singular values of `A` computed by `svd` and `svds` using a logarithmic scale.

```
s2 = diag(S1);
semilogy(s2(1:6),'r.')
```

```
hold on
semilogy(s,'ro','MarkerSize',10)
title('Singular Values of west0479')
legend('svd','svds')
```



Fix Convergence Problem

Create a sparse diagonal matrix and calculate the six largest singular values.

```
A = diag(sparse([1e4*ones(1, 8) 1e4:-1:1]));
s = svds(A)
```

```
Warning: Only 2 of the 6 requested singular values converged. Singular values that did

s = 6×1
10^4 ×

    1.0000
    0.9999
       NaN
       NaN
       NaN
       NaN
```

The `svds` algorithm produces a warning since the maximum number of iterations were performed but the tolerance could not be met.

The most effective way to address convergence problems is to increase the maximum size of the Krylov subspace used in the calculation by using a larger value for `'SubspaceDimension'`. Do this by passing in the name-value pair `'SubspaceDimension'` with a value of `60`.

```
s = svds(A,6,'largest','SubspaceDimension',60)
```

```
s = 6×1
10^4 ×

    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
```

Use QR Decomposition to Compute SVD of Nearly Singular Matrix

Compute the 10 smallest singular values of a nearly singular matrix.

```
rng default
format shortg
B = spdiags([repelem([1; 1e-7], [198, 2]) ones(200, 1)], [0 1], 200, 200);
s1 = svds(B,10,'smallest')
```

```
Warning: Large residual norm detected. This is likely due to bad condition of the input

s1 = 10×1

       7.0945
       7.0945
       7.0945
       7.0945
       7.0945
       7.0945
       7.0945
       7.0945
      0.25927
   7.0888e-16
```

The warning indicates that `svds` fails to calculate the proper singular values. The failure with `svds` is because of the gap between the smallest and second smallest singular values. `svds(...,'smallest')` needs to invert `B`, which leads to large numerical error.

For comparison, compute the exact singular values using `svd`.

```
s = svd(full(B));
s = s(end-9:end)
```

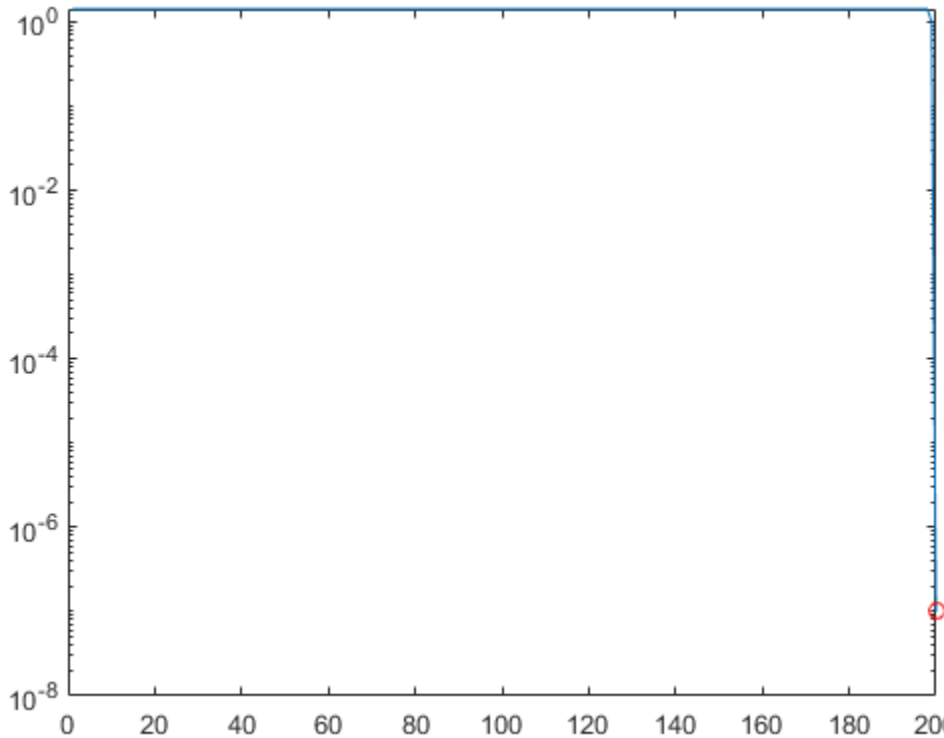
```
s = 10×1

      0.14196
      0.12621
      0.11045
     0.094686
     0.078914
     0.063137
     0.047356
     0.031572
     0.015787
   7.0888e-16
```

In order to reproduce this calculation with `svds`, do a QR decomposition of `B`. The singular values of the triangular matrix `R` are the same as for `B`.

```
[Q,R,p] = qr(B,0);
```

Plot the norm of each row of `R`.

```
rownormR = sqrt(diag(R*R'));
semilogy(rownormR)
hold on;
semilogy(size(R, 1), rownormR(end), 'ro')
```



The last entry in R is nearly zero, which causes instability in the solution.

Prevent this entry from corrupting the good parts of the solution by setting the last row of R to be exactly zero.

```
R(end,:) = 0;
```

Use `svds` to find the 10 smallest singular values of R. The results are comparable to those obtained by `svd`.

```
sr = svds(R,10,'smallest')
```

```
sr = 10×1

      0.14196
      0.12621
      0.11045
     0.094686
     0.078914
     0.063137
     0.047356
     0.031572
     0.015787
            0
```

To compute the singular vectors of B using this method, transform the left and right singular vectors using Q and the permutation vector p.

```
[U,S,V] = svds(R,20,'s');
U = Q*U;
V(p,:) = V;
```

Input Arguments

A — Input matrix
matrix

Input matrix. A is typically, but not always, a large and sparse matrix.

Data Types: double
Complex Number Support: Yes

k — Number of singular values to compute
scalar

Number of singular values to compute, specified as a positive scalar integer. svds returns fewer singular values than requested if either of these conditions are met:

- k is larger than min(size(A))
- sigma = 'smallestnz' and k is larger than the number of nonzero singular values of A

If k is too large, then svds replaces it with the maximum valid value of k.

Example: svds(A,2) returns the two largest singular values of A.

sigma — Type of singular values

'largest' (default) | 'smallest' | 'smallestnz' | scalar

Type of singular values, specified as one of these values.

Option	Description
'largest' (default)	Largest singular values
'smallest'	Smallest singular values
'smallestnz'	Smallest nonzero singular values
scalar	Singular values closest to a scalar

Example: svds(A,k,'smallest') computes the k smallest singular values.

Example: svds(A,k,100) computes the k singular values closest to 100.

Data Types: double | char | string

opts — Options structure

structure

Options structure, specified as a structure containing one or more of the fields in this table.

Note Use of the options structure to specify options is not recommended. Use name-value pairs instead.

Option Field	Description	Name-Value Pair
tol	Convergence tolerance	'Tolerance'
maxit	Maximum number of iterations	'MaxIterations'
p	Maximum size of Krylov subspace	'SubspaceDimension'
u0	Left initial starting vector	'LeftStartVector'

Option Field	Description	Name-Value Pair
`v0`	Right initial starting vector	`'RightStartVector'`
`disp`	Diagnostic information display level	`'Display'`
`fail`	Treatment of nonconverged singular values in the output	`'FailureTreatment'`

Note `svds` ignores the option `p` when using a numeric scalar shift `sigma`.

Example: `opts.tol = 1e-6, opts.maxit = 500` creates a structure with values set for the fields `tol` and `maxit`.

Data Types: `struct`

`Afun` — Matrix function
function handle

Matrix function, specified as a function handle. The function `Afun` must satisfy these conditions:

- `Afun(x,'notransp')` accepts a vector `x` and returns the product `A*x`.
- `Afun(x,'transp')` accepts a vector `x` and returns the product `A'*x`.

Note Use function handles only in the case where `sigma = 'largest'` (which is the default).

Example: `svds(Afun,[1000 1200])`

`n` — Size of matrix used by `Afun`
two-element vector

Size of matrix `A` that is used by `Afun`, specified as a two-element size vector `[m n]`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `s = svds(A,k,sigma,'Tolerance',1e-10,'MaxIterations',100)` loosens the convergence tolerance and uses fewer iterations.

`Tolerance` — Convergence tolerance
`1e-14` (default) | nonnegative real scalar

Convergence tolerance, specified as the comma-separated pair consisting of `'Tolerance'` and a nonnegative real numeric scalar.

Example: `s = svds(A,k,sigma,'Tolerance',1e-3)`

`MaxIterations` — Maximum number of algorithm iterations
`300` (default) | positive integer

Maximum number of algorithm iterations, specified as the comma-separated pair consisting of `'MaxIterations'` and a positive integer.

Example: `s = svds(A,k,sigma,'MaxIterations',350)`

`SubspaceDimension` — Maximum size of Krylov subspace
`max(3*k,15)` (default) | nonnegative integer

Maximum size of Krylov subspace, specified as the comma-separated pair consisting of `'SubspaceDimension'` and a nonnegative integer. The `'SubspaceDimension'` value must be greater than or equal to `k + 2`, where `k` is the number of singular values.

For problems where `svds` fails to converge, increasing the value of `'SubspaceDimension'` can improve the convergence behavior.

This option is ignored for numeric values of `sigma`.

Example: `s = svds(A,k,sigma,'SubspaceDimension',25)`

`LeftStartVector` — Left initial starting vector
vector

Left initial starting vector, specified as the comma-separated pair consisting of `'LeftStartVector'` and a numeric vector.

You can specify either `'LeftStartVector'` or `'RightStartVector'`, but not both. If neither option is specified, then for an `m`-by-`n` matrix `A`, the default is:

- `m < n` — Left initial starting vector set to `randn(m,1)`
- `m >= n` — Right initial starting vector set to `randn(n,1)`

The primary reason to specify a different random starting vector is to control the random number stream used to generate the vector.

Note `svds` selects the starting vectors in a reproducible manner using a private random number stream. Changing the random number seed does *not* affect this use of `randn`.

Example: `s = svds(A,k,sigma,'LeftStartVector',randn(m,1))` uses a random starting vector that draws values from the global random number stream.

Data Types: `double`

`RightStartVector` — Right initial starting vector
vector

Right initial starting vector, specified as the comma-separated pair consisting of `'RightStartVector'` and a numeric vector.

You can specify either `'LeftStartVector'` or `'RightStartVector'`, but not both. If neither option is specified, then for an `m`-by-`n` matrix `A`, the default is:

- `m < n` — Left initial starting vector set to `randn(m,1)`
- `m >= n` — Right initial starting vector set to `randn(n,1)`

The primary reason to specify a different random starting vector is to control the random number stream used to generate the vector.

Note `svds` selects the starting vectors in a reproducible manner using a private random number stream. Changing the random number seed does *not* affect this use of `randn`.

Example: `s = svds(A,k,sigma,'RightStartVector',randn(n,1))` uses a random starting vector that draws values from the global random number stream.

Data Types: `double`

`FailureTreatment` — Treatment of nonconverged singular values
`'replacenan'` | `'keep'` | `'drop'`

Treatment of nonconverged singular values, specified as the comma-separated pair consisting of `'FailureTreatment'` and one of the options: `'replacenan'`, `'keep'`, or `'drop'`.

The value of `'FailureTreatment'` determines how nonconverged singular values are displayed in the output.

Option	Affect on Output
`'drop'`	Nonconverged singular values are removed from the output, which can result in `svds` returning fewer singular values than requested. This value is the default for numeric values of `sigma`.
`'replacenan'`	Nonconverged singular values are replaced with `NaN` values. This value is the default whenever `sigma` is not numeric.
`'keep'`	Nonconverged singular values are included in the output.

Example: `s = svds(A,k,sigma,'FailureTreatment','drop')` removes nonconverged singular values from the output.

Data Types: `char` | `string`

`Display` — Toggle for diagnostic information display
`false` (default) | `true` | `0` | `1`

Toggle for diagnostic information display, specified as `false`, `true`, `0`, or `1`. Values of `false` or `0` turn off the display, while values of `true` or `1` turn it on.

Output Arguments

`s` — Singular values
column vector

Singular values, returned as a column vector. The singular values are nonnegative real numbers listed in decreasing order.

`U` — Left singular vectors
matrix

Left singular vectors, returned as the columns of a matrix. If `A` is an `m`-by-`n` matrix and you request `k` singular values, then `U` is an `m`-by-`k` matrix with orthonormal columns.

Different machines, releases of MATLAB, or parameters (such as the starting vector and subspace dimension) can produce different singular vectors that are still numerically accurate. Corresponding columns in `U` and `V` can flip their signs, since this does not affect the value of the expression `A = U*S*V'`.

`S` — Singular values
diagonal matrix

Singular values, returned as a diagonal matrix. The diagonal elements of `S` are nonnegative singular values. If `A` is an `m`-by-`n` matrix and you request `k` singular values, then `S` is `k`-by-`k`.

`V` — Right singular vectors
matrix

Right singular vectors, returned as the columns of a matrix. If `A` is an `m`-by-`n` matrix and you request `k` singular values, then `V` is an `n`-by-`k` matrix with orthonormal columns.

Different machines, releases of MATLAB, or parameters (such as the starting vector and subspace dimension) can produce different singular vectors that are still numerically accurate. Corresponding columns in `U` and `V` can flip their signs, since this does not affect the value of the expression `A = U*S*V'`.

`flag` — Convergence flag
scalar

Convergence flag, returned as a scalar. A value of `0` indicates that all the singular values converged. Otherwise, not all the singular values converged.

Use of this convergence flag output suppresses warnings about failed convergence.

Tips

- `svds` generates the default starting vectors using a private random number stream to ensure reproducibility across runs. Setting the random number generator state using `rng` before calling `svds` does not affect the output.
- Using `svds` is not the most efficient way to find a few singular values of small, dense matrices. For such problems, using `svd(full(A))` might be quicker. For example,

finding three singular values in a 500-by-500 matrix is a relatively small problem that `svd` can handle easily.

- If `svds` fails to converge for a given matrix, increase the size of the Krylov subspace by increasing the value of `'SubspaceDimension'`. As secondary options, adjusting the maximum number of iterations (`'MaxIterations'`) and the convergence tolerance (`'Tolerance'`) also can help with convergence behavior.
- Increasing `k` can sometimes improve performance, especially when the matrix has repeated singular values.

Compatibility Considerations

Reproducibility change

Behavior changed in R2016a

- **Reproducibility**

  Calling `svds` multiple times in succession now produces the same result. To change this behavior:

  - In R2017a or earlier, set the `u0` or `v0` field of the options structure to a random vector.
  - In R2017b or later, prefer setting `'LeftStartVector'` or `'RightStartVector'` to a random vector.

References

[1] Baglama, J. and L. Reichel, “Augmented Implicitly Restarted Lanczos Bidiagonalization Methods.” *SIAM Journal on Scientific Computing*. Vol. 27, 2005, pp. 19–42.

[2] Larsen, R. M. “Lanczos Bidiagonalization with partial reorthogonalization.” *Dept. of Computer Science, Aarhus University*. DAIMI PB-357, 1998.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- If you provide the `sigma` parameter, the value must be `'largest'`.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- If you provide the `sigma` parameter, the value must be `'largest'`.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`eigs` | `svd`

Topics

“Singular Values”

Introduced before R2006a

swapbytes

Swap byte ordering

Syntax

```
Y = swapbytes(X)
```

Description

`Y = swapbytes(X)` swaps the byte ordering of each element in array X from little endian to big endian (or vice versa). All elements of the input array must be full, noncomplex, and numeric.

Examples

Swap Byte Ordering of Scalar

Swap the byte ordering for a scalar 32-bit value. Use hexadecimal representation to show the conversion from little endian to big endian.

```
X = uint32(hex2dec('4D3C2B1A'));
Y = dec2hex(swapbytes(X))
```

```
Y =
'1A2B3C4D'
```

Starting in R2019b, you can also write a hexadecimal value as a literal using the `0x` or `0X` prefix (for more information, see “Hexadecimal and Binary Values”). Specify the number X as a hexadecimal literal. Swap the byte ordering.

```
X = 0x4D3C2B1A;
Y = dec2hex(swapbytes(X))
```

```
Y =
'1A2B3C4D'
```

Swap Byte Ordering of Vector

Swap the byte ordering for each element of a 1-by-4 vector.

```
format short
X = uint16([0 1 128 65535])
```

```
X = 1x4 uint16 row vector

       0       1     128   65535
```

```
Y = swapbytes(X)
```

```
Y = 1x4 uint16 row vector

       0     256   32768   65535
```

Examine the output in hexadecimal notation to show the endian conversion.

```
format hex
X
```

```
X = 1x4 uint16 row vector

   0000   0001   0080   ffff
```

```
Y
```

```
Y = 1x4 uint16 row vector

   0000   0100   8000   ffff
```

Swap Byte Ordering of 3-D Array

Create a three-dimensional array A of 16-bit integers. Swap the byte ordering of each element. Use hexadecimal representation to show the endian conversion.

```
format hex
X = uint16(magic(3)*150);
X(:,:,2) = X*40;
X
```

```
X = 3x3x2 uint16 array
X(:,:,1) =

   04b0   0096   0384
   01c2   02ee   041a
   0258   0546   012c


X(:,:,2) =

   bb80   1770   8ca0
   4650   7530   a410
   5dc0   d2f0   2ee0
```

```
Y = swapbytes(X)
```

```
Y = 3x3x2 uint16 array
Y(:,:,1) =

   b004   9600   8403
   c201   ee02   1a04
   5802   4605   2c01


Y(:,:,2) =

   80bb   7017   a08c
   5046   3075   10a4
   c05d   f0d2   e02e
```

Input Arguments

X — Numeric value

scalar | vector | matrix | multidimensional array

Numeric value, specified as a scalar, vector, matrix, or multidimensional array. The `swapbytes` operation is elementwise when `X` is nonscalar.

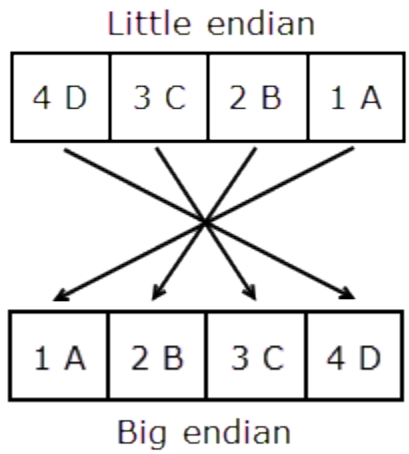
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

More About

Convert Little Endian to Big Endian

Computer systems store data in pieces of bytes. There are two ways to store a numeric value in computer memory: little endian and big endian. In little-endian style, the bytes are written from left to right in increasing significance. In big-endian style, the bytes are written from left to right in decreasing significance. The `swapbytes` function swaps the byte ordering in memory, converting little endian to big endian (and vice versa).

The following figure illustrates the conversion for a 32-bit (4-byte) hexadecimal number, 4D3C2B1A. In hexadecimal notation, one byte is represented by two digits. Each box in the little-endian picture shows one byte of data (4D, 3C, 2B, and 1A). The `swapbytes` function swaps the order of these boxes or bytes.



Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Inheritance of the class of the input to `swapbytes` in a MATLAB Function block is supported only when the class of the input is `double`. For non-double inputs, the input port data types must be specified, not inherited.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

Inheritance of the class of the input to `swapbytes` in a MATLAB Function block is supported only when the class of the input is `double`. For non-double inputs, the input port data types must be specified, not inherited.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`typecast`

Introduced before R2006a

switch, case, otherwise

Execute one of several groups of statements

Syntax

```
switch switch_expression
   case case_expression
      statements
   case case_expression
      statements
    ...
   otherwise
      statements
end
```

Description

`switch` *`switch_expression`*`, case` *`case_expression`*`, end` evaluates an expression and chooses to execute one of several groups of statements. Each choice is a case.

The `switch` block tests each case until one of the case expressions is true. A case is true when:

- For numbers, *`case_expression == switch_expression`*.
- For character vectors, `strcmp(`*`case_expression`*`,`*`switch_expression`*`) == 1`.
- For objects that support the `eq` function, *`case_expression == switch_expression`*. The output of the overloaded `eq` function must be a logical value.
- For a cell array *`case_expression`*, at least one of the elements of the cell array matches *`switch_expression`*, as defined above for numbers, character vectors, and objects.

When a case expression is true, MATLAB executes the corresponding statements and exits the `switch` block.

An evaluated *switch_expression* must be a scalar or character vector. An evaluated *case_expression* must be a scalar, a character vector, or a cell array of scalars or character vectors.

The `otherwise` block is optional. MATLAB executes the statements only when no case is true.

Examples

Compare Single Values

Display different text conditionally, depending on a value entered at the command prompt.

```
n = input('Enter a number: ');

switch n
    case -1
        disp('negative one')
    case 0
        disp('zero')
    case 1
        disp('positive one')
    otherwise
        disp('other value')
end
```

At the command prompt, enter the number 1.

```
positive one
```

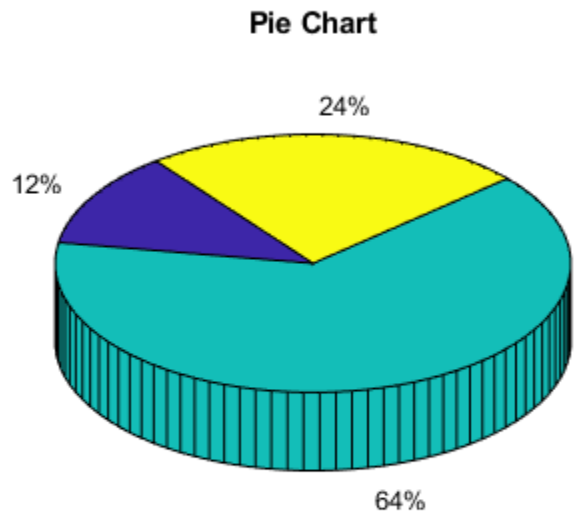
Repeat the code and enter the number 3.

```
other value
```

Compare Against Multiple Values

Determine which type of plot to create based on the value of `plottype`. If `plottype` is either `'pie'` or `'pie3'`, create a 3-D pie chart. Use a cell array to contain both values.

```
x = [12 64 24];
plottype = 'pie3';

switch plottype
    case 'bar'
        bar(x)
        title('Bar Graph')
    case {'pie','pie3'}
        pie3(x)
        title('Pie Chart')
    otherwise
        warning('Unexpected plot type. No plot created.')
end
```



Tips

- A *case_expression* cannot include relational operators such as < or > for comparison against the *switch_expression*. To test for inequality, use `if, elseif, else` statements.
- The MATLAB `switch` statement does not fall through like a C language `switch` statement. If the first `case` statement is `true`, MATLAB does not execute the other `case` statements. For example:

```
result = 52;

switch(result)
   case 52
      disp('result is 52')
   case {52, 78}
      disp('result is 52 or 78')
end
```

```
result is 52
```

- Define all variables necessary for code in a particular case within that case. Since MATLAB executes only one case of any `switch` statement, variables defined within one case are not available for other cases. For example, if your current workspace does not contain a variable `x`, only cases that define `x` can use it:

```
switch choice
   case 1
      x = -pi:0.01:pi;
   case 2
      % does not know anything about x
end
```

- The MATLAB `break` statement ends execution of a `for` or `while` loop, but does not end execution of a `switch` statement. This behavior is different than the behavior of `break` and `switch` in C.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If all case expressions are scalar integer values, then the code generator produces a C `switch` statement. At run time, if the switch value is not an integer, then the code generator produces an error.
- When the case expressions contain noninteger or nonscalar values, the code generator produces C `if` statements in place of a C `switch` statement.

See Also

`end` | `for` | `if` | `while`

Introduced before R2006a

sylvester

Solve Sylvester equation $AX + XB = C$ for X

Syntax

```
X = sylvester(A,B,C)
```

Description

`X = sylvester(A,B,C)` returns the solution, `X`, to the Sylvester equation on page 1-15046.

Input `A` is an m-by-m matrix, input `B` is an n-by-n matrix, and both `C` and `X` are m-by-n matrices.

Examples

Solve Sylvester Equation with 3-by-3 Output

Create the coefficient matrices `A` and `B`.

```
A = [1 -1 1; 1 1 -1; 1 1 1];
B = magic(3);
```

Define `C` as the 3-by-3 identity matrix.

```
C = eye(3);
```

Use the `sylvester` function to solve the Sylvester equation for these values of `A`, `B`, and `C`.

```
X = sylvester(A,B,C)
```

```
X = 3×3
```

```
    0.1223   -0.0725    0.0131
   -0.0806   -0.0161    0.1587
   -0.0164    0.1784   -0.1072
```

The result is a 3-by-3 matrix.

Solve Sylvester Equation with 4-by-2 Output

Create a 4-by-4 coefficient matrix, `A`, and 2-by-2 coefficient matrix, `B`.

```
A = [1 0 2 3; 4 1 0 2; 0 5 5 6; 1 7 9 0];
B = [0 -1; 1 0];
```

Define `C` as a 4-by-2 matrix to match the corresponding sizes of `A` and `B`.

```
C = [1 0; 2 0; 0 3; 1 1]
```

```
C = 4×2

     1     0
     2     0
     0     3
     1     1
```

Use the `sylvester` function to solve the Sylvester equation for these values of `A`, `B`, and `C`.

```
X = sylvester(A,B,C)
```

```
X = 4×2

    0.4732   -0.3664
   -0.4006    0.3531
    0.3305   -0.1142
    0.0774    0.3560
```

The result is a 4-by-2 matrix.

Input Arguments

A,B,C — Input matrices
matrices

Input matrices, specified as matrices. Input A is an m-by-m square matrix, input B is an n-by-n square matrix, and input C is an m-by-n rectangular matrix. The function returns an error if any input matrix is sparse.

Data Types: single | double
Complex Number Support: Yes

Output Arguments

X — Solution
matrix

Solution, returned as a matrix of the same size as C. The function returns an error if the eigenvalues of A and -B are not distinct (in this case, the solution, X, is singular or not unique).

More About

Sylvester Equation

The Sylvester equation is

$$AX + XB = C\,.$$

The equation has a unique solution when the eigenvalues of A and -B are distinct. In terms of the Kronecker tensor product, ⊗, the equation is

$$\left[I \otimes A + B^T \otimes I\right]X(:) = C(:),$$

where I is the identity matrix, and X(:) and C(:) denote the matrices X and C as single column vectors.

See Also

`ctranspose` | `eig` | `kron` | `mldivide` | `mtimes`

Introduced in R2014a

symamd

Symmetric approximate minimum degree permutation

Syntax

```
p = symamd(S)
p = symamd(S,knobs)
[p,stats] = symamd(...)
```

Description

`p = symamd(S)` for a symmetric positive definite matrix `S`, returns the permutation vector `p` such that `S(p,p)` tends to have a sparser Cholesky factor than `S`. To find the ordering for `S`, `symamd` constructs a matrix `M` such that `spones(M'*M) = spones (S)`, and then computes `p = colamd(M)`. The `symamd` function may also work well for symmetric indefinite matrices.

`S` must be square; only the strictly lower triangular part is referenced.

`p = symamd(S,knobs)` where `knobs` is a scalar. If `S` is `n`-by-`n`, rows and columns with more than `knobs*n` entries are removed prior to ordering, and ordered last in the output permutation `p`. If the `knobs` parameter is not present, then `knobs = spparms('wh_frac')`.

`[p,stats] = symamd(...)` produces the optional vector `stats` that provides data about the ordering and the validity of the matrix `S`.

`stats(1)`	Number of dense or empty rows ignored by `symamd`
`stats(2)`	Number of dense or empty columns ignored by `symamd`
`stats(3)`	Number of garbage collections performed on the internal data structure used by `symamd` (roughly of size `8.4*nnz(tril(S,-1)) + 9n` integers)
`stats(4)`	`0` if the matrix is valid, or `1` if invalid

`stats(5)`	Rightmost column index that is unsorted or contains duplicate entries, or `0` if no such column exists
`stats(6)`	Last seen duplicate or out-of-order row index in the column index given by `stats(5)`, or `0` if no such row index exists
`stats(7)`	Number of duplicate and out-of-order row indices

Although, MATLAB built-in functions generate valid sparse matrices, a user may construct an invalid sparse matrix using the MATLAB C or Fortran APIs and pass it to `symamd`. For this reason, `symamd` verifies that `S` is valid:

- If a row index appears two or more times in the same column, `symamd` ignores the duplicate entries, continues processing, and provides information about the duplicate entries in `stats(4:7)`.
- If row indices in a column are out of order, `symamd` sorts each column of its internal copy of the matrix `S` (but does not repair the input matrix `S`), continues processing, and provides information about the out-of-order entries in `stats(4:7)`.
- If `S` is invalid in any other way, `symamd` cannot continue. It prints an error message, and returns no output arguments (`p` or `stats`).

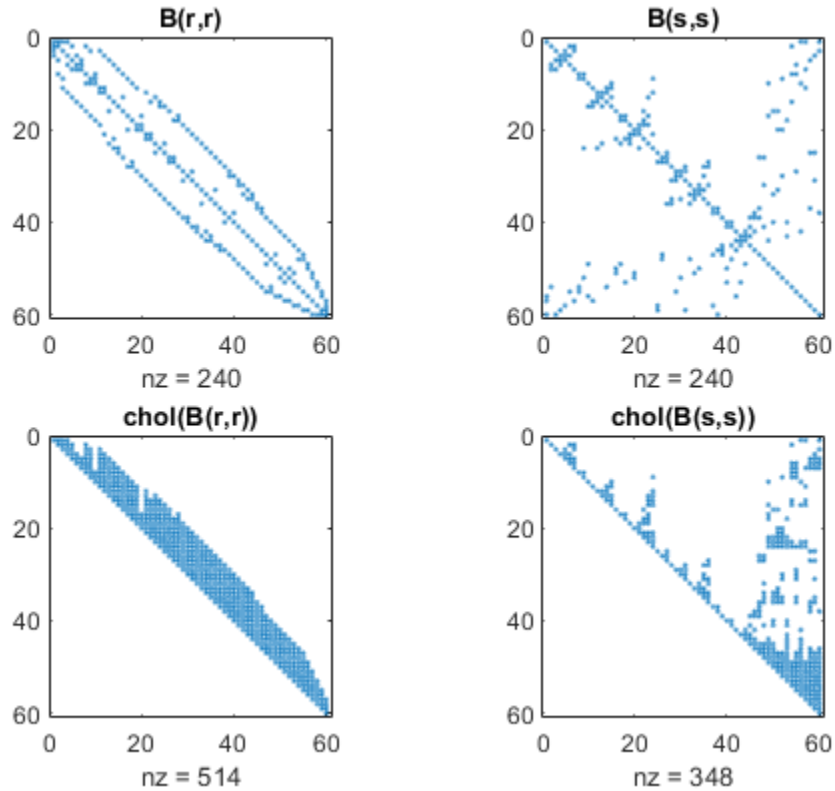
The ordering is followed by a symmetric elimination tree post-ordering.

Examples

Compare Reverse Cuthill-McKee and Minimum Degree

Here is a comparison of reverse Cuthill-McKee and minimum degree on the Bucky ball example mentioned in the `symrcm` reference page.

```
B = bucky+4*speye(60);
r = symrcm(B);
p = symamd(B);
R = B(r,r);
S = B(p,p);
subplot(2,2,1), spy(R,4), title('B(r,r)')
subplot(2,2,2), spy(S,4), title('B(s,s)')
subplot(2,2,3), spy(chol(R),4), title('chol(B(r,r))')
subplot(2,2,4), spy(chol(S),4), title('chol(B(s,s))')
```



Even though this is a very small problem, the behavior of both orderings is typical. RCM produces a matrix with a narrow bandwidth which fills in almost completely during the Cholesky factorization. Minimum degree produces a structure with large blocks of contiguous zeros which do not fill in during the factorization. Consequently, the minimum degree ordering requires less time and storage for the factorization.

References

The authors of the code for `symamd` are Stefan I. Larimore and Timothy A. Davis (`davis@cise.ufl.edu`), University of Florida. The algorithm was developed in collaboration with John Gilbert, Xerox PARC, and Esmond Ng, Oak Ridge National

Laboratory. Sparse Matrix Algorithms Research at the University of Florida: `https://www.cise.ufl.edu/research/sparse/`

See Also

`amd` | `colamd` | `colperm` | `dissect` | `spparms` | `symrcm`

Introduced before R2006a

symbfact

Symbolic factorization analysis

Syntax

```
count = symbfact(A)
count = symbfact(A,'sym')
count = symbfact(A,'col')
count = symbfact(A,'row')
count = symbfact(A,'lo')
[count,h,parent,post,R] = symbfact(...)
[count,h,parent,post,L] = symbfact(A,type,'lower')
```

Description

`count = symbfact(A)` returns the vector of row counts of `R=chol(A)`. `symbfact` should be much faster than `chol(A)`.

`count = symbfact(A,'sym')` is the same as `count = symbfact(A)`.

`count = symbfact(A,'col')` returns row counts of `R=chol(A'*A)` (without forming it explicitly).

`count = symbfact(A,'row')` returns row counts of `R=chol(A*A')`.

`count = symbfact(A,'lo')` is the same as `count = symbfact(A)` and uses `tril(A)`.

`[count,h,parent,post,R] = symbfact(...)` has several optional return values.

The flop count for a subsequent Cholesky factorization is `sum(count.^2)`

Return Value	Description
`h`	Height of the elimination tree
`parent`	The elimination tree itself

Return Value	Description
`post`	Postordering of the elimination tree
`R`	0-1 matrix having the structure of `chol(A)` for the symmetric case, `chol(A'*A)` for the `'col'` case, or `chol(A*A')` for the `'row'` case.

`symbfact(A)` and `symbfact(A,'sym')` use the upper triangular part of `A (triu(A))` and assume the lower triangular part is the transpose of the upper triangular part. `symbfact(A,'lo')` uses `tril(A)` instead.

`[count,h,parent,post,L] = symbfact(A,type,'lower')` where `type` is one of `'sym'`,`'col'`, `'row'`, or`'lo'` returns a lower triangular symbolic factor `L=R'`. This form is quicker and requires less memory.

See Also

`chol` | `etree` | `treelayout`

Introduced before R2006a

symmlq

Symmetric LQ method

Syntax

```
x = symmlq(A,b)
symmlq(A,b,tol)
symmlq(A,b,tol,maxit)
symmlq(A,b,tol,maxit,M)
symmlq(A,b,tol,maxit,M1,M2)
symmlq(A,b,tol,maxit,M1,M2,x0)
[x,flag] = symmlq(A,b,...)
[x,flag,relres] = symmlq(A,b,...)
[x,flag,relres,iter] = symmlq(A,b,...)
[x,flag,relres,iter,resvec] = symmlq(A,b,...)
[x,flag,relres,iter,resvec,resveccg] = symmlq(A,b,...)
```

Description

`x = symmlq(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be symmetric but need not be positive definite. It should also be large and sparse. The column vector `b` must have length `n`. You can specify `A` as a function handle, `afun`, such that `afun(x)` returns `A*x`.

“Parameterizing Functions” explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `symmlq` converges, a message to that effect is displayed. If `symmlq` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`symmlq(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `symmlq` uses the default, `1e-6`.

`symmlq(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `symmlq` uses the default, `min(n,20)`.

`symmlq(A,b,tol,maxit,M)` and `symmlq(A,b,tol,maxit,M1,M2)` use the symmetric positive definite preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(sqrt(M))*A*inv(sqrt(M))*y = inv(sqrt(M))*b` for `y` and then return `x = in(sqrt(M))*y`. If `M` is `[]` then `symmlq` applies no preconditioner. `M` can be a function handle `mfun` such that `mfun(x)` returns `M\x`.

`symmlq(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is `[]`, then `symmlq` uses the default, an all-zero vector.

`[x,flag] = symmlq(A,b,...)` also returns a convergence flag.

Flag	Convergence
0	`symmlq` converged to the desired tolerance `tol` within `maxit` iterations.
1	`symmlq` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	`symmlq` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `symmlq` became too small or too large to continue computing.
5	Preconditioner `M` was not symmetric positive definite.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = symmlq(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = symmlq(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = symmlq(A,b,...)` also returns a vector of estimates of the `symmlq` residual norms at each iteration, including `norm(b-A*x0)`.

`[x,flag,relres,iter,resvec,resveccg] = symmlq(A,b,...)` also returns a vector of estimates of the conjugate gradients residual norms at each iteration.

Examples

Example 1

```
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -2*on],-1:1,n,n);
b = sum(A,2);
tol = 1e-10;
maxit = 50; M1 = spdiags(4*on,0,n,n);

x = symmlq(A,b,tol,maxit,M1);
symmlq converged at iteration 49 to a solution with relative
residual 4.3e-015
```

Example 2

This example replaces the matrix `A` in Example 1 with a handle to a matrix-vector product function `afun`. The example is contained in the function `run_symmlq` that:

- Calls `symmlq` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_symmlq` are available to `afun`.

The following shows the code for `run_symmlq`:

```
function x1 = run_symmlq
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x1 = symmlq(@afun,b,tol,maxit,M1);

       function y = afun(x)
          y = 4 * x;
          y(2:n) = y(2:n) - 2 * x(1:n-1);
          y(1:n-1) = y(1:n-1) - 2 * x(2:n);
```

```
        end
end
```

When you enter

```
x1=run_symmlq;
```

MATLAB software displays the message

```
symmlq converged at iteration 49 to a solution with relative
residual 4.3e-015
```

Example 3

Use a symmetric indefinite matrix that fails with `pcg`.

```
A = diag([20:-1:1,-1:-1:-20]);
b = sum(A,2);      % The true solution is the vector of all ones.
x = pcg(A,b);      % Errors out at the first iteration.
pcg stopped at iteration 1 without converging to the desired
tolerance 1e-006 because a scalar quantity became too small or
too large to continue computing.
The iterate returned (number 0) has relative residual 1
```

However, `symmlq` can handle the indefinite matrix `A`.

```
x = symmlq(A,b,1e-6,40);
symmlq converged at iteration 39 to a solution with relative
residual 1.3e-007
```

References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Paige, C. C. and M. A. Saunders, "Solution of Sparse Indefinite Systems of Linear Equations." *SIAM J. Numer. Anal.*, Vol.12, 1975, pp. 617-629.

Extended Capabilities

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- If `M1` is a function, then it is applied independently to each row.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bicg` | `bicgstab` | `cgs` | `gmres` | `lsqr` | `minres` | `mldivide` | `pcg` | `qmr`

Topics

“Create Function Handle”

Introduced before R2006a

symrcm

Sparse reverse Cuthill-McKee ordering

Syntax

```
r = symrcm(S)
```

Description

`r = symrcm(S)` returns the symmetric reverse Cuthill-McKee ordering of `S`. This is a permutation `r` such that `S(r,r)` tends to have its nonzero elements closer to the diagonal. This is a good preordering for LU or Cholesky factorization of matrices that come from long, skinny problems. The ordering works for both symmetric and nonsymmetric `S`.

For a real, symmetric sparse matrix, `S`, the eigenvalues of `S(r,r)` are the same as those of `S`, but `eig(S(r,r))` probably takes less time to compute than `eig(S)`.

Examples

Reverse Cuthill-McKee Ordering

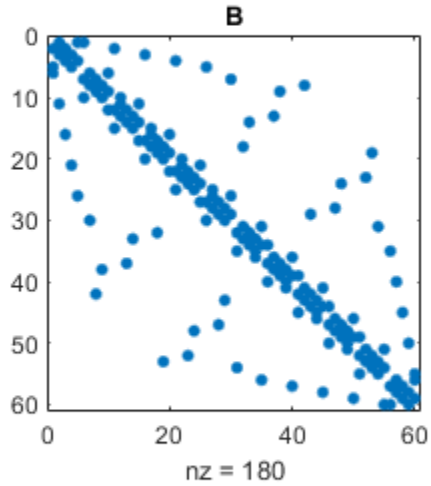
The statement

```
B = bucky;
```

uses a function in the `demos` toolbox to generate the adjacency graph of a truncated icosahedron. This is better known as a soccer ball, a Buckminster Fuller geodesic dome (hence the name `bucky`), or, more recently, as a 60-atom carbon molecule. There are 60 vertices. The vertices have been ordered by numbering half of them from one hemisphere, pentagon by pentagon; then reflecting into the other hemisphere and gluing the two halves together.

With this numbering, the matrix does not have a particularly narrow bandwidth, as the first spy plot shows:

```
figure();
subplot(1,2,1),spy(B),title('B')
```

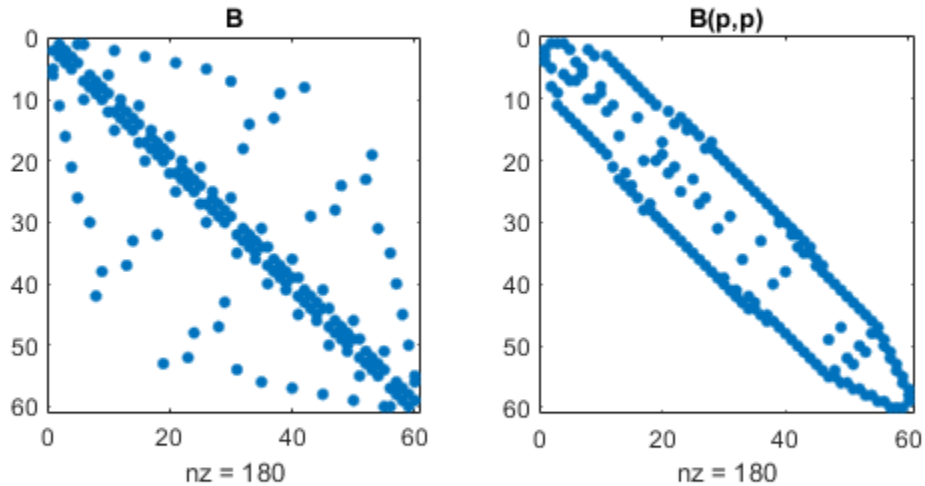


The reverse Cuthill-McKee ordering is obtained with:

```
p = symrcm(B);
R = B(p,p);
```

The `spy` plot shows a much narrower bandwidth.

```
subplot(1,2,2),spy(R),title('B(p,p)')
```



This example is continued in the reference page for `symamd`.

The bandwidth can also be computed with:

```
[i,j] = find(B);
bw = max(i-j) + 1;
```

The bandwidths of `B` and `R` are 35 and 12, respectively.

Algorithms

The algorithm first finds a pseudoperipheral vertex of the graph of the matrix. It then generates a level structure by breadth-first search and orders the vertices by decreasing distance from the pseudoperipheral vertex. The implementation is based closely on the SPARSPAK implementation described by George and Liu.

References

[1] George, Alan and Joseph Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, 1981.

[2] Gilbert, John R., Cleve Moler, and Robert Schreiber, “Sparse Matrices in MATLAB: Design and Implementation,” *SIAM Journal on Matrix Analysis*, 1992. A slightly expanded version is also available as a technical report from the Xerox Palo Alto Research Center.

See Also

`colamd` | `colperm` | `dissect` | `symamd`

Introduced before R2006a

symvar

Determine symbolic variables in expression

Syntax

```
C = symvar(expr)
```

Description

`C = symvar(expr)` searches the expression, `expr`, for identifiers other than `i`, `j`, `pi`, `inf`, `nan`, `eps`, and common functions. These identifiers are the names of variables in the expression. `symvar` returns the identifiers in a cell array of character vectors, `C`. If `symvar` finds no identifiers, then `C` is an empty cell array.

Examples

Find Identifiers in Expression

Create a character vector that contains an expression. Find the identifiers in the expression and return them in a cell array.

```
C = symvar('cos(pi*x - beta1)')
```

```
C = 2x1 cell array
    {'beta1'}
    {'x'    }
```

Input Arguments

expr — Input expression
character vector | string scalar

Input expression, specified as a character vector or string scalar.

See Also

`eval` | `evalc` | `strfind`

Introduced before R2006a

synchronize

Synchronize timetables to common time vector, and resample or aggregate data from input timetables

Syntax

```
TT = synchronize(TT1,TT2)
```

```
TT = synchronize(TT1,TT2,newTimeBasis,method)
TT = synchronize(TT1,TT2,newTimeStep,method)
TT = synchronize(TT1,TT2,'regular',method,'TimeStep',dt)
TT = synchronize(TT1,TT2,'regular',method,'SampleRate',Fs)
TT = synchronize(TT1,TT2,newTimes,method)
```

```
TT = synchronize(TT1,TT2,newTimeBasis)
TT = synchronize(TT1,TT2,newTimeStep)
TT = synchronize(TT1,TT2,'regular','TimeStep',dt)
TT = synchronize(TT1,TT2,'regular','SampleRate',Fs)
TT = synchronize(TT1,TT2,newTimes)
```

```
TT = synchronize(TT1,TT2,___,Name,Value)
TT = synchronize(TT1,...,TTN,___)
```

Description

The `synchronize` function collects the variables from all input timetables, synchronizes them to a common time vector, and returns the result as a single timetable. The effect is similar to a horizontal concatenation, though the input timetables can have different row times. When the `synchronize` function synchronizes timetable variables to different times, it also resamples or aggregates the data in the variables using a method that you specify.

`TT = synchronize(TT1,TT2)` creates a timetable, `TT`, that contains all variables from both the input timetables `TT1` and `TT2`, synchronized to a vector of row times that is the union of the row times from `TT1` and `TT2`. The row times of `TT` are in sorted order with no repeated times.

In effect, `synchronize` horizontally concatenates the variables of `TT1` and `TT2`, even when they have row times that differ. As a result, `synchronize` inserts a missing data indicator in `TT` wherever it has:

- A row time only from `TT1` but no corresponding data from the variables of `TT2`.
- A row time only from `TT2` but no corresponding data from the variables of `TT1`.

If `TT1` and `TT2` have variables with the same names, then `synchronize` renames them and copies both variables into `TT`.

To interpolate or fill in values in `TT` using different methods for different variables, specify the `VariableContinuity` property of each input timetable. For more information, see "Retime and Synchronize Timetable Variables Using Different Methods".

`TT = synchronize(TT1,TT2,newTimeBasis,method)` creates `TT` by synchronizing the variables from `TT1` and `TT2` to a new time vector specified by `newTimeBasis`. `synchronize` resamples or aggregates data from the variables in `TT1` and `TT2` using the operation specified by `method`. The input argument, `newTimeBasis`, specifies how `synchronize` constructs the row times of `TT` from the row times of `TT1` and `TT2`.

For example, if `newTimeBasis` is `'union'` and `method` is `'linear'`, then `TT` contains the row times from `TT1` and `TT2`, and `synchronize` uses linear interpolation to resample the data from the input timetables to the output row times.

`TT = synchronize(TT1,TT2,newTimeStep,method)` creates `TT` by synchronizing the variables from `TT1` and `TT2` to a new time vector that is regularly spaced by the time unit specified by `newTimeStep`.

For example, if `newTimeStep` is `'daily'` and `method` is `'mean'`, then `TT` contains row times that are one day apart, and `TT` contains daily means for each variable from `TT1` and `TT2`.

`TT = synchronize(TT1,TT2,'regular',method,'TimeStep',dt)` uses the time step `dt` to calculate regularly spaced row times. `dt` is a scalar duration or calendar duration. The row times of `TT` span the range of row times of `TT1` and `TT2`.

`TT = synchronize(TT1,TT2,'regular',method,'SampleRate',Fs)` uses the sample rate `Fs` to calculate regularly spaced row times. `Fs` is a positive numeric scalar that specifies the number of samples per second (Hz).

`TT = synchronize(TT1,TT2,newTimes,method)` creates `TT` by synchronizing the variables from `TT1` and `TT2` to `newTimes`, a datetime or duration vector of unique, sorted times that you specify. The times in `newTimes` become the row times of `TT`.

`TT = synchronize(TT1,TT2,newTimeBasis)` creates `TT` by synchronizing the variables from `TT1` and `TT2` to a new time vector. `synchronize` inserts missing data indicators where needed in `TT`.

`TT = synchronize(TT1,TT2,newTimeStep)` creates `TT` by synchronizing the variables from `TT1` and `TT2` to a new time vector that is regularly spaced by the time unit specified by `newTimeStep`. The `synchronize` function inserts missing data indicators where needed in `TT`.

`TT = synchronize(TT1,TT2,'regular','TimeStep',dt)` uses the time step `dt` to calculate regularly spaced row times. `dt` is a scalar duration or calendar duration. The row times of `TT` span the range of row times of `TT1` and `TT2`.

`TT = synchronize(TT1,TT2,'regular','SampleRate',Fs)` uses the sample rate `Fs` to calculate regularly spaced row times. `Fs` is a positive numeric scalar that specifies the number of samples per second (Hz).

`TT = synchronize(TT1,TT2,newTimes)` creates `TT` by synchronizing the variables from `TT1` and `TT2` to the time vector, `newTimes`. The `synchronize` function inserts missing data indicators where needed in `TT`.

`TT = synchronize(TT1,TT2, ___ ,Name,Value)` synchronizes timetables with additional options specified by one or more `Name,Value` pairs. You can use this syntax with the input arguments of any of the previous syntaxes.

`TT = synchronize(TT1,...,TTN, ___ )` creates the timetable, `TT`, by synchronizing the `N` timetables `TT1,...,TTN`. You can use this syntax with the input arguments of any of the previous syntaxes.

Examples

Synchronize Timetables and Insert Missing Data Indicators

Load two sample timetables from a file. Then synchronize their data to a vector of new row times.

```
load smallTT
```

Display the timetables. TT1 has row times that are out of order. TT1 and TT2 have different variables.

```
TT1
```

```
TT1=3×2 timetable
            Time            Temp
    ____________________    ____

    18-Dec-2015 12:00:00    42.3
    18-Dec-2015 08:00:00    37.3
    18-Dec-2015 10:00:00    39.1
```

```
TT2
```

```
TT2=3×2 timetable
            Time            Pressure
    ____________________    ________

    18-Dec-2015 09:00:00      30.1
    18-Dec-2015 11:00:00     30.03
    18-Dec-2015 13:00:00      29.9
```

Synchronize TT1 and TT2. The output timetable, TT, contains all the row times from both timetables, in sorted order. In TT, Temp contains NaN for row times from TT2, and Pressure contains NaN for row times from TT1.

```
TT = synchronize(TT1,TT2)
```

```
TT=6×3 timetable
            Time            Temp    Pressure
    ____________________    ____    ________

    18-Dec-2015 08:00:00    37.3       NaN
    18-Dec-2015 09:00:00     NaN      30.1
    18-Dec-2015 10:00:00    39.1       NaN
    18-Dec-2015 11:00:00     NaN     30.03
    18-Dec-2015 12:00:00    42.3       NaN
    18-Dec-2015 13:00:00     NaN      29.9
```

Interpolate Data Values to Row Times

Load two sample timetables that contain weather measurements. Synchronize their data to the union of the row times from the timetables.

```
load firstTT
```

Display the timetables.

```
TT1
```

```
TT1=3×2 timetable
            Time            Temp
    ____________________    ____

    06-Jun-2016 15:00:00    79.7
    06-Jun-2016 16:00:00    76.3
    06-Jun-2016 17:00:00    74.9
```

```
TT2
```

```
TT2=4×2 timetable
            Time            Humidity
    ____________________    ________

    06-Jun-2016 14:35:48      49.7
    06-Jun-2016 15:35:48      52.2
    06-Jun-2016 16:35:48      56.7
    06-Jun-2016 17:35:48        60
```

Synchronize the timetables. To select the union of row times, specify `'union'`. To resample `TT1.Temp` and `TT2.Humidity` using linear interpolation, specify `'linear'`.

```
TT = synchronize(TT1,TT2,'union','linear')
```

```
TT=7×3 timetable
            Time             Temp     Humidity
    ____________________    ______    ________

    06-Jun-2016 14:35:48    81.071       49.7
    06-Jun-2016 15:00:00      79.7     50.708
```

```
    06-Jun-2016 15:35:48    77.671       52.2
    06-Jun-2016 16:00:00      76.3     54.014
    06-Jun-2016 16:35:48    75.464       56.7
    06-Jun-2016 17:00:00      74.9      58.03
    06-Jun-2016 17:35:48    74.064         60
```

Aggregate Data When Synchronizing Timetables

Synchronize two timetables to new row times that specify time bins that each span a day. Aggregate the data from the input timetables into the daily time bins.

Load sample timetables that contain two different sets of environmental measurements, `indoors` and `outdoors`. The air quality data come from a sensor inside a building, while the weather measurements come from sensors outside. The timetables include measurements taken from November 15, 2015, to November 19, 2015.

```
load indoors
load outdoors
```

Display the first three lines of each timetable. They do not contain the same row times or variables.

```
indoors(1:3,:)
```

```
ans=3×3 timetable
           Time            Humidity    AirQuality
    ___________________    ________    __________

    2015-11-15 00:00:24       36           80
    2015-11-15 01:13:35       36           80
    2015-11-15 02:26:47       37           79
```

```
outdoors(1:3,:)
```

```
ans=3×4 timetable
           Time            Humidity    TemperatureF    PressureHg
    ___________________    ________    ____________    __________

    2015-11-15 00:00:24        49          51.3          29.61
    2015-11-15 01:30:24      48.9          51.5          29.61
```

```
    2015-11-15 03:00:24      48.9          51.5          29.61
```

Aggregate the data from the timetables into daily time bins using the `synchronize` function. Specify `'daily'` to aggregate the data into time bins that span one day apiece. Specify `'mean'` to obtain the mean values in each time bin for each variable.

```
TT = synchronize(indoors,outdoors,'daily','mean');
TT(1:3,:)
```

```
ans=3×6 timetable
           Time            Humidity_indoors    AirQuality    Humidity_outdoors    Tempe
    ___________________    ________________    __________    _________________    _____

    2015-11-15 00:00:00          36.5            80.05            48.931             51
    2015-11-16 00:00:00         36.85            80.35            47.924             51
    2015-11-17 00:00:00         36.85            79.45             48.45             51
```

Synchronize Timetables to Regular Times

Synchronize two small timetables with weather measurements to a set of regular row times that span the row times in the input timetables. Specify a time step as the interval between consecutive row times in the output timetable. Use linear interpolation to resample data for times in the time vector that do not match row times from the input timetables.

First, load the two timetables.

```
load indoors
load outdoors
```

Display the first three lines of each timetable. They do not contain the same row times or variables.

```
indoors(1:3,:)
```

```
ans=3×3 timetable
           Time            Humidity    AirQuality
    ___________________    ________    __________

    2015-11-15 00:00:24       36           80
```

```
    2015-11-15 01:13:35       36           80
    2015-11-15 02:26:47       37           79
```

```
outdoors(1:3,:)
```

```
ans=3×4 timetable
           Time            Humidity    TemperatureF    PressureHg
    ___________________    ________    ____________    __________

    2015-11-15 00:00:24        49          51.3          29.61
    2015-11-15 01:30:24      48.9          51.5          29.61
    2015-11-15 03:00:24      48.9          51.5          29.61
```

Aggregate the data from the timetables into 30-minute time bins using the `synchronize` function. Specify a regular time step using the `'regular'` input argument and the `'TimeStep'` name-value pair argument. You can use these arguments to create a timetable that is regular, but whose time step is not a predefined step such as `'hourly'`.

```
TT = synchronize(indoors,outdoors,'regular','linear','TimeStep',minutes(30));
TT(1:3,:)
```

```
ans=3×6 timetable
           Time            Humidity_indoors    AirQuality    Humidity_outdoors    Tempe
    ___________________    ________________    __________    _________________    _____

    2015-11-15 00:00:00           36                80                 49             5
    2015-11-15 00:30:00           36                80             48.967             5
    2015-11-15 01:00:00           36                80             48.934             5
```

Synchronize Timetables to Arbitrary Time Vector

Synchronize two small timetables with weather measurements to an arbitrary time vector. Use linear interpolation to resample data for times in the time vector that do not match row times from the input timetables.

Load two small timetables, with row times for measurements taken at the half-hour mark. However, in each timetable, there is a row time for data that was not collected at the half-hour mark. Both timetables are irregular, which means that the time step is different between consecutive row times.

```
load halfHoursTT
```

Display the timetables. TT1 and TT2 have three rows and different variables.

```
TT1
```

```
TT1=3×2 timetable
            Time             Temp
    ____________________    ____

    18-Dec-2015 08:30:00    47.6
    18-Dec-2015 09:30:00    49.2
    18-Dec-2015 10:47:23    51.4
```

```
TT2
```

```
TT2=3×2 timetable
            Time             Pressure
    ____________________    ________

    18-Dec-2015 07:00:00      29.7
    18-Dec-2015 09:30:00     30.03
    18-Dec-2015 10:30:00      29.9
```

Specify a time vector that includes some of the row times from each timetable.

```
newTimes = datetime('2015-12-18 08:30:00') + hours(0:1:2)
```

```
newTimes = 1x3 datetime array
   18-Dec-2015 08:30:00   18-Dec-2015 09:30:00   18-Dec-2015 10:30:00
```

Synchronize TT1 and TT2 to newTimes. The newTimes vector has times that are not row times of TT1 or TT2. To interpolate data values for times in newTimes that do not match row times in TT1 or TT2, specify 'linear'.

```
TT = synchronize(TT1,TT2,newTimes,'linear')
```

```
TT=3×3 timetable
            Time             Temp     Pressure
    ____________________    ______    ________

    18-Dec-2015 08:30:00      47.6     29.898
    18-Dec-2015 09:30:00      49.2      30.03
```

```
    18-Dec-2015 10:30:00    50.906       29.9
```

Synchronize Timetables with Intersecting Row Times

Load two sample timetables and synchronize their data to a vector of row times they have in common.

```
load intersectTT
```

Display the timetables. TT1 and TT2 have three rows and different variables.

```
TT1
```

```
TT1=3×2 timetable
            Time            Temp
    ____________________    ____

    18-Dec-2015 08:00:00    37.3
    18-Dec-2015 10:00:00    39.1
    18-Dec-2015 12:00:00    42.3
```

```
TT2
```

```
TT2=3×2 timetable
            Time            Pressure
    ____________________    ________

    18-Dec-2015 06:00:00      30.1
    18-Dec-2015 08:00:00     30.03
    18-Dec-2015 10:00:00      29.9
```

Synchronize TT1 and TT2, specifying `'intersection'` as the basis for the row times of the output timetable. TT has only two rows because TT1 and TT2 have only two row times in common.

```
TT = synchronize(TT1,TT2,'intersection')
```

```
TT=2×3 timetable
            Time            Temp    Pressure
    ____________________    ____    ________
```

```
    18-Dec-2015 08:00:00    37.3     30.03
    18-Dec-2015 10:00:00    39.1      29.9
```

There is no need to interpolate or fill unmatched rows, because TT1 and TT2 both have data for the row times they have in common.

Synchronize Timetables to Hourly Times

Synchronize two small timetables with weather measurements to an hourly time vector.

Load two small timetables. In each timetable, there is a row time for data that was not collected on the hour. Both timetables are irregular, which means that the time step is different between consecutive row times.

```
load irregularTT
```

Display the timetables. TT1 and TT2 have three rows and different variables.

```
TT1
```

```
TT1=3×2 timetable
            Time            Temp
    ____________________    ____

    18-Dec-2015 08:00:00    37.3
    18-Dec-2015 09:11:17    39.1
    18-Dec-2015 10:00:00    42.3
```

```
TT2
```

```
TT2=3×2 timetable
            Time            Pressure
    ____________________    ________

    18-Dec-2015 08:00:00      29.8
    18-Dec-2015 09:27:23      29.7
    18-Dec-2015 10:00:00      30.3
```

Synchronize TT1 and TT2, specifying `'hourly'` as the time step for the row times of the output timetable. TT has data from TT1 and TT2 where they have row times that are on the hour. TT has missing data indicators where it has a row time that TT1 and TT2 do not have.

```
TT = synchronize(TT1,TT2,'hourly')
```

```
TT=3×3 timetable
            Time            Temp    Pressure
    ____________________    ____    ________

    18-Dec-2015 08:00:00    37.3      29.8
    18-Dec-2015 09:00:00     NaN       NaN
    18-Dec-2015 10:00:00    42.3      30.3
```

Synchronize Timetables Using Multiple Methods

Synchronize two timetables. Apply the `mean` method to some timetable variables and the `sum` method to others.

Load two small timetables that contain weather measurements for Boston and Natick. Each timetable contains temperature and rainfall readings.

```
load citiesTT
Boston
```

```
Boston=6×3 timetable
           Time            Temp    Rain
    ___________________    ____    ____

    2016-06-09 06:03:00    59.5    0.05
    2016-06-09 12:00:23      63    0.08
    2016-06-09 18:02:57    61.7    0.13
    2016-06-10 06:01:47    55.4    0.15
    2016-06-10 12:06:00    62.3    0.87
    2016-06-10 18:02:57    58.8    0.33
```

```
Natick
```

```
Natick=5×3 timetable
           Time            Temp     Rain
```

```
    ____________________    _____    _____

    2016-06-09 12:00:23     61.2    0.076
    2016-06-09 17:59:00     60.3     0.11
    2016-06-10 09:03:01     56.1     0.19
    2016-06-10 12:07:03    62.17     0.72
    2016-06-10 17:59:57     58.3      0.1
```

Synchronize the measurements to daily times to produce mean temperatures and the sums of the rainfall measurements. `synchronize` applies the specified method to all timetable variables. To apply different methods to different timetable variables, index into the timetables to select different variables, and call `synchronize` for each method you use.

```
BOS = Boston(:,'Temp');
NTK = Natick(:,'Temp');
TT1 = synchronize(BOS,NTK,'daily','mean')
```

```
TT1=2×3 timetable
           Time            Temp_BOS    Temp_NTK
    ____________________    ________    ________

    2016-06-09 00:00:00       61.4       60.75
    2016-06-10 00:00:00     58.833      58.857
```

```
BOS = Boston(:,'Rain');
NTK = Natick(:,'Rain');
TT2 = synchronize(BOS,NTK,'daily','sum')
```

```
TT2=2×3 timetable
           Time            Rain_BOS    Rain_NTK
    ____________________    ________    ________

    2016-06-09 00:00:00      0.26       0.186
    2016-06-10 00:00:00      1.35        1.01
```

To combine all results in one timetable, concatenate `TT1` and `TT2`.

```
TT = [TT1 TT2]
```

```
TT=2×5 timetable
           Time            Temp_BOS    Temp_NTK    Rain_BOS    Rain_NTK
```

```
____________________    ________    ________    ________    ________

2016-06-09 00:00:00       61.4       60.75       0.26       0.186
2016-06-10 00:00:00     58.833      58.857       1.35        1.01
```

Input Arguments

TT1,TT2 — Input timetables
timetables

Input timetables.

newTimeBasis — Basis for computing row times of output timetable
character vector

Basis for computing row times of output timetable, specified as a character vector. `newTimeBasis` can be any of the listed methods.

Method	Description
`'union'` (default)	Union of the row times
`'intersection'`	Intersection of the row times
`'commonrange'`	Union of the row times, but over the intersection of the time ranges
`'first'`	Row times from the first input timetable only
`'last'`	Row times from the last input timetable only

newTimeStep — Time step for spacing times in output timetable
character vector

Time step for spacing times in the output timetable, specified as a character vector. `newTimeStep` can be any of the listed time steps.

Time Step	Description
`'yearly'`	One year
`'quarterly'`	One quarter
`'monthly'`	One month
`'weekly'`	One week
`'daily'`	One day
`'hourly'`	One hour
`'minutely'`	One minute
`'secondly'`	One second

The first row time of `TT` is at the beginning of the time unit that includes the earliest row time from the input timetables. The range of row times in `TT` covers the range of row times from `TT1` and `TT2`. However, `TT` might not include any of the actual row times from `TT1` or `TT2`, since they can have row times that are not at the beginnings of any time unit.

`Fs` — Sample rate
positive numeric scalar

Sample rate, specified as a positive numeric scalar. `Fs` specifies the number of samples per second (Hz).

`dt` — Time step
datetime scalar | duration scalar

Time step, specified as a datetime scalar or duration scalar.

Data Types: `datetime` | `duration` | `calendarDuration`

`newTimes` — New time vector
datetime vector | duration vector

New time vector, specified as a datetime vector or a duration vector. The new time vector must be a column vector. `newTimes` can have a different number of rows than any of the input timetables.

`method` — Method for adjusting timetable data
character vector | string scalar | function handle

Method for adjusting timetable data, specified as a character vector, string scalar, or function handle. You can use any of the listed methods to adjust the data from the input timetables.

Fill Methods

Copy data from the rows of each input timetable when row times of the output timetable match row times of the corresponding input. Then, fill the remaining elements of the output timetable with missing data indicators.

To fill the remaining elements with a constant instead of a missing data indicator, use the `'Constant'` name-value pair argument.

Method	Description
`'fillwithmissing'` (default)	Fill gaps in output with missing data indicators (for example, `NaN` for numeric variables).
`'fillwithconstant'`	Fill gaps in output with the value of the `'Constant'` name-value pair argument. The default value is `0`.

Nearest Neighbor Methods

Copy data from the rows of the input timetables to output timetable rows whose row times are the nearest match, according to the specified method. Input timetables must be sorted by row times.

Method	Description
`'previous'`	Copy data from the nearest preceding neighbor in the input timetable, proceeding from the end of the vector of row times. If there are duplicate row times, then `'previous'` indicates the last of the duplicates.
`'next'`	Copy data from the nearest following neighbor in the input timetable, proceeding from the beginning of the vector of row times. If there are duplicate row times, then `'next'` indicates the first of the duplicates.

Method	Description
`'nearest'`	Copy data from the nearest neighbor in the input timetable.

Interpolation Methods

Interpolate data values in the output timetable from data values in neighboring rows of the input timetables. Input timetables must have row times that are sorted and unique. To control how the data are extrapolated beyond the first and last row times of the input timetables, use the `'EndValues'` name-value pair argument.

Method	Description
`'linear'`	Use linear interpolation.
`'spline'`	Use piecewise cubic spline interpolation.
`'pchip'`	Use shape-preserving piecewise cubic interpolation.
`'makima'`	Use modified Akima cubic Hermite interpolation.

Aggregation Methods

Aggregate data from rows of the input timetables over time bins specified by the row times of the output timetable. Each row time of `TT` is the left edge of a time bin, with the next consecutive row time being the right edge. By default, the left edges are included in the time bins. To control whether the left or the right bin edges are included in the time bins, use the `'IncludedEdge'` name-value pair argument.

If you specify the time vector, `newTimes`, then `newTimes` must be sorted in ascending order. If you specify the time basis of the output timetable as `'first'` or `'last'`, then the row times of either the first or the last timetable must be sorted in ascending order.

All the listed methods omit `NaN`s, `NaT`s, and other missing data indicators, except for `func`. To include missing data indicators, specify `func` as a function handle to a function that includes them when aggregating data.

Method	Description
`'sum'`	Sum the values in each time bin.

Method	Description
`'mean'`	Calculate the mean of the values in each time bin.
`'prod'`	Calculate the product of the values in each time bin.
`'min'`	Calculate the minimum of the values in each time bin.
`'max'`	Calculate the maximum of the values in each time bin.
`'count'`	Count the number of values in each time bin.
`'firstvalue'`	Use the first value in each time bin.
`'lastvalue'`	Use the last value in each time bin.
`@func`	Use the function specified by the function handle (for example, `@std` to calculate the standard deviation for the values in each time bin). `func` must return an output argument that is a scalar or a row vector, and must accept empty inputs.

Default Method

The default method is equivalent to leaving `method` unspecified.

Method	Description
`'default'` (default)	Either fill gaps with missing data indicators, or use per-variable methods if they are specified by the `VariableContinuity` property of the input timetables.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `TT = synchronize(TT1,TT2,newTimes,'fillwithconstant','Constant',-1)` synchronizes the timetables `TT1` and `TT2` and assigns the value `-1` to elements in rows of `TT` with row times that do not match row times in the corresponding input timetables.

`Constant` — Value for filling gaps when method is `'fillwithconstant'`
array

Value for filling gaps when the method is `'fillwithconstant'`, specified as the comma-separated pair consisting of `'Constant'` and an array. The default value is `0`. The data type of the value specified by `'Constant'` must be compatible with the data types of the timetable variables.

Example: `TT = synchronize(TT1,TT2,'hourly','fillwithconstant','Constant','NONE')` fills gaps in `TT` with the character vector `'NONE'` when all the variables in `TT` contain text.

`EndValues` — Method for extrapolation when using interpolation method
`'extrap'` (default) | array

Method for extrapolation when using an interpolation method, specified as the comma-separated pair consisting of `'EndValues'` and either `'extrap'`, or an array. If you specify an array, then its data type must be compatible with the timetable variables.

Method	Description
`'extrap'` (default)	Extrapolate using the method specified by the `method` input argument
array	Extrapolate by filling gaps outside the range of input row times with an array

Example: `TT = synchronize(TT1,TT2,'daily','previous','EndValues',1000)` fills gaps in `TT` with previous row values where `TT` has row times within the range of row times from `TT1` and `TT2`, and with the value `1000` where `TT` has row times outside that range.

`IncludedEdge` — Edges to include in each time bin
`'left'` (default) | `'right'`

Edges to include in each time bin, specified as the comma-separated pair consisting of `'IncludedEdge'` and either `'left'` or `'right'`. Each row time of `TT` is the left edge of a time bin, with the next consecutive row time being the right edge.

Edges to Include	Description
`'left'` (default)	All bins include the left bin edge, except for the last bin, which includes both edges
`'right'`	All bins include the right bin edge, except for the first bin, which includes both edges

If you specify `'left'`, then the time bins include the left edges except for the last bin, which includes both edges. If you specify `'right'`, then the time bins include the right edges except for the first bin, which includes both edges.

Example: `TT = synchronize(TT1,TT2,'hourly','mean','IncludedEdge','right')` includes the right bin edge of each time bin.

Compatibility Considerations

`'SamplingRate'` is not recommended

Not recommended starting in R2018b

The `'SamplingRate'` name-value pair argument is not recommended. Use `'SampleRate'` instead. The corresponding timetable property is also named `SampleRate`.

For backward compatibility, you still can specify `'SamplingRate'` as the name of the name-value pair. However, the value is assigned to the `SampleRate` property.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- The `newTimes` input must be strictly increasing instead of strictly monotonic.

- The `'commonrange'` option for the `newTimeBasis` input is not supported.
- The `'spline'` interpolation method is not supported.
- The `'EndValues'` name-value pair is not supported.
- Synchronization to `'regular'` spaced row times is not supported.
- All variables in input timetables must support missing values (floating point, categorical, datetime, duration, or text) for the methods `'nearest'`, `'next'`, and `'previous'`.

For more information, see “Tall Arrays”.

See Also

`horzcat` | `innerjoin` | `outerjoin` | `retime` | `synchronize` | `vertcat`

Topics

“Create Timetables”
“Clean Timetable with Missing, Duplicate, or Nonuniform Times”
“Resample and Aggregate Data in Timetable”
“Combine Timetables and Synchronize Their Data”
“Retime and Synchronize Timetable Variables Using Different Methods”

Introduced in R2016b

syntax

Two ways to call MATLAB functions

Description

You can call MATLAB functions using either *command syntax* or *function syntax*, as described below.

Command Syntax

A function call in this syntax consists of the function name followed by one or more arguments separated by spaces:

```
functionname arg1 arg2 ... argn
```

Command syntax does not allow you to obtain any values that might be returned by the function. Attempting to assign output from the function to a variable using command syntax generates an error. Use function syntax instead.

Examples of command syntax:

```
save mydata.mat x y z
import java.awt.Button java.lang.String
```

Arguments are treated as character vectors. See the examples below, under “Argument Passing” on page 1-15087.

Function Syntax

A function call in this syntax consists of the function name followed by one or more arguments separated by commas and enclosed in parentheses:

```
functionname(arg1, arg2, ..., argn)
```

You can assign the output of the function to one or more output values. When assigning to more than one output variable, separate the variables by commas or spaces and enclose them in square brackets (`[]`):

```
[out1,out2,...,outn] = functionname(arg1, arg2, ..., argn)
```

Examples of function syntax:

```
copyfile('srcfile', '..\mytests', 'writable')
[x1,x2,x3,x4] = deal(A{:})
```

Arguments are passed to the function by value. See the examples below, under “Argument Passing” on page 1-15087.

Argument Passing

When calling a function using command syntax, MATLAB passes the arguments as character vectors. When using function syntax, arguments are passed by value.

In the following example, assign a value to `A` and then call `disp` on the variable to display the value passed. Calling `disp` with command syntax passes the variable name, `'A'`:

```
A = pi;
disp A
   A
```

while function syntax passes the value assigned to `A`:

```
A = pi;
disp(A)
   3.1416
```

The next example passes two character vectors to `strcmp` for comparison. Calling the function with command syntax compares the variable names, `'str1'` and `'str2'`:

```
str1 = 'one';    str2 = 'one';
strcmp str1 str2
ans =
     0        (unequal)
```

while function syntax compares the values assigned to the variables, `'one'` and `'one'`:

```
str1 = 'one';    str2 = 'one';
strcmp(str1, str2)
ans =
     1        (equal)
```

Passing Character Vectors

When using the function syntax to pass a character vector to a function, you must enclose the character vector in single quotes, (`'text'`). For example, to create a new folder called `myapptests`, use

```
mkdir('myapptests')
```

On the other hand, variables that contain character vectors do not need to be enclosed in quotes:

```
folder = 'myapptests';
mkdir(folder)
```

See Also

`mlint`

Topics

"Check Code for Errors and Warnings"

system

Execute operating system command and return output

Syntax

```
status = system(command)
[status,cmdout] = system(command)
[status,cmdout] = system(command,'-echo')
```

Description

`status = system(command)` calls the operating system to execute the specified command. The operation waits for the command to finish execution before returning the exit status of the command to the `status` variable.

The function starts a new cmd/shell process, executes `command`, exits the process, and returns to the MATLAB process. Updates to the system environment made by `command` are not visible to MATLAB.

`[status,cmdout] = system(command)` also returns the output of the command to `cmdout`. This syntax is most useful for commands that do not require user input, such as `dir`.

`[status,cmdout] = system(command,'-echo')` also displays (echoes) the command output in the MATLAB Command Window. This syntax is most useful for commands that require user input and that run correctly in the MATLAB Command Window.

Examples

Windows: Display Operating System Command Status and Output

Display the current folder using the `cd` command. A `status` of zero indicates that the command completed successfully. MATLAB returns a character vector containing the current folder in `cmdout`.

```
command = 'cd';
[status,cmdout] = system(command)
```

Windows: Save Command Exit Status

To create a folder named `mynew`, call the `mkdir` command and save the exit status to a variable. A `status` of zero indicates that the `mynew` folder was created successfully.

```
command = 'mkdir mynew';
status = system(command)
```

Windows: Open and Run a UI Command

Open Microsoft Notepad and immediately return the exit status to MATLAB by appending an ampersand (&) to the `notepad` command. A `status` of zero indicates that Notepad successfully started.

```
status = system('notepad &')
```

Windows: Save Command Output

Execute the `dir` command and view the exit status and command output. `cmdout` contains the command output.

```
[~,cmdout] = system('dir');
```

Windows: Save Unsuccessful Command Status and Error Message

Attempt to execute a command called `badcmd`. Then, view the `status` and `results` output arguments. When you call an invalid command, `status` indicates failure and `results` contains the DOS error message.

```
[status,results] = system('badcmd')
```

UNIX: Save Command Exit Status and Output

List all users who are currently logged in, and save the command exit status and output. Then, view the status. A `status` of zero indicates that the command completed successfully. MATLAB® returns a list of users in `cmdout`.

```
command = 'who';
[status,cmdout] = system(command);
status
```

```
status = 0
```

Input Arguments

`command` — Operating system command
string | character vector

Operating system command, specified as a string or a character vector. The `command` executes in a system shell, which might not be the shell from which you started MATLAB.

Example: `'dir'`

Example: `'ls'`

Output Arguments

`status` — Command exit status
0 | nonzero integer

Command exit status, returned as either `0` or a nonzero integer. When the command is successful, `status` is `0`. Otherwise, `status` is a nonzero integer.

- If `command` includes the ampersand character (&), then `status` is the exit status when `command` starts
- If `command` does not include the ampersand character (&), then `status` is the exit status upon `command` completion.

`cmdout` — Output of operating system command
character vector

Output of the operating system command, returned as a character vector. The system shell might not properly represent non-Unicode characters.

Limitations

- MATLAB converts characters to the encoding that your operating system shell accepts. Output from the `command` is converted to the MATLAB encoding to be displayed in the command window. If you get unexpected results from the command, enter the `command` argument directly at the operating system prompt to see how the operating system treats your input.
- Callback functions are not called while the `system` command is executing.

More About

Windows Tips and Limitations

- MS-DOS does not support UNC path names. Therefore, if the current folder uses a UNC path name, then running `system` with a DOS `command` that relies on the current folder fails. To work around this limitation, change the folder to a mapped drive before calling `system`.
- You can override an environment variable in the `system` command. For example, the following code sets the `PATH` variable to *`myPath`*, then calls the system command *`dosCommand`* with that value.

  ```
  system(['set PATH=' myPath ' && ' dosCommand])
  ```

- To execute the operating system command in the background, include the trailing character, &, in the `command` argument. For example, type `'notepad &'`. The exit status is immediately returned to the `status` variable. This syntax is useful for console programs that require interactive user command input while they run, and that do not run correctly in the MATLAB Command Window.

  If `command` includes the trailing & character, then `cmdout` is empty.

- The `system` command uses the same credentials as the credentials used to start MATLAB. To set credentials for program `myprogram.exe` to require admin privileges, type:

  ```
  system('cmd /C myprogram.exe');
  ```

UNIX Tips and Limitations

- MATLAB uses a shell program to execute the given command. It determines which shell program to use by checking environment variables on your system. MATLAB first checks the `MATLAB_SHELL` variable, and if either empty or not defined, then checks `SHELL`. If `SHELL` is also empty or not defined, MATLAB uses `/bin/sh`.
- The `system` function redirects `stdin` to `command` by default. This redirection also passes MATLAB script commands and the keyboard type-ahead buffer to the invoked command while the `system` function executes. This behavior can lead to corrupted output when `system` does not complete execution immediately. To disable `stdin` and type-ahead redirection, include the formatted text `< /dev/null` in the call to the invoked command.
- You can override an environment variable in the `system` command. The syntax depends on the UNIX shell. For example, using the BASH shell, the following code sets the `PATH` variable to *`myPath`*, then calls the system command *`command`* with that value.

  ```
  system(['export PATH=' myPath ' ; ' command])
  ```
- To execute the operating system command in the background, include the trailing character, &, in the `command` argument. For example, type `'emacs &'`. The exit status is immediately returned to the `status` variable. This syntax is useful for console programs that require interactive user command input while they run, and that do not run correctly in the MATLAB Command Window.

  If `command` includes the trailing & character, then `cmdout` is empty.

See Also

`computer` | `perl`

Topics

"Run External Commands, Scripts, and Programs"
`!` (exclamation point)

External Websites

Why does the output of my FORTRAN script not show up in the MATLAB command window when I execute it using the SYSTEM function in MATLAB 7.14 (R2012a)?
On some Linux platforms, system might return partial results when called in a loop

Introduced before R2006a

table

Table array with named variables that can contain different types

Description

`table` arrays store column-oriented or tabular data, such as columns from a text file or spreadsheet. Tables store each piece of column-oriented data in a *variable*. Table variables can have different data types and sizes as long as all variables have the same number of rows. Table variables have names, just as the fields of a structure have names. Use the `summary` function to get information about a table.

To index into a table, use smooth parentheses `()` to return a subtable or curly braces `{}` to extract the contents. You can access variables and rows using names. For more information on indexing using numbers and names, see “Access Data in Tables”.

Creation

You can read data from a file into a table using either the `Import Tool` or the `readtable` function. Alternatively, use the `table` function described below to create a table from existing workspace variables.

You also can create a table that allows space for variables whose values are filled in later. To create a table with preallocated space for variables, use the `table` function with `'Size'` as the first input argument, as described below.

Syntax

```
T = table(var1,...,varN)
T = table('Size',sz,'VariableTypes',varTypes)
T = table( ___ ,'VariableNames',varNames)
T = table( ___ ,'RowNames',rowNames)
T = table
```

Description

`T = table(var1,...,varN)` creates a table from the input variables `var1,...,varN`. The variables can have different sizes and data types, but all variables must have the same number of rows.

If the inputs are workspace variables, then `table` assigns their names as the variable names in the output table. Otherwise, `table` assigns variable names of the form `'Var1',...,'Var*N*'`, where *N* is the number of variables.

`T = table('Size',sz,'VariableTypes',varTypes)` creates a table and preallocates space for the variables that have data types you specify. `sz` is a two-element numeric array, where `sz[1]` specifies the number of rows and `sz[2]` specifies the number of variables. `varTypes` is a cell array of character vectors specifying data types.

`T = table( ___ ,'VariableNames',varNames)` specifies the names of the variables in the output table. You can use this syntax with the input arguments from any of the other syntaxes for this function.

`T = table( ___ ,'RowNames',rowNames)` specifies names of the rows in the output table. You can use this syntax with the input arguments of any of the previous syntaxes.

`T = table` creates an empty 0-by-0 table.

Input Arguments

`var1,...,varN` — Input variables
arrays

Input variables, specified as arrays with the same number of rows. The input variables can have different sizes and different data types.

Common input variables are numeric arrays, logical arrays, character arrays, structure arrays, or cell arrays. Input variables also can be objects that are arrays. Such an array must support indexing of the form `var(index1,...,indexN)`, where `index1` is a numeric or logical vector that corresponds to rows of the variable `var`. In addition, the array must implement both a `vertcat` method and a `size` method with a `dim` argument.

Example: `table([1:4]',ones(4,3,2),eye(4,2))` creates a table from variables with four rows, but different sizes.

Example: `table([1:3]', {'one';'two';'three'},categorical({'A';'B';'C'}))` creates a table from variables with three rows, but different data types.

sz — Size of preallocated table
two-element numeric vector

Size of the preallocated table, specified as a two-element numeric vector. The first element of `sz` specifies the number of rows, and the second element specifies the number of table variables.

To create variables only, without any rows, specify `0` as the first element of `sz`.

Example: `T = table('Size',[50 3],'VariableTypes', {'string','double','datetime'})` preallocates 50 rows for a table that contains a string array, a double array, and a datetime array.

Example: `T = table('Size',[0 4],'VariableTypes',varTypes)` specifies zero rows and four variables.

varTypes — Data types of preallocated variables
cell array of character vectors | string array

Data types of the preallocated variables, specified as a cell array of character vectors or a string array. The number of types specified by `varTypes` must equal the number of variables specified by the second element of `sz`.

`varTypes` can contain the names of any data types, including the names shown in the table.

Data Type Name	Initial Value in Each Element
`'double'`, `'single'`	Double- or single-precision `0`
`'doublenan'`, `'doubleNaN'`, `'singlenan'`, `'singleNaN'`	Double- or single-precision `NaN`
`'int8'`, `'int16'`, `'int32'`, `'int64'`	Signed 8-, 16-, 32-, or 64-bit integer `0`
`'uint8'`, `'uint16'`, `'uint32'`, `'uint64'`	Unsigned 8-, 16-, 32-, or 64-bit integer `0`
`'logical'`	`0` (`false`)

Data Type Name	Initial Value in Each Element
`'categorical'`	`<undefined>` categorical value
`'datetime'`	`NaT` datetime value
`'duration'`	`0` seconds, as a duration value
`'calendarDuration'`	`0` days, as a calendar duration value
`'string'`	`""` (1-by-1 string with no characters)
`'cellstr'`	`{''}` (cell with 0-by-0 character array)
`'cell'`	`{[]}` (cell with 0-by-0 double array)
`'struct'`	Scalar structure with no fields
`'table'`	Table with no variables

For any other data type, the initial value is the value used by that type or class to "in-fill" unassigned elements of an array.

If you specify `'char'` as a data type, then `table` preallocates the corresponding variable as a cell array of character vectors, not as a character array. Best practice is to avoid creating table or timetable variables that are character arrays. When working with text data in a table or a timetable, consider using a string array or a categorical array.

`varNames` — Variable names
cell array of character vectors | string array

Variable names in the output table, specified as a cell array of character vectors or a string array, whose elements are nonempty and distinct. The number of names in `varNames` must equal the number of variables. The `table` function also stores the variable names in the `VariableNames` property of the table.

Variable names can have any Unicode characters, including spaces and non-ASCII characters.

`rowNames` — Row names
cell array of character vectors | string array

Row names in the output table, specified as a cell array of character vectors or a string array. The number of names in `rowNames` must equal the number of rows. The `table` function also stores the row names in the `RowNames` property of the table.

Example: `T = table([10;20;30],{'M';'F';'F'},'VariableNames',{'Age','Gender'},'RowNames',{'P1','P2','P3'})` creates a table with both variable names and row names.

Properties

Access Table Metadata Properties

A table contains metadata properties that describe the table and its variables. Access these properties using the syntax *`tableName`*`.Properties.`*`PropertyName`*, where *`PropertyName`* is the name of a property. For example, you can access the names of the variables in table `T` using the syntax `T.Properties.VariableNames`.

You can return a summary of all the metadata properties using the syntax *`tableName`*`.Properties`.

Tables provide metadata access through the `Properties` property because you can access table data directly using dot syntax. For example, if table `T` has a variable named `Var1`, then you can access the variable as an array using the syntax `T.Var1`.

Table Metadata

`DimensionNames` — Dimension names
`{'Row','Variables'}` (default) | two-element cell array of character vectors | two-element string array

Dimension names, specified as a two-element cell array of character vectors or a two-element string array.

You can access table data using the two dimension names. If the table has row names, and you use dot syntax and the first dimension name, then you can access the row names as a vector. If you use dot syntax and the second dimension name, then the data from all the variables are concatenated together in one array, as though you had indexed into the timetable using `{:,:}` syntax.

If you specify this property using a string array, then it is converted and stored as a cell array of character vectors.

Dimension names can have any Unicode characters, including spaces and non-ASCII characters.

Example

Create a table and display its dimension names. You can access row names and data using dimension names with dot syntax.

```
load patients
T = table(Age,Height,Weight,Systolic,Diastolic, ...
          'RowNames',LastName);
T.Properties.DimensionNames
```

```
ans = 1x2 cell array
    {'Row'}    {'Variables'}
```

Access the row names using the first dimension name. Display the first five names.

```
T.Row(1:5)
```

```
ans = 5x1 cell array
    {'Smith'   }
    {'Johnson' }
    {'Williams'}
    {'Jones'   }
    {'Brown'   }
```

Access the data using the second dimension name. This syntax is equivalent to `T{:,:}`.

```
T.Variables
```

```
ans = 100×5

    38    71   176   124    93
    43    69   163   109    77
    38    64   131   125    83
    40    67   133   117    75
    49    64   119   122    80
    46    68   142   121    70
    33    64   142   130    88
    40    68   180   115    82
    28    68   183   115    78
    31    66   132   118    86
      ⋮
```

Modify the names of its dimensions using the `Properties.DimensionNames` property. Having changed the dimension names, you can access the row times and data using the syntaxes `T.Patient` and `T.Data` respectively.

```
T.Properties.DimensionNames = {'Patient','Data'};
T.Properties
```

```
ans = 
  TableProperties with properties:

             Description: ''
                UserData: []
          DimensionNames: {'Patient'  'Data'}
           VariableNames: {'Age'  'Height'  'Weight'  'Systolic'  'Diastolic'}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                 RowNames: {100x1 cell}
        CustomProperties: No custom properties are set.
      Use addprop and rmprop to modify CustomProperties.
```

`RowNames` — Row names

`{}` (default) | cell array of character vectors | string array

Row names, specified as a cell array of character vectors or a string array, whose elements are nonempty and distinct. If `RowNames` is not empty, then the number of row names must equal the number of rows in the table. MATLAB removes any leading or trailing whitespace from the row names.

The row names are visible when you view the table. Furthermore, you can use the row names within parentheses or curly braces to access the table data.

Another way to access the row names is to use dot syntax and the name of the first dimension of the table.

If you specify this property using a string array, then it is converted and stored as a cell array of character vectors.

Example

Create a table. Then add row names and access rows by their names.

```
load patients
T = table(Gender,Age,Height,Weight,Smoker,Systolic,Diastolic);
```

Add row names using the `Properties.RowNames` property. By default, tables do not have row names, but you can add them at any time.

```
T.Properties.RowNames = LastName;
head(T,4)
```

```
ans=4×7 table
                  Gender      Age    Height    Weight    Smoker    Systolic    Diastoli
                _________     ___    ______    ______    ______    ________    ________

    Smith       {'Male'  }    38       71       176      true        124          93
    Johnson     {'Male'  }    43       69       163      false       109          77
    Williams    {'Female'}    38       64       131      false       125          83
    Jones       {'Female'}    40       67       133      false       117          75
```

Another way to access the row names is by using dot syntax with the name of the first dimension of the table. Display the first five row names.

```
T.Properties.DimensionNames
```

```
ans = 1x2 cell array
    {'Row'}    {'Variables'}
```

```
T.Row(1:5)
```

```
ans = 5x1 cell array
    {'Smith'   }
    {'Johnson' }
    {'Williams'}
    {'Jones'   }
    {'Brown'   }
```

Index into the table by row names.

```
T({'Smith','Williams'},:)
```

```
ans=2×7 table
                  Gender      Age    Height    Weight    Smoker    Systolic    Diastoli
                _________     ___    ______    ______    ______    ________    ________

    Smith       {'Male'  }    38       71       176      true        124          93
    Williams    {'Female'}    38       64       131      false       125          83
```

`Description` — Table description

`''` (default) | character vector | string scalar

Table description, specified as a character vector or string scalar. This description is visible when using the `summary` function.

If you specify this property using a string scalar, then it is converted and stored as a character vector.

Example

Create a table. Modify the description of the table. Display a summary of the result.

```
load patients
T = table(Gender,Age,Height,Weight);
T.Properties.Description = 'Simulated patient data';
summary(T)
```

```
Description:  Simulated patient data

Variables:

    Gender: 100x1 cell array of character vectors

    Age: 100x1 double

        Values:

            Min        25
            Median     39
            Max        50

    Height: 100x1 double

        Values:

            Min          60
            Median       67
            Max          72

    Weight: 100x1 double

        Values:

            Min          111
```

```
            Median     142.5
            Max          202
```

UserData — Additional table information

[ ] (default) | array

Additional table information, specified as an array. You can attach data of any kind to a table using this property.

Example

Create a table. Attach an anonymous function as a piece of user data that is associated with the table.

```
load patients
T = table(Gender,Age,Height,Weight,Smoker,Systolic,Diastolic);
formula = @(x) x.^2;
T.Properties.UserData = formula;
T.Properties
```

```
ans =
  TableProperties with properties:

             Description: ''
                UserData: @(x)x.^2
          DimensionNames: {'Row'  'Variables'}
           VariableNames: {1x7 cell}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                RowNames: {}
        CustomProperties: No custom properties are set.
      Use addprop and rmprop to modify CustomProperties.
```

Variable Metadata

VariableNames — Variable names

cell array of character vectors | string array

Variable names, specified as a cell array of character vectors or a string array, whose elements are nonempty and distinct. The number of names must equal the number of variables.

Variable names can have any Unicode characters, including spaces and non-ASCII characters.

The variable names are visible when viewing the table and when using the `summary` function. Furthermore, you can use the variable names within parentheses, within curly braces, or with dot indexing to access table data.

If you specify this property using a string array, then it is converted and stored as a cell array of character vectors.

Example

Create a table with default variable names. Then modify the names using the `Properties.VariableNames` property.

```
T = table({'M';'M';'F';'F';'F'},[38;43;38;40;49], ...
          [71;69;64;67;64],[176;163;131;133;119])
```

```
T=5×4 table
    Var1     Var2    Var3    Var4
    _____    ____    ____    ____

    {'M'}     38      71     176
    {'M'}     43      69     163
    {'F'}     38      64     131
    {'F'}     40      67     133
    {'F'}     49      64     119
```

```
T.Properties.VariableNames = {'Gender','Age','Height','Weight'}
```

```
T=5×4 table
    Gender    Age    Height    Weight
    ______    ___    ______    ______

    {'M'}     38       71       176
    {'M'}     43       69       163
    {'F'}     38       64       131
    {'F'}     40       67       133
    {'F'}     49       64       119
```

A fundamental way to display and modify variables is to access them by name using dot syntax.

```
T.Age
```

```
ans = 5×1

    38
    43
    38
    40
    49
```

```
T.Age(1) = 53
```

```
T=5×4 table
    Gender    Age    Height    Weight
    ______    ___    ______    ______

    {'M'}     53       71       176
    {'M'}     43       69       163
    {'F'}     38       64       131
    {'F'}     40       67       133
    {'F'}     49       64       119
```

`VariableDescriptions` — Variable descriptions

`{}` (default) | cell array of character vectors | string array

Variable descriptions, specified as a cell array of character vectors or a string array This property can be an empty cell array, which is the default. If the array is not empty, then it must contain as many elements as there are variables. You can specify an individual empty character vector or empty string for a variable that does not have a description.

The variable descriptions are visible when using the `summary` function.

If you specify this property using a string array, then it is converted and stored as a cell array of character vectors.

Example

Create a table. Modify the variable descriptions. Display a summary of the result.

```
load patients
T = table(Gender,Age,Height,Weight,Smoker,Systolic,Diastolic);
T.Properties.VariableDescriptions = {'','','','', ...
                                     'Has the patient ever been a smoker', ...
```

```
                                        'Systolic Pressure','Diastolic Pressure'};
summary(T)
Variables:

    Gender: 100x1 cell array of character vectors

    Age: 100x1 double

        Values:

            Min        25
            Median     39
            Max        50

    Height: 100x1 double

        Values:

            Min          60
            Median       67
            Max          72

    Weight: 100x1 double

        Values:

            Min          111
            Median     142.5
            Max          202

    Smoker: 100x1 logical

        Properties:
            Description:  Has the patient ever been a smoker
        Values:

            True        34
            False       66

    Systolic: 100x1 double

        Properties:
            Description:  Systolic Pressure
        Values:
```

```
            Min          109
            Median       122
            Max          138

    Diastolic: 100x1 double

        Properties:
            Description:  Diastolic Pressure
        Values:

            Min            68
            Median       81.5
            Max            99
```

`VariableUnits` — Variable units

`{}` (default) | cell array of character vectors | string array

Variable units, specified as a cell array of character vectors or a string array. This property can be an empty cell array, which is the default. If the array is not empty, then it must contain as many elements as there are variables. You can specify an individual empty character vector or empty string for a variable that does not have units.

The variable units are visible when using the `summary` function.

If you specify this property using a string array, then it is converted and stored as a cell array of character vectors.

Example

Create a table. Modify the variable units. Display a summary of the result.

```
load patients
T = table(Gender,Age,Height,Weight,Smoker,Systolic,Diastolic);
T.Properties.VariableUnits = {'','Yrs','In','Lbs','','mm Hg','mm Hg'};
summary(T)
```

```
Variables:

    Gender: 100x1 cell array of character vectors

    Age: 100x1 double

        Properties:
            Units:  Yrs
```

```
    Values:

        Min        25
        Median     39
        Max        50

Height: 100x1 double

    Properties:
        Units:  In
    Values:

        Min          60
        Median       67
        Max          72

Weight: 100x1 double

    Properties:
        Units:  Lbs
    Values:

        Min          111
        Median     142.5
        Max          202

Smoker: 100x1 logical

    Values:

        True        34
        False       66

Systolic: 100x1 double

    Properties:
        Units:  mm Hg
    Values:

        Min          109
        Median       122
        Max          138

Diastolic: 100x1 double
```

```
    Properties:
        Units:  mm Hg
    Values:

        Min            68
        Median       81.5
        Max            99
```

`VariableContinuity` — Status as continuous or discrete variables
`[ ]` (default) | cell array of character vectors | string array

Status as continuous or discrete variables, specified as a cell array of character vectors or a string array.

While tables and timetables both have this property, only timetables use it. For more information, see the `VariableContinuity` property of `timetable`.

Custom Metadata

`CustomProperties` — Customized metadata of table and its variables
`CustomProperties` object

Customized metadata of a table and its variables, specified as a `CustomProperties` object.

The `CustomProperties` object is a container for customized metadata that you can add to a table. By default, `CustomProperties` has zero properties. Each property you add to `CustomProperties` can contain either table metadata or variable metadata. If a property contains variable metadata, then its value must be an array, and the number of elements in the array must equal the number of table variables.

- To add properties for customized metadata to a table, use the `addprop` function.
- To access or modify customized metadata, use the syntax *`tableName`*`.Properties.CustomProperties.`*`PropertyName`*. In this syntax, *`PropertyName`* is the name you chose when you added that property using `addprop`.
- To remove properties, use the `rmprop` function.

Note: You can add or remove **only** properties for customized metadata using `addprop` and `rmprop`. You cannot add or remove properties of the *`tableName`*`.Properties` object.

Example

Create a table.

```
load patients
T = table(Gender,Age,Height,Weight,Smoker,Systolic,Diastolic);
```

Add properties that can hold customized metadata about the table and its variables. In this example, the metadata are names of instruments, `true` and `false` values indicating whether variables are to be plotted, and the name of an output file. To add properties, use the `addprop` function.

```
T = addprop(T,{'Instrument','ToPlot','OutputFile'},{'variable','variable','table'});
T.Properties
```

```
ans = 
  TableProperties with properties:

             Description: ''
                UserData: []
          DimensionNames: {'Row'   'Variables'}
           VariableNames: {1x7 cell}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                RowNames: {}

   Custom Properties (access using t.Properties.CustomProperties.<name>):
             OutputFile: []
             Instrument: []
                 ToPlot: []
```

Assign values to the customized metadata using dot syntax. When you assign an array of text values to customized metadata, the best practice is to use a string array, not a cell array of character vectors. If a property of `CustomProperties` is a cell array of character vectors, then there is no mechanism to prevent you from later assigning nontext values as elements of the cell array.

```
T.Properties.CustomProperties.Instrument = ["","","height rod","scale","","blood pressu
T.Properties.CustomProperties.ToPlot = [false false true true false true true];
T.Properties.CustomProperties.OutputFile = 'patientData.csv';
T.Properties
```

```
ans = 
  TableProperties with properties:
```

```
             Description: ''
                UserData: []
          DimensionNames: {'Row'  'Variables'}
           VariableNames: {1x7 cell}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                RowNames: {}

   Custom Properties (access using t.Properties.CustomProperties.<name>):
             OutputFile: 'patientData.csv'
             Instrument: [1x7 string]
                 ToPlot: [0 0 1 1 0 1 1]
```

Remove the `OutputFile` property from `T`.

```
T = rmprop(T,'OutputFile');
T.Properties
```

```
ans =
  TableProperties with properties:

             Description: ''
                UserData: []
          DimensionNames: {'Row'  'Variables'}
           VariableNames: {1x7 cell}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                RowNames: {}

   Custom Properties (access using t.Properties.CustomProperties.<name>):
             Instrument: [1x7 string]
                 ToPlot: [0 0 1 1 0 1 1]
```

Examples

Store Related Data Variables in Table

Store data about a group of patients in a table. You can perform calculations and store results in the same table. Also, you can annotate the table to describe your work and the variables of the table.

First, create workspace variables that have the patient data. The variables can have any data types but must have the same number of rows.

```
LastName = {'Sanchez';'Johnson';'Li';'Diaz';'Brown'};
Age = [38;43;38;40;49];
Smoker = logical([1;0;1;0;1]);
Height = [71;69;64;67;64];
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];
```

Create a table, `T`, as a container for the workspace variables. The `table` function uses the workspace variable names as the names of the table variables in `T`. A table variable can have multiple columns. For example, the `BloodPressure` variable in T is a 5-by-2 array.

```
T = table(LastName,Age,Smoker,Height,Weight,BloodPressure)
```

```
T=5×6 table
     LastName      Age    Smoker    Height    Weight    BloodPressure
    ___________    ___    ______    ______    ______    _____________

    {'Sanchez'}    38     true        71       176       124     93
    {'Johnson'}    43     false       69       163       109     77
    {'Li'     }    38     true        64       131       125     83
    {'Diaz'   }    40     false       67       133       117     75
    {'Brown'  }    49     true        64       119       122     80
```

You can use dot indexing to access table variables. For example, calculate the mean height of the patients using the values in `T.Height`.

```
meanHeight = mean(T.Height)
```

```
meanHeight = 67
```

Calculate body mass index (BMI), and add it as a new table variable. You also can add and name table variables in one step, using dot syntax.

```
T.BMI = (T.Weight*0.453592)./(T.Height*0.0254).^2
```

```
T=5×7 table
     LastName      Age    Smoker    Height    Weight    BloodPressure     BMI
    ___________    ___    ______    ______    ______    _____________    ______

    {'Sanchez'}    38     true        71       176       124     93      24.547
    {'Johnson'}    43     false       69       163       109     77      24.071
    {'Li'     }    38     true        64       131       125     83      22.486
    {'Diaz'   }    40     false       67       133       117     75      20.831
    {'Brown'  }    49     true        64       119       122     80      20.426
```

Annotate the table with a description of the BMI calculation. You can annotate `T` and its variables using metadata accessed through `T.Properties`.

```
T.Properties.Description = 'Patient data, including body mass index (BMI) calculated us
T.Properties
```

```
ans =
  TableProperties with properties:

             Description: 'Patient data, including body mass index (BMI) calculated usi
                UserData: []
          DimensionNames: {'Row'  'Variables'}
           VariableNames: {1x7 cell}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                RowNames: {}
        CustomProperties: No custom properties are set.
      Use addprop and rmprop to modify CustomProperties.
```

Access All Table Data as Matrix

Access all the data from a table as a matrix, using the name of the second dimension of the table.

Create a table that has five rows of data about a set of patients.

```
Age = [38;43;38;40;49];
Smoker = logical([1;0;1;0;1]);
Height = [71;69;64;67;64];
```

```
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];

T = table(Age,Smoker,Height,Weight,BloodPressure)
```

```
T=5×5 table
    Age    Smoker    Height    Weight    BloodPressure
    ___    ______    ______    ______    _____________

    38     true        71       176       124     93
    43     false       69       163       109     77
    38     true        64       131       125     83
    40     false       67       133       117     75
    49     true        64       119       122     80
```

Display the names of the table dimensions using the `DimensionNames` property. The default name of the second dimension is `Variables`.

```
T.Properties.DimensionNames
```

```
ans = 1x2 cell array
    {'Row'}    {'Variables'}
```

Access the table data as a matrix using the syntax `T.Variables`. This syntax is equivalent to accessing all the contents using curly brace syntax, `T{:,:}`. If the table data cannot be concatenated into a matrix, then an error message is raised.

```
T.Variables
```

```
ans = 5×6

    38     1    71   176   124    93
    43     0    69   163   109    77
    38     1    64   131   125    83
    40     0    67   133   117    75
    49     1    64   119   122    80
```

Rename the second dimension. If you change the name, then you can use the new name to access the data.

```
T.Properties.DimensionNames{2} = 'PatientData';
T.PatientData
```

```
ans = 5×6

    38     1    71   176   124    93
    43     0    69   163   109    77
    38     1    64   131   125    83
    40     0    67   133   117    75
    49     1    64   119   122    80
```

Specify Size and Variable Types

Preallocate a table by specifying its size and the data types of the variables. The `table` function fills the variables with default values that are appropriate for the data types you specify. It also gives the variables default names, but you also can assign variable names of your own. Preallocation provides room for data you add to the table later.

```
sz = [4 3];
varTypes = {'double','datetime','string'};
T = table('Size',sz,'VariableTypes',varTypes)
```

```
T=4×3 table
    Var1    Var2      Var3
    ____    ____    _________

     0      NaT     <missing>
     0      NaT     <missing>
     0      NaT     <missing>
     0      NaT     <missing>
```

To specify names for the variables, use the `'VariableNames'` name-value pair argument.

```
varNames = {'Temperature','Time','Station'};
T2 = table('Size',sz,'VariableTypes',varTypes,'VariableNames',varNames)
```

```
T2=4×3 table
    Temperature    Time     Station
    ___________    ____    _________

         0         NaT     <missing>
         0         NaT     <missing>
```

```
         0         NaT     <missing>
         0         NaT     <missing>
```

Add rows of data to the first two rows of `T2`. Preallocation can be a useful technique when your code adds one row of data, or a few rows of data, at a time. Instead of growing the table every time you add a row, you can fill in table variables that already have room for your data.

```
T2(1,:) = {75,datetime('now'),"S1"};
T2(2,:) = {68,datetime('now')+1,"S2"}
```

```
T2=4×3 table
    Temperature            Time             Station
    ___________    ____________________    _________

        75         27-Aug-2019 00:01:36    "S1"
        68         28-Aug-2019 00:01:36    "S2"
         0                           NaT    <missing>
         0                           NaT    <missing>
```

You can encapsulate a row of data values in a cell array. When you assign a row from a cell array, elements from the cell array are assigned to the row in the table.

Specify Variable Names

Create a table from arrays. To specify table variable names, use the `'VariableNames'` name-value pair argument. For example, you can use `'VariableNames'` to specify names when the other input arguments are not workspace variables.

```
T = table(categorical({'M';'F';'M'}),[45;32;34],...
          {'NY';'CA';'MA'},logical([1;0;0]),...
          'VariableNames',{'Gender','Age','State','Vote'})
```

```
T=3×4 table
    Gender    Age    State     Vote
    ______    ___    ______    _____

      M       45     {'NY'}    true
      F       32     {'CA'}    false
```

```
      M       34     {'MA'}    false
```

Create a table with the state names as row names. You can specify both the `'VariableNames'` and `'RowNames'` name-value pairs when using the `table` function.

```
T = table(categorical({'M';'F';'M'}),[45;32;34],logical([1;0;0]),...
          'VariableNames',{'Gender','Age','Vote'},...
          'RowNames',{'NY';'CA';'MA'})
```

```
T=3×3 table
          Gender    Age    Vote
          ______    ___    _____

    NY      M       45     true
    CA      F       32     false
    MA      M       34     false
```

Specify Row Names

Specify row names for a table. Tables do not have to have row names, but if you specify them, then you can index into a table by row name. You also can access the set of row names using the name of the first dimension of a table.

Create arrays containing patient data.

```
LastName = {'Sanchez';'Johnson';'Lee';'Diaz';'Brown'};
Age = [38;43;38;40;49];
Height = [71;69;64;67;64];
Weight = [176;163;131;133;119];
```

Create a table containing the arrays. Specify `LastName` as the source of row names for the table. The table has only three variables. The row names are not a table variable, but instead a property of the table.

```
T = table(Age,Weight,Height,'RowNames',LastName)
```

```
T=5×3 table
                Age    Weight    Height
                ___    ______    ______
```

```
    Sanchez    38      176        71
    Johnson    43      163        69
    Lee        38      131        64
    Diaz       40      133        67
    Brown      49      119        64
```

Since the rows have row names, you can index into the rows of T by name.

```
T('Lee',:)
```

```
ans=1×3 table
           Age    Weight    Height
           ___    ______    ______

    Lee    38      131        64
```

To specify multiple rows, use a cell array.

```
T({'Lee','Brown'},:)
```

```
ans=2×3 table
             Age    Weight    Height
             ___    ______    ______

    Lee      38      131        64
    Brown    49      119        64
```

To access all the row names of T as a cell array, use the syntax T.Row. By default, Row is the name of the first dimension of a table.

```
T.Row
```

```
ans = 5x1 cell array
    {'Sanchez'}
    {'Johnson'}
    {'Lee'    }
    {'Diaz'   }
    {'Brown'  }
```

Change the name of the first dimension. If you change the name, then you can access the row names using the new name.

```
T.Properties.DimensionNames{1} = 'LastNames';
T.LastNames
```

```
ans = 5x1 cell array
    {'Sanchez'}
    {'Johnson'}
    {'Lee'    }
    {'Diaz'   }
    {'Brown'  }
```

Specify Table Variables and Row Names Using String Arrays

Starting in R2017a, you can create strings using double quotes, and add string arrays as table variables.

```
FlightNum = [1261;547;3489];
Customer = ["Jones";"Brown";"Smith"];
Date = datetime(2016,12,20:22)';
Rating = categorical(["Good";"Poor";"Fair"]);
Comment = ["Flight left on time, not crowded";...
           "Late departure, ran out of dinner options";...
           "Late, but only by half an hour. Otherwise fine."];
T = table(FlightNum,Customer,Date,Rating,Comment)
```

```
T=3×5 table
    FlightNum    Customer       Date        Rating                          Comment
    _________    ________    ___________    ______    ___________________________________

      1261       "Jones"     20-Dec-2016     Good     "Flight left on time, not crowdec
       547       "Brown"     21-Dec-2016     Poor     "Late departure, ran out of dinne
      3489       "Smith"     22-Dec-2016     Fair     "Late, but only by half an hour.
```

To use the text in a string array as row names, convert the string array to a cell array of character vectors. Then create a table with row names.

```
Customer = cellstr(Customer);
T = table(FlightNum,Date,Rating,Comment,'RowNames',Customer)
```

```
T=3×4 table
             FlightNum       Date        Rating                          Comment
```

```
                _________    ____________    ______    ____________________________________

    Jones        1261        20-Dec-2016     Good     "Flight left on time, not crowded"
    Brown         547        21-Dec-2016     Poor     "Late departure, ran out of dinner c
    Smith        3489        22-Dec-2016     Fair     "Late, but only by half an hour. Oth
```

Build Table by Assigning Variables Individually

Create workspace variables containing snowfall totals on different dates at three locations. These variables are row vectors.

```
Date = {'12/25/11','1/2/12','1/23/12','2/7/12','2/15/12'};
location1 = [20 5 13 0 17];
location2 = [18 9 21 5 12];
location3 = [26 10 16 3 15];
```

One way to create a table from these variables is to call the `table` function with the syntax `T = table(Date',location1',location2',location3')`. Because the workspace variables are row vectors, you must transpose them to put them into the table as column-oriented data. Therefore, the input arguments are expressions, not simple variables. As a result, `table` creates `T` with the default variable names `Var1`, `Var2`, `Var3`, and `Var4`. You can assign more meaningful names to `T.Properties.VariableNames` after you create `T`. But, it might be more convenient to create an empty table, and then add variables one at a time with new names.

Create an empty table. Transpose the workspace variables and add them to the table as column vectors. As part of assigning each workspace variable into `T`, provide a meaningful name for the table variable.

```
T = table;
T.Date = Date';
T.Natick = location1';
T.Boston = location2';
T.Worcester = location3'
```

```
T=5×4 table
        Date        Natick    Boston    Worcester
    ____________    ______    ______    _________

    {'12/25/11'}      20        18         26
```

```
    {'1/2/12'  }       5         9         10
    {'1/23/12' }      13        21         16
    {'2/7/12'  }       0         5          3
    {'2/15/12' }      17        12         15
```

Specify Variable Names Using Any Characters

Starting in R2019b, you can specify table variable names that are not valid MATLAB® identifiers. Such variable names can include spaces, non-ASCII characters, and can have any character as the leading character. When you access such a variable name, enclose it quotation marks.

Create a table that stores data about a set of patients. Start with their ages and status as smokers.

```
Age = [38;43;38;40;49];
Smoker = logical([1;0;1;0;1]);
Height = [71;69;64;67;64];
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];
T = table(Age,Smoker)
```

```
T=5×2 table
    Age    Smoker
    ___    ______

    38     true
    43     false
    38     true
    40     false
    49     true
```

Add the blood pressure readings with the variable name `'29-May-2019 Blood Pressure Reading'`. You can use dot syntax to add or access the variable. Since its name is not a valid MATLAB identifier, use parentheses and quotation marks with dot syntax.

```
T.('29-May-2019 Blood Pressure Reading') = BloodPressure
```

```
T=5×3 table
    Age    Smoker    29-May-2019 Blood Pressure Reading
```

```
    ___    ______    ____________________________________

    38     true                  124     93
    43     false                 109     77
    38     true                  125     83
    40     false                 117     75
    49     true                  122     80
```

When table variable names are valid MATAB identifiers, you can use dot syntax without parentheses and quotation marks.

```
T.Height = Height;
T.Weight = Weight
```

```
T=5×5 table
    Age    Smoker    29-May-2019 Blood Pressure Reading    Height    Weight
    ___    ______    ____________________________________    ______    ______

    38     true                  124     93                  71        176
    43     false                 109     77                  69        163
    38     true                  125     83                  64        131
    40     false                 117     75                  67        133
    49     true                  122     80                  64        119
```

Index into T using variable names.

```
T(:,{'Age','Smoker','29-May-2019 Blood Pressure Reading'})
```

```
ans=5×3 table
    Age    Smoker    29-May-2019 Blood Pressure Reading
    ___    ______    ____________________________________

    38     true                  124     93
    43     false                 109     77
    38     true                  125     83
    40     false                 117     75
    49     true                  122     80
```

Limitations

- Use single quotes for the input names `'Size'`, `'VariableTypes'`, `'VariableNames'`, and `'RowNames'`. To avoid confusion with variable inputs, do not use double-quoted string scalars (such as `"Size"`) for these names.

Tips

- For a list of functions that accept or return tables, see “Tables”.

Compatibility Considerations

Variable names can contain leading and trailing whitespace characters

Behavior changed in R2019b

Table and timetable variable names with leading or trailing whitespace characters are not modified.

In previous releases, leading and trailing whitespace characters were deleted from variable names when you specified them using the `'VariableNames'` name-value pair argument, or assigned them to the `VariableNames` property.

To manually remove such characters, first use the `strtrim` function on the names, then assign them as variable names to the table or timetable.

Variable names and dimension names must be unique

Behavior changed in R2019b

MATLAB raises an error if you assign a table variable name that matches a dimension name, or a dimension name that matches a variable name. In previous releases, MATLAB raised a warning and modified the names so they were unique.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitation:

- The syntax `TT = table(T1,T2,...)` constructs a tall table from several tall arrays `(T1,T2,...)`. You can use the `'VariableNames'` name-value pair argument to specify variable names.

For more information, see “Tall Arrays for Out-of-Memory Data”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Starting in R2019b, you can use tables in MATLAB code intended for code generation. For more information, see “Code Generation for Tables” (MATLAB Coder) and “Table Limitations for Code Generation” (MATLAB Coder).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- The input arrays must be distributed and have the same number of rows.
- The result is distributed, using a 1D distribution scheme over the first dimension.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`array2table` | `cell2table` | `isvarname` | `readtable` | `struct2table` | `summary` | `tall` | `uitable`

Topics

"Create and Work with Tables"
"Modify Units, Descriptions, and Table Variable Names"
"Access Data in Tables"
"Calculations on Tables"

Introduced in R2013b

join

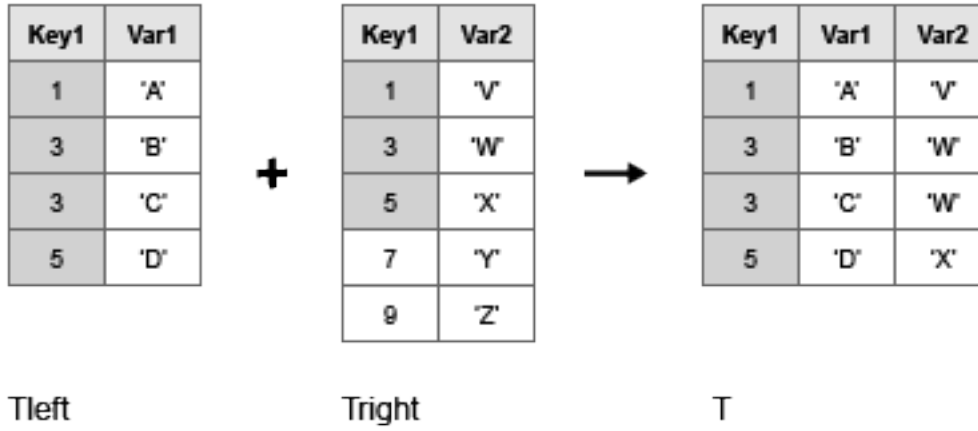
Combine two tables or timetables by rows using key variables

Syntax

```
T = join(Tleft,Tright)
T = join(Tleft,Tright,Name,Value)
[T,iright] = join( ___ )
```

Description

`T = join(Tleft,Tright)` combines tables or timetables `Tleft` and `Tright` using key variables on page 1-15143. All variables with the same names in both tables are key variables. A table join appends rows from the right table where its key variables match values in the key variables of the left table. For example, if `Tleft` has variables named `Key1` and `Var1`, and `Tright` has variables `Key1` and `Var2`, then `T=join(Tleft,Tright)` uses `Key1` as a key variable.



The matching values of the key variables in the left and right tables do not have to be in the same order. Also, the key variables of `Tright` must contain all values in the key variables of `Tleft`. Each value must occur only once in the key variables of `Tright`, but can occur multiple times in the key variables of `Tleft`.

The inputs can be tables, timetables, or one of each.

- If `Tleft` is a table, then `join` returns `T` as a table.
- If `Tleft` is a timetable, then `join` returns `T` as a timetable.

The vectors of row labels of `Tleft` and `Tright` can be key variables. Row labels are the row names of a table, or the row times of a timetable.

`T = join(Tleft,Tright,Name,Value)` joins the tables or timetables with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify which variables to use as key variables.

`[T,iright] = join( ___ )` also returns an index vector `iright` such that each element of `iright` identifies the row in `Tright` that corresponds to that row in `T`. You can use this syntax with any of the input arguments of the previous syntaxes.

Examples

Append Values from One Table to Another

Create a table, `Tleft`.

```
Tleft = table({'Janice','Jonas','Javier','Jerry','Julie'}',[1;2;1;2;1],...
        'VariableNames',{'Employee' 'Department'})
```

```
Tleft=5×2 table
     Employee     Department
    __________    __________

    {'Janice'}        1
    {'Jonas' }        2
    {'Javier'}        1
    {'Jerry' }        2
    {'Julie' }        1
```

Create a table, `Tright`, with a variable in common with `Tleft`.

```
Tright = table([1 2]',{'Mary' 'Mona'}',...
         'VariableNames',{'Department' 'Manager'})
```

```
Tright=2×2 table
    Department    Manager
    __________    ________

        1         {'Mary'}
        2         {'Mona'}
```

Create a new table, `T`, containing data from tables `Tleft` and `Tright`. Use the `join` function to repeat and append `Manager` data from table `Tright` to the data from table `Tleft`, based on the key variable, `Department`.

```
T = join(Tleft,Tright)
```

```
T=5×3 table
     Employee     Department    Manager
    __________    __________    ________
```

```
    {'Janice'}        1         {'Mary'}
    {'Jonas' }        2         {'Mona'}
    {'Javier'}        1         {'Mary'}
    {'Jerry' }        2         {'Mona'}
    {'Julie' }        1         {'Mary'}
```

Merge Tables with One Variable in Common

Create a table, `Tleft`.

```
Tleft = table([5;12;23;2;6],...
        {'cereal';'pizza';'salmon';'cookies';'pizza'},...
        'VariableNames',{'Age','FavoriteFood'},...
        'RowNames',{'Amy','Bobby','Holly','Harry','Sally'})
```

```
Tleft=5×2 table
             Age    FavoriteFood
             ___    ____________

    Amy       5     {'cereal' }
    Bobby    12     {'pizza'  }
    Holly    23     {'salmon' }
    Harry     2     {'cookies'}
    Sally     6     {'pizza'  }
```

Create a table, `Tright`, with one variable in common with `Tleft`.

```
Tright = table({'cereal';'cookies';'pizza';'salmon';'cake'},...
         [110;160;140;367;243],...
         {'B';'D';'B-';'A';'C-'},...
         'VariableNames',{'FavoriteFood','Calories','NutritionGrade'})
```

```
Tright=5×3 table
    FavoriteFood    Calories    NutritionGrade
    ____________    ________    ______________

    {'cereal' }       110           {'B' }
    {'cookies'}       160           {'D' }
    {'pizza'  }       140           {'B-'}
    {'salmon' }       367           {'A' }
```

```
    {'cake'   }       243           {'C-'}
```

Create a new table, T, with data from tables `Tleft` and `Tright`. The variable in common, `FavoriteFood`, is used as a key variable by the `join` function.

```
T = join(Tleft,Tright)
```

```
T=5×4 table
             Age    FavoriteFood    Calories    NutritionGrade
             ___    ____________    ________    ______________

    Amy       5     {'cereal' }       110           {'B' }
    Bobby    12     {'pizza'  }       140           {'B-'}
    Holly    23     {'salmon' }       367           {'A' }
    Harry     2     {'cookies'}       160           {'D' }
    Sally     6     {'pizza'  }       140           {'B-'}
```

Table T does not include information from the last row of table `Tright` about `'cake'` because there is no corresponding entry in table `Tleft`.

Merge Tables by Specifying One Key Variable

Create a table, `Tleft`.

```
Tleft = table([10;4;2;3;7],[5;4;9;6;1],[10;3;8;8;4])
```

```
Tleft=5×3 table
    Var1    Var2    Var3
    ____    ____    ____

     10      5       10
      4      4        3
      2      9        8
      3      6        8
      7      1        4
```

Create a table, `Tright`, giving `Var2` of table `Tright` the same contents as `Var2` from table `Tleft`.

```
Tright = table([6;1;1;6;8],[5;4;9;6;1])
```

```
Tright=5×2 table
    Var1    Var2
    ____    ____

     6       5
     1       4
     1       9
     6       6
     8       1
```

Create a new table, `T`, containing data from tables `Tleft` and `Tright`. Use `Var2` in tables `Tleft` and `Tright` as the key variable to the `join` function.

```
T = join(Tleft,Tright,'Keys','Var2')
```

```
T=5×4 table
    Var1_Tleft    Var2    Var3    Var1_Tright
    __________    ____    ____    ___________

        10         5       10          6
         4         4        3          1
         2         9        8          1
         3         6        8          6
         7         1        4          8
```

`join` adds a unique suffix to the nonkey variable, `Var1`, to distinguish the data from tables `Tleft` and `Tright`.

Keep One Copy of Nonkey Variables

Create a new table with data from tables `Tleft` and `Tright`. If any nonkey variables have the same name in both tables, keep only the copy from table `Tleft`.

Create a table, `Tleft`.

```
Tleft = table([10;4;2;3;7],[5;4;9;6;1])
```

```
Tleft=5×2 table
    Var1    Var2
    ____    ____
```

```
    10      5
     4      4
     2      9
     3      6
     7      1
```

Create a table, `Tright`, giving `Var2` of table `Tright` the same contents as `Var2` from table `Tleft`.

```
Tright = table([6;1;1;6;8],[5;4;9;6;1],[10;3;8;8;4])
```

```
Tright=5×3 table
    Var1    Var2    Var3
    ____    ____    ____

     6       5       10
     1       4        3
     1       9        8
     6       6        8
     8       1        4
```

Create a new table, `T`, with data from tables `Tleft` and `Tright`. Use `Var2` as a key variable to the `join` function and keep only the copy of `Var1` from table `Tleft`. The output table `T` does not contain the `Var1` data from table `Tright`.

```
T = join(Tleft,Tright,'Keys','Var2','KeepOneCopy','Var1')
```

```
T=5×3 table
    Var1    Var2    Var3
    ____    ____    ____

     10      5       10
      4      4        3
      2      9        8
      3      6        8
      7      1        4
```

Merge Tables Using Row Names as Keys

Create a table, `Tleft`.

```
Tleft = table(['M';'M';'F';'F';'F'],[38;43;38;40;49],...
        'VariableNames',{'Gender' 'Age'},...
        'RowNames',{'Smith' 'Johnson' 'Williams' 'Jones' 'Brown'})
```

```
Tleft=5×2 table
                Gender    Age
                ______    ___

    Smith         M       38
    Johnson       M       43
    Williams      F       38
    Jones         F       40
    Brown         F       49
```

Create a table, `Tright`, such that the rows of `Tleft` and the rows of `Tright` have a one-to-one correspondence.

```
Tright = table([64;69;67;71;64],...
         [119;163;133;176;131],...
         [122 80; 109 77; 117 75; 124 93; 125 83],...
         'VariableNames',{'Height' 'Weight' 'BloodPressure'},...
         'RowNames',{'Brown' 'Johnson' 'Jones' 'Smith' 'Williams'})
```

```
Tright=5×3 table
                Height    Weight    BloodPressure
                ______    ______    _____________

    Brown         64       119       122     80
    Johnson       69       163       109     77
    Jones         67       133       117     75
    Smith         71       176       124     93
    Williams      64       131       125     83
```

Create a new table, `T`, with data from tables `Tleft` and `Tright`. Use the vectors of row names as key variables. (The name of the vector of row names of a table is `'Row'`, as shown by `Tleft.Properties.DimensionNames{1}`.)

```
T = join(Tleft,Tright,'Keys','Row')
```

```
T=5×5 table
                Gender    Age    Height    Weight    BloodPressure
                ______    ___    ______    ______    _____________

    Smith         M       38       71       176       124     93
    Johnson       M       43       69       163       109     77
    Williams      F       38       64       131       125     83
    Jones         F       40       67       133       117     75
    Brown         F       49       64       119       122     80
```

The rows of T are in the same order as `Tleft`.

Merge Tables Using Left and Right Keys

Create a table, `Tleft`.

```
Tleft = table([10;4;2;3;7],[5;4;9;6;1],[10;3;8;8;4])
```

```
Tleft=5×3 table
    Var1    Var2    Var3
    ____    ____    ____

     10      5       10
      4      4        3
      2      9        8
      3      6        8
      7      1        4
```

Create a table, `Tright`, giving `Var2` of table `Tright` the same contents as `Var1` from table `Tleft`, but in a different order.

```
Tright = table([6;1;1;6;8],[2;3;4;7;10])
```

```
Tright=5×2 table
    Var1    Var2
    ____    ____

     6        2
     1        3
     1        4
```

```
     6        7
     8       10
```

Create a new table, `T`, containing data from tables `Tleft` and `Tright`. Use `Var1` from table `Tleft` with `Var2` from table `Tright` as key variables to the `join` function.

```
[T,iright] = join(Tleft,Tright,'LeftKeys',1,'RightKeys',2)
```

```
T=5×4 table
    Var1_Tleft    Var2    Var3    Var1_Tright
    __________    ____    ____    ___________

        10         5       10          8
         4         4        3          1
         2         9        8          6
         3         6        8          1
         7         1        4          6


iright = 5×1

     5
     3
     1
     2
     4
```

`T` is the horizontal concatenation of `Tleft` and `Tright(iright,1)`.

Merge Timetables

Create two timetables that have the same row times but different variables.

```
Traffic = [0.8 0.9 0.1 0.7 0.9]';
Noise = [0 1 1.5 2 2.3]';
Tleft = timetable(hours(1:5)',Traffic,Noise)
```

```
Tleft=5×3 timetable
    Time    Traffic    Noise
    ____    _______    _____
```

```
    1 hr      0.8         0
    2 hr      0.9         1
    3 hr      0.1       1.5
    4 hr      0.7         2
    5 hr      0.9       2.3
```

```
Distance = [0.88 0.86 0.91 0.9 0.86]';
Tright = timetable(hours(1:5)',Distance)
```

```
Tright=5×2 timetable
    Time    Distance
    ____    ________

    1 hr      0.88
    2 hr      0.86
    3 hr      0.91
    4 hr       0.9
    5 hr      0.86
```

Merge the timetables. `join` uses the row times as the key variables.

```
T = join(Tleft,Tright)
```

```
T=5×4 timetable
    Time    Traffic    Noise    Distance
    ____    _______    _____    ________

    1 hr      0.8         0       0.88
    2 hr      0.9         1       0.86
    3 hr      0.1       1.5       0.91
    4 hr      0.7         2        0.9
    5 hr      0.9       2.3       0.86
```

Merge Timetable and Table

Create a timetable and a table.

```
Measurements = [0.13 0.22 0.31 0.42 0.53 0.57 0.67 0.81 0.90 1.00]';
Device = ['A';'B';'A';'B';'A';'B';'A';'B';'A';'B'];
Tleft = timetable(seconds(1:10)',Measurements,Device)
```

```
Tleft=10×3 timetable
     Time     Measurements    Device
    ______    ____________    ______

    1 sec         0.13          A
    2 sec         0.22          B
    3 sec         0.31          A
    4 sec         0.42          B
    5 sec         0.53          A
    6 sec         0.57          B
    7 sec         0.67          A
    8 sec         0.81          B
    9 sec          0.9          A
    10 sec           1          B
```

```
Device = ['A';'B'];
Accuracy = [0.023;0.037];
Tright = table(Device,Accuracy)
```

```
Tright=2×2 table
    Device    Accuracy
    ______    ________

      A        0.023
      B        0.037
```

Merge the timetable and table. `Device` is the key variable because both `Tleft` and `Tright` have a variable with that name. `T` is a timetable.

```
T = join(Tleft,Tright)
```

```
T=10×4 timetable
     Time     Measurements    Device    Accuracy
    ______    ____________    ______    ________

    1 sec         0.13          A        0.023
    2 sec         0.22          B        0.037
    3 sec         0.31          A        0.023
    4 sec         0.42          B        0.037
```

```
    5 sec         0.53          A        0.023
    6 sec         0.57          B        0.037
    7 sec         0.67          A        0.023
    8 sec         0.81          B        0.037
    9 sec          0.9          A        0.023
    10 sec           1          B        0.037
```

Input Arguments

`Tleft` — Left table
table | timetable

Left table, specified as a table or a timetable. For all key variables, each row of `Tleft` must match exactly one row in `Tright`.

`Tright` — Right table
table | timetable

Right table, specified as a table or a timetable. For all key variables, each row of `Tright` must match exactly one row in `Tleft`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Keys',2` uses the second variable in `Tleft` and the second variable in `Tright` as key variables.

Keys — Variables to use as keys
positive integer | vector of positive integers | character vector | cell array of character vectors | string array | logical vector

Variables to use as keys, specified as the comma-separated pair consisting of `'Keys'` and a positive integer, vector of positive integers, character vector, cell array of character vectors, string array, logical vector, or `'RowNames'`.

You cannot use the `'Keys'` name-value pair argument with the `'LeftKeys'` and `'RightKeys'` name-value pair arguments.

A vector of row labels can be a key variable. Row labels are the row names of a table or the row times of a timetable. You can specify `'Keys'` as the row labels only, as one or more variables in `Tleft` and `Tright`, or as row labels and variables together.

Example: `'Keys',[1 3]` uses the first and third variables from `Tleft` and `Tright` as key variables.

Example: `'Keys',{'X','Y'}` uses the variables named `X` and `Y` in `Tleft` and `Tright` as key variables.

Example: `'Keys','Row'` uses the vectors of row names of `Tleft` and `Tright` as key variables, if both `Tleft` and `Tright` are tables with row names.

`LeftKeys` — Variables to use as keys in `Tleft`

positive integer | vector of positive integers | character vector | cell array of character vectors | string array | logical vector

Variables to use as keys in `Tleft`, specified as the comma-separated pair consisting of `'LeftKeys'` and a positive integer, vector of positive integers, character vector, cell array of character vectors, string array, or logical vector.

You must use the `'LeftKeys'` name-value pair argument in conjunction with the `'RightKeys'` name-value pair argument. `'LeftKeys'` and `'RightKeys'` both must specify the same number of key variables. `join` pairs key values in `Tleft` and `Tright` based on their order.

A vector of row labels can be a key variable. Row labels are the row names of a table or the row times of a timetable. You can specify `'LeftKeys'` as the row labels of `Tleft`, as one or more variables in `Tleft`, or as row labels and variables together.

Example: `'LeftKeys',1` uses only the first variable in `Tleft` as a key variable.

`RightKeys` — Variables to use as keys in `Tright`

positive integer | vector of positive integers | character vector | cell array of character vectors | string array | logical vector

Variables to use as keys in `Tright`, specified as the comma-separated pair consisting of `'RightKeys'` and a positive integer, vector of positive integers, character vector, cell array of character vectors, string array, or logical vector.

You must use the `'RightKeys'` name-value pair argument in conjunction with the `'LeftKeys'` name-value pair argument. `'LeftKeys'` and `'RightKeys'` both must specify the same number of key variables. `join` pairs key values in `Tleft` and `Tright` based on their order.

A vector of row labels can be a key variable. Row labels are the row names of a table or the row times of a timetable. You can specify `'RightKeys'` as the row labels of `Tright`, as one or more variables in `Tright`, or as row labels and variables together.

Example: `'RightKeys',3` uses only the third variable in `Tright` as a key variable.

`LeftVariables` — Variables from `Tleft` to include in `T`
positive integer | vector of positive integers | character vector | cell array of character vectors | string array | logical vector

Variables from `Tleft` to include in `T`, specified as the comma-separated pair consisting of `'LeftVariables'` and a positive integer, vector of positive integers, character vector, cell array of character vectors, string array, or logical vector.

You can use `'LeftVariables'` to include or exclude key variables, as well as nonkey variables, from `T`. However, you cannot include row names or row times from `Tleft`, because they are not variables.

By default, `join` includes all variables from `Tleft`.

`RightVariables` — Variables from `Tright` to include in `T`
positive integer | vector of positive integers | character vector | cell array of character vectors | string array | logical vector

Variables from `Tright` to include in `T`, specified as the comma-separated pair consisting of `'RightVariables'` and a positive integer, vector of positive integers, character vector, cell array of character vectors, string array, or logical vector.

You can use `'RightVariables'` to include or exclude key variables, as well as nonkey variables, from `T`. However, you cannot include row names or row times from `Tright`, because they are not variables.

By default, `join` includes all variables from `Tright` except the key variables.

`KeepOneCopy` — Variables for which `join` retains only the copy from `Tleft`
character vector | cell array of character vectors | string array

Variables for which `join` retains only the copy from `Tleft`, specified as the comma-separated pair consisting of `'KeepOneCopy'` and a character vector, cell array of character vectors, or string array that specifies variable names.

Key variables appear once in `T`, but if nonkey variables with identical names occur in `Tleft` and `Tright`, then `join` retains both copies in `T` by default. Use the `'KeepOneCopy'` name-value pair to retain only the copy from `Tleft`.

Example: `'KeepOneCopy',Var2` keeps only the copy from `Tleft` of the nonkey variable `Var2`.

Output Arguments

`T` — Merged data from `Tleft` and `Tright`
table | timetable

Merged data from `Tleft` and `Tright`, returned as a table or a timetable. The table, `T`, contains one row for each row in `Tleft`, appearing in the same order.

`join` creates `T` by horizontally concatenating `Tleft(:,LeftVars)` and `Tright(iright,RightVars)`. By default, `LeftVars` is all the variables of `Tleft`, and `RightVars` is all the nonkey variables from `Tright`. Otherwise, `LeftVars` consists of the variables specified by the `'LeftVariables'` name-value pair argument, and `RightVars` consists of the variables specified by the `'RightVariables'` name-value pair argument.

If `Tleft` and `Tright` contain nonkey variables with the same name, `join` adds a unique suffix to the corresponding variable names in `T`, unless you specify the `'KeepOneCopy'` name-value pair argument.

If `Tleft` is a table, then `T` is also a table. If `Tleft` is a timetable and `Tright` is either a timetable or a table, then `T` is a timetable.

You can store additional metadata in `T`, such as descriptions, variable units, variable names, and row names. For more information, see the Properties sections of `table` or `timetable`.

`iright` — Index to `Tright`
column vector

Index to `Tright`, returned as a column vector. Each element of `iright` identifies the row in `Tright` that corresponds to that row in the output table or timetable, `T`.

More About

Key Variable

Variable used to match and combine data between input tables `Tleft` and `Tright`.

Key Value

Value in a key variable of `Tleft` or `Tright`.

Algorithms

The `join` function first finds one or more key variables. Then, `join` uses the key variables to find the row in input table `Tright` that matches each row in input table `Tleft`, and combines those rows to create a row in output table `T`.

- If there is a one-to-one mapping between key values in `Tleft` and `Tright`, then `join` sorts the data in `Tright` and appends it to table `Tleft`.
- If there is a many-to-one mapping between key values in `Tleft` and `Tright`, then `join` sorts and repeats the data in `Tright` before appending it to table `Tleft`.
- If there is data in a key variable of `Tright` that does not map to a key value in `Tleft`, then `join` does not include that data in the output table, `T`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- To join a tall timetable and a tall table, the timetable must be the first input to `join`.

For more information, see “Tall Arrays”.

See Also

Join Tables | `innerjoin` | `outerjoin`

Introduced in R2013b

table2array

Convert table to homogeneous array

Syntax

```
A = table2array(T)
```

Description

`A = table2array(T)` converts the table, `T`, to a homogeneous array, `A`.

Examples

Convert Table of Numeric Data to Array

Create a table, `T`, consisting of numeric data.

```
T = table([1;2;3],[2 8; 4 10; 6 12],[3 12 21; 6 15 24; 9 18 27],...
    'VariableNames',{'One' 'Two' 'Three'})
```

```
T=3×3 table
    One      Two          Three
    ___    _______    _____________

     1      2     8     3    12    21
     2      4    10     6    15    24
     3      6    12     9    18    27
```

Convert table, `T`, to an array.

```
A = table2array(T)
```

```
A = 3×6
```

```
     1     2     8     3    12    21
     2     4    10     6    15    24
     3     6    12     9    18    27
```

A contains two columns from variable Two and three columns from variable Three.

Convert Numeric Subset of Table to Array

Define the numeric subset of a table to convert to an array.

Create a table with nonnumeric data in the first variable.

```
T = table(categorical({'M';'M';'F';'F';'F'}),[38;43;38;40;49],...
    [71;69;64;67;64],[176;163;131;133;119],...
    'VariableNames',{'Gender' 'Age' 'Height' 'Weight'})
```

```
T=5×4 table
    Gender    Age    Height    Weight
    ______    ___    ______    ______

      M       38       71       176
      M       43       69       163
      F       38       64       131
      F       40       67       133
      F       49       64       119
```

Convert T(:,2:4) to an array.

```
A = table2array(T(:,2:4))
```

```
A = 5×3

    38    71   176
    43    69   163
    38    64   131
    40    67   133
    49    64   119
```

A does not include data from the variable Gender.

Convert Table with Three-Dimensional Variables to Array

Create a table, T, with two rows and three variables where each variable has three dimensions.

```
T = table(ones(2,1,3),2*ones(2,2,3),3*ones(2,3,3),...
    'VariableNames',{'One' 'Two' 'Three'})
```

```
T=2×3 table
         One                Two                Three
    ______________    ______________    ______________

    [1x1x3 double]    [1x2x3 double]    [1x3x3 double]
    [1x1x3 double]    [1x2x3 double]    [1x3x3 double]
```

The size of the table is 2-by-3.

Convert table T to an array.

```
A = table2array(T)
```

```
A =
A(:,:,1) =

     1     2     2     3     3     3
     1     2     2     3     3     3


A(:,:,2) =

     1     2     2     3     3     3
     1     2     2     3     3     3


A(:,:,3) =

     1     2     2     3     3     3
     1     2     2     3     3     3
```

The size of A is 2-by-6-by-3.

Input Arguments

T — Input table
table

Input table, specified as a table. All variables in T must have sizes and data types that are compatible for horizontal concatenation. Specifically, the size of all variable dimensions greater than 2 must match.

- If T is an m-byn table with variables that each have one column, then each variable becomes one column in A, and A is an m-by-n array.
- If T contains variables that consist of more than one column, those variables become multiple columns in A, and the size of A is greater than the size of T.
- If T contains variables with more than two dimensions, the number of dimensions of A is the same as the number of variable dimensions.

Tips

- `table2array` horizontally concatenates the variables in T to create A. If the variables in T are cell arrays, `table2array` does not concatenate their contents, and A is a cell array, equivalent to `table2cell(T)`. To create an array containing the contents of variables that are all cell arrays, use `cell2mat(table2cell(T))`.
- `table2array(T)` is equivalent to `T{:,:}`.

Algorithms

If T contains variables with different data types that are compatible for horizontal concatenation, `table2array` creates a homogeneous array, A, of the dominant data type. For example, if T contains `double` and `single` numeric data, `table2array(T)` returns an array with data type `single`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- In generated code, the names of the table variables must be constant when using this function. For more information, see “Code Generation for Tables” (MATLAB Coder) and “Table Limitations for Code Generation” (MATLAB Coder).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`array2table` | `table` | `table2cell` | `table2struct`

Topics

“Concatenating Objects of Different Classes”

Introduced in R2013b

table2cell

Convert table to cell array

Syntax

```
C = table2cell(T)
```

Description

`C = table2cell(T)` converts the table, `T`, to a cell array, `C`. Each variable in `T` becomes a column of cells in `C`.

Examples

Convert Table to Cell Array

Create a table, `T`, with five rows and three variables.

```
T = table(categorical({'M';'M';'F';'F';'F'}),[38;43;38;40;49],...
    [124 93;109 77; 125 83; 117 75; 122 80],...
    'VariableNames',{'Gender' 'Age' 'BloodPressure'},...
    'RowNames',{'Smith' 'Johnson' 'Williams' 'Jones' 'Brown'})
```

```
T=5×3 table
                Gender    Age    BloodPressure
                ______    ___    _____________

    Smith         M       38      124     93
    Johnson       M       43      109     77
    Williams      F       38      125     83
    Jones         F       40      117     75
    Brown         F       49      122     80
```

Convert `T` to a cell array.

```
C = table2cell(T)
```

```
C=5×3 cell
    {[M]}    {[38]}    {1x2 double}
    {[M]}    {[43]}    {1x2 double}
    {[F]}    {[38]}    {1x2 double}
    {[F]}    {[40]}    {1x2 double}
    {[F]}    {[49]}    {1x2 double}
```

`C` is a 5-by-3 cell array.

Vertically concatenate the table property, `T.Properties.VariableNames`, with `C` to include column headings for the cell array.

```
[T.Properties.VariableNames;C]
```

```
ans=6×3 cell
    {'Gender'}    {'Age'}    {'BloodPressure'}
    {[M     ]}    {[ 38]}    {1x2 double     }
    {[M     ]}    {[ 43]}    {1x2 double     }
    {[F     ]}    {[ 38]}    {1x2 double     }
    {[F     ]}    {[ 40]}    {1x2 double     }
    {[F     ]}    {[ 49]}    {1x2 double     }
```

`T.Properties.VariableNames` is a cell array of character vectors.

Input Arguments

T — Input table
table

Input table, specified as a table.

If `T` is an `m`-by`n` table, then `C` is an `m`-by-`n` cell array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This function fully supports code generation. For more information, see “Code Generation for Tables” (MATLAB Coder).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cell2table` | `table` | `table2array` | `table2struct`

Introduced in R2013b

table2struct

Convert table to structure array

Syntax

```
S = table2struct(T)
S = table2struct(T,'ToScalar',true)
```

Description

`S = table2struct(T)` converts the table, `T`, to a structure array, `S`. Each variable of `T` becomes a field in `S`. If `T` is an `m`-by-`n` table, then `S` is a `m`-by-1 structure array with `n` fields.

`S = table2struct(T,'ToScalar',true)` converts the table, `T`, to a scalar structure `S`. Each variable of `T` becomes a field in `S`. If `T` is a `m`-by-`n` table, then `S` has `n` fields, each of which has `m` rows.

Examples

Convert Table to Structure Array

Create a table, `T`, with five rows and three variables.

```
T = table(categorical({'M';'M';'F';'F';'F'}),[38;43;38;40;49],...
    [124 93;109 77; 125 83; 117 75; 122 80],...
    'VariableNames',{'Gender' 'Age' 'BloodPressure'})
```

```
T=5×3 table
    Gender    Age    BloodPressure
    ______    ___    _____________

      M       38      124     93
      M       43      109     77
```

```
      F       38      125     83
      F       40      117     75
      F       49      122     80
```

Convert T to a structure array.

```
S = table2struct(T)
```

```
S=5×3 struct
    Gender
    Age
    BloodPressure
```

The structure is 5-by-1, corresponding to the five rows of the table, T. The three fields of S correspond to the three variables from T.

Display the field data for the first element of S.

```
S(1)
```

```
ans = struct with fields:
           Gender: M
              Age: 38
    BloodPressure: [124 93]
```

The information corresponds to the first row of the table.

Convert Table to Scalar Structure

Create a table, T, with five rows and three variables.

```
T = table(categorical({'M';'M';'F';'F';'F'}),[38;43;38;40;49],...
    [124 93;109 77; 125 83; 117 75; 122 80],...
    'VariableNames',{'Gender' 'Age' 'BloodPressure'})
```

```
T=5×3 table
    Gender    Age    BloodPressure
    ______    ___    _____________

      M       38      124     93
```

```
      M       43      109     77
      F       38      125     83
      F       40      117     75
      F       49      122     80
```

Convert T to a scalar structure.

```
S = table2struct(T,'ToScalar',true)
```

```
S = struct with fields:
           Gender: [5x1 categorical]
              Age: [5x1 double]
    BloodPressure: [5x2 double]
```

The data in the fields of the scalar structure are 5-by-1, corresponding to the five rows in the table T.

Display the data for the field `BloodPressure`.

```
S.BloodPressure
```

```
ans = 5×2

   124    93
   109    77
   125    83
   117    75
   122    80
```

The structure field `BloodPressure` contains all of the data that was in the variable of the same name from table T.

Convert Table with Row Names to Structure

Create a table, T, that includes row names.

```
T = table(categorical({'M';'M';'F';'F';'F'}),[38;43;38;40;49],...
    [124 93;109 77; 125 83; 117 75; 122 80],...
    'VariableNames',{'Gender' 'Age' 'BloodPressure'},...
    'RowNames',{'Smith' 'Johnson' 'Williams' 'Jones' 'Brown'})
```

```
T=5×3 table
                Gender    Age    BloodPressure
                ______    ___    _____________

    Smith         M       38      124     93
    Johnson       M       43      109     77
    Williams      F       38      125     83
    Jones         F       40      117     75
    Brown         F       49      122     80
```

Convert T to a scalar structure.

```
S = table2struct(T,'ToScalar',true)
```

```
S = struct with fields:
           Gender: [5x1 categorical]
              Age: [5x1 double]
    BloodPressure: [5x2 double]
```

Add a field for the row names from the table.

```
S.RowNames = T.Properties.RowNames
```

```
S = struct with fields:
           Gender: [5x1 categorical]
              Age: [5x1 double]
    BloodPressure: [5x2 double]
         RowNames: {5x1 cell}
```

If S is a nonscalar structure, use `[S.RowNames] = T.Properties.RowNames{:}` to include a field with the row names from the table.

Input Arguments

T — Input table
table

Input table, specified as a table.

If T has variables whose names are not valid MATLAB identifiers, then `table2struct` modifies them to create valid field names, primarily by removing spaces and replacing non-ASCII characters with underscores.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- In generated code, the input table must be constant when using this function. For more information, see “Code Generation for Tables” (MATLAB Coder) and “Table Limitations for Code Generation” (MATLAB Coder).

See Also

`struct2table` | `table` | `table2array` | `table2cell`

Introduced in R2013b

table2timetable

Convert table to timetable

Syntax

```
TT = table2timetable(T)
TT = table2timetable(T,'RowTimes',timeVarName)
TT = table2timetable(T,'RowTimes',rowTimes)
TT = table2timetable(T,'SampleRate',Fs)
TT = table2timetable(T,'TimeStep',dt)
TT = table2timetable( ___ ,'StartTime',t0)
```

Description

`TT = table2timetable(T)` converts the table `T` to a timetable. The first datetime or duration variable in `T` becomes the vector of row times of `TT`. The remaining variables of `T` become the variables of `TT`.

- If `T` is an `M`-by-`N` table without row names, then `TT` is an `M`-by-`(N-1)` timetable.
- If `T` is an `M`-by-`N` table with row names, then `table2timetable` assigns the row names of `T` to a variable of `TT`. As a result, `TT` is an `M`-by-`N` timetable.

For more information on creating and using timetables, see “Timetables”.

To create a timetable from data in a text or spreadsheet file, first use the `readtable` function to read the data to a table. Then convert the table to a timetable with `table2timetable`.

`TT = table2timetable(T,'RowTimes',timeVarName)` assigns the table variable `timeVarName` as the vector of row times of the output timetable. `timeVarName` can be the name of any variable in `T` that contains datetime or duration values. The remaining variables of `T` become the variables of `TT`.

`TT = table2timetable(T,'RowTimes',rowTimes)` assigns the vector `rowTimes` as the vector of row times of the output timetable. All of the variables of `T` become variables of `TT`.

`TT = table2timetable(T,'SampleRate',Fs)` uses the sample rate `Fs` to calculate regularly spaced row times. `Fs` is a positive numeric scalar that specifies the number of samples per second (Hz). The first row time is zero seconds.

`TT = table2timetable(T,'TimeStep',dt)` uses the time step `dt` to calculate regularly spaced row times. `dt` is a duration or calendar duration value that specifies the length of time between consecutive row times. The first row time is zero seconds.

`TT = table2timetable( ___ ,'StartTime',t0)` specifies start time `t0`, instead of zero seconds, as the first row time. You can use this syntax when you create a regular timetable using either the `'SampleRate'` or `'TimeStep'` name-value pair arguments from either of the previous two syntaxes.

Examples

Convert Table to Timetable

Convert a table that contains dates and times to a timetable.

Read power outage data from the file `outages.csv` to a table. The table contains both outage and restoration times.

```
T = readtable('outages.csv');
T(1:5,:)
```

```
ans=5×6 table
       Region           OutageTime        Loss     Customers     RestorationTime
    _____________    ________________    ______    __________    ________________    _

    {'SouthWest'}    2002-02-01 12:18    458.98    1.8202e+06    2002-02-07 16:50    {
    {'SouthEast'}    2003-01-23 00:49    530.14    2.1204e+05                NaT    {
    {'SouthEast'}    2003-02-07 21:15     289.4    1.4294e+05    2003-02-17 08:14    {
    {'West'     }    2004-04-06 05:44    434.81    3.4037e+05    2004-04-06 06:10    {
    {'MidWest'  }    2002-03-16 06:18    186.44    2.1275e+05    2002-03-18 23:23    {
```

Convert the table to a timetable. The first variable with times, `OutageTime`, becomes the time vector of `TT`.

```
TT = table2timetable(T);
TT(1:5,:)
```

```
ans=5×6 timetable
       OutageTime          Region         Loss     Customers     RestorationTime
    ________________    _____________    ______    __________    ________________    __

    2002-02-01 12:18    {'SouthWest'}    458.98    1.8202e+06    2002-02-07 16:50    {
    2003-01-23 00:49    {'SouthEast'}    530.14    2.1204e+05                 NaT    {
    2003-02-07 21:15    {'SouthEast'}     289.4    1.4294e+05    2003-02-17 08:14    {
    2004-04-06 05:44    {'West'     }    434.81    3.4037e+05    2004-04-06 06:10    {
    2002-03-16 06:18    {'MidWest'  }    186.44    2.1275e+05    2002-03-18 23:23    {
```

Index into TT using row times from its time vector. You can treat the row times as labels that specify rows.

```
TT('2003-02-07 21:15',:)
```

```
ans=1×6 timetable
       OutageTime          Region        Loss     Customers     RestorationTime
    ________________    _____________    _____    __________    ________________    __

    2003-02-07 21:15    {'SouthEast'}    289.4    1.4294e+05    2003-02-17 08:14    {'v
```

Calculate the duration of power outages. Use dot syntax to extract the row times as a vector.

```
TT.OutageDuration = TT.RestorationTime - TT.OutageTime;
TT(1:5,:)
```

```
ans=5×7 timetable
       OutageTime          Region         Loss     Customers     RestorationTime
    ________________    _____________    ______    __________    ________________    __

    2002-02-01 12:18    {'SouthWest'}    458.98    1.8202e+06    2002-02-07 16:50    {
    2003-01-23 00:49    {'SouthEast'}    530.14    2.1204e+05                 NaT    {
    2003-02-07 21:15    {'SouthEast'}     289.4    1.4294e+05    2003-02-17 08:14    {
    2004-04-06 05:44    {'West'     }    434.81    3.4037e+05    2004-04-06 06:10    {
    2002-03-16 06:18    {'MidWest'  }    186.44    2.1275e+05    2002-03-18 23:23    {
```

Specify Table Variable as Row Times

Convert a table to a timetable and specify the table variable that becomes the time vector of the timetable.

Read power outage data from the file `outages.csv` to a table. The table contains both outage and restoration times.

```
T = readtable('outages.csv');
T(1:5,:)
```

```
ans=5×6 table
       Region           OutageTime        Loss     Customers     RestorationTime
    _____________    _________________    ______    __________    _________________

    {'SouthWest'}    2002-02-01 12:18    458.98    1.8202e+06    2002-02-07 16:50    {
    {'SouthEast'}    2003-01-23 00:49    530.14    2.1204e+05                  NaT    {
    {'SouthEast'}    2003-02-07 21:15     289.4    1.4294e+05    2003-02-17 08:14    {
    {'West'     }    2004-04-06 05:44    434.81    3.4037e+05    2004-04-06 06:10    {
    {'MidWest'  }    2002-03-16 06:18    186.44    2.1275e+05    2002-03-18 23:23    {
```

Convert the table to a timetable. Specify the second variable with dates and times, `RestorationTime`, as the time vector of the timetable.

```
TT = table2timetable(T,'RowTimes','RestorationTime');
TT(1:5,:)
```

```
ans=5×6 timetable
    RestorationTime        Region           OutageTime        Loss     Customers
    _________________    _____________    _________________    ______    __________    _

    2002-02-07 16:50    {'SouthWest'}    2002-02-01 12:18    458.98    1.8202e+06    {
    NaT                 {'SouthEast'}    2003-01-23 00:49    530.14    2.1204e+05    {
    2003-02-17 08:14    {'SouthEast'}    2003-02-07 21:15     289.4    1.4294e+05    {
    2004-04-06 06:10    {'West'     }    2004-04-06 05:44    434.81    3.4037e+05    {
    2002-03-18 23:23    {'MidWest'  }    2002-03-16 06:18    186.44    2.1275e+05    {
```

Convert Table with Separate Time Vector

Convert a table to a timetable by adding a separate time vector that contains the row times. All the table variables become variables of the timetable.

```
Reading1 = [98;97.5;97.9;98.1;97.9];
Reading2 = [120;111;119;117;116];
T = table(Reading1,Reading2)
```

```
T=5×2 table
    Reading1    Reading2
    ________    ________

        98        120
      97.5        111
      97.9        119
      98.1        117
      97.9        116
```

```
Time = [seconds(1):seconds(1):seconds(5)];
TT = table2timetable(T,'RowTimes',Time)
```

```
TT=5×3 timetable
    Time     Reading1    Reading2
    _____    ________    ________

    1 sec        98        120
    2 sec      97.5        111
    3 sec      97.9        119
    4 sec      98.1        117
    5 sec      97.9        116
```

Specify Sample Rate

Create a table.

```
Reading1 = [98;97.5;97.9;98.1;97.9];
Reading2 = [120;111;119;117;116];
T = table(Reading1,Reading2)
```

```
T=5×2 table
    Reading1    Reading2
```

```
    ________    ________

       98         120
     97.5         111
     97.9         119
     98.1         117
     97.9         116
```

Specify a sample rate of 500 Hz. Convert T to a timetable using that sample rate and a starting row time of zero seconds.

```
TT = table2timetable(T,'SampleRate',500)
```

```
TT=5×3 timetable
      Time       Reading1    Reading2
    _________    ________    ________

    0 sec            98        120
    0.002 sec      97.5        111
    0.004 sec      97.9        119
    0.006 sec      98.1        117
    0.008 sec      97.9        116
```

Specify Time Step

Create a table.

```
Reading1 = [98;97.5;97.9;98.1;97.9];
Reading2 = [120;111;119;117;116];
T = table(Reading1,Reading2)
```

```
T=5×2 table
    Reading1    Reading2
    ________    ________

       98         120
     97.5         111
     97.9         119
     98.1         117
     97.9         116
```

Specify a time step of ten seconds, using the `seconds` function. Convert `T` to a timetable using that sampling rate and a starting row time of zero seconds.

```
TT = table2timetable(T,'TimeStep',seconds(10))
```

```
TT=5×3 timetable
      Time      Reading1    Reading2
    ________    ________    ________

    00:00:00        98         120
    00:00:10      97.5         111
    00:00:20      97.9         119
    00:00:30      98.1         117
    00:00:40      97.9         116
```

Specify Start Time

Create a table.

```
Reading1 = [98;97.5;97.9;98.1;97.9];
Reading2 = [120;111;119;117;116];
T = table(Reading1,Reading2)
```

```
T=5×2 table
    Reading1    Reading2
    ________    ________

        98         120
      97.5         111
      97.9         119
      98.1         117
      97.9         116
```

Convert it to a timetable by specifying a time step of ten seconds and a start time of 5 seconds.

```
dt = seconds(10);
t0 = seconds(5);
TT = table2timetable(T,'TimeStep',dt,'StartTime',t0)
```

```
TT=5×3 timetable
     Time     Reading1    Reading2
    ______    ________    ________

    5 sec         98        120
    15 sec      97.5        111
    25 sec      97.9        119
    35 sec      98.1        117
    45 sec      97.9        116
```

Input Arguments

T — Input table
table

Input table.

timeVarName — Name of variable from input table
character vector | string scalar

Name of a variable from the input table, specified as a character vector or a string scalar.

rowTimes — Row times assigned to output timetable
datetime vector | duration vector

Row times assigned to the output timetable, specified as a datetime vector or a duration vector. The number of elements of `rowTimes` must equal the number of rows of the input table. The time values in `rowTimes` do not need to be unique, sorted, or regular.

Fs — Sample rate
positive numeric scalar

Sample rate, specified as a positive numeric scalar. `Fs` specifies the number of samples per second (Hz).

dt — Time step
datetime scalar | duration scalar

Time step, specified as a datetime scalar or duration scalar.

Data Types: `datetime` | `duration` | `calendarDuration`

`t0` — Start time
datetime scalar | duration scalar

Start time, specified as a datetime scalar or duration scalar.

- If `t0` is a datetime value, then the row times of `TT` are datetime values.
- If `t0` is a duration, then the row times are durations.

If the time step `dt` is a calendar duration value, then `t0` must be a datetime value.

Data Types: `datetime` | `duration`

Output Arguments

`TT` — Output timetable
timetable

Output timetable. The timetable can store metadata such as descriptions, variable units, variable names, and row times. For more information, see the Properties sections of `timetable`.

Tips

- In certain cases, you can call `table2timetable` with a syntax that specifies a regular time step between row times, and yet `table2timetable` returns an irregular timetable. This result occurs when you specify the time step using a calendar unit of time and there is a row time that introduces an irregular step. For example, if you create a timetable with a time step of one calendar month, starting on January 31, 2019, then it is irregular with respect to months.

```
stime = datetime(2019,1,31);
tstep = calmonths(1);
T = table([1:3]');
TT = table2timetable(T,'TimeStep',tstep,'StartTime',stime)
```

```
TT =

  3×1 timetable
```

```
   Time        Var1
___________    ____

31-Jan-2019     1
28-Feb-2019     2
31-Mar-2019     3
```

In addition, there are other cases where irregularities are due to shifts from Daylight Saving Time (DST) or to row times that are leap seconds. This table specifies the row time values and time steps that can produce irregular timetables unexpectedly.

Row Time Value	Time Step
Start time specified as the 29th, 30th, or 31st day of the month	Number of calendar months or quarters
Start time specified as February 29	Number of calendar years
Any row time occurring between 1:00 a.m. and 2:00 a.m. on a day shifting from DST to standard time (when row times are specified as datetime values whose time zone observes DST)	Number of calendar days or months
Any row time that is a leap second (when row times are specified as datetime values whose time zone is the `UTCLeapSeconds` time zone)	Time step specified in any calendar unit (days, weeks, months, quarters, or years)

Compatibility Considerations

`'SamplingRate'` is not recommended

Not recommended starting in R2018b

The `'SamplingRate'` name-value pair argument is not recommended. Use `'SampleRate'` instead. The corresponding timetable property is also named `SampleRate`.

For backward compatibility, you still can specify `'SamplingRate'` as the name of the name-value pair. However, the value is assigned to the `SampleRate` property.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`array2timetable` | `isvarname` | `summary` | `timetable` | `uitable`

Topics

“Create Timetables”
“Resample and Aggregate Data in Timetable”
“Combine Timetables and Synchronize Their Data”
“Select Timetable Data by Row Time and Variable Type”
“Access Data in Tables”
“Tables”
“Represent Dates and Times in MATLAB”

Introduced in R2016b

tail

Get bottom rows of table, timetable, or tall array

Syntax

```
B = tail(A)
B = tail(A,k)
```

Description

`B = tail(A)` returns the last eight rows of table or timetable `A`.

`B = tail(A,k)` returns the last `k` rows of `A`.

Examples

Preview Table

Create a table that contains 100 rows and five variables.

```
load patients
T = table(LastName,Gender,Age,Height,Weight);
size(T)
```

```
ans = 1×2

   100     5
```

Preview the last eight rows.

```
T2 = tail(T)
```

```
T2=8×5 table
      LastName         Gender      Age    Height    Weight
```

```
    _____________    ___________    ___    ______    ______

    {'Foster'   }    {'Female'}    30       70        124
    {'Gonzales' }    {'Male'  }    48       71        174
    {'Bryant'   }    {'Female'}    48       66        134
    {'Alexander'}    {'Male'  }    25       69        171
    {'Russell'  }    {'Male'  }    44       69        188
    {'Griffin'  }    {'Male'  }    49       70        186
    {'Diaz'     }    {'Male'  }    45       68        172
    {'Hayes'    }    {'Male'  }    48       66        177
```

Preview Contents of Tall Table

Create a tall table and preview the bottom few rows of data.

Create a tall table for the `airlinesmall.csv` data set. Select a subset of the variables to work with. Use `tail` to extract the last few rows of data.

```
varnames = {'Year','Month','ArrDelay','DepDelay','UniqueCarrier'};
ds = datastore('airlinesmall.csv','TreatAsMissing','NA',...
    'SelectedVariableNames',varnames);
T = tall(ds)
```

```
T =

  Mx5 tall table

    Year    Month    ArrDelay    DepDelay    UniqueCarrier
    ____    _____    ________    ________    _____________

    1987     10          8          12          {'PS'}
    1987     10          8           1          {'PS'}
    1987     10         21          20          {'PS'}
    1987     10         13          12          {'PS'}
    1987     10          4          -1          {'PS'}
    1987     10         59          63          {'PS'}
    1987     10          3          -2          {'PS'}
    1987     10         11          -1          {'PS'}
     :        :          :           :              :
     :        :          :           :              :
```

```
tt = tail(T)
```

```
tt =

  Mx5 tall table

    Year    Month    ArrDelay    DepDelay    UniqueCarrier
    ____    _____    ________    ________    _____________

     ?        ?         ?           ?              ?
     ?        ?         ?           ?              ?
     ?        ?         ?           ?              ?
     :        :         :           :              :
     :        :         :           :              :
```

Collect the results into memory to view the data.

```
last_rows = gather(tt)
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 1.1 sec
Evaluation completed in 1.4 sec

last_rows=8×5 table
    Year    Month    ArrDelay    DepDelay    UniqueCarrier
    ____    _____    ________    ________    _____________

    2008     12         14           1          {'DL'}
    2008     12         -8          -1          {'DL'}
    2008     12          1           9          {'DL'}
    2008     12         -8          -4          {'DL'}
    2008     12         15          -2          {'DL'}
    2008     12        -15          -1          {'DL'}
    2008     12        -12           1          {'DL'}
    2008     12         -1          11          {'DL'}
```

Retrieve Specified Number of Rows in Tall Array

Preview the last 20 rows of data in a tall table.

Create a tall table for the `airlinesmall.csv` data set. Select a subset of the variables to work with, and treat `'NA'` values as missing data so that `datastore` replaces them with `NaN` values. Use `tail` to view the last 20 rows of data.

```
varnames = {'Year','Month','ArrDelay','DepDelay','UniqueCarrier'};
ds = datastore('airlinesmall.csv','TreatAsMissing','NA',...
    'SelectedVariableNames',varnames);
T = tall(ds)
```

```
T =

  Mx5 tall table

    Year    Month    ArrDelay    DepDelay    UniqueCarrier
    ____    _____    ________    ________    _____________

    1987     10          8          12          {'PS'}
    1987     10          8           1          {'PS'}
    1987     10         21          20          {'PS'}
    1987     10         13          12          {'PS'}
    1987     10          4          -1          {'PS'}
    1987     10         59          63          {'PS'}
    1987     10          3          -2          {'PS'}
    1987     10         11          -1          {'PS'}
     :        :          :           :              :
     :        :          :           :              :
```

```
tt = tail(T,20)
```

```
tt =

  Mx5 tall table

    Year    Month    ArrDelay    DepDelay    UniqueCarrier
    ____    _____    ________    ________    _____________

     ?        ?          ?           ?              ?
     ?        ?          ?           ?              ?
     ?        ?          ?           ?              ?
     :        :          :           :              :
     :        :          :           :              :
```

Collect the results into memory to view the data.

```
b20 = gather(tt)
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 1.4 sec
Evaluation completed in 1.7 sec
```

```
b20=20×5 table
    Year    Month    ArrDelay    DepDelay    UniqueCarrier
    ____    _____    ________    ________    _____________

    2008     12          0          -4          {'CO'}
    2008     12        -16          13          {'CO'}
    2008     12         17          -3          {'CO'}
    2008     12          3          -5          {'CO'}
    2008     12          2           6          {'DL'}
    2008     12          6          -2          {'DL'}
    2008     12         37          35          {'DL'}
    2008     12         -1          -6          {'DL'}
    2008     12         39          12          {'DL'}
    2008     12         -3          -6          {'DL'}
    2008     12         -6          -1          {'DL'}
    2008     12         -2           1          {'DL'}
    2008     12         14           1          {'DL'}
    2008     12         -8          -1          {'DL'}
    2008     12          1           9          {'DL'}
    2008     12         -8          -4          {'DL'}
      ⋮
```

Input Arguments

A — Input array
table | timetable

Input array, specified as a table or timetable.

Data Types: `table` | `timetable`

k — Number of rows to extract
scalar

Number of rows to extract, specified as a positive scalar integer. If `A` has fewer than `k` rows, then `tail` returns all of `A`.

Output Arguments

B — Requested rows
table | timetable

Requested rows, returned as a table or timetable. The data type of B is the same as A.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

You can use `head` and `tail` with tall arrays of any valid underlying data type (`single`, `double`, `int8`, `datetime`, `table`, and so on).

If you are unsure whether the result returned by `gather(A)` will fit in memory, then use `gather(head(A))` or `gather(tail(A))`. These commands still fully evaluate the tall array A, but only return a small subset of the result in memory.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`gather` | `head` | `summary` | `table` | `tall` | `timetable` | `topkrows`

Topics

"Index and View Tall Array Elements"

Introduced in R2016b

matlab.tall.transform

Package: `matlab.tall`

Transform array by applying function handle to blocks of data

Syntax

```
tA = matlab.tall.transform(fcn,tX)
tA = matlab.tall.transform(fcn,tX,tY,...)
[tA,tB,...] = matlab.tall.transform(fcn,tX,tY,...)
[tA,tB,...] = matlab.tall.transform( ___ ,'OutputsLike',{PA,PB,...})
```

Description

`tA = matlab.tall.transform(fcn,tX)` applies the function handle `fcn` to each block on page 1-15184 of array `tX` and returns a transformed array, `tA`.

`tA = matlab.tall.transform(fcn,tX,tY,...)` specifies several arrays `tX,tY,...` that are inputs to `fcn`. The same rows of each array are operated on by `fcn`; for example, `fcn(tX(n:m,:),tY(n:m,:))`. Inputs with a height of one are passed to every call of `fcn`.

`[tA,tB,...] = matlab.tall.transform(fcn,tX,tY,...)` , where `fcn` is a function that returns multiple outputs, returns arrays `tA,tB,...`, each corresponding to one of the output arguments of `fcn`. All outputs of `fcn` must have the same height, and the number of outputs must be the same as the number that are requested from `matlab.tall.transform`.

`[tA,tB,...] = matlab.tall.transform( ___ ,'OutputsLike',{PA,PB,...})` specifies that the outputs `tA,tB,...` have the same data types as the prototype arrays `PA,PB,...`, respectively. You can use any of the input argument combinations in previous syntaxes.

Examples

Apply Function to Tall Vector

Use `matlab.tall.transform` to build a tall array of zeros with attributes similar to another array.

Create a tall table for the `airlinesmall.csv` data set. The data contains information about arrival and departure times of US flights. Extract the `ArrDelay` variable, which is a vector of arrival delays.

```
ds = tabularTextDatastore('airlinesmall.csv','TreatAsMissing','NA');
ds.SelectedVariableNames = {'ArrDelay' 'DepDelay'};
tt = tall(ds);
tX = tt.ArrDelay
```

```
tX =

  Mx1 tall double column vector

     8
     8
    21
    13
     4
    59
     3
    11
    :
    :
```

Write an anonymous function that creates an array of zeros with the same size and data type as the input.

```
zerosLike = @(in) zeros(size(in),'like',in);
```

Use `matlab.tall.transform` to apply the `zerosLike` function to the vector of arrival delays. The result is a tall vector of the same size, but whose values are all zero.

```
s = matlab.tall.transform(zerosLike, tX)
```

```
s =

  Mx1 tall double column vector

     0
     0
```

```
     0
     0
     0
     0
     0
     0
     :
     :
```

Transform Two Vectors

Calculate the mean total flight delay from vectors of arrival and departure delays.

Create a tall table for the `airlinesmall.csv` data set. The data contains information about arrival and departure times of US flights. Extract the `ArrDelay` and `DepDelay` variables, which are vectors of arrival and departure delays.

```
ds = tabularTextDatastore('airlinesmall.csv','TreatAsMissing','NA');
ds.SelectedVariableNames = {'ArrDelay' 'DepDelay'};
tt = tall(ds);
tX = tt.ArrDelay;
tY = tt.DepDelay;
```

The `meanDelay` function concatenates the input vectors into a matrix, sums the values in each row (ignoring NaNs), and then it calculates the mean. Display the contents of that function file.

```
type meanDelay
```

```
function D = meanDelay(a,b)
X = [a b];
Y = sum(X,2,'omitnan');
D = mean(Y);
end
```

Use `matlab.tall.transform` to apply the `meanDelay` function to each block of data in `tX` and `tY`. The result is the mean total delay in each block of data.

```
d = matlab.tall.transform(@meanDelay, tX, tY)
```

```
d =
```

```
  7x1 tall double column vector

   14.0621
   11.1639
   17.2311
   15.1852
   12.5860
   19.8596
   14.4036
```

This operation assumes that the result of reducing each block of data to a scalar value can fit in memory. For extremely large data sets and data sets that use a small block size, that assumption might not be true.

Apply Function with Multiple Outputs

Find the maximum value and the index of that value in each row of data.

Create a tall table for the `airlinesmall.csv` data set. The data contains information about arrival and departure times of US flights. Extract the `ArrDelay` and `DepDelay` variables, which are vectors of arrival and departure delays.

```
ds = tabularTextDatastore('airlinesmall.csv','TreatAsMissing','NA');
ds.SelectedVariableNames = {'ArrDelay' 'DepDelay'};
tt = tall(ds);
tX = tt.ArrDelay;
tY = tt.DepDelay;
```

The `maxDelay` function concatenates the input vectors, and then it finds the maximum arrival or departure delay duration and its column index. Display the contents of that file.

```
type maxDelay
```

```
function [M,I] = maxDelay(A,B)
X = [A B];
[M,I] = max(X,[],2);
end
```

Use `matlab.tall.transform` to apply the `maxDelay` function to each block of data in `tX` and `tY`. The result is the maximum arrival or departure delay for each row of data, as well as an index vector indicating which column the maximum value came from. An index

of 1 indicates that the arrival delay in that row is larger, and an index of 2 indicates that the departure delay is larger.

```
[M, idx] = matlab.tall.transform(@maxDelay, tX, tY)
```

```
M =

  Mx1 tall double column vector

    12
     8
    21
    13
     4
    63
     3
    11
    :
    :


idx =

  Mx1 tall double column vector

     2
     1
     1
     1
     1
     2
     1
     1
     :
     :
```

Output Table with Different Variables

Use the `'OutputsLike'` option to return a table from `matlab.tall.transform` that has different variables from the input table.

Create a tall table with two variables of random values.

```
T = tall(table(rand(1e6,1),rand(1e6,1)))
```

```
T =

  1,000,000x2 tall table

     Var1       Var2
    _______    _______

    0.81472    0.90399
    0.90579    0.94095
    0.12699    0.80252
    0.91338    0.24205
    0.63236    0.97566
    0.09754    0.31723
     0.2785    0.81279
    0.54688    0.69743
       :          :
       :          :
```

The function `tableDiff` calculates the difference between two input table variables and adds the result as a new variable in the table. Display the contents of the file.

```
type tableDiff
```

```
function Tout = tableDiff(Tin)
d = Tin.Var2 - Tin.Var1;
Tin.Var3 = abs(d);
Tout = Tin;
end
```

Use `matlab.tall.transform` to apply the `tableDiff` function to each block of data in `T`. Since the output table has different variables from the input table, use the `'OutputsLike'` name-value pair to supply a prototype table with similar variables as the output (three variables with the default names `Var1`, `Var2`, and `Var3`).

```
Z = matlab.tall.transform(@tableDiff, T, 'OutputsLike', {table(1,1,1)})
```

```
Z =

  Mx3 tall table

     Var1       Var2        Var3
    _______    _______    ________
```

```
    0.81472    0.90399    0.089267
    0.90579    0.94095    0.035156
    0.12699    0.80252     0.67553
    0.91338    0.24205     0.67133
    0.63236    0.97566      0.3433
    0.09754    0.31723     0.21969
     0.2785    0.81279     0.53429
    0.54688    0.69743     0.15054
       :          :          :
       :          :          :
```

Input Arguments

`fcn` — Transform function to apply
function handle | anonymous function

Transform function to apply, specified as a function handle or anonymous function. Each output of `fcn` must be the same type as the first input `tX`. You can use the `'OutputsLike'` option to return outputs of different data types. If `fcn` returns more than one output, then the outputs must all have the same height.

The general functional signature of `fcn` is

```
[a, b, c, ...] = fcn(x, y, z, ...)
```

`fcn` must satisfy these requirements:

1. **Input Arguments** — The inputs `[x, y, z, ...]` are blocks of data that fit in memory. The blocks are produced by extracting data from the respective tall array inputs `[tX, tY, tZ, ...]`. The inputs `[x, y, z, ...]` satisfy these properties:

   - All of `[x, y, z, ...]` have the same size in the first dimension after any allowed expansion.
   - The blocks of data in `[x, y, z, ...]` come from the same index in the tall dimension, assuming the tall array is nonsingleton in the tall dimension. For example, if `tX` and `tY` are nonsingleton in the tall dimension, then the first set of blocks might be `x = tX(1:20000,:)` and `y = tY(1:20000,:)`.
   - If the first dimension of any of `[tX, tY, tZ, ...]` has a size of `1`, then the corresponding block `[x, y, z, ...]` consists of all the data in that tall array.

2. **Output Arguments** — The outputs `[a, b, c, ...]` are blocks that fit in memory, to be sent to the respective outputs `[tA, tB, tC, ...]`. The outputs `[a, b, c, ...]` satisfy these properties:
   - All of `[a, b, c, ...]` must have the same size in the first dimension.
   - All of `[a, b, c, ...]` are vertically concatenated with the respective results of previous calls to `fcn`.
   - All of `[a, b, c, ...]` are sent to the same index in the first dimension in their respective destination output arrays.
3. **Functional Rules** — `fcn` must satisfy the functional rule:
   - `F([inputs1; inputs2]) == [F(inputs1); F(inputs2)]`: Applying the function to the concatenation of the inputs should be the same as applying the function to the inputs separately and then concatenating the results.
4. **Empty Inputs** — Ensure that `fcn` can handle an input that has a height of 0. Empty inputs can occur when a file is empty or if you have done a lot of filtering on the data.

For example, this function accepts two input arrays, squares them, and returns two output arrays:

```
function [xx,yy] = sqInputs(x,y)
xx = x.^2;
yy = y.^2;
end
```

After you save this function to an accessible folder, you can invoke the function to square `tX` and `tY` with this command:

```
[tA,tB] = matlab.tall.transform(@sqInputs,tX,tY)
```

Example: `tA = matlab.tall.transform(@(x) x .* 2, tX)` specifies an anonymous function to multiply the values in `tX` by 2.

Example: `tC = matlab.tall.transform(@plus,tX,tY)` specifies a function handle `@plus` to add two arrays together.

Data Types: `function_handle`

tX, tY — Input arrays

scalars | vectors | matrices | multidimensional arrays

Input arrays, specified as scalars, vectors, matrices, or multidimensional arrays. The input arrays are used as inputs to the specified function `fcn`. Each input array `tX,tY,...`

must have compatible heights. Two inputs have compatible height when they have the same height, or when one input is of height one.

PA, PB — Prototype of output arrays
arrays

Prototype of output arrays, specified as arrays. When you specify `'OutputsLike'`, the output arrays `tA,tB,...` returned by `matlab.tall.transform` have the same data types as the specified arrays `{PA,PB,...}`.

Example: `tA = matlab.tall.transform(fcn,tX,'OutputsLike',{int8(1)});`, where `tX` is a double-precision array, returns `A` as `int8` instead of `double`.

Output Arguments

tA, tB — Output arrays
scalars | vectors | matrices | multidimensional arrays

Output arrays, returned as scalars, vectors, matrices, or multidimensional arrays. If any input to `matlab.tall.transform` is tall, then all output arguments are also tall. Otherwise, all output arguments are in-memory arrays.

The size and data type of the output arrays depend on the specified function `fcn`. In general, the outputs `tA,tB,...` must all have the same data type as the first input `X`. However, you can specify `'OutputsLike'` to return different data types. The output arrays `tA,tB,...` all have the same height.

More About

Tall Array Blocks

When you create a tall array from a datastore, the underlying datastore facilitates the movement of data during a calculation. The data moves in discrete pieces called *blocks* or *chunks*, where each block is a set of consecutive rows that can fit in memory. For example, one block of a 2-D array (such as a table) is `X(n:m,:)`, for some subscripts `n` and `m`. The size of each block is based on the value of the `ReadSize` property of the datastore, but the block might not be exactly that size. For the purposes of `matlab.tall.transform`, a tall array is considered to be the vertical concatenation of many such blocks:

For example, if you use the `sum` function as the transform function, the result is the sum *per block*. Therefore, instead of returning a single scalar value for the sum of the elements, the result is a vector with length equal to the number of blocks.

```
ds = tabularTextDatastore('airlinesmall.csv','TreatAsMissing','NA');
ds.SelectedVariableNames = {'ArrDelay' 'DepDelay'};
tt = tall(ds);
tX = tt.ArrDelay;
```

```
f = @(x) sum(x,'omitnan');
s = matlab.tall.transform(f, tX);
s = gather(s)
```

```
s =

      140467
      101065
      164355
      135920
      111182
      186274
       21321
```

See Also

`matlab.tall.reduce` | `tall`

Topics

"Develop Custom Tall Array Algorithms"

Introduced in R2018b

matlab.tall.reduce

Package: `matlab.tall`

Reduce arrays by applying reduction algorithm to blocks of data

Syntax

```
tA = matlab.tall.reduce(fcn,reducefcn,tX)
tA = matlab.tall.reduce(fcn,reducefcn,tX,tY,...)
[tA,tB,...] = matlab.tall.reduce(fcn,reducefcn,tX,tY,...)
[tA,tB,...] = matlab.tall.reduce( ___ ,'OutputsLike',{PA,PB,...})
```

Description

`tA = matlab.tall.reduce(fcn,reducefcn,tX)` applies the function `fcn` to each block on page 1-15200 of array `tX` to generate partial results. Then the function applies `reducefcn` to the vertical concatenation of partial results repeatedly until it has one final result, `tA`.

`tA = matlab.tall.reduce(fcn,reducefcn,tX,tY,...)` specifies several arrays `tX,tY,...` that are inputs to `fcn`. The same rows of each array are operated on by `fcn`; for example, `fcn(tX(n:m,:),tY(n:m,:))`. Inputs with a height of one are passed to every call of `fcn`. With this syntax, `fcn` must return one output, and `reducefcn` must accept one input and return one output.

`[tA,tB,...] = matlab.tall.reduce(fcn,reducefcn,tX,tY,...)` , where `fcn` and `reducefcn` are functions that return multiple outputs, returns arrays `tA,tB,...`, each corresponding to one of the output arguments of `fcn` and `reducefcn`. This syntax has these requirements:

- `fcn` must return the same number of outputs as were requested from `matlab.tall.reduce`.
- `reducefcn` must have the same number of inputs and outputs as the number of outputs requested from `matlab.tall.reduce`.
- Each output of `fcn` and `reducefcn` must be the same type as the first input `tX`.

- Corresponding outputs of `fcn` and `reducefcn` must have the same height.

`[tA,tB,...] = matlab.tall.reduce( ___ ,'OutputsLike',{PA,PB,...})` specifies that the outputs `tA,tB,...` have the same data types as the prototype arrays `PA,PB,...`, respectively. You can use any of the input argument combinations in previous syntaxes.

Examples

Apply Reduction Functions to Tall Vector

Create a tall table, extract a tall vector from the table, and then find the total number of elements in the vector.

Create a tall table for the `airlinesmall.csv` data set. The data contains information about arrival and departure times of US flights. Extract the `ArrDelay` variable, which is a vector of arrival delays.

```
ds = tabularTextDatastore('airlinesmall.csv','TreatAsMissing','NA');
ds.SelectedVariableNames = {'ArrDelay' 'DepDelay'};
tt = tall(ds);
tX = tt.ArrDelay;
```

Use `matlab.tall.reduce` to count the total number of non-`NaN` elements in the tall vector. The first function `numel` counts the number of elements in each block of data, and the second function `sum` adds together all of the counts for each block to produce a scalar result.

```
s = matlab.tall.reduce(@numel,@sum,tX)
```

```
s =

  MxNx... tall double array

    ?    ?    ?    ...
    ?    ?    ?    ...
    ?    ?    ?    ...
    :    :    :
    :    :    :
```

Gather the result into memory.

```
s = gather(s)
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 2.1 sec
Evaluation completed in 2.4 sec

s = 123523
```

Calculate Mean Values of Tall Vectors

Create a tall table, extract two tall vectors form the table, and then calculate the mean value of each vector.

Create a tall table for the `airlinesmall.csv` data set. The data contains information about arrival and departure times of US flights. Extract the `ArrDelay` and `DepDelay` variables, which are vectors of arrival and departure delays.

```
ds = tabularTextDatastore('airlinesmall.csv','TreatAsMissing','NA');
ds.SelectedVariableNames = {'ArrDelay' 'DepDelay'};
tt = tall(ds);
tt = rmmissing(tt);
tX = tt.ArrDelay;
tY = tt.DepDelay;
```

In the first stage of the algorithm, calculate the sum and element count for each block of data in the vectors. To do this you can write a function that accepts two inputs and returns one output with the sum and count for each input. This function is listed as a local function at the end of the example.

```
function bx = sumcount(tx,ty)
  bx = [sum(tx) numel(tx) sum(ty) numel(ty)];
end
```

In the reduction stage of the algorithm, you need to add together all of the intermediate sums and counts. Thus, `matlab.tall.reduce` returns the overall sum of elements and number of elements for each input vector, and calculating the mean is then a simple division. For this step you can apply the `sum` function to the first dimension of the 1-by-4 vector outputs from the first stage.

```
 reducefcn = @(x) sum(x,1);
 s = matlab.tall.reduce(@sumcount,reducefcn,tX,tY)
```

```
s =

  MxNx... tall double array

    ?    ?    ?    ...
    ?    ?    ?    ...
    ?    ?    ?    ...
    :    :    :
    :    :    :
```

```
s = gather(s)
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 2.1 sec
Evaluation completed in 2.2 sec

s = 1×4

      860584      120866      982764      120866
```

The first two elements of `s` are the sum and count for `tX`, and the second two elements are the sum and count for `tY`. Dividing the sums and counts yields the mean values, which you can compare to the answer returned by the `mean` function.

```
my_mean = [s(1)/s(2) s(3)/s(4)]
```

```
my_mean = 1×2

    7.1201    8.1310
```

```
m = gather(mean([tX tY]))
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 0.99 sec
Evaluation completed in 1.2 sec

m = 1×2

    7.1201    8.1310
```

Local Functions

Listed here is the `sumcount` function that `matlab.tall.reduce` calls to calculate the intermediate sums and element counts.

```
function bx = sumcount(tx,ty)
  bx = [sum(tx) numel(tx) sum(ty) numel(ty)];
end
```

Calculate Statistics by Group

Create a tall table, then calculate the mean flight delay for each year in the data.

Create a tall table for the `airlinesmall.csv` data set. The data contains information about arrival and departure times of US flights. Remove rows of missing data from the table and extract the `ArrDelay`, `DepDelay`, and `Year` variables. These variables are vectors of arrival and departure delays and of the associated years for each flight in the data set.

```
ds = tabularTextDatastore('airlinesmall.csv','TreatAsMissing','NA');
ds.SelectedVariableNames = {'ArrDelay' 'DepDelay' 'Year'};
tt = tall(ds);
tt = rmmissing(tt);
```

Use `matlab.tall.reduce` to apply two functions to the tall table. The first function combines the `ArrDelay` and `DepDelay` variables to find the total mean delay for each flight. The function determines how many unique years are in each chunk of data, and then cycles through each year and calculates the average total delay for flights in that year. The result is a two-variable table containing the year and mean total delay. This intermediate data needs to be reduced further to arrive at the mean delay per year. Save this function in your current folder as `transform_fcn.m`.

```
type transform_fcn
```

```
function t = transform_fcn(a,b,c)
ii = gather(unique(c));

for k = 1:length(ii)
    jj = (c == ii(k));
    d = mean([a(jj) b(jj)], 2);

    if k == 1
```

```
        t = table(c(jj),d,'VariableNames',{'Year' 'MeanDelay'});
    else
        t = [t; table(c(jj),d,'VariableNames',{'Year' 'MeanDelay'})];
    end
end

end
```

The second function uses the results from the first function to calculate the mean total delay for each year. The output from `reduce_fcn` is compatible with the output from `transform_fcn`, so that blocks of data can be concatenated in any order and continually reduced until only one row remains for each year.

```
type reduce_fcn
```

```
function TT = reduce_fcn(t)
[groups,Y] = findgroups(t.Year);
D = splitapply(@mean, t.MeanDelay, groups);

TT = table(Y,D,'VariableNames',{'Year' 'MeanDelay'});
end
```

Apply the transform and reduce functions to the tall vectors. Since the inputs (type `double`) and outputs (type `table`) have different data types, use the `'OutputsLike'` name-value pair to specify that the output is a table. A simple way to specify the type of the output is to call the transform function with dummy inputs.

```
a = tt.ArrDelay;
b = tt.DepDelay;
c = tt.Year;
d1 = matlab.tall.reduce(@transform_fcn, @reduce_fcn, a, b, c, 'OutputsLike',{transform_
```

```
d1 =

  Mx2 tall table

    Year    MeanDelay
    ____    _________

     ?          ?
     ?          ?
     ?          ?
     :          :
     :          :
```

Gather the results into memory to see the mean total flight delay per year.

```
d1 = gather(d1)
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 2.9 sec
Evaluation completed in 3.4 sec
```

```
d1=22×2 table
    Year    MeanDelay
    ____    _________

    1987     7.6889
    1988     6.7918
    1989     8.0757
    1990     7.1548
    1991     4.0134
    1992     5.1767
    1993     5.4941
    1994     6.0303
    1995     8.4284
    1996     9.6981
    1997     8.4346
    1998     8.3789
    1999     8.9121
    2000     10.595
    2001     6.8975
    2002     3.4325
       ⋮
```

Alternative Approach

Another way to calculate the same statistics by group is to use `splitapply` to call `matlab.tall.reduce` (rather than using `matlab.tall.reduce` to call `splitapply`).

Using this approach, you call `findgroups` and `splitapply` directly on the data. The function `mySplitFcn` that operates on each group of data includes a call to `matlab.tall.reduce`. The transform and reduce functions employed by `matlab.tall.reduce` do not need to group the data, so those functions just perform calculations on the pregrouped data that `splitapply` passes to them.

```
type mySplitFcn
```

```
function T = mySplitFcn(a,b,c)
T = matlab.tall.reduce(@non_group_transform_fcn, @non_group_reduce_fcn, ...
```

```
    a, b, c, 'OutputsLike', {non_group_transform_fcn(0,0,0)});

    function t = non_group_transform_fcn(a,b,c)
        d = mean([a b], 2);
        t = table(c,d,'VariableNames',{'Year' 'MeanDelay'});
    end

    function TT = non_group_reduce_fcn(t)
        D = mean(t.MeanDelay);
        TT = table(t.Year(1),D,'VariableNames',{'Year' 'MeanDelay'});
    end

end
```

Call `findgroups` and `splitapply` to operate on the data and apply `mySplitFcn` to each group of data.

```
groups = findgroups(c);
d2 = splitapply(@mySplitFcn, a, b, c, groups);
d2 = gather(d2)
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 2: Completed in 0.88 sec
- Pass 2 of 2: Completed in 2.1 sec
Evaluation completed in 3.8 sec

d2=22×2 table
    Year    MeanDelay
    ____    _________

    1987     7.6889
    1988     6.7918
    1989     8.0757
    1990     7.1548
    1991     4.0134
    1992     5.1767
    1993     5.4941
    1994     6.0303
    1995     8.4284
    1996     9.6981
    1997     8.4346
    1998     8.3789
    1999     8.9121
    2000     10.595
    2001     6.8975
```

```
    2002     3.4325
      ⋮
```

Weighted Standard Deviation and Variance of Tall Vectors

Calculate weighted standard deviation and variance of a tall array using a vector of weights. This is one example of how you can use `matlab.tall.reduce` to work around functionality that tall arrays do not support yet.

Create two tall vectors of random data. `tX` contains random data, and `tP` contains corresponding probabilities such that `sum(tP)` is `1`. These probabilities are suitable to weight the data.

```
rng default
tX = tall(rand(1e4,1));
p = rand(1e4,1);
tP = tall(normalize(p,'scale',sum(p)));
```

Write an identity function that returns outputs equal to the inputs. This approach skips the transform step of `matlab.tall.reduce` and passes the data directly to the reduction step, where the reduction function is repeatedly applied to reduce the size of the data.

```
type identityTransform.m
```

```
function [A,B] = identityTransform(X,Y)
  A = X;
  B = Y;
end
```

Next, write a reduction function that operates on blocks of the tall vectors to calculate the weighted variance and standard deviation.

```
type weightedStats.m
```

```
function [wvar, wstd] = weightedStats(X, P)
  wvar = var(X,P);
  wstd = std(X,P);
end
```

Use `matlab.tall.reduce` to apply these functions to the blocks of data in the tall vectors.

```
[tX_var_weighted, tX_std_weighted] = matlab.tall.reduce(@identityTransform, @weightedSt
```

```
tX_var_weighted =

  MxNx... tall double array

    ?    ?    ?    ...
    ?    ?    ?    ...
    ?    ?    ?    ...
    :    :    :
    :    :    :


tX_std_weighted =

  MxNx... tall double array

    ?    ?    ?    ...
    ?    ?    ?    ...
    ?    ?    ?    ...
    :    :    :
    :    :    :
```

Input Arguments

`fcn` — Transform function to apply
function handle | anonymous function

Transform function to apply, specified as a function handle or anonymous function. Each output of `fcn` must be the same type as the first input `tX`. You can use the `'OutputsLike'` option to return outputs of different data types. If `fcn` returns more than one output, then the outputs must all have the same height.

The general functional signature of `fcn` is

```
[a, b, c, ...] = fcn(x, y, z, ...)
```

`fcn` must satisfy these requirements:

1. **Input Arguments** — The inputs `[x, y, z, ...]` are blocks of data that fit in memory. The blocks are produced by extracting data from the respective tall array inputs `[tX, tY, tZ, ...]`. The inputs `[x, y, z, ...]` satisfy these properties:

- All of `[x, y, z, ...]` have the same size in the first dimension after any allowed expansion.
- The blocks of data in `[x, y, z, ...]` come from the same index in the tall dimension, assuming the tall array is nonsingleton in the tall dimension. For example, if `tX` and `tY` are nonsingleton in the tall dimension, then the first set of blocks might be `x = tX(1:20000,:)` and `y = tY(1:20000,:)`.
- If the first dimension of any of `[tX, tY, tZ, ...]` has a size of `1`, then the corresponding block `[x, y, z, ...]` consists of all the data in that tall array.

2 **Output Arguments** — The outputs `[a, b, c, ...]` are blocks that fit in memory, to be sent to the respective outputs `[tA, tB, tC, ...]`. The outputs `[a, b, c, ...]` satisfy these properties:

- All of `[a, b, c, ...]` must have the same size in the first dimension.
- All of `[a, b, c, ...]` are vertically concatenated with the respective results of previous calls to `fcn`.
- All of `[a, b, c, ...]` are sent to the same index in the first dimension in their respective destination output arrays.

3 **Functional Rules** — `fcn` must satisfy the functional rule:

- `F([inputs1; inputs2]) == [F(inputs1); F(inputs2)]`: Applying the function to the concatenation of the inputs should be the same as applying the function to the inputs separately and then concatenating the results.

4 **Empty Inputs** — Ensure that `fcn` can handle an input that has a height of 0. Empty inputs can occur when a file is empty or if you have done a lot of filtering on the data.

For example, this function accepts two input arrays, squares them, and returns two output arrays:

```
function [xx,yy] = sqInputs(x,y)
xx = x.^2;
yy = y.^2;
end
```

After you save this function to an accessible folder, you can invoke the function to square `tX` and `tY` and find the maximum value with this command:

```
tA = matlab.tall.reduce(@sqInputs, @max, tX, tY)
```

Example: `tC = matlab.tall.reduce(@numel,@sum,tX,tY)` finds the number of elements in each block, and then it sums the results to count the total number of elements.

Data Types: `function_handle`

`reducefcn` — Reduction function to apply
function handle | anonymous function

Reduction function to apply, specified as a function handle or anonymous function. Each output of `reducefcn` must be the same type as the first input `tX`. You can use the `'OutputsLike'` option to return outputs of different data types. If `reducefcn` returns more than one output, then the outputs must all have the same height.

The general functional signature of `reducefcn` is

```
[rA, rB, rC, ...] = reducefcn(a, b, c, ...)
```

`reducefcn` must satisfy these requirements:

1. **Input Arguments** — The inputs `[a, b, c, ...]` are blocks that fit in memory. The blocks of data are either outputs returned by `fcn`, or a partially reduced output from `reducefcn` that is being operated on again for further reduction. The inputs `[a, b, c, ...]` satisfy these properties:
   - The inputs `[a, b, c, ...]` have the same size in the first dimension.
   - For a given index in the first dimension, every row of the blocks of data `[a, b, c, ...]` either originates from the input, or originates from the same previous call to `reducefcn`.
   - For a given index in the first dimension, every row of the inputs `[a, b, c, ...]` for that index originates from the same index in the first dimension.
2. **Output Arguments** — All outputs `[rA, rB, rC, ...]` must have the same size in the first dimension. Additionally, they must be vertically concatenable with the respective inputs `[a, b, c, ...]` to allow for repeated reductions when necessary.
3. **Functional Rules** — `reducefcn` must satisfy these functional rules (up to roundoff error):
   - `F(input) == F(F(input))`: Applying the function repeatedly to the same inputs should not change the result.
   - `F([input1; input2]) == F([input2; input1])`: The result should not depend on the order of concatenation.
   - `F([input1; input2]) == F([F(input1); F(input2)])`: Applying the function once to the concatenation of some intermediate results should be the same as applying it separately, concatenating, and applying it again.

4 **Empty Inputs** — Ensure that `reducefcn` can handle an input that has a height of 0. Empty inputs can occur when a file is empty or if you have done a lot of filtering on the data. For this call, all input blocks are empty arrays of the correct type and size in dimensions beyond the first.

Some examples of suitable reduction functions are built-in dimension reduction functions such as `sum`, `prod`, `max`, and so on. These functions can work on intermediate results produced by `fcn` and return a single scalar. These functions have the properties that the order in which concatenations occur and the number of times the reduction operation is applied do not change the final answer. Some functions, such as `mean` and `var`, should generally be avoided as reduction functions because the number of times the reduction operation is applied can change the final answer.

Example: `tC = matlab.tall.reduce(@numel,@sum,tX)` finds the number of elements in each block, and then it sums the results to count the total number of elements.

Data Types: `function_handle`

`tX, tY` — Input arrays
scalars | vectors | matrices | multidimensional arrays

Input arrays, specified as scalars, vectors, matrices, or multidimensional arrays. The input arrays are used as inputs to the transform function `fcn`. Each input array `tX,tY,...` must have compatible heights. Two inputs have compatible height when they have the same height, or when one input is of height one.

`PA, PB` — Prototype of output arrays
arrays

Prototype of output arrays, specified as arrays. When you specify `'OutputsLike'`, the output arrays `tA,tB,...` returned by `matlab.tall.reduce` have the same data types and attributes as the specified arrays `{PA,PB,...}`.

Example: `tA = matlab.tall.reduce(fcn,reducefcn,tX,'OutputsLike',{int8(1)});`, where `tX` is a double-precision tall array, returns `tA` as `int8` instead of `double`.

Output Arguments

`tA, tB` — Output arrays
scalars | vectors | matrices | multidimensional arrays

Output arrays, returned as scalars, vectors, matrices, or multidimensional arrays. If any input to `matlab.tall.reduce` is tall, then all output arguments are also tall. Otherwise, all output arguments are in-memory arrays.

The size and data type of the output arrays depend on the specified functions `fcn` and `reducefcn`. In general, the outputs `tA,tB,...` must all have the same data type as the first input `tX`. However, you can specify `'OutputsLike'` to return different data types. The output arrays `tA,tB,...` all have the same height.

More About

Tall Array Blocks

When you create a tall array from a datastore, the underlying datastore facilitates the movement of data during a calculation. The data moves in discrete pieces called *blocks* or *chunks*, where each block is a set of consecutive rows that can fit in memory. For example, one block of a 2-D array (such as a table) is `X(n:m,:)`, for some subscripts `n` and `m`. The size of each block is based on the value of the `ReadSize` property of the datastore, but the block might not be exactly that size. For the purposes of `matlab.tall.reduce`, a tall array is considered to be the vertical concatenation of many such blocks:

For example, if you use the `sum` function as the transform function, the intermediate result is the sum *per block*. Therefore, instead of returning a single scalar value for the sum of the elements, the result is a vector with length equal to the number of blocks.

```
ds = tabularTextDatastore('airlinesmall.csv','TreatAsMissing','NA');
ds.SelectedVariableNames = {'ArrDelay' 'DepDelay'};
tt = tall(ds);
tX = tt.ArrDelay;
```

```
f = @(x) sum(x,'omitnan');
s = matlab.tall.reduce(f, @(x) x, tX);
s = gather(s)
```

```
s =

      140467
      101065
      164355
      135920
      111182
      186274
       21321
```

See Also

`matlab.tall.transform` | `tall`

Topics

"Develop Custom Tall Array Algorithms"

Introduced in R2018b

matlab.tall.movingWindow

Package: `matlab.tall`

Apply moving window function to blocks of data

Syntax

```
tA = matlab.tall.movingWindow(fcn,window,tX)
[tA,tB,...] = matlab.tall.movingWindow(fcn,window,tX,tY,...)
[ ___ ] = matlab.tall.movingWindow( ___ ,Name,Value)
```

Description

`tA = matlab.tall.movingWindow(fcn,window,tX)` applies the function `fcn` once per window as the window moves over the first dimension of `tX`. The output `tA` is the vertical concatenation of the results of applying `fcn` to each window.

`[tA,tB,...] = matlab.tall.movingWindow(fcn,window,tX,tY,...)`, where `fcn` is a function handle that returns multiple outputs, returns arrays `tA,tB,...`, each corresponding to one of the output arguments of `fcn`. The inputs to `fcn` are windows of data from the arguments `tX, tY, ...`. This syntax has these requirements:

- `fcn` must return the same number of outputs as were requested from `matlab.tall.movingWindow`.
- Each output of `fcn` must be the same type as the first data input `tX`.
- All outputs `tA,tB,...` must have the same height.

`[ ___ ] = matlab.tall.movingWindow( ___ ,Name,Value)` specifies additional options with one or more name-value pair arguments using any of the previous syntaxes. For example, to adjust the step size between windows, you can specify `'Stride'` and a scalar. Or to change the treatment of endpoints where there are not enough elements to complete a window, you can specify `'EndPoints'` and a valid option (`'shrink'`, `'discard'`, or a numeric padding value).

Examples

Moving Window Calculation with Tall Array

Use `matlab.tall.movingWindow` to calculate the moving median of airline arrival and departure delays.

Create a datastore for the `airlinesmall.csv` data set and convert it into a tall array. The data contains information about arrival and departure times of US flights. Extract the `ArrDelay` and `DepDelay` variables, which are vectors of flight delays, to create a tall array containing the delays as separate columns.

```
varnames = {'ArrDelay', 'DepDelay'};
ds = datastore('airlinesmall.csv', 'TreatAsMissing', 'NA', ...
    'SelectedVariableNames', varnames);
tt = tall(ds);
tX = [tt.ArrDelay tt.DepDelay]
```

```
tX =

  Mx2 tall double matrix

     8    12
     8     1
    21    20
    13    12
     4    -1
    59    63
     3    -2
    11    -1
    :     :
    :     :
```

Use `matlab.tall.movingWindow` to calculate the moving median of the data in the first dimension. Use a window size of 5,000.

```
fcn = @(x) median(x,1,'omitnan');
tA = matlab.tall.movingWindow(fcn,5000,tX)
```

```
tA =

  MxNx... tall double array
```

```
    ?    ?    ?    ...
    ?    ?    ?    ...
    ?    ?    ?    ...
    :    :    :
    :    :    :
```

Gather the unique rows of the result into memory.

```
tA = gather(unique(tA,'rows'))
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 2: Completed in 1.8 sec
- Pass 2 of 2: Completed in 1 min 27 sec
Evaluation completed in 1 min 29 sec

tA = 31×2

   -4.0000   -2.0000
   -3.5000   -2.0000
   -3.0000   -2.0000
   -3.0000   -1.5000
   -3.0000   -1.0000
   -3.0000   -0.5000
   -3.0000         0
   -2.5000   -1.0000
   -2.5000         0
   -2.0000   -1.0000
      ⋮
```

Apply Window Function with Multiple Outputs

Use `matlab.tall.movingWindow` to apply a function with multiple outputs to windows of data.

Create a tall array from an in-memory random matrix.

```
X = rand(1000,5);
tX = tall(X)
```

```
tX =
```

```
  1,000x5 tall double matrix

    0.8147    0.6312    0.7449    0.3796    0.4271
    0.9058    0.3551    0.8923    0.3191    0.9554
    0.1270    0.9970    0.2426    0.9861    0.7242
    0.9134    0.2242    0.1296    0.7182    0.5809
    0.6324    0.6525    0.2251    0.4132    0.5403
    0.0975    0.6050    0.3500    0.0986    0.7054
    0.2785    0.3872    0.2871    0.7346    0.0050
    0.5469    0.1422    0.9275    0.6373    0.7825
      :         :         :         :         :
      :         :         :         :         :
```

Create a function that finds the sum, mean, median, and mode of each window of data in the first dimension. Each output needs to have the same size in the first dimension, but the other dimensions can have different sizes. For each window of data, the sum calculation produces a scalar, while the other calculations produce 1-by-N vectors.

Save the function in your local workspace.

```
function [S,mn,mdn,md] = mystats(X)
  S = sum(X,[2 1]);
  mn = mean(X,1);
  mdn = median(X,1);
  md = mode(X,1);
end
```

Note: This function is included at the end of the example as a local function.

Use `matlab.tall.movingWindow` to apply the `mystats` function to the data with a window size of 250. Specify four output arguments to return all of the outputs from `mystats`. Use the `'EndPoints'` name-value pair to discard incomplete windows.

```
[tS,tmn,tmdn,tmd] = matlab.tall.movingWindow(@mystats, 250, tX, 'EndPoints', 'discard')
```

```
tS =

  MxNx... tall double array

    ?    ?    ?    ...
    ?    ?    ?    ...
    ?    ?    ?    ...
    :    :    :
    :    :    :
```

```
tmn =

  MxNx... tall double array

    ?    ?    ?    ...
    ?    ?    ?    ...
    ?    ?    ?    ...
    :    :    :
    :    :    :


tmdn =

  MxNx... tall double array

    ?    ?    ?    ...
    ?    ?    ?    ...
    ?    ?    ?    ...
    :    :    :
    :    :    :


tmd =

  MxNx... tall double array

    ?    ?    ?    ...
    ?    ?    ?    ...
    ?    ?    ?    ...
    :    :    :
    :    :    :
```

```
function [S,mn,mdn,md] = mystats(X)
  S = sum(X,[2 1]);
  mn = mean(X,1);
  mdn = median(X,1);
  md = mode(X,1);
end
```

Input Arguments

`fcn` — Window function to apply
function handle | anonymous function

Window function to apply, specified as a function handle or anonymous function. Each output of `fcn` must be the same type as the first input `tX`. You can use the `'OutputsLike'` option to return outputs of different data types.

The general functional signature of `fcn` is

```
[a, b, c, ...] = fcn(x, y, z, ...)
```

`fcn` must satisfy these requirements:

1. **Input Arguments** — The inputs `[x, y, z, ...]` are blocks of data that fit in memory. The blocks are produced by extracting data from the respective tall array inputs `[tX, tY, tZ, ...]`. The inputs `[x, y, z, ...]` satisfy these properties:

   - All of the inputs `[x, y, z, ...]` have the same size in the first dimension.
   - The blocks of data in `[x, y, z, ...]` come from the same index in the tall dimension, assuming the tall array is nonsingleton in the tall dimension. For example, if `tX` and `tY` are nonsingleton in the tall dimension, then the first set of blocks might be `x = tX(1:20000,:)` and `y = tY(1:20000,:)`.
   - When the first dimension of any of `[tX, tY, tZ, ...]` has a size of `1`, the corresponding block `[x, y, z, ...]` consists of all the data in that tall array.
   - Applying `fcn` must result in a reduction of the input data to a scalar or a slice of an array of height 1.

     When the input is a matrix, N-D array, table, or timetable, applying `fcn` must result in a reduction of the input data in each of its columns or variables.

2. **Output Arguments** — The outputs `[a, b, c, ...]` are blocks that fit in memory, to be sent to the respective outputs `[tA, tB, tC, ...]`. The outputs `[a, b, c, ...]` satisfy these properties:

   - All of the outputs `[a, b, c, ...]` must have the same size in the first dimension.
   - All of the outputs `[a, b, c, ...]` are vertically concatenated with the respective results of previous calls to `fcn`.

- All of the outputs `[a, b, c, ...]` are sent to the same index in the first dimension in their respective destination output arrays.

3 **Functional Rules** — `fcn` must satisfy the functional rule:

- `F([inputs1; inputs2]) == [F(inputs1); F(inputs2)]`: Applying the function to the concatenation of the inputs should be the same as applying the function to the inputs separately and then concatenating the results.

For example, this function calculates the mean and standard deviation of the elements in a window and returns two output arrays:

```
function [mv,sd] = movstats(tX)
mv = mean(tX,1,'omitnan');
sd = std(tX,1,'omitnan');
end
```

After you save this function to an accessible folder, you can invoke the function with a window size of 5 using this command:

```
[tA,tB] = matlab.tall.movingWindow(@movstats,5,tX)
```

Example: `tA = matlab.tall.movingWindow(@(x) std(x,1,'omitnan'), tX)` specifies an anonymous function to calculate the standard deviation of each window, ignoring `NaN`s.

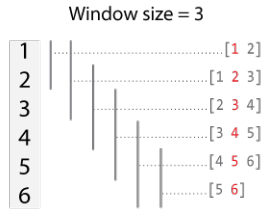
Example: `tA = matlab.tall.movingWindow(@mean,3,tX)` specifies a function handle `@mean` to calculate the mean value of each three-element window.

Data Types: `function_handle`

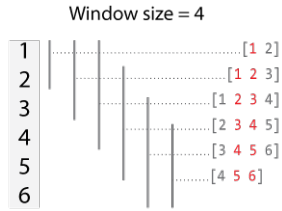
`window` — Window size
positive integer scalar | two-element row vector

Window size, specified as a positive integer scalar or a two-element row vector `[NB NF]`.

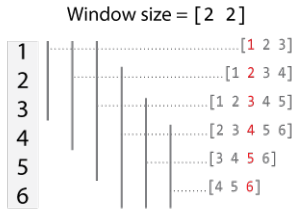
- If `window` is a scalar, then:
  - When the window size is **odd**, each window is centered on the corresponding element in the data.



- When the window size is **even**, each window is centered about the current and previous elements.



- If `window` is a vector `[NB NF]`, then the window includes the previous `NB` elements, the current element, and the next `NF` elements of the inputs.



By default, the window size is automatically truncated at the endpoints when not enough elements are available to fill the window. When the window is truncated in this manner, the function operates only on the elements that fill the window. You can change this behavior with the `EndPoints` name-value pair.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`tX, tY` — Input arrays (as separate arguments)

scalars | vectors | matrices | multidimensional arrays | tables | timetables

Input arrays, specified as separate arguments of scalars, vectors, matrices, multidimensional arrays, tables, or timetables. The input arrays can be tall or in-memory

arrays. The input arrays are used as inputs to the transform function `fcn`. Each input array `tX,tY,...` must have the same height.

Name-Value Pair Arguments

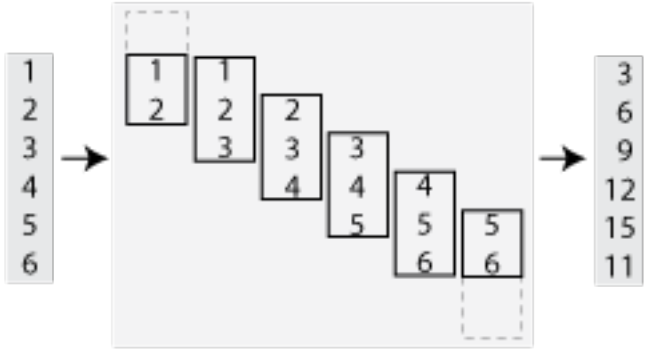
Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `tA = matlab.tall.movingWindow(@myFcn, window, tX, 'Stride', 2)`

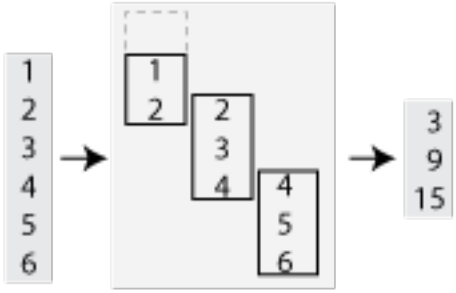
`Stride` — Step size between windows
1 (default) | positive integer scalar

Step size between windows, specified as the comma-separated pair consisting of `'Stride'` and a positive integer scalar. After `fcn` operates on a window of data, the calculation advances by the `'Stride'` value before operating on the next window. Increasing the value of `'Stride'` from the default value of 1 is the same as reducing the size of the output by picking out every other element, or every third element, and so on.

By default, the value of `'Stride'` is `1`, so that each window is centered on each element in the input. For example, here is a moving sum calculation with a window size of 3 operating on the vector `[1 2 3 4 5 6]'`:



If the value of `'Stride'` is `2`, then the calculation changes so that each window is centered on every second element in the input (1, 3, 5). The moving sum now returns three partial sums rather than six:

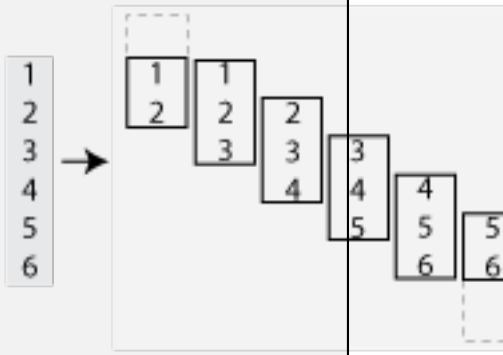


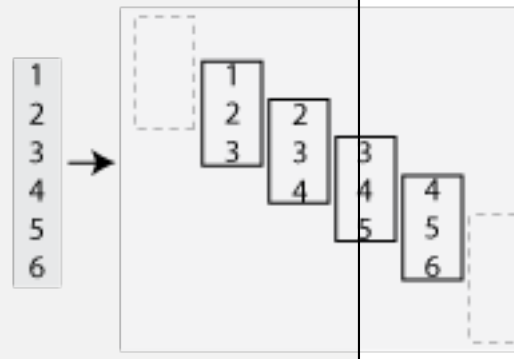
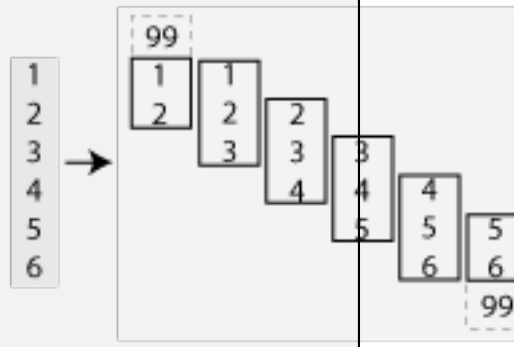
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`EndPoints` — Method to treat leading and trailing windows
`'shrink'` (default) | `'discard'` | padding value

Method to treat leading and trailing windows, specified as the comma-separated pair consisting of `'EndPoints'` and one of the values in the table.

At the beginning and end of a windowed calculation, the window of elements being operated on is incomplete. The `'EndPoints'` option specifies how to treat these incomplete windows.

`'EndPoints'` Value	Description	Example: Moving Sum
`'shrink'`	Shrink the window size near the endpoints of the input to include only existing elements.	

'EndPoints' Value	Description	Example: Moving Sum
'discard'	Do not output any results where the window does not completely overlap with existing elements.	
Numeric or logical padding value	Substitute nonexisting elements with a specified numeric or logical value. • The padding value must have the same type as tX. • The size of the padding value in the first dimension must be equal to 1, and the size in other dimensions must match tX.	

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char | string

OutputsLike — Prototype of output arrays
cell array

Prototype of output arrays, specified as the comma-separated pair consisting of 'OutputsLike' and a cell array containing prototype arrays. When you specify 'OutputsLike', the output arrays tA,tB,... returned by matlab.tall.movingWindow have the same data types and attributes as the specified prototype arrays {PA,PB,...}. You must specify 'OutputsLike' whenever the data type of an output array is different than that of the input array. If you specify 'OutputsLike', then you must specify a prototype array for each output.

The prototype arrays `{PA,PB,...}` that you specify must have the same data type and non-tall dimension sizes as the corresponding output arrays.

Example: `tA = matlab.tall.movingWindow(..., tX, 'OutputsLike', {int8(1)});`, where `tX` is a double-precision tall array, returns `tA` as `int8` instead of `double`.

Data Types: `cell`

Output Arguments

`tA, tB` — Output arrays
scalars | vectors | matrices | multidimensional arrays

Output arrays, returned as scalars, vectors, matrices, or multidimensional arrays. If any input to `matlab.tall.movingWindow` is tall, then all output arguments are also tall. Otherwise, all output arguments are in-memory arrays.

- The size and data type of the output arrays depend on the specified window function `fcn`.
- The output arrays `tA,tB,...` all have the same height, which depends on the value of `'Stride'` and `'EndPoints'`. By default the output arrays are the same size as the input arrays.
- In general, the outputs `tA,tB,...` must all have the same data type as the first input `tX`. However, you can specify `'OutputsLike'` to return different data types. In cases where the input arrays `tX, tY, ...` are empty, or when `'EndPoints'` is `'discard'` and there are not enough elements to fill a full-sized window, `matlab.tall.movingWindow` returns empty outputs. The sizes of the empty outputs are based on the size of the input array `tX`, or on the sizes of the prototype arrays provided to `'OutputsLike'`, if specified.

Tips

- Use `matlab.tall.movingWindow` for simple sliding-window calculations. `matlab.tall.blockMovingWindow` is an advanced API designed to provide more flexibility to perform sliding-window calculations on tall arrays. As such, it is more complicated to use since the functions must accurately process blocks of data that contain many complete windows. However, with properly vectorized calculations, you can reduce the necessary number of function calls and improve performance.

See Also

`matlab.tall.blockMovingWindow` | `tall`

Topics

"Develop Custom Tall Array Algorithms"

Introduced in R2019a

matlab.tall.blockMovingWindow

Package: `matlab.tall`

Apply moving window function and block reduction to padded blocks of data

Syntax

```
tA = matlab.tall.blockMovingWindow(windowfcn,blockfcn,window,tX)
[tA,tB,...] = matlab.tall.blockMovingWindow(windowfcn,blockfcn,
window,tX,tY,...)
[ ___ ] = matlab.tall.blockMovingWindow( ___ ,Name,Value)
```

Description

`tA = matlab.tall.blockMovingWindow(windowfcn,blockfcn,window,tX)` applies `blockfcn` to complete windows of data and `windowfcn` to incomplete windows of data near the edges. `window` specifies the size of the sliding window. The result contains the vertical concatenation of applying `blockfcn` and `windowfcn` to these windows of data.

`[tA,tB,...] = matlab.tall.blockMovingWindow(windowfcn,blockfcn, window,tX,tY,...)`, where `windowfcn` and `blockfcn` are function handles that return multiple outputs, returns arrays `tA, tB, ...`, each corresponding to one of the output arguments of `windowfcn` and `blockfcn`. The inputs to `windowfcn` and `blockfcn` are pieces of data from the arguments `tX, tY, ...`. This syntax has these requirements:

- `windowfcn` and `blockfcn` must return the same number of outputs as were requested from `matlab.tall.blockMovingWindow`.
- Each output of `windowfcn` and `blockfcn` must be the same type as the first data input `tX`.
- All outputs `tA,tB,...` must have the same height.

`[ ___ ] = matlab.tall.blockMovingWindow( ___ ,Name,Value)` specifies additional options with one or more name-value pair arguments using any of the previous

syntaxes. For example, to adjust the step size between windows, you can specify `'Stride'` and a scalar. Or to change the treatment of endpoints where there are not enough elements to complete a window, you can specify `'EndPoints'` and a valid option (`'shrink'`, `'discard'`, or a numeric padding value).

Examples

Moving Window Calculation with Tall Array

Use `matlab.tall.blockMovingWindow` to calculate the moving mean of airline arrival and departure delays.

Create a datastore for the `airlinesmall.csv` data set and convert it into a tall array. The data contains information about arrival and departure times of US flights. Extract the `ArrDelay` and `DepDelay` variables, which are vectors of flight delays, to create a tall array containing the delays as separate columns.

```
varnames = {'ArrDelay', 'DepDelay'};
ds = datastore('airlinesmall.csv', 'TreatAsMissing', 'NA', ...
    'SelectedVariableNames', varnames);
tt = tall(ds);
tX = [tt.ArrDelay tt.DepDelay]
```

```
tX =

  Mx2 tall double matrix

     8    12
     8     1
    21    20
    13    12
     4    -1
    59    63
     3    -2
    11    -1
    :     :
    :     :
```

Use `matlab.tall.blockMovingWindow` to calculate the moving mean of the data in the first dimension with a window size of 10. Since `windowfcn` applies only to single windows of data, you can use the `mean` function to reduce the windows of data down into

a matrix with one row. The `blockfcn` applies to whole blocks of data, so use the `movmean` function to calculate the mean of each full window of data in the blocks.

```
windowfcn = @(info,x) mean(x,1,'omitnan');
blockfcn = @(info,x) movmean(x,info.Window,1,'omitnan','EndPoints','discard');
A = matlab.tall.blockMovingWindow(windowfcn, blockfcn, 10, tX)
```

```
A =

  MxNx... tall double array

    ?    ?    ?    ...
    ?    ?    ?    ...
    ?    ?    ?    ...
    :    :    :
    :    :    :
```

Gather a portion of the results into memory.

```
gather(A(1:10,:))
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 2: Completed in 1.5 sec
- Pass 2 of 2: Completed in 8.4 sec
Evaluation completed in 11 sec

ans = 10×2

   10.8000    8.8000
   18.8333   17.8333
   16.5714   15.0000
   15.8750   13.0000
   14.4444   11.8889
   13.2000   10.8000
   14.0000   11.1000
   13.5000   11.9000
   15.3000   11.4000
   19.7000   13.4000
```

Moving Table Statistics

Calculate moving statistics on the variables of a table.

Load the `outages.csv` data set as a tall table. The data contains information about power outages.

```
T = tall(readtable('outages.csv'))
```

```
T =

  1,468×6 tall table

      Region           OutageTime        Loss     Customers     RestorationTime
    ____________    _________________    ______    __________    _________________

    {'SouthWest'}    2002-02-01 12:18    458.98    1.8202e+06    2002-02-07 16:50    {
    {'SouthEast'}    2003-01-23 00:49    530.14    2.1204e+05                  NaT    {
    {'SouthEast'}    2003-02-07 21:15     289.4    1.4294e+05    2003-02-17 08:14    {
    {'West'     }    2004-04-06 05:44    434.81    3.4037e+05    2004-04-06 06:10    {
    {'MidWest'  }    2002-03-16 06:18    186.44    2.1275e+05    2002-03-18 23:23    {
    {'West'     }    2003-06-18 02:49         0             0    2003-06-18 10:54    {
    {'West'     }    2004-06-20 14:39    231.29           NaN    2004-06-20 19:16    {
    {'West'     }    2002-06-06 19:28    311.86           NaN    2002-06-07 00:51    {
          :                  :              :           :                 :
          :                  :              :           :                 :
```

Use `matlab.tall.blockMovingWindow` to apply a moving-window function to blocks of the tall table. Specify these options:

- `blkstats` as the block function to operate on complete blocks of data (included at the end of the example as a local function).
- A window size of 50 and a stride of 5.
- `EndPoints` as `'discard'` to ignore incomplete windows of data. With this value, the `windowfcn` input can be specified as empty `[]` since only complete windows of data are operated on.
- The input table has six variables, but the two outputs are double-precision vectors. Specify scalar doubles as the value for `OutputsLike` so that the function permits this change in data type and size.

```
[A, B] = matlab.tall.blockMovingWindow([], @blkstats, 50, T, 'Stride', 5, ...
    'EndPoints', 'discard', 'OutputsLike', {1, 1});
```

Preview a few rows in the results.

```
[A,B] = gather(head(A),head(B))
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 2: Completed in 0.16 sec
- Pass 2 of 2: Completed in 0.24 sec
Evaluation completed in 0.63 sec

A = 8×1

  254.0861
  254.0861
  340.3499
  452.0191
  464.8524
  471.9737
  464.8524
  464.8524


B = 8×1
10^5 ×

    1.3447
    1.0779
    1.4227
    1.4509
    1.2888
    1.2888
    1.2308
    1.3722
```

The `blkstats` function calculates the moving median value of the `Loss` and `Customers` table variables in the first dimension using the specified window size. The function applies the `Stride` value to reduce the size of the output, and then it returns the results as two vectors.

```
function [out1, out2] = blkstats(info, t)
    a = movmedian([t.Loss t.Customers], info.Window, 1, 'omitnan', 'EndPoints', 'disca
    a = a(1:info.Stride:end, :);
    out1 = a(:,1);
    out2 = a(:,2);
end
```

Input Arguments

`windowfcn` — Function to apply to incomplete windows of data

function handle | anonymous function | [ ]

Function to apply to incomplete windows of data, specified as a function handle, anonymous function, or `[]`. `windowfcn` is invoked once per incomplete window as the calculation moves over data in the tall dimension. `matlab.tall.blockMovingWindow` applies `windowfcn` only when `'EndPoints'` has the default value of `'shrink'`. If you specify a different value for `'EndPoints'`, then set `windowfcn` to `[]`.

Each output of `windowfcn` must be the same type as the first data input `tX`. You can use the `'OutputsLike'` option to return outputs of different data types.

The general functional signature of `windowfcn` is

```
[a, b, c, ...] = windowfcn(info, x, y, ...)
```

The `info` input is a structure provided by `matlab.tall.blockMovingWindow` that includes these fields:

- `Stride` — Specified step size between windows (default: 1). Set this value with the `'Stride'` name-value pair.
- `Window` — Specified window size. Set this value with the `window` input argument.

`windowfcn` must satisfy these requirements:

1. **Input Arguments** — The inputs `[x, y, z, ...]` are blocks of data that fit in memory. The blocks are produced by extracting data from the respective tall array inputs `[tX, tY, tZ, ...]`. The inputs `[x, y, z, ...]` satisfy these properties:
   - All of the inputs `[x, y, z, ...]` have the same size in the first dimension.
   - The blocks of data in `[x, y, z, ...]` come from the same index in the tall dimension, assuming the tall array is nonsingleton in the tall dimension. For example, if `tX` and `tY` are nonsingleton in the tall dimension, then the first set of blocks might be `x = tX(1:20000,:)` and `y = tY(1:20000,:)`.
   - When the first dimension of any of `[tX, tY, tZ, ...]` has a size of `1`, the corresponding block `[x, y, z, ...]` consists of all the data in that tall array.
   - Applying `windowfcn` must result in a reduction of the input data to a scalar or a slice of an array of height 1.

When the input is a matrix, N-D array, table, or timetable, applying `windowfcn` must result in a reduction of the input data in each of its columns or variables.

2 **Output Arguments** — The outputs `[a, b, c, ...]` are blocks that fit in memory to be sent to the respective outputs `[tA, tB, tC, ...]`. The outputs `[a, b, c, ...]` satisfy these properties:

- All of the outputs `[a, b, c, ...]` must have the same size in the first dimension.
- All of the outputs `[a, b, c, ...]` are vertically concatenated with the respective results of previous calls to `windowfcn`.
- All of the outputs `[a, b, c, ...]` are sent to the same index in the first dimension in their respective destination output arrays.

3 **Functional Rules** — `windowfcn` must satisfy this functional rule:

- `F([inputs1; inputs2]) == [F(inputs1); F(inputs2)]`: Applying the function to the concatenation of the inputs should be the same as applying the function to the inputs separately and then concatenating the results.

Example: `A = matlab.tall.blockMovingWindow(@windowfcn, @blockfcn, 10, tX)`

Example: `A = matlab.tall.blockMovingWindow([], @blockfcn, 10, tX, 'EndPoints', 'discard')`

Data Types: `function_handle`

`blockfcn` — Function to apply to blocks of data
function handle | anonymous function

Function to apply to blocks of data, specified as a function handle or anonymous function. `blockfcn` is applied to blocks of data that contain complete windows of data. Thus, `blockfcn` must operate in a vectorized manner on entire blocks of data and return output that has the proper size for the specified window size and stride.

Each output of `blockfcn` must be the same type as the first data input `tX`. You can use the `'OutputsLike'` option to return outputs of different data types.

`matlab.tall.blockMovingWindow` applies `blockfcn` to blocks of data whenever the block contains only complete windows:

- For middle blocks when `'EndPoints'` is set to `'shrink'` (default behavior). In this case `windowfcn` operates on the incomplete windows of data on the ends.

- For all blocks when `'EndPoints'` is set to `'discard'` or a padding value.

The general functional signature of `blockfcn` is

```
[a, b, c, ...] = blockfcn(info, bX, bY, bZ, ...)
```

The `info` input is a structure provided by `matlab.tall.blockMovingWindow` that includes these fields:

- `Stride` — Specified step size between windows (default: 1). Set this value with the `'Stride'` name-value pair.
- `Window` — Specified window size. Set this value with the `window` input argument.

The blocks of data `bX, bY, bZ, ...` that `matlab.tall.blockMovingWindow` provides to `blockfcn` have these properties:

- The blocks contain only full-sized windows. `blockfcn` does not have to define a behavior for incomplete windows of data.
- The first window of data starts at the first element of the block. The last element of the last window is the last element of the block.

`blockfcn` must satisfy these requirements:

1. **Input Arguments** — The inputs `[bX, bY, bZ, ...]` are blocks of data that fit in memory. The blocks are produced by extracting data from the respective tall array inputs `[tX, tY, tZ, ...]`. The inputs `[bX, bY, bZ, ...]` satisfy these properties:

   - All of the inputs `[bX, bY, bZ, ...]` have the same size in the first dimension after any allowed expansion.
   - The blocks of data in `[bX, bY, bZ, ...]` come from the same index in the tall dimension, assuming the tall array is nonsingleton in the tall dimension. For example, if `tX` and `tY` are nonsingleton in the tall dimension, then the first set of blocks might be `bX = tX(1:20000,:)` and `bY = tY(1:20000,:)`.
   - If the first dimension of any of the data inputs `[tX, tY, tZ, ...]` has a size of `1`, then the corresponding block `[bX, bY, bZ, ...]` consists of all the data in that tall array.
   - Applying `blockfcn` must result in a reduction of the input data such that the result has height equal to the number of windows in the block. You can use `info.Window` and `info.Stride` to determine the number of windows in a block.

If the input is a matrix, N-D array, table, or timetable, then applying `blockfcn` must result in a reduction of the input data in each of its columns or variables.

2. **Output Arguments** — The outputs `[a, b, c, ...]` are blocks that fit in memory, to be sent to the respective outputs `[tA, tB, tC, ...]`. The outputs `[a, b, c, ...]` satisfy these properties:

   - All of the outputs `[a, b, c, ...]` must have the same size in the first dimension.
   - All of the outputs `[a, b, c, ...]` are vertically concatenated with the respective results of previous calls to `blockfcn`.
   - All of the outputs `[a, b, c, ...]` are sent to the same index in the first dimension in their respective destination output arrays.

3. **Functional Rules** — `blockfcn` must satisfy this functional rule:

   - `F([inputs1; inputs2]) == [F(inputs1); F(inputs2)]`: Applying the function to the concatenation of the inputs should be the same as applying the function to the inputs separately and then concatenating the results.

Example: `A = matlab.tall.blockMovingWindow(@windowfcn, @blockfcn, 10, tX)`

Example: `A = matlab.tall.blockMovingWindow([], @blockfcn, 10, tX, 'EndPoints', 'discard')`

Data Types: `function_handle`

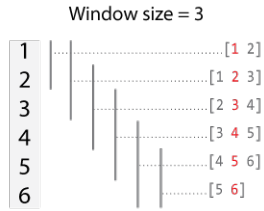
`window` — Window size

positive integer scalar | two-element row vector

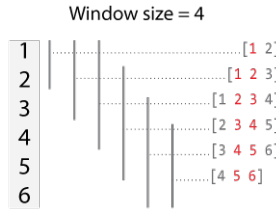
Window size, specified as a positive integer scalar or a two-element row vector `[NB NF]`.

- If `window` is a scalar, then:

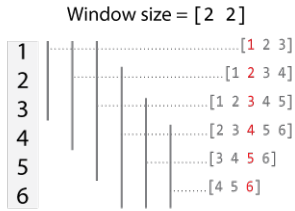
  - When the window size is **odd**, each window is centered on the corresponding element in the data.



- When the window size is **even**, each window is centered about the current and previous elements.



- If `window` is a vector `[NB NF]`, then the window includes the previous `NB` elements, the current element, and the next `NF` elements of the inputs.



By default, the window size is automatically truncated at the endpoints when not enough elements are available to fill the window. When the window is truncated in this manner, the function operates only on the elements that fill the window. You can change this behavior with the `EndPoints` name-value pair.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`tX, tY` — Input arrays (as separate arguments)

scalars | vectors | matrices | multidimensional arrays | tables | timetables

Input arrays, specified as separate arguments of scalars, vectors, matrices, multidimensional arrays, tables, or timetables. The input arrays can be tall or in-memory

arrays. The input arrays are used as inputs to the transform function `fcn`. Each input array `tX,tY,...` must have the same height.

Name-Value Pair Arguments

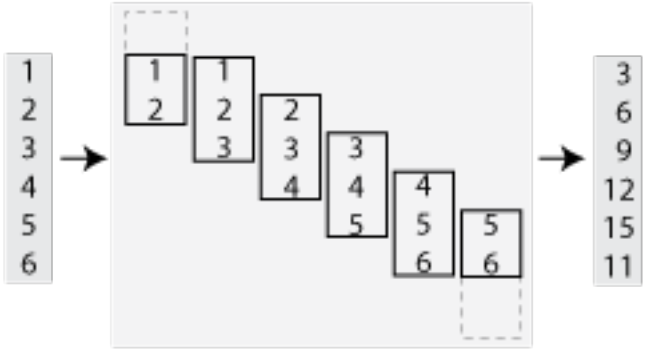
Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `tA = matlab.tall.blockMovingWindow(@windowfcn, blockfcn, window, tX, 'Stride', 2)`

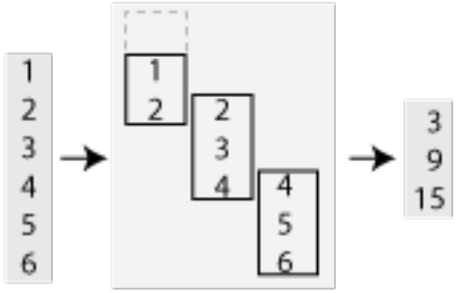
`Stride` — Step size between windows
1 (default) | positive integer scalar

Step size between windows, specified as the comma-separated pair consisting of `'Stride'` and a positive integer scalar. After `fcn` operates on a window of data, the calculation advances by the `'Stride'` value before operating on the next window. Increasing the value of `'Stride'` from the default value of 1 is the same as reducing the size of the output by picking out every other element, or every third element, and so on.

By default, the value of `'Stride'` is `1`, so that each window is centered on each element in the input. For example, here is a moving sum calculation with a window size of 3 operating on the vector `[1 2 3 4 5 6]'`:



If the value of `'Stride'` is `2`, then the calculation changes so that each window is centered on every second element in the input (1, 3, 5). The moving sum now returns three partial sums rather than six:

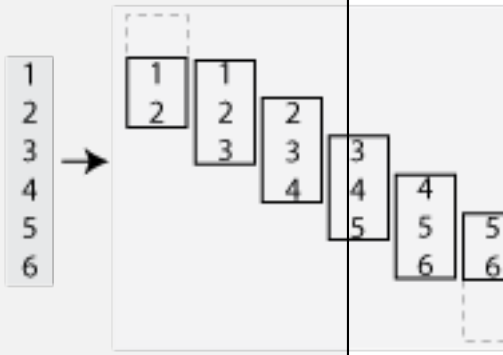


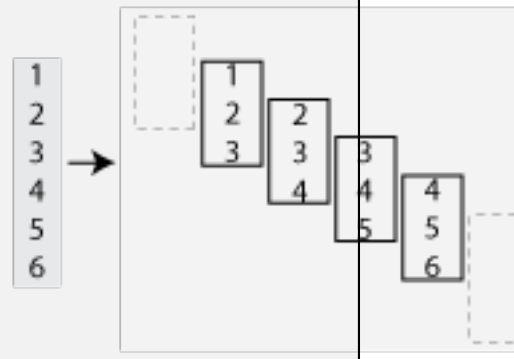
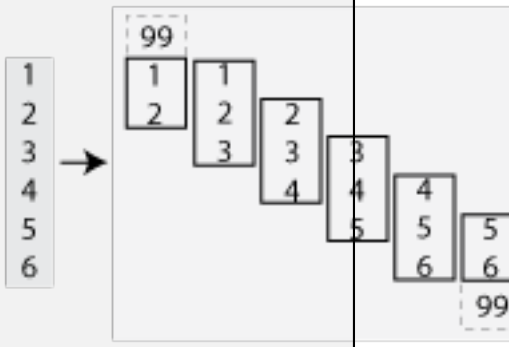
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`EndPoints` — Method to treat leading and trailing windows
`'shrink'` (default) | `'discard'` | padding value

Method to treat leading and trailing windows, specified as the comma-separated pair consisting of `'EndPoints'` and one of the values in the table.

At the beginning and end of a windowed calculation, the window of elements being operated on is incomplete. The `'EndPoints'` option specifies how to treat these incomplete windows.

`'EndPoints'` Value	Description	Example: Moving Sum
`'shrink'`	Shrink the window size near the endpoints of the input to include only existing elements.	

'EndPoints' Value	Description	Example: Moving Sum
'discard'	Do not output any results where the window does not completely overlap with existing elements.	
Numeric or logical padding value	Substitute nonexisting elements with a specified numeric or logical value. • The padding value must have the same type as tX. • The size of the padding value in the first dimension must be equal to 1, and the size in other dimensions must match tX.	

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char | string

OutputsLike — Prototype of output arrays
cell array

Prototype of output arrays, specified as the comma-separated pair consisting of 'OutputsLike' and a cell array containing prototype arrays. When you specify 'OutputsLike', the output arrays tA,tB,... returned by matlab.tall.movingWindow have the same data types and attributes as the specified prototype arrays {PA,PB,...}. You must specify 'OutputsLike' whenever the data type of an output array is different than that of the input array. If you specify 'OutputsLike', then you must specify a prototype array for each output.

The prototype arrays `{PA,PB,...}` that you specify must have the same data type and non-tall dimension sizes as the corresponding output arrays.

Example: `tA = matlab.tall.blockMovingWindow(..., tX, 'OutputsLike', {int8(1)});`, where `tX` is a double-precision tall array, returns `tA` as `int8` instead of `double`.

Data Types: `cell`

Output Arguments

`tA, tB` — Output arrays
scalars | vectors | matrices | multidimensional arrays

Output arrays, returned as scalars, vectors, matrices, or multidimensional arrays. If any input to `matlab.tall.blockMovingWindow` is tall, then all output arguments are also tall. Otherwise, all output arguments are in-memory arrays.

- The size and data type of the output arrays depend on the specified window functions `windowfcn` and `blockfcn`.
- The output arrays `tA,tB,...` all have the same height, which depends on the value of `'Stride'` and `'EndPoints'`. By default the output arrays are the same size as the input arrays.
- In general, the outputs `tA,tB,...` must all have the same data type as the first data input `tX`. However, you can specify `'OutputsLike'` to return different data types. In cases where the input arrays `tX, tY, ...` are empty, or when `'EndPoints'` is `'discard'` and there are not enough elements to fill a full-sized window, `matlab.tall.blockMovingWindow` returns empty outputs. The sizes of the empty outputs are based on the size of the input array `tX`, or on the sizes of the prototype arrays provided to `'OutputsLike'`, if specified.

Tips

- Use `matlab.tall.movingWindow` for simple sliding-window calculations. `matlab.tall.blockMovingWindow` is an advanced API designed to provide more flexibility to perform sliding-window calculations on tall arrays. As such, it is more complicated to use since the functions must accurately process blocks of data that contain many complete windows. However, with properly vectorized calculations, you can reduce the necessary number of function calls and improve performance.

See Also

`matlab.tall.movingWindow` | `tall`

Topics

"Develop Custom Tall Array Algorithms"

Introduced in R2019a

tall

Create tall array

Syntax

```
t = tall(ds)
t = tall(A)
```

Description

`t = tall(ds)` creates a tall array on top of datastore `ds`.

- If `ds` is a datastore for tabular data (so that the `read` and `readall` methods of datastore return tables), then `t` is a tall table. Tabular data is data that is arranged in a rectangular fashion with each row having the same number of entries.
- Otherwise, `t` is a tall cell array.

`t = tall(A)` converts the in-memory array `A` into a tall array. The underlying data type of `t` is the same as `class(A)`. This syntax is useful when you need to quickly create a tall array, such as for debugging or prototyping algorithms.

In R2019b and later, you can cast in-memory arrays into tall arrays for more efficient operations on the array. After you convert into a tall array, MATLAB avoids making temporary copies of the data for most operations. This enables you to perform a wider range of operations on the array without running out of memory.

Examples

Create Tall Array

Convert a datastore into a tall array.

First, create a datastore for the data set. You can specify either a full or relative file location for the data set using `datastore(location)` to create the datastore. The `location` argument can specify:

- A single file, such as `'airlinesmall.csv'`
- Several files with the same extension, such as `'*.csv'`
- An entire folder of files, such as `'C:\MyData'`

`datastore` also has several options to specify file and text format properties when you create the datastore.

Create a datastore for the `airlinesmall.csv` data set. Treat `'NA'` values as missing data so that they are replaced with `NaN` values. Select a small subset of the variables to work with.

```
varnames = {'ArrDelay', 'DepDelay', 'Origin', 'Dest'};
ds = datastore('airlinesmall.csv', 'TreatAsMissing', 'NA', ...
    'SelectedVariableNames', varnames);
```

Use `tall` to create a tall array for the data in the datastore. Since the data in `ds` is tabular, the result is a tall table. If the data is not tabular, then `tall` creates a tall cell array instead.

```
T = tall(ds)
```

```
T =

  Mx4 tall table

    ArrDelay    DepDelay    Origin      Dest
    ________    ________    _______    _______

        8           12       {'LAX'}    {'SJC'}
        8            1       {'SJC'}    {'BUR'}
       21           20       {'SAN'}    {'SMF'}
       13           12       {'BUR'}    {'SJC'}
        4           -1       {'SMF'}    {'LAX'}
       59           63       {'LAX'}    {'SJC'}
        3           -2       {'SAN'}    {'SFO'}
       11           -1       {'SEA'}    {'LAX'}
       :            :           :          :
       :            :           :          :
```

You can use many common MATLAB® operators and functions to work with tall arrays. To see if a function works with tall arrays, check the **Extended Capabilities** section at the bottom of the function reference page.

Calculate Size of Tall Array

Convert a datastore into a tall table, calculate its size using a deferred calculation, and then perform the calculation and return the result in memory.

First, create a datastore for the `airlinesmall.csv` data set. Treat `'NA'` values as missing data so that they are replaced with `NaN` values. Set the text format of a few columns so that they are read as a cell array of character vectors. Convert the datastore into a tall table.

```
ds = datastore('airlinesmall.csv', 'TreatAsMissing', 'NA');
ds.SelectedFormats{strcmp(ds.SelectedVariableNames, 'TailNum')} = '%s';
ds.SelectedFormats{strcmp(ds.SelectedVariableNames, 'CancellationCode')} = '%s';

T = tall(ds)
```

```
T =

  Mx29 tall table

    Year    Month    DayofMonth    DayOfWeek    DepTime    CRSDepTime    ArrTime    CRS
    ____    _____    __________    _________    _______    __________    _______    ___

    1987     10          21            3          642          630         735
    1987     10          26            1         1021         1020        1124
    1987     10          23            5         2055         2035        2218
    1987     10          23            5         1332         1320        1431
    1987     10          22            4          629          630         746
    1987     10          28            3         1446         1343        1547
    1987     10           8            4          928          930        1052
    1987     10          10            6          859          900        1134
     :        :           :            :           :            :           :
     :        :           :            :           :            :           :
```

The display of the tall table indicates that MATLAB® does not yet know how many rows of data are in the table.

Calculate the size of the tall table. Since calculating the size of a tall array requires a full pass through the data, MATLAB does not immediately calculate the value. Instead, like most operations with tall arrays, the result is an unevaluated tall array whose values and size are currently unknown.

```
s = size(T)
```

```
s =

  1x2 tall double row vector

    ?    ?
```

Use the `gather` function to perform the deferred calculation and return the result in memory. The result returned by `size` is a trivially small 1-by-2 vector, which fits in memory.

```
sz = gather(s)
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 1.9 sec
Evaluation completed in 2.1 sec

sz = 1×2

      123523          29
```

If you use `gather` on an unreduced tall array, then the result might not fit in memory. If you are unsure whether the result returned by `gather` can fit in memory, use `gather(head(X))` or `gather(tail(X))` to bring only a small portion of the calculation result into memory.

Convert In-Memory Arrays to Tall Arrays

Create an in-memory array of random numbers, and then convert it into a tall array. Creating tall arrays from in-memory arrays in this manner is useful for debugging or prototyping new programs. The in-memory array is still bound by normal memory constraints, and even after it is converted into a tall array it cannot grow beyond the limits of memory.

```
A = rand(100,4);
tA = tall(A)
```

```
tA =

  100x4 tall double matrix

    0.8147    0.1622    0.6443    0.0596
    0.9058    0.7943    0.3786    0.6820
    0.1270    0.3112    0.8116    0.0424
    0.9134    0.5285    0.5328    0.0714
    0.6324    0.1656    0.3507    0.5216
    0.0975    0.6020    0.9390    0.0967
    0.2785    0.2630    0.8759    0.8181
    0.5469    0.6541    0.5502    0.8175
      :         :         :         :
      :         :         :         :
```

In R2019b and later releases, when you convert in-memory arrays into tall arrays, you can perform calculations on the array without requiring extra memory for temporary copies of the data. For example, this code normalizes the data in a large matrix and then calculates the sum of all the rows and columns. An in-memory version of this calculation needs to not only store the 5GB array but also have enough memory available to create temporary copies of the array.

```
N = 25000;
tA = tall(rand(N));
tB = tA - mean(tA);
S = gather(sum(tB, [1,2]))
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 2: Completed in 10 sec
- Pass 2 of 2: Completed in 25 sec
Evaluation completed in 37 sec

S = 2.2646e-10
```

Input Arguments

ds — Input datastore
datastore object

Input datastore, specified as a datastore object. Use the `datastore` function to create a datastore object for your data set.

Tall arrays work only with datastores that are deterministic. That is, if you use `read` on the datastore, reset the datastore with `reset`, and then read the datastore again, then the data returned must be the same in both cases. Tall array calculations involving a datastore that is not deterministic can produce unpredictable results. See "Select Datastore for File Format or Application" for more information.

Example: `ds = datastore('airlinesmall.csv')` specifies a single file.

Example: `ds = datastore('*.csv')` specifies a collection of `.csv` files.

Example: `ds = datastore('C:\MyData')` specifies a folder of files.

Example: `ds = datastore('hdfs:///data/')` specifies a data set in an HDFS file system.

`A` — In-memory variable
array

In-memory variable, specified as an array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `table` | `timetable` | `string` | `cell` | `categorical` | `datetime` | `duration` | `calendarDuration`
Complex Number Support: Yes

Output Arguments

`t` — Tall array
array

Tall array, returned as one of these types:

- When converting a datastore, `t` is a tall table for tabular datastores. Otherwise, `t` is a tall cell array.
- When converting an in-memory array, the underlying data type of `t` is the same as `class(A)`.

See "Deferred Evaluation of Tall Arrays" for information about how to effectively work with tall arrays.

Tips

- See “Extend Tall Arrays with Other Products” for information on how to use tall arrays with:
  - Statistics and Machine Learning Toolbox
  - Parallel Computing Toolbox
  - MATLAB Parallel Server
  - Database Toolbox
  - MATLAB Compiler

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

If you have Parallel Computing Toolbox installed, then when you use `tall`, MATLAB automatically opens a parallel pool of workers on your local machine. MATLAB runs the computations across the available workers. Control parallel behavior with the parallel preferences, including scaling up to a cluster.

For details, see “Use Tall Arrays on a Parallel Pool” (Parallel Computing Toolbox).

See Also

`classUnderlying` | `datastore` | `gather` | `isaUnderlying` | `istall` | `mapreducer` | `topkrows` | `write`

Topics

"Tall Arrays for Out-of-Memory Data"
"Deferred Evaluation of Tall Arrays"
"Visualization of Tall Arrays"

Introduced in R2016b

tallrng

Control random number generation for tall arrays

Syntax

```
tallrng('default')
tallrng('shuffle')
tallrng(seed)
tallrng(seed,generator)
tallrng('shuffle',generator)
state = tallrng
tallrng(state)
```

Description

`tallrng('default')` puts the settings of the random number generator used in tall array calculations to their default values. The random numbers produced are the same as if you restart MATLAB.

`tallrng('shuffle')` sets the seed of the random number generator based on the current time. Use this syntax when you want different sequences of random numbers each time they are generated.

`tallrng(seed)` sets the starting point, or seed, of the random number generator. Use this syntax when you want to produce predictable sequences of numbers.

`tallrng(seed,generator)` or `tallrng('shuffle',generator)` selects the random number generator. For example, `tallrng(0,'mlfg6331_64')`.

`state = tallrng` returns the current state of the random number generator as a structure. Use the structure to restore the random number generator to the captured state at a later time with `tallrng(state)`.

`tallrng(state)` restores the state of the random number generator using settings previously captured with `state = tallrng`.

Examples

Capture and Restore Generator Settings

Capture the generator settings, generate an array of random numbers, and then restore the generator to the initial settings to create predictable arrays of random numbers.

Save the current state of the random number generator.

```
state = tallrng
```

```
state = struct with fields:
           Type: 'threefry'
           Seed: 0
    StreamIndex: 1
      Substream: 1
```

Create a tall array of random numbers using `arrayfun`.

```
t = tall(zeros(10,1));
n1 = arrayfun(@(x) x+randn, t)
```

```
n1 =

  10x1 tall double column vector

   -0.3479
    0.1057
    0.3969
    0.6544
   -1.8228
    0.9587
    0.5360
   -0.5779
      :
      :
```

Restore the generator to the previous state and regenerate the array of random numbers. The two arrays `n1` and `n2` are equal.

```
tallrng(state)
n2 = arrayfun(@(x) x+randn, t)
```

```
n2 =

  10x1 tall double column vector

   -0.3479
    0.1057
    0.3969
    0.6544
   -1.8228
    0.9587
    0.5360
   -0.5779
       :
       :
```

Input Arguments

seed — Random number seed
nonnegative integer

Random number seed, specified as a nonnegative integer. The seed specifies the starting point for the algorithm to generate random numbers. Specify the seed when you want reproducible results.

The maximum value of the seed is $2^{32}-1$.

Example: `tallrng(3)`

`generator` — Random number generator
`'threefry'` (default) | `'multFibonacci'` | `'philox'` | `'combRecursive'`

Random number generator, specified as one of the random number generators in this table. You can specify the regular or convenience name.

Name	Convenience Name	Description
`'threefry4x64_20'` (default)	`'threefry'`	Threefry 4x64 generator with 20 rounds
`'mrg32k3a'`	`'combRecursive'`	Combined multiple recursive generator

Name	Convenience Name	Description
`'mlfg6331_64'`	`'multFibonacci'`	Multiplicative lagged Fibonacci generator
`'philox4x32_10'`	`'philox'`	Philox 4x32 generator with 10 rounds

`tallrng` supports random number generators that have multiple stream and substream support. See “Choosing a Random Number Generator” for a complete list of available generators in MATLAB.

Example: `tallrng(0,'mlfg6331_64')`

`state` — Previous random number generator state
structure

Previous random number generator state, specified as a structure previously created using `state = tallrng`.

Example: `state = tallrng` captures the current state of the random number generator, and `tallrng(state)` restores the generator to those settings.

Data Types: `struct`

Output Arguments

`state` — Random number generator state
structure

Random number generator state, returned as a structure with these fields.

Field	Description	Example Value
`Type`	Type of random number generator	`'threefry'`
`Seed`	Number indicating the starting point for the algorithm (for reproducibility)	`0`

Field	Description	Example Value
`StreamIndex`	Index indicating the associated global stream	1
`Substream`	Index indicating the associated substream	1

Example: `state = tallrng` captures the current state of the random number generator, and `tallrng(state)` restores the generator to those settings.

Data Types: `struct`

Tips

- If you have Statistics and Machine Learning Toolbox, then `tallrng` controls the random numbers that functions such as `datasample`, `cvpartition`, and `TreeBagger` generate during tall array calculations.

Compatibility Considerations

Default random number generator change for `tallrng`

Behavior changed in R2019a

Starting in R2019a, the default random number generator for `tallrng` is `threefry`. This generator offers performance enhancements for parallel calculations over the old default. In releases up to R2018b, the default random number generator for `tallrng` was `combRecursive`.

With a different default generator, MATLAB will generate different sequences of random numbers by default in the context of tall arrays. However, the statistics of these calculations will remain unaffected. Therefore, you should update any code that relies on the *specific* random numbers being generated. However, most calculations on the random numbers should be unaffected.

To set the generator to the settings used by default in R2018b and earlier releases, use the command:

```
tallrng(0,'combRecursive')
```

See Also

`rng` | `tall`

Introduced in R2017b

tan

Tangent of argument in radians

Syntax

```
Y = tan(X)
```

Description

`Y = tan(X)` returns the tangent of each element of `X`. The `tan` function operates element-wise on arrays. The function accepts both real and complex inputs.

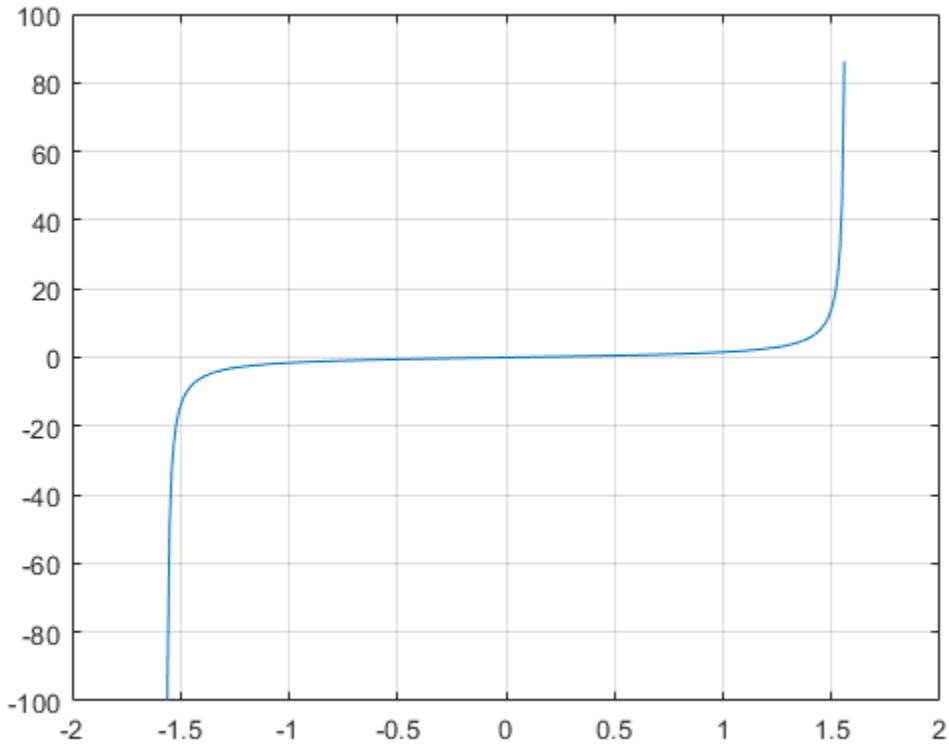
- For real values of `X`, `tan(X)` returns real values in the interval [-∞, ∞].
- For complex values of `X`, `tan(X)` returns complex values.

Examples

Plot Tangent Function

Plot the tangent function over the domain $-\pi/2 \le x \le \pi/2$.

```
x = (-pi/2)+0.01:0.01:(pi/2)-0.01;
plot(x,tan(x)), grid on
```



Tangent of Vector of Complex Angles

Calculate the tangent of the complex angles in vector `x`.

```
x = [-i pi+i*pi/2 -1+i*4];
y = tan(x)
```

```
y = 1×3 complex

   0.0000 - 0.7616i  -0.0000 + 0.9172i  -0.0006 + 1.0003i
```

Input Arguments

X — Input angle in radians
scalar | vector | matrix | multidimensional array

Input angle in radians, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Tangent of input angle
scalar | vector | matrix | multidimensional array

Tangent of input angle, returned as a real-valued or complex-valued scalar, vector, matrix or multidimensional array.

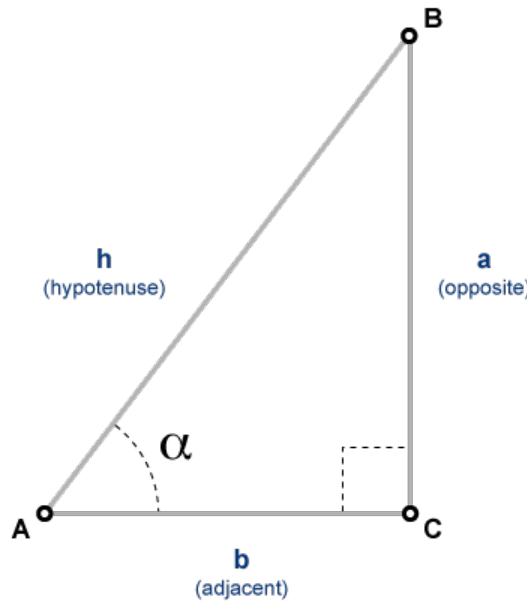
More About

Tangent Function

The tangent of an angle, α, defined with reference to a right angled triangle is

$$\tan(\alpha) = \frac{\text{opposite side}}{\text{adjacent side}} = \frac{a}{b} \ .$$

.



The tangent of a complex argument, α, is

$$\tan(\alpha) = \frac{e^{i\alpha} - e^{-i\alpha}}{i\left(e^{i\alpha} + e^{-i\alpha}\right)} .$$

.

Tips

- In floating-point arithmetic, `tan` is a bounded function. That is, `tan` does not return values of `Inf` or `-Inf` at points of divergence that are multiples of `pi`, but a large magnitude number instead. This stems from the inaccuracy of the floating-point representation of π.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`atan` | `atan2` | `atan2d` | `atand` | `tand` | `tanh`

Introduced before R2006a

tand

Tangent of argument in degrees

Syntax

```
Y = tand(X)
```

Description

`Y = tand(X)` returns the tangent of the elements of `X`, which are expressed in degrees.

Examples

Tangent of 90 degrees compared to tangent of π/2 radians

```
tand(90)
```

```
ans = Inf
```

```
tan(pi/2)
```

```
ans = 1.6331e+16
```

`tand(90)` is infinite, whereas `tan(pi/2)` is large but finite.

Tangent of vector of complex angles, specified in degrees

```
z = [180+i 15+2i 10+3i];
y = tand(z)
```

```
y = 1×3 complex
```

```
0.0000 + 0.0175i   0.2676 + 0.0374i   0.1758 + 0.0539i
```

Input Arguments

X — Angle in degrees
scalar value | vector | matrix | N-D array

Angle in degrees, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `tand` operation is element-wise when X is nonscalar.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

Y — Tangent of angle
scalar value | vector | matrix | N-D array

Tangent of angle, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as X.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- In some cases, generated code returns `-Inf` when MATLAB returns `Inf`.
- In some cases, generated code returns `Inf` when MATLAB returns `-Inf`.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`atan` | `atand` | `tan`

Introduced before R2006a

tanh

Hyperbolic tangent

Syntax

```
Y = tanh(X)
```

Description

`Y = tanh(X)` returns the hyperbolic tangent of the elements of `X`. The `tanh` function operates element-wise on arrays. The function accepts both real and complex inputs. All angles are in radians.

Examples

Hyperbolic Tangent of Vector

Create a vector and calculate the hyperbolic tangent of each value.

```
X = [0 pi 2*pi 3*pi];
Y = tanh(X)
```

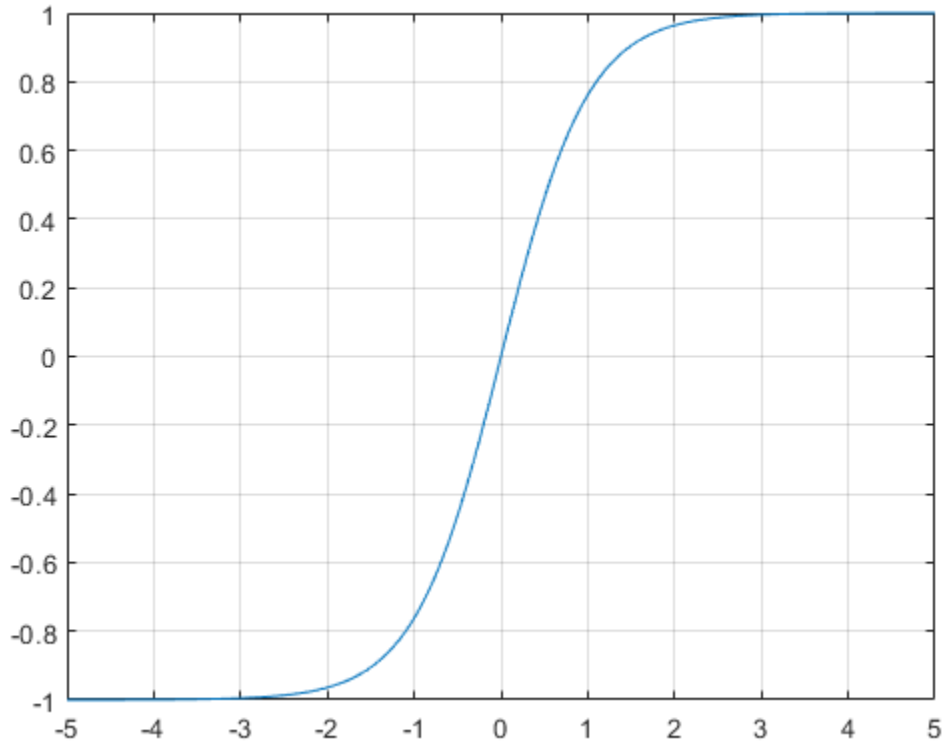
```
Y = 1×4

         0    0.9963    1.0000    1.0000
```

Graph of Hyperbolic Tangent Function

Plot the hyperbolic tangent function over the domain $-5 \le x \le 5$.

```
x = -5:0.01:5;
y = tanh(x);
```

```
plot(x,y)
grid on
```



Input Arguments

X — Input angles in radians
scalar | vector | matrix | multidimensional array

Input angles in radians, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Hyperbolic Tangent

The hyperbolic tangent of an angle x is the ratio of the hyperbolic sine and hyperbolic cosine

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^{2x} - 1}{e^{2x} + 1}.$$

In terms of the traditional tangent function with a complex argument, the identity is

$$\tanh(x) = -i\tan(ix).$$

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`atan` | `atan2` | `atanh` | `cosh` | `sinh` | `tan`

Introduced before R2006a

tar

Compress files into tar file

Syntax

```
tar(tarfilename,filenames)
tar(tarfilename,filenames,rootfolder)
entrynames = tar( ___ )
```

Description

`tar(tarfilename,filenames)` compresses contents of `filenames` into the tar file `tarfilename`. `tar` recursively compresses the content in folders. The resulting tar file contains the paths of `filenames` relative to the current folder. The tar file does not store absolute paths.

`tar(tarfilename,filenames,rootfolder)` specifies the paths for `filenames` relative to `rootfolder` rather than the current folder.

`entrynames = tar( ___ )` returns a cell array of character vectors containing the names of the files included in `tarfilename`. You can use this syntax with any of the input argument combinations in the previous syntaxes.

Examples

Compress Select Files to TAR Files

Compress all files with a `.m` and `.mlx` extension in the current folder to the file `backup.tar`.

```
tarfiles = tar('backup.tar',{'*.m','*.mlx'})
```

```
tarfiles = 1x1 cell array
    {'CompressFilesToTarExample.mlx'}
```

Create Tar File of a Folder

Create a tar file of a folder including all subfolders, and store the relative paths in the file.

Create a folder `myfolder` containing a subfolder `mysubfolder` and the files `membrane.m` and `logo.m`.

```
mkdir myfolder;
movefile('membrane.m','myfolder');
movefile('logo.m','myfolder');
cd myfolder;
mkdir mysubfolder;
cd ..
```

Create a tar file of the contents of `myfolder`, including all subfolders.

```
tarfiles = tar('myfiles.tar','myfolder');
```

Input Arguments

`tarfilename` — Name of tar file
character vector | string scalar

Name of tar file to create, specified as a character vector or a string scalar. If `tarfilename` has no extension, MATLAB appends the `.tar` extension. If you specify a `.tgz` or `.gz` extension, `tar` compresses the specified files using `gzip`.

`tarfilename` must include a path relative to the current folder or an absolute path.

Data Types: `char` | `string`

`filenames` — Names of files or folders
character vector | cell array of character vectors | string array

Names of files or folders to compress, specified as a character vector, a cell array of character vectors, or a string array. `filenames` must be smaller than 2 GB.

Files that are on the MATLAB path can include a partial path. Otherwise, files must include a path relative to the current folder or an absolute path.

Folders must include a path relative to the current folder or an absolute path. On UNIX systems, folders also can start with `~/` or `~username/`, which expands to the current user's home folder or the specified user's home folder, respectively. You can use the wildcard character * when specifying files or folders, except when relying on the MATLAB path to resolve a file name or partial path name.

Data Types: `char` | `string`

`rootfolder` — Root paths
character vector | string scalar

Root paths for the files to compress, specified as a character vector or a string scalar.

Data Types: `char` | `string`

See Also

`gunzip` | `gzip` | `untar` | `unzip` | `zip`

Introduced before R2006a

targetupdater

Set up support package that is already installed (Not recommended)

Note `targetupdater` is not recommended. To manage your support packages and setup your hardware, on the MATLAB® **Home** tab, in the **Environment** section, click **Add-Ons** > **Manage Add-Ons**.

Syntax

```
targetupdater
```

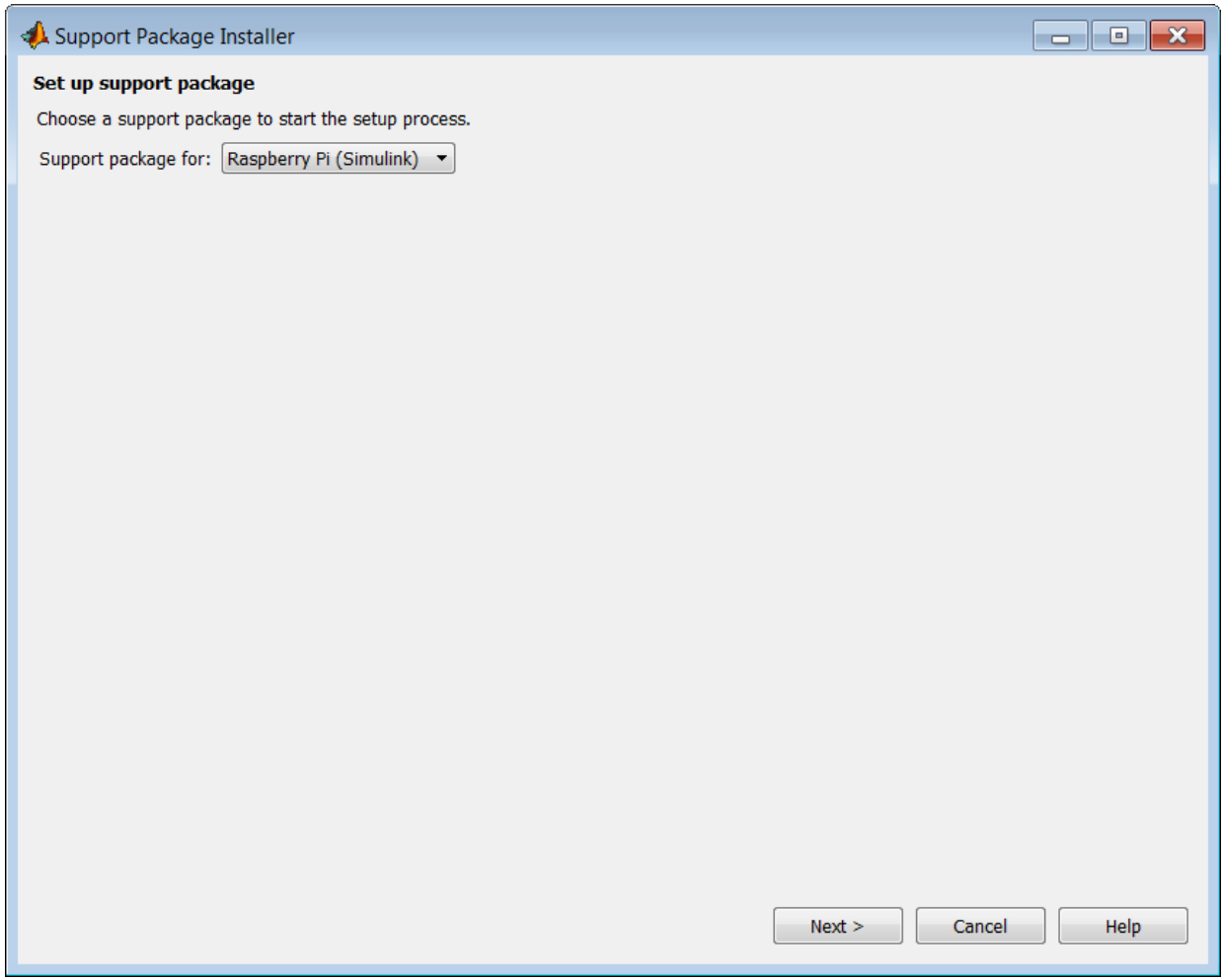
Description

`targetupdater` sets up a support package that is already installed.

Setting up a support package can include:

- Updating firmware on the target hardware.
- Validating the installation folder of third-party software.
- Validating licenses.

The `targetupdater` function starts Support Package Installer at the **Set up support package** screen.



Tip Some support packages do not require set up. If the **Set up support package** screen does not display an installed support package, the support package does not require set up.

To check whether a support package is installed, use the `matlabshared.supportpkg.getInstalled` function.

See Also

`matlabshared.supportpkg.checkForUpdate` | `matlabshared.supportpkg.getInstalled` | `supportPackageInstaller`

tcpclient

Create TCP/IP client object to communicate over TCP/IP

Syntax

```
t = tcpclient('Address',Port)
t = tcpclient('Address',Port,'Timeout',<timeout_value>)
t = tcpclient('Address',
Port,'ConnectTimeout',<connect_timeout_value>)
```

Description

`t = tcpclient('Address',Port)` constructs a TCP/IP object, `t`, associated with remote host, `Address`, and remote port value, `Port`. The address can be either a remote host name or a remote IP address. The port must be a positive integer between `1` and `65535`.

If an invalid address or port is specified, or the connection to the server cannot be established, the object will not be created.

`t = tcpclient('Address',Port,'Timeout',<timeout_value>)` additionally sets a timeout value. The `Timeout` property specifies the waiting time to complete read and write operations in seconds, and the default is `10`.

`t = tcpclient('Address',
Port,'ConnectTimeout',<connect_timeout_value>)` additionally sets a connection timeout value. The `ConnectTimeout` property specifies the maximum time in seconds to wait for a connection request to the specified remote host to succeed or fail. The default is `Inf`.

Examples

Create Object Using Host Name

Create the TCP/IP object `t` using the host address shown and Port of `80`.

```
t = tcpclient('www.mathworks.com', 80)
```

```
t = 

   tcpclient with properties:

          Address: 'www.mathworks.com'
             Port: 80
          Timeout: 10
   BytesAvailable: 0
   ConnectTimeout: Inf
```

Note When connecting using a host name, such as a specified web address or `'localhost'`, the IP address will default to an IPv6 format. If the server you are connecting to is expecting an IPv4 format, it will fail. If this happens, you can connect by specifying an explicit IP address, rather than a host name.

Create Object Using IP Address

Create the TCP/IP object `t` using the IP address shown and Port of `4012`.

```
t = tcpclient('172.28.154.231', 4012)
```

```
t = 

   tcpclient with properties:

          Address: '172.28.154.231'
             Port: 4012
          Timeout: 10
   BytesAvailable: 0
   ConnectTimeout: Inf
```

Create Object and Set the Timeout Property

Create the TCP/IP object `t` and increase the `Timeout` to `20` seconds.

```
t = tcpclient('172.28.154.231', 4012, 'Timeout', 20)
```

```
t = 

   tcpclient with properties:

           Address: '172.28.154.231'
              Port: 4012
           Timeout: 20
   BytesAvailable: 0
   ConnectTimeout: Inf
```

The output reflects the `Timeout` property change.

Create Object and Set the Connect Timeout Property

Create the TCP/IP object `t` and set the `ConnectTimeout` to `30` seconds.

```
t = tcpclient('172.28.154.231', 4012, 'ConnectTimeout', 30)
```

```
t = 

   tcpclient with properties:

           Address: '172.28.154.231'
              Port: 4012
           Timeout: 10
   BytesAvailable: 0
   ConnectTimeout: 30
```

The output reflects the `ConnectTimeout` property change.

Input Arguments

`Address` — Remote host name or IP address for connection
character vector

Remote host name or IP address for connection, specified as a character vector. Specify address as the first argument when you create the `tcpclient` object.

Example: `t = tcpclient('www.mathworks.com', 4012)`

Data Types: `char`

`Port` — Remote host port for connection
numeric scalar

Remote host port for connection, specified as a numeric scalar. Specify port number as the second argument when you create the `tcpclient` object. The `Port` must be a positive integer between `1` and `65535`.

Example: `t = tcpclient('www.mathworks.com', 4012)`

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `t = tcpclient('172.28.154.231', 120, 'Timeout', 15, 'ConnectTimeout', 30)`

`Timeout` — Timeout for read/write operation
10 (default) | numeric scalar

Timeout for read/write operation specified as the comma-separated pair consisting of `'Timeout'` and a positive value of type `double`. You can change the value either during object creation or after you create the object.

For information on how to change the timeout value after object creation, see Configure Properties for TCP/IP Communication.

Example: `t = tcpclient('172.28.154.231', 4012, 'Timeout', 20)`

Data Types: `double`

`ConnectTimeout` — Timeout for connection to remote host
`Inf` (default) | numeric scalar

Timeout for connection to remote host specified as the comma-separated pair consisting of `'ConnectTimeout'` and a positive value of type `double`. The property specifies the maximum time in seconds to wait for a connection request to the specified remote host to succeed or fail. You can only change the value during object creation.

Example: `t = tcpclient('172.28.154.231', 4012, 'ConnectTimeout', 30)`

Data Types: `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Topics

"Create a TCP/IP Connection"
"Configure Properties for TCP/IP Communication"
"Write and Read Data over TCP/IP Interface"

Introduced in R2014b

tempdir

Name of temporary folder for the system

Syntax

```
temp = tempdir
```

Description

`temp = tempdir` returns the name of the temporary folder for the system, if one exists. `tempdir` does not create a new folder.

Examples

Store Temporary Folder Name on Windows System

```
temp = tempdir
```

```
temp =

    'C:\Users\guest\AppData\Local\Temp\'
```

See Also

`delete` | `recycle` | `tempname`

Topics

"Create Temporary Files"

Introduced before R2006a

tempname

Unique name for temporary file

Syntax

```
tmpName = tempname
tmpName = tempname(folderName)
```

Description

`tmpName = tempname` returns a character vector containing a unique path for a temporary file in your system's temporary folder.

`tmpName = tempname(folderName)` returns a unique path for a temporary file in the specified folder.

Examples

Create Temporary File Name with Extension

Create a temporary file name that has the extension, `.dat`, by concatenating two character vectors.

```
tmpName = [tempname,'.dat'];
```

Create Temporary File Name in Folder

Create a temporary file name for use in the `C:\myTemporaryFiles` folder.

```
tmpName = tempname('C:\myTemporaryFiles');
```

Write Data to Temporary File

Create a temporary file name.

```
filename = tempname;
```

Create a new file with the temporary file name, and write data to the file.

```
fileID = fopen(filename,'w');
fwrite(fileID,magic(5));
fclose(fileID);
```

Input Arguments

`folderName` — Folder name
string scalar | character vector

Folder name, specified as a string scalar or a character vector containing the folder name.

Limitations

- In most cases, `tempname` generates a universally unique identifier (UUID). However, if you run MATLAB without JVM software, then `tempname` generates a random character vector using the CPU counter and time, which is not guaranteed to be unique. For more information about the MATLAB startup option that does not load JVM software, see “Commonly Used Startup Options”.

See Also

`tempdir`

Topics

“Create Temporary Files”

Introduced before R2006a

testsuite

Create suite of tests

Syntax

```
suite = testsuite
suite = testsuite(tests)
suite = testsuite(tests,Name,Value)
```

Description

`suite = testsuite` creates a suite of tests from your current folder, and returns the suite as a `Test` array.

To run a test suite created with `testsuite`, use the `run` method of `matlab.unittest.TestSuite`, `matlab.unittest.TestRunner`, or `matlab.perftest.TimeExperiment`.

`suite = testsuite(tests)` creates a suite from a set of specified tests.

`suite = testsuite(tests,Name,Value)` creates a suite of tests with additional options specified by one or more `Name,Value` pair arguments.

Examples

Test Suite from Working Folder

Create a folder `myExample` in your current working folder, make it your current working folder, and create a couple of tests.

In the `myExample` folder, create a script-based test, `onesTest.m`.

```
%% Test double class
expClass = 'double';
```

```
act = ones;
assert(isa(act,expClass))

%% Test single class
expClass = 'single';
act = ones('single');
assert(isa(act,expClass))

%% Test uint16 class
expClass = 'uint16';
act = ones('uint16');
assert(isa(act,expClass))

%% Test size
expSize = [7 13];
act = ones([7 13]);
assert(isequal(size(act),expSize))

%% Test values
act = ones(42);
assert(unique(act) == 1)
```

In the `myExample` folder, create a function-based test, `eyeTest.m`.

```
function tests = eyeTest
tests = functiontests(localfunctions);

function doubleClassTest(testCase)
actValue = eye;
verifyClass(testCase,actValue,'double')

function singleClassTest(testCase)
actValue = eye('single');
verifyClass(testCase,actValue,'single')

function uint16ClassTest(testCase)
actValue = eye('uint16');
verifyClass(testCase,actValue,'uint16')

function sizeTest(testCase)
expSize = [7 13];
actValue = eye(expSize);
verifySize(testCase,actValue,expSize);

function valueTest(testCase)
```

```
actValue = eye(42);
verifyEqual(testCase,unique(diag(actValue)),1)    % diagonal are 1s
verifyEqual(testCase,unique(triu(actValue,1)),0)  % upper tri vals are 0
verifyEqual(testCase,unique(tril(actValue,-1)),0) % lower tri vals are 0
```

Create a test suite from all tests in the current folder.

```
suite = testsuite
```

```
suite =

  1×10 Test array with properties:

    Name
    BaseFolder
    ProcedureName
    SharedTestFixtures
    Parameterization
    Tags

Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

If `onesTest` and `eyesTest` are the only tests in your folder, MATLAB creates a suite of 10 tests.

View the names of the tests in `suite`.

```
{suite.Name}'
```

```
ans =

    'eyeTest/doubleClassTest'
    'eyeTest/singleClassTest'
    'eyeTest/uint16ClassTest'
    'eyeTest/sizeTest'
    'eyeTest/valueTest'
    'onesTest/TestDoubleClass'
    'onesTest/TestSingleClass'
    'onesTest/TestUint16Class'
    'onesTest/TestSize'
    'onesTest/TestValues'
```

Create a test suite from all tests in `eyeTest`.

```
suite2 = testsuite('eyeTest')
```

```
suite2 =

  1×5 Test array with properties:

    Name
    BaseFolder
    ProcedureName
    SharedTestFixtures
    Parameterization
    Tags

Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

Test Suite from Select Tests

In your working folder, create a class-based test, `testZeros.m`. This class contains five test methods.

```
classdef testZeros < matlab.unittest.TestCase
    properties (TestParameter)
        type = {'single','double','uint16'};
        outSize = struct('s2d',[3 3], 's3d',[2 5 4]);
    end

    methods (Test)
        function testClass(testCase, type, outSize)
            testCase.verifyClass(zeros(outSize,type), type);
        end

        function testSize(testCase, outSize)
            testCase.verifySize(zeros(outSize), outSize);
        end

        function testDefaultClass(testCase)
            testCase.verifyClass(zeros, 'double');
        end
        function testDefaultSize(testCase)
            testCase.verifySize(zeros, [1 1]);
        end

        function testDefaultValue(testCase)
```

```
            testCase.verifyEqual(zeros,0);
        end
    end
end
```

The full test suite has 11 test elements: 6 from the `testClass` method, 2 from the `testSize` method, and 1 each from the `testDefaultClass`, `testDefaultSize`, and `testDefaultValue` methods.

Create a test suite from the test elements with test names that contain `'Default'`.

```
suite = testsuite('testZeros','Name','*Default*')
```

```
suite = 

  1x3 Test array with properties:

    Name
    ProcedureName
    TestClass
    BaseFolder
    Parameterization
    SharedTestFixtures
    Tags

Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

Create a test suite from the test elements that use the `outSize` parameter property.

```
suite = testsuite('testZeros','ParameterProperty','outSize')
```

```
suite = 

  1x8 Test array with properties:

    Name
    ProcedureName
    TestClass
    BaseFolder
    Parameterization
```

```
    SharedTestFixtures
    Tags

Tests Include:
   5 Unique Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

The test suite contains eight tests that use the `outSize` parameter property: six from the `testClass` method and two from the `testSize` method.

Input Arguments

`tests` — Array of tests
character vector | cell array of character vectors

Suite of tests specified as a character vector or cell array of character vectors. Each character vector in the cell array can contain the name of a test file, a test class, a test suite element name, a package containing your test classes, a folder containing your test files, or a project folder containing test files.

Example: `testsuite('ATestFile.m')`

Example: `testsuite('ATestFile/aTest')`

Example: `testsuite('mypackage.MyTestClass')`

Example: `testsuite(pwd)`

Example: `testsuite({'mypackage.MyTestClass','ATestFile.m',pwd,'mypackage.subpackage'})`

Example: `testsuite('C:/projects/project1/')`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `suite = testsuite(tests,'Name','productA_*')` creates a test suite from `tests` that have names starting with `'productA_'`.

`BaseFolder` — Name of base folder
character vector

Name of the folder that contains the file defining the test class, function, or script, specified as a character vector. This argument filters `TestSuite` array elements. To include a test element in the suite, the specified base folder must contain that test element. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match a single character. For test files defined in packages, the base folder is the parent of the top-level package folder.

`IncludeSubfolders` — Indicator to include tests in subfolders
`false` (default) | `true` | `0` | `1`

Indicator to include tests in subfolders in the suite, specified as `false` or `true` (`0` or `1`). By default the framework creates a suite from tests in the specified folders and not in their subfolders.

Data Types: `logical`

`IncludeSubpackages` — Indicator to include tests in subpackages
`false` (default) | `true` | `0` | `1`

Indicator to include tests in subpackages in the suite, specified as `false` or `true` (`0` or `1`). By default the framework creates a suite from tests in the specified package and not in their subpackages.

Data Types: `logical`

`IncludeReferencedProjects` — Indicator to include tests from referenced projects
`false` (default) | `true` | `0` | `1`

Indicator to include tests from referenced projects, specified as logical `false` or `true`. For more information on referenced projects, see “Componentize Large Projects”.

Data Types: `logical`

`Name` — Name of suite element
character vector

Name of the suite element, specified as a character vector. This argument filters `TestSuite` array elements. For the testing framework to include a test in the suite, the `Name` property of the test element must match the specified name. Use the wildcard

character, *, to match any number of characters. Use the question mark character, ?, to match a single character.

`ParameterName` — Name of parameter
character vector

Name of a parameter used by the test suite element, specified as a character vector. This argument filters `TestSuite` array elements. Use the wildcard character, *, to match any number of characters. Use the question mark character, ?, to match a single character.

`ParameterProperty` — Name of parameterization property
character vector

Name of a property that defines a parameter used by the test suite element, specified as a character vector. This argument filters `TestSuite` array elements. Use the wildcard character, *, to match any number of characters. Use the question mark character, ?, to match a single character.

`ProcedureName` — Name of the test procedure in the test
character vector

Name of test procedure, specified as a character vector. This argument filters `TestSuite` array elements. Use the wildcard character, *, to match any number of characters. Use the question mark character, ?, to match to exactly one character.

In a class-based test, the `ProcedureName` is the name of the test method. In a function-based, test it is the name of the local function that contains the test. In a script-based test, it is a name generated from the test section title. Unlike `Name`, the name of the test procedure does not include any class or package name or information about parameterization.

`Superclass` — Name of a class that the test class derives from
character vector | string scalar

Name of a class that the test class derives from, specified as a character vector or string scalar. This argument filters `TestSuite` array elements.

`Tag` — Name of test element tag
character vector | string scalar

Name of test element tag, specified as a character vector or string scalar. This argument filters `TestSuite` array elements. Use the wildcard character, *, to match any number of characters. Use the question mark character, ?, to match to exactly one character.

Tips

- If you do not need to create a test suite explicitly, use `runtests` or `runperf` to create the suite implicitly before running the tests.
- An alternative way to create an explicit test suite is to use the `matlab.unittest.TestSuite` methods.

See Also

`matlab.unittest.TestSuite` | `run` | `run` | `run`

Introduced in R2016a

tetramesh

Tetrahedron mesh plot

Syntax

```
tetramesh(T,X,c)
tetramesh(T,X)
tetramesh(TR)
h = tetramesh(...)
tetramesh(...,'param','value','param','value'...)
```

Description

`tetramesh(T,X,c)` displays the tetrahedrons defined in the `m`-by-4 matrix `T` as mesh. `T` is usually the output of a Delaunay triangulation of a 3-D set of points. A row of `T` contains indices into `X` of the vertices of a tetrahedron. `X` is an `n`-by-3 matrix, representing `n` points in 3 dimension. The tetrahedron colors are defined by the vector `C`, which is used as indices into the current colormap.

`tetramesh(T,X)` uses `C = 1:m` as the color for the `m` tetrahedra. Each tetrahedron has a different color (modulo the number of colors available in the current colormap).

`tetramesh(TR)` displays the tetrahedra in a triangulation representation.

`h = tetramesh(...)` returns a vector of tetrahedron handles. Each element of `h` is a handle to the set of patches forming one tetrahedron. You can use these handles to view a particular tetrahedron by turning the patch `'Visible'` property `'on'` or `'off'`.

`tetramesh(...,'param','value','param','value'...)` allows additional patch property name/property value pairs to be used when displaying the tetrahedrons. For example, the default transparency parameter is set to `0.9`. You can overwrite this value by using the property name/property value pair `('FaceAlpha',value)` where `value` is a number between `0` and `1`. See Patch for information about the available properties.

Examples

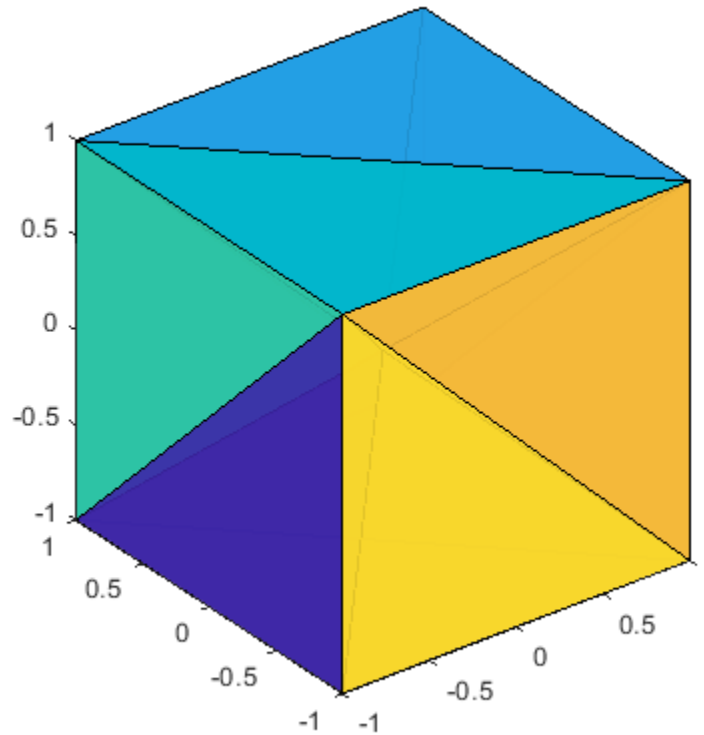
Plot Tetrahedrons of 3-D Delaunay Triangulation

Generate a 3-D Delaunay triangulation, then use `tetramesh` to visualize the tetrahedrons.

```
d = [-1 1];
[x,y,z] = meshgrid(d,d,d); % a cube
x = [x(:);0];
y = [y(:);0];
z = [z(:);0];
```

`[x,y,z]` are corners of a cube plus the center.

```
DT = delaunayTriangulation(x,y,z);
tetramesh(DT);
camorbit(20,0)
```



See Also

`delaunayTriangulation` | `delaunayn` | `freeBoundary(triangulation)` | `patch` | `triangulation` | `trimesh` | `trisurf`

Introduced before R2006a

matlab.test.behavior.Missing class

Package: `matlab.test.behavior`
Superclasses: `matlab.unittest.TestCase`

Test if class satisfies contract for `missing` values

Description

To test if the `missing` value for your class satisfies the missing contract on page 1-15292 in MATLAB, create a test class that derives from the `matlab.test.behavior.Missing` class. If your class represents a data type and you want MATLAB to treat `missing` values of your class similar to built-in classes, ensure that your class satisfies the `missing` contract.

Typically, you use the behavior test as part of a test-driven development workflow. If you want the missing value for your class to satisfy the missing contract with MATLAB, write the behavior test and modify the class under test until the test results are as you expect. For example, if your class supports comparisons and ordering and is usable as a missing value indicator, all tests should pass. If your class does not support ordering, set the `SupportsOrdering` property to `false` and expect that MATLAB filters out tests associated with ordering.

Your behavior test only must derive from `matlab.test.behavior.Missing` and define the abstract properties. However, since `matlab.test.behavior.Missing` is a subclass of `matlab.unittest.TestCase`, you can use the functionality of the Unit Testing Framework.

Class Attributes

```
Abstract                                                    true
HandleCompatible                                            true
```

For information on class attributes, see “Class Attributes”.

Properties

Define values for all `abstract` properties in the `properties` block of your test class.

`MissingValue` — Missing value for class under test
valid MATLAB scalar value | expression that returns non-missing value

Missing value for class under test, specified as a valid MATLAB scalar value or an expression that returns a non-missing value.

Example: `NaN` or `missing` or `' '`

Attributes:

```
SetAccess                                   public
GetAccess                                   public
Abstract                                    true
```

`PrototypeValue` — Non-missing value for class under test
valid MATLAB scalar value | expression that returns non-missing value

Any non-missing value for the class under test, specified as a valid MATLAB scalar value or an expression that returns a non-missing value. Typically, if the class constructor returns a non-missing value, `PrototypeValue` is a call to the constructor.

Example: 0 or `datetime`

Attributes:

```
SetAccess                                   public
GetAccess                                   public
Abstract                                    true
```

`ClassesWithSupportedConversions` — Classes that can be converted to class under test
string array of class names

List of classes that can be converted to the class under test, specified as a string array of class names.

If you can convert to the other class and then back to your class, then your class has a supported conversion from the other class. For example, if `MyClass(OtherClass(missing)) == MyClass(missing)` than `OtherClass` is a class with supported covnersions.

Example: `"string"` or `["double","single"]`

Attributes:

```
SetAccess                                   public
GetAccess                                   public
Abstract                                    true
```

Data Types: `string`

If necessary, redefine values for concrete properties in a function in the `TestClassSetup` methods block of your test class.

`SupportsComparison` — Class supports comparison
`true` (default) | `false`

Indicator that the class supports comparison, specified as `true` or `false`. A class that supports comparisons allows use of == and ~=.

If you set `SupportsComparison` to `false`, expect comparison and ordering tests to be filtered by assumption failure. MATLAB does not run the tests.

Attributes:

```
SetAccess                           protected
GetAccess                           public
```

Data Types: `logical`

`SupportsOrdering` — logical scalar to control the run of <, >, and so forth
`true` (default) | `false`

Indicator that the class supports ordering, specified as `true` or `false`. A class that supports ordering allows use of <, >, <=, and >=.

If you set `SupportsOrdering` to `false`, expect associated tests to be filtered by assumption failure. MATLAB does not run the tests.

Attributes:

```
SetAccess                           protected
GetAccess                           public
```

Data Types: `logical`

`UsableAsMissingIndicator` — Class is usable as missing value indicator
`true` (default) | `false`

Indicator that the class is usable as a missing value indicator to the `ismissing` function, specified as `true` or `false`.

If you set `UsableAsMissingIndicator` to `false`, expect associated tests to be filtered by assumption failure. MATLAB does not run the tests.

Attributes:

```
SetAccess                           protected
GetAccess                           public
```

Data Types: `logical`

`FillValue` — Value for growing array
value of `MissingValue` property (default) | valid MATLAB scalar value

The fill value that your class uses for growing arrays, specified as a valid MATLAB scalar value. By default, the value of `FillValue` is the same as the value of the `MissingValue` property.

Example: 0

Attributes:

```
SetAccess                           protected
GetAccess                           public
```

Examples

Verify Class Satisfies Missing Value Contract

Create the `MyDataClass` class that can contain a missing value. The class supports comparison and ordering, and the missing value implementation should satisfy the missing contract in MATLAB. If you call the constructor with no inputs, it returns a missing value.

```
classdef MyDataClass
    properties
```

```
        SomeData;
        MissingVal = false;
    end

    methods
        function obj = MyDataClass(value)
            if nargin
                m = size(value,1);
                n = size(value,2);
                for i = 1:m
                    for j = 1:n
                        if ismissing(value(i,j))
                            obj(i,j).MissingVal = true;
                        else
                            obj(i,j).SomeData = value(i,j);
                            obj(i,j).MissingVal = false;
                        end
                    end
                end
            else
                obj.MissingVal = true;
            end
        end

        % Define ismissing behavior
        function m = ismissing(obj,v)
            if nargin > 1
                m = isequaln(obj,v);
            else
                m = [obj.MissingVal];
            end
            m = reshape(m,size(obj));
        end
    end
end
```

To create a simple test class that checks that `MyDataClass` satisfies the missing value contract, subclass `matlab.test.behavior.Missing`. The test can use the functionality of the unit testing framework, but `MissingValueTest` is checking the missing contract only.

```
classdef MissingValueTest < matlab.test.behavior.Missing
    properties
        MissingValue = MyDataClass;
```

```
        PrototypeValue = MyDataClass(7);
        ClassesWithSupportedConversions = [];
    end
end
```

Run the tests and review the results. The tests for comparison, ordering, equality, and using `MyDataClass` as the second input to `ismissing` fail.

```
results = runtests('MissingValueTest');
```

```
Running MissingValueTest
....
================================================================================
Error occurred in MissingValueTest/comparison and it did not run to completion.

    ---------
    Error ID:
    ---------
    'MATLAB:UndefinedFunction'

    --------------
    Error Details:
    --------------
    Undefined function 'eq' for input arguments of type 'MyDataClass'.

    Error in matlab.test.behavior.Missing/comparison (line 129)
                testCase.verifyFalse(testCase.MissingValue == testCase.MissingValue,
                getString(message('MATLAB:test:behavior:missing:EqualFalse')));
================================================================================
.
================================================================================
Error occurred in MissingValueTest/ordering and it did not run to completion.

    ---------
    Error ID:
    ---------
    'MATLAB:UndefinedFunction'

    --------------
    Error Details:
    --------------
    Undefined function 'lt' for input arguments of type 'MyDataClass'.

    Error in matlab.test.behavior.Missing/ordering (line 136)
                testCase.verifyFalse(testCase.MissingValue < testCase.MissingValue,
                getString(message('MATLAB:test:behavior:missing:LessThanFalse')));
================================================================================
.
================================================================================
Verification failed in MissingValueTest/isequalRules.

    ----------------
    Test Diagnostic:
    ----------------
```

```
    isequal(MissingValue, MissingValue) must return false, because all missing values are unequal.

    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyFalse failed.
    --> The value must evaluate to "false".

    Actual Value:
      logical

       1

    ------------------
    Stack Information:
    ------------------
    In <matlabroot>\toolbox\matlab\datatypes\+matlab\+test\+behavior\Missing.m (Missing.isequalRules) at 145
================================================================================

================================================================================
Verification failed in MissingValueTest/isequalRules.

    ----------------
    Test Diagnostic:
    ----------------
    isequaln(MissingValue, missing) must return true.

    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyTrue failed.
    --> The value must evaluate to "true".

    Actual Value:
      logical

       0

    ------------------
    Stack Information:
    ------------------
    In <matlabroot>\toolbox\matlab\datatypes\+matlab\+test\+behavior\Missing.m (Missing.isequalRules) at 147
================================================================================
.
================================================================================
Verification failed in MissingValueTest/IsMissing2ndInput.

    ----------------
    Test Diagnostic:
    ----------------
    ismissing(MissingValue, missing) must return true.

    ---------------------
    Framework Diagnostic:
    ---------------------
```

```
    verifyTrue failed.
    --> The value must evaluate to "true".

    Actual Value:
      logical

       0

    ------------------
    Stack Information:
    ------------------
    In <matlabroot>\toolbox\matlab\datatypes\+matlab\+test\+behavior\Missing.m (Missing.IsMissing2ndInput) a
================================================================================
...
Done MissingValueTest
__________

Failure Summary:

     Name                                Failed  Incomplete  Reason(s)
    ==============================================================================
     MissingValueTest/comparison           X         X       Errored.
    ------------------------------------------------------------------------------
     MissingValueTest/ordering             X         X       Errored.
    ------------------------------------------------------------------------------
     MissingValueTest/isequalRules         X                 Failed by verification.
    ------------------------------------------------------------------------------
     MissingValueTest/IsMissing2ndInput    X                 Failed by verification.
```

Iteratively update `MyDataClass` to satisfy the missing contract. To satisfy comparison and ordering, define `eq`, `ne`, `lt`, `gt`, `le`, and `ge` in the `methods` block of `MyDataClass`.

```
% Class supports comparison
function tf = eq(obj1,obj2)
    tf = ~any(ismissing([obj1 obj2])) && eq(obj1.SomeData,obj2.SomeData);
end
function tf = ne(obj1,obj2)
    tf = ~eq(obj1,obj2);
end

% Class supports ordering
function tf = lt(obj1,obj2)
    tf = ~any(ismissing([obj1 obj2])) && lt(obj1.SomeData,obj2.SomeData);
end
function tf = gt(obj1,obj2)
    tf = lt(obj2,obj1);
end
function tf = le(obj1,obj2)
    tf = ~any(ismissing([obj1 obj2])) && ~gt(obj1,obj2);
end
function tf = ge(obj1,obj2)
    tf = le(obj2,obj1);
end
```

Run the tests with a terse level output detail and review the results.

```
results = runtests('MissingValueTest','OutputDetail',1);
```

```
......
FAIL: MissingValueTest/isequalRules in Missing.isequalRules at 145 :: verifyFalse failed.

FAIL: MissingValueTest/isequalRules in Missing.isequalRules at 147 :: verifyTrue failed.
.
FAIL: MissingValueTest/IsMissing2ndInput in Missing.IsMissing2ndInput at 154 :: verifyTrue failed.
...
```

Iteratively update `MyDataClass` to satisfy the equality rules. Define `isqual` and `isequaln` in the `methods` block of `MyDataClass`.

```
        % Class supports isequal/isequaln rules
        function tf = isequal(obj1,obj2)
            tf = eq(obj1,obj2);
        end
        function tf = isequaln(obj1,obj2)
            tf = all(ismissing([obj1 obj2])) || eq(obj1,obj2);
        end
```

Run the tests and review the results. The tests pass and `MyDataClass` satisfies the missing value contract.

```
results = runtests('MissingValueTest');
```

```
Running MissingValueTest
..........
Done MissingValueTest
__________
```

More About

Contract Details

Basic Validation

The behavior test asserts the following conditions are true.

- The missing value in the `MissingValue` property is the same class as the value in the `PrototypeValue` property.
- The `MissingValue` and `PrototypeValue` values are scalar.
- The `ismissing` function called with the `MissingValue` value returns `true`.
- The `ismissing` function called with the `PrototypeValue` value returns `false`.

Conversion

The behavior test verifies that converting the missing value to an instance of your class returns the same value as the `MissingValue` property.

Additionally, for each class that your class can convert, the behavior test verifies conversion of missing values. For example, if you can convert between `OtherClass` and `MyClass`, the behavior test verifies that `MyClass(OtherClass(missing)) == MyClass(missing)`. The `ClassesWithSupportedConversions` property indicates which classes your class can convert.

Subscripted Assignment

Given an array of objects of your class, subscripted assignment tests verify that you can expand the array using the missing value and assign the missing value to a range of elements in the array.

Concatenation

Concatenation tests verify that arrays of objects of your class can be concatenated with missing values.

Equality Rules

Equality tests verify that `isequal` returns `false` for two missing values and `isequaln` returns `true` for two missing values.

`SupportsComparison`

By default, the behavior test assumes that your class supports comparison of values, including missing values. Comparison tests verify that you implemented == and ~= as MATLAB expects, with respect to missing values.

- The comparison of two missing values with == returns `false`.
- The comparison of two missing values with ~= returns `true`.

If your class does not support comparisons, set the `SupportsComparison` property to `false` so MATLAB does not run these tests. In this case, comparison tests are filtered as assumption failures. Additionally, if `SupportsComparison = false`, MATLAB does not run the ordering tests.

`SupportsOrdering`

By default, the behavior test assumes that your class supports ordering of values, including missing values. Ordering tests verify that you implemented <, >, <=, and >= as MATLAB expects, with respect to missing values. For missing values, MATLAB expects that all the ordering operations return `false`.

If your class supports ordering, to satisfy the behavior contract, it must also support comparisons. If `SupportsComparison = false`, MATLAB does not run the ordering tests or the comparison tests. In this case, ordering tests are marked as assumption failures (filtered tests).

If your class does not support ordering, set the `SupportsOrdering` property to `false` so MATLAB does not run these tests. In this case, ordering tests are marked as assumption failures (filtered tests).

`UsableAsMissingIndicator`

By default, the behavior test assumes that an instance of your class can be used as the optional input argument to the `ismissing` function that represents the missing value indicator.

If your class is unable to be a missing value indicator, set the `UsableAsMissingIndicator` property to `false` so MATLAB does not run these tests. In this case, ordering tests are marked as assumption failures (filtered tests).

`FillValue`

The behavior test verifies that when MATLAB grows an array of elements of your class, the new elements have the value specified by the `FillValue` property. By default, the `FillValue` property is the same as the `MissingValue` property. However, you can specify a different value for `FillValue`.

See Also

`ismissing` | `matlab.unittest.TestCase` | `missing`

Introduced in R2018b

matlab.unittest.Test class

Package: `matlab.unittest`
Superclasses: `matlab.unittest.TestSuite`

Specification of single test method

Description

The `matlab.unittest.Test` class holds the information required by the `TestRunner` object to run a single `Test` method of a `TestCase` class. A scalar `Test` instance is the fundamental element contained in `TestSuite` arrays. A simple array of `Test` instances is a commonly used form of a `TestSuite` array.

Properties

`Name`

Name of the `Test` element.

`ProcedureName`

Name of the test procedure that runs for the `Test` element. In a class-based test, the `ProcedureName` is the name of the test method. In a function-based test, it is the name of the local function that contains the test. In a script-based test, it is a name generated from the test section title. Unlike the `Name` property, `ProcedureName` does not include any class or package name or information about parameterization.

`TestClass`

Name of the test class for the `TestCase`. If a `Test` element is not a class-based test, then `TestClass` is an empty string.

`BaseFolder`

Name of the folder that contains the file defining the test content. For tests defined in packages, the base folder is the parent of the top-level package folder.

Parameterization

Row vector of parameters required for the test. The `Parameterization` property contains all the parameterized data required by the `TestRunner`.

SharedTestFixtures

Row vector of fixtures required for the test. The `SharedTestFixtures` property contains all the fixtures specified by the `SharedTestFixtures` class-level attribute of the `TestCase` class.

Tags

Cell array of tags applied to the `Test` element.

Examples

Show Elements of TestSuite Array

In your current folder, create a file named `BankAccountTest.m` containing the following test class.

```
classdef BankAccountTest < matlab.unittest.TestCase

    methods (TestClassSetup)
        function addBankAccountClassToPath(testCase)
            p = path;
            testCase.addTeardown(@path,p);
            addpath(fullfile(matlabroot,'help','techdoc','matlab_oop',...
                'examples'));
        end
    end

    methods (Test)
        function testConstructor(testCase)
            b = BankAccount(1234, 100);
            testCase.verifyEqual(b.AccountNumber, 1234, ...
                'Constructor failed to correctly set account number');
            testCase.verifyEqual(b.AccountBalance, 100, ...
                'Constructor failed to correctly set account balance');
        end

        function testConstructorNotEnoughInputs(testCase)
            import matlab.unittest.constraints.Throws;
            testCase.verifyThat(@()BankAccount, ...
                Throws('MATLAB:minrhs'));
        end

        function testDesposit(testCase)
```

```
            b = BankAccount(1234, 100);
            b.deposit(25);
            testCase.verifyEqual(b.AccountBalance, 125);
        end

        function testWithdraw(testCase)
            b = BankAccount(1234, 100);
            b.withdraw(25);
            testCase.verifyEqual(b.AccountBalance, 75);
        end

        function testNotifyInsufficientFunds(testCase)
            callbackExecuted = false;
            function testCallback(~,~)
                callbackExecuted = true;
            end

            b = BankAccount(1234, 100);
            b.addlistener('InsufficientFunds', @testCallback);

            b.withdraw(50);
            testCase.assertFalse(callbackExecuted, ...
                'The callback should not have executed yet');
            b.withdraw(60);
            testCase.verifyTrue(callbackExecuted, ...
                'The listener callback should have fired');
        end
    end
end
```

Create a suite of `Test` objects from all `Test` methods in the `BankAccountTest` class.

```
import matlab.unittest.TestSuite
suite = TestSuite.fromClass(?BankAccountTest);
whos suite
```

```
  Name       Size             Bytes  Class                    Attributes

  suite      1x5              10502  matlab.unittest.Test
```

Each test is a `matlab.unittest.Test` object.

Display the `Test` element names.

```
{suite.Name}'
```

```
ans =

  5×1 cell array

    {'BankAccountTest/testConstructor'                }
    {'BankAccountTest/testConstructorNotEnoughInputs'}
    {'BankAccountTest/testDesposit'                   }
```

```
    {'BankAccountTest/testWithdraw'                  }
    {'BankAccountTest/testNotifyInsufficientFunds'   }
```

See Also

matlab.unittest.TestCase | matlab.unittest.TestRunner | matlab.unittest.TestSuite | matlab.unittest.fixtures

matlab.unittest.TestCase class

Package: `matlab.unittest`

Superclass of all `matlab.unittest` test classes

Description

The `TestCase` class is the means by which a test is written in the `matlab.unittest` framework. It provides the means to write and identify test content, as well as test fixture setup and teardown routines. Creating such a test requires deriving from `TestCase` to produce a `TestCase` subclass. Then, subclasses can leverage the metadata attributes to specify tests and test fixtures.

Construction

Use the `forInteractiveUse` static method to create a `TestCase` for interactive, command line use. When tests are run in the framework, `TestCase` instances are constructed by the `matlab.unittest.TestRunner`.

Methods

addTeardown	Dynamically add teardown routine
applyFixture	Use fixture with `TestCase`
forInteractiveUse	Create TestCase for interactive use
getSharedTestFixtures	Provide access to shared test fixtures
log	Record diagnostic information
onFailure	Dynamically add diagnostics for test failures
run	Run `TestCase` test

Inherited Methods

The `TestCase` class inherits methods from the following classes:

matlab.unittest.qualifications.Assertable	Qualification to validate preconditions of a test
matlab.unittest.qualifications.Assumable	Qualification to filter test content
matlab.unittest.qualifications.FatalAssertable	Qualification to abort test execution
matlab.unittest.qualifications.Verifiable	Qualification to produce soft failure conditions

Attributes

Class Attributes

`TestCase` objects support the following class level attributes. Specify class-level attributes in the `classdef` block before the class name.

`SharedTestFixtures`	Class block to contain shared test fixtures. You must define `SharedTestFixtures` as a cell array of `matlab.unittest.fixtures.Fixture` instances.
`TestTags`	Class block to contain tests tagged with a specified value. You must define `TestTags` as a cell array of non-empty character vectors or an array of non-empty strings, where each element is a tag for the test.

Method Attributes

Classes that derive from `TestCase` can define `methods` blocks which contain `matlab.unittest` framework-specific attributes to specify test content.

`Test`	Method block to contain test methods.
`TestMethodSetup`	Method block to contain setup code.
`TestMethodTeardown`	Method block to contain teardown code.

`TestClassSetup`	Method block to contain class level setup code.
`TestClassTeardown`	Method block to contain class level teardown code.
`ParameterCombination`	Method block to contain parameterized testing code. This attribute accepts the following values: • `'exhaustive'` (default): Test methods are invoked for all combinations of parameters. • `'sequential'`: Test methods are invoked with corresponding values from each parameter. Each parameter must contain the same number of values. • `'pairwise'`: Test methods are invoked for every pair of parameter values at least once.
`TestTags`	Method block to contain tests tagged with a specified value. You must define `TestTags` as a cell array of non-empty character vectors or an array of non-empty strings, where each element is a tag for the test.

Property Attributes

Classes that derive from `TestCase` can define `properties` blocks which contain `matlab.unittest` framework-specific attributes to specify test content.

`ClassSetupParameter`	Property block to define parameterized testing properties for methods in the `TestClassSetup` block
`MethodSetupParameter`	Property block to define parameterized testing properties for methods in the `MethodSetup` block
`TestParameter`	Property block to define parameterized testing properties for methods in the `Test` block

Events

`VerificationFailed`	Triggered upon failing verification. A `QualificationEventData` object is passed to listener callback functions.

`VerificationPassed`	Triggered upon passing verification. A `QualificationEventData` object is passed to listener callback functions.
`AssertionFailed`	Triggered upon failing assertion. A `QualificationEventData` object is passed to listener callback functions.
`AssertionPassed`	Triggered upon passing assertion. A `QualificationEventData` object is passed to listener callback functions.
`FatalAssertionFailed`	Triggered upon failing fatal assertion. A `QualificationEventData` object is passed to listener callback functions.
`FatalAssertionPassed`	Triggered upon passing fatal assertion. A `QualificationEventData` object is passed to listener callback functions.
`AssumptionFailed`	Triggered upon failing assumption. A `QualificationEventData` object is passed to listener callback functions.
`AssumptionPassed`	Triggered upon passing assumption. A `QualificationEventData` object is passed to listener callback functions.
`ExceptionThrown`	Triggered by the `TestRunner` when an exception is thrown. An `ExceptionEventData` object is passed to listener callback functions.
`DiagnosticLogged`	Triggered by the `TestRunner` upon a call to the `log` method. A `LoggedDiagnosticEventData` object is passed to the listener callback functions.

Examples

Create Test Case Class

Create a test case class, `FigurePropertiesTest`, with `TestMethodSetup` and `TestMethodTeardown` methods.

```
classdef FigurePropertiesTest < matlab.unittest.TestCase

    properties
        TestFigure
    end

    methods(TestMethodSetup)
        function createFigure(testCase)
```

```
            testCase.TestFigure = figure;
        end
    end

    methods(TestMethodTeardown)
        function closeFigure(testCase)
            close(testCase.TestFigure)
        end
    end

    methods(Test)

        function defaultCurrentPoint(testCase)

            cp = testCase.TestFigure.CurrentPoint;
            testCase.verifyEqual(cp, [0 0], ...
                'Default current point is incorrect')
        end

        function defaultCurrentObject(testCase)
            import matlab.unittest.constraints.IsEmpty

            co = testCase.TestFigure.CurrentObject;
            testCase.verifyThat(co, IsEmpty, ...
                'Default current object should be empty')
        end

    end

end
```

See Also

TestRunner | addlistener | matlab.unittest.constraints | matlab.unittest.diagnostics.LoggedDiagnosticEventData | matlab.unittest.qualifications | matlab.unittest.qualifications.ExceptionEventData | matlab.unittest.qualifications.QualificationEventData

Topics

"Create Basic Parameterized Test"
"Create Advanced Parameterized Test"

"Tag Unit Tests"
"Method Attributes"
"Class Attributes"

Introduced in R2013a

addTeardown

Class: `matlab.unittest.TestCase`
Package: `matlab.unittest`

Dynamically add teardown routine

Syntax

```
addTeardown(testCase,tearDownFcn)
addTeardown(testCase,tearDownFcn,arg1,...,argN)
```

Description

`addTeardown(testCase,tearDownFcn)` adds the `tearDownFcn` function handle that defines fixture teardown code to the `testCase` instance.

The function handle executes in the scope where it is registered. This scope can be a test class, a test method, or a shared fixture. The teardown code is executed in the reverse order to which it is added. This is a last-in, first-out (LIFO) execution order policy. To restore the correct pre-test state, the function handle is dynamically invoked with the conclusion of the scope.

`addTeardown(testCase,tearDownFcn,arg1,...,argN)` provides input arguments to the `tearDownFcn`.

Input Arguments

`testCase`

`matlab.unittest.TestCase` instance

Default:

`tearDownFcn`

Function, specified as a function handle, that defines the fixture teardown code

Default:

`arg1,...,argN`

Input arguments, 1 through N (if any), required by `tearDownFcn`, specified by any type. The argument type is specified by the function argument list.

Default:

Examples

Call addTeardown in a TestMethodSetup Method

```
classdef SomeTest < matlab.unittest.TestCase

    methods(TestMethodSetup)
        function createFixture(testCase)
            p = path;
            testCase.addTeardown(@path, p);
            addpath(fullfile(pwd,'testHelpers'));
        end
    end
end
```

See Also

applyFixture

Class: `matlab.unittest.TestCase`
Package: `matlab.unittest`

Use fixture with `TestCase`

Syntax

```
applyFixture(testCase,fixture)
f = applyFixture(testCase,fixture)
```

Description

`applyFixture(testCase,fixture)` prepares the specified fixture for use with the `TestCase`. This method enables the use of a fixture within the scope of a single `Test` method or `TestCase` class. The life cycle of the fixture is tied to the `TestCase`. When the `TestCase` goes out of scope, the testing framework tears down the fixture.

Call `applyFixture` within a `Test` method or `TestMethodSetup` method to use a fixture for the current test method alone. Use `applyFixture` within a `TestClassSetup` method to set up a fixture for the entire class.

`f = applyFixture(testCase,fixture)` also returns `fixture` as an output once it has been set up.

Input Arguments

`testCase`

`matlab.unittest.TestCase` instance

`fixture`

`matlab.unittest.fixtures.Fixture` instance

Examples

Apply Fixtures to TestCase Class

Create a temporary folder and make it the current working folder.

```
classdef applyFixtureTest < matlab.unittest.TestCase
    methods(TestMethodSetup)
        function addHelpers(testCase)
            import matlab.unittest.fixtures.TemporaryFolderFixture
            import matlab.unittest.fixtures.CurrentFolderFixture

            % Create a temporary folder and make it the current working folder.
            tempFixture = testCase.applyFixture(TemporaryFolderFixture);
            testCase.applyFixture(CurrentFolderFixture(tempFixture.Folder));
        end
    end
end
```

Each test method can write files to the current working folder, which is the temporary folder. After each test method runs, the testing framework restores the working folder to its previous state and deletes the temporary folder.

See Also

`matlab.unittest.fixtures`

matlab.unittest.TestCase.forInteractiveUse

Class: `matlab.unittest.TestCase`
Package: `matlab.unittest`

Create TestCase for interactive use

Syntax

```
tc = matlab.unittest.TestCase.forInteractiveUse
tc = matlab.unittest.TestCase.forInteractiveUse(testclass)
```

Description

`tc = matlab.unittest.TestCase.forInteractiveUse` creates a `TestCase` instance for interactive use. The `TestCase` is configured for experimentation at the command prompt. It reacts to qualification tests by printing messages to the screen for both passing and failing conditions.

`tc = matlab.unittest.TestCase.forInteractiveUse(testclass)` creates an instance of the `testclass` class for interactive use.

Input Arguments

`testclass`

`meta.class` instance that describes a `matlab.unittest.TestCase` subclass

Examples

Verify Values Using Interactive TestCase

Create a `TestCase` for interactive use.

```
import matlab.unittest.TestCase;
testCase = TestCase.forInteractiveUse;
```

Produce a passing verification.

```
testCase.verifyTrue(true, 'true should be true');
```

```
Interactive verification passed.
```

Produce a failing verification.

```
testCase.verifyTrue(false);
```

```
Interactive verification failed.

----------------------
Framework Diagnostic:
----------------------
verifyTrue failed.
--> The value must evaluate to "true".

Actual Value:
         0
```

Create Interactive TestCase for Specified Class

In a file in your working folder, create `ExampleTest.m`. This class is a subclass of `TestCase` and provides a helper verification method, `verifySameSize`.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods
        function verifySameSize(testCase, actual, expected)
            import matlab.unittest.constraints.ReturnsTrue;

            diagnostic = ['Actual and expected value sizes do not match.'...
                '\nActual size: ' num2str(size(actual)) ...
                '\nExpected size: ' num2str(size(expected))];

            testCase.verifyThat(@()isequal(size(actual),size(expected)),...
                ReturnsTrue, sprintf(diagnostic));
        end
    end
end
```

At the command prompt, create an interactive test case from the `ExampleTest` class.

```
tc = matlab.unittest.TestCase.forInteractiveUse(?ExampleTest);
```

Use the test case at the command prompt to call the `verifySameSize` method interactively.

```
tc.verifySameSize(1:10,5)
```

```
Interactive verification failed.

----------------
Test Diagnostic:
----------------
Actual and expected value sizes do not match.
Actual size: 1  10
Expected size: 1  1

---------------------
Framework Diagnostic:
---------------------
ReturnsTrue failed.
--> The function handle should have evaluated to "true".
--> Returned value:
             0

Actual function_handle:
        @()isequal(size(actual),size(expected))
```

See Also

`matlab.unittest.TestCase` | `matlab.unittest.qualifications`

Introduced in R2014a

getSharedTestFixtures

Class: `matlab.unittest.TestCase`
Package: `matlab.unittest`

Provide access to shared test fixtures

Syntax

```
fixtures = getSharedTestFixtures(testCase)
fixtures = getSharedTestFixtures(testCase,fixtureClassName)
```

Description

`fixtures = getSharedTestFixtures(testCase)` provides access to the array of all shared test fixtures for `testCase`. `getSharedTestFixtures` returns an array of fixture objects, `fixtures`. Specify shared fixtures are using the `SharedTestFixtures` attribute for the `testCase` class.

`fixtures = getSharedTestFixtures(testCase,fixtureClassName)` returns only the shared fixtures that have the class name `fixtureClassName`.

Input Arguments

`testCase`

`matlab.unittest.TestCase` instance

Default:

`fixtureClassName`

Name of test fixture class, specified as a character vector or string scalar

Examples

Obtain Array of All Shared Fixtures

Create the following class, `myTest`, on your MATLAB path. Two shared fixtures are used within the test method. This example assumes that the subfolder `helperFiles` exists in your working folder. Create the subfolder `helperFiles` in your working folder if it does not exist.

```
classdef (SharedTestFixtures={...
matlab.unittest.fixtures.PathFixture('helperFiles'),...
        matlab.unittest.fixtures.TemporaryFolderFixture}) ...
        myTest < matlab.unittest.TestCase
    methods(Test)
        function accessFixtures(testCase)
            myFixtures = testCase.getSharedTestFixtures
        end
    end
end
```

At the command prompt, run the test.

```
run(myTest);
```

```
Setting up PathFixture.
Description: Adds 'H:\Documents\doc_examples\helperFiles' to the path.
__________

Setting up TemporaryFolderFixture.
Description: Creates a temporary folder.
__________

Running myTest

myFixtures =

  1x2 heterogeneous Fixture (PathFixture, TemporaryFolderFixture) array with no propert

.
Done myTest
__________

Tearing down TemporaryFolderFixture.
```

```
Description: Deletes the temporary folder and all its contents.
__________

Tearing down PathFixture.
Description: Restores the path to its previous state.
__________
```

Access Shared Fixtures of Particular Class

Create the class, `mySecondTest`, on your MATLAB path.

```
classdef (SharedTestFixtures={...
        matlab.unittest.fixtures.TemporaryFolderFixture})...
        mySecondTest < matlab.unittest.TestCase
    methods(Test)
        function accessTemporaryFolderFixture(testCase)
            tempFolderFixture = testCase.getSharedTestFixtures...
                ('matlab.unittest.fixtures.TemporaryFolderFixture');
            temporaryFolder = tempFolderFixture.Folder
        end
    end
end
```

At the command prompt, run the test. The name of the temporary folder varies.

```
run(mySecondTest);
```

```
Setting up TemporaryFolderFixture.
Description: Creates a temporary folder.
__________

Running mySecondTest

temporaryFolder =

C:\Temp\tpb92c9c67_02fa_4714_bfb0_b2127df0f31d

.
Done mySecondTest
__________

Tearing down TemporaryFolderFixture.
```

```
Description: Deletes the temporary folder and all its contents.
__________
```

See Also

`matlab.unittest.TestCase` | `matlab.unittest.fixtures`

Topics

Class Attributes

log

Class: `matlab.unittest.TestCase`
Package: `matlab.unittest`

Record diagnostic information

Syntax

```
log(testCase,diagnostic)
log(testCase,v,diagnostic)
```

Description

`log(testCase,diagnostic)` logs the supplied diagnostic. The log method provides a means for tests to log information during their execution. The testing framework displays logged messages only if you configure it to do so by adding an appropriate plugin, such as the `matlab.unittest.plugins.LoggingPlugin`.

`log(testCase,v,diagnostic)` logs the diagnostic at the specified verbosity level, `v`.

Input Arguments

`testCase` — Instance of test case
`matlab.unittest.TestCase` instance

Instance of test case, specified as a `matlab.unittest.TestCase`.

`diagnostic` — Diagnostic information to display upon a failure
character array | string array | function handle | `matlab.unittest.diagnostics.Diagnostic` instance

Diagnostic information to display upon a failure, specified as a string array, character array, function handle, or `matlab.unittest.diagnostics.Diagnostic` instance.

v — Verbosity level
2 (default) | 1 | 3 | 4 | `matlab.unittest.Verbosity` enumeration | enumeration name as string or char vector

Verbosity level, specified as an integer value between 1 and 4, a `matlab.unittest.Verbosity` enumeration object, or a string scalar or character vector corresponding to one of the predefined enumeration member names. The default verbosity level for diagnostic messages is `Concise`. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration.

Numeric Representation	Enumeration Member Name	Verbosity Description
1	`Terse`	Minimal information
2	`Concise`	Moderate amount of information
3	`Detailed`	Some supplemental information
4	`Verbose`	Lots of supplemental information

Examples

Log Diagnostic Information

Create a function-based test in a file, `sampleLogTest.m`, in your working folder.

```
function tests = sampleLogTest
tests = functiontests(localfunctions);

function svdTest(testCase)
import matlab.unittest.Verbosity

log(testCase,'Generating matrix.');
m = rand(1000);

log(testCase,1,'About to call SVD.');
[U,S,V] = svd(m);

log(testCase,Verbosity.Terse,'SVD finished.');
```

```
verifyEqual(testCase,U*S*V',m,'AbsTol',1e-6)
```

At the command prompt, run the test.

```
results = run(sampleLogTest);
```

```
Running sampleLogTest
   [Terse] Diagnostic logged (2014-04-14T14:20:59): About to call SVD.
   [Terse] Diagnostic logged (2014-04-14T14:20:59): SVD finished.
.
Done sampleLogTest
__________
```

The default runner reports the diagnostics at level 1 (`Terse`).

Create a test runner to report the diagnostics at levels 1 and 2, and rerun the test.

```
import matlab.unittest.TestRunner
import matlab.unittest.plugins.LoggingPlugin

runner = TestRunner.withNoPlugins;
p = LoggingPlugin.withVerbosity(2);
runner.addPlugin(p);

results = runner.run(sampleLogTest);
```

```
 [Concise] Diagnostic logged (2014-04-14T14:28:14): Generating matrix.
   [Terse] Diagnostic logged (2014-04-14T14:28:14): About to call SVD.
   [Terse] Diagnostic logged (2014-04-14T14:28:15): SVD finished.
```

See Also

`matlab.unittest.Verbosity` | `matlab.unittest.plugins.LoggingPlugin`

Introduced in R2014b

onFailure

Class: `matlab.unittest.TestCase`
Package: `matlab.unittest`

Dynamically add diagnostics for test failures

Syntax

```
onFailure(testcase,failureDiag)
onFailure(testcase,failureDiag,'IncludingAssumptionFailures',tf)
```

Description

`onFailure(testcase,failureDiag)` adds diagnostics for test failures. If a test fails, then the test framework executes the diagnostics. By default, these diagnostics execute upon verification failures, assertion failures, fatal assertion failures, and uncaught exceptions.

`onFailure(testcase,failureDiag,'IncludingAssumptionFailures',tf)` indicates if the test framework also executes diagnostics upon assumption failures. To execute diagnostics upon assumption failures also, set `tf` to `true`.

Input Arguments

`testCase` — Instance of test case
`matlab.unittest.TestCase` instance

Instance of test case, specified as a `matlab.unittest.TestCase`.

`failureDiag` — Diagnostic information
character vector | string array | function handle | array of `matlab.unittest.diagnostics.Diagnostic` instances

Diagnostic information to display upon a failure, specified as a character vector, string array, function handle, or array of `matlab.unittest.diagnostics.Diagnostic` instances.

Example: `@() disp('Failure Detected')`

Example: `matlab.unittest.diagnostics.ScreenshotDiagnostic`

`tf` — React to assumption failures
`false` (default) | `true`

React to assumption failures, specified as `false` (`logical` 0) or `true` (`logical` 1). By default, this value is `false` and the test framework executes diagnostics upon verification failures, assertion failures, fatal assertion failures, and uncaught exceptions. However, the test framework does not execute diagnostics upon assumption failures. To execute additional diagnostics upon assumption failures, specify this value as `true`.

Examples

Add Diagnostics to Execute Upon Test Failure

In your current working folder, create the following test class `SampleOnFailureTest.m`.

```
classdef SampleOnFailureTest < matlab.unittest.TestCase
    methods(TestMethodSetup)
        function addFailureDiag(testCase)
            testCase.onFailure('Failure Detected');
        end
    end
    methods (Test)
        function verificationFailTest(testCase)
            testCase.onFailure(@()disp(datetime))
            testCase.verifyEqual(42,13)
        end
        function passingTest(testCase)
            testCase.assertTrue(true)
        end
        function assumptionFailTest(testCase)
            testCase.assumeEmpty(rand(2))
        end
        function assertionFailTest(testCase)
            act = randi(100,1,15);
```

```
            floor = randi(100,1,15);
            f = figure; hold on;
            plot(1:length(act),act,1:length(floor),floor);
            legend('actual','floor')
            testCase.addTeardown(@close,f)

            import matlab.unittest.diagnostics.FigureDiagnostic
            testCase.onFailure(FigureDiagnostic(f,'Formats','png'))

            testCase.assertGreaterThan(act,floor)
        end
    end
end
```

At the command prompt, run the tests. The `SampleOnFailureTest` class has these results.

- The diagnostic message `'Failure Detected'` displays for each test with a verification, assertion, or fatal assertion failure because `addFailureDiag` calls `onFailure` in a function in the `TestMethodSetup` block.
- The `verificationFailTest` test adds another diagnostic upon failure that displays the current date and time.
- The `assumptionFailTest` test fails by assumption. Therefore the `'Failure Detected'` message does not display.
- The `assertionFailTest` test plots the data. If the test fails, the test framework saves the plot.

```
results = runtests('SampleOnFailureTest');
```

```
Running SampleOnFailureTest

================================================================================
Verification failed in SampleOnFailureTest/verificationFailTest.

    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyEqual failed.
    --> The values are not equal using "isequaln".
    --> Failure table:
            Actual    Expected    Error     RelativeError
            ______    ________    _____    _______________

            42        13          29       2.23076923076923

    Actual Value:
```

```
        42
    Expected Value:
        13

    ----------------------
    Additional Diagnostic:
    ----------------------
    Failure Detected

    ------------------
    Stack Information:
    ------------------
    In C:\work\SampleOnFailureTest.m (SampleOnFailureTest.verificationFailTest) at 10
================================================================================
..
================================================================================
SampleOnFailureTest/assumptionFailTest was filtered.
Details
================================================================================
.
================================================================================
Assertion failed in SampleOnFailureTest/assertionFailTest and it did not run to completion.

    ---------------------
    Framework Diagnostic:
    ---------------------
    assertGreaterThan failed.
    --> Each element must be greater than each corresponding element of the minimum value array.

        Failing Indices:
             1     2     3     4     5     6     9    12    13    14

    Actual Value:
        55    68     4    81    75    13    53    33    55    40    42    19    26     3    93
    Minimum Value (Exclusive):
        66    94    17    93    80    58    45    26    76    23     7    77    68    72    65

    ----------------------
    Additional Diagnostic:
    ----------------------
    Failure Detected

    ----------------------
    Additional Diagnostic:
    ----------------------
    Figure saved to:
    --> C:\Temp\070a23db-3903-4abb-9976-2d29ec0a1e38\Figure_c2528bb2-6d72-48d5-a8ed-2a9d5ae7b3ee.png

    ------------------
    Stack Information:
    ------------------
    In C:\work\SampleOnFailureTest.m (SampleOnFailureTest.assertionFailTest) at 28
================================================================================
.
Done SampleOnFailureTest
```

```
__________

Failure Summary:

     Name                                       Failed  Incomplete  Reason(s)
    ==========================================================================================
     SampleOnFailureTest/verificationFailTest    X                   Failed by verification.
    ------------------------------------------------------------------------------------------
     SampleOnFailureTest/assumptionFailTest                 X       Filtered by assumption.
    ------------------------------------------------------------------------------------------
     SampleOnFailureTest/assertionFailTest       X          X       Failed by assertion.
```

Tips

- To add a diagnostic for each test in your class, call the `onFailure` method from a function in the `TestMethodSetup` block.

See Also

`matlab.unittest.diagnostics`

Introduced in R2017b

run

Class: `matlab.unittest.TestCase`
Package: `matlab.unittest`

Run `TestCase` test

Syntax

```
result = run(testCase)
result = run(testCase,testMethod)
```

Description

`result = run(testCase)` uses `testCase` as a prototype to run a `TestSuite` array created from all test methods in the class defining `testCase`. This suite is run using a `TestRunner` object configured for text output.

`result = run(testCase,testMethod)` uses `testCase` as a prototype to run a `TestSuite` array created from `testMethod`. This test is run using a `TestRunner` object configured for text output.

This is a convenience method to allow interactive experimentation of `TestCase` classes in MATLAB, yet running the tests contained in them using a supported `TestRunner` object.

Input Arguments

`testCase`

`matlab.unittest.TestCase` instance

Default:

`testMethod`

Name of desired test method, specified as one of the following:

- character vector
- string scalar
- `meta.method` instance

The method must correspond to a valid `Test` method of the `testCase` instance.

Default:

Output Arguments

`result`

A `matlab.unittest.TestResult` object containing the result of the test run.

Examples

Run Test Directly from Test Case

Add the `FigurePropertiesTest.m` test case file to a folder on your MATLAB path.

```
classdef FigurePropertiesTest < matlab.unittest.TestCase

    properties
        TestFigure
    end

    methods(TestMethodSetup)
        function createFigure(testCase)
            % comment
            testCase.TestFigure = figure;
        end
    end

    methods(TestMethodTeardown)
        function closeFigure(testCase)
            close(testCase.TestFigure)
        end
    end
```

```
    methods(Test)

        function defaultCurrentPoint(testCase)

            cp = testCase.TestFigure.CurrentPoint;
            testCase.verifyEqual(cp, [0 0], ...
                'Default current point is incorrect')
        end

        function defaultCurrentObject(testCase)
            import matlab.unittest.constraints.IsEmpty

            co = testCase.TestFigure.CurrentObject;
            testCase.verifyThat(co, IsEmpty, ...
                'Default current object should be empty')
        end

    end

end
```

Create a testcase object.

```
tc = FigurePropertiesTest;
```

Run the tests.

```
tc.run;
```

```
Running FigurePropertiesTest
..
Done FigurePropertiesTest
__________
```

All tests passed.

See Also

run | run

matlab.unittest.TestResult class

Package: `matlab.unittest`

Result of running test suite

Description

The `matlab.unittest.TestResult` class holds the information describing the result of running a test suite using the `matlab.unittest` framework. The results include information describing whether the test passed, failed, or ran to completion, as well as the duration of each test.

Construction

`TestResult` arrays are created and returned by the test runner, and are of the same size as the suite which was run.

Properties

`Name`

The name of the `TestSuite` object for the result.

The `Name` property is a character vector that holds the name of the test corresponding to this result.

`Passed`

Logical value showing if the test passed.

When the `Passed` property is TRUE, then the test completed as expected without any failure. When it is FALSE, then the test did not run to completion and/or encountered a failure condition.

Failed

Logical value showing if test failed.

A TRUE `Failed` property indicates some form of test failure. When `Failed` is FALSE, then no failing conditions were encountered. A failing result can occur with a failure condition either in a test or in setting up and tearing down test fixtures. Failures can occur due to the following:

- Verification failures
- Assertion failures
- Uncaught MExceptions

Fatal assertions are also failing conditions, but in the event of a fatal assertion failure, the entire framework aborts and a `TestResult` object is never produced.

Incomplete

Logical value showing if test did not run to completion.

A TRUE `Incomplete` property indicates a test did not run to completion. When it is FALSE, then no conditions were encountered that prevented the test from completing. In other words, when FALSE there were no stack disruptions out of the running test content. An incomplete result can occur with a stack disruption in either a test or when setting up and tearing down test fixtures. Incomplete tests can occur due to the following:

- Assertion failures
- Tests filtered through assumption
- Uncaught MExceptions

Fatal assertions are also conditions that prevent the completion of tests, but in the event of a fatal assertion failure the entire framework aborts and a `TestResult` object is never produced.

Duration

Time elapsed running test.

The `Duration` property indicates the amount of time taken to run a particular test, including the time taken setting up and tearing down any test fixtures.

Fixture setup time is accounted for in the duration of the first test suite array element that uses the fixture. Fixture teardown time is accounted for in the duration of the last test suite array element that uses the fixture.

The total run time for a suite of tests exceeds the sum of the durations for all the elements of the suite because the `Duration` property does not include all the overhead of the `TestRunner` object, nor any of the time consumed by test runner plugins.

`Details`

Data for test result.

The `Details` property contains additional information relevant to the test execution. The type of information depends on the configuration of the `TestRunner` and its plugins. For example, the `DiagnosticsRecordingPlugin` uses this property to include diagnostic information it encounters during the test.

Tips

- Create a table from the `TestResult` object for access to `table` functionality such as sorting rows, displaying a summary, and writing the table to a file. For example,

  ```
  rt = table(results);
  ```

See Also

`TestRunner` | `TestSuite` | `table`

Topics

"Analyze Test Case Results"
"Analyze Failed Test Results"
Property Attributes

matlab.unittest.TestRunner class

Package: `matlab.unittest`

Class for running tests in `matlab.unittest` framework

Description

The `matlab.unittest.TestRunner` class is the fundamental API used to run a suite of tests in the `matlab.unittest` framework. It runs and operates on `TestSuite` arrays. Use this class to customize running tests.

The `TestRunner` class is a sealed class; you cannot derive classes from the `TestRunner` class.

Construction

To create a simple, silent `TestRunner` object, call the static `withNoPlugins` method.

```
runner = matlab.unittest.TestRunner.withNoPlugins
```

To create a `TestRunner` object to run tests from the MATLAB Command Window, call the static `withTextOutput` method.

```
runner = matlab.unittest.TestRunner.withTextOutput
```

To create a customized `TestRunner` object, call the `addPlugin` method.

```
runner = TestRunner.withNoPlugins;
runner.addPlugin(SomePlugin())
```

Properties

`ArtifactsRootFolder` — Root folder where test run artifacts are stored
string scalar | character vector

Root folder where test run artifacts are stored, specified as a string scalar or character vector. By default, the value of `ArtifactsRootFolder` is the value from `string(tempdir)`, but you can set it to any writable folder.

Any artifacts produced during a test run are stored in a subfolder within `ArtifactsRootFolder`. The subfolder name is a unique identifier associated with the specific test run. MATLAB creates a subfolder only if the test run produces artifacts.

For example, assume the `ArtifactsRootFolder` is set to `"C:\Temp"` and the automatically-generated test run identifier is `"1231df38-7515-4dbe-a869-c3d9f885f379"`. If a test run produces an artifact, `"artifact.txt"`, the artifact is stored as `"C:\Temp\1231df38-7515-4dbe-a869-c3d9f885f379\artifact.txt"`.

`PrebuiltFixtures` — Fixtures set up outside the test runner

scalar `Fixture` instance | row vector of `Fixture` instances

Fixtures that are set up outside the test runner, specified as a scalar or row vector of `matlab.unittest.fixtures.Fixture` instances. Use this property to specify that the environmental configuration is performed manually instead of automatically during fixture setup and teardown.

The test runner considers these fixtures as already set up and never attempts to set up or tear down any fixtures specified by the `PrebuiltFixtures` property. If a test suite requires a shared test fixture and that test fixture is specified as a prebuilt fixture, the test runner does not perform set up or tear down actions.

Note The test runner uses a prebuilt fixture only if it is specified by the `PrebuiltFixtures` property and is listed as a `SharedTestFixture` in the test class definition. The test runner does not use a prebuilt fixture if the fixture is registered using the `TestCase.applyFixture` method.

Methods

addPlugin	Add plugin to `TestRunner` object
run	Run all tests in `TestSuite` array
runInParallel	Run all tests in `TestSuite` array in parallel
withNoPlugins	Create simplest runner possible
withTextOutput	Create `TestRunner` object for command window output

Examples

Create TestRunner Object Configured for Text Output

Add `matlab.unittest` classes to the current import list.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;
```

Create a `TestSuite` array.

```
suite = TestSuite.fromClass(?mypackage.MyTestClass);
```

Create the `TestRunner` object and run the suite.

```
runner = TestRunner.withTextOutput;
result = run(runner,suite);
```

Include Prebuilt Fixture

This example uses a shared test fixture and then specifies the fixture as prebuilt. The test runner does not set up and tear down the prebuilt fixture. Since the test assumes that the fixture exists, you must manually perform the setup work that the fixture ordinarily performs.

Create a test class in a file in your working folder. The test class uses a `PathFixture` as a shared test fixture. This example assumes that the subfolder, `helperFiles`, exists in your working folder.

```
classdef (SharedTestFixtures={ ...
        matlab.unittest.fixtures.PathFixture('helperFiles')}) ...
        SampleTest < matlab.unittest.TestCase
    methods(Test)
        function test1(testCase)
            f = testCase.getSharedTestFixtures;

            import matlab.unittest.constraints.ContainsSubstring
            testCase.assertThat(path,ContainsSubstring(f.Folder))
        end
    end
end
```

Create a test suite and test runner at the command prompt.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;

suite = TestSuite.fromClass(?SampleTest);
runner = TestRunner.withTextOutput;
```

Run the tests using the shared test fixture. In this case, the fixture is not prebuilt.

```
runner.run(suite);
```

```
Setting up PathFixture
Done setting up PathFixture: Added 'C:\Work\helperFiles' to the path.
__________

Running SampleTest
.
Done SampleTest
__________

Tearing down PathFixture
Done tearing down PathFixture: Restored the path to its original state.
__________
```

The test runner sets up and tears down the shared test fixture.

Create an instance of the fixture and add it to the test runner.

```
f = matlab.unittest.fixtures.PathFixture('helperFiles');
runner.PrebuiltFixtures = f;
```

Manually add the `'helperFiles'` folder to your path. The `PathFixture` adds the specified folder to your path, and the tests rely on this setup action. However, since the fixture is defined as prebuilt, the test runner does not perform set up or tear down actions, and you must perform them manually. In this case, if you do not manually add it to your path, the test fails.

```
p = fullfile(pwd,'helperFiles');
oldPath = addpath(p);
```

Run the tests.

```
runner.run(suite);
```

```
Running SampleTest
.
Done SampleTest
__________
```

The test runner assumes that the fixture is prebuilt and does not set it up or tear it down.

Manually reset your path.

```
path(oldPath)
```

See Also

`TestResult` | `TestSuite`

Topics

"Write Tests Using Shared Fixtures"

Introduced in R2013a

addPlugin

Class: `matlab.unittest.TestRunner`
Package: `matlab.unittest`

Add plugin to `TestRunner` object

Syntax

```
addPlugin(runner,plugin)
```

Description

`addPlugin(runner,plugin)` adds `plugin` to `runner`.

Input Arguments

`runner`

`matlab.unittest.TestRunner` object.

Default:

`plugin`

Mechanism provided to customize the manner in which a `TestSuite` array is run, specified as a `TestRunnerPlugin` object.

Default:

Examples

Run Test with Custom Plugin

Add `matlab.unittest` classes to the current import list.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;
```

Create a `TestSuite` array.

```
suite = TestSuite.fromClass(?mypackage.MyTestClass);
```

Create a `TestRunner` object.

```
runner = TestRunner.withNoPlugins;
```

Add a custom plugin.

```
import matlab.unittest.plugins.DiagnosticsValidationPlugin;
runner.addPlugin(DiagnosticsValidationPlugin);
```

Run the test.

```
result = run(runner,suite);
```

See Also

run

Class: `matlab.unittest.TestRunner`
Package: `matlab.unittest`

Run all tests in `TestSuite` array

Syntax

```
result = run(runner,suite)
```

Description

`result = run(runner,suite)` runs the `TestSuite` array defined by `suite` using the `TestRunner` object provided in `runner`, and returns the result in `result`.

This method runs all of the appropriate methods of the `TestCase` class to set up fixtures and run test content. It handles errors and qualification failures and records the information in `result`.

Input Arguments

`runner`

`matlab.unittest.TestRunner` object.

Default:

`suite`

Set of tests, specified as a `matlab.unittest.TestSuite` array.

Output Arguments

result

A `matlab.unittest.TestResult` object containing the result of the test run. `result` is the same size as `suite` and each element is the result of the corresponding element in `suite`.

Examples

Run All Tests in a Package

Add `matlab.unittest` classes to the current import list.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;
```

Create a test suite, and a test runner that displays text.

```
suite = TestSuite.fromClass(?mypackage.MyTestClass);
runner = TestRunner.withTextOutput;
```

Run the test suite.

```
result = runner.run(suite)
```

See Also

`run` | `run`

runInParallel

Class: `matlab.unittest.TestRunner`
Package: `matlab.unittest`

Run all tests in `TestSuite` array in parallel

Syntax

```
result = runInParallel(runner,suite)
```

Description

`result = runInParallel(runner,suite)` runs all tests in the `TestSuite` array in parallel and returns the results in a `TestResult` object. The `runInParallel` method divides `suite` into separate groups and uses `runner` to run each group on the current parallel pool.

Note The `runInParallel` method requires Parallel Computing Toolbox. The testing framework might vary the order and number of groups or which tests it includes in each group.

When you select a test suite to run in parallel, consider possible resource contention. For example, if your test fixtures access global resources, such as a shared file on the same network, the parallel sessions could conflict with each other. In such cases, consider using a prebuilt shared test fixture.

Input Arguments

`runner` — Test runner for parallel test groups
`matlab.unittest.TestRunner` instance

Test runner for parallel test groups, specified as a `matlab.unittest.TestRunner` instance.

Consider your test runner configuration before running tests in parallel. Since the `runInParallel` method runs separate groups of tests on different workers, some plugins, such as `StopOnFailuresPlugin`, are not well suited for parallelization. The testing framework supports running tests in parallel with a custom plugin, provided that the plugin subclasses the `Parallelizable` interface.

`suite` — Set of tests to run in parallel
`matlab.unittest.Test` array

Set of tests to run in parallel, specified as a `matlab.unittest.Test` array.

Examples

Run Tests in Parallel

Create the following parameterized test in a file in your current working folder.

```
classdef TestRand < matlab.unittest.TestCase
    properties (TestParameter)
        dim1 = createDimensionSizes;
        dim2 = createDimensionSizes;
        dim3 = createDimensionSizes;
        type = {'single','double'};
    end

    methods (Test)
        function testRepeatable(testCase,dim1,dim2,dim3)
            state = rng;
            firstRun = rand(dim1,dim2,dim3);
            rng(state)
            secondRun = rand(dim1,dim2,dim3);
            testCase.verifyEqual(firstRun,secondRun);
        end
        function testClass(testCase,dim1,dim2,type)
            testCase.verifyClass(rand(dim1,dim2,type),type)
        end
    end
end

function sizes = createDimensionSizes
% Create logarithmicly spaced sizes up to 100
```

```
sizes = num2cell(round(logspace(0,2,10)));
end
```

At the command prompt, create a suite from `TestRand.m` and a test runner that displays text in the Command Window.

```
suite = matlab.unittest.TestSuite.fromClass(?TestRand);
runner = matlab.unittest.TestRunner.withTextOutput();
```

The suite contains 1200 test elements.

Run the test suite in parallel.

```
result = runInParallel(runner,suite)
```

```
Split tests into 12 groups and running them on 4 workers.
----------------
Finished Group 2
----------------
Running TestRand
..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
.........
Done TestRand
__________


----------------
Finished Group 4
----------------
Running TestRand
..........
..........
..........
..........
..........
..........
```

```
..........
..........
..........
..........
.....
Done TestRand
__________

----------------
Finished Group 3
----------------
Running TestRand
..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
.......
Done TestRand
__________

----------------
Finished Group 1
----------------
Running TestRand
..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
..
```

```
Done TestRand
__________

----------------
Finished Group 7
----------------
Running TestRand
..........
..........
..........
..........
..........
..........
..........
..........
..........
.........
Done TestRand
__________

----------------
Finished Group 5
----------------
Running TestRand
..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
...
Done TestRand
__________

----------------
Finished Group 6
----------------
```

```
Running TestRand
..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
.
Done TestRand
__________


----------------
Finished Group 8
----------------
Running TestRand
..........
..........
..........
..........
..........
..........
..........
..........
..........
.......
Done TestRand
__________


-----------------
Finished Group 11
-----------------
Running TestRand
..........
..........
..........
..........
..........
..........
```

```
..........
..........
..........
.
Done TestRand
__________


-----------------
Finished Group 12
-----------------
Running TestRand
..........
..........
..........
..........
..........
..........
..........
..........
........
Done TestRand
__________


-----------------
Finished Group 10
-----------------
Running TestRand
..........
..........
..........
..........
..........
..........
..........
..........
..........
...
Done TestRand
__________


----------------
```

```
Finished Group 9
----------------
Running TestRand
..........
..........
..........
..........
..........
..........
..........
..........
..........
.....
Done TestRand
_________


result =

  1200x1 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
   1200 Passed, 0 Failed, 0 Incomplete.
   11.4023 seconds testing time.
```

See Also

matlab.unittest.TestResult | matlab.unittest.TestRunner | matlab.unittest.TestSuite | matlab.unittest.plugins.Parallelizable | matlab.unittest.plugins.TestRunnerPlugin | run

Topics

"Parallel Computing Fundamentals" (Parallel Computing Toolbox)
"Run Tests in Parallel with Custom Plugin"

Introduced in R2015a

matlab.unittest.TestRunner.withNoPlugins

Class: `matlab.unittest.TestRunner`
Package: `matlab.unittest`

Create simplest runner possible

Syntax

```
runner = matlab.unittest.TestRunner.withNoPlugins
```

Description

`runner = matlab.unittest.TestRunner.withNoPlugins` creates a TestRunner that is guaranteed to have no plugins installed and returns it in `runner`. It is the method one can use to create the simplest runner possible without violating the guarantees a test writer has when writing TestCase classes. This runner is a silent runner, meaning that regardless of passing or failing tests, this runner produces no command window output, although the results returned after running a test suite are accurate.

This method can also be used when it is desirable to have complete control over which plugins are installed and in what order. It is the only method guaranteed to produce the minimal TestRunner with no plugins, so one can create it and add additional plugins as desired.

Output Arguments

`runner`

`matlab.unittest.TestRunner` object.

Default:

Attributes

`Static`	`true`

To learn about attributes of methods, see Method Attributes.

Examples

Create a Silent TestRunner Object with no Plugins

Add `matlab.unittest` classes to the current import list.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;
```

Create a `TestSuite` array.

```
suite = TestSuite.fromClass(?mypackage.MyTestClass);
```

Create a `TestRunner` object.

```
runner = TestRunner.withNoPlugins;
```

```
% Run the suite silently
result = run(runner,suite)
```

Control Plugins

Using the `TestRunner` object created in the previous example, control which plugins are installed and in what order they are installed.

Add `matlab.unittest` class to the current import list.

```
import matlab.unittest.plugins;
```

Add specific plugins.

```
runner.addPlugin(DiagnosticsValidationPlugin);
runner.addPlugin(TestRunProgressPlugin.withVerbosity(2));
```

Rerun the tests.

```
result = run(runner,suite)
```

See Also

matlab.unittest.TestRunner.withTextOutput

Class: `matlab.unittest.TestRunner`
Package: `matlab.unittest`

Create `TestRunner` object for command window output

Syntax

```
runner = matlab.unittest.TestRunner.withTextOutput
runner = matlab.unittest.TestRunner.withTextOutput(Name,Value)
```

Description

`runner = matlab.unittest.TestRunner.withTextOutput` creates a `TestRunner` object that is configured for running tests from the MATLAB Command Window and returns it in `runner`. The output produced includes test progress as well as diagnostics in the event of test failures.

`runner = matlab.unittest.TestRunner.withTextOutput(Name,Value)` creates a `TestRunner` with additional options specified by one or more `Name,Value` pair arguments. For example, to create a `TestRunner` that excludes logged diagnostics, specify `matlab.unittest.TestRunner.withTextOutput('LoggingLevel',0)`.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `matlab.unittest.TestRunner.withTextOutput('OutputDetail',4)` displays event detail at them most verbose level

`LoggingLevel` — Maximum verbosity level for logged diagnostics

1 (default) | 0 | 2 | 3 | 4 | `matlab.unittest.Verbosity` enumeration | enumeration name as string or char vector

Maximum verbosity level for logged diagnostics included by the `TestRunner` instance, specified as an integer value from 0 through 4, a `matlab.unittest.Verbosity` enumeration object, or a string scalar or character vector corresponding to one of the predefined enumeration member names. The `TestRunner` includes diagnostics that are logged at this level and below. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration.

By default the `TestRunner` includes diagnostics logged at the `matlab.unittest.Verbosity.Terse` level (level 1). To exclude logged diagnostics, specify `LoggingLevel` as `Verbosity.None` (level 0).

Logged diagnostics are diagnostics that you supply to the testing framework with a call to the `log` or `log` method.

Numeric Representation	Enumeration Member Name	Verbosity Description
`0`	`None`	No information
`1`	`Terse`	Minimal information
`2`	`Concise`	Moderate amount of information
`3`	`Detailed`	Some supplemental information
`4`	`Verbose`	Lots of supplemental information

`OutputDetail` — Display level for event details

3 (default) | 0 | 1 | 2 | 4 | `matlab.unittest.Verbosity` enumeration | enumeration name as string or char vector

Display level for event details, specified as an integer value from 0 through 4, or as a `matlab.unittest.Verbosity` enumeration object. Integer values correspond to the members of the `matlab.unittest.Verbosity` enumeration, or a string scalar or character vector corresponding to one of the predefined enumeration member names.

The `TestRunner` displays failing and logged events with the amount of detail specified by `OutputDetail`. By default the `TestRunner` displays events at the `matlab.unittest.Verbosity.Detailed` level (level 3).

Numeric Representation	Enumeration Member Name	Verbosity Description
0	None	No information
1	Terse	Minimal information
2	Concise	Moderate amount of information
3	Detailed	Some supplemental information
4	Verbose	Lots of supplemental information

Examples

Display Test Results in Command Window

Add `matlab.unittest` classes to the current import list.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;
```

Create a `TestSuite` array.

```
suite = TestSuite.fromClass(?mypackage.MyTestClass);
```

Create a `TestRunner` object that produced output to the Command Window.

```
runner = TestRunner.withTextOutput;

% Run the suite
result = run(runner,suite)
```

Specify Logged Diagnostic Detail

Create the follow class In a file in your current working folder, `ExampleLogTest.m`.

```
classdef ExampleLogTest < matlab.unittest.TestCase
    methods(Test)
        function testOne(testCase)
            log(testCase,matlab.unittest.Verbosity.Detailed,'Starting Test')
```

```
            log(testCase,'Testing 5==5')
            testCase.verifyEqual(5,5)
            log(testCase,matlab.unittest.Verbosity.Verbose,'Test Complete')
        end
    end
end
```

At the command prompt, run the test.

```
result = run(ExampleLogTest);
```

```
Running ExampleLogTest
.
Done ExampleLogTest
__________
```

Create a test runner to display logged messages at verbosity level 4 and lower, and then run the test.

```
import matlab.unittest.TestRunner
import matlab.unittest.TestSuite
suite = TestSuite.fromClass(?ExampleLogTest);
runner = TestRunner.withTextOutput('LoggingLevel',4);

results = runner.run(suite);
```

```
Running ExampleLogTest

[Detailed] Diagnostic logged (2018-04-12 12:53:47): Starting Test

[Concise] Diagnostic logged (2018-04-12 12:53:47): Testing 5==5

[Verbose] Diagnostic logged (2018-04-12 12:53:47): Test Complete
.
Done ExampleLogTest
__________
```

See Also

`log` | `log` | `matlab.unittest.Verbosity` | `run`

matlab.unittest.TestSuite class

Package: `matlab.unittest`

Class for grouping tests to run

Description

The `matlab.unittest.TestSuite` class is the fundamental interface used to group and run a set of tests in the unit test framework. The `matlab.unittest.TestRunner` object can only run arrays of `TestSuite` objects.

Construction

`TestSuite` arrays are created using static methods of the `TestSuite` class. These methods may return subclasses of the `TestSuite` class depending on the method call and context.

Methods

fromClass	Create suite from `TestCase` class
fromFile	Create `TestSuite` array from test file
fromFolder	Create `TestSuite` array from all tests in folder
fromProject	Create test suite array from tests in project
fromMethod	Create `TestSuite` array from single test method
fromName	Create `Test` object from name of test element
fromPackage	Create `TestSuite` array from all tests in package
run	Run `TestSuite` array using `TestRunner` object configured for text output
selectIf	Select test suite elements that satisfy conditions
sortByFixtures	Reorder test suite based on shared fixtures

Examples

Create Test Suite of Every Type of Test Set

Add the `matlab.unittest.TestSuite` class to the current import list.

```
import matlab.unittest.TestSuite;
```

Create test suites using each method.

```
fileSuite    = TestSuite.fromFile('SomeTestFile.m');
folderSuite  = TestSuite.fromFolder(pwd);
packageSuite = TestSuite.fromPackage('mypackage.subpackage');
classSuite   = TestSuite.fromClass(?mypackage.MyTestClass);
methodSuite  = TestSuite.fromMethod(?SomeTestClass,'testMethod');
```

Concatenate the suites.

```
largeSuite = [fileSuite, folderSuite, packageSuite, classSuite, methodSuite];
```

Run the full suite.

```
result = run(largeSuite)
```

See Also

`Test` | `TestResult` | `TestRunner`

matlab.unittest.TestSuite.fromClass

Class: `matlab.unittest.TestSuite`
Package: `matlab.unittest`

Create suite from `TestCase` class

Syntax

```
suite = matlab.unittest.TestSuite.fromClass(testClass)
suite = matlab.unittest.TestSuite.fromClass(testClass,s)
suite = matlab.unittest.TestSuite.fromClass( ___ ,Name,Value)
```

Description

`suite = matlab.unittest.TestSuite.fromClass(testClass)` creates a `TestSuite` array from all of the `Test` methods contained in `testClass` and returns that array in `suite`.

`suite = matlab.unittest.TestSuite.fromClass(testClass,s)` creates a `TestSuite` array from all of the `Test` methods contained in `testClass` that satisfy the conditions specified by the selector, `s`.

`suite = matlab.unittest.TestSuite.fromClass( ___ ,Name,Value)` creates a `TestSuite` array with additional options specified by one or more `Name,Value` pair arguments. You can use this syntax with any of the input arguments of the previous syntaxes.

Input Arguments

`testClass`

Class containing test methods, specified as a `meta.class` instance. Use the `?` operator to create a `meta.class` instance. `testClass` must derive from `matlab.unittest.TestCase`.

s

Selector, specified as an instance of a class from the `matlab.unittest.selector` package.

Name-Value Pair Arguments

`BaseFolder`

Character vector or string scalar indicating the name of the parent of the top-level package folder that contains the file defining the test class or function. This argument filters `TestSuite` array elements. For a test element to be included in the suite, the test element must be contained in the specified base folder. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`ExternalParameters`

Array of `matlab.unittest.parameters.Parameter` instances for use in parameterized tests. The framework uses these external parameters in place of the corresponding parameters that are defined within a parameterized test. For more information, see “Use External Parameters in Parameterized Test”.

`Name`

Character vector or string scalar indicating the name of the suite element. This argument filters `TestSuite` array elements. To include a test element in the suite, the `Name` property of the test element must match the specified name. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`ParameterName`

Character vector or string scalar indicating the name of a parameter used by the test suite element. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`ParameterProperty`

Character vector or string scalar indicating the name of a property that defines a parameter used by the test suite element. This argument filters `TestSuite` array

elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`ProcedureName`

Name of test procedure, specified as a character vector or string scalar. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

In a class-based test, the `ProcedureName` is the name of the test method. In a function-based, test it is the name of the local function that contains the test. In a script-based test, it is a name generated from the test section title. Unlike `Name`, the name of the test procedure does not include any class or package name or information about parameterization.

`Superclass`

Name of a class that the test class derives from, specified as a character vector or string scalar. This argument filters `TestSuite` array elements.

`Tag`

Character vector or string scalar indicating the name of the tag applied to the test suite element. This argument filters `TestSuite` array elements. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match exactly one character.

Output Arguments

`suite`

Set of tests, specified as a `matlab.unittest.Test` array.

Attributes

`Static`	`true`

To learn about attributes of methods, see Method Attributes.

Examples

Run Tests in a Package Class

Add the `matlab.unittest.TestSuite` class to the current import list.

```
import matlab.unittest.TestSuite;

suite = TestSuite.fromClass(?mypackage.MyTestClass);
result = run(suite)
```

Run Tests in a Class Without a Package

Add the `matlab.unittest.TestSuite` class to the current import list.

```
import matlab.unittest.TestSuite;

suite = TestSuite.fromClass(?MyTestClass);
result = run(suite)
```

Create Suite of Test Elements Using Selector

In your working folder, create the following `testZeros.m` test file. This class contains four test methods.

```
classdef testZeros < matlab.unittest.TestCase
    properties (TestParameter)
        type = {'single','double','uint16'};
        outSize = struct('s2d',[3 3], 's3d',[2 5 4]);
    end

    methods (Test)
        function testClass(testCase, type, outSize)
            testCase.verifyClass(zeros(outSize,type), type);
        end

        function testSize(testCase, outSize)
            testCase.verifySize(zeros(outSize), outSize);
        end
```

```
        function testDefaultClass(testCase)
            testCase.verifyClass(zeros, 'double');
        end
        function testDefaultSize(testCase)
            testCase.verifySize(zeros, [1 1]);
        end

        function testDefaultValue(testCase)
            testCase.verifyEqual(zeros,0);
        end
    end
end
```

The test class contains two parameterized test methods, `testClass` and `testSize`.

At the command prompt, create a test suite from the test elements that tests the `'double'` data type.

```
import matlab.unittest.TestSuite;
import matlab.unittest.selectors.HasParameter;

suite = TestSuite.fromClass(?testZeros, ...
    HasParameter('Property','type','Name','double'));
{suite.Name}'
```

```
ans =

    'testZeros/testClass(type=double,outSize=s2d)'
    'testZeros/testClass(type=double,outSize=s3d)'
```

Create Suite of Test Elements Using Name-Value Arguments

Create the `testZeros.m` class from the previous example.

At the command prompt, create a test suite from the test elements that tests the `'double'` data type.

```
import matlab.unittest.TestSuite;

suite = TestSuite.fromClass(?testZeros, ...
    'ParameterProperty','type', 'ParameterName','double');
{suite.Name}'
```

```
ans =

    'testZeros/testClass(type=double,outSize=s2d)'
    'testZeros/testClass(type=double,outSize=s3d)'
```

Tips

- `testClass` must be on the MATLAB path when using this method to create `suite`, as well as when `suite` is run.

See Also

`TestRunner` | `fromMethod` | `fromPackage` | `matlab.unittest.selectors`

Introduced in R2013a

matlab.unittest.TestSuite.fromFile

Class: `matlab.unittest.TestSuite`
Package: `matlab.unittest`

Create `TestSuite` array from test file

Syntax

```
suite = matlab.unittest.TestSuite.fromFile(file)
suite = matlab.unittest.TestSuite.fromFile(file,s)
suite = matlab.unittest.TestSuite.fromFile( ___ ,Name,Value)
```

Description

`suite = matlab.unittest.TestSuite.fromFile(file)` creates a `TestSuite` array from all of the tests in `file`. When the test suite is run, MATLAB changes the current folder to the folder that defines the test content, and adds it to the path for the duration of the test run.

`suite = matlab.unittest.TestSuite.fromFile(file,s)` creates a `TestSuite` array from all of the tests in `file` that satisfy the conditions specified by the selector, `s`.

`suite = matlab.unittest.TestSuite.fromFile( ___ ,Name,Value)` creates a `TestSuite` array with additional options specified by one or more `Name,Value` pair arguments. You can use this syntax with any of the input arguments of the previous syntaxes.

Input Arguments

`file`

Absolute or relative path to the test file, specified as a character vector or string scalar.

s

Selector, specified as an instance of a class from the `matlab.unittest.selector` package.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`BaseFolder`

Character vector or string scalar indicating the name of the folder that contains the file defining the tests. This argument filters `TestSuite` array elements. For a test element to be included in the suite, the test element must be contained in the specified base folder. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`ExternalParameters`

Array of `matlab.unittest.parameters.Parameter` instances for use in parameterized tests. The framework uses these external parameters in place of the corresponding parameters that are defined within a parameterized test. For more information, see “Use External Parameters in Parameterized Test”.

`Name`

Character vector or string scalar indicating the name of the suite element. This argument filters `TestSuite` array elements. To include a test element in the suite, the `Name` property of the test element must match the specified name. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`ParameterName`

Character vector or string scalar indicating the name of a parameter used by the test suite element. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`ParameterProperty`

Character vector or string scalar indicating the name of a property that defines a parameter used by the test suite element. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`ProcedureName`

Name of test procedure, specified as a character vector or string scalar. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

In a class-based test, the `ProcedureName` is the name of the test method. In a function-based, test it is the name of the local function that contains the test. In a script-based test, it is a name generated from the test section title. Unlike `Name`, the name of the test procedure does not include any class or package name or information about parameterization.

`Superclass`

Name of a class that the test class derives from, specified as a character vector or string scalar. This argument filters `TestSuite` array elements.

`Tag`

Character vector or string scalar indicating the name of the tag applied to the test suite element. This argument filters `TestSuite` array elements. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match exactly one character.

Output Arguments

`suite`

Set of tests, specified as a `matlab.unittest.Test` array.

Attributes

Static	true

To learn about attributes of methods, see Method Attributes.

Examples

Run Tests in Class File

Function for unit testing:

```
function res = add5(x)
% ADD5 Increment input by 5.
if ~isa(x,'numeric')
    error('add5:InputMustBeNumeric','Input must be numeric.')
end
res = x + 5;
end
```

`TestCase` class containing test methods:

```
classdef Add5Test < matlab.unittest.TestCase
    methods (Test)
        function testDoubleOut(testCase)
            actOutput = add5(1);
            testCase.verifyClass(actOutput,'double')
        end
        function testNonNumericInput(testCase)
            testCase.verifyError(@()add5('0'),'add5:InputMustBeNumeric')
        end
    end
end
```

Create a test suite from the `Add5Test` class file.

```
suite = matlab.unittest.TestSuite.fromFile('Add5Test.m')
```

```
result = run(suite);
```

```
Running Add5Test
..
```

```
Done Add5Test
__________
```

Create Suite of Test Elements Using Selector

In your working folder, create `testZeros.m`. This class contains four test methods.

```
classdef testZeros < matlab.unittest.TestCase
    properties (TestParameter)
        type = {'single','double','uint16'};
        outSize = struct('s2d',[3 3], 's3d',[2 5 4]);
    end

    methods (Test)
        function testClass(testCase, type, outSize)
            testCase.verifyClass(zeros(outSize,type), type);
        end

        function testSize(testCase, outSize)
            testCase.verifySize(zeros(outSize), outSize);
        end

        function testDefaultClass(testCase)
            testCase.verifyClass(zeros, 'double');
        end
        function testDefaultSize(testCase)
            testCase.verifySize(zeros, [1 1]);
        end

        function testDefaultValue(testCase)
            testCase.verifyEqual(zeros,0);
        end
    end
end
```

The test class contains two parameterized test methods, `testClass` and `testSize`.

At the command prompt, create a test suite from all parameterized test methods in `testZeros.m` using the `HasParameter` selector.

```
import matlab.unittest.TestSuite;
import matlab.unittest.selectors.HasParameter;
```

```
suite = TestSuite.fromFile('testZeros.m', HasParameter)
```

```
suite = 

  1×8 Test array with properties:

    Name
    ProcedureName
    TestClass
    BaseFolder
    Parameterization
    SharedTestFixtures
    Tags

Tests Include:
   5 Unique Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

Create a test suite from only the test elements from the `testSize` method using the `HasName` selector with the `StartsWithSubstring` constraint.

```
import matlab.unittest.selectors.HasName;
import matlab.unittest.constraints.StartsWithSubstring;

suite = TestSuite.fromFile('testZeros.m',...
    HasName(StartsWithSubstring('testZeros/testSize')));
{suite.Name}'
```

```
ans = 

    'testZeros/testSize(outSize=s2d)'
    'testZeros/testSize(outSize=s3d)'
```

The test suite contains the two parameterized tests from the `testSize` method.

Create Suite of Test Elements Using Name-Value Arguments

Create the `testZeros.m` class from the previous example.

At the command prompt, create a test suite from all test methods in `testZeros.m` that have a name starting with `'testZeros/testSize'`. This test suite contains parameterized tests from the `testSize` method.

```
import matlab.unittest.TestSuite;

suite = TestSuite.fromFile('testZeros.m', 'Name', 'testZeros/testSize*');
{suite.Name}'
```

```
ans =

    'testZeros/testSize(outSize=s2d)'
    'testZeros/testSize(outSize=s3d)'
```

To ensure that a test suite is comprised of test elements associated with one particular test method, use the `fromMethod` method of `TestSuite`.

At the command prompt, create a test suite from all test methods in `testZeros.m` that have a name ending in `'Size'`.

```
import matlab.unittest.TestSuite;

suite = TestSuite.fromFile('testZeros.m', 'Name', '*Size');
{suite.Name}'
```

```
ans =

    'testZeros/testDefaultSize'
```

Note that elements from the `testSize` method are not included in the test suite. The name of these elements contains information about the parameterization, and therefore it does not end with `'Size'`.

Create a test suite of all tests that use the parameter name `'double'`.

```
suite = TestSuite.fromFile('testZeros.m', 'ParameterName', 'double');
{suite.Name}'
```

```
ans =

    'testZeros/testClass(type=double,outSize=s2d)'
    'testZeros/testClass(type=double,outSize=s3d)'
```

To construct the same test suite using selectors, use `suite = TestSuite.fromFile('testZeros.m', HasParameter('Name','double'))`.

See Also

`TestRunner` | `fromFolder` | `matlab.unittest.selectors`

Introduced in R2013a

matlab.unittest.TestSuite.fromFolder

Class: `matlab.unittest.TestSuite`
Package: `matlab.unittest`

Create `TestSuite` array from all tests in folder

Syntax

```
suite = matlab.unittest.TestSuite.fromFolder(folder)
suite = matlab.unittest.TestSuite.fromFolder(folder,s)
suite = matlab.unittest.TestSuite.fromFolder( ___ ,Name,Value)
```

Description

`suite = matlab.unittest.TestSuite.fromFolder(folder)` creates a `TestSuite` array from all of the `Test` methods of all concrete `TestCase` classes contained in `folder` and returns that array in `suite`. If tests are function-based or script-based, a `Test` file is included in the `TestSuite` array if it follows the naming convention of starting or ending in the word 'test', which is case-insensitive. Class-based tests do not need to follow this naming convention. The method is not recursive, returning only those tests directly in the specified folder.

When the test suite is run, MATLAB changes the current folder to the folder that defines the test content, and adds it to the path for the duration of the test run.

`suite = matlab.unittest.TestSuite.fromFolder(folder,s)` creates a `TestSuite` array from all of the `Test` methods contained in `folder` that satisfy the conditions specified by the selector, `s`.

`suite = matlab.unittest.TestSuite.fromFolder( ___ ,Name,Value)` creates a `TestSuite` array with additional options specified by one or more `Name,Value` pair arguments. You can use this syntax with any of the input arguments of the previous syntaxes.

Input Arguments

`folder`

Folder containing tests, specified as a character vector or string scalar. `folder` can be either an absolute or relative path to the desired folder.

`s`

Selector, specified as an instance of a class from the `matlab.unittest.selector` package.

Name-Value Pair Arguments

`BaseFolder`

Character vector or string scalar indicating the name of the folder that contains the file defining the tests. This argument filters `TestSuite` array elements. For a test element to be included in the suite, the test element must be contained in the specified base folder. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`ExternalParameters`

Array of `matlab.unittest.parameters.Parameter` instances for use in parameterized tests. The framework uses these external parameters in place of the corresponding parameters that are defined within a parameterized test. For more information, see “Use External Parameters in Parameterized Test”.

`IncludingSubfolders`

Indicator for whether to include tests from any `folder` subfolders, excluding package, class, and private folders, specified as `false` or `true` (`logical 0` or `1`). This property is `false` by default. You can specify it as `true` during construction.

`Name`

Character vector or string scalar indicating the name of the suite element. This argument filters `TestSuite` array elements. To include a test element in the suite, the `Name` property of the test element must match the specified name. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`ParameterName`

Character vector or string scalar indicating the name of a parameter used by the test suite element. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`ParameterProperty`

Character vector or string scalar indicating the name of a property that defines a parameter used by the test suite element. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`ProcedureName`

Name of test procedure, specified as a character vector or string scalar. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

In a class-based test, the `ProcedureName` is the name of the test method. In a function-based, test it is the name of the local function that contains the test. In a script-based test, it is a name generated from the test section title. Unlike `Name`, the name of the test procedure does not include any class or package name or information about parameterization.

`Superclass`

Name of a class that the test class derives from, specified as a character vector or string scalar. This argument filters `TestSuite` array elements.

`Tag`

Character vector or string scalar indicating the name of the tag applied to the test suite element. This argument filters `TestSuite` array elements. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match exactly one character.

Output Arguments

suite

Set of tests, specified as a `matlab.unittest.Test` array.

Attributes

`Static`	`true`

To learn about attributes of methods, see Method Attributes.

Examples

Run Tests in Current Folder

Add the `matlab.unittest.TestSuite` class to the current import list.

```
import matlab.unittest.TestSuite;

suite = TestSuite.fromFolder(pwd);
result = run(suite);
```

Run Tests in Subfolders

```
suite = TestSuite.fromFolder(pwd, 'IncludingSubfolders', true);
result = run(suite);
```

Create Suite of Test Elements Using Selector

In your working folder, create a new folder, `myTests`. In that folder, create the following `testZeros.m` test file. This class contains four test methods.

```
classdef testZeros < matlab.unittest.TestCase
    properties (TestParameter)
```

```
        type = {'single','double','uint16'};
        outSize = struct('s2d',[3 3], 's3d',[2 5 4]);
    end

    methods (Test)
        function testClass(testCase, type, outSize)
            testCase.verifyClass(zeros(outSize,type), type);
        end

        function testSize(testCase, outSize)
            testCase.verifySize(zeros(outSize), outSize);
        end

        function testDefaultClass(testCase)
            testCase.verifyClass(zeros, 'double');
        end
        function testDefaultSize(testCase)
            testCase.verifySize(zeros, [1 1]);
        end

        function testDefaultValue(testCase)
            testCase.verifyEqual(zeros,0);
        end
    end
end
```

The test class contains two parameterized test methods, `testClass` and `testSize`.

At the command prompt, create a test suite from all parameterized tests that use the parameter name `'double'`.

```
import matlab.unittest.TestSuite;
import matlab.unittest.selectors.HasParameter;

suite = TestSuite.fromFolder('myTests', HasParameter('Name','double'));
{suite.Name}'
```

```
ans =

    'testZeros/testClass(type=double,outSize=s2d)'
    'testZeros/testClass(type=double,outSize=s3d)'
```

Create Suite of Test Elements Using Name-Value Arguments

Create the `myTests` folder and `testZeros.m` class from the previous example.

Create a test suite of all tests that use the parameter name `'double'`.

```
import matlab.unittest.TestSuite;

suite = TestSuite.fromFolder('myTests', 'ParameterName', 'double');
{suite.Name}'
```

```
ans =

    'testZeros/testClass(type=double,outSize=s2d)'
    'testZeros/testClass(type=double,outSize=s3d)'
```

See Also

`TestRunner` | `fromFile` | `matlab.unittest.selectors`

Introduced in R2013a

matlab.unittest.TestSuite.fromProject

Class: `matlab.unittest.TestSuite`
Package: `matlab.unittest`

Create test suite array from tests in project

Syntax

```
suite = matlab.unittest.TestSuite.fromProject(project)
suite = matlab.unittest.TestSuite.fromProject(project,selector)
suite = matlab.unittest.TestSuite.fromProject(project,Name,Value)
```

Description

`suite = matlab.unittest.TestSuite.fromProject(project)` creates a `TestSuite` array from all test files contained in the specified project that are labeled with the `Test` classification. The `project` input is either a loaded `matlab.project.Project` object or the root folder of a project. This method is not recursive. It includes only those tests in the project specified. To include tests from referenced projects, set `'IncludingReferencedProjects'` to `true`. For more information on projects, see “Projects”.

`suite = matlab.unittest.TestSuite.fromProject(project,selector)` creates a `TestSuite` array from all test files contained in the specified project that are labeled with the `Test` classification and that satisfy the `selector`. For more information on selectors, see `matlab.unittest.selectors Package`.

`suite = matlab.unittest.TestSuite.fromProject(project,Name,Value)` creates a `TestSuite` array from all test files contained in the specified project that are labeled with the `Test` classification and that satisfy the conditions specified by one or more `Name,Value` pair arguments.

Input Arguments

`project` — Project containing test files
path to project root folder | open `Project` object

Project containing test files, specified as the path to the project root folder or an open `Project` object. A test file is a file that is classified as test by adding the `Test` label in the project.

Example: `'C:\MyProjects\ThisProject'`

Data Types: `char` | `string`

`selector` — Filter for `TestSuite` array elements
instance of a class in the `matlab.unittest.selectors` package

Filter for `TestSuite` array elements, specified as an instance of a selector class from the `matlab.unittest.selectors Package`.

Example:
`matlab.unittest.selectors.HasBaseFolder(fullfile(pwd,'MyTests','Feature1'))`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`BaseFolder` — Folder containing tests to include in `TestSuite` array
path to folder

Folder containing tests to include in `TestSuite` array, specified as an absolute or relative path. Use this argument to filter `TestSuite` array elements to those tests in the specified folder. For a test element to be included in the suite, the test element must be contained in the specified base folder. Use the wildcard character, *, to match any number of characters. Use the question mark character, ?, to match to exactly one character.

Data Types: `char` | `string`

`ExternalParameters` — External parameters to use with test
instance of a class in the `matlab.unittest.parameter` package

External parameters to use with test, specified as an instance of a class in the `matlab.unittest.parameters Package`. Parameter instances provide external data for use in parameterized tests. The framework uses these external parameters in place of the corresponding parameters that are defined within a parameterized test. For more information, see “Use External Parameters in Parameterized Test”.

`IncludingReferencedProjects` — Indicator to include tests from referenced projects
false (default) | true | `0` | `1`

Indicate whether to include tests from referenced projects in the `TestSuite`, specified as logical `true` or `false`. By default, `fromProject` includes test files only from the project specified in the input. Passing a value of `true` for `IncludingReferencedProjects` results in a `TestSuite` array that includes the tests from the project specified in the input and tests from projects referenced from the parent project. For more information on referenced projects, see “Componentize Large Projects”.

Example: `suite = matlab.unittest.TestSuite.fromProject(project,'IncludingReferencedProjects',true);`

Data Types: `logical`

`Name` — Name of test element in test suite array
name of a test element

Name of test element in `TestSuite` array, specified as a character vector or string scalar. Use this argument to filter `TestSuite` array elements to those tests that have the specified name. To include a test element in the suite, the `Name` property of the test element must match the specified value for `Name`. Use the wildcard character, *, to match any number of characters. Use the question mark character, ?, to match to exactly one character.

Example: `suite = matlab.unittest.TestSuite.fromProject(project,'Name','testZeros/testSize');`

Data Types: `char` | `string`

`ParameterName` — Name of parameter used by test suite array element
name of a parameter

Name of a parameter used by the test suite element, specified as a character vector or a string scalar. Use this argument to filter `TestSuite` array elements to those tests that

use the specified parameter. Use the wildcard character, *, to match any number of characters. Use the question mark character, ?, to match to exactly one character.

Data Types: `char` | `string`

`ParameterProperty` — Name of property that defines a parameter
name of property

Name of property that defines a parameter used by `TestSuite` array element, specified as character vector or a string scalar. Use this argument to filter `TestSuite` array elements to those tests that have the named property. Use the wildcard character, *, to match any number of characters. Use the question mark character, ?, to match to exactly one character.

Data Types: `char` | `string`

`ProcedureName` — Name of test procedure
test name

Name of test procedure, specified as a character vector or a string scalar. Use this argument to filter `TestSuite` array elements to those that have the named test procedure. Use the wildcard character, *, to match any number of characters. Use the question mark character, ?, to match to exactly one character.

Data Types: `char` | `string`

`Superclass` — Name of the test class superclass
superclass name

Name of the test class superclass, specified as a character vector or string scalar. Use this argument to filter `TestSuite` array elements to those tests that derive from the specified superclass.

Data Types: `char` | `string`

`Tag` — Name of the tag applied to test suite array element
test tag

Name of the tag applied to `TestSuite` array element, specified as a character array or string scalar. Use this argument to filter `TestSuite` array elements to those tests with the specified tag. Use the wildcard character, *, to match any number of characters. Use the question mark character, ?, to match to exactly one character.

Data Types: `char` | `string`

Output Arguments

`suite` — Set of tests
test suite

Set of tests, returned as a `matlab.unittest.Test` array

Attributes

`Static`	`true`

To learn about attributes of methods, see Method Attributes.

Examples

Build Test Suite from Project Files

Build a test suite from project files that are labeled as Test files. This example assumes that a project folder at `C:/projects/project1` contains test files that are labeled with the `Test` classification. Use the `matlab.unittest.TestSuite.fromProject` static method to create a test suite using those tests.

Open `project1` and pass the `matlab.project.Project` object to `fromProject`. Run the test suite and capture the results.

```
import matlab.unittest.TestSuite
project = openProject('C:/projects/project1/');
suite = TestSuite.fromProject(project);
result = run(suite)
```

Include Tests from Referenced Projects

Build a test suite from project files that are labeled as Test files in the project and all referenced projects.

```
import matlab.unittest.TestSuite
project = openProject('C:/projects/project1/');
```

```
suite = TestSuite.fromProject(project,'IncludingReferencedProjects',true);
result = run(suite)
```

See Also

matlab.unittest.Test | matlab.unittest.TestRunner | matlab.unittest.TestSuite.fromFolder | matlab.unittest.TestSuite.fromPackage

Introduced in R2019a

matlab.unittest.TestSuite.fromMethod

Class: `matlab.unittest.TestSuite`
Package: `matlab.unittest`

Create `TestSuite` array from single test method

Syntax

```
suite = matlab.unittest.TestSuite.fromMethod(testClass,testMethod)
suite = matlab.unittest.TestSuite.fromMethod(testClass,testMethod,s)
suite = matlab.unittest.TestSuite.fromMethod( ___ ,Name,Value)
```

Description

`suite = matlab.unittest.TestSuite.fromMethod(testClass,testMethod)` creates a `TestSuite` array from the test class described by `testClass` and the test method described by `testMethod` and returns it in `suite`.

`suite = matlab.unittest.TestSuite.fromMethod(testClass,testMethod,s)` creates a `TestSuite` array from all of the `Test` methods contained in `testMethod` that satisfy the conditions specified by the selector, `s`.

`suite = matlab.unittest.TestSuite.fromMethod( ___ ,Name,Value)` creates a `TestSuite` array with additional options specified by one or more `Name,Value` pair arguments. You can use this syntax with any of the input arguments of the previous syntaxes.

Input Arguments

`testClass`

Class describing the test methods, specified as a `meta.class` instance which must derive from `matlab.unittest.TestCase`.

`testMethod`

Test method, specified by either the `meta.method` instance or the name as a character vector or string scalar. The method must be defined with a `Test` method attribute.

`s`

Selector, specified as an instance of a class from the `matlab.unittest.selector` package.

Name-Value Pair Arguments

`BaseFolder`

Character vector or string scalar indicating the name of the parent of the top-level package folder that contains the file defining the test class or function. This argument filters `TestSuite` array elements. For a test element to be included in the suite, the test element must be contained in the specified base folder. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`ExternalParameters`

Array of `matlab.unittest.parameters.Parameter` instances for use in parameterized tests. The framework uses these external parameters in place of the corresponding parameters that are defined within a parameterized test. For more information, see “Use External Parameters in Parameterized Test”.

`Name`

Character vector or string scalar indicating the name of the suite element. This argument filters `TestSuite` array elements. To include a test element in the suite, the `Name` property of the test element must match the specified name. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`ParameterName`

Character vector or string scalar indicating the name of a parameter used by the test suite element. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`ParameterProperty`

Character vector or string scalar indicating the name of a property that defines a parameter used by the test suite element. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`ProcedureName`

Name of test procedure, specified as a character vector or string scalar. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

In a class-based test, the `ProcedureName` is the name of the test method. In a function-based, test it is the name of the local function that contains the test. In a script-based test, it is a name generated from the test section title. Unlike `Name`, the name of the test procedure does not include any class or package name or information about parameterization.

`Superclass`

Name of a class that the test class derives from, specified as a character vector or string scalar. This argument filters `TestSuite` array elements.

`Tag`

Character vector or string scalar indicating the name of the tag applied to the test suite element. This argument filters `TestSuite` array elements. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match exactly one character.

Output Arguments

`suite`

Set of tests, specified as a `matlab.unittest.Test` array.

Attributes

Static	true

To learn about attributes of methods, see Method Attributes.

Examples

Run a Single Test Method

Add the `matlab.unittest.TestSuite` class to the current import list.

```
import matlab.unittest.TestSuite;

cls = ?mypackage.MyTestClass;

% Create the suite using the method name
suite = TestSuite.fromMethod(cls, 'testMethod');
result = run(suite)

% Create the suite using the meta.method instance
metaMethod = findobj(cls.MethodList, 'Name', 'testMethod');
suite = TestSuite.fromMethod(cls, metaMethod);
result = run(suite)
```

Create Suite of Test Elements Using Selector

In your working folder, create the following `testZeros.m` test file. This class contains four test methods.

```
classdef testZeros < matlab.unittest.TestCase
    properties (TestParameter)
        type = {'single','double','uint16'};
        outSize = struct('s2d',[3 3], 's3d',[2 5 4]);
    end

    methods (Test)
        function testClass(testCase, type, outSize)
            testCase.verifyClass(zeros(outSize,type), type);
```

```
        end

        function testSize(testCase, outSize)
            testCase.verifySize(zeros(outSize), outSize);
        end

        function testDefaultClass(testCase)
            testCase.verifyClass(zeros, 'double');
        end
        function testDefaultSize(testCase)
            testCase.verifySize(zeros, [1 1]);
        end

        function testDefaultValue(testCase)
            testCase.verifyEqual(zeros,0);
        end
    end
end
```

The test class contains two parameterized test methods, `testClass` and `testSize`.

At the command prompt, create a test suite from all parameterized tests from the `testClass` method that use the parameter name `'single'`.

```
import matlab.unittest.TestSuite;
import matlab.unittest.selectors.HasParameter;

suite = TestSuite.fromMethod(?testZeros,'testClass', ...
    HasParameter('Name','single'));
{suite.Name}'
```

```
ans =

    'testZeros/testClass(type=single,outSize=s2d)'
    'testZeros/testClass(type=single,outSize=s3d)'
```

Create Suite of Test Elements Using Name-Value Arguments

Create the `testZeros.m` class from the previous example.

At the command prompt, create a test suite from all parameterized tests from the `testClass` method that use the parameter name `'single'`.

```
import matlab.unittest.TestSuite;

suite = TestSuite.fromMethod(?testZeros,'testClass', ...
    'ParameterName','single');
{suite.Name}'
```

```
ans =

    'testZeros/testClass(type=single,outSize=s2d)'
    'testZeros/testClass(type=single,outSize=s3d)'
```

Tips

- `testClass` must be on the MATLAB path when using this method to create `suite`, as well as when `suite` is run.

See Also

`TestRunner` | `fromClass` | `fromPackage` | `matlab.unittest.selectors`

Introduced in R2013a

matlab.unittest.TestSuite.fromName

Class: `matlab.unittest.TestSuite`
Package: `matlab.unittest`

Create `Test` object from name of test element

Syntax

```
testObj = matlab.unittest.TestSuite.fromName(name)
testObj = matlab.unittest.TestSuite.fromName(
name,'ExternalParameters',ExternalParameters)
```

Description

`testObj = matlab.unittest.TestSuite.fromName(name)` creates a scalar `Test` object, `testObj`, from the name of the test element, `name`.

`testObj = matlab.unittest.TestSuite.fromName(name,'ExternalParameters',ExternalParameters)` allows the suite to use the specified external parameters in place of corresponding parameters that are defined within a parameterized test.

Input Arguments

`name`

Name of the `matlab.unittest.Test` element, specified as a character vector or string scalar. For class-based tests, `name` contains the name of the `TestCase` class and the test method, as well as information about parameterization. For function-based tests, `name` contains the name of the main function and the local test function. For script-based tests, `name` contains the name of the script and the title of the test section or cell. If the section does not have a title, MATLAB assigns one. The `name` argument corresponds to the `Name` property of the `Test` object.

The test class, function or script described by `name` must be on the MATLAB path when you are creating and running the `TestSuite`.

`ExternalParameters`

Array of `matlab.unittest.parameters.Parameter` instances for use in parameterized tests. The framework uses these external parameters in place of the corresponding parameters that are defined within a parameterized test. For more information, see “Use External Parameters in Parameterized Test”.

Attributes

`Static`	`true`

To learn about attributes of methods, see Method Attributes.

Examples

Create and Run Test from Test Suite Element

Create a function to test, `add5`, in a file on your MATLAB path.

```
function res = add5(x)
% ADD5 Increment input by 5.
if ~isa(x,'numeric')
    error('add5:InputMustBeNumeric','Input must be numeric.')
end
res = x + 5;
end
```

Create a file, `Add5Test.m`, on your MATLAB path that contains the following `TestCase` class.

```
classdef Add5Test < matlab.unittest.TestCase
    properties (TestParameter)
        Type = {'double','single','int8','int32'};
    end

    methods (Test)
```

```
        function testNonNumericInput(testCase)
            testCase.verifyError(@()add5('0'),'add5:InputMustBeNumeric')
        end
        function testResultType(testCase, Type)
            actOutput = add5(cast(1,Type));
            testCase.verifyClass(actOutput, Type)
        end

    end
end
```

At the command prompt, create a test object for the `testNonNumericInput` method in the `Add5Test` class.

```
import matlab.unittest.TestSuite
testObj = TestSuite.fromName('Add5Test/testNonNumericInput');
```

Run the test

```
result = run(testObj);
```

```
Running Add5Test
.
Done Add5Test
__________
```

Create a parameterized test for the `testResultType` method in the `Add5Test` class, and run the test.

```
testObj = TestSuite.fromName('Add5Test/testResultType(Type=single)');
result = run(testObj);
```

```
Running Add5Test
.
Done Add5Test
__________
```

See Also

`TestRunner`

matlab.unittest.TestSuite.fromPackage

Class: `matlab.unittest.TestSuite`
Package: `matlab.unittest`

Create `TestSuite` array from all tests in package

Syntax

```
suite = matlab.unittest.TestSuite.fromPackage(package)
suite = matlab.unittest.TestSuite.fromPackage(package,s)
suite = matlab.unittest.TestSuite.fromPackage( ___ ,Name,Value)
```

Description

`suite = matlab.unittest.TestSuite.fromPackage(package)` creates a `TestSuite` array from all of the `Test` methods of all the tests contained in `package` and returns that array in `suite`. The method is not recursive, returning only those tests directly in the package specified.

`suite = matlab.unittest.TestSuite.fromPackage(package,s)` creates a `TestSuite` array from all the tests contained in `package` that satisfy the conditions specified by the selector, `s`.

`suite = matlab.unittest.TestSuite.fromPackage( ___ ,Name,Value)` creates a `TestSuite` array with additional options specified by one or more `Name,Value` pair arguments. You can use this syntax with any of the input arguments of the previous syntaxes.

Input Arguments

`package`

The name of the desired package to find tests, specified as a character vector or string scalar.

`s`

Selector, specified as an instance of a class from the `matlab.unittest.selector` package.

Name-Value Pair Arguments

`BaseFolder`

Character vector or string scalar indicating the name of the parent of the top-level package folder that contains the tests. This argument filters `TestSuite` array elements. For a test element to be included in the suite, the test element must be contained in the specified base folder. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`ExternalParameters`

Array of `matlab.unittest.parameters.Parameter` instances for use in parameterized tests. The framework uses these external parameters in place of the corresponding parameters that are defined within a parameterized test. For more information, see “Use External Parameters in Parameterized Test”.

`IncludingSubpackages`

Indicator for whether to include subpackages in the `TestSuite` array, specified as `false` or `true` (`logical 0` or `1`). This property is `false` by default. You can specify it as `true` during construction.

`Name`

Character vector or string scalar indicating the name of the suite element. This argument filters `TestSuite` array elements. To include a test element in the suite, the `Name` property of the test element must match the specified name. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`ParameterName`

Character vector or string scalar indicating the name of a parameter used by the test suite element. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

ParameterProperty

Character vector or string scalar indicating the name of a property that defines a parameter used by the test suite element. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

ProcedureName

Name of test procedure, specified as a character vector or string scalar. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

In a class-based test, the `ProcedureName` is the name of the test method. In a function-based, test it is the name of the local function that contains the test. In a script-based test, it is a name generated from the test section title. Unlike `Name`, the name of the test procedure does not include any class or package name or information about parameterization.

Superclass

Name of a class that the test class derives from, specified as a character vector or string scalar. This argument filters `TestSuite` array elements.

Tag

Character vector or string scalar indicating the name of the tag applied to the test suite element. This argument filters `TestSuite` array elements. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match exactly one character.

Output Arguments

suite

Set of tests, specified as a `matlab.unittest.Test` array.

Attributes

Static	true

To learn about attributes of methods, see Method Attributes.

Examples

Run All Tests in a Package

Add the `matlab.unittest.TestSuite` class to the current import list.

```
import matlab.unittest.TestSuite;
```

```
suite = TestSuite.fromPackage('mypackage.subpackage');
result = run(suite)
```

Run tests in `mypackage` including all subpackages.

```
suite = TestSuite.fromPackage('mypackage','IncludingSubpackages',true);
result = run(suite)
```

Create Suite of Test Elements Using Selector

In your working folder, create a new package by creating a new folder, `+myPackage`. In that folder, create the following `testZeros.m` test file. This class contains four test methods.

```
classdef testZeros < matlab.unittest.TestCase
    properties (TestParameter)
        type = {'single','double','uint16'};
        outSize = struct('s2d',[3 3], 's3d',[2 5 4]);
    end

    methods (Test)
        function testClass(testCase, type, outSize)
            testCase.verifyClass(zeros(outSize,type), type);
        end
```

```
        function testSize(testCase, outSize)
            testCase.verifySize(zeros(outSize), outSize);
        end

        function testDefaultClass(testCase)
            testCase.verifyClass(zeros, 'double');
        end
        function testDefaultSize(testCase)
            testCase.verifySize(zeros, [1 1]);
        end

        function testDefaultValue(testCase)
            testCase.verifyEqual(zeros,0);
        end
    end
end
```

The test class contains two parameterized test methods, `testClass` and `testSize`.

At the command prompt, create a test suite from all parameterized tests that use the parameter property `'outSize'`.

```
import matlab.unittest.TestSuite;
import matlab.unittest.selectors.HasParameter;

suite = TestSuite.fromPackage('myPackage', ...
    HasParameter('Property','outSize'));
{suite.Name}'
```

```
ans =

    'myPackage.testZeros/testClass(type=single,outSize=s2d)'
    'myPackage.testZeros/testClass(type=single,outSize=s3d)'
    'myPackage.testZeros/testClass(type=double,outSize=s2d)'
    'myPackage.testZeros/testClass(type=double,outSize=s3d)'
    'myPackage.testZeros/testClass(type=uint16,outSize=s2d)'
    'myPackage.testZeros/testClass(type=uint16,outSize=s3d)'
    'myPackage.testZeros/testSize(outSize=s2d)'
    'myPackage.testZeros/testSize(outSize=s3d)'
```

Create Suite of Test Elements Using Name-Value Arguments

Create the `+myPackage` folder and `testZeros.m` class from the previous example.

At the command prompt, create a test suite from all parameterized tests that use the parameter property `'outSize'`.

```
import matlab.unittest.TestSuite;

suite = TestSuite.fromPackage('myPackage', ...
    'ParameterProperty', 'outSize');
{suite.Name}'
```

```
ans =

    'myPackage.testZeros/testClass(type=single,outSize=s2d)'
    'myPackage.testZeros/testClass(type=single,outSize=s3d)'
    'myPackage.testZeros/testClass(type=double,outSize=s2d)'
    'myPackage.testZeros/testClass(type=double,outSize=s3d)'
    'myPackage.testZeros/testClass(type=uint16,outSize=s2d)'
    'myPackage.testZeros/testClass(type=uint16,outSize=s3d)'
    'myPackage.testZeros/testSize(outSize=s2d)'
    'myPackage.testZeros/testSize(outSize=s3d)'
```

Tips

- The root folder(s) where `package` is defined must be on the MATLAB path when creating `suite` using this method as well as when `suite` is run.

See Also

`TestRunner` | `fromClass` | `fromMethod` | `matlab.unittest.selectors`

Introduced in R2013a

run

Class: `matlab.unittest.TestSuite`
Package: `matlab.unittest`

Run `TestSuite` array using `TestRunner` object configured for text output

Syntax

```
result = run(suite)
```

Description

`result = run(suite)` runs the `TestSuite` object defined by `suite` using a `TestRunner` object configured for text output.

Input Arguments

`suite`

Set of tests, specified as a `matlab.unittest.TestSuite` array.

Output Arguments

`result`

A `matlab.unittest.TestResult` object containing the result of the test run. `result` is the same size as `suite` and each element is the result of the corresponding element in `suite`.

Examples

Compare TestSuite.run with TestRunner.run

Add `matlab.unittest` classes to the current import list.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;
```

Create a test suite and a test runner.

```
suite = TestSuite.fromClass(?mypackage.MyTestClass);
runner = TestRunner.withTextOutput;
```

The following test results are equivalent.

```
result = runner.run(suite)
result = run(suite)
```

Tips

- This is a convenience method which is equivalent to using a `TestRunner` object created from the `TestRunner.withTextOutput` method to run `suite`.

See Also

`run` | `run`

selectIf

Class: `matlab.unittest.TestSuite`
Package: `matlab.unittest`

Select test suite elements that satisfy conditions

Syntax

```
newsuite = selectIf(suite,s)
newsuite = selectIf(suite,Name,Value)
```

Description

`newsuite = selectIf(suite,s)` selects from `suite` the test elements that satisfy the conditions specified by the selector, `s`, and returns them in the `TestSuite` array, `newsuite`.

`newsuite = selectIf(suite,Name,Value)` creates a `TestSuite` array with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`suite`

Set of tests, specified as a `matlab.unittest.TestSuite` array.

`s`

Selector, specified as an instance of a class from the `matlab.unittest.selector` package.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`BaseFolder`

Character vector indicating the name of the folder that contains the tests. This argument filters `TestSuite` array elements. For a test element to be included in the suite, the test element must be contained in the specified base folder. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character. For tests defined in packages, the base folder is the parent of the top-level package folder.

`Name`

Character vector indicating the name of the suite element. This argument filters `TestSuite` array elements. To include a test element in the suite, the `Name` property of the test element must match the specified name. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`ParameterName`

Character vector indicating the name of a parameter used by the test suite element. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`ParameterProperty`

Character vector indicating the name of a property that defines a parameter used by the test suite element. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

`ProcedureName`

Name of test procedure, specified as a character vector or string scalar. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match to exactly one character.

In a class-based test, the `ProcedureName` is the name of the test method. In a function-based, test it is the name of the local function that contains the test. In a script-based

test, it is a name generated from the test section title. Unlike `Name`, the name of the test procedure does not include any class or package name or information about parameterization.

`Superclass`

Name of a class that the test class derives from, specified as a character vector or string scalar. This argument filters `TestSuite` array elements.

`Tag`

Character vector or string scalar indicating the name of the tag applied to the test suite element. This argument filters `TestSuite` array elements. This argument filters `TestSuite` array elements. Use the wildcard character, `*`, to match any number of characters. Use the question mark character, `?`, to match exactly one character.

Examples

Select Test Elements Using Selector

In your working folder, create the file `ExampleTest.m` containing the following test class.

```
classdef (SharedTestFixtures={...
        matlab.unittest.fixtures.PathFixture(fullfile(...
        matlabroot, 'help', 'techdoc', 'matlab_oop', 'examples'))})...
        ExampleTest < matlab.unittest.TestCase
    methods(Test)
        function testPathAdd(testCase)
            % test code
        end
        function testOne(testCase)
            % test code
        end
         function testTwo(testCase)
            % test code
        end
    end
end
```

At the command prompt, create a test suite from the `ExampleTest` class.

```
import matlab.unittest.TestSuite;
import matlab.unittest.selectors.HasSharedTestFixture;
import matlab.unittest.selectors.HasName;
import matlab.unittest.fixtures.PathFixture;
import matlab.unittest.constraints.EndsWithSubstring;
import matlab.unittest.constraints.ContainsSubstring;

suite = TestSuite.fromClass(?ExampleTest)
```

```
suite =

  1×3 Test array with properties:

    Name
    ProcedureName
    TestClass
    BaseFolder
    Parameterization
    SharedTestFixtures
    Tags

Tests Include:
   0 Parameterizations, 1 Unique Shared Test Fixture Class, 0 Tags.
```

The test suite contains three test elements.

Create a filtered test suite of tests comprising tests with names that contain the case-insensitive text `'pAtH'`.

```
newSuite = selectIf(suite,HasName(ContainsSubstring('pAtH','IgnoringCase',true)))
```

```
newSuite =

  Test with properties:

                  Name: 'ExampleTest/testPathAdd'
         ProcedureName: 'testPathAdd'
             TestClass: "ExampleTest"
            BaseFolder: 'C:\work'
      Parameterization: [0×0 matlab.unittest.parameters.EmptyParameter]
    SharedTestFixtures: [1×1 matlab.unittest.fixtures.PathFixture]
                  Tags: {1×0 cell}

Tests Include:
   0 Parameterizations, 1 Unique Shared Test Fixture Class, 0 Tags.
```

Only the `testPathAdd` test is part of the suite.

Alternatively, create the same suite using a name-value pair.

```
newSuite = selectIf(suite,'Name','*Path*');
```

However, unlike the `ContainsSubstring` constraint, the name-value pair does not have an option to ignore case.

Create a filtered suite of tests comprising tests that use a shared path fixture and do not have names ending with `'One'`.

```
newSuite = suite.selectIf(~HasName(EndsWithSubstring('One')) ...
    & HasSharedTestFixture(PathFixture(fullfile(matlabroot, 'help',...
    'techdoc', 'matlab_oop', 'examples'))));
{newSuite.Name}
```

```
ans =

    'ExampleTest/testPathAdd'    'ExampleTest/testTwo'
```

The test suite contains two tests. All of the tests use the specified path fixture, but the test named `'testOne'` is excluded from the suite.

Select Test Elements Using Parameterization

In your working folder, create `testZeros.m`. This class contains four test methods.

```
classdef testZeros < matlab.unittest.TestCase
    properties (TestParameter)
        type = {'single','double','uint16'};
        outSize = struct('s2d',[3 3], 's3d',[2 5 4]);
    end

    methods (Test)
        function testClass(testCase, type, outSize)
            testCase.verifyClass(zeros(outSize,type), type);
        end

        function testSize(testCase, outSize)
            testCase.verifySize(zeros(outSize), outSize);
        end
```

```
        function testDefaultClass(testCase)
            testCase.verifyClass(zeros, 'double');
        end
        function testDefaultSize(testCase)
            testCase.verifySize(zeros, [1 1]);
        end

        function testDefaultValue(testCase)
            testCase.verifyEqual(zeros,0);
        end
    end
end
```

The test class contains two parameterized test methods, `testClass` and `testSize`.

At the command prompt, create a test suite from the file.

```
s = matlab.unittest.TestSuite.fromFile('testZeros.m');
{s.Name}'
```

```
ans =

  11×1 cell array

    {'testZeros/testClass(type=single,outSize=s2d)'}
    {'testZeros/testClass(type=single,outSize=s3d)'}
    {'testZeros/testClass(type=double,outSize=s2d)'}
    {'testZeros/testClass(type=double,outSize=s3d)'}
    {'testZeros/testClass(type=uint16,outSize=s2d)'}
    {'testZeros/testClass(type=uint16,outSize=s3d)'}
    {'testZeros/testSize(outSize=s2d)'             }
    {'testZeros/testSize(outSize=s3d)'             }
    {'testZeros/testDefaultClass'                  }
    {'testZeros/testDefaultSize'                   }
    {'testZeros/testDefaultValue'                  }
```

The suite contains 11 test elements. Six from the parameterized `testClass` method, two from the parameterized `testSize` method, and one from each of the `testDefaultClass`, `testDefaultSize`, and `testDefaultValue` methods.

Select all of the test elements from parameterized test methods.

```
import matlab.unittest.selectors.HasParameter;
```

```
s1 = s.selectIf(HasParameter);
{s1.Name}'

ans =

  8×1 cell array

    {'testZeros/testClass(type=single,outSize=s2d)'}
    {'testZeros/testClass(type=single,outSize=s3d)'}
    {'testZeros/testClass(type=double,outSize=s2d)'}
    {'testZeros/testClass(type=double,outSize=s3d)'}
    {'testZeros/testClass(type=uint16,outSize=s2d)'}
    {'testZeros/testClass(type=uint16,outSize=s3d)'}
    {'testZeros/testSize(outSize=s2d)'             }
    {'testZeros/testSize(outSize=s3d)'             }
```

The suite contains the eight test elements from the two parameterized test methods.

Select all of the test elements from non-parameterized test methods.

```
s2 = s.selectIf(~HasParameter);
{s2.Name}'

ans =

  3×1 cell array

    {'testZeros/testDefaultClass'}
    {'testZeros/testDefaultSize' }
    {'testZeros/testDefaultValue'}
```

Select all test elements that are parameterized and have a property named `'type'` with a parameter name `'double'`.

```
s3 = s.selectIf('ParameterProperty','type', 'ParameterName','double');
{s3.Name}'

ans =

  2×1 cell array

    {'testZeros/testClass(type=double,outSize=s2d)'}
    {'testZeros/testClass(type=double,outSize=s3d)'}
```

The resulting suite contains two elements. The `testClass` method is the only method in `testZeros` that uses the `'type'` property, and selecting only `'double'` from the parameters results in two test elements — one for each value of `'outSize'`.

Select all test elements that are parameterized and have a parameters defined by a property starting with `'t'`.

```
s4 = s.selectIf('ParameterProperty','t*');
{s4.Name}'
```

```
ans =

  6×1 cell array

    {'testZeros/testClass(type=single,outSize=s2d)'}
    {'testZeros/testClass(type=single,outSize=s3d)'}
    {'testZeros/testClass(type=double,outSize=s2d)'}
    {'testZeros/testClass(type=double,outSize=s3d)'}
    {'testZeros/testClass(type=uint16,outSize=s2d)'}
    {'testZeros/testClass(type=uint16,outSize=s3d)'}
```

The resulting suite contains the six parameterized test elements from the `testClass` method. The `testSize` method is parameterized, but the elements from the method are not included in the suite because the method does not use a property that starts with `'t'`.

Select all test elements that are parameterized and test the `zeros` function with a `2-D` array. A parameter value representing a `2-D` array will have a length of `1` (e.g. `zeros(3)`) or `2` (e.g. `zeros(2,3)`).

```
import matlab.unittest.constraints.HasLength;

s5 = s.selectIf(HasParameter('Property','outSize',...
    'Value', HasLength(1)|HasLength(2)));
{s5.Name}'
```

```
ans =

  4×1 cell array

    {'testZeros/testClass(type=single,outSize=s2d)'}
    {'testZeros/testClass(type=double,outSize=s2d)'}
    {'testZeros/testClass(type=uint16,outSize=s2d)'}
    {'testZeros/testSize(outSize=s2d)'              }
```

Select only the test element that tests that the output is a `double` data type and the has the correct size for a `2-D` array.

```
s6 = s.selectIf(HasParameter('Property','type','Name','double')...
    & HasParameter('Property','outSize','Name','s2d'))
```

```
s6 =

  Test with properties:

                  Name: 'testZeros/testClass(type=double,outSize=s2d)'
         ProcedureName: 'testClass'
             TestClass: "testZeros"
            BaseFolder: 'C:\work'
      Parameterization: [1×2 matlab.unittest.parameters.TestParameter]
    SharedTestFixtures: [0×0 matlab.unittest.fixtures.EmptyFixture]
                  Tags: {1×0 cell}

Tests Include:
   2 Unique Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

Select Test Elements Using Tags

Create the following test class in a file, `ExampleTest.m`, in your current folder.

```
classdef ExampleTest < matlab.unittest.TestCase
    methods (Test)
        function testA (testCase)
            % test code
        end
    end
    methods (Test, TestTags = {'Unit'})
        function testB (testCase)
            % test code
        end
        function testC (testCase)
            % test code
        end
    end
    methods (Test, TestTags = {'Unit','FeatureA'})
        function testD (testCase)
            % test code
        end
```

```
    end
    methods (Test, TestTags = {'System','FeatureA'})
        function testE (testCase)
            % test code
        end
    end
end
```

At the command prompt, create a test suite from the `ExampleTest` class and examine the contents.

```
import matlab.unittest.TestSuite
import matlab.unittest.selectors.HasTag

suite = TestSuite.fromClass(?ExampleTest)
```

```
suite = 

  1×5 Test array with properties:

    Name
    ProcedureName
    TestClass
    BaseFolder
    Parameterization
    SharedTestFixtures
    Tags

Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 3 Unique Tags.
```

Click the hyperlink for `3 Unique Tags` to display all the tags in the suite.

```
        Tag
    ____________

    {'FeatureA'}
    {'System'  }
    {'Unit'    }
```

Select all the test suite elements that have the tag `'Unit'`.

```
s1 = suite.selectIf(HasTag('Unit'))
```

```
s1 = 
```

```
  1×3 Test array with properties:

    Name
    ProcedureName
    TestClass
    BaseFolder
    Parameterization
    SharedTestFixtures
    Tags

Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 2 Unique Tags.
```

Select all the test suite elements that do not contain the tag `'FeatureA'`.

```
s2 =  suite.selectIf(~HasTag('FeatureA'));
{s2.Name}
```

```
ans =

  1×3 cell array

    {'ExampleTest/testB'}    {'ExampleTest/testC'}    {'ExampleTest/testA'}
```

Select all the test suite elements that have no tags.

```
s3 =  suite.selectIf(~HasTag)
```

```
s3 =

  Test with properties:

                  Name: 'ExampleTest/testA'
         ProcedureName: 'testA'
             TestClass: "ExampleTest"
            BaseFolder: 'C:\work'
      Parameterization: [0×0 matlab.unittest.parameters.EmptyParameter]
    SharedTestFixtures: [0×0 matlab.unittest.fixtures.EmptyFixture]
                  Tags: {1×0 cell}

Tests Include:
   0 Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

See Also

`matlab.unittest.constraints` | `matlab.unittest.selectors`

Introduced in R2014a

sortByFixtures

Class: `matlab.unittest.TestSuite`
Package: `matlab.unittest`

Reorder test suite based on shared fixtures

Syntax

```
sortedSuite = sortByFixtures(suite)
[sortedSuite,I] = sortByFixtures(suite)
```

Description

`sortedSuite = sortByFixtures(suite)` reorders the test suite to reduce shared fixture setup and teardown operations. Do not rely on the order of elements in `sortedSuite` as it might change in a future release.

`[sortedSuite,I] = sortByFixtures(suite)` also returns a sort index `I` that describes the arrangement of the elements of `suite` into `sortedSuite`.

Input Arguments

`suite` — Set of tests
`matlab.unittest.Test` array

Set of tests, specified as a `matlab.unittest.Test` array.

Output Arguments

`sortedSuite` — Ordered set of tests
`matlab.unittest.Test` array

Ordered set of tests, returned as a `matlab.unittest.Test` array. `sortedSuite` is a permutation of the test elements of `suite`.

I — Sort index
vector | matrix | multidimensional array

Sort index, returned as a vector, matrix, or multidimensional array. `I` is the same size as `suite` and describes the arrangement of the elements of `suite` into `orderedSuite`. Specifically, `sortedSuite = suite(I)`.

Examples

Reorder Test Suite Based on Shared Fixtures

Create three test classes in your current working folder. Test classes `MyTestClassA` and `MyTestClassC` use the same shared path fixture.

```
classdef (SharedTestFixtures={ ...
        matlab.unittest.fixtures.PathFixture('offPathFolder')}) ...
        MyTestClassA < matlab.unittest.TestCase
    methods (Test)
        function test_A(testCase)
            % test content
        end
    end
end

classdef MyTestClassB < matlab.unittest.TestCase
    methods (Test)
        function test_B(testCase)
            % test content
        end
    end
end

classdef (SharedTestFixtures={ ...
        matlab.unittest.fixtures.PathFixture('offPathFolder')}) ...
        MyTestClassC < matlab.unittest.TestCase
    methods (Test)
        function test_C(testCase)
            % test content
        end
    end
end
```

Create a test suite from each class.

```
import matlab.unittest.TestSuite;
suiteA = TestSuite.fromClass(?MyTestClassA);
suiteB = TestSuite.fromClass(?MyTestClassB);
suiteC = TestSuite.fromClass(?MyTestClassC);
```

Concatenate the suites and view the order of test elements.

```
suite = [suiteA suiteB suiteC];
{suite.Name}'
```

```
ans =

  3×1 cell array

    {'MyTestClassA/test_A'}
    {'MyTestClassB/test_B'}
    {'MyTestClassC/test_C'}
```

Sort the suite by shared fixtures and view the order of test elements.

```
sortedSuite = sortByFixtures(suite);
{sortedSuite.Name}'
```

```
ans =

  3×1 cell array

    {'MyTestClassA/test_A'}
    {'MyTestClassC/test_C'}
    {'MyTestClassB/test_B'}
```

Since the tests in `MyTestClassA` and `MyTestClassC` have the same shared test fixture, the test elements are reordered so that they are adjacent in the suite.

Tips

If you create a test suite using a single call to the `testsuite` function instead of several calls to a method of `matlab.unittest.TestSuite`, the suite is automatically sorted based on shared fixtures. However, if you add, remove, or reorder elements after initial suite creation, call the `sortByFixtures` method to sort the suite.

See Also

`matlab.unittest.fixtures`

Topics

"Write Tests Using Shared Fixtures"

Introduced in R2018b

texlabel

Format text with TeX characters

Syntax

```
txt = texlabel(f)
txt = texlabel(f,'literal')
```

Description

`txt = texlabel(f)` converts the MATLAB expression `f` into the TeX equivalent for use in text. `texlabel` converts Greek variable names (for example, lambda, delta, and so on) into a character vector that is displayed as Greek letters. The `txt` output is useful as an argument to annotation functions such as `title`, `xlabel`, and `text`.

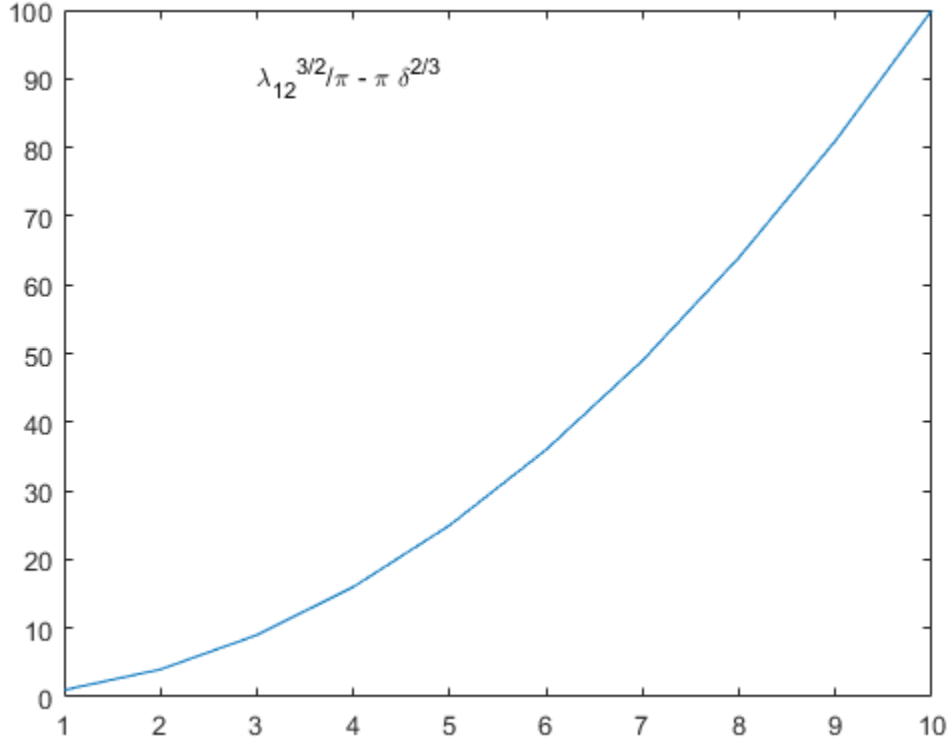
`txt = texlabel(f,'literal')` interprets Greek variable names literally.

Examples

Create Text with TeX Characters

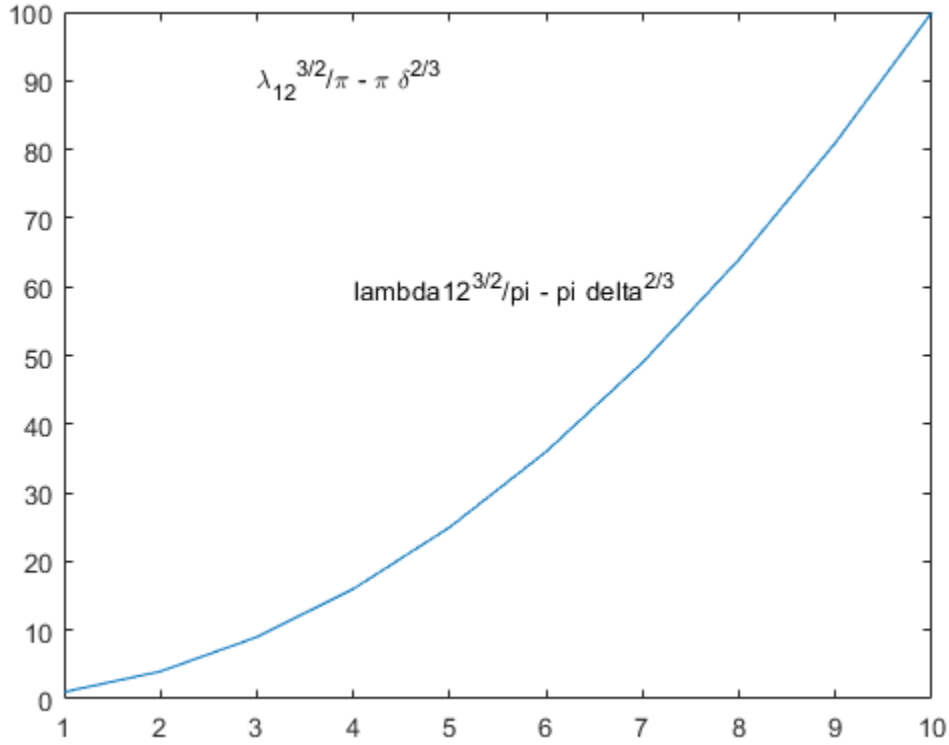
Plot a line. Create text that contains TeX characters and add the text to the line plot.

```
plot((1:10).^2)
txt = texlabel('lambda12^(3/2)/pi - pi*delta^(2/3)');
text(3,90,txt)
```



If you include the `'literal'` argument, `texlabel` interprets Greek variable names literally. Add text with the literal variable names to the line plot.

```
text(4,60,texlabel('lambda12^(3/2)/pi - pi*delta^(2/3)','literal'))
```



Input Arguments

f — Input MATLAB expression
character vector | string scalar

Input MATLAB expression, specified as a character vector or string scalar.

Example: `'theta (degrees)'`

See Also

`text` | `title` | `xlabel` | `ylabel` | `zlabel`

Introduced before R2006a

text

Add text descriptions to data points

Syntax

```
text(x,y,txt)
text(x,y,z,txt)
text( ___ ,Name,Value)
text(ax, ___ )

t = text( ___ )
```

Description

`text(x,y,txt)` adds a text description to one or more data points in the current axes using the text specified by `txt`. To add text to one point, specify `x` and `y` as scalars. To add text to multiple points, specify `x` and `y` as vectors with equal length.

`text(x,y,z,txt)` positions the text in 3-D coordinates.

`text( ___ ,Name,Value)` specifies `Text` object properties using one or more name-value pairs. For example, `'FontSize',14` sets the font size to 14 points. You can specify text properties with any of the input argument combinations in the previous syntaxes. If you specify the `Position` and `String` properties as name-value pairs, then you do not need to specify the `x`, `y`, `z`, and `txt` inputs.

`text(ax, ___ )` creates the text in the Cartesian, polar, or geographic axes specified by `ax` instead of in the current axes (`gca`). The option `ax` can precede any of the input argument combinations in the previous syntaxes.

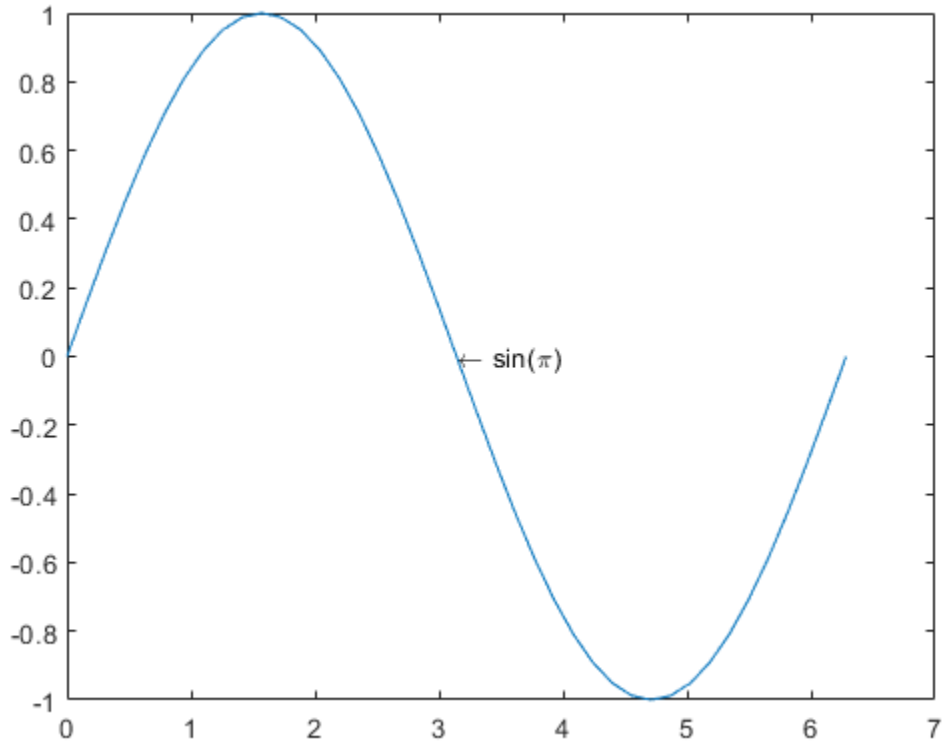
`t = text( ___ )` returns one or more text objects. Use `t` to modify properties of the text objects after they are created. For a list of properties and descriptions, see Text. You can specify an output with any of the previous syntaxes.

Examples

Add Text Description to Data Point

Plot a sine curve. At the point $(\pi, 0)$, add the text description $\sin(\pi)$. Use the TeX markup `\pi` for the Greek letter π. Use `\leftarrow` to display a left-pointing arrow.

```
x = 0:pi/20:2*pi;
y = sin(x);
plot(x,y)
text(pi,0,'\leftarrow sin(\pi)')
```

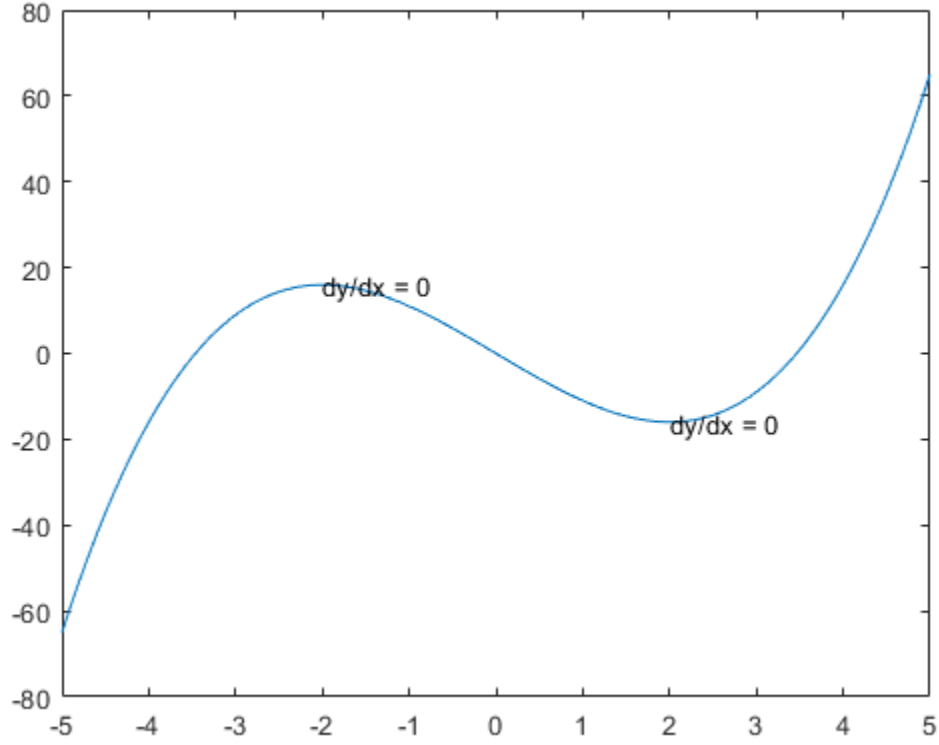


For a list of Greek characters and other TeX markup, see the `Interpreter` property description.

Add Text to Multiple Data Points

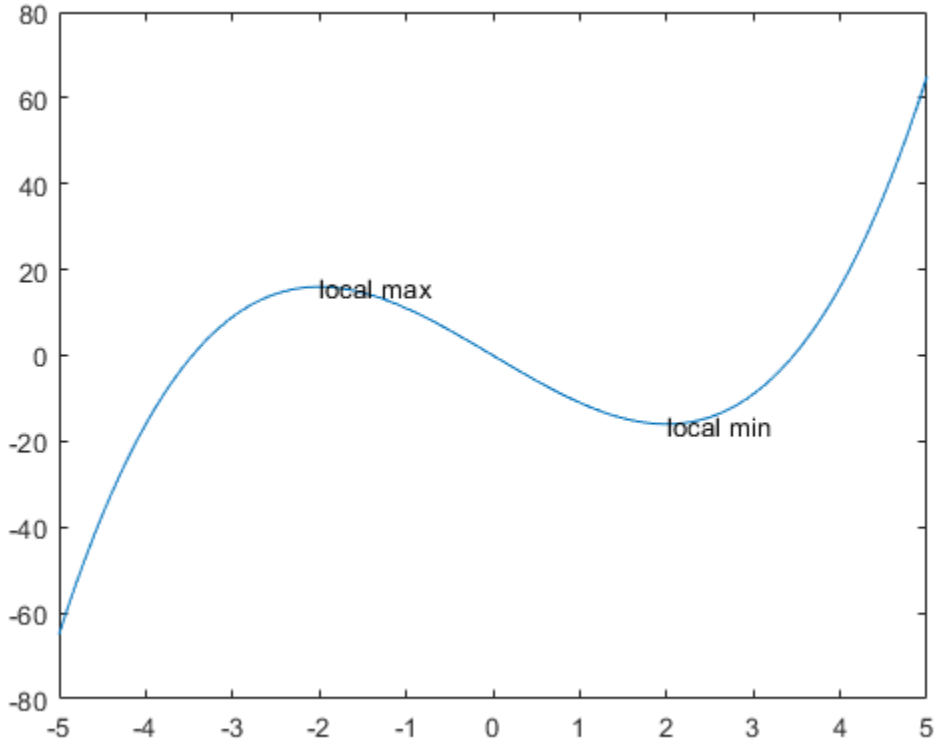
Plot a line. Add the same text to two points along the line.

```
x = linspace(-5,5);
y = x.^3-12*x;
plot(x,y)

xt = [-2 2];
yt = [16 -16];
str = 'dy/dx = 0';
text(xt,yt,str)
```



Add different text to each point by specifying `str` as a cell array.

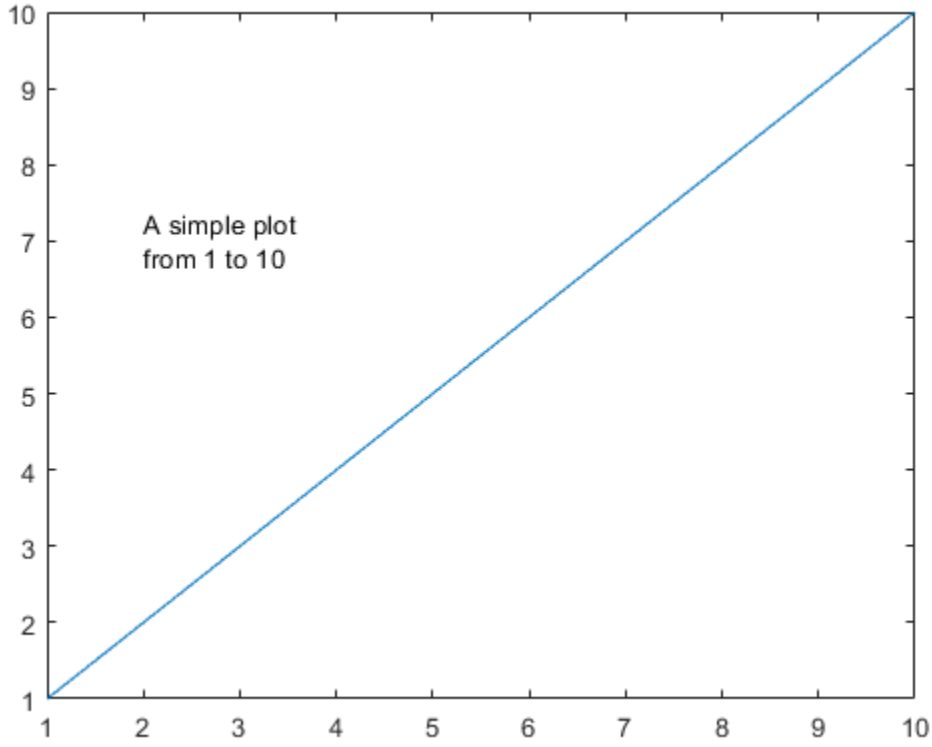
```
x = linspace(-5,5);
y = x.^3-12*x;
plot(x,y)

xt = [-2 2];
yt = [16 -16];
str = {'local max','local min'};
text(xt,yt,str)
```



Display Multiline Text

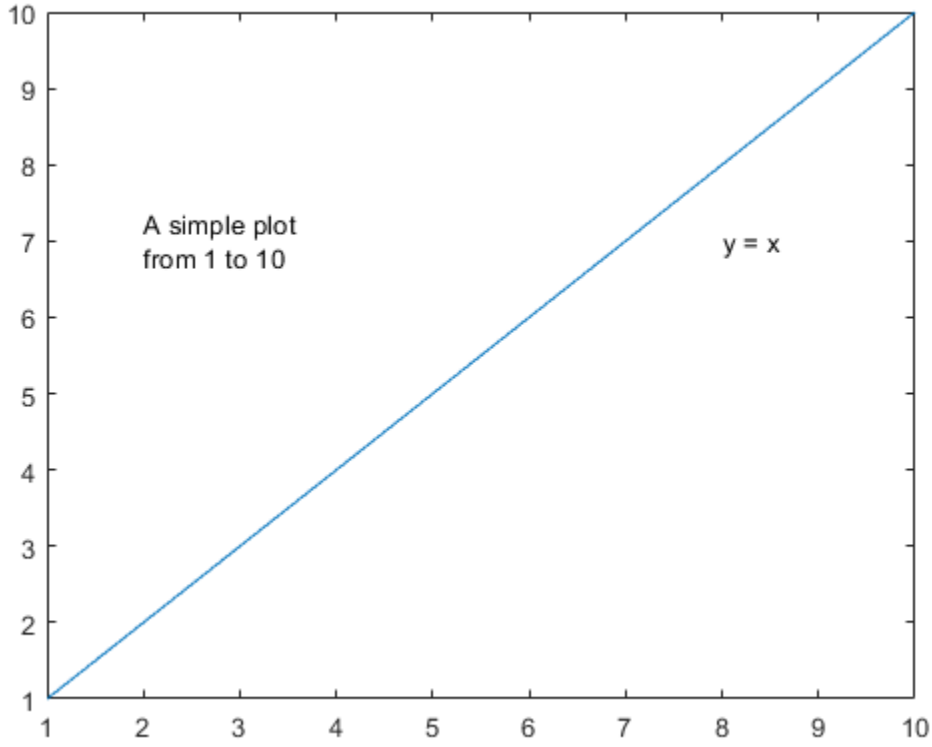
Create a line plot and add one text description to the axes. Display multiline text by specifying `str` as a cell array.

```
plot(1:10)
str = {'A simple plot','from 1 to 10'};
text(2,7,str)
```



Create a line plot and add two text descriptions to the axes. When adding multiple text descriptions to the axes, display multiline text by specifying nested cell arrays.

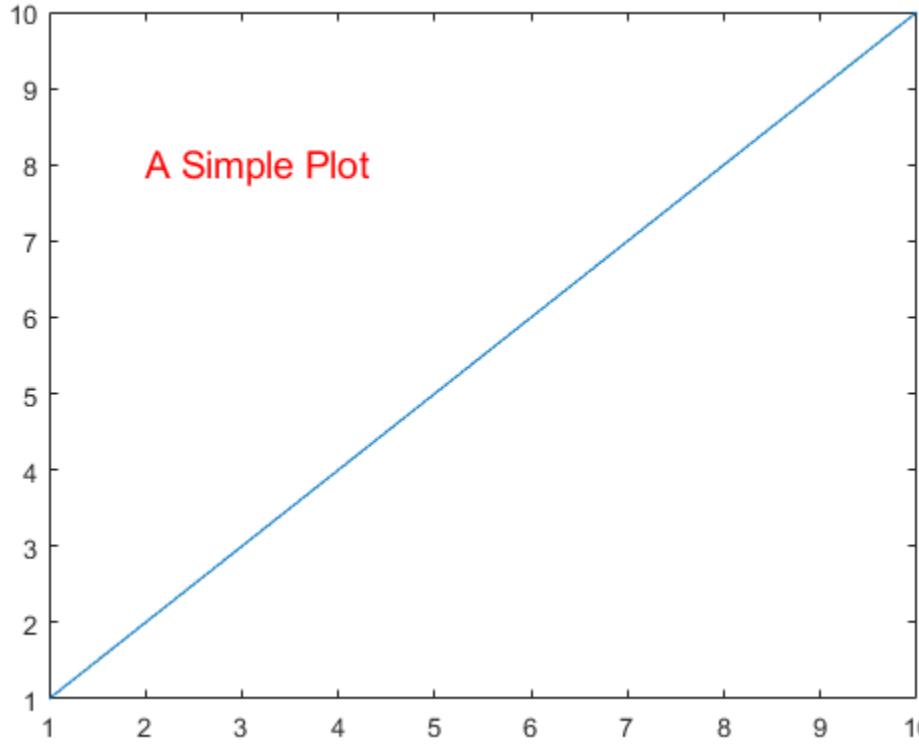
```
plot(1:10)
str = {{'A simple plot','from 1 to 10'},'y = x'};
text([2 8],[7 7],str)
```



Specify Text Size and Color

Create a line plot and add a text description to the axes. Use red, size 14 font.

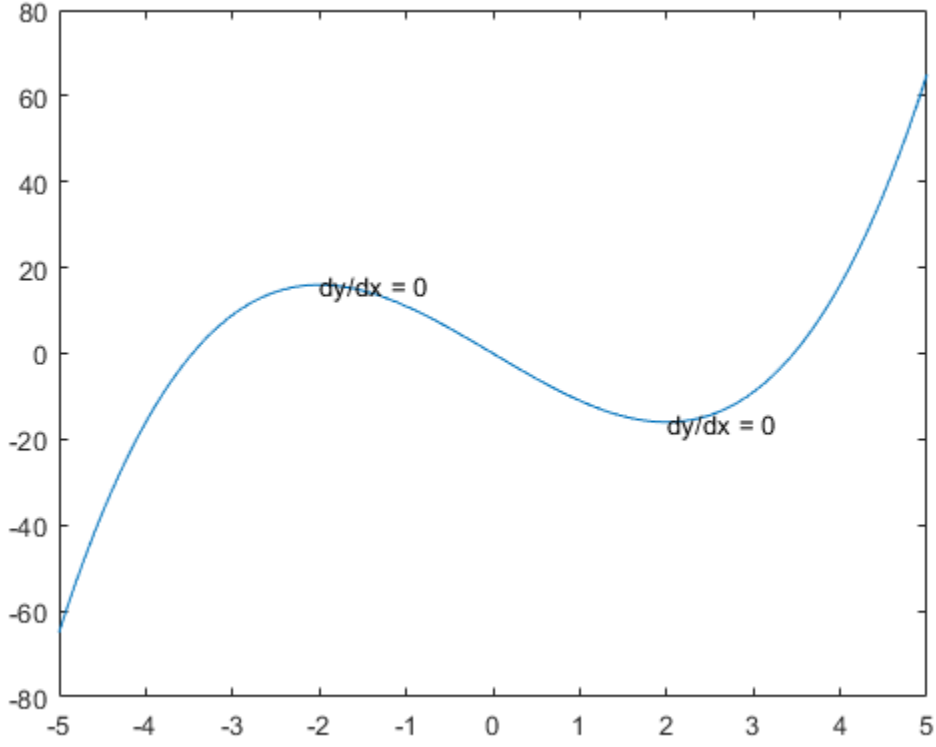
```
plot(1:10)
text(2,8,'A Simple Plot','Color','red','FontSize',14)
```



Modify Existing Text

Create a line plot and add two text descriptions along the line. Return the text objects, `t`.

```
x = linspace(-5,5);
y = x.^3-12*x;
plot(x,y)
t = text([-2 2],[16 -16],'dy/dx = 0')
```

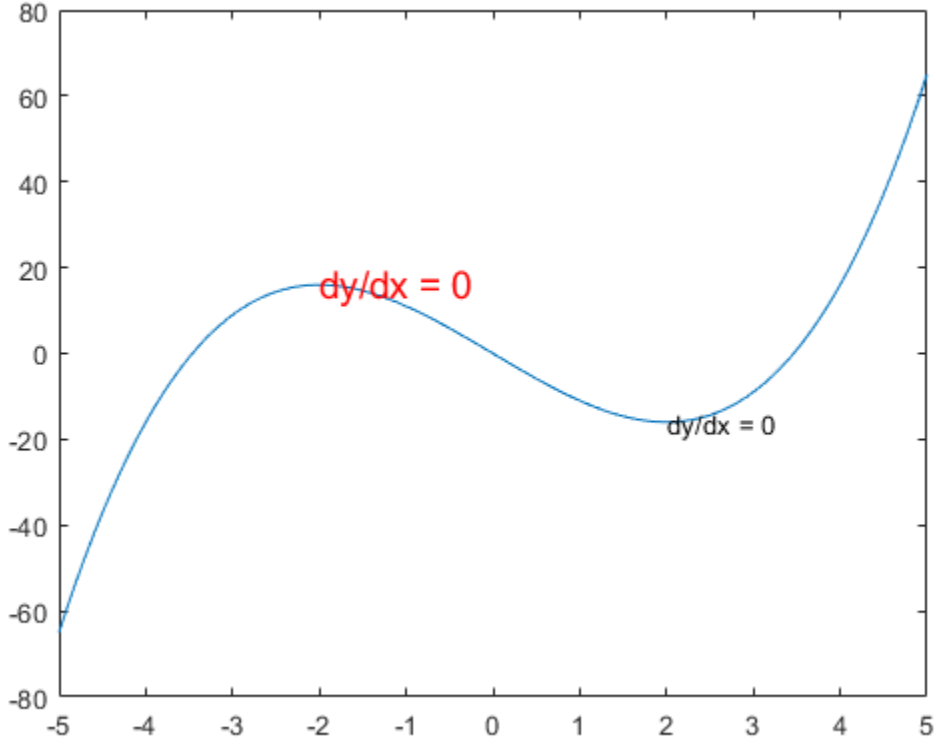



```
t =
  2x1 Text array:

  Text    (dy/dx = 0)
  Text    (dy/dx = 0)
```

The `text` function creates one text object for each text description. Thus, `t` contains two text objects.

Change the color and font size for the first text object using `t(1)`. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead.

```
t(1).Color = 'red';
t(1).FontSize = 14;
```



Input Arguments

x — First coordinate
scalar | vector

First coordinate, specified in one of these forms:

- Scalar — Add text to a single point.

- Vector — Add text to multiple points.

The interpretation of the first coordinate depends on the type of axes:

- For Cartesian axes, the first coordinate is *x*-axis position in data units. To change units, set the `Units` property for the `Text` object.
- For polar axes, the first coordinate is the polar angle θ in radians.
- For geographic axes, the first coordinate is latitude in degrees.

`x` and `y` must be equal sizes.

To control the placement of the text with relation to the specified point, use the `HorizontalAlignment`, `VerticalAlignment`, and `Extent` properties of the `Text` object.

Data Types: `double` | `categorical` | `datetime` | `duration`

y — Second coordinate
scalar | vector

Second coordinate, specified in one of these forms:

- Scalar — Add text to a single point.
- Vector — Add text to multiple points.

The interpretation of the second coordinate depends on the type of axes:

- For Cartesian axes, the second coordinate is *y*-axis position in data units. To change units, set the `Units` property for the `Text` object.
- For polar axes, the second coordinate is the radius in data units.
- For geographic axes, the second coordinate is longitude in degrees.

`x` and `y` must be equal sizes.

To control the placement of the text with relation to the specified point, use the `HorizontalAlignment`, `VerticalAlignment`, and `Extent` properties of the `Text` object.

Data Types: `double` | `categorical` | `datetime` | `duration`

z — Third coordinate
0 (default) | scalar | vector

Third coordinate, specified in one of these forms:

- Scalar — Add text to a single point.
- Vector — Add text to multiple points.

The interpretation of the third coordinate depends on the type of axes:

- For Cartesian axes, the third coordinate is *z*-axis position in data units. To change units, set the `Units` property for the `Text` object.
- For polar and geographic axes, the third coordinate affects the layering of text on the axes.

`x`, `y`, and `z` must be equal sizes.

To control the placement of the text with relation to the specified point, use the `HorizontalAlignment`, `VerticalAlignment`, and `Extent` properties of the `Text` object.

Data Types: `double` | `categorical` | `datetime` | `duration`

`txt` — Text to display

character vector | string scalar | character array | string array | cell array | categorical array

Text to display, specified as a character vector, string scalar, character array, string array, cell array, or categorical array.

Text for Single Data Point

To add text to a single point, specify `txt` as a character vector or string scalar. For example, `text(.5,.5,'my text')`.

For multiline text:

- Use a cell array, where each cell contains a line of text. For example, `text(.5,.5,{'first','second'})`.
- Use a string array, where each element is a line of text. For example, `text(.5,.5,["first","second"])`.
- Use a character array with multiple rows, where each row contains the same number of characters. For example, `text(.5,.5,['first '; 'second'])`.
- Use `sprintf` to create a new line character. For example, `text(.5,.5,sprintf('first \nsecond'))`.

If you specify the text as a categorical array, MATLAB uses the values in the array, not the categories.

Text for Multiple Data Points

To display the same text at each location, specify `txt` as a character vector or string. For example, `text([0 1],[0 1],'my text')`.

To display different text at each location, use a cell array. For example, `text([0 1],[0 1],{'first','second'})`.

For multiline text, use nested cell arrays. For example, `text([0 1],[0 1], {{'first','new line'},'second'})`.

Greek Letters, Special Characters, and Numeric Variables

To include special characters, such as superscripts, subscripts, Greek letters, or mathematical symbols, use TeX markup. For a list of supported markup, see the `Interpreter` property.

To include numeric variables in the text, use the `num2str` function.

```
v = 42;
txt = ['The value is ',num2str(v)];
```

ax — Target axes

`Axes` object | `PolarAxes` object | `GeographicAxes` object

Target axes, specified as an `Axes` object, a `PolarAxes` object, or a `GeographicAxes` object. If you do not specify the axes, then `text` uses the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `text(.5,.5,'my text','FontSize',14,'Color','red')`

The properties listed here are only a subset. For a full list, see Text.

FontSize — Font size

scalar value greater than zero

Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals `1/72` inch. To change the font units, use the `FontUnits` property.

Example: 12

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`FontWeight` — Character thickness
`'normal'` (default) | `'bold'`

Character thickness, specified as `'normal'` or `'bold'`.

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.

`FontName` — Font name
supported font name | `'FixedWidth'`

Font name, specified as a supported font name or `'FixedWidth'`. To display and print text properly, you must choose a font that your system supports. The default font depends on your operating system and locale.

To use a fixed-width font that looks good in any locale, use `'FixedWidth'`. The fixed-width font relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`Color` — Text color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Text color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to

F. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`HorizontalAlignment` — Horizontal alignment of text with respect to position point

`'left'` (default) | `'center'` | `'right'`

Horizontal alignment of the text with respect to the `x` value in the `Position` property, specified as one of the values in this table. The vertical line indicates where the `x` value lies in relation to the text.

Value	Result
`'left'` (default)	
`'center'`	
`'right'`	

`Position` — Location of text

`[0 0 0]` (default) | two-element vector of form `[x y]` | three-element vector of form `[x y z]`

Location of the text, specified as a two-element vector of the form `[x y]` or a three-element vector of the form `[x y z]`. If you omit the third element, `z`, then MATLAB sets it to `0`.

Specify the position using numeric values. To convert datetime or duration values to the appropriate numeric values for a particular coordinate direction, see `ruler2num`.

By default, the position value is defined in data units. To change the units, use the `Units` property.

Example: `[0.5 0.5 0]`

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Units` — Position and extent units

`'data'` (default) | `'normalized'` | `'inches'` | `'centimeters'` | `'characters'` | `'points'` | `'pixels'`

Position units, specified as one of the values in this table.

Units	Description
`'data'` (default)	Data coordinates.
`'normalized'`	Normalized with respect to the axes. The lower left corner of the axes maps to `(0,0)` and the upper right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the default system font character size. • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.
`'points'`	Points. One point equals 1/72 inch.
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.

All units, except for `'data'`, are measured from the lower left corner of the axes. This property affects the `Position` and `Extent` properties.

If you specify the `Position` and `Units` properties as `Name,Value` pairs when creating the `Text` object, then the order of specification matters. To define the position with particular units, set the `Units` property before the `Position` property.

`Interpreter` — Interpretation of text characters

`'tex'` (default) | `'latex'` | `'none'`

Interpretation of text characters, specified as one of these values:

- `'tex'` — Interpret characters using a subset of TeX markup.
- `'latex'` — Interpret characters using LaTeX markup.
- `'none'` — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text.

Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they modify only the next character or the characters within the curly braces. When you set the `Interpreter` property to `'tex'`, the supported modifiers are as follows.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{`*`specifier`*`}`	Font name — Replace *`specifier`* with the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{`*`specifier`*`}`	Font size —Replace *`specifier`* with a numeric scalar value in point units.	`'\fontsize{15} text'`

Modifier	Description	Example
`\color{specifier}`	Font color — Replace *specifier* with one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Replace *specifier* with a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	β	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ϵ	`\Delta`	Δ	`\spadesuit`	♠
`\zeta`	ζ	`\Theta`	Θ	`\leftrightarrow`	↔
`\eta`	η	`\Lambda`	Λ	`\leftarrow`	←
`\theta`	θ	`\Xi`	Ξ	`\Leftarrow`	⇐
`\vartheta`	ϑ	`\Pi`	Π	`\uparrow`	↑
`\iota`	ι	`\Sigma`	Σ	`\rightarrow`	→

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	µ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. Use dollar symbols around the text, for example, use `'$\int_1^{20} x^2 dx$'` for inline mode or `'$$ \int_1^{20} x^2 dx$$'` for display mode.

The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup.

The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters. For multiline text, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project website at https://www.latex-project.org/.

Output Arguments

`t` — `Text` objects
`Text` objects

`Text` objects. Use the elements of `t` to modify a specific `Text` object after it is created. For a list of properties, see Text.

Tips

- By default, the `Clipping` property for text objects is set to `'off'` so the text might appear outside the axes. To clip the text to the axes boundaries, set the property to `'on'`.

See Also

Functions
`annotation` | `int2str` | `num2str` | `title` | `xlabel` | `ylabel` | `zlabel`

Properties
Text

Topics
"Add Text to Chart"
"Greek Letters and Special Characters in Chart Text"
"Plot Dates and Durations"

Introduced before R2006a

Text Properties

Axes text appearance and behavior

Description

`Text` properties control the appearance and behavior of a `Text` object. By changing property values, you can modify certain aspects of the text.

Starting in R2014b, you can use dot notation to query and set properties.

```
t = text(0.5,0.5,'text here');
s = t.FontSize;
t.FontSize = 12;
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Text

`String` — Text to display

`''` (default) | character vector | cell array of character vectors | string array | categorical array | numeric value

Text to display, specified as a character vector, cell array of character vectors, string array, categorical array, or numeric value. Specify a value according to how you want the text to appear. This table lists the most common situations.

Appearance of Text	Description of Value	Example

One line of text	Character vector or 1-by-1 string array.	`str = 'My Text'`
		or
		`str = "My Text"`

Appearance of Text	Description of Value	Example

Multiple lines of text	Cell array of character vectors or a string array.	`str = {'First line','Second line'};`
		or
		`str = ["First line", "Second line"];`

Text that includes a numeric variable	Array that includes the number converted to a `char` array. Use `num2str` to convert the value.	`x = 42;` `str = ['The value is ',num2str(x)];`
Text that includes special characters such as Greek letters or mathematical symbols	Array that includes TeX markup. For a list of supported markup, see the `Interpreter` property.	`str = 'x ranges from 0 to 2\pi'`

Note

- The words `default`, `factory`, and `remove` are reserved words that do not appear in text when quoted as normal characters. To display any of these words individually, precede them with a backslash, such as `'\default'` or `'\remove'`.
- If you specify this property as a categorical array, MATLAB uses the values in the array, not the categories.
- If you specify text that contains only a numeric value, the value is converted using `sprintf('%g',value)`. For example, `12345678` displays as `1.23457e+07`.

`Color` — Text color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Text color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: 'blue'

Example: [0 0 1]

Example: '#0000FF'

Interpreter — Interpretation of text characters

'tex' (default) | 'latex' | 'none'

Interpretation of text characters, specified as one of these values:

- 'tex' — Interpret characters using a subset of TeX markup.
- 'latex' — Interpret characters using LaTeX markup.
- 'none' — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text.

Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they modify only the next character or the characters within the curly braces. When you set the Interpreter property to 'tex', the supported modifiers are as follows.

Modifier	Description	Example
^{ }	Superscript	'text^{superscript}'
_{ }	Subscript	'text_{subscript}'
\bf	Bold font	'\bf text'
\it	Italic font	'\it text'
\sl	Oblique font (usually the same as italic font)	'\sl text'
\rm	Normal font	'\rm text'

Modifier	Description	Example
`\fontname{`*`specifier`*`}`	Font name — Replace *`specifier`* with the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{`*`specifier`*`}`	Font size —Replace *`specifier`* with a numeric scalar value in point units.	`'\fontsize{15} text'`
`\color{`*`specifier`*`}`	Font color — Replace *`specifier`* with one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Replace *`specifier`* with a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	β	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ϵ	`\Delta`	Δ	`\spadesuit`	♠
`\zeta`	ζ	`\Theta`	Θ	`\leftrightarrow`	↔

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	µ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. Use dollar symbols around the text, for example, use `'$\int_1^{20} x^2 dx$'` for inline mode or `'$$ \int_1^{20} x^2 dx$$'` for display mode.

The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup.

The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters. For multiline text, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project website at https://www.latex-project.org/.

Font

`FontName` — Font name

supported font name | `'FixedWidth'`

Font name, specified as a supported font name or `'FixedWidth'`. To display and print text properly, you must choose a font that your system supports. The default font depends on your operating system and locale.

To use a fixed-width font that looks good in any locale, use `'FixedWidth'`. The fixed-width font relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`FontSize` — Font size

scalar value greater than zero

Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals `1/72` inch. To change the font units, use the `FontUnits` property.

Example: 12

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`FontWeight` — Character thickness

`'normal'` (default) | `'bold'`

Character thickness, specified as `'normal'` or `'bold'`.

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.

`FontAngle` — Character slant
`'normal'` (default) | `'italic'`

Character slant, specified as `'normal'` or `'italic'`.

Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.

`FontUnits` — Font size units
`'points'` (default) | `'inches'` | `'centimeters'` | `'normalized'` | `'pixels'`

Font size units, specified as one of the values in this table.

Units	Description
`'points'`	Points. One point equals 1/72 inch.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'normalized'`	Interpret font size as a fraction of the axes plot box height. If you resize the axes, the font size modifies accordingly. For example, if the `FontSize` is `0.1` in normalized units, then the text is 1/10 of the plot box height.

Units	Description
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.

If you set both the font size and the font units in one function call, you must set the `FontUnits` property first so that the axes correctly interprets the specified font size.

`FontSmoothing` — Smooth font character appearance
`'on'` (default) | `'off'`

Smooth font character appearance, specified as one of these values:

- `'on'` — Apply font smoothing. Reduce the appearance of jaggedness in the text characters to make the text easier to read.
- `'off'` — Do not apply font smoothing.

Text Box

`Rotation` — Text orientation
`0` (default) | scalar value in degrees

Text orientation, specified as a scalar value in degrees. The default rotation of 0 degrees makes the text horizontal. For vertical text, set this property to `90` or `-90`. Positive values rotate the text counterclockwise. Negative values rotate the text clockwise.

Example: `90`

Example: `-90`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

EdgeColor — Color of box outline

`'none'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of box outline, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: 'blue'

Example: [0 0 1]

Example: '#0000FF'

BackgroundColor — Color of text box background
'none' (default) | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...

Color of text box background, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`LineStyle` — Line style of box outline

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style of box outline, specified as one of the options in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	Line is invisible	

`LineWidth` — Width of box outline
`0.5` (default) | scalar numeric value

Width of box outline, specified as a scalar numeric value in point units. One point equals 1/72 inch.

Example: `1.5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Margin` — Space around text within the text box
3 (default) | scalar numeric value

The space around the text within the text box, specified as scalar numeric value in point units.

MATLAB uses the `Extent` property value plus the `Margin` property value to determine the size of the text box.

Example: 8

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Clipping` — Clipping to axes plot box
`'off'` (default) | `'on'`

Clipping of the text to the axes plot box, which is the box defined by the axis limits, specified as one of these values:

- `'off'` — Do not clip the text. Portions of it might appear outside the axes plot box.
- `'on'` — Clips the text to the axes plot box.
  - If the axes `ClippingStyle` property is set to `'3dbox'`, which is the default, then MATLAB either displays the entire text or none of the text, depending on the text position. If the point defined by the text `Position` property lies inside the axes, then MATLAB displays the entire text. If the point lies outside the axes, then MATLAB displays none of it.
  - If the axes `ClippingStyle` property is set to `'rectangle'`, then MATLAB displays portions of the text lying inside the axes plot box and does not display portions of the text lying outside the axes plot box.

Note If the `Clipping` property of the associated axes is set to `'on'`, which is the default, then each individual object controls its own clipping behavior. If the `Clipping` property of the axes is set to `'off'`, then MATLAB does not clip any objects in the axes, regardless of the `Clipping` property of the individual object.

Position

`Position` — Location of text

`[0 0 0]` (default) | two-element vector of form `[x y]` | three-element vector of form `[x y z]`

Location of the text, specified as a two-element vector of the form `[x y]` or a three-element vector of the form `[x y z]`. If you omit the third element, `z`, then MATLAB sets it to `0`.

Specify the position using numeric values. To convert datetime or duration values to the appropriate numeric values for a particular coordinate direction, see `ruler2num`.

By default, the position value is defined in data units. To change the units, use the `Units` property.

Example: `[0.5 0.5 0]`

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Extent` — Size and location of rectangle that encloses text

four-element vector

This property is read-only.

Size and location of the rectangle that encloses the text, not including the margin, returned as a four-element vector of the form `[left bottom width height]`. The first two elements, `left` and `bottom`, define the position of the lower left corner of the rectangle. The last two elements, `width` and `height`, define the dimensions of the rectangle.

By default, the extent value is defined in data units. To change the units, use the `Units` property.

Example: `[0.5 0.5 0.4 0.2]`

`Units` — Position and extent units
`'data'` (default) | `'normalized'` | `'inches'` | `'centimeters'` | `'characters'` | `'points'` | `'pixels'`

Position units, specified as one of the values in this table.

Units	Description
`'data'` (default)	Data coordinates.
`'normalized'`	Normalized with respect to the axes. The lower left corner of the axes maps to `(0,0)` and the upper right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'characters'`	Based on the default system font character size. • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.
`'points'`	Points. One point equals 1/72 inch.

Units	Description
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.

All units, except for `'data'`, are measured from the lower left corner of the axes. This property affects the `Position` and `Extent` properties.

If you specify the `Position` and `Units` properties as `Name,Value` pairs when creating the `Text` object, then the order of specification matters. To define the position with particular units, set the `Units` property before the `Position` property.

`HorizontalAlignment` — Horizontal alignment of text with respect to position point

`'left'` (default) | `'center'` | `'right'`

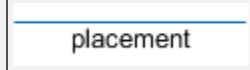
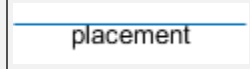
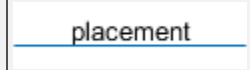
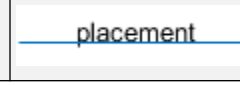
Horizontal alignment of the text with respect to the `x` value in the `Position` property, specified as one of the values in this table. The vertical line indicates where the `x` value lies in relation to the text.

Value	Result
`'left'` (default)	
`'center'`	

Value	Result
`'right'`	

`VerticalAlignment` — Vertical alignment of text with respect to position point
`'middle'` (default) | `'top'` | `'bottom'` | `'baseline'` | `'cap'`

Vertical alignment of the text with respect to the `y` value in the `Position` property, specified as one of the values in this table. The horizontal line indicates where the `y` value lies in relation to the text.

Value	Result
`'middle'`	
`'top'`	
`'cap'`	
`'bottom'`	
`'baseline'`	

Interactivity

`Editing` — Interactive edit mode
`'off'` (default) | `'on'`

Interactive edit mode, specified as one of these values:

- `'off'` — Do no allow interactive text editing. To change the text, you must set the `String` property. This is the default value.
- `'on'` — Allow interactive text editing. MATLAB places an insert cursor within the text and typing changes the text. To apply the new text, do any of the following:
  - Press the **Esc** key.

- Click anywhere away from the text.
- Reset the `Editing` property to `'off'`.

MATLAB updates the `String` property to contain the new text and resets the `Editing` property to `'off'`.

`Visible` — State of visibility
`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`UIContextMenu` — Context menu
empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state
`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles
`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.

- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks only when visible. The `Visible` property must be set to `'on'`. The `HitTest` property determines if the `Text` object responds to the click or if an ancestor does.
- `'all'` — Capture mouse clicks regardless of visibility. The `Visible` property can be set to `'on'` or `'off'`. The `HitTest` property determines if the `Text` object responds to the click or if an ancestor does.

- `'none'` — Cannot capture mouse clicks. Clicking the `Text` object passes the click to the object below it in the current view of the figure window, which is typically the axes or the figure. The `HitTest` property has no effect.

If you want an object to be clickable when it is underneath other objects that you do not want to be clickable, then set the `PickableParts` property of the other objects to `'none'` so that the click passes through them.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `Text` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `Text` object that has one of these:
  - `HitTest` property set to `'on'`
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

---

Note The `PickableParts` property determines if the `Text` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent
`Axes` object | `PolarAxes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `PolarAxes`, `Group`, or `Transform` object.

`Children` — Children

empty `GraphicsPlaceholder` array

The object has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'text'`

This property is read-only.

Type of graphics object, returned as `'text'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
[ ] (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`text`

Topics

“Access Property Values”
“Graphics Object Properties”

Introduced before R2006a

Text Properties

Tiled layout text appearance and behavior

Description

`Text` properties control the appearance and behavior of the `Text` objects used for shared titles and axis labels in tiled chart layouts. By changing property values, you can modify certain aspects of the text.

```
t = tiledlayout(2,2);
txt = title(t,'My Title');
t.FontSize = 16;
```

Properties

Text

`String` — Text to display
`''` (default) | character vector | cell array of character vectors | string array | categorical array | numeric value

Text to display, specified as a character vector, cell array of character vectors, string array, categorical array, or numeric value. Specify a value according to how you want the text to appear. This table lists the most common situations.

Appearance of Text	Description of Value	Example

One line of text	Character vector or 1-by-1 string array.	`str = 'My Text'`
		or
		`str = "My Text"`

Appearance of Text	Description of Value	Example

Multiple lines of text	Cell array of character vectors or a string array.	`str = {'First line','Second line'};`
		or
		`str = ["First line", "Second line"];`

Text that includes a numeric variable	Array that includes the number converted to a `char` array. Use `num2str` to convert the value.	`x = 42;` `str = ['The value is ',num2str(x)];`
Text that includes special characters such as Greek letters or mathematical symbols	Array that includes TeX markup. For a list of supported markup, see the `Interpreter` property.	`str = 'x ranges from 0 to 2\pi'`

Note

- The words `default`, `factory`, and `remove` are reserved words that do not appear in text when quoted as normal characters. To display any of these words individually, precede them with a backslash, such as `'\default'` or `'\remove'`.
- If you specify this property as a categorical array, MATLAB uses the values in the array, not the categories.
- If you specify text that contains only a numeric value, the value is converted using `sprintf('%g',value)`. For example, `12345678` displays as `1.23457e+07`.

`Color` — Text color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Text color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: 'blue'

Example: [0 0 1]

Example: '#0000FF'

Interpreter — Interpretation of text characters

'tex' (default) | 'latex' | 'none'

Interpretation of text characters, specified as one of these values:

- 'tex' — Interpret characters using a subset of TeX markup.
- 'latex' — Interpret characters using LaTeX markup.
- 'none' — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text.

Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they modify only the next character or the characters within the curly braces. When you set the Interpreter property to 'tex', the supported modifiers are as follows.

Modifier	Description	Example
^{ }	Superscript	'text^{superscript}'
_{ }	Subscript	'text_{subscript}'
\bf	Bold font	'\bf text'
\it	Italic font	'\it text'
\sl	Oblique font (usually the same as italic font)	'\sl text'
\rm	Normal font	'\rm text'

Modifier	Description	Example
`\fontname{`*`specifier`*`}`	Font name — Replace *`specifier`* with the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{`*`specifier`*`}`	Font size —Replace *`specifier`* with a numeric scalar value in point units.	`'\fontsize{15} text'`
`\color{`*`specifier`*`}`	Font color — Replace *`specifier`* with one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Replace *`specifier`* with a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	β	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ϵ	`\Delta`	Δ	`\spadesuit`	♠
`\zeta`	ζ	`\Theta`	Θ	`\leftrightarrow`	↔

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	µ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. Use dollar symbols around the text, for example, use `'$\int_1^{20} x^2 dx$'` for inline mode or `'$$ \int_1^{20} x^2 dx$$'` for display mode.

The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup.

The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters. For multiline text, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project website at https://www.latex-project.org/.

`HorizontalAlignment` — Horizontal alignment

`'center'` (default) | `'left'` | `'right'`

Horizontal alignment of the text, specified as `'center'`, `'left'`, or `'right'`.

`VerticalAlignment` — Vertical alignment

`'middle'` | `'top'` | `'bottom'` | `'baseline'` | `'cap'`

Vertical alignment of the text, specified as `'middle'`, `'top'`, `'bottom'`, `'baseline'`, or `'cap'`. The default value is `'bottom'` for shared titles and *y*-axis labels. For shared *x*-axis labels, the default value is `'top'`.

`Rotation` — Text orientation

`0` (default) | scalar value in degrees

Text orientation, specified as a scalar value in degrees. The default rotation of `0` degrees makes the text horizontal. For vertical text, set this property to `90` or `-90`. Positive values rotate the text counterclockwise. Negative values rotate the text clockwise.

Font

`FontName` — Font name

supported font name | `'FixedWidth'`

Font name, specified as a supported font name or `'FixedWidth'`. To display and print text properly, you must choose a font that your system supports. The default font depends on your operating system and locale.

To use a fixed-width font that looks good in any locale, use `'FixedWidth'`. The fixed-width font relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`FontSize` — Font size
scalar value greater than zero

Font size, specified as a scalar value greater than zero in point units. One point equals `1/72` inch. The default font size for a shared title is `13`. For a shared x-axis or y-axis label, the default font size is `12`.

`FontWeight` — Character thickness
`'normal'` (default) | `'bold'`

Character thickness, specified as `'normal'` or `'bold'`.

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.

`FontAngle` — Character slant
`'normal'` (default) | `'italic'`

Character slant, specified as `'normal'` or `'italic'`.

Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.

Interactivity

`Visible` — State of visibility
`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

Parent/Child

`Parent` — Parent
`TiledChartLayout` object

Parent, specified as a `TiledChartLayout` object.

`Children` — Children
empty `GraphicsPlaceholder` array

The object has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

See Also

TiledChartLayout Properties | `tiledlayout`

Introduced in R2019b

Text Properties

Subplot text appearance and behavior

Description

`Text` properties control the appearance and behavior of the `Text` object used to title a grid of subplots. By changing property values, you can modify certain aspects of the text. For example:

```
t = sgtitle('Title Text')
t.FontSize = 14;
```

Properties

Text

`String` — Text to display
`''` (default) | character vector | cell array of character vectors | string array | categorical array | numeric value

Text to display, specified as a character vector, cell array of character vectors, string array, categorical array, or numeric value.

Example: `'my label'`

Example: `{'first line','second line'}`

Example: `string('my label')`

Example: `123`

To include numeric variables with text, use the `num2str` function. For example:

```
x = 42;
txt = ['The value is ',num2str(x)];
```

To include special characters, such as superscripts, subscripts, Greek letters, or mathematical symbols use TeX markup. For a list of supported markup, see the `Interpreter` property.

To create multiline text:

- Use a cell array, where each cell contains a line of text, such as `{'first line','second line'}`.
- Use a string array, where each element contains a line of text, such as `string({'line one','line two'})`.
- Use a character array, where each row contains the same number of characters, such as `['abc'; 'ab ']`.
- Use `sprintf` to create text with a new line character, such as `sprintf('first line \n second line')`. This property converts text with new line characters to cell arrays.

Text that contains only a numeric value is converted using `sprintf('%g',value)`. For example, `12345678` displays as `1.23457e+07`.

If you specify this property as a categorical array, MATLAB uses the values in the array, not the categories.

`Color` — Text color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Text color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`Interpreter` — Interpretation of text characters
`'tex'` (default) | `'latex'` | `'none'`

Interpretation of text characters, specified as one of these values:

- `'tex'` — Interpret characters using a subset of TeX markup.
- `'latex'` — Interpret characters using LaTeX markup.
- `'none'` — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text.

Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they modify only the next character or the characters within the curly braces. When you set the `Interpreter` property to `'tex'`, the supported modifiers are as follows.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{`*`specifier`*`}`	Font name — Replace *`specifier`* with the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{`*`specifier`*`}`	Font size —Replace *`specifier`* with a numeric scalar value in point units.	`'\fontsize{15} text'`

Modifier	Description	Example
`\color{specifier}`	Font color — Replace *specifier* with one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Replace *specifier* with a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	β	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ϵ	`\Delta`	Δ	`\spadesuit`	♠
`\zeta`	ζ	`\Theta`	Θ	`\leftrightarrow`	↔
`\eta`	η	`\Lambda`	Λ	`\leftarrow`	←
`\theta`	θ	`\Xi`	Ξ	`\Leftarrow`	⇐
`\vartheta`	ϑ	`\Pi`	Π	`\uparrow`	↑
`\iota`	ι	`\Sigma`	Σ	`\rightarrow`	→

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	µ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. Use dollar symbols around the text, for example, use `'$\int_1^{20} x^2 dx$'` for inline mode or `'$$ \int_1^{20} x^2 dx$$'` for display mode.

The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup.

The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters. For multiline text, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project website at https://www.latex-project.org/.

`HorizontalAlignment` — Horizontal alignment

`'center'` (default) | `'left'` | `'right'`

Horizontal alignment of text within the text box, specified as `'center'`, `'left'`, or `'right'`.

Font

`FontName` — Font name

supported font name | `'FixedWidth'`

Font name, specified as a supported font name or `'FixedWidth'`. To display and print text properly, you must choose a font that your system supports. The default font depends on your operating system and locale.

To use a fixed-width font that looks good in any locale, use `'FixedWidth'`. The fixed-width font relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`FontSize` — Font size

scalar value greater than zero

Font size, specified as a scalar value greater than zero in point units. The default font size depends on the specific operating system and locale. One point equals `1/72` inch. To change the font units, use the `FontUnits` property.

Example: 12

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`FontSizeMode` — Selection mode for font size

`'auto'` (default) | `'manual'`

Selection mode for the font size, specified as either `'auto'` or `'manual'`.

`FontWeight` — Character thickness
`'normal'` (default) | `'bold'`

Character thickness, specified as `'normal'` or `'bold'`.

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.

`FontAngle` — Character slant
`'normal'` (default) | `'italic'`

Character slant, specified as `'normal'` or `'italic'`.

Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.

`FontUnits` — Font size units
`'points'` (default) | `'inches'` | `'centimeters'` | `'normalized'` | `'pixels'`

Font size units, specified as one of the values in this table.

Units	Description
`'points'`	Points. One point equals 1/72 inch.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'normalized'`	Interpret font size as a fraction of the axes plot box height. If you resize the axes, the font size modifies accordingly. For example, if the `FontSize` is `0.1` in normalized units, then the text is 1/10 of the plot box height.

Units	Description
'pixels'	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.

If you set both the font size and the font units in one function call, you must set the `FontUnits` property first so that the axes correctly interprets the specified font size.

Text Box

`EdgeColor` — Color of box outline

`'none'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of box outline, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`BackgroundColor` — Color of text box background

`'none'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of text box background, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: 'blue'

Example: [0 0 1]

Example: '#0000FF'

LineStyle — Line style of box outline
'-' (default) | '--' | ':' | '-.' | 'none'

Line style of box outline, specified as one of the options in this table.

Line Style	Description	Resulting Line
'-'	Solid line	
'--'	Dashed line	
':'	Dotted line	
'-.'	Dash-dotted line	
'none'	Line is invisible	

LineWidth — Width of box outline
0.5 (default) | scalar numeric value

Width of box outline, specified as a scalar numeric value in point units. One point equals 1/72 inch.

Example: 1.5

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Margin — Space around text within the text box
3 (default) | scalar numeric value

The space around the text within the text box, specified as scalar numeric value in point units.

MATLAB uses the `Extent` property value plus the `Margin` property value to determine the size of the text box.

Example: 8

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Interactivity

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.

- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles

`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

Callbacks

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.

- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Ability to capture mouse clicks
`'visible'` (default) | `'all'` | `'none'`

Ability to capture mouse clicks, specified as one of these values:

- `'visible'` — Capture mouse clicks only when visible. The `Visible` property must be set to `'on'`. The `HitTest` property determines if the `Text` object responds to the click or if an ancestor does.
- `'all'` — Capture mouse clicks regardless of visibility. The `Visible` property can be set to `'on'` or `'off'`. The `HitTest` property determines if the `Text` object responds to the click or if an ancestor does.

- `'none'` — Cannot capture mouse clicks. Clicking the `Text` object passes the click to the object below it in the current view of the figure window, which is typically the axes or the figure. The `HitTest` property has no effect.

If you want an object to be clickable when it is underneath other objects that you do not want to be clickable, then set the `PickableParts` property of the other objects to `'none'` so that the click passes through them.

`HitTest` — Response to captured mouse clicks
`'on'` (default) | `'off'`

Response to captured mouse clicks, specified as one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `Text` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Trigger the callbacks for the nearest ancestor of the `Text` object that has one of these:
  - `HitTest` property set to `'on'`
  - `PickableParts` property set to a value that enables the ancestor to capture mouse clicks

---

Note The `PickableParts` property determines if the `Text` object can capture mouse clicks. If it cannot, then the `HitTest` property has no effect.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent
`Figure` object

Parent, specified as a `Figure` object.

`Children` — Children
empty `GraphicsPlaceholder` array

The object has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object
`'subplottext'`

This property is read-only.

Type of graphics object, returned as `'subplottext'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Introduced in R2018b

textread

Read data from text file; write to multiple outputs

Note `textread` is not recommended. Use `textscan` instead.

Syntax

```
[A,B,C,...] = textread(filename,format)
[A,B,C,...] = textread(filename,format,N)
[...] = textread(...,param,value,...)
```

Description

`[A,B,C,...] = textread(filename,format)` reads data from the file `filename` into the variables `A`, `B`, `C`, and so on, using the specified `format`, until the entire file is read. Specify `filename` and `format` inputs as character vectors or string scalars. `textread` is useful for reading text files with a known format. `textread` handles both fixed and free format files.

Note When reading large text files, reading from a specific point in a file, or reading file data into a cell array rather than multiple outputs, you might prefer to use the `textscan` function.

`textread` matches and converts groups of characters from the input. Each input field is defined as a group of non-white-space characters that extends to the next white-space or delimiter character, or to the maximum field width. Repeated delimiter characters are significant, while repeated white-space characters are treated as one.

The `format` input, specified as a character vector or a string scalar, determines the number and types of return arguments. The number of return arguments is the number of items indicated by the contents of `format`. `format` supports a subset of the conversion

specifiers and conventions of the C language `fscanf` routine. Values for `format` are listed in the table below. White-space characters in `format` are ignored.

format	Action	Output
Literals (ordinary characters)	Ignore the matching characters. For example, in a file that has `Dept` followed by a number (for department number), to skip the `Dept` and read only the number, use `'Dept'` in the format specifier `format`.	None
`%d`	Read a signed integer value.	Double array
`%u`	Read an integer value.	Double array
`%f`	Read a floating-point value.	Double array
`%s`	Read a white-space or delimiter-separated text.	Cell array of character vectors
`%q`	Read double quoted text, ignoring the quotes.	Cell array of character vectors
`%c`	Read characters, including white space.	Character array
`%[...]`	Read the longest group of characters containing characters specified in the brackets.	Cell array of character vectors
`%[^...]`	Read the longest nonempty group of characters containing characters that are not specified in the brackets.	Cell array of character vectors
`%*...` `instead of %`	Ignore the matching characters specified by `*`.	No output
`%w...` `instead of %`	Read field width specified by `w`. The `%f` format supports `%w.pf`, where `w` is the field width and `p` is the precision.	

`[A,B,C,...] = textread(filename,format,N)` reads the data, reusing the format specified in `format`, `N` times, where `N` is an integer greater than zero. If `N` is smaller than zero, `textread` reads the entire file.

`[...] = textread(...,param,value,...)` customizes `textread` using `param/value` pairs, as listed in the table below.

param	value	Action
`bufsize`	Positive integer	Specifies the maximum length of the character vector, in bytes. Default is `4095`.

param	value	Action
`commentstyle`	`matlab`	Ignores characters after `%`.
`commentstyle`	`shell`	Ignores characters after `#`.
`commentstyle`	`c`	Ignores characters between `/*` and `*/`.
`commentstyle`	`c++`	Ignores characters after `//`.
`delimiter`	One or more characters	Act as delimiters between elements. Default is none.
`emptyvalue`	Scalar double	Value given to empty cells when reading delimited files. Default is 0.
`endofline`	Single character or `'\r\n'`	Character that denotes the end of a line. Default is determined from file
`expchars`	Exponent characters	Default is `eEdD`.
`headerlines`	Positive integer	Ignores the specified number of lines at the beginning of the file.
`whitespace`	Any from the list below:	Treats vector of characters as white space. Default is `' \b\t'`.
	`' '` Space `\b` Backspace `\n` Newline `\r` Carriage return `\t` Horizontal tab	

Note When `textread` reads a consecutive series of `whitespace` values, it treats them as one white space. When it reads a consecutive series of `delimiter` values, it treats each as a separate delimiter.

Examples

Example 1 — Read All Fields in Free Format File Using %

The first line of `mydata.dat` is

```
Sally    Level1 12.34 45 Yes
```

Read the first line of the file as a free format file using the `%` format.

```
[names, types, x, y, answer] = textread('mydata.dat', ...
'%s %s %f %d %s', 1)
```

returns

```
names =
    'Sally'
types =
    'Level1'
x =
  12.34000000000000
y =
    45
answer =
    'Yes'
```

Example 2 — Read as Fixed Format File, Ignoring the Floating Point Value

The first line of `mydata.dat` is

```
Sally    Level1 12.34 45 Yes
```

Read the first line of the file as a fixed format file, ignoring the floating-point value.

```
[names, types, y, answer] = textread('mydata.dat', ...
'%9c %6s %*f %2d %3s', 1)
```

returns

```
names =
Sally
types =
    'Level1'
y =
    45
answer =
    'Yes'
```

`%*f` in `format` causes `textread` to ignore the floating point value, in this case, `12.34`.

Example 3 — Read Using Literal to Ignore Matching Characters

The first line of `mydata.dat` is

```
Sally    Type1 12.34 45 Yes
```

Read the first line of the file, ignoring the characters `Type` in the second field.

```
[names, typenum, x, y, answer] = textread('mydata.dat', ...
'%s Type%d %f %d %s', 1)
```

returns

```
names =
    'Sally'
typenum =
    1
x =
  12.34000000000000
y =
    45
answer =
    'Yes'
```

Specifying `Type%d` in `format` causes the characters `Type` in the second field to be ignored, while the rest of the second field is read as a signed integer, in this case, `1`.

Example 4 — Specify Value to Fill Empty Cells

For files with empty cells, use the `emptyvalue` parameter. Suppose the file `data.csv` contains:

```
1,2,3,4,,6
7,8,9,,11,12
```

Read the file using `NaN` to fill any empty cells:

```
data = textread('data.csv', '', 'delimiter', ',', ...
                'emptyvalue', NaN);
```

Example 5 — Read File into a Cell Array of Character Vectors

Read the file `fft.m` into a cell array of character vectors.

```
file = textread('fft.m', '%s', 'delimiter', '\n', ...
                'whitespace', '');
```

Tips

If you want to preserve leading and trailing spaces in the text, use the `whitespace` parameter as shown here:

```
textread('myfile.txt', '%s', 'whitespace', '')
ans =
    '   An  example      of preserving    spaces      '
```

See Also

`fscanf` | `readmatrix` | `textscan`

Introduced before R2006a

textscan

Read formatted data from text file or string

Syntax

```
C = textscan(fileID,formatSpec)
C = textscan(fileID,formatSpec,N)

C = textscan(chr,formatSpec)
C = textscan(chr,formatSpec,N)

C = textscan( ___ ,Name,Value)

[C,position] = textscan( ___ )
```

Description

`C = textscan(fileID,formatSpec)` reads data from an open text file into a cell array, `C`. The text file is indicated by the file identifier, `fileID`. Use `fopen` to open the file and obtain the `fileID` value. When you finish reading from a file, close the file by calling `fclose(fileID)`.

`textscan` attempts to match the data in the file to the conversion specifier in `formatSpec`. The `textscan` function reapplies `formatSpec` throughout the entire file and stops when it cannot match `formatSpec` to the data.

`C = textscan(fileID,formatSpec,N)` reads file data using the `formatSpec` `N` times, where `N` is a positive integer. To read additional data from the file after `N` cycles, call `textscan` again using the original `fileID`. If you resume a text scan of a file by calling `textscan` with the same file identifier (`fileID`), then `textscan` automatically resumes reading at the point where it terminated the last read.

`C = textscan(chr,formatSpec)` reads the text from character vector `chr` into cell array `C`. When reading text from a character vector, repeated calls to `textscan` restart the scan from the beginning each time. To restart a scan from the last position, request a `position` output.

`textscan` attempts to match the data in character vector `chr` to the format specified in `formatSpec`.

`C = textscan(chr,formatSpec,N)` uses the `formatSpec` `N` times, where `N` is a positive integer.

`C = textscan( ___ ,Name,Value)` specifies options using one or more `Name,Value` pair arguments, in addition to any of the input arguments in the previous syntaxes.

`[C,position] = textscan( ___ )` returns the position in the file or the character vector at the end of the scan as the second output argument. For a file, this is the value that `ftell(fileID)` would return after calling `textscan`. For a character vector, `position` indicates how many characters `textscan` read.

Examples

Read Floating-Point Numbers

Read a character vector containing floating-point numbers.

```
chr = '0.41 8.24 3.57 6.24 9.27';
C = textscan(chr,'%f');
```

The specifier `'%f'` in `formatSpec` tells `textscan` to match each field in `chr` to a double-precision floating-point number.

Display the contents of cell array `C`.

```
celldisp(C)
```

```
C{1} =

    0.4100
    8.2400
    3.5700
    6.2400
    9.2700
```

Read the same character vector, and truncate each value to one decimal digit.

```
C = textscan(chr,'%3.1f %*1d');
```

The specifier `%3.1f` indicates a field width of 3 digits and a precision of 1. The `textscan` function reads a total of 3 digits, including the decimal point and the 1 digit after the decimal point. The specifier, `%*1d`, tells `textscan` to skip the remaining digit.

Display the contents of cell array `C`.

```
celldisp(C)
```

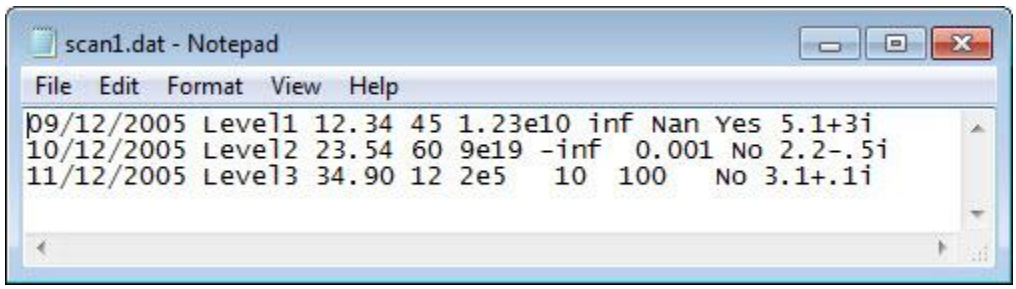
```
C{1} =

    0.4000
    8.2000
    3.5000
    6.2000
    9.2000
```

Read Different Types of Data

Load the data file and read each column with the appropriate type.

Load file `scan1.dat` and preview its contents in a text editor. A screen shot is shown below.

```
filename = fullfile(matlabroot,'examples','matlab','scan1.dat');
```



Open the file, and read each column with the appropriate conversion specifier. `textscan` returns a `1-by-9` cell array `C`.

```
fileID = fopen(filename);
C = textscan(fileID,'%s %s %f32 %d8 %u %f %f %s %f');
fclose(fileID);
whos C
```

```
  Name      Size            Bytes  Class    Attributes

  C         1x9              2249  cell
```

View the MATLAB® data type of each of the cells in `C`.

```
C
```

```
C=1×9 cell
  Columns 1 through 5

    {3x1 cell}    {3x1 cell}    {3x1 single}    {3x1 int8}    {3x1 uint32}

  Columns 6 through 9

    {3x1 double}    {3x1 double}    {3x1 cell}    {3x1 double}
```

Examine the individual entries. Notice that `C{1}` and `C{2}` are cell arrays. `C{5}` is of data type `uint32`, so the first two elements of `C{5}` are the maximum values for a 32-bit unsigned integer, or `intmax('uint32')`.

```
celldisp(C)
```

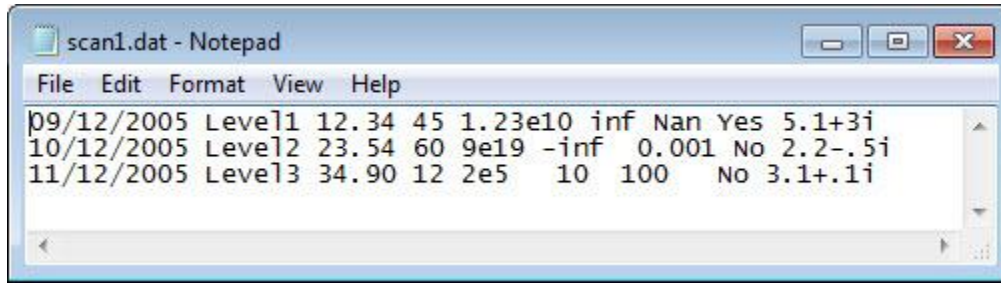
```
C{1}{1} =

09/12/2005


C{1}{2} =

10/12/2005


C{1}{3} =
```

```
11/12/2005


C{2}{1} =

Level1


C{2}{2} =

Level2


C{2}{3} =

Level3


C{3} =

   12.3400
   23.5400
   34.9000



C{4} =

   45
   60
   12



C{5} =

   4294967295
   4294967295
       200000



C{6} =
```

```
   Inf
  -Inf
    10



C{7} =

       NaN
    0.0010
  100.0000



C{8}{1} =

Yes


C{8}{2} =

No


C{8}{3} =

No


C{9} =

   5.1000 + 3.0000i
   2.2000 - 0.5000i
   3.1000 + 0.1000i
```

Remove Literal Text

Remove the literal text `'Level'` from each field in the second column of the data from the previous example. A preview of the file is shown below.



Open the file and match the literal text in the formatSpec input.

```
filename = fullfile(matlabroot,'examples','matlab','scan1.dat');
fileID = fopen(filename);
C = textscan(fileID,'%s Level%d %f32 %d8 %u %f %f %s %f');
fclose(fileID);
C{2}
```

```
ans = 3x1 int32 column vector

   1
   2
   3
```

View the MATLAB® data type of the second cell in `C`. The second cell of the `1-by-9` cell array, `C`, is now of data type `int32`.

```
disp( class(C{2}) )
```

```
int32
```

Skip the Remainder of a Line

Read the first column of the file in the previous example into a cell array, skipping the rest of the line.

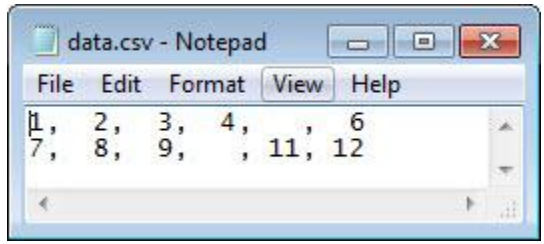
```
filename = fullfile(matlabroot,'examples','matlab','scan1.dat');
fileID = fopen(filename);
dates = textscan(fileID,'%s %*[^\n]');
fclose(fileID);
dates{1}
```

```
ans = 3x1 cell array
    {'09/12/2005'}
    {'10/12/2005'}
    {'11/12/2005'}
```

`textscan` returns a cell array dates.

Specify Delimiter and Empty Value Conversion

Load the file `data.csv` and preview its contents in a text editor. A screen shot is shown below. Notice the file contains data separated by commas and also contains empty values.



Read the file, converting empty cells to `-Inf`.

```
filename = fullfile(matlabroot,'examples','matlab','data.csv');
fileID = fopen(filename);
C = textscan(fileID,'%f %f %f %f %u8 %f',...
'Delimiter',',','EmptyValue',-Inf);
fclose(fileID);
column4 = C{4}, column5 = C{5}
```

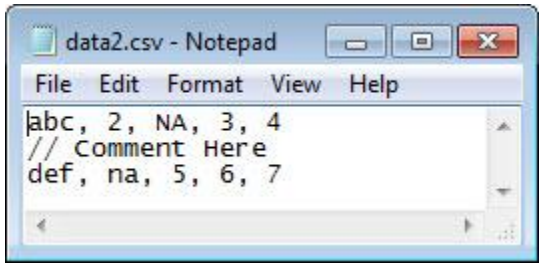
```
column4 = 2×1

     4
  -Inf


column5 = 2x1 uint8 column vector

    0
   11
```

`textscan` returns a `1-by-6` cell array, `C`. The `textscan` function converts the empty value in `C{4}` to `-Inf`, where `C{4}` is associated with a floating-point format. Because MATLAB® represents unsigned integer `-Inf` as `0`, `textscan` converts the empty value in `C{5}` to `0`, and not `-Inf`.

Specify Text to be Treated as Empty or Comments

Load the file `data2.csv` and preview its contents in a text editor. A screen shot is shown below. Notice the file contains data that can be interpreted as comments and other entries such as `'NA'` or `'na'` that may indicate empty fields.

```
filename = fullfile(matlabroot,'examples','matlab','data2.csv');
```



Designate the input that `textscan` should treat as comments or empty values and scan the data into `C`.

```
fileID = fopen(filename);
C = textscan(fileID,'%s %n %n %n %n','Delimiter',',',...
'TreatAsEmpty',{'NA','na'},'CommentStyle','//');
fclose(fileID);
```

Display the output.

```
celldisp(C)
```

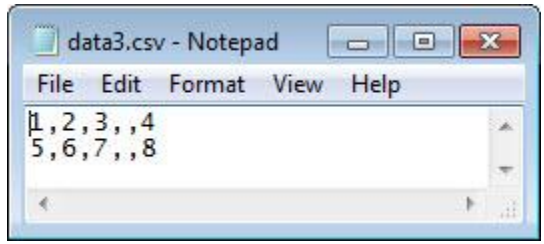
```
C{1}{1} =

abc
```

```
C{1}{2} =

def


C{2} =

     2
   NaN



C{3} =

   NaN
     5



C{4} =

     3
     6



C{5} =

     4
     7
```

Treat Repeated Delimiters as One

Load the file `data3.csv` and preview its contents in a text editor. A screen shot is shown below. Notice the file contains repeated delimiters.

```
filename = fullfile(matlabroot,'examples','matlab','data3.csv');
```



To treat the repeated commas as a single delimiter, use the `MultipleDelimsAsOne` parameter, and set the value to `1` (`true`).

```
fileID = fopen(filename);
C = textscan(fileID,'%f %f %f %f','Delimiter',',',...
'MultipleDelimsAsOne',1);
fclose(fileID);
```

```
celldisp(C)
```

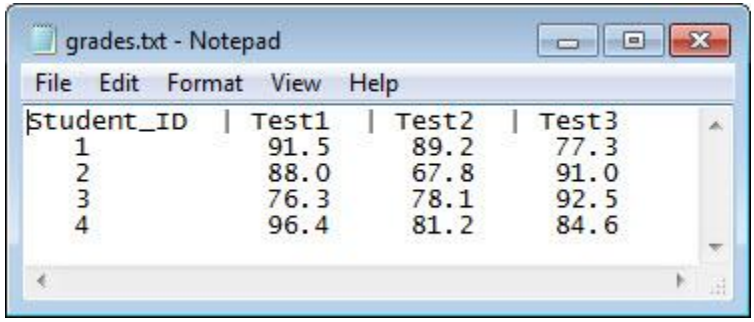
```
C{1} =

     1
     5



C{2} =

     2
     6



C{3} =

     3
     7



C{4} =
```

```
4
8
```

Specify Repeated Conversion Specifiers and Collect Numeric Data

Load the data file `grades.txt` for this example and preview its contents in a text editor. A screen shot is shown below. Notice the file contains repeated delimiters.

```
filename = fullfile(matlabroot,'examples','matlab','grades.txt');
```



Read the column headers using the format `'%s'` four times.

```
fileID = fopen(filename);
formatSpec = '%s';
N = 4;
C_text = textscan(fileID,formatSpec,N,'Delimiter','|');
```

Read the numeric data in the file.

```
C_data0 = textscan(fileID,'%d %f %f %f')
```

```
C_data0=1×4 cell
    {4x1 int32}    {4x1 double}    {4x1 double}    {4x1 double}
```

The default value for `CollectOutput` is `0` (`false`), so `textscan` returns each column of the numeric data in a separate array.

Set the file position indicator to the beginning of the file.

```
frewind(fileID);
```

Reread the file and set CollectOutput to 1 (true) to collect the consecutive columns of the same class into a single array. You can use the `repmat` function to indicate that the `%f` conversion specifier should appear three times. This technique is useful when a format repeats many times.

```
C_text = textscan(fileID,'%s',N,'Delimiter','|');
C_data1 = textscan(fileID,['%d',repmat('%f',[1,3])],'CollectOutput',1)
```

```
C_data1=1×2 cell
    {4x1 int32}    {4x3 double}
```

The test scores, which are all double, are collected into a single 4-by-3 array.

Close the file.

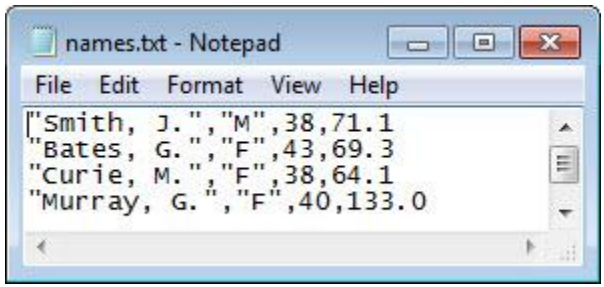
```
fclose(fileID);
```

Read or Skip Quoted Text and Numeric Fields

Read the first and last columns of data from a text file. Skip a column of text and a column of integer data.

Load the file `names.txt` and preview its contents in a text editor. A screen shot is shown below. Notice that the file contains two columns of quoted text, followed by a column of integers, and finally a column of floating point numbers.

```
filename = fullfile(matlabroot,'examples','matlab','names.txt');
```



Read the first and last columns of data in the file. Use the conversion specifier, `%q` to read the text enclosed by double quotation marks (`"`). `%*q` skips the quoted text, `%*d` skips the integer field, and `%f` reads the floating-point number. Specify the comma delimiter using the `'Delimiter'` name-value pair argument.

```
fileID = fopen(filename,'r');
C = textscan(fileID,'%q %*q %*d %f','Delimiter',',');
fclose(fileID);
```

Display the output. `textscan` returns a cell array `C` where the double quotation marks enclosing the text are removed.

```
celldisp(C)
```

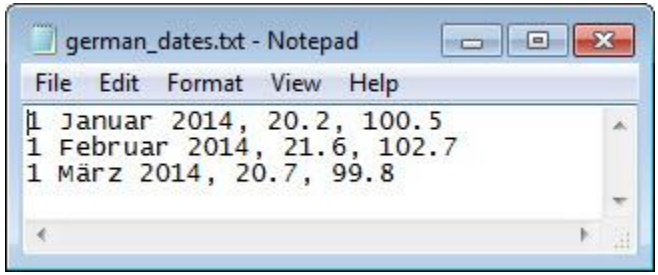
```
C{1}{1} =

Smith, J.


C{1}{2} =

Bates, G.


C{1}{3} =

Curie, M.


C{1}{4} =

Murray, G.


C{1}{5} =

Brown, K.


C{2} =

   71.1000
   69.3000
   64.1000
```

```
  133.0000
   64.9000
```

Read Foreign-Language Dates

Load the file `german_dates.txt` and preview its contents in a text editor. A screen shot is shown below. Notice that the first column of values contains dates in German and the second and third columns are numeric values.

```
filename = fullfile(matlabroot,'examples','matlab','german_dates.txt');
```



Open the file. Specify the character encoding scheme associated with the file as the last input to `fopen`.

```
fileID = fopen(filename,'r','n','ISO-8859-15');
```

Read the file. Specify the format of the dates in the file using the `%{dd % MMMM yyyy}D` specifier. Specify the locale of the dates using the `DateLocale` name-value pair argument.

```
C = textscan(fileID,'%{dd MMMM yyyy}D %f %f',...
    'DateLocale','de_DE','Delimiter',',');
fclose(fileID);
```

View the contents of the first cell in C. The dates display in the language MATLAB uses depending on your system locale.

```
C{1}
```

```
ans = 3x1 datetime array
   01 January 2014
   01 February 2014
   01 March 2014
```

Read Nondefault Control Characters

Use `sprintf` to convert nondefault escape sequences in your data.

Create text that includes a form feed character, `\f`. Then, to read the text using `textscan`, call `sprintf` to explicitly convert the form feed.

```
lyric = sprintf('Blackbird\fsinging\fin\fthe\fdead\fof\fnight');
C = textscan(lyric,'%s','delimiter',sprintf('\f'));
C{1}
```

```
ans = 7x1 cell array
    {'Blackbird'}
    {'singing'  }
    {'in'       }
    {'the'      }
    {'dead'     }
    {'of'       }
    {'night'    }
```

`textscan` returns a cell array, `C`.

Resume Scanning

Resume scanning from a position other than the beginning.

If you resume a scan of the text, `textscan` reads from the beginning each time. To resume a scan from any other position, use the two-output argument syntax in your initial call to `textscan`.

For example, create a character vector called `lyric`. Read the first word of the character vector, and then resume the scan.

```
lyric = 'Blackbird singing in the dead of night';
[firstword,pos] = textscan(lyric,'%9c',1);
lastpart = textscan(lyric(pos+1:end),'%s');
```

Input Arguments

`fileID` — File identifier

numeric scalar

File identifier of an open text file, specified as a number. Before reading a file with `textscan`, you must use `fopen` to open the file and obtain the `fileID`.

Data Types: `double`

`formatSpec` — Format of the data fields

character vector | string

Format of the data fields, specified as a character vector or a string of one or more conversion specifiers. When `textscan` reads the input, it attempts to match the data to the format specified in `formatSpec`. If `textscan` fails to match a data field, it stops reading and returns all fields read before the failure.

The number of conversion specifiers determines the number of cells in output array, `C`.

Numeric Fields

This table lists available conversion specifiers for numeric inputs.

Numeric Input Type	Conversion Specifier	Output Class
Integer, signed	%d	int32
	%d8	int8
	%d16	int16
	%d32	int32
	%d64	int64
Integer, unsigned	%u	uint32
	%u8	uint8

Numeric Input Type	Conversion Specifier	Output Class
	`%u16`	`uint16`
	`%u32`	`uint32`
	`%u64`	`uint64`
Floating-point number	`%f`	`double`
	`%f32`	`single`
	`%f64`	`double`
	`%n`	`double`

Nonnumeric Fields

This table lists available conversion specifiers for inputs that include nonnumeric characters.

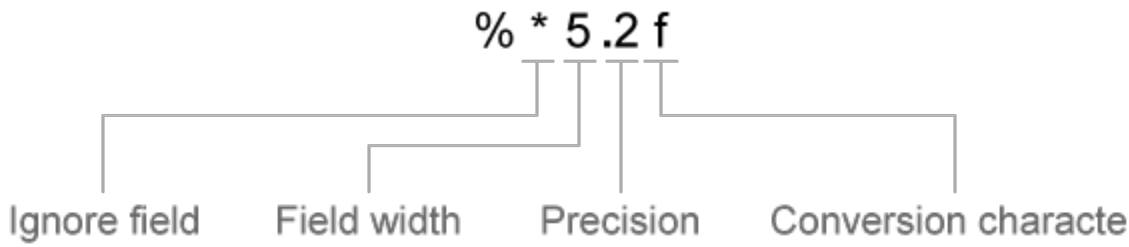
Nonnumeric Input Type	Conversion Specifier	Details
Character	`%c`	Read any single character, including a delimiter.
Text Array	`%s`	Read as a cell array of character vectors.
	`%q`	Read as a cell array of character vectors. If the text begins with a double quotation mark (`"`), omit the leading quotation mark and its accompanying closing mark, which is the second instance of a lone double quotation mark. Replace escaped double quotation marks (for example, `""abc""`) with lone double quotation marks (`"abc"`). `%q` ignores any double quotation marks that appear after the closing double quotation mark. Example: `'%q'` reads `'"Joe ""Lightning"" Smith, Jr."'` as `'Joe "Lightning" Smith, Jr.'`.
Dates and time	`%D`	Read the same way as `%q` above, and then convert to a datetime value.

Nonnumeric Input Type	Conversion Specifier	Details
	%{*fmt*}D	Read the same way as %q above, and then convert it to a datetime value. *fmt* describes the format of the input text. The *fmt* input is a character vector of letter identifiers that is a valid value for the Format property of a datetime. textscan converts text that does not match this format to NaT values. For more information about datetime display formats, see the Format property for datetime arrays. Example: '%{dd-MMM-yyyy}D' specifies the format of a date such as '01-Jan-2014' .
Duration	%T	Read the same way as %q above, and then convert to a duration value.
	%{*fmt*}T	Read the same way as %q above, and then convert it to a duration value. *fmt* describes the format of the input text. The *fmt* input is a character vector of letter identifiers that is a valid value for the Format property of a duration. textscan converts text that does not match this format to NaN values. For more information about duration display formats, see the format property for duration arrays. Example: '%{hh:mm:ss}T' specifies the format of a duration such as '10:30:15', which represents 10 hours, 30 minutes, and 15 seconds.
Category	%C	Read the same way as %q, and then convert to a category name in a categorical array. textscan converts <undefined> text to an undefined value in the output categorical array.

Nonnumeric Input Type	Conversion Specifier	Details
Pattern-matching	`%[...]`	Read as a cell array of character vectors, the characters inside the brackets up to the first nonmatching character. To include `]` in the set, specify it first: `%[]...]`. Example: `%[mus]` reads `'summer '` as `'summ'`.
	`%[^...]`	Exclude characters inside the brackets, reading until the first matching character. To exclude `]`, specify it first: `%[^]...]`. Example: `%[^xrg]` reads `'summer '` as `'summe'`.

Optional Operators

Conversion specifiers in `formatSpec` can include optional operators, which appear in the following order (includes spaces for clarity):



Optional operators include:

- Fields and Characters to Ignore

  `textscan` reads all characters in your file in sequence, unless you tell it to ignore a particular field or a portion of a field.

  Insert an asterisk character (*) after the percent character (%) to skip a field or a portion of a character field.

Operator	Action Taken
`%*k`	Skip the field. *k* is any conversion specifier identifying the field to skip. `textscan` does not create an output cell for any such fields. Example: `'%s %*s %s %s %*s %*s %s'` (spaces are optional) converts the text `'Blackbird singing in the dead of night'` into four output cells with `'Blackbird' 'in' 'the' 'night'`
`'%*ns'`	Skip up to *n* characters, where *n* is an integer less than or equal to the number of characters in the field. Example: `'%*3s %s'` converts `'abcdefg'` to `'defg'`. When the delimiter is a comma, the same delimiter converts `'abcde,fghijkl'` to a cell array containing `'de';'ijkl'`.
`'%*nc'`	Skip *n* characters, including delimiter characters.

- Field Width

  `textscan` reads the number of characters or digits specified by the field width or precision, or up to the first delimiter, whichever comes first. A decimal point, sign (`+` or `-`), exponent character, and digits in the numeric exponent are counted as characters and digits within the field width. For complex numbers, the field width refers to the individual widths of the real part and the imaginary part. For the imaginary part, the field width includes `+` or `−` but not `i` or `j`. Specify the field width by inserting a number after the percent character (%) in the conversion specifier.

  Example: `%5f` reads `'123.456'` as `123.4`.

  Example: `%5c` reads `'abcdefg'` as `'abcde'`.

  When the field width operator is used with single characters (`%c`), `textscan` also reads delimiter, white-space, and end-of-line characters.
  Example: `%7c` reads 7 characters, including white-space, so`'Day and night'` reads as `'Day and'`.

- Precision

  For floating-point numbers (`%n`, `%f`, `%f32`, `%f64`), you can specify the number of decimal digits to read.

Example: `%7.2f` reads `'123.456'` as `123.45`.

- Literal Text to Ignore

  `textscan` ignores the text appended to the `formatSpec` conversion specifier.

  Example: `Level%u8` reads `'Level1'` as `1`.

  Example: `%u8Step` reads `'2Step'` as `2`.

Data Types: `char` | `string`

`N` — Number of times to apply `formatSpec`
`Inf` (default) | positive integer

Number of times to apply `formatSpec`, specified as a positive integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`chr` — Input text
character vector | string

Input text to read.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Names are not case sensitive.

Example: `C = textscan(fileID,formatSpec,'HeaderLines',3,'Delimiter',',')` skips the first three lines of the data, and then reads the remaining data, treating commas as a delimiter.

`CollectOutput` — Logical indicator determining data concatenation
`false` (default) | `true`

Logical indicator determining data concatenation, specified as the comma-separated pair consisting of `'CollectOutput'` and either `true` or `false`. If `true`, then the importing function concatenates consecutive output cells of the same fundamental MATLAB class into a single array.

`CommentStyle` — Symbols designating text to ignore
character vector | cell array of character vectors | string | string array

Symbols designating text to ignore, specified as the comma-separated pair consisting of `'CommentStyle'` and a character vector, cell array of character vectors, string, or string array.

For example, specify a character such as `'%'` to ignore text following the symbol on the same line. Specify a cell array of two character vectors, such as `{'/*','*/'}`, to ignore any text between those sequences.

MATLAB checks for comments only at the start of each field, not within a field.

Example: `'CommentStyle',{'/*','*/'}`

Data Types: `char` | `string`

`DateLocale` — Locale for reading dates
character vector | string

Locale for reading dates, specified as the comma-separated pair consisting of `'DateLocale'` and a character vector in the form *xx_YY*, where *xx* is a lowercase ISO 639-1 two-letter code that specifies a language, and *YY* is an uppercase ISO 3166-1 alpha-2 code that specifies a country. For a list of common values for the locale, see the `Locale` name-value pair argument for the `datetime` function.

Use `DateLocale` to specify the locale in which textscan should interpret month and day of week names and abbreviations when reading text as dates using the `%D` format specifier.

Example: `'DateLocale','ja_JP'`

`Delimiter` — Field delimiter characters
character vector | cell array of character vectors | string | string array

Field delimiter characters, specified as the comma-separated pair consisting of `'Delimiter'` and a character vector or a cell array of character vectors. Specify multiple delimiters in a cell array of character vectors.

Example: `'Delimiter',{';','*'}`

`textscan` interprets repeated delimiter characters as separate delimiters, and returns an empty value to the output cell.

Within each row of data, the default field delimiter is white-space. White-space can be any combination of space (`' '`), backspace (`'\b'`), or tab (`'\t'`) characters. If you do not specify a delimiter, then:

- the delimiter characters are the same as the white-space characters. The default white-space characters are `' '`, `'\b'`, and `'\t'`. Use the `'Whitespace'` name-value pair argument to specify alternate white-space characters.
- `textscan` interprets repeated white-space characters as a single delimiter.

When you specify one of the following escape sequences as a delimiter, `textscan` converts that sequence to the corresponding control character:

`\b`	Backspace
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\\`	Backslash (`\`)

Data Types: `char` | `string`

`EmptyValue` — Returned value for empty numeric fields
`NaN` (default) | scalar

Returned value for empty numeric fields in delimited text files, specified as the comma-separated pair consisting of `'EmptyValue'` and a scalar.

`EndOfLine` — End-of-line characters
character vector | string

End-of-line characters, specified as the comma-separated pair consisting of `'EndOfLine'` and a character vector or string. The character vector must be `'\r\n'` or it must specify a single character. Common end-of-line characters are a newline character (`'\n'`) or a carriage return (`'\r'`). If you specify `'\r\n'`, then the importing function treats any of `\r`, `\n`, and the combination of the two (`\r\n`) as end-of-line characters.

The default end-of-line sequence is `\n`, `\r`, or `\r\n`, depending on the contents of your file.

If there are missing values and an end-of-line sequence at the end of the last line in a file, then the importing function returns empty values for those fields. This ensures that individual cells in output cell array, `C`, are the same size.

Example: `'EndOfLine',':'`

Data Types: `char` | `string`

`ExpChars` — Exponent characters
`'eEdD'` (default) | character vector | string

Exponent characters, specified as the comma-separated pair consisting of `'ExpChars'` and a character vector or string. The default exponent characters are `e`, `E`, `d`, and `D`.

Data Types: `char` | `string`

`HeaderLines` — Number of header lines
`0` (default) | positive integer

Number of header lines, specified as the comma-separated pair consisting of `'HeaderLines'` and a positive integer. `textscan` skips the header lines, including the remainder of the current line.

`MultipleDelimsAsOne` — Multiple delimiter handling
`0 (false)` (default) | `1 (true)`

Multiple delimiter handling, specified as the comma-separated pair consisting of `'MultipleDelimsAsOne'` and either `true` or `false`. If `true`, then the importing function treats consecutive delimiters as a single delimiter. Repeated delimiters separated by white-space are also treated as a single delimiter. You must also specify the `Delimiter` option.

Example: `'MultipleDelimsAsOne',1`

`ReturnOnError` — Behavior when `textscan` fails to read or convert
`1 (true)` (default) | `0 (false)`

Behavior when `textscan` fails to read or convert, specified as the comma-separated pair consisting of `'ReturnOnError'` and either `true` or `false`. If `true`, `textscan` terminates without an error and returns all fields read. If `false`, `textscan` terminates with an error and does not return an output cell array.

`TreatAsEmpty` — Placeholder text to treat as empty value
character vector | cell array of character vectors | string | string array

Placeholder text to treat as empty value, specified as the comma-separated pair consisting of `'TreatAsEmpty'` and a character vector, cell array of character vectors, string, or string array. This option only applies to numeric fields.

Data Types: `char` | `string`

`Whitespace` — White-space characters
`' \b\t'` (default) | character vector | string

White-space characters, specified as the comma-separated pair consisting of `'Whitespace'` and a character vector or string containing one or more characters. `textscan` adds a space character, `char(32)`, to any specified `Whitespace`, unless `Whitespace` is empty (`''`) and `formatSpec` includes any conversion specifier.

When you specify one of the following escape sequences as any white-space character, `textscan` converts that sequence to the corresponding control character:

`\b`	Backspace
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\\`	Backslash (`\`)

Data Types: `char` | `string`

`TextType` — Output data type of text
`'char'` (default) | `'string'`

Output data type of text, specified as the comma-separated pair consisting of `'TextType'` and either `'char'` or `'string'`. If you specify the value `'char'`, then `textscan` returns text as a cell array of character vectors. If you specify the value `'string'`, then `textscan` returns text as an array of type `string`.

Output Arguments

`C` — File or text data
cell array

File or text data, returned as a cell array.

For each numeric conversion specifier in `formatSpec`, the `textscan` function returns a `K`-by-1 MATLAB numeric vector to the output cell array, `C`, where `K` is the number of times that `textscan` finds a field matching the specifier.

For each text conversion specifier (`%s`, `%q`, or `%[...]`) in `formatSpec`, the `textscan` function returns a `K`-by-1 cell array of character vectors, where `K` is the number of times that `textscan` finds a field matching the specifier. For each character conversion that includes a field width operator, `textscan` returns a `K`-by-`M` character array, where `M` is the field width.

For each datetime or categorical conversion specifier in `formatSpec`, the `textscan` function returns a `K`-by-1 datetime or categorical vector to the output cell array, `C`, where `K` is the number of times that `textscan` finds a field matching the specifier.

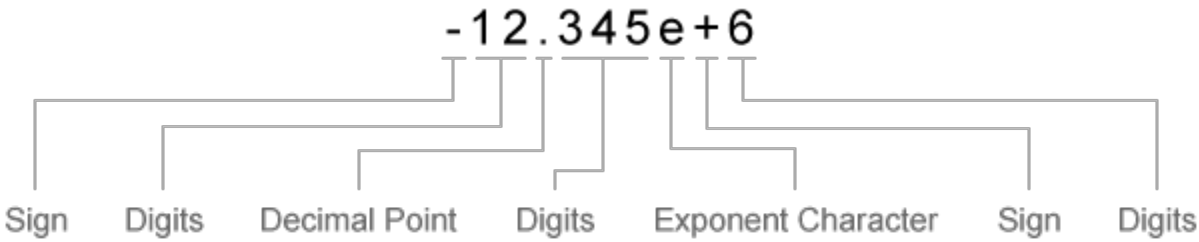
`position` — Position in the file or character vector
integer

Position at the end of the scan, in the file or the character vector, returned as an integer of class `double`. For a file, `ftell(fileID)` would return the same value after calling `textscan`. For a character vector, `position` indicates how many characters `textscan` read.

Algorithms

`textscan` converts numeric fields to the specified output type according to MATLAB rules regarding overflow, truncation, and the use of `NaN`, `Inf`, and `-Inf`. For example, MATLAB represents an integer `NaN` as zero. If `textscan` finds an empty field associated with an integer format specifier (such as `%d` or `%u`), it returns the empty value as zero and not `NaN`.

When matching data to a text conversion specifier, `textscan` reads until it finds a delimiter or an end-of-line character. When matching data to a numeric conversion specifier, `textscan` reads until it finds a nonnumeric character. When `textscan` can no longer match the data to a particular conversion specifier, it attempts to match the data to the next conversion specifier in the `formatSpec`. Sign (`+` or `-`), exponent characters, and decimal points are considered numeric characters.



Sign	Digits	Decimal Point	Digits	Exponent Character	Sign	Digits
Read one sign character if it exists.	Read one or more digits.	Read one decimal point if it exists.	If there is a decimal point, read one or more digits that immediately follow it.	Read one exponent character if it exists.	If there is an exponent character, read one sign character.	If there is an exponent character, read one or more digits that follow it.

`textscan` imports any complex number as a whole into a complex numeric field, converting the real and imaginary parts to the specified numeric type (such as `%d` or `%f`). Valid forms for a complex number are:

`±<real>±<imag>i|j`	Example: `5.7-3.1i`
`±<imag>i|j`	Example: `-7j`

Do not include embedded white space in a complex number. `textscan` interprets embedded white space as a field delimiter.

See Also

`fopen` | `fread` | `fscanf` | `load` | `readcell` | `readmatrix` | `readtable` | `readvars` | `uiimport`

Topics

"Import Block of Mixed Data from Text File into Table or Cell Array"
"Access Data in Cell Array"

"Import Text Files"

Introduced before R2006a

textwrap

Wrap text for user interface control

Note

Use this function only with GUIDE, or with apps created using the `figure` function

Syntax

```
wrappedtext = textwrap(c,txt)
wrappedtext = textwrap(c,txt,numchar)
[wrappedtext,position] = textwrap( ___ )
```

Description

`wrappedtext = textwrap(c,txt)` returns text wrapped at a character width that fits within the specified `UIControl` object, `c`. The UI control object must be one created with the `uicontrol` function whose `'Style'` property value is set to `'text'` or `'edit'`. For example, `c = uicontrol('Style','text')`.

`wrappedtext = textwrap(c,txt,numchar)` returns text that wraps each line at the specified number of characters. Spaces are included in the character count. `textwrap` avoids splitting words when possible. If a word cannot be accommodated within the specified number of characters, then `textwrap` moves it to the start of the next line.

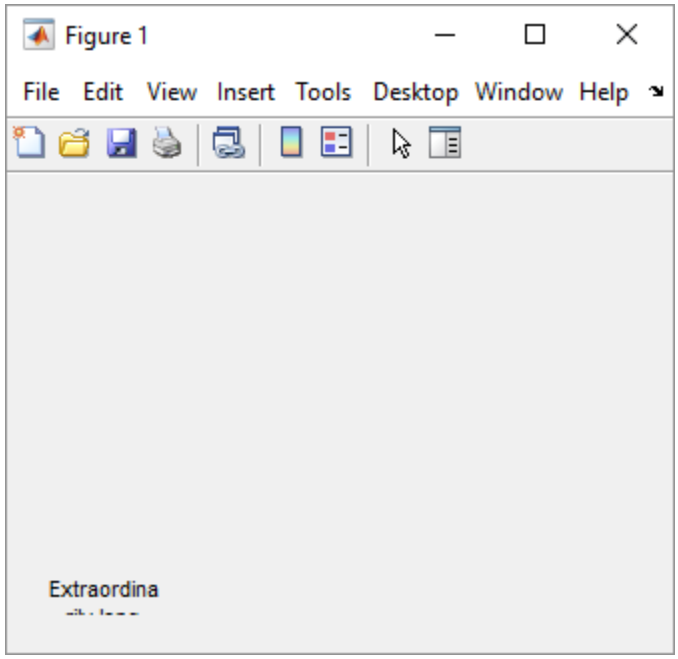
`[wrappedtext,position] = textwrap( ___ )` also returns the recommended position for the UI control based on the text to be wrapped. The returned position is one that allows the full text to display in the `uicontrol` without clipping. If a UI control is not specified, the position vector contains all zeros.

Examples

Preview Cell Array of Wrapped Text

Specify two lines of text for the `String` property of a static text field.

```
c = uicontrol('Style','text');
c.String = {'Extraordinarily long text will be wrapped', ...
'inside of a static text field.'};
```



The text is cut off and wraps across more than two lines because the default width and height of the text field are not large enough to accommodate the full length of the specified text. Notice how the word "Extraordinarily" is also split across two lines.

Preview the cell array of character vectors that is recommended for wrapping the text so that it fits inside the width of the UI control with the least amount of resizing.

```
wrappedtext = textwrap(c,c.String)
```

```
wrappedtext =

  7×1 cell array
```

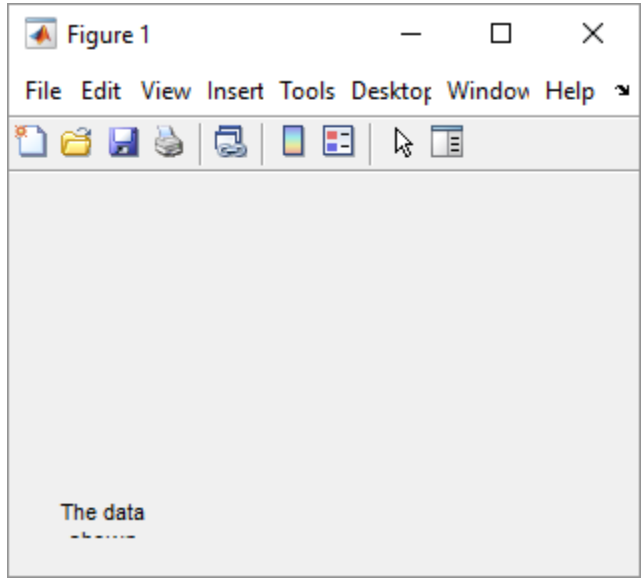
```
{'Extraordinarily'}
{'long text'      }
{'will be'        }
{'wrapped'        }
{'inside of a'    }
{'static text'    }
{'field.'         }
```

Display Wrapped Text in UI Control

Wrap text at a specified character width and display it in a static text field.

Create a static text field at the default position of `[20 20 60 20]`. Specify text to display in it.

```
c = uicontrol('Style','text');
c.String = {'The data shown represents 18 months of observations.'};
```



The text is cut off and displays on multiple lines because the default width and height values of the `uicontrol` are too small to accommodate the full text.

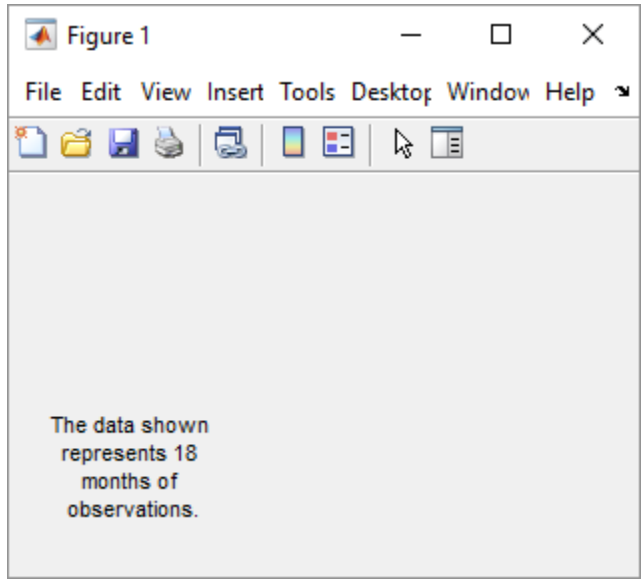
Preview the wrapped text and the recommended position of the `uicontrol` based on a maximum text width of 16 characters.

```
[wrappedtext,position] = textwrap(c,c.String,16)
```

```
wrappedtext =

  4×1 cell array

    {'The data shown '}
    {'represents 18 ' }
    {'months of '     }
    {'observations.'  }


position =

    20    20    86    64
```

Display the wrapped text in the text field and move it to the recommended position.

```
c.String = wrappedtext;
c.Position = position;
```



Input Arguments

c — UI control object
`UIControl` object (default)

UI control object, specified as a `UIControl` object. The `UIControl` object must support multiline text. For instance, its `'Style` property can be `'text'` or `'edit'`. Use this argument to determine how text wraps in the specified UI control, or to determine the recommended size for the UI control based on the text to be wrapped.

`txt` — Text
cell array of character vectors | string array

Text to wrap, specified as a cell array of character vectors, a string array, or a string scalar.

Example: `{'Please select an answer from the options below.'}`

Example: `["Enter your name using","the format LastName, FirstName"]`

`numchar` — Number of characters
positive integer

Number of characters in each line of text, specified as a positive integer. Use this argument to specify the maximum character width for each line. If `numchar` exceeds the number of characters in `txt`, then the text does not wrap.

Output Arguments

`wrappedtext` — Wrapped text
cell array of character vectors

Wrapped text, returned as a cell array of character vectors. To display the text on the specified UI control, you must assign `wrappedtext` to the `String` property of the UI control.

`position` — Position
`[left bottom width height]`

Position recommended for the UI control, returned as a four-element vector of the form `[left bottom width height]`. The units are the same as the units of the UI control. The returned position optimizes the width and height of the `uicontrol` so that the specified text can display across multiple lines, without clipping. If a UI control is not specified, then the position vector contains all zeros.

See Also

`align` | `uicontrol`

Introduced before R2006a

tfqmr

Transpose-free quasi-minimal residual method

Syntax

```
x = tfqmr(A,b)
x = tfqmr(afun,b)
x = tfqmr(a,b,tol)
x = tfqmr(a,b,tol,maxit)
x = tfqmr(a,b,tol,maxit,m)
x = tfqmr(a,b,tol,maxit,m1,m2,x0)
[x,flag] = tfqmr(A,B,...)
[x,flag,relres] = tfqmr(A,b,...)
[x,flag,relres,y]y(A,b,...)
[x,flag,relres,iter,resvec] = tfqmr(A,b,...)
```

Description

`x = tfqmr(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and the right-hand side column vector `b` must have length `n`.

`x = tfqmr(afun,b)` accepts a function handle, `afun`, instead of the matrix `A`. The function, `afun(x)`, accepts a vector input `x` and returns the matrix-vector product `A*x`. In all of the following syntaxes, you can replace `A` by `afun`. “Parameterizing Functions” explains how to provide additional parameters to the function `afun`.

`x = tfqmr(a,b,tol)` specifies the tolerance of the method. If `tol` is `[]` then `tfqmr` uses the default, 1e-6.

`x = tfqmr(a,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]` then `tfqmr` uses the default, `min(N,20)`.

`x = tfqmr(a,b,tol,maxit,m)` and `x = tfqmr(a,b,tol,maxit,m1,m2)` use preconditioners `m` or `m=m1*m2` and effectively solve the system `A*inv(M)*x = B` for `x`. If

`M` is `[]` then a preconditioner is not applied. `M` may be a function handle `mfun` such that `mfun(x)` returns `m\x`.

`x = tfqmr(a,b,tol,maxit,m1,m2,x0)` specifies the initial guess. If `x0` is `[]` then `tfqmr` uses the default, an all zero vector.

`[x,flag] = tfqmr(A,B,...)` also returns a convergence flag:

Flag	Convergence
0	`tfqmr` converged to the desired tolerance `tol` within `maxit` iterations.
1	`tfqmr` iterated `maxit` times but did not converge.
2	Preconditioner `m` was ill-conditioned.
3	`tfqmr` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `tfqmr` became too small or too large to continue computing.

`[x,flag,relres] = tfqmr(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is 0, then `relres <= tol`.

`[x,flag,relres,y]y(A,b,...)` also returns the iteration number at which `x` was computed: `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = tfqmr(A,b,...)` also returns a vector of the residual norms at each iteration, including `norm(b-A*x0)`.

Examples

Using tfqmr with Matrix or Function Handle Input

This example shows how to use `tfqmr` with a matrix input and with a function input.

```
n = 100; on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x = tfqmr(A,b,tol,maxit,M1,M2,[]);
```

You can also use a matrix-vector product function as input:

```
function y = afun(x,n)
y = 4 * x;
y(2:n) = y(2:n) - 2 * x(1:n-1);
y(1:n-1) = y(1:n-1) - x(2:n);
x1 = tfqmr(@(x)afun(x,n),b,tol,maxit,M1,M2);
```

If `applyOp` is a function suitable for use with `qmr`, it may be used with `tfqmr` by wrapping it in an anonymous function:

```
x1 = tfqmr(@(x)applyOp(x,'notransp'),b,tol,maxit,M1,M2);
```

Using tfqmr with a Preconditioner

This example demonstrates the use of a preconditioner.

Load `A = west0479`, a real 479-by-479 nonsymmetric sparse matrix.

```
load west0479;
A = west0479;
```

Define `b` so that the true solution is a vector of all ones.

```
b = full(sum(A,2));
```

Set the tolerance and maximum number of iterations.

```
tol = 1e-12;
maxit = 20;
```

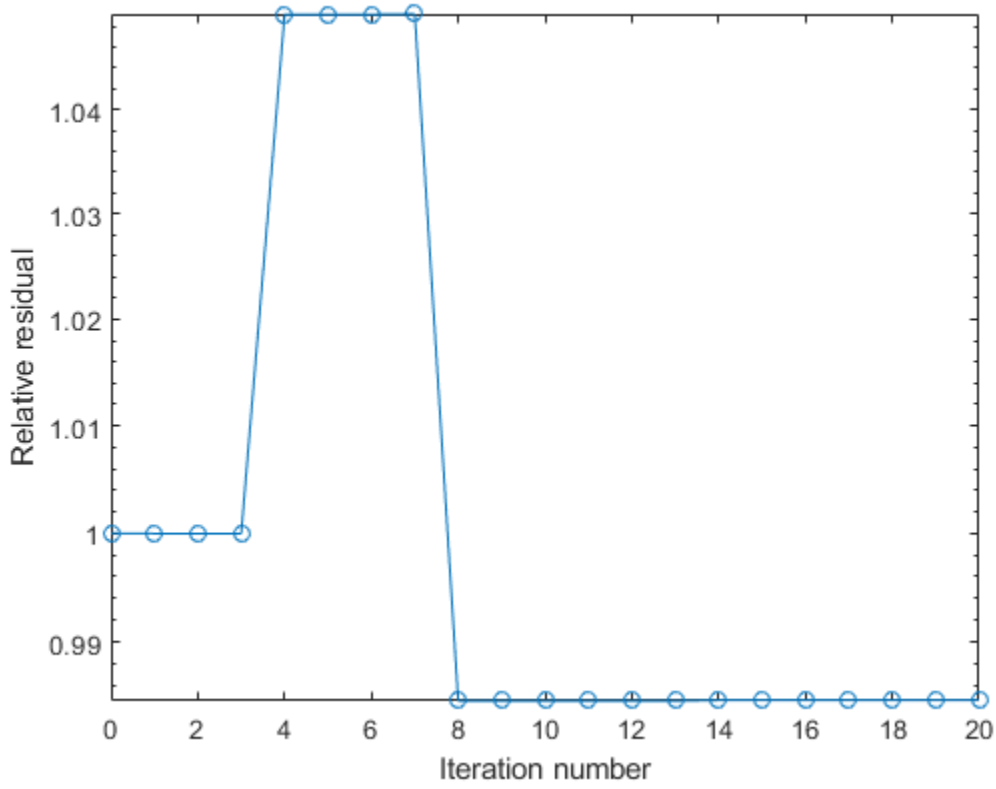
Use `tfqmr` to find a solution at the requested tolerance and number of iterations.

```
[x0,fl0,rr0,it0,rv0] = tfqmr(A,b,tol,maxit);
```

`fl0` is 1 because `tfqmr` does not converge to the requested tolerance `1e-12` within the requested 20 iterations. The seventeenth iterate is the best approximate solution and is the one returned as indicated by `it0 = 17`. MATLAB® stores the residual history in `rv0`.

Plot the behavior of `tfqmr`.

```
semilogy(0:maxit,rv0(1:maxit+1)/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



Note that like `bicgstab`, `tfqmr` keeps track of half iterations. The plot shows that the solution does not converge. You can use a preconditioner to improve the outcome.

Create the preconditioner with `ilu`, since the matrix `A` is nonsymmetric.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-5));
```

```
Error using ilu
There is a pivot equal to zero. Consider decreasing
the drop tolerance or consider using the 'udiag' option.
```

MATLAB cannot construct the incomplete LU as it would result in a singular factor, which is useless as a preconditioner.

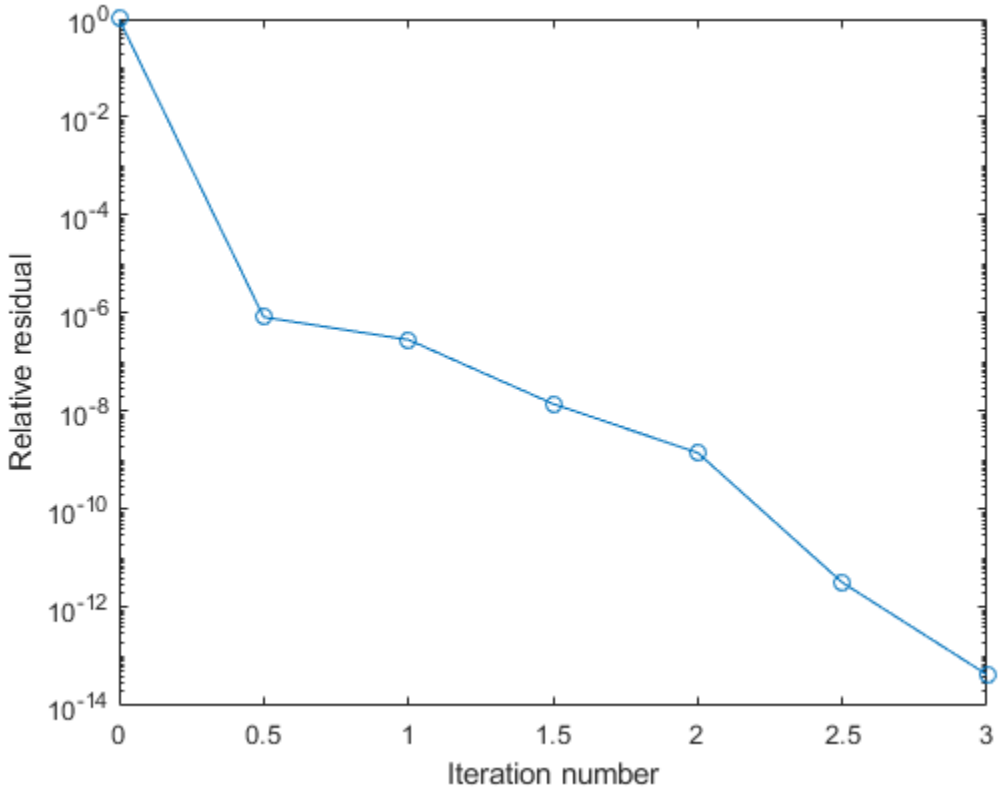
You can try again with a reduced drop tolerance, as indicated by the error message.

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
[x1,fl1,rr1,it1,rv1] = tfqmr(A,b,tol,maxit,L,U);
```

`fl1` is 0 because `tfqmr` drives the relative residual to `4.1410e-014` (the value of `rr1`). The relative residual is less than the prescribed tolerance of `1e-12` at the sixth iteration (the value of `it1`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. The output `rv1(1)` is `norm(b)`, and the output `rv1(7)` is `norm(b-A*x2)`.

You can follow the progress of `tfqmr` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0).

```
semilogy(0:0.5:it1,rv1/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



Extended Capabilities

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- If `m1` is a function, then it is applied independently to each row.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bicg` | `bicgstab` | `bicgstabl` | `cgs` | `gmres` | `ilu` | `lsqr` | `minres` | `mldivide` | `pcg` | `qmr` | `symmlq`

Topics

“Create Function Handle”

thetalim

Set or query theta-axis limits for polar axes

Syntax

```
thetalim(limits)
```

```
thetalim('auto')
thetalim('manual')
```

```
tl = thetalim
m = thetalim('mode')
```

```
___ = thetalim(pax, ___ )
```

Description

`thetalim(limits)` specifies the theta-axis limits for the current polar axes. Specify `limits` as a two-element vector of the form `[thetamin thetamax]`, where `thetamax` is a numeric value greater than `thetamin`.

`thetalim('auto')` lets MATLAB choose the theta-axis limits. This command sets the `ThetaLimMode` property for the polar axes object to `'auto'`.

`thetalim('manual')` prevents the limits from changing automatically. This command sets the `ThetaLimMode` property for the polar axes object to `'manual'`.

`tl = thetalim` returns a two-element vector containing the limits for the current polar axes.

`m = thetalim('mode')` returns the current value of the limits mode, which is either `'auto'` or `'manual'`. By default, the mode is automatic unless you specify limits or set the mode to manual.

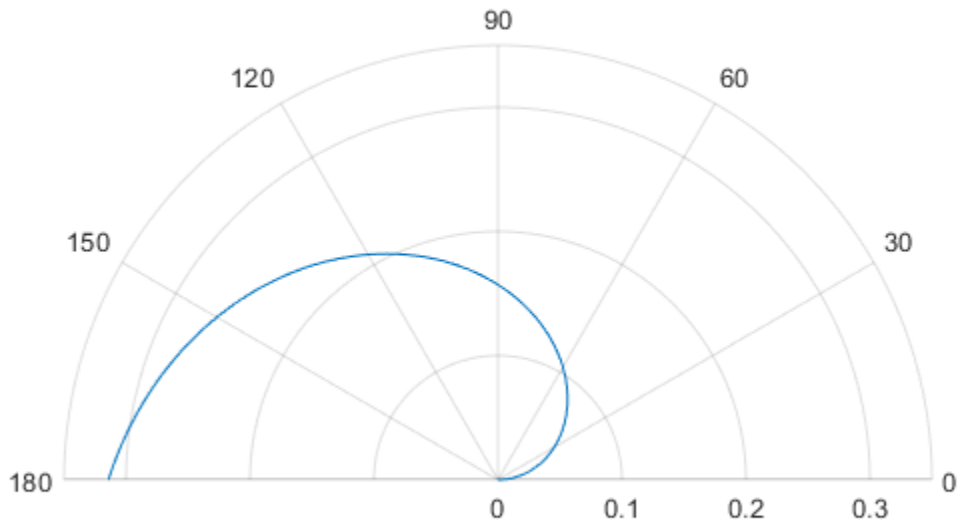
`___ = thetalim(pax, ___ )` uses the polar axes object specified by `pax` instead of the current polar axes. Specify `pax` as the first input argument. Include additional input or output arguments only if the original syntax supported them.

Examples

Specify theta-Axis Limits

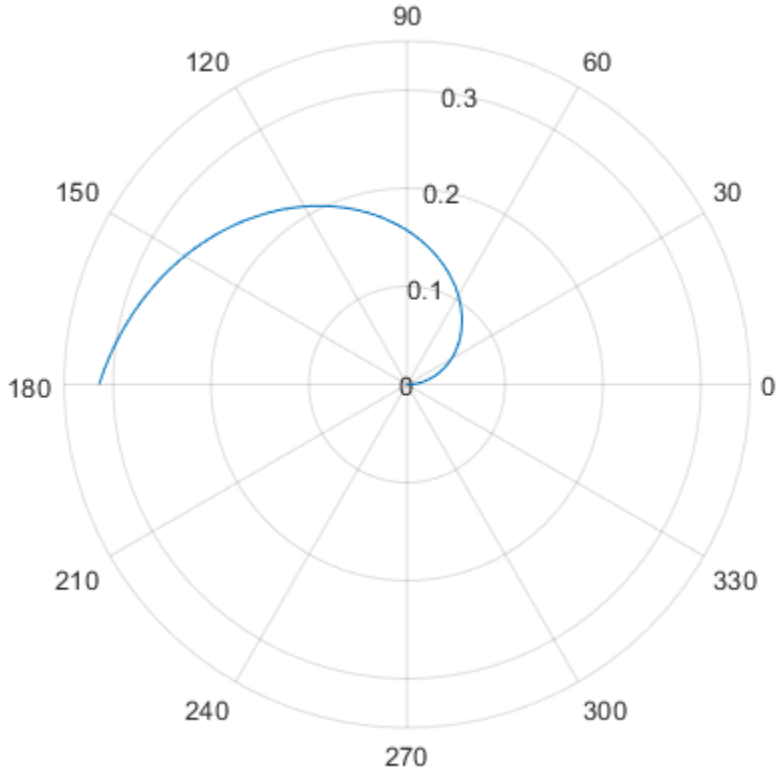
Create a polar plot and change the theta-axis limits.

```
theta = linspace(0,pi);
rho = theta/10;
polarplot(theta,rho)
thetalim([0 180])
```



Set the limits back to the original values.

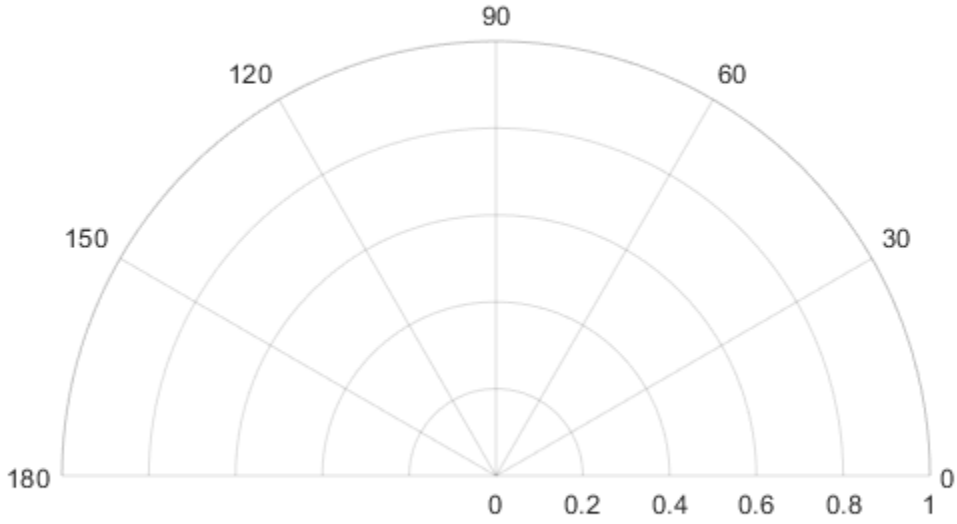
```
thetalim('auto')
```



Specify theta-Axis Limits for Specific Polar Axes

Set the limits for a specific polar axes by specifying the polar axes object as the first input to `thetalim`. Otherwise, `thetalim` sets the limits for the current axes.

```
pax = polaraxes;
thetalim(pax,[0 180])
```



Input Arguments

`limits` — Minimum and maximum limits
two-element vector

Minimum and maximum limits, specified as a two-element vector of the form `[thetamin thetamax]`. When you specify the limits, the `ThetaLim` property for the polar axes object updates to the specified values.

Example: `[0 1]`

pax — Polar axes object
polar axes object

Polar axes object. If you do not specify a polar axes object, then `thetalim` sets the limits for the current polar axes.

Output Arguments

`tl` — Current limits
two-element vector

Current limits, returned as a two-element vector of the form `[thetamin thetamax]`. Querying the limits returns the value of the `ThetaLim` property for the polar axes object.

`m` — Current limits mode
`'auto'` | `'manual'`

Current limits mode, returned as one of these values:

- `'auto'` — Limits automatically update to reflect changes in the data.
- `'manual'` — Limits do not automatically update.

Querying the theta-axis limits mode returns the value of the `ThetaLimMode` property for the polar axes object.

See Also

Functions
`deg2rad` | `polarplot` | `rad2deg` | `rlim` | `title`

Properties
PolarAxes

Introduced in R2016a

thetatickformat

Specify *theta*-axis tick label format

Syntax

```
thetatickformat(fmt)
thetatickformat(pax, ___ )
```

```
tfmt = thetatickformat
tfmt = thetatickformat(pax)
```

Description

`thetatickformat(fmt)` sets the format for the *theta*-axis tick labels. For example, specify `fmt` as `'usd'` to display the labels in U.S. dollars.

`thetatickformat(pax, ___ )` uses the axes specified by `ax` instead of the current axes. Specify `pax` as the first input argument.

`tfmt = thetatickformat` returns the format style used for *theta*-axis tick labels of the current axes. Depending on the type of labels along the *theta*-axis, `tfmt` is a character vector of a numeric format.

`tfmt = thetatickformat(pax)` returns the format style used for the axes specified by `pax` instead of the current axes.

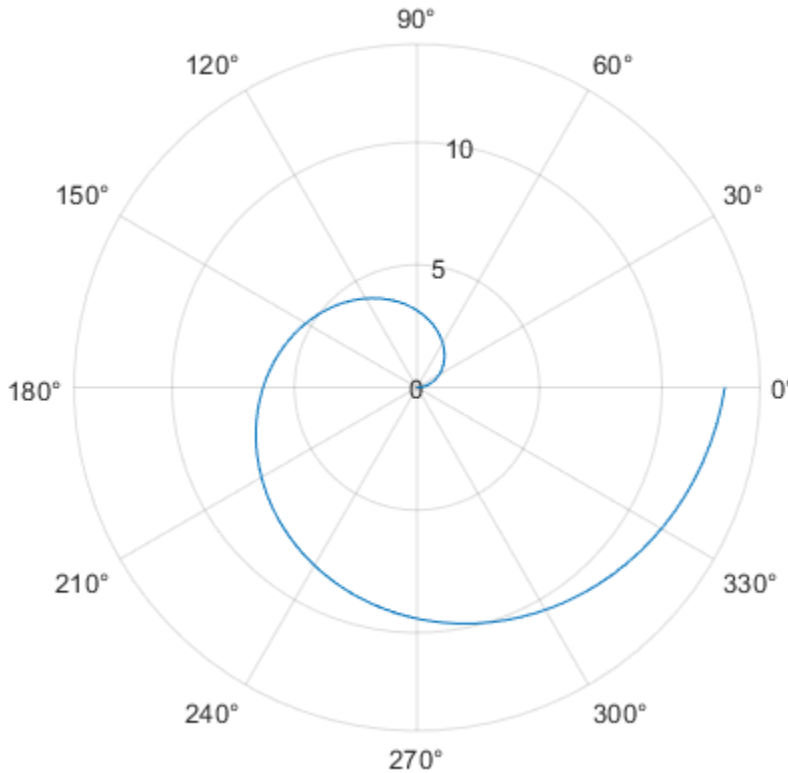
Examples

Display *theta*-Axis Tick Labels as Degrees

Create a polar plot. Display the tick labels along the *theta*-axis as degree values.

```
th = linspace(0,2*pi);
r = 2*th;
```

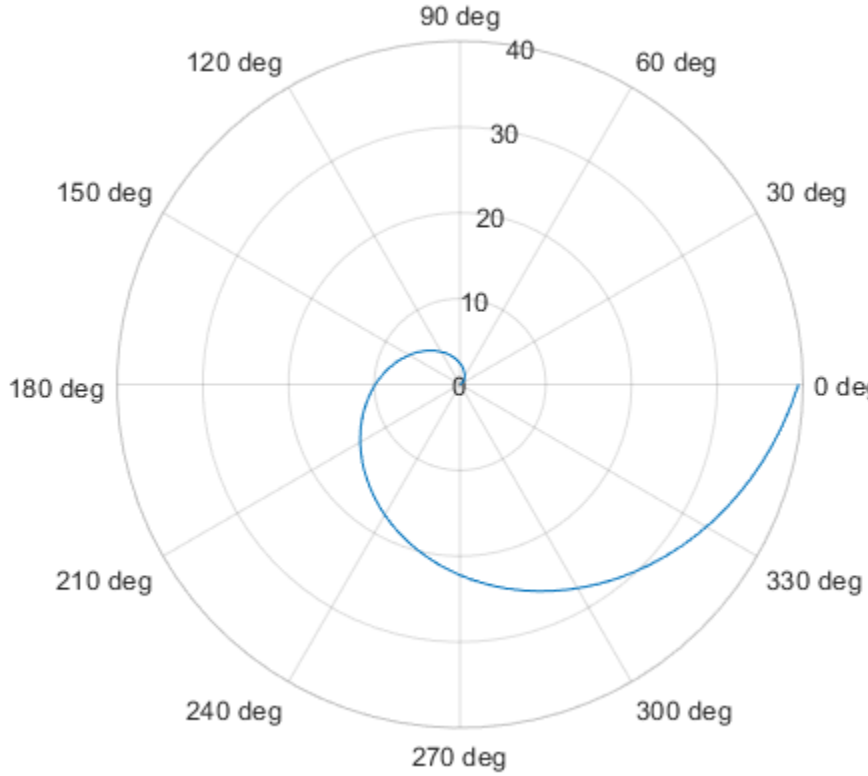
```
polarplot(th,r)
thetatickformat('degrees')
```



Display Text After Each *theta*-Axis Tick Label

Create a polar plot. Display the tick labels along the *theta*-axis with the text "deg" after each value.

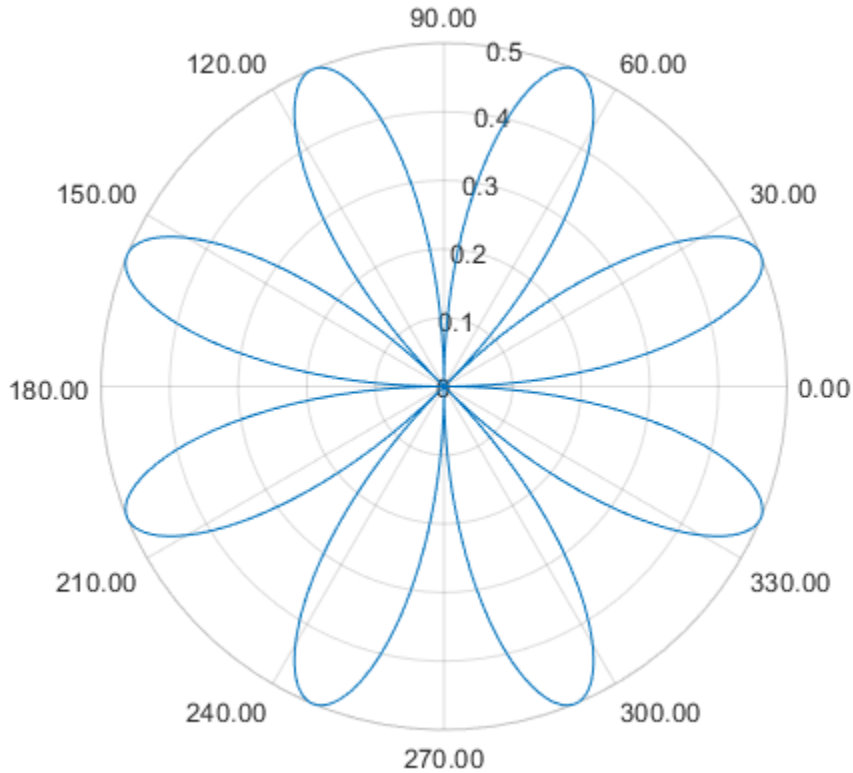
```
th = linspace(0,2*pi);
r = th.^2;
polarplot(th,r)
thetatickformat('%g deg')
```



Control Number of Decimals for *theta*-Axis Tick Labels

Display the *theta*-axis tick labels with two decimal places. Control the decimal places by passing `thetatickformat` a character vector of a numeric format that uses fixed-point notation for the conversion character and a precision value of 2.

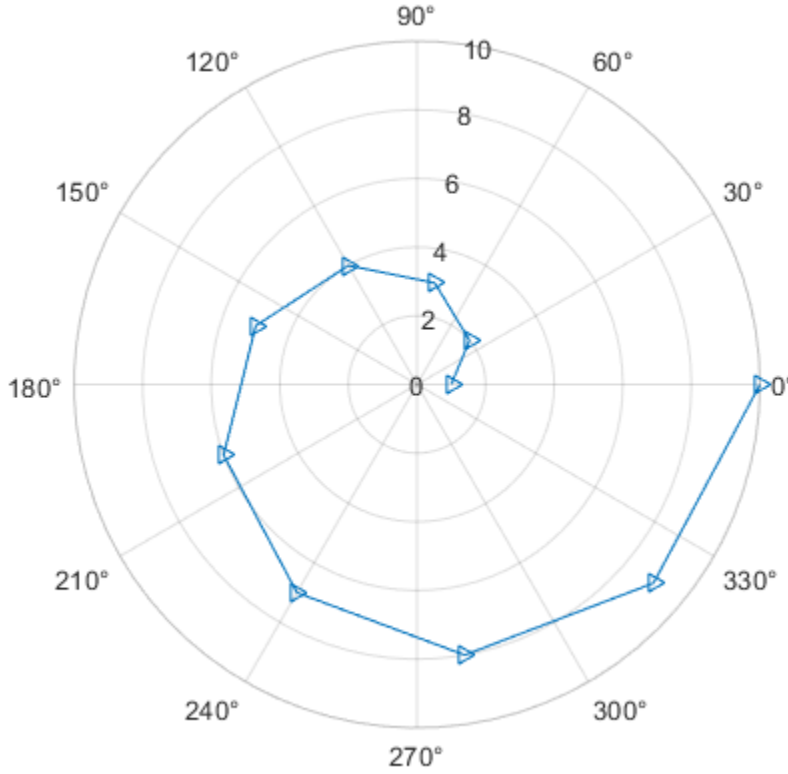
```
theta = 0:0.01:2*pi;
rho = sin(2*theta).*cos(2*theta);
polarplot(theta,rho)
thetatickformat('%.2f')
```



Specify *theta*-Axis Tick Label Format for Specific Polar Axes

Create a polar plot and assign the polar axes object to the variable `pax`. Ensure that `thetatickformat` affects the polar axes you just created by passing `pax` as the first input argument to the function.

```
polarplot(1:10,'->')
pax = gca;
thetatickformat(pax,'degrees')
```



Input Arguments

`fmt` — Format for numeric tick labels
`'%g'` (default) | character vector | string

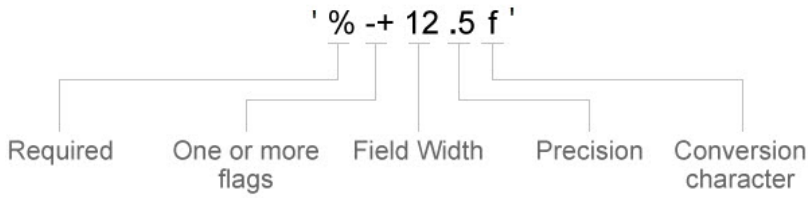
Format for numeric tick labels, specified as a character vector or string. You can specify one of the formats listed in this table. Alternatively, you can specify a custom format.

Predefined Format	Description
`'usd'`	U.S. dollars. This option is equivalent using `'$%,.2f'`. If the labels use scientific notation, this option sets the exponent value to 0.
`'eur'`	Euro. This option is equivalent to using `'\x20AC%,.2f'` with an exponent value of 0.
`'gbp'`	British pound. This option is equivalent to using `'\x00A3%,.2f'` with an exponent value of 0.
`'jpy'`	Japanese yen. This option is equivalent to using `'\x00A5%,d'` with an exponent value of 0.
`'degrees'`	Display degree symbol after values. This option is equivalent to using `'%g\x00B0'` with the default exponent value.
`'percentage'`	Display percent sign after values. This option is equivalent to using `'%g%%'` with the default exponent value.
`'auto'`	Default format of `'%g'` with the default exponent value.

Example: `thetatickformat('usd')`

Custom Numeric Format

You can specify a custom numeric format by creating a character vector or string containing identifiers.



Identifiers are optional, except the percent sign and conversion character. Construct the format in this order:

- One or more flags — Options such as adding a plus sign before positive values. For a full list of options, see the table of Optional Flags.
- Field width — Minimum number of characters to print in the tick label. Specify the field width as an integer value. If the number of significant digits in the tick value is smaller than the field width, then the label is padded with spaces.
- Precision — Number of digits to the right of the decimal point or the number of significant digits, depending on the conversion character. Specify the precision as an integer value.
- Conversion character — Value type. For a full list of options, see the table of Conversion Characters. If you specify a conversion that does not fit the data, then MATLAB overrides the specified conversion, and uses `%e`.

Also, you can specify literal text at the beginning or end of the format. To print a single quotation mark, use `''`. To print a percent character, use `%%`.

Example: `thetatickformat('%.2f')` displays the values using fixed-point notation with two decimal places.

Example: `thetatickformat('$%.2f')` displays a dollar sign before each value.

Example: `thetatickformat('%.2f million')` displays `million` after each value.

Optional Flags

Identifier	Description	Example of Numeric Format
`,`	Display commas every three digits, such as `'1,000'`.	`'%,4.4g'`
`+`	Print the sign character (`+`) for positive values, such as `'+100'`.	`'%+4.4g'`
`0`	Pad the field width with leading zeros instead of spaces, such as `'0100'`.	`'%04.4g'`
`-`	Left-justify, which pads the end of the value with spaces instead of the beginning. For example, if the field width is `4`, then this flag formats the label as `'100 '` instead of `' 100'`.	`'%-4.4g'`
`#`	For the `%f`, `%e`, and `%g` conversion characters, print the decimal point even when the precision is 0, such as `'100.'`. For `%g`, do not remove trailing zeros.	`'%#4.4g'`

Conversion Characters

Identifier	Description	Example
`d` or `i`	Signed integer with base 10. The precision value indicates the number of significant digits.	`'%.4d'` displays π as `0003`.
`f`	Fixed-point notation. The precision value indicates the number of decimal places.	`'%.4f'` displays π as `3.1416`.
`e`	Exponential notation. The precision value indicates the number of decimal places.	`'%.4e'` displays π as `3.1416x10`^{0}.
`g`	The more compact version of `e` or `f`, with no trailing zeros. The precision value indicates the maximum number of decimal places.	`'%.4g'` displays π as `3.1416`.

pax — One or more polar axes
current axes (default) | single object | vector of objects

One or more polar axes, specified as a single object or a vector of objects. If you do not specify a polar axes object, then `thetatickformat` uses the current axes.

Algorithms

The `thetatickformat` function sets and queries the `TickLabelFormat` property of the ruler object associated with the *theta*-axis.

See Also

`rtickformat` | `thetalim` | `thetaticklabels` | `thetaticks`

Introduced in R2016b

thetaticklabels

Set or query *theta*-axis tick labels

Syntax

```
thetaticklabels(labels)
tl = thetaticklabels
```

```
thetaticklabels('auto')
thetaticklabels('manual')
m = thetaticklabels('mode')
```

```
___ = thetaticklabels(pax, ___ )
```

Description

`thetaticklabels(labels)` sets the *theta*-axis tick labels for the current axes. Specify `labels` as a string array or a cell array of character vectors; for example, `{'E','N','W','S'}`. If you specify the labels, then the *theta*-axis tick values and tick labels no longer update automatically based on changes to the axes.

`tl = thetaticklabels` returns the *theta*-axis tick labels for the current axes.

`thetaticklabels('auto')` sets an automatic mode, enabling the axes to determine the *theta*-axis tick labels. Use this option if you set the labels and then want to set them back to the default values.

`thetaticklabels('manual')` sets a manual mode, freezing the *theta*-axis tick labels at the current values.

`m = thetaticklabels('mode')` returns the current value of the *theta*-axis tick labels mode, which is either `'auto'` or `'manual'`. By default, the mode is automatic unless you specify the tick labels or set the mode to manual.

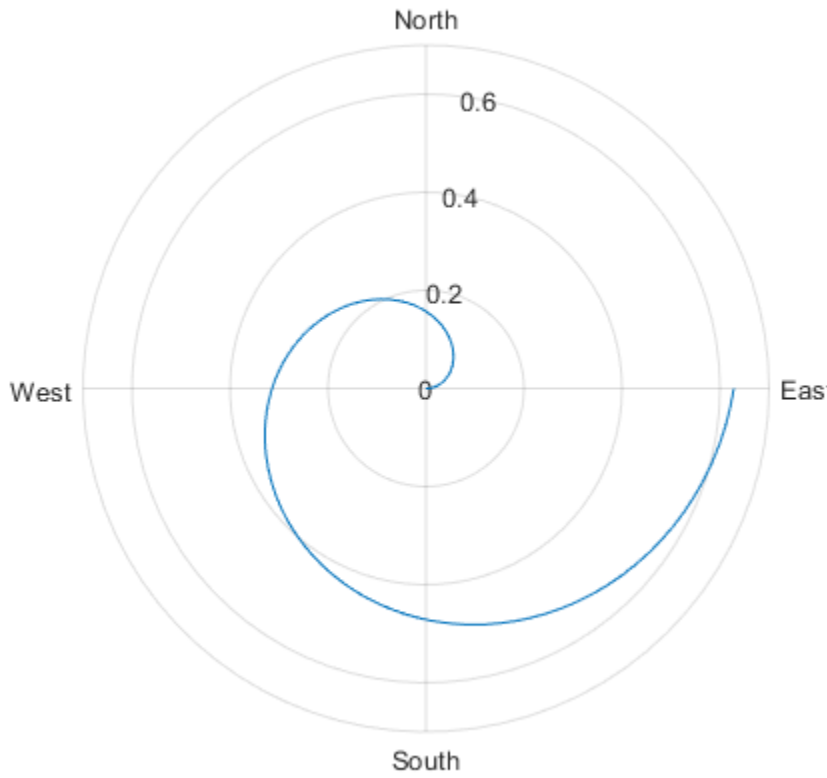
`___ = thetaticklabels(pax, ___ )` uses the axes specified by `pax` instead of the current axes. Specify `ax` as the first input argument for any of the previous syntaxes.

Examples

Specify theta-Axis Tick Values and Labels

Create a polar plot. Display tick marks and grid lines along the *theta*-axis at 0, 90, 180, and 270 degrees. Then, specify a label for each tick mark.

```
theta = linspace(0,2*pi);
rho = theta/10;
polarplot(theta,rho)

thetaticks([0 90 180 270])
thetaticklabels({'East','North','West','South'})
```

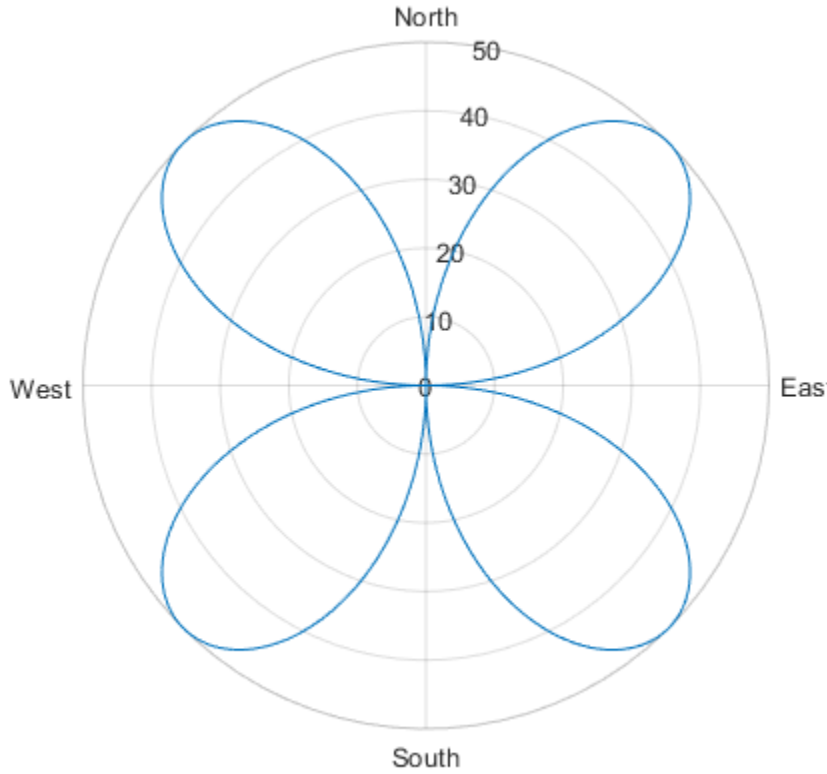


Set theta-Axis Tick Labels for Specific Polar Axes

Create a polar plot and assign the polar axes object to the variable `pax`. Then, specify the *theta*-axis tick values and labels for the polar axes. Ensure that the `thetaticks` and `thetaticklabels` functions affect `pax` by passing the polar axes object as the first input argument to the functions.

```
theta = 0:0.01:2*pi;
rho = 50*sin(2*theta);
polarplot(theta,rho)
pax = gca;
```

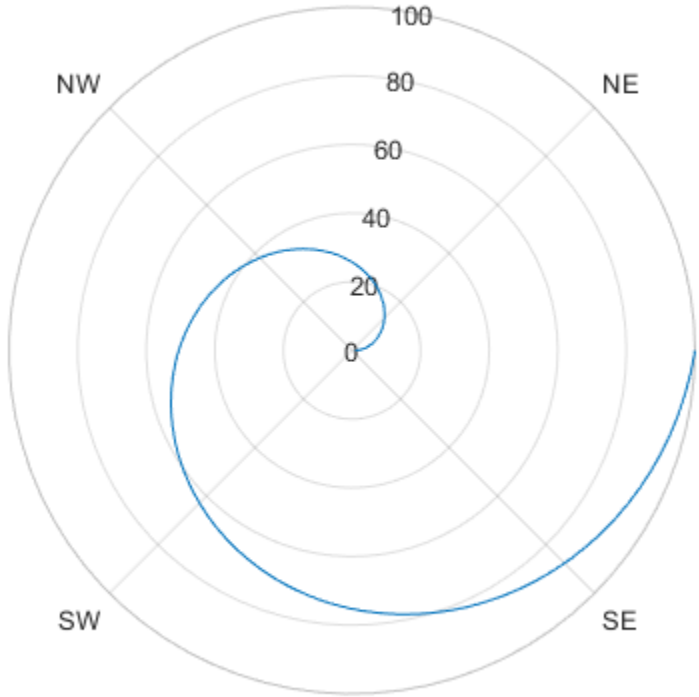
```
thetaticks(pax,[0 90 180 270])
thetaticklabels(pax,{'East','North','West','South'})
```



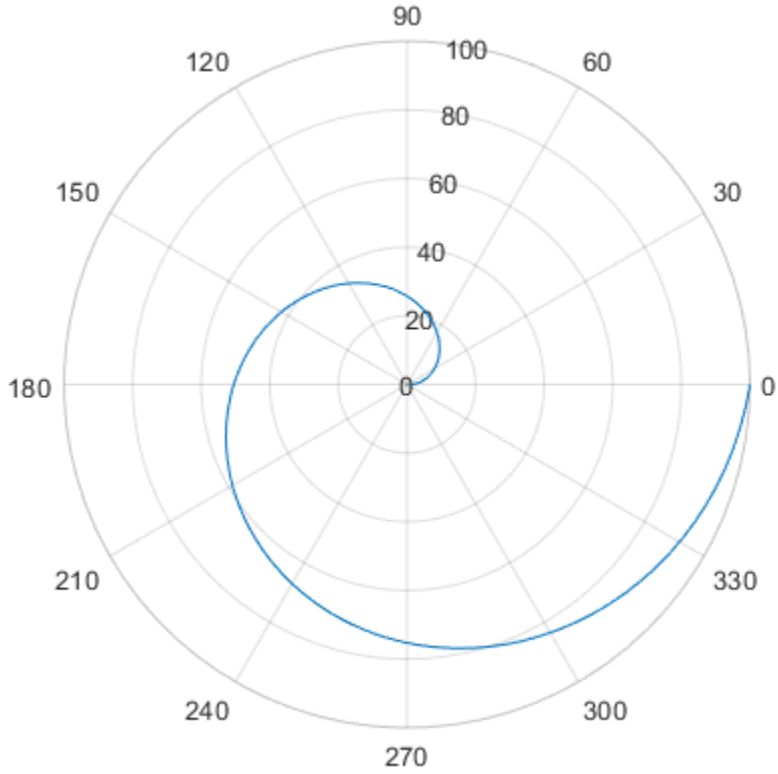
Set theta-Axis Tick Labels Back to Default Labels

Create a polar plot and specify the *theta*-axis tick values and corresponding labels. Then, set the *theta*-axis tick values and labels back to the default values.

```
polarplot(1:100)
thetaticks([45 135 225 315])
thetaticklabels({'NE','NW','SW','SE'})
```



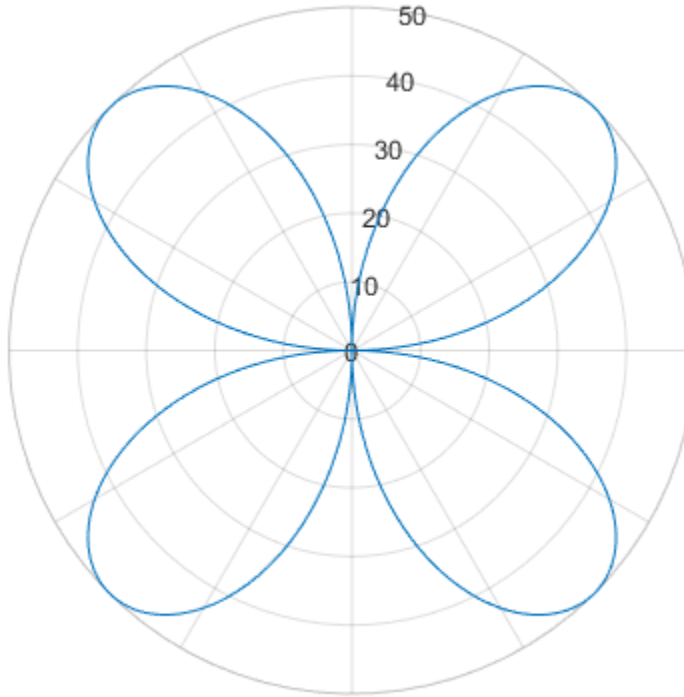

```
thetaticks('auto')
thetaticklabels('auto')
```



Remove theta-Axis Tick Labels

Remove the tick labels along the *theta*-axis by specifying the tick labels as an empty array.

```
theta = 0:0.01:2*pi;
rho = 50*sin(2*theta);
polarplot(theta,rho)
thetaticklabels({})
```



Input Arguments

`labels` — Tick labels

cell array of character vectors | string array | categorical array

Tick labels on page 1-15565, specified as a cell array of character vectors, string array, or categorical array. If you do not want tick labels to show, then specify an empty cell array `{}`. Tick labels support TeX and LaTeX markup. See the `TickLabelInterpreter` property of the axes object for more information.

Example: `thetaticklabels({'0','\pi','2\pi'})`

Example: `thetaticklabels({'January','Febrary','March'})`

Example: `thetaticklabels({})`

Note

- If you specify the tick labels as a categorical array, MATLAB uses the values in the array, not the categories.
- To specify the tick values, use the `thetaticks` function.

---

pax — One or more polar axes
current axes (default) | single object | vector of objects

One or more polar axes, specified as a single object or a vector of objects. If you do not specify the polar axes, then `thetaticklabels` uses the current axes.

Output Arguments

`tl` — Current tick labels
cell array of character vectors | character array

Current tick labels, returned as a cell array of character vectors or a character array.

`m` — Current tick labels mode
`'auto'` | `'manual'`

Current tick labels mode, returned as one of these values:

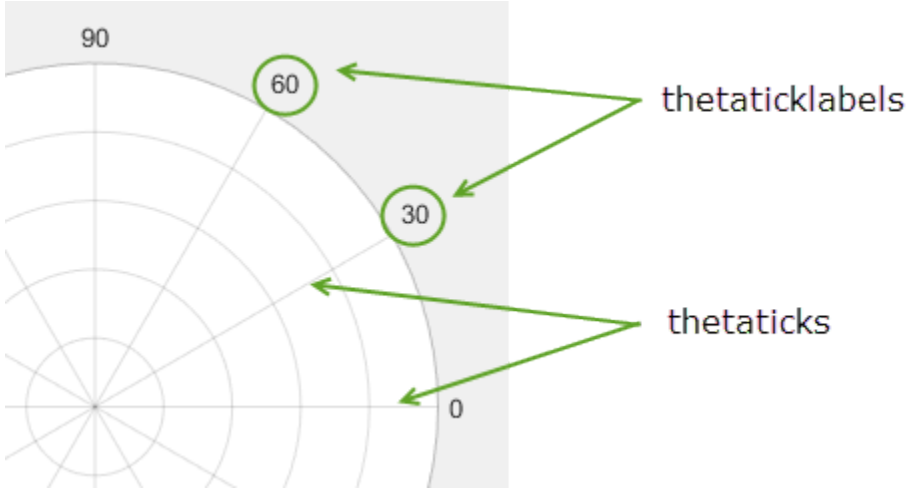
- `'auto'` — Axes automatically determines the *theta*-axis tick labels.
- `'manual'` — Axes uses manually specified *theta*-axis tick labels.

More About

Tick Labels

The tick labels are the labels that you see next to each tick mark. The tick values are the locations along the *theta*-axis where the tick marks appear. Set the tick values using the

`thetaticks` function. Set the corresponding tick labels using the `thetaticklabels` function.



Algorithms

The `thetaticklabels` function sets and queries several polar axes properties related to the *theta*-axis tick labels.

- `ThetaTickLabel` — Property that stores the text for the *theta*-axis tick labels.
- `ThetaTickLabelMode` — Property that stores the *theta*-axis tick label mode. When you set the *theta*-axis tick labels using `thetaticklabels`, this property changes to `'manual'`.
- `ThetaTickMode` — Property that stores the *theta*-axis tick value mode. When you set the *theta*-axis tick labels using `thetaticklabels`, this property changes to `'manual'`.

See Also

Functions

`rticklabels` | `thetalim` | `thetatickformat` | `thetaticks`

Properties
PolarAxes

Introduced in R2016b

thetaticks

Set or query *theta*-axis tick values

Syntax

```
thetaticks(ticks)
tv = thetaticks
```

```
thetaticks('auto')
thetaticks('manual')
m = thetaticks('mode')
```

```
___ = thetaticks(pax, ___ )
```

Description

`thetaticks(ticks)` sets the *theta*-axis tick values on page 1-15577, which are the locations along the *theta*-axis where the tick marks and grid lines appear. Specify `ticks` as a vector of increasing values; for example, `[0 90 180 270]`. This command affects the current axes.

`tv = thetaticks` returns the current *theta*-axis tick values as a vector.

`thetaticks('auto')` sets an automatic mode, enabling the axes to determine the *theta*-axis tick values. Use this option if you change the tick values and then want to set them back to the default values.

`thetaticks('manual')` sets a manual mode, freezing the *theta*-axis tick values at the current values. Use this option if you want to retain the current tick values when resizing the axes or adding new data to the axes.

`m = thetaticks('mode')` returns the current *theta*-axis tick labels mode, which is either `'auto'` or `'manual'`. By default, the mode is automatic unless you specify tick values or change the mode to manual.

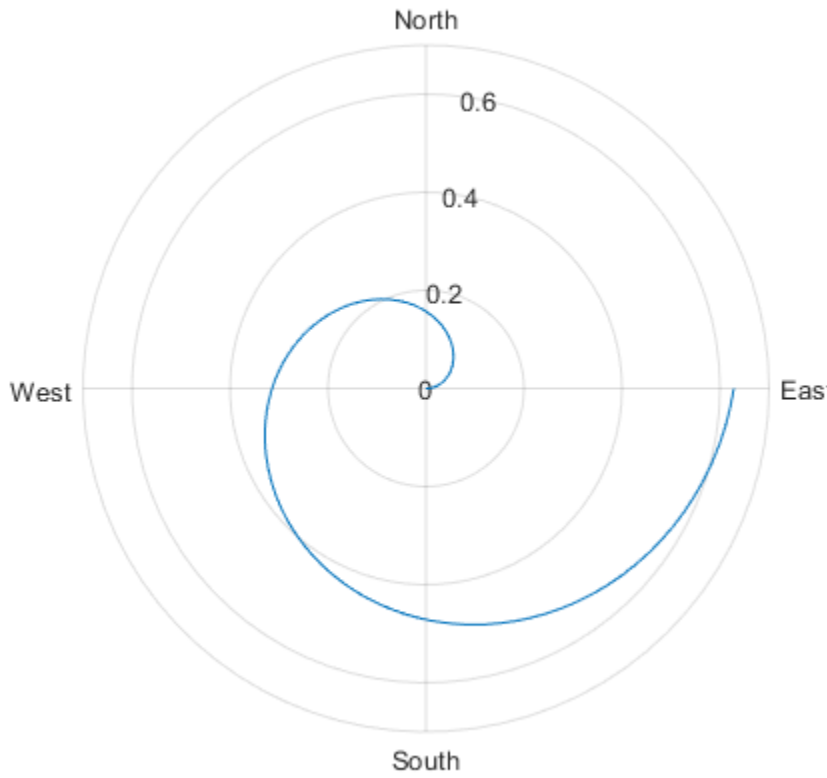
`___ = thetaticks(pax, ___ )` uses the axes specified by `pax` instead of the current axes. Specify `pax` as the first input argument for any of the previous syntaxes.

Examples

Specify theta-Axis Tick Values and Labels

Create a polar plot. Display tick marks and grid lines along the *theta*-axis at 0, 90, 180, and 270 degrees. Then, specify a label for each tick mark.

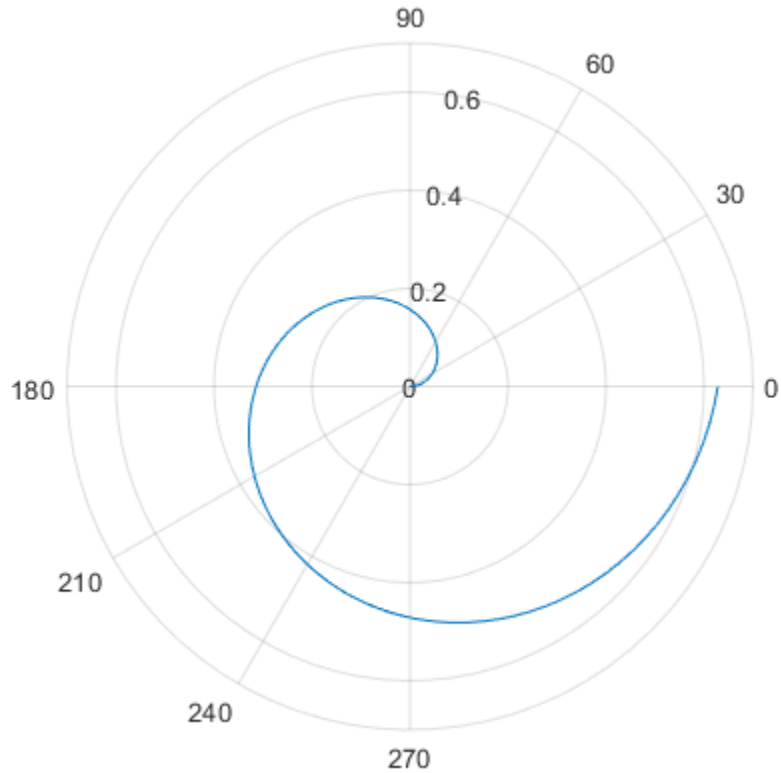
```
theta = linspace(0,2*pi);
rho = theta/10;
polarplot(theta,rho)

thetaticks([0 90 180 270])
thetaticklabels({'East','North','West','South'})
```



Specify Nonuniform theta-Axis Tick Values

Display tick marks along the *theta*-axis at nonuniform values between 0 and 360 degrees.

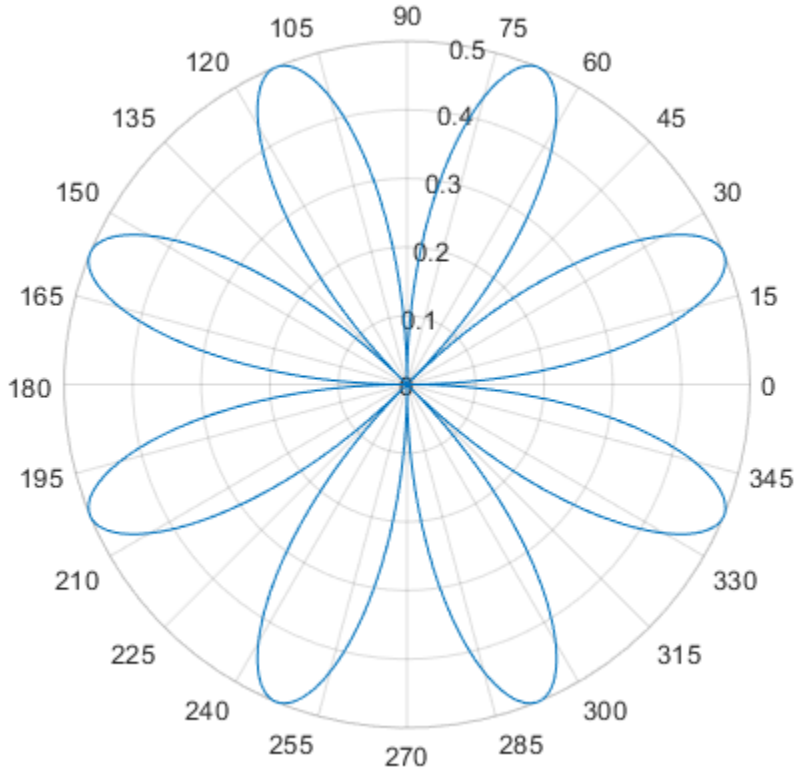
```
theta = linspace(0,2*pi);
rho = theta/10;
polarplot(theta,rho)
thetaticks([0 30 60 90 180 210 240 270])
```



Increment theta-Axis Values by 15 Degrees

Display tick marks along the *theta*-axis every 15 degrees.

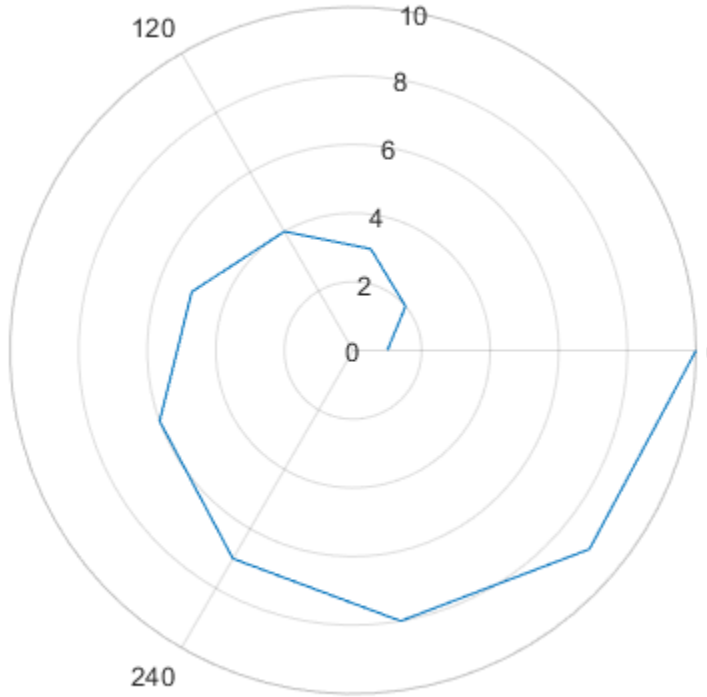
```
theta = 0:0.01:2*pi;
rho = sin(2*theta).*cos(2*theta);
polarplot(theta,rho)
thetaticks(0:15:360)
```



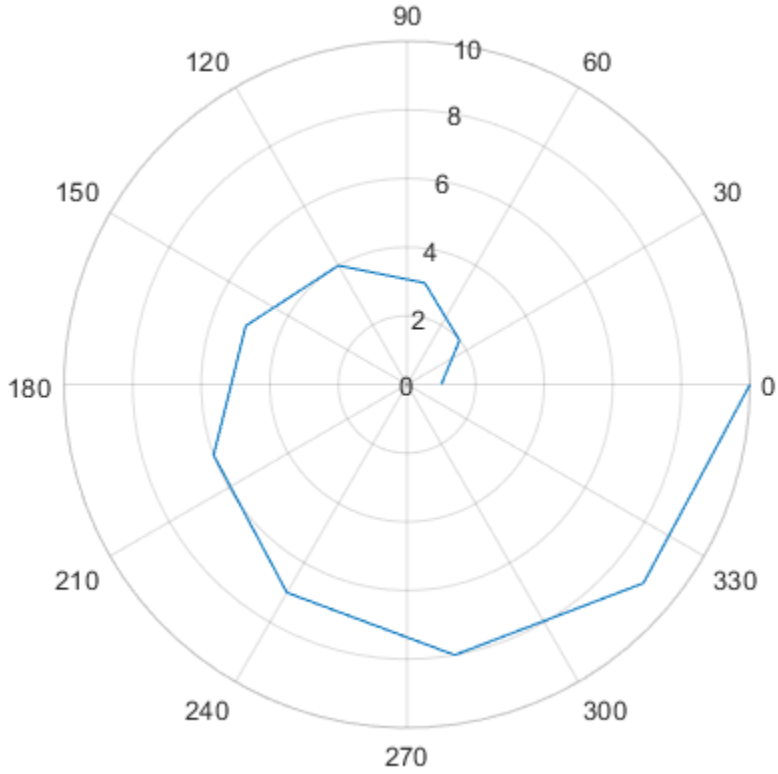
Set theta-Axis Tick Values Back to Default Values

Create a polar plot and specify the *theta*-axis tick values. Then, set the *theta*-axis tick values back to the default values.

```
polarplot(1:10)
thetaticks([0 120 240])
```



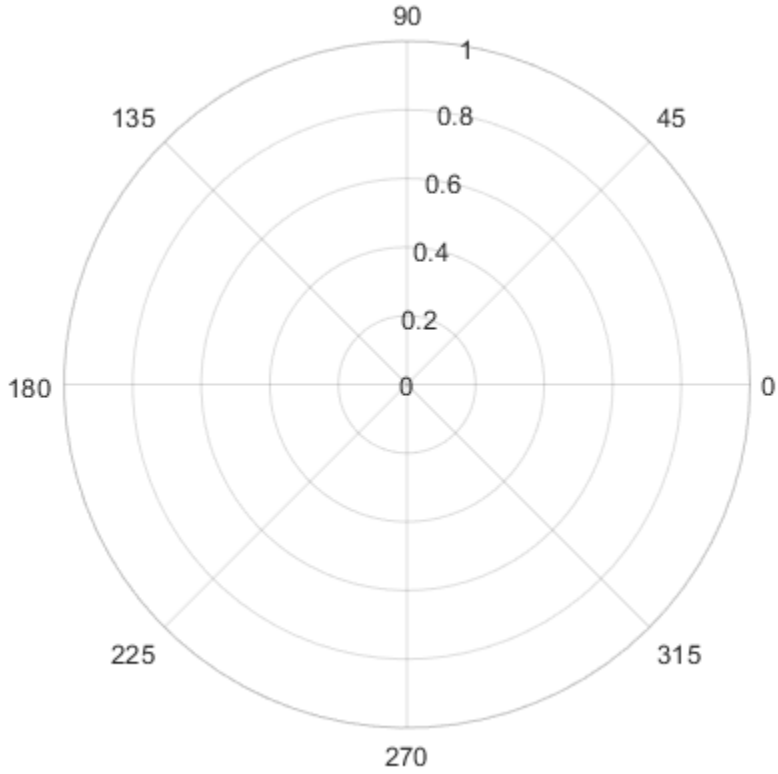

```
thetaticks('auto')
```



Set theta-Axis Tick Values for Specific Polar Axes

Create polar axes and return the polar axes object `pax`. Ensure that `thetaticks` affects the polar axes you just created by passing `pax` as the first input argument to the function.

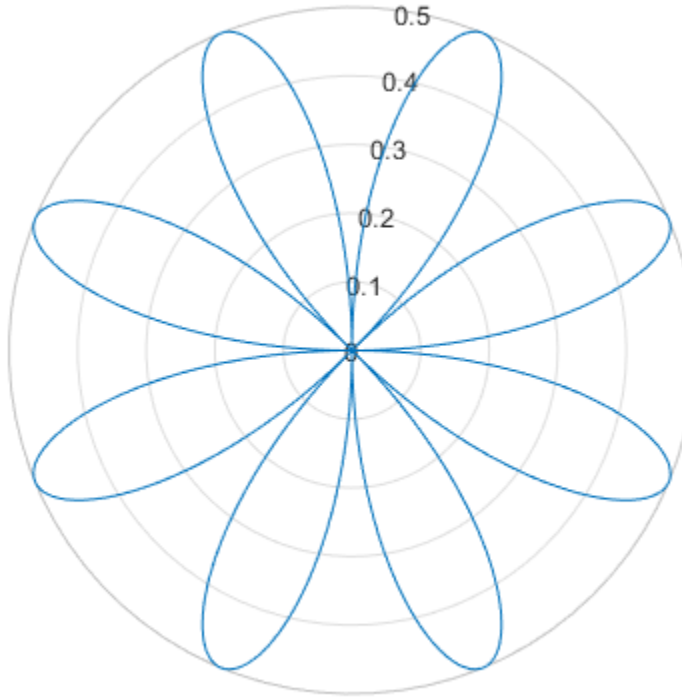
```
pax = polaraxes;
thetaticks(pax,0:45:360)
```



Remove theta-Axis Tick Marks and Grid Lines

Remove the tick marks and grid lines along the *theta*-axis by specifying the tick values as an empty array.

```
theta = 0:0.01:2*pi;
rho = sin(2*theta).*cos(2*theta);
polarplot(theta,rho)
thetaticks([])
```



Input Arguments

`ticks` — Tick values

vector of increasing values

Tick values on page 1-15577, specified as a vector of increasing values. The tick values are interpreted in units determined by the `ThetaAxisUnits` property. By default, the units are degrees.

If you do not want tick marks along the *theta*-axis, specify an empty vector `[]`.

Example: `thetaticks([0 90 180 270])`

Example: `thetaticks([0:30:330])`

Example: `thetaticks([])`

Note To specify the tick labels, use the `thetaticklabels` function.

pax — One or more polar axes
current axes (default) | single object | vector of objects

One or more polar axes, specified as a single object or a vector of objects. If you do not specify the polar axes, then `thetaticks` uses the current axes.

Output Arguments

`tv` — Current tick values
vector

Current tick values, returned as a vector.

`m` — Current mode
`'auto'` | `'manual'`

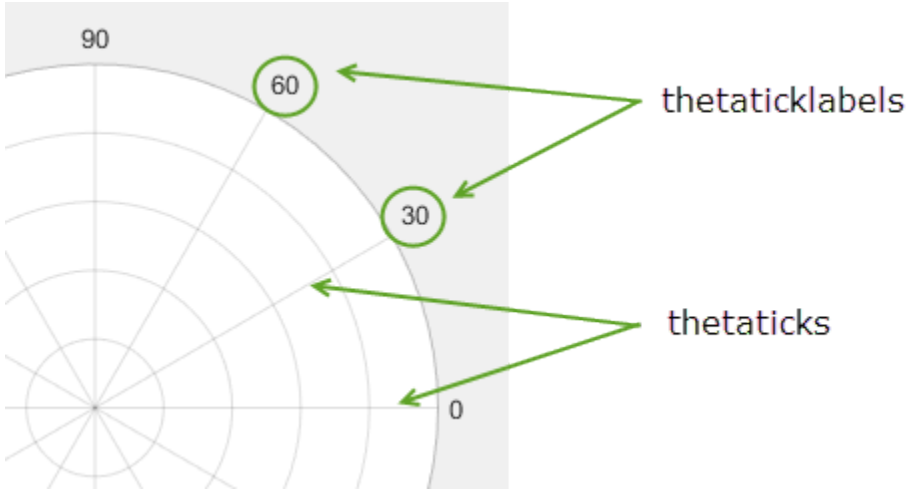
Current mode, returned as one of these values:

- `'auto'` — Axes automatically determines the *theta*-axis tick values.
- `'manual'` — Axes uses manually specified *theta*-axis tick values.

More About

Tick Values

The tick values are the locations along the *theta*-axis where the tick marks appear. The tick labels are the labels that you see next to each tick mark. Set the tick values using the `thetaticks` function. Set the corresponding tick labels using the `thetaticklabels` function.



Algorithms

The `thetaticks` function sets and queries several axes properties related to the *theta*-axis tick values.

- `ThetaTick` — Property that stores the *theta*-axis tick values.
- `ThetaTickMode` — Property that stores the *theta*-axis tick value mode. When you set the *theta*-axis tick values, this property changes to `'manual'`.

See Also

Functions
`rticklabels` | `rticks` | `thetalim` | `thetatickformat` | `thetaticklabels`

Properties
PolarAxes

Introduced in R2016b

thingSpeakRead

Read data stored in a ThingSpeak channel

Syntax

```
data = thingSpeakRead(channelID)
data = thingSpeakRead(channelID,Name,Value)
data = thingSpeakRead( ___ ,'ReadKey','channel Read API key')

[data,timestamps] = thingSpeakRead( ___ )

[data,timestamps,channelInfo] = thingSpeakRead( ___ )
```

Description

`data = thingSpeakRead(channelID)` reads the most recent data from all fields of the specified public channel on ThingSpeak.com and returns the data as a numeric type.

`data = thingSpeakRead(channelID,Name,Value)` uses additional options specified by one or more `Name`,`Value` pair arguments. To read nonnumeric data, you must specify the '`OutputFormat`'.

`data = thingSpeakRead( ___ ,'ReadKey','channel Read API key')` uses the ThingSpeak™ Read API key to read from a private channel.

`[data,timestamps] = thingSpeakRead( ___ )` also returns timestamps from the specified channel on ThingSpeak.com and can include any of the input arguments in previous syntaxes.

`[data,timestamps,channelInfo] = thingSpeakRead( ___ )` also returns channel information.

Examples

Retrieve Three Points of Data from a Channel

Retrieve the last five minutes of data from fields 1 and 4 of a public channel, and return the data in a timetable.

```
data = thingSpeakRead(12397,'Fields',[1,4],'NumPoints',3,'OutputFormat','TimeTable')
```

```
data =

  3×2 timetable

        Timestamps         WindDirectionNorth0Degrees    TemperatureF
    ___________________    __________________________    ____________

    15-Jan-2019 08:16:15               129                    26.1
    15-Jan-2019 08:17:16               115                    26.1
    15-Jan-2019 08:18:19               133                    26.2
```

Return Five Minutes of Data in a Timetable

Read the last five minutes of data from fields 1 and 4 of a public channel, and return the data in a table.

```
data = thingSpeakRead(12397,'Fields',[1,4],'Numminutes',5,'OutputFormat','TimeTable')
```

Retrieve Data Along with Timestamp

Retrieve the most recent result for all fields of a public channel, including the timestamps.

```
[data,timestamps] = thingSpeakRead(12397)
```

Return Five Minutes of Data Along with Timestamps and Channel Information

Retrieve the last five minutes of data from fields 1 and 4 of a public channel, including the timestamp and channel information.

```
[data,timestamps,channelInfo] = thingSpeakRead(12397,'Fields',[1,4],'NumMinutes',5)
```

Retrieve a Specific Date Range of Data with Timestamps and Channel Information

Retrieve the data from 11:59:01 on August 9, 2018 through 12:02:52 on August 10, 2018 for fields 1 and 4 of a public channel, including the timestamp and channel information.

```
[data,timestamps,channelInfo] = thingSpeakRead(12397,'Fields',[1 4],'DateRange',[datet
```

Input Arguments

`channelID` — Channel identification number
numeric

The channel identification number, specified as a positive integer. For private channels, you must also specify the ‘`ReadKey`’ argument.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `thingSpeakRead(12397,'Location',true);`

`DateRange` — Range of data to return
datetime vector

Range of data to return, specified as a comma-separated pair consisting of `'DateRange'` and an array of values that have `[startdate,enddate]` in MATLAB `datetime` values. ThingSpeak server limits the number of points returned to a maximum of 8000. Adjust your ranges or make multiple calls if you need more than 8000 points of data.

Note You cannot use `DateRange` with `NumDays` or `NumMinutes`.

Example: `thingSpeakRead(12397,'DateRange',[datetime('Aug 8, 2014'),datetime('Aug 12, 2014')]);`

`Fields` — Channel fields
positive numeric scalar

Channel field IDs, specified as a comma-separated pair consisting of `'Fields'` and numeric values of the channel field ID to retrieve data from.

Example: `thingSpeakRead(12397,'Fields',[1,3,4]);`

`Location` — Positional information of data
logical

Indicator to display positional information of data from the channel, specified as a comma-separated pair consisting of `'Location'` and a logical value. Location information includes latitude, longitude, and altitude.

Example: `thingSpeakRead(12397,'Location',true);`

`NumDays` — Number of days of data
positive numeric scalar

Number of days of data to extract data from, specified as a comma-separated pair consisting of `'NumDays'` and a numeric value. Number of days is measured in 24-hour periods. The ThingSpeak server limits the number of points returned to a maximum of 8000. Adjust your number of days, or make multiple calls if you need more than 8000 points of data.

Note You cannot use `NumDays` with `DateRange` or `NumMinutes`.

Example: `thingSpeakRead(12397,'NumDays',2);`

`NumMinutes` — Number of minutes of data
positive numeric scalar

Number of minutes of data to extract data from, specified as a comma-separated pair consisting of `'NumMinutes'` and a numeric value. Number of minutes of data to retrieve

is measured from the time the command is issued. The ThingSpeak server limits the number of points returned to a maximum of 8000. Adjust your number of minutes, or make multiple calls if you need more than 8000 points of data.

Note You cannot use `NumMinutes` with `DateRange` or `NumDays`.

Example: `thingSpeakRead(12397,'NumMinutes',20);`

Data Types: `double`

`NumPoints` — Number of data points
positive numeric scalar

Number of data points to extract, specified as a comma-separated pair consisting of `'NumPoints'` and a numeric value. The number of points or rows of data is counted from the time the command is issued. The ThingSpeak server limits the number of points returned to a maximum of 8000.

Example: `thingSpeakRead(12397,'NumPoints',200);`

`OutputFormat` — Class of the output data
matrix (default) | table | timetable

Class of the output data, specified as a comma-separated pair consisting of `'OutputFormat'` and a text value. Valid values are `'matrix'`, `'table'`, and `'timetable'`. Use `'matrix'` format for numeric data. You can use `'timetable'` or `'table'` for any nonnumeric data. The `'table'` format outputs `[data,channelInfo]`, where `data` contains the timestamps and the data from the fields of the channel. The `'timetable'` format outputs `[data,channelInfo]`, where `data` is a timetable and contains as many variables as the number of requested `'fields'`.

Example: `thingSpeakRead(12397,'OutputFormat','table');`

Example: `thingSpeakRead(12397,'OutputFormat','timetable');`

ReadKey — Read API key of channel
character vector of API Key | string scalar of API Key

Read API key of the channel, specified as a comma-separated pair consisting of `'ReadKey'` and a character vector representing the channel read API key. The Read API key allows you to read data from a private channel. If you are reading data from a public

channel, you do not need a Read API key. Save your channel Read API key in a variable for convenience.

Example: `thingSpeakRead(12397,'ReadKey','F6CSCVKX42WFZN9Y');`

Data Types: `char`

`Timeout` — Server connection timeout period
10 (default) | positive numeric scalar

Server connection timeout period, specified as a comma-separated pair consisting of `'Timeout'` and a numeric value. The value is the number of seconds allowed for `thingSpeakRead` to connect to the server.

Example: `thingSpeakRead(12397,'Timeout',15);`

Output Arguments

`data` — Data from the channel
array (default) | table | timetable

Data from the channel, returned as an array, table, or timetable.

`timestamps` — Timestamp of each data element
array (default)

Timestamp of each data element, returned as an array of `datetime` values.

Data Types: `datetime`

`channelInfo` — Channel information
structure

Channel information, returned as a structure with fields in the order shown in the table..

Field	Description
`ChannelID`	Channel identification number
`Name`	Name of the channel
`Description`	Channel description

Field	Description
`Latitude`	Latitude for the channel as provided in the channel settings
`Longitude`	Longitude for the channel as provided in the channel settings
`Altitude`	Elevation for the channel as provided in the channel settings
`Created`	Datetime of channel creation date
`Updated`	Datetime when the channels settings were last changed
`Last EntryID`	Entry ID for the most recent channel update
`Field Descriptions`	List of field names
`Field IDs`	List of enabled fields
`URL`	URL for feed data

More About

ThingSpeak

ThingSpeak is an IoT analytics platform service that allows you to aggregate, visualize, and analyze live data streams in the cloud. See License Options and the Commercial Use for more information.

See Also

Functions
`thingSpeakWrite` | `webread`

Introduced in R2019a

thingSpeakWrite

Write data to a ThingSpeak channel

Syntax

```
thingSpeakWrite(channelID,data,'WriteKey','channel Write API key')
thingSpeakWrite( ___ ,Name,Value)
response = thingSpeakWrite( ___ )
```

Description

`thingSpeakWrite(channelID,data,'WriteKey','channel Write API key')` writes the data to the specified channel. The Write API key is specified as a comma-separated pair consisting of `'WriteKey'` and a character vector or string representing the channel write key.

`thingSpeakWrite( ___ ,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`response = thingSpeakWrite( ___ )` returns the response provided by the ThingSpeak server on successful completion of the write operation.

Examples

Write Data to a ThingSpeak channel

Write a single numeric value to Field 1 of a channel.

```
thingSpeakWrite(17504,2.3,'WriteKey','23ZLGOBBU9TWHG2H')
```

Write Multiple Values and View Response

Write numeric values to the first four consecutive fields [1,2,3,4] of a channel.

```
response = thingSpeakWrite(17504,[2.3,1.2,3.2,0.1],'WriteKey','23ZLGOBBU9TWHG2H')
```

Write Nonnumeric Data

Write nonnumeric data to the first three consecutive fields [1,2,3] of a channel.

```
thingSpeakWrite(17504,{2.3,'on','good'},'WriteKey','23ZLGOBBU9TWHG2H')
```

Write Mixed Data

Write mixed values to nonconsecutive fields [1,4,6] of a channel.

```
thingSpeakWrite(17504,'Fields',[1,4,6],'Values',{2.3,'on','good'},'WriteKey','23ZLGOBBU
```

Write Field and Location Data

Update three fields, and write latitude, longitude, and altitude data for the entry.

```
thingSpeakWrite(17504,[1.1,2.3,4],'Location',[-40,23,35],'WriteKey','23ZLGOBBU9TWHG2H')
```

Write Location Data Only

Write latitude, longitude, and altitude data to a channel without adding values to fields.

```
thingSpeakWrite(17504,'Location',[-40,23,3500],'WriteKey','23ZLGOBBU9TWHG2H')
```

Write Data with Timestamp

Write a timestamp for the value being written to a channel. The timestamp provided is interpreted as local time.

```
tStamp = datetime('now')
thingSpeakWrite(17504,[2.3,1.2,3.2,0.1],'WriteKey','23ZLGOBBU9TWHG2H','TimeStamp',tStar
```

Write a Matrix of Data

Write a matrix of data to the first three fields of a channel. The timestamps provided are interpreted as local time.

```
% Generate Random Data
data = randi(10,10,3);

% Generate timestamps for the data
tStamps = datetime('now')-minutes(9):minutes(1):datetime('now');

channelID = 17504; % Change to your Channel ID
writeKey = '23ZLGOBBU9TWHG2H'; % Change to your Write API Key

% Write 10 values to each field of your channel along with timestamps
thingSpeakWrite(channelID,data,'TimeStamp',tStamps,'WriteKey',writeKey)
```

Write a TimeTable of Data

Write a timetable of data to the first two fields of a channel. The timestamps provided are interpreted as local time.

```
% Generate random data
dataField1 = randi(10,10,1);
dataField2 = randi(10,10,1);

% Generate timestamps for the data
tStamps = [datetime('now')-minutes(9):minutes(1):datetime('now')]';

% Create timetable
dataTable = timetable(tStamps,dataField1,dataField2);
```

```
channelID = 17504; % Change to your channel ID
writeKey  = '23ZLGOBBU9TWHG2H'; % Change to your Write API Key

% Write 10 values to each field of your channel along with timestamps
thingSpeakWrite(channelID,dataTable,'WriteKey',writeKey)
```

Input Arguments

`channelID` — Channel identification number
numeric

The channel identification number, specified as a positive integer.

`data` — Data to write to channel
array | numeric | table | timetable

Data to write to channel, specified as a numeric scalar or numeric vector, a cell array, a string of cell arrays, table, or a timetable. If you specify scalar data, the value is written to the first field in the channel (Field 1). For a vector or a 1-D cell array, data is written to consecutive fields starting with Field 1.

Example: `thingSpeakWrite(17504,{2,3,'on','good'},'WriteKey','channel write api key');`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `thingSpeakWrite(17504,[1.1,2.2],'Fields',[1,2,3],'WriteKey','channel write api key');`

`Fields` — Channel fields
numeric

Channel fields IDs, specified as a comma-separated pair consisting of `'Fields'` and a 1-by-n positive integer value.

Example: `thingSpeakWrite(17504,[2.3,1.2,3.2,0.1],'Fields',[1,2,3,4],'WriteKey','channel write api key');`

`Values` — Data values
numeric | array | table | timetable

Data values, specified as a comma-separated pair consisting of `'Values'` and numeric scalar, numeric vector, a cell array, a string of cell arrays, table, or timetable values. Specify the data to be written to channel fields specified by the `'Fields'` parameter.

Example: `thingSpeakWrite(17504,'Fields',[1,3,4],'Values',[1,2,3],'WriteKey','channel write api key');`

`Location` — Positional information of data
numeric

Positional information of data in the channel, specified as a comma-separated pair consisting of `'Location'` and numeric array. Location information includes latitude, longitude, and altitude. This example writes the same location information for each of the three points provided to field 1.

Example: `thingSpeakWrite(17504,[2.3,1.2,3.2,0.1],'Location',[-40,23,200],'WriteKey',' write api key');`

`Timeout` — Server connection timeout period
10 (default) | numeric

Server connection timeout period, specified as a comma-separated pair consisting of `'TimeOut'` and a numeric value. The timeout period is the number of seconds allowed for `thingSpeakRead` to connect to the server before the request is terminated.

Example: `thingSpeakWrite(17504,[2.3,1.2,3.2,0.1],'WriteKey','channel Write API key','Timeout',15);`

`Timestamp` — Timestamp of data values
datetime

Timestamp of data values written to fields in the write channel, specified as a comma-separated pair consisting of `'Timestamp'` and a value. The value must be a MATLAB `datetime` value. All timestamps must be unique. If you submit duplicate timestamps, all your updates are rejected, otherwise only updates with timestamps already in channel are rejected.

Note Do not specify `'Timestamp'` if you have specified `'Values'` as a timetable.

Example: `thingSpeakWrite(17504,[2.3,1.2,3.2,0.1],'WriteKey','channel write api key','Timestamp',[datetime('2/6/2018 9:27:12','format','MM/dd/uuuu HH:mm:ss')]);`

Data Types: `datetime`

Output Arguments

`response` — Channel response of the write operation
structure

Channel response of the write operation, returned as a structure with fields in the order shown in the table. The fields of the structure indicate the values written to the fields of the channel:

Field	Description
`FieldX`	FieldX data for the entry, up to eight fields
`Latitude`	Latitude for the write operation
`Longitude`	Longitude for the write operation
`Channel ID`	Channel Identification number
`Created`	Datetime of write operation and feed entry
`Last EntryID`	Entry ID for the most recent channel update
`Created`	Datetime of channel creation date
`Updated`	Datetime when the channels settings were last changed
`Last EntryID`	Entry ID for the most recent channel update
`Altitude`	Elevation for the write operation

Limitations

- All timestamps must be unique. You cannot write new data with timestamps that match existing data in the channel.
- The allowed update frequency for a channel is limited based on your license type. See Frequently Asked Questions and How to Buy for specific information. Updates with multiple entries, such as in a table or timetable, are limited to the slower update frequency. For users of free accounts, the number of messages in a single call to `thingSpeakWrite` is limited to 960 messages. For users of paid accounts, the limit is 14,400 messages. Attempts to publish at a rate faster than the allowed limit will return the following error: Requests are too frequent. For further information, see Limitations in the documentation.

More About

ThingSpeak

ThingSpeak is an IoT analytics platform service that allows you to aggregate, visualize, and analyze live data streams in the cloud. See License Options and the Commercial Use for more information.

See Also

Functions
`thingSpeakRead` | `webwrite`

Introduced in R2019a

tic

Start stopwatch timer

Syntax

```
tic
timerVal = tic
```

Description

`tic` starts a stopwatch timer to measure performance. The function records the internal time at execution of the `tic` command. Display the elapsed time with the `toc` function.

`timerVal = tic` returns the value of the internal timer at the execution of the `tic` command, so that you can record time for simultaneous time spans.

Output Arguments

`timerVal`

Value of the internal timer at the execution of the `tic` command. This value is used as an input argument for a subsequent call to `toc`. You should not rely on the meaning of this value.

Examples

Measure time to generate two random matrices and compute element-by-element multiplication of their transposes.

```
tic
A = rand(12000, 4400);
B = rand(12000, 4400);
toc
```

```
C = A'.*B';
toc
```

Measure how the time required to solve a linear system varies with the order of a matrix:

```
t = zeros(1,100);
for n = 1:100
    A = rand(n,n);
    b = rand(n,1);
    tic;
    x = A\b;
    t(n) = toc;
end
plot(t)
```

Measure multiple time spans simultaneously using two pairs of `tic/toc` calls. To do this, measure the minimum and average time to compute a summation of Bessel functions:

```
REPS = 1000;   minTime = Inf;   nsum = 10;
tic;  % TIC, pair 1

for i=1:REPS
   tStart = tic;  % TIC, pair 2
   total = 0;
   for j=1:nsum
      total = total + besselj(j,REPS);
   end

   tElapsed = toc(tStart);  % TOC, pair 2
   minTime = min(tElapsed, minTime);
end
averageTime = toc/REPS;  % TOC, pair 1
```

Tips

- Consecutive `tic` commands overwrite the internally recorded starting time.
- The `clear` function does not reset the starting time recorded by a `tic` command.
- The following actions result in unexpected output:
  - Using `tic` and `toc` to time `timeit`
  - Using `tic` and `toc` within a function timed by `timeit`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The data type returned from `tic` is different between MATLAB and the generated code. To avoid errors, do not use a MEX-function `tic` output value in MATLAB, or a MATLAB `tic` output value in a MEX function.
- If you call `tic` in a MATLAB session and `toc` in a MEX function, or vice versa, the timing results are not coordinated.
- The C/C++ implementation for `tic` in the generated code differs depending on the hardware settings stored in the code generation configuration object. By default, the hardware settings are configured for the host platform, with `Hardware Board` in the MATLAB Coder app set to `MATLAB Host Computer`.
  - When generating code on Windows with Windows-compatible hardware settings, the generated C/C++ implementation uses the Windows API functions `QueryPerformanceFrequency` and `QueryPerformanceCounter`.
  - In all other cases, the implementation uses the POSIX API `clock_gettime`. When compiling code that uses the POSIX API, the preprocessor macro `_POSIX_C_SOURCE` must be set to an integer greater than or equal to `199309L`. The code generator sets the macro to `199309L` for compilation.

See Also

`clock` | `cputime` | `etime` | `profile` | `timeit` | `toc`

Introduced before R2006a

Tiff

MATLAB Gateway to LibTIFF library routines

Description

A `Tiff` object represents a connection to a Tagged Image File Format (TIFF) file and provides access to many of the functions of the LibTIFF library. `Tiff` offers more capabilities than the `imread` and `imwrite` functions, such as reading subimages, writing tiles and strips of image data, and modifying individual TIFF tags.

In most cases, the syntax of the `Tiff` object function is similar to the syntax of the corresponding LibTIFF library function. To fully understand the capabilities of the `Tiff` object, refer to the LibTIFF API and the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

MATLAB supports LibTIFF version 4.0.10. For copyright information on the `LibTIFF` library, see the `libtiffcopyright.txt` file.

Creation

Syntax

```
obj = Tiff(filename)
obj = Tiff(filename,mode)
```

Description

`obj = Tiff(filename)` creates a `Tiff` object for read access to the TIFF file `filename`.

`obj = Tiff(filename,mode)` creates a `Tiff` object with the type of access to the TIFF file specified by `mode`.

Input Arguments

`filename` — Name of file
character vector | string scalar

Name of file, specified as a character vector or string scalar.

Example: `'myfile.tif'`

`mode` — File access type
`'r'` (default) | `'w'` | `'w8'` | `'a'` | `'r+'`

File access type, specified as one of these values.

Parameter	Description
`'r'`	Open file for reading (default).
`'w'`	Open file for writing; discard existing contents.
`'w8'`	Open file for writing a BigTIFF file; discard existing contents.
`'a'`	Open or create file for writing; append data to end of file.
`'r+'`	Open (do not create) file for reading and writing.

When you open a TIFF file for writing or appending, the `Tiff` object automatically creates an IFD in the file for writing subsequent data. This IFD has all the default values specified in TIFF Revision `6.0`.

Properties

TIFF files consist of image file directories (IFDs) that contain image data and associated tags. The tags contain image related information, such as the image width, the image height, and the number of samples. Each TIFF property is a structure that provides values for a tag. Set the tag values using the `setTag` function. For instance, create a file and specify the JPEG compression.

```
t = Tiff('myfile.tif','w');
setTag(t,'Compression',Tiff.Compression.JPEG);
```

When you create a file, before writing data to the file, you must set these tags:

ImageWidth	Compression
ImageLength	PlanarConfiguration
BitsPerSample	Photometric
SamplesPerPixel	

Depending on the layout of the image, you must set these additional tags:

- Stripped layout — Set the `RowsPerStrip` tag.
- Tiled layout — Set the `TileWidth` and `TileHeight` tags.

`Compression` — Scheme to compress image data
structure

Scheme to compress image data, stored in a structure with these fields.

Field Name
None
CCITTRLE (Read-only)
CCITTFax3
CCITTFax4
LZW
OJPEG
JPEG
AdobeDeflate
...

To see a full list of values that MATLAB supports for the `Compression` tag, type `Tiff.Compression` in the command window.

Example: `setTag(t,'Compression',Tiff.Compression.JPEG);`

`ExtraSamples` — Extra channel description
structure

Extra channel description, stored in a structure with these fields.

Field Name	Description
`Unspecified`	Unspecified data
`AssociatedAlpha`	Associated alpha (premultiplied)
`UnassociatedAlpha`	Unassociated alpha data

If extra channels exist in addition to the usual colorimetric channels, then the `ExtraSamples` tag is required. For an example usage, see “Write Tiff Image with Color and Alpha Channel Data” on page 1-15607.

Example: `setTag(t,'ExtraSamples',Tiff.ExtraSamples.AssociatedAlpha)`

`Group3Options` — Group 3 Fax compression options
structure

Group 3 Fax compression options, stored in a structure with these fields.

Field Name	Description
`Encoding2D`	Bit 0 is 1. This value specifies two-dimensional coding. If more than one strip is specified, each strip must begin with a one-dimensionally coded line. That is, `RowsPerStrip` must be a multiple of parameter `K`, as documented in the CCITT specification.
`Uncompressed`	Bit 1 is 1. This value specifies an uncompressed mode when encoding.
`FillBits`	Bit 2 is 1. Add fill bits as necessary before the EOL codes, such that EOL always ends on a byte boundary. This ensures that a zero nibble precedes an EOL sequence by 1 byte. For example, `xxxx-0000 0000-0001`.

This property also is referred to as `Fax3` or `T4Options`. The value of the property is a bit mask controlled by the first `3` bits.

Example: `setTag(t,'Group3Options',Tiff.Group3Options.Uncompressed);`

`InkSet` — Separated image ink set
structure

Separated image ink set, stored in a structure with these fields.

Field Name	Description
`CMYK`	Order of components: cyan, magenta, yellow, black. Usually, a value of 0 represents 0% ink coverage and a value of 255 represents 100% ink coverage for that component, but consult the TIFF specification for `DotRange`. When you specify CMYK, do not set the `InkNames` tag.
`MultiInk`	Any ordering other than CMYK. Consult the TIFF specification for `InkNames` field for a description of the inks used.

In the context of this property, separated refers to the photometric interpretation (not the planar configuration).

Example: `setTag(t,'InkSet',Tiff.InkSet.CMYK);`

`JPEGColorMode` — Color mode
structure

Color mode, stored in a structure with these fields.

Field Name	Description
`Raw` (default)	Keep input as separate Y, Cb, and Cr matrices.
`RGB`	Convert RGB input to YCbCr.

Do not use this property to read `YCbCr` imagery as `RGB`. Instead use the `RGBA` interface provided by the `readRGBAImage`, `readRGBAStrip`, and `readRGBATile` functions.

For an example, see “Create YCbCr/JPEG image from RGB data” on page 1-15608.

Example: `setTag(t,'JPEGColorMode',Tiff.JPEGColorMode.RGB);`

`Orientation` — Visual orientation of image data

structure

Visual orientation of the image data, stored in a structure with these fields.

Field Name
`TopLeft`
`TopRight`
`BottomRight`
`BottomLeft`
`LeftTop`
`RightTop`
`RightBottom`
`LeftBottom`

The `Orientation` property describes the image orientation with respect to rows and columns. For instance, when `Orientation` is set to `TopLeft`, then the first row represents the top of the image, and the first column represents the left side. The value specified in the `Orientation` is for informational purposes only, and it does not affect how MATLAB reads or writes the image data.

Example: `setTag(t,'Orientation',Tiff.Orientation.TopLeft);`

`Photometric` — Color space of image data

structure

Color space of image data, stored in a structure with these fields.

Field Name
`MinIsWhite`
`MinIsBlack`
`RGB`
`Palette`
`Mask`
`Separated (CMYK)`

Field Name
YCbCr
CIELab
ICCLab
ITULab
LogL
LogLUV
CFA
LinearRaw

Example: `setTag(t,'Photometric',Tiff.Photometric.RGB);`

`PlanarConfiguration` — Storage configuration
structure

Storage configuration of the image component values, stored in a structure with these fields.

Field Name	Description
Chunky	Store component values for each pixel contiguously. For example, in the case of RGB data, store the first three pixels in the file as RGBRGBRGB. Almost all TIFF images have contiguous planar configurations.
Separate	Store component values for each pixel separately. For example, in the case of RGB data, the red component is stored separately in the file from the green and blue components.

Example:
`setTag(t,'PlanarConfiguration',Tiff.PlanarConfiguration.Chunky);`

`ResolutionUnit` — Resolution units
structure

Resolution units to interpret the values contained in `XResolution` and `YResolution` tags, stored in a structure with these fields.

Field Name	Description
`None`	Default value.
`Inch`	Assign unit inches for values contained in `XResolution` and `YResolution` tags.
`Centimeter`	Assign unit centimeters for values contained in `XResolution` and `YResolution` tags.

For example, the following code sets the value of the image resolution in `X` and `Y` directions to `300` pixels per inch:

```
setTag(t,'ResolutionUnit',Tiff.ResolutionUnit.Inch);
setTag(t,'XResolution',300);
setTag(t,'YResolution',300);
```

Example: `setTag(t,'ResolutionUnit',Tiff.ResolutionUnit.Inch);`

`SampleFormat` — Pixel sample format
structure

Pixel sample format, stored in a structure with these fields.

Field Name	Description
`UInt` (default)	Unsigned integer data
`Int`	Two's complement signed integer data
`IEEEFP`	IEEE floating point data

MATLAB does not support the formats `Void`, `ComplexInt`, and `ComplexIEEEFP`.

Example: `setTag(t,'SampleFormat',Tiff.SampleFormat.IEEEFP);`

`SGILogDataFmt` — `SGIL` codec data format
structure

`SGIL` codec data format, stored in a structure with these fields.

Field Name	Description
`Float`	Single precision samples.
`Bits8`	`uint8` samples (read-only).

Setting the `SGILogDataFmt` to `Float` or `Bits8` implies a `SamplesPerPixel` value of `3` for `LogLuv` images and a value of `1` for `LogL` images.

You can set the `SGILogDataFmt` tag only once per instance for a `LogL/LogLuv` TIFF image object.

Use the following code to create a `Tiff` object, set the SGIL data format, and then read the image data.

```
tiffobj = Tiff('example.tif','r');
setDirectory(tiffobj,3); % image three is a LogLuv image
setTag(tiffobj,'SGILogDataFmt',Tiff.SGILogDataFmt.Float);
imdata = read(tiffobj);
```

Example: `setTag(t,'SGILogDataFmt',Tiff.SGILogDataFmt.Float);`

`SubFileType` — Type of image
structure

Type of the image, stored in a structure with these fields.

Field Name	Description
`Default`	Default value for single image file or first image.
`ReducedImage`	The image is a single image of a multiimage (or multipage) file.
`Page`	Unassociated alpha data.
`Mask`	The image is a transparency mask for another image in the file. The photometric interpretation value must be `Photometric.Mask`.

Example: `setTag(t,'SubFileType',Tiff.SubFileType.Mask);`

`TagID` — TIFF tag IDs
structure

TIFF tag IDs that MATLAB supports, stored in a structure with these fields.

Field Name	Value
`SubFileType`	254
`ImageWidth`	256
`ImageLength`	257
`BitsPerSample`	258
`Compression`	259
`Photometric`	262
`Thresholding`	263
`FillOrder`	266
`...`	`...`

For a complete list of tag names and their corresponding tag IDs, type `Tiff.TagID` in the Command Window.

Use this property to specify a tag when you are using the `setTag` function. For example, `Tiff.TagID.ImageWidth` returns the ID of the `ImageWidth` tag. To get a list of all supported tags, use the `Tiff.getTagNames` function.

Example: `setTag(t,Tiff.TagID.ImageWidth,300);`

`Thresholding` — Thresholding technique
structure

Thresholding technique, stored in a structure with these fields.

Field Name
`BiLevel`
`HalfTone`
`ErrorDiffuse`

Use this tag to specify which algorithm to use when converting pixels from grayscale to black and white.

Example: `setTag(t,'Thresholding',Tiff.Thresholding.HalfTone);`

`YCbCrPositioning` — Position of chrominance samples
structure

Position of chrominance samples relative to luminance samples, stored in a structure with these fields.

Field Name	Description
`Centered`	Specify for compatibility with industry standards, such as PostScript Level 2.
`Cosited`	Specify for compatibility with most digital video standards, such as CCIR Recommendation 601-1.

Example: `setTag(t,'YCbCrPositioning',Tiff.YCbCrPositioning.Centered);`

Object Functions

A TIFF file is made up of one or more image file directories (IFDs). An IFD contains image data and its associated metadata. IFDs can contain subIFDs, which also contain image data and metadata. When you open a TIFF file for reading, the `Tiff` object makes the first IFD in the file the current IFD. The Tiff object functions operate on the current IFD.

Read Image

read	Read entire TIFF image
readEncodedStrip	Read data from specified strip
readEncodedTile	Read data from specified tile
readRGBAImage	Read image using RGBA interface
readRGBAStrip	Read strip data using RGBA interface
readRGBATile	Read tile data using RGBA interface
close	Close Tiff object

Write Image

write	Write entire image
writeEncodedStrip	Write data to specified strip
writeEncodedTile	Write data to specified tile
close	Close Tiff object

Set or Get Tags

getTag	Value of specified tag
setTag	Set value of tag
Tiff.getTagNames	List of recognized TIFF tags

Change Image File Directory (IFD)

currentDirectory	Return index of current IFD
lastDirectory	Determine if current IFD is last in file
nextDirectory	Make next IFD the current IFD
setDirectory	Make specified IFD the current IFD
setSubDirectory	Make subIFD the current IFD
rewriteDirectory	Write modified metadata to existing IFD
writeDirectory	Create new IFD and make it current IFD

Get Tile, Strip, or Library Information

isTiled	Determine if image is tiled
computeTile	Index number of tile containing specified coordinates
numberOfTiles	Total number of tiles in image
computeStrip	Index number of strip containing specified coordinate
numberOfStrips	Total number of strips in image
getVersion	LibTIFF library version

Examples

Create New TIFF File Using Tiff object

Create a new file called `myfile.tif`. To run this example, you must have write access to the folder.

```
t = Tiff('myfile.tif','w');
```

Close the `Tiff` object.

```
close(t);
```

Write Tiff Image with Color and Alpha Channel Data

For a dataset with color and alpha channels, set the `Tiff` tags and then write the data to a file.

Create an array of data, `data`, that contains color channels and an alpha channel.

```
rgb = imread('example.tif');
numrows = size(rgb,1);
numcols = size(rgb,2);
alpha = 255*ones([numrows numcols], 'uint8');
data = cat(3,rgb,alpha);
```

Create a `Tiff` object.

```
t = Tiff('myfile.tif','w');
```

Set the Tiff tags and specify the value of the `ExtraSamples` tag because the data contains the alpha channel in addition to the color channels.

```
setTag(t,'Photometric',Tiff.Photometric.RGB);
setTag(t,'Compression',Tiff.Compression.None);
setTag(t,'BitsPerSample',8);
setTag(t,'SamplesPerPixel',4);
setTag(t,'SampleFormat',Tiff.SampleFormat.UInt);
setTag(t,'ExtraSamples',Tiff.ExtraSamples.Unspecified);
setTag(t,'ImageLength',numrows);
setTag(t,'ImageWidth',numcols);
setTag(t,'TileLength',32);
setTag(t,'TileWidth',32);
setTag(t,'PlanarConfiguration',Tiff.PlanarConfiguration.Chunky);
```

Write the data to the TIFF file and close the `Tiff` object.

```
write(t,data);
close(t);
```

Create YCbCr/JPEG image from RGB data

Write RGB image data to a TIFF file as an YCbCr/JPEG image.

Get RGB data.

```
rgb = imread('example.tif');
```

Create a `Tiff` object, `t`, and set the tags. Specify that the input data is RGB using the `JPEGColorMode` tag.

```
t = Tiff('myfile.tif','w');
setTag(t,'Photometric',Tiff.Photometric.YCbCr);
setTag(t,'Compression',Tiff.Compression.JPEG);
setTag(t,'YCbCrSubSampling',[2 2]);
setTag(t,'BitsPerSample',8);
setTag(t,'SamplesPerPixel',3);
setTag(t,'SampleFormat',Tiff.SampleFormat.UInt);
setTag(t,'ImageLength',size(rgb,1));
setTag(t,'ImageWidth',size(rgb,2));
setTag(t,'TileLength',32);
setTag(t,'TileWidth',32);
setTag(t,'PlanarConfiguration',Tiff.PlanarConfiguration.Chunky);
setTag(t,'JPEGColorMode',Tiff.JPEGColorMode.RGB);
setTag(t,'JPEGQuality',75);
```

Write the data to the TIFF file and close the `Tiff` object.

```
write(t,rgb);
close(t);
```

See Also

`imfinfo` | `imread` | `imwrite`

Topics

"Importing Images"
"Exporting to Images"

Introduced in R2009b

timeit

Measure time required to run function

Syntax

```
t = timeit(f)
t = timeit(f,numOutputs)
```

Description

`t = timeit(f)` measures the typical time (in seconds) required to run the function specified by the function handle `f`.

`t = timeit(f,numOutputs)` calls `f` with the desired number of outputs, `numOutputs`. By default, `timeit` calls the function `f` with one output (or no outputs, if the function does not return any outputs).

Examples

Determine Time to Obtain Current Date

Use `timeit` to time a function call to `date`. This example uses a handle to a function that accepts no input.

```
f = @date;
t = timeit(f)
```

```
t = 1.1061e-04
```

Determine Time to Compute Matrix Summation

Time the combination of several mathematical matrix operations: matrix transposition, element-by-element multiplication, and summation of columns.

```
A = rand(12000,400);
B = rand(400,12000);
f = @() sum(A.'.*B, 1);
timeit(f)
```

```
ans = 0.0865
```

Compare Time to Run svd with Multiple Outputs

Determine how long it takes to run `svd` with one output argument, `s = svd(X)`.

```
X = rand(100);
f = @() svd(X);
t1 = timeit(f)
```

```
t1 = 0.0026
```

Compare the results to `svd` with three output arguments, `[U,S,V] = svd(X)`.

```
t2 = timeit(f,3)
```

```
t2 = 0.0050
```

Compare Time to Execute Custom Preallocation to Calling `zeros`

Create a short function to allocate a matrix using nested loops. Preallocating an array using a nested loop is inefficient, but is shown here for illustrative purposes.

```
function mArr = preAllocFcn(x,y)
for m = 1:x
    for n = 1:y
        mArr(m,n) = 0;
    end
end
```

```
end
```

Compare the time to allocate zeros to a matrix using nested loops and using the `zeros` function.

```
x = 1000;
y = 500;
g = @() preAllocFcn(x,y);
h = @() zeros(x,y);
diffRunTime = timeit(g)-timeit(h)
```

```
diffRunTime =

    0.1584
```

Input Arguments

`f` — function to be measured
function handle

Function to be measured, specified as a function handle. `f` is either a handle to a function that takes no input, or a handle to an anonymous function with an empty argument list.

`numOutputs` — Number of desired outputs from `f`
integer

Number of desired outputs from `f`, specified as an integer. If the function specified by `f` has a variable number of outputs, `numOutputs` specifies which syntax `timeit` uses to call the function. For example, the `svd` function returns a single output, `s`, or three outputs, `[U,S,V]`. Set `numOutputs` to `1` to time the `s = svd(X)` syntax, or set it to `3` to time the `[U,S,V] = svd(X)` syntax.

Tips

- The following actions result in unexpected output:
  - Using `timeit` between `tic` and `toc`

- Using `timeit` to time a function that includes calls to `tic` and `toc`
- Using `timeit` recursively

Algorithms

`timeit` calls the specified function multiple times, and computes the median of the measurements.

See Also

`cputime` | `tic` | `toc`

Topics

"Create Function Handle"
Anonymous Functions
Analyzing Your Program's Performance
MATLAB Performance Measurement White Paper on MATLAB Central File Exchange

Introduced in R2013b

toc

Read elapsed time from stopwatch

Syntax

```
toc
elapsedTime = toc
toc(timerVal)
elapsedTime = toc(timerVal)
```

Description

`toc` reads the elapsed time from the stopwatch timer started by the `tic` function. The function reads the internal time at the execution of the `toc` command, and displays the elapsed time since the most recent call to the `tic` function that had no output, in seconds.

`elapsedTime = toc` returns the elapsed time in a variable.

`toc(timerVal)` displays the time elapsed since the `tic` command corresponding to `timerVal`.

`elapsedTime = toc(timerVal)` returns the elapsed time since the `tic` command corresponding to `timerVal`.

Input Arguments

`timerVal`

Value of the internal timer saved from a previous call to the `tic` command.

Output Arguments

elapsedTime

Scalar `double` representing the time elapsed between `tic` and `toc` commands, in seconds.

Examples

Measure time to generate two random matrices and compute element-by-element multiplication of their transposes.

```
tic
A = rand(12000, 4400);
B = rand(12000, 4400);
toc
C = A'.*B';
toc
```

Measure how the time required to solve a linear system varies with the order of a matrix:

```
t = zeros(1,100);
for n = 1:100
    A = rand(n,n);
    b = rand(n,1);
    tic;
    x = A\b;
    t(n) = toc;
end
plot(t)
```

Measure multiple time spans simultaneously using two pairs of `tic/toc` calls. To do this, measure the minimum and average time to compute a summation of Bessel functions:

```
REPS = 1000;   minTime = Inf;   nsum = 10;
tic;  % TIC, pair 1

for i=1:REPS
   tStart = tic;  % TIC, pair 2
   total = 0;
   for j=1:nsum
      total = total + besselj(j,REPS);
```

```
    end

    tElapsed = toc(tStart);  % TOC, pair 2
    minTime = min(tElapsed, minTime);
end
averageTime = toc/REPS;  % TOC, pair 1
```

Tips

- Consecutive calls to the `toc` function with no input return the elapsed time since the most recent `tic`. Therefore, you can take multiple measurements from a single point in time.

  Consecutive calls to the `toc` function with the same `timerVal` input return the elapsed time since the `tic` function call that corresponds to that input.
- The following actions result in unexpected output:
  - Using `tic` and `toc` to time `timeit`
  - Using `tic` and `toc` within a function timed by `timeit`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The data type returned from `toc` is different between MATLAB and the generated code. To avoid errors, do not use a MEX-function `toc` output value in MATLAB, or a MATLAB `toc` output value in a MEX function.
- If you call `tic` in a MATLAB session and `toc` in a MEX function, or vice versa, the timing results are not coordinated.
- The C/C++ implementation for `toc` in the generated code differs depending on the hardware settings stored in the code generation configuration object. By default, the hardware settings are configured for the host platform, with `Hardware Board` in the MATLAB Coder app set to `MATLAB Host Computer`.

- When generating code on Windows with Windows-compatible hardware settings, the generated C/C++ implementation uses the Windows API functions `QueryPerformanceFrequency` and `QueryPerformanceCounter`.
- In all other cases, the implementation uses the POSIX API `clock_gettime`. When compiling code that uses the POSIX API, the preprocessor macro `_POSIX_C_SOURCE` must be set to an integer greater than or equal to `199309L`. The code generator sets the macro to `199309L` for compilation.

See Also

`clock` | `cputime` | `etime` | `profile` | `tic` | `timeit`

Introduced before R2006a

time

Convert time of calendar duration to duration

Syntax

```
t = time(d)
```

Description

`t = time(d)` returns the time components of the calendar duration values in `d` as durations. The `t` output is the same size as `d`.

Examples

Convert Time of Calendar Durations to Durations

Create an array of calendar durations.

```
d = caldays(8:10) + hours(1.2345)
```

```
d = 1x3 calendarDuration array
    8d 1h 14m 4.2s    9d 1h 14m 4.2s   10d 1h 14m 4.2s
```

Convert the time components of the array to durations.

```
t = time(d)
```

```
t = 1x3 duration array
   01:14:04   01:14:04   01:14:04
```

View the data type of `t`.

```
whos t
```

```
  Name      Size            Bytes  Class       Attributes

  t         1x3                40  duration
```

Input Arguments

d — Input calendar duration
scalar | vector | matrix | multidimensional array

Input calendar duration, specified as a scalar, vector, matrix, or multidimensional `calendarDuration` array.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also
`caldays` | `calmonths` | `calquarters` | `calweeks` | `calyears` | `duration`

Introduced in R2014b

timeofday

Elapsed time since midnight for datetimes

Syntax

```
d = timeofday(t)
```

Description

`d = timeofday(t)` returns an array of durations equal to the elapsed time since midnight for each of the `datetime` values in `t`. The output argument, `d`, is equal to the result of `t - dateshift(t,'start','day')`, and is the same size as `t`.

- If you have `datetime` values with no time zone, then `d` also is equal to

  ```
  e = hours(t.Hour) + minutes(t.Minute) + seconds(t.Second)
  ```
- If a `datetime` value has its `TimeZone` property set to a time zone that does not observe Daylight Saving Time (DST), then `d` is equal to `e`.
- If a `datetime` value has its `TimeZone` property set to a time zone that observes DST, then `timeofday` accounts for the DST shift on days when the shift occurs. On those days, for times after the DST shift occurs, `d` differs from `e` by the amount of the shift.

Examples

Calculate Times of Day

Create a `datetime` array. Calculate the elapsed time since midnight for each `datetime` value.

```
t = datetime('now') + hours(1:3)
```

```
t = 1x3 datetime array
   26-Aug-2019 18:35:01   26-Aug-2019 19:35:01   26-Aug-2019 20:35:01
```

```
d = timeofday(t)
```

```
d = 1x3 duration array
   18:35:01   19:35:01   20:35:01
```

List the data type of `d`. The `timeofday` function returns a `duration` array, not a `datetime` array.

```
whos d
```

```
  Name      Size            Bytes  Class       Attributes

  d         1x3                40  duration
```

Calculate Elapsed Times on DST Shift

Calculate elapsed times since midnight on a day with a Daylight Saving Time (DST) shift.

Create a `datetime` array. Set its `TimeZone` property to a time zone that observes DST. Set the date to a date when a DST shift occurred.

```
tz = 'America/New_York';
fmt = 'dd-MMM-yyyy HH:mm:ss z';
t = datetime(2015,3,8,'TimeZone',tz,'Format',fmt) + hours(1:4)
```

```
t = 1x4 datetime array
Columns 1 through 2

   08-Mar-2015 01:00:00 EST   08-Mar-2015 03:00:00 EDT

Columns 3 through 4

   08-Mar-2015 04:00:00 EDT   08-Mar-2015 05:00:00 EDT
```

Calculate the elapsed times. The DST shift occurred at 02:00 on March 8, 2015 in this time zone. `timeofday` accounts for the shift for times at or after 02:00 on this date.

```
d = timeofday(t)
```

```
d = 1x4 duration array
   01:00:00   02:00:00   03:00:00   04:00:00
```

Set Times of Day for Datetimes

Set the times of day in a `datetime` array according to the times of day in another `datetime` array. There are two ways to set the times of day. Only the second method is correct across Daylight Saving Time (DST) shifts.

Create a `datetime` array. Each element has a different time component.

```
t1 = datetime(2015,3,7) + hours(1:4)
```

```
t1 = 1x4 datetime array
Columns 1 through 3

   07-Mar-2015 01:00:00   07-Mar-2015 02:00:00   07-Mar-2015 03:00:00

Column 4

   07-Mar-2015 04:00:00
```

Create a second `datetime` array. Each element has the same date and time components.

```
t2 = datetime(2015,3,[8 8 8 8])
```

```
t2 = 1x4 datetime array
   08-Mar-2015   08-Mar-2015   08-Mar-2015   08-Mar-2015
```

Set the times of day in `t2` according to the times of day in `t1`.

```
t2 = dateshift(t2,'start','day') + timeofday(t1)
```

```
t2 = 1x4 datetime array
Columns 1 through 3

   08-Mar-2015 01:00:00   08-Mar-2015 02:00:00   08-Mar-2015 03:00:00

Column 4
```

```
   08-Mar-2015 04:00:00
```

Create a `datetime` array with elements that have the `TimeZone` property set to `'America/New_York'`.

```
tz = 'America/New_York';
fmt = 'dd-MMM-yyyy HH:mm:ss z';
t3 = datetime(2015,3,8,'TimeZone',tz,'Format',fmt) + hours(1:4)
```

```
t3 = 1x4 datetime array
Columns 1 through 2

   08-Mar-2015 01:00:00 EST   08-Mar-2015 03:00:00 EDT

Columns 3 through 4

   08-Mar-2015 04:00:00 EDT   08-Mar-2015 05:00:00 EDT
```

Calculate the elapsed time since midnight. `timeofday` accounts for the DST shift that occurred on March 8, 2015.

```
d = timeofday(t3)
```

```
d = 1x4 duration array
   01:00:00   02:00:00   03:00:00   04:00:00
```

Set the times of day in `t4` according to times of day in `t1`. To set the times of day correctly regardless of the time zone or the day of year, use the `Hour`, `Minute`, and `Second` properties of `t1`.

```
t4 = datetime(2015,3,[8 8 8 8],'TimeZone',tz,'Format',fmt);
t4.Hour = t1.Hour;
t4.Minute = t1.Minute;
t4.Second = t1.Second;
t4
```

```
t4 = 1x4 datetime array
Columns 1 through 2

   08-Mar-2015 01:00:00 EST   08-Mar-2015 03:00:00 EDT
```

```
Columns 3 through 4

   08-Mar-2015 03:00:00 EDT   08-Mar-2015 04:00:00 EDT
```

In this time zone 2:00 a.m. Eastern Standard Time did not exist on March 8, 2015, because the DST shift occurred then. The second element of the result has a time component of 3:00 a.m. Eastern Daylight Time.

Input Arguments

t — Input date and time
`datetime` array

Input date and time, specified as a `datetime` array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`dateshift` | `duration` | `hms` | `hour` | `minute` | `second`

Introduced in R2014b

timer class

Create object to schedule execution of MATLAB commands

Description

Use a `timer` object to schedule the execution of MATLAB commands one or multiple times. If you schedule the timer to execute multiple times, you can define the time between executions and how to handle queuing conflicts.

The `timer` object uses callback functions to execute commands. Callback functions execute code during some event. For the `timer` object, you can specify the callback function as a function handle or as a character vector. If the callback function is a character vector, MATLAB evaluates it as executable code. The timer object supports callback functions when a timer starts (`StartFcn`), executes (`TimerFcn`), stops (`StopFcn`), or encounters an error (`ErrorFcn`).

Note The `timer` object is subject to the limitations of your hardware, operating system, and software. Avoid using timer objects for real-time applications.

Construction

`t = timer` creates an empty `timer` object to schedule execution of MATLAB commands. An error occurs if the timer starts and `TimerFcn` is not defined.

`t = timer(Name,Value)` creates a `timer` object with additional options that you specify using one or more `Name,Value` pair arguments.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The argument name, `Name`, corresponds to a `timer` property name. In the constructor, the property values are specified using `Name,Value` pair arguments.

`BusyMode`

Character vector or string scalar that indicates action taken when a timer has to execute `TimerFcn` before the completion of previous execution of the `TimerFcn`. When `Running='on'`, `BusyMode` is read only. This table summarizes the busy modes.

BusyMode Values	Behavior if Queue Empty	Behavior if Queue Not Empty	Notes
`'drop'`	Adds task to queue	Drops task	Possible skipping of `TimerFcn` calls
`'error'`	Adds task to queue	Completes task; throws error specified by `ErrorFcn`; stops timer	Stops timer after completing task in execution queue
`'queue'`	Adds task to queue	Waits for queue to clear, and then enters task in queue	Adjusts `Period` property to manage tasks in execution queue

See “Handling Timer Queuing Conflicts” for more information.

Default: `'drop'`

`ErrorFcn`

Character vector, string scalar, function handle, or cell array defining the function that the timer executes when an error occurs. If there is an error, this function executes, and then calls `StopFcn`.

- If you specify this property using a character vector or string scalar, when MATLAB executes the callback it evaluates the MATLAB code contained in the character vector.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function.

The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.

- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

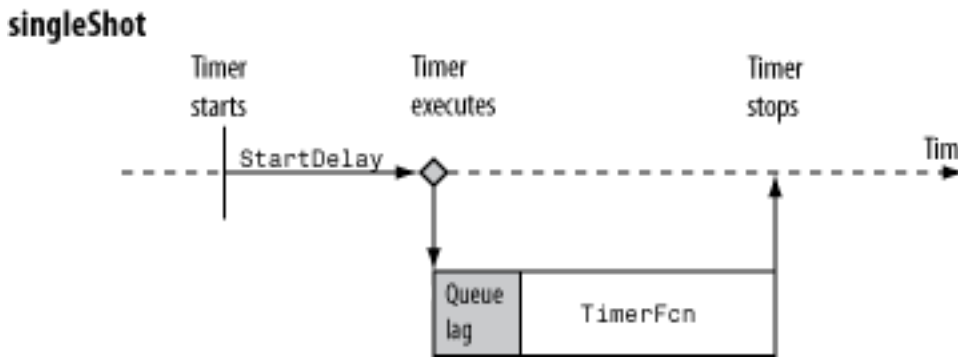
For more information, see “Timer Callback Functions”.

`ExecutionMode`

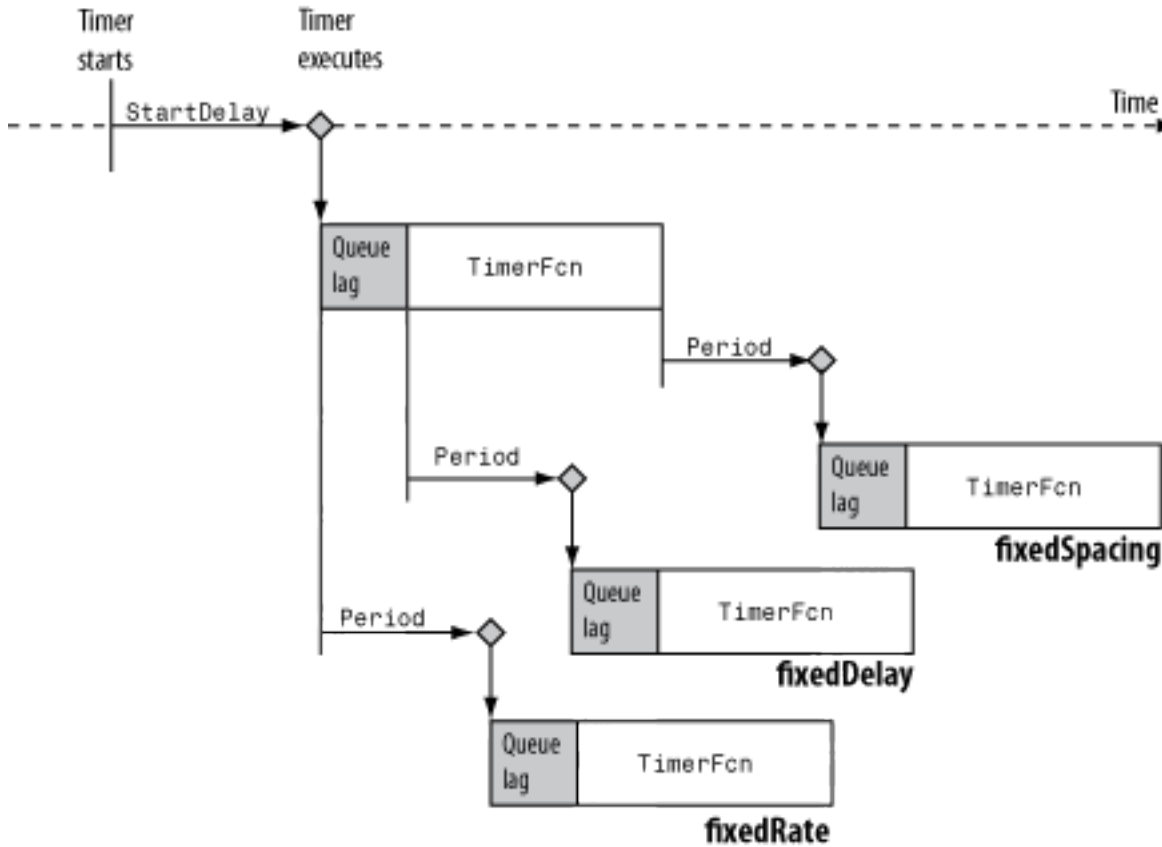
Character vector or string scalar that defines how the `timer` object schedules timer events. When `Running='on'`, `ExecutionMode` is read only. This table summarizes the execution modes.

Execution Mode	Time Period Start Point
`'singleShot'`	In this mode, the timer callback function is only executed once. Therefore, the `Period` property has no effect. This is the default execution mode.
`'fixedRate'`	Starts immediately after the timer callback function is added to the MATLAB execution queue
`'fixedDelay'`	Starts when the timer function callback restarts execution after a time lag due to delays in the MATLAB execution queue
`'fixedSpacing'`	Starts when the timer callback function finishes executing.

- `'singleShot'` is the single execution mode for the `timer` class, and is the default value.



- `'fixedDelay'`, `'fixedRate'`, and `'fixedSpacing'` are the three supported multiexecution modes. These modes define the starting point of the `Period` property. The `Period` property specifies the amount of time between executions, which remains the same. Only the point at which execution begins is different.



Default: `'singleShot'`

`Name`

Character vector or string scalar representing the `timer` name.

Default: `'timer-i'`, where *`i`* is a number indicating the *`i`*th timer object created this session. To reset *`i`* to 1, execute the `clear classes` command.

ObjectVisibility

Character vector or string scalar with possible values of `'on'` or `'off'`, that provides a way for you to discourage end-user access to the timer objects your application creates. The `timerfind` function does not return an object whose `ObjectVisibility` property is set to `'off'`. Objects that are not visible are still valid. To retrieve a list of all the timer objects in memory, including the invisible ones, use the `timerfindall` function.

Default: `'on'`

Period

Number greater than 0.001 that specifies the delay, in seconds, between executions of `TimerFcn`. For the timer to use `Period`, you must set `ExecutionMode` and `TasksToExecute` to schedule multiple timer object callback events.

Default: 1.0

StartDelay

Number greater than or equal to 0 that specifies the delay, in seconds, between the start of the timer and the first execution of the function specified in `TimerFcn`. When `Running = 'on'`, `StartDelay` is read only.

Default: 0

StartFcn

Character vector, string scalar, function handle, or cell array defining the function that executes when the timer starts.

- If you specify this property using a character vector or string scalar, when MATLAB executes the callback it evaluates the MATLAB code contained in the character vector.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see “Timer Callback Functions”.

StopFcn

Character vector, string scalar, function handle, or cell array defining the function that executes when the timer stops.

- If you specify this property using a character vector or string scalar, when MATLAB executes the callback it evaluates the MATLAB code contained in the character vector.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see “Timer Callback Functions”.

The timer stops when

- You call the timer `stop` method.
- The timer finishes executing `TimerFcn`. In other words, the value of `TasksExecuted` reaches the limit set by `TasksToExecute`.
- An error occurs. The `ErrorFcn` callback is called first, followed by the `StopFcn` callback.

You can use `StopFcn` to define clean up actions, such as deleting the timer object from memory.

Tag

Character vector or string scalar that represents a label for the object.

TasksToExecute

Number greater than 0, indicating the number of times the timer object is to execute the `TimerFcn` callback. Use the `TasksToExecute` property to set the number of executions. To use `TasksToExecute`, you must set `ExecutionMode` to schedule multiple timer callback events.

Default: `Inf`

TimerFcn

Character vector, string scalar, function handle, or cell array defining the timer callback function. You must define this property before you can start the timer.

- If you specify this property using a character vector or string scalar, when MATLAB executes the callback it evaluates the MATLAB code contained in the character vector.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see “Timer Callback Functions”.

UserData

Generic field for data that you want to add to the object.

Read-Only Properties

AveragePeriod

Average time in seconds between `TimerFcn` executions since the timer started. Value is `NaN` until timer executes two timer callbacks.

InstantPeriod

The time in seconds between the last two executions of `TimerFcn`. Value is `NaN` until timer executes two timer callbacks.

Running

Character vector defined as `'off'` or `'on'`, indicating whether the timer is currently executing callback functions.

TasksExecuted

The number of times the timer called `TimerFcn` since the timer started.

Type

Character vector that identifies the object type.

Methods

delete	Remove timer object from memory
get	Query property values for timer object
isvalid	Determine timer object validity
set	Set property values for timer object
start	Start timer object
startat	Schedule timer to fire at specified time
stop	Stop timer object
timerfind	Find timer object
timerfindall	Find timer object, regardless of visibility
wait	Block command prompt until timer stops running

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Display Message Using Timer

Display a message using an anonymous function as a callback function. It is important to note that the first two arguments the callback function passes are a handle to the timer object and an event structure. Even if the function doesn't use these arguments, the function definition requires them.

Wait 3 seconds, and then display the message ‘3 seconds have elapsed’.

```
t = timer;
t.StartDelay = 3;
```

```
t.TimerFcn = @(myTimerObj, thisEvent)disp('3 seconds have elapsed');
start(t)
```

```
3 seconds have elapsed
```

Suppose the function does not require the timer or event object. Use the tilde (~) operator to ignore the inputs.

```
t.TimerFcn = @(~,~) disp('3 seconds have elapsed');
start(t)
```

```
3 seconds have elapsed
```

Delete the timer object.

```
delete(t)
```

Execute Callback Function Multiple Times

Display the event and date/time output when the timer starts, fires, and stops. The timer callback function will be executed 3 times with 2 seconds between calls. The first two arguments the callback function passes are a handle to the timer object and an event structure. The event structure contains two fields: `Type` is a character vector that identifies the type of event that caused the callback, and `Data` is a structure that contains a date time vector of when the event occurred.

```
t = timer;
t.StartFcn = @(~,thisEvent)disp([thisEvent.Type ' executed '...
    datestr(thisEvent.Data.time,'dd-mmm-yyyy HH:MM:SS.FFF')]);
t.TimerFcn = @(~,thisEvent)disp([thisEvent.Type ' executed '...
     datestr(thisEvent.Data.time,'dd-mmm-yyyy HH:MM:SS.FFF')]);
t.StopFcn = @(~,thisEvent)disp([thisEvent.Type ' executed '...
    datestr(thisEvent.Data.time,'dd-mmm-yyyy HH:MM:SS.FFF')]);
t.Period = 2;
t.TasksToExecute = 3;
t.ExecutionMode = 'fixedRate';
start(t)
```

```
StartFcn executed 14-Mar-2013 09:08:50.865
TimerFcn executed 14-Mar-2013 09:08:50.865
TimerFcn executed 14-Mar-2013 09:08:52.865
TimerFcn executed 14-Mar-2013 09:08:54.866
StopFcn executed 14-Mar-2013 09:08:54.869
```

Delete the timer object.

```
delete(t)
```

Define Custom Callback Functions

Create a timer object to remind yourself to take 30-second ergonomic breaks every 10 minutes over the course of 8 hours.

Create a function in a file named `createErgoTimer.m` that returns a timer object. Have this file include three local functions to handle timer start, execute, and stop tasks.

```
function t = createErgoTimer()
secondsBreak = 30;
secondsBreakInterval = 600;
secondsPerHour = 60^2;
secondsWorkTime = 8*secondsPerHour;

t = timer;
t.UserData = secondsBreak;
t.StartFcn = @ergoTimerStart;
t.TimerFcn = @takeBreak;
t.StopFcn = @ergoTimerCleanup;
t.Period = secondsBreakInterval+secondsBreak;
t.StartDelay = t.Period-secondsBreak;
t.TasksToExecute = ceil(secondsWorkTime/t.Period);
t.ExecutionMode = 'fixedSpacing';
end
```

Using `StartDelay` allows the timer to start without directing you to take a break immediately. Set the execution mode to `'fixedSpacing'` so that `10` minutes and `30` seconds (`t.Period`) elapses after the completion of a `TimerFcn` execution. This allows you to stretch for `30` seconds before the start of the next `10` minute interval.

In the `createErgoTimer.m` file, add a local function to handle the tasks associated with starting the timer. By default, the `timer` object passes itself and event data to the callback function. The function disregards the event data.

```
function ergoTimerStart(mTimer,~)
secondsPerMinute = 60;
secondsPerHour = 60*secondsPerMinute;
str1 = 'Starting Ergonomic Break Timer.  ';
```

```
str2 = sprintf('For the next %d hours you will be notified',...
    round(mTimer.TasksToExecute*(mTimer.Period + ...
    mTimer.UserData)/secondsPerHour));
str3 = sprintf(' to take a %d second break every %d minutes.',...
    mTimer.UserData, (mTimer.Period - ...
    mTimer.UserData)/secondsPerMinute);
disp([str1 str2 str3])
end
```

Add a local function to handle the tasks associated with executing the timer. The `TimerFcn` callback should tell you to take a `30` second break.

```
function takeBreak(mTimer,~)
disp('Take a 30 second break.')
end
```

Add a local function to handle the tasks associated with stopping the timer.

```
function ergoTimerCleanup(mTimer,~)
disp('Stopping Ergonomic Break Timer.')
delete(mTimer)
end
```

Deleting the timer object removes it from memory.

From the command line, call the `createErgoTimer` function to create and start a timer.

```
t = createErgoTimer;
start(t)
```

```
Starting Ergonomic Break Timer.  For the next 8 hours you will be notified to take a 30 second break every 1
```

Every `10` minutes, you will be reminded to take a `30` second break.

```
Take a break.
```

You can leave the timer running for `8` hours or stop it manually. Recall that you included the task of deleting the timer from memory in the `StopFcn` callback.

```
stop(t)
```

```
Stopping Ergonomic Break Timer.
```

Tips

- To force the execution of the callback functions in the event queue, include a call to the `drawnow` function in your code. The `drawnow` function flushes the event queue.

See Also

Topics

Timer Callback Functions
"Handling Timer Queuing Conflicts"
"Ignore Function Outputs"
"Create Function Handle"
Property Attributes

Introduced before R2006a

delete

Class: `timer`

Remove timer object from memory

Syntax

```
delete(t)
```

Description

`delete(t)` removes the timer object, `t`, from memory. If `t` is an array of timer objects, `delete` removes all the objects from memory.

When you delete a timer object, it becomes invalid and you cannot reuse it. If multiple references to a timer object exist in the workspace, deleting the timer object invalidates the remaining references. To remove invalid timer objects references from the workspace, use the `clear` command.

Input Arguments

t

Object of class `timer`.

Examples

Delete a Timer

Create and start a timer that generates a 10-by-10 array of random numbers.

```
t = timer('TimerFcn','rand(10);');
start(t)
```

Delete the timer from memory.

```
delete(t)
```

Call the `whos` function to see if a reference still exists in the workspace.

```
whos
```

```
  Name       Size            Bytes  Class     Attributes

  ans       10x10              800  double
  t          1x1               104  timer
```

Try to restart the timer.

```
start(t)
```

```
Error using timer/start (line 27)
Invalid timer object. This object has been deleted and should be removed from your worl
```

The timer cannot be restarted.

Clear the timer object reference from the workspace.

```
clear t
```

Delete Multiple Timers Using timerfind

Use `delete` with the `timerfind` method to remove all visible timers from memory. This is an alternative to deleting individual timers by variable name.

Create and start three timers that compute the sine, cosine and tangent of `pi/4`.

```
t1 = timer('TimerFcn','sin(pi/4);');
t2 = timer('TimerFcn','cos(pi/4);');
t3 = timer('TimerFcn','tan(pi/4);');
```

Delete the timers from memory using `timerfind`. This removes all visible timer objects from memory.

```
delete(timerfind)
```

```
Warning: You are deleting one or more running timer objects.  MATLAB has automatically
```

Tips

- Use the `isvalid` method to determine if a timer object exists in memory, but is not cleared from the workspace.
- Use the `timerfind` and `timerfindall` methods to return timer objects currently existing in memory. This approach is useful if the reference to the timer object is cleared from the workspace (using the `clear` command), but not deleted from memory.

See Also

`isvalid` | `timer` | `timerfind` | `timerfindall`

Introduced before R2006a

get

Class: `timer`

Query property values for timer object

Syntax

```
get(t)
V = get(t)
V = get(t,propName)
```

Description

`get(t)` queries property values for timer object, `t`, and displays all property names and current values. `t` must be a scalar timer object.

`V = get(t)` queries property values for timer object, `t` and returns a structure, `V`, where each field name is the name of a property of `t` and each field contains the value of that property. If `t` is an M-by-1 vector of timer objects, `V` is an M-by-1 array of structures.

`V = get(t,propName)` returns the value, `V`, of the timer object property specified in `propName`. If `propNames` is a vector cell array of N property names, and `t` is a vector of M timer objects, `v` is an M-by-N cell array of property values.

Input Arguments

`t`

Object of class `timer`.

`propName`

Character vector or string scalar that specifies a `timer` property name.

Examples

Display All Properties of Timer

```
t = timer;
get(t)
```

```
       AveragePeriod: NaN
            BusyMode: 'drop'
            ErrorFcn: ''
       ExecutionMode: 'singleShot'
       InstantPeriod: NaN
                Name: 'timer-2'
    ObjectVisibility: 'on'
              Period: 1
             Running: 'off'
          StartDelay: 0
            StartFcn: ''
             StopFcn: ''
                 Tag: ''
       TasksExecuted: 0
      TasksToExecute: Inf
            TimerFcn: ''
                Type: 'timer'
            UserData: []
```

Delete the timer from memory.

```
delete(t)
```

Obtain Properties for Array of Timers

Create three timers.

```
t1 = timer;
t2 = timer;
t3 = timer;
```

Get properties of an array of timers.

```
V = get([t1,t2,t3])
```

```
V=3×18 struct
    BusyMode
    ErrorFcn
    ExecutionMode
    Name
    ObjectVisibility
    Period
    StartDelay
    StartFcn
    StopFcn
    Tag
    TasksToExecute
    TimerFcn
    UserData
    AveragePeriod
    InstantPeriod
    Running
    TasksExecuted
    Type
      ⋮
```

Delete the timers from memory.

```
delete([t1,t2,t3])
```

Obtain Single Property for Timer

Create a timer and determine if it is running.

```
t = timer;
get(t,'Running')
```

```
ans =
'off'
```

Delete the timer from memory.

```
delete(t)
```

Obtain Specified Properties for Array of Timers

Create three timers.

```
t1 = timer;
t2 = timer;
t3 = timer;
```

Obtain name, period, and running property values from the array of timers.

```
V = get([t1,t2,t3],{'Name','Running','Period'})
```

```
V=3×3 cell
    {'timer-2'}    {'off'}    {[1]}
    {'timer-3'}    {'off'}    {[1]}
    {'timer-4'}    {'off'}    {[1]}
```

Delete the timers

```
delete([t1,t2,t3])
```

Alternatives

You can also use dot notation can also be used to query timer object properties. For example, `t.Running` returns the same value as `get(t,'Running')`.

See Also

`set` | `timer`

Introduced before R2006a

isvalid

Class: `timer`

Determine timer object validity

Syntax

```
validCheck = isvalid(t)
```

Description

`validCheck = isvalid(t)` determines timer object validity and returns a logical array, `validCheck`, that contains a `0` where the elements of `t` are invalid timer objects and a `1` where the elements of `t` are valid timer objects.

An invalid timer object is an object that is deleted from memory using `delete` and cannot be reused. Use the `clear` command to remove an invalid timer object from the workspace.

Input Arguments

`t`

Object or array of objects of class `timer`

Examples

Check Timer Object Validity

Create a timer object. While this timer object is valid, it cannot start because a `TimerFcn` is not defined.

```
t = timer;
out = isvalid(t)
```

```
out =

     1
```

Delete the timer object, making it invalid.

```
delete(t)
out1 = isvalid(t)
```

```
out1 =

     0
```

Tips

- To return timer objects existing in memory, use the `timerfind` and `timerfindall` methods. This practice is useful if the reference to the timer object is cleared from the workspace (using the `clear` function), but the object has not been deleted from memory.

See Also

`delete` | `timer` | `timerfind` | `timerfindall`

Introduced before R2006a

set

Class: `timer`

Set property values for timer object

Syntax

```
set(t)
propStruct = set(t)

set(t,Name)
propCell = set(t,Name)

set(t,Name,Value)
set(t,S)
set(t,PN,PV)
```

Description

`set(t)` displays the property names and their possible values for all configurable properties of timer object `t`.

`propStruct = set(t)` returns the values in a `struct`.

`set(t,Name)` displays the possible values for the specified property, `Name`, of timer object, `t`.

`propCell = set(t,Name)` returns the values in a cell.

`set(t,Name,Value)` sets the properties specified by one or more `Name,Value` pair arguments. `t` can be a single timer object or a vector of timer objects, in which case `set` configures the property values for all timer objects, `t`.

`set(t,S)` configures the properties of `t`, with the values specified in `S`, where `S` is a structure whose field names are object property names.

`set(t,PN,PV)` configures the properties specified in the cell array of character vectors, `PN`, to the corresponding values in the cell array `PV`, for the timer object `t`.

Input Arguments

t

Object of class `timer`.

S

Structure whose field names are `timer` property names.

PN,PV

Cell array of character vectors, `PN`, and corresponding cell array of values, `PV`. The cell array of `PN` must be a 1-by-N or N-by-1 array. If `t` is an array of timer objects, `PV` can be an M-by-N cell array, where M is equal to the length of `t` and N is equal to the length of `PN`. In this case, each timer object is updated with a different set of values for the list of property names contained in `PN`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

BusyMode

Character vector or string scalar that indicates action taken when a timer has to execute `TimerFcn` before the completion of previous execution of the `TimerFcn`. When `Running='on'`, `BusyMode` is read only. This table summarizes the busy modes.

BusyMode Values	Behavior if Queue Empty	Behavior if Queue Not Empty	Notes
`'drop'`	Adds task to queue	Drops task	Possible skipping of `TimerFcn` calls

BusyMode Values	Behavior if Queue Empty	Behavior if Queue Not Empty	Notes
`'error'`	Adds task to queue	Completes task; throws error specified by `ErrorFcn`; stops timer	Stops timer after completing task in execution queue
`'queue'`	Adds task to queue	Waits for queue to clear, and then enters task in queue	Adjusts `Period` property to manage tasks in execution queue

See “Handling Timer Queuing Conflicts” for more information.

Default: `'drop'`

`ErrorFcn`

Character vector, string scalar, function handle, or cell array defining the function that the timer executes when an error occurs. If there is an error, this function executes, and then calls `StopFcn`.

- If you specify this property using a character vector or string scalar, when MATLAB executes the callback it evaluates the MATLAB code contained in the character vector.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

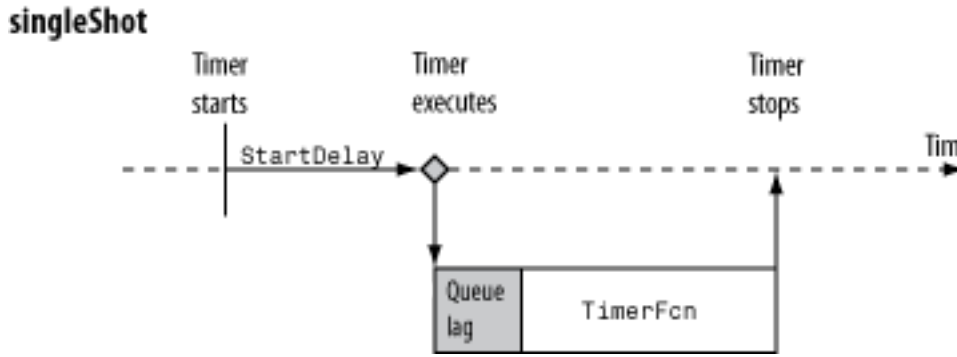
For more information, see “Timer Callback Functions”.

`ExecutionMode`

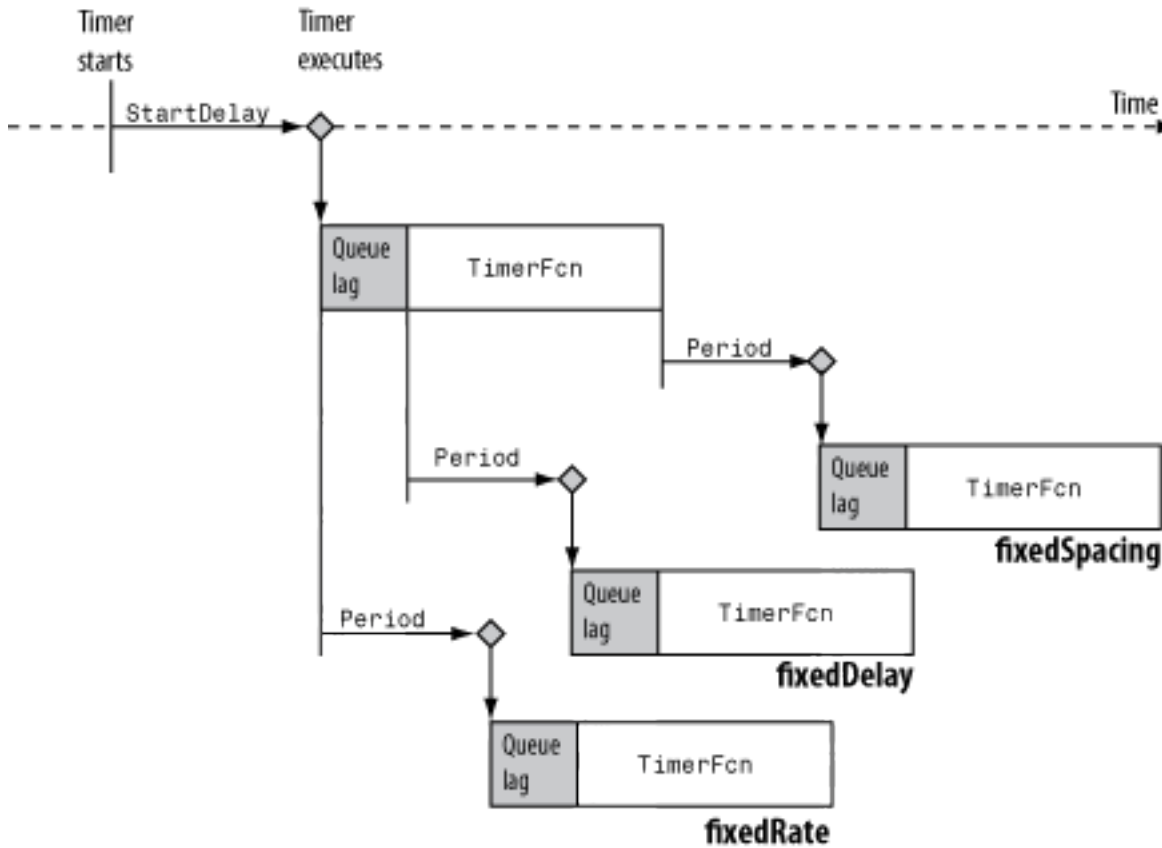
Character vector or string scalar that defines how the `timer` object schedules timer events. When `Running='on'`, `ExecutionMode` is read only. This table summarizes the execution modes.

Execution Mode	Time Period Start Point
`'singleShot'`	In this mode, the timer callback function is only executed once. Therefore, the `Period` property has no effect. This is the default execution mode.
`'fixedRate'`	Starts immediately after the timer callback function is added to the MATLAB execution queue
`'fixedDelay'`	Starts when the timer function callback restarts execution after a time lag due to delays in the MATLAB execution queue
`'fixedSpacing'`	Starts when the timer callback function finishes executing.

- `'singleShot'` is the single execution mode for the `timer` class, and is the default value.



- `'fixedDelay'`, `'fixedRate'`, and `'fixedSpacing'` are the three supported multiexecution modes. These modes define the starting point of the `Period` property. The `Period` property specifies the amount of time between executions, which remains the same. Only the point at which execution begins is different.



Default: `'singleShot'`

`Name`

Character vector or string scalar representing the `timer` name.

Default: `'timer-i'`, where *i* is a number indicating the *i*th timer object created this session. To reset *i* to 1, execute the `clear classes` command.

`ObjectVisibility`

Character vector or string scalar with possible values of `'on'` or `'off'`, that provides a way for you to discourage end-user access to the timer objects your application creates.

The `timerfind` function does not return an object whose `ObjectVisibility` property is set to `'off'`. Objects that are not visible are still valid. To retrieve a list of all the timer objects in memory, including the invisible ones, use the `timerfindall` function.

Default: `'on'`

`Period`

Number greater than 0.001 that specifies the delay, in seconds, between executions of `TimerFcn`. For the timer to use `Period`, you must set `ExecutionMode` and `TasksToExecute` to schedule multiple timer object callback events.

Default: 1.0

`StartDelay`

Number greater than or equal to 0 that specifies the delay, in seconds, between the start of the timer and the first execution of the function specified in `TimerFcn`. When `Running = 'on'`, `StartDelay` is read only.

Default: 0

`StartFcn`

Character vector, string scalar, function handle, or cell array defining the function that executes when the timer starts.

- If you specify this property using a character vector or string scalar, when MATLAB executes the callback it evaluates the MATLAB code contained in the character vector.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see “Timer Callback Functions”.

StopFcn

Character vector, string scalar, function handle, or cell array defining the function that executes when the timer stops.

- If you specify this property using a character vector or string scalar, when MATLAB executes the callback it evaluates the MATLAB code contained in the character vector.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see “Timer Callback Functions”.

The timer stops when

- You call the timer `stop` method.
- The timer finishes executing `TimerFcn`. In other words, the value of `TasksExecuted` reaches the limit set by `TasksToExecute`.
- An error occurs. The `ErrorFcn` callback is called first, followed by the `StopFcn` callback.

You can use `StopFcn` to define clean up actions, such as deleting the timer object from memory.

Tag

Character vector or string scalar that represents a label for the object.

TasksToExecute

Number greater than 0, indicating the number of times the timer object is to execute the `TimerFcn` callback. Use the `TasksToExecute` property to set the number of executions. To use `TasksToExecute`, you must set `ExecutionMode` to schedule multiple timer callback events.

Default: `Inf`

TimerFcn

Character vector, string scalar, function handle, or cell array defining the timer callback function. You must define this property before you can start the timer.

- If you specify this property using a character vector or string scalar, when MATLAB executes the callback it evaluates the MATLAB code contained in the character vector.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see “Timer Callback Functions”.

UserData

Generic field for data that you want to add to the object.

Output Arguments

propStruct

Configurable properties of `t`, returned as a structure. The field names of `propStruct` are the property names of `t`, and the associated values of `propStruct` are cell arrays of the possible property values. If the property in `t` does not have a finite set of possible values, the property value in `propStruct` is an empty cell array.

propCell

Possible values of a given property name, returned as cell array of character vectors. If the property does not have a finite set of possible values, `set` returns an empty cell array.

Examples

Display Configurable Timer Object Properties

Instantiate a timer object and call the `set` method.

```
t = timer;
set(t)
```

```
    BusyMode: [ {drop} | queue | error ]
    ErrorFcn: string -or- function handle -or- cell array
    ExecutionMode: [ {singleShot} | fixedSpacing | fixedDelay | fixedRate ]
    Name
    ObjectVisibility: [ {on} | off ]
    Period
    StartDelay
    StartFcn: string -or- function handle -or- cell array
    StopFcn: string -or- function handle -or- cell array
    Tag
    TasksToExecute
    TimerFcn: string -or- function handle -or- cell array
    UserData
```

Some of the timer properties, such as `Running`, are not displayed because they are read only.

Use the `set` method to output a structure.

```
out = set(t)
```

```
out = struct with fields:
            BusyMode: {3x1 cell}
            ErrorFcn: {}
       ExecutionMode: {4x1 cell}
                Name: {}
    ObjectVisibility: {2x1 cell}
              Period: {}
          StartDelay: {}
            StartFcn: {}
             StopFcn: {}
                 Tag: {}
      TasksToExecute: {}
            TimerFcn: {}
            UserData: {}
```

Delete the timer object from memory.

```
delete(t)
```

Display Possible Property Values

Instantiate a timer object and display possible values for the `BusyMode` property.

```
t = timer;
set(t,'BusyMode')
```

```
[ {drop} | queue | error ]
```

The output shows the three possible values for `BusyMode`. The default value, `drop`, is indicated by curly braces.

Display the possible values for `ErrorFcn`.

```
set(t,'ErrorFcn')
```

```
string scalar -or- character vector -or- function handle -or- cell array
```

A description of the possible values is displayed since `ErrorFcn` does not have a set list of possible values.

Output the possible property values.

```
out1 = set(t,'BusyMode')
```

```
out1 = 3x1 cell array
    {'drop' }
    {'queue'}
    {'error'}


out2 = set(t,'ErrorFcn')
```

```
out2 =

  0x0 empty cell array
```

While `set(t,'ErrorFcn')` displays a description of the possible values, `out2 = set(t,'ErrorFcn')` returns an empty cell.

Delete the timer from memory.

```
delete(t)
```

Set Timer Object Property

Instantiate a timer object and make the object invisible. Display the object's visibility and delete the object.

```
t = timer;
set(t,'ObjectVisibility','off')
get(t,'ObjectVisibility')
```

```
ans =
'off'
```

```
delete(t)
```

Set Timer Object Properties Using Structure

Construct a structure to modify several timer object properties.

```
s.BusyMode = 'queue';
s.ExecutionMode = 'fixedDelay';
s.ObjectVisibility = 'off'
```

```
s = struct with fields:
            BusyMode: 'queue'
       ExecutionMode: 'fixedDelay'
    ObjectVisibility: 'off'
```

Create a timer, display the properties in `s`, modify the timer, display the new values of the properties, and delete the timer.

```
t = timer;
get(t,{'BusyMode','ExecutionMode','ObjectVisibility'})
```

```
ans = 1x3 cell array
    {'drop'}    {'singleShot'}    {'on'}
```

```
set(t,s)
get(t,{'BusyMode','ExecutionMode','ObjectVisibility'})
```

```
ans = 1x3 cell array
    {'queue'}    {'fixedDelay'}    {'off'}
```

```
delete(t)
```

Set Timer Objects Using Cell Arrays

Create a cell array of properties to modify, and a cell array of the values of the corresponding properties. Instantiate a timer, and display the initial values of the properties in the property name cell array, `nameArr`.

```
nameArr = {'BusyMode','ExecutionMode','Period'};
valArr = {'queue','fixedDelay',3};
t = timer;
get(t,nameArr)
```

```
ans=1×3 cell
    {'drop'}    {'singleShot'}    {[1]}
```

Modify the timer object and display the new property values. Delete the timer.

```
set(t,nameArr,valArr)
get(t,nameArr)
```

```
ans=1×3 cell
    {'queue'}    {'fixedDelay'}    {[3]}
```

```
delete(t)
```

Instantiate an array of three timers. Create a new property name cell array to modify the `BusyMode`, `ExecutionMode`, and `UserData` properties. Display the initial values of the properties for each timer.

```
tArr = [timer timer timer];
nameArr = {'BusyMode','ExecutionMode','UserData'};
get(tArr,nameArr)
```

```
ans=3×3 cell
    {'drop'}    {'singleShot'}    {0x0 double}
    {'drop'}    {'singleShot'}    {0x0 double}
    {'drop'}    {'singleShot'}    {0x0 double}
```

Assign each property a different value in each timer. Create a cell array containing the new values. Each row indicates the values for the properties in the corresponding timer.

```
valArr = {'queue','fixedDelay',3;...
            'error','fixedSpacing',42;...
            'drop','fixedRate','hello'};
```

Modify the timer object properties and display the updated values.

```
set(tArr,nameArr,valArr)
get(tArr,nameArr)
```

```
ans=3×3 cell
    {'queue'}    {'fixedDelay'  }    {[    3]}
    {'error'}    {'fixedSpacing'}    {[   42]}
    {'drop' }    {'fixedRate'   }    {'hello'}
```

Delete the timers from memory.

```
delete(tArr)
```

Alternatives

You can also use dot notation to set timer object properties. For example, `t.ObjectVisibility = 'off'` sets the property to the same value as `set(t,'ObjectVisibility','off')`.

See Also

`get` | `timer`

Introduced before R2006a

start

Class: `timer`

Start timer object

Syntax

```
start(t)
```

Description

`start(t)` starts the timer object, `t`. If `t` is an array of timer objects, `start` starts all the timers.

The `start` method sets the `Running` property of the timer object to `'on'`, executes the `StartFcn` callback, and initiates `TimerFcn` callback .

Input Arguments

t

Object of class `timer`.

Examples

Start Timer

Create and start a timer that displays the message 'timer started.' as the `StartFcn` callback and generates a random number as the `TimerFcn` callback. Delete the timer.

```
t = timer('StartFcn',@(~,~)disp('timer started.'),'TimerFcn',@(~,~)disp(rand(1)));
start(t)
delete(t)
```

```
timer started.
    0.9706
```

Your output from `rand` will vary.

Start Several Timers

Create and start three timers that displays a message for the `StartFcn` callbacks and compute the sine, cosine, and tangent of `pi/4` as the `TimerFcn` callbacks. Delete the timers.

```
t1 = timer('StartFcn',@(~,~)disp('t1 started.'),'TimerFcn',@(~,~)sin(pi/4));
t2 = timer('StartFcn',@(~,~)disp('t2 started.'),'TimerFcn',@(~,~)cos(pi/4));
t3 = timer('StartFcn',@(~,~)disp('t3 started.'),'TimerFcn',@(~,~)tan(pi/4));
start([t1 t2 t3]);
delete([t1 t2 t3]);
```

```
t1 started.
t2 started.
t3 started.
```

See Also

`delete` | `startat` | `stop` | `timer`

Introduced before R2006a

startat

Class: `timer`

Schedule timer to fire at specified time

Syntax

```
startat(t,firingTime)
startat(t,Y,M,D)
startat(t,Y,M,D,H,MI,S)
```

Description

`startat(t,firingTime)` schedules timer, `t`, to fire at specified time, `firingTime`. A timer fires by executing the callback function, `timerFcn`. `firingTime` must be within 25 days of the current time.

- If `t` is an array of timer objects and `firingTime` is a scalar, `startat` sets all the timers to fire at the specified time.
- If `t` is an array of timer objects and `firingTime` is an array of the same size as `t`, `startat` sets each timer to fire at the corresponding time.

`startat(t,Y,M,D)` starts the timer and schedules execution of `TimerFcn` at the year (`Y`), month (`M`), and day (`D`) specified.

`startat(t,Y,M,D,H,MI,S)` also specifies the hour (`H`), minute (`MI`), and second (`S`) specified.

Input Arguments

`t`

Object of class `timer`.

firingTime

Time at which the timer object is to fire, specified as a serial date number, character representation of date format, or a date vector. `firingTime` can be a single date or an array of dates with the same number of rows as timer objects in `t`.

- A serial date number indicates the number of days that have elapsed since 1-Jan-0000 (starting at 1). See `datenum` for additional information about serial date numbers.
- To specify character representation of dates, use the following date formats defined by the `datestr` function: 0, 1, 2, 6, 13, 14, 15, 16, or 23. These numeric identifiers correspond to formats defined by the `formatOut` property of the `datestr` function. Dates with two-character years are interpreted to be within the 100 years centered on the current year.
- Date vectors are specified as an `m-by-6` or `m-by-3` matrix containing `m` full or partial date vectors, respectively. A full date vector has six elements indicating year, month, day, hour, minute, and second, in that order. A partial date vector has three elements indicating year, month, and day, in that order.

Y,M,D

Time at which the timer object is to fire, specified as numbers indicating the year (`Y`), month (`M`), and day (`D`). Month values less than 1 are set to 1; other arguments can wrap and have negative values.

Y,M,D,H,MI,S

Time at which the timer object is to fire, specified as numbers indicating the year (`Y`), month (`M`), day (`D`), hour (`H`), minute (`MI`), and second (`S`) specified. Month values less than 1 are set to 1; other arguments can wrap and have negative values.

Examples

Start Timer in 2 Seconds

Create a timer that displays messages at start time and firing time.

```
t = timer('TimerFcn', @(~,~)disp('Fired.'), ...
    'StartFcn', @(~,~)disp('Started.'));
```

Set the timer to fire 2 seconds from the present time using a serial date. A serial date is specified in days.

```
two = 2/(60^2*24); % two seconds in serial time
fTime = now + two
startat(t,fTime);
```

```
fTime =

   7.3527e+05

Started.
Fired.
```

Wait for the timer to fire, and then delete the timer.

```
delete(t)
```

Start Timer Using Year, Month, Day

Create a timer that displays messages at start time and firing time.

```
t = timer('TimerFcn', @(~,~)disp('Fired.'), ...
    'StartFcn', @(~,~)disp('Started.'));
```

Schedule the timer to start 2 days from present at 00:00:00

```
[Y, M, D, H, MI, S] = datevec(now+2);
startat(t,Y,M,D)
```

```
Started.
```

Manually stop and delete the timer.

```
stop(t)
delete(t)
```

Algorithms

- The `startat` method specifies when the timer object executes the `TimerFcn` callback, not when the timer starts running. The timer starts running with the call to the `startat` method.

- Based on the specified time, `startat` computes and sets the required `StartDelay` property of the timer object, `t`. Additionally, it sets the `Running` property of the timer object to `'on'`, and executes the `StartFcn` callback.
- `startat` modifies the `timer` object's `startDelay` property. As such, `startat` overrides specified values of the timer's `startDelay` property.

See Also

`delete` | `start` | `stop` | `timer`

Topics

"Carryover in Date Vectors and Strings"

Introduced before R2006a

stop

Class: `timer`

Stop timer object

Syntax

```
stop(t)
```

Description

`stop(t)` stops the timer object, `t`. If `t` is an array of timer objects, the `stop` method stops each timer.

The `stop` method sets the `Running` property of the timer object to `'off'` and executes the `StopFcn` callback .

Input Arguments

`t`

Object of class `timer`.

Examples

Stop Timer

Create a timer object that generates 100 random numbers and executes one million times. Define a `StopFcn` callback that displays the message 'Timer has stopped.' Start the timer and verify the timer is running

```
t = timer('TimerFcn','rand(100,1);',...
    'ExecutionMode','fixedSpacing','TasksToExecute',1e6,...
```

```
    'StopFcn','disp(''Timer has stopped.'')');
start(t)
t.Running
```

```
ans =
'on'
```

Manually stop the timer and verify it is no longer running. Delete the timer.

```
stop(t)
t.Running
```

```
ans =
'off'
```

```
delete(t)
```

Tips

- Use the `stop` method to stop a timer manually. The timer automatically stops when the `TimerFcn` callback executes the number of times specified by the `ExecutionMode` and `TasksToExecute` properties or when an error occurs while executing a `TimerFcn` callback.

See Also

`delete` | `start` | `startat` | `timer`

Introduced before R2006a

timerfind

Class: `timer`

Find timer object

Syntax

```
out = timerfind
out = timerfind(Name,Value)
out = timerfind(t,Name,Value)
out = timerfind(S)
```

Description

`out = timerfind` finds the “visible timer objects” on page 1-15681and returns an array, `out`.

`out = timerfind(Name,Value)` finds “visible timer objects” on page 1-15681 with property values matching those passed as `Name,Value` pair arguments and returns an array, `out`.

`out = timerfind(t,Name,Value)` matches `Name,Value` pair arguments to the timer objects listed in `t`, where `t` can be an array of timer objects, and returns an array, `out`.

`out = timerfind(S)` matches property values defined in the structure, `S`, and returns an array, `out`. The field names of `S` are timer object property names and the field values are the corresponding property values.

Input Arguments

`t`

Array of objects of class `timer`

S

Structure with field names corresponding to `timer` object property names. Field values are the corresponding property values.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

BusyMode

Character vector or string scalar that indicates action taken when a timer has to execute `TimerFcn` before the completion of previous execution of the `TimerFcn`. When `Running='on'`, `BusyMode` is read only. This table summarizes the busy modes.

BusyMode Values	Behavior if Queue Empty	Behavior if Queue Not Empty	Notes
`'drop'`	Adds task to queue	Drops task	Possible skipping of `TimerFcn` calls
`'error'`	Adds task to queue	Completes task; throws error specified by `ErrorFcn`; stops timer	Stops timer after completing task in execution queue
`'queue'`	Adds task to queue	Waits for queue to clear, and then enters task in queue	Adjusts `Period` property to manage tasks in execution queue

See “Handling Timer Queuing Conflicts” for more information.

Default: `'drop'`

ErrorFcn

Character vector, string scalar, function handle, or cell array defining the function that the timer executes when an error occurs. If there is an error, this function executes, and then calls `StopFcn`.

- If you specify this property using a character vector or string scalar, when MATLAB executes the callback it evaluates the MATLAB code contained in the character vector.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

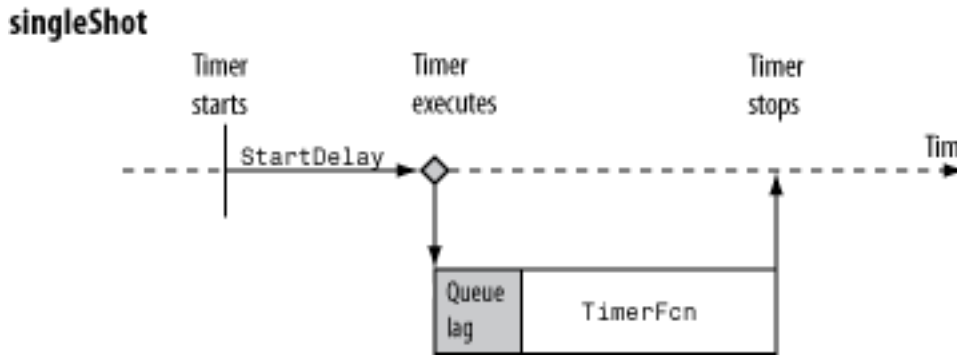
For more information, see “Timer Callback Functions”.

ExecutionMode

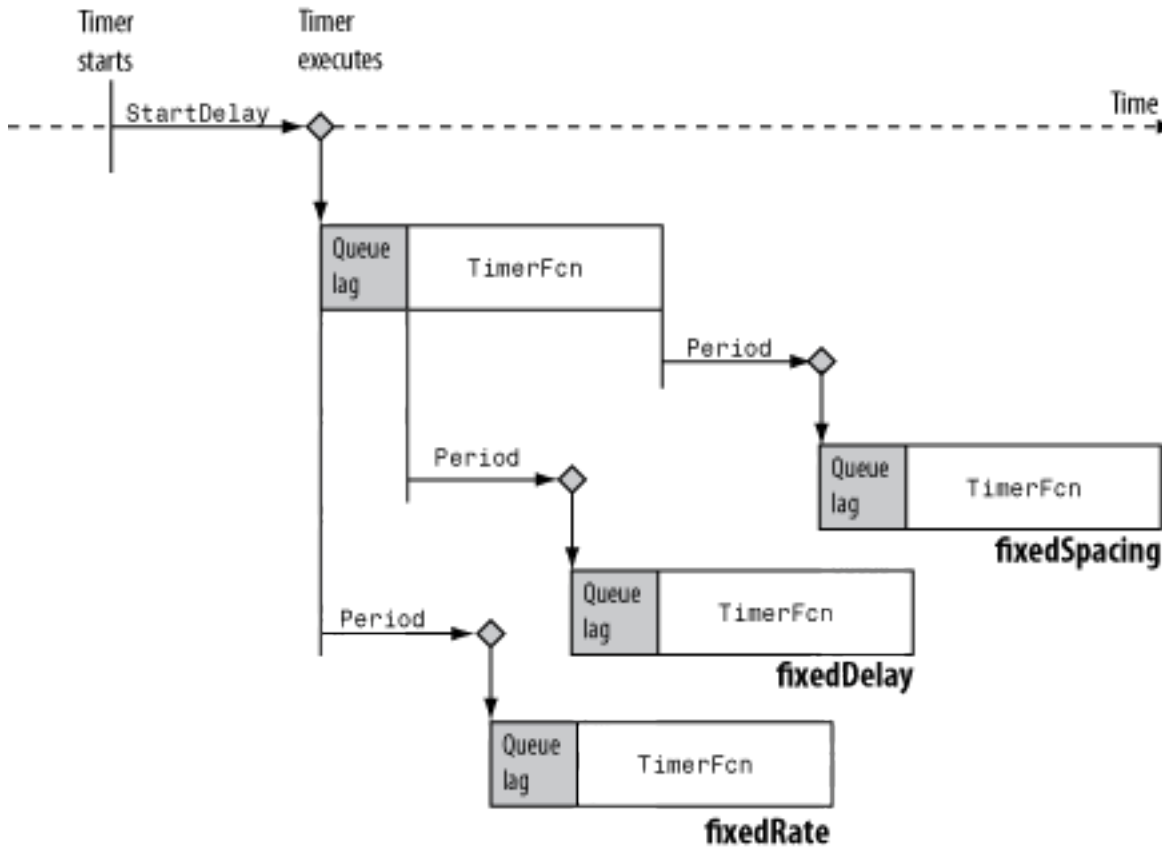
Character vector or string scalar that defines how the `timer` object schedules timer events. When `Running='on'`, `ExecutionMode` is read only. This table summarizes the execution modes.

Execution Mode	Time Period Start Point
`'singleShot'`	In this mode, the timer callback function is only executed once. Therefore, the `Period` property has no effect. This is the default execution mode.
`'fixedRate'`	Starts immediately after the timer callback function is added to the MATLAB execution queue
`'fixedDelay'`	Starts when the timer function callback restarts execution after a time lag due to delays in the MATLAB execution queue
`'fixedSpacing'`	Starts when the timer callback function finishes executing.

- `'singleShot'` is the single execution mode for the `timer` class, and is the default value.



- `'fixedDelay'`, `'fixedRate'`, and `'fixedSpacing'` are the three supported multiexecution modes. These modes define the starting point of the `Period` property. The `Period` property specifies the amount of time between executions, which remains the same. Only the point at which execution begins is different.



Default: `'singleShot'`

`Name`

Character vector or string scalar representing the `timer` name.

Default: `'timer-i'`, where *i* is a number indicating the *i*th timer object created this session. To reset *i* to 1, execute the `clear classes` command.

`ObjectVisibility`

Character vector or string scalar with possible values of `'on'` or `'off'`, that provides a way for you to discourage end-user access to the timer objects your application creates.

The `timerfind` function does not return an object whose `ObjectVisibility` property is set to `'off'`. Objects that are not visible are still valid. To retrieve a list of all the timer objects in memory, including the invisible ones, use the `timerfindall` function.

Default: `'on'`

`Period`

Number greater than 0.001 that specifies the delay, in seconds, between executions of `TimerFcn`. For the timer to use `Period`, you must set `ExecutionMode` and `TasksToExecute` to schedule multiple timer object callback events.

Default: 1.0

`StartDelay`

Number greater than or equal to 0 that specifies the delay, in seconds, between the start of the timer and the first execution of the function specified in `TimerFcn`. When `Running = 'on'`, `StartDelay` is read only.

Default: 0

`StartFcn`

Character vector, string scalar, function handle, or cell array defining the function that executes when the timer starts.

- If you specify this property using a character vector or string scalar, when MATLAB executes the callback it evaluates the MATLAB code contained in the character vector.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see “Timer Callback Functions”.

StopFcn

Character vector, string scalar, function handle, or cell array defining the function that executes when the timer stops.

- If you specify this property using a character vector or string scalar, when MATLAB executes the callback it evaluates the MATLAB code contained in the character vector.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see “Timer Callback Functions”.

The timer stops when

- You call the timer `stop` method.
- The timer finishes executing `TimerFcn`. In other words, the value of `TasksExecuted` reaches the limit set by `TasksToExecute`.
- An error occurs. The `ErrorFcn` callback is called first, followed by the `StopFcn` callback.

You can use `StopFcn` to define clean up actions, such as deleting the timer object from memory.

Tag

Character vector or string scalar that represents a label for the object.

TasksToExecute

Number greater than 0, indicating the number of times the timer object is to execute the `TimerFcn` callback. Use the `TasksToExecute` property to set the number of executions. To use `TasksToExecute`, you must set `ExecutionMode` to schedule multiple timer callback events.

Default: `Inf`

`TimerFcn`

Character vector, string scalar, function handle, or cell array defining the timer callback function. You must define this property before you can start the timer.

- If you specify this property using a character vector or string scalar, when MATLAB executes the callback it evaluates the MATLAB code contained in the character vector.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see “Timer Callback Functions”.

`UserData`

Generic field for data that you want to add to the object.

Read Only Name-Value Pair Arguments

`AveragePeriod`

Average time in seconds between `TimerFcn` executions since the timer started. Value is `NaN` until timer executes two timer callbacks.

`InstantPeriod`

The time in seconds between the last two executions of `TimerFcn`. Value is `NaN` until timer executes two timer callbacks.

`Running`

Character vector defined as `'off'` or `'on'`, indicating whether the timer is currently executing callback functions.

`TasksExecuted`

The number of times the timer called `TimerFcn` since the timer started.

Type

Character vector that identifies the object type.

Examples

Find Timer Objects Existing in Memory

Create several individual timers and an array of timers.

```
t1 = timer('Tag','broadcastProgress','UserData','Monday');
t2 = timer('Tag','displayProgress','UserData','Monday');
timerArr = [timer('Tag','broadcastProgress','UserData','Tuesday');
    timer('Tag','displayProgress','UserData','Tuesday');
    timer('Tag','displayProgress','UserData','Wednesday');];
```

Find all the timers in memory.

```
out1 = timerfind
```

```
out1=1×6 object
Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:                Name:
   1       fixedSpacing    30       @(~,~)logStack()         timer-1
   2       singleShot      1        ''                       timer-2
   3       singleShot      1        ''                       timer-3
   4       singleShot      1        ''                       timer-4
   5       singleShot      1        ''                       timer-5
   6       singleShot      1        ''                       timer-6
```

Find only those timers in memory that have the `Tag` property value `'displayProgress'`.

```
out2 = timerfind('Tag','displayProgress')
```

```
out2=1×3 object
Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:                Name:
   1       singleShot      1        ''                       timer-3
```

```
   2       singleShot      1        ''                        timer-5
   3       singleShot      1        ''                        timer-6
```

Limit the search for timers to timer objects in `timerArr` with the `Tag` property value `'displayProgress'`.

```
out3 = timerfind(timerArr,'Tag','displayProgress')
```

```
out3=2×1 object
Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:                Name:
   1       singleShot      1        ''                       timer-5
   2       singleShot      1        ''                       timer-6
```

Define a `struct` containing the `Tag` and `UserData` properties of interest.

```
searchStruct = struct('Tag','broadcastProgress','UserData','Monday')
```

```
searchStruct = struct with fields:
         Tag: 'broadcastProgress'
    UserData: 'Monday'
```

Use the `struct` as the search criteria to find timer objects in memory.

```
out4 = timerfind(searchStruct)
```

```
out4 =
   Timer Object: timer-2

   Timer Settings
      ExecutionMode: singleShot
             Period: 1
           BusyMode: drop
            Running: off

   Callbacks
           TimerFcn: ''
           ErrorFcn: ''
           StartFcn: ''
            StopFcn: ''
```

Delete the timer objects.

```
delete(t1)
delete(t2)
delete(timerArr)
```

Delete Timer by Name

Simulate having existing timers in memory by creating an array of timers. Create a new timer with a custom name. List all visible timers.

```
existingTimers = [timer timer timer];

myTimerName = 'myTimer';
anotherTimer = timer('Name',myTimerName);

timerfind
```

```
Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:                Name:
   1       singleShot      1        ''                       timer-1
   2       singleShot      1        ''                       timer-2
   3       singleShot      1        ''                       timer-3
   4       singleShot      1        ''                       myTimer
```

Delete the specified timer and list all visible timers.

```
delete(timerfind('Name',myTimerName));
timerfind
```

```
Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:                Name:
   1       singleShot      1        ''                       timer-1
   2       singleShot      1        ''                       timer-2
   3       singleShot      1        ''                       timer-3
```

Delete all visible timers from memory.

```
delete(timerfind)
```

Find Valid Timer Objects Cleared from Workspace

Use `timerfind` to find 'lost' timer object references. References are lost when you clear the timer object from the workspace, but do not delete it from memory.

Create two timer objects. Since the callback function does not require the timer or event object, you can use the tilde (~) operator to ignore the inputs in the function handle.

```
t1 = timer('TimerFcn',@(~,~)disp('Timer 1 Fired!'));
t2 = timer('TimerFcn',@(~,~)disp('Timer 2 Fired!'));
whos
```

```
  Name      Size            Bytes  Class    Attributes

  t1        1x1               104  timer
  t2        1x1               104  timer
```

Clear one of the timer objects from the workspace. To actually remove the timer from memory, you need to both clear it and delete it.

```
clear t1
whos
```

```
  Name      Size            Bytes  Class    Attributes

  t2        1x1               104  timer
```

Try to delete the timer, `t1`.

```
delete(t1)
```

```
Undefined function or variable 't1'.
```

The timer, `t1`, cannot be removed from memory using delete because its reference has been cleared.

Find valid timer objects in memory.

```
out = timerfind
```

```
Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:                Name:
   1       singleShot      1        1x1 function_handle arraytimer-1
   2       singleShot      1        1x1 function_handle arraytimer-2
```

Since two timers were found, determine which timer does not exist in the workspace.

```
out ~= t2
```

```
ans =

     1     0
```

The first timer object in `out` is not equal to `t2`. This was previously `t1`. It is reassigned to `t1`. Since it is still valid, the timer can be started.

```
t1 = out(1);
start(t1)
```

```
Timer 1 Fired!
```

Delete timer objects. `timerfind` provides a way of accessing timer objects in memory. It does not copy the objects; therefore you do not need to delete `out` from memory. To verify, use `timerfind`.

```
delete(t1)
delete(t2)
timerfind
```

```
ans =

     []
```

Delete All Timer Objects in Memory

Create four timer objects.

```
t1 = timer('TimerFcn',@(~,~)disp('Timer 1 Fired!'));
t2 = timer('TimerFcn',@(~,~)disp('Timer 2 Fired!'));
t3 = timer('TimerFcn',@(~,~)disp('Timer 3 Fired!'));
t4 = timer('TimerFcn',@(~,~)disp('Timer 4 Fired!'));
```

Clear two timers from the workspace.

```
clear t2 t3
```

Pass `timerfind` to `delete` to remove all timer objects from memory, whether or not they exist in the workspace.

```
delete(timerfind)
timerfind
```

```
ans =

     []
```

More About

visible timer objects

Visible timer objects are timer objects that are in memory and have the `ObjectVisibility` property set to 'on'.

Tips

- `timerfind` only finds “visible timer objects” on page 1-15681. Visible timer objects are those that are in memory and have the `ObjectVisibility` property set to `'on'`. To find objects that are hidden, but still valid, use `timerfindall`.

See Also

`delete` | `timer` | `timerfindall`

Introduced before R2006a

timerfindall

Class: `timer`

Find timer object, regardless of visibility

Syntax

```
out = timerfindall
out = timerfindall(Name,Value)
out = timerfindall(t,Name,Value)
out = timerfindall(S)
```

Description

`out = timerfindall` finds timer objects existing in memory, regardless of visibility and returns an array, `out`. Use the `ObjectVisibility` property to set the object's visibility.

`out = timerfindall(Name,Value)` finds timer objects existing in memory, regardless of visibility whose property values match those passed as `Name,Value` pair arguments and returns an array, `out`.

`out = timerfindall(t,Name,Value)` matches `Name,Value` pair arguments to the timer objects listed in `t`, where `t` can be an array of timer objects, and returns an array, `out`.

`out = timerfindall(S)` matches property values defined in the structure, `S` and returns an array, `out`. The field names of `S` are timer object property names and the field values are the corresponding property values.

Input Arguments

t

Array of objects of class `timer`.

S

Structure with field names corresponding to `timer` object property names. Field values are the corresponding property values.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`BusyMode`

Character vector or string scalar that indicates action taken when a timer has to execute `TimerFcn` before the completion of previous execution of the `TimerFcn`. When `Running='on'`, `BusyMode` is read only. This table summarizes the busy modes.

BusyMode Values	Behavior if Queue Empty	Behavior if Queue Not Empty	Notes
`'drop'`	Adds task to queue	Drops task	Possible skipping of `TimerFcn` calls
`'error'`	Adds task to queue	Completes task; throws error specified by `ErrorFcn`; stops timer	Stops timer after completing task in execution queue
`'queue'`	Adds task to queue	Waits for queue to clear, and then enters task in queue	Adjusts `Period` property to manage tasks in execution queue

See “Handling Timer Queuing Conflicts” for more information.

Default: `'drop'`

ErrorFcn

Character vector, string scalar, function handle, or cell array defining the function that the timer executes when an error occurs. If there is an error, this function executes, and then calls `StopFcn`.

- If you specify this property using a character vector or string scalar, when MATLAB executes the callback it evaluates the MATLAB code contained in the character vector.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

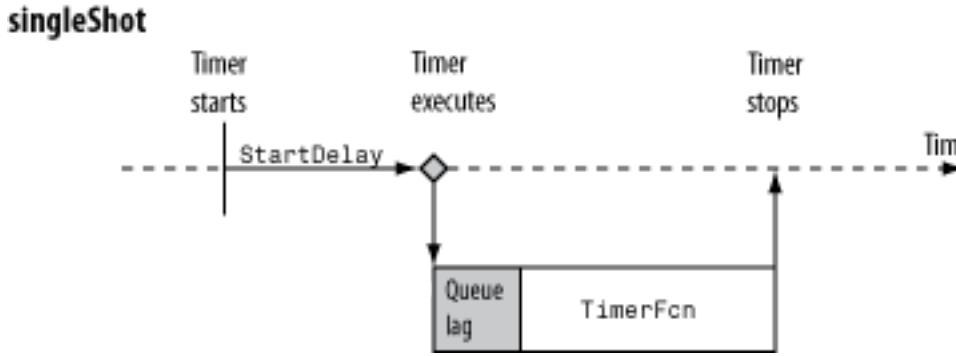
For more information, see “Timer Callback Functions”.

ExecutionMode

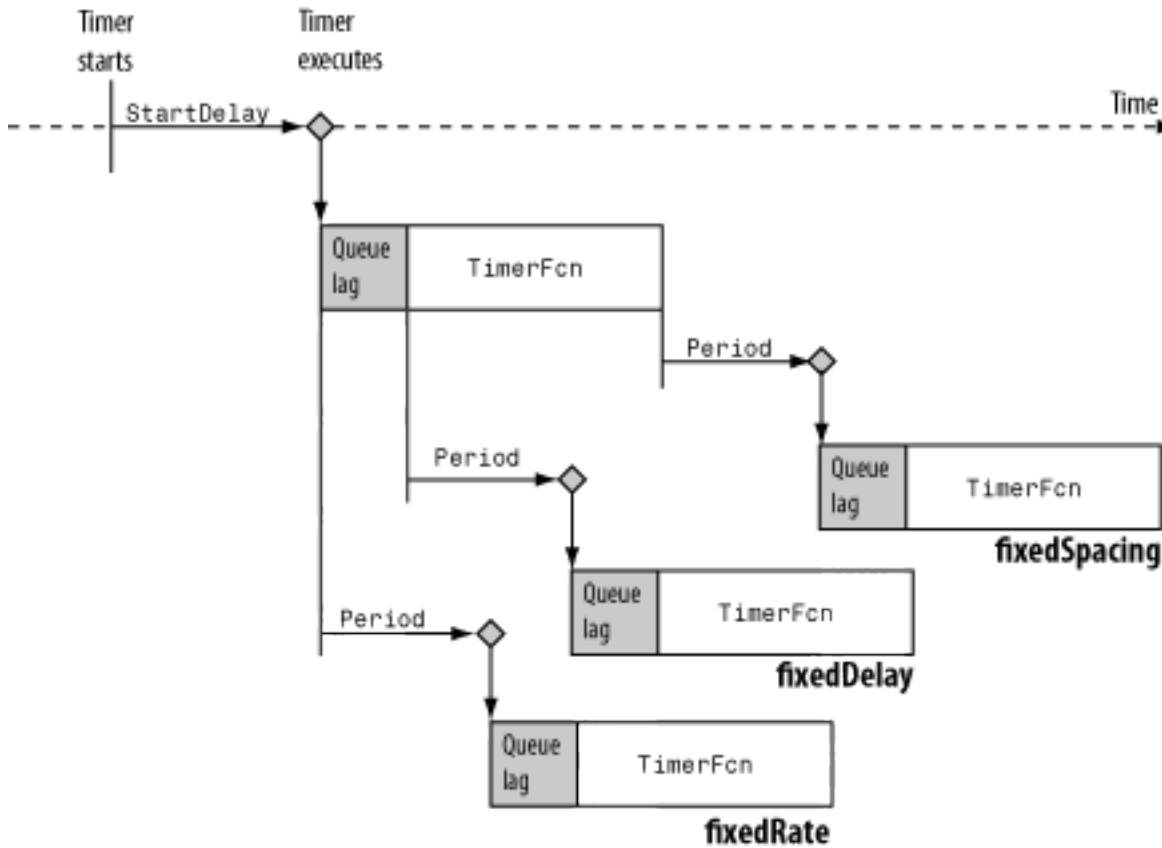
Character vector or string scalar that defines how the `timer` object schedules timer events. When `Running='on'`, `ExecutionMode` is read only. This table summarizes the execution modes.

Execution Mode	Time Period Start Point
`'singleShot'`	In this mode, the timer callback function is only executed once. Therefore, the `Period` property has no effect. This is the default execution mode.
`'fixedRate'`	Starts immediately after the timer callback function is added to the MATLAB execution queue
`'fixedDelay'`	Starts when the timer function callback restarts execution after a time lag due to delays in the MATLAB execution queue
`'fixedSpacing'`	Starts when the timer callback function finishes executing.

- `'singleShot'` is the single execution mode for the `timer` class, and is the default value.



- `'fixedDelay'`, `'fixedRate'`, and `'fixedSpacing'` are the three supported multiexecution modes. These modes define the starting point of the `Period` property. The `Period` property specifies the amount of time between executions, which remains the same. Only the point at which execution begins is different.



Default: `'singleShot'`

`Name`

Character vector or string scalar representing the `timer` name.

Default: `'timer-i'`, where *i* is a number indicating the *i*th timer object created this session. To reset *i* to 1, execute the `clear classes` command.

`ObjectVisibility`

Character vector or string scalar with possible values of `'on'` or `'off'`, that provides a way for you to discourage end-user access to the timer objects your application creates.

The `timerfind` function does not return an object whose `ObjectVisibility` property is set to `'off'`. Objects that are not visible are still valid. To retrieve a list of all the timer objects in memory, including the invisible ones, use the `timerfindall` function.

Default: `'on'`

`Period`

Number greater than 0.001 that specifies the delay, in seconds, between executions of `TimerFcn`. For the timer to use `Period`, you must set `ExecutionMode` and `TasksToExecute` to schedule multiple timer object callback events.

Default: 1.0

`StartDelay`

Number greater than or equal to 0 that specifies the delay, in seconds, between the start of the timer and the first execution of the function specified in `TimerFcn`. When `Running = 'on'`, `StartDelay` is read only.

Default: 0

`StartFcn`

Character vector, string scalar, function handle, or cell array defining the function that executes when the timer starts.

- If you specify this property using a character vector or string scalar, when MATLAB executes the callback it evaluates the MATLAB code contained in the character vector.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see “Timer Callback Functions”.

StopFcn

Character vector, string scalar, function handle, or cell array defining the function that executes when the timer stops.

- If you specify this property using a character vector or string scalar, when MATLAB executes the callback it evaluates the MATLAB code contained in the character vector.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see “Timer Callback Functions”.

The timer stops when

- You call the timer `stop` method.
- The timer finishes executing `TimerFcn`. In other words, the value of `TasksExecuted` reaches the limit set by `TasksToExecute`.
- An error occurs. The `ErrorFcn` callback is called first, followed by the `StopFcn` callback.

You can use `StopFcn` to define clean up actions, such as deleting the timer object from memory.

Tag

Character vector or string scalar that represents a label for the object.

TasksToExecute

Number greater than 0, indicating the number of times the timer object is to execute the `TimerFcn` callback. Use the `TasksToExecute` property to set the number of executions. To use `TasksToExecute`, you must set `ExecutionMode` to schedule multiple timer callback events.

Default: `Inf`

`TimerFcn`

Character vector, string scalar, function handle, or cell array defining the timer callback function. You must define this property before you can start the timer.

- If you specify this property using a character vector or string scalar, when MATLAB executes the callback it evaluates the MATLAB code contained in the character vector.
- If you specify this property using a function handle, when MATLAB executes the callback it passes the `timer` object and an event structure to the callback function. The event structure contains the type of event in the `Type` field and the time of the event in the `Data` field.
- If your callback function accepts arguments in addition to the `timer` object and event data, specify this property as a cell array containing the function handle and the additional arguments.

For more information, see “Timer Callback Functions”.

`UserData`

Generic field for data that you want to add to the object.

Read Only Name-Value Pair Arguments

`AveragePeriod`

Average time in seconds between `TimerFcn` executions since the timer started. Value is `NaN` until timer executes two timer callbacks.

`InstantPeriod`

The time in seconds between the last two executions of `TimerFcn`. Value is `NaN` until timer executes two timer callbacks.

`Running`

Character vector defined as `'off'` or `'on'`, indicating whether the timer is currently executing callback functions.

`TasksExecuted`

The number of times the timer called `TimerFcn` since the timer started.

Type

Character vector that identifies the object type.

Examples

Find and Delete All Timers From Memory

Create four timer objects.

```
t1 = timer('TimerFcn',@(~,~)disp('Timer 1 Fired!'));
t2 = timer('TimerFcn',@(~,~)disp('Timer 2 Fired!'));
t3 = timer('TimerFcn',@(~,~)disp('Timer 3 Fired!'));
t4 = timer('TimerFcn',@(~,~)disp('Timer 4 Fired!'));
```

Set timers `t2` and `t4` to be invisible.

```
t2.ObjectVisibility = 'off';
t4.ObjectVisibility = 'off';
```

Clear timers `t1` and `t2` from the workspace.

```
clear t1 t2
whos
```

```
  Name      Size            Bytes  Class    Attributes

  t3        1x1               104  timer
  t4        1x1               104  timer
```

Find all visible timers in memory.

```
timerfind
```

```
Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:               Name:
   1       singleShot      1        1x1 function_handle arraytimer-1
   2       singleShot      1        1x1 function_handle arraytimer-3
```

`timerfind` finds only timers `t1` and `t2`, since they are visible. Timer `t2` is still valid and in memory even though it was cleared from the workspace

Find all timers in memory.

```
timerfindall
```

```
Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:                Name:
   1       singleShot      1        1x1 function_handle arraytimer-1
   2       singleShot      1        1x1 function_handle arraytimer-2
   3       singleShot      1        1x1 function_handle arraytimer-3
   4       singleShot      1        1x1 function_handle arraytimer-4
```

`timerfindall` finds all four valid timers in memory even though `t2` and `t4` are invisible and `t1` and `t2` were cleared from the workspace.

Delete all timers from memory.

```
delete(timerfindall)
```

Find Timer Objects Existing in Memory

Create several individual timers and an array of timers.

```
t1 = timer('Tag', 'broadcastProgress','UserData','Monday');
t2 = timer('Tag', 'displayProgress','UserData','Monday');
timerArr = [timer('Tag', 'broadcastProgress','UserData','Tuesday');
    timer('Tag', 'displayProgress','UserData','Tuesday');
    timer('Tag', 'displayProgress','UserData','Wednesday');];
```

Make timer `t1` and `timerArr(2)` invisible.

```
t1.ObjectVisibility = 'off';
timerArr(2).ObjectVisibility = 'off';
```

Find all the timers in memory using `timerfind`.

```
out1 = timerfind
```

```
Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:                Name:
   1       singleShot      1        ''                       timer-2
   2       singleShot      1        ''                       timer-3
   3       singleShot      1        ''                       timer-5
```

`timerfind` does not find the hidden timers.

Find all the timers in memory using `timerfindall`.

```
out2 = timerfindall
```

```
Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:                Name:
   1       singleShot      1        ''                       timer-1
   2       singleShot      1        ''                       timer-2
   3       singleShot      1        ''                       timer-3
   4       singleShot      1        ''                       timer-4
   5       singleShot      1        ''                       timer-5
```

`timerfindall` finds all timers, even the invisible ones.

Find only those timers in memory that have the value, `'displayProgress'`, as the `Tag` property.

```
out3 = timerfindall('Tag','displayProgress')
```

```
Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:                Name:
   1       singleShot      1        ''                       timer-2
   2       singleShot      1        ''                       timer-4
   3       singleShot      1        ''                       timer-5
```

Limit the search for timers that have the value, `'displayProgress'`, as the `Tag` property to timer objects in `timerArr`.

```
out4 = timerfindall(timerArr,'Tag','displayProgress')
```

```
Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:                Name:
   1       singleShot      1        ''                       timer-4
   2       singleShot      1        ''                       timer-5
```

Define a `struct` containing the `Tag` and `UserData` properties of interest.

```
searchStruct = struct('Tag','broadcastProgress','UserData','Monday')
```

```
searchStruct =
```

```
         Tag: 'broadcastProgress'
    UserData: 'Monday'
```

Use the `struct` as the search criteria to find timer objects in memory.

```
out5 = timerfindall(searchStruct)
```

```
   Timer Object: timer-1

   Timer Settings
      ExecutionMode: singleShot
             Period: 1
           BusyMode: drop
            Running: off

   Callbacks
           TimerFcn: ''
           ErrorFcn: ''
           StartFcn: ''
            StopFcn: ''
```

Delete the timer objects.

```
delete(timerfindall)
```

Find Invisible Timers

Create four timer objects.

```
t1 = timer('TimerFcn',@(~,~)disp('Timer 1 Fired!'));
t2 = timer('TimerFcn',@(~,~)disp('Timer 2 Fired!'));
t3 = timer('TimerFcn',@(~,~)disp('Timer 3 Fired!'));
t4 = timer('TimerFcn',@(~,~)disp('Timer 4 Fired!'));
```

Set timers `t2` and `t4` to be invisible, and clear timers `t1` and `t2` from the workspace.

```
t2.ObjectVisibility = 'off';
t4.ObjectVisibility = 'off';
clear t1 t2;
whos
```

```
  Name      Size            Bytes  Class    Attributes
```

```
  t3        1x1                104  timer
  t4        1x1                104  timer
```

Find all valid invisible timers.

```
out = timerfindall('ObjectVisibility','off')
```

```
Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:                Name:
   1       singleShot      1        1x1 function_handle arraytimer-2
   2       singleShot      1        1x1 function_handle arraytimer-4
```

Both valid invisible timers were found by `timerfindall`, regardless of whether they were in the workspace.

Tips

- `timerfindall` finds timer objects in memory, regardless of the value of the `ObjectVisibility` property. To limit the search to objects with `ObjectVisibility` set to `'on'`, use `timerfind`. .

See Also

`delete` | `timer` | `timerfind`

Introduced before R2006a

wait

Class: `timer`

Block command prompt until timer stops running

Syntax

```
wait(t)
```

Description

`wait(t)` blocks the command prompt until timer, `t`, stops running. If `t` is an array of timer objects, `wait` blocks the MATLAB command line until each timer in `t` has stopped running.

To block the command line, the timer object must first start via `start` or `startat` before calling the `wait` method. If the timer is not running, `wait` returns immediately.

Input Arguments

`t`

Array of `timer` objects

Default:

Examples

Block Command Prompt While Timer Runs

Create a timer that waits 10 seconds, and then displays a message. Start the timer and wait for it to finish.

```
T = timer('TimerFcn',@(~,~)disp('Fired.'),'StartDelay',10);
start(T)
```

```
Fired.
```

Notice that after the timer starts, the MATLAB prompt returns.

Start the timer and use the wait method to block anyone from entering commands at the MATLAB command line. You must start the timer before calling the wait command.

```
start(T)
wait(T)
```

```
Fired.
```

Notice that after the timer starts, the MATLAB prompt disappears until the timer stops.

Delete the timer.

```
delete(T)
```

See Also

`start` | `timer`

Introduced before R2006a

nexttile

Create axes in tiled chart layout

Syntax

```
nexttile
nexttile(tilenum)
nexttile(span)
nexttile(tilenum,span)
nexttile(t, ___ )
ax = nexttile( ___ )
```

Description

`nexttile` creates an axes object and places it into the next empty tile of the tiled chart layout that is in the current figure. If there is no layout in the current figure, then `nexttile` creates a new layout and configures it using the `'flow'` tile arrangement. The resulting axes object is the current axes, so the next plotting command can plot into it.

`nexttile(tilenum)` specifies the number of the tile to place the axes into. The tile numbers start at `1` and increase from left-to-right and top-to-bottom. If the tile is occupied by an axes or chart object, `nexttile` makes that object the current axes.

`nexttile(span)` creates an axes object that spans across multiple rows or columns. Specify `span` as a vector of the form `[r c]`. The axes spans `r` rows by `c` columns of tiles. The upper left corner of the axes is positioned in the upper left corner of the first empty `r`-by-`c` region.

`nexttile(tilenum,span)` creates an axes object that spans across multiple rows or columns. In this case, `nexttile` places the upper left corner of the axes in the tile specified by `tilenum`.

`nexttile(t, ___ )` places the axes object in the tiled chart layout specified by `t`. Specify `t` before all other input arguments. This syntax is useful when you are working with multiple layouts, or when the layout is in a panel or tab instead of a figure.

`ax = nexttile( ___ )` returns the axes object. Use `ax` to set properties on the axes. You can also pass `ax` as an input argument to other graphics functions that operate on an axes object. For example, you can call the `colormap` or `colororder` function to change the color scheme for the axes.

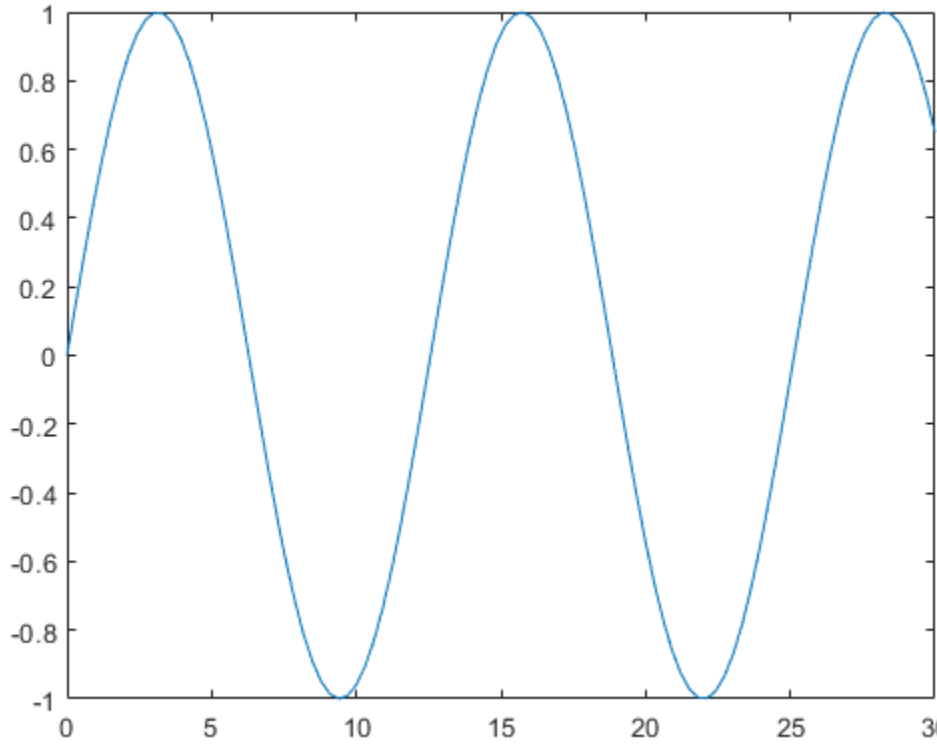
Examples

Create Flow Layout

If a tiled chart layout does not already exist, `nexttile` creates one.

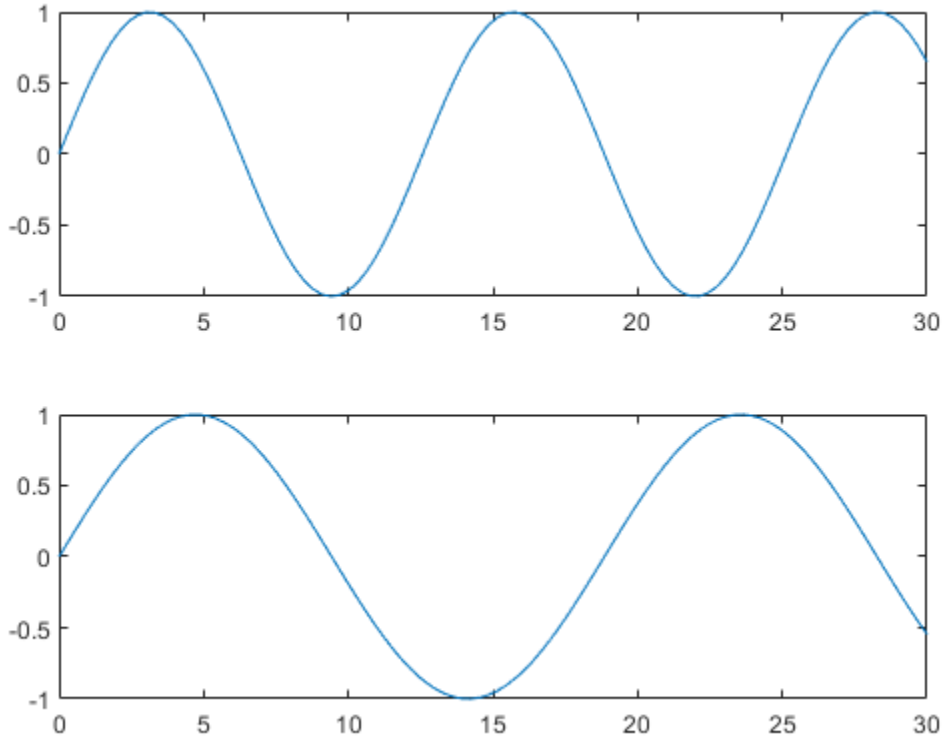
Create four coordinate vectors: `x`, `y1`, `y2`, and `y3`. Next, call the `nexttile` function to create a tiled chart layout and an axes object in the first tile. Then plot `y1` in the first tile. This first plot fills the entire layout because `nexttile` creates the layout using the `'flow'` tile arrangement.

```
x = linspace(0,30);
y1 = sin(x/2);
y2 = sin(x/3);
y3 = sin(x/4);

% Plot into first tile three times
nexttile
plot(x,y1)
```



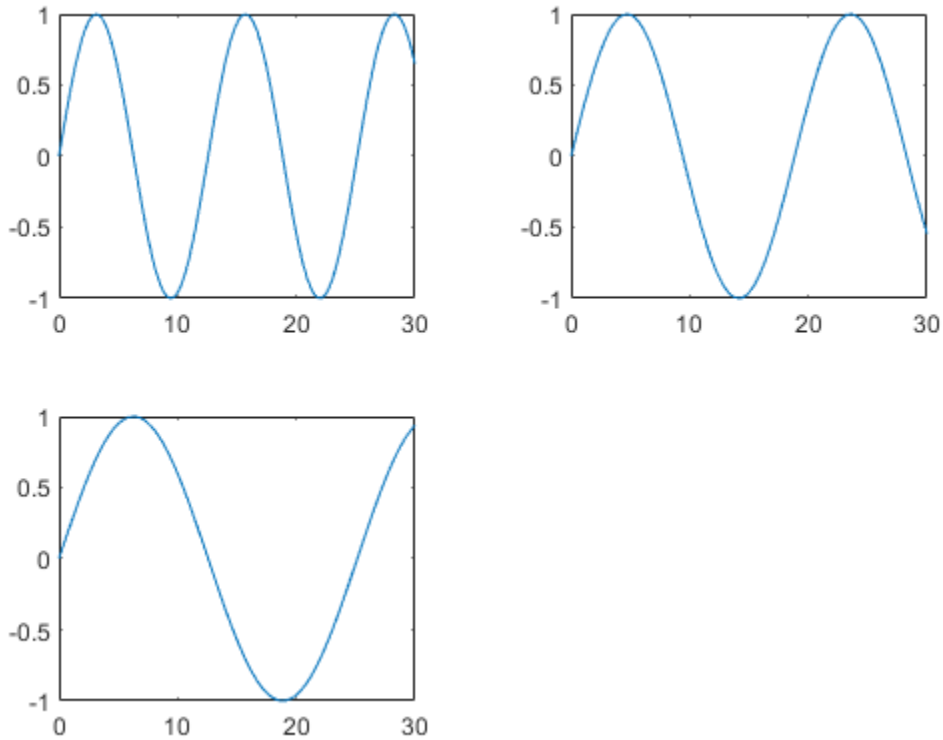
Create a second tile and axes, and plot into the axes.

```
nexttile
plot(x,y2)
```



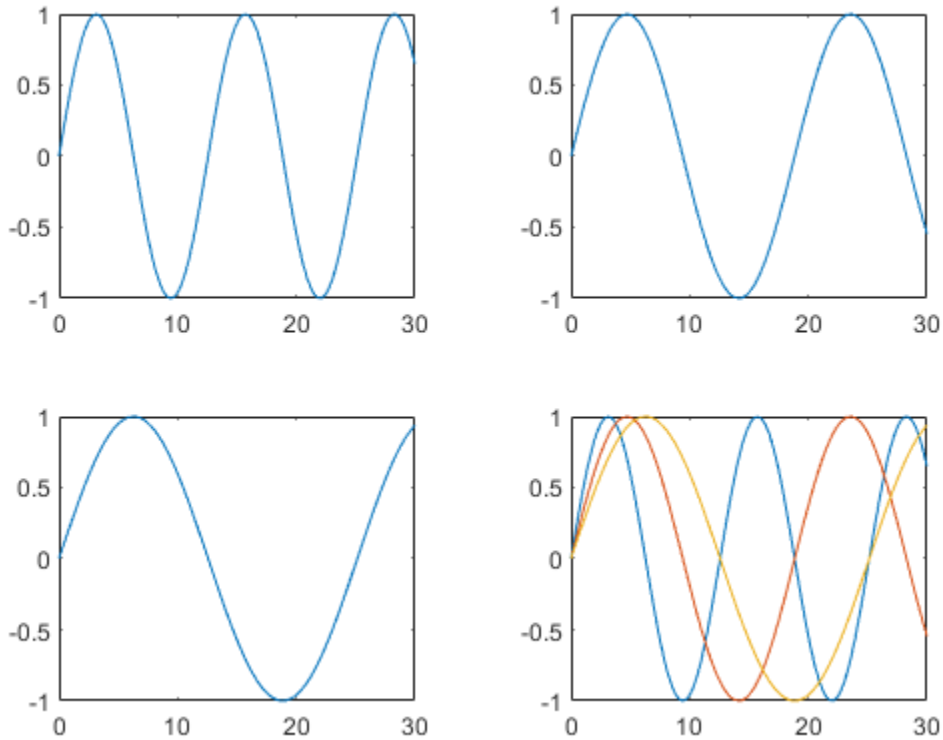
Repeat the process to create a third plot.

```
nexttile
plot(x,y3)
```



Repeat the process to create a fourth plot. This time, plot all three lines in the same axes by calling `hold on` after plotting `y1`.

```
nexttile
plot(x,y1)
hold on
plot(x,y2)
plot(x,y3)
hold off
```

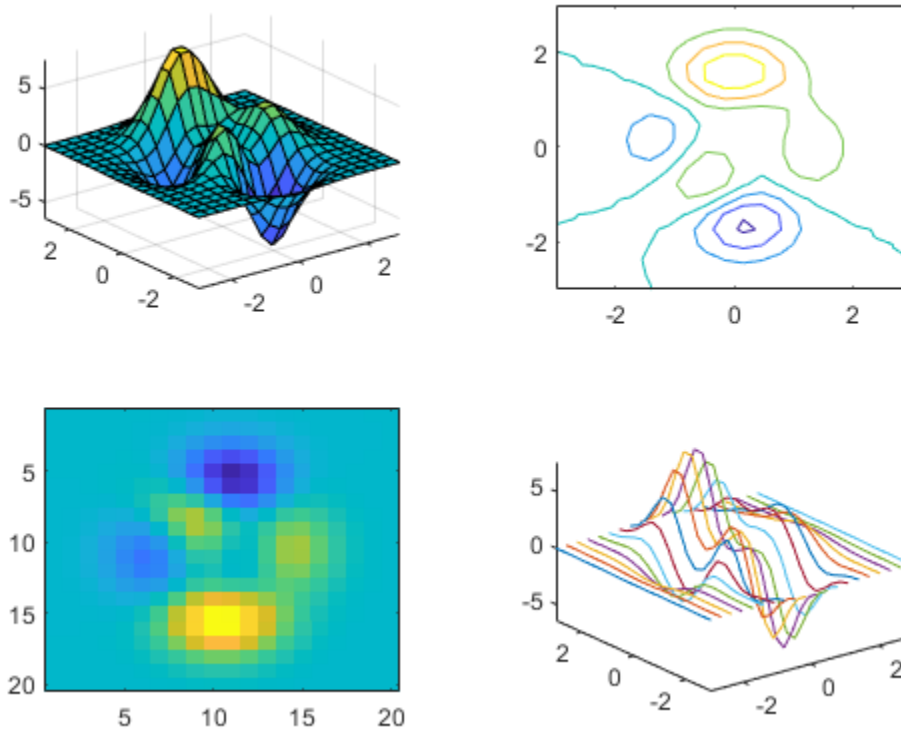


Plot Into 2-by-2 Layout

Call the `tiledlayout` function to create a 2-by-2 tiled chart layout, and call the `peaks` function to get the coordinates of a predefined surface. Create an axes object in the first tile by calling the `nexttile` function. Then call the `surf` function to plot into the axes. Repeat the process using different plotting functions for the other three tiles.

```
tiledlayout(2,2);
[X,Y,Z] = peaks(20);

% Tile 1
```

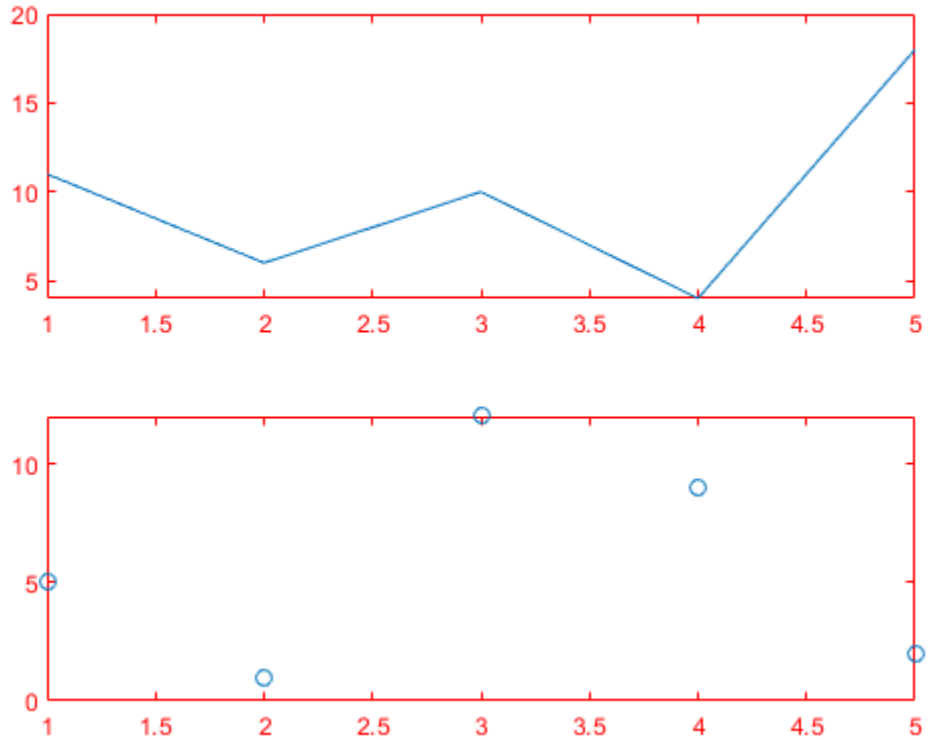
```
nexttile
surf(X,Y,Z)

% Tile 2
nexttile
contour(X,Y,Z)

% Tile 3
nexttile
imagesc(Z)

% Tile 4
nexttile
plot3(X,Y,Z)
```



Set Properties on the Axes

Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function with an output argument to store the axes. Then plot into the axes, and set the *x*- and y-axis colors to red. Repeat the process in the second tile.

```
t = tiledlayout(2,1);

% First tile
ax1 = nexttile;
plot([1 2 3 4 5],[11 6 10 4 18]);
ax1.XColor = [1 0 0];
ax1.YColor = [1 0 0];

% Second tile
ax2 = nexttile;
plot([1 2 3 4 5],[5 1 12 9 2],'o');
ax2.XColor = [1 0 0];
ax2.YColor = [1 0 0];
```

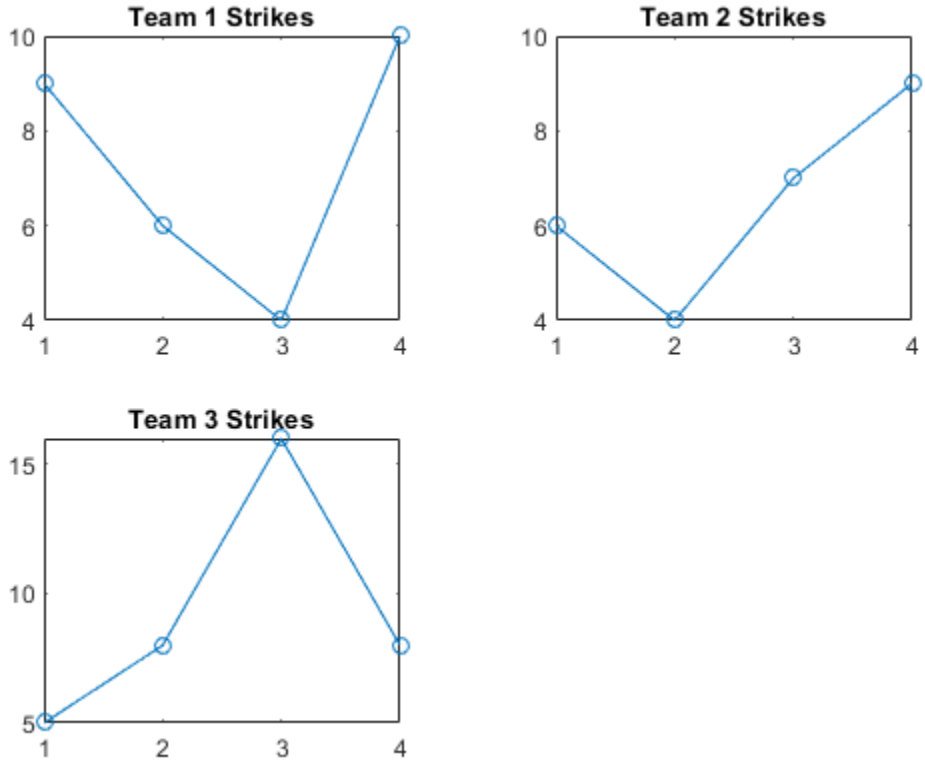


Create Axes That Span Multiple Rows and Columns

Define `scores` and `strikes` as vectors containing bowling league data over four games. Then create a tiled chart layout and display three plots showing the number of strikes for each team.

```
scores = [444 460 380
          387 366 500
          365 451 611
          548 412 452];
```

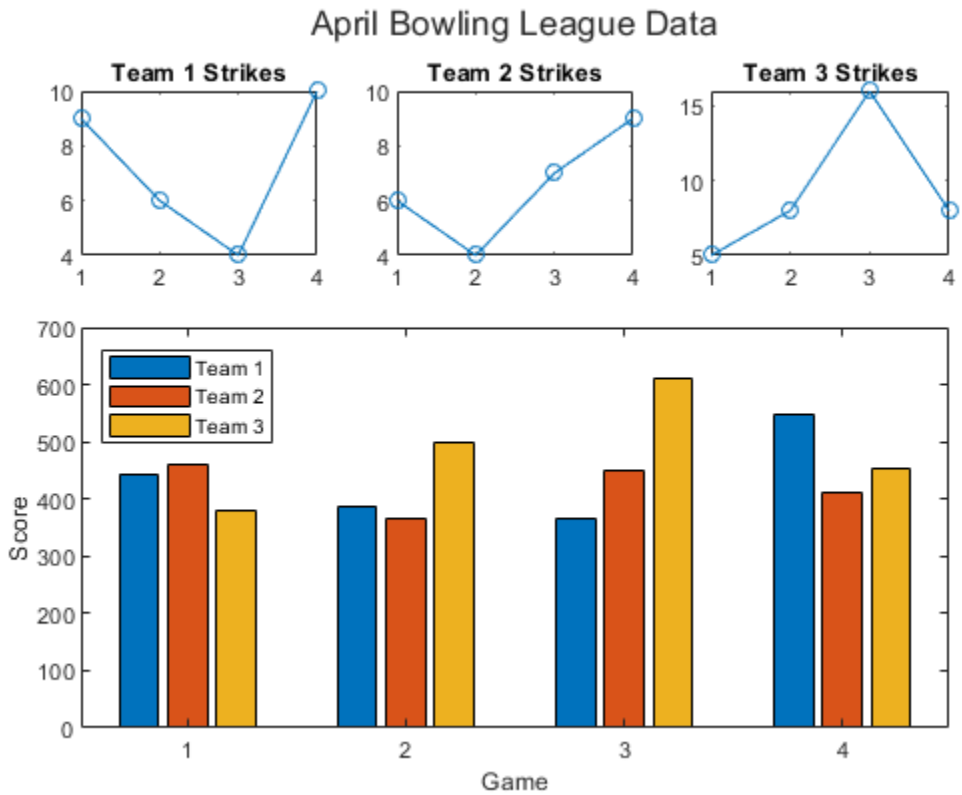
```
strikes = [9  6  5
           6  4  8
           4  7  16
           10 9  8];

t = tiledlayout('flow');

% Team 1
nexttile
plot([1 2 3 4],strikes(:,1),'-o')
title('Team 1 Strikes')

% Team 2
nexttile
plot([1 2 3 4],strikes(:,2),'-o')
title('Team 2 Strikes')

% Team 3
nexttile
plot([1 2 3 4],strikes(:,3),'-o')
title('Team 3 Strikes')
```



Call the `nexttile` function to create an axes object that spans two rows by three columns. Then display a bar graph in the axes with a legend, and configure the axis tick values and labels. Call the `title` function to add a tile to the layout.

```
nexttile([2 3]);
bar([1 2 3 4],scores)
legend('Team 1','Team 2','Team 3','Location','northwest')

% Configure ticks and axis labels
xticks([1 2 3 4])
xlabel('Game')
ylabel('Score')
```

```
% Add layout title
title(t,'April Bowling League Data')
```

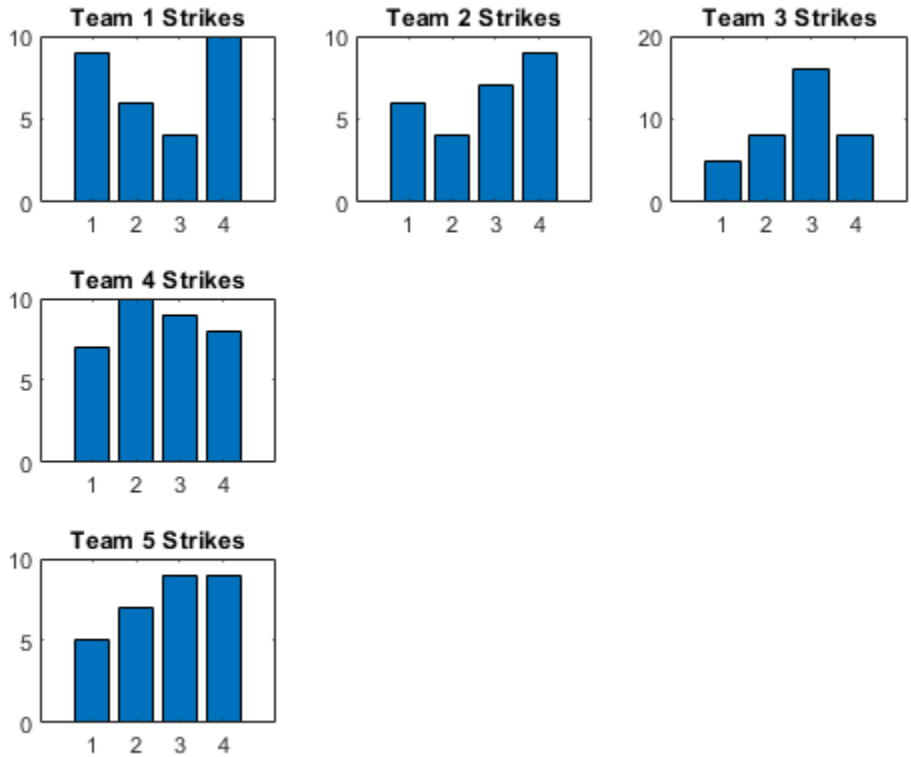


Span Axes at Specific Tile Number

To span an axes object from a specific location, specify the tile number and the span value.

Define `scores` and `strikes` as vectors containing bowling league data over four games. Then create a 3-by-3 tiled chart layout and display five bar graphs showing the number of strikes for each team.

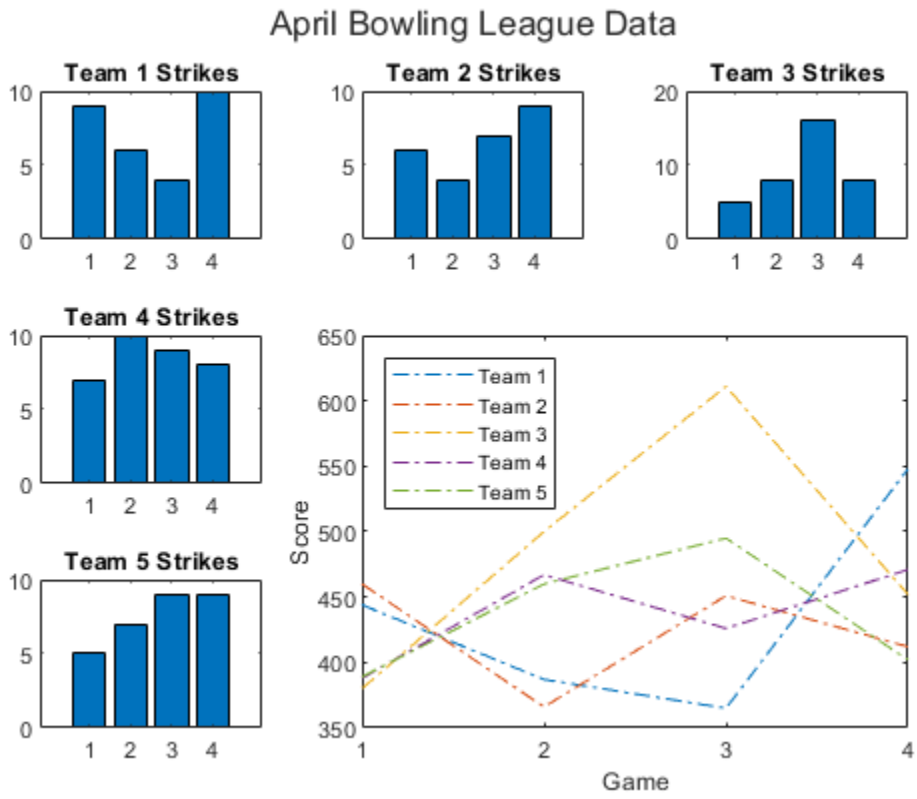
```
scores = [444 460 380 388 389
          387 366 500 467 460
          365 451 611 426 495
          548 412 452 471 402];

strikes = [9  6  5  7  5
           6  4  8 10  7
           4  7 16  9  9
           10  9  8  8  9];

t = tiledlayout(3,3);

% Team 1
nexttile
bar([1 2 3 4],strikes(:,1))
title('Team 1 Strikes')

% Team 2
nexttile
bar([1 2 3 4],strikes(:,2))
title('Team 2 Strikes')

% Team 3
nexttile
bar([1 2 3 4],strikes(:,3))
title('Team 3 Strikes')

% Team 4
nexttile
bar([1 2 3 4],strikes(:,4))
title('Team 4 Strikes')

% Team 5
nexttile(7)
bar([1 2 3 4],strikes(:,5))
title('Team 5 Strikes')
```



Display a larger plot with a legend. Call the `nexttile` function to place the upper left corner of the axes in the fifth tile, and span the axes across two rows by two columns of tiles. Plot the scores for all the teams. Configure the *x*-axis to display four ticks, and add labels to each axis. Then add a shared title at the top of the layout.

```
nexttile(5,[2 2]);
plot([1 2 3 4],scores,'-.')
labels = {'Team 1','Team 2','Team 3','Team 4','Team 5'};
legend(labels,'Location','northwest')

% Configure ticks and axis labels
xticks([1 2 3 4])
xlabel('Game')
ylabel('Score')
```

```
% Add layout title
title(t,'April Bowling League Data')
```

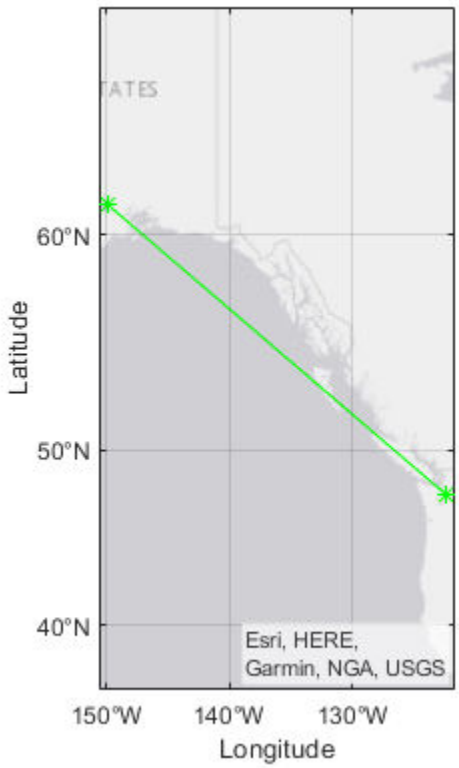


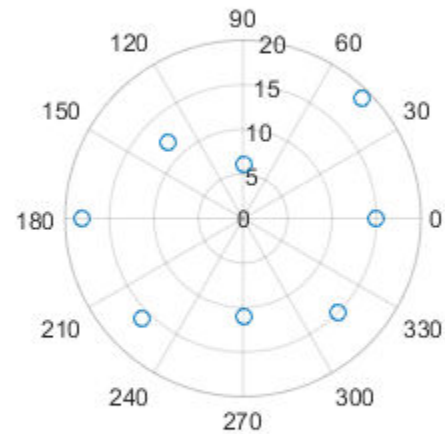
Display Polar and Geographic Plots

Create a 1-by-2 tiled chart layout. In the first tile, display a geographic plot containing a line that connects two cities on a map. In the second tile, create a scatter plot in polar coordinates.

```
tiledlayout(1,2)
```

```
% Display geographic plot
nexttile
geoplot([47.62 61.20],[-122.33 -149.90],'g-*')

% Display polar plot
nexttile
theta = pi/4:pi/4:2*pi;
rho = [19 6 12 18 16 11 15 15];
polarscatter(theta,rho)
```



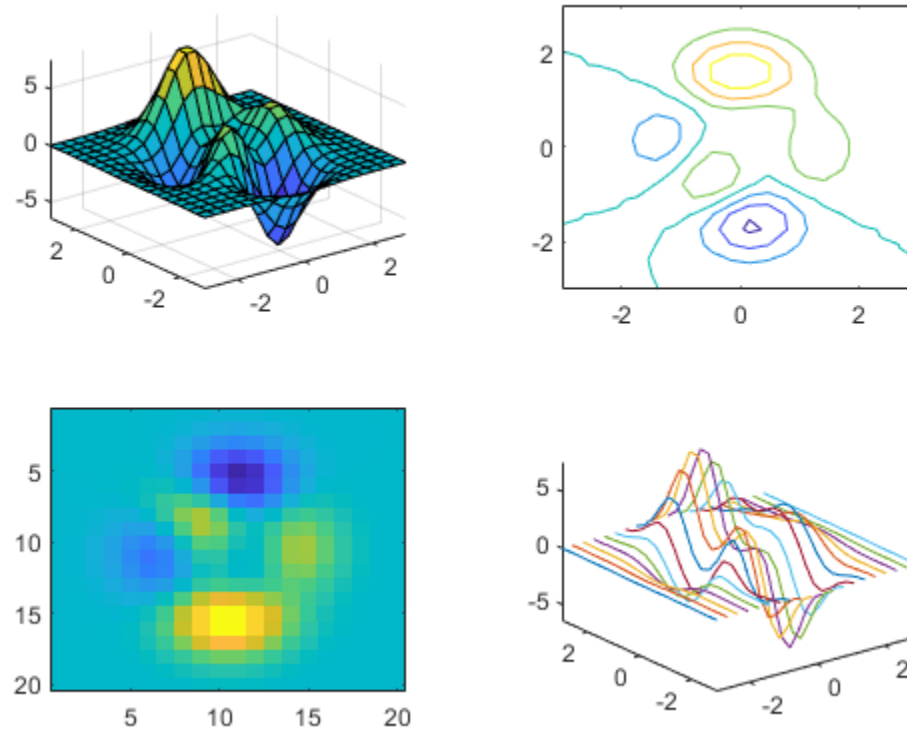


Reconfigure Content in Previous Tile

One of the ways that the `nexttile` output argument is useful is when you want to adjust the content in a previous tile. For example, you might decide to reconfigure the colormap used in a previous plot.

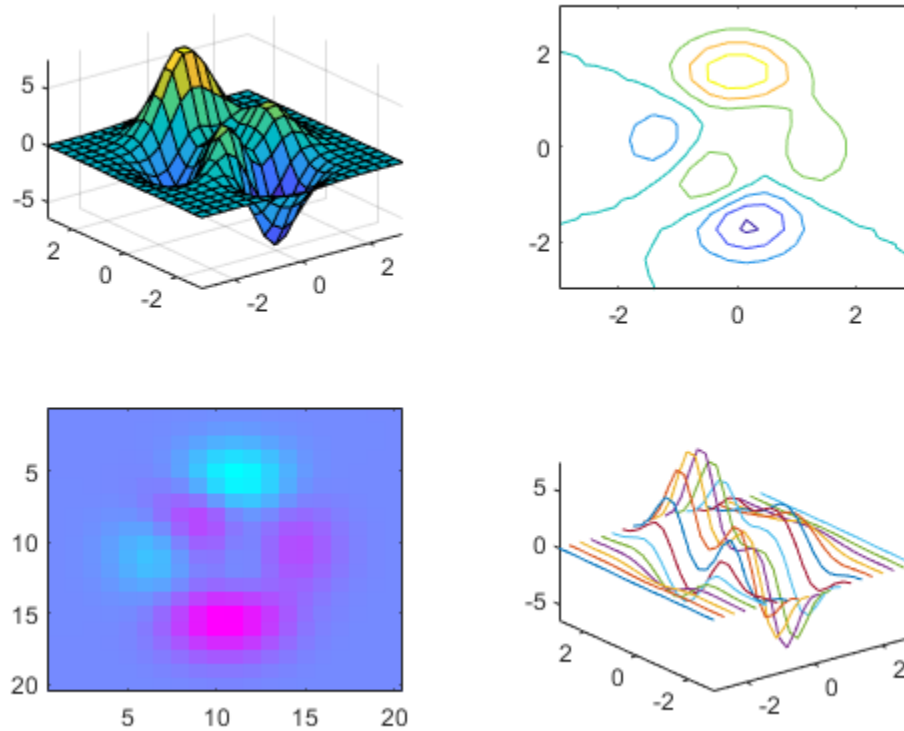
Create a 2-by-2 tiled chart layout. Call the `peaks` function to get the coordinates for a predefined surface. Then create a different plot of the surface in each tile.

```
tiledlayout(2,2);
[X,Y,Z] = peaks(20);

% Tile 1
nexttile
surf(X,Y,Z)

% Tile 2
nexttile
contour(X,Y,Z)

% Tile 3
nexttile
imagesc(Z)

% Tile 4
nexttile
plot3(X,Y,Z)
```



To change the colormap in the third tile, get the axes in that tile. Call the `nexttile` function by specifying the tile number, and return the axes output argument. Then pass the axes to the `colormap` function.

```
ax = nexttile(3);
colormap(ax,cool)
```

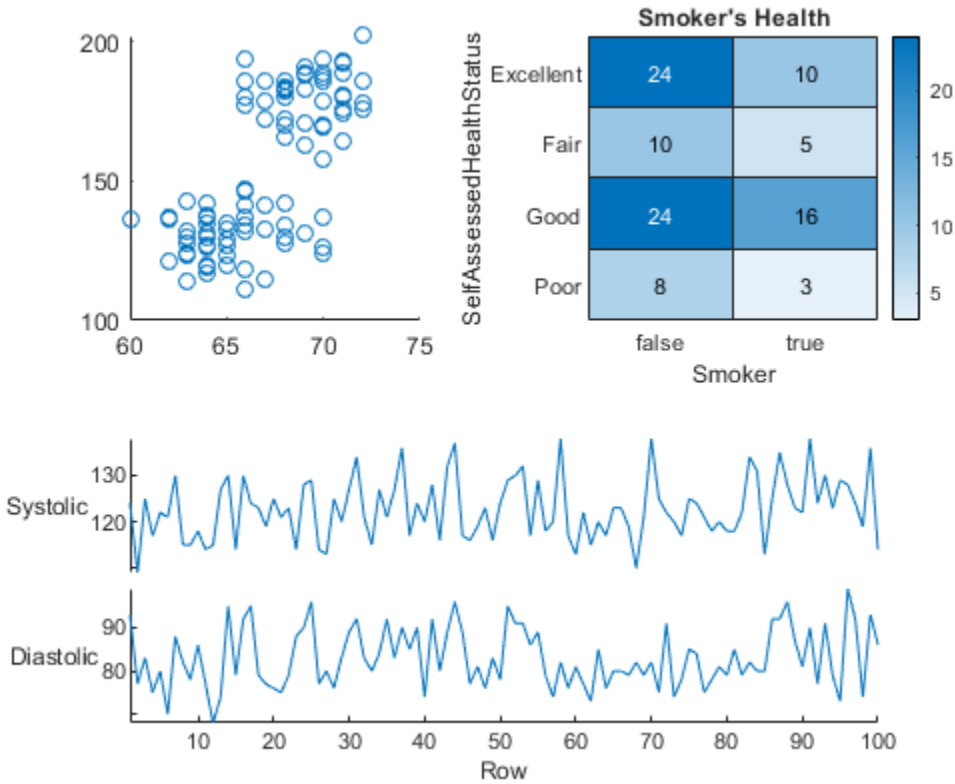


Replace Content in Previous Tile

Load the `patients` data set and create a table from a subset of the variables. Then create a 2-by-2 tiled chart layout. Display a scatter plot in the first tile, a heatmap in the second tile, and a stacked plot across the bottom two tiles.

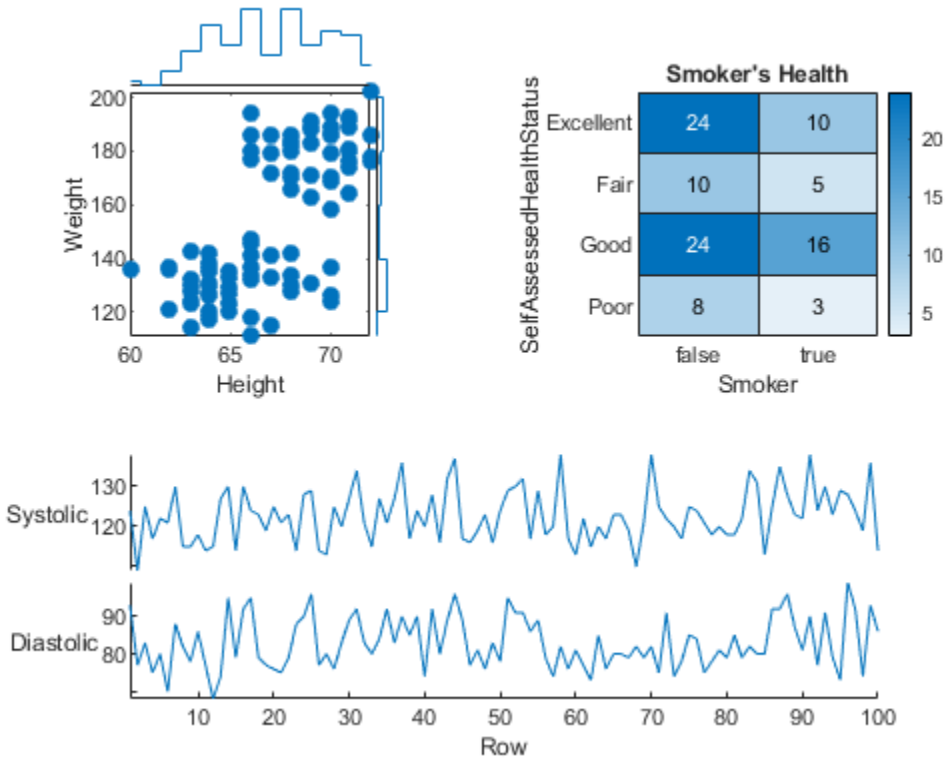
```
load patients
tbl = table(Diastolic,Smoker,Systolic,Height,Weight,SelfAssessedHealthStatus);
tiledlayout(2,2)

% Scatter plot
```

```
nexttile
scatter(tbl.Height,tbl.Weight)

% Heatmap
nexttile
heatmap(tbl,'Smoker','SelfAssessedHealthStatus','Title','Smoker''s Health');

% Stacked plot
nexttile([1 2])
stackedplot(tbl,{'Systolic','Diastolic'});
```



Call `nexttile`, and specify the tile number as `1` to make the axes in that tile the current axes. Replace the contents of that tile with a scatter histogram.

```
nexttile(1)
scatterhistogram(tbl,'Height','Weight');
```

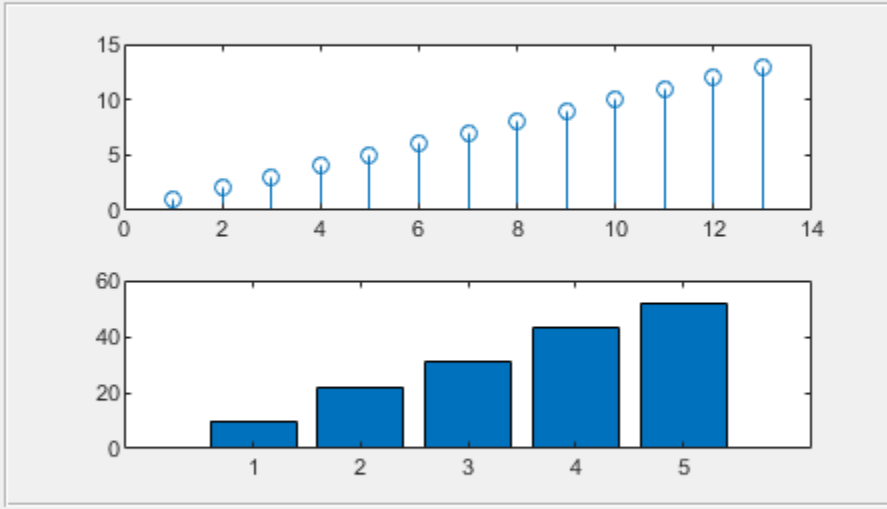


Display Layout in a Panel

Create a panel in a figure. Then create a tiled chart layout `t` in the panel by specifying the panel object as the first argument to the `tiledlayout` function. By default, `nexttile` looks for the layout in the figure. However the layout is in a panel instead of a figure, so you must specify `t` as an input argument when you call `nexttile`.

```
p = uipanel('Position',[.1 .2 .8 .6]);
t = tiledlayout(p,2,1);
```

```
% Tile 1
nexttile(t)
stem(1:13)

% Tile 2
nexttile(t)
bar([10 22 31 43 52])
```

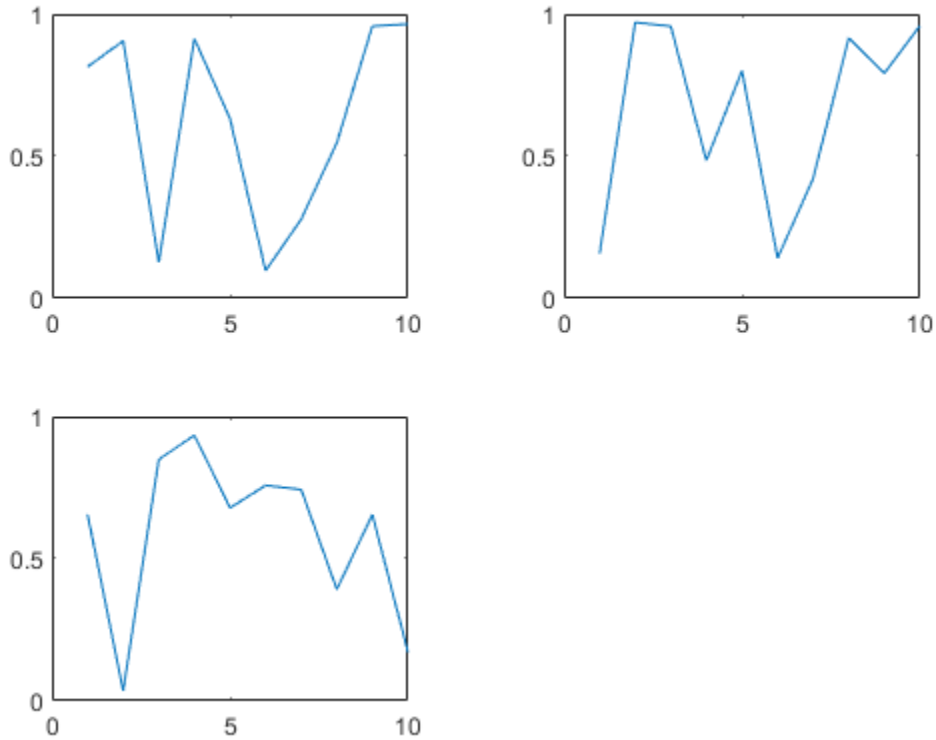


Create and Configure Axes Manually

Occasionally, you might need to create the axes manually before calling a plotting function. When you create the axes, specify the `parent` argument as the tiled chart layout. Then position the axes by setting the `Layout` property on the axes.

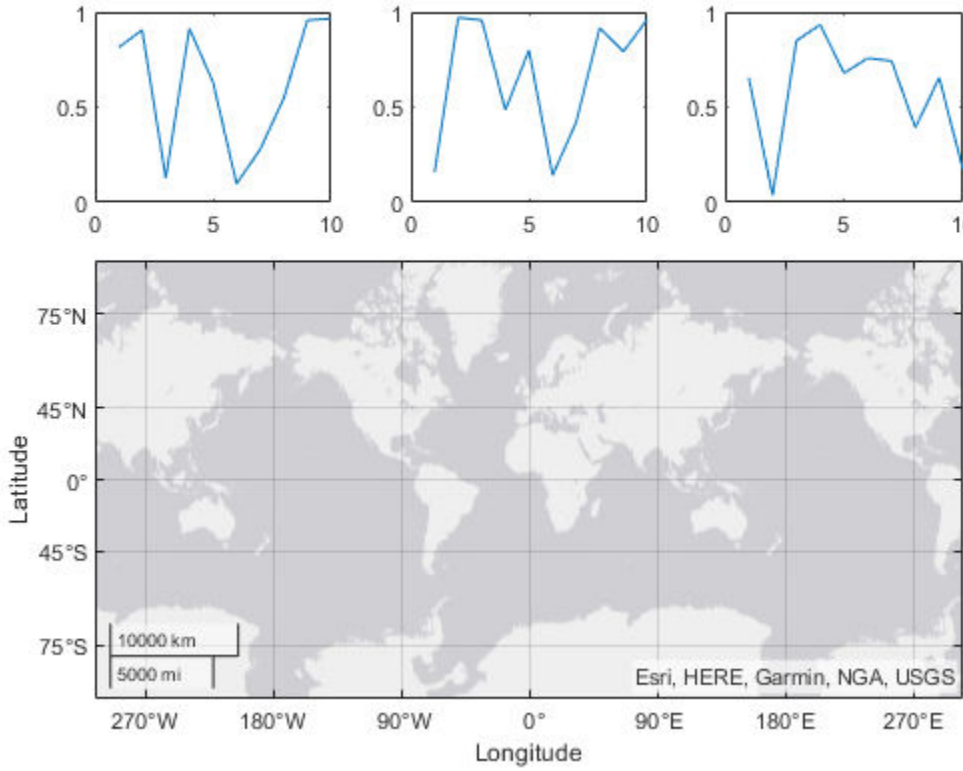
Create a tiled chart layout `t` and specify the `'flow'` tile arrangement. Display a plot in each of the first three tiles.

```
t = tiledlayout('flow');
nexttile
plot(rand(1,10));
nexttile
plot(rand(1,10));
nexttile
plot(rand(1,10));
```



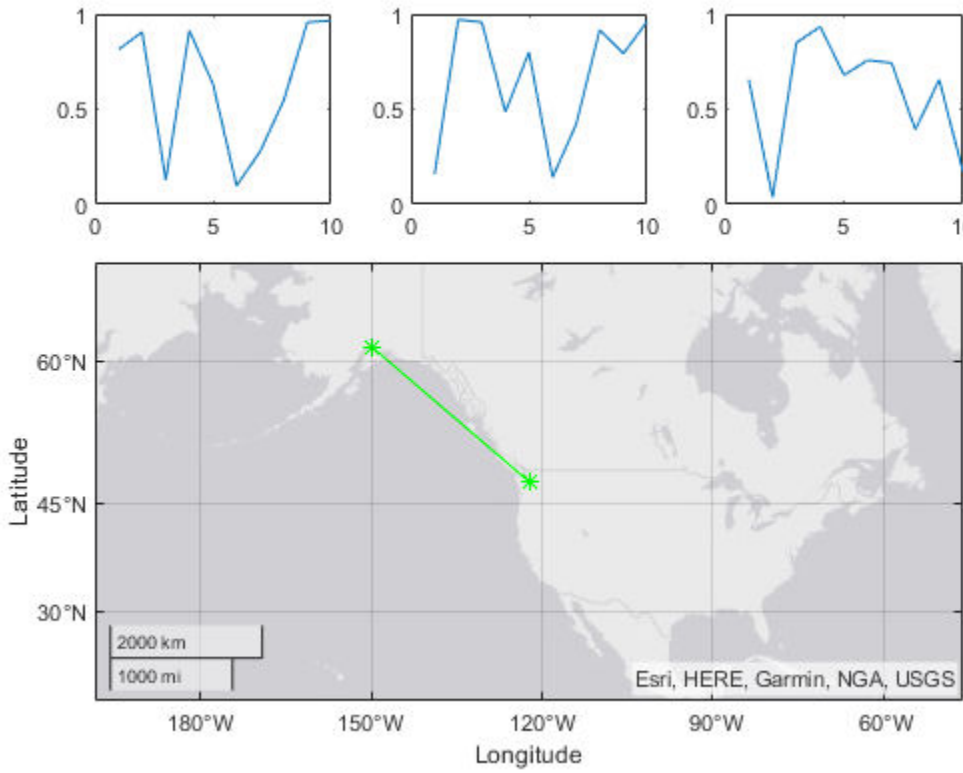
Create a geographic axes object `gax` by calling the `geoaxes` function and specify `t` as the parent argument. By default, the axes goes into the first tile, so move it to the fourth tile by setting `gax.Layout.Tile` to 4. Span the axes across a 2-by-3 region of tiles by setting `gax.Layout.TileSpan` to `[2 3]`.

```
gax = geoaxes(t);
gax.Layout.Tile = 4;
gax.Layout.TileSpan = [2 3];
```



Call the `geoplot` function. Then configure the map center and zoom level for the axes.

```
geoplot(gax,[47.62 61.20],[-122.33 -149.90],'g-*')
gax.MapCenter = [47.62 -122.33];
gax.ZoomLevel = 2;
```



Input Arguments

`tilenum` — Tile number
positive whole number

Tile number, specified as a positive whole number. The tile numbers start at 1 and increase from left-to-right and top-to-bottom.

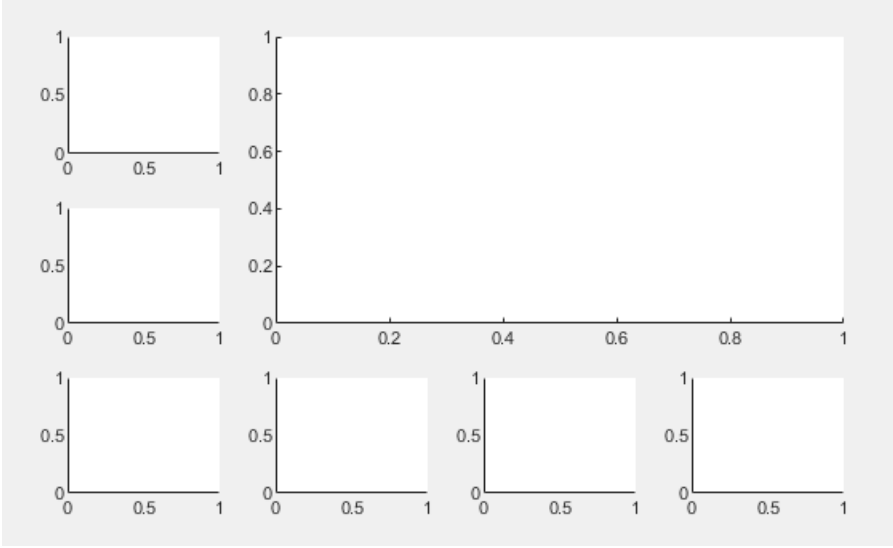
If the tile is empty, `nexttile` places an axes object into that tile. If the tile contains an axes or chart object, then that object becomes the current axes, so the next plotting command can plot into it.

span — Tile span
`[1 1]` (default) | `[r c]`

Tile span, specified as a vector of the form `[r c]`, where `r` and `c` are positive whole numbers. Use this argument to make the axes span `r` rows by `c` columns of tiles in the layout.

If you specify the `span` argument without the `tilenum` argument, `nexttile` places the upper left corner of the axes in the upper left corner of the first empty `r`-by-`c` region in the layout.

However, if you specify both the `tilenum` and `span` arguments, `nexttile` places the upper left corner of the axes in the upper left corner of the tile specified by `tilenum`. For example, the large axes on the right side of this 3-by-4 layout has a tile number of 2 and a span of `[2 3]`.



t — Tiled chart layout
`TiledChartLayout` object

`TiledChartLayout` object to place the axes into. This argument is useful when you are working with multiple layouts, or when the layout is in a panel or tab instead of a figure. When you do not specify `t`, `nexttile` looks for the layout in the current figure.

See Also

Functions
`tiledlayout`

Properties
TiledChartLayout Properties

Introduced in R2019b

TiledChartLayout Properties

Tiled chart layout appearance and behavior

Description

A tiled chart layout is a container for displaying a tiling of plots in a figure. Each tile can contain an axes object for displaying a plot. By changing property values, you can modify certain aspects of the layout.

```
t = tiledlayout(2,2);
t.TileSpacing = 'compact';
```

Properties

Layout

`GridSize` — Grid size

vector of the form `[m n]`

This property is read-only.

Grid size, returned as a vector of the form `[m n]`, where `m` is the number of rows and `n` is the number of columns. If the `TileArrangement` property is set to `'flow'`, then the value of this property might change as the parent container resizes or the number of axes in the layout changes.

`TileArrangement` — Tile arrangement

`'fixed'` | `'flow'`

This property is read-only.

Tile arrangement, returned as one of these values:

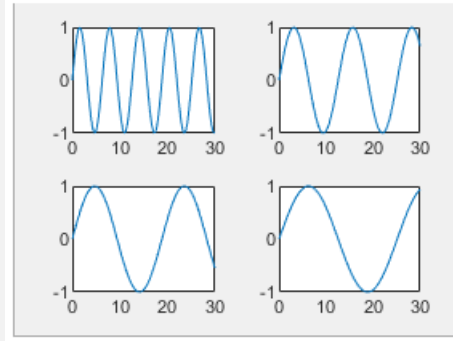
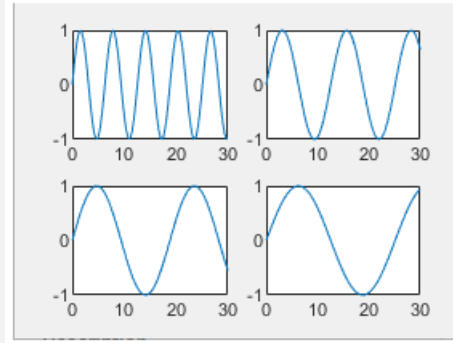
- `'fixed'` — The layout has a fixed number of rows and columns as determined by the `GridSize` property. An `m`-by-`n` layout with this tile arrangement can display up to `m*n` plots.

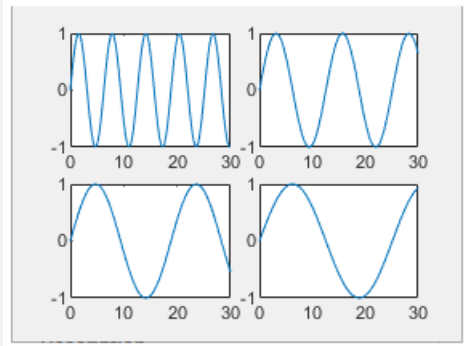
- `'flow'` — The layout has a variable number of rows and columns that can display any number of plots. Each time you call `nexttile`, the layout reflows as needed to accommodate the new axes while maintaining an aspect ratio of roughly 4:3 for all the tiles.

`TileSpacing` — Tile spacing
`'normal'` (default) | `'compact'` | `'none'`

Tile spacing, specified as `'normal'`, `'compact'`, or `'none'`. Use this property to control the spacing between the tiles. The layout provides space for all decorations, such as axis labels, regardless of the value of this property.

This table shows how each value affects the appearance of a 2-by-2 layout.

Value	Appearance
`'normal'`	
`'compact'`	

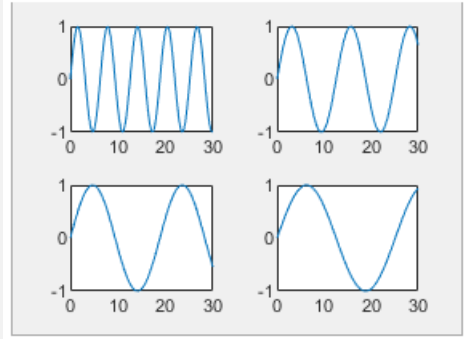
Value	Appearance
`'none'`	

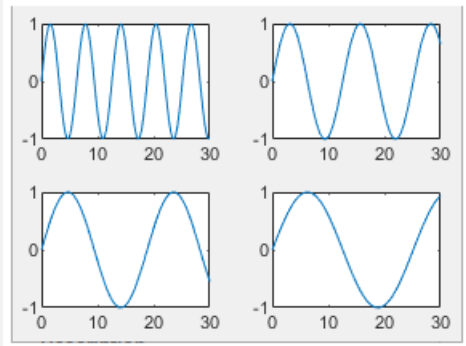
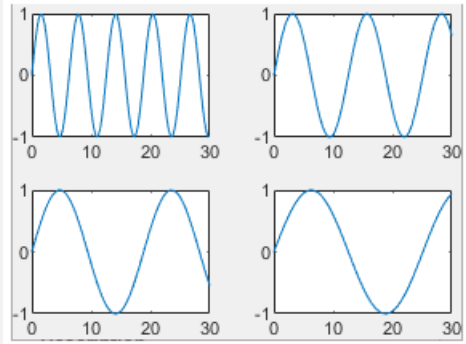
`Padding` — Padding around layout

`'normal'` (default) | `'compact'` | `'none'`

Padding around the perimeter of the layout, specified as `'normal'`, `'compact'`, or `'none'`. The layout provides space for all decorations, such as axis labels, regardless of the value of this property.

This table shows how each value affects the appearance of a 2-by-2 layout.

Value	Appearance
`'normal'`	

Value	Appearance
`'compact'`	
`'none'`	

Labels

`Title` — Text object for shared title

`Text` object

`Text` object for a shared title. To add a shared title, set the `String` property of the `Text` object. To change the title appearance, such as the font style or color, set other properties. For example, create a shared title for a 2-by-2 layout that has a bold font weight.

```
t = tiledlayout(2,2);
t.Title.String = 'My Title';
t.Title.FontWeight = 'bold';
```

Alternatively, pass the `TiledChartLayout` object to the `title` function.

```
title(t,'My Title','FontWeight','normal')
```

For a complete list of properties that you can specify, see Text.

Note The `Text` object is not contained in the `Children` property of the layout, so it cannot be returned by the `findobj` function.

`XLabel` — Text object for shared *x*-axis label

`Text` object

`Text` object for a shared *x*-axis label. To add a shared *x*-axis label, set the `String` property of the text object. To change the label appearance, such as the font size, set other properties. For example, create a shared *x*-axis label with a 14-point font for a 2-by-2 layout.

```
t = tiledlayout(2,2);
t.XLabel.String = 'My x-Axis Label';
t.XLabel.FontSize = 14;
```

Alternatively, pass the `TiledChartLayout` object to the `xlabel` function.

```
xlabel(t,'My x-Axis Label','FontSize',14)
```

For a complete list of properties you can specify, see Text.

Note The `Text` object is not contained in the axes `Children` property, so it cannot be returned by `findobj`.

`YLabel` — Text object for shared *y*-axis label

`Text` object

`Text` object for a shared *y*-axis label. To add a shared *y*-axis label, set the `String` property of the text object. To change the label appearance, such as the font size, set other properties. For example, create a shared *y*-axis label with a 14-point font for a 2-by-2 layout.

```
t = tiledlayout(2,2);
t.YLabel.String = 'My y-Axis Label';
t.YLabel.FontSize = 14;
```

Alternatively, pass the `TiledChartLayout` object to the `ylabel` function.

```
ylabel(t,'My y-Axis Label','FontSize',14)
```

For a complete list of properties you can specify, see Text.

Note The `Text` object is not contained in the axes `Children` property, so it cannot be returned by `findobj`.

Interactivity

`Toolbar` — Shared data exploration toolbar

empty `GraphicPlaceholder` array (default) | `AxesToolbar` object

Shared data exploration toolbar, specified as an `AxesToolbar` object returned by the `axtoolbar` function. The toolbar appears in the top-right corner of the of the layout when you hover over it.

By default, there is no shared toolbar, and each of the individual axes objects has its own toolbar. When you create a shared toolbar, the toolbars on the individual axes objects become hidden. If you do not want any toolbars in the layout, leave this property empty and set the set the `Visible` property of the `AxesToolbar` object to `'off'` for each axes.

```
t = tiledlayout(2,1);
ax = nexttile;
ax.Toolbar.Visible = 'off';
```

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

Callbacks

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.

- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container
`Figure` object | `Panel` object | `Tab` object

Parent container, specified as a `Figure`, `Panel`, or a `Tab` object.

`Children` — Children
empty `GraphicsPlaceholder` array | array of graphics objects

Children, returned as an array of graphics objects. Use this property to view a list of the children or to reorder the children by setting the property to a permutation of itself.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the child graphics object to the `TiledChartLayout` object.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object
`'tiledlayout'`

This property is read-only.

Type of graphics object returned as `'tiledlayout'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`nexttile` | `tiledlayout`

Introduced in R2019b

tiledlayout

Create tiled chart layout

Syntax

```
tiledlayout(m,n)
tiledlayout('flow')
tiledlayout( ___ ,Name,Value)
tiledlayout(parent, ___ )
t = tiledlayout( ___ )
```

Description

`tiledlayout(m,n)` creates a tiled chart layout for displaying multiple plots in the current figure. The layout has a fixed `m`-by-`n` tile arrangement that can display up to `m*n` plots. If there is no figure, MATLAB creates a figure and places the layout into it. If the current figure contains an existing layout, MATLAB replaces that layout with a new one.

A tiled chart layout contains an invisible grid of tiles that covers the entire figure or parent container. Each tile can contain an axes for displaying a plot. After creating a layout, call the `nexttile` function to place an axes object into the layout. Then call a plotting function to plot into the axes.

`tiledlayout('flow')` specifies the `'flow'` tile arrangement for the layout. Initially, there is only one empty tile that fills the entire layout. As you call `nexttile`, the layout reflows as needed to accommodate the new axes while maintaining an aspect ratio of roughly 4:3 for all the tiles.

`tiledlayout( ___ ,Name,Value)` specifies additional options for the layout using one or more name-value pair arguments. Specify the options after all other input arguments. For example, `tiledlayout(2,2,'TileSpacing','compact')` creates a 2-by-2 layout with minimal spacing between the tiles. For a list of properties, see TiledChartLayout Properties.

`tiledlayout(parent, ___ )` creates the layout in the specified parent container rather than in the current figure. Specify the parent container before all other input arguments.

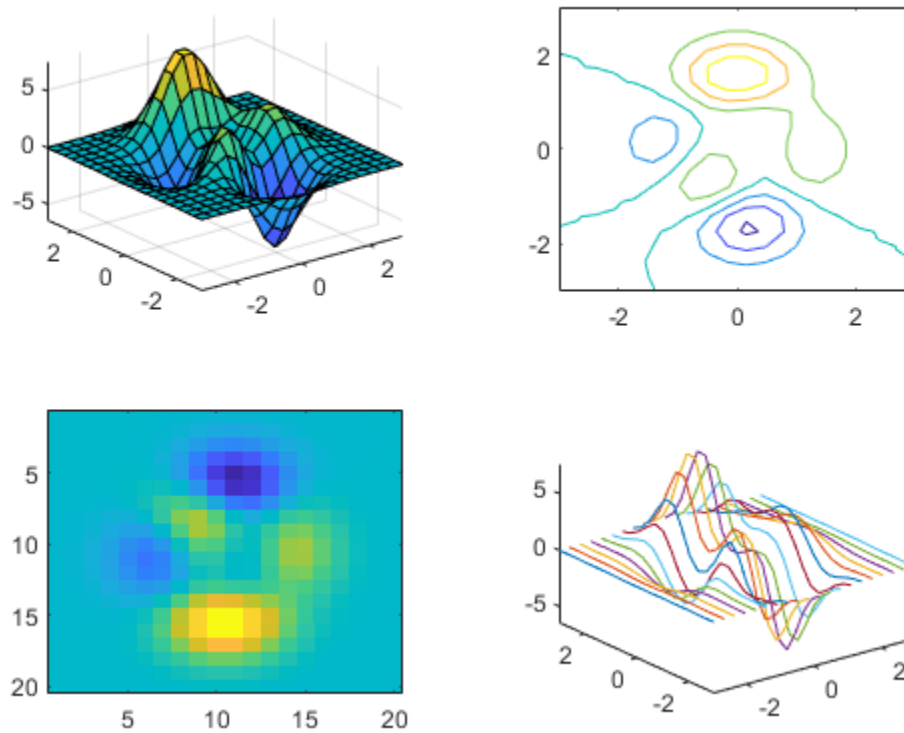
`t = tiledlayout( ___ )` returns the `TiledChartLayout` object. Use `t` to configure properties of the layout after creating it.

Examples

Create 2-by-2 Layout

Create a 2-by-2 tiled chart layout, and call the `peaks` function to get the coordinates of a predefined surface. Create an axes object in the first tile by calling the `nexttile` function. Then call the `surf` function to plot into the axes. Repeat the process using different plotting functions for the other three tiles.

```
tiledlayout(2,2);
[X,Y,Z] = peaks(20);

% Tile 1
nexttile
surf(X,Y,Z)

% Tile 2
nexttile
contour(X,Y,Z)

% Tile 3
nexttile
imagesc(Z)

% Tile 4
nexttile
plot3(X,Y,Z)
```

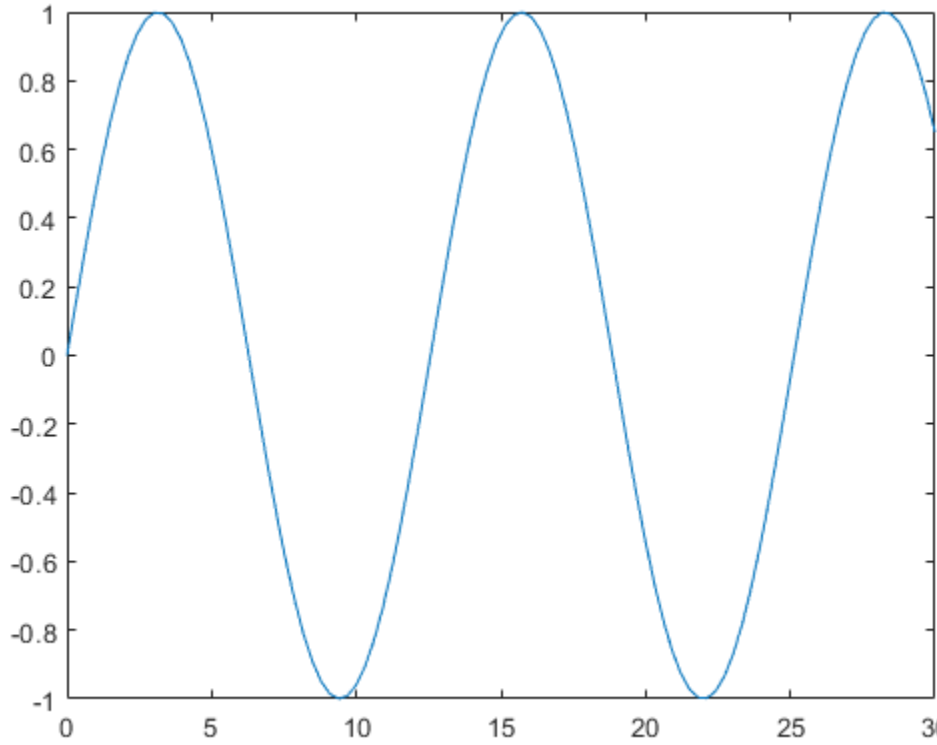


Specify Flow Tile Arrangement

Create four coordinate vectors: `x`, `y1`, `y2`, and `y3`. Call the `tiledlayout` function with the `'flow'` argument to create a tiled chart layout that can accommodate any number of axes. Call the `nexttile` function to create the first axes. Then plot `y1` in the first tile. This first plot fills the entire layout.

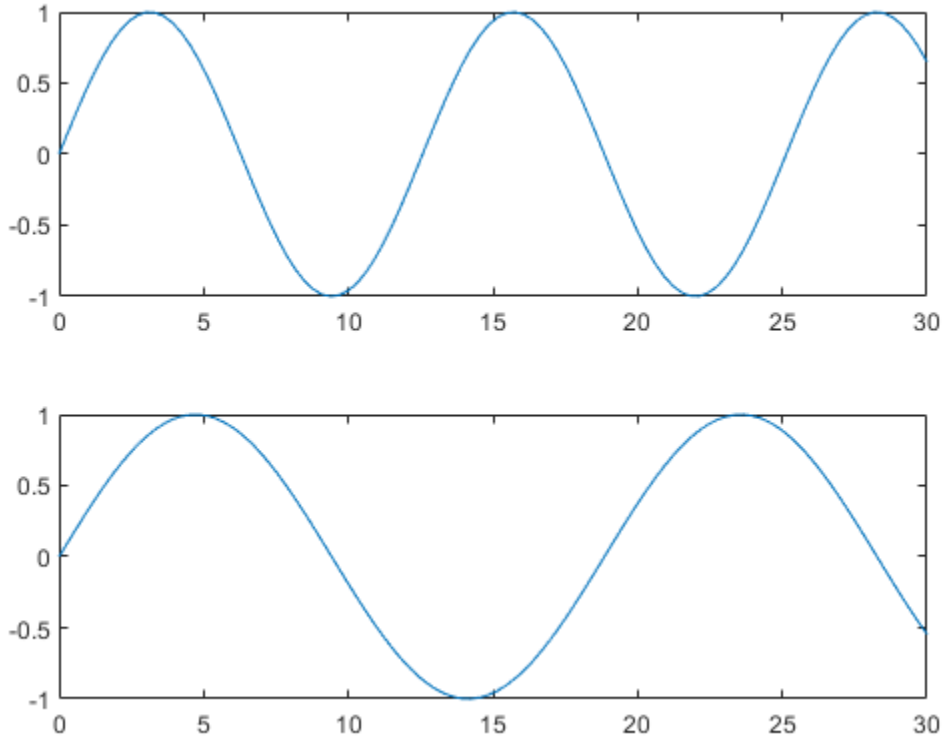
```
x = linspace(0,30);
y1 = sin(x/2);
y2 = sin(x/3);
y3 = sin(x/4);
```

```
% Plot into first tile three times
tiledlayout('flow')
nexttile
plot(x,y1)
```



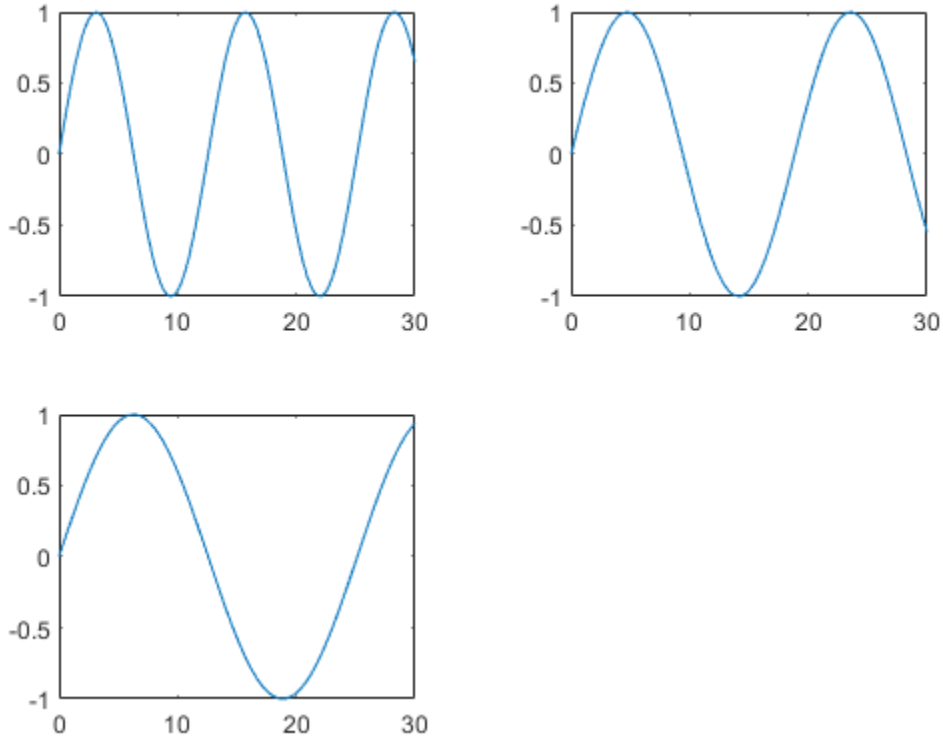
Create a second tile and axes, and plot into the axes.

```
nexttile
plot(x,y2)
```



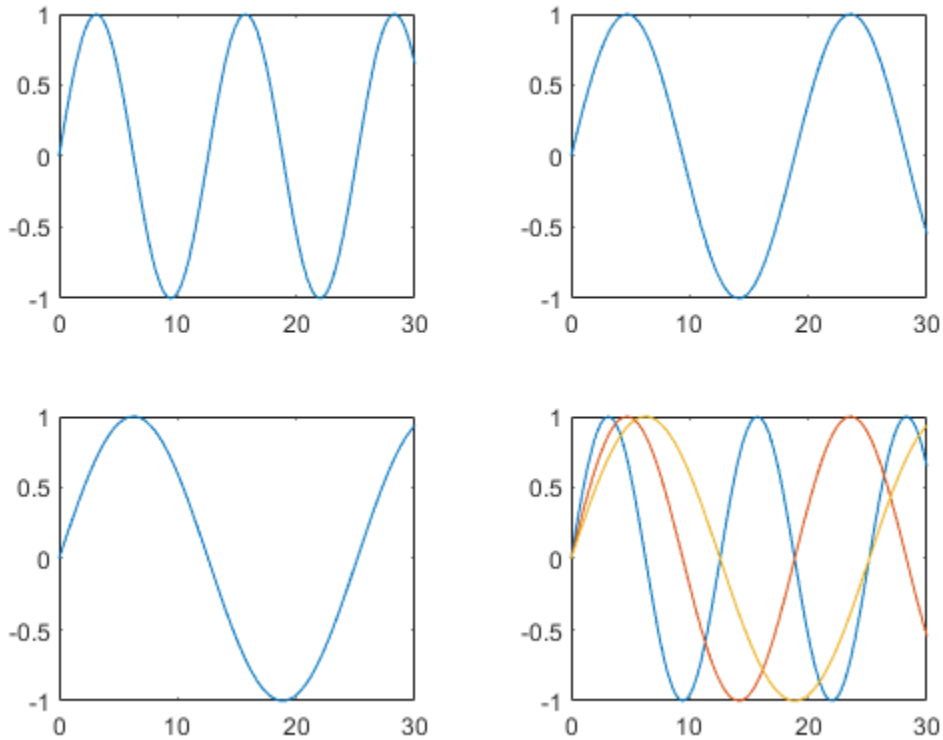
Repeat the process to create a third plot.

```
nexttile
plot(x,y3)
```



Repeat the process to create a fourth plot. This time, plot all three lines in the same axes by calling `hold on` after plotting `y1`.

```
nexttile
plot(x,y1)
hold on
plot(x,y2)
plot(x,y3)
hold off
```

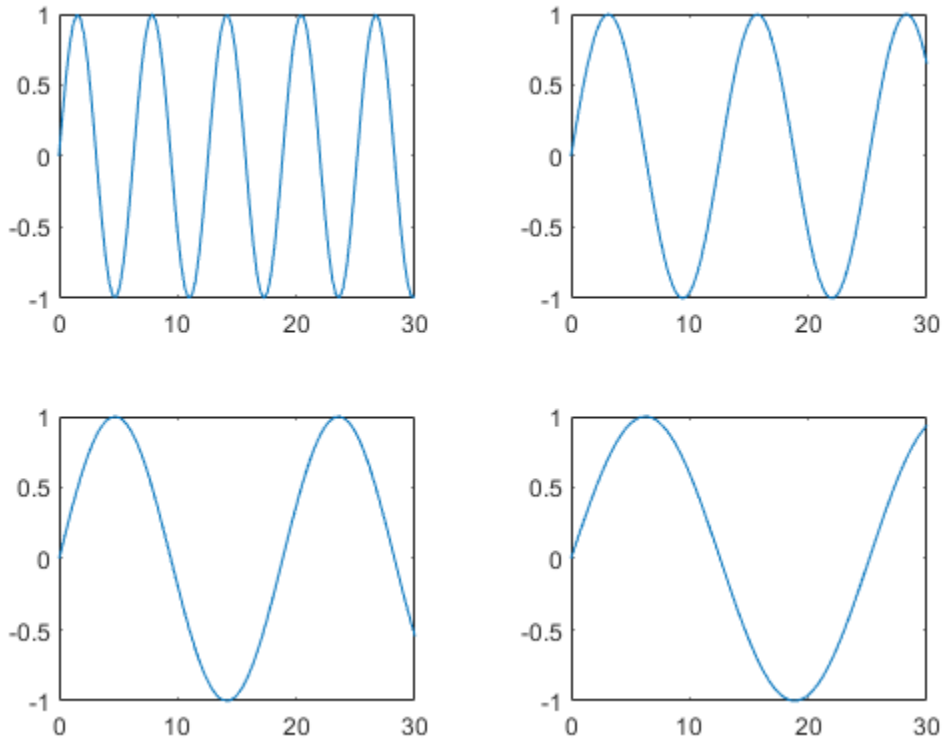


Adjust Layout Spacing

Create five coordinate vectors: `x`, `y1`, `y2`, `y3`, and `y4`. Then call the `tiledlayout` function to create a 2-by-2 layout and specify a return argument to store the `TileChartLayout` object. Call the `nexttile` function to create an axes object in the next empty tile before calling the `plot` function.

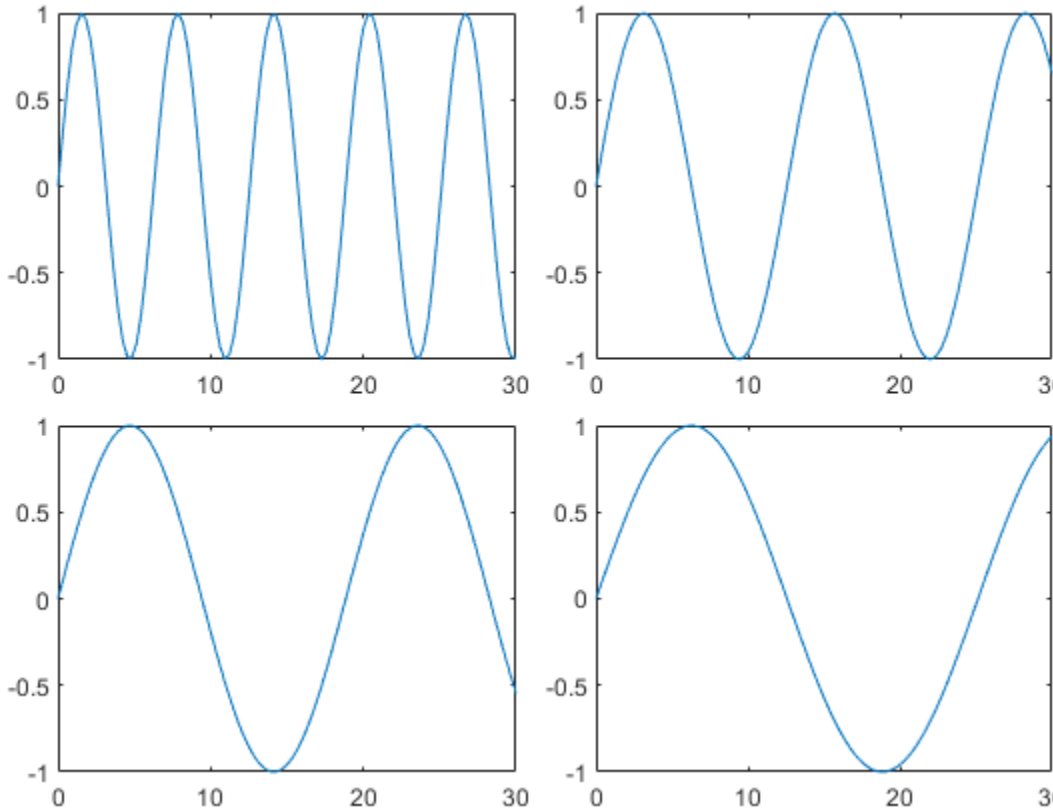
```
x = linspace(0,30);
y1 = sin(x);
y2 = sin(x/2);
y3 = sin(x/3);
```

```
y4 = sin(x/4);
t = tiledlayout(2,2);

% Tile 1
nexttile
plot(x,y1)

% Tile 2
nexttile
plot(x,y2)

% Tile 3
nexttile
plot(x,y3)

% Tile 4
nexttile
plot(x,y4)
```



Decrease the amount of space between the tiles by setting the `TileSpacing` property to `'compact'`. Then decrease the space between the edges of the layout and the edges of the figure by setting the `Padding` property to `'compact'`.

```
t.TileSpacing = 'compact';
t.Padding = 'compact';
```

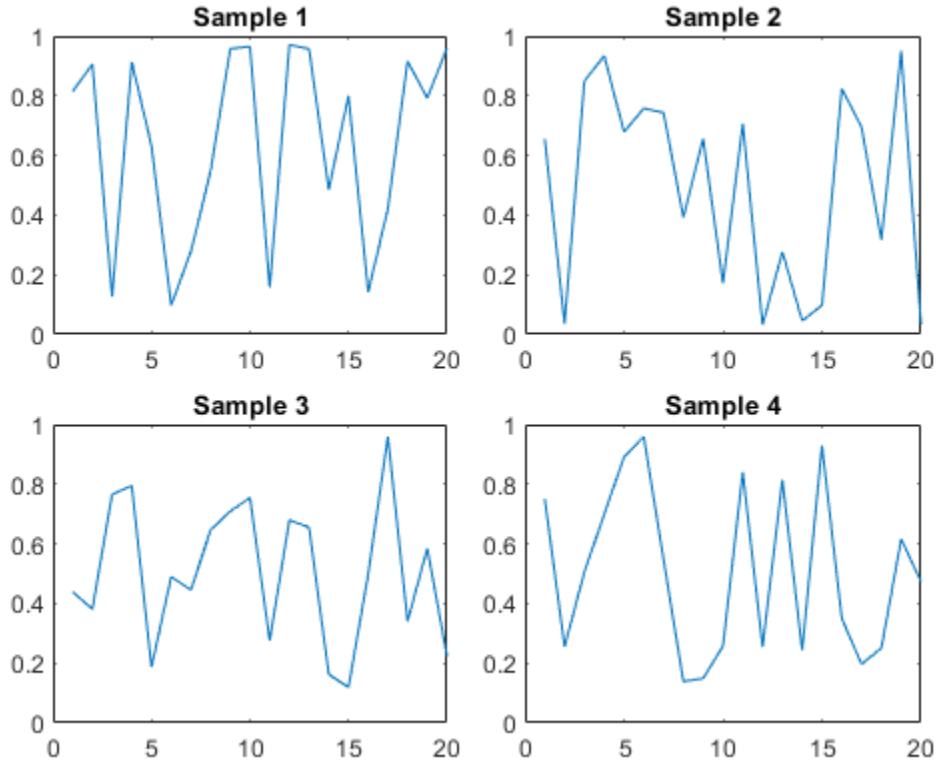


Create Shared Title and Axis Labels

Create a 2-by-2 tiled chart layout `t`. Specify the `TileSpacing` name-value pair argument to minimize the space between the tiles. Then create a titled plot in each tile.

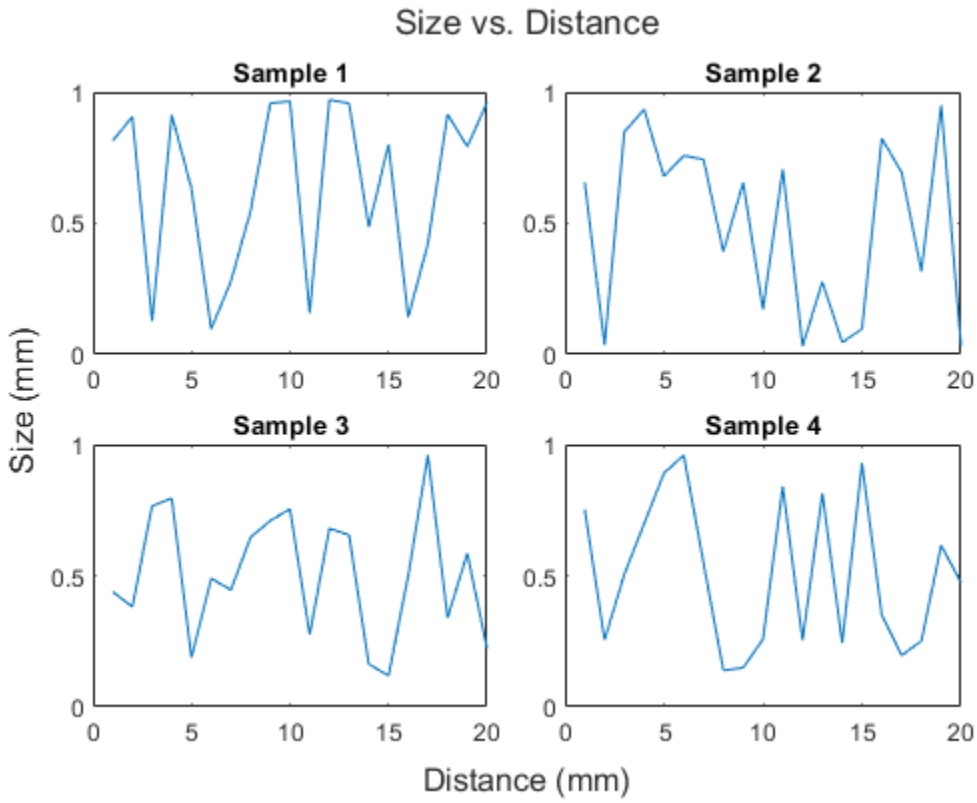
```
t = tiledlayout(2,2,'TileSpacing','Compact');

% Tile 1
nexttile
plot(rand(1,20))
title('Sample 1')
```

```
% Tile 2
nexttile
plot(rand(1,20))
title('Sample 2')

% Tile 3
nexttile
plot(rand(1,20))
title('Sample 3')

% Tile 4
nexttile
plot(rand(1,20))
title('Sample 4')
```



Display a shared title and axis labels by passing `t` to the `title`, `xlabel`, and `ylabel` functions.

```
title(t,'Size vs. Distance')
xlabel(t,'Distance (mm)')
ylabel(t,'Size (mm)')
```

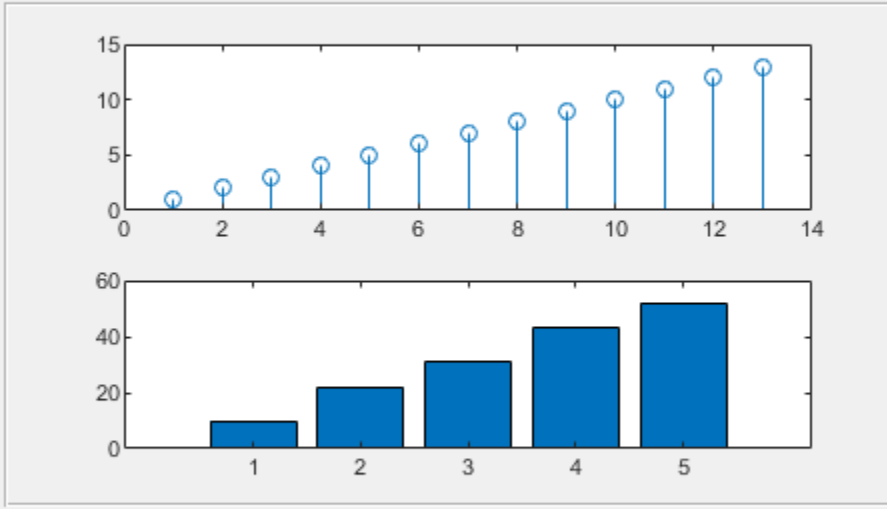


Create Layout in a Panel

Create a panel in a figure. Then create a tiled chart layout in the panel by specifying the panel object as the first argument to the `tiledlayout` function. Display a plot in each tile.

```
p = uipanel('Position',[.1 .2 .8 .6]);
t = tiledlayout(p,2,1);

% Tile 1
nexttile(t)
```

```
stem(1:13)

% Tile 2
nexttile(t)
bar([10 22 31 43 52])
```



Input Arguments

m — Number of rows
positive whole number

Number of rows, specified as a positive whole number.

Example: `tiledlayout(2,3)` creates a tiled chart layout that has two rows and three columns of tiles.

n — Number of columns
positive whole number

Number of columns, specified as a positive whole number.

Example: `tiledlayout(2,3)` creates a tiled chart layout that has two rows and three columns of tiles.

parent — Parent container
`Figure` object | `Panel` object | `Tab` object

Parent container, specified as a `Figure`, `Panel`, or `Tab` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

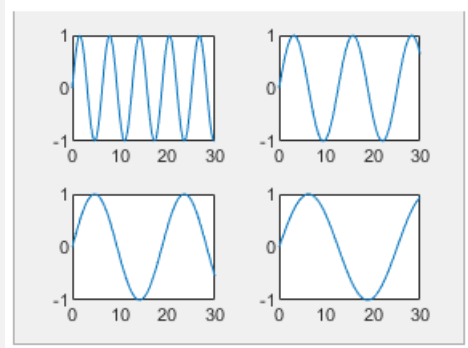
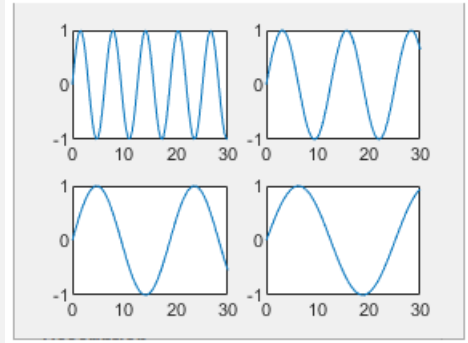
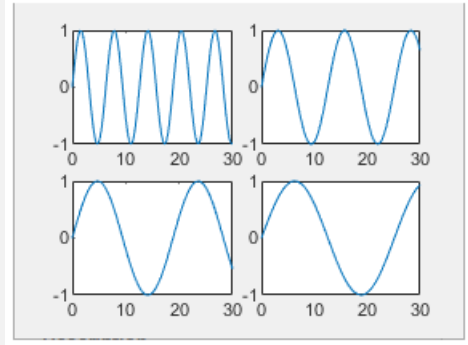
Example: `tiledlayout(2,2,'TileSpacing','compact')` creates a 2-by-2 layout that has minimal spacing between the tiles.

Note The properties listed here are only a subset. For a complete list, see TiledChartLayout Properties.

TileSpacing — Tile spacing
`'normal'` (default) | `'compact'` | `'none'`

Tile spacing, specified as `'normal'`, `'compact'`, or `'none'`. Use this property to control the spacing between the tiles. The layout provides space for all decorations, such as axis labels, regardless of the value of this property.

This table shows how each value affects the appearance of a 2-by-2 layout.

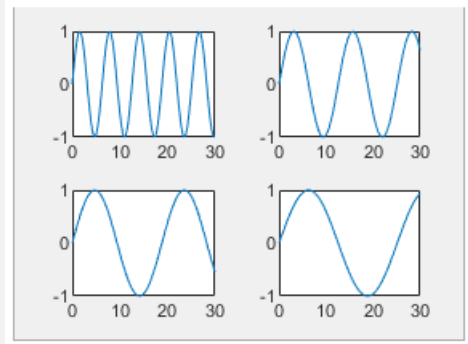
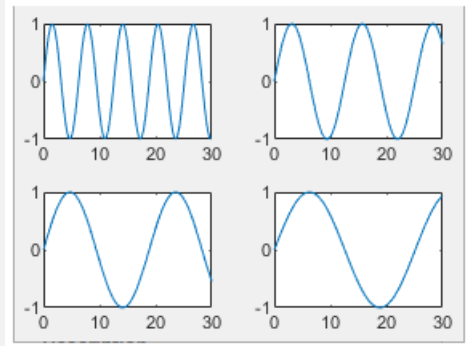
Value	Appearance
`'normal'`	
`'compact'`	
`'none'`	

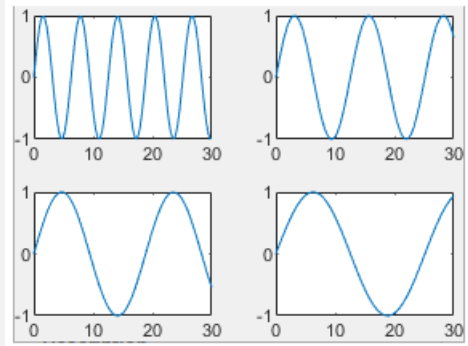
Padding — Padding around layout

`'normal'` (default) | `'compact'` | `'none'`

Padding around the perimeter of the layout, specified as `'normal'`, `'compact'`, or `'none'`. The layout provides space for all decorations, such as axis labels, regardless of the value of this property.

This table shows how each value affects the appearance of a 2-by-2 layout.

Value	Appearance
`'normal'`	
`'compact'`	

Value	Appearance
`'none'`	

See Also

Functions
`nexttile`

Properties
TiledChartLayout Properties

Introduced in R2019b

times, .*

Multiplication

Syntax

```
C = A.*B
C = times(A,B)
```

Description

`C = A.*B` multiplies arrays `A` and `B` by multiplying corresponding elements. The sizes of `A` and `B` must be the same or be compatible.

If the sizes of `A` and `B` are compatible, then the two arrays implicitly expand to match each other. For example, if one of `A` or `B` is a scalar, then the scalar is combined with each element of the other array. Also, vectors with different orientations (one row vector and one column vector) implicitly expand to form a matrix.

`C = times(A,B)` is an alternate way to execute `A.*B`, but is rarely used. It enables operator overloading for classes.

Examples

Multiply Two Vectors

Create two vectors, `A` and `B`, and multiply them element by element.

```
A = [1 0 3];
B = [2 3 7];
C = A.*B
```

```
C = 1×3

     2     0    21
```

Multiply Two Arrays

Create two 3-by-3 arrays, A and B, and multiply them element by element.

```
A = [1 0 3; 5 3 8; 2 4 6];
B = [2 3 7; 9 1 5; 8 8 3];
C = A.*B
```

```
C = 3×3

     2     0    21
    45     3    40
    16    32    18
```

Multiply Row and Column Vectors

Create a row vector a and a column vector b, then multiply them. The 1-by-3 row vector and 4-by-1 column vector combine to produce a 4-by-3 matrix.

```
a = 1:3;
b = (1:4)';
a.*b
```

```
ans = 4×3

     1     2     3
     2     4     6
     3     6     9
     4     8    12
```

The result is a 4-by-3 matrix, where each (i,j) element in the matrix is equal to `a(j).*b(i)`:

$$a = \begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix}, \qquad b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}, \qquad a\ .*\ b = \begin{bmatrix} a_1b_1 & a_2b_1 & a_3b_1 \\ a_1b_2 & a_2b_2 & a_3b_2 \\ a_1b_3 & a_2b_3 & a_3b_3 \\ a_1b_4 & a_2b_4 & a_3b_4 \end{bmatrix}.$$

Input Arguments

A, B — Operands
scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. A and B must either be the same size or have sizes that are compatible (for example, A is an M-by-N matrix and B is a scalar or 1-by-N row vector). For more information, see “Compatible Array Sizes for Basic Operations”.

- Operands with an integer data type cannot be complex.
- If A and B are datetime, duration, or calendar duration arrays, then they must be the same size unless one is a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `categorical` | `duration` | `calendarDuration`
Complex Number Support: Yes

Compatibility Considerations

Implicit expansion change affects arguments for operators

Behavior changed in R2016b

Starting in R2016b with the addition of implicit expansion, some combinations of arguments for basic operations that previously returned errors now produce results. For example, you previously could not add a row and a column vector, but those operands are now valid for addition. In other words, an expression like `[1 2] + [1; 2]` previously returned a size mismatch error, but now it executes.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a `try/catch` block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see “Compatible Array Sizes for Basic Operations”.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Multiplication of pure imaginary numbers by non-finite numbers might not match MATLAB. The code generator does not specialize multiplication by pure imaginary numbers—it does not eliminate calculations with the zero real part. For example, `(Inf + 1i)*1i = (Inf*0 - 1*1) + (Inf*1 + 1*0)i = NaN + Infi`.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Multiplication of pure imaginary numbers by non-finite numbers might not match MATLAB. The code generator does not specialize multiplication by pure imaginary numbers—it does not eliminate calculations with the zero real part. For example, `(Inf + 1i)*1i = (Inf*0 - 1*1) + (Inf*1 + 1*0)i = NaN + Infi`.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- 64-bit integers are not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`mtimes`

Topics

“Combine Categorical Arrays Using Multiplication”
“Array vs. Matrix Operations”
“Operator Precedence”
“MATLAB Operators and Special Characters”

Introduced before R2006a

timerange

Time range for timetable row subscripting

Syntax

```
S = timerange(startTime,endTime)
S = timerange(startTime,endTime,intervalType)
```

```
S = timerange(startPeriod,endPeriod,unitOfTime)
S = timerange(timePeriod,unitOfTime)
```

Description

`S = timerange(startTime,endTime)` creates a subscript to select rows of a timetable within a range of times. `S` selects all rows whose times are in the time interval specified by `startTime` and `endTime`, including `startTime` but not `endTime`. In other words, the time interval is a half-open interval. `startTime` and `endTime` are datetime or duration scalars, or character vectors or strings that specify dates and times.

If `startTime` and `endTime` are datetime values, then you only can use `S` to subscript into a timetable whose row times are datetime values. Similarly, if `startTime` and `endTime` are duration values, then you only can use `S` to subscript into a timetable whose row times are duration values.

`S = timerange(startTime,endTime,intervalType)` creates a subscript over the type of interval specified by `intervalType`. For example, if `intervalType` is `'closed'`, then `S` includes both `startTime` and `endTime` in the time range it specifies.

`S = timerange(startPeriod,endPeriod,unitOfTime)` creates a subscript over the time period between `startPeriod` and `endPeriod`, using the date or time component specified by `unitOfTime`. For example, if `unitOfTime` is `'months'`, then `S` includes the start of the month that is a component of `startPeriod`, and the end of the month of `endPeriod`. The inputs `startPeriod` and `endPeriod` can be datetime scalars, or if text, they can be character vectors or strings that specify dates and times.

`S = timerange(timePeriod,unitOfTime)` creates a subscript spanning the beginning and end of `timePeriod`, using the date or time component specified by

`unitOfTime`. For example, if `unitOfTime` is `'day'`, then `S` includes the start and end of the day that is a component of `timePeriod`.

Examples

Select Timetable Rows in Specified Time Interval

Create a timetable that contains times along with measurements of temperature, pressure, and wind speed and direction. Select rows whose times fall within a specified time interval.

```
Time = datetime({'12/18/2015 08:00:00';'12/18/2015 10:00:0';'12/18/2015 12:00:00';...
                 '12/18/2015 14:00:00';'12/18/2015 16:00:00';'12/18/2015 18:00:00'});
Temp = [37.3;39.1;42.3;45.7;41.2;39.9];
Pressure = [30.1;30.03;29.9;29.8;30.0;29.9];
WindSpeed = [13.4;6.5;7.3;8.5;9.2;4.3];
WindDirection = categorical({'NW';'N';'NW';'NW';'NNW';'N'});
TT = timetable(Time,Temp,Pressure,WindSpeed,WindDirection)
```

```
TT=6×5 timetable
            Time            Temp    Pressure    WindSpeed    WindDirection
    ____________________    ____    ________    _________    _____________

    18-Dec-2015 08:00:00    37.3      30.1        13.4            NW
    18-Dec-2015 10:00:00    39.1     30.03         6.5            N
    18-Dec-2015 12:00:00    42.3      29.9         7.3            NW
    18-Dec-2015 14:00:00    45.7      29.8         8.5            NW
    18-Dec-2015 16:00:00    41.2        30         9.2            NNW
    18-Dec-2015 18:00:00    39.9      29.9         4.3            N
```

Specify a time range between `12/18/2015 08:00:00` and `12/18/2015 12:00:00`.

```
S = timerange('12/18/2015 08:00:00','12/18/2015 12:00:00')
```

```
S = 
    timetable timerange subscript:

        Select timetable rows with times in the half-open interval:
        [18-Dec-2015 08:00:00, 18-Dec-2015 12:00:00)
```

```
    See Select Timetable Data by Row Time and Variable Type.
```

Select rows with times in the range specified by S. The output timetable includes the start of the time range, but not the end.

```
TT2 = TT(S,:)
```

```
TT2=2×5 timetable
            Time            Temp    Pressure    WindSpeed    WindDirection
    ____________________    ____    ________    _________    _____________

    18-Dec-2015 08:00:00    37.3      30.1        13.4            NW
    18-Dec-2015 10:00:00    39.1     30.03         6.5            N
```

Select Timetable Rows in Closed Time Interval

Create a timetable.

```
Time = [seconds(1):seconds(1):seconds(5)];
TT = timetable(Time',[98;97.5;97.9;98.1;97.9],[120;111;119;117;116],...
               'VariableNames',{'Reading1','Reading2'})
```

```
TT=5×3 timetable
    Time     Reading1    Reading2
    _____    ________    ________

    1 sec        98        120
    2 sec      97.5        111
    3 sec      97.9        119
    4 sec      98.1        117
    5 sec      97.9        116
```

Specify a closed time interval between two and four seconds.

```
S = timerange(seconds(2),seconds(4),'closed')
```

```
S =
    timetable timerange subscript:

        Select timetable rows with times in the closed interval:
```

```
        [2 sec, 4 sec]

    See Select Timetable Data by Row Time and Variable Type.
```

Select rows with times in the range specified by S. The closed interval includes both the start and end times.

```
TT2 = TT(S,:)
```

```
TT2=3×3 timetable
    Time     Reading1    Reading2
    _____    ________    ________

    2 sec      97.5        111
    3 sec      97.9        119
    4 sec      98.1        117
```

Specify Time Period

Create a timetable containing prices set at the middle of each month.

```
Time = datetime(2018,1:12,15)';
Price = randi([85 110],12,1);
TT = timetable(Time,Price)
```

```
TT=12×2 timetable
       Time        Price
    ___________    _____

    15-Jan-2018     106
    15-Feb-2018     108
    15-Mar-2018      88
    15-Apr-2018     108
    15-May-2018     101
    15-Jun-2018      87
    15-Jul-2018      92
    15-Aug-2018      99
    15-Sep-2018     109
    15-Oct-2018     110
    15-Nov-2018      89
```

```
    15-Dec-2018     110
```

Specify a time range using `'quarters'` as the unit of time. The start of the time range is the quarter that includes January 1, 2018. The end of the range is the quarter that includes May 1, 2018. The time range includes whole quarters, meaning that the end of the range is the instant before the start of July 1, 2018.

```
S = timerange('2018-01-01','2018-05-01','quarters')
```

```
S = 
    timetable timerange subscript:

        Select timetable rows with times in: QUARTERS
          Starting at, including:   01-Jan-2018 00:00:00
          Ending at, but excluding: 01-Jul-2018 00:00:00

    See Select Timetable Data by Row Time and Variable Type.
```

Select rows of TT. The output timetable includes the rows for May 15 and June 15, 2018, but not the row for July 15, or any row with a time outside the first two quarters of 2018.

```
TT(S,:)
```

```
ans=6×2 timetable
       Time        Price
    ___________    _____

    15-Jan-2018     106
    15-Feb-2018     108
    15-Mar-2018      88
    15-Apr-2018     108
    15-May-2018     101
    15-Jun-2018      87
```

Specify Time Period from One Date

Create a timetable containing prices set at the beginning and middle of each month.

```
Time = datetime({'2018-01-01';'2018-01-15';'2018-02-01';'2018-02-15';
                 '2018-03-01';'2018-03-15'});
```

```
Price = randi([85 110],6,1);
TT = timetable(Time,Price)
```

```
TT=6×2 timetable
       Time        Price
    ___________    _____

    01-Jan-2018     106
    15-Jan-2018     108
    01-Feb-2018      88
    15-Feb-2018     108
    01-Mar-2018     101
    15-Mar-2018      87
```

Specify a time range using `'months'` as the unit of time. As the first input is a date in February, 2018, the time range spans the whole month of February.

```
S = timerange('2018-02-01','months')
```

```
S = 
    timetable timerange subscript:

        Select timetable rows with times in: MONTHS
          Starting at, including:   01-Feb-2018 00:00:00
          Ending at, but excluding: 01-Mar-2018 00:00:00

    See Select Timetable Data by Row Time and Variable Type.
```

Select rows of `TT`.

```
TT(S,:)
```

```
ans=2×2 timetable
       Time        Price
    ___________    _____

    01-Feb-2018      88
    15-Feb-2018     108
```

Input Arguments

`startTime,endTime` — Start and end times of time range

pair of datetime scalars | pair of duration scalars | pair of character vectors | pair of string scalars

Start and end times of time range, specified as a pair of datetime or duration scalars, or as a pair of character vectors or string scalars.

If `startTime` and `endTime` are character vectors or string scalars, then they specify dates and times. If `startTime` and `endTime` have formats that `timerange` does not recognize, then convert them to datetime or duration values using the `datetime` or `duration` function. Specify the format using the `'InputFormat'` argument of `datetime` or `duration`.

To create one-sided time ranges, use `'-inf'` or `'inf'` as start or end times. The syntax `timerange('-inf',endTime)` specifies all dates and times before `endTime`, while `timerange(startTime,'inf')` specifies all dates and times after `startTime`.

`intervalType` — Type of time range interval

character vector | string scalar

Type of time range interval, specified as a character vector or string scalar. The table shows the types of time range intervals.

Interval Type	Description
`'open'`	Select rows with times that satisfy the open interval `startTime < rowTime` and `rowTime < endTime`.
`'closed'`	Select rows with times that satisfy the closed interval `startTime <= rowTime` and `rowTime <= endTime`.
`'openleft'`	Select rows with times that satisfy the half-open interval `startTime < rowTime` and `rowTime <= endTime`.
`'openright'` (default)	Select rows with times that satisfy the half-open interval `startTime <= rowTime` and `rowTime < endTime`.

Interval Type	Description
`'closedright'`	Equivalent to `'openleft'`.
`'closedleft'`	Equivalent to `'openright'`.

`startPeriod,endPeriod` — Start and end time periods
pair of datetime scalars | pair of character vectors | pair of string scalars

Start and end time periods, specified as a pair of datetime scalars, or as a pair of character vectors or string scalars.

If `startPeriod` and `endPeriod` are character vectors or string scalars, then they specify dates and times. If `startPeriod` and `endPeriod` have formats that `timerange` does not recognize, then convert them to datetime values using the `datetime` function. Specify the format using the `'InputFormat'` argument of `datetime`.

To create one-sided time ranges, use `'-inf'` or `'inf'` as start or end periods. For example, the syntax `timerange('-inf',endPeriod,'days')` specifies all dates and times before the end of the day of `endPeriod`. The syntax `timerange(startTime,'inf','days')` specifies all dates and times after the start of the day of `startPeriod`.

`timePeriod` — Time period
datetime scalar | character vector | string scalar

Time period, specified as a datetime scalar, character vector, or string scalar. If `timePeriod` is a character vector or a string scalar, then it specifies a date and time that the `datetime` function can convert into a datetime value.

`unitOfTime` — Component of time periods
character vector | string scalar

Component of time periods, specified as a character vector or string scalar. The table shows the components that you can specify.

Note: You can specify `unitOfTime` only when the other input arguments specify datetime values, and not duration values.

Date or Time Component	Description
`'years'`	Select rows with times such that `year(startPeriod) <= year(rowTime)` and `year(rowTime) <= year(endPeriod)`.
`'quarters'`	Select rows with times such that `quarter(startPeriod) <= quarter(rowTime)` and `quarter(rowTime) <= quarter(endPeriod)`.
`'months'`	Select rows with times such that `month(startPeriod) <= month(rowTime)` and `month(rowTime) <= month(endPeriod)`.
`'weeks'`	Select rows with times such that `week(startPeriod) <= week(rowTime)` and `week(rowTime) <= week(endPeriod)`.
`'days'`	Select rows with times such that `day(startPeriod) <= day(rowTime)` and `day(rowTime) <= day(endPeriod)`.
`'hours'`	Select rows with times such that `hour(startPeriod) <= hour(rowTime)` and `hour(rowTime) <= hour(endPeriod)`.
`'minutes'`	Select rows with times such that `minute(startPeriod) <= minute(rowTime)` and `minute(rowTime) <= minute(endPeriod)`.
`'seconds'`	Select rows with times such that `second(startPeriod) <= second(rowTime)` and `second(rowTime) <= second(endPeriod)`.

See Also

`retime` | `synchronize` | `vartype` | `withtol`

Topics

"Create Timetables"
"Select Timetable Data by Row Time and Variable Type"
"Tables"
"Represent Dates and Times in MATLAB"

Introduced in R2016b

timetable

Timetable array with time-stamped rows and variables of different types

Description

`timetable` is a type of table that associates a time with each row. Like tables, timetables store column-oriented data variables that can have different data types and sizes as long as they have the same number of rows. In addition, timetables provide time-specific functions to align, combine, and perform calculations with time-stamped data in one or more timetables.

The *row times* of a timetable are `datetime` or `duration` values that label the rows. You can index into a timetable by row time and variable. To index into a timetable, use smooth parentheses `()` to return a subtable or curly braces `{}` to extract the contents. You can reference variables and the vector of row times using names. For more information on indexing, see “Select Timetable Data by Row Time and Variable Type” and “Access Data in Tables”.

Creation

You can read data from a file into a table using the `readtimetable` function, or convert variables to timetables using the `array2timetable` or `table2timetable` functions. Alternatively, you can use the `timetable` function described below.

When you use the `timetable` function, you can specify a vector of row times, or create row times using a sample rate or time step. Also, you can create a timetable with preallocated space for variables whose values are filled in later.

Syntax

```
TT = timetable(rowTimes,var1,...,varN)
TT = timetable(var1,...,varN,'RowTimes',rowTimes)
TT = timetable(var1,...,varN,'SampleRate',Fs)
TT = timetable(var1,...,varN,'TimeStep',dt)
```

```
TT = timetable('Size',sz,'VariableTypes',varTypes,'RowTimes',
rowTimes)
TT = timetable('Size',sz,'VariableTypes',varTypes,'SampleRate',Fs)
TT = timetable('Size',sz,'VariableTypes',varTypes,'TimeStep',dt)
```

```
TT = timetable( ___ ,Name,Value)
```

Description

`TT = timetable(rowTimes,var1,...,varN)` creates a timetable from the input data variables `var1,...,varN` and the time vector `rowTimes`. The data variables can have different sizes and data types as long as they have the same number of rows. `rowTimes` must be a datetime or duration vector, also with the same number of rows.

`TT = timetable(var1,...,varN,'RowTimes',rowTimes)` specifies `rowTimes` as the source of the row times of `TT`. When you use this syntax, the name of the row times vector of `TT` is always `Time`, even when `rowTimes` is a workspace variable with a different name.

`TT = timetable(var1,...,varN,'SampleRate',Fs)` uses the sample rate `Fs` to calculate regularly spaced row times. `Fs` is a positive numeric scalar that specifies the number of samples per second (Hz). The first row time is zero seconds.

`TT = timetable(var1,...,varN,'TimeStep',dt)` uses the time step `dt` to calculate regularly spaced row times. `dt` is a duration or calendar duration value that specifies the length of time between consecutive row times. The first row time is zero seconds.

`TT = timetable('Size',sz,'VariableTypes',varTypes,'RowTimes', rowTimes)` creates a timetable and preallocates space for the variables that have data types you specify. `sz` is a two-element numeric array, where `sz[1]` specifies the number of rows and `sz[2]` specifies the number of variables. `varTypes` is a cell array of character vectors specifying data types. The number of times in `rowTimes` must equal `sz[1]`.

`TT = timetable('Size',sz,'VariableTypes',varTypes,'SampleRate',Fs)` preallocates variables with data types and adds row times using the sample rate `Fs`. The first row time is zero seconds.

`TT = timetable('Size',sz,'VariableTypes',varTypes,'TimeStep',dt)` preallocates variables with data types and adds row times using the time step `dt`. The first row time is zero seconds.

`TT = timetable( ___ ,Name,Value)` specifies additional input arguments using one or more name-value pair arguments. For example, you can specify variable names using the `'VariableNames'` name-value pair. You also can specify a start time using the `'StartTime'` name-value pair with a sample rate or time step. You can use this syntax with any of the input arguments of the previous syntaxes.

Input Arguments

`var1,...,varN` — Input variables

arrays

Input variables, specified as arrays with the same number of rows. The input variables can have different sizes and different data types, as long as they have the same number of rows.

Common input variables are numeric arrays, logical arrays, string arrays, structure arrays, and cell arrays.

Input variables also can be objects that are arrays. Such an array must support indexing of the form `var(index1,...,indexN)`, where `index1` is a numeric or logical vector that corresponds to rows of the variable `var`. In addition, the array must implement both a `vertcat` method and a `size` method with a `dim` argument.

`rowTimes` — Times associated with rows of timetable

datetime vector | duration vector

Times associated with rows of a timetable, specified as a datetime vector or a duration vector. Each time labels a row in the output timetable, `TT`. The time values in `rowTimes` do not need to be unique, sorted, or regular.

Data Types: `datetime` | `duration`

`sz` — Size of preallocated timetable

two-element numeric vector

Size of the preallocated timetable, specified as a two-element numeric vector. The first element of `sz` specifies the number of rows, and the second element specifies the number of timetable variables.

varTypes — Data types of preallocated variables
cell array of character vectors | string array

Data types of the preallocated variables, specified as a cell array of character vectors or a string array. The number of types specified by `varTypes` must equal the number of variables specified by the second element of `sz`.

`varTypes` can contain the names of any data types, including the names shown in the table.

Data Type Name	Initial Value in Each Element
`'double'`, `'single'`	Double- or single-precision `0`
`'doublenan'`, `'doubleNaN'`, `'singlenan'`, `'singleNaN'`	Double- or single-precision `NaN`
`'int8'`, `'int16'`, `'int32'`, `'int64'`	Signed 8-, 16-, 32-, or 64-bit integer `0`
`'uint8'`, `'uint16'`, `'uint32'`, `'uint64'`	Unsigned 8-, 16-, 32-, or 64-bit integer `0`
`'logical'`	`0` (`false`)
`'categorical'`	`<undefined>` categorical value
`'datetime'`	`NaT` datetime value
`'duration'`	`0` seconds, as a duration value
`'calendarDuration'`	`0` days, as a calendar duration value
`'string'`	`""` (1-by-1 string with no characters)
`'cellstr'`	`{''}` (cell with 0-by-0 character array)
`'cell'`	`{[]}` (cell with 0-by-0 double array)
`'struct'`	Scalar structure with no fields
`'table'`	Table with no variables

For any other data type, the initial value is the value used by that type or class to "in-fill" unassigned elements of an array.

If you specify `'char'` as a data type, then `timetable` preallocates the corresponding variable as a cell array of character vectors, not as a character array. Best practice is to avoid creating table or timetable variables that are character arrays. When working with text data in a table or a timetable, consider using a string array or a categorical array.

`Fs` — Sample rate
positive numeric scalar

Sample rate, specified as a positive numeric scalar. `Fs` specifies the number of samples per second (Hz).

`dt` — Time step
duration scalar

Time step, specified as a duration scalar.

If you specify `dt` as a calendar duration, and you specify the `'StartTime'` name-value pair argument, then the value of `'StartTime'` must be a datetime scalar.

Data Types: `duration` | `calendarDuration`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `TT = timetable(rowTimes,T,W,'VariableNames',{'Temperature','WindSpeed'})` creates a timetable from input arrays `T` and `W`, and names the corresponding timetable variables `Temperature` and `WindSpeed`.

`VariableNames` — Variable names
cell array of character vectors | string array

Variable names, specified as the comma-separated pair consisting of `'VariableNames'` and a cell array of character vectors or a string array whose elements are nonempty and distinct. The number of names in the array must equal the number of timetable variables. The `timetable` function also stores the variable names in the `VariableNames` property of the timetable.

Variable names can have any Unicode characters, including spaces and non-ASCII characters.

`StartTime` — Start time
datetime scalar | duration scalar

Start time, specified as the comma-separated pair consisting of `'StartTime'` and a datetime scalar or duration scalar. The start time is also the row time of the first row of the timetable.

You can specify `'StartTime'` only when you also specify the `'SampleRate'` or `'TimeStep'` name-value pair argument.

- If the start time is a datetime value, then the row times of `TT` are datetime values.
- If the start time is a duration, then the row times are durations.
- If the time step `dt` is a calendar duration value, then the start time must be a datetime value.

Data Types: `datetime` | `duration`

Properties

Access Timetable Metadata Properties

A timetable contains metadata properties that describe the timetable, its row times, and its variables. Access these properties using the syntax *`timetableName`*`.Properties.`*`PropertyName`*, where *`PropertyName`* is the name of a property. For example, you can access the names of the variables in the timetable `TT` through the `TT.Properties.VariableNames` property.

You can return a summary of the metadata properties using the syntax *`timetableName`*`.Properties`.

Timetables provides metadata access through the `Properties` property because you can access timetable data directly using dot syntax. For example, if timetable `TT` has a variable named `Var1`, then you can access the variable using the syntax `TT.Var1`, accessing the values in an array.

Timetable Metadata

`DimensionNames` — Dimension names
`{'Time','Variables'}` (default) | two-element cell array of character vectors | two-element string array

Dimension names, specified as a two-element cell array of character vectors or two-element string array.

You can access timetable data using the two dimension names. If you use dot syntax and the first dimension name, then you can access the row times as a vector. If you use dot syntax and the second dimension name, then the data from all the variables are concatenated together in one array, as though you had indexed into the timetable using `{:,:}` syntax.

If you specify this property using a string array, then it is converted and stored as a cell array of character vectors.

Dimension names can have any Unicode characters, including spaces and non-ASCII characters.

Example

Create a timetable and display its dimension names. You can access row times and data using dimension names with dot syntax.

```
TT = timetable(datetime({'2015-12-18';'2015-12-19';'2015-12-20'}), ...
               [37.3;39.1;42.3],[30.1;30.03;29.9],[13.4;6.5;7.3]);
TT.Properties.DimensionNames
```

```
ans = 1x2 cell array
    {'Time'}    {'Variables'}
```

Access the row times using the first dimension name.

```
TT.Time
```

```
ans = 3x1 datetime array
   18-Dec-2015
   19-Dec-2015
   20-Dec-2015
```

Access the data using the second dimension name. This syntax is equivalent to `TT{:,:,}`.

```
TT.Variables
```

```
ans = 3×3

   37.3000   30.1000   13.4000
   39.1000   30.0300    6.5000
   42.3000   29.9000    7.3000
```

Modify the names of its dimensions using the `Properties.DimensionNames` property. Having changed the dimension names, you can access the row times and data using the syntaxes `TT.Date` and `TT.WeatherData` respectively.

```
TT.Properties.DimensionNames = {'Date','WeatherData'};
TT.Properties
```

```
ans = 
  TimetableProperties with properties:

             Description: ''
                UserData: []
          DimensionNames: {'Date'   'WeatherData'}
           VariableNames: {'Var1'   'Var2'   'Var3'}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                 RowTimes: [3x1 datetime]
               StartTime: 18-Dec-2015
              SampleRate: NaN
                TimeStep: 1d
        CustomProperties: No custom properties are set.
      Use addprop and rmprop to modify CustomProperties.
```

`Description` — Timetable description

`''` (default) | character vector | string scalar

Timetable description, specified as a character vector or string scalar. This description is visible when using the `summary` function.

If you specify this property using a string scalar, then it is converted and stored as a character vector.

Example

Create a timetable. Modify the variable names and the description of the timetable. Display a summary of the result.

```
TT = timetable(datetime({'2015-12-18';'2015-12-19';'2015-12-20'}), ...
               [37.3;39.1;42.3],[30.1;30.03;29.9],[13.4;6.5;7.3]);
TT.Properties.VariableNames = {'Temp','Pressure','WindSpeed'};
TT.Properties.Description = 'Weather Data from December 2015';
summary(TT)
```

```
Description:  Weather Data from December 2015

RowTimes:

    Time: 3x1 datetime
        Values:
            Min           18-Dec-2015
            Median        19-Dec-2015
            Max           20-Dec-2015
            TimeStep      24:00:00

Variables:

    Temp: 3x1 double

        Values:

            Min        37.3
            Median     39.1
            Max        42.3

    Pressure: 3x1 double

        Values:

            Min          29.9
            Median      30.03
            Max          30.1

    WindSpeed: 3x1 double

        Values:

            Min           6.5
            Median        7.3
            Max          13.4
```

`UserData` — Additional timetable information
`[ ]` (default) | array

Additional timetable information, specified as an array. You can attach data of any kind to a timetable using this property.

Example

Create a timetable. Modify the variable names. Attach an anonymous function as a piece of user data that is associated with the timetable.

```
TT = timetable(datetime({'2015-12-18';'2015-12-19';'2015-12-20'}), ...
               [37.3;39.1;42.3],[30.1;30.03;29.9],[13.4;6.5;7.3]);
TT.Properties.VariableNames = {'Temp','Pressure','WindSpeed'};
Fahrenheit2Celsius = @(x) (5.0/9.0).*(x - 32);
TT.Properties.UserData = Fahrenheit2Celsius;
TT.Properties
```

```
ans = 
  TimetableProperties with properties:

             Description: ''
                UserData: @(x)(5.0/9.0).*(x-32)
          DimensionNames: {'Time'   'Variables'}
           VariableNames: {'Temp'   'Pressure'   'WindSpeed'}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                 RowTimes: [3x1 datetime]
               StartTime: 18-Dec-2015
              SampleRate: NaN
                TimeStep: 1d
        CustomProperties: No custom properties are set.
      Use addprop and rmprop to modify CustomProperties.
```

Row Times Metadata

`RowTimes` — Row times
datetime vector | duration vector

Row times, specified as a datetime vector or duration vector. There must be a row time for every row of the timetable. However, a timetable can have row times that are duplicates, out of order, or `NaT` or `NaN` values.

The row times are visible when you view the timetable. Furthermore, you can use the row times within parentheses or curly braces to access the timetable data.

Another way to access the row times is to use dot syntax and the name of the first dimension of the table.

Example

Create a timetable. Then replace its row times using the `Properties.RowTimes` property.

```
TT = timetable(datetime({'2015-12-18';'2015-12-19';'2015-12-20'}), ...
               [37.3;39.1;42.3],[30.1;30.03;29.9],[13.4;6.5;7.3])
```

```
TT=3×4 timetable
       Time        Var1    Var2     Var3
    ___________    ____    _____    ____

    18-Dec-2015    37.3     30.1    13.4
    19-Dec-2015    39.1    30.03     6.5
    20-Dec-2015    42.3     29.9     7.3
```

```
Dates = datetime(2017,1,1:3);
TT.Properties.RowTimes = Dates
```

```
TT=3×4 timetable
       Time        Var1    Var2     Var3
    ___________    ____    _____    ____

    01-Jan-2017    37.3     30.1    13.4
    02-Jan-2017    39.1    30.03     6.5
    03-Jan-2017    42.3     29.9     7.3
```

Another way to access the row times is by using dot syntax with the name of the first dimension of the timetable.

```
TT.Properties.DimensionNames
```

```
ans = 1x2 cell array
    {'Time'}    {'Variables'}
```

```
TT.Time
```

```
ans = 3x1 datetime array
   01-Jan-2017
   02-Jan-2017
   03-Jan-2017
```

StartTime — Start time of row times

datetime scalar | duration scalar

Start time of the row times, specified as a datetime scalar or duration scalar. The start time is equal to the row time for the first row of the timetable, and has the same data type.

- If the start time is a datetime, then the row times of TT are datetime values.
- If the start time is a duration, then the row times are durations.
- If the time step is a calendar duration, then the start time must be a datetime value.

If the timetable is empty, then the start time is NaN.

Example

Create a timetable. In this timetable, the time step between consecutive rows is not the same, so the timetable is irregular.

```
TT = timetable(datetime({'2015-12-18';'2015-12-20';'2015-12-21'}), ...
               [37.3;39.1;42.3],[13.4;6.5;7.3],{'N';'SE';'NW'});
TT.Properties.VariableNames = {'Temperature','WindSpeed','WindDirection'}
```

```
TT=3×4 timetable
       Time        Temperature    WindSpeed    WindDirection
    ___________    ___________    _________    _____________

    18-Dec-2015       37.3          13.4          {'N' }
    20-Dec-2015       39.1           6.5          {'SE'}
    21-Dec-2015       42.3           7.3          {'NW'}
```

Display its properties. The value of the StartTime property is equal to the first row time. Whether a timetable is regular or irregular, it always has a start time, unless it is empty.

```
TT.Properties
```

```
ans = 
  TimetableProperties with properties:

             Description: ''
                UserData: []
          DimensionNames: {'Time'  'Variables'}
           VariableNames: {'Temperature'  'WindSpeed'  'WindDirection'}
    VariableDescriptions: {}
```

```
          VariableUnits: {}
     VariableContinuity: []
               RowTimes: [3x1 datetime]
              StartTime: 18-Dec-2015
             SampleRate: NaN
               TimeStep: NaN
       CustomProperties: No custom properties are set.
     Use addprop and rmprop to modify CustomProperties.
```

Modify the `StartTime` property. Note that all the row times have new values. For each row time, the change in value is equal to the difference between the original value of the first row time and the new start time.

```
TT.Properties.StartTime = datetime('2018-04-09')
```

```
TT=3×4 timetable
            Time            Temperature    WindSpeed    WindDirection
    ____________________    ___________    _________    _____________

    09-Apr-2018 00:00:00       37.3          13.4          {'N' }
    11-Apr-2018 00:00:00       39.1           6.5          {'SE'}
    12-Apr-2018 00:00:00       42.3           7.3          {'NW'}
```

`SampleRate` — Sample rate

positive numeric scalar

Sample rate, specified as a positive numeric scalar. The sample rate is the number of samples per second (Hz).

If the row times are not regular, or the timetable is empty, then the sample rate is `NaN`.

Example

`TimeStep` — Time step

duration scalar

Time step, specified as a duration scalar.

- If you specify the time step as a calendar duration (for example, calendar months), then the vector of row times must be a datetime vector.
- If you specify the time step as a duration (for example, seconds), then the vector of row times either can be a datetime or duration vector.

If the row times are not regular, or the timetable is empty, then the time step is `NaN`.

Example

Create a regular timetable. In this timetable, the row times are durations, created using the same time step.

```
Intensity = [100;98.7;95.2;101.4;99.1];
TT = timetable(Intensity,'TimeStep',seconds(0.01))
```

```
TT=5×2 timetable
      Time      Intensity
    ________    _________

    0 sec           100
    0.01 sec       98.7
    0.02 sec       95.2
    0.03 sec      101.4
    0.04 sec       99.1
```

Display its properties. The `TimeStep` property stores the time step as a duration.

```
TT.Properties
```

```
ans = 
  TimetableProperties with properties:

             Description: ''
                UserData: []
          DimensionNames: {'Time'  'Variables'}
           VariableNames: {'Intensity'}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                 RowTimes: [5x1 duration]
               StartTime: 0 sec
              SampleRate: 100
                TimeStep: 0.01 sec
        CustomProperties: No custom properties are set.
      Use addprop and rmprop to modify CustomProperties.
```

Modify the `TimeStep` property. The start time is the same, but all the other row times are different because the time step is larger. However, the variable `Intensity` remains the same.

```
TT.Properties.TimeStep = seconds(0.04)
```

```
TT=5×2 timetable
      Time      Intensity
    ________    _________

    0 sec           100
    0.04 sec       98.7
    0.08 sec       95.2
    0.12 sec      101.4
    0.16 sec       99.1
```

Data Types: `duration` | `calendarDuration`

Variable Metadata

`VariableNames` — Variable names

cell array of character vectors | string array

Variable names, specified as a cell array of character vectors or a string array, whose elements are nonempty and distinct. The number of names must equal the number of variables.

Variable names can have any Unicode characters, including spaces and non-ASCII characters.

The variable names are visible when viewing the timetable and when using the `summary` function. Furthermore, you can use the variable names within parentheses, within curly braces, or with dot indexing to access timetable data.

If you specify this property using a string array, then it is converted and stored as a cell array of character vectors.

Example

Create a timetable with default variable names. Then modify the names using the `Properties.VariableNames` property.

```
TT = timetable(datetime({'2015-12-18';'2015-12-19';'2015-12-20'}), ...
               [37.3;39.1;42.3],[30.1;30.03;29.9],[13.4;6.5;7.3])
```

```
TT=3×4 timetable
       Time        Var1    Var2     Var3
```

```
    ___________    ____    _____    ____

    18-Dec-2015    37.3     30.1    13.4
    19-Dec-2015    39.1    30.03     6.5
    20-Dec-2015    42.3     29.9     7.3
```

```
TT.Properties.VariableNames = {'Temp','Pressure','WindSpeed'}
```

```
TT=3×4 timetable
       Time        Temp    Pressure    WindSpeed
    ___________    ____    ________    _________

    18-Dec-2015    37.3      30.1        13.4
    19-Dec-2015    39.1     30.03         6.5
    20-Dec-2015    42.3      29.9         7.3
```

A fundamental way to display and modify variables is to access them by name using dot syntax.

```
TT.Temp
```

```
ans = 3×1

   37.3000
   39.1000
   42.3000
```

```
TT.Pressure(3) = 30
```

```
TT=3×4 timetable
       Time        Temp    Pressure    WindSpeed
    ___________    ____    ________    _________

    18-Dec-2015    37.3      30.1        13.4
    19-Dec-2015    39.1     30.03         6.5
    20-Dec-2015    42.3        30         7.3
```

`VariableDescriptions` — Variable descriptions

`{}` (default) | cell array of character vectors | string array

Variable descriptions, specified as a cell array of character vectors or a string array. This property can be an empty cell array, which is the default. If the array is not empty, then it

must contain as many elements as there are variables. You can specify an individual empty character vector or empty string for a variable that does not have a description.

The variable descriptions are visible when using the `summary` function.

If you specify this property using a string array, then it is converted and stored as a cell array of character vectors.

Example

Create a timetable. Modify the variable names and descriptions. Display a summary of the result.

```
TT = timetable(datetime({'2015-12-18';'2015-12-19';'2015-12-20'}), ...
               [37.3;39.1;42.3],[30.1;30.03;29.9],[13.4;6.5;7.3]);
TT.Properties.VariableNames = {'Temp','Pressure','WindSpeed'};
TT.Properties.VariableDescriptions = {'Temperature (external)', ...
                                      'Pressure in Hg', ...
                                      'Wind speed at sensor 123'};
summary(TT)
```

```
RowTimes:

    Time: 3x1 datetime
        Values:
            Min           18-Dec-2015
            Median        19-Dec-2015
            Max           20-Dec-2015
            TimeStep      24:00:00

Variables:

    Temp: 3x1 double

        Properties:
            Description:  Temperature (external)
        Values:

            Min        37.3
            Median     39.1
            Max        42.3

    Pressure: 3x1 double

        Properties:
```

```
            Description:  Pressure in Hg
        Values:

            Min          29.9
            Median      30.03
            Max          30.1

    WindSpeed: 3x1 double

        Properties:
            Description:  Wind speed at sensor 123
        Values:

            Min           6.5
            Median        7.3
            Max          13.4
```

`VariableUnits` — Variable units

`{}` (default) | cell array of character vectors | string array

Variable units, specified as a cell array of character vectors or a string array. This property can be an empty cell array, which is the default. If the array is not empty, then it must contain as many elements as there are variables. You can specify an individual empty character vector or empty string for a variable that does not have units.

The variable units are visible when using the `summary` function.

If you specify this property using a string array, then it is converted and stored as a cell array of character vectors.

Example

Create a timetable. Modify the variable names and units. Display a summary of the result.

```
TT = timetable(datetime({'2015-12-18';'2015-12-19';'2015-12-20'}), ...
               [37.3;39.1;42.3],[30.1;30.03;29.9],[13.4;6.5;7.3]);
TT.Properties.VariableNames = {'Temp','Pressure','WindSpeed'};
TT.Properties.VariableUnits = {'degrees F','mm Hg','mph'};
summary(TT)
```

```
RowTimes:

    Time: 3x1 datetime
        Values:
            Min           18-Dec-2015
```

```
            Median        19-Dec-2015
            Max           20-Dec-2015
            TimeStep      24:00:00

Variables:

    Temp: 3x1 double

        Properties:
            Units:  degrees F
        Values:

            Min        37.3
            Median     39.1
            Max        42.3

    Pressure: 3x1 double

        Properties:
            Units:  mm Hg
        Values:

            Min          29.9
            Median      30.03
            Max          30.1

    WindSpeed: 3x1 double

        Properties:
            Units:  mph
        Values:

            Min           6.5
            Median        7.3
            Max          13.4
```

`VariableContinuity` — Status as continuous or discrete variables

`[]` (default) | cell array of character vectors | string array

Status as continuous or discrete variables, specified as a cell array of character vectors or a string array. This property can be an empty array, which is the default. If the array is not empty, then it must contain as many elements as there are variables. Each element can be either `'unset'`, `'continuous'`, `'step'`, or `'event'`.

The values in `VariableContinuity` affect how the `retime` or `synchronize` functions work. If you specify `VariableContinuity` and call `retime` or `synchronize`, then you do not need to specify a method. Instead, `retime` and `synchronize` fill in the output timetable variables using the following default methods:

- `'unset'` — Fill in values using missing data indicator for that type (such as `NaN` for numeric variables).
- `'continuous'` — Fill in values using linear interpolation.
- `'step'` — Fill in values using previous value.
- `'event'` — Fill in values using missing data indicator for that type (such as `NaN` for numeric variables).

If you do specify a method as an input argument to `retime` or `synchronize`, then that method overrides the values you specify in `VariableContinuity`.

For more information on using the `VariableContinuity` property, see “Retime and Synchronize Timetable Variables Using Different Methods”.

Example

Create a timetable. Specify a `matlab.tabular.Continuity` value for each variable.

```
TT = timetable(datetime({'2015-12-18';'2015-12-19';'2015-12-20'}), ...
               [37.3;39.1;42.3],[13.4;6.5;7.3],{'N';'SE';'NW'});
TT.Properties.VariableNames = {'Temperature','WindSpeed','WindDirection'};
TT.Properties.VariableContinuity = {'continuous','event','event'};
TT.Properties
```

```
ans = 
  TimetableProperties with properties:

             Description: ''
                UserData: []
           DimensionNames: {'Time'   'Variables'}
            VariableNames: {'Temperature'   'WindSpeed'   'WindDirection'}
    VariableDescriptions: {}
            VariableUnits: {}
      VariableContinuity: [continuous    event    event]
                 RowTimes: [3x1 datetime]
               StartTime: 18-Dec-2015
              SampleRate: NaN
                TimeStep: 1d
        CustomProperties: No custom properties are set.
```

```
Use addprop and rmprop to modify CustomProperties.
```

Custom Metadata

`CustomProperties` — Customized metadata of timetable and its variables
`CustomProperties` object

Customized metadata of a timetable and its variables, specified as a `CustomProperties` object.

The `CustomProperties` object is a container for customized metadata that you can add to a timetable. By default, `CustomProperties` has zero properties. Each property you add to `CustomProperties` can contain either table metadata or variable metadata. If a property contains variable metadata, then its value must be an array, and the number of elements in the array must equal the number of timetable variables.

- To add properties for customized metadata to a timetable, use the `addprop` function.
- To access or modify customized metadata, use the syntax *`timetableName`*`.Properties.CustomProperties.`*`PropertyName`*. In this syntax, *`PropertyName`* is the name you chose when you added that property using `addprop`.
- To remove properties, use the `rmprop` function.

Note: You can add or remove **only** properties for customized metadata using `addprop` and `rmprop`. You cannot add or remove properties of the *`timetableName`*`.Properties` object.

Example

Create a timetable containing weather data.

```
TT = timetable(datetime({'2015-12-18';'2015-12-20';'2015-12-21'}), ...
               [37.3;39.1;42.3],[13.4;6.5;7.3],{'N';'SE';'NW'});
TT.Properties.VariableNames = {'Temperature','WindSpeed','WindDirection'}
```

```
TT=3×4 timetable
       Time        Temperature    WindSpeed    WindDirection
    ___________    ___________    _________    _____________

    18-Dec-2015       37.3          13.4          {'N' }
    20-Dec-2015       39.1           6.5          {'SE'}
    21-Dec-2015       42.3           7.3          {'NW'}
```

To describe the instruments that measured these data, and the name of an output file, add customized metadata using the `addprop` function. The `Instruments` property has variable metadata that apply to the variables of `TT`. The `OutputFile` property has table metadata.

```
TT = addprop(TT,{'Instruments','OutputFile'},{'variable','table'});
TT.Properties
```

```
ans = 
  TimetableProperties with properties:

             Description: ''
                UserData: []
          DimensionNames: {'Time'   'Variables'}
           VariableNames: {'Temperature'   'WindSpeed'   'WindDirection'}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                RowTimes: [3x1 datetime]
               StartTime: 18-Dec-2015
              SampleRate: NaN
                TimeStep: NaN

   Custom Properties (access using t.Properties.CustomProperties.<name>):
             OutputFile: []
            Instruments: []
```

Assign values to the customized metadata using dot syntax. When you assign an array of text values to customized metadata, the best practice is to use a string array, not a cell array of character vectors. If a property of `CustomProperties` is a cell array of character vectors, then there is no mechanism to prevent you from later assigning nontext values as elements of the cell array.

```
TT.Properties.CustomProperties.Instruments = ["thermometer","anemometer","wind vane"];
TT.Properties.CustomProperties.OutputFile = 'weatherReadings.csv';
TT.Properties
```

```
ans = 
  TimetableProperties with properties:

             Description: ''
                UserData: []
          DimensionNames: {'Time'   'Variables'}
```

```
           VariableNames: {'Temperature'  'WindSpeed'  'WindDirection'}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                 RowTimes: [3x1 datetime]
                StartTime: 18-Dec-2015
               SampleRate: NaN
                 TimeStep: NaN

   Custom Properties (access using t.Properties.CustomProperties.<name>):
             OutputFile: 'weatherReadings.csv'
            Instruments: ["thermometer"    "anemometer"    "wind vane"]
```

Remove the `OutputFile` property from `TT`.

```
TT = rmprop(TT,'OutputFile');
TT.Properties
```

```
ans =
  TimetableProperties with properties:

            Description: ''
               UserData: []
          DimensionNames: {'Time'  'Variables'}
           VariableNames: {'Temperature'  'WindSpeed'  'WindDirection'}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                 RowTimes: [3x1 datetime]
                StartTime: 18-Dec-2015
               SampleRate: NaN
                 TimeStep: NaN

   Custom Properties (access using t.Properties.CustomProperties.<name>):
            Instruments: ["thermometer"    "anemometer"    "wind vane"]
```

Examples

Store and Synchronize Related Data Variables in Timetable

Store data about weather conditions measured at different times in a timetable. Aside from storage, timetables provide functions to synchronize data to times that you specify. Also, you can annotate the timetable to describe your work and the variables of the timetable.

Create a timetable from workspace variables. The values in `MeasurementTime` become the row times of the timetable. All the other input arguments become the timetable variables. When you use this syntax, the names of the row times vector and the variables of `TT` are the names of the corresponding input arguments.

```
MeasurementTime = datetime({'2015-12-18 08:03:05';'2015-12-18 10:03:17';'2015-12-18 12
Temp = [37.3;39.1;42.3];
Pressure = [30.1;30.03;29.9];
WindSpeed = [13.4;6.5;7.3];
TT = timetable(MeasurementTime,Temp,Pressure,WindSpeed)
```

```
TT=3×4 timetable
      MeasurementTime       Temp    Pressure    WindSpeed
    ____________________    ____    ________    _________

    18-Dec-2015 08:03:05    37.3      30.1        13.4
    18-Dec-2015 10:03:17    39.1     30.03         6.5
    18-Dec-2015 12:03:13    42.3      29.9         7.3
```

Synchronize the weather data to regular times with an hourly time step. Adjust the data to the new times using linear interpolation.

```
TT2 = retime(TT,'hourly','linear')
```

```
TT2=6×4 timetable
      MeasurementTime        Temp     Pressure    WindSpeed
    ____________________    ______    ________    _________

    18-Dec-2015 08:00:00    37.254     30.102      13.577
    18-Dec-2015 09:00:00    38.152     30.067      10.133
    18-Dec-2015 10:00:00    39.051     30.032      6.6885
    18-Dec-2015 11:00:00    40.613     29.969      6.8783
    18-Dec-2015 12:00:00    42.214     29.903      7.2785
    18-Dec-2015 13:00:00    43.815     29.838      7.6788
```

Since the row times of the output are not the measured times, rename the vector of row times. This vector is also the name of the first dimension of the timetable.

```
TT2.Properties.DimensionNames{1} = 'InterpolatedTimes'
```

```
TT2=6×4 timetable
     InterpolatedTimes       Temp     Pressure    WindSpeed
    ____________________    ______    ________    _________

    18-Dec-2015 08:00:00    37.254     30.102      13.577
    18-Dec-2015 09:00:00    38.152     30.067      10.133
    18-Dec-2015 10:00:00    39.051     30.032      6.6885
    18-Dec-2015 11:00:00    40.613     29.969      6.8783
    18-Dec-2015 12:00:00    42.214     29.903      7.2785
    18-Dec-2015 13:00:00    43.815     29.838      7.6788
```

Annotate `TT2` with a description. You can annotate `TT2` and its variables using metadata accessed through `TT2.Properties`.

```
TT2.Properties.Description = 'Weather data, interpolated to regular hourly times';
TT2.Properties
```

```
ans =
  TimetableProperties with properties:

             Description: 'Weather data, interpolated to regular hourly times'
                UserData: []
          DimensionNames: {'InterpolatedTimes'  'Variables'}
           VariableNames: {'Temp'  'Pressure'  'WindSpeed'}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                 RowTimes: [6x1 datetime]
               StartTime: 18-Dec-2015 08:00:00
              SampleRate: 2.7778e-04
                TimeStep: 01:00:00
        CustomProperties: No custom properties are set.
      Use addprop and rmprop to modify CustomProperties.
```

Access Row Times and Variable Data

You can use dot syntax to access the row times of a timetable. Also, you can access individual variables using dot syntax, or all the data in a timetable using its second dimension name.

Load a timetable from the MAT-file `outdoors`. Display the first three rows.

```
load outdoors
outdoors(1:3,:)
```

```
ans=3×4 timetable
           Time            Humidity    TemperatureF    PressureHg
    ____________________    ________    ____________    __________

    2015-11-15 00:00:24        49          51.3          29.61
    2015-11-15 01:30:24      48.9          51.5          29.61
    2015-11-15 03:00:24      48.9          51.5          29.61
```

Access the row times using the name of the row times vector. This name is also the name of the first dimension of the timetable. `outdoors` stores the row times as a datetime vector. Display the first three times.

```
outdoors.Time(1:3)
```

```
ans = 3x1 datetime array
   2015-11-15 00:00:24
   2015-11-15 01:30:24
   2015-11-15 03:00:24
```

Access the temperatures as a numeric vector, using its variable name.

```
outdoors.TemperatureF(1:3)
```

```
ans = 3×1

   51.3000
   51.5000
   51.5000
```

Access all the timetable data as a matrix, using the syntax `outdoors.Variables`. This syntax uses the second dimension name of the timetable, and is equivalent to accessing

all the contents using curly brace indexing, `outdoors{:,:}`. However, the matrix does not include row times, because the vector of row times is timetable metadata, not a variable. If the timetable data cannot be concatenated into a matrix, then an error message is raised.

```
outdoors.Variables
```

```
ans = 51×3

   49.0000   51.3000   29.6100
   48.9000   51.5000   29.6100
   48.9000   51.5000   29.6100
   48.8000   51.5000   29.6100
   48.7000   51.5000   29.6000
   48.8000   51.5000   29.6000
   49.0000   51.5000   29.6000
   49.1000   51.3000   29.6000
   49.1000   51.3000   29.6100
   49.1000   51.5000   29.6100
      ⋮
```

Rename the second dimension of `outdoors`. If you change the name, then you can use the new name to access the data.

```
outdoors.Properties.DimensionNames{2} = 'Data';
outdoors.Data
```

```
ans = 51×3

   49.0000   51.3000   29.6100
   48.9000   51.5000   29.6100
   48.9000   51.5000   29.6100
   48.8000   51.5000   29.6100
   48.7000   51.5000   29.6000
   48.8000   51.5000   29.6000
   49.0000   51.5000   29.6000
   49.1000   51.3000   29.6000
   49.1000   51.3000   29.6100
   49.1000   51.5000   29.6100
      ⋮
```

Specify Time Vector

Create a timetable using the `'RowTimes'` name-value pair argument. Note that the name of the row times vector of `TT` is `Time`, not `MeasurementTime`. When you use this syntax, the name of the row times vector is always `Time`.

```
MeasurementTime = datetime({'2015-12-18 08:03:05';'2015-12-18 10:03:17';'2015-12-18 12
Temp = [37.3;39.1;42.3];
Pressure = [29.4;29.6;30.0];
Precip = [0.1;0.9;0.0];
StormDuration = [hours(1);hours(2);NaN];
TT = timetable(Temp,Pressure,Precip,StormDuration,'RowTimes',MeasurementTime)
```

```
TT=3×5 timetable
            Time            Temp    Pressure    Precip    StormDuration
    ____________________    ____    ________    ______    _____________

    18-Dec-2015 08:03:05    37.3      29.4       0.1           1 hr
    18-Dec-2015 10:03:17    39.1      29.6       0.9           2 hr
    18-Dec-2015 12:03:13    42.3        30         0         NaN hr
```

Default Names

Create a timetable. If there are input arguments that are not workspace variables, then the `timetable` function assigns default names to the corresponding row times vector and the variables of the timetable. For example, if you transpose some input arguments to make them column vectors, then those input arguments are not workspace variables. The default names are `Time` for the vector of row times, and `VarN` for the `N`th timetable variable.

```
T = hours(1:3);
Temp = [37.3;39.1;42.3];
P = [29.4 29.6 30];
TT = timetable(T',Temp,P')
```

```
TT=3×3 timetable
    Time    Temp    Var2
    ____    ____    ____

    1 hr    37.3    29.4
    2 hr    39.1    29.6
```

```
    3 hr    42.3      30
```

Specify Sample Rate

Create a regular timetable using a sample rate of 100 Hz.

```
Intensity = [100;98.7;95.2;101.4;99.1];
TT = timetable(Intensity,'SampleRate',100)
```

```
TT=5×2 timetable
      Time      Intensity
    ________    _________

    0 sec           100
    0.01 sec       98.7
    0.02 sec       95.2
    0.03 sec      101.4
    0.04 sec       99.1
```

Create a timetable with 30 seconds as the first row time. To specify a start time, use the `'StartTime'` name-value pair argument.

```
TT = timetable(Intensity,'SampleRate',100,'StartTime',seconds(30))
```

```
TT=5×2 timetable
      Time       Intensity
    _________    _________

    30 sec           100
    30.01 sec       98.7
    30.02 sec       95.2
    30.03 sec      101.4
    30.04 sec       99.1
```

Specify Time Step

Create a regular timetable using a time step of 0.01 seconds. You must specify the time step as a duration or calendar duration value.

```
Intensity = [100;98.7;95.2;101.4;99.1];
TT = timetable(Intensity,'TimeStep',seconds(0.01))
```

```
TT=5×2 timetable
      Time       Intensity
    ________     _________

    0 sec            100
    0.01 sec        98.7
    0.02 sec        95.2
    0.03 sec       101.4
    0.04 sec        99.1
```

Create a timetable with 30 seconds as the first row time. To specify a start time, use the `'StartTime'` name-value pair argument.

```
TT = timetable(Intensity,'TimeStep',seconds(0.01),'StartTime',seconds(30))
```

```
TT=5×2 timetable
      Time        Intensity
    _________     _________

    30 sec            100
    30.01 sec        98.7
    30.02 sec        95.2
    30.03 sec       101.4
    30.04 sec        99.1
```

Specify Size and Variable Types

Preallocate a table by specifying its size and the data types of the variables. The `timetable` function fills the variables with default values that are appropriate for the data types you specify. It also gives the variables default names.

```
T = [datetime('now') datetime(2017,11,1:3)];
sz = [4 3];
```

```
varTypes = {'double','double','string'};
TT = timetable('Size',sz,'VariableTypes',varTypes,'RowTimes',T)
```

```
TT=4×4 timetable
            Time            Var1    Var2      Var3
    ____________________    ____    ____    _________

    27-Aug-2019 03:01:34     0       0      <missing>
    01-Nov-2017 00:00:00     0       0      <missing>
    02-Nov-2017 00:00:00     0       0      <missing>
    03-Nov-2017 00:00:00     0       0      <missing>
```

To specify names for the variables, use the `'VariableNames'` name-value pair argument.

```
varNames = {'Temperature','WindSpeed','Station'};
TT = timetable('Size',sz,'VariableTypes',varTypes,'RowTimes',T,'VariableNames',varNames
```

```
TT=4×4 timetable
            Time            Temperature    WindSpeed     Station
    ____________________    ___________    _________    _________

    27-Aug-2019 03:01:34         0             0        <missing>
    01-Nov-2017 00:00:00         0             0        <missing>
    02-Nov-2017 00:00:00         0             0        <missing>
    03-Nov-2017 00:00:00         0             0        <missing>
```

Add a row of data to TT. Preallocation can be a useful technique when your code adds one row of data, or a few rows of data, at a time. Instead of growing the timetable every time you add a row, you can fill in table variables that already have room for your data. You can encapsulate a row of data values in a cell array, and assign it to a row of the timetable.

Subscript into a row by its time and assign a row of data values. You also can subscripting into rows and variables by number. However, subscripting into a timetable by time is a useful technique.

```
TT(datetime(2017,11,2),:) = {48.2,13.33,"S1"}
```

```
TT=4×4 timetable
            Time            Temperature    WindSpeed     Station
    ____________________    ___________    _________    _________

    27-Aug-2019 03:01:34         0              0      <missing>
```

```
    01-Nov-2017 00:00:00          0              0      <missing>
    02-Nov-2017 00:00:00       48.2          13.33      "S1"
    03-Nov-2017 00:00:00          0              0      <missing>
```

You can encapsulate a row of data values in a cell array. When you assign a row from a cell array, the assignment converts the cell array into a timetable row.

Preallocate Timetable Using Sample Rate

Specify a sample rate of 1000 Hz and preallocate a timetable. You also can specify a start time.

```
sz = [4 3];
varTypes = {'uint64','double','duration'};
TT = timetable('Size',sz,'VariableTypes',varTypes,'SampleRate',1000,'StartTime',seconds
```

```
TT=4×4 timetable
       Time       Var1    Var2      Var3
    __________    ____    ____    ________

    15 sec         0       0      00:00:00
    15.001 sec     0       0      00:00:00
    15.002 sec     0       0      00:00:00
    15.003 sec     0       0      00:00:00
```

Index into the third row, by specifying its time, and add a row of data.

```
TT(seconds(15.002),:) = {50,1.37,minutes(76)}
```

```
TT=4×4 timetable
       Time       Var1    Var2      Var3
    __________    ____    ____    ________

    15 sec          0        0    00:00:00
    15.001 sec      0        0    00:00:00
    15.002 sec     50     1.37    01:16:00
    15.003 sec      0        0    00:00:00
```

Preallocate Timetable Using Time Step

Specify a time step, and names for the variables.

```
sz = [3 2];
varTypes = {'double','double'};
TT = timetable('Size',sz,'VariableTypes',varTypes,'TimeStep',seconds(0.1),'VariableName
```

```
TT=3×3 timetable
     Time      Intensity    Distance
    _______    _________    ________

    0 sec          0           0
    0.1 sec        0           0
    0.2 sec        0           0
```

Index into the second row, by specifying its time, and add a row of data.

```
TT(seconds(0.1),:) = {93.6,11.27}
```

```
TT=3×3 timetable
     Time      Intensity    Distance
    _______    _________    ________

    0 sec           0            0
    0.1 sec      93.6        11.27
    0.2 sec         0            0
```

Create Timetable and Specify Variable Names

Create a timetable and specify the names of the timetable variables. The vector of row times is a duration vector, whose units are seconds.

```
Time = seconds(1:5)';
TT = timetable(Time,[98;97.5;97.9;98.1;97.9],[120;111;119;117;116],...
               'VariableNames',{'Reading1','Reading2'})
```

```
TT=5×3 timetable
    Time     Reading1    Reading2
    _____    ________    ________
```

```
    1 sec        98        120
    2 sec      97.5        111
    3 sec      97.9        119
    4 sec      98.1        117
    5 sec      97.9        116
```

Starting in R2019b you can specify timetable variable names that are not valid MATLAB® identifiers. Such variable names can include spaces, non-ASCII characters, and can have any character as the leading character.

For example, use dates as the names of the timetable variables that contain the readings.

```
TT = timetable(Time,[98;97.5;97.9;98.1;97.9],[120;111;119;117;116],...
               'VariableNames',{'29-May-2019','30-May-2019'})
```

```
TT=5×3 timetable
    Time     29-May-2019    30-May-2019
    _____    ___________    ___________

    1 sec         98            120
    2 sec       97.5            111
    3 sec       97.9            119
    4 sec       98.1            117
    5 sec       97.9            116
```

To use dot notation when the name is not a valid identifier, include parentheses and quotation marks.

```
TT.('29-May-2019')
```

```
ans = 5×1

   98.0000
   97.5000
   97.9000
   98.1000
   97.9000
```

Limitations

- Use single quotes for these input names:
  - `'RowTimes'`
  - `'SampleRate'`
  - `'Size'`
  - `'StartTime'`
  - `'TimeStep'`
  - `'VariableTypes'`
  - `'VariableNames'`

  To avoid confusion with variable inputs, do not use double-quoted string scalars (such as `"RowTimes"`) for these names.

Tips

- For a list of functions that accept or return timetables, see “Timetables”.
- In certain cases, you can call `timetable` with a syntax that specifies a regular time step between row times, and yet `timetable` returns an irregular timetable. This result occurs when you specify the time step using a calendar unit of time and there is a row time that introduces an irregular step. For example, if you create a timetable with a time step of one calendar month, starting on January 31, 2019, then it is irregular with respect to months.

```
stime = datetime(2019,1,31);
tstep = calmonths(1);
TT = timetable('Size',[3 1],'VariableTypes',{'double'},...
               'TimeStep',tstep,'StartTime',stime)
```

```
TT =

  3×1 timetable

       Time        Var1
    ___________    ____

    31-Jan-2019     0
```

```
28-Feb-2019     0
31-Mar-2019     0
```

In addition, there are other cases where irregularities are due to shifts from Daylight Saving Time (DST) or to row times that are leap seconds. This table specifies the row time values and time steps that can produce irregular timetables unexpectedly.

Row Time Value	Time Step
Start time specified as the 29th, 30th, or 31st day of the month	Number of calendar months or quarters
Start time specified as February 29	Number of calendar years
Any row time occurring between 1:00 a.m. and 2:00 a.m. on a day shifting from DST to standard time (when row times are specified as datetime values whose time zone observes DST)	Number of calendar days or months
Any row time that is a leap second (when row times are specified as datetime values whose time zone is the `UTCLeapSeconds` time zone)	Time step specified in any calendar unit (days, weeks, months, quarters, or years)

Compatibility Considerations

Variable names can contain leading and trailing whitespace characters

Behavior changed in R2019b

Table and timetable variable names with leading or trailing whitespace characters are not modified.

In previous releases, leading and trailing whitespace characters were deleted from variable names when you specified them using the `'VariableNames'` name-value pair argument, or assigned them to the `VariableNames` property.

To manually remove such characters, first use the `strtrim` function on the names, then assign them as variable names to the table or timetable.

`'SamplingRate'` is not recommended

Not recommended starting in R2018b

The `'SamplingRate'` name-value pair argument is not recommended. Use `'SampleRate'` instead. The corresponding timetable property is also named `SampleRate`.

For backward compatibility, you still can specify `'SamplingRate'` as the name of the name-value pair. However, the value is assigned to the `SampleRate` property.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays with these usage notes.

- **Creation**. There are several ways to create a tall timetable:

  1. Convert an existing tall table using `table2timetable`.

     ```
     ds = datastore('data/folder/path.csv');
     tt = tall(ds);
     TT = table2timetable(tt);
     ```

     The default behavior is to use the first datetime or duration variable in the tall table `tt` for the row times. To specify the row times yourself, use the `'RowTimes'` name-value pair to specify either a tall datetime or a tall duration vector of row times.

     ```
     TT = table2timetable(tt,'RowTimes',rowTimes)
     ```

  2. Convert an existing tall array using `array2timetable`.
  3. Manually construct a tall timetable from the variables in a tall table using the `timetable` constructor.

     ```
     ds = datastore('data/folder/path.csv');
     tt = tall(ds);
     TT = timetable(rowTimes, tt.Var1, tt.Var2, ...)
     ```

4. Convert an in-memory timetable into a tall timetable using the syntax `TT = tall(tt)`.

- **Indexing**. The `timerange`, `withtol`, and `vartype` functions are supported for indexing into tall timetables.
- **Supported Functions**. These functions support tall timetables.

head	join	stack	topkrows
height	ndims	standardizeMissing	timetable2table
horzcat	numel	summary	unique
isempty	retime	synchronize	varfun
innerjoin	size	table2array	width
ismember	sortrows	table2cell	
ismissing	splitapply	tail	

For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- Only these syntaxes are supported:

  ```
  TT = timetable(rowTimes,var1,...,varN)
  ```

  ```
  TT = timetable(var1,...,varN,'RowTimes',rowTimes)
  ```

  ```
  TT = timetable( ___ ,'VariableNames',{'name1',...,'nameN'})
  ```
- All data variables must be distributed.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`addprop` | `array2timetable` | `rmprop` | `summary` | `table` | `table2timetable` | `timeseries` | `timetable2table` | `uitable`

Topics

"Create Timetables"
"Clean Timetable with Missing, Duplicate, or Nonuniform Times"
"Represent Dates and Times in MATLAB"

Introduced in R2016b

timetable2table

Convert timetable to table

Syntax

```
T = timetable2table(TT)
T = timetable2table(TT,'ConvertRowTimes',false)
```

Description

`T = timetable2table(TT)` converts the `M`-by-`N` timetable `TT` to an `M`-by-`(N+1)` table. The vector of row times from `TT` becomes the first variable in `T`.

To write a timetable out to a text or spreadsheet file, first convert it to a table with `timetable2table`. Then write the table to a file with the `writetable` function.

`T = timetable2table(TT,'ConvertRowTimes',false)` converts the `M`-by-`N` timetable `TT` to an `M`-by-`N` table. `timetable2table` discards the vector of row times from `TT`.

Examples

Convert Timetable to Table

Create a timetable and convert it to a table.

```
Time = datetime({'2015-12-18';'2015-12-19';'2015-12-20'});
Temp = [37.3;39.1;42.3];
Pressure = [29.4;29.6;30.0];
Precip = [0.1;0.9;0.0];
TT = timetable(Time,Temp,Pressure,Precip);
T = timetable2table(TT)
```

```
T=3×4 table
       Time        Temp    Pressure    Precip
```

```
    ___________    ____    ________    ______

    18-Dec-2015    37.3      29.4       0.1
    19-Dec-2015    39.1      29.6       0.9
    20-Dec-2015    42.3        30         0
```

Display the sizes of T and TT. T has one more variable than TT because `timetable2table` converts the row times of TT to a variable of T.

```
whos T TT
```

```
  Name      Size            Bytes  Class        Attributes

  T         3x4              1845  table
  TT        3x3              1615  timetable
```

Convert Timetable and Discard Row Times

Create a timetable.

```
Time = datetime({'2015-12-18';'2015-12-19';'2015-12-20'});
Temp = [37.3;39.1;42.3];
Pressure = [29.4;29.6;30.0];
Precip = [0.1;0.9;0.0];
TT = timetable(Time,Temp,Pressure,Precip)
```

```
TT=3×4 timetable
       Time        Temp    Pressure    Precip
    ___________    ____    ________    ______

    18-Dec-2015    37.3      29.4       0.1
    19-Dec-2015    39.1      29.6       0.9
    20-Dec-2015    42.3        30         0
```

Convert TT to a table and discard its row times.

```
T = timetable2table(TT,'ConvertRowTimes',false)
```

```
T=3×3 table
    Temp    Pressure    Precip
    ____    ________    ______
```

```
    37.3      29.4       0.1
    39.1      29.6       0.9
    42.3        30         0
```

Input Arguments

TT — Input timetable
timetable

Input timetable.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`array2timetable` | `isvarname` | `summary` | `table2timetable` | `timetable` | `uitable`

Topics

"Create Timetables"
"Select Timetable Data by Row Time and Variable Type"
"Access Data in Tables"
"Tables"
"Represent Dates and Times in MATLAB"

Introduced in R2016b

timezones

List time zones

Syntax

```
timezones
timezones(area)
```

```
T = ( ___ )
```

Description

`timezones` displays a list of all IANA time zones accepted by the `datetime` function. For each time zone, the list includes the offset from UTC (in hours, where east is positive) and the daylight saving time shift (in hours).

`timezones(area)` displays a list containing all IANA time zones in the specified geographic area.

`T = ( ___ )` returns the list in a table.

Examples

List All Time Zones in Australia

Store all Australian time zones accepted by the `datetime` function in a table.

```
T = timezones('Australia')
```

```
T=12×4 table
              Name                    Area       UTCOffset    DSTOffset
    ___________________________    _________    _________    _________

    {'Australia/Adelaide'   }    Australia       9.5            1
```

```
    {'Australia/Brisbane'   }    Australia        10            0
    {'Australia/Broken_Hill'}    Australia       9.5            1
    {'Australia/Currie'     }    Australia        10            1
    {'Australia/Darwin'     }    Australia       9.5            0
    {'Australia/Eucla'      }    Australia      8.75            0
    {'Australia/Hobart'     }    Australia        10            1
    {'Australia/Lindeman'   }    Australia        10            0
    {'Australia/Lord_Howe'  }    Australia      10.5          0.5
    {'Australia/Melbourne'  }    Australia        10            1
    {'Australia/Perth'      }    Australia         8            0
    {'Australia/Sydney'     }    Australia        10            1
```

Sort the table by UTC offset.

```
sortrows(T,'UTCOffset')
```

```
ans=12×4 table
              Name                    Area       UTCOffset    DSTOffset
    ________________________    _________    _________    _________

    {'Australia/Perth'      }    Australia         8            0
    {'Australia/Eucla'      }    Australia      8.75            0
    {'Australia/Adelaide'   }    Australia       9.5            1
    {'Australia/Broken_Hill'}    Australia       9.5            1
    {'Australia/Darwin'     }    Australia       9.5            0
    {'Australia/Brisbane'   }    Australia        10            0
    {'Australia/Currie'     }    Australia        10            1
    {'Australia/Hobart'     }    Australia        10            1
    {'Australia/Lindeman'   }    Australia        10            0
    {'Australia/Melbourne'  }    Australia        10            1
    {'Australia/Sydney'     }    Australia        10            1
    {'Australia/Lord_Howe'  }    Australia      10.5          0.5
```

Input Arguments

area — Geographic area
`'All'` | `'Africa'` | `'America'` | `'Antarctica'` | …

Geographic area, specified as one of the areas in the table.

'Africa'	'Asia'	'Europe'
'America'	'Atlantic'	'Indian'
'Antarctica'	'Australia'	'Pacific'
'Arctic'	'Etc'	'All'

See Also

datetime

Topics

"Specify Time Zones"

Introduced in R2015b

title

Add title

Syntax

```
title(txt)
title(target,txt)

title( ___ ,Name,Value)
t = title( ___ )
```

Description

`title(txt)` adds the specified title to the axes or chart returned by the `gca` command. Reissuing the `title` command causes the new title to replace the old title.

`title(target,txt)` adds the title to the specified target object.

`title( ___ ,Name,Value)` modifies the title appearance using one or more name-value pair arguments. For example, `'FontSize',12` sets the font size to 12 points. Specify name-value pair arguments after all other input arguments. Modifying the title appearance is not supported for all types of charts.

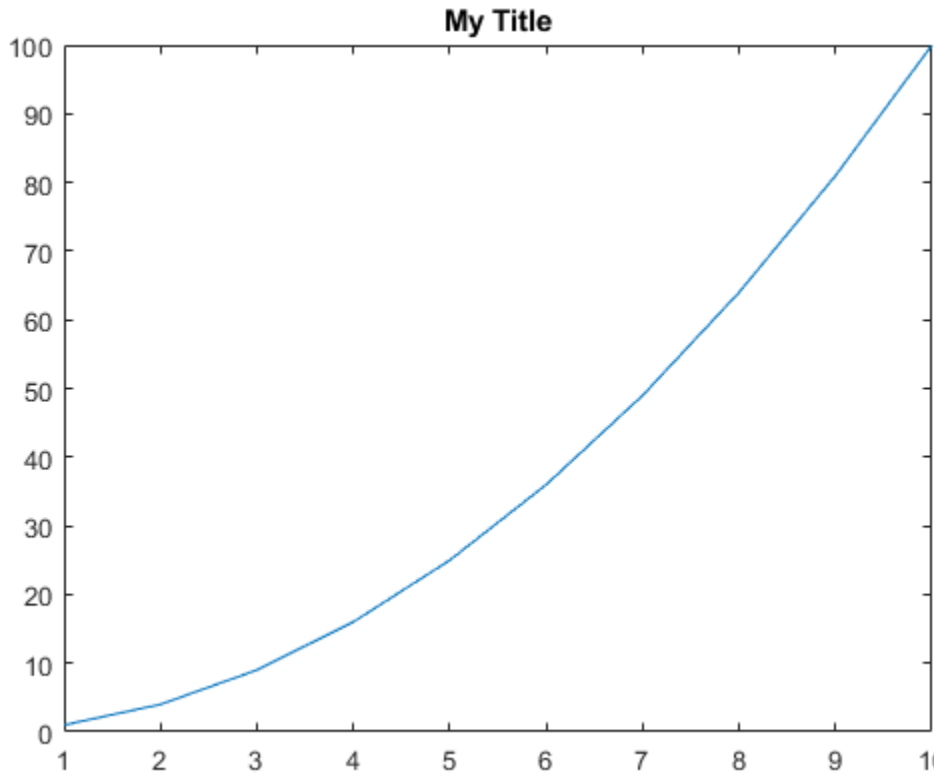
`t = title( ___ )` returns the object used for the title. Use `t` to make future modifications to the title.

Examples

Add Title to Current Axes

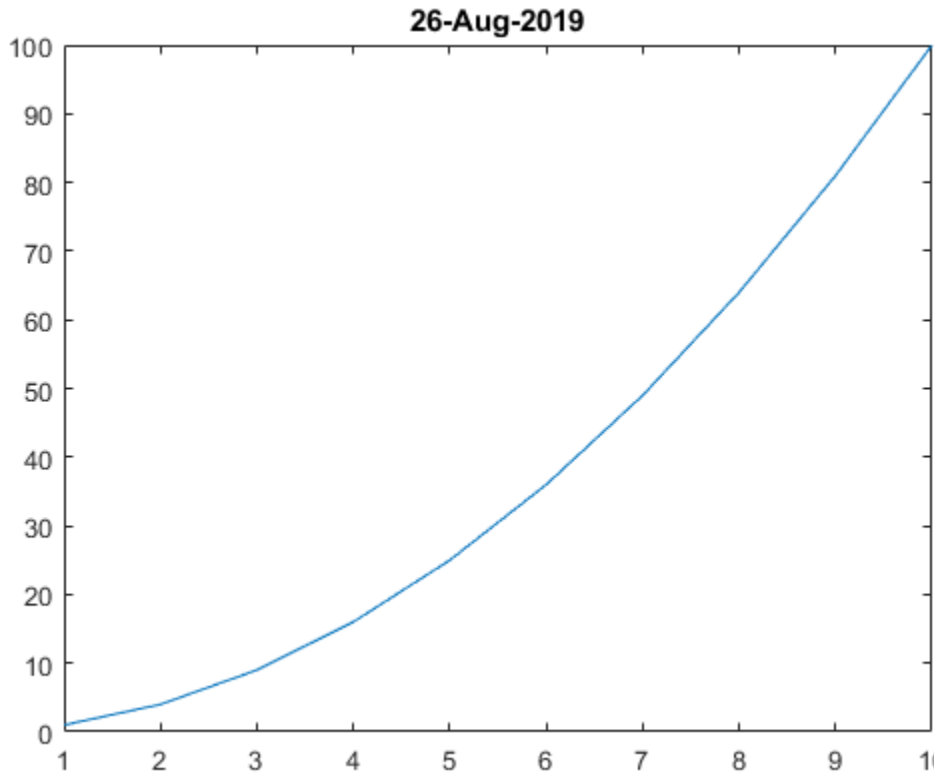
Create a figure and display a title in the current axes.

```
plot((1:10).^2)
title('My Title')
```



You also can call `title` with a function that returns text. For example, the `date` function returns text with today's date.

```
title(date)
```

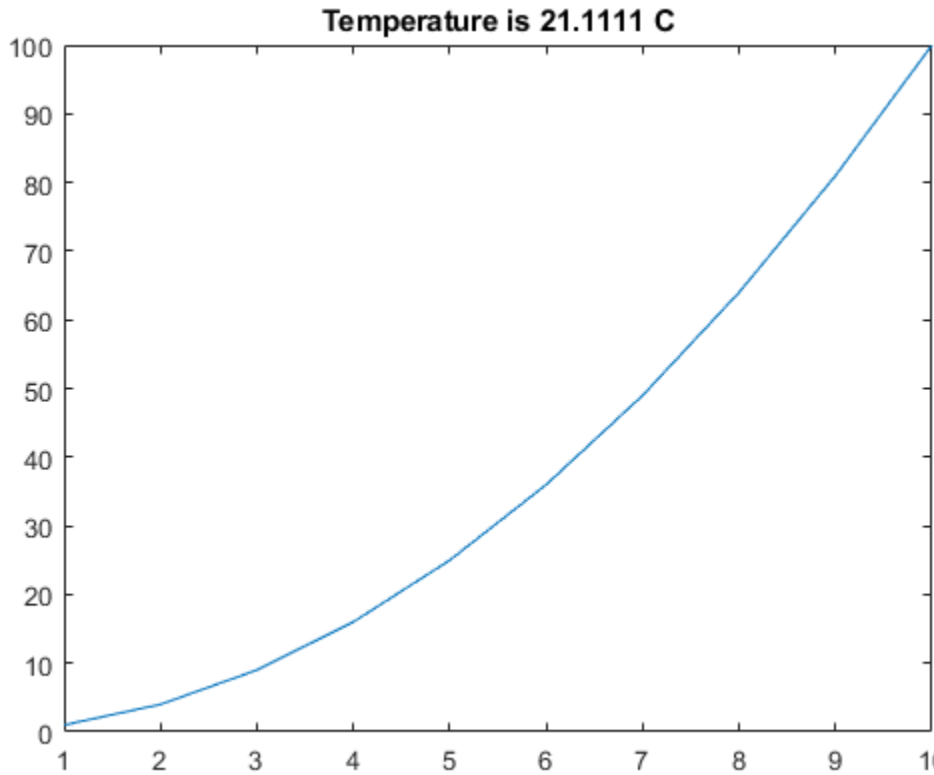


MATLAB® sets the output of `date` as the axes title.

Include Variable's Value in Title

Include the value of variable `c` in a title.

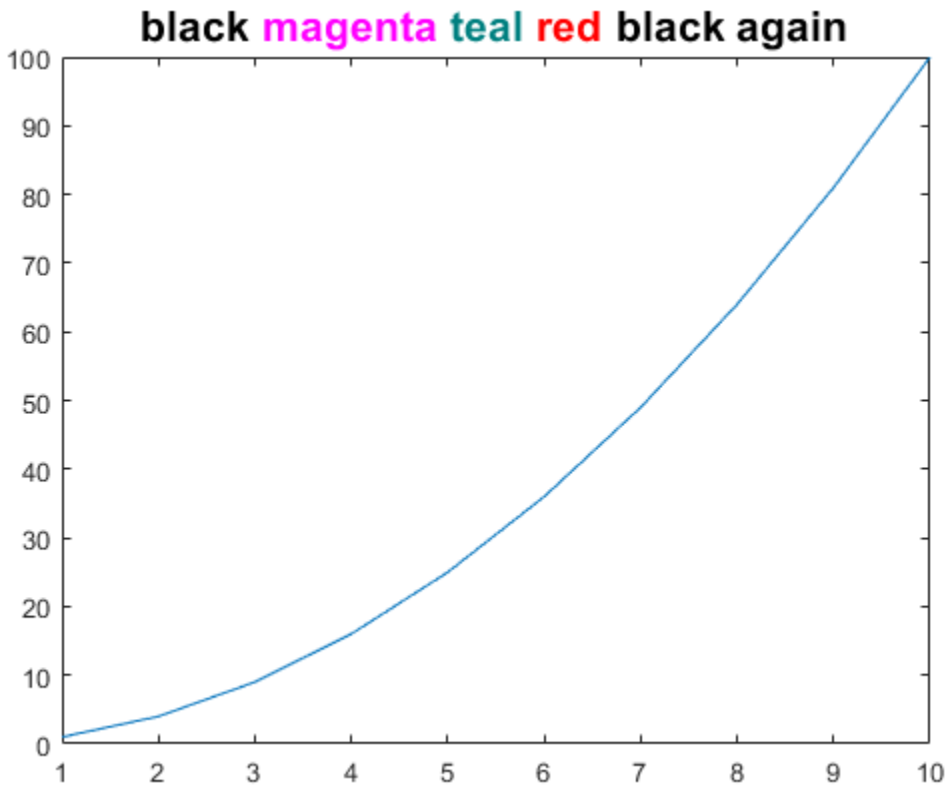
```
figure
plot((1:10).^2)
f = 70;
c = (f-32)/1.8;
title(['Temperature is ',num2str(c),' C'])
```



Create Multicolored Title Using TeX Markup

Use the color modifier `\color` to change the color of characters following it from the previous color.

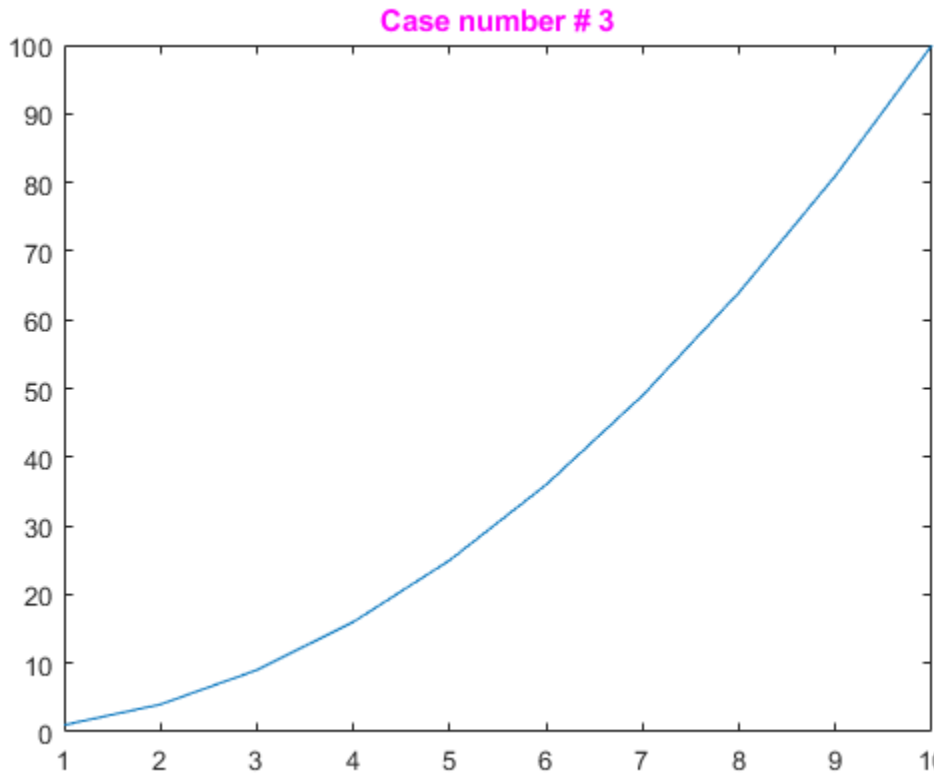
```
plot((1:10).^2)
title(['\fontsize{16}black {\color{magenta}magenta '...
'\color[rgb]{0 .5 .5}teal \color{red}red} black again'])
```



Create Colored Title Using Name,Value Pair Argument

Use the `Name,Value` pair `'Color','m'` to set the color of the title to magenta.

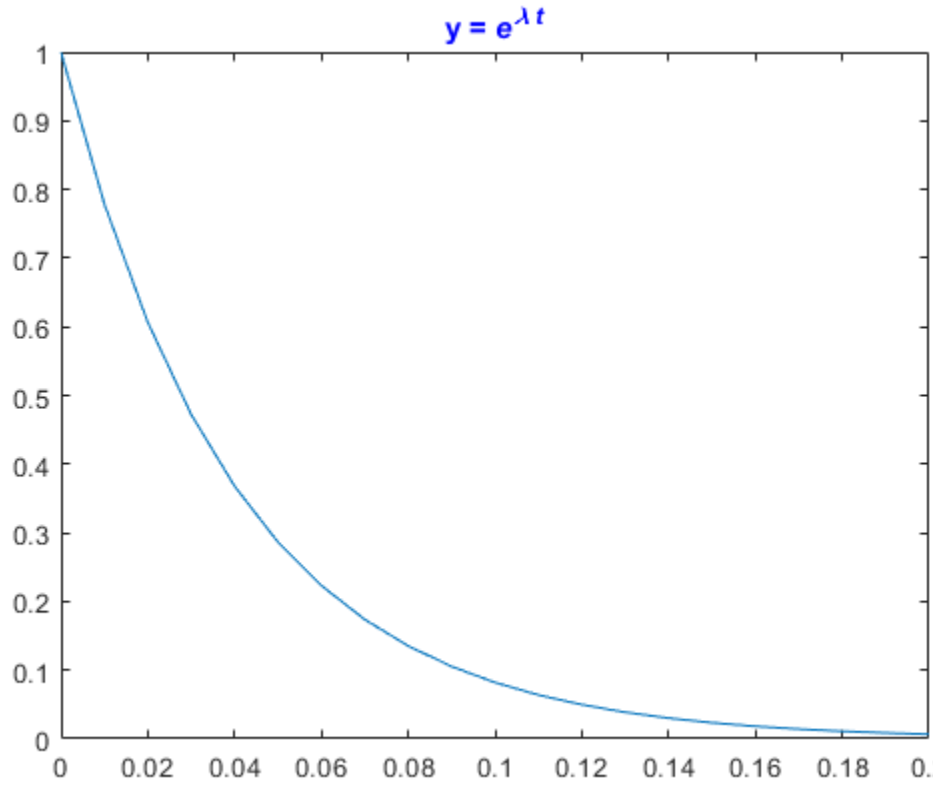
```
figure
plot((1:10).^2)
title('Case number # 3','Color', 'm')
```



Include Greek Symbols in Title

Use TeX markup to include Greek symbols in a title.

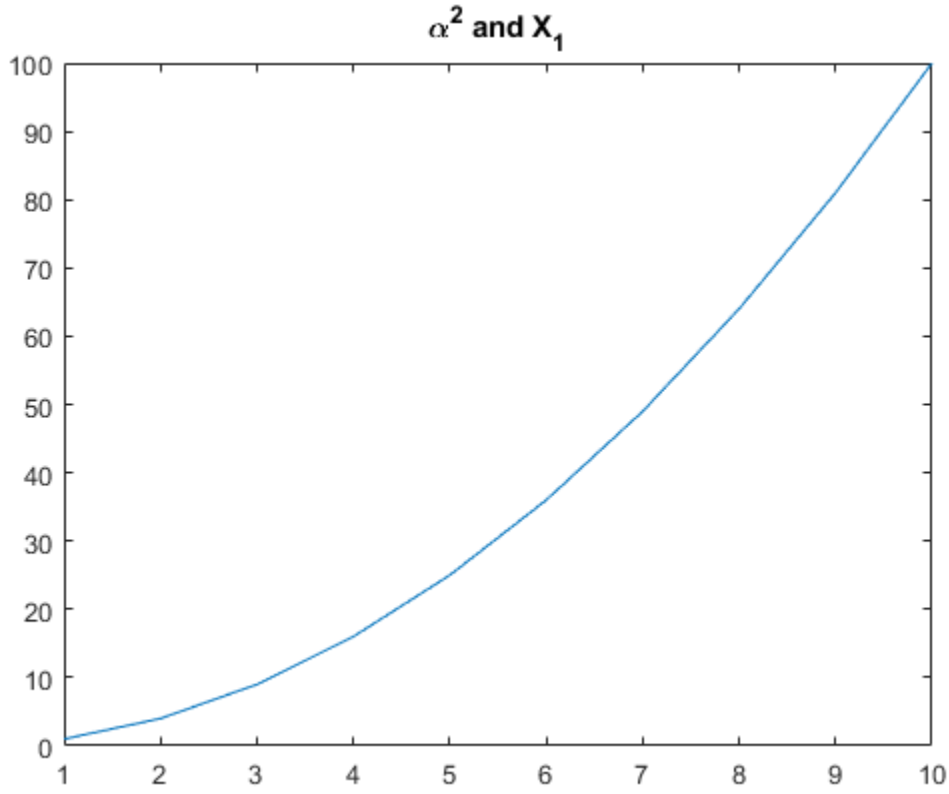
```
t = (0:0.01:0.2);
y = exp(-25*t);
figure
plot(t,y)
title('y = \ite^{\lambda t}','Color','b')
```



The `'Interpreter'` property must be `'tex'` (the default).

Include Superscript or Subscript Character in Title

```
figure
plot((1:10).^2)
title('\alpha^2 and X_1')
```

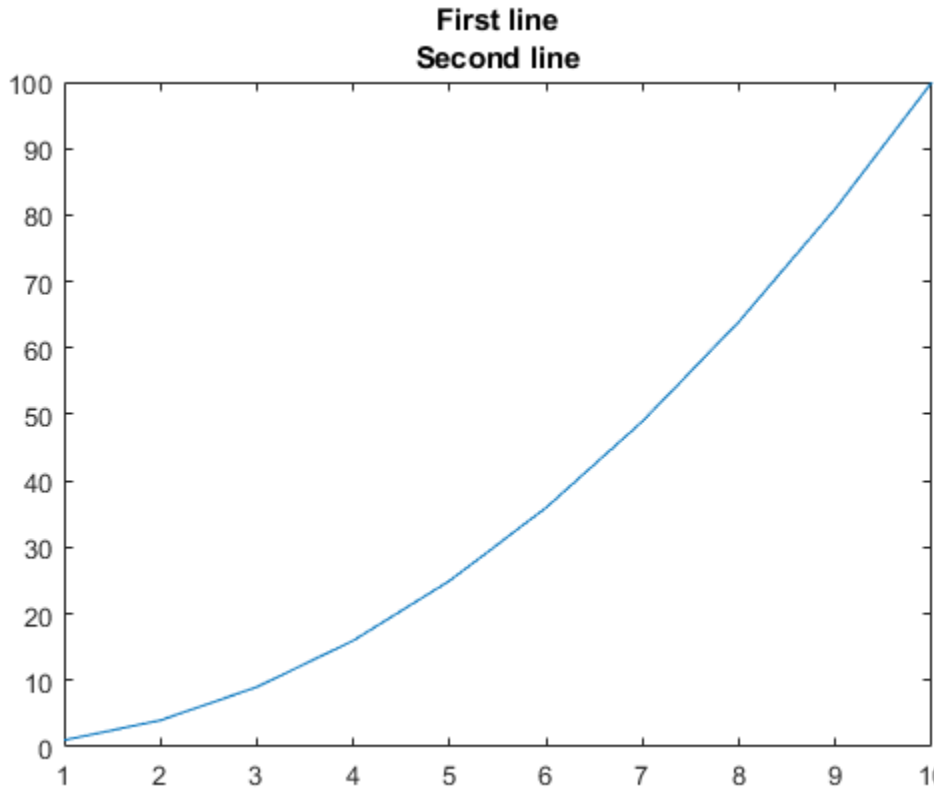


The superscript character, "^", and the subscript character, "_", modify the character or substring defined in braces immediately following.

Create Multiline Title

Create a multiline title using a multiline cell array.

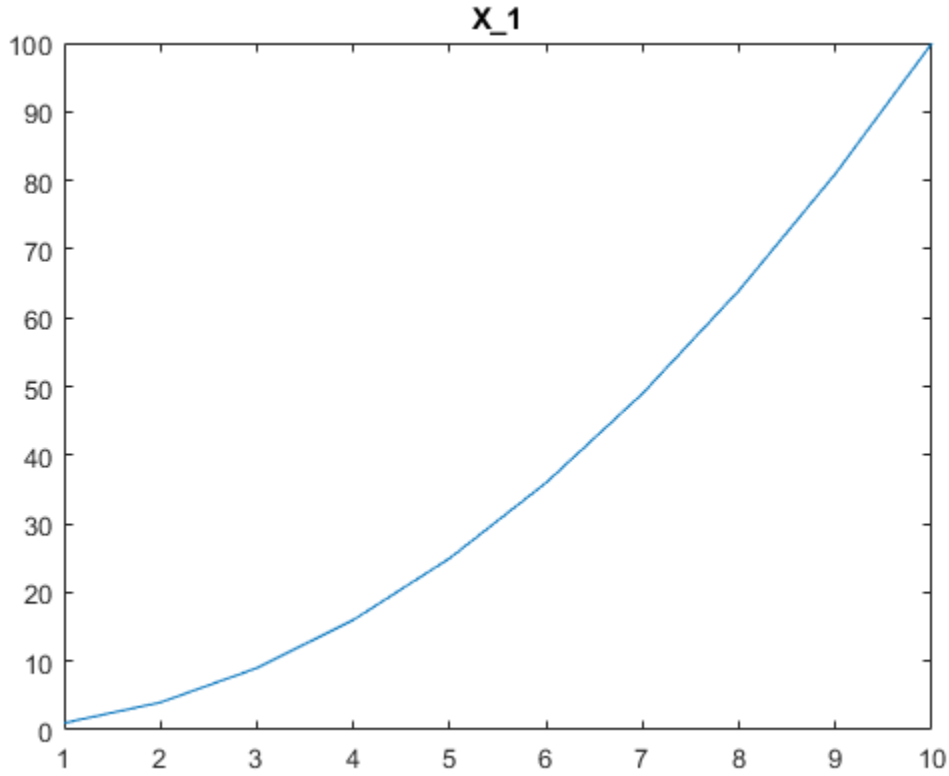
```
figure
plot((1:10).^2)
title({'First line';'Second line'})
```



Display Text As Typed

Set the `Interpreter` property as `'none'` so that the text `X_1` is displayed in the figure as typed, without making `1` a subscript of `X`.

```
plot((1:10).^2)
title('X_1','Interpreter','none')
```

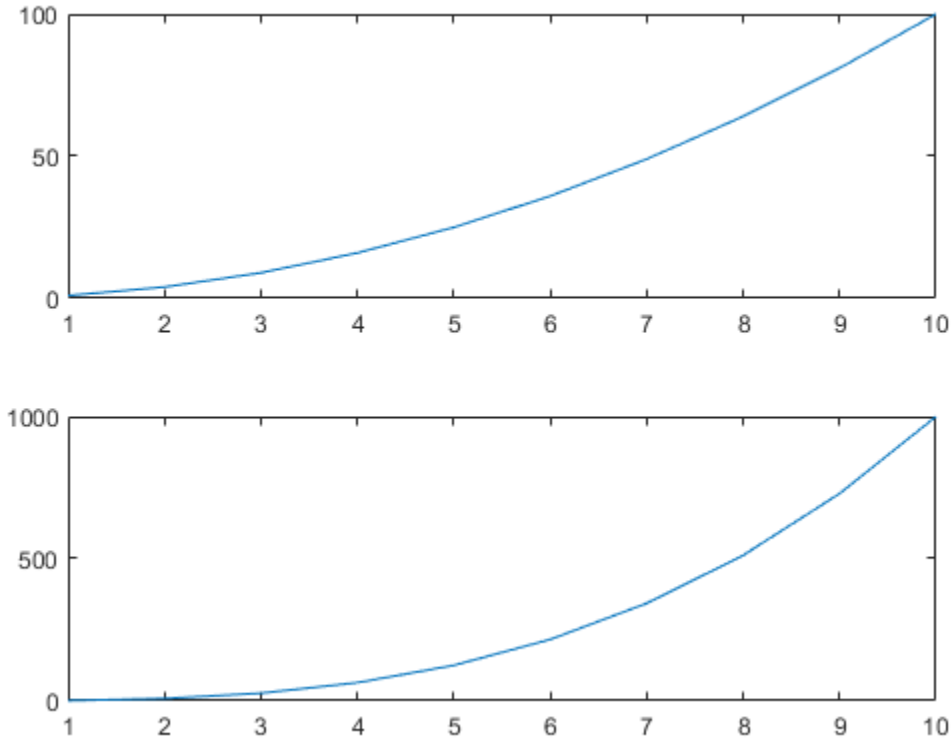


Add Title to Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Then plot data into each axes.

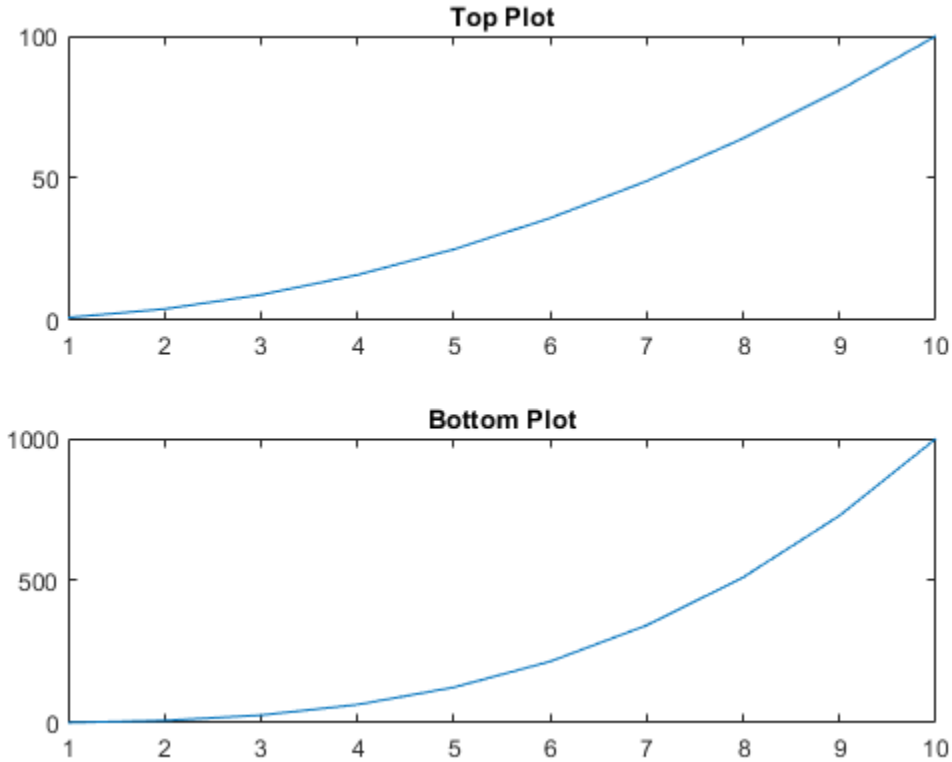
```
tiledlayout(2,1)
ax1 = nexttile;
plot(ax1,(1:10).^2)
```

```
ax2 = nexttile;
plot(ax2,(1:10).^3)
```



Add a title to each axes by passing `ax1` and `ax2` to the `title` function.

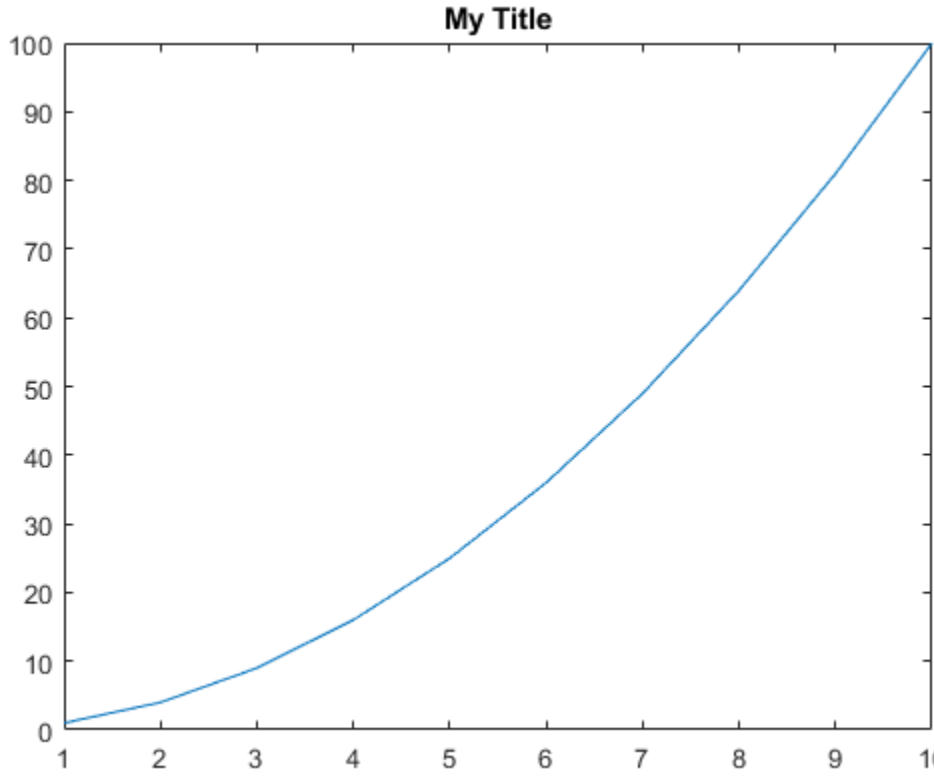
```
title(ax1,'Top Plot')
title(ax2,'Bottom Plot')
```



Add Title and Return Text Handle

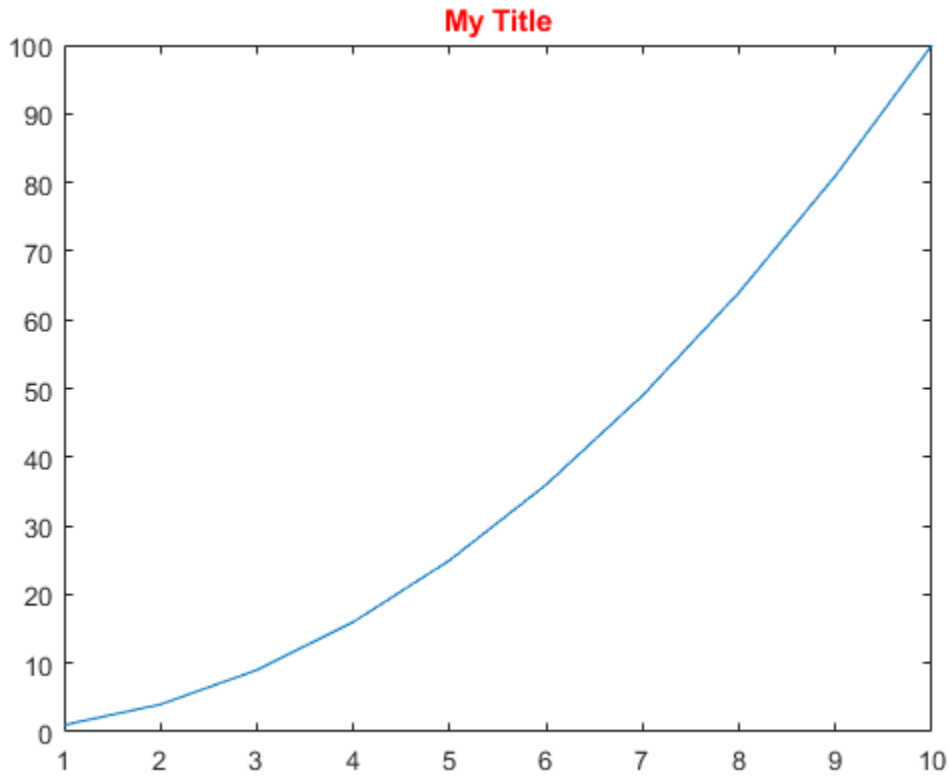
Add a title to a plot and return the text object.

```
plot((1:10).^2)
t = title('My Title');
```



Set the color of the title to red. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead.

```
t.Color = 'red';
```



Input Arguments

`txt` — Text to display as title

string scalar | character vector | character array | cell array | categorical array | numeric value

Text to display as title, specified as a string scalar, character vector, string array, character array, cell array, categorical array, or numeric value.

Example: `'my label'`

Example: `{'first line','second line'}`

Example: 123

To include numeric variables with text in a title, use the `num2str` function. For example:

```
x = 42;
str = ['The value is ',num2str(x)];
```

To include special characters, such as superscripts, subscripts, Greek letters, or mathematical symbols, use TeX markup. For a list of supported markup, see the `Interpreter` property.

To create multiline titles:

- Use a string array where each element contains a line of text, such as `["first line", "second line"]`.
- Use a cell array where each cell contains a line of text, such as `{'first line','second line'}`.
- Use a character array where each row contains a line of text, such as `['abc'; 'ab ']`. If you use this technique, each row must have the same number of characters.
- Use `sprintf` to create a character vector with a new line character, such as `sprintf('first line \n second line')`.

Numeric titles are converted to text using `sprintf('%g',value)`. For example, `12345678` displays as `1.23457e+07`.

Note

- If you specify the label as a categorical array, MATLAB uses the values in the array, not the categories.
- The words `default`, `factory`, and `remove` are reserved words that will not appear in a title when quoted as a normal character vector. To display any of these words individually, precede them with a backslash, such as `'\default'` or `'\remove'`.

`target` — Target for title

axes | legend | tiled chart layout | graphics object

Target for the title, specified as any of the following objects:

- Any type of axes: an `Axes`, `PolarAxes`, or `GeographicAxes` object
- A `Legend` object
- A `TiledChartLayout` object
- A graphics object that has a `Title` property. For example, you can add a title to a `HeatmapChart` object.

If you do not specify the target for the title, then the `title` function adds the title to the graphics object returned by the `gca` command.

Note Some charts do not support modifying the title appearance, such as the color, or returning the text object as an output argument.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Color','red','FontSize',14` adds a title with red, 14-point font.

In addition to the following, you can specify other text object properties using `Name,Value` pair arguments. See Text.

`FontSize` — Font size
`11` (default) | scalar value greater than 0

Font size, specified as a scalar value greater than `0` in point units. One point equals 1/72 inch. To change the font units, use the `FontUnits` property.

If you add a title to an axes object, then the font size properties for the axes also affect the title font size. The title font size updates to equal the axes font size multiplied by the title scale factor. The `FontSize` property of the axes contains the axes font size. The `TitleFontSizeMultiplier` property of the axes contains the scale factor. By default, the axes font size is 10 points and the scale factor is 1.1, so the title font size is 11 points.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`FontWeight` — Thickness of text characters

`'bold'` (default) | `'normal'`

Thickness of the text characters, specified as one of these values:

- `'bold'` — Thicker characters outlines than normal
- `'normal'` — Normal weight as defined by the particular font

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold font weight. Therefore, specifying a bold font weight could still result in the normal font weight.

If you add a title to an axes object, then the `TitleFontWeight` property for the associated axes affects the `FontWeight` value for the title.

Note The `'light'` and `'demi'` font weight values have been removed. Use `'normal'` instead.

`FontName` — Font name

supported font name | `'FixedWidth'`

Font name, specified as a supported font name or `'FixedWidth'`. To display and print text properly, you must choose a font that your system supports. The default font depends on your operating system and locale.

To use a fixed-width font that looks good in any locale, use `'FixedWidth'`. The fixed-width font relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`Color` — Text color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Text color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`Interpreter` — Interpretation of text characters

`'tex'` (default) | `'latex'` | `'none'`

Interpretation of text characters, specified as one of these values:

- `'tex'` — Interpret characters using a subset of TeX markup.
- `'latex'` — Interpret characters using LaTeX markup.
- `'none'` — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text.

Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they modify only the next character or the characters within the curly braces. When you set the `Interpreter` property to `'tex'`, the supported modifiers are as follows.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{`*`specifier`*`}`	Font name — Replace *`specifier`* with the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`

Modifier	Description	Example
`\fontsize{`*`specifier`*`}`	Font size —Replace *`specifier`* with a numeric scalar value in point units.	`'\fontsize{15} text'`
`\color{`*`specifier`*`}`	Font color — Replace *`specifier`* with one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Replace *`specifier`* with a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	β	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ϵ	`\Delta`	Δ	`\spadesuit`	♠
`\zeta`	ζ	`\Theta`	Θ	`\leftrightarrow`	↔
`\eta`	η	`\Lambda`	Λ	`\leftarrow`	←
`\theta`	θ	`\Xi`	Ξ	`\Leftarrow`	⇐
`\vartheta`	ϑ	`\Pi`	Π	`\uparrow`	↑

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	μ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. Use dollar symbols around the text, for example, use `'$\int_1^{20} x^2 dx$'` for inline mode or `'$$ \int_1^{20} x^2 dx$$'` for display mode.

The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup.

The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters. For multiline text, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project website at https://www.latex-project.org/.

Output Arguments

`t` — Object used for title text
text object

Object used for the title text, returned as one of these types of objects:

- Text object — If you add a title to axes, then `title` returns a text object. Use this text object to access and modify properties of the title after it is created. For a list of text object properties, see Text. You also can access the title through the `Title` property of the axes object.
- Legend text object — If you add a title to a legend, then `title` returns a legend text object. Use this legend text object to access and modify properties of the title after it is created. For a list of legend text object properties, see Legend Text. You also can access the title through the `Title` property of the legend object.

See Also

Functions
`num2str` | `text` | `xlabel` | `ylabel` | `zlabel`

Properties
Text

Topics

"Add Title and Axis Labels to Chart"

Introduced before R2006a

todatenum

Convert CDF epoch object to MATLAB serial date number

Syntax

```
n = todatenum(obj)
```

Description

`n = todatenum(obj)` converts the CDF epoch object `ep_obj` into a MATLAB serial date number. Note that a CDF epoch is the number of milliseconds since `01-Jan-0000` whereas a MATLAB datenum is the number of days since 00`-Jan-0000`.

Examples

Construct a CDF epoch object from text that represents a date. Then, convert the object back into its original form.

```
dstr = datestr(today)
dstr =
   08-Oct-2003

obj = cdfepoch(dstr)
obj =
     cdfepoch object:
     08-Oct-2003 00:00:00

dstr2 = datestr(todatenum(obj))
dstr2 =
   08-Oct-2003
```

See Also

`cdfepoch` | `cdfinfo` | `cdfread` | `datenum` | `datetime`

Introduced before R2006a

toeplitz

Toeplitz matrix

Syntax

```
T = toeplitz(c,r)
T = toeplitz(r)
```

Description

`T = toeplitz(c,r)` returns a nonsymmetric Toeplitz matrix on page 1-15845 with `c` as its first column and `r` as its first row. If the first elements of `c` and `r` differ, `toeplitz` issues a warning and uses the column element for the diagonal.

`T = toeplitz(r)` returns the symmetric Toeplitz matrix where:

- If `r` is a real vector, then `r` defines the first row of the matrix.
- If `r` is a complex vector with a real first element, then `r` defines the first row and `r'` defines the first column.
- If the first element of `r` is complex, the Toeplitz matrix is Hermitian off the main diagonal, which means $T_{i,j} = \mathrm{conj}(T_{j,i})$ for $i \neq j$. The elements of the main diagonal are set to `r(1)`.

Examples

Create Symmetric Toeplitz Matrix

```
r = [1 2 3];
toeplitz(r)
```

```
ans = 3×3

     1     2     3
```

```
     2     1     2
     3     2     1
```

Create Nonsymmetric Toeplitz Matrix

Create a nonsymmetric Toeplitz matrix with a specified column and row vector. Because the first elements of the column and row vectors do not match, `toeplitz` issues a warning and uses the column for the diagonal element.

```
c = [1  2  3 4];
r = [4 5 6];
toeplitz(c,r)
```

```
Warning: First element of input column does not match first element of input row. <br>
```

```
ans = 4×3

     1     5     6
     2     1     5
     3     2     1
     4     3     2
```

Create a Toeplitz matrix with complex row and column vectors.

```
c = [1+3i 2-5i -1+3i];
r = [1+3i 3-1i -1-2i];
T = toeplitz(c,r)
```

```
T = 3×3 complex

   1.0000 + 3.0000i   3.0000 - 1.0000i  -1.0000 - 2.0000i
   2.0000 - 5.0000i   1.0000 + 3.0000i   3.0000 - 1.0000i
  -1.0000 + 3.0000i   2.0000 - 5.0000i   1.0000 + 3.0000i
```

Create Circulant Matrices Using `toeplitz` Function

You can create circulant matrices using `toeplitz`. Circulant matrices are used in applications such as circular convolution.

Create a circulant matrix from vector `v` using toeplitz.

```
v = [9 1 3 2];
toeplitz([v(1) fliplr(v(2:end))], v)
```

```
ans = 4×4

     9     1     3     2
     2     9     1     3
     3     2     9     1
     1     3     2     9
```

Perform discrete-time circular convolution by using `toeplitz` to form the circulant matrix for convolution.

Define the periodic input `x` and the system response `h`.

```
x = [1 8 3 2 5];
h = [3 5 2 4 1];
```

Form the column vector `c` to create a circulant matrix where `length(c) = length(h)`.

```
c = [x(1) fliplr(x(end-length(h)+2:end))]
```

```
c = 1×5

     1     5     2     3     8
```

Form the row vector `r` from `x`.

```
r = x;
```

Form the convolution matrix `xConv` using `toeplitz`. Find the convolution using `h*xConv`.

```
xConv = toeplitz(c,r)
```

```
xConv = 5×5
```

```
     1     8     3     2     5
     5     1     8     3     2
     2     5     1     8     3
     3     2     5     1     8
     8     3     2     5     1
```

```
h*xConv
```

```
ans = 1×5

    52    50    73    46    64
```

If you have the Signal Processing Toolbox™, you can use the `cconv` function to find the circular convolution.

Discrete-Time Convolution Using `Toeplitz`

Perform discrete-time convolution by using `toeplitz` to form the arrays for convolution.

Define the input `x` and system response `h`.

```
x = [1 8 3 2 5];
h = [3 5 2];
```

Form `r` by padding `x` with zeros. The length of `r` is the convolution length `x + h - 1`.

```
r = [x zeros(1,length(h)-1)]
```

```
r = 1×7

     1     8     3     2     5     0     0
```

Form the column vector `c`. Set the first element to `x(1)` because the column determines the diagonal. Pad `c` because `length(c)` must equal `length(h)` for convolution.

```
c = [x(1) zeros(1,length(h)-1)]
```

```
c = 1×3
```

```
     1     0     0
```

Form the convolution matrix `xConv` using `toeplitz`. Then, find the convolution using `h*xConv`.

```
xConv = toeplitz(c,r)
```

```
xConv = 3×7

     1     8     3     2     5     0     0
     0     1     8     3     2     5     0
     0     0     1     8     3     2     5
```

```
h*xConv
```

```
ans = 1×7

     3    29    51    37    31    29    10
```

Check that the result is correct using `conv`.

```
conv(x,h)
```

```
ans = 1×7

     3    29    51    37    31    29    10
```

Input Arguments

c — Column of Toeplitz matrix
scalar | vector

Column of Toeplitz matrix, specified as a scalar or vector. If the first elements of `c` and `r` differ, `toeplitz` uses the column element for the diagonal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`
Complex Number Support: Yes

`r` — Row of Toeplitz matrix
scalar | vector

Row of Toeplitz matrix, specified as a scalar or vector. If the first elements of `c` and `r` differ, then `toeplitz` uses the column element for the diagonal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`
Complex Number Support: Yes

More About

Toeplitz Matrix

A Toeplitz matrix is a diagonal-constant matrix, which means all elements along a diagonal have the same value. For a Toeplitz matrix A, we have $A_{i,j} = a_{i-j}$ which results in the form

$$A = \begin{bmatrix} a_0 & a_{-1} & a_{-2} & \cdots & \cdots & a_{1-n} \\ a_1 & a_0 & a_{-1} & \ddots & \ddots & \vdots \\ a_2 & a_1 & a_0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & a_{-2} \\ \vdots & \ddots & \ddots & \ddots & a_0 & a_{-1} \\ a_{n-1} & \cdots & \cdots & a_2 & a_1 & a_0 \end{bmatrix}.$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`hankel` | `kron`

Introduced before R2006a

ToolbarDropdown Properties

Modify drop-down menus in axes toolbar

Description

`ToolbarDropdown` properties control the appearance and behavior of the drop-down menus in an axes toolbar. By changing property values, you can modify certain aspects of the drop-down menu.

Properties

Button

`Icon` — Button icon

`'none'` (default) | file name | m-by-n-by-3 array | m-by-n matrix | predefined icon

Button icon, specified as one of these values. The axes toolbar displays icons as grayscale images. MATLAB scales down the image to fit, if necessary.

- File name — Specify the file name as a character vector or a string scalar. The file name can be preceded by a path. The image file must be in JPEG, GIF, or PNG format.
- Array — Specify an m-by-n-by-3 array of RGB triplets that represent pixels in an image.
- Matrix — Specify an m-by-n matrix of numeric values that represent pixels in an image.
  - If the values are of an integer type, then specify values between `0` and `63`. A value of `0` is opaque and a value of `63` is transparent. `NaN` is also transparent.
  - If the values are of type `double`, then specify values between `1` and `64`. A value of `1` is opaque and a value of `64` is transparent. `NaN` is also transparent.
- Predefined icon — Specify one of the names in this table.

Icon Name	Result
`'brush'`	
`'datacursor'`	
`'export'`	
`'rotate'`	
`'pan'`	
`'zoomin'`	
`'zoomout'`	
`'restoreview'`	
`'none'`	No icon

Example: `btn.Icon = 'icon.png'`

Example: `btn.Icon = 'C:\Documents\icon.png'`

Example: `btn.Icon = 'rotate'`

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

Callbacks

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.

- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.

- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent
`AxesToolbar` object

Parent container, specified as an `AxesToolbar` object.

`Children` — Child buttons
graphics array

Child buttons, specified as a graphics array of `ToolbarPushButton` objects. Use this property to view the list of buttons contained in the drop-down menu.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object
`'toolbardropdown'` (default)

This property is read-only.

Type of graphics object, returned as `'toolbardropdown'`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

UserData — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

Unused Properties

UIContextMenu — Context menu

`uicontextmenu` object

Context menu, specified as a `ContextMenu` object.

Note The drop-down does not use this property. Setting this property has no effect.

ButtonDownFcn — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as a function handle, a cell array, or a character vector.

Note The drop-down does not use this property. Setting this property has no effect.

See Also

Topics

"Access Property Values"
"Graphics Object Properties"

Introduced in R2019b

ToolbarStateButton Properties

Axes toolbar state button appearance and behavior

Description

`ToolbarStateButton` properties control the appearance and behavior of the axes `ToolbarStateButton` object. By changing property values, you can modify certain aspects of the state button.

```
tb = axtoolbar('default')
btn = axtoolbarbtn(tb,'state')
btn.Tooltip = 'My State Button';
```

Properties

Button

`Icon` — Button icon

`'none'` (default) | file name | m-by-n-by-3 array | m-by-n matrix | predefined icon

Button icon, specified as one of these values. The axes toolbar displays icons as grayscale images. MATLAB scales down the image to fit, if necessary.

- File name — Specify the file name as a character vector or a string scalar. The file name can be preceded by a path. The image file must be in JPEG, GIF, or PNG format.
- Array — Specify an m-by-n-by-3 array of RGB triplets that represent pixels in an image.
- Matrix — Specify an m-by-n matrix of numeric values that represent pixels in an image.
  - If the values are of an integer type, then specify values between `0` and `63`. A value of `0` is opaque and a value of `63` is transparent. `NaN` is also transparent.
  - If the values are of type `double`, then specify values between `1` and `64`. A value of `1` is opaque and a value of `64` is transparent. `NaN` is also transparent.
- Predefined icon — Specify one of the names in this table.

Icon Name	Result
`'brush'`	
`'datacursor'`	
`'export'`	
`'rotate'`	
`'pan'`	
`'zoomin'`	
`'zoomout'`	
`'restoreview'`	
`'none'`	No icon

Example: `btn.Icon = 'icon.png'`

Example: `btn.Icon = 'C:\Documents\icon.png'`

Example: `btn.Icon = 'rotate'`

`Tooltip` — Button tool tip
`''` (default) | character vector | cell array of character vectors | string array

Button tool tip, specified as a character vector, cell array of character vectors, or a string array. Use this property to display a message when you hover the pointer over the button. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array displays a separate line of text.

Example: `btn.Tooltip = 'My Tooltip'`

`Value` — Current state of button
`'on'` | `'off'`

Current state of the button, specified as `'on'` (pressed) or `'off'` (depressed).

`Visible` — State of visibility
`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

Callbacks

`ValueChangedFcn` — State button callback

`''` (default) | function handle | cell array | character vector

State button callback that executes when you click the button, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when you click a state button. It does not execute if a state button `Value` property changes programmatically.

If you specify this property using a function handle, then MATLAB passes the source `StateButton` object and an event data structure as the first and second input arguments to the function. This table describes the fields in the event data structure.

Field	Description
`Source`	`StateButton` object
`Axes`	Associated `Axes` object
`EventName`	`'ValueChanged'`
`Value`	Current value (either `'on'` or `'off'`)
`PreviousValue`	Previous value (either `'on'` or `'off'`)

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.

- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent
`AxesToolbar` object

Parent container, specified as an `AxesToolbar` object.

`Children` — Children
empty `GraphicsPlaceholder` array

This object has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object
`'toolbarstatebutton'` (default)

This property is read-only.

Type of graphics object, returned as `'toolbarstatebutton'`.

`Tag` — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

Unused Properties

`UIContextMenu` — Context menu
`uicontextmenu` object

Context menu, specified as a `ContextMenu` object.

Note The button does not use this property.

`ButtonDownFcn` — Mouse-click callback
`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as a function handle, a cell array, or a character vector.

Note The button does not use this property.

See Also

`axtoolbarbtn`

Topics

"Access Property Values"
"Graphics Object Properties"

Introduced in R2018b

ToolbarPushButton Properties

Axes toolbar push button appearance and behavior

Description

`ToolbarPushButton` properties control the appearance and behavior of the axes `ToolbarPushButton` object. By changing property values, you can modify certain aspects of the push button.

```
tb = axtoolbar('default')
btn = axtoolbarbtn(tb,'push')
btn.Tooltip = 'My Push Button';
```

Properties

Button

`Icon` — Button icon

`'none'` (default) | file name | m-by-n-by-3 array | m-by-n matrix | predefined icon

Button icon, specified as one of these values. The axes toolbar displays icons as grayscale images. MATLAB scales down the image to fit, if necessary.

- File name — Specify the file name as a character vector or a string scalar. The file name can be preceded by a path. The image file must be in JPEG, GIF, or PNG format.
- Array — Specify an m-by-n-by-3 array of RGB triplets that represent pixels in an image.
- Matrix — Specify an m-by-n matrix of numeric values that represent pixels in an image.
  - If the values are of an integer type, then specify values between `0` and `63`. A value of `0` is opaque and a value of `63` is transparent. `NaN` is also transparent.
  - If the values are of type `double`, then specify values between `1` and `64`. A value of `1` is opaque and a value of `64` is transparent. `NaN` is also transparent.
- Predefined icon — Specify one of the names in this table.

Icon Name	Result
`'brush'`	
`'datacursor'`	
`'export'`	
`'rotate'`	
`'pan'`	
`'zoomin'`	
`'zoomout'`	
`'restoreview'`	
`'none'`	No icon

Example: `btn.Icon = 'icon.png'`

Example: `btn.Icon = 'C:\Documents\icon.png'`

Example: `btn.Icon = 'rotate'`

`Tooltip` — Button tool tip

`''` (default) | character vector | cell array of character vectors | string array

Button tool tip, specified as a character vector, cell array of character vectors, or a string array. Use this property to display a message when you hover the pointer over the button. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array displays a separate line of text.

Example: `btn.Tooltip = 'My Tooltip'`

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

Callbacks

`ButtonPushedFcn` — Push button callback

`''` (default) | function handle | cell array | character vector

Push button callback that executes when you click the button, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

If you specify this property using a function handle, then MATLAB passes the source `PushButton` object and an event data structure as the first and second input arguments to the function. This table describes the fields in the event data structure.

Field	Description
`Source`	`PushButton` object
`Axes`	Associated `Axes` object
`EventName`	`'ButtonPushed'`

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you

do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent
`AxesToolbar` object

Parent container, specified as an `AxesToolbar` object.

`Children` — Children
empty `GraphicsPlaceholder` array

This object has no children. You cannot set this property.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object
`'toolbarpushbutton'` (default)

This property is read-only.

Type of graphics object, returned as `'toolbarpushbutton'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[ ]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

Unused Properties

`UIContextMenu` — Context menu

`uicontextmenu` object

Context menu, specified as a `ContextMenu` object.

Note The button does not use this property.

`ButtonDownFcn` — Mouse-click callback

`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as a function handle, a cell array, or a character vector.

Note The button does not use this property.

See Also

`axtoolbarbtn`

Topics

“Access Property Values”
“Graphics Object Properties”

Introduced in R2018b

toolboxdir

Root folder for specified toolbox

Syntax

```
s = toolboxdir(toolboxName)
```

Description

`s = toolboxdir(toolboxName)` returns a character vector containing the absolute path to the root folder for the specified toolbox.

It is important to use the `toolboxdir` function when writing code using MATLAB Compiler. A toolbox can have a different path depending on whether it is running from MATLAB or from an application deployed with the MATLAB Compiler. `toolboxdir` ensures that the correct path is used.

Examples

Get Toolbox Path

Suppose that you have MATLAB R2017b installed with the Control System Toolbox. Get the path for the Control System Toolbox.

```
s = toolboxdir('control')
```

```
s  = C:\Program Files\MATLAB\R2017b\toolbox\control
```

Input Arguments

`toolboxName` — Toolbox name
character vector | string scalar

Toolbox name, specified as a character vector or string scalar.

Data Types: `char` | `string`

Tips

- To determine the toolbox name to use for a given toolbox, run this code, substituting *`toolbxfcn`* with the name of a function in the product.

  ```
  n = 'toolbxfcn';
  pat = '(?<=[\\/]toolbox[\\/])[^\\/]+';
  regexp(which(n), pat, 'match', 'once')
  ```

  For example, to determine the toolbox name for Control System Toolbox, set `n` to the name of a function unique to Control System Toolbox, such as `dss`.

  ```
  n = 'dss'
  pat = '(?<=[\\/]toolbox[\\/])[^\\/]+'
  regexp(which(n), pat, 'match', 'once')
  ```

  ```
  control
  ```

See Also

`fullfile` | `matlabroot` | `path`

Introduced in R2006a

topkrows

Top rows in sorted order

Syntax

```
B = topkrows(X,k)
B = topkrows(X,k,col)
B = topkrows( ___ ,direction)
B = topkrows( ___ ,'ComparisonMethod',method)
[B,I] = topkrows(X, ___ )

B = topkrows(T,k)
B = topkrows(T,k,vars)
B = topkrows(T,k,vars,direction)
B = topkrows( ___ ,'ComparisonMethod',method)
[B,I] = topkrows(T, ___ )
```

Description

`B = topkrows(X,k)` returns the first `k` rows in array `X` sorted in descending order (for numeric data) or reverse alphabetical order (for text data). `topkrows` sorts based on the elements in the first column. When the first column contains elements of equal value, `topkrows` sorts according to the elements in the next column and repeats this behavior for succeeding equal values.

`B = topkrows(X,k,col)` sorts the results by the columns specified by `col`. Use this syntax to perform multiple column sorts in succession. For example, `topkrows(X,k,5)` sorts the rows of `X` in descending order based on the elements in the fifth column. `topkrows(X,k,[4 6])` first sorts the rows in descending order by the elements in the fourth column, and then it sorts based on the elements in the sixth column to break ties.

`B = topkrows( ___ ,direction)` specifies the direction of the sorting using any of the previous syntaxes. `direction` can be `'ascend'`, `'descend'`, or a cell array containing those values.

For example, `topkrows(A,2,[2 3],{'ascend' 'descend'})` gets the top 2 rows by first sorting rows in ascending order by the elements in column 2. Then, it sorts the rows with equal entries in column 2 in descending order by the elements in column 3.

`B = topkrows( ___ ,'ComparisonMethod',method)` specifies how to compare complex numbers. The comparison method can be `'auto'`, `'real'`, or `'abs'`.

`[B,I] = topkrows(X, ___ )` also returns an index vector `I` that describes the order of the selected rows such that `B = X(I,:)`.

`B = topkrows(T,k)` returns the first `k` rows in table or timetable `T`, in sorted order. Table rows are in descending sorted order by all of their variables, and timetable rows are in descending sorted order by time.

`B = topkrows(T,k,vars)` sorts the results by the variables specified by `vars`. Use this syntax to sort with multiple variables in succession. For example, `topkrows(T,k, {'Var1','Var2'})` first sorts the rows of `T` based on the elements in `Var1`, and then it sorts by the elements in `Var2`.

`B = topkrows(T,k,vars,direction)` specifies the direction of the sorting. For example, use `'ascend'` to sort `T` in ascending order.

`B = topkrows( ___ ,'ComparisonMethod',method)` specifies how to compare complex numbers. The comparison method can be `'auto'`, `'real'`, or `'abs'`.

`[B,I] = topkrows(T, ___ )` also returns an index vector `I` that describes the order of the selected rows such that `B = T(I,:)`.

Examples

Sort Rows of Matrix

Sort the rows of a matrix using different sorting orders and view the top rows.

Create a 20-by-5 matrix of random integers between 1 and 10.

```
rng default % for reproducibility
X = randi(10,20,5);
```

Sort the rows of X in descending order and return the top 4 rows. By default, `topkrows` sorts using the first column of the matrix. For any rows that have equal elements in a particular column, the sorting is based on the column immediately to the right.

```
TA = topkrows(X,4)
```

```
TA = 4×5

    10    10     8     7     6
    10     7     8     2     4
    10     4     4     3     5
    10     3     7     9     6
```

When called with three input arguments, `topkrows` bases the sort entirely on the column specified in the third argument. This means that rows with equal values in the specified column remain in their original order. Sort X in descending order using the values in the third column and return the top 5 rows.

```
TB = topkrows(X,5,3)
```

```
TB = 5×5

     5     7    10     2     6
     2     9     8     6     6
    10    10     8     7     6
    10     7     8     2     4
    10     2     8     3     6
```

Sort X using both the third and fourth columns. In this case, `topkrows` sorts the rows by column 3. Then, for any rows with equal values in column 3, it sorts by column 4.

```
TC = topkrows(X,5,[3 4])
```

```
TC = 5×5

     5     7    10     2     6
    10    10     8     7     6
     2     9     8     6     6
    10     2     8     3     6
    10     7     8     2     4
```

Specify Sorting Directions for Matrix Columns

Sort a matrix using several columns with different sorting directions.

Create a 100-by-5 matrix of random integers between 1 and 10.

```
rng default % for reproducibility
X = randi(10,100,5);
```

Sort X using the first three columns and return the top 10 rows. Specify a sorting direction for each column using a cell array.

```
TA = topkrows(X,10,1:3,{'descend','ascend','ascend'})
```

```
TA = 10×5

    10     1     4     6     7
    10     1     8     5     1
    10     2     3     4     7
    10     3     5    10     5
    10     4     7     2     4
    10     5     5     2     7
    10     5     5     6     7
    10     6     5     5     7
    10     6     6     1     5
    10     7     7     8     1
```

Sort Rows of Table

Sort rows of heterogeneous data in a table.

Create a table from the `patients.mat` data set, which includes basic health information for a group of patients. Include the patients age, gender, height, and their self-assessed health status in the table. Make the `SelfAssessedHealthStatus` variable an ordinal categorical array.

```
load patients
vals = {'Poor','Fair','Good','Excellent'};
SelfAssessedHealthStatus = categorical(SelfAssessedHealthStatus,vals,'Ordinal',true);
T = table(Age,Gender,Height,SelfAssessedHealthStatus);
```

Find the top 10 rows when the table is sorted in descending order. The result is sorted by the first variable, `Age`, in descending order. The remaining columns are subsorted to break ties:

- The `Gender` variable is subsorted to break ties with age.
- The `Height` variable breaks ties with gender.
- The `SelfAssessedHealthStatus` variable breaks ties with height.

```
TA = topkrows(T,10)
```

```
TA=10×4 table
    Age      Gender      Height    SelfAssessedHealthStatus
    ___    __________    ______    ________________________

    50     {'Male'  }      72             Excellent
    50     {'Male'  }      68             Good
    49     {'Male'  }      70             Fair
    49     {'Male'  }      68             Poor
    49     {'Female'}      64             Good
    49     {'Female'}      63             Good
    48     {'Male'  }      71             Good
    48     {'Male'  }      71             Good
    48     {'Male'  }      66             Fair
    48     {'Female'}      66             Excellent
```

Find the top 10 rows containing the youngest women by sorting on the `Gender` variable and subsorting on the `Age` variable.

```
TB = topkrows(T,10,{'Gender','Age'},'ascend')
```

```
TB=10×4 table
    Age      Gender      Height    SelfAssessedHealthStatus
    ___    __________    ______    ________________________

    25     {'Female'}      63             Good
    25     {'Female'}      64             Excellent
    27     {'Female'}      69             Fair
    28     {'Female'}      65             Good
    28     {'Female'}      65             Good
    28     {'Female'}      66             Good
    29     {'Female'}      63             Excellent
    29     {'Female'}      68             Excellent
    29     {'Female'}      64             Good
```

```
    30     {'Female'}      67             Excellent
```

Find the top 10 oldest women by changing the sorting direction of the `Age` variable to `'descend'`.

```
TB = topkrows(T,10,{'Gender','Age'},{'ascend','descend'})
```

```
TB=10×4 table
    Age      Gender      Height    SelfAssessedHealthStatus
    ___    __________    ______    ________________________

    49     {'Female'}      64             Good
    49     {'Female'}      63             Good
    48     {'Female'}      65             Excellent
    48     {'Female'}      66             Excellent
    48     {'Female'}      64             Excellent
    48     {'Female'}      64             Good
    48     {'Female'}      66             Excellent
    47     {'Female'}      66             Excellent
    46     {'Female'}      68             Good
    45     {'Female'}      68             Excellent
```

Sort Complex Numbers

Sort a matrix of complex numbers by absolute value and then by real part.

Create a 100-by-2 matrix of random complex numbers.

```
valRange = [-10 10];
X = randi(valRange,100,2) + 1i*randi(valRange,100,2);
```

Find the top 10 rows of the matrix. By default, `topkrows` compares the complex numbers by absolute value.

```
TA = topkrows(X,10)
```

```
TA = 10×2 complex

 -10.0000 + 9.0000i  10.0000 - 2.0000i
  -8.0000 + 9.0000i   2.0000 - 8.0000i
```

```
   9.0000 + 8.0000i   4.0000 + 7.0000i
  -6.0000 +10.0000i  -8.0000 - 7.0000i
   6.0000 -10.0000i  -1.0000 - 5.0000i
   6.0000 -10.0000i   0.0000 + 5.0000i
  -7.0000 + 9.0000i  -2.0000 - 5.0000i
   9.0000 - 7.0000i  10.0000 + 7.0000i
   9.0000 - 7.0000i   6.0000 + 6.0000i
  -9.0000 - 7.0000i   9.0000 + 9.0000i
```

Find the top 10 rows of the matrix using only the real part of the complex numbers by specifying the `'ComparisonMethod'` name-value pair.

```
TB = topkrows(X,10,'ComparisonMethod','real')
```

```
TB = 10×2 complex

  10.0000 + 4.0000i  -3.0000 - 7.0000i
  10.0000 + 3.0000i   4.0000 + 5.0000i
  10.0000 + 2.0000i   5.0000 - 7.0000i
  10.0000 - 1.0000i  -1.0000 - 8.0000i
  10.0000 - 1.0000i  -6.0000 +10.0000i
  10.0000 - 4.0000i  -9.0000 + 0.0000i
  10.0000 - 5.0000i  -8.0000 - 3.0000i
   9.0000 + 8.0000i   4.0000 + 7.0000i
   9.0000 + 5.0000i -10.0000 + 0.0000i
   9.0000 + 1.0000i   1.0000 - 9.0000i
```

Input Arguments

X — Input array

numeric array | logical array | character array | string array | categorical array | datetime array | duration array

Input array, specified as a numeric, logical, character, string, categorical, datetime, or duration array.

- If `X` is a nonordinal categorical array, then `topkrows` sorts the elements in descending order based on the order of the categories returned by `categories(X)`.
- If `X` contains `NaN`, `NaT`, or other missing values, then `topkrows` places the missing values at the end of a descending sort.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `categorical` | `datetime` | `duration`

`T` — Input table
table | timetable

Input table, specified as a table or timetable.

Data Types: `table` | `timetable`

`k` — Number of rows to return
scalar

Number of rows to return, specified as a nonnegative scalar integer. If `k` is greater than the number of rows in `X`, then `topkrows` returns all of the rows in `X`.

`col` — Columns to sort by
scalar | vector

Columns to sort by, specified as a positive scalar integer or a vector of positive integers.

Example: `B = topkrows(X,100,[1 3])` sorts over the first and third columns before returning the top 100 rows.

`vars` — Variables to sort by
scalar | vector | variable name | cell array | `'RowNames'`

Variables to sort by, specified as one of the options in this table.

Option	Example	Description
positive integer	`topkrows(T,k,3)`	The integer `n` specifies the index of the variable to sort by as returned by `T.Properties.VariableNames{n}`.
vector of positive integers	`topkrows(T,k,[1 3])`	The vector `[n1 n2 …]` specifies the indices of several variables to sort by as returned by `T.Properties.VariableNames{[n1 n2 …]}`.

Option	Example	Description
logical vector	`topkrows(T,k,[true false true])`	Specifies one or more variables to sort by using values of `true` or `false`.
variable name	`topkrows(T,k,'Var3')`	Specifies the sorting variable as one of the variable names listed in `T.Properties.VariableNames`.
cell array	`topkrows(T,k,{'Var1' 'Var3'})`	Specifies several sorting variables selected from `T.Properties.VariableNames`.
`'RowNames'`	`topkrows(T,k,'RowNames')`	**For tables only**. This option sorts the results by the row names.

Example: `B = topkrows(X,k,[1 3])` sorts over the first and third columns.

Example: `B = topkrows(X,k,'Year')` sorts using the `Year` variable.

`direction` — Sorting direction
`'descend'` (default) | `'ascend'` | cell array

Sorting direction, specified as either `'descend'`, `'ascend'`, or a cell array that specifies some combination of these values.

If `direction` is a cell array, then it must contain `'descend'` or `'ascend'` for each sorting column specified by `col` or `vars`. If you do not specify `col` or `vars`, then the cell array must contain `'descend'` or `'ascend'` for each column in `X` or variable in `T`.

`method` — Comparison method for complex numbers
`'auto'` (default) | `'real'` | `'abs'`

Comparison method for complex numbers, specified as one of these values:

- `'auto'` — (default) Compares real numbers according to `'real'` and complex numbers according to `'abs'`.
- `'real'` — Compares numbers by real part `real(A)`. Numbers with equal real part are subsorted by imaginary part `imag(A)`.

- `'abs'` — Compares numbers by absolute value `abs(A)`. Numbers with equal magnitude are subsorted by phase angle `angle(A)`.

This option does not support nonnumeric input data (`datetime`, `duration`, `string`, and so on).

Output Arguments

`B` — Requested rows
array

Requested rows, returned as an array of the same class as `X` or `T`.

`I` — Row indices
vector

Row indices, returned as a vector. `I` describes the order of the selected rows such that `B = X(I,:)` or `B = T(I,:)`.

Tips

- `topkrows` does not do a full sort of the input data, so it is generally faster than `sort` and `sortrows` when the number of requested rows is small.

Compatibility Considerations

Behavior change with tall arrays

Behavior changed in R2017b

Some behaviors of `topkrows` operating on tall arrays have changed:

- `topkrows` places `NaN`, `NaT`, and other missing values at the end of a descending sort. In previous releases `topkrows` placed missing values at the beginning of a descending sort.
- `topkrows` no longer accepts tall cell arrays containing only scalar numeric values as inputs. Use `cell2mat` to convert the tall cell array of scalar numeric values into a tall matrix before using `topkrows`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- The `'ComparisonMethod'` name-value pair is not supported.
- The `'RowNames'` option for tables is not supported.

For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`head` | `maxk` | `mink` | `sort` | `sortrows` | `tail`

Introduced in R2016b

trace

Sum of diagonal elements

Syntax

```
b = trace(A)
```

Description

`b = trace(A)` calculates the sum of the diagonal elements of matrix `A`:

$$\mathrm{tr}(A) = \sum_{i=1}^{n} a_{ii} = a_{11} + a_{22} + \ldots + a_{nn}\,.$$

Examples

Sum of Matrix Diagonal

Create a 3-by-3 matrix and calculate the sum of the diagonal elements.

```
A = [1 -5 2;
    -3  7 9;
     4 -1 6];
```

```
b = trace(A)
```

```
b = 14
```

The result $\mathrm{tr}(A) = 14$ agrees with a manual calculation.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & -5 & 2 \\ -3 & 7 & 9 \\ 4 & -1 & 6 \end{bmatrix},$$

$$\mathrm{tr}(A) = \sum_{i=1}^{3} a_{\mathrm{ii}} = a_{11} + a_{22} + a_{33} = 1 + 7 + 6 = 14\,.$$

Matrix Trace Properties

Verify several properties of the trace of a matrix (up to round-off error).

Create two matrices. Verify that $\mathrm{tr}(A + B) = \mathrm{tr}(A) + \mathrm{tr}(B)$.

```
A = magic(3);
B = rand(3);
trace(A+B)
```

```
ans = 17.4046
```

```
trace(A) + trace(B)
```

```
ans = 17.4046
```

Verify that $\mathrm{tr}(A) = \mathrm{tr}\left(A^{T)}\right)$.

```
trace(A)
```

```
ans = 15
```

```
trace(A')
```

```
ans = 15
```

Verify that $\mathrm{tr}\left(A^T B\right) = \mathrm{tr}\left(\mathrm{AB}^T\right)$.

```
trace(A'*B)
```

```
ans = 22.1103
```

```
trace(A*B')
```

```
ans = 22.1103
```

Verify that $\mathrm{tr}(\mathrm{cA}) = c\,\mathrm{tr}(A)$ for a scalar c.

```
c = 5;
trace(c*A)
```

```
ans = 75
```

```
c*trace(A)
```

```
ans = 75
```

Verify that the trace equals the sum of the eigenvalues $\mathrm{tr}(A) = \sum_i \lambda_i$.

```
trace(A)
```

```
ans = 15
```

```
sum(eig(A))
```

```
ans = 15.0000
```

Input Arguments

A — Input matrix
square matrix

Input matrix, specified as a square matrix. A can be full or sparse.

Data Types: `single` | `double`
Complex Number Support: Yes

Algorithms

`trace` extracts the diagonal elements and adds them together with the command `sum(diag(A))`. The value of the trace is the same (up to round-off error) as the sum of the matrix eigenvalues `sum(eig(A))`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`det` | `eig` | `sum`

Introduced before R2006a

Transform Properties

Transform object appearance and behavior

Description

`Transform` properties control the behavior of `Transform` objects. By changing property values, you can modify certain aspects of the `Transform` object.

Starting in R2014b, you can use dot notation to query and set properties.

```
h = hgtransform;
c = h.Children;
h.Matrix = makehgtform('scale',0.5);
```

If you are using an earlier release, use the `get` and `set` functions instead.

Properties

Transform Matrix

`Matrix` — Transform matrix

4-by-4 matrix

Transform matrix applied to the transform object and its children, specified as a 4-by-4 matrix. For more information about defining this matrix, see “Transforms Supported by hgtransform”.

Data Types: `double`

Legend

`DisplayName` — Legend label

`''` (default) | character vector | string scalar

Legend label, specified as a character vector or string scalar. The legend does not display until you call the `legend` command. If you do not specify the text, then `legend` sets the label using the form `'dataN'`.

`Annotation` — Control for including or excluding object from legend

`Annotation` object

This property is read-only.

Control for including or excluding the object from a legend, returned as an `Annotation` object. Set the underlying `IconDisplayStyle` property to one of these values:

- `'on'` — Include the object in the legend.
- `'off'` — Do not include the object in the legend. (default)
- `'children'` — Include only children of the object as separate items in the legend.

Interactivity

`Visible` — Visibility

`'on'` (default) | `'off'`

Visibility, specified as one of these values:

- `'on'` — Display all objects in the `Transform` object.
- `'off'` — Hide all objects in the `Transform` object. You still can access the properties of invisible `Transform` objects. Setting the `Visible` property for the `Transform` object does not change the `Visible` property for objects in the `Transform` object.

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | `ContextMenu` object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when you right-click the object. Create the context menu using the `uicontextmenu` function.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then the context menu does not appear.

`Selected` — Selection state

`'off'` (default) | `'on'`

Selection state, specified as one of these values:

- `'on'` — Selected. If you click the object when in plot edit mode, then MATLAB sets its `Selected` property to `'on'`. If the `SelectionHighlight` property also is set to `'on'`, then MATLAB displays selection handles around the object.
- `'off'` — Not selected.

`SelectionHighlight` — Display of selection handles
`'on'` (default) | `'off'`

Display of selection handles when selected, specified as one of these values:

- `'on'` — Display selection handles when the `Selected` property is set to `'on'`.
- `'off'` — Never display selection handles, even when the `Selected` property is set to `'on'`.

Callbacks

`ButtonDownFcn` — Mouse-click callback
`''` (default) | function handle | cell array | character vector

Mouse-click callback, specified as one of these values:

- Function handle
- Cell array containing a function handle and additional arguments
- Character vector that is a valid MATLAB command or function, which is evaluated in the base workspace (not recommended)

Use this property to execute code when you click the object. If you specify this property using a function handle, then MATLAB passes two arguments to the callback function when executing the callback:

- Clicked object — Access properties of the clicked object from within the callback function.
- Event data — Empty argument. Replace it with the tilde character (~) in the function definition to indicate that this argument is not used.

For more information on how to use function handles to define callback functions, see “Callback Definition”.

Note If the `PickableParts` property is set to `'none'` or if the `HitTest` property is set to `'off'`, then this callback does not execute.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Callback Definition”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object.

If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`PickableParts` — Children that can capture mouse clicks
`'visible'` (default) | `'none'`

Children that can capture mouse clicks, specified as one of these values:

- `'visible'` — Any child object can capture a mouse click, depending on the `PickableParts` property value of the child.

- `'none'` — No child objects can capture mouse clicks, regardless of the `PickableParts` property value of the child.

`HitTest` — Response to mouse clicks captured by children
`'on'` (default) | `'off'`

Response to mouse clicks captured by children, specified one of these values:

- `'on'` — Trigger the `ButtonDownFcn` callback of the `Transform` object. If you have defined the `UIContextMenu` property, then invoke the context menu.
- `'off'` — Do not trigger the callbacks of the `Transform` object. Instead, trigger the callbacks for the nearest ancestor that has a `HitTest` property set to `'on'` and a `PickableParts` property value that enables the ancestor to capture mouse clicks.

A `Transform` object cannot capture mouse clicks. However, if you click a child of the `Transform` object and if the child has a `HitTest` property set to `'off'`, then the child passes the click to the `Transform` object.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent
`Axes` object | `Group` object | `Transform` object

Parent, specified as an `Axes`, `Group`, or `Transform` object.

`Children` — Children
empty `GraphicsPlaceholder` array | array of graphics objects

Children, returned as an array of graphics objects. Use this property to view a list of the children or to reorder the children by setting the property to a permutation of itself.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the child graphics object to the `Transform` object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object

`'hgtransform'`

This property is read-only.

Type of graphics object, returned as `'hgtransform'`. Use this property to find all objects of a given type within a plotting hierarchy, for example, searching for the type using `findobj`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object

elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[ ]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`hgtransform`

Topics

“Access Property Values”
“Graphics Object Properties”

Introduced before R2006a

transpose, .'

Transpose vector or matrix

Syntax

```
B = A.'
B = transpose(A)
```

Description

`B = A.'` returns the nonconjugate transpose of `A`, that is, interchanges the row and column index for each element. If `A` contains complex elements, then `A.'` does not affect the sign of the imaginary parts. For example, if `A(3,2)` is `1+2i` and `B = A.'`, then the element `B(2,3)` is also `1+2i`.

`B = transpose(A)` is an alternate way to execute `A.'` and enables operator overloading for classes.

Examples

Real Matrix

Create a matrix of real numbers and compute its transpose. `B` has the same elements as `A`, but the rows of `B` are the columns of `A` and the columns of `B` are the rows of `A`.

```
A = magic(4)
```

```
A = 4×4

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

```
B = A.'
```

```
B = 4×4

    16     5     9     4
     2    11     7    14
     3    10     6    15
    13     8    12     1
```

Complex Matrix

Create a matrix containing complex elements and compute its nonconjugate transpose. B contains the same elements as A, except the rows and columns are interchanged. The signs of the imaginary parts are unchanged.

```
A = [1 3 4-1i 2+2i; 0+1i 1-1i 5 6-1i]
```

```
A = 2×4 complex

   1.0000 + 0.0000i   3.0000 + 0.0000i   4.0000 - 1.0000i   2.0000 + 2.0000i
   0.0000 + 1.0000i   1.0000 - 1.0000i   5.0000 + 0.0000i   6.0000 - 1.0000i
```

```
B = A.'
```

```
B = 4×2 complex

   1.0000 + 0.0000i   0.0000 + 1.0000i
   3.0000 + 0.0000i   1.0000 - 1.0000i
   4.0000 - 1.0000i   5.0000 + 0.0000i
   2.0000 + 2.0000i   6.0000 - 1.0000i
```

Input Arguments

A — Input array
vector | matrix

Input array, specified as a vector or matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `cell` | `categorical` | `datetime` | `duration` | `calendarDuration`
Complex Number Support: Yes

Tips

- The complex conjugate transpose operator, `A'`, also negates the sign of the imaginary part of the complex elements in `A`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`conj` | `ctranspose` | `permute`

Topics

"Array vs. Matrix Operations"
"Operator Precedence"
"MATLAB Operators and Special Characters"

Introduced before R2006a

trapz

Trapezoidal numerical integration

Syntax

```
Q = trapz(Y)
Q = trapz(X,Y)
Q = trapz( ___ ,dim)
```

Description

`Q = trapz(Y)` computes the approximate integral of `Y` via the trapezoidal method on page 1-15908 with unit spacing. The size of `Y` determines the dimension to integrate along:

- If `Y` is a vector, then `trapz(Y)` is the approximate integral of `Y`.
- If `Y` is a matrix, then `trapz(Y)` integrates over each column and returns a row vector of integration values.
- If `Y` is a multidimensional array, then `trapz(Y)` integrates over the first dimension whose size does not equal 1. The size of this dimension becomes 1, and the sizes of other dimensions remain unchanged.

`Q = trapz(X,Y)` integrates `Y` with respect to the coordinates or scalar spacing specified by `X`.

- If `X` is a vector of coordinates, then `length(X)` must be equal to the size of the first dimension of `Y` whose size does not equal 1.
- If `X` is a scalar spacing, then `trapz(X,Y)` is equivalent to `X*trapz(Y)`.

`Q = trapz( ___ ,dim)` integrates along the dimension `dim` using any of the previous syntaxes. You must specify `Y`, and optionally can specify `X`. If you specify `X`, then it can be a scalar or a vector with length equal to `size(Y,dim)`. For example, if `Y` is a matrix, then `trapz(X,Y,2)` integrates each row of `Y`.

Examples

Integrate Vector of Data with Unit Spacing

Calculate the integral of a vector where the spacing between data points is 1.

Create a numeric vector of data.

```
Y = [1 4 9 16 25];
```

`Y` contains function values for $f(x) = x^2$ in the domain [1, 5].

Use `trapz` to integrate the data with unit spacing.

```
Q = trapz(Y)
```

```
Q = 42
```

This approximate integration yields a value of `42`. In this case, the exact answer is a little less, $41\frac{1}{3}$. The `trapz` function overestimates the value of the integral because *f(x)* is concave up.

Integrate Vector of Data with Nonunit Spacing

Calculate the integral of a vector where the spacing between data points is uniform, but not equal to 1.

Create a domain vector.

```
X = 0:pi/100:pi;
```

Calculate the sine of `X`.

```
Y = sin(X);
```

Integrate `Y` using `trapz`.

```
Q = trapz(X,Y)
```

```
Q = 1.9998
```

When the spacing between points is constant, but not equal to 1, an alternative to creating a vector for X is to specify the scalar spacing value. In that case, `trapz(pi/100,Y)` is the same as `pi/100*trapz(Y)`.

Integrate Matrix with Nonuniform Spacing

Integrate the rows of a matrix where the data has a nonuniform spacing.

Create a vector of *x*-coordinates and a matrix of observations that take place at the irregular intervals. The rows of Y represent velocity data, taken at the times contained in X, for three different trials.

```
X = [1 2.5 7 10];
Y = [5.2   7.7   9.6   13.2;
     4.8   7.0  10.5   14.5;
     4.9   6.5  10.2   13.8];
```

Use `trapz` to integrate each row independently and find the total distance traveled in each trial. Since the data is not evaluated at constant intervals, specify X to indicate the spacing between the data points. Specify `dim = 2` since the data is in the rows of Y.

```
Q1 = trapz(X,Y,2)
```

```
Q1 = 3×1

   82.8000
   85.7250
   82.1250
```

The result is a column vector of integration values, one for each row in Y.

Multiple Numerical Integrations

Create a grid of domain values.

```
x = -3:.1:3;
y = -5:.1:5;
[X,Y] = meshgrid(x,y);
```

Calculate the function $f(x, y) = x^2 + y^2$ on the grid.

```
F = X.^2 + Y.^2;
```

`trapz` integrates numeric data rather than functional expressions, so in general the expression does not need to be known to use `trapz` on a matrix of data. In cases where the functional expression is known, you can instead use `integral`, `integral2`, or `integral3`.

Use `trapz` to approximate the double integral

$$I = \int_{-5}^{5}\int_{-3}^{3}\left(x^2 + y^2\right)dx\,dy$$

To perform double or triple integrations on an array of numeric data, nest function calls to `trapz`.

```
I = trapz(y,trapz(x,F,2))
```

```
I = 680.2000
```

`trapz` performs the integration over *x* first, producing a column vector. Then, the integration over *y* reduces the column vector to a single scalar. `trapz` slightly overestimates the exact answer of 680 because *f(x,y)* is concave up.

Input Arguments

Y — Numeric data
vector | matrix | multidimensional array

Numeric data, specified as a vector, matrix, or multidimensional array. By default, `trapz` integrates along the first dimension of `Y` whose size does not equal 1.

Data Types: `single` | `double`
Complex Number Support: Yes

X — Point spacing
1 (default) | uniform scalar spacing | vector of coordinates

Point spacing, specified as `1` (default), a uniform scalar spacing, or a vector of coordinates.

- If `X` is a scalar, then it specifies a uniform spacing between the data points and `trapz(X,Y)` is equivalent to `X*trapz(Y)`.
- If `X` is a vector, then it specifies *x*-coordinates for the data points and `length(X)` must be the same as the size of the integration dimension in `Y`.

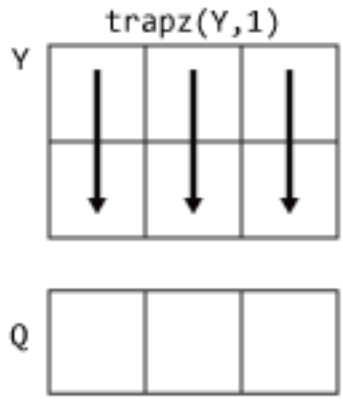
Data Types: `single` | `double`

`dim` — Dimension to operate along
positive integer scalar

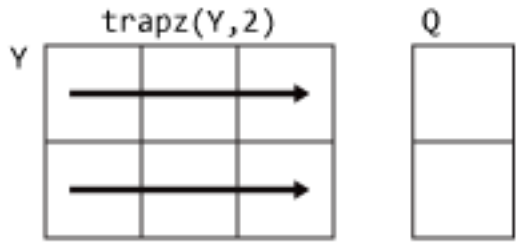
Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Consider a two-dimensional input array, `Y`:

- `trapz(Y,1)` works on successive elements in the columns of `Y` and returns a row vector.



- `trapz(Y,2)` works on successive elements in the rows of `Y` and returns a column vector.

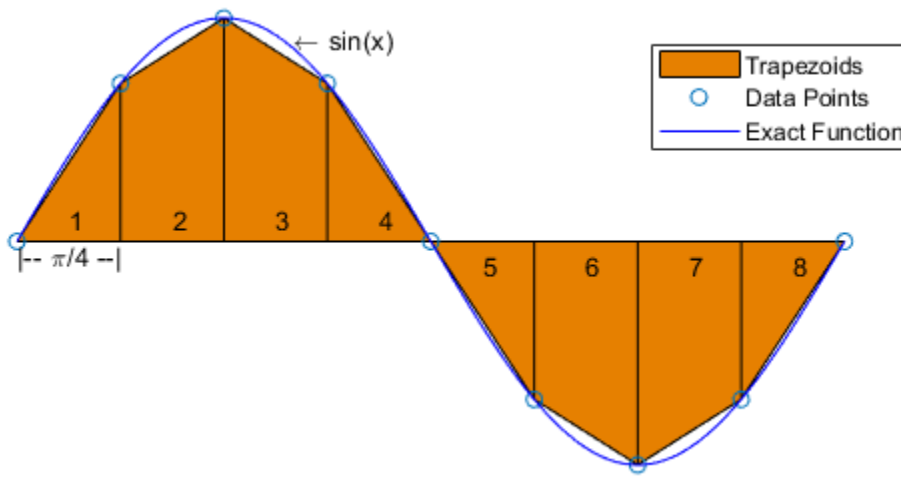


If `dim` is greater than `ndims(Y)`, then `trapz` returns an array of zeros of the same size as `Y`.

More About

Trapezoidal Method

`trapz` performs numerical integration via the trapezoidal method. This method approximates the integration over an interval by breaking the area down into trapezoids with more easily computable areas. For example, here is a trapezoidal integration of the sine function using eight evenly-spaced trapezoids:



For an integration with `N+1` evenly spaced points, the approximation is

$$\int_a^b f(x)dx \approx \frac{b-a}{2N}\sum_{n=1}^{N}(f(x_n)+f(x_{n+1}))$$
$$= \frac{b-a}{2N}[f(x_1)+2f(x_2)+\ldots+2f(x_N)+f(x_{N+1})] ,$$

where the spacing between each point is equal to the scalar value $\frac{b-a}{N}$. By default MATLAB uses a spacing of 1.

If the spacing between the `N+1` points is not constant, then the formula generalizes to

$$\int_a^b f(x)dx \approx \frac{1}{2}\sum_{n=1}^{N}(x_{n+1}-x_n)[f(x_n)+f(x_{n+1})] ,$$

where $a = x_1 < x_2 < \ldots < x_N < x_{N+1} = b$, and $(x_{n+1} - x_n)$ is the spacing between each consecutive pair of points.

Tips

- Use `trapz` and `cumtrapz` to perform numerical integrations on discrete data sets. Use `integral`, `integral2`, or `integral3` instead if a functional expression for the data is available.
- `trapz` reduces the size of the dimension it operates on to 1, and returns only the final integration value. `cumtrapz` also returns the intermediate integration values, preserving the size of the dimension it operates on.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If you supply `dim`, then it must be a constant.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cumsum` | `cumtrapz` | `integral` | `integral2` | `integral3`

Topics

“Integration of Numeric Data”

Introduced before R2006a

treelayout

Lay out tree or forest

Syntax

```
[x,y] = treelayout(parent,post)
[x,y,h,s] = treelayout(parent,post)
```

Description

`[x,y] = treelayout(parent,post)` lays out a tree or a forest. `parent` is the vector of parent pointers, with `0` for a root. `post` is an optional postorder permutation on the tree nodes. If you omit `post`, `treelayout` computes it. `x` and `y` are vectors of coordinates in the unit square at which to lay out the nodes of the tree to make a nice picture.

`[x,y,h,s] = treelayout(parent,post)` also returns the height of the tree `h` and the number of vertices `s` in the top-level separator.

See Also

`etree` | `etreeplot` | `symbfact` | `treeplot`

Introduced before R2006a

treeplot

Plot picture of tree

Syntax

```
treeplot(p)
treeplot(p,nodeSpec,edgeSpec)
```

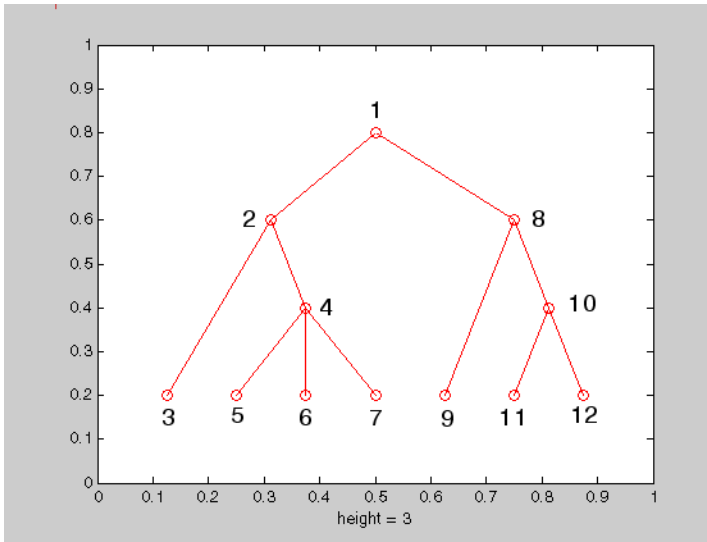
Description

`treeplot(p)` plots a picture of a tree given a vector of parent pointers, with `p(i) = 0` for a root.

`treeplot(p,nodeSpec,edgeSpec)` allows optional parameters `nodeSpec` and `edgeSpec` to set the node or edge color, marker, and linestyle. Use `''` to omit one or both.

Examples

To plot a tree with 12 nodes, call `treeplot` with a 12-element input vector. The index of each element in the vector is shown adjacent to each node in the figure below. (These indices are shown only for the point of illustrating the example; they are not part of the `treeplot` output.)



To generate this plot, set the value of each element in the `nodes` vector to the index of its parent, (setting the parent of the root node to zero).

The node marked `1` in the figure is represented by `nodes(1)` in the input vector, and because this is the root node which has a parent of zero, you set its value to zero:

```
nodes(1) = 0;      % Root node
```

`nodes(2)` and `nodes(8)` are children of `nodes(1)`, so set these elements of the input vector to `1`:

```
nodes(2) = 1;      nodes(8) = 1;
```

`nodes(5:7)` are children of `nodes(4)`, so set these elements to `4`:

```
nodes(5) = 4;      nodes(6) = 4;      nodes(7) = 4;
```

Continue in this manner until each element of the vector identifies its parent. For the plot shown above, the `nodes` vector now looks like this:

```
nodes = [0 1 2 2 4 4 4 1 8 8 10 10];
```

Now call `treeplot` to generate the plot:

```
treeplot(nodes)
```

See Also

`etree` | `etreeplot` | `treelayout`

Introduced before R2006a

tril

Lower triangular part of matrix

Syntax

```
L = tril(A)
L = tril(A,k)
```

Description

`L = tril(A)` returns the lower triangular on page 1-15917 portion of matrix `A`.

`L = tril(A,k)` returns the elements on and below the kth diagonal of `A`.

Examples

Extract Lower Triangular Portions of Matrix

Create a 4-by-4 matrix of ones. Extract the lower triangular portion.

```
A = ones(4)
```

```
A = 4×4

     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1
```

```
B = tril(A)
```

```
B = 4×4

     1     0     0     0
```

```
     1     1     0     0
     1     1     1     0
     1     1     1     1
```

Extract only the elements below the main diagonal.

```
C = tril(A,-1)
```

```
C = 4×4

     0     0     0     0
     1     0     0     0
     1     1     0     0
     1     1     1     0
```

Input Arguments

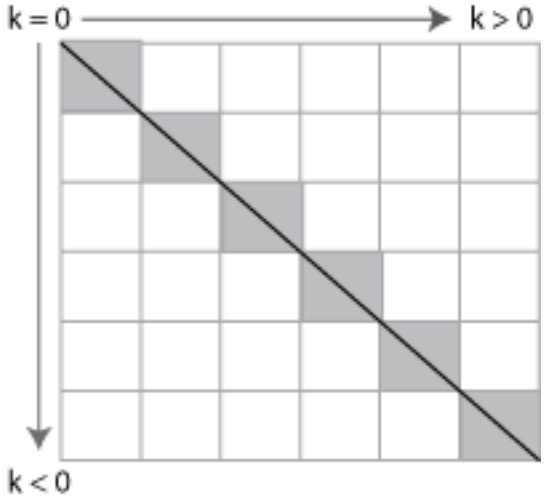
A — Input matrix
matrix

Input matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

k — Diagonals to include
`0` (default) | scalar

Diagonals to include, specified as a scalar. `k = 0` is the main diagonal, `k > 0` is above the main diagonal, and `k < 0` is below the main diagonal.

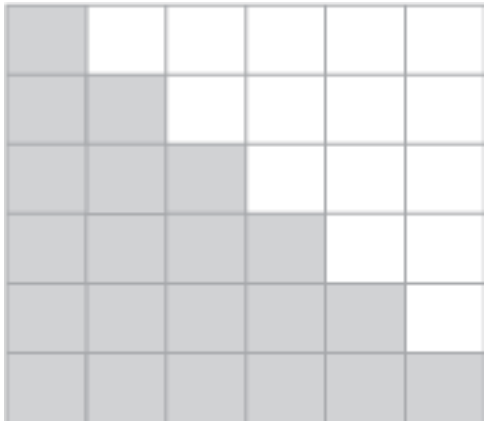


Example: `tril(A,3)`

More About

Lower Triangular

The lower triangular portion of a matrix includes the main diagonal and all elements below it. The shaded blocks in this graphic depict the lower triangular portion of a 6-by-6 matrix.



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If you supply the argument that represents the order of the diagonal matrix, then it must be a real and scalar integer value.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- If you supply the argument that represents the order of the diagonal matrix, then it must be a real and scalar integer value.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bandwidth` | `diag` | `istril` | `triu`

Introduced before R2006a

trimesh

Triangular mesh plot

Syntax

```
trimesh(T,x,y)
trimesh(T,x,y,z)
trimesh(x,y,z,c)
trimesh(TO)
trimesh( ___ ,Name,Value)
h = trimesh( ___ )
```

Description

`trimesh(T,x,y)` plots the 2-D triangular mesh defined by the points in vectors `x` and `y` and a triangle connectivity matrix `T`.

`trimesh(T,x,y,z)` plots a 3-D triangular mesh.

`trimesh(x,y,z,c)` also specifies the mesh edge color for a 3-D triangular mesh.

`trimesh(TO)` plots the mesh defined by a 2-D or 3-D `triangulation` or `delaunayTriangulation` object.

`trimesh( ___ ,Name,Value)` specifies one or more properties of the mesh plot using name-value pairs. For example, `'LineWidth',2` sets the edge width to 2 points.

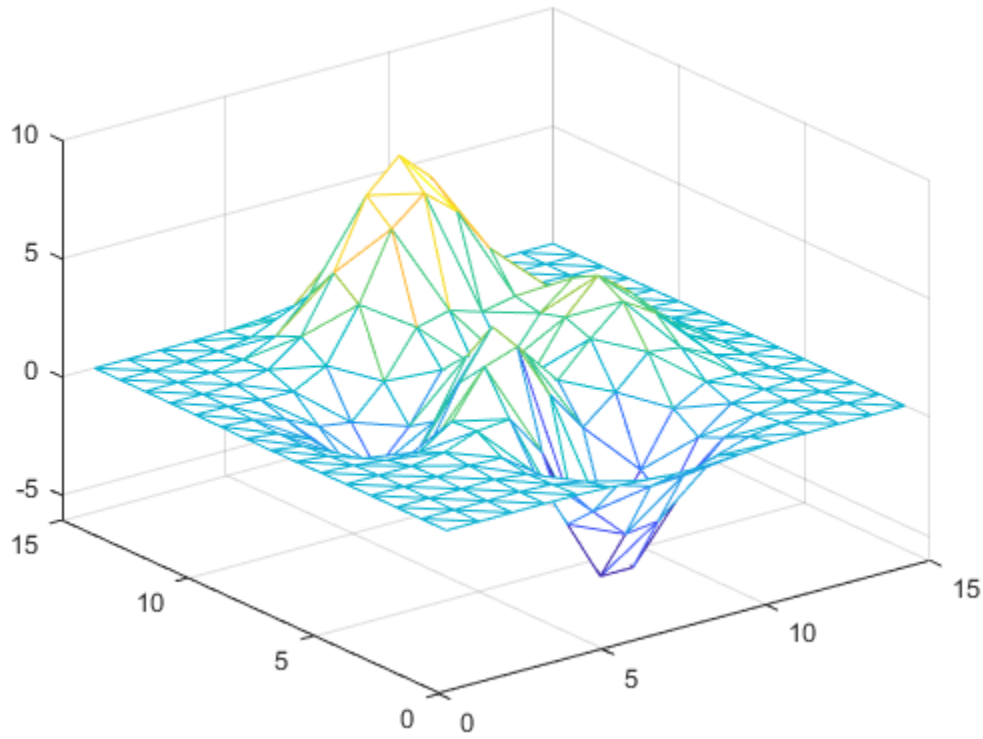
`h = trimesh( ___ )` returns a `patch` object used to create the mesh plot. Use `h` to query and modify properties of the plot. For more information, see Patch Properties.

Examples

3-D Triangular Mesh

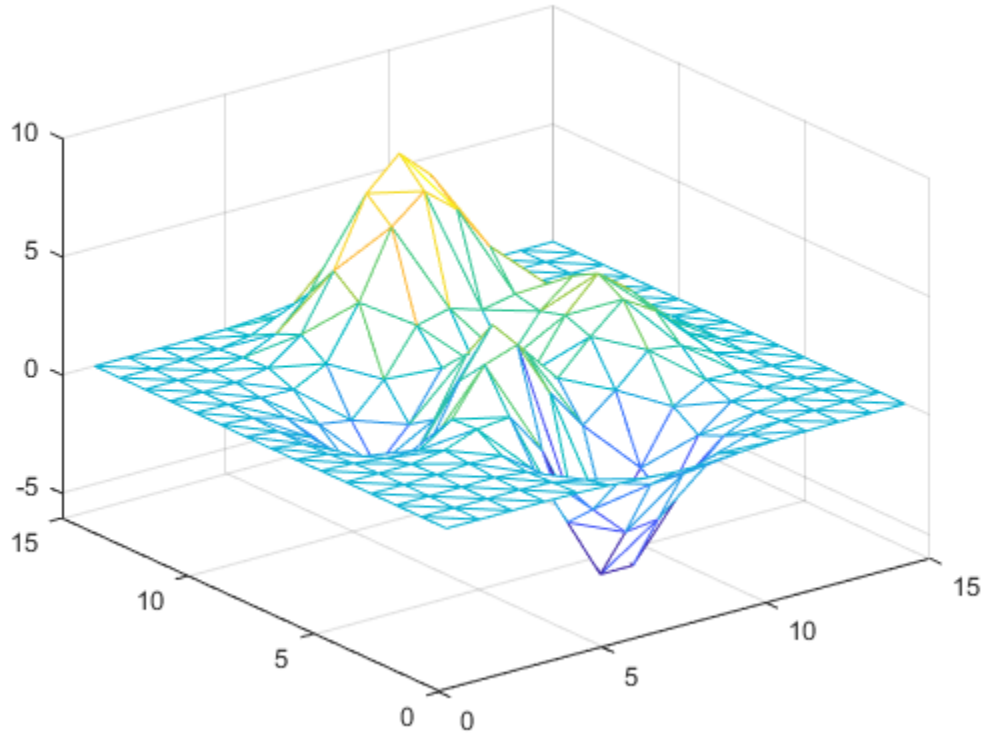
Create a set of 3-D points. Compute a Delaunay triangulation connectivity matrix using the `delaunay` function, and plot the mesh.

```
[x,y] = meshgrid(1:15,1:15);
z = peaks(15);
T = delaunay(x,y);
trimesh(T,x,y,z)
```



Alternatively, you can create a `triangulation` object to plot the mesh.

```
TO = triangulation(T,x(:),y(:),z(:));
trimesh(TO)
```



Input Arguments

T — Triangle connectivity
matrix

Triangle connectivity, specified as a 3-column matrix where each row contains the point vertices defining a triangle.

x — *x*-coordinates
column vector

x-coordinates, specified as a column vector.

y — *y*-coordinates
column vector

y-coordinates, specified as a column vector.

z — *z*-coordinates
column vector

z-coordinates, specified as a column vector.

c — Edge color
matrix

Edge color, specified as a matrix of colormap indices the same size as `z`. For more information, see "How Surface Plot Data Relates to a Colormap". For additional control over the surface coloring, use the `'EdgeColor'` and `'FaceColor'` name-value pairs.

TO — Triangulation object
`triangulation` | `delaunayTriangulation`

Triangulation object, specified as a `triangulation` or `delaunayTriangulation` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `trimesh(TO,'LineStyle','--')`

The properties listed here are only a subset. For a complete list, see Patch Properties.

`FaceColor` — Face color
`[0.85 0.85 0.85]` (default) | color name | RGB triplet | `'none'`

Face color, specified as the comma-separated pair consisting of `'FaceColor'` and a color name, an RGB triplet, or `'none'`.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`;

for example, `[0.4 0.6 0.7]`. This table lists the long and short color name options and the equivalent RGB triplet values.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black'`	`'k'`	`[0 0 0]`

`FaceAlpha` — Face transparency

1 (default) | scalar in range `[0,1]`

Face transparency, specified as the comma-separated pair consisting of `'FaceAlpha'` and a scalar in the range `[0,1]`. A value of 1 is opaque and 0 is completely transparent. Values between 0 and 1 are semitransparent.

`EdgeColor` — Edge color

`[0 0 0]` (default) | color name | RGB triplet | `'none'`

Edge color, specified as the comma-separated pair consisting of `'EdgeColor'` and a color name, an RGB triplet, or `'none'`. The default color of `[0 0 0]` corresponds to black boundaries.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. This table lists the long and short color name options and the equivalent RGB triplet values.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`

Long Name	Short Name	RGB Triplet
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black'`	`'k'`	`[0 0 0]`

`LineWidth` — Line width
`0.5` (default) | positive numeric value

Line width, specified as the comma-separated pair consisting of `'LineWidth'` and a positive numeric value in points.

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`delaunay` | `delaunayTriangulation` | `patch` | `triangulation` | `trisurf`

Introduced before R2006a

triplequad

(Not recommended) Numerically evaluate triple integral

Note `triplequad` is not recommended. Use `integral3` instead.

Syntax

```
q = triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax)
q = triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol)
q = triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol,method)
```

Description

`q = triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax)` evaluates the triple integral `fun(x,y,z)` over the three dimensional rectangular region `xmin <= x <= xmax`, `ymin <= y <= ymax`, `zmin <= z <= zmax`. The first input, `fun`, is a function handle. `fun(x,y,z)` must accept a vector `x` and scalars `y` and `z`, and return a vector of values of the integrand.

“Parameterizing Functions” explains how to provide additional parameters to the function `fun`, if necessary.

`q = triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol)` uses a tolerance `tol` instead of the default, which is `1.0e-6`.

`q = triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol,method)` uses the quadrature function specified as `method`, instead of the default `quad`. Valid values for `method` are `@quadl` or the function handle of a user-defined quadrature method that has the same calling sequence as `quad` and `quadl`.

Examples

Pass function handle `@integrnd` to `triplequad`:P

```
Q = triplequad(@integrnd,0,pi,0,1,-1,1);
```

where the file `integrnd.m` is

```
function f = integrnd(x,y,z)
f = y*sin(x)+z*cos(x);
```

Pass anonymous function handle `F` to `triplequad`:

```
F = @(x,y,z)y*sin(x)+z*cos(x);
Q = triplequad(F,0,pi,0,1,-1,1);
```

This example integrates `y*sin(x)+z*cos(x)` over the region `0 <= x <= pi`, `0 <= y <= 1`, `-1 <= z <= 1`. Note that the integrand can be evaluated with a vector `x` and scalars `y` and `z`.

See Also

`dblquad` | `integral` | `integral2` | `integral3` | `quad` | `quad2d` | `quadgk` | `quadl`

Topics

"Anonymous Functions"
"Create Function Handle"

Introduced before R2006a

triplot

2-D triangular plot

Syntax

```
triplot(T,x,y)
triplot(T,x,y,LineSpec)
triplot(TO)
triplot( ___ ,Name,Value)
h = triplot( ___ )
```

Description

`triplot(T,x,y)` plots the 2-D triangulation defined by the points in vectors `x` and `y` and a triangle connectivity matrix `T`.

`triplot(T,x,y,LineSpec)` also specifies the line style of the triangulation.

`triplot(TO)` plots the triangles defined by a 2-D `triangulation` or `delaunayTriangulation` object.

`triplot( ___ ,Name,Value)` specifies one or more properties of the plot using name-value pairs. For example, `'LineWidth',2` sets the edge width to 2 points.

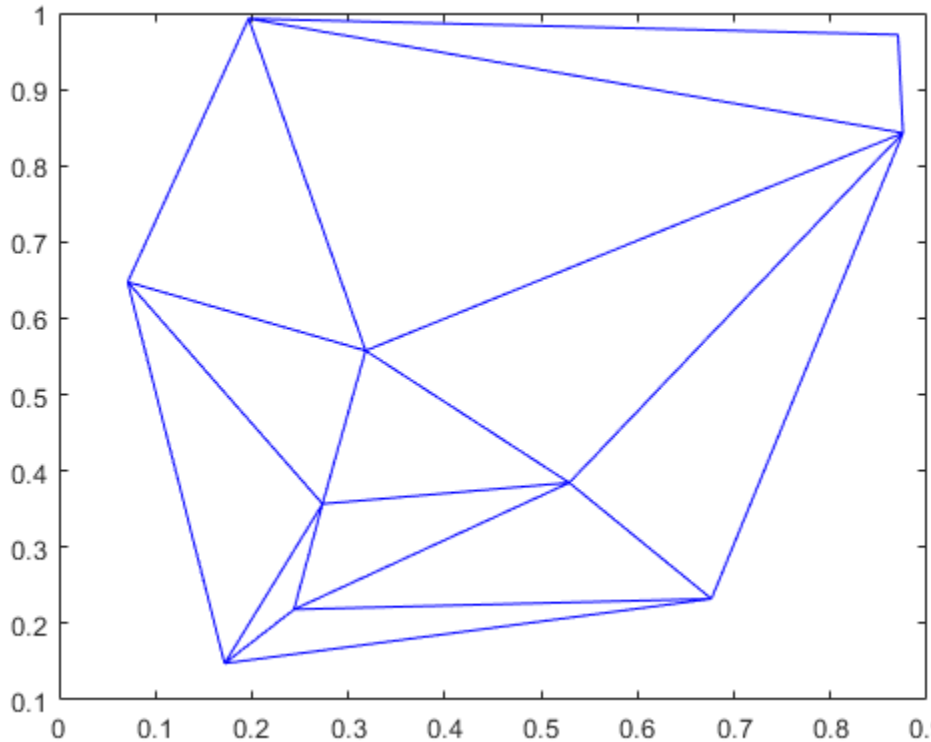
`h = triplot( ___ )` returns a chart line handle used to create the plot. Use `h` to query and modify properties of the plot. For more information, see Line Properties.

Examples

Delaunay Triangulation

Create and plot a Delaunay triangulation from a set of 2-D points.

```
P = gallery('uniformdata',10,2,2);
T = delaunayTriangulation(P);
triplot(T)
```



Input Arguments

T — Triangle connectivity
matrix

Triangle connectivity, specified as a 3-column matrix where each row contains the point vertices defining a triangle.

x — *x*-coordinates
column vector

x-coordinates, specified as a column vector.

y — *y*-coordinates
column vector

y-coordinates, specified as a column vector.

`LineSpec` — Line style, marker, and color
character vector | string

Line style, marker, and color, specified as a character vector or string containing symbols. The symbols can appear in any order. You do not need to specify all three characteristics (line style, marker, and color). For example, if you omit the line style and specify the marker, then the plot shows only the marker and no line.

Example: `'--or'` is a red dashed line with circle markers

Line Style	Description
`-`	Solid line (default)
`--`	Dashed line
`:`	Dotted line
`-.`	Dash-dot line

Marker	Description
`o`	Circle
`+`	Plus sign
`*`	Asterisk
`.`	Point
`x`	Cross
`s`	Square
`d`	Diamond
`^`	Upward-pointing triangle
`v`	Downward-pointing triangle

Marker	Description
`>`	Right-pointing triangle
`<`	Left-pointing triangle
`p`	Pentagram
`h`	Hexagram

Color	Description
`y`	yellow
`m`	magenta
`c`	cyan
`r`	red
`g`	green
`b`	blue
`w`	white
`k`	black

`T0` — Triangulation object
`triangulation` | `delaunayTriangulation`

Triangulation object, specified as a 2-D `triangulation` or `delaunayTriangulation` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `triplot(T0,'LineStyle','--')`

The properties listed here are only a subset. For a complete list, see Line Properties.

`Color` — Line color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `[0 0 0]` corresponds to black.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

`LineWidth` — Line width
`0.5` (default) | positive value

Line width, specified as a positive value in points, where 1 point = 1/72 of an inch. If the line has markers, then the line width also affects the marker edges.

See Also

`delaunay` | `delaunayTriangulation` | `triangulation` | `trimesh` | `trisurf`

Introduced before R2006a

TriRep class

(Not recommended) Triangulation representation

Note `TriRep` is not recommended. Use `triangulation` instead.

Description

`TriRep` provides topological and geometric queries for triangulations in 2-D and 3-D space. For example, for triangular meshes you can query triangles attached to a vertex, triangles that share an edge, neighbor information, circumcenters, or other features. You can create a `TriRep` directly using existing triangulation data. Alternatively, you can create a Delaunay triangulation, via `DelaunayTri`, which provides access to the `TriRep` functionality.

Construction

TriRep	(Not recommended) Triangulation representation

Methods

baryToCart	(Not recommended) Convert point coordinates from barycentric to Cartesian
cartToBary	(Not recommended) Convert point coordinates from Cartesian to barycentric
circumcenters	(Not recommended) Circumcenters of specified simplices
edgeAttachments	(Not recommended) Simplices attached to specified edges
edges	(Not recommended) Triangulation edges
faceNormals	(Not recommended) Unit normals to specified triangles
featureEdges	(Not recommended) Sharp edges of surface triangulation
freeBoundary	(Not recommended) Facets referenced by only one simplex
incenters	(Not recommended) Incenters of specified simplices
isEdge	(Not recommended) Test if vertices are joined by edge
neighbors	(Not recommended) Simplex neighbor information
size	(Not recommended) Size of triangulation matrix
vertexAttachments	(Not recommended) Return simplices attached to specified vertices

Properties

X	Coordinates of the points in the triangulation
Triangulation	Triangulation data structure

Copy Semantics

Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

Indexing

`TriRep` objects support indexing into the triangulation using parentheses (). The syntax is the same as for arrays.

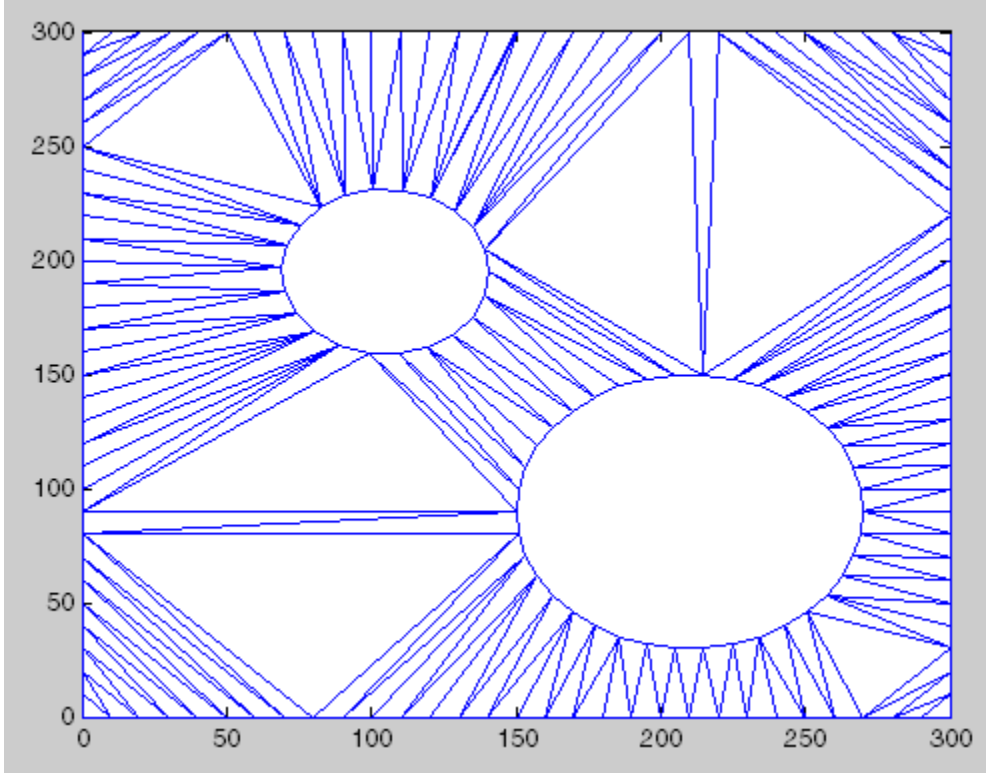
Examples

Load a 2-D triangulation and use the `TriRep` constructor to build an array of the free boundary edges:

```
load trimesh2d
```

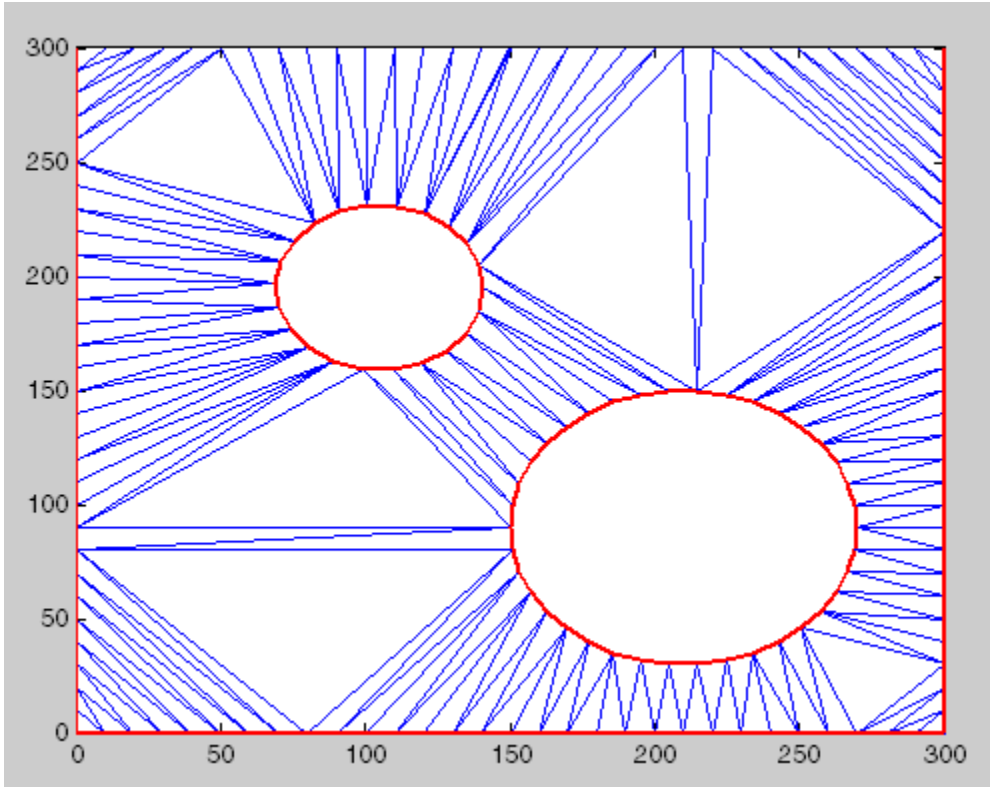
This loads triangulation `tri` and vertex coordinates `x`, `y`:

```
trep = TriRep(tri, x,y);
fe = freeBoundary(trep)';
triplot(trep);
```



You can add the free edges `fe` to the plot:

```
hold on;
plot(x(fe), y(fe), 'r','LineWidth',2);
hold off;
axis([-50 350 -50 350]);
axis equal;
```



See Also

`delaunayTriangulation` | `scatteredInterpolant` | `triangulation`

TriRep

Class: `TriRep`

(Not recommended) Triangulation representation

Note `TriRep` is not recommended. Use `triangulation` instead.

Syntax

```
TR = TriRep(TRI, X, Y)
TR = TriRep(TRI, X, Y, Z)
TR = TriRep(TRI, X)
```

Description

`TR = TriRep(TRI, X, Y)` creates a 2-D triangulation representation from the triangulation matrix `TRI` and the vertex coordinates `(X, Y)`. `TRI` is an `m`-by-3 matrix that defines the triangulation in face-vertex format, where `m` is the number of triangles. Each row of `TRI` is a triangle defined by indices into the column vector of vertex coordinates `(X, Y)`.

`TR = TriRep(TRI, X, Y, Z)` creates a 3-D triangulation representation from the triangulation matrix `TRI` and the vertex coordinates `(X, Y, Z)`. `TRI` is an `m`-by-3 or `m`-by-4 matrix that defines the triangulation in simplex-vertex format, where `m` is the number of simplices; triangles or tetrahedra in this case. Each row of `TRI` is a simplex defined by indices into the column vector of vertex coordinates `(X, Y, Z)`.

`TR = TriRep(TRI, X)` creates a triangulation representation from the triangulation matrix `TRI` and the vertex coordinates `X`. `TRI` is an `m`-by-`n` matrix that defines the triangulation in simplex-vertex format, where `m` is the number of simplices and `n` is the number of vertices per simplex. Each row of `TRI` is a simplex defined by indices into the array of vertex coordinates `X`. `X` is an `mpts`-by-`ndim` matrix where `mpts` is the number of

points and `ndim` is the dimension of the space where the points reside, where $2 \leq$ `ndim` ≤ 3.

Examples

Load a 3-D tetrahedral triangulation compute the free boundary. First, load triangulation `tet` and vertex coordinates `X`.

```
load tetmesh
```

Create the triangulation representation and compute the free boundary.

```
trep = TriRep(tet, X);
[tri, Xb] = freeBoundary(trep);
```

See Also

`delaunayTriangulation` | `scatteredInterpolant`

TriScatteredInterp class

(Not recommended) Interpolate scattered data

Note `TriScatteredInterp` is not recommended. Use `scatteredInterpolant` instead.

Description

`TriScatteredInterp` is used to perform interpolation on a scattered dataset that resides in 2-D or 3-D space. A scattered data set defined by locations `X` and corresponding values `V` can be interpolated using a Delaunay triangulation of `X`. This produces a surface of the form `V = F(X)`. The surface can be evaluated at any query location `QX`, using `QV = F(QX)`, where `QX` lies within the convex hull of `X`. The interpolant `F` always goes through the data points specified by the sample.

Construction

TriScatteredInterp	(Not recommended) Interpolate scattered data

Properties

`X`	Defines locations of scattered data points in 2-D or 3-D space.	
`V`	Defines value associated with each data point.	
`Method`	Defines method used to interpolate the data .	
	`natural`	Natural neighbor interpolation

	`linear`	Linear interpolation (default)
	`nearest`	Nearest neighbor interpolation

Copy Semantics

Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

Examples

Create a data set:

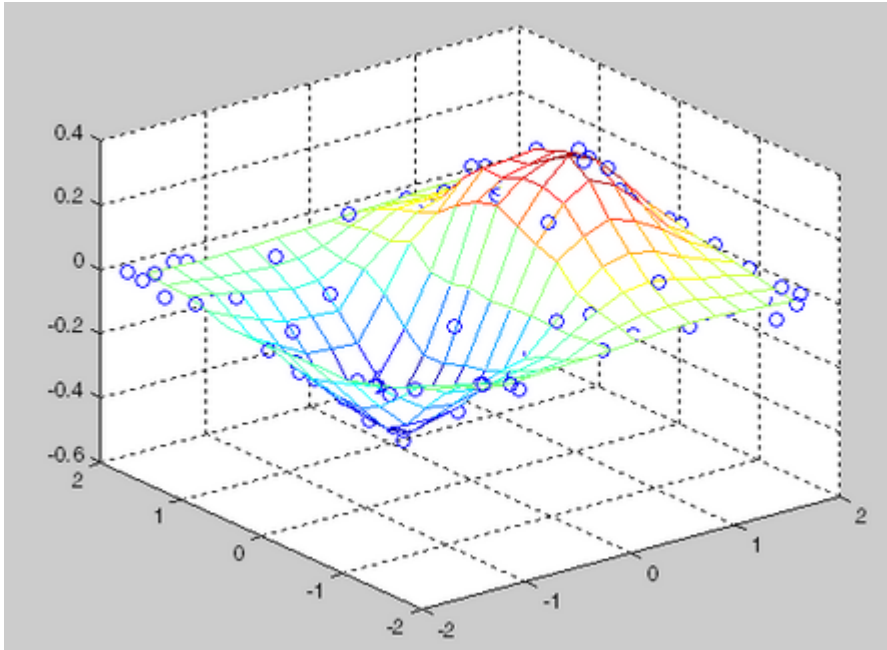
```
x = rand(100,1)*4-2;
y = rand(100,1)*4-2;
z = x.*exp(-x.^2-y.^2);
```

Construct the interpolant:

```
F = TriScatteredInterp(x,y,z);
```

Evaluate the interpolant at the locations `(qx, qy)`. The corresponding value at these locations is `qz`:

```
ti = -2:.25:2;
[qx,qy] = meshgrid(ti,ti);
qz = F(qx,qy);
mesh(qx,qy,qz);
hold on;
plot3(x,y,z,'o');
```



More About

Delaunay Triangulation

The Delaunay triangulation of a set of points is a triangulation such that the unique circle circumscribed about each triangle contains no other points in the set. The convex hull of a set of points is the smallest convex set containing all points of the original set. These definitions extend naturally to higher dimensions.

See Also

`interp1` | `interp2` | `interp3` | `meshgrid` | `scatteredInterpolant` | `triangulation`

TriScatteredInterp

Class: `TriScatteredInterp`

(Not recommended) Interpolate scattered data

Note `TriScatteredInterp` is not recommended. Use `scatteredInterpolant` instead.

Syntax

```
F = TriScatteredInterp()
F = TriScatteredInterp(X, V)
F = TriScatteredInterp(X, Y, V)
F = TriScatteredInterp(X, Y, Z, V)
F = TriScatteredInterp(DT, V)
F = TriScatteredInterp(..., method)
```

Description

`F = TriScatteredInterp()` creates an empty scattered data interpolant. This can subsequently be initialized with sample data points and values (`Xdata`, `Vdata`) via `F.X = Xdata` and `F.V = Vdata`.

`F = TriScatteredInterp(X, V)` creates an interpolant that fits a surface of the form `V = F(X)` to the scattered data in (`X`, `V`). `X` is a matrix of size `mpts`-by-`ndim`, where `mpts` is the number of points and `ndim` is the dimension of the space where the points reside (`ndim` is 2 or 3). The column vector `V` defines the values at `X`, where the length of `V` equals `mpts`.

`F = TriScatteredInterp(X, Y, V)` and `F = TriScatteredInterp(X, Y, Z, V)` allow the data point locations to be specified in alternative column vector format when working in 2-D and 3-D.

`F = TriScatteredInterp(DT, V)` uses the specified `DelaunayTri` object `DT` as a basis for computing the interpolant. `DT` is a Delaunay triangulation of the scattered data locations, `DT.X`. The matrix `DT.X` is of size `mpts`-by-`ndim`, where `mpts` is the number of points and `ndim` is the dimension of the space where the points reside, `2 <= ndim <= 3`. `V` is a column vector that defines the values at `DT.X`, where the length of `V` equals `mpts`.

`F = TriScatteredInterp(..., method)` allows selection of the technique `method` used to interpolate the data.

Input Arguments

`X`	Matrix of size `mpts`-by-`ndim`, where `mpts` is the number of points and `ndim` is the dimension of the space where the points reside. Input may also be specified as column vectors (`X`, `Y`) or (`X`, `Y`, `Z`)	
`V`	Column vector that defines the values at `X`, where the length of `V` equals `mpts`.	
`DT`	Delaunay triangulation of the scattered data locations	
`method`	`natural`	Natural neighbor interpolation
	`linear`	Linear interpolation (default)
	`nearest`	Nearest-neighbor interpolation

Output Arguments

`F`	Creates an interpolant that fits a surface of the form `V = F(X)` to the scattered data.

Evaluation

To evaluate the interpolant, express the statement in Monge's form `Vq = F(Xq)`, `Vq = F(Xq,Yq)`, or `Vq = F(Xq,Yq,Zq)` where `Vq` is the value of the interpolant at the query location and `Xq`, `Yq`, and `Zq` are the vectors of point locations.

Examples

Create a data set:

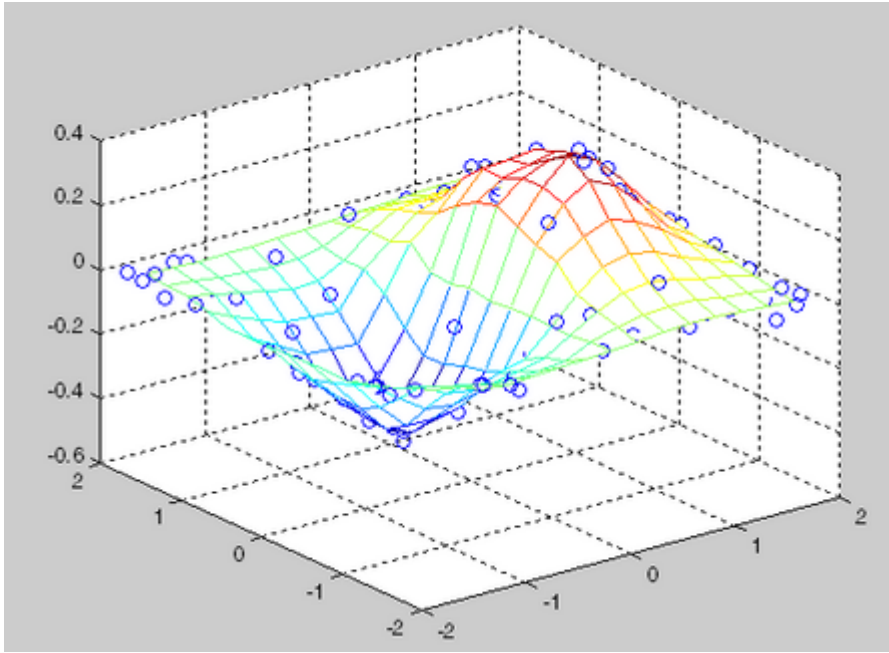
```
x = rand(100,1)*4-2;
y = rand(100,1)*4-2;
z = x.*exp(-x.^2-y.^2);
```

Construct the interpolant:

```
F = TriScatteredInterp(x,y,z);
```

Evaluate the interpolant at the locations `(qx, qy)`. The corresponding value at these locations is `qz` .

```
ti = -2:.25:2;
[qx,qy] = meshgrid(ti,ti);
qz = F(qx,qy);
mesh(qx,qy,qz);
hold on;
plot3(x,y,z,'o');
```



More About

Delaunay Triangulation

The Delaunay triangulation of a set of points is a triangulation such that the unique circle circumscribed about each triangle contains no other points in the set.

See Also

`delaunayTriangulation` | `interp1` | `interp2` | `interp3` | `meshgrid`

trisurf

Triangular surface plot

Syntax

```
trisurf(T,x,y,z)
trisurf(TO)
trisurf( ___ ,c)
trisurf( ___ ,Name,Value)
h = trisurf( ___ )
```

Description

`trisurf(T,x,y,z)` plots the 3-D triangular surface defined by the points in vectors `x`, `y`, and `z`, and a triangle connectivity matrix `T`.

`trisurf(TO)` plots the surface defined by a 3-D `triangulation` or `delaunayTriangulation` object.

`trisurf( ___ ,c)` also specifies the surface color for either of the previous syntaxes.

`trisurf( ___ ,Name,Value)` specifies one or more properties of the surface plot using name-value pairs. For example, `'FaceColor','y'` sets the face color to yellow.

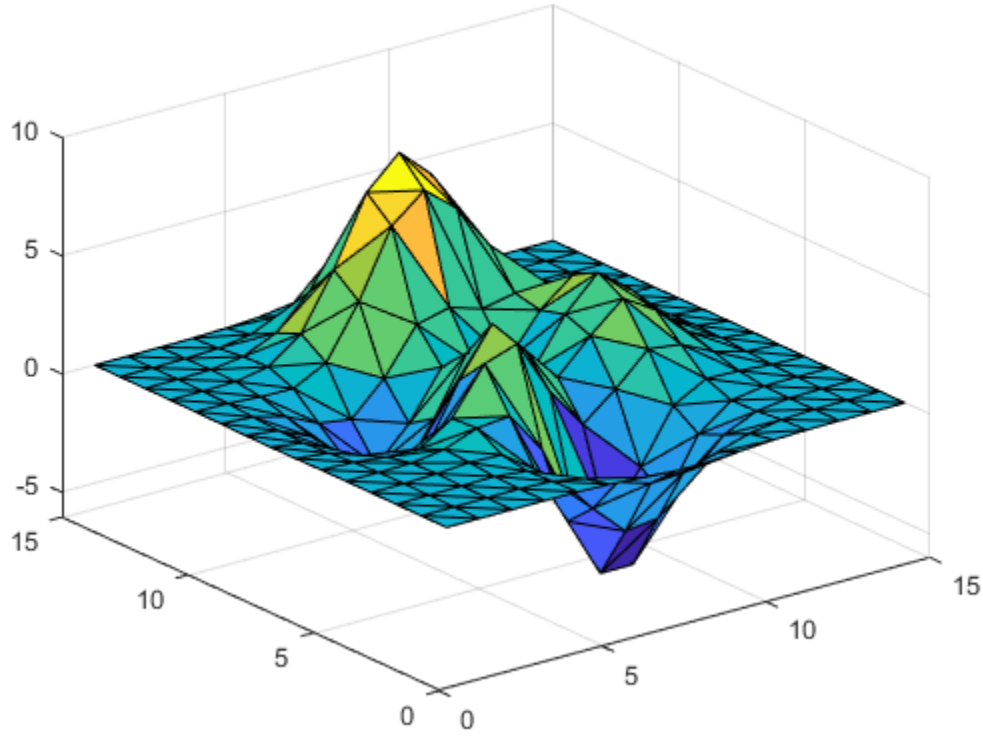
`h = trisurf( ___ )` returns a `patch` object used to create the surface plot. Use `h` to query and modify properties of the plot. For more information, see Patch Properties.

Examples

3-D Surface Plot

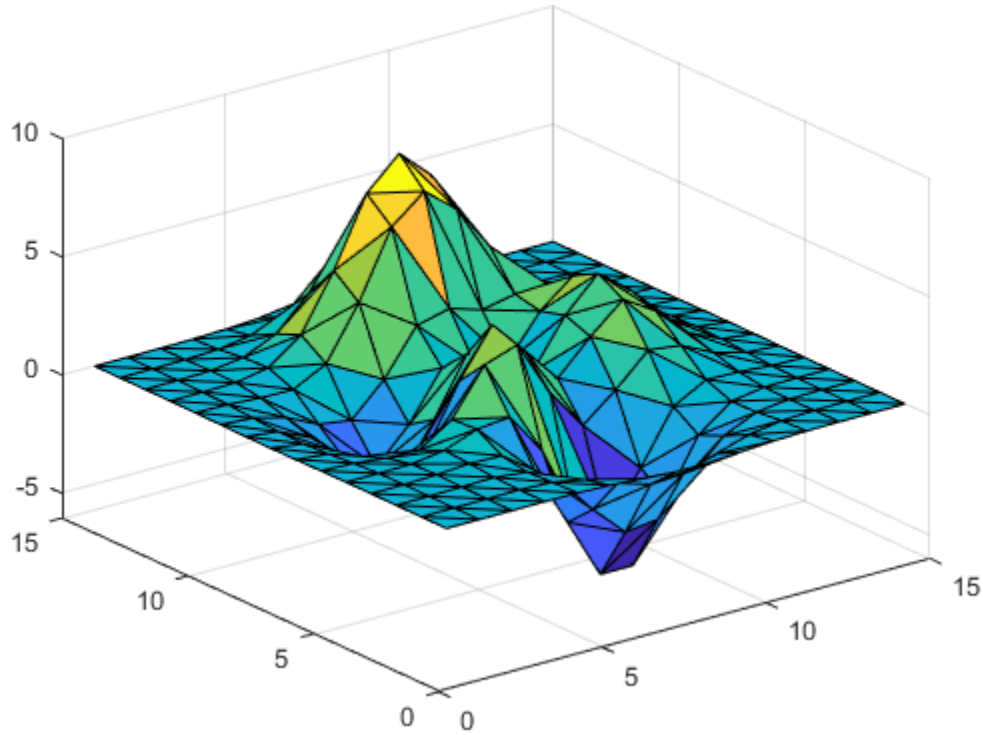
Create a set of 3-D points and compute the Delaunay triangulation using the `delaunay` function. Plot the surface defined by the triangulation.

```
[x,y] = meshgrid(1:15,1:15);
z = peaks(15);
T = delaunay(x,y);
trisurf(T,x,y,z)
```



Alternatively, you can create and plot a `triangulation` object.

```
T0 = triangulation(T,x(:),y(:),z(:));
trisurf(T0)
```



Input Arguments

T — Triangle connectivity
matrix

Triangle connectivity, specified as a 3-column matrix where each row contains the point vertices defining a triangle face.

x — *x*-coordinates
column vector

x-coordinates, specified as a column vector.

y — *y*-coordinates
column vector

y-coordinates, specified as a column vector.

z — *z*-coordinates
column vector

z-coordinates, specified as a column vector.

c — Face color
matrix

Face color, specified as a matrix of colormap indices the same size as `z`. For more information, see "How Surface Plot Data Relates to a Colormap". For additional control over the surface coloring, use the `'EdgeColor'` and `'FaceColor'` name-value pairs.

`TO` — Triangulation object
`triangulation` | `delaunayTriangulation`

Triangulation object, specified as a `triangulation` or `delaunayTriangulation` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `trisurf(TO,'FaceColor','r')`

The properties listed here are only a subset. For a complete list, see Patch Properties.

`FaceColor` — Face color
`[0.85 0.85 0.85]` (default) | color name | RGB triplet | `'none'`

Face color, specified as the comma-separated pair consisting of `'FaceColor'` and a color name, an RGB triplet, or `'none'`.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`;

for example, `[0.4 0.6 0.7]`. This table lists the long and short color name options and the equivalent RGB triplet values.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black'`	`'k'`	`[0 0 0]`

`FaceAlpha` — Face transparency
1 (default) | scalar in range `[0,1]`

Face transparency, specified as the comma-separated pair consisting of `'FaceAlpha'` and a scalar in the range `[0,1]`. A value of 1 is opaque and 0 is completely transparent. Values between 0 and 1 are semitransparent.

`EdgeColor` — Edge color
`[0 0 0]` (default) | color name | RGB triplet | `'none'`

Edge color, specified as the comma-separated pair consisting of `'EdgeColor'` and a color name, an RGB triplet, or `'none'`. The default color of `[0 0 0]` corresponds to black boundaries.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. This table lists the long and short color name options and the equivalent RGB triplet values.

Long Name	Short Name	RGB Triplet
`'yellow'`	`'y'`	`[1 1 0]`
`'magenta'`	`'m'`	`[1 0 1]`
`'cyan'`	`'c'`	`[0 1 1]`

Long Name	Short Name	RGB Triplet
`'red'`	`'r'`	`[1 0 0]`
`'green'`	`'g'`	`[0 1 0]`
`'blue'`	`'b'`	`[0 0 1]`
`'white'`	`'w'`	`[1 1 1]`
`'black'`	`'k'`	`[0 0 0]`

`LineWidth` — Line width
`0.5` (default) | positive numeric value

Line width, specified as the comma-separated pair consisting of `'LineWidth'` and a positive numeric value in points.

`LineStyle` — Line style
`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as one of the options listed in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`delaunay` | `delaunayTriangulation` | `patch` | `surf` | `tetramesh` | `triangulation` | `trimesh` | `triplot`

Introduced before R2006a

triu

Upper triangular part of matrix

Syntax

```
U = triu(A)
U = triu(A,k)
```

Description

`U = triu(A)` returns the upper triangular on page 1-15959 portion of matrix `A`.

`U = triu(A,k)` returns the elements on and above the kth diagonal of `A`.

Examples

Extract Upper Triangular Portions of Matrix

Create a 4-by-4 matrix of ones. Extract the upper triangular portion.

```
A = ones(4)
```

```
A = 4×4

     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1
```

```
B = triu(A)
```

```
B = 4×4

     1     1     1     1
```

```
     0     1     1     1
     0     0     1     1
     0     0     0     1
```

Extract only the elements above the main diagonal.

```
C = triu(A,1)
```

```
C = 4×4

     0     1     1     1
     0     0     1     1
     0     0     0     1
     0     0     0     0
```

Input Arguments

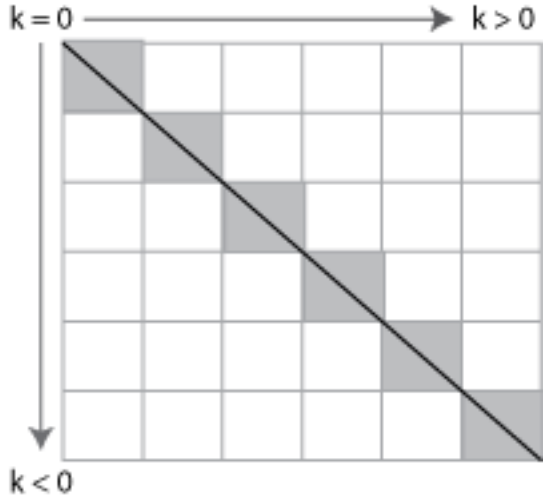
A — Input matrix
matrix

Input matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: Yes

k — Diagonals to include
`0` (default) | scalar

Diagonals to include, specified as a scalar. `k = 0` is the main diagonal, `k > 0` is above the main diagonal, and `k < 0` is below the main diagonal.

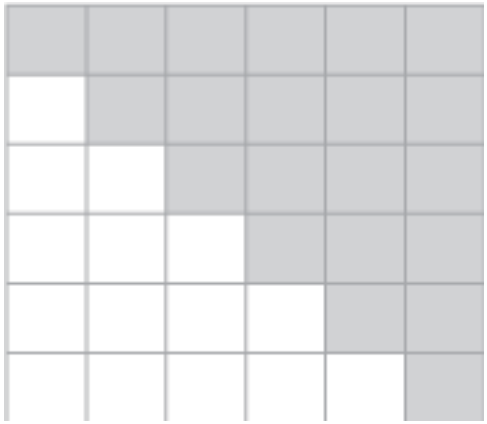


Example: `triu(A,3)`

More About

Upper Triangular

The upper triangular portion of a matrix includes the main diagonal and all elements above it. The shaded blocks in this graphic depict the upper triangular portion of a 6-by-6 matrix.



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If you supply the argument that represents the order of the diagonal matrix, then it must be a real and scalar integer value.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- If you supply the argument that represents the order of the diagonal matrix, then it must be a real and scalar integer value.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`bandwidth` | `diag` | `istriu` | `tril`

Introduced before R2006a

true

Logical 1 (true)

Syntax

```
true
T = true(n)
T = true(sz)
T = true(sz1,...,szN)
T = true( ___ ,'like',p)
```

Description

`true` is shorthand for the logical value `1`.

`T = true(n)` is an `n`-by-`n` matrix of logical ones.

`T = true(sz)` is an array of logical ones where the size vector, `sz`, defines `size(T)`. For example, `true([2 3])` returns a 2-by-3 array of logical ones.

`T = true(sz1,...,szN)` is a `sz1`-by-`...`-by-`szN` array of logical ones where `sz1,...,szN` indicates the size of each dimension. For example, `true(2,3)` returns a 2-by-3 array of logical ones.

`T = true( ___ ,'like',p)` returns an array of logical ones of the same sparsity as the logical variable `p` using any of the previous size syntaxes.

Examples

Generate Square Matrix of Logical Ones

Use `true` to generate a 3-by-3 square matrix of logical ones.

```
A = true(3)
```

```
A = 3x3 logical array

   1   1   1
   1   1   1
   1   1   1
```

```
class(A)
```

```
ans =
'logical'
```

The result is of class `logical`.

Generate Array of Logical Ones with Arbitrary Dimensions

Use `true` to generate a 3-by-2-by-2 matrix of logical ones.

```
true(3,2,2)
```

```
ans = 3x2x2 logical array
ans(:,:,1) =

   1   1
   1   1
   1   1


ans(:,:,2) =

   1   1
   1   1
   1   1
```

Alternatively, you can use a size vector to specify the size of the matrix.

```
true([3,2,2])
```

```
ans = 3x2x2 logical array
ans(:,:,1) =

   1   1
```

```
   1   1
   1   1


ans(:,:,2) =

   1   1
   1   1
   1   1
```

Note that specifying multiple vector inputs returns an error.

Execute Logic Statement

`true` along with `false` can be used to execute logic statements.

Test the logical statement

`~(A and B) = (~A) or (~B)`

for `A = true` and `B = false`.

```
~(true & false) == (~true) | (~false)
```

```
ans = logical
   1
```

The result is logical 1 (true), since the logical statements on both sides of the equation are equivalent. This logical statement is an instance of De Morgan's Law.

Generate Logical Array of Selected Sparsity

Generate a logical array of the same sparsity as the selected array.

```
A = logical(sparse(5,3));
whos A
```

```
  Name      Size            Bytes  Class     Attributes

  A         5x3                41  logical   sparse
```

```
T = true(4,'like',A);
whos T
```

```
  Name      Size            Bytes  Class     Attributes

  T         4x4               184  logical   sparse
```

The output array `T` has the same `sparse` attribute and data-type as the specified array `A`.

Input Arguments

`n` — Size of square matrix

integer

Size of square matrix, specified as an integer. `n` sets the output array size to `n`-by-`n`. For example, `true(3)` returns a 3-by-3 array of logical ones.

- If `n` is `0`, then `T` is an empty matrix.
- If `n` is negative, then it is treated as `0`.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz` — Size vector

row vector of integers

Size vector, specified as a row vector of integers. For example, `true([2 3])` returns a 2-by-3 array of logical ones.

- If the size of any dimension is `0`, then `T` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- If any trailing dimensions greater than `2` have a size of `1`, then the output, `T`, does not include those dimensions.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz1,...,szN` — Size inputs

comma-separated list of integers

Size inputs, specified by a comma-separated list of integers. For example, `true(2,3)` returns a 2-by-3 array of logical ones.

- If the size of any dimension is `0`, then `T` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- If any trailing dimensions greater than `2` have a size of `1`, then the output, `T`, does not include those dimensions.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

p — Prototype
logical variable

Prototype, specified as a logical variable.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`
Complex Number Support: Yes

Output Arguments

T — Output of logical ones
scalar | vector | matrix | N-D array

Output of logical ones, returned as a scalar, vector, matrix, or N-D array.

Data Types: `logical`

Tips

- `true(n)` is much faster and more memory efficient than `logical(true(n))`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Dimensions must be real, nonnegative, integers.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Dimensions must be real, nonnegative, integers.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `true` in the Parallel Computing Toolbox documentation.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `true` in the Parallel Computing Toolbox documentation.

See Also

`false` | `logical`

Topics

"Class Support for Array-Creation Functions"

Introduced before R2006a

try, catch

Execute statements and catch resulting errors

Syntax

```
try
   statements
catch exception
   statements
end
```

Description

`try` *`statements`*`, catch` *`statements`* `end` executes the statements in the `try` block and catches resulting errors in the `catch` block. This approach allows you to override the default error behavior for a set of program statements. If any statement in a `try` block generates an error, program control goes immediately to the `catch` block, which contains your error handling statements.

`exception` is an `MException` object that allows you to identify the error. The `catch` block assigns the current exception object to the variable in *`exception`*.

Both `try` and `catch` blocks can contain nested `try/catch` statements.

Examples

Supplement Error Message

Create two matrices that you cannot concatenate vertically.

```
A = rand(3);
B = ones(5);

C = [A; B];
```

```
Error using vertcat
Dimensions of matrices being concatenated are not consistent.
```

Use `try/catch` to display more information about the dimensions.

```
try
   C = [A; B];
catch ME
   if (strcmp(ME.identifier,'MATLAB:catenate:dimensionMismatch'))
      msg = ['Dimension mismatch occurred: First argument has ', ...
            num2str(size(A,2)),' columns while second has ', ...
            num2str(size(B,2)),' columns.'];
        causeException = MException('MATLAB:myCode:dimensions',msg);
        ME = addCause(ME,causeException);
   end
   rethrow(ME)
end
```

```
Error using vertcat
Dimensions of matrices being concatenated are not consistent.

Caused by:
    Dimension mismatch occurred: First argument has 3 columns while second has 5 colum
```

If matrix dimensions do not agree, MATLAB displays more information about the mismatch. Any other errors appear as usual.

Repackage Error as Warning

Catch any exception generated by calling the nonexistent function, `notaFunction`. If there is an exception, issue a warning and assign the output a value of 0.

```
try
    a = notaFunction(5,6);
catch
    warning('Problem using function.  Assigning a value of 0.');
    a = 0;
end
```

```
Warning: Problem using function.  Assigning a value of 0.
```

By itself, the call to `notaFunction` results in an error. If you use `try` and `catch`, this code catches any exception and repackages it as a warning, allowing MATLAB to continue executing subsequent commands.

Handle Different Types of Errors

Use `try/catch` to handle different types of errors in different ways.

- If the function `notaFunction` is undefined, issue a warning instead of an error and assign the output a value of `NaN`.
- If `notaFunction.m` exists, but is a script instead of a function, issue a warning instead of an error, run the script, and assign the output a value of `0`.
- If MATLAB throws an error for any other reason, rethrow the exception.

```
try
    a = notaFunction(5,6);
catch ME
    switch ME.identifier
        case 'MATLAB:UndefinedFunction'
            warning('Function is undefined.  Assigning a value of NaN.');
            a = NaN;
        case 'MATLAB:scriptNotAFunction'
            warning(['Attempting to execute script as function. '...
                'Running script and assigning output a value of 0.']);
            notaFunction;
            a = 0;
        otherwise
            rethrow(ME)
    end
end
```

```
Warning: Function is undefined.  Assigning a value of NaN.
```

Tips

- You cannot use multiple `catch` blocks within a `try` block, but you can nest complete `try/catch` blocks.
- Unlike some other languages, MATLAB does not allow the use of a `finally` block within `try/catch` statements.

See Also

`MException` | `assert` | `error`

Introduced before R2006a

tscollection

Create `tscollection` object

Description

A `tscollection` object is a collection of one or more `timeseries` objects with the same time vector. Collections of `timeseries` are useful for analyzing multiple data trends that occur simultaneously.

Creation

To create a `tscollection` object, use the `tscollection` function.

Syntax

```
tsc = tscollection(ts)
tsc = tscollection(timevals)
```

Description

`tsc = tscollection(ts)` creates a `tscollection` object containing the `timeseries` objects in `ts`. The input `ts` can be a single `timeseries` object or a cell array of `timeseries` objects. Each `timeseries` must have the same time vector.

`tsc = tscollection(timevals)` defines the time vector of a `tscollection`.

Input Arguments

`ts` — Input `timeseries`
scalar | cell array

Input `timeseries`, specified as a scalar or a cell array.

`timevals` — Sample times
scalar | vector

Sample times, specified as a numeric scalar or vector, or a cell array of date character vectors. Valid date character vectors can have the following forms:

Format	Example
`dd-mmm-yyyy HH:MM:SS`	`01-Mar-2000 15:45:17`
`dd-mmm-yyyy`	`01-Mar-2000`
`mm/dd/yy`	`03/01/00`
`mm/dd`	`03/01`
`HH:MM:SS`	`15:45:17`
`HH:MM:SS PM`	`3:45:17 PM`
`HH:MM`	`15:45`
`HH:MM PM`	`3:45 PM`
`mmm.dd,yyyy HH:MM:SS`	`Mar.01,2000 15:45:17`
`mmm.dd,yyyy`	`Mar.01,2000`
`mm/dd/yyyy`	`03/01/2000`

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `cell`

Name-Value Pair Arguments

Example: `tsc = tscollection(ts,'Name','MyTSC')`

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify the name and value pair arguments in any order as `Name1,Value1,Name2,Value2`.

`Name` — `tscollection` name
character vector

`tscollection` name, specified as a character vector.

`isDatenum` — `datenum` indicator
`true` (default) | `false`

datenum indicator, specified as true or false. A value of true indicates that the time vector of the tscollection object consists of datenum values. For more information, see datenum.

Data Types: logical

Properties

Name — tscollection name
character vector

tscollection name, represented as a character vector.

Time — Time values
vector

Time values, represented as a vector.

When TimeInfo.StartDate is empty, values are measured relative to 0. When TimeInfo.StartDate is defined, values represent date character vectors measured relative to the StartDate.

The length of Time must be the same as the first or the last dimension of Data for each collection.

TimeInfo — Time information
collection of fields

Time information, represented as a collection of the following fields describing the time vector:

- Units — Time units with the value 'weeks', 'days', 'hours', 'minutes', 'seconds', 'milliseconds', 'microseconds', or 'nanoseconds'.
- Start — Start time.
- End — End time (read only).
- Increment — Interval between subsequent time values (NaN when times are not uniformly sampled).
- Length — Length of time vector (read only).
- Format — Character vector defining the date display (see datestr).

- `StartDate` — Date character vector defining the reference date (see `setabstime`).
- `UserData` — Additional user-defined information.

To access the value of a field, use the form `tsc.TimeInfo.field` for a `tscollection` object `tsc`.

Object Functions

Modify and Combine

addsampletocollection	Add sample to tscollection
addts	Add timeseries to tscollection
delsamplefromcollection	Delete sample from tscollection
horzcat	Horizontally concatenate tscollection objects
removets	Remove timeseries from tscollection
resample	Resample tscollection time vector
set	Set tscollection properties
setabstime	Set tscollection times as date character vectors
settimeseriesnames	Rename timeseries in tscollection
vertcat	Vertically concatenate tscollection objects

Query

get	Query tscollection properties
getabstime	Convert tscollection time vector to cell array
getsampleusingtime	Subset of tscollection data
gettimeseriesnames	Names of timeseries in tscollection
isempty	Determine if tscollection is empty
length	Length of tscollection time vector
size	Size of tscollection

Examples

Create `tscollection`

Create a `tscollection` object from two `timeseries` objects.

```
ts1 = timeseries([1.1 2.9 3.7 4.0 3.0]',1:5,...
'Name','Acceleration');
```

```
ts2 = timeseries([3.2 4.2 6.2 8.5 1.1]',1:5,...
'Name','Speed');
ts = tscollection({ts1;ts2})
```

```
Time Series Collection Object: unnamed

Time vector characteristics

      Start time            1 seconds
      End time              5 seconds

Member Time Series Objects:

      Acceleration
      Speed
```

See Also

timeseries

Topics

"Time Series Objects and Collections"

Introduced before R2006a

tsdata.event

Create `tsdata.event` object

Description

A `tsdata.event` object contains event information for a `timeseries` object within its properties. To add events in a `tsdata.event` to a `timeseries`, you can use the `addevent` function.

Creation

To create a `tsdata.event` object, use the `tsdata.event` function.

Syntax

```
tse = tsdata.event(eventname,eventtime)
tse = tsdata.event(eventname,eventtime,'Datenum')
```

Description

`tse = tsdata.event(eventname,eventtime)` creates a `tsdata.event` object named `eventname` occurring at the time `eventtime`.

`tse = tsdata.event(eventname,eventtime,'Datenum')` indicates that the `eventtime` value is a serial date number generated by the `datenum` function. `eventtime` is converted to a date character vector after the event is created.

Input Arguments

`eventname` — Event name
character vector

Event name, specified as a character vector.

`eventtime` — Event time
scalar

Event time, specified as a numeric scalar.

Properties

`EventData` — Event data
array

Event data, represented as an array containing user-defined information about the event.

`Name` — Event name
character vector

Event name, represented as a character vector.

`Time` — Event time
vector

Time values, represented as a numeric scalar.

`Units` — Time units
`'weeks'` | `'days'` | `'hours'` | `'minutes'` | `'seconds'` | `'milliseconds'` | `'microseconds'` | `'nanoseconds'`

Time units, specified as `'weeks'`, `'days'`, `'hours'`, `'minutes'`, `'seconds'`, `'milliseconds'`, `'microseconds'`, or `'nanoseconds'`.

`StartDate` — Start date
character vector

Start date, specified as a valid `datestr` character vector.

Object Functions

findEvent	Query tsdata.event by name
get	Query tsdata.event properties
getTimeStr	Query tsdata.event times
set	Set tsdata.event properties

Examples

Create Event

Create a `tsdata.event` object.

```
tse = tsdata.event('MyEvent',6)
```

```
    EventData: []
         Name: 'MyEvent'
         Time: 6
        Units: 'seconds'
    StartDate: ''
```

Create a `timeseries` object and add the event to the `timeseries`.

```
ts = timeseries((1:5)',1:5);
ts = addevent(ts,tse);
ts.Events
```

```
    EventData: []
         Name: 'MyEvent'
         Time: 6
        Units: 'seconds'
    StartDate: ''
```

See Also

`addevent` | `datenum` | `datestr` | `timeseries`

Topics

"Time Series Objects and Collections"

Introduced before R2006a

findEvent

Query `tsdata.event` by name

Syntax

```
[event,ind] = findEvent(tse,tsename)
[event,ind] = findEvent(tse,tsename,n)
```

Description

`[event,ind] = findEvent(tse,tsename)` returns the first `tsdata.event` object `event` with the name `tsename` in an array of `tsdata.event` objects `tse`. The output argument `ind` is the index of `tse` corresponding to `event`.

`[event,ind] = findEvent(tse,tsename,n)` returns the `n`th `tsdata.event` object in the array `tse` and its index.

Examples

Find Event by Name

Create a vector containing two `tsdata.event` objects with the name `MyEvent`. Find the first event named `MyEvent`.

```
tse1 = tsdata.event('MyEvent',1);
tse2 = tsdata.event('MyEvent',2);
tse = [tse1 tse2];
[event1,ind1] = findEvent(tse,'MyEvent');
event1
```

```
    EventData: []
         Name: 'MyEvent'
         Time: 1
        Units: 'seconds'
    StartDate: ''
```

```
ind1
```

```
ind1 = 1
```

Find the second event named `MyEvent`.

```
[event2,ind2] = findEvent(tse,'MyEvent',2);
event2
```

```
    EventData: []
         Name: 'MyEvent'
         Time: 2
        Units: 'seconds'
    StartDate: ''
```

```
ind2
```

```
ind2 = 2
```

Input Arguments

`tse` — Input `tsdata.event`
array

Input `tsdata.event`, specified as an array.

`tsename` — `tsdata.event` name
character vector

`tsdata.event` name, specified as a character vector.

`n` — `tsdata.event` number
integer scalar

`tsdata.event` number, specified as an integer scalar.

Output Arguments

`event` — Output `tsdata.event`
scalar

Output `tsdata.event`, specified as a scalar.

`ind` — `tsdata.event` index
integer scalar

`tsdata.event` index, specified as an integer scalar.

See Also

`timeseries` | `tscollection` | `tsdata.event`

Introduced before R2006a

get

Query `tsdata.event` properties

Syntax

```
tseinfo = get(tse)
tseinfo = get(tse,propname)
```

Description

`tseinfo = get(tse)` returns all properties of the `tsdata.event` object `tse`.

`tseinfo = get(tse,propname)` returns the property value of the property `propname`. For a full list of `tsdata.event` property names, see `tsdata.event`.

Examples

Display `tsdata.event` Property

Create a `tsdata.event` object and display its time units.

```
tse = tsdata.event('MyEvent',3);
tseinfo = get(tse,'Units')
```

```
tseinfo =
'seconds'
```

Input Arguments

`tse` — Input `tsdata.event`
scalar

Input `tsdata.event`, specified as a scalar.

`propname` — Property name
character vector

Property name, specified as a character vector. For a full list of `tsdata.event` property names, see `tsdata.event`.

See Also

`set` | `timeseries` | `tscollection` | `tsdata.event`

Introduced before R2006a

getTimeStr

Query `tsdata.event` times

Syntax

```
eventtimes = getTimeStr(tse,timeunits)
```

Description

`eventtimes = getTimeStr(tse,timeunits)` returns a cell array containing the times for each `tsdata.event` in an array `tse` with units `timeunits`.

Examples

Event Times

Create an array of `tsdata.event` objects and display their times in seconds.

```
tse1 = tsdata.event('MyEvent',1);
tse2 = tsdata.event('MyEvent',2);
tse = [tse1 tse2];
eventtimes = getTimeStr(tse,'seconds')
```

```
eventtimes = 1x2 cell array
    {'1.000'}    {'2.000'}
```

Input Arguments

`tse` — Input `tsdata.event`
array

Input `tsdata.event`, specified as an array.

`timeunits` — Time units
`'weeks'` | `'days'` | `'hours'` | `'minutes'` | `'seconds'` | `'milliseconds'` | `'microseconds'` | `'nanoseconds'`

Time units, specified as `'weeks'`, `'days'`, `'hours'`, `'minutes'`, `'seconds'`, `'milliseconds'`, `'microseconds'`, or `'nanoseconds'`.

See Also

`timeseries` | `tscollection` | `tsdata.event`

Introduced before R2006a

set

Set `tsdata.event` properties

Syntax

```
tseprop = set(tse)
tseprop = set(tse,propname)
set(tse,propname,propval)
```

Description

`tseprop = set(tse)` returns all property values of a `tsdata.event` object `tse`.

`tseprop = set(tse,propname)` displays the property specified in `propname`. This syntax is equivalent to the command `tseprop = tse.propname`. For a full list of `tsdata.event` property names, see `tsdata.event`.

`set(tse,propname,propval)` sets the property `propname` of `tse` to the value `propval`. This syntax is equivalent to the command `tse.propname = propval`. You can set multiple properties and their values at a time by using comma-separated pairs. For a full list of `tsdata.event` property names and corresponding valid values, see `tsdata.event`.

Examples

Set `tsdata.event` Property

Create a `tsdata.event` object and set the time units to `'hours'`.

```
tse = tsdata.event('MyEvent',3);
set(tse,'Units','hours');
tseunits = tse.Units
```

```
tseunits =
'hours'
```

Input Arguments

`tse` — Input `tsdata.event`
scalar

Input `tsdata.event`, specified as a scalar.

`propname` — Property name
character vector

Property name, specified as a character vector. For a full list of `tsdata.event` property names, see `tsdata.event`.

`propval` — Property value
scalar

Property value, specified as a scalar. For a full list of `tsdata.event` property names and valid values, see `tsdata.event`.

See Also

`get` | `timeseries` | `tscollection` | `tsdata.event`

Introduced before R2006a

tsearchn

N-D closest simplex search

Syntax

```
t = tsearchn(X,TRI,XI)
[t,P] = tsearchn(X,TRI,XI)
```

Description

`t = tsearchn(X,TRI,XI)` returns the indices `t` of the enclosing simplex of the Delaunay triangulation `TRI` for each point in `XI`. `X` is an `m`-by-`n` matrix, representing `m` points in N-dimensional space. `XI` is a `p`-by-`n` matrix, representing `p` points in N-dimensional space. `tsearchn` returns `NaN` for all points outside the convex hull of `X`. If a point in `XI` lies on the boundary of two or more simplexes, then the largest index is returned. `tsearchn` requires a triangulation `TRI` of the points `X` obtained from `delaunayn`.

`[t,P] = tsearchn(X,TRI,XI)` also returns the barycentric coordinate `P` of `XI` in the simplex `TRI`. `P` is a `p`-by-`n+1` matrix. Each row of `P` is the barycentric coordinate of the corresponding point in `XI`. It is useful for interpolation.

Tips

- The `pointLocation` function can provide better search performance for some 2-D and 3-D triangulations, such as finite element meshes.

See Also

`delaunayTriangulation`

Introduced before R2006a

type

Display contents of file

Syntax

```
type filename
```

Description

`type filename` displays the contents of the specified file in the MATLAB Command Window.

Examples

View File Contents

Display the contents of the file `diceRoll`. Since `diceRoll` does not exist, but `diceRoll.m` does exist, MATLAB displays the contents of `diceRoll.m`.

```
type diceRoll
```

```
function [d1,d2] = diceRoll
str = '@()randi([1 6],1)';
d1 = str2func(str);
d2 = eval(str);
end

function r = randi(~,~)
r = 1;
end
```

Input Arguments

`filename` — File name
character vector | string

File name to display, specified as a character vector or a string. `filename` can be an absolute or relative path and can include a path and a file extension. `type` supports file names with these extensions.

File Extension	Description
`.mlx`	MATLAB Live Script — An interactive document created using MATLAB Live Editor
`.mlapp`	MATLAB App File — An app created using MATLAB App Designer
`.m`	MATLAB Code — A MATLAB script, function, or class

If you do not specify a file extension and a file without an extension does not exist, then `type` assumes that the extension is `.mlx`, `.mlapp`, or `.m`. For example, if a file `myscript.m` exists, then the command `type myscript` displays the contents of that file.

See Also

`dbtype` | `more` | `which`

Introduced in R2006a

typecast

Convert data type without changing underlying data

Syntax

```
Y = typecast(X,type)
```

Description

`Y = typecast(X,type)` converts the bit patterns of `X` to the data type specified by `type` without changing the underlying data. `X` must be a full, noncomplex, numeric scalar or vector.

Examples

Convert Integer to Unsigned Integer of Same Storage Size

Convert an integer to an unsigned integer of the same storage size.

```
X = int16(-1)
```

```
X = int16
    -1
```

```
Y = typecast(X,'uint16')
```

```
Y = uint16
    65535
```

Show the bit patterns in hexadecimal representation. Converting the data type by using `typecast` does not change the underlying data.

```
format hex
X
```

```
X = int16
   ffff
```

```
Y = typecast(X,'uint16')
```

```
Y = uint16
   ffff
```

Convert 8-Bit Integer to Single Precision

Create a 1-by-4 vector of 8-bit integers.

```
X = int8([77 60 43 26])
```

```
X = 1x4 int8 row vector

   77   60   43   26
```

Convert the four 8-bit integers, which use 4 bytes (32 bits) of storage, to a single-precision number that also uses 4 bytes of storage.

```
Y = typecast(X,'single')
```

```
Y = single
    3.5411e-23
```

Show the bit patterns in hexadecimal representation. In hexadecimal notation, 1 byte (8 bits) is represented by two digits. The `typecast` function rearranges the bit patterns without modifying the data.

```
format hex
X
```

```
X = 1x4 int8 row vector

   4d   3c   2b   1a
```

```
Y = typecast(X,'single')
```

```
Y = single
   1a2b3c4d
```

Convert 32-Bit Integer to 8-Bit Integer

Create a 1-by-3 vector of 32-bit unsigned integers.

```
X = uint32([1 255 256])
```

```
X = 1x3 uint32 row vector

     1   255   256
```

Cast X into 8-bit unsigned integers using `typecast`. Each 32-bit value is divided into four 8-bit segments. Running this code on a little-endian system produces the following results.

```
Y = typecast(X,'uint8')
```

```
Y = 1x12 uint8 row vector

     1     0     0     0   255     0     0     0     0     1     0     0
```

The third element of X, 256, exceeds the maximum value that 8 bits can hold. The converted value in Y(9) thus overflows to Y(10).

```
Y(9:12)
```

```
ans = 1x4 uint8 row vector

   0   1   0   0
```

You can convert Y back to 32-bit unsigned integers without changing the underlying data.

```
X2 = typecast(Y,'uint32')
```

```
X2 = 1x3 uint32 row vector
```

```
     1   255   256
```

Compare the output of `typecast` and the output of `cast` to see the difference between the two functions.

```
Z = cast(X,'uint8')
```

```
Z = 1x3 uint8 row vector

     1   255   255
```

```
X2 = cast(Z,'uint32')
```

```
X2 = 1x3 uint32 row vector

     1   255   255
```

Convert 8-Bit Integer to 16-Bit Integer

Casts integers from a smaller data type (`uint8`) into a larger one (`uint16`). Use hexadecimal representation to show the rearrangement of the bit patterns. The `typecast` function returns the output in little-endian style, combining the four 8-bit segments of the input data to produce two 16-bit segments.

```
format hex
X = uint8([44 55 66 77])
```

```
X = 1x4 uint8 row vector

   2c   37   42   4d
```

```
Y = typecast(X,'uint16')
```

```
Y = 1x2 uint16 row vector

   372c   4d42
```

You can convert the little-endian output to big-endian (and vice versa) using the `swapbytes` function.

```
Y = swapbytes(typecast(X,'uint16'))
```

```
Y = 1x2 uint16 row vector

   2c37   424d
```

Input Arguments

X — Input array
scalar | vector

Input array, specified as a scalar or vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

type — New data type
`'single'` | `'double'` | `'int8'` | `'int16'` | `'int32'` | `'int64'` | `'uint8'` | `'uint16'` | `'uint32'` | `'uint64'`

New data type, specified as `'single'`, `'double'`, `'int8'`, `'int16'`, `'int32'`, `'int64'`, `'uint8'`, `'uint16'`, `'uint32'`, or `'uint64'`.

If the bit size of `type` is `n` times larger than the bit size of each element of `X`, then `X` must contain a multiple of `n` elements to convert `X` to the data type `type`. Otherwise, MATLAB throws an error.

Tips

- `typecast` is different from the MATLAB `cast` function in that it does not alter the input data. `typecast` always returns the same number of bytes in the output `Y` as in the input `X`. For example, casting the 16-bit integer 1000 to `uint8` with `typecast` returns the full 16 bits in two 8-bit segments (3 and 232), thus keeping the original value (3*256 + 232 = 1000). The `cast` function, on the other hand, truncates the input value to 255.

- The format of `typecast` output can differ depending on the system you use. Some computer systems store data starting with the least significant byte (an ordering called *little-endian*), while others start with the most significant byte (called *big-endian*). You can use the `swapbytes` function to reverse the byte ordering from little-endian to big-endian (and vice versa).

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The value of the data type argument must be lowercase.
- When you use `typecast` with inherited input port data types in MATLAB Function blocks, the software can throw a size error. To avoid this error, specify the block input port data types explicitly.
- Integer input or result classes must map directly to a C type on the target hardware.
- The input must be a variable-length vector or a fixed-size vector. See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).
- The output vector always has the same orientation as the input vector.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- The value of the data type argument must be lowercase.
- When you use `typecast` with inherited input port data types in MATLAB Function blocks, the software can throw a size error. To avoid this error, specify the block input port data types explicitly.
- Integer input or result classes must map directly to a C type on the target hardware.
- The input must be a variable-length vector or a fixed-size vector.

- The output vector always has the same orientation as the input vector.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cast` | `class` | `swapbytes`

Introduced before R2006a

tzoffset

Time zone offset from UTC

Syntax

```
dt = tzoffset(t)
[dt,dst] = tzoffset(t)
```

Description

`dt = tzoffset(t)` returns an array of durations equal to the time zone offset from UTC of each datetime value in `t`. That is, `dt` is the amount of time that each datetime in `t` differs from UTC. For datetimes that occur during Daylight Saving Time (DST), `dt` includes the time shift for DST. The offset for unzoned datetime values is not defined.

`[dt,dst] = tzoffset(t)` additionally returns the time shift for Daylight Saving Time for each datetime in `t`.

Examples

Find Time Zone Offset

Create a `datetime` array in the time zone for New York City.

```
t1 = datetime('today','TimeZone','America/New_York');
t = dateshift(t1,'end','month',[1:3:9])
```

```
t = 1x3 datetime array
   30-Sep-2019   31-Dec-2019   31-Mar-2020
```

Find the time zone offset from UTC and the time shift for Daylight Saving Time for each datetime value.

```
[dt,dst] = tzoffset(t)
```

```
dt = 1x3 duration array
   -04:00   -05:00   -04:00


dst = 1x3 duration array
   01:00   00:00   01:00
```

Input Arguments

t — Input date and time
`datetime` array

Input date and time, specified as a `datetime` array.

Output Arguments

dt — Time zone offset from UTC
scalar | vector | matrix | multidimensional array

Time zone offset from UTC, returned as a scalar, vector, matrix, or multidimensional `duration` array. `dt` is the same size as `t`.

dst — Time shift for Daylight Saving Time
scalar | vector | matrix | multidimensional array

Time shift for Daylight Saving Time, returned as a scalar, vector, matrix, or multidimensional `duration` array. `dt` is the same size as `t`. A value of `01:00` indicates that the corresponding datetime in `t` occurs during Daylight Saving Time in a location that observes it. For locations that do not observe Daylight Saving Time, the elements of the `dst` array are all `00:00`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`isdst`

Introduced in R2014b

uibuttongroup

Create button group to manage radio buttons and toggle buttons

Syntax

```
bg = uibuttongroup
bg = uibuttongroup(Name,Value)
bg = uibuttongroup(parent)
bg = uibuttongroup(parent,Name,Value)
```

Description

`bg = uibuttongroup` creates a button group in the current figure and returns the `ButtonGroup` object. If there is no figure available, MATLAB calls the `figure` function to create one.

`bg = uibuttongroup(Name,Value)` specifies button group property values using one or more name-value pair arguments.

`bg = uibuttongroup(parent)` creates the button group in the specified parent container. The parent container can be a figure created with either the `figure` or `uifigure` function, or a child container such as a panel. Property values for `uibuttongroup` vary slightly depending on whether the app is created with the `figure` or `uifigure` function. For more information, see “Name-Value Pair Arguments” on page 1-16006.

`bg = uibuttongroup(parent,Name,Value)` specifies the parent container and one or more properties values.

Examples

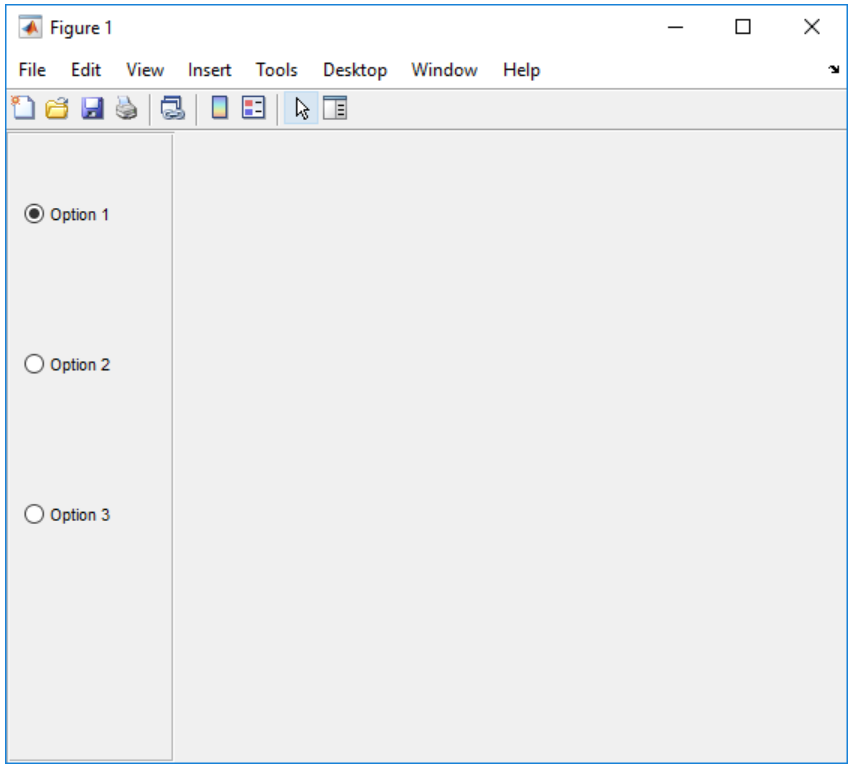
Code Response to Radio Button Selection

Show the previous and current button selections in the MATLAB Command Window when the user selects a different radio button in a button group.

Copy and paste this code into the Editor and run it to see how it works.

```
function myui
bg = uibuttongroup('Visible','off',...
                   'Position',[0 0 .2 1],...
                   'SelectionChangedFcn',@bselection);

% Create three radio buttons in the button group.
r1 = uicontrol(bg,'Style',...
                   'radiobutton',...
                   'String','Option 1',...
                   'Position',[10 350 100 30],...
                   'HandleVisibility','off');

r2 = uicontrol(bg,'Style','radiobutton',...
                   'String','Option 2',...
                   'Position',[10 250 100 30],...
                   'HandleVisibility','off');

r3 = uicontrol(bg,'Style','radiobutton',...
                   'String','Option 3',...
                   'Position',[10 150 100 30],...
                   'HandleVisibility','off');

% Make the uibuttongroup visible after creating child objects.
bg.Visible = 'on';

    function bselection(source,event)
       disp(['Previous: ' event.OldValue.String]);
       disp(['Current: ' event.NewValue.String]);
       disp('------------------');
    end
end
```

The `bselection` function displays the `OldValue` and `NewValue` properties of `event`.



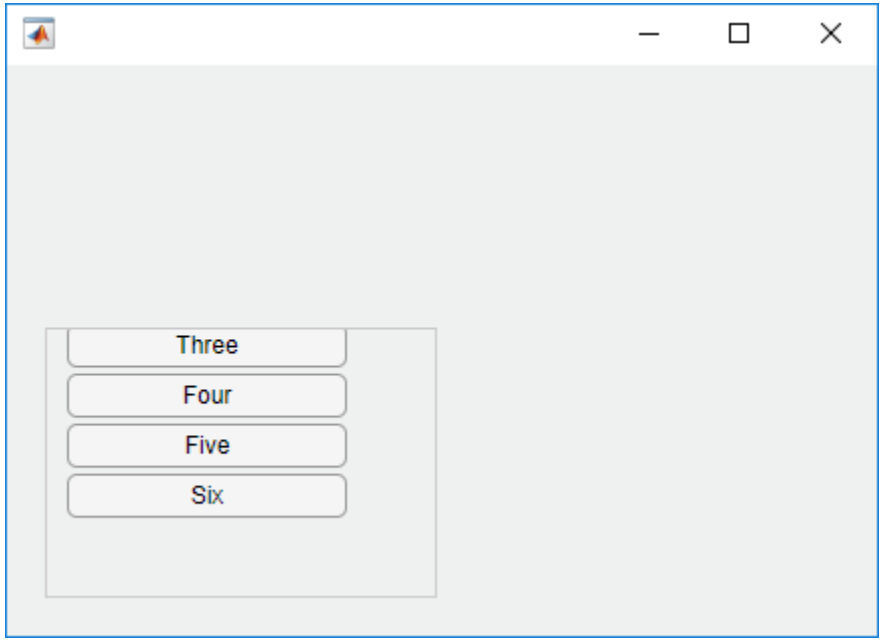
Scrollable Button Group

The `Scrollable` property enables scrolling within a button group that has components outside its borders. Scrolling is only possible when the button group is in a figure created with the `uifigure` function. App Designer uses this type of figure for creating apps.

Create a button group within a figure. Add six toggle buttons, with the first three lying outside the upper border of the button group.

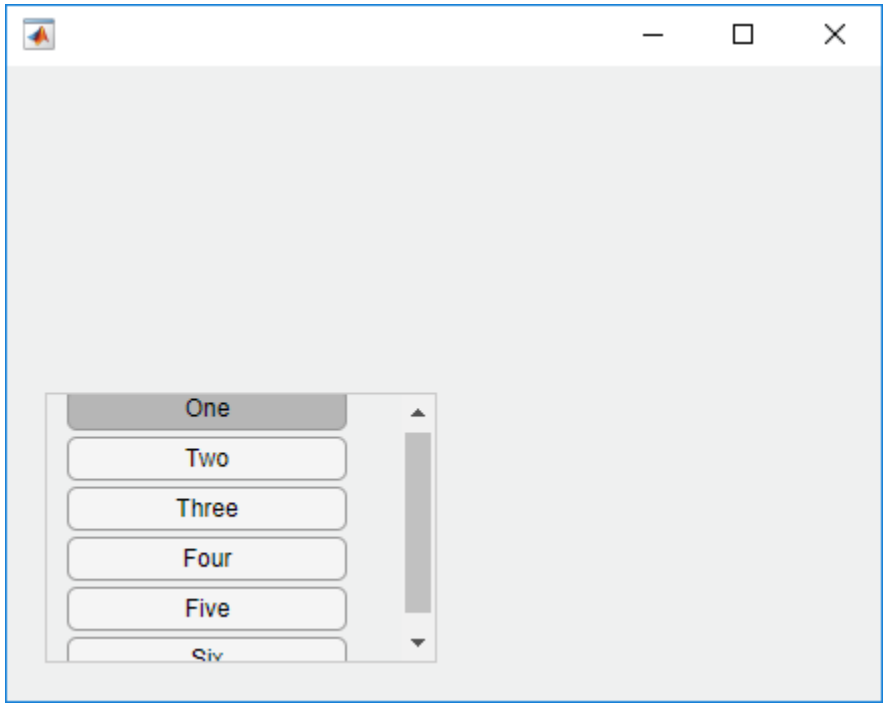
```
f = uifigure;
bg = uibuttongroup(f,'Position',[20 20 196 135]);
tb1 = uitogglebutton(bg,'Position',[11 165 140 22],'Text','One');
tb2 = uitogglebutton(bg,'Position',[11 140 140 22],'Text','Two');
tb3 = uitogglebutton(bg,'Position',[11 115 140 22],'Text','Three');
tb4 = uitogglebutton(bg,'Position',[11 90 140 22],'Text','Four');
```

```
tb5 = uitogglebutton(bg,'Position',[11 65 140 22],'Text','Five');
tb6 = uitogglebutton(bg,'Position',[11 40 140 22],'Text','Six');
```



Enable scrolling by setting the `Scrollable` property of the button group to `'on'`. By default, the scroll box displays at the top.

```
bg.Scrollable = 'on';
```



Input Arguments

parent — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a figure created with either the `figure` or `uifigure` function, or a child container:

- Panels, tabs and button groups can be containers in either type of figure.
- Grid layouts can be containers only in figures created with the `uifigure` function.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (`' '`). You can specify several name and value pair arguments as `Name1,Value1,...,NameN,ValueN`.

Example: `'Title','Options'` specifies that the button group title is `Options`.

- For a list of properties available for GUIDE, or for creating apps with the `figure` function, see Uibuttongroup for GUIDE.
- For a list of properties available for App Designer, or for creating apps with the `uifigure` function, see ButtonGroup for App Designer.

Tips

- A button group can contain any UI component type, but it only manages the selection of radio buttons and toggle buttons.
- To make your program respond when the app user selects a radio button or toggle button that is inside a button group. define a `SelectionChangedFcn` callback function for the button group. You cannot define callbacks for the individual buttons.
- To determine which radio button or toggle button is selected, query the `SelectedObject` property of the button group. You can execute this query anywhere in your code.
- If you set the `Visible` property of a button group object to `'off'`, then any child objects it contains (buttons, other button groups, etc.) become invisible along with the parent button group. However, the `Visible` *property value* of each child object remains unaffected.

See Also

Functions
`uicontrol` | `uipanel` | `uiradiobutton` | `uitogglebutton`

Properties
ButtonGroup

Introduced before R2006a

ButtonGroup Properties

Control appearance and behavior of button group

Note The properties listed here are valid for button groups in GUIDE or in apps created with the `figure` function. If you are using App Designer or the `uifigure` function, see ButtonGroup Properties instead. For more information, see "GUIDE Migration Strategies".

Description

Button groups are containers for managing exclusive selection of radio buttons and toggle buttons. Properties control the appearance and behavior of a button group. Use dot notation to refer to a specific object and property.

```
f = figure;
b = uibuttongroup(f);
b.BackgroundColor = [.5 .5 .5];
```

Properties

Title

`Title` — Title

character vector | string scalar | categorical array

Title, specified as a character vector, string scalar, or categorical array. If you specify this property as a categorical array, MATLAB displays only the first element in the array.

MATLAB does not interpret a vertical slash (`'|'`) character as a line break, it displays as a vertical slash in the title.

If you want to specify a Unicode character, pass the Unicode decimal code to the `char` function. For example, `['Multiples of ' char(960)]` displays as `Multiples of` π.

`TitlePosition` — Title location

`'lefttop'` (default) | `'centertop'` | `'righttop'` | `'leftbottom'` | `'centerbottom'` | `'rightbottom'`

Title location, specified as `'lefttop'`, `'centertop'`, `'righttop'`, `'leftbottom'`, `'centerbottom'`, or `'rightbottom'`.

Color and Styling

`ForegroundColor` — Title color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Title color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `[0 0 1]`

Example: `'b'`

Example: `'blue'`

`BackgroundColor` — Background color

`[.94 .94 .94]` (default) | RGB Triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

BorderType — Border type

'etchedin' (default) | 'etchedout' | 'beveledin' | 'beveledout' | 'line' | 'none'

Border type, specified as 'etchedin', 'none', 'etchedout', 'beveledin', 'beveledout', or 'line'.

- For a 3-D appearance, use etched or beveled borders.

  Use the HighlightColor and ShadowColor properties to specify the color of 3-D borders.

- For a simpler appearance, use a line border.

  Use the `HighlightColor` property to specify the line border color.

`BorderWidth` — Border width
1 (default) | positive integer value

Border width, specified as a positive integer value. The unit of measurement is pixels. Etched and beveled borders wider than three pixels might not appear correctly at the corners.

`HighlightColor` — Border highlight color
RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Border highlight color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`ShadowColor` — Border shadow color

RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Border shadow color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`Clipping` — Child component clipping (not recommended)
`'on'` (default) | `'off'`

Note The behavior of the `Clipping` property has changed. It no longer has any effect on `ButtonGroup` objects. Child objects are now clipped to the boundaries of their parent container regardless of the value of this property. This property might be removed in a future release.

Font

`FontName` — Font name

system supported font name | `'FixedWidth'`

Font name, specified as a system supported font name or `'FixedWidth'`. The default font depends on the specific operating system and locale.

To use a fixed-width font that looks good in any locale, specify `'FixedWidth'`. The actual fixed-width font used depends on the `FixedWidthFontName` property of the root object. Changing the `FixedWidthFontName` property causes an immediate update of the display to use the new font.

Example: `'Arial'`

`FontSize` — Font size

positive number

Font size, specified as a positive number. The `FontUnits` property specifies the units. The default size is system-dependent.

Example: `12`

Example: `12.5`

`FontWeight` — Font weight

`'normal'` (default) | `'bold'`

Font weight, specified as a value from the following table.

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than normal

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold font weight. Therefore, specifying a bold font weight still can result in the normal font weight.

Note The `'light'` and `'demi'` font weight values have been removed in R2014b. If you specify either of these values, the result is a normal font weight.

`FontAngle` — Font angle

`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. MATLAB uses this property to select a font from those available on your system. Setting this property to `'italic'` selects a slanted version of the font, if it is available on your system.

Note The `'oblique'` value has been removed. Use `'italic'` instead.

FontUnits — Font units

`'points'` (default) | `'normalized'` | `'inches'` | `'centimeters'` | `'pixels'`

Font units, specified as one of the values from this table.

Units Value	Description
`'points'`	Points. One point is 1/72nd of an inch.
`'normalized'`	Normalized values for specifying the font size as a fraction of the height. When you resize a UI component, MATLAB scales the displayed font to maintain that fraction.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.

Interactivity

Visible — Button group visibility

`'on'` (default) | `'off'`

Button group visibility, specified as `'on'` or `'off'`. The `Visible` property determines whether the button group displays on the screen. If the `Visible` property is set to

`'off'`, the entire button group is invisible, but you can still specify and access its properties.

Changing the size of an invisible container triggers the `SizeChangedFcn` callback when it becomes visible.

Note Changing the `Visible` property of a container does *not* change the `Visible` property of its child components even though hiding the container prevents its children from displaying.

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | uicontextmenu object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when the user right-clicks on a UI component. Create the context menu using the `uicontextmenu` function.

`SelectedObject` — Currently selected radio button or toggle button

`UIControl` object | `[]`

Currently selected radio button or toggle button, specified as a `UIControl` object.

Use this property to determine the currently selected button within a button group. You can also use this property to set a default button selection. If you want no selection, then set this property to `[]`.

The default value of the `SelectedObject` property is the first radio button or toggle button that you add to the button group.

Note The `SelectionChangedFcn` callback does not execute when you set the `SelectedObject` property programmatically.

`Selected` — Selection state (not recommended)

`'off'` (default) | `'on'`

Note The behavior of the `Selected` property changed in R2014b, and it is not recommended. It no longer has any effect on `ButtonGroup` objects. This property might be removed in a future release.

SelectionHighlight — Display of selection handles (not recommended)

'on' (default) | 'off'

Note The behavior of the SelectionHighlight property changed in R2014b, and it is not recommended. It no longer has any effect on ButtonGroup objects. This property might be removed in a future release.

Position

Position — Location and size (including borders and title)

[left bottom width height]

Location and size (including borders and title), specified as a four-element vector of the form [left bottom width height]. This table describes each element in the vector.

Element	Description
left	Distance from the inner left edge of the parent container to the outer left edge of the button group
bottom	Distance from the inner bottom edge of the parent container to the outer bottom edge of the button group
width	Distance between the right and left outer edges of the button group
height	Distance between the top and bottom outer edges of the button group

All measurements are in units specified by the Units property.

Note The Position values are relative to the parent container's drawable area. The drawable area is the area inside the borders of the container and does not include the area occupied by the title. If the parent container is a figure, then the drawable area also excludes the menu bar and tool bar.

Example: Modify One Value in the Position Vector

You can combine dot notation and array indexing when you want to change one value in the Position vector. For example, this code sets the width to 0.5:

```
b = uibuttongroup;
b.Position(3) = 0.5;
b.Position

ans =

      0   0   0.5000  1.0000
```

`InnerPosition` — Location and size (excluding borders and title)
`[left bottom width height]`

This property is read-only.

Location and size (excluding borders and title), returned as a four-element vector of the form `[left bottom width height]`. This table describes each element in the vector.

Value	Description
`left`	Distance from the inner left edge of the parent container to the inner left edge of the button group.
`bottom`	Distance from the inner bottom edge of the parent container to the inner bottom edge of the button group.
`width`	Distance between the inner edges of the button group's right and left borders.
`height`	Distance between the inner edges of the button group's top and bottom borders. This distance excludes the title, if it exists.

All measurements are in units specified by the `Units` property.

Note These are some important points to consider when using the `InnerPosition` property:

- `InnerPosition` values are affected by the presence of a title, font characteristics, `BorderType`, and `BorderWidth`.
- `InnerPosition` values are relative to the parent container’s drawable area. The drawable area is the area inside the borders of the container and exclude the area occupied by the title. If the parent container is a figure, then the drawable area also excludes the menu bar and tool bar.

`OuterPosition` — Location and size (including borders and title)
`[left bottom width height]`

Location and size (including borders and title), specified as a four-element vector of the form `[left bottom width height]`. All measurements are in units specified by the `Units` property.

This property value is identical to the `Position` property value.

`Units` — Units of measurement
`'normalized'` (default) | `'pixels'` | `'inches'` | `'centimeters'` | `'points'` | `'characters'`

Units of measurement, specified one of the values from this table.

Units Value	Description
`'normalized'`	These units are normalized with respect to the parent container. The lower-left corner of the container maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'points'`	Points. One point equals 1/72nd of an inch.

Units Value	Description
`'characters'`	These units are based on the default uicontrol font of the graphics root object: • Character width = width of the letter `x`. • Character height = distance between the baselines of two lines of text. To access the default uicontrol font, use `get(groot,'defaultuicontrolFontName')` or `set(groot,'defaultuicontrolFontName')`.

MATLAB measures all units from the lower left corner of the parent object.

If you change the value of the `Units` property, it is good practice to return it to its default value after completing your computation to avoid affecting other functions that assume the `Units` property is set to the default value.

The order in which you specify the Units and Position properties has these effects:

- If you specify the `Units` property before the `Position` property, then MATLAB sets `Position` using the units you specified.
- If you specify the `Units` property after the `Position` property, MATLAB sets the position using the default `Units`. Then, MATLAB converts the `Position` values to the equivalent values in the units you specified.

Callbacks

`SelectionChangedFcn` — Selection changed callback

`''` (default) | function handle | cell array | character vector

---

Note The name of this property changed from `SelectionChangeFcn` to `SelectionChangedFcn` in R2014b.

Selection changed callback, specified as one of these values:

- A function handle.

- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

This callback function executes when the user selects a different button within the button group.

If you specify this property as a function handle (or cell array containing a function handle), MATLAB passes an object containing callback data as the second argument to the callback function. This object contains the properties described in this table. You can access these properties inside the callback function using dot notation.

Property	Description
`OldValue`	Previously selected button, or `[]` if none was selected
`NewValue`	Currently selected button
`Source`	The parent `ButtonGroup` object
`EventName`	`'SelectionChanged'`

Define a `SelectionChangedFcn` callback to make your program respond when the user selects different buttons within the button group. Do not code the response in the individual component callback functions.

If you want another component to respond to the selection, then that component's callback function can access the selected radio button or toggle button from the `SelectedObject` property of the `ButtonGroup` object.

`SizeChangedFcn` — Size change callback function
`''` (default) | function handle | cell array | character vector

Size change callback function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

Define this callback to customize the app layout when the size of this container changes (e.g., when the user resizes the window).

The `SizeChangedFcn` callback executes when:

- This container becomes visible for the first time.
- This container is visible while its size changes.
- This container becomes visible for the first time after its size changes. This situation occurs when the size changes while the container is invisible, and then it becomes visible later.

Other important points to consider when defining a `SizeChangedFcn` callback:

- Consider delaying the display of this container until after all the variables that the `SizeChangedFcn` uses are defined. This practice can prevent the `SizeChangedFcn` callback from returning an error. To delay the display of the container, set its `Visible` property to `'off'`. Then, set the `Visible` property to `'on'` after you define the variables that your `SizeChangedFcn` callback uses.
- If your app contains nested containers, they resize from the inside out.
- To access the container that is resizing from within the `SizeChangedFcn`, refer to the source object (the first input argument in the callback) or use the `gcbo` function.

Tip As an easy alternative to specifying a `SizeChangedFcn` callback, you can set the `Units` property of all the objects you put inside a container to `'normalized'`. Doing so makes those components scale proportionally with the container.

See “Lay Out a UI Programmatically” for more information about managing layouts with `SizeChangedFcn` callbacks.

`ButtonDownFcn` — Button-press callback function

`''` (default) | function handle | cell array | character vector

Button-press callback function, specified as one of these values:

- A function handle.

- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

The `ButtonDownFcn` callback is a function that executes when the user clicks a mouse button within the container.

`CreateFcn` — Component creation function

`''` (default) | function handle | cell array | character vector

Component creation function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

This property specifies a callback function to execute when MATLAB creates the component. MATLAB initializes all component property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Use the `gcbo` function in your `CreateFcn` code to get the component object that is being created.

Setting the `CreateFcn` property on an existing component object has no effect.

`DeleteFcn` — Component deletion function

`''` (default) | function handle | cell array | character vector

Component deletion function, specified as one of these values:

- A function handle.

- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

The `DeleteFcn` property specifies a callback function to execute when MATLAB deletes the component (for example, when the user closes the window). MATLAB executes the `DeleteFcn` callback before destroying the properties of the component object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

Use the `gcbo` function in your `DeleteFcn` code to get the component object that is being deleted.

`ResizeFcn` — Resize callback function (not recommended)

`''` (default) | function handle | cell array | character vector

Resize callback function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

Note Use of the `ResizeFcn` property is not recommended. It might be removed in a future release. Use `SizeChangedFcn` instead.

Data Types: `function_handle` | `cell` | `char`

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed:

- A value of `'on'` allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- A value of `'off'` blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn`, or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.
- MATLAB does not save the state of properties or the display when an interruption occurs. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

See “Interrupt Callback Execution” for an example that shows how the `Interruptible` and `BusyAction` properties affect the behavior of a program.

BusyAction — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` (default) or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

The `BusyAction` property of the source of the interrupting callback determines how MATLAB handles its execution. The `BusyAction` property has these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Do not execute the interrupting callback.

Whenever MATLAB invokes a callback, that callback always attempts to interrupt an executing callback. The `Interruptible` property of the object whose callback is running determines if interruption is allowed. If `Interruptible` is set to:

- `on` — Interruption occurs at the next point where MATLAB processes the queue. This is the default.
- `off` — The `BusyAction` property (of the object owning the interrupting callback) determines if MATLAB enqueues or ignores the interrupting callback.

See “Interrupt Callback Execution” for an example that shows how the `BusyAction` and `Interruptible` properties affect the behavior of a program.

BeingDeleted — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

HitTest — Ability to become current object

'on' (default) | 'off'

Ability to become current object, specified as 'on' or 'off':

- 'on' — Sets the current object to the ButtonGroup when the user clicks the component in the running app. Both the CurrentObject property of the Figure and the gco function return the ButtonGroup as the current object.
- 'off' — Sets the current object to be the closest ancestor of the ButtonGroup whose HitTest is 'on' when the user clicks the component in the running app.

Parent/Child

Parent — Parent object

Figure | Panel | ButtonGroup | Tab

Parent object, specified as a Figure, Panel, ButtonGroup, or Tab object. Use this property to specify the parent container when creating a UI component or to move an existing UI component to a different parent container.

Children — ButtonGroup children

empty GraphicsPlaceholder array (default) | 1-D array of component objects

ButtonGroup children, returned as an empty GraphicsPlaceholder or a 1-D array of component objects. Although a button group manages only the selection of radio buttons and toggle buttons, its children can be instances of Axes, Panel, ButtonGroup, or any style of UIControl objects.

You cannot add or remove children using the Children property. Use this property to view the list of children or to reorder the children. The order of the children reflects the front-to-back order (stacking order) of the components on the screen. MATLAB might not allow you to change the order of certain objects. For example, UIControl and Legend objects are always in front of Axes objects.

To add a child to this list, set the Parent property of the child component to be the ButtonGroup object.

Objects with the HandleVisibility property set to 'off' do not list in the Children property.

HandleVisibility — Visibility of ButtonGroup handle

'on' (default) | 'callback' | 'off'

Visibility of `ButtonGroup` handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the `ButtonGroup` handle in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. The `HandleVisibility` property also controls the visibility of the object's handle in the parent figure's `CurrentObject` property. Handles are still valid even if they are not visible. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

HandleVisibility Value	Description
`'on'`	The `ButtonGroup` handle is always visible.
`'callback'`	The `ButtonGroup` handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the `ButtonGroup` at the command-line, but allows callback functions to access it.
`'off'`	The `ButtonGroup` handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.

Set the graphics root `ShowHiddenHandles` property to `'on'` to make all handles visible, regardless of their `HandleVisibility` value. This setting has no effect on their `HandleVisibility` values.

Note Do not try to access radio buttons and toggle buttons that are managed by a `uibuttongroup` outside of the button group. Set the `HandleVisibility` of those radio buttons and toggle buttons to `'off'` to prevent accidental access.

Identifiers

Type — Type of graphics object
`'uibuttongroup'`

This property is read-only.

Type of graphics object, returned as `'uibuttongroup'`.

`Tag` — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any array. Specifying `UserData` can be useful for sharing data within apps. See “Share Data Among Callbacks” for more information.

See Also

`guide` | `uibuttongroup` | `uicontrol`

Topics

“Access Property Values”
“Default Property Values”

Introduced before R2006a

uicontextmenu

Create context menu

Note

Use this function only with GUIDE, or with figures created using the `figure` function.

Syntax

```
c = uicontextmenu
c = uicontextmenu(Name,Value)
```

```
c = uicontextmenu(parent)
c = uicontextmenu(parent,Name,Value)
```

Description

`c = uicontextmenu` creates a context menu in the current figure and returns the `ContextMenu` object as `c`. If a figure does not exist, then MATLAB calls the `figure` function to create one.

`c = uicontextmenu(Name,Value)` creates a context menu with property values specified using one or more name-value pair arguments. Use this syntax to override the default property values.

`c = uicontextmenu(parent)` creates a context menu in the specified parent figure.

Note Specifying a parent alone does not make the context menu accessible in the UI. You must also take these steps:

- Assign the context menu to a component using the `UIContextMenu` property.
- Create at least one child `uimenu` for the `uicontextmenu`.

See the “Tips” on page 1-16038 section for more information.

`c = uicontextmenu(parent,Name,Value)` specifies a parent figure for the context menu and one or more name-value pair arguments.

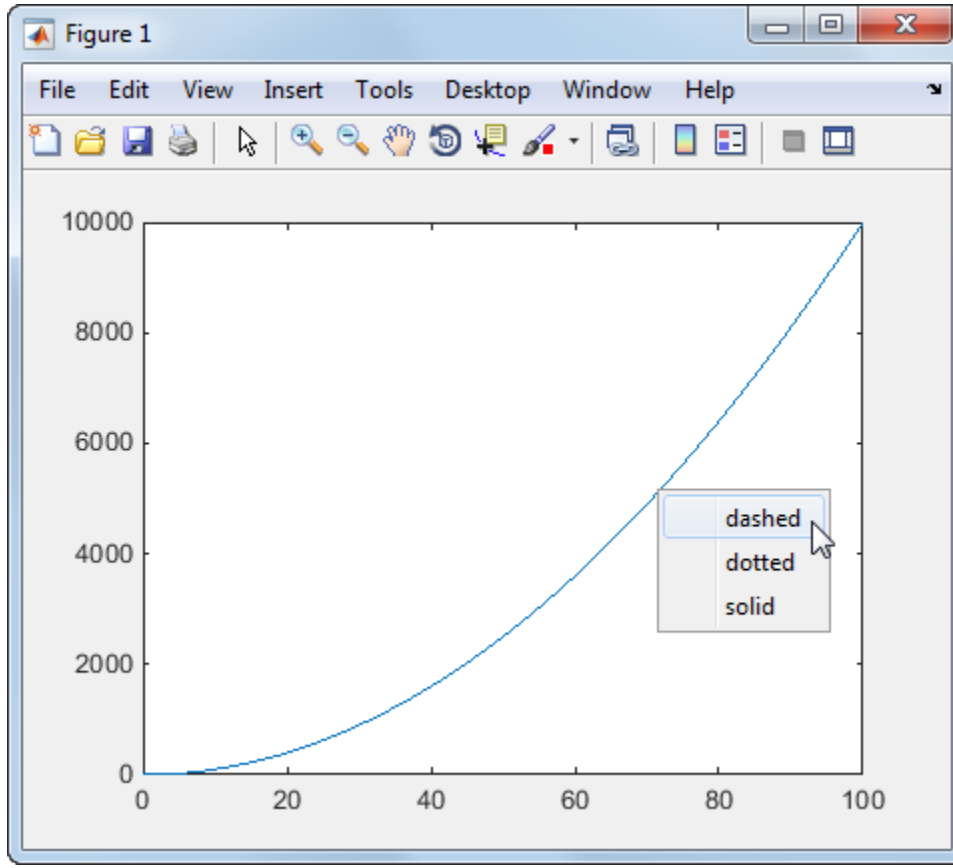
Examples

Context Menu Attached to a Plot Line

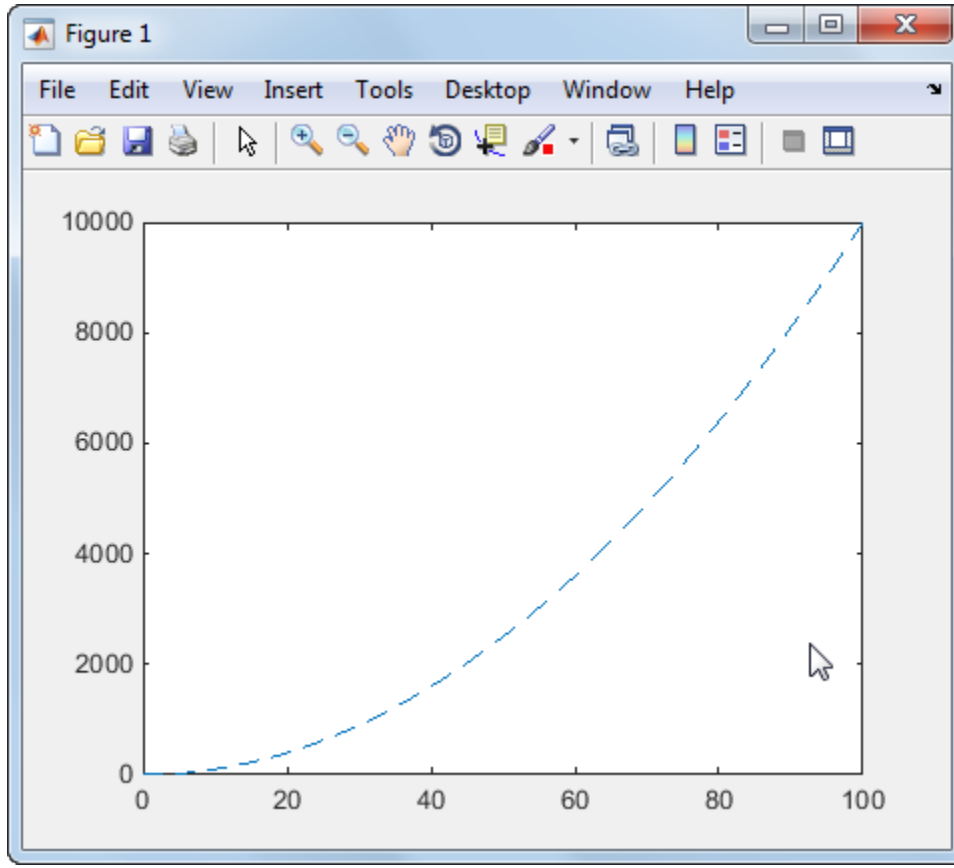
Specify a value for the line object's `UIContextMenu` property to attach a `uicontextmenu` to that line. The context menu becomes visible when the user right-clicks on the line. For instance, create a program file called `myprogram.m` that creates a plot and attaches a `uicontextmenu` to the plot line:

```
function myprogram

    f = figure('WindowStyle','normal');
    ax = axes;
    x = 0:100;
    y = x.^2;

    plotline = plot(x,y);
    c = uicontextmenu;

    % Assign the uicontextmenu to the plot line
    plotline.UIContextMenu = c;

    % Create child menu items for the uicontextmenu
    m1 = uimenu(c,'Label','dashed','Callback',@setlinestyle);
    m2 = uimenu(c,'Label','dotted','Callback',@setlinestyle);
    m3 = uimenu(c,'Label','solid','Callback',@setlinestyle);

        function setlinestyle(source,callbackdata)
            switch source.Label
                case 'dashed'
                    plotline.LineStyle = '--';
                case 'dotted'
                    plotline.LineStyle = ':';
                case 'solid'
                    plotline.LineStyle = '-';
            end
        end
end
```

The context menu appears when the user right-clicks the plot line.



Selecting an item from the context menu changes the line style.



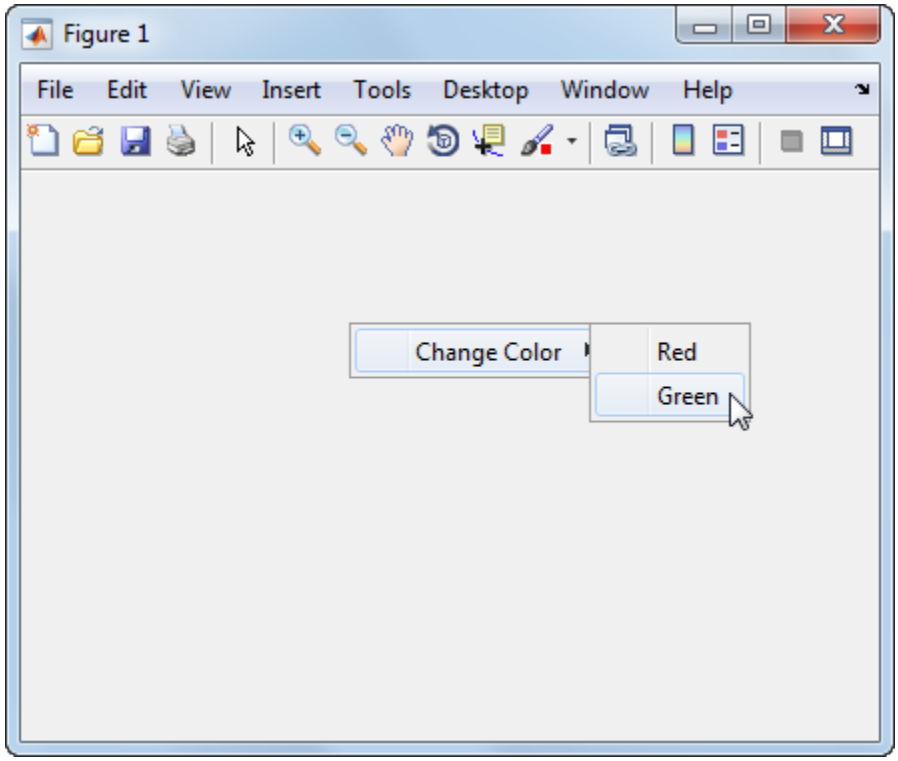
Context Menu with Submenus

Specify the `Parent` property value of any `uimenu` to make it into a submenu. For instance, create a program file called `myprogram2` that creates a context menu containing one top-level menu and two submenu items:

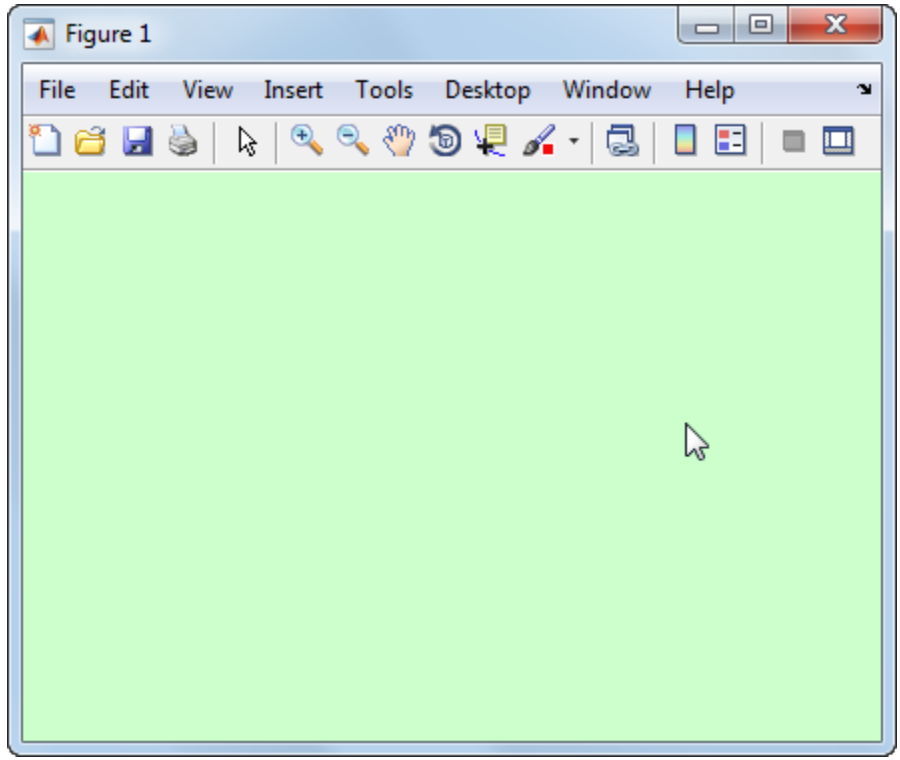
```
function myprogram2
    f = figure('WindowStyle','normal');
    c = uicontextmenu(f);

    % Assign the uicontextmenu to the figure
```

```
    f.UIContextMenu = c;

    % Create child menu of the uicontextmenu
    topmenu = uimenu('Parent',c,'Label','Change Color');

    % Create submenu items
    m1 = uimenu('Parent',topmenu,'Label','Red','Callback',@changecolor);
    m2 = uimenu('Parent',topmenu,'Label','Green','Callback',@changecolor);

    function changecolor(source,callbackdata)
        switch source.Label
            case 'Red'
                f.Color = [1.0 0.80 0.80];
            case 'Green'
                f.Color = [0.80 1.0 0.80];
        end
    end
end
```

The resulting context menu appears when the user right-clicks the mouse inside the figure.



Selecting a color from the context menu changes the color of the window.



Input Arguments

`parent` — Parent figure
`Figure` object

Parent figure, specified as a `Figure` object created with the `figure`. If a parent figure is not specified, then MATLAB calls the `figure` function to create one that serves as the parent.

Name-Value Pair Arguments

The properties listed here are only a subset, for a complete list see Uicontextmenu.

Example: `'Callback',@myfunction` specifies `myfunction` to be the function that executes when the user interacts with the context menu.

`Callback` — Context menu callback function
`''` (default) | function handle | cell array | character vector

Context menu callback function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

Data Types: `function_handle` | `cell` | `char`

`Children` — Menu children
empty `GraphicsPlaceholder` array (default) | 1-D array of `Menu` objects

Menu children, returned as an empty `GraphicsPlaceholder` or a 1-D array of `Menu` objects. The children of menus are other menus that function as submenus.

You cannot add or remove children using the `Children` property. Use this property to view the list of children or to reorder the child menu items. The order of the children in this array reflects the order of the displayed menu items.

To add a child to this list, set the `Parent` property of the child component to be the `uicontextmenu` object.

Objects with the `HandleVisibility` property set to `'off'` do not list in the `Children` property.

Tips

The context menu becomes accessible in the UI when these requirements are met:

- You assign the context menu to a component using the `UIContextMenu` property. The assigned component must be a child of the same figure as the context menu.
- You create at least one child `uimenu` for the `uicontextmenu`.

For example:

```
f = figure;
c = uicontextmenu(f);
% Create a new component and assign the uicontextmenu to it
b = uicontrol(f,'UIContextMenu',c);
% Create a child menu for the uicontextmenu
m = uimenu('Parent',c,'Label','Disable');
```

See Also

Functions
`uibuttongroup` | `uicontrol` | `uimenu` | `uipanel`

Properties
Uicontextmenu

Introduced before R2006a

ContextMenu Properties

Control appearance and behavior of context menu

Note

The properties listed here are valid for `ContextMenu` objects in GUIDE or in apps created with the `figure` function.

Description

A context menu is a menu that appears when the user right-clicks on a graphics object or UI component. The `uicontextmenu` function creates a context menu and sets any required properties. By changing property values, you can modify the appearance and behavior of a context menu. Use dot notation to refer to a specific object and property.

```
c = uicontextmenu;
c.Position = [20 20];
```

Properties

Interactivity

`Visible` — Context menu visibility

`'off'` (default) | `'on'`

Context menu visibility, specified as `'on'` or `'off'`. You can use the `Visible` property in one of two ways:

- Query the value to find out if the context menu is currently posted. The context menu is posted while the `Visible` property value is set to `'on'`.
- Set the `Visible` property to `'on'` to make the context menu to appear. Set this property to `'off'` to make the context menu to disappear.

Position

`Position` — Context menu location

`[left bottom]`

Content menu location, specified as the vector, `[left bottom]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the figure to the left edge of the context menu
`bottom`	Distance from the inner bottom edge of the figure to the top edge of the context menu

The `Position` property defines the location of a context menu when its `Visible` property is set to `'on'`.

Callbacks

`Callback` — Context menu callback function

`''` (default) | function handle | cell array | character vector

Context menu callback function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

Data Types: `function_handle` | `cell` | `char`

`CreateFcn` — Component creation function

`''` (default) | function handle | cell array | character vector

Component creation function, specified as one of these values:

- A function handle.

- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

This property specifies a callback function to execute when MATLAB creates the component. MATLAB initializes all component property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Use the `gcbo` function in your `CreateFcn` code to get the component object that is being created.

Setting the `CreateFcn` property on an existing component object has no effect.

`DeleteFcn` — Component deletion function

`''` (default) | function handle | cell array | character vector

Component deletion function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

The `DeleteFcn` property specifies a callback function to execute when MATLAB deletes the component (for example, when the user closes the window). MATLAB executes the `DeleteFcn` callback before destroying the properties of the component object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

Use the `gcbo` function in your `DeleteFcn` code to get the component object that is being deleted.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed:

- A value of `'on'` allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- A value of `'off'` blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn`, or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

- MATLAB does not save the state of properties or the display when an interruption occurs. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

See “Interrupt Callback Execution” for an example that shows how the `Interruptible` and `BusyAction` properties affect the behavior of a program.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` (default) or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

The `BusyAction` property of the source of the interrupting callback determines how MATLAB handles its execution. The `BusyAction` property has these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Do not execute the interrupting callback.

Whenever MATLAB invokes a callback, that callback always attempts to interrupt an executing callback. The `Interruptible` property of the object whose callback is running determines if interruption is allowed. If `Interruptible` is set to:

- `on` — Interruption occurs at the next point where MATLAB processes the queue. This is the default.
- `off` — The `BusyAction` property (of the object owning the interrupting callback) determines if MATLAB enqueues or ignores the interrupting callback.

See “Interrupt Callback Execution” for an example that shows how the `BusyAction` and `Interruptible` properties affect the behavior of a program.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent object

figure

Parent object, specified as a figure. You can move a `ContextMenu` to a different figure by setting this property to the target `Figure` object.

`Children` — Menu children

empty `GraphicsPlaceholder` array (default) | 1-D array of `Menu` objects

Menu children, returned as an empty `GraphicsPlaceholder` or a 1-D array of `Menu` objects. The children of menus are other menus that function as submenus.

You cannot add or remove children using the `Children` property. Use this property to view the list of children or to reorder the child menu items. The order of the children in this array reflects the order of the displayed menu items.

To add a child to this list, set the `Parent` property of the child component to be the `ContextMenu` object.

Objects with the `HandleVisibility` property set to `'off'` do not list in the `Children` property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object handle in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. The `HandleVisibility` property also controls the visibility of the object’s handle in the parent figure's `CurrentObject` property. Handles are still valid even if they are not

visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object handle is always visible.
`'callback'`	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.

Set the graphics root `ShowHiddenHandles` property to `'on'` to make all handles visible, regardless of their `HandleVisibility` value. This setting has no effect on their `HandleVisibility` values.

Identifiers

Type — Type of graphics object
`'uicontextmenu'`

This property is read-only.

Type of graphics object, returned as `'uicontextmenu'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any array. Specifying `UserData` can be useful for sharing data within apps. See “Share Data Among Callbacks” for more information.

See Also

`uicontextmenu`

Topics

“Access Property Values”
“Default Property Values”

Introduced before R2006a

uicontrol

Create user interface control

Note Use this function only with GUIDE, or with apps created using the `figure` function. App Designer is the recommended environment for building apps. For more information, see “GUIDE Migration Strategies”.

Syntax

```
c = uicontrol
c = uicontrol(Name,Value)
c = uicontrol(parent)
c = uicontrol(parent,Name,Value)

uicontrol(c)
```

Description

`c = uicontrol` creates a push button (the default user interface control) in the current figure, and returns the `UIControl` object. If a figure does not exist, then MATLAB calls the `figure` function to create one.

`c = uicontrol(Name,Value)` creates a user interface control with property values specified using one or more name-value pair arguments. For example, `'Style','checkbox'` creates a check box.

`c = uicontrol(parent)` creates the default user interface control in the specified parent, instead of defaulting to the current figure.

`c = uicontrol(parent,Name,Value)` specifies the parent for the user interface control and one or more name-value pair arguments.

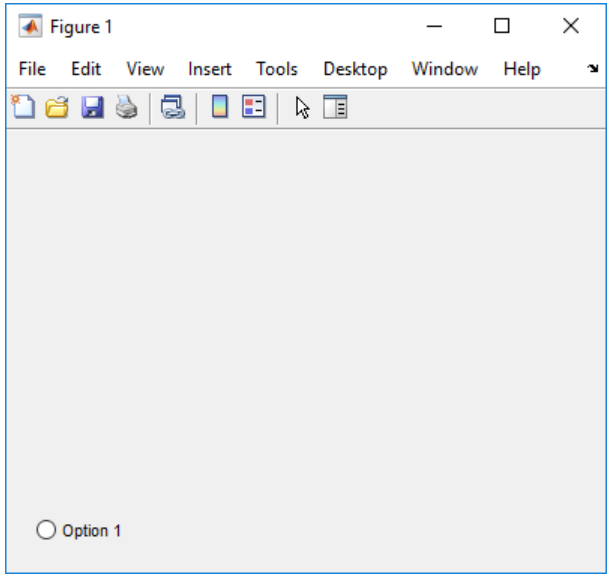
`uicontrol(c)` gives focus to a previously defined user interface control.

Examples

Create Radio Button

Create a radio button by specifying the `'Style'` name-value pair argument as `'radiobutton'`. Label the radio button by specifying a value for the `'String'` name-value pair argument.

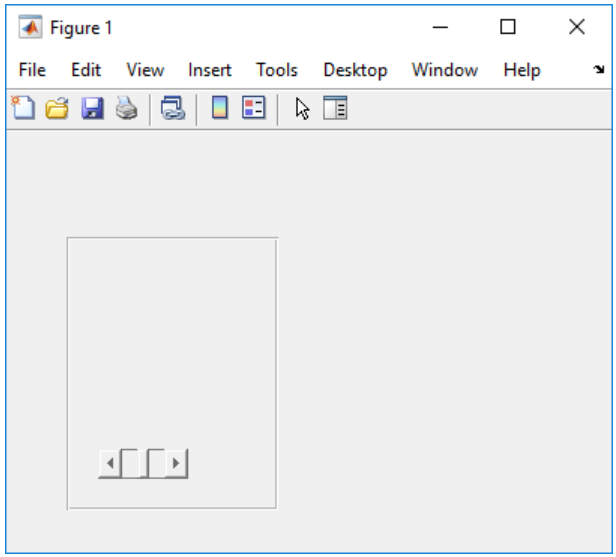
```
c = uicontrol('Style','radiobutton','String','Option 1');
```



Create Slider Within Panel

Create a figure and a panel positioned within it. Then, create a slider within the panel by calling the `uicontrol` function with the panel specified as the parent and `'Style'` specified as `'slider'`. Next, set the slider `Value` property to `0.5`.

```
f = figure;
p = uipanel(f,'Position',[0.1 0.1 0.35 0.65]);
c = uicontrol(p,'Style','slider');
c.Value = 0.5;
```



Code Response to Pop-Up Menu Selection

Create a pop-up menu that displays a list of choices when clicked. Use a callback function to determine the list item selected by the user and display the selection in the MATLAB Command Window.

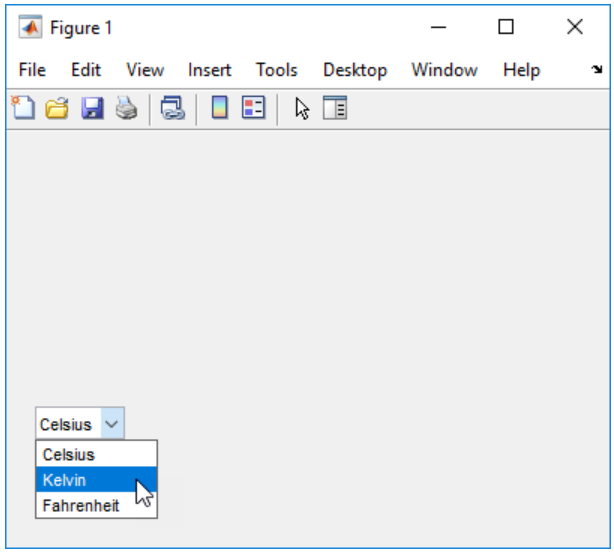
Save this code as `mytemps.m`. This code creates a figure window with a pop-up menu containing three list items. Then, it uses a callback function to query the `Value` and `String` properties of the pop-up menu and displays the selected item at the command line.

```
function mytemps
f = figure;
c = uicontrol(f,'Style','popupmenu');
c.Position = [20 75 60 20];
c.String = {'Celsius','Kelvin','Fahrenheit'};
c.Callback = @selection;

    function selection(src,event)
        val = c.Value;
        str = c.String;
        str{val};
```

```
        disp(['Selection: ' str{val}]);
    end

end
```

Run the program to generate the figure and its contents.

```
mytemps
```



Choose a different menu item to change the selection. For example, if you select "Kelvin" from the pop-up menu, the command line then displays the text `Selection: Kelvin`.

Code Response to Button Click

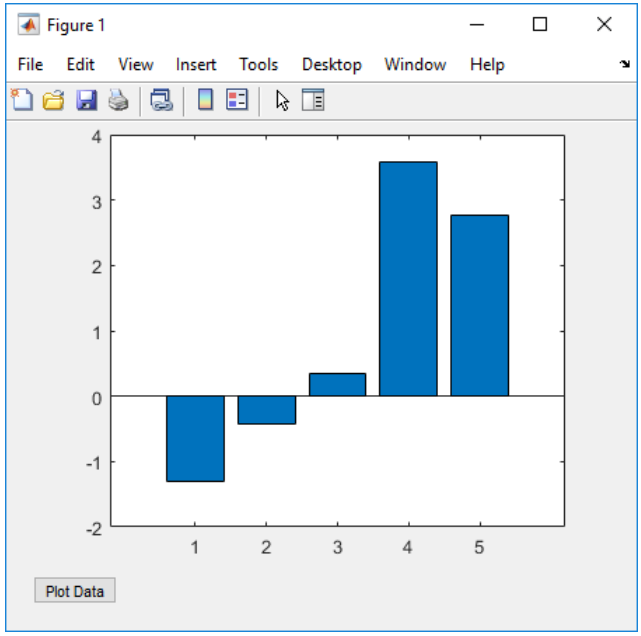
Create a push button that plots data when you click it.

Save this code as `pushbuttonPlot.m`. This code creates a figure window that contains axes and a push button. Each time you click the button, the callback function executes and plots a bar chart of five normally distributed random numbers.

```
function pushbuttonPlot
f = figure;
```

```
ax = axes(f);
ax.Units = 'pixels';
ax.Position = [75 75 325 280]
c = uicontrol;
c.String = 'Plot Data';
c.Callback = @plotButtonPushed;

    function plotButtonPushed(src,event)
        bar(randn(1,5));
    end

end
```

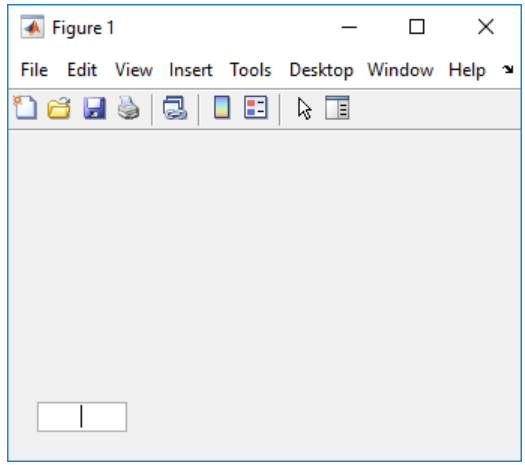
Run `pushbuttonPlot`, and then click the push button. MATLAB plots the data.



Bring Editable Text Field into Focus

Create an editable text field and bring it into focus by passing its function handle into the `uicontrol` function. This action causes the cursor to become active, and blink, within the editable text field.

```
c = uicontrol('Style','edit');
uicontrol(c);
```



Input Arguments

parent — Parent object

current figure (default) | `Figure` object | `Panel` object | `ButtonGroup` object | `Tab` object

Parent object, specified as a `Figure` object created using the `figure` function, or as one of its child containers: a `Panel`, `ButtonGroup`, or `Tab` object. Use this argument to specify the parent container when creating a user interface control.

c — User interface control object

`UIControl` object

User interface control object, specified as a `UIControl` object. Use this argument to specify a previously defined user interface control that you wish to bring into focus.

Example: `uicontrol(c)`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

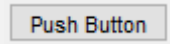
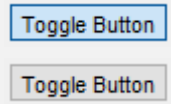
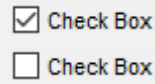
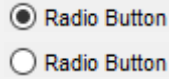
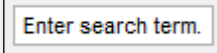
Example: `uicontrol('Style','checkbox')` specifies a check box as the style of the user interface control.

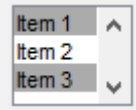
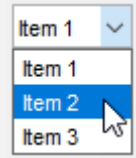
Note The properties listed here are only a subset. For a complete list, see UIControl Properties.

`Style` — Style of `UIControl` object

`'pushbutton'` (default) | `'togglebutton'` | `'checkbox'` | `'radiobutton'` | ...

Style of `UIControl` object, specified as a value from the following table.

Style Property Value	Example	Description
`'pushbutton'`		Button that appears to depress until you release the mouse button.
`'togglebutton'`		Button that looks like a push button, but that visually indicates its state: selected or cleared.
`'checkbox'`		Option that can be selected or cleared independently.
`'radiobutton'`		Option that is intended to form part of a group, such that, when selected, it clears the other options within the group. To implement mutually exclusive behavior for a set of radio buttons, place them within a `uibuttongroup`.
`'edit'`		Editable text field. To enable multiple lines of text, set the `Max` and `Min` properties so that `Max-Min > 1`.

Style Property Value	Example	Description
`'text'`	Select an item below:	Static text field. Use static text to label other user interface controls, provide information to the user, or indicate values associated with a slider. To make static text respond to mouse clicks, set the `Enable` property to `'Inactive'` and code a response with the `ButtonDownFcn` callback.
`'slider'`		"Thumb" button that the user moves along a horizontal or vertical bar. The location of the button along the bar indicates a value within a specified range.
`'listbox'`		List of items from which the user can select one or more items. Unlike pop-up menus, list boxes do not expand when clicked. To enable multiple selection of items, set the `Max` and `Min` properties so that `Max-Min > 1`. To delay action when multiple items can be selected from a list box, you can associate a **Done** push button with the list box. Then, use the callback for that button to evaluate the list box `Value` property.
`'popupmenu'`		Pop-up menu (also known as drop-down menu) that expands to display a list of choices. When closed, a pop-up menu indicates the current choice. Use pop-up menus when you want to provide a number of mutually exclusive choices.
`'frame'`	The `'frame'` option is not recommended. Use `uipanel` or `uibuttongroup` instead of frames. GUIDE continues to support frames in UIs that contain them, but the frame component does not appear in the GUIDE Layout Editor component palette.	

`String` — Text to display

character vector | cell array of character vectors | string array | categorical array | ...

Text to display, specified as a character vector, cell array of character vectors, string array, categorical array, or pipe-delimited row vector. The `Style` property dictates the array format you can use.

Style Property	Supported Array Formats	Examples

`'pushbutton'`
`'togglebutton'`
`'checkbox'`
`'radiobutton'`
`'edit'`
`'text'`
	Character vector
	Cell array of character vectors
	String array
	Categorical array
		`'Option 1'`
		`{'Option 1'}`
		`"Option 1"`
		`categorical({'Option 1'})`

`'listbox'`
`'popupmenu'`
	Character vector
	Cell array of character vectors
	String array
	Categorical array
	Pipe-delimited row vector
		`'One'`
		`{'One','Two','Three'}`
		`["One" "Two" "Three"]`
		`categorical({'one','two', 'three'})`
		`'One|Two|Three'`

Note If you specify a cell array or a categorical array for a push button, toggle button, check box, or radio button, MATLAB displays only the first element in the array.

`Position` — Location and size

`[20 20 60 20]` (default) | `[left bottom width height]`

Location and size, specified as a four-element vector of the form `[left bottom width height]`. Default measurement units are in pixels. The table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the user interface control.

Element	Description
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the user interface control.
`width`	Distance between the right and left outer edges of the user interface control.
`height`	Distance between the top and bottom outer edges of the user interface control.

`Position` values are relative to the parent container's drawable area. The drawable area is the area inside the borders of the container and does not include the area occupied by the title. If the parent container is a figure, the drawable area also excludes the menu bar and tool bar.

`Value` — Current value

number

Current value, specified as a number. Use to query or modify the status of certain user interface controls. The table describes the `Value` property in relation to specific `UIControl` styles.

Style Property	Description of Value Property
`'togglebutton'`	• **Selected:** Value of the `Max` property. • **Cleared:** Value of the `Min` property.
`'checkbox'`	• **Selected:** Value of the `Max` property. • **Cleared:** Value of the `Min` property.
`'radiobutton'`	• **Selected:** Value of the `Max` property. • **Cleared:** Value of the `Min` property.
`'slider'`	Value associated with the thumb location along the slider bar.
`'listbox'`	Array index corresponding to the selected item in the list box. A value of `1` (default) corresponds to the first item in the list. When multiple items are selected, the `Value` property stores the row indexes as a vector.
`'popupmenu'`	Array index corresponding to the selected item in the pop-up menu. A value of `1` (default) corresponds to the first item in the pop-up menu.

See Also

Functions
`figure` | `uibuttongroup`

Properties
UIControl Properties

Topics

"Add Components to a Programmatic App"
"Callbacks for Specific Components"
"Access Property Values"

Introduced before R2006a

UIControl Properties

Control appearance and behavior of user interface control

Note **The properties listed here are valid for `UIControl` objects in GUIDE or in apps created with the `figure` function.** App Designer is the recommended environment for building apps. For more information, see “GUIDE Migration Strategies”.

Description

User interface controls are components such as buttons and sliders that users can interact with. The `uicontrol` function creates a user interface control and sets any required properties before displaying it. By changing property values, you can modify the appearance and behavior of user interface controls. Use dot notation to refer to a specific object and property.

```
f = figure;
b = uicontrol(f,'Style','pushbutton');
b.Position = [100 100 50 20];
```

Properties

Type of Control

`Style` — Style of `UIControl`

`'pushbutton'` (default) | `'togglebutton'` | `'checkbox'` | `'radiobutton'` | ...

Style of `UIControl`, specified as a value from the following table.

Style Value	Description
`'pushbutton'`	Button that appears to depress until you release the mouse button.
`'togglebutton'`	Button that can have two states: up or depressed. The state of the toggle button changes every time you click it.

Style Value	Description
`'checkbox'`	Check box that can have two states: checked or unchecked. The state changes when the user clicks and releases the mouse button over it.
`'radiobutton'`	Button that can have two states: selected or deselected. Radio buttons are intended to be mutually exclusive within a group of related radio buttons.
`'edit'`	Editable text field.
`'text'`	Static text field.
`'slider'`	Button that the user pushes along a horizontal or vertical bar. The position of the button indicates a value within a specified range.
`'listbox'`	List of items from which the user can select one or more items. Unlike pop-up menus, list boxes do not expand when clicked.
`'popupmenu'`	Isolated menu that expands to display a list of choices when you click it. When it is collapsed, the menu shows the current choice.

`Value` — Current value

number

Current value, specified as a number. The `Value` property is useful in querying or modifying the status of certain uicontrols:

Style of uicontrol	Description of Value Property
`'togglebutton'`	• **Raised:** `Value` property equals the value of the `Min` property. • **Depressed:** `Value` property equals the value of the `Max` property.
`'checkbox'`	• **Unchecked:** `Value` property changes to the value of the `Min` property. • **Checked:** `Value` property changes to the value of the `Max` property.

Style of uicontrol	Description of Value Property
`'radiobutton'`	• **Deselected:** `Value` property changes to the value of the `Min` property. • **Selected:** `Value` property changes to the value of the `Max` property.
`'slider'`	`Value` property equals the corresponding slider value.
`'listbox'`	`Value` property equals an array index corresponding to the selected item in the list box. A value of `1` corresponds to the first item in the list.
`'popupmenu'`	`Value` property equals an array index corresponding to the selected item in the pop-up menu. A value of `1` corresponds to the first item in the pop-up menu.

`Max` — Maximum value

`1` (default) | number

Maximum value, specified as a number. The `Max` property affects the presentation of certain uicontrols:

Style of uicontrol	Description of Value Property
`'togglebutton'`	When the toggle button is depressed, the `Value` property changes to the value of the `Max` property.
`'checkbox'`	When the check box is checked, the `Value` property changes to the value of the `Max` property.
`'radiobutton'`	When the radio button is selected, the `Value` property changes to the value of the `Max` property.
`'edit'`	The edit text box accepts multiple lines of input when `Max` – `Min` > `1`. Otherwise, the edit text box accepts a single line of input. The absolute values of `Max` and `Min` have no effect on the number of possible lines. As long as the difference is greater than `1`, the edit box can contain any number of lines.
`'slider'`	The `Max` property value is the maximum slider value, which must be greater than the `Min` property value.

Style of uicontrol	Description of Value Property
`'listbox'`	The `Max` property value helps determine whether the user can select multiple items in the list box simultaneously. If `Max` – `Min` > 1, then the user can select multiple items simultaneously. Otherwise, the user cannot select multiple items simultaneously. If you set the `Max` and `Min` properties to allow multiple selections, then the `Value` property value can be a vector of indices.

`Min` — Minimum value

`0` (default) | number

Minimum value, specified as a number. The `Min` property affects the presentation of certain uicontrols:

Style of uicontrol	Description of Value Property
`'togglebutton'`	When the toggle button is raised, the `Value` property changes to the value of the `Min` property.
`'checkbox'`	When the check box is unchecked, the `Value` property changes to the value of the `Min` property.
`'radiobutton'`	When the radio button is deselected, the `Value` property changes to the value of the `Min` property.
`'edit'`	The edit text box accepts multiple lines of input when `Max` – `Min` > `1`. Otherwise, the edit text box accepts a single line of input. The absolute values of `Max` and `Min` have no effect on the number of possible lines. As long as the difference is greater than `1`, the edit box can contain any number of lines.
`'slider'`	The `Min` property value is the minimum slider value, which must be less than the `Max` property value.

Style of uicontrol	Description of Value Property
`'listbox'`	The `Max` property value helps determine whether the user can select multiple items in the list box simultaneously. If `Max` – `Min` > 1, then the user can select multiple items simultaneously. Otherwise, the user cannot select multiple items simultaneously. If you set the `Max` and `Min` properties to allow multiple selections, then the `Value` property value can be a vector of indices.

`SliderStep` — Slider step size

`[0.01 0.10]` (default) | `[minorstep majorstep]`

Slider step size, specified as the array, `[minorstep majorstep]`. This property controls the magnitude of the slider value change when the user clicks the arrow buttons or the slider trough (slider channel):

- `minorstep` is the fraction of the slider range by which the `Value` property increases or decreases when the user clicks one of the arrow buttons.
- `majorstep` is the fraction of the slider range by which the `Value` property increases or decreases when the user clicks the slider trough.

When the value of `majorstep` is greater than 1, the slider fraction is no greater than 100%.

Both `minorstep` and `majorstep` must be greater than `1e-6`, and `minorstep` must be less than or equal to `majorstep`.

As `majorstep` increases, the slider thumb indicator grows longer. When `majorstep` is equal to `1`, the thumb indicator is half as long as the trough. The thumb indicator is larger for `majorstep` values greater than `1`.

Example

This slider has a range of `Max` – `Min` = `1`. The slider `Value` property changes by 1% when the user presses an arrow button. It changes by 10% when the user clicks in the trough.

```
s = uicontrol('Style','Slider','Min',0,'Max',1,'SliderStep',[0.01 0.10]);
```

`ListboxTop` — Index of top item in list box

1 (default) | integer value

Index of top item in list box, specified as an integer value. This property applies only to the listbox style of `UIControl`. This property specifies which item appears in the top-

most position in a list box that is not large enough to display all list entries. The `ListboxTop` value is an index into the array of you specify as the `String` property value. The `ListboxTop` value must be between `1` and the number of elements in the array. Noninteger values are fixed to the next lowest integer.

Note The `String` and `Value` properties might override the value of the `ListboxTop` property regardless of the `ListboxTop` value you specify. The `ListboxTop` value can change depending on the value of other `UIControl` properties. For example, explicitly setting the `Value` property scrolls the list to that value.

For the most reliable results, query or modify the `ListboxTop` property after MATLAB finishes drawing the user interface control on the screen.

Text and Styling

`String` — Text to display

character vector | cell array of character vectors | string array | categorical array | ...

Text to display, specified as a character vector, cell array of character vectors, string array, categorical array, or pipe-delimited row vector. The `Style` property dictates the array format you can use.

Style Property	Supported Array Formats	Examples

`'pushbutton'`
`'togglebutton'`
`'checkbox'`
`'radiobutton'`
`'edit'`
`'text'`
	Character vector
	Cell array of character vectors
	String array
	Categorical array
		`'Option 1'`
		`{'Option 1'}`
		`"Option 1"`
		`categorical({'Option 1'})`

`'listbox'`	Character vector
	Cell array of character vectors
	String array
	Categorical array
		`'One'`
		`{'One','Two','Three'}`
		`["One" "Two" "Three"]`
		`categorical({'one','two', 'three'})`

Style Property	Supported Array Formats	Examples
`'popupmenu'`	Pipe-delimited row vector	`'One|Two|Three'`

Note If you specify a cell array or categorical array for a push button, toggle button, check box, or radio button, MATLAB displays only the first element in the array.

`ForegroundColor` — Text color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Text color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Note If you change the value of `ForegroundColor` for a listbox, then MATLAB uses that color for all listbox items, except for the currently selected listbox item. For selected items, MATLAB uses a color that ensures good contrast between the text of selected items and the selection color.

Example: `[0 0 1]`

Example: `'b'`

Example: `'blue'`

Data Types: `double` | `char`

`BackgroundColor` — Background color
`[.94 .94 .94]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to

F. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`CData` — Optional icon

3-D array of truecolor RGB values

Optional icon, specified as a 3-D array of truecolor RGB values. The values in the array can be:

- Double-precision values between `0.0` and `1.0`
- `uint8` values between `0` and `255`

Push buttons and toggle buttons are the only `UIControl` objects that fully support `CData`. If you specify the `CData` property for a radio button or check box, the image might overlap with the text. Also, specifying an image for radio button or check box disables the ability to show when they are selected or deselected.

Data Types: `double` | `uint8`

Font

`FontName` — Font name

system supported font name | `'FixedWidth'`

Font name, specified as a system supported font name or `'FixedWidth'`. The default font depends on the specific operating system and locale.

To use a fixed-width font that looks good in any locale, specify `'FixedWidth'`. The actual fixed-width font used depends on the `FixedWidthFontName` property of the root object. Changing the `FixedWidthFontName` property causes an immediate update of the display to use the new font.

Example: `'Arial'`

`FontSize` — Font size

positive number

Font size, specified as a positive number. The `FontUnits` property specifies the units. The default size is system-dependent.

Example: `12`

Example: `12.5`

`FontWeight` — Font weight

`'normal'` (default) | `'bold'`

Font weight, specified as a value from the following table.

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than normal

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold font weight. Therefore, specifying a bold font weight still can result in the normal font weight.

Note The `'light'` and `'demi'` font weight values have been removed in R2014b. If you specify either of these values, the result is a normal font weight.

`FontAngle` — Font angle

`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. MATLAB uses this property to select a font from those available on your system. Setting this property to `'italic'` selects a slanted version of the font, if it is available on your system.

Note The `'oblique'` value has been removed. Use `'italic'` instead.

`FontUnits` — Font units

`'points'` (default) | `'normalized'` | `'inches'` | `'centimeters'` | `'pixels'`

Font units, specified as one of the values from this table.

Units Value	Description
`'points'`	Points. One point is 1/72nd of an inch.
`'normalized'`	Normalized values for specifying the font size as a fraction of the height. When you resize a UI component, MATLAB scales the displayed font to maintain that fraction.
`'inches'`	Inches.
`'centimeters'`	Centimeters.

Units Value	Description
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.

Interactivity

`Visible` — User interface control visibility

`'on'` (default) | `'off'`

User interface control visibility, specified as `'on'` or `'off'`. When `Visible` is `'off'`, the user interface control is not visible, but you can query and set its properties.

To make your app start faster, set the `Visible` property of all user interface controls that are not initially displayed to `'off'`.

`Enable` — Operational state of user interface control

`'on'` (default) | `'off'` | `'inactive'`

Operational state of user interface control, specified as `'on'`, `'off'`, or `'inactive'`. The `Enable` property controls whether or not the user interface control responds to user interaction. These are the possible values:

- `'on'` – The user interface control is operational.
- `'off'` – The user interface control is not operational and appears grayed-out.
- `'inactive'` – The user interface control is not operational, but it has the same appearance as when `Enable` is set to `'on'`.

`Tooltip` — Tooltip

character vector | string scalar | categorical array

Tooltip, specified as a character vector, string scalar, or categorical array. Use this property to display a message when the user hovers the pointer over the component at

run time. The tooltip does not display when the component is disabled. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

To create multiple lines of text, use the `sprintf` function to insert newline characters (`'\n'`) in your text. For example:

```
txt = sprintf('Line 1\nLine 2');
```

Then set the `Tooltip` property to the value returned by `sprintf`.

`UIContextMenu` — Context menu
empty `GraphicsPlaceholder` array (default) | uicontextmenu object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when the user right-clicks on a UI component. Create the context menu using the `uicontextmenu` function.

`TooltipString` — Tooltip (not recommended)
character vector | string scalar | categorical array

Tooltip, specified as a character vector, string scalar, or categorical array. The tooltip displays when the user hovers the mouse pointer over the component in the app. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Note The `TooltipString` property is not recommended starting in R2018b. Use the `Tooltip` property instead.

`Selected` — Selection state (not recommended)
`'off'` (default) | `'on'`

Note The behavior of the `Selected` property changed in R2014b, and it is not recommended. It no longer has any effect on `UIControl` objects. This property might be removed in a future release.

`SelectionHighlight` — Display of selection handles (not recommended)
`'on'` (default) | `'off'`

Note The behavior of the `SelectionHighlight` property changed in R2014b, and it is not recommended. It no longer has any effect on `UIControl` objects. This property might be removed in a future release.

Position

`Position` — Location and size

`[left bottom width height]`

Location and size, specified as a four-element vector of the form `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the user interface control
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the user interface control
`width`	Distance between the right and left outer edges of the user interface control
`height`	Distance between the top and bottom outer edges of the user interface control

All measurements are in units specified by the `Units` property.

Note The `Position` values are relative to the parent container's drawable area. The drawable area is the area inside the borders of the container and does not include the area occupied by the title. If the parent container is a figure, then the drawable area also excludes the menu bar and tool bar.

Modify One Value in the Position Vector

You can combine dot notation and array indexing when you want to change one value in the `Position` vector. For example, this code changes the width of the user interface control to `52`:

```
b = uicontrol;
b.Position(3) = 52;
b.Position

ans =

    20    20    52    20
```

`InnerPosition` — Location and size
`[left bottom width height]`

Location and size, specified as a four-element vector of the form `[left bottom width height]`. All measurements are in units specified by the `Units` property.

This property value is identical to the `Position` and `OuterPosition` property values.

`OuterPosition` — Location and size
`[left bottom width height]`

Location and size, specified as a four-element vector of the form `[left bottom width height]`. All measurements are in units specified by the `Units` property.

This property value is identical to the `Position` and `InnerPosition` property values.

`Extent` — Size of enclosing rectangle
four-element row vector

This property is read-only.

Size of enclosing rectangle, returned as a four-element row vector. The first two elements of the vector are always zero. The third and fourth elements are the width and height of the rectangle, respectively. All measurements are in units specified by the `Units` property.

MATLAB determines the size of the rectangle based on the size of the `String` property value and the font characteristics. To adjust the width and height to accommodate the size of the `String` value, set the `Position` width and height values to be slightly larger than the `Extent` width and height values.

For a `String` value that is a single line of text, the height element of the `Extent` property indicates the height of a single line. The width element indicates the width of the longest line, even if the text wraps when displayed on the control. For multiple lines of

text, the `Extent` rectangle encompasses all lines of text. Editable text fields are considered multiline if `Max` – `Min` > `1`.

`Units` — Units of measurement
`'pixels'` (default) | `'normalized'` | `'inches'` | `'centimeters'` | `'points'` | `'characters'`

Units of measurement, specified as one of the values from this table.

Units Value	Description
`'pixels'` (default)	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.
`'normalized'`	These units are normalized with respect to the parent container. The lower-left corner of the container maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'points'`	Points. One point equals 1/72nd of an inch.

Units Value	Description
`'characters'`	These units are based on the default uicontrol font of the graphics root object: • Character width = width of the letter `x`. • Character height = distance between the baselines of two lines of text. To access the default uicontrol font, use `get(groot,'defaultuicontrolFontName')` or `set(groot,'defaultuicontrolFontName')`.

MATLAB measures all units from the lower left corner of the parent object.

This property affects the `Position` property. If you change the `Units` property, consider returning its value to the default value after completing your computation to avoid affecting other functions that assume the default value.

The order in which you specify the `Units` and `Position` properties has these effects:

- If you specify the `Units` before the `Position` property, then MATLAB sets `Position` using the units you specify.
- If you specify the `Units` property after the `Position` property, MATLAB sets the position using the default `Units`. Then, MATLAB converts the `Position` value to the equivalent value in the units you specify.

`HorizontalAlignment` — Alignment of uicontrol text

`'center'` (default) | `'left'` | `'right'`

Alignment of the uicontrol text, specified as `'center'`, `'left'`, or `'right'`. This property determines the justification of the `String` property text.

The `HorizontalAlignment` property affects only `'text'` and `'edit'` styles of uicontrols.

Callbacks

`Callback` — Primary callback function

`''` (default) | function handle | cell array | character vector

Primary callback function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This function executes in response to user interaction, such as push button clicks, slider movements, or check box selections. This function can execute only when the `Enable` property of the `UIControl` object is set to `'on'`.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

Data Types: `function_handle` | `cell` | `char`

`ButtonDownFcn` — Button-press callback function
`''` (default) | function handle | cell array | character vector

Button-press callback function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

The `ButtonDownFcn` callback is a function that executes when the user clicks a mouse button on a UI component. The callback executes in the following situations:

- The user right-clicks the UI component, and the `Enable` property is set to `'on'`.
- The user right-clicks or left-clicks the UI component, and the `Enable` property is set to `'off'` or `'inactive'`.

`KeyPressFcn` — Key press callback function
`''` (default) | function handle | cell array | character vector

Key press callback function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

This callback function executes when the `UIControl` object has focus and the user presses a key. If you do not define a function for this property, MATLAB passes key presses to the parent figure. Repeated key presses retain the focus of the `UIControl` object, and the function executes with each key press. If the user presses multiple keys at approximately the same time, MATLAB detects the key press for the last key pressed.

If you specify this property as a function handle (or cell array containing a function handle), MATLAB passes an object containing callback data as the second argument to the callback function. This object contains the properties described in the following table. You can access these properties inside the callback function using dot notation.

Property	**Description**	**Examples:**			
		a	**=**	**Shift**	**Shift-a**
`Character`	The character that displays as a result of pressing a key or keys. The character can be empty or unprintable.	`'a'`	`'='`	`''`	`'A'`
`Modifier`	A cell array containing the names of one or more modifier keys that are being pressed (such as, **Ctrl**, **Alt**, **Shift**).	`{1x0 cell}`	`{1x0 cell}`	`{'shift'}`	`{'shift'}`
`Key`	The key being pressed, identified by the (lowercase) label on the key, or a text description.	`'a'`	`'equal'`	`'shift'`	`'a'`
`Source`	The object that has focus when the user presses the key.	`UIControl` object	`UIControl` object	`UIControl` object	`UIControl` object

Property	Description	Examples:			
		a	=	Shift	Shift-a
EventName	The action that caused the callback function to execute.	'KeyPress'	'KeyPress'	'KeyPress'	'KeyPress'

Pressing modifier keys affects the callback data in the following ways:

- Modifier keys can affect the `Character` property, but do not change the `Key` property.
- Certain keys, and keys modified with **Ctrl**, put unprintable characters in the `Character` property.
- **Ctrl**, **Alt**, **Shift**, and several other keys, do not generate `Character` property data.

You also can query the `CurrentCharacter` property of the figure to determine which character the user pressed.

`KeyReleaseFcn` — Key-release callback function
`''` (default) | function handle | cell array | character vector

Key-release callback function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

This callback function executes when the `UIControl` object has focus and the user releases a key.

If you specify this property as a function handle (or cell array containing a function handle), MATLAB passes an object containing callback data as the second argument to the callback function. This object contains the properties described in the following table. You can access these properties inside the callback function using dot notation.

Property	Description	Examples:			
		a	=	Shift	Shift-a
`Character`	Character interpretation of the key that was released.	`'a'`	`'='`	`''`	`'A'`
`Modifier`	Current modifier, such as `'control'`, or an empty cell array if there is no modifier.	`{1x0 cell}`	`{1x0 cell}`	`{1x0 cell}`	`{1x0 cell}`
`Key`	Name of the key that was released, identified by the lowercase label on the key, or a text description.	`'a'`	`'equal'`	`'shift'`	`'a'`
`Source`	The object that has focus when the user presses the key.	`UIControl` object	`UIControl` object	`UIControl` object	`UIControl` object
`EventName`	The action that caused the callback function to execute.	`'ase'`	`'ase'`	`'ase'`	`'ase'`

Pressing modifier keys affects the callback data in the following ways:

- Modifier keys can affect the `Character` property, but do not change the `Key` property.
- Certain keys, and keys modified with **Ctrl**, put unprintable characters in the `Character` property.
- **Ctrl**, **Alt**, **Shift**, and several other keys, do not generate `Character` property data.

You also can query the `CurrentCharacter` property of the figure to determine which character the user pressed.

`CreateFcn` — Component creation function

`''` (default) | function handle | cell array | character vector

Component creation function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

This property specifies a callback function to execute when MATLAB creates the component. MATLAB initializes all component property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Use the `gcbo` function in your `CreateFcn` code to get the component object that is being created.

Setting the `CreateFcn` property on an existing component object has no effect.

`DeleteFcn` — Component deletion function

`''` (default) | function handle | cell array | character vector

Component deletion function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

The `DeleteFcn` property specifies a callback function to execute when MATLAB deletes the component (for example, when the user closes the window). MATLAB executes the `DeleteFcn` callback before destroying the properties of the component object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

Use the `gcbo` function in your `DeleteFcn` code to get the component object that is being deleted.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed:

- A value of `'on'` allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- A value of `'off'` blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn`, or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.
- MATLAB does not save the state of properties or the display when an interruption occurs. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

See “Interrupt Callback Execution” for an example that shows how the `Interruptible` and `BusyAction` properties affect the behavior of a program.

BusyAction — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` (default) or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

The `BusyAction` property of the source of the interrupting callback determines how MATLAB handles its execution. The `BusyAction` property has these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Do not execute the interrupting callback.

Whenever MATLAB invokes a callback, that callback always attempts to interrupt an executing callback. The `Interruptible` property of the object whose callback is running determines if interruption is allowed. If `Interruptible` is set to:

- `on` — Interruption occurs at the next point where MATLAB processes the queue. This is the default.
- `off` — The `BusyAction` property (of the object owning the interrupting callback) determines if MATLAB enqueues or ignores the interrupting callback.

See “Interrupt Callback Execution” for an example that shows how the `BusyAction` and `Interruptible` properties affect the behavior of a program.

BeingDeleted — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

`HitTest` — Ability to become current object

`'on'` (default) | `'off'`

Ability to become current object, specified as `'on'` or `'off'`:

- `'on'` — Sets the current object to the `UIControl` when the user clicks the component in the running app. Both the `CurrentObject` property of the `Figure` and the `gco` function return the `UIControl` as the current object.
- `'off'` — Sets the current object to be the closest ancestor of the `UIControl` whose `HitTest` is `'on'` when the user clicks the component in the running app.

Note Use the Enable property to enable or disable a component.

Parent/Child

`Parent` — Parent object

`Figure` | `Panel` | `ButtonGroup` | `Tab`

Parent object, specified as a `Figure`, `Panel`, `ButtonGroup`, or `Tab` object. Use this property to specify the parent container when creating a UI component or to move an existing UI component to a different parent container.

`Children` — `UIControl` children

empty array

`UIControl` children, returned as an empty array. `UIControl` objects have no children. Setting this property has no effect.

`HandleVisibility` — Visibility of `UIControl` handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of `UIControl` handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the `UIControl` handle in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. The `HandleVisibility` property also controls the visibility of the object’s handle in the parent figure's `CurrentObject` property. Handles are still valid even if they are not visible. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

HandleVisibility Value	Description
`'on'`	The `UIControl` handle is always visible.
`'callback'`	The `UIControl` handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the `UIControl` at the command-line, but allows callback functions to access it.
`'off'`	The `UIControl` handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.

Set the graphics root `ShowHiddenHandles` property to `'on'` to make all handles visible, regardless of their `HandleVisibility` value. This setting has no effect on their `HandleVisibility` values.

Note Do not try to access radio buttons and toggle buttons that are managed by a `uibuttongroup` outside of the button group. Set the `HandleVisibility` of those radio buttons and toggle buttons to `'off'` to prevent accidental access.

Identifiers

Type — Type of graphics object
`'uicontrol'`

This property is read-only.

Type of graphics object, returned as `'uicontrol'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

[ ] (default) | array

User data, specified as any array. Specifying `UserData` can be useful for sharing data within apps. See “Share Data Among Callbacks” for more information.

See Also

`guide` | `uicontrol`

Topics

“Access Property Values”
“Default Property Values”

Introduced before R2006a

uidatepicker

Create date picker component

Syntax

```
d = uidatepicker
d = uidatepicker(Name,Value)
d = uidatepicker(parent)
d = uidatepicker(parent,Name,Value)
```

Description

`d = uidatepicker` creates a date picker in a new figure and returns the `DatePicker` object. MATLAB calls the `uifigure` function to create the figure.

`d = uidatepicker(Name,Value)` specifies `DatePicker` property values using one or more `Name,Value` pair arguments.

`d = uidatepicker(parent)` creates a date picker in the specified parent container. The parent can be a `Figure` created using the `uifigure` function, or one of its child containers.

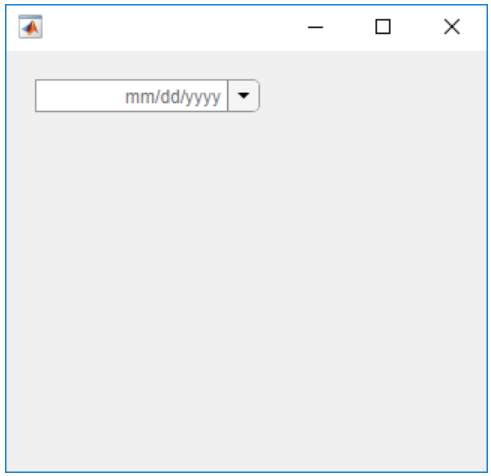
`d = uidatepicker(parent,Name,Value)` creates the date picker in the specified container and sets one or more `DatePicker` property values.

Examples

Create a Date Picker in a Figure

Create a date picker in the upper left corner of a figure.

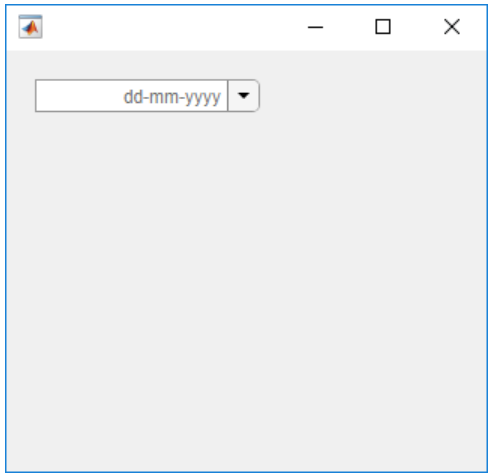
```
f = uifigure('Position',[500 500 320 280]);
d = uidatepicker(f,'Position',[18 235 150 22]);
```



Display Custom Date Format

Create a date picker that displays the date in the text field using the `dd-MM-yyyy` format. The watermark in the running app displays the new format, and all selected dates use that format.

```
f = uifigure('Position',[500 500 320 280]);
d = uidatepicker(f,'Position',[18 235 150 22]);
d.DisplayFormat = 'dd-MM-yyyy';
```

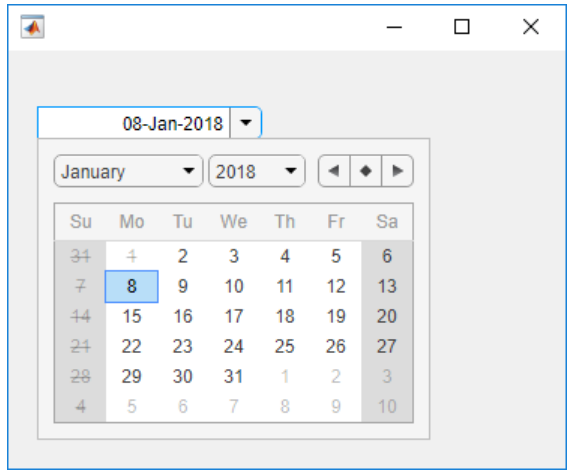


Disable Sundays and a Holiday

Create a date picker that disables Sundays and New Year's day 2018.

```
f = uifigure('Position',[500 500 375 280]);
d = uidatepicker(f,'Position',[18 225 150 22]);
d.DisabledDaysOfWeek = 1;
d.DisabledDates = datetime(2018,1,1);
```

When you expand the date picker and browse to January 2018, the first day of the year and all Sundays are disabled.



Create a Callback for Date Selection

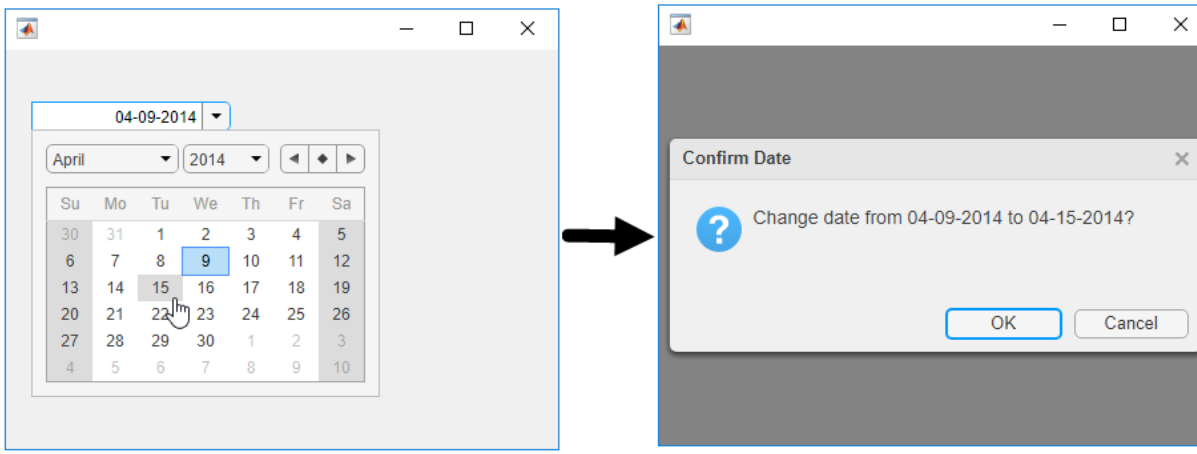
Create a program file called `mydateapp.m` that creates a figure and a date picker with a `ValueChangedFcn` callback.

```
function mydateapp
f = uifigure('Position',[340 400 415 300]);
d = uidatepicker(f,'DisplayFormat','MM-dd-yyyy',...
    'Position',[130 190 150 22],...
    'Value',datetime(2014,4,9),...
    'ValueChangedFcn', @datechange);

    function datechange (src,event)
        lastdate = char(event.PreviousValue);
        newdate = char(event.Value);
        msg = ['Change date from ' lastdate ' to ' newdate '?'];
        % Confirm new date
        selection = uiconfirm(f,msg,'Confirm Date');

        if (strcmp(selection,'Cancel'))
            % Revert to previous selection if cancelled
            d.Value = event.PreviousValue;
        end
    end
end
```

The `datechange` function displays a confirmation dialog box and determines which button the user clicks in that dialog box. The date picker reverts to the previous date if the user clicks **Cancel**.

Run the program, and click a date to see the confirmation dialog box.

```
mydateapp
```



Input Arguments

`parent` — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If you do not specify a parent container, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `d = uidatepicker('Value',datetime('today'))` creates a date picker with today's date selected in the UI.

Note The properties listed here are only a subset. For a complete list, see DatePicker Properties.

`Value` — Selected date

`NaT` (default) | `datetime` object

Selected date, specified as a `datetime` object within the range of the `Limits` property. To make the selected date unspecified, set this property to `NaT`.

If the specified `datetime` object contains time information, only the date information is preserved in the `Value` property.

Example: `d = uidatepicker('Value',datetime('today'))`

Data Types: `datetime`

`DisplayFormat` — Display format

character vector | string scalar

Display format for the date picker text field, specified as a character vector or string scalar. The default format depends on the locale of the system running the app.

The format you specify must use valid letter identifiers that correspond to the Unicode Locale Data Markup Language (LDML) standard for dates and times. To separate fields, you can include nonletter characters such as a hyphen, space, colon, or any non-ASCII characters.

Example: `d = uidatepicker('DisplayFormat','dd/MM/yy')`

Examples of Common Formats

This table lists common display formats. The examples show formatted output for the date, Wednesday, April 9, 2014.

Value of Format	Example
'yyyy-MM-dd'	2014-04-09
'dd/MM/yyyy'	09/04/2014
'dd.MM.yyyy'	09.04.2014
'yyyy 年 MM 月 dd 日'	2014 年 04 月 09 日
'MMMM d, yyyy'	April 9, 2014

All Date and Time Formats

Use these letter identifiers to create a display format. The third column of this table shows output for the date, Wednesday, April 9, 2014.

Letter Identifier	Description	Display
G	Era	CE
y	Year, with no leading zeros.	2014
yy	Year, using last two digits.	14
yyy, yyyy ...	Year, using at least as many digits as there are instances of 'y'	For the year 2014, 'yyy' displays 2014, while 'yyyyy' displays 02014.
u, uu, ...	ISO year, a single number designating the year.	2014
Q	Quarter, using one digit	2
QQ	Quarter, using two digits	02
QQQ	Quarter, abbreviated	Q2
QQQQ	Quarter, full name	2nd quarter
M	Month, numerical, using one or two digits	4
MM	Month, numerical, using two digits	04
MMM	Month, abbreviated name	Apr
MMMM	Month, full name	April
MMMMM	Month, capitalized first letter	A

Letter Identifier	Description	Display
W	Week of the month, using one digit	2
d	Day of the month, using one or two digits	9
dd	Day of the month, using two digits	09
D	Day of the year, using one, two, or three digits	99
DD	Day of the year, using two digits	99
DDD	Day of the year using three digits	099
e	Day of the week, numerical, using one or two digits	4, where Sunday is the first day of the week
ee	Day of the week, numerical, using two digits	04
eee	Day, abbreviated name	Wed
eeee	Day, full name	Wednesday
eeeee	Day, capitalized first letter	W

Note

- The edit field in the running app accepts delimited numeric values, even when the `DisplayFormat` includes words. For instance, if the month format is specified as `'MMMM'`, the app accepts a numeric month such as `04`, but will display a month name such as `'April'`.
- If the user specifies a day-of-year number in the running app, and the format contains identifiers for both the day of year (`D`) and Gregorian year (`y`), then `datetime` might not read the day-of-year number correctly. Use ISO year (`u`) in place of `y`.
- Use one or more `u` characters instead of `y` characters to represent the year when working with year numbers near zero.

`ValueChangedFcn` — Value changed function

`''` (default) | function handle | cell array | character vector

Value changed function, specified as one of the following:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

The `ValueChangedFcn` callback executes when the user changes the date by typing in the text field or by expanding the date picker and selecting a date.

This callback function can access specific information about the user's interaction with the date picker. MATLAB passes this information in a `ValueChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can get the object properties using dot notation. For example, `event.PreviousValue` gets the previously selected date. The `ValueChangedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangedData` object.

Property	Value
`Value`	New selected date
`PreviousValue`	Previously selected date
`Source`	Component that executes the callback
`EventName`	`'ValueChanged'`

The `ValueChangedFcn` callback does not execute when the user re-selects or re-types the currently selected date. The callback also does not execute when the `Value` property changes programmatically.

For more information about creating callbacks in App Designer, see "Write Callbacks in App Designer".

`Position` — Location and size

`[100 100 150 22]` (default) | `[left bottom width height]`

Location and size of the collapsed date picker relative to the parent container, specified as a vector of the form `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the date picker
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the date picker
`width`	Distance between the right and left outer edges of the date picker
`height`	Distance between the top and bottom outer edges of the date picker

All measurements are in pixel units.

See Also

Functions
`NaT` | `datetime`

Properties
DatePicker Properties

Introduced in R2018a

uigetdir

Open folder selection dialog box

Syntax

```
selpath = uigetdir
selpath = uigetdir(path)
selpath = uigetdir(path,title)
```

Description

`selpath = uigetdir` opens a modal on page 1-16099 dialog box that displays the folders in the current working directory and returns the path that the user selects from the dialog box.

This dialog box enables a user to navigate to a folder and select it (or type the name of a folder). If the specified folder exists, then MATLAB returns the selected path when the user clicks **OK**. If the user clicks **Cancel** or the close button (X) on the title bar, then MATLAB returns `0`.

`selpath = uigetdir(path)` specifies the start path in which the dialog box opens. If `path` is empty or is not a valid path, then the dialog box opens in the current working directory.

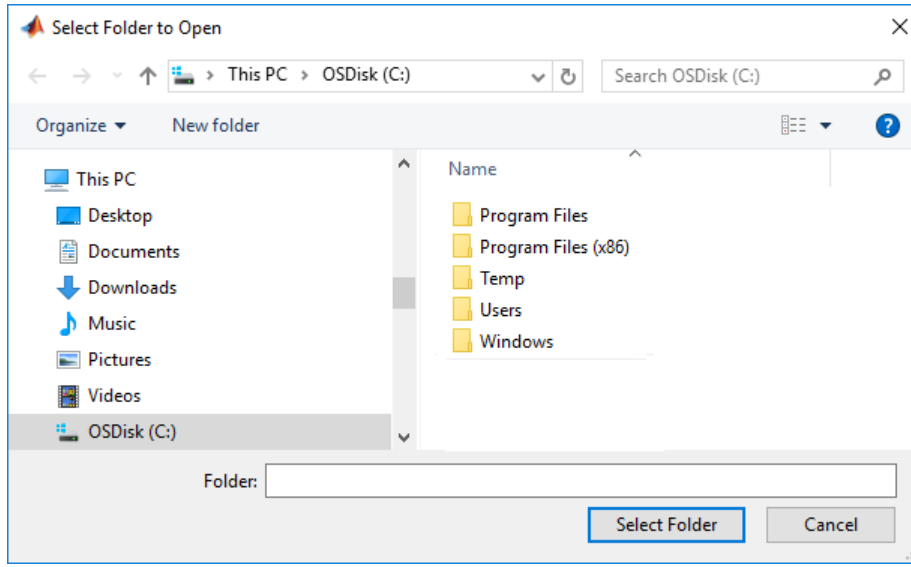
`selpath = uigetdir(path,title)` specifies a title for the dialog box.

Note The visual characteristics of the dialog box depend on the operating system that runs your code. For instance, some operating systems do not show title bars on dialog boxes. If you pass a dialog box title to the uigetdir function, those operating systems do not display the title.

Examples

Display Folders on C: Drive

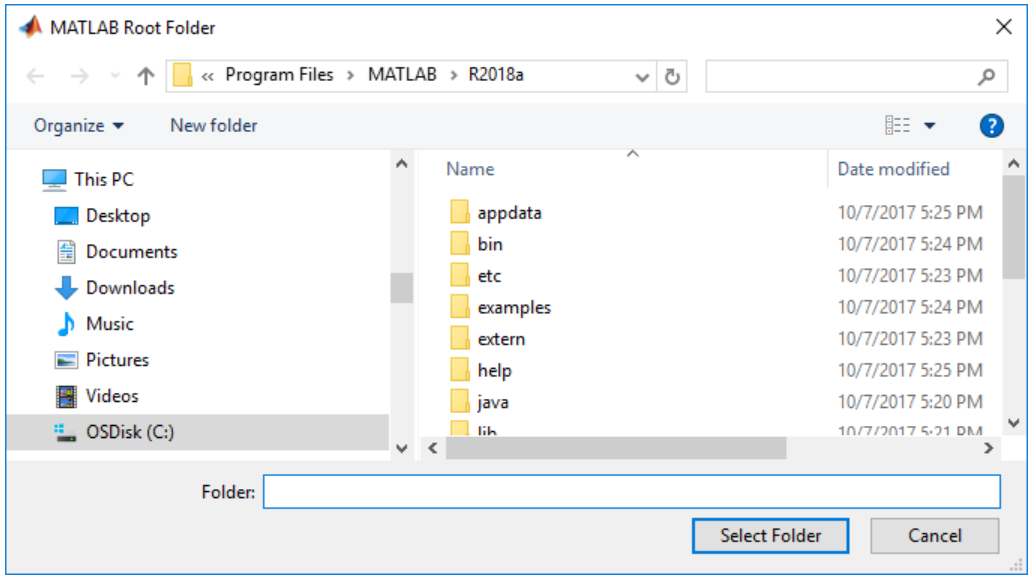
```
dname = uigetdir('C:\');
```



Display Folders in MATLAB Root Folder

Pass the command form of the `matlabroot` function to `uigetdir` to display folders in the MATLAB root folder.

```
uigetdir(matlabroot,'MATLAB Root Folder');
```



Input Arguments

`path` — Initial folder

current working directory (default) | character vector | string scalar

The initial folder in which the dialog box opens, specified as character vector or string scalar.

Example: `'C:\Users\hharvey\Documents'`

`title` — Dialog box title

`Select Folder to Open` (default) | character vector | string scalar

The dialog box title, specified as a character vector or string scalar

Example: `'Open Directory'`

More About

Modal Dialog Box

A modal dialog box prevents a user from interacting with other MATLAB windows before responding to the modal dialog box.

See Also

`uigetfile` | `uiputfile`

Introduced before R2006a

uigetfile

Open file selection dialog box

Syntax

```
file = uigetfile
[file,path] = uigetfile
[file,path,indx] = uigetfile
___ = uigetfile(filter)
___ = uigetfile(filter,title)
___ = uigetfile(filter,title,defname)
___ = uigetfile( ___ ,'MultiSelect',mode)
```

Description

`file = uigetfile` opens a modal on page 1-16112 dialog box that lists files in the current folder. It enables a user to select or enter the name of a file. If the file exists and is valid, `uigetfile` returns the file name when the user clicks **Open**. If the user clicks **Cancel** or the window close button (X), `uigetfile` returns `0`.

`[file,path] = uigetfile` returns the file name and path to the file when the user clicks **Open**. If the user clicks **Cancel** or the window close button (X), then `uigetfile` returns `0` for both of the output arguments.

`[file,path,indx] = uigetfile` returns the index of the filter selected in the dialog box when the user clicks **Open**.

`___ = uigetfile(filter)` specifies a file extension by which files displayed in the dialog box are filtered. Use this syntax with any of the output argument combinations in the previous syntaxes.

Typically, only files with a matching file extension are displayed. On some platforms, `uigetfile` displays files that do not match the filter, but dims those file names. If the filter is missing or empty, `uigetfile` uses the default list of file types (for example, all MATLAB files).

`___ = uigetfile(filter,title)` specifies a dialog box title. To filter using the default file filter, but specify a custom title, use empty quotes for the filter value. For example:

```
file = uigetfile('','Select a File')
```

`___ = uigetfile(filter,title,defname)` specifies a default file name for the **File name** field.

`___ = uigetfile(___,'MultiSelect',mode)` specifies whether a user can select multiple files. Set the mode to `'on'` to enable multifile selection. By default it is set to `'off'`.

Windows libraries can span multiple folders.

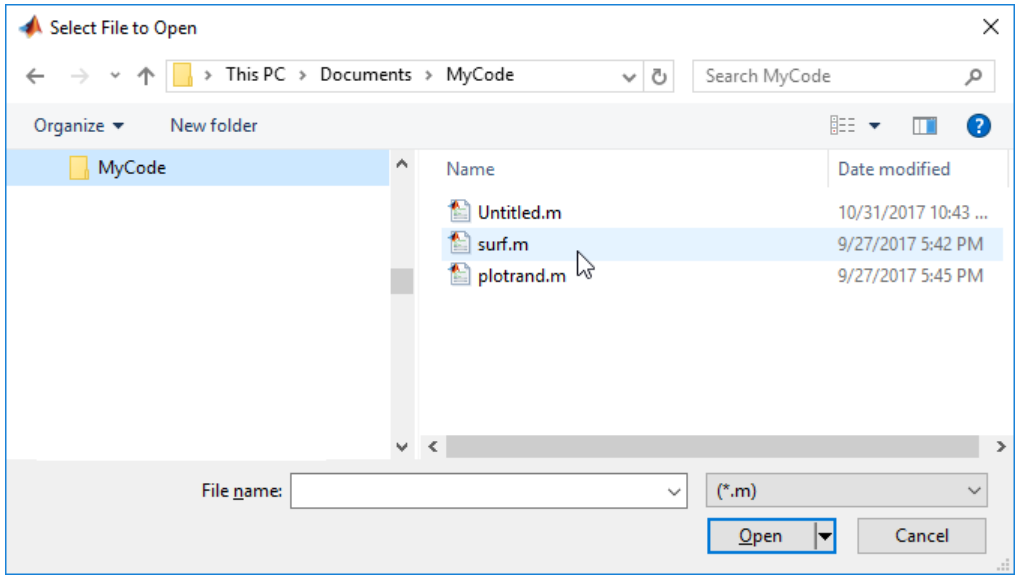
Note The visual characteristics of the dialog box depend on the operating system that runs your code. For instance, some operating systems do not show title bars on dialog boxes. If you pass a dialog box title to the uigetfile function, those operating systems do not display the title.

Examples

Display Full File Specification

Display the full file specification of the file selected in the dialog box. Use the `disp` and `fullfile` functions to add explanatory text and concatenate the `path` and `file` output values.

```
[file,path] = uigetfile('*.m');
if isequal(file,0)
   disp('User selected Cancel');
else
   disp(['User selected ', fullfile(path,file)]);
end
```

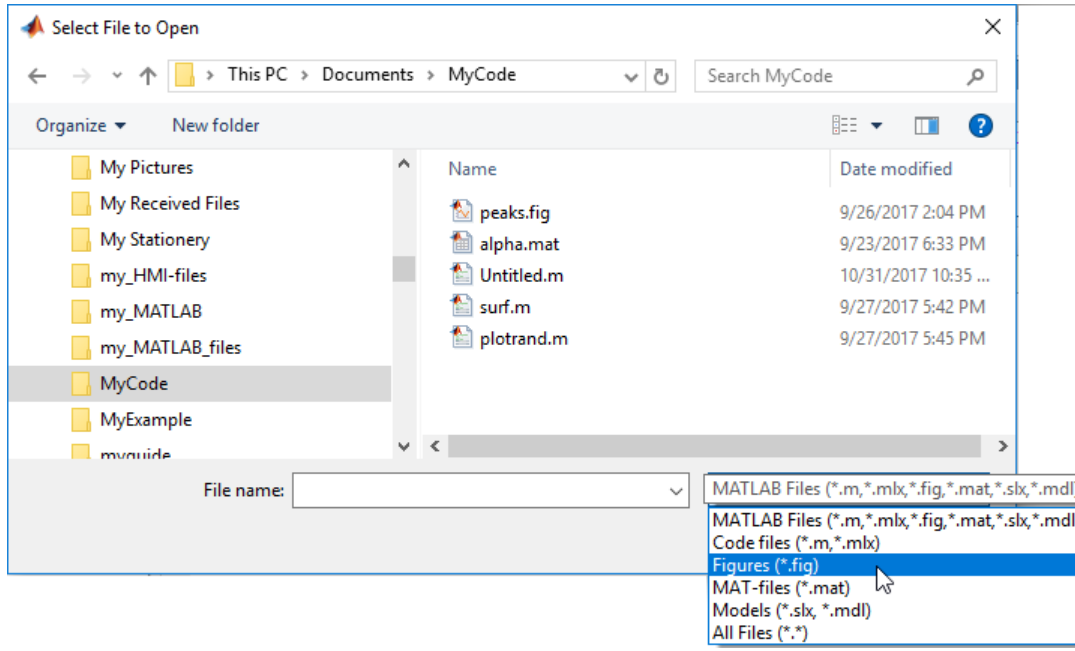



```
User selected H:\Documents\MyCode\surf.m
```

Display Filter Index Selection

Display the filter index selection with explanatory text in the Command Window. Use the `num2str` function to convert the numeric filter index value (`indx`) to a character array. Doing so makes the value valid input to the `disp` function.

```
[file,path,indx] = uigetfile;
if isequal(file,0)
   disp('User selected Cancel')
else
   disp(['User selected ', fullfile(path, file),...
         ' and filter index: ', num2str(indx)])
end
```

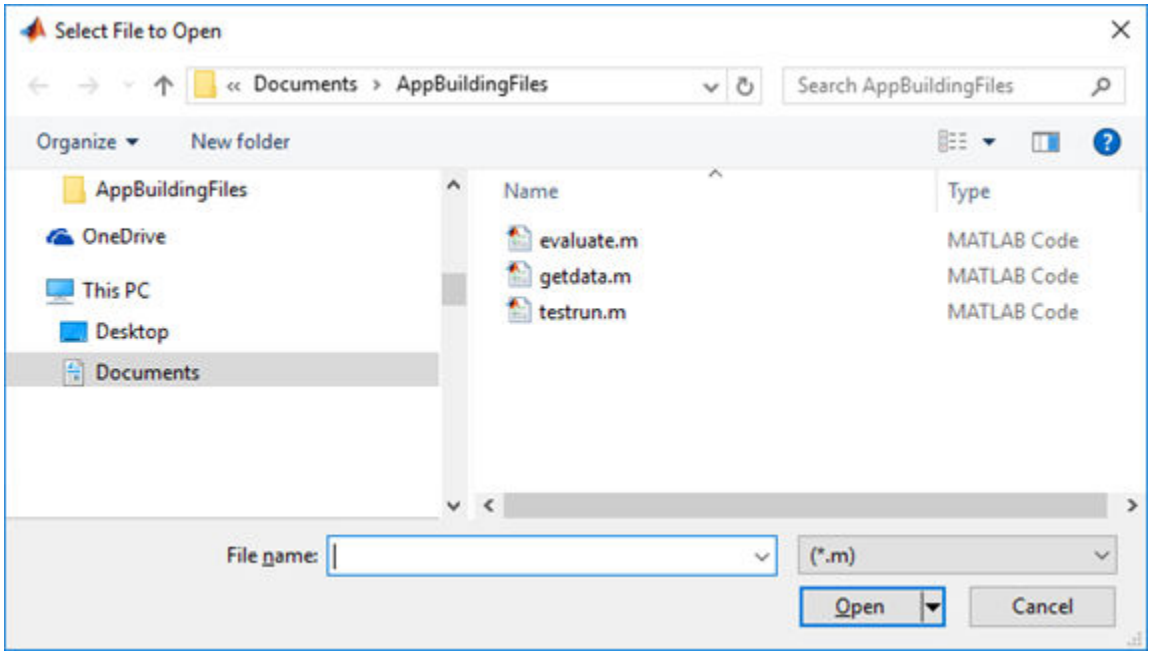



```
User selected H:\Documents\MyCode\peaks.fig and filter index: 3
```

Filter Files by Extension

Display only files with a `.m` extension in the dialog box by specifying `'*. m'` as the `filter` input argument.

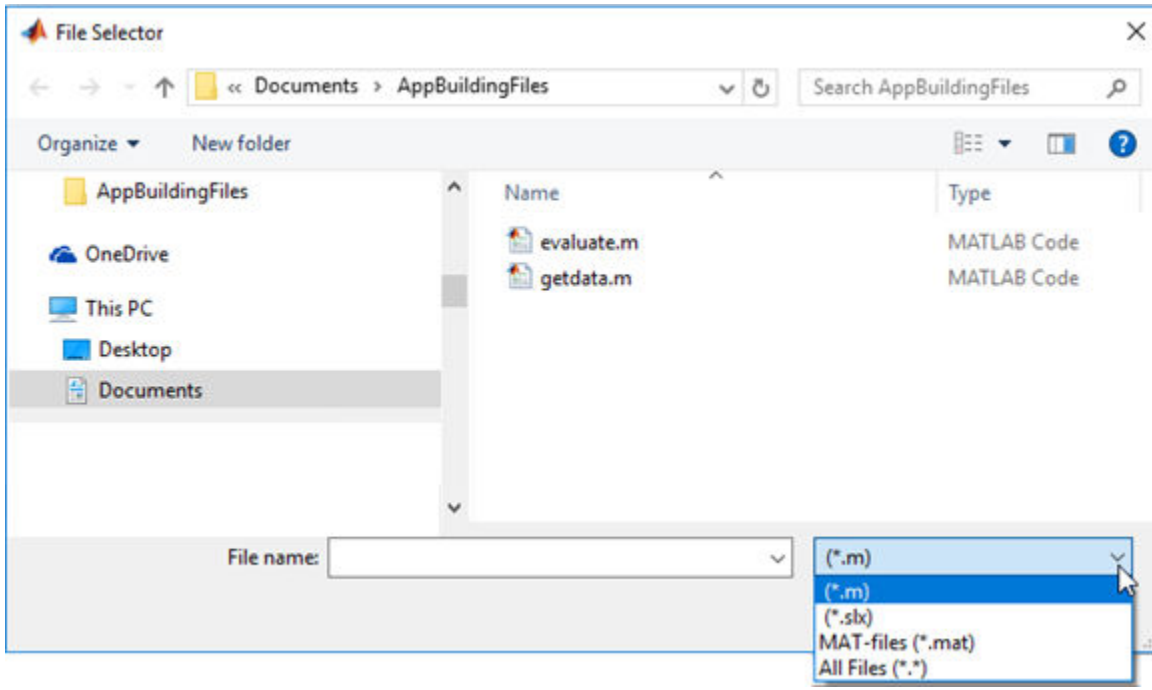
```
[file,path] = uigetfile('*.m');
```



Specify Filter List and Dialog Box Title

Create a list of file extensions in the file filter drop-down list. Pass the `filter` input argument as a cell array of character vectors and separate the file extensions with semicolons.

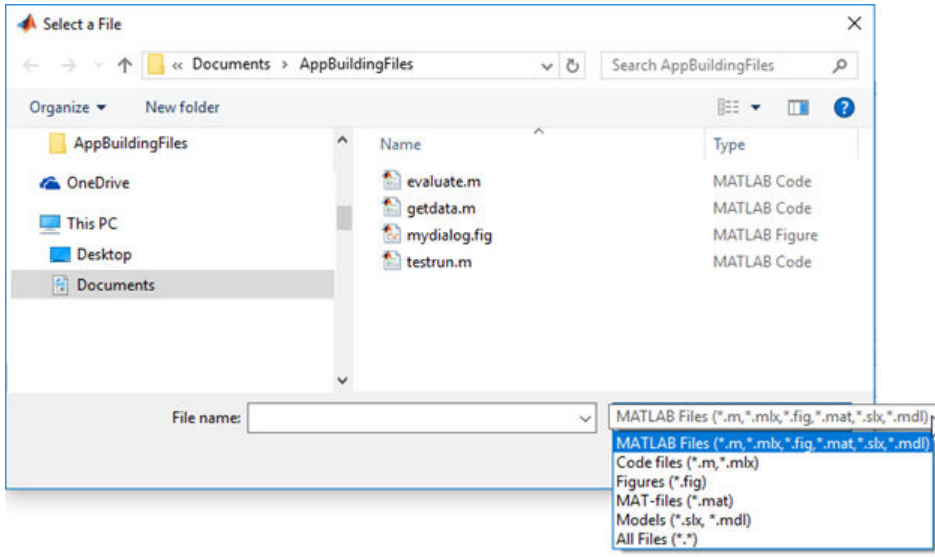
```
[file,path] = uigetfile({'*.m';'*.slx';'*.mat';'*.*'},...
                        'File Selector');
```



Specify Filters and Filter Descriptions

Create a list of file extensions and give them descriptions by passing the `filter` input argument as a cell array of character vectors. The first column of the cell array contains the file extensions, and the second contains custom descriptions of the file types. This example also associates multiple file types with the `'MATLAB Files'` and `'Models'` descriptions.

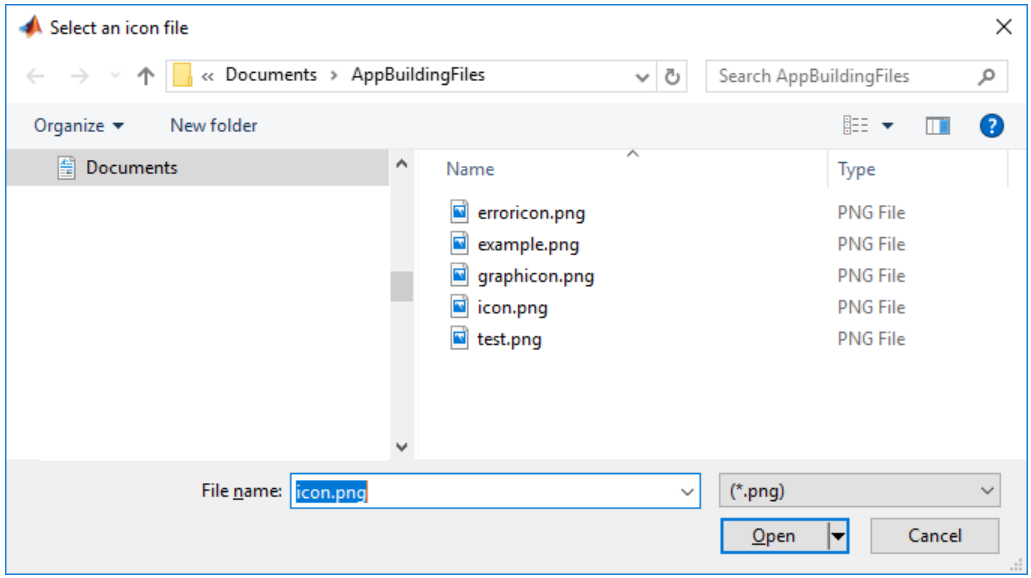
```
[file,path,indx] = uigetfile( ...
{'*.m;*.mlx;*.fig;*.mat;*.slx;*.mdl',...
    'MATLAB Files (*.m,*.mlx,*.fig,*.mat,*.slx,*.mdl)';
   '*.m;*.mlx','Code files (*.m,*.mlx)'; ...
   '*.fig','Figures (*.fig)'; ...
   '*.mat','MAT-files (*.mat)'; ...
   '*.mdl;*.slx','Models (*.slx, *.mdl)'; ...
   '*.*',  'All Files (*.*)'}, ...
   'Select a File');
```



Specify Default File Name

To display a default file name in the **File name** field when the dialog box opens, pass the file name as the `defname` input argument

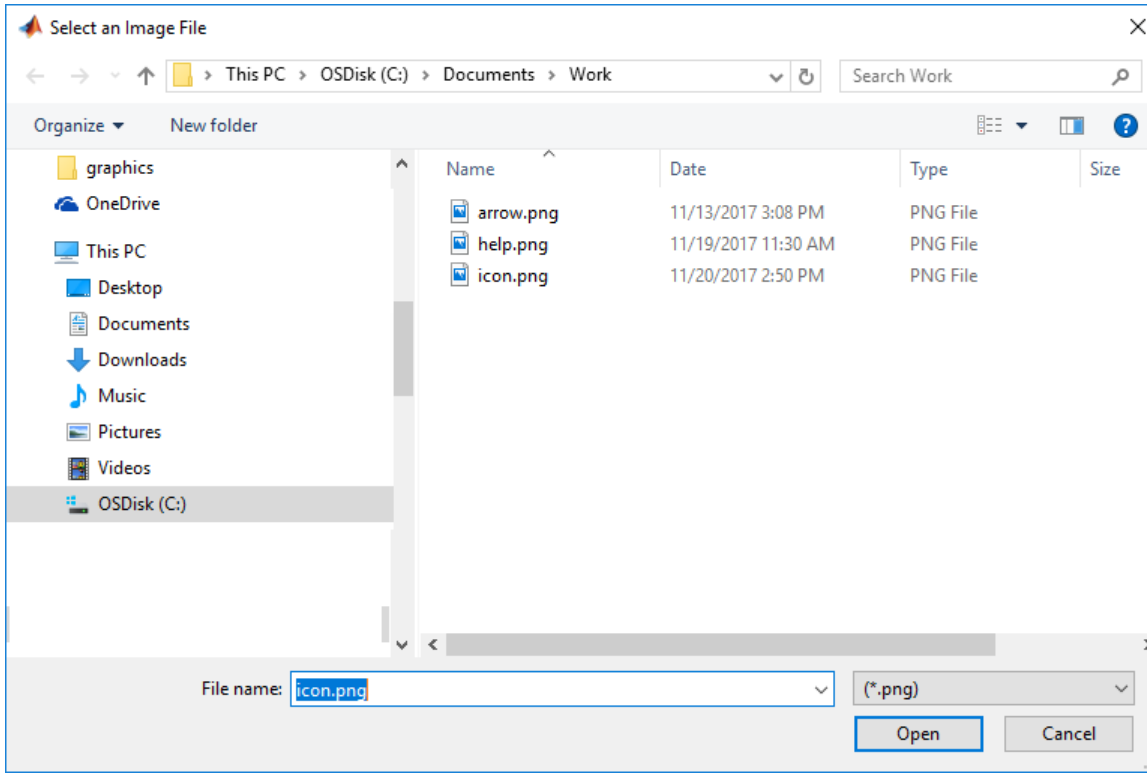
```
[file,path] = uigetfile('*.png',...
              'Select an icon file','icon.png')
```



Specify Default Path and File

To display a default path and file name in the **File name** field when the dialog box opens, pass the full file name as the `defname` input argument.

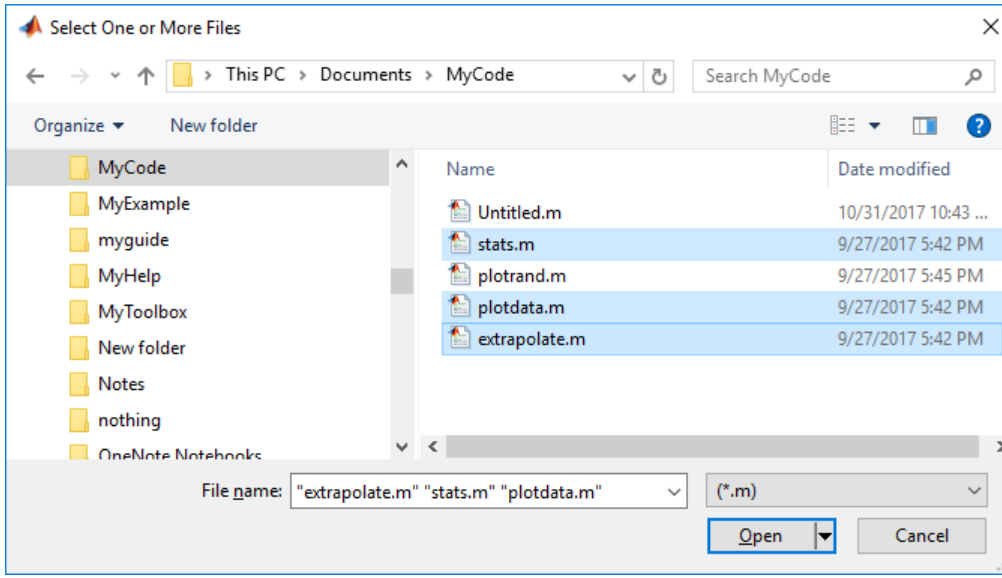
```
[file,path] = uigetfile('C:\Documents\Work\icon.png',...
                        'Select an Image File')
```



Enable Multifile Selection

Enable multifile selection by setting the `'Multiselect'` option to 'on'. Users can select multiple files by holding down the **Shift** or **Ctrl** key and clicking file names.

```
[file,path] = uigetfile('*.m',...
   'Select One or More Files', ...
   'MultiSelect', 'on');
```



Input Arguments

`filter` — File filter

character vector (default) | character vector | cell array of character vectors | string array

File filter, specified as a character vector, cell array of character vectors, or a string array.

- If `filter` is a file name, then that file name appears in the **File name** field. The extension of the file is the default filter value. (The filter field is unlabeled and appears to the right of the **File name** field.)
- The `filter` can include a path. That path can contain the following characters:
  - `.`
  - `..`
  - `\`
  - `/`
  - `~`

For example, `'../*.m'` lists all code files with a `.m` extension in the folder above the current folder.

- If you or a user includes either an asterisk (`*`) or a question mark (`?`) in a file name, then `uigetfile` does not respond to clicking **Open**. The dialog box remains open until the user clicks **Cancel** or removes the wildcard characters from the name. This restriction applies to all platforms, even to file systems that permit these characters in file names
- If the specified path does not exist, then `uigetfile` opens the dialog box in the current folder.
- If `filter` is a folder name, then MATLAB displays the contents of that folder. The **File name** field is empty, and no filter applies. To specify a folder name, the last character of `filter` must be either a backslash (`\`) or a slash (`/`).
- If `filter` is a cell array of character vectors or a string array, it can include two columns. The first column contains a list of file extensions. The optional second column contains a corresponding list of descriptions. These descriptions replace standard descriptions in the filter field. A description cannot be empty.

Example: `'myfile.m'`

Example: `'../myfile.m'`

Example: `'../..'`

`title` — Dialog box title

`'Select File to Open'` (default) | character vector

Dialog box title, specified as a character vector.

Example: `'Select a File'`

`defname` — Default File name field value

`' '` (default) | character vector | string scalar

Default **File name** field value, specified as a character vector or a string scalar. The `defname` value can specify a path, or a path and a file name.

- If you specify a path, it can contain the following characters:
  - `.`
  - `..`
  - `\`

  - `/`
  - `~`
- To specify a folder name only, make the last character of `DefaultName` either a backslash (`\`) or a slash (`/`).

Example: `'myfile.mat'`

Example: `'C:\Documents\my_MATLAB_files'`

Example: `'..\myfile.mat'`

Example: `'..\Documents\'`

`mode` — Multiselect mode
'off' (default) | 'on'

Multiselect mode, specified as `'on'` or `'off'`. If multiselect mode is off, then a user can select one file only. If multiselect mode is on, then a user can select multiple files. If a user selects multiple files, then they must be in the same folder; otherwise MATLAB displays a warning dialog box. Microsoft Windows libraries can span multiple folders.

Output Arguments

`file` — File name
character vector | cell array of character vectors

File name that the user specified in the dialog box, returned as a character vector or a cell array of character vectors.

A cell array of character vectors is returned when `'MultiSelect'` is set to `'on'` and a user selects multiple files. Each array element contains the name of a selected file. File names in the cell array are sorted in the order that the user's platform uses. If a user selects multiple files, they must be in the same folder, otherwise MATLAB displays a warning dialog box.

If the user clicks the **Cancel** button or the window close button (X), then MATLAB returns the file value as `0`.

`path` — Path to specified file
character vector

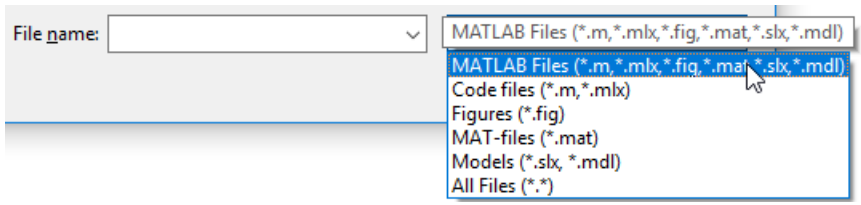
Path to the specified file or files, returned as a character vector.

If the user clicks the **Cancel** button or the window close button (X), then MATLAB returns the file value as `0`.

`indx` — Selected filter index
integer

Selected filter index, returned as an integer.

The filter is the unlabeled dialog box control to the right of the **File name** field in the dialog box. The filter index value corresponds to the item selected in the filter drop-down list. The index of the first row is 1.



If the user clicks the **Cancel** button or the window close button (X), then MATLAB returns an index value of `0`.

More About

Modal Dialog Box

A modal dialog box prevents a user from interacting with other MATLAB windows before responding to the modal dialog box.

Tips

- Use the path and file name that `uigetfile` returns to open, read, or analyze the file using various input and output functions in MATLAB and MATLAB toolboxes. For example: listed here.
  - `imread` for reading images.
  - `xlsread` for reading Microsoft Excel files.

- `open`, `edit`, or `run` with MATLAB code files. For example, this code creates a dialog box to get a MATLAB code file name from the user, builds a full file name from the returned values, and then runs the user-specified code file.

  ```
  [file,path] = uigetfile('*.m');
  selectedfile = fullfile(path,file);
  run(selectedfile);
  ```

Alternative Functionality

Use the `dir` function to return a filtered or unfiltered list of files in your current folder or a folder you specify. The `dir` function can return file attributes too.

See Also

`uigetdir` | `uiopen` | `uiputfile`

Introduced before R2006a

uigetpref

Create dialog box that opens according to user preference

Syntax

```
pval = uigetpref(group,pref,title,quest,pbtns)
[pval,tf] = uigetpref(group,pref,title,quest,pbtns)
[ ___ ] = uigetpref( ___ ,Name,Value)
```

Description

`pval = uigetpref(group,pref,title,quest,pbtns)` creates a nonmodal on page 1-16123 dialog box that opens with the specified group name and preference name. The group and preference names, in combination, uniquely identify the dialog box. The dialog box contains:

- The specified question (`quest`) and one or more buttons (`pbtns`) that enable the dialog box user to answer the question. The clicked button value is returned as `pval`.
- A check box that, by default, is labeled **Do not show this dialog again**.

If the user selects the check box, MATLAB stores and returns the value of the clicked button as the preference value (`pval`). Subsequent calls to `uigetpref` for the same dialog box, detect that the preference value was stored and apply that choice without opening the dialog box.

If the user does not select the check box, then MATLAB returns, but does not store the value of the clicked button. Instead, MATLAB stores the value `'ask'`. MATLAB opens the specified dialog box on subsequent calls to `uigetpref`.

`[pval,tf] = uigetpref(group,pref,title,quest,pbtns)` returns a logical value that indicates whether the dialog box opened. If the dialog box opened, then the value of `tf` is `1` (true). Otherwise, the value of `tf` is `0` (false).

`[ ___ ] = uigetpref( ___ ,Name,Value)` specifies one or more optional name-value pairs that enable you to do any of the following.

- Customize the check box label.
- Specify whether the check box is selected when the dialog box opens.
- Provide a help button and the help button callback.
- Specify buttons that are not mapped to preference values.
- Specify the value that `uigetpref` returns for `pval` if the user closes the dialog box without clicking a preference button. For example, this value is returned if the user clicks the dialog box close button (X), presses the keyboard **Esc** key, or clicks a button that is not mapped to a preference value.

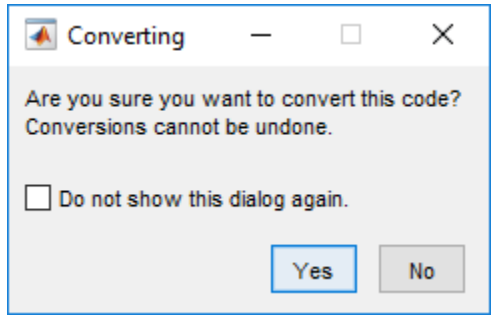
Use this option with any of the output argument combinations in the previous syntaxes.

Examples

Create Basic Dialog Box

Define each of the required `uigetpref` input arguments, and then pass them to `uigetpref`.

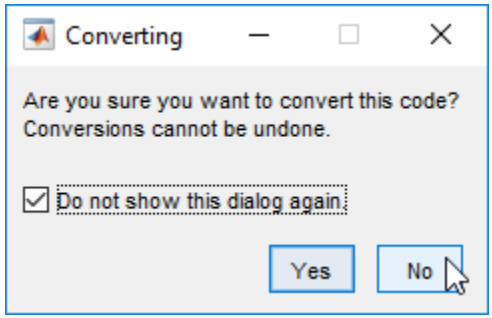
```
group = 'Updates';
pref =  'Conversion';
title = 'Converting';
quest = {'Are you sure you want to convert this code?',...
         'Conversions cannot be undone.'};
pbtns = {'Yes','No'};

[pval,tf] = uigetpref(group,pref,title,quest,pbtns)
```



Click **Yes**. The MATLAB Command Window shows that the value of `pval` is `'yes'` and that value of `tf` is `1`, indicating that the dialog box was displayed.

Run the `uigetpref` command again, but this time select **Do not show this dialog again**, and then click **No**.

```
[pval,tf] = uigetpref(group,pref,title,quest,pbtns)
```



The MATLAB Command Window shows that the value of `pval` is `'no'` and that value of `tf` is `1`.

Run the `uigetpref` command again.

```
[pval,tf] = uigetpref(group,pref,title,quest,pbtns)
```

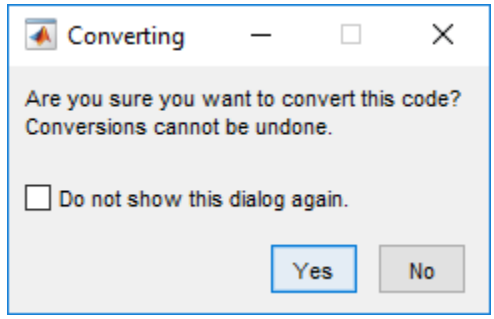
As expected, the dialog box does not display. The MATLAB Command Window shows that the value of `pval` is `'no'` and that value of `tf` is `0`.

Reenable the dialog box display by setting the preference value to `'ask'`.

```
setpref('Updates','Conversion','ask');
```

Run the `uigetpref` command again. The dialog box opens.

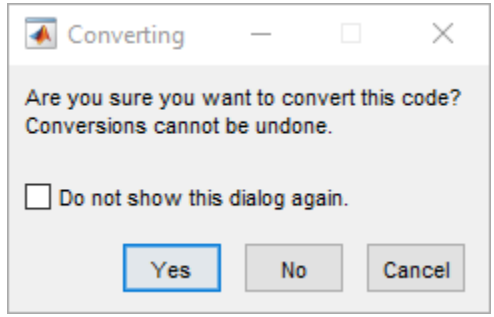
```
[pval,tf] = uigetpref(group,pref,title,quest,pbtns)
```



Add Cancel Button to Dialog Box

Specify `'ExtraOptions','Cancel'` as a name-value pair to add a Cancel button to the dialog box. If the user clicks **Cancel**, MATLAB returns the button label to `pval`.

```
group = 'Updates';
pref =  'Conversion';
title = 'Converting';
quest = {'Are you sure you want to convert this code?',...
         'Conversions cannot be undone.'};
pbtns = {'Yes','No'};
[pval] = uigetpref(group,pref,title,quest,pbtns,...
'ExtraOptions','Cancel');
```



Get Preferences in Figure `CloseRequestFcn` Callback

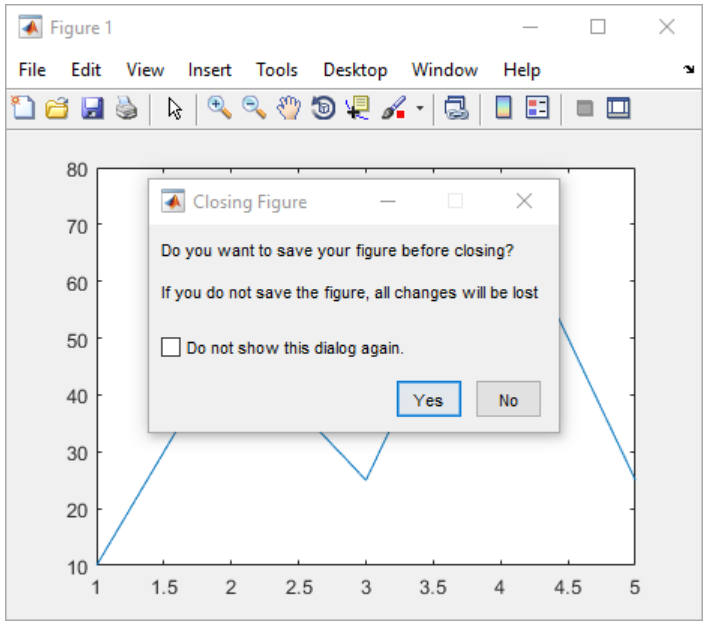
Create a function that creates a preferences dialog box. The dialog box asks the user about saving the figure before closing it. Based on the value of the button that the user clicks, the function opens a Save dialog box or closes the figure without saving it.

```
function savefigconditionally
fig = gcf;

group ='mygraphics';
pref = 'savefigbeforeclosing';
title = 'Closing Figure';
quest = {'Do you want to save your figure before closing?'
         ''
         'If you do not save the figure, all changes will be lost'};
pbtns = {'Yes','No'};
[pval,tf] = uigetpref(group,pref,title,quest,pbtns);

switch pval
    case 'yes'
    [file,path,indx] = uiputfile('fig', ...
                                 'Save current figure', ...
                                 'untitled.fig');
       if indx == 0
           delete(fig);
       else
           saveas(fig,[path,file])
           delete(fig);
       end
   case 'no'
       delete(fig);
       return
 end
```

To run this example, copy and paste the code into a new program file. Name the file `savefigconditionally.m` and save it on your search path. To use the function as a `CloseRequestFcn` callback, create a figure and optionally plot some data.

```
figure('CloseRequestFcn','savefigconditionally');
x = [1 2 3 4 5];
y = [10 50 25 75 25];
plot(x,y);
```

Each time you run the preceding block of commands and click the close button (X) in the figure title bar, the dialog box opens unless you select **Do not show this dialog again**.



Input Arguments

`group` — Preference group name
character vector | string scalar

Preference group name, specified as a character vector or string scalar. The group includes the preference specified by the `pref` input argument. If the group does not exist, MATLAB creates it.

Example: `'My Graphics'`

`pref` — Preference name
character vector | string scalar

Preference name, specified as a character vector or string scalar.

This preference stores the value of the button within the specified `pbtns` that the user clicks. If the preference name does not exist, then MATLAB creates it.

Example: `'Save Graphic'`

`title` — Dialog box title
character vector | string scalar

Dialog box title, specified as a character vector or string scalar.

Example: `'Save preference'`

`quest` — Dialog box question
character vector | cell array of character vectors | string array

Dialog box question, specified as a character vector, a cell array of character vectors, or a string array. Line breaks occur within the question text as follows:

- If the question is specified as a character vector, then line breaks occur after a vertical bar (`|`) character or a newline character specified with the `newline` function.
- If the question is specified as a cell array of character vectors, then line breaks occur after each cell array element.

Example: `{'Are you sure you want to convert this code?', 'Conversions cannot be undone.'}`

Example: `'Do you want to save this file before closing?'`

`pbtns` — Preference button labels
character vector | cell array of character vectors | vertical bar-delimited character vector | string array

Preference button labels, specified as a character vector, a cell array of character vectors, a vertical bar-delimited character vector, or a string array.

If you want to specify internal preference values that are different from the button labels, then specify the `pbtns` value as a 2-by-*n* cell array or string array. The first row contains the preference names and the second row contains the associated button labels. For example, consider using this approach if you plan to localize your dialog box for various languages. You can specify the button labels using a foreign language without having to change your code logic (for instance, `switch` and `case` statements) for each localization.

When `pbtns` is not a 2-by-*n* cell array, MATLAB stores the label name as the preference value.

Example: `'Yes'`

Example: `{'Yes','No'}`

Example: `['Yes|No']`

Example: `{'Oui','Non';'yes','no'}` sets the button labels to `'Oui'` and `'Non'` and their corresponding preference values to `'yes'` and `'no'`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: line(x,y,'Color','red','LineWidth',3) creates a red line that is 3 points wide.

`CheckboxState` — Initial state of check box

0 (default) | true | false | 1 | 0

Initial state of check box when the dialog box opens, specified as the comma-separated pair consisting of `'CheckboxState'` and `1`,`0`, `true`, or `false`.

- The check box is selected when the value is `true` or `1`
- The check box is not selected when the value is `false` or `0`.

Example: `'CheckboxState',0`

`CheckboxString` — Check box label

`'Do not show this dialog again.'` (default) | character vector | string scalar

Check box label, specified as the comma-separated pair consisting of `'CheckboxString'` and a character vector or string scalar.

Example: `'CheckboxSting','Do not ask again.'`

`HelpString` — Help button label

character vector | string scalar

Help button label, specified as the comma-separated pair consisting of `'Help'` and a character vector or a string scalar. If you do not specify this name-value pair, no help button is displayed in the dialog box. If you specify this name-value pair, but do not specify the `HelpFcn` name-value pair, then MATLAB uses `'HelpFcn','doc(uigetpref)'` by default.

Example: `'HelpString','Info'`

`HelpFcn` — Help button callback
function handle | string array | character vector

Help button callback, specified as the comma-separated pair consisting of `'HelpFcn'` and a function handle, string array, or character vector (not recommended). The callback executes when a user clicks the help button. When you specify this name-value pair, you also must specify the `'HelpString'` name-value pair.

See also, “Write Callbacks for Apps Created Programmatically”

Example: `'HelpFcn',@myfunc` specifies the `HelpFcn` callback function as a function handle.

`ExtraOptions` — Labels for extra buttons
{ } (default) | character vector | cell array of character vectors | string array

Labels for extra buttons, specified as the comma separated pair consisting of `'ExtraOptions'` and a character vector, cell array of character vectors, or a string array. The additional buttons are not mapped to any preferences. If a user clicks any of these buttons, the dialog box closes and returns the button label as the output argument `pval`.

Example: `'ExtraOptions','Cancel'`

`DefaultButton` — Default button selection
character vector | string scalar

Default button selection, specified as the comma separated pair consisting of `'DefaultButton'` and a character vector or string scalar. The default button selection is returned to `p` if a user closes the dialog box without clicking any button. This value does not have to correspond to any preference button or an `ExtraOption` button.

Example: `'nobtn'`

Output Arguments

`pval` — Selected preference button
string scalar

Selected preference button returned as a string scalar. The returned value is one of the following:

- Label of the clicked preference button (`pbtns`)
- Internal value of the clicked preference button

`tf` — True or false result
1 | 0

True or false result, returned as 1 or 0. The function returns `1` (true) if the dialog box opened. Otherwise, it returns `0`. This value corresponds to the check box selection stored the last time the dialog box was open and the user selected the check box (which is labeled `'Do not show this dialog again'` by default).

More About

Nonmodal Dialog Box

A nonmodal dialog box enables a user to interact with other MATLAB windows before responding to the nonmodal dialog box. A nonmodal dialog box is also referred to as `normal`.

Preferences

Preferences enable you to specify how applications behave and how users interact with them. Preferences persist across sessions and are stored in a preference data base.

The `uigetpref` function uses the same preference data base as MATLAB built-in products. However, `uigetpref` registers the preferences it sets as a separate list, so that it and `uisetpref` can manage those preferences.

To modify preferences registered with `uigetpref`, use `uisetpref` or `setpref`. For example, use `setpref` to change a preference value to `'ask'`.

Tips

- `uigetpref` creates specified groups and preferences, if they do not currently exist. To delete a preference group you no longer need, use `rmpref`.
- To get a structure of previously created groups and preferences, use the `getpref` function.

- After a user selects the check box **Do not show this dialog again** and closes the dialog box, the dialog box does not open again for the same group and preference. To reenable dialog boxes that are being suppressed set the preference value to `'ask'` using `setpref`.
- Users of your dialog box do not know the group and preference names you specified when creating the dialog box. Therefore, to reenable dialog boxes that are being suppressed by preferences, users can call the `uisetpref` command.

  ```
  uisetpref('clearall')
  ```

  Executing `uisetpref` as shown reenables *all* preference dialog boxes defined with `uigetpref`, not just the most recent one.

See Also

`addpref` | `getpref` | `ispref` | `rmpref` | `setpref` | `uisetpref`

Introduced before R2006a

uigridlayout

Create grid layout manager

Syntax

```
g = uigridlayout
g = uigridlayout(parent)
g = uigridlayout( ___ ,sz)
g = uigridlayout( ___ ,Name,Value)
```

Description

`g = uigridlayout` creates a grid layout manager for an app. It positions UI components along the rows and columns of an invisible grid that spans the entire figure or a container within the figure. This command creates a 2-by-2 grid layout in a new figure and returns the `GridLayout` object. MATLAB calls the `uifigure` function to create the figure. Use a grid layout manager if you do not want to position components by setting pixel values in `Position` vectors.

If you add components to the grid layout manager, but you do not specify the `Layout` property of the components, then the grid layout manager adds the components from left to right and top to bottom.

`g = uigridlayout(parent)` creates a grid layout in the specified parent container. The parent can be a figure created using the `uifigure` function, or one of its child containers.

`g = uigridlayout( ___ ,sz)` specifies the size of the grid as the vector `sz`. The first element in this vector is the number of rows, and the second element is the number of columns. Specify the `sz` argument by itself or after the `parent` argument.

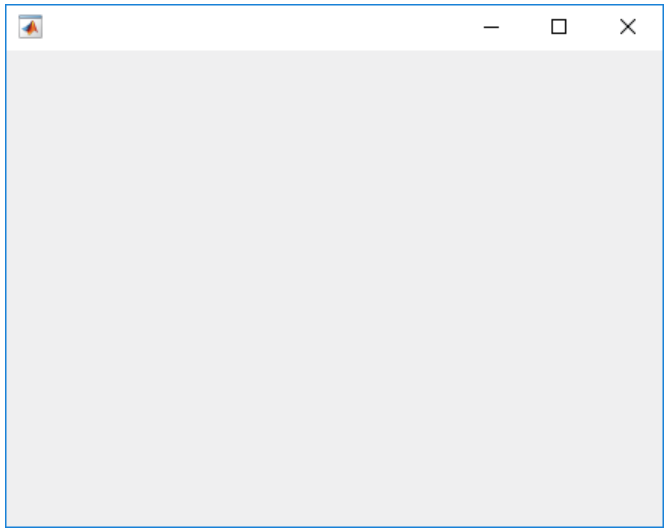
`g = uigridlayout( ___ ,Name,Value)` specifies `GridLayout` property values using one or more name-value pair arguments. Specify the name-value pair arguments after all other arguments in any of the previous syntaxes.

Examples

Simple 3-by-2 Grid

Create a figure and a grid. Then configure the grid by setting the row heights and column widths. In this case, configure a 3-by-2 grid in which the top two rows are fixed at 22 pixels, while the bottom row has a variable height. Set the left column to be 150 pixels wide, and set the right column to have a variable width. The grid fills the entire figure, but because the grid is invisible, the figure appears to be empty.

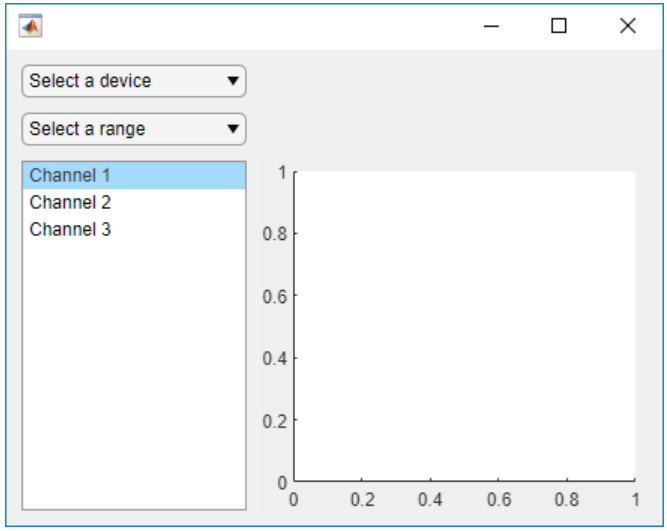
```
fig = uifigure('Position',[100 100 440 320]);
g = uigridlayout(fig);
g.RowHeight = {22,22,'1x'};
g.ColumnWidth = {150,'1x'};
```



Add two drop-downs, a list box, and an axes component to the grid. If you do not specify row or column locations for the `Layout` property of components, they populate the grid from left to right and top to bottom by default. In this case, move the second drop-down (`dd2`) and the list box (`chanlist`) to specific rows and columns by setting the `Layout` property.

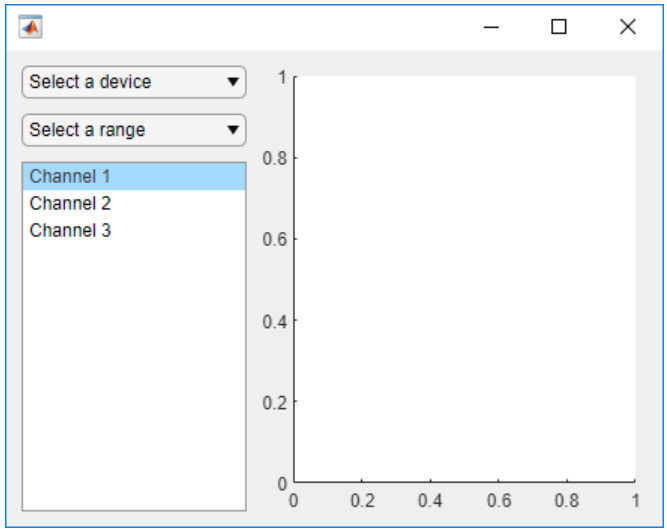
```
% Device drop-down
dd1 = uidropdown(g);
```

```
dd1.Items = {'Select a device'};

% Range drop-down
dd2 = uidropdown(g);
dd2.Items = {'Select a range'};
dd2.Layout.Row = 2;
dd2.Layout.Column = 1;

% List box
chanlist = uilistbox(g);
chanlist.Items = {'Channel 1','Channel 2','Channel 3'};
chanlist.Layout.Row = 3;
chanlist.Layout.Column = 1;

% Axes
ax = uiaxes(g);
```

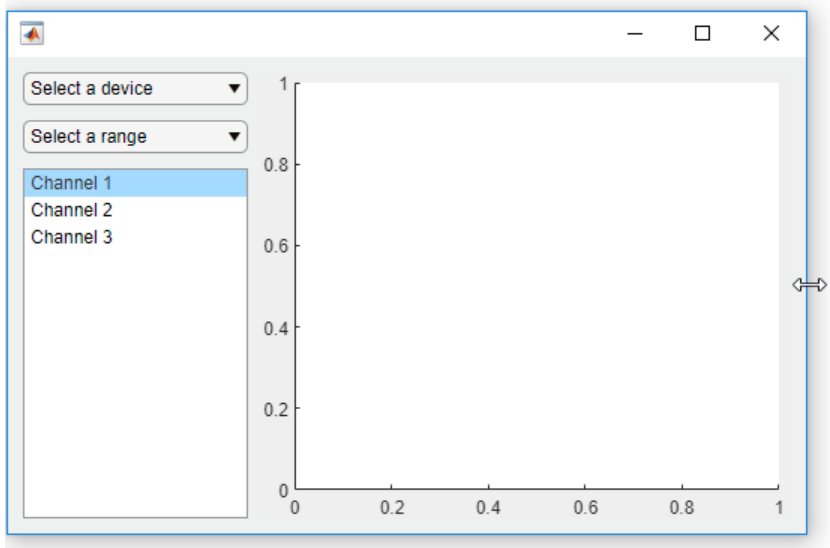


Set the axes to span rows 1 through 3, filling the vertical space.

```
ax.Layout.Row = [1 3];
```



When you resize the figure, the axes grow and shrink to fill the available space that the left column does not use. The components on the left side remain fixed because that column is defined as `150` pixels wide.

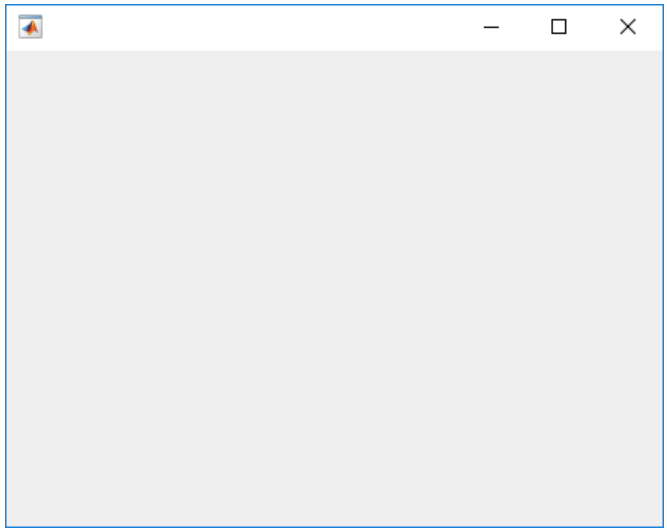


Variable-Width Columns That Have Different Weights

When you assign different weights to variable-width columns, the width of each column is proportional to the weight. Variable-height rows allocate space the same way.

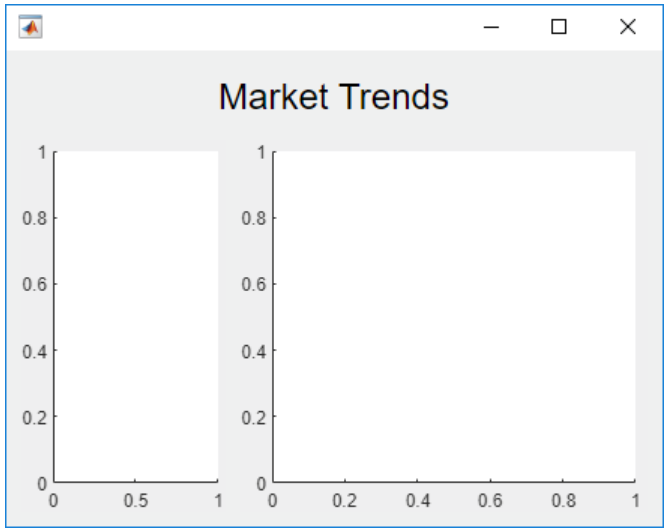
Create a figure and a 2-by-2 grid. Then configure the row heights and the column widths. In this case, the top row is fixed at 40 pixels, while the bottom row has a variable height. Both columns have a variable width, but the second column has twice the weight of the first column. Because the grid is invisible, the figure appears to be empty.

```
fig = uifigure('Position',[100 100 437 317]);
g = uigridlayout(fig,[2 2]);
g.RowHeight = {40,'1x'};
g.ColumnWidth = {'1x','2x'};
```



Add a label across both columns of the top row. Then add an axes component to each column of the bottom row.

```
% Add title
title = uilabel(g,'Text','Market Trends');
title.HorizontalAlignment = 'center';
title.FontSize = 24;
title.Layout.Row = 1;
title.Layout.Column = [1,2];

% Add two axes
ax1 = uiaxes(g);
ax2 = uiaxes(g);
```



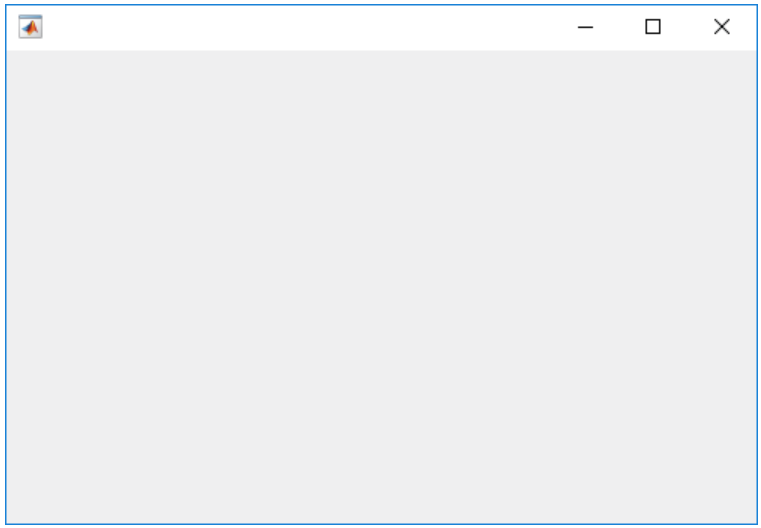
Because the left column has a weight of 1, and the right column has a weight of 2, the axes on the right use twice as much of the horizontal space as the axes on the left. The grid maintains this ratio when you resize the figure.

Figure and Panel That Each Have a Grid

One way to arrange UI components in a panel is to create a grid in the panel.

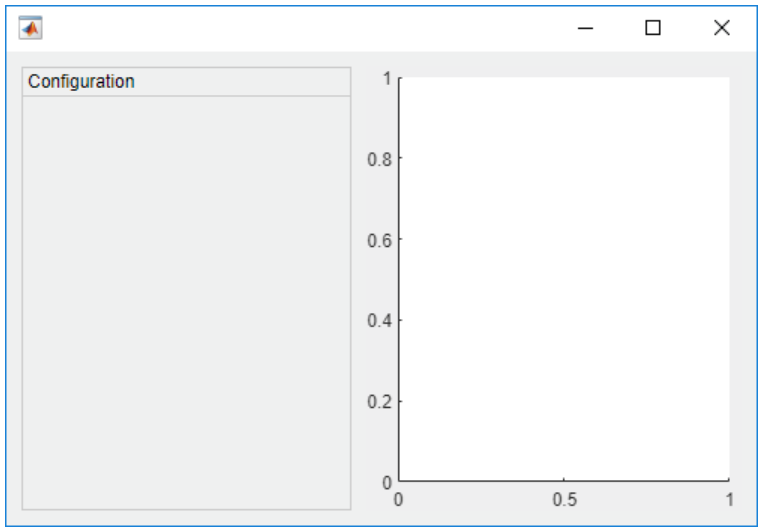
Create a figure and a 1-by-2 grid. Then configure the column widths. In this case, the left column is fixed at 220 pixels, while the right column has a variable width. Because the grid is invisible, the figure appears to be empty.

```
fig = uifigure('Position',[100 100 500 315]);
grid1 = uigridlayout(fig,[1 2]);
grid1.ColumnWidth = {220,'1x'};
```



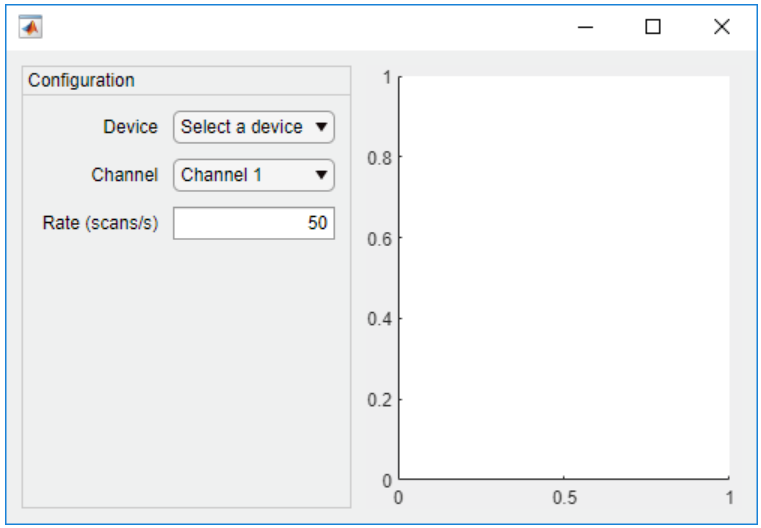
Add a panel and axes to `grid1`.

```
p = uipanel(grid1,'Title','Configuration');
ax = uiaxes(grid1);
```

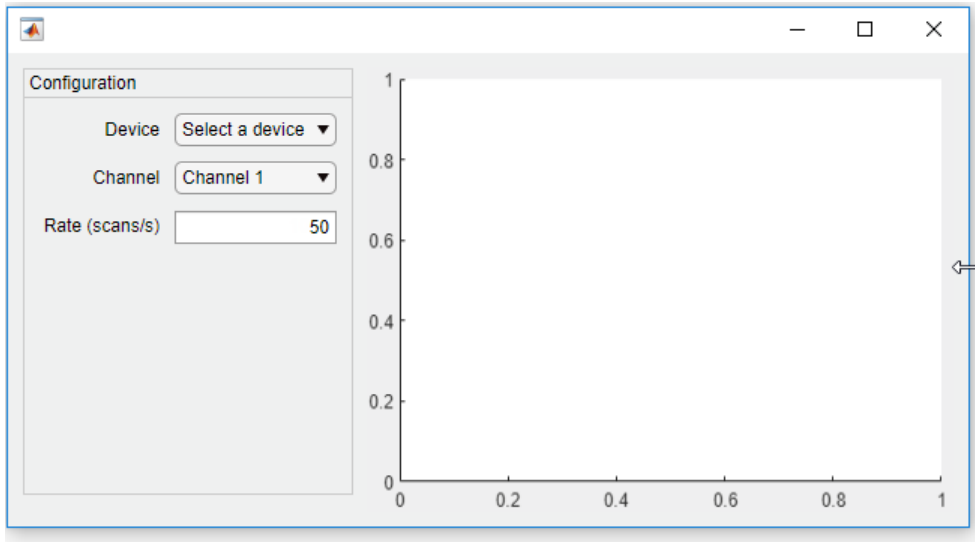


Create a grid called `grid2` inside the panel. Then place three components and three labels inside `grid2`. The left column of `grid2` aligns the labels with the components. Since each row uses only 22 pixels, the panel has extra space below the third row.

```
% Grid in the panel
grid2 = uigridlayout(p,[3 2]);
grid2.RowHeight = {22,22,22};
grid2.ColumnWidth = {80,'1x'};

% Device label
dlabel = uilabel(grid2);
dlabel.HorizontalAlignment = 'right';
dlabel.Text = 'Device';

% Device drop-down
devicedd = uidropdown(grid2);
devicedd.Items = {'Select a device'};

% Channel label
chlabel = uilabel(grid2);
chlabel.HorizontalAlignment = 'right';
chlabel.Text = 'Channel';

% Channel drop-down
channeldd = uidropdown(grid2);
channeldd.Items = {'Channel 1', 'Channel 2'};

% Rate Label
ratelabel = uilabel(grid2);
ratelabel.HorizontalAlignment = 'right';
ratelabel.Text = 'Rate (scans/s)';

% Rate edit field
ef = uieditfield(grid2, 'numeric');
ef.Value = 50;
```



When you resize the figure, the axes grow and shrink to fill the available space that the left column of `grid1` does not use. The panel remains fixed because that column is defined as 220 pixels wide.

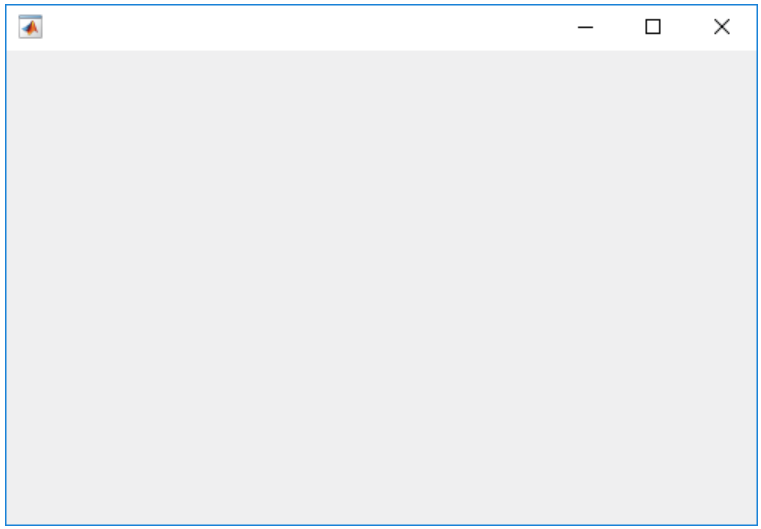


Nested Grid

Nested grids allow you to manage subsets of components. In this case, there are three grids: one grid that covers the entire figure, a second grid that manages a vertical stack of components, and a third grid that manages two buttons that are side by side at the bottom of the vertical stack.

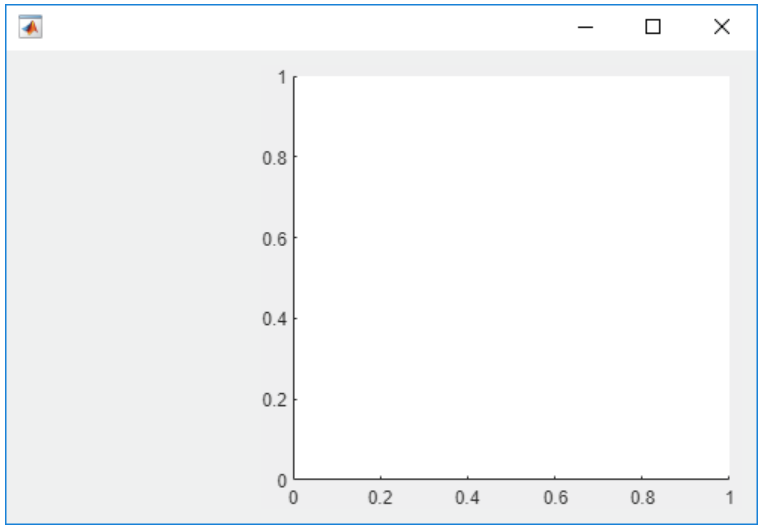
Create a figure and a 1-by-2 grid. Then configure the column widths. In this case, the left column is fixed at `150` pixels, while the right column has a variable width. Because the grid is invisible, the figure appears to be empty.

```
fig = uifigure('Position',[100 100 500 315]);
grid1 = uigridlayout(fig,[1 2]);
grid1.ColumnWidth = {150,'1x'};
```



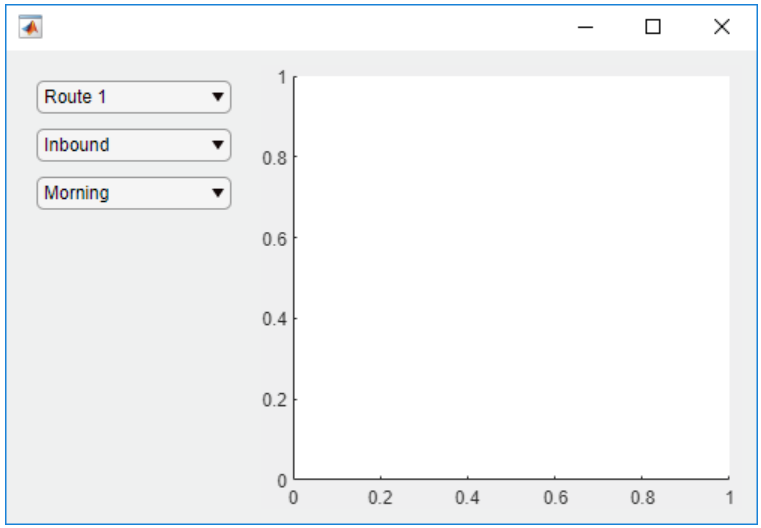
Create a 4-by-1 grid called `grid2` inside the first column of `grid1`. Then add an axes component to the second column of `grid1`.

```
grid2 = uigridlayout(grid1,[4 1]);
grid2.RowHeight = {22,22,22,44};

ax = uiaxes(grid1);
```



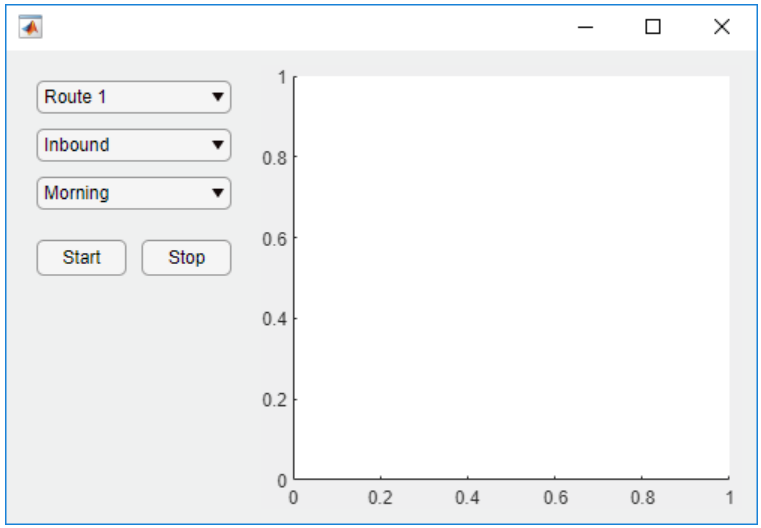
Add three drop-downs to the first three rows of `grid2`.

```
% Routes drop-down
ddRoutes = uidropdown(grid2);
ddRoutes.Items = {'Route 1', 'Route 2'};

% Direction drop-down
ddDirection = uidropdown(grid2);
ddDirection.Items = {'Inbound', 'Outbound'};

% Time drop-down
ddTime = uidropdown(grid2);
ddTime.Items = {'Morning', 'Afternoon'};
```

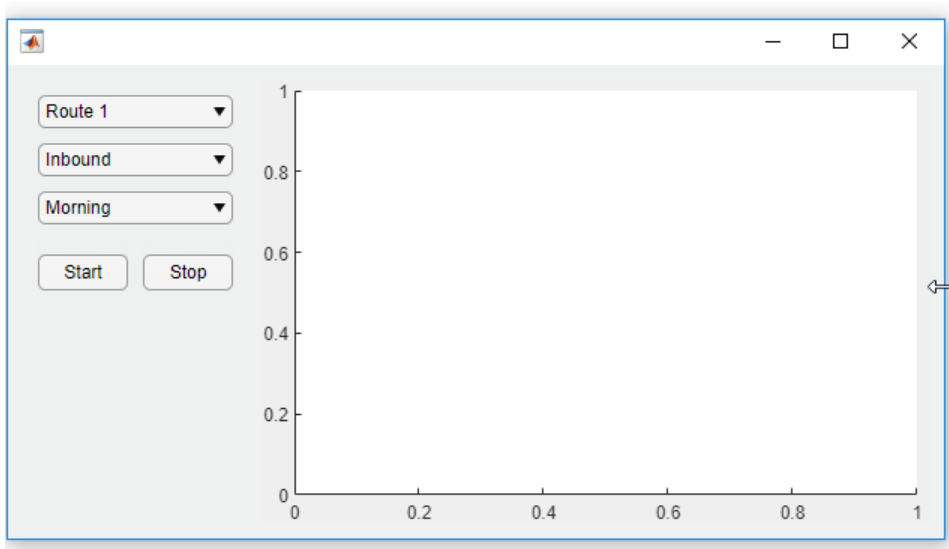


Create a 1-by-2 grid called `grid3` inside the last row of `grid2`. Then add two buttons to `grid3`. Remove the padding on the left and right edges of `grid3` so that the buttons align with the left and right edges of the drop-downs.

```
grid3 = uigridlayout(grid2,[1 2]);
grid3.Padding = [0 10 0 10];
b1 = uibutton(grid3,'Text','Start');
b2 = uibutton(grid3,'Text','Stop');
```



When you resize the figure, the axes grow and shrink to fill the available space that the left column of `grid1` does not use. The components on the left side remain fixed because that column is defined as `150` pixels wide.

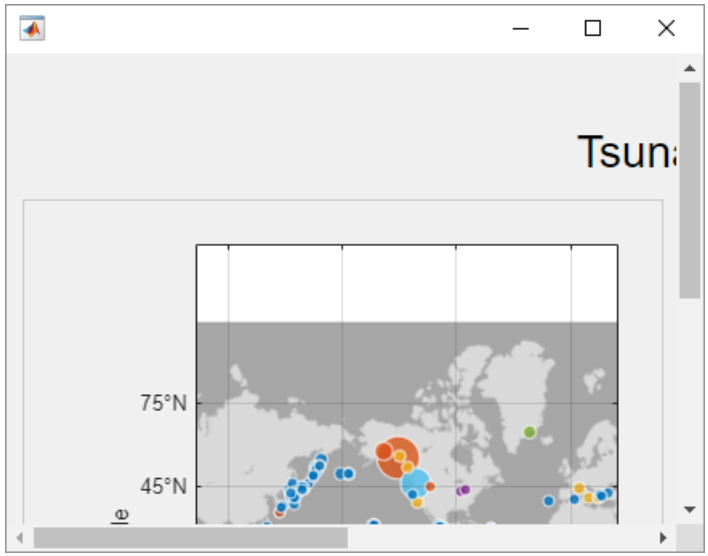


Scroll to Location in Grid

Bring child components of a scrollable grid layout into view by specifying pixel coordinates or a location name.

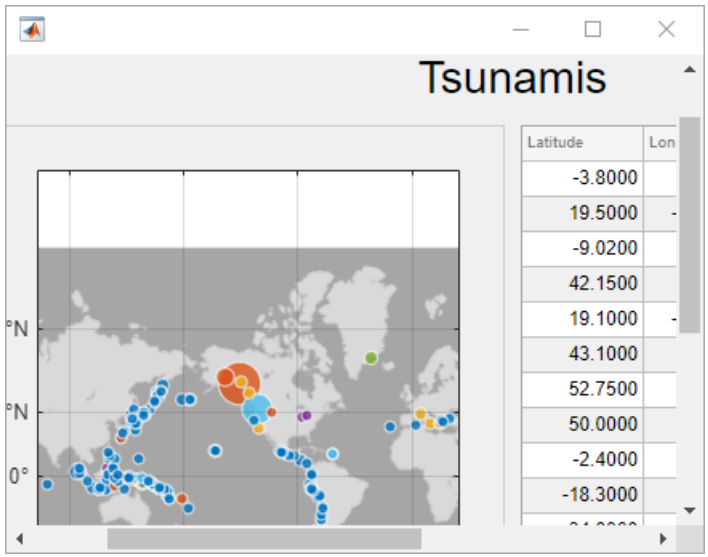
Create a 5-by-2 grid layout and set the `Scrollable` property of the grid to `'on'`. Then add a label, a table, and a panel to the grid. Set the `Scrollable` property of the panel to `'off'` and then add a chart to the panel.

```
fig = uifigure('Position',[782 497 435 311]);
g = uigridlayout(fig,'Scrollable','on');
g.RowHeight = {22,40,22,22,400};
g.ColumnWidth = {400,400};

lbl = uilabel(g,'Text','Tsunamis');
lbl.Layout.Row = 2;
lbl.Layout.Column = [1,2];
lbl.HorizontalAlignment = 'center';
lbl.FontSize = 28;

tsunamis = readtable('tsunamis.xlsx');
tsunamis.Cause = categorical(tsunamis.Cause);
t = uitable(g,'Data',tsunamis);
t.Layout.Row = [3,5];
t.Layout.Column = 2;

p = uipanel(g);
p.Scrollable = 'off';
p.Layout.Row = [3,5];
p.Layout.Column = 1;
gb = geobubble(p,tsunamis.Latitude,tsunamis.Longitude,...
    tsunamis.MaxHeight,tsunamis.Cause);
```



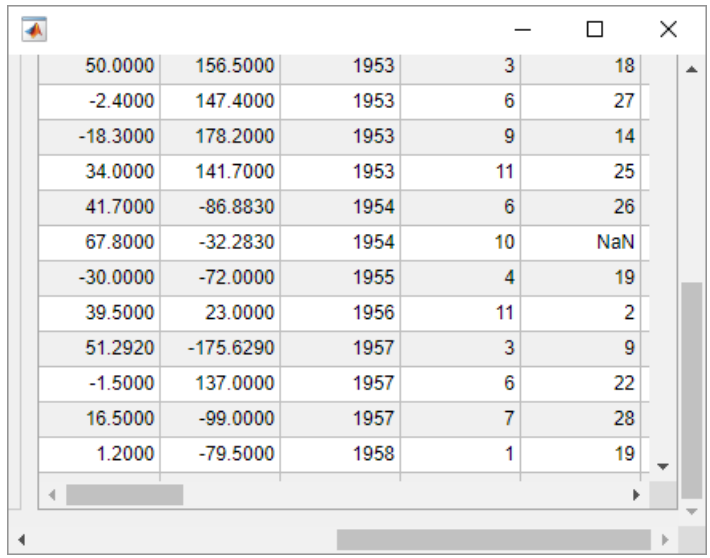
Scroll to a location in the grid.

```
scroll(g,100,-30);
```



Now use location names to scroll to the bottom-right corner of the grid.

```
scroll(g,'bottom','right');
```



50.0000	156.5000	1953	3	18
-2.4000	147.4000	1953	6	27
-18.3000	178.2000	1953	9	14
34.0000	141.7000	1953	11	25
41.7000	-86.8830	1954	6	26
67.8000	-32.2830	1954	10	NaN
-30.0000	-72.0000	1955	4	19
39.5000	23.0000	1956	11	2
51.2920	-175.6290	1957	3	9
-1.5000	137.0000	1957	6	22
16.5000	-99.0000	1957	7	28
1.2000	-79.5000	1958	1	19

Create Grid That Automatically Adjusts to Fit Content

Create a grid layout manager that automatically adjusts its row and column sizes to fit components like labels as you add them to your app, or when the font size of text-based components changes dynamically.

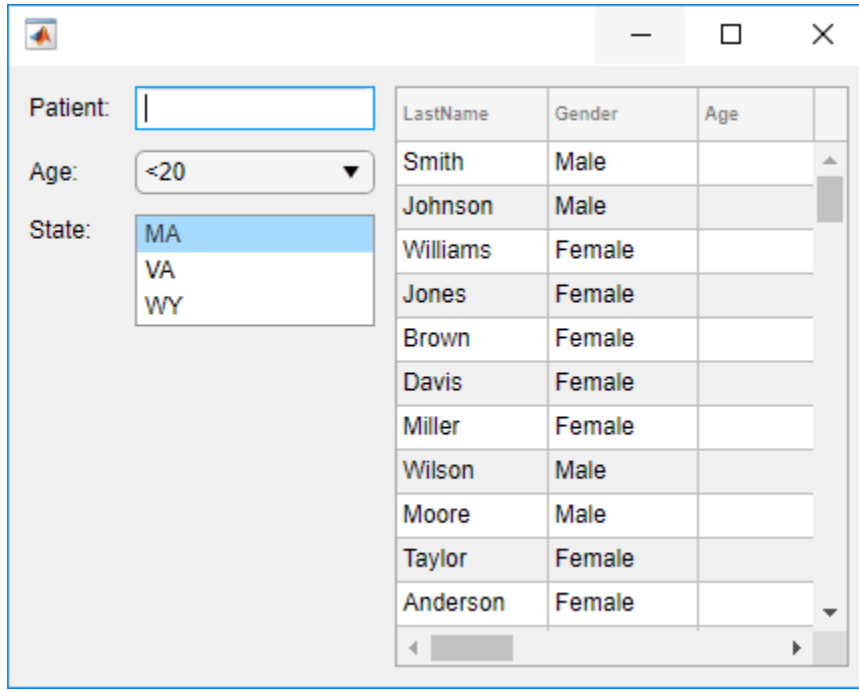
Create a 5-by-2 grid layout manager. Specify `'fit'` for the first column width and the first four row heights.

```
fig = uifigure('Position',[500 500 430 310]);

g = uigridlayout(fig);
g.ColumnWidth = {'fit',120,'1x'};
g.RowHeight = {'fit','fit','fit','fit','1x'};
```

Create a title label that spans all the columns. Then, create labels and controls along the rows of the first and second columns. Lastly, create a table UI component in the last column that spans rows two through five. Notice that as you add components to the rows

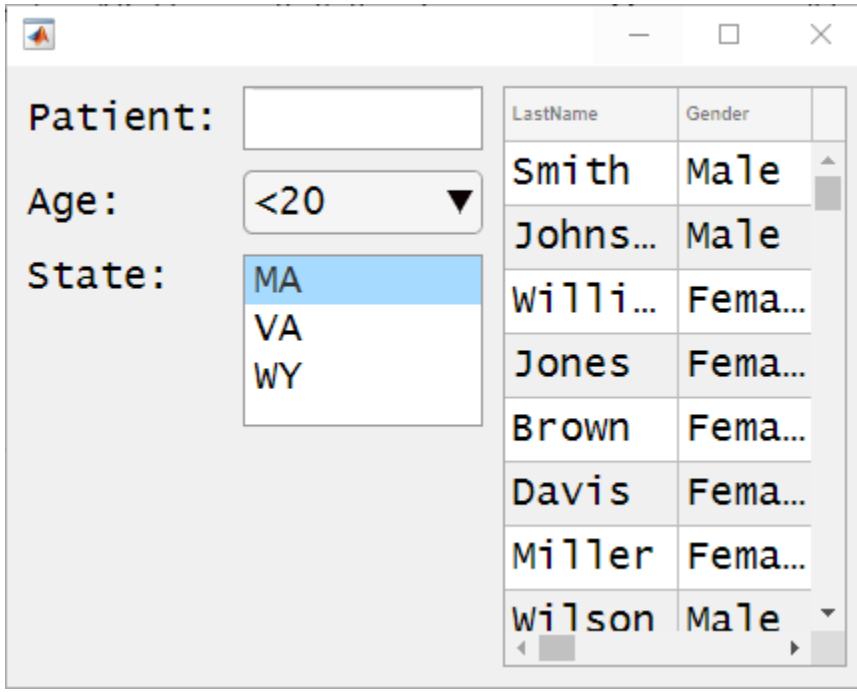
and columns with `'fit'` height and width, the size of those row and columns automatically adjusts to fit the new content.

```
patientLabel = uilabel(g,'Text','Patient:');
patientLabel.Layout.Row = 2;
patientLabel.Layout.Column = 1;
patientEdit = uieditfield(g);

ageLabel = uilabel(g,'Text','Age:');
ageLabel.Layout.Row = 3;
ageLabel.Layout.Column = 1;
ageEdit = uidropdown(g,'Items',{'<20','20 - 40','40 - 60','>60'});

stateLabel = uilabel(g,'Text','State:');
stateLabel.VerticalAlignment = 'top';
stateLabel.Layout.Row = 4;
stateLabel.Layout.Column = 1;
stateLB = uilistbox(g,'Items',{'MA','VA','WY'});

tdata = readtable('patients.dat');
uit = uitable(g);
uit.Data = tdata;
uit.Layout.Row = [2 5];
uit.Layout.Column = 3;
```



Now, change the font name and font size of all the UI components in the grid.

```
set(allchild(g),'FontName','Lucida Console','FontSize',20)
```



Hide Rows Based on Run-Time Conditions

Hide components within a row of a grid based on the user's selection in a drop-down menu.

Create a program file called `showhide.m`. In the file,

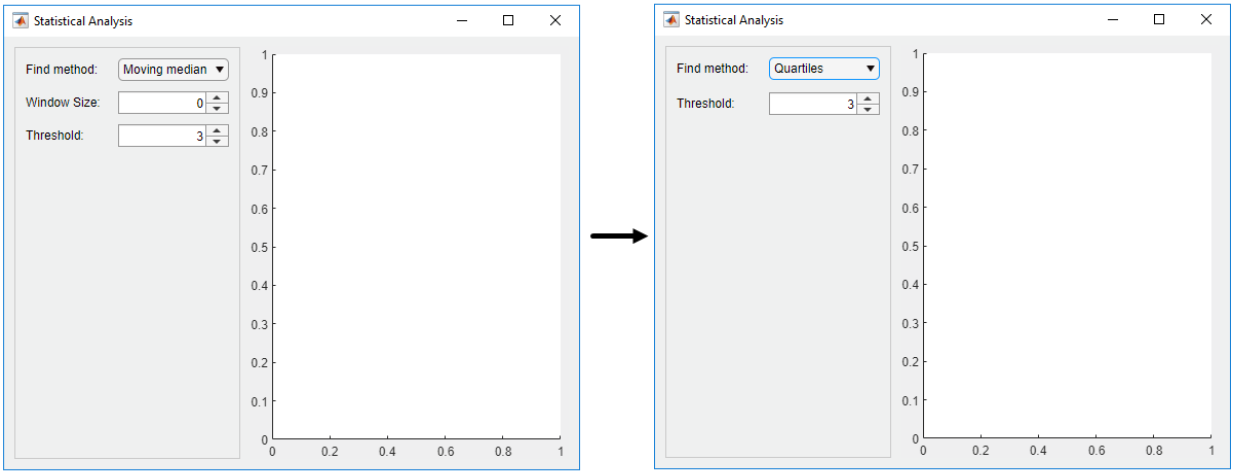
- Create a 1-by-2 grid in the figure and call it `grid1`. Use this grid to manage a panel and an axes component.
- Create a 3-by-2 grid inside the panel and call it `grid2`. Use this grid to manage the layout of a drop-down menu, two spinners, and their labels.
- Create a callback function called `findMethodSelected` for the drop-down menu. When the value of the drop-down menu changes to `'Quartiles'`, the callback hides the components in a second row of `grid2` by setting `grid2.RowHeight{2}` to `0`.

Then, run the file.

```
function showhide

    f = uifigure('Name','Statistical Analysis');

    % Create grid1 in the figure
    grid1 = uigridlayout(f);
    grid1.RowHeight = {'1x'};
    grid1.ColumnWidth= {220,'1x'};

    % Add a panel and axes
    p = uipanel(grid1);
    ax = uiaxes(grid1);

    % Create grid2 in the panel
    grid2 = uigridlayout(p);
    grid2.RowHeight = {22, 22, 22};
    grid2.ColumnWidth = {80,'1x'};

    % Add method label and drop-down
    findMethodLabel = uilabel(grid2,'Text','Find Method:');
    findMethod = uidropdown(grid2);
    findMethod.Items = {'Moving median','Quartiles'};
    findMethod.ValueChangedFcn = @findMethodSelected;

    % Add window size label and spinner
    winSizeLabel = uilabel(grid2,'Text','Window Size:');
    winSize = uispinner(grid2,'Value',0);

    % Add threshold label and spinner
    thresLabel = uilabel(grid2,'Text','Threshold:');
    thres = uispinner(grid2,'Value',3);

        function findMethodSelected(src,~)
            method = src.Value;

            switch method
            case 'Quartiles'
                % Collapse the second row (hides winSize spinner)
                grid2.RowHeight{2} = 0;
            case 'Moving median'
                % Expand the second row
                grid2.RowHeight{2} = 22;
            end
```

```
        end
end
```

When you set the **Find Method** to **Quartiles** in the app, the **Window Size** label and the spinner next to it become hidden.



Input Arguments

parent — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If you do not specify a parent container, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

sz — Size

`[2 2]` (default) | two-element vector

Size of the grid, specified as a two-element vector. The first element is the number of rows, and the second element is the number of columns. Both values must be greater

than `0`. When you specify this argument without setting the `RowHeight` or `ColumnWidth` properties, MATLAB automatically sets the row heights and column widths to `'1x'`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `uigridlayout('RowHeight',{100,100})` creates a grid containing two rows that are 100 pixels in height.

Note The properties listed here are only a subset. For a complete list, see GridLayout Properties.

`ColumnWidth` — Column width

`{'1x','1x'}` (default) | cell array | string array | numeric array

Column width, specified as a cell array containing either `'fit'`, numbers, or numbers paired with `'x'` characters. You can specify any combination of values. The number of elements in the cell array controls the number of columns in the grid. For example, to create a 4-column grid, specify a 1-by-4 cell array. Column width can be specified as a string array or numeric array, only if the elements specified are of the same type, like `["1x" "2x" "1x"]` or `[100 200 50]`.

There are three different types of column widths:

- Fit width — Specify `'fit'`. Column width automatically adjusts to fit its contents. For text-based components, `'fit'` width adjusts with font properties to show the whole text. For non text-based components, `'fit'` width is based on the default size of the component and other factors. Use `'fit'` width if you want to avoid hard-coding the column width to fit components, or if your app is translated to another language or runs on different platforms.
- Fixed width in pixels — Specify a number. The column width is fixed at the number of pixels you specify. When the parent container resizes, the column width does not change.
- Variable width — Specify a number paired with an `'x'` character (for example, `'1x'`). When the parent container resizes, the column width grows or shrinks. Variable-width

columns fill the remaining horizontal space that the fixed-width columns do not use. The number you pair with the `'x'` character is a weight for dividing up the remaining space among all the variable-width columns. If the grid has only one variable-width column, then it uses all the remaining space regardless of the number. If there are multiple variable-width columns that use the same number, then they share the space equally. Otherwise, the amount of space is proportional to the number.

For example, `{'fit',200,'2x','1x'}` specifies that the width of the first column is sized to fit its content, the second column is fixed at 200 pixels, and the last two columns share the remaining horizontal space. The third column uses twice as much space as the fourth column.

Changing certain aspects of a layout can affect the value of this property. For example, adding more components to a fully populated grid changes the size of the grid to accommodate the new components.

Changing the `ColumnWidth` property on a grid layout that already contains components does not change the layout of the components. For example, if you try to dynamically delete a column that contains components, the `ColumnWidth` property does not change until you move those components out of that column.

`RowHeight` — Row height

`{'1x','1x'}` (default) | cell array | string array | numeric array

Row height, specified as a cell array containing either `'fit'`, numbers, or numbers paired with `'x'` characters. You can specify any combination of values. The number of elements in the cell array controls the number of rows in the grid. For example, to create a grid that has 4 rows, specify a 1-by-4 cell array. Row height can be specified as a string array or numeric array, only if the elements specified are of the same type, like `["1x" "2x" "1x"]` or `[100 200 50]`.

There are three different types of row heights:

- Fit height — Specify `'fit'`. Row height automatically adjusts to fit its contents. For text-based components, `'fit'` height adjusts with font properties to show the whole text. For non text-based components, `'fit'` height is based on the default size of the component and other factors. Use `'fit'` height if you want to avoid hard-coding the row height to fit components, or if your app is translated to another language or runs on different platforms.
- Fixed height in pixels — Specify a number. The row height is fixed at the number of pixels you specify. When the parent container resizes, the row height does not change.

- Variable height — Specify a number paired with an `'x'` character (for example, `'1x'`). When the parent container resizes, the row grows or shrinks. Variable-height rows fill the remaining vertical space that the fixed-height rows do not use. The number you pair with the `'x'` character is a weight for dividing up the remaining space among all the variable-height rows. If the grid has only one variable-height row, then it uses all the remaining space regardless of the number. If there are multiple variable-height rows that use the same number, then they share the space equally. Otherwise, the amount of space is proportional to the number.

For example, `{'fit',200,'2x','1x'}` specifies that the height of the first row is sized to fit its content, the second row is fixed at 200 pixels, and the last two rows share the remaining vertical space. The third row uses twice as much space as the fourth row.

Changing certain aspects of a layout can affect the value of this property. For example, adding more components to a fully populated grid changes the size of the grid to accommodate the new components.

Changing the `RowHeight` property on a grid layout that already contains components does not change the layout of the components. For example, if you try to dynamically delete a row that contains components, the `RowHeight` property does not change until you move those components out of that row.

Tips

To view the list of component objects in the grid, query the `Children` property of the grid. Changing the order in the list does not change the layout in the grid.

See Also

Functions
`appdesigner` | `scroll` | `uifigure`

Properties
GridLayout Properties

Topics
"Use Grid Layout Managers"

Introduced in R2018b

uihtml

Create HTML UI component

Syntax

```
h = uihtml
h = uihtml(parent)
h = uihtml( ___ ,Name,Value)
```

Description

`h = uihtml` creates an HTML UI component in a new figure and returns the `HTML` UI component object. MATLAB calls the `uifigure` function to create the figure.

With the `uihtml` function, you can embed HTML, JavaScript, or CSS content in your app and interface with third-party libraries to display content like widgets or data visualizations. All supporting files (including HTML, JavaScript, CSS, images) must be saved in a location that your local file system can access.

`h = uihtml(parent)` creates an HTML UI component in the specified parent container. The parent can be a `Figure` object created with the `uifigure` function, or one of its child containers.

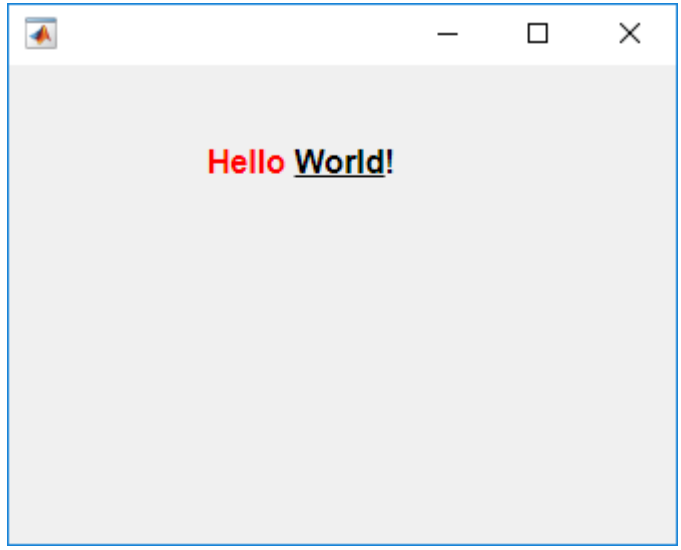
`h = uihtml( ___ ,Name,Value)` specifies HTML UI component properties using one or more name-value pair arguments. For example, `'HTMLSource','timepicker.html'` sets the HTML source to the specified HTML file.

Examples

Display Formatted Text

Create an HTML component and display formatted text in it using HTML markup.

```
fig = uifigure('Position',[561 497 333 239]);
h = uihtml(fig);
h.HTMLSource = '<p style="font-family:arial;"><b><span style="color:red;">Hello</span>
```



Embed Audio and Video Using HTML File

Create an HTML file to embed audio and video elements in your app and reference media source files.

First, create an empty HTML UI component in a figure.

```
fig = uifigure;
fig.Position = [500 500 380 445];

h = uihtml(fig);
h.Position = [10 10 360 420];
```

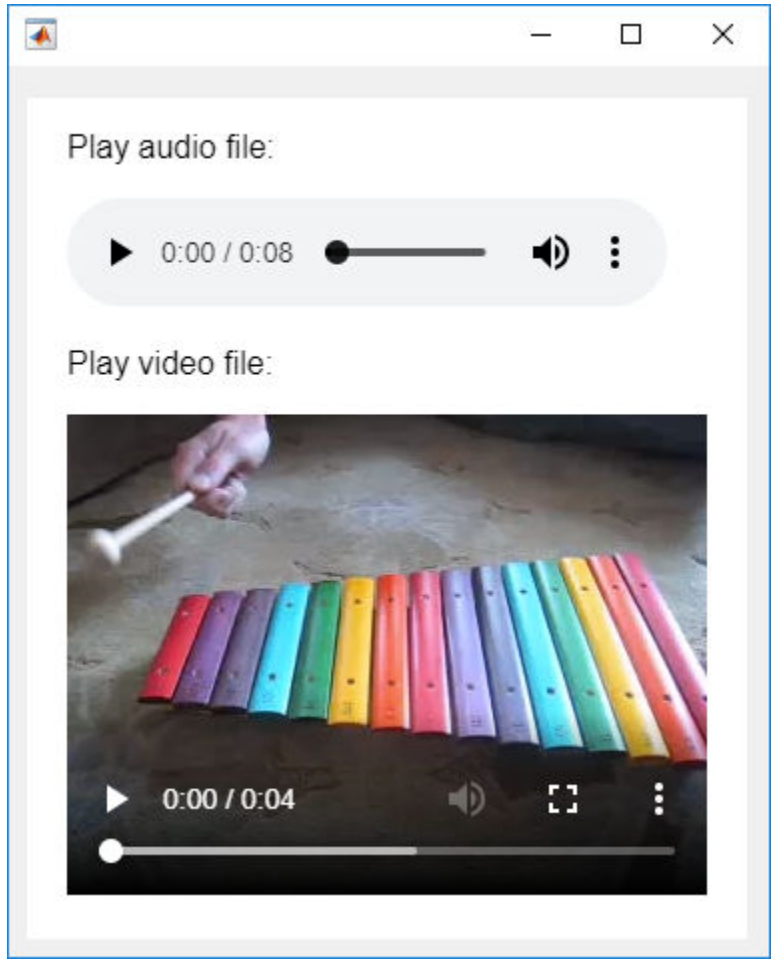
Then, create an HTML file called `audio_videoplayers.html`. In the file, add audio and video elements and specify their file sources relative to the path of the HTML file. In this example, the media files are in the same directory as the HTML file.

```
<!DOCTYPE html>
<html>
    <body style="background-color:white;font-family:arial;">
        <p style="padding-left:20px">
        Play audio file:
        </p>
```

```
        <audio controls style="padding-left:20px">
            <source src="./handel_audio.wav" type="audio/wav">
            <source src="./handel_audio.oga" type="audio/ogg">
            Your browser does not support the audio tag.
        </audio>

        <p style="padding-left:20px">
        Play video file:
        </p>

        <video width="320" height="240" controls style="padding-left:20px">
            <source src="./xylophone_video.mp4" type="video/mp4">
            <source src="./xylophone_video.oga" type="video/ogg">
            Your browser does not support the video tag.
        </video>

    </body>
</html>
```

Now, embed the HTML file in your app by setting the `HTMLSource` property to the full path to the file.

```
h.HTMLSource = fullfile(pwd,'audio_videoplayers.html');
```



Code JavaScript Response to Data Change in MATLAB

Create an `htmlComponent` JavaScript® object that displays the data that is set on an HTML UI component in MATLAB®.

First, create an empty HTML UI component and position it within a figure.

```
fig = uifigure;
fig.Position = [500 500 490 180];

h = uihtml(fig);
h.Position = [20 20 450 130];
```

Then, create an HTML file called `displayDataFromMATLAB.html`. In the file,

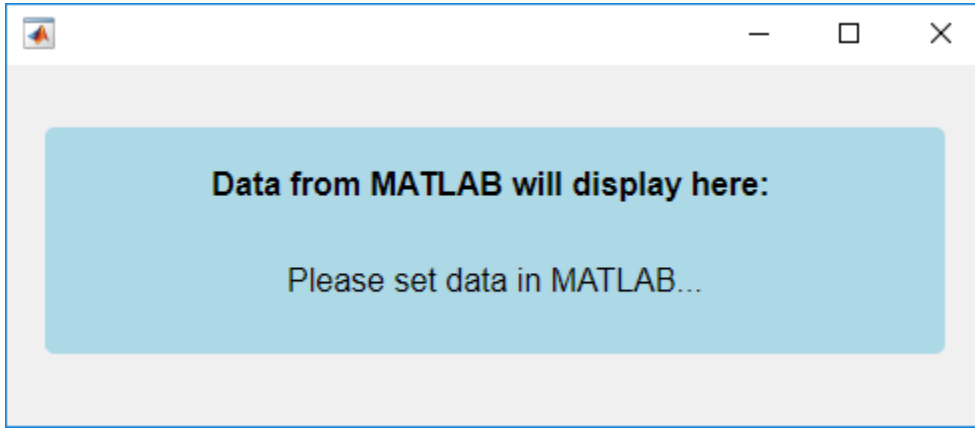
- Use a `style` tag to define internal CSS styles for the `<div>` elements that display text.
- Write a `setup` function inside of a `<script>` tag to connect your JavaScript object, called `htmlComponent`, to the HTML UI component you created in MATLAB.
- Add an event listener within the `setup` function. The event listener listens for `"DataChanged"` events on the `htmlComponent` JavaScript object and creates a listener callback function. When a `"DataChanged"` event occurs, the listener callback function updates the `"dataDisplay" <div> element` with the value that is set to the `Data` property in MATLAB.

```
<!DOCTYPE html>
<html>
<head>
<style>
#prompt {
  font-family: sans-serif;
  text-align:center;
  border-radius: 5px;
  background-color: lightblue;
  padding: 20px;
}

#dataDisplay {
  border-radius: 5px;
  height: 10vh;
  padding: 12px 20px;
}
</style>

<script type="text/javascript">

        function setup(htmlComponent) {

            htmlComponent.addEventListener("DataChanged", function(event) {
            document.getElementById("dataDisplay").innerHTML = htmlComponent.Data;
```

```
            });
        }
</script>
</head>

<body>
     <div id="prompt">

    <span><label for="prompt"><strong>Data from MATLAB will display here:</strong></lab
    <br/><br/>
    <div id ="dataDisplay">
    Please set data in MATLAB...
    </div>

     </div>
</body>

</html>
```

Now, in MATLAB, set the value of the `HTMLSource` property to the path to the HTML file.

```
h.HTMLSource = fullfile(pwd,'displayDataFromMATLAB.html');
```



Set the value of the `Data` property for the HTML UI component. Notice how the `htmlComponent` JavaScript object updates in your HTML UI component.

```
h.Data = "Hello World!";
```

Code MATLAB Response to Data Change in JavaScript

Create an `htmlComponent` JavaScript® object that generates a random number when a button is clicked, and display the number in MATLAB®.

First, create an empty HTML UI component and position it within a figure.

```
fig = uifigure;
fig.Position = [500 500 490 180];

h = uihtml(fig);
h.Position = [20 20 450 130];
```

Then, create an HTML file called `generateJSRandomNumber.html`. In the file,

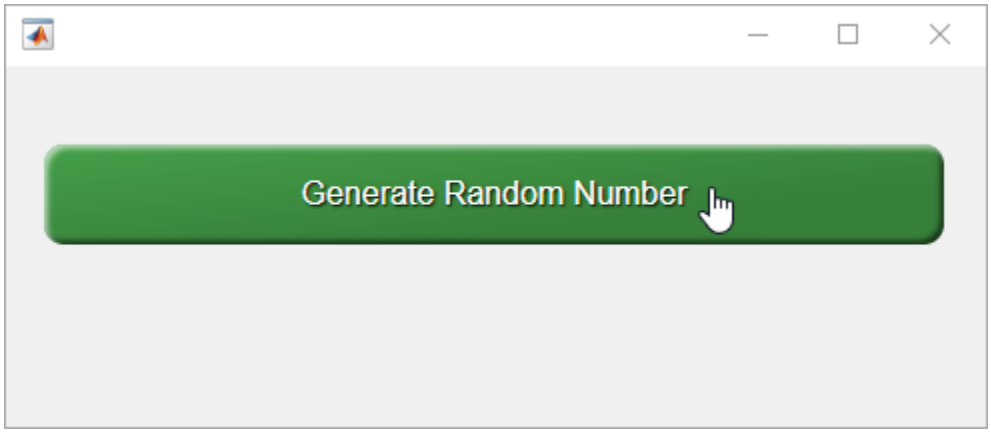
- Use a `style` tag to define internal CSS styles for a button element.
- Write a `setup` function inside of a `<script>` tag to connect your JavaScript object, called `htmlComponent`, to the HTML UI component you created in MATLAB.
- Add an event listener within the `setup` function. The event listener listens for `"click"` events on the button and creates a listener callback function. When a click event occurs, the listener callback function sets the `Data` property of the `htmlComponent` JavaScript object to a randomly generated integer between `1` and `100`.

```
<!DOCTYPE html>
<html>
<head>
<style>
button {
  width: 100%;
  font-family: sans-serif;
  background-color: #4CAF50;
  color: white;
  font-size: 1rem;
  text-shadow: 1px 1px 1px #000;
  padding: 5px 20px;
  margin: 8px 0;
  border: none;
  border-radius: 10px;
  line-height: 2.5;
  cursor: pointer;
  background-image: linear-gradient(to top left,
                                    rgba(0, 0, 0, .2),
                                    rgba(0, 0, 0, .2) 30%,
                                    rgba(0, 0, 0, 0));
  box-shadow: inset 2px 2px 3px rgba(255, 255, 255, .6),
              inset -2px -2px 3px rgba(0, 0, 0, .6);
}

button:hover {
  background-color: #45a049;
}

button:focus {
outline: 0;
}

button:active {
    box-shadow: inset -2px -2px 3px rgba(255, 255, 255, .6),
                inset 2px 2px 3px rgba(0, 0, 0, .6);
}
</style>

   <script type="text/javascript">

      function setup(htmlComponent) {

         document.getElementById("generateRandomNumber").addEventListener("click", func
```

```
            htmlComponent.Data = Math.floor(Math.random() * 100) + 1;
        });
      }
   </script>
</head>

   <body>
     <button id="generateRandomNumber">Generate Random Number</button><br/>
   </body>

</html>
```

Now, in MATLAB, set the value of the `HTMLSource` property to the path to the HTML file. Create a `DataChangedFcn` callback function that displays the random number your JavaScript code generates. Click the **Generate Random Number** button to see the value of the `Data` property display in the MATLAB Command Window.

```
h.HTMLSource = fullfile(pwd,'generateJSRandomNumber.html');
h.DataChangedFcn = @(src,event)disp(h.Data);
```



Input Arguments

parent — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If you do not specify a parent container, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `h = uihtml(uifigure,'HTMLSource','C:\Work\expenses.html');`

Note The properties listed here are only a subset. For a complete list, see HTML Properties.

`HTMLSource` — HTML markup or file

`''` (default) | character vector | string scalar

HTML markup or file, specified as a character vector or string scalar that contains either HTML markup or the path to an HTML file. All HTML markup and files must be well formed. If the specified character vector or string scalar ends with `.html`, then it is assumed to be the path to an HTML file.

When embedding HTML markup, you do not need to specify `<html>` or `<body>` tags. If the markup you specify is supported in the Chromium browser used by MATLAB figures created with the `uifigure` function, then it will render. If you need more structure to your code, then consider using an HTML file instead.

If you set the `HTMLSource` property to the path to an HTML file, the HTML file must be located in a place that your local file system can access. If you are using supporting files, such as JavaScript, CSS, libraries, or images, then put the files in a location that your local file system can access, reference them in an HTML file, and set the `HTMLSource` property to the path to the HTML file. Supporting files cannot be in a relative directory adjacent to or above the HTML file.

Example: `h = uihtml('HTMLSource','CustomCharts.html')` specifies an HTML file.

Example: `h = uihtml('HTMLSource','<p>This is <span style="color:red">red</span> text.</p>')` specifies markup.

`Data` — MATLAB data
any MATLAB data type

MATLAB data, specified as any MATLAB data type. Use this argument when the value of `HTMLSource` is the path to an HTML file defining a JavaScript object. Then, this data can be synchronized between the MATLAB HTML UI component and the JavaScript object.

To synchronize the value of the `Data` property between MATLAB and the third-party content that you are embedding in your app, create a `setup` function in an HTML file that connects a JavaScript object called `htmlComponent` to the HTML UI component in MATLAB. Then, set the `HTMLSource` property value to the path to the file.

Changes in the value of the `Data` property trigger callback events in only one direction. Specifically,

- When the `Data` property is set in MATLAB, the `Data` property of the `htmlComponent` JavaScript object also updates and triggers JavaScript event listeners of the `'DataChanged'` event.
- When `htmlComponent.Data` is set in JavaScript, the `Data` property of the HTML UI component in MATLAB also updates and triggers the `DataChangedFcn` callback.

If you have data like nested cell arrays, arrays of structures, or MATLAB table array data, you might need more information about how data is converted between MATLAB and JavaScript. When the `Data` property of an HTML UI component is set in MATLAB, the data is converted using the `jsonencode` function, synchronized with JavaScript, parsed using JSON.parse(), and finally set to the `Data` property of the `htmlComponent` JavaScript object. Conversions happen in the opposite direction as well by way of JSON.stringify() and the `jsondecode` function. The `Data` property is the only property that synchronizes with JavaScript.

Refer to the `jsonencode` function for more information about the data types it supports. If the `jsonencode` function does not support your data type, then use a data type conversion function, like `num2str`, to convert your data to a supported type. Then, set it as the value of the `Data` property.

For more information about how to write an HTML file that enables data synchronization and the properties of the `htmlComponent` JavaScript object, see “Create HTML File That Can Trigger or Respond to Data Changes”.

`DataChangedFcn` — Callback when data changes
`''` (default) | function handle | cell array | character vector

Callback when data changes, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

When the value of the `Data` property changes in the `htmlComponent` JavaScript object, the `Data` property of the MATLAB HTML UI component is updated, and this callback executes.

This callback function can access specific information about changes in data in the JavaScript object. MATLAB passes the information in a `DataChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the `DataChangedData` object properties using dot notation. For example, `event.PreviousData` returns the previous value of `Data` before it was last updated or changed. The `DataChangedData` object is not available to callback functions specified as character vectors.

This table lists the properties of the `DataChangedData` object.

Property	Description
`Data`	Most recent data resulting from the last `DataChanged` event
`PreviousData`	Data before the last `DataChanged` event
`Source`	MATLAB HTML UI component that executes the callback
`EventName`	`'DataChanged'`

For more information about writing callbacks, see “Write Callbacks in App Designer”.

`Position` — Location and size of HTML UI component
`[100 100 100 100]` (default) | `[left bottom width height]`

Location and size of HTML UI component relative to the parent container, specified as a four-element vector of the form `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the HTML UI component
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the HTML UI component
`width`	Distance between the right and left outer edges of the HTML UI component
`height`	Distance between the top and bottom outer edges of the HTML UI component

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

All measurements are in pixel units.

Limitations

- You cannot link to URL web resources or embed applications from external web pages using the `uihtml` function.
- HTML files that you specify for the `HTMLSource` property cannot access third-party JavaScript libraries by way of a Content Delivery Network (CDN). Save libraries in a location that your local file system can access.
- Common web file types, like JavaScript and CSS, can be referenced from the HTML file you specify for the `HTMLSource` property, but other web file types might not be supported.
- Content cannot overflow the bounds of the HTML UI component. If you are creating dynamic content, like pop-up widgets, make the size of your HTML UI component large enough to display the widget when the pop-up is in its opened state.
- On MATLAB desktop, certain HTML features such as web plugins and access to webcams or microphones are not supported.

- You cannot set the `HTMLSource` property to an HTML file that has query parameters appended to its file name. As a workaround, set these parameters in `htmlComponent.Data`.
- Hyperlinks with `matlab:` operations are not supported.

More About

Potential Security Risks

Note

This list is not intended to be comprehensive. Consult external resources for additional risks and security considerations when interfacing with third-party libraries.

Applications that interface with third-party JavaScript libraries are potentially vulnerable to *data* or *code injection* attacks.

- MATLAB features like the `eval` function can increase the risk of injection attacks. As a countermeasure, validate untrusted input before using it in MATLAB.
- Any MATLAB function that processes code-like input (XML, JSON, SQL, etc.) is potentially vulnerable to code injection.
- Applications that access the operating system using the MATLAB `system`, `dos`, or `unix` functions might be vulnerable to code injection.

For information about how to reduce the potential risk to your deployed web apps, see "Authoring Secure Web Apps" (MATLAB Compiler).

Tips

- If you are creating an HTML file to embed in your app, first verify that the static HTML renders properly in a browser. For more information, see "Create HTML File That Can Trigger or Respond to Data Changes".
- If you want to refresh the value of the `HTMLSource` property, use this workaround to save the original value of `HTMLSource` to a variable, reset `HTMLSource` it to an empty

value and update the figure to clear its contents, and finally set `HTMLSource` back to its original value.

```
oldValue = h.HTMLSource;
h.HTMLSource = '';
drawnow()
h.HTMLSource = oldValue;
```

- MATLAB figures created with the `uifigure` function use the Chromium browser and run HTML5. New Chromium versions are released frequently and can vary between operating systems. Consult external Chromium resources for supported features and functionality. Take this factor into consideration if you plan to share your app or use it in different environments.
- Check external resources, like https://caniuse.com/, for the types of features that are supported across different desktop and web browsers.
- MathWorks does not promote any third-party libraries.

See Also

Functions
`jsondecode` | `jsonencode` | `uifigure`

Properties
HTML Properties

Topics

"Create HTML File That Can Trigger or Respond to Data Changes"

Introduced in R2019b

uiimage

Create image component

Syntax

```
im = uiimage
im = uiimage(Name,Value)
im = uiimage(parent)
im = uiimage(parent,Name,Value)
```

Description

`im = uiimage` creates an image component in a new figure and returns the `Image` object. MATLAB calls the `uifigure` function to create the new figure. Use `uiimage` to display a picture, icon, or logo in your app.

`im = uiimage(Name,Value)` specifies `Image` property values using one or more name-value pair arguments.

`im = uiimage(parent)` creates an image component in the specified parent container. The parent can be a figure created using the `uifigure` function, or one of its child containers.

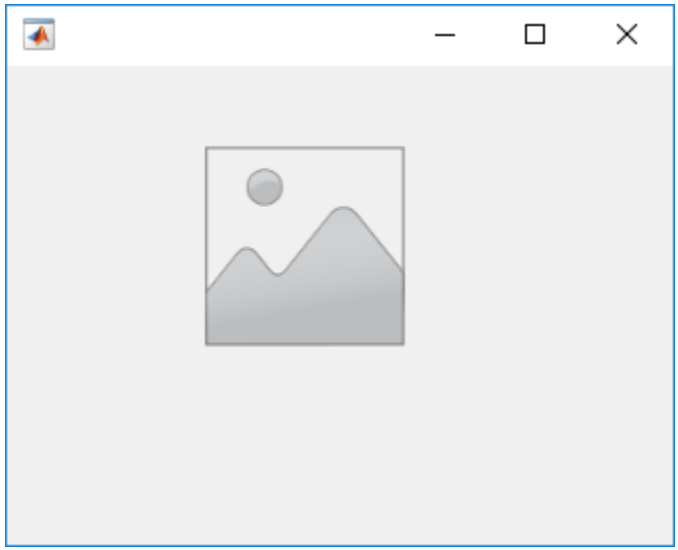
`im = uiimage(parent,Name,Value)` specifies the parent container and one or more property values.

Examples

Display a Picture

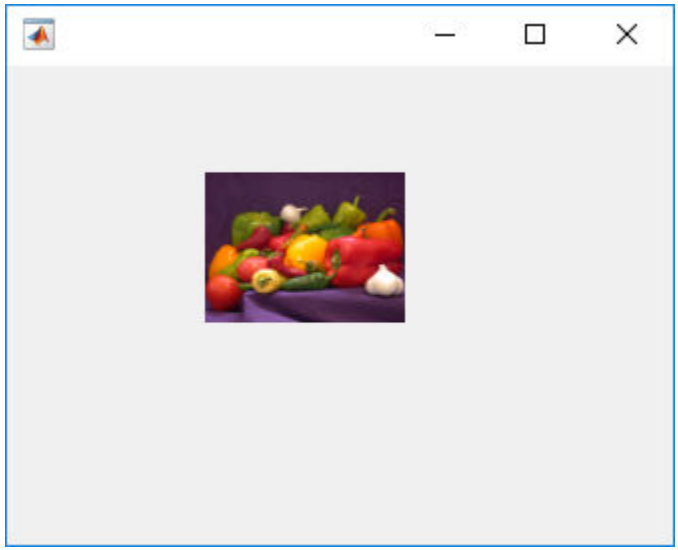
Create an image component within a figure. The default image displays.

```
f = uifigure;
im = uiimage(f);
```



Now, add a picture to the image component.

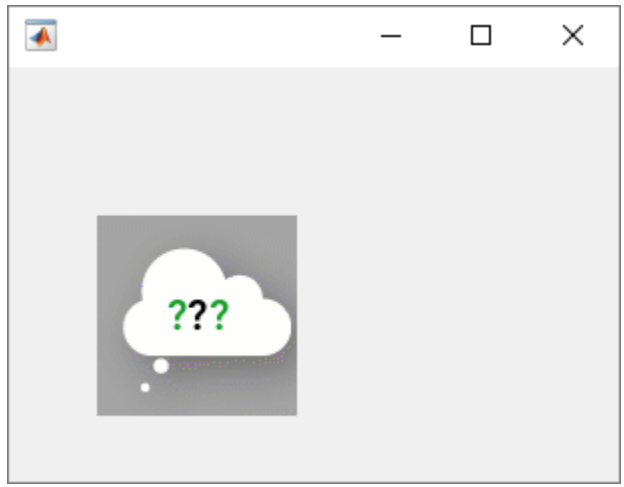
```
im.ImageSource = 'peppers.png';
```



Display Animated GIF With Letterboxing

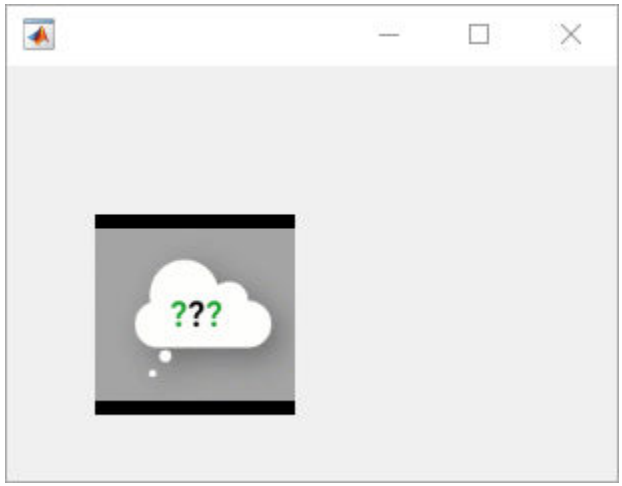
Create an image component that displays an animated GIF using the actual size of the image.

```
f = uifigure;
im = uiimage(f,'ImageSource','questions.gif');
im.ScaleMethod = 'none';
```



Now, scale the image so that it fits within the default component area, preserving aspect ratio and without clipping. Then, apply a black background to create the appearance of letterboxing (black bars above and below the image).

```
im.ScaleMethod = 'scaledown';
im.BackgroundColor = 'black';
```



Configure Image to Perform Action When Clicked

Create an image and a callback that executes when the image is clicked. In this case, the image opens the MathWorks® website.

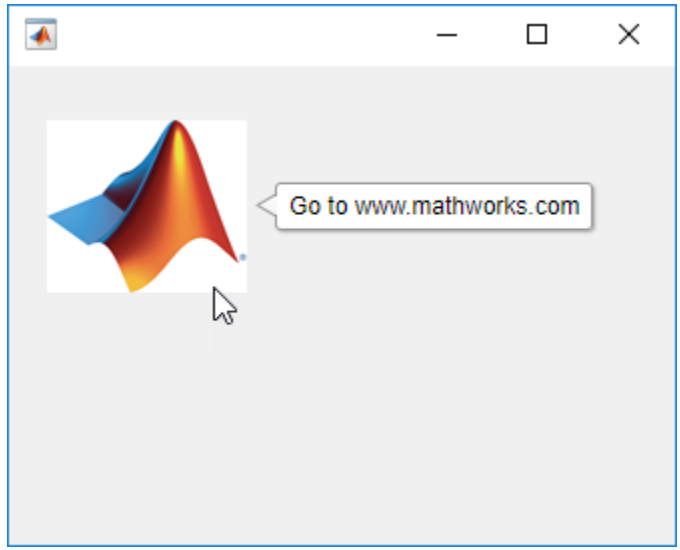
This program file, called `imagetoURL.m`, shows you how to:

- Create an image component with an `ImageClickedFcn` callback.
- Use the `web` function within the callback to open a URL in the MATLAB® browser.
- Create a tooltip that appears when you hover over the image.

When you run the program file, click the image to open the MathWorks® website.

```
function imagetoURL
f = uifigure('Visible','off');
f.Position(3:4) = [333 239];

im = uiimage(f);
im.Position = [20 120 100 100];
im.ImageSource = 'membrane.png';
im.ImageClickedFcn = @ImageClicked;
im.Tooltip = 'Go to www.mathworks.com';
```

```
    function ImageClicked(src,event)
        url = 'https://www.mathworks.com/';
        web(url);
    end

f.Visible = 'on';
end
```



Input Arguments

parent — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If you do not specify a parent container, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `im = uiimage('ScaleMethod','none')`

Note The properties listed here are only a subset. For a complete list, see Image Properties.

`ImageSource` — Image source or file

`''` (default) | file path | `m`-by-`n`-by-3 truecolor image array

Image source or file, specified as a file path or an `m`-by-`n`-by-3 truecolor image array. Supported image formats include JPEG, PNG, GIF, SVG, or `m`-by-`n`-by-3 truecolor image array.

For more information on truecolor image arrays, see “Image Types”.

Example: `im = uiimage('ImageSource','peppers.png');`

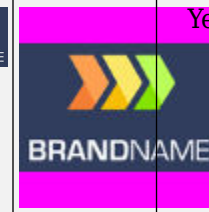
Example: `im.ImageSource = 'C:\TEMP\ngc6543a.jpg';`

`ScaleMethod` — Image scaling method

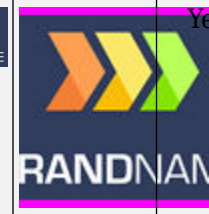
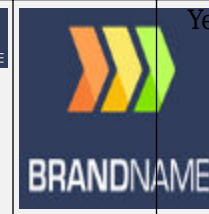
`'fit'` (default) | `'fill'` | `'none'` | `'scaledown'` | `'scaleup'` | `'stretch'`

Image scaling method, specified as one of the values listed in the table. Use this name-value pair argument to specify how you want your image to render within the component area.

The table also demonstrates each scale method with an example image. In the rendered image examples, the `BackgroundColor` property of the image component has been set to `'magenta'`. The scaling behavior of SVG image files may vary based on how the file is defined.

Value	Description	Example		Scales Up	Scales Down	Maintains Aspect Ratio	Clips Image
		Original Image	Rendered Image				
`'fit'`	Scales in any direction to display the image within the component area, and maintains aspect ratio without clipping.			Yes	Yes	Yes	No
`'fill'`	Scales in any direction to fill the component area, maintaining aspect ratio and clipping if necessary.			Yes	Yes	Yes	Yes
`'none'`	Uses the actual size of the image and maintains aspect ratio. If the component area is smaller than the image, the image is clipped.			No	No	Yes	Yes

Value	Description	Example		Scales Up	Scales Down	Maintains Aspect Ratio	Clips Image
		Original Image	Rendered Image				
`'scaledown'`	Scales down and maintains aspect ratio without clipping. If the original image is larger than the component area, the image scales down and renders as if the `ScaleMethod` was set to `'fit'`. If the original image is smaller than the component area, the image does not scale down and renders as if the `ScaleMethod` was set to `'none'`.			No	Yes	Yes	No

Value	Description	Example		Scales Up	Scales Down	Maintains Aspect Ratio	Clips Image
		Original Image	Rendered Image				
`'scaleup'`	Scales up and maintains aspect ratio with clipping. If the original image is smaller than the component area, the image scales up and renders as if the `ScaleMethod` was set to `'fit'`. If the original image is larger than the component area, the image does not scale up and renders as if the `ScaleMethod` was set to `'none'`.			Yes	No	Yes	Yes
`'stretch'`	Scales in any direction to fill the component area, without maintaining the aspect ratio and without clipping.			Yes	Yes	No	No

`ImageClickedFcn` — Image clicked callback

' ' (default) | function handle | cell array | character vector

Image clicked callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when the user clicks the image in the app.

This callback function can access specific information about the user's interaction with the image. MATLAB passes this information in an `ImageClickedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.Source` returns the `Image` object that the user is interacting with to trigger the callback. The `ImageClickedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ImageClickedData` object.

Property	Value
`EventName`	`'ImageClicked'`
`Source`	Component executing the callback

For more information about writing callbacks, see “Write Callbacks in App Designer”.

`Position` — Location and size of image component
`[100 100 100 100]` (default) | `[left bottom width height]`

Location and size of image component relative to the parent, specified as a four element vector of the form `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the image component
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the image component

Element	Description
`width`	Distance between the right and left outer edges of the image component
`height`	Distance between the top and bottom outer edges of the image component

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

All measurements are in pixel units.

Tips

- For image analysis and processing, see graphics functions `image` and `imshow`.

See Also

Functions
`appdesigner` | `image` | `imshow` | `uifigure`

Properties
Image Properties

Introduced in R2019a

uimenu

Create menu or menu items

Syntax

```
m = uimenu
m = uimenu(Name,Value)
m = uimenu(parent)
m = uimenu(parent,Name,Value)
```

Description

`m = uimenu` creates a menu in the current figure and returns the `Menu` object. If there is no figure available, MATLAB calls the `figure` function to create one.

`m = uimenu(Name,Value)` specifies menu property values using one or more name-value pair arguments.

`m = uimenu(parent)` creates the menu in the specified parent container. The parent container can be a figure created with either the `figure` or `uifigure` function, or another `Menu` object. Property values for `uimenu` vary slightly depending on whether the app is created with the `figure` or `uifigure` function. For more information, see “Name-Value Pair Arguments” on page 1-16184.

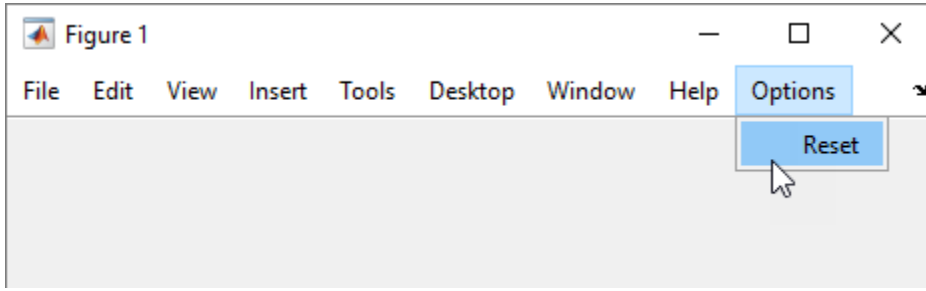
`m = uimenu(parent,Name,Value)` specifies the parent container and one or more property values.

Examples

Menu in Default Menu Bar

Create a figure that displays the default menu bar. Add a menu and a menu item.

```
f = figure('Toolbar','none');
m = uimenu('Text','Options');
mitem = uimenu(m,'Text','Reset');
```



Menu Item with Keyboard Shortcuts and Callback

Add a menu item with keyboard shortcuts to the menu bar and define a callback that executes when the menu item is selected.

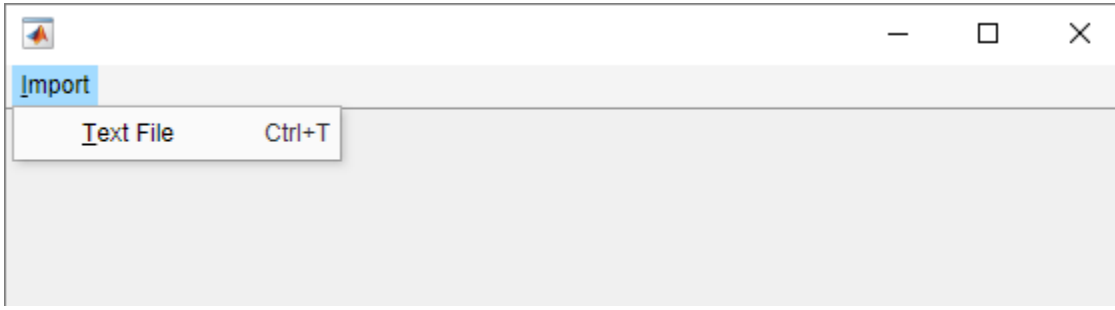
First, create a program file called `importmenu.m`. Within the program file:

- Create a figure.
- Add a menu called **Import**. Create a mnemonic keyboard shortcut for the menu by specifying `'&Import'` as the text label.
- Create a menu item and specify mnemonic and accelerator keyboard shortcuts.
- Define a `MenuSelectedFcn` callback that executes when the user clicks the menu item or uses the mnemonic or accelerator keyboard shortcuts.

Run the program file.

```
function importmenu
f = uifigure;
m = uimenu(f,'Text','&Import');

mitem = uimenu(m,'Text','&Text File');
mitem.Accelerator = 'T';
mitem.MenuSelectedFcn = @MenuSelected;

    function MenuSelected(src,event)
        file = uigetfile('*.txt');
```

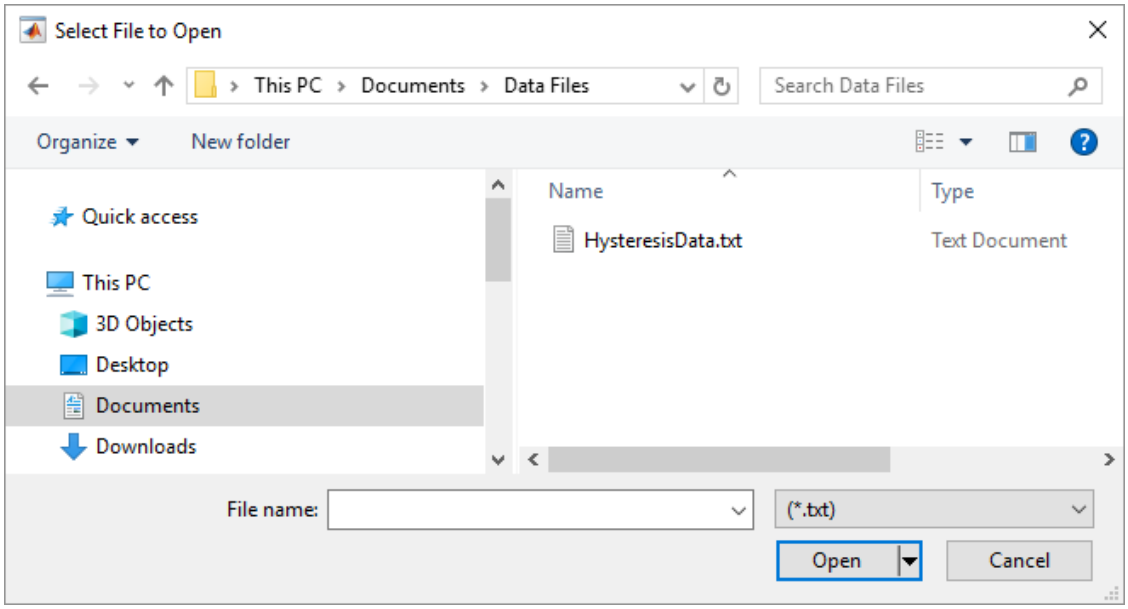
```
    end
end
```



You can interact with the menu and menu item, using the keyboard, in the following ways:

- Select the **Import** menu by pressing **Alt+I**.
- Select the **Text File** menu item and execute the callback by pressing **Alt+I+T**.
- Select the **Text File** menu item and execute the callback by using the accelerator **Ctrl +T**.

When you select the **Text File** menu item, the Select File to Open dialog box opens with the extension field filtered to text files.



Menu with Checked Menu Item and Shared Callback

Create a checked menu item that can be selected or cleared to show a grid in axes. Share the callback with a push button so that pushing it also shows or hides the grid.

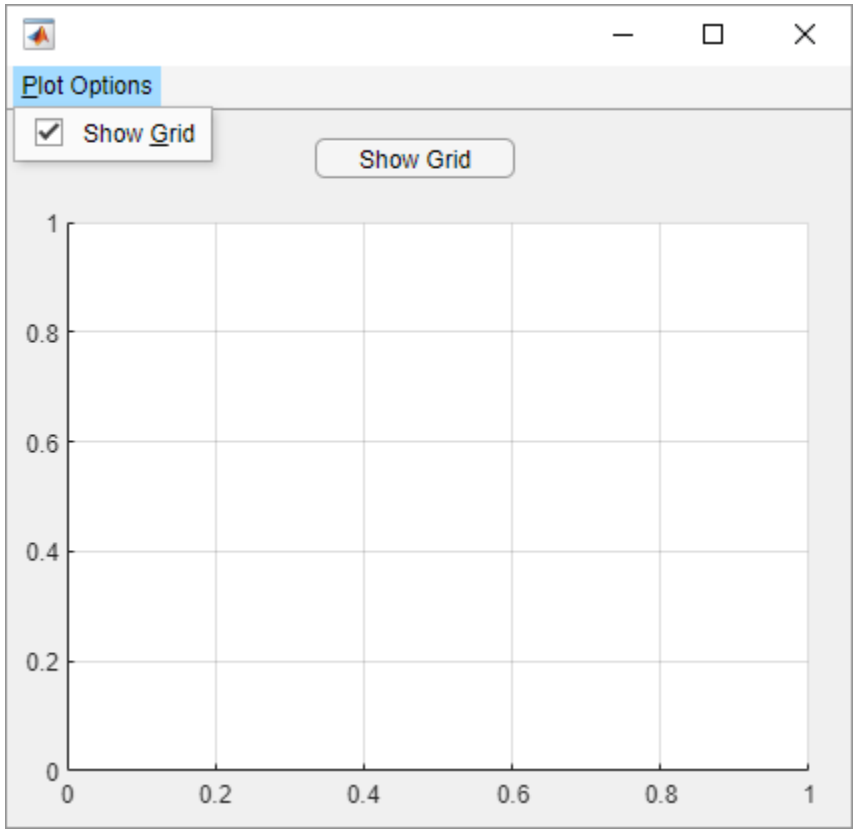
First, create a program file called `plotOptions.m`. Within the program file:

- Create a figure with a push button, and axes that display a grid.
- Add a menu and a menu item with mnemonics. Specify that the menu item is checked.
- Define a `MenuSelectedFcn` callback that hides or shows the grid when the user interacts with the menu item.
- Define a `ButtonPushedFcn` that uses the same callback function as the menu item.

Run the program file.

```
function plotOptions
f = uifigure;
ax = uiaxes(f);
grid(ax);
```

```
btn = uibutton(f,'Text','Show Grid');
btn.Position = [155 325 100 20];

m = uimenu(f,'Text','&Plot Options');
mitem = uimenu(m,'Text','Show &Grid','Checked','on');
mitem.MenuSelectedFcn = @ShowGrid;
btn.ButtonPushedFcn = @ShowGrid;

    function ShowGrid(src,event)
        grid(ax);
        if strcmp(mitem.Checked,'on')
            mitem.Checked = 'off';
        else
            mitem.Checked = 'on';
        end
    end
end
```



Input Arguments

parent — Parent container
`Figure` object | `Menu` object

Parent container, specified as a `Figure` object created with either the `figure` or `uifigure` function, or another `Menu` object. If you do not specify a parent container, then MATLAB calls `figure` to create one, and places the menu in the menu bar of that figure. Specify the parent as an existing `Menu` object to add menu items to a menu, or to nest menu items.

To add menu items to a context menu in GUIDE, or context menus in a figure created with the `figure` function, specify the parent as a `ContextMenu` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `m = uimenu('Text','Open')` creates a menu and sets its label to `'Open'`.

- For a list of properties available for App Designer, or for creating apps with the `uifigure` function, see Menu for App Designer.
- For a list of properties available for GUIDE, or for creating apps with the `figure` function, see Menu for GUIDE.

See Also

Properties
Menu Properties

Topics

"Use One Callback for Multiple App Designer Components"

Introduced before R2006a

Menu Properties

Control appearance and behavior of menu

Note The properties listed here are valid for menus in GUIDE or in apps created with the `figure` function. If you are using App Designer or the `uifigure` function, see Menu Properties instead. For more information, see “GUIDE Migration Strategies”.

Description

Menus display drop-down lists of options at the top of an app window. Call the `uimenu` function to create a menu or add a submenu to an existing menu. Properties control the appearance and behavior of a menu. Use dot notation to refer to a specific object and property.

```
f = figure;
m = uimenu(f);
m.Text = 'Open Selection';
```

Properties

Menu

`Text` — Menu label

character vector | string scalar

Menu label, specified as a character vector or string scalar. This property specifies the label that appears on the menu or menu item.

Avoid using these case-sensitive reserved words: "default", "remove", and "factory". If you must use a reserved word, then specify a backslash character before the word. For instance, specify "default" as `'\default'`.

You can specify a mnemonic keyboard shortcut (**Alt+***`mnemonic`*) by using the ampersand (&) character in the text for the label. The character that follows the ampersand appears underlined in the menu when **Alt** is pressed. You can select the menu item by holding down the **Alt** key and typing the character shown.

To use mnemonics, you must specify a mnemonic for all menus and menu items that you define in the app. If you define mnemonics only for some menus or menu items, pressing the **Alt** key does not have any effect. Mnemonics cannot be used on menus or menu items within a context menu.

The table shows some examples:

Text Value	Menu Label with Mnemonic Hints
`'&Open Selection'`	Open Selection
`'O&pen Selection'`	Open Selection
`'&Save && Go'`	Save & Go

`Accelerator` — Keyboard shortcut
character

Keyboard shortcut, specified as a character. Use this property to define a keyboard shortcut for selecting a menu item.

Example: `mitem.Accelerator = 'H'`

Specifying an accelerator value enables users to select the menu item by pressing a character and another key, instead of using the mouse. The key sequence is platform specific.

- Windows systems: **Ctrl+***accelerator*
- Macintosh systems: **Command+***accelerator*
- Linux systems: **Ctrl+***accelerator*

Things to keep in mind when using accelerators:

- The app window must be in focus when entering the accelerator key sequence.
- Accelerators cannot be used on top-level menus.
- Accelerators only work when the menu item meets all these criteria.
  - It does not contain any submenu items.
  - It executes a callback function.

- It has the `Visible` property set to `'on'`.

If the menu item is nested within a context menu, then the accelerator will only work if the context menu is in focus.

`Separator` — Separator line above menu item

`'off'` (default) | `'on'`

Separator line above menu item, specified as `'off'` or `'on'`. Setting this property to `'on'` draws a dividing line above the menu item.

Note The `Separator` property is ignored when the menu item is a top-level menu item.

`Checked` — Menu check indicator

`'off'` (default) | `'on'`

Menu check indicator, specified as `'off'` or `'on'`. Setting this property to `'on'` places a check mark next to the corresponding menu item. Setting it to `'off'` removes the check mark. You can use this feature to show the state of menu items that enable or disable functionality in your application.

Note The `Checked` property is ignored when the menu item is:

- A top-level menu item
- A menu item that contains one or more child menu items

`ForegroundColor` — Menu label color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Menu label color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`Label` — Menu label (not recommended)
character vector | string scalar

Menu label, specified as a character vector or string scalar. This property specifies the label that appears on the menu (or menu item).

Note Use of the `Label` property is not recommended. It might be removed in a future release. Use the `Text` property instead.

`Position` — Relative menu position (not recommended)

scalar integer value

Relative menu position, specified as a scalar integer value. The value of `Position` property indicates placement on the menu bar or within a menu. Top-level menus appear from left to right on the menu bar according to the value of their `Position` property, with `1` representing the left-most position. The individual items within a given menu appear from top to bottom according to the value of their `Position` property, with `1` representing the top-most position.

Note Use of the `Position` property is not recommended. It might be removed in a future release. To change the order of menu items, use the `Children` property or the `uistack` function instead.

Interactivity

`Visible` — Component visibility

`'on'` (default) | `'off'`

Component visibility, specified as `'on'` or `'off'`. When the `Visible` property is set to `'off'`, the component is not visible in the UI, but you can query and set its properties.

`Enable` — Operational state of menu

`'on'` (default) | `'off'`

Operational state of menu, specified as `'on'` or `'off'`. This property controls whether the user can select a menu item. When the value is `'off'`, the menu label appears dimmed, indicating that the user cannot select it.

Callbacks

`MenuSelectedFcn` — Menu selected callback

`''` (default) | function handle | cell array | character vector

Menu selected callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

This callback responds depending on the location of the menu item and the type of interaction:

- Left-clicking a menu expands that menu and triggers its callback.
- While any menu is expanded, hovering any other parent menu (or top-level menu) expands that menu and triggers its callback.

Note Do not use a callback to dynamically change menu items. Deleting, adding, and replacing menu items in a callback can result in a blank menu. Instead, use the `Visible` property to hide or show menu items. You can also enable and disable menu items by setting the `Enable` property. To fully repopulate menu items, delete and create them outside the callback.

`CreateFcn` — Component creation function

`''` (default) | function handle | cell array | character vector

Component creation function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

This property specifies a callback function to execute when MATLAB creates the component. MATLAB initializes all component property values before executing the

`CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Use the `gcbo` function in your `CreateFcn` code to get the component object that is being created.

Setting the `CreateFcn` property on an existing component object has no effect.

`DeleteFcn` — Component deletion function
`''` (default) | function handle | cell array | character vector

Component deletion function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

The `DeleteFcn` property specifies a callback function to execute when MATLAB deletes the component (for example, when the user closes the window). MATLAB executes the `DeleteFcn` callback before destroying the properties of the component object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

Use the `gcbo` function in your `DeleteFcn` code to get the component object that is being deleted.

`Callback` — Menu selected callback (not recommended)
`''` (default) | function handle | cell array | character vector

Menu selected callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

Note Use of the `Callback` property is not recommended. It might be removed in a future release. Use `MenuSelectedFcn` instead.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed:

- A value of `'on'` allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- A value of `'off'` blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn`, or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.

- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.
- MATLAB does not save the state of properties or the display when an interruption occurs. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

See “Interrupt Callback Execution” for an example that shows how the `Interruptible` and `BusyAction` properties affect the behavior of a program.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` (default) or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

The `BusyAction` property of the source of the interrupting callback determines how MATLAB handles its execution. The `BusyAction` property has these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Do not execute the interrupting callback.

Whenever MATLAB invokes a callback, that callback always attempts to interrupt an executing callback. The `Interruptible` property of the object whose callback is running determines if interruption is allowed. If `Interruptible` is set to:

- `on` — Interruption occurs at the next point where MATLAB processes the queue. This is the default.
- `off` — The `BusyAction` property (of the object owning the interrupting callback) determines if MATLAB enqueues or ignores the interrupting callback.

See “Interrupt Callback Execution” for an example that shows how the `BusyAction` and `Interruptible` properties affect the behavior of a program.

BeingDeleted — Deletion status

'off' | 'on'

This property is read-only.

Deletion status, returned as 'off' or 'on'. MATLAB sets the BeingDeleted property to 'on' when the DeleteFcn callback begins execution. The BeingDeleted property remains set to 'on' until the component object no longer exists.

Check the value of the BeingDeleted property to verify that the object is not about to be deleted before querying or modifying it.

HitTest — Ability to become current object (not recommended)

'on' (default) | 'off'

This property has no effect on Menu objects.

Parent/Child

Parent — Parent object

Figure object | ContextMenu object | Menu object

This property is read-only.

Parent object, specified as a Figure, ContextMenu, or a Menu object. You can move a menu item to a different window, or move it under a different menu by setting this property.

Children — Menu children

empty GraphicsPlaceholder array (default) | 1-D array of Menu objects

Menu children, returned as an empty GraphicsPlaceholder or a 1-D array of Menu objects. The children of menus are other menus that function as submenus.

You cannot add or remove children using the Children property. Use this property to view the list of children or to reorder the child menu items. The order of the children in this array reflects the order of the displayed menu items.

To add a child to this list, set the Parent property of the child component to be the Menu object.

Objects with the HandleVisibility property set to 'off' do not list in the Children property.

HandleVisibility — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object handle in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. The `HandleVisibility` property also controls the visibility of the object’s handle in the parent figure's `CurrentObject` property. Handles are still valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object handle is always visible.
`'callback'`	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.

Set the graphics root `ShowHiddenHandles` property to `'on'` to make all handles visible, regardless of their `HandleVisibility` value. This setting has no effect on their `HandleVisibility` values.

Identifiers

Type — Type of graphics object

`'uimenu'`

This property is read-only.

Type of graphics object, returned as `'uimenu'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any array. Specifying `UserData` can be useful for sharing data within apps. See “Share Data Among Callbacks” for more information.

See Also

`guide` | `uimenu`

Topics

“Default Property Values”

Introduced before R2006a

uint8

8-bit unsigned integer arrays

Description

Variables in MATLAB of data type (class) `uint8` are stored as 1-byte (8-bit) unsigned integers. For example:

```
y = uint8(10);
whos y
```

```
  Name      Size            Bytes  Class    Attributes

  y         1x1                 1  uint8
```

For more information on integer types, see “Integers”.

Creation

Some array creation functions allow you to specify the data type. For instance, `zeros(100,'uint8')` creates a 100-by-100 matrix of zeros of type `uint8`.

If you have an array of a different type, such as `double` or `single`, then you can convert that array to an array of type `uint8` by using the `uint8` function.

Syntax

```
Y = uint8(X)
```

Description

`Y = uint8(X)` converts the values in `X` to type `uint8`. Values outside the range $[0,2^8-1]$ map to the nearest endpoint.

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint16` | `uint32` | `uint64` | `logical` | `char`

Examples

Convert to 8-Bit Unsigned Integer Variable

Convert a double-precision variable to an 8-bit unsigned integer.

```
x = 100;
xtype = class(x)
```

```
xtype =
'double'
```

```
y = uint8(x)
```

```
y = uint8
    100
```

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cast` | `int8` | `typecast` | `uint16` | `uint32` | `uint64`

Topics

“Integers”
“Identifying Numeric Classes”
“Hexadecimal and Binary Values”

Introduced before R2006a

uint16

16-bit unsigned integer arrays

Description

Variables in MATLAB of data type (class) `uint16` are stored as 2-byte (16-bit) unsigned integers. For example:

```
y = uint16(10);
whos y
```

```
  Name      Size            Bytes  Class     Attributes

  y         1x1                 2  uint16
```

For more information on integer types, see “Integers”.

Creation

Some array creation functions allow you to specify the data type. For instance, `zeros(100,'uint16')` creates a 100-by-100 matrix of zeros of type `uint16`.

If you have an array of a different type, such as `double` or `single`, then you can convert that array to an array of type `uint16` by using the `uint16` function.

Syntax

```
Y = uint16(X)
```

Description

`Y = uint16(X)` converts the values in `X` to type `uint16`. Values outside the range $[0,2^{16}-1]$ map to the nearest endpoint.

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint32` | `uint64` | `logical` | `char`

Examples

Convert to 16-Bit Unsigned Integer Variable

Convert a double-precision variable to a 16-bit unsigned integer.

```
x = 100;
xtype = class(x)
```

```
xtype =
'double'
```

```
y = uint16(x)
```

```
y = uint16
    100
```

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cast` | `int16` | `typecast` | `uint32` | `uint64` | `uint8`

Topics

“Integers”
“Identifying Numeric Classes”
“Hexadecimal and Binary Values”

Introduced before R2006a

uint32

32-bit unsigned integer arrays

Description

Variables in MATLAB of data type (class) `uint32` are stored as 4-byte (32-bit) unsigned integers. For example:

```
y = uint32(10);
whos y
```

```
  Name      Size            Bytes  Class     Attributes

  y         1x1                 4  uint32
```

For more information on integer types, see “Integers”.

Creation

Some array creation functions allow you to specify the data type. For instance, `zeros(100,'uint32')` creates a 100-by-100 matrix of zeros of type `uint32`.

If you have an array of a different type, such as `double` or `single`, then you can convert that array to an array of type `uint32` by using the `uint32` function.

Syntax

```
Y = uint32(X)
```

Description

`Y = uint32(X)` converts the values in `X` to type `uint32`. Values outside the range $[0,2^{32}-1]$ map to the nearest endpoint.

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint64` | `logical` | `char`

Examples

Convert to 32-Bit Unsigned Integer Variable

Convert a double-precision variable to a 32-bit unsigned integer.

```
x = 100;
xtype = class(x)
```

```
xtype =
'double'
```

```
y = uint32(x)
```

```
y = uint32
    100
```

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cast` | `int32` | `typecast` | `uint16` | `uint64` | `uint8`

Topics

“Integers”
“Identifying Numeric Classes”
“Hexadecimal and Binary Values”

Introduced before R2006a

uint64

64-bit unsigned integer arrays

Description

Variables in MATLAB of data type (class) `uint64` are stored as 8-byte (64-bit) unsigned integers. For example:

```
y = uint64(10);
whos y
```

```
  Name      Size            Bytes  Class     Attributes

  y         1x1                 8  uint64
```

For more information on integer types, see “Integers”.

Creation

Some array creation functions allow you to specify the data type. For instance, `zeros(100,'uint64')` creates a 100-by-100 matrix of zeros of type `uint64`.

If you have an array of a different type, such as `double` or `single`, then you can convert that array to an array of type `uint64` by using the `uint64` function.

Syntax

```
Y = uint64(X)
```

Description

`Y = uint64(X)` converts the values in `X` to type `uint64`. Values outside the range $[0,2^{64}-1]$ map to the nearest endpoint.

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `logical` | `char`

Examples

Convert to 64-Bit Unsigned Integer Variable

Convert a double-precision variable to a 64-bit unsigned integer.

```
x = 100;
xtype = class(x)
```

```
xtype =
'double'
```

```
y = uint64(x)
```

```
y = uint64
    100
```

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cast` | `int64` | `typecast` | `uint16` | `uint32` | `uint8`

Topics

“Integers”
“Identifying Numeric Classes”
“Hexadecimal and Binary Values”

Introduced before R2006a

uiopen

Open file selection dialog box and load selected file into workspace

Syntax

```
uiopen
uiopen(type)
uiopen(file)
uiopen(file,tf)
```

Description

`uiopen` opens a modal on page 1-16215 dialog box entitled Open. The file filter in the dialog box is set to all MATLAB files. If the user enters a valid file name in the **File Name** field and clicks **Open**, then MATLAB opens the specified file in the appropriate application. If the appropriate application is not installed, then either the file opens in the MATLAB Editor or an error is returned.

Note The visual characteristics of the dialog box depend on the operating system that runs your code. For instance, some operating systems do not show title bars on dialog boxes.

`uiopen(type)` sets the file filter according to the type of file specified (which is not the same as the file extension). For example, if type is `'figure'`, then MATLAB sets the filter to all figure files `(*.fig)`.

`uiopen(file)` specifies the file name that appears as the default value in the dialog box **File name** field. Only files that have the same file extension are displayed in the dialog box.

If you specify a wildcard character and a file extension (such as `*.m`), then no file name appears in the **File name** field. The dialog box displays only files with the specified extension. The `type` value appears in the filter field to the right of the **File name** field.

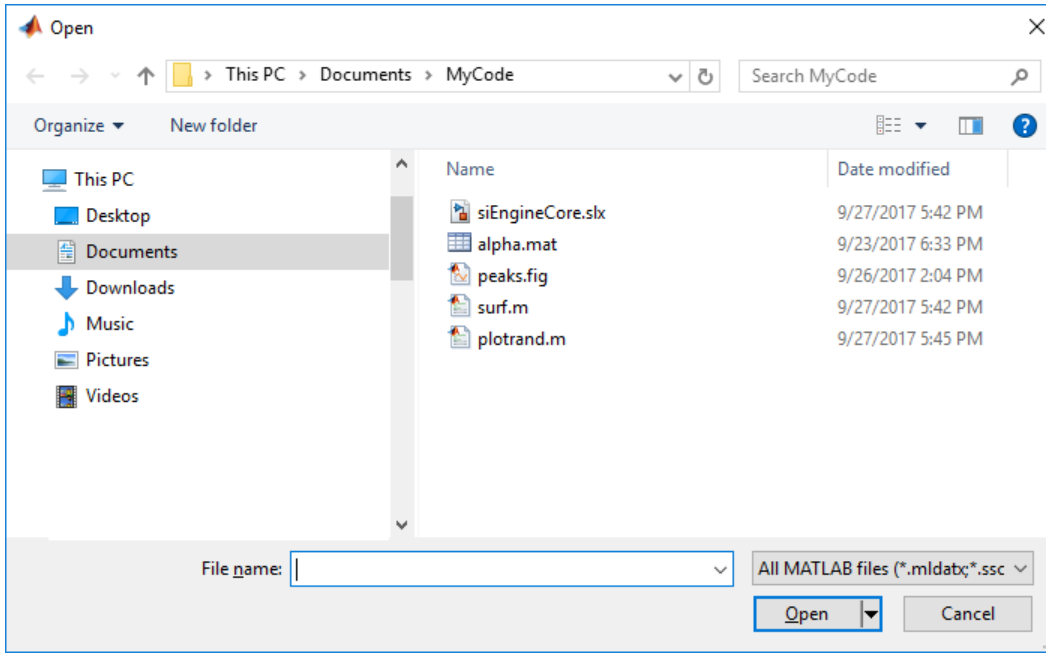
`uiopen(file,tf)` opens the specified file without displaying the Open dialog box when the value of `tf` is logical true (1). The Open dialog box is displayed when the value of `tf` is logical false (0).

Examples

Filter Files by Type

Display all the files in the current folder in the Open dialog box. Then, rerun the `uiopen` command to filter the files by the `'editor'` type.

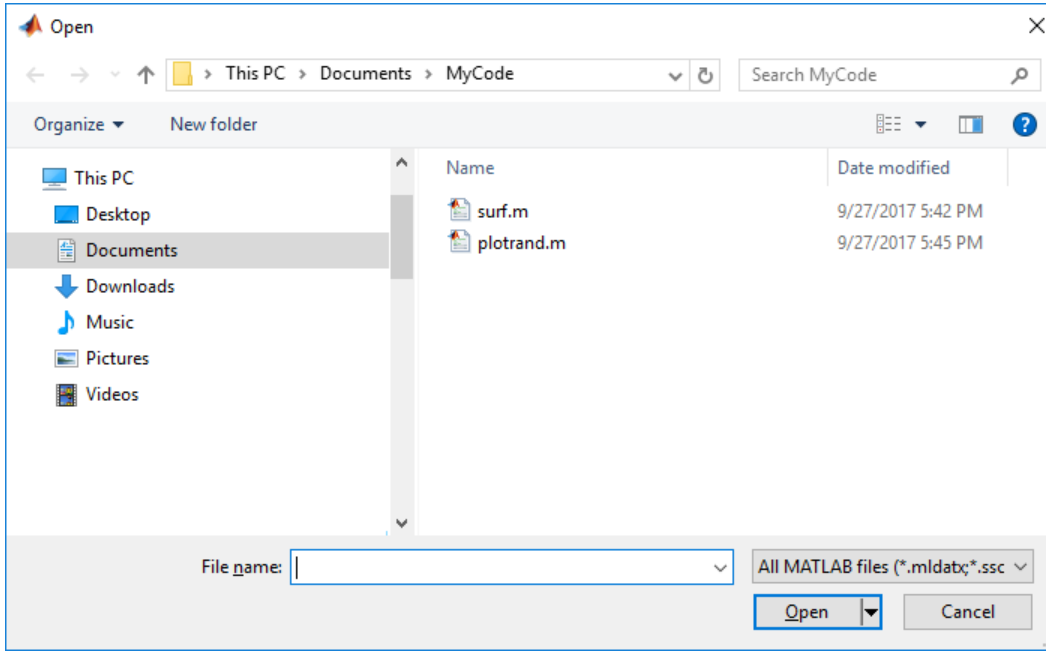
```
uiopen
```



The `uiopen` command with no arguments displays all MATLAB files in the current folder.

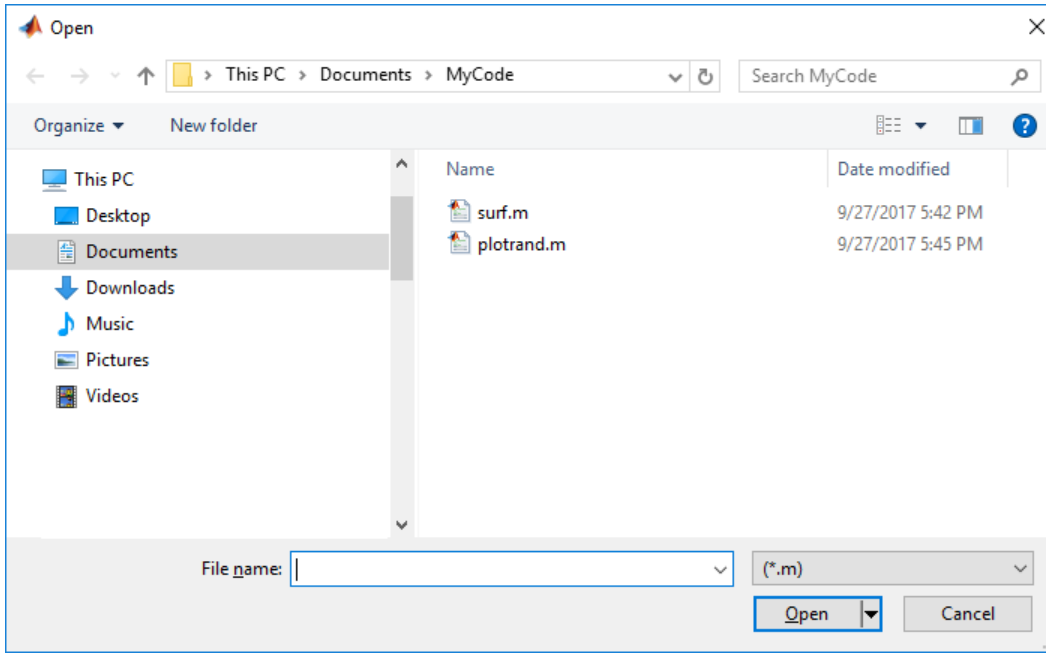
Set the `type` argument to `'editor'` to display only those files that can be opened in the Editor. FIG-files, MAT-files, SLX-files, MLAPP-files, and MLAPPINSTALL-files are excluded from the list presented in the dialog box.

```
uiopen('editor')
```



Filter Files by File Extension

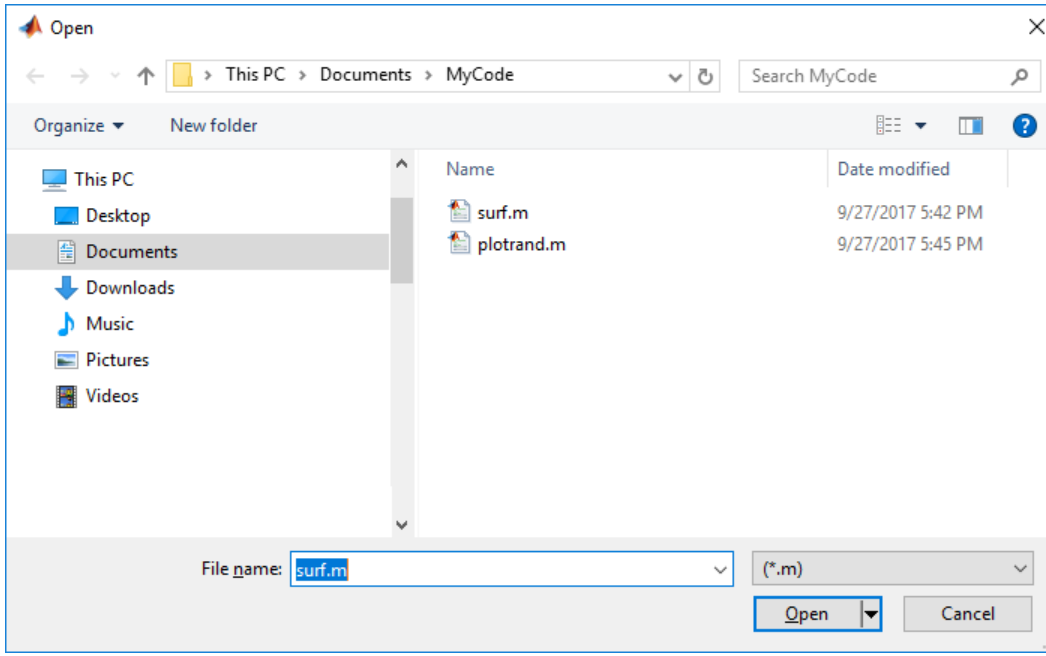
```
uiopen('*.m')
```



Specify Default File

Set the `file` input argument to a file name. That file appears in the **File name** field when the dialog box opens.

```
uiopen('surf.m')
```



If the user clicks **Open**, then `surf.m` opens in the MATLAB Editor.

If the user enters a different file in the **File name** field and then clicks **Open**, then that file opens if it is in the current folder. However, notice that only files with the same file extension as the one specified in the `uiopen` function call are listed in the Open dialog box.

Input Arguments

`type` — file filter

'matlab' (default) | 'load' | 'figure' | 'simulink' | 'editor'

File filter, specified as `'matlab'`, `'load'`,`'figure'`,`'simulink'`, or `'editor'`. The files that MATLAB displays for each type value are as follows.

Type Value	Files Displayed
`'matlab'`	All MATLAB files.

Type Value	Files Displayed
`'load'`	All MAT-files (`*.mat`)
`'figure'`	All figure files (`*.fig`)
`'simulink'`	All Simulink model files (`*.mdl` and `*.slx`)
`'editor'`	All MATLAB files except for `.mat`, `.fig`, `.slx` , `.mlapp`, and `.mlappinstall` files

The only the form of `uiopen` that you can compile into a standalone application is `uiopen('load')`. To create a file selection dialog box that you can compile, use `uigetfile`.

`file` — File name
character vector | string scalar

A file name, specified as a character vector or string scalar that includes the file extension. If `tf` is false (the default), then the file name can be a wildcard character plus a file extension. For example, `*.txt` displays a list of all files with the `.txt` file extension.

Example: `'surf.m'`

Example: `'t.fig'`

Example: `'*.mat'`

`tf` — Logical open file
0 (default) | 1 | true | false

Logical open file specified as logical true (`1`), logical false (`0`), `true`, `false`, or a MATLAB expression that evaluates to logical true or false.

If `tf` is set to `true`, `1`, or evaluates to `1`, then the Open dialog box does not appear and MATLAB attempts to open the specified file in the appropriate tool.

If `tf` is set to `false`, `0` or evaluates to `0`, then the Open dialog box appears.

Example: `1`

Example: `true`

More About

Modal Dialog Box

A modal dialog box prevents a user from interacting with other MATLAB windows before responding to the modal dialog box.

Tips

- This table lists file extensions and the application in which such a file opens when a user selects a file with that extension and clicks **Open**. If the listed tool is not installed and the file is a text file, then the file opens in the MATLAB Editor. MATLAB returns an error if Simulink is not installed and the user selects a file with an `.mdl` or `.slx` file extension.

File Extension	Where File Loads
`.m` or `.mlx`	MATLAB Editor
`.fig`	MATLAB Figure window.
`.mat`	MATLAB workspace.
`.mlapp`	MATLAB App Designer.
`.mlappinstall`	MATLAB app installer
`.mdl` or `.slx`	Simulink

- To open a file in Microsoft Windows, use the `winopen` function.

Alternatives

You can also use these functions to open a file in a MATLAB code file or in a command.

File to Open	Function
Based on file extension	`open`
In the Editor	`edit`
FIG-file	`load`

File to Open	Function
MAT-file	`openfig`

See Also

`uisave`

Introduced before R2006a

uipanel

Create panel container object

Syntax

```
p = uipanel
p = uipanel(Name,Value)
p = uipanel(parent)
p = uipanel(parent,Name,Value)
```

Description

`p = uipanel` creates a panel in the current figure and returns the `Panel` object. If there is no figure available, MATLAB calls the `figure` function to create one.

`p = uipanel(Name,Value)` specifies panel property values using one or more name-value pair arguments.

`p = uipanel(parent)` creates the panel in the specified parent container. The parent container can be a figure created with either the `figure` or `uifigure` function, or a child container such as a tab or grid layout. Property values for `uipanel` vary slightly depending on whether the app is created with the `figure` or `uifigure` function. For more information, see “Name-Value Pair Arguments” on page 1-16220.

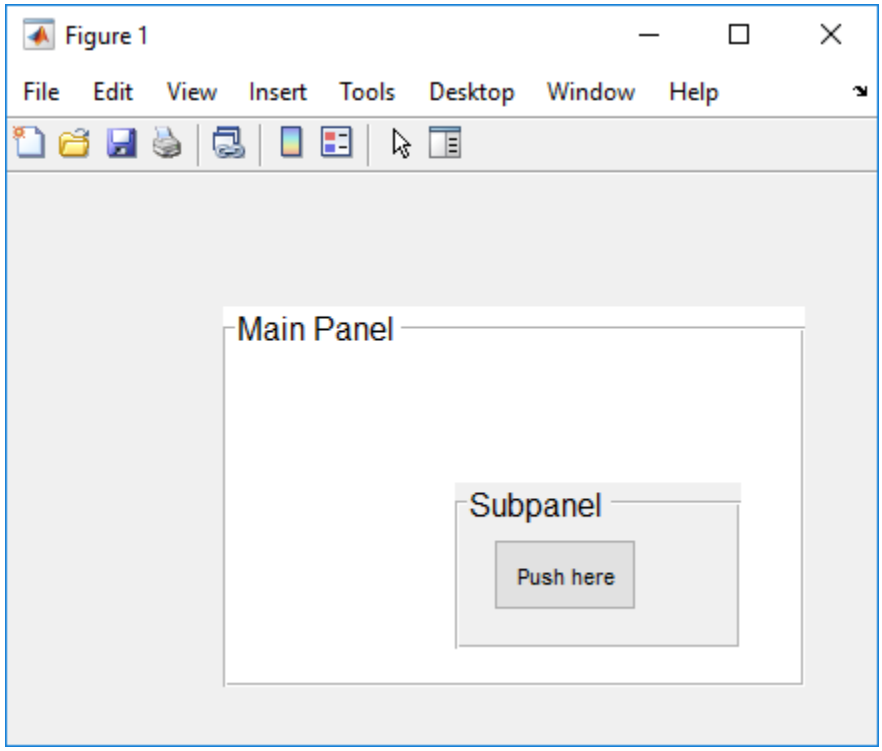
`p = uipanel(parent,Name,Value)` specifies the parent container and one or more properties values.

Examples

Nested Panels

Create a figure containing two panels and a push button. The panels use the default `Units` property value, `'normalized'`. The default units for the `uicontrol` is `'pixels'`.

```
h = figure;
hp = uipanel('Title','Main Panel','FontSize',12,...
             'BackgroundColor','white',...
             'Position',[.25 .1 .67 .67]);
hsp = uipanel('Parent',hp,'Title','Subpanel','FontSize',12,...
              'Position',[.4 .1 .5 .5]);
hbsp = uicontrol('Parent',hsp,'String','Push here',...
              'Position',[18 18 72 36]);
```

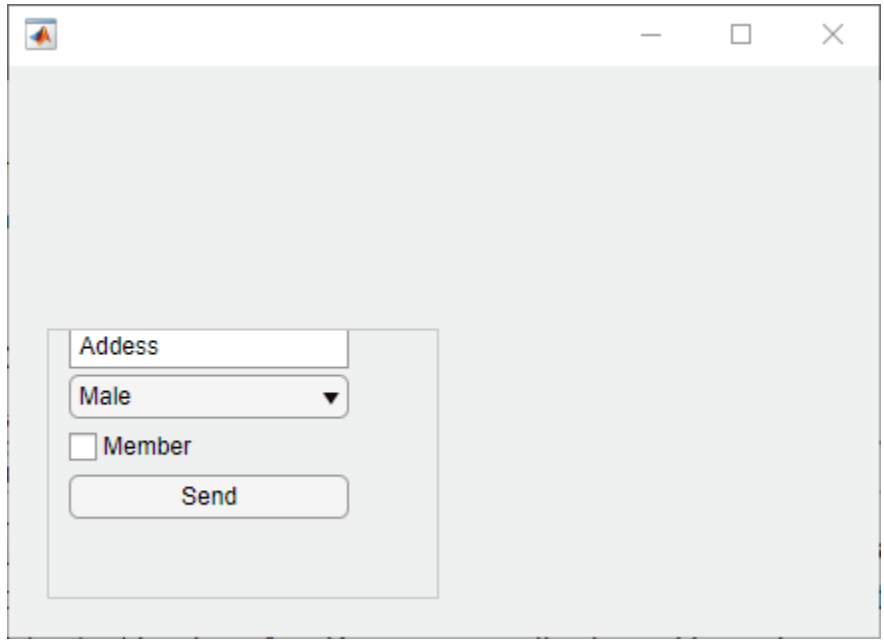


Scrollable Panel

The `Scrollable` property enables scrolling within a panel that has components outside its borders. Scrolling is only possible when the panel is in a figure created with the `uifigure` function. App Designer uses this type of figure for creating apps.

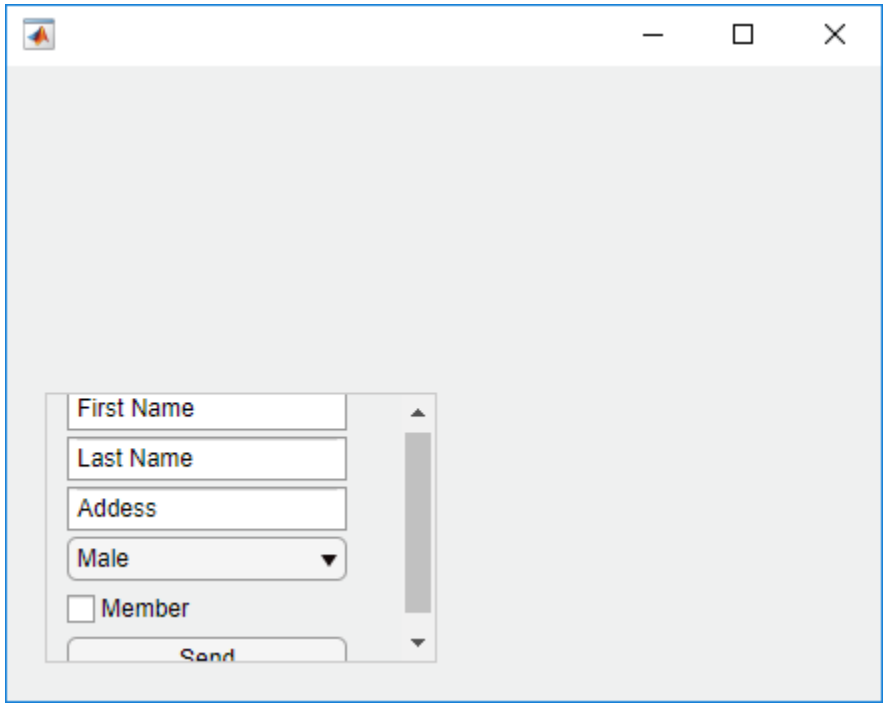
Create a panel within a figure. Add six UI components to the panel, with the first three lying outside the upper border of the panel.

```
f = uifigure;
p = uipanel(f,'Position',[20 20 196 135]);
ef1 = uieditfield(p,'text','Position',[11 165 140 22],'Value','First Name');
ef2 = uieditfield(p,'text','Position',[11 140 140 22],'Value','Last Name');
ef3 = uieditfield(p,'text','Position',[11 115 140 22],'Value','Addess');
dd = uidropdown(p,'Position',[11 90 140 22],'Items',{'Male','Female'});
cb = uicheckbox(p,'Position',[11 65 140 22],'Text','Member');
b = uibutton(p,'Position',[11 40 140 22],'Text','Send');
```



Enable scrolling by setting the `Scrollable` property of the panel to `'on'`. By default, the scroll box displays at the top.

```
p.Scrollable = 'on';
```



Input Arguments

parent — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a figure created with either the `figure` or `uifigure` function, or a child container:

- Panels, tabs, and button groups can be containers in either type of figure.
- Grid layouts can be containers only in figures created with the `uifigure` function.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (`' '`). You can specify several name and value pair arguments as `Name1,Value1,...,NameN,ValueN`.

Example: `'Title','Options'` specifies that the panel title is `Options`.

- For a list of properties available for GUIDE, or for creating apps with the `figure` function, see Uipanel for GUIDE.
- For a list of properties available for App Designer, or for creating apps with the `uifigure` function, see Panel for App Designer.

Tips

If you set the `Visible` property of a panel object to `'off'`, then any child objects it contains (buttons, button groups, axes, etc.) become invisible along with the parent panel. However, the `Visible` *property value* of each child object remains unaffected.

See Also

Properties
Panel

Introduced before R2006a

uiprogressdlg

Create progress dialog box

Syntax

```
d = uiprogressdlg(f)
d = uiprogressdlg(f,Name,Value)
```

Description

`d = uiprogressdlg(f)` displays a determinate progress dialog box in figure `f` and returns the `ProgressDialog` object. The figure must be created using the `uifigure` function.

`d = uiprogressdlg(f,Name,Value)` specifies `ProgressDialog` property values using `Name,Value` pair arguments. Use property values to control the appearance and behavior of the dialog box. For example, you can add a title or message to the dialog box, or specify an indeterminate progress bar.

Examples

Determinate Progress Bar

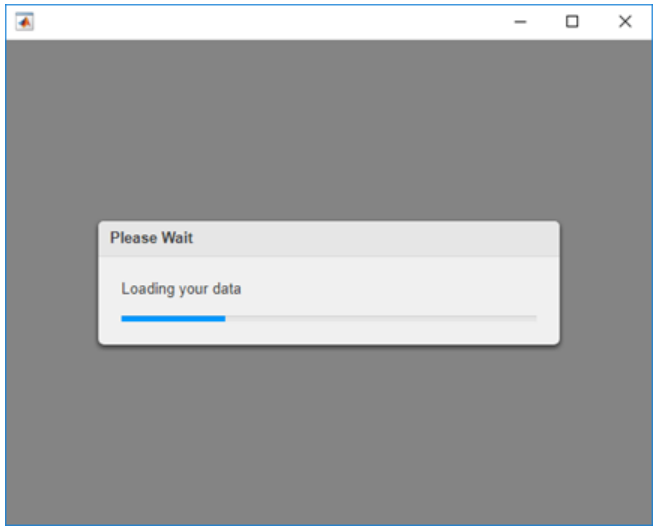
Create a program file called `myprogress1.m` that creates a figure and a progress dialog box. Update the `Value` and `Message` properties at three different points in the code.

```
function myprogress1
    f = uifigure;
    d = uiprogressdlg(f,'Title','Please Wait',...
        'Message','Opening the application');
    pause(.5)

    % Perform calculations
    % ...
```

```
    d.Value = .33;
    d.Message = 'Loading your data';
    pause(1)

    % Perform calculations
    % ...
    d.Value = .67;
    d.Message = 'Processing the data';
    pause(1)

    % Finish calculations
    % ...
    d.Value = 1;
    d.Message = 'Finishing';
    pause(1)

    % Close dialog box
    close(d);
end
```

Run the program to display the progress dialog box.

```
myprogress1
```



Indeterminate Progress Bar

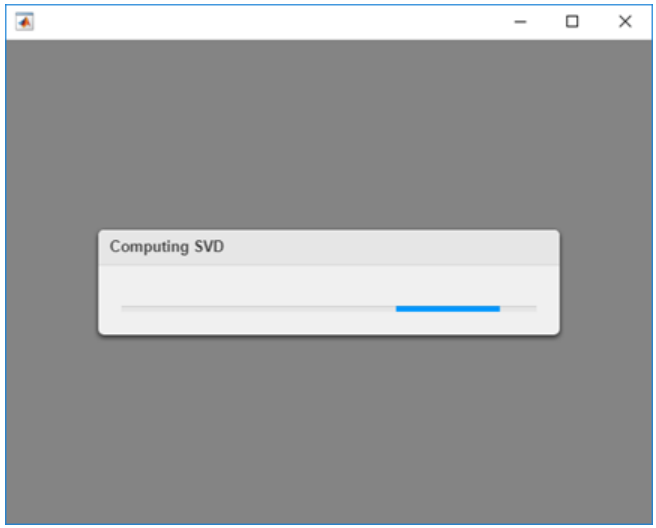
Create a program file called `myprogress2.m` that creates a figure and displays an indeterminate progress bar during a singular value decomposition.

```
function myprogress2
    f = uifigure;
    d = uiprogressdlg(f,'Title','Computing SVD',...
        'Indeterminate','on');

    % Do the SVD computation
    svd(rand(5000));

    % close the dialog box
    close(d);
end
```

Setting the `Indeterminate` property to `'on'` animates the progress bar to indicate that there is no projected completion time. After completing the calculation, the `close` function closes the dialog box.

Run the program to perform the singular value decomposition and display the progress dialog box.

```
myprogress2
```



Determinate Progress Bar with Cancel Button

Create a program file called `myprogress3.m` that creates a figure and displays a progress bar while approximating the value of pi.

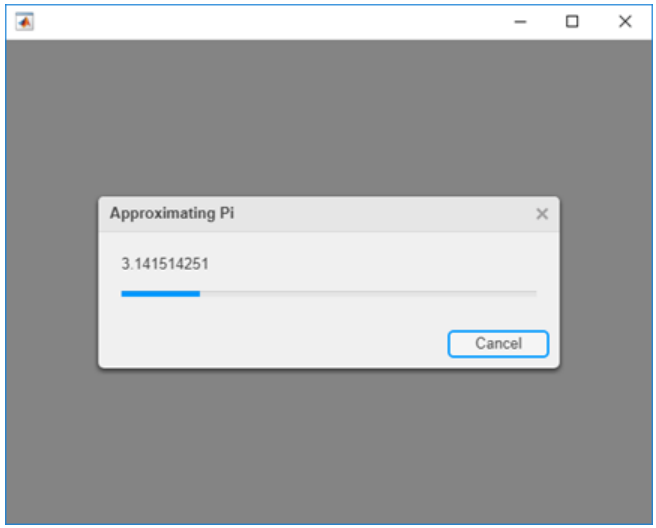
```
function myprogress3
    f = uifigure;
    d = uiprogressdlg(f,'Title','Approximating Pi',...
        'Message','1','Cancelable','on');

    % Approximate pi^2/8 as: 1 + 1/9 + 1/25 + 1/49 + ...
    pisqover8 = 1;
    denom = 3;
    valueofpi = sqrt(8 * pisqover8);
    steps = 20000;
    for step = 1:steps
        % Check for Cancel button press
        if d.CancelRequested
            break
        end
        % Update progress, report current estimate
        d.Value = step/steps;
        d.Message = sprintf('%12.9f',valueofpi);
```

```
        % Calculate next estimate
        pisqover8 = pisqover8 + 1 / (denom * denom);
        denom = denom + 2;
        valueofpi = sqrt(8 * pisqover8);
    end

    % Close the dialog box
    close(d);
end
```

Setting the `Cancelable` property to `'on'` creates a cancel button with the default label, **Cancel**. The first command in the `for` loop checks the value of `d.CancelRequested` to see if the user clicked the cancel button. If the value is `true`, the program exits the loop. Finally, the `close(d)` command closes the dialog box after the `for` loop finishes or the user cancels.

Run the program to approximate pi and display the progress dialog box.

```
myprogress3
```



Input Arguments

f — Target figure
`Figure` object

Target figure, specified as a `Figure` object. The figure must be created with the `uifigure` function.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `d = uiprogressdlg(uifigure,'Value',0.25)`

Note The properties listed here are only a subset. For a complete list, see ProgressDialog Properties.

`Value` — Fraction complete
`0` (default) | number between `0` and `1`

Fraction complete, specified as a number between `0` and `1`. The progress bar reaches its full length when the value is `1`. Change `Value` at different points in your code to provide a visual indication of progress in the running app.

Data Types: `double`

`Message` — message
`''` (default) | character vector | cell array of character vectors | string array

Message, specified as a character vector, cell array of character vectors, or string array. The message displays within the dialog box, above the progress bar.

To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array corresponds to a line of text. Hard breaks within each element, such as `'\n'`, create additional lines of text.

Example: `d = uiprogressdlg(uifigure,'Message','Calculating result.');`

`Title` — Title
`''` (default) | character vector | string scalar

Title, specified as a character vector or a string scalar. The title displays in the title bar of the dialog box.

Example: `d = uiprogressdlg(uifigure,'Title','Calculating');`

`Indeterminate` — Indeterminate progress
`'off'` (default) | `'on'`

Indeterminate progress, specified as `'off'` or `'on'`. Set this property to `'on'` to provide an animated bar without any specific progress information. This animation is useful when you do not know how long a calculation will take.

To prevent indeterminate progress bars from displaying indefinitely, call the `close` function after completing your calculations.

`Cancelable` — Allow cancellation
`'off'` (default) | `'on'`

Allow cancellation, specified as `'off'` or `'on'`. A value of `'on'` displays a cancel button in the dialog box. You can customize the button label by specifying the `CancelText` property.

When you allow cancellation, you must check the value of the `CancelRequested` property, and call the `close` function when the value is `true`. Otherwise, the dialog box displays indefinitely.

See Also

Functions
`uialert` | `uiconfirm`

Properties
ProgressDialog Properties

Introduced in R2018a

Panel Properties

Control appearance and behavior of panel

Note The properties listed here are valid for panels in GUIDE or in apps created with the `figure` function. If you are using App Designer or the `uifigure` function, see Panel Properties instead. For more information, see "GUIDE Migration Strategies".

Description

Panels are containers for grouping together UI components. Properties control the appearance and behavior of a panel. Use dot notation to refer to a particular object and property.

```
f = figure;
p = uipanel(f);
p.Position = [.1 .1 .7 .8];
```

Properties

Title

`Title` — Title

character vector | string scalar | categorical array

Title, specified as a character vector, string scalar, or categorical array. If you specify this property as a categorical array, MATLAB displays only the first element in the array.

MATLAB does not interpret a vertical slash (`'|'`) character as a line break, it displays as a vertical slash in the title.

If you want to specify a Unicode character, pass the Unicode decimal code to the `char` function. For example, `['Multiples of ' char(960)]` displays as `Multiples of` π.

`TitlePosition` — Title location

`'lefttop'` (default) | `'centertop'` | `'righttop'` | `'leftbottom'` | `'centerbottom'` | `'rightbottom'`

Title location, specified as `'lefttop'`, `'centertop'`, `'righttop'`, `'leftbottom'`, `'centerbottom'`, or `'rightbottom'`.

Color and Styling

`ForegroundColor` — Title color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Title color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: [0 0 1]

Example: 'b'

Example: 'blue'

BackgroundColor — Background color
[.94 .94 .94] (default) | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

BorderType — Border type
'etchedin' (default) | 'etchedout' | 'beveledin' | 'beveledout' | 'line' | 'none'

Border type, specified as 'etchedin', 'none', 'etchedout', 'beveledin', 'beveledout', or 'line'.

- For a 3-D appearance, use etched or beveled borders.

  Use the HighlightColor and ShadowColor properties to specify the color of 3-D borders.
- For a simpler appearance, use a line border.

  Use the HighlightColor property to specify the line border color.

`BorderWidth` — Border width

1 (default) | positive integer value

Border width, specified as a positive integer value. The unit of measurement is pixels. Etched and beveled borders wider than three pixels might not appear correctly at the corners.

`HighlightColor` — Border highlight color

RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Border highlight color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`ShadowColor` — Border shadow color

RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Border shadow color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`Clipping` — Child component clipping (not recommended)
`'on'` (default) | `'off'`

Note The behavior of the `Clipping` property has changed. It no longer has any effect on `Panel` objects. Child objects are now clipped to the boundaries of their parent container regardless of the value of this property. This property might be removed in a future release.

Font

`FontName` — Font name
system supported font name | `'FixedWidth'`

Font name, specified as a system supported font name or `'FixedWidth'`. The default font depends on the specific operating system and locale.

To use a fixed-width font that looks good in any locale, specify `'FixedWidth'`. The actual fixed-width font used depends on the `FixedWidthFontName` property of the root object. Changing the `FixedWidthFontName` property causes an immediate update of the display to use the new font.

Example: `'Arial'`

`FontSize` — Font size

positive number

Font size, specified as a positive number. The `FontUnits` property specifies the units. The default size is system-dependent.

Example: `12`

Example: `12.5`

`FontWeight` — Font weight

`'normal'` (default) | `'bold'`

Font weight, specified as a value from the following table.

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than normal

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold font weight. Therefore, specifying a bold font weight still can result in the normal font weight.

Note The `'light'` and `'demi'` font weight values have been removed in R2014b. If you specify either of these values, the result is a normal font weight.

`FontAngle` — Font angle

`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. MATLAB uses this property to select a font from those available on your system. Setting this property to `'italic'` selects a slanted version of the font, if it is available on your system.

Note The `'oblique'` value has been removed. Use `'italic'` instead.

FontUnits — Font units

`'points'` (default) | `'normalized'` | `'inches'` | `'centimeters'` | `'pixels'`

Font units, specified as one of the values from this table.

Units Value	Description
`'points'`	Points. One point is 1/72nd of an inch.
`'normalized'`	Normalized values for specifying the font size as a fraction of the height. When you resize a UI component, MATLAB scales the displayed font to maintain that fraction.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.

Interactivity

Visible — Panel visibility

`'on'` (default) | `'off'`

Panel visibility, specified as `'on'` or `'off'`. The `Visible` property determines whether the panel displays on the screen. If the `Visible` property is set to `'off'`, the entire panel is invisible, but you can still specify and access its properties.

Changing the size of an invisible container triggers the `SizeChangedFcn` callback when it becomes visible.

Note Changing the `Visible` property of a container does *not* change the `Visible` property of its child components even though hiding the container prevents its children from displaying.

UIContextMenu — Context menu

empty `GraphicsPlaceholder` array (default) | uicontextmenu object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when the user right-clicks on a UI component. Create the context menu using the `uicontextmenu` function.

Selected — Selection state (not recommended)

`'off'` (default) | `'on'`

Note The behavior of the `Selected` property changed in R2014b, and it is not recommended. It no longer has any effect on `Panel` objects. This property might be removed in a future release.

SelectionHighlight — Display of selection handles (not recommended)

`'on'` (default) | `'off'`

Note The behavior of the `SelectionHighlight` property changed in R2014b, and it is not recommended. It no longer has any effect on `Panel` objects. This property might be removed in a future release.

Position

Position — Location and size (including borders and title)

`[left bottom width height]`

Location and size (including borders and title), specified as the four-element vector of the form `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the panel
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the panel
`width`	Distance between the right and left outer edges of the panel

Element	Description
`height`	Distance between the top and bottom outer edges of the panel

All measurements are in units specified by the `Units` property.

Note The `Position` values are relative to the parent container's drawable area. The drawable area is the area inside the borders of the container and does not include the area occupied by the title. If the parent container is a figure, then the drawable area also excludes the menu bar and tool bar.

Example: Modify One Value in the Position Vector

You can combine dot notation and array indexing when you want to change one value in the `Position` vector. For example, this code changes the width of the panel to `0.5`:

```
p = uipanel;
p.Position(3) = 0.5;
p.Position
```

```
ans =

      0   0   0.5000  1.0000
```

`InnerPosition` — Location and size (excluding borders and title)

`[left bottom width height]`

This property is read-only.

Location and size (excluding borders and title), returned as a four-element vector of the form `[left bottom width height]`. This table describes each element in the vector.

Value	Description
`left`	Distance from the inner left edge of the parent container to the inner left edge of the container.
`bottom`	Distance from the inner bottom edge of the parent container to the inner bottom edge of the container.

Value	Description
`width`	Distance between the inner edges of the container's right and left borders.
`height`	Distance between the inner edges of the container's top and bottom borders. This distance excludes the title, if it exists.

All measurements are in units specified by the `Units` property.

Note These are some important points to consider when using the `InnerPosition` property:

- `InnerPosition` values are affected by the presence of a title, font characteristics, `BorderType`, and `BorderWidth`.
- `InnerPosition` values are relative to the parent container's drawable area. The drawable area is the area inside the borders of the container and exclude the area occupied by the title. If the parent container is a figure, then the drawable area also excludes the menu bar and tool bar.

`OuterPosition` — Location and size (including borders and title)

`[left bottom width height]`

Location and size (including borders and title), specified as a four-element vector of the form `[left bottom width height]`. All measurements are in units specified by the `Units` property.

This property value is identical to the `Position` property value.

`Units` — Units of measurement

`'normalized'` (default) | `'pixels'` | `'inches'` | `'centimeters'` | `'points'` | `'characters'`

Units of measurement, specified one of the values from this table.

Units Value	Description
`'normalized'`	These units are normalized with respect to the parent container. The lower-left corner of the container maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'points'`	Points. One point equals 1/72nd of an inch.
`'characters'`	These units are based on the default uicontrol font of the graphics root object: • Character width = width of the letter `x`. • Character height = distance between the baselines of two lines of text. To access the default uicontrol font, use `get(groot,'defaultuicontrolFontName')` or `set(groot,'defaultuicontrolFontName')`.

MATLAB measures all units from the lower left corner of the parent object.

If you change the value of the `Units` property, it is good practice to return it to its default value after completing your computation to avoid affecting other functions that assume the `Units` property is set to the default value.

The order in which you specify the Units and Position properties has these effects:

- If you specify the `Units` property before the `Position` property, then MATLAB sets `Position` using the units you specified.
- If you specify the `Units` property after the `Position` property, MATLAB sets the position using the default `Units`. Then, MATLAB converts the `Position` values to the equivalent values in the units you specified.

Callbacks

`SizeChangedFcn` — Size change callback function

`' '` (default) | function handle | cell array | character vector

Size change callback function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

Define this callback to customize the app layout when the size of this container changes (e.g., when the user resizes the window).

The `SizeChangedFcn` callback executes when:

- This container becomes visible for the first time.
- This container is visible while its size changes.
- This container becomes visible for the first time after its size changes. This situation occurs when the size changes while the container is invisible, and then it becomes visible later.

Other important points to consider when defining a `SizeChangedFcn` callback:

- Consider delaying the display of this container until after all the variables that the `SizeChangedFcn` uses are defined. This practice can prevent the `SizeChangedFcn` callback from returning an error. To delay the display of the container, set its `Visible`

property to `'off'`. Then, set the `Visible` property to `'on'` after you define the variables that your `SizeChangedFcn` callback uses.

- If your app contains nested containers, they resize from the inside out.
- To access the container that is resizing from within the `SizeChangedFcn`, refer to the source object (the first input argument in the callback) or use the `gcbo` function.

Tip As an easy alternative to specifying a `SizeChangedFcn` callback, you can set the `Units` property of all the objects you put inside a container to `'normalized'`. Doing so makes those components scale proportionally with the container.

See “Lay Out a UI Programmatically” for more information about managing layouts with `SizeChangedFcn` callbacks.

`ButtonDownFcn` — Button-press callback function

`''` (default) | function handle | cell array | character vector

Button-press callback function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

The `ButtonDownFcn` callback is a function that executes when the user clicks a mouse button within the container.

`CreateFcn` — Component creation function

`''` (default) | function handle | cell array | character vector

Component creation function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

This property specifies a callback function to execute when MATLAB creates the component. MATLAB initializes all component property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Use the `gcbo` function in your `CreateFcn` code to get the component object that is being created.

Setting the `CreateFcn` property on an existing component object has no effect.

`DeleteFcn` — Component deletion function
`''` (default) | function handle | cell array | character vector

Component deletion function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

The `DeleteFcn` property specifies a callback function to execute when MATLAB deletes the component (for example, when the user closes the window). MATLAB executes the `DeleteFcn` callback before destroying the properties of the component object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

Use the `gcbo` function in your `DeleteFcn` code to get the component object that is being deleted.

`ResizeFcn` — Resize callback function (not recommended)
`''` (default) | function handle | cell array | character vector

Resize callback function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

Note Use of the `ResizeFcn` property is not recommended. It might be removed in a future release. Use `SizeChangedFcn` instead.

Data Types: `function_handle` | `cell` | `char`

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed:

- A value of `'on'` allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.
    - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
    - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.

- A value of `'off'` blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn`, or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.
- MATLAB does not save the state of properties or the display when an interruption occurs. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

See “Interrupt Callback Execution” for an example that shows how the `Interruptible` and `BusyAction` properties affect the behavior of a program.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` (default) or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

The `BusyAction` property of the source of the interrupting callback determines how MATLAB handles its execution. The `BusyAction` property has these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Do not execute the interrupting callback.

Whenever MATLAB invokes a callback, that callback always attempts to interrupt an executing callback. The `Interruptible` property of the object whose callback is running determines if interruption is allowed. If `Interruptible` is set to:

- `on` — Interruption occurs at the next point where MATLAB processes the queue. This is the default.
- `off` — The `BusyAction` property (of the object owning the interrupting callback) determines if MATLAB enqueues or ignores the interrupting callback.

See “Interrupt Callback Execution” for an example that shows how the `BusyAction` and `Interruptible` properties affect the behavior of a program.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

`HitTest` — Ability to become current object
`'on'` (default) | `'off'`

Ability to become current object, specified as `'on'` or `'off'`:

- `'on'` — Sets the current object to the `Panel` when the user clicks the component in the running app. Both the `CurrentObject` property of the `Figure` and the `gco` function return the `Panel` as the current object.
- `'off'` — Sets the current object to be the closest ancestor of the `Panel` whose `HitTest` is `'on'` when the user clicks the component in the running app.

Parent/Child

`Parent` — Parent object
`Figure` | `Panel` | `ButtonGroup` | `Tab`

Parent object, specified as a `Figure`, `Panel`, `ButtonGroup`, or `Tab` object. Use this property to specify the parent container when creating a UI component or to move an existing UI component to a different parent container.

`Children` — Panel children

empty `GraphicsPlaceholder` array (default) | 1-D array of component objects

Panel children, returned as an empty `GraphicsPlaceholder` or a 1-D array of component objects. The children of `Panel` objects can be `Axes`, `Panel`, `ButtonGroup`, or any style of `UIControl` objects.

You cannot add or remove children using the `Children` property. Use this property to view the list of children or to reorder the children. The order of the children reflects the front-to-back order (stacking order) of the components on the screen. MATLAB might not allow you to change the order of certain objects. For example, `UIControl` and `Legend` objects are always in front of `Axes` objects.

To add a child to this list, set the `Parent` property of the child component to be the `Panel` object.

Objects with the `HandleVisibility` property set to `'off'` do not list in the `Children` property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object handle in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. The `HandleVisibility` property also controls the visibility of the object's handle in the parent figure's `CurrentObject` property. Handles are still valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object handle is always visible.

HandleVisibility Value	Description
`'callback'`	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.

Set the graphics root `ShowHiddenHandles` property to `'on'` to make all handles visible, regardless of their `HandleVisibility` value. This setting has no effect on their `HandleVisibility` values.

Identifiers

Type — Type of graphics object
`'uipanel'`

This property is read-only.

Type of graphics object, returned as `'uipanel'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any array. Specifying `UserData` can be useful for sharing data within apps. See “Share Data Among Callbacks” for more information.

See Also

`figure` | `guide` | `uipanel`

Topics

"Access Property Values"
"Default Property Values"

Introduced before R2006a

uipushtool

Create push button on toolbar

Note

Use this function only with GUIDE, or with figures created using the `figure` function.

Syntax

```
p = uipushtool
p = uipushtool(Name,Value,...)
p = uipushtool(parent)
p = uipushtool(parent,Name,Value,...)
```

Description

`p = uipushtool` creates a uipushtool in the uitoolbar of the current figure and returns the uipushtool object, `p`. If there is no uitoolbar available, then MATLAB creates a new uitoolbar in the current figure to serve as the parent. Similarly, if there no figure is available, then MATLAB creates a new figure with a uitoolbar.

`p = uipushtool(Name,Value,...)` creates a uipushtool and specifies one or more uipushtool property names and corresponding values. Use this syntax to override the default uipushtool properties.

`p = uipushtool(parent)` creates a uipushtool and designates a specific parent object. The `parent` argument must be a uitoolbar object.

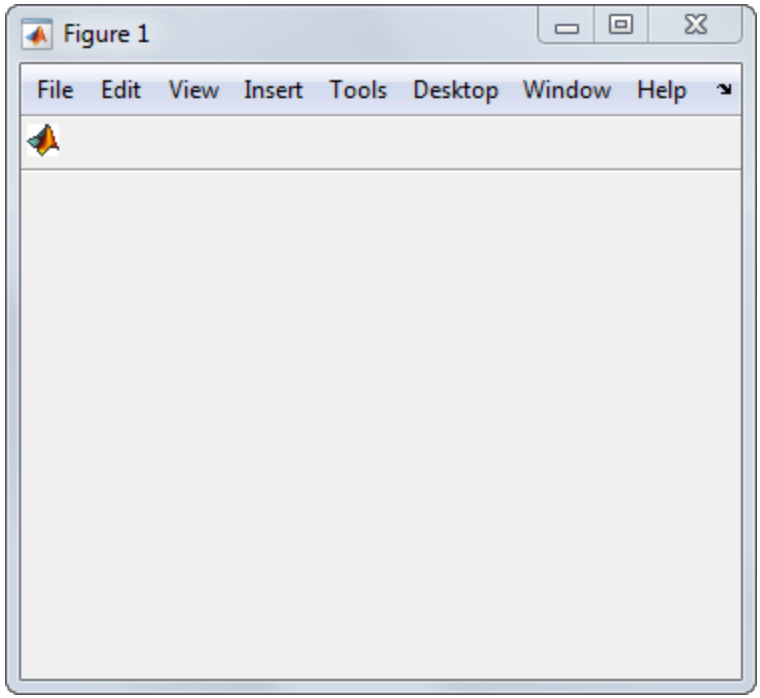
`p = uipushtool(parent,Name,Value,...)` creates a uipushtool with a specific parent and one or more uipushtool properties.

A uipushtool is a push button that appears in the figure tool bar. The button has no icon, but its borders highlight when the user hovers over it with the mouse. You can create a button icon by setting the `CData` property of the uipushtool.

Examples

Create uipushtool With Icon and Tool Tip

```
f  = figure('ToolBar','none');
t = uitoolbar(f);

% Read an image
[img,map] = imread(fullfile(matlabroot,...
            'toolbox','matlab','icons','matlabicon.gif'));

% Convert image from indexed to truecolor
icon = ind2rgb(img,map);

% Create a uipushtool in the toolbar
p = uipushtool(t,'TooltipString','Toolbar push button',...
                 'ClickedCallback',...
                 'disp(''Clicked uipushtool.'')');

% Set the button icon
p.CData = icon;
```



Tips

- Uitoolbars (and their child uipushtools) do not appear in figures whose `WindowStyle` property is set to `'Modal'`. If a figure containing a uitoolbar is changed to `'Modal'`, the uitoolbar still exists in the `Children` property of the figure. However, the uitoolbar does not display while `WindowStyle` is set to `'Modal'`.
- Unlike uicontrol push buttons, uipushtools do not set the figure's `SelectionType` property to `'open'` on the second click.

See Also

Uipushtool | `uitoggletool` | `uitoolbar`

Topics

"Access Property Values"

“Create Toolbars for Programmatic Apps”

Introduced before R2006a

PushTool Properties

Control appearance and behavior of push tool

Note

The properties listed here are valid for `PushTool` objects in GUIDE or in apps created with the `figure` function.

Description

Push tools are push buttons that appear on the tool bar at the top of the a figure. The `uipushtool` function creates a push button on a tool bar and sets any required properties before displaying it. By changing property values, you can modify certain aspects of its appearance and behavior. Use dot notation to refer to a specific object and property.

```
p = uipushtool;
p.Separator = 'on';
```

Properties

Button Appearance

`CData` — Optional icon

3-D array of truecolor RGB values

Optional icon, specified as a 3-D array of truecolor RGB values. The values in the array can be:

- Double-precision values between `0.0` and `1.0`
- `uint8` values between `0` and `255`

The length of the array's first and second dimensions must be less than or equal to 16. Otherwise, it might be clipped or distorted when it displays.

Data Types: `double` | `uint8`

`Separator` — Separator line mode

`'off'` (default) | `'on'`

Separator line mode, specified as `'off'` or `'on'`. Setting this property to `'on'` draws a dividing line to left of a tool in the tool bar.

Interactivity

`Visible` — Component visibility

`'on'` (default) | `'off'`

Component visibility, specified as `'on'` or `'off'`. When the `Visible` property is set to `'off'`, the component is not visible in the UI, but you can query and set its properties.

`Enable` — Operational state of tool

`'on'` (default) | `'off'`

Operational state of tool, specified as `'on'` or `'off'`. The `Enable` property controls whether the tool responds to button clicks. The are two possible values:

- `'on'` – The tool is operational.
- `'off'` – The tool is not operational and appears grayed out.

The value of the `Enable` property and the type of button click determine the response.

Enable Value	Response to Left-Click	Response to Right-Click
`'on'`	The `ClickedCallback` function executes.	The tool is not operational. No callback executes.
`'off'`	The tool is not operational. No callback executes.	The tool is not operational. No callback executes.

`Tooltip` — Tooltip

character vector | string scalar | categorical array

Tooltip, specified as a character vector, string scalar, or categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip does not display when the component is disabled. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

To create multiple lines of text, use the `sprintf` function to insert newline characters (`'\n'`) in your text. For example:

```
txt = sprintf('Line 1\nLine 2');
```

Then set the `Tooltip` property to the value returned by `sprintf`.

`UIContextMenu` — Context menu (not recommended)
empty `GraphicsPlaceholder` array (default) | uicontextmenu handle

This property has no effect on `PushTool` objects.

`TooltipString` — Tooltip (not recommended)
character vector | string scalar | categorical array

Tooltip, specified as a character vector, string scalar, or categorical array. The tooltip displays when the user hovers the mouse pointer over the component in the app. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Note The `TooltipString` property is not recommended starting in R2018b. Use the `Tooltip` property instead.

Callbacks

`ClickedCallback` — Clicked callback
`''` (default) | function handle | cell array | character vector

Clicked callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying callback functions as function handles, cell arrays, or character vectors, see “How to Specify Callback Property Values”.

`CreateFcn` — Component creation function
`''` (default) | function handle | cell array | character vector

Component creation function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

This property specifies a callback function to execute when MATLAB creates the component. MATLAB initializes all component property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Use the `gcbo` function in your `CreateFcn` code to get the component object that is being created.

Setting the `CreateFcn` property on an existing component object has no effect.

`DeleteFcn` — Component deletion function

`''` (default) | function handle | cell array | character vector

Component deletion function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

The `DeleteFcn` property specifies a callback function to execute when MATLAB deletes the component (for example, when the user closes the window). MATLAB executes the `DeleteFcn` callback before destroying the properties of the component object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

Use the `gcbo` function in your `DeleteFcn` code to get the component object that is being deleted.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed:

- A value of `'on'` allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- A value of `'off'` blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn`, or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

- MATLAB does not save the state of properties or the display when an interruption occurs. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

See “Interrupt Callback Execution” for an example that shows how the `Interruptible` and `BusyAction` properties affect the behavior of a program.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` (default) or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

The `BusyAction` property of the source of the interrupting callback determines how MATLAB handles its execution. The `BusyAction` property has these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Do not execute the interrupting callback.

Whenever MATLAB invokes a callback, that callback always attempts to interrupt an executing callback. The `Interruptible` property of the object whose callback is running determines if interruption is allowed. If `Interruptible` is set to:

- `on` — Interruption occurs at the next point where MATLAB processes the queue. This is the default.
- `off` — The `BusyAction` property (of the object owning the interrupting callback) determines if MATLAB enqueues or ignores the interrupting callback.

See “Interrupt Callback Execution” for an example that shows how the `BusyAction` and `Interruptible` properties affect the behavior of a program.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

`HitTest` — Ability to become current object (not recommended)
`'on'` (default) | `'off'`

This property has no effect on `PushTool` objects.

Parent/Child

`Parent` — Parent object
`Toolbar` object

Parent object, specified as a `Toolbar` object. Use this property to specify the parent tool bar when creating a tool or to move an existing tool to a different tool bar.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'callback'` | `'off'`

Visibility of object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object handle in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. The `HandleVisibility` property also controls the visibility of the object’s handle in the parent figure's `CurrentObject` property. Handles are still valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object handle is always visible.
`'callback'`	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.

HandleVisibility Value	Description
`'off'`	The object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.

Set the graphics root `ShowHiddenHandles` property to `'on'` to make all handles visible, regardless of their `HandleVisibility` value. This setting has no effect on their `HandleVisibility` values.

Identifiers

Type — Type of graphics object
`'uipushtool'`

This property is read-only.

Type of graphics object, returned as `'uipushtool'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any array. Specifying `UserData` can be useful for sharing data within apps. See “Share Data Among Callbacks” for more information.

See Also

`uipushtool` | `uitoolbar`

Topics

“Access Property Values”

“Default Property Values”

Introduced before R2006a

uiputfile

Open dialog box for saving files

Syntax

```
file = uiputfile
[file,path] = uiputfile
[file,path,indx] = uiputfile
___ = uiputfile(filter)
___ = uiputfile(filter,title)
___ = uiputfile(filter,title,defname)
```

Description

`file = uiputfile` opens a modal on page 1-16280 dialog box for selecting or specifying a file. The dialog box lists the files and folders in the current folder.

- If the user specifies a valid file name and clicks **Save**, then MATLAB returns that file name in `file`.
- If the user cancels the dialog box, then MATLAB returns `0` to `file`.

Note

- Successful execution of `uiputfile` returns the name of a new or existing file that the user specifies. It does *not* create a file.
- The visual characteristics of the dialog box depend on the operating system that runs your code. For instance, some operating systems do not show title bars on dialog boxes. If you pass a dialog box title to the `uiputfile` function, those operating systems do not display the title.

`[file,path] = uiputfile` returns the selected or specified file path to `path`. If the user cancels the dialog box, then MATLAB returns `0` to both output arguments.

`[file,path,indx] = uiputfile` returns the index of the **Save as type** value selected in the dialog box. Indexing starts at 1. If the user clicks the **Cancel** button or the window close button (X), or if the file does not exist, then MATLAB returns `0` to `indx`. If the user cancels the dialog box, then MATLAB returns `0` to all output arguments.

`___ = uiputfile(filter)` displays only those files with extensions that match `filter`. On some platforms `uiputfile` also displays any files that do not match `filter`, but they are dimmed. The `uiputfile` function appends **All Files** to the list of file types.

If `filter` is a file name, then `uiputfile` displays the file name in the **File name** field and uses the file extension as the default filter.

Use this syntax with any of the output argument combinations in the previous syntaxes.

`___ = uiputfile(filter,title)` opens a dialog box with the specified title. To filter using the default file filter, but specify a custom title, use empty quotes for the filter value.

```
file = uiputfile('','Select a File')
```

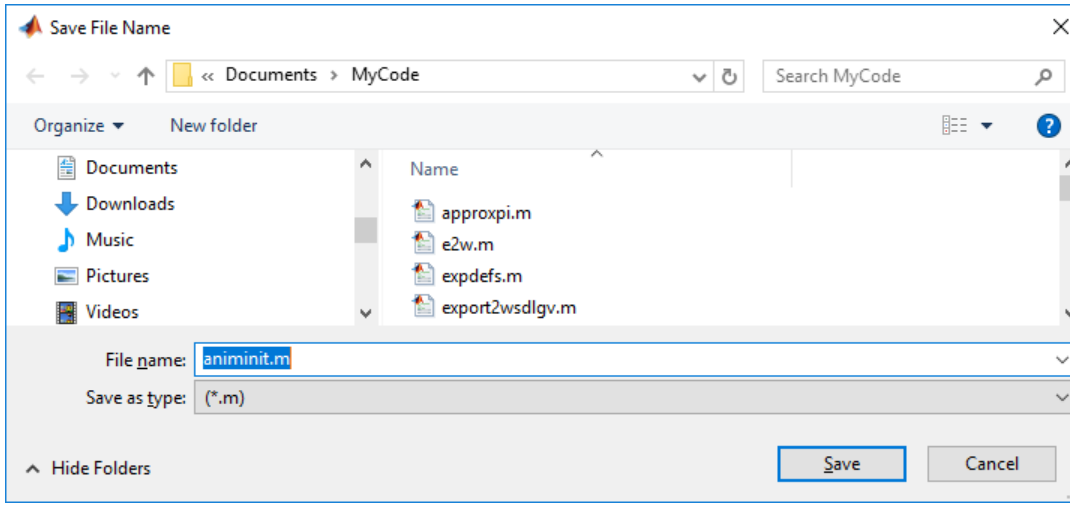
`___ = uiputfile(filter,title,defname)` opens a dialog box where the file name specified by `defname` appears in the **File name** field.

Examples

Specify File Name and Type

Create a dialog box and specify the `filter` as `animinit.m`. When the code runs, the **File name** field contains the specified file name and the **Save as type** field is set to `*.m`.

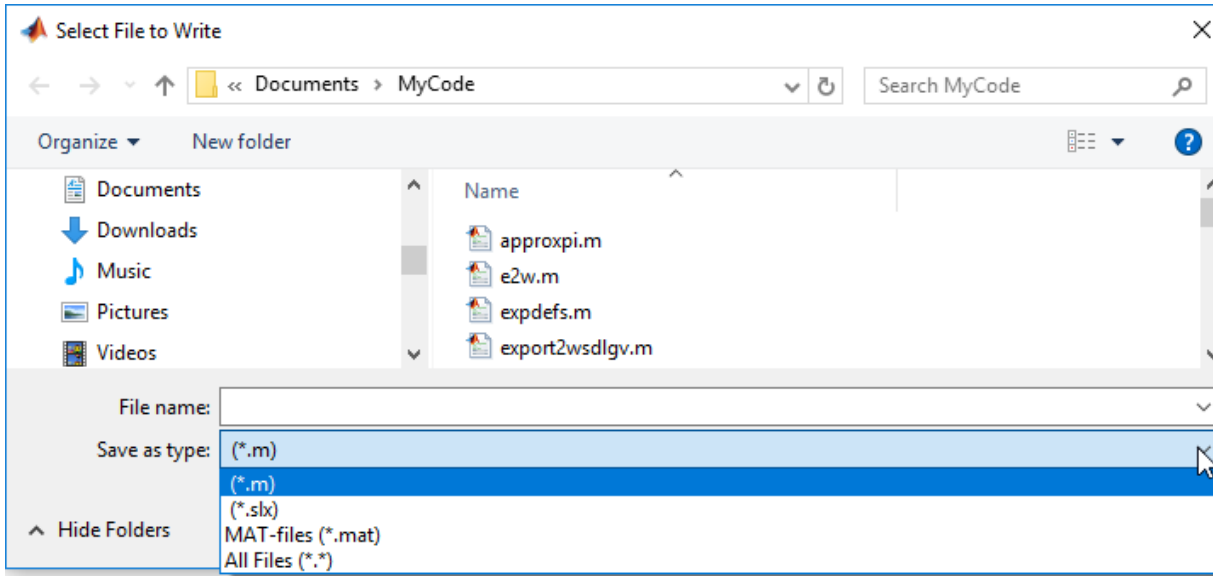
```
[file,path,indx] = uiputfile('animinit.m');
```



Display Several File Types as Filters

Display several file types in the **Save as type** list box, by separating each file extension in the `filter` input argument with a semicolon.

```
filter = {'*.m';'*.slx';'*.mat';'*.*'};
[file, path] = uiputfile(filter);
```

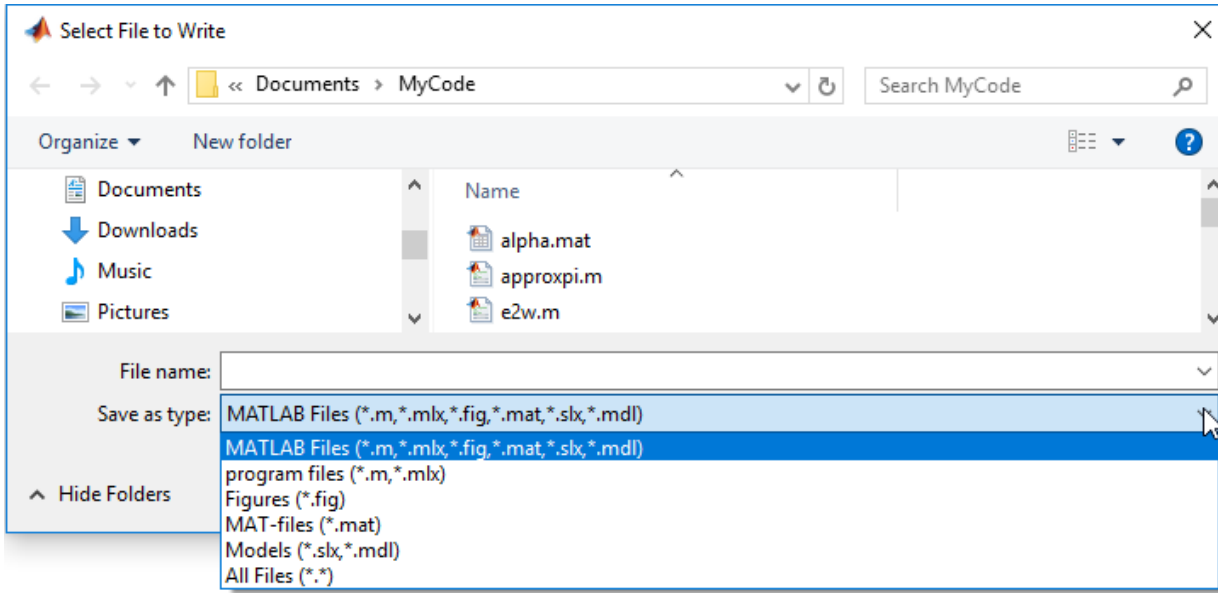


Specify File Types with Custom Descriptions

Create a list of file types and give them descriptions that are different from the defaults by using a cell array for the `filter` input value. Associate multiple file types with the `'MATLAB Files'` and `'Models'` descriptions.

The first column of the input `filter` cell array contains the file extensions, and the second contains the descriptions of the file types. For example, the first entry of column 1 contains several extensions separated by semicolons. These file types are all associated with the description `'MATLAB Files (*.m,*.mlx,*.fig,*.mat,*.slx,*.mdl)'`.

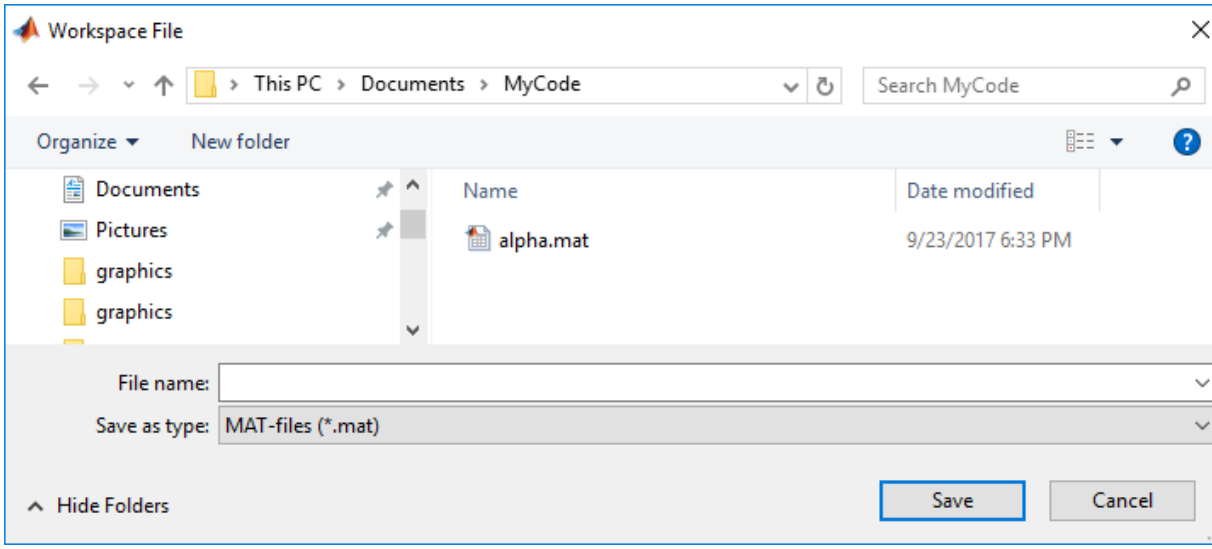
```
[filename, pathname, filterindex] = uiputfile( ...
{'*.m;*.fig;*.mat;*.slx;*.mdl',...
 'MATLAB Files (*.m,*.mlx,*.fig,*.mat,*.slx,*.mdl)';
 '*.m;*.mlx', 'program files (*.m,*.mlx)';...
 '*.fig','Figures (*.fig)';...
 '*.mat','MAT-files (*.mat)';...
 '*.slx;*.mdl','Models (*.slx,*.mdl)';...
 '*.*',  'All Files (*.*)'});
```



Specify Dialog Box Title and File Type

Create a dialog box entitled `'Workspace File'` with the **Save as type** field set to MAT-files.

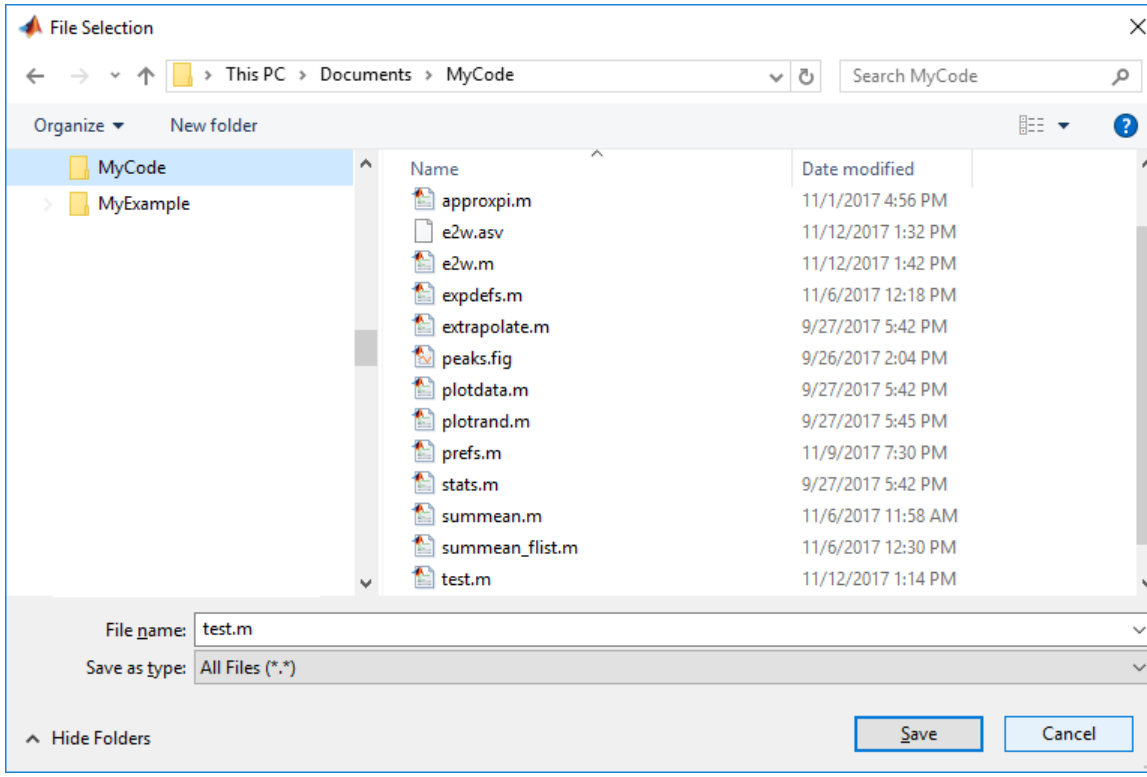
```
[file,path] = uiputfile('*.mat','Workspace File');
```



Specify Filter and Default File Name

Specify a wildcard for the filter and a default file name to display a default file in the **File name** field, but enable users to view files of all types in the dialog box.

```
[file,name,path] = uiputfile('*.*','File Selection','test.m');
```



Display User Action in Command Window

Open the Select a File to Write dialog box, and then select a file. MATLAB automatically opens a Confirm Save As dialog box.

- If you click **OK** in the confirmation dialog box, then MATLAB closes both dialog boxes and displays your selection in the Command Window.
- If you click **No** in the confirmation dialog box and click **Cancel** in the Select a File to Write dialog box, then the Command Window displays `User clicked Cancel.`

```
[file,path] = uiputfile('*.m');
if isequal(file,0) || isequal(path,0)
   disp('User clicked Cancel.')
else
```

```
   disp(['User selected ',fullfile(path,file),...
         ' and then clicked Save.'])
end
```

Input Arguments

defname — Default file name

character vector | string scalar

Default file name to display in the **File name** field when the dialog box opens, specified as a character vector or a string scalar.

The value of `defname` can include a path or consist of a path only. You can use any of these characters in the `defname` argument:

```
.
..
\
/
~
```

To specify `defname` as a folder only, specify the last character in `defname` as a backslash `\` or a forward slash `/`. When you do so, MATLAB opens the dialog box in the folder specified by the path. If you specify a path that does not exist, then MATLAB opens the dialog box in the current folder.

Example: `'myfile.m'`

Example: `'../myfile.m'`

filter — File type filter specification

`All MATLAB Files` (default) | character vector | cell array of character vectors | string array

File type filter specification, specified as a character vector, a cell array of character vectors, or a string array. MATLAB appends **All Files** to the list of file types presented in the dialog box. The `filter` value can include the wildcard character (*).

Example: `*.m`

Example: `'MATLAB Files (*.m,*.mlx,*.fig,*.mat,*.slx,*.mdl)'`

title — Dialog box title

`Select File to Write` (default) | character vector | string scalar

Dialog box title, specified as a character vector or string scalar. To filter using the default file filter, but specify a custom title, use empty quotes for the filter value. For example:

```
uiputfile(' ','Select File')
```

Example: `'File Selector'`

Output Arguments

`file` — User-specified file name

character vector | string scalar

User-specified file name, returned as a character vector or string scalar. The user can specify the file name by selecting it in the dialog box, typing a file name in the **File name** field, or accepting the default file name (if you provide one). If the user cancels the dialog box, then MATLAB returns `file` as `0`.

- If the user specifies a valid file name and clicks **Save**, then MATLAB returns that file name in `file`.
- If the user types a file name in the dialog box **File name** field that includes the asterisk character (`*`) or the question mark character (`?`), then MATLAB does not respond to clicking the **Save** button. The dialog box remains open until the user cancels it or removes the wildcard or question mark characters. This restriction applies to all platforms, even on platforms that permit these characters in file names.
- If the user specifies an existing file name, then a warning dialog box opens stating that the file already exists and provides an option to replace the file.
  - If the user clicks **Yes** in the warning dialog box, then MATLAB replaces the existing file and returns the file name.
  - If the user clicks **No** in the warning dialog box, then control returns to the `uiputfile` dialog box, enabling the user to specify a different file name.
- If the user cancels the dialog box, then MATLAB returns `0` to `file`.

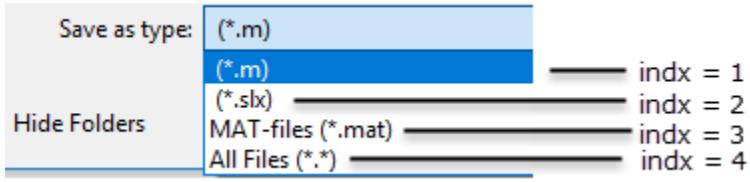
`path` — Path to the user-specified file name

character vector | 0

Path to the user-specified file name, returned as a character vector or `0`. If the user cancels the dialog box, then MATLABreturns `path` as `0`.

`indx` — Save as type index
integer

Save as type index, returned as an integer. As shown in this figure, the index corresponds to the **Save as type** row selection. Indexing starts at 1.



If the user clicks the **Cancel** button or the dialog box close button (X), or if the file does not exist, then MATLAB returns `indx` as `0`.

More About

Modal Dialog Box

A modal dialog box prevents a user from interacting with other MATLAB windows before responding to the modal dialog box.

Tips

- To write data to a user-specified file and location using MATLAB and MATLAB toolbox functions, use the path and file name that uiputfile returns. For example:
  - `fprintf` for writing data to a text file.
  - `imwrite` for writing an image to a graphics file.
  - `xlswrite` for writing a matrix to a Microsoft Excel spreadsheet. For example, this code creates a matrix, `A`, creates a dialog box to get a file name from the user, builds a full file name from the returned values, and then writes the matrix to the user-specified Excel file.

    ```
    A = [12.7 5.02 -98 63.9 0 -.2 56];
    [file,path] = uiputfile('*.xlsx');
    ```

```
filename = fullfile(path,file);
xlswrite(filename,A);
```

See Also

save | uigetdir | uigetfile | uisave

Introduced before R2006a

uiresume

Resume execution of blocked program

Syntax

```
uiresume(h)
```

Description

`uiresume(h)` resumes the program execution that `uiwait` suspended.

Examples

This example code creates a window containing a push button. The `uiwait` function blocks MATLAB execution until the user clicks the push button.

```
f = figure;
h = uicontrol('Position',[20 20 200 40],'String','Continue',...
              'Callback','uiresume(gcbf)');
disp('This will print immediately');
uiwait(gcf);
disp('This will print after you click Continue');
close(f);
```

The `gcbf` function returns the current figure.

Tips

The `uiwait` and `uiresume` functions block and resume MATLAB program execution. When creating a dialog box, you should have a uicontrol component with a callback that calls `uiresume` or a callback that destroys the dialog box. These are the only methods that resume program execution after the `uiwait` function blocks execution.

When used in conjunction with a modal dialog box, `uiresume` can resume the execution of the program that `uiwait` suspended while presenting a dialog box.

See Also

`dialog` | `figure` | `uicontrol` | `uifigure` | `uimenu` | `uiwait` | `waitfor`

Topics

"Modal Dialog Box in GUIDE"

Introduced before R2006a

uisave

Open dialog box for saving variables to MAT-file

Syntax

```
uisave
uisave(vars)
uisave(vars,file)
```

Description

`uisave` opens a modal on page 1-16280 Save Workspace Variables dialog box. If the user clicks **Save**, then MATLAB saves all variables from the user's workspace to the file displaying in the dialog box **File name** field.

If the named file already exists in the folder displayed at the top of the dialog box, then a confirmation dialog box opens and gives the user an opportunity to cancel the operation or overwrite the existing file.

`uisave(vars)` specifies which variables from the user's workspace to save.

`uisave(vars,file)` specifies the file name that appears in the **File name** field of the Save Workspace Variables dialog box when it opens, instead of the default, `matlab.mat`.

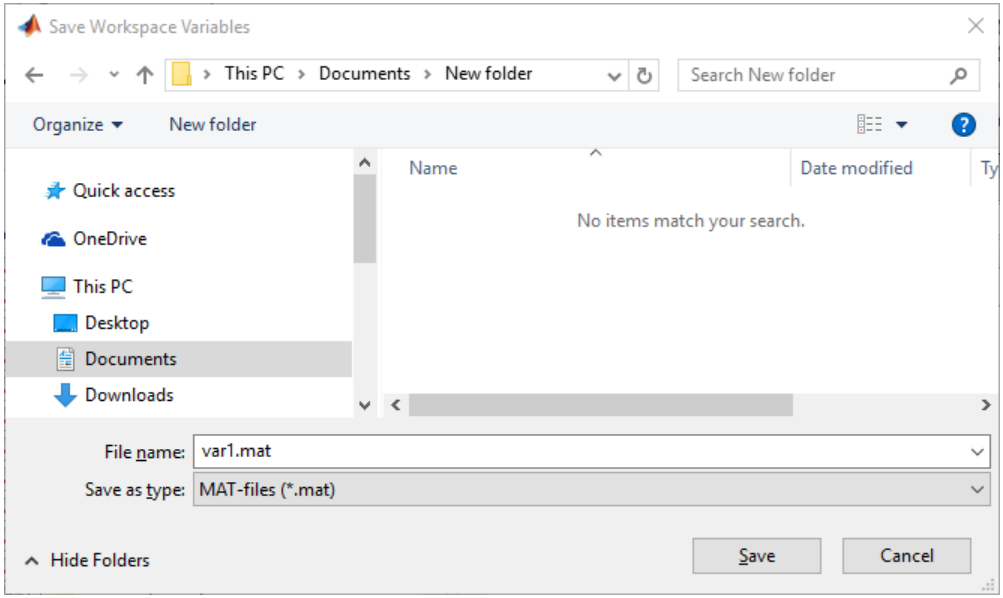
Examples

Customize Save Workspace Variables Dialog Box

Create three workspace variables, `d`, `w`, and `y`. Then, open a Save Workspace Variables dialog box with a default file, `var1.mat`, displaying in the **File name** field.

```
d = 'Sunday';
w = 31;
```

```
y = 2017;
uisave({'d','w'},'var1')
```



Click **Save** to save the workspace variables `d` and `w` to `var1.mat` in the folder that is displayed at the top of the dialog box.

Input Arguments

`vars` — Set of workspace variables

all workspace variables (default) | character vector | cell array of character vectors | string array

Set of workspace variables to save in a MAT-file, specified as a character vector, cell array of character vectors, or a string array. To specify more than one variable, use a cell array of character vectors.

Example: `'y'`

Example: `{x','y'}`

`file` — file name

'matlab.mat' (default) | character vector | string scalar

File name that displays in the **File name** field when the Save Workspace Variables dialog box opens. You can omit a file extension or specify the file extension as `.mat`.

Example: `'data1'`

Example: `'data1.mat'`

Example: `'v'`

More About

Modal Dialog Box

A modal dialog box prevents the user from interacting with other MATLAB windows before responding.

See Also

`uiopen`

Topics

"Save and Load Workspace Variables"

Introduced before R2006a

uisetcolor

Open color picker

Syntax

```
c = uisetcolor
c = uisetcolor(RGB)
c = uisetcolor(obj)
```

```
c = uisetcolor( ___ ,title)
```

Description

`c = uisetcolor` displays a modal on page 1-16286 color picker and returns the selected color as an RGB triplet. The RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the selected color. The intensities are in the range [0,1] and are of type `double`.

`c = uisetcolor(RGB)` specifies the default color selection as an RGB triplet.

`c = uisetcolor(obj)` sets the default color selection to be the color of an object (such as a `Figure`). If you select a different color in the color picker, the color of the object changes to the new color after you click **OK**. The object must have a property that controls some aspect of color. For example, some objects have a `Color` or `BackgroundColor` property.

`c = uisetcolor( ___ ,title)` specifies a custom title for the dialog box. Specify the title as the last argument for any of the above syntaxes.

Examples

Specify the Default Color

Open the color picker with light blue as the default color.

```
c = uisetcolor([0.6 0.8 1])
```

Clicking the **OK** button without selecting a different color returns the default color.

```
c =

    0.6000    0.8000    1.0000
```

Specify a Custom Title for the Dialog Box

Open the color picker with yellow as the default color and set the title to `'Select a Color'`.

```
c = uisetcolor([1 1 0],'Select a color')
```

Clicking the **OK** button without selecting a different color returns the default color.

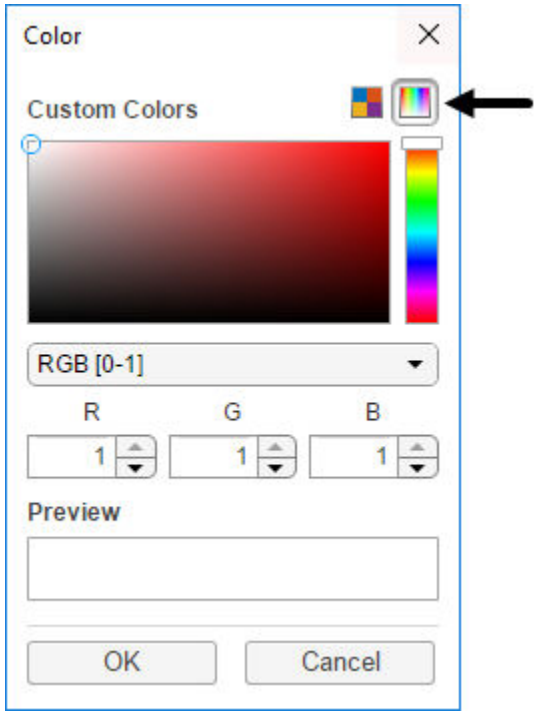
```
c =

     1     1     0
```

Select a Color from a Gradient

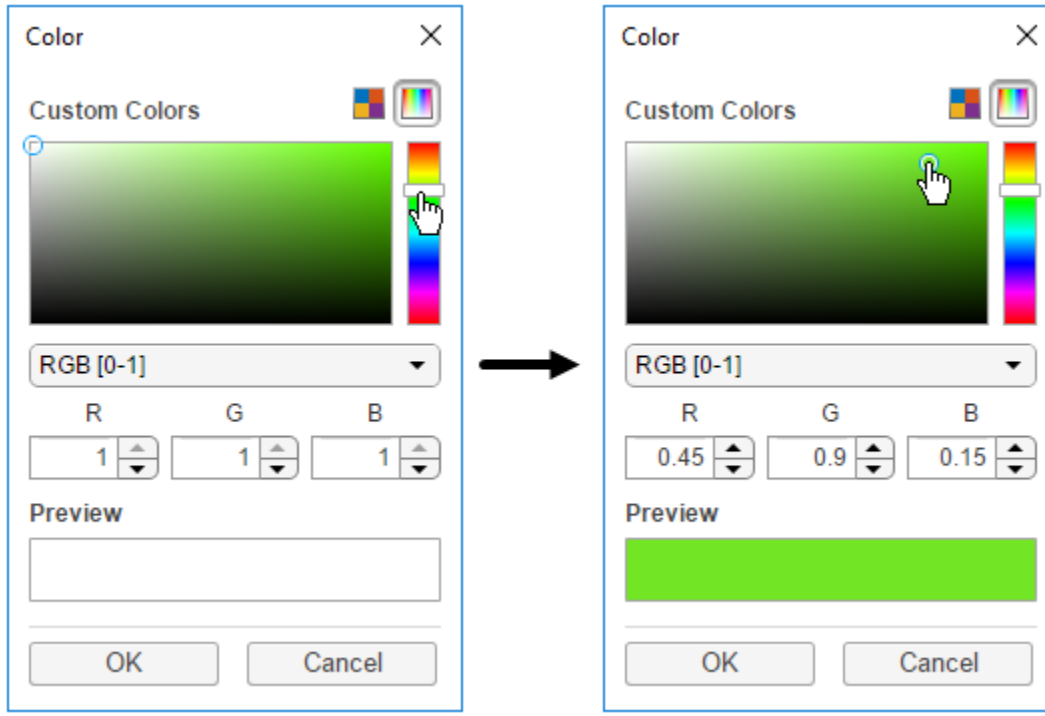
Starting in R2018b, the color picker provides a tab for selecting custom colors from a gradient.

Open the color picker, and click the **Custom Colors** tab in the upper-right corner.

```
c = uisetcolor
```



Move the vertical slider to display the desired region of color space. Then click the color gradient to select a color. When you click **OK**, `uisetcolor` returns your selection as an RGB triplet.




```
c =

    0.4471    0.9020    0.1451
```

Input Arguments

RGB — Default color
`[1 1 1]` (default) | RGB triplet

Default color, specified as an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the selected color. The intensities must be in the range [0,1].

Example: `c = uisetcolor([1 0 0])` specifies red as the default color selection.

Example: `c = uisetcolor([0.5 0.5 0.5])` specifies gray as the default color selection.

Data Types: `single` | `double`

`obj` — Source object for default color
graphics object

Source object for default color, specified as a graphics object. The object must have a property that controls some aspect of color. For example, some objects have a `Color` or `BackgroundColor` property.

Example: `c = uisetcolor(figure)` creates a `Figure` and sets the default color selection to be the same color as the figure.

`title` — Title of dialog box
`'Color'` (default) | character vector | string scalar

Title of dialog box, specified as a character vector or string scalar.

Example: `c = uisetcolor('Choose a Color')` specifies `'Choose a Color'` as the dialog box title.

Data Types: `char`

More About

Modal Window

A window that blocks interaction with other windows until the user closes it.

See Also

`dialog` | `questdlg` | `uigetdir` | `uigetfile` | `uiputfile`

Introduced before R2006a

uisetfont

Open font selection dialog box

Syntax

```
uisetfont
uisetfont(h)
uisetfont(optsin)
uisetfont( ___ ,title)
optsout = uisetfont( ___ )
```

Description

`uisetfont` opens a modal on page 1-16292 Font dialog box with default font name and font style values selected. If the user clicks **OK**, then the selected font properties are returned to the MATLAB Command Window. If the user clicks **Cancel**, or the close button (X) in the dialog box title bar, then a value of `0` is returned to the Command Window.

`uisetfont(h)` specifies the object on which the Font dialog box will operate. The values displaying in the dialog box when it opens are the current settings for that object.

The value of `h` can be a `text`, `axes`, or `uicontrol` object.

`uisetfont(optsin)` initializes the Font dialog box selections with the values defined for the specified `optsin` structure.

`uisetfont( ___ ,title)` specifies a custom title for the Font dialog box. This syntax can include any of the input arguments in the previous syntaxes.

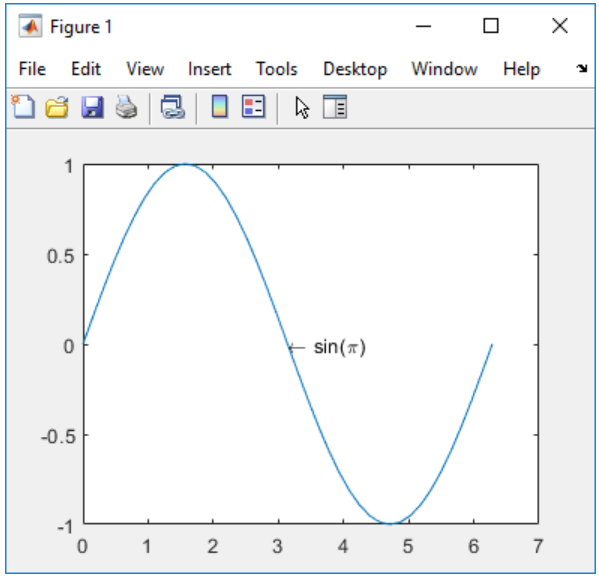
`optsout = uisetfont( ___ )` returns the values selected in the dialog box to the `optsout` structure when the user clicks **OK**. If the user clicks **Cancel** or an error occurs, then `optsout` is set to `0`. This syntax can specify no input arguments or any of the input arguments in the previous syntaxes.

Examples

Set Font Characteristics for Plot Annotation

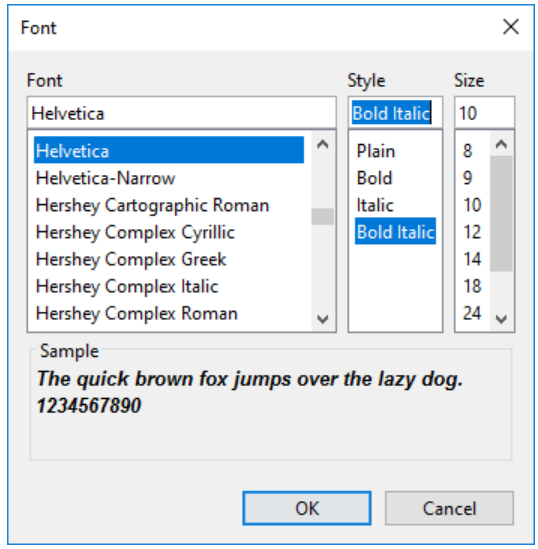
Create a plot containing a text annotation.

```
f = figure('Position',[200 200 392 294]);
x = 0:pi/20:2*pi;
y = sin(x);
plot(x,y);
t = text(pi,0,'\leftarrow sin(\pi)');
```

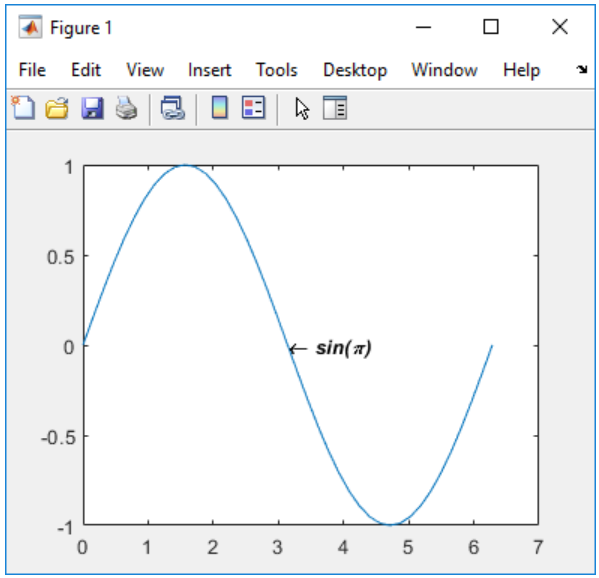


Open the **Font** dialog box, and select the **Bold Italic** style. (The font names displayed in the dialog box are system-dependent.)

```
optsout = uisetfont(t);
```



Click **OK**. The font style changes in the plot annotation.



Input Arguments

h — Target object
axes object | text object | uicontrol object

Target object for font changes, specified as an `axes`, `text`, or `uicontrol` object.

`optsin` — Font
structure

Font values, specified as a structure of font properties.

The following table lists the structure fields, the default values, and the dialog box controls with which a user can change the values.

Structure Fields

Structure Field	Default Value	Dialog Box Control
`FontName`	system-dependent	**Font** list.
`FontWeight`	`'normal'`	**Style** list.
`FontAngle`	`'normal'`	**Style** list.
`FontUnits`	`'points'`	Not provided because the value of `FontUnits` is always `'points'` for the supported objects.
`FontSize`	system-dependent	**Size** list

A structure field value of `'normal'` corresponds to `Plain` in the dialog box **Style** list.

The user can select a **Size** value from the list or type in an unlisted value. If the user types a floating-point number, MATLAB rounds it to the closest integer value before storing it.

Example: `optsin.FontWeight = 'bold';`

`title` — Dialog box title
`'Font'` (default) | character vector | string scalar

Dialog box title, specified as a character vector or a string scalar.

Example: `'Font Selection'`

Output Arguments

`optsout` — Font values
structure

Font values, returned as a structure of font property values. The structure fields are:

- `FontName`
- `FontWeight`
- `FontAngle`
- `FontUnits`

- `FontSize`

If the user typed a floating-point number for the dialog box field that corresponds to the `FontSize` value, then MATLAB rounds it to the closest integer value before returning it.

More About

Modal Dialog Box

A modal dialog box prevents a user from interacting with other MATLAB windows before responding to the modal dialog box.

See Also

`axes` | `text` | `uicontrol`

Introduced before R2006a

uisetpref

Manage preferences used in `uigetpref`

Syntax

```
uisetpref('clearall')
```

Description

`uisetpref('clearall')` resets the value of all preferences registered through `uigetpref` to `'ask'`. This causes the dialog box to display when you call `uigetpref`.

Note Use `setpref` to set the value of a particular preference to `'ask'`.

See Also

`setpref` | `uigetpref`

Introduced before R2006a

uistack

Reorder visual stacking of UI components

Note

Use this function only with GUIDE, or with apps created using the `figure` function.

Syntax

```
uistack(comp)
uistack(comp,moveto)
uistack(comp,moveto,step)
```

Description

`uistack(comp)` shifts the specified component up one level within the front-to-back visual stacking order of UI components. If `comp` is specified as a vector of UI components, each component in the vector gets shifted up by one.

`uistack(comp,moveto)` moves the UI component to another position in the stack. For example, `uistack(f,'top')` moves component `f` to the top of the current stack.

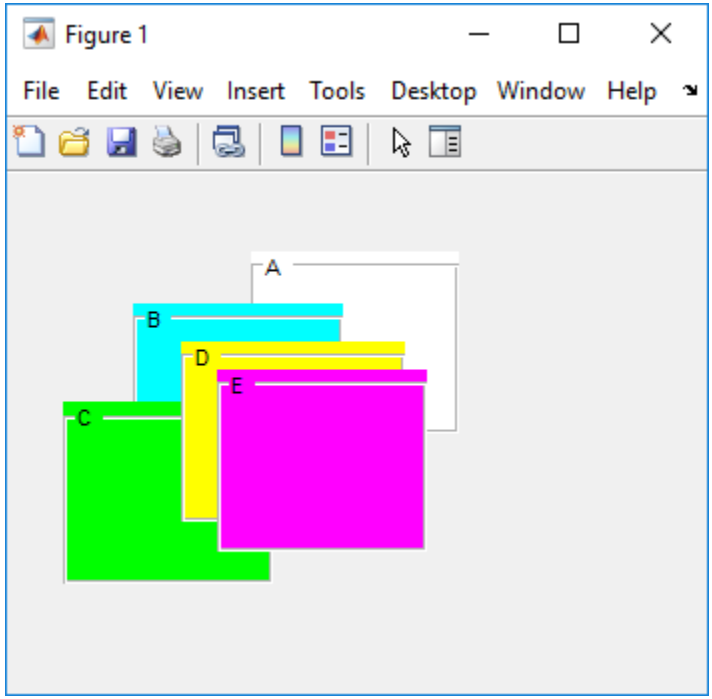
`uistack(comp,moveto,step)` specifies the number of levels to move the UI component up or down. For example, `uistack(c,'up',2)` moves `c` up two levels in the current stack.

Examples

Reorder Vector of Panels

Create five overlapping panels with different titles and background colors.

```
f = figure;

a = uipanel(f,'Title','A','BackgroundColor','white');
b = uipanel(f,'Title','B','BackgroundColor','cyan');
c = uipanel(f,'Title','C','BackgroundColor','green');
d = uipanel(f,'Title','D','BackgroundColor','yellow');
e = uipanel(f,'Title','E','BackgroundColor','magenta');

a.Position = [0.35 0.50 0.30 0.35];
b.Position = [0.18 0.40 0.30 0.35];
c.Position = [0.08 0.21 0.30 0.35];
d.Position = [0.25 0.33 0.32 0.35];
e.Position = [0.30 0.27 0.30 0.35];
```



List the order of children in the figure.

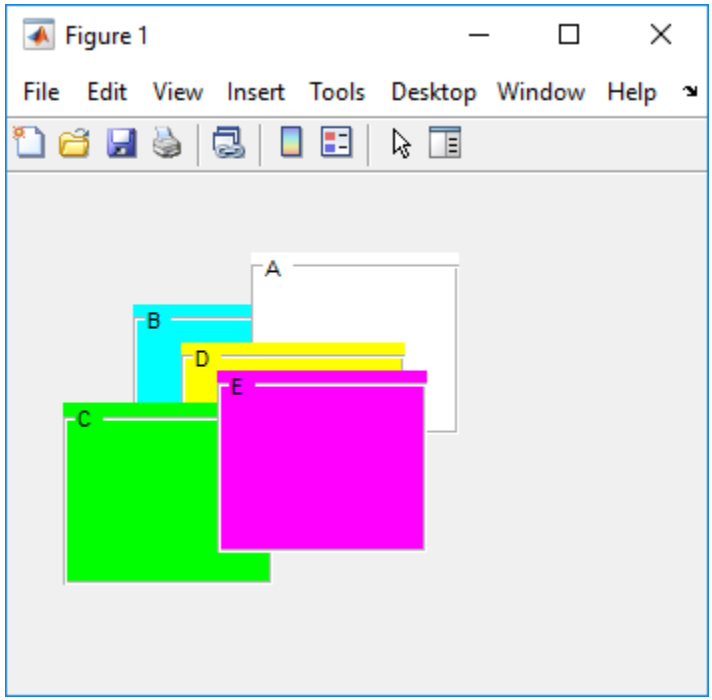
```
figChildren = f.Children
```

```
figChildren =
```

```
5×1 Panel array:

Panel    (E)
Panel    (D)
Panel    (C)
Panel    (B)
Panel    (A)
```

Reorder the panels by shifting a and c up one level in the stack, relative to their previous positions.

```
comp = [a c];
uistack(comp);
```



List the children again. Changing the stacking order of the panels also changes the order of the children in the figure.

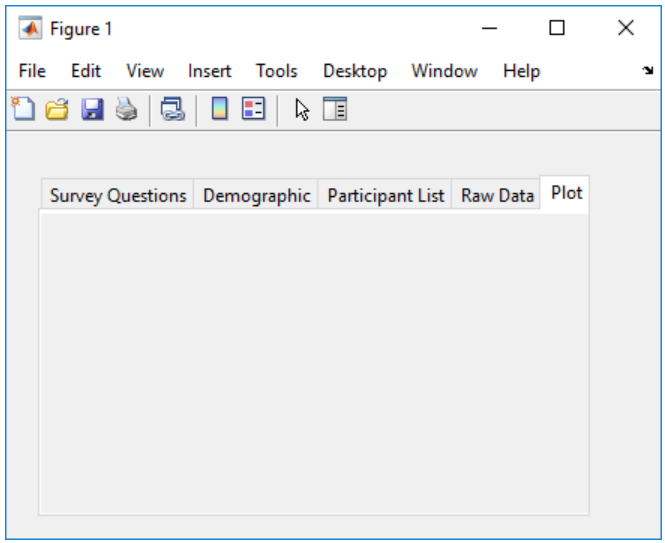
```
figChildren = f.Children
```

```
figChildren =

  5×1 Panel array:

  Panel    (E)
  Panel    (C)
  Panel    (D)
  Panel    (A)
  Panel    (B)
```

Reorder Tabs in Tab Group

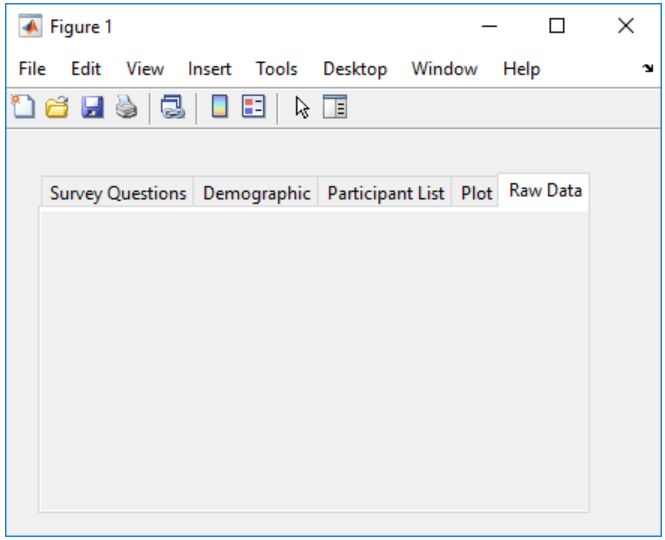
Create a figure that contains a tab group with five tabs.

```
f = figure;
tg = uitabgroup(f,'Position',[0.05 0.05 0.85 0.85]);
t1 = uitab('Title','Survey Questions');
t2 = uitab('Title','Demographic');
t3 = uitab('Title','Participant List');
t4 = uitab('Title','Raw Data');
t5 = uitab('Title','Plot');
```



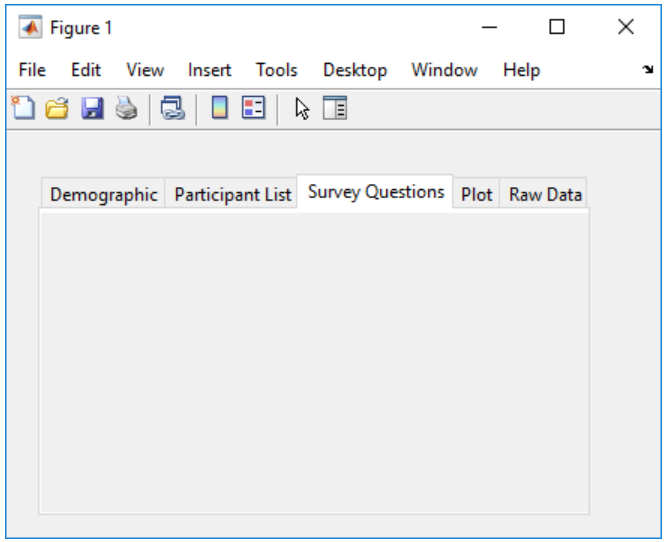
Move the **Raw Data** tab to the bottom of the stack.

```
uistack(t4,'bottom');
```



Move the **Survey Questions** tab down two levels.

```
uistack(t1,'down',2);
```



Input Arguments

`comp` — UI component
single object | vector of objects

UI component to reorder, specified as a single object or vector of objects, such as `Figure`, `Panel`, `ButtonGroup`, `UIControl`, `Axes`, or `Tab` objects. Use this argument to specify the UI components you wish to reorder within the current stack.

If `comp` is specified as a vector of UI components, each component in the vector must share the same parent and the vector must be a subset of the children of the parent container. For example, if a figure has six child UI components, the vector `comp` can have no more than five elements.

`moveto` — Location
`'up'` (default) | `'down'` | `'top'` | `'bottom'`

Location to move a UI component, specified as one of the values in the following table. Use this argument to specify the stack position you want a UI component to move to.

Move To Value	Description
`'up'`	Up `step` positions (one position by default).
`'down'`	Down `step` positions (one position by default).
`'top'`	To the top of the current stack.
`'bottom'`	To the bottom of the current stack.

`step` — Number of levels to shift
1 (default) | positive integer

Number of levels to shift a UI component up or down in a stack, specified as a positive integer.

If you specify a step number that is greater than the number of stack levels available to move, then the UI component will move to the top or bottom of the stack. For example, if you have five stack levels and you specify a component to move down six steps, that component will move to the bottom of the stack.

Compatibility Considerations

Stacking Order Might Display Differently in Older Apps

Behavior changed in R2014b

In previous releases, the order of components listed in the `Children` property matched the order in which they were created. However, this order did not necessarily match the visual stacking order of the components on the screen. To provide more consistent behavior, starting in R2014b, the order of components listed in the `Children` property matches the stacking order of child components on the screen. For more information about how the stacking behavior of UI components has changed and how to update your code, see “Why Are Some Components Missing or Partially Obscured?”

See Also

`figure` | `uibuttongroup` | `uipanel` | `uitab`

Topics

“Manage the Stacking Order of Grouped Components”

Introduced before R2006a

uistyle

Create style for table UI component

Syntax

```
s = uistyle
s = uistyle(Name,Value)
```

Description

`s = uistyle` creates an empty style for a table UI component and returns the `Style` object. Use `Style` objects to create color, font, and alignment styles for cells in table UI components. Use this syntax to create a style you want to add properties to later.

`s = uistyle(Name,Value)` specifies `Style` property values using one or more name-value pair arguments. For example, `'BackgroundColor','g'` sets the background color to green.

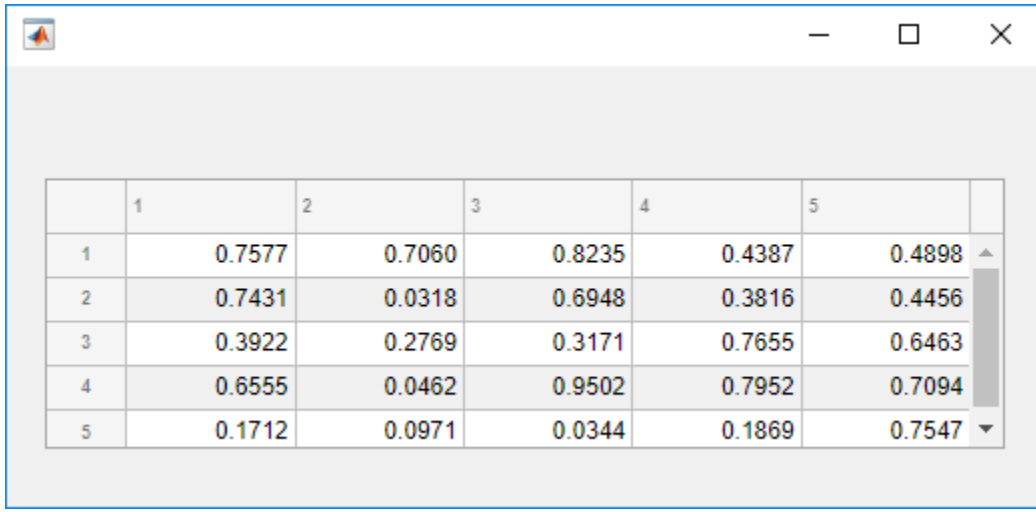
Examples

Change Appearance of Table Cells

Change the background color for a table column by creating a style and applying it to the table.

Create a figure with a table UI component in it and populate the table with numeric data.

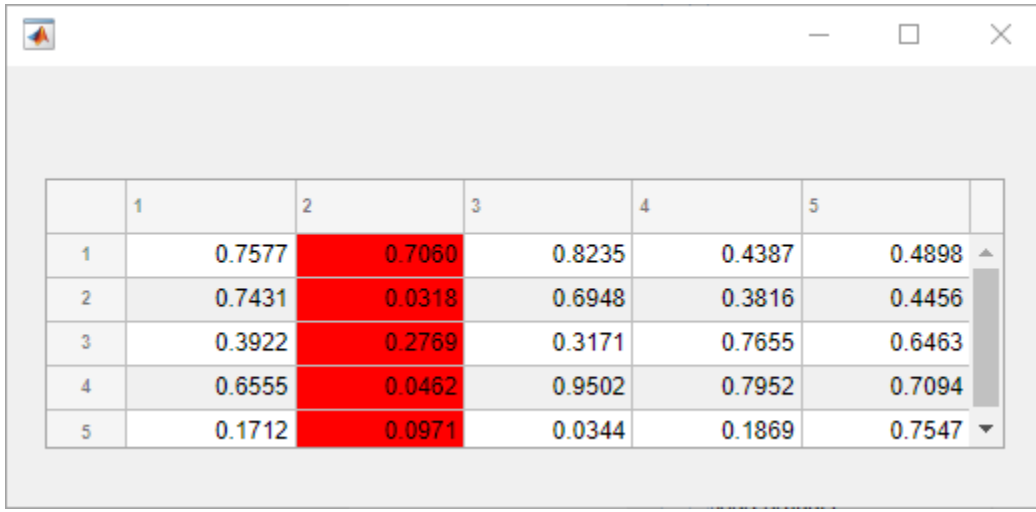
```
fig = uifigure;
fig.Position = [500 500 520 220];
uit = uitable(fig);
uit.Data = rand(5);
uit.Position = [20 30 480 135];
```



	1	2	3	4	5
1	0.7577	0.7060	0.8235	0.4387	0.4898
2	0.7431	0.0318	0.6948	0.3816	0.4456
3	0.3922	0.2769	0.3171	0.7655	0.6463
4	0.6555	0.0462	0.9502	0.7952	0.7094
5	0.1712	0.0971	0.0344	0.1869	0.7547

Then, create a style with a specific background color and add the style to the second column of the table using the `addStyle` function.

```
s = uistyle('BackgroundColor','red');
addStyle(uit,s,'column',2)
```



	1	2	3	4	5
1	0.7577	0.7060	0.8235	0.4387	0.4898
2	0.7431	0.0318	0.6948	0.3816	0.4456
3	0.3922	0.2769	0.3171	0.7655	0.6463
4	0.6555	0.0462	0.9502	0.7952	0.7094
5	0.1712	0.0971	0.0344	0.1869	0.7547

Emphasize Different Parts of Table Using Multiple Styles

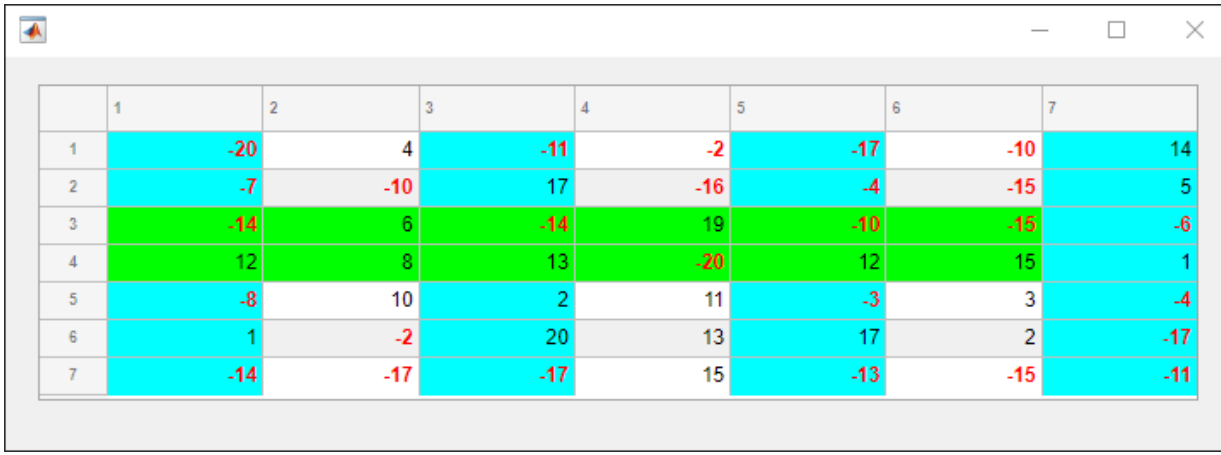
Create multiple styles and add them to different parts of a table UI component.

Create a figure with a table UI component in it and display numeric data in the table. Find the row and column subscripts for elements in the table with a value less than zero so you can style these cells later.

```
fig = uifigure;
fig.Position = [500 500 720 230];

uit = uitable(fig);
uit.Data = randi([-20,20],7);
uit.Position = [20 30 680 185];

[row,col] = find(uit.Data < 0);
```

Create two background color styles and one style that specifies font color and weight. Add a cyan background color to columns 1, 3, and 5. Emphasize the cells with negative values by making their font red and bold. Then, style rows 3 and 4 with a green background color. Finally, reuse the cyan background color style and add it to column 7. For cells where multiple styles of the same type are added, the style that is added last is the one that displays in the cell.

```
s1 = uistyle;
s1.BackgroundColor = 'cyan';
addStyle(uit,s1,'column',[1 3 5])

s2 = uistyle;
s2.FontColor = 'red';
s2.FontWeight = 'bold';
addStyle(uit,s2,'cell',[row,col])

s3 = uistyle;
s3.BackgroundColor = 'green';
addStyle(uit,s3,'row',[3 4])

addStyle(uit,s1,'column',7)
```



	1	2	3	4	5	6	7
1	-20	4	-11	-2	-17	-10	14
2	-7	-10	17	-16	-4	-15	5
3	-14	6	-14	19	-10	-15	-6
4	12	8	13	-20	12	15	1
5	-8	10	2	11	-3	3	-4
6	1	-2	20	13	17	2	-17
7	-14	-17	-17	15	-13	-15	-11

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `s = uistyle('BackgroundColor','blue')`

`BackgroundColor` — Background color
`[]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to

F. The values are not case sensitive. Therefore, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`FontColor` — Font color

`[]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`FontWeight` — Font weight
`''` (default) | `'normal'` | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. For fonts that do not, specifying `'bold'` results in the normal font weight.

`FontAngle` — Font angle
`''` (default) | `'normal'` | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Not all fonts have an italic font angle. For fonts that do not, specifying `'italic'` results in the normal font angle.

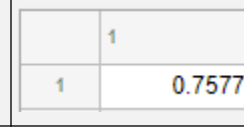
`FontName` — Font name
`''` (default) | system supported font name

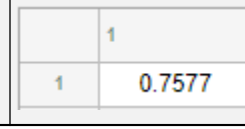
Font name, specified as a system-supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

`HorizontalAlignment` — Horizontal alignment of cell data
`''` (default) | `'left'` | `'right'` | `'center'`

Horizontal alignment of cell text, specified as one of the values in the table.

Value	Example
`'left'`	
`'right'`	

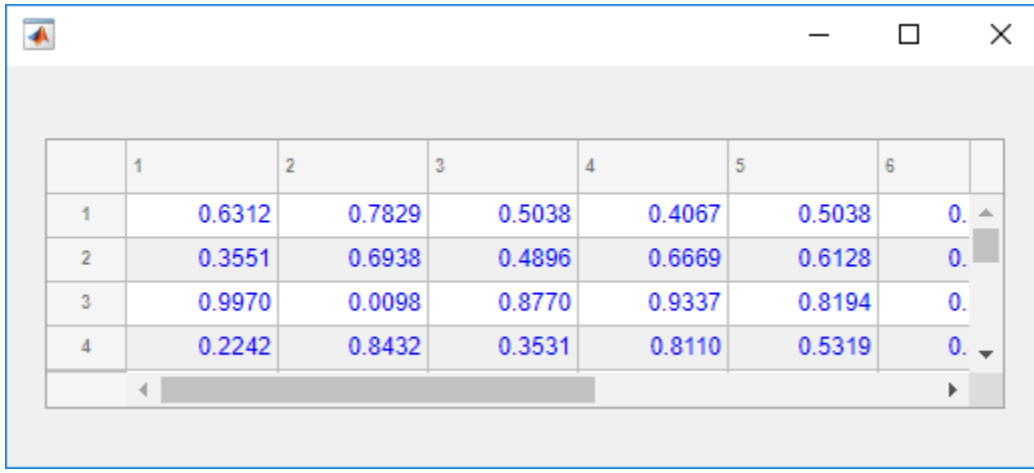
Value	Example
`'center'`	

If you add a `Style` object to the table UI component that specifies horizontal alignment, it takes precedence over the justifications associated with cell format values that you specify for the `ColumnFormat` property on the `Table` object. The `ColumnFormat` property still converts values as it normally would.

Algorithms

`Style` objects that you add to the table UI component take visual precedence over properties you set on the `Table` object, no matter the order in which you created them. For example, in this code the blue font color displays in the table even though red foreground color was set on the `Table` object last.

```
uit = uitable(uifigure,'Data',rand(100,10));

s = uistyle;
s.FontColor = 'blue';
addStyle(uit,s);

uit.ForegroundColor = 'red';
```



See Also

Functions
`addStyle` | `removeStyle` | `uitable`

Properties
Style Properties | Table Properties

Introduced in R2019b

uitab

Create tabbed panel

Syntax

```
t = uitab
t = uitab(Name,Value)
t = uitab(parent)
t = uitab(parent,Name,Value)
```

Description

`t = uitab` creates a tab inside a tab group and returns the `Tab` object. If there is no tab group available, MATLAB calls the `figure` function to create a figure. Then it creates a tab group in that figure, and places the tab inside the tab group.

`t = uitab(Name,Value)` specifies tab property values using one or more name-value pair arguments.

`t = uitab(parent)` creates the tab in the specified parent container. The parent container can be a tab group in a figure created with either the `figure` or `uifigure` function. Property values for `uitab` vary slightly depending on whether the app is created with the `figure` or `uifigure` function. For more information, see “Name-Value Pair Arguments” on page 1-16314.

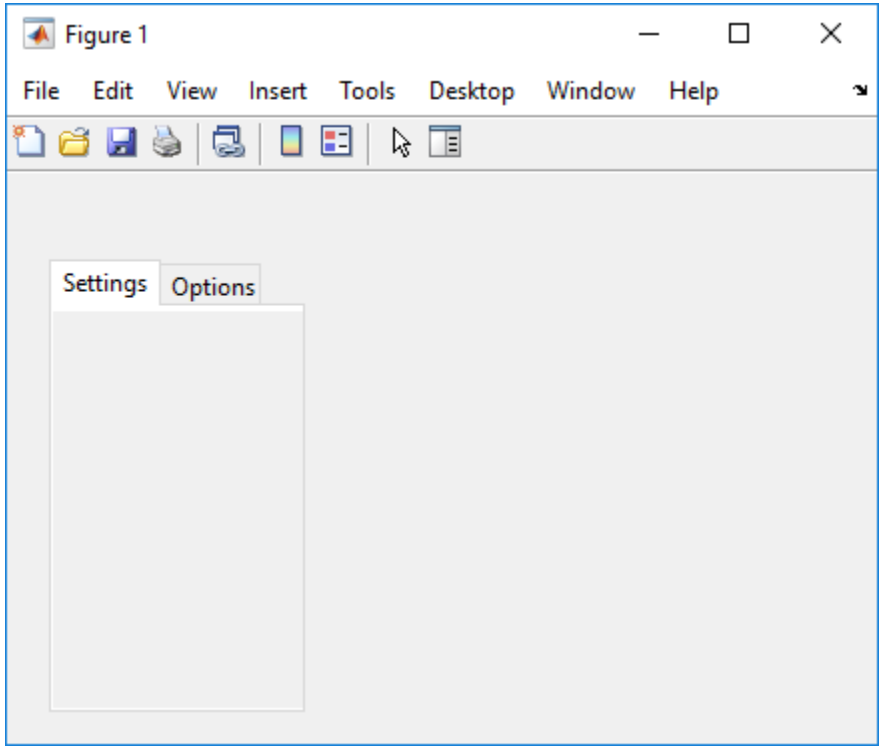
`t = uitab(parent,Name,Value)` specifies the parent container and one or more properties values.

Examples

Two Tabs in a Figure

Create a figure containing a tab group and two tabs.

```
f = figure;
tabgp = uitabgroup(f,'Position',[.05 .05 .3 .8]);
tab1 = uitab(tabgp,'Title','Settings');
tab2 = uitab(tabgp,'Title','Options');
```



Get the title of the first tab.

```
tab1.Title
```

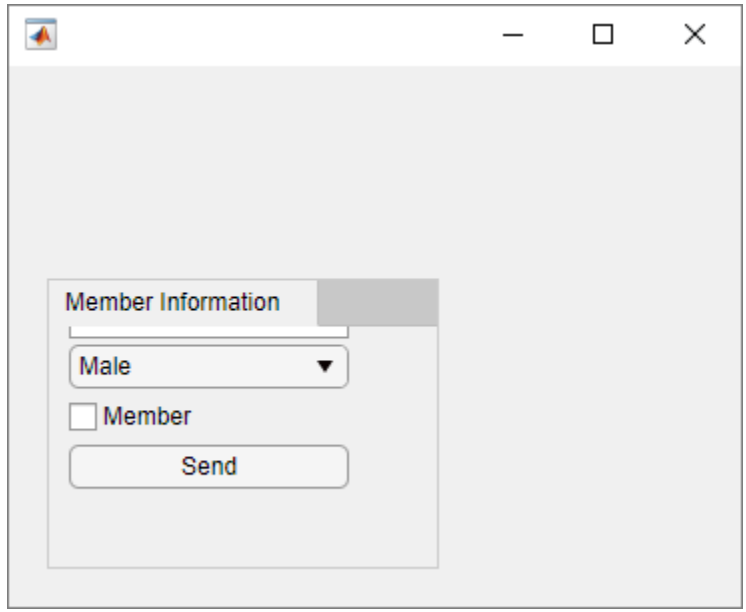
```
ans =

Settings
```

Scrollable Tab

The `Scrollable` property enables scrolling within a tab that has components outside its borders. Scrolling is only possible when the parent tab group is in a figure created with the `uifigure` function. App Designer uses this type of figure for creating apps.

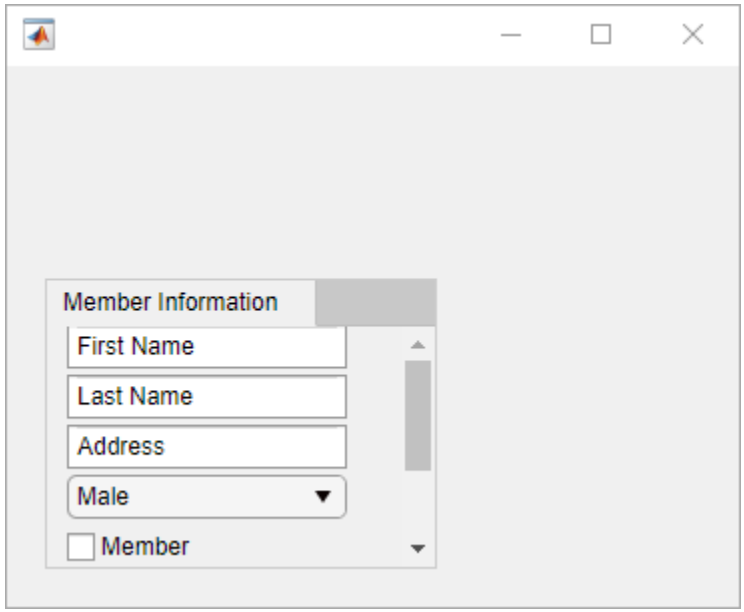
Create a tab group containing one tab. Add six UI components to the tab, with the first three lying outside the upper border of the tab.

```
f = uifigure;
tg = uitabgroup(f,'Position',[20 20 196 145]);
t = uitab(tg,'Title','Member Information');
ef1 = uieditfield(t,'text','Position',[11 165 140 22],'Value','First Name');
ef2 = uieditfield(t,'text','Position',[11 140 140 22],'Value','Last Name');
ef3 = uieditfield(t,'text','Position',[11 115 140 22],'Value','Address');
dd = uidropdown(t,'Position',[11 90 140 22],'Items',{'Male','Female'});
cb = uicheckbox(t,'Position',[11 65 140 22],'Text','Member');
b = uibutton(t,'Position',[11 40 140 22],'Text','Send');
```



Enable scrolling by setting the `Scrollable` property of the tab to `'on'`. By default, the scroll box displays at the top.

```
t.Scrollable = 'on';
```



Input Arguments

parent — Parent container
TabGroup object

Parent container, specified as a tab group. The tab group can be a child of a figure created with either the `figure` or `uifigure` function.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments as `Name1,Value1,...,NameN,ValueN`.

Example: `'Title','Options'` specifies that the tab title is `Options`.

- For a list of properties available for GUIDE, or for creating apps with the `figure` function, see Uitab for GUIDE.

- For a list of properties available for App Designer, or for creating apps with the `uifigure` function, see Tab for App Designer.

See Also

Functions
`uitabgroup`

Properties
Tab

Introduced in R2014b

Tab Properties

Control appearance and behavior of tab

Note The properties listed here are valid for tabs in GUIDE or in apps created with the `figure` function. If you are using App Designer or the `uifigure` function, see Tab Properties instead. For more information, see “GUIDE Migration Strategies”.

Description

Tabs are containers for grouping for grouping UI components together with a tabbed label. Use the `uitab` function to create a tab inside a tab group. Properties control the appearance and behavior of a tab. Use dot notation to refer to a particular object and property:

```
f = figure;
tg = uitabgroup(f);
t = uitab;
t.Title = 'Data';
```

Properties

Title and Color

`Title` — Title

character vector | string scalar | categorical array

Title, specified as a character vector, string scalar, or categorical array. If you specify this property as a categorical array, MATLAB displays only the first element in the array.

MATLAB does not interpret a vertical slash (`'|'`) character as a line break, it displays as a vertical slash in the title.

If you want to specify a Unicode character, pass the Unicode decimal code to the `char` function. For example, `['Multiples of ' char(960)]` displays as `Multiples of` π.

`ForegroundColor` — Title color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Title color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: [0 0 1]

Example: 'b'

Example: 'blue'

BackgroundColor — Background color

[.94 .94 .94] (default) | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Interactivity

`Tooltip` — Tooltip

character vector | string scalar | categorical array

Tooltip, specified as a character vector, string scalar, or categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip does not display when the component is disabled. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

To create multiple lines of text, use the `sprintf` function to insert newline characters (`'\n'`) in your text. For example:

```
txt = sprintf('Line 1\nLine 2');
```

Then set the `Tooltip` property to the value returned by `sprintf`.

UIContextMenu — Context menu
empty GraphicsPlaceholder array (default) | uicontextmenu object

Context menu, specified as a ContextMenu object. Use this property to display a context menu when the user right-clicks on a UI component. Create the context menu using the uicontextmenu function.

TooltipString — Tooltip (not recommended)
character vector | string scalar | categorical array

Tooltip, specified as a character vector, string scalar, or categorical array. The tooltip displays when the user hovers the mouse pointer over the component in the app. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Note The TooltipString property is not recommended starting in R2018b. Use the Tooltip property instead.

Location and Size

Position — Location and size
[left bottom width height]

This property is read-only.

Location and size, returned as a vector of the form [left bottom width height]. This table describes each element in the vector.

Value	Description
left	Distance from the inner left edge of the parent tab group to the inner left edge of the tab.
bottom	Distance from the inner bottom edge of the parent tab group to the inner bottom edge of the tab.
width	Distance between the right and left inner edges of the tab.
height	Distance between the top and bottom inner edges of the tab.

All measurements exclude the tab label and are in units specified by the Units property.

Note These are some important points to consider when using the `Position` property:

- `Position` values are affected by the length of the tab title and the parent tab group's `TabLocation` property.
- `Position` values are relative to the parent tab group's drawable area. The drawable area of a tab group is the area inside its borders.

`InnerPosition` — Location and size

`[left bottom width height]`

This property is read-only.

Location and size, returned as a four-element vector of the form `[left bottom width height]`. All measurements are in units specified by the `Units` property.

This property value is identical to the `Position` and `OuterPosition` property values.

`OuterPosition` — Location and size

`[left bottom width height]`

This property is read-only.

Location and size, returned as a four-element vector of the form `[left bottom width height]`. All measurements are in units specified by the `Units` property.

This property value is identical to the `Position` and `InnerPosition` property values.

`Units` — Units of measurement

`'normalized'` (default) | `'pixels'` | `'inches'` | `'centimeters'` | `'points'` | `'characters'`

Units of measurement, specified one of the values from this table.

Units Value	Description
`'normalized'`	These units are normalized with respect to the parent container. The lower-left corner of the container maps to `(0,0)` and the upper-right corner maps to `(1,1)`.

Units Value	Description
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'points'`	Points. One point equals 1/72nd of an inch.
`'characters'`	These units are based on the default uicontrol font of the graphics root object: • Character width = width of the letter `x`. • Character height = distance between the baselines of two lines of text. To access the default uicontrol font, use `get(groot,'defaultuicontrolFontName')` or `set(groot,'defaultuicontrolFontName')`.

MATLAB uses these units to display the location and size values in the `Position` property.

MATLAB measures all units from the lower left corner of the parent object.

If you change the value of the `Units` property, it is good practice to return it to its default value after completing your computation to avoid affecting other functions that assume the `Units` property is set to the default value.

Callbacks

`SizeChangedFcn` — Size change callback function

`''` (default) | function handle | cell array | character vector

Size change callback function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

Define this callback to customize the app layout when the size of this container changes (e.g., when the user resizes the window).

The `SizeChangedFcn` callback executes when the parent tab group container:

- Becomes visible for the first time.
- Is visible while its size changes.
- Becomes visible for the first time after its size changes. This situation occurs when the size changes while the container is invisible, and then it becomes visible later.

Other important points to consider when defining a `SizeChangedFcn` callback:

- Consider delaying the display of this container until after all the variables that the `SizeChangedFcn` uses are defined. This practice can prevent the `SizeChangedFcn` callback from returning an error. To achieve this delay, you delay the display of the parent tab group container, by setting its `Visible` property to `'off'`. Then, set the `Visible` property to `'on'` after you define the variables that your `SizeChangedFcn` callback uses.
- If your app contains nested containers, they resize from the inside out.
- To access the container that is resizing from within the `SizeChangedFcn`, refer to the source object (the first input argument in the callback) or use the `gcbo` function.

Tip As an easy alternative to specifying a `SizeChangedFcn` callback, you can set the `Units` property of all the objects you put inside a container to `'normalized'`. Doing so makes those components scale proportionally with the container.

See “Lay Out a UI Programmatically” for more information about managing layouts with `SizeChangedFcn` callbacks.

`ButtonDownFcn` — Button-press callback function

`''` (default) | function handle | cell array | character vector

Button-press callback function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

The `ButtonDownFcn` callback is a function that executes when the user clicks a mouse button within the container.

`CreateFcn` — Component creation function

`''` (default) | function handle | cell array | character vector

Component creation function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

This property specifies a callback function to execute when MATLAB creates the component. MATLAB initializes all component property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Use the `gcbo` function in your `CreateFcn` code to get the component object that is being created.

Setting the `CreateFcn` property on an existing component object has no effect.

`DeleteFcn` — Component deletion function

`''` (default) | function handle | cell array | character vector

Component deletion function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

The `DeleteFcn` property specifies a callback function to execute when MATLAB deletes the component (for example, when the user closes the window). MATLAB executes the `DeleteFcn` callback before destroying the properties of the component object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

Use the `gcbo` function in your `DeleteFcn` code to get the component object that is being deleted.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed:

- A value of `'on'` allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- A value of `'off'` blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn`, or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.
- MATLAB does not save the state of properties or the display when an interruption occurs. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

See “Interrupt Callback Execution” for an example that shows how the `Interruptible` and `BusyAction` properties affect the behavior of a program.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` (default) or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

The `BusyAction` property of the source of the interrupting callback determines how MATLAB handles its execution. The `BusyAction` property has these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Do not execute the interrupting callback.

Whenever MATLAB invokes a callback, that callback always attempts to interrupt an executing callback. The `Interruptible` property of the object whose callback is running determines if interruption is allowed. If `Interruptible` is set to:

- `on` — Interruption occurs at the next point where MATLAB processes the queue. This is the default.
- `off` — The `BusyAction` property (of the object owning the interrupting callback) determines if MATLAB enqueues or ignores the interrupting callback.

See “Interrupt Callback Execution” for an example that shows how the `BusyAction` and `Interruptible` properties affect the behavior of a program.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent object

`TabGroup` object

Parent object, specified as a `TabGroup` object. You can move a tab to a different tab group by setting this property to the target `TabGroup` object.

`Children` — Tab children

empty `GraphicsPlaceholder` array (default) | 1-D array of component objects

`Tab` children, returned as an empty `GraphicsPlaceholder` or a 1-D array of UI component objects. The children of tabs can be `Axes`, `Panel`, `ButtonGroup`, and `UIControl` objects.

You cannot add or remove children using the `Children` property. Use this property to view the list of children or to reorder the children. The order of the children reflects the front-to-back order (stacking order) of the components on the screen. MATLAB might not allow you to change the order of certain objects. For example, `UIControl` and `Legend` objects are always in front of `Axes` objects.

To add a child to this list, set the `Parent` property of the child component to be the `Tab` object.

Objects with the `HandleVisibility` property set to `'off'` do not list in the `Children` property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object handle in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. The `HandleVisibility` property also controls the visibility of the object’s handle in the parent figure's `CurrentObject` property. Handles are still valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object handle is always visible.
`'callback'`	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.

HandleVisibility Value	Description
`'off'`	The object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.

Set the graphics root `ShowHiddenHandles` property to `'on'` to make all handles visible, regardless of their `HandleVisibility` value. This setting has no effect on their `HandleVisibility` values.

Identifiers

Type — Type of graphics object
`'uitab'`

This property is read-only.

Type of graphics object, returned as `'uitab'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any array. Specifying `UserData` can be useful for sharing data within apps. See “Share Data Among Callbacks” for more information.

See Also

`guide` | `uitab` | `uitabgroup`

Topics

“Access Property Values”

"Default Property Values"

Introduced in R2014b

uitabgroup

Create container for tabbed panels

Syntax

```
tg = uitabgroup
tg = uitabgroup(Name,Value)
tg = uitabgroup(parent)
tg = uitabgroup(parent,Name,Value)
```

Description

`tg = uitabgroup` creates a tab group in the current figure and returns the `TabGroup` object. If there is no figure available, MATLAB calls the `figure` function to create one. Tab groups are containers for tabs. They allow you to identify the selected tab, and detect when the user selects a different tab.

`tg = uitabgroup(Name,Value)` specifies tab group property values using one or more name-value pair arguments.

`tg = uitabgroup(parent)` creates the tab group in the specified parent container. The parent container can be a figure created with either the `figure` or `uifigure` function, or a child container such as a panel. Property values for `uitabgroup` vary slightly depending on whether the app is created with the `figure` or `uifigure` function. For more information, see “Name-Value Pair Arguments” on page 1-16335.

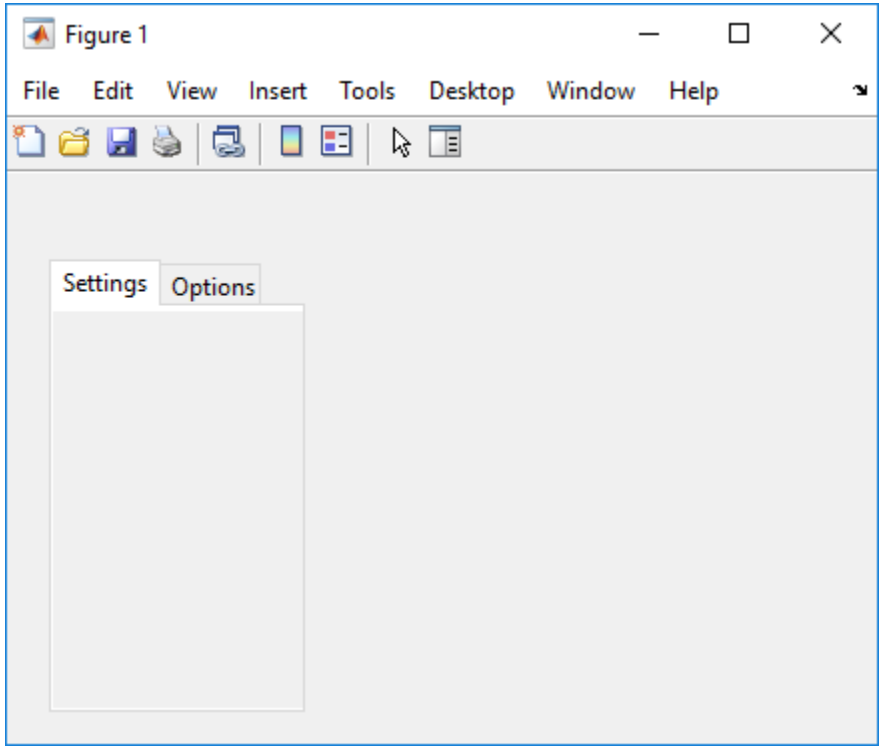
`tg = uitabgroup(parent,Name,Value)` specifies the parent container and one or more properties values.

Examples

Tab Group Containing Two Tabs

Create a figure containing a tab group and two tabs.

```
f = figure;
tabgp = uitabgroup(f,'Position',[.05 .05 .3 .8]);
tab1 = uitab(tabgp,'Title','settings');
tab2 = uitab(tabgp,'Title','Options');
```



Get the currently selected tab.

```
currenttab = tabgp.SelectedTab
```

```
currenttab = 

  Tab (settings) with properties:

              Title: 'settings'
    BackgroundColor: [0.9400 0.9400 0.9400]
```

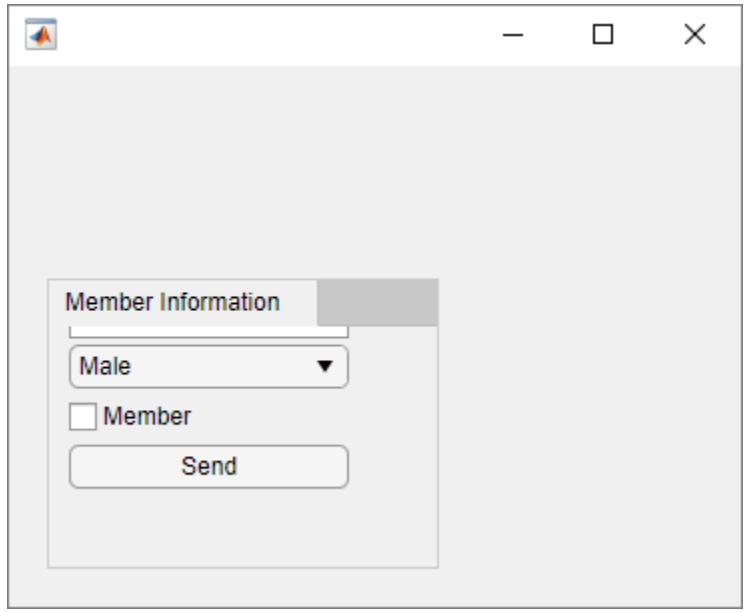
```
            Position: [0.0119 0.0089 0.9702 0.9107]
               Units: 'normalized'
```

Scrollable Tab in a Tab Group

The `Scrollable` property enables scrolling within a tab that has components outside its borders. To use scrolling, the figure must be created with the `uifigure` function. App Designer uses this type of figure for creating apps.

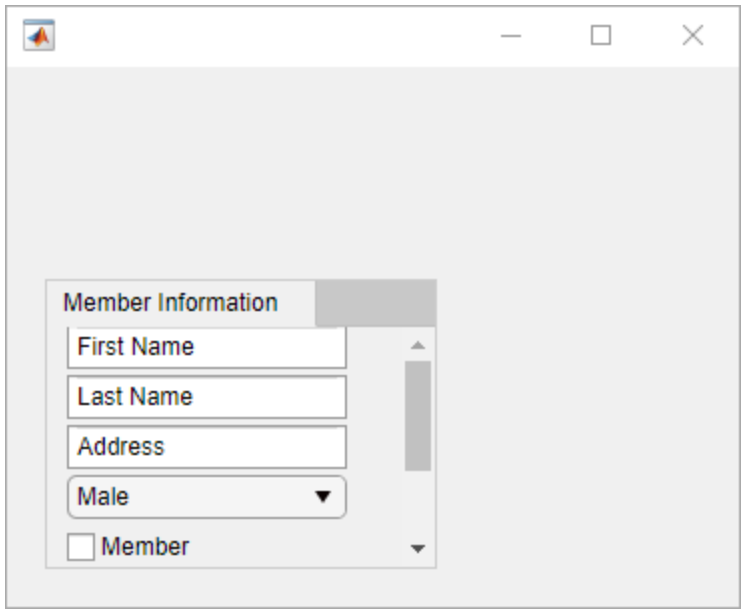
Create a tab group containing one tab. Add six UI components to the tab, where the first three lye outside the upper border of the tab.

```
f = uifigure;
tg = uitabgroup(f,'Position',[20 20 196 145]);
t = uitab(tg,'Title','Member Informaion');
ef1 = uieditfield(t,'text','Position',[11 165 140 22],'Value','First Name');
ef2 = uieditfield(t,'text','Position',[11 140 140 22],'Value','Last Name');
ef3 = uieditfield(t,'text','Position',[11 115 140 22],'Value','Addess');
dd = uidropdown(t,'Position',[11 90 140 22],'Items',{'Male','Female'});
cb = uicheckbox(t,'Position',[11 65 140 22],'Text','Member');
b = uibutton(t,'Position',[11 40 140 22],'Text','Send');
```



Enable scrolling by setting the `Scrollable` property of the tab to `'on'`. By default, the scroll bar displays at the top.

```
t.Scrollable = 'on';
```



Input Arguments

`parent` — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a figure created with either the `figure` or `uifigure` function, or a child container:

- Panels, tabs and button groups can be containers in either type of figure.
- Grid layouts can be containers only in figures created with the `uifigure` function.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments as `Name1,Value1,...,NameN,ValueN`.

Example: `'TabLocation','bottom'` specifies that the tab labels are at the bottom of the tab group.

- For a list of properties available for GUIDE, or for creating apps with the `figure` function, see Uitabgroup for GUIDE.
- For a list of properties available for App Designer, or for creating apps with the `uifigure` function, see TabGroup for App Designer.

See Also

Functions
`uitab`

Properties
TabGroup

Introduced in R2014b

TabGroup Properties

Control appearance and behavior of tab group

Note The properties listed here are valid for tab groups in GUIDE or in apps created with the `figure` function. If you are using App Designer or the `uifigure` function, see TabGroup Properties instead. For more information, see "GUIDE Migration Strategies".

Description

Tab groups are containers for grouping and managing tabs. Properties control the appearance and behavior of a tab group. Use dot notation to refer to a particular object and property.

```
f = figure;
tg = uitabgroup(f);
tg.Position = [.1 .1 .8 .8];
```

Properties

Tabs

`TabLocation` — Tab label location

`'top'` (default) | `'bottom'` | `'left'` | `'right'`

Tab label location, specified as `'top'`, `'bottom'`, `'left'`, or `'right'`. This property specifies the location of the tab labels with respect to the tab group.

`SelectedTab` — Currently selected tab

`Tab` object

Currently selected tab, specified as a `Tab` object.

Use this property to determine the currently selected tab within a tab group. You can also use this property to set the default selection.

The default value of the `SelectedTab` property is the first tab that you add to the tab group.

Interactivity

`Visible` — Tab group visibility

`'on'` (default) | `'off'`

Tab group visibility, specified as `'on'` or `'off'`. The `Visible` property determines whether the tab group displays on the screen. If the `Visible` property is set to `'off'`, the entire tab group is invisible, but you can still specify and access its properties.

Changing the size of an invisible container triggers the `SizeChangedFcn` callback if and when the container becomes visible.

Note Changing the `Visible` property of a container does *not* change the `Visible` property of its child components even though hiding the container prevents its children from displaying.

`UIContextMenu` — Context menu

empty `GraphicsPlaceholder` array (default) | uicontextmenu object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when the user right-clicks on a UI component. Create the context menu using the `uicontextmenu` function.

Position

`Position` — Location and size

`[left bottom width height]`

Location and size, specified as a four-element vector of the form `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the tag group
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the tab group

Element	Description
`width`	Distance between the right and left outer edges of the tab group
`height`	Distance between the top and bottom outer edges of the tab group

All measurements are in units specified by the `Units` property.

Note The `Position` values are relative to the parent container's drawable area. The drawable area is the area inside the borders of the container and does not include the area occupied by the title. If the parent container is a figure, then the drawable area also excludes the menu bar and tool bar.

Example: Modify One Value in the Position Vector

You can combine dot notation and array indexing when you want to change one value in the `Position` vector. For example, this code sets the width of the tab group to `.5` without changing any of the other `Position` values:

```
tg = uitabgroup;
tg.Position(3) = .5;
```

`InnerPosition` — Location and size

`[left bottom width height]`

Location and size, specified as a four-element vector of the form `[left bottom width height]`. All measurements are in units specified by the `Units` property.

This property value is identical to the `Position` and `OuterPosition` property values.

`OuterPosition` — Location and size

`[left bottom width height]`

Location and size, specified as a four-element vector of the form `[left bottom width height]`. All measurements are in units specified by the `Units` property.

This property value is identical to the `Position` and `InnerPosition` property values.

`Units` — Units of measurement

`'normalized'` (default) | `'pixels'` | `'inches'` | `'centimeters'` | `'points'` | `'characters'`

Units of measurement, specified one of the values from this table.

Units Value	Description
`'normalized'`	These units are normalized with respect to the parent container. The lower-left corner of the container maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'points'`	Points. One point equals 1/72nd of an inch.

Units Value	Description
`'characters'`	These units are based on the default uicontrol font of the graphics root object: • Character width = width of the letter `x`. • Character height = distance between the baselines of two lines of text. To access the default uicontrol font, use `get(groot,'defaultuicontrolFontName')` or `set(groot,'defaultuicontrolFontName')`.

MATLAB measures all units from the lower left corner of the parent object.

If you change the value of the `Units` property, it is good practice to return it to its default value after completing your computation to avoid affecting other functions that assume the `Units` property is set to the default value.

The order in which you specify the Units and Position properties has these effects:

- If you specify the `Units` property before the `Position` property, then MATLAB sets `Position` using the units you specified.
- If you specify the `Units` property after the `Position` property, MATLAB sets the position using the default `Units`. Then, MATLAB converts the `Position` values to the equivalent values in the units you specified.

Callbacks

`SelectionChangedFcn` — Selection changed callback

`''` (default) | function handle | cell array | character vector

Selection changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback function executes when the user selects a different tab within the tab group.

If you set the `SelectionChangedFcn` property to a function handle (or cell array containing a function handle), MATLAB passes a `SelectionChangedData` object containing event data as the second argument to the callback function. This object contains the properties described in the following table. You can access these properties inside the callback function using dot notation. For example, if the second argument in your callback is called `event`, then `event.NewValue` returns the currently selected `Tab`.

Property	Description
`OldValue`	Previously selected `Tab`, or `[]` if none was selected
`NewValue`	Currently selected `Tab`
`Source`	Component that executes the callback
`EventName`	`'SelectionChanged'`

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

`SizeChangedFcn` — Size change callback function

`' '` (default) | function handle | cell array | character vector

Size change callback function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

Define this callback to customize the app layout when the size of this container changes (e.g., when the user resizes the window).

The `SizeChangedFcn` callback executes when:

- This container becomes visible for the first time.
- This container is visible while its size changes.
- This container becomes visible for the first time after its size changes. This situation occurs when the size changes while the container is invisible, and then it becomes visible later.

Other important points to consider when defining a `SizeChangedFcn` callback:

- Consider delaying the display of this container until after all the variables that the `SizeChangedFcn` uses are defined. This practice can prevent the `SizeChangedFcn` callback from returning an error. To delay the display of the container, set its `Visible` property to `'off'`. Then, set the `Visible` property to `'on'` after you define the variables that your `SizeChangedFcn` callback uses.
- If your app contains nested containers, they resize from the inside out.
- To access the container that is resizing from within the `SizeChangedFcn`, refer to the source object (the first input argument in the callback) or use the `gcbo` function.

Tip As an easy alternative to specifying a `SizeChangedFcn` callback, you can set the `Units` property of all the objects you put inside a container to `'normalized'`. Doing so makes those components scale proportionally with the container.

See “Lay Out a UI Programmatically” for more information about managing layouts with `SizeChangedFcn` callbacks.

`ButtonDownFcn` — Button-press callback function

`''` (default) | function handle | cell array | character vector

Button-press callback function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

The `ButtonDownFcn` callback is a function that executes when the user clicks a mouse button within the container.

`CreateFcn` — Component creation function

`''` (default) | function handle | cell array | character vector

Component creation function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

This property specifies a callback function to execute when MATLAB creates the component. MATLAB initializes all component property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Use the `gcbo` function in your `CreateFcn` code to get the component object that is being created.

Setting the `CreateFcn` property on an existing component object has no effect.

`DeleteFcn` — Component deletion function

`''` (default) | function handle | cell array | character vector

Component deletion function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

The `DeleteFcn` property specifies a callback function to execute when MATLAB deletes the component (for example, when the user closes the window). MATLAB executes the `DeleteFcn` callback before destroying the properties of the component object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

Use the `gcbo` function in your `DeleteFcn` code to get the component object that is being deleted.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed:

- A value of `'on'` allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- A value of `'off'` blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn`, or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

- MATLAB does not save the state of properties or the display when an interruption occurs. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

See “Interrupt Callback Execution” for an example that shows how the `Interruptible` and `BusyAction` properties affect the behavior of a program.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` (default) or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

The `BusyAction` property of the source of the interrupting callback determines how MATLAB handles its execution. The `BusyAction` property has these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Do not execute the interrupting callback.

Whenever MATLAB invokes a callback, that callback always attempts to interrupt an executing callback. The `Interruptible` property of the object whose callback is running determines if interruption is allowed. If `Interruptible` is set to:

- `on` — Interruption occurs at the next point where MATLAB processes the queue. This is the default.
- `off` — The `BusyAction` property (of the object owning the interrupting callback) determines if MATLAB enqueues or ignores the interrupting callback.

See “Interrupt Callback Execution” for an example that shows how the `BusyAction` and `Interruptible` properties affect the behavior of a program.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent object

`Figure` | `Panel` | `Tab`

Parent object, specified as a `Figure`, `Panel`, or `Tab` object. Use this property to specify the parent container when creating a tab group or to move an existing tab group to a different parent container.

`Children` — TabGroup children

empty `GraphicsPlaceholder` array (default) | 1-D array of component objects

`TabGroup` children, returned as an empty `GraphicsPlaceholder` or a 1-D array of component objects. The children of `TabGroup` objects are `Tab` objects.

You cannot add or remove children using the `Children` property. Use this property to view the list of children or to reorder the children. The order of the children in this array reflects the order of the tabs displayed on the screen.

To add a child to this list, set the `Parent` property of the child component to be the `TabGroup` object.

Objects with the `HandleVisibility` property set to `'off'` do not list in the `Children` property.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object handle in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. The `HandleVisibility` property also controls the visibility of the object’s handle in the

parent figure's `CurrentObject` property. Handles are still valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object handle is always visible.
`'callback'`	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.

Set the graphics root `ShowHiddenHandles` property to `'on'` to make all handles visible, regardless of their `HandleVisibility` value. This setting has no effect on their `HandleVisibility` values.

Identifiers

Type — Type of graphics object
`'uitabgroup'`

This property is read-only.

Type of graphics object, returned as `'uitabgroup'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any array. Specifying `UserData` can be useful for sharing data within apps. See “Share Data Among Callbacks” for more information.

See Also

`guide` | `uitab` | `uitabgroup`

Topics

“Access Property Values”
“Default Property Values”

Introduced in R2014b

uitable

Create table user interface component

Syntax

```
uit = uitable
uit = uitable(Name,Value)
uit = uitable(parent)
uit = uitable(parent,Name,Value)
```

Description

`uit = uitable` creates a table user interface component in the current figure and returns the `Table` UI component object. If there is no figure available, MATLAB calls the `figure` function to create one.

`uit = uitable(Name,Value)` specifies property values of the table UI component using one or more name-value pair arguments.

`uit = uitable(parent)` creates the table in the specified parent container. The parent container can be a figure created with either the `figure` or `uifigure` function, or a child container such as a panel. Property values for `uitable` vary slightly depending on whether the app is created with the `figure` or `uifigure` function. For more information, see “Name-Value Pair Arguments” on page 1-16358.

`uit = uitable(parent,Name,Value)` specifies the parent container and one or more property values.

Examples

Display Table Array Data

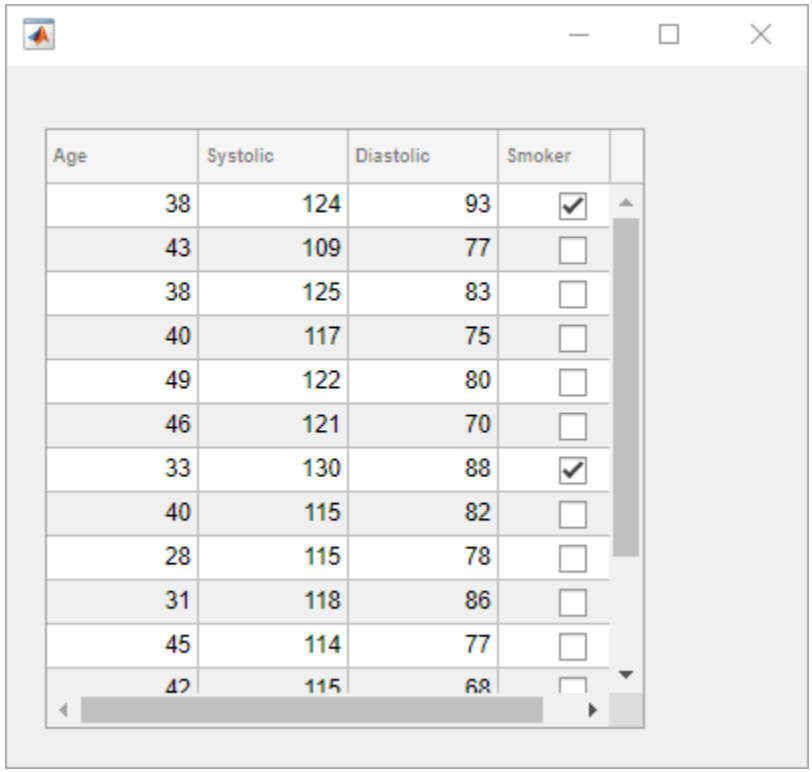
Starting in R2018a, you can display `table` array data in a table UI component. This type of data is supported only when the table UI component is in a figure created with the `uifigure` function. App Designer uses this type of figure for creating apps.

Create table array `t` by calling the `readtable` function to read data from a file. Select four variables and 15 rows from `t`.

```
t = readtable('patients.xls');
vars = {'Age','Systolic','Diastolic','Smoker'};
t = t(1:15,vars);
```

Create a table UI component, and specify `t` as the data.

```
fig = uifigure;
uit = uitable(fig,'Data',t);
```



Update Visualizations for Sorted Display Data

Starting in R2019a, you can sort the columns of a table UI component that has `table` array data stored in its `Data` property. This type of data is supported only when the table UI component is in a figure created with the `uifigure` function. App Designer uses this type of figure for creating apps.

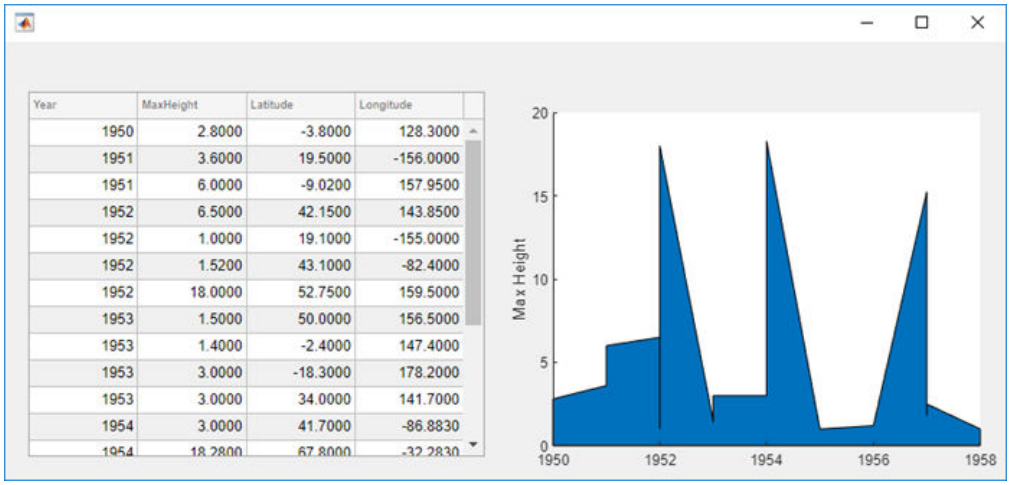
Display `table` array data and update the plot when a user sorts the columns or edits the cells.

First, create a program file called `tsunamisData.m`. Within the program file:

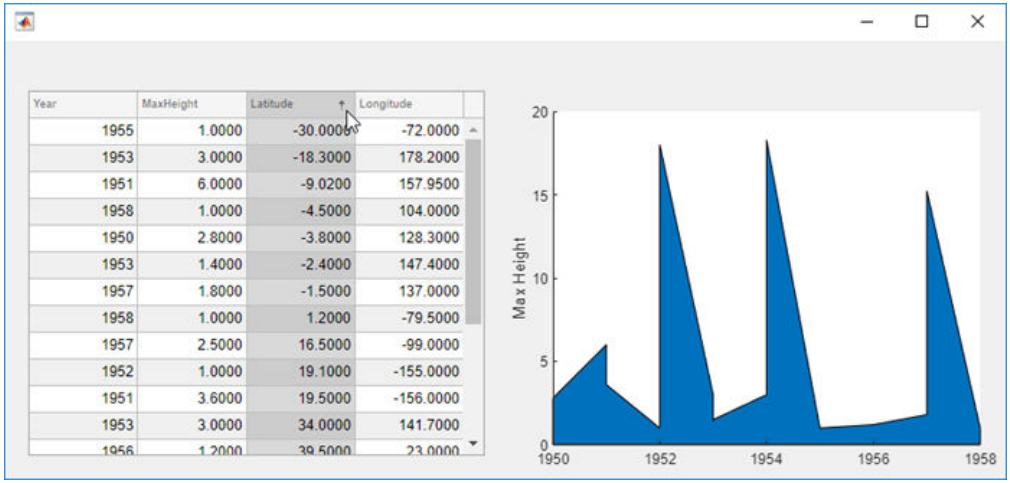
- Create a `table` array by calling the `readtable` function.

- Create a sortable and editable table UI component to display in a figure. Store the `table` array to component's `Data` property.
- Specify a `DisplayDataChangedFcn` callback that uses the `DisplayData` property to update the plot when a user sorts columns or edits cells in the table UI component.

```
function tsunamisData
t = readtable('tsunamis.xlsx');
vars = {'Year','MaxHeight','Latitude','Longitude'};
t = t(1:20,vars);

fig = uifigure;
fig.Position(3:4) = [822 360];

uit = uitable(fig);
uit.Data = t;
uit.ColumnSortable = [false true true true];
uit.ColumnEditable = true;
uit.Position(3) = 375;
uit.DisplayDataChangedFcn = @updatePlot;

ax = uiaxes(fig);
ax.Position(1) = 415;
ax.YLabel.String = 'Max Height';
x = t.Year;
y = t.MaxHeight;
area(ax,x,y)

    function updatePlot(src,event)
        t = uit.DisplayData;
        x = t.Year;
        y = t.MaxHeight;
        area(ax,x,y)
    end

end
```



A sortable column displays arrows in the header when you hover your mouse over it. Find a sortable column and sort the table. Notice how the displayed data and the plot update after sorting.



Add Style to Specific Cells in Table

Starting in R2019b, you can style rows, columns, or cells of a table UI component using the `uistyle` and `addStyle` functions. Styles are only supported when the table UI component is in a figure created with the `uifigure` function. App Designer uses this type of figure for creating apps.

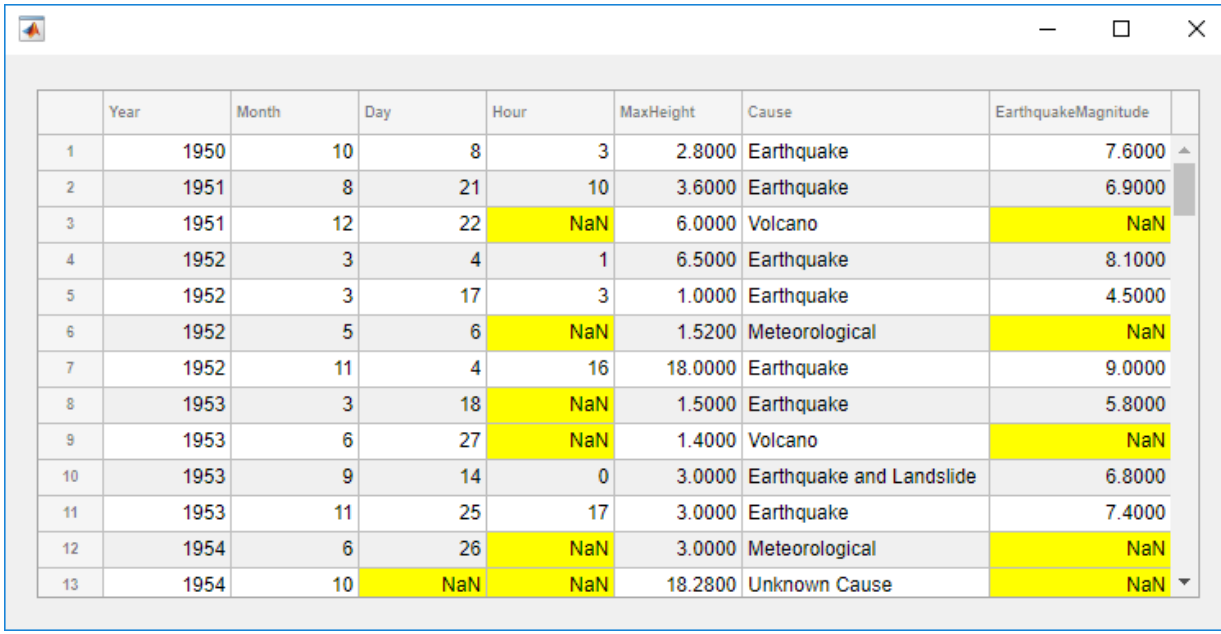
Style cells in a table UI component that contain missing values. In this case, add a yellow background color style to cells that have `NaN` values.

Read tsunami sample data into the workspace as a table array. Then, create a table UI component to display the data.

```
tdata = readtable('tsunamis.xlsx');
vars = {'Year','Month','Day','Hour','MaxHeight','Cause','EarthquakeMagnitude'};
tdata = tdata(1:100,vars);

fig = uifigure('Position',[500 500 750 350]);
uit = uitable(fig);
uit.Position = [20 20 710 310];
uit.Data = tdata;
uit.RowName = 'numbered';
```

Use the `ismissing` function to get a logical array of the table elements that contain missing values. Find the row and column subscripts for the elements that have `NaN` values. Finally, create a yellow background color style and add it to the cells with `NaN` values in the table UI component.

```
styleIndices = ismissing(tdata);
[row,col] = find(styleIndices);

s = uistyle('BackgroundColor','yellow');
addStyle(uit,s,'cell',[row,col]);
```

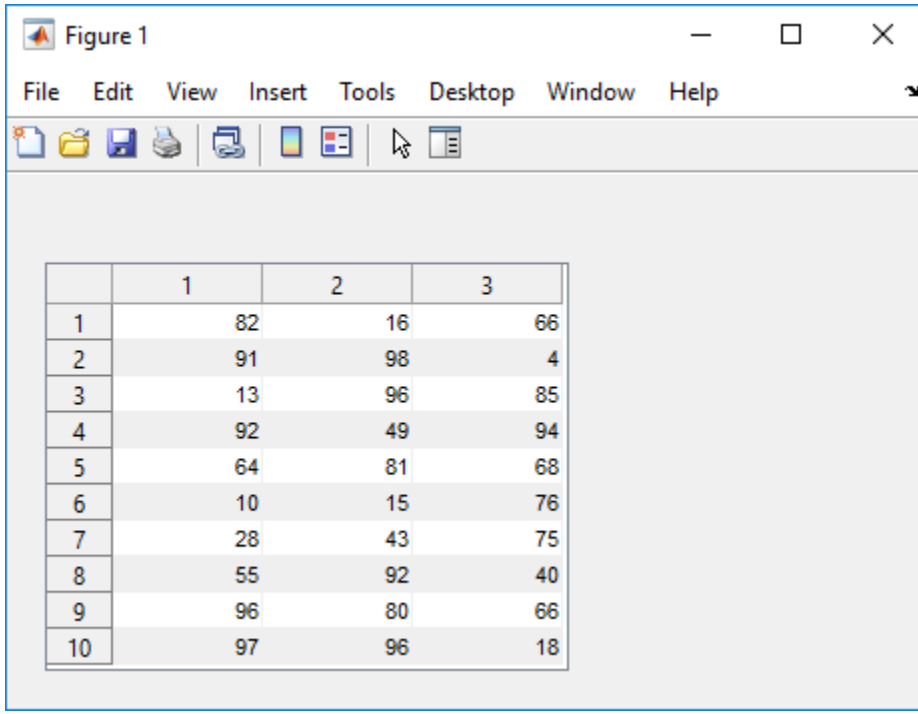


	Year	Month	Day	Hour	MaxHeight	Cause	EarthquakeMagnitude
1	1950	10	8	3	2.8000	Earthquake	7.6000
2	1951	8	21	10	3.6000	Earthquake	6.9000
3	1951	12	22	NaN	6.0000	Volcano	NaN
4	1952	3	4	1	6.5000	Earthquake	8.1000
5	1952	3	17	3	1.0000	Earthquake	4.5000
6	1952	5	6	NaN	1.5200	Meteorological	NaN
7	1952	11	4	16	18.0000	Earthquake	9.0000
8	1953	3	18	NaN	1.5000	Earthquake	5.8000
9	1953	6	27	NaN	1.4000	Volcano	NaN
10	1953	9	14	0	3.0000	Earthquake and Landslide	6.8000
11	1953	11	25	17	3.0000	Earthquake	7.4000
12	1954	6	26	NaN	3.0000	Meteorological	NaN
13	1954	10	NaN	NaN	18.2800	Unknown Cause	NaN

Display Array of Numbers

Create a table UI component that displays a 10-by-3 array of random integers. The `Data` property specifies the values to display, and the `Position` property specifies the location and size of the table within the figure.

```
f = figure;
uit = uitable(f,'Data',randi(100,10,3),'Position',[20 20 262 204]);
```



Display Mixed Data Types

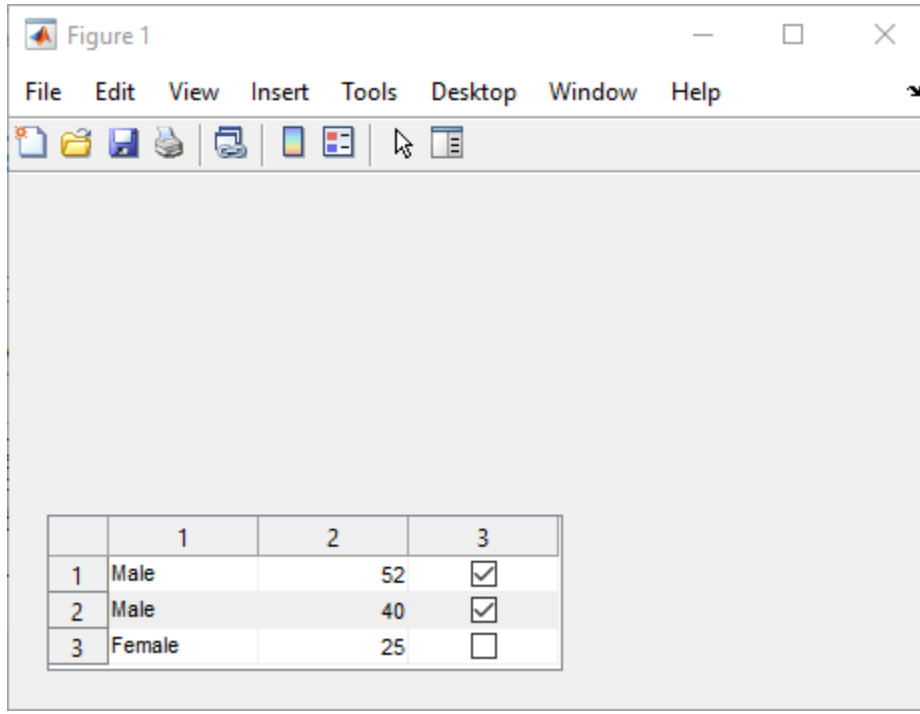
Table UI components can accommodate a mixture of different data types across the columns.

Create an empty `Table` UI component.

```
f = figure;
uit = uitable(f);
```

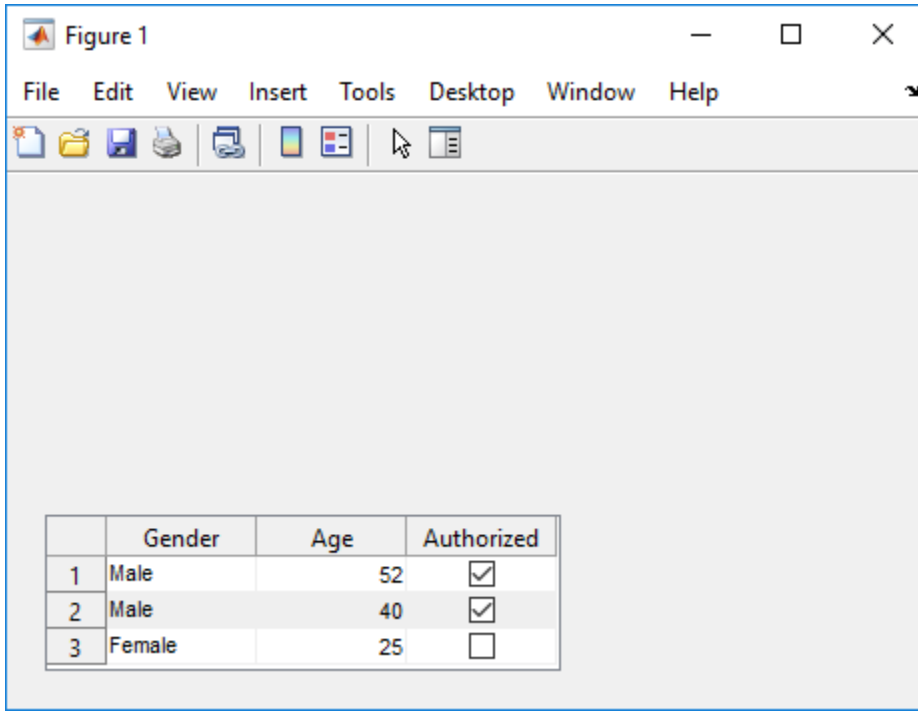
Set the `Data` property to populate the data as a cell array that contains a mixture of different types. Then set the `Position` property to adjust the location and size of the table to fit the data.

```
d = {'Male',52,true;'Male',40,true;'Female',25,false};
uit.Data = d;
uit.Position = [20 20 258 78];
```



Set the `ColumnName` property to change the column headings to descriptive names. Set the `ColumnEditable` property to `true` so that users can edit the data in the UI. When a user changes a value in the UI, the `Data` property updates to reflect that change.

```
uit.ColumnName = {'Gender','Age','Authorized'};
uit.ColumnEditable = true;
```



Input Arguments

parent — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a figure created with either the `figure` or `uifigure` function, or a child container:

- Panels, tabs and button groups can be containers in either type of figure.
- Grid layouts can be containers only in figures created with the `uifigure` function.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (`' '`). You can specify several name and value pair arguments as `Name1,Value1,...,NameN,ValueN`.

Example: `'Data',[1 2 3; 4 5 6]`

You can set `Table` properties using Name-Value pair arguments.

- For a list of properties available for App Designer, or for creating apps with the `uifigure` function, see Table for App Designer.
- For a list of properties available for GUIDE, or for creating apps with the `figure` function, see Uitable for GUIDE.

See Also

Functions
`addStyle` | `removeStyle` | `uistyle`

Properties
Style Properties | Table

Topics

"Table Array Data Types in App Designer Apps"
"Programmatic App that Displays a Table"

Introduced in R2008a

Table Properties

Control appearance and behavior of table UI component

Note The properties listed here are valid for tables in GUIDE or in apps created with the `figure` function. If you are using App Designer or the `uifigure` function, see Table Properties instead. For more information, see “GUIDE Migration Strategies”.

Description

Table UI components display rows and columns of data in an app. The `uitable` function creates a table UI component and sets any required properties before displaying it. By changing property values, you can modify certain aspects of its appearance and behavior. Use dot notation to refer to a specific object and property.

```
f = figure;
t = uitable(f,'Data',[1 2 3; 4 5 6; 7 8 9]);
t.FontSize = 9;
```

Properties

Table

`Data` — Table data

numeric array | logical array | cell array

Table data, specified as a numeric array, logical array, or cell array. The table data can be any numeric type, `logical`, or `char`. Use a cell array to specify a mixture of different data types.

Table data is displayed in the format specified by the `ColumnFormat` property. If there is a mismatch between the data type of the table data and the value of the `ColumnFormat` property, MATLAB converts the data or displays a warning. See the `ColumnFormat` property description for more information.

To prevent warnings that might occur when users enter invalid data, write a `CellEditCallback` function to convert the data to the appropriate type.

If the number of rows in the `Data` property array does not match the number of elements in the `RowName` array, then the number of rows in the resulting table is the larger of the two values. The same is true when the `ColumnName` property does not match the number of columns in the `Data` property array.

Example: `t = uitable('Data',rand(10,3))`

Example: `t = uitable('Data',{'blue' 5 true; 'orange' 25 false})`

`ColumnName` — Column heading names
`'numbered'` (default) | n-by-1 cell array of character vectors | n-by-1 categorical array | empty cell array (`{}`) | ...

Column heading names, specified as one of these values:

- `'numbered'` — The column headings are sequential numbers that start at `1`.
- Cell array of character vectors or categorical array — Each element of the array becomes the name of a column. If you specify a `1`-by-`n` cell array, MATLAB stores and returns the value as a `n`-by-`1` cell array. If you specify an `m`-by-`n` array, MATLAB reshapes the array into a column vector. Specify a multiline column name by including a vertical slash (`|`) in the column name. For example, the value, `'Telephone|Number'`, produces a column heading with a newline character between the words, “Telephone” and “Number”.
- Empty cell array (`{}`) — The table has no column headings.
- Empty matrix (`[]`) — The table has no column headings

If the number of columns in the `Data` property array does not match the number of elements in the `ColumnName` array, then the number of columns in the resulting table is the larger of the two values.

Example: `t = uitable('ColumnName',{'Name'; 'Number'},'Data',{'Bob' 5})`

Example: `t = uitable('ColumnName',{'Name'; []},'Data',{'Bob' 5})`

Example: `t = uitable('ColumnName',{'Name'; 'Telephone|Number'},'Data',{'Bob','555-1212'})`

`ColumnWidth` — Width of table columns
`'auto'` (default) | 1-by-n cell array

Width of table columns, specified as `'auto'` or as a `1`-by-`n` cell array.

Each column in the cell array corresponds to a column in the table. The values are in pixel units. If you specify `'auto'`, then MATLAB calculates the width of the column automatically using several factors, one of which is the `ColumnName` property value.

You can combine fixed column widths and `'auto'` column widths in a cell array, or you can specify a single value of `'auto'` to make all column widths automatic.

Selecting **Auto Width** in the Table Property Editor has the same effect as setting the `ColumnWidth` property to `'auto'`.

Example: `t = uitable('ColumnWidth','auto','Data',[1 2 3; 4 5 6])`

Example: `t = uitable('ColumnWidth',{64 60 40},'Data',[1 2 3; 4 5 6])`

Example: `t = uitable('ColumnWidth',{64 'auto' 40},'Data',[1 2 3; 4 5 6])`

`ColumnEditable` — Ability to edit column cells

`[]` (default) | logical 1-by-n array | logical scalar

Ability to edit column cells, specified as:

- An empty logical array (`[]`) — No columns are editable.
- A logical 1-by-n array — This array specifies which columns are editable. The value of n is equal to the number of columns in the table. Each value in the array corresponds to a table column. A value of `true` in the array makes the cells in that column editable. A value of `false` makes the cells in that column uneditable.
- A logical scalar— The entire table editable or uneditable.

Example: `t = uitable('Data',rand(10,3),'ColumnEditable',[false true true])`

Example: `t = uitable('Data',rand(10,3),'ColumnEditable',false)`

Table columns that contain check boxes or pop-up menus must be editable so the user can interact with these controls.

`ColumnFormat` — Cell display format

empty cell array (`{}`) (default) | 1-by-n cell array of character vectors

Cell display format, specified as an empty cell array or a 1-by-n cell array of character vectors.

This property determines how the data in each column displays, and the constraints for editing that data in the UI. The elements of the cell array correspond to columns in the `Data` property array. If you do not want to specify a display format for a particular column, specify `[]` for that column. If you do not specify a format for a column, MATLAB determines the default display by the data type of the data in the cell.

Elements of the cell array must be one of the values described in the following table.

Cell Format Value	Description
`'char'`	Display a left-justified values To edit a cell, the user types text to replace the existing value. If an element in the `Data` property array is logical, then `true` or `false` appears in the table.
`'logical'`	Display check boxes. To edit a cell, the user selects or clears the check box. Then, MATLAB sets the corresponding `Data` value to `true` or `false`. The `ColumnEditable` property value must be `true` to allow users to select or deselect the check boxes. Initially, a check box is selected when the corresponding `Data` value is `true`. The corresponding values in the `Data` property array must be of type `logical` to ensure the data displays correctly in the table.

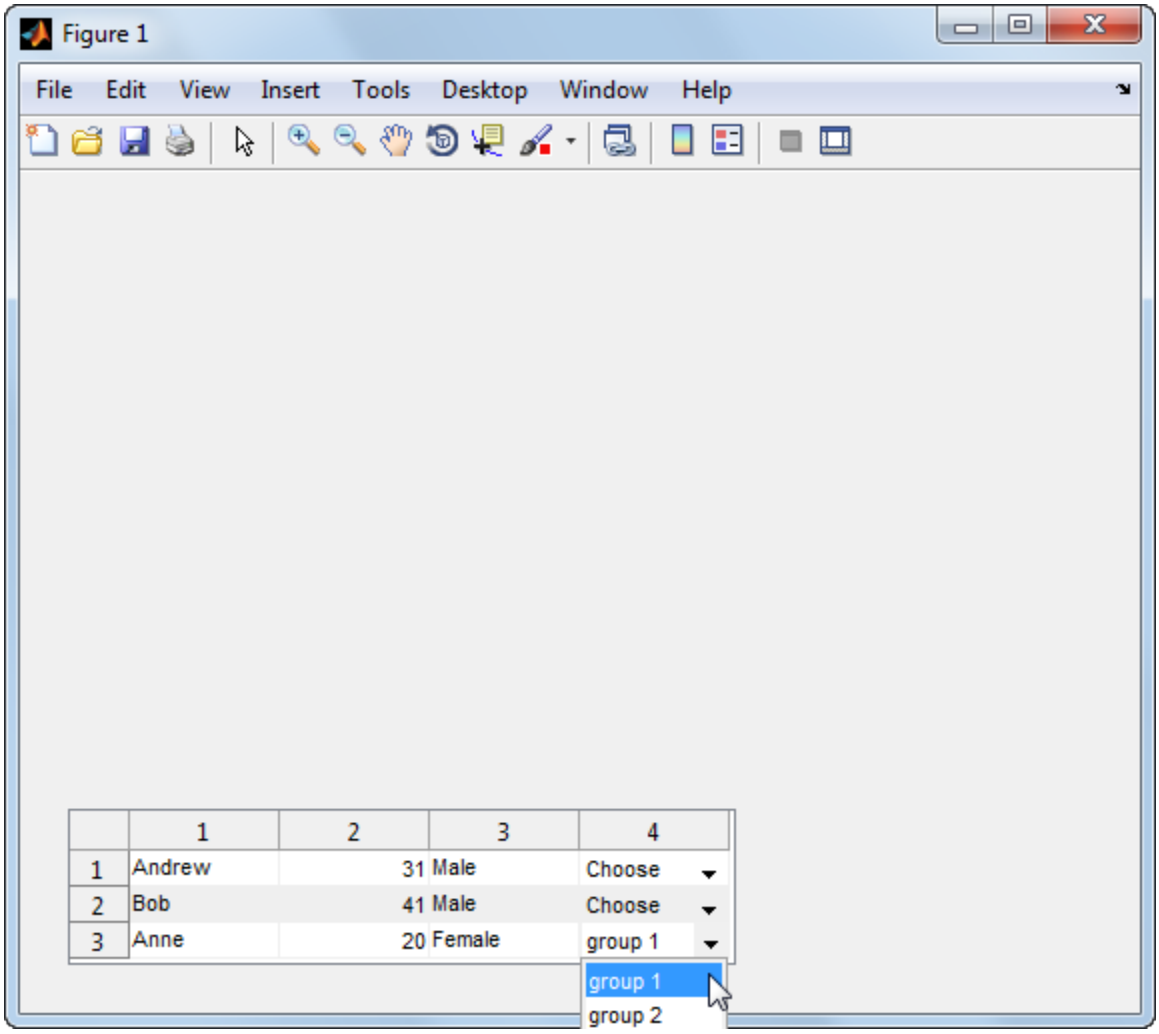
Cell Format Value	Description
`'numeric'`	Display a right-justified value equivalent to the Command Window display for numeric data. If an element in the `Data` property array is logical, then `1` or `0` appears in the table. If an element in the `Data` property array is not numeric and not logical, then `NaN` appears in the table. To edit a cell, the user can enter any text. If a user enters text that represents a constant, such as `pi`, you can code the `CellEditCallback` function to convert the value to the numeric equivalent. In this case, MATLAB attempts to convert the user-entered text to a numeric value and stores it in the `Data` property. Then, the `CellEditCallback` function executes. See the `CellEditCallback` description for an example.
A `1`-by-`n` cell array of character vectors, such as `{'one' 'two' 'three'}`	Display a pop-up menu. To edit a cell, the user selects an item from the pop-up menu. MATLAB sets the corresponding `Data` property array value to the selected menu item. The `ColumnEditable` property value must be `true` to allow users to select items in the pop-up menu.
A format name accepted by the `format` function, such as: `'short'` or `'long'`	Display the `Data` property values using the specified format.

Effect of Pop-Up Menu ColumnFormat and Various Data Types

If the `ColumnFormat` value defines a pop-up menu, the initial `Data` value does not have to be one of the options in that menu. The initial `Data` value appears until the user makes a different selection.

For instance, suppose the `Data` property value for a given column is `'Choose'` for all rows, and the `ColumnFormat` value specifies a pop-up menu with the choices of `'group 1'` and `'group 2'`. When MATLAB creates the table, those table cells display `'Choose'` until the user selects an item in the pop-up menu:

```
f = figure;
myData = {'Andrew' 31 'Male' 'Choose'; ...
          'Bob' 41 'Male' 'Choose';  ...
          'Anne' 20 'Female' 'Choose';};
t = uitable('Parent', f,...
            'Position', [25 25 334 78],...
            'ColumnFormat',({[] [] [] {'group 1' 'group 2'}}),...
            'ColumnEditable', true,...
            'Data', myData);
```



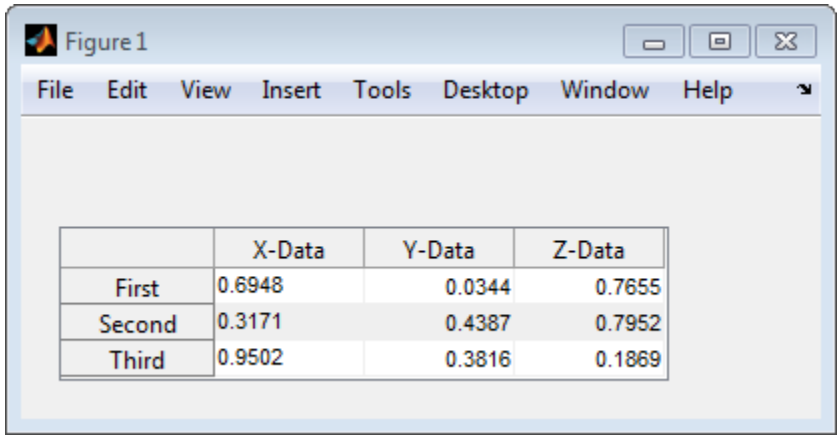
Data Display of Editable Columns

This table describes how various data types display with specific `ColumnFormat` values.

	ColumnFormat		
	`'numeric'`	`'char'`	`'logical'`

Data Type of Data Array Value	Any numeric type	Table displays number as-is.	MATLAB converts the value to text and displays it left-justified in the table. If MATLAB cannot convert the value, then `NaN` displays.	Not recommended. MATLAB might return a warning when the user edits the cell, unless you define a `CellEditCallback` function.
	`char`	Table displays the value right-justified, as if it is a number.	Table displays the value as-is.	Not recommended. MATLAB might return a warning when the user edits the cell, unless you define a `CellEditCallback` function.
	`logical`	Table displays logical values as numbers. MATLAB might return a warning when the user edits the cell, unless you define a `CellEditCallback` function.	Table displays logical value as left-justified `'true'` or `'false'`. MATLAB might return a warning when the user edits the cell, unless you define a `CellEditCallback` function.	Table displays logical values as check boxes.

For example, in the following table, the first column (X-Data) is left justified because the `ColumnFormat` value for that column is `'char'`.



RowName — Row heading names

`'numbered'` (default) | n-by-1 cell array of character vectors | n-by-1 categorical array | empty cell array (`{}`) | ...

Row heading names, specified as one of these values:

- `'numbered'` — The row headings are sequential numbers that start at `1`.
- Cell array of character vectors or categorical array — Each element of the array becomes the name of a row. Row names are restricted to one line of text. If you specify a `1`-by-`n` cell array, MATLAB stores and returns the value as a `n`-by-`1` cell array. If you specify an `m`-by-`n` array, MATLAB reshapes the array into a column vector.
- Empty cell array (`{}`) — The table has no row headings.
- Empty matrix (`[]`) — The table has no row headings

If the number of rows in the `Data` property array does not match the number of elements in the `RowName` array, then the number of rows in the resulting table is the larger of the two values.

Example: `t = uitable('RowName',{'Name';'Number'},'Data',{'Bob';5})`

Example: `t = uitable('RowName',{'Name';[]},'Data',{'Bob';5})`

RearrangeableColumns — Ability to rearrange table columns

`'off'` (default) | `'on'`

Ability to rearrange table columns, specified as `'off'` or `'on'`. This property provides a way to let users reorder the table columns (but not the labels) by clicking and dragging the column headers.

Note Rearranging table columns in the UI does not affect the columns in the `Data` property array.

Font

`FontName` — Font name

system supported font name | `'FixedWidth'`

Font name, specified as a system supported font name or `'FixedWidth'`. The default font depends on the specific operating system and locale.

To use a fixed-width font that looks good in any locale, specify `'FixedWidth'`. The actual fixed-width font used depends on the `FixedWidthFontName` property of the root object. Changing the `FixedWidthFontName` property causes an immediate update of the display to use the new font.

Example: `'Arial'`

`FontSize` — Font size

positive number

Font size, specified as a positive number. The `FontUnits` property specifies the units. The default size is system-dependent.

Example: `12`

Example: `12.5`

`FontWeight` — Font weight

`'normal'` (default) | `'bold'`

Font weight, specified as a value from the following table.

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than normal

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold font weight. Therefore, specifying a bold font weight still can result in the normal font weight.

Note The `'light'` and `'demi'` font weight values have been removed in R2014b. If you specify either of these values, the result is a normal font weight.

`FontAngle` — Font angle

`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. MATLAB uses this property to select a font from those available on your system. Setting this property to `'italic'` selects a slanted version of the font, if it is available on your system.

Note The `'oblique'` value has been removed. Use `'italic'` instead.

`FontUnits` — Font units

`'points'` (default) | `'normalized'` | `'inches'` | `'centimeters'` | `'pixels'`

Font units, specified as one of the values from this table.

Units Value	Description
`'points'`	Points. One point is 1/72nd of an inch.
`'normalized'`	Normalized values for specifying the font size as a fraction of the height. When you resize a UI component, MATLAB scales the displayed font to maintain that fraction.
`'inches'`	Inches.
`'centimeters'`	Centimeters.

Units Value	Description
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.

Interactivity

`Visible` — Table visibility

`'on'` (default) | `'off'`

Table visibility, specified as `'on'` or `'off'`. When `Visible` is `'off'`, the table is not visible, but you can query and set its properties.

To make your app start faster, set the `Visible` property of all components that are not initially displayed to `'off'`.

`Enable` — Operational state of table

`'on'` (default) | `'inactive'` | `'off'`

Operational state of table, specified as `'on'`, `'off'`, or `'inactive'`. The `Enable` property controls whether a UI component responds to button clicks. The are three possible values:

- `'on'` – The UI component is operational.
- `'off'` – The UI component is not operational and appears grayed-out.
- `'inactive'` – The UI component is not operational, but it has the same appearance as when `Enable` is `'on'`.

The value of the `Enable` property and the type of button click determine how the UI responds.

Enable Value	Response to Left-Click	Response to Right-Click
`'on'`	The `CellSelectionCallback` function executes (only for table cells, not header cells). The `Indices` property updates in the callback data object that MATLAB passes to the callback function.	**1** The figure's `WindowButtonDownFcn` callback executes. **2** The `ButtonDownFcn` callback executes.
`'off'` or `'inactive'`	**1** The figure's `WindowButtonDownFcn` callback executes. **2** The `ButtonDownFcn` callback executes.	**1** The figure's `WindowButtonDownFcn` callback executes. **2** The `ButtonDownFcn` callback executes.

`Tooltip` — Tooltip
character vector | string scalar | categorical array

Tooltip, specified as a character vector, string scalar, or categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip does not display when the component is disabled. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

To create multiple lines of text, use the `sprintf` function to insert newline characters (`'\n'`) in your text. For example:

```
txt = sprintf('Line 1\nLine 2');
```

Then set the `Tooltip` property to the value returned by `sprintf`.

`UIContextMenu` — Context menu
empty `GraphicsPlaceholder` array (default) | uicontextmenu object

Context menu, specified as a `ContextMenu` object. Use this property to display a context menu when the user right-clicks on a UI component. Create the context menu using the `uicontextmenu` function.

`TooltipString` — Tooltip (not recommended)
character vector | string scalar | categorical array

Tooltip, specified as a character vector, string scalar, or categorical array. The tooltip displays when the user hovers the mouse pointer over the component in the app. If you

specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Note The `TooltipString` property is not recommended starting in R2018b. Use the `Tooltip` property instead.

`Selected` — Selection state (not recommended)

`'off'` (default) | `'on'`

Note The behavior of the `Selected` property changed in R2014b, and it is not recommended. It no longer has any effect on `Table` objects. This property might be removed in a future release.

`SelectionHighlight` — Display of selection handles (not recommended)

`'on'` (default) | `'off'`

Note The behavior of the `SelectionHighlight` property changed in R2014b, and it is not recommended. It no longer has any effect on `Table` objects. This property might be removed in a future release.

Color and Styling

`ForegroundColor` — Cell text color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Cell text color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`BackgroundColor` — Background color

`[1 1 1; 0.94 0.94 0.94]` (default) | RGB triplet | m-by-3 matrix of RGB triplets

Background color, specified as an RGB triplet or an m-by-3 matrix of RGB triplets. An RGB triplet is a row vector that specifies the intensities of the red, green, and blue components of the color. The intensities must be in the range, `[0,1]`. Color names are not valid.

Specify an m-by-3 matrix when you want the shading of the table rows to follow a repeating pattern of m different colors. Each row of the matrix must be an RGB triplet.

MATLAB uses the rows of the matrix when the `RowStriping` property is `'on'`. The table background is not striped unless both `RowStriping` is `'on'` and `BackgroundColor` is an m-by-3 matrix.

Example: `t = uitable('Data',rand(10,3),'BackgroundColor',[0.85 0.85 1])`

Example: `t = uitable('Data',rand(10,3),'BackgroundColor',[1 1 1 ;0.85 0.85 1])`

The following table lists the RGB triplets for certain colors.

Color	RGB Triplet
Yellow	`[1 1 0]`
Magenta	`[1 0 1]`
Cyan	`[0 1 1]`
Red	`[1 0 0]`
Green	`[0 1 0]`
Blue	`[0 0 1]`
White	`[1 1 1]`
Black	`[0 0 0]`

`RowStriping` — Alternate row shading
`'on'` (default) | `'off'`

Alternate row shading, specified as `'on'` or `'off'`. This property controls the shading pattern of the table rows.

When the `RowStriping` value is set to `'on'`, the `BackgroundColor` matrix specifies the row colors to display in a repeating pattern. If the `BackgroundColor` matrix has only one row, then the shading is the same in all table rows.

When `RowStriping` is set to `'off'`, then the first color in the `BackgroundColor` matrix defines the shading for all rows in the table.

Position

`Position` — Location and size
`[left bottom width height]`

Location and size, specified as a four-element vector of the form `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the table
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the table
`width`	Distance between the right and left outer edges of the table
`height`	Distance between the top and bottom outer edges of the table

All measurements are in units specified by the `Units` property.

Note The `Position` values are relative to the parent container's drawable area. The drawable area is the area inside the borders of the container and does not include the area occupied by the title. If the parent container is a figure, then the drawable area also excludes the menu bar and tool bar.

Use the `Extent` property to determine proper sizing for a table with respect to the data. Set the `width` and `height` of `Position` property to the width and height of the `Extent` property. Be aware that if the table has large extents, doing this can cause the table to extend beyond the right or top edge of its parent container.

Example: Set Width and Height to Accommodate Data Size

You can combine dot notation and array indexing when you want to change one or two consecutive values in the `Position` vector. For example, this code sets the width and height of the table to match the `Extent` of the table:

```
t = uitable('Data',rand(10,3));
t.Position(3:4) = t.Extent(3:4);
```

`InnerPosition` — Location and size

`[left bottom width height]`

Location and size, specified as a four-element vector of the form `[left bottom width height]`. All measurements are in units specified by the `Units` property.

This property value is identical to the `Position` and `OuterPosition` property values.

`OuterPosition` — Location and size
`[left bottom width height]`

Location and size, specified as a four-element vector of the form `[left bottom width height]`. All measurements are in units specified by the `Units` property.

This property value is identical to the `Position` and `InnerPosition` property values.

`Extent` — Size of enclosing rectangle
four-element row vector

This property is read-only.

Size of enclosing rectangle, returned as a four-element row vector. The first two elements of the vector are always zero. The third and fourth elements are the width and height of the rectangle containing the table, respectively. All measurements are in units specified by the `Units` property.

MATLAB determines the size of the rectangle based on the current `Data`, `RowName` and `ColumnName` property values. MATLAB estimates the width and height values using the column and row widths. The estimated extent can be larger than the figure.

Consider using the `Extent` property value when specifying the width and height values of the `Position` property.

`Units` — Units of measurement
`'pixels'` (default) | `'normalized'` | `'inches'` | `'centimeters'` | `'points'` | `'characters'`

Units of measurement, specified as one of the values from this table.

Units Value	Description
`'pixels'` (default)	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.
`'normalized'`	These units are normalized with respect to the parent container. The lower-left corner of the container maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'points'`	Points. One point equals 1/72nd of an inch.
`'characters'`	These units are based on the default uicontrol font of the graphics root object: • Character width = width of the letter `x`. • Character height = distance between the baselines of two lines of text. To access the default uicontrol font, use `get(groot,'defaultuicontrolFontName')` or `set(groot,'defaultuicontrolFontName')`.

MATLAB measures all units from the lower left corner of the parent object.

This property affects the `Position` property. If you change the `Units` property, consider returning its value to the default value after completing your computation to avoid affecting other functions that assume the default value.

The order in which you specify the `Units` and `Position` properties has these effects:

- If you specify the `Units` before the `Position` property, then MATLAB sets `Position` using the units you specify.
- If you specify the `Units` property after the `Position` property, MATLAB sets the position using the default `Units`. Then, MATLAB converts the `Position` value to the equivalent value in the units you specify.

Callbacks

`CellEditCallback` — Cell edit callback function

function handle | cell array | character vector

Cell edit callback function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

This function executes when the user changes the contents of a table cell. You can use this callback function to perform calculations or validate user input.

If you specify this property as a function handle (or cell array containing a function handle), MATLAB passes a `CellEditData` object containing event data as the second argument to the callback function. This object contains the properties described in the following table. You can access these properties inside the callback function using dot notation.

Property	Description
`Indices`	1-by-2 array containing the row and column indices of the cell the user edited.
`PreviousData`	Previous cell data. The default is an empty matrix, `[]`.

Property	Description
EditData	User-entered value.
NewData	Value that MATLAB wrote to the Data property array. This value is either the same as EditData or a converted value. The NewData property is empty if MATLAB detects an error in the user-entered data.
Error	Error message returned if MATLAB detects an error in the user-entered data. The Error property is empty when MATLAB successfully writes the value to the Data property. If the Error property is not empty, then the CellEditCallback can display the message, or it can attempt to fix the problem.
Source	Table object that is executing the CellEditCallback function.
EventName	'CellEdit'

When the user edits a table cell, MATLAB performs these steps:

1. Tries to store the new value into the Data property of the table
2. Calls the CellEditCallback function (if it exists)

If the value results in an error and there is no CellEditCallback function, then the cell data reverts to its previous value and no error displays.

Example: Evaluate User Input

This example shows how to create a callback function that evaluates a user-entered data in a numeric table cell. Paste this code into an editor and run it to see how it works.

```
function myui
    f = figure;
    myData = { 'A '  31; 'B'  41; 'C'  5; 'D' 2.6};
    t = uitable('Parent',f,...
                'Position', [25 25 700 200], ...
                'Data',myData,...
                'ColumnEditable', [false true], ...
                'CellEditCallback',@converttonum);
        function converttonum(hObject,callbackdata)
             numval = eval(callbackdata.EditData);
             r = callbackdata.Indices(1)
```

```
            c = callbackdata.Indices(2)
            hObject.Data{r,c} = numval;
        end
end
```

When you run `myui`, you can change a value in the second column of the table. In response, the `converttonum` callback function executes. The `converttonum` function uses the `eval` function to evaluate your input. Then, it sets the cell data to the value of `numval`. For example, if you enter `pi` or`1+1`, the `converttonum` function sets the table cell value to a numeric representation of the input. Because there is no error checking in the `converttonum` function, invalid expressions return an error and the new value of the table cell becomes `NaN`.

`CellSelectionCallback` — Cell selection callback function

function handle | cell array | character vector

Cell selection callback function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

This callback function executes when the user performs one of the following actions:

- Highlights a data cell (not a row or column header cell) by clicking it or navigating to it with an arrow key
- Selects a rectangular group of cells by holding the **Shift** key while selecting the cells
- Deselects a cell by **Ctrl**-clicking it

If you specify this property as a function handle (or cell array containing a function handle), MATLAB passes a `CellSelectionChangeData` object containing event data as the second argument to the callback function. This object contains the properties listed in the following table.

Property	Description
`Indices`	n-by-2 array containing the row and column indices of the cell the user selected.
`Source`	The table object that is executing the `CellSelectionCallback` function.
`EventName`	`'CellSelection'`

`ButtonDownFcn` — Button-press callback function
`''` (default) | function handle | cell array | character vector

Button-press callback function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

The `ButtonDownFcn` callback is a function that executes when the user clicks a mouse button on the UI component. The callback executes in the following situations:

- The user right-clicks the table, and the `Enable` property is set to `'on'`.
- The user right-clicks or left-clicks the table, and the `Enable` property is set to `'off'` or `'inactive'`.

`KeyPressFcn` — Key press callback function
`''` (default) | function handle | cell array | character vector

Key press callback function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

This callback function executes when the `Table` object has focus and the user presses a key. If you do not define a function for this property, MATLAB passes key presses to the parent figure. Repeated key presses retain the focus of the `Table` object, and the function executes with each key press. If the user presses multiple keys at approximately the same time, MATLAB detects the key press for the last key pressed.

If you specify this property as a function handle (or cell array containing a function handle), MATLAB passes an object containing callback data as the second argument to the callback function. This object contains the properties described in the following table. You can access these properties inside the callback function using dot notation.

Property	**Description**	**Examples:**			
		a	**=**	**Shift**	**Shift-a**
`Character`	The character that displays as a result of pressing a key or keys. The character can be empty or unprintable.	`'a'`	`'='`	`''`	`'A'`
`Modifier`	A cell array containing the names of one or more modifier keys that are being pressed (such as, **Ctrl**, **Alt**, **Shift**).	`{1x0 cell}`	`{1x0 cell}`	`{'shift'}`	`{'shift'}`
`Key`	The key being pressed, identified by the (lowercase) label on the key, or a text description.	`'a'`	`'equal'`	`'shift'`	`'a'`
`Source`	The object that has focus when the user presses the key.	`Table` object	`Table` object	`Table` object	`Table` object
`EventName`	The action that caused the callback function to execute.	`'KeyPress'`	`'KeyPress'`	`'KeyPress'`	`'KeyPress'`

Pressing modifier keys affects the callback data in the following ways:

- Modifier keys can affect the `Character` property, but do not change the `Key` property.
- Certain keys, and keys modified with **Ctrl**, put unprintable characters in the `Character` property.

- **Ctrl**, **Alt**, **Shift**, and several other keys, do not generate `Character` property data.

You also can query the `CurrentCharacter` property of the figure to determine which character the user pressed.

`KeyReleaseFcn` — Key-release callback function
`''` (default) | function handle | cell array | character vector

Key-release callback function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

This callback function executes when the `Table` object has focus and the user releases a key.

If you specify this property as a function handle (or cell array containing a function handle), MATLAB passes an object containing callback data as the second argument to the callback function. This object contains the properties described in the following table. You can access these properties inside the callback function using dot notation.

Property	Description	Examples:			
		a	=	Shift	Shift-a
`Character`	Character interpretation of the key that was released.	`'a'`	`'='`	`''`	`'A'`
`Modifier`	Current modifier, such as `'control'`, or an empty cell array if there is no modifier.	`{1x0 cell}`	`{1x0 cell}`	`{1x0 cell}`	`{1x0 cell}`
`Key`	Name of the key that was released, identified by the lowercase label on the key, or a text description.	`'a'`	`'equal'`	`'shift'`	`'a'`

Property	Description	Examples:			
		a	=	Shift	Shift-a
`Source`	The object that has focus when the user presses the key.	`Table` object	`Table` object	`Table` object	`Table` object
`EventName`	The action that caused the callback function to execute.	`'ase'`	`'ase'`	`'ase'`	`'ase'`

Pressing modifier keys affects the callback data in the following ways:

- Modifier keys can affect the `Character` property, but do not change the `Key` property.
- Certain keys, and keys modified with **Ctrl**, put unprintable characters in the `Character` property.
- **Ctrl**, **Alt**, **Shift**, and several other keys, do not generate `Character` property data.

You also can query the `CurrentCharacter` property of the figure to determine which character the user pressed.

`CreateFcn` — Component creation function

`''` (default) | function handle | cell array | character vector

Component creation function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

This property specifies a callback function to execute when MATLAB creates the component. MATLAB initializes all component property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Use the `gcbo` function in your `CreateFcn` code to get the component object that is being created.

Setting the `CreateFcn` property on an existing component object has no effect.

`DeleteFcn` — Component deletion function

`''` (default) | function handle | cell array | character vector

Component deletion function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

The `DeleteFcn` property specifies a callback function to execute when MATLAB deletes the component (for example, when the user closes the window). MATLAB executes the `DeleteFcn` callback before destroying the properties of the component object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

Use the `gcbo` function in your `DeleteFcn` code to get the component object that is being deleted.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed:

- A value of `'on'` allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.

  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- A value of `'off'` blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn`, or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.
- MATLAB does not save the state of properties or the display when an interruption occurs. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

See “Interrupt Callback Execution” for an example that shows how the `Interruptible` and `BusyAction` properties affect the behavior of a program.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` (default) or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

The `BusyAction` property of the source of the interrupting callback determines how MATLAB handles its execution. The `BusyAction` property has these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Do not execute the interrupting callback.

Whenever MATLAB invokes a callback, that callback always attempts to interrupt an executing callback. The `Interruptible` property of the object whose callback is running determines if interruption is allowed. If `Interruptible` is set to:

- `on` — Interruption occurs at the next point where MATLAB processes the queue. This is the default.
- `off` — The `BusyAction` property (of the object owning the interrupting callback) determines if MATLAB enqueues or ignores the interrupting callback.

See “Interrupt Callback Execution” for an example that shows how the `BusyAction` and `Interruptible` properties affect the behavior of a program.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

`HitTest` — Ability to become current object
`'on'` (default) | `'off'`

Ability to become current object, specified as `'on'` or `'off'`:

- `'on'` — Sets the current object to the `Table` when the user clicks the component in the running app. Both the `CurrentObject` property of the `Figure` and the `gco` function return the `Table` as the current object.
- `'off'` — Sets the current object to be the closest ancestor of the `Table` whose `HitTest` is `'on'` when the user clicks the component in the running app.

Note Use the Enable property to enable or disable a component.

Parent/Child

`Parent` — Parent object

`Figure` | `Panel` | `ButtonGroup` | `Tab`

Parent object, specified as a `Figure`, `Panel`, `ButtonGroup`, or `Tab` object. Use this property to specify the parent container when creating a UI component or to move an existing UI component to a different parent container.

`Children` — `Table` children

empty array

`Table` children, returned as an empty array. `Table` objects have no children. Setting this property has no effect.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object handle in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. The `HandleVisibility` property also controls the visibility of the object’s handle in the parent figure's `CurrentObject` property. Handles are still valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object handle is always visible.
`'callback'`	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.

HandleVisibility Value	Description
`'off'`	The object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.

Set the graphics root `ShowHiddenHandles` property to `'on'` to make all handles visible, regardless of their `HandleVisibility` value. This setting has no effect on their `HandleVisibility` values.

Identifiers

Type — Type of graphics object
`'uitable'`

This property is read-only.

Type of graphics object, returned as `'uitable'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any array. Specifying `UserData` can be useful for sharing data within apps. See “Share Data Among Callbacks” for more information.

See Also

`guide` | `uitable`

Topics

“Access Property Values”

“Default Property Values”

Introduced in R2008a

Table Properties

Control appearance and behavior of table UI component

Description

Table UI components display rows and columns of data in an app. The `uitable` function creates a table UI component and sets any required properties before displaying it. By changing property values of a `Table` object, you can modify certain aspects of its appearance and behavior. Use dot notation to refer to a specific object and property.

```
fig = uifigure;
uit = uitable(fig,'Data',[1 2 3; 4 5 6; 7 8 9]);
uit.FontSize = 10;
```

The properties listed here are valid for table UI components in App Designer, or in apps created with the `uifigure` function. For table UI components used in GUIDE, or in apps created with the `figure` function, see Uitable.

Properties

Table

`Data` — Table data

table array | numeric array | logical array | cell array | string array | ...

Table data, specified as one of the following types of array:

- Table array — Displays any combination of data types that `table` arrays support, such as `datetime`, `duration`, and `categorical`.
- Numeric array — Displays numeric values such as `double` or `single`.
- Logical array — Displays check boxes. `true` values correspond to selected boxes, whereas `false` values display cleared boxes.
- Cell array — Displays any combination of numeric, logical, or character array values.
- String array — Displays characters and text.

- Cell array of character vectors — Displays characters and text.

To prevent warnings or `NaN` values that display when users enter invalid data into an editable cell, write a `CellEditCallback` function to convert the data to the appropriate type. When a user edits a cell, the `Data` property updates.

Specify a Table Array

Table arrays provide a convenient way to store tabular data as a MATLAB variable. The `table`, `readtable`, and `array2table` functions create table arrays. By contrast, the `uitable` function creates a `Table` UI component (a user interface component for an app).

When you specify the `Data` property of a `Table` UI component as a table array, then MATLAB sets the format of the `Table` UI component automatically based on the values in the table array:

- By default, the column names displayed in the app match the `VariableNames` property of the table array. Changing the `ColumnName` property of the `Table` UI component updates the UI, but it does not update the variable names in the table array.
- By default, the row names displayed in the app match the `RowNames` property of the table array. Changing the `RowName` property of the `Table` UI component updates the UI, but it does not update the row names in the table array.
- The data type of each table array variable controls formatting for the corresponding column in the app. If you try to set the `ColumnFormat` property, MATLAB returns a warning.

For more information on displaying table array data, see “Table Array Data Types in App Designer Apps”.

Specify Numeric, Logical, Cell, String Array, or Cell Array of Character Vectors

Use the `ColumnFormat` property to specify the format for data that is a numeric, logical, cell, or string array, or a cell array of character vectors. If data is edited and results in a mismatch between the data type of the data and the `ColumnFormat` property, MATLAB converts the data or displays a warning. See “Data Display of Editable Columns” on page 1-0 in the `ColumnFormat` property description for more information.

`DisplayData` — Table data in current display

table array | numeric array | logical array | cell array | string array | ...

This property is read-only.

Table data in the current display, returned as a table, numeric, logical, cell, or string array, or as a cell array of character vectors.

Use this property if you want to update your visualizations based on whether a user has sorted columns or edited cells in a table.

`DisplayData` updates when table columns are sorted or when cells are edited. If a user does not sort columns, then `DisplayData` has the same content as the value stored in the `Data` property. When a user edits a cell, the `Data` and `DisplayData` properties both update.

`ColumnName` — Column names

`'numbered'` | n-by-1 cell array of character vectors | n-by-1 categorical array | empty cell array (`{}`) | ...

Column names, specified as one of these values:

- `'numbered'` — The column headings are sequential numbers that start at `1`.
- Cell array of character vectors or categorical array — Each element of the array becomes the name of a column. Column names are restricted to one line of text. If you specify a 1-by-n cell array, MATLAB stores and returns the value as an n-by-1 cell array. If you specify an m-by-n array, MATLAB reshapes the array into a column vector.
- Empty cell array (`{}`) — The table has no column headings.
- Empty matrix (`[]`) — The table has no column headings

If the number of columns in the `Data` property array does not match the number of elements in the `ColumnName` array, then the number of columns in the resulting table is the larger of the two values.

If you specify the `Data` property as a table array, then the default column names match the `VariableNames` property of the table array. Changing the `ColumnName` property of the `Table` UI component updates the UI, but it will not update the variable names in the table array.

Example: `uit = uitable(uifigure,'ColumnName',{'Name';'Number'},'Data',{'Bob',5})`

Example: `uit = uitable(uifigure,'ColumnName',{'Name';[]},'Data',{'Bob',5})`

`ColumnWidth` — Width of table columns

`'auto'` (default) | 1-by-n cell array

Width of table columns, specified as `'auto'` or as a 1-by-n cell array.

Each column in the cell array corresponds to a column in the table. The values are in pixel units. If you specify `'auto'`, then MATLAB calculates the width of the column automatically using several factors, one of which is the `ColumnName` property value.

You can combine fixed column widths and `'auto'` column widths in a cell array, or you can specify a single value of 'auto' to make all column widths automatic. If the array you specify has fewer values than the number of columns, then the columns with no specified value keep the default value, `'auto'`. If the array has more values than the number of columns, MATLAB ignores the excess values.

Example: `uit = uitable(uifigure,'ColumnWidth','auto','Data',[1 2 3;4 5 6])`

Example: `uit = uitable(uifigure,'ColumnWidth',{64,60,40},'Data',[1 2 3;4 5 6])`

Example: `uit = uitable(uifigure,'ColumnWidth',{64,'auto',40},'Data',[1 2 3;4 5 6])`

`ColumnEditable` — Ability to edit column cells

`[]` (default) | logical 1-by-n array | logical scalar

Ability to edit column cells, specified as:

- An empty logical array (`[]`) — No columns are editable.
- A logical 1-by-n array — This array specifies which columns are editable. The value of `n` is equal to the number of columns in the table. Each value in the array corresponds to a table column. A value of `true` in the array makes the cells in that column editable. A value of `false` makes the cells in that column uneditable. If the array has more values than the number of columns, MATLAB ignores the excess values. If the array has fewer values than the number of columns, then the columns with no specified value are not editable.
- A logical scalar — The entire table is editable or uneditable.

When a user edits a cell, the `Data` property updates.

Example: `uit = uitable(uifigure,'Data',rand(10,3),'ColumnEditable',[false true true])`

Example: `uit = uitable(uifigure,'Data',rand(10,3),'ColumnEditable',false)`

To enable users to interact with the controls in table columns that contain check boxes or pop-up menus, set the `ColumnEditable` property to `true`.

If the `Data` property is a table array, then any variables that are multicolumn or contain non-editable data types, like `duration`, are not editable in the running app even when the `ColumnEditable` property is `true`. Table array variables that contain mixed data types in a cell array are editable in the running app, as long as the data types are editable.

`ColumnSortable` — Ability to sort columns

`[]` (default) | logical scalar | logical 1-by-`n` array

Ability to sort columns, specified as:

- An empty logical array (`[]`) — No columns are sortable.
- A logical scalar — The entire table is sortable (`true`) or unsortable (`false`).
- A logical 1-by-`n` array — This array specifies which columns are sortable. The value of `n` is equal to the number of columns in the table. Each value in the array corresponds to a table column. A value of `true` in the array makes that column sortable. A value of `false` makes that column unsortable. If the array has more values than the number of columns, MATLAB ignores the excess values. If the array has fewer values than the number of columns, then the columns that do not have specified values are not sortable.

Example: `uit = uitable(uifigure,'Data',rand(5),'ColumnSortable',true);`

Example: `uit = uitable(uifigure,'Data',rand(3),'ColumnSortable',[true true false]);`

If the `Data` property contains cell array data or table array data with cell array columns, then only columns with uniform data types of numeric or character arrays, or cell array of character vectors are sortable in the running app. Cell array columns with uniform logical data or nonuniform data types cannot be sorted in the running app, even when the `ColumnSortable` property is `true`.

`ColumnFormat` — Cell display format

empty cell array (`{}`) (default) | 1-by-`n` cell array of character vectors

Cell display format, specified as an empty cell array or a 1-by-n cell array of character vectors.

Do not set this property when the `Data` property contains a `table` array. For more information, see “Table Array Data Types in App Designer Apps”.

This property sets the format for displaying numeric, logical, cell, or string array, and cell array of character vectors data types. The elements of the cell array correspond to columns in the `Data` property array. If you do not want to specify a display format for a particular column, specify `[]` for that column. If you do not specify a format for a column, MATLAB determines the default display by the data type of the data in the cell.

Elements of the cell array must be one of the values described in the table.

Cell Format Value	Description
`'char'`	Display left-justified values. If an element in the `Data` property array is logical, then `true` or `false` appears in the table. To edit a cell, the user types text to replace the existing value.
`'logical'`	Display a center-justified check box. Initially, a check box is selected when the corresponding `Data` value evaluates to `true`. The corresponding values in the `Data` property array must be of type `logical` to ensure that the data displays correctly in the table. To edit a cell, the user selects or clears the check box. Then, MATLAB sets the corresponding `Data` value to `true` or `false`. The `ColumnEditable` property value must be `true` to allow users to select or clear the check boxes.

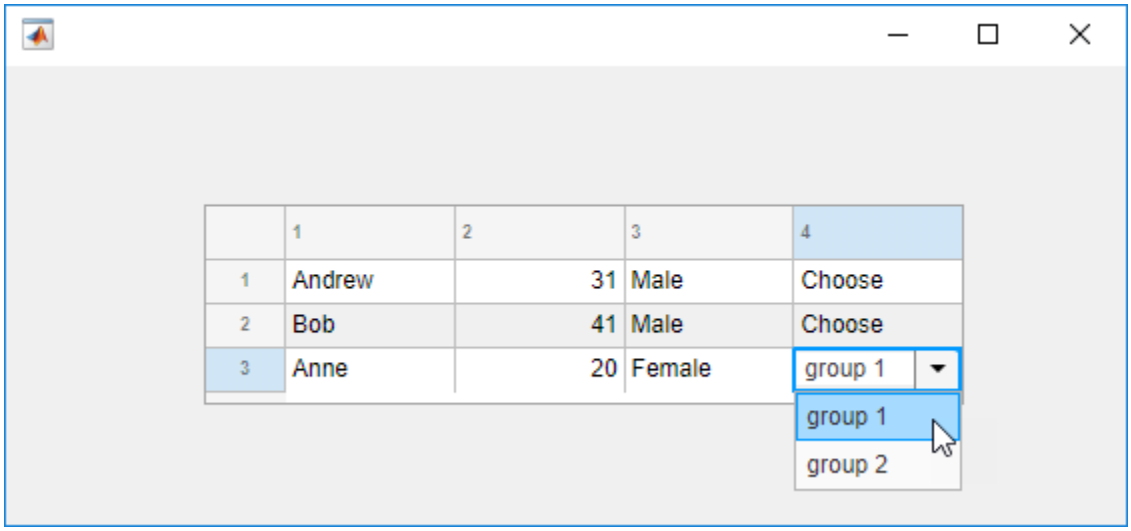
Cell Format Value	Description
`'numeric'`	Display a right-justified value equivalent to the Command Window display for numeric data. If an element in the `Data` property array is logical, then `1` or `0` appears in the table. If an element in the `Data` property array is not numeric and not logical, then `NaN` appears in the table. To edit a cell, the user can enter any text. If a user enters text that represents a constant, such as `pi`, you can code the `CellEditCallback` function to convert the value to the numeric equivalent. In this case, MATLAB attempts to convert the user-entered text to a numeric value and stores it in the `Data` property. Then, the `CellEditCallback` function executes. See the `CellEditCallback` description for an example.
A 1-by-`n` cell array of character vectors, such as `{'one','two','three'}`	Display a pop-up menu in an editable column. The value displays as left-justified whether the `ColumnEditable` property is set to `true` or `false`. To edit a cell, the user selects an item from the pop-up menu, or enters text to create a new item. MATLAB sets the corresponding `Data` property array value to the selected menu item. The `ColumnEditable` property value must be `true` to allow users to select items in the pop-up menu.
A format name accepted by the `format` function, such as: `'short'` or `'long'`	Display the `Data` property values using the specified format. The values display as right-justified.

Effect of Pop-Up Menu ColumnFormat and Various Data Types

If the `ColumnFormat` value defines a pop-up menu, the initial `Data` value does not have to be one of the options in that menu. The initial `Data` value appears until the user makes a different selection.

For instance, suppose the `Data` property value for a given column is `'Choose'` for all rows, and the `ColumnFormat` value specifies a pop-up menu with the choices of `'group 1'` and `'group 2'`. When MATLAB creates the table, those table cells display `'Choose'` until the user selects an item in the pop-up menu:

```
fig = uifigure;
myData = {'Andrew' 31 'Male' 'Choose'; ...
    'Bob' 41 'Male' 'Choose';  ...
    'Anne' 20 'Female' 'Choose';};
uit = uitable('Parent', fig, ...
    'Position', [100 150 380 100], ...
    'ColumnFormat',({[] [] [] {'group 1' 'group 2'}}), ...
    'ColumnEditable',true, ...
    'Data',myData);
```



Data Display of Editable Columns

This table describes how various data types display with specific `ColumnFormat` values.

		ColumnFormat		
		`'numeric'`	`'char'`	`'logical'`
Data Type of Data Array Value	Any numeric type	Table displays number as-is.	MATLAB converts the value to text and displays it left-justified in the table. If MATLAB cannot convert the value, then `NaN` displays.	Not recommended. MATLAB might return a warning when the user edits the cell, unless you define a `CellEditCallback` function.

	`char`	Table displays the value right-justified, as if it is a number.	Table displays the value as-is.	Not recommended. MATLAB might return a warning when the user edits the cell, unless you define a `CellEditCallback` function.
	`logical`	Table displays logical values as numbers. MATLAB might return a warning when the user edits the cell, unless you define a `CellEditCallback` function.	Table displays logical value as left-justified `'true'` or `'false'`. MATLAB might return a warning when the user edits the cell, unless you define a `CellEditCallback` function.	Table displays logical values as check boxes.

`RowName` — Row names

`'numbered'` | `n`-by-1 cell array of character vectors | `n`-by-1 categorical array | empty cell array (`{}`) | ...

Row names, specified as one of these values:

- `'numbered'` — The row headings are sequential numbers that start at `1`.
- Cell array of character vectors or categorical array — Each element of the array becomes the name of a row. Row names are restricted to one line of text. If you specify a 1-by-`n` cell array, MATLAB stores and returns the value as an `n`-by-1 cell array. If you specify an `m`-by-`n` array, MATLAB reshapes the array into a column vector.
- Empty cell array (`{}`) — The table has no row headings.
- Empty matrix (`[]`) — The table has no row headings

If the number of rows in the `Data` property array does not match the number of elements in the `RowName` array, then the number of rows in the resulting table reflects the number of rows in the `Data` property.

If you specify the `Data` property as a table array, then the default row names match the `RowNames` property of the table array. Changing the `RowName` property of the table UI component updates the UI, but it will not update the row names in the table array.

Example: `uit = uitable(uifigure,'RowName',{'Name';'Number'},'Data',{'Bob';5})`

Example: `uit = uitable(uifigure,'RowName',{'Name';[]},'Data',{'Bob';5})`

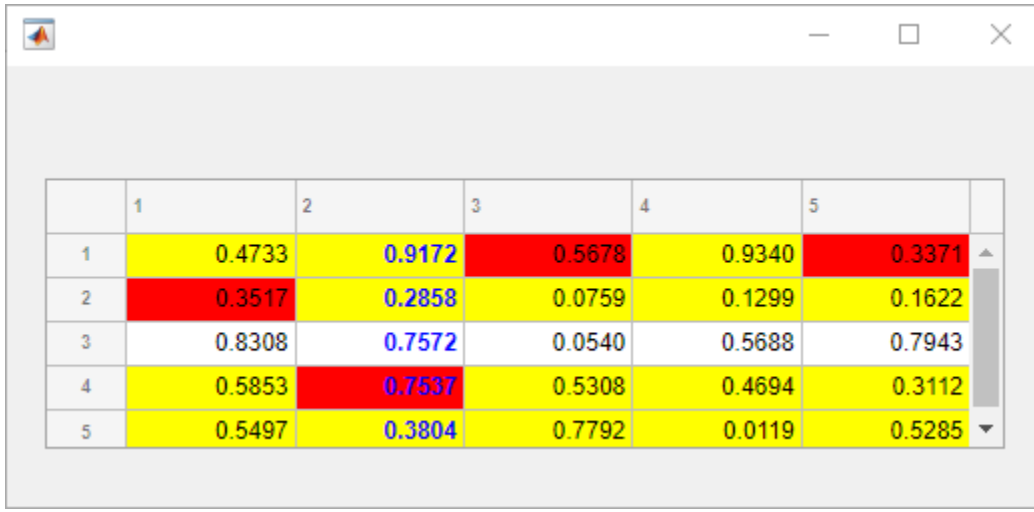
`StyleConfigurations` — Configurations of added styles
empty n-by-3 table array (default) | n-by-3 table array

This property is read-only.

Configuration of added styles, returned as an `n`-by-3 table array. Each row of the table array corresponds to a style that is currently applied to the table UI component. Styles that are added consecutively are given a style order number of `n+1`. The `Target` and `TargetIndex` columns specify the part of the table UI component that the style was added to. The `Style` column specifies the style class name.

Use this property if you want to remove a style from the table UI component using the `removeStyle` function. For instance, in this example, three styles are added to a table UI component.

```
s1 = uistyle('BackgroundColor','yellow');
s2 = uistyle('BackgroundColor','red');
s3 = uistyle('FontColor','b','FontWeight','bold');

fig = uifigure;
fig.Position = [100 100 520 220];
uit = uitable(fig);
uit.Data = rand(5);
uit.Position = [20 30 480 135];

addStyle(uit,s1,'row',[1 2 4 5]);
addStyle(uit,s2,'cell',[2 1; 4 2; 1 3; 1 5])
addStyle(uit,s3,'column',2);
```



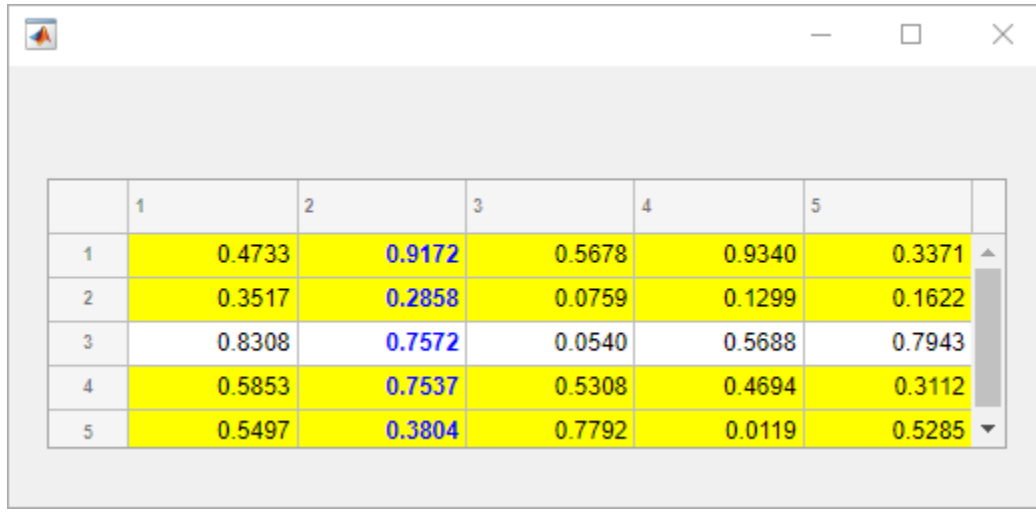
When you query `uit.StyleConfigurations`, a 3-by-3 table array is returned. The row style was added to the table UI component first, so it is style order number `1`. The `TargetIndex` value for the row style, `{1×4 double}`, indicates that four rows were specified when the style was added. Similarly, the second style was added to four cells in the table. The third style was added to the second column.

```
uit.StyleConfigurations
```

```
ans =

  3×3 table

        Target    TargetIndex                Style
        ______    ___________    ____________________________

    1   row       {1×4 double}   [1×1 matlab.ui.style.Style]
    2   cell      {4×2 double}   [1×1 matlab.ui.style.Style]
    3   column    {[       2]}   [1×1 matlab.ui.style.Style]
```

Remove the second style that was added to the table, by specifying style order number `2`. Notice how the table UI component updates.

```
removeStyle(uit,2)
```



	1	2	3	4	5
1	0.4733	0.9172	0.5678	0.9340	0.3371
2	0.3517	0.2858	0.0759	0.1299	0.1622
3	0.8308	0.7572	0.0540	0.5688	0.7943
4	0.5853	0.7537	0.5308	0.4694	0.3112
5	0.5497	0.3804	0.7792	0.0119	0.5285

Font

`FontName` — Font name

system-supported font name

Font name, specified as a system-supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

`FontSize` — Font size

positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

`FontWeight` — Font weight

`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font

- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. For fonts that do not, specifying `'bold'` results in the normal font weight.

`FontAngle` — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Not all fonts have an italic font angle. For fonts that do not, specifying `'italic'` results in the normal font angle.

`FontUnits` — Font size units
`'pixels'`

Font size units, specified as `'pixels'`.

Interactivity

`Visible` — Table visibility
`'on'` (default) | `'off'`

Table visibility, specified as `'on'` or `'off'`. When `Visible` is `'off'`, the table is not visible, but you can query and set its properties.

To make your app start faster, set the `Visible` property of all components that are not initially displayed to `'off'`.

`Enable` — Operational state of table
`'on'` (default) | `'off'`

Operational state of table, specified as `'on'` or `'off'`. The `Enable` property controls whether the table responds to user interaction. The are two possible values:

- `'on'` – The table is operational.
- `'off'` – The table appears grayed-out and does not respond to user interaction.

`Tooltip` — Tooltip
`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the

pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Color and Styling

`ForegroundColor` — Cell text color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Cell text color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table. When you set cell text color using the `ForegroundColor` property it applies to all the cells in the table UI component.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`BackgroundColor` — Table background color

`[1 1 1; 0.94 0.94 0.94]` (default) | RGB triplet | m-by-3 matrix of RGB triplets

Table background color, specified as an RGB triplet or an `m`-by-3 matrix of RGB triplets. An RGB triplet is a row vector that specifies the intensities of the red, green, and blue components of the color. The intensities must be in the range, `[0,1]`. Color names are not valid.

Specify an `m`-by-3 matrix when you want the shading of the table rows to follow a repeating pattern of `m` different colors. Each row of the matrix must be an RGB triplet. MATLAB uses the rows of the matrix when the `RowStriping` property is `'on'`. The table background is not striped unless both `RowStriping` is `'on'` and `BackgroundColor` is an `m`-by-3 matrix.

Example: `uit = uitable(uifigure,'Data',rand(10,3),'BackgroundColor',[0.85 0.85 1])`

Example: `uit = uitable(uifigure,'Data',rand(10,3),'BackgroundColor',[1 1 1 ;0.85 0.85 1])`

The following table lists the RGB triplets for certain colors.

Color	RGB Triplet
Yellow	`[1 1 0]`
Magenta	`[1 0 1]`

Color	RGB Triplet
Cyan	`[0 1 1]`
Red	`[1 0 0]`
Green	`[0 1 0]`
Blue	`[0 0 1]`
White	`[1 1 1]`
Black	`[0 0 0]`

`RowStriping` — Alternate row shading
`'on'` (default) | `'off'`

Alternate row shading, specified as `'on'` or `'off'`. This property controls the shading pattern of the table rows.

When the `RowStriping` value is set to `'on'`, the `BackgroundColor` matrix specifies the row colors to display in a repeating pattern. If the `BackgroundColor` matrix has only one row, then the shading is the same in all table rows.

When `RowStriping` is set to `'off'`, then the first color in the `BackgroundColor` matrix defines the shading for all rows in the table.

Position

`Position` — Location and size of table
`[left bottom width height]`

Location and size of the table, specified as a four-element vector of the form `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the table
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the table
`width`	Distance between the right and left outer edges of the table

Element	Description
`height`	Distance between the top and bottom outer edges of the table

All measurements are in units specified by the `Units` property.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

`InnerPosition` — Location and size of table
`[left bottom width height]`

Location and size of the table, specified as a four-element vector of the form `[left bottom width height]`. All measurements are in units specified by the `Units` property.

This property value is identical to the `Position` and `OuterPosition` property values.

`OuterPosition` — Location and size of table
`[left bottom width height]`

Location and size of the table, specified as a four-element vector of the form `[left bottom width height]`. All measurements are in units specified by the `Units` property.

This property value is identical to the `Position` and `InnerPosition` property values.

`Units` — Units of measurement
`'pixels'` (default)

Units of measurement, specified as `'pixels'`. MATLAB measures all units from the lower left corner of the parent object.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid

layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a table UI component in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
uit = uitable(g,'Data',rand(10,3));
uit.Layout.Row = 3;
uit.Layout.Column = 2;
```

To make the table span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this table spans columns `2` through `3`:

```
uit.Layout.Column = [2 3];
```

Callbacks

`CellEditCallback` — Cell edit callback function

function handle | cell array | character vector

Cell edit callback function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

Use this callback function to perform calculations or validate input when the app user changes the contents of a table cell.

This callback function can access specific information about the user's interaction with the cell (such as the cell indices). MATLAB passes this information in a `CellEditData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.Indices` returns the indices of the selected cell. The `CellEditData` object is not available to callback functions specified as character vectors.

The following table describes properties of the `CellEditData` object.

Property	Description
`Indices`	This is a 1-by-2 array containing the row and column indices of the cell the user edited in the running app. When a column is sorted, `Indices` returns the original 1-by-2 array of a cell before it was sorted—`DisplayIndices` returns the new location of the edited cell that displays visually in the sorted table.
`DisplayIndices`	This is a 1-by-2 array containing the row and column indices corresponding to the location of the edited cell in the display of the sorted table. If a user does not sort columns, then `DisplayIndices` has the same content as the `Indices` property.
`PreviousData`	This is the previous cell data. The default is an empty matrix, `[]`.
`EditData`	This is the user-entered value.
`NewData`	This is the value that MATLAB wrote to the `Data` property array. The `NewData` property is empty if MATLAB detects an error in the user-entered data.
`Error`	This is the error message returned if MATLAB detects an error in the user-entered data. The `Error` property is empty when MATLAB successfully writes the value to the `Data` property. If the `Error` property is not empty, then the `CellEditCallback` can display the message, or it can attempt to fix the problem.
`Source`	Component executing the callback.
`EventName`	`'CellEdit'`.

When the user edits a table cell, MATLAB performs these steps:

1. Tries to store the new value into the `Data` property of the table
2. Calls the `CellEditCallback` function (if it exists)

If the value results in an error and there is no `CellEditCallback` function, then the cell data reverts to its previous value and no error displays.

For more information about writing callbacks, see “Write Callbacks in App Designer”.

`CellSelectionCallback` — Cell selection callback function
function handle | cell array | character vector

Cell selection callback function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback function executes when the user selects cells. A single data cell (not a row or column heading) can be selected by clicking it or navigating to it with an arrow key. Multiple cells can be selected with any of the actions described in the tables. When selecting multiple cells, use different actions depending on whether they are contiguous or discontiguous.

Contiguous Selection Options
Click a cell and drag.
Click one cell, then **Shift+click** another cell to select all cells in between.
Click one cell, then **Shift+arrow key** to add contiguous cells.
Click on a row or column header to select the whole row or column.
Click a cell, then **Ctrl+space** to select all the cells in that column (regardless of whether all the cells are in view).
Click a cell, then **Shift+space** to select all the cells in that row (regardless of whether all the cells are in view).
Select a row or column. Then **Shift+click** another row or column header to select all the rows or columns in between.
Click one cell, then **Shift+PgUp** or **Shift+PgDn** to select all visible cells above or below that cell.
Press **Ctrl+A** to select all of the cells in the table

Discontiguous Selection Options
Click and drag to select a contiguous group of cells (or select a single cell). Then, **Ctrl +click** to focus another cell, and finally **Shift+click** to select the cell that is in focus, or **Shift+click** and drag to select a range of cells that spread from it. (Pressing **Shift +arrow key** also selects more than one cell.)

Discontiguous Selection Options
Ctrl+click on a row or column header to select the entire row or column. Then repeat to select another non-adjacent row or column.

When a cell is focused using **Ctrl+click**, the cell outline turns blue (

) . When a cell is selected using click or **Shift+click**, the cell fill-color changes to blue (

).

This callback function can access specific information about the user's interaction with the cell (such as the cell indices). MATLAB passes this information in a `CellSelectionChangeData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.Indices` returns the indices of the selected cell. The `CellSelectionChangeData` object is not available to callback functions specified as character vectors.

The following table describes properties of the `CellSelectionChangeData` object.

Property	Description
`Indices`	This is an n-by-2 array containing the row and column indices of the cell the user selected in the running app. For multicolumn variables in a table array, the column indices refer to the whole column. When a column is sorted, `Indices` returns the original 1-by-2 array of a cell before it was sorted—`DisplayIndices` returns the new location of the selected cell that displays visually in the sorted table.
`DisplayIndices`	This is an n-by-2 array containing the row and column indices corresponding to the location of the selected cell in the display of the sorted table. If a user does not sort the table, then `DisplayIndices` has the same content as the `Indices` property.
`Source`	Component executing the callback.
`EventName`	`'CellSelection'`.

For more information about writing callbacks, see "Write Callbacks in App Designer".

`DisplayDataChangedFcn` — Callback when display data changes

`''` (default) | function handle | cell array | character vector

Callback when display data changes, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when the `DisplayData` changes, because a user either edits a cell or sorts columns of a table.

Use this callback if you want information about user interactions that caused the `DisplayData` to change. If you need specific information about edited cells, create a `CellEditCallback` instead.

This callback function can access specific information about whether columns are sorted or cells are edited. MATLAB passes this information in a `DisplayDataChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.InteractionColumn` returns the index of the column that was interacted with in `Data`. The `DisplayDataChangedData` object is not available to callback functions specified as character vectors.

Property	Description
`DisplayRowName`	Cell array of `RowName` property values in the sorted display. `DisplayRowName` will always be a permutation of the original `RowName` property.
`DisplayColumnName`	Cell array of `ColumnName` property values in the sorted display. `DisplayColumnName` will always be a permutation of the original `ColumnName` property.
`Interaction`	`'sort'` or `'edit'`
`InteractionColumn`	Index of modified column in `Data`.
`InteractionDisplayColumn`	Index of modified column in `DisplayData`.

Property	Description
`InteractionVariable`	`VariableNames` property of the modified column for table array data. If `Data` contains a data type other than a table array, then `InteractionVariable` returns an empty character array.
`Source`	Component executing the callback.
`EventName`	`'DisplayDataChanged'`.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.

  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`Children` — Children of table
empty array

Children of table, returned as an empty array. Table objects have no children, so setting this property has no effect.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If

you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uitable'`

This property is read-only.

Type of graphics object, returned as `'uitable'`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`addStyle` | `appdesigner` | `removeStyle` | `uistyle` | `uitable`

Topics

“Table Array Data Types in App Designer Apps”
“Programmatic App that Displays a Table”

Introduced in R2016b

uitoggletool

Create toggle button on toolbar

Note

Use this function only with GUIDE, or with figures created using the `figure` function.

Syntax

```
t = uitoggletool
t = uitoggletool(Name,Value,...)
t = uitoggletool(parent)
t = uitoggletool(parent,Name,Value,...)
```

Description

`t = uitoggletool` creates a uitoggletool in the current figure's uitoolbar and returns the uitoggletool object, `t`. If there is no uitoolbar available, then MATLAB creates a new uitoolbar in the current figure to serve as the parent. Similarly, if there no figure is available, then MATLAB creates a new figure with a uitoolbar.

`t = uitoggletool(Name,Value,...)` creates a uitoggletool and specifies one or more uitoggletool property names and corresponding values. Use this syntax to override the default uitoggletool properties.

`t = uitoggletool(parent)` creates a uitoggletool and designates a specific parent object. The `parent` argument must be a uitoolbar object.

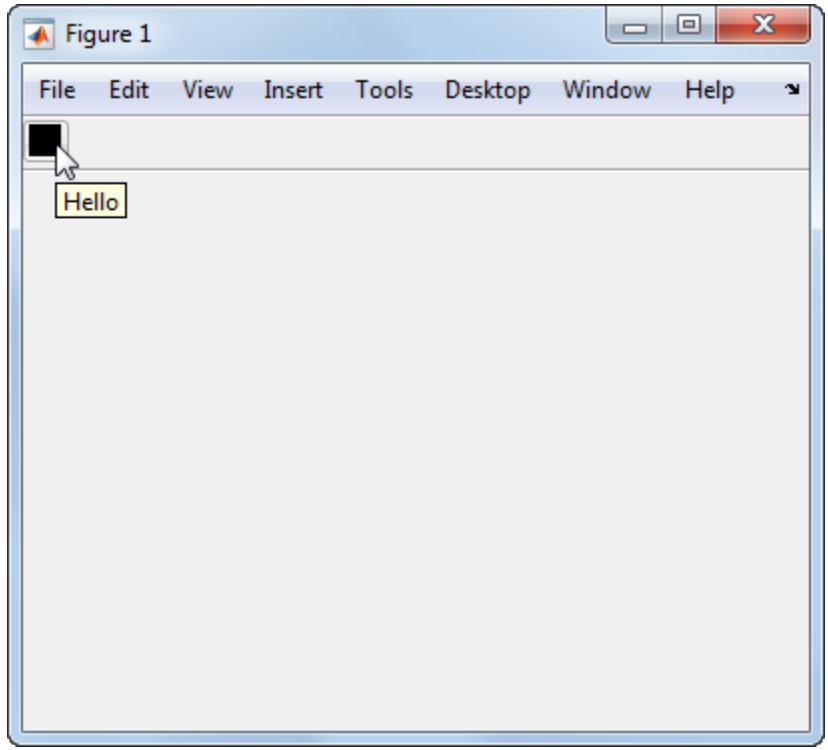
`t = uitoggletool(parent,Name,Value,...)` creates a uitoggletool with a specific parent and one or more uitoggletool properties.

A uitoggletool is a toggle button that appears in the figure's tool bar. The button has no icon, but its borders highlight when the user hovers over it with the mouse. You can create a button icon by setting the uitoggletool's `CData` property.

Examples

This example creates a `uitoolbar` and places a `uitoggletool` inside it.

```
f = figure('ToolBar','none');
tb = uitoolbar(f);
img = zeros(16,16,3);
t = uitoggletool(tb,'CData',img,'TooltipString','Hello');
```



The `CData` property is set to a 16-by-16-by-3 array in which all elements are `0`. The zero values make the uitoggletool appear black.

Tips

- Uitoolbars (and their child uitoggletools) do not appear in figures whose `WindowStyle` property is set to `'Modal'`. If a figure containing a uitoolbar is changed to `'Modal'`, the uitoolbar still exists in the `Children` property of the figure. However, the uitoolbar does not display while `WindowStyle` is set to `'Modal'`.

Alternatives

You can create toolbars with toggle tools using GUIDE.

See Also

Uitoggletool | `uipushtool` | `uitoolbar`

Topics

"Access Property Values"
"Create Toolbars for Programmatic Apps"

Introduced before R2006a

ToggleTool Properties

Control appearance and behavior of toggle tool

Note

The properties listed here are valid for `ToggleTool` objects in GUIDE or in apps created with the `figure` function.

Description

Toggle tools are toggle buttons that appear on the tool bar at the top of the a figure. The `uitoggletool` function creates a toggle button on a tool bar and sets any required properties before displaying it. By changing property values, you can modify certain aspects of its appearance and behavior. Use dot notation to refer to a specific object and property.

```
t = uitoggletool;
t.Separator = 'on';
```

Properties

Button Appearance

`State` — Toggle button state

`'off'` (default) | `'on'`

Toggle button state, specified as `'off'` or `'on'`. When this property value is `'on'`, the toggle button appears in the down (depressed) position. When this property value is `'off'`, the toggle button appears in the up position. Changing the state causes the appropriate `OnCallback` or `OffCallback` function to execute.

`CData` — Optional icon

3-D array of truecolor RGB values

Optional icon, specified as a 3-D array of truecolor RGB values. The values in the array can be:

- Double-precision values between `0.0` and `1.0`
- `uint8` values between `0` and `255`

The length of the array's first and second dimensions must be less than or equal to 16. Otherwise, it might be clipped or distorted when it displays.

Data Types: `double` | `uint8`

`Separator` — Separator line mode
`'off'` (default) | `'on'`

Separator line mode, specified as `'off'` or `'on'`. Setting this property to `'on'` draws a dividing line to left of a tool in the tool bar.

Interactivity

`Visible` — Component visibility
`'on'` (default) | `'off'`

Component visibility, specified as `'on'` or `'off'`. When the `Visible` property is set to `'off'`, the component is not visible in the UI, but you can query and set its properties.

`Enable` — Operational state of tool
`'on'` (default) | `'off'`

Operational state of tool, specified as `'on'` or `'off'`. The `Enable` property controls whether the tool responds to button clicks. The are two possible values:

- `'on'` – The tool is operational.
- `'off'` – The tool is not operational and appears grayed-out.

The value of the `Enable` property and the type of button click determine the response.

Enable Value	Response to Left-Click	Response to Right-Click
`'on'`	**1** The `OnCallback` or `OffCallback` function execute, depending on the current state of the tool. **2** The `ClickedCallback` function executes.	The tool is not operational. No callback executes.

Enable Value	Response to Left-Click	Response to Right-Click
`'off'`	The tool is not operational. No callback executes.	The tool is not operational. No callback executes.

`Tooltip` — Tooltip
character vector | string scalar | categorical array

Tooltip, specified as a character vector, string scalar, or categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip does not display when the component is disabled. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

To create multiple lines of text, use the `sprintf` function to insert newline characters (`'\n'`) in your text. For example:

```
txt = sprintf('Line 1\nLine 2');
```

Then set the `Tooltip` property to the value returned by `sprintf`.

`UIContextMenu` — Context menu (not recommended)
empty `GraphicsPlaceholder` array (default) | uicontextmenu handle

This property has no effect on `ToggleTool` objects.

`TooltipString` — Tooltip (not recommended)
character vector | string scalar | categorical array

Tooltip, specified as a character vector, string scalar, or categorical array. The tooltip displays when the user hovers the mouse pointer over the component in the app. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Note The `TooltipString` property is not recommended starting in R2018b. Use the `Tooltip` property instead.

Callbacks

`ClickedCallback` — Tool clicked callback
`''` (default) | function handle | cell array | character vector

Tool clicked callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback function executes after the `OnCallback` function or `OffCallback` function executes (depending on the state of the button).

For more information about specifying callback functions as function handles, cell arrays, or character vectors, see “How to Specify Callback Property Values”.

Data Types: `function_handle` | `cell` | `char`

`OnCallback` — Callback to execute when user turns on toggle button
`''` (default) | function handle | cell array | character vector

Callback to execute when user turns on the toggle button, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying callback functions as function handles, cell arrays, or character vectors, see “How to Specify Callback Property Values”.

Data Types: `function_handle` | `cell` | `char`

`OffCallback` — Callback to execute when user turns off the toggle button
`''` (default) | function handle | cell array | character vector

Callback to execute when user turns off toggle button, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying callback functions as function handles, cell arrays, or character vectors, see “How to Specify Callback Property Values”.

Data Types: `function_handle` | `cell` | `char`

`CreateFcn` — Component creation function

`''` (default) | function handle | cell array | character vector

Component creation function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

This property specifies a callback function to execute when MATLAB creates the component. MATLAB initializes all component property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Use the `gcbo` function in your `CreateFcn` code to get the component object that is being created.

Setting the `CreateFcn` property on an existing component object has no effect.

`DeleteFcn` — Component deletion function

`''` (default) | function handle | cell array | character vector

Component deletion function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

The `DeleteFcn` property specifies a callback function to execute when MATLAB deletes the component (for example, when the user closes the window). MATLAB executes the `DeleteFcn` callback before destroying the properties of the component object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

Use the `gcbo` function in your `DeleteFcn` code to get the component object that is being deleted.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed:

- A value of `'on'` allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- A value of `'off'` blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn`, or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.
- MATLAB does not save the state of properties or the display when an interruption occurs. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

See “Interrupt Callback Execution” for an example that shows how the `Interruptible` and `BusyAction` properties affect the behavior of a program.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` (default) or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

The `BusyAction` property of the source of the interrupting callback determines how MATLAB handles its execution. The `BusyAction` property has these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Do not execute the interrupting callback.

Whenever MATLAB invokes a callback, that callback always attempts to interrupt an executing callback. The `Interruptible` property of the object whose callback is running determines if interruption is allowed. If `Interruptible` is set to:

- `on` — Interruption occurs at the next point where MATLAB processes the queue. This is the default.

- `off` — The `BusyAction` property (of the object owning the interrupting callback) determines if MATLAB enqueues or ignores the interrupting callback.

See “Interrupt Callback Execution” for an example that shows how the `BusyAction` and `Interruptible` properties affect the behavior of a program.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

`HitTest` — Ability to become current object (not recommended)
`'on'` (default) | `'off'`

This property has no effect on `ToggleTool` objects.

Parent/Child

`Parent` — Parent object
`Toolbar` object

Parent object, specified as a `Toolbar` object. Use this property to specify the parent tool bar when creating a tool or to move an existing tool to a different tool bar.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'callback'` | `'off'`

Visibility of object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object handle in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. The `HandleVisibility` property also controls the visibility of the object’s handle in the parent figure's `CurrentObject` property. Handles are still valid even if they are not

visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object handle is always visible.
`'callback'`	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.

Set the graphics root `ShowHiddenHandles` property to `'on'` to make all handles visible, regardless of their `HandleVisibility` value. This setting has no effect on their `HandleVisibility` values.

Identifiers

Type — Type of graphics object

`'uitoggletool'`

This property is read-only.

Type of graphics object, returned as `'uitoggletool'`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

UserData — User data

`[]` (default) | array

User data, specified as any array. Specifying `UserData` can be useful for sharing data within apps. See “Share Data Among Callbacks” for more information.

See Also

`uitoggletool` | `uitoolbar`

Topics

“Access Property Values”
“Default Property Values”

Introduced before R2006a

uitoolbar

Create toolbar on figure

Note

Use this function only with GUIDE, or with figures created using the `figure` function.

Syntax

```
t = uitoolbar
t = uitoolbar(Name,Value,...)
t = uitoolbar(parent)
t = uitoolbar(parent,Name,Value,...)
```

Description

`t = uitoolbar` creates a toolbar in an existing figure and returns the `Toolbar` object, `t`. If a figure does not exist, then MATLAB calls the `figure` function to create one.

`t = uitoolbar(Name,Value,...)` creates a toolbar and specifies one or more `Toolbar` property name-value pair arguments. Use this syntax to override the default toolbar properties.

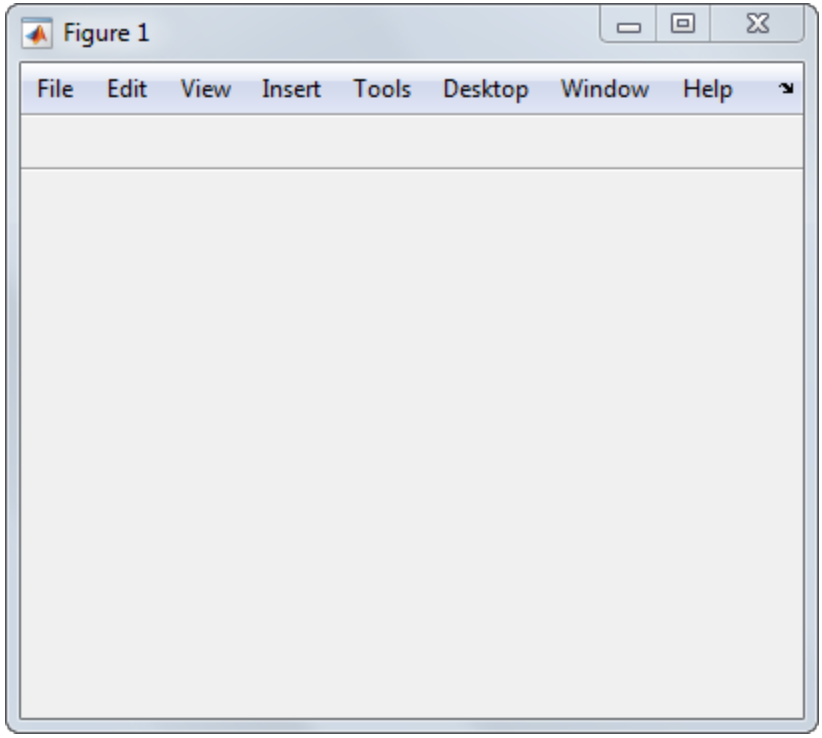
`t = uitoolbar(parent)` creates a toolbar in the specified parent figure.

`t = uitoolbar(parent,Name,Value,...)` specifies a parent figure for the toolbar and one or more name-value pair arguments.

Examples

This example creates a figure containing an empty toolbar.

```
f = figure('ToolBar','none')
t = uitoolbar(f)
```



Tips

`Toolbar` objects do not appear in figures whose `WindowStyle` property is set to `'Modal'`. If a figure containing a toolbar child is changed to `'Modal'`, the toolbar child still exists in the `Children` property of the figure. However, the toolbar does not display while `WindowStyle` is set to `'Modal'`.

See Also

Uitoolbar | `figure` | `uipushtool` | `uitoggletool`

Topics

"Create Toolbars for Programmatic Apps"
"Create Toolbars for GUIDE UIs"
"Access Property Values"

Introduced before R2006a

Toolbar Properties

Control appearance and behavior of tool bar

Note

The properties listed here are valid for `Toolbar` objects in GUIDE or in apps created with the `figure` function.

Description

A tool bar is a container for a horizontal list of buttons at the top of a figure window. The `uitoolbar` function creates a tool bar in a figure and sets any required properties before displaying it. By changing property values, you can modify the appearance and behavior of a tool bar. Use dot notation to refer to a particular object and property.

```
t = uitoolbar;
t.Visible = 'off';
```

Properties

Interactivity

`Visible` — Component visibility
`'on'` (default) | `'off'`

Component visibility, specified as `'on'` or `'off'`. When the `Visible` property is set to `'off'`, the component is not visible in the UI, but you can query and set its properties.

`UIContextMenu` — Context menu (not recommended)
empty `GraphicsPlaceholder` array (default) | uicontextmenu handle

This property has no effect on `Toolbar` objects.

Callbacks

`CreateFcn` — Component creation function
`''` (default) | function handle | cell array | character vector

Component creation function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

This property specifies a callback function to execute when MATLAB creates the component. MATLAB initializes all component property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Use the `gcbo` function in your `CreateFcn` code to get the component object that is being created.

Setting the `CreateFcn` property on an existing component object has no effect.

`DeleteFcn` — Component deletion function

`''` (default) | function handle | cell array | character vector

Component deletion function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “How to Specify Callback Property Values”.

The `DeleteFcn` property specifies a callback function to execute when MATLAB deletes the component (for example, when the user closes the window). MATLAB executes the `DeleteFcn` callback before destroying the properties of the component object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

Use the `gcbo` function in your `DeleteFcn` code to get the component object that is being deleted.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed:

- A value of `'on'` allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause`.
  - If the running callback contains one of these commands, then MATLAB stops the execution of the callback at this point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of these commands, then MATLAB finishes executing the callback without interruption.
- A value of `'off'` blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn`, or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.

- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.
- MATLAB does not save the state of properties or the display when an interruption occurs. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

See “Interrupt Callback Execution” for an example that shows how the `Interruptible` and `BusyAction` properties affect the behavior of a program.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing specified as `'queue'` (default) or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

The `BusyAction` property of the source of the interrupting callback determines how MATLAB handles its execution. The `BusyAction` property has these values:

- `'queue'` — Put the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Do not execute the interrupting callback.

Whenever MATLAB invokes a callback, that callback always attempts to interrupt an executing callback. The `Interruptible` property of the object whose callback is running determines if interruption is allowed. If `Interruptible` is set to:

- `on` — Interruption occurs at the next point where MATLAB processes the queue. This is the default.
- `off` — The `BusyAction` property (of the object owning the interrupting callback) determines if MATLAB enqueues or ignores the interrupting callback.

See “Interrupt Callback Execution” for an example that shows how the `BusyAction` and `Interruptible` properties affect the behavior of a program.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

`HitTest` — Ability to become current object (not recommended)
`'on'` (default) | `'off'`

This property has no effect on `Toolbar` objects.

Parent/Child

`Parent` — Parent object
figure

Parent object, specified as a figure. You can move a `Toolbar` to a different figure by setting this property to the target `Figure` object.

`Children` — `Toolbar` children
empty `GraphicsPlaceholder` array (default) | 1-D array of component objects

`Toolbar` children, returned as an empty `GraphicsPlaceholder` or a 1-D array of component objects. The children of `Toolbar` objects are `PushTool` and `ToggleTool` objects.

You cannot add or remove children using the `Children` property. Use this property to view the list of children or to reorder the children. The order of the children in this array reflects the order of the components displayed in the menu.

To add a child to this list, set the `Parent` property of the child component to be the `Toolbar` object.

Objects with the `HandleVisibility` property set to `'off'` do not list in the `Children` property.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'callback'` | `'off'`

Visibility of object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object handle in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. The `HandleVisibility` property also controls the visibility of the object's handle in the parent figure's `CurrentObject` property. Handles are still valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object handle is always visible.
`'callback'`	The object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object handle is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.

Set the graphics root `ShowHiddenHandles` property to `'on'` to make all handles visible, regardless of their `HandleVisibility` value. This setting has no effect on their `HandleVisibility` values.

Identifiers

Type — Type of graphics object

`'uitoolbar'`

This property is read-only.

Type of graphics object, returned as `'uitoolbar'`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object

elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[ ]` (default) | array

User data, specified as any array. Specifying `UserData` can be useful for sharing data within apps. See “Share Data Among Callbacks” for more information.

See Also

`uitoolbar`

Topics

“Access Property Values”
“Default Property Values”

Introduced before R2006a

uiwait

Block program execution and wait to resume

Syntax

```
uiwait
uiwait(h)
uiwait(h,timeout)
```

Description

`uiwait` blocks execution until `uiresume` is called or the current figure is deleted. This syntax is the same as `uiwait(gcf)`.

`uiwait(h)` blocks execution until `uiresume` is called or the figure `h` is deleted. The figure can be one that is created with `figure` or `uifigure` function. Only figures created with the `figure` function are returned by `gcf`.

`uiwait(h,timeout)` blocks execution until `uiresume` is called, the figure `h` is deleted, or `timeout` seconds elapse. The minimum value of `timeout` is 1. If `uiwait` receives a smaller value, it issues a warning and uses a 1 second `timeout`.

Examples

This example creates a UI with a **Continue** push button. The example calls `uiwait` to block MATLAB execution until `uiresume` is called. This happens when the user clicks the **Continue** push button because the push button's `Callback`, which responds to the click, calls `uiresume`.

```
f = figure;
h = uicontrol('Position',[20 20 200 40],'String','Continue',...
              'Callback','uiresume(gcbf)');
disp('This will print immediately');
uiwait(gcf);
```

```
disp('This will print after you click Continue');
close(f);
```

`gcbf` is the handle of the figure that contains the object whose callback is executing.

Tips

The `uiwait` and `uiresume` functions block and resume MATLAB and Simulink program execution. `uiwait` also blocks the execution of Simulink models. The functions `pause` (with no argument) and `waitfor` also block execution in this manner. `uiwait` is a convenient way to use the `waitfor` command. You typically use it in conjunction with a dialog box. It provides a way to block the execution of the MATLAB program that created the dialog, until the user responds to the dialog box. When used in conjunction with a modal dialog, `uiwait` can block the execution of the program file *and* restrict user interaction to the dialog only.

See Also

`dialog` | `figure` | `uicontrol` | `uifigure` | `uimenu` | `uiresume` | `waitfor`

Introduced before R2006a

matlab.uitest.TestCase class

Package: `matlab.uitest`

`TestCase` to write tests with app testing framework

Description

Use the `matlab.uitest.TestCase` class to write tests that use the app testing framework. The `matlab.uitest.TestCase` derives from the `matlab.unittest.TestCase` class.

To avoid user interference with the app during testing, the framework locks new figure instances. The components of locked figures only respond to programmatic gestures via `TestCase` methods such as `press` and `choose`.

Construction

The testing framework constructs the `matlab.uitest.TestCase` instances.

Methods

choose	Perform choose gesture on UI component
drag	Perform drag gesture on UI component
forInteractiveUse	Create a `TestCase` object for interactive use
hover	Perform hover gesture on UI component
press	Perform press gesture on UI component
type	Type in UI component

Inherited Methods

addTeardown	Dynamically add teardown routine
applyFixture	Use fixture with `TestCase`
forInteractiveUse	Create TestCase for interactive use
getSharedTestFixtures	Provide access to shared test fixtures
log	Record diagnostic information
onFailure	Dynamically add diagnostics for test failures
run	Run `TestCase` test

Also, the `TestCase` class inherits methods from these classes:

matlab.unittest.qualifications.Assertable	Qualification to validate preconditions of a test
matlab.unittest.qualifications.Assumable	Qualification to filter test content
matlab.unittest.qualifications.FatalAssertable	Qualification to abort test execution
matlab.unittest.qualifications.Verifiable	Qualification to produce soft failure conditions

Attributes

`Abstract`	`true`
`HandleCompatibile`	`true`

For information on class attributes, see “Class Attributes”.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

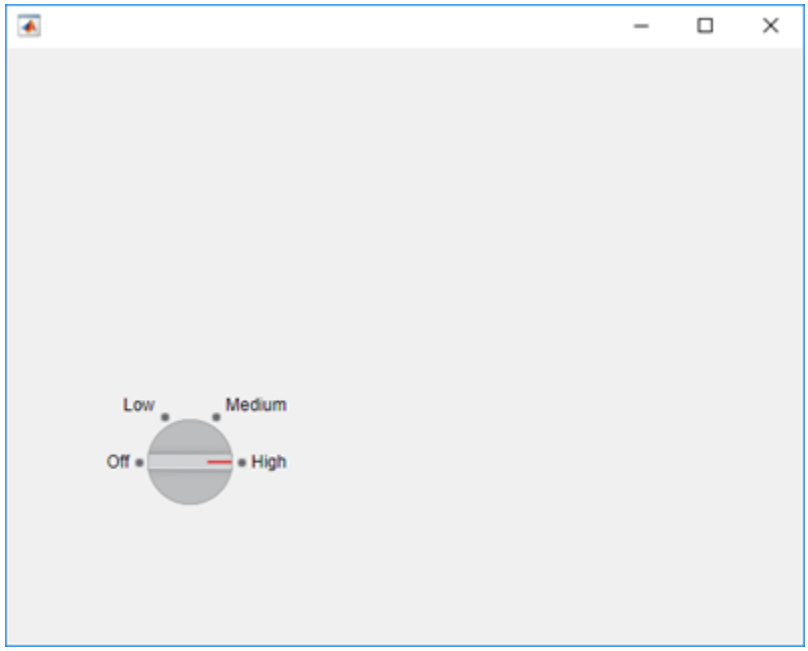
Choose Value on Discrete Knob

Create a discrete knob.

```
knob = uiknob('discrete');
```

Create an interactive test case and choose the 'High' knob value. An animated blue dot performs the programmatic choose gesture.

```
tc = matlab.uitest.TestCase.forInteractiveUse;
tc.choose(knob,'High')
```



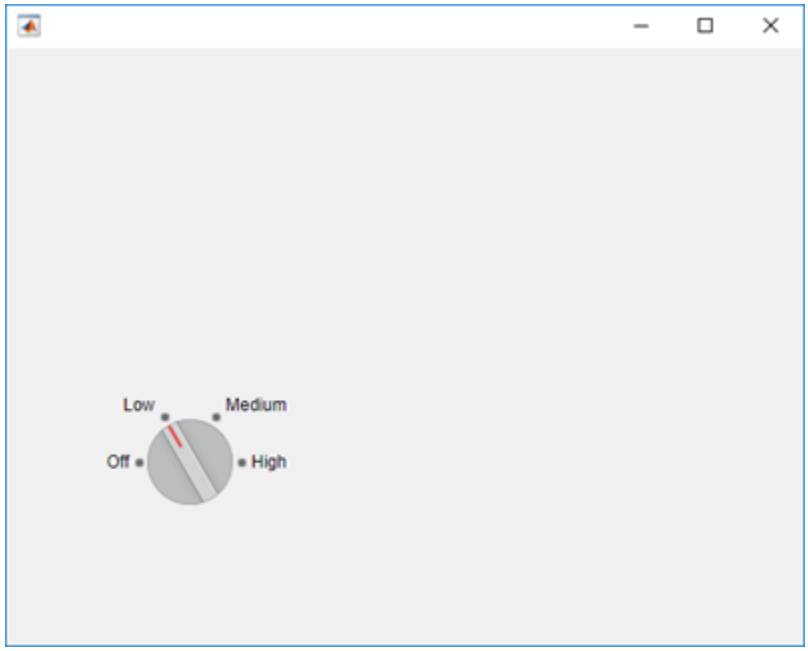
View the value of the `Items` property on the knob.

```
knob.Items
```

```
ans =

  1×4 cell array

    {'Off'}    {'Low'}    {'Medium'}    {'High'}
```

Choose the 'Low' knob value by index. The knob moves from `'High'` to `'Low'`.

```
tc.choose(knob,2)
```



Press Button and Verify Value Change

Create a state button.

```
b = uibutton('state');
```

Create an interactive test case and verify that the value of the state button is `false`.

```
tc = matlab.uitest.TestCase.forInteractiveUse;
tc.verifyFalse(b.Value)
```

```
Verification passed.
```

Press the button and verify the state changes to `true`. A blue dot representing the programmatic push gesture appears and then disappears on the button.

```
tc.press(b)
tc.verifyTrue(b.Value)
```

```
Verification passed.
```

Drag Slider and Verify Stop Value

Create a slider with a minimum value of -237, a maximum value of 237, and a starting value of 7.

```
slider = uislider('Limits',[-237 237],'Value',7);
```

Create an interactive test case and verify the initial value of the slider.

```
tc = matlab.uitest.TestCase.forInteractiveUse;
tc.verifyEqual(slider.Value,7)
```

```
Verification passed.
```

Drag the slider between two values and verify the final value. Since the framework mimics a user manipulating the component to an arbitrarily precisioned value, it is a best practice to use a tolerance to compare the actual and expected slider values.

```
val = 26.75;
tc.drag(slider,-val,val)
tc.verifyEqual(slider.Value,val,'AbsTol',0.1)
```

```
Verification passed.
```

See Also

`matlab.unittest.TestCase`

Topics

"Overview of App Testing Framework"
"Write Test for App"
"Write Test That Uses App Testing and Mocking Frameworks"

Introduced in R2018a

choose

Class: `matlab.uitest.TestCase`
Package: `matlab.uitest`

Perform choose gesture on UI component

Syntax

```
choose(testcase,comp,option)
choose(testcase,compNoOpts)
```

Description

`choose(testcase,comp,option)` performs a choose gesture on the specified item on the UI component `comp`.

`choose(testcase,compNoOpts)` performs a choose gesture on a UI component that does not require additional information, such as a tab or a tree node. For example, use this syntax to choose a specific tab, but use the previous syntax to choose a particular tab from a tab group.

Input Arguments

`testcase` — Instance of test case
`matlab.uitest.TestCase` object

Instance of the test case, specified as a `matlab.uitest.TestCase` object.

`comp` — Component to choose
UI component object

Component to choose during test, specified as a UI component object that supports a choose gesture. Components that support choose gestures include check boxes, knobs, switches, and drop-down lists.

Supported Component	Typical Creation Function

Button	`uiradiobutton`
	
	`uitogglebutton`

Check Box	`uicheckbox`
Drop Down	`uidropdown`
List Box	`uilistbox`
Slider	`uislider`
State Button	`uibutton('state')`
Button Group	`uibuttongroup`
Tab Group	`uitabgroup`
Knob, Discrete Knob	`uiknob`
Switch, Rocker Switch, Toggle Switch	`uiswitch`

Data Types: `matlab.ui.container.ButtonGroup` | `matlab.ui.container.TabGroup` | `matlab.ui.control.CheckBox` | `matlab.ui.control.DiscreteKnob` | `matlab.ui.control.DropDown` | `matlab.ui.control.Knob` | `matlab.ui.control.ListBox` | `matlab.ui.control.RadioButton` | `matlab.ui.control.RockerSwitch` | `matlab.ui.control.Slider` | `matlab.ui.control.StateButton` | `matlab.ui.control.Switch` | `matlab.ui.control.ToggleButton` | `matlab.ui.control.ToggleSwitch`

`option` — Item to choose
depends on component

Item to choose in the component. The data type of `option` depends on the type of component under test. For example, if `comp` is a switch, `option` is a text or numeric value from the `Items` property of the switch. If `comp` is a check box, `option` is a logical value.

When a component has an `Items` property, `option` can be the value of an element in `Items` or an index to an element in `Items`. For example, for a default discrete knob, you can choose 'Medium' using a value for `option` that is either `'Medium'` or 3.

`compNoOpts` — Component to choose without options
UI component object

Component to choose, specified as a UI component object that supports a choose gesture and does not require additional information. Components that support choose gestures include tabs and tree nodes.

Supported Component	Typical Creation Function
Tab	`uitab`
Tree Node	`uitreenode`

Data Types: `matlab.ui.container.Tab` | `matlab.ui.container.TreeNode`

Examples

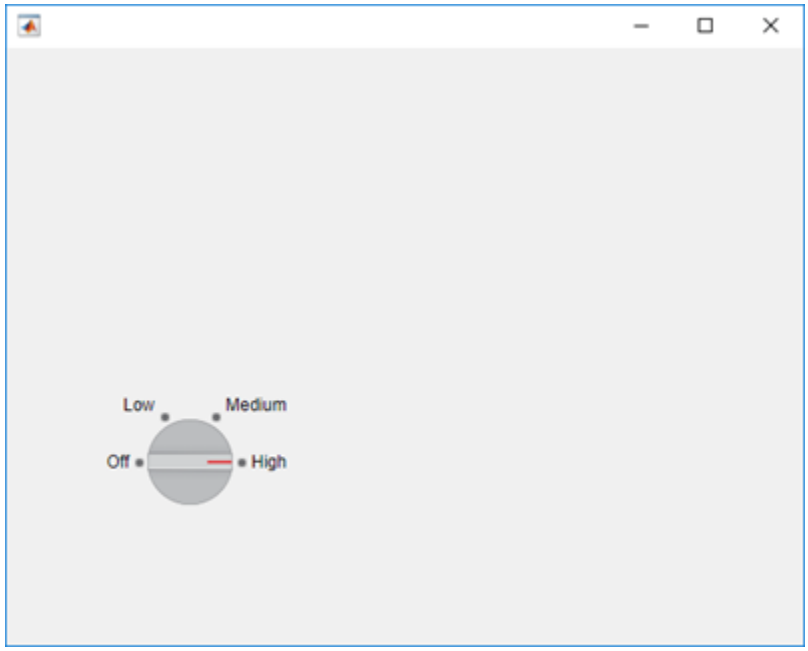
Choose Value on Discrete Knob

Create a discrete knob.

```
knob = uiknob('discrete');
```

Create an interactive test case and choose the 'High' knob value. An animated blue dot performs the programmatic choose gesture.

```
tc = matlab.uitest.TestCase.forInteractiveUse;
tc.choose(knob,'High')
```



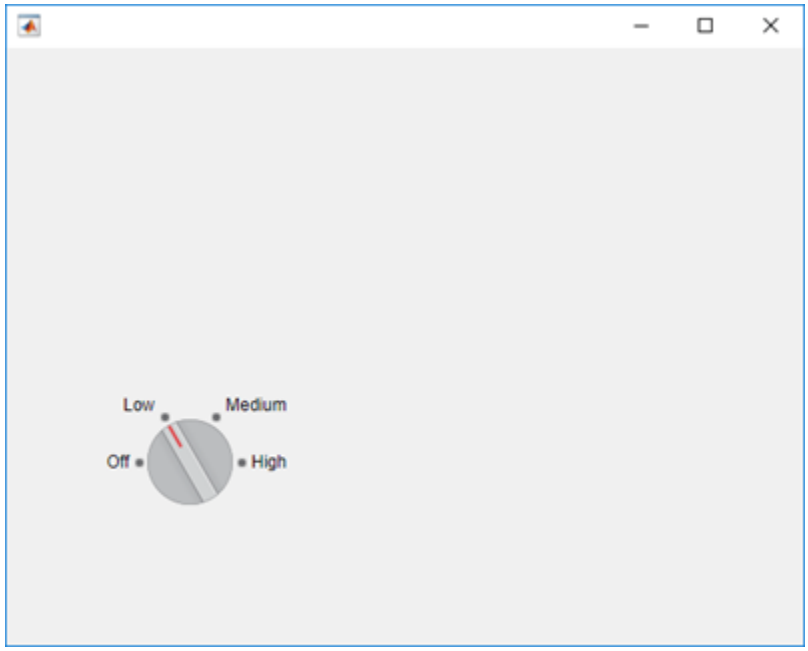
View the value of the `Items` property on the knob.

```
knob.Items
```

```
ans =

  1×4 cell array

    {'Off'}    {'Low'}    {'Medium'}    {'High'}
```

Choose the 'Low' knob value by index. The knob moves from `'High'` to `'Low'`.

```
tc.choose(knob,2)
```

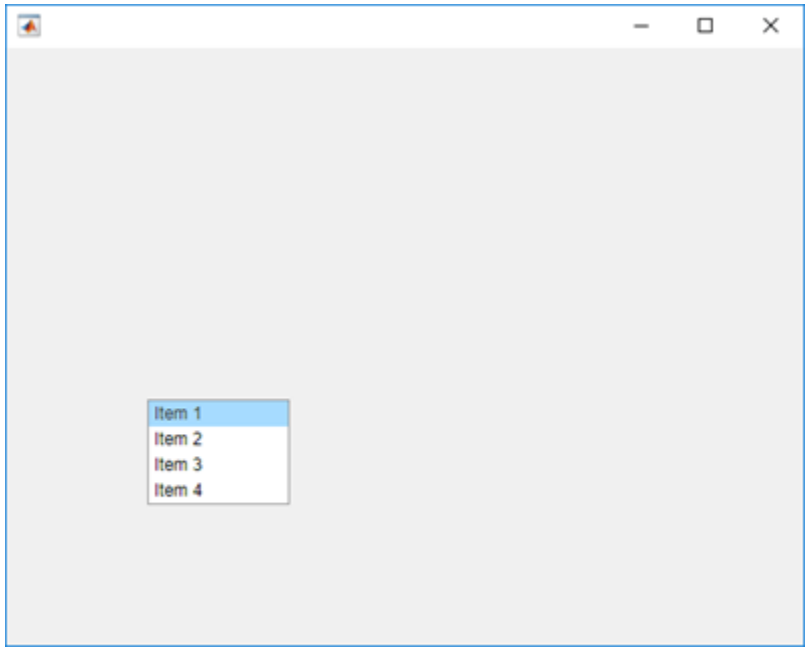


Choose Multiple Items in List Box

Create a list box and enable multiple node selection.

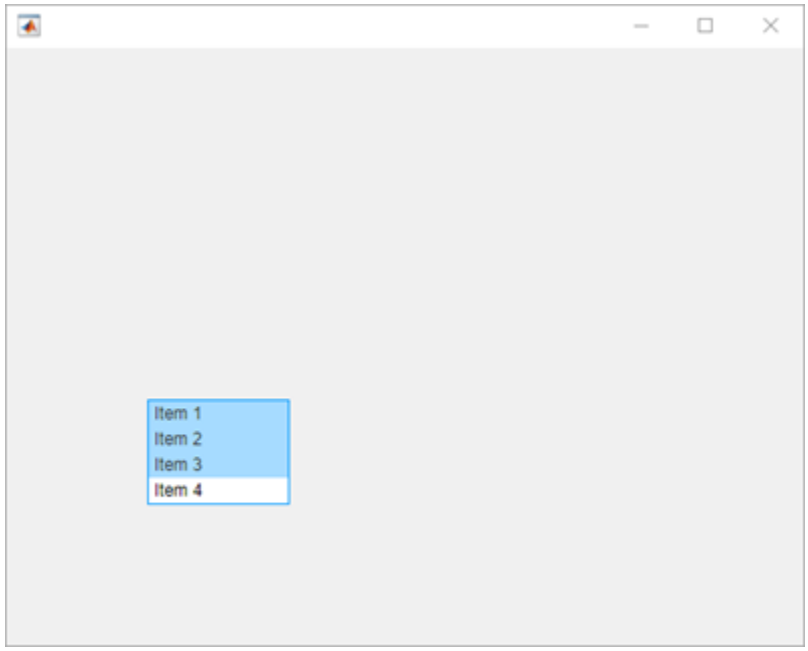
```
listbox = uilistbox('Multiselect','on')
```

```
listbox = 

  ListBox (Item 1) with properties:

              Value: {'Item 1'}
              Items: {'Item 1'  'Item 2'  'Item 3'  'Item 4'}
          ItemsData: []
        Multiselect: 'on'
    ValueChangedFcn: ''
           Position: [100 100 100 74]

  Show all properties
```



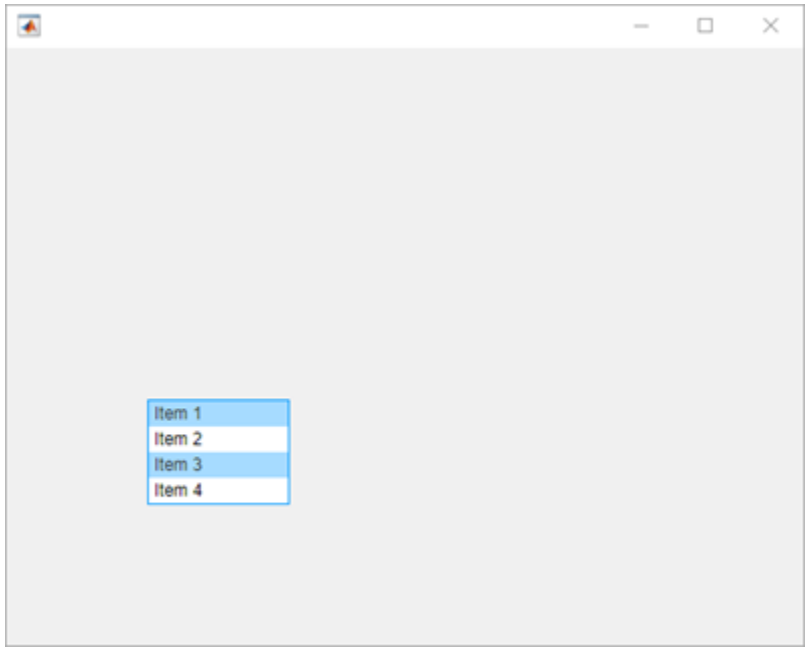
Create an interactive test case and choose items 1 through 3.

```
tc = matlab.uitest.TestCase.forInteractiveUse;
tc.choose(listbox,1:3)
```



Choose items 1 and 3 using the values of the `Items` property.

```
tc.choose(listbox,{'Item 1','Item 3'})
```



Choose Slider Value and Verify Change

Create a slider.

```
s = uislider;
```

Create an interactive test case and verify that the value of the slider button is `0`.

```
tc = matlab.uitest.TestCase.forInteractiveUse;
tc.verifyEqual(s.Value,0)
```

```
Verification passed.
```

Choose a new slider value and verify the slider value changes. Since the framework mimics a user manipulating the component to an arbitrarily precisioned value, it is a best practice to use a tolerance to compare the actual and expected slider values.

```
expVal = 42;
tc.choose(s,expVal)
tc.verifyEqual(s.Value,expVal,'AbsTol',0.1)
```

```
Verification passed.
```

Choose Tab from Group and Verify Change

Create a figure with two tabs.

```
fig = uifigure;
group = uitabgroup(fig);
tab1 = uitab(group,'Title','Tab #1');
tab2 = uitab(group,'Title','Tab #2');
```

Create an interactive test case and verify that the selected tab title contains the substring `'#1'`.

```
tc = matlab.uitest.TestCase.forInteractiveUse;
tc.verifySubstring(group.SelectedTab.Title,'#1')
```

```
verification passed.
```

Choose tab 2 and verify that the selected tab changes.

```
tc.choose(group,'Tab #2')
tc.verifyEqual(group.SelectedTab,tab2)
```

```
Verification passed.
```

See Also

`press`

Topics

"Overview of App Testing Framework"
"Write Test for App"

Introduced in R2018a

drag

Class: `matlab.uitest.TestCase`
Package: `matlab.uitest`

Perform drag gesture on UI component

Syntax

```
drag(testcase,comp,start,stop)
```

Description

`drag(testcase,comp,start,stop)` performs a drag gesture from `start` to `stop` on the UI component `comp`.

Input Arguments

`testcase` — Instance of test case
`matlab.uitest.TestCase` object

Instance of the test case, specified as a `matlab.uitest.TestCase` object.

`comp` — Component to drag
UI component object

Component to drag during test, specified as a UI component object that supports a drag gesture. Components that support drag gestures include continuous knobs and sliders.

Supported Component	Typical Creation Function
Knob	`uiknob`
Slider	`uislider`

Data Types: `matlab.ui.control.Knob` | `matlab.ui.control.Slider`

`start` — Starting value
numeric value within component limits

Starting value of the drag gesture, specified as a numeric value within component limits. Limits are defined by the `Limits` property of the component.

`stop` — Stopping value
numeric value within component limits

Stopping value of the drag gesture, specified as a numeric value within component limits. Limits are defined by the `Limits` property of the component.

Examples

Drag Knob Between Values

Create a knob.

```
knob = uiknob;
```

Create an interactive test case and drag the knob between two values. A blue dot representing the programmatic drag gesture appears and then disappears when the knob reaches the `stop` value.

```
tc = matlab.uitest.TestCase.forInteractiveUse;
tc.drag(knob,13,42)
```

Drag Slider and Verify Stop Value

Create a slider with a minimum value of -237, a maximum value of 237, and a starting value of 7.

```
slider = uislider('Limits',[-237 237],'Value',7);
```

Create an interactive test case and verify the initial value of the slider.

```
tc = matlab.uitest.TestCase.forInteractiveUse;
tc.verifyEqual(slider.Value,7)
```

```
Verification passed.
```

Drag the slider between two values and verify the final value. Since the framework mimics a user manipulating the component to an arbitrarily precisioned value, it is a best practice to use a tolerance to compare the actual and expected slider values.

```
val = 26.75;
tc.drag(slider,-val,val)
tc.verifyEqual(slider.Value,val,'AbsTol',0.1)
```

```
Verification passed.
```

See Also

Topics

"Overview of App Testing Framework"

Introduced in R2018a

matlab.uitest.TestCase.forInteractiveUse

Class: `matlab.uitest.TestCase`
Package: `matlab.uitest`

Create a `TestCase` object for interactive use

Syntax

```
tc = matlab.uitest.TestCase.forInteractiveUse
```

Description

`tc = matlab.uitest.TestCase.forInteractiveUse` creates a `matlab.uitest.TestCase` instance for interactive use. The `TestCase` is configured so you can experiment with it at the MATLAB command prompt. The `TestCase` reacts to qualification failures and successes by displaying messages to the screen for both passing and failing conditions.

Examples

Construct Interactive TestCase

Create a red lamp component that changes to green when a button is pressed.

```
fig = uifigure;
lamp = uilamp(fig,'Position',[50 100 20 20],'Color','red');
button = uibutton(fig,'ButtonPushedFcn',@(btn,event) set(lamp,'Color','green'));
```

Create a `TestCase` object configured for interactive use at the MATLAB command prompt.

```
tc = matlab.uitest.TestCase.forInteractiveUse;
```

Press the button.

```
tc.press(button)
```

Verify the lamp color is green. This test fails because the lamp stores color as an RGB triplet.

```
tc.verifyEqual(lamp.Color,'green')
```

```
Verification failed.

----------------------
Framework Diagnostic:
----------------------
verifyEqual failed.
--> Classes do not match.

    Actual Class:
        double
    Expected Class:
        char

Actual Value:
     0     1     0
Expected char:
    green
```

Verify the lamp color is the RGB triplet for green.

```
tc.verifyEqual(lamp.Color,[0 1 0])
```

```
Verification passed.
```

See Also

Introduced in R2018a

hover

Class: `matlab.uitest.TestCase`
Package: `matlab.uitest`

Perform hover gesture on UI component

Syntax

```
hover(testcase,comp)
hover(testcase,comp,location)
```

Description

`hover(testcase,comp)` performs a hover gesture on the UI component `comp` that supports this gesture.

`hover(testcase,comp,location)` specifies the location to hover at within the component. If you do not specify the location, MATLAB hovers at the center of `comp`.

Input Arguments

`testcase` — Instance of test case
`matlab.uitest.TestCase` object

Instance of test case, specified as a `matlab.uitest.TestCase` object.

`comp` — Component to hover
UI component object

Component to hover on during test, specified as a UI component object that supports a hover gesture. Components that support hover gestures include `Axes`, `UIAxes`, `PolarAxes`, and `Figure` objects.

Supported Component	Typical Creation Function
Axes	`axes`
UI Axes	`uiaxes`
Polar Axes	`polaraxes`
UI Figure	`uifigure`

Data Types: `matlab.graphics.axis.Axes` | `matlab.ui.control.UIAxes` | `matlab.graphics.axis.PolarAxes` | `matlab.ui.Figure`

`location` — Location to hover at
1-by-2 or 1-by-3 numeric array

Location to hover at, specified as the coordinates of the hover point:

- Axes and UI Axes — A 1-by-2 or 1-by-3 numeric array containing *x*-, *y*-, and optionally *z*-coordinates.
- Polar Axes — A 1-by-2 numeric array containing θ- and *r*-coordinates.
- UI Figure — A 1-by-2 numeric array containing *x*- and *y*-coordinates. Specify the coordinates of the hover point measured in pixels from the lower-left corner of the UI figure.

Example: [32.5 13 0.25] (UI axes)

Example: [pi/2 0.5] (Polar axes)

Example: [100 200] (UI figure)

Attributes

`Sealed`	`true`

To learn about attributes of methods, see Method Attributes.

Examples

Hover on UI Axes Object

Create a UI axes and an interactive test case instance.

```
uiAx = uiaxes;
testCase = matlab.uitest.TestCase.forInteractiveUse;
```

Hover over the point defined by the axes coordinates (0.2, 0.4).

```
testCase.hover(uiAx,[0.2 0.4])
```

Hover on UI Figure Object

Create a UI figure and an interactive test case instance.

```
uiFig = uifigure;
testCase = matlab.uitest.TestCase.forInteractiveUse;
```

Hover over the point defined by the pixel coordinates (100,200).

```
testCase.hover(uiFig,[100 200])
```

See Also

`axes` | `polaraxes` | `press` | `uiaxes` | `uifigure`

Topics

"Overview of App Testing Framework"
"Write Test for App"
"Write Test That Uses App Testing and Mocking Frameworks"

Introduced in R2019a

press

Class: `matlab.uitest.TestCase`
Package: `matlab.uitest`

Perform press gesture on UI component

Syntax

```
press(testcase,comp)
press(testcase,comp,location)
press(testcase,spn,direction)
```

Description

`press(testcase,comp)` performs a press gesture on the UI component `comp`.

`press(testcase,comp,location)` specifies the location to press within the component. You can specify the location within an `Axes`, `UIAxes`, `PolarAxes`, or `Figure` object. If you do not specify the location, MATLAB presses at the center of `comp`.

`press(testcase,spn,direction)` specifies whether to press the 'up' or 'down' `direction` button in the spinner `spn`.

Input Arguments

`testcase` — Instance of test case
`matlab.uitest.TestCase` object

Instance of the test case, specified as a `matlab.uitest.TestCase` object.

`comp` — Component to press
UI component object

Component to press during the test, specified as a UI component object that supports a press gesture. Components that support press gestures include images, buttons, check boxes, switches, menus, axes, and figures.

Supported Component	Typical Creation Function
Image	`uiimage`

Button	`uibutton`
	`uiradiobutton`
	`uitogglebutton`

Check Box	`uicheckbox`
Switch	`uiswitch`
Menu	`uimenu`
Axes	`axes`
UI Axes	`uiaxes`
Polar Axes	`polaraxes`
UI Figure	`uifigure`

Data Types: `matlab.ui.control.Image` | `matlab.ui.control.Button` | `matlab.ui.control.StateButton` | `matlab.ui.control.RadioButton` | `matlab.ui.control.ToggleButton` | `matlab.ui.control.CheckBox` | `matlab.ui.control.Switch` | `matlab.ui.control.ToggleSwitch` | `matlab.ui.control.RockerSwitch` | `matlab.ui.container.Menu` | `matlab.graphics.axis.Axes` | `matlab.ui.control.UIAxes` | `matlab.graphics.axis.PolarAxes` | `matlab.ui.Figure`

`location` — Location to press

1-by-2 or 1-by-3 numeric array

Location to press, specified as the coordinates of the point:

- Axes and UI Axes — A 1-by-2 or 1-by-3 numeric array containing *x*-, *y*-, and optionally *z*-coordinates.
- Polar Axes — A 1-by-2 numeric array containing θ- and *r*-coordinates.

- UI Figure — A 1-by-2 numeric array containing *x*- and *y*-coordinates. Specify the coordinates of the point to press measured in pixels from the lower-left corner of the UI figure.

Example: [32.5 13 0.25] (UI axes)

Example: [pi/2 0.5] (Polar axes)

Example: [100 200] (UI figure)

`spn` — Spinner component to press
`matlab.ui.control.Spinner` object

Spinner component to press during the test, specified as a `matlab.ui.control.Spinner` object. Spinner components are typically created with the `uispinner` function.

Data Types: `matlab.ui.control.Spinner`

`direction` — Direction of change
`'up'` | `'down'`

Direction of change for the spinner, specified as `'up'` or `'down'`. To increment the value of the spinner, use `'up'`. To decrement the value, use `'down'`.

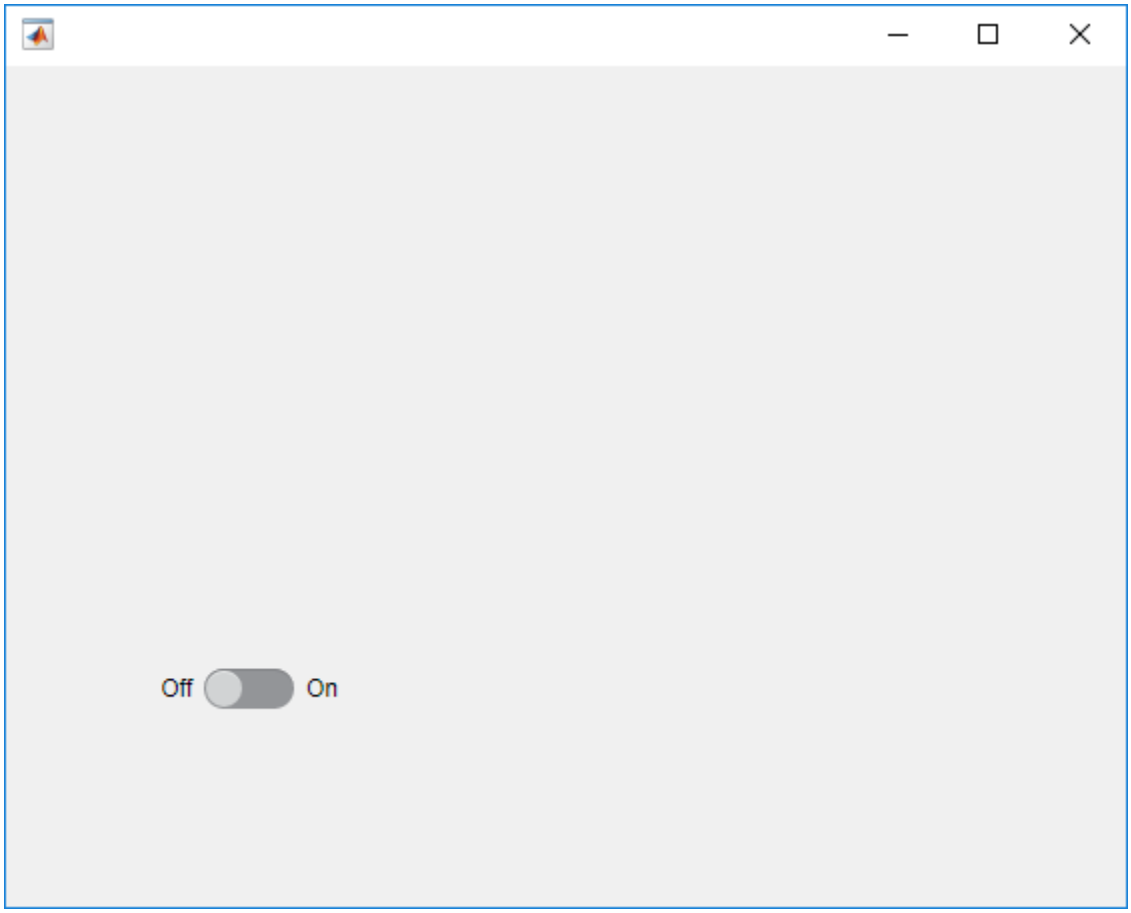
Data Types: `char` | `string`

Examples

Press Switch

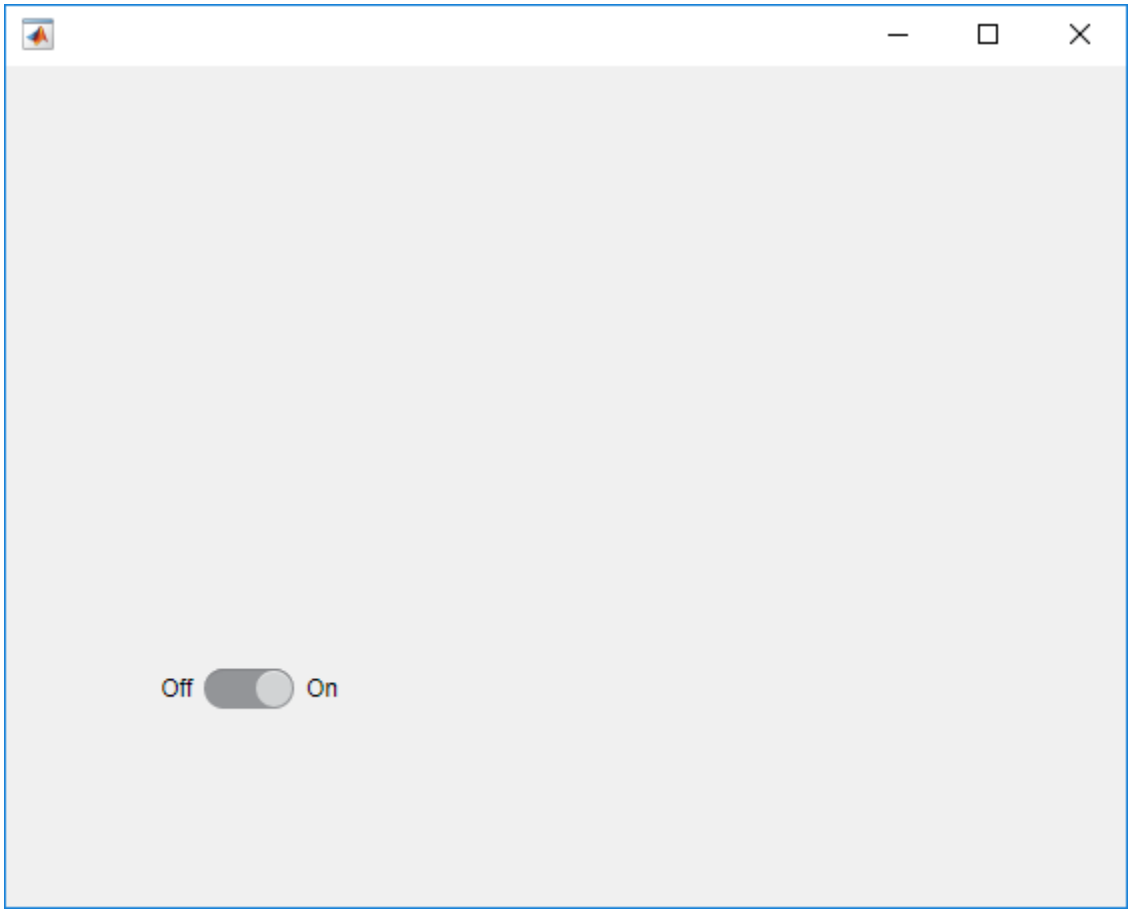
Create a slider switch.

```
s = uiswitch('slider');
```



Create an interactive test case and press the switch. A blue dot representing the programmatic push gesture appears and then disappears at the center of the switch. The switch moves from `'Off'` to `'On'`.

```
tc = matlab.uitest.TestCase.forInteractiveUse;
tc.press(s);
```



Press Axes Location

Create UI axes and an interactive test case instance.

```
ax = uiaxes;
tc = matlab.uitest.TestCase.forInteractiveUse;
```

Press the center of the axes. A blue dot representing the programmatic push gesture appears and then disappears at the center of the axes.

```
tc.press(ax)
```

Press the axes at the coordinates (0.85,0.2). A blue dot representing the programmatic push gesture appears and then disappears at the specified axes coordinate.

```
tc.press(ax,[0.85 0.2])
```

Press Button and Verify Value Change

Create a state button.

```
b = uibutton('state');
```

Create an interactive test case and verify that the value of the state button is `false`.

```
tc = matlab.uitest.TestCase.forInteractiveUse;
tc.verifyFalse(b.Value)
```

```
Verification passed.
```

Press the button and verify the state changes to `true`. A blue dot representing the programmatic push gesture appears and then disappears on the button.

```
tc.press(b)
tc.verifyTrue(b.Value)
```

```
Verification passed.
```

Modify Value of Spinner and Verify Change

Create a spinner with an initial value of 42.

```
s = uispinner('Value',42);
initVal = s.Value;
```

Create an interactive test case and increment the spinner. Verify that the new value of the spinner is the initial value plus the spinner step value.

```
tc = matlab.uitest.TestCase.forInteractiveUse;
tc.press(s,'up')
tc.verifyEqual(s.Value,initVal+s.Step)
```

```
Verification passed.
```

Decrement the value of the spinner and verify that the value of the spinner is equal to the initial value again.

```
tc.press(s,'down')
tc.verifyEqual(s.Value,initVal)
```

```
Verification passed.
```

See Also

choose

Topics

"Overview of App Testing Framework"
"Write Test for App"
"Write Test That Uses App Testing and Mocking Frameworks"

Introduced in R2018a

type

Class: `matlab.uitest.TestCase`
Package: `matlab.uitest`

Type in UI component

Syntax

```
type(testcase,comp,value)
```

Description

`type(testcase,comp,value)` types `value` in the UI component `comp`.

Input Arguments

`testcase` — Instance of test case
`matlab.uitest.TestCase` object

Instance of the test case, specified as a `matlab.uitest.TestCase` object.

`comp` — Component to type in
UI component object

Component to type in during test, specified as a UI component object that supports a type gesture. Components that support type gestures include edit fields and text areas.

Supported Component	Typical Creation Function
Drop Down	`uidropdown`
Edit Field (Numeric, Text)	`uieditfield`
Text Area	`uitextarea`

Supported Component	Typical Creation Function
Spinner	`uispinner`
Date Picker	`uidatepicker`

Data Types: `matlab.ui.control.DropDown` | `matlab.ui.control.EditField` | `matlab.ui.control.NumericEditField` | `matlab.ui.control.Spinner` | `matlab.ui.control.TextArea` | `matlab.ui.control.DatePicker`

`value` — Value to type
depends on component

Value to type into the component. The data type of `value` depends on the type of component under test. For example, if `comp` is a spinner, `value` is specified as a numeric. If `comp` is a text area, `value` is specified as a character vector or string.

Examples

Type in Edit Field and Verify Change

Create a text edit field.

```
ed = uieditfield('Value','Hello')
```

Create an interactive test case and verify the initial value.

```
tc = matlab.uitest.TestCase.forInteractiveUse;
tc.verifyEqual(ed.Value,'Hello')
```

```
Verification passed.
```

Type the word "Goodbye" in the edit field and verify the new value.

```
value = 'Goodbye';
tc.type(ed,value)
tc.verifyEqual(ed.Value,value)
```

```
Verification passed.
```

Add Item to Drop-down List and Verify Change

Create an editable drop-down list.

```
dropdown = uidropdown('Editable','on');
```

Create an interactive test case and add a custom item to the drop-down list.

```
tc = matlab.uitest.TestCase.forInteractiveUse;
tc.type(dropdown,'Custom Item')
```

Verify the new value.

```
tc.verifyEqual(dropdown.Value,'Custom Item')
```

```
Verification passed.
```

See Also

Topics

"Overview of App Testing Framework"

Introduced in R2018a

matlab.uitest.unlock

Unlock figure locked by app testing framework

Syntax

```
matlab.uitest.unlock(fig)
```

Description

`matlab.uitest.unlock(fig)` unlocks a figure that the app testing framework locked so that you can interact with the figure components.

Examples

Unlock Figure

Create a class-based unit test that derives from `matlab.uitest.TestCase` and contains a `keyboard` statement.

```
classdef SimpleUITest < matlab.uitest.TestCase
    methods (Test)
        function test1(testCase)
            fig = uifigure;
            testCase.addTeardown(@delete,fig);
            button = uibutton(fig);
            keyboard;
        end
    end
end
```

Run the test. MATLAB enters debug mode at the `keyboard` command. The figure is locked and you cannot interactively press the button.

```
runtests('SimpleUITest')
```

While in debug mode, unlock the figure. You can interact with the button.

```
K>> matlab.uitest.unlock(fig);
```

Continue execution of the test. The test completes and closes the figure during tear down.

```
K>> dbcont
```

Input Arguments

`fig` — Figure to unlock
figure handle | array of figure handles

Figure to unlock, specified as a figure handle or an array of figure handles. Each figure handle must correspond to a figure created with the `uifigure` function.

Example: `fh`

Example: `[fh1 fh2]`

Data Types: `matlab.ui.Figure`

See Also

`matlab.uitest.TestCase` | `uifigure`

Introduced in R2018a

uminus, -

Unary minus

Syntax

```
C = -A
C = uminus(A)
```

Description

`C = -A` negates the elements of `A` and stores the result in `C`.

`C = uminus(A)` is an alternative way to execute `-A`, but is rarely used. It enables operator overloading for classes.

Examples

Negate Elements of Matrix

Create a 2-by-2 matrix, `A`.

```
A = [1 -3; -2 4]
```

```
A = 2×2

     1    -3
    -2     4
```

Negate the elements of `A`.

```
C = -A
```

```
C = 2×2
```

```
    -1     3
     2    -4
```

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. If A has an integer data type, then it cannot be complex.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `datetime` | `duration` | `calendarDuration`
Complex Number Support: Yes

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`minus` | `uplus`

Topics

“Array vs. Matrix Operations”
“Operator Precedence”
“MATLAB Operators and Special Characters”

Introduced before R2006a

underlyingValue

Underlying numeric value for C++ enumeration object created in MATLAB

Syntax

```
underlyingValue(cppEnum)
```

Description

`underlyingValue(cppEnum)` returns the underlying numeric value for a C++ enumeration object created in MATLAB. `underlyingValue` is an instance method of the enumeration.

Examples

Display Underlying Color Value

Create a file `enums.hpp` with these statements.

```
enum class COLOR {

RED=100,
BLUE=500,
BLACK=1000,
WHITE=5000,

};
```

Build the C++ interface.

```
clibgen.buildInterface('enums.hpp')
addpath enums
```

Type `clib.enums.COLOR.`, then press **Tab**. Select `BLACK`.

Display the underlying numeric value.

```
underlyingValue(clib.enums.COLOR.BLACK)
```

```
ans = 1000
```

Input Arguments

`cppEnum` — C++ enumeration created in MATLAB
C++ enumeration object

C++ enumeration created in MATLAB, specified as a C++ enumeration object.

Example: `clib.enums.COLOR.BLACK`

See Also

`clibgen.buildInterface`

Introduced in R2019a

undocheckout

(Has been removed) Undo previous checkout from source control system (UNIX platforms)

Note `undocheckout` has been removed.

Syntax

```
undocheckout('filename')
undocheckout({'filename1','filename2', ...,'filenamen'})
```

Description

`undocheckout('filename')` makes the file `filename` available for checkout, where `filename` does not reflect any of the changes you made after you last checked it out. Use the full path for `filename` and include the file extension.

`undocheckout({'filename1','filename2', ...,'filenamen'})` makes `filename1` through `filenamen` available for checkout, where the files do not reflect any of the changes you made after you last checked them out. Use the full paths for the file names and include the file extensions.

Examples

Undo the checkouts of `/myserver/myfiles/clock.m` and `/myserver/myfiles/calendar.m` from the source control system:

```
undocheckout({'/myserver/myfiles/clock.m', ...
'/myserver/myfiles/calendar.m'})
```

Introduced before R2006a

unicode2native

Convert Unicode character representation to numeric bytes

Syntax

```
bytes = unicode2native(unicodestr)
bytes = unicode2native(unicodestr,encoding)
```

Description

`bytes = unicode2native(unicodestr)` converts the input Unicode character representation, `unicodestr`, to the user default encoding, and returns the bytes as a `uint8` vector, `bytes`. Output vector `bytes` has the same general array shape as the `unicodestr` input. You can save the output of `unicode2native` to a file using the `fwrite` function. `unicodestr` can be a character vector or a string scalar.

`bytes = unicode2native(unicodestr,encoding)` converts `unicodestr` to the character encoding scheme specified by `encoding`. The input argument `encoding` must have no characters (`''`) or must be a name or alias for an encoding scheme. Some examples are `'UTF-8'`, `'latin1'`, `'US-ASCII'`, and `'Shift_JIS'`. If `encoding` is unspecified or has no characters (`''`), the default encoding scheme is used. `encoding` can be a character vector or a string scalar.

Examples

This example begins with two character vectors containing Unicode character representations. It assumes that `str1` contains text in a Western European language and that `str2` contains Japanese text. The example writes both character vectors into the same file, using the ISO-8859-1 character encoding scheme for the first character vector and the Shift-JIS encoding scheme for the second character vector. The example uses `unicode2native` to convert `str1` and `str2` to the appropriate encoding schemes.

```
fid = fopen('mixed.txt', 'w');
bytes1 = unicode2native(str1, 'ISO-8859-1');
```

```
fwrite(fid, bytes1, 'uint8');
bytes2 = unicode2native(str2, 'Shift_JIS');
fwrite(fid, bytes2, 'uint8');
fclose(fid);
```

See Also

native2unicode

Introduced before R2006a

union

Set union of two arrays

Syntax

```
C = union(A,B)
C = union(A,B,setOrder)
C = union(A,B, ___ ,'rows')
C = union(A,B,'rows', ___ )
[C,ia,ib] = union( ___ )

[C,ia,ib] = union(A,B,'legacy')
[C,ia,ib] = union(A,B,'rows','legacy')
```

Description

`C = union(A,B)` returns the combined data from `A` and `B` with no repetitions. `C` is in sorted order.

- If `A` and `B` are tables or timetables, then `union` returns the combined set of rows from both tables. For timetables, `union` takes row times into account to determine equality, and sorts the output timetable `C` by row times.

`C = union(A,B,setOrder)` returns `C` in a specific order. `setOrder` can be `'sorted'` or `'stable'`.

`C = union(A,B, ___ ,'rows')` and `C = union(A,B,'rows', ___ )` treat each row of `A` and each row of `B` as single entities and return the combined rows from `A` and `B`, with no repetitions. You must specify `A` and `B` and optionally can specify `setOrder`.

The `'rows'` option does not support cell arrays, unless one of the inputs is either a categorical array or a datetime array.

`[C,ia,ib] = union( ___ )` also returns index vectors `ia` and `ib` using any of the previous syntaxes.

- Generally, the values in `C` are a sorted combination of the elements of `A(ia)` and `B(ib)`.
- If the `'rows'` option is specified, then the rows of `C` are a sorted combination of the rows of `A(ia,:)` and `B(ib,:)`.
- If `A` and `B` are tables or timetables, then `C` is a sorted combination of the rows of `A(ia,:)` and `B(ib,:)`.

`[C,ia,ib] = union(A,B,'legacy')` and `[C,ia,ib] = union(A,B,'rows','legacy')` preserve the behavior of the `union` function from R2012b and prior releases.

The `'legacy'` option does not support categorical arrays, datetime arrays, duration arrays, tables, or timetables.

Examples

Union of Two Vectors

Define two vectors with a value in common.

```
A = [5 7 1];
B = [3 1 1];
```

Find the union of vectors `A` and `B`.

```
C = union(A,B)
```

```
C = 1×4

     1     3     5     7
```

Union of Two Tables

Define two tables with rows in common.

```
A = table([1:5]',['A';'B';'C';'D';'E'],logical([0;1;0;1;0]))
```

```
A=5×3 table
    Var1    Var2    Var3
    ____    ____    _____

     1       A      false
     2       B      true
     3       C      false
     4       D      true
     5       E      false
```

```
B = table([1:2:10]',['A';'C';'E';'G';'I'],logical(zeros(5,1)))
```

```
B=5×3 table
    Var1    Var2    Var3
    ____    ____    _____

     1       A      false
     3       C      false
     5       E      false
     7       G      false
     9       I      false
```

Find the union of tables A and B.

```
C = union(A,B)
```

```
C=7×3 table
    Var1    Var2    Var3
    ____    ____    _____

     1       A      false
     2       B      true
     3       C      false
     4       D      true
     5       E      false
     7       G      false
     9       I      false
```

Union of Two Vectors and Their Indices

Define two vectors with a value in common.

```
A = [5 7 1];
B = [3 1 1];
```

Find the union of vectors A and B, as well as the index vectors, ia and ib.

```
[C,ia,ib] = union(A,B)
```

```
C = 1×4

     1     3     5     7


ia = 3×1

     3
     1
     2


ib = 1
```

The values in C are the combined values of A(ia) and B(ib).

Union of Two Tables and Their Indices

Define a table, A, of gender, age, and height for three people.

```
A = table(['M';'M';'F'],[27;52;31],[74;68;64],...
'VariableNames',{'Gender' 'Age' 'Height'},...
'RowNames',{'Ted' 'Fred' 'Betty'})
```

```
A=3×3 table
             Gender    Age    Height
             ______    ___    ______

    Ted        M       27       74
    Fred       M       52       68
    Betty      F       31       64
```

Define a table, B with the same variables as A.

```
B = table(['F';'M'],[64;68],[31;47],...
'VariableNames',{'Gender' 'Height' 'Age'},...
'RowNames',{'Meg' 'Joe'})
```

```
B=2×3 table
           Gender    Height    Age
           ______    ______    ___

    Meg      F         64      31
    Joe      M         68      47
```

Find the union of tables A and B, as well as the index vectors, `ia` and `ib`.

```
[C,ia,ib] = union(A,B)
```

```
C=4×3 table
             Gender    Age    Height
             ______    ___    ______

    Betty      F       31       64
    Ted        M       27       74
    Joe        M       47       68
    Fred       M       52       68


ia = 3×1

     3
     1
     2


ib = 2
```

The data for `Meg` and `Betty` are the same. `union` only returns the index from A, which corresponds to `Betty`.

Union of Rows in Two Matrices

Define two matrices with a row in common.

```
A = [2 2 2; 0 0 1];
B = [1 2 3; 2 2 2; 2 2 2];
```

Find the combined rows of A and B, with no repetition, as well as the index vectors ia and ib.

```
[C,ia,ib] = union(A,B,'rows')
```

```
C = 3×3

     0     0     1
     1     2     3
     2     2     2


ia = 2×1

     2
     1


ib = 1
```

The rows of C are the combined rows of A(ia,:) and B(ib,:).

Union of Two Vectors with Specified Output Order

Use the setOrder argument to specify the ordering of the values in C.

Specify 'stable' if you want the values in C to have the same order as in A and B.

```
A = [5 7 1];
B = [3 1 1];
[C,ia,ib] = union(A,B,'stable')
```

```
C = 1×4

     5     7     1     3


ia = 3×1

     1
```

```
     2
     3


ib = 1
```

Alternatively, you can specify `'sorted'` order.

```
A = [5 7 1];
B = [3 1 1];
[C,ia,ib] = union(A,B,'sorted')
```

```
C = 1×4

     1     3     5     7


ia = 3×1

     3
     1
     2


ib = 1
```

Union of Vectors Containing NaNs

Define two vectors containing `NaN`.

```
A = [5 NaN 1];
B = [4 NaN NaN];
```

Find the union of vectors `A` and `B`.

```
C = union(A,B)
```

```
C = 1×6

     1     4     5   NaN   NaN   NaN
```

`union` treats `NaN` values as distinct.

Cell Array of Character Vectors with Trailing White Space

Create a cell array of character vectors, A.

```
A = {'dog','cat','fish','horse'};
```

Create a cell array of character vectors, B, where some of the vectors have trailing white space.

```
B = {'dog ','cat','fish ','horse'};
```

Combine the elements of A and B.

```
[C,ia,ib] = union(A,B)
```

```
C = 1x6 cell array
    {'cat'}    {'dog'}    {'dog '}    {'fish'}    {'fish '}    {'horse'}


ia = 4×1

     2
     1
     3
     4


ib = 2×1

     1
     3
```

union treats trailing white space in cell arrays of character vectors as distinct characters.

Union of Vectors of Different Classes and Shapes

Create a column vector character array.

```
A = ['A';'B';'C']
```

```
A = 3x1 char array
    'A'
    'B'
    'C'
```

```
class(A)
```

```
ans =
'char'
```

Create a row vector containing elements of numeric type `double`.

```
B = [68 69 70]
```

```
B = 1×3

    68    69    70
```

```
class(B)
```

```
ans =
'double'
```

The `union` of `A` and `B` returns a column vector character array.

```
C = union(A,B)
```

```
C = 6x1 char array
    'A'
    'B'
    'C'
    'D'
    'E'
    'F'
```

```
class(C)
```

```
ans =
'char'
```

Union of Char and Cell Array of Character Vectors

Create a character vector containing the letters a , b, and c.

```
A = ['a';'b';'c'];
class(A)
```

```
ans =
'char'
```

Create a cell array of character vectors containing the letters c, d, and e.

```
B = {'c','d','e'};
class(B)
```

```
ans =
'cell'
```

Combine the elements of A and B.

```
C = union(A,B)
```

```
C = 5x1 cell array
    {'a'}
    {'b'}
    {'c'}
    {'d'}
    {'e'}
```

The result, C, is a cell array of character vectors.

```
class(C)
```

```
ans =
'cell'
```

Preserve Legacy Behavior of union

Use the `'legacy'` flag to preserve the behavior of `union` from R2012b and prior releases in your code.

Find the union of A and B with the current behavior.

```
A = [5 7 1];
B = [3 1 1];
[C1,ia1,ib1] = union(A,B)
```

```
C1 = 1×4

     1     3     5     7


ia1 = 3×1

     3
     1
     2


ib1 = 1
```

Find the union of A and B, and preserve the legacy behavior.

```
A = [5 7 1];
B = [3 1 1];
[C2,ia2,ib2] = union(A,B,'legacy')
```

```
C2 = 1×4

     1     3     5     7


ia2 = 1×2

     1     2


ib2 = 1×2

     3     1
```

Input Arguments

A,B — Input arrays

numeric arrays | logical arrays | character arrays | string arrays | categorical arrays | datetime arrays | duration arrays | cell arrays of character vectors | tables | timetables

Input arrays, specified as numeric arrays, logical arrays, character arrays, string arrays, categorical arrays, datetime arrays, duration arrays, cell arrays of character vectors, tables, or timetables. If you specify the `'rows'` option, `A` and `B` must have the same number of columns.

`A` and `B` must be of the same class with the following exceptions:

- `logical`, `char`, and all numeric classes can combine with `double` arrays.
- Cell arrays of character vectors can combine with character arrays or string arrays.
- Categorical arrays can combine with character arrays, cell arrays of character vectors, or string arrays.
- Datetime arrays can combine with cell arrays of date character vectors or single date character vectors.

There are additional requirements for `A` and `B` based on data type:

- If `A` and `B` are both ordinal categorical arrays, they must have the same sets of categories, including their order. If neither `A` nor `B` are ordinal, they need not have the same sets of categories, and the comparison is performed using the category names. In this case, the categories of `C` are the sorted union of the categories from `A` and `B`.
- If `A` and `B` are tables or timetables, they must have the same variable names (except for order). For tables, row names are ignored, so that two rows that have the same values, but different names, are considered equal. For timetables, row times are taken into account, so that two rows that have the same values, but different times, are not considered equal.
- If `A` and `B` are datetime arrays, they must be consistent with each other in whether they specify a time zone.

`A` and `B` also can be objects with the following class methods:

- `sort` (or `sortrows` for the `'rows'` option)
- `ne`

The object class methods must be consistent with each other. These objects include heterogeneous arrays derived from the same root class. For example, A and B can be arrays of handles to graphics objects.

setOrder — Order flag
'sorted' (default) | 'stable'

Order flag, specified as 'sorted' or 'stable', indicates the order of the values (or rows) in C.

Flag	Description
'sorted'	The values (or rows) in C return in sorted order as returned by sort. **Example** C = union([5 5 3],[1 2 5],'sorted') C = 1 2 3 5
'stable'	The values (or rows) in C return in the same order as they appear in A, then B. **Example** C = union([5 5 3],[1 2 5],'stable') C = 5 3 1 2

Data Types: char | string

Output Arguments

C — Combined data of A and B
vector | matrix | table | timetable

Combined data of A and B, returned as a vector, matrix, table, or timetable. If the inputs A and B are tables or timetables, then the order of the variables in C is the same as the order of the variables in A.

The following describes the shape of C when the inputs are vectors or matrices and when the `'legacy'` flag is not specified:

- If the `'rows'` flag is not specified, then C is a column vector unless both A and B are row vectors, in which case C is a row vector. For example, `union([],[1 2])` returns a column vector.
- If the `'rows'` flag is specified, then C is a matrix containing the combined rows of A and B.

The class of the inputs A and B determines the class of C:

- If the class of A and B are the same, then C is the same class.
- If you combine a `char` or nondouble numeric class with `double`, then C is the same class as the nondouble input.
- If you combine a `logical` class with `double`, then C is `double`.
- If you combine a cell array of character vectors with `char`, then C is a cell array of character vectors.
- If you combine a categorical array with a character vector, cell array of character vectors, or string, then C is a categorical array.
- If you combine a datetime array with a cell array of date character vectors or single date character vector, then C is a datetime array.
- If you combine a string array with a character vector or cell array of character vectors, then C is a string array.

`ia` — Index to A
column vector

Index to A, returned as a column vector when the `'legacy'` flag is not specified. `ia` indicates the values (or rows) in A that contribute to the union. If a value (or row) appears multiple times in A, then `ia` contains the index to the first occurrence of the value (or row). If a value appears in both A and B, then `ia` contains the index to the first occurrence in A.

`ib` — Index to B
column vector

Index to `B`, returned as a column vector when the `'legacy'` flag is not specified. `ib` indicates the values (or rows) in `B` that contribute to the union. If there is a repeated value (or row) appearing exclusively in `B`, then `ib` contains the index to the first occurrence of the value. If a value (or row) appears in both `A` and `B`, then `ib` does not contain an index to the value (or row).

Tips

- To find the union with respect to a subset of variables from a table or timetable, you can use column subscripting. For example, you can use `union(A(:,vars),B(:,vars))`, where *`vars`* is a positive integer, a vector of positive integers, a variable name, a cell array of variable names, or a logical vector. Alternatively, you can use `vartype` to create a subscript that selects variables of a specified type.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- The `'stable'` sorting order is not supported.
- The `'legacy'` flag is not supported.
- Inputs of type `char` are not supported.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support cell arrays for the first or second arguments.

- When you do not specify the `'rows'` option:
  - Inputs `A` and `B` must be vectors with the same orientation. If you specify the `'legacy'` option, then inputs `A` and `B` must be row vectors.
  - The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1.
  - The input`[]` is not supported. Use a 1-by-0 or 0-by-1 input, for example , `zeros(1,0)`, to represent the empty set.
  - If you specify the `'legacy'` option, then empty outputs are row vectors, 1-by-0. They are never 0-by-0.
- When you specify both the `'legacy'` option and the `'rows'` option, the outputs `ia` and `ib` are column vectors. If these outputs are empty, they are 0-by-1. They are never 0-by-0, even if the output `C` is 0-by-0.
- When the `setOrder` is not `'stable'` or when you specify the `'legacy'` option, the inputs must already be sorted in ascending order. The first output, `C`, is sorted in ascending order.
- Complex inputs must be `single` or `double`.
- When one input is complex and the other input is real, do one of the following:
  - Set `setOrder` to `'stable'`.
  - Sort the real input in complex ascending order (by absolute value). Suppose the real input is `x`. Use `sort(complex(x))`or `sortrows(complex(x))`.
- See “Code Generation for Complex Data with Zero-Valued Imaginary Parts” (MATLAB Coder).

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `'legacy'` flag is not supported.
- 64-bit integers are not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`intersect` | `ismember` | `issorted` | `setdiff` | `setxor` | `sort` | `unique`

Topics

"Combine Categorical Arrays"

Introduced before R2006a

unique

Unique values in array

Syntax

```
C = unique(A)
C = unique(A,setOrder)
C = unique(A,occurrence)
C = unique(A, ___ ,'rows')
C = unique(A,'rows', ___ )
[C,ia,ic] = unique( ___ )
```

```
[C,ia,ic] = unique(A,'legacy')
[C,ia,ic] = unique(A,'rows','legacy')
[C,ia,ic] = unique(A,occurrence,'legacy')
[C,ia,ic] = unique(A,'rows',occurrence,'legacy')
```

Description

`C = unique(A)` returns the same data as in `A`, but with no repetitions. `C` is in sorted order.

- If `A` is a table or timetable, then `unique` returns the unique rows in `A` in sorted order. For timetables, `unique` takes row times and row values into account when determining whether rows are unique, and sorts the output timetable `C` by row times.

`C = unique(A,setOrder)` returns the unique values of `A` in a specific order. `setOrder` can be `'sorted'` (default) or `'stable'`.

`C = unique(A,occurrence)` specifies which indices to return in case of repeated values. `occurrence` can be `'first'` (default) or `'last'`.

`C = unique(A, ___ ,'rows')` and `C = unique(A,'rows', ___ )` treat each row of `A` as a single entity and return the unique rows of `A` in sorted order. You must specify `A` and optionally can specify `setOrder` or `occurrence`.

The `'rows'` option does not support cell arrays.

`[C,ia,ic] = unique( ___ )` also returns index vectors `ia` and `ic` using any of the previous syntaxes.

- If `A` is a vector, then `C = A(ia)` and `A = C(ic)`.
- If `A` is a matrix or array, then `C = A(ia)` and `A(:) = C(ic)`.
- If the `'rows'` option is specified, then `C = A(ia,:)` and `A = C(ic,:)`.
- If `A` is a table or a timetable, then `C = A(ia,:)` and `A = C(ic,:)`.

`[C,ia,ic] = unique(A,'legacy')`, `[C,ia,ic] = unique(A,'rows','legacy')`, `[C,ia,ic] = unique(A,occurrence,'legacy')`,and `[C,ia,ic] = unique( A,'rows',occurrence,'legacy')` preserve the behavior of the `unique` function from R2012b and prior releases.

The `'legacy'` option does not support categorical arrays, datetime arrays, duration arrays, calendarDuration arrays, tables, or timetables.

Examples

Unique Values in Vector

Define a vector with a repeated value.

```
A = [9 2 9 5];
```

Find the unique values of `A`.

```
C = unique(A)
```

```
C = 1×3

     2     5     9
```

Unique Rows in Table

Create a table with some repeated data.

```
Name = {'Fred';'Betty';'Bob';'George';'Jane'};
Age = [38;43;38;40;38];
Height = [71;69;64;67;64];
Weight = [176;163;131;185;131];
A = table(Age,Height,Weight,'RowNames',Name)
```

```
A=5×3 table
              Age    Height    Weight
              ___    ______    ______

    Fred      38       71       176
    Betty     43       69       163
    Bob       38       64       131
    George    40       67       185
    Jane      38       64       131
```

Find the unique rows of `A`. `unique` returns the rows of `A` in sorted order by the first variable `Age` and then by the second variable `Height`.

```
C = unique(A)
```

```
C=4×3 table
              Age    Height    Weight
              ___    ______    ______

    Bob       38       64       131
    Fred      38       71       176
    George    40       67       185
    Betty     43       69       163
```

Find the table rows with unique values in the first variable `Age`. If you only want one table variable to contain unique values, you can use the indices returned by `unique` to extract those rows from the table.

```
[C,ia] = unique(A.Age);
B = A(ia,:)
```

```
B=3×3 table
              Age    Height    Weight
              ___    ______    ______

    Fred      38       71       176
    George    40       67       185
```

```
    Betty     43       69       163
```

Unique Values and Their Indices

Define a vector with a repeated value.

```
A = [9 2 9 5];
```

Find the unique values of `A` and the index vectors `ia` and `ic`, such that `C = A(ia)` and `A = C(ic)`.

```
[C, ia, ic] = unique(A)
```

```
C = 1×3

     2     5     9


ia = 3×1

     2
     4
     1


ic = 4×1

     3
     1
     3
     2
```

Unique Rows in Matrix

Create a 10-by-3 matrix with some repeated rows.

```
A = randi(3,10,3)
```

```
A = 10×3

     3     1     2
     3     3     1
     1     3     3
     3     2     3
     2     3     3
     1     1     3
     1     2     3
     2     3     2
     3     3     2
     3     3     1
```

Find the unique rows of A based on the data in the first two columns. Specify three outputs to return the index vectors ia and ic.

```
[C,ia,ic] = unique(A(:,1:2),'rows')
```

```
C = 7×2

     1     1
     1     2
     1     3
     2     3
     3     1
     3     2
     3     3


ia = 7×1

     6
     7
     3
     5
     1
     4
     2


ic = 10×1

     5
     7
```

```
     3
     6
     4
     1
     2
     4
     7
     7
```

Use `ia` to index into `A` and retrieve the rows that have unique combinations of elements in the first two columns.

```
uA = A(ia,:)
```

```
uA = 7×3

     1     1     3
     1     2     3
     1     3     3
     2     3     3
     3     1     2
     3     2     3
     3     3     1
```

Count of Unique Elements

Find the unique elements in a vector and then use `accumarray` to count the number of times each unique element appears.

Create a vector of random integers from 1 through 5.

```
a = randi([1 5],200,1);
```

Find the unique elements in the vector. Return the index vectors `ia` and `ic`.

```
[C,ia,ic] = unique(a);
```

Count the number of times each element in `C` appears in `a`. Specify `ic` as the first input to `accumarray` and `1` as the second input so that the function counts repeated subscripts in `ic`. Summarize the results.

```
a_counts = accumarray(ic,1);
value_counts = [C, a_counts]
```

```
value_counts = 5×2

     1    46
     2    36
     3    38
     4    39
     5    41
```

Unique Values in Vector with Specified Order

Use the `setOrder` argument to specify the ordering of the values in `C`.

Specify `'stable'` if you want the values in `C` to have the same order as in `A`.

```
A = [9 2 9 5];
[C, ia, ic] = unique(A,'stable')
```

```
C = 1×3

     9     2     5


ia = 3×1

     1
     2
     4


ic = 4×1

     1
     2
     1
     3
```

Alternatively, you can specify `'sorted'` order.

```
[C, ia, ic] = unique(A,'sorted')
```

```
C = 1×3

     2     5     9


ia = 3×1

     2
     4
     1


ic = 4×1

     3
     1
     3
     2
```

Unique Values in Array Containing NaNs

Define a vector containing `NaN`.

```
A = [5 5 NaN NaN];
```

Find the unique values of `A`.

```
C = unique(A)
```

```
C = 1×3

     5   NaN   NaN
```

`unique` treats `NaN` values as distinct.

Unique Elements in Presence of Numerical Error

Create a vector `x`. Obtain a second vector `y` by transforming and untransforming `x`. This transformation introduces round-off differences in `y`.

```
x = (1:6)'*pi;
y = 10.^log10(x);
```

Verify that `x` and `y` are not identical by taking the difference.

```
x-y
```

```
ans = 6×1
10^-14 ×

    0.0444
         0
         0
         0
         0
   -0.3553
```

Use `unique` to find the unique elements in the concatenated vector `[x;y]`. The `unique` function performs exact comparisons and determines that some values in `x` are not exactly equal to values in `y`. These are the same elements that have a nonzero difference in `x-y`. Thus, `c` contains values that *appear* to be duplicates.

```
c = unique([x;y])
```

```
c = 8×1

    3.1416
    3.1416
    6.2832
    9.4248
   12.5664
   15.7080
   18.8496
   18.8496
```

Use `uniquetol` to perform the comparison using a small tolerance. `uniquetol` treats elements that are within tolerance as equal.

```
C = uniquetol([x;y])
```

```
C = 6×1

    3.1416
    6.2832
    9.4248
   12.5664
   15.7080
   18.8496
```

Unique Entries in Cell Array of Character Vectors

Create a cell array of character vectors.

```
A = {'one','two','twenty-two','One','two'};
```

Find the unique character vectors contained in A.

```
C = unique(A)
```

```
C = 1x4 cell array
    {'One'}    {'one'}    {'twenty-two'}    {'two'}
```

Cell Array of Character Vectors with Trailing White Space

Create a cell array of character vectors, A, where some of the vectors have trailing white space.

```
A = {'dog','cat','fish','horse','dog ','fish '};
```

Find the unique character vectors contained in A.

```
C = unique(A)
```

```
C = 1x6 cell array
    {'cat'}    {'dog'}    {'dog '}    {'fish'}    {'fish '}    {'horse'}
```

`unique` treats trailing white space in cell arrays of character vectors as distinct characters.

Preserve Legacy Behavior of unique

Use the `'legacy'` flag to preserve the behavior of `unique` from R2012b and prior releases in your code.

Find the unique elements of `A` with the current behavior.

```
A = [9 2 9 5];
[C1, ia1, ic1] = unique(A)
```

```
C1 = 1×3

     2     5     9


ia1 = 3×1

     2
     4
     1


ic1 = 4×1

     3
     1
     3
     2
```

Find the unique elements of `A`, and preserve the legacy behavior.

```
[C2, ia2, ic2] = unique(A, 'legacy')
```

```
C2 = 1×3

     2     5     9


ia2 = 1×3
```

```
     2     4     3

ic2 = 1×4

     3     1     3     2
```

Input Arguments

A — Input array
array

Input array.

- If A is a table, then unique does *not* take row names into account. Two rows that have the same values, but different names, are considered equal.
- If A is a timetable, then unique takes row times into account. Two rows that have the same values, but different times, are not considered equal.

A can also be an object with the following class methods:

- sort (or sortrows for the 'rows' option)
- ne

The object class methods must be consistent with each other. These objects include heterogeneous arrays derived from the same root class. For example, A can be an array of handles to graphics objects.

setOrder — Order flag
'sorted' (default) | 'stable'

Order flag, specified as 'sorted' or 'stable', indicates the order of the values (or rows) in C.

Flag	Description

`'sorted'`	The values (or rows) in `C` return in sorted order as returned by `sort`.
	Example
	`C = unique([5 5 3 4],'sorted')`
	`C =`
	`3 4 5`

`'stable'`	The values (or rows) in `C` return in the same order as in `A`.
	Example
	`C = unique([5 5 3 4],'stable')`
	`C =`
	`5 3 4`

Data Types: `char` | `string`

`occurrence` — Occurrence flag
`'first'` (default) | `'last'`

Occurrence flag, specified as `'first'` or `'last'`, indicates whether `ia` should contain the first or last indices to repeated values found in `A`.

Occurrence Flag	Meaning
`'last'`	If there are repeated values (or rows) in `A`, then `ia` contains the index to the last occurrence of the repeated value. For example: `[C,ia,ic] = unique([9 9 9],'last','legacy')` returns `ia = 3`. This is the default behavior when the `'legacy'` flag is specified.

Occurrence Flag	Meaning
`'first'`	If there are repeated values (or rows) in `A`, then `ia` contains the index to the first occurrence of the repeated value. For example: `[C,ia,ic] = unique([9 9 9],'first')` returns `ia = 1`. This is the default behavior.

Data Types: `char` | `string`

Output Arguments

`C` — Unique data of `A`

array

Unique data of `A`, returned as an array. The class of `C` is the same as the class of the input `A`. The shape of `C` depends on whether the input is a vector or a matrix:

- If the `'rows'` flag is not specified and `A` is a row vector, then `C` is a row vector.
- If the `'rows'` flag is not specified and `A` is not a row vector, then `C` is a column vector.
- If the `'rows'` flag is specified, then `C` is a matrix containing the unique rows of `A`.

`ia` — Index to `A`

column vector

Index to `A`, returned as a column vector of indices to the *first* occurrence of repeated elements. When the `'legacy'` flag is specified, `ia` is a row vector that contains indices to the *last* occurrence of repeated elements.

The indices generally satisfy `C = A(ia)`. If `A` is a table, or if the `'rows'` option is specified, then `C = A(ia,:)`.

`ic` — Index to `C`

column vector

Index to `C`, returned as a column vector when the `'legacy'` flag is not specified. `ic` contains indices that satisfy the following properties.

- If `A` is a vector, then `A = C(ic)`.

- If `A` is a matrix or array, then `A(:) = C(ic)`.
- If `A` is a table, or if the `'rows'` option is specified, then `A = C(ic,:)`.

Tips

- Use `uniquetol` to find unique floating-point numbers using a tolerance.
- To find unique rows in tables or timetables with respect to a subset of variables, you can use column subscripting. For example, you can use `unique(A(:,vars))`, where *`vars`* is a positive integer, a vector of positive integers, a variable name, a cell array of variable names, or a logical vector. Alternatively, you can use `vartype` to create a subscript that selects variables of a specified type.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

- For tall vectors and tall tables, use the syntaxes:
  - `C = unique(A)`
  - `[C,ia,ic] = unique(A)`
- For tall matrices, use the syntaxes:
  - `C = unique(A,'rows')`
  - `[C,ia,ic] = unique(A,'rows')`

For more information, see “Tall Arrays for Out-of-Memory Data”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support cell arrays for the first argument.
- When you do not specify the `'rows'` option:
  - The input `A` must be a vector. If you specify the `'legacy'` option, the input `A` must be a row vector.
  - The first dimension of a variable-size row vector must have fixed length 1. The second dimension of a variable-size column vector must have fixed length 1.
  - The input `[]` is not supported. Use a 1-by-0 or 0-by-1 input, for example, `zeros(1,0)`, to represent the empty set.
  - If you specify the `'legacy'` option, then empty outputs are row vectors, 1-by-0. They are never 0-by-0.
- When you specify both the `'rows'` option and the `'legacy'`option, outputs `ia` and `ic` are column vectors. If these outputs are empty, then they are 0-by-1, even if the output `C` is 0-by-0.
- When the `setOrder` is not `'stable'` or when you specify the `'legacy'` option, the input `A` must already be sorted in ascending order. The first output, `C`, is sorted in ascending order.
- Complex inputs must be `single` or `double`.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `'legacy'` flag is not supported.
- 64-bit integers are not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`intersect` | `ismember` | `issorted` | `setdiff` | `setxor` | `sort` | `union` | `uniquetol`

Introduced before R2006a

uniquetol

Unique values within tolerance

Syntax

```
C = uniquetol(A,tol)
C = uniquetol(A)
[C,IA,IC] = uniquetol( ___ )
[ ___ ] = uniquetol( ___ ,Name,Value)
```

Description

`C = uniquetol(A,tol)` returns the unique elements in `A` using tolerance `tol`. Two values, `u` and `v`, are within tolerance if

```
abs(u-v) <= tol*max(abs(A(:)))
```

That is, `uniquetol` scales the `tol` input based on the magnitude of the data.

`uniquetol` is similar to `unique`. Whereas `unique` performs exact comparisons, `uniquetol` performs comparisons using a tolerance.

`C = uniquetol(A)` uses a default tolerance of `1e-6` for single-precision inputs and `1e-12` for double-precision inputs.

`[C,IA,IC] = uniquetol( ___ )` returns index vectors `IA` and `IC`, such that `C = A(IA)` and `A~C(IC)` (or `A(:)~C(IC)` if `A` is a matrix), where ~ means the values are within tolerance of each other. You can use any of the input arguments in previous syntaxes.

`[ ___ ] = uniquetol( ___ ,Name,Value)` uses additional options specified by one or more Name-Value pair arguments using any of the input or output argument combinations in previous syntaxes. For example, `uniquetol(A,'ByRows',true)` determines the unique rows in `A`.

Examples

Unique Elements in Presence of Numerical Error

Create a vector `x`. Obtain a second vector `y` by transforming and untransforming `x`. This transformation introduces round-off differences in `y`.

```
x = (1:6)'*pi;
y = 10.^log10(x);
```

Verify that `x` and `y` are not identical by taking the difference.

```
x-y
```

```
ans = 6×1
10^-14 ×

    0.0444
         0
         0
         0
         0
   -0.3553
```

Use `unique` to find the unique elements in the concatenated vector `[x;y]`. The `unique` function performs exact comparisons and determines that some values in `x` are not exactly equal to values in `y`. These are the same elements that have a nonzero difference in `x-y`. Thus, `c` contains values that *appear* to be duplicates.

```
c = unique([x;y])
```

```
c = 8×1

    3.1416
    3.1416
    6.2832
    9.4248
   12.5664
   15.7080
   18.8496
   18.8496
```

Use `uniquetol` to perform the comparison using a small tolerance. `uniquetol` treats elements that are within tolerance as equal.

```
C = uniquetol([x;y])
```

```
C = 6×1

    3.1416
    6.2832
    9.4248
   12.5664
   15.7080
   18.8496
```

Determine Unique Rows

By default, `uniquetol` looks for unique *elements* that are within tolerance, but it also can find unique *rows* of a matrix that are within tolerance.

Create a numeric matrix, `A`. Obtain a second matrix, `B`, by transforming and untransforming `A`. This transformation introduces round-off differences to `B`.

```
A = [0.05 0.11 0.18; 0.18 0.21 0.29; 0.34 0.36 0.41; 0.46 0.52 0.76];
B = log10(10.^A);
```

Use `unique` to find the unique rows in `A` and `B`. The `unique` function performs exact comparisons and determines that all of the rows in the concatenated matrix `[A;B]` are unique, even though some of the rows differ by only a small amount.

```
unique([A;B],'rows')
```

```
ans = 8×3

    0.0500    0.1100    0.1800
    0.0500    0.1100    0.1800
    0.1800    0.2100    0.2900
    0.1800    0.2100    0.2900
    0.3400    0.3600    0.4100
    0.3400    0.3600    0.4100
    0.4600    0.5200    0.7600
```

```
    0.4600    0.5200    0.7600
```

Use `uniquetol` to find the unique rows. `uniquetol` treats rows that are within tolerance as equal.

```
uniquetol([A;B],'ByRows',true)
```

```
ans = 4×3

    0.0500    0.1100    0.1800
    0.1800    0.2100    0.2900
    0.3400    0.3600    0.4100
    0.4600    0.5200    0.7600
```

Prepare Vectors for Exact Comparison

Create a vector, `x`. Obtain a second vector, `y`, by transforming and untransforming `x`. This transformation introduces round-off differences to some elements in `y`.

```
x = (1:5)'*pi;
y = 10.^log10(x);
```

Combine `x` and `y` into a single vector, `A`. Use `uniquetol` to reconstruct `A`, treating the values that are within tolerance as equal.

```
A = [x;y]
```

```
A = 10×1

    3.1416
    6.2832
    9.4248
   12.5664
   15.7080
    3.1416
    6.2832
    9.4248
   12.5664
   15.7080
```

```
[C,IA,IC] = uniquetol(A);
newA = C(IC)
```

```
newA = 10×1

    3.1416
    6.2832
    9.4248
   12.5664
   15.7080
    3.1416
    6.2832
    9.4248
   12.5664
   15.7080
```

You can use `newA` with `==` or functions that use exact equality like `isequal` or `unique` in subsequent code.

```
D1 = unique(A)
```

```
D1 = 6×1

    3.1416
    3.1416
    6.2832
    9.4248
   12.5664
   15.7080
```

```
D2 = unique(newA)
```

```
D2 = 5×1

    3.1416
    6.2832
    9.4248
   12.5664
   15.7080
```

Subset Data Using Large Tolerance

Create a cloud of 2-D sample points constrained to be inside a circle of radius `0.5` centered at the point $(\frac{1}{2}, \frac{1}{2})$.

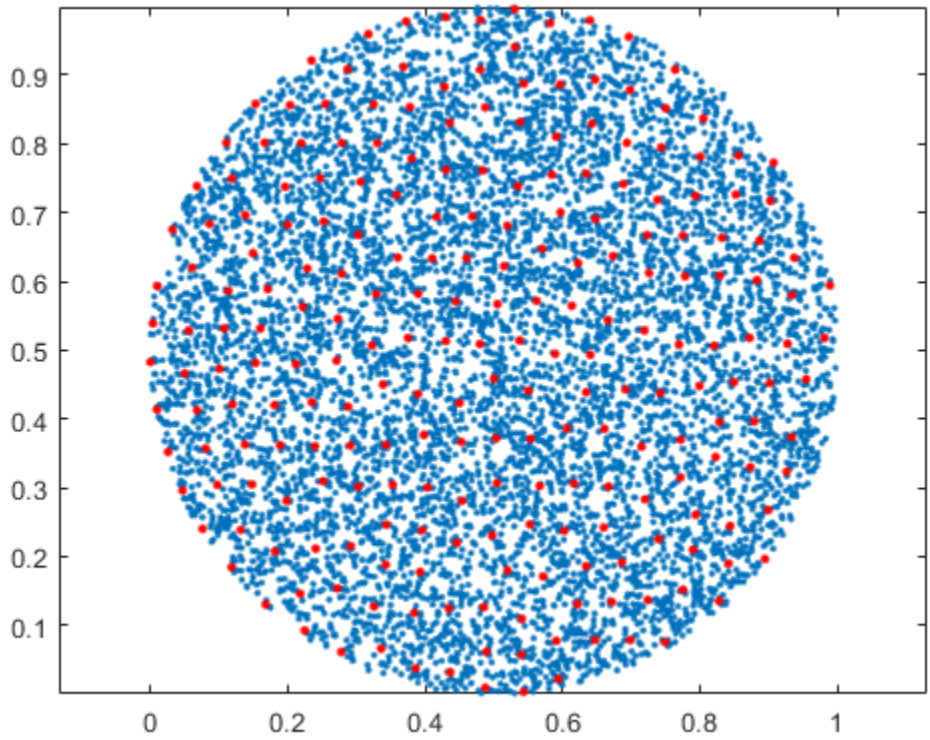
```
x = rand(10000,2);
insideCircle = sqrt((x(:,1)-.5).^2+(x(:,2)-.5).^2)<0.5;
y = x(insideCircle,:);
```

Find a reduced set of points, such that each point of the original dataset is within tolerance of a point.

```
tol = 0.05;
C = uniquetol(y,tol,'ByRows',true);
```

Plot the reduced set of points as red dots on top of the original data set. The red dots are all members of the original data set. All the red dots are at least a distance `tol` apart.

```
plot(y(:,1),y(:,2),'.')
hold on
axis equal
plot(C(:,1), C(:,2), '.r', 'MarkerSize', 10)
```



Average Similar Values in Vector

Create a vector of random numbers and determine the unique elements using a tolerance. Specify `OutputAllIndices` as `true` to return all of the indices for the elements that are within tolerance of the unique values.

```
A = rand(100,1);
[C,IA] = uniquetol(A,1e-2,'OutputAllIndices',true);
```

Find the average value of the elements that are within tolerance of the value `C(2)`.

```
C(2)
```

```
ans = 0.0318
```

```
allA = A(IA{2})
```

```
allA = 3×1

    0.0357
    0.0318
    0.0344
```

```
aveA = mean(allA)
```

```
aveA = 0.0340
```

Specify Absolute Tolerance

By default, `uniquetol` uses a tolerance test of the form `abs(u-v) <= tol*DS`, where `DS` automatically *scales* based on the magnitude of the input data. You can specify a different `DS` value to use with the `DataScale` option. However, absolute tolerances (where `DS` is a scalar) do not scale based on the magnitude of the input data.

First, compare two small values that are a distance `eps` apart. Specify `tol` and `DS` to make the within tolerance equation: `abs(u-v) <= 10^-6`.

```
x = 0.1;
uniquetol([x, exp(log(x))], 10^-6, 'DataScale', 1)
```

```
ans = 0.1000
```

Next, increase the magnitude of the values. The round-off error in the calculation `exp(log(x))` is proportional to the magnitude of the values, specifically to `eps(x)`. Even though the two large values are a distance `eps` from one another, `eps(x)` is now much larger. Therefore, `10^-6` is no longer a suitable tolerance.

```
x = 10^10;
uniquetol([x, exp(log(x))], 10^-6, 'DataScale', 1)
```

```
ans = 1×2
10^10 ×

    1.0000    1.0000
```

Correct this issue by using the default (scaled) value of `DS`.

```
format long
Y = [0.1 10^10];
uniquetol([Y, exp(log(Y))])
```

```
ans = 1×2
10^10 ×

   0.000000000010000   1.000000000000000
```

Specify DataScale by Column

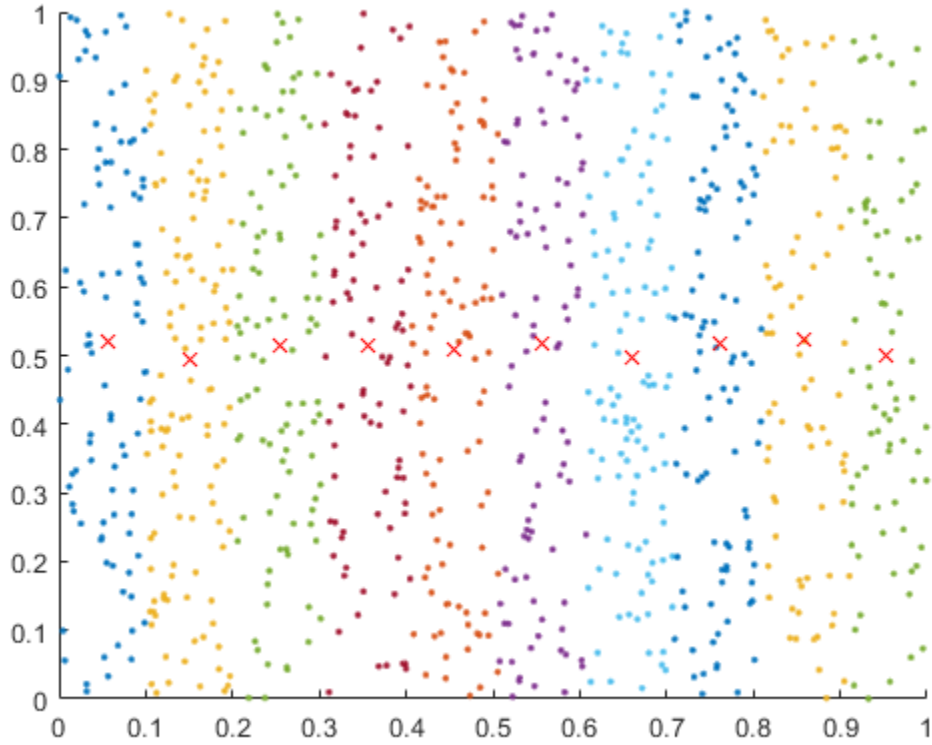
Create a set of random 2-D points, then use `uniquetol` to group the points into vertical bands that have a similar (within tolerance) x-coordinate. Use these options with `uniquetol`:

- Specify `ByRows` as `true` since the point coordinates are in the rows of `A`.
- Specify `OutputAllIndices` as `true` to return the indices for all points that have an x-coordinate within tolerance of each other.
- Specify `DataScale` as `[1 Inf]` to use an absolute tolerance for the `x`-coordinate while ignoring the `y`-coordinate.

```
A = rand(1000,2);
DS = [1 Inf];
[C,IA] = uniquetol(A, 0.1, 'ByRows', true, ...
    'OutputAllIndices', true, 'DataScale', DS);
```

Plot the points and average value for each band.

```
hold on
for k = 1:length(IA)
    plot(A(IA{k},1), A(IA{k},2), '.')
    meanAi = mean(A(IA{k},:));
    plot(meanAi(1), meanAi(2), 'xr')
end
```



Input Arguments

A — Query array
scalar | vector | matrix | multidimensional array

Query array, specified as a scalar, vector, matrix, or multidimensional array. A must be full.

Data Types: `single` | `double`

`tol` — Comparison tolerance
positive, real scalar

Comparison tolerance, specified as a positive, real scalar. `uniquetol` scales the `tol` input using the maximum absolute value in input array `A`. Then `uniquetol` uses the resulting scaled comparison tolerance to determine which elements in `A` are unique. If two elements in `A` are within tolerance of each other, then `uniquetol` considers them to be equal.

Two values, `u` and `v`, are within tolerance if `abs(u-v) <= tol*max(abs(A))`.

To specify an absolute tolerance, specify both `tol` and the `'DataScale'` Name-Value pair.

Example: `tol = 0.05`

Example: `tol = 1e-8`

Example: `tol = eps`

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `C = uniquetol(A,'ByRows',true)`

`OutputAllIndices` — Output index type
`false` (default) | `true` | `0` | `1`

Output index type, specified as the comma-separated pair consisting of `'OutputAllIndices'` and either `false` (default), `true`, `0`, or `1`. `uniquetol` interprets numeric `0` as `false` and numeric `1` as `true`.

When `OutputAllIndices` is `true`, the `uniquetol` function returns the second output, `IA`, as a cell array. The cell array contains the indices for *all* elements in `A` that are within tolerance of a value in `C`. That is, each cell in `IA` corresponds to a value in `C`, and the values in each cell correspond to locations in `A`.

Example: `[C,IA] = uniquetol(A,tol,'OutputAllIndices',true)`

`ByRows` — Row comparison toggle
`false` (default) | `true` | `0` | `1`

Row comparison toggle, specified as the comma-separated pair consisting of `'ByRows'` and either `false` (default), `true`, `0`, or `1`. `uniquetol` interprets numeric `0` as `false` and numeric `1` as `true`. Use this option to find rows in `A` that are unique, within tolerance.

When `ByRows` is `true`:

- `A` must be a 2-D array.
- `uniquetol` compares the rows of `A` by considering each column separately. For two rows to be within tolerance of one another, each column has to be in tolerance.
- Each row in `A` is within tolerance of a row in `C`. However, no two rows in `C` are within tolerance of each other.

Two rows, `u` and `v`, are within tolerance if `all(abs(u-v) <= tol*max(abs(A),[],1))`.

Example: `C = uniquetol(A,tol,'ByRows',true)`

`DataScale` — Scale of data
scalar | vector

Scale of data, specified as the comma-separated pair consisting of `'DataScale'` and either a scalar or vector. Specify `DataScale` as a numeric scalar, `DS`, to change the tolerance test to be `abs(u-v) <= tol*DS`.

When used together with the `ByRows` option, the `DataScale` value also can be a vector. In this case, each element of the vector specifies `DS` for a corresponding column in `A`. If a value in the `DataScale` vector is `Inf`, then `uniquetol` ignores the corresponding column in `A`.

Example: `C = uniquetol(A,'DataScale',1)`

Example: `[C,IA,IC] = uniquetol(A,'ByRows',true,'DataScale',[eps(1) eps(10) eps(100)])`

Data Types: `single` | `double`

Output Arguments

`C` — Unique elements in `A`
vector | matrix

Unique elements in A (within tolerance), returned as a vector or matrix. If A is a row vector, then C is also a row vector. Otherwise, C is a column vector. The elements in C are sorted in ascending order. Each element in A is within tolerance of an element in C, but no two elements in C are within tolerance of each other.

If the ByRows option is true, then C is a matrix containing the unique rows in A. In this case, the rows in C are sorted in ascending order by the first column. Each row in A is within tolerance of a row in C, but no two rows in C are within tolerance of each other.

IA — Index to A

column vector | cell array

Index to A, returned as a column vector of indices to the first occurrence of repeated elements, or as a cell array. IA generally satisfies `C = A(IA)`, with the following exceptions:

- If the ByRows option is true, then `C = A(IA,:)`.
- If the OutputAllIndices option is true, then IA is a cell array and `C(i)~A(IA{i})` where ~ means the values are within tolerance of each other.

IC — Index to C

column vector

Index to C, returned as a column vector of indices. IC satisfies the following properties, where ~ means the values are within tolerance of each other.

- If A is a vector, then `A~C(IC)`.
- If A is a matrix, then `A(:)~C(IC)`.
- If the ByRows option is true, then `A~C(IC,:)`.

Tips

- There can be multiple valid C outputs that satisfy the condition, *no two elements in C are within tolerance of each other*. The uniquetol function just returns one of the valid outputs.

  uniquetol sorts the input lexicographically, and then starts at the lowest value to find unique values within tolerance. As a result, changing the sorting of the input could change the output. For example, `uniquetol(-A)` might not give the same results as `-uniquetol(A)`.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- The `'ByRows'` and `'OutputAllIndices'` options are not supported.
- 64-bit integers are not supported.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`eps` | `ismember` | `ismembertol` | `unique`

Topics

“Average Similar Data Points Using a Tolerance”

Introduced in R2015a

matlab.unittest Package

Summary of packages and classes in MATLAB Unit Test Framework

Description

The `matlab.unittest` package consists of the following classes and packages.

Classes

matlab.unittest.FunctionTestCase	TestCase used for function-based tests
matlab.unittest.Scope	Test scope enumeration class
matlab.unittest.Test	Specification of single test method
matlab.unittest.TestCase	Superclass of all matlab.unittest test classes
matlab.unittest.TestResult	Result of running test suite
matlab.unittest.TestRunner	Class for running tests in matlab.unittest framework
matlab.unittest.TestSuite	Class for grouping tests to run
matlab.unittest.Verbosity	Verbosity level enumeration class

Packages

matlab.unittest.plugins.constraints	Summary of classes in MATLAB Constraints Interface
matlab.unittest.plugins.diagnostics	Summary of classes in MATLAB Diagnostics Interface
matlab.unittest.plugins.fixtures	Summary of classes in MATLAB Fixtures Interface
matlab.unittest.plugins.parameters	Summary of classes associated with MATLAB Unit Test parameters
matlab.unittest.plugins.plugins	Summary of classes in MATLAB Plugins Interface
matlab.unittest.plugins.qualifications	Summary of classes in MATLAB Qualifications Interface
matlab.unittest.plugins.selectors	Summary of classes in MATLAB Selectors Interface

See Also

Introduced in R2013a

unix

Execute UNIX command and return output

Note For platform-independent code, use the `system` command.

Syntax

```
[status,cmdout] = unix(command)
```

Description

`[status,cmdout] = unix(command)` calls the operating system to execute the specified command and returns the standard output of the command to `cmdout`.

MATLAB uses a shell program to execute the given command. It determines which shell program to use by checking environment variables on your system. MATLAB first checks the `MATLAB_SHELL` variable, and if either empty or not defined, then checks `SHELL`. If `SHELL` is also empty or not defined, MATLAB uses `/bin/sh`.

For tips and information about limitations, see `system`.

Examples

Display Operating System Command Status and Output

Display the current folder using the `cd` command. A `status` of zero indicates that the command completed successfully. MATLAB returns a character vector containing the current folder in `cmdout`.

```
command = 'cd';
[status,cmdout] = unix(command)
```

Input Arguments

command — UNIX command
character vector

UNIX command, specified as a character vector. The `command` executes in a UNIX shell, which might not be the shell from which you started MATLAB.

Example: `'ls'`

Output Arguments

status — Command exit status
0 | nonzero integer

Command exit status, returned as either `0` or a nonzero integer. When the command is successful, `status` is `0`. Otherwise, `status` is a nonzero integer.

- If `command` includes the ampersand character (&), then `status` is the exit status when `command` starts
- If `command` does not include the ampersand character (&), then `status` is the exit status upon `command` completion.

cmdout — Output of operating system command
character vector

Output of the operating system command, returned as a character vector. The system shell might not properly represent non-Unicode characters.

See Also

`system`

Topics

"Run External Commands, Scripts, and Programs"

Introduced before R2006a

unloadlibrary

Unload shared C library from memory

Syntax

```
unloadlibrary libname
```

Description

`unloadlibrary libname` unloads library `libname` from memory.

Examples

Load and Unload shrlibsample Library

Add the `shrlib` examples folder to the path.

```
addpath(fullfile(matlabroot,'extern','examples','shrlib'))
```

Load the library, if it is not already loaded.

```
if ~libisloaded('shrlibsample')
   loadlibrary('shrlibsample')
end
```

When finished, unload the library to free memory.

```
unloadlibrary shrlibsample
```

Input Arguments

`libname` — Name of shared library
character vector

Name of shared library, specified as a character vector. Do not include the path or file extension in `libname`.

If you call `loadlibrary` using the `alias` option, then you must use the alias name for the `libname` argument.

Data Types: `char`

Limitations

- Use with libraries that are loaded using the `loadlibrary` function.

Tips

- To unload a MEX file, use the `clear` function.

See Also

`clear` | `libisloaded` | `loadlibrary`

Introduced before R2006a

unmesh

Convert edge matrix to coordinate and Laplacian matrices

Syntax

```
[L,XY] = unmesh(E)
```

Description

`[L,XY] = unmesh(E)` returns the Laplacian matrix `L` and mesh vertex coordinate matrix `XY` for the `M`-by-4 edge matrix `E`. Each row of the edge matrix must contain the coordinates `[x1 y1 x2 y2]` of the edge endpoints.

Input Arguments

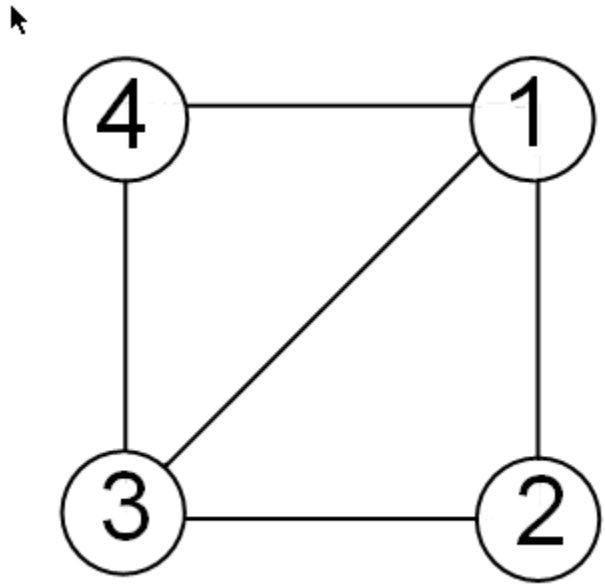
`E`	M-by-4 edge matrix `E`.

Output Arguments

`L`	Laplacian matrix representation of the graph.
`XY`	Mesh vertex coordinate matrix.

Examples

Take a simple example of a square with vertices at (1,1), (1,–1),(–1,–1), and (–1,1), where the connections between vertices are the four perpendicular edges of the square plus one diagonal connection between (–1, –1) and (1,1).



The edge matrix E for this graph is:

```
E=[1 1 1 -1;  % edge from 1 to 2
1 -1 -1 -1;   % edge from 2 to 3
-1 -1 -1 1;   % edge from 3 to 4
-1 -1 1 1;    % edge from 3 to 1
-1 1 1 1]     % edge from 4 to 1
```

Use `unmesh` to create the output matrices,

```
[A,XY]=unmesh(E);
4 vertices:
4/4
```

The Laplacian matrix is defined as

$$L_{ij} = \begin{cases} \deg(v_i) & \text{if } i = j \\ -1 & \text{if } i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise} \end{cases}$$

`unmesh` returns the Laplacian matrix `L` in sparse notation.

```
L

L =

   (1,1)        3
   (2,1)       -1
   (3,1)       -1
   (4,1)       -1
   (1,2)       -1
   (2,2)        2
   (4,2)       -1
   (1,3)       -1
   (3,3)        2
   (4,3)       -1
   (1,4)       -1
   (2,4)       -1
   (3,4)       -1
```

To see `L` in regular matrix notation, use the `full` command.

```
full(L)

ans =

     3    -1    -1    -1
    -1     2     0    -1
    -1     0     2    -1
    -1    -1    -1     3
```

The mesh coordinate matrix `XY` returns the coordinates of the corners of the square.

```
XY

XY =

    -1    -1
    -1     1
     1    -1
     1     1
```

See Also

gplot | treeplot

unmkpp

Extract piecewise polynomial details

Syntax

```
[breaks,coefs,L,order,dim] = unmkpp(pp)
```

Description

`[breaks,coefs,L,order,dim] = unmkpp(pp)` extracts information from the fields of the piecewise polynomial structure `pp`.

Examples

Unwrap Piecewise Polynomial Information

Create a piecewise polynomial structure for the polynomial $f(x) = x^2 + x + 1$ on the interval [0 3], and then extract the information from the fields of the structure.

```
pp = mkpp([0 3],[1 1 1])
```

```
pp = struct with fields:
      form: 'pp'
    breaks: [0 3]
     coefs: [1 1 1]
    pieces: 1
     order: 3
       dim: 1
```

```
[breaks,coefs,L,order,dim] = unmkpp(pp)
```

```
breaks = 1×2
```

```
     0     3

coefs = 1×3

     1     1     1

L = 1

order = 3

dim = 1
```

Input Arguments

pp — Piecewise polynomial
structure

Piecewise polynomial, specified as a structure. You can create `pp` using `spline`, `pchip`, `interp1`, or the spline utility function `mkpp`.

Output Arguments

`breaks` — Break points
vector

Break points, returned as a vector of length `L+1` with strictly increasing elements that represent the start and end of each of `L` intervals.

`coefs` — Polynomial coefficients
matrix

Polynomial coefficients, returned as an `L`-by-`k` matrix with each row `coefs(i,:)` containing the local coefficients of an order `k` polynomial on the ith interval, `[breaks(i),breaks(i+1)]`.

L — Number of intervals
scalar

Number of intervals, returned as a scalar.

`order` — Order of polynomials
scalar

Order of polynomials, returned as a scalar.

`dim` — Dimension of target
scalar | vector

Dimension of target, returned as a scalar or vector.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `pp` must be a valid piecewise polynomial structure created by `mkpp`, `spline`, or `pchip` in MATLAB or by the code generator.
- Code generation does not support `pp` structures created by `interp1` in MATLAB.

See Also

`mkpp` | `pchip` | `ppval` | `spline`

Introduced before R2006a

unregisterallevents

Unregister all event handlers associated with COM object events

Syntax

```
unregisterallevents(c)
```

Description

`unregisterallevents(c)` unregisters all events previously registered with COM object `c`. After calling `unregisterallevents`, the object no longer responds to any events.

Examples

Remove All Event Handlers from Excel Workbook

Unregister the event handlers from all Microsoft Excel workbooks events.

To run this example, create a workbook and register events.

```
myApp = actxserver('Excel.Application');
wbs = myApp.Workbooks;
wb = Add(wbs);
registerevent(wb,{'Activate' 'EvtActivateHndlr'; 'Deactivate' 'EvtDeactivateHndlr'})
eventlisteners(wb)
```

```
ans =

  2×2 cell array

    'Activate'      'EvtActivateHndlr'
    'Deactivate'    'EvtDeactivateHndlr'
```

Unregister the event handlers.

```
unregisterallevents(wb)
eventlisteners(wb)
```

```
ans =

  0×0 empty cell array
```

Input Arguments

c — COM object
function handle

COM object, specified as a function handle.

Limitations

- COM functions are available on Microsoft Windows systems only.

See Also

`eventlisteners` | `isevent` | `registerevent` | `unregisterevent`

Introduced before R2006a

unregisterevent

Unregister event handler associated with COM object event at run time

Syntax

```
unregisterevent(c,eventhandler)
```

Description

`unregisterevent(c,eventhandler)` removes the association of a specific event handler routine from its corresponding event. Once you unregister an event, the object no longer responds to the event. You can unregister events at any time after creating a control.

Examples

Remove Event Handler from Excel Workbook

Unregister the `EvtDeactivateHndlr` event handler from a Microsoft Excel workbooks `Deactivate` event.

To run this example, create a workbook and register events.

```
myApp = actxserver('Excel.Application');
wbs = myApp.Workbooks;
wb = Add(wbs);
registerevent(wb,{'Activate' 'EvtActivateHndlr'; 'Deactivate' 'EvtDeactivateHndlr'})
eventlisteners(wb)
```

```
ans =

  2×2 cell array

    'Activate'      'EvtActivateHndlr'
    'Deactivate'    'EvtDeactivateHndlr'
```

Unregister the `Deactivate` event handler. MATLAB shows the remaining registered event (`Activate`) with its corresponding event handler.

```
unregisterevent(wb,{'Deactivate' 'EvtDeactivateHndlr'})
eventlisteners(wb)
```

```
ans =

  1×2 cell array

    'Activate'    'EvtActivateHndlr'
```

Input Arguments

c — COM object
function handle

COM object, specified as a function handle.

`eventhandler` — Function to call when event occurs
cell array

Function to call when event occurs, specified as a cell array, specifies both events and event handlers.

Specify events in the `eventhandler` argument using the names of the events. Strings or character vectors used in the `eventhandler` argument are not case-sensitive. `unregisterevent` does not accept numeric event identifiers.

Limitations

- COM functions are available on Microsoft Windows systems only.

See Also

`eventlisteners` | `isevent` | `registerevent` | `unregisterallevents`

Topics

"COM Event Handlers"

Introduced before R2006a

unstack

Unstack data from single variable into multiple variables

Syntax

```
U = unstack(S,vars,ivar)
U = unstack(S,vars,ivar,Name,Value)
[U,is] = unstack( ___ )
```

Description

`U = unstack(S,vars,ivar)` converts the table or timetable, `S`, to an equivalent table or timetable, `U`, that is unstacked. `vars` specifies variables in `S`, each of which is unstacked into multiple variables in `U`. In general, `U` contains more variables, but fewer rows, than `S`.

The `ivar` input argument specifies the variable in `S` that `unstack` uses as an indicator variable. The values in `ivar` determine which variables in `U` contain elements taken from `vars` after unstacking.

The `unstack` function treats the remaining variables differently in tables and timetables.

- If `S` is a table, then `unstack` treats the remaining variables as grouping variables on page 1-16562. Each unique combination of values in the grouping variables identifies a group of rows in `S` that is unstacked into a single row of `U`.
- If `S` is a timetable, then `unstack` discards the remaining variables. However, `unstack` treats the vector of row times as a grouping variable.

You cannot unstack the row names of a table, or the row times of a timetable, or specify either as the indicator variable. You can specify row names or row times as constant variables with the `'ConstantVariables'` argument.

`U = unstack(S,vars,ivar,Name,Value)` converts the table or timetable `S` with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify how `unstack` converts variables from `S` to variables in `U`.

`[U,is] = unstack( ___ )` also returns an index vector, `is`, indicating the correspondence between rows in `U` and rows in `S`. You can use any of the previous input arguments.

Examples

Separate One Variable into Three Variables

Create a table indicating the amount of snowfall in various towns for various storms.

```
Storm = [3;3;1;3;1;1;4;2;4;2;4;2];
Town = {'T1';'T3';'T1';'T2';'T2';'T3';...
    'T2';'T1';'T3';'T3';'T1';'T2'};
Snowfall = [0;3;5;5;9;10;12;13;15;16;17;21];
```

```
S = table(Storm,Town,Snowfall)
```

```
S=12×3 table
    Storm     Town     Snowfall
    _____    ______    ________

      3      {'T1'}        0
      3      {'T3'}        3
      1      {'T1'}        5
      3      {'T2'}        5
      1      {'T2'}        9
      1      {'T3'}       10
      4      {'T2'}       12
      2      {'T1'}       13
      4      {'T3'}       15
      2      {'T3'}       16
      4      {'T1'}       17
      2      {'T2'}       21
```

`S` contains three snowfall entries for each storm, one for each town. `S` is in stacked format.

Separate the variable `Snowfall` into three variables, one for each town specified in the variable, `Town`. The output table, `U`, is in unstacked format.

```
U = unstack(S,'Snowfall','Town')
```

```
U=4×4 table
    Storm    T1    T2    T3
    _____    __    __    __

      3       0     5     3
      1       5     9    10
      4      17    12    15
      2      13    21    16
```

Each row in U contains data from rows in S that have the same value in the grouping variable, Storm. The order of the unique values in Storm determines the order of the data in U.

Apply Aggregation Function to Each Group

Unstack data and apply an aggregation function to multiple rows in the same group that have the same values in the indicator variable.

Create a table containing data on the price of two stocks over 2 days.

```
Date = [repmat({'4/12/2008'},6,1);...
    repmat({'4/13/2008'},5,1)];
Stock = {'Stock1';'Stock2';'Stock1';'Stock2';...
    'Stock2';'Stock2';'Stock1';'Stock2';...
    'Stock2';'Stock1';'Stock2'};
Price = [60.35;27.68;64.19;25.47;28.11;27.98;...
    63.85;27.55;26.43;65.73;25.94];

S = table(Date,Stock,Price)
```

```
S=11×3 table
        Date           Stock       Price
    _____________    __________    _____

    {'4/12/2008'}    {'Stock1'}    60.35
    {'4/12/2008'}    {'Stock2'}    27.68
    {'4/12/2008'}    {'Stock1'}    64.19
    {'4/12/2008'}    {'Stock2'}    25.47
    {'4/12/2008'}    {'Stock2'}    28.11
    {'4/12/2008'}    {'Stock2'}    27.98
    {'4/13/2008'}    {'Stock1'}    63.85
```

```
    {'4/13/2008'}    {'Stock2'}    27.55
    {'4/13/2008'}    {'Stock2'}    26.43
    {'4/13/2008'}    {'Stock1'}    65.73
    {'4/13/2008'}    {'Stock2'}    25.94
```

`S` contains two prices for `STOCK1` during the first day and four prices for `STOCK2` during the first day.

Create a table containing separate variables for each stock and one row for each day. Use `Date` as the grouping variable and apply the aggregation function, `@mean`, to the numeric values from the variable, `Price`, for each group.

```
[U,is] = unstack(S,'Price','Stock',...
    'AggregationFunction',@mean)
```

```
U=2×3 table
        Date         Stock1    Stock2
    _____________    ______    ______

    {'4/12/2008'}    62.27     27.31
    {'4/13/2008'}    64.79     26.64


is = 2×1

     1
     7
```

`U` contains the average price for each stock grouped by date.

`is` identifies the index of the first value for each group of rows in `S`. The first value for the group `'4/13/2008'` is in the seventh row of `S`.

Input Arguments

S — Input table
table | timetable

Input table, specified as a table or a timetable. S must contain data variables to unstack, vars, and an indicator variable, ivar. The remaining variables in S can be treated as either grouping variables or constant variables.

vars — Variables in S to unstack
positive integer | vector of positive integers | character vector | cell array of character vectors | string array | logical vector

Variables in S to unstack, specified as a positive integer, vector of positive integers, character vector, cell array of character vectors, string array, or logical vector.

ivar — Indicator variable in S
positive integer | character vector | string scalar

Indicator variable in S, specified as a positive integer, a character vector, or a string scalar. The values in the variable specified by ivar indicate which variables in U contain elements taken from the variables specified by vars.

The variable specified by ivar can be a numeric vector, logical vector, character array, cell array of character vectors, string array, or categorical vector.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'AggregationFunction',@mean applies the aggregation function @mean to the values in vars.

GroupingVariables — Grouping variables in S that define groups of rows
positive integer | vector of positive integers | character vector | cell array of character vectors | string array | logical vector

Grouping variables in S that define groups of rows, specified as the comma-separated pair consisting of 'GroupingVariables' and a positive integer, vector of positive integers, character vector, cell array of character vectors, string array, or logical vector. Each group of rows in S becomes one row in U.

S can have row labels along its first dimension. If S is a table, then it can have row names as the labels. If S is a timetable, then it must have row times as the labels. unstack can treat row labels as grouping variables.

- If you do not specify 'GroupingVariables', and S is a timetable, then unstack treats the row times as a grouping variable.
- If you specify 'GroupingVariables', and S has row names or row times, then unstack does not treat them as grouping variables, unless you include them in the value of 'GroupingVariables'.

ConstantVariables — Variables constant within a group
positive integer | vector of positive integers | character vector | cell array of character vectors | string array | logical vector

Variables constant within a group, specified as the comma-separated pair consisting of 'ConstantVariables' and a positive integer, vector of positive integers, character vector, cell array of character vectors, string array, or logical vector.

The values for these variables in U are taken from the first row in each group in S.

You can include the row names or row times of S when you specify the value of 'ConstantVariables'.

NewDataVariableNames — Names for new data variables in U
cell array of character vectors | string array

Names for the new data variables in U, specified as the comma-separated pair consisting of 'NewDataVariableNames' and a cell array of character vectors or string array.

If you do not specify 'NewDataVariableNames', then unstack creates names for the new data variables in U based on values in the indicator variable specified by ivar.

AggregationFunction — Aggregation function from values in vars to single value
function handle

Aggregation function from values in vars to a single value, specified as the comma-separated pair consisting of 'AggregationFunction' and a function handle. unstack applies this function to rows from the same group that have the same value in ivar. The function must aggregate the data values into a single value.

For a numeric data variable, the default is `@sum`. For nonnumeric variables, there is no default function, and you must specify the `'AggregationFunction'` name-value pair argument if multiple rows in the same group have the same value in `ivar`.

Output Arguments

`U` — Output table
table | timetable

Output table, returned as a table or a timetable. `U` contains the unstacked data variables, the grouping variables, and the first value of each group from any constant variables.

The order of the data in `U` is based on the order of the unique values in the grouping variables.

You can store additional metadata such as descriptions, variable units, variable names, and row names in `U`. For more information, see the Properties sections of `table` or `timetable`.

`is` — Index to `S`
column vector

Index to `S`, returned as a column vector. For each row in `U`, the index vector, `is`, identifies the index of the first value in the corresponding group of rows in `S`.

More About

Grouping Variables

Grouping variables are utility variables used to group, or categorize, data. Grouping variables are useful for summarizing or visualizing data by group. You can define groups in your table by specifying one or more grouping variables.

A grouping variable can be any of the following:

- Categorical vector
- Cell array of character vectors

- String array
- Character array
- Numeric vector, typically containing positive integers
- Logical vector

Rows that have the same grouping variable value belong to the same group. If you use multiple grouping variables, rows that have the same combination of grouping variable values belong to the same group.

Tips

- You can specify more than one data variable in `S`, and each variable becomes a set of unstacked data variables in `U`. Use a vector of positive integers, a cell array or string array containing multiple variable names, or a logical vector to specify `vars`. The one indicator variable, specified by the input argument, `ivar`, applies to all data variables specifies by `vars`.

See Also

`join` | `stack`

Introduced in R2013b

untar

Extract contents of tar file

Syntax

```
untar(tarfilename)
untar(tarfilename,outputfolder)
filenames = untar( ___ )
```

Description

`untar(tarfilename)` extracts the archived contents of `tarfilename` into the current folder, preserving the attributes and timestamps of each file. `untar` can extract files from your local system or files from an Internet URL.

If a file exists with the same name and the file is not read-only, MATLAB overwrites it. On Microsoft Windows platforms, MATLAB does not set the hidden, system, and archive attributes.

`untar(tarfilename,outputfolder)` extracts the archived contents of `tarfilename` into `outputfolder`. If `outputfolder` does not exist, MATLAB creates it.

`filenames = untar( ___ )` returns a cell array of character vectors containing the names of the extracted files. You can use this syntax with any of the input argument combinations in the previous syntaxes.

Examples

Extract Tar File to New Folder

Create a tar file, and then extract it to the folder `backup`.

Create a tar file of all the program files in the current folder.

```
tar('myfiles.tar.gz',{'*.m','*.mlx'});
```

Extract `myfiles.tar.gz` to the folder `backup`.

```
programFiles = untar('myfiles','backup')
```

```
programFiles = 1x3 cell array
    {'backup\myfile1.m'}    {'backup\myfile2.m'}    {'backup\ExtractT...'}
```

Extract Archive File from URL

Download and extract an archive file from a URL to a local folder.

Suppose you have the archive file `example.tar.gz` stored at the URL `http://example.com/example.tar.gz`. Download and extract the file to the `example` folder.

```
url = 'http://example.com/example.tar.gz';
gunzip(url, 'example');
untar('example/example.tar','example');
```

Input Arguments

`tarfilename` — Name of tar file
character vector | string scalar

Name of tar file to extract from, specified as a character vector or a string scalar. If `tarfilename` has no extension, MATLAB searches for `tarfilename` appended with `.tgz`, `.tar.gz`, or `.tar`. If you specify a `.tgz` or `.gz` extension, then `untar` extracts the specified file using `gunzip`.

You can specify `tarfilename` as an absolute path, or a path relative to the current folder.

If `tarfilename` is a URL, then `tarfilename` must include the protocol type (for example, `http://`). MATLAB downloads the URL to the temporary folder on your system, and then deletes it the URL on cleanup.

Data Types: `char` | `string`

`outputfolder` — Target folder
character vector | string scalar

Target folder for the extracted files, specified as a character vector or a string scalar.

Data Types: `char` | `string`

See Also

`gunzip` | `gzip` | `tar` | `unzip` | `zip`

Introduced before R2006a

unwrap

Shift phase angles

Syntax

```
Q = unwrap(P)
Q = unwrap(P,tol)
Q = unwrap(P,[],dim)
Q = unwrap(P,tol,dim)
```

Description

`Q = unwrap(P)` unwraps the radian phase angles in a vector `P`. Whenever the jump between consecutive angles is greater than or equal to π radians, `unwrap` shifts the angles by adding multiples of $\pm 2\pi$ until the jump is less than π. If `P` is a matrix, `unwrap` operates columnwise. If `P` is a multidimensional array, `unwrap` operates on the first dimension whose size is larger than 1.

`Q = unwrap(P,tol)` compares the jump between elements of `P` to the jump threshold `tol` instead of the default value π radians. If you specify a jump threshold less than π, unwrap uses the default jump threshold π.

`Q = unwrap(P,[],dim)` unwraps along the dimension `dim`.

`Q = unwrap(P,tol,dim)` unwraps along the dimension `dim` using the jump threshold `tol`.

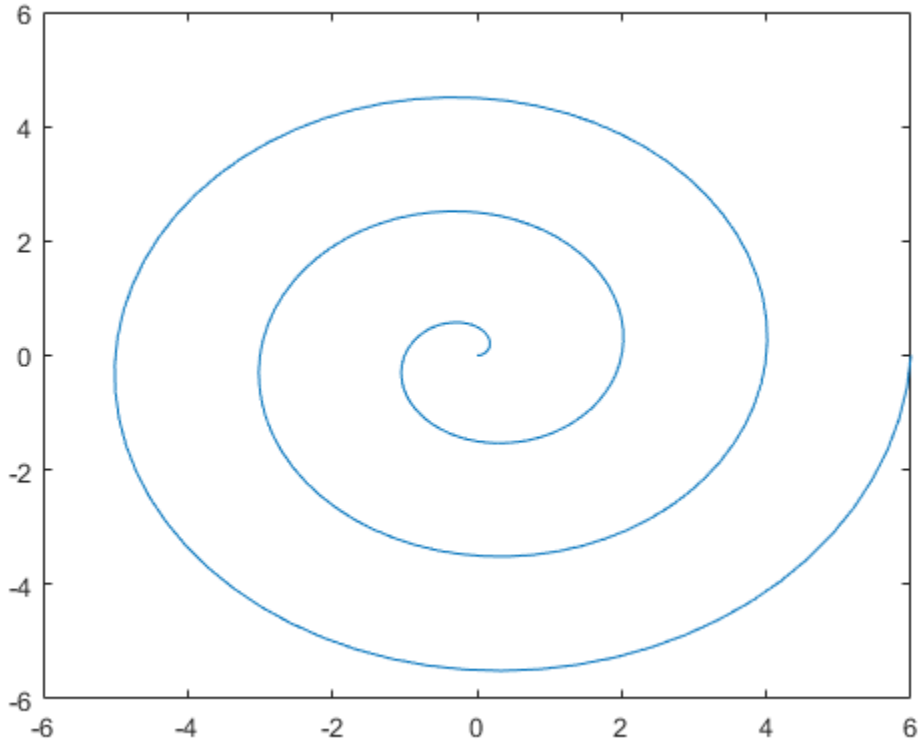
Examples

Correct Phase Angle of Spiral

Define the x- and y-coordinates of a spiral with phase angle from 0 to 6π. Plot the spiral.

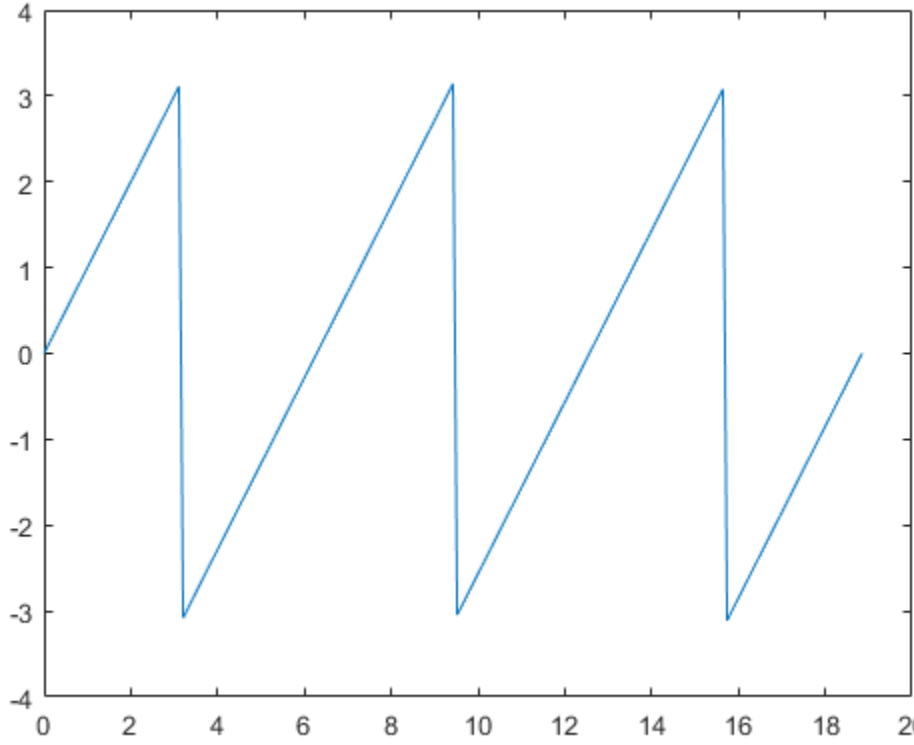
```
t = linspace(0,6*pi,201);
x = t/pi.*cos(t);
```

```
y = t/pi.*sin(t);
plot(x,y)
```



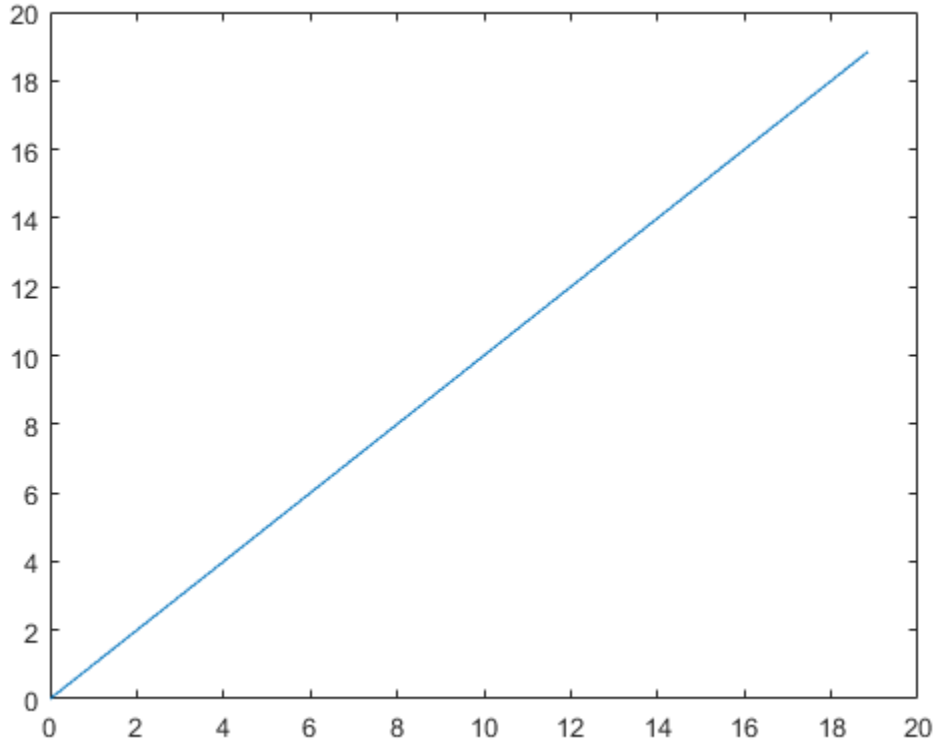
Find the phase angle of the spiral from the *x*- and *y*-coordinates using the `atan2` function. The `atan2` function returns the angle values within the closed interval from $-\pi$ to π.

```
P = atan2(y,x);
plot(t,P)
```



Note that this plot has discontinuities. Use `unwrap` to eliminate the discontinuities. `unwrap` adds multiples of $\pm 2\pi$ when the phase difference between consecutive elements of `P` are greater than or equal to the jump threshold π radians. The shifted phase angle `Q` lies in the interval from 0 to 6π.

```
Q = unwrap(P);
plot(t,Q)
```

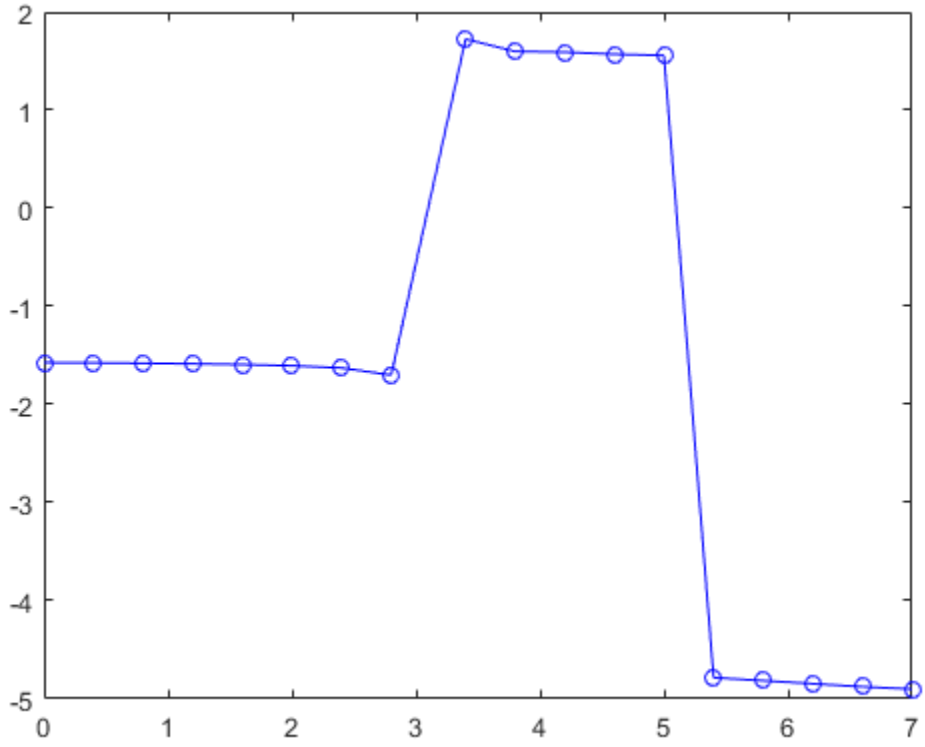


Shift Phase Angle with Different Thresholds

Shift the phase angle of a frequency response. The phase curve has two jumps. The first jump is `3.4250` radians between `W = 3` and `W = 3.4`, and the second jump is `6.3420` radians between `W = 5` and `W = 5.4`. Plot the phase curve.

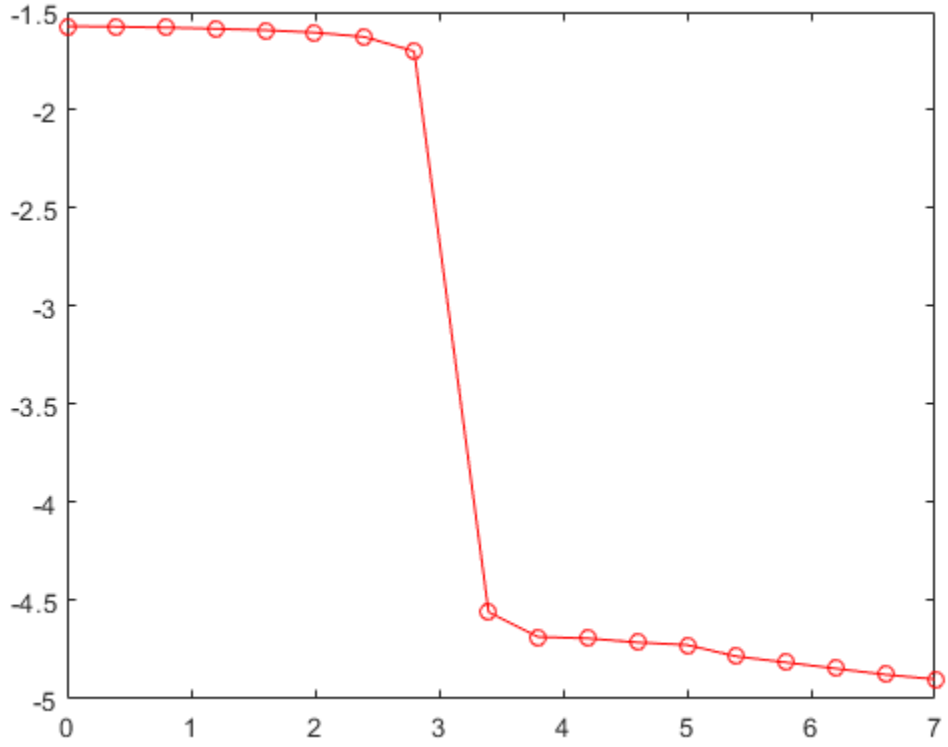
```
clear; close all;
W = [0:0.4:3, 3.4:0.4:5, 5.4:0.4:7];
P = [-1.5723
     -1.5747
     -1.5790
```

```
     -1.5852
     -1.5922
     -1.6044
     -1.6269
     -1.6998
      1.7252
      1.5989
      1.5916
      1.5708
      1.5582
     -4.7838
     -4.8143
     -4.8456
     -4.8764
     -4.9002];
plot(W,P,'bo-')
```



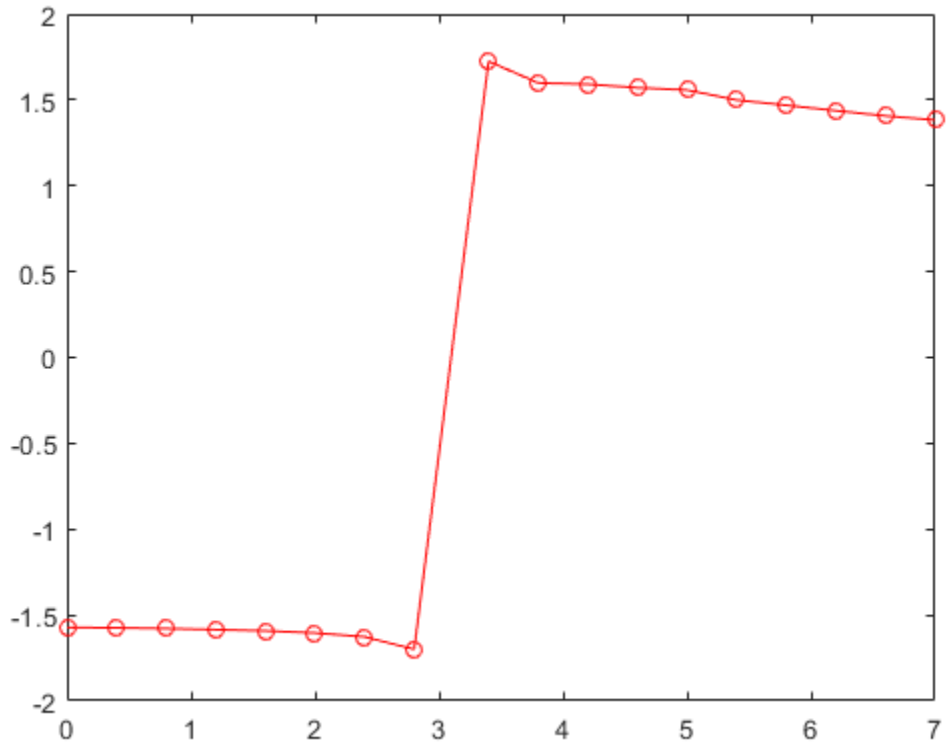
Use `unwrap` to shift the phase angle using the default jump threshold π radians. Plot the shifted phase curve. Both jumps are shifted since they are greater than the jump threshold π radians.

```
plot(W,unwrap(P),'ro-')
```



Now shift the phase angle using a jump threshold of 5 radians. Plot the shifted phase curve. The first jump is not shifted since it is less than the jump threshold 5 radians.

```
plot(W,unwrap(P,5),'ro-')
```



Apply Phase Shift to Matrix

Define a two-column matrix P that contains phase angles.

```
P = [0 7.07; 0.19 0.98; 6.67 1.18; 0.59 1.37; 0.78 1.56]
```

```
P = 5×2

         0    7.0700
    0.1900    0.9800
    6.6700    1.1800
    0.5900    1.3700
```

```
    0.7800    1.5600
```

The phase angles `P(1,2) = 7.07` and `P(3,1) = 6.67` have phase differences that are larger than π compared to the rest of the data.

Unwrap the phase angles by first comparing the elements columnwise. Specify the `dim` argument as 1. Use the default jump threshold π by specifying the second argument as `[]`.

```
dim = 1;
P1 = unwrap(P,[],dim)
```

```
P1 = 5×2

         0    7.0700
    0.1900    7.2632
    0.3868    7.4632
    0.5900    7.6532
    0.7800    7.8432
```

To shift phase angles by rows instead of by columns, specify `dim` as 2 instead of 1.

```
dim = 2;
P2 = unwrap(P1,[],dim)
```

```
P2 = 5×2

         0    0.7868
    0.1900    0.9800
    0.3868    1.1800
    0.5900    1.3700
    0.7800    1.5600
```

Input Arguments

P — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `single` | `double`

`tol` — Jump threshold to apply phase shift
`pi` (default) | scalar

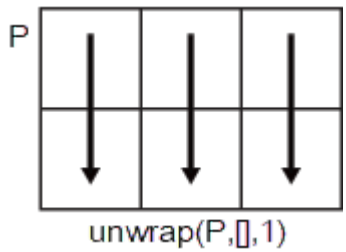
Jump threshold to apply phase shift, specified as a scalar. A jump threshold less than π has the same effect as the default threshold π.

Data Types: `single` | `double`

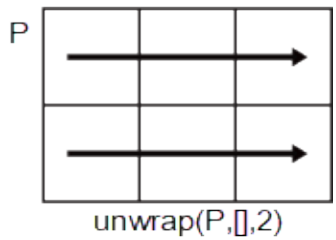
`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

- `unwrap(P,[],1)` operates along the columns of `P` and returns the shifted phase angle of each column.



- `unwrap(P,[],2)` operates along the rows of `P` and returns the shifted phase angle of each row.



If `dim` is greater than `ndims(P)`, then `unwrap(P,[],dim)` returns `P`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`Q` — Shifted phase angle
vector | matrix | multidimensional array

Shifted phase angle, returned as a vector, matrix, or multidimensional array. The size of the output `Q` is always the same as the size of the input `P`.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Row vector input is only supported when the first two inputs are vectors and nonscalar.
- Performs arithmetic in the output class. Therefore, results might not match MATLAB due to different rounding errors.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`abs` | `angle` | `atan` | `atan2`

Introduced before R2006a

unzip

Extract contents of zip file

Syntax

```
unzip(zipfilename)
unzip(zipfilename,outputfolder)
filenames = unzip( ___ )
```

Description

`unzip(zipfilename)` extracts the archived contents of `zipfilename` into the current folder, preserving the attributes and timestamps of each file. `unzip` can extract files from your local system or from an Internet URL.

If a file exists with the same name and the file is not read-only, MATLAB overwrites it. Otherwise, MATLAB issues a warning. On Microsoft Windows platforms, the hidden, system, and archive attributes are not set.

`unzip(zipfilename,outputfolder)` extracts `zipfilename` into `outputfolder`. If `outputfolder` does not exist, MATLAB creates it.

`filenames = unzip( ___ )` returns a cell array of character vectors containing the names of the extracted files.

Examples

Extract Zip File to New Folder

Create a zip file and extract it to the folder `archive`.

Create the zip file `examples.zip` containing example MAT-files.

```
zip('examples.zip','*.mat',...
    fullfile(matlabroot,'toolbox','matlab','demos'))
```

Extract `examples.zip` to the folder `archive`.

```
exampleFiles = unzip('examples.zip','archive')
```

```
exampleFiles = 1×40 cell array
    {'archive\accidents.mat'}    {'archive\airfoil.mat'}    {'archive\airlineResults.ma
```

Extract Zip File from URL

Download and extract a zip file from a URL to a local folder.

Suppose you have the zip file `example.zip` stored at the URL `http://example.com/example.zip`. Download and extract the file to the `example` folder.

```
url = 'http://example.com/example.zip';
unzip(url);
```

Input Arguments

`zipfilename` — Name of zip file
character vector | string scalar

Name of zip file to extract from, specified as a character vector or a string scalar. If `zipfilename` has no extension, MATLAB searches for `zipfilename` appended with `.zip`.

`zipfilename` must include a path relative to the current folder or an absolute path.

If `zipfilename` is a URL, `zipfilename` must include the protocol type (for example, `http://`). MATLAB downloads the URL to the temporary folder on your system, and then it deletes the URL on cleanup.

`unzip` does not support password-protected or encrypted zip archives.

Data Types: `char` | `string`

`outputfolder` — Target folder
character vector | string scalar

Target folder for the extracted files, specified as a character vector or a string scalar.

Data Types: `char` | `string`

Tips

- To extract a zip file that contains non-7-bit ASCII characters, extract the file on a machine that has the appropriate language/encoding settings.

See Also

`gunzip` | `gzip` | `tar` | `unzip` | `zip`

Topics

"Create and Extract from Zip Archives"

Introduced before R2006a

uplus, +

Unary plus

Syntax

```
C = +A
C = uplus(A)
```

Description

`C = +A` returns array `A` and stores it in `C`.

`C = uplus(A)` is an alternate way to execute `+A`, but is rarely used. It enables operator overloading for classes.

Examples

Unary Plus of Matrix

Create a 2-by-2 matrix, `A`.

```
A = [1 -3; -2 4]
```

```
A = 2×2

     1    -3
    -2     4
```

Use unary plus on `A`.

```
C = +A
```

```
C = 2×2
```

```
     1    -3
    -2     4
```

`C` and `A` are the same.

Input Arguments

`A` — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `duration` | `calendarDuration`
Complex Number Support: Yes

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`plus` | `uminus`

Topics

“Array vs. Matrix Operations”
“Operator Precedence”
“MATLAB Operators and Special Characters”

Introduced before R2006a

upper

Convert strings to uppercase

Syntax

```
newStr = upper(str)
```

Description

`newStr = upper(str)` converts all lowercase characters in `str` to the corresponding uppercase characters and leaves all other characters unchanged.

Examples

Convert Character Vector to Uppercase

```
upper('Hello, World.')
```

```
ans =
'HELLO, WORLD.'
```

Convert String Array to Uppercase

Starting in R2017a, you can create string arrays using double quotes.

Convert a string array to contain uppercase characters.

```
str = ["The SOONER,";"the BETTER."]
```

```
str = 2x1 string array
    "The SOONER,"
    "the BETTER."
```

```
newStr = upper(str)
```

```
newStr = 2x1 string array
    "THE SOONER,"
    "THE BETTER."
```

Input Arguments

str — Input array
string array | character array | cell array of character vectors

Input array, specified as a string array, character array, or cell array of character vectors.

Tips

For character arrays, the `upper` function supports the character sets:

- PC: Windows Latin-1
- Other: ISO Latin-1 (ISO 8859-1)

For string arrays, the `upper` function supports Unicode.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Input array must be a string scalar or a character array.
- Input values must be in the range 0–127.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`char` | `isstrprop` | `lower` | `reverse` | `string`

Topics

“Text in String and Character Arrays”
“Create String Arrays”
“Analyze Text Data with String Arrays”
“Search and Replace Text”

Introduced before R2006a

urlread

Download URL content to character vector (not recommended)

Note `urlread` is not recommended. Use `webread` or `webwrite` instead.

Syntax

```
str = urlread(URL)
str = urlread(URL,Name,Value)
[str,status] = urlread( ___ )
```

Description

`str = urlread(URL)` downloads the HTML web content from the specified `URL` into the character vector `str`. `urlread` does not retrieve hyperlink targets and images.

`str = urlread(URL,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`[str,status] = urlread( ___ )` suppresses the display of error messages, using any of the input arguments in the previous syntaxes. When the operation is successful, `status` is `1`. Otherwise, `status` is `0`

Examples

Download Web Content by Specifying Complete URL

Download the HTML for the page on the MATLAB® Central File Exchange that lists submissions related to `urlread`.

```
fullURL = ['https://www.mathworks.com/matlabcentral/fileexchange' ...
       '?term=urlread'];
str = urlread(fullURL);
```

`urlread` reads from the specified URL and downloads the HTML content to the character vector `str`.

Download Web Content Related to Term

Download the HTML for the page on the MATLAB® Central File Exchange that lists submissions related to `urlread`.

```
URL = 'https://www.mathworks.com/matlabcentral/fileexchange';
str = urlread(URL,'Get',{'term','urlread'});
```

`urlread` reads from `https://www.mathworks.com/matlabcentral/fileexchange/?term=urlread` and downloads the HTML content to the character vector `str`.

Specify Timeout Duration

Download content from a page on the MATLAB® Central File Exchange as in the first example, and specify a timeout duration of 5 seconds.

```
fullURL = ['https://www.mathworks.com/matlabcentral/fileexchange' ...
       '?term=urlread'];
str = urlread(fullURL,'Timeout',5);
```

Input Arguments

URL — Content location
character vector

Content location, specified as a character vector. Include the transfer protocol, such as `http`, `ftp`, or `file`.

Example: `'https://www.mathworks.com/matlabcentral'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Timeout',10,'Charset','UTF-8'` specifies that `urlread` should time out after 10 seconds, and the character encoding of the file is UTF-8.

`Get` — Data to send to the web form using the GET method
cell array

Parameters of the data to send to the web form using the GET method, specified as the comma-separated pair consisting of `'get'` and a cell array of paired parameter names and values. The supported parameters depend upon the URL.

`'Get'` includes the data in the URL, separated by ? and & characters.

Example: `'Get',{'term','urlread'}`

`Post` — Data to send to the web form using the POST method
cell array

Parameters of the data to send to the web form using the POST method, specified as the comma-separated pair consisting of `'post'` and a cell array of paired parameter names and values. The supported parameters depend upon the URL.

`'Post'` submits the data as part of the request headers, not explicitly in the URL.

`Charset` — Character encoding
character vector

Character encoding, specified as the comma-separated pair consisting of `'Charset'` and a character vector. If you do not specify `Charset`, the function attempts to determine the character encoding from the headers of the file. If the character encoding cannot be determined, `Charset` defaults to the native encoding for the file protocol, and UTF-8 for all other protocols.

Example: `'Charset','ISO-8859-1'`

`Timeout` — Timeout duration
scalar

Timeout duration in seconds, specified as the comma-separated pair consisting of `'Timeout'` and a scalar. The timeout duration determines when the function errors rather than continues to wait for the server to respond or send data.

Example: `'Timeout',10`

`UserAgent` — Client user agent identification
character vector

Client user agent identification, specified as the comma-separated pair consisting of `'UserAgent'` and a character vector.

Example: `'UserAgent','MATLAB R2012b'`

`Authentication` — HTTP authentication mechanism
`'Basic'`

HTTP authentication mechanism, specified as the comma-separated pair consisting of `'Authentication'` and a character vector. Currently, only the value `'Basic'` is supported. `'Authentication','Basic'` specifies basic authentication.

If you include the `Authentication` argument, you must also include the `Username` and `Password` arguments.

`Username` — User identifier
character vector

User identifier, specified as the comma-separated pair consisting of `'Username'` and a character vector. If you include the `Username` argument, you must also include the `Password` and `Authentication` arguments.

Example: `'Username','myName'`

`Password` — User authentication password
character vector

User authentication password, specified as the comma-separated pair consisting of `'Password'` and a character vector. If you include the `Password` argument, you must also include the `Username` and `Authentication` arguments.

Example: `'Password','myPassword123'`

Output Arguments

`str` — Contents of the file at the specified URL
character vector

Contents of the file at the specified URL, returned as a character vector. For example, if the URL corresponds to an HTML page, `str` contains the text and markup in the HTML file. If the URL corresponds to a binary file, `str` is not readable.

`status` — Download status
`1` | `0`

Download status, returned as either `1` or `0`. When the download is successful, `status` is `1`. Otherwise, `status` is `0`.

Tips

- `urlread` saves web content to a character vector. To save content to a file, use `urlwrite`.
- `urlread` and `urlwrite` can download content from FTP sites. Alternatively, use the `ftp` function to connect to an FTP server and the `mget` function to download a file.

See Also

`ftp` | `mget` | `urlwrite` | `web` | `webread` | `webwrite`

Topics

"Specify Proxy Server Settings for Connecting to the Internet"

Introduced before R2006a

urlwrite

Download URL content and save to file (not recommended)

Note `urlwrite` is not recommended. Use `websave` instead.

Syntax

```
urlwrite(URL,filename)
urlwrite(URL,filename,Name,Value)
```

```
[filestr,status] = urlwrite( ___ )
```

Description

`urlwrite(URL,filename)` reads web content at the specified `URL` and saves it to the file specified by `filename`.

`urlwrite(URL,filename,Name,Value)`uses additional options specified by one or more `Name,Value` pair arguments.

`[filestr,status] = urlwrite( ___ )` stores the file path in variable `filestr`, and suppresses the display of error messages, using any of the input arguments in the previous syntaxes. When the operation is successful, `status` is `1`. Otherwise, `status` is `0`.

Examples

Download Web Content by Specifying Complete URL

Download the HTML for the page on the MATLAB® Central File Exchange that lists submissions related to `urlwrite`. Save the results to `samples.html` in the current folder.

```
fullURL = ['https://www.mathworks.com/matlabcentral/fileexchange' ...
           '?term=urlwrite'];
filename = 'samples.html';
urlwrite(fullURL,filename);
```

View the file.

```
web(filename)
```

Download Web Content Related to Term

Download the HTML for the page on the MATLAB® Central File Exchange that lists submissions related to `urlwrite`. Save the results to `samples.html` in the current folder.

```
URL = 'https://www.mathworks.com/matlabcentral/fileexchange';
filename = 'samples.html';
urlwrite(URL,filename,'get',{'term','urlwrite'});
```

`urlwrite` downloads the HTML content from `https://www.mathworks.com/matlabcentral/fileexchange/?term=urlwrite` and writes it to `samples.html`.

Specify Timeout Duration

Download content from a page on the MATLAB® Central File Exchange as in the first example, and specify a timeout duration of 5 seconds.

```
fullURL = ['https://www.mathworks.com/matlabcentral/fileexchange' ...
       '?term=urlwrite'];
filename = 'samples.html';
urlwrite(fullURL,filename,'Timeout',5);
```

Input Arguments

URL — Content location
character vector

Content location, specified as a character vector. Include the transfer protocol, such as `http`, `ftp`, or `file`.

Example: `'https://www.mathworks.com/matlabcentral'`

`filename` — Name of file to store web content
character vector

Name of the file to store the web content, specified as a character vector. If you do not specify the path for `filename`, `urlwrite` saves the file in the current folder.

Example: `'myfile.html'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Timeout',10,'Charset','UTF-8'` specifies that `urlread` should time out after 10 seconds, and the character encoding of the file is UTF-8.

`Get` — Data to send to the web form using the GET method
cell array

Parameters of the data to send to the web form using the GET method, specified as the comma-separated pair consisting of `'get'` and a cell array of paired parameter names and values. The supported parameters depend upon the URL.

`'Get'` includes the data in the URL, separated by ? and & characters.

Example: `'Get',{'term','urlread'}`

`Post` — Data to send to the web form using the POST method
cell array

Parameters of the data to send to the web form using the POST method, specified as the comma-separated pair consisting of `'post'` and a cell array of paired parameter names and values. The supported parameters depend upon the URL.

`'Post'` submits the data as part of the request headers, not explicitly in the URL.

`Charset` — Character encoding
character vector

Character encoding, specified as the comma-separated pair consisting of `'Charset'` and a character vector. If you do not specify `Charset`, the function attempts to determine the character encoding from the headers of the file. If the character encoding cannot be determined, `Charset` defaults to the native encoding for the file protocol, and UTF-8 for all other protocols.

Example: `'Charset','ISO-8859-1'`

`Timeout` — Timeout duration
scalar

Timeout duration in seconds, specified as the comma-separated pair consisting of `'Timeout'` and a scalar. The timeout duration determines when the function errors rather than continues to wait for the server to respond or send data.

Example: `'Timeout',10`

`UserAgent` — Client user agent identification
character vector

Client user agent identification, specified as the comma-separated pair consisting of `'UserAgent'` and a character vector.

Example: `'UserAgent','MATLAB R2012b'`

`Authentication` — HTTP authentication mechanism
`'Basic'`

HTTP authentication mechanism, specified as the comma-separated pair consisting of `'Authentication'` and a character vector. Currently, only the value `'Basic'` is supported. `'Authentication','Basic'` specifies basic authentication.

If you include the `Authentication` argument, you must also include the `Username` and `Password` arguments.

`Username` — User identifier
character vector

User identifier, specified as the comma-separated pair consisting of `'Username'` and a character vector. If you include the `Username` argument, you must also include the `Password` and `Authentication` arguments.

Example: `'Username','myName'`

`Password` — User authentication password
character vector

User authentication password, specified as the comma-separated pair consisting of `'Password'` and a character vector. If you include the `Password` argument, you must also include the `Username` and `Authentication` arguments.

Example: `'Password','myPassword123'`

Output Arguments

`filestr` — Path of the file
character vector

Path of the file specified by `filename`, returned as a character vector.

`status` — Download status
`1` | `0`

Download status, returned as either `1` or `0`. When the download is successful, `status` is `1`. Otherwise, `status` is `0`.

Tips

- `urlread` and `urlwrite` can download content from FTP sites. Alternatively, use the `ftp` function to connect to an FTP server and the `mget` function to download a file.

See Also

`ftp` | `mget` | `urlread` | `web` | `websave`

Topics

"Specify Proxy Server Settings for Connecting to the Internet"

Introduced before R2006a

usejava

Determine if Java feature is available

Syntax

```
tf = usejava(feature)
```

Description

`tf = usejava(feature)` returns logical `1` (`true`) if the specified feature is supported. Otherwise, it returns logical `0` (`false`). If the Java feature is unavailable, then you can use this function for error handling.

Examples

Display Error Message

Use the following code snippet to test that the AWT GUI components are available before attempting to display a Java Frame. If the AWT is not available on your system, MATLAB displays the message `Unable to open a Java Frame`.

```
if usejava('awt')
   myFrame = java.awt.Frame;
else
   disp('Unable to open a Java Frame')
end
```

Call error Function

Use the following code snippet to terminate a script if MATLAB does not have access to JVM software.

The variable, `filename`, is a function that contains Java code.

```
if ~usejava('jvm')
   error([filename ' requires Java to run.'])
end
```

Input Arguments

feature — Java feature
`'awt'` | `'desktop'` | `'jvm'` | `'swing'`

Java feature, specified as one of these values.

`'awt'`	UI components in the Java Abstract Window Toolkit (AWT) are available.
`'desktop'`	MATLAB interactive desktop is running.
`'jvm'`	Java Virtual Machine software (JVM) is running.
`'swing'`	Swing components (Java lightweight UI components in the Java Foundation Classes) are available.

See Also

`error` | `javachk`

Topics

"Java Class Path"

Introduced before R2006a

userpath

View or change default user work folder

Syntax

```
userpath
```

```
userpath(newpath)
userpath('reset')
userpath('clear')
```

Description

`userpath` returns the first folder on the search path, specified as a character vector. MATLAB adds the folder specified by `userpath` to the search path at startup.

`userpath(newpath)` sets the first folder on the search path to `newpath`. The folder specified using `userpath` appears at the top of the search path immediately, and at startup in future sessions. MATLAB removes the folder previously specified by `userpath` from the search path.

`userpath('reset')` sets the first folder on the search path to the default for your platform. The default *`userpath`* folder is platform-specific.

- Windows platforms — `%USERPROFILE%/Documents/MATLAB`.
- Mac platforms — `$home/Documents/MATLAB`.
- Linux platforms — `$home/Documents/MATLAB` if `$home/Documents` exists.

MATLAB immediately adds the default folder to the top of the search path, and also adds it to the search path at startup in future sessions. On Windows and Mac platforms, the default folder is created if it does not exist. On Linux, the default folder is not created if it does not exist.

`userpath('clear')` removes the first folder from the search path immediately, and for future MATLAB sessions.

Examples

View *userpath* Folder

This example assumes the *userpath* folder is set to the default value `Documents\MATLAB` on the Windows platform. Start MATLAB and display the current folder:

```
pwd
```

```
H:\Documents\MATLAB
```

In this example, `H` is the drive at which `Documents` is located.

Confirm that the current folder is the *userpath* folder.

```
userpath
```

```
H:\Documents\MATLAB
```

Display the search path. MATLAB returns the search path. The folder specified by `userpath` is at the top.

```
path
```

```
MATLABPATH

    H:\Documents\MATLAB
    C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
    C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
...
```

Set New Value for *userpath*

Assume the *userpath* folder is set to the default value on the Windows platform, `Documents\MATLAB`.

Change the value from the default to `C:\Research_Project`.

```
newpath = 'C:\Research_Project';
userpath(newpath)
```

View the effect of the change on the search path.

```
path
```

```
MATLABPATH

    C:\Research_Project
    C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
    C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
...
```

MATLAB displays the search path, with the folder specified by `userpath` at the top. MATLAB automatically removes the previous value of the *`userpath`* folder, `H:\Documents\MATLAB`, from the search path when you assign a new folder using `userpath`.

Clear the Value for *`userpath`*

Assume that the *`userpath`* folder is set to the default value, but you do not want it to be added to the search path at startup.

Confirm that the default is set.

```
userpath
```

```
H:\Documents\MATLAB
```

Verify that it is at the top of the search path.

```
path
```

```
MATLABPATH

H:\Documents\MATLAB
C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
...
```

Clear the value.

```
userpath('clear')
```

Verify the result.

```
userpath
```

```
ans =
     ''
```

Confirm that the former *userpath* folder is removed from the search path.

```
path
```

```
MATLABPATH

C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
...
```

Note If you use `userpath('clear')`, the startup folder will not necessarily be on the search path. This can also occur if you remove the *userpath* folder from the search path and save the changes.

Input Arguments

`newpath` — New value for the userpath folder
character vector | string scalar

New value for the *userpath* folder, specified as a character vector or string scalar. `newpath` must be an absolute path.

Example: `'C:\myFolder'`

Data Types: `char` | `string`

Limitations

- In MATLAB Online, you only can view the folder specified by `userpath`. Changing the folder is not supported.

Tips

- To specify the startup folder, set the **Initial working folder** preference, described in “General Preferences”.

- To add a folder to the top of the search path, specify the path for the folder in the `MATLABPATH` environment variable. MATLAB adds the paths specified in the `MATLABPATH` environment variable to the search path below the *`userpath`* folder at startup.

See Also

`addpath` | `path` | `rmpath` | `savepath` | `startup`

Topics

"What Is the MATLAB Search Path?"
"MATLAB Startup Folder"

Introduced in R2008a

ValueIterator

An iterator over intermediate values for use with mapreduce

Description

The `mapreduce` function automatically creates a `ValueIterator` object during execution and uses it to store the values associated with each unique intermediate key added by the map function. Although you never need to explicitly create a `ValueIterator` object to use `mapreduce`, you do need to interact with this object in the reduce function. Use the `hasnext` and `getnext` object functions to retrieve the values associated with each unique key in the intermediate `KeyValueStore` object.

Creation

The `mapreduce` function automatically creates `ValueIterator` objects during execution.

Properties

Key — Intermediate key
numeric scalar | character vector

This property is read-only.

Intermediate key, specified as a numeric scalar or character vector. `Key` is one of the unique keys added by a map function. All the values in the `ValueIterator` object are associated with this key.

Object Functions

hasnext	Determine if ValueIterator has one or more values available
getnext	Get next value from ValueIterator

Examples

Get Values from ValueIterator in Reduce Function

Use the `hasnext` and `getnext` functions in a `while` loop within the reduce function to iteratively get values from the `ValueIterator`. For example,

```
function MeanDistReduceFun(sumLenKey, sumLenIter, outKVStore)
    sumLen = [0, 0];
    while hasnext(sumLenIter)
        sumLen = sumLen + getnext(sumLenIter);
    end
    add(outKVStore, 'Mean', sumLen(1)/sumLen(2));
end
```

Always call `hasnext` before `getnext` to confirm availability of a value. `mapreduce` returns an error if you call `getnext` with no remaining values in the `ValueIterator`.

See Also

`mapreduce`

Topics

"Getting Started with MapReduce"
`KeyValueStore`

Introduced in R2014b

validateattributes

Check validity of array

Syntax

```
validateattributes(A,classes,attributes)
validateattributes(A,classes,attributes,argIndex)
validateattributes(A,classes,attributes,funcName)
validateattributes(A,classes,attributes,funcName,varName)
validateattributes(A,classes,attributes,funcName,varName,argIndex)
```

Description

`validateattributes(A,classes,attributes)` validates that array `A` belongs to at least one of the specified classes (or its subclass) and has all the specified attributes. If `A` does not meet the criteria, then MATLAB throws an error and displays a formatted error message. Otherwise, `validateattributes` completes without displaying any output.

`validateattributes(A,classes,attributes,argIndex)` includes the position of the input in your function argument list as part of any generated error messages.

`validateattributes(A,classes,attributes,funcName)` includes the specified function name in generated error identifiers.

`validateattributes(A,classes,attributes,funcName,varName)` includes the specified variable name in generated error messages.

`validateattributes(A,classes,attributes,funcName,varName,argIndex)` includes the specified information in generated error messages or identifiers.

Examples

Validate Array Size

```
classes = {'numeric'};
attributes = {'size',[4,6,2]};

A = rand(3,5,2);
validateattributes(A,classes,attributes)
```

```
Expected input to be of size 4x6x2 when it is actually size 3x5x2.
```

Because A did not match the specified attributes, MATLAB throws an error message.

Validate Array Monotonicity

Determine if an array is increasing or nondecreasing.

```
A = [1 5 8 2;
     9 6 9 4]
validateattributes(A, {'double'},{'nondecreasing'})
validateattributes(A, {'double'},{'increasing'})
```

```
A =

     1     5     8     2
     9     6     9     4
```

Since A is both increasing and nondecreasing, `validateattributes` does not throw an error for either attribute check.

Setting `A(2,3)` equal to `A(1,3)` results in a column that is no longer strictly increasing, so `validateattributes` throws an error.

```
A(2,3) = 8
validateattributes(A, {'double'},{'increasing'})
```

```
A =

     1     5     8     2
     9     6     8     4

Expected input to be strictly increasing.
```

However, the columns remain nondecreasing since each column element is equal to or greater than the previous column element. The following code does not throw an error.

```
validateattributes(A, {'double'},{'nondecreasing'})
```

Check Complex Number Attributes

Assuming that a is the second input argument to a function, check that it is nonnegative.

```
a = complex(1,1);
validateattributes(a,{'numeric'},{'nonnegative'},2)
```

```
Expected input number 2 to be nonnegative.
```

Because complex numbers lack a well-defined ordering in the complex plane, `validateattributes` does not recognize them as positive or negative.

Ensure Array Values Are Within Specified Range

Check that the values in an array are 8-bit integers from 0 through 10.

Assume that this code occurs in a function called `Rankings`.

```
classes = {'uint8','int8'};
attributes = {'>',0,'<',10};
funcName = 'Rankings';
A = int8(magic(4));

validateattributes(A,classes,attributes,funcName)
```

```
Error using Rankings
Expected input to be an array with all of the values < 10.
```

Validate Function Input Parameters Using `inputParser`

Create a custom function that checks input parameters with `inputParser`, and use `validateattributes` as the validating function for the `addRequired` and `addOptional` methods.

Define the function.

```
function a = findArea(shape,dim1,varargin)
   p = inputParser;
```

```
   charchk = {'char'};
   numchk = {'numeric'};
   nempty = {'nonempty'};

   addRequired(p,'shape',@(x)validateattributes(x,charchk,nempty))
   addRequired(p,'dim1',@(x)validateattributes(x,numchk,nempty))
   addOptional(p,'dim2',1,@(x)validateattributes(x,numchk,nempty))
   parse(p,shape,dim1,varargin{:})

   switch shape
      case 'circle'
         a = pi * dim1.^2;
      case 'rectangle'
         a = dim1 .* p.Results.dim2;
   end
end
```

Call the function with a nonnumeric third input.

```
myarea = findArea('rectangle',3,'x')
```

```
Error using findArea (line 10)
The value of 'dim2' is invalid. Expected input to be one of these types:

double, single, uint8, uint16, uint32, uint64, int8, int16, int32, int64
```

Validate Function Arguments

Check the inputs of a function and include information about the input name and position in generated error.

Define the function.

```
function v = findVolume(shape,ht,wd,ln)
   validateattributes(shape,{'char'},{'nonempty'},mfilename,'Shape',1)
   validateattributes(ht,{'numeric'},{'nonempty'},mfilename,'Height',2)
   validateattributes(wd,{'numeric'},{'nonempty'},mfilename,'Width',3)
   validateattributes(ln,{'numeric'},{'nonempty'},mfilename,'Length',4)
```

Call the function without the `shape` input argument.

```
vol = findVolume(10,7,4)
```

```
Error using findVolume
Expected input number 1, Shape, to be one of these types:

char

Instead its type was double.

Error in findVolume (line 2)
validateattributes(shape,{'char'},{'nonempty'},mfilename,'Shape',1)
```

The function name becomes part of the error identifier.

```
MException.last.identifier
```

```
ans =

MATLAB:findVolume:invalidType
```

Input Arguments

A — Input

any type of array

Input, specified as any type of array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `cell` | `function_handle`
Complex Number Support: Yes

`classes` — Valid data types

cell array of character vectors | string array

Valid data types, specified as a cell array of character vectors or a string array. Each element of `classes` can be the name of any built-in or custom class, including:

`'single'`	Single-precision number
`'double'`	Double-precision number
`'int8'`	Signed 8-bit integer
`'int16'`	Signed 16-bit integer
`'int32'`	Signed 32-bit integer

`'int64'`	Signed 64-bit integer
`'uint8'`	Unsigned 8-bit integer
`'uint16'`	Unsigned 16-bit integer
`'uint32'`	Unsigned 32-bit integer
`'uint64'`	Unsigned 64-bit integer
`'logical'`	Logical `1` (`true`) or `0` (`false`)
`'char'`	Character
`'string'`	String array
`'struct'`	Structure array
`'cell'`	Cell array
`'table'`	Table
`'function_handle'`	Function handle

`'numeric'`	Any data type for which the `isa(A,'numeric')` function returns true, including `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, `single`, or `double`
`'<class_name>'`	Any other class name

Data Types: `cell` | `string`

`attributes` — Valid attributes
cell array | string array

Valid attributes, specified as a cell array or a string array.

Some attributes also require numeric values, such as attributes that specify the size or number of elements of `A`. For these attributes, the numeric value or vector must immediately follow the attribute name in a cell array. A string array cannot be used to represent numeric values in `attributes`.

These attributes describe the size and shape of array `A`.

`'2d'`	Two-dimensional array, including scalars, vectors, matrices, and empty arrays
`'3d'`	Array with three or fewer dimensions

`'column'`	Column vector, `N`-by-1
`'row'`	Row vector, 1-by-`N`
`'scalar'`	Scalar value, 1-by-1
`'scalartext'`	Either a string scalar or a character vector, including inputs with zero characters
`'vector'`	Row or column vector, or a scalar value
`'size', [d1,...,dN]`	Array with dimensions `d1`-by-...-by-`dN`. To skip checking a particular dimension, specify `NaN` for that dimension, such as `[3,4,NaN,2]`.
`'numel', N`	Array with `N` elements
`'ncols', N`	Array with `N` columns
`'nrows', N`	Array with `N` rows
`'ndims', N`	`N`-dimensional array
`'square'`	Square matrix; in other words, a two-dimensional array with equal number of rows and columns
`'diag'`	Diagonal matrix
`'nonempty'`	No dimensions that equal zero
`'nonsparse'`	Array that is not sparse

These attributes specify valid ranges for values in `A`.

`'>', N`	All values greater than `N`
`'>=', N`	All values greater than or equal to `N`
`'<', N`	All values less than `N`
`'<=', N`	All values less than or equal to `N`
`'finite'`	All values are finite
`'nonnan'`	No values are NaN (Not a Number)

These attributes check types of values in a numeric or logical array, `A`.

`'binary'`	Array of ones and zeros
`'even'`	Array of even integers (includes zero)

`'odd'`	Array of odd integers
`'integer'`	Array of integer values
`'real'`	Array of real values
`'nonnegative'`	No element is less than zero
`'nonzero'`	No element is equal to zero
`'positive'`	No element is less than or equal to zero
`'decreasing'`	Each element of a column is less than the previous element and no element is `NaN`.
`'increasing'`	Each element of a column is greater than the previous element and no element is `NaN`.
`'nondecreasing'`	Each element of a column is greater than or equal to the previous element and no element is `NaN`.
`'nonincreasing'`	Each element of a column is less than or equal to the previous element and no element is `NaN`.

Data Types: `cell`

`funcName` — Name of function for validation
character vector | string scalar

Name of function for validation, specified as a character vector or as a string scalar. If you specify an empty character vector, `''`, or the `<missing>` string, then the `validateattributes` function ignores the `funcName` input.

Data Types: `char` | `string`

`varName` — Name of input variable
character vector | string scalar

Name of input variable, specified as a character vector or a string scalar. If you specify an empty character vector, `''`, or the `<missing>` string, then the `validateattributes` function ignores the `varName` input.

Data Types: `char` | `string`

`argIndex` — Position of input argument
positive integer

Position of input argument, specified as a positive integer.

Data Types: `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Some error messages are a simplified version of the MATLAB error message.
- The `classes`, `funcName`, `varName`, and `argIndex` arguments must be constant.
- Attribute names must be constant.
- In the generated code, the format of numbers in an error message might be different from the format in MATLAB. For example, here is an error message in MATLAB:

  ```
  Expected input to be an array with all of the values > 3.
  ```

  Here is the error message in the generated code:

  ```
  Expected input to be an array with all of the values >   3.000000000000000e+00.
  ```

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`arguments` | `inputParser` | `is*` | `isa` | `isnumeric` | `validatestring`

Topics

"Function Argument Validation"

Introduced in R2007b

validateFunctionSignaturesJSON

Validate `functionSignatures.json` files

Syntax

```
validateFunctionSignaturesJSON
validateFunctionSignaturesJSON(filenames)
T = validateFunctionSignaturesJSON( ___ )
```

Description

`validateFunctionSignaturesJSON` displays validation messages for the `functionSignatures.json` file in the current folder. Validation messages indicate the location of any invalid code and the reason the code is invalid. The line number in the message is a hyperlink that you can click to go directly to that line in the Editor.

JSON syntax errors in the `functionSignatures.json` file impact validation of the file. If `validateFunctionSignaturesJSON` reports JSON syntax errors, correct these errors and then revalidate the file.

For information on creating a `functionSignatures.json` file, see “Customize Code Suggestions and Completions”.

`validateFunctionSignaturesJSON(filenames)` validates the `functionSignatures.json` files in `filenames`. Use this syntax to validate function signature files in multiple folders.

`T = validateFunctionSignaturesJSON( ___ )` returns a table of results. You can use this syntax with any of the input argument combinations in the previous syntaxes.

Examples

Validate Function Signature File

Create the `myFunc` function in your current working folder.

```
function myFunc(reqA,reqB,varargin)
    NV1 = true;
    NV2 = 'Default';
    posA = [];

    if nargin > 3
        if rem(nargin,2)
            posA = varargin{1};
            V = varargin(2:end);
        else
            V = varargin;
        end
        for n = 1:2:size(V,2)
            switch V{n}
                case 'Name1'
                    NV1 = V{n+1};
                case 'Name2'
                    NV2 = V{n+1}
                otherwise
                    error('Error.')
            end
        end
    end
end
```

Create the following function signature file, as `functionSignatures.json`, in your current working folder. It contains three validation issues.

1. A property (attribute) is incorrect in the first argument object. The property should be `"name"` instead of `"argument"`.
2. The argument object for `in3` is in the incorrect order. You must specify positional arguments before name-value pairs.
3. There is an unnecessary comma after the last argument object. A comma in this position is a JSON syntax error.

```
{
  "_schemaVersion": "1.0.0",
  "myFunc":
  {
     "inputs":
     [
        {"argument":"input1",  "kind":"required", "type":["numeric"], "purpose":"ID of item"},
        {"name":"input2",  "kind":"positional", "type":["numeric"], "purpose":"# Items"},
        {"name":"Name1", "kind":"namevalue", "type":["logical","scalar"],"purpose":"Option"},
        {"name":"in3",  "kind":"positional", "type":["numeric"], "purpose":"Input Value"},
        {"name":"Name2", "kind":"namevalue", "type":["char", "choices={'Default','Choice1','Choice2'}"]},
     ]
```

```
  }
}
```

Validate the `functionSignatures.json` file. The validation function does not report the first two validation issues because it encounters a JSON syntax error.

```
validateFunctionSignaturesJSON
```

```
functionSignatures.json
========================
L 12 (C 6-7): JSON syntax error at line 12, column 6 (character 551): expected value but found ']'.
```

Remove the extra column at the end of line 12 and, to view the remaining validation issues, revalidate the file.

```
validateFunctionSignaturesJSON
```

```
functionSignatures.json
========================
L 7 (C 10-19): Invalid attribute "argument".
L 10 (C 32-43): "positional" argument must appear before all "namevalue" and "flag" arguments.
```

Correct the remaining issues.

```
{
  "_schemaVersion": "1.0.0",
  "myFunc":
  {
     "inputs":
     [
        {"name":"input1",  "kind":"required", "type":["numeric"], "purpose":"ID of item"},
        {"name":"input2",  "kind":"positional", "type":["numeric"], "purpose":"# Items"},
        {"name":"in3",  "kind":"positional", "type":["numeric"], "purpose":"Input Value"},
        {"name":"Name1", "kind":"namevalue", "type":["logical","scalar"],"purpose":"Option"},
        {"name":"Name2", "kind":"namevalue", "type":["char", "choices={'Default','Choice1','Choice2'}"]}
     ]
  }
}
```

Revalidate the `functionSignatures.json` file.

```
validateFunctionSignaturesJSON
```

```
validateFunctionSignaturesJSON completed without producing any messages.
```

Input Arguments

`filenames` — Path to `functionSignatures.json` files

string array | character vector | cell array of character vectors

Absolute or relative path to `functionSignatures.json` files, specified as a character vector, cell array of character vectors, or string array.

Example: `["ProjectA/functionSignatures.json" "ProjectB/functionSignatures.json"]`

Example: `'myFolder/functionSignatures.json'`

See Also

Topics

"Customize Code Suggestions and Completions"

Introduced in R2018b

validatestring

Check validity of text

Syntax

```
matchedStr = validatestring(str,validStrings)

matchedStr = validatestring(str,validStrings,argIdx)
matchedStr = validatestring(str,validStrings,funcName)
matchedStr = validatestring(str,validStrings,funcName,varName)
matchedStr = validatestring(str,validStrings,funcName,varName,
argIdx)
```

Description

`matchedStr = validatestring(str,validStrings)` checks the validity of `str` against `validStrings`. The text is valid if it is an unambiguous, case-insensitive match to any element in `validStrings`. The `validatestring` function supports partial matching of leading characters.

If `str` is valid, then `validatestring` returns the matched text. Otherwise, MATLAB throws an error.

`matchedStr = validatestring(str,validStrings,argIdx)` includes the position of the input in your function argument list as part of any generated error messages. Use this syntax to format any generated error messages.

`matchedStr = validatestring(str,validStrings,funcName)` includes the specified function name in generated error identifiers.

`matchedStr = validatestring(str,validStrings,funcName,varName)` includes the specified variable name in generated error messages. Use this syntax to format any generated error messages.

`matchedStr = validatestring(str,validStrings,funcName,varName, argIdx)` includes the position of the input in your function argument list as part of any generated error messages. Use this syntax to format any generated error messages.

Examples

Validate Text

Check if a string is in a set of valid values.

```
validStrings = ["wind","wonder","when"];
str = "wind";
validStr = validatestring(str,validStrings)
```

```
validStr =

    "wind"
```

Check if `"WON"` is in the set of valid values defined by `validStrings`. The string is a case-insensitive, partial-match to `"wonder"`.

```
str = "WON";
validStr = validatestring(str,validStrings)
```

```
validStr =

    "wonder"
```

If multiple partial matches exist and each string is not a substring of another, then `validatestring` throws an error.

```
validStrings = ["showcase","show up","showtimes"];
str = "show";
validStr = validatestring(str,validStrings)
```

```
Expected input to match one of these values:

'showcase', 'show up', 'showtimes'

The input, show, matched more than one valid value.
```

However, if multiple partial matches exist and each string is a substring of another, then `validatestring` returns the shortest match.

```
validStrings = ["righteously","right","righteous"];
str = "rig";
validStr = validatestring(str,validStrings)
```

```
validStr =

    "right"
```

Validate Text Input to Function

Create a function in a file named `findArea.m`. The validation for `shape` includes the position of the input in your function argument list as part of any generated error messages. The validation for `units` also includes the variable name (`'units'`) in the error message and the file name in the error identifier. Use the `mfilename` function to find the file name.

```
function a = findArea(shape,h,w,units)
    expectedShapes = {'square','rectangle','triangle'};
    expectedUnits = {'cm','m','in','ft','yds'};

    shapeName = validatestring(shape,expectedShapes,1);
    unitAbbrev = validatestring(units,expectedUnits,mfilename,'units',4);

    switch shapeName
        case {'square','rectangle'}
            a = h*w;
        case {'triangle'}
            a = h*w/2;
        otherwise
            error('Unknown shape passing validation.')
    end
end
```

Call the function with a valid shape name. The value of `'Rect'` is valid because it is a case-insensitive, partial match to `'rectangle'`.

```
a = findArea('Rect',10,3,'cm')
```

```
a =

    30
```

Call the function with an invalid shape name. The error message contains the position of the invalid text. Here, the invalid text is the first input argument.

```
a = findArea('octagon',7,13,'in')
```

```
Error using findArea (line 5)
Expected input number 1 to match one of these values:

'square', 'rectangle', 'triangle'

The input, 'octagon', did not match any of the valid values.
```

Call the function with an invalid unit. The error message contains the variable name and the position of the invalid text. The invalid text is the fourth input argument.

```
a = findArea('TRI',10,3,'mi')
```

```
Error using findArea (line 6)
Expected input number 4, units, to match one of these values:

'cm', 'm', 'in', 'ft', 'yds'

The input, 'mi', did not match any of the valid values.
```

Use `mException` to view the error identifier, which includes the file name.

```
id = MException.last.identifier
```

```
id =

    'MATLAB:findArea:unrecognizedStringChoice'
```

Input Arguments

`str` — Text to validate

string scalar | character vector

Text to validate, specified as a string scalar or a character vector.

Example: `'textToValidate'`

Example: `"otherTextToValidate"`

`validStrings` — Text to match

string array | cell array of character vectors

Text to match, specified as a string array or a cell array of character vectors.

Example: `["value1","value2"]`

Example: `{'val1','val2',val3'}`

`funcName` — Name of function
string scalar | character vector

Name of the function whose input to validate, specified as a string scalar or character vector. If you specify an empty character vector `''` or the `<missing>` string, then the `validatestring` function ignores the `funcName` input.

Example: `"myFunctionName"`

Example: Call to `mfilename` function, as in the code `validatestring(units,expectedUnits,mfilename)`

`varName` — Name of input variable
string scalar | character vector

Name of input variable to validate, specified as a string scalar or character vector. If you specify an empty character vector `''` or the `<missing>` string, then the `validatestring` function ignores the `varName` input.

Example: `"inputVariable1"`

Example: `'variableB'`

`argIdx` — Position of input argument
positive integer

Position of the input argument to validate, specified as a positive integer.

Output Arguments

`matchedStr` — Matched text
string scalar | character vector

Matched text, returned as a string scalar if `validStrings` is a string array or as a character vector if `validStrings` is a cell array of character vectors.

Example — Match 'ball' with . . .	Return Value	Type of Match
`ball`, `barn`, `bell`	`ball`	Exact match
`balloon`, `barn`	`balloon`	Partial match (leading characters)
`ballo`, `balloo`, `balloon`	`ballo` (shortest match)	Multiple partial matches where each character vector is a subset of another
`balloon`, `ballet`	Error	Multiple partial matches to unique character vectors
`barn`, `bell`	Error	No match

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- You must supply the third argument and it must be `funcName`.
- The `validStrings`, `funcName`, `varName`, and `argIdx` arguments must be constant.
- MATLAB and the generated code behave differently when multiple partial matches exist and each string is a substring of another string. In MATLAB, `validatestring` returns the shortest match. In the generated code, `validatestring` throws an error.

See Also

`arguments` | `inputParser` | `validateattributes`

Topics

"Ways to Parse Function Inputs"

Introduced in R2006b

values

Package: `containers`

Return values of `Map` object

Syntax

```
valueSet = values(M)
valueSet = values(M,keySet)
```

Description

`valueSet = values(M)` returns all the values of the input `Map` object as a cell array.

`valueSet = values(M,keySet)` returns the values that correspond to the keys specified in the cell array `keySet`. The output argument `valueSet` has the same size as `keySet`.

Examples

Values of Map

Create a `Map` object.

```
ids = [437 1089 2362];
names = {'Li, N.','Jones, R.','Sanchez, C.'};
M = containers.Map(ids,names)
```

```
M = 
  Map with properties:

        Count: 3
      KeyType: double
    ValueType: char
```

Return a cell array containing its values.

```
valueSet = values(M)
```

```
valueSet = 1x3 cell array
    {'Li, N.'}    {'Jones, R.'}    {'Sanchez, C.'}
```

Values Corresponding to Specified Keys

Create a `Map` object.

```
months = {'Jan','Feb','Mar','Apr'};
rainfall = [327.2 368.2 197.6 178.4];
M = containers.Map(months,rainfall)
```

```
M = 
  Map with properties:

        Count: 4
      KeyType: char
    ValueType: double
```

Return values that correspond to specified keys.

```
keySet = {'Jan','Feb'};
valueSet = values(M,keySet)
```

```
valueSet=1×2 cell
    {[327.2000]}    {[368.2000]}
```

Return one value. Even when you specify one key, you must specify it as a cell array.

```
keySet = {'Apr'};
valueSet = values(M,keySet)
```

```
valueSet = 1x1 cell array
    {[178.4000]}
```

Input Arguments

M — Input Map object
Map object

Input Map object.

keySet — Keys corresponding to values to return
cell array

Keys corresponding to values to return from the Map object, specified as a cell array.

Even when you specify keys as strings, the keys must be contained in a cell array.

See Also

containers.Map | isKey | keys | remove

Topics

"Overview of Map Data Structure"
"Examine Contents of Map"
"Read and Write Using Key Index"
"Modify Keys and Values in Map"

Introduced in R2008b

vander

Vandermonde matrix

Syntax

```
A = vander(v)
```

Description

`A = vander(v)` returns the “Vandermonde Matrix” on page 1-16631 such that its columns are powers of the vector `v`.

Examples

Find the Vandermonde Matrix for Vector Input

Use the colon operator to create vector `v`. Find the Vandermonde matrix for `v`.

```
v = 1:.5:3
```

```
v = 1×5

    1.0000    1.5000    2.0000    2.5000    3.0000
```

```
A = vander(v)
```

```
A = 5×5

    1.0000    1.0000    1.0000    1.0000    1.0000
    5.0625    3.3750    2.2500    1.5000    1.0000
   16.0000    8.0000    4.0000    2.0000    1.0000
   39.0625   15.6250    6.2500    2.5000    1.0000
   81.0000   27.0000    9.0000    3.0000    1.0000
```

Find the alternate form of the Vandermonde matrix using `fliplr`.

```
A = fliplr(vander(v))
```

```
A = 5×5

    1.0000    1.0000    1.0000    1.0000    1.0000
    1.0000    1.5000    2.2500    3.3750    5.0625
    1.0000    2.0000    4.0000    8.0000   16.0000
    1.0000    2.5000    6.2500   15.6250   39.0625
    1.0000    3.0000    9.0000   27.0000   81.0000
```

Input Arguments

v — Input
numeric vector

Input, specified as a numeric vector.

Data Types: `single` | `double`
Complex Number Support: Yes

More About

Vandermonde Matrix

For input vector $v = \begin{bmatrix} v_1 & v_2 & \dots & v_N \end{bmatrix}$, the Vandermonde matrix is

$$\begin{bmatrix} v_1^{N-1} & \cdots & v_1^1 & v_1^0 \\ v_2^{N-1} & \cdots & v_2^1 & v_2^0 \\ & \iddots & \vdots & \vdots \\ v_N^{N-1} & & v_N^1 & v_N^0 \end{bmatrix}$$

The matrix is described by the formula $A(i, j) = v(i)^{(N-j)}$ such that its columns are powers of the vector `v`.

An alternate form of the Vandermonde matrix flips the matrix along the vertical axis, as shown. Use `fliplr(vander(v))` to return this form.

$$\begin{bmatrix} v_1^0 & v_1^1 & \cdots & v_1^{N-1} \\ v_2^0 & v_2^1 & \cdots & v_2^{N-1} \\ \vdots & \vdots & \ddots & \\ v_N^0 & v_N^1 & & v_N^{N-1} \end{bmatrix}$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

gallery | hilb | mpower | pascal | power | rosser | toeplitz

Introduced before R2006a

var

Variance

Syntax

```
V = var(A)
V = var(A,w)
V = var(A,w,'all')
V = var(A,w,dim)
V = var(A,w,vecdim)
V = var( ___ ,nanflag)
```

Description

`V = var(A)` returns the variance on page 1-16640 of the elements of `A` along the first array dimension whose size does not equal 1.

- If `A` is a vector of observations, the variance is a scalar.
- If `A` is a matrix whose columns are random variables and whose rows are observations, `V` is a row vector containing the variances corresponding to each column.
- If `A` is a multidimensional array, then `var(A)` treats the values along the first array dimension whose size does not equal 1 as vectors. The size of this dimension becomes `1` while the sizes of all other dimensions remain the same.
- The variance is normalized by the number of observations`-1` by default.
- If `A` is a scalar, `var(A)` returns `0`. If `A` is a `0`-by-`0` empty array, `var(A)` returns `NaN`.

`V = var(A,w)` specifies a weighting scheme. When `w = 0` (default), `V` is normalized by the number of observations`-1`. When `w = 1`, it is normalized by the number of observations. `w` can also be a weight vector containing nonnegative elements. In this case, the length of `w` must equal the length of the dimension over which `var` is operating.

`V = var(A,w,'all')` computes the variation over all elements of `A` when `w` is either 0 or 1. This syntax is valid for MATLAB versions R2018b and later.

`V = var(A,w,dim)` returns the variance along the dimension `dim`. To maintain the default normalization while specifying the dimension of operation, set `w = 0` in the second argument.

`V = var(A,w,vecdim)` computes the variance over the dimensions specified in the vector `vecdim` when `w` is 0 or 1. For example, if `A` is a matrix, then `var(A,0,[1 2])` computes the variance over all elements in `A`, since every element of a matrix is contained in the array slice defined by dimensions 1 and 2.

`V = var( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. For example, `var(A,'includenan')` includes all `NaN` values in `A` while `var(A,'omitnan')` ignores them.

Examples

Variance of Matrix

Create a matrix and compute its variance.

```
A = [4 -7 3; 1 4 -2; 10 7 9];
var(A)
```

```
ans = 1×3

   21.0000   54.3333   30.3333
```

Variance of Array

Create a 3-D array and compute its variance.

```
A(:,:,1) = [1 3; 8 4];
A(:,:,2) = [3 -4; 1 2];
var(A)
```

```
ans =
ans(:,:,1) =
```

```
   24.5000    0.5000


ans(:,:,2) =

     2    18
```

Specify Variance Weight Vector

Create a matrix and compute its variance according to a weight vector `w`.

```
A = [5 -4 6; 2 3 9; -1 1 2];
w = [0.5 0.25 0.25];
var(A,w)
```

```
ans = 1×3

    6.1875    9.5000    6.1875
```

Specify Dimension for Variance

Create a matrix and compute its variance along the first dimension.

```
A = [4 -2 1; 9 5 7];
var(A,0,1)
```

```
ans = 1×3

   12.5000   24.5000   18.0000
```

Compute the variance of `A` along the second dimension.

```
var(A,0,2)
```

```
ans = 2×1

     9
```

```
     4
```

Variance of Array Page

Create a 3-D array and compute the variance over each page of data (rows and columns).

```
A(:,:,1) = [2 4; -2 1];
A(:,:,2) = [9 13; -5 7];
A(:,:,3) = [4 4; 8 -3];
V = var(A,0,[1 2])
```

```
V =
V(:,:,1) =

    6.2500


V(:,:,2) =

    60


V(:,:,3) =

   20.9167
```

Variance Excluding NaN

Create a vector and compute its variance, excluding NaN values.

```
A = [1.77 -0.005 3.98 -2.95 NaN 0.34 NaN 0.19];
V = var(A,'omitnan')
```

```
V = 5.1970
```

Input Arguments

A — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `single` | `double`
Complex Number Support: Yes

w — Weight
`0` (default) | `1` | vector

Weight, specified as one of:

- `0` — normalizes by the number of observations `-1`. If there is only one observation, the weight is 1.
- `1` — normalizes by the number of observations.
- a vector made up of nonnegative scalar weights corresponding to the dimension of `A` along which the variance is calculated.

Data Types: `single` | `double`

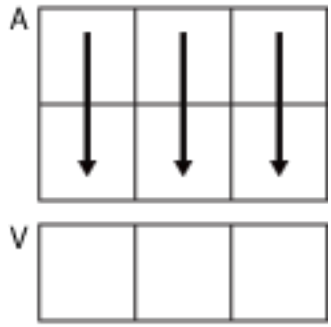
dim — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Dimension `dim` indicates the dimension whose length reduces to `1`. The `size(V,dim)` is `1`, while the sizes of all other dimensions remain the same.

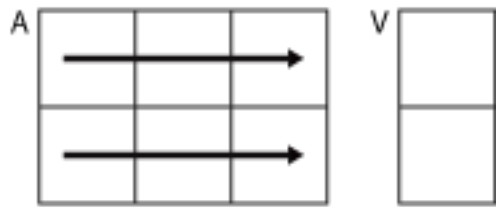
Consider a two-dimensional input array, `A`.

- If `dim = 1`, then `var(A,0,1)` returns a row vector containing the variance of the elements in each column.



var(A,0,1)

- If `dim = 2`, then `var(A,0,2)` returns a column vector containing the variance of the elements in each row.



var(A,0,2)

`var` returns an array of zeros the same size as `A` when `dim` is greater than `ndims(A)`.

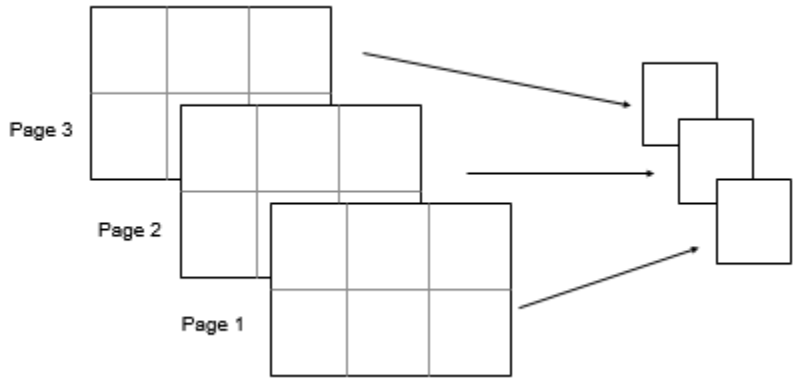
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`vecdim` — Vector of dimensions

vector of positive integers

Vector of dimensions, specified as a vector of positive integers. Each element represents a dimension of the input array. The lengths of the output in the specified operating dimensions are 1, while the others remain the same.

Consider a 2-by-3-by-3 input array, `A`. Then `var(A,0,[1 2])` returns a 1-by-1-by-3 array whose elements are the variances computed over each page of `A`.



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`nanflag` — `NaN` condition
`'includenan'` (default) | `'omitnan'`

`NaN` condition, specified as one of these values:

- `'includenan'` — the variance of input containing `NaN` values is also `NaN`.
- `'omitnan'` — all `NaN` values appearing in either the input array or weight vector are ignored.

Data Types: `char`

More About

Variance

For a random variable vector A made up of N scalar observations, the variance is defined as

$$V = \frac{1}{N-1}\sum_{i=1}^{N}|A_i - \mu|^2$$

where μ is the mean of A,

$$\mu = \frac{1}{N}\sum_{i=1}^{N} A_i .$$

Some definitions of variance use a normalization factor of *N* instead of *N-1*, which can be specified by setting `w` to `1`. In either case, the mean is assumed to have the usual normalization factor *N*.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitation:

The weighting scheme cannot be a vector.

For more information, see “Tall Arrays for Out-of-Memory Data”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If specified, `dim` must be a constant.
- See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- If specified, `dim` must be a constant.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`corrcoef` | `cov` | `mean` | `std`

Introduced before R2006a

varargin

Variable-length input argument list

Syntax

```
varargin
```

Description

`varargin` is an input variable in a function definition statement that enables the function to accept any number of input arguments. Specify `varargin` using lowercase characters, and include it as the last input argument after any explicitly declared inputs.

When the function executes, `varargin` is a 1-by-*N* cell array, where *N* is the number of inputs that the function receives after the explicitly declared inputs. However, if the function receives no inputs after the explicitly declared inputs, then `varargin` is an empty cell array.

Examples

Variable Number of Function Inputs

Define a function in a file named `acceptVariableNumInputs.m` that accepts a variable number of inputs and displays the values of each input.

```
type acceptVariableNumInputs
```

```
function acceptVariableNumInputs(varargin)
    disp("Number of input arguments: " + nargin)
    celldisp(varargin)
end
```

Call the function with several inputs.

```
acceptVariableNumInputs(ones(3),'some text',pi)
```

```
Number of input arguments: 3

ans{1} =

     1     1     1
     1     1     1
     1     1     1



ans{2} =

some text


ans{3} =

    3.1416
```

varargin and Declared Inputs

Define a function in a file named `definedAndVariableNumInputs.m` that expects two inputs and accepts an additional number of inputs.

```
type definedAndVariableNumInputs
```

```
function definedAndVariableNumInputs(X,Y,varargin)
    disp("Total number of input arguments: " + nargin)

    formatSpec = "Size of varargin cell array: %dx%d";
    str = compose(formatSpec,size(varargin));
    disp(str)

end
```

Call the function with several inputs.

```
definedAndVariableNumInputs(7,pi,rand(4),datetime('now'),'hello')
```

```
Total number of input arguments: 5
Size of varargin cell array: 1x3
```

Call the function with two inputs. `varargin` is an empty cell array.

```
definedAndVariableNumInputs(13,42)
```

```
Total number of input arguments: 2
Size of varargin cell array: 0x0
```

Variable Number of Inputs and Outputs

Define a function in a file named `variableNumInputAndOutput.m` that accepts a variable number of inputs and outputs.

```
type variableNumInputAndOutput
```

```
function varargout = variableNumInputAndOutput(varargin)
    disp(['Number of provided inputs: ' num2str(length(varargin))])
    disp(['Number of requested outputs: ' num2str(nargout)])

    for k = 1:nargout
        varargout{k} = k;
    end
end
```

Call the function with two inputs and three outputs.

```
[d,g,p] = variableNumInputAndOutput(6,'Nexus')
```

```
Number of provided inputs: 2
Number of requested outputs: 3

d = 1

g = 2

p = 3
```

Call the function again with no inputs or outputs.

```
variableNumInputAndOutput
```

```
Number of provided inputs: 0
Number of requested outputs: 0
```

Create Function Wrapper

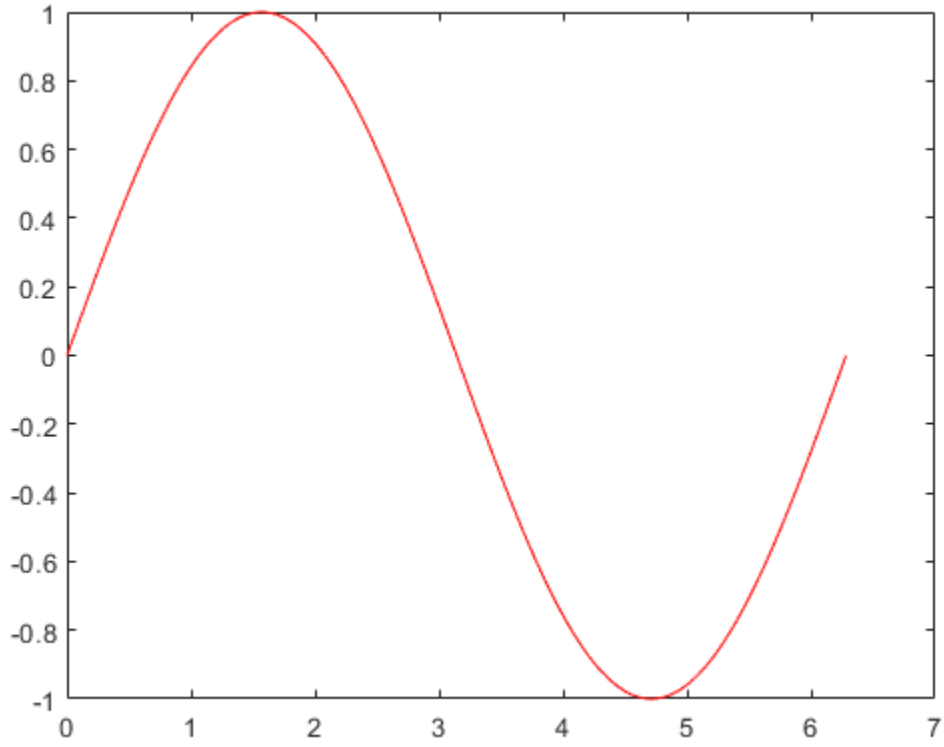
In a file in your working folder, create a wrapper to the plot function that plots a red line. The `redplot` function accepts a variable-length input argument list and returns a variable-length output argument list. It sets the line color to red, and forwards other input values to the `plot` function. This function wrapper enables you to pass `redplot` the same inputs as `plot` and not specify that the line color is red.

```
type redplot.m
```

```
function varargout = redplot(varargin)
    [varargout{1:nargout}] = plot(varargin{:},'Color',[1,0,0]);
end
```

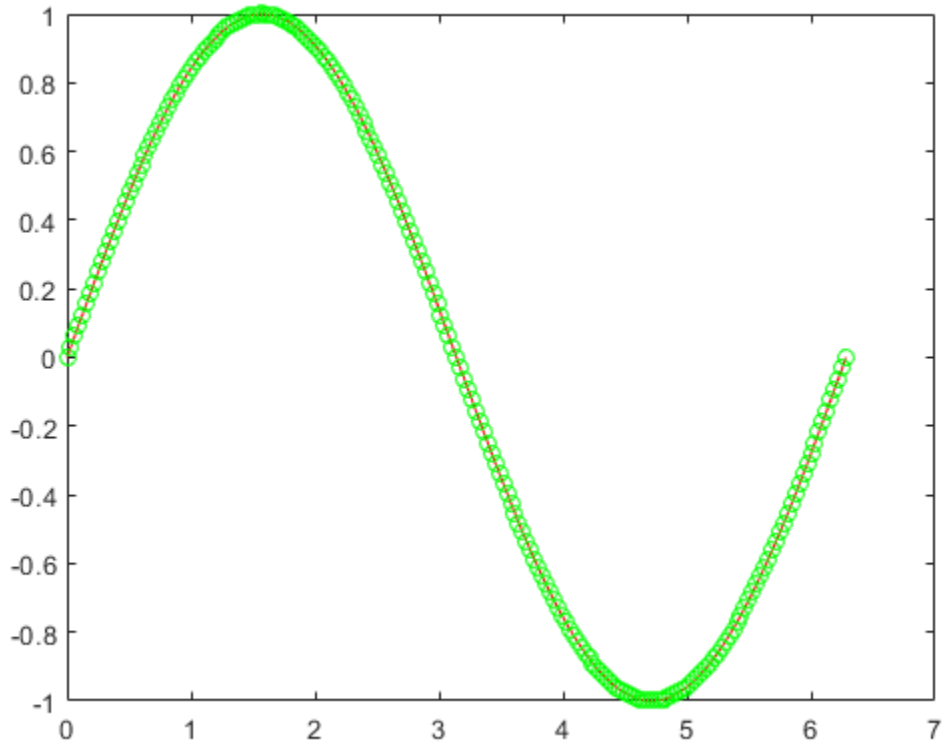
Use `redplot` to create a line plot.

```
x = 0:pi/100:2*pi;
y = sin(x);
redplot(x,y)
```



Call `redplot` again, and specify input and output arguments to forward to the `plot` function.

```
h = redplot(x,y,'Marker','o','MarkerEdgeColor','green');
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If you use `varargin` to define an argument to an entry-point (top-level) function, the code generator produces a C/C++ function with a fixed number of input arguments.

The number of arguments that you specify when you generate code determines the fixed number of arguments.

- You cannot write to `varargin`. If you want to write to input arguments, first copy the input arguments into a local variable.
- The index into `varargin` must be a compile-time constant.

See Also

`nargin` | `narginchk` | `varargout`

Topics

"Support Variable Number of Inputs"
"Argument Checking in Nested Functions"

Introduced before R2006a

varargout

Variable-length output argument list

Syntax

```
varargout
```

Description

`varargout` is an output variable in a function definition statement that enables the function to return any number of output arguments. Specify `varargout` using lowercase characters, and include it as the last output argument after any explicitly declared outputs.

When the function executes, `varargout` is a 1-by-*N* cell array, where *N* is the number of outputs requested after the explicitly declared outputs.

Examples

Variable Number of Function Outputs

Define a function in a file named `returnVariableNumOutputs.m` that returns an output size vector `s` and a variable number of additional outputs.

```
type returnVariableNumOutputs
```

```
function [s,varargout] = returnVariableNumOutputs(x)
    nout = max(nargout,1) - 1;
    s = size(x);
    for k = 1:nout
        varargout{k} = s(k);
    end
end
```

Output `s` contains the dimensions of the input array `x`. Additional outputs correspond to the individual dimensions within `s`.

Call the function with a three-dimensional array and request three outputs.

```
A = rand(4,5,2);
[s,rows,cols] = returnVariableNumOutputs(A)
```

```
s = 1×3

     4     5     2


rows = 4

cols = 5
```

Call the function again with a four-dimensional array and request four outputs. This time, the function does not return the individual fourth dimension.

```
A = zeros(1,4,5,2);
[s,dim1,dim2,dim3] = returnVariableNumOutputs(A)
```

```
s = 1×4

     1     4     5     2


dim1 = 1

dim2 = 4

dim3 = 5
```

Call the function once more on `A` and request one output. Now the function returns the dimensions of `A` and not `varargout`.

```
s = returnVariableNumOutputs(A)
```

```
s = 1×4

     1     4     5     2
```

Variable Number of Inputs and Outputs

Define a function in a file named `variableNumInputAndOutput.m` that accepts a variable number of inputs and outputs.

```
type variableNumInputAndOutput
```

```
function varargout = variableNumInputAndOutput(varargin)
    disp(['Number of provided inputs: ' num2str(length(varargin))])
    disp(['Number of requested outputs: ' num2str(nargout)])

    for k = 1:nargout
        varargout{k} = k;
    end
end
```

Call the function with two inputs and three outputs.

```
[d,g,p] = variableNumInputAndOutput(6,'Nexus')
```

```
Number of provided inputs: 2
Number of requested outputs: 3

d = 1

g = 2

p = 3
```

Call the function again with no inputs or outputs.

```
variableNumInputAndOutput
```

```
Number of provided inputs: 0
Number of requested outputs: 0
```

Create Function Wrapper

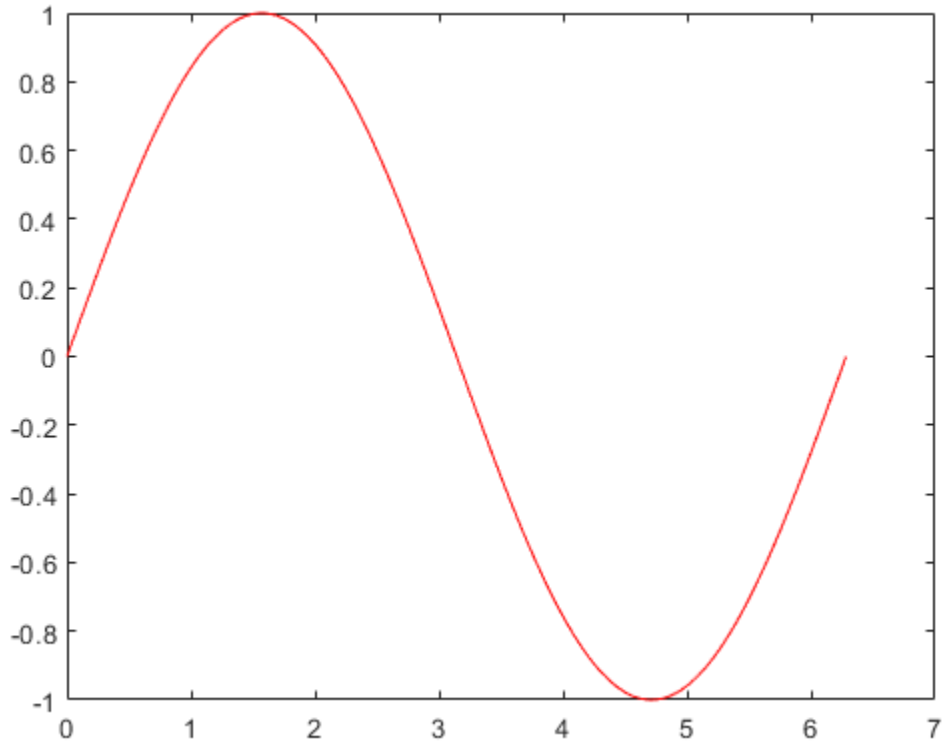
In a file in your working folder, create a wrapper to the plot function that plots a red line. The `redplot` function accepts a variable-length input argument list and returns a variable-length output argument list. It sets the line color to red, and forwards other input values to the `plot` function. This function wrapper enables you to pass `redplot` the same inputs as `plot` and not specify that the line color is red.

```
type redplot.m
```

```
function varargout = redplot(varargin)
    [varargout{1:nargout}] = plot(varargin{:},'Color',[1,0,0]);
end
```

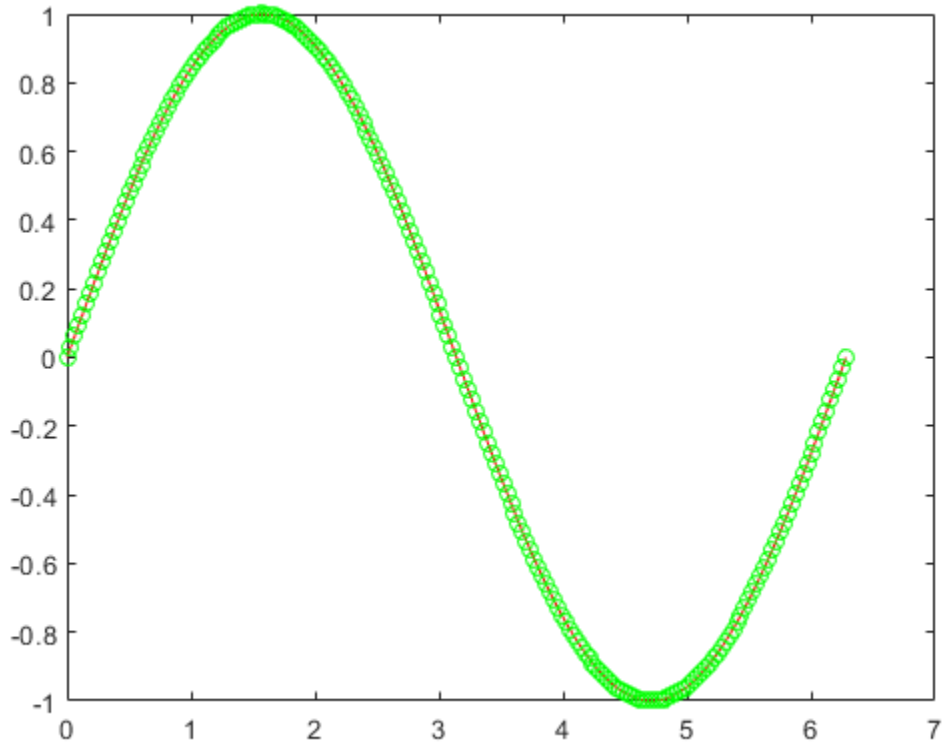
Use `redplot` to create a line plot.

```
x = 0:pi/100:2*pi;
y = sin(x);
redplot(x,y)
```



Call `redplot` again, and specify input and output arguments to forward to the `plot` function.

```
h = redplot(x,y,'Marker','o','MarkerEdgeColor','green');
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For code generation, to use `varargout` in an entry-point (top-level) function, you must specify the number of output parameters by using the `codegen -nargout` syntax.

- The index into `varargout` must be a compile-time constant.

See Also

`nargout` | `nargoutchk` | `varargin`

Topics

"Support Variable Number of Outputs"
"Argument Checking in Nested Functions"

Introduced before R2006a

varfun

Apply function to table or timetable variables

Syntax

```
B = varfun(func,A)
B = varfun(func,A,Name,Value)
```

Description

`B = varfun(func,A)` applies the function `func` separately to each variable of the table or timetable `A` and returns the results in the table or timetable `B`.

The function `func` must take one input argument and return arrays with the same number of rows each time it is called. The `i`th variable in the output argument, `B{:,i}`, is equal to `func(A{:,i})`.

If `A` is a timetable and `func` aggregates data over groups of rows, then `varfun` assigns the first row time from each group of rows in `A` as the corresponding row time in `B`. To return `B` as a table without row times, specify `'OutputFormat'` as `'table'`.

`B = varfun(func,A,Name,Value)` applies the function `func` separately to each variable of the table or timetable `A` with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify which variables to pass to the function.

Examples

Apply Element-wise Function

Define and apply an element-wise function to the variables of a table to square all the elements.

Define a table containing numeric variables.

```
A = table([0.71;-2.05;-0.35;-0.82;1.57],[0.23;0.12;-0.18;0.23;0.41])
```

```
A=5×2 table
    Var1     Var2
    _____    _____

     0.71     0.23
    -2.05     0.12
    -0.35    -0.18
    -0.82     0.23
     1.57     0.41
```

Define the anonymous function to find the square of an input.

```
func = @(x) x.^2;
```

Apply the function to all the variables of table A.

```
B = varfun(func,A)
```

```
B=5×2 table
    Fun_Var1    Fun_Var2
    ________    ________

     0.5041      0.0529
     4.2025      0.0144
     0.1225      0.0324
     0.6724      0.0529
     2.4649      0.1681
```

The variables of B have names based on the function and the variable names from A.

Apply Function that Returns Scalar From Vector

Compute the mean of each variable in a 5-by-2 table.

Define a table containing numeric variables.

```
A = table([0.71;-2.05;-0.35;-0.82;1.57],[0.23;0.12;-0.18;0.23;0.41])
```

```
A=5×2 table
    Var1     Var2
    _____    _____

     0.71     0.23
    -2.05     0.12
    -0.35    -0.18
    -0.82     0.23
     1.57     0.41
```

Define the anonymous function to find the mean of an input.

```
func = @mean;
```

`func` uses an existing MATLAB® function to define the operation.

Apply the function to all the variables of table `A`.

```
B = varfun(func,A)
```

```
B=1×2 table
    mean_Var1    mean_Var2
    _________    _________

     -0.188        0.162
```

`B` is a table containing the average value from each variable. To return a numeric vector instead of a table, you can use `B = varfun(func,A,'OutputFormat','uniform')`.

Apply Function to Groups Within Variables

Compute the group-wise means of variables in a table, `A`, and return them as rows in a table, `B`.

Create a table where one variable defines groups.

```
A = table({'test2';'test1';'test2';'test3';'test1'},...
    [0.71;-2.05;-0.35;-0.82;1.57],[0.23;0.12;-0.18;0.23;0.41])
```

```
A=5×3 table
      Var1       Var2     Var3
```

```
    _________    _____    _____

    {'test2'}     0.71     0.23
    {'test1'}    -2.05     0.12
    {'test2'}    -0.35    -0.18
    {'test3'}    -0.82     0.23
    {'test1'}     1.57     0.41
```

Define the anonymous function to find the mean of an input.

```
func = @mean;
```

`func` uses an existing MATLAB® function to define the operation.

Apply the function to each group of data defined by `Var1`.

```
B = varfun(func,A,'GroupingVariables','Var1')
```

```
B=3×4 table
      Var1       GroupCount    mean_Var2    mean_Var3
    _________    __________    _________    _________

    {'test1'}        2           -0.24        0.265
    {'test2'}        2            0.18        0.025
    {'test3'}        1           -0.82         0.23
```

`B` contains a variable called `GroupCount` to indicate the number of entries from table `A` in that group.

Apply Function to Groups Within Timetable Variables

Create a timetable containing sample data. The row times of the timetable also define groups.

```
dt = datetime(2016,1,1)+days([0 1 1 2 3 3])';
A = timetable(dt,[0.71;-2.05;-0.35;-0.82;1.57;0.09],...
              [0.23;0.12;-0.18;0.23;0.41;0.02],...
              'VariableNames',{'x' 'y'})
```

```
A=6×3 timetable
             dt                x        y
```

```
    ____________________    _____    _____

    01-Jan-2016 00:00:00     0.71     0.23
    02-Jan-2016 00:00:00    -2.05     0.12
    02-Jan-2016 00:00:00    -0.35    -0.18
    03-Jan-2016 00:00:00    -0.82     0.23
    04-Jan-2016 00:00:00     1.57     0.41
    04-Jan-2016 00:00:00     0.09     0.02
```

Compute the group-wise means of the variables in the timetable. `varfun` returns `B` as a timetable because `A` is a timetable. When you specify the row times as the grouping variable, you cannot specify any variable as another grouping variable.

```
B = varfun(@mean,A,'GroupingVariables','dt')
```

```
B=4×4 timetable
             dt             GroupCount    mean_x    mean_y
    ____________________    __________    ______    ______

    01-Jan-2016 00:00:00        1          0.71      0.23
    02-Jan-2016 00:00:00        2          -1.2     -0.03
    03-Jan-2016 00:00:00        1         -0.82      0.23
    04-Jan-2016 00:00:00        2          0.83     0.215
```

Input Arguments

`func` — Function
function handle

Function, specified as a function handle. You can define the function in a file or as an anonymous function. If `func` corresponds to more than one function file (that is, if `func` represents a set of overloaded functions), MATLAB determines which function to call based on the class of the input arguments.

Use the `'OutputFormat','cell'` name-value pair argument, if the function `func` take one input argument and returns arrays with a different numbers of rows each time it is called. Otherwise, `func` must return arrays with the same number of rows.

Example: `func = @(x) x.^2;` computes the square of each element of an input.

A — Input table
table | timetable

Input table, specified as a table or a timetable.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'InputVariables',2` uses only the second variable in `A` as an input to `func`.

`InputVariables` — Specifiers for selecting variables of `A` to pass to `func`
positive integer | vector of positive integers | character vector | cell array of character vectors | string array | logical vector | function handle

Specifiers for selecting variables of `A` to pass to `func`, specified as the comma-separated pair consisting of `'InputVariables'` and a positive integer, vector of positive integers, character vector, cell array of character vectors, string array, logical vector, or a function handle.

If you specify `'InputVariables'` as a function handle, then it must return a logical scalar, and `varfun` passes only the variables in `A` where the function returns `1` (`true`).

`GroupingVariables` — One or more variables in `A` that define groups of rows
positive integer | vector of positive integers | character vector | cell array of character vectors | string array | logical vector

One or more variables in `A` that define groups of rows, specified as the comma-separated pair consisting of `'GroupingVariables'` and a positive integer, vector of positive integers, character vector, cell array of character vectors, string array, or logical vector.

The value of `'GroupingVariables'` specifies which table variables are the grouping variables, not their data types. A grouping variable can be numeric, or have data type `categorical`, `calendarDuration`, `datetime`, `duration`, `logical`, or `string`.

Rows in `A` that have the same grouping variable values belong to the same group. `varfun` applies `func` to each group of rows within each of the variables of `A`, rather than to each entire variable.

If any grouping variable contains NaNs or missing values (such as NaTs, undefined categorical values, or missing strings), then the corresponding rows do not belong to any group, and are excluded from the output.

Row labels can be grouping variables. You can group on row labels alone, on one or more variables in A, or on row labels and variables together.

- If A is a table, then the labels are row names.
- If A is a timetable, then the labels are row times.

The output, B, has one row for each group when you specify `'OutputFormat','uniform'` or `'OutputFormat','cell'`. When you specify `'OutputFormat','table'` or `'OutputFormat','timetable'`, the sizes of the outputs from `func` determine how many rows of B correspond to each group.

`OutputFormat` — Format of B

`'table'` (default) | `'timetable'` | `'uniform'` | `'cell'`

Format of B, specified as the comma-separated pair consisting of `'OutputFormat'` and either the value `'uniform'`, `'table'`, `'timetable'`, or `'cell'`.

`'table'`	`varfun` returns a table with one variable for each variable in A (or each variable specified with `'InputVariables'`). For grouped computation, B, also contains the grouping variables. `'table'` allows you to use a function that returns values of different sizes or data types for the different variables in A. However, for ungrouped computation, `func` must return arrays with the same number of rows each time it is called. For grouped computation, `func` must return values with the same number of rows each time it is called for a given group. If A is a table, then this is the default output format.

`'timetable'`	`varfun` returns a timetable with one variable for each variable in `A` (or each variable specified with `'InputVariables'`). For grouped computation, `B` also contains the grouping variables. `varfun` creates the row times of `B` from the row times of `A`. If the row times assigned to `B` do not make sense in the context of the calculations performed using `func`, then specify the output format as `'OutputFormat','table'`. If `A` is a timetable, then this is the default output format.
`'uniform'`	`varfun` concatenates the values into a vector. `func` must return a scalar with the same data type each time it is called.
`'cell'`	`varfun` returns `B` as a cell array. `'cell'` allows you to use a function that returns values of different sizes or data types.

`ErrorHandler` — Function to call if `func` fails

function handle

Function to call if `func` fails, specified as the comma-separated pair consisting of `'ErrorHandler'` and a function handle. Define this function so that it rethrows the error or returns valid outputs for function `func`.

MATLAB calls the specified error-handling function with two input arguments:

- A structure with these fields:

`identifier`	Error identifier.
`message`	Error message text.
`index`	Index of the variable for which the error occurred.
`name`	Name of the variable for which the error occurred.

- The set of input arguments to function `func` at the time of the error.

For example,

```
function [A, B] = errorFunc(S, varargin)
warning(S.identifier, S.message)
A = NaN; B = NaN;
```

Output Arguments

B — Output table
table | timetable

Output table, returned as a table or a timetable. B can store metadata such as descriptions, variable units, variable names, and row names. For more information, see the Properties sections of `table` or `timetable`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays with the limitations:

- The `func` input must always return a tall array.
- Supported name-value pairs are:
  - `'InputVariables'` — Cannot be specified as a function handle.
  - `'OutputFormat'` — Value can be `'uniform'`, `'table'`, `'timetable'`, or `'cell'` only.
- When the input array is a tall timetable and `'OutputFormat'` is `'timetable'`, the specified function must return an array with the same size in the first dimension as the input. Specify `'OutputFormat'` as `'table'` when the input function is a reduction function such as `mean`.

For more information, see “Tall Arrays”.

See Also

`arrayfun` | `cellfun` | `findgroups` | `groupsummary` | `rowfun` | `splitapply` | `structfun`

Topics

“Calculations on Tables”

“Split Table Data Variables and Apply Functions”
“Anonymous Functions”

Introduced in R2013b

vartype

Subscript into table or timetable by variable type

Syntax

```
S = vartype(type)
```

Description

`S = vartype(type)` creates a subscript to select table variables of a specified type. The `type` input argument is a character vector that specifies any type that is accepted by the `isa` function, such as `'numeric'`, `'float'`, `'integer'`, or `'string'`. It also can be `'cellstr'` to select variables that contain cell arrays of character vectors.

For example, `S = vartype('numeric'); T2 = T1(:,S)` returns `T2` as a table that contains only the numeric variables from the table `T1`.

Examples

Select Variables in Table

Create a table that contains numeric and string variables. Then subscript into the table to get only its numeric variables.

```
LastName = string({'Smith';'Johnson';'Williams';'Jones';'Brown'});
Age = [38;43;38;40;49];
Height = [71;69;64;67;64];
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];
T = table(LastName,Age,Height,Weight,BloodPressure)
```

```
T=5×5 table
     LastName     Age    Height    Weight    BloodPressure
    __________    ___    ______    ______    _____________
```

```
    "Smith"       38       71       176       124     93
    "Johnson"     43       69       163       109     77
    "Williams"    38       64       131       125     83
    "Jones"       40       67       133       117     75
    "Brown"       49       64       119       122     80
```

Create a subscript with the `vartype` function. Subscript into the second dimension of `T` to return a table that contains only the numeric variables.

```
S = vartype('numeric');
T2 = T(:,S)
```

```
T2=5×4 table
    Age    Height    Weight    BloodPressure
    ___    ______    ______    _____________

    38       71       176       124     93
    43       69       163       109     77
    38       64       131       125     83
    40       67       133       117     75
    49       64       119       122     80
```

You can create a subscript for any type that the `isa` function accepts. Select the string variable from `T`.

```
S = vartype('string');
T3 = T(:,S)
```

```
T3=5×1 table
     LastName
    __________

    "Smith"
    "Johnson"
    "Williams"
    "Jones"
    "Brown"
```

Select Variables in Timetable

Create a timetable that contains numeric, string, and categorical variables. Then subscript into the table to get only its numeric variables.

```
Date = datetime({'12/18/2015';'12/19/2015';'12/20/2015'});
Temp = [45;33;36];
Pressure = [30.1;29.3;29.7];
Location = string({'Boston';'Boston';'Worcester'});
SensorType = categorical({'S1';'X7';'S1'});
TT = timetable(Date,Temp,Pressure,Location,SensorType)
```

```
TT=3×5 timetable
       Date        Temp    Pressure     Location      SensorType
    ___________    ____    ________    ___________    __________

    18-Dec-2015     45       30.1      "Boston"           S1
    19-Dec-2015     33       29.3      "Boston"           X7
    20-Dec-2015     36       29.7      "Worcester"        S1
```

Create a subscript with the `vartype` function. Subscript into the second dimension of `TT` to return a timetable that contains only the numeric variables. `TT2` also has the row times from `TT` because the times identify the rows. The vector of row times is a property of the timetable, and not one of its variables.

```
S = vartype('numeric');
TT2 = TT(:,S)
```

```
TT2=3×3 timetable
       Date        Temp    Pressure
    ___________    ____    ________

    18-Dec-2015     45       30.1
    19-Dec-2015     33       29.3
    20-Dec-2015     36       29.7
```

Input Arguments

type — Type of variables to select from a table or a timetable
character vector | string scalar

Type of variables to select from a table or a timetable, specified as a character vector or string scalar. `type` can be any type or category that is accepted by the `isa` function. It also can be `'cellstr'`.

See Also

`isa` | `table` | `timerange` | `timetable` | `withtol`

Topics

"Select Timetable Data by Row Time and Variable Type"
"Tables"
"Represent Dates and Times in MATLAB"

Introduced in R2016b

vecnorm

Vector-wise norm

Syntax

```
N = vecnorm(A)
N = vecnorm(A,p)
N = vecnorm(A,p,dim)
```

Description

`N = vecnorm(A)` returns the 2-norm or Euclidean norm on page 1-16673 of `A`:

- If `A` is a vector, then `vecnorm` returns the norm of the vector.
- If `A` is a matrix, then `vecnorm` returns the norm of each column.
- If `A` is a multidimensional array, then `vecnorm` returns the norm along the first array dimension whose size does not equal 1.

`N = vecnorm(A,p)` calculates the generalized vector p-norm on page 1-16673.

`N = vecnorm(A,p,dim)` operates along dimension `dim`. The size of this dimension reduces to 1 while the sizes of all other dimensions remain the same.

Examples

1-Norm and 2-Norm of Vector

Calculate the 2-norm of a vector corresponding to the point (2,2,2) in 3-D space. The 2-norm is equal to the Euclidean length of the vector, $\sqrt{12}$.

```
x = [2 2 2];
n = vecnorm(x)
```

```
n = 3.4641
```

Calculate the 1-norm of the vector, which is the sum of the element magnitudes.

```
n = vecnorm(x,1)
```

```
n = 6
```

2-Norm of Matrix Columns

Calculate the 2-norm of the columns of a matrix.

```
A = [2 0 1;-1 1 0;-3 3 0]
```

```
A = 3×3

     2     0     1
    -1     1     0
    -3     3     0
```

```
n = vecnorm(A)
```

```
n = 1×3

    3.7417    3.1623    1.0000
```

As an alternative, you can use the `norm` function to calculate the 2-norm of the entire matrix.

Input Arguments

A — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array. By convention, `vecnorm` returns `NaN` values if the vector being operated on contains a `NaN` value.

Data Types: `single` | `double`
Complex Number Support: Yes

p — Norm type
2 (default) | positive scalar | `Inf`

Norm type, specified as 2 (default), a positive scalar, or `Inf`.

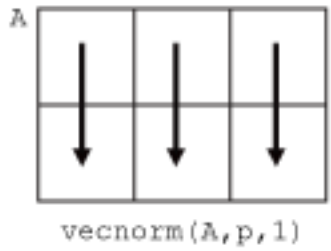
dim — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If you do not specify a value, then the default is the first array dimension whose size does not equal 1.

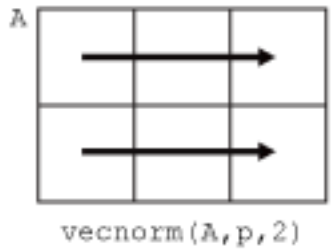
Dimension `dim` indicates the dimension whose length reduces to 1. In other words, `size(N,dim)` is `1`, while the sizes of all other dimensions remain the same.

Consider a two-dimensional input array, `A`:

- `vecnorm(A,p,1)` calculates the norm of each column.



- `vecnorm(A,p,2)` calculates the norm of each row.



`vecnorm` returns `abs(A)` when `dim` is greater than `ndims(A)` or when `size(A,dim)` is `1`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

More About

Euclidean Norm

The Euclidean norm (also called the vector magnitude, Euclidean length, or 2-norm) of a vector `v` with `N` elements is defined by

$$\|v\| = \sqrt{\sum_{k=1}^{N} |v_k|^2} .$$

General Vector Norm

The general definition for the p-norm of a vector `v` that has `N` elements is

$$\|v\|_p = \left[\sum_{k=1}^{N} |v_k|^p\right]^{1/p},$$

where `p` is any positive real value or `Inf`. Some interesting values of `p` are:

- If `p = 1`, then the resulting 1-norm is the sum of the absolute values of the vector elements.
- If `p = 2`, then the resulting 2-norm gives the vector magnitude or Euclidean length of the vector.
- If `p = Inf`, then $\|v\|_\infty = \max_i(|v(i)|)$.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If you supply `dim`, then it must be constant.
- For limitations related to variable-size inputs, see “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).
- Code generation does not support sparse matrix inputs for this function.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`norm`

Introduced in R2017b

vectorize

Vectorize expression

Syntax

```
vectorize(c)
vectorize(f)
```

Description

`vectorize(c)`, where `c` is a character row vector or string scalar, inserts a `.` before any `^`, `*` or `/` in `c`. The result is a character row vector.

Note `vectorize` will not accept inline function objects (`f`) in a future release.

`vectorize(f)` where `f` is an inline function object, vectorizes the formula for `f`. The result is the vectorized version of the function.

See Also

`cd` | `dbtype` | `delete` | `dir` | `inline` | `path` | `what` | `who`

Introduced before R2006a

ver

Version information for MathWorks products

Syntax

```
ver
ver product
```

```
product_info = ver(product)
```

Description

`ver` displays:

- A header containing the current MATLAB product family version number, license number, operating system, and version of Java software for the MATLAB product.
- The version numbers for MATLAB and all other installed MathWorks products.

`ver product` displays, in addition to the header information:

- The current version number for `product`, where `product` is the name of the folder that contains the `Contents.m` file for the product you are inquiring about.

`product_info = ver(product)` returns product information to the structure array, `product_info`.

Examples

Version for All Installed Products

Display version information for all installed products. The output shown here is representative. Your results might differ.

```
ver
```

```
-----------------------------------------------------------------------------------------------------
MATLAB Version: 8.2.0.29 (R2013b)
MATLAB License Number: 234567
Operating System: Microsoft Windows 7 Version 6.1 (Build 7601: Service Pack 1)
Java Version: Java 1.7.0_11-b21 with Oracle Corporation Java HotSpot(TM) 64-Bit Server VM mixed mode
-----------------------------------------------------------------------------------------------------
MATLAB                                                Version 8.2       (R2013b)
Simulink                                              Version 8.2       (R2013b)
Control System Toolbox                                Version 9.6       (R2013b)
```

Version for a Particular Product

Display version information for MATLAB and the Control System Toolbox product. The output shown here is representative. Your results might differ.

Determine the product name for Control System Toolbox by setting `n` to the name of a function unique to Control System Toolbox, such as `dss`:

```
n = 'dss';
pat = '(?<=^.+[\\/]toolbox[\\/])[^\\/]+';
regexp(which(n), pat, 'match', 'once')
```

```
ans=
control
```

Specify the value returned in the previous step as an argument to `ver`:

```
ver control
```

```
-----------------------------------------------------------------------------------------------------
MATLAB Version: 8.2.0.29 (R2013b)
MATLAB License Number: 234567
Operating System: Microsoft Windows 7 Version 6.1 (Build 7601: Service Pack 1)
Java Version: Java 1.7.0_11-b21 with Oracle Corporation Java HotSpot(TM) 64-Bit Server VM mixed mode
-----------------------------------------------------------------------------------------------------
Control System Toolbox                                Version 9.6       (R2013b)
```

Structure Containing Version for MATLAB Products

Create a structure containing version information, and then display the structure values. The output shown here is representative. Your results might differ.

```
v = ver;
for k = 1:length(v)
  fprintf('%s\n', v(k).Name)
```

```
  fprintf('  Version: %s\n', v(k).Version)
end
```

```
MATLAB
 Version: 8.2

MATLAB Compiler
 Version: 5.0

My Custom Toolbox
 Version: 1.0
```

Structure Containing Version for a Particular Product

Create a structure containing version information for just the Symbolic Math Toolbox™ product. The output shown here is representative. Your results might differ.

Determine the product name for Symbolic Math Toolbox by setting `n` to the name of a function unique to Symbolic Math Toolbox, such as `sym`.

```
n = 'sym';
pat = '(?<=^.+[\\/]toolbox[\\/])[^\\/]+';
regexp(which(n), pat, 'match', 'once')
```

```
ans =

symbolic
```

Specify the value returned in the previous step as an argument to `ver`.

```
v = ver('symbolic')
```

```
v =

        Name: 'Symbolic Math Toolbox'
     Version: '5.11'
     Release: '(R2013b)'
        Date: '19-May-2013'
```

Input Arguments

`product` — product-specific information
character vector

The product or toolbox for which you want to view version information, specified as a character vector.

Output Arguments

`product_info` — product name, version, release, and date
structure array

Product name, version, release, and date, returned as a structure array with these fields: `Name`, `Version`, `Release`, and `Date`. If a license is a trial version, the letter `T` precedes the value in the `Version` field.

Tips

- To view version information about installed Add-Ons, including MathWorks products in the Add-On Manager, see “Get and Manage Add-Ons”.

See Also

`computer` | `help` | `license` | `matlabshared.supportpkg.getInstalled` | `verlessthan` | `version`

Topics

“Information About Your Installation”
“Create Help Summary Files — Contents.m”

Introduced before R2006a

matlab.unittest.Verbosity class

Package: `matlab.unittest`

Verbosity level enumeration class

Description

The `matlab.unittest.Verbosity` enumeration class provides a means to specify the level of detail related to running tests. A higher value results in a higher level of detail. The enumeration class contains the following members.

Numeric Representation	Enumeration Member Name	Verbosity Description
`0`	`None`	No information
`1`	`Terse`	Minimal information
`2`	`Concise`	Moderate amount of information
`3`	`Detailed`	Some supplemental information
`4`	`Verbose`	Lots of supplemental information

Construction

`matlab.unittest.Verbosity.`*`MemberName`* creates an instance of the verbosity level enumeration class.

Examples

Create Instance of Enumeration Class

```
n = matlab.unittest.Verbosity.Detailed
```

```
n =

    Detailed
```

Display information about the variables.

```
whos n
```

```
  Name      Size            Bytes  Class                         Attributes

  n         1x1               112  matlab.unittest.Verbosity
```

See Also

log | matlab.unittest.TestRunner.withTextOutput | matlab.unittest.plugins.LoggingPlugin | matlab.unittest.plugins.TestRunProgressPlugin

Topics

"Define Enumeration Classes"

Introduced in R2014b

verctrl

(Has been removed) Source control actions (Windows platforms)

Note `verctrl` has been removed. Access source control actions through the context menu instead.

Syntax

```
verctrl('action',{'filename1','filename2',....},0)
result=verctrl('action',{'filename1','filename2',....},0)
verctrl('action','filename',0)
result=verctrl('isdiff','filename',0)
list = verctrl('all_systems')
```

Description

`verctrl('action',{'filename1','filename2',....},0)` performs the source control operation specified by `'action'` for a single file or multiple files. Enter one file as a character vector; specify multiple files using a cell array of character vectors. Use the full paths for each file name and include the extensions. Specify `0` as the last argument. Complete the resulting dialog box to execute the operation. Available values for `'action'` are as follows:

action Argument	Purpose
`'add'`	Adds files to the source control system. Files can be open in the Editor or closed when added.
`'checkin'`	Checks files into the source control system, storing the changes and creating a new version.
`'checkout'`	Retrieves files for editing.

action Argument	Purpose
`'get'`	Retrieves files for viewing and compiling, but not editing. When you open the files, they are labeled as read-only.
`'history'`	Displays the history of files.
`'remove'`	Removes files from the source control system. It does not delete the files from disk, but only from the source control system.
`'runscc'`	Starts the source control system. The file name can be an empty character vector.
`'uncheckout'`	Cancels a previous checkout operation and restores the contents of the selected files to the precheckout version. All changes made to the files since the checkout are lost.

`result=verctrl('action',{'filename1','filename2',....},0)` performs the source control operation specified by `'action'` on a single file or multiple files. The action can be any one of: `'add'`, `'checkin'`, `'checkout'`, `'get'`, `'history'`, or `'undocheckout'`. `result` is a logical `1` (true) when you complete the operation by clicking **OK** in the resulting dialog box, and is a logical `0` (false) when you abort the operation by clicking **Cancel** in the resulting dialog box.

`verctrl('action','filename',0)` performs the source control operation specified by `'action'` for a single file. Use the absolute path for `'filename'`. Specify `0` as the last argument. Complete any resulting dialog boxes to execute the operation. Available values for `'action'` are as follows:

action Argument	Purpose
`'showdiff'`	Displays the differences between a file and the latest checked in version of the file in the source control system.
`'properties'`	Displays the properties of a file.

`result=verctrl('isdiff','filename',0)` compares `filename` with the latest checked in version of the file in the source control system. `result` is a logical `1` (true) when the files are different, and is a logical `0` (false) when the files are identical. Use the full path for `'filename'`. Specify `0` as the last argument.

`list = verctrl('all_systems')` displays in the Command Window a list of all source control systems installed on your computer.

Examples

Check In a File

Check in `D:\file1.ext` to the source control system:

```
result = verctrl('checkin','D:\file1.ext', 0)
```

This opens the Check in file(s) dialog box. Click **OK** to complete the check in. MATLAB displays

```
 result = 1
```

indicating the checkin was successful.

Add Files to the Source Control System

Add `D:\file1.ext` and `D:\file2.ext` to the source control system.

```
verctrl('add',{'D:\file1.ext','D:\file2.ext'}, 0)
```

This opens the Add to source control dialog box. Click **OK** to complete the operation.

Display the Properties of a File

Display the properties of `D:\file1.ext`.

```
verctrl('properties','D:\file1.ext', 0)
```

This opens the source control properties dialog box for your source control system. The function is complete when you close the properties dialog box.

Show Differences for a File

To show the differences between the version of `file1.ext` that you just edited and saved, with the last version in source control, run

```
verctrl('showdiff','D:\file1.ext',0)
```

MATLAB displays differences dialog boxes and results specific to your source control system. After checking in the file, if you run this statement again, MATLAB displays

```
??? The file is identical to latest version under source control.
```

List All Installed Source Control Systems

To view all of the source control systems installed on your computer, type

```
list = verctrl ('all_systems')
```

MATLAB displays all the source control systems currently installed on your computer. For example:

```
list =
'Microsoft Visual SourceSafe'
'ComponentSoftware RCS'
```

See Also

Topics

"MSSCCI Source Control Interface"

Introduced before R2006a

verLessThan

Compare toolbox version to specified character vector

Syntax

```
tf = verLessThan(toolbox,version)
```

Description

`tf = verLessThan(toolbox,version)` returns logical 1 (`true`) if the version of the toolbox is older than the value specified by `version`. Otherwise, it returns logical 0 (`false`). When there are differences in the behavior of the code in the different versions, use this function to write code that runs on multiple versions of MATLAB.

Examples

Write Version-Specific Code

Modify code that runs in MATLAB R2014a, but that generates an error in R2014b or later.

Create two surface plots. The default color palettes are different depending on which version of MATLAB you are using.

```
s1 = surface(magic(5));
s2 = surface(magic(5)*10,'FaceColor','yellow');
```

Modify surface `s2` by the color of the surface underneath. Starting in R2014b, the `EraseMode` property has been removed from all graphics objects. Replace the `EraseMode` property with a value of the `FaceAlpha` property for code running in MATLAB R2014b and later.

```
if verLessThan('matlab','8.4')
    % -- Code to run in MATLAB R2014a and earlier here --
    s2.EraseMode = 'xor';
```

```
else
    % -- Code to run in MATLAB R2014b and later here --
    s2.FaceAlpha = .25;
end
```

Check for Minimum Required Version

Compare the Simulink version that is running against Version 4.0. If the version is earlier than 4.0, display an error message because the feature is not supported.

```
if verLessThan('simulink','4.0')
    error('Simulink 4.0 or higher is required.')
end
```

Check Toolbox Version

Compare the Data Acquisition Toolbox version that MATLAB is running.

Find the name of the toolbox folder. Your output depends on the toolboxes installed on your system.

```
dir([matlabroot '/toolbox/d*'])
```

```
daq           datafeed      dig           dnnfpga       driving
database      diagram       dmr           dotnetbuilder dsp
```

Use the toolbox folder name, `daq`.

```
verLessThan('daq','3')
```

```
ans =

     0
```

MATLAB is running Data Acquisition Toolbox Version 3 or later.

Input Arguments

`toolbox` — Name of MATLAB toolbox folder
character vector

Name of MATLAB toolbox folder, specified as a character vector. To specify `toolbox`, find the folder containing the `Contents.m` file for the toolbox and use that folder name. To see a list of all toolbox folder names, type:

```
dir([matlabroot '/toolbox'])
```

If `toolbox` does not exist, MATLAB displays an error.

Example: `'images'`

`version` — Version number
character vector

Version number of the program or toolbox to compare against, specified as a character vector. Specify the version number in the form of `major[.minor[.revision]]`.

Example: `'9.2'`

See Also

`dir` | `ismac` | `ispc` | `isunix` | `license` | `ver` | `version`

Introduced in R2007a

version

Version number for MATLAB and libraries

Syntax

```
version
version = version('-versionOption')
[version date] = version
```

Description

`version` displays the version and release number for the currently runningMATLAB.

`version = version('-versionOption')` returns information for the specified option.

`[version date] = version` returns the version, release number, and release date. No input arguments are allowed in this syntax.

Examples

Display MATLAB Version

```
version
```

```
ans =
'9.7.0.1190202 (R2019b)'
```

Display MATLAB Release

Display the release, prefaced by descriptive text.

```
['Release R' version('-release')]
```

```
ans =
'Release R2019b'
```

Display Release Version and Date as Separate Values

```
[v d] = version
```

```
v =
'9.7.0.1190202 (R2019b)'

d =
'August 21, 2019'
```

Display Java Version

```
version -java
```

```
ans =
'Java 1.8.0_202-b08 with Oracle Corporation Java HotSpot(TM) 64-Bit Server VM mixed mo
```

Input Arguments

`versionOption` — Version option
`'-date'` | `'-description'` | `'-release'` | `'-java'`

Version option, specified as one of these values.

`'-date'`	Release date.

`'-description'`	Description of the version. Returns: • empty - general release • 'Update *n*' - update release *n*, where *n* is an integer • 'Prerelease' - prerelease • 'Prerelease *n*' - prerelease update *n*, where *n* is an integer
`'-release'`	Release number.
`'-java'`	Version of the Oracle JVM software that MATLAB uses.

Output Arguments

`version` — Version information
character vector

Version and release information, returned as a character vector.

`date` — Release date
character vector

Release date, returned as a character vector.

See Also

`computer` | `ver` | `verLessThan`

Topics

"Update an Existing Installation"

Introduced before R2006a

VersionResults

Version results object

Description

A `VersionResults` object contains the results of each operation when upgrading the personal settings of a toolbox.

Creation

Create a `ReleaseCompatibilityResults` object for a specific toolbox version number by using the `matlab.settings.loadSettingsCompatibilityResults` function. Then, access the `Results` property to get the `VersionResults` object.

For example, this code gets the `VersionResults` object for version 2 of the toolbox `mytoolbox`.

```
myCompatibilityResults = matlab.settings.loadSettingsCompatibilityResults('mytoolbox','Version2');
myCompatibilityResults.Results
```

```
ans = 
  VersionResults with properties:
      VersionLabel: "Version2"
    VersionChanges: [1×2 matlab.settings.OperationResult]
```

Properties

`VersionLabel` — Toolbox version of results
string scalar

Toolbox version of results, specified as a string scalar.

Example: `"Version2"`

`VersionChanges` — Status for each upgrade operation
array of operation result objects

Status for each upgrade operation, specified as an array of operation result objects.

Examples

Get Results of Upgrading Personal Settings

Create functions to create and then upgrade a toolbox factory tree and then test that the upgrade completes successfully.

Create the function `createMyToolboxFactoryTree` that creates the factory settings tree for the toolbox `mytoolbox`.

```
function myToolboxFactoryTree = createMyToolboxFactoryTree()
    myToolboxFactoryTree = matlab.settings.FactoryGroup.createToolboxGroup('mytoolbox', ...
        'Hidden',false);

    toolboxFontGroup = addGroup(myToolboxFactoryTree,'font','Hidden',false)
    addSetting(toolboxFontGroup,'MyFontSize','FactoryValue',11,'Hidden',false, ...
        'ValidationFcn',@matlab.settings.mustBeNumericScalar)
    addSetting(toolboxFontGroup,'MyFontColor','FactoryValue','Black', ...
        'Hidden',false,'ValidationFcn',@matlab.settings.mustBeStringScalar);
end
```

Create the function `createMyToolboxSettingsFileUpgraders` with an empty settings file upgrader object.

```
function upgraders = createMyToolboxSettingsFileUpgraders()
    upgraders = matlab.settings.SettingsFileUpgrader.empty;
end
```

Create the `settingsInfo.json` file for the toolbox. Specify `mytoolbox` as the root settings group name, `createMyToolboxFactoryTree` as the settings tree creation function, and `createMyToolboxSettingsFileUpgraders` as the settings tree upgrade function. Place `settingsInfo.json` in the toolbox `resources` folder.

```
{
"ToolboxGroupName" : "mytoolbox",
"Hidden" : "false",
"CreateTreeFcn" : "createMyToolboxFactoryTree",
"CreateUpgradersFcn" : "createMyToolboxSettingsFileUpgraders"
}
```

Add the folder that contains the settings tree creation function and the toolbox resources folder to the MATLAB path. Then, load the factory settings tree for `mytoolbox`.

```
matlab.settings.reloadFactoryFile('mytoolbox');
```

Use the `settings` function to access the root of the settings tree and set the personal value for the `MyFontSize` setting.

```
s = settings;
s.mytoolbox.font.MyFontSize.PersonalValue = 15;
```

Change the settings names in `createMyToolboxFactoryTree` to be `FontSize` and `FontColor`.

```
function myToolboxFactoryTree = createMyToolboxFactoryTree()
    myToolboxFactoryTree = matlab.settings.FactoryGroup.createToolboxGroup('mytoolbox', ...
        'Hidden',false);

    toolboxFontGroup = addGroup(myToolboxFactoryTree,'font','Hidden',false)
    addSetting(toolboxFontGroup,'FontSize','FactoryValue',11,'Hidden',false, ...
        'ValidationFcn',@matlab.settings.mustBeNumericScalar)
    addSetting(toolboxFontGroup,'FontColor','FactoryValue','Black', ...
        'Hidden',false,'ValidationFcn',@matlab.settings.mustBeStringScalar);
end
```

Record the rename of the two settings in the `createMyToolboxSettingsFileUpgraders` function as changes to the settings tree for version 2 of `mytoolbox`.

```
function upgraders = createMyToolboxSettingsFileUpgraders()
    upgraders = matlab.settings.SettingsFileUpgrader('Version2');
    upgraders.move('mytoolbox.font.MyFontSize','mytoolbox.font.FontSize');
    upgraders.move('mytoolbox.font.MyFontColor','mytoolbox.font.FontColor');
end
```

Reload the factory settings tree for `mytoolbox`.

```
matlab.settings.reloadFactoryFile('mytoolbox');
```

Use the `settings` function to access the root of the settings tree and verify that the personal value for the `FontSize` setting was correctly moved from the `MyFontSize` setting.

```
s = settings;
s.mytoolbox.font.FontSize
```

```
ans = 
  Setting 'mytoolbox.font.FontSize' with properties:
       ActiveValue: 15
     TemporaryValue: <no value>
      PersonalValue: 15
       FactoryValue: 11
```

Get the results of each upgrade operation for version 2 of `mytoolbox`.

```
compatibilityResults = matlab.settings.loadSettingsCompatibilityResults('mytoolbox','Version2');
compatibilityResults.Results
```

```
ans =
  VersionResults with properties:
      VersionLabel: "Version2"
    VersionChanges: [1×2 matlab.settings.OperationResult]
```

See Also

`OperationResult` | `ReleaseCompatibilityResults` | `matlab.settings.loadSettingsCompatibilityResults`

Topics

"Create Factory Settings for Toolboxes"
"Access and Modify Settings"

Introduced in R2019b

vertcat

Concatenate arrays vertically

Syntax

```
C = vertcat(A,B)
C = vertcat(A1,A2,…,An)
```

Description

`C = vertcat(A,B)` concatenates `B` vertically to the end of `A` when `A` and `B` have compatible sizes (the lengths of the dimensions match except in the first dimension).

`C = vertcat(A1,A2,…,An)` concatenates `A1`, `A2`, … , `An` vertically.

`vertcat` is equivalent to using square brackets for vertically concatenating arrays. For example, `[A; B]` is equal to `vertcat(A,B)` when `A` and `B` are compatible arrays.

Examples

Two Matrices

Create two matrices and concatenate them vertically, first by using square bracket notation, and then by using `vertcat`.

```
A = [1 2 3; 4 5 6]
```

```
A = 2×3

     1     2     3
     4     5     6
```

```
B = [7 8 9]
```

```
B = 1×3

     7     8     9
```

```
C = [A; B]
```

```
C = 3×3

     1     2     3
     4     5     6
     7     8     9
```

```
D = vertcat(A,B)
```

```
D = 3×3

     1     2     3
     4     5     6
     7     8     9
```

Two Tables

Create a table A with three rows and five variables.

```
A = table([5;6;5],['M';'M';'M'],[45;41;40],[45;32;34],{'NY';'CA';'MA'},...
    'VariableNames',{'Age' 'Gender' 'Height' 'Weight' 'Birthplace'},...
    'RowNames',{'Thomas' 'Gordon' 'Percy'})
```

```
A=3×5 table
              Age    Gender    Height    Weight    Birthplace
              ___    ______    ______    ______    __________

    Thomas     5       M         45        45        {'NY'}
    Gordon     6       M         41        32        {'CA'}
    Percy      5       M         40        34        {'MA'}
```

Create a table B with the same variables as A except for order.

```
B = table(['F';'M';'F'],[6;6;5],{'AZ';'NH';'CO'},[31;42;33],[39;43;40],...
    'VariableNames',{'Gender' 'Age' 'Birthplace' 'Weight' 'Height'})
```

```
B=3×5 table
    Gender    Age    Birthplace    Weight    Height
    ______    ___    __________    ______    ______

      F        6       {'AZ'}        31        39
      M        6       {'NH'}        42        43
      F        5       {'CO'}        33        40
```

Vertically concatenate tables A and B. The variables of C are in the same order as the variables of A and default row names are used for the rows from B.

```
C = vertcat(A,B)
```

```
C=6×5 table
              Age    Gender    Height    Weight    Birthplace
              ___    ______    ______    ______    __________

    Thomas     5       M         45        45        {'NY'}
    Gordon     6       M         41        32        {'CA'}
    Percy      5       M         40        34        {'MA'}
    Row4       6       F         39        31        {'AZ'}
    Row5       6       M         43        42        {'NH'}
    Row6       5       F         40        33        {'CO'}
```

Dates with Different Types

Concatenate a date character vector, a string date, and a datetime into a single column of dates. The result is a datetime column vector.

```
chardate = '2016-03-24';
strdate = "2016-04-19";
t = datetime('2016-05-10','InputFormat','yyyy-MM-dd');
C = vertcat(chardate,strdate,t)
```

```
C = 3x1 datetime array
   24-Mar-2016
   19-Apr-2016
```

```
10-May-2016
```

String Array

Concatenate three string arrays into a single array.

```
A1 = ["str1" "str2"];
A2 = ["str3" "str4"];
A3 = ["str5" "str6"];
C = vertcat(A1,A2,A3)
```

```
C = 3x2 string array
    "str1"    "str2"
    "str3"    "str4"
    "str5"    "str6"
```

Matrices in a Cell Array

Create a cell array containing two matrices. Vertically concatenate the matrices from the cell array into one matrix.

```
M1 = [1 2; 3 4];
M2 = [5 6; 7 8];
A1 = {M1,M2};
C = vertcat(A1{:})
```

```
C = 4×2

     1     2
     3     4
     5     6
     7     8
```

Input Arguments

A — First input
scalar | vector | matrix | multidimensional array | table | timetable

First input, specified as a scalar, vector, matrix, multidimensional array, table, or timetable.

B — Second input
scalar | vector | matrix | multidimensional array | table | timetable

Second input, specified as a scalar, vector, matrix, multidimensional array, table, or timetable.

- The elements of B are concatenated to the end of the first input along the first dimension. The sizes of the input arguments must be compatible. For example, if the first input is a matrix of size 3-by-2, then B must have 2 columns.
- You can concatenate valid combinations of different types. For more information, see "Valid Combinations of Unlike Classes".

A1,A2,…,An — List of inputs
comma-separated list

List of inputs, specified as a comma-separated list of elements to concatenate in the order they are specified.

- The inputs must have compatible sizes. For example, if A1 is a row vector of length *m*, then the remaining inputs must each have *m* columns to concatenate vertically.
- You can concatenate valid combinations of different types. For more information, see "Valid Combinations of Unlike Classes".

Algorithms

When concatenating an empty array to a nonempty array, `vertcat` omits the empty array in the output. For example, `vertcat([1; 2],[])` returns the column vector `[1; 2]`.

If all input arguments are empty and have compatible sizes, then `vertcat` returns an empty array whose size is equal to the output size as when the inputs are nonempty. For example, `vertcat(zeros(1,0),zeros(2,0))` returns a 3-by-0 empty array. If the input sizes are not compatible, then `vertcat` returns a 0-by-0 empty array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- Vertical concatenation of character arrays is not supported.
- When vertically concatenating two tall arrays, the result is a tall array based on a different datastore than the input tall arrays. Therefore, if you try to combine the result of the concatenation with one of the input tall arrays, the arrays are incompatible. For example, the following command results in an error, since the original tall table `T` is based on one datastore, and the output of the concatenation operation is based on another datastore:

  ```
  T.Var3 = [T.Var1(1); T.Var2(2:end)]
  ```

  Alternatively, if you only concatenate in-memory values to a tall array, then the result of the concatenation is based on the same datastore. For example, if you replace `T.Var1(1)` in the previous example with an in-memory scalar, then the command executes:

  ```
  T.Var3 = [0; T.Var2(2:end)]
  ```

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`cat` | `horzcat`

Topics

“Creating, Concatenating, and Expanding Matrices”
“Valid Combinations of Unlike Classes”
“Combine Categorical Arrays”
“Concatenating Objects of Different Classes”
“Concatenation Methods”

Introduced before R2006a

vertcat

Vertically concatenate `tscollection` objects

Syntax

```
tscout = vertcat(tsc1,tsc2,…,tscN)
```

Description

`tscout = vertcat(tsc1,tsc2,…,tscN)` vertically concatenates the `tscollection` objects `tsc1`, `tsc2`,..., `tscN` by appending them. The time vectors of the `tscollection` objects in the list must not overlap. The last time in a `tscollection` must occur before the first time of the following `tscollection` in the list. All `tscollection` objects must have the same `timeseries` members.

Examples

Vertical Concatenation

Create two `tscollection` objects and vertically concatenate them.

```
ts1 = timeseries([1.1 2.9 3.7 4.0 3.0]',1:5,...
                 'Name','Speed');
ts2 = timeseries([3.2 4.2 6.2 8.5 1.1]',6:10,...
                 'Name','Speed');
tsc1 = tscollection(ts1);
tsc2 = tscollection(ts2);
tscout = vertcat(tsc1,tsc2)
```

```
Time Series Collection Object: unnamed

Time vector characteristics

      Start time            1 seconds
      End time              10 seconds
```

```
Member Time Series Objects:

      Speed
```

Input Arguments

`tsc1,tsc2,…,tscN` — List of `tscollection` objects
comma-separated list

List of `tscollection` objects to concatenate, specified as a comma-separated list of scalar `tscollection` objects.

See Also

`horzcat` | `timeseries` | `tscollection`

Introduced before R2006a

vertexAttachments

Class: `TriRep`

(Not recommended) Return simplices attached to specified vertices

---

Note `vertexAttachments(TriRep)` is not recommended. Use `vertexAttachments(triangulation)` instead.

`TriRep` is not recommended. Use `triangulation` instead.

Syntax

```
SI = vertexAttachments(TR, VI)
```

Description

`SI = vertexAttachments(TR, VI)` returns the vertex-to-simplex information for the specified vertices `VI`. For 2-D triangulations in MATLAB, the triangles `SI` are arranged in counter-clockwise order around the attached vertex `VI`.

Input Arguments

`TR`	Triangulation representation
`VI`	`VI` is a column vector of indices into the array of points representing the vertex coordinates, `TR.X`. The simplices associated with vertex `i` are the `i`'th entry in the cell array. If `VI` is not specified the vertex-simplex information for the entire triangulation is returned.

Output Arguments

`SI`	Cell array of indices of the simplices attached to a vertex. A cell array is used to store the information because the number of simplices associated with each vertex can vary. The simplices associated with vertex `i` are in the `i`'th entry in the cell array `SI`.

Examples

Example 1

Load a 2-D triangulation and use `TriRep` to compute the vertex-to-triangle relations.

```
load trimesh2d
trep = TriRep(tri, x, y);
```

Find the indices of the tetrahedra attached to the first vertex:

```
Tv = vertexAttachments(trep, 1)
Tv{:}
```

Example 2

Perform a direct query of a 2-D triangulation created using `DelaunayTri`.

```
x = rand(20,1);
y = rand(20,1);
dt = DelaunayTri(x,y);
```

Find the triangles attached to vertex 5:

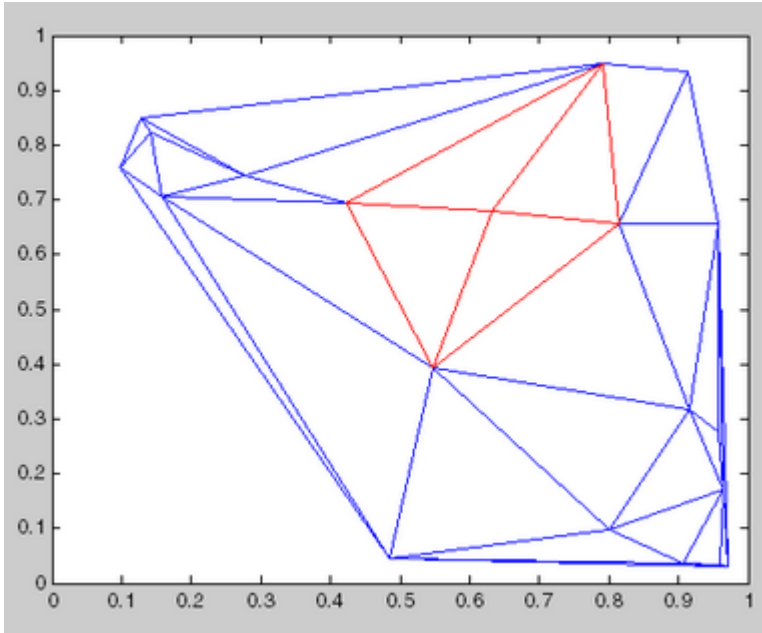
```
t = vertexAttachments(dt,5);
```

Plot the triangulation:

```
triplot(dt);
hold on;
```

Plot the triangles attached to vertex 5 (in red):

```
triplot(dt(t{:},:),x,y,'Color','r');
hold off;
```



More About

Simplex

A simplex is a triangle/tetrahedron or higher dimensional equivalent.

See Also

`delaunayTriangulation` | `triangulation`

VideoReader

Create object to read video files

Description

Use a `VideoReader` object to read files containing video data. The object contains information about the video file and enables you to read data from the video. You can create a `VideoReader` object using the `VideoReader` function, query information about the video using the object properties, and then read the video using object functions.

For more information, see “Supported Video and Audio File Formats”.

Creation

Syntax

```
v = VideoReader(filename)
v = VideoReader(filename,Name,Value)
```

Description

`v = VideoReader(filename)` creates object `v` to read video data from the file named `filename`.

`v = VideoReader(filename,Name,Value)` sets the properties `CurrentTime`, `Tag`, and `UserData` using name-value pairs. For example, `VideoReader('myfile.mp4','CurrentTime',1.2)` starts reading `1.2` seconds into the video. You can specify multiple name-value pairs. Enclose each property name in single quotes followed by the corresponding value.

Input Arguments

`filename` — File name
character vector | string scalar

File name, specified as a character vector or string scalar.

For more information, see “Supported Video and Audio File Formats”.

Example: `'myFile.mp4'`

Example: `'../dir/videos/myFile.avi'`

Data Types: `char` | `string`

Properties

The `VideoReader` object has properties that contain information about the video file. Properties are read-only, except `CurrentTime`, `Tag`, and `UserData`. You can view or modify the value of a property after creating the object. For example, this command finds the value of the `Duration` property of the `VideoReader` object, `v`.

```
D = v.Duration;
```

`BitsPerPixel` — Bits per pixel of video data
numeric scalar

This property is read-only.

Bits per pixel of the video data, specified as a numeric scalar.

Data Types: `double`

`CurrentTime` — Timestamp of video frame to read
numeric scalar

Timestamp of the video frame to read, specified as a numeric scalar. The timestamp is specified in seconds from the start of the video file. The value of `CurrentTime` can be between zero and the duration of the video.

On some platforms, when you create a `VideoReader` object, the `'CurrentTime'` property might contain a value close to, but not exactly, zero. This variation in the value of

the `'CurrentTime'` property is due to differences in how each platform processes and reads videos.

Example: `5.6`

Data Types: `double`

`Duration` — Length of file
numeric scalar

This property is read-only.

Length of the file in seconds, specified as a numeric scalar.

Data Types: `double`

`FrameRate` — Number of video frames per second
numeric scalar

This property is read-only.

Number of video frames per second, specified as a numeric scalar. For variable-frame rate video, `FrameRate` is the average frame rate.

Note: For OS X Yosemite (Version 10.10) and later, MPEG-4/H.264 files written using `VideoWriter` play correctly, but display an inexact frame rate.

Data Types: `double`

`Height` — Height of video frame
numeric scalar

This property is read-only.

Height of the video frame in pixels, specified as a numeric scalar.

Data Types: `double`

`Name` — File name
character vector | string scalar

This property is read-only.

File name, specified as a character vector or string scalar.

Data Types: `char` | `string`

`NumFrames` — Number of frames in video stream
numeric scalar

This property is read-only.

Number of frames in the video stream, specified as a numeric scalar.

Note For certain length videos, the value of the `NumFrames` property is not immediately available. To get the `NumFrames` property, type `v.NumFrames` in the command line.

Data Types: `double`

`Path` — Full path to video file
character vector | string scalar

This property is read-only.

Full path to the video file associated with the reader object, specified as a character vector or string scalar.

Data Types: `char` | `string`

`Tag` — Generic text
`''` (default) | character vector | string scalar

Generic text, specified as a character vector or string scalar.

Example: `'Experiment 109'`

Data Types: `char` | `string`

`UserData` — User-defined data
`[]` (default) | any data type

User-defined data, specified as a value of any data type.

`VideoFormat` — MATLAB representation of video format
character vector | string scalar

This property is read-only.

MATLAB representation of the video format, specified as a character vector or string scalar.

File types, except for Motion JPEG 2000 files, have one of these `VideoFormat` values.

Video Format	Value of `VideoFormat`
AVI or MPEG-4 files with RGB24 video	`'RGB24'`
AVI files with indexed video	`'Indexed'`
AVI files with grayscale video	`'Grayscale'`

Motion JPEG 2000 files, have one of the following `VideoFormat` values.

Format of Image Data	Value of `VideoFormat`
Single-band `uint8`	`'Mono8'`
Single-band `int8`	`'Mono8 Signed'`
Single-band `uint16`	`'Mono16'`
Single-band `int16`	`'Mono16 Signed'`
Three-banded `uint8`	`'RGB24'`
Three-banded `int8`	`'RGB24 Signed'`
Three-banded `uint16`	`'RGB48'`
Three-banded `int16`	`'RGB48 Signed'`

Data Types: `char` | `string`

`Width` — Width of video frame
numeric scalar

This property is read-only.

Width of the video frame in pixels, specified as a numeric scalar.

Data Types: `double`

Object Functions

hasFrame	Determine if video frame is available to read
read	Read one or more video frames
readFrame	Read next video frame
VideoReader.getFileFormats	File formats that VideoReader supports

Examples

Create VideoReader Object and Read Video

Create a `VideoReader` object for the example movie file `xylophone.mp4`.

```
v = VideoReader('xylophone.mp4');
```

Read all the frames from the video, one frame at a time.

```
while hasFrame(v)
    frame = readFrame(v);
end
```

Display information about the last frame returned by `readFrame`.

```
whos frame
```

```
  Name         Size                Bytes  Class    Attributes

  frame      240x320x3            230400  uint8
```

Read Video Frames Starting At Specific Time

Create a video reader object and read video frames starting at a specific time.

Create an object to read the example movie file `xylophone.mp4`.

```
v = VideoReader('xylophone.mp4');
```

Specify the reading to begin 2.5 seconds from the beginning of the video.

```
v.CurrentTime = 2.5;
```

Create an axes object to display the frame. Then, continue to read and display video frames until no more frames are available to read.

```
currAxes = axes;
while hasFrame(v)
    vidFrame = readFrame(v);
    image(vidFrame, 'Parent', currAxes);
```

```
    currAxes.Visible = 'off';
    pause(1/v.FrameRate);
end
```

Read Video Frames Using Frame Index

Create a video reader object and read one or more video frames using the frame indices.

Create a `VideoReader` object for the example movie file `xylophone.mp4`.

```
v = VideoReader('xylophone.mp4');
```

Read only the first video frame.

```
frame = read(v,1);
```

Read only the last video frame.

```
frame = read(v,Inf);
```

Read frames 5 through 10.

```
frames = read(v,[5 10]);
```

Read from the 50th frame to the end of the video file.

```
frames = read(v,[50 Inf]);
```

Read Video Using Frame Index and Time Interchangeably

Read a frame from a video by specifying a frame index, and then read the remaining frames of the video one frame at a time.

Create a `VideoReader` object and display the value of the `CurrentTime` property. For this video, a zero value for the `CurrentTime` property indicates that no frames have been read from the video.

```
vidObj = VideoReader('xylophone.mp4');
vidObj.CurrentTime
```

```
ans = 0
```

Read the 20th frame from the video by specifying the frame index. Then, display the value of the `CurrentTime` property. The `read` method automatically updates the `CurrentTime` property to reflect that the 20th frame has been read. At this step, a call to the `readFrame` function would return the 21st frame.

```
frame20 = read(vidObj,20);
vidObj.CurrentTime
```

```
ans = 0.6667
```

Read the remaining frames of the video using the `readFrame` method. The `readFrame` method returns the frame corresponding to the time in the `CurrentTime` property. For instance, this code reads and displays the frames starting at the 21st frame and continues until there are no more frames to read.

```
while(hasFrame(vidObj))
    frame = readFrame(vidObj);
    imshow(frame);
    title(sprintf('Current Time = %.3f sec', vidObj.CurrentTime));
    pause(2/vidObj.FrameRate);
end
```

Copyright 2019 The MathWorks, Inc.

Tips

- The macOS platform no longer supports certain older video file formats. To read such files using `VideoReader`:
  - Open the video file using the QuickTime player. If the player detects the file to be of an older format, then it automatically converts the file to a newer format.
  - Save the newly converted video file.
  - Use `VideoReader` to read this newly converted video file.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Code generation for `VideoReader` supports most formats, syntaxes, methods, and functions with the following limitations.

- Video Format Support:
  - If `filename` is a compile-time constant, then code generation supports all the formats supported in MATLAB. For more information on video formats that MATLAB supports, see “Supported Video and Audio File Formats”.
  - If `filename` is not a compile-time constant, then code generation supports only video files with data that can be decoded to `uint8` datatype. Supported video formats include: `.MP4`, `.MOV`, and `.AVI`.
- Object Construction:
  - For MEX targets, partial path to the video file is supported.
  - For RTW targets, you must provide full or relative path to the video file.
- Methods and Functions :
  - `read` and `readFrame` — Code generation does not support the optional positional argument `native`.
  - `VideoReader.getFileFormats` — Code generation does not support this method.
  - `saveobj` and `loadobj` — Code generation does not support these functions.
  - `inspect` — Code generation does not support this function.
  - Code generation does not support `VideoReader` object display.
- Platform Dependencies — If the generated code for `VideoReader` on one specific machine does not work on another machine, then:
  - Ensure that the suitable codecs for your video are available on the target machine.

- Add test code to check if the video object created on the target machine is valid. Test code can include checking if the video object has valid height or width. For example:

```
videoObj = VideoReader(filename);
if isnan(videoObj.Height)
   fprintf('Failed to create video object.\n');
   return
end
```

- Generate Code That Uses Row-Major Layout — “Generate Code That Uses Row-Major Array Layout” (MATLAB Coder).
- Array Size Restrictions — For code generation, the maximum number of elements of an array is constrained by the code generator and the target hardware. For more information, see “Array Size Restrictions for Code Generation” (MATLAB Coder).

See Also

`VideoWriter` | `mmfileinfo`

Topics

“Read Video Files”

Introduced in R2010b

VideoReader.getFileFormats

File formats that `VideoReader` supports

Syntax

```
formats = VideoReader.getFileFormats()
```

Description

`formats = VideoReader.getFileFormats()` returns an array of `audiovideo.FileFormatInfo` objects that indicate which formats `VideoReader` can read on the current system.

Examples

View Supported File Formats for Your System

View the list of file formats that `VideoReader` supports on your system.

```
fmts = VideoReader.getFileFormats()
```

```
Video File Formats:
    .asf - ASF File
    .asx - ASX File
    .avi - AVI File
    .m4v - MPEG-4 Video
    .mj2 - Motion JPEG2000
    .mov - QuickTime movie
    .mp4 - MPEG-4
    .mpg - MPEG-1
    .wmv - Windows Media Video
```

This output is for a Linux® system. Your results might differ.

Check if File Format is Supported

Check whether `VideoReader` can read AVI files on the current system.

```
fmtList = VideoReader.getFileFormats();

if any(ismember({fmtList.Extension},'avi'))
     disp('VideoReader can read AVI files on this system.');
else
     disp('VideoReader cannot read AVI files on this system.');
end
```

```
VideoReader can read AVI files on this system.
```

Create Dialog Box to Select Video File

Create a dialog box that allows a user to select a video file interactively.

Get the supported file formats.

```
formats = VideoReader.getFileFormats()
```

```
Video File Formats:
    .asf - ASF File
    .asx - ASX File
    .avi - AVI File
    .m4v - MPEG-4 Video
    .mj2 - Motion JPEG2000
    .mov - QuickTime movie
    .mp4 - MPEG-4
    .mpg - MPEG-1
    .wmv - Windows Media Video
```

This output is for a Windows system. Your results might differ.

Convert the `formats` array to a filter list.

```
filterSpec = getFilterSpec(formats)
```

```
filterSpec =
```

```
'*.asf;*.asx;*.avi;*.m4v;*.mj2;*.mov;*.mp4;*.mpg;*.wmv;'     'All Video Files'
'*.asf'                                                      'ASF File (*.asf)'
'*.asx'                                                      'ASX File (*.asx)'
'*.avi'                                                      'AVI File (*.avi)'
'*.m4v'                                                      'MPEG-4 Video (*.m4v)'
'*.mj2'                                                      'Motion JPEG2000 (*.mj2
'*.mov'                                                      'QuickTime movie (*.mov
'*.mp4'                                                      'MPEG-4 (*.mp4)'
'*.mpg'                                                      'MPEG-1 (*.mpg)'
'*.wmv'                                                      'Windows Media Video (*
'*.*'                                                        'All Files (*.*)'
```

`filterSpec` is a cell array that includes `'All Video Files'` in the first row, and `'All Files (*.*)'` in the last row.

Create the dialog box using `uigetfile`.

```
[filename,pathname] = uigetfile(filterSpec);
```

Output Arguments

`formats` — Supported video file formats
array of `audiovideo.FileFormatInfo` objects

Supported video file formats, returned as an array of `audiovideo.FileFormatInfo` objects with the following properties.

`Extension`	File extension.
`Description`	Text description of the file format.
`ContainsVideo`	Logical value that indicates whether `VideoReader` can read video from this format.
`ContainsAudio`	Logical value that indicates whether `VideoReader` can read audio from this format.

`VideoReader` cannot always read a particular video file even if `getFileFormats` lists its format. For more information, see Supported Video File Formats.

On Windows and UNIX systems, the list of file formats does not always contain all the formats that `VideoReader` can read on your system. `getFileFormats` returns a platform-dependent, static list of formats that `VideoReader` can read on most systems.

See Also

`VideoReader` | `uigetfile`

Introduced in R2010b

hasFrame

Determine if video frame is available to read

Syntax

```
tf = hasFrame(v)
```

Description

`tf = hasFrame(v)` returns logical `1` (`true`) if there is a video frame available to read from the file. Otherwise, it returns logical `0` (`false`).

Examples

Create VideoReader Object and Read Video

Create a `VideoReader` object for the example movie file `xylophone.mp4`.

```
v = VideoReader('xylophone.mp4');
```

Read all the frames from the video, one frame at a time.

```
while hasFrame(v)
    frame = readFrame(v);
end
```

Display information about the last frame returned by `readFrame`.

```
whos frame
```

```
  Name          Size                 Bytes  Class    Attributes

  frame       240x320x3             230400  uint8
```

Read and Play Back Movie File

Read and play back the sample movie file, `xylophone.mp4`.

Create a `VideoReader` object to read data from the sample file. Then, determine the width and height of the video.

```
xyloObj = VideoReader('xylophone.mp4');

vidWidth = xyloObj.Width;
vidHeight = xyloObj.Height;
```

Create a movie structure array, `mov`.

```
mov = struct('cdata',zeros(vidHeight,vidWidth,3,'uint8'),...
    'colormap',[]);
```

Read one frame at a time until the end of the video is reached.

```
k = 1;
while hasFrame(xyloObj)
    mov(k).cdata = readFrame(xyloObj);
    k = k+1;
end
```

Size a figure based on the width and height of the video. Then, play back the movie once at the video frame rate.

```
hf = figure;
set(hf,'position',[150 150 vidWidth vidHeight]);

movie(hf,mov,1,xyloObj.FrameRate);
```

Input Arguments

v — Input VideoReader object
`VideoReader` object

Input VideoReader object. Use the `VideoReader` function to create a `VideoReader` object from your video file.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`VideoReader` | `movie` | `readFrame`

Topics

"Read Video Files"

Introduced in R2014b

read

Read one or more video frames

Syntax

```
video = read(v)
video = read(v,index)
video = read( ___ ,'native')
```

Description

`video = read(v)` reads all video frames from the file associated with `v`.

`video = read(v,index)` reads only the frames specified by `index`.

`video = read( ___ ,'native')` returns data in the format specified by the `VideoFormat` property and can include any of the input arguments in the previous syntaxes.

Examples

Read Video Frames Using Frame Index

Create a video reader object and read one or more video frames using the frame indices.

Create a `VideoReader` object for the example movie file `xylophone.mp4`.

```
v = VideoReader('xylophone.mp4');
```

Read only the first video frame.

```
frame = read(v,1);
```

Read only the last video frame.

```
frame = read(v,Inf);
```

Read frames 5 through 10.

```
frames = read(v,[5 10]);
```

Read from the 50th frame to the end of the video file.

```
frames = read(v,[50 Inf]);
```

Input Arguments

v — Input VideoReader object
`VideoReader` object

Input VideoReader object. Use the `VideoReader` function to create a VideoReader object from your video file.

`index` — Frames to read
`[1 Inf]` (default) | numeric scalar | two-element array

Frames to read, specified as a numeric scalar or a two-element array.

- To read a single frame, specify a numeric scalar.
- To read a range of frames, specify a two-element array.

The first frame number is 1, and `Inf` represents the last frame of the file.

Example: `10`

Example: `Inf`

Example: `[1 20]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`'native'` — Flag to use format specified in `VideoFormat` property
character vector | string scalar

Flag to use format specified in `VideoFormat` property, specified as the character vector or a string scalar, `'native'`.

Data Types: `char` | `string`

Output Arguments

`video` — Video frame data
numeric array | structure array

Video frame data, returned as a numeric or structure array. The size and format of `video` depends on the `VideoFormat` property of `v`. When the `VideoFormat` property of `v` is `'Indexed'` or `'Grayscale'`, the data type and dimensions of `video` depend on whether you call `read` with the `'native'` argument. For most files, the data type and dimensions of `video` are as follows.

`H` is the image frame height, `W` is the image frame width, and `F` is the number of frames read.

Value of `obj.VideoFormat`	Data Type of `video`	Dimensions of `video`	Description
`'RGB24'`, with or without specifying `'native'`	`uint8`	`H`-by-`W`-by-3-by-`F`	RGB24 image
`'Grayscale'`, without specifying `'native'`	`uint8`	`H`-by-`W`-by-1-by-`F`	Grayscale image
`'Indexed'`, without specifying `'native'`	`uint8`	`H`-by-`W`-by-3-by-`F`	RGB24 image
`'Grayscale'` or `'Indexed'`, specifying `'native'`	`struct`	1-by-`F`	MATLAB movie, which is an array of frame structure arrays, each containing the fields `cdata` and `colormap`.

For Motion JPEG 2000 files, the data type and dimensions of `video` are as follows.

Value of obj.VideoFormat	Data Type of video	Dimensions of video	Description
'Mono8'	uint8	H-by-W-by-1-by-F	Mono image
'Mono8 Signed'	int8	H-by-W-by-1-by-F	Mono signed image
'Mono16'	uint16	H-by-W-by-1-by-F	Mono image
'Mono16 Signed'	int16	H-by-W-by-1-by-F	Mono signed image
'RGB24'	uint8	H-by-W-by-3-by-F	RGB24 image
'RGB24 Signed'	int8	H-by-W-by-3-by-F	RGB24 signed image
'RGB48'	uint16	H-by-W-by-3-by-F	RGB48 image
'RGB48 Signed'	int16	H-by-W-by-3-by-F	RGB48 signed image

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation for `read` does not support the optional positional argument `native`.
- Generate Code That Uses Row-Major Layout — “Generate Code That Uses Row-Major Array Layout” (MATLAB Coder).

See Also

`VideoReader` | `movie` | `readFrame`

Introduced in R2010b

readFrame

Read next video frame

Syntax

```
video = readFrame(v)
video = readFrame(v,'native')
```

Description

`video = readFrame(v)` reads the next available video frame from the file associated with `v`.

`video = readFrame(v,'native')` returns data in the format specified by the `VideoFormat` property.

Examples

Create VideoReader Object and Read Video

Create a `VideoReader` object for the example movie file `xylophone.mp4`.

```
v = VideoReader('xylophone.mp4');
```

Read all the frames from the video, one frame at a time.

```
while hasFrame(v)
    frame = readFrame(v);
end
```

Display information about the last frame returned by `readFrame`.

```
whos frame
```

```
  Name         Size                Bytes  Class    Attributes

  frame      240x320x3            230400  uint8
```

Read Video Frames Starting At Specific Time

Create a video reader object and read video frames starting at a specific time.

Create an object to read the example movie file `xylophone.mp4`.

```
v = VideoReader('xylophone.mp4');
```

Specify the reading to begin 2.5 seconds from the beginning of the video.

```
v.CurrentTime = 2.5;
```

Create an axes object to display the frame. Then, continue to read and display video frames until no more frames are available to read.

```
currAxes = axes;
while hasFrame(v)
    vidFrame = readFrame(v);
    image(vidFrame, 'Parent', currAxes);
    currAxes.Visible = 'off';
    pause(1/v.FrameRate);
end
```

Read and Play Back Movie File

Read and play back the sample movie file, `xylophone.mp4`.

Create a `VideoReader` object to read data from the sample file. Then, determine the width and height of the video.

```
xyloObj = VideoReader('xylophone.mp4');

vidWidth = xyloObj.Width;
vidHeight = xyloObj.Height;
```

Create a movie structure array, `mov`.

```
mov = struct('cdata',zeros(vidHeight,vidWidth,3,'uint8'),...
    'colormap',[]);
```

Read one frame at a time until the end of the video is reached.

```
k = 1;
while hasFrame(xyloObj)
    mov(k).cdata = readFrame(xyloObj);
    k = k+1;
end
```

Size a figure based on the width and height of the video. Then, play back the movie once at the video frame rate.

```
hf = figure;
set(hf,'position',[150 150 vidWidth vidHeight]);

movie(hf,mov,1,xyloObj.FrameRate);
```

Input Arguments

v — Input VideoReader object

`VideoReader` object

Input VideoReader object. Use the `VideoReader` function to create a VideoReader object from your video file.

Output Arguments

video — Video frame data

array

Video frame data, returned as an array. The dimensions and data type of `video` depend on the `VideoFormat` property of `obj`.

The following table lists the data type and dimensions of `video` for most files. `H` is the image frame height and `W` is the image frame width. When the `VideoFormat` property of `obj` is `'Indexed'`, the data type and dimensions of `video` depend on whether you call `read` with the `'native'` argument.

Value of obj.VideoFormat	Data Type of video	Dimensions of video	Description
'RGB24', with or without specifying 'native'	uint8	H-by-W-by-3	RGB24 image
'Grayscale', without specifying 'native'	uint8	H-by-W-by-1	Grayscale image
'Indexed', without specifying 'native'	uint8	H-by-W-by-3	RGB24 image
'Grayscale' or 'Indexed', specifying 'native'	struct	1-by-1	MATLAB movie, which is an array of frame structure arrays, each containing the fields cdata and colormap.

For Motion JPEG 2000 files, the data type and dimensions of video are as follows.

Value of obj.VideoFormat	Data Type of video	Dimensions of video	Description
'Mono8'	uint8	H-by-W-by-1	Mono image
'Mono8 Signed'	int8	H-by-W-by-1	Mono signed image
'Mono16'	uint16	H-by-W-by-1	Mono image
'Mono16 Signed'	int16	H-by-W-by-1	Mono signed image
'RGB24'	uint8	H-by-W-by-3	RGB24 image
'RGB24 Signed'	int8	H-by-W-by-3	RGB24 signed image
'RGB48'	uint16	H-by-W-by-3	RGB48 image
'RGB48 Signed'	int16	H-by-W-by-3	RGB48 signed image

'native' — Flag to use format specified in VideoFormat property
character vector | string scalar

Flag to use format specified in VideoFormat property, specified as the character vector or a string scalar, 'native'.

Data Types: `char` | `string`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation for `readFrame` does not support the optional positional argument `native`.
- Generate Code That Uses Row-Major Layout — "Generate Code That Uses Row-Major Array Layout" (MATLAB Coder).
- Array Size Restrictions — For code generation, the maximum number of elements of an array is constrained by the code generator and the target hardware. For more information, see "Array Size Restrictions for Code Generation" (MATLAB Coder).

See Also

`VideoReader` | `hasFrame` | `movie` | `read`

Topics

"Read Video Files"

Introduced in R2014b

VideoWriter

Create object to write video files

Description

Use a `VideoWriter` object to create a video file from an array or MATLAB movie. The object contains information about the video and the properties that control the output video. You can create a `VideoWriter` object using the `VideoWriter` function, specify its properties, and then write the video using object functions.

Creation

Syntax

```
v = VideoWriter(filename)
v = VideoWriter(filename,profile)
```

Description

`v = VideoWriter(filename)` creates a `VideoWriter` object to write video data to an AVI file with Motion JPEG compression.

`v = VideoWriter(filename,profile)` additionally applies a set of properties tailored to a specific file format (such as `'MPEG-4'` or `'Uncompressed AVI'`).

Input Arguments

`filename` — File name
character vector | string scalar

File name, specified as a character vector or string scalar. `VideoWriter` creates the file.

`VideoWriter` supports these file extensions.

`.avi`	AVI file
`.mj2`	Motion JPEG 2000 file
`.mp4` or `.m4v`	MPEG-4 file (systems with Windows 7 or later, or macOS 10.7 and later)

If you do not specify a valid file extension, `VideoWriter` appends the extension `.avi`, `.mj2`, or `.mp4`, depending on the value of the `profile` argument. If you do not specify a value for `profile`, then `VideoWriter` creates a Motion JPEG compressed AVI file with the extension `.avi`.

Example: `'myFile.avi'`

Example: `'../dir/videos/myFile.mj2'`

Data Types: `char` | `string`

`profile` — File type
`'Motion JPEG AVI'` (default) | `'Archival'` | `'Motion JPEG 2000'` | ...

File type, specified as one of these.

Value of `profile`	Description
`'Archival'`	Motion JPEG 2000 file with lossless compression
`'Motion JPEG AVI'`	AVI file using Motion JPEG encoding
`'Motion JPEG 2000'`	Motion JPEG 2000 file
`'MPEG-4'`	MPEG-4 file with H.264 encoding (systems with Windows 7 or later, or macOS 10.7 and later)
`'Uncompressed AVI'`	Uncompressed AVI file with `RGB24` video
`'Indexed AVI'`	Uncompressed AVI file with indexed video
`'Grayscale AVI'`	Uncompressed AVI file with grayscale video

`profile` sets default values for video properties, such as `VideoCompressionMethod`.

Data Types: `char` | `string`

Properties

The `VideoWriter` object contains properties that control the output video. You can create a `VideoWriter` object with properties tailored to a specific format, such as `'Uncompressed AVI'` or `'MPEG-4'`, by using the `VideoWriter` function with a predefined *profile*. For example, you can create a `VideoWriter` object with a `Motion JPEG AVI` profile and assign a value to the `Quality` property.

```
v = VideoWriter('newfile.avi','Motion JPEG AVI');
v.Quality = 95;
```

After you call the `open` function on the `VideoWriter` object, you cannot change the value of a property. Therefore, modify property values before opening the video file for writing.

`ColorChannels` — Number of color channels

positive integer

This property is read-only.

Number of color channels in each output video frame, specified as a positive integer:

- Uncompressed AVI, Motion JPEG AVI, and MPEG-4 files have three color channels.
- Indexed and grayscale AVI files have one color channel.
- For Motion JPEG 2000 files, the number of channels depends on the input data to the `writeVideo` function: one for monochrome data or three for color data.

Data Types: `double`

`Colormap` — Color information for video file

P-by-3 numeric matrix

Color information for video file, specified as a numeric matrix with three columns and a maximum of 256 rows. Each row in the matrix defines one color using an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`.

You can set the colormap explicitly before the call to `open`, or by using the `colormap` field of a movie frame structure at the time of writing the first frame.

The `Colormap` property only applies to objects used for writing indexed AVI files.

Example: `colormap(summer(256))`

Data Types: `double` | `uint8`

`CompressionRatio` — Target compression ratio
`10` (default) | integer greater than 1

Target compression ratio, specified as an integer greater than 1. The compression ratio is the ratio between the number of bytes in the input image and the number of bytes in the compressed image. The video data is compressed as much as possible, up to the specified target.

`CompressionRatio` is available only for objects used for writing Motion JPEG 2000 files. After you call `open`, you cannot change the `CompressionRatio` value. If you previously set `LosslessCompression` to `true`, then setting `CompressionRatio` generates an error.

Example: `5`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Duration` — Duration of output file
scalar value

This property is read-only.

Duration of the output file in seconds, specified as a scalar value.

Data Types: `double`

`FileFormat` — Type of file to write
`'avi'` | `'mp4'` | `'mj2'`

This property is read-only.

Type of file to write, specified as `'avi'`, `'mp4'`, or `'mj2'`.

Data Types: `char` | `string`

`Filename` — Name of file
character vector | string scalar

This property is read-only.

Name of file, specified as a character vector or string scalar.

Data Types: `char` | `string`

`FrameCount` — Number of frames
integer

This property is read-only.

Number of frames written to the video file, specified as an integer.

Data Types: `double`

`FrameRate` — Rate of video playback
`30` (default) | positive number

Rate of video playback in frames per second, specified as a positive number.

After you call `open`, you cannot change the `FrameRate` value.

Example: `10`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Height` — Height of each video frame
scalar

This property is read-only.

Height of each video frame in pixels, specified as a scalar. The `writeVideo` method sets values for `Height` and `Width` based on the dimensions of the first frame.

MPEG-4 files require frame dimensions that are divisible by two. If the input frame height for an MPEG-4 file is not an even number, then `VideoWriter` pads the frame with a row of black pixels at the bottom. For MPEG-4 files on Windows systems, the allowed value depends on the version of Windows.

Data Types: `double`

`LosslessCompression` — Lossless compression
`true` | `false`

Lossless compression, specified as `true` or `false`. The `LosslessCompression` property is only available for objects used for writing Motion JPEG 2000 files.

If `LosslessCompression` is `true`, then:

- The `writeVideo` function writes data so that the decompressed data is identical to the input data.
- `VideoWriter` ignores any specified value for `CompressionRatio`.

After you call `open`, you cannot change the `LosslessCompression` value.

By default, `LosslessCompression` is `false` for the `'Motion JPEG 2000'` profile and `true` for the `'Archival'` profile.

Data Types: `logical`

`MJ2BitDepth` — Bit depth for Motion JPEG 2000 files
integer in the range `[1,16]`

Bit depth for Motion JPEG 2000 files, specified as an integer in the range `[1,16]`. The bit depth is the number of least-significant bits in the input image data

`MJ2BitDepth` is available only for objects used for writing Motion JPEG 2000 files. If you do not specify a value before calling the `open` method, then `VideoWriter` sets the bit depth based on the input data type. For example, if the input data to `writeVideo` is an array of `uint8` or `int8` values, then `MJ2BitDepth` is `8`.

Example: `8`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Path` — Full path to video file
character vector | string scalar

This property is read-only.

Full path to the video file, specified as a character vector or string scalar.

Data Types: `char` | `string`

`Quality` — Video quality
`75` (default) | integer in the range `[0,100]`

Video quality, specified as an integer in the range, `[0,100]`. Higher quality numbers result in higher video quality and larger file sizes. Lower quality numbers result in lower video quality and smaller file sizes.

`Quality` is available only for objects associated with the `MPEG-4` or `Motion JPEG AVI` profile. After you call `open`, you cannot change the `Quality` value.

Example: `50`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`VideoBitsPerPixel` — Number of bits per pixel
numeric scalar

This property is read-only.

Number of bits per pixel in each output video frame, specified as a numeric scalar.

AVI files with truecolor video, Motion JPEG AVI, and MPEG-4 files have 24 bits per pixel (8 bits for each of three color bands).

Indexed and grayscale AVI files have 8 bits per pixel.

For Motion JPEG 2000 files, the number of bits per pixel depends on the value of `MJ2BitDepth` and the number of bands of image data. For example, if the input data to `writeVideo` is a three-dimensional array of `uint16` or `int16` values, then the default value of `MJ2BitDepth` is `16`, and `VideoBitsPerPixel` is `48` (three times the bit depth).

Data Types: `double`

`VideoCompressionMethod` — Type of video compression
`'None'` | `'H.264'` | `'Motion JPEG'` | `'Motion JPEG 2000'`

This property is read-only.

Type of video compression, specified as `'None'`, `'H.264'`, `'Motion JPEG'`, or `'Motion JPEG 2000'`.

Data Types: `char` | `string`

`VideoFormat` — MATLAB representation of video format
character vector | string scalar

This property is read-only.

MATLAB representation of the video format, specified as a character vector or string scalar.

For file types other than Motion JPEG 2000 files, `VideoWriter` sets `VideoFormat` to one of the following.

Video Format	Value of `VideoFormat`
Uncompressed AVI, Motion JPEG AVI, or MPEG-4 files	`'RGB24'`
AVI files with indexed video	`'Indexed'`
AVI files with grayscale video	`'Grayscale'`

For Motion JPEG 2000 files, `VideoWriter` sets the `VideoFormat` based on the value of `MJ2BitDepth` and the format of the input image data to the `writeVideo` method. For example, if you do not specify the `MJ2BitDepth` property, then `VideoWriter` sets the format as shown here.

Format of Image Data	Value of `VideoFormat`
Single-band `uint8`	`'Mono8'`
Single-band `int8`	`'Mono8 Signed'`
Single-band `uint16`	`'Mono16'`
Single-band `int16`	`'Mono16 Signed'`
Three-banded `uint8`	`'RGB24'`
Three-banded `int8`	`'RGB24 Signed'`
Three-banded `uint16`	`'RGB48'`
Three-banded `int16`	`'RGB48 Signed'`

Data Types: `char` | `string`

`Width` — Width of each video frame
numeric scalar

This property is read-only.

Width of each video frame in pixels, specified as a numeric scalar. The `writeVideo` function sets values for `Height` and `Width` based on the dimensions of the first frame.

MPEG-4 files require frame dimensions that are divisible by two. If the input frame width for an MPEG-4 file is not an even number, then `VideoWriter` pads the frame with a

column of black pixels along the right side. For MPEG-4 files on Windows systems, the allowed value depends on the version of Windows.

Data Types: `double`

Object Functions

open	Open file for writing video data
close	Close file after writing video data
writeVideo	Write video data to file
VideoWriter.getProfiles	Profiles and file formats that VideoWriter supports

Examples

Create VideoWriter Object and Write Video

Write a video to a file by creating an array with random data, making a `VideoWriter` object for the output file, and then writing the array to the video.

Create a 300-by-300 matrix of data.

```
A = rand(300);
```

Create a `VideoWriter` object to write a Motion JPEG AVI file named `newfile.avi` and open the file for writing.

```
v = VideoWriter('newfile.avi');
open(v)
```

Write the matrix of data `A` to the video file.

```
writeVideo(v,A)
```

Close the file.

```
close(v)
```

Specify Profile for Uncompressed AVI and Write Video

Create a video object with a specified profile, make an array containing an RGB image, and then write the array into a video file.

Create a `VideoWriter` object for a new uncompressed AVI file for RGB24 video.

```
v = VideoWriter('newfile.avi','Uncompressed AVI');
```

Open the file for writing.

```
open(v)
```

Create an array containing data from the sample still image, `peppers.png`. Write the image in `A` to the video file.

```
A = imread('peppers.png');
writeVideo(v,A)
```

Close the file.

```
close(v)
```

Create AVI File from Animation

Write a sequence of frames to a compressed AVI file by generating a sequence of frames, creating a video object for the file to write to, and then writing the frames to the video file.

Setup the axes and figure properties to generate frames for the video.

```
Z = peaks;
surf(Z);
axis tight manual
set(gca,'nextplot','replacechildren');
```

Create a video writer object for the output video file and open the object for writing.

```
v = VideoWriter('peaks.avi');
open(v);
```

Generate a set of frames, get the frame from the figure, and then write each frame to the file.

```
for k = 1:20
   surf(sin(2*pi*k/20)*Z,Z)
   frame = getframe(gcf);
   writeVideo(v,frame);
end

close(v);
```

See Also

VideoReader | mmfileinfo

Introduced in R2010b

close

Close file after writing video data

Syntax

```
close(v)
```

Description

`close(v)` closes the video file associated with `v`. You must call `close` to ensure that all data is written to the file. The object, `v`, remains in the workspace. If you call the `open` function after closing the video file, `open` discards all existing contents of the file.

Examples

Open and Close Video File for Writing

Create a `VideoWriter` object to write a new Motion JPEG AVI file.

```
v = VideoWriter('newfile');
```

Open the file for writing.

```
open(v)
```

Write a 300-by-300 matrix of data to the file.

```
writeVideo(v,rand(300))
```

Close the file.

```
close(v)
```

Input Arguments

v — Input `VideoWriter` object
`VideoWriter` object

Input `VideoWriter` object. Use `VideoWriter` to create the object.

See Also

`VideoWriter` | `open` | `writeVideo`

Introduced in R2010b

VideoWriter.getProfiles

Profiles and file formats that `VideoWriter` supports

Syntax

```
profiles = VideoWriter.getProfiles()
```

Description

`profiles = VideoWriter.getProfiles()` returns an array of `audiovideo.writer.ProfileInfo` objects that indicate the types of files that `VideoWriter` can create.

Examples

Get Profile Information

View available profiles and get specific information about the `'Uncompressed AVI'` profile.

View profiles that `VideoWriter` supports.

```
profiles = VideoWriter.getProfiles()
```

```
  Summary of installed VideoWriter profiles:

         Name                                    Description
    ---------------- ---------------------------------------------------------------
    Archival         Video file compression with JPEG 2000 codec with lossless mode ena
    Grayscale AVI    An AVI file with Grayscale Video Data
    Indexed AVI      An AVI file with Indexed Video Data
    MPEG-4           A MPEG-4 file with H.264 Compression
    Motion JPEG 2000 Video file compression with JPEG 2000 codec.
    Motion JPEG AVI  An AVI file with Motion JPEG compression
    Uncompressed AVI An AVI file with uncompressed RGB24 video data
```

Find the index number for the `'Uncompressed AVI'` profile.

```
uncompAVI = find(ismember({profiles.Name},'Uncompressed AVI'));
```

View the properties of the `'Uncompressed AVI'` profile.

```
profiles(uncompAVI)
```

```
  audiovideo.writer.ProfileInfo

    ProfileInfo Properties:

       Name:                      'Uncompressed AVI'
       Description:               'An AVI file with uncompressed RGB24 video data'
       FileExtensions:            {'.avi'}
       ColorChannels:             3
       FrameRate:                 30
       VideoBitsPerPixel:         24
       VideoCompressionMethod:    'None'
       VideoFormat:               'RGB24'
```

Get the file extensions associated with the profile.

```
ext = profiles(uncompAVI).FileExtensions
```

```
ext = 1x1 cell array
    {'.avi'}
```

Output Arguments

`profiles` — Supported profiles and video file formats
array of `audiovideo.writer.ProfileInfo` objects

Supported profiles and video file formats, returned as an array of `audiovideo.writer.ProfileInfo` objects, which have the following read-only properties.

Property	Description
`Name`	Character vector indicating the profile name, such as `'Uncompressed AVI'`.

Property	Description
`Description`	Character vector indicating the description of the profile.
`FileExtensions`	Cell array of character vectors containing file extensions supported by the file format.
`ColorChannels`	Number of color channels in each output video frame.
`CompressionRatio`	Number greater than 1 that specifies the target ratio between the number of bytes in the input image and the number of bytes in the compressed image. Only applies to objects associated with Motion JPEG 2000 files. Default: `10`.
`FrameRate`	Rate of playback for the video in frames per second. Default: `30`.
`LosslessCompression`	Boolean value (logical `true` or `false`) that specifies whether to use reversible mode, so that the decompressed data is identical to the input data. When `true`, `VideoWriter` ignores values for `CompressionRatio`. Only applies to objects associated with Motion JPEG 2000 files.
`MJ2BitDepth`	Number of least-significant bits in the input image data, from 1 to 16. Applied only to objects associated with Motion JPEG 2000 files.
`Quality`	Number from 0 through 100. Higher values correspond to higher quality video and larger files. Only applies to objects associated with the `MPEG-4` or `Motion JPEG AVI` profile. Default: `75`.
`VideoBitsPerPixel`	Number of bits per pixel in each output video frame.

Property	Description
`VideoCompressionMethod`	Character vector indicating the type of video compression, such as `'None'` or `'Motion JPEG'`.
`VideoFormat`	Character vector indicating the MATLAB representation of the video format, such as `'RGB24'`.

See Also

`VideoWriter`

Introduced in R2010b

open

Open file for writing video data

Syntax

```
open(v)
```

Description

`open(v)` opens the file associated with `v` for writing. When you open the file, all properties of the object become read only. `open` discards all existing contents of the file.

Examples

Open and Close Video File for Writing

Create a `VideoWriter` object to write a new Motion JPEG AVI file.

```
v = VideoWriter('newfile');
```

Open the file for writing.

```
open(v)
```

Write a 300-by-300 matrix of data to the file.

```
writeVideo(v,rand(300))
```

Close the file.

```
close(v)
```

Input Arguments

v — Input `VideoWriter` object
`VideoWriter` object

Input `VideoWriter` object. Use `VideoWriter` to create the object.

See Also

`VideoWriter` | `close` | `writeVideo`

Introduced in R2010b

writeVideo

Write video data to file

Syntax

```
writeVideo(v,img)
writeVideo(v,frame)
```

Description

`writeVideo(v,img)` writes data from an array to the video file associated with `v`.

You must call `open(v)` before calling `writeVideo`.

`writeVideo(v,frame)` writes one or more movie frames typically returned by the `getframe` function.

Examples

Write Image to Motion JPEG 2000 File

Write an RGB image to a Motion JPEG 2000 file with lossless compression.

Create an array containing data from the sample still image, `peppers.png`.

```
A = imread('peppers.png');
```

Create a `VideoWriter` object for a new video file. Use the `'Archival'` profile to specify a Motion JPEG 2000 file with lossless compression.

```
v = VideoWriter('myFile','Archival');
```

Verify the type of video compression for the new file.

```
v.VideoCompressionMethod
```

```
ans =
'Motion JPEG 2000'
```

Open the video file for writing. Then, write the image data in A to the file.

```
open(v)
writeVideo(v,A)
```

Close the video file.

```
close(v)
```

Write Indexed Image to AVI File

Read image and colormap data from the sample indexed image file, `corn.tif`.

```
[X,map] = imread('corn.tif');
```

Create a `VideoWriter` object for a new indexed AVI file.

```
v = VideoWriter('myIndexed.avi','Indexed AVI');
```

Assign the colormap data to the `Colormap` property of `v`.

```
v.Colormap = map;
```

Open the file for writing. After you open the file, you cannot change the properties of `v`.

```
open(v)
```

Write the image data in X to the video file. Then, close the file.

```
writeVideo(v,X)
close(v)
```

Convert MPEG-4 to AVI File

Convert the example file, `xylophone.mp4`, to an uncompressed AVI file.

Create objects to read and write the video, and open the AVI file for writing.

```
reader = VideoReader('xylophone.mp4');
writer = VideoWriter('transcoded_xylophone.avi', ...
                        'Uncompressed AVI');

writer.FrameRate = reader.FrameRate;
open(writer);
```

Read and write each frame.

```
while hasFrame(reader)
   img = readFrame(reader);
   writeVideo(writer,img);
end

close(writer);
```

Create AVI File from Animation

Write a sequence of frames to a compressed AVI file by generating a sequence of frames, creating a video object for the file to write to, and then writing the frames to the video file.

Setup the axes and figure properties to generate frames for the video.

```
Z = peaks;
surf(Z);
axis tight manual
set(gca,'nextplot','replacechildren');
```

Create a video writer object for the output video file and open the object for writing.

```
v = VideoWriter('peaks.avi');
open(v);
```

Generate a set of frames, get the frame from the figure, and then write each frame to the file.

```
for k = 1:20
   surf(sin(2*pi*k/20)*Z,Z)
   frame = getframe(gcf);
   writeVideo(v,frame);
end
```

```
close(v);
```

Input Arguments

v — Input VideoWriter object
VideoWriter object

Input VideoWriter object. Use VideoWriter to create the object.

img — Values representing grayscale or RGB color images
2-D array | 3-D array | 4-D array

Values representing grayscale or RGB color images, specified as a 2-D, 3-D, or 4-D array:

- For a single grayscale, monochrome, or indexed image, img must be two dimensional: height-by-width
- For a single truecolor (RGB) image, img is three dimensional: height-by-width-by-3.
- For a sequence of grayscale images, img is four dimensional:. height-by-width-by-1-by-frames. The height and width must be consistent for all frames within a file.
- For a sequence of RGB images, img is four dimensional: height-by-width-by-3-by-frames. The height and width must be consistent for all frames within a file.

When creating AVI or MPEG-4 files:

- img is an array of single, double, or uint8 values representing one or more grayscale or RGB color images, which writeVideo writes as one or more RGB video frames.
- Data of type single or double must be in the range [0,1], except when writing indexed AVI files.

When creating Motion JPEG 2000 files:

- img is an array of uint8, int8, uint16, or int16 values representing one or more monochrome or RGB color images.

Data Types: single | double | int8 | int16 | uint8 | uint16

frame — Frame data
1-by-1 structure array | 1-by-F array of structures

Frame data, specified as a 1-by-1 structure array representing a single frame, or a 1-by-`F` array of structures representing multiple frames. Each frame contains two fields: `cdata` and `colormap`. The `frame` array is typically returned by the `getframe` function.

If `colormap` is not empty, then each element of `cdata` should be a 2-D (height-by-width) array. The height and width must be consistent for all frames within a file.

`colormap` can contain a maximum of 256 entries. Each element of `colormap` must be in the range `[0,1]`.

When you create a `VideoWriter` object. the `profile` input and the size of `cdata` determine how `writeVideo` uses `frame`.

`profile` of `VideoWriter` object	**Size of each element of `cdata`**	**Behavior of `writeVideo`**
`'Indexed AVI'`	2-D (height-by-width)	Use frame as provided.
`'Grayscale AVI'`	2-D (height-by-width)	Use frame as provided. `colormap` should be empty.
All other profiles	2-D (height-by-width)	Construct RGB image frames using the `colormap` field
	3-D (height-by-width-by-3)	Ignore the `colormap` field. Construct RGB image frames using the `cdata` field

Data Types: `struct`

See Also

`VideoWriter` | `close` | `getframe` | `open`

Topics

"Image Types"

Introduced in R2010b

view

Camera line of sight

Syntax

```
view(az,el)
view(v)
view(dim)
view(ax, ___ )
[caz,cel] = view( ___ )
```

Description

`view(az,el)` sets the azimuth and elevation angles of the camera's line of sight on page 1-16772 for the current axes.

`view(v)` sets the line of sight according to `v`, which is a two- or three-element array:

- Two-element array — The values are the azimuth and elevation angles respectively.
- Three-element array — The values are the *x*-, *y*-, and *z*-coordinates of a vector that starts at the center of the plot box and points toward the camera. MATLAB calculates the azimuth and elevation angles using a unit vector pointing in the same direction.

`view(dim)` uses the default line of sight for 2-D or 3-D plots. Specify `dim` as `2` for the default 2-D view or `3` for the default 3-D view.

`view(ax, ___ )` specifies the target axes when changing the line of sight.

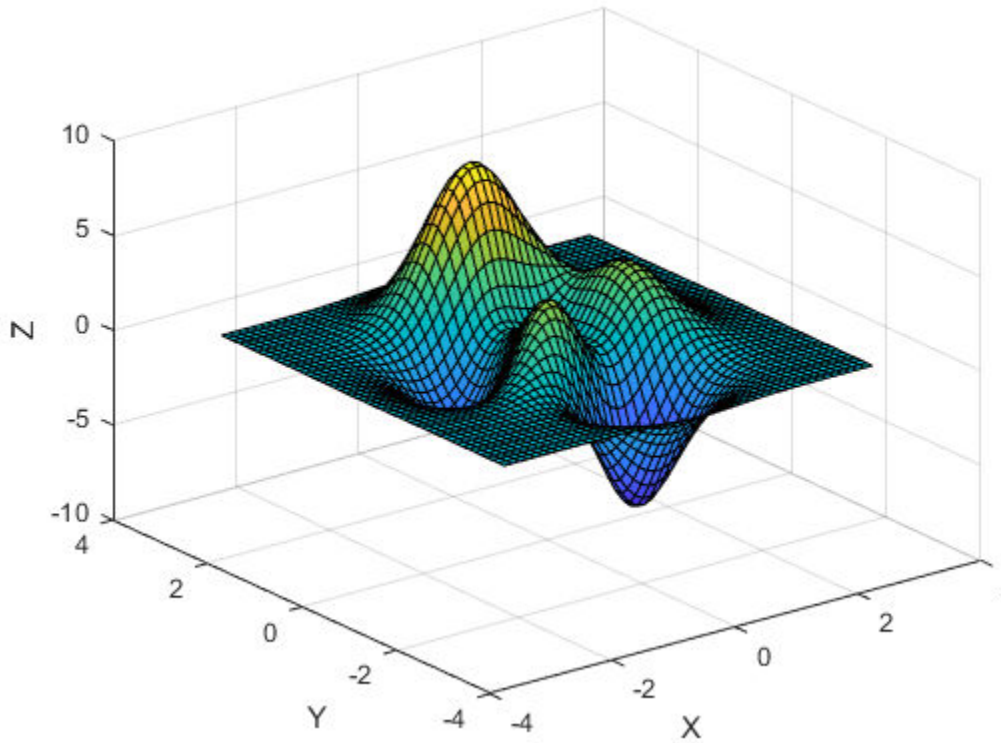
`[caz,cel] = view( ___ )` returns the azimuth and elevation angles as `caz` and `cel`, respectively. Specify input arguments from any of the previous syntaxes to get the angles for the new line of sight. Or, specify no input arguments to get the angles for the current line of sight.

Examples

Change View of Surface Plot

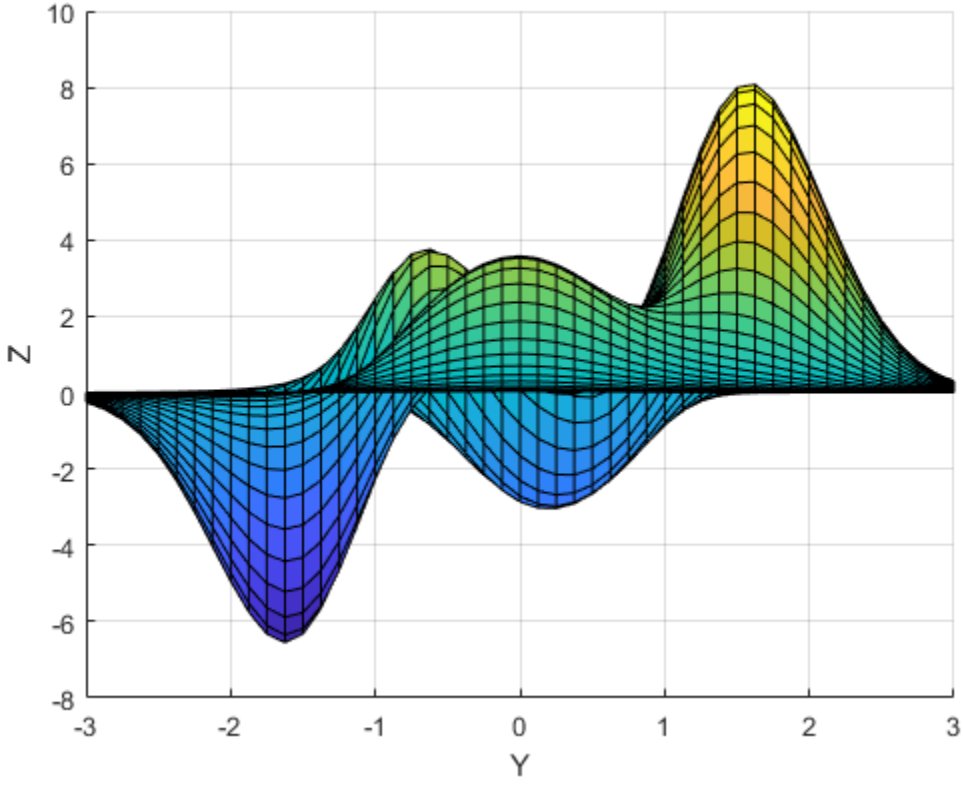
Use the `peaks` function to get the *x*-, *y*-, and *z*-coordinates of a surface. Then plot the surface and label each axis.

```
[X,Y,Z] = peaks;
surf(X,Y,Z)
xlabel('X')
ylabel('Y')
zlabel('Z')
```



View the plot using an azimuth of `90` degrees and an elevation of `0` degrees. The new line of sight is along the *x*-axis.

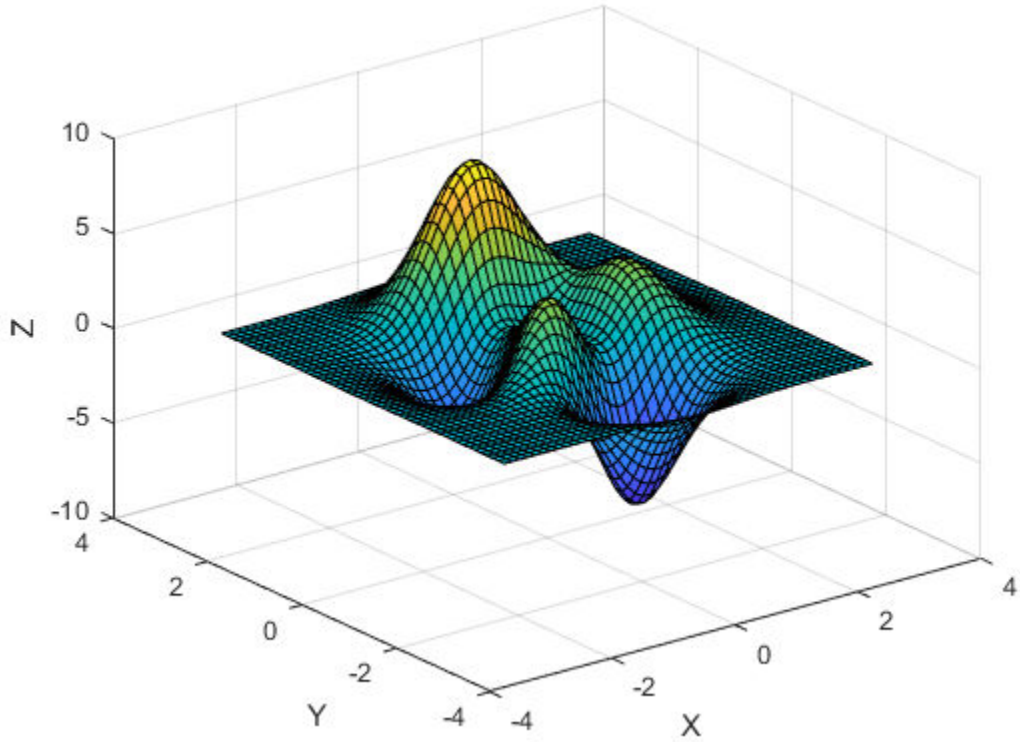
```
view(90,0)
```



Display Surface in a 2-D View

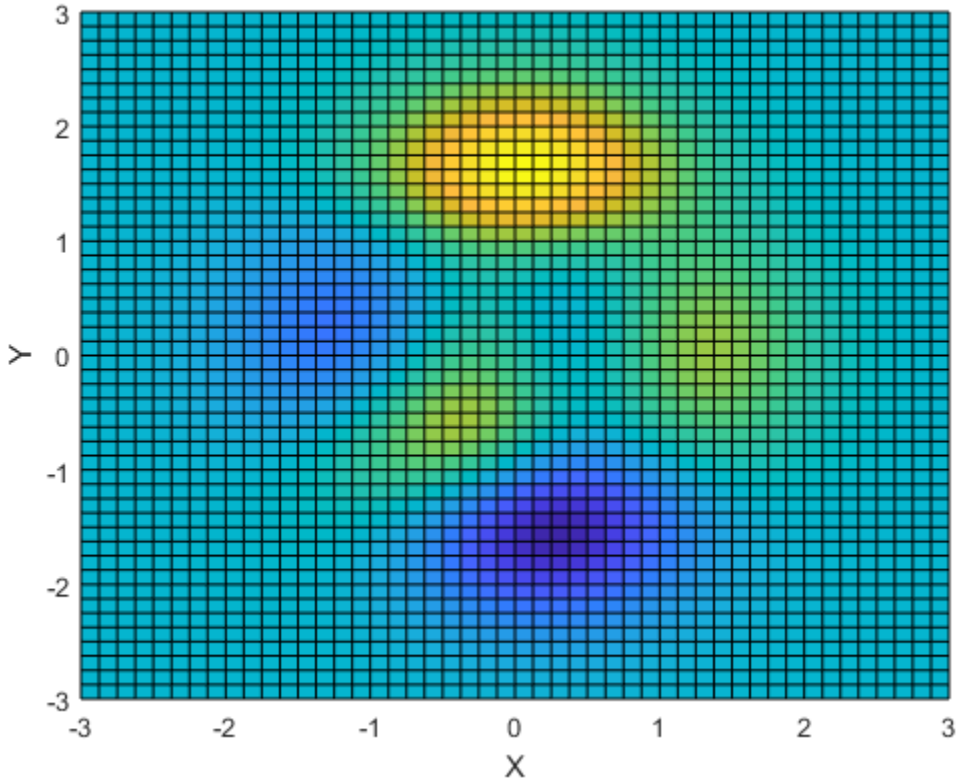
Use the `peaks` function to get the *x*-, *y*-, and *z*-coordinates of a surface. Then plot the surface and label each axis.

```
[X,Y,Z] = peaks;
surf(X,Y,Z)
xlabel('X')
ylabel('Y')
zlabel('Z')
```



Display the plot in a 2-D view.

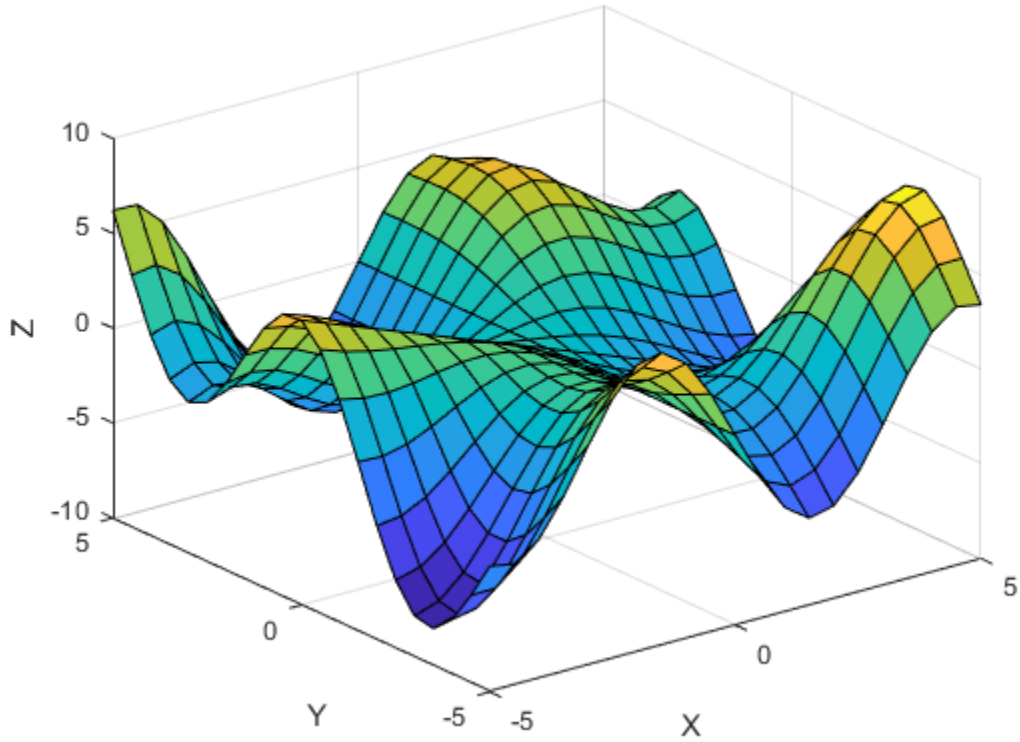
```
view(2)
```



Change the View Using a Vector

Create a set of *x*-, *y*-, and *z*-coordinates and use them to plot a surface. Then label each axis.

```
[X,Y] = meshgrid(-5:.5:5);
Z = Y.*sin(X) - X.*cos(Y);
surf(X,Y,Z)
xlabel('X')
ylabel('Y')
zlabel('Z')
```



Get the azimuth and elevation angles for this plot.

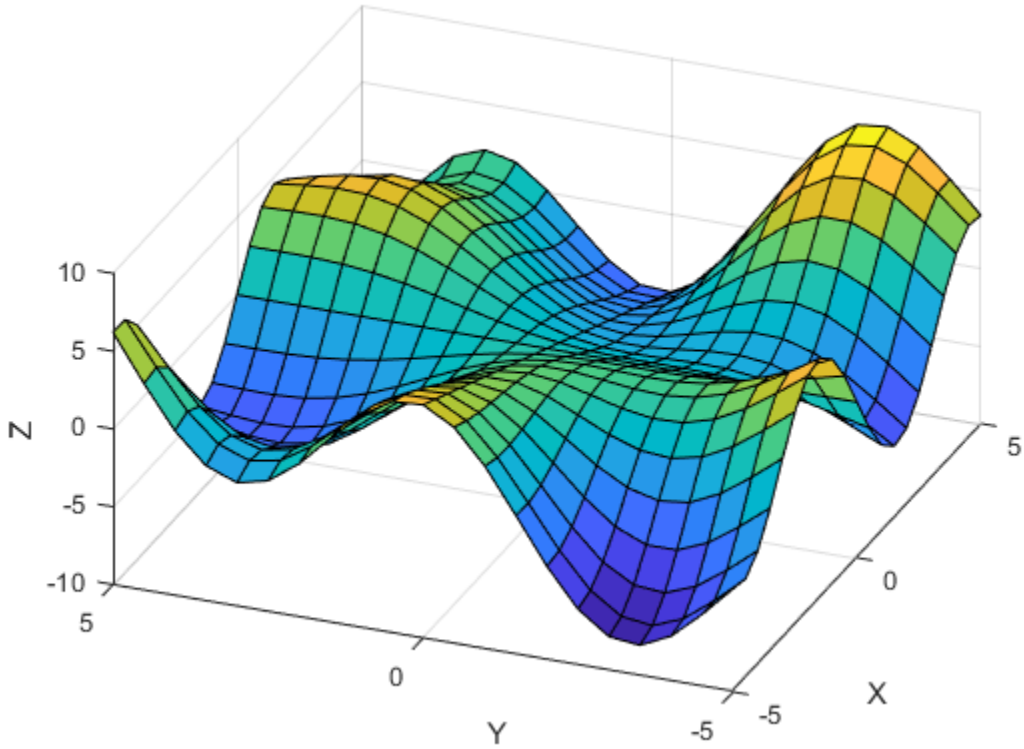
```
[caz,cel] = view
```

```
caz = -37.5000
```

```
cel = 30
```

Change the view by specifying `v` as the *x*- *y*- and *z*-coordinates of a vector, and return the new azimuth and elevation angles. The new angles are based on a unit vector pointing in the same direction as `v`.

```
v = [-5 -2 5];
[caz,cel] = view(v)
```

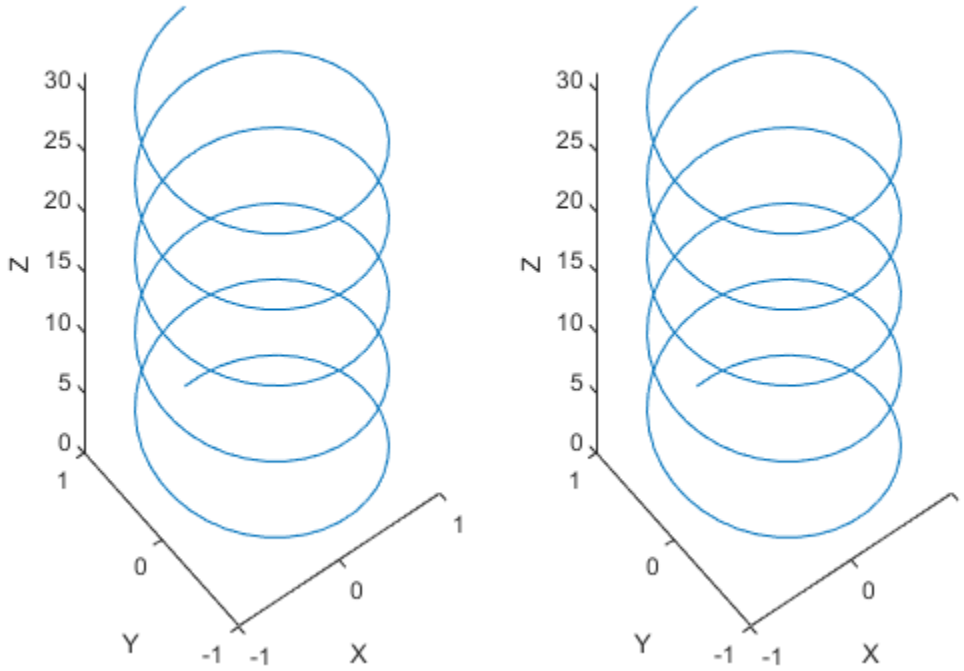



```
caz = -68.1986

cel = 42.8760
```

Change View of Specific Axes

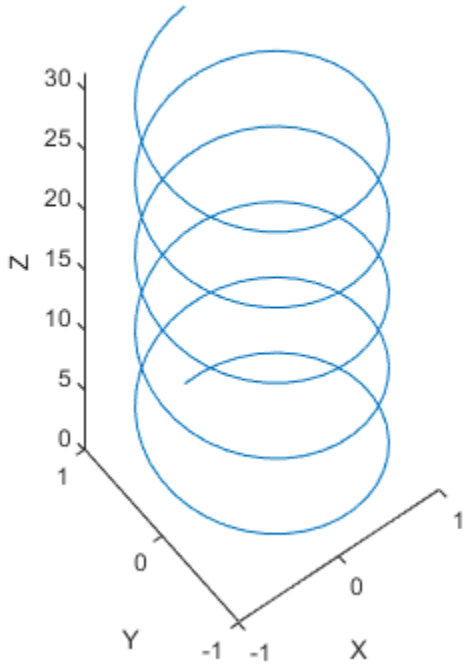
Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 1-by-2 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Use them to create separate but identical line plots.

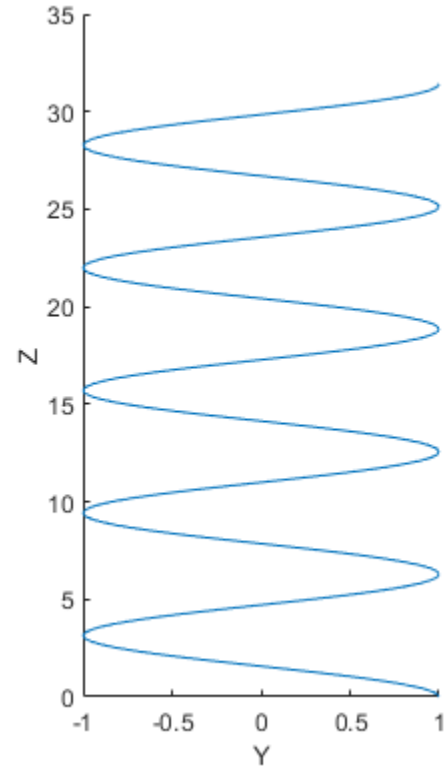
```
t = 0:pi/20:10*pi;
xt1 = sin(t);
yt1 = cos(t);
tiledlayout(1,2)

% Left plot
ax1 = nexttile;
plot3(ax1,xt1,yt1,t)
xlabel('X')
ylabel('Y')
zlabel('Z')

% Right plot
ax2 = nexttile;
plot3(ax2,xt1,yt1,t)
xlabel('X')
ylabel('Y')
zlabel('Z')
```



Change the view of the right plot to a side view along the *x*-axis.

```
view(ax2,[90 0]);
```





Input Arguments

az — Azimuth
number

Azimuth, specified as an angle in degrees from the negative *y*-axis. Increasing this angle corresponds to counterclockwise rotation about the *z*-axis when viewing the *x*-*y* plane from above.

The default value depends on whether your chart is in a 2-D or 3-D view. For 2-D charts, the default value is `0`. For 3-D charts, the default value is `-37.5`.

Example: `view(45,25)` sets the azimuth to 45 degrees and the elevation to 25 degrees.

`el` — Elevation
number

Elevation, specified the minimum angle in degrees between the line of sight and the *x*-*y* plane. Increasing the elevation from `-90` to `90` degrees corresponds to a rotation from the negative *z*-axis to the positive *z*-axis.

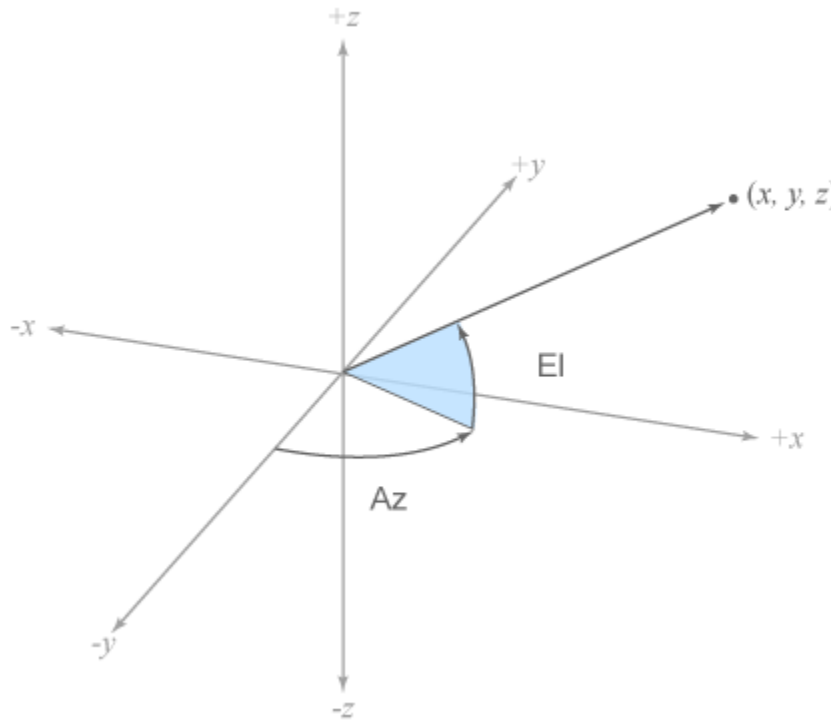
The default value depends on whether your chart is in a 2-D or 3-D view. For 2-D charts, the default value is `90`. For 3-D charts, the default value is `30`.

Example: `view(45,25)` sets the azimuth to 45 degrees and the elevation to 25 degrees.

`v` — Line-of-sight vector
`[az el]` | `[x y z]`

Line-of-sight vector, specified as one of the following:

- A two-element array containing the azimuth and elevation angles respectively.
- A three-element array containing the *x*-, *y*-, and *z*-coordinates of a vector that originates from the center of the plot box and points toward the camera. MATLAB calculates the azimuth and elevation from a unit vector pointing in the same direction. Thus, the magnitude of the `v` has no effect on the line of sight.



Example: `view([45 25])` sets the azimuth to 45 degrees and the elevation to 25 degrees.

Example: `view([20 25 5])` sets the line of sight to a vector that points in the same direction as the vector `[20 25 5]`.

`dim` — Dimensions
2 | 3

Dimensions, specified as 2 or 3. This argument sets the azimuth and elevation to the default values that MATLAB uses for a 2-D or 3-D plot.

Value of `dim`	Azimuth	Elevation
2	0	90
3	-37.5	30

ax — Target axes
`Axes` object

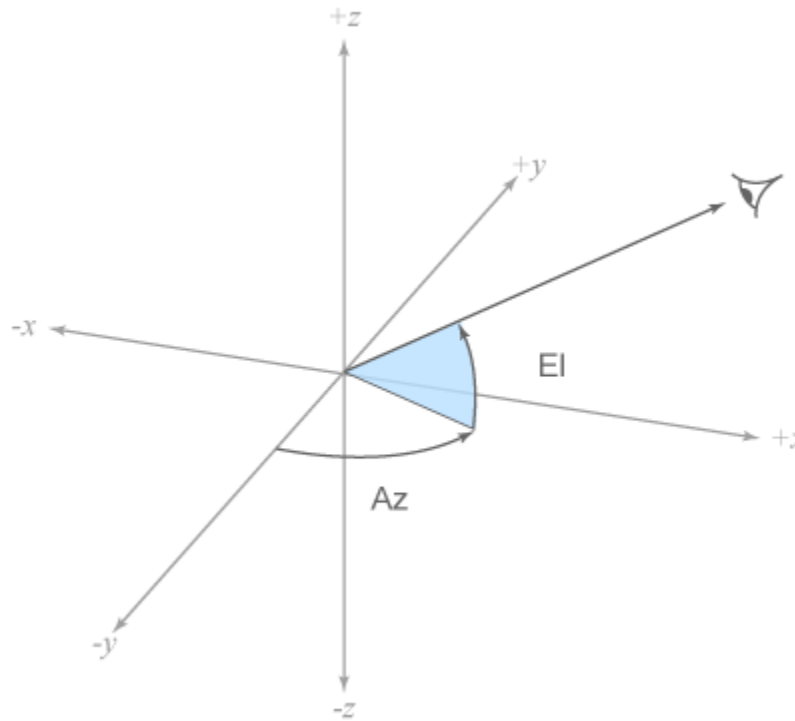
Target axes, specified as an `Axes` object.

More About

Line of Sight

The line of sight starts at the center of the plot box and points toward the camera. MATLAB defines this line using two angles, the azimuth and the elevation. These angles are measured within a 3-D coordinate system that has its origin at the center of the plot box.

- Azimuth — The angle of rotation around the *z*-axis, as measured from the negative *y*-axis. Increasing this angle corresponds to counterclockwise rotation about the *z*-axis when viewing the *x*-*y* plane from above.
- Elevation — The minimum angle between the line of sight and the *x*-*y* plane. Increasing the elevation from `-90` to `90` degrees corresponds to a rotation from the negative *z*-axis to the positive *z*-axis.



Tips

The `view` function controls the direction of the line of sight, but it does not control the location that the line starts from, nor the camera position. For more control, use the camera functions such as `campos`, `camtarget`, `camroll`, and `cameratoolbar`.

See Also

`rotate3d`

Topics

"View Overview"
"Low-Level Camera Properties"

“View Control with the Camera Toolbar”

Introduced before R2006a

viewmtx

View transformation matrices

Syntax

```
viewmtx
T = viewmtx(az,el)
T = viewmtx(az,el,phi)
T = viewmtx(az,el,phi,xc)
```

Description

`viewmtx` computes a 4-by-4 orthographic or perspective transformation matrix that projects four-dimensional homogeneous vectors onto a two-dimensional view surface (e.g., your computer screen).

`T = viewmtx(az,el)` returns an *orthographic* transformation matrix corresponding to azimuth `az` and elevation `el`. `az` is the azimuth (i.e., horizontal rotation) of the viewpoint in degrees. `el` is the elevation of the viewpoint in degrees.

`T = viewmtx(az,el,phi)` returns a *perspective* transformation matrix. `phi` is the perspective viewing angle in degrees. `phi` is the subtended view angle of the normalized plot cube (in degrees) and controls the amount of perspective distortion.

Phi	Description
0 degrees	Orthographic projection
10 degrees	Similar to telephoto lens
25 degrees	Similar to normal lens
60 degrees	Similar to wide-angle lens

`T = viewmtx(az,el,phi,xc)` returns the perspective transformation matrix using `xc` as the target point within the normalized plot cube (i.e., the camera is looking at the point `xc`). `xc` is the target point that is the center of the view. You specify the point as a three-element vector, `xc = [xc,yc,zc]`, in the interval [0,1]. The default value is `xc = [0,0,0]`.

A four-dimensional homogenous vector is formed by appending a 1 to the corresponding three-dimensional vector. For example, `[x,y,z,1]` is the four-dimensional vector corresponding to the three-dimensional point `[x,y,z]`.

Examples

Compute Transformation Matrices

Determine the projected two-dimensional vector corresponding to the three-dimensional point (0.5,0.0,-3.0) using the default view direction. Note that the point is a column vector.

```
A = viewmtx(-37.5,30);
x4d = [.5 0 -3 1]';
x2d = A*x4d;
x2d = x2d(1:2)
```

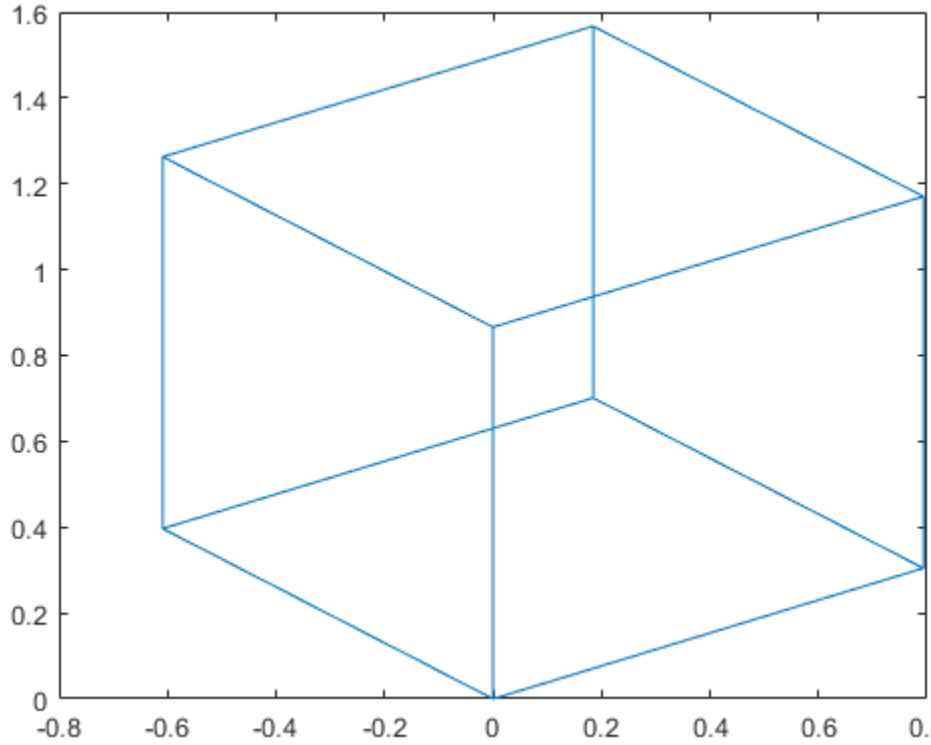
```
x2d = 2×1

    0.3967
   -2.4459
```

Create vectors that trace the edges of a unit cube.

```
x = [0  1  1  0  0  0  1  1  0  0  1  1  1  1  0  0];
y = [0  0  1  1  0  0  0  1  1  0  0  0  1  1  1  1];
z = [0  0  0  0  0  1  1  1  1  1  1  0  0  1  1  0];
```

Transform the points in these vectors to the screen, then plot the object.

```
A = viewmtx(-37.5,30);
[m,n] = size(x);
x4d = [x(:),y(:),z(:),ones(m*n,1)]';
x2d = A*x4d;
x2 = zeros(m,n); y2 = zeros(m,n);
x2(:) = x2d(1,:);
y2(:) = x2d(2,:);
plot(x2,y2)
```



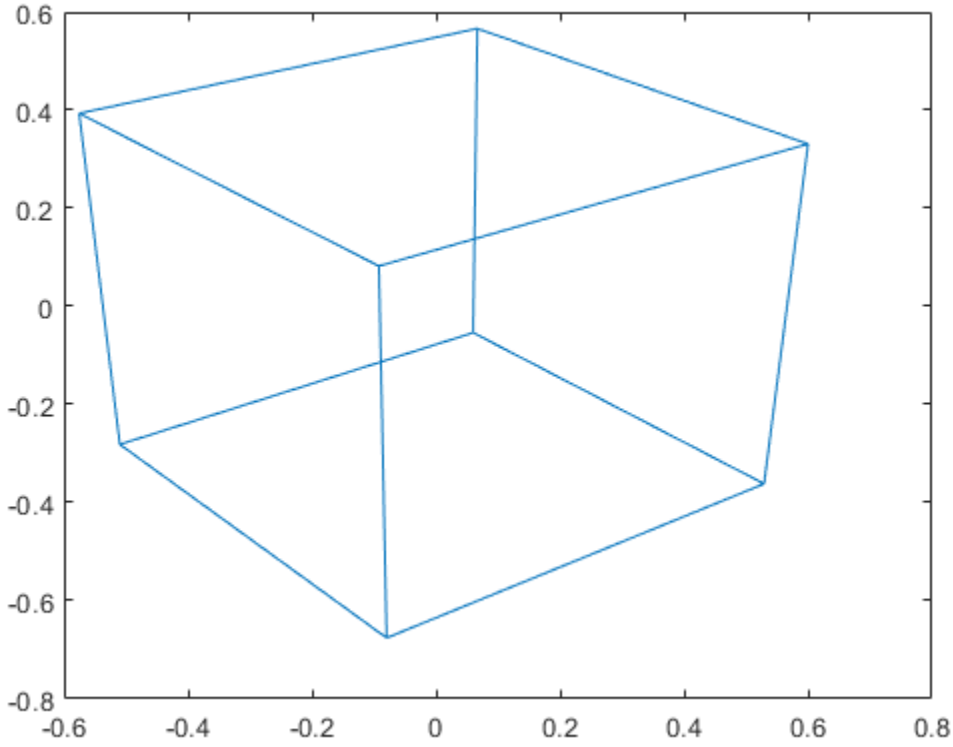
Use a perspective transformation with a 25 degree viewing angle.

```
A = viewmtx(-37.5,30,25);
x4d = [.5 0 -3 1]';
x2d = A*x4d;
x2d = x2d(1:2)/x2d(4)
```

```
x2d = 2×1

    0.1777
   -1.8858
```

Transform the cube vectors to the screen and plot the object.

```
A = viewmtx(-37.5,30,25);
[m,n] = size(x);
x4d = [x(:),y(:),z(:),ones(m*n,1)]';
x2d = A*x4d;
x2 = zeros(m,n); y2 = zeros(m,n);
x2(:) = x2d(1,:)./x2d(4,:);
y2(:) = x2d(2,:)./x2d(4,:);
plot(x2,y2)
```



See Also

hgtransform | view

Topics

"Camera Graphics Terminology"

Introduced before R2006a

visdiff

Compare two files or folders

Syntax

```
visdiff(name1,name2)
visdiff(name1,name2,type)
comparison=visdiff( ___ )
```

Description

`visdiff(name1,name2)` displays the differences between the two specified files or folders in the Comparison Tool. The File Comparison Tool supports many file types such as MATLABcode files, zip files, MAT-files, and various other MATLAB and Simulink file formats.

Use `visdiff` to compare two Simulink models and identify changes or merge differences between the models. For more information, see “Model Comparison” (Simulink).

`visdiff(name1,name2,type)` compares the two files or folders using the specified comparison type. If you do not specify a type, `visdiff` uses the default comparison type for your selected files. The `type` option does not apply when comparing folders.

`comparison=visdiff( ___ )` compares two model files and returns a comparison object containing the differences between the two files. This syntax does not open the Comparison Tool and is only supported for comparing Simulink models. `name1` and `name2` must be model files. Use the comparison object to manipulate the comparison at the command line, for example by applying filters and publishing comparison reports.

Examples

Compare Two Files

Compare two files using both relative and full paths.

Compare the two files `lengthofline.m` and `lengthofline2.m` in the current folder.

```
visdiff('lengthofline.m','lengthofline2.m')
```

Compare the two demo files `gatlin.mat` and `gatlin2.mat` using a fully qualified file name.

```
visdiff(fullfile(matlabroot,'toolbox','matlab','demos','gatlin.mat'), ...
fullfile(matlabroot,'toolbox','matlab','demos','gatlin2.mat'))
```

Compare Two Files and Specify Type

Compare the two files `lengthofline.m` and `lengthofline2.m` as binary.

If you do not specify type, `visdiff` compares the two files using the default text comparison type. By changing to the binary comparison type you can examine differences such as end-of-line characters.

```
visdiff('lengthofline.m','lengthofline2.m','binary')
```

Publish Comparison Report

When comparing Simulink models, you can manipulate the comparison at the command line by specifying an output argument to `visdiff`.

Compare two model files and get the comparison object output from `visdiff`.

```
comparison = visdiff(modelname1, modelname2);
```

(Optional) Filter the comparison result. Supported filters are: `'unfiltered'` – removes all filtering from the comparison, or `'default'` – default filtering strategy for comparisons.

```
filter(comparison, 'unfiltered');
```

To publish a comparison report to a file, use `publish` on the comparison object. The default format of the published report is HTML. `publish` saves the file in the current folder as `filename1_filename2.html`.

```
file = publish(comparison);
web(file)
```

To specify options for the report, use the following name-value pairs with `publish`.

- `'Format'` – The file format of the published report.

  Options are: `'html'` (default) – single-file HTML Document, or `'docx'` – Microsoft Word document.
- `'Name'` – The name of the report.

  A character vector or scalar string array. If you specify a file extension, then `'Format'` is ignored.
- `'OutputFolder'` – Where to save the report.

  A character vector or scalar string array. The default is the current folder.

Alternatively, you can specify the name-value pairs in an `options` structure: `publish(comparison, options)`.

Input Arguments

`name` — File or folder name
character vector

File or folder name to compare, specified as a character vector. `name` can include a relative path, but the relative path must be in the current folder. Otherwise, `name` must include a full path.

`type` — Comparison type
`'text'` | `'binary'` | `'xml'`

Comparison type, specified as `'text'`, `'binary'`, or `'xml'`. Some comparisons do not support all of the comparison types. If an unsupported comparison type is specified, MATLAB displays an error.

To examine differences such as end-of-line characters in text files, specify a `'binary'` comparison type.

The default comparison type for XML files depends on the XML type, and is either `'text'` or `'xml'`. Specify `'text'` or `'xml'` to override the default comparison type for the file.

Output Arguments

`comparison` — Comparison object
object

Comparison object for manipulating the comparison at the command line, for example by applying filters and publishing comparison reports. Only supported for comparing Simulink models.

Tips

In some cases, there might be multiple comparison types available for your selection (e.g., text, binary, file list, or XML comparison). To select a different comparison type, use the `visdiff` function and specify the `type` argument. You also can select from the available comparison types in the Comparison Tool. For more information, see “Compare Files and Folders and Merge Files”.

Alternative Functionality

As an alternative to the `visdiff` function, use the Comparison Tool to compare files or folders interactively. To open the Comparison Tool, go to the **Home** tab, and in the **File** section, click **Compare**. Then select the files or folders to compare.

You can also compare a file that is open in the Editor. On the **Editor** or **Live Editor** tab, in the **File** section, select **Compare**. To compare with the autosave version or the version on disk, select **Compare** ▾, and then select `Save and Compare with Autosave` or `Compare with Version on Disk`. For more information, see “Compare Files with Autosave Version or Version on Disk”.

To compare a file or folder in the Current Folder browser, right-click the selected file or folder and select **Compare Against**. To compare two files or subfolders, right-click the selected files or folders and select **Compare Selected Files/Folders**.

See Also

Topics

"Compare Files and Folders and Merge Files"
"Model Comparison" (Simulink)

Introduced in R2008b

volumebounds

Coordinate and color limits for volume data

Syntax

```
lims = volumebounds(X,Y,Z,V)
lims = volumebounds(X,Y,Z,U,V,W)
lims = volumebounds(V)
lims = volumebounds(U,V,W)
```

Description

`lims = volumebounds(X,Y,Z,V)` returns the x, y, z, and color limits of the current axes for scalar volume data. `lims` is returned as a vector:

```
[xmin xmax ymin ymax zmin zmax cmin cmax]
```

You can pass this vector to the `axis` command.

`lims = volumebounds(X,Y,Z,U,V,W)` returns the x, y, and z limits of the current axes for vector volume data. `lims` is returned as a vector:

```
[xmin xmax ymin ymax zmin zmax]
```

`lims = volumebounds(V)` and `lims = volumebounds(U,V,W)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

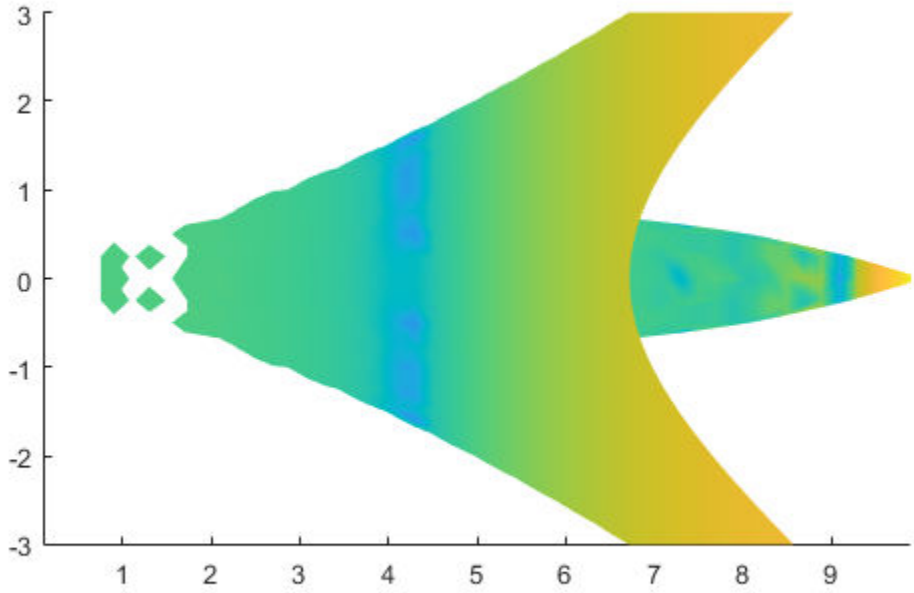
where `[m n p] = size(V)`.

Examples

Set Axes and Color Limits for Isosurface

Use `volumebounds` to set the axes and color limits for an isosurface returned by the `flow` function.

```
[x,y,z,v] = flow;
p = patch(isosurface(x,y,z,v,-3));
isonormals(x,y,z,v,p);
daspect([1 1 1]);
isocolors(x,y,z,fliplr(v),p);
shading interp
axis(volumebounds(x,y,z,v));
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`isosurface` | `streamslice`

Introduced before R2006a

voronoi

Voronoi diagram

Syntax

```
voronoi(x,y)
voronoi(x,y,T)
voronoi(TO)
[vx,vy] = voronoi( ___ )
h = voronoi( ___ )
```

Description

`voronoi(x,y)` plots the bounded cells of the Voronoi diagram on page 1-16790 for the 2-D points in vectors `x` and `y`.

`voronoi(x,y,T)` uses the Delaunay triangulation `T` to plot the Voronoi diagram.

`voronoi(TO)` uses the `delaunayTriangulation` object `TO` to plot the Voronoi diagram.

`[vx,vy] = voronoi( ___ )` returns the 2-D vertices of the Voronoi edges.

`h = voronoi( ___ )` returns a graphics array of two line object handles representing the points and edges of the diagram.

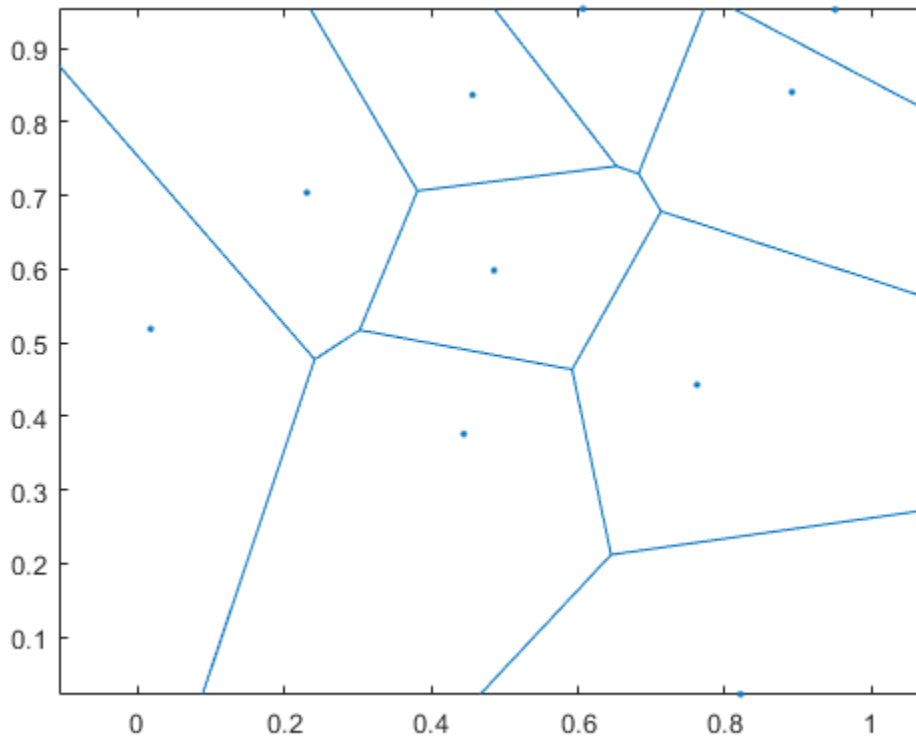
Examples

Voronoi Diagram with 2-D Points

Create two vectors containing the coordinates of 10 2-D points, and plot the Voronoi diagram.

```
x = gallery('uniformdata',[1 10],0);
y = gallery('uniformdata',[1 10],1);
```

```
voronoi(x,y)
axis equal
```



Input Arguments

x — *x*-coordinates
column vector

x-coordinates, specified as a column vector.

y — *y*-coordinates
column vector

y-coordinates, specified as a column vector.

T — Delaunay triangulation
matrix

Delaunay triangulation, specified as a 3-column matrix. Each row of T contains the row indices of the input points that define a triangle in the triangulation.

T0 — Constrained Delaunay triangulation object
`delaunayTriangulation` object

Constrained Delaunay triangulation object, specified as a `delaunayTriangulation` object.

Output Arguments

vx — Edge *x*-coordinates
column vector

x-coordinates of Voronoi edges, returned as a column vector.

vy — Edge *y*-coordinates
column vector

y-coordinates of Voronoi edges, returned as a column vector.

h — Graphics array
array

Graphics array, returned as an array of two line object handles representing the points and edges of the diagram.

More About

Voronoi Diagram

Given a point in a set of coplanar points, you can draw a boundary around it that includes all points closer to it than to any other point in the set. This boundary defines a single

Voronoi polygon. The collection of all Voronoi polygons for every point in the set is called a *Voronoi diagram*.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`delaunay` | `delaunayTriangulation` | `voronoin`

Topics

“Voronoi Diagrams”

Introduced before R2006a

voronoiDiagram

Class: `DelaunayTri`

(Not recommended) Voronoi diagram

Note `voronoiDiagram(DelaunayTri)` is not recommended. Use `voronoiDiagram(delaunayTriangulation)` instead.

`DelaunayTri` is not recommended. Use `delaunayTriangulation` instead.

Syntax

```
[V, R] = voronoiDiagram(DT)
```

Description

`[V, R] = voronoiDiagram(DT)` returns the vertices `V` and regions `R` of the Voronoi diagram of the points `DT.X`. The region `R{i}` is a cell array of indices into `V` that represents the Voronoi vertices bounding the region. The Voronoi region associated with the `i`'th point, `DT.X(i)` is `R{i}`. For 2-D, vertices in `R{i}` are listed in adjacent order, i.e. connecting them will generate a closed polygon (Voronoi diagram). For 3-D the vertices in `R{i}` are listed in ascending order.

The Voronoi regions associated with points that lie on the convex hull of `DT.X` are unbounded. Bounding edges of these regions radiate to infinity. The vertex at infinity is represented by the first vertex in `V`.

Input Arguments

`DT`	Delaunay triangulation.

Output Arguments

V	numv-by-ndim matrix representing the coordinates of the Voronoi vertices, where numv is the number of vertices and ndim is the dimension of the space where the points reside.
R	Vector cell array of length(DR.X), representing the Voronoi cell associated with each point.

Examples

Compute the Voronoi Diagram of a set of points:

```
X = [ 0.5    0
      0      0.5
     -0.5   -0.5
     -0.2   -0.1
     -0.1    0.1
      0.1   -0.1
      0.1    0.1 ]
dt = DelaunayTri(X)
[V,R] = voronoiDiagram(dt)
```

More About

Voronoi Diagram

The Voronoi diagram of a discrete set of points X decomposes the space around each point X(i) into a region of influence R{i}. Locations within the region are closer to point i than any other point. The region of influence is called the Voronoi region. The collection of all the Voronoi regions is the Voronoi diagram.

Convex Hull

The convex hull of a set of points X is the smallest convex polygon (or polyhedron in higher dimensions) containing all of the points of X.

See Also

`delaunayTriangulation` | `triangulation` | `voronoi` | `voronoin`

voronoin

N-D Voronoi diagram

Syntax

```
[v,c] = voronoin(P)
[v,c] = voronoin(P,opts)
```

Description

`[v,c] = voronoin(P)` returns the Voronoi vertices `v` and the Voronoi cells `c` of the Voronoi diagram on page 1-16798 for the N-D points in a matrix `P`.

`[v,c] = voronoin(P,opts)` also specifies the Qhull options used to compute the Voronoi diagram.

Examples

2-D Voronoi Diagram

Create a matrix of 2-D points and compute the Voronoi vertices and diagram cells.

```
P = [0.5 0; 0 0.5; -0.5 -0.5; -0.2 -0.1; -0.1 0.1; 0.1 -0.1; 0.1 0.1]
```

```
P = 7×2

    0.5000         0
         0    0.5000
   -0.5000   -0.5000
   -0.2000   -0.1000
   -0.1000    0.1000
    0.1000   -0.1000
    0.1000    0.1000
```

```
[v,c] = voronoin(P)
```

```
v = 10×2

       Inf       Inf
    0.7000   -1.6500
   -0.0500   -0.0500
   -0.0500   -0.5250
   -1.4500    0.6500
   -1.7500    0.7500
         0    0.2875
    0.3833    0.3833
    0.2875         0
         0         0


c=7×1 cell
    {1x4 double}
    {1x5 double}
    {1x4 double}
    {1x4 double}
    {1x4 double}
    {1x5 double}
    {1x4 double}
```

Input Arguments

P — Points
matrix

Points, specified as a matrix whose columns contain the coordinates for the corresponding dimension. For example, to define a set of 2-D points, place the *x*-coordinates in the first column of P and the corresponding *y*-coordinates in the second column.

opts — Qhull options
cell array of character vectors

Qhull options, specified as a cell array of character vectors indicating which Qhull algorithms to use. For a list of options, see http://www.qhull.org/html/qh-optq.htm.

By default, `opts` is set to `{'Qbb'}` for 2- and 3-dimensional input. For 4-dimensional input and higher, `opts` is set to `{'Qbb','Qx'}`.

Output Arguments

v — Voronoi vertices
matrix

Voronoi vertices, returned as a matrix with the same number of columns as the input. Each row contains the coordinates of an N-D point in the Voronoi diagram, with the first row containing `Inf` values. A row of `Inf` values represents an unbounded cell.

c — Indices
cell array

Indices, returned as a cell array. Each element of `c` contains the row indices of the Voronoi vertices `v` that make up a Voronoi cell.

More About

Voronoi Diagram

Given a point in a set of coplanar points, you can draw a boundary around it that includes all points closer to it than to any other point in the set. This boundary defines a single *Voronoi polygon*. The collection of all Voronoi polygons for every point in the set is called a *Voronoi diagram*.

Tips

- You can plot individual bounded cells of an N-D Voronoi diagram. To do this, use the `convhulln` function to compute the vertices of the facets that make up the Voronoi cell. Then, use `patch` or other plotting functions to generate the figure.

Algorithms

`voronoin` is based on Qhull [1]. For information, see `http://www.qhull.org/`.

References

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, “The Quickhull Algorithm for Convex Hulls,” *ACM Transactions on Mathematical Software*, Vol. 22, No. 4, Dec. 1996, p. 469-483.

See Also

`convhull` | `convhulln` | `delaunay` | `delaunayTriangulation` | `delaunayn` | `voronoi`

Introduced before R2006a

waitbar

Create or update wait bar dialog box

Note If you are using App Designer or creating apps with the `uifigure` function, then use `uiprogressdlg` instead. For more information, see "GUIDE Migration Strategies".

Syntax

```
f = waitbar(x,msg)
f = waitbar(x,msg,Name,Value)
```

```
waitbar(x)
waitbar(x,f)
waitbar(x,f,msg)
```

Description

`f = waitbar(x,msg)` creates a nonmodal on page 1-16809 dialog box containing a wait bar on page 1-16809 with the specified message. The wait bar is fractional length `x`. The dialog box remains open until the code that controls it closes it or the user clicks the close button (X) in the dialog box title bar. The `Figure` object is returned as `f`.

`f = waitbar(x,msg,Name,Value)` specifies additional options using one or more name-value pair arguments that set `figure` properties on the dialog box. For example, `'Name','Progress'` sets the dialog box name to Progress. Specify name-value pair arguments after all other input arguments.

`waitbar(x)` updates the length of the bar in the current wait bar dialog box to `x`. Successive values of `x` normally increase. If they decrease, the wait bar runs in reverse.

`waitbar(x,f)` updates the length of the bar in the wait bar dialog box `f` to the new position `x`.

`waitbar(x,f,msg)` updates the message in the wait bar dialog box `f`.

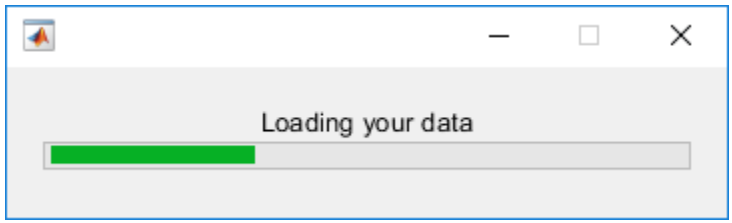
Examples

Create Basic Wait Bar Dialog Box

Create a wait bar dialog box that updates the fractional wait bar length and message at three points in the code.

Use this code to simulate performing a lengthy computation. The `pause` function passes the figure object of the first wait bar dialog box to subsequent `waitbar` function calls. Passing the figure object this way ensures that the dialog box is updated at each point, rather than recreated. When the simulated calculation completes, the dialog box closes.

```
function basicwaitbar
f = waitbar(0,'Please wait...');
pause(.5)

waitbar(.33,f,'Loading your data');
pause(1)

waitbar(.67,f,'Processing your data');
pause(1)

waitbar(1,f,'Finishing');
pause(1)

close(f)
end
```



Update Wait Bar Message and Add Cancel Button

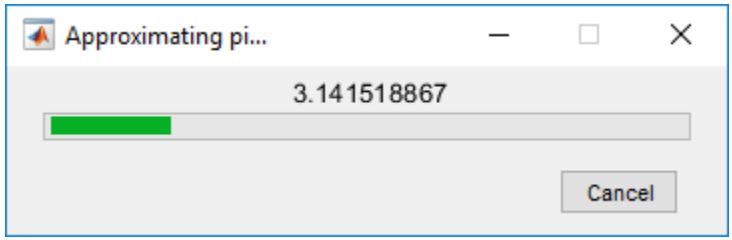
Add a **Cancel** button to enable a user to stop an in-progress computation.

Use the following code to converge on the value of pi in `for` loop iterations, updating the wait bar and message with each iteration. When a user clicks the **Cancel** button MATLAB

sets a logical flag, `'canceling'`, to 1 (true) in the figure application data (`appdata`). The code tests for that value within the for loop and exits the loop if the flag value is `1`. For more information on application data, see “Store Data as Application Data”.

Set the `CreateCancelBtn` callback to a character vector value. This practice not recommended unless the code specified by the value is simple, such as is the case in this callback code. For more information, see “Write Callbacks for Apps Created Programmatically”.

```
function approxpi

f = waitbar(0,'1','Name','Approximating pi...',...
    'CreateCancelBtn','setappdata(gcbf,''canceling'',1)');

setappdata(f,'canceling',0);

% Approximate pi^2/8 as: 1 + 1/9 + 1/25 + 1/49 + ...
pisqover8 = 1;
denom = 3;
valueofpi = sqrt(8 * pisqover8);

steps = 20000;
for step = 1:steps
    % Check for clicked Cancel button
    if getappdata(f,'canceling')
        break
    end

    % Update waitbar and message
    waitbar(step/steps,f,sprintf('%12.9f',valueofpi))

    % Calculate next estimate
    pisqover8 = pisqover8 + 1 / (denom * denom);
    denom = denom + 2;
    valueofpi = sqrt(8 * pisqover8);
end

delete(f)
end
```



Input Arguments

x — Fractional wait bar length

real number

Fractional wait bar length, specified as a real number between `0` and `1`. Successive values of `x` normally increase. If they decrease, then the wait bar runs in reverse.

Example: `.75`

msg — Wait bar message

character vector | cell array of character vectors | string array

Wait bar message, specified as a character vector, cell array of character vectors, or string array. The message displays within the dialog box, above the progress bar.

- If you specify the message as a character vector, then a long message wraps to fit the dialog box.
- If you specify the message as a cell array of character vectors, then line breaks occur between each array element. Long elements wrap to fit the dialog box.

Example: `'Please Wait'`

Example: `{'Please wait', 'The operation is processing'}`

f — Figure object

figure object

Figure object of a previously created wait bar dialog box, specified as the output variable used when that figure object was created.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Name','Wait Bar'`

Note The properties listed here are only a subset of the properties you can set. For a full list, see Figure.

`CreateCancelBtn` — Cancel button callback

function handle | cell array | character vector

Cancel button callback, specified as a function handle, cell array, or character vector (not recommended). MATLAB sets both the **Cancel** button callback and the figure `CloseRequestFcn` callback value to the commands specified by the `callback` argument value. For more information, see “Write Callbacks for Apps Created Programmatically”.

Note Call the `delete` function to remove a wait bar dialog box when you specify a `CreateCancelBtn` callback. When you specify the `CreateCancelBtn` callback, the `close` function does not close the dialog box and makes the **Cancel** and close (X) buttons unresponsive. This situation occurs because the `CreateCancelBtn` callback recursively calls itself. In such a situation, you must forcibly remove the wait bar, for example using code such as this:

```
set(groot,'ShowHiddenHandles','on')
delete(get(groot,'Children'))
```

However, issuing these commands deletes *all* open figures, not just the wait bar dialog box.

Example: `waitbar(.5,'Processing','CreateCancelBtn',@myfun)` specifies the `CreateCancelBtn` callback function as a function handle.

Example: `waitbar(.5,'Processing','CreateCancelBtn',{@myfun,x})` specifies the `CreateCancelBtn` callback function as a cell array. In this case, the function accepts the input argument, `x`.

Name — Name
`''` (default) | character vector | string scalar

Name of the figure, specified as a character vector or a string scalar.

Example: `figure('Name','Results')` sets the name of the figure to `'Results'`.

By default, the name is `'Figure n'`, where *n* is an integer. When you specify the `Name` property, the title of the figure becomes `'Figure n: name'`. If you want only the `Name` value to appear, set `IntegerHandle` or `NumberTitle` to `'off'`.

Position — Location and size of drawable area
`[left bottom width height]`

Location and size of the drawable area, specified as a vector of the form `[left bottom width height]`. This area excludes the figure borders, title bar, menu bar, and tool bars.

This table describes each element in the `Position` vector.

Element	Description
`left`	Distance from the left edge of the primary display to the inner left edge of the window. This value can be negative on systems that have more than one monitor. If the figure is docked, then this value is relative to the Figure panel within the MATLAB desktop.
`bottom`	Distance from the bottom edge of the primary display to the inner bottom edge of the window. This value can be negative on systems that have more than one monitor. If the figure is docked, then this value is relative to the Figure panel within the MATLAB desktop.
`width`	Distance between the right and left inner edges of the waitbar.
`height`	Distance between the top and bottom inner edges of the window.

All measurements are in units specified by the `Units` property.

You cannot specify the figure `Position` property when the figure is docked.

In MATLAB Online, the `bottom` and `left` elements of the `Position` vector are ignored.

To place the full window, including the borders, title bar, menu bar, tool bars, use the `OuterPosition` property.

Note The Windows operating system enforces a minimum window width and a maximum window size. If you specify a figure size outside of those limits, the displayed figure will conform to the limits instead of the size you specified.

`Units` — Units of measurement

`'pixels'` (default) | `'normalized'` | `'inches'` | `'centimeters'` | `'points'` | `'characters'`

Units of measurement, specified as one of the values from this table.

Units Value	Description
`'pixels'` (default)	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux systems, the size of a pixel is determined by your system resolution.
`'normalized'`	These units are normalized with respect to the parent container. The lower-left corner of the container maps to `(0,0)` and the upper-right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'points'`	Points. One point equals 1/72nd of an inch.

Units Value	Description
`'characters'`	These units are based on the default uicontrol font of the graphics root object: • Character width = width of the letter `x`. • Character height = distance between the baselines of two lines of text. To access the default uicontrol font, use `get(groot,'defaultuicontrolFontName')` or `set(groot,'defaultuicontrolFontName')`.

MATLAB measures all units from the lower left corner of the parent object.

This property affects the `Position` property. If you change the `Units` property, consider returning its value to the default value after completing your computation to avoid affecting other functions that assume the default value.

The order in which you specify the `Units` and `Position` properties has these effects:

- If you specify the `Units` before the `Position` property, then MATLAB sets `Position` using the units you specify.
- If you specify the `Units` property after the `Position` property, MATLAB sets the position using the default `Units`. Then, MATLAB converts the `Position` value to the equivalent value in the units you specify.

`WindowStyle` — Window style
`'normal'` (default) | `'modal'` | `'docked'`

Window style, specified as one of the following:

- `'normal'` — The figure window is independent of other windows, and the other windows are accessible while the figure is displaying.
- `'modal'` — The figure displays on top of all existing figure windows, making them inaccessible as long as the top figure exists and remains modal. However, any new figures created after a modal figure will display.

  When multiple modal windows exist, the most recently created window keeps focus and stays above all other windows until it becomes invisible, or is returned to a normal

window style, or is deleted. At that time, focus reverts to the window that last had focus.

- `'docked'` — The figure displays in the desktop or a document window. When the `WindowStyle` property is set to `'docked'`, you cannot set the `DockControls` property to `'off'`. The `'docked'` option is not supported in MATLAB Online.

Note These are some important characteristics of the `WindowStyle` property and some recommended best practices:

- When you create UI windows, always specify the `WindowStyle` property. If you also want to set the `Resize`, `Position`, or `OuterPosition` properties of the figure, then set the `WindowStyle` property first.
- You can change the `WindowStyle` property of a figure at any time, including when the figure is visible and contains children. However on some systems, setting this property might cause the figure to flash or disappear and reappear, depending on the system's implementation of normal and modal windows. For best visual results, set the `WindowStyle` property at creation time or when the figure is invisible.
- Calling `reset` on a figure does not change the value of the `WindowStyle` property.

Modal Window Style Behavior

When `WindowStyle` is set to `'modal'`, the figure window traps all keyboard and mouse actions over all MATLAB windows as long as the windows are visible. Windows belonging to applications other than MATLAB are unaffected.

Typing **Ctrl+C** when a modal figure has focus causes that figure to revert to a `'normal'` `WindowStyle` property setting. This allows the user to type at the command line.

Figures with the `WindowStyle` property set to `'modal'` and the `Visible` property set to `'off'` do not behave modally until MATLAB makes them visible. Therefore, you can hide a modal window for later reuse, instead of destroying it.

Modal figures do not display menu children, built-in menus, or toolbars. But, it is not an error to create menus in a modal figure or to change the `WindowStyle` property setting to `'modal'` on a figure with menu children. The `Menu` objects exist and the figure retains them. If you reset the figure's `WindowStyle` property to `'normal'`, the menus display.

More About

Modal Dialog Box

A modal dialog box prevents a user from interacting with other MATLAB windows before responding to the modal dialog box.

Nonmodal Dialog Box

A nonmodal dialog box enables a user to interact with other MATLAB windows before responding to the nonmodal dialog box. A nonmodal dialog box is also referred to as `normal`.

Wait Bar

An indicator that displays what percentage of a calculation is complete as the calculation proceeds by progressively filling a bar with color from left to right. A wait bar is also referred to as a progress bar.

See Also

`close` | `delete` | `getappdata` | `setappdata`

Introduced before R2006a

waitfor

Block execution and wait for condition

Syntax

```
waitfor(obj)
waitfor(obj,propname)
waitfor(obj,propname,propvalue)
```

Description

`waitfor(obj)` blocks statements from executing until the specified object closes (is deleted). When the object no longer exists, `waitfor` returns, enabling execution to resume. If the object does not exist, `waitfor` returns immediately.

`waitfor(obj,propname)` specifies a property name of the object and blocks execution until the value of the property changes or the object closes. For example, `waitfor(mytable,'Data')` pauses execution until the value of `'Data'` changes for `mytable`. If the specified property name is invalid, execution remains blocked.

`waitfor(obj,propname,propvalue)` specifies a value that the property must change to before execution can resume. If the specified property is already equal to `propvalue`, then `waitfor` returns immediately and execution resumes.

Examples

Wait for Warning Dialog to Close

Create a warning dialog and wait for it to close. Commands after `waitfor` do not execute until you close the dialog.

```
mydlg = warndlg('This is a warning.', 'A Warning Dialog');
waitfor(mydlg);
disp('This prints after you close the warning dialog.');
```

Wait for Property Value Change

Wait for the user to select a check box before adding data to a table. The `Value` property of the check box is `0` when not selected, and `1` when selected.

```
t = uitable;
c = uicontrol('Style','checkbox','String','Add data');
c.Position = [320 100 80 20];
waitfor(c,'Value');
t.Data = magic(5);
```

Wait for Property to Change to Specific Value

Change the background color of a text field when the user stops editing it and clicks elsewhere in the figure. When the text field loses focus, the `Editing` property changes from `'on'` to `'off'`.

```
txt = text(.5,.5,'Edit text and click');
txt.Editing = 'on';
txt.BackgroundColor = [1 1 1];

waitfor(txt,'Editing','off');
txt.BackgroundColor = [1 1 0];
```

Input Arguments

`obj` — Object
object

Object, such as an `Axes`, `Text`, `Panel`, `ButtonGroup`, `Table`, or `UIControl` object. The object can be the child of a `Figure` object created with the `figure` or `uifigure` function, or it can be the child of a container in a `Figure` object.

`propname` — Property name
character vector | string scalar

Property name, specified as a character vector or string scalar. Use this argument to specify a property of `obj` whose value must change before execution resumes.

`propvalue` — Property value
valid property value associated with `propname`

Property value, specified as a valid property value associated with `propname`. Use this argument to indicate a specific value that the property must change to before execution resumes.

Tips

- If you close the figure while `waitfor` is executing, an error occurs because the code attempts to access objects that no longer exist. You can handle the error by enclosing `waitfor` in a `try/catch` block.

Algorithms

Typically, callbacks can still run if `waitfor` has been used to prevent programs or Simulink models from continuing execution. For example, callbacks that respond to user actions (like pressing a mouse button) can still run even if `waitfor` has been called.

`waitfor` can also be used to block nested function calls. For example, a callback that executes while the `waitfor` function is running can also call `waitfor`.

If a callback function of a UI component is currently executing the `waitfor` function, then that callback can be interrupted regardless of what the `Interruptible` property value for that component has been set to.

See Also

`drawnow` | `keyboard` | `pause` | `uiresume` | `uiwait` | `waitforbuttonpress`

Introduced before R2006a

waitforbuttonpress

Wait for click or key press

Note

Use this function only with GUIDE, or with apps created with the `figure` function.

Syntax

```
w = waitforbuttonpress
```

Description

`w = waitforbuttonpress` blocks statements from executing until the user has clicked a mouse button or pressed a key in the current figure.

The return argument, `w`, can have the following values:

- `0` if it detects a click
- `1` if it detects a key press

The `waitforbuttonpress` function does not return a value when any of the following keys are pressed by themselves or together: **Ctrl**, **Shift**, **Alt**, **Caps Lock**, **Num Lock**, or **Scroll Lock**.

Examples

Create Axes After Button Press

Create a figure and call the `waitforbuttonpress` function. Then, create axes and note that they do not appear.

```
figure;
w = waitforbuttonpress;
axes;
```

Click the figure. Now, `waitforbuttonpress` returns, execution continues, and the axes appear.

Tips

- To determine the last key pressed, the mouse selection type, or the location of the mouse pointer within the current figure, query the `Figure` properties `CurrentCharacter`, `SelectionType`, and `CurrentPoint`, respectively.

Algorithms

Some important points to consider when using `waitforbuttonpress`:

- If a `WindowButtonDownFcn` is defined for the figure, it executes before `waitforbuttonpress` returns a value.
- The `waitforbuttonpress` function errors if the user closes the figure by clicking the window close button unless your code calls the `waitforbuttonpress` function within a `try/catch` block.

Compatibility Considerations

waitforbuttonpress is Figure-specific

Behavior changed in R2014b

To resume program execution after the `waitforbuttonpress` function is called, a click or key press must occur in the current figure, and not just any figure. This change in behavior was introduced in order to prevent program execution from proceeding in an unintended figure. Prior to R2014b, `waitforbuttonpress` would detect a click or key press event in any open figure. The execution stream would then resume in the figure in which the event was detected.

See Also

`dragrect` | `figure` | `ginput` | `rbbox` | `waitfor`

Introduced before R2006a

warndlg

Create warning dialog box

Note If you are using App Designer or creating apps with the `uifigure` function, then use `uialert` instead. For more information, see “GUIDE Migration Strategies”.

Syntax

```
f = warndlg(msg)
f = warndlg(msg,title)
f = warndlg(msg,title,opts)
f = warndlg
```

Description

`f = warndlg(msg)` creates a nonmodal on page 1-16822 warning dialog box with the specified message and returns the dialog box figure object `f`. The message text wraps to fit the dialog box. The dialog box title is `Warning Dialog`.

`f = warndlg(msg,title)` specifies a custom dialog box title.

`f = warndlg(msg,title,opts)` specifies the window style when `opts` is set to `'non-modal'`, `'modal'`, or `'replace'`. It specifies the window style and an interpreter for the `msg` argument when `opts` is a structure array.

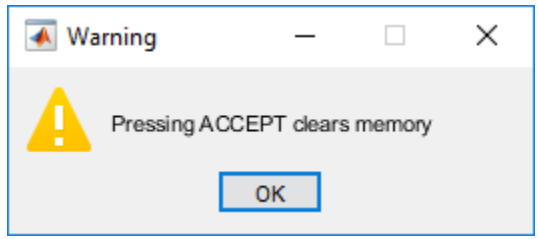
`f = warndlg` creates a warning dialog box with a default title and message, as follows.

- Default title: `Warning Dialog`
- Default message: `This is the default warning.`

Examples

Specify Message and Title for Warning Dialog Box

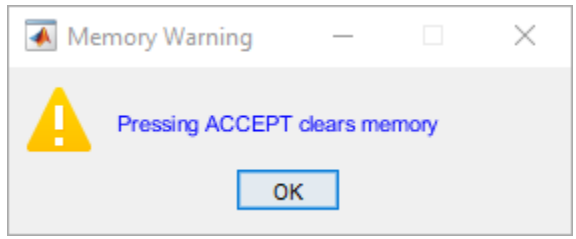
```
f = warndlg('Pressing ACCEPT clears memory','Warning');
```



Interpret Message for Modal Warning Dialog Box

Create a structure to specify a modal on page 1-16822 window style and the TeX interpreter. Then, create a warning dialog box specifying the `opts` structure as an input argument. The TeX interpreter reads the `\color{blue}` characters in the message text as an instruction to render the message text in blue.

```
opts = struct('WindowStyle','modal',...
              'Interpreter','tex');
f = warndlg('\color{blue} Pressing ACCEPT clears memory',...
             'Memory Warning', opts);
```



Input Arguments

`msg` — Warning message

`'This is the default warning'` (default) | character vector | cell array of character vectors | string array

Warning message, specified as a character vector, cell array of character vectors, or a string array.

- If you specify the message as a character vector, then MATLAB wraps the text to fit the dialog box.
- If you specify the message a cell array, then MATLAB wraps the text after each cell array element. MATLAB wraps the text of long cell array elements to fit the dialog box.

Example: `'This command clears your workspace.'`

`title` — Dialog box title
`'Warning Dialog'` (default) | character vector | string scalar

Dialog box title, specified as a character vector or string scalar.

Example: `'Workspace Warning'`

`opts` — Dialog box settings
'non-modal' | 'modal' | 'replace' | structure array

Dialog box settings specified as a window style or a structure. The structure specifies the window style and an interpreter for the `msg` argument.

To specify the window style only, set `opts` to one of the values in this table.

Value	Description
`'non-modal'`	Create a warning dialog box that is nonmodal on page 1-16822. This dialog box has no effect on other open dialog boxes.

Value	Description
`'modal'`	Specify a warning dialog box that is modal on page 1-16822. If other warning dialog boxes have the same title, then MATLAB modifies the most recently active one with the current specifications. MATLAB deletes all other open error, message, and warning dialog boxes with the same dialog box title as the most recently active warning dialog box. The affected dialog boxes can be modal or nonmodal. Message and error dialog boxes are created with the `msgbox`, and `errordlg` functions, respectively.
`'replace'`	Specify a warning dialog box that is nonmodal. If other warning dialog boxes have the same title, then MATLAB modifies the most recently active one with the current specifications. MATLAB deletes all other open error, message, or warning dialog boxes with the same title as the most recently active warning dialog box. The affected dialog boxes can be modal or nonmodal. Message and error dialog boxes are created with the `msgbox`, and `errordlg` functions, respectively.

To specify the window style and an interpreter for the warning dialog box message, create an `opts` structure with the fields `WindowStyle` and `Interpreter`. This table shows valid values for the fields. The structure must include both fields.

Field	Values
`WindowStyle`	`'non-modal'`,`'modal'`, or `'replace'`.
`Interpreter`	`'none'` or `'tex'`. If set to `'tex'`, hen MATLAB renders the message using the TeX interpreter. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the message text.

Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they modify only the next character or the characters within the curly braces. When you set the `Interpreter` property to `'tex'`, the supported modifiers are as follows.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{`*`specifier`*`}`	Font name — Replace *`specifier`* with the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{`*`specifier`*`}`	Font size —Replace *`specifier`* with a numeric scalar value in point units.	`'\fontsize{15} text'`
`\color{`*`specifier`*`}`	Font color — Replace *`specifier`* with one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Replace *`specifier`* with a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\alpha	α	\upsilon	υ	\sim	∼
\angle	∠	\phi	ϕ	\leq	≤
\ast	*	\chi	χ	\infty	∞
\beta	β	\psi	ψ	\clubsuit	♣
\gamma	γ	\omega	ω	\diamondsuit	♦
\delta	δ	\Gamma	Γ	\heartsuit	♥
\epsilon	ϵ	\Delta	Δ	\spadesuit	♠
\zeta	ζ	\Theta	Θ	\leftrightarrow	↔
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	µ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

Example: `opts = 'modal'`

Example: `opts.WindowStyle = 'non-modal'; opts.Interpreter = 'tex';`

More About

Modal Dialog Box

A modal dialog box prevents a user from interacting with other MATLAB windows before responding to the modal dialog box.

Nonmodal Dialog Box

A nonmodal dialog box enables a user to interact with other MATLAB windows before responding to the nonmodal dialog box. A nonmodal dialog box is also referred to as `normal`.

Tips

- MATLAB program execution continues even when a modal Warning dialog box is active. To block program execution until the user closes the dialog box, use the `uiwait` function.

See Also

`dialog` | `errordlg` | `helpdlg` | `msgbox`

Introduced before R2006a

warning

Display warning message

Syntax

```
warning(msg)
warning(msg,A1,...,An)
warning(msgID, ___ )

warning(state)
warning(state,msgID)
warning

warnStruct = warning
warning(warnStruct)

warning(state,mode)
warnStruct = warning(state,mode)
```

Description

`warning(msg)` displays the warning message and sets the warning state for the `lastwarn` function. If `msg` is empty, `warning` resets the warning state for `lastwarn`, but does not display any text.

`warning(msg,A1,...,An)` displays a message that contains formatting conversion characters, such as those used with the MATLAB `sprintf` function. Each conversion character in `msg` is converted to one of the values `A1,...,An`.

`warning(msgID, ___ )` attaches a warning identifier to the warning message. You can include any of the input arguments in the previous syntaxes. The identifier enables you to distinguish warnings and to control what happens when MATLAB encounters the warnings.

`warning(state)` enables, disables, or displays the state of all warnings.

`warning(state,msgID)` acts on the state of a specified warning.

`warning` displays the state of all of the warnings. It is equivalent to `warning('query')`.

`warnStruct = warning` returns a structure or array of structures that contains information about which warnings are enabled and disabled. `warnStruct` includes an `identifier` field with a `msgID` or `'all'`, and a `state` field that indicates the state of the corresponding warning.

`warning(warnStruct)` sets the current warning settings as indicated in the structure array, `warnStruct`.

`warning(state,mode)` controls whether MATLAB displays the stack trace or additional information about the warning.

`warnStruct = warning(state,mode)` returns a structure with an `identifier` field containing the `mode` and a `state` field containing the current state of `mode`. If you pass the output structure, `warnStruct`, into the `warning` function, you set the state of the `mode`, not which warnings are enabled or disabled.

Examples

Issue Warning Message

Generate a warning that displays a message.

```
n = 7;
if ~ischar(n)
   warning('Input must be a character vector')
end
```

```
Warning: Input must be a character vector
```

Include information about `n` in the warning message.

```
if ~ischar(n)
   warning('Input must be a character vector, not a %s',class(n))
end
```

```
Warning: Input must be a character vector, not a double
```

Attach a message identifier to the warning message.

```
if ~ischar(n)
   warning('MyComponent:incorrectType',...
       'Input must be a character vector, not a %s',class(n))
end
```

```
Warning: Input must be a character vector, not a double
```

Set and Query Warning State

Disable all warnings.

```
warning('off')
```

Query the warnings.

```
warning
```

```
All warnings have the state 'off'.
```

Enable all warnings, disable the singular matrix warning, and query all warnings.

```
warning('on')
warning('off','MATLAB:singularMatrix')
warning
```

```
The default warning state is 'on'. Warnings not set to the default are

State  Warning Identifier

    off  MATLAB:singularMatrix
```

Re-enable the singular matrix warning.

```
warning('on','MATLAB:singularMatrix')
```

Save and Restore Warning Settings

Enable all warnings, and then disable the singular matrix warning.

```
warning('on')
warning('off','MATLAB:singularMatrix')
```

Save the current warning settings.

```
s = warning
```

```
s =

2x1 struct array with fields:

    identifier
    state
```

Examine the two structures.

```
s(1)
```

```
ans =

    identifier: 'all'
         state: 'on'
```

```
s(2)
```

```
ans =

    identifier: 'MATLAB:singularMatrix'
         state: 'off'
```

All warnings are enabled except for `'MATLAB:singularMatrix'`.

Disable and query all warnings.

```
warning('off')
warning('query')
```

```
All warnings have the state 'off'.
```

Restore the saved warning state structure, and query the state.

```
warning(s)
warning('query')
```

```
The default warning state is 'on'. Warnings not set to the default are

State  Warning Identifier
```

```
    off  MATLAB:singularMatrix
```

Control Warning Verbosity

Ensure `verbose` and `backtrace` settings are the default values.

```
warning('off','verbose')
warning('on','backtrace')
```

Turn on all warnings, and remove a folder that does not exist on the MATLAB path.

```
warning('on')
rmpath('nosuchfolder')
```

```
Warning: "nosuchfolder" not found in path.
> In rmpath at 57
```

Enable verbosity to display an extended warning message.

```
warning('on','verbose')
rmpath('nosuchfolder')
```

```
Warning: "nosuchfolder" not found in path.
(Type "warning off MATLAB:rmpath:DirNotFound" to suppress this warning.)

> In rmpath at 57
```

Disable display of the stack trace.

```
warning('off','backtrace')
rmpath('nosuchfolder')
```

```
Warning: "nosuchfolder" not found in path.
(Type "warning off MATLAB:rmpath:DirNotFound" to suppress this warning.)
```

Temporarily Disable Warning

Compute a singular matrix.

```
A = eye(2);
B = [3 6; 4 8];
C = B\A;
```

```
Warning: Matrix is singular to working precision.
```

Find the warning ID, save the current warning state, and disable the specific warning

```
[msgStr,msgId] = lastwarn;
warnStruct = warning('off',msgId);
C = B\A;
```

Restore previous warning state.

```
warning(warnStruct);
C = B\A;
```

```
Warning: Matrix is singular to working precision.
```

Input Arguments

`msg` — Information about cause of warning

character vector | string scalar

Information about the cause of the warning and how you might correct it, specified as a character vector or string scalar. To format the message, use escape sequences, such as `\t` or `\n`. You also can use any format specifiers supported by the `sprintf` function, such as `%s` or `%d`. Specify values for the conversion specifiers via the `A1,...,An` input arguments. For more information, see “Formatting Text”.

Note You must specify more than one input argument with `warning` if you want MATLAB to convert special characters (such as `\t`, `\n`, `%s`, and `%d`) in the warning message.

Example: `'Input must be a character vector.'`

`A1,...,An` — Values

character vector | string scalar | numeric scalar

Values that replace the conversion specifiers in `msg`, specified as a character vector, string scalar, or numeric scalar.

`msgID` — Identifier for warning
character vector | string scalar | `'all'` | `'last'`

Identifier for the warning, specified as a character vector, string scalar, `'all'`, or `'last'`. Use the warning identifier to help identify the source of the warning or to control a selected subset of the warnings in your program.

The message identifier includes a `component` and `mnemonic`. The identifier must always contain a colon and follows this simple format: `component:mnemonic`. The `component` and `mnemonic` fields must each begin with a letter. The remaining characters can be alphanumerics (A–Z, a–z, 0–9) and underscores. No whitespace characters can appear anywhere in `msgID`. For more information, see “Message Identifiers”.

When you set the `state` of a warning, the `msgID` can have a value of `'all'` or `'last'`. Use `'all'` to set the state of all warnings, and use `'last'` to set the state of the last issued warning.

Example: `'MATLAB:singularMatrix'`

Example: `'MATLAB:narginchk:notEnoughInputs'`

`state` — Warning control indicator
`'on'` | `'off'` | `'query'`

Warning control indicator specified as `'on'`, `'off'`, or `'query'`. Use `'on'` or `'off'` to control whether MATLAB issues a warning. Use `'query'` to query the current state of the warning.

`warnStruct` — Warning settings
structure | array of structures

Warning settings, specified as a structure or array of structures that contains information about which warnings are enabled and which are disabled. `warnStruct` includes an `identifier` field with a `msgID` or `'all'`, and `state` field indicating the state of the corresponding warning.

`mode` — Verbosity and stack trace display settings
`'backtrace'` | `'verbose'`

Verbosity and the stack trace display of settings, specified by `'backtrace'` or `'verbose'`. By default, the state of verbosity is set to `'off'` and the state of stack trace display is set to `'on'`.

See Also

`dbstop` | `disp` | `error` | `errordlg` | `lasterror` | `lastwarn` | `sprintf` | `warndlg`

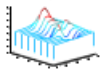
Topics

"Issue Warnings and Errors"
"Suppress Warnings"
"Restore Warnings"
"Change How Warnings Display"

Introduced before R2006a

waterfall

Waterfall plot



Syntax

```
waterfall(Z)
waterfall(X,Y,Z)
waterfall(...,C)
waterfall(ax,...)
h = waterfall(...)
```

Description

The `waterfall` function draws a mesh similar to the `meshz` function, but it does not generate lines from the columns of the matrices. This produces a “waterfall” effect.

`waterfall(Z)` creates a waterfall plot using `x = 1:size(Z,2)` and `y = 1:size(Z,1)`. `Z` determines the color, so color is proportional to surface height.

`waterfall(X,Y,Z)` creates a waterfall plot using the values specified in `X`, `Y`, and `Z`. `Z` also determines the color, so color is proportional to the surface height. If `X` and `Y` are vectors, `X` corresponds to the columns of `Z`, and `Y` corresponds to the rows, where `length(x) = n`, `length(y) = m`, and `[m,n] = size(Z)`. `X` and `Y` are vectors or matrices that define the *x*- and *y*-coordinates of the plot. `Z` is a matrix that defines the *z*-coordinates of the plot (i.e., height above a plane). If `C` is omitted, color is proportional to `Z`.

`waterfall(...,C)` uses scaled color values to obtain colors from the current colormap. Color scaling is determined by the range of `C`, which must be the same size as `Z`. MATLAB performs a linear transformation on `C` to obtain colors from the current colormap.

`waterfall(ax,...)` plots into the axes `ax` instead of the current axes (`gca`).

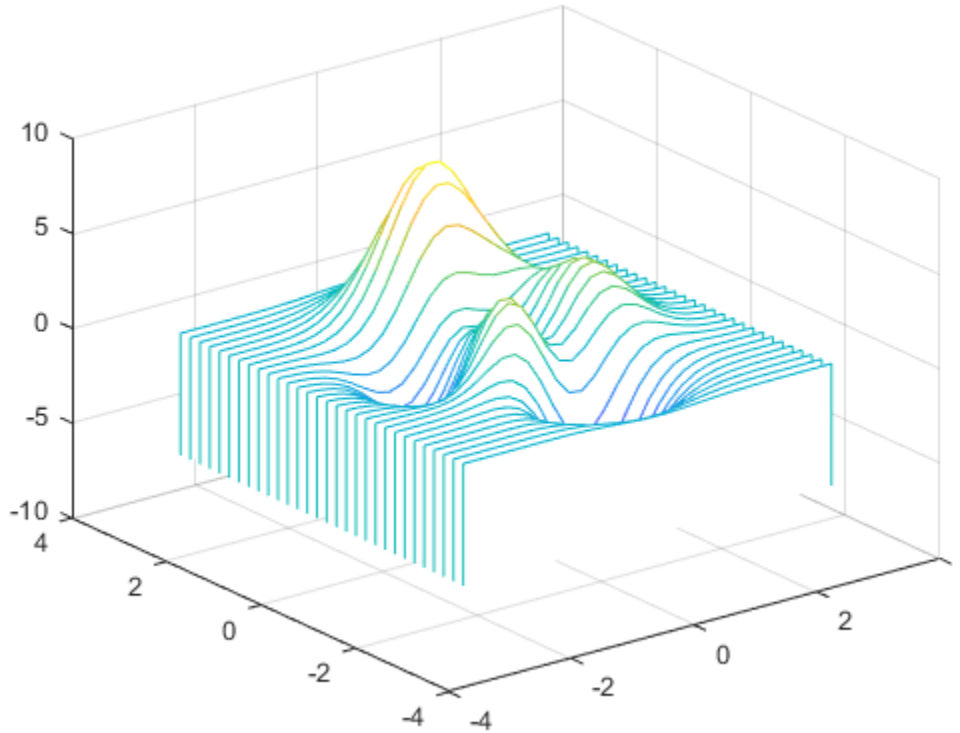
`h = waterfall(...)` returns the patch graphics object used to draw the plot.

Examples

Create Waterfall Plot

Create a waterfall plot of the peaks function.

```
figure
[X,Y,Z] = peaks(30);
waterfall(X,Y,Z)
```



Tips

For column-oriented data analysis, use `waterfall(Z')` or `waterfall(X',Y',Z')`.

Algorithms

The range of `X`, `Y`, and `Z`, or the current setting of the axes `XLim`, `YLim`, and `ZLim` properties, determines the range of the axes (also set by `axis`). The range of `C`, or the current setting of the axes `CLim` property, determines the color scaling (also set by `caxis`).

The `CData` property for the patch graphics objects specifies the color at every point along the edge of the patch, which determines the color of the lines.

The waterfall plot looks like a mesh surface; however, it is a patch graphics object. To create a surface plot similar to `waterfall`, use the `meshz` function and set the `MeshStyle` property of the surface to `'Row'`. For a discussion of parametric surfaces and related color properties, see `surf`.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function accepts GPU arrays, but does not run on a GPU.

For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- This function operates on distributed arrays, but executes in the client MATLAB.

For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`axis` | `caxis` | `meshz` | `ribbon` | `surf`

Introduced before R2006a

web

Open web page or file in browser

Syntax

```
web
web(url)
web(url,opt)
web(url,opt1,...,optN)

stat = web( ___ )
[stat,h] = web( ___ )
[stat,h,url] = web( ___ )
```

Description

`web` opens an empty MATLAB web browser.

`web(url)` opens the page specified by `url` in the MATLAB web browser. If `url` is an external site, `web(url)` opens the page in your system browser. If multiple browsers are open, the page displays in the one that was most recently used.

`web(url,opt)` opens the page using the specified browser option, such as `'-new'` to create a new browser instance or `'-browser'` to use the system browser.

`web(url,opt1,...,optN)` opens the page using one or more browser options.

`stat = web( ___ )` returns the status of the operation: `0` if successful, `1` or `2` if unsuccessful. You can include any of the input arguments in previous syntaxes.

`[stat,h] = web( ___ )` returns a handle to the MATLAB web browser that allows you to close it using the command `close(h)`. If the page opens in a system browser, `web` returns an empty handle.

If you do not specify any inputs to the `web` function, such as `[stat,h] = web`, then the handle corresponds to the most recently used MATLAB web browser.

`[stat,h,url] = web( ___ )` returns the URL of the current page in the MATLAB web browser. If the page opens in a system browser, `web` returns an empty URL.

Examples

Open Web Page in System Browser

Open the MathWorks® Web site home page in the system browser.

```
url = 'https://www.mathworks.com';
web(url)
```

View File in MATLAB Web Browser

View an HTML file that resides on your system.

Create an HTML file by publishing an example program file. Copy the program file to the current folder so that the code can run during the publishing process.

```
program = fullfile(matlabroot,'help','techdoc',...
          'matlab_env','examples','fourier_demo2.m');
copyfile(program);
htmlFile = publish('fourier_demo2.m');
```

View the file by specifying the file name.

```
web(htmlFile)
```

View the file in a new instance of the browser that does not include a toolbar.

```
web(htmlFile,'-new','-notoolbar')
```

Alternatively, you can use the `file:///` URL scheme, as long as you include the full path. The `publish` function returns the path in the `htmlFile` output.

```
url = ['file:///',htmlFile];
web(url)
```

Send Email from System Browser

Send an email from your system browser's default mail application using the `mailto:` URL scheme.

To run this example, replace the value for `email` with a valid email address.

```
email = 'myaddress@provider.ext';
url = ['mailto:',email];
web(url)
```

Display Text in MATLAB Web Browser

View formatted text using the `text://` URL scheme.

```
web('text://<html><h1>Hello World</h1></html>')
```

Input Arguments

`url` — Web page address or file location
character vector | string

Web page address or file location, specified as a character vector or a string. File locations can include an absolute or relative path.

If `url` is an external site, then the page opens in your system browser. If `url` corresponds to a file in the installed product documentation, then the page displays in the MATLAB Help browser instead of the web browser.

Example: `'https:\\www.mathworks.com'`

Example: `"myfolder/myfile.html"`

`opt` — Browser option
`'-browser'` | `'-new'` | `'-noaddressbox'` | `'-notoolbar'`

Browser option, specified as one of the following. Options can appear in any order.

`'-browser'`	Opens the page in a system browser window instead of the MATLAB web browser. On Microsoft Windows and Apple Macintosh platforms, the operating system determines the system web browser. On other systems, the default is the Mozilla® Firefox® browser, but you can change the default using MATLAB web preferences.
`'-new'`	Opens the page in a new MATLAB web browser window. Does not apply when the page opens in your system browser.
`'-noaddressbox'`	Opens the page in a browser that does not display the address box. Only applies to new instances of the MATLAB web browser.
`'-notoolbar'`	Opens the page in a browser that does not display a toolbar or address box. Only applies to new instances of the MATLAB web browser.

Example: `'-new','-noaddressbox'`

Output Arguments

`stat` — Browser status
`0` | `1` | `2`

Browser status, returned as an integer with one of these values:

`0`	Found and started system browser.
`1`	Could not find system browser.
`2`	Found, but could not start system browser.

`h` — Handle to most recent MATLAB web browser
scalar

Handle to the most recent MATLAB web browser, returned as a scalar instance of the associated Java class. If the page opens in a system browser, `h` is empty, `[]`.

If you do not request the handle when you open the page, the handle might not correspond to your most recent use of the `web` function. Other MATLAB functionality also uses the `web` function, such as links to external sites from the Help browser.

`url` — Current page address
character vector | string

Current page address in the most recent MATLAB web browser, returned as a character vector or string. `url` has the same data type as the input argument `url`. If the page opens in a system browser, `url` is empty, `''`.

Limitations

- MATLAB Online only supports the `web(url)` syntax. Calling `web(url)` in MATLAB Online opens the page specified by `url` in your web browser instead of the MATLAB web browser.

Tips

- To open external sites in the MATLAB browser instead of your system browser by default, go to the **Home** tab, and in the **Environment** section, click **Preferences**. Select **MATLAB > Web** and in the **System Web browser** section, clear **Use system web browser when opening links to external sites (recommended)**. Using the system browser is recommended when opening external sites.
- If you plan to deploy an application that calls the `web` function using the MATLAB Compiler product, then use the `'-browser'` option.
- If you are displaying Japanese streaming text in the MATLAB web browser, specify a header that includes the `charset` attribute. For example:

```
web(['text://<html><head><meta http-equiv="content-type" ' ...
     'content="text/html;charset=utf-8"></head><body>TEXT</body></html>'])
```

Compatibility Considerations

`web` function opens external sites using your system browser by default.

Behavior changed in R2019b

The `web` function now opens external sites using your system browser by default. Previously, the `web` function opened external sites using the MATLAB browser. Using the system browser is recommended when opening external sites.

To use the MATLAB browser as the default browser for external sites, go to the **Home** tab, and in the **Environment** section, click **Preferences**. Select **MATLAB > Web** and in the **System Web browser** section, clear the **Use system web browser when opening links to external sites (recommended)** option.

See Also

`matlab.net.http.RequestMessage` | `urlread` | `urlwrite` | `weboptions` | `webread` | `websave` | `webwrite`

Topics

"Web Browsers and MATLAB"
"Specify Proxy Server Settings for Connecting to the Internet"
"Specify the System Browser for Linux Platforms"
"HTTP Interface"

Introduced before R2006a

weboptions

Specify parameters for RESTful web service

Syntax

```
options = weboptions
options = weboptions(Name,Value)
```

Description

`options = weboptions` returns a default `weboptions` object to specify parameters for a request to a web service. A `weboptions` object can be an optional input argument to the `webread`, `websave`, and `webwrite` functions. For options not supported by the `weboptions` function, see the “HTTP Interface”.

`options = weboptions(Name,Value)` specifies one or more properties of a `weboptions` object.

Examples

Default `weboptions` Object

Create a default `weboptions` object and display the default values for its properties.

```
options = weboptions
```

```
options = 

  weboptions with properties:

      CharacterEncoding: 'auto'
              UserAgent: 'MATLAB 9.7.0.1112323 (R2019b)'
                Timeout: 5
               Username: ''
               Password: ''
                KeyName: ''
               KeyValue: ''
            ContentType: 'auto'
```

```
          ContentReader: []
              MediaType: 'auto'
          RequestMethod: 'auto'
            ArrayFormat: 'csv'
           HeaderFields: []
    CertificateFilename: 'default'
```

User Name and Password in weboptions Object

Set your web service user name and password in a `weboptions` object. You can use the object as an input argument to `webread`, `websave`, or `webwrite` when your web service requires authentication.

```
options = weboptions('Username','jdoe','Password','mypassword');
```

The password is obscured when you display the `weboptions` object. However, the object stores the password as plain text. You can retrieve the password from the `weboptions.Password` property.

```
options.Password
```

```
ans =
'mypassword'
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `weboptions('Timeout',60)` creates a `weboptions` object that sets the timeout connection duration to 60 seconds.

`CharacterEncoding` — Character encoding
`'auto'` (default) | string scalar | character vector

The encoding used by `webread` to convert web content to characters, specified as a string scalar or character vector. Common encodings include `'US-ASCII'`, `'UTF-8'`,

`'latin1'`, `'Shift_JIS'`, and `'ISO-8859-1'`. The default encoding depends on the content type. If you get garbled text, then the `webread` encoding might be different from the encoding used by the document. Try setting `CharacterEncoding` to `UTF-8`.

`UserAgent` — User agent identification
`['MATLAB ' version]` (default) | string scalar | character vector

User agent identification, specified as a string scalar or character vector indicating the client user agent.

`Timeout` — Time out connection duration
5 (default) | positive numeric scalar | `Inf`

Time out connection duration in seconds, specified as a positive numeric scalar. The value is the number of seconds to wait to receive the initial response (header) from the server after sending the last packet of a request. `Timeout` is equivalent to the ResponseTimeout property in the `matlab.net.http.HTTPOptions` class. The maximum value is 2147.483647 seconds. Use `Inf` to set the maximum value.

Some operating systems have a maximum timeout enforced by the system. This timeout takes effect even if the value of `Timeout` is greater than the maximum. For example, on Windows 10, this timeout is 21 seconds.

`Username` — User identifier
`''` (default) | string scalar | character vector

User identifier, specified as a string scalar or character vector for basic and/or digest HTTP authentication (no encryption). For information about Basic HTTP Authentication Scheme, see https://tools.ietf.org/html/rfc7617. For information about HTTP Digest Access Authentication, see https://tools.ietf.org/html/rfc7616.

`Password` — User authentication password
`''` (default) | string scalar | character vector

User authentication password, specified as a string scalar or character vector for basic and/or digest HTTP authentication (no encryption). If you display a `weboptions` object with `Password` set, then the value is displayed as a character vector containing '*'. However, the object stores the value of `Password` as plain text.

`KeyName` — Name of key
`''` (default) | string scalar | character vector

Name of a key, specified as a string scalar or character vector. `KeyName` is an additional name to add to the HTTP request header. For example, `KeyName` can be a web service API key name.

Example: `weboptions('KeyName','duration','KeyValue',7)` creates a `weboptions` object that contains a key name, `duration`, defined by a web service.

`KeyValue` — Value of key
`''` (default) | string scalar | character vector | numeric | logical

Value of a key, specified as a string scalar, a character vector, or a numeric or logical value to add to the HTTP request header. `KeyValue` is the value of a key specified by `KeyName`.

Example: `weboptions('KeyName','duration','KeyValue',7)` creates a `weboptions` object that contains a key value, `7`, paired with a key name, `duration`.

`HeaderFields` — Names and values of header fields
m-by-2 array of strings or cell array of character vectors

Names and values of header fields, specified as an m-by-2 array of strings or cell array of character vectors to add to the HTTP request header. `HeaderFields{i,1}` is the name of a field and `HeaderFields{i,2}` is its value.

These header fields add to or replace fields automatically added by `webread`, `webwrite`, or `websave`. Normally these fields are added, but if the name of one of these fields is a case-insensitive match to one of the fields that would be automatically added, and that field does not support multiple values (for example, Content-Type), then the value you specify is used instead. Some fields whose value is necessary to send a request successfully, such as Connection and Content-Length, cannot be overridden.

Example: `weboptions('HeaderFields',{'Content-Length' '78';'Content-Type' 'application/json'})` creates a `weboptions` object that contains two header fields: `Content-Length` with value `78` and `Content-Type` with value `application/json`.

`ContentType` — Content type
`'auto'` (default) | string scalar | character vector

Content type, specified as a string scalar or character vector. Use `ContentType` to request that the server preferentially return data in a particular format. `webread` uses this value to convert the response to a MATLAB type. The server returns this content type if possible, but is not obligated to do so.

ContentType Specifier	Output Type
`'auto'` (default)	Output type automatically determined based on content type specified by the server.

`'text'`	Character vector for content types:
	
	`text/plain`
	`text/html`
	`text/xml`
	`application/xml`
	`application/javascript`
	`application/x-javascript`
	`application/x-www-form-urlencoded`
	
	If a web service returns a MATLAB file with a `.m` extension, the function returns its content as a character vector.

`'image'`	Numeric or logical matrix for `image/format` content. If the first output argument is an indexed image, the second output argument is the colormap, and the third output argument is the alpha channel. For supported image formats, see “Supported File Formats for Import and Export”.
`'audio'`	Numeric matrix for `audio/format` content with numeric scalar sampling rate as a second output argument. For supported audio formats, see “Supported File Formats for Import and Export”.
`'binary'`	`uint8` column vector for binary content (that is, content not to be treated as type `char`).
`'table'`	Scalar table object for spreadsheet and CSV (`text/csv`) content.

ContentType Specifier	Output Type
`'json'`	`char`, numeric, logical, structure, or cell array, for `application/json` content.
`'xmldom'`	Java Document Object Model (DOM) node for `text/xml` or `application/xml` content. If not specified, the function returns XML content as a character vector.
`'raw'`	`char` column vector for `'text'`, `'xmldom'`, and `'json'` content. The function returns any other content type as a `uint8` column vector.

Example: `weboptions('ContentType','text')` creates a `weboptions` object that instructs `webread` to return text, JSON, or XML content as a character vector.

`ContentReader` — Content reader
`[]` (default) | function handle

Content reader, specified as a function handle. You can create a `weboptions` object with `ContentReader` specified, and pass the object as an input argument to `webread`. Then `webread` downloads data from a web service and reads the data with the function specified by the function handle. `webread` ignores `ContentType` when `ContentReader` is specified.

Example: `weboptions('ContentReader',@readtable)` creates a `weboptions` object that instructs `webread` to use `readtable` to read content as a table.

`MediaType` — Media type
`'auto'` (default) | `'application/x-www-form-urlencoded'` | string scalar | character vector | `matlab.net.http.MediaType`

Media type, specified as a string scalar, a character vector, or a `matlab.net.http.MediaType` object. `MediaType` specifies the type of data `webwrite` sends to the web service. It specifies the content type that MATLAB specifies to the server, and it controls how the `webwrite` `data` argument, if specified, is converted.

For a complete list of media types, see Internet Media Types.

The default value is `'auto'` which indicates that MATLAB chooses the type based on the input to `webwrite`. If using `PostName/PostValue` argument pairs, then MATLAB uses

`'application/x-www-form-urlencoded'` to send the pairs. If using a `data` argument that is a scalar string or character vector, then MATLAB assumes it is a form-encoded string and sends it as-is using `'application/x-www-form-urlencoded'`. If `data` is anything else, then MATLAB converts it to JSON using `jsonencode` and uses the content type `'application/json'`.

If you specify a `MediaType` containing `'json'` or `'javascript'`, and `data` is a character vector, then it is sent as-is. All other types, including scalar strings, are converted using `jsonencode`.

If you specify `'application/x-www-form-urlencoded'`, then `PostName/PostValue` pairs are sent form-encoded. `data`, if present, must be a string or character vector to be sent as-is.

If you specify a `MediaType` that contains `'xml'`, and `data` is a Document Object Model object (a Java `org.apache.xerces.dom.DocumentImpl`), then it is converted to XML. `data`, if present, must be a string or character vector to be sent as-is.

If you specify any other `MediaType`, and `data` is a string or character vector, then `weboptions` sends the value as-is.

`PostName/PostValue` pairs are accepted only for `MediaType` values `'auto'` and `'application/x-www-form-urlencoded'`, and character vectors are always sent as-is regardless of the `MediaType`.

You can specify semicolon-separated `name=value` parameters within the `MediaType` string, for example, `'application/json; odata=verbose'`. Some servers require this format as part of the Content-Type header field in the request.

Example: `weboptions('MediaType','application/json')` creates a `weboptions` object that instructs `webwrite` to encode character vector data as JSON to post it to a web service.

`RequestMethod` — HTTP request method

`'auto'` (default) | string scalar | character vector | `matlab.net.http.RequestMethod` enumeration

HTTP request method, specified as a string scalar, a character vector, or a `matlab.net.http.RequestMethod` enumeration as one of these values:

- `'auto'`

  - `webread` and `websave` use the HTTP GET method.
  - `webwrite` uses the HTTP POST method.
- `'get'` for use with the `webread` and `websave` functions.
- `'post'` for use with the `webread`, `webwrite`, and `websave` functions.
- `'put'` for use with the `webread`, `webwrite`, and `websave` functions.
- `'delete'` for use with the `webread`, `webwrite`, and `websave` functions.
- `'patch'` for use with the `webread`, `webwrite`, and `websave` functions.

The `webread` and `websave` functions put the query into the URL regardless of the `RequestMethod`. `webwrite` puts the query into the data regardless of the `RequestMethod`.

Example: `weboptions('RequestMethod','post')` creates a `weboptions` object that instructs `webread`, `websave`, or `webwrite` to use the HTTP POST method of a web service.

`ArrayFormat` — Format to form-encode query or post values that represent multiple values
`'csv'` (default) | `'json'` | `'repeating'` | `'php'`

Format to form-encode query or post values that represent multiple values, specified as `'csv'`, `'json'`, `'repeating'`, or `'php'`. A query or post value contains multiple values if it is

- A numeric, logical, or `datetime` vector
- A character array with more than one row
- A cell vector, where each element is a numeric, logical, or `datetime` scalar or a character vector with one row

No other data types or dimensions are supported.

This table shows form-encoded conversions for each format, for a query parameter named `'parameter'` and a query value of `[1 2 3]`. The web service specifies the conversion to use.

ArrayFormat Specifier	Form-Encoded Conversion
`'csv'` (default)	`parameter=1,2,3`
`'json'`	`parameter=[1,2,3]`

ArrayFormat Specifier	Form-Encoded Conversion
'repeating'	parameter=1&parameter=2&parameter=3
'php'	parameter[]=1&parameter[]=2&parameter[]=3

To encode a scalar as a one-element array with the `'json'` or `'php'` specifiers, place the scalar in a one-element cell array.

Example: `weboptions('ArrayFormat','repeating')` creates a `weboptions` object that instructs `webread`, `websave`, or `webwrite` to form-encode any query or post value with multiple values as repeating query parameters.

`CertificateFilename` — File name of root certificates
`'default'` (default) | string scalar | character vector

File name of root certificates, specified as a string scalar or character vector denoting the location of a file containing certificates. The file is in privacy-enhanced mail (PEM) format. The location must be in the current folder, in a folder on the MATLAB path, or a full or relative path to a file. If you specify the value `'default'`, then system certificates are used.

If you request an HTTPS connection, then the certificate from the server is validated against the certification authority certificates in the PEM file. Standard HTTPS mechanisms use this validation to validate the signature on the server certificate and the entire certificate chain. If verification fails, a connection is not allowed. You can disable the verification in cases where the server's certificate does not match the URI used to access it, by creating a `matlab.net.http.RequestMessage` and setting the `matlab.net.http.HTTPOptions.VerifyServerName` property to `false`. Use this option if you are confident that you are communicating directly with the intended server.

If you need additional certificates, add them to the system certificates. PEM files are ASCII files which are easily modified. Since security of HTTPS connections depends on the integrity of this file, protect it appropriately. MATLAB does not manage certificates or certificate files, but there are third-party tools for managing PEM files.

If `CertificateFilename` is empty, then MATLAB checks if the certificate domain of the server matches the host name of the server and that it is not expired. The signature is not validated.

Set `CertificateFilename` to empty (`''`) only if you cannot establish a connection due to a missing or expired certificate.

Attributes:

GetAccess

public

SetAccess

public

See Also

`HTTPOptions` | `RequestMethod` | `webread` | `websave` | `webwrite`

Topics

"HTTP Interface"

Introduced in R2014b

webread

Read content from RESTful web service

Syntax

```
data = webread(url)
data = webread(url,
QueryName1,QueryValue1,...,QueryNameN,QueryValueN)
data = webread( ___ ,options)
```

```
[data,colormap,alpha] = webread( ___ )
```

```
[data,Fs] = webread( ___ )
```

Description

`data = webread(url)` reads content from the web service specified by `url` and returns the content in `data`.

The web service provides a “RESTful” on page 1-16863 that returns data formatted as an internet media type such as JSON, XML, image, or text.

`data = webread(url, QueryName1,QueryValue1,...,QueryNameN,QueryValueN)` appends query parameters to `url`, as specified by one or more pairs of name-value arguments. To put a query into the body of the message, use `webwrite`. The web service defines the query parameters.

`data = webread( ___ ,options)` adds other HTTP request options, specified by the `weboptions` object `options`. You can use this syntax with any of the input arguments of the previous syntaxes.

To return data as a specific output type, specify the `ContentType` property of `options`.

To read content with a function, specify the `ContentReader` property of `options` as a handle to the function. `webread` downloads data from a web service and reads the data with the specified function:

- If you specify a handle to a function that returns multiple output arguments, `webread` returns all output arguments.
- If you specify a handle to a function that returns no output argument (such as Image Processing Toolbox function `@implay` for video files), `webread` returns no output argument.

`[data,colormap,alpha] = webread( ___ )` reads an image from the web service specified by `url` and returns the image in `data`. You can use the previous syntaxes to return the image only. Use this syntax to return the colormap and alpha channels associated with the image.

`webread` returns an image when the HTTP response has a `Content-Type` header field that specifies an image media type and if `imread` supports the image format. For supported image formats, see “Supported File Formats for Import and Export”.

`[data,Fs] = webread( ___ )` reads audio data from the web service specified by `url` and returns the audio data in `data`. You can use the previous syntaxes to return the audio data only. Use this syntax to return the sample rate of the audio data in hertz.

`webread` returns audio data when the HTTP response has a `Content-Type` header field that specifies an audio media type and if `audioread` supports the audio format. For supported audio formats, see “Supported File Formats for Import and Export”.

Examples

Read Image from Website

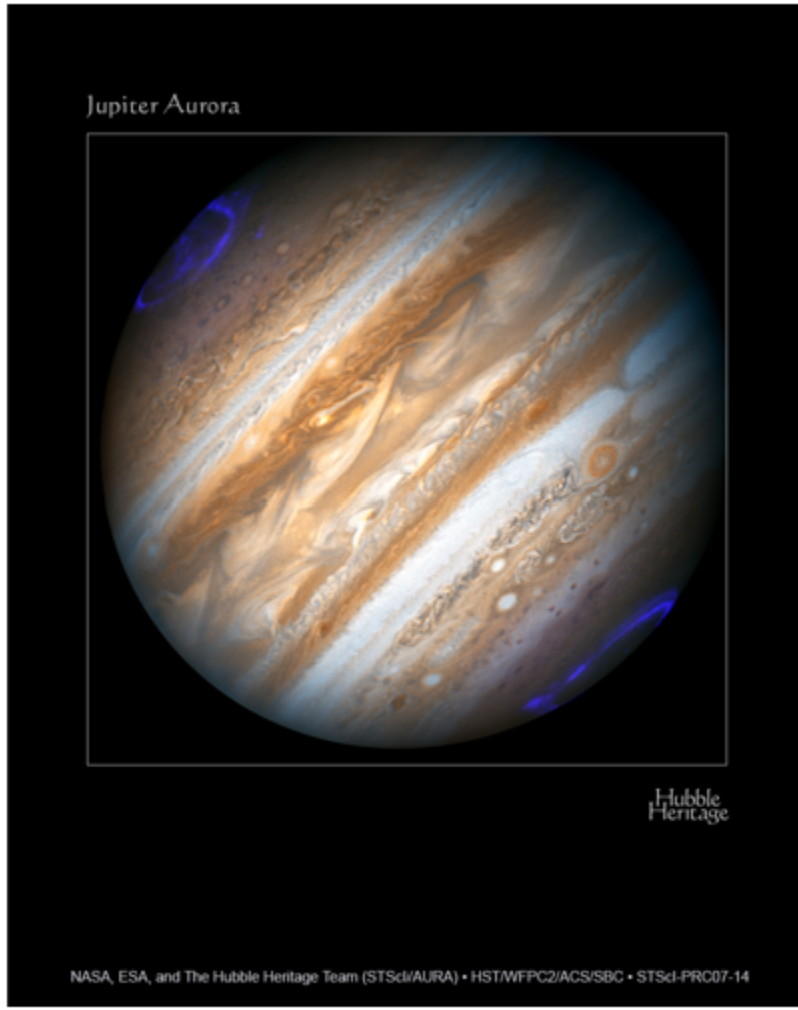
Read an image of Jupiter from the Hubble Heritage website and display it.

```
url = 'http://heritage.stsci.edu/2007/14/images/p0714aa.jpg';
rgb = webread(url);
whos rgb
```

```
  Name         Size                 Bytes  Class    Attributes

  rgb       1000x800x3            2400000  uint8
```

Resize and display the image.

```
rgb = imresize(rgb,0.6);
imshow(rgb)
```


Jupiter Aurora
Hubble Heritage
NASA, ESA, and The Hubble Heritage Team (STScI/AURA) • HST/WFPC2/ACS/SBC • STScI-PRC07-14

1-16855

Jupiter image courtesy of NASA, ESA, and the Hubble Heritage Team (STScI/AURA). (See Hubble Heritage Information Center for terms of use.)

Read Data from Web Service API

Read temperature data for the USA from the World Bank Climate Data API. Plot temperatures from the years 1901–2012.

Read data from the World Bank. This API returns data as a JSON object.

```
api = 'http://climatedataapi.worldbank.org/climateweb/rest/v1/';
url = [api 'country/cru/tas/year/USA'];
S = webread(url)
```

```
S =

112x1 struct array with fields:

    year
    data
```

`webread` converts the JSON object to a structure array. Each structure contains the year and the average temperature in the USA for that year, in degrees Celsius.

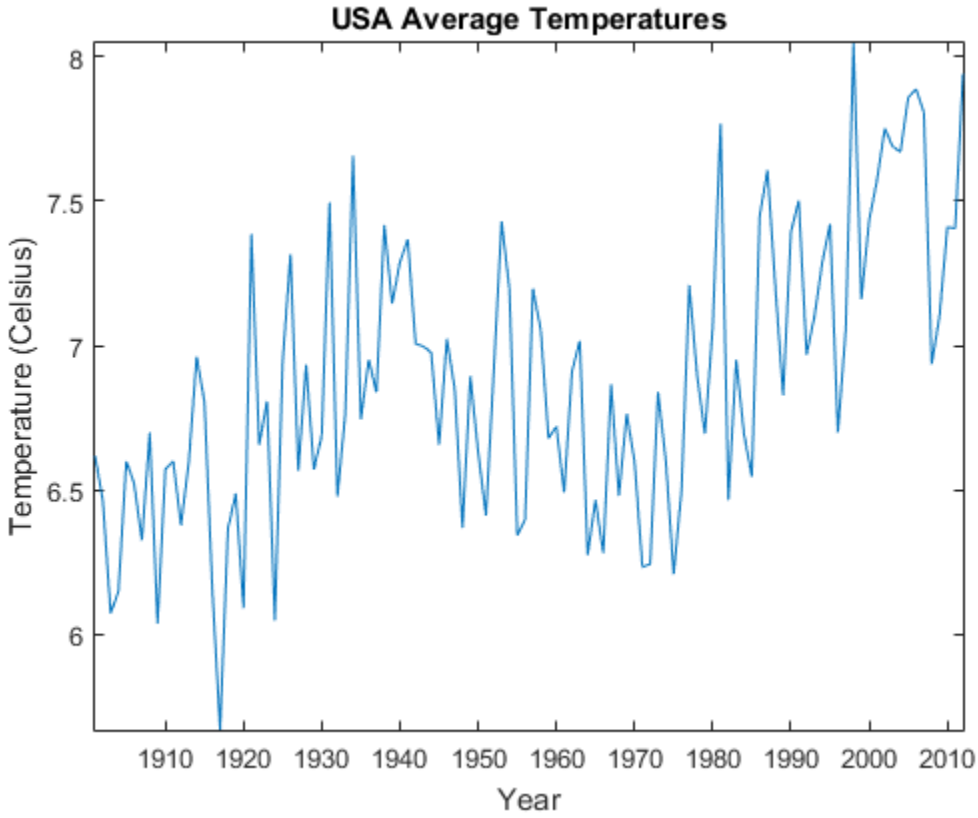
Display the temperature for the first year.

```
S(1)
```

```
ans =

    year: 1901
    data: 6.6187
```

Plot the average temperatures. Concatenate `S.year` and `S.data` into arrays and plot them.

```
year = [S.year];
data = [S.data];
plot(year,data)
xlabel('Year');
ylabel('Temperature (Celsius)');
title('USA Average Temperatures')
axis tight
```



API and data courtesy of the World Bank: Climate Data API. (See World Bank: Climate Data API for more information about the API, and World Bank: Terms of Use.)

Specify Web Service Query Parameters

Search the File Exchange for files uploaded within the past seven days that contain the word *Simulink*.

Specify the query parameters. `webread` appends web service query parameter names and values to the URL. The File Exchange web service defines the `term` and `duration` query parameters, not the `webread` function.

```
url = 'https://www.mathworks.com/matlabcentral/fileexchange/';
data = webread(url,'term','simulink','duration',7);
```

`webread` returns the HTML for the search result page as a character array.

Specify Request Option

Specify an additional request option to read data from the World Bank Climate Data API to a character array.

Create a `weboptions` object and set its `ContentType` to `'text'`. The `webread` function converts the JSON object to a character array instead of a structure array. Display the beginning of the character array.

```
api = 'http://climatedataapi.worldbank.org/climateweb/rest/v1/';
url = [api 'country/cru/tas/year/USA'];
options = weboptions('ContentType','text');
data = webread(url,options);
data(1:62)
```

```
ans =

[{"year":1901,"data":6.6187487},{"year":1902,"data":6.4643273}
```

API and data courtesy of the World Bank: Climate Data API. (See World Bank: Climate Data API for more information about the API, and World Bank: Terms of Use.)

Read Data with POST Request

Send an HTTP POST request to search File Exchange for files uploaded within the past seven days that contain the word *Simulink*.

```
url = 'https://www.mathworks.com/matlabcentral/fileexchange/';
options = weboptions('RequestMethod','post');
data = webread(url,'term','simulink','duration',7,options);
```

Many web services provide a POST method for requesting data in addition to GET.

Specify Date and Time as Query Parameters

Read a Blue Marble: Next Generation image for December 2004 from the NASA Earth Observation (NEO) Web Mapping Service.

Specify the date of the requested image with a `datetime` object. Specify the `Format` property of `D` so that the format matches the format required by the web service.

```
url = 'http://neowms.sci.gsfc.nasa.gov/wms/wms';
D = datetime(2004,12,01,'Format','yyyy-MM-dd');
rgb = webread(url,'Time',D, ...
     'Service','WMS','Layers','BlueMarbleNG-TB','CRS','CRS:84', ...
     'Format','image/jpeg','Height',256,'Width',512, ...
     'BBOX','-180.0,-90.0,180.0,90.0','Version','1.3.0','Request','GetMap');
imshow(rgb)
```

`webread` converts `datetime` objects so that they can be values of web service query parameters. All the name-value pairs in the example provide query parameters specified by the NEO Web Mapping Service.

Blue Marble: Next Generation + Topography and Bathymetry image courtesy of NASA's Earth Observatory. Access to imagery and services provided by the NEO Web Mapping

Service (WMS). (See NASA Earth Observations for credit and terms of use. For WMS query parameters, search the NASA Earth Observations site, WMS 1.3.0 Capabilities.)

Input Arguments

`url` — URL to web service

character vector | string scalar

URL to a web service, specified as a character vector or string scalar. The web service implements a RESTful interface. See “RESTful” on page 1-16863 for more information.

Example: `webread('https://www.mathworks.com/matlabcentral')` reads the web page and returns its HTML as a character array.

`QueryName1,QueryValue1,...,QueryNameN,QueryValueN` — Web service query parameters

name-value pairs

Web service query parameters, specified as one or more pairs of name-value arguments. A `QueryName` argument must specify the name of a query parameter, as a character vector or string scalar. A `QueryValue` argument must be a character vector, a string scalar, or a numeric, logical, or `datetime` value that specifies the value of the query parameter. Numeric, logical, and `datetime` values can be in arrays. The web service defines name-value pairs that it accepts as part of a request.

When you specify `QueryValue` as a `datetime` object, you must specify its `Format` property so that it is consistent with the format required by the web service. If the `Format` property includes a time zone or offset, and the `datetime` object is not zoned, then `webread` specifies `'Local'` as the time zone.

When `QueryValue` contains multiple values in an array, you might need to specify the `ArrayFormat` property of a `weboptions` object to form-encode the array as specified by the web service.

Example: `webread('https://www.mathworks.com/matlabcentral/fileexchange/','term','webread')` retrieves a list of files uploaded to the File Exchange that contain the word `webread`.

`options` — Additional HTTP request options

`weboptions` object

Additional HTTP request options, specified as a `weboptions` object.

You can specify the `ContentType` property of a `weboptions` object, and pass the object as an input argument to `webread`. Then `webread` returns `data` as that type of output. The table lists the valid content types you can specify in a `weboptions` object.

`ContentType` Specifier	**Output Type**
`'auto'` (default)	Output type automatically determined based on content type specified by the server.

`'text'`	Character vector for content types:
	
	`text/plain`
	`text/html`
	`text/xml`
	`application/xml`
	`application/javascript`
	`application/x-javascript`
	`application/x-www-form-urlencoded`
	
	If a web service returns a MATLAB file with a `.m` extension, the function returns its content as a character vector.

`'image'`	Numeric or logical matrix for `image/format` content. If the first output argument is an indexed image, the second output argument is the colormap, and the third output argument is the alpha channel. For supported image formats, see “Supported File Formats for Import and Export”.
`'audio'`	Numeric matrix for `audio/format` content with numeric scalar sampling rate as a second output argument. For supported audio formats, see “Supported File Formats for Import and Export”.

ContentType Specifier	Output Type
`'binary'`	`uint8` column vector for binary content (that is, content not to be treated as type `char`).
`'table'`	Scalar table object for spreadsheet and CSV (`text/csv`) content.
`'json'`	`char`, numeric, logical, structure, or cell array, for `application/json` content.
`'xmldom'`	Java Document Object Model (DOM) node for `text/xml` or `application/xml` content. If not specified, the function returns XML content as a character vector.
`'raw'`	`char` column vector for `'text'`, `'xmldom'`, and `'json'` content. The function returns any other content type as a `uint8` column vector.

See `weboptions` for all request options that are `weboptions` properties.

Output Arguments

`data` — Content from web service
scalar | array | structure | table

Content read from a web service, returned as a scalar, array, structure, or table.

`colormap` — Colormap associated with indexed image
numeric array

Colormap associated with an indexed image, returned as a numeric array.

`alpha` — Alpha channels associated with indexed image
numeric array

Alpha channels associated with an indexed image, returned as a numeric array.

`Fs` — Sample rate of audio data in hertz
positive numeric scalar

Sample rate of audio data in hertz, returned as a positive numeric scalar.

More About

RESTful

REST means *representational state transfer*, a common architectural style for web services. RESTful interfaces provide standard HTTP methods such as GET, PUT, POST, or DELETE.

Tips

- For functionality not supported by the RESTful web services functions, see the “HTTP Interface”.
- `webread` supports HTTP GET and POST methods. Many web services provide both GET and POST methods to request data. To send an HTTP POST request, specify the `RequestMethod` property of `options` as `'post'`. However, `webread` puts query options into the `url`, not in the body of the request message. To put a query into the body, use `webwrite`.
- For HTTP POST requests, the `webread` function supports only the `application/x-www-form-urlencoded` media type. To send a POST request with content of any other internet media type, use `webwrite`.
- This function does not examine the document contents to determine how to process it. For example, HTML and XML documents often contain a `<meta>` tag that specifies the document character encoding. If the encoding is different from the default `webread` encoding, then specify the correct `CharacterEncoding` option in `weboptions`.

See Also

`audioread` | `datetime` | `imread` | `jsondecode` | `readtable` | `weboptions` | `websave` | `webwrite` | `xmlread`

Topics

“Download Data from Web Service”
“Convert Data from Web Service”

"Represent Dates and Times in MATLAB"
"HTTP Interface"

External Websites

Representational State Transfer
JavaScript Object Notation (JSON)

Introduced in R2014b

websave

Save content from RESTful web service to file

Syntax

```
outfilename = websave(filename,url)
outfilename = websave(filename,url,
QueryName1,QueryValue1,...,QueryNameN,QueryValueN)
outfilename = websave( ___ ,options)
```

Description

`outfilename = websave(filename,url)` saves content from the web service specified by `url` and writes it to `filename`. The `websave` function returns the full `filename` path as `outfilename`.

The web service provides a “RESTful” on page 1-16870 that returns data formatted as an internet media type such as JSON, XML, image, or text.

`outfilename = websave(filename,url, QueryName1,QueryValue1,...,QueryNameN,QueryValueN)` appends query parameters to `url`, as specified by one or more pairs of name-value arguments. The web service defines the query parameters.

`outfilename = websave( ___ ,options)` adds other HTTP request options, specified by the `weboptions` object `options`. You can use this syntax with any of the input arguments of the previous syntaxes.

`websave` supports HTTP GET and POST methods. To send an HTTP POST request, specify the `RequestMethod` property of `options` as `'post'`. Many web services provide both GET and POST methods to request data.

Examples

Save Image from Website

Save an image of Jupiter from the Hubble Heritage website.

```
url = 'http://heritage.stsci.edu/2007/14/images/p0714aa.jpg';
filename = 'jupiter_aurora.jpg';
outfilename = websave(filename,url)
```

```
outfilename =

C:\Libraries\Documents\jupiter_aurora.jpg
```

`websave` saves the image as a JPEG file, as specified by the Hubble web service, even when you give `filename` a different extension. (Jupiter image courtesy of NASA, ESA, and the Hubble Heritage Team (STScI/AURA). See Hubble Heritage Information Center for terms of use.)

Save Search Results to File

Search File Exchange for files uploaded within the past seven days that contain the word *Simulink* and display the results of the search.

```
url = 'https://www.mathworks.com/matlabcentral/fileexchange/';
filename = 'simulink_search.html';
outfilename = websave(filename,url,'term','simulink','duration',7)
```

```
outfilename =

C:\Libraries\Documents\simulink_search.html
```

Display the HTML file in a web browser.

```
web(outfilename)
```

Save Data to File

Save sunspot data from the National Geophysical Data Center (NGDC) to an ASCII file. Use a `weboptions` object to set the timeout value to `Inf` so that the connection does not time out.

```
api = 'http://www.ngdc.noaa.gov/stp/space-weather/';
url = [api 'solar-data/solar-indices/sunspot-numbers/' ...
```

```
       'american/lists/list_aavso-arssn_yearly.txt'];
filename = 'sunspots_annual.txt';
options = weboptions('Timeout',Inf);
outfilename = websave(filename,url,options)
```

```
outfilename =

C:\Libraries\Documents\sunspots_annual.txt
```

Aggregated data and web service courtesy of the NGDC. Sunspot data courtesy of the American Association of Variable Star Observers (AAVSO), originally published in AAVSO Sunspot Counts: 1943-2013, AAVSO Solar Section (R. Howe, Chair). (See NGDC Privacy Policy, Disclaimer, and Copyright for NGDC terms of use, and AAVSO Solar Section for AAVSO terms of use.)

Save Data with POST Request

Send an HTTP POST request to search File Exchange for files uploaded within the past seven days that contain the word *Simulink* . Save the results of the search to an HTML file.

```
url = 'https://www.mathworks.com/matlabcentral/fileexchange/';
filename = 'simulink_search.html';
options = weboptions('RequestMethod','post');
outfilename = websave(filename,url,'term','simulink','duration',7,options);
```

Many web services provide a POST method for requesting data in addition to GET.

Save Data for Specified Date and Time

Save the Blue Marble: Next Generation image for December 2004 from the NASA Earth Observation (NEO) Web Mapping Service.

Specify the date of the requested image with a `datetime` object. Specify the `Format` property of `D` so that the format matches the format required by the web service.

```
url = 'http://neowms.sci.gsfc.nasa.gov/wms/wms';
D = datetime(2004,12,01,'Format','yyyy-MM-dd');
filename = 'BlueMarble.jpg';
outfilename = websave(filename,url,'Time',D, ...
```

```
     'Service','WMS','Layers','BlueMarbleNG-TB','CRS','CRS:84', ...
     'Format','image/jpeg','Height',256,'Width',512, ...
     'BBOX','-180.0,-90.0,180.0,90.0','Version','1.3.0','Request','GetMap')
```

```
outfilename =

C:\Libraries\Documents\BlueMarble.jpg
```

`websave` converts `datetime` objects so that they can be values of web service query parameters. All the name-value pairs in the example provide query parameters specified by the NEO Web Mapping Service.

Blue Marble: Next Generation + Topography and Bathymetry image courtesy of NASA's Earth Observatory. Access to imagery and services provided by the NEO Web Mapping Service (WMS). (See NASA Earth Observations for credit and terms of use. For WMS query parameters, search the NASA Earth Observations site, WMS 1.3.0 Capabilities.)

Save JSON Data

Read JSON data from a website and save in file `test.txt`.

```
uri = matlab.net.URI('http://httpbin.org/get');
websave('test.txt',uri,weboptions('ContentType','json'));
```

Read the text from the file into a structure of JSON data.

```
js = jsondecode(fileread('test.txt'))
```

```
js =

  struct with fields:

       args: [1×1 struct]
    headers: [1×1 struct]
     origin: '144.444.4.4'
        url: 'http://httpbin.org/get'
```

Input Arguments

`filename` — Name of file to save content to
character vector | string scalar

Name of file to save content to, specified as a character vector or string scalar. `websave` saves the content as is. `websave` ignores `options.ContentType` and `options.ContentReader`, even if these properties are set.

Example: `websave('matlabcentral.html','https://www.mathworks.com/matlabcentral')` reads the web page and saves its HTML to the file `matlabcentral.html`.

`url` — URL to web service
character vector | string scalar

URL to a web service, specified as a character vector or string scalar. The web service implements a RESTful interface. See “RESTful” on page 1-16870 for more information.

`QueryName1,QueryValue1,...,QueryNameN,QueryValueN` — Web service query parameters
name-value pairs

Web service query parameters, specified as one or more pairs of name-value arguments. A `QueryName` argument must specify the name of a query parameter. A `QueryValue` argument must be a character vector, a string scalar, or a numeric, logical, or `datetime` value that specifies the value of the query parameter. Numeric, logical, and `datetime` values can be in arrays. The web service defines name-value pairs that it accepts as part of a request.

When you specify `QueryValue` as a `datetime` object, you must specify its `Format` property to be consistent with the format required by the web service. If the `Format` property includes a time zone or offset, and the `datetime` object is not zoned, then `websave` specifies `'Local'` as the time zone.

When `QueryValue` contains multiple values in an array, you might need to specify the `ArrayFormat` property of a `weboptions` object to form-encode the array as specified by the web service.

Example: `websave('webread_search.html','https://www.mathworks.com/matlabcentral/fileexchange/','term','simulink')` retrieves a list of files uploaded to the File Exchange that contain the word *`simulink`* and saves the search results to an HTML file.

`options` — Additional HTTP request options
`weboptions` object

Additional HTTP request options, specified as a `weboptions` object. For all request options that are `weboptions` properties, see `weboptions`.

More About

RESTful

REST means *representational state transfer*, a common architectural style for web services. RESTful interfaces provide standard HTTP methods such as GET, PUT, POST, or DELETE.

Tips

- For functionality not supported by the RESTful web services functions, see the “HTTP Interface”.
- For HTTP POST requests, the `websave` function supports only the `application/x-www-form-urlencoded` media type. To send a POST request with content of any other internet media type, use `webwrite`.

See Also

`datetime` | `weboptions` | `webread` | `webwrite`

Topics

“Download Web Page and Files”
“Represent Dates and Times in MATLAB”
“HTTP Interface”

External Websites

Representational State Transfer
JavaScript Object Notation (JSON)

Introduced in R2014b

webwrite

Write data to RESTful web service

Syntax

```
response = webwrite(url,
PostName1,PostValue1,...,PostNameN,PostValueN)
response = webwrite(url,data)
response = webwrite( ___ ,options)
```

Description

`response = webwrite(url, PostName1,PostValue1,...,PostNameN,PostValueN)` writes content to the web service specified by `url` and returns `response`. The input arguments `PostName1,PostValue1,...,PostNameN,PostValueN` specify the content as name-value pairs. `webwrite` form-encodes the name-value pairs in the body of an HTTP POST request to the web service. The web service defines `response`.

The web service provides a “RESTful” on page 1-16877 that returns data formatted as an internet media type such as JSON, XML, image, or text.

`response = webwrite(url,data)` posts `data` to the web service specified by `url` and sets the media type based on the `data`.

The input argument `data` specifies the content as a form-encoded character array. `webwrite` puts `data` in the body of an HTTP POST request to the web service. The web service defines `response`.

`response = webwrite( ___ ,options)` adds other HTTP request options, specified by the `weboptions` object `options`. You can use this syntax with any of the input arguments of the previous syntaxes.

To write content as an internet media type other than a form-encoded character array (`'application/x-www-form-urlencoded'`), specify the `MediaType` property of `options`.

To request data with an HTTP POST request and read the response with a function, specify the `ContentReader` property of `options` as a handle to the function. If you specify a handle to a function that returns multiple output arguments, `webwrite` returns all output arguments.

Examples

Write Data to Web Service

Write a number to a channel feed on the ThingSpeak server and read it back.

To run this code, create a ThingSpeak account. Call `webwrite` using the Write API key and Channel ID from your ThingSpeak account. The default field name is `'field1'`.

```
thingSpeakURL = 'http://api.thingspeak.com/';
thingSpeakWriteURL = [thingSpeakURL 'update'];
writeApiKey = 'Your Write API Key';
fieldName = 'field1';
fieldValue = 42;
response = webwrite(thingSpeakWriteURL,'api_key',writeApiKey,fieldName,fieldValue)
```

If this call to `webwrite` is the first update to your ThingSpeak channel, `response` is `1`.

Read back the number you wrote to your channel. ThingSpeak provides a different URL to get the last entry to your channel. Your Channel ID is part of the URL.

```
channelID = num2str(Your Channel ID);
thingSpeakReadURL = [thingSpeakURL 'channels/' channelID '/fields/' fieldName '/last']
data = webread(thingSpeakReadURL,'api_key',writeApiKey)
```

```
data =

42
```

Write Form-Encoded Character Array

Write a number as a form-encoded character array to a channel feed on the ThingSpeak server and read it back.

To run this code, create a ThingSpeak account. Call `webwrite` using the Write API key and Channel ID from your ThingSpeak account. Encode your Write API Key and the number as a form-encoded character array. The default field name is `'field1'`.

```
thingSpeakURL = 'http://api.thingspeak.com/';
thingSpeakWriteURL = [thingSpeakURL 'update'];
writeApiKey = 'Your Write API Key';
data = 42;
data = num2str(data);
data = ['api_key=',writeApiKey,'&field1=',data];
response = webwrite(thingSpeakWriteURL,data)
```

If this call to `webwrite` is the first update to your ThingSpeak channel, `response` is `1`.

Read back the number you wrote to your channel. ThingSpeak provides a different URL to get the last entry to your channel. Your Channel ID is part of the URL.

```
channelID = num2str(Your Channel ID);
thingSpeakReadURL = [thingSpeakURL 'channels/' channelID '/fields/field1/last'];
data = webread(thingSpeakReadURL,'api_key',writeApiKey)
```

```
data =

42
```

Write JSON Object

Write a number as a JSON object to a channel feed on the ThingSpeak server and read the response as a JSON object.

To run this code, create a ThingSpeak account. Call `webwrite` using the Write API key and Channel ID from your ThingSpeak account. Create a structure where the fields are your Write API Key and the number. The default field name is `'field1'`. Specify the media type as `'application/json'`.

```
thingSpeakURL = 'http://api.thingspeak.com/update.json';
writeApiKey = 'Your Write API Key';
data = 42;
data = struct('api_key',writeApiKey,'field1',data);
options = weboptions('MediaType','application/json');
response = webwrite(thingSpeakURL,data,options)
```

```
response =

    channel_id: Your Channel ID
        field1: 42
        field2: []
        field3: []
        field4: []
        field5: []
        field6: []
        field7: []
        field8: []
    created_at: '2014-11-14T20:08:14Z'
      entry_id: 1
        status: []
      latitude: []
     longitude: []
     elevation: []
      location: []
```

As a response `webwrite` receives a JSON object that contains the number you wrote to your ThingSpeak channel. `webwrite` converts the JSON object and returns it as a structure in `response`.

Specify Data and Time

Write a number and a specific date to a channel feed on the ThingSpeak server. Read the number and date back.

To run this code, create a ThingSpeak account. Call `webwrite` using the Write API key and Channel ID from your ThingSpeak account. Specify the date for the feed entry with a `datetime` object.

```
thingSpeakURL = 'http://api.thingspeak.com/';
thingSpeakWriteURL = [thingSpeakURL 'update'];
writeApiKey = 'Your Write API Key';
fieldName = 'field1';
fieldValue = 42;
D = datetime(2015,3,22,8,15,30,'Format','yyyy-MM-dd HH:mm:ss');
response = webwrite(thingSpeakWriteURL,'api_key',writeApiKey,...
    fieldName,fieldValue,'created_at',D)
```

If this call to `webwrite` is the first update to your ThingSpeak channel, `response` is `1`.

Read back the last entry to your channel. ThingSpeak provides a different URL to get the last entry to your channel. Append `last.json` to the URL to get the data as a JSON object. Your Channel ID is part of the URL.

```
channelID = num2str(Your Channel ID);
thingSpeakReadURL = [thingSpeakURL 'channels/' channelID '/fields/' ...
    fieldName '/last.json'];
data = webread(thingSpeakReadURL,'api_key',writeApiKey)
```

```
data =

    created_at: '2015-03-22T08:15:30Z'
      entry_id: 1
        field1: '42'
```

The date in the `created_at` field matches the date specified in `D`.

Input Arguments

`url` — URL to web service
character vector | string scalar

URL to a web service, specified as a character vector or string scalar. The web service implements a RESTful interface. See “RESTful” on page 1-16877 for more information.

`PostName1,PostValue1,...,PostNameN,PostValueN` — Web service post parameters
name-value pairs

Web service post parameters, specified as one or more pairs of name-value arguments. A `PostName` argument must specify the name of a post parameter. A `PostValue` argument must be a character vector, a string scalar, or a numeric, logical, or `datetime` value that specifies the value of the post parameter. Numeric, logical, and `datetime` values can be in arrays. The web service defines name-value pairs that it accepts as part of a request. `webwrite` encodes the name-value pairs as a form-encoded character array in the body of an HTTP POST request and sets the content type to `application/x-www-form-urlencoded` by default.

When you specify `PostValue` as a `datetime` object, you must specify its `Format` property so that it is consistent with the format required by the web service. If the `Format` property includes a time zone or offset, and the `datetime` object is not zoned, then `webwrite` specifies `'Local'` as the time zone.

When a `PostValue` argument contains multiple values in an array, specify the `ArrayFormat` property of a `weboptions` object to form-encode the array as specified by the web service.

Example: `webwrite('https://www.mathworks.com/matlabcentral/fileexchange/','term','webwrite','duration',7)` retrieves a list of files uploaded to the File Exchange within the past 7 days that contain the word `webwrite`. The File Exchange web service defines the `term` and `duration` parameters.

`data` — Data to write to web service
character vector | string scalar | ...

Data to post to a web service, specified as a character vector, a string scalar, or as numeric, cell, logical, or structure for `MediaType` value `'json'`, or as Document Object Model for `MediaType` value `'XML'`. If `data` is a character string or character vector, then `webwrite` sends it without conversion. All other types are converted based on the `weboptions.MediaType` value. For a complete list of media types, see Internet Media Types.

Example: `webwrite('https://www.mathworks.com/matlabcentral/fileexchange/','term=webwrite&duration=7')` retrieves a list of files uploaded to the File Exchange within the past 7 days that contain the word `webwrite`. The File Exchange web service defines the `term` and `duration` parameters.

`options` — Additional HTTP request options
`weboptions` object

Additional HTTP request options, specified as a `weboptions` object. See `weboptions` for all request options that are `weboptions` properties.

Output Arguments

`response` — Response from web service
scalar | array | structure | table

Response from a web service, returned as a scalar, array, structure, or table.

More About

RESTful

REST means *representational state transfer*, a common architectural style for web services. RESTful interfaces provide standard HTTP methods such as GET, PUT, POST, or DELETE.

Tips

- For functionality not supported by the RESTful web services functions, see the “HTTP Interface”.
- The `webwrite` function writes `PostName,PostValue` input arguments as form-encoded character arrays. If you also specify the `options` input argument, then its `MediaType` property must be `'application/x-www-form-urlencoded'`.
- `webwrite` cannot convert `datetime` objects to JSON, because JSON does not define a standard date format.
- `webwrite` always puts `PostName,PostValue` query parameters into the body of the message regardless of the value of the `RequestMethod` property of `options`.

See Also

`datetime` | `jsonencode` | `weboptions` | `webread` | `websave` | `xmlwrite`

Topics

“Represent Dates and Times in MATLAB”
“HTTP Interface”

External Websites

Representational State Transfer
JavaScript Object Notation (JSON)

Introduced in R2015a

week

Week number

Syntax

```
w = week(t)
w = week(t,weekType)
```

Description

`w = week(t)` returns the week-of-year numbers of the datetime values in `t`. The `m` output is a `double` array the same size as `t` and contains integer values from 1 to 53.

`w = week(t,weekType)` returns the type of week number specified by `weekType`.

Examples

Find Week of Year Numbers of Dates

```
t = datetime(2013,05,31):calmonths(3):datetime(2014,06,15)
```

```
t = 1x5 datetime array
   31-May-2013   31-Aug-2013   30-Nov-2013   28-Feb-2014   31-May-2014
```

```
w = week(t)
```

```
w = 1×5

    22    35    48     9    22
```

Input Arguments

t — Input date and time
`datetime` array

Input date and time, specified as a `datetime` array.

weekType — Type of week values
`'weekofyear'` (default) | `'weekofmonth'`

Type of week values, specified as either `'weekofyear'` or `'weekofmonth'`.

- If `weekType` is `'weekofyear'`, then `month` returns the week-of-year number. Week 1 is the week in which January 1 falls, even if fewer than 4 days of that week fall in the same year.
- If `weekType` is `'weekofmonth'`, then `month` returns the week-of-month number. Week 1 in a month is defined as the week in which the first day of the month falls, even if fewer than 4 days of that week fall in the same month.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`day` | `month` | `quarter` | `year`

Introduced in R2014b

weekday

Day of week

Syntax

```
DayNumber = weekday(D)
[DayNumber,DayName] = weekday(D)
[DayNumber,DayName] = weekday(D,DayForm)
[DayNumber,DayName] = weekday(D,language)
[DayNumber,DayName] = weekday(D,DayForm,language)
```

Description

`DayNumber = weekday(D)` returns a number representing the day of the week for each element in `D`.

`[DayNumber,DayName] = weekday(D)` additionally returns abbreviated English names for the day of the week, in `DayName`.

`[DayNumber,DayName] = weekday(D,DayForm)` returns the name for the day of the week in the format specified by `DayForm`, in US English.

`[DayNumber,DayName] = weekday(D,language)` returns the abbreviated name for the day of the week in the language of the locale specified in `language`.

`[DayNumber,DayName] = weekday(D,DayForm,language)` returns the name for the day of the week in the specified format and in the language of the specified locale. You can specify `DayForm` and `language` in either order.

Examples

Return Day of Week of Date String

Determine the day of the week of December 21, 2012.

```
D = '21-Dec-2012';
[DayNumber,DayName] = weekday(D)
```

```
DayNumber = 6

DayName =
'Fri'
```

December 21, 2012 falls on a Friday.

Return Full Day Names of Multiple Date Numbers

Return the full name of the day of the week for a vector of serial date numbers.

```
D = [734999;735015];
DayForm = 'long';
[DayNumber,DayName] = weekday(D,DayForm)
```

```
DayNumber = 2×1

     5
     7


DayName = 2x8 char array
    'Thursday'
    'Saturday'
```

Return Full Day Names in Local Language

Return a day name in U.S. English using the `language` input argument.

```
D = 728647;
DayForm = 'long';
language = 'en_US';
[DayNumber,DayName] = weekday(D,DayForm,language)
```

```
DayNumber = 2
```

```
DayName =
'Monday'
```

In U.S. English, the name of the day of the week is `Monday`.

Return day names in the language of the current locale.

```
language = 'local';
[DayNumber,DayName] = weekday(D,DayForm,language);
```

The value of `DayName` depends on the locale. For example, in a French locale, the name of the day of the week is `Lundi`.

Return Day of Week of Date String in Custom Format

Determine the day of the week for a date specified in the format `mmm.dd.yyyy`. Call `datenum` inside of `weekday` to specify the format of the input text representing a date.

```
[DayNumber,DayName] = weekday(datenum('Dec.21.2012','mmmm.dd.yyyy'))
```

```
DayNumber = 6

DayName =
'Fri'
```

Input Arguments

`D` — Serial date numbers or text representing dates and times
vector | matrix | character vector | cell array of character vectors | string array | character array

Serial date numbers or text representing dates and times. Date numbers can be specified as a vector or matrix. Text can be specified as a character vector, a cell array of character vectors, a string array, or a character array where each row represents a date. If `D` is a cell array of character vectors or a string array, then it must be 1-by-`n` or `n`-by-1.

If `D` is a character vector, a cell array of character vectors, a string array, or a character array, then the dates can be in one of the following formats.

Date Format	Example
dd-mmm-yyyy	01-Mar-2000
mm/dd/yyyy	03/01/2000
yyyy-mm-dd	2000-03-01

For text representing dates in other formats, first convert the dates to serial date numbers using the `datenum` function, before passing them to `weekday`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `cell` | `string`

`DayForm` — Format of output day names
`'short'` (default) | `'long'`

Format of the output day names, specified as one of the following values.

DayForm	Format of DayName Names	Example
'short'	Abbreviated name	Mon
'long'	Full name	Monday

`language` — Output language of day names
`'en_US'` (default) | `'local'`

Output language of day names in `DayName`, specified as one of the following values.

language	Description
'en_US'	US English
'local'	Language of the current locale

Output Arguments

`DayNumber` — Value representing day of week
array of integers in the range `[1,7]`

Value representing the day of the week, returned as an array of integers in the range `[1,7]`, where 1 represents Sunday, and 7 represents Saturday.

- If input `D` is a numeric array, then the size of `DayNumber` is equivalent to the size of `D`.
- If input `D` is a cell array of character vectors, then `DayNumber` is an `m`-by-1 vector, where `m` is equivalent to the length of `D`.

`DayName` — Name of day of week
character array

Name of the day of the week, returned as a character array. The content of `DayName` depends on `DayForm`.

- If `DayForm` is `'short'`, then `DayName` contains an abbreviated name (for example, `Tues`).
- If `DayForm` is `'long'`, then `DayName` contains the full name of the weekday (for example, `Tuesday`).

`DayName` is `m`-by-`n`, where `m` is the number of dates represented in `D`.

See Also

`datenum` | `datevec` | `eomday`

Introduced before R2006a

what

List MATLAB files in folder

Syntax

```
what
what folderName
s = what( ___ )
```

Description

`what` lists the path for the current folder and all the MATLAB relevant files and folders found in the current folder. This includes MATLAB program files (`.m` and `.mlx`), MAT-files, Simulink model files (`.mdl` and `.slx`), MEX-files, MATLAB app files (`.mlapp`), and P-files, as well as all class and package folders.

`what folderName` lists path, file, and folder information for `folderName`. You do not need to specify the `@` or `+` symbol for class and package folders. For example, `what table` lists the MATLAB files and folders in `toolbox/matlab/datatypes/@table`.

`s = what( ___ )` returns the results in a structure array. You can use this syntax with any of the input argument combinations in the previous syntaxes.

Examples

List Path, File, and Folder Information

List the MATLAB files and folders in the `graph2d` folder. The location of this folder depends on your installation.

```
what graph2d
```

```
MATLAB Code files in folder matlabroot\toolbox\matlab\graph2d

Contents              figtoolset            loglog                putdowntext           setscribeobjectdata
axis                  getcolumn             middrag               rbbox                 sgtitle
```

```
basicfitdatastat       getobj                 moveaxis               scribeclearmode        subplot
box                    getorcreateobj         pan                    scribeeventhandler     texlabel
doclick                getscribecontextmenu   plot                   scriberestoresavefcns  title
dokeypress             getscribeobjectdata    plotedit               scribetextdlg          xlabel
domymenu               grid                   plotyy                 semilogx               ylabel
doresize               gtext                  polar                  semilogy               zoom
enddrag                jpropeditutils         prepdrag               setscribecontextmenu

P-files in folder matlabroot\toolbox\matlab\graph2d

axis     sgtitle  subplot  zoom

Classes in folder matlabroot\toolbox\matlab\graph2d

arrowline     axisobj       editline      fighandle     hgbin         scribehgobj
axischild     axistext      editrect      figobj        scribehandle
```

Find Supporting Files for Toolbox Package

Find the Fixed-Point Designer™ folders that contain MATLAB files.

```
s = what('fixpoint')
```

```
s =

  struct with fields:

        path: 'matlabroot\toolbox\fixpoint'
           m: {4×1 cell}
       mlapp: {0×1 cell}
         mlx: {0×1 cell}
         mat: {0×1 cell}
         mex: {0×1 cell}
         mdl: {0×1 cell}
         slx: {0×1 cell}
         sfx: {0×1 cell}
           p: {4×1 cell}
     classes: {0×1 cell}
    packages: {5×1 cell}
```

List the packages in the *matlabroot*`\toolbox\fixpoint` folder.

```
s.packages
```

```
ans =

  5×1 cell array

    {'DataTypeOptimization' }
    {'DataTypeWorkflow'     }
    {'FunctionApproximation'}
    {'IRInstrumentation'    }
    {'cpopt'                }
```

Find the supporting files for the `DataTypeWorkflow` package.

```
s2 = what('DataTypeWorkflow');
s2.p
```

```
ans =

  6×1 cell array

    {'DTWRun.p'                 }
    {'RangeCollectionMode.p'    }
    {'Result.p'                 }
    {'VerificationResult.p'     }
    {'findDecoupledSubsystems.p'}
    {'hyperlink.p'              }
```

Input Arguments

`folderName` — Name of folder
character vector | string scalar

Name of folder, specified as a character vector or string scalar.

For a local folder, it is not necessary to give the full path of the folder. Instead, you can specify a relative partial path. For example, `what strfun` and `what matlab/strfun` both list the MATLAB files in `toolbox/matlab/strfun`. However, if the folder is at a remote location, then `folderName` must contain a full path specified as a uniform resource locator (URL). For more information, see “Work with Remote Data”.

The case sensitivity of `folderName` is determined by your file system. In general, Windows and Mac systems are case insensitive and therefore, `exist` returns results for all case variations of `folderName`. For example, suppose the folder `c:\mycode` containing the file `myfile.m` exists on a Windows file system. `what` successfully locates the folder even if the incorrect case is specified.

```
what MYCodE
```

```
MATLAB Code files in folder c:\mycode
myfile
```

On Linux systems, which are generally case sensitive, what returns zero results.

Output Arguments

s — List of path, files, and folders
structure array

List of path, files, and folders returned as a structure array with these fields.

Field	Description
`path`	Full path to folder
`m`	Cell array of MATLAB program file names
`mlapp`	Cell array of MLAPP-file names
`mlx`	Cell array of MLX-file names
`mat`	Cell array of MAT-file names
`mex`	Cell array of MEX-file names
`mdl`	Cell array of MDL-file names
`slx`	Cell array of SLX-file names
`sfx`	Cell array of SFX-file names
`p`	Cell array of P-file names
`classes`	Cell array of class folders
`packages`	Cell array of package folders

See Also

`dir` | `exist` | `lookfor` | `ls` | `which` | `who`

Topics

"Work with Remote Data"

Introduced before R2006a

whatsnew

Release Notes

Note `whatsnew` will be removed in a future release.

Syntax

```
whatsnew
```

Description

`whatsnew` displays the MATLAB Release Notes in the Help browser, presenting information about new features, problems from previous releases that have been fixed in the current release, and compatibility issues.

See Also

`help` | `version`

Introduced before R2006a

which

Locate functions and files

Syntax

```
which item
which fun1 in fun2

which ___ -all
which ___ -all

str = which(item)
str = which(fun1,'in',fun2)

str = which( ___ ,'-all')
```

Description

`which item` displays the full path for `item`.

- If `item` is a MATLAB function in a MATLAB code file (`.m`,`.mlx`, or `.p` extension), or a saved Simulink model (`.slx` or `.mdl` extension), then `which` displays the full path for the corresponding file. `item` must be on the MATLAB path.
- If `item` is a method in a loaded Java class, then `which` displays the package, class, and method name for that method.
- If `item` is a workspace variable, then `which` displays a message identifying `item` as a variable.
- If `item` is an unsaved Simulink model that is loaded in Simulink, then `which` displays a message identifying `item` as a new Simulink model.
- If `item` is a file name including the extension, and it is in the current working folder or on the MATLAB path, then `which` displays the full path of `item`.

If `item` is an overloaded function or method, then `which item` returns only the path of the first function or method found.

`which fun1 in fun2` displays the path to function `fun1` that is called by file `fun2`. Use this syntax to determine whether a local function is being called instead of a function on the path. This syntax does not locate nested functions.

`which ___ -all` displays the paths to all items on the MATLAB path with the requested name. Such items include methods of instantiated classes. `which ___ -all` also displays any files in special folders that have been implicitly added to the path. For more information about these special folders, see What Is the MATLAB Search Path. You can use `-all` with the input arguments of any of the previous syntaxes.

`str = which(item)` returns the full path for `item` to `str`.

`str = which(fun1,'in',fun2)` returns the path to function `fun1` that is called by file `fun2`. Use this syntax to determine whether a local function is being called instead of a function on the path. This syntax does not locate nested functions.

`str = which(___,'-all')` returns the results of `which` to `str`. You can use this syntax with any of the input arguments in the previous syntax group.

Examples

Locate MATLAB Function

Locate the `pinv` function.

```
which pinv
```

```
matlabroot\toolbox\matlab\matfun\pinv.m
```

`pinv` is in the `matfun` folder of MATLAB.

You also can use function syntax to return the path to `str`. When using the function form of `which`, enclose all input arguments in single quotes.

```
str = which('pinv');
```

Locate Method in a Loaded Java Class

Create an instance of the Java® class. This loads the class into MATLAB®.

```
myDate = java.util.Date;
```

Locate the `setMonth` method.

```
which setMonth
```

```
setMonth is a Java method  % java.util.Date method
```

Locate Private Function

Find the `orthog` function in a private folder.

```
which private/orthog
```

```
matlabroot\toolbox\matlab\elmat\private\orthog.m  % Private to elmat
```

MATLAB displays the path for `orthog.m` in the `/private` subfolder of `toolbox/matlab/elmat`.

Determine If Local Function Is Called

Determine which `parseargs` function is called by `area.m`.

```
which parseargs in area
```

```
% Local function of area
matlabroot\toolbox\matlab\specgraph\area.m (parseargs)
```

You also can use function syntax to return the path to `str`. When using the function form of `which`, enclose all input arguments in single quotes.

```
str = which('parseargs','in','area');
```

Locate Function Invoked with Given Input Arguments

Suppose that you have a `matlab.io.MatFile` object that corresponds to the example MAT-file `'topography.mat'`:

```
matObj = matfile('topography.mat');
```

Display the path of the implementation of `who` that is invoked when called with the input argument `(matObj)`.

```
which who(matObj)
```

```
% matlab.io.MatFile method
matlabroot\toolbox\matlab\iofun\+matlab\+io\MatFile.m
```

Store the result to the variable `str`.

```
str = which('who(matObj)')
```

```
str =
matlabroot\toolbox\matlab\iofun\+matlab\+io\MatFile.m
```

If you do not specify the input argument `(matObj)`, then `which` returns only the path of the first function or method found.

```
which who
```

```
built-in (matlabroot\\toolbox\matlab\general\who)
```

Locate All Items with Given Name

Display the paths to all items on the MATLAB path with the name `fopen`.

```
which fopen -all
```

```
built-in (matlabroot\toolbox\matlab\iofun\fopen)
% serial method
matlabroot\toolbox\matlab\iofun\@serial\fopen.m

% icinterface method
matlabroot\toolbox\shared\instrument\@icinterface\fopen.m
matlabroot\toolbox\instrument\instrument\@i2c\fopen.m
```

Return Path Names

Return the results of `which` to `str`.

Find the `orthog` function in a private folder. You must use the function form of `which`, enclosing all arguments in parentheses and single quotes.

```
str = which('private/orthog','-all');
whos str
```

```
  Name      Size            Bytes  Class     Attributes

  str       1x1               206  cell
```

Input Arguments

item — Function or file to locate

character vector | string scalar

Function or file to locate, specified as a character vector or string scalar. When using the function form of `which`, enclose all `item` inputs in single or double quotes. `item` can be in one of the following forms.

Form of the `item` Input	Path to Display
fun	Display full path for `fun`, which can be a MATLAB function, Simulink model, workspace variable, method in a loaded Java class, or file name that includes the file extension. To display the path for a file that has no file extension, type `which file.` (The period following the file name is required). Use `exist` to check for the existence of files anywhere else.
/fun	Limit the search to functions named `fun` that are on the search path. For example, `which /myfunction` displays the full path for function `myfunction.m`, but not built-in or JAVA functions with the same name.
`private/`*fun*	Limit the search to private functions named `fun`. For example, `which private/orthog` or `which('private/orthog')` displays the path for `orthog.m` in the `/private` subfolder of the parent folder.

Form of the `item` Input	Path to Display
`fun(a1,...,an)`	Display the path to the implementation of function `fun` which would be invoked if called with the input arguments `a1,...,an`. Use this syntax to query overloaded functions. See the example, "Locate Function Invoked with Given Input Arguments" on page 1-16893.

Data Types: `char` | `string`

`fun1` — Function to locate
character vector | string scalar

Function to locate, specified as a character vector or string scalar. `fun1` can be the name of a function, or it can be in the form `fun(a1,...,an)`. For more information about the form, `fun(a1,...,an)`, see "Locate Function Invoked with Given Input Arguments" on page 1-16893.

When using the function form of `which`, enclose all `fun1` inputs in single or double quotes, for example, `which('myfun1','in','myfun2')`.

Data Types: `char` | `string`

`fun2` — Calling file
character vector | string scalar

Calling file, specified as a character vector or string scalar. `fun2` can be the name of a file, or it can be in the form `fun(a1,...,an)`. For more information about the form, `fun(a1,...,an)`, see "Locate Function Invoked with Given Input Arguments" on page 1-16893.

When using the function form of `which`, enclose all `fun2` inputs in single or double quotes, for example, `which('myfun1','in','myfun2')`.

Data Types: `char` | `string`

Output Arguments

`str` — Function or file location
character vector | cell array of character vectors

Function or file location, returned as a character vector or cell array of character vectors if you use `'-all'`.

- If `item` is a workspace variable, then `str` is the character vector `'variable'`.
- If `str` is a cell array of character vectors, then each row of `str` identifies a result of `which`. The results are ordered according to the "Function Precedence Order". If there are shadowed results, you should not rely on the order of the shadowed functions and methods in `str`. To determine if a result is shadowed, call `which` without specifying `str`. `which` indicates shadowed results by the comment `% Shadowed`.

Limitations

- When the class is not loaded, `which` only finds methods if they are defined in separate files in an @-folder and are not in any packages.

Tips

- For more information about how MATLAB uses scope and precedence when calling a function, see "Function Precedence Order".

See Also

`dir` | `doc` | `exist` | `fileparts` | `lookfor` | `mfilename` | `path` | `type` | `what` | `who`

Topics

"What Is the MATLAB Search Path?"

Introduced before R2006a

while

`while` loop to repeat when condition is true

Syntax

```
while expression
    statements
end
```

Description

`while` *`expression`*`,` *`statements`*`, end` evaluates an expression on page 1-16900, and repeats the execution of a group of statements in a loop while the expression is true. An expression is true when its result is nonempty and contains only nonzero elements (logical or real numeric). Otherwise, the expression is false.

Examples

Repeat Statements Until Expression Is False

Use a `while` loop to calculate `factorial(10)`.

```
n = 10;
f = n;
while n > 1
    n = n-1;
    f = f*n;
end
disp(['n! = ' num2str(f)])
```

```
n! = 3628800
```

Skip to Next Loop Iteration

Count the number of lines of code in the file `magic.m`. Skip blank lines and comments using a `continue` statement. `continue` skips the remaining instructions in the `while` loop and begins the next iteration.

```
fid = fopen('magic.m','r');
count = 0;
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line) || strncmp(line,'%',1) || ~ischar(line)
        continue
    end
    count = count + 1;
end
count
```

```
count = 31
```

```
fclose(fid);
```

Exit Loop Before Expression Is False

Sum a sequence of random numbers until the next random number is greater than an upper limit. Then, exit the loop using a `break` statement.

```
limit = 0.8;
s = 0;

while 1
    tmp = rand;
    if tmp > limit
        break
    end
    s = s + tmp;
end
```

More About

Expression

An expression can include relational operators (such as < or ==) and logical operators (such as &&, ||, or ~). Use the logical operators `and` and `or` to create compound expressions. MATLAB evaluates compound expressions from left to right, adhering to operator precedence rules.

Within the conditional expression of a `while...end` block, logical operators & and | behave as short-circuit operators. This behavior is the same as && and ||, respectively. Since && and || consistently short-circuit in conditional expressions and statements, it is good practice to use && and || instead of & and | within the expression. For example,

```
x = 42;
while exist('myfunction.m','file') && (myfunction(x) >= pi)
    disp('Expressions are true')
    break
end
```

The first part of the expression evaluates to false. Therefore, MATLAB does not need to evaluate the second part of the expression, which would result in an undefined function error.

Tips

- If you inadvertently create an infinite loop (that is, a loop that never ends on its own), stop execution of the loop by pressing **Ctrl+C**.
- If the conditional expression evaluates to a matrix, MATLAB evaluates the statements only if all elements in the matrix are true (nonzero). To execute statements if any element is true, wrap the expression in the `any` function.
- To programmatically exit the loop, use a `break` statement. To skip the rest of the instructions in the loop and begin the next iteration, use a `continue` statement.
- When nesting a number of `while` statements, each `while` statement requires an `end` keyword.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`Logical Operators: Short Circuit` | `break` | `continue` | `end` | `for` | `if` | `return` | `switch`

Topics

"Array Comparison with Relational Operators"

Introduced before R2006a

whitebg

Change axes background color

Note `whitebg` is not recommended.

Syntax

```
whitebg
whitebg(fig)
whitebg(c)
whitebg(fig,c)
whitebg(fig,c)
whitebg(fig)
```

Description

`whitebg` complements the colors in the current figure.

`whitebg(fig)` complements colors in all figures specified in the vector `fig`.

`whitebg(c)` and `whitebg(fig,c)` change the color of the axes, which are children of the figure, to the color specified by `c`. The value of `c` can be a color name or an RGB triplet. The possible color names are: `'red'`, `'green'`, `'blue'`, `'cyan'`, `'magenta'`, `'yellow'`, `'black'`, or `'white'`. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

`whitebg(fig,c)` sets the default axes background color of the figures in the vector `fig` to the color specified by `c`. Other axes properties and the figure background color can change as well, so that graphs maintain adequate contrast.

`whitebg(fig)` complements the colors of the objects in the specified figures. This syntax is typically used to toggle between black and white axes background colors, and is where `whitebg` gets its name. Include the root window handle `(0)` in `fig` to affect the default properties for new windows or for `clf reset`.

Examples

Set the background color to blue-gray.

```
whitebg([0 .5 .6])
```

Set the background color to blue.

```
whitebg('blue')
```

Tips

`whitebg` works best in cases where all the axes in the figure have the same background color.

Without a figure specification, `whitebg` or `whitebg(c)` affects the current figure and the root's default properties so subsequent plots and new figures use the new colors.

`whitebg` changes the colors of the figure's children, with the exception of shaded surfaces. This ensures that all objects are visible against the new background color. `whitebg` sets the default properties on the root such that all subsequent figures use the new background color.

See Also

`colordef`

Introduced before R2006a

who

List variables in workspace

Syntax

```
who
who -file filename
who global
who ___ var1 ... varN
who ___ -regexp expr1 ... exprN
C = who( ___ )
```

Description

`who` lists in alphabetical order the names of all variables in the currently active workspace.

`who -file filename` lists the variable names in the specified MAT-file.

`who global` lists the variable names in the global workspace.

`who ___ var1 ... varN` lists only the specified variables. Use this syntax with any of the arguments in the previous syntaxes.

`who ___ -regexp expr1 ... exprN` lists only the variables that match the specified regular expressions.

`C = who( ___ )` stores the names of the variables in the cell array `C`.

Note You must use the functional form of `who` when there is an output argument.

Examples

Display Workspace Variable Names

List the names of variables in the current workspace that start with the letter `a`.

```
who a*
```

Display the names of variables in the current workspace that end with `ion`.

```
who -regexp ion$
```

List Names of Variables in a MAT-File

List the names of variables stored in the sample MAT-file `durer.mat`.

```
who -file durer.mat
```

```
Your variables are:

X        caption  map
```

Store Variable List in Cell Array

Store the list of variable names in `durer.mat` in cell array `C`.

```
C = who('-file','durer.mat');
```

Display the contents of `C`.

```
for k=1:length(C)
   disp(C{k})
end
```

```
X
caption
map
```

List Workspace Variables Within Nested or Anonymous Function

List all the variable names in the current workspace while paused in a nested function.

Create a file `who_demo.m`, that contains these statements.

```
function who_demo
date_time = datestr(now,'dd-mmm-yyyy');

date_time_array = strsplit(date_time,{'-',''});
get_date(date_time_array);

   function get_date(d)
      day = d{1};  %#ok<*NASGU>
      mon = d{2};
      year = d{3};
      keyboard
   end

end
```

```
K>>
```

Run `who_demo`. MATLAB pauses at the line with the `keyboard` command.

```
who_demo
```

Call the `who` function. MATLAB displays the names of the variables in the nested `get_date` function and in all functions containing the nested function.

```
K>> who

Your variables are:

d                 mon               date_time
day               year              date_time_array
```

Input Arguments

`var1 ... varN` — Variables to display
character vectors | string scalars

Variables to display, specified as one or more character vectors or string scalars. Use the `'*'` wildcard to match patterns. For example, `who A* S*` lists the names of all the variables in the workspace that start with `A` or `S`.

`expr1 ... exprN` — Regular expressions
character vectors | string scalars

Regular expressions that define the variables to display, specified as one or more character vectors or string scalars. For example, `who -regexp ^Mon ^Tues` lists only the variable names in the workspace that begin with `Mon` or `Tues`. For more information about creating regular expressions, see “Regular Expressions”.

`filename` — Name of MAT-file
character vector | string scalar

Name of MAT-file, specified as a character vector or string scalar. The file name can include the full, relative, or partial path. For example, `who -file myFile.mat` lists the names of all variables in the MAT-file named `myFile.mat`.

Data Types: `char` | `string`

Output Arguments

`C` — List of variables
cell array of character vector

List of variables, specified as a cell array of character vector.

Alternatives

- To view the variables in the workspace, use the Workspace browser. To view the contents of MAT-files, use the Details Panel of the Current Folder browser.

See Also

Workspace Browser | `assignin` | `clear` | `clearvars` | `exist` | `inmem` | `load` | `save` | `what`

Topics

"Create and Edit Variables"

Introduced before R2006a

whos

List variables in workspace, with sizes and types

Syntax

```
whos
whos -file filename
whos global
whos ___ var1 ... varN
whos ___ -regexp expr1 ... exprN
S = whos( ___ )
```

Description

`whos` lists in alphabetical order the names, sizes, and types of all variables in the currently active workspace.

`whos -file filename` lists variables in the specified MAT-file.

`whos global` lists variables in the global workspace.

`whos ___ var1 ... varN` lists only the specified variables. Use this syntax with any of the arguments in the previous syntaxes.

`whos ___ -regexp expr1 ... exprN` lists only the variables that match the specified regular expressions.

`S = whos( ___ )` stores information about the variables in the structure array `S`.

Note You must use the functional form of `whos` when there is an output argument.

Examples

Display Workspace Variable Information

Display information about specific variables in the current workspace. For example, list information about variables with names that start with the letter `a`.

```
whos a*
```

Now, list information about variables with names that end with `ion`.

```
whos -regexp ion$
```

Display Information on Variables Stored in a MAT-File

Display all the information on the variables stored in the sample MAT-file `durer.mat`.

```
whos -file durer.mat
```

```
  Name           Size                Bytes  Class     Attributes

  X            648x509            2638656  double
  caption        2x28                 112  char
  map          128x3                 3072  double
```

Store Variable Information in a Structure Array

Store information about the variables in `durer.mat` in structure array `S`.

```
S = whos('-file','durer.mat');
```

Display the contents of `S`.

```
for k = 1:length(S)
   disp(['  ' S(k).name ...
         '  ' mat2str(S(k).size) ...
         '  ' S(k).class]);
end
```

```
  X  [648 509]  double
  caption  [2 28]  char
  map  [128 3]  double
```

Display Variable Attribute Information

Create variables with various attributes, and then display information about them.

Create a file, `show_attributes.m`, that contains these statements.

```
function show_attributes
persistent p;
global g;
p = 1;
g = 2;
s = sparse(eye(5));
c = [4+5i 9-3i 7+6i];
whos
```

Call `show_attributes`. When MATLAB® executes the `whos` command at the end of `show_attributes`, it lists each variable and its corresponding attribute.

```
show_attributes
```

```
  Name      Size            Bytes  Class     Attributes

  c         1x3                48  double    complex
  g         1x1                 8  double    global
  p         1x1                 8  double    persistent
  s         5x5               128  double    sparse
```

List Workspace Variables Within Nested or Anonymous Function

List all the variables in the current workspace while paused in a nested function.

Create a file, `whos_demo.m`, that contains these statements.

```
function whos_demo
date_time = datestr(now,'dd-mmm-yyyy');

date_time_array = strsplit(date_time,{'-',''});
```

```
get_date(date_time_array);

   function get_date(d)
      day = d{1};  %#ok<*NASGU>
      mon = d{2};
      year = d{3};
      keyboard
   end

end
```

Run `whos_demo`. MATLAB pauses at the line with the `keyboard` command.

```
whos_demo
```

```
K>>
```

Call the `whos` function. MATLAB displays the variables in the nested `get_date` function, and the variables in all functions that contain the nested function, grouped by function workspace.

```
K>> whos
```

```
Name                  Size            Bytes  Class    Attributes

  ---- whos_demo/get_date ----------------------------------------
  d                     1x3               354  cell
  day                   1x2                 4  char
  mon                   1x3                 6  char
  year                  1x4                 8  char

  ---- whos_demo -------------------------------------------------
  date_time             1x11               22  char
  date_time_array       1x3               354  cell
```

Input Arguments

`var1 ... varN` — Variables to display

character vectors | string scalars

Variables to display, specified as one or more character vectors or string scalars. Use the `'*'` wildcard to match patterns. For example, `whos A* S*` lists the names of all the variables in the workspace that start with `A` or `S`.

`expr1 ... exprN` — Regular expressions
character vectors | string scalars

Regular expressions that define the variables to display, specified as one or more character vectors or string scalars. For example, `whos -regexp ^Mon ^Tues` lists only the variable names in the workspace that begin with `Mon` or `Tues`. For more information about creating regular expressions, see “Regular Expressions”.

`filename` — Name of MAT-file
character vector | string scalar

Name of MAT-file, specified as a character vector or string scalar. The file name can include the full, relative, or partial path. For example, `whos -file myFile.mat` lists all variables in the MAT-file named `myFile.mat`. The `whos -file filename` command does not return the sizes of any MATLAB objects in file *`filename`*.

Data Types: `char` | `string`

Output Arguments

S — Variable information
nested structure array

Variable information, returned as a nested structure array containing a scalar `struct` for each variable. Each scalar `struct` contains these fields.

Field	Description
`name`	Name of the variable.
`size`	Dimensions of the variable array.

Field	Description
`bytes`	Number of bytes allocated for the variable array. `whos` returns the number of bytes each variable occupies in the workspace, which is not necessarily the same as the number of bytes each variable occupies in a MAT-file. MAT-files Version 7 and later are compressed, so the number of bytes required in the workspace is typically larger than the number of bytes in the file. `whos` does not report the number of bytes consumed by handle objects. If a variable contains handle objects, the number of bytes that the `whos` function displays for the variable might be smaller than expected.
`class`	Class of the variable. If the variable has no value, class is `'(unassigned)'`.
`global`	`true` if the variable is global.
`sparse`	`true` if the variable is sparse.
`complex`	`true` if the variable is complex.
`nesting`	Structure with these fields: • `function` — Name of the nested or outer function that defines the variable. • `level` — Nesting level of that function.
`persistent`	`true` if the variable is persistent.

Alternatives

- You also can view the contents of MAT-files using the Details Panel of the Current Folder browser.

See Also

`clear` | `exist` | `what` | `who`

Topics

“Create and Edit Variables”

Introduced before R2006a

width

Number of table variables

Syntax

```
W = width(T)
```

Description

`W = width(T)` returns the number of variables in table `T`.

`width(T)` is equivalent to `size(T,2)`.

Examples

Number of Variables in Table

Create a table, `T`.

```
LastName = {'Smith';'Johnson';'Williams';'Jones';'Brown'};
Age = [38;43;38;40;49];
Height = [71;69;64;67;64];
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];

T = table(Age,Height,Weight,BloodPressure,'RowNames',LastName)
```

```
T=5×4 table
                Age    Height    Weight    BloodPressure
                ___    ______    ______    _____________

    Smith       38       71       176       124     93
    Johnson     43       69       163       109     77
    Williams    38       64       131       125     83
    Jones       40       67       133       117     75
```

```
    Brown       49       64       119       122     80
```

Find the number of variables in table `T`.

```
W = width(T)
```

```
W = 4
```

`T` contains `4` variables; `width` does not count the row names.

The variable `BloodPressure` counts as one variable even though it contains two columns.

Input Arguments

`T` — Input table
table

Input table, specified as a table.

Variables in a table can have multiple columns, but `width(T)` only counts the number of variables.

Extended Capabilities

Tall Arrays
Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

This function fully supports code generation. For more information, see “Code Generation for Tables” (MATLAB Coder).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`height` | `numel` | `size`

Introduced in R2013b

wilkinson

Wilkinson's eigenvalue test matrix

Syntax

```
W = wilkinson(n)
W = wilkinson(n,classname)
```

Description

`W = wilkinson(n)` returns one of J. H. Wilkinson's `n`-by-`n` eigenvalue test matrices. `W` is a symmetric, tridiagonal matrix with pairs of nearly equal eigenvalues.

`W = wilkinson(n,classname)` returns a matrix of class `classname`, which can be `'single'` or `'double'`.

Examples

Wilkinson Test Matrix

Compute a 7-by-7 Wilkinson eigenvalue test matrix. The most frequently used case is `wilkinson(21)`, whose two largest eigenvalues are approximately 10.746. The eigenvalues agree to 14, but not 15, decimal places.

```
W = wilkinson(7)
```

```
W = 7×7

     3     1     0     0     0     0     0
     1     2     1     0     0     0     0
     0     1     1     1     0     0     0
     0     0     1     0     1     0     0
     0     0     0     1     1     1     0
     0     0     0     0     1     2     1
```

```
     0     0     0     0     0     1     3
```

Input Arguments

n — Matrix order
scalar, nonnegative integer

Matrix order, specified as a scalar, nonnegative integer.

Example: `wilkinson(10)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`

`classname` — Matrix class
`'double'` (default) | `'single'`

Matrix class, specified as either `'double'` or `'single'`.

Example: `wilkinson(10,'single')`

Data Types: `char`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `n` must be a fixed-size scalar.

See Also

`eig` | `gallery` | `pascal`

Introduced before R2006a

winopen

Open file in appropriate application (Windows)

Syntax

```
winopen name
```

Description

`winopen name` opens the specified file in the Microsoft Windows application associated with the extension of the file. `winopen` uses a Windows shell command, and performs the same action as double-clicking the file in Windows Explorer.

Examples

Open File in Microsoft® Word®

Open the file `thesis.docx` in Microsoft Word.

```
winopen thesis.docx
```

Open File in System Web Browser

Open the file `my_sine_wave.html` in the system Web browser.

```
winopen my_sine_wave.html
```

Open Current Folder in Windows Explorer Tool

```
winopen(cd)
```

Input Arguments

name — Name of file
character vector

Name of file to open, specified as a character vector. `name` can be an absolute or relative path.

See Also

`dos` | `open` | `web`

Introduced before R2006a

winqueryreg

Item from Windows registry

Syntax

```
valnames = winqueryreg('name',rootkey,subkey)
value = winqueryreg(rootkey,subkey,valname)
value = winqueryreg(rootkey,subkey)
```

Description

`valnames = winqueryreg('name',rootkey,subkey)` returns all value names in `rootkey\subkey` of Microsoft Windows operating system registry. The first argument is the literal, quoted, case-sensitive string or character vector `'name'`.

`value = winqueryreg(rootkey,subkey,valname)` returns the value for `valname` in `rootkey\subkey`.

`value = winqueryreg(rootkey,subkey)` returns a value in `rootkey\subkey` that has no value name property.

Examples

Display `Control Panel\Mouse` Values

Display `Control Panel\Mouse` values.

Get a list in variable `mousechar` for registry subkey `Mouse`, which is under subkey `Control Panel`, which is under root key `HKEY_CURRENT_USER`.

```
mousechar = winqueryreg('name','HKEY_CURRENT_USER','control panel\mouse');
```

For each name in the `mousechar` list, get its value from the registry and then display the name and its value:

```
for k = 1:length(mousechar)
   setting = winqueryreg('HKEY_CURRENT_USER','control panel\mouse',mousechar{k});
   str = sprintf('%s = %s',mousechar{k},num2str(setting));
   disp(str)
end
```

```
mousechar =
    'ActiveWindowTracking'
    'Beep'
    'DoubleClickHeight'
    'DoubleClickSpeed'
    'DoubleClickWidth'
    'ExtendedSounds'
    'MouseHoverHeight'
    'MouseHoverTime'
    'MouseHoverWidth'
    'MouseSensitivity'
    'MouseSpeed'
    'MouseThreshold1'
    'MouseThreshold2'
    'MouseTrails'
    'SmoothMouseXCurve'
    'SmoothMouseYCurve'
    'SnapToDefaultButton'
    'SwapMouseButtons'
```

Input Arguments

`rootkey` — Root key name
string | character vector

Root key name, specified as a case-sensitive string or character vector.

Example: `'HKEY_CLASSES_ROOT'`

`subkey` — Subkey name
string | character vector

Subkey name, specified as a string or a character vector. The value is not case-sensitive.

Example: `'control panel\mouse'`

`valname` — Name of value
string | character vector

Name of value, specified as a string or a character vector. The name is not case-sensitive.

Output Arguments

`valnames` — Value names
cell array of character vectors

Value names, specified as a cell array of character vectors.

`value` — Value of specified name
character vector | `int32`

Value of specified name, specified as a character vector if the value retrieved from the registry is a registry string value. If the value is a 32-bit integer, `winqueryreg` returns the value as an integer of the MATLAB type `int32`.

Limitations

- This function works for the following registry value types only:
  - strings (REG_SZ)
  - expanded strings (REG_EXPAND_SZ)
  - 32-bit integer (REG_DWORD)

See Also

Introduced before R2006a

winter

Winter colormap array

Syntax

```
c = winter
c = winter(m)
```

Description

`c = winter` returns the winter colormap as a three-column array with the same number of rows as the colormap for the current figure. If no figure exists, then the number of rows is equal to the default length of 256. Each row in the array contains the red, green, and blue intensities for a specific color. The intensities are in the range [0,1], and the color scheme looks like this image.

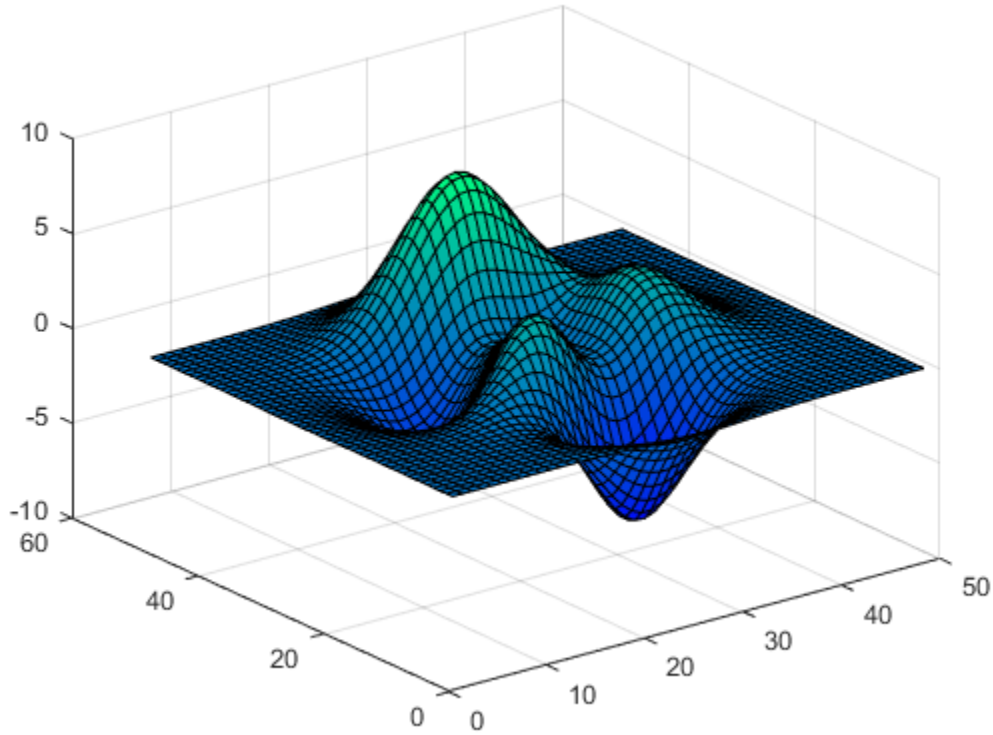
`c = winter(m)` returns the colormap with `m` colors.

Examples

Reverse the Colormap

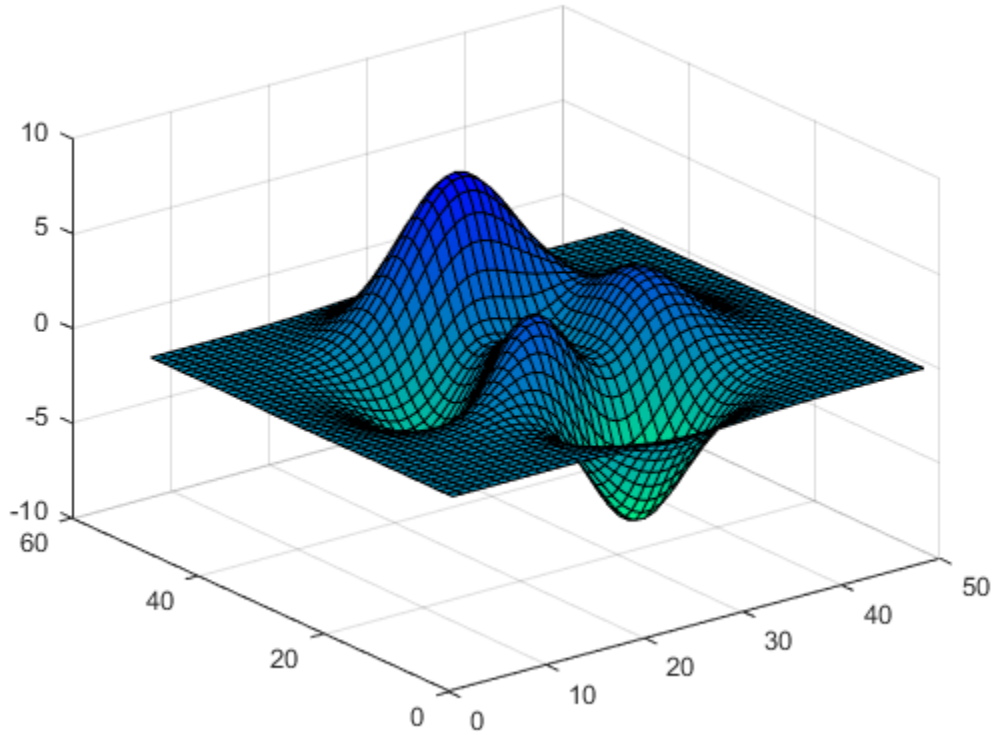
Plot a surface and assign the winter colormap.

```
surf(peaks);
colormap('winter');
```



Get the winter colormap array and reverse the order. Then apply the modified colormap to the surface.

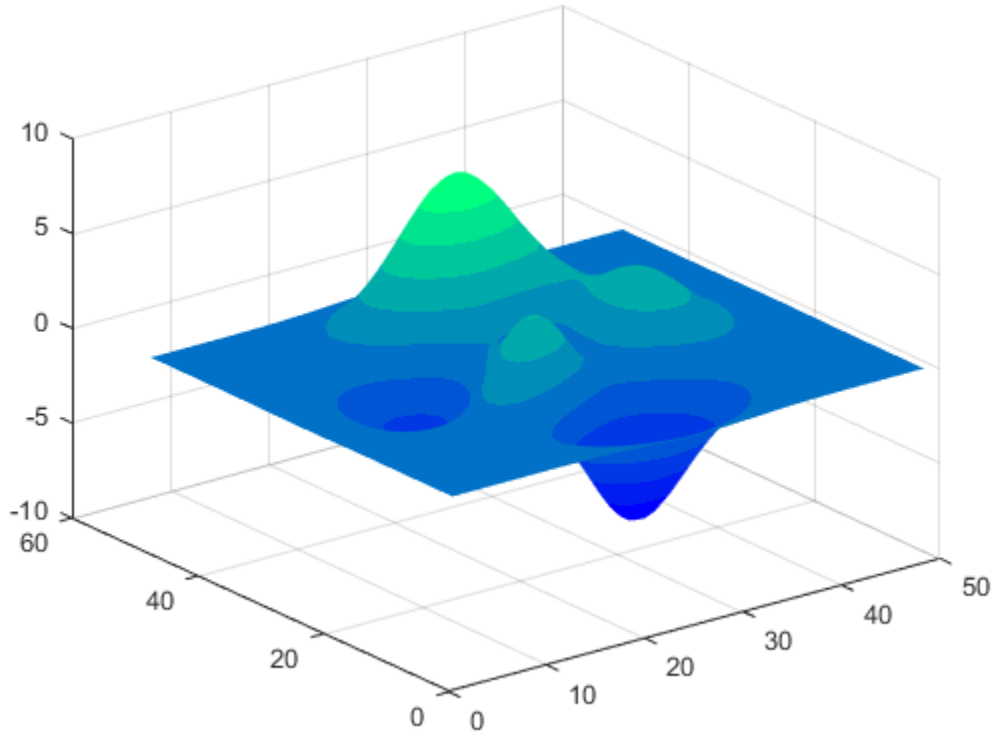
```
c = winter;
c = flipud(c);
colormap(c);
```



Downsample the Winter Colormap

Get a downsampled version of the winter colormap containing only ten colors. Then display the contours of the peaks function by applying the colormap and interpolated shading.

```
c = winter(10);
surf(peaks);
colormap(c);
shading interp;
```



Input Arguments

m — Number of colors
256 (default) | scalar integer value

Number of colors, specified as a scalar integer value. The default value of `m` is equal to the length of the colormap for the current figure. If no figure exists, the default value is 256.

Data Types: `single` | `double`

Compatibility Considerations

Colormaps Have 256 Colors by Default

Behavior changed in R2019b

Starting in R2019b, colormaps have 256 colors by default.

In R2019a and previous releases, the default size is 64. If you have code that depends on a colormap having 64 colors, specify the number of colors when you set the colormap for the figure, axes, or chart. For example, `colormap(winter(64))` sets the figure's colormap to the 64-color `winter` colormap.

Alternatively, you can change the default colormap for all figures within your MATLAB session:

```
set(groot,'defaultFigureColormap',winter(64))
```

See Also

`colormap`

Topics

"Change Color Scheme Using a Colormap"

Introduced before R2006a

withtol

Time tolerance for timetable row subscripting

Syntax

```
S = withtol(rowTimes,tol)
```

Description

`S = withtol(rowTimes,tol)` creates a subscript to select rows of a timetable. `S` selects all rows whose row times match a time in `rowTimes` within the tolerance specified by `tol`. The `rowTimes` argument is a datetime or duration array, or a cell array of character vectors that specify dates and times. `tol` must be a duration value.

If `rowTimes` contains datetime values, then you can only use `S` to subscript into a timetable whose row times are datetime values. Similarly, if `rowTimes` contains duration values, then you can only use `S` to subscript into a timetable whose row times are duration values.

Examples

Specify Tolerance to Match Row Times

Create a timetable that contains temperature and pressure data with row times in hours. There is a slight random variance in the row times. Select rows with row times that match corresponding times in a time vector within a tolerance of five seconds.

```
Time = datetime(2015,12,18) + hours(1:10)' + seconds(randn(10,1));
Temp = [37.3 39.1 42.3 42.6 43 43.9 44.1 43.3 42.5 42]';
Pressure = [29.4 29.6 30.0 30.0 30.1 29.9 29.9 29.8 29.6 29.7]';
TT = timetable(Time,Temp,Pressure)
```

```
TT=10×3 timetable
            Time            Temp    Pressure
```

```
    ____________________    ____    ________

    18-Dec-2015 01:00:00    37.3      29.4
    18-Dec-2015 02:00:01    39.1      29.6
    18-Dec-2015 02:59:57    42.3        30
    18-Dec-2015 04:00:00    42.6        30
    18-Dec-2015 05:00:00      43      30.1
    18-Dec-2015 05:59:58    43.9      29.9
    18-Dec-2015 06:59:59    44.1      29.9
    18-Dec-2015 08:00:00    43.3      29.8
    18-Dec-2015 09:00:03    42.5      29.6
    18-Dec-2015 10:00:02      42      29.7
```

Create a time vector spanning the hours from 3:00 to 8:00.

```
newTimes = datetime(2015,12,18) + hours(3:8)
```

```
newTimes = 1x6 datetime array
Columns 1 through 3

   18-Dec-2015 03:00:00   18-Dec-2015 04:00:00   18-Dec-2015 05:00:00

Columns 4 through 6

   18-Dec-2015 06:00:00   18-Dec-2015 07:00:00   18-Dec-2015 08:00:00
```

Select rows of `TT` with row times that match times in `newTimes` within five seconds.

```
S = withtol(newTimes,seconds(5));
TT2 = TT(S,:)
```

```
TT2=6×3 timetable
            Time            Temp    Pressure
    ____________________    ____    ________

    18-Dec-2015 02:59:57    42.3        30
    18-Dec-2015 04:00:00    42.6        30
    18-Dec-2015 05:00:00      43      30.1
    18-Dec-2015 05:59:58    43.9      29.9
    18-Dec-2015 06:59:59    44.1      29.9
    18-Dec-2015 08:00:00    43.3      29.8
```

Input Arguments

`rowTimes` — Times to match in timetable
datetime array | duration array | cell array of character vectors | string array

Times to match in a timetable, specified as a datetime array, duration array, cell array of character vectors, or string array. `rowTimes` contains times that do not exactly match times in the row times of a timetable, but that might be within a specified tolerance.

If `rowTimes` is a cell array of character vectors or string array, then the elements of the array specify dates and times that the `datetime` or `duration` functions can convert.

`tol` — Tolerance for matching times to row times of timetable
duration | character vector | string scalar

Tolerance for matching times to the row times of a timetable, specified as a duration, character vector, or string scalar.

If `tol` is a character vector or string scalar, then `tol` specifies a time that the `duration` function can convert.

See Also

`retime` | `synchronize` | `timerange` | `vartype`

Topics

"Create Timetables"
"Select Timetable Data by Row Time and Variable Type"
"Tables"
"Represent Dates and Times in MATLAB"

Introduced in R2016b

wordcloud

Create word cloud chart from text data

Syntax

```
wc = wordcloud(tbl,wordVar,sizeVar)
wc = wordcloud(words,sizeData)
wc = wordcloud(C)
```

```
wc = wordcloud(parent, ___ )
wc = wordcloud( ___ ,Name,Value)
```

Description

`wc = wordcloud(tbl,wordVar,sizeVar)` creates a word cloud chart from the table `tbl`. The variables `wordVar` and `sizeVar` in the table specify the words and word sizes respectively.

`wc = wordcloud(words,sizeData)` creates a word cloud chart from elements of `words` with word sizes specified by `SizeData`.

`wc = wordcloud(C)` creates a word cloud chart from the unique elements of categorical array `C` with sizes corresponding to their frequency counts. If you have Text Analytics Toolbox™, then `C` can be a string array, character vector, or a cell array of character vectors.

`wc = wordcloud(parent, ___ )` creates the word cloud in the figure, panel, or tab specified by `parent`.

`wc = wordcloud( ___ ,Name,Value)` specifies additional `WordCloudChart` properties using one or more name-value pair arguments.

Note Text Analytics Toolbox extends the functionality of the `wordcloud` (MATLAB) function. It adds support for creating word clouds directly from string arrays, and creating word clouds from bag-of-words models, bag-of-n-gram models, and LDA topics. For the `wordcloud` (Text Analytics Toolbox) reference page, see `wordcloud`.

Examples

Create Word Cloud from Table

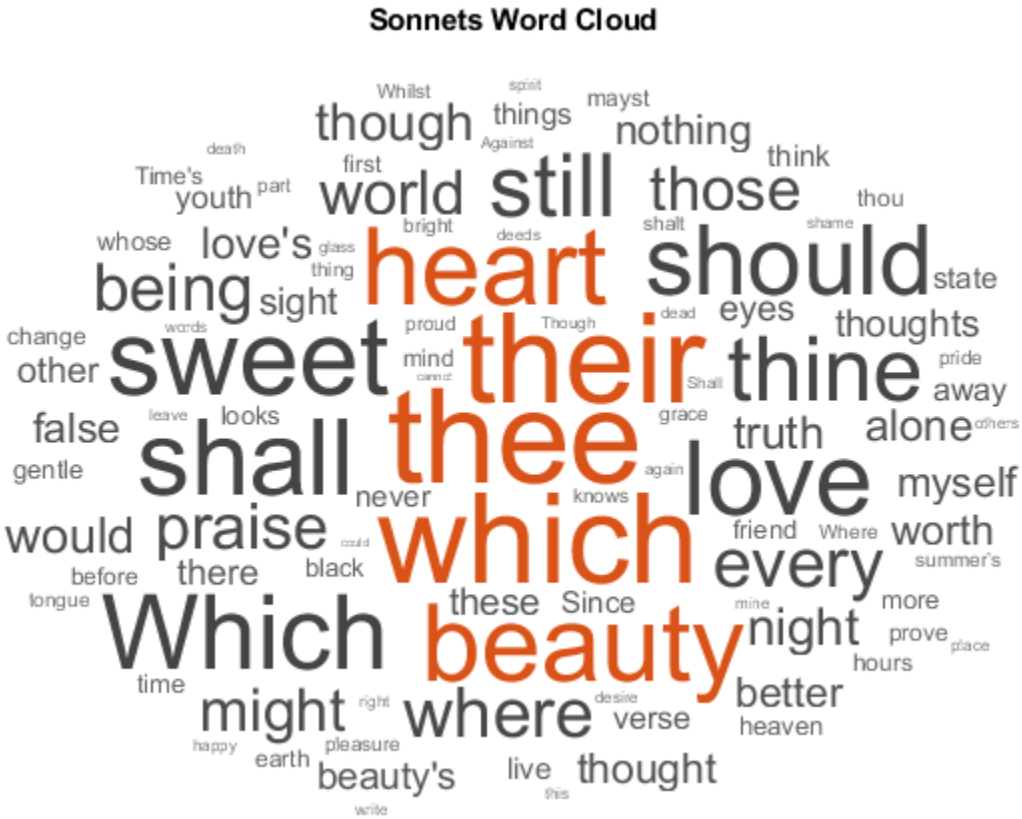
Load the example data `sonnetsTable`. The table `tbl` contains a list of words in the variable `Word`, and the corresponding frequency counts in the variable `Count`.

```
load sonnetsTable
head(tbl)
```

```
ans=8×2 table
       Word        Count
    ___________    _____

    {'''tis'  }      1
    {''Amen'' }      1
    {''Fair'  }      2
    {''Gainst'}      1
    {''Since' }      1
    {''This'  }      2
    {''Thou'  }      1
    {''Thus'  }      1
```

Plot the table data using `wordcloud`. Specify the words and corresponding word sizes to be the `Word` and `Count` variables respectively.

```
figure
wordcloud(tbl,'Word','Count');
title("Sonnets Word Cloud")
```



Prepare Text Data for Word Clouds

If you have Text Analytics Toolbox™ installed, then you can create word clouds directly from string arrays. For more information, see `wordcloud` (Text Analytics Toolbox). If you do not have Text Analytics Toolbox, then you must preprocess the text data manually.

This example shows how to create a word cloud from plain text by reading it into a string array, preprocessing it, and passing it to the `wordcloud` function.

Read the text from Shakespeare's Sonnets with the `fileread` function and convert it to string.

```
sonnets = string(fileread('sonnets.txt'));
extractBefore(sonnets,"II")
```

```
ans =
    "THE SONNETS

     by William Shakespeare




       I

       From fairest creatures we desire increase,
       That thereby beauty's rose might never die,
       But as the riper should by time decease,
       His tender heir might bear his memory:
       But thou, contracted to thine own bright eyes,
       Feed'st thy light's flame with self-substantial fuel,
       Making a famine where abundance lies,
       Thy self thy foe, to thy sweet self too cruel:
       Thou that art now the world's fresh ornament,
       And only herald to the gaudy spring,
       Within thine own bud buriest thy content,
       And tender churl mak'st waste in niggarding:
         Pity the world, or else this glutton be,
         To eat the world's due, by the grave and thee.

       "
```

Split `sonnets` into a string array whose elements contain individual words. To do this, remove the punctuation characters and join all the string elements into a 1-by-1 string and then split on the space characters. Then, remove words with fewer than five characters and convert the words to lowercase.

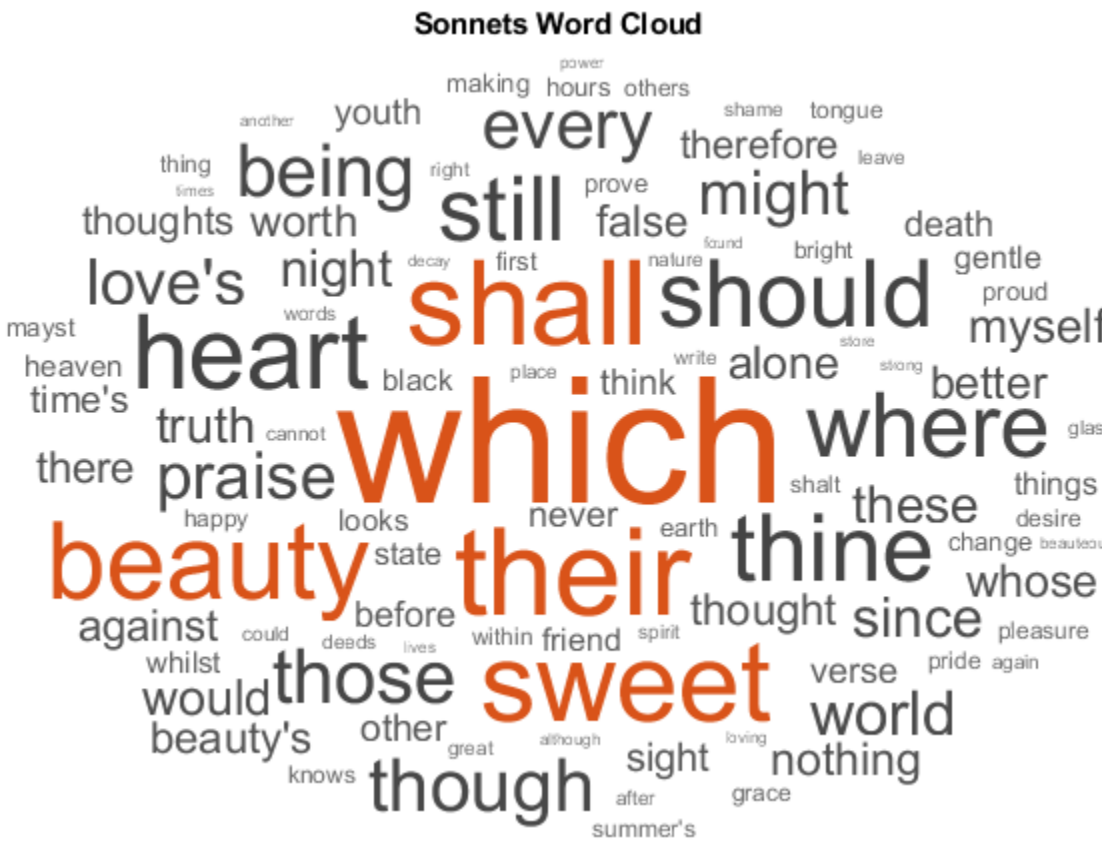
```
punctuationCharacters = ["." "?" "!" "," ";" ":"];
sonnets = replace(sonnets,punctuationCharacters," ");
words = split(join(sonnets));
words(strlength(words)<5) = [];
words = lower(words);
words(1:10)
```

```
ans = 10x1 string array
    "sonnets"
```

```
    "william"
    "shakespeare"
    "fairest"
    "creatures"
    "desire"
    "increase"
    "thereby"
    "beauty's"
    "might"
```

Convert `sonnets` to a categorical array and then plot using `wordcloud`. The function plots the unique elements of `C` with sizes corresponding to their frequency counts.

```
C = categorical(words);
figure
wordcloud(C);
title("Sonnets Word Cloud")
```



Specify Word Sizes

Create a word cloud from plain text by reading it into a string array, preprocessing it, and passing it to the `wordcloud` function.

Read the text from Shakespeare's Sonnets with the `fileread` function and convert it to string.

```
sonnets = string(fileread('sonnets.txt'));
extractBefore(sonnets,"II")
```

```
ans =
    "THE SONNETS

     by William Shakespeare




       I

       From fairest creatures we desire increase,
       That thereby beauty's rose might never die,
       But as the riper should by time decease,
       His tender heir might bear his memory:
       But thou, contracted to thine own bright eyes,
       Feed'st thy light's flame with self-substantial fuel,
       Making a famine where abundance lies,
       Thy self thy foe, to thy sweet self too cruel:
       Thou that art now the world's fresh ornament,
       And only herald to the gaudy spring,
       Within thine own bud buriest thy content,
       And tender churl mak'st waste in niggarding:
         Pity the world, or else this glutton be,
         To eat the world's due, by the grave and thee.

       "
```

Split `sonnets` into a string array whose elements contain individual words. To do this, remove the punctuation characters and join all the string elements into a 1-by-1 string and then split on the space characters. Then, remove words with fewer than five characters and convert the words to lowercase.

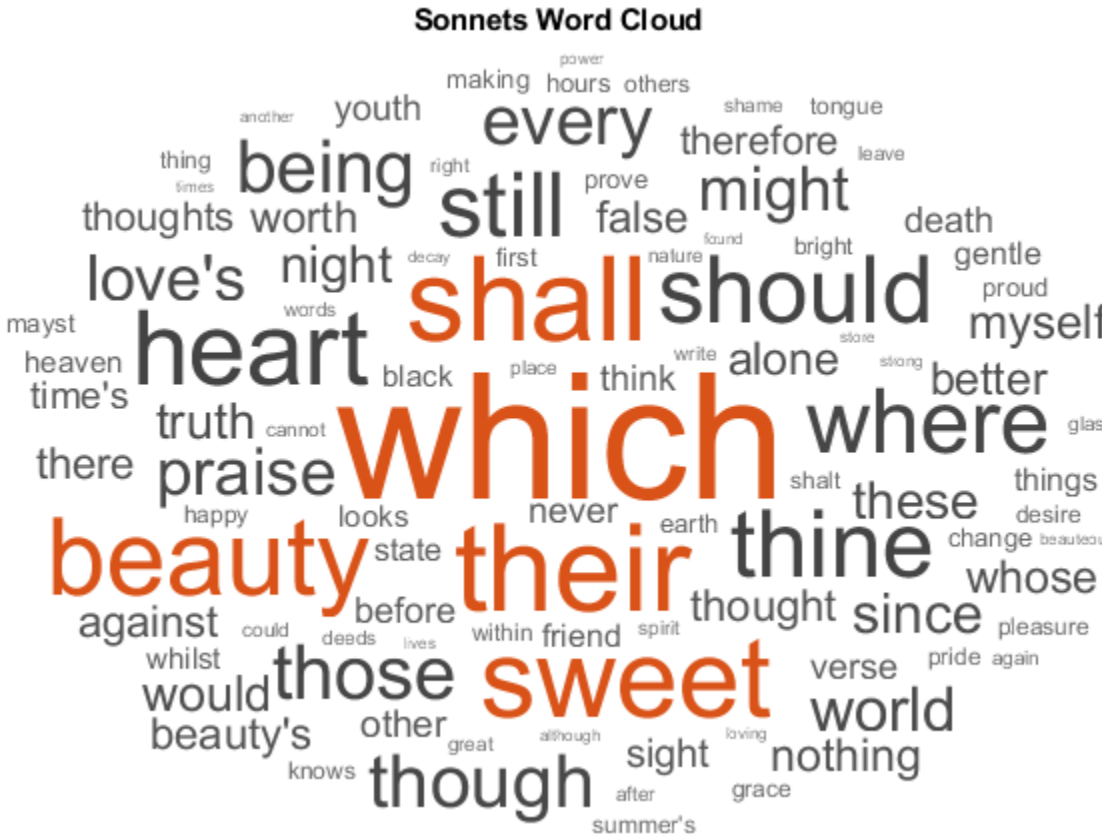
```
punctuationCharacters = ["." "?" "!" "," ";" ":"];
sonnets = replace(sonnets,punctuationCharacters," ");
words = split(join(sonnets));
words(strlength(words)<5) = [];
words = lower(words);
words(1:10)
```

```
ans = 10x1 string array
    "sonnets"
    "william"
    "shakespeare"
    "fairest"
```

```
    "creatures"
    "desire"
    "increase"
    "thereby"
    "beauty's"
    "might"
```

Find the unique words in `sonnets` and count their frequency. Create a word cloud using the frequency counts as size data.

```
[numOccurrences,uniqueWords] = histcounts(categorical(words));
figure
wordcloud(uniqueWords,numOccurrences);
title("Sonnets Word Cloud")
```



Specify Word Colors

Load the example data `sonnetsTable`. The table `tbl` contains a list of words in the `Word` variable, and corresponding frequency counts in the `Count` variable.

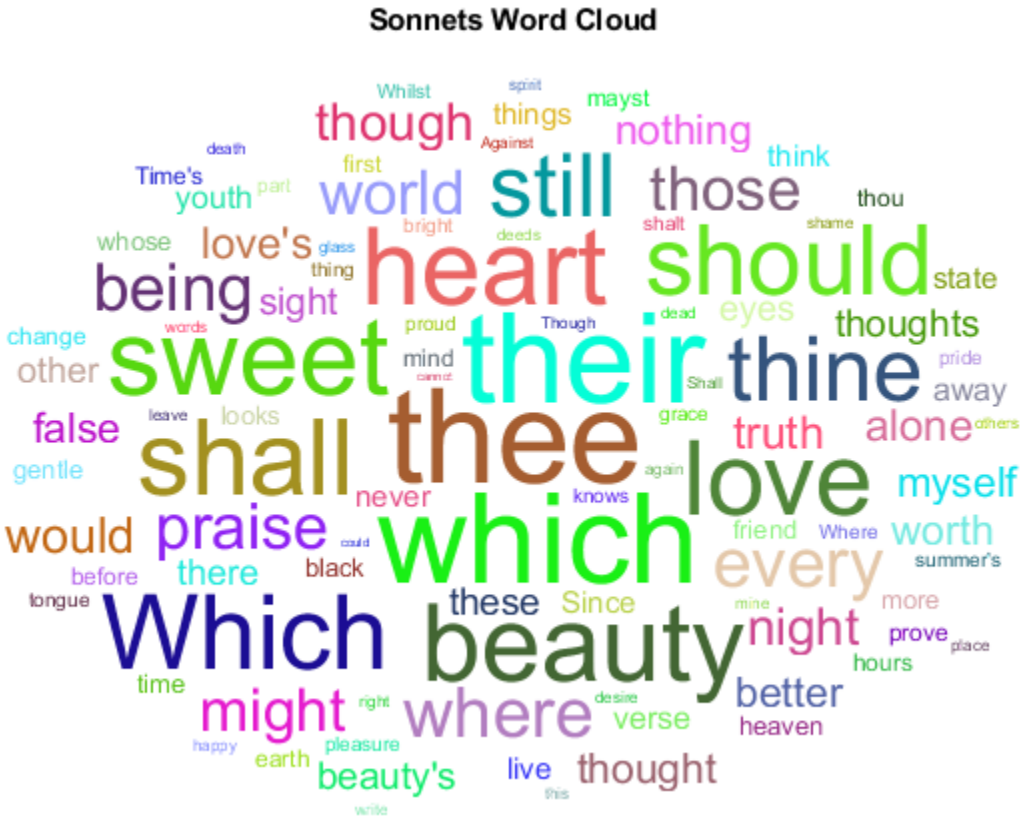
```
load sonnetsTable
head(tbl)
```

```
ans=8×2 table
       Word        Count
    ___________    _____
```

```
    {'''tis'  }      1
    {''Amen'' }      1
    {''Fair'  }      2
    {''Gainst'}      1
    {''Since' }      1
    {''This'  }      2
    {''Thou'  }      1
    {''Thus'  }      1
```

Plot the table data using `wordcloud`. Specify the words and corresponding word sizes to be the `Word` and `Count` variables respectively. To set the word colors to random values, set `'Color'` to a random matrix or RGB triplets with one row for each word.

```
numWords = size(tbl,1);
colors = rand(numWords,3);
figure
wordcloud(tbl,'Word','Count','Color',colors);
title("Sonnets Word Cloud")
```



Create Word Cloud Using Text Analytics Toolbox

If you have Text Analytics Toolbox installed, then you can create word clouds directly from string arrays. If you do not have Text Analytics Toolbox, then you must preprocess the text data manually. For an example showing how to create a word cloud without Text Analytics Toolbox, see “Prepare Text Data for Word Clouds” on page 1-16937.

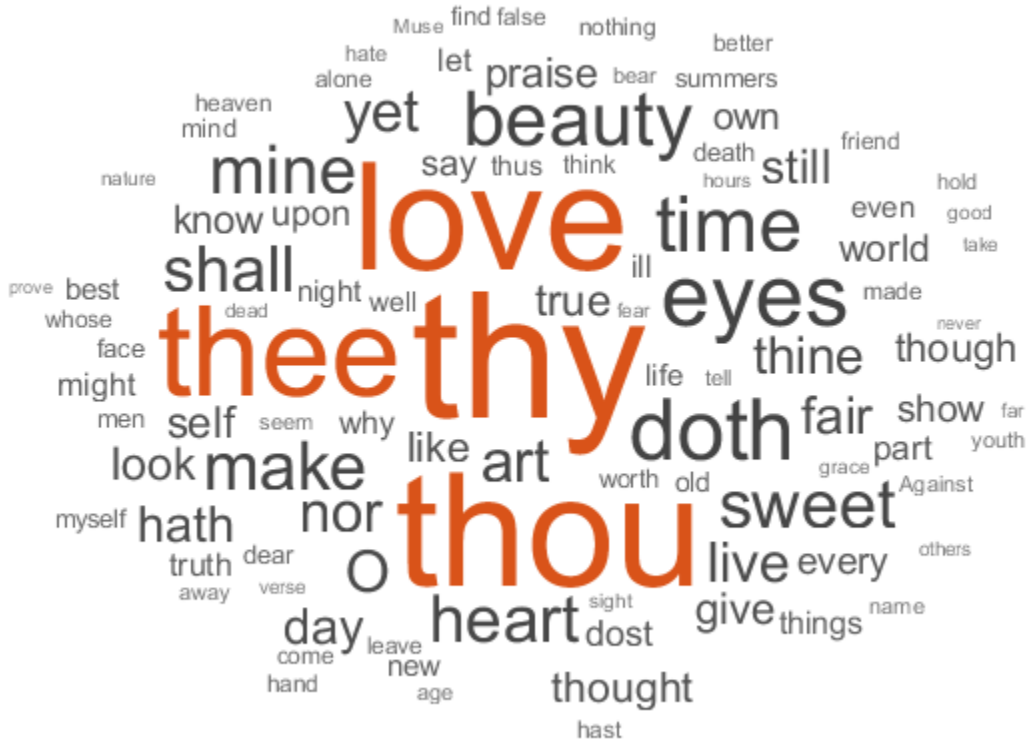
Extract the text from `sonnets.txt` using `extractFileText`.

```
str = extractFileText("sonnets.txt");
extractBefore(str,"II")
```

```
ans =

    "THE SONNETS

     by William Shakespeare



       I

       From fairest creatures we desire increase,
       That thereby beauty's rose might never die,
       But as the riper should by time decease,
       His tender heir might bear his memory:
       But thou, contracted to thine own bright eyes,
       Feed'st thy light's flame with self-substantial fuel,
       Making a famine where abundance lies,
       Thy self thy foe, to thy sweet self too cruel:
       Thou that art now the world's fresh ornament,
       And only herald to the gaudy spring,
       Within thine own bud buriest thy content,
       And tender churl mak'st waste in niggarding:
         Pity the world, or else this glutton be,
         To eat the world's due, by the grave and thee.

       "
```

Display the words from the sonnets in a word cloud.

```
figure
wordcloud(str);
```



Input Arguments

`tbl` — Input table
table

Input table, with columns specifying the words and word sizes. Specify the words and the corresponding word sizes in the variables given by `wordVar` and `sizeVar` input arguments respectively.

Data Types: `table`

`wordVar` — Table variable for word data
string scalar | character vector | numeric index | logical vector

Table variable for word data, specified as a string scalar, character vector, numeric index, or a logical vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string`

`sizeVar` — Table variable for size data
string scalar | character vector | numeric index | logical vector

Table variable for size data, specified as a string scalar, character vector, numeric index, or a logical vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string`

`C` — Input categorical data
categorical array

Input categorical data, specified as a categorical array. The function plots each unique element of `C` with size corresponding to `histcounts(C)`.

Data Types: `categorical`

`words` — Input words
string vector | cell array of character vectors

Input words, specified as a string vector or cell array of character vectors.

Data Types: `string` | `cell`

`sizeData` — Word size data
numeric vector

Word size data, specified as a numeric vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`parent` — Parent container
`Figure` object | `Panel` object | `Tab` object | `TiledChartLayout` object

Parent container, specified as a `Figure`, `Panel`, `Tab`, or `TiledChartLayout` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'HighlightColor','red'` sets the highlight color to red.

The `WordCloudChart` properties listed here are only a subset. For a complete list, see WordCloudChart Properties.

`MaxDisplayWords` — Maximum number of words to display
100 (default) | nonnegative integer

Maximum number of words to display, specified as a non-negative integer. The software displays the `MaxDisplayWords` largest words.

`Color` — Word color
`[0.2510 0.2510 0.2510]` (default) | RGB triplet | character vector containing a color name | matrix

Word color, specified as an RGB triplet, a character vector containing a color name, or an `N`-by-3 matrix where `N` is the length of `WordData`. If `Color` is a matrix, then each row corresponds to an RGB triplet for the corresponding word in `WordData`.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `'blue'`

Example: `[0 0 1]`

`HighlightColor` — Word highlight color

`[0.8510 0.3255 0.0980]` (default) | RGB triplet | character vector containing a color name

Word highlight color, specified as an RGB triplet, or a character vector containing a color name. The software highlights the largest words with this color.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'blue'`

Example: `[0 0 1]`

Shape — Shape of word cloud
`'oval'` (default) | `'rectangle'`

Shape of word cloud chart, specified as `'oval'` or `'rectangle'`.

Example: `'rectangle'`

`LayoutNum` — Word placement layout
1 (default) | nonnegative integer

Word placement layout, specified as a nonnegative integer. If you repeatedly call `wordcloud` with the same inputs, then the word placement layouts will be the same each time. To get different word placement layouts, use different values of `LayoutNum`.

Output Arguments

`wc` — `WordCloudChart` object
`WordCloudChart` object

`WordCloudChart` object. You can modify the properties of a `WordCloudChart` after it is created. For more information, see WordCloudChart Properties.

Tips

Text Analytics Toolbox extends the functionality of the `wordcloud` (MATLAB) function. It adds support for creating word clouds directly from string arrays, and creating word clouds from bag-of-words models, bag-of-n-gram models, and LDA topics. For the `wordcloud` (Text Analytics Toolbox) reference page, see `wordcloud`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- The syntax `wc = wordcloud(str)`, where `str` is a string array, character vector, or cell array of character vectors (these inputs require Text Analytics Toolbox) is not supported.
- When the `words` and `sizedata` inputs are provided as tall arrays, then they are gathered into memory and thus, must fit into memory.

See Also

`join` | `lower` | `replace` | `split` | `splitlines` | `string` | `wordcloud`

Topics

"Analyze Text Data with String Arrays"
"Create String Arrays"
"Search and Replace Text"
"Compare Text"
"Test for Empty Strings and Missing Values"

Introduced in R2017b

WordCloudChart Properties

Control word cloud chart appearance and behavior

Description

`WordCloudChart` properties control the appearance and behavior of a `WordCloudChart` object. By changing property values, you can modify certain aspects of the word cloud chart.

Properties

Title

`Title` — Text displayed above word cloud chart in figure

`''` (default) | character array | cell array of character vectors | string | cell array of strings | numeric value | categorical array

Text displayed above word cloud chart in figure, specified as a character array, cell array of character vectors, scalar string, cell array of strings, a numeric value, or a categorical array. If you specify this property as a categorical array, then MATLAB uses the values in the array, not the full set of categories. You can also use the `title` function to set this value.

Example: `"Word Cloud"`

Data Types: `string` | `char` | `cell`

`TitleFontName` — Font used for title text in the word cloud chart

`'Helvetica'` (default) | character vector | string

Font used for title text in the word cloud chart, specified as a string or character vector. To display and print properly, the font name must be a font that your system supports. The default font depends on the specific operating system and locale.

Example: `'Cambria'`

Data Types: `char` | `string`

Color and Styling

`Color` — Word color

`[0.2510 0.2510 0.2510]` (default) | RGB triplet | character vector containing a color name | matrix

Word color, specified as an RGB triplet, a character vector containing a color name, or an `N`-by-3 matrix where `N` is the length of `WordData`. If `Color` is a matrix, then each row corresponds to an RGB triplet for the corresponding word in `WordData`.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: 'blue'

Example: [0 0 1]

HighlightColor — Word highlight color

[0.8510 0.3255 0.0980] (default) | RGB triplet | character vector containing a color name

Word highlight color, specified as an RGB triplet, or a character vector containing a color name. The software highlights the largest words with this color.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `'blue'`

Example: `[0 0 1]`

`FontName` — Font used for text in the word cloud chart
`'Helvetica'` (default) | character vector | string

Font used for text in the word cloud chart, specified as a string or character vector. To display and print properly, the font name must be a font that your system supports. The default font depends on the specific operating system and locale.

Example: `'Cambria'`

Data Types: `char` | `string`

`MaxDisplayWords` — Maximum number of words to display
100 (default) | nonnegative integer

Maximum number of words to display, specified as a non-negative integer. The software displays the `MaxDisplayWords` largest words.

`Box` — Axes box outline
`'off'` (default) | `'on'`

Axes box outline, specified as one of these values:

- `'off'` — Do not display the box outline around the axes.
- `'on'` — Display the box outline around the axes.

Example: `'on'`

`Shape` — Shape of word cloud
`'oval'` (default) | `'rectangle'`

Shape of word cloud chart, specified as `'oval'` or `'rectangle'`.

Example: `'rectangle'`

`LayoutNum` — Word placement layout
1 (default) | nonnegative integer

Word placement layout, specified as a nonnegative integer. If you repeatedly call `wordcloud` with the same inputs, then the word placement layouts will be the same each time. To get different word placement layouts, use different values of `LayoutNum`.

`SizePower` — Power to apply to `SizeData`
0.5 (default) | positive scalar

Power to apply to `SizeData`, specified as a positive scalar. The software displays words with sizes given by `SizeData .^ SizePower`.

Example: 2

Data

`WordData` — Word data
`""` (default) | string vector | cell array of character vectors

Word data, specified as a string vector, or a cell array of character vectors.

Example: `["one" "two" "three"]`

Data Types: `string` | `cell`

`SizeData` — Size data

`[]` (default) | numeric vector

Size data, specified as a numeric vector.

Example: `[0.1 0.2 0.3]`

`WordVariable` — Word variable in source table

character vector

Word variable in the source table, specified as a character vector.

Data Types: `char`

`SizeVariable` — Size variable in source table

character vector

Size variable in the source table, specified as a character vector.

Data Types: `char`

`SourceTable` — Source table

table

Source table containing word and size data.

Data Types: `table`

Position

`OuterPosition` — Size and position of word cloud chart within its parent

`[0 0 1 1]` (default) | 1-by-4 numeric vector

Size and position of word cloud chart within its parent, specified as a 1-by-4 numeric vector of the form `[left,bottom,width,height]`. This includes labels and margins. The default value of `[0 0 1 1]` includes the whole interior of the container.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`InnerPosition` — Inner size and position

`[0 0 1 1]` (default) | 1-by-4 numeric vector

Inner size and position of the word cloud within the parent container (typically a figure, panel, or tab) returned as a 1-by-4 numeric vector of the form `[left bottom width height]`.

- The `left` and `bottom` elements define the distance from the lower left corner of the container to the lower left corner of the word cloud.
- The `width` and `height` elements are the word cloud dimensions.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`Position` — Inner size and position

four element vector

Inner size and position of the word cloud within the parent container (typically a figure, panel, or tab) specified as a four-element vector of the form `[left bottom width height]`. This property is equivalent to the `InnerPosition` property.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`ActivePositionProperty` — Position property to hold constant

`'outerposition'` | `'innerposition'`

Position property to hold constant when adding, removing, or changing decorations, specified as one of the following values:

- `'outerposition'` — The `OuterPosition` property remains constant when you add, remove, or change decorations such as a title or an axis label. If any positional adjustments are needed, MATLAB adjusts the `InnerPosition` property.
- `'innerposition'` — The `InnerPosition` property remains constant when you add, remove, or change decorations such as a title or an axis label. If any positional adjustments are needed, MATLAB adjusts the `OuterPosition` property.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

Units — Units used by OuterPosition property

'normalized' (default) | 'points' | 'pixels' | 'characters' | 'inches' | 'centimeters'

Units used by OuterPosition property, specified as 'normalized', 'points', 'pixels', 'characters', 'inches', or 'centimeters'.

Layout — Layout options

empty LayoutOptions array (default) | TiledChartLayoutOptions object

Layout options, specified as a TiledChartLayoutOptions object. This property specifies options when the chart is a child of a tiled chart layout. Use this property to position the chart within the layout by setting the Tile and TileSpan properties on the TiledChartLayoutOptions object.

For example, this code places chart c in the third tile of a tiled chart layout.

```
c.Layout.Tile = 3;
```

To make the chart span multiple tiles, specify the TileSpan property as a two-element vector. For example, this chart spans 2 rows and 3 columns of tiles.

```
c.Layout.TileSpan = [2 3];
```

If the chart is not a child of a tiled chart layout (for example, if it is a child of a figure or panel), then this property is empty and has no effect.

Visible — State of visibility

'on' (default) | 'off'

State of visibility, specified as one of these values:

- 'on' — Display the object.
- 'off' — Hide the object without deleting it. You still can access the properties of an invisible object.

Parent/Child

Parent — Parent container

Figure object | Panel object | Tab object | TiledChartLayout object

Parent container, specified as a Figure, Panel, Tab, or TiledChartLayout object.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

See Also

`wordcloud`

Topics

"Analyze Text Data with String Arrays"
"Create String Arrays"
"Search and Replace Text"
"Compare Text"
"Test for Empty Strings and Missing Values"

Introduced in R2017b

Workspace Browser

Open Workspace browser to manage workspace

Description

The Workspace browser enables you to view and interactively manage the contents of the workspace in MATLAB. For each variable or object in the workspace, the Workspace browser also can display statistics, when relevant, such as the minimum, maximum, and mean.

You can edit the contents of scalar (1-by-1) variables directly in the Workspace browser. Right-click the variable and select **Edit Value**. To edit other variables, double-click the variable name in the Workspace browser to open it in the Variables editor.

Open the Workspace Browser

To open the Workspace browser if it is not currently visible, do one of the following:

- MATLAB Toolstrip: On the **Home** tab, in the **Environment** section, click **Layout**. Then, in the **Show** section, select **Workspace**.
- MATLAB command prompt: Enter `workspace`.

Examples

Create and Edit a Variable

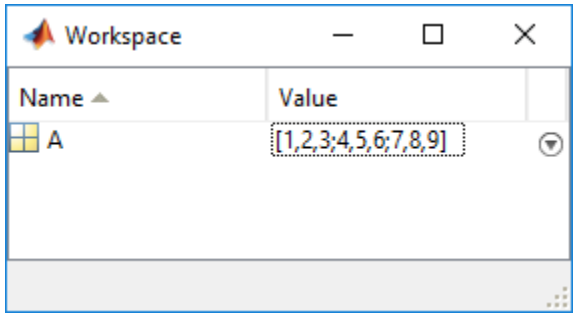
Create a 3x3 matrix and then edit the values of the matrix in the Workspace browser.

Create a 3x3 matrix and store it in `A`.

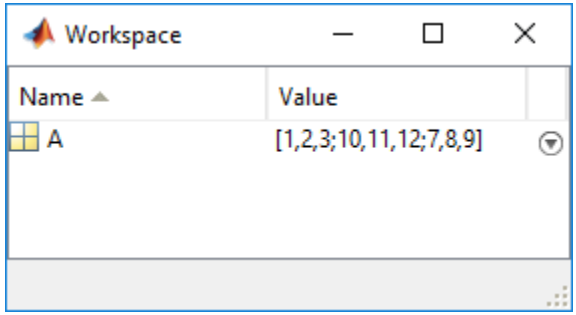
```
A = [1 2 3; 4 5 6; 7 8 9];
```

Open the Workspace browser to view `A`.

```
workspace
```



Right-click the variable and select **Edit Value**. Replace the numbers 4, 5, and 6 with the numbers 10, 11, and 12, respectively.



Programmatic Use

`workspace` displays the Workspace browser. If the Workspace browser is already open, MATLAB selects the tool.

See Also

`clear` | `load` | `openvar` | `save` | `who`

Topics

"Create and Edit Variables"
"Enter Statements in Command Window"

“Use MATLAB Engine Workspace in Python”

Introduced before R2006a

write

Write tall array to local and remote locations for checkpointing

Syntax

```
write(location,tA)
write(filepattern,tA)
write( ___ ,Name,Value)
```

Description

`write(location,tA)` calculates the values in tall array `tA` and writes the array to files in the folder specified by `location`. The data is stored in an efficient binary format suitable for reading back using `datastore(location)`.

`write(filepattern,tA)` uses the file extension from `filepattern` to determine the output format. `filepattern` must include a folder to write the files into, followed by a file name that includes a wildcard `*`. The wildcard represents incremental numbers for generating unique file names. For example, `write('folder/myfile_*.csv',tA)`.

`write( ___ ,Name,Value)` specifies additional options with one or more name-value pair arguments using any of the previous syntaxes. For example, you can specify the file type with `'FileType'` and a valid file type (`'mat'`, `'seq'`, `'parquet'`, `'text'`, or `'spreadsheet'`), or you can specify a custom write function to process the data with `'WriteFcn'` and a function handle.

Examples

Write and Reconstruct Tall Array

Write a tall array to disk, and then recover the tall array by creating a new datastore for the written files. This process is useful to save your work or share a tall array with a colleague.

Create a datastore for the `airlinesmall.csv` data set. Select only the `Year`, `Month`, and `UniqueCarrier` variables, and treat `'NA'` values as missing data. Convert the datastore into a tall table.

```
ds = datastore('airlinesmall.csv');
ds.TreatAsMissing = 'NA';
ds.SelectedVariableNames = {'Month','Year','UniqueCarrier'};
tt = tall(ds)
```

```
tt =

  M×3 tall table

    Month    Year    UniqueCarrier
    _____    ____    _____________

     10      1987        'PS'
     10      1987        'PS'
     10      1987        'PS'
     10      1987        'PS'
     10      1987        'PS'
     10      1987        'PS'
     10      1987        'PS'
     10      1987        'PS'
      :        :            :
      :        :            :
```

Sort the data in descending order by year and extract the top 25 rows. The resulting tall table is unevaluated.

```
tt_new = topkrows(tt,25,'Year')
```

```
tt_new =

  M×3 tall table

    Month    Year    UniqueCarrier
    _____    ____    _____________

      ?       ?            ?
      ?       ?            ?
      ?       ?            ?
      :       :            :
      :       :            :
```

```
Preview deferred. Learn more.
```

Save the results to a new folder named `ExampleData` on the `C:\` disk. (You can specify a different write location, especially if you are not using a Windows® computer.) The `write` function evaluates the tall array prior to writing the files, so there is no need to use the `gather` function prior to saving the data.

```
location = 'C:\ExampleData';
write(location,tt_new)
```

```
Writing tall data to folder C:\ExampleData
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 0.52 sec
Evaluation completed in 1.2 sec
```

Clear `tt` and `ds` from your working directory. To recover the tall table that was written to disk, first create a new datastore that references the same directory. Then convert the datastore into a tall table. Since the tall table was evaluated before being written to disk, the display now includes a preview of the values.

```
clear tt ds
ds2 = datastore(location);
tt2 = tall(ds2)
```

```
tt2 =

  M×3 tall table

    Month    Year    UniqueCarrier
    _____    ____    _____________

      1      2008        'WN'
      1      2008        'WN'
      1      2008        'WN'
      1      2008        'WN'
      1      2008        'WN'
      1      2008        'WN'
      1      2008        'WN'
      1      2008        'WN'
      :       :            :
      :       :            :
```

Write Spreadsheet File to Cloud Storage

Create a tall table referencing the `tsunamis.xlsx` data file, which contains time-stamped data about the location, magnitude, and cause of tsunamis.

```
ds = datastore('tsunamis.xlsx');
T = tall(ds)
```

```
T =

  Mx20 tall table

    Latitude    Longitude    Year    Month    Day    Hour    Minute    Second    Valid
    ________    _________    ____    _____    ___    ____    ______    ______    _____

      -3.8        128.3      1950     10       8       3       23       NaN          2
      19.5         -156      1951      8      21      10       57       NaN          4
     -9.02       157.95      1951     12      22     NaN      NaN       NaN          2
     42.15       143.85      1952      3       4       1       22        41          4
      19.1         -155      1952      3      17       3       58       NaN          4
      43.1        -82.4      1952      5       6     NaN      NaN       NaN          1
     52.75        159.5      1952     11       4      16       58       NaN          4
        50        156.5      1953      3      18     NaN      NaN       NaN          3
       :            :          :        :       :      :        :         :
       :            :          :        :       :      :        :         :
```

Combine the `Year`, `Month`, `Day`, `Hour`, `Minute`, and `Second` variables into a single datetime variable, and then remove those variables from the table. Remove any rows that contain missing data.

```
T.DateTime = datetime(T.Year, T.Month, T.Day, T.Hour, T.Minute, T.Second);
T(:,3:8) = [];
TT = rmmissing(T)
```

```
TT =

  Mx15 tall table

    Latitude    Longitude    ValidityCode          Validity          CauseCode
    ________    _________    ____________    ____________________    _________    ____

     42.15        143.85          4          {'definite tsunami'}        1        {'Ea
     58.34       -136.52          4          {'definite tsunami'}        3        {'Ea
     -39.5         -74.5          4          {'definite tsunami'}        1        {'Ea
      -6.8         -80.7          4          {'definite tsunami'}        1        {'Ea
```

```
    61.1        -147.5          4          {'definite tsunami'}        3        {'Ear
   38.65         139.2          4          {'definite tsunami'}        1        {'Ear
     0.2         119.8          4          {'definite tsunami'}        1        {'Ear
    -3.1         118.9          4          {'definite tsunami'}        1        {'Ear
     :             :             :                   :                  :
     :             :             :                   :                  :
```

Write the table as a spreadsheet file to a remote location in Amazon S3® storage. To read or write data to Amazon S3 you must set the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables using the credentials for your account. For more information, see “Work with Remote Data”.

```
setenv('AWS_ACCESS_KEY_ID', 'YOUR_AWS_ACCESS_KEY_ID');
setenv('AWS_SECRET_ACCESS_KEY', 'YOUR_AWS_SECRET_ACCESS_KEY');

location = 's3://bucketname/preprocessedData/';
write(location, TT, 'FileType', 'spreadsheet')
```

To read the data back, use `datastore` to point to the remote location where the data now resides.

```
ds = datastore(location);
tt = tall(ds);
```

Write Images with Custom Write Function

Create and use a custom writing function to write data in additional formats that are not directly supported by `write`, such as image files.

Create an image datastore that references all of the sample images in the `toolbox/matlab/demos` folder. The selected images have the extensions `.jpg`, `.tif`, and `.png`. Convert the datastore to a tall cell array.

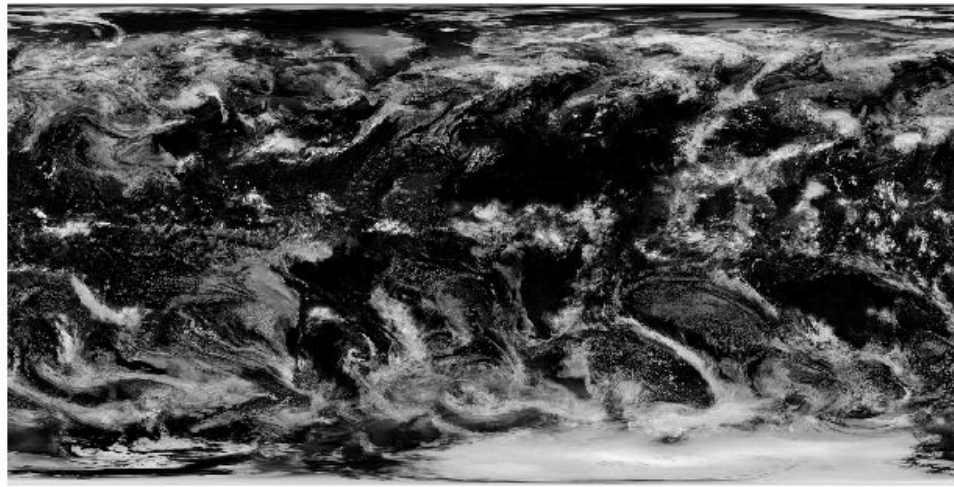
```
demoFolder = fullfile(matlabroot,'toolbox','matlab','demos');
ds = imageDatastore(demoFolder,'FileExtensions',{'.jpg' '.tif' '.png'});
T = tall(ds);
```

Bring one of the images into memory and display it.

```
I = gather(T(1));
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 3 sec
Evaluation completed in 3.2 sec
```

```
imshow(I{1},'InitialMagnification',30)
```



`write` does not support image files directly, so to write the images out in a different format, you must create a new function to handle the file writing. The writing function receives two inputs from `write`:

- **`info`** is a structure containing fields with information about the current block of data. You can use these fields to construct your own unique file name, or simply use the `SuggestedFilename` field to use a name suggested by `write`.
- **`data`** is the current block of data, obtained by using `read` on the datastore.

The function `imageWriter` uses the filename suggested by `write`, and uses `imwrite` to write the image files to disk as `.jpg` files. Save this function in your current working folder.

```
type imageWriter
```

```
function imageWriter(info, data)
  filename = info.SuggestedFilename;
  imwrite(data{:}, filename)
end
```

Write the images in the datastore to a new folder named `exampleImages` on the `C:\` disk. (You can use a different location, especially if you are not using a Windows® computer.) Pass `imageWriter` as the custom write function using the `'WriteFcn'` name-value pair argument.

```
location = 'C:\exampleImages\image_*.jpg';
write(location, T, 'WriteFcn', @imageWriter)
```

```
Writing tall data to folder C:\exampleImages
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 0.71 sec
Evaluation completed in 0.98 sec
```

Display the contents of the folder where the files were written.

```
dir C:\exampleImages
```

```
.                   image_1_000001.jpg  image_3_000001.jpg  image_5_000001.jpg
..                  image_2_000001.jpg  image_4_000001.jpg  image_6_000001.jpg
```

To read the images back into MATLAB®, create a datastore that references the same location.

```
ds = imageDatastore(location);
T = tall(ds)
```

```
T =

  6×1 tall cell array

    {1024×2048×3 uint8}
    { 650×600×3  uint8}
    {1024×2048×3 uint8}
    { 650×600×3  uint8}
    { 480×640×3  uint8}
    { 480×640×3  uint8}
```

Input Arguments

`location` — Folder location to write data

character vector | string

Folder location to write data, specified as a character vector or string. `location` can specify a full or relative path. The specified folder can be either of these options:

- Existing empty folder
- New folder that `write` creates

You can write data to local folders on your computer, folders on a shared network, or to remote locations in HDFS, Azure®, or Amazon S3. For more information about reading and writing data to remote locations, see “Work with Remote Data”.

Additional considerations apply for Hadoop and Apache Spark:

- If the folder is not available locally, then the full path of the folder must be a uniform resource locator (URL) of the form: `hdfs:///path_to_file`.
- Before writing to HDFS, set the `HADOOP_HOME`, `HADOOP_PREFIX`, or `MATLAB_HADOOP_INSTALL` environment variable to the folder where Hadoop is installed.
- Before writing to Apache Spark, set the `SPARK_HOME` environment variable to the folder where Apache Spark is installed.

Example: `location = 'hdfs:///some/output/folder'` specifies an HDFS URL.

Example: `location = '../../dir/data'` specifies a relative file path.

Example: `location = 'C:\Users\MyName\Desktop\data'` specifies an absolute path to a Windows desktop folder.

Example: `location = 'file:///path/to/data'` specifies an absolute URI path to a folder.

Data Types: `char` | `string`

`tA` — Input array

tall array

Input array, specified as a tall array.

`filepattern` — File naming pattern
string | character vector

File naming pattern, specified as a string or a character vector. The file naming pattern must contain a folder to write the files into, followed by a file name that includes a wildcard `*`. The `write` function replaces the wildcard with sequential numbers to ensure unique file names.

Example: `write('folder/data_*.txt',tA)` writes the tall array `tA` as a series of `.txt` files in `folder` with the file names `data_1.txt`, `data_2.txt`, and so on.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `write('C:\myData', tX, 'FileType', 'text', 'WriteVariableNames', false)` writes the tall array `tX` to `C:\myData` as a collection of text files that do not use variable names as column headings.

General Options

`FileType` — Type of file
`'auto'` (default) | `'mat'` | `'parquet'` | `'seq'` | `'text'` | `'spreadsheet'`

Type of file, specified as the comma-separated pair consisting of `'FileType'` and one of the allowed file types: `'auto'`, `'mat'`, `'parquet'`, `'seq'`, `'text'`, or `'spreadsheet'`.

Use the `'FileType'` name-value pair with the `location` argument to specify what type of files to write. By default, `write` attempts to automatically detect the proper file type. You do not need to specify the `'FileType'` name-value pair argument if `write` can determine the file type from an extension in the `location` or `filepattern` arguments. `write` can determine the file type from these extensions:

- `.mat` for MATLAB data files
- `.parquet` or `.parq` for Parquet files
- `.seq` for sequence files

- `.txt`, `.dat`, or `.csv` for delimited text files
- `.xls`, `.xlsx`, `.xlsb`, `.xlsm`, `.xltx`, or `.xltm` for spreadsheet files

Example: `write('C:\myData', tX, 'FileType', 'text')`

`WriteFcn` — Custom writing function
function handle

Custom writing function, specified as the comma-separated pair consisting of `'WriteFcn'` and a function handle. The specified function receives blocks of data from `tA` and is responsible for creating the output files. You can use the `'WriteFcn'` name-value pair argument to write data in a variety of formats, even if `write` does not directly support the output format.

Functional Signature

The custom writing function must accept two input arguments, `info` and `data`:

```
function myWriter(info, data)
```

- `data` contains a block of data from `tA`.
- `info` is a structure with fields that contain information about the block of data. You can use the fields to build a new file name that is globally unique within the final location. The structure fields are

Field	Description
`RequiredLocation`	Fully qualified path to a temporary output folder. Only files written to this folder are copied to the final destination. The files cannot be in subfolders.
`RequiredFilePattern`	The file pattern required for output file names. This field is empty if only a folder name is specified.
`SuggestedFilename`	A fully qualified, globally unique file name that meets the location and naming requirements.
`PartitionIndex`	Index of the tall array partition being written.

Field	Description
`NumPartitions`	Total number of partitions in the tall array.
`BlockIndexInPartition`	Position of current data block within the partition.
`IsFinalBlock`	`true` if current block is the final block of the partition.

File Naming

The file name used for the output files determines the order in which `datastore` later reads the files back in. If the order of the files matters, then the best practice is to use the `SuggestedFilename` field to name the files, since the suggested name guarantees the file order. If you do not use the suggested file name, then the custom writing function must create globally unique, correctly ordered file names. The file names should follow the naming pattern outlined in `RequiredFilePattern`. When running in parallel with Parallel Computing Toolbox, the file names must be unique and correctly ordered between workers, even though each worker writes to its own local folder.

Arrays with Multiple Partitions

You can divide a tall array into partitions to facilitate running calculations on the array in parallel with Parallel Computing Toolbox. Each of the partitions is still comprised of smaller blocks that individually fit into memory.

`info` contains several fields related to partitions: `PartitionIndex`, `NumPartitions`, `BlockIndexInPartition`, and `IsFinalBlock`. These fields are useful when you are writing out a single file and appending to it, which is a common task for arrays with large partitions that have been split into many blocks. The custom writing function is called once per block, and the blocks in one partition are always written in order by the same worker. However, different partitions can be written by different workers.

Example Function

A simple writing function that writes spreadsheet files is

```
function dataWriter(info, data)
  filename = info.SuggestedFilename;
  writetable(data, filename, 'FileType', 'spreadsheet')
end
```

To invoke `dataWriter` as the writing function for some data `tt`, use these commands.

```
tt = tall(array2table(rand(5000,3)));
location = '/tmp/MyData/tt_*.xlsx';
write(location, tt, 'WriteFcn', @dataWriter);
```

For each block, the `dataWriter` function uses the suggested file name in the `info` structure and calls `writetable` to write a spreadsheet file. The suggested file name takes into account the file naming pattern that is specified in the `location` argument.

Data Types: `function_handle`

Text or Spreadsheet Files

`WriteVariableNames` — Indicator for writing variable names as column headings

`true` or `1` (default) | `false` or `0`

Indicator for writing variable names as column headings, specified as the comma-separated pair consisting of `'WriteVariableNames'` and a numeric or logical `1` (`true`) or `0` (`false`).

Indicator	Behavior
`true`	Variable names are included as the column headings of the output. (default)
`false`	Variable names are not included in the output.

`DateLocale` — Locale for writing dates

character vector | string scalar

Locale for writing dates, specified as the comma-separated pair consisting of `'DateLocale'` and a character vector or a string scalar. When writing `datetime` values to the file, use `DateLocale` to specify the locale in which `write` should write month and day-of-week names and abbreviations. The character vector or string takes the form *xx_YY*, where *xx* is a lowercase ISO 639-1 two-letter code indicating a language, and *YY* is an uppercase ISO 3166-1 alpha-2 code indicating a country. For a list of common values for the locale, see the `Locale` name-value pair argument for the `datetime` function.

For Excel files, `write` writes variables containing `datetime` arrays as Excel dates and ignores the `'DateLocale'` parameter value. If the `datetime` variables contain years prior to either 1900 or 1904, then `write` writes the variables as text. For more information on Excel dates, see Differences between the 1900 and the 1904 date system in Excel.

Example: `'DateLocale','ja_JP'` or `'DateLocale',"ja_JP"`

Data Types: `char` | `string`

Text Files Only

`Delimiter` — Field delimiter character

`','` or `'comma'` | `' '` or `'space'` | ...

Field delimiter character, specified as the comma-separated pair consisting of `'Delimiter'` and one of these specifiers.

Specifier	Field Delimiter
`','` `'comma'`	Comma (default)
`' '` `'space'`	Space
`'\t'` `'tab'`	Tab
`';'` `'semi'`	Semicolon
`'|'` `'bar'`	Vertical bar

You can use the `'Delimiter'` name-value pair argument only for delimited text files.

Example: `'Delimiter','space'` or `'Delimiter',"space"`

`QuoteStrings` — Indicator for writing quoted text

`false` (default) | `true`

Indicator for writing quoted text, specified as the comma-separated pair consisting of `'QuoteStrings'` and either `false` or `true`. If `'QuoteStrings'` is set to `true`, then `write` encloses the text in double quotation marks and replaces any double-quote characters that appear as part of that text with two double-quote characters. For an example, see “Write Quoted Text to CSV File” on page 1-17014.

You can use the `'QuoteStrings'` name-value pair argument only with delimited text files.

`Encoding` — Character encoding scheme
`'system'` | `'UTF-8'` | `'ISO-8859-1'` | `'windows-1251'` | `'windows-1252'` | ...

Character encoding scheme associated with the file, specified as the comma-separated pair consisting of `'Encoding'` and `'system'` or a standard character encoding scheme name like one of the values in this table. When you do not specify any encoding or specify encoding as `'system'`, the `write` function uses your system default encoding to write the file.

'Big5'	'ISO-8859-1'	'windows-874'
'Big5-HKSCS'	'ISO-8859-2'	'windows-949'
'CP949'	'ISO-8859-3'	'windows-1250'
'EUC-KR'	'ISO-8859-4'	'windows-1251'
'EUC-JP'	'ISO-8859-5'	'windows-1252'
'EUC-TW'	'ISO-8859-6'	'windows-1253'
'GB18030'	'ISO-8859-7'	'windows-1254'
'GB2312'	'ISO-8859-8'	'windows-1255'
'GBK'	'ISO-8859-9'	'windows-1256'
'IBM866'	'ISO-8859-11'	'windows-1257'
'KOI8-R'	'ISO-8859-13'	'windows-1258'
'KOI8-U'	'ISO-8859-15'	'US-ASCII'
	'Macintosh'	'UTF-8'
	'Shift_JIS'	

Example: `'Encoding','system'` or `'Encoding',"system"` uses the system default encoding.

Spreadsheet Files Only

`Sheet` — Target worksheet
character vector | string scalar | positive integer

Target worksheet, specified as the comma-separated pair consisting of `'Sheet'` and a character vector or a string scalar containing the worksheet name or a positive integer

indicating the worksheet index. The worksheet name cannot contain a colon (`:`). To determine the names of sheets in a spreadsheet file, use `sheets = sheetnames(filename)`. For more information, see `sheetnames`.

If the sheet does not exist, then `write` adds a new sheet at the end of the worksheet collection. If the sheet is an index larger than the number of worksheets, then `write` appends empty sheets until the number of worksheets in the workbook equals the sheet index. In either case, `write` generates a warning indicating that it has added a new worksheet.

You can use the `'Sheet'` name-value pair argument only with spreadsheet files.

Example: `'Sheet',2`

Example: `'Sheet', 'MySheetName'`

Data Types: `char` | `string` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Parquet Files Only

`VariableCompression` — Parquet compression algorithm
`'Snappy'` (default) | `'Brotli'` | `'Gzip'` | `'Uncompressed'` | cell array of character vectors | string vector

Parquet compression algorithm, specified as one of these values.

- `'Snappy'`, `'Brotli'`, `'Gzip'`, or `'Uncompressed'`. If you specify one compression algorithm then `write` compresses all variables using the same algorithm.
- Alternatively, you can specify a cell array of character vectors or a string vector containing the names of the compression algorithms to use for each variable.

In general, `'Snappy'` has better performance for reading and writing, `'Gzip'` has a higher compression ratio at the cost of more CPU processing time, and `'Brotli'` typically produces the smallest file size at the cost of compression speed.

The `write` function writes Parquet 2.0 files using the Parquet dictionary encoding scheme. This encoding scheme is most efficient when the number of unique values is not too large. If the size of the dictionary or number of unique values grows to be too big, then the encoding automatically falls back to plain encoding.

Example: `write('C:\myData', tX, 'FileType', 'parquet', 'VariableCompression', 'Brotli')`

Example: `write('C:\myData', tX, 'FileType', 'parquet', 'VariableCompression', {'Brotli' 'Snappy' 'Gzip'})`

Limitations

In some cases, `write(location, T, 'FileType', type)` creates files that do not represent the original array `T` exactly. If you use `datastore(location)` to read the files, then the result might not have the same format or contents as the original tall table.

1. For the `'text'` and `'spreadsheet'` file types, `write` uses these rules:

   - `write` outputs numeric variables using `longG` format on page 1-0 and categorical, character, or string variables as unquoted text.
   - For nontext variables that have more than one column, `write` outputs multiple delimiter-separated fields on each line and constructs suitable column headings for the first line of the file.
   - `write` outputs variables with more than two dimensions as two-dimensional variables, with trailing dimensions collapsed.
   - For cell-valued variables with contents that are numeric, logical, character, or categorical, `write` outputs the contents of each cell as a single row, in multiple delimiter-separated fields. If the cells have a different data type, `write` outputs a single empty field.

   Do not use the `'text'` or `'spreadsheet'` file types if you need to write an exact checkpoint of the tall array.

2. For the `'parquet'` file type, there are some cases where the Parquet format cannot fully represent the MATLAB table or timetable data types. If you use `parquetread` or `datastore` to read the files, then the result might not have the same format or contents as the original tall table. For more information, see “Apache Parquet Data Type Mappings”.

Tips

- Use the `write` function to create *checkpoints* or *snapshots* of your data as you work, especially when working with huge data sets. This practice allows you to reconstruct tall arrays directly from files on disk rather than re-executing all of the commands that produced the tall array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`datastore` | `tall`

Topics

“Tall Arrays for Out-of-Memory Data”
“Work with Remote Data”

Introduced in R2016b

write

Write data to remote host over TCP/IP

Syntax

```
write(t,data)
```

Description

`write(t,data)` sends the N-dimensional matrix of data from `tcpclient` object `t` connected to the remote host. The function waits until the specified values are written to the remote host.

Examples

Read and Write uint8 Data

Create a TCP/IP object called `t`, connecting to a TCP/IP echo server, with Port of `7`. This requires you to have an `echotcpip` server running on Port 7.

```
t = tcpclient('localhost', 7)
```

```
t = 

   tcpclient with properties:

          Address: 'local host'
             Port: 7
          Timeout: 10
   BytesAvailable: 0
```

The `write` function synchronously writes data to the remote host connected to the `tcpclient` object. First specify the data, then write the data. The function waits until the specified number of values is written to the remote host.

Assign 10 bytes of `uint8` data to the variable `data`.

```
data = uint8(1:10)
```

```
data =

  1    2    3    4    5    6    7    8    9    10
```

Check the data.

```
whos data
```

```
Name      Size      Bytes      Class      Attributes

data      1x10         10      uint8
```

Write data to the echo server.

```
write(t, data)
```

Check that the data was written using the `BytesAvailable` property.

```
t.BytesAvailable
```

```
ans =

    10
```

Read the data from the server.

```
read(t)
```

```
ans =

  1    2    3    4    5    6    7    8    9    10
```

Close the connection between the TCP/IP client object and the remote host by clearing the object.

```
clear t
```

Input Arguments

data — Data to write to the remote host
1xN matrix of numeric data

Data to write to the remote host, specified as a 1xN matrix of numeric data.

Example: `write(t, data)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Topics

"Create a TCP/IP Connection"
"Configure Properties for TCP/IP Communication"
"Write and Read Data over TCP/IP Interface"

Introduced in R2014b

writecell

Write cell array to file

Syntax

```
writecell(C)
writecell(C,filename)
writecell( ___ ,Name,Value)
```

Description

`writecell(C)` writes cell array `C` to a comma delimited text file. The file name is the workspace name of the cell array, appended with the extension `.txt`. If `writecell` cannot construct the file name from the input cell array name, then it writes to the file `cell.txt`.

Each column of each variable in `C` becomes a column in the output file. The `writecell` function overwrites any existing file.

`writecell(C,filename)` writes to a file with the name and extension specified by `filename`.

`writecell` determines the file format based on the specified extension. The extension must be one of the following:

- `.txt`, `.dat`, or `.csv` for delimited text files
- `.xls`, `.xlsm`, or `.xlsx` for Excel spreadsheet files
- `.xlsb` for Excel spreadsheet files supported on systems with Excel for Windows

`writecell( ___ ,Name,Value)` writes the cell array to a file with additional options specified by one or more `Name,Value` pair arguments and can include any of the input arguments in previous syntaxes.

Examples

Write Cell Array to Text File

Create a cell array, write it to a comma-separated text file, and then write the cell array to another text file with a different delimiter character.

Create a simple cell array in the workspace.

```
C =  {1,2,3;
     'text',datetime('today'),hours(1)}
```

```
C = 2×3 cell array
    {[   1]}    {[           2]}    {[   3]}
    {'text'}    {[09-Jan-2019]}    {[1 hr]}
```

Write the cell array to a comma delimited text file and display the file contents. The `writecell` function outputs a text file named `C.txt`.

```
writecell(C)
type 'C.txt'
```

```
1,2,3
text,09-Jan-2019,1 hr
```

To write the same cell array to a text file with a different delimiter character, use the `'Delimiter'` name-value pair.

```
writecell(C,'C_tab.txt','Delimiter','tab')
type 'C_tab.txt'
```

```
1    2    3
text    09-Jan-2019    1 hr
```

Write Cell Array to Spreadsheet File

Create a cell array, write it to a spreadsheet file, and then read and display the contents of the file.

Create a cell array in the workspace.

```
C =  {1,2,3;
     'text',datetime('today'),hours(1)}
```

```
C = 2×3 cell array
    {[   1]}    {[          2]}    {[   3]}
    {'text'}    {[09-Jan-2019]}    {[1 hr]}
```

Write the cell array to a spreadsheet file.

```
writecell(C,'C.xls')
```

Read and display the matrix from `C.xls`.

```
readcell('C.xls')
```

```
ans = 2×3 cell array
    {[   1]}    {[          2]}    {[   3]}
    {'text'}    {[09-Jan-2019]}    {'1 hr'}
```

Write Cell Array to Specified Sheet and Range

Create a cell array and write it to a specified sheet and range in a spreadsheet file.

Create a cell array in the workspace.

```
C =  {1,2,3;
     'text',datetime('today'),hours(1)}
```

```
C = 2×3 cell array
    {[   1]}    {[          2]}    {[   3]}
    {'text'}    {[09-Jan-2019]}    {[1 hr]}
```

Write the cell array to the file `C.xls`, in the second worksheet in the file, starting at the third row.

```
writecell(C,'C.xls','Sheet',2,'Range','A3:C5')
```

Read and display the cell array.

```
readcell('C.xls','Sheet',2,'Range','A3:C5')
```

```
ans = 2×3 cell array
    {[   1]}    {[          2]}    {[   3]}
    {'text'}    {[09-Jan-2019]}    {'1 hr'}
```

Input Arguments

`C` — Input cell array
cell array

Input data, specified as a cell array.

`filename` — File name
character vector | string scalar

File name, specified as a character vector or string scalar.

Depending on the location you are writing to, `filename` can take on one of these forms.

Location	Form
Current folder	To write to the current folder, specify the name of the file in `filename`. **Example:** `'myTextFile.csv'`
Other folders	To write to a folder different from the current folder, specify the full or relative path name in `filename`. **Example:** `'C:\myFolder\myTextFile.csv'` **Example:** `'myFolder\myExcelFile.xlsx'`

<table>
<tr><th>Location</th><th>Form</th></tr>
<tr><td>Remote Location</td><td>To write to a remote location, <code>filename</code> must contain the full path of the file specified as a uniform resource locator (URL) of the form:

<code>scheme_name://path_to_file/my_file.ext</code>

Based on your remote location, <code>scheme_name</code> can be one of the values in this table.
<table>
<tr><th>Remote Location</th><th>scheme_name</th></tr>
<tr><td>Amazon S3</td><td>s3</td></tr>
<tr><td>Windows Azure Blob Storage</td><td>wasb, wasbs</td></tr>
<tr><td>HDFS</td><td>hdfs</td></tr>
</table>
For more information, see “Work with Remote Data”.

Example: <code>'s3://bucketname/path_to_file/my_file.xlsx'</code></td></tr>
</table>

- If `filename` includes the file extension, then the writing function determines the file format from the extension. Otherwise, the writing function creates a comma separated text file and appends the extension `.txt`. Alternatively, you can specify `filename` without the file’s extension, and then include the `'FileType'` name-value pair arguments to indicate the type of file.
- If `filename` does not exist, then the writing function creates the file.
- If `filename` is the name of an existing text file, then the writing function overwrites the file.
- If `filename` is the name of an existing spreadsheet file, then the writing function writes the data to the specified location, but does not overwrite any values outside the range of the input data.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'FileType',text` indicates that the variable names should not be included as the first row of the output file.

Text and Spreadsheet Files

`FileType` — Type of file
`'text'` | `'spreadsheet'`

Type of file, specified as the comma-separated pair consisting of `'FileType'` and a character vector or string containing `'text'` or `'spreadsheet'`.

The `'FileType'` name-value pair must be used with the `filename` input argument. You do not need to specify the `'FileType'` name-value pair argument if the `filename` input argument includes a standard file extension. The following standard file extensions are recognized by the writing function:

- `.txt`, `.dat`, or `.csv` for delimited text files
- `.xls`, `.xlsm`, or `.xlsx` for Excel spreadsheet files
- `.xlsb` for Excel spreadsheet files supported on systems with Excel for Windows

Example: `'FileType','spreadsheet'`

Data Types: `char` | `string`

`DateLocale` — Locale for writing dates
character vector | string scalar

Locale for writing dates, specified as the comma-separated pair consisting of `'DateLocale'` and a character vector or a string scalar. When writing `datetime` values to the file, use `DateLocale` to specify the locale in which `writetable` should write month and day-of-week names and abbreviations. The character vector or string takes the form *xx_YY*, where *xx* is a lowercase ISO 639-1 two-letter code indicating a language, and *YY* is an uppercase ISO 3166-1 alpha-2 code indicating a country. For a list of common values for the locale, see the `Locale` name-value pair argument for the `datetime` function.

The writing function ignores the `'DateLocale'` parameter value whenever dates can be written as Excel-formatted dates.

Example: `'DateLocale','ja_JP'`

Data Types: `char` | `string`

Text Files Only

`Delimiter` — Field delimiter character

character vector | string scalar

Field delimiter character, specified as the comma-separated pair consisting of `'Delimiter'` and a character vector or string scalar containing one of the following specifiers.

Specifier	Field Delimiter
`','` `'comma'`	Comma. This is the default behavior.
`' '` `'space'`	Space
`'\t'` `'tab'`	Tab
`';'` `'semi'`	Semicolon
`'|'` `'bar'`	Vertical bar

You can use the `'Delimiter'` name-value pair only for delimited text files.

Example: `'Delimiter','space'`

Data Types: `char` | `string`

`QuoteStrings` — Indicator for writing quoted text

`false` (default) | `true`

Indicator for writing quoted text, specified as the comma-separated pair consisting of `'QuoteStrings'` and either `false` or `true`. If `'QuoteStrings'` is `true`, then the writing function encloses the text in double quotation marks, and replaces any double-quote characters that appear as part of that text with two double-quote characters.

You can use the `'QuoteStrings'` name-value pair only with delimited text files.

`Encoding` — Character encoding scheme
`'UTF-8'` | `'ISO-8859-1'` | `'windows-1251'` | `'windows-1252'` | ...

Character encoding scheme associated with the file, specified as the comma-separated pair consisting of `'Encoding'` and `'system'` or a standard character encoding scheme name like one of the values in this table. When you do not specify any encoding or specify encoding as `'system'`, the writing function uses your system default encoding to write the file.

'Big5'	'ISO-8859-1'	'windows-874'
'Big5-HKSCS'	'ISO-8859-2'	'windows-949'
'CP949'	'ISO-8859-3'	'windows-1250'
'EUC-KR'	'ISO-8859-4'	'windows-1251'
'EUC-JP'	'ISO-8859-5'	'windows-1252'
'EUC-TW'	'ISO-8859-6'	'windows-1253'
'GB18030'	'ISO-8859-7'	'windows-1254'
'GB2312'	'ISO-8859-8'	'windows-1255'
'GBK'	'ISO-8859-9'	'windows-1256'
'IBM866'	'ISO-8859-11'	'windows-1257'
'KOI8-R'	'ISO-8859-13'	'windows-1258'
'KOI8-U'	'ISO-8859-15'	'US-ASCII'
	'Macintosh'	'UTF-8'
	'Shift_JIS'	

Example: `'system'` uses the system default encoding.

Data Types: `char` | `string`

Spreadsheet Files Only

`Sheet` — Worksheet to write to
character vector | string scalar | positive integer

Worksheet to write to, specified as the comma-separated pair consisting of `'Sheet'` and a character vector or a string scalar containing the worksheet name or a positive integer

indicating the worksheet index. The worksheet name cannot contain a colon (`:`). To determine the names of sheets in a spreadsheet file, use `sheets = sheetnames(filename)`. For more information, see `sheetnames`.

Specify the worksheet to write to by name or index:

- name — If the specified sheet name does not exist in the file, then the writing function adds a new sheet at the end of the worksheet collection.
- index — If the specified sheet index is an index larger than the number of worksheets, then the writing function appends empty sheets until the number of worksheets in the workbook equals the sheet index. The writing function also generates a warning indicating that it has added a new worksheet.

You can use the `'Sheet'` name-value pair only with spreadsheet files.

Example: `'Sheet',2`

Example: `'Sheet', 'MySheetName'`

Data Types: `char` | `string` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Range — Rectangular portion of worksheet to write to
character vector | string scalar

Rectangular portion of worksheet to write to, specified as the comma-separated pair consisting of `'Range'` and a character vector or string scalar in one of the following forms.

Form of the Value of Range	Description
`'Corner1'`	*`Corner1`* specifies the first cell of the region to write. The writing function writes the data starting at this cell. **Example:** `'Range','D2'`

Form of the Value of Range	Description
`'Corner1:Corner2'`	*`Corner1`* and *`Corner2`* are two opposing corners that define the region to write. For example, `'D2:H4'` represents the 3-by-5 rectangular region between the two corners `D2` and `H4` on the worksheet. The `'Range'` name-value pair argument is not case sensitive, and uses Excel A1 reference style (see Excel help). **Example:** `'Range','D2:H4'` • If the range you specify is smaller than the size of the input data, then the writing function writes only a subset of the input data that fits into the range. • If the range you specify is larger than the size of the input data, then the writing function leaves the remainder of the region as it is.

The `'Range'` name-value pair can only be used with Excel files.

Example: `'Range'`, `'A1:F10'`

Data Types: `char` | `string`

`UseExcel` — Flag to start instance of Microsoft Excel for Windows
`false` (default) | `true`

Flag to start an instance of Microsoft Excel for Windows when writing spreadsheet data, specified as the comma-separated pair consisting of `'UseExcel'` and either `true`, or `false`.

You can set the `'UseExcel'` parameter to one of these values:

- `true` — The writing function starts an instance of Microsoft Excel when writing the file.
- `false` — The writing function does not start an instance of Microsoft Excel when writing the file. When operating in this mode, functionality for writing differs in the support of file formats and interactive features, such as formulas and macros.

UseExcel	true	false
Supported file formats	.xls, .xlsx, .xlsm, .xltx, .xltm, .xlsb, .ods	.xls, .xlsx, .xlsm, .xltx, .xltm
Support for interactive features, such as formulas and macros	Yes	No

When writing to spreadsheet files on Windows platforms, if you want to start an instance of Microsoft Excel, then set the `'UseExcel'` parameter to `true`.

Algorithms

There are some instances where the `writecell` function creates a file that does not represent the input data exactly. You will notice this when you use the `readcell` function to read that file. The resulting data might not have the exact same format or contents as the original data. If you need to save your cell array and retrieve it at a later time to exactly match the original cell array, with the same data and organization, then save it as a MAT-file. `writecell` writes an inexact table in the following instances:

- `writecell` writes out numeric data using `long g` format, and categorical or character data as unquoted text.
- `writecell` writes out cell arrays that have more than two dimensions as two dimensional arrays, with the trailing dimensions collapsed.

See Also

`readcell` | `writematrix` | `writetable` | `writetimetable`

Introduced in R2019a

writematrix

Write a matrix to a file

Syntax

```
writematrix(A)
writematrix(A,filename)
writematrix( ___ ,Name,Value)
```

Description

`writematrix(A)` writes homogeneous array `A` to a comma delimited text file. The file name is the workspace variable name of the array, appended with the extension `.txt`. If `writematrix` cannot construct the file name from the array name, then it writes to the file `matrix.txt`.

Each column of each variable in `A` becomes a column in the output file. The `writematrix` function overwrites any existing file.

`writematrix(A,filename)` writes to a file with the name and extension specified by `filename`.

`writematrix` determines the file format based on the specified extension. The extension must be one of the following:

- `.txt`, `.dat`, or `.csv` for delimited text files
- `.xls`, `.xlsm`, or `.xlsx` for Excel spreadsheet files
- `.xlsb` for Excel spreadsheet files supported on systems with Excel for Windows

`writematrix( ___ ,Name,Value)` writes an array to a file with additional options specified by one or more `Name,Value` pair arguments and can include any of the input arguments in previous syntaxes.

Examples

Write Matrix to Text File

Create a matrix, write it to a comma-separated text file, and then write the matrix to another text file with a different delimiter character.

Create a matrix in the workspace.

```
M = magic(5)
```

```
M = 5×5

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

Write the matrix to a comma delimited text file and display the file contents. The `writematrix` function outputs a text file named `M.txt`.

```
writematrix(M)
type 'M.txt'
```

```
17,24,1,8,15
23,5,7,14,16
4,6,13,20,22
10,12,19,21,3
11,18,25,2,9
```

To write the same matrix to a text file with a different delimiter character, use the `'Delimiter'` name-value pair.

```
writematrix(M,'M_tab.txt','Delimiter','tab')
type 'M_tab.txt'
```

```
17    24    1    8    15
23    5    7    14    16
4    6    13    20    22
10    12    19    21    3
11    18    25    2    9
```

Write Matrix to Spreadsheet File

Create a matrix, write it to a spreadsheet file, and then read and display the contents of the file.

Create a matrix in the workspace.

```
M = magic(5)
```

```
M = 5×5

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

Write the matrix to a spreadsheet file.

```
writematrix(M,'M.xls')
```

Read and display the matrix from `M.xls`.

```
readmatrix('M.xls')
```

```
ans = 5×5

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

Write Matrix to Specified Sheet and Range

Create a matrix and write it to a specified sheet and range in a spreadsheet file.

Create a matrix in the workspace.

```
M = magic(5)
```

```
M = 5×5

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

Write the matrix to `M.xls`, to the second worksheet in the file, starting at the third row.

```
writematrix(M,'M.xls','Sheet',2,'Range','A3:E8')
```

Read and display the matrix.

```
readmatrix('M.xls','Sheet',2,'Range','A3:E8')
```

```
ans = 5×5

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

Input Arguments

A — Input data
matrix

Input data, specified as a matrix.

`filename` — File name
character vector | string scalar

File name, specified as a character vector or string scalar.

Depending on the location you are writing to, `filename` can take on one of these forms.

<table>
<tr><th>Location</th><th>Form</th></tr>
<tr><td>Current folder</td><td>To write to the current folder, specify the name of the file in <code>filename</code>.

Example: <code>'myTextFile.csv'</code></td></tr>
<tr><td>Other folders</td><td>To write to a folder different from the current folder, specify the full or relative path name in <code>filename</code>.

Example: <code>'C:\myFolder\myTextFile.csv'</code>

Example: <code>'myFolder\myExcelFile.xlsx'</code></td></tr>
<tr><td>Remote Location</td><td>To write to a remote location, <code>filename</code> must contain the full path of the file specified as a uniform resource locator (URL) of the form:

<code>scheme_name://path_to_file/my_file.ext</code>

Based on your remote location, scheme_name can be one of the values in this table.
<table>
<tr><th>Remote Location</th><th>scheme_name</th></tr>
<tr><td>Amazon S3</td><td>s3</td></tr>
<tr><td>Windows Azure Blob Storage</td><td>wasb, wasbs</td></tr>
<tr><td>HDFS</td><td>hdfs</td></tr>
</table>
For more information, see “Work with Remote Data”.

Example: <code>'s3://bucketname/path_to_file/my_file.xlsx'</code></td></tr>
</table>

- If `filename` includes the file extension, then the writing function determines the file format from the extension. Otherwise, the writing function creates a comma separated text file and appends the extension `.txt`. Alternatively, you can specify `filename` without the file’s extension, and then include the `'FileType'` name-value pair arguments to indicate the type of file.
- If `filename` does not exist, then the writing function creates the file.
- If `filename` is the name of an existing text file, then the writing function overwrites the file.

- If `filename` is the name of an existing spreadsheet file, then the writing function writes the data to the specified location, but does not overwrite any values outside the range of the input data.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'FileType',text` indicates that the variable names should not be included as the first row of the output file.

Text and Spreadsheet Files

`FileType` — Type of file
`'text'` | `'spreadsheet'`

Type of file, specified as the comma-separated pair consisting of `'FileType'` and a character vector or string containing `'text'` or `'spreadsheet'`.

The `'FileType'` name-value pair must be used with the `filename` input argument. You do not need to specify the `'FileType'` name-value pair argument if the `filename` input argument includes a standard file extension. The following standard file extensions are recognized by the writing function:

- `.txt`, `.dat`, or `.csv` for delimited text files
- `.xls`, `.xlsm`, or `.xlsx` for Excel spreadsheet files
- `.xlsb` for Excel spreadsheet files supported on systems with Excel for Windows

Example: `'FileType','spreadsheet'`

Data Types: `char` | `string`

`DateLocale` — Locale for writing dates
character vector | string scalar

Locale for writing dates, specified as the comma-separated pair consisting of `'DateLocale'` and a character vector or a string scalar. When writing `datetime` values

to the file, use `DateLocale` to specify the locale in which `writetable` should write month and day-of-week names and abbreviations. The character vector or string takes the form *xx_YY*, where *xx* is a lowercase ISO 639-1 two-letter code indicating a language, and *YY* is an uppercase ISO 3166-1 alpha-2 code indicating a country. For a list of common values for the locale, see the `Locale` name-value pair argument for the `datetime` function.

The writing function ignores the `'DateLocale'` parameter value whenever dates can be written as Excel-formatted dates.

Example: `'DateLocale','ja_JP'`

Data Types: `char` | `string`

Text Files Only

`Delimiter` — Field delimiter character

character vector | string scalar

Field delimiter character, specified as the comma-separated pair consisting of `'Delimiter'` and a character vector or string scalar containing one of the following specifiers.

Specifier	Field Delimiter
`','` `'comma'`	Comma. This is the default behavior.
`' '` `'space'`	Space
`'\t'` `'tab'`	Tab
`';'` `'semi'`	Semicolon
`'|'` `'bar'`	Vertical bar

You can use the `'Delimiter'` name-value pair only for delimited text files.

Example: `'Delimiter','space'`

Data Types: `char` | `string`

`QuoteStrings` — Indicator for writing quoted text
`false` (default) | `true`

Indicator for writing quoted text, specified as the comma-separated pair consisting of `'QuoteStrings'` and either `false` or `true`. If `'QuoteStrings'` is `true`, then the writing function encloses the text in double quotation marks, and replaces any double-quote characters that appear as part of that text with two double-quote characters.

You can use the `'QuoteStrings'` name-value pair only with delimited text files.

`Encoding` — Character encoding scheme
`'UTF-8'` | `'ISO-8859-1'` | `'windows-1251'` | `'windows-1252'` | ...

Character encoding scheme associated with the file, specified as the comma-separated pair consisting of `'Encoding'` and `'system'` or a standard character encoding scheme name like one of the values in this table. When you do not specify any encoding or specify encoding as `'system'`, the writing function uses your system default encoding to write the file.

'Big5'	'ISO-8859-1'	'windows-874'
'Big5-HKSCS'	'ISO-8859-2'	'windows-949'
'CP949'	'ISO-8859-3'	'windows-1250'
'EUC-KR'	'ISO-8859-4'	'windows-1251'
'EUC-JP'	'ISO-8859-5'	'windows-1252'
'EUC-TW'	'ISO-8859-6'	'windows-1253'
'GB18030'	'ISO-8859-7'	'windows-1254'
'GB2312'	'ISO-8859-8'	'windows-1255'
'GBK'	'ISO-8859-9'	'windows-1256'
'IBM866'	'ISO-8859-11'	'windows-1257'
'KOI8-R'	'ISO-8859-13'	'windows-1258'
'KOI8-U'	'ISO-8859-15'	'US-ASCII'
	'Macintosh'	'UTF-8'

	'Shift_JIS'	

Example: 'system' uses the system default encoding.

Data Types: char | string

Spreadsheet Files Only

Sheet — Worksheet to write to

character vector | string scalar | positive integer

Worksheet to write to, specified as the comma-separated pair consisting of 'Sheet' and a character vector or a string scalar containing the worksheet name or a positive integer indicating the worksheet index. The worksheet name cannot contain a colon (:). To determine the names of sheets in a spreadsheet file, use sheets = sheetnames(filename). For more information, see sheetnames.

Specify the worksheet to write to by name or index:

- name — If the specified sheet name does not exist in the file, then the writing function adds a new sheet at the end of the worksheet collection.
- index — If the specified sheet index is an index larger than the number of worksheets, then the writing function appends empty sheets until the number of worksheets in the workbook equals the sheet index. The writing function also generates a warning indicating that it has added a new worksheet.

You can use the 'Sheet' name-value pair only with spreadsheet files.

Example: 'Sheet',2

Example: 'Sheet', 'MySheetName'

Data Types: char | string | single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Range — Rectangular portion of worksheet to write to

character vector | string scalar

Rectangular portion of worksheet to write to, specified as the comma-separated pair consisting of 'Range' and a character vector or string scalar in one of the following forms.

Form of the Value of Range	Description
'*Corner1*'	*Corner1* specifies the first cell of the region to write. The writing function writes the data starting at this cell. **Example:** `'Range','D2'`
'*Corner1:Corner2*'	*Corner1* and *Corner2* are two opposing corners that define the region to write. For example, `'D2:H4'` represents the 3-by-5 rectangular region between the two corners `D2` and `H4` on the worksheet. The `'Range'` name-value pair argument is not case sensitive, and uses Excel A1 reference style (see Excel help). **Example:** `'Range','D2:H4'` • If the range you specify is smaller than the size of the input data, then the writing function writes only a subset of the input data that fits into the range. • If the range you specify is larger than the size of the input data, then the writing function leaves the remainder of the region as it is.

The `'Range'` name-value pair can only be used with Excel files.

Example: `'Range'`, `'A1:F10'`

Data Types: `char` | `string`

`UseExcel` — Flag to start instance of Microsoft Excel for Windows
`false` (default) | `true`

Flag to start an instance of Microsoft Excel for Windows when writing spreadsheet data, specified as the comma-separated pair consisting of `'UseExcel'` and either `true`, or `false`.

You can set the `'UseExcel'` parameter to one of these values:

- `true` — The writing function starts an instance of Microsoft Excel when writing the file.
- `false` — The writing function does not start an instance of Microsoft Excel when writing the file. When operating in this mode, functionality for writing differs in the support of file formats and interactive features, such as formulas and macros.

UseExcel	true	false
Supported file formats	`.xls, .xlsx, .xlsm, .xltx, .xltm, .xlsb, .ods`	`.xls, .xlsx, .xlsm, .xltx, .xltm`
Support for interactive features, such as formulas and macros	Yes	No

When writing to spreadsheet files on Windows platforms, if you want to start an instance of Microsoft Excel, then set the `'UseExcel'` parameter to `true`.

Algorithms

There are some instances where the `writematrix` function creates a file that does not represent the input data exactly. You will notice this when you use the `readmatrix` function to read that file. The resulting data might not have the exact same format or contents as the original array. If you need to save your array and retrieve it at a later time to match the original array exactly, with the same data and organization, then save it as a MAT-file. `writematrix` writes inexact data in the following instances:

- `writematrix` writes out numeric data using `long g` format, and categorical or character data as unquoted text.
- `writematrix` writes out arrays that have more than two dimensions as two dimensional arrays, with the trailing dimensions collapsed.

See Also

`readmatrix` | `writecell` | `writematrix` | `writetable` | `writetimetable`

Introduced in R2019a

writetable

Write table to file

Syntax

```
writetable(T)
writetable(T,filename)
writetable( ___ ,Name,Value)
```

Description

`writetable(T)` writes table `T` to a comma delimited text file. The file name is the workspace variable name of the table, appended with the extension `.txt`. If `writetable` cannot construct the file name from the input table name, then it writes to the file `table.txt`.

Each column of each variable in `T` becomes a column in the output file. The variable names of `T` become column headings in the first line of the file.

`writetable(T,filename)` writes to a file with the name and extension specified by `filename`.

`writetable` determines the file format based on the specified extension. The extension must be one of the following:

- `.txt`, `.dat`, or `.csv` for delimited text files
- `.xls`, `.xlsm`, or `.xlsx` for Excel spreadsheet files
- `.xlsb` for Excel spreadsheet files supported on systems with Excel for Windows

`writetable( ___ ,Name,Value)` writes the table to a file with additional options specified by one or more `Name,Value` pair arguments and can include any of the input arguments in previous syntaxes.

For example, you can specify whether to write the variable names as column headings in the output file.

Examples

Write Table to Text File

Create a table.

```
T = table(['M';'F';'M'],[45 45;41 32;40 34],...
    {'NY';'CA';'MA'},[true;false;false])
```

```
T=3×4 table
    Var1      Var2       Var3     Var4
    ____    ________    ______    _____

     M      45    45    {'NY'}    true
     F      41    32    {'CA'}    false
     M      40    34    {'MA'}    false
```

Write the table to a comma delimited text file and display the file contents.

```
writetable(T)
```

`writetable` outputs a text file named `T.txt`.

```
type 'T.txt'
```

```
Var1,Var2_1,Var2_2,Var3,Var4
M,45,45,NY,1
F,41,32,CA,0
M,40,34,MA,0
```

`writetable` appends a unique suffix to the variable name, `Var2`, above the two columns of corresponding data.

Write Table to Space-Delimited Text File

Create a table.

```
T = table(['M';'F';'M'],[45 45;41 32;40 34],...
    {'NY';'CA';'MA'},[true;false;false])
```

```
T=3×4 table
    Var1      Var2       Var3     Var4
    ____    ________    ______    _____

     M      45    45    {'NY'}    true
     F      41    32    {'CA'}    false
     M      40    34    {'MA'}    false
```

Write the table to a space-delimited text file named `myData.txt` and display the file contents.

```
writetable(T,'myData.txt','Delimiter',' ')
type 'myData.txt'
```

```
Var1 Var2_1 Var2_2 Var3 Var4
M 45 45 NY 1
F 41 32 CA 0
M 40 34 MA 0
```

`writetable` appends a unique suffix to the variable name, `Var2`, above the two columns of corresponding data.

Write Table to Text File Including Row Names

Create a table.

```
LastName = {'Smith';'Johnson';'Williams';'Jones';'Brown'};
Age = [38;43;38;40;49];
Height = [71;69;64;67;64];
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];

T = table(Age,Height,Weight,BloodPressure,...
    'RowNames',LastName)
```

```
T=5×4 table
                Age    Height    Weight    BloodPressure
                ___    ______    ______    _____________

    Smith       38       71       176       124     93
    Johnson     43       69       163       109     77
```

```
    Williams    38       64       131       125     83
    Jones       40       67       133       117     75
    Brown       49       64       119       122     80
```

Write the table, T, to a comma delimited text file, called `myPatientData.dat`, and display the file contents.

```
writetable(T,'myPatientData.dat','WriteRowNames',true)
type 'myPatientData.dat'
```

```
Row,Age,Height,Weight,BloodPressure_1,BloodPressure_2
Smith,38,71,176,124,93
Johnson,43,69,163,109,77
Williams,38,64,131,125,83
Jones,40,67,133,117,75
Brown,49,64,119,122,80
```

The first column, which contains the row names, has the column heading, Row. This is the first dimension name for the table from the property `T.Properties.DimensionNames`.

Write Foreign-Language Dates to Text File

Convert English dates in a table to German and write the table to file.

Create a table that contains a `datetime` array with dates in English. Create column vectors of numeric data to go with the dates.

```
D = datetime({'01-Jan-2014';'01-Feb-2014';'01-Mar-2014'});
D.Format = 'dd MMMM yyyy';
X1 = [20.2;21.6;20.7];
X2 = [100.5;102.7;99.8];
T = table(D,X1,X2)
```

```
T =

           D              X1      X2
    ________________     ____    _____

    01 January 2014      20.2    100.5
    01 February 2014     21.6    102.7
```

```
    01 March 2014       20.7     99.8
```

Write the table to a text file. Specify German for the locale of the dates using the `DateLocale` name-value pair argument, and display the dates in the text file.

```
writetable(T,'myfile.txt','DateLocale','de_DE');
type myfile.txt
```

```
D,X1,X2
01 Januar 2014,20.2,100.5
01 Februar 2014,21.6,102.7
01 März 2014,20.7,99.8
```

Write Foreign-Language Characters to Text Files

When your data contains foreign-language or non-ASCII characters, use the encoding parameter to ensure the file is written correctly. First, load the provided table into the workspace. Then, write the table to a file using the default encoding. Finally, write the table using `'UTF-8'` encoding and examine the result.

Load `Table_Japanese_Characters.mat` which contains a table T. A preview of the table is shown below. Notice that the table contains two columns with Japanese characters.

T

9x3 table

	1 A	2 B	3 C
1	"南西"	458.9772	"冬の嵐"
2	"南東"	530.1399	"冬の嵐"
3	"南東"	289.4035	"冬の嵐"
4	"西"	434.8054	"機器の故障"
5	"中西部"	186.4368	"深刻な嵐"
6	"西"	0	"攻撃"
7	"西"	231.2947	"機器の故障"
8	"西"	311.8607	"機器の故障"
9	"北東"	239.9328	"火災"

```
load('Table_Japanese_Characters.mat')
```

Write the table to a file. The `writetable` function uses your system default encoding when writing files. Results may differ based on your system settings. To examine the resulting file, read the table back into the workspace by using the `readtable` function. Notice that `writetable` did not succeed in writing columns (1 and 3) containing foreign-language characters.

```
writetable(T,'myTable.txt')
myTable = readtable('myTable.txt')
```

```
myTable=9×3 table
      A         B           C
    _____     ______     _______

    ''      458.98     ''
    ''      530.14     ''
    ''       289.4     ''
    ''       434.81     ''
    ''     186.44     ''
    ''            0     ''
    ''       231.29     ''
    ''       311.86     ''
    ''      239.93     ''
```

If your table contains foreign-language or non-ASCII characters, you must use the `'Encoding'` parameter to ensure your data is written correctly. Set `'Encoding'` to `'UTF-8'` which supports a wide range of foreign-language and non-ASCII characters. To examine the resulting file, read the table back into the workspace by using the `readtable` function. With the correct encoding parameter the `writetable` function is successful in writing the data.

```
writetable(T,'myTable_UTF8.txt','Encoding','UTF-8')
myTable_UTF8 = readtable('myTable_UTF8.txt','Encoding','UTF-8')
```

```
myTable_UTF8=9×3 table
       A          B             C
    _______    ______    ___________

    '南西'     458.98    '冬の嵐'
    '南東'     530.14    '冬の嵐'
    '南東'      289.4    '冬の嵐'
    '西'       434.81    '機器の故障'
    '中西部'    186.44    '深刻な嵐'
    '西'            0    '攻撃'
    '西'       231.29    '機器の故障'
    '西'       311.86    '機器の故障'
    '北東'     239.93    '火災'
```

Write Quoted Text to CSV File

Create a table.

```
T = table(['M';'F';'M'],[45;41;36],...
    {'New York, NY';'San Diego, CA';'Boston, MA'},[true;false;false])
```

```
T=3×4 table
    Var1    Var2          Var3           Var4
    ____    ____    _________________    _____

     M       45     {'New York, NY' }    true
     F       41     {'San Diego, CA'}    false
     M       36     {'Boston, MA'   }    false
```

Write the table to a comma-separated text file named `myData.csv` and view the file contents. Use the `'QuoteStrings'` name-value pair argument to ensure that the commas in the third column are not treated as delimiters.

```
writetable(T,'myData.csv','Delimiter',',','QuoteStrings',true)
type 'myData.csv'
```

```
Var1,Var2,Var3,Var4
"M",45,"New York, NY",1
"F",41,"San Diego, CA",0
"M",36,"Boston, MA",0
```

Write Table to Specific Sheet and Range in Spreadsheet

Create a table.

```
T = table(['M';'F';'M'],[45 45;41 32;40 34],...
    {'NY';'CA';'MA'},[true;false;false])
```

```
T=3×4 table
    Var1      Var2       Var3     Var4
    ____    ________    ______    _____

     M      45    45    {'NY'}    true
     F      41    32    {'CA'}    false
     M      40    34    {'MA'}    false
```

Write the table to a spreadsheet named `myData.xls`. Include the data on the first sheet in the 5-by-5 region with corners at `B2` and `F6`. You can change the worksheet to write to by specifying the index corresponding to the worksheet.

```
writetable(T,'myData.xls','Sheet',1,'Range','B2:F6')
```

Excel® fills the row of the spreadsheet from `B6` to `F6` with `#N/A` since the range specified is larger than the size of the input table `T`.

Write Tabular Data Containing Arbitrary Variable Names

You can write tabular data containing variable names that have any characters, including spaces and non-ASCII characters. Create a table with arbitrary variable names and write

the table to a text file. Then, read the tabular data back while preserving the original variable names.

Create a table containing three variables with arbitrary variable names. The first and third variable names contain spaces and non-ASCII characters.

```
LastName = {'Sanchez';'Johnson';'Li';'Diaz';'Brown'};
Age = [38;43;38;40;49];
Smoker = logical([1;0;1;0;1]);
varNames = {'Last Name','Age','Smoker (1 or 0)'};
T = table(LastName,Age,Smoker,'VariableNames',varNames)
```

```
T=5×3 table
     Last Name     Age    Smoker (1 or 0)
    ___________    ___    _______________

    {'Sanchez'}    38          true
    {'Johnson'}    43          false
    {'Li'     }    38          true
    {'Diaz'   }    40          false
    {'Brown'  }    49          true
```

Write the data to the file `'sampletable.txt'`.

```
writetable(T,'sampletable.txt')
```

Read the tabular data back using `readtable`. By default, MATLAB® converts any variable names that include spaces and non-ASCII characters into valid MATLAB® identifiers. For example, MATLAB® converts the variable names `'Last Name'` to `'LastName'`, and `'Smoker (1 or 0)'` to `'Smoker_1or0_'`. To read the tabular data while preserving variable names, set the `'PreserveVariableNames'` parameter to `true`.

```
T_preserve = readtable('sampletable.txt','PreserveVariableNames',true)
```

```
T_preserve=5×3 table
     Last Name     Age    Smoker (1 or 0)
    ___________    ___    _______________

    {'Sanchez'}    38            1
    {'Johnson'}    43            0
    {'Li'     }    38            1
    {'Diaz'   }    40            0
```

```
    {'Brown'  }    49            1
```

Input Arguments

T — Input data
table

Input data, specified as a table.

`filename` — File name
character vector | string scalar

File name, specified as a character vector or string scalar.

Depending on the location you are writing to, `filename` can take on one of these forms.

Location	Form
Current folder	To write to the current folder, specify the name of the file in `filename`. **Example:** `'myTextFile.csv'`
Other folders	To write to a folder different from the current folder, specify the full or relative path name in `filename`. **Example:** `'C:\myFolder\myTextFile.csv'` **Example:** `'myFolder\myExcelFile.xlsx'`

<table>
<tr><th>Location</th><th>Form</th></tr>
<tr><td>Remote Location</td><td>To write to a remote location, <code>filename</code> must contain the full path of the file specified as a uniform resource locator (URL) of the form:

<code>scheme_name://path_to_file/my_file.ext</code>

Based on your remote location, <code>scheme_name</code> can be one of the values in this table.
<table>
<tr><th>Remote Location</th><th>scheme_name</th></tr>
<tr><td>Amazon S3</td><td>s3</td></tr>
<tr><td>Windows Azure Blob Storage</td><td>wasb, wasbs</td></tr>
<tr><td>HDFS</td><td>hdfs</td></tr>
</table>
For more information, see “Work with Remote Data”.

Example: <code>'s3://bucketname/path_to_file/my_file.xlsx'</code></td></tr>
</table>

- If `filename` includes the file extension, then the writing function determines the file format from the extension. Otherwise, the writing function creates a comma separated text file and appends the extension `.txt`. Alternatively, you can specify `filename` without the file’s extension, and then include the `'FileType'` name-value pair arguments to indicate the type of file.
- If `filename` does not exist, then the writing function creates the file.
- If `filename` is the name of an existing text file, then the writing function overwrites the file.
- If `filename` is the name of an existing spreadsheet file, then the writing function writes the data to the specified location, but does not overwrite any values outside the range of the input data.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'WriteVariableNames',false` indicates that the variable names should not be included as the first row of the output file.

Text and Spreadsheet Files

`FileType` — Type of file

`'text'` | `'spreadsheet'`

Type of file, specified as the comma-separated pair consisting of `'FileType'` and a character vector or string containing `'text'` or `'spreadsheet'`.

The `'FileType'` name-value pair must be used with the `filename` input argument. You do not need to specify the `'FileType'` name-value pair argument if the `filename` input argument includes a standard file extension. The following standard file extensions are recognized by the writing function:

- `.txt`, `.dat`, or `.csv` for delimited text files
- `.xls`, `.xlsm`, or `.xlsx` for Excel spreadsheet files
- `.xlsb` for Excel spreadsheet files supported on systems with Excel for Windows

Example: `'FileType','spreadsheet'`

Data Types: `char` | `string`

`WriteVariableNames` — Indicator for writing variable names as column headings

`true` (default) | `false`

Indicator for writing variable names as column headings, specified as the comma-separated pair consisting of `'WriteVariableNames'` and either `true`, or `false`.

Indicator	Behavior
`true`	The writing function includes variable names as the column headings of the output. This is the default behavior.
`false`	The writing function does not include variable names in the output.

`WriteRowNames` — Indicator for writing row names in first column

`false` (default) | `true`

Indicator for writing row names in first column, specified as the comma-separated pair consisting of `'WriteRowNames'` and either `false` or `true`.

Indicator	Behavior
`false`	`writetable` does not include the row names from `T` in the output. This is the default behavior.
`true`	`writetable` includes the row names from `T` as the first column of the output. If both the `'WriteVariableNames'` and `'WriteRowNames'` logical indicators are `true`, then `writetable` uses the first dimension name from the property `T.Properties.DimensionNames` as the column heading for the first column of the output.

`DateLocale` — Locale for writing dates
character vector | string scalar

Locale for writing dates, specified as the comma-separated pair consisting of `'DateLocale'` and a character vector or a string scalar. When writing `datetime` values to the file, use `DateLocale` to specify the locale in which `writetable` should write month and day-of-week names and abbreviations. The character vector or string takes the form *xx_YY*, where *xx* is a lowercase ISO 639-1 two-letter code indicating a language, and *YY* is an uppercase ISO 3166-1 alpha-2 code indicating a country. For a list of common values for the locale, see the `Locale` name-value pair argument for the `datetime` function.

The writing function ignores the `'DateLocale'` parameter value whenever dates can be written as Excel-formatted dates.

Example: `'DateLocale','ja_JP'`

Data Types: `char` | `string`

Text Files Only

`Delimiter` — Field delimiter character
character vector | string scalar

Field delimiter character, specified as the comma-separated pair consisting of `'Delimiter'` and a character vector or string scalar containing one of the following specifiers.

Specifier	Field Delimiter

`','`
 	
`'comma'`
	Comma. This is the default behavior.

`' '`
 	
`'space'`
	Space

`'\t'`
 	
`'tab'`
	Tab

`';'`
 	
`'semi'`
	Semicolon

`'|'`
 	
`'bar'`
	Vertical bar

You can use the `'Delimiter'` name-value pair only for delimited text files.

Example: `'Delimiter','space'`

Data Types: `char` | `string`

`QuoteStrings` — Indicator for writing quoted text
`false` (default) | `true`

Indicator for writing quoted text, specified as the comma-separated pair consisting of `'QuoteStrings'` and either `false` or `true`. If `'QuoteStrings'` is `true`, then the writing function encloses the text in double quotation marks, and replaces any double-quote characters that appear as part of that text with two double-quote characters.

You can use the `'QuoteStrings'` name-value pair only with delimited text files.

`Encoding` — Character encoding scheme
`'UTF-8'` | `'ISO-8859-1'` | `'windows-1251'` | `'windows-1252'` | ...

Character encoding scheme associated with the file, specified as the comma-separated pair consisting of `'Encoding'` and `'system'` or a standard character encoding scheme name like one of the values in this table. When you do not specify any encoding or specify encoding as `'system'`, the writing function uses your system default encoding to write the file.

'Big5'	'ISO-8859-1'	'windows-874'
'Big5-HKSCS'	'ISO-8859-2'	'windows-949'
'CP949'	'ISO-8859-3'	'windows-1250'
'EUC-KR'	'ISO-8859-4'	'windows-1251'
'EUC-JP'	'ISO-8859-5'	'windows-1252'
'EUC-TW'	'ISO-8859-6'	'windows-1253'
'GB18030'	'ISO-8859-7'	'windows-1254'
'GB2312'	'ISO-8859-8'	'windows-1255'
'GBK'	'ISO-8859-9'	'windows-1256'
'IBM866'	'ISO-8859-11'	'windows-1257'
'KOI8-R'	'ISO-8859-13'	'windows-1258'
'KOI8-U'	'ISO-8859-15'	'US-ASCII'
	'Macintosh'	'UTF-8'
	'Shift_JIS'	

Example: `'system'` uses the system default encoding.

Data Types: `char` | `string`

Spreadsheet Files Only

Sheet — Worksheet to write to

character vector | string scalar | positive integer

Worksheet to write to, specified as the comma-separated pair consisting of `'Sheet'` and a character vector or a string scalar containing the worksheet name or a positive integer indicating the worksheet index. The worksheet name cannot contain a colon (`:`). To determine the names of sheets in a spreadsheet file, use `sheets = sheetnames(filename)`. For more information, see `sheetnames`.

Specify the worksheet to write to by name or index:

- name — If the specified sheet name does not exist in the file, then the writing function adds a new sheet at the end of the worksheet collection.
- index — If the specified sheet index is an index larger than the number of worksheets, then the writing function appends empty sheets until the number of worksheets in the

workbook equals the sheet index. The writing function also generates a warning indicating that it has added a new worksheet.

You can use the `'Sheet'` name-value pair only with spreadsheet files.

Example: `'Sheet',2`

Example: `'Sheet', 'MySheetName'`

Data Types: `char` | `string` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Range — Rectangular portion of worksheet to write to
character vector | string scalar

Rectangular portion of worksheet to write to, specified as the comma-separated pair consisting of `'Range'` and a character vector or string scalar in one of the following forms.

Form of the Value of Range	Description
`'`*`Corner1`*`'`	*`Corner1`* specifies the first cell of the region to write. The writing function writes the data starting at this cell. **Example:** `'Range','D2'`
`'`*`Corner1:Corner2`*`'`	*`Corner1`* and *`Corner2`* are two opposing corners that define the region to write. For example, `'D2:H4'` represents the 3-by-5 rectangular region between the two corners `D2` and `H4` on the worksheet. The `'Range'` name-value pair argument is not case sensitive, and uses Excel A1 reference style (see Excel help). **Example:** `'Range','D2:H4'` • If the range you specify is smaller than the size of the input data, then the writing function writes only a subset of the input data that fits into the range. • If the range you specify is larger than the size of the input data, then the writing function leaves the remainder of the region as it is.

The `'Range'` name-value pair can only be used with Excel files.

Example: `'Range'`, `'A1:F10'`

Data Types: `char` | `string`

`UseExcel` — Flag to start instance of Microsoft Excel for Windows
`false` (default) | `true`

Flag to start an instance of Microsoft Excel for Windows when writing spreadsheet data, specified as the comma-separated pair consisting of `'UseExcel'` and either `true`, or `false`.

You can set the `'UseExcel'` parameter to one of these values:

- `true` — The writing function starts an instance of Microsoft Excel when writing the file.
- `false` — The writing function does not start an instance of Microsoft Excel when writing the file. When operating in this mode, functionality for writing differs in the support of file formats and interactive features, such as formulas and macros.

`UseExcel`	`true`	`false`
Supported file formats	`.xls, .xlsx, .xlsm, .xltx, .xltm, .xlsb, .ods`	`.xls, .xlsx, .xlsm, .xltx, .xltm`
Support for interactive features, such as formulas and macros	Yes	No

When writing to spreadsheet files on Windows platforms, if you want to start an instance of Microsoft Excel, then set the `'UseExcel'` parameter to `true`.

Tips

- **Creating Empty Sheets in Spreadsheet Files:** Create an empty sheet in a spreadsheet file using `writetable` by writing an empty table. For example, create an empty sheet named `'MySheetName'`.

  ```
  writetable(table(),'empty.xls','Sheet','MySheetName')
  ```

  Or create three empty sheets called `'Sheet1'`, `'Sheet2'`, and `'Sheet3'`.

```
writetable(table(),'empty.xls','Sheet',3)
```

Algorithms

- If both the `'WriteVariableNames'` and `'WriteRowNames'` logical indicators are `true`, then the `writetable` function uses the first dimension name from the property `T.Properties.DimensionNames` as the column heading for the first column of the output.
- Excel converts `Inf` values to `65535`. MATLAB converts `NaN`, `NaT`, `<undefined>` categorical values, and `<missing>` string values to empty cells.
- For Excel files, `writetable` writes `table` variables containing `datetime` arrays as Excel dates. If the `table` contains `datetime` arrays with years prior to either 1900 or 1904, then `writetable` writes the variables as text. For more information on Excel dates, see `https://support.microsoft.com/en-us/kb/214330`.
- There are some instances where the `writetable` function creates a file that does not represent `T` exactly. You will notice this when you use `readtable` to read that file. The resulting table might not have the same format or contents as the original table. If you need to save a table and retrieve it at a later time to match the original table exactly, with the same data and organization, then save it as a MAT-file. `writetable` writes an inexact table in the following instances:
  - When writing to text files, `writetable` outputs numeric variables using `long g` format, and categorical or character variables as unquoted characters.
  - For variables that have more than one column, `writetable` appends a unique identifier to the variable name to use as the column headings.
  - For output variables that have more than two dimensions, `writetable` outputs these variables as two dimensional where the trailing dimensions are collapsed. For example, `writetable` outputs a 4-by-3-by-2 variable as if its size were 4-by-6.
  - For variables with a `cell` data type, `writetable` outputs the contents of each cell as a single row, in multiple fields. If the contents are other than numeric, logical, character, or categorical, then `writetable` outputs a single empty field.

See Also

`readtable` | `table` | `writecell` | `writematrix` | `writetimetable`

Introduced in R2013b

writetimetable

Write timetable to file

Syntax

```
writetimetable(TT)
writetimetable(TT,filename)
writetimetable( ___ ,Name,Value)
```

Description

`writetimetable(TT)` writes the timetable `TT` to a comma-delimited text file. The file name is the workspace variable name of the timetable, appended with the extension `.txt`. If `writetimetable` cannot construct the file name from the input timetable name, then it writes to the file `timetable.txt`.

Each column of each variable in `TT` becomes a column in the output file. The variable names of `TT` become column headings in the first line of the file. The `writetimetable` function overwrites any existing file.

`writetimetable(TT,filename)` writes to a file with the name and extension specified by `filename`.

`writetimetable` determines the file format based on the specified extension. The extension must be one of the following:

- `.txt`, `.dat`, or `.csv` for delimited text files
- `.xls`, `.xlsm`, or `.xlsx` for Excel spreadsheet files
- `.xlsb` for Excel spreadsheet files supported on systems with Excel for Windows

`writetimetable( ___ ,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments and can include any of the input arguments in previous syntaxes.

For example, you can specify whether to write the variable names as column headings in the output file.

Examples

Write Timetable to Text File

Create a timetable, write it to a comma-separated text file, and then write the timetable to another text file with a different delimiter character.

Define a timetable in the workspace with row times duration vector `RowTimes`.

```
RowTimes = seconds(1:5)';
TT = timetable(RowTimes,[98;97.5;97.9;98.1;97.9],[120;111;119;117;116],...
               'VariableNames',{'Reading1','Reading2'})
```

```
TT=5×2 timetable
    RowTimes    Reading1    Reading2
    ________    ________    ________

    1 sec           98        120
    2 sec         97.5        111
    3 sec         97.9        119
    4 sec         98.1        117
    5 sec         97.9        116
```

Write the timetable to a comma delimited text file and display the file contents. The `writetimetable` function outputs a text file named `TT.txt`.

```
writetimetable(TT)
type 'TT.txt'
```

```
RowTimes,Reading1,Reading2
1 sec,98,120
2 sec,97.5,111
3 sec,97.9,119
4 sec,98.1,117
5 sec,97.9,116
```

To write the same timetable to a text file with a different delimiter character, use the `'Delimiter'` name-value pair.

```
writetimetable(TT,'TT_bar.txt','Delimiter','bar')
type 'TT_bar.txt'
```

```
RowTimes|Reading1|Reading2
1 sec|98|120
2 sec|97.5|111
3 sec|97.9|119
4 sec|98.1|117
5 sec|97.9|116
```

Write Timetable to Spreadsheet File

Create a timetable, write it to a spreadsheet file, and then display the contents of the file.

Define a timetable in the workspace with row times as a `datetime` vector.

```
Y = [2014:2018]'; M = ones(5,1); D = ones(5,1);
RowTimes = datetime(Y,M,D); % Create Row Times
TT = timetable(RowTimes,[98;97.5;97.9;98.1;97.9],[120;111;119;117;116],...
               'VariableNames',{'Reading1','Reading2'})
```

```
TT=5×2 timetable
     RowTimes      Reading1    Reading2
    ___________    ________    ________

    01-Jan-2014        98        120
    01-Jan-2015      97.5        111
    01-Jan-2016      97.9        119
    01-Jan-2017      98.1        117
    01-Jan-2018      97.9        116
```

Write the table to spreadsheet file `'TT.xlsx'`. By default, the `writetimetable` function writes the first worksheet in the file. Use the `'Sheet'` name-value pair to specify the worksheet to write to.

```
writetimetable(TT,'TT.xlsx','Sheet',2)
```

Read and display the file contents.

```
readtimetable('TT.xlsx','Sheet',2)
```

```
ans=5×2 timetable
          RowTimes           Reading1    Reading2
    ____________________    ________    ________
```

```
    01-Jan-2014 00:00:00        98        120
    01-Jan-2015 00:00:00      97.5        111
    01-Jan-2016 00:00:00      97.9        119
    01-Jan-2017 00:00:00      98.1        117
    01-Jan-2018 00:00:00      97.9        116
```

Input Arguments

TT — Input timetable
timetable

Input timetable.

`filename` — File name
character vector | string scalar

File name, specified as a character vector or string scalar.

Depending on the location you are writing to, `filename` can take on one of these forms.

Location	Form
Current folder	To write to the current folder, specify the name of the file in `filename`. **Example:** `'myTextFile.csv'`
Other folders	To write to a folder different from the current folder, specify the full or relative path name in `filename`. **Example:** `'C:\myFolder\myTextFile.csv'` **Example:** `'myFolder\myExcelFile.xlsx'`

<table>
<tr><th>Location</th><th>Form</th></tr>
<tr><td>Remote Location</td><td>To write to a remote location, <code>filename</code> must contain the full path of the file specified as a uniform resource locator (URL) of the form:

<code>scheme_name://path_to_file/my_file.ext</code>

Based on your remote location, <code>scheme_name</code> can be one of the values in this table.
<table>
<tr><th>Remote Location</th><th><code>scheme_name</code></th></tr>
<tr><td>Amazon S3</td><td><code>s3</code></td></tr>
<tr><td>Windows Azure Blob Storage</td><td><code>wasb</code>, <code>wasbs</code></td></tr>
<tr><td>HDFS</td><td><code>hdfs</code></td></tr>
</table>
For more information, see “Work with Remote Data”.

Example: <code>'s3://bucketname/path_to_file/my_file.xlsx'</code></td></tr>
</table>

- If `filename` includes the file extension, then the writing function determines the file format from the extension. Otherwise, the writing function creates a comma separated text file and appends the extension `.txt`. Alternatively, you can specify `filename` without the file’s extension, and then include the `'FileType'` name-value pair arguments to indicate the type of file.
- If `filename` does not exist, then the writing function creates the file.
- If `filename` is the name of an existing text file, then the writing function overwrites the file.
- If `filename` is the name of an existing spreadsheet file, then the writing function writes the data to the specified location, but does not overwrite any values outside the range of the input data.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'WriteVariableNames',false` indicates that the variable names should not be included as the first row of the output file.

Text and Spreadsheet Files

`FileType` — Type of file
`'text'` | `'spreadsheet'`

Type of file, specified as the comma-separated pair consisting of `'FileType'` and a character vector or string containing `'text'` or `'spreadsheet'`.

The `'FileType'` name-value pair must be used with the `filename` input argument. You do not need to specify the `'FileType'` name-value pair argument if the `filename` input argument includes a standard file extension. The following standard file extensions are recognized by the writing function:

- `.txt`, `.dat`, or `.csv` for delimited text files
- `.xls`, `.xlsm`, or `.xlsx` for Excel spreadsheet files
- `.xlsb` for Excel spreadsheet files supported on systems with Excel for Windows

Example: `'FileType','spreadsheet'`

Data Types: `char` | `string`

`WriteVariableNames` — Indicator for writing variable names as column headings
`true` (default) | `false`

Indicator for writing variable names as column headings, specified as the comma-separated pair consisting of `'WriteVariableNames'` and either `true`, or `false`.

Indicator	Behavior
`true`	The writing function includes variable names as the column headings of the output. This is the default behavior.
`false`	The writing function does not include variable names in the output.

`DateLocale` — Locale for writing dates
character vector | string scalar

Locale for writing dates, specified as the comma-separated pair consisting of `'DateLocale'` and a character vector or a string scalar. When writing `datetime` values

to the file, use `DateLocale` to specify the locale in which `writetable` should write month and day-of-week names and abbreviations. The character vector or string takes the form *xx_YY*, where *xx* is a lowercase ISO 639-1 two-letter code indicating a language, and *YY* is an uppercase ISO 3166-1 alpha-2 code indicating a country. For a list of common values for the locale, see the `Locale` name-value pair argument for the `datetime` function.

The writing function ignores the `'DateLocale'` parameter value whenever dates can be written as Excel-formatted dates.

Example: `'DateLocale','ja_JP'`

Data Types: `char` | `string`

Text Files Only

`Delimiter` — Field delimiter character

character vector | string scalar

Field delimiter character, specified as the comma-separated pair consisting of `'Delimiter'` and a character vector or string scalar containing one of the following specifiers.

Specifier	Field Delimiter
`','` `'comma'`	Comma. This is the default behavior.
`' '` `'space'`	Space
`'\t'` `'tab'`	Tab
`';'` `'semi'`	Semicolon
`'|'` `'bar'`	Vertical bar

You can use the `'Delimiter'` name-value pair only for delimited text files.

Example: `'Delimiter','space'`

Data Types: `char` | `string`

`QuoteStrings` — Indicator for writing quoted text
`false` (default) | `true`

Indicator for writing quoted text, specified as the comma-separated pair consisting of `'QuoteStrings'` and either `false` or `true`. If `'QuoteStrings'` is `true`, then the writing function encloses the text in double quotation marks, and replaces any double-quote characters that appear as part of that text with two double-quote characters.

You can use the `'QuoteStrings'` name-value pair only with delimited text files.

`Encoding` — Character encoding scheme
`'UTF-8'` | `'ISO-8859-1'` | `'windows-1251'` | `'windows-1252'` | ...

Character encoding scheme associated with the file, specified as the comma-separated pair consisting of `'Encoding'` and `'system'` or a standard character encoding scheme name like one of the values in this table. When you do not specify any encoding or specify encoding as `'system'`, the writing function uses your system default encoding to write the file.

'Big5'	'ISO-8859-1'	'windows-874'
'Big5-HKSCS'	'ISO-8859-2'	'windows-949'
'CP949'	'ISO-8859-3'	'windows-1250'
'EUC-KR'	'ISO-8859-4'	'windows-1251'
'EUC-JP'	'ISO-8859-5'	'windows-1252'
'EUC-TW'	'ISO-8859-6'	'windows-1253'
'GB18030'	'ISO-8859-7'	'windows-1254'
'GB2312'	'ISO-8859-8'	'windows-1255'
'GBK'	'ISO-8859-9'	'windows-1256'
'IBM866'	'ISO-8859-11'	'windows-1257'
'KOI8-R'	'ISO-8859-13'	'windows-1258'
'KOI8-U'	'ISO-8859-15'	'US-ASCII'
	'Macintosh'	'UTF-8'

	'Shift_JIS'	

Example: 'system' uses the system default encoding.

Data Types: char | string

Spreadsheet Files Only

Sheet — Worksheet to write to

character vector | string scalar | positive integer

Worksheet to write to, specified as the comma-separated pair consisting of 'Sheet' and a character vector or a string scalar containing the worksheet name or a positive integer indicating the worksheet index. The worksheet name cannot contain a colon (:). To determine the names of sheets in a spreadsheet file, use sheets = sheetnames(filename). For more information, see sheetnames.

Specify the worksheet to write to by name or index:

- name — If the specified sheet name does not exist in the file, then the writing function adds a new sheet at the end of the worksheet collection.
- index — If the specified sheet index is an index larger than the number of worksheets, then the writing function appends empty sheets until the number of worksheets in the workbook equals the sheet index. The writing function also generates a warning indicating that it has added a new worksheet.

You can use the 'Sheet' name-value pair only with spreadsheet files.

Example: 'Sheet',2

Example: 'Sheet', 'MySheetName'

Data Types: char | string | single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Range — Rectangular portion of worksheet to write to

character vector | string scalar

Rectangular portion of worksheet to write to, specified as the comma-separated pair consisting of 'Range' and a character vector or string scalar in one of the following forms.

Form of the Value of Range	Description
'*Corner1*'	*Corner1* specifies the first cell of the region to write. The writing function writes the data starting at this cell. **Example:** `'Range','D2'`
'*Corner1:Corner2*'	*Corner1* and *Corner2* are two opposing corners that define the region to write. For example, `'D2:H4'` represents the 3-by-5 rectangular region between the two corners `D2` and `H4` on the worksheet. The `'Range'` name-value pair argument is not case sensitive, and uses Excel A1 reference style (see Excel help). **Example:** `'Range','D2:H4'` • If the range you specify is smaller than the size of the input data, then the writing function writes only a subset of the input data that fits into the range. • If the range you specify is larger than the size of the input data, then the writing function leaves the remainder of the region as it is.

The `'Range'` name-value pair can only be used with Excel files.

Example: `'Range'`, `'A1:F10'`

Data Types: `char` | `string`

`UseExcel` — Flag to start instance of Microsoft Excel for Windows
`false` (default) | `true`

Flag to start an instance of Microsoft Excel for Windows when writing spreadsheet data, specified as the comma-separated pair consisting of `'UseExcel'` and either `true`, or `false`.

You can set the `'UseExcel'` parameter to one of these values:

- `true` — The writing function starts an instance of Microsoft Excel when writing the file.
- `false` — The writing function does not start an instance of Microsoft Excel when writing the file. When operating in this mode, functionality for writing differs in the support of file formats and interactive features, such as formulas and macros.

UseExcel	true	false
Supported file formats	`.xls, .xlsx, .xlsm, .xltx, .xltm, .xlsb, .ods`	`.xls, .xlsx, .xlsm, .xltx, .xltm`
Support for interactive features, such as formulas and macros	Yes	No

When writing to spreadsheet files on Windows platforms, if you want to start an instance of Microsoft Excel, then set the `'UseExcel'` parameter to `true`.

Algorithms

There are some instances where the `writetimetable` function creates a file that does not represent the input data exactly. You will notice this when you use the `readtimetable` function to read that file. The resulting data might not have the exact same format or contents as the original data. If you need to save your timetable and retrieve it at a later time to exactly match the original timetable, with the same data and organization, then save it as a MAT-file. `writetimetable` writes an inexact table in the following instances:

- `writetimetable` writes out numeric data using `long g` format, and categorical or character data as unquoted text.
- `writetimetable` writes out variables that have more than two dimensions as two dimensional variables, with the trailing dimensions collapsed.
- For cell-valued variables, `writetimetable` writes out the variables differently based on the data type of the content in the cells.
  - If the content of the cell is of the data type numeric, text, logical, categorical, datetime, or duration, then the `writetimetable` function writes out the contents of each cell as a single row, in multiple, delimiter-separated fields.
  - Otherwise, the `writetimetable` function writes out a single empty field.

See Also

`readtimetable` | `writecell` | `writematrix` | `writetable`

Introduced in R2019a

write

Write entire image

Syntax

```
write(t,imageData)
write(t,Y,Cb,Cr)
```

Description

`write(t,imageData)` writes `imageData` to TIFF file associated with the `Tiff` object `t`. The `write` method breaks the data into strips or tiles, depending on the value of the `RowsPerStrip` tag, or the `TileLength` and `TileWidth` tags.

`write(t,Y,Cb,Cr)` writes the YCbCr component data to the TIFF file. Use this syntax only for images with a YCbCr photometric interpretation.

Examples

Write Image Data to TIFF File

Write a RGB image and multiple tag values that characterize the image to a new TIFF file.

Load the image data and display the image.

```
load('peppers_RGB.mat');
imshow(RGB);
title('Peppers Image (RGB)');
```

Create a `Tiff` object to write the image data to a TIFF file.

```
t = Tiff('myfile.tif','w');
```

Use the `tagstruct` structure to define multiple tag values that characterize the image.

```
tagstruct.ImageLength = size(RGB,1);
tagstruct.ImageWidth = size(RGB,2);
tagstruct.Photometric = Tiff.Photometric.RGB;
tagstruct.BitsPerSample = 8;
tagstruct.SamplesPerPixel = 3;
tagstruct.PlanarConfiguration = Tiff.PlanarConfiguration.Chunky;
tagstruct.Software = 'MATLAB';
```

Set the tag values for the `Tiff` object.

```
setTag(t,tagstruct)
```

Write image data to the file and then close the `Tiff` object.

```
write(t,RGB);
close(t);
```

Write YCbCr Image Data to TIFF File

Write a YCbCr image and multiple tag values that characterize the image to a new TIFF file.

Load and display the Y component of the image.

```
load('peppers_YCbCr.mat');
imshow(Y);
title('Peppers Image (Y Component)');
```

Create a `Tiff` object to write the image data to a TIFF file.

```
t = Tiff('myfile_YCbCr.tif','w');
```

Use the `tagstruct` structure to define multiple tag values that characterize the image.

```
tagstruct.ImageLength = size(Y,1);
tagstruct.ImageWidth = size(Y,2);
tagstruct.SampleFormat = 1; % uint
tagstruct.Photometric = Tiff.Photometric.YCbCr;
tagstruct.BitsPerSample = 8 ;
tagstruct.SamplesPerPixel = 3;
tagstruct.YCbCrSubSampling = [1,1];
```

```
tagstruct.Compression = Tiff.Compression.None;
tagstruct.PlanarConfiguration = Tiff.PlanarConfiguration.Chunky;
tagstruct.Software = 'MATLAB';
```

Set the tag values for the `Tiff` object.

```
setTag(t,tagstruct)
```

Write image data to the file, and then close the `Tiff` object.

```
write(t,Y,Cb,Cr)
close(t)
```

Input Arguments

`t` — Tiff object
`Tiff` object

`Tiff` object representing a TIFF file. Use the `Tiff` function to create the object.

`imageData` — Image data
numeric array

Image data, specified as a numeric array. For example, for an RGB image `imageData` is an `M`-by-`N`-by-`3` array. Where `M` and `N` are the number of rows and columns in the image, respectively.

Data Types: `double`

`Y` — Luma component
numeric array

Luma component of the image data, specified as a two-dimensional numeric array.

Data Types: `double`

`Cb` — Blue-difference chroma component
numeric array

Blue-difference chroma component of the image data, specified as a two-dimensional numeric array.

Data Types: `double`

`Cr` — Red-difference chroma component
numeric array

Red-difference chroma component of the image data, specified as a two-dimensional numeric array.

Data Types: `double`

Tips

- Writing TIFF images with certain combinations of photometric configuration and the number of samples per pixel is not recommended. The value of `SamplesPerPixel` must be equal to the sum of `Photometric` color channels and the `ExtraSamples` specified in the `Tiff` object.

See Also

`Tiff` | `writeDirectory` | `writeEncodedStrip` | `writeEncodedTile`

Topics

"Exporting to Images"

Introduced in R2009b

writeDirectory

Create new IFD and make it current IFD

Syntax

```
writeDirectory(t)
```

Description

`writeDirectory(t)` creates a new image file directory (IFD) and makes it the current IFD.

Examples

Add IFD to TIFF File

Create a new TIFF file and add two new IFDs to the file.

Create a `Tiff` object for a new file.

```
t = Tiff('myfile.tif','w');
```

Add a new IFD to the file and display the current directory number. You can use this new IFD to add image data and tags to the TIFF file.

```
writeDirectory(t);
currentDirectory(t)
```

```
ans = 1
```

Add another IFD to the file and display the current directory number. You can use this IFD to save an additional image to the same TIFF file.

```
writeDirectory(t);
currentDirectory(t)
```

```
ans = 2
```

Close the `Tiff` object.

```
close(t);
```

Input Arguments

`t` — Tiff object
`Tiff` object

`Tiff` object representing a TIFF file. Use the `Tiff` function to create the object.

Tips

- You do not need this function when creating TIFF files that contain only one image.

Algorithms

References

This function corresponds to the `TIFFWriteDirectory` function in the LibTIFF C API. To use this function, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`close` | `write`

Topics

"Importing Images"
"Exporting to Images"

Introduced in R2009b

writeEncodedStrip

Write data to specified strip

Syntax

```
writeEncodedStrip(t,stripNumber,imageData)
writeEncodedStrip(t,stripNumber,Y,Cb,Cr)
```

Description

`writeEncodedStrip(t,stripNumber,imageData)` writes `imageData` to the strip specified by `stripNumber` to the TIFF file associated with the `Tiff` object `t`.

`writeEncodedStrip(t,stripNumber,Y,Cb,Cr)` writes the YCbCr component data to the strip specified by `stripNumber` to the TIFF file associated with the `Tiff` object `t`. To use this syntax, you must set the value of the `YCbCrSubSampling` tag.

Examples

Write Image Strip to TIFF File

Read two strips from a TIFF file and write them to a new TIFF file in different positions.

Open a TIFF file with image data in stripped layout, get the image data and the number of strips in the image.

```
tr = Tiff('peppers_RGB_stripped.tif','r');
imageR = read(tr);
nStrips = numberOfStrips(tr)
```

```
nStrips = 6
```

Read the 2nd and 5th strips of the image.

```
stripTwo = readEncodedStrip(tr,2);
stripFive = readEncodedStrip(tr,5);
```

Create a `Tiff` object for a new file and copy the image and tag information from the first file.

```
tw = Tiff('write_strip.tif','w');
tagstruct.ImageLength = getTag(tr,'ImageLength');
tagstruct.ImageWidth =  getTag(tr,'ImageWidth');
tagstruct.Photometric = getTag(tr,'Photometric');
tagstruct.RowsPerStrip = getTag(tr,'RowsPerStrip');
tagstruct.BitsPerSample = getTag(tr,'BitsPerSample');
tagstruct.SamplesPerPixel = getTag(tr,'SamplesPerPixel');
tagstruct.PlanarConfiguration = getTag(tr,'PlanarConfiguration');
setTag(tw,tagstruct);
write(tw,imageR)
```

Write `stripFive` in the position for strip number `2` and `stripTwo` in the position for strip number `5`.

```
writeEncodedStrip(tw,2,stripFive);
writeEncodedStrip(tw,5,stripTwo);
```

Read and display the new image next to the original image.

```
imageW = read(tw);
subplot(121);
imshow(imageR);
title('Original Image')
subplot(122);
imshow(imageW);
title('Strips Shuffled Image')
```

Close the `Tiff` objects.

```
close(tr);
close(tw);
```

Write Image Strip to YCbCr TIFF File

Read two strips from a YCbCr TIFF file and write them to a new TIFF file in different positions.

Open a TIFF file containing YCbCr image data in stripped layout, get the image data and the number of strips in the image.

```
tr = Tiff('peppers_YCbCr_stripped.tif','r');
[Yr,Cbr,Crr] = read(tr);
nStrips = numberOfStrips(tr)
```

```
nStrips = 6
```

Read the 2nd and 5th strips of the image.

```
[Y2,Cb2,Cr2] = readEncodedStrip(tr,2);
[Y5,Cb5,Cr5] = readEncodedStrip(tr,5);
```

Create a `Tiff` object for a new file and copy the image and tag information from the first file.

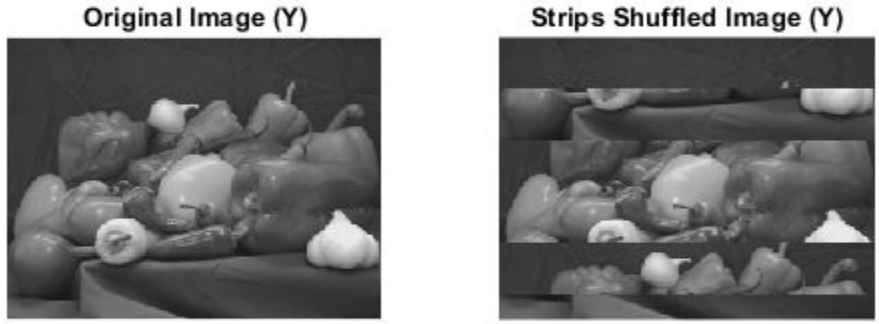
```
tw = Tiff('write_strip.tif','w');
tagstruct.ImageLength = getTag(tr,'ImageLength');
tagstruct.ImageWidth =  getTag(tr,'ImageWidth');
tagstruct.SampleFormat = getTag(tr,'SampleFormat');
tagstruct.Photometric = getTag(tr,'Photometric');
tagstruct.RowsPerStrip = getTag(tr,'RowsPerStrip');
tagstruct.BitsPerSample = getTag(tr,'BitsPerSample');
tagstruct.SamplesPerPixel = getTag(tr,'SamplesPerPixel');
tagstruct.YCbCrSubSampling = getTag(tr,'YCbCrSubSampling');
tagstruct.Compression = getTag(tr,'Compression');
tagstruct.PlanarConfiguration = getTag(tr,'PlanarConfiguration');
setTag(tw,tagstruct);
write(tw,Yr,Cbr,Crr)
```

Write strip number 5 in the position for strip number 2 and strip number 2 in the position for strip number 5.

```
writeEncodedStrip(tw,2,Y5,Cb5,Cr5);
writeEncodedStrip(tw,5,Y2,Cb2,Cr2);
```

Read and display the Y component of the new image next to the original image.

```
[Yw,Crw,Cbw] = read(tw);
subplot(121);
imshow(Yr);
title('Original Image (Y)')
subplot(122);
imshow(Yw);
title('Strips Shuffled Image (Y)')
```



Close the `Tiff` objects.

```
close(tr);
close(tw);
```

Input Arguments

t — Tiff object
`Tiff` object

`Tiff` object representing a TIFF file. Use the `Tiff` function to create the object.

`stripNumber` — Strip number
positive integer

Strip number, specified as a positive integer. Strip numbers are one-based numbers.

Example: 15

Data Types: `double`

`imageData` — Image data
numeric array

Image data, specified as a numeric array.

- If `imageData` has more number of bytes than the size of the strip, then `writeEncodedStrip` issues a warning and truncates the data.
- If `imageData` has fewer number of bytes than the size of the strip, then `writeEncodedStrip` silently pads the strip.

To determine the size of the strip, view the value of the `RowsPerStrip` tag.

Data Types: `double`

`Y` — Luma component
numeric array

Luma component of the image data, specified as a two-dimensional numeric array.

Data Types: `double`

`Cb` — Blue-difference chroma component
numeric array

Blue-difference chroma component of the image data, specified as a two-dimensional numeric array.

Data Types: `double`

`Cr` — Red-difference chroma component
numeric array

Red-difference chroma component of the image data, specified as a two-dimensional numeric array.

Data Types: `double`

Algorithms

References

This function corresponds to the `TIFFWriteEncodedStrip` function in the LibTIFF C API. To use this function, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff` | `write` | `writeEncodedTile`

Topics

"Importing Images"
"Exporting to Images"

Introduced in R2009b

writeEncodedTile

Write data to specified tile

Syntax

```
writeEncodedTile(t,tileNumber,imageData)
writeEncodedTile(t,tileNumber,Y,Cb,Cr)
```

Description

`writeEncodedTile(t,tileNumber,imageData)` writes the data in `imageData` to the tile specified by `tileNumber` to the TIFF file associated with the `Tiff` object `t`.

`writeEncodedTile(t,tileNumber,Y,Cb,Cr)` writes the YCbCr component data to the tile specified by `tileNumber` to the TIFF file associated with the `Tiff` object `t`. To use this syntax, you must set the `YCbCrSubSampling` tag.

Examples

Write Image Tile to TIFF File

Read two tiles from a TIFF file and write them to a new TIFF file in different positions.

Open a TIFF file with image data in tiled layout, get the image data and the number of tiles in the image.

```
tr = Tiff('peppers_RGB_tiled.tif','r');
imageR = read(tr);
nTiles = numberOfTiles(tr)
```

```
nTiles = 36
```

Read the 8th and 29th tiles of the image.

```
tile8 = readEncodedTile(tr,8);
tile29 = readEncodedTile(tr,29);
```

Create a `Tiff` object for a new file and copy the image and tag information from the first file.

```
tw = Tiff('write_tile.tif','w');
tagstruct.ImageLength = getTag(tr,'ImageLength');
tagstruct.ImageWidth =  getTag(tr,'ImageWidth');
tagstruct.Photometric = getTag(tr,'Photometric');
tagstruct.TileLength = getTag(tr,'TileLength');
tagstruct.TileWidth = getTag(tr,'TileWidth');
tagstruct.BitsPerSample = getTag(tr,'BitsPerSample');
tagstruct.SamplesPerPixel = getTag(tr,'SamplesPerPixel');
tagstruct.PlanarConfiguration = getTag(tr,'PlanarConfiguration');
setTag(tw,tagstruct);
write(tw,imageR)
```

Write `tile29` in the position for tile number 8 and `tile8` in the position for tile number 29.

```
writeEncodedTile(tw,8,tile29);
writeEncodedTile(tw,29,tile8);
```

Read and display the new image next to the original image.

```
imageW = read(tw);
subplot(121);
imshow(imageR);
title('Original Image')
subplot(122);
imshow(imageW);
title('Tiles Shuffled Image')
```

Close the `Tiff` objects.

```
close(tr);
close(tw);
```

Write Image Tile to YCbCr TIFF File

Read two tiles from a YCbCr TIFF file and write them to a new TIFF file in different positions.

Open a TIFF file with image data in tiled layout, get the image data and the number of tiles in the image.

```
tr = Tiff('peppers_YCbCr_tiled.tif','r');
[Yr,Cbr,Crr] = read(tr);
nTiles = numberOfTiles(tr)
```

```
nTiles = 36
```

Read the 8th and 29th tiles of the image.

```
[Y8,Cb8,Cr8] = readEncodedTile(tr,8);
[Y29,Cb29,Cr29] = readEncodedTile(tr,29);
```

Create a `Tiff` object for a new file and copy the image and tag information from the first file.

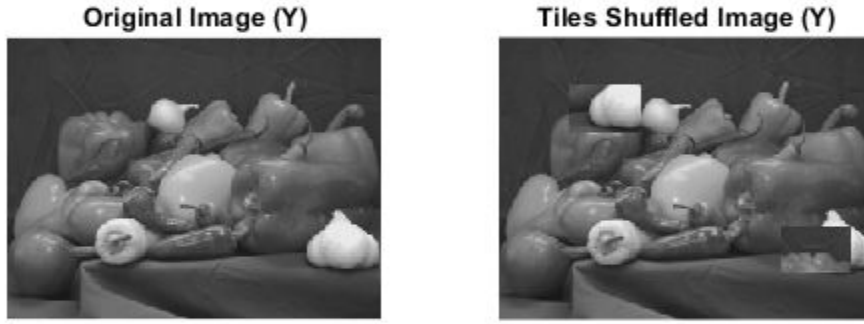
```
tw = Tiff('write_tile.tif','w');
tagstruct.ImageLength = getTag(tr,'ImageLength');
tagstruct.ImageWidth =  getTag(tr,'ImageWidth');
tagstruct.SampleFormat = getTag(tr,'SampleFormat');
tagstruct.Photometric = getTag(tr,'Photometric');
tagstruct.TileLength = getTag(tr,'TileLength');
tagstruct.TileWidth = getTag(tr,'TileWidth');
tagstruct.BitsPerSample = getTag(tr,'BitsPerSample');
tagstruct.SamplesPerPixel = getTag(tr,'SamplesPerPixel');
tagstruct.YCbCrSubSampling = getTag(tr,'YCbCrSubSampling');
tagstruct.Compression = getTag(tr,'Compression');
tagstruct.PlanarConfiguration = getTag(tr,'PlanarConfiguration');
setTag(tw,tagstruct);
write(tw,Yr,Cbr,Crr)
```

Write tile number 29 in the position for tile number 8 and tile number 8 in the position for tile number 29.

```
writeEncodedTile(tw,8,Y29,Cb29,Cr29);
writeEncodedTile(tw,29,Y8,Cb8,Cr8);
```

Read and display the Y component of the new image next to the original image.

```
[Yw,Crw,Cbw] = read(tw);
subplot(121);
imshow(Yr);
title('Original Image (Y)')
subplot(122);
imshow(Yw);
title('Tiles Shuffled Image (Y)')
```



Close the `Tiff` objects.

```
close(tr);
close(tw);
```

Input Arguments

t — Tiff object
`Tiff` object

`Tiff` object representing a TIFF file. Use the `Tiff` function to create the object.

`tileNumber` — Tile number
positive integer

Tile number, specified as a positive integer. Tile numbers are one-based numbers.

Example: 15

Data Types: `double`

`imageData` — Image data
numeric array

Image data, specified as a numeric array.

- If `imageData` has fewer number of bytes than the size of the tile, then `writeEncodedTile` silently pads the tile.
- If `imageData` has more bytes than the size of the tile, then `writeEncodedTile` issues a warning and truncates the data.

To see the size of the image tile, get the values of the `TileLength` and `TileWidth` tags.

Data Types: `double`

`Y` — Luma component
numeric array

Luma component of the image tile, specified as a two-dimensional numeric array.

Data Types: `double`

`Cb` — Blue-difference chroma component
numeric array

Blue-difference chroma component of the image tile, specified as a two-dimensional numeric array.

Data Types: `double`

`Cr` — Red-difference chroma component
numeric array

Red-difference chroma component of the image tile, specified as a two-dimensional numeric array.

Data Types: `double`

Algorithms

References

This function corresponds to the `TIFFWriteEncodedTile` function in the LibTIFF C API. To use this function, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff` | `write` | `writeEncodedStrip`

Topics

"Exporting Image Data and Metadata to TIFF Files"
"Reading Image Data and Metadata from TIFF Files"

Introduced in R2009b

xcorr

Cross-correlation

Syntax

```
r = xcorr(x,y)
r = xcorr(x)

r = xcorr( ___ ,maxlag)
r = xcorr( ___ ,scaleopt)

[r,lags] = xcorr( ___ )
```

Description

`r = xcorr(x,y)` returns the cross-correlation on page 1-17066 of two discrete-time sequences. Cross-correlation measures the similarity between a vector `x` and shifted (lagged) copies of a vector `y` as a function of the lag. If `x` and `y` have different lengths, the function appends zeros to the end of the shorter vector so it has the same length as the other.

`r = xcorr(x)` returns the autocorrelation sequence of `x`. If `x` is a matrix, then `r` is a matrix whose columns contain the autocorrelation and cross-correlation sequences for all combinations of the columns of `x`.

`r = xcorr( ___ ,maxlag)` limits the lag range from `-maxlag` to `maxlag` for either of the previous syntaxes.

`r = xcorr( ___ ,scaleopt)` also specifies a normalization option for the cross-correlation or autocorrelation. Any option other than `'none'` (the default) requires `x` and `y` to have the same length.

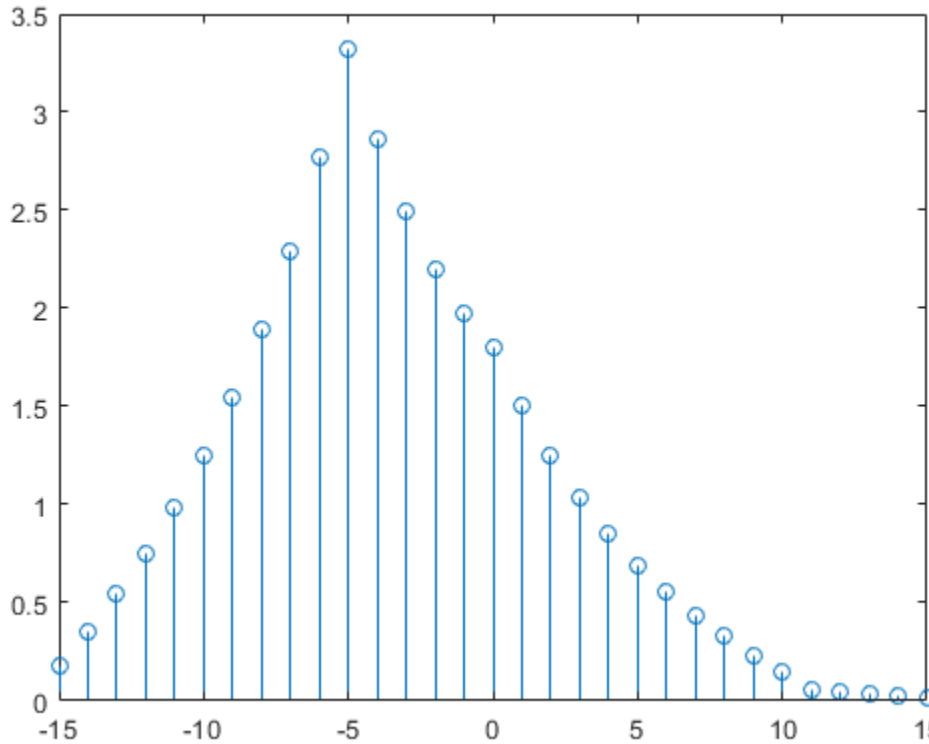
`[r,lags] = xcorr( ___ )` also returns the lags at which the correlations are computed.

Examples

Cross-Correlation of Two Vectors

Create a vector x and a vector y that is equal to x shifted by 5 elements to the right. Compute and plot the estimated cross-correlation of x and y. The largest spike occurs at the lag value when the elements of x and y match exactly (-5).

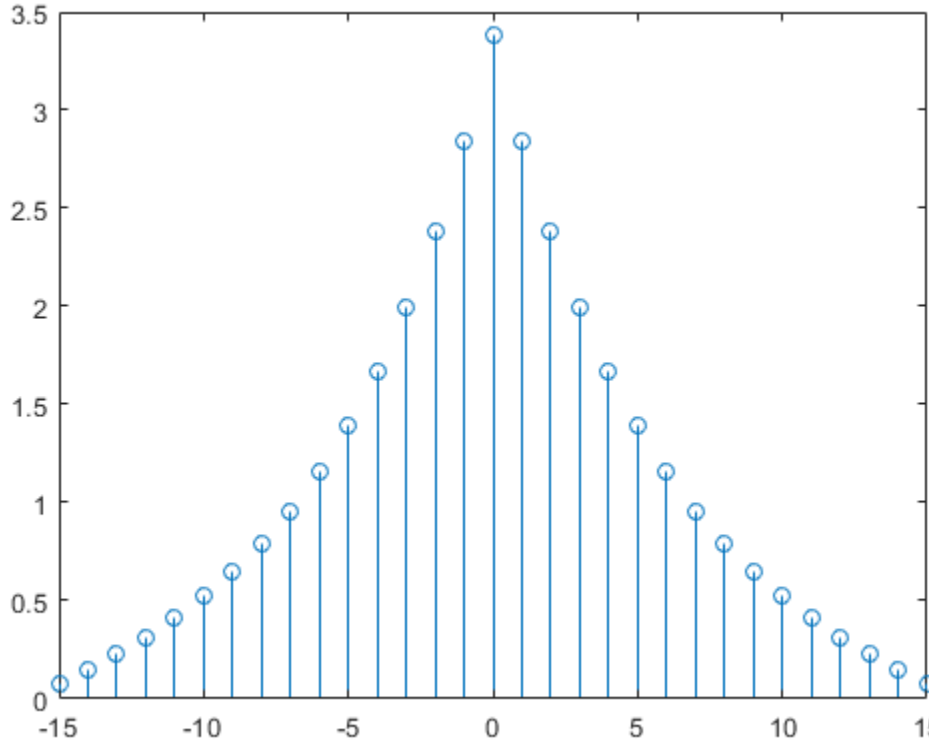
```
n = 0:15;
x = 0.84.^n;
y = circshift(x,5);
[c,lags] = xcorr(x,y);
stem(lags,c)
```



Autocorrelation of Vector

Compute and plot the estimated autocorrelation of a vector `x`. The largest spike occurs at zero lag, when `x` matches itself exactly.

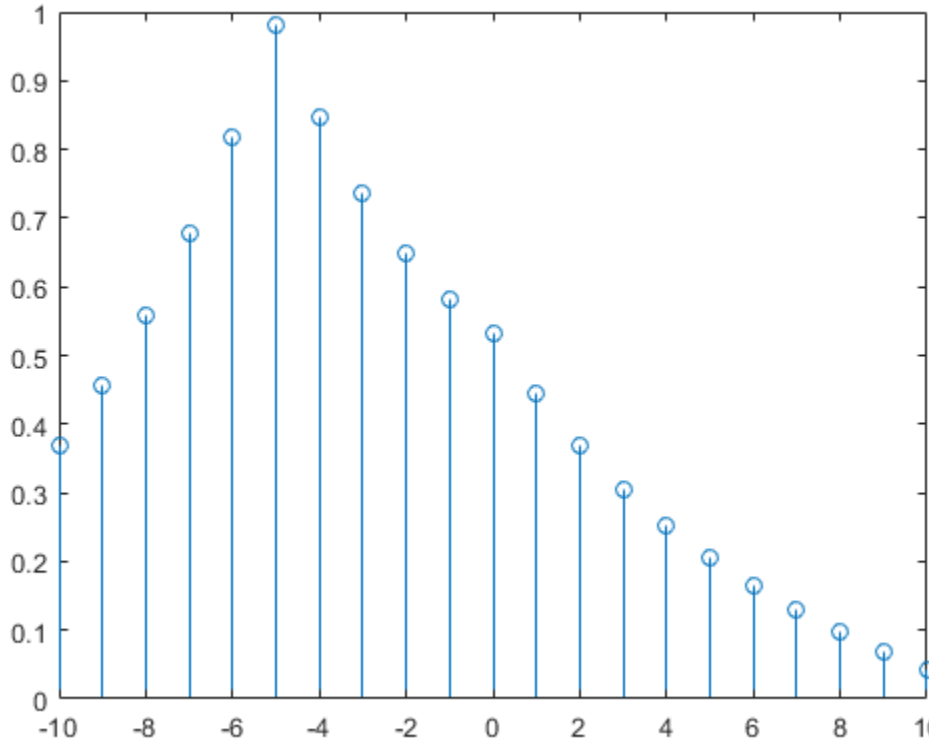
```
n = 0:15;
x = 0.84.^n;
[c,lags] = xcorr(x);
stem(lags,c)
```



Normalized Cross-Correlation

Compute and plot the normalized cross-correlation of vectors `x` and `y` with unity peak, and specify a maximum lag of `10`.

```
n = 0:15;
x = 0.84.^n;
y = circshift(x,5);
[c,lags] = xcorr(x,y,10,'normalized');
stem(lags,c)
```



Input Arguments

x — Input array
vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array. If `x` is a multidimensional array, then `xcorr` operates column-wise across all dimensions and returns each autocorrelation and cross-correlation as the columns of a matrix.

Data Types: `single` | `double`
Complex Number Support: Yes

y — Input array
vector

Input array, specified as a vector.

Data Types: `single` | `double`
Complex Number Support: Yes

`maxlag` — Maximum lag
integer scalar

Maximum lag, specified as an integer scalar. If you specify `maxlag`, the returned cross-correlation sequence ranges from `-maxlag` to `maxlag`. If you do not specify `maxlag`, the lag range equals $2N - 1$, where N is the greater of the lengths of `x` and `y`.

Data Types: `single` | `double`

`scaleopt` — Normalization option
`'none'` (default) | `'biased'` | `'unbiased'` | `'normalized'` | `'coeff'`

Normalization option, specified as one of the following.

- `'none'` — Raw, unscaled cross-correlation. `'none'` is the only valid option when `x` and `y` have different lengths.
- `'biased'` — Biased estimate of the cross-correlation:

$$\widehat{R}_{xy,\,\text{biased}}(m) = \frac{1}{N}\widehat{R}_{xy}(m)\,.$$

- `'unbiased'` — Unbiased estimate of the cross-correlation:

$$\widehat{R}_{xy,\,\text{unbiased}}(m) = \frac{1}{N - |m|}\widehat{R}_{xy}(m)\,.$$

- `'normalized'` or `'coeff'` — Normalizes the sequence so that the autocorrelations at zero lag equal 1:

$$\widehat{R}_{xy,\,\text{coeff}}(m) = \frac{1}{\sqrt{\widehat{R}_{xx}(0)\widehat{R}_{yy}(0)}}\widehat{R}_{xy}(m)\,.$$

Output Arguments

`r` — Cross-correlation or autocorrelation
vector | matrix

Cross-correlation or autocorrelation, returned as a vector or matrix.

If `x` is an $M \times N$ matrix, then `xcorr(x)` returns a $(2M - 1) \times N^2$ matrix with the autocorrelations and cross-correlations of the columns of `x`. If you specify `maxlag`, then `r` has size $(2 \times \texttt{maxlag} + 1) \times N^2$.

For example, if `S` has three columns, $\mathrm{S} = \begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix}$, then the result of `R = xcorr(S)` is organized as

$$\mathrm{R} = \begin{pmatrix} R_{x_1x_1} & R_{x_1x_2} & R_{x_1x_3} & R_{x_2x_1} & R_{x_2x_2} & R_{x_2x_3} & R_{x_3x_1} & R_{x_3x_2} & R_{x_3x_3} \end{pmatrix}.$$

`lags` — Lag indices
vector

Lag indices, returned as a vector.

More About

Cross-Correlation and Autocorrelation

The result of `xcorr` can be interpreted as an estimate of the correlation between two random sequences or as the deterministic correlation between two deterministic signals.

The true cross-correlation sequence of two jointly stationary random processes, x_n and y_n, is given by

$$R_{xy}(m) = E\left\{x_{n+m}y_n^*\right\} = E\left\{x_n y_{n-m}^*\right\},$$

where $-\infty < n < \infty$, the asterisk denotes complex conjugation, and E is the expected value operator. `xcorr` can only estimate the sequence because, in practice, only a finite segment of one realization of the infinite-length random process is available.

By default, `xcorr` computes raw correlations with no normalization:

$$\widehat{R}_{xy}(m) = \begin{cases} \sum_{n=0}^{N-m-1} x_{n+m} y_n^*, & m \geq 0, \\ \widehat{R}_{yx}^*(-m), & m < 0. \end{cases}$$

The output vector, `c`, has elements given by

$$\mathrm{c(m)} = \widehat{R}_{xy}(m - N), \quad m = 1, 2, \ldots, 2N - 1.$$

In general, the correlation function requires normalization to produce an accurate estimate. You can control the normalization of the correlation by using the input argument `scaleopt`.

References

[1] Buck, John R., Michael M. Daniel, and Andrew C. Singer. *Computer Explorations in Signals and Systems Using MATLAB*. 2nd Edition. Upper Saddle River, NJ: Prentice Hall, 2002.

[2] Stoica, Petre, and Randolph Moses. *Spectral Analysis of Signals*. Upper Saddle River, NJ: Prentice Hall, 2005.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- The `x` input must be a tall column vector.
- The `y` input must be a non-tall vector.
- The syntax `xcorr(x)` is not supported.
- The `scaleopt` option is not supported.
- If you specify `maxlag`, then it must satisfy `maxlag <= max(numel(x),numel(y))-1`.

- The `lags` output is returned as a tall column vector.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Leading ones in `size(x)` (dimension lengths of 1 before the first dimension length not equal to 1) must be constant for every input `x`. If `x` is variable-size and is a row vector, it must be 1-by-`:`. It cannot be `:`-by-`:` with `size(x,1) = 1` at run time.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

For example, create a `gpuArray` object from a signal `x` and compute the normalized autocorrelation.

```
t = 0:0.001:10-0.001;
x = cos(2*pi*10*t) + randn(size(t));
X = gpuArray(x);
[r,lags] = xcorr(X,200,'normalized');
r = gather(r);
```

See Also

`conv` | `corrcoef` | `cov` | `xcov`

xcov

Cross-covariance

Syntax

```
c = xcov(x,y)
c = xcov(x)

c = xcov( ___ ,maxlag)
c = xcov( ___ ,scaleopt)

[c,lags] = xcov( ___ )
```

Description

`c = xcov(x,y)` returns the cross-covariance on page 1-17076 of two discrete-time sequences. Cross-covariance measures the similarity between a vector `x` and shifted (lagged) copies of a vector `y` as a function of the lag. If `x` and `y` have different lengths, the function appends zeros to the end of the shorter vector so it has the same length as the other.

`c = xcov(x)` returns the autocovariance on page 1-17076 sequence of `x`. If `x` is a matrix, then `c` is a matrix whose columns contain the autocovariance and cross-covariance sequences for all combinations of the columns of `x`.

`c = xcov( ___ ,maxlag)` sets the lag range from `-maxlag` to `maxlag` for either of the previous syntaxes.

`c = xcov( ___ ,scaleopt)` also specifies a normalization option for the cross-covariance or autocovariance. Any option other than `'none'` (the default) requires the inputs `x` and `y` to have the same length.

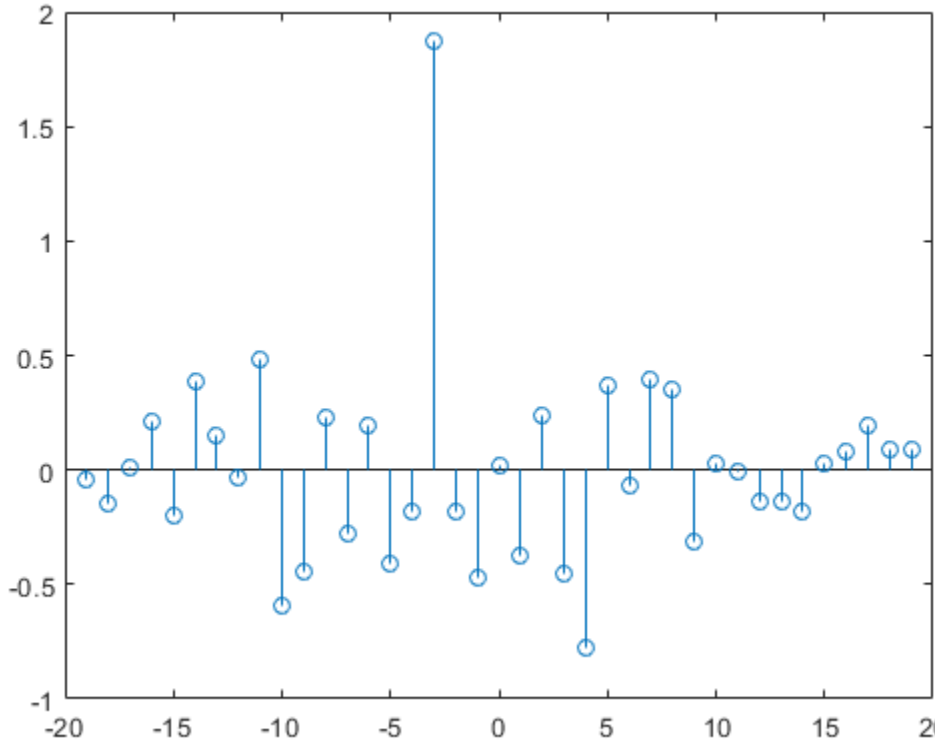
`[c,lags] = xcov( ___ )` also returns the lags at which the covariances are computed.

Examples

Cross-Covariance of Two Random Vectors

Create a vector of random numbers x and a vector y that is equal to x shifted by 3 elements to the right. Compute and plot the estimated cross-covariance of x and y. The largest spike occurs at the lag value when the elements of x and y match exactly (-3).

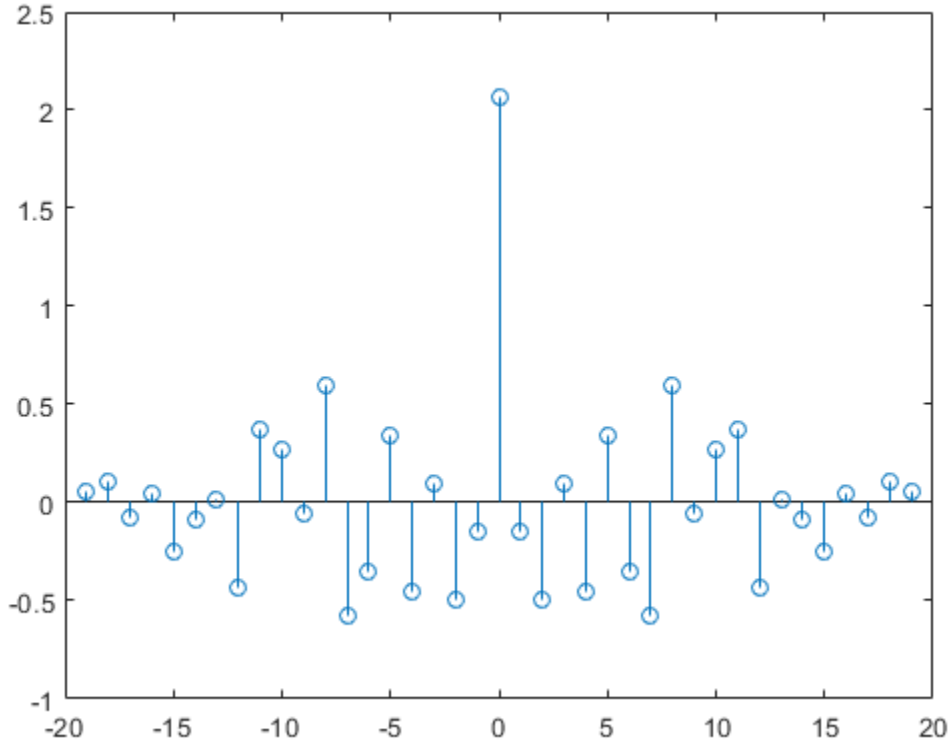
```
rng default
x = rand(20,1);
y = circshift(x,3);
[c,lags] = xcov(x,y);
stem(lags,c)
```



Autocovariance of Random Vector

Create a random vector, then compute and plot the estimated autocovariance. The largest spike occurs at zero lag, where the vector is exactly equal to itself.

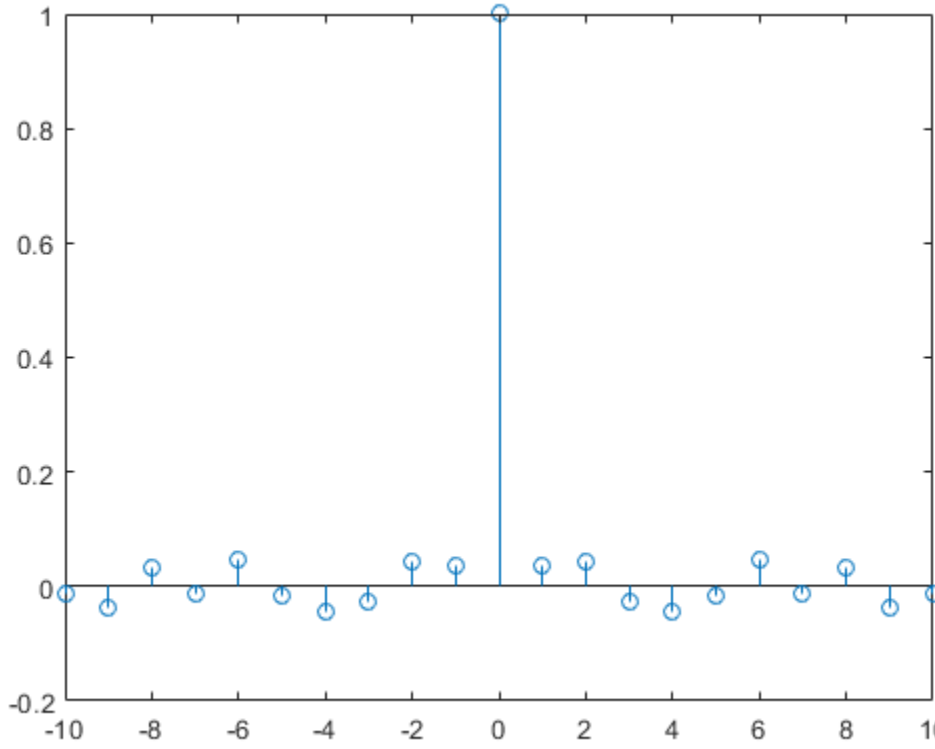
```
rng default
x = rand(20,1);
[c,lags] = xcov(x);
stem(lags,c)
```



Normalized Autocovariance of Noise

Compute and plot the estimated autocovariance of white Gaussian noise, $c(m)$, for $-10 \le m \le 10$. Normalize the sequence so that it is unity at zero lag.

```
rng default
x = randn(1000,1);
maxlag = 10;
[c,lags] = xcov(x,maxlag,'normalized');
stem(lags,c)
```



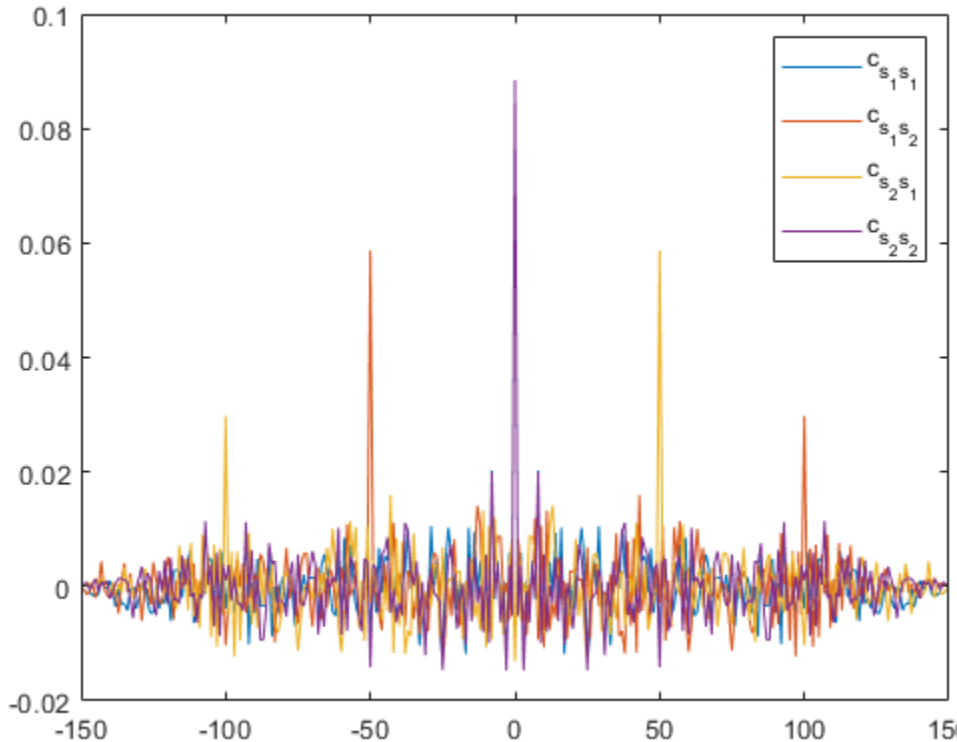
Biased Cross-Covariance of Two Shifted Signals

Create a signal made up of two signals that are circularly shifted from each other by 50 samples.

```
rng default
shft = 50;
s1 = rand(150,1);
s2 = circshift(s1,[shft 0]);
x = [s1 s2];
```

Compute and plot biased estimates of the autocovariance and mutual cross-covariance sequences. The output matrix c is organized as four column vectors such that c = $\left(c_{s_1s_1}\ c_{s_1s_2}\ c_{s_2s_1}\ c_{s_2s_2}\right)$. $c_{s_1s_2}$ has maxima at -50 and +100 and $c_{s_2s_1}$ has maxima at +50 and -100 as a result of the circular shift.

```
[c,lags] = xcov(x,'biased');
plot(lags,c)
legend('c_{s_1s_1}','c_{s_1s_2}','c_{s_2s_1}','c_{s_2s_2}')
```



Input Arguments

x — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array. If `x` is a multidimensional array, then `xcov` operates column-wise across all dimensions and returns each autocovariance and cross-covariance as the columns of a matrix.

Data Types: `single` | `double`
Complex Number Support: Yes

y — Input array

vector

Input array, specified as a vector.

Data Types: `single` | `double`
Complex Number Support: Yes

maxlag — Maximum lag

integer scalar

Maximum lag, specified as an integer scalar. If you specify `maxlag`, the returned cross-covariance sequence ranges from `-maxlag` to `maxlag`. By default, the lag range equals $2N - 1$, where N is the greater of the lengths of inputs `x` and `y`.

Data Types: `single` | `double`

scaleopt — Normalization option

`'none'` (default) | `'biased'` | `'unbiased'` | `'normalized'` | `'coeff'`

Normalization option, specified as one of the following.

- `'none'` — Raw, unscaled cross-covariance. `'none'` is the only valid option when inputs `x` and `y` have different lengths.
- `'biased'` — Biased estimate of the cross-covariance.
- `'unbiased'` — Unbiased estimate of the cross-covariance.
- `'normalized'` or `'coeff'` — Normalizes the sequence so that the autocovariances at zero lag equal 1.

Output Arguments

c — Cross-covariance or autocovariance
vector | matrix

Cross-covariance or autocovariance, returned as a vector or matrix.

If `x` is an $M \times N$ matrix, then `xcov(x)` returns a $(2M - 1) \times N^2$ matrix with the autocovariances and cross-covariances of the columns of `x`. If you specify a maximum lag `maxlag`, then the output `c` has size $(2 \times \texttt{maxlag} + 1) \times N^2$.

For example, if `S` has three columns, $S = \begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix}$, then the result of `C = xcov(S)` is organized as

$$c = \begin{pmatrix} c_{x_1x_1} & c_{x_1x_2} & c_{x_1x_3} & c_{x_2x_1} & c_{x_2x_2} & c_{x_2x_3} & c_{x_3x_1} & c_{x_3x_2} & c_{x_3x_3} \end{pmatrix}.$$

`lags` — Lag indices
vector

Lag indices, returned as a vector.

More About

Cross-Covariance and Autocovariance

`xcov` computes the mean of its inputs, subtracts the mean, and then calls `xcorr`.

The result of `xcov` can be interpreted as an estimate of the covariance between two random sequences or as the deterministic covariance between two deterministic signals.

The true cross-covariance sequence of two jointly stationary random processes, x_n and y_n, is the cross-correlation of mean-removed sequences,

$$\phi_{xy}(m) = E\left\{(x_{n+m} - \mu_x)(y_n - \mu_y)^*)\right\},$$

where μ_x and μ_y are the mean values of the two stationary random processes, the asterisk denotes complex conjugation, and E is the expected value operator. `xcov` can only estimate the sequence because, in practice, only a finite segment of one realization of the infinite-length random process is available.

By default, `xcov` computes raw covariances with no normalization:

$$c_{xy}(m) = \begin{cases} \sum_{n=0}^{N-m-1} \left(x_{n+m} - \frac{1}{N} \sum_{i=0}^{N-1} x_i \right) \left(y_n^* - \frac{1}{N} \sum_{i=0}^{N-1} y_i^* \right), & m \geq 0, \\ c_{yx}^*(-m), & m < 0. \end{cases}$$

The output vector `c` has elements given by

$$\mathrm{c(m)} = c_{xy}(m - N), \quad m = 1, \ldots, 2N - 1.$$

The covariance function requires normalization to estimate the function properly. You can control the normalization of the correlation by using the input argument `scaleopt`.

References

[1] Orfanidis, Sophocles J. *Optimum Signal Processing: An Introduction*. 2nd Edition. New York: McGraw-Hill, 1996.

[2] Larsen, Jan. “Correlation Functions and Power Spectra.” November, 2009. http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/4932/pdf/imm4932.pdf

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- The `x` input must be a tall column vector.
- The `y` input must be a non-tall vector.
- The syntax `xcov(x)` is not supported.
- The `scaleopt` option is not supported.
- If you specify `maxlag`, then it must satisfy `maxlag <= max(numel(x),numel(y))-1`.

- The `lags` output is returned as a tall column vector.

For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

For example, create a `gpuArray` object from a signal `x` and compute the estimated autocovariance.

```
t = 0:0.001:10-0.001;
x = cos(2*pi*10*t) + randn(size(t));
X = gpuArray(x);
[c,lags] = xcov(X,200);
c = gather(c);
```

See Also

`conv` | `corrcoef` | `cov` | `xcorr`

xlabel

Label x-axis

Syntax

```
xlabel(txt)
xlabel(target,txt)

xlabel( ___ ,Name,Value)
t = xlabel( ___ )
```

Description

`xlabel(txt)` labels the *x*-axis of the current axes or chart returned by the `gca` command. Reissuing the `xlabel` command replaces the old label with the new label.

`xlabel(target,txt)` adds the label to the specified target object.

`xlabel( ___ ,Name,Value)` modifies the label appearance using one or more name-value pair arguments. For example, `'FontSize',12` sets the font size to 12 points. Specify name-value pair arguments after all other input arguments. Modifying the label appearance is not supported for all types of charts.

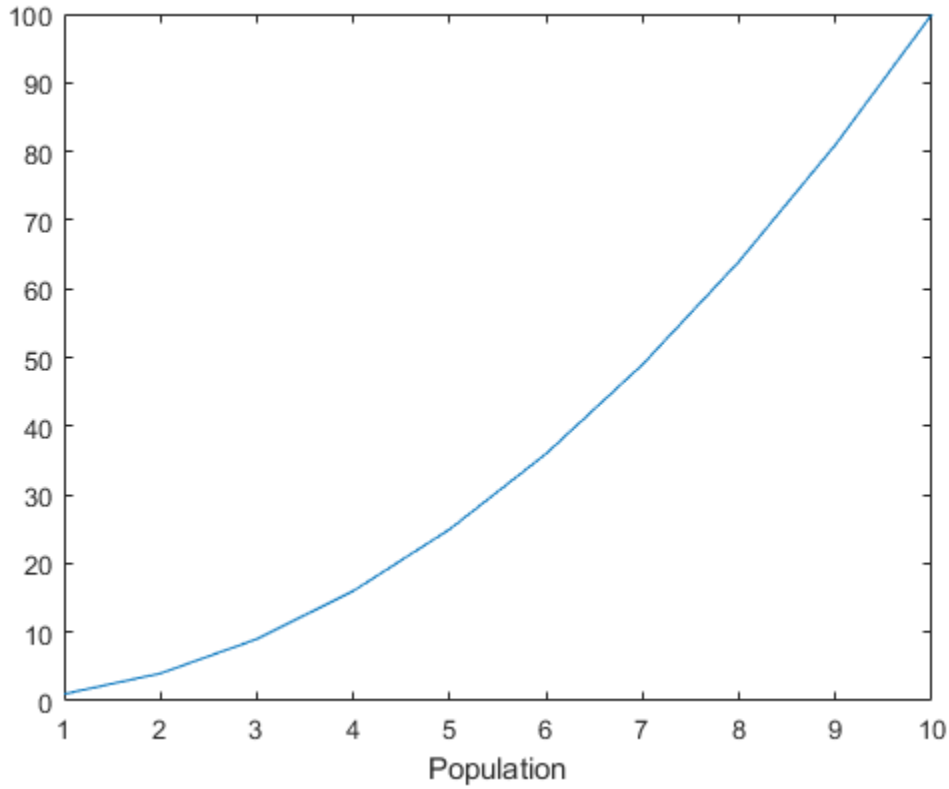
`t = xlabel( ___ )` returns the text object used as the *x*-axis label. Use `t` to make future modifications to the label after it is created.

Examples

Label x-Axis

Display `Population` beneath the *x*-axis.

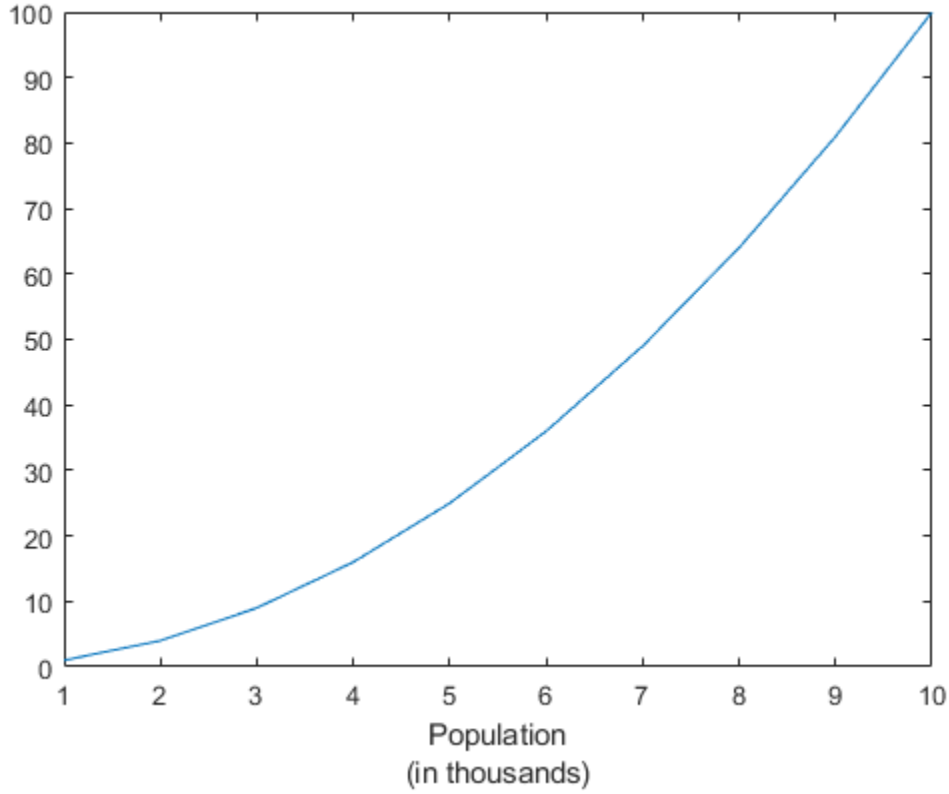
```
plot((1:10).^2)
xlabel('Population')
```



Create Multiline x-Axis Label

Create a multiline label using a cell array of character vectors.

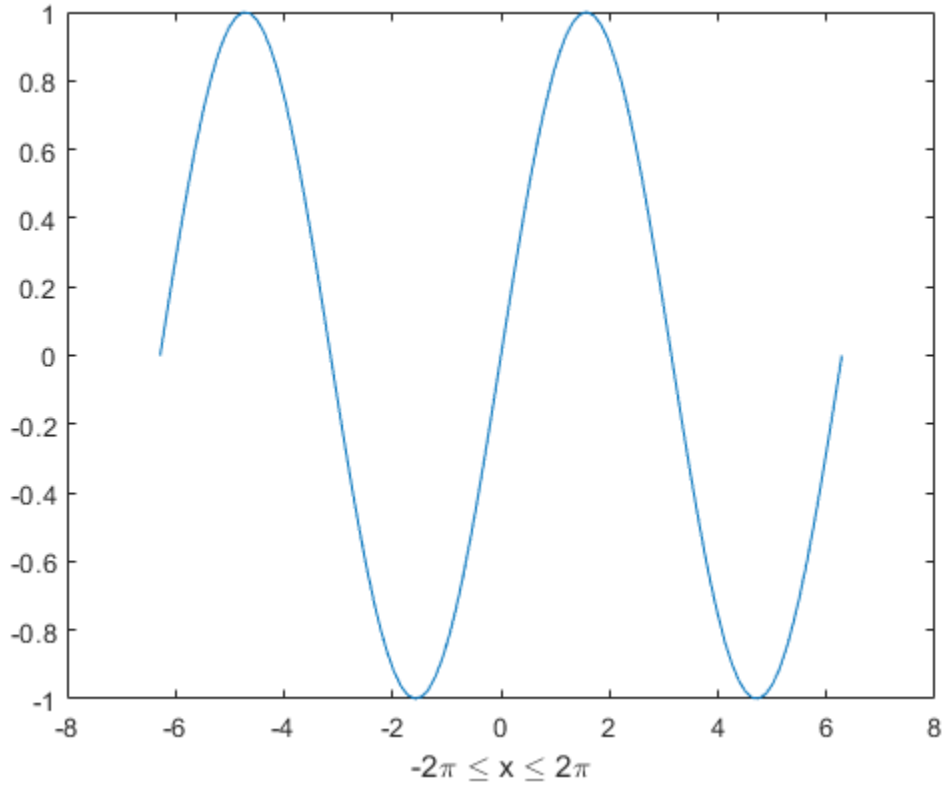
```
plot((1:10).^2)
xlabel({'Population','(in thousands)'})
```



Include Greek Letters in x-Axis Label

Include Greek letters and other special characters in the label using TeX markup.

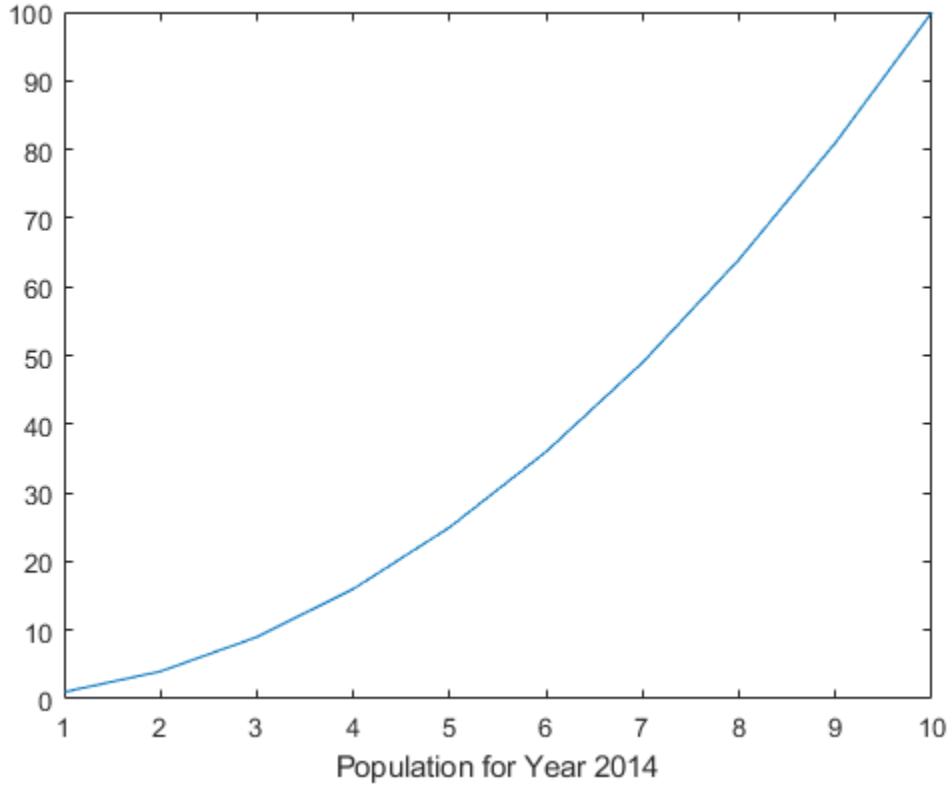
```
x = linspace(-2*pi,2*pi);
y = sin(x);
plot(x,y)
xlabel('-2\pi \leq x \leq 2\pi')
```



Include Variable Value in x-Axis Label

Add a label with text and a variable value. Use the `num2str` function to include a variable value in the label.

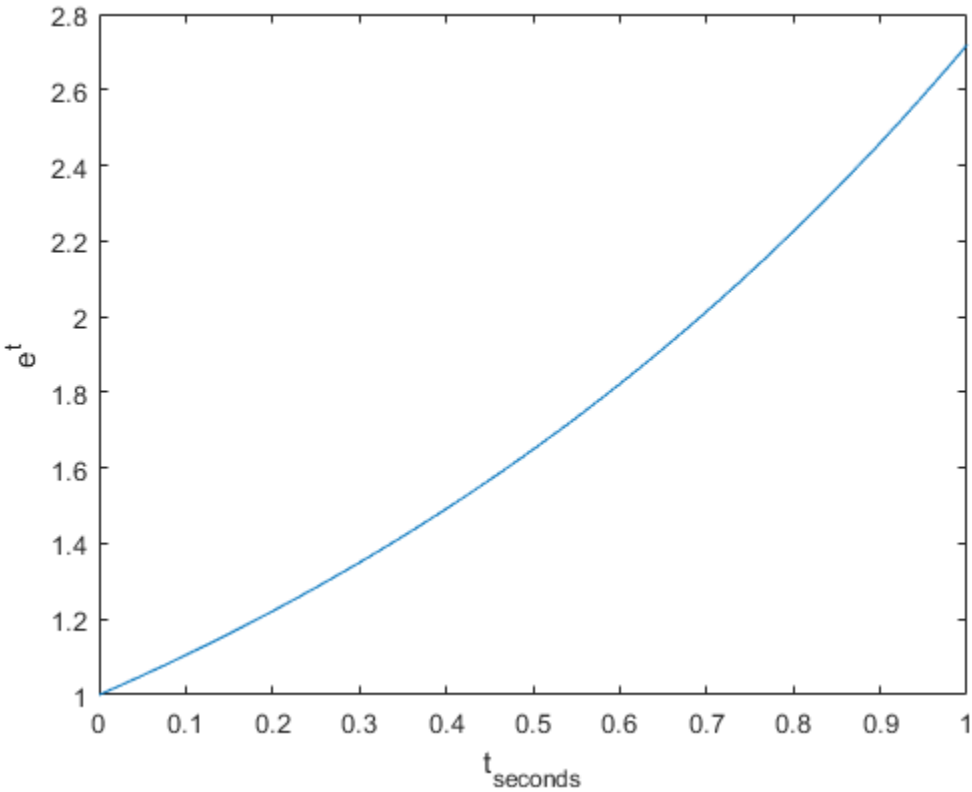
```
plot((1:10).^2)
year = 2014;
xlabel(['Population for Year ',num2str(year)])
```



Include Superscript and Subscript in Axis Labels

Use the `'^'` and `'_'` characters to include superscripts and subscripts in the axis labels. Use curly braces `{}` to modify more than one character.

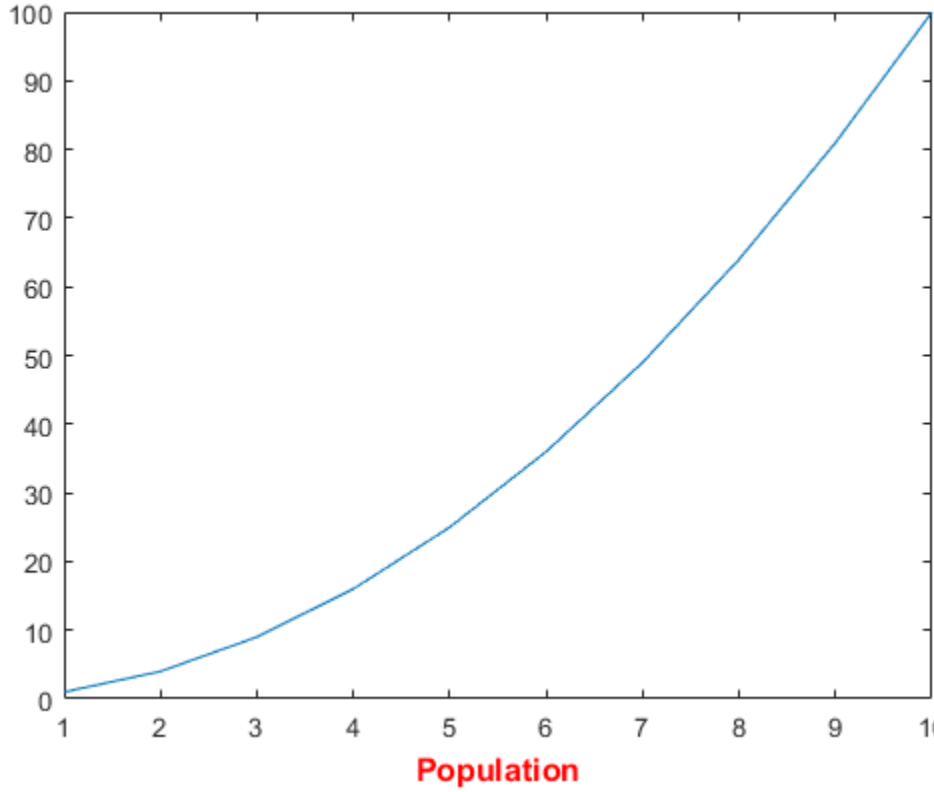
```
t = linspace(0,1);
y = exp(t);
plot(t,y)
xlabel('t_{seconds}')
ylabel('e^t')
```



Change x-Axis Label Font Size and Color

Use `Name,Value` pair arguments to set the font size, font weight, and text color properties of the *x*-axis label.

```
plot((1:10).^2)
xlabel('Population','FontSize',12,'FontWeight','bold','Color','r')
```

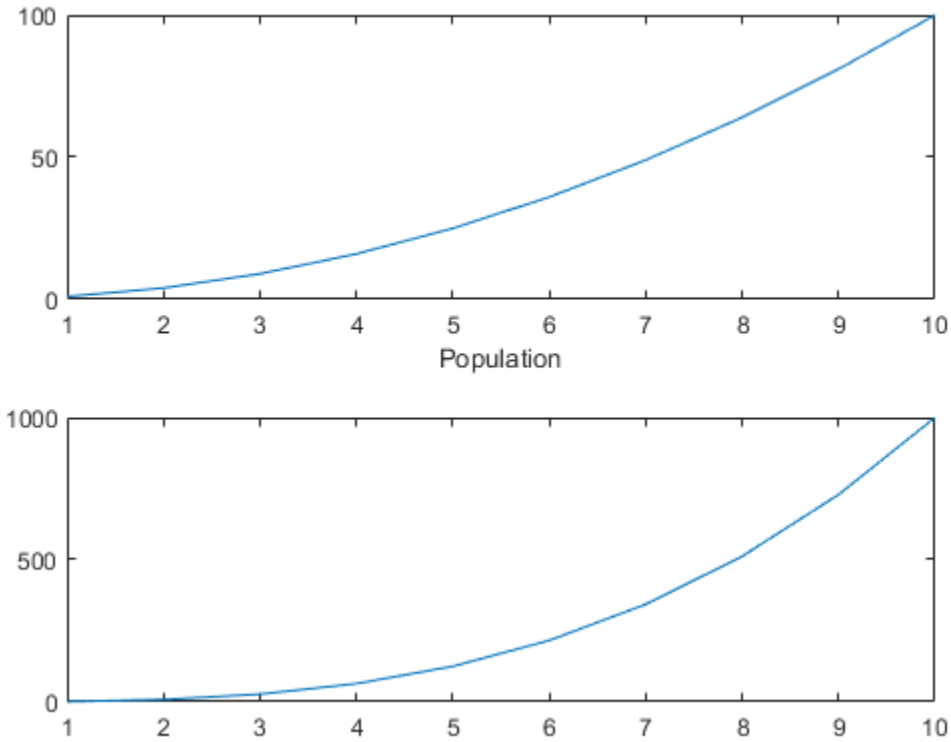


Label x-Axis of Specific Plot

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Plot data into each axes, and create an *x*-axis label for the top plot.

```
tiledlayout(2,1)
ax1 = nexttile;
plot((1:10).^2)
xlabel(ax1,'Population')
```

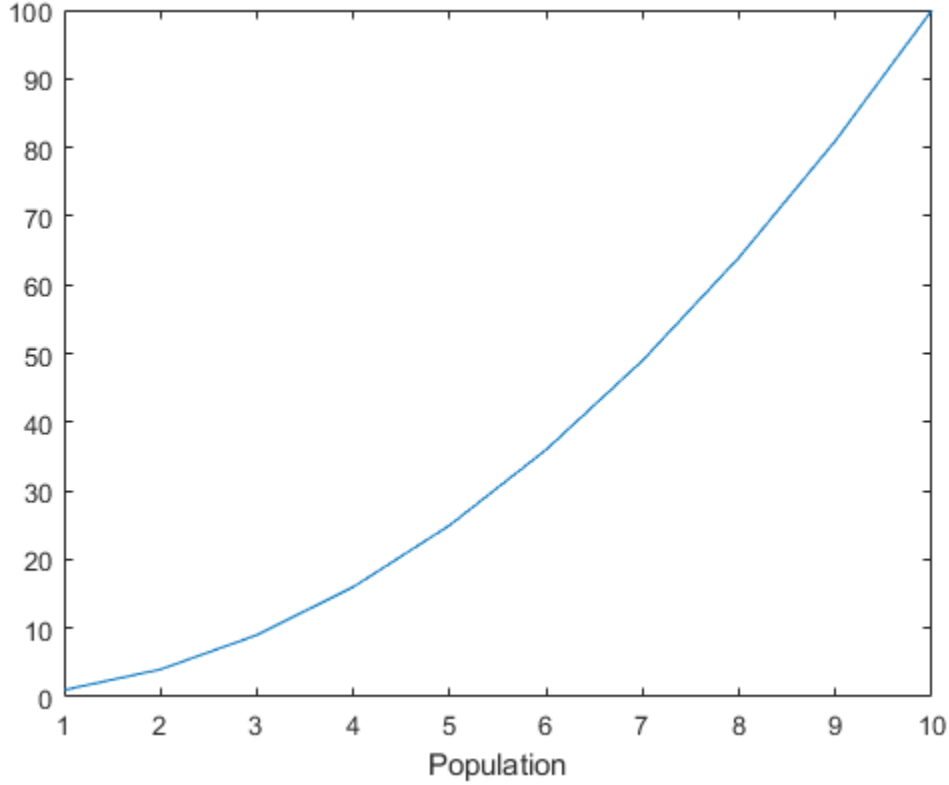
```
ax2 = nexttile;
plot((1:10).^3)
```



Modify x-Axis Label After Creation

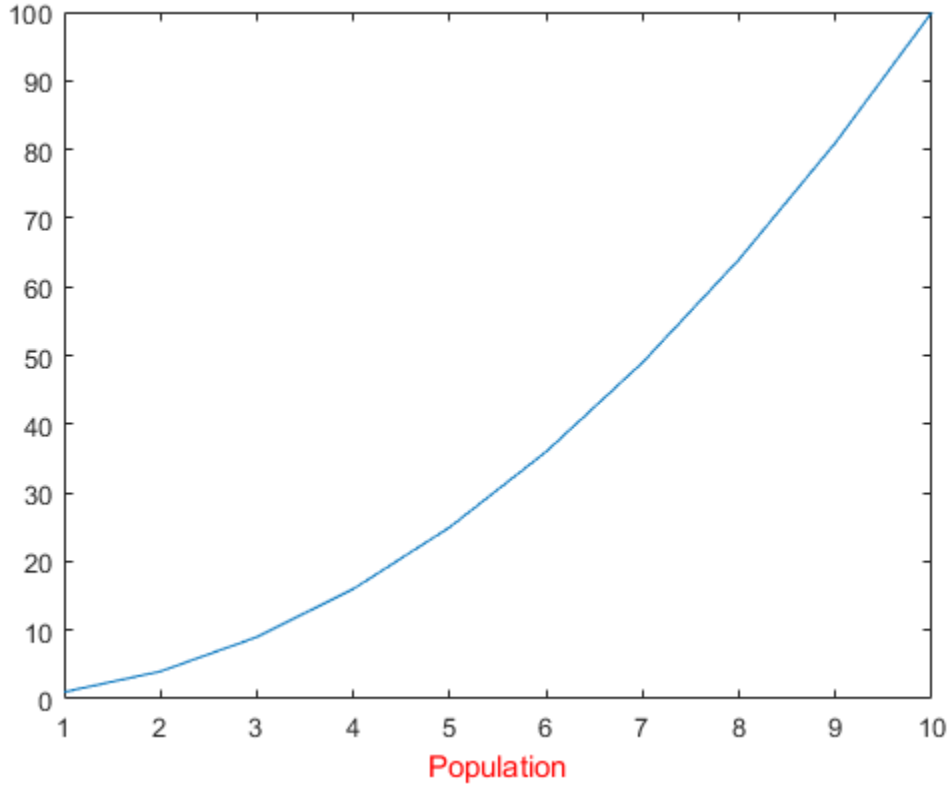
Label the *x*-axis and return the text object used as the label.

```
plot((1:10).^2)
t = xlabel('Population');
```



Use `t` to set text properties of the label after it has been created. For example, set the color of the label to red. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead.

```
t.Color = 'red';
```



Input Arguments

`txt` — Axis label

string scalar | character vector | string array | character array | cell array | categorical array | numeric value

Axis label, specified as a string scalar, character vector, string array, character array, cell array, categorical array, or numeric value.

Example: `'my label'`

Example: `{'first line','second line'}`

Example: 123

To include numeric variables with text in a label, use the `num2str` function. For example:

```
x = 42;
txt = ['The value is ',num2str(x)];
```

To include special characters, such as superscripts, subscripts, Greek letters, or mathematical symbols use TeX markup. For a list of supported markup, see the `Interpreter` property.

To create multiline labels:

- Use a string array, where each element contains a line of text, such as `["first line","second line"]`.
- Use a cell array, where each cell contains a line of text, such as `{'first line','second line'}`.
- Use a character array, where each row contains the same number of characters, such as `['abc'; 'ab ']`.
- Use `sprintf` to create text with a new line character, such as `sprintf('first line \n second line')`.

Numeric labels are converted to text using `sprintf('%g',value)`. For example, `12345678` displays as `1.23457e+07`.

Note

- If you specify the label as a categorical array, MATLAB uses the values in the array, not the categories.
- The words `default`, `factory`, and `remove` are reserved words that will not appear in a label when quoted as a normal characters. To display any of these words individually, precede them with a backslash, such as `'\default'` or `'\remove'`.

`target` — Target for label

`Axes` object | `TiledChartLayout` object | graphics object

Target for label, specified as an `Axes` object, a `TiledChartLayout` object, or a graphics object that has an `XLabel` property. For example, you can add an *x*-axis label to a `HeatmapChart` object.

If you do not specify the target, then the `xlabel` function adds the label to the graphics object returned by the `gca` command.

Note Some charts do not support modifying the label appearance, such as the color, or returning the text object as an output argument.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Color','red','FontSize',12` specifies red, 12-point font.

The text properties listed here are only a subset. For a complete list, see Text.

`FontSize` — Font size
`11` (default) | scalar value greater than `0`

Font size, specified as a scalar value greater than `0` in point units. One point equals 1/72 inch. To change the font units, use the `FontUnits` property.

Setting the font size properties for the associated axes also affects the label font size. The label font size updates to equal the axes font size times the label scale factor. The `FontSize` property of the axes contains the axes font size. The `LabelFontSizeMultiplier` property of the axes contains the label scale factor. By default, the axes font size is 10 points and the scale factor is 1.1, so the *x*-axis label font size is 11 points.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`FontWeight` — Character thickness
`'normal'` (default) | `'bold'`

Character thickness, specified as `'normal'` or `'bold'`.

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.

`FontName` — Font name
supported font name | `'FixedWidth'`

Font name, specified as a supported font name or `'FixedWidth'`. To display and print text properly, you must choose a font that your system supports. The default font depends on your operating system and locale.

To use a fixed-width font that looks good in any locale, use `'FixedWidth'`. The fixed-width font relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`Color` — Text color
`[0.15 0.15 0.15]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Text color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`Interpreter` — Interpretation of text characters
`'tex'` (default) | `'latex'` | `'none'`

Interpretation of text characters, specified as one of these values:

- `'tex'` — Interpret characters using a subset of TeX markup.
- `'latex'` — Interpret characters using LaTeX markup.
- `'none'` — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text.

Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they modify only the next character or the characters within the curly braces. When you set the `Interpreter` property to `'tex'`, the supported modifiers are as follows.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{`*`specifier`*`}`	Font name — Replace *`specifier`* with the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{`*`specifier`*`}`	Font size —Replace *`specifier`* with a numeric scalar value in point units.	`'\fontsize{15} text'`
`\color{`*`specifier`*`}`	Font color — Replace *`specifier`* with one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`

Modifier	Description	Example
`\color[rgb] {specifier}`	Custom font color — Replace *specifier* with a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	β	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ϵ	`\Delta`	Δ	`\spadesuit`	♠
`\zeta`	ζ	`\Theta`	Θ	`\leftrightarrow`	↔
`\eta`	η	`\Lambda`	Λ	`\leftarrow`	←
`\theta`	θ	`\Xi`	Ξ	`\Leftarrow`	⇐
`\vartheta`	ϑ	`\Pi`	Π	`\uparrow`	↑
`\iota`	ι	`\Sigma`	Σ	`\rightarrow`	→
`\kappa`	κ	`\Upsilon`	ϒ	`\Rightarrow`	⇒
`\lambda`	λ	`\Phi`	Φ	`\downarrow`	↓
`\mu`	µ	`\Psi`	Ψ	`\circ`	º
`\nu`	ν	`\Omega`	Ω	`\pm`	±
`\xi`	ξ	`\forall`	∀	`\geq`	≥
`\pi`	π	`\exists`	∃	`\propto`	∝

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. Use dollar symbols around the text, for example, use `'$\int_1^{20} x^2 dx$'` for inline mode or `'$$ \int_1^{20} x^2 dx$$'` for display mode.

The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup.

The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters. For multiline text, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project website at https://www.latex-project.org/.

Output Arguments

`t` — Text object
text object

Text object used as the *x*-axis label. Use `t` to access and modify properties of the label after it has been created.

See Also

Functions
`num2str` | `text` | `title` | `ylabel` | `zlabel`

Properties
Text

Topics
"Add Title and Axis Labels to Chart"

Introduced before R2006a

xline

Vertical line with constant x-value

Syntax

```
xline(xvalue)
xline(xvalue,LineSpec)
xline(xvalue,LineSpec,label)
xline( ___ ,Name,Value)
xline(ax, ___ )
xl = xline( ___ )
```

Description

`xline(xvalue)` creates a constant vertical line at the specified *x*-value in the current axes. For example, `xline(2)` creates a line at `x = 2`.

`xline(xvalue,LineSpec)` specifies either the line style, the line color, or both. For example, `'-.'` creates a dash-dot line, `'b'` creates a blue line, and `'-.b'` creates a blue dash-dot line.

`xline(xvalue,LineSpec,label)` adds the specified label to the line.

`xline( ___ ,Name,Value)` specifies constant line properties using one or more name-value pair arguments. For a list of properties, see ConstantLine Properties. Specify name-value pairs after all other input arguments.

`xline(ax, ___ )` creates the line in the specified axes instead of the current axes. To determine the current axes use `gca`.

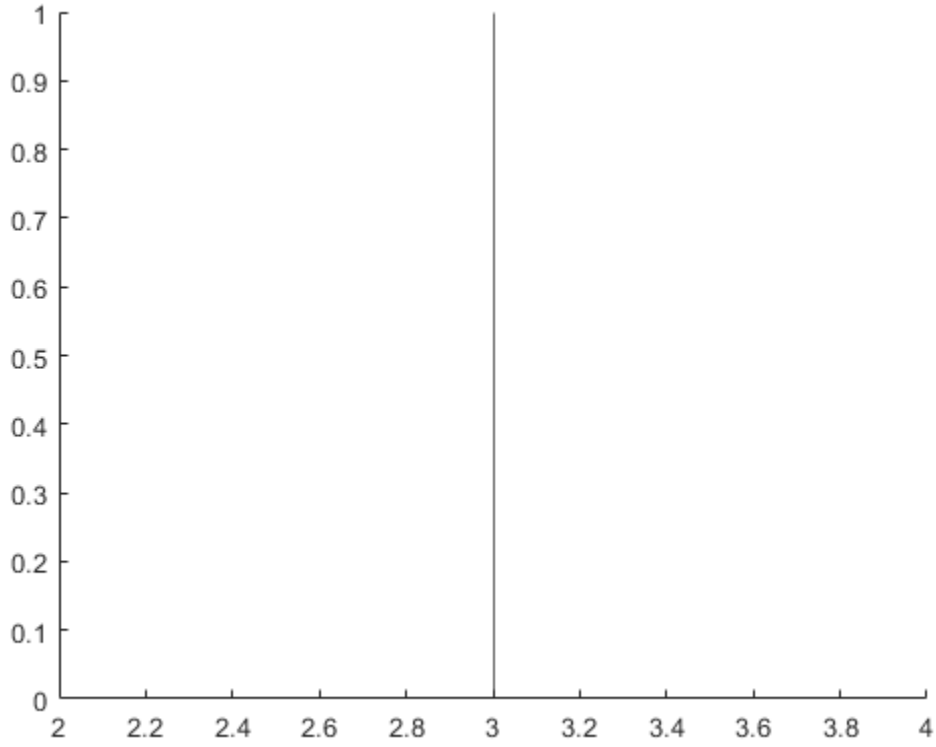
`xl = xline( ___ )` returns the `ConstantLine` object created. Use `xl` to modify or query properties of the line after it is created.

Examples

Create Vertical Line

Create a vertical line at x = 3.

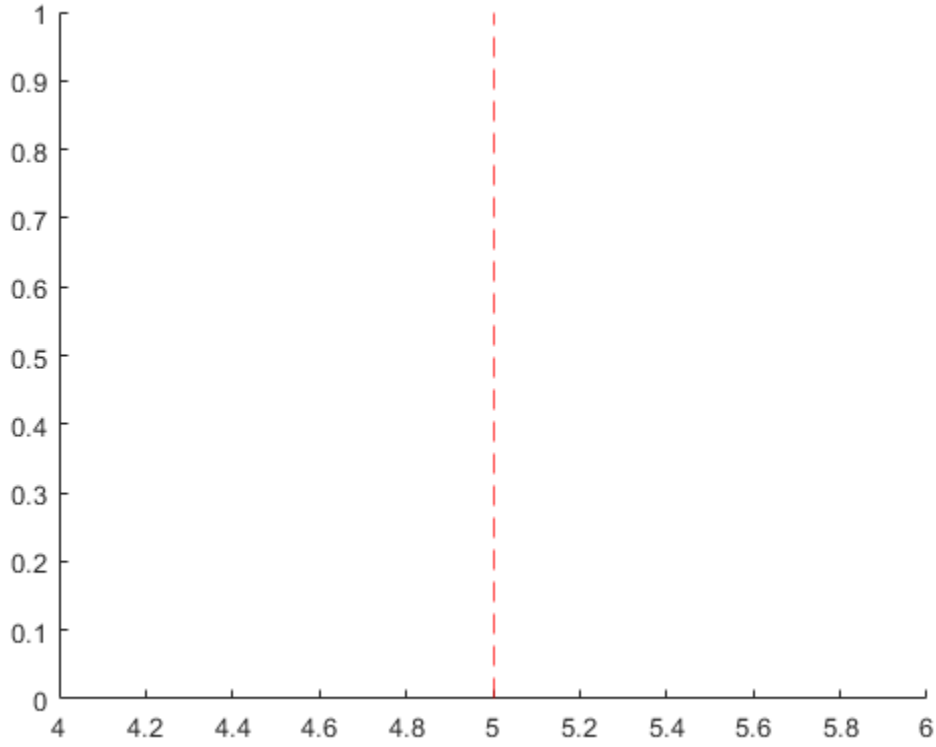
```
xline(3);
```



Specify Vertical Line Style and Color

Create a red dashed line at x = 5.

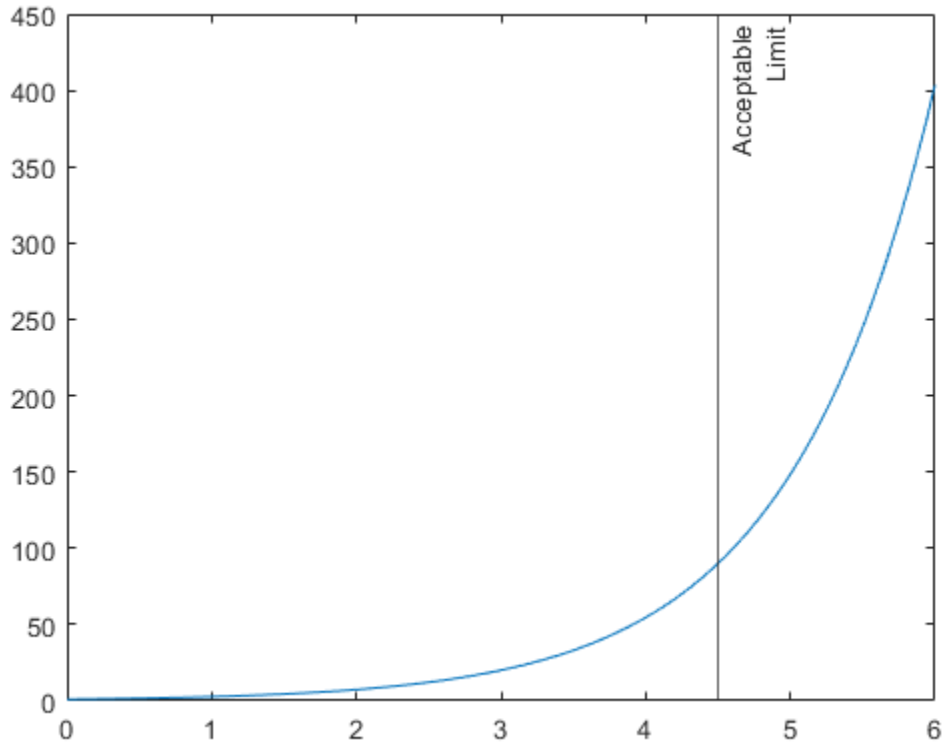
```
xline(5,'--r');
```



Create Vertical Line with Label

To create a line with a label, you must also specify the line style. The default line style is a solid line, `'-'`.

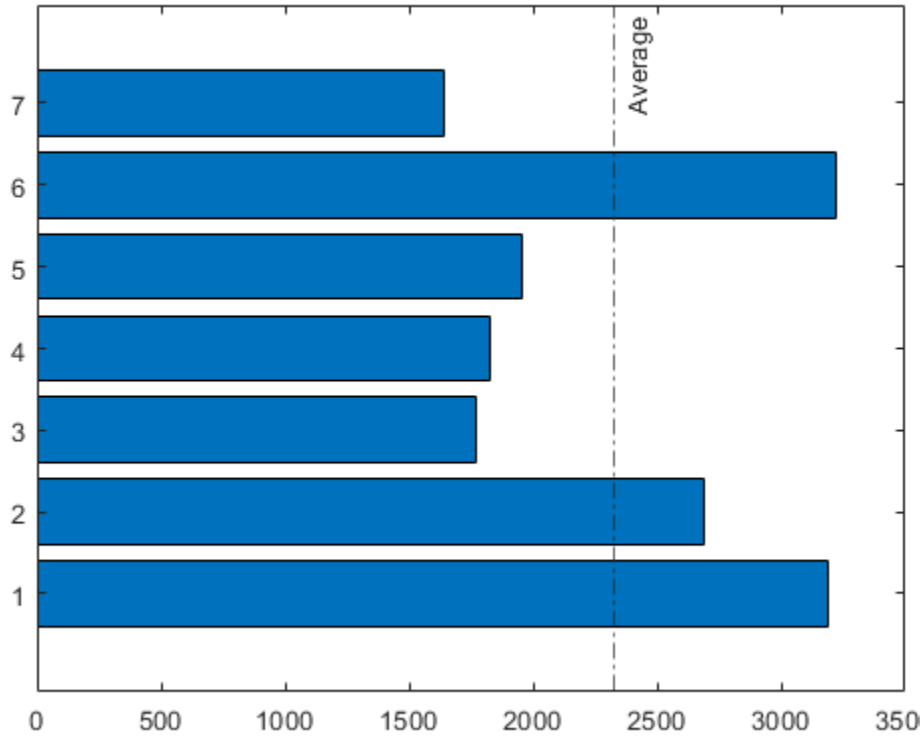
```
x = linspace(0,6,100);
y = exp(x);
plot(x,y)
xline(4.5,'-',{'Acceptable','Limit'});
```



Specify Vertical Line Properties

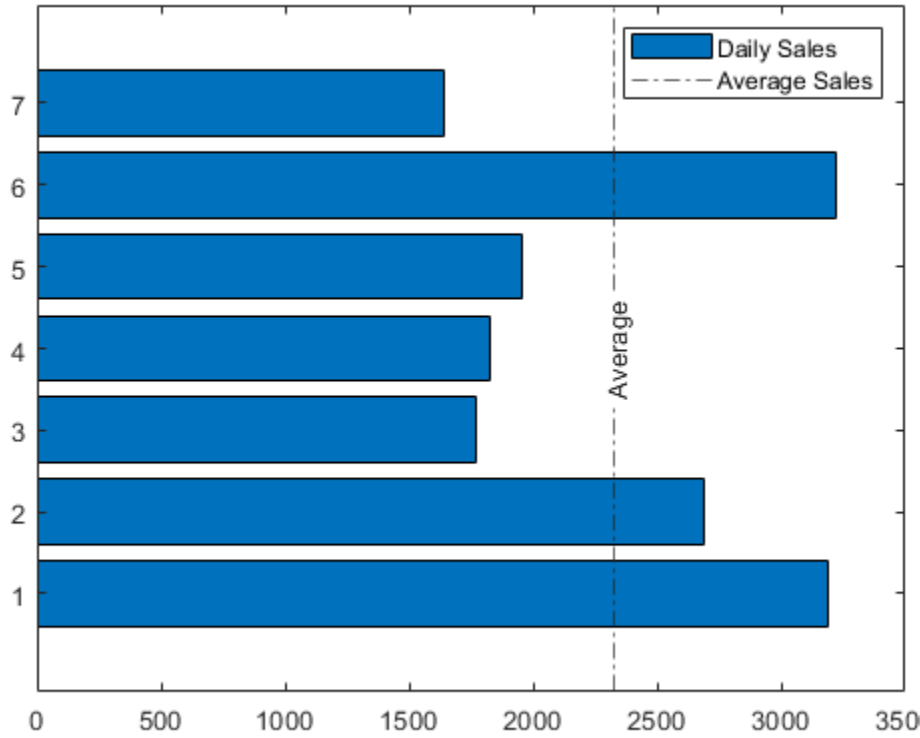
Create a vertical dash-dot line with both a label and a display name for a legend.

```
y = [3187 2693 1771 1826 1958 3222 1645];
barh(y,'DisplayName','Daily Sales')
xl = xline(2328,'-.','Average','DisplayName','Average Sales');
```



Adjust the vertical and horizontal alignment of the line, then display the legend.

```
xl.LabelVerticalAlignment = 'middle';
xl.LabelHorizontalAlignment = 'center';
legend('show');
```

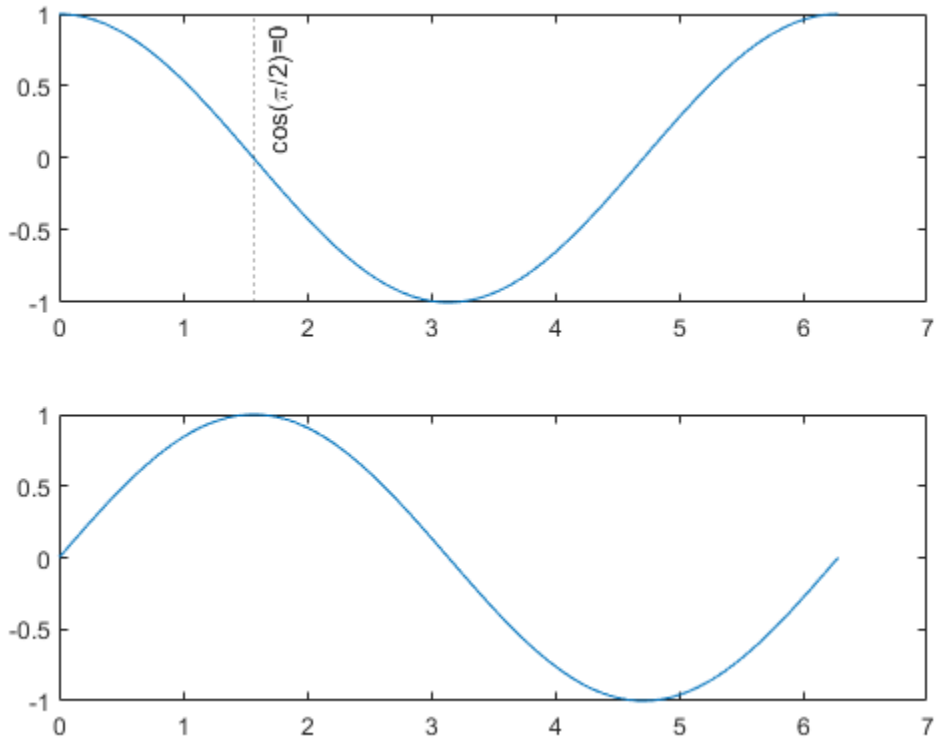


Plot Vertical Line in Specific Axes

Create subplots with cosine and sine waves, then add a vertical line to the cosine plot. Label the line as `'cos(\pi/2)=0'`.

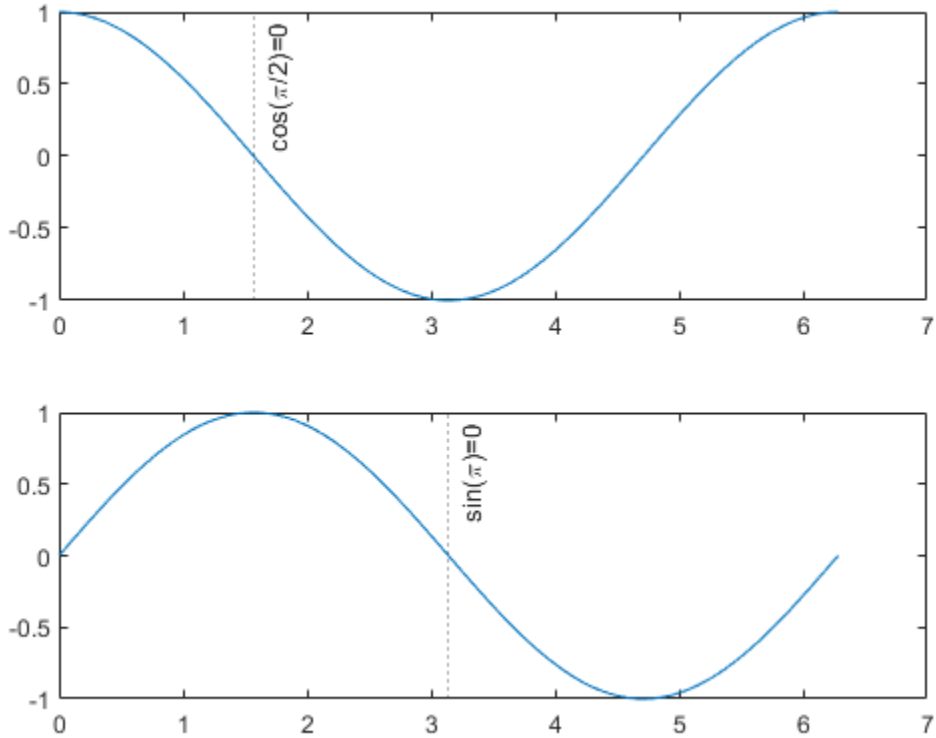
```
ax1 = subplot(2,1,1);
x = linspace(0,2*pi,200);
y1 = cos(x);
plot(ax1,x,y1)

ax2 = subplot(2,1,2);
```

```
x = linspace(0,2*pi,200);
y2 = sin(x);
plot(ax2,x,y2)

xlcosine = xline(ax1,pi/2,':','cos(\pi/2)=0');
```



Add a line to the sine subplot.

```
xlsine = xline(ax2,pi,':','sin(\pi)=0');
```



Input Arguments

`xvalue` — Location of vertical line
scalar

Location of the vertical line on the *x*-axis, specified as a scalar. You can specify `xvalue` as a numeric, categorical, `datetime`, or duration value. However, the type of value you specify must match the data type for the values along the *x*-axis.

Example: `xline(12.4)` plots the line at x = 12.4.

Example: `xline(D(3))` plots the line at the value of the third element in the array.

Example: `xline(categorical({'orange'}))` plots the line at the category value `'orange'`.

`LineSpec` — Line style and color
character vector (default) | string scalar

Line style and color, specified as a character vector or string scalar containing symbols. The characters and symbols can appear in any order. You can specify line style, line color, or both. You cannot specify a marker symbol.

Example: `'--'` is a dashed line.

Example: `'--g'` is a green dashed line.

Line Style	Description
`'-'`	Solid line (default)
`'--'`	Dashed line
`':'`	Dotted line
`'-.'`	Dash-dot line

Color	Description
`'y'`	Yellow
`'m'`	Magenta
`'c'`	Cyan
`'r'`	Red
`'g'`	Green
`'b'`	Blue
`'w'`	White
`'k'`	Black

ax — Target axes
current axes (default) | `Axes` object

Target axes, specified as an `Axes` object. Use this argument if you want `xline` to plot in axes other than the current axes.

`label` — Line label

`''` (default) | character vector | cell array of character vectors | string array | numeric array

Line label, specified as a character vector, cell array of character vectors, string array, or numeric array. To create a multiline label, use a string array or a cell array of character vectors.

Example: `xline(-pi/2,':','Lower Bound')`

Example: `xline(4,'-',{'first','second','third'})`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `xline(5,'-','Max','LabelVerticalAlignment','middle')` specifies the vertical alignment of the label as `'middle'`.

Note The properties listed here are only a subset. For a complete list, see ConstantLine Properties.

`Color` — Line color

`[0.15 0.15 0.15]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'g'`

Example: `[0.6 0.2 0.5]`

Example: `'#D2F9A7'`

`LineWidth` — Line width

`0.5` (default) | positive value

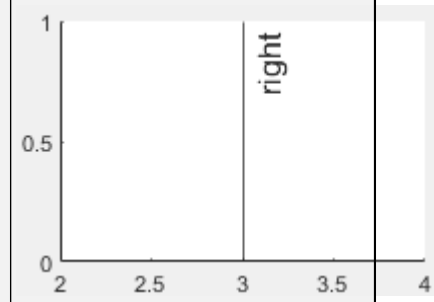
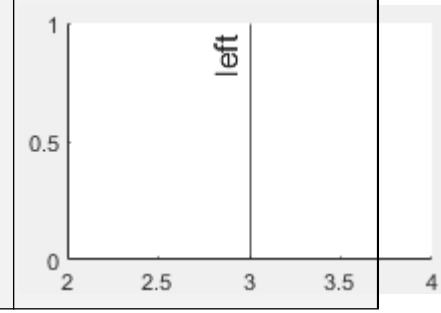
Line width, specified as a positive value in points.

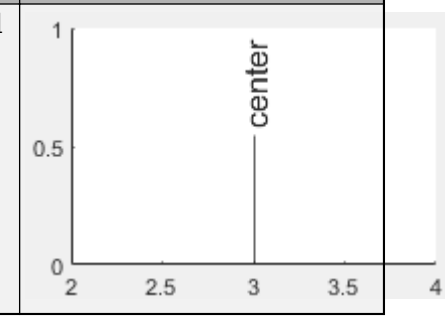
`DisplayName` — Legend label
`''` (default) | character vector | string scalar

Legend label, specified as a character vector or string scalar. The legend does not display until you call the `legend` command. If you do not specify the text, then `legend` sets the label using the form `'dataN'`.

`LabelHorizontalAlignment` — Horizontal alignment of label
`'right'` (default) | `'left'` | `'center'`

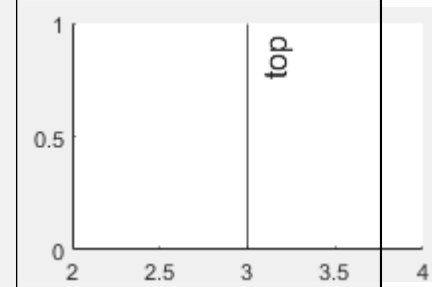
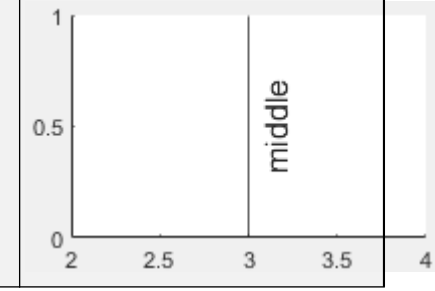
Horizontal alignment of the label with respect to the line, specified as one of the options in the table.

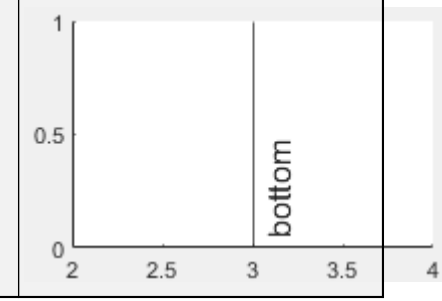
Option	Description	Example
`'right'`	Right side of the line.	
`'left'`	Left side of the line.	

Option	Description	Example
`'center'`	Center of the line. The label segments the line.	

`LabelVerticalAlignment` — Vertical alignment of label
`'top'` (default) | `'middle'` | `'bottom'`

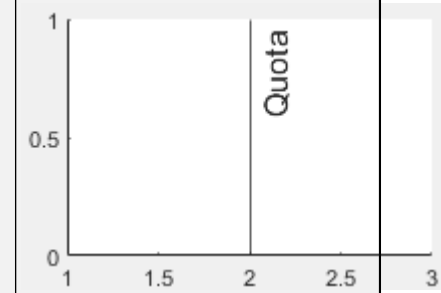
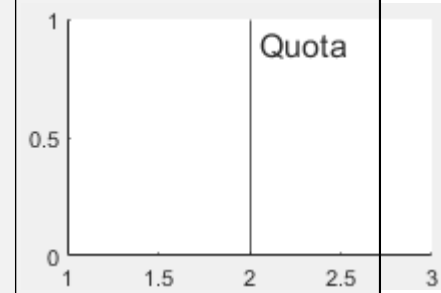
Vertical alignment of the label with respect to the line, specified as one of the options in the table.

Option	Description	Example
`'top'`	Top of the line	
`'middle'`	Middle of the line	

Option	Description	Example
`'bottom'`	Bottom of the line	

`LabelOrientation` — Label orientation
`'aligned'` (default) | `'horizontal'`

Label orientation, specified as `'aligned'` or `'horizontal'`. Examples are shown in the table.

Orientation	Description	Example
`'aligned'`	Label has the same orientation as the line.	
`'horizontal'`	Label is horizontal, regardless of the line orientation.	

Algorithms

In a 3-D view of the axes, the constant line appears in the *x*-*y* plane at the midpoint of the *z*-axis limits. If the axes are rotated, then the constant line rotates accordingly.

See Also

Functions
`plot` | `yline`

Properties
ConstantLine Properties

Introduced in R2018b

xtickangle

Rotate *x*-axis tick labels

Syntax

```
xtickangle(angle)
xtickangle(ax,angle)
```

```
ang = xtickangle
ang = xtickangle(ax)
```

Description

`xtickangle(angle)` rotates the *x*-axis tick labels for the current axes to the specified angle in degrees, where `0` is horizontal. Specify a positive value for counterclockwise rotation or a negative value for clockwise rotation.

`xtickangle(ax,angle)` rotates the tick labels for the axes specified by `ax` instead of the current axes.

`ang = xtickangle` returns the rotation angle for the *x*-axis tick labels of the current axes as a scalar value in degrees. Positive values indicate counterclockwise rotation. Negative values indicate clockwise rotation.

`ang = xtickangle(ax)` uses the axes specified by `ax` instead of the current axes.

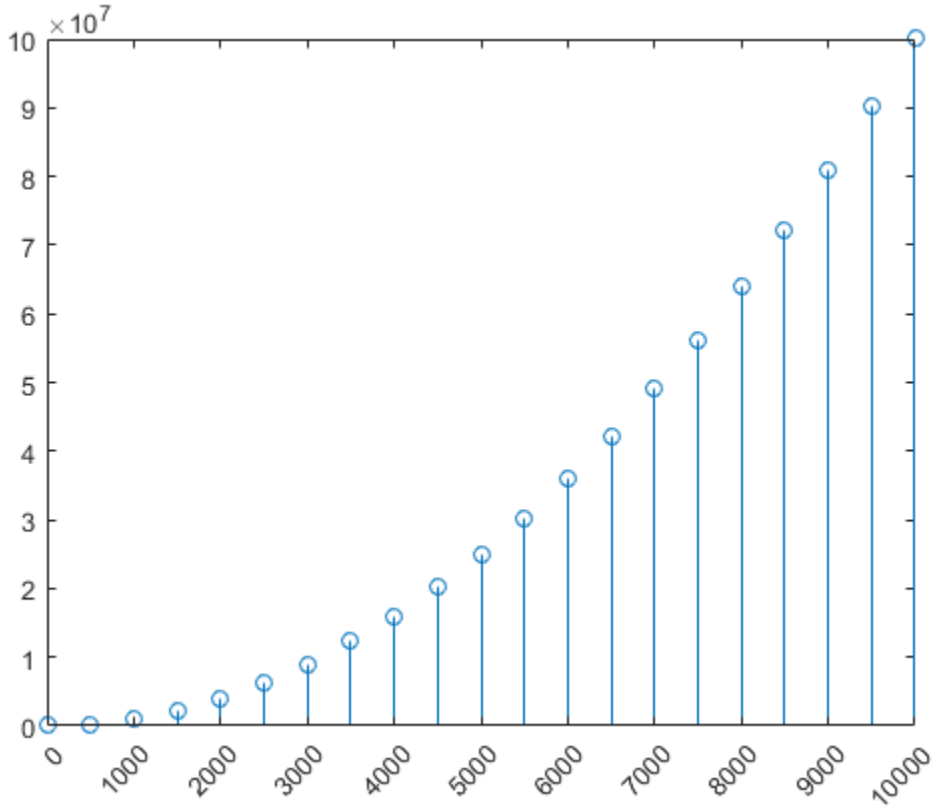
Examples

Rotate *x*-Axis Tick Labels

Create a stem chart and rotate the *x*-axis tick labels so that they appear at a 45-degree angle from the horizontal plane.

```
x = linspace(0,10000,21);
y = x.^2;
```

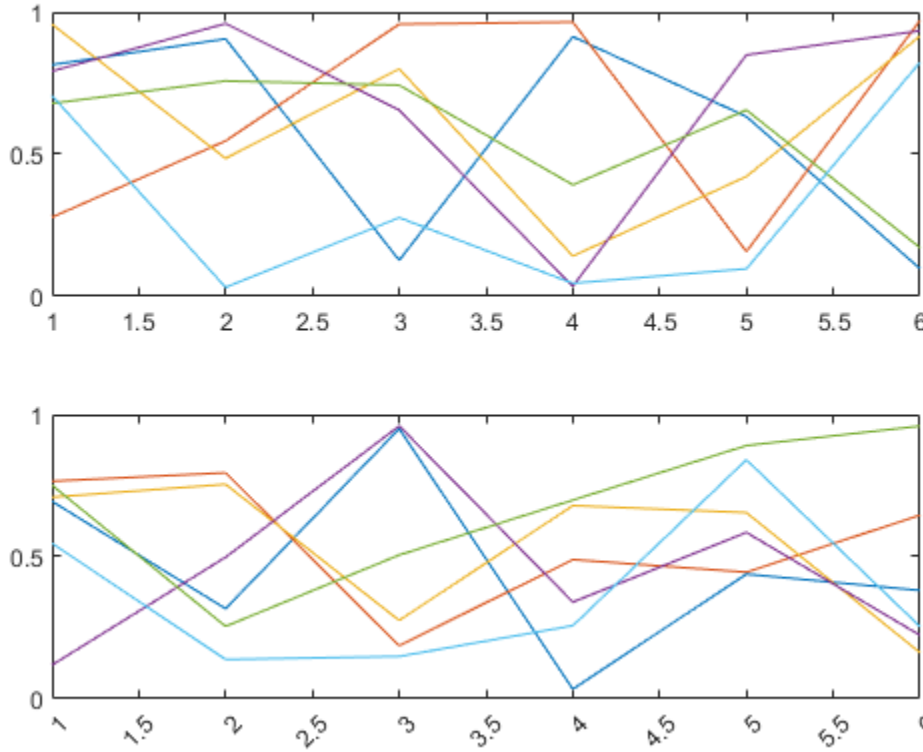
```
stem(x,y)
xtickangle(45)
```



Rotate *x*-Axis Tick Labels for Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Plot into each of the axes. Then rotate the *x*-axis tick labels for the lower plot by specifying `ax2` as the first input argument to `xtickangle`.

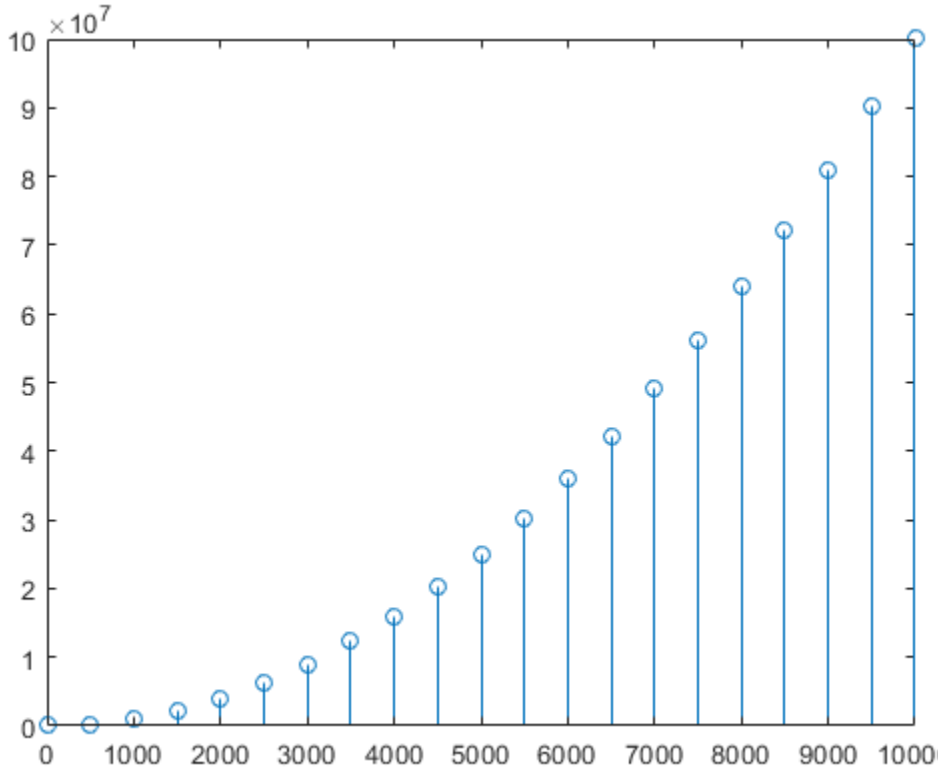
```
tiledlayout(2,1)
ax1 = nexttile;
plot(ax1,rand(6))

ax2 = nexttile;
plot(ax2,rand(6))
xtickangle(ax2,45)
```



Query Current Angle for *x*-Axis Tick Labels

Create a stem chart. Then, query the rotation angle for the *x*-axis tick labels. By default, the labels are not rotated.

```
x = linspace(0,10000,21);
y = x.^2;
stem(x,y)
```




```
ang = xtickangle
```

```
ang = 0
```

Input Arguments

angle — Rotation of tick labels
`0` (default) | scalar value in degrees

Rotation of tick labels, specified as a scalar value in degrees, where 0 is horizontal.

Example: `xtickangle(90)`

ax — Target axes
current axes (default) | single `Axes` object | vector of `Axes` objects

Target axes, specified as a single `Axes` object or a vector of `Axes` objects. If you do not specify the axes, then `xtickangle` uses the current axes.

Algorithms

The `xtickangle` function sets and queries the `XTickLabelRotation` property of the `Axes` object.

See Also

Functions
`xlim` | `xtickformat` | `xticklabels` | `xticks` | `ytickangle` | `ztickangle`

Properties
Axes

Introduced in R2016b

xtickformat

Specify *x*-axis tick label format

Syntax

```
xtickformat(fmt)
xtickformat(datefmt)
xtickformat(durationfmt)
xtickformat(ax, ___ )

xfmt = xtickformat
xfmt = xtickformat(ax)
```

Description

`xtickformat(fmt)` sets the format for numeric *x*-axis tick labels. For example, specify `fmt` as `'usd'` to display the labels in U.S. dollars.

`xtickformat(datefmt)` sets the format for labels that show dates or times. For example, specify `datefmt` as `'MM-dd-yy'` to display dates such as `04-19-16`. This option applies only to an *x*-axis with datetime values.

`xtickformat(durationfmt)` sets the format for labels that show durations. For example, specify `durationfmt` as `'m'` to display the durations in minutes. This option applies only to an *x*-axis with duration values.

`xtickformat(ax, ___ )` uses the axes specified by `ax` instead of the current axes. Specify `ax` as the first input argument.

`xfmt = xtickformat` returns the format style used for *x*-axis tick labels of the current axes. Depending on the type of labels along the *x*-axis, `xfmt` is a character vector of a numeric format, date format, or duration format.

`xfmt = xtickformat(ax)` returns the format style used for the axes specified by `ax` instead of the current axes.

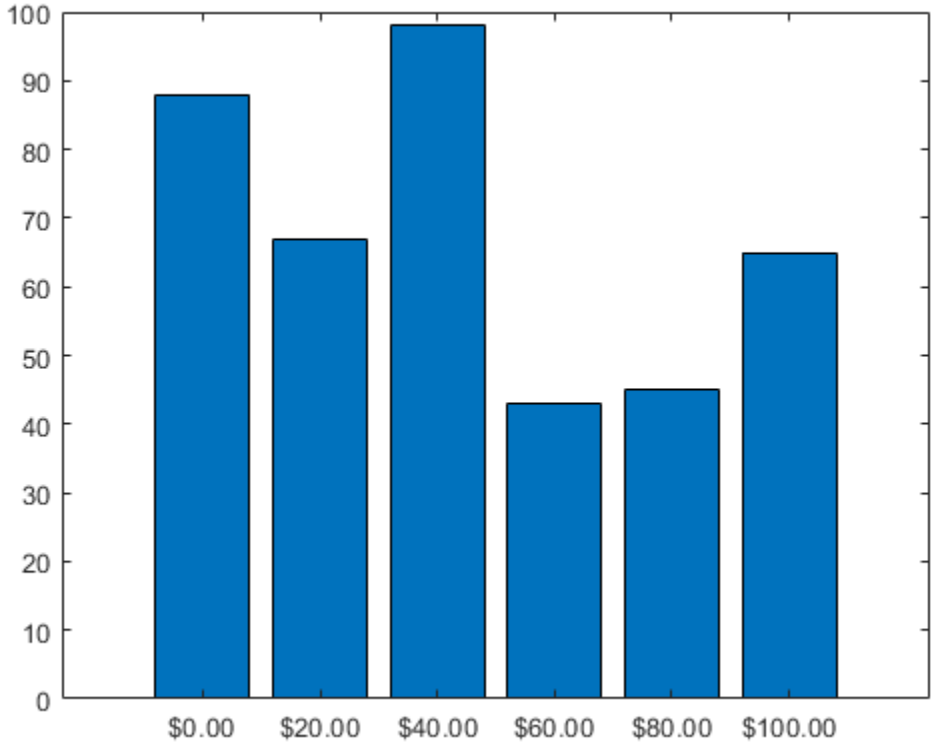
Examples

Display Tick Labels in Dollars

Display *x*-axis tick labels in dollars with and without decimal values.

Create a bar chart. Display the tick labels along the *x*-axis in dollars.

```
x = 0:20:100;
y = [88 67 98 43 45 65];
bar(x,y)
xtickformat('usd')
```



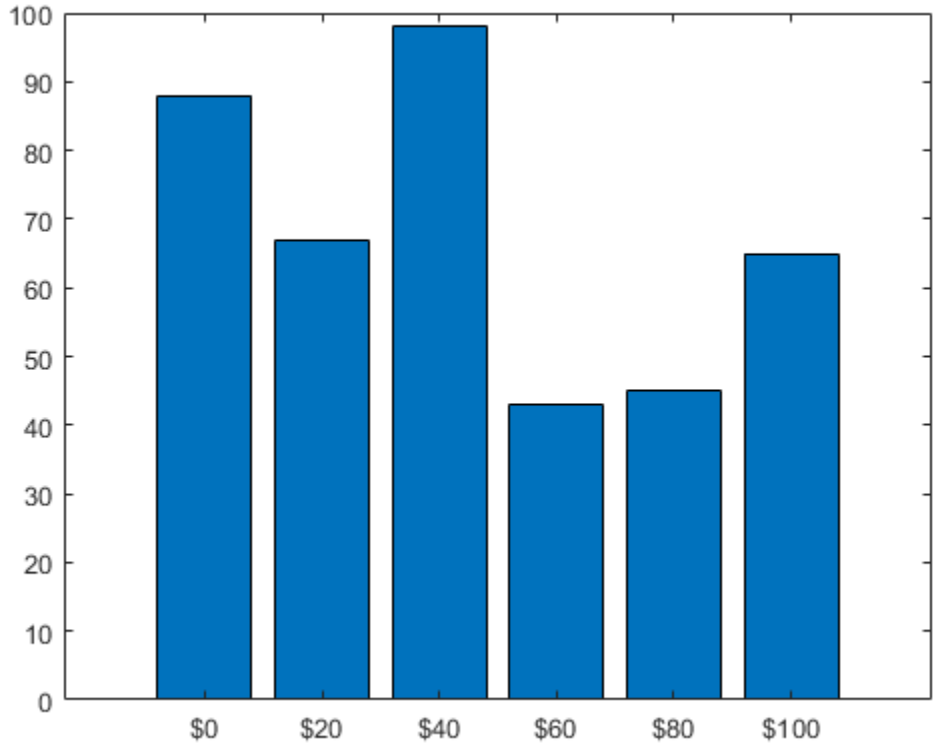
Query the tick label format. MATLAB® returns the format as a character vector containing the formatting operators.

```
fmt = xtickformat
```

```
fmt =
'$%,.2f'
```

Display the tick labels with no decimal values by tweaking the numeric format to use a precision value of 0 instead of 2.

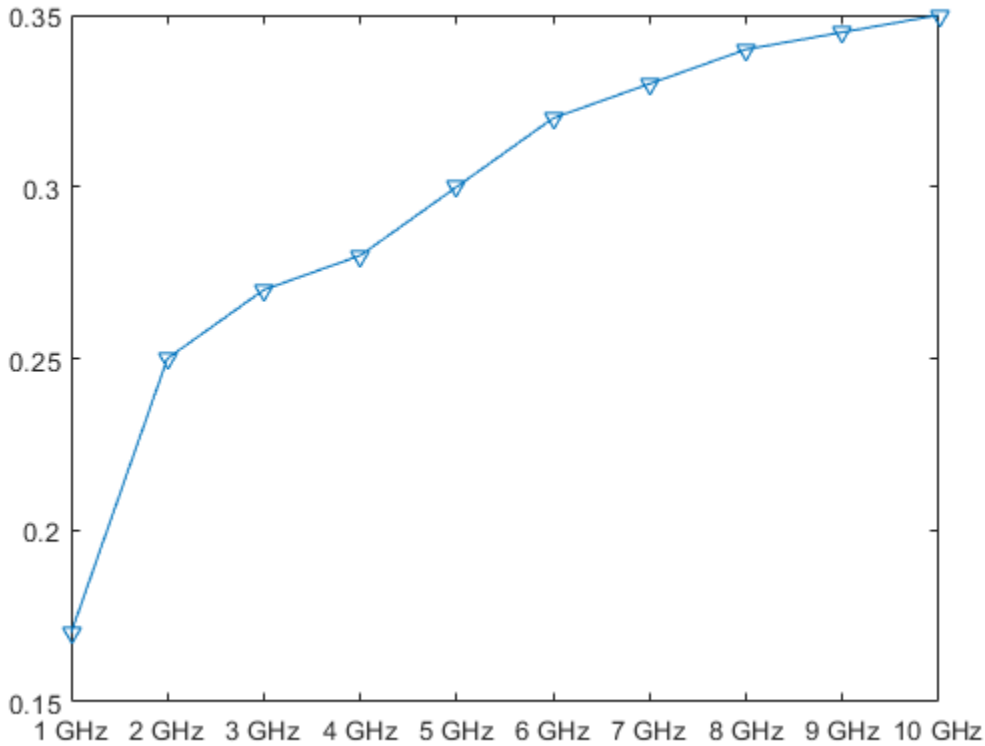
```
xtickformat('$%,.0f')
```



Display Text After Each Tick Label

After creating a line chart, display the tick labels along the *x*-axis with the text "GHz" after each value.

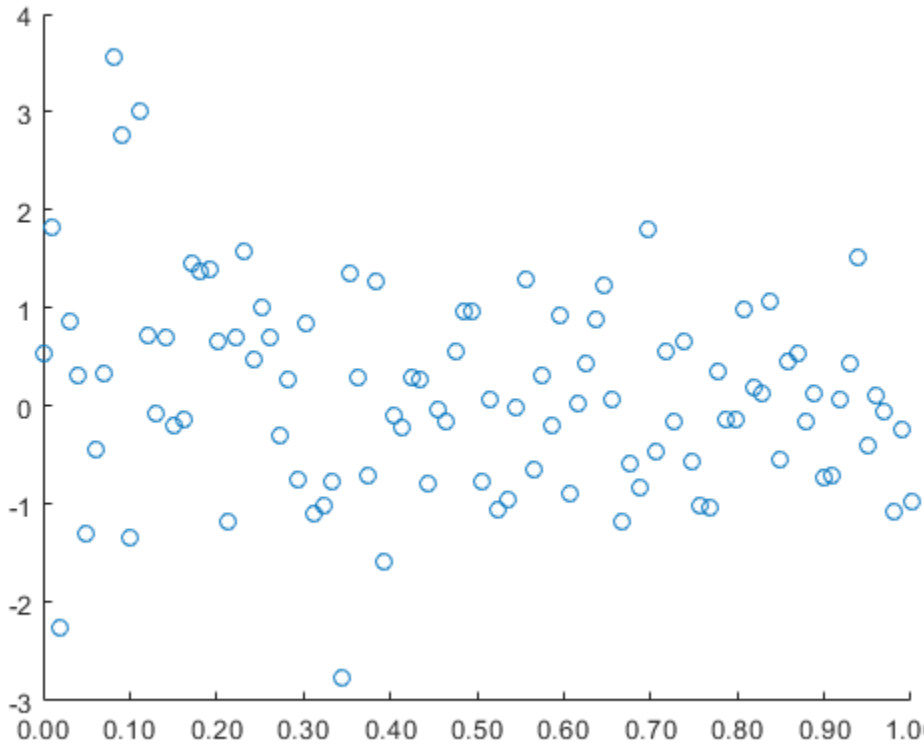
```
x = 1:10;
y = [.17 .25 .27 .28 .3 .32 .33 .34 .345 .35];
plot(x,y,'-V')
xtickformat('%g GHz')
```



Control Number of Decimals Displayed

After creating a scatter plot, display the *x*-axis tick labels with two decimal places. Control the decimal places by passing `xtickformat` a character vector of a numeric format that uses fixed-point notation for the conversion character and a precision value of 2.

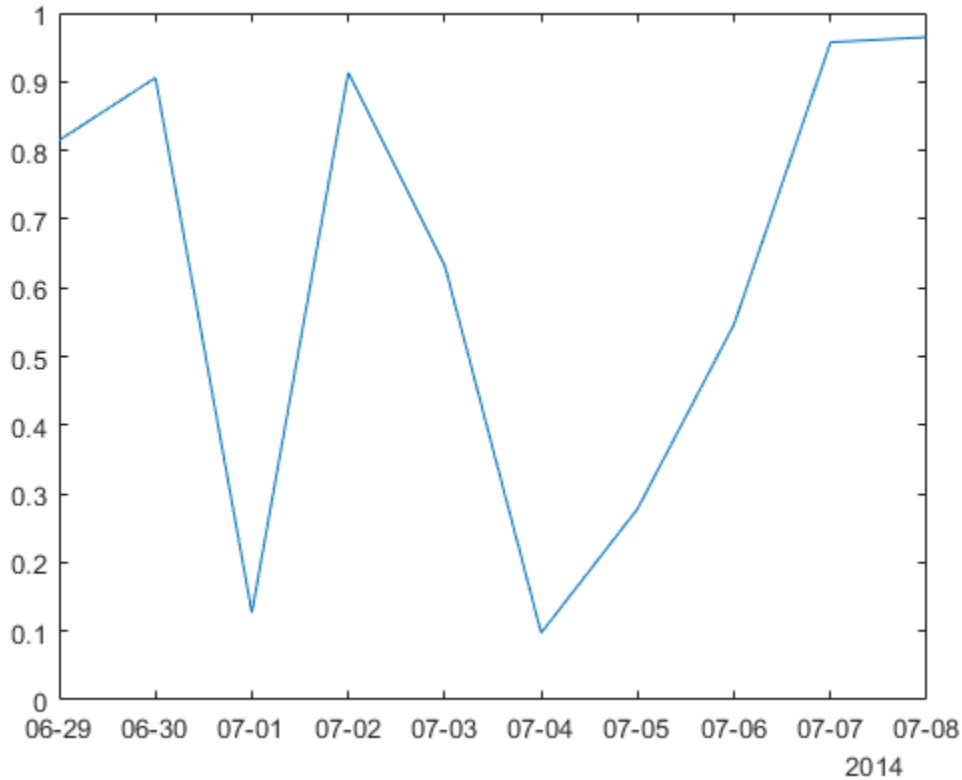
```
x = linspace(0,1,100);
y = randn(1,100).*cos(x);
scatter(x,y)
xtickformat('%.2f')
```



Format for Date Tick Labels

Create a line plot with datetime values along the *x*-axis. Then, change the format of the dates.

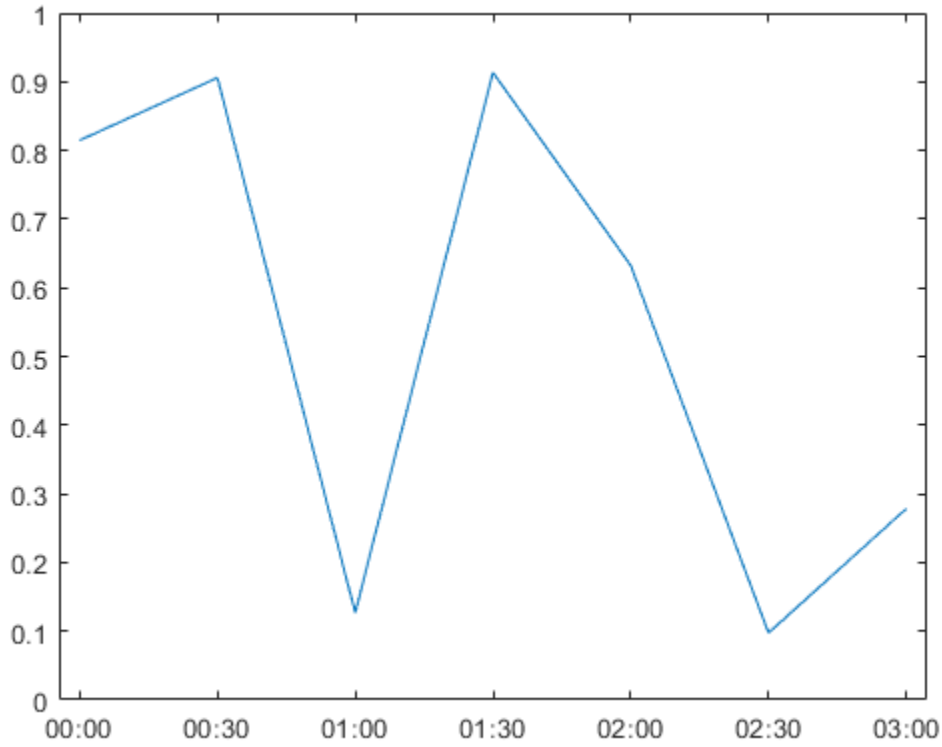
```
t = datetime(2014,6,28) + caldays(1:10);
y = rand(1,10);
plot(t,y);
xtickformat('MM-dd')
```



Format for Duration Tick Labels

Create a line plot with duration values along the *x*-axis. Then, change the format of the tick labels.

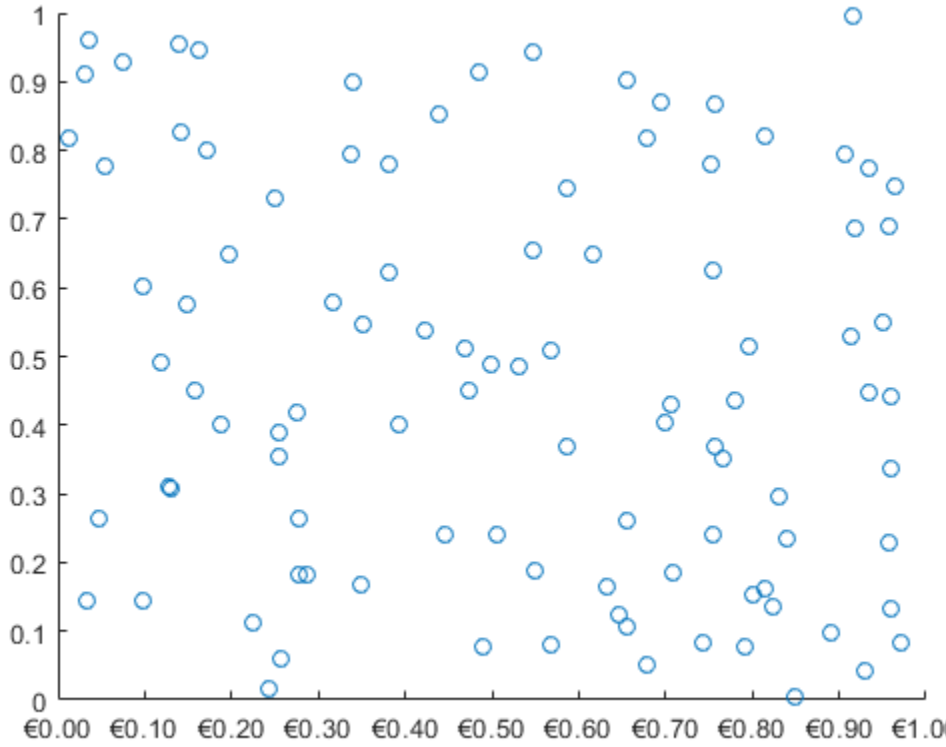
```
t = 0:seconds(30):minutes(3);
y = rand(1,7);
plot(t,y)
xtickformat('mm:ss')
```



Query Current Tick Label Format

Create a scatter plot and display the *x*-axis tick labels in Euro. Then, query the tick label format. MATLAB® returns the format as a character vector containing the formatting operators.

```
x = rand(100,1);
y = rand(100,1);
scatter(x,y)
xtickformat('eur')
```



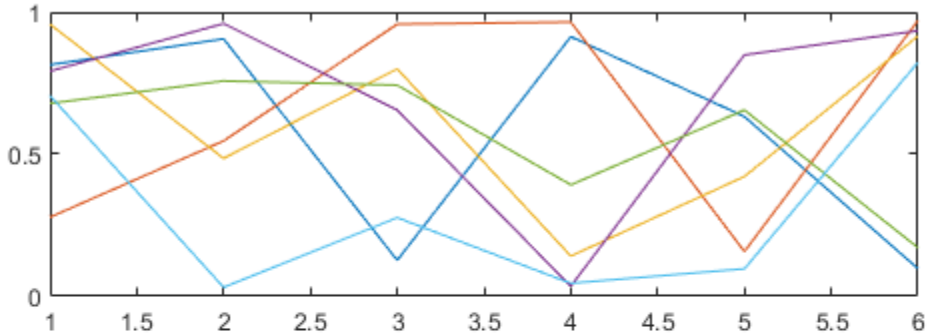

```
xfmt = xtickformat
```

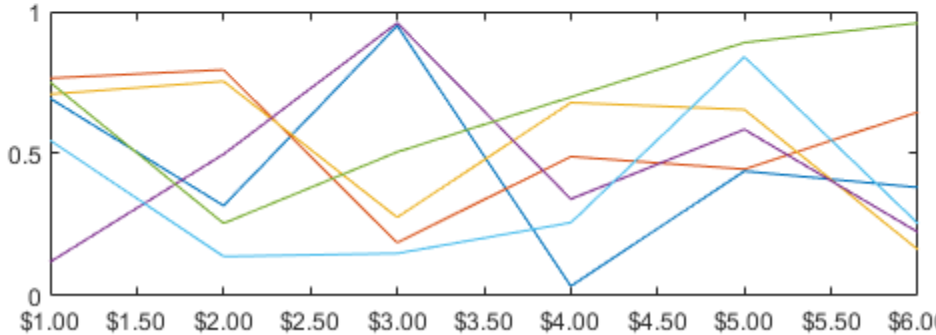
```
xfmt =
'\x20AC%,.2f'
```

Tick Label Format for Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Plot into each of the axes. Specify the tick label format for the *x*-axis of the lower plot by specifying `ax2` as the first input argument to `xtickformat`. Display the tick labels in U.S. dollars.

```
tiledlayout(2,1)
ax1 = nexttile;
plot(ax1,rand(6))

ax2 = nexttile;
plot(ax2,rand(6))
xtickformat(ax2,'usd')
```





Input Arguments

`fmt` — Format for numeric tick labels

`'%g'` (default) | character vector | string scalar

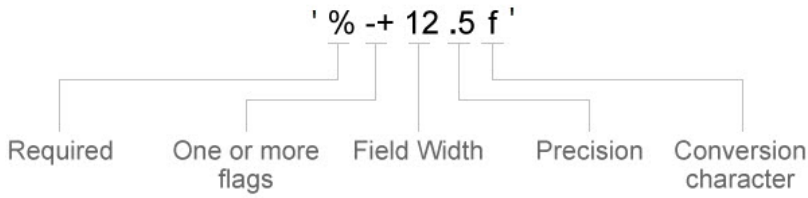
Format for numeric tick labels, specified as a character vector or string scalar. You can specify one of the formats listed in this table. Alternatively, you can specify a custom format.

Predefined Format	Description
`'usd'`	U.S. dollars. This option is equivalent using `'$%,.2f'`. If the labels use scientific notation, this option sets the exponent value to 0.
`'eur'`	Euro. This option is equivalent to using `'\x20AC%,.2f'` with an exponent value of 0.
`'gbp'`	British pound. This option is equivalent to using `'\x00A3%,.2f'` with an exponent value of 0.
`'jpy'`	Japanese yen. This option is equivalent to using `'\x00A5%,d'` with an exponent value of 0.
`'degrees'`	Display degree symbol after values. This option is equivalent to using `'%g\x00B0'` with the default exponent value.
`'percentage'`	Display percent sign after values. This option is equivalent to using `'%g%%'` with the default exponent value.
`'auto'`	Default format of `'%g'` with the default exponent value.

Example: `xtickformat('usd')`

Custom Numeric Format

You can specify a custom numeric format by creating a character vector or string containing identifiers.



Identifiers are optional, except the percent sign and conversion character. Construct the format in this order:

- One or more flags — Options such as adding a plus sign before positive values. For a full list of options, see the table of Optional Flags.
- Field width — Minimum number of characters to print in the tick label. Specify the field width as an integer value. If the number of significant digits in the tick value is smaller than the field width, then the label is padded with spaces.
- Precision — Number of digits to the right of the decimal point or the number of significant digits, depending on the conversion character. Specify the precision as an integer value.
- Conversion character — Value type. For a full list of options, see the table of Conversion Characters. If you specify a conversion that does not fit the data, then MATLAB overrides the specified conversion, and uses `%e`.

Also, you can specify literal text at the beginning or end of the format. To print a single quotation mark, use `''`. To print a percent character, use `%%`.

Example: `xtickformat('%.2f')` displays the values using fixed-point notation with two decimal places.

Example: `xtickformat('$%.2f')` displays a dollar sign before each value.

Example: `xtickformat('%.2f million')` displays `million` after each value.

Optional Flags

Identifier	Description	Example of Numeric Format
`,`	Display commas every three digits, such as `'1,000'`.	`'%,4.4g'`
`+`	Print the sign character (`+`) for positive values, such as `'+100'`.	`'%+4.4g'`
`0`	Pad the field width with leading zeros instead of spaces, such as `'0100'`.	`'%04.4g'`
`-`	Left-justify, which pads the end of the value with spaces instead of the beginning. For example, if the field width is `4`, then this flag formats the label as `'100 '` instead of `' 100'`.	`'%-4.4g'`
`#`	For the `%f`, `%e`, and `%g` conversion characters, print the decimal point even when the precision is 0, such as `'100.'`. For `%g`, do not remove trailing zeros.	`'%#4.4g'`

Conversion Characters

Identifier	Description	Example
`d` or `i`	Signed integer with base 10. The precision value indicates the number of significant digits.	`'%.4d'` displays π as `0003`.
`f`	Fixed-point notation. The precision value indicates the number of decimal places.	`'%.4f'` displays π as `3.1416`.
`e`	Exponential notation. The precision value indicates the number of decimal places.	`'%.4e'` displays π as `3.1416x10`^{0}.
`g`	The more compact version of `e` or `f`, with no trailing zeros. The precision value indicates the maximum number of decimal places.	`'%.4g'` displays π as `3.1416`.

`datefmt` — Format for dates and times
`'auto'` | character vector | string scalar

Format for dates and times, specified as `'auto'`, a character vector, or a string scalar. The default format is based on the data.

Example: `xtickformat('yyyy-MM-dd')` displays a date and time such as `2016-04-19`.

Example: `xtickformat('eeee, MMMM d, yyyy HH:mm:ss')` displays a date and time such as `Saturday, April 19, 2016 21:41:06`.

Example: `xtickformat('MMMM d, yyyy HH:mm:ss Z')` displays a date and time such as `April 19, 2016 21:41:06 -0400`.

The display format tables show the letter identifiers that you can use to construct the format. To separate the fields, you can include nonletter characters such as a hyphen, space, colon, or any non-ASCII characters. The identifiers correspond to the Unicode Locale Data Markup Language (LDML) standard for dates.

Date and Time Formats

Use these identifiers to specify the display formats of the date and time fields.

Letter Identifier	Description	Display
G	Era	CE
y	Year, with no leading zeros. See the Note that follows this table.	2014
yy	Year, using last two digits. See the Note that follows this table.	14
yyy, yyyy ...	Year, using at least the number of digits specified by the number of instances of 'y'	For the year 2014, 'yyy' displays 2014, while 'yyyyy' displays 02014.
u, uu, ...	ISO year. A single number designating the year. An ISO year value assigns positive values to CE years and negative values to BCE years, with 1 BCE being year 0.	2014
Q	Quarter, using one digit	2
QQ	Quarter, using two digits	02
QQQ	Quarter, abbreviated	Q2
QQQQ	Quarter, full name	2nd quarter
M	Month, numerical using one or two digits	4
MM	Month, numerical using two digits	04
MMM	Month, abbreviated name	Apr
MMMM	Month, full name	April
MMMMM	Month, capitalized first letter	A
W	Week of the month	1
d	Day of the month, using one or two digits	5
dd	Day of the month using two digits	05
D	Day of the year, using one, two or three digits	95
DD	Day of the year using two digits	95

Letter Identifier	Description	Display
DDD	Day of the year using three digits	095
e	Day of the week, numerical using one or two digits.	7, where Sunday is the first day of the week.
ee	Day of the week, numerical using two digits	07
eee	Day, abbreviated name	Sat
eeee	Day, full name	Saturday
eeeee	Day, capitalized first letter	S
a	Day period (AM or PM)	PM
h	Hour, 12-hour clock notation using one or two digits	9
hh	Hour, 12-hour clock notation using two digits	09
H	Hour, 24-hour clock notation using one or two digits	21
HH	Hour, 24-hour clock notation using two digits	21
m	Minute, using one or two digits	41
mm	Minute, using two digits	41
s	Second, using one or two digits	6
ss	Second, using two digits	06
S, SS, ..., SSSSSSSSS	Fractional second, using the number of digits specified by the number of instances of 'S' (up to 9 digits).	'SSS' truncates 6.12345 seconds to 123.

Some tips and considerations:

- Use one or more u characters instead of y characters to represent the year when working with year numbers near zero.
- Datetime values later than 144683 years CE or before 140743 BCE display only the year numbers, regardless of the specified format value.

Time Zone Offset Formats

Use these identifiers to specify the display format of the time zone offset. A time zone offset is the amount of time that a specific date and time is offset from UTC. A time zone offset is different from a time zone in that it comprises rules that determine the offsets for specific times of the year. Include a time zone offset identifier when you want to ensure that the time components are displayed unambiguously.

Letter Identifier	Description	Display
`z`	Abbreviated name of the time zone offset. If this value is not available, then the time zone offset uses the short UTC format, such as `UTC-4`.	`EDT`
`Z`	ISO 8601 basic format with hours, minutes, and optional seconds fields.	`-0400`
`ZZZZ`	Long UTC format.	`UTC-04:00`
`ZZZZZ`	ISO 8601 extended format with hours, minutes, and optional seconds fields. A time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	`-04:00`
`x` or `X`	ISO 8601 basic format with hours field and optional minutes field. If you specify `X`, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	`-04`
`xx` or `XX`	ISO 8601 basic format with hours and minutes fields. If you specify `XX`, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	`-0400`

Letter Identifier	Description	Display
xxx or XXX	ISO 8601 extended format with hours and minutes fields. If you specify XXX, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-04:00
xxxx or XXXX	ISO 8601 basic format with hours, minutes, and optional seconds fields. If you specify XXXX, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-0400
xxxxx or XXXXX	ISO 8601 extended format with hours, minutes, and optional seconds fields. If you specify XXXXX, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-04:00

`durationfmt` — Format for duration values

character vector | string scalar

Format for duration values, specified as a character vector or string scalar.

To display a duration as a single number that includes a fractional part (for example, 1.234 hours), specify one of the following formats.

Format	Description
'y'	Number of exact, fixed-length years. A fixed-length year is equal to 365.2425 days.
'd'	Number of exact, fixed-length days. A fixed-length day is equal to 24 hours.
'h'	Number of hours
'm'	Number of minutes
's'	Number of seconds

To display a duration in the form of a digital timer, specify one of these formats:

- `'dd:hh:mm:ss'`
- `'hh:mm:ss'`
- `'mm:ss'`
- `'hh:mm'`

In addition, you can display up to nine fractional second digits by appending up to nine `S` characters. For example, `'hh:mm:ss.SSS'` displays the milliseconds of a duration value to three digits.

ax — Target axes
current axes (default) | single `Axes` object | vector of `Axes` objects

Target axes, specified as a single `Axes` object or a vector of `Axes` objects. If you do not specify the axes, then `xtickformat` uses the current axes.

Algorithms

The `xtickformat` function sets and queries the `TickLabelFormat` property of the ruler object associated with the *x*-axis.

See Also

Functions
`xlim` | `xtickangle` | `xticklabels` | `xticks` | `ytickformat` | `ztickformat`

Properties
DatetimeRuler | DurationRuler | NumericRuler

Introduced in R2016b

xticklabels

Set or query *x*-axis tick labels

Syntax

```
xticklabels(labels)
xl = xticklabels
```

```
xticklabels('auto')
xticklabels('manual')
m = xticklabels('mode')
```

```
___ = xticklabels(ax, ___ )
```

Description

`xticklabels(labels)` sets the *x*-axis tick labels on page 1-17145 for the current axes. Specify `labels` as a string array or a cell array of character vectors; for example, `{'January','February','March'}`. If you specify the labels, then the *x*-axis tick values and tick labels no longer update automatically based on changes to the axes.

`xl = xticklabels` returns the *x*-axis tick labels for the current axes.

`xticklabels('auto')` sets an automatic mode, enabling the axes to determine the *x*-axis tick labels. Use this option if you set the labels and then want to set them back to the default values.

`xticklabels('manual')` sets a manual mode, freezing the *x*-axis tick labels at the current values.

`m = xticklabels('mode')` returns the current value of the *x*-axis tick labels mode, which is either `'auto'` or `'manual'`. By default, the mode is automatic unless you specify the tick labels or set the mode to manual.

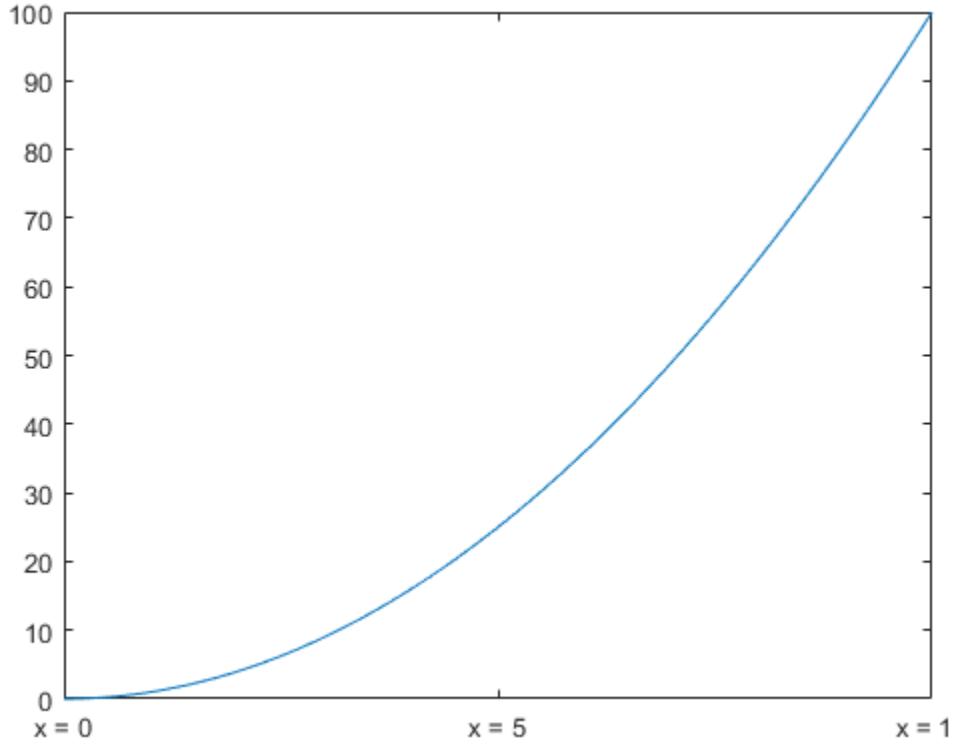
`___ = xticklabels(ax, ___ )` uses the axes specified by `ax` instead of the current axes. Specify `ax` as the first input argument for any of the previous syntaxes.

Examples

Specify *x*-Axis Tick Values and Labels

Create a line plot. Display tick marks along the *x*-axis at the values 0, 5, and 10. Then specify a label for each tick mark.

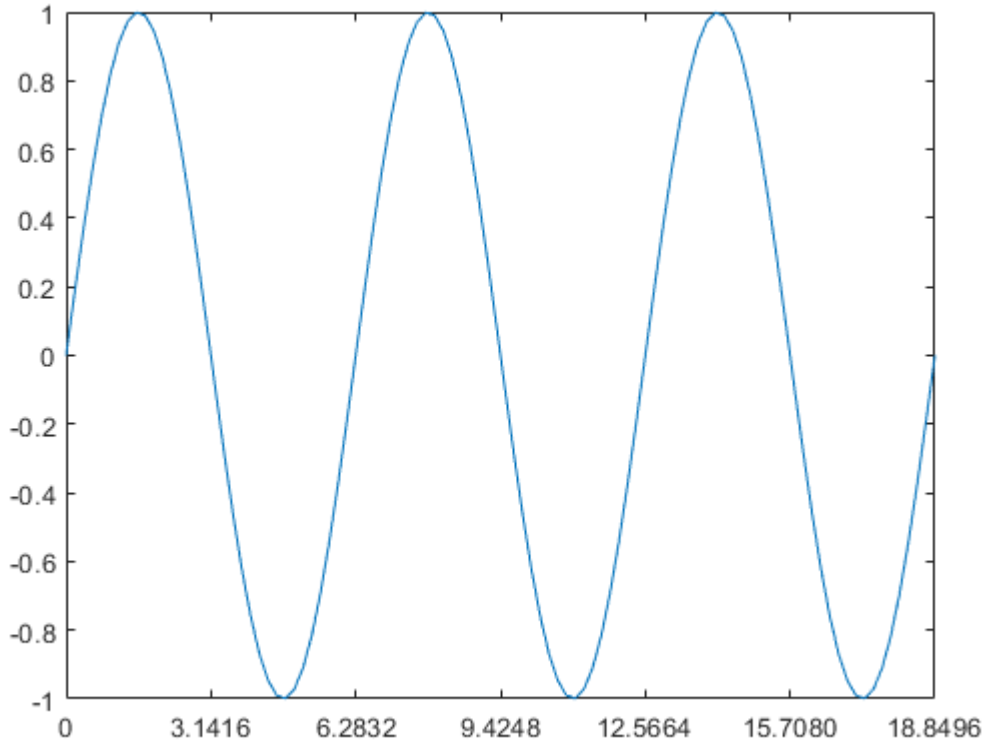
```
x = linspace(0,10);
y = x.^2;
plot(x,y)
xticks([0 5 10])
xticklabels({'x = 0','x = 5','x = 10'})
```



Display *x*-Axis Tick Labels in Terms of Pi

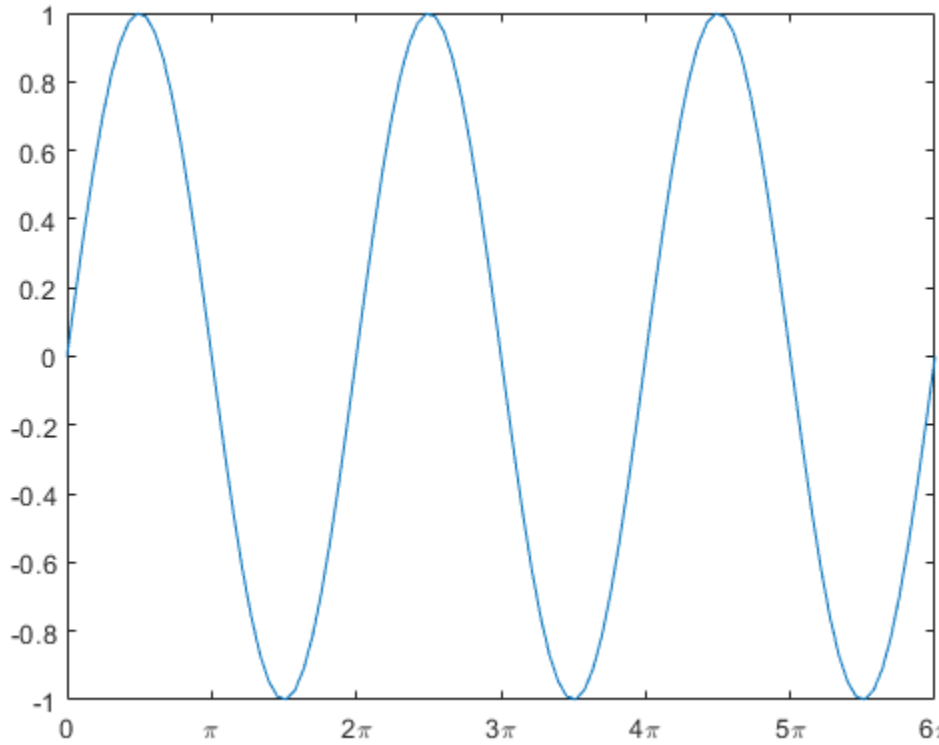
Create a line plot. Specify the *x*-axis limits and display tick marks along at the *x*-axis at increments of π.

```
x = linspace(0,6*pi);
y = sin(x);
plot(x,y)
xlim([0 6*pi])
xticks(0:pi:6*pi)
```



MATLAB® labels the tick marks with the numeric values. Change the labels to show the π symbol by specifying text for each label.

```
xticklabels({'0','\pi','2\pi','3\pi','4\pi','5\pi','6\pi'})
```

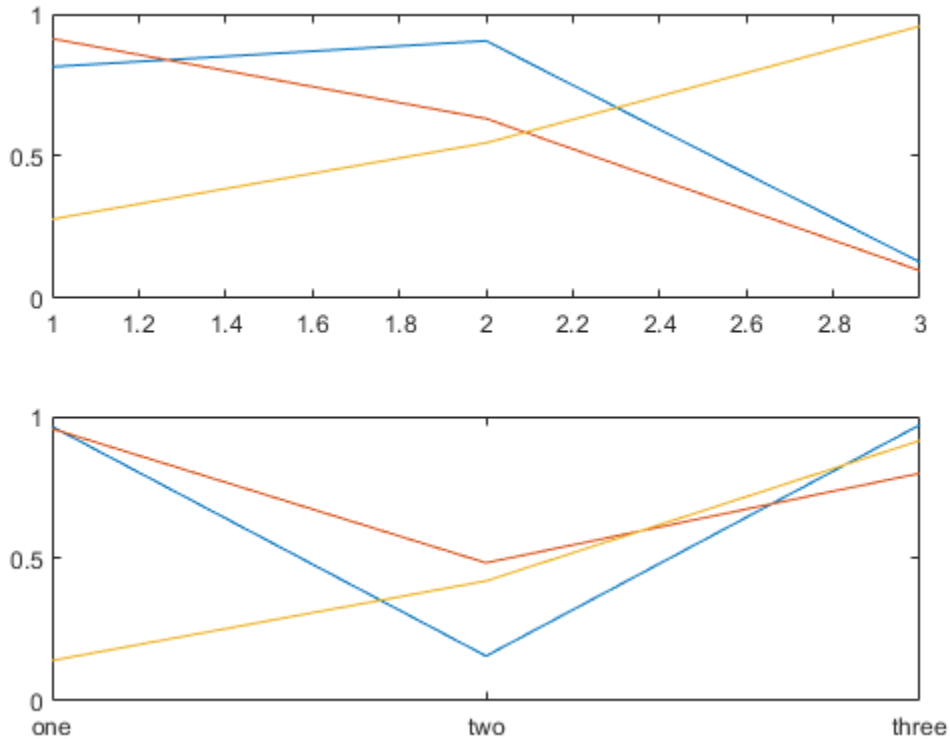


Specify *x*-Axis Tick Labels for Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Create two plots of

random data. Set the *x*-axis tick values and labels for the second plot by specifying `ax2` as the first input argument.

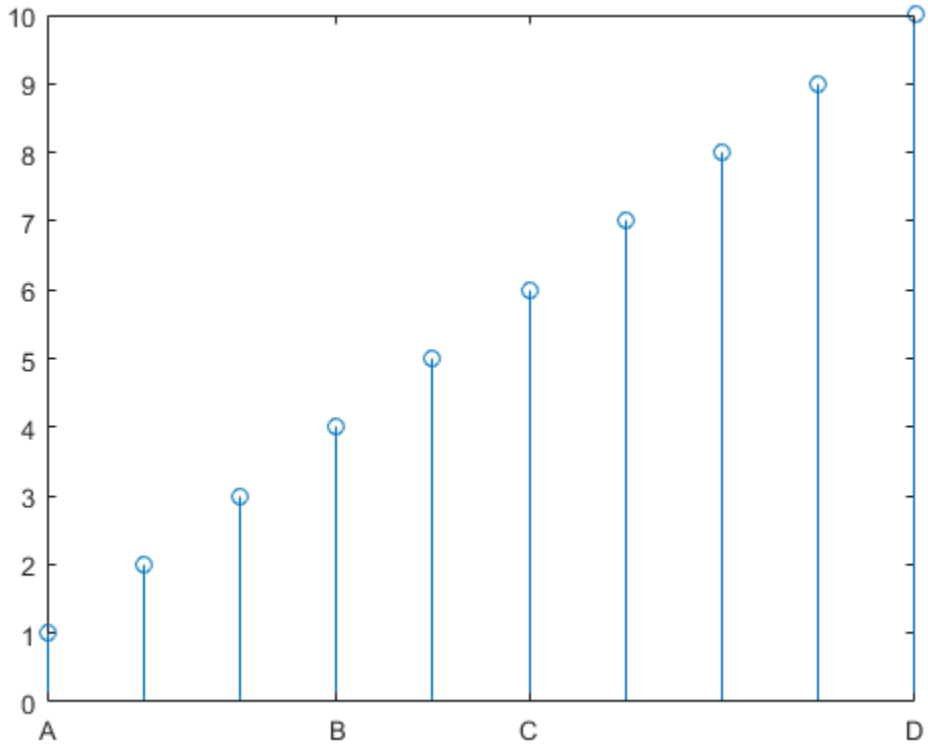
```
tiledlayout(2,1)
ax1 = nexttile;
plot(ax1,rand(3))

ax2 = nexttile;
plot(ax2,rand(3))
xticks(ax2,[1 2 3])
xticklabels(ax2,{'one','two','three'})
```



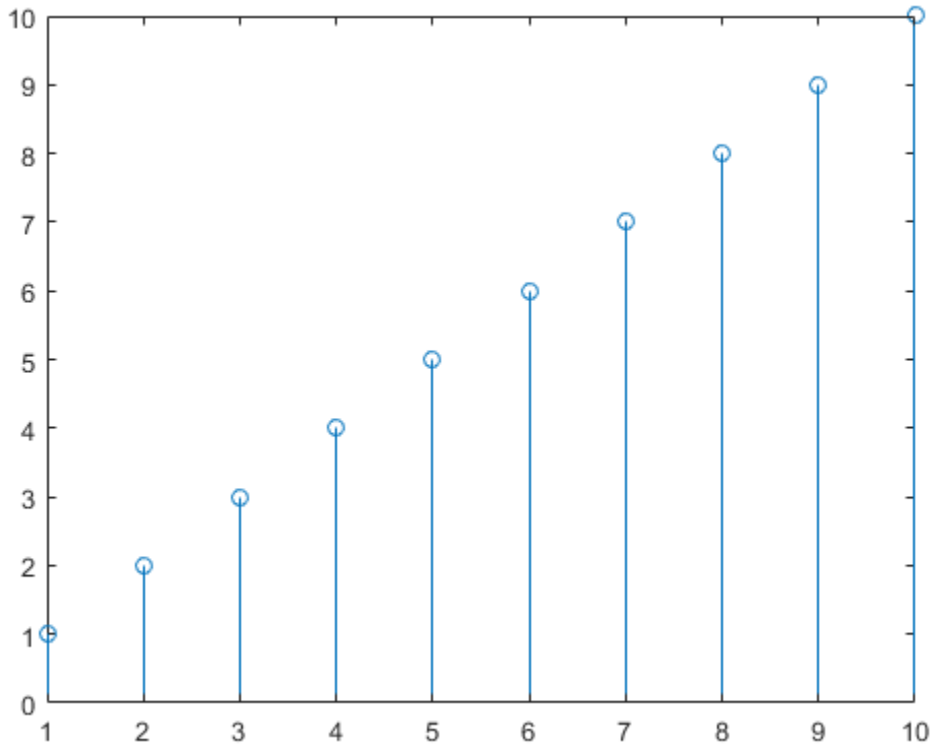
Set *x*-Axis Tick Labels Back to Default Labels

Create a stem chart, and specify the *x*-axis tick values and corresponding labels. Then set the *x*-axis tick values and labels back to the default values.

```
stem(1:10)
xticks([1 4 6 10])
xticklabels({'A','B','C','D'})
```



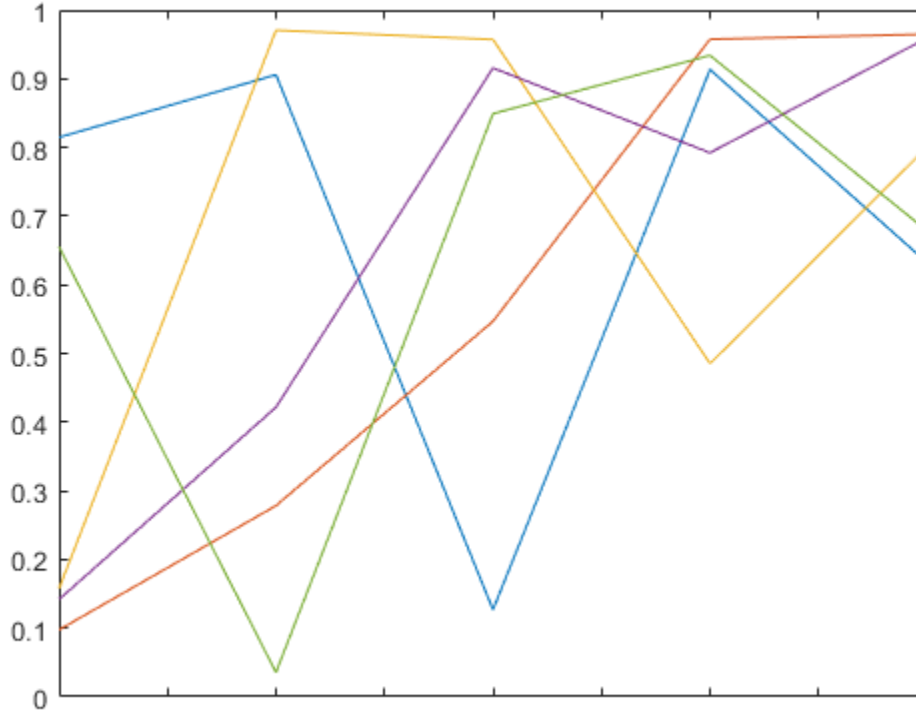

```
xticks('auto')
xticklabels('auto')
```



Remove *x*-Axis Tick Labels

Remove the tick labels along the *x*-axis by specifying the tick labels as an empty array.

```
plot(rand(5))
xticklabels({})
```



Input Arguments

`labels` — Tick labels

cell array of character vectors | string array | categorical array

Tick labels on page 1-17145, specified as a cell array of character vectors, string array, or categorical array. If you do not want tick labels to show, then specify an empty cell array `{}`. Tick labels support TeX and LaTeX markup. See the `TickLabelInterpreter` property of the `Axes` object for more information.

Example: `xticklabels({'0','\pi','2\pi'})`

Example: `xticklabels({'January','Febrary','March'})`

Example: `xticklabels({})`

Note

- If you specify the tick labels as a categorical array, MATLAB uses the values in the array, not the categories.
- To specify the tick values, use the `xticks` function.

---

ax — Target axes
current axes (default) | single `Axes` object | vector of `Axes` objects

Target axes, specified as a single `Axes` object or a vector of `Axes` objects. If you do not specify the axes, then `xticklabels` uses the current axes.

Output Arguments

`xl` — Current tick labels
cell array of character vectors | character array

Current tick labels, returned as a cell array of character vectors or a character array.

`m` — Current tick labels mode
`'auto'` | `'manual'`

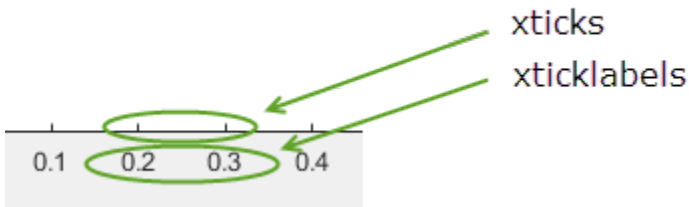
Current tick labels mode, returned as one of these values:

- `'auto'` — Automatically determine the *x*-axis tick labels.
- `'manual'` — Use manually specified *x*-axis tick labels.

More About

Tick Labels

The tick labels are the labels that you see next to each tick mark. The tick values are the locations along the *x*-axis where the tick marks appear. Set the values using the `xticks` function. Set the corresponding labels using the `xticklabels` function.



Algorithms

The `xticklabels` function sets and queries several axes properties related to the *x*-axis tick labels.

- `XTickLabel` — Property that stores the text for the *x*-axis tick labels.
- `XTickLabelMode` — Property that stores the *x*-axis tick label mode. When you set the *x*-axis tick labels using `xticklabels`, this property changes to `'manual'`.
- `XTickMode` — Property that stores the *x*-axis tick value mode. When you set the *x*-axis tick labels using `xticklabels`, this property changes to `'manual'`.

See Also

Functions
`xlim` | `xtickangle` | `xtickformat` | `xticks` | `yticklabels` | `zticklabels`

Properties
Axes

Topics

"Specify Axis Tick Values and Labels"

Introduced in R2016b

xticks

Set or query *x*-axis tick values

Syntax

```
xticks(ticks)
xt = xticks

xticks('auto')
xticks('manual')
m = xticks('mode')

___ = xticks(ax, ___ )
```

Description

`xticks(ticks)` sets the *x*-axis tick values on page 1-17160, which are the locations along the *x*-axis where the tick marks appear. Specify `ticks` as a vector of increasing values; for example, `[0 2 4 6]`. This command affects the current axes.

`xt = xticks` returns the current *x*-axis tick values as a vector.

`xticks('auto')` sets an automatic mode, enabling the axes to determine the *x*-axis tick values. Use this option if you change the tick values and then want to set them back to the default values.

`xticks('manual')` sets a manual mode, freezing the *x*-axis tick values at the current values. Use this option if you want to retain the current tick values when resizing the axes or adding new data to the axes.

`m = xticks('mode')` returns the current *x*-axis tick values mode, which is either `'auto'` or `'manual'`. By default, the mode is automatic unless you specify tick values or change the mode to manual.

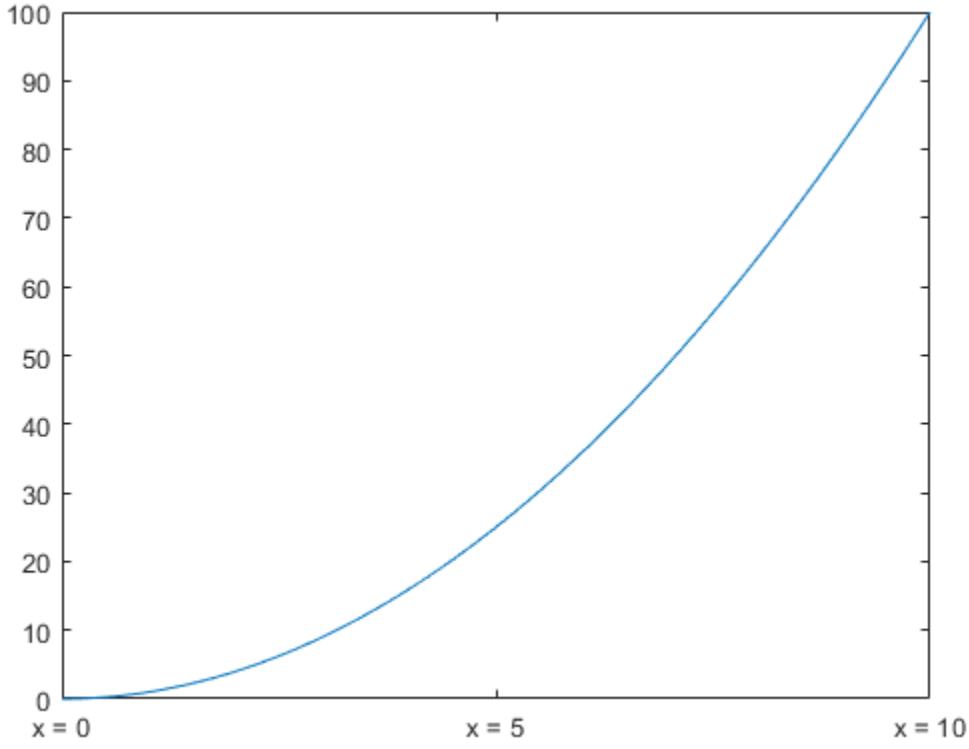
`___ = xticks(ax, ___ )` uses the axes specified by `ax` instead of the current axes. Specify `ax` as the first input argument for any of the previous syntaxes.

Examples

Specify *x*-Axis Tick Values and Labels

Create a line plot. Display tick marks along the *x*-axis at the values 0, 5, and 10. Then specify a label for each tick mark.

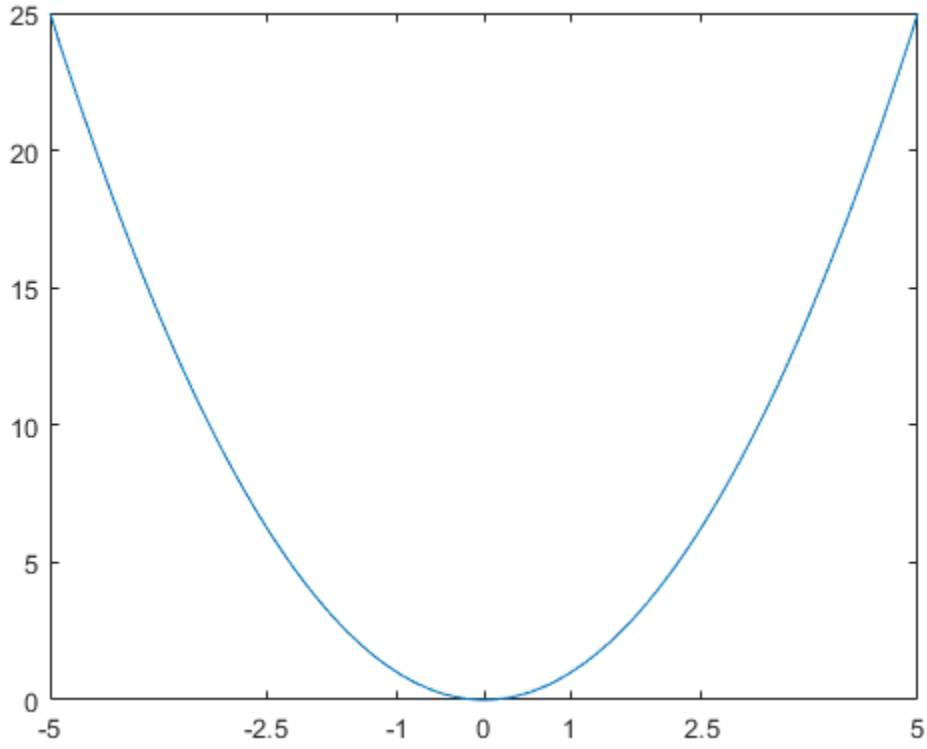
```
x = linspace(0,10);
y = x.^2;
plot(x,y)
xticks([0 5 10])
xticklabels({'x = 0','x = 5','x = 10'})
```



Specify Nonuniform *x*-Axis Tick Values

Display tick marks along the *x*-axis at nonuniform values between -5 and 5. MATLAB® labels the tick marks with the numeric values.

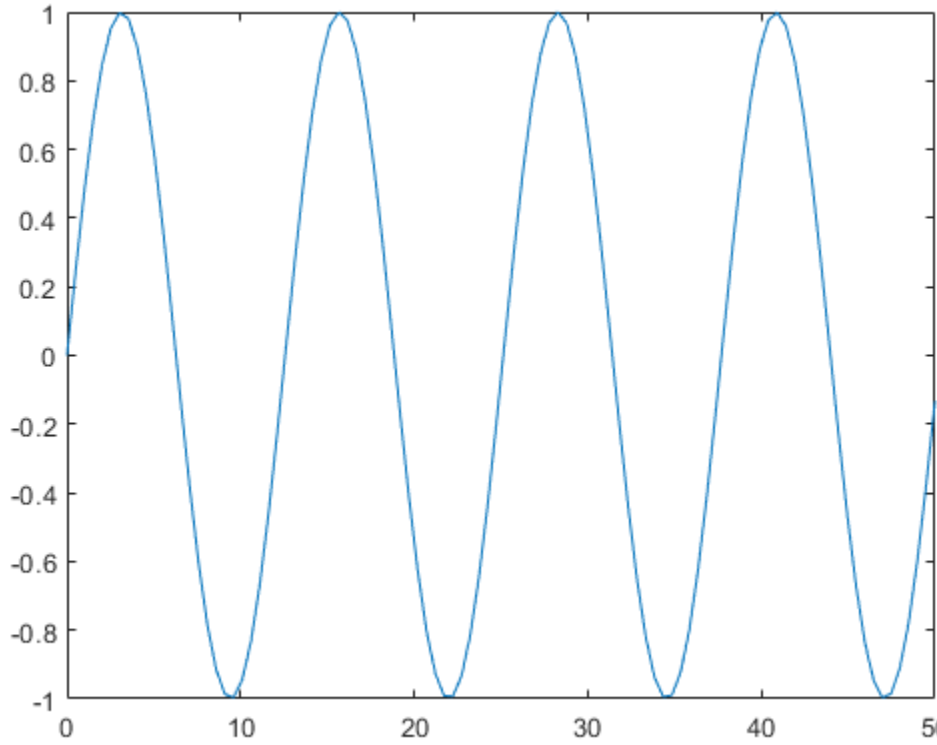
```
x = linspace(-5,5);
y = x.^2;
plot(x,y)
xticks([-5 -2.5 -1 0 1 2.5 5])
```



Increment *x*-Axis Tick Values by 10

Display tick marks along the *x*-axis at increments of 10, starting from 0 and ending at 50.

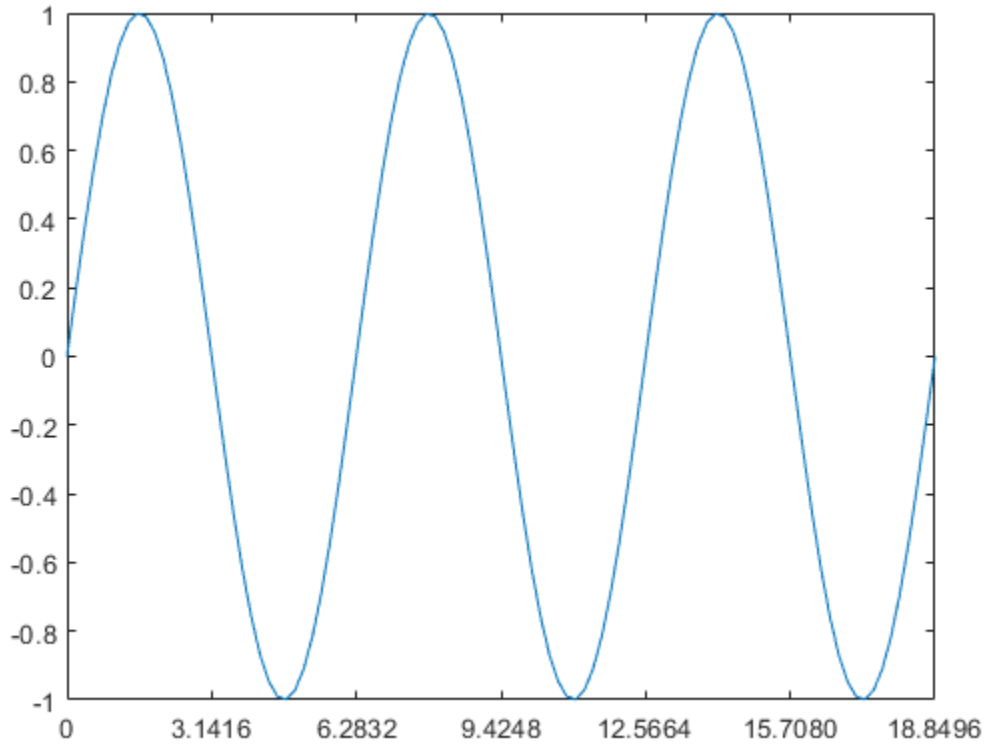
```
x = linspace(0,50);
y = sin(x/2);
plot(x,y)
xticks(0:10:50)
```



Increment *x*-Axis Tick Values by Pi

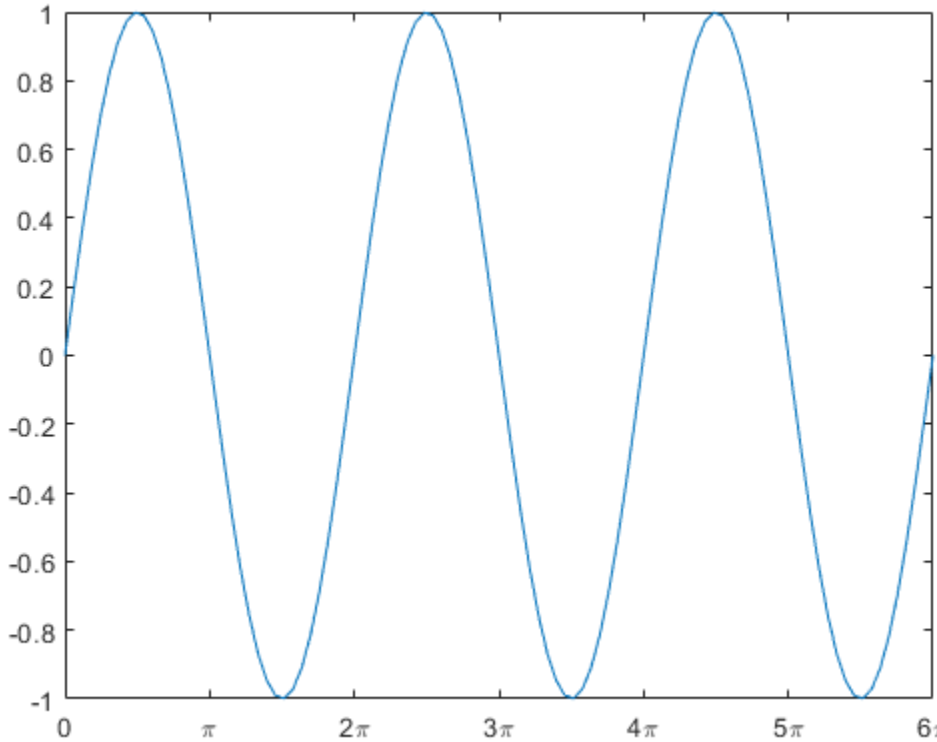
Create a line plot. Specify the *x*-axis limits as 0 to 6*π*. Then, display tick marks along the *x*-axis at increments of *π*.

```
x = linspace(0,6*pi);
y = sin(x);
plot(x,y)
xlim([0 6*pi])
xticks(0:pi:6*pi)
```



MATLAB® labels the tick marks with the numeric values. Change the labels to show the *π* symbol by specifying the text for each label.

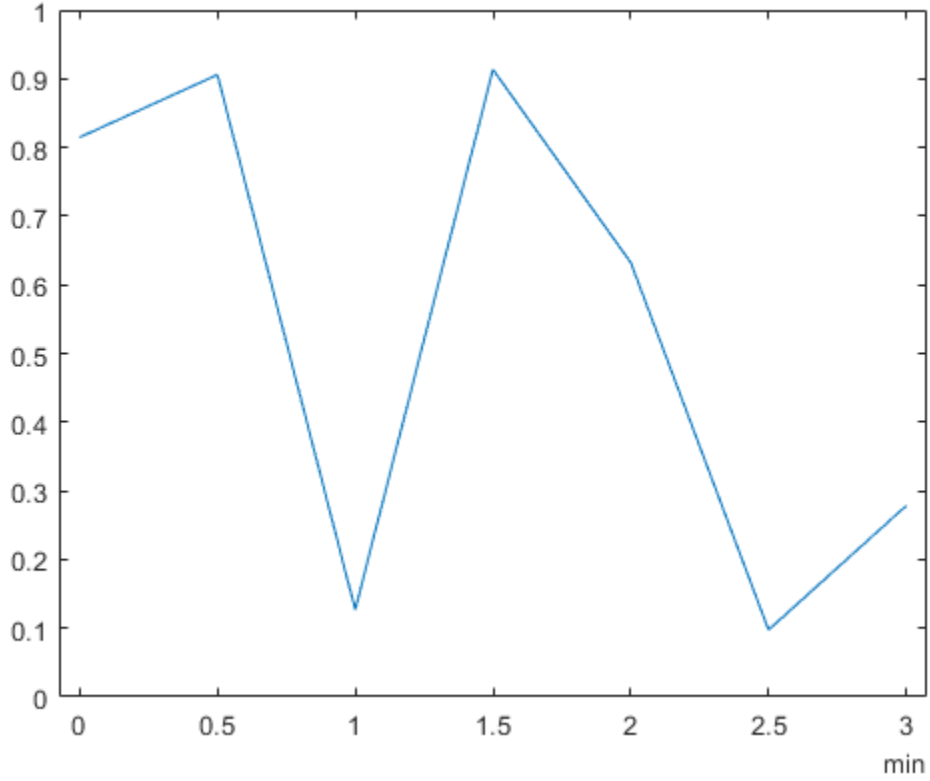
```
xticklabels({'0','\pi','2\pi','3\pi','4\pi','5\pi','6\pi'})
```



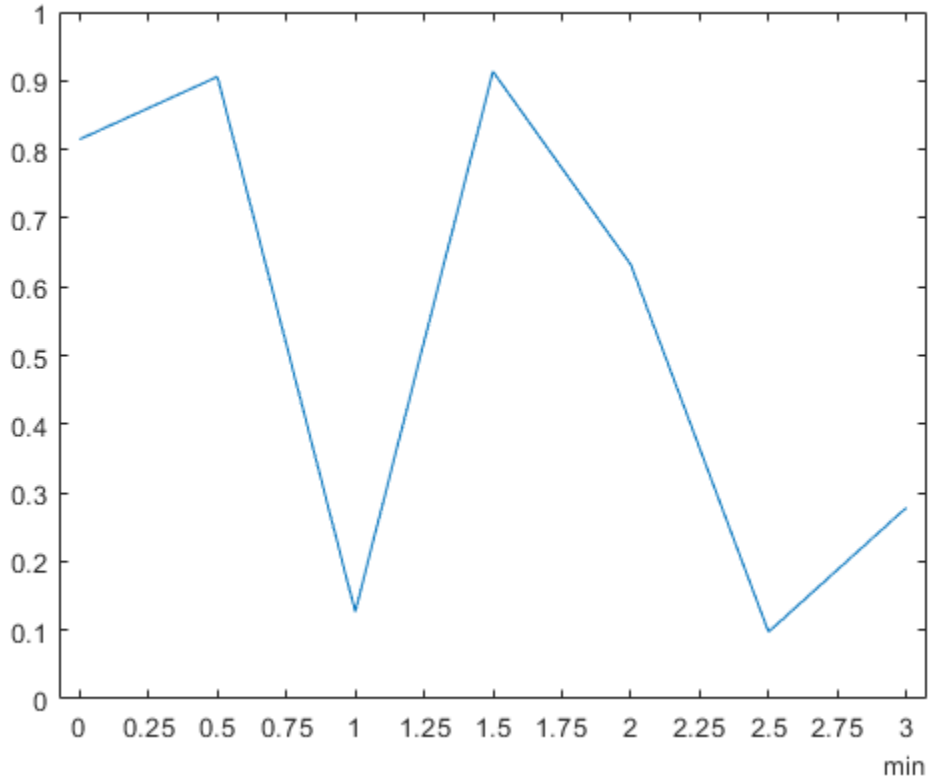
Change Tick Values for *x*-Axis with Durations

Create a plot with duration values along the *x*-axis. Then, change the duration values where the tick marks are located.

```
t = minutes(0:.5:3);
y = rand(1,7);
plot(t,y)
```



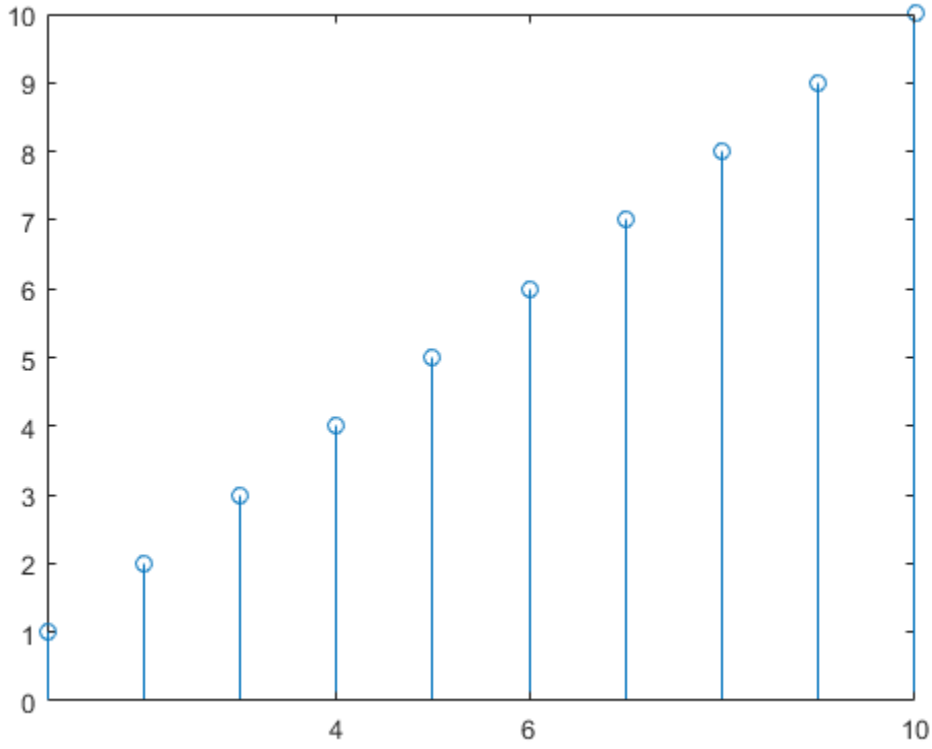

```
ticks = minutes(0:.25:3);
xticks(ticks)
```



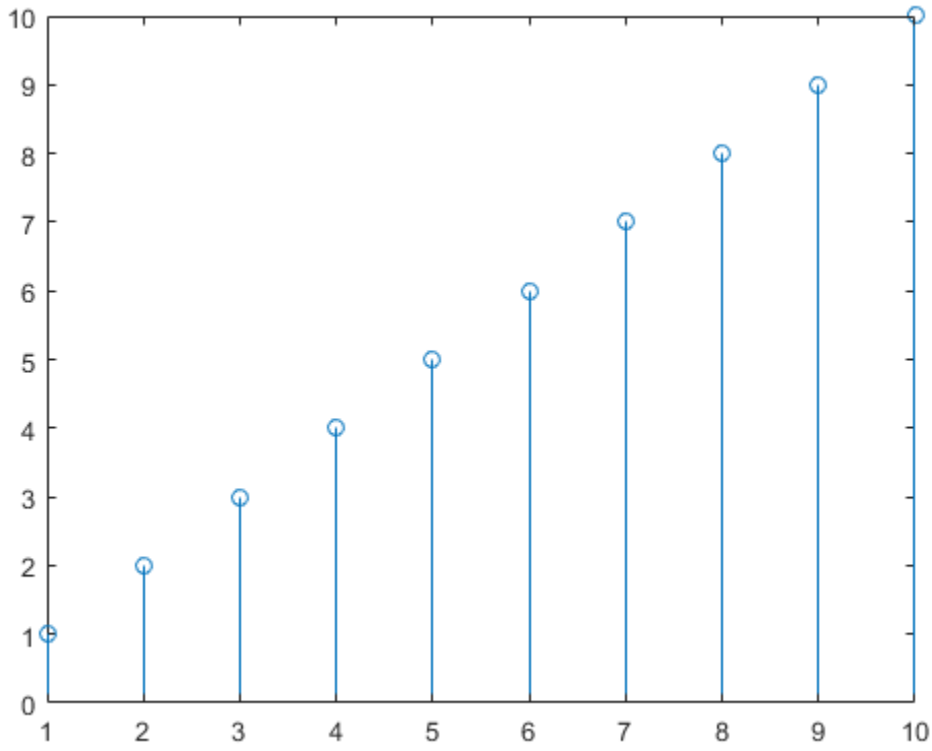
Set *x*-Axis Tick Values Back to Default Values

Create a stem chart and specify the *x*-axis tick values. Then, set the *x*-axis tick values back to the default values.

```
stem(1:10)
xticks([0 4 6 10])
```




```
xticks('auto')
```

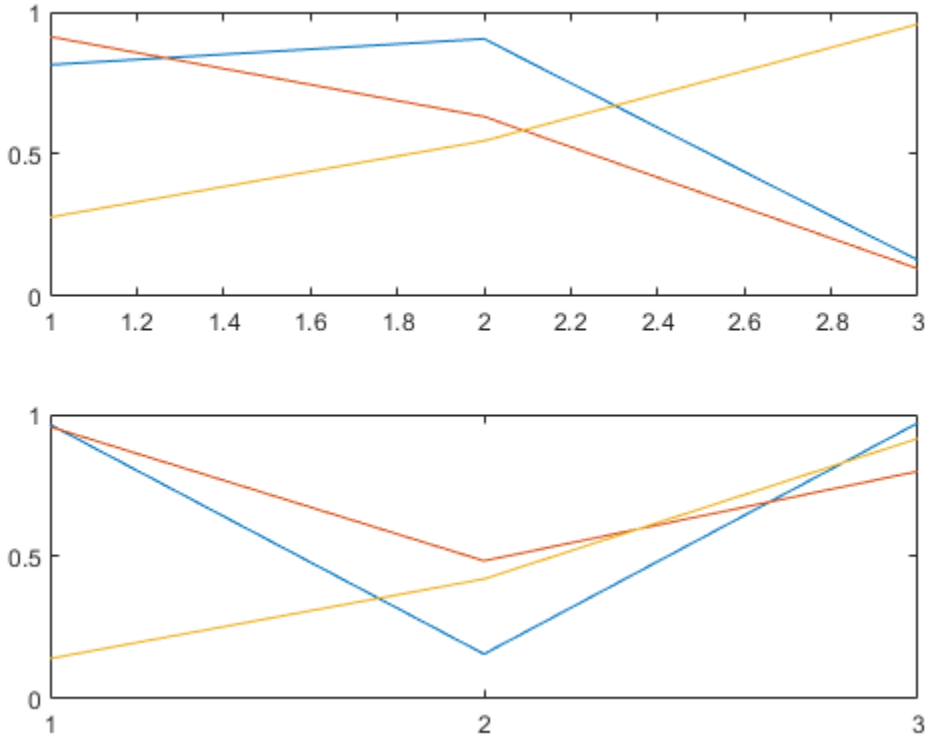


Specify *x*-Axis Tick Values for Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Plot random data into each axes. Then set the *x*-axis tick values for the lower plot by passing `ax2` as the first input argument to the `xticks` function.

```
tiledlayout(2,1)
ax1 = nexttile;
plot(rand(3))
```

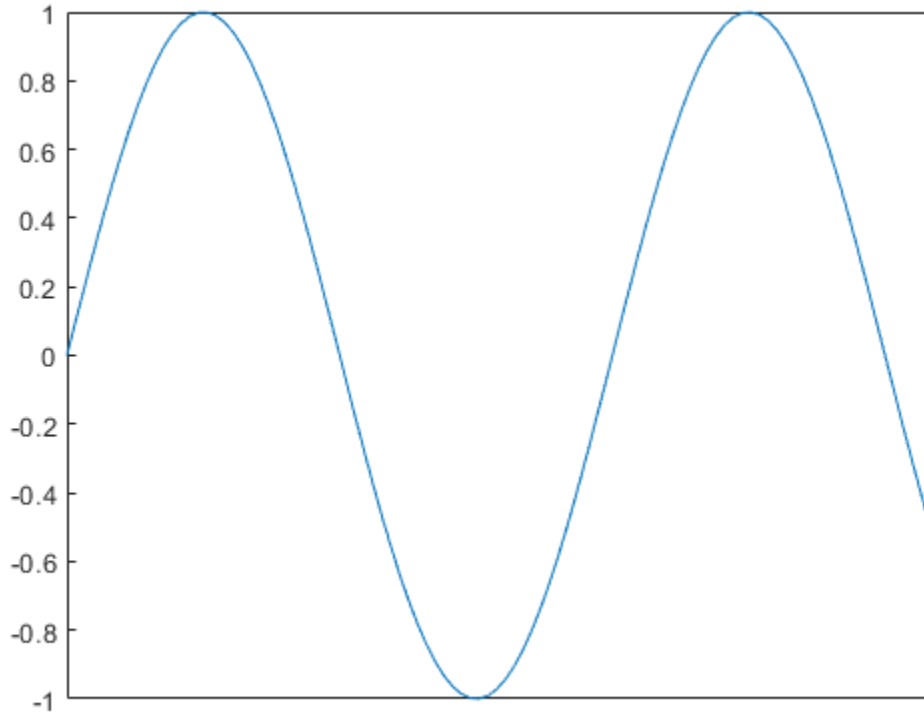
```
ax2 = nexttile;
plot(rand(3))
xticks(ax2,[1 2 3])
```



Remove *x*-Axis Tick Marks

Remove the tick marks along the *x*-axis by specifying the tick values as an empty array.

```
x = linspace(0,10);
y = sin(x);
plot(x,y)
xticks([])
```



Input Arguments

`ticks` — Tick values
vector of increasing values

Tick values on page 1-17160, specified as a vector of increasing values. If you do not want tick marks along the *x*-axis, specify an empty vector `[]`.

You can specify the tick values as numeric, categorical, datetime, or duration values. However, the type of values that you specify must match the type of values along the *x*-axis.

Example: `xticks([pi 2*pi 3*pi 4*pi])`

Example: `xticks(0:10:100)`

Example: `xticks([])`

Note To specify the tick labels, use the `xticklabels` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

ax — Target axes
current axes (default) | single `Axes` object | vector of `Axes` objects

Target axes, specified as a single `Axes` object or a vector of `Axes` objects. If you do not specify the axes, then `xticks` uses the current axes.

Output Arguments

`xt` — Current tick values
vector

Current tick values, returned as a vector.

`m` — Current mode
`'auto'` | `'manual'`

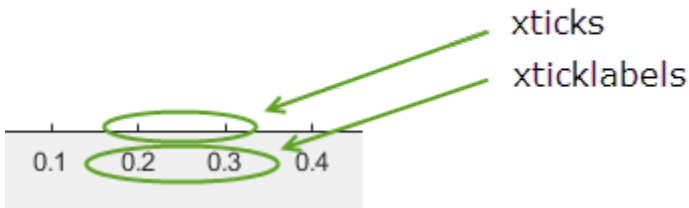
Current mode, returned as one of these values:

- `'auto'` — Automatically determine the *x*-axis tick values.
- `'manual'` — Use manually specified *x*-axis tick values.

More About

Tick Values

The tick values are the locations along the *x*-axis where the tick marks appear. The tick labels are the labels that you see next to each tick mark. Set the values using the `xticks` function. Set the corresponding labels using the `xticklabels` function.



Algorithms

The `xticks` function sets and queries several axes properties related to the *x*-axis tick values.

- `XTick` — Property that stores the *x*-axis tick values.
- `XTickMode` — Property that stores the *x*-axis tick value mode. When you set the *x*-axis tick values, this property changes to `'manual'`.

See Also

Functions
`xlim` | `xtickangle` | `xtickformat` | `xticklabels` | `yticks` | `zticks`

Properties
Axes

Topics
"Specify Axis Tick Values and Labels"

Introduced in R2016b

matlab.wsdl.createWSDLClient

Create interface to SOAP-based web service

Syntax

```
matlab.wsdl.createWSDLClient(wsdlURL)
matlab.wsdl.createWSDLClient(wsdlURL,folder)
matlab.wsdl.createWSDLClient(wsdlURL,folder,'silent')

matlab.wsdl.createWSDLClient( ___ ,options)

classname = matlab.wsdl.createWSDLClient( ___ )
```

Description

`matlab.wsdl.createWSDLClient(wsdlURL)` creates an interface to a service based on a Web Services Description Language (WSDL) document specified by `wsdlURL`. This function creates a MATLAB class file for each Simple Object Access Protocol (SOAP) binding in the WSDL in the current folder. If necessary, the function creates additional support files. You can package and distribute these files to other users.

You must install the WSDL tools, Java JDK™ and Apache CXF software, then set the tool paths using the `matlab.wsdl.setWSDLToolPath` function.

You must have write permission for the current folder.

`matlab.wsdl.createWSDLClient(wsdlURL,folder)` creates the interface files in `folder`, which must be on the MATLAB path.

`matlab.wsdl.createWSDLClient(wsdlURL,folder,'silent')` suppresses display of generated files and folders.

`matlab.wsdl.createWSDLClient( ___ ,options)` specifies additional information needed to access the WSDL document. You can use this syntax with any of the input arguments of the previous syntaxes.

`classname = matlab.wsdl.createWSDLClient( ___ )` returns a handle to the created class. If multiple classes were created, the function returns a cell array of handles. You can use this syntax with any of the input arguments of the previous syntaxes.

To get information on using the class, call the MATLAB `help` function on the class name.

Examples

Get Map Name from USGSImageryOnly_MapServer Web Service

To run this example, install the WSDL tools and set the tool paths. `jdk` and `cxf` are the paths to these tools on your system.

```
matlab.wsdl.setWSDLToolPath('JDK',jdk,'CXF',cxf)
```

Create the class files.

```
url = ...
'http://basemap.nationalmap.gov/arcgis/services/USGSImageryOnly/MapServer?wsdl';
matlab.wsdl.createWSDLClient(url)
```

```
Created USGSImageryOnly_MapServer.
  .\USGSImageryOnly_MapServer.m
  .\+wsdl

In order to use USGSImageryOnly_MapServer, you must run javaaddpath('.\+wsdl\mapserver.jar').
```

Add the jar files to the Java path.

```
javaaddpath('.\+wsdl\mapserver.jar')
```

Create the service.

```
wsdl = USGSImageryOnly_MapServer;
```

Read help for the service and its functions.

```
help USGSImageryOnly_MapServer
```

```
USGSImageryOnly_MapServer   A client to connect to the USGSImageryOnly_MapServer service
     SERVICE = USGSImageryOnly_MapServer  connects to http://basemap.nationalmap.gov/arcgis/services/USGSImageryOnly/MapSe

     To communicate with the service, call a function on the SERVICE:

        [...] = FUNCTION(SERVICE,arg,...)

     See doc USGSImageryOnly_MapServer for a list of functions.
```

Call one of the methods, for example `GetDefaultMapName` that returns the map name.

```
GetDefaultMapName(wsdl)
```

```
ans =

Layers
```

Input Arguments

`wsdlURL` — WSDL URL or file path

string | character vector

WSDL URL or file path, specified as a string or a character vector, that defines service methods, arguments, and transactions.

`wsdlURL` can be an `http` or `https` URL or a local path. `wsdlURL` cannot be a `file://` URL. On Microsoft Windows, UNC paths are not supported.

Example: `'http://www.mywebservice.com/servicename?WSDL'`

`folder` — Folder for generated files

string | character vector

Folder for generated files, specified as a string or a character vector. If omitted or empty (`''`), `matlab.wsdl.createWSDLClient` uses the current folder. You must have write permission for the folder. The function overwrites existing files with the same names as the generated files.

Example: `'c:\work'`

`options` — Additional options

`weboptions` object

Additional options, specified as a `weboptions` object, needed to access the WSDL document, such as `Username`, `Password`, and `Timeout`.

The additional options specified in `weboptions` do not apply to any other documents, such as other imported WSDL documents or schemas, that the primary WSDL document references. The options also do not apply to requests made to servers by the generated code.

Example: `'Username','john','Password','mypassword'`

Limitations

- “Limitations to WSDL Document Support”

Tips

- If you create WSDL files in multiple locations on your computer, avoid confusion by deleting the class files from duplicate locations, and then call `clear java`.

See Also

`clear` | `createClassFromWsdl` | `matlab.wsdl.setWSDLToolPath` | `weboptions` | `webread`

Topics

“Set Up WSDL Tools”

Introduced in R2014b

matlab.wsdl.setWSDLToolPath

Location of WSDL tools

Syntax

```
matlab.wsdl.setWSDLToolPath(Name,Value)
```

```
paths = matlab.wsdl.setWSDLToolPath
```

Description

`matlab.wsdl.setWSDLToolPath(Name,Value)` sets the paths to Java JDK and Apache CXF software. You must download these tools to use the `matlab.wsdl.createWSDLClient` interface.

Specify values for both `Name` arguments `'JDK'` and `'CXF'` before calling `matlab.wsdl.createWSDLClient`. The values are saved across sessions in your user preferences.

`paths = matlab.wsdl.setWSDLToolPath` displays paths to the JDK and CXF software.

Examples

Set Path to Tools

Set `jdk` and `cxf` to valid paths on your system. For example,

```
jdk = 'E:/Program Files/win64/jdk';
cxf = 'c:\apache-cxf-3.2.4'
matlab.wsdl.setWSDLToolPath('JDK',jdk,'CXF',cxf)
```

Check If Tool Paths Are Set

```
p = matlab.wsdl.setWSDLToolPath;
if (isempty(p.JDK) || isempty(p.CXF))
    disp('Install the Java Development Kit (JDK) and Apache CXF programs.')
    disp('See the Set Up WSDL Tools example link at the end of this page.')
else
    disp('Paths set to:')
    matlab.wsdl.setWSDLToolPath
end
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'CXF','C:\apache-cxf-3.2.4'`

JDK — Path to Java JDK software
character vector

Path to Java Development Kit (JDK) software, specified as a character vector. Download the software from https://adoptopenjdk.net/. Choose OpenJDK™ 8 (Hot Spot).

Example: `'E:/Program Files/win64/jdk'`

CXF — Path to Apache CXF software
character vector

Path to Apache CXF software, specified as a character vector. CXF is an open-source services framework. Download the software from https://cxf.apache.org/download and choose the latest release of 3.2.

Example: `'C:\apache-cxf-3.2.4'`

Output Arguments

paths — Tool paths
structure

Tool paths, returned as a structure with the fields:

`JDK` — Path to Java JDK software
character vector

Path to Java JDK software, specified as a character vector. If the value of `JDK` is empty, you cannot call `matlab.wsdl.createWSDLClient`.

`CXF` — Path to Apache CXF software
character vector

Path to Apache CXF software, specified as a character vector. If the value of `CXF` is empty, you cannot call `matlab.wsdl.createWSDLClient`.

See Also

`matlab.wsdl.createWSDLClient`

Topics

"Set Up WSDL Tools"

External Websites

https://adoptopenjdk.net/
https://cxf.apache.org/download

Introduced in R2014b

year

Year number

Syntax

```
y = year(t)
y = year(t,yearType)
```

Description

`y = year(t)` returns the ISO year numbers for the datetime values in `t`. ISO year numbers include a year zero and represent years BCE using negative values. The `y` output is a `double` array the same size as `t`.

`y = year(t,yearType)` returns the type of year number specified by `yearType`.

The `year` function returns the year numbers of datetime values. To assign ISO year values to a datetime array, `t`, use `t.Year` and modify the `Year` property.

Examples

Extract Year Number from Dates

Extract the year numbers from an array of dates.

```
t = datetime(2010,05,31):calyears(1):datetime(2015,02,28)
```

```
t = 1x5 datetime array
   31-May-2010   31-May-2011   31-May-2012   31-May-2013   31-May-2014
```

```
y = year(t)
```

```
y = 1×5
```

```
2010        2011        2012        2013        2014
```

Input Arguments

t — Input date and time
`datetime` array

Input date and time, specified as a `datetime` array.

yearType — Type of year values
`'iso'` (default) | `'gregorian'`

Type of year values, specified as either `'iso'` or `'gregorian'`.

- If `yearType` is `'iso'`, then `year` returns the ISO year number, which includes a year zero and represents years BCE using negative values.
- If `yearType` is `'gregorian'`, then `year` returns the Gregorian year number, which is an unsigned integer. For example, the Gregorian year number for 5 CE and 5 BCE is 5 in both cases. Gregorian year numbers do not have a year zero.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`datetime` | `day` | `month` | `quarter` | `week` | `ymd`

Introduced in R2014b

years

Duration in years

Syntax

```
Y = years(X)
```

Description

`Y = years(X)` returns an array of years equivalent to the values in `X`.

- If `X` is a numeric array, then `Y` is a `duration` array in units of fixed-length years. A fixed-length year is equal to 365.2425 days.
- If `X` is a `duration` array, then `Y` is a `double` array with each element equal to the number of fixed-length years in the corresponding element of `X`.

The `years` function converts between `duration` and `double` values. To display a duration in units of years, set its `Format` property to `'y'`.

Leap years have one extra day. To create an array of calendar years that account for leap years when used in calendar calculations, use the `calyears` function.

Examples

Create Duration Array of Fixed-Length Years

```
X = [1 3 5; 10 12 15]
```

```
X = 2×3

     1     3     5
    10    12    15
```

```
Y = years(X)
```

```
Y = 2x3 duration array
     1 yr    3 yrs    5 yrs
   10 yrs   12 yrs   15 yrs
```

Convert Durations to Numeric Array of Years

Find the difference between two arrays of dates. The output is a `duration` array.

```
t1 = datetime(2007:2010,10,1);
t2 = datetime(2014,05,1);
dt = t2 - t1
```

```
dt = 1x4 duration array
   57696:00:00   48912:00:00   40152:00:00   31392:00:00
```

Convert each duration in `dt` to a number of fixed-length years.

```
Y = years(dt)
```

```
Y = 1×4

    6.5819    5.5799    4.5805    3.5812
```

Input Arguments

X — Input array
numeric array | duration array | logical array

Input array, specified as a numeric array, duration array, or logical array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This function fully supports code generation. For more information, see “Code Generation for Duration Arrays” (MATLAB Coder).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`calyears`

Introduced in R2014b

ylabel

Label y-axis

Syntax

```
ylabel(txt)
ylabel(target,txt)

ylabel( ___ ,Name,Value)
t = ylabel( ___ )
```

Description

`ylabel(txt)` labels the *y*-axis of the current axes or chart returned by the `gca` command. Reissuing the `ylabel` command causes the new label to replace the old label.

`ylabel(target,txt)` adds the label to the specified target object.

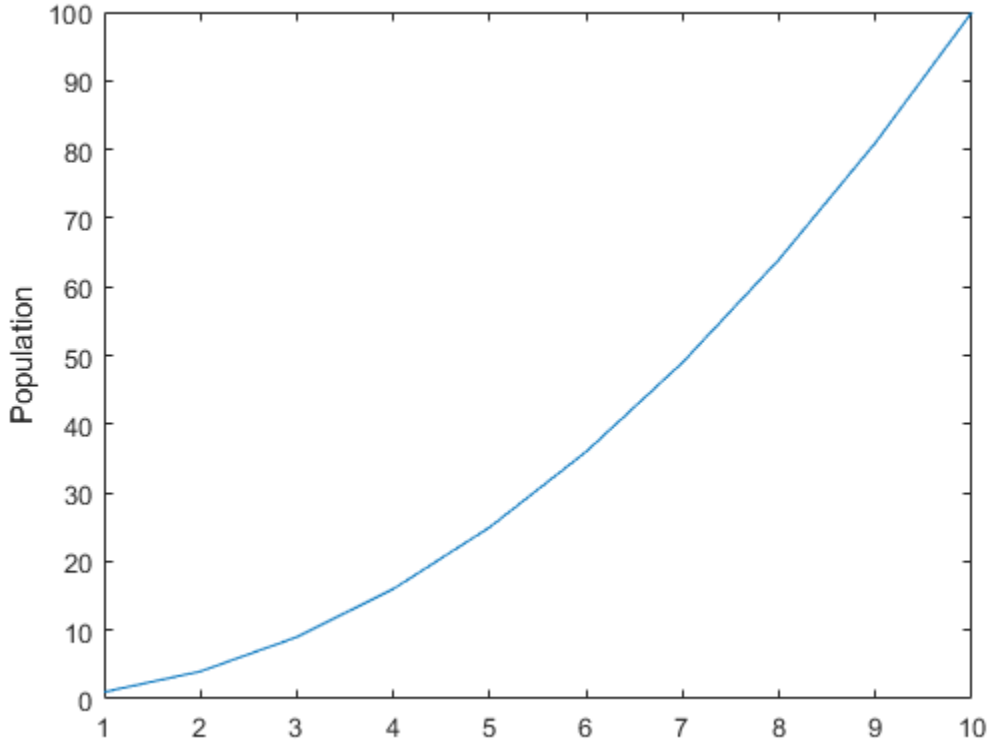
`ylabel( ___ ,Name,Value)` modifies the label appearance using one or more name-value pair arguments. For example, `'FontSize',12` sets the font size to 12 points. Specify name-value pair arguments after all other input arguments. Modifying the label appearance is not supported for all types of charts.

`t = ylabel( ___ )` returns the text object used as the *y*-axis label. Use `t` to make future modifications to the label after it is created.

Examples

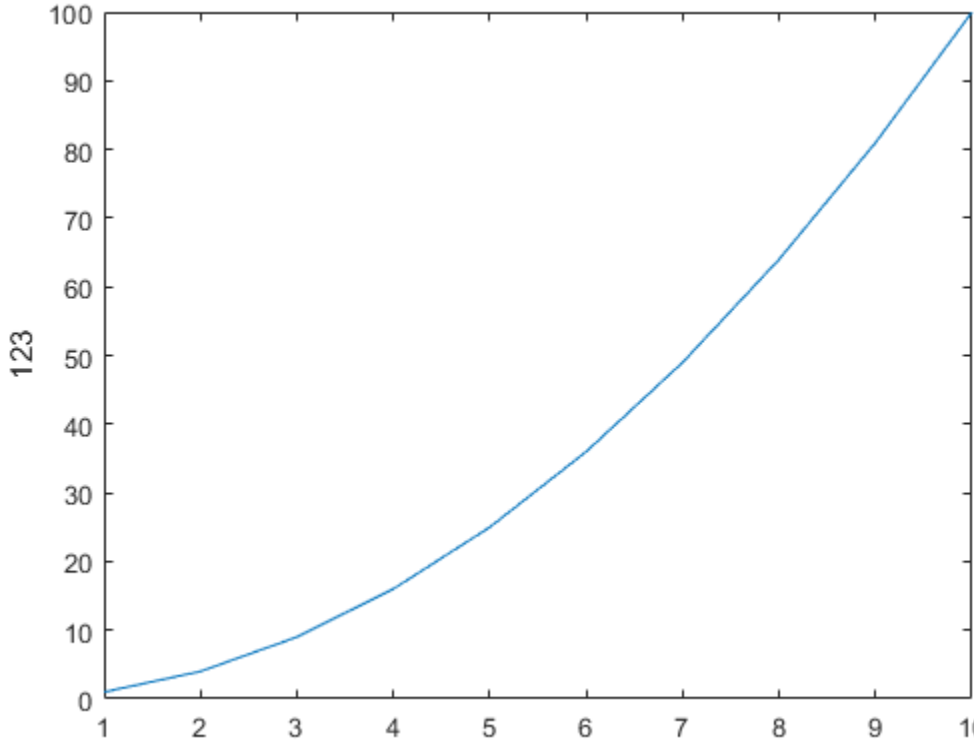
Label y-Axis

```
plot((1:10).^2)
ylabel('Population')
```



Label y-Axis with Numeric Input

```
figure
plot((1:10).^2)
ylabel(123)
```

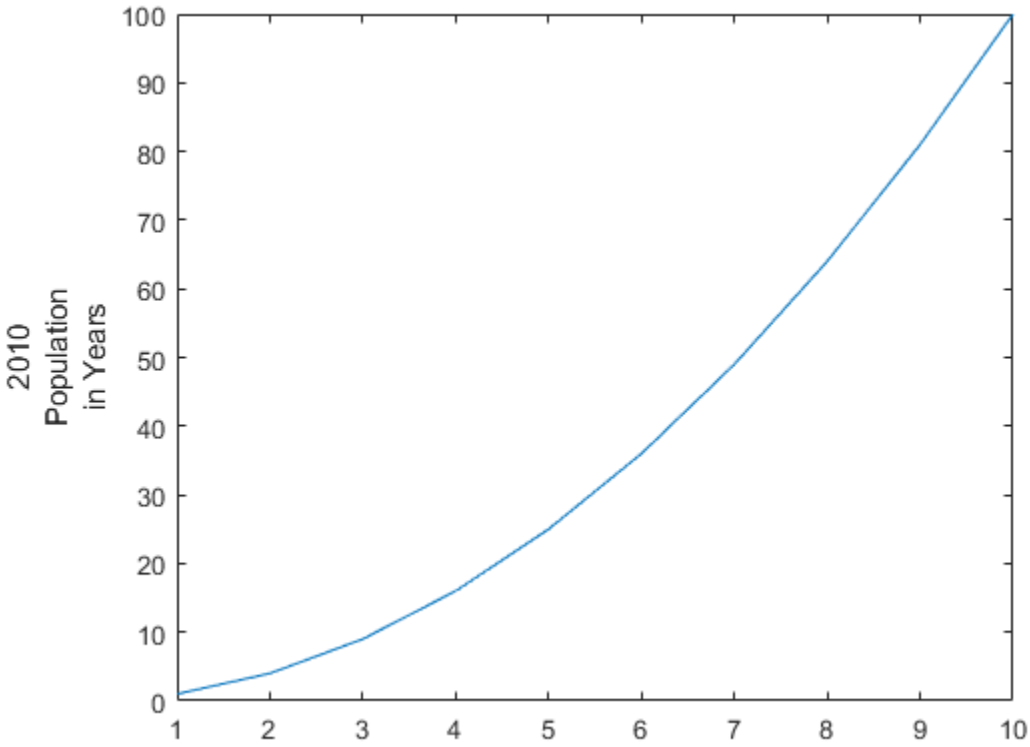


MATLAB® displays 123 beside the *y*-axis.

Create Multiline Label

Create a multiline label using a multiline cell array.

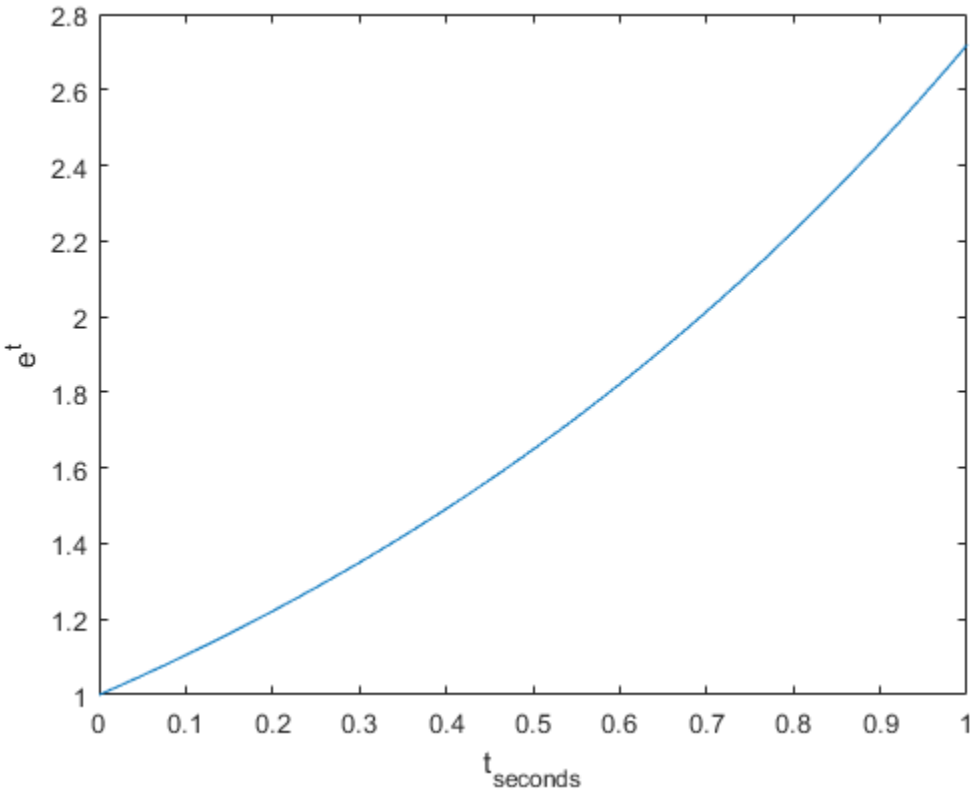
```
figure
plot((1:10).^2)
ylabel({2010;'Population';'in Years'})
```



Include Superscript and Subscript in Axis Labels

Use the `'^'` and `'_'` characters to include superscripts and subscripts in the axis labels. Use curly braces `{}` to modify more than one character.

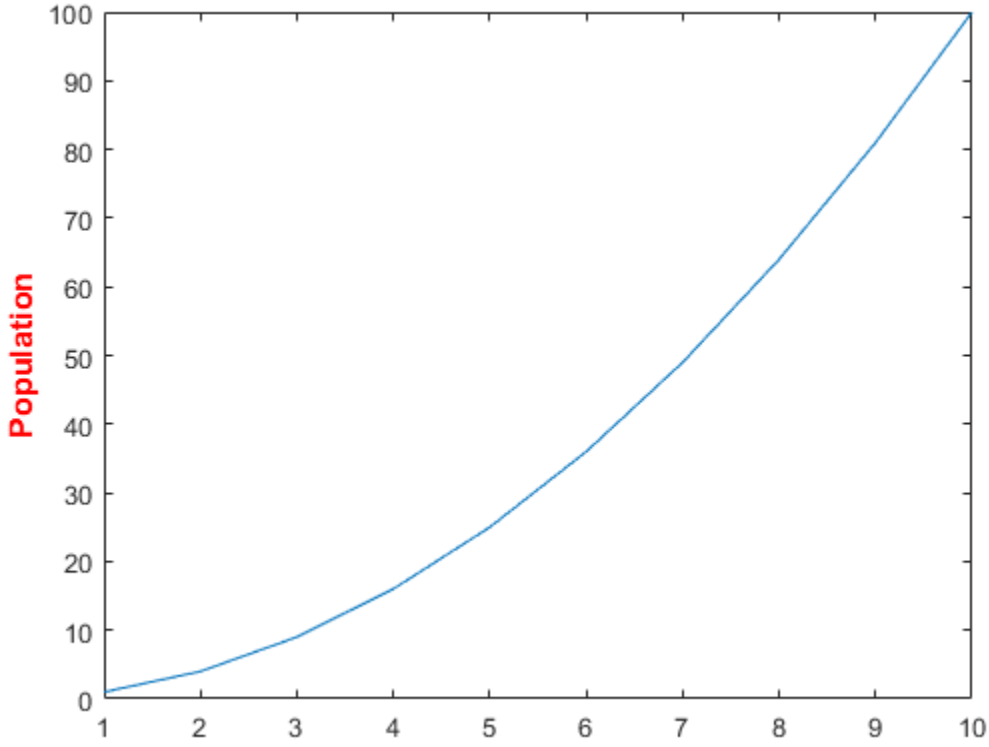
```
t = linspace(0,1);
y = exp(t);
plot(t,y)
xlabel('t_{seconds}')
ylabel('e^t')
```



Create y-Axis Label and Set Font Properties

Use `Name,Value` pairs to set the font size, font weight, and text color properties of the *y*-axis label.

```
figure
plot((1:10).^2)
ylabel('Population','FontSize',12,...
       'FontWeight','bold','Color','r')
```

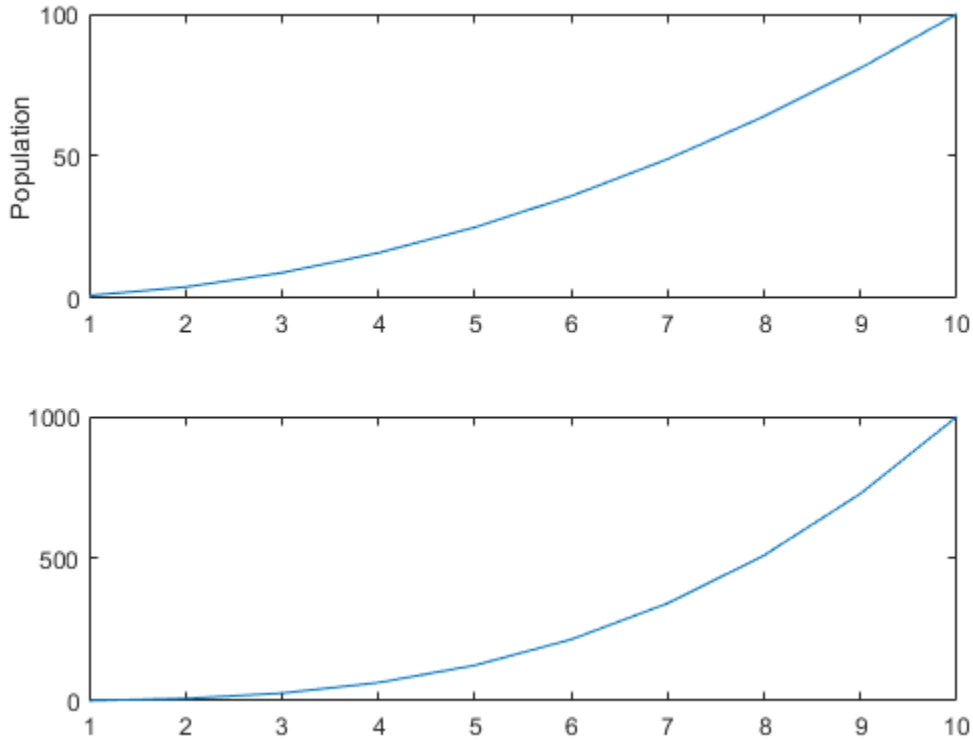


`'FontSize',12` displays the label text in 12-point font. `'FontWeight','bold'` makes the text bold. `'Color','r'` sets the text color to red.

Label y-Axis of Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Plot data into each axes, and create an *y*-axis label for the top plot.

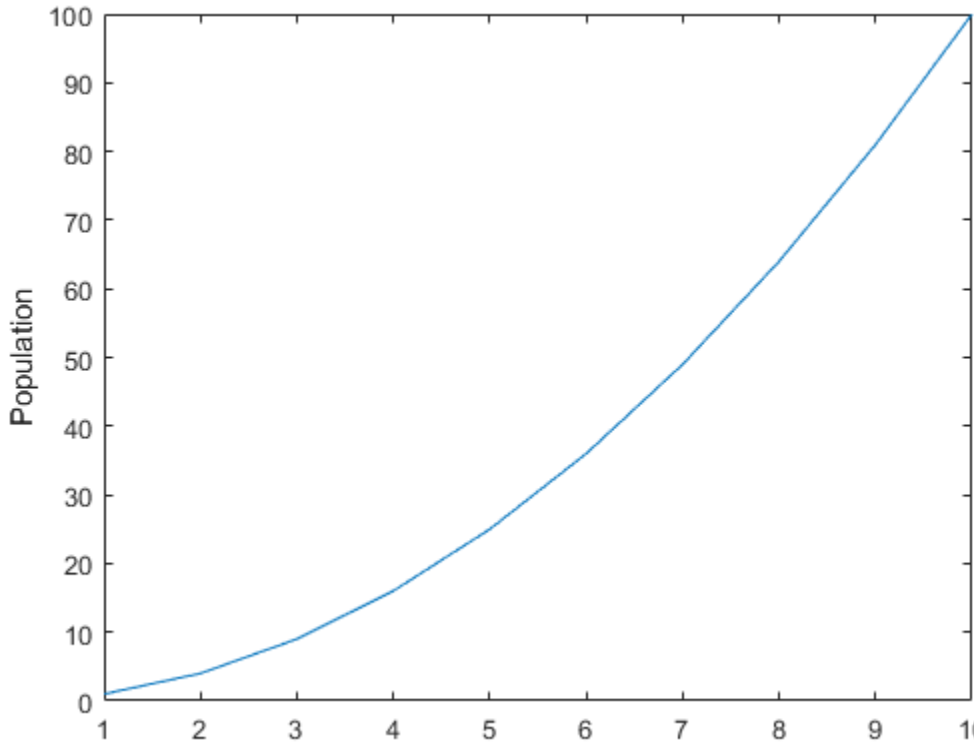
```
tiledlayout(2,1)
ax1 = nexttile;
plot((1:10).^2)
ylabel(ax1,'Population')

ax2 = nexttile;
plot((1:10).^3)
```



Label y-Axis and Return Object Handle

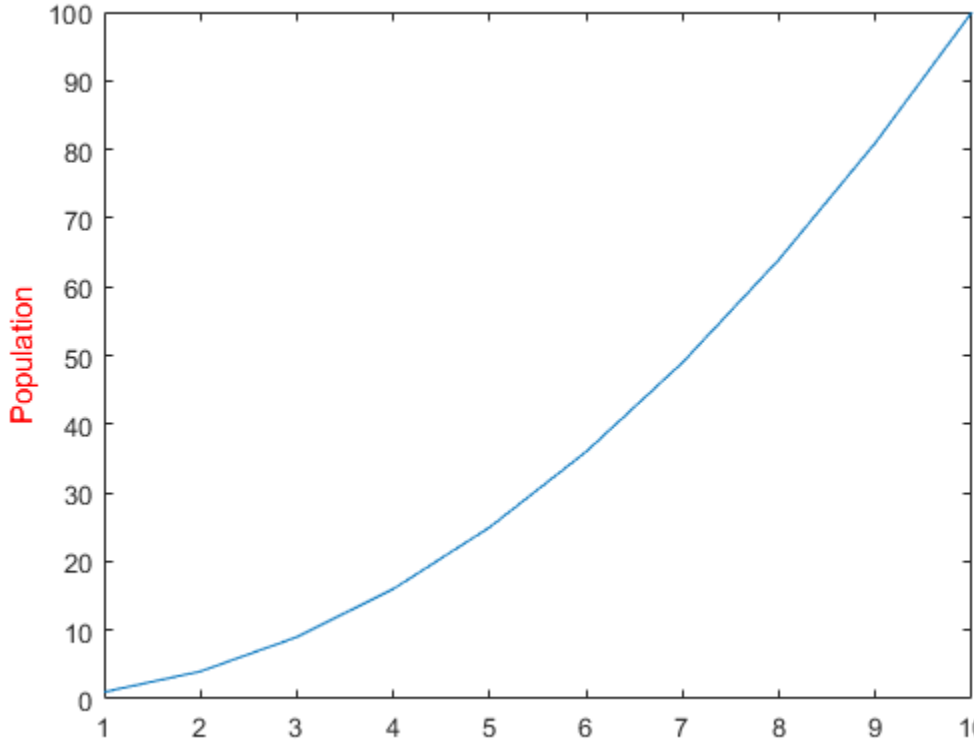
Label the *y*-axis and return the handle to the text object used as the label.

```
plot((1:10).^2)
t = ylabel('Population');
```



Set the color of the label to red. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead.

```
t.Color = 'red';
```



Input Arguments

`txt` — Axis label

string scalar | character vector | string array | character array | cell array | categorical array | numeric value

Axis label, specified as a string scalar, character vector, string array, character array, cell array, categorical array, or numeric value.

Example: `'my label'`

Example: `{'first line','second line'}`

Example: 123

To include numeric variables with text in a label, use the `num2str` function. For example:

```
x = 42;
txt = ['The value is ',num2str(x)];
```

To include special characters, such as superscripts, subscripts, Greek letters, or mathematical symbols use TeX markup. For a list of supported markup, see the `Interpreter` property.

To create multiline labels:

- Use a string array, where each element contains a line of text, such as `["first line","second line"]`.
- Use a cell array, where each cell contains a line of text, such as `{'first line','second line'}`.
- Use a character array, where each row contains the same number of characters, such as `['abc'; 'ab ']`.
- Use `sprintf` to create text with a new line character, such as `sprintf('first line \n second line')`.

Numeric labels are converted to text using `sprintf('%g',value)`. For example, `12345678` displays as `1.23457e+07`.

Note

- If you specify the label as a categorical array, MATLAB uses the values in the array, not the categories.
- The words `default`, `factory`, and `remove` are reserved words that will not appear in a label when quoted as a normal characters. To display any of these words individually, precede them with a backslash, such as `'\default'` or `'\remove'`.

`target` — Target for label

`Axes` object | `TiledChartLayout` object | graphics object

Target for label, specified as an `Axes` object, a `TiledChartLayout` object, or a graphics object that has a `YLabel` property. For example, you can add a *y*-axis label to a `HeatmapChart` object.

If you do not specify the target, then the `ylabel` function adds the label to the graphics object returned by the `gca` command.

Note Some charts do not support modifying the label appearance, such as the color, or returning the text object as an output argument.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Color','red','FontSize',12` specifies red, 12-point font.

In addition to the following, you can specify other text object properties using `Name,Value` pair arguments. See Text.

`FontSize` — Font size
`11` (default) | scalar value greater than `0`

Font size, specified as a scalar value greater than `0` in point units. One point equals 1/72 inch. To change the font units, use the `FontUnits` property.

Setting the font size properties for the associated axes also affects the label font size. The label font size updates to equal the axes font size times the label scale factor. The `FontSize` property of the axes contains the axes font size. The `LabelFontSizeMultiplier` property of the axes contains the label scale factor. By default, the axes font size is 10 points and the scale factor is 1.1, so the *y*-axis label font size is 11 points.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`FontWeight` — Character thickness
`'normal'` (default) | `'bold'`

Character thickness, specified as `'normal'` or `'bold'`.

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.

`FontName` — Font name
supported font name | `'FixedWidth'`

Font name, specified as a supported font name or `'FixedWidth'`. To display and print text properly, you must choose a font that your system supports. The default font depends on your operating system and locale.

To use a fixed-width font that looks good in any locale, use `'FixedWidth'`. The fixed-width font relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`Color` — Text color
`[0.15 0.15 0.15]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Text color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: 'blue'

Example: [0 0 1]

Example: '#0000FF'

Interpreter — Interpretation of text characters
'tex' (default) | 'latex' | 'none'

Interpretation of text characters, specified as one of these values:

- 'tex' — Interpret characters using a subset of TeX markup.
- 'latex' — Interpret characters using LaTeX markup.
- 'none' — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text.

Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they modify only the next character or the characters within the curly braces. When you set the `Interpreter` property to `'tex'`, the supported modifiers are as follows.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{`*`specifier`*`}`	Font name — Replace *`specifier`* with the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{`*`specifier`*`}`	Font size —Replace *`specifier`* with a numeric scalar value in point units.	`'\fontsize{15} text'`
`\color{`*`specifier`*`}`	Font color — Replace *`specifier`* with one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`

Modifier	Description	Example
`\color[rgb]{specifier}`	Custom font color — Replace *specifier* with a three-element RGB triplet.	`'\color[rgb]{0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	β	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ϵ	`\Delta`	Δ	`\spadesuit`	♠
`\zeta`	ζ	`\Theta`	Θ	`\leftrightarrow`	↔
`\eta`	η	`\Lambda`	Λ	`\leftarrow`	←
`\theta`	θ	`\Xi`	Ξ	`\Leftarrow`	⇐
`\vartheta`	ϑ	`\Pi`	Π	`\uparrow`	↑
`\iota`	ι	`\Sigma`	Σ	`\rightarrow`	→
`\kappa`	κ	`\Upsilon`	ϒ	`\Rightarrow`	⇒
`\lambda`	λ	`\Phi`	Φ	`\downarrow`	↓
`\mu`	µ	`\Psi`	Ψ	`\circ`	º
`\nu`	ν	`\Omega`	Ω	`\pm`	±
`\xi`	ξ	`\forall`	∀	`\geq`	≥
`\pi`	π	`\exists`	∃	`\propto`	∝

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. Use dollar symbols around the text, for example, use `'$\int_1^{20} x^2 dx$'` for inline mode or `'$$ \int_1^{20} x^2 dx$$'` for display mode.

The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup.

The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters. For multiline text, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project website at https://www.latex-project.org/.

Output Arguments

`t` — Text object
text object

Text object used as the *y*-axis label. Use `t` to access and modify properties of the label after its created.

See Also

Functions
`num2str` | `text` | `title` | `xlabel` | `zlabel`

Properties
Text

Topics
"Add Title and Axis Labels to Chart"

Introduced before R2006a

yline

Horizontal line with constant y-value

Syntax

```
yline(yvalue)
yline(yvalue,LineSpec)
yline(yvalue,LineSpec,label)
yline( ___ ,Name,Value)
yline(ax, ___ )
yl = yline( ___ )
```

Description

`yline(yvalue)` creates a constant horizontal line at the specified *y*-value in the current axes. For example, `yline(6)` creates a line at `y = 6`.

`yline(yvalue,LineSpec)` specifies either the line style, the line color, or both. For example, `':'` creates a dotted line, `'m'` creates a magenta line, and `':m'` creates a magenta dotted line.

`yline(yvalue,LineSpec,label)` adds the specified label to the line.

`yline( ___ ,Name,Value)` specifies constant line properties using one or more name-value pair arguments. For a list of properties, see ConstantLine Properties. Specify name-value pairs after all other input arguments.

`yline(ax, ___ )` creates the line in the specified axes instead of the current axes. To determine the current axes use `gca`.

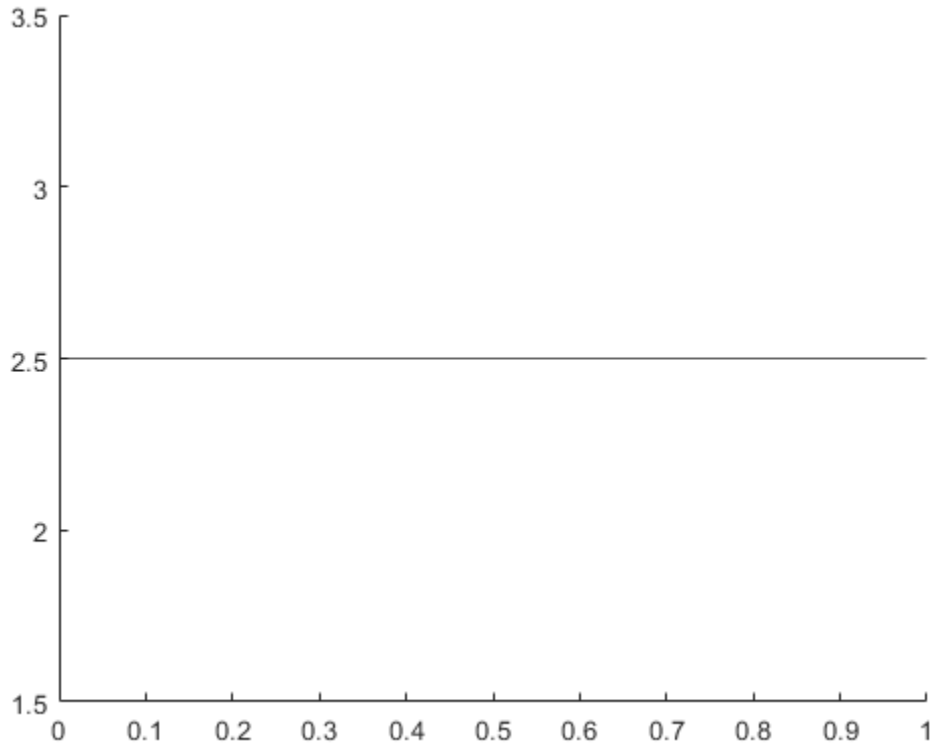
`yl = yline( ___ )` returns the `ConstantLine` object created. Use `yl` to modify or query properties of the line after it is created.

Examples

Create Horizontal Line

Create a horizontal line at y = 2.5.

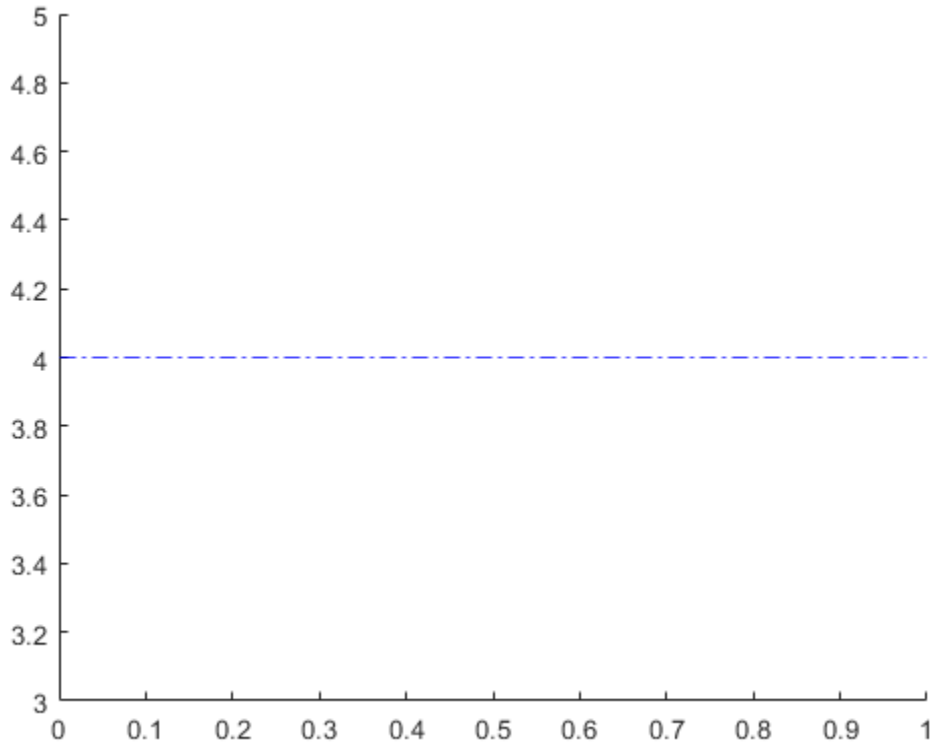
```
yline(2.5);
```



Specify Horizontal Line Style and Color

Create a blue dash-dot line at y = 4.

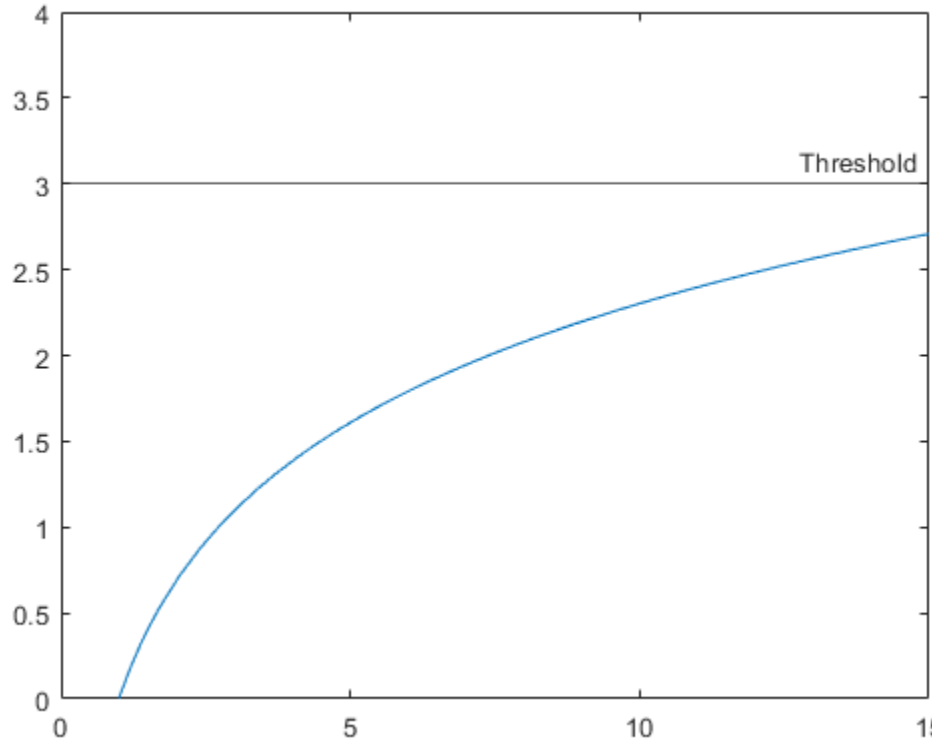
```
yline(4,'-.b');
```



Create Horizontal Line with Label

To create a line with a label, you must also specify the line style. The default line style is a solid line, `'-'`.

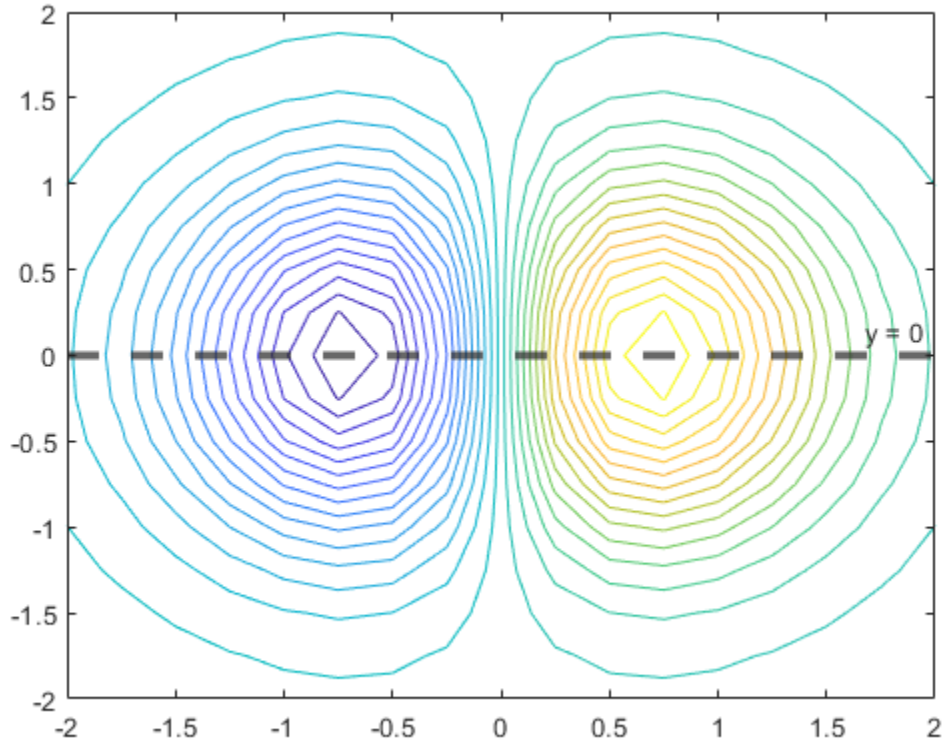
```
fplot(@(x) log(x));
xlim([0 15]);
ylim([0 4]);
yline(3,'-','Threshold');
```



Specify Horizontal Line Properties

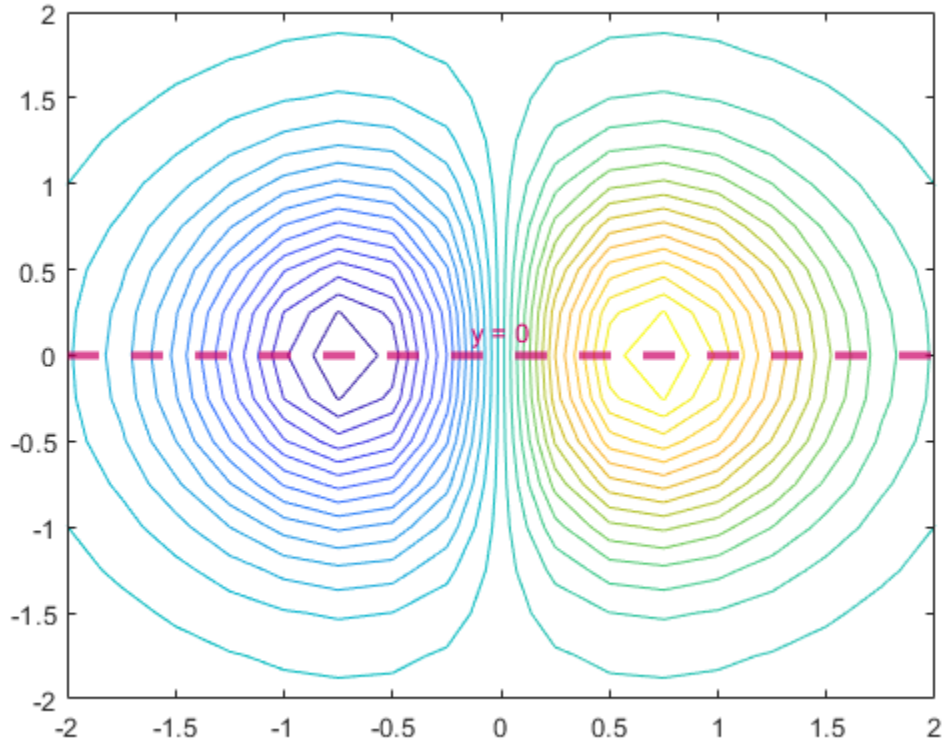
Create a horizontal dashed line with a label and a specified line width.

```
x = -2:0.25:2;
[X,Y] = meshgrid(x);
Z = X.*exp(-X.^2-Y.^2);
contour(X,Y,Z,30)

yl = yline(0,'--','y = 0','LineWidth',3);
```



Adjust the horizontal alignment and color of the line.

```
yl.LabelHorizontalAlignment = 'center';
yl.Color = [.80 0 .40];
```

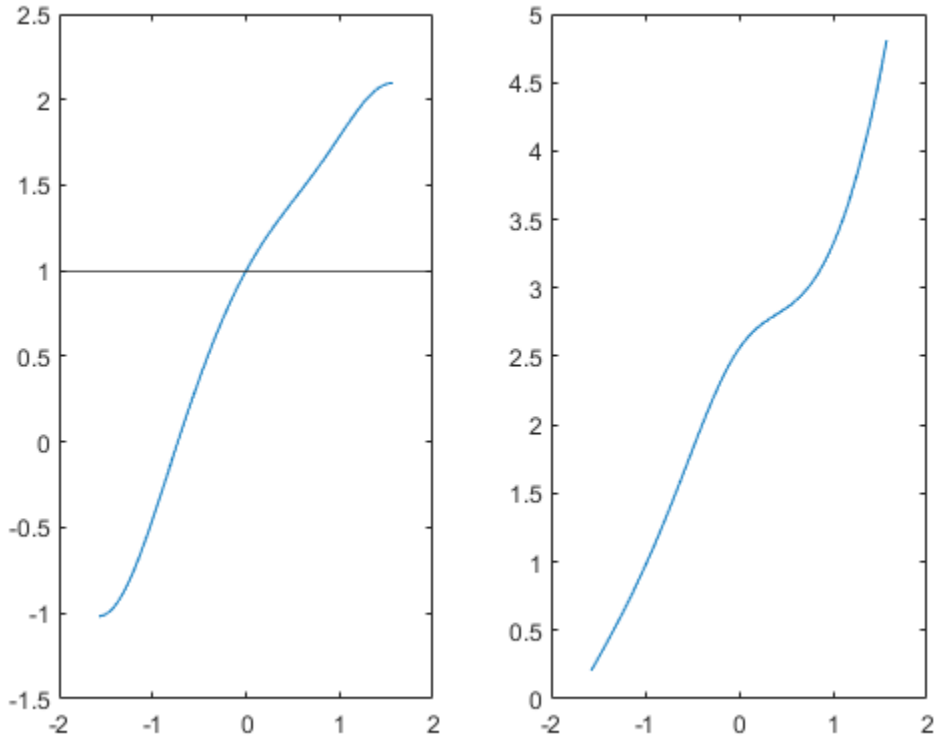


Plot Horizontal Line in Specific Axes

Create subplots in a 1-by-2 grid and plot data in each. Then, add a horizontal line to the first subplot.

```
ax1 = subplot(1,2,1);
x = -pi/2:pi/60:pi/2;
y1 = tan(sin(x)) + cos(sin(x));
plot(ax1,x,y1)

ax2 = subplot(1,2,2);
```

```
x = -pi/2:pi/60:pi/2;
y2 = tan(cos(x)) + exp(x);
plot(ax2,x,y2)

yline(ax1,1.0);
```



Input Arguments

yvalue — Location of horizontal line
scalar

Location of the horizontal line on the *y*-axis, specified as a scalar. You can specify `yvalue` as a numeric, categorical, `datetime`, or duration value. However, the type of value you specify must match the data type for the values along the *y*-axis.

Example: `yline(8.45)` plots the line at y = 8.45.

Example: `yline(C(2))` plots the line at the value of the second element in the array.

Example: `yline(datetime('today'))` plots the line at the current date and time.

`LineSpec` — Line style and color
character vector (default) | string scalar

Line style and color, specified as a character vector or string scalar containing symbols. The characters and symbols can appear in any order. You can specify line style, line color, or both. You cannot specify a marker symbol.

Example: `'--'` is a dashed line.

Example: `'--g'` is a green dashed line.

Line Style	Description
`'-'`	Solid line (default)
`'--'`	Dashed line
`':'`	Dotted line
`'-.'`	Dash-dot line

Color	Description
`'y'`	Yellow
`'m'`	Magenta
`'c'`	Cyan
`'r'`	Red
`'g'`	Green
`'b'`	Blue
`'w'`	White
`'k'`	Black

ax — Target axes
current axes (default) | `Axes` object

Target axes, specified as an `Axes` object. Use this argument if you want `yline` to plot in axes other than the current axes.

`label` — Line label

`''` (default) | character vector | cell array of character vectors | string array | numeric array

Line label, specified as a character vector, cell array of character vectors, string array, or numeric array. To create a multiline label, use a string array or a cell array of character vectors.

Example: `yline(55,'-.','Target Sales')`

Example: `yline(75,'-',["Average","Student Score"])`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `yline(3,':','Median Price','LabelHorizontalAlignment','center')` specifies the horizontal alignment of the label as `'center'`.

Note The properties listed here are only a subset. For a complete list, see ConstantLine Properties.

`Color` — Line color

`[0.15 0.15 0.15]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Line color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'g'`

Example: `[0.6 0.2 0.5]`

Example: `'#D2F9A7'`

`LineWidth` — Line width
`0.5` (default) | positive value

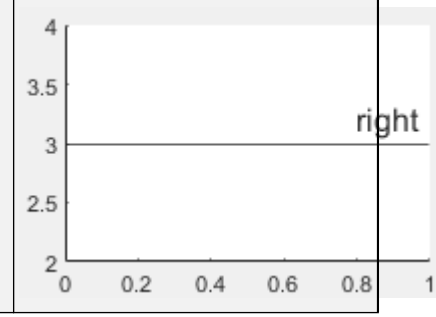
Line width, specified as a positive value in points.

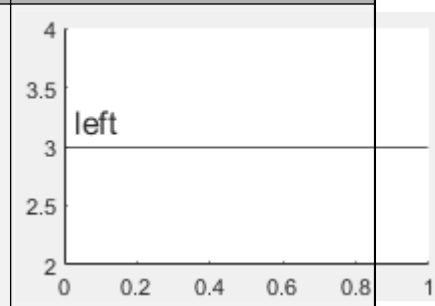
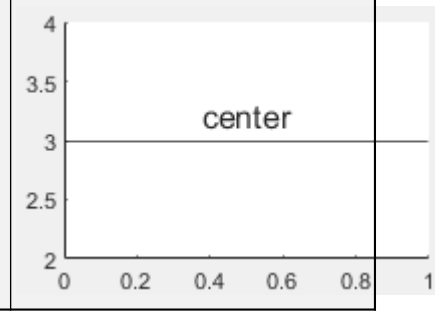
`DisplayName` — Legend label
`''` (default) | character vector | string scalar

Legend label, specified as a character vector or string scalar. The legend does not display until you call the `legend` command. If you do not specify the text, then `legend` sets the label using the form `'dataN'`.

`LabelHorizontalAlignment` — Horizontal alignment of label
`'right'` (default) | `'left'` | `'center'`

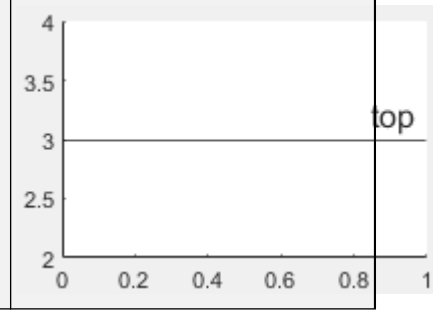
Horizontal alignment of the label with respect to the line, specified as one of the options in the table.

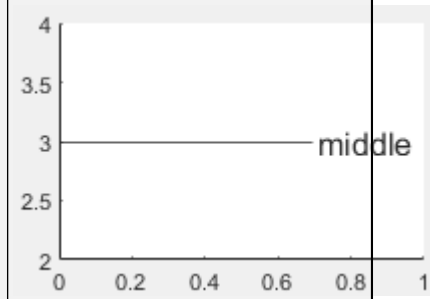
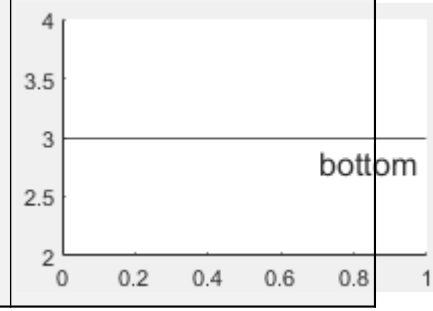
Option	Description	Example
`'right'`	Right side of the line	

Option	Description	Example
`'left'`	Left side of the line	
`'center'`	Center of the line	

`LabelVerticalAlignment` — Vertical alignment of label
`'top'` (default) | `'middle'` | `'bottom'`

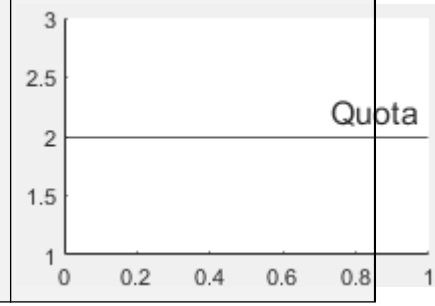
Vertical alignment of the label with respect to the line, specified as one of the options in the table.

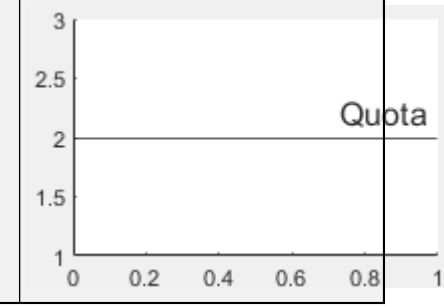
Option	Description	Example
`'top'`	Top of the line.	

Option	Description	Example
`'middle'`	Middle of the line. The label segments the line.	
`'bottom'`	Bottom of the line.	

`LabelOrientation` — Label orientation
`'aligned'` (default) | `'horizontal'`

Label orientation, specified as `'aligned'` or `'horizontal'`. Examples are shown in the table.

Orientation	Description	Example
`'aligned'`	Label has the same orientation as the specified line.	

Orientation	Description	Example
`'horizontal'`	Label is horizontal, regardless of the line orientation.	

Algorithms

In a 3-D view of the axes, the constant line appears in the *x*-*y* plane at the midpoint of the *z*-axis limits. If the axes are rotated, then the constant line rotates accordingly.

See Also

Functions
`plot` | `xline`

Properties
ConstantLine Properties

Introduced in R2018b

ymd

Year, month, and day numbers of datetime

Syntax

```
[y,m,d] = ymd(t)
```

Description

`[y,m,d] = ymd(t)` returns the year, month, and day numbers of the datetime values in `t` as separate numeric arrays. The `y`, `m`, and `d` outputs are the same size as `t`, and contain integer values.

The `ymd` function is equivalent to calling the `year`, `month`, and `day` functions on the same datetime array.

Examples

Find Year, Month, and Day Numbers of Dates

```
t = datetime(2013,05,31):calmonths(3):datetime(2014,06,15)
```

```
t = 1x5 datetime array
   31-May-2013   31-Aug-2013   30-Nov-2013   28-Feb-2014   31-May-2014
```

```
[y,m,d] = ymd(t)
```

```
y = 1×5

        2013        2013        2013        2014        2014


m = 1×5
```

```
     5     8    11     2     5

d = 1×5

    31    31    30    28    31
```

`ymd` returns the year, month, and day values in separate arrays.

Input Arguments

`t` — Input date and time
`datetime` array

Input date and time, specified as a `datetime` array.

Output Arguments

`y` — ISO year numbers
scalar | vector | matrix | multidimensional array

ISO year numbers, returned as a scalar, vector, matrix, or multidimensional array of integer values. ISO year numbers include a year zero and represent years BCE using negative values. `y` is of type `double` and is the same size as `t`.

`m` — Month numbers
scalar | vector | matrix | multidimensional array

Month numbers, returned as a scalar, vector, matrix, or multidimensional array of integer values from 1 to 12. `m` is of type `double` and is the same size as `t`.

`d` — Day of month numbers
scalar | vector | matrix | multidimensional array

Day of month numbers, returned as a scalar, vector, matrix, or multidimensional array of integer values from 1 to 28, 29, 30, or 31, depending on the month and year. `d` is of type `double` and is the same size as `t`.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`day` | `hms` | `month` | `quarter` | `week` | `year`

Introduced in R2014b

ytickangle

Rotate *y*-axis tick labels

Syntax

```
ytickangle(angle)
ytickangle(ax,angle)
```

```
ang = ytickangle
ang = ytickangle(ax)
```

Description

`ytickangle(angle)` rotates the *y*-axis tick labels for the current axes to the specified angle in degrees, where `0` is horizontal. Specify a positive value for counterclockwise rotation or a negative value for clockwise rotation.

`ytickangle(ax,angle)` rotates the tick labels for the axes specified by `ax` instead of the current axes.

`ang = ytickangle` returns the rotation angle for the *y*-axis tick labels of the current axes as a scalar value in degrees. Positive values indicate counterclockwise rotation. Negative values indicate clockwise rotation.

`ang = ytickangle(ax)` uses the axes specified by `ax` instead of the current axes.

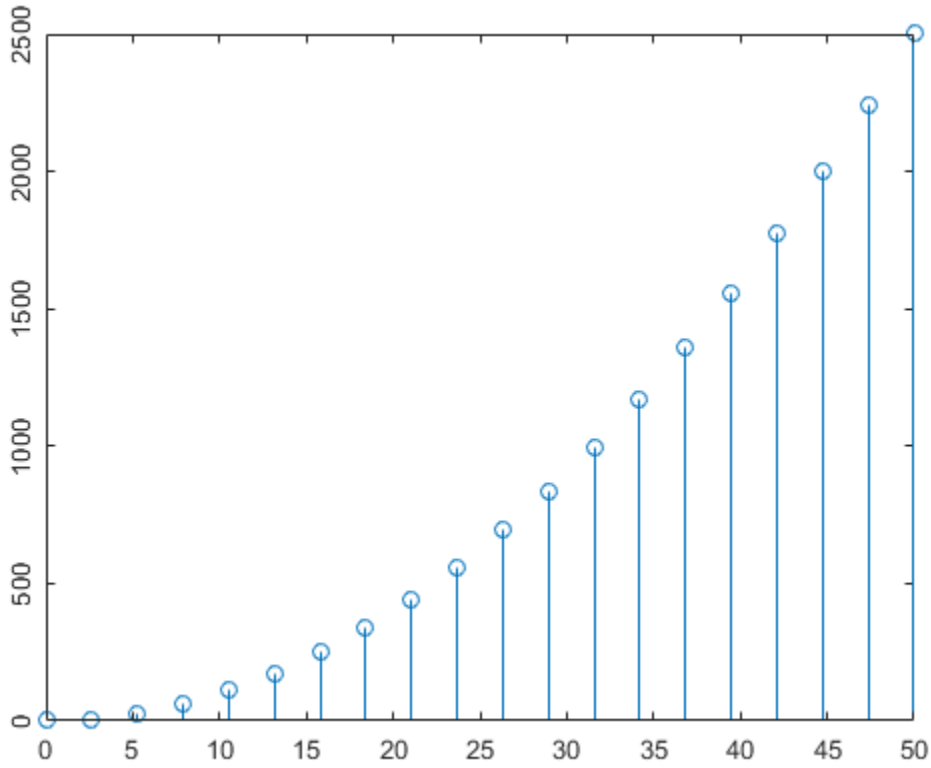
Examples

Rotate y-Axis Tick Labels

Create a stem chart and rotate the *y*-axis tick labels 90 degrees so that they appear vertically.

```
x = linspace(0,50,20);
y = x.^2;
```

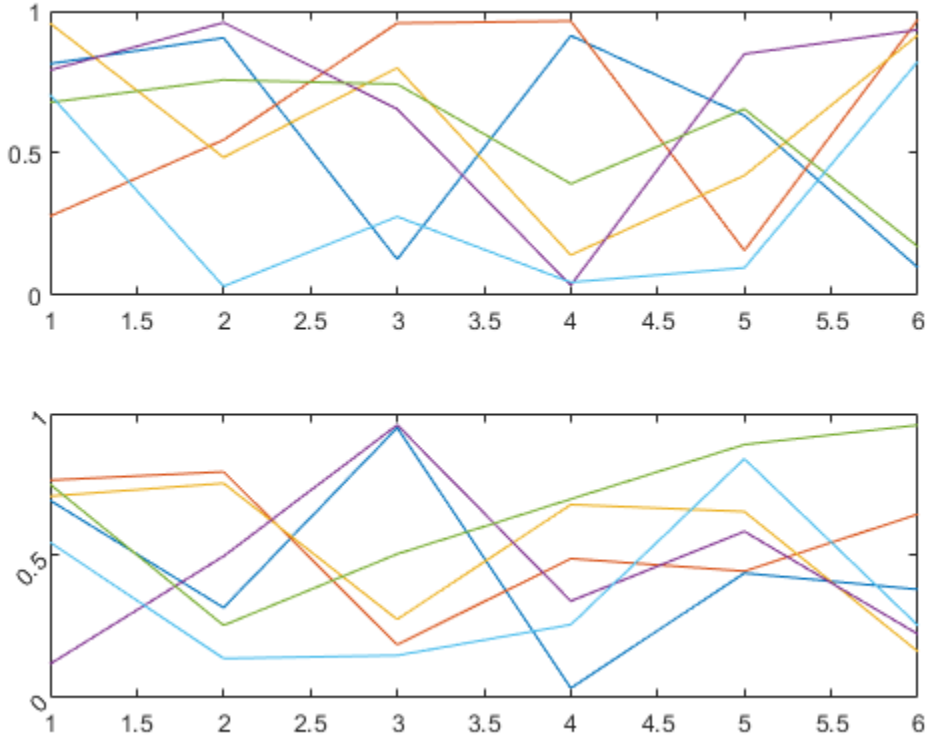
```
stem(x,y)
ytickangle(90)
```



Rotate y-Axis Tick Labels for Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Plot into each of the axes. Then rotate the *y*-axis tick labels for the lower plot by specifying `ax2` as the first input argument to `ytickangle`.

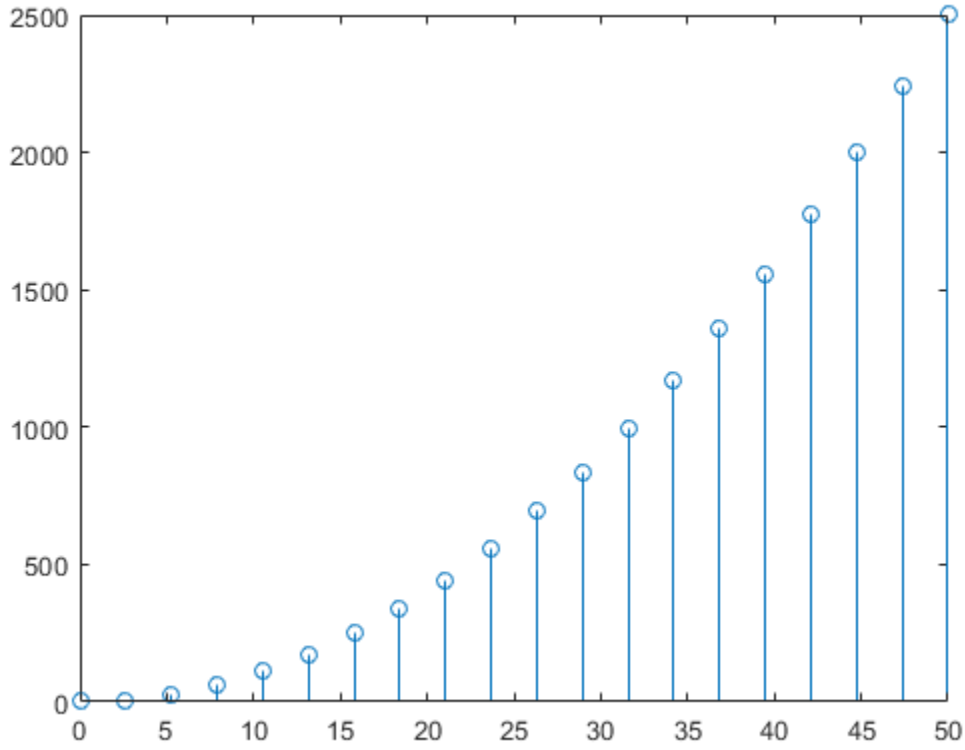
```
tiledlayout(2,1)
ax1 = nexttile;
plot(ax1,rand(6))

ax2 = nexttile;
plot(ax2,rand(6))
ytickangle(ax2,45)
```



Query Current Angle for y-Axis Tick Labels

Create a stem chart. Then, query the rotation angle for the *y*-axis tick labels. By default, the labels are not rotated.

```
x = linspace(0,50,20);
y = x.^2;
stem(x,y)
```




```
ang = ytickangle
```

```
ang = 0
```

Input Arguments

angle — Rotation of tick labels
`0` (default) | scalar value in degrees

Rotation of tick labels, specified as a scalar value in degrees, where 0 is horizontal.

Example: `ytickangle(90)`

ax — Target axes
current axes (default) | single `Axes` object | vector of `Axes` objects

Target axes, specified as a single `Axes` object or a vector of `Axes` objects. If you do not specify the axes, then `ytickangle` uses the current axes.

Algorithms

The `ytickangle` function sets and queries the `YTickLabelRotation` property of the `Axes` object.

See Also

Functions
`xtickangle` | `ylim` | `ytickformat` | `yticklabels` | `yticks` | `ztickangle`

Properties
Axes

Introduced in R2016b

ytickformat

Specify *y*-axis tick label format

Syntax

```
ytickformat(fmt)
ytickformat(datefmt)
ytickformat(durationfmt)
ytickformat(ax, ___ )

yfmt = ytickformat
yfmt = ytickformat(ax)
```

Description

`ytickformat(fmt)` sets the format for numeric *y*-axis tick labels. For example, specify `fmt` as `'usd'` to display the labels in U.S. dollars.

`ytickformat(datefmt)` sets the format for labels that show dates or times. For example, specify `datefmt` as `'MM-dd-yy'` to display dates such as `04-19-16`. This option applies only to a *y*-axis with datetime values.

`ytickformat(durationfmt)` sets the format for labels that show durations. For example, specify `durationfmt` as `'m'` to display the durations in minutes. This option applies only to a *y*-axis with duration values.

`ytickformat(ax, ___ )` uses the axes specified by `ax` instead of the current axes. Specify `ax` as the first input argument.

`yfmt = ytickformat` returns the format style used for *y*-axis tick labels of the current axes. Depending on the type of labels along the *y*-axis, `yfmt` is a character vector of a numeric format, date format, or duration format.

`yfmt = ytickformat(ax)` returns the format style used for the axes specified by `ax` instead of the current axes.

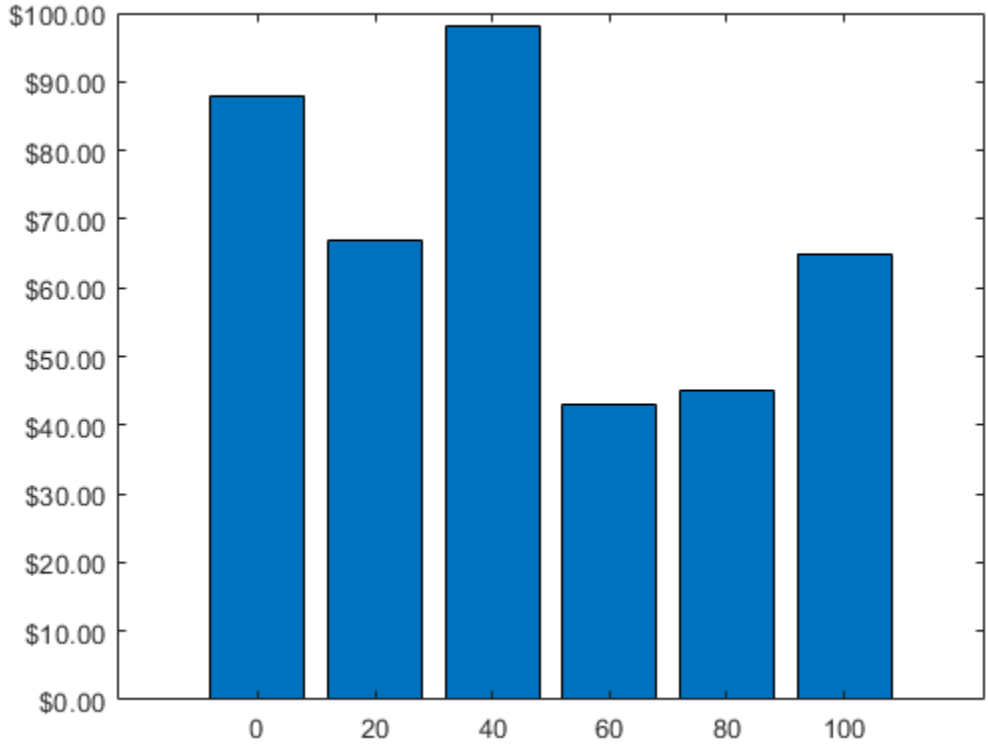
Examples

Display Tick Labels in Dollars

Display the *y*-axis tick labels in dollars with and without decimal values.

Create a bar chart. Display the tick labels along the *y*-axis in dollars.

```
x = 0:20:100;
y = [88 67 98 43 45 65];
bar(x,y)
ytickformat('usd')
```



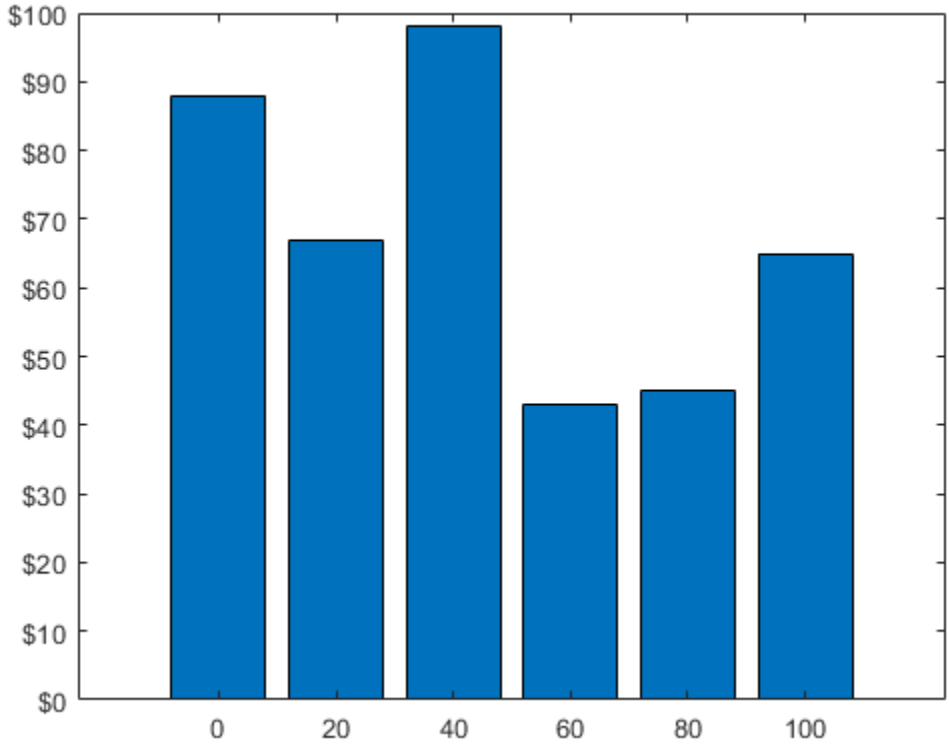
Query the tick label format. MATLAB® returns the format as a character vector containing the formatting operators.

```
fmt = ytickformat
```

```
fmt =
'$%,.2f'
```

Display the tick labels with no decimal values by tweaking the numeric format to use a precision value of 0 instead of 2.

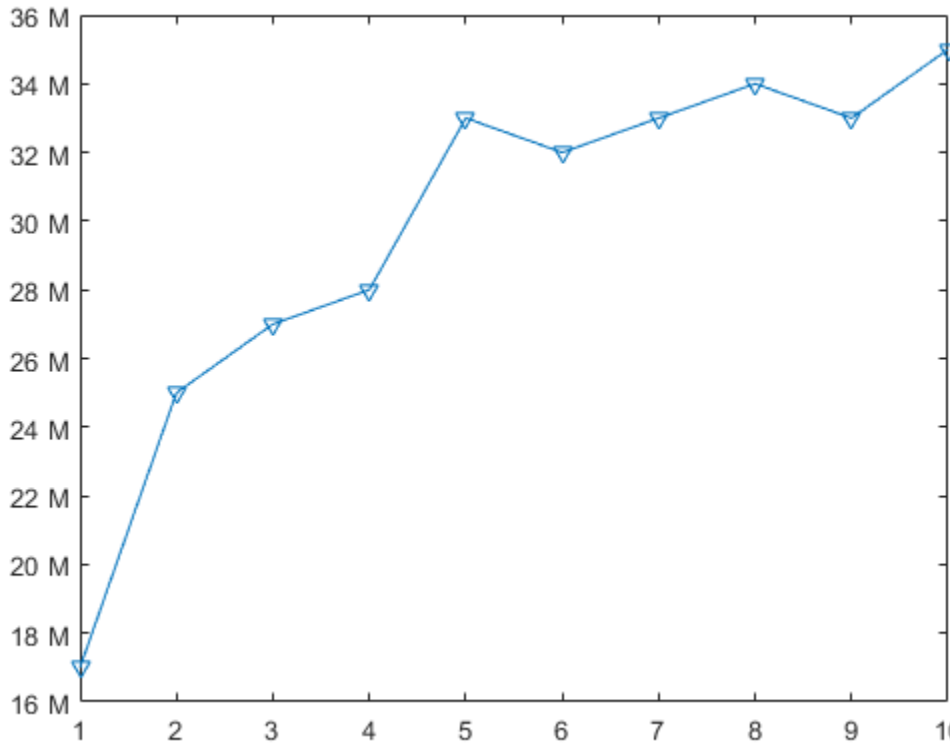
```
ytickformat('$%,.0f')
```



Display Text After Each Tick Label

Create a line chart. Display the tick labels along the *y*-axis with the text "M" after each value.

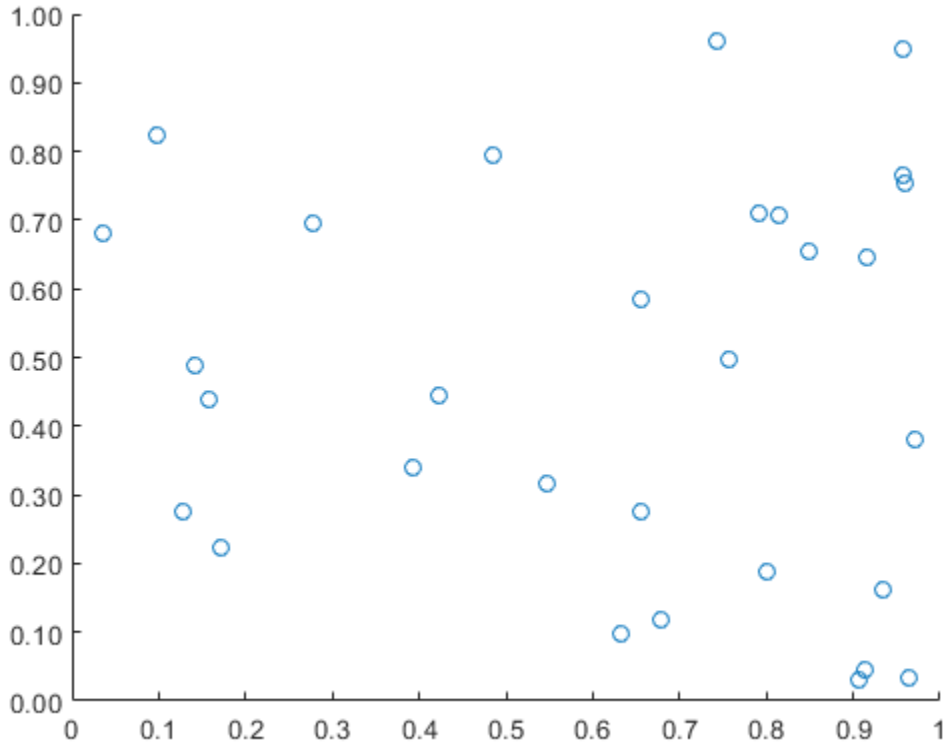
```
x = 1:10;
y = [17 25 27 28 33 32 33 34 33 35];
plot(x,y,'-V')
ytickformat('%g M')
```



Control Number of Decimals Displayed

Create a scatter plot of random data. Display the *y*-axis tick labels with two decimal places. Control the decimal places by passing `ytickformat` a character vector of a numeric format that uses fixed-point notation for the conversion character and a precision value of 2.

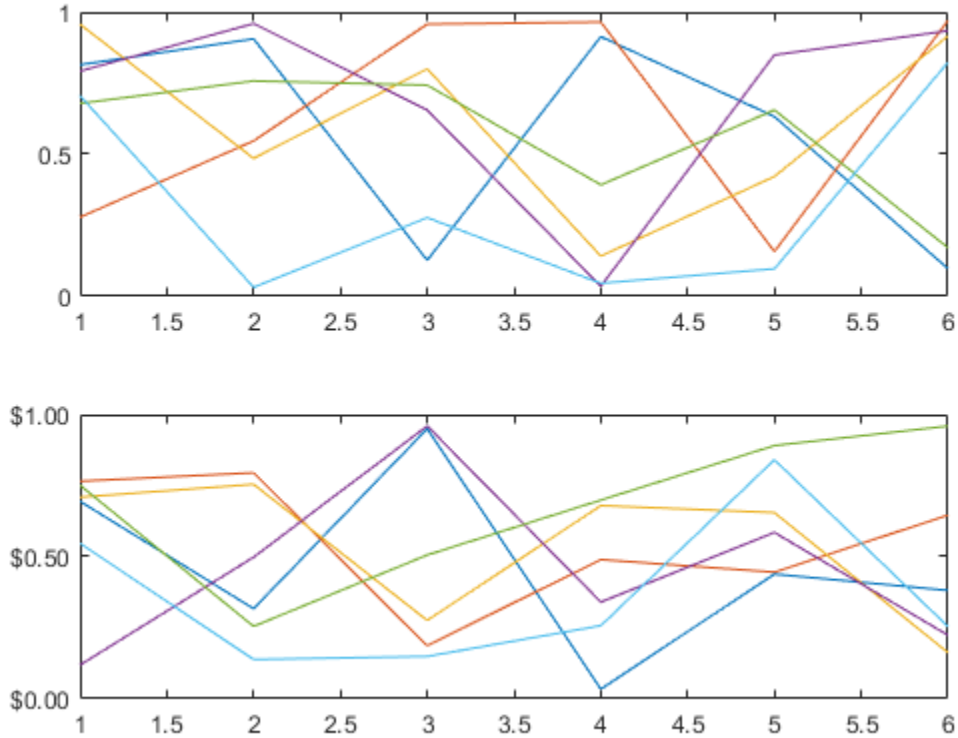
```
x = rand(30,1);
y = rand(30,1);
scatter(x,y);
ytickformat('%.2f')
```



Tick Label Format for Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Plot into each of the axes. Specify the tick label format for the *y*-axis of the lower plot by specifying `ax2` as the first input argument to `ytickformat`. Display the tick labels in U.S. dollars.

```
tiledlayout(2,1)
ax1 = nexttile;
plot(ax1,rand(6))

ax2 = nexttile;
plot(ax2,rand(6))
ytickformat(ax2,'usd')
```



Input Arguments

`fmt` — Format for numeric tick labels
`'%g'` (default) | character vector | string scalar

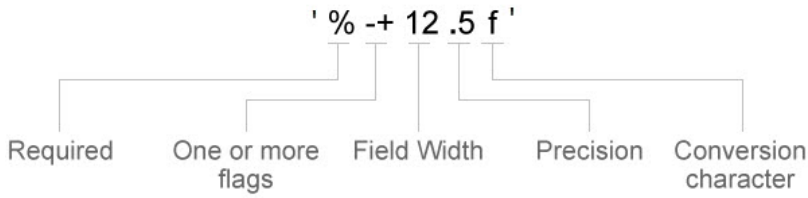
Format for numeric tick labels, specified as a character vector or string scalar. You can specify one of the formats listed in this table. Alternatively, you can specify a custom format.

Predefined Format	Description
`'usd'`	U.S. dollars. This option is equivalent using `'$%,.2f'`. If the labels use scientific notation, this option sets the exponent value to 0.
`'eur'`	Euro. This option is equivalent to using `'\x20AC%,.2f'` with an exponent value of 0.
`'gbp'`	British pound. This option is equivalent to using `'\x00A3%,.2f'` with an exponent value of 0.
`'jpy'`	Japanese yen. This option is equivalent to using `'\x00A5%,d'` with an exponent value of 0.
`'degrees'`	Display degree symbol after values. This option is equivalent to using `'%g\x00B0'` with the default exponent value.
`'percentage'`	Display percent sign after values. This option is equivalent to using `'%g%%'` with the default exponent value.
`'auto'`	Default format of `'%g'` with the default exponent value.

Example: `ytickformat('usd')`

Custom Numeric Format

You can specify a custom numeric format by creating a character vector or string containing identifiers.



Identifiers are optional, except the percent sign and conversion character. Construct the format in this order:

- One or more flags — Options such as adding a plus sign before positive values. For a full list of options, see the table of Optional Flags.
- Field width — Minimum number of characters to print in the tick label. Specify the field width as an integer value. If the number of significant digits in the tick value is smaller than the field width, then the label is padded with spaces.
- Precision — Number of digits to the right of the decimal point or the number of significant digits, depending on the conversion character. Specify the precision as an integer value.
- Conversion character — Value type. For a full list of options, see the table of Conversion Characters. If you specify a conversion that does not fit the data, then MATLAB overrides the specified conversion, and uses `%e`.

Also, you can specify literal text at the beginning or end of the format. To print a single quotation mark, use `''`. To print a percent character, use `%%`.

Example: `ytickformat('%.2f')` displays the values using fixed-point notation with two decimal places.

Example: `ytickformat('$%.2f')` displays a dollar sign before each value.

Example: `ytickformat('%.2f million')` displays `million` after each value.

Optional Flags

Identifier	Description	Example of Numeric Format
`,`	Display commas every three digits, such as `'1,000'`.	`'%,4.4g'`
`+`	Print the sign character (`+`) for positive values, such as `'+100'`.	`'%+4.4g'`
`0`	Pad the field width with leading zeros instead of spaces, such as `'0100'`.	`'%04.4g'`
`-`	Left-justify, which pads the end of the value with spaces instead of the beginning. For example, if the field width is `4`, then this flag formats the label as `'100 '` instead of `' 100'`.	`'%-4.4g'`
`#`	For the `%f`, `%e`, and `%g` conversion characters, print the decimal point even when the precision is 0, such as `'100.'`. For `%g`, do not remove trailing zeros.	`'%#4.4g'`

Conversion Characters

Identifier	Description	Example
`d` or `i`	Signed integer with base 10. The precision value indicates the number of significant digits.	`'%.4d'` displays π as `0003`.
`f`	Fixed-point notation. The precision value indicates the number of decimal places.	`'%.4f'` displays π as `3.1416`.
`e`	Exponential notation. The precision value indicates the number of decimal places.	`'%.4e'` displays π as `3.1416x10`^{0}.
`g`	The more compact version of `e` or `f`, with no trailing zeros. The precision value indicates the maximum number of decimal places.	`'%.4g'` displays π as `3.1416`.

`datefmt` — Format for dates and times
`'auto'` | character vector | string scalar

Format for dates and times, specified as `'auto'`, a character vector, or a string scalar. The default format is based on the data.

Example: `ytickformat('yyyy-MM-dd')` displays a date and time such as `2016-04-19`.

Example: `ytickformat('eeee, MMMM d, yyyy HH:mm:ss')` displays a date and time such as `Saturday, April 19, 2016 21:41:06`.

Example: `ytickformat('MMMM d, yyyy HH:mm:ss Z')` displays a date and time such as `April 19, 2016 21:41:06 -0400`.

The display format tables show the letter identifiers that you can use to construct the format. To separate the fields, you can include nonletter characters such as a hyphen, space, colon, or any non-ASCII characters. The identifiers correspond to the Unicode Locale Data Markup Language (LDML) standard for dates.

Date and Time Formats

Use these identifiers to specify the display formats of the date and time fields.

Letter Identifier	Description	Display
G	Era	CE
y	Year, with no leading zeros. See the Note that follows this table.	2014
yy	Year, using last two digits. See the Note that follows this table.	14
yyy, yyyy ...	Year, using at least the number of digits specified by the number of instances of 'y'	For the year 2014, 'yyy' displays 2014, while 'yyyyy' displays 02014.
u, uu, ...	ISO year. A single number designating the year. An ISO year value assigns positive values to CE years and negative values to BCE years, with 1 BCE being year 0.	2014
Q	Quarter, using one digit	2
QQ	Quarter, using two digits	02
QQQ	Quarter, abbreviated	Q2
QQQQ	Quarter, full name	2nd quarter
M	Month, numerical using one or two digits	4
MM	Month, numerical using two digits	04
MMM	Month, abbreviated name	Apr
MMMM	Month, full name	April
MMMMM	Month, capitalized first letter	A
W	Week of the month	1
d	Day of the month, using one or two digits	5
dd	Day of the month using two digits	05
D	Day of the year, using one, two or three digits	95
DD	Day of the year using two digits	95

Letter Identifier	Description	Display
DDD	Day of the year using three digits	095
e	Day of the week, numerical using one or two digits.	7, where Sunday is the first day of the week.
ee	Day of the week, numerical using two digits	07
eee	Day, abbreviated name	Sat
eeee	Day, full name	Saturday
eeeee	Day, capitalized first letter	S
a	Day period (AM or PM)	PM
h	Hour, 12-hour clock notation using one or two digits	9
hh	Hour, 12-hour clock notation using two digits	09
H	Hour, 24-hour clock notation using one or two digits	21
HH	Hour, 24-hour clock notation using two digits	21
m	Minute, using one or two digits	41
mm	Minute, using two digits	41
s	Second, using one or two digits	6
ss	Second, using two digits	06
S, SS, ..., SSSSSSSSS	Fractional second, using the number of digits specified by the number of instances of 'S' (up to 9 digits).	'SSS' truncates 6.12345 seconds to 123.

Some tips and considerations:

- Use one or more u characters instead of y characters to represent the year when working with year numbers near zero.
- Datetime values later than 144683 years CE or before 140743 BCE display only the year numbers, regardless of the specified format value.

Time Zone Offset Formats

Use these identifiers to specify the display format of the time zone offset. A time zone offset is the amount of time that a specific date and time is offset from UTC. A time zone offset is different from a time zone in that it comprises rules that determine the offsets for specific times of the year. Include a time zone offset identifier when you want to ensure that the time components are displayed unambiguously.

Letter Identifier	Description	Display
`z`	Abbreviated name of the time zone offset. If this value is not available, then the time zone offset uses the short UTC format, such as `UTC-4`.	`EDT`
`Z`	ISO 8601 basic format with hours, minutes, and optional seconds fields.	`-0400`
`ZZZZ`	Long UTC format.	`UTC-04:00`
`ZZZZZ`	ISO 8601 extended format with hours, minutes, and optional seconds fields. A time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	`-04:00`
`x` or `X`	ISO 8601 basic format with hours field and optional minutes field. If you specify `X`, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	`-04`
`xx` or `XX`	ISO 8601 basic format with hours and minutes fields. If you specify `XX`, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	`-0400`

Letter Identifier	Description	Display
xxx or XXX	ISO 8601 extended format with hours and minutes fields. If you specify XXX, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-04:00
xxxx or XXXX	ISO 8601 basic format with hours, minutes, and optional seconds fields. If you specify XXXX, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-0400
xxxxx or XXXXX	ISO 8601 extended format with hours, minutes, and optional seconds fields. If you specify XXXXX, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-04:00

`durationfmt` — Format for duration values

character vector | string scalar

Format for duration values, specified as a character vector or string scalar.

To display a duration as a single number that includes a fractional part (for example, 1.234 hours), specify one of the following formats.

Format	Description
'y'	Number of exact, fixed-length years. A fixed-length year is equal to 365.2425 days.
'd'	Number of exact, fixed-length days. A fixed-length day is equal to 24 hours.
'h'	Number of hours
'm'	Number of minutes
's'	Number of seconds

To display a duration in the form of a digital timer, specify one of these formats:

- `'dd:hh:mm:ss'`
- `'hh:mm:ss'`
- `'mm:ss'`
- `'hh:mm'`

In addition, you can display up to nine fractional second digits by appending up to nine `S` characters. For example, `'hh:mm:ss.SSS'` displays the milliseconds of a duration value to three digits.

ax — Target axes
current axes (default) | single `Axes` object | vector of `Axes` objects

Target axes, specified as a single `Axes` object or a vector of `Axes` objects. If you do not specify the axes, then `ytickformat` uses the current axes.

Algorithms

The `ytickformat` function sets and queries the `TickLabelFormat` property of the ruler object associated with the *y*-axis.

See Also

Functions
`xtickformat` | `ylim` | `ytickangle` | `yticklabels` | `yticks` | `ztickformat`

Properties
DatetimeRuler | DurationRuler | NumericRuler

Introduced in R2016b

yticklabels

Set or query *y*-axis tick labels

Syntax

```
yticklabels(labels)
yl = yticklabels
```

```
yticklabels('auto')
yticklabels('manual')
m = yticklabels('mode')
```

```
___ = yticklabels(ax, ___ )
```

Description

`yticklabels(labels)` sets the *y*-axis tick labels on page 1-17238 for the current axes. Specify `labels` as a string array or a cell array of character vectors; for example, `{'January','February','March'}`. If you specify the labels, then the *y*-axis tick values and tick labels no longer update automatically based on changes to the axes.

`yl = yticklabels` returns the *y*-axis tick labels for the current axes.

`yticklabels('auto')` sets an automatic mode, enabling the axes to determine the *y*-axis tick labels. Use this option if you set the labels and then want to set them back to the default values.

`yticklabels('manual')` sets a manual mode, freezing the *y*-axis tick labels at the current values.

`m = yticklabels('mode')` returns the current value of the *y*-axis tick labels mode, which is either `'auto'` or `'manual'`. By default, the mode is automatic unless you specify the tick labels or set the mode to manual.

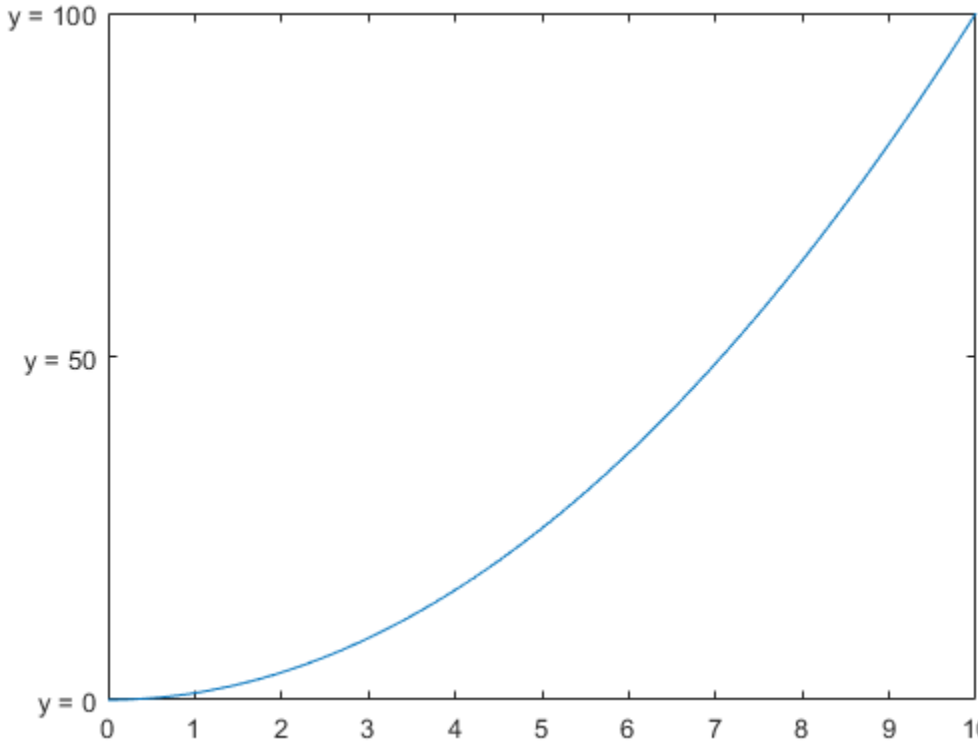
`___ = yticklabels(ax, ___ )` uses the axes specified by `ax` instead of the current axes. Specify `ax` as the first input argument for any of the previous syntaxes.

Examples

Specify y-Axis Tick Values and Labels

Create a line plot. Display tick marks along the *y*-axis at the values 0, 50, and 100. Then, specify a label for each tick mark.

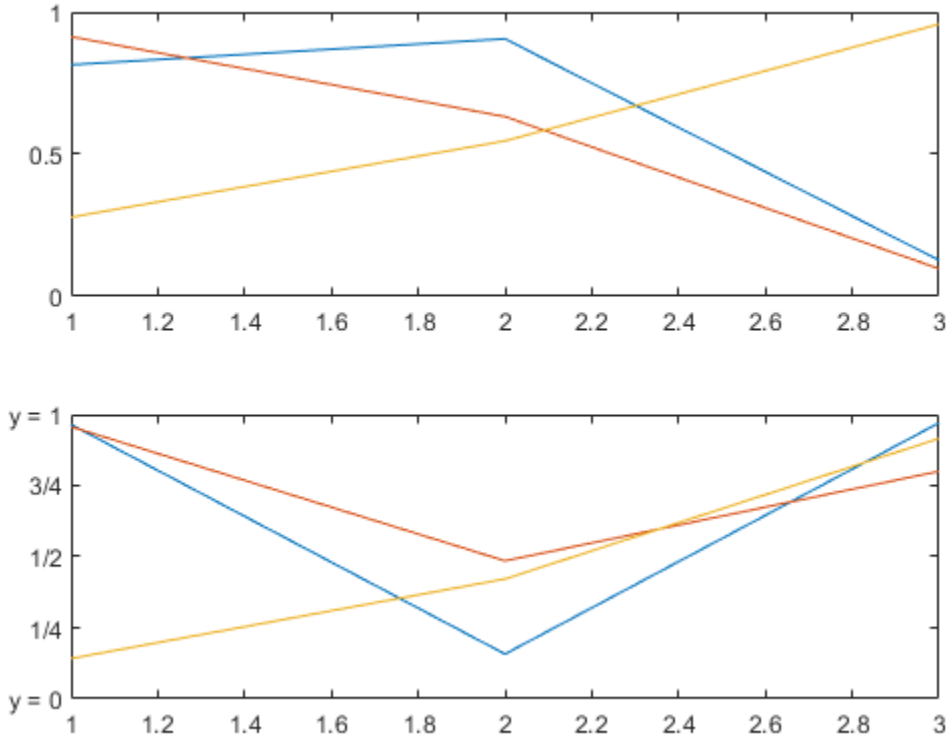
```
x = linspace(0,10);
y = x.^2;
plot(x,y)
yticks([0 50 100])
yticklabels({'y = 0','y = 50','y = 100'})
```



Specify y-Axis Tick Labels for Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Create two plots of random data. Set the *y*-axis tick values and labels for the second plot by passing `ax2` as the first input argument to the `yticks` and `yticklabels` functions.

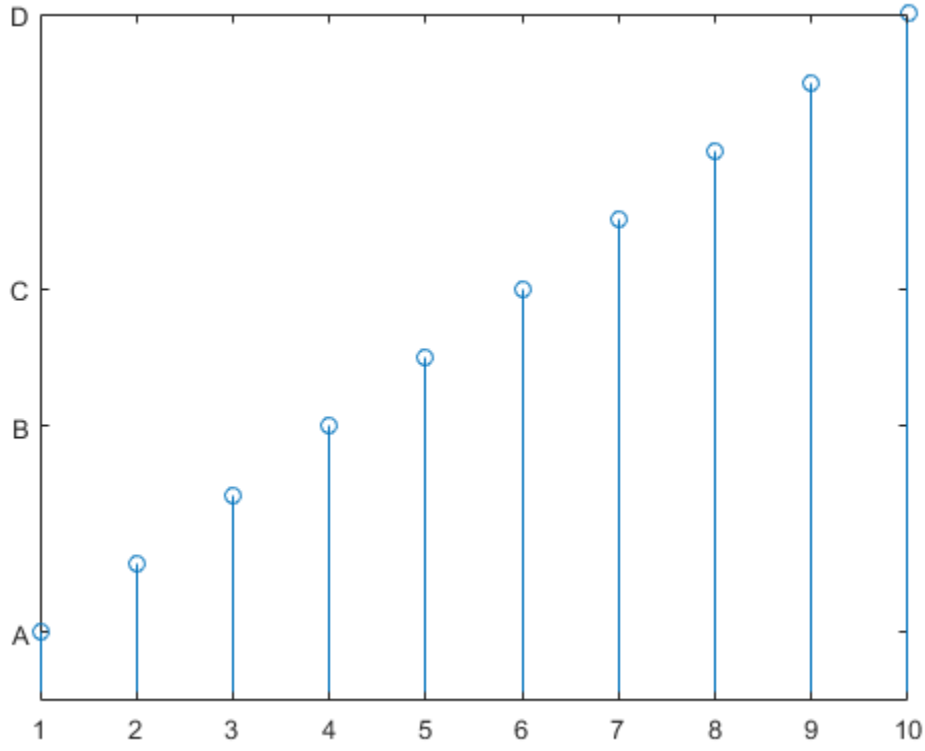
```
tiledlayout(2,1)
ax1 = nexttile;
plot(rand(3))

ax2 = nexttile;
plot(rand(3))
yticks(ax2,[0 .25 .5 .75 1])
yticklabels(ax2,{'y = 0','1/4','1/2','3/4','y = 1'})
```



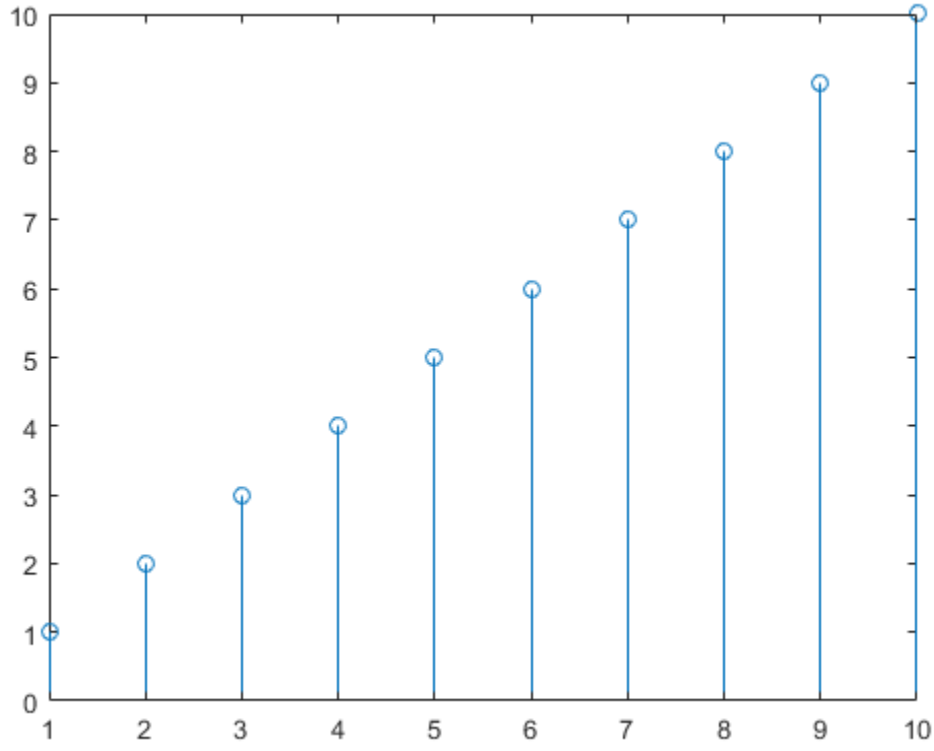
Set y-Axis Tick Labels Back to Default Labels

Create a stem chart and specify the *y*-axis tick values and corresponding labels. Then, set the *y*-axis tick values and labels back to the default values.

```
stem(1:10)
yticks([1 4 6 10])
yticklabels({'A','B','C','D'})
```



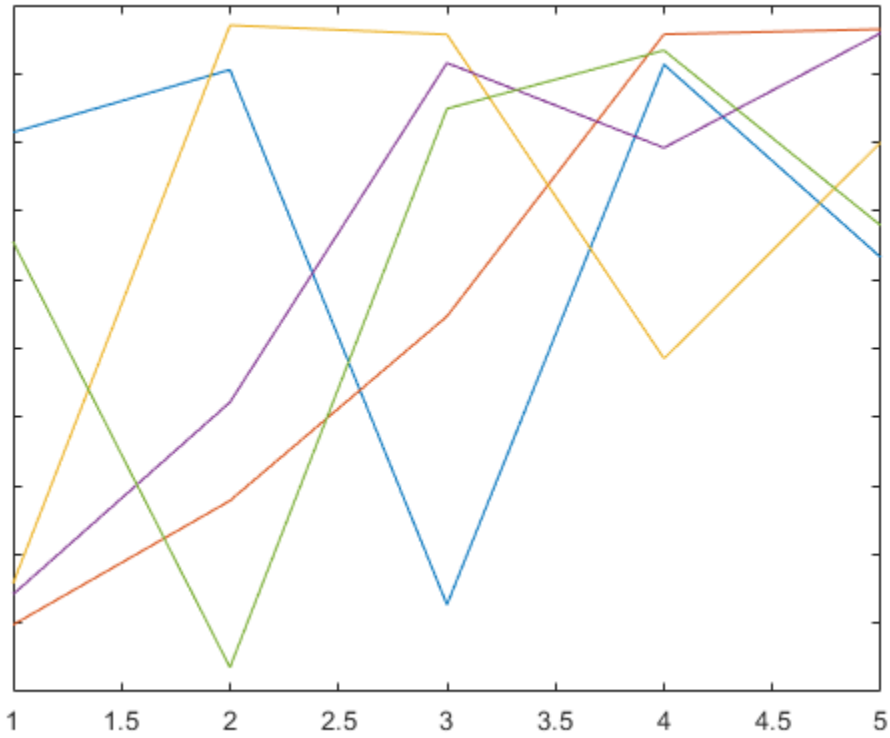

```
yticks('auto')
yticklabels('auto')
```



Remove *y*-Axis Tick Labels

Remove the tick labels along the *y*-axis by specifying the tick labels as an empty array.

```
plot(rand(5))
yticklabels({})
```



Input Arguments

`labels` — Tick labels

cell array of character vectors | string array | categorical array

Tick labels on page 1-17238, specified as a cell array of character vectors, string array, or categorical array. If you do not want tick labels to show, then specify an empty cell array `{}`. Tick labels support TeX and LaTeX markup. See the `TickLabelInterpreter` property of the `Axes` object for more information.

Example: `yticklabels({'0','\pi','2\pi'})`

Example: `yticklabels({'January','Febrary','March'})`

Example: `yticklabels({})`

Note

- If you specify the tick labels as a categorical array, MATLAB uses the values in the array, not the categories.
- To specify the tick values, use the `yticks` function.

---

ax — Target axes
current axes (default) | single `Axes` object | vector of `Axes` objects

Target axes, specified as a single `Axes` object or a vector of `Axes` objects. If you do not specify the axes, then `yticklabels` uses the current axes.

Output Arguments

`yl` — Current tick labels
cell array of character vectors | character array

Current tick labels, returned as a cell array of character vectors or a character array.

`m` — Current tick labels mode
`'auto'` | `'manual'`

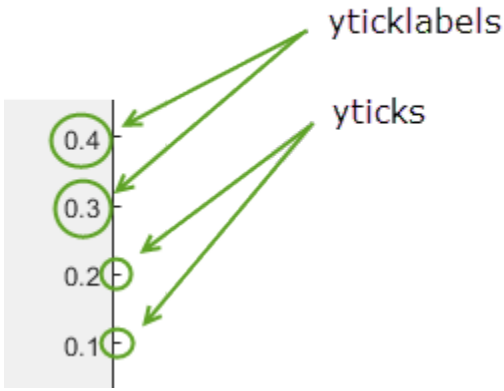
Current tick labels mode, returned as one of these values:

- `'auto'` — Automatically determine the *y*-axis tick labels.
- `'manual'` — Use manually specified *y*-axis tick labels.

More About

Tick Labels

The tick labels are the labels that you see next to each tick mark. The tick values are the locations along the *y*-axis where the tick marks appear. Set the tick values using the `yticks` function. Set the corresponding tick labels using the `yticklabels` function.



Algorithms

The `yticklabels` function sets and queries several axes properties related to the *y*-axis tick labels.

- `YTickLabel` — Property that stores the text for the *y*-axis tick labels.
- `YTickLabelMode` — Property that stores the *y*-axis tick label mode. When you set the *y*-axis tick labels using `yticklabels`, this property changes to `'manual'`.
- `YTickMode` — Property that stores the *y*-axis tick value mode. When you set the *y*-axis tick labels using `yticklabels`, this property changes to `'manual'`.

See Also

Functions
`xticklabels` | `ylim` | `ytickangle` | `ytickformat` | `yticks` | `zticklabels`

Properties
Axes

Topics
"Specify Axis Tick Values and Labels"

Introduced in R2016b

yticks

Set or query *y*-axis tick values

Syntax

```
yticks(ticks)
yt = yticks
```

```
yticks('auto')
yticks('manual')
m = yticks('mode')
```

```
___ = yticks(ax, ___ )
```

Description

`yticks(ticks)` sets the *y*-axis tick values on page 1-17249, which are the locations along the *y*-axis where the tick marks appear. Specify `ticks` as a vector of increasing values; for example, `[0 2 4 6]`. This command affects the current axes.

`yt = yticks` returns the current *y*-axis tick values as a vector.

`yticks('auto')` sets an automatic mode, enabling the axes to determine the *y*-axis tick values. Use this option if you change the tick values and then want to set them back to the default values.

`yticks('manual')` sets a manual mode, freezing the *y*-axis tick values at the current values. Use this option if you want to retain the current tick values when resizing the axes or adding new data to the axes.

`m = yticks('mode')` returns the current *y*-axis tick values mode, which is either `'auto'` or `'manual'`. By default, the mode is automatic unless you specify tick values or change the mode to manual.

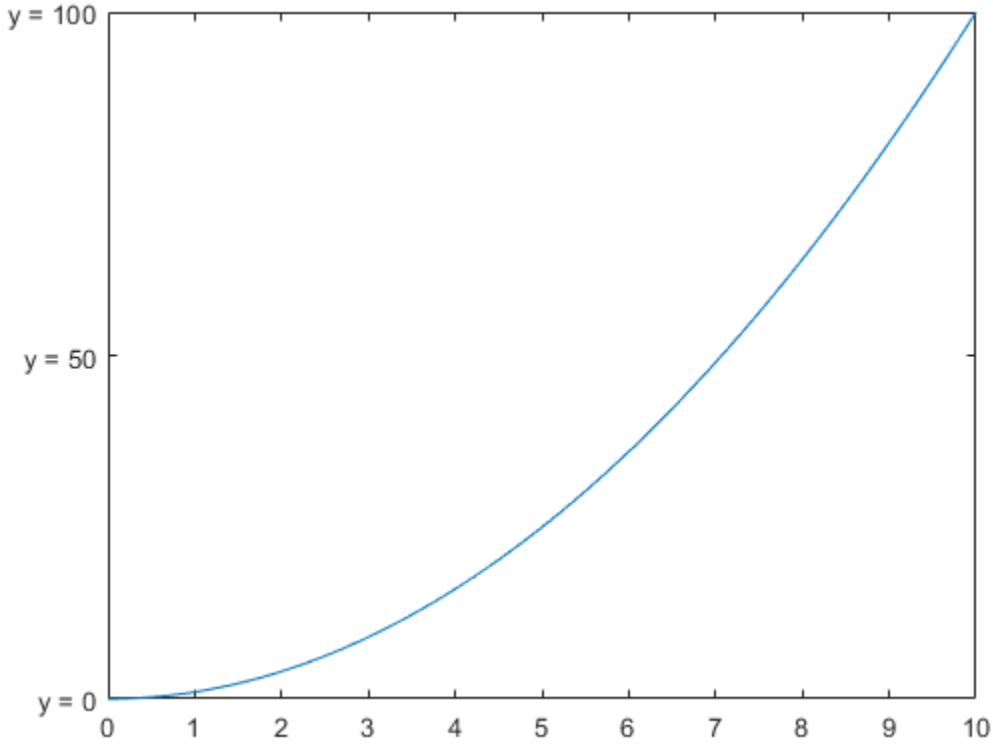
`___ = yticks(ax, ___ )` uses the axes specified by `ax` instead of the current axes. Specify `ax` as the first input argument for any of the previous syntaxes.

Examples

Specify y-Axis Tick Values and Labels

Create a line plot. Display tick marks along the *y*-axis at the values 0, 50, and 100. Then, specify a label for each tick mark.

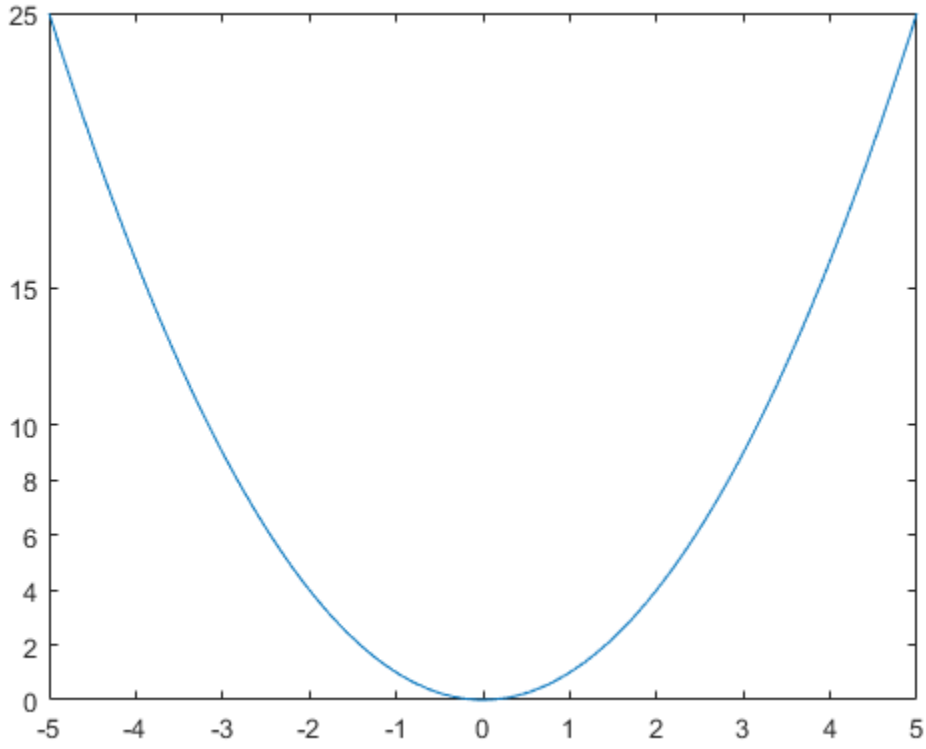
```
x = linspace(0,10);
y = x.^2;
plot(x,y)
yticks([0 50 100])
yticklabels({'y = 0','y = 50','y = 100'})
```



Specify Nonuniform y-Axis Tick Values

Display tick marks along the *y*-axis at nonuniform values between 0 and 25. MATLAB® labels the tick marks with the numeric values.

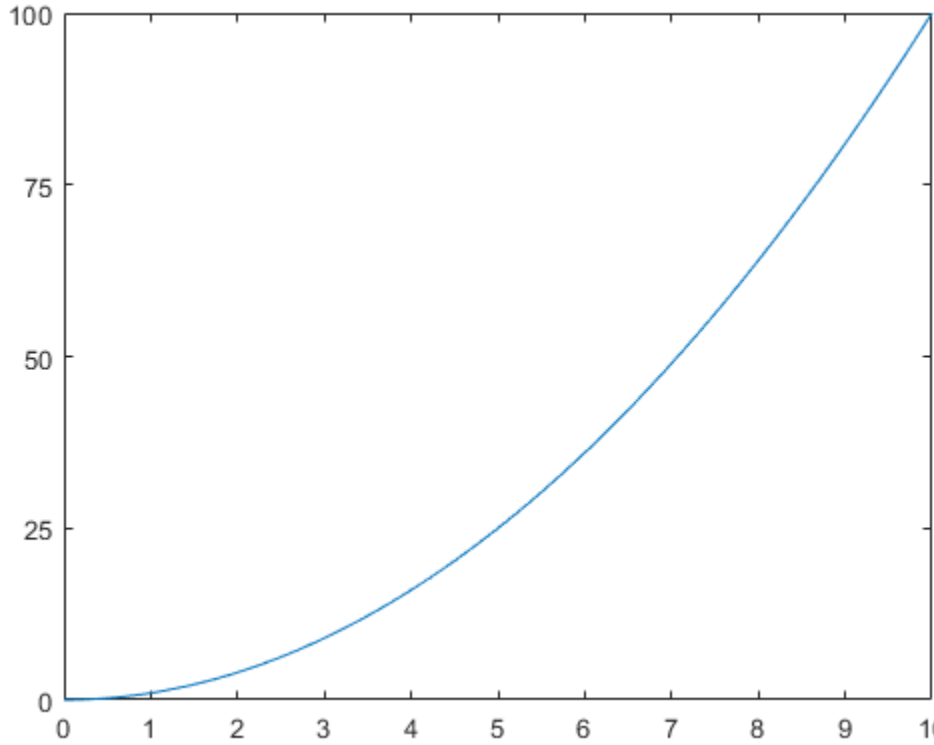
```
x = linspace(-5,5);
y = x.^2;
plot(x,y)
yticks([0 2 4 6 8 10 15 25])
```



Increment y-Axis Tick Values by 25

Display tick marks along the *y*-axis at increments of 25, starting from 0 and ending at 100.

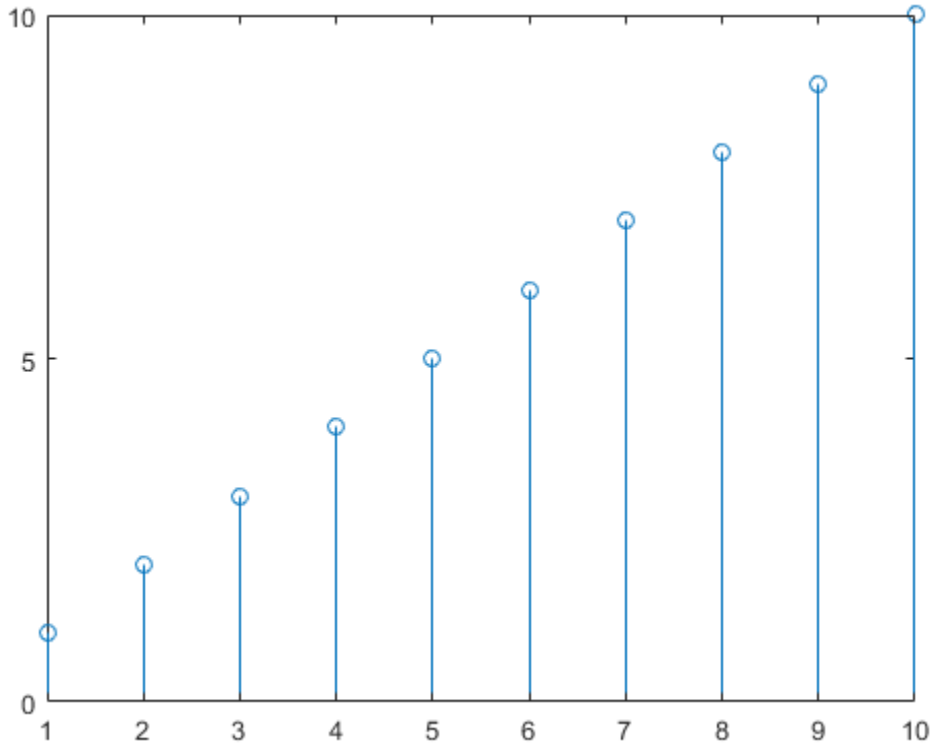
```
x = linspace(0,10);
y = x.^2;
plot(x,y)
yticks(0:25:100)
```



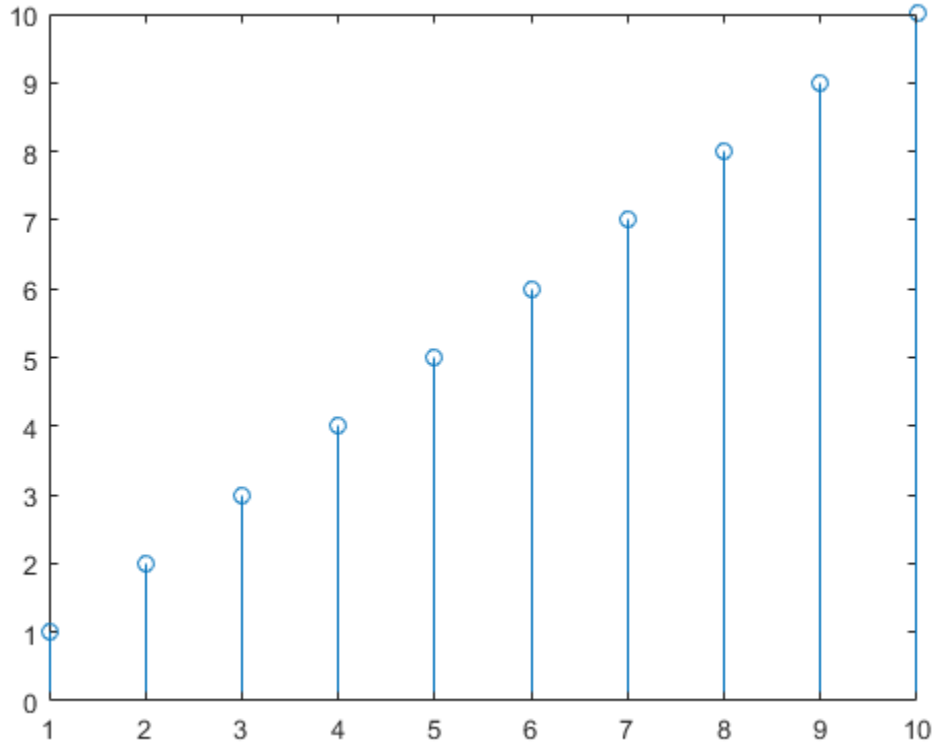
Set y-Axis Tick Values Back to Default Values

Create a stem chart and specify the *y*-axis tick values. Then, set the *y*-axis tick values back to the default values.

```
stem(1:10)
yticks([0 5 10])
```




```
yticks('auto')
```

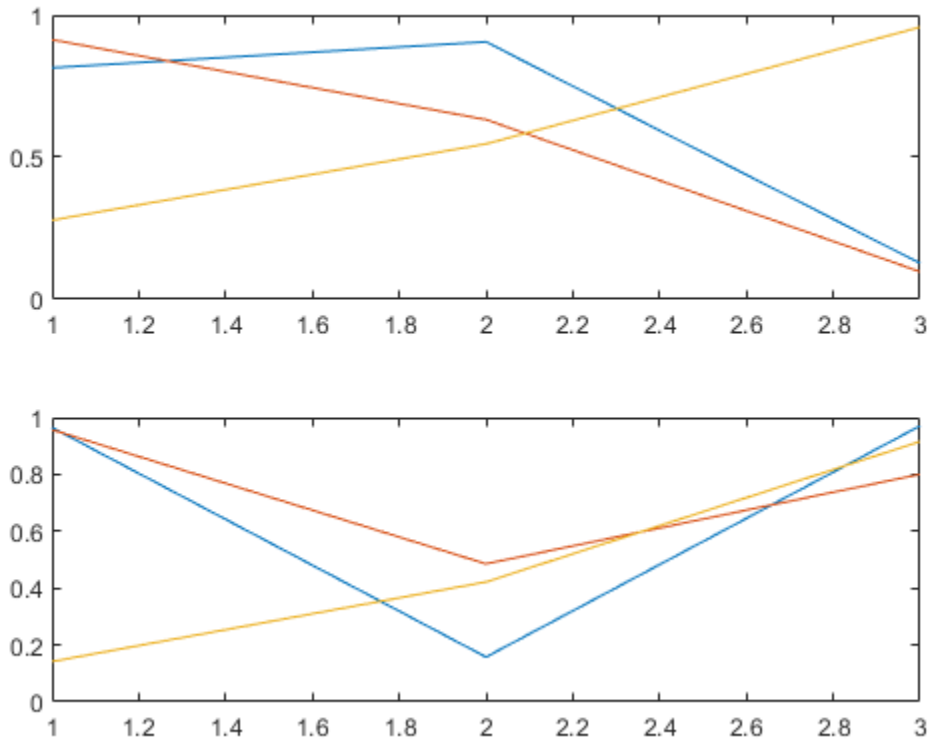


Specify y-Axis Tick Values for Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Plot data into each axes. Set the *y*-axis ticks for the lower plot by passing `ax2` as the first input argument to the `yticks` function.

```
tiledlayout(2,1)
ax1 = nexttile;
plot(rand(3))
```

```
ax2 = nexttile;
plot(rand(3))
yticks(ax2,0:.2:1)
```

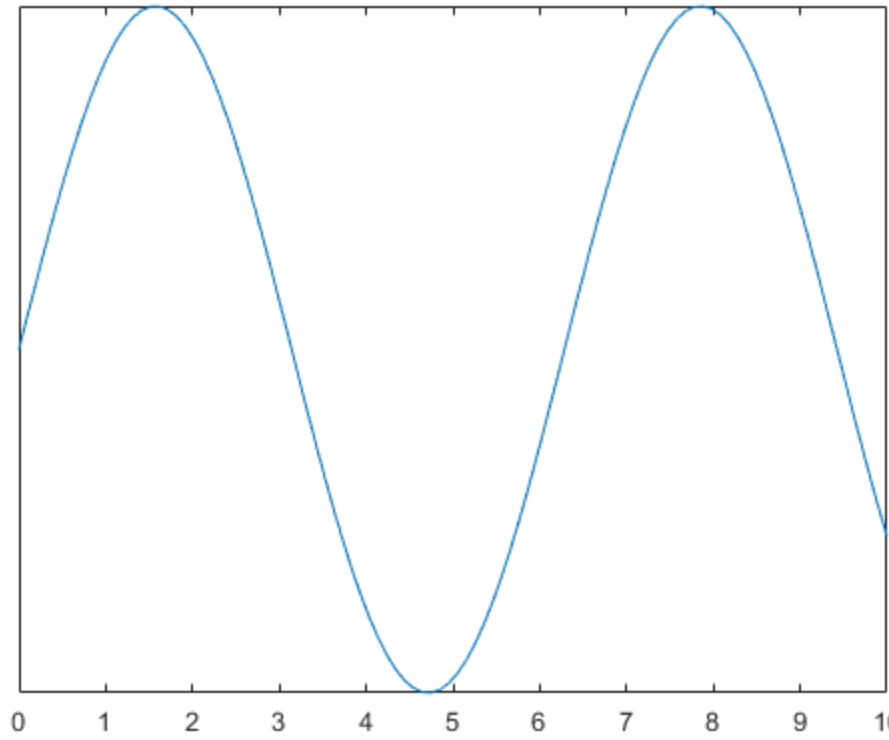


Remove y-Axis Tick Marks

Remove the tick marks along the *y*-axis by specifying the tick values as an empty array.

```
x = linspace(0,10);
y = sin(x);
```

```
plot(x,y)
yticks([])
```



Input Arguments

`ticks` — Tick values
vector of increasing values

Tick values on page 1-17249, specified as a vector of increasing values. If you do not want tick marks along the *y*-axis, specify an empty vector `[]`.

You can specify the tick values as numeric, categorical, datetime, or duration values. However, the type of values that you specify must match the type of values along the *y*-axis.

Example: `yticks([pi 2*pi 3*pi 4*pi])`

Example: `yticks(0:10:100)`

Example: `yticks([])`

Note To specify the tick labels, use the `yticklabels` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`ax` — Target axes
current axes (default) | single `Axes` object | vector of `Axes` objects

Target axes, specified as a single `Axes` object or a vector of `Axes` objects. If you do not specify the axes, then `yticks` uses the current axes.

Output Arguments

`yt` — Current tick values
vector

Current tick values, returned as a vector.

`m` — Current mode
`'auto'` | `'manual'`

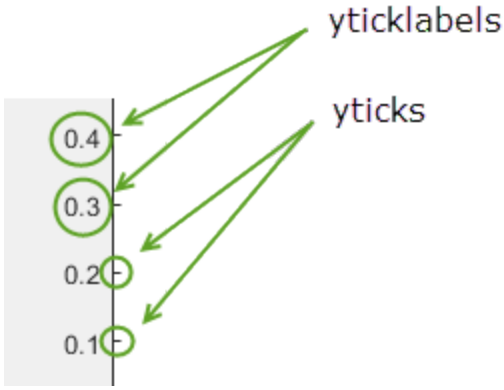
Current mode, returned as one of these values:

- `'auto'` — Automatically determine the *y*-axis tick values.
- `'manual'` — Use manually specified *y*-axis tick values.

More About

Tick Values

The tick values are the locations along the *y*-axis where the tick marks appear. The tick labels are the labels that you see next to each tick mark. Set the tick values using the `yticks` function. Set the corresponding tick labels using the `yticklabels` function.



Algorithms

The `yticks` function sets and queries several axes properties related to the *y*-axis tick values.

- `YTick` — Property that stores the *y*-axis tick values.
- `YTickMode` — Property that stores the *y*-axis tick value mode. When you set the *y*-axis tick values, this property changes to `'manual'`.

See Also

Functions
`xticks` | `ylim` | `ytickangle` | `ytickformat` | `yticklabels` | `zticks`

Properties
Axes

Topics
"Specify Axis Tick Values and Labels"

Introduced in R2016b

yyaxis

Create chart with two y-axes

Syntax

```
yyaxis left
yyaxis right
```

```
yyaxis(ax, ___ )
```

Description

`yyaxis left` activates the side of the current axes associated with the left *y*-axis. Subsequent graphics commands target the left side. If the current axes do not include two *y*-axes, then this command adds a second *y*-axis. If there are no axes, then this command first creates them.

`yyaxis right` activates the side of the current axes associated with the right *y*-axis. Subsequent graphics commands target the right side.

`yyaxis(ax, ___ )` specifies the active side for the axes `ax` instead of the current axes. If the axes do not include two *y*-axes, then this command adds a second *y*-axis. Specify the axes as the first input argument. Use single quotes around `'left'` and `'right'`.

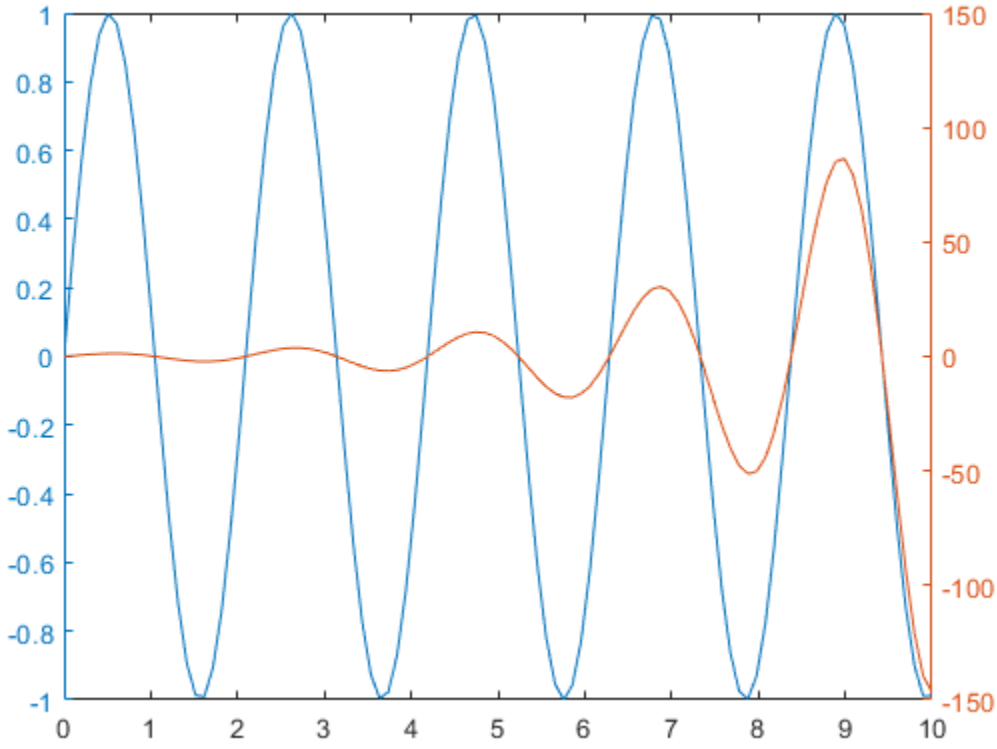
Examples

Plot Data Using Two y-Axes

Create axes with a *y*-axis on both the left and right sides. Plot a set of data against the left *y*-axis. Then, use `yyaxis right` to activate the right side so that subsequent graphics functions target it. Plot a second set of data against the right *y*-axis and set the limits for the right *y*-axis.

```
x = linspace(0,10);
y = sin(3*x);
```

```
yyaxis left
plot(x,y)

z = sin(3*x).*exp(0.5*x);
yyaxis right
plot(x,z)
ylim([-150 150])
```

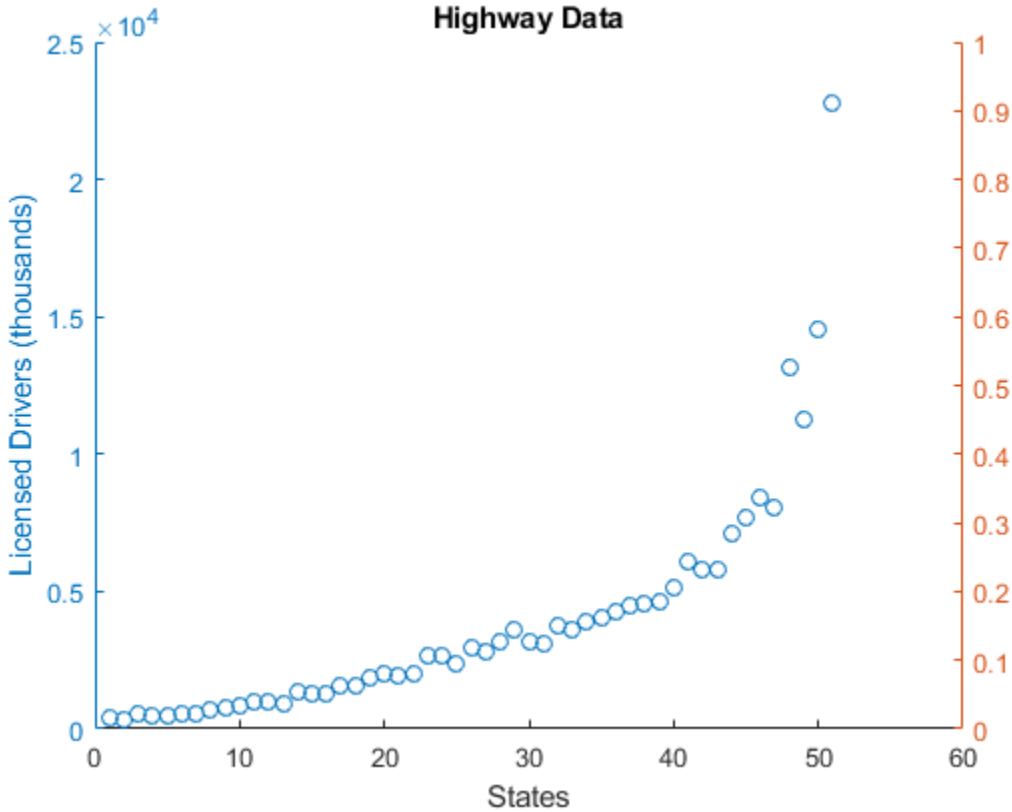


Add Title and Axis Labels to Each Side

Create a chart with two *y*-axes and add a title and axis labels to each side.

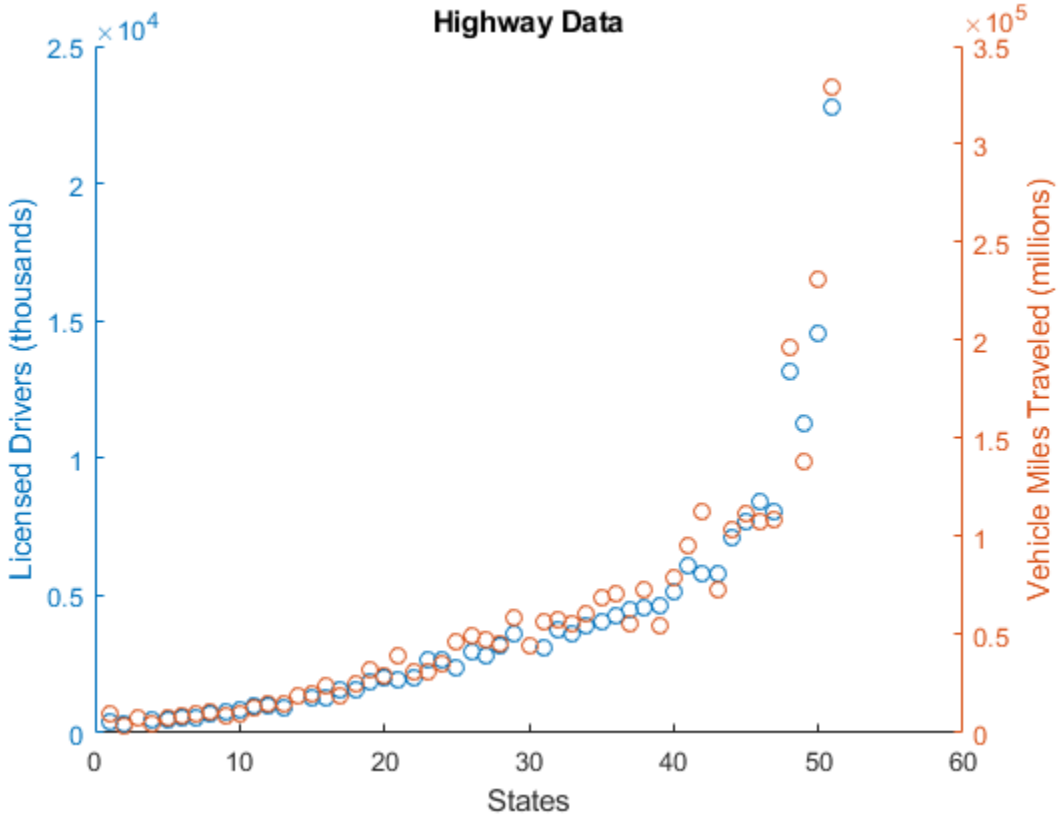
Load the matrix `hwydata` from the example file `accidents.mat`. Create a scatter plot of the fifth column in `hwydata` against the left *y*-axis. Add a title and axis labels.

```
load('accidents.mat','hwydata')
ind = 1:51;
drivers = hwydata(:,5);
yyaxis left
scatter(ind,drivers)
title('Highway Data')
xlabel('States')
ylabel('Licensed Drivers (thousands)')
```



Create a second scatter plot of the seventh column in `hwydata` against the right *y*-axis. Then, label the right *y*-axis.

```
pop = hwydata(:,7);
yyaxis right
scatter(ind,pop)
ylabel('Vehicle Miles Traveled (millions)')
```

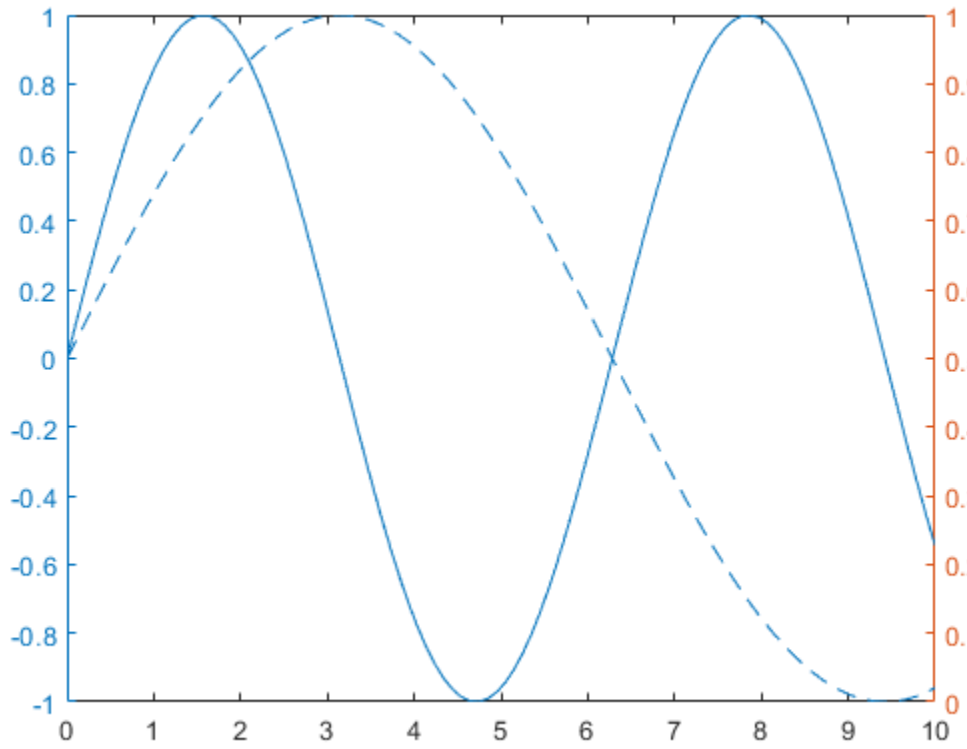


Plot Multiple Sets of Data on Each Side

Plot two lines against the left *y*-axis by using the `hold on` command.

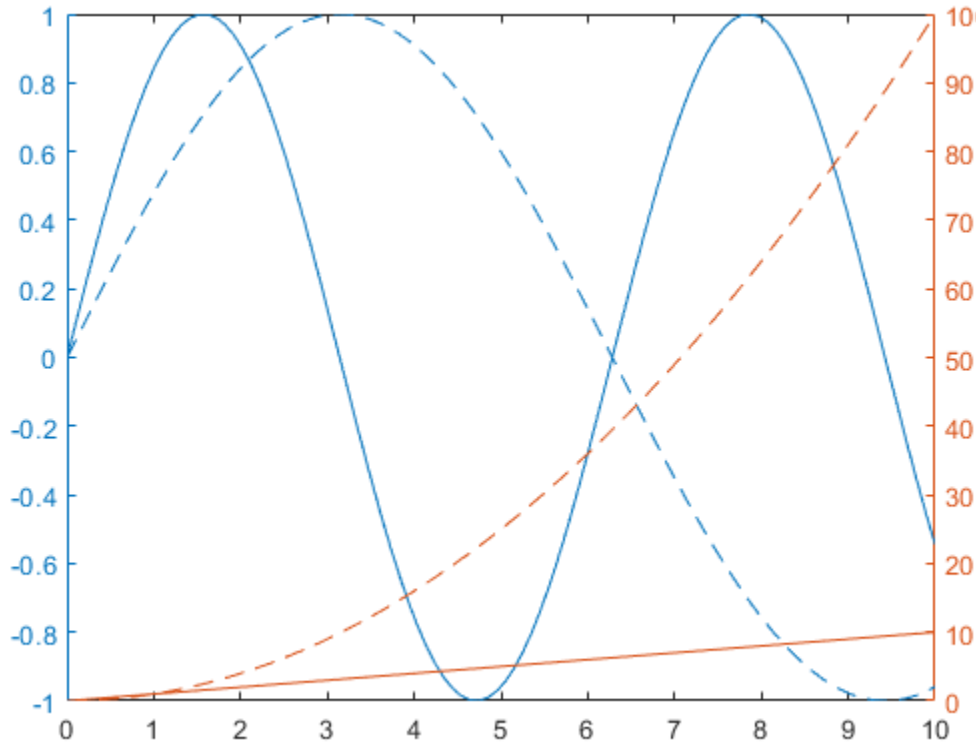
```
x = linspace(0,10);
yl1 = sin(x);
yl2 = sin(x/2);
```

```
yyaxis left
plot(x,yl1)
hold on
plot(x,yl2)
```



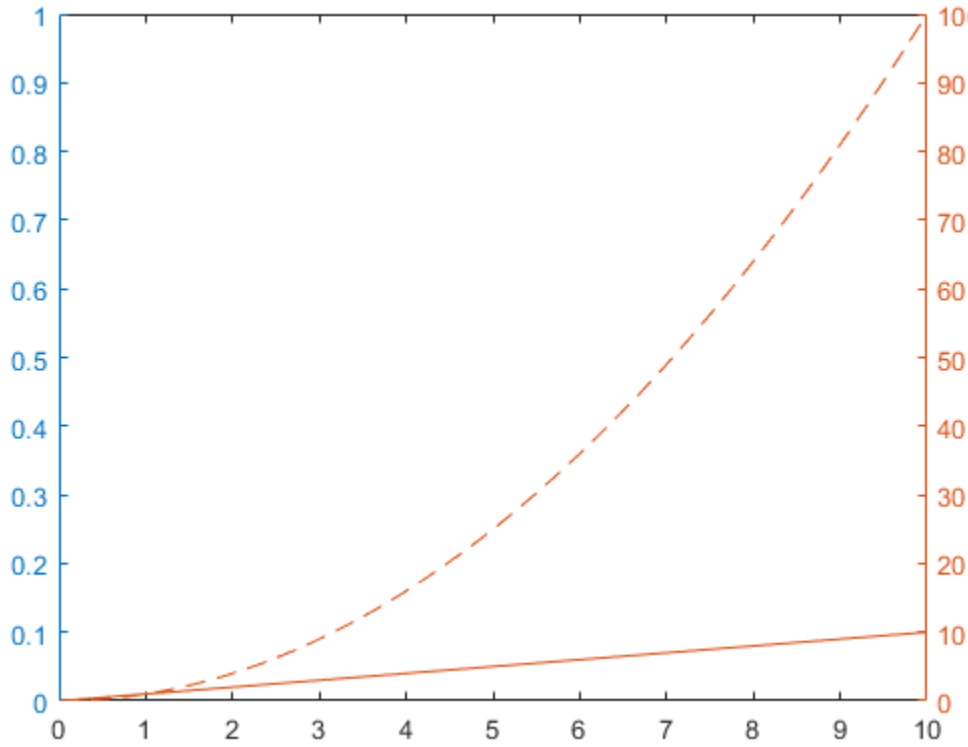
Plot two lines against the right *y*-axis. The `hold` command affects both the left and right *y*-axes, so you do not need to reissue it. After plotting, turn `hold` back off.

```
yr1 = x;
yr2 = x.^2;
yyaxis right
plot(x,yr1)
plot(x,yr2)
hold off
```



Clear the left side by making it active and then using the `cla` command.

```
yyaxis left
cla
```

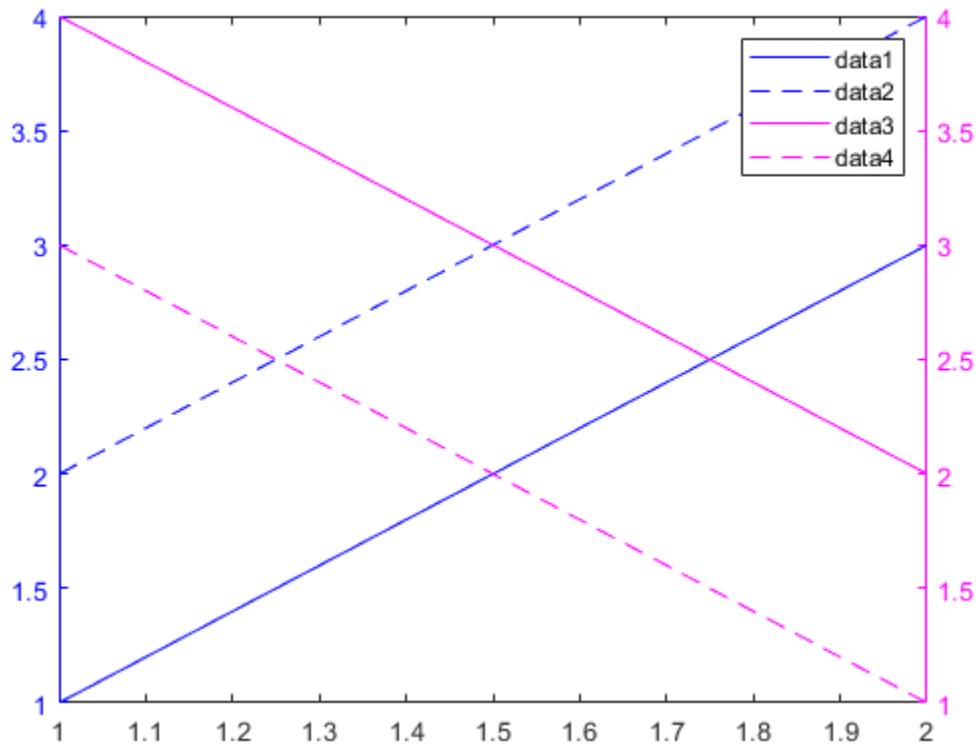


Control Colors for Each Side

Specify the color scheme for each side of the axes by setting the color order to the two colors that you want to use. Starting in R2019b, you can use the `colororder` function to set the color order. Then, plot two lines against the left *y*-axis and two lines against the right *y*-axis. Add a legend.

```
colororder({'b','m'})

yyaxis left
y = [1 2; 3 4];
plot(y)
```

```
yyaxis right
z = [4 3; 2 1];
plot(z)
legend
```



Control Individual Plot Colors

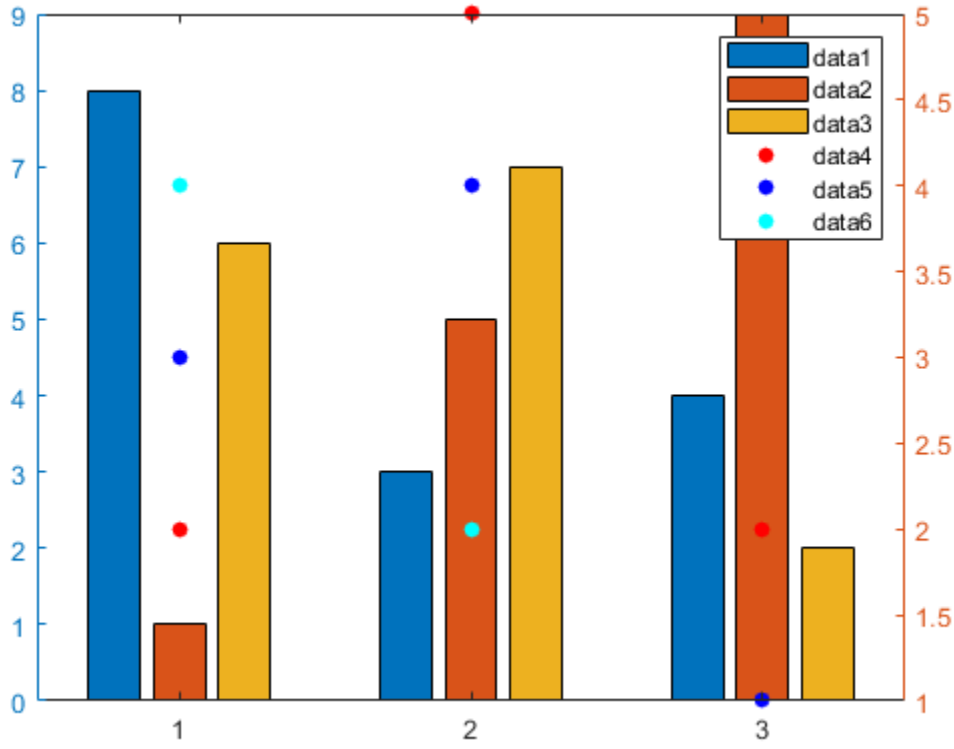
Control individual plot colors by setting the color order for each side of the axes.

Plot three bar charts against the left side. Use a different color for each bar series by setting the color order for the left side to the default color order.

```
yyaxis left
bar(magic(3));
colororder('default')
```

Plot three scatter plots against the right side. Use a different color for each scatter plot by setting the color order to an array of color names. Alternatively, you can specify the colors using a matrix of RBG triplets. Then add a legend.

```
yyaxis right
scatter([1 2 3],[2 5 2],'filled')
hold on
scatter([1 2 3],[3 4 1],'filled')
scatter([1 2 3],[4 2 4],'filled')
hold off
colororder({'r','b','c'})
legend
```

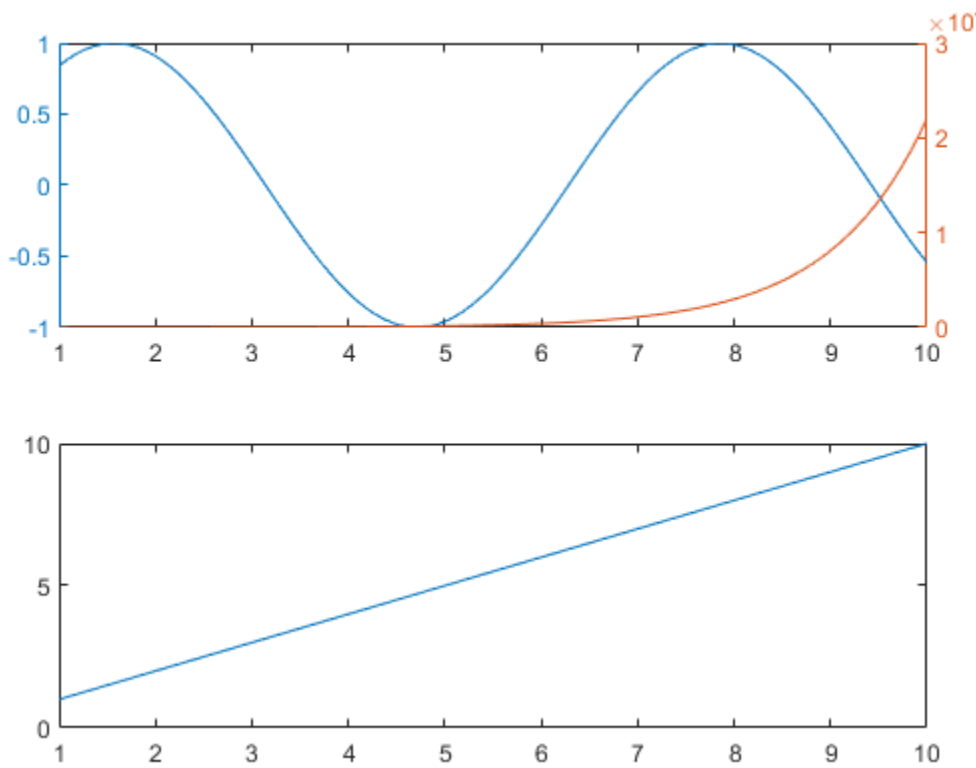


Add Second *y*-Axis to Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Add a second *y*-axis to the top axes by specifying `ax1` as the first input to `yyaxis`. If you do not specify the axes, then `yyaxis` adds a second *y*-axis to the current axes.

```
x = linspace(1,10);
tiledlayout(2,1)
```

```
% Top plot
ax1 = nexttile;
yyaxis(ax1,'left')
plot(ax1,x,sin(x))
yyaxis(ax1,'right')
plot(ax1,x,exp(x))

% Bottom plot
ax2 = nexttile;
plot(ax2,1:10)
```



Input Arguments

ax — Target axes
current axes (default) | `Axes` object

Target axes, specified as an `Axes` object. If you do not specify an `Axes` object, then `yyaxis` uses the current axes.

Limitations

- When working with two y-axes, you cannot:
  - Rotate the axes (2-D view only).
  - Pin annotations.
  - Copy the axes object using `copyobj`.

Tips

- To determine which side of the axes is active, query the `YAxisLocation` property for the `Axes` object. The property is set to `'left'` when the left side is active and `'right'` when the right side is active. The `YAxisLocation` property for an `Axes` object with two *y*-axes is read only.
- To clear the active side, use `cla`. To clear both sides of the axes and remove the right *y*-axis, use `cla reset`. Alternatively, you can wait to clear both sides and remove the right *y*-axis until the next plotting command by setting the `NextPlot` property of the `Axes` object to `'replaceall'`.
- The `Children` property of the `Axes` object only contains the children for the active side. To access all the children for both sides, use the `allchild` function.

Algorithms

Grid Lines

Grid lines correspond with the tick mark locations along the left *y*-axis.

Colors and Line Styles

Plots associated with a particular side of the axes use the same color as the *y*-axis on that side. If a side contains multiple lines, then the lines cycle through the line style order. The left *y*-axis uses the first color in the color order of the `Axes` object, and the right *y*-axis uses the second color.

If you add a second *y*-axis to an `Axes` object that contains charts, then the existing charts and the left *y*-axis do not change colors. The right *y*-axis uses the next color in the color order.

Axes Properties

Axes properties related to the *y*-axis have two values. However, MATLAB gives access only the value for the active side. For example, if the left side is active, then the `YLim` property of the `Axes` object contains the limits for the left *y*-axis. However, if the right side is active, then the `YLim` property contains the limits for the right *y*-axis.

An exception is that the `YAxis` property of the `Axes` object contains an array of two ruler objects (one for each *y*-axis). You can use the rulers to access each *y*-axis without depending on the active side. For an example, see “Modify Properties of Charts with Two y-Axes”.

See Also

`bar` | `cla` | `hold` | `plot` | `stairs` | `stem`

Topics

“Create Chart with Two y-Axes”
“Modify Properties of Charts with Two y-Axes”

Introduced in R2016a

yyyymmdd

Convert MATLAB datetime to YYYYMMDD numeric value

Syntax

```
d = yyyymmdd(t)
```

Description

`d = yyyymmdd(t)` returns a `double` array containing integers whose digits represent the datetime values in `t`. For example, the date July 16, 2014 is converted to the integer 20140716. The conversion is performed this way:

```
d = 10000*year(t) + 100*month(t) + day(t)
```

Examples

Convert Datetime Array to YYYYMMDD Numeric Values

Create an array of YYYYMMDD numeric values that represent dates.

```
d = [20140628 20140701 20140704]
```

```
d = 1×3

    20140628    20140701    20140704
```

Convert the dates to datetime values.

```
t = datetime(d,'ConvertFrom','yyyymmdd')
```

```
t = 1x3 datetime array
   28-Jun-2014 00:00:00   01-Jul-2014 00:00:00   04-Jul-2014 00:00:00
```

Convert the datetime values back to YYYYMMDD numeric values.

```
d2 = yyyymmdd(t)
```

```
d2 = 1×3

    20140628    20140701    20140704
```

Input Arguments

t — Input date and time
`datetime` array

Input date and time, specified as a `datetime` array.

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`datenum` | `datetime` | `exceltime` | `juliandate` | `posixtime`

Introduced in R2014b

zlabel

Label z-axis

Syntax

```
zlabel(txt)
zlabel(txt,Name,Value)

zlabel(ax, ___ )

h = zlabel( ___ )
```

Description

`zlabel(txt)` labels the *z*-axis of the current axes with the text, `txt`. Reissuing the `zlabel` command causes the new label to replace the old label.

`zlabel(txt,Name,Value)` additionally specifies the text object properties using one or more `Name,Value` pair arguments.

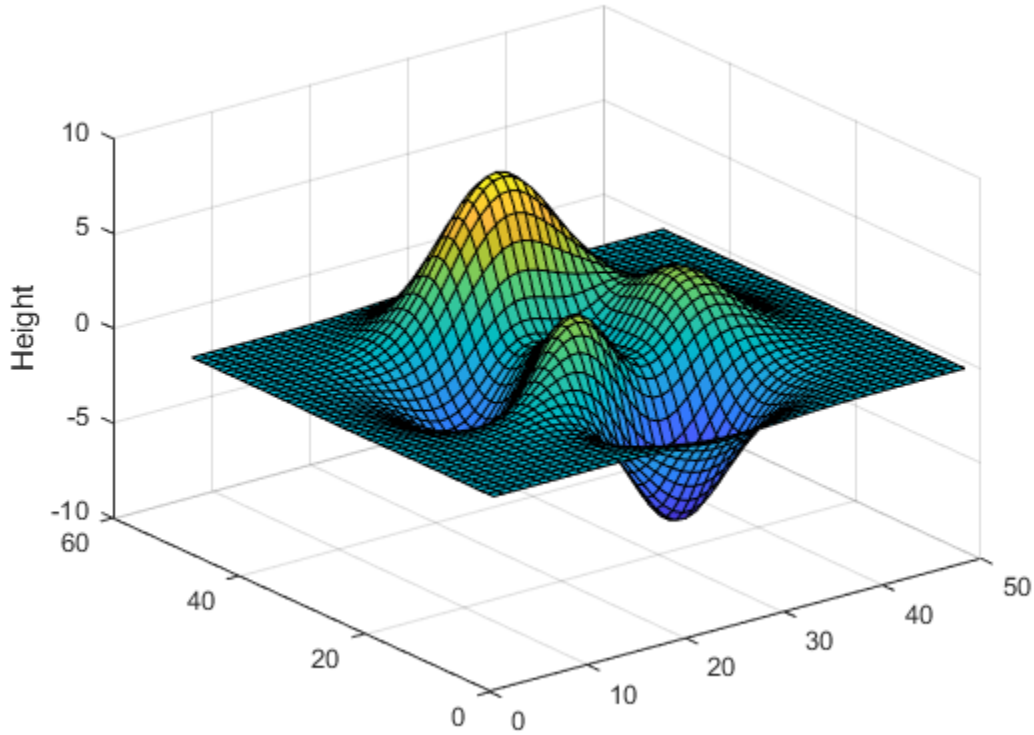
`zlabel(ax, ___ )` adds the label to the axes specified by `ax`. This syntax allows you to specify the axes to which to add a label. `ax` can precede any of the input argument combinations in the previous syntaxes.

`h = zlabel( ___ )` returns the handle to the text object used as the *z*-axis label. The handle is useful when making future modifications to the label.

Examples

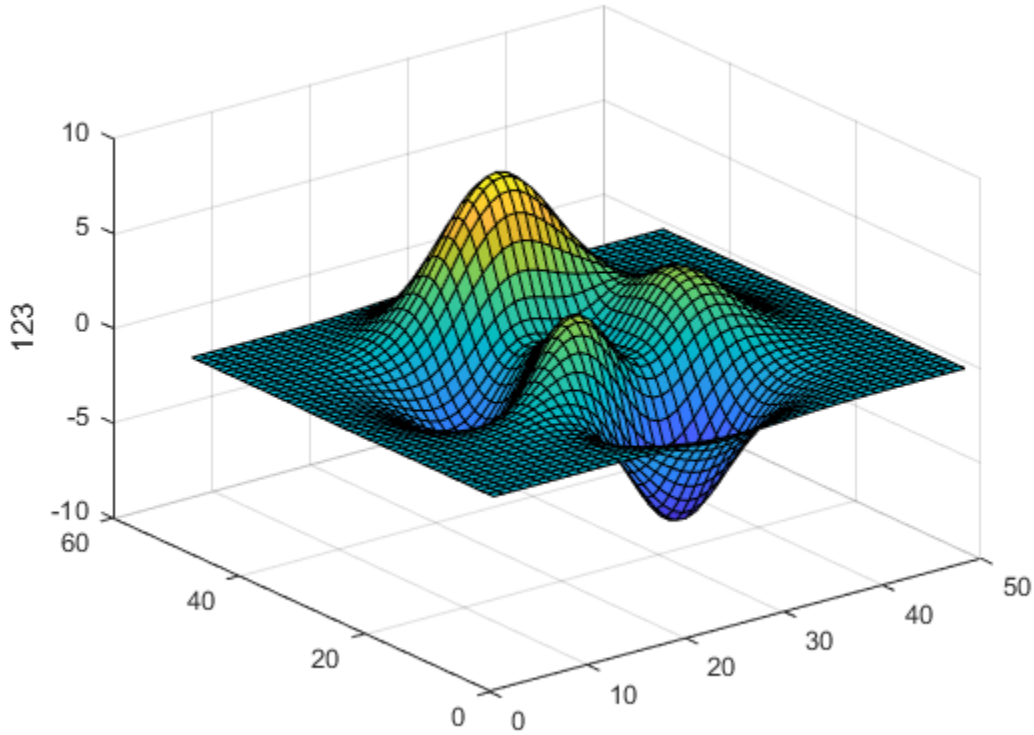
Label z-Axis

```
surf(peaks)
zlabel('Height')
```



Label z-Axis with Numeric Input

```
figure
surf(peaks)
zlabel(123)
```

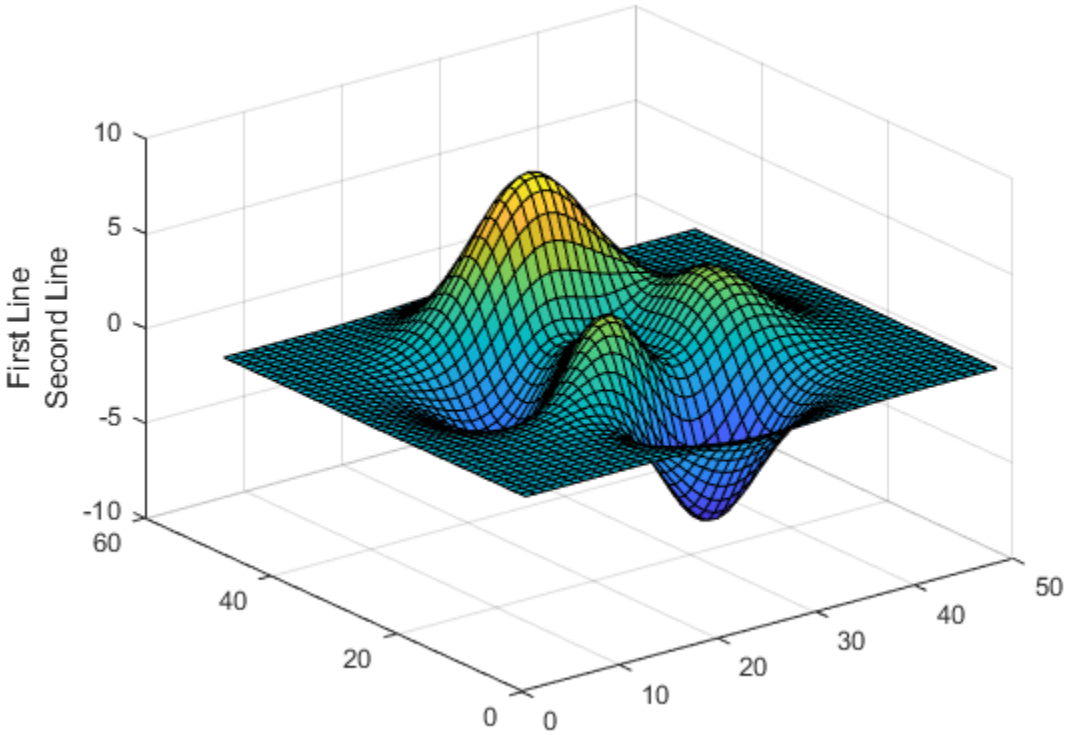


MATLAB® displays 123 beside the *z*-axis.

Create Multiline z-Axis Label

Create a multiline label using a multiline cell array.

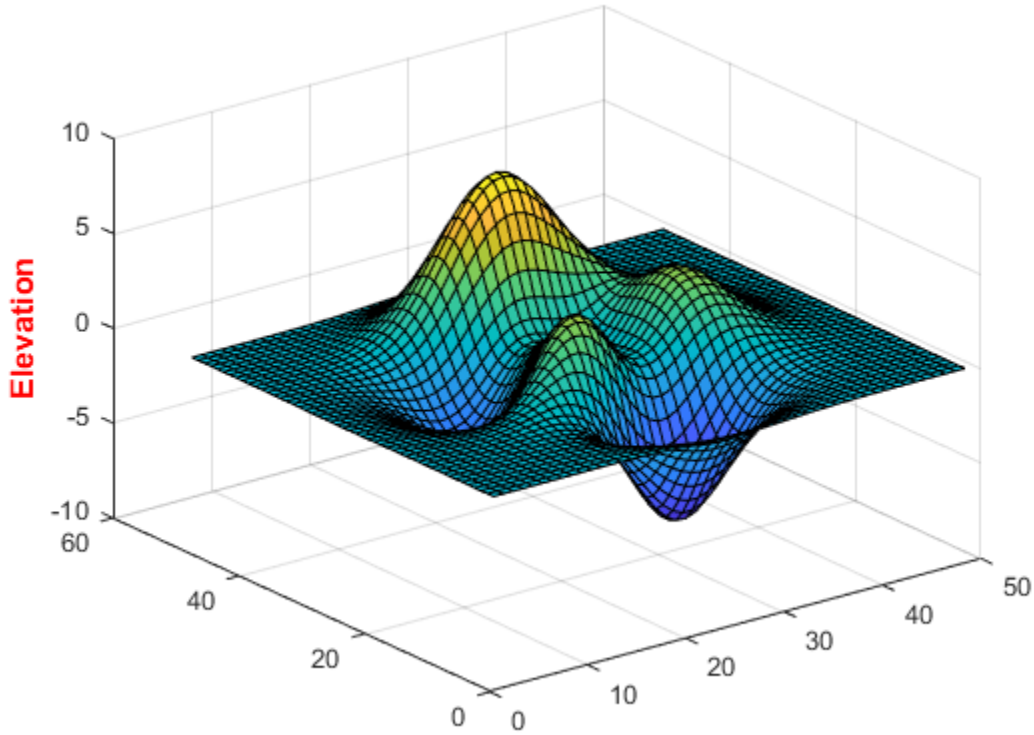
```
figure
surf(peaks)
zlabel({'First Line';'Second Line'})
```



Label z-Axis and Set Font Properties

Use `Name,Value` pairs to set the font size, font weight, and text color properties of the *z*-axis label.

```
figure
surf(peaks)
zlabel('Elevation','FontSize',12,...
       'FontWeight','bold','Color','r')
```

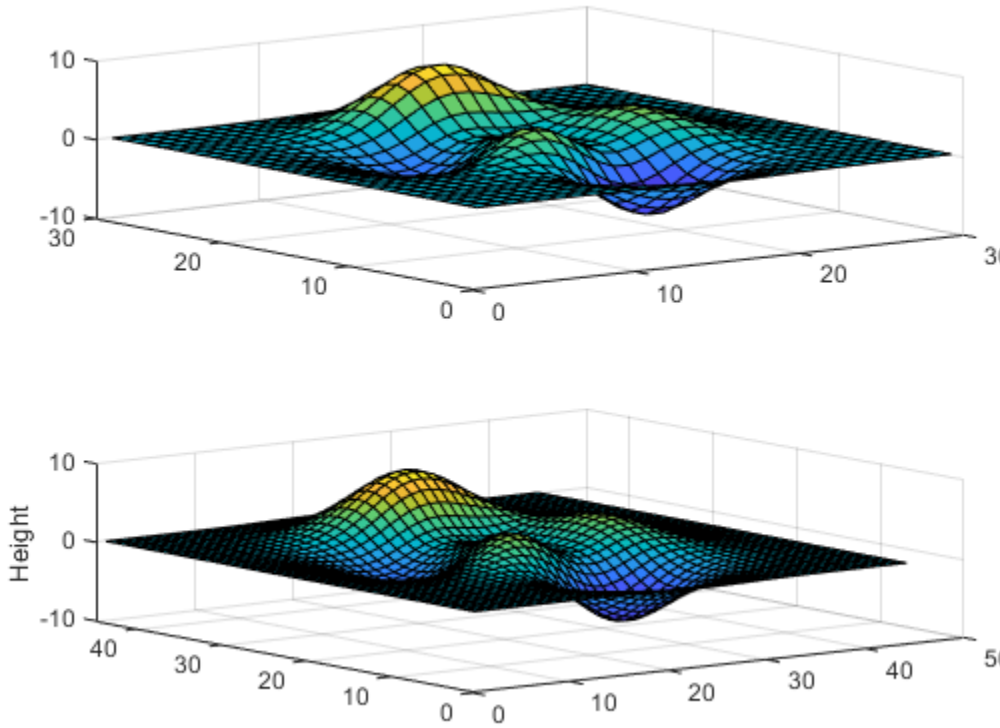


`'FontSize',12` displays the label text in 12-point font. `'FontWeight','bold'` makes the text bold. `'Color','r'` sets the text color to red.

Label z-Axis of Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Create two surface plots, and add a *z*-axis label to the second plot by specifying `ax2` as the first input argument to `zlabel`.

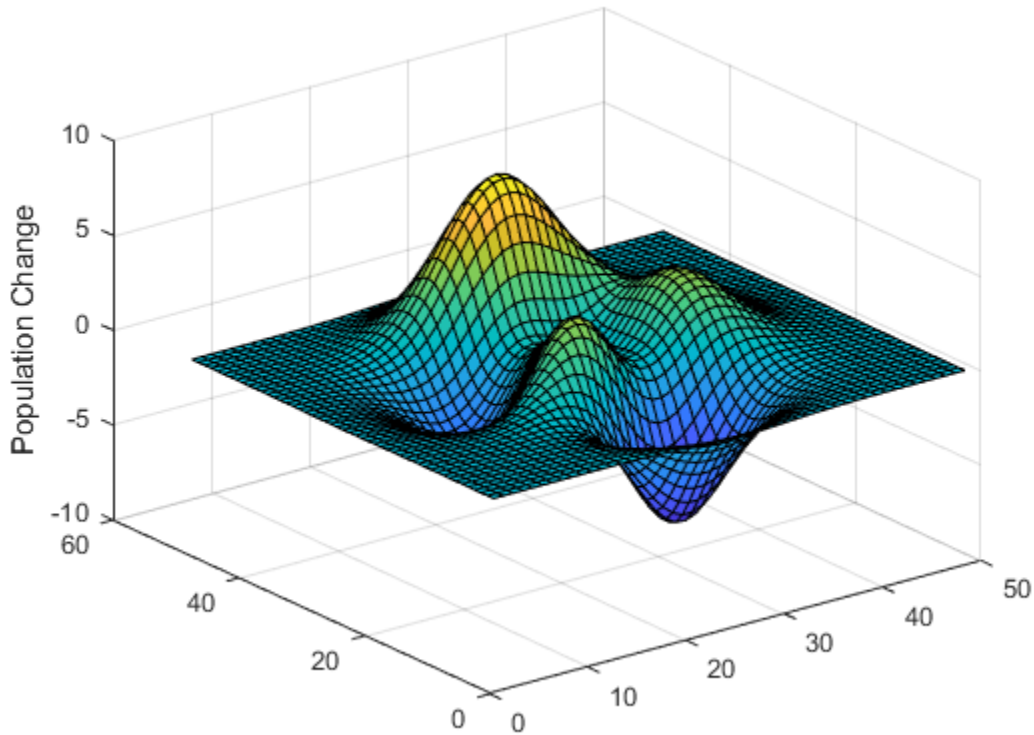
```
tiledlayout(2,1)
ax1 = nexttile;
surf(ax1,peaks(30))

ax2 = nexttile;
surf(ax2,peaks(45))
zlabel(ax2,'Height')
```



Label z-Axis and Return Object Handle

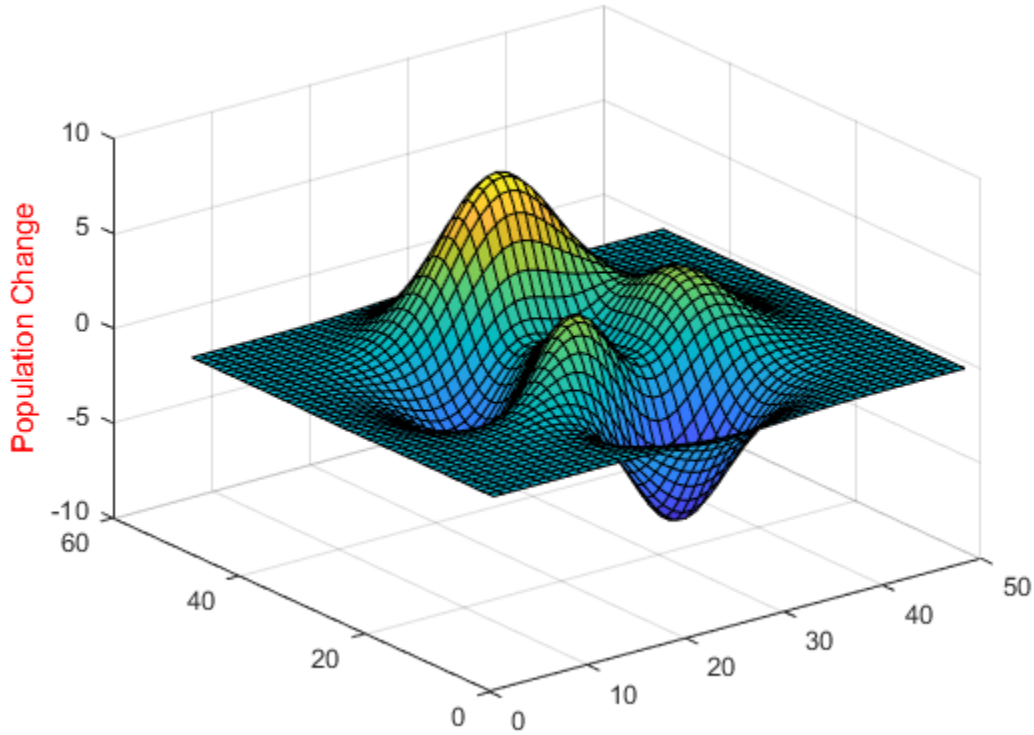
Label the *z*-axis and return the text object used as the label.

```
surf(peaks)
t = zlabel('Population Change');
```



Set the color of the label to red. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead.

```
t.Color = 'red';
```



Input Arguments

`txt` — Axis label

string scalar | character vector | string array | character array | cell array | categorical array | numeric value

Axis label, specified as a string scalar, character vector, string array, character array, cell array, categorical array, or numeric value.

Example: `'my label'`

Example: `{'first line','second line'}`

Example: 123

To include numeric variables with text in a label, use the `num2str` function. For example:

```
x = 42;
txt = ['The value is ',num2str(x)];
```

To include special characters, such as superscripts, subscripts, Greek letters, or mathematical symbols use TeX markup. For a list of supported markup, see the `Interpreter` property.

To create multiline labels:

- Use a string array, where each element contains a line of text, such as `["first line","second line"]`.
- Use a cell array, where each cell contains a line of text, such as `{'first line','second line'}`.
- Use a character array, where each row contains the same number of characters, such as `['abc'; 'ab ']`.
- Use `sprintf` to create text with a new line character, such as `sprintf('first line \n second line')`.

Numeric labels are converted to text using `sprintf('%g',value)`. For example, `12345678` displays as `1.23457e+07`.

Note

- If you specify the label as a categorical array, MATLAB uses the values in the array, not the categories.
- The words `default`, `factory`, and `remove` are reserved words that will not appear in a label when quoted as a normal characters. To display any of these words individually, precede them with a backslash, such as `'\default'` or `'\remove'`.

ax — Axes object

axes object

Axes object. If you do not specify an axes, then the `zlabel` function uses the current axes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Color','red','FontSize',12` specifies red, 12-point font.

In addition to the following, you can specify other text object properties using `Name,Value` pair arguments. See Text.

`FontSize` — Font size
`11` (default) | scalar value greater than `0`

Font size, specified as a scalar value greater than `0` in point units. One point equals 1/72 inch. To change the font units, use the `FontUnits` property.

Setting the font size properties for the associated axes also affects the label font size. The label font size updates to equal the axes font size times the label scale factor. The `FontSize` property of the axes contains the axes font size. The `LabelFontSizeMultiplier` property of the axes contains the label scale factor. By default, the axes font size is 10 points and the scale factor is 1.1, so the *z*-axis label font size is 11 points.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`FontWeight` — Character thickness
`'normal'` (default) | `'bold'`

Character thickness, specified as `'normal'` or `'bold'`.

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.

`FontName` — Font name
supported font name | `'FixedWidth'`

Font name, specified as a supported font name or `'FixedWidth'`. To display and print text properly, you must choose a font that your system supports. The default font depends on your operating system and locale.

To use a fixed-width font that looks good in any locale, use `'FixedWidth'`. The fixed-width font relies on the root `FixedWidthFontName` property. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`Color` — Text color

`[0.15 0.15 0.15]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Text color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Example: `'blue'`

Example: `[0 0 1]`

Example: `'#0000FF'`

`Interpreter` — Interpretation of text characters

`'tex'` (default) | `'latex'` | `'none'`

Interpretation of text characters, specified as one of these values:

- `'tex'` — Interpret characters using a subset of TeX markup.
- `'latex'` — Interpret characters using LaTeX markup.
- `'none'` — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the font type and color, and include special characters in the text.

Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they modify only the next character or the characters within the curly braces. When you set the `Interpreter` property to `'tex'`, the supported modifiers are as follows.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{`*`specifier`*`}`	Font name — Replace *`specifier`* with the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{`*`specifier`*`}`	Font size —Replace *`specifier`* with a numeric scalar value in point units.	`'\fontsize{15} text'`
`\color{`*`specifier`*`}`	Font color — Replace *`specifier`* with one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Custom font color — Replace *`specifier`* with a three-element RGB triplet.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\ast	*	\chi	χ	\infty	∞
\beta	β	\psi	ψ	\clubsuit	♣
\gamma	γ	\omega	ω	\diamondsuit	♦
\delta	δ	\Gamma	Γ	\heartsuit	♥
\epsilon	ϵ	\Delta	Δ	\spadesuit	♠
\zeta	ζ	\Theta	Θ	\leftrightarrow	↔
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	µ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	′
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `Interpreter` property to `'latex'`. Use dollar symbols around the text, for example, use `'$\int_1^{20} x^2 dx$'` for inline mode or `'$$ \int_1^{20} x^2 dx$$'` for display mode.

The displayed text uses the default LaTeX font style. The `FontName`, `FontWeight`, and `FontAngle` properties do not have an effect. To change the font style, use LaTeX markup.

The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters. For multiline text, this reduces by about 10 characters per line.

For more information about the LaTeX system, see The LaTeX Project website at https://www.latex-project.org/.

Output Arguments

h — Text object
text object

Text object used as the *z*-axis label. Use `h` to access and modify properties of the label after its created.

See Also

Functions
`num2str` | `text` | `title` | `xlabel` | `ylabel`

Properties
Text

Topics
"Add Title and Axis Labels to Chart"

Introduced before R2006a

xlim

Set or query *x*-axis limits

Syntax

```
xlim(limits)
xl = xlim
```

```
xlim auto
xlim manual
m = xlim('mode')
```

```
___ = xlim(target, ___ )
```

Description

`xlim(limits)` sets the *x*-axis limits for the current axes or chart. Specify `limits` as a two-element vector of the form `[xmin xmax]`, where `xmax` is greater than `xmin`.

`xl = xlim` returns the current limits as a two-element vector.

`xlim auto` sets an automatic mode, enabling the axes to determine the *x*-axis limits. The limits span the range of the plotted data. Use this option if you change the limits and then want to set them back to the default values. This command sets the `XLimMode` property for the axes to `'auto'`.

`xlim manual` sets a manual mode, freezing the limits at the current values. Use this option if you want to retain the current limits when adding new data to the axes using the `hold on` command. This command sets the `XLimMode` property for the axes to `'manual'`.

`m = xlim('mode')` returns the current *x*-axis limits mode, which is either `'auto'` or `'manual'`. By default, the mode is automatic unless you specify limits or set the mode to manual.

`___ = xlim(target, ___ )` uses the axes or chart specified by `target` instead of the current axes. Specify `target` as the first input argument for any of the previous syntaxes.

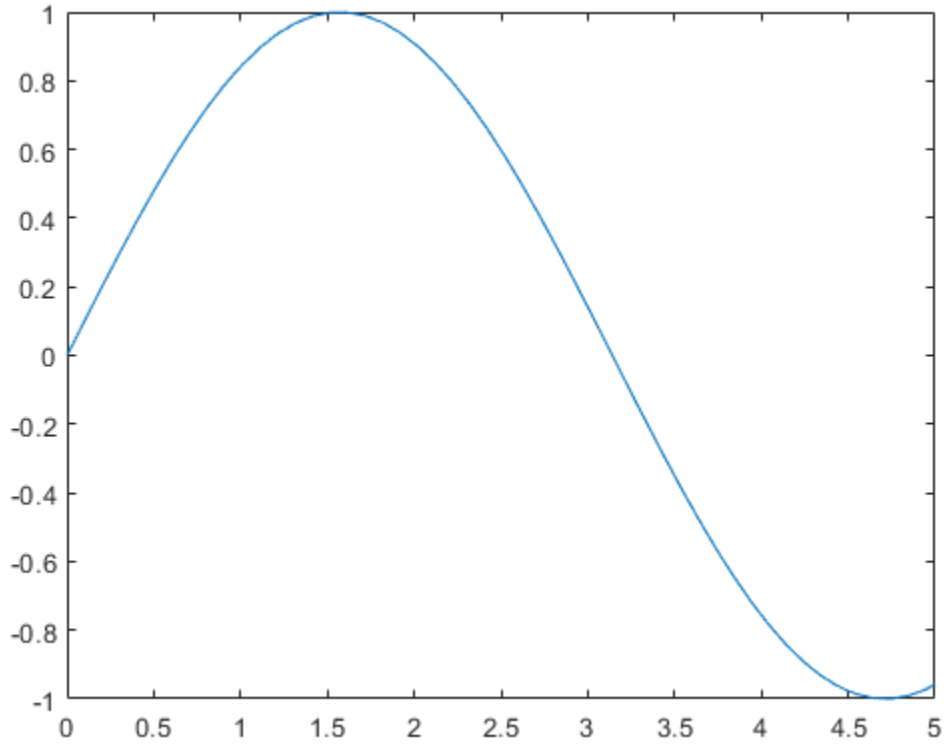
You can include an output argument if the original syntax supports an output argument. Use single quotes around the mode inputs, for example, `xlim(target,'auto')`.

Examples

Set *x*-Axis Limits

Plot a line and set the *x*-axis limits to range from 0 to 5.

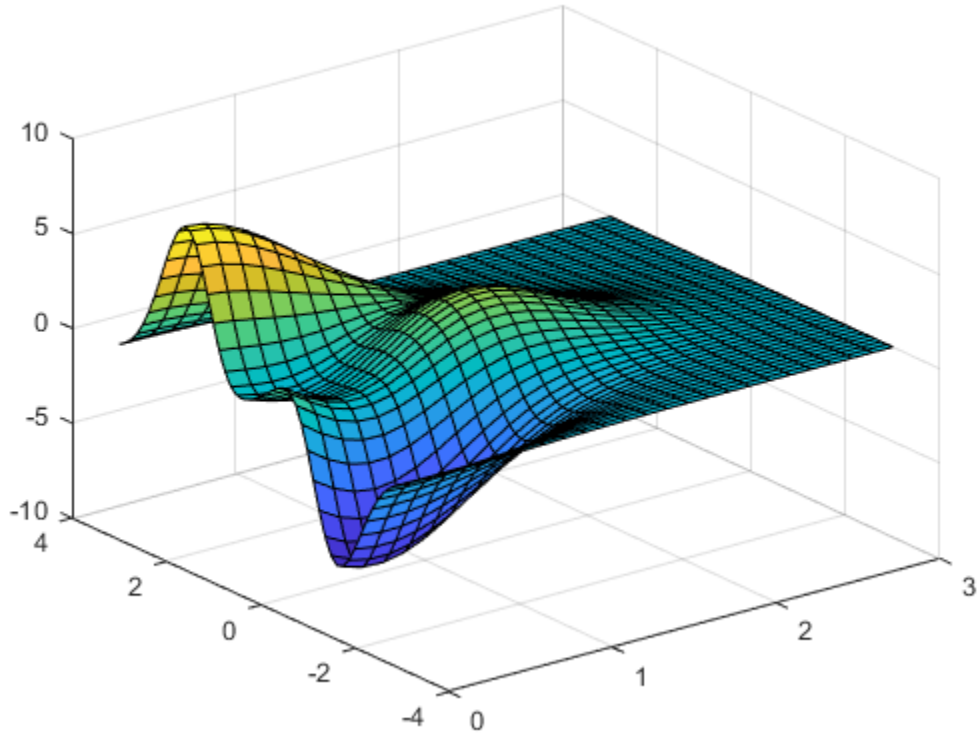
```
x = linspace(0,10);
y = sin(x);
plot(x,y)
xlim([0 5])
```



Use Semiautomatic *x*-Axis Limits

Create a surface plot and show only *x* values greater than 0. Specify the minimum *x*-axis limit as 0 and let MATLAB choose the maximum limit.

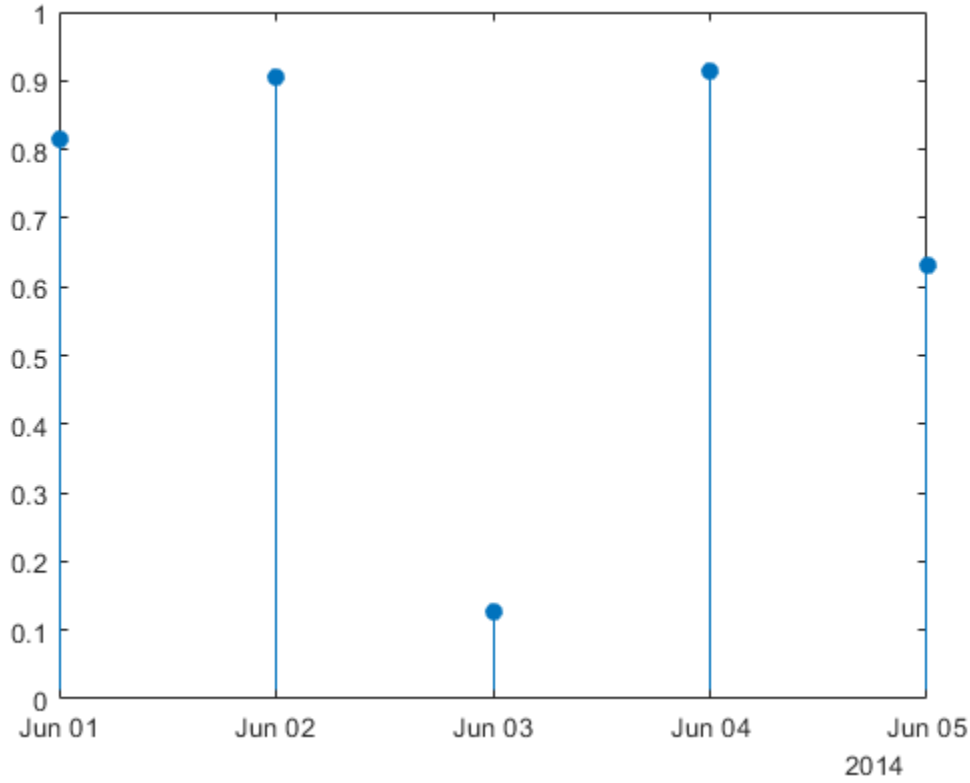
```
[X,Y,Z] = peaks;
surf(X,Y,Z)
xlim([0 inf])
```



Set Limits for *x*-Axis with Dates

Create a stem chart with dates along the *x*-axis. Set the *x*-axis limits to range from June 1, 2014 to June 5, 2014.

```
t = datetime(2014,06,1) + caldays(0:10);
y = rand(11,1);
stem(t,y,'filled')

tstart = datetime(2014,06,1);
tend = datetime(2014,06,5);
xlim([tstart tend])
```

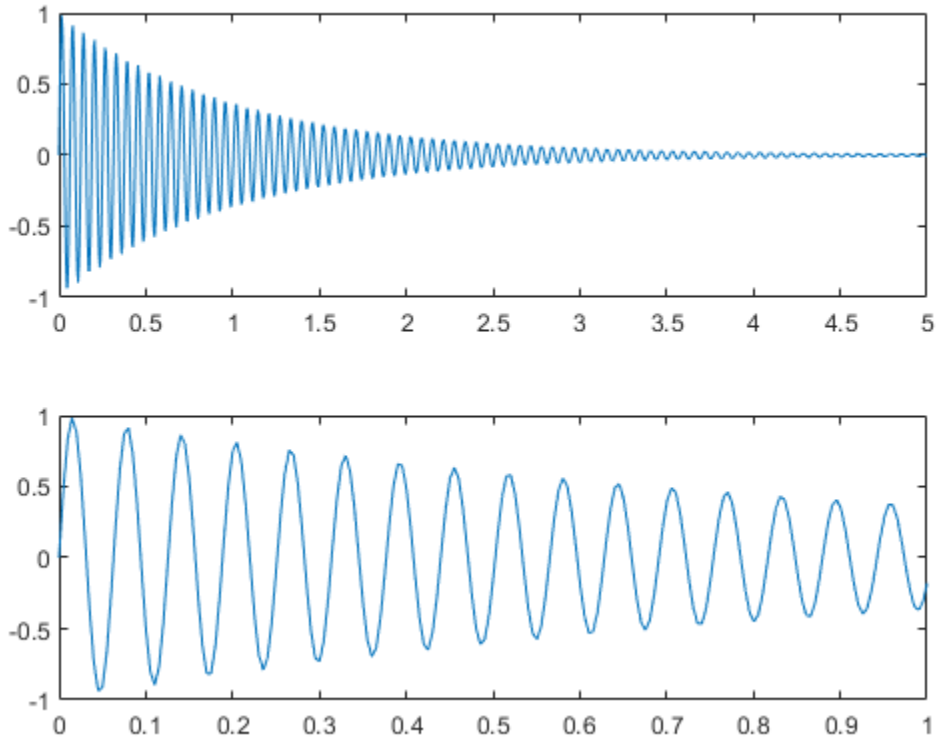


Set *x*-Axis Limits for Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Plot data into each axes. Then set the *x*-axis limits for the bottom plot by specifying `ax2` as the first input argument to `xlim`.

```
tiledlayout(2,1)
x = linspace(0,5,1000);
y = sin(100*x)./exp(x);
```

```
ax1 = nexttile;
plot(ax1,x,y)

ax2 = nexttile;
plot(ax2,x,y)
xlim(ax2,[0 1])
```

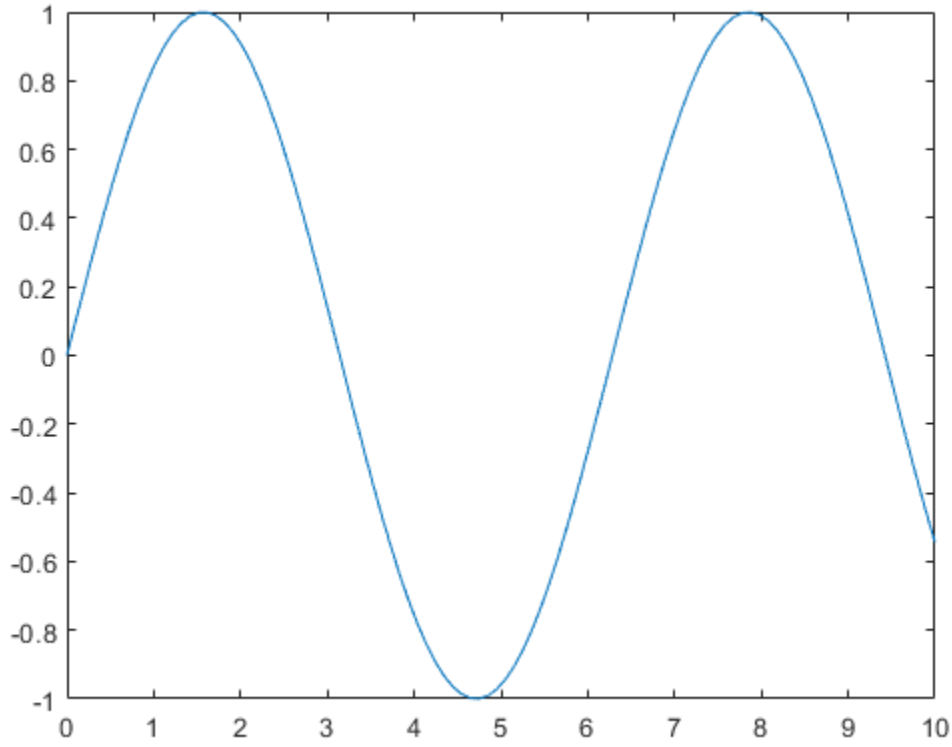


Maintain Current *x*-Axis Limits

Use manual mode to maintain the current x-axis limits when you add more plots to the axes.

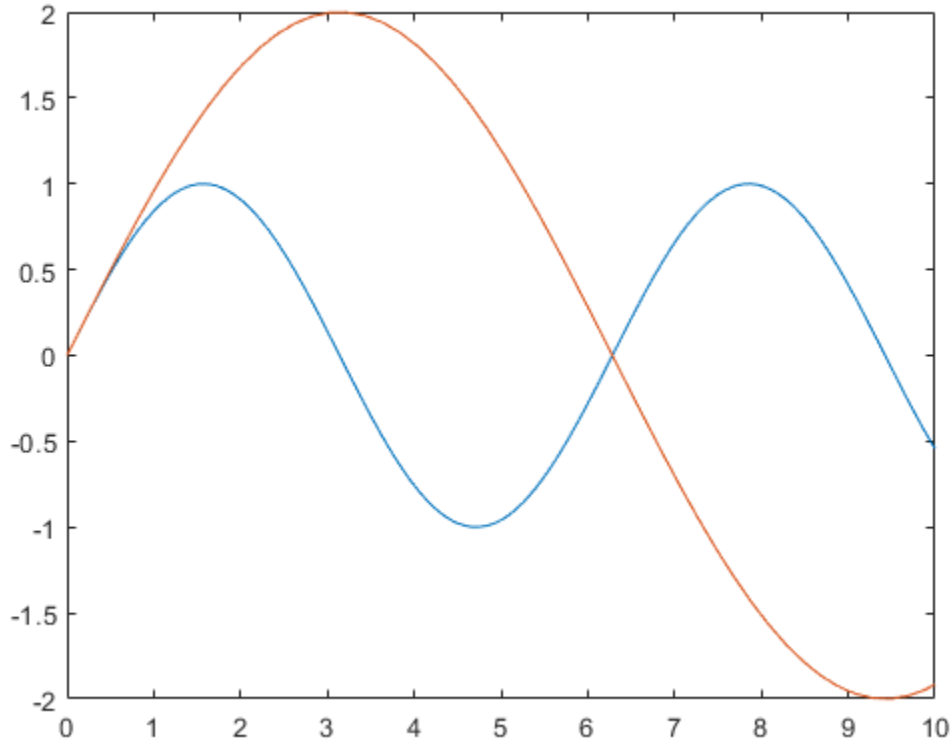
First, plot a line.

```
x = linspace(0,10);
y = sin(x);
plot(x,y);
```



Set the *x*-axis limits mode to manual so that the limits do not change. Use `hold on` to add a second plot to the axes.

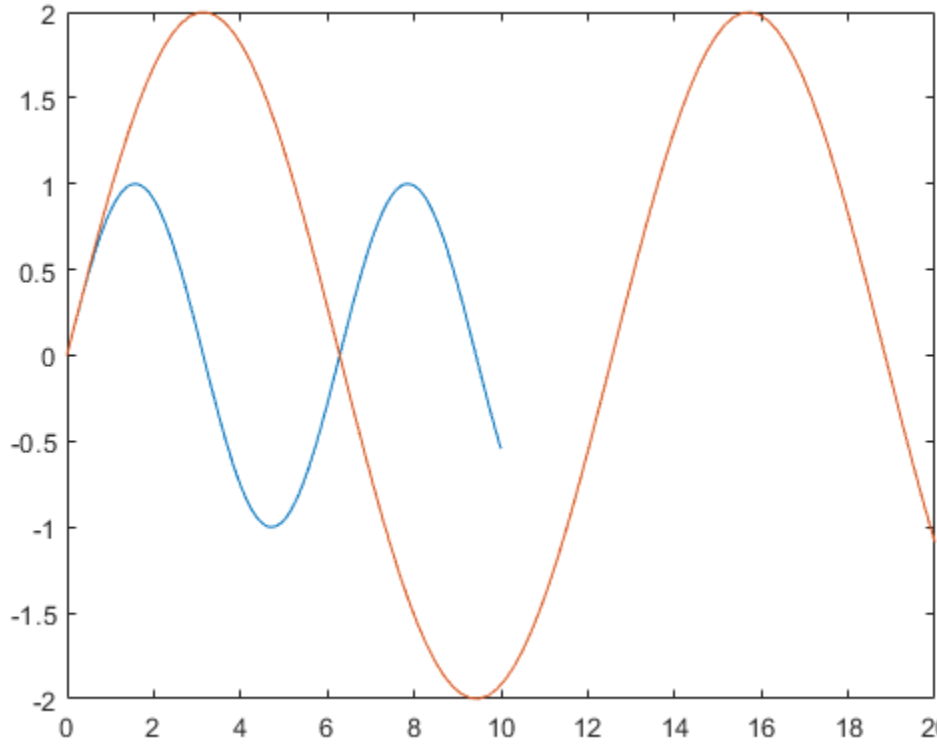
```
xlim manual
hold on
plot(2*x,2*y)
hold off
```



The *x*-axis limits do not update to incorporate the new plot.

Switch back to automatically updated limits by resetting the mode to automatic.

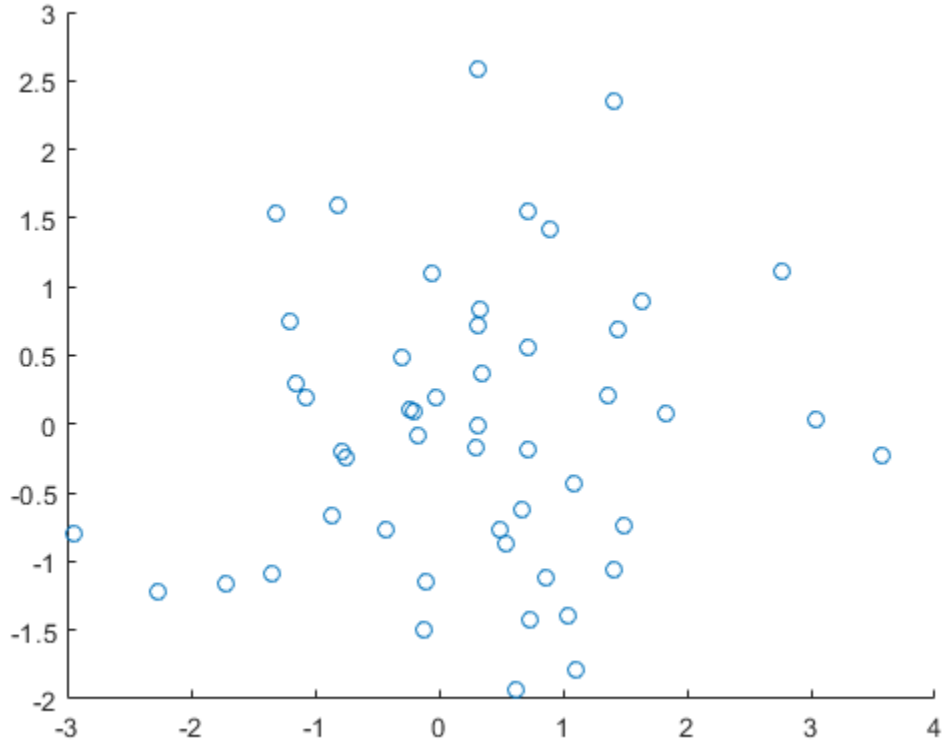
```
xlim auto
```



Return *x*-Axis Limits

Create a scatter plot of random data. Return the values of the *x*-axis limits.

```
x = randn(50,1);
y = randn(50,1);
scatter(x,y)
```




```
xl = xlim
```

```
xl = 1×2

    -3     4
```

Input Arguments

`limits` — Minimum and maximum limits
two-element vector

Minimum and maximum limits, specified as a two-element vector of the form `[xmin xmax]`, where `xmax` is greater than `xmin`. You can specify the limits as numeric, categorical, datetime, or duration values. However, the type of values that you specify must match the type of values along the *x*-axis.

You can specify both limits, or specify one limit and let MATLAB automatically calculate the other. For an automatically calculated minimum or maximum limit, use `-inf` or `inf`, respectively.

Example: `xlim([0 1])`

Example: `xlim([-inf 1])`

Example: `xlim([0 inf])`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`target` — Target axes or chart
`Axes` object | graphics object

Target axes or chart, specified as an `Axes` object or a graphics object with an `XLimits` property, such as a `HeatmapChart` object. If you do not specify the axes or chart, then `xlim` sets the limits for the current axes or chart (returned by `gca`).

Output Arguments

`xl` — Current limits
two-element vector

Current limits, returned as a two-element vector of the form `[xmin xmax]`.

Querying the limits returns the `XLim` or `XLimits` property value for corresponding `Axes` or graphics object.

`m` — Current limits mode
`'auto'` | `'manual'`

Current limits mode, returned as one of these values:

- `'auto'` — Automatically determine the limits.

- `'manual'` — Use manually specified limits that do not update to reflect changes in the data.

Querying the *x*-axis limits mode returns `XLimMode` property value for the corresponding `Axes` object.

Algorithms

The `xlim` function sets and queries several axes properties related to the *x*-axis limits.

- `XLim` — Property that stores the *x*-axis limits.
- `XLimMode` — Property that stores the *x*-axis limits mode. When you set the *x*-axis limits, this property changes to `'manual'`.

See Also

Functions
`grid` | `xlabel` | `xtickformat` | `xticklabels` | `xticks` | `ylim` | `zlim`

Properties
Axes

Topics
"Specify Axis Limits"

Introduced before R2006a

ylim

Set or query *y*-axis limits

Syntax

```
ylim(limits)
yl = ylim
```

```
ylim auto
ylim manual
m = ylim('mode')
```

```
___ = ylim(target, ___ )
```

Description

`ylim(limits)` sets the *y*-axis limits for the current axes or chart. Specify `limits` as a two-element vector of the form `[ymin ymax]`, where `ymax` is greater than `ymin`.

`yl = ylim` returns the current limits as a two-element vector.

`ylim auto` sets an automatic mode, enabling the axes to determine the *y*-axis limits. The limits span the range of the plotted data. Use this option if you change the limits and then want to set them back to the default values. This command sets the `YLimMode` property for the axes to `'auto'`.

`ylim manual` sets a manual mode, freezing the limits at the current values. Use this option if you want to retain the current limits when adding new data to the axes using the `hold on` command. This command sets the `YLimMode` property for the axes to `'manual'`.

`m = ylim('mode')` returns the current *y*-axis limits mode, which is either `'auto'` or `'manual'`. By default, the mode is automatic unless you specify limits or set the mode to manual.

`___ = ylim(target, ___ )` uses the axes or chart specified by `target` instead of the current axes. Specify `target` as the first input argument for any of the previous syntaxes.

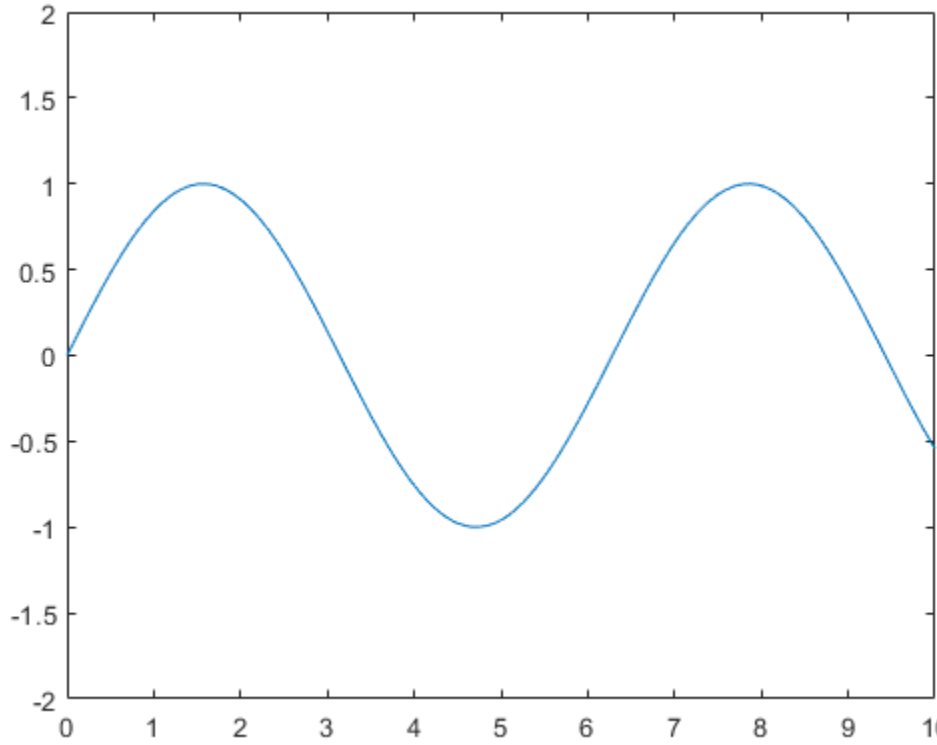
You can include an output argument if the original syntax supports an output argument. Use single quotes around the mode inputs, for example, `ylim(target,'auto')`.

Examples

Set *y*-Axis Limits

Plot a line and set the *y*-axis limits to range from -2 to 2.

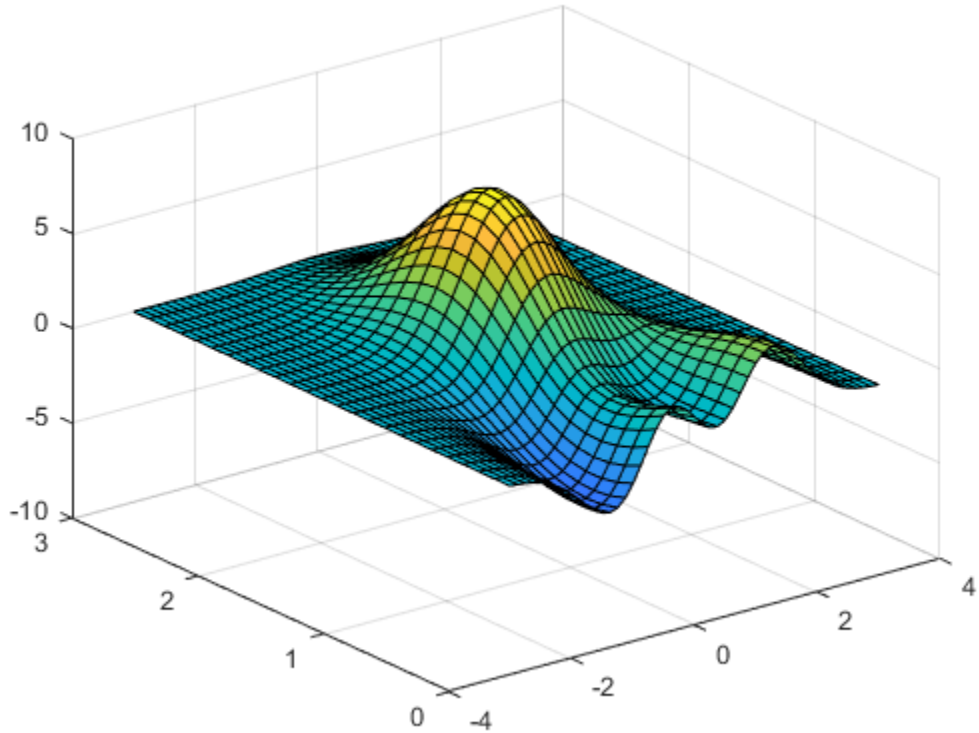
```
x = linspace(0,10);
y = sin(x);
plot(x,y)
ylim([-2 2])
```



Use Semiautomatic *y*-Axis Limits

Create a surface plot and show only *y* values greater than 0. Specify the minimum *y*-axis limit as 0 and let MATLAB choose the maximum limit.

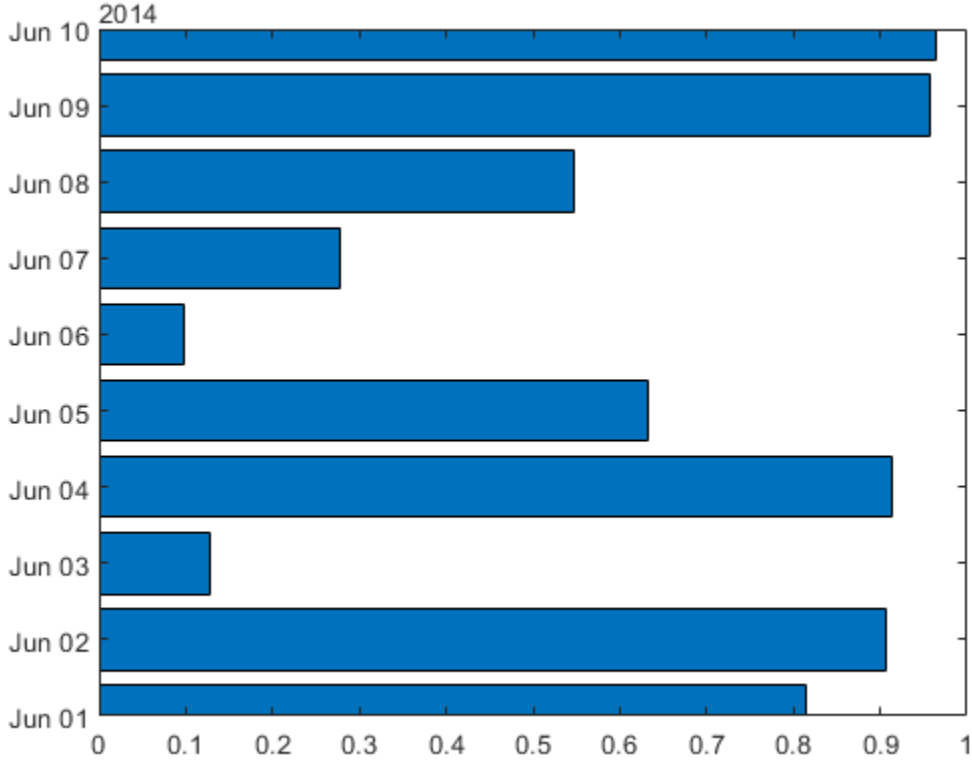
```
[X,Y,Z] = peaks;
surf(X,Y,Z)
ylim([0 inf])
```



Set Limits for *y*-Axis with Dates

Create a horizontal bar chart with dates along the y-axis. Set the y-axis limits to range from June 1, 2014 to June 10, 2014.

```
t = datetime(2014,06,1) + caldays(0:20);
y = rand(21,1);
barh(t,y)

tstart = datetime(2014,06,1);
tend = datetime(2014,06,10);
ylim([tstart tend])
```

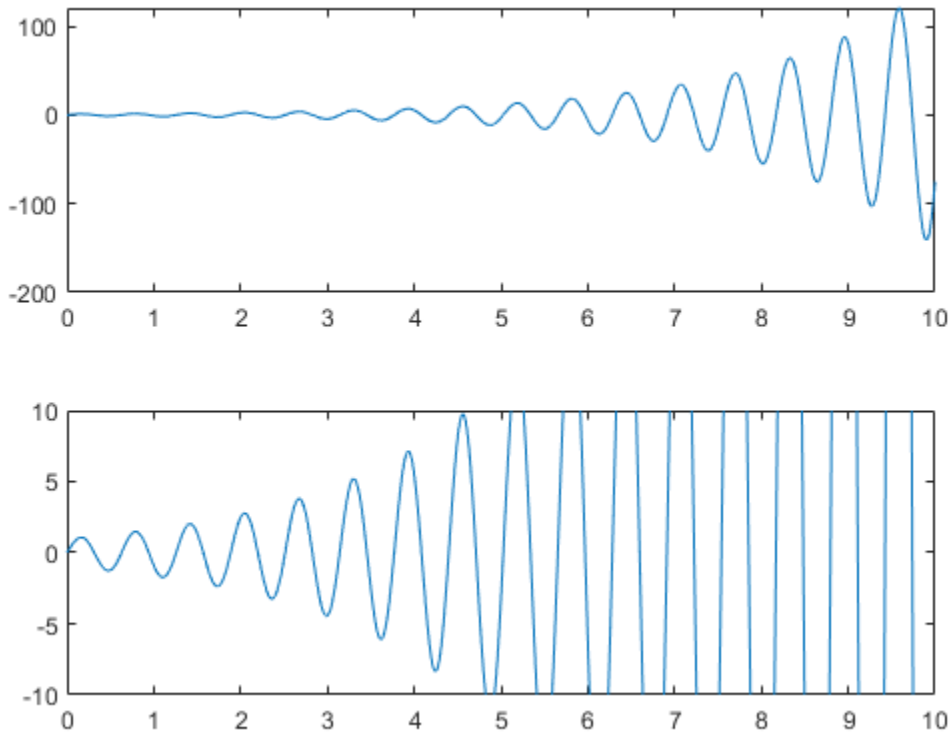


Set *y*-Axis Limits for Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Plot data into each axes. Then set the *y*-axis limits for the bottom plot by specifying `ax2` as the first input argument to `ylim`.

```
tiledlayout(2,1)
x = linspace(0,10,1000);
y = sin(10*x).*exp(.5*x);
```

```
ax1 = nexttile;
plot(ax1,x,y)

ax2 = nexttile;
plot(ax2,x,y)
ylim(ax2,[-10 10])
```

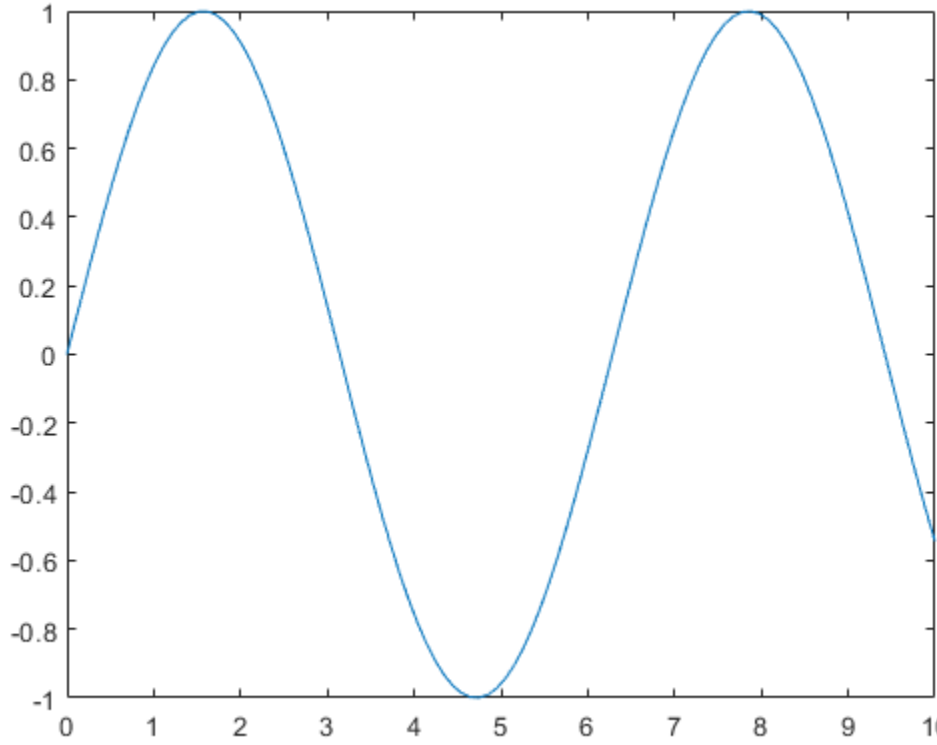


Maintain Current *y*-Axis Limits

Use manual mode to maintain the current *y*-axis limits when you add more plots to the axes.

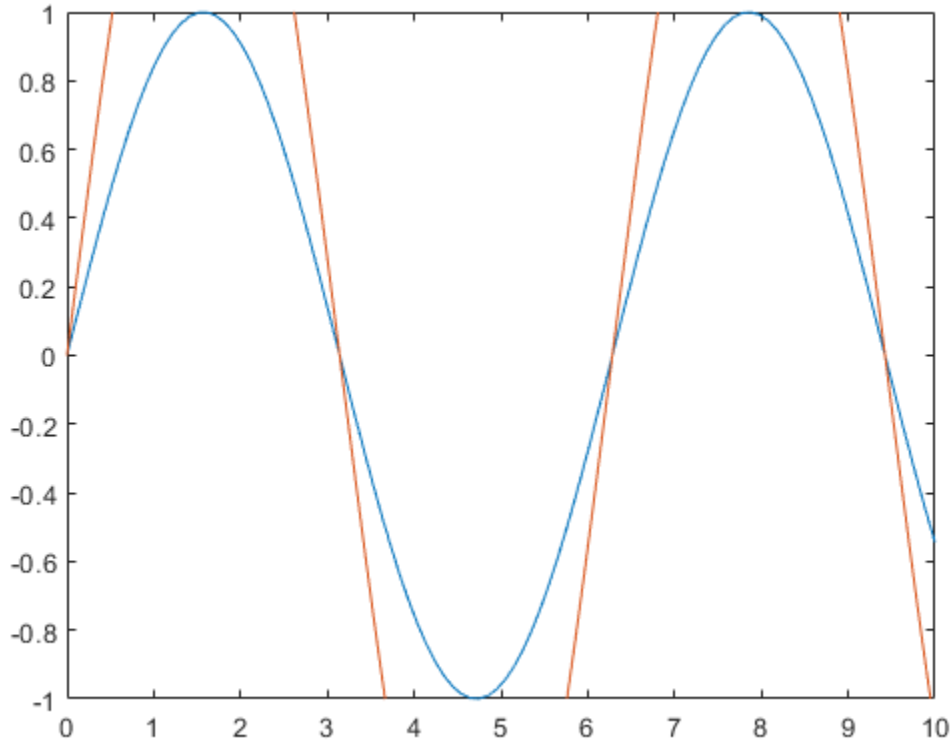
First, plot a line.

```
x = linspace(0,10);
y = sin(x);
plot(x,y)
```



Set the *y*-axis limits mode to manual so that the limits to not change. Use `hold on` to add a second plot to the axes.

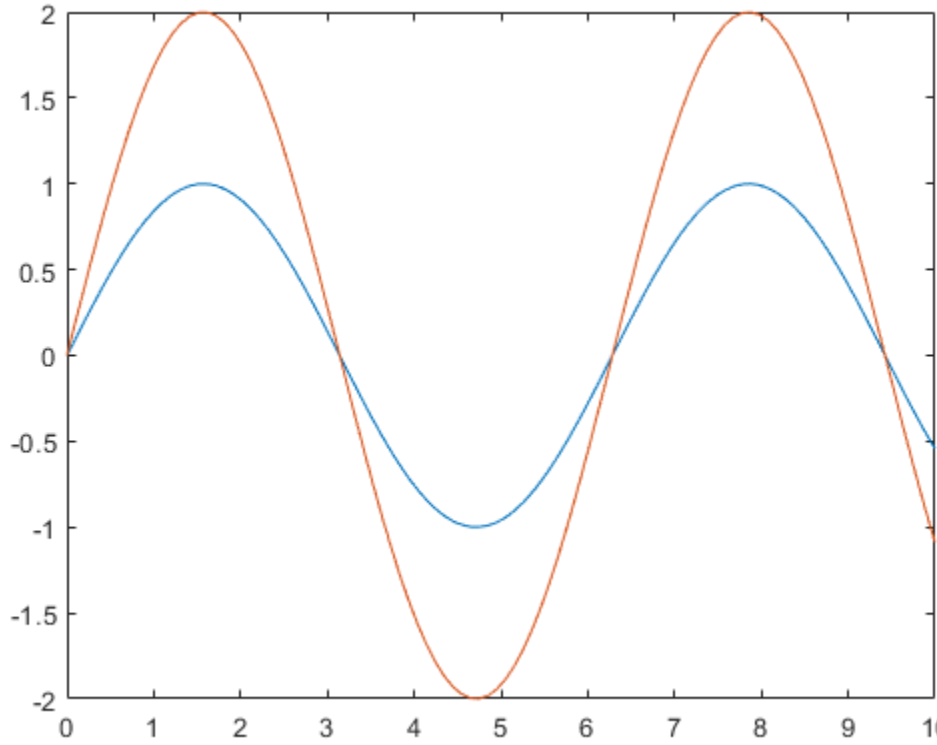
```
ylim manual
hold on
y2 = 2*sin(x);
plot(x,y2)
hold off
```



The *y*-axis limits do not update to incorporate the new plot.

Switch back to automatically updated limits by resetting the mode to automatic.

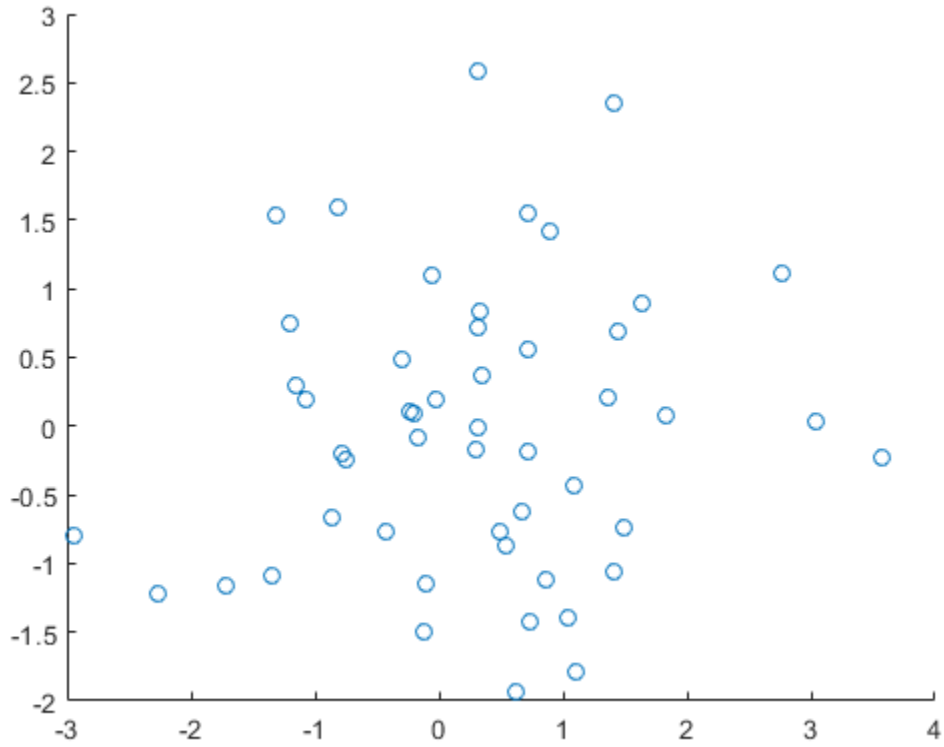
```
ylim auto
```



Return *y*-Axis Limits

Create a scatter plot of random data. Return the values of the *y*-axis limits.

```
x = randn(50,1);
y = randn(50,1);
scatter(x,y)
```




```
yl =  ylim
```

```
yl = 1×2

    -2     3
```

Input Arguments

`limits` — Minimum and maximum limits
two-element vector

Minimum and maximum limits, specified as a two-element vector of the form `[ymin ymax]`, where `ymax` is greater than `ymin`. You can specify the limits as numeric, categorical, datetime, or duration values. However, the type of values that you specify must match the type of values along the *y*-axis.

You can specify both limits, or specify one limit and let MATLAB automatically calculate the other. For an automatically calculated minimum or maximum limit, use `-inf` or `inf`, respectively.

Example: `ylim([0 1])`

Example: `ylim([-inf 1])`

Example: `ylim([0 inf])`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`target` — Target axes or chart
`Axes` object | graphics object

Target axes or chart, specified as an `Axes` object or a graphics object with an `YLimits` property, such as a `HeatmapChart` object. If you do not specify the axes or chart, then `ylim` sets the limits for the current axes or chart (returned by `gca`).

Output Arguments

`yl` — Current limits
two-element vector

Current limits, returned as a two-element vector of the form `[ymin ymax]`.

Querying the limits returns the `YLim` or `YLimits` property value for the corresponding `Axes` or graphics object.

`m` — Current limits mode
`'auto'` | `'manual'`

Current limits mode, returned as one of these values:

- `'auto'` — Automatically determine the limits.

- `'manual'` — Use manually specified limits that do not update to reflect changes in the data.

Querying the *y*-axis limits mode returns `YLimMode` property value for the corresponding `Axes` object.

Algorithms

The `ylim` function sets and queries several axes properties related to the *y*-axis limits.

- `YLim` — Property that stores the *y*-axis limits.
- `YLimMode` — Property that stores the *y*-axis limits mode. When you set the *y*-axis limits, this property changes to `'manual'`.

See Also

Functions
`grid` | `xlim` | `ylabel` | `ytickformat` | `yticklabels` | `yticks` | `zlim`

Properties
Axes

Topics
"Specify Axis Limits"

Introduced before R2006a

zlim

Set or query *z*-axis limits

Syntax

```
zlim(limits)
zl = zlim
```

```
zlim auto
zlim manual
m = zlim('mode')
```

```
___ = zlim(ax,___)
```

Description

`zlim(limits)` sets the *z*-axis limits for the current axes. Specify `limits` as a two-element vector of the form `[zmin zmax]`, where `zmax` is greater than `zmin`.

`zl = zlim` returns the current limits as a two-element vector.

`zlim auto` sets an automatic mode, enabling the axes to determine the *z*-axis limits. The limits span the range of the plotted data. Use this option if you change the limits and then want to set them back to the default values. This command sets the `ZLimMode` property for the axes to `'auto'`.

`zlim manual` sets a manual mode, freezing the limits at the current values. Use this option if you want to retain the current limits when adding new data to the axes using the `hold on` command. This command sets the `ZLimMode` property for the axes to `'manual'`.

`m = zlim('mode')` returns the current *z*-axis limits mode, which is either `'auto'` or `'manual'`. By default, the mode is automatic unless you specify limits or set the mode to manual.

`___ = zlim(ax,___)` uses the axes specified by `ax` instead of the current axes. Specify `ax` as the first input argument for any of the previous syntaxes. You can include

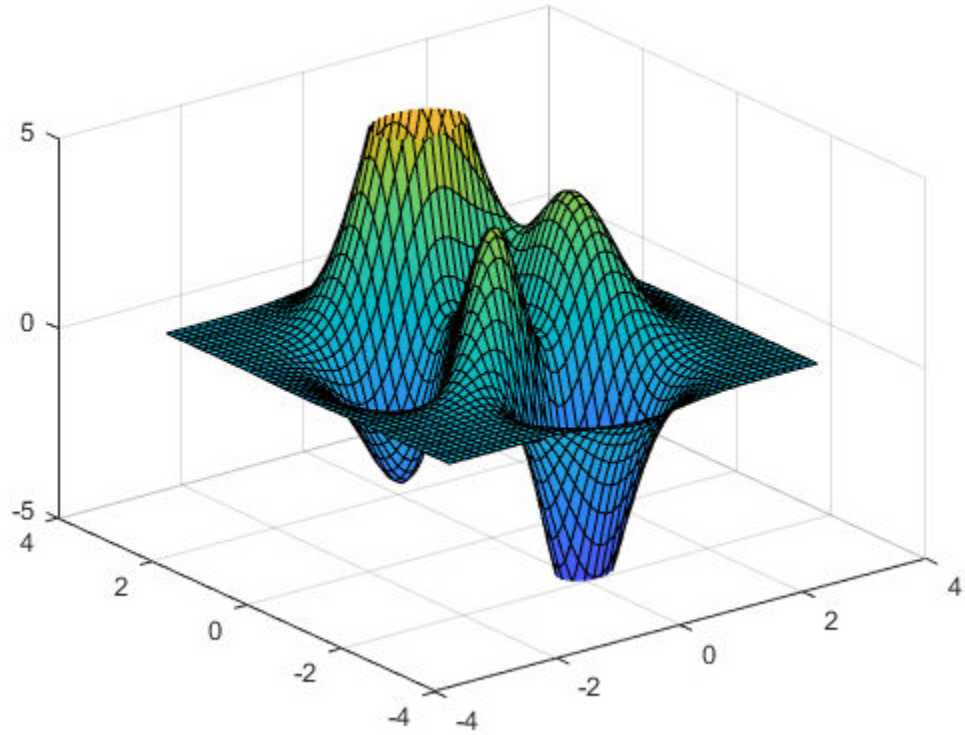
an output argument if the original syntax supports an output argument. Use single quotes around the mode inputs, for example, `zlim(ax,'auto')`.

Examples

Set *z*-Axis Limits

Plot a surface and set the *z*-axis limits to range from -5 to 5.

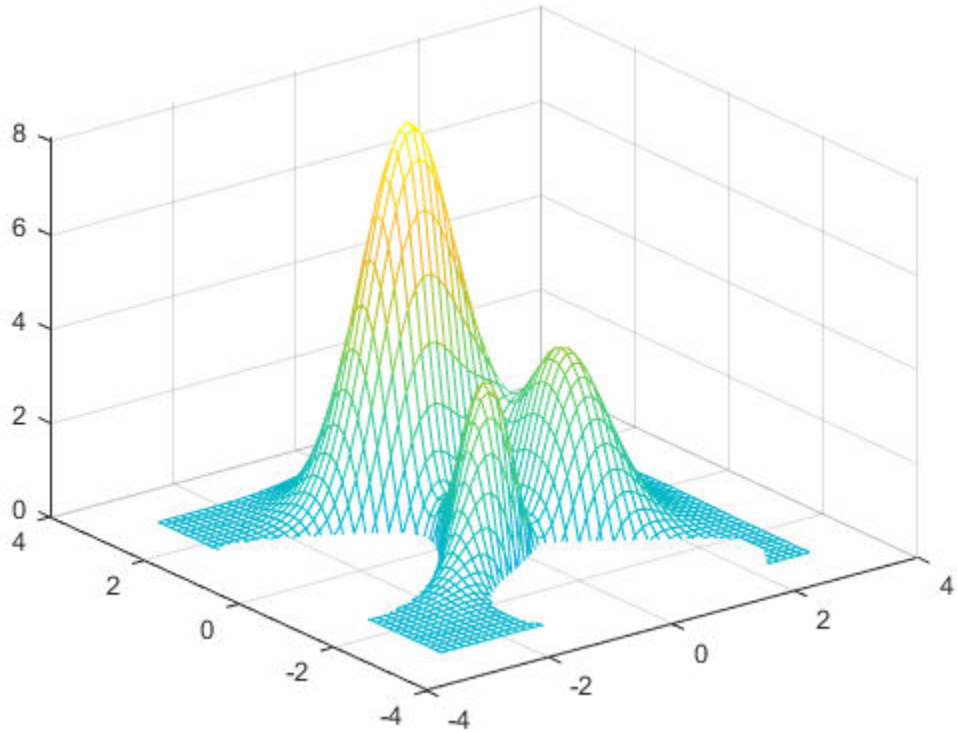
```
[X,Y,Z] = peaks;
surf(X,Y,Z);
zlim([-5 5])
```



Use Semiautomatic *z*-Axis Limits

Create a mesh surface plot and show only *z* values greater than 0. Specify the minimum *z*-axis limit as 0 and let MATLAB choose the maximum limit.

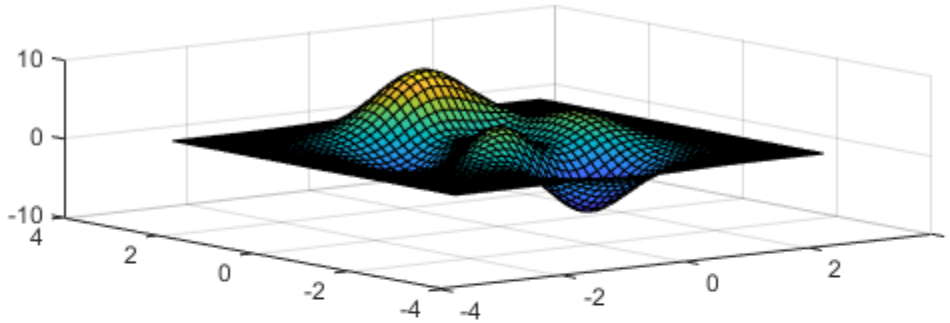
```
[X,Y,Z] = peaks;
mesh(X,Y,Z)
zlim([0 inf])
```

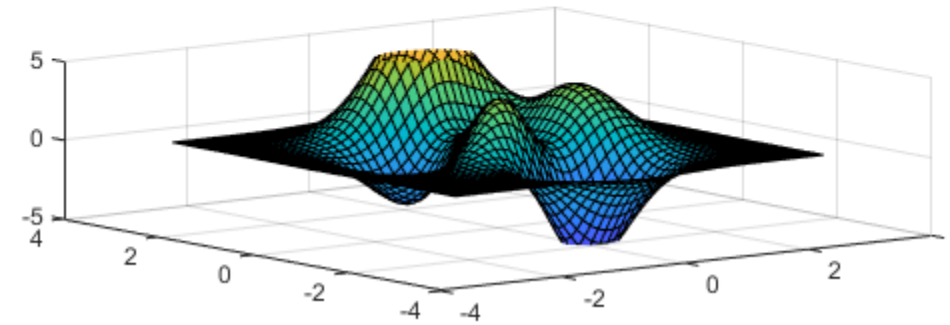


Set *z*-Axis Limits for Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Plot data into each of the axes. Then set the *z*-axis limits for the bottom plot by specifying `ax2` as the first input argument to `zlim`.

```
[X,Y,Z] = peaks;
tiledlayout(2,1)
ax1 = nexttile;
surf(X,Y,Z)

ax2 = nexttile;
surf(X,Y,Z)
zlim(ax2,[-5 5])
```



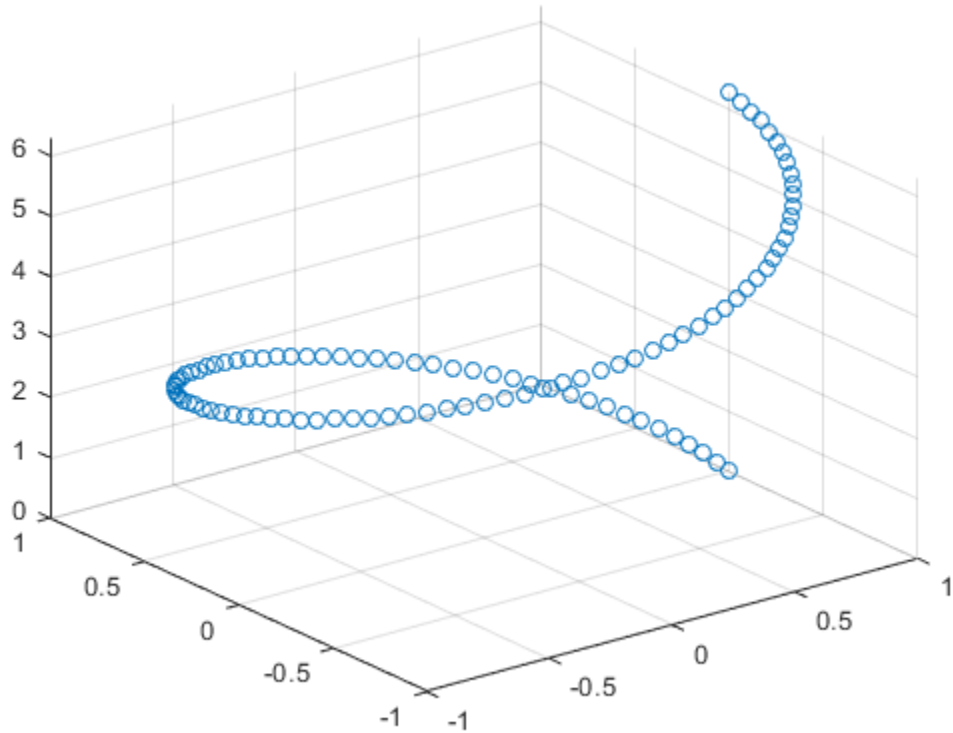


Maintain Current *z*-Axis Limits

Use manual mode to maintain the current z-axis limits when you add more plots to the axes.

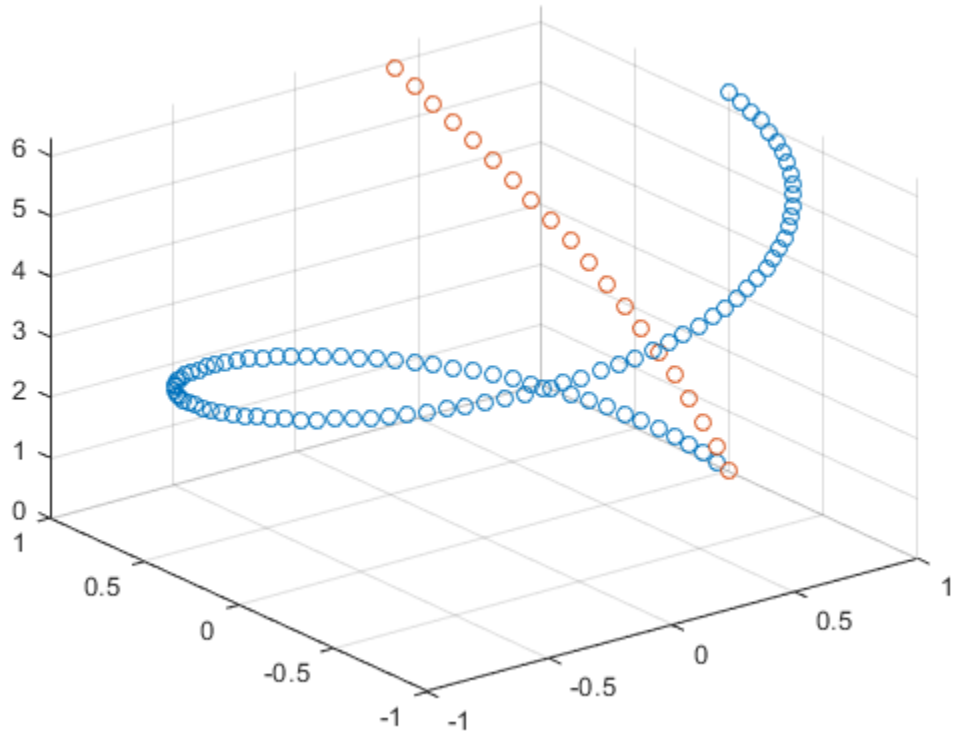
First, create a 3-D scatter plot.

```
theta = linspace(0,2*pi);
X = cos(theta);
Y = sin(theta);
Z = theta;
scatter3(X,Y,Z)
```



Set the *z*-axis limits mode to manual so that the limits to not change. Use `hold on` to add a second plot to the axes.

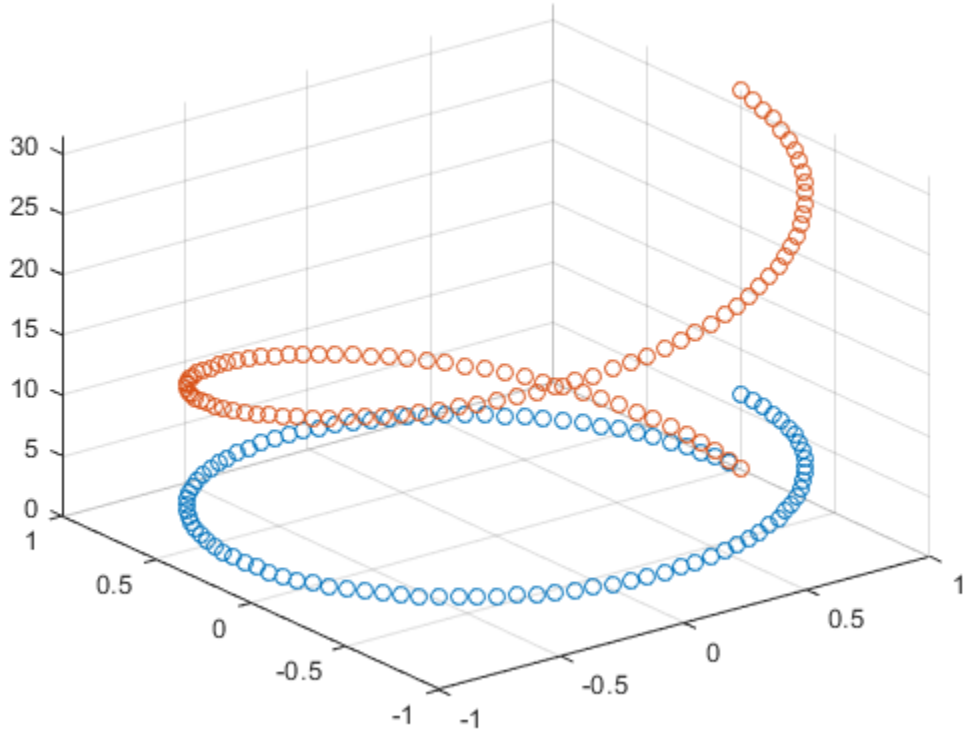
```
zlim manual
hold on
Znew = 5*theta;
scatter3(X,Y,Znew)
hold off
```



The *z*-axis limits do not update to incorporate the new plot.

Switch back to automatically updated limits by setting the mode to automatic.

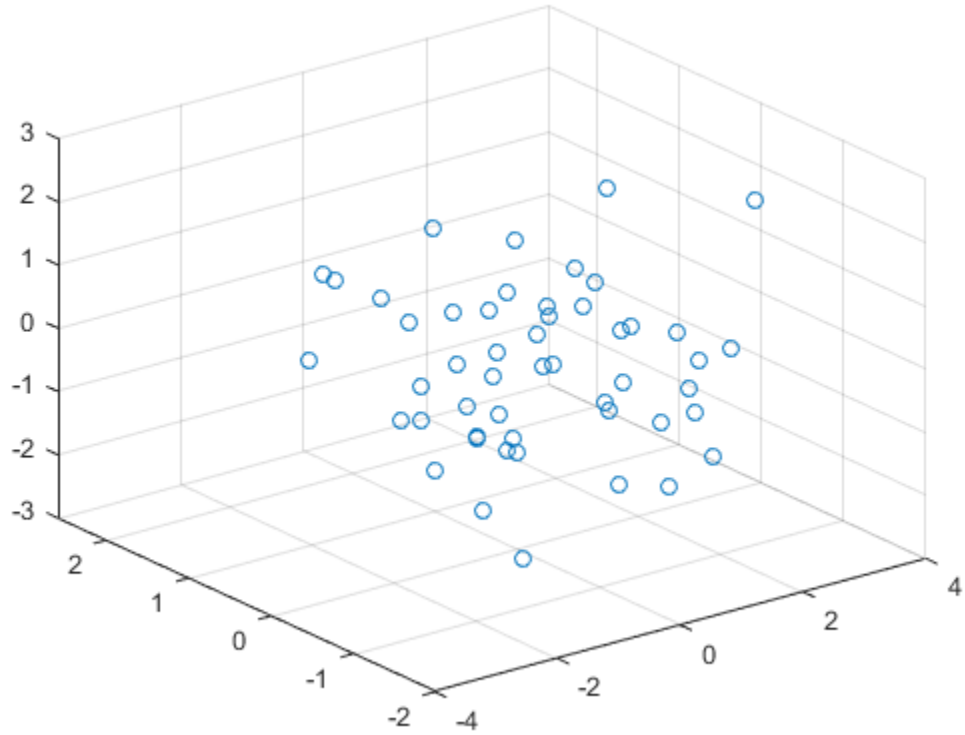
```
zlim auto
```



Return *z*-Axis Limits

Create a 3-D scatter plot of random data. Return the values of the *z*-axis limits.

```
x = randn(50,1);
y = randn(50,1);
z = randn(50,1);
scatter3(x,y,z)
```




```
zl = zlim
```

```
zl = 1×2

    -3     3
```

Input Arguments

`limits` — Minimum and maximum limits
two-element vector

Minimum and maximum limits, specified as a two-element vector of the form `[zmin zmax]`, where `zmax` is greater than `zmin`. You can specify the limits as numeric, categorical, datetime, or duration values. However, the type of values that you specify must match the type of values along the *z*-axis.

You can specify both limits, or specify one limit and let MATLAB automatically calculate the other. For an automatically calculated minimum or maximum limit, use `-inf` or `inf`, respectively.

Example: `zlim([0 1])`

Example: `zlim([-inf 1])`

Example: `zlim([0 inf])`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`ax` — `Axes` object
`Axes` object

`Axes` object. If you do not specify an `Axes` object, then `zlim` sets the limits for the current axes (`gca`).

Output Arguments

`zl` — Current limits
two-element vector

Current limits, returned as a two-element vector of the form `[zmin zmax]`.

Querying the limits returns the `ZLim` property value for the corresponding `Axes` object.

`m` — Current limits mode
`'auto'` | `'manual'`

Current limits mode, returned as one of these values:

- `'auto'` — Automatically determine the limits.
- `'manual'` — Use manually specified limits that do not update to reflect changes in the data.

Querying the *z*-axis limits mode returns `ZLimMode` property value for the corresponding `Axes` object.

Algorithms

The `zlim` function sets and queries several axes properties related to the *z*-axis limits.

- `ZLim` — Property that stores the *z*-axis limits.
- `ZLimMode` — Property that stores the *z*-axis limits mode. When you set the *z*-axis limits, this property changes to `'manual'`.

See Also

Functions
`grid` | `xlim` | `ylim` | `zlabel` | `ztickformat` | `zticklabels` | `zticks`

Properties
Axes

Topics
"Specify Axis Limits"

Introduced before R2006a

ztickangle

Rotate *z*-axis tick labels

Syntax

```
ztickangle(angle)
ztickangle(ax,angle)

ang = ztickangle
ang = ztickangle(ax)
```

Description

`ztickangle(angle)` rotates the *z*-axis tick labels for the current axes to the specified angle in degrees, where `0` is horizontal. Specify a positive value for counterclockwise rotation or a negative value for clockwise rotation.

`ztickangle(ax,angle)` rotates the tick labels for the axes specified by `ax` instead of the current axes.

`ang = ztickangle` returns the rotation angle for the *z*-axis tick labels of the current axes as a scalar value in degrees. Positive values indicate counterclockwise rotation. Negative values indicate clockwise rotation.

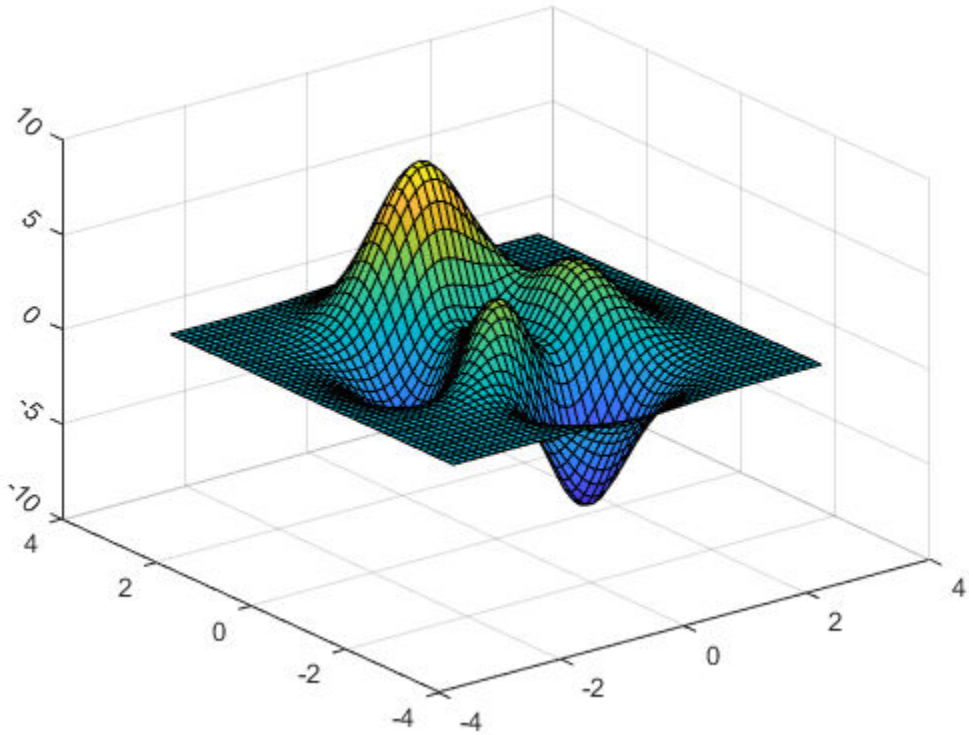
`ang = ztickangle(ax)` uses the axes specified by `ax` instead of the current axes.

Examples

Rotate z-Axis Tick Labels

Create a surface chart and rotate the *z*-axis tick labels 45 degrees clockwise.

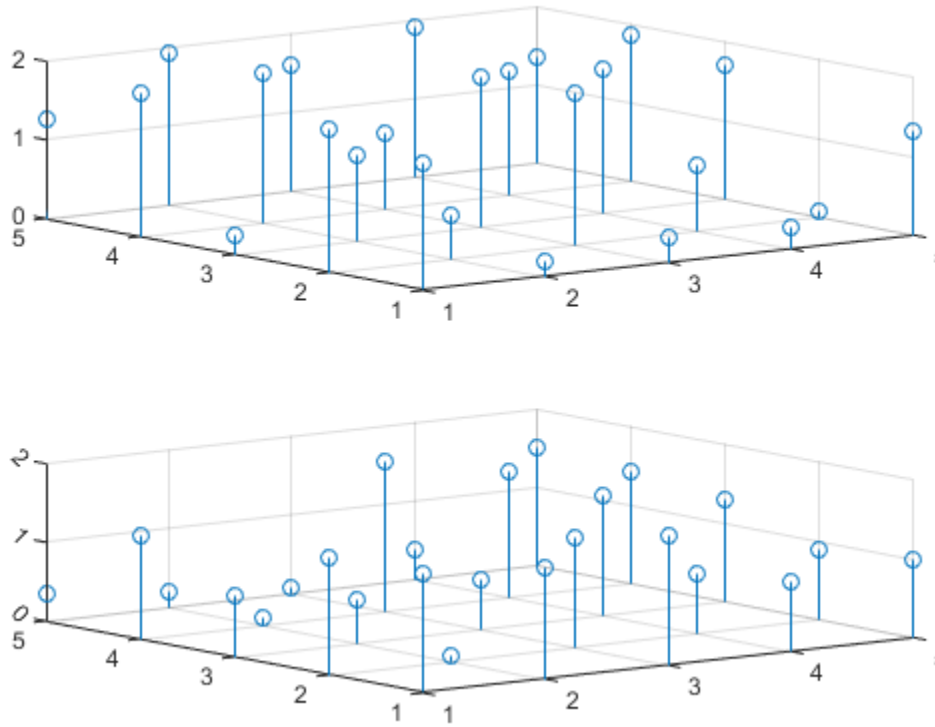
```
[x,y,z] = peaks;
surf(x,y,z)
ztickangle(-45)
```



Rotate z-Axis Tick Labels for Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Plot into each of the axes. Then rotate the *z*-axis tick labels for the lower plot by specifying `ax2` as the first input argument to `ztickangle`.

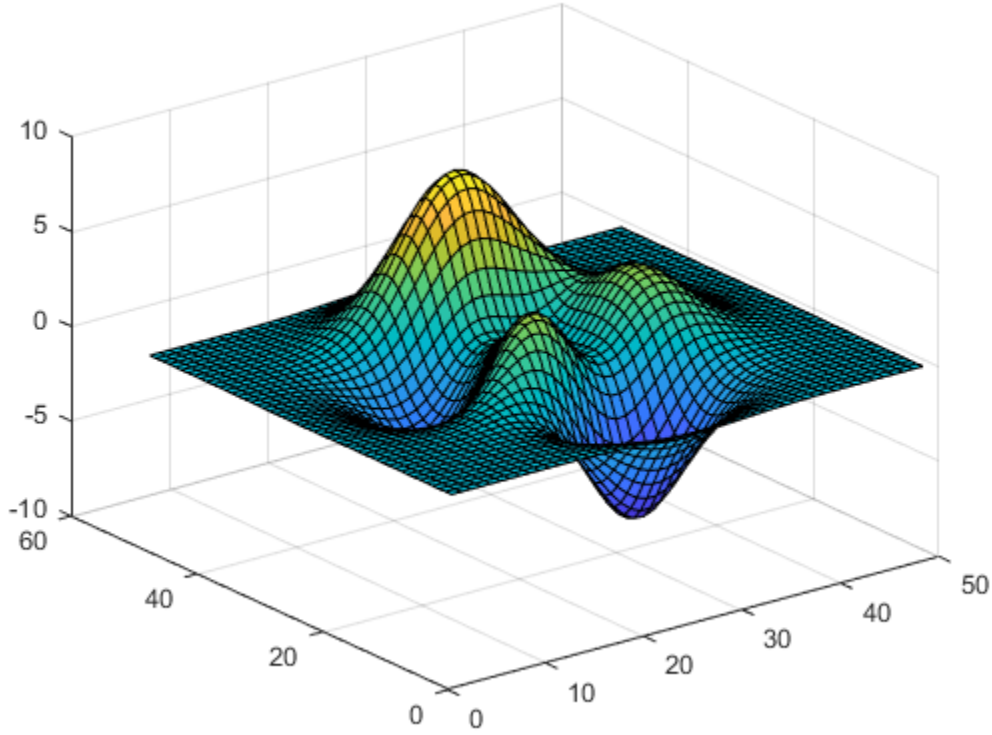
```
tiledlayout(2,1)
ax1 = nexttile;
stem3(ax1,2*rand(5))

ax2 = nexttile;
stem3(ax2,2*rand(5))
ztickangle(ax2,-45)
```



Query Current Angle for z-Axis Tick Labels

Create a surface chart. Then, query the rotation angle for the *z*-axis tick labels. By default, the labels are not rotated.

```
[x,y,z] = peaks;
surf(peaks)
```




```
ang = ztickangle
```

```
ang = 0
```

Input Arguments

angle — Rotation of tick labels
`0` (default) | scalar value in degrees

Rotation of tick labels, specified as a scalar value in degrees, where 0 is horizontal.

Example: `ztickangle(90)`

ax — Target axes
current axes (default) | single `Axes` object | vector of `Axes` objects

Target axes, specified as a single `Axes` object or a vector of `Axes` objects. If you do not specify the axes, then `ztickangle` uses the current axes.

Algorithms

The `ztickangle` function sets and queries the `ZTickLabelRotation` property of the `Axes` object.

See Also

Functions
`xtickangle` | `ytickangle` | `zlim` | `ztickformat` | `zticklabels` | `zticks`

Properties
Axes

Introduced in R2016b

ztickformat

Specify *z*-axis tick label format

Syntax

```
ztickformat(fmt)
ztickformat(datefmt)
ztickformat(durationfmt)
ztickformat(ax, ___ )

zfmt = ztickformat
zfmt = ztickformat(ax)
```

Description

`ztickformat(fmt)` sets the format for numeric *z*-axis tick labels. For example, specify `fmt` as `'usd'` to display the labels in U.S. dollars.

`ztickformat(datefmt)` sets the format for labels that show dates or times. For example, specify `datefmt` as `'MM-dd-yy'` to display dates such as `04-19-16`. This option applies only to a *z*-axis with datetime values.

`ztickformat(durationfmt)` sets the format for labels that show durations. For example, specify `durationfmt` as `'m'` to display the durations in minutes. This option applies only to a *z*-axis with duration values.

`ztickformat(ax, ___ )` uses the axes specified by `ax` instead of the current axes. Specify `ax` as the first input argument.

`zfmt = ztickformat` returns the format style used for *z*-axis tick labels of the current axes. Depending on the type of labels along the *z*-axis, `zfmt` is a character vector of a numeric format, date format, or duration format.

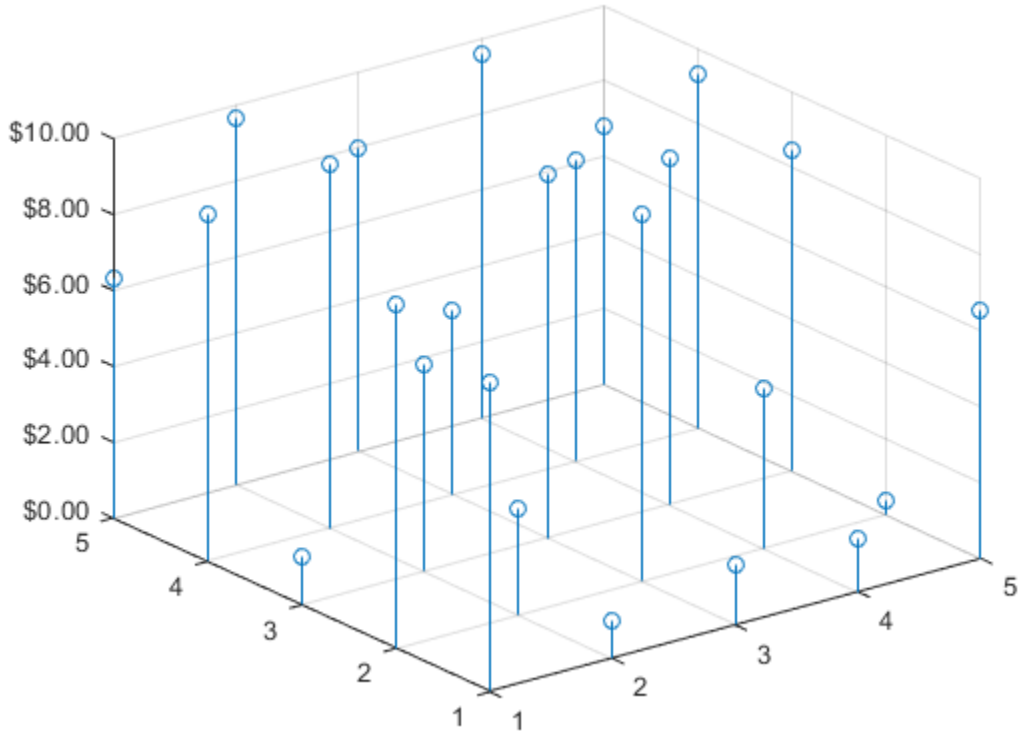
`zfmt = ztickformat(ax)` returns the format style used for the axes specified by `ax` instead of the current axes.

Examples

Display Tick Labels in Dollars

Create a 3-D stem chart of random data. Display the tick labels along the *z*-axis in US dollars.

```
z = 10*rand(5);
stem3(z)
ztickformat('usd')
```



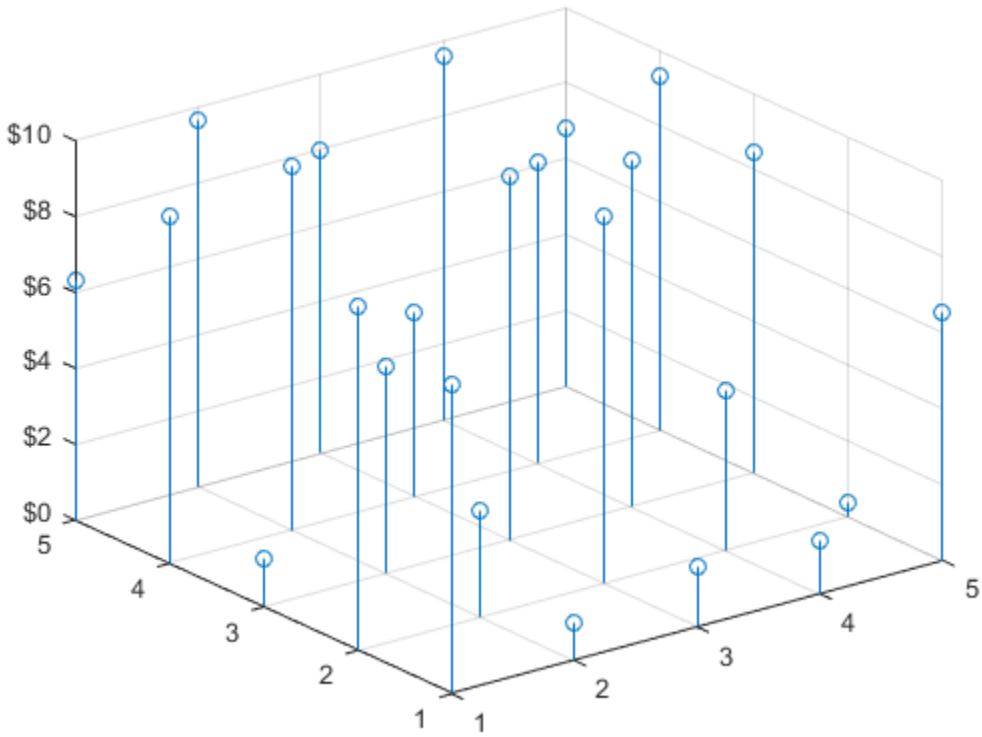
Query the tick label format. MATLAB® returns the format as a character vector containing the formatting operators.

```
fmt = ztickformat
```

```
fmt =
'$%,.2f'
```

Display the tick labels with no decimal values by tweaking the numeric format to use a precision value of 0 instead of 2.

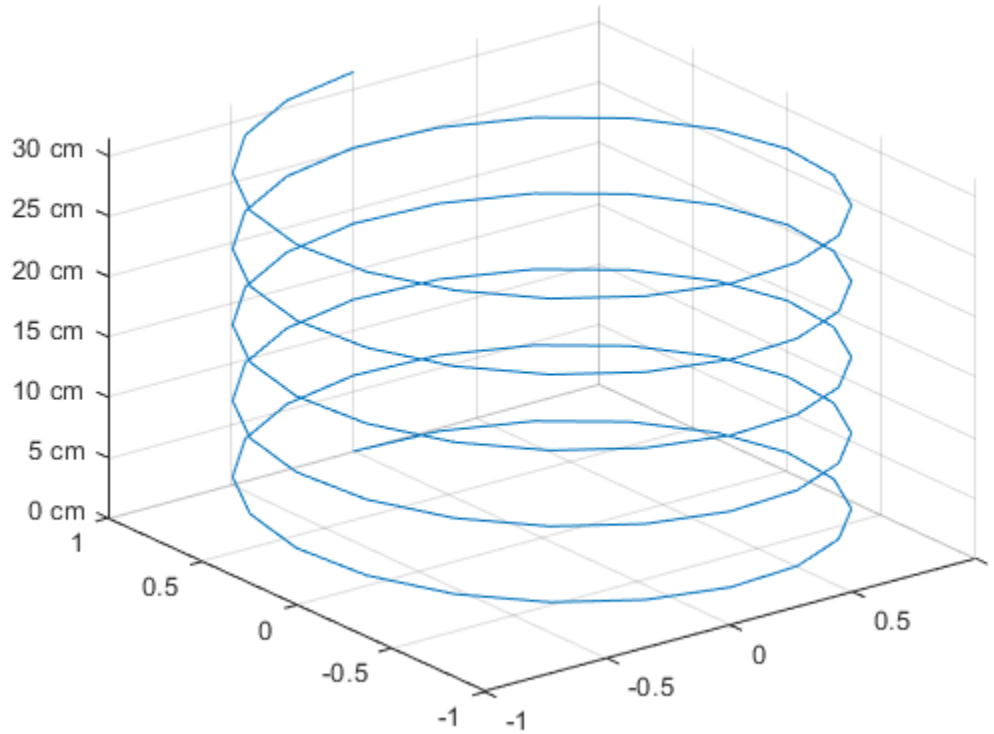
```
ztickformat('$%,.0f')
```



Display Text After Each Tick Label

Create a 3-D line chart. Display the tick labels along the *z*-axis with the text "cm" after each value.

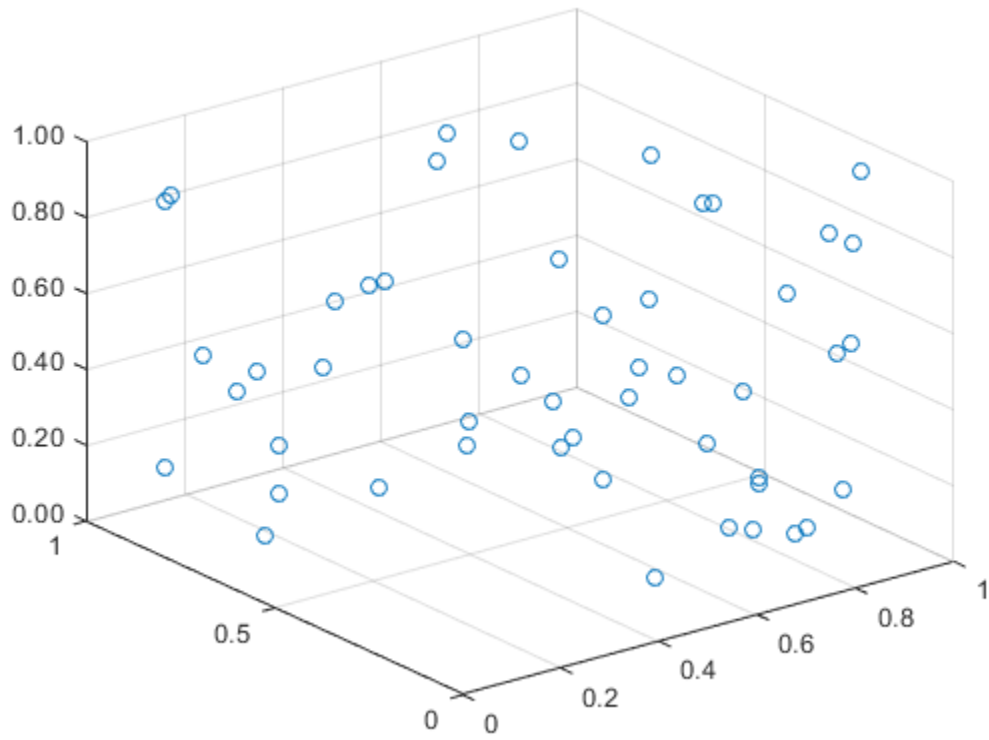
```
t = 0:pi/10:10*pi;
st = sin(t);
ct = cos(t);
plot3(st,ct,t)
grid on
ztickformat('%g cm')
```



Control Number of Decimals Displayed

Create a 3-D scatter plot of random data. Display the *z*-axis tick labels with two decimal places. Control the decimal places by passing `ztickformat` a character vector of a numeric format that uses fixed-point notation for the conversion character and a precision value of 2.

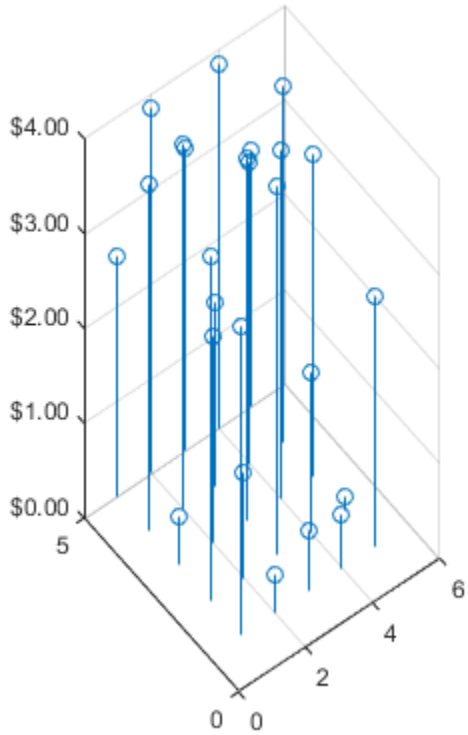
```
x = rand(50,1);
y = rand(50,1);
z = rand(50,1);
scatter3(x,y,z)
ztickformat('%.2f')
```

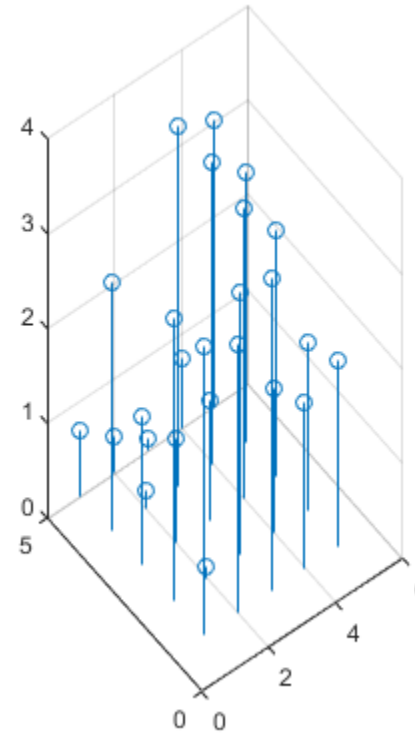


Tick Label Format for Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 1-by-2 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Create two 3-D stem plots, and specify the tick label format for the *z*-axis of the left plot by specifying `ax1` as the first input argument to `ztickformat`. Display the tick labels in U.S. dollars.

```
tiledlayout(1,2)
ax1 = nexttile;
stem3(ax1,4*rand(5))
ztickformat(ax1,'usd')

ax2 = nexttile;
stem3(ax2,4*rand(5))
```





Input Arguments

`fmt` — Format for numeric tick labels
`'%g'` (default) | character vector | string

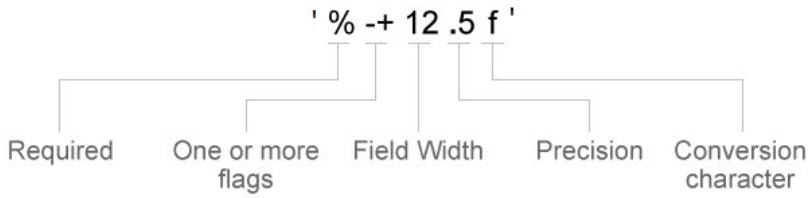
Format for numeric tick labels, specified as a character vector or string. You can specify one of the formats listed in this table. Alternatively, you can specify a custom format.

Predefined Format	Description
`'usd'`	U.S. dollars. This option is equivalent using `'$%,.2f'`. If the labels use scientific notation, this option sets the exponent value to 0.
`'eur'`	Euro. This option is equivalent to using `'\x20AC%,.2f'` with an exponent value of 0.
`'gbp'`	British pound. This option is equivalent to using `'\x00A3%,.2f'` with an exponent value of 0.
`'jpy'`	Japanese yen. This option is equivalent to using `'\x00A5%,d'` with an exponent value of 0.
`'degrees'`	Display degree symbol after values. This option is equivalent to using `'%g\x00B0'` with the default exponent value.
`'percentage'`	Display percent sign after values. This option is equivalent to using `'%g%%'` with the default exponent value.
`'auto'`	Default format of `'%g'` with the default exponent value.

Example: `ztickformat('usd')`

Custom Numeric Format

You can specify a custom numeric format by creating a character vector or string containing identifiers.



Identifiers are optional, except the percent sign and conversion character. Construct the format in this order:

- One or more flags — Options such as adding a plus sign before positive values. For a full list of options, see the table of Optional Flags.
- Field width — Minimum number of characters to print in the tick label. Specify the field width as an integer value. If the number of significant digits in the tick value is smaller than the field width, then the label is padded with spaces.
- Precision — Number of digits to the right of the decimal point or the number of significant digits, depending on the conversion character. Specify the precision as an integer value.
- Conversion character — Value type. For a full list of options, see the table of Conversion Characters. If you specify a conversion that does not fit the data, then MATLAB overrides the specified conversion, and uses `%e`.

Also, you can specify literal text at the beginning or end of the format. To print a single quotation mark, use `''`. To print a percent character, use `%%`.

Example: `ztickformat('%.2f')` displays the values using fixed-point notation with two decimal places.

Example: `ztickformat('$%.2f')` displays a dollar sign before each value.

Example: `ztickformat('%.2f million')` displays `million` after each value.

Optional Flags

Identifier	Description	Example of Numeric Format
`,`	Display commas every three digits, such as `'1,000'`.	`'%,4.4g'`
`+`	Print the sign character (`+`) for positive values, such as `'+100'`.	`'%+4.4g'`
`0`	Pad the field width with leading zeros instead of spaces, such as `'0100'`.	`'%04.4g'`
`-`	Left-justify, which pads the end of the value with spaces instead of the beginning. For example, if the field width is `4`, then this flag formats the label as `'100 '` instead of `' 100'`.	`'%-4.4g'`
`#`	For the `%f`, `%e`, and `%g` conversion characters, print the decimal point even when the precision is 0, such as `'100.'`. For `%g`, do not remove trailing zeros.	`'%#4.4g'`

Conversion Characters

Identifier	Description	Example
`d` or `i`	Signed integer with base 10. The precision value indicates the number of significant digits.	`'%.4d'` displays π as `0003`.
`f`	Fixed-point notation. The precision value indicates the number of decimal places.	`'%.4f'` displays π as `3.1416`.
`e`	Exponential notation. The precision value indicates the number of decimal places.	`'%.4e'` displays π as `3.1416x10`^{0}.
`g`	The more compact version of `e` or `f`, with no trailing zeros. The precision value indicates the maximum number of decimal places.	`'%.4g'` displays π as `3.1416`.

`datefmt` — Format for dates and times
`'auto'` | character vector | string

Format for dates and times, specified as `'auto'`, a character vector, or a string. The default format is based on the data.

Example: `ztickformat('yyyy-MM-dd')` displays a date and time such as `2016-04-19`.

Example: `ztickformat('eeee, MMMM d, yyyy HH:mm:ss')` displays a date and time such as `Saturday, April 19, 2016 21:41:06`.

Example: `ztickformat('MMMM d, yyyy HH:mm:ss Z')` displays a date and time such as `April 19, 2016 21:41:06 -0400`.

The display format tables show the letter identifiers that you can use to construct the format. To separate the fields, you can include nonletter characters such as a hyphen, space, colon, or any non-ASCII characters. The identifiers correspond to the Unicode Locale Data Markup Language (LDML) standard for dates.

Date and Time Formats

Use these identifiers to specify the display formats of the date and time fields.

Letter Identifier	Description	Display
G	Era	CE
y	Year, with no leading zeros. See the Note that follows this table.	2014
yy	Year, using last two digits. See the Note that follows this table.	14
yyy, yyyy ...	Year, using at least the number of digits specified by the number of instances of 'y'	For the year 2014, 'yyy' displays 2014, while 'yyyyy' displays 02014.
u, uu, ...	ISO year. A single number designating the year. An ISO year value assigns positive values to CE years and negative values to BCE years, with 1 BCE being year 0.	2014
Q	Quarter, using one digit	2
QQ	Quarter, using two digits	02
QQQ	Quarter, abbreviated	Q2
QQQQ	Quarter, full name	2nd quarter
M	Month, numerical using one or two digits	4
MM	Month, numerical using two digits	04
MMM	Month, abbreviated name	Apr
MMMM	Month, full name	April
MMMMM	Month, capitalized first letter	A
W	Week of the month	1
d	Day of the month, using one or two digits	5
dd	Day of the month using two digits	05
D	Day of the year, using one, two or three digits	95
DD	Day of the year using two digits	95

Letter Identifier	Description	Display
DDD	Day of the year using three digits	095
e	Day of the week, numerical using one or two digits.	7, where Sunday is the first day of the week.
ee	Day of the week, numerical using two digits	07
eee	Day, abbreviated name	Sat
eeee	Day, full name	Saturday
eeeee	Day, capitalized first letter	S
a	Day period (AM or PM)	PM
h	Hour, 12-hour clock notation using one or two digits	9
hh	Hour, 12-hour clock notation using two digits	09
H	Hour, 24-hour clock notation using one or two digits	21
HH	Hour, 24-hour clock notation using two digits	21
m	Minute, using one or two digits	41
mm	Minute, using two digits	41
s	Second, using one or two digits	6
ss	Second, using two digits	06
S, SS, ..., SSSSSSSSS	Fractional second, using the number of digits specified by the number of instances of 'S' (up to 9 digits).	'SSS' truncates 6.12345 seconds to 123.

Some tips and considerations:

- Use one or more u characters instead of y characters to represent the year when working with year numbers near zero.
- Datetime values later than 144683 years CE or before 140743 BCE display only the year numbers, regardless of the specified format value.

Time Zone Offset Formats

Use these identifiers to specify the display format of the time zone offset. A time zone offset is the amount of time that a specific date and time is offset from UTC. A time zone offset is different from a time zone in that it comprises rules that determine the offsets for specific times of the year. Include a time zone offset identifier when you want to ensure that the time components are displayed unambiguously.

Letter Identifier	Description	Display
`z`	Abbreviated name of the time zone offset. If this value is not available, then the time zone offset uses the short UTC format, such as `UTC-4`.	`EDT`
`Z`	ISO 8601 basic format with hours, minutes, and optional seconds fields.	`-0400`
`ZZZZ`	Long UTC format.	`UTC-04:00`
`ZZZZZ`	ISO 8601 extended format with hours, minutes, and optional seconds fields. A time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	`-04:00`
`x` or `X`	ISO 8601 basic format with hours field and optional minutes field. If you specify `X`, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	`-04`
`xx` or `XX`	ISO 8601 basic format with hours and minutes fields. If you specify `XX`, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	`-0400`

Letter Identifier	Description	Display
xxx or XXX	ISO 8601 extended format with hours and minutes fields. If you specify XXX, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-04:00
xxxx or XXXX	ISO 8601 basic format with hours, minutes, and optional seconds fields. If you specify XXXX, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-0400
xxxxx or XXXXX	ISO 8601 extended format with hours, minutes, and optional seconds fields. If you specify XXXXX, a time offset of zero is displayed as the ISO 8601 UTC indicator “Z”.	-04:00

`durationfmt` — Format for duration values

character vector | string

Format for duration values, specified as a character vector or string.

To display a duration as a single number that includes a fractional part (for example, 1.234 hours), specify one of the following formats.

Format	Description
'y'	Number of exact, fixed-length years. A fixed-length year is equal to 365.2425 days.
'd'	Number of exact, fixed-length days. A fixed-length day is equal to 24 hours.
'h'	Number of hours
'm'	Number of minutes
's'	Number of seconds

To display a duration in the form of a digital timer, specify one of these formats:

- `'dd:hh:mm:ss'`
- `'hh:mm:ss'`
- `'mm:ss'`
- `'hh:mm'`

In addition, you can display up to nine fractional second digits by appending up to nine `S` characters. For example, `'hh:mm:ss.SSS'` displays the milliseconds of a duration value to three digits.

ax — Target axes
current axes (default) | single `Axes` object | vector of `Axes` objects

Target axes, specified as a single `Axes` object or a vector of `Axes` objects. If you do not specify the axes, then `ztickformat` uses the current axes.

Algorithms

The `ztickformat` function sets and queries the `TickLabelFormat` property of the ruler object associated with the *z*-axis.

See Also

Functions
`xtickformat` | `ytickformat` | `zlim` | `ztickangle` | `zticklabels` | `zticks`

Properties
DatetimeRuler | DurationRuler | NumericRuler

Introduced in R2016b

zticklabels

Set or query *z*-axis tick labels

Syntax

```
zticklabels(labels)
zl = zticklabels

zticklabels('auto')
zticklabels('manual')
m = zticklabels('mode')

___ = zticklabels(ax,___)
```

Description

`zticklabels(labels)` sets the *z*-axis tick labels on page 1-17347 for the current axes. Specify `labels` as a string array or a cell array of character vectors; for example, `{'January','February','March'}`. If you specify the labels, then the *z*-axis tick values and tick labels no longer update automatically based on changes to the axes.

`zl = zticklabels` returns the *z*-axis tick labels for the current axes.

`zticklabels('auto')` sets an automatic mode, enabling the axes to determine the *z*-axis tick labels. Use this option if you set the labels and then want to set them back to the default values.

`zticklabels('manual')` sets a manual mode, freezing the *z*-axis tick labels at the current values.

`m = zticklabels('mode')` returns the current value of the *z*-axis tick labels mode, which is either `'auto'` or `'manual'`. By default, the mode is automatic unless you specify the tick labels or set the mode to manual.

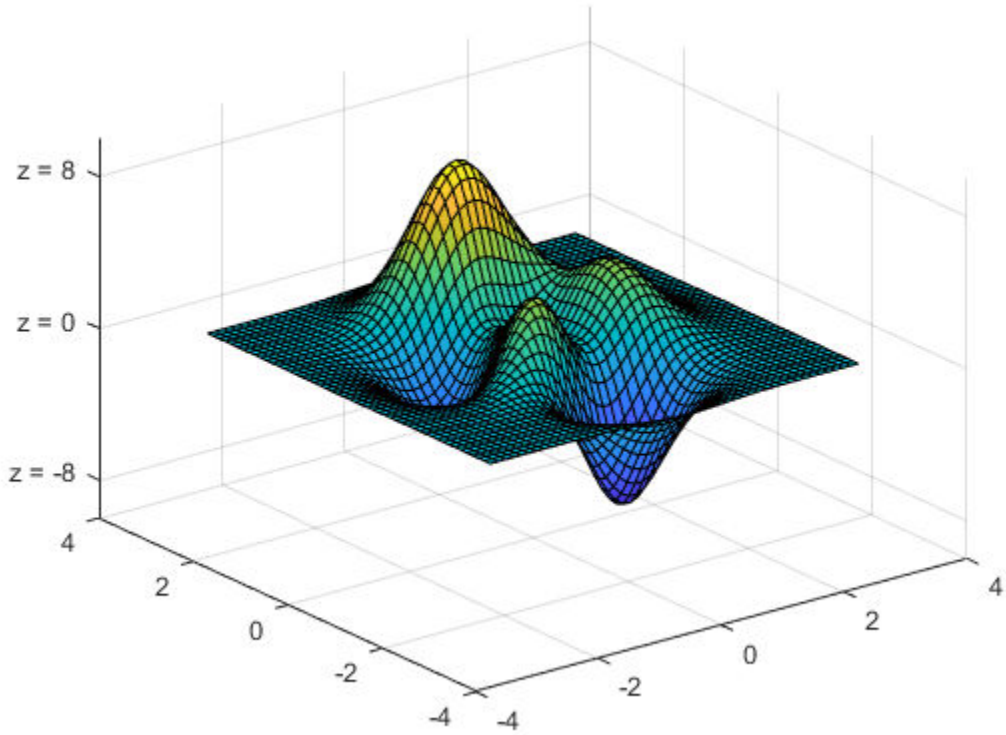
`___ = zticklabels(ax,___)` uses the axes specified by `ax` instead of the current axes. Specify `ax` as the first input argument for any of the previous syntaxes.

Examples

Specify z-Axis Tick Values and Labels

Create a surface plot. Display tick marks along the *z*-axis at the values -8, 0, and 8. Then, specify a label for each tick mark.

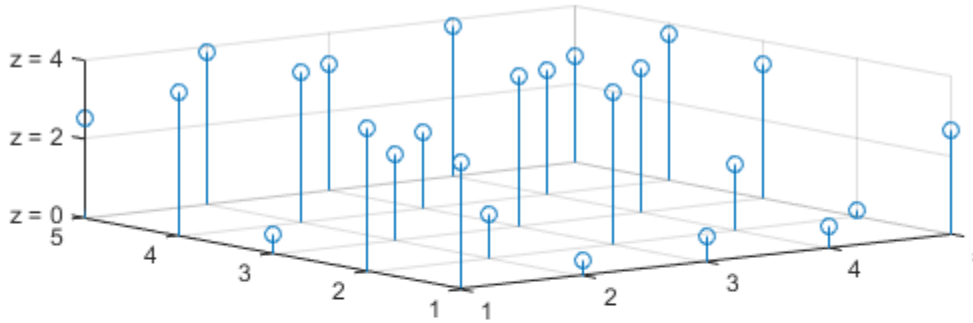
```
[x,y,z] = peaks;
surf(x,y,z)
zticks([-8 0 8])
zticklabels({'z = -8','z = 0','z = 8'})
```

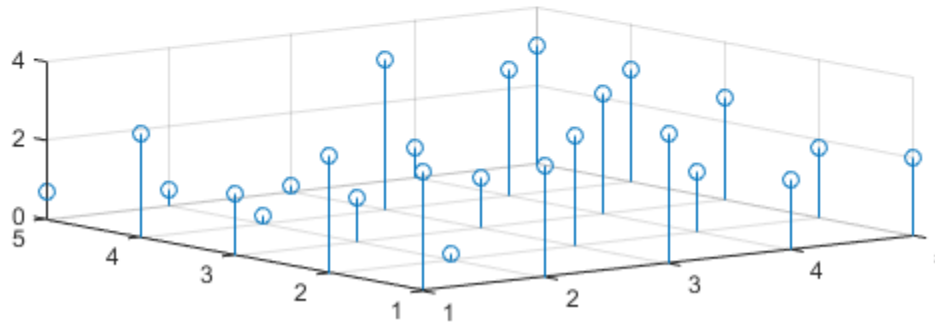


Specify z-Axis Tick Labels for Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Create two 3-D stem plots. Set the *z*-axis tick values and labels for the first plot by passing `ax1` as the first input argument to the `zticks` and `zticklabels` functions.

```
tiledlayout(2,1)
ax1 = nexttile;
stem3(ax1,4*rand(5))
zticks(ax1,[0 2 4])
zticklabels(ax1,{'z = 0','z = 2','z = 4'})

ax2 = nexttile;
stem3(ax2,4*rand(5))
```

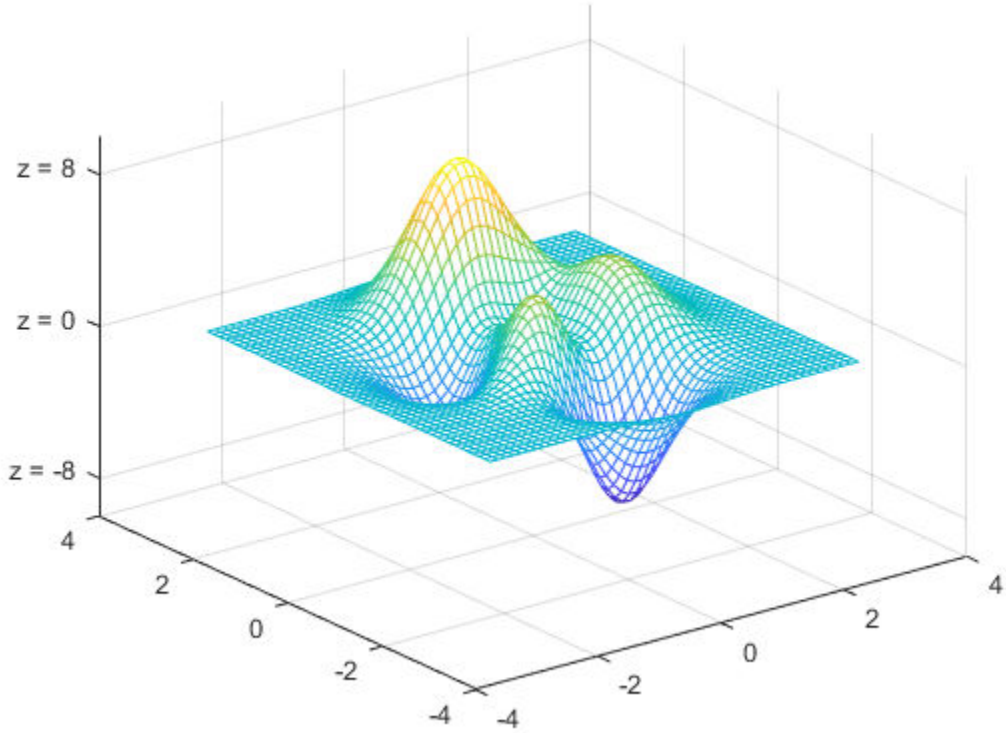




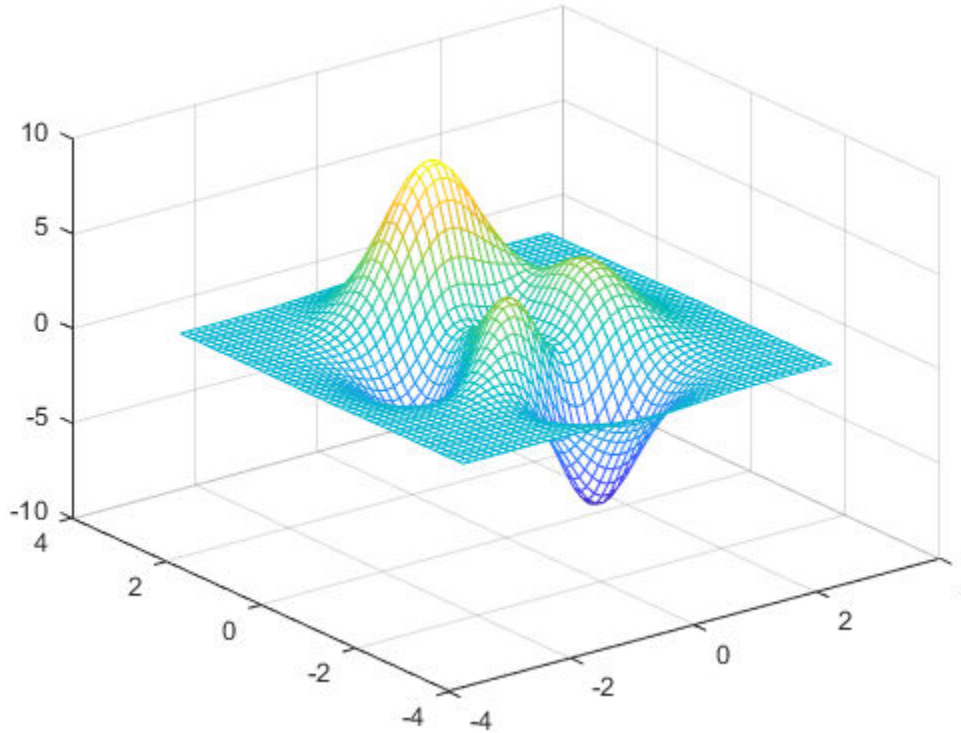
Set z-Axis Tick Labels Back to Default Labels

Create a surface plot and specify the *z*-axis tick values and labels. Then, set the values and labels back to the default values.

```
[x,y,z] = peaks;
mesh(x,y,z)
zticks([-8 0 8])
zticklabels({'z = -8','z = 0','z = 8'})
```



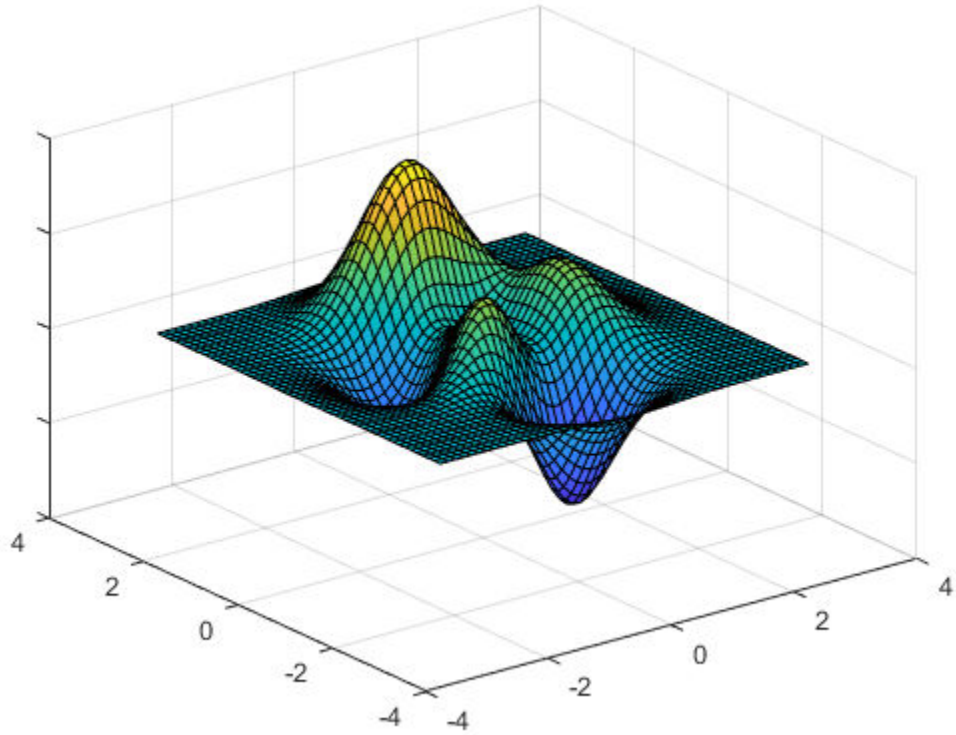

```
zticks('auto')
zticklabels('auto')
```



Remove z-Axis Tick Labels

Remove the tick labels along the *z*-axis by specifying the tick labels as an empty array.

```
[x,y,z] = peaks;
surf(x,y,z)
zticklabels({})
```



Input Arguments

`labels` — Tick labels

cell array of character vectors | string array | categorical array

Tick labels on page 1-17347, specified as a cell array of character vectors, string array, or categorical array. If you do not want tick labels to show, then specify an empty cell array `{}`. Tick labels support TeX and LaTeX markup. See the `TickLabelInterpreter` property of the `Axes` object for more information.

Example: `zticklabels({'0','\pi','2\pi'})`

Example: `zticklabels({'January','Febrary','March'})`

Example: `zticklabels({})`

Note

- If you specify the tick labels as a categorical array, MATLAB uses the values in the array, not the categories.
- To specify the tick values, use the `zticks` function.

`ax` — Target axes
current axes (default) | single `Axes` object | vector of `Axes` objects

Target axes, specified as a single `Axes` object or a vector of `Axes` objects. If you do not specify the axes, then `zticklabels` uses the current axes.

Output Arguments

`zl` — Current tick labels
cell array of character vectors | character array

Current tick labels, returned as a cell array of character vectors or a character array.

`m` — Current tick labels mode
`'auto'` | `'manual'`

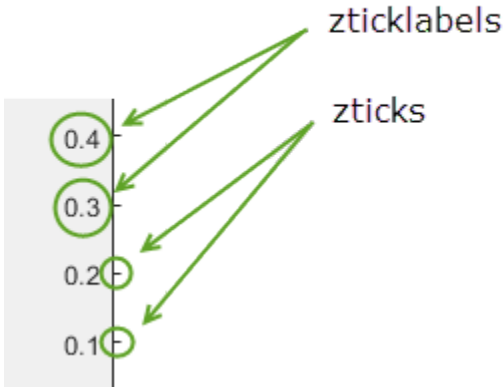
Current tick labels mode, returned as one of these values:

- `'auto'` — Automatically determine the *z*-axis tick labels.
- `'manual'` — Use manually specified *z*-axis tick labels.

More About

Tick Labels

The tick labels are the labels that you see next to each tick mark. The tick values are the locations along the *z*-axis where the tick marks appear. Set the tick values using the `zticks` function. Set the corresponding tick labels using the `zticklabels` function.



Algorithms

The `zticklabels` function sets and queries several axes properties related to the *z*-axis tick labels.

- `ZTickLabel` — Property that stores the text for the *z*-axis tick labels.
- `ZTickLabelMode` — Property that stores the *z*-axis tick label mode. When you set the *z*-axis tick labels using `zticklabels`, this property changes to `'manual'`.
- `ZTickMode` — Property that stores the *z*-axis tick value mode. When you set the *z*-axis tick labels using `zticklabels`, this property changes to `'manual'`.

See Also

Functions
`xticklabels` | `yticklabels` | `zlim` | `ztickangle` | `ztickformat` | `zticks`

Properties
Axes

Introduced in R2016b

zticks

Set or query *z*-axis tick values

Syntax

```
zticks(ticks)
zt = zticks

zticks('auto')
zticks('manual')
m = zticks('mode')

___ = zticks(ax, ___ )
```

Description

`zticks(ticks)` sets the *z*-axis tick values on page 1-17358, which are the locations along the *z*-axis where the tick marks appear. Specify `ticks` as a vector of increasing values; for example, `[0 2 4 6]`. This command affects the current axes.

`zt = zticks` returns the current *z*-axis tick values as a vector.

`zticks('auto')` sets an automatic mode, enabling the axes to determine the *z*-axis tick values. Use this option if you change the tick values and then want to set them back to the default values.

`zticks('manual')` sets a manual mode, freezing the *z*-axis tick values at the current values. Use this option if you want to retain the current tick values when resizing the axes or adding new data to the axes.

`m = zticks('mode')` returns the current *z*-axis tick values mode, which is either `'auto'` or `'manual'`. By default, the mode is automatic unless you specify tick values or change the mode to manual.

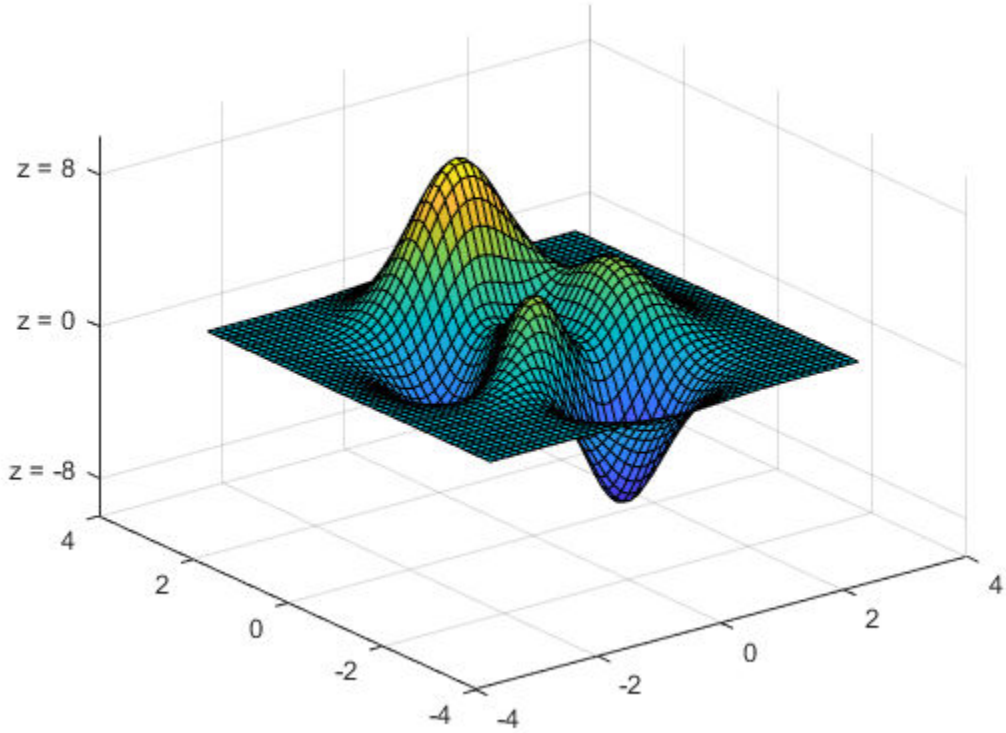
`___ = zticks(ax, ___ )` uses the axes specified by `ax` instead of the current axes. Specify `ax` as the first input argument for any of the previous syntaxes.

Examples

Specify z-Axis Tick Values and Labels

Create a surface plot. Display tick marks along the *z*-axis at the values -8, 0, and 8. Then, specify a label for each tick mark.

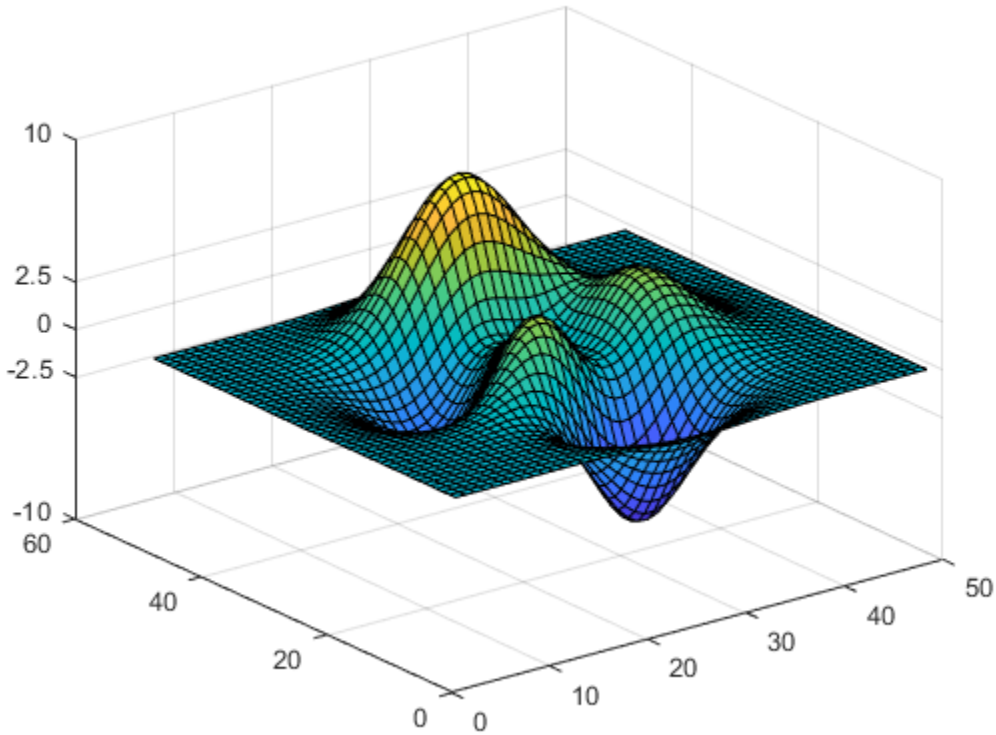
```
[x,y,z] = peaks;
surf(x,y,z)
zticks([-8 0 8])
zticklabels({'z = -8','z = 0','z = 8'})
```



Specify Nonuniform z-Axis Tick Values

Display tick marks along the *z*-axis at nonuniform values between -10 and 10. MATLAB® labels the tick marks with the numeric values.

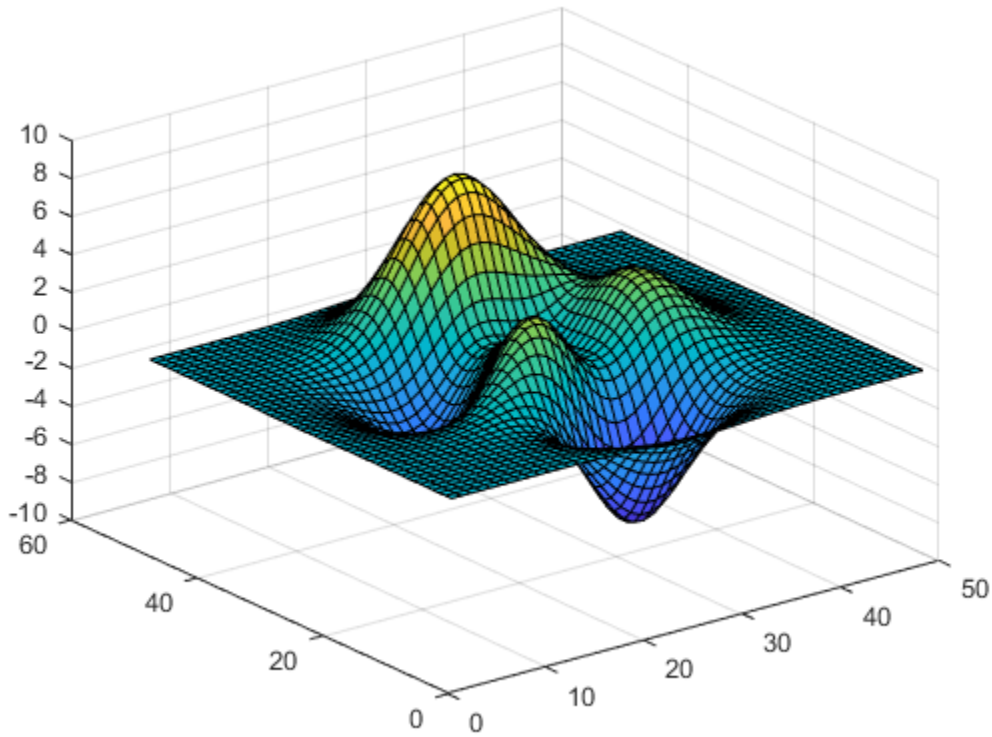
```
[x,y,z] = peaks;
surf(peaks)
zticks([-10 -2.5 0 2.5 10])
```



Increment z-Axis Tick Values by 2

Display tick marks along the *z*-axis at increments of 2, starting from -10 and ending at 10.

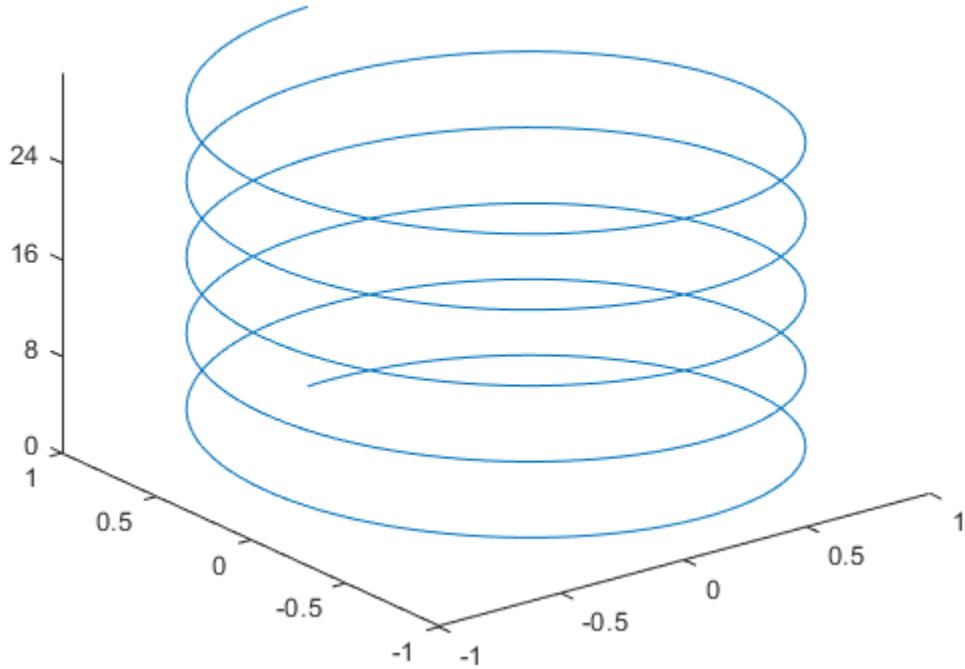
```
[x,y,z] = peaks;
surf(peaks)
zticks([-10:2:10])
```



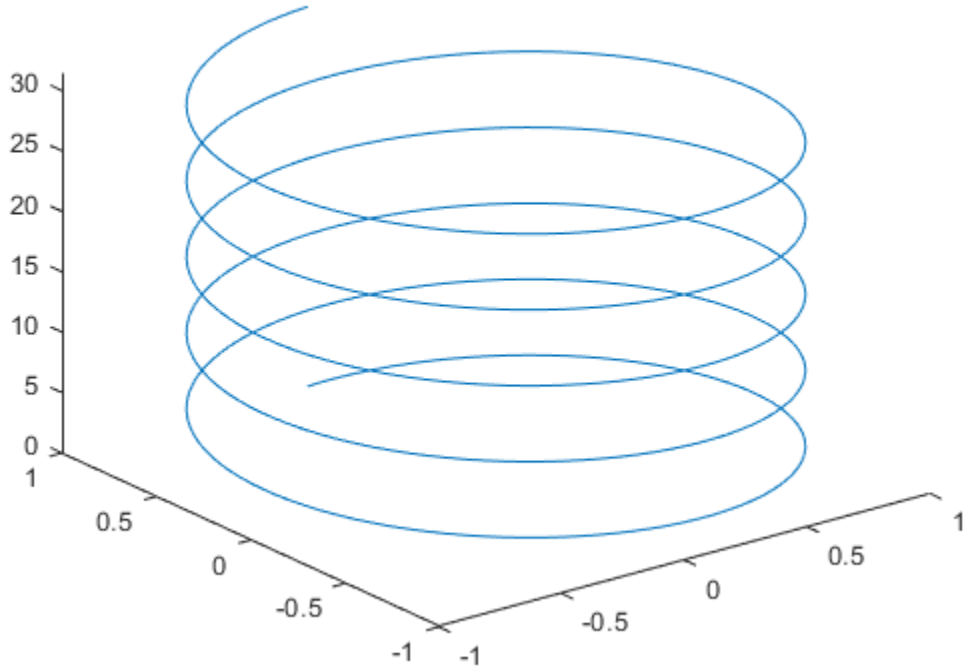
Set z-Axis Tick Values Back to Default Values

Create a plot and specify the *z*-axis tick values. Then, set the *z*-axis tick values back to the default values.

```
t = 0:pi/50:10*pi;
st = sin(t);
ct = cos(t);
plot3(st,ct,t)
zticks(0:8:40)
```




```
zticks('auto')
```

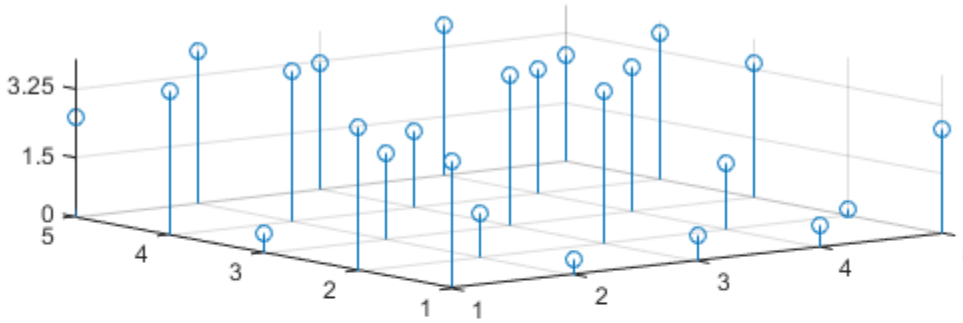


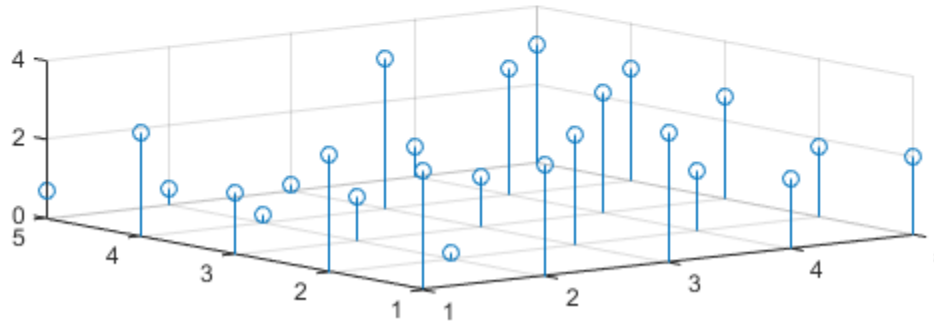
Specify z-Axis Tick Values for Specific Axes

Starting in R2019b, you can display a tiling of plots using the `tiledlayout` and `nexttile` functions. Call the `tiledlayout` function to create a 2-by-1 tiled chart layout. Call the `nexttile` function to create the axes objects `ax1` and `ax2`. Create two 3-D stem plots, and set the *z*-axis tick values for the upper plot by passing `ax1` as the first input argument to the `zticks` function.

```
tiledlayout(2,1)
ax1 = nexttile;
stem3(ax1,4*rand(5))
```

```
zticks(ax1,[0 1.5 3.25])

ax2 = nexttile;
stem3(ax2,4*rand(5))
```

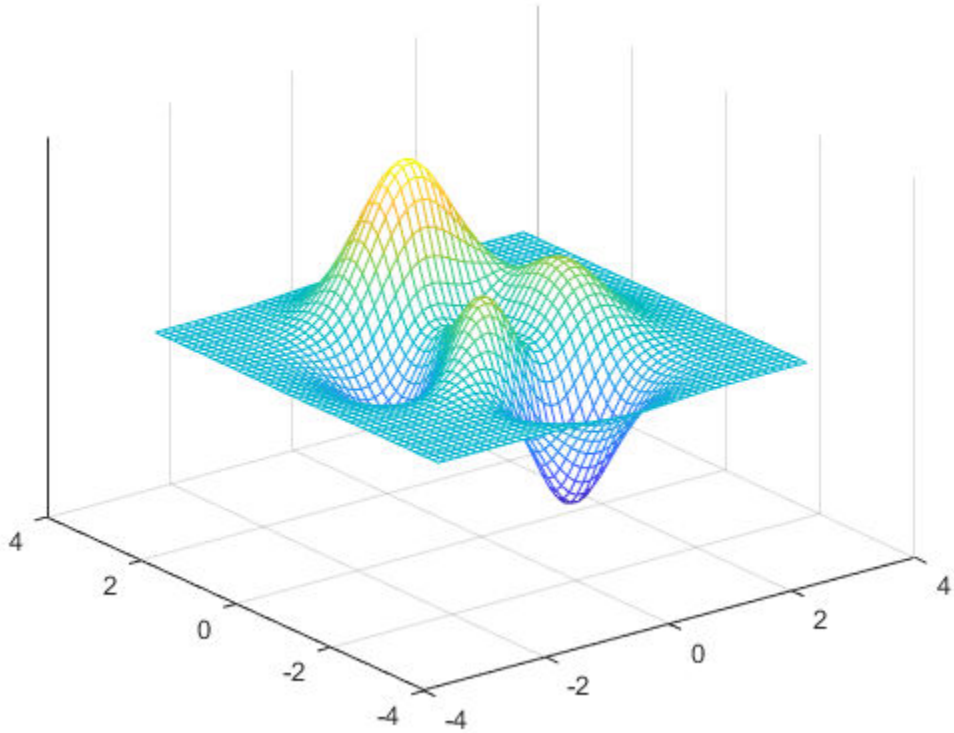




Remove z-Axis Tick Marks

Remove the tick marks along the *z*-axis by specifying the tick values as an empty array. Removing the tick marks also removes any grid lines in the *z* direction.

```
[x,y,z] = peaks;
mesh(x,y,z)
zticks([])
```



Input Arguments

`ticks` — Tick values
vector of increasing values

Tick values on page 1-17358, specified as a vector of increasing values. If you do not want tick marks along the *z*-axis, specify an empty vector `[]`.

You can specify the tick values as numeric, categorical, datetime, or duration values. However, the type of values that you specify must match the type of values along the *z*-axis.

Example: `zticks([pi 2*pi 3*pi 4*pi])`

Example: `zticks(0:10:100)`

Example: `zticks([])`

Note To specify the tick labels, use the `zticklabels` function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`ax` — Target axes
current axes (default) | single `Axes` object | vector of `Axes` objects

Target axes, specified as a single `Axes` object or a vector of `Axes` objects. If you do not specify the axes, then `zticks` uses the current axes.

Output Arguments

`zt` — Current tick values
vector

Current tick values, returned as a vector.

`m` — Current mode
`'auto'` | `'manual'`

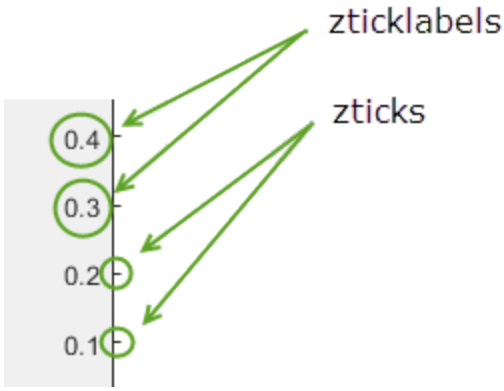
Current mode, returned as one of these values:

- `'auto'` — Automatically determine the *z*-axis tick values.
- `'manual'` — Use manually specified *z*-axis tick values.

More About

Tick Values

The tick values are the locations along the *z*-axis where the tick marks appear. The tick labels are the labels that you see next to each tick mark. Set the tick values using the `zticks` function. Set the corresponding tick labels using the `zticklabels` function.



Algorithms

The `zticks` function sets and queries several axes properties related to the *z*-axis tick values.

- `ZTick` — Property that stores the *z*-axis tick values.
- `ZTickMode` — Property that stores the *z*-axis tick value mode. When you set the *z*-axis tick values, this property changes to `'manual'`.

See Also

Functions
`xlim` | `xtickangle` | `xtickformat` | `xticklabels` | `xticks` | `yticks`

Properties
Axes

Introduced in R2016b

xlsfinfo

(Not recommended) Determine if file contains Microsoft Excel spreadsheet

Note `xlsfinfo` is not recommended. Use `sheetnames` instead. For more information, see Compatibility Considerations.

Syntax

```
status = xlsfinfo(filename)
[status,sheets] = xlsfinfo(filename)
[status,sheets,xlFormat] = xlsfinfo(filename)
```

Description

`status = xlsfinfo(filename)` indicates if `filename` is a file that the `xlsread` function can read.

`[status,sheets] = xlsfinfo(filename)` additionally returns the name of each spreadsheet in the file.

`[status,sheets,xlFormat] = xlsfinfo(filename)` also returns the format description that Excel returns for the file. On systems without Excel for Windows, `xlFormat` is an empty character vector, `''`.

Examples

View Information About Spreadsheet File

Create a sample Excel® file named `myExample.xlsx`.

```
values = {1, 2, 3 ; 4, 5, 'x' ; 7, 8, 9};
headers = {'First', 'Second', 'Third'};
xlswrite('myExample.xlsx', [headers; values]);
```

Call `xlsfinfo` to get information about the file.

```
[status,sheets,xlFormat] = xlsfinfo('myExample.xlsx')
```

```
status =

Microsoft Excel Spreadsheet


sheets =

    'Sheet1'    'Sheet2'    'Sheet3'


xlFormat =

xlOpenXMLWorkbook
```

`status` contains descriptive text which indicates that the `xlsread` function can read the sample file.

Input Arguments

`filename` — Name of file
character vector | string

Name of file, specified as a character vector or a string.

Example: `'myFile.xlsx'`

Data Types: `char` | `string`

Output Arguments

`status` — Type of file
character vector

Type of file, returned as a character vector.

- If `filename` is a file that `xlsread` can read, then `status` is descriptive text, such as `'Microsoft Excel Spreadsheet'`.
- If `filename` is not a file that `xlsread` can read, then `status` is empty, `''`.
- If MATLAB cannot find the file, then `xlsfinfo` returns an error.

`sheets` — Worksheet names
1-by-n cell array of character vectors

Worksheet names, returned as a 1-by-n cell array of character vectors, where n is the number of worksheets in the file. Each cell contains the name of a worksheet. If `xlsread` cannot read a particular worksheet, the corresponding cell contains an error message.

If `xlsfinfo` cannot read the file, then `sheets` contains an error message.

`xlFormat` — File format description returned by Excel
character vector

File format description returned by Excel, returned as a character vector.

On Windows systems with Excel software, `xlFormat` is one of the following.

`'xlOpenXMLWorkbook'`	Spreadsheet in XLSX format (Excel 2007 or later)
`'xlWorkbookNormal'` or `'xlExcel8'`	Spreadsheet in XLS format (compatible with Excel 97-2003)
`'xlCSV'`	File in comma-separated value (CSV) format
`'xlHtml'` or `'xlWebArchive'`	Spreadsheet exported to HTML format

On all other systems, `xlFormat` is an empty character vector, `''`.

Limitations

- `xlsfinfo` supports only 7-bit ASCII characters.

Tips

- If `xlsfinfo` warns that it cannot start an ActiveX server, then the COM server, which is part of the typical Excel installation, is unavailable. In this case, consider reinstalling your Excel software. On systems with Excel for Windows, `xlsfinfo` uses the COM server to obtain information.

Compatibility Considerations

`xlsfinfo` is not recommended

Not recommended starting in R2019b

`xlsfinfo` is not recommended. Use `sheetnames` instead. There are no plans to remove `xlsfinfo`.

Starting in R2019b, use the `sheetnames` function to get names of worksheets from a spreadsheet file. The `sheetnames` function has these advantages over the `xlsfinfo` function:

- Support for sheet names containing non-ASCII characters
- Better cross-platform support and performance
- Ability to work with remotely stored data

This table shows typical usages of `xlsfinfo` and how to update your code to use `sheetnames`.

Not Recommended	Recommended
`[~,sheets] = xlsfinfo('myData.xls')`	`sheets = sheetnames('myData.xls')`

See Also

`readcell` | `readmatrix` | `writecell` | `writematrix`

Introduced before R2006a

xlsread

(Not recommended) Read Microsoft Excel spreadsheet file

Note `xlsread` is not recommended. Use `readtable`, `readmatrix`, or `readcell` instead. For more information, see Compatibility Considerations.

Syntax

```
num = xlsread(filename)
num = xlsread(filename,sheet)
num = xlsread(filename,xlRange)
num = xlsread(filename,sheet,xlRange)
num = xlsread(filename,sheet,xlRange,'basic')
[num,txt,raw] = xlsread( ___ )
```

```
___ = xlsread(filename,-1)
```

```
[num,txt,raw,custom] = xlsread(filename,sheet,xlRange,'',processFcn)
```

Description

`num = xlsread(filename)` reads the first worksheet in the Microsoft Excel spreadsheet workbook named `filename` and returns the numeric data in a matrix.

`num = xlsread(filename,sheet)` reads the specified worksheet.

`num = xlsread(filename,xlRange)` reads from the specified range of the first worksheet in the workbook. Use Excel range syntax, such as `'A1:C3'`.

`num = xlsread(filename,sheet,xlRange)` reads from the specified worksheet and range.

`num = xlsread(filename,sheet,xlRange,'basic')` reads data from the spreadsheet in `basic` import mode. If your computer does not have Excel for Windows or if you are using MATLAB Online, `xlsread` automatically operates in `basic` import mode, which supports XLS, XLSX, XLSM, XLTX, and XLTM files.

If you do not specify all the arguments, use empty character vectors, `''` , as placeholders, for example, `num = xlsread(filename,'','','basic')`.

`[num,txt,raw] = xlsread( ___ )` additionally returns the text fields in cell array `txt`, and both numeric and text data in cell array `raw`, using any of the input arguments in the previous syntaxes.

`___ = xlsread(filename,-1)` opens an Excel window to interactively select data. Select the worksheet, drag and drop the mouse over the range you want, and click **OK**. This syntax is supported only on Windows computers with Microsoft Excel software installed.

`[num,txt,raw,custom] = xlsread(filename,sheet,xlRange,'',processFcn)`, where `processFcn` is a function handle, reads from the spreadsheet, calls `processFcn` on the data, and returns the final results as numeric data in array `num`. The `xlsread` function returns the text fields in cell array `txt`, both the numeric and text data in cell array `raw`, and the second output from `processFcn` in array `custom`. The `xlsread` function does not change the data stored in the spreadsheet. This syntax is supported only on Windows computers with Excel software.

Examples

Read Worksheet Into Numeric Matrix

Create an Excel file named `myExample.xlsx`.

```
values = {1, 2, 3 ; 4, 5, 'x' ; 7, 8, 9};
headers = {'First','Second','Third'};
xlswrite('myExample.xlsx',[headers; values]);
```

`Sheet1` of `myExample.xlsx` contains:

```
   First    Second    Third
       1         2        3
       4         5    x
       7         8        9
```

Read numeric data from the first worksheet.

```
filename = 'myExample.xlsx';
A = xlsread(filename)
```

```
A =
     1     2     3
     4     5   NaN
     7     8     9
```

Read Range of Cells

Read a specific range of data from the Excel file in the previous example.

```
filename = 'myExample.xlsx';
sheet = 1;
xlRange = 'B2:C3';

subsetA = xlsread(filename,sheet,xlRange)
```

```
subsetA =
     2     3
     5   NaN
```

Read Column

Read the second column from the Excel file in the first example.

```
filename = 'myExample.xlsx';

columnB = xlsread(filename,'B:B')
```

```
columnB =
     2
     5
     8
```

For better performance, include the row numbers in the range, such as `'B1:B3'`.

Request Numeric, Text, and Raw Data

Request the numeric data, text data, and combined data from the Excel file in the first example.

```
[num,txt,raw] = xlsread('myExample.xlsx')
```

```
num =
     1     2     3
     4     5   NaN
     7     8     9

txt =
    'First'    'Second'    'Third'
    ''         ''          ''
    ''         ''          'x'

raw =
    'First'    'Second'    'Third'
    [    1]    [     2]    [    3]
    [    4]    [     5]    'x'
    [    7]    [     8]    [    9]
```

Execute a Function on a Worksheet

In the Editor, create a function to process data from a worksheet. In this case, set values outside the range `[0.2,0.8]` to `0.2` or `0.8`.

```
function [Data] = setMinMax(Data)

minval = 0.2;
maxval = 0.8;

for k = 1:Data.Count
  v = Data.Value{k};
  if v > maxval
    Data.Value{k} = maxval;
  elseif v < minval
    Data.Value{k} = minval;
  end
end
```

In the Command Window, add random data to `myExample.xlsx`.

```
A = rand(5);
xlswrite('myExample.xlsx',A,'MyData')
```

The worksheet named `MyData` contains values ranging from 0 to 1.

Read the data from the worksheet, and reset any values outside the range `[0.2,0.8]`. Specify the sheet name, but use `''` as placeholders for the `xlRange` and `'basic'` inputs.

```
trim = xlsread('myExample.xlsx','MyData','','',@setMinMax);
```

Request Custom Output

Execute a function on a worksheet and display the custom index output.

In the Editor, modify the function `setMinMax` from the previous example to return the indices of the changed elements (custom output).

```
function [Data,indices] = setMinMax(Data)

minval = 0.2;
maxval = 0.8;
indices = [];

for k = 1:Data.Count
  v = Data.Value{k};
  if v > maxval
    Data.Value{k} = maxval;
    indices = [indices k];
  elseif v < minval
    Data.Value{k} = minval;
    indices = [indices k];
  end
end
```

Read the data from the worksheet `MyData`, and request the custom index output, `idx`.

```
[trim,txt,raw,idx] = xlsread('myExample.xlsx',...
    'MyData','','',@setMinMax);
```

Input Arguments

`filename` — File name
character vector | string

File name, specified as a character vector or a string. If you do not include an extension, `xlsread` searches for a file with the specified name and a supported Excel extension. `xlsread` can read data saved in files that are currently open in Excel for Windows.

Example: `'myFile.xlsx'` or `"myFile.xlsx"`

Data Types: `char` | `string`

`sheet` — Worksheet
character vector | string | positive integer

Worksheet, specified as one of the following:

- Character vector or string that contains the worksheet name. The name cannot contain a colon (`:`). To determine the names of the sheets in a spreadsheet file, use `xlsfinfo`. For XLS files in `basic` mode, `sheet` is case sensitive.
- Positive integer that indicates the worksheet index. This option is not supported for XLS files in `basic` mode.

Data Types: `char` | `string` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`xlRange` — Rectangular range
character vector | string

Rectangular range, specified as a character vector or a string.

Specify `xlRange` using two opposing corners that define the region to read. For example, `'D2:H4'` represents the 3-by-5 rectangular region between the two corners `D2` and `H4` on the worksheet. The `xlRange` input is not case sensitive, and uses Excel A1 reference style (see Excel help).

Range selection is not supported when reading XLS files in `basic` mode. In this case, use `''` in place of `xlRange`.

If you do not specify `sheet`, then `xlRange` must include both corners and a colon character, even for a single cell (such as `'D2:D2'`). Otherwise, `xlsread` interprets the input as a worksheet name (such as `'sales'` or `'D2'`).

If you specify `sheet`, then `xlRange`:

- Does not need to include a colon and opposite corner to describe a single cell.

- Can refer to a named range that you defined in the Excel file (see the Excel help).

When the specified `xlRange` overlaps merged cells:

- On Windows computers with Excel, `xlsread` expands the range to include all merged cells.
- On computers without Excel for Windows, `xlsread` returns data for the specified range only, with empty or `NaN` values for merged cells.

Data Types: `char` | `string`

`'basic'` — Flag to request reading in `basic` mode
character vector | string

Flag to request reading in `basic` mode, specified as the character vector or a string, `'basic'`.

`basic` mode is the default for computers without Excel for Windows. In `basic` mode, `xlsread`:

- Reads XLS, XLSX, XLSM, XLTX, and XLTM files only.
- Does not support an `xlRange` input when reading XLS files. In this case, use `''` in place of `xlRange`.
- Does not support function handle inputs.
- Imports all dates as Excel serial date numbers. Excel serial date numbers use a different reference date than MATLAB date numbers.

Data Types: `char` | `string`

`processFcn` — Handle to a custom function
function handle

Handle to a custom function. This argument is supported only on Windows computers with Excel software. `xlsread` reads from the spreadsheet, executes your function on a copy of the data, and returns the final results. `xlsread` does not change the data stored in the spreadsheet.

When `xlsread` calls the custom function, it passes a range interface from the Excel application to provide access to the data. The custom function must include this interface both as an input and output argument. (See “Execute a Function on a Worksheet” on page 1-17367)

Example: `@myFunction`

Output Arguments

`num` — Numeric data
matrix

Numeric data, returned as a matrix of `double` values. The array does not contain any information from header lines, or from outer rows or columns that contain nonnumeric data. Text data in inner spreadsheet rows and columns appear as `NaN` in the `num` output.

`txt` — Text data
cell array

Text data, returned as a cell array. Numeric values in inner spreadsheet rows and columns appear as empty character vectors, `''`, in `txt`.

For XLS files in `basic` import mode, the `txt` output contains empty character vectors, `''`, in place of leading columns of numeric data that precede text data in the spreadsheet. In all other cases, `txt` does not contain these additional columns.

Undefined values (such as `'#N/A'`) appear in the `txt` output as `'#N/A'`, except for XLS files in `basic` mode.

`raw` — Numeric and text data
cell array

Numeric and text data from the worksheet, returned as a cell array.

On computers with Excel for Windows, undefined values (such as `'#N/A'`) appear in the `raw` output as `'ActiveX VT_ERROR:'`. For XLSX, XLSM, XLTX, and XLTM files on other computers or in MATLAB Online, undefined values appear as `'#N/A'`.

`custom` — Second output of the function corresponding to `processFcn`
defined by the function

Second output of the function corresponding to `processFcn`. The value and data type of `custom` are determined by the function.

Limitations

- `xlsread` reads only 7-bit ASCII characters.
- `xlsread` does not support non-contiguous ranges.
- If your computer does not have Excel for Windows or if you are using MATLAB Online, `xlsread` automatically operates in `basic` import mode.
- On Linux and Mac platforms, `xlsread` cannot open spreadsheet files written by the `writetable` function.

Algorithms

- `xlsread` imports formatted text representing dates (such as `'10/31/96'`), except when importing in `basic` mode.

Compatibility Considerations

`xlsread` is not recommended

Not recommended starting in R2019a

`xlsread` is not recommended. Use `readtable`, `readmatrix`, or `readcell` instead. There are no plans to remove `xlsread`.

Starting in R2019a, import spreadsheet data as a table, a matrix, or a cell array by using `readtable`, `readmatrix`, or `readcell` respectively. The `readtable`, `readmatrix`, and `readcell` functions have these advantages over the `xlsread` function:

- Better cross-platform support and performance
- Automatic detection of data format and types
- Ability to use import options to control the data import process, including the handling of errors and missing data

This table shows typical usages of `xlsread` and how to update your code to use `readtable`, `readmatrix`, or `readcell` instead.

Not Recommended	Recommended

Read spreadsheet data as a matrix using `xlsread`:
 	
`M = xlsread(filename)`
	Read spreadsheet data as a table:
	
	`T = readtable(filename)`
	
	However, to continue reading your data as a matrix, use:
	
	`M = readmatrix(filename)`

Read spreadsheet data as a cell array using `xlsread`:
 	
`[~,~,C] = xlsread(filename)`
	Import spreadsheet data as a table:
	
	`T = readtable(filename)`
	
	However, to continue importing your data as a cell arrary, use:
	
	`C = readcell(filename)`

Read a specific sheet and range as a matrix using `xlsread`:
 	
`M = xlsread(filename,sheet,range)`
	Read a specific sheet and range as a table:
	
	`T = readtable(filename,'Sheet',sheet,'Range',range`
	
	However, to continue reading your data as a matrix, use:
	
	`M = readmatrix(filename,'Sheet',sheet,'Range',rang`

Read a specific sheet and range as a cell array using `xlsread`:
 	
`[~,~,C] = xlsread(filename,sheet,range)`
	Read a specific sheet and range as a table:
	
	`T = readtable(filename,'Sheet',sheet,'Range',range`
	
	However, to continue reading your data as a cell array:
	
	`C = readcell(filename,'Sheet',sheet,'Range',range)`

See Also

`importdata` | `readcell` | `readmatrix` | `readtable` | `uiimport` | `xlsfinfo` | `xlswrite`

Topics

“Create Function Handle”

Introduced before R2006a

xlswrite

(Not recommended) Write Microsoft Excel spreadsheet file

Note `xlswrite` is not recommended. Use `writetable`, `writematrix`, or `writecell` instead. For more information, see Compatibility Considerations.

Syntax

```
xlswrite(filename,A)
xlswrite(filename,A,sheet)
xlswrite(filename,A,xlRange)
xlswrite(filename,A,sheet,xlRange)

status = xlswrite( ___ )
[status,message] = xlswrite( ___ )
```

Description

`xlswrite(filename,A)` writes matrix `A` to the first worksheet in the Microsoft Excel spreadsheet workbook `filename` starting at cell `A1`.

`xlswrite(filename,A,sheet)` writes to the specified worksheet.

`xlswrite(filename,A,xlRange)` writes to the rectangular region specified by `xlRange` in the first worksheet of the workbook. Use Excel range syntax, such as `'A1:C3'`.

`xlswrite(filename,A,sheet,xlRange)` writes to the specified worksheet and range.

`status = xlswrite( ___ )` returns the status of the write operation, using any of the input arguments in previous syntaxes. When the operation is successful, status is `1`. Otherwise, status is `0`.

`[status,message] = xlswrite( ___ )` additionally returns any warning or error message generated by the write operation in structure `message`.

Examples

Write Vector to Spreadsheet

Write a 7-element vector to an Excel® file.

```
filename = 'testdata.xlsx';
A = [12.7 5.02 -98 63.9 0 -.2 56];
xlswrite(filename,A)
```

Write to Specific Sheet and Range in Spreadsheet

Write mixed text and numeric data to an Excel® file starting at cell `E1` of `Sheet2`.

```
filename = 'testdata.xlsx';
A = {'Time','Temperature'; 12,98; 13,99; 14,97};
sheet = 2;
xlRange = 'E1';
xlswrite(filename,A,sheet,xlRange)
```

Input Arguments

`filename` — File name
character vector | string

File name, specified as a character vector or a string.

If `filename` does not exist, `xlswrite` creates a file, determining the format based on the specified extension. To create a file compatible with Excel 97-2003 software, specify an extension of `.xls`. To create files in Excel 2007 formats, specify an extension of `.xlsx`, `.xlsb`, or `.xlsm`. If you do not specify an extension, `xlswrite` uses the default, `.xls`.

Example: `'myFile.xlsx'` or `"myFile.xlsx"`

Example: `'C:\myFolder\myFile.xlsx'`

Example: `'myFile.csv'`

Data Types: `char` | `string`

A — Input matrix
matrix

Input matrix, specified as a two-dimensional numeric, character array, or string array, or, if each cell contains a single element, a cell array.

If `A` is a cell array containing something other than a scalar numeric or text, then `xlswrite` silently leaves the corresponding cell in the spreadsheet empty.

The maximum size of array `A` depends on the associated Excel version. For more information on Excel specifications and limits, see the Excel help.

Example: `[10,2,45;-32,478,50]`

Example: `{92.0,'Yes',45.9,'No'}`

Example: `"ABCDEF"`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `cell`

`sheet` — Worksheet name
character vector | string | positive integer

Worksheet name, specified as one of the following:

- Character vector or string that contains the worksheet name. The name cannot contain a colon (`:`). To determine the names of the sheets in a spreadsheet file, use `xlsfinfo`.
- Positive integer that indicates the worksheet index.

If `sheet` does not exist, `xlswrite` adds a new sheet at the end of the worksheet collection. If `sheet` is an index larger than the number of worksheets, `xlswrite` appends empty sheets until the number of worksheets in the workbook equals `sheet`. In either case, `xlswrite` generates a warning indicating that it has added a new worksheet.

Data Types: `char` | `string` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`xlRange` — Rectangular range
character vector | string

Rectangular range, specified as a character vector or a string.

Specify `xlRange` using two opposing corners that define the region to write. For example, `'D2:H4'` represents the 3-by-5 rectangular region between the two corners `D2` and `H4` on the worksheet. The `xlRange` input is not case sensitive, and uses Excel A1 reference style (see Excel help). `xlswrite` does not recognize named ranges.

- If you do not specify `sheet`, then `xlRange` must include both corners and a colon character, even for a single cell (such as `'D2:D2'`). Otherwise, `xlswrite` interprets the input as a worksheet name (such as `'D2'`).
- If you specify `sheet`, then `xlRange` can specify only the first cell (such as `'D2'`). `xlswrite` writes input array `A` beginning at this cell.
- If `xlRange` is larger than the size of input array `A`, Excel software fills the remainder of the region with `#N/A`. If `xlRange` is smaller than the size of `A`, then `xlswrite` writes only the subset that fits into `xlRange` to the file.

Data Types: `char` | `string`

Output Arguments

`status` — Status of the write operation
`1` | `0`

Status of the write operation, returned as either `1` (`true`) or `0` (`false`). When the write operation is successful, `status` is `1`. Otherwise, `status` is `0`.

`message` — Error or warning generated during the write operation
structure array

Error or warning generated during the write operation, returned as a structure array containing two fields:

`message`	Text of the warning or error message.
`identifier`	Message identifier.

Limitations

- If your computer does not have Excel for Windows or you are using MATLAB Online, then the `xlswrite` function:

- Writes array `A` to a text file in comma-separated value (CSV) format. `A` must be a numeric matrix.
- Ignores the `sheet` and `xlRange` arguments.

This limitation also applies when the COM server (part of the typical installation of Excel) is not available.

Tips

- If your computer has Microsoft Office 2003 software, but you want to create a file in an Excel 2007 format, install the Office 2007 Compatibility Pack.
- Excel and MATLAB can store dates as text that represents those dates (such as `'10/31/96'`) or serial date numbers (such as `729329`). If your array includes serial date numbers, convert these dates to their text representation using `datestr` before calling `xlswrite`.
- To write data to Excel files with custom formats (such as fonts or colors), access the Windows COM server directly using `actxserver` rather than `xlswrite`. For example, this MathWorks Support Answer uses `actxserver` to establish a connection between MATLAB and Excel, writes data to a worksheet, and specifies the colors of the cells.

Algorithms

Excel converts `Inf` values to `65535`. MATLAB converts `NaN` values to empty cells.

Compatibility Considerations

`xlswrite` is not recommended

Not recommended starting in R2019a

`xlswrite` is not recommended. Use `writetable`, `writematrix`, or `writecell` instead. There are no plans to remove `xlswrite`.

Starting in R2019a, use `writetable`, `writematrix`, or `writecell` instead. The `writetable`, `writematrix`, and `writecell` functions have better cross-platform support and performance over the `xlswrite` function.

This table shows typical usages of `xlswrite` and how to update your code to use `writetable`, `writematrix`, or `writecell` instead.

Not Recommended	Recommended

`xlswrite(filename,M)`	To write tabular data to spreadsheets, use one of these options instead.
	
	Write a table:
	
	`writetable(T,filename)`
	
	Write a matrix:
	
	`writematrix(M,filename)`
	
	Write a cell array:
	
	`writecell(C,filename)`

See Also

`dlmwrite` | `writecell` | `writematrix` | `writetable` | `xlsfinfo` | `xlsread`

Introduced before R2006a

xmlread

Read XML document and return Document Object Model node

Syntax

```
DOMnode = xmlread(filename)
DOMnode = xmlread(filename,'AllowDoctype',tf)
```

Description

`DOMnode = xmlread(filename)` reads the specified XML file and returns `DOMnode` a Document Object Model node representing the document. Document Object Model is defined by the World Wide Web consortium. For more information, see “What Is an XML Document Object Model (DOM)?”.

`DOMnode = xmlread(filename,'AllowDoctype',tf)` also specifies whether DOCTYPE declarations are permitted. If `tf` is `false`, reading an input XML file containing DOCTYPE declarations results in an error. Otherwise, `xmlread` returns an output `DOMnode` for the XML file. The default value of `tf` is `true`.

Examples

Read XML File into Document Object Model (DOM) Node

Examine the contents of a sample XML file and then read the XML file into a Document Object Model(DOM) node.

Display the contents of the file `sample.xml`.

```
sampleXMLfile = 'sample.xml';
type(sampleXMLfile)
```

```
<productinfo>
```

```
<matlabrelease>R2012a</matlabrelease>
<name>Example Manager</name>
<type>internal</type>
<icon>ApplicationIcon.DEMOS</icon>

<list>
<listitem>
<label>Example Manager</label>
<callback>com.mathworks.xwidgets.ExampleManager.showViewer
</callback>
<icon>ApplicationIcon.DEMOS</icon>
</listitem>
</list>

</productinfo>
```

Read the XML file into a DOM node.

```
DOMnode = xmlread(sampleXMLfile);
```

Read XML File into MATLAB® Structure Array

Create a parsing function to read an XML file into a MATLAB® structure, and then read a sample XML file into the MATLAB workspace.

To create the function `parseXML`, copy and paste this code into an m-file `parseXML.m`, or use the `parseXML.m` included in this example. The `parseXML` function parses data from an XML file into a MATLAB structure array with fields `Name`, `Attributes`, `Data`, and `Children`.

```
type('parseXML.m')
```

```
function theStruct = parseXML(filename)
% PARSEXML Convert XML file to a MATLAB structure.
try
   tree = xmlread(filename);
catch
   error('Failed to read XML file %s.',filename);
end

% Recurse over child nodes. This could run into problems
% with very deeply nested trees.
```

```
try
   theStruct = parseChildNodes(tree);
catch
   error('Unable to parse XML file %s.',filename);
end


% ----- Local function PARSECHILDNODES -----
function children = parseChildNodes(theNode)
% Recurse over node children.
children = [];
if theNode.hasChildNodes
   childNodes = theNode.getChildNodes;
   numChildNodes = childNodes.getLength;
   allocCell = cell(1, numChildNodes);

   children = struct(             ...
      'Name', allocCell, 'Attributes', allocCell,    ...
      'Data', allocCell, 'Children', allocCell);

    for count = 1:numChildNodes
        theChild = childNodes.item(count-1);
        children(count) = makeStructFromNode(theChild);
    end
end

% ----- Local function MAKESTRUCTFROMNODE -----
function nodeStruct = makeStructFromNode(theNode)
% Create structure of node info.

nodeStruct = struct(                        ...
   'Name', char(theNode.getNodeName),       ...
   'Attributes', parseAttributes(theNode),  ...
   'Data', '',                              ...
   'Children', parseChildNodes(theNode));

if any(strcmp(methods(theNode), 'getData'))
   nodeStruct.Data = char(theNode.getData);
else
   nodeStruct.Data = '';
end

% ----- Local function PARSEATTRIBUTES -----
function attributes = parseAttributes(theNode)
```

```
% Create attributes structure.

attributes = [];
if theNode.hasAttributes
   theAttributes = theNode.getAttributes;
   numAttributes = theAttributes.getLength;
   allocCell = cell(1, numAttributes);
   attributes = struct('Name', allocCell, 'Value', ...
                       allocCell);

   for count = 1:numAttributes
      attrib = theAttributes.item(count-1);
      attributes(count).Name = char(attrib.getName);
      attributes(count).Value = char(attrib.getValue);
   end
end
```

Use the `parseXML` function to parse the sample file `info.xml` into a MATLAB structure.

```
sampleXMLfile = 'info.xml';
mlStruct = parseXML(sampleXMLfile)
```

```
mlStruct = struct with fields:
          Name: 'productinfo'
    Attributes: [1x2 struct]
          Data: ''
      Children: [1x13 struct]
```

Input Arguments

`filename` — File name
character vector | string scalar

File name, specified as a character vector or string scalar containing the name of the local file or URL.

Data Types: `char` | `string`

See Also

`xmlwrite` | `xslt`

Topics

"Importing XML Documents"

External Websites

DOM Package Summary (methods and properties for nodes)

Introduced before R2006a

xmlwrite

Write XML Document Object Model node

Syntax

```
xmlwrite(filename,DOMnode)
chr = xmlwrite(DOMnode)
```

Description

`xmlwrite(filename,DOMnode)` writes the Document Object Model (DOM) node `DOMnode` to the file `filename`.

`chr = xmlwrite(DOMnode)` returns a serialized DOM node as a character vector.

Examples

Create XML File

Write an XML file by, first, creating a Document Object Model(DOM) node containing the XML data. Then, write the DOM node to an XML file. The final XML file should contain this text.

```
<?xml version="1.0" encoding="utf-8"?>

<toc version="2.0">

<tocitem target="upslope_product_page.html">Upslope Area Toolbox<!--
Functions -->

<tocitem target="demFlow_help.html">demFlow</tocitem>

<tocitem target="facetFlow_help.html">facetFlow</tocitem>
```

```
<tocitem target="flowMatrix_help.html">flowMatrix</tocitem>

<tocitem target="pixelFlow_help.html">pixelFlow</tocitem>

</tocitem>

</toc>
```

First, create the DOM node object and root element, and populate the elements and the attributes of the node corresponding to the XML data.

```
docNode = com.mathworks.xml.XMLUtils.createDocument('toc');
```

Identify the root element, and set the `version` attribute.

```
toc = docNode.getDocumentElement;
toc.setAttribute('version','2.0');
```

Add the `tocitem` element node for the product page. Each `tocitem` element in this file has a `target` attribute and a child text node.

```
product = docNode.createElement('tocitem');
product.setAttribute('target','upslope_product_page.html');
product.appendChild(docNode.createTextNode('Upslope Area Toolbox'));
toc.appendChild(product);
```

Add comment.

```
product.appendChild(docNode.createComment(' Functions '));
```

Add a `tocitem` element node for each function, where the `target` is of the form *`function`*`_help.html`.

```
functions = {'demFlow','facetFlow','flowMatrix','pixelFlow'};
for idx = 1:numel(functions)
    curr_node = docNode.createElement('tocitem');

    curr_file = [functions{idx} '_help.html'];
    curr_node.setAttribute('target',curr_file);

    % Child text is the function name.
    curr_node.appendChild(docNode.createTextNode(functions{idx}));
    product.appendChild(curr_node);
end
```

Finally, export the DOM node to an XML file named `infoUAT.xml`, and view the file using the `type` function.

```
xmlwrite('infoUAT.xml',docNode);
type('infoUAT.xml');
```

```
<?xml version="1.0" encoding="utf-8"?>
<toc version="2.0">
   <tocitem target="upslope_product_page.html">Upslope Area Toolbox<!-- Functions --><t
      <tocitem target="facetFlow_help.html">facetFlow</tocitem>
      <tocitem target="flowMatrix_help.html">flowMatrix</tocitem>
      <tocitem target="pixelFlow_help.html">pixelFlow</tocitem>
   </tocitem>
</toc>
```

Get Document Object Model(DOM) Node as Serialized Text

Read a DOM node from a sample XML file and get the contents of the DOM node as a character vector.

Display the contents of the sample XML file, and then import the DOM node from the file.

```
sampleXMLfile = 'sample.xml';
type(sampleXMLfile)
```

```
<productinfo>

<matlabrelease>R2012a</matlabrelease>
<name>Example Manager</name>
<type>internal</type>
<icon>ApplicationIcon.DEMOS</icon>

<list>
<listitem>
<label>Example Manager</label>
<callback>com.mathworks.xwidgets.ExampleManager.showViewer
</callback>
<icon>ApplicationIcon.DEMOS</icon>
</listitem>
</list>

</productinfo>
```

```
DOMnode = xmlread(sampleXMLfile);
```

Use `xmlwrite` to return the `DOMnode` object as a serialized character vector.

```
text = xmlwrite(DOMnode)
```

```
text =
    '<?xml version="1.0" encoding="utf-8"?>
     <productinfo>

        <matlabrelease>R2012a</matlabrelease>
        <name>Example Manager</name>
        <type>internal</type>
        <icon>ApplicationIcon.DEMOS</icon>

        <list>
           <listitem>
              <label>Example Manager</label>
              <callback>com.mathworks.xwidgets.ExampleManager.showViewer
     </callback>
              <icon>ApplicationIcon.DEMOS</icon>
           </listitem>
        </list>

     </productinfo>'
```

Input Arguments

`filename` — File name

character vector | string scalar

File name, specified as a character vector or string scalar containing the name of the local file or URL.

Data Types: `char` | `string`

`DOMnode` — Document Object Model(DOM) node

DOM node object

Document Object Model(DOM) node, specified as a DOM node object.

Document Object Model is defined by the World Wide Web consortium. For more information, see “What Is an XML Document Object Model (DOM)?”.

See Also

`xmlread` | `xslt`

Topics

“Exporting to XML Documents”

External Websites

DOM Package Summary (methods and properties for nodes)

Introduced before R2006a

xor

Find logical exclusive-OR

Syntax

```
C = xor(A,B)
```

Description

`C = xor(A,B)` performs a logical exclusive-OR of arrays `A` and `B` and returns an array containing elements set to either logical 1 (`true`) or logical 0 (`false`). An element of the output array is set to logical 1 (`true`) if `A` or `B`, but not both, contains a nonzero element at that same array location. Otherwise, the array element is set to 0.

Examples

Exclusive-OR Truth Table

Create a truth table for `xor`.

```
A = [true false]
```

```
A = 1x2 logical array

   1   0
```

```
B = [true; false]
```

```
B = 2x1 logical array

   1
   0
```

```
C = xor(A,B)
```

```
C = 2x2 logical array

   0   1
   1   0
```

Input Arguments

A, B — Operands
scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs `A` and `B` must either be the same size or have sizes that are compatible (for example, `A` is an `M`-by-`N` matrix and `B` is a scalar or `1`-by-`N` row vector). For more information, see "Compatible Array Sizes for Basic Operations".

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`

Compatibility Considerations

Implicit expansion change affects arguments for operators

Behavior changed in R2016b

Starting in R2016b with the addition of implicit expansion, some combinations of arguments for basic operations that previously returned errors now produce results. For example, you previously could not add a row and a column vector, but those operands are now valid for addition. In other words, an expression like `[1 2] + [1; 2]` previously returned a size mismatch error, but now it executes.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a `try/catch` block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see "Compatible Array Sizes for Basic Operations".

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays”.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

This function fully supports distributed arrays. For more information, see “Run MATLAB Functions with Distributed Arrays” (Parallel Computing Toolbox).

See Also

`Logical Operators: Short Circuit` | `all` | `and` | `any` | `find` | `not` | `or`

Introduced before R2006a

xslt

Transform XML document using XSLT engine

Syntax

```
result = xslt(source,style,dest)
[result,style] = xslt( ___ )
xslt( ___ ,'-web')
```

Description

`result = xslt(source,style,dest)` transforms an XML document using a stylesheet and returns the resulting document's URL. The function uses these inputs, the first of which is required:

- `source` is the filename or URL of the source XML file specified as a character vector or string scalar. `source` can also specify a DOM node.
- `style` is the filename or URL of an XSL stylesheet specified as a character vector or string scalar.
- `dest` is the filename or URL of the desired output document specified as a character vector or string scalar. If `dest` is absent or empty, the function uses a temporary filename. If `dest` is `'-tostring'`, the function returns the output document as a character vector.

`[result,style] = xslt( ___ )` returns a processed stylesheet appropriate for passing to subsequent XSLT calls as `style`. This prevents costly repeated processing of the stylesheet.

`xslt( ___ ,'-web')` displays the resulting document in the Help Browser.

Examples

This example converts the file `info.xml` using the stylesheet `info.xsl`, writing the output to the file `info.html`. It launches the resulting HTML file in the MATLAB Web Browser.

```
xslt('info.xml', 'info.xsl', 'info.html', '-web')
```

Tips

MATLAB uses the Saxon XSLT processor, version 6.5.5, which supports XSLT 1.0 expressions. For more information, see `http://saxon.sourceforge.net/saxon6.5.5/`

For additional information on writing XSL stylesheets, see the World Wide Web Consortium (W3C®) web site, `https://www.w3.org/Style/XSL/`.

See Also

`xmlread` | `xmlwrite`

Introduced before R2006a

zeros

Create array of all zeros

Syntax

```
X = zeros
X = zeros(n)
X = zeros(sz1,...,szN)
X = zeros(sz)

X = zeros( ___ ,typename)
X = zeros( ___ ,'like',p)
```

Description

`X = zeros` returns the scalar `0`.

`X = zeros(n)` returns an `n`-by-`n` matrix of zeros.

`X = zeros(sz1,...,szN)` returns an `sz1`-by-...-by-`szN` array of zeros where `sz1,...,szN` indicate the size of each dimension. For example, `zeros(2,3)` returns a 2-by-3 matrix.

`X = zeros(sz)` returns an array of zeros where size vector `sz` defines `size(X)`. For example, `zeros([2 3])` returns a 2-by-3 matrix.

`X = zeros( ___ ,typename)` returns an array of zeros of data type `typename`. For example, `zeros('int8')` returns a scalar, 8-bit integer `0`. You can use any of the input arguments in the previous syntaxes.

`X = zeros( ___ ,'like',p)` returns an array of zeros like `p`; that is, of the same data type (class), sparsity, and complexity (real or complex) as `p`. You can specify `typename` or `'like'`, but not both.

Examples

Matrix of Zeros

Create a 4-by-4 matrix of zeros.

```
X = zeros(4)
```

```
X = 4×4

     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
```

3-D Array of Zeros

Create a 2-by-3-by-4 array of zeros.

```
X = zeros(2,3,4);
size(X)
```

```
ans = 1×3

     2     3     4
```

Clone Size from Existing Array

Create an array of zeros that is the same size as an existing array.

```
A = [1 4; 2 5; 3 6];
sz = size(A);
X = zeros(sz)
```

```
X = 3×2
```

```
     0     0
     0     0
     0     0
```

It is a common pattern to combine the previous two lines of code into a single line:

```
X = zeros(size(A));
```

Specify Data Type of Zeros

Create a 1-by-3 vector of zeros whose elements are 32-bit unsigned integers.

```
X = zeros(1,3,'uint32')
```

```
X = 1x3 uint32 row vector

   0   0   0
```

```
class(X)
```

```
ans =
'uint32'
```

Clone Complexity from Existing Array

Create a scalar `0` that is complex like an existing array instead of real valued.

First, create a complex vector.

```
p = [1+2i 3i];
```

Create a scalar `0` that is complex like `p`.

```
X = zeros('like',p)
```

```
X = 0.0000 + 0.0000i
```

Clone Sparsity from Existing Array

Create a 10-by-10 sparse matrix.

```
p = sparse(10,10,pi);
```

Create a 2-by-3 matrix of zeros that is sparse like p.

```
X = zeros(2,3,'like',p)
```

```
X =
   All zero sparse: 2x3
```

Clone Size and Data Type from Existing Array

Create a 2-by-3 array of 8-bit unsigned integers.

```
p = uint8([1 3 5; 2 4 6]);
```

Create an array of zeros that is the same size and data type as p.

```
X = zeros(size(p),'like',p)
```

```
X = 2x3 uint8 matrix

   0   0   0
   0   0   0
```

```
class(X)
```

```
ans =
'uint8'
```

Clone Distributed Array

If you have Parallel Computing Toolbox, create a 1000-by-1000 distributed array of zeros with underlying data type `int8`. For the `distributed` data type, the `'like'` syntax clones the underlying data type in addition to the primary data type.

```
p = zeros(1000,'int8','distributed');
```

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 6 workers.
```

Create an array of zeros that is the same size, primary data type, and underlying data type as `p`.

```
X = zeros(size(p),'like',p);
```

```
class(X)
```

```
ans =

distributed
```

```
classUnderlying(X)
```

```
ans =
int8
```

Input Arguments

n — Size of square matrix

integer value

Size of square matrix, specified as an integer value.

- If `n` is `0`, then `X` is an empty matrix.
- If `n` is negative, then it is treated as `0`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

sz1,...,szN — Size of each dimension (as separate arguments)

integer values

Size of each dimension, specified as separate arguments of integer values.

- If the size of any dimension is `0`, then `X` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.

- Beyond the second dimension, `zeros` ignores trailing dimensions with a size of `1`. For example, `zeros([3,1,1,1])` produces a 3-by-1 vector of zeros.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz` — Size of each dimension (as a row vector)
integer values

Size of each dimension, specified as a row vector of integer values. Each element of this vector indicates the size of the corresponding dimension:

- If the size of any dimension is `0`, then `X` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- Beyond the second dimension, `zeros` ignores trailing dimensions with a size of `1`. For example, `zeros([3,1,1,1])` produces a 3-by-1 vector of zeros.

Example: `sz = [2,3,4]` creates a 2-by-3-by-4 array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`typename` — Data type (class) to create
`'double'` (default) | `'single'` | `'logical'` | `'int8'` | `'uint8'` | ...

Data type (class) to create, specified as `'double'`, `'single'`, `'logical'`,`'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, `'int64'`, `'uint64'`, or the name of another class that provides `zeros` support.

`p` — Prototype of array to create
array

Prototype of array to create, specified as an array.

Data Types: `double` | `single` | `logical` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`
Complex Number Support: Yes

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Dimensions must be nonnegative real integers.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Dimensions must be nonnegative real integers.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `zeros` in the Parallel Computing Toolbox documentation.

Distributed Arrays

Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

Usage notes and limitations:

- See `zeros` in the Parallel Computing Toolbox documentation.

See Also

`complex` | `eye` | `false` | `ones` | `rand` | `size`

Topics

"Class Support for Array-Creation Functions"
"Preallocation"

Introduced before R2006a

zip

Compress files into zip file

Syntax

```
zip(zipfilename,filenames)
zip(zipfilename,filenames,rootfolder)
entrynames = zip( ___ )
```

Description

`zip(zipfilename,filenames)` compresses the contents of `filenames` into the zip file `zipfilename`. `zip` recursively compresses the content in folders. The resulting zip file contains the paths of `filenames` relative to the current folder. The zip file does not store absolute paths.

`zip(zipfilename,filenames,rootfolder)` specifies the paths for `filenames` relative to `rootfolder` rather than the current folder.

`entrynames = zip( ___ )` returns a cell array of character vectors containing the names of the files included in `zipfilename`. You can use this syntax with any of the input argument combinations in the previous syntaxes.

Examples

Compress a File

Create a zip file of the file `membrane.m`. Save the zip file `tmwlogo.zip` in the current folder.

```
zip('tmwlogo','membrane.m');
```

Compress Selected Files

Compress the files `membrane.m` and `logo.m` into a file named `tmwlogo.zip`.

```
zippedfiles = zip('tmwlogo.zip',{'membrane.m','logo.m'});
```

Compress all Code in Current Folder

Compress all `.m` and `.mlx` files in the current folder to the file `backup.zip`.

```
zip('backup',{'*.m','*.mlx'});
```

Compress a Folder

Compress the contents of a folder including all subfolders, and store the relative paths in the zip file.

Create a folder `myfolder` containing a subfolder `mysubfolder` and the files `membrane.m` and `logo.m`.

```
mkdir myfolder;
movefile('membrane.m','myfolder');
movefile('logo.m','myfolder');
cd myfolder;
mkdir mysubfolder;
cd ..
```

Compress the contents of `myfolder`, including all subfolders.

```
zippedfiles = zip('myfiles.zip','myfolder');
```

Compress Files Into Specified Folder

Suppose that you have the files `thesis.doc` and `defense.ppt` located in the folder `d:/PhD`. Compress these files into `thesis.zip`, one level up from the current folder.

```
zip('../thesis.zip',{'thesis.doc','defense.ppt'},'d:/PhD');
```

Input Arguments

`zipfilename` — Name of zip file

character vector | string scalar

Name of zip file to create, specified as a character vector or string scalar. If `zipfilename` does not have a `.zip` extension, MATLAB appends the `.zip` extension.

`zipfilename` must include a path relative to the current folder or an absolute path.

Data Types: `char` | `string`

`filenames` — Names of files or folders

character vector | cell array of character vectors | string array

Names of files or folders to compress, specified as a character vector, a cell array of character vectors, or a string array. The total number of files specified by `filenames` must be less than 65535, and each individual file must be smaller than 4 GB.

Files that are on the MATLAB path can include a partial path. Otherwise, files must include a path relative to the current folder or an absolute path.

Folders must include a path relative to the current folder or an absolute path. On UNIX systems, folders also can start with `~/` or `~username/`, which expands to the current user's home folder or the specified user's home folder, respectively. You can use the wildcard character * when specifying files or folders, except when relying on the MATLAB path to resolve a file name or partial path name.

Data Types: `char` | `string`

`rootfolder` — Root paths

character vector | string scalar

Root paths for the files to compress, specified as a character vector or string scalar.

Data Types: `char` | `string`

Alternative Functionality

To zip files in the Current Folder browser, select the file, right-click to open the context menu, and then select **Create Zip File**.

See Also

`gunzip` | `gzip` | `tar` | `untar` | `unzip`

Topics

"Create and Extract from Zip Archives"

Introduced before R2006a

zoom

Turn zooming on or off or magnify by factor

Syntax

```
zoom on
zoom off
zoom out
zoom reset
zoom
zoom xon
zoom yon
zoom(factor)
zoom(fig, option)
h = zoom(figure_handle)
```

Description

`zoom on` turns on zoom mode. When zoom mode is enabled in a figure, pressing a mouse button while your cursor is within an axes zooms into the point or out from the point beneath the mouse. Zooming changes the axes limits. When using zoom mode, you

- Zoom in by positioning the mouse cursor where you want the center of the plot to be and either
  - Press the mouse button or
  - Rotate the mouse scroll wheel away from you (upward).
- Zoom out by positioning the mouse cursor where you want the center of the plot to be and either
  - Simultaneously press **Shift** and the mouse button, or
  - Rotate the mouse scroll wheel toward you (downward).

Each mouse click or scroll wheel click zooms in or out by a factor of 2.

Clicking and dragging over an axes when zooming in is enabled draws a rubberband box. When you release the mouse button, the axes zoom in to the region enclosed by the rubberband box.

Double-clicking over an axes returns the axes to its initial zoom setting in both zoom-in and zoom-out modes.

`zoom off` turns off zoom mode. Starting in R2018b, some zoom interactions are enabled by default, regardless of the zoom mode. If you want to disable these default interactions, then use the `disableDefaultInteractivity` function.

`zoom out` returns the plot to its initial zoom setting.

`zoom reset` remembers the current zoom setting as the initial zoom setting. Later calls to `zoom out`, or double-clicks when interactive `zoom` mode is enabled, will return to this zoom level.

`zoom` toggles the zoom mode between off and on (restoring the most recently used zoom tool).

`zoom xon` and `zoom yon` set `zoom on` for the *x*- and *y*-axis, respectively.

`zoom(factor)` zooms in or out by the specified zoom factor, without affecting the interactive zoom mode. Values greater than 1 zoom in by that amount, while numbers greater than 0 and less than 1 zoom out by `1/factor`.

`zoom(fig, option)` Any of the preceding options can be specified on a figure other than the current figure using this syntax.

`h = zoom(figure_handle)` returns a zoom *mode object* for the figure `figure_handle` for you to customize the mode's behavior.

Using Zoom Mode Objects

You can access the following properties.

- *`Enable`* `'on'|'off'` — Specifies whether this figure mode is currently enabled on the figure.
- `FigureHandle <handle>` — The associated figure handle, a read-only property that cannot be set.

- *Motion* `'horizontal'|'vertical'|'both'` — The type of zooming enabled for the figure. This property only affects axes in a 2-D view (`[0 90]`).
- *Direction* `'in'|'out'` — The direction of the zoom operation.
- *RightClickAction* `'InverseZoom'|'PostContextMenu'` — The behavior of a right-click action.

  A value of `'InverseZoom'` causes a right-click to zoom out. A value of `'PostContextMenu'` displays a context menu. This setting persists between MATLAB sessions.
- `UIContextMenu <handle>` — Specifies a custom context menu to be displayed during a right-click action.

  This property is ignored if the `RightClickAction` property has been set to `'InverseZoom'`.

Zoom Mode Callbacks

You can program the following callbacks for zoom mode operations.

- `ButtonDownFilter <function_handle>` — Function to intercept `ButtonDown` events

  The application can inhibit the zoom operation under circumstances the programmer defines, depending on what the callback returns. The input function handle should reference a function with two implicit arguments (similar to graphics object callbacks), as follows:

  ```
  function [res] = myfunction(obj,event_obj)
  % obj          handle to the object clicked on
  % event_obj    struct for event data (empty in this release)
  % res [output] a logical flag determines whether the zoom
  %              operation should take place(for 'res' set
  %              to 'false' or the 'ButtonDownFcn' property
  %              of the object should take precedence (when
  %              'res' is 'true')
  ```
- `ActionPreCallback <function_handle>` — Function to execute before zooming

  Set this callback if you want to execute code when a zoom operation starts. The input function handle should reference a function with two implicit arguments (similar to graphics object callbacks), as follows:

```
function myfunction(obj,event_obj)
% obj         handle to the figure clicked on
% event_obj   object containing struct of event data
```

The event data has the following field.

`Axes`	The handle of the axes that is being zoomed

- `ActionPostCallback <function_handle>` — Function to execute after zooming

  Set this callback if you want to execute code when a zoom operation finishes. The input function handle should reference a function with two implicit arguments (similar to graphics object callbacks), as follows:

```
function myfunction(obj,event_obj)
% obj         handle to the figure clicked on
% event_obj   object containing struct of event data
%             (same as the event data of the
%             'ActionPreCallback' callback)
```

Zoom Mode Utility Functions

The following functions in zoom mode query and set certain of its properties.

- `flags = isAllowAxesZoom(h,ax)` — Function querying permission to zoom axes.

  Calling the function `isAllowAxesZoom` on the zoom object, `h`, with a vector of axes handles, `ax`, as input returns a logical array of the same dimension as the axes handle vector, which indicates whether a zoom operation is permitted on the axes objects.
- `setAllowAxesZoom(h,ax,flag)` — Function to set permission to zoom axes.

  Calling the function `setAllowAxesZoom` on the zoom object, `h`, with a vector of axes handles, `ax`, and a logical scalar, `flag`, either allows or disallows a zoom operation on the axes objects.
- `cn = getAxesZoomConstraint(h,ax)` — Function to get constraints of zoom operations.

  Calling the function `getAxesZoomConstraint` on the zoom object, `h`, with an axes object, `ax`, as input returns the constraint for the axes. The returned constraint is one of these values: `'x'`, `'y'`, `'z'`, `'xy'`, `'xz'`, `'yz'`, or `'unconstrained'`.

- `setAxesZoomConstraint(h,ax,cnstr)` — Function to set constraints of zoom operations.

  Calling the function `setAxesZoomConstraint` on the zoom object, `h`, with an axes object, `ax`, and a constraint option, `cnstr`, sets the constraint for the axes. Specify the constraint as one of these values: `'x'`, `'y'`, `'z'`, `'xy'`, `'xz'`, `'yz'`, or `'unconstrained'`.
- `sty = getAxes3DPanAndZoomStyle(h,ax)` — Function to get style of zoom operations.

  Calling the function `getAxes3DPanAndZoomStyle` on the zoom object, `h`, with a vector of axes handles, `ax`, as input returns the style of zooming for each axes. The returned value for each axes is either `'limits'` or `'camera'`.
- `setAxes3DPanAndZoomStyle(h,ax,style)` — Function to set style of zoom operations.

  Calling the function `setAxes3DPanAndZoomStyle` on the zoom object, `h`, with a vector of axes handles, `ax`, and a character array, `style`, sets the style of zooming on each axes. Specify the style as either `'limits'` or `'camera'`.
- `cns = getAxesZoomMotion(h,ax)` — Function to get constraints of zoom operations (not recommended, use `getAxesZoomConstraint`).

  Calling the function `getAxesZoomMotion` on the zoom object, `h`, with a vector of axes objects, `ax`, as input returns a character cell array of the same dimension as `ax`, which indicates the constraint for each axes. The returned value for each axes is `'horizontal'`, `'vertical'` or `'both'`.
- `setAxesZoomMotion(h,ax,constraints)` — Function to set constraints of zoom operations (not recommended, use `setAxesZoomConstraint`).

  Calling the function `setAxesZoomMotion` on the zoom object, `h`, with a vector of axes objects, `ax`, and a character array, `style`, sets the constraint for each axes. Specify the constraints as `'horizontal'`, `'vertical'` or `'both'`.

Examples

Example 1 — Entering Zoom Mode

Plot a graph and turn on Zoom mode:

```
plot(1:10);
zoom on
% zoom in on the plot
```

Example 2 — Constrained Zoom

Create zoom mode object and constrain to *x*-axis zooming.

Note Starting in R2014b, you can set properties using dot notation. If you are using an earlier release, use the `set` function instead, such as `set(h,'Motion','horizontal')`.

```
plot(1:10);
h = zoom;
h.Motion = 'horizontal';
h.Enable = 'on';
% zoom in on the plot in the horizontal direction.
```

Example 3 — Constrained Zoom in Subplots

Create four axes as subplots and set zoom style differently for each by setting a different property for each axes handle:

```
ax1 = subplot(2,2,1);
plot(1:10);
h = zoom;
ax2 = subplot(2,2,2);
plot(rand(3));
setAllowAxesZoom(h,ax2,false);
ax3 = subplot(2,2,3);
plot(peaks);
setAxesZoomMotion(h,ax3,'horizontal');
ax4 = subplot(2,2,4);
contour(peaks);
setAxesZoomMotion(h,ax4,'vertical');
% Zoom in on the plots.
```

Example 4 — Coding a ButtonDown Callback

Create a buttonDown callback for zoom mode objects to trigger. Copy the following code to a new file, execute it, and observe zooming behavior.

Note Starting in R2014b, you can set properties using dot notation. If you are using an earlier release, use the `set` function instead, such as `set(hLine,'ButtonDownFcn','disp(''This executes'')')`.

```
function demo
% Allow a line to have its own 'ButtonDownFcn' callback.
hLine = plot(rand(1,10));
hLine.ButtonDownFcn = 'disp(''This executes'')';
hLine.Tag = 'DoNotIgnore';
h = zoom;
h.ButtonDownFilter = @mycallback;
h.Enable = 'on';
% mouse click on the line
%
function [flag] = mycallback(obj,event_obj)
% If the tag of the object is 'DoNotIgnore', then return true.
objTag = obj.Tag;
if strcmpi(objTag,'DoNotIgnore')
   flag = true;
else
   flag = false;
end
```

Example 5 — Coding Pre- and Post-Callback Behavior

Create callbacks for pre- and post-buttonDown events for zoom mode objects to trigger. Copy the following code to a new file, execute it, and observe zoom behavior.

Note Starting in R2014b, you can set properties using dot notation. If you are using an earlier release, use the `set` function instead, such as `set(h,'ActionPreCallback',@myprecallback)`.

```
function demo
% Listen to zoom events
```

```
plot(1:10);
h = zoom;
h.ActionPreCallback = @myprecallback;
h.ActionPostCallback = @mypostcallback;
h.Enable = 'on';
%
function myprecallback(obj,evd)
disp('A zoom is about to occur.');
%
function mypostcallback(obj,evd)
newLim = evd.Axes.XLim;
msgbox(sprintf('The new X-Limits are [%.2f %.2f].',newLim));
```

Example 6 — Creating a Context Menu for Zoom Mode

Coding a context menu that lets the user to switch to Pan mode by right-clicking.

Note Starting in R2014b, you can set properties using dot notation. If you are using an earlier release, use the `set` function instead, such as `set(hZoom,'UIContextMenu',hCMZ)`.

```
figure
plot(magic(10))
hCMZ = uicontextmenu;
hZMenu = uimenu('Parent',hCMZ,'Label','Switch to pan',...
'Callback','pan(gcbf,''on'')');
hZoom = zoom(gcf);
hZoom.UIContextMenu = hCMZ;
zoom('on')
```

You cannot add items to the built-in zoom context menu, but you can replace it with your own.

Tips

`zoom` changes the axes limits by a factor of 2 (in or out) each time you press the mouse button while the cursor is within an axes. You can also click and drag the mouse to define a zoom area, or double-click to return to the initial zoom level.

You can create a zoom mode object once and use it to customize the behavior of different axes, as Example 3 illustrates. You can also change its callback functions on the fly.

Note **Do not change figure callbacks within an interactive mode.** While a mode is active (when panning, zooming, etc.), you will receive a warning if you attempt to change any of the figure's callbacks and the operation will not succeed. The one exception to this rule is the figure `WindowButtonMotionFcn` callback, which can be changed from within a mode. Therefore, if you are creating a UI that updates a figure's callbacks, the UI should somehow keep track of which interactive mode is active, if any, before attempting to do this.

When you assign different zoom behaviors to different `subplot` axes via a mode object and then link them using the `linkaxes` function, the behavior of the axes you manipulate with the mouse carries over to the linked axes, regardless of the behavior you previously set for the other axes.

Alternatives

Use the Zoom tools in the toolbar to zoom in or zoom out on a plot, or select **Zoom In** or **Zoom Out** from the figure's **Tools** menu.

See Also

`linkaxes` | `pan` | `rotate3d`

Introduced before R2006a

zoomInteraction

Zoom interaction

Description

A zoom interaction allows you to zoom into a chart without having to select any buttons in the axes toolbar. To enable zooming, set the `Interactions` property of the axes to a `zoomInteraction` object. When this interaction is enabled, you can zoom by scrolling or pinching within the chart.

To enable multiple interactions, set the `Interactions` property to an array of objects.

Creation

Syntax

```
z = zoomInteraction
z = zoomInteraction('Dimensions',d)
```

Description

`z = zoomInteraction` creates a zoom interaction object.

`z = zoomInteraction('Dimensions',d)` sets the `Dimensions` property. Use this property to constrain zooming to specific dimensions. For example, `z = zoomInteraction('Dimensions','x')` constrains zooming to the *x*-dimension.

Properties

`Dimensions` — Dimensions
`'xyz'` (default) | `'xy'` | `'yz'` | `'xz'` | `'x'` | `'y'` | `'z'`

Dimensions to allow zooming, specified as one of these values:

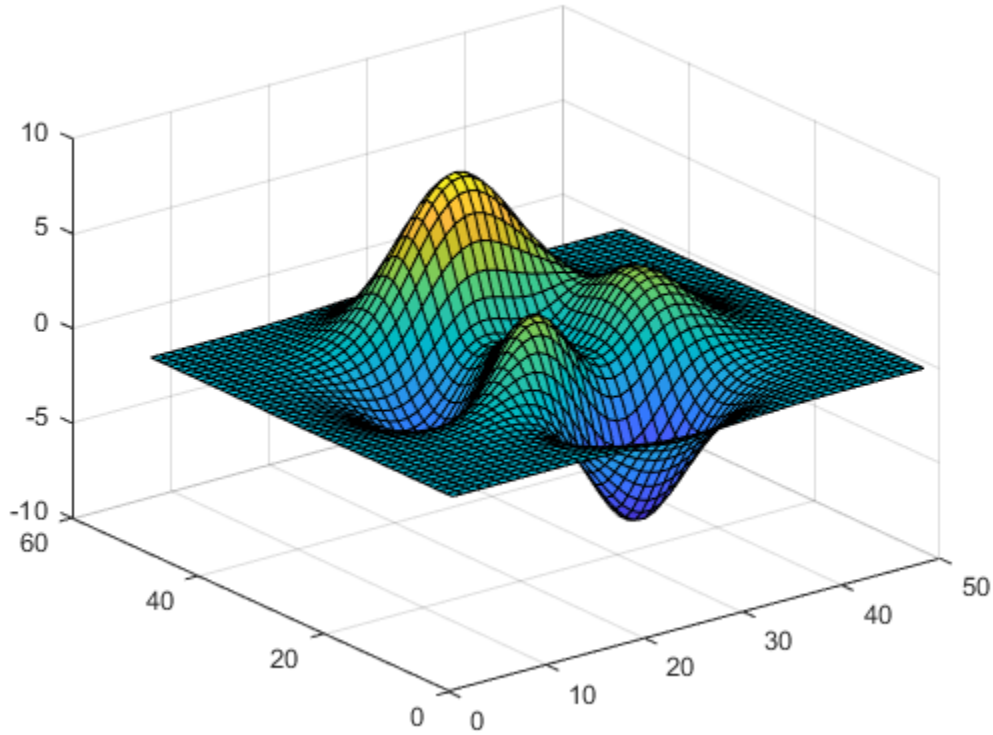
- `'xyz'` — Allows zooming in all dimensions.
- `'xy'` — Allows zooming in the *x* and *y* dimensions only.
- `'yz'` — Allows zooming in the *y* and *z* dimensions only.
- `'xz'` — Allows zooming in the *x* and *z* dimensions only.
- `'x'` — Allows zooming in the *x* dimension only.
- `'y'` — Allows zooming in the *y* dimension only.
- `'z'` — Allows zooming in the *z* dimension only.

Examples

Axes with Zoom Interaction

Create a surface plot. Get the current axes and replace the default interactions with just the zoom interaction. Then scroll or pinch to zoom in or out.

```
surf(peaks)
ax = gca;
ax.Interactions = zoomInteraction;
```

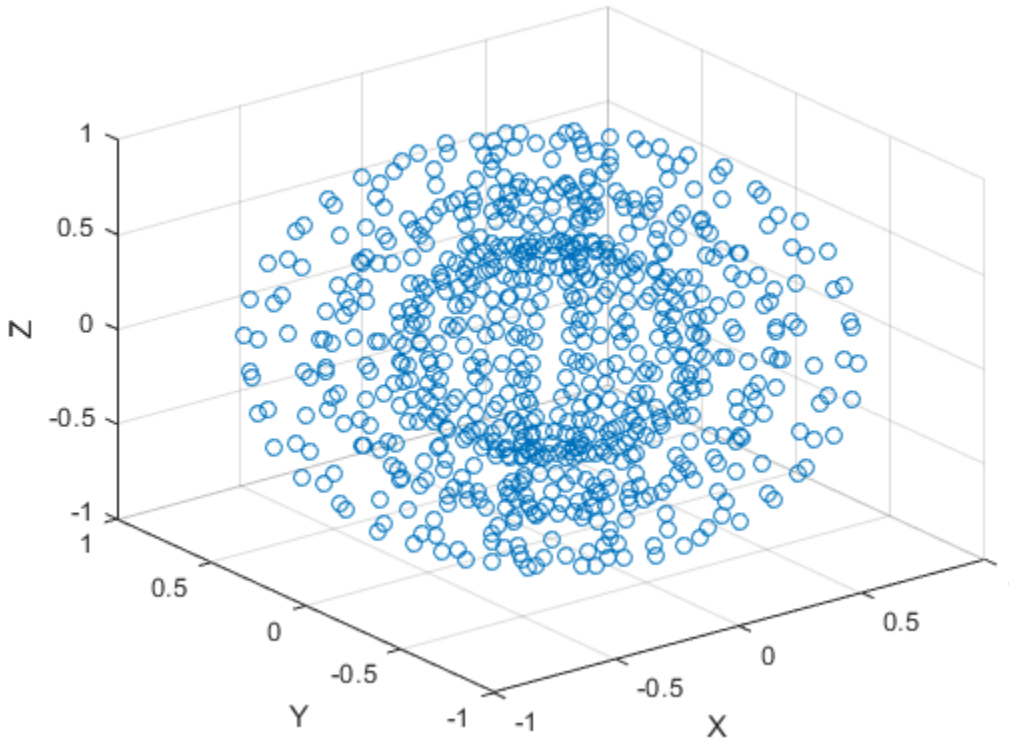


Restrict Zooming to X-Dimension

Use the `sphere` function to define vectors `x`, `y`, and `z`. Then create a scatter plot using those vectors. Replace the default set of interactions with a zoom interaction that is restricted to the *x*-dimension. Then scroll or pinch within the plot to zoom.

```
[X,Y,Z] = sphere(16);
x = [0.5*X(:); 0.75*X(:); X(:)];
y = [0.5*Y(:); 0.75*Y(:); Y(:)];
z = [0.5*Z(:); 0.75*Z(:); Z(:)];
scatter3(x,y,z)
xlabel('X')
```

```
ylabel('Y')
zlabel('Z')
ax = gca;
ax.Interactions = zoomInteraction('Dimensions','x');
```



Limitations

The `zoomInteraction` function is not supported in the Live Editor.

Tips

In most cases, the axes have a default set of interactions which depend on the type of chart you are displaying. You can replace the default set with a new set of interactions, but you cannot access or modify any of the interactions in the default set.

See Also

`dataTipInteraction` | `disableDefaultInteractivity` | `enableDefaultInteractivity` | `panInteraction` | `regionZoomInteraction` | `rotateInteraction` | `rulerPanInteraction`

Topics

"Control Chart Interactivity"

Introduced in R2019a

appdesigner

Open App Designer Start Page or existing app file

Syntax

```
appdesigner
appdesigner(filename)
```

Description

`appdesigner` opens the App Designer Start Page. From the Start Page you can create a new blank app, or a new 2-panel or 3-panel app with auto-reflow, or you can:

- View a list of your most recently used apps.
- Explore featured examples to help you get started.
- Take a guided tutorial to learn the basics of App Designer.
- Discover new App Designer features highlighted in the carousel.

`appdesigner(filename)` opens the specified `.mlapp` file in App Designer. If the `.mlapp` file is not on the MATLAB path, specify the full path.

Examples

Create New App with Auto-Reflow

Create a new app that automatically resizes and reflows content to respond to different device screen sizes.

Open the App Designer Start Page.

```
appdesigner
```

Select one of the preconfigured apps with auto-reflow from the Start Page. A new file titled `app1.mlapp` opens in App Designer. The `app1.mlapp` file does not appear in the MATLAB Current Folder browser until you save it.

Open Existing App File

Open and display an existing app file from your MATLAB search path.

```
appdesigner("mortgage")
```

Input Arguments

`filename` — Name or path of app file
character vector | string scalar

Name or path of an app file, specified as a character vector or string scalar. Including `.mlapp` in the file name is optional. Only one file name can be passed into the `appdesigner` function at time.

If `filename` appears in multiple folders on the search path, then include a path to the file. The file specification must be the complete or relative path to a file, or the name of a file on the MATLAB path.

Example: `appdesigner('app1.mlapp')`

Example: `appdesigner("app1")`

Example: `appdesigner("C:\Documents\Data\watertables.mlapp")`

Tips

You can also create or edit apps using the App Designer development environment in MATLAB Online (supported only for Google Chrome™ browsers).

See Also

Topics

"Ways to Build Apps"
"Create and Run a Simple App Using App Designer"
"GUIDE Migration Strategies"
"Displaying Graphics in App Designer"

Introduced in R2016a

Button Properties

Control button appearance and behavior

Description

Buttons are UI components that respond when the user presses and releases them. By changing property values, you can modify the appearance and behavior of a button. Use dot notation to refer to a specific object and property.

```
uf = uifigure;
b = uibutton(uf);
b.Text = 'Plot';
```

Properties

Button

`Text` — Button label

`'Button'` (default) | character vector | cell array of character vectors | string scalar | string array | ...

Button label, specified as a character vector, cell array of character vectors, string scalar, string array, or 1-D categorical array. Specify a character vector or string scalar to label the button with a single line of text. Use a cell array or string array to label the button with multiple lines of text. Each element in the array represents a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

`Icon` — File name of button icon

`''` (default) | character vector | string scalar

File name of the button icon, specified as a character vector or string scalar.

The file name can be an image file name on the MATLAB path or a full path to an image file. If you plan to share an app with others, put the image file on the MATLAB path to facilitate app packaging.

The image file type must be JPEG, GIF, or PNG.

- If the button text takes up all the space specified by the `Position` property value, then MATLAB does not display the icon.
- If some room is available for the icon, then MATLAB scales down the image to fit, if necessary.

Example: `'icon.png'`

Example: `'C:\Documents\icon.png'`

Font and Color

`FontName` — Font name

system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size

positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

`FontWeight` — Font weight

`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

FontAngle — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

FontColor — Font color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`BackgroundColor` — Background color

`[.96 .96 .96]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Interactivity

`Visible` — Button visibility

`'on'` (default) | `'off'`

Button visibility, specified as `'on'` or `'off'`. The `Visible` property determines whether the button is displayed on the screen. If the `Visible` property is set to `'off'`, the entire button is hidden, but you can still specify and access its properties.

To make your app start faster, set the `Visible` property of all components that do not need to appear at startup to `'off'`.

`Enable` — Operational state of button

`'on'` (default) | `'off'`

Operational state of the button, specified as `'on'` or `'off'`.

- If you set this property to `'on'`, the app user can press the button.
- If you set this property to `'off'`, the button appears dimmed, indicating that the app user cannot press it, and that the button will not trigger a callback.

`Tooltip` — Tooltip

`' '` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size of button

`[100 100 100 22]` (default) | `[left bottom width height]`

Location and size of button, specified as the vector of the form `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the button
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the button
`width`	Distance between the right and left outer edges of the button
`height`	Distance between the top and bottom outer edges of the button

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

All measurements are in pixel units.

`InnerPosition` — Location and size of button

`[100 100 100 22]` (default) | `[left bottom width height]`

Location and size of button, specified as a four element vector of the form `[left bottom width height]`. All measurements are in pixel units. This property value is identical to the `Position` property.

`OuterPosition` — Location and size of button
`[100 100 100 22]` (default) | `[left bottom width height]`

This property is read-only.

Location and size of button, returned as a four element vector of the form `[left bottom width height]`. All measurements are in pixel units. This property value is identical to the `Position` property.

`HorizontalAlignment` — Horizontal alignment of icon and text
`'center'` (default) | `'left'` | `'right'`

Horizontal alignment of the icon and text, specified as `'center'`, `'left'`, or `'right'`. The horizontal alignment is relative to the area inside the borders of the button. Setting this property when the text spans the full width of the button has no apparent effect in the UI.

`VerticalAlignment` — Vertical alignment of icon and text
`'center'` (default) | `'top'` | `'bottom'`

Vertical alignment of the icon and the text, specified as `'center'`, `'top'`, or `'bottom'`. The vertical alignment is relative to the area inside the borders of the button. Setting this property when the text height fills the height of the button has no apparent effect in the UI.

`IconAlignment` — Location of icon relative to button text
`'left'` (default) | `'right'` | `'center'` | `'top'` | `'bottom'`

Location of icon relative to the button text, specified as `'left'`, `'right'`, `'top'`, or `'bottom'`. If the `Text` property is empty, then the icon uses the `HorizontalAlignment` and `VerticalAlignment` properties instead of the `IconAlignment` property.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then

this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a button in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
b = uibutton(g);
b.Layout.Row = 3;
b.Layout.Column = 2;
```

To make the button span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this button spans columns `2` through `3`:

```
b.Layout.Column = [2 3];
```

Callbacks

`ButtonPushedFcn` — Button pushed callback

`''` (default) | function handle | cell array | character vector

Button pushed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when the user clicks the button in the app.

This callback function can access specific information about the user's interaction with the button. MATLAB passes this information in a `ButtonPushedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object's properties using dot notation. For example, `event.Source` returns the `Button` object. The `ButtonPushedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ButtonPushedData` object.

Property	Value
`EventName`	`'ButtonPushed'`
`Source`	Component that executes the callback

For more information about writing callbacks, see “Write Callbacks in App Designer”.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.

  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.

- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.

HandleVisibility Value	Description
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uibutton'`

This property is read-only.

Type of graphics object, returned as `'uibutton'`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uibutton` | `uifigure`

Introduced in R2016a

ButtonGroup Properties

Control button group appearance and behavior

Description

Button groups are containers for managing exclusive selection of radio buttons and toggle buttons. Properties control the appearance and behavior of a button group. Use dot notation to refer to a specific object and property.

```
uf = uifigure;
bg = uibuttongroup(uf);
bg.Title = 'Options';
```

The properties listed here are valid for button groups in App Designer, or in apps created with the `uifigure` function. For button groups used in GUIDE, or in apps created with the `figure` function, see Uibuttongroup.

Properties

Title

`Title` — Title

character vector | string scalar | categorical array

Title, specified as a character vector, string scalar, or categorical array. If you specify this property as a categorical array, MATLAB displays only the first element in the array.

MATLAB does not interpret a vertical slash (`'|'`) character as a line break, it displays as a vertical slash in the title.

If you want to specify a Unicode character, pass the Unicode decimal code to the `char` function. For example, `['Multiples of ' char(960)]` displays as `Multiples of` π.

`TitlePosition` — Location of title

`'lefttop'` (default) | `'centertop'` | `'righttop'`

Location of the title, specified as `'lefttop'`, `'centertop'`, or `'righttop'`.

Color and Styling

`ForegroundColor` — Title color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Title color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`BackgroundColor` — Background color

`[.94 .94 .94]` (default) | RGB Triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`BorderType` — Button group border
`'line'` (default) | `'none'`

Button group border, specified as `'line'` or `'none'`.

Font

`FontName` — Font name
system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size
positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

FontWeight — Font weight
`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

FontAngle — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

FontUnits — Font unit of measurement
`'pixels'` (default)

Font unit of measurement, specified as `'pixels'`. Distances in pixels are independent of your system resolution on Windows and Macintosh systems:

- On Windows systems, a pixel is 1/96th of an inch.
- On Macintosh systems, a pixel is 1/72nd of an inch.
- On Linux systems, the size of a pixel is determined by your system resolution.

Interactivity

Visible — Button group visibility
`'on'` (default) | `'off'`

Button group visibility, specified as `'on'` or `'off'`. The `Visible` property determines whether the button group is displayed on the screen. If the `Visible` property is set to `'off'`, the entire button group is hidden. However, you can still specify and access properties.

To make your app start faster, set the `Visible` property of all components that are not initially displayed to `'off'`.

Note Changing the `Visible` property of a `ButtonGroup` object does *not* change the values of the `Visible` properties of its child components. This is true even though hiding the button group causes the children to be hidden.

`Buttons` — Buttons managed by button group

array of `RadioButton` objects | array of `ToggleButton` objects

This property is read-only.

Buttons managed by the button group, returned as an array of `RadioButton` objects or an array of `ToggleButton` objects.

This property does not necessarily return all the child components of the `ButtonGroup` object. If you need to access child components that are not `RadioButton` or `ToggleButton` objects, use the `Children` property.

`SelectedObject` — Currently selected radio button or toggle button

first radio button or toggle button in button group (default)

Currently selected radio button or toggle button, specified as a `RadioButton` or a `ToggleButton` object.

Get the value of this property to determine which button is currently selected within the button group.

Set the value of this property to change the currently selected button. When you change the selection using this property, MATLAB adjusts the `Value` property for the other buttons within the button group accordingly.

For example, suppose that your button group contains three radio buttons and you set the `SelectedObject` property to `radiobutton3`. MATLAB sets the `Value` property for each child `RadioButton` as follows:

- `radiobutton1.Value = false;`
- `radiobutton2.Value = false;`
- `radiobutton3.Value = true;`

In other words, setting the `SelectedObject` property has the same effect as setting the `Value` property of a `RadioButton` or `ToggleButton`.

`Scrollable` — Ability to scroll

`'off'` (default) | `'on'`

Ability to scroll, specified as `'off'` or `'on'`. Setting this property to `'on'` enables scrolling within the container. However, there are additional requirements:

- The child components in the container must occupy a larger area than the container can display at one time.
- Components that do not fit in the container must be above or to the right of the container. You cannot scroll to components that are below or to the left of the container.

Certain types of charts and axes do not support scrollable containers. However, you can place the chart or axes in a nonscrollable panel, and then place the panel in the scrollable container. For more information, see “Displaying Graphics in App Designer”.

`Tooltip` — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size of button group, including borders and title

`[left bottom width height]`

Location and size of the button group, including borders and title, specified as a vector of the form `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the button group
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the button group
`width`	Distance between the right and left outer edges of the button group
`height`	Distance between the top and bottom outer edges of the button group

All measurements specified by the `Units` property value.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

`InnerPosition` — Location and size of button group, excluding borders and title
`[left bottom width height]`

This property is read-only.

Location and size of the button group, excluding borders and title, returned as a four-element vector of the form `[left bottom width height]`. This table describes each element in the vector.

Value	Description
`left`	Distance from the inner left edge of the parent container to the inner left edge of the button group.
`bottom`	Distance from the inner bottom edge of the parent container to the inner bottom edge of the button group.
`width`	Distance between the inner edges of the button group's right and left borders.
`height`	Distance between the inner edges of the button group's top and bottom borders. This distance excludes the title, if it exists.

All measurements are in units specified by the `Units` property.

Note These are some important points to consider when using the `InnerPosition` property:

- `InnerPosition` values are affected by the presence (or absence) of a title, the font characteristics, and the `BorderType`.
- `InnerPosition` values are relative to the parent container's drawable area. The drawable area is the area inside the borders of the container and does not include the area occupied by the title.

`OuterPosition` — Location and size of button group, including borders and title
`[left bottom width height]`

Location and size of the button group, including borders and title, specified as a four-element vector of the form `[left bottom width height]`. All measurements are in units specified by the `Units` property.

This property value is identical to the `Position` property value.

`Units` — Units of measurement
`'pixels'` (default)

Units of measurement, specified as `'pixels'`.

`AutoResizeChildren` — Automatically resize children
`'on'` (default) | `'off'`

Automatically resize children, specified as `'on'` or `'off'`.

- `'on'` — Child components automatically resize when the container resizes.
- `'off'` — Child components do not resize.

The `AutoResizeChildren` property affects direct children of the container, not children inside nested containers.

To customize the resize behavior, set the `AutoResizeChildren` property to `'off'` and create a `SizeChangedFcn` callback for the container. For more information, see "Managing Resizable Apps in App Designer".

To disable resizing of an app, set the `Resize` property of the figure to `'off'`.

Layout — Layout options

empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a button group in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
bg = uibuttongroup(g);
bg.Layout.Row = 3;
bg.Layout.Column = 2;
```

To make the button group span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this button group spans columns `2` through `3`:

```
bg.Layout.Column = [2 3];
```

Callbacks

SelectionChangedFcn — Selection changed callback

`''` (default) | function handle | cell array | character vector

Selection changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when the user selects a different button within the button group in the app. It does not execute if a radio or toggle button `Value` property changes programmatically.

This callback function can access specific information about the user's interaction with the buttons. MATLAB passes this information in a `SelectionChangedData` object as the

second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.NewValue` returns the currently selected button. The `SelectionChangedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `SelectionChangedData` object.

Property	Description
`OldValue`	Previously selected button
`NewValue`	Currently selected button
`Source`	Component that executes the callback
`EventName`	`'SelectionChanged'`

For more information about writing callbacks, see “Write Callbacks in App Designer”.

`SizeChangedFcn` — Size change callback

`''` (default) | function handle | cell array | character vector

Size change callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

Define this callback to customize the app layout when the size of this container changes (e.g., when the user resizes the window).

Note The `SizeChangedFcn` callback will not execute unless the `AutoResizeChildren` property of this container is set to `'off'`. In App Designer, you can make the `SizeChangedFcn` executable by clearing the **AutoResizeChildren** check box in the **UI Figure Properties** panel.

The `SizeChangedFcn` callback executes when:

- This container becomes visible for the first time.

- This container is visible while its size changes.
- This container becomes visible for the first time after its size changes. This situation occurs when the size changes while the container is invisible, and then it becomes visible later.

Other important points to consider when defining a `SizeChangedFcn` callback:

- Consider delaying the display of this container until after all the variables that the `SizeChangedFcn` uses are defined. This practice can prevent the `SizeChangedFcn` callback from returning an error. To delay the display of the container, set its `Visible` property to `'off'`. Then, set the `Visible` property to `'on'` after you define the variables that your `SizeChangedFcn` callback uses.
- If your app contains nested containers, they resize from the inside out.
- To access the container that is resizing from within the `SizeChangedFcn`, refer to the source object (the first input argument in the callback) or use the `gcbo` function.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`Children` — `ButtonGroup` children
empty `GraphicsPlaceholder` array (default) | 1-D array of component objects

`ButtonGroup` children, returned as an empty `GraphicsPlaceholder` or a 1-D array of component objects. The children of a `ButtonGroup` object can be any UI component object, including another `ButtonGroup` object.

You cannot add or remove child components using the `Children` property of the `ButtonGroup`. Use the `Children` property value to view the list of children or to reorder

the children. The order of the children in the property value array reflects the front-to-back order (stacking order) of the components on the screen.

To add a child to this list, set the `Parent` property of the child component to the `ButtonGroup`.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object
`'uibuttongroup'`

This property is read-only.

Type of graphics object, returned as `'uibuttongroup'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uibuttongroup` | `uifigure`

Introduced in R2016a

CheckBox Properties

Control check box appearance and behavior

Description

A check box is a UI component for indicating the state of a preference or option. Properties control the appearance and behavior of a check box. Use dot notation to refer to a specific object and property.

```
uf = uifigure;
cb = uicheckbox(uf);
cb.Text = 'Show value';
```

Properties

Check Box

`Value` — State of check box

0 (default) | 1

State of the check box, specified as `0` (false) or `1` (true). When the `Value` property is set to `1`, the check box is checked. When the `Value` property is set to `0`, the check box is not checked.

`Text` — Check box label

`'Check Box'` (default) | character vector | cell array of character vectors | string scalar | string array | ...

Check box label, specified as one of the following:

- Character vector or string scalar — Displays a single line of text. For example:

  ```
  uicheckbox('Text','Filter');
  ```

- Cell array of character vectors, string array, or 1-D categorical array— Displays multiple lines of text. Each row of the array is one line of text.

```
uicheckbox('Text',{'Filter','Results'},...
'Position',[100 100 84 30]);
```

If you set the `Text` property to a cell array or string array, change the `Position` property to accommodate the additional lines of text.

If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Font and Color

`FontName` — Font name

system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size

positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

`FontWeight` — Font weight

`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

`FontAngle` — Font angle

`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

`FontColor` — Font color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Interactivity

`Visible` — Check box visibility

`'on'` (default) | `'off'`

Check box visibility, specified as `'on'` or `'off'`. The `Visible` property determines whether the check box is displayed on the screen. If the `Visible` property is set to `'off'`, the check box is hidden, but you can still specify and access its properties.

To make your app start faster, set the `Visible` property of all components that do not need to appear at startup to `'off'`.

`Enable` — Operational state of check box

`'on'` (default) | `'off'`

Operational state of the check box, specified as `'on'` or `'off'`.

- If you set this property to `'on'`, then the app user can select or clear the check box.
- If you set this property to `'off'`, then the check box appears dimmed, indicating that the app user cannot select or clear it, and that it will not trigger a callback.

`Tooltip` — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this

property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size of check box

`[100 100 84 22]` (default) | `[left bottom width height]`

Location and size of the check box relative to the parent, specified as the vector `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the check box
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the check box
`width`	Distance between the right and left outer edges of the check box
`height`	Distance between the top and bottom outer edges of the check box

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

All measurements are in pixel units.

Example: `[200 200 102 15]`

`InnerPosition` — Inner location and size of check box

`[100 100 84 22]` (default) | `[left bottom width height]`

Inner location and size of the check box, specified as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property value for check boxes.

`OuterPosition` — Outer location and size of check box

`[100 100 84 22]` (default) | `[left bottom width height]`

This property is read-only.

Outer location and size of the check box returned as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property value.

Layout — Layout options

empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a check box in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
c = uicheckbox(g);
c.Layout.Row = 3;
c.Layout.Column = 2;
```

To make the check box span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this check box spans columns `2` through `3`:

```
c.Layout.Column = [2 3];
```

Callbacks

ValueChangedFcn — Value changed callback

[ ] (default) | function handle | cell array | character vector

Value changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when the user selects or clears the check box in the app. The callback does not execute if the check box value changes programmatically.

This callback function can access specific information about the user's interaction with the check box. MATLAB passes this information in a `ValueChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.PreviousValue` returns the previous value of the check box. The `ValueChangedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangedData` object.

Property	Value
`Value`	Value of check box after most recent app user interaction with it.
`PreviousValue`	Value of check box before most recent app user interaction with it.
`Source`	Component that executes the callback.
`EventName`	`'ValueChanged'`

For more information about writing callbacks, see "Write Callbacks in App Designer".

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see "Write Callbacks in App Designer".

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running

callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that

obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uicheckbox'`

This property is read-only.

Type of graphics object, returned as `'uicheckbox'`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uicheckbox` | `uifigure`

Introduced in R2016a

DiscreteKnob Properties

Control discrete knob appearance and behavior

Description

A discrete knob is a UI component for selecting an option from a discrete set. By changing property values, you can modify the appearance and behavior of a discrete knob. Use dot notation to refer to a specific object and property.

```
uf = uifigure;
k = uiknob(uf,'discrete');
k.Items = {'Freezing','Cold','Warm','Hot'};
```

Properties

Knob

`Value` — Value
element of `Items` | element of `ItemsData`

Value, specified as an element of the `Items` or `ItemsData` arrays. By default, `Value` is the first element in `Items`.

Specifying `Value` as an element of `Items` sets the knob selector to the corresponding label on the knob. If `ItemsData` is not empty, then `Value` must be set to an element of `ItemsData`, and the knob selector will point to the associated label.

`Items` — Knob options
`{'Off','Low','Medium','High'}` (default) | cell array of character vectors | string array | ...

Knob options, specified as a cell array of character vectors, string array, or 1-D categorical array. The array must contain at least two elements. The knob displays as many options as there are elements in the `Items` array. The options display in clockwise order. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Example: `{'Off','Slow','Fast'}`

Example: `{'1','2','3','4'}`

`ItemsData` — Data associated with each element of `Items`
empty array `[]` (default) | 1-by-n numeric array | 1-by-n cell array

Data associated with each element of the `Items` property value, specified as a 1-by-n numeric array or a 1-by-n cell array. It is valid to specify duplicate array elements in the `ItemsData` value.

When the number of array elements in the `ItemsData` and `Items` do not match:

- If the `ItemsData` value is empty, then all `Items` elements are presented to the app user.
- If the `ItemsData` value has more elements than the `Items` value, then all the `Items` elements are presented to the app user and MATLAB ignores the extra `ItemsData` elements.
- If the `ItemsData` value has fewer elements than the `Items` value (but greater than none), then the only `Items` elements presented to the app user are those that have a corresponding `ItemsData` value.

Example: `{'One' 'Two' 'Three'}`

Example: `{10 20 30 40}`

Font

`FontName` — Font name
system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size
positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

FontWeight — Font weight
`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

FontAngle — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

FontColor — Font color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Interactivity

`Visible` — Visibility of knob

`'on'` (default) | `'off'`

Visibility of the knob, specified as `'on'` or `'off'`. The `Visible` property determines whether the knob is displayed on the screen. If the `Visible` property is set to `'off'`, the entire knob is hidden, but you can still specify and access its properties.

To make your app start faster, set the `Visible` property of all components that do not need to appear at startup to `'off'`.

`Enable` — Operational state of knob

`'on'` (default) | `'off'`

Operational state of the knob, specified as `'on'` or `'off'`.

- If you set this property to `'on'`, then the app user can turn the knob.
- If you set this property to `'off'`, then the appearance of the knob appears dimmed, indicating that the app user cannot turn the knob and it will not trigger a callback.

`Tooltip` — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size of knob

`[100 100 60 60]` (default) | `[left bottom width height]`

Location and size of the knob, excluding the state marks and labels, specified as the vector, `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the knob
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the knob
`width`	Distance between the right and left outer edges of the knob, excluding tick marks and labels
`height`	Distance between the top and bottom outer edges of the knob, excluding tick marks and labels

All measurements are in pixel units. Due to aspect ratio constraints, you cannot change the knob height and width independently of each other. To increase the knob size, set width and height using `Position(3:4) = [width height]`.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[100 200 60 60]`

`InnerPosition` — Inner location and size of knob
`[100 100 60 60]` (default) | `[left bottom width height]`

Inner location and size of the knob, excluding state marks and state labels, specified as the vector, `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to `Position` for knob components.

`OuterPosition` — Outer location and size of knob
`[73 100 127 76]` (default) | `[left bottom width height]`

This property is read-only.

Outer location and size of the knob, including state marks and labels, returned as the vector, `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a discrete knob in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
k = uiknob(g,'discrete');
k.Layout.Row = 3;
k.Layout.Column = 2;
```

To make the knob span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this knob spans columns `2` through `3`:

```
k.Layout.Column = [2 3];
```

Callbacks

`ValueChangedFcn` — Value changed callback

`''` (default) | function handle | cell array | character vector

Value changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when the user turns the knob selector in the app. The callback does not execute if the `Value` property changes programmatically.

This callback can access specific information about the user's interaction with the knob. MATLAB passes this information in a `ValueChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.PreviousValue` returns the previous value of the knob. The `ValueChangedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangedData` object.

Property	Value
`Value`	Value of knob after app user's most recent interaction with it.
`PreviousValue`	Value of knob before app user's most recent interaction with it.
`Source`	Component that executes the callback.
`EventName`	`'ValueChanged'`

For more information about writing callbacks, see "Write Callbacks in App Designer".

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.

- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

Parent — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

HandleVisibility — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uidiscreteknob'`

This property is read-only.

Type of graphics object, returned as `'uidiscreteknob'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uifigure` | `uiknob`

Introduced in R2016a

DropDown Properties

Control drop-down list appearance and behavior

Description

Drop-down lists are UI components that enable the user to select an option or type in text. Properties control the appearance and behavior of a drop-down list. Use dot notation to refer to a specific object and property.

```
uf = uifigure;
dd = uidropdown(uf);
dd.Items = {'Red','Green','Blue'};
```

Properties

Drop-Down

`Value` — Value
element of `Items` | element of `ItemsData`

Value, specified as an element of the `Items` or `ItemsData` arrays. By default, `Value` is the first element in `Items`.

Specifying `Value` as an element of `Items` selects the drop-down item that matches that element. If `ItemsData` is not empty, then `Value` must be set to an element of `ItemsData`, and the drop-down will select the associated item in the list.

`Items` — Drop-down items
`{'Option 1','Option 2','Option 3','Option 4'}` (default) | cell array of character vectors | string array | ...

Drop-down items, specified as a cell array of character vectors, string array, or 1-D categorical array. Duplicate elements are allowed. The drop-down component displays as many options as there are elements in the `Items` array. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Example: `{'Red','Yellow','Blue'}`

Example: `{'1','2','3'}`

`ItemsData` — Data associated with each element of the `Items` property value
empty array (`[]`) (default) | 1-by-n numeric array | 1-by-n cell array

Data associated with each element of the `Items` property value, specified as a 1-by-n numeric array or a 1-by-n cell array. Duplicate elements are allowed.

For example, if you set the `Items` value to employee names, you might set the `ItemsData` value to corresponding employee ID numbers. The `ItemsData` value is not visible to the app user.

If the number of array elements in the `ItemsData` value and the `Items` value do not match, one of the following occurs:

- When the `ItemsData` value is empty, then all the elements of the `Items` value are presented to the app user.
- When the `ItemsData` value has more elements than the `Items` value, then all the elements of the `Items` value are presented to the app user. MATLAB ignores the extra `ItemsData` elements.
- When the `ItemsData` value is not empty, but has fewer elements than the `Items` value, the only elements of the `Items` value presented to the app user are those that have a corresponding element in the `ItemsData` value.

Example: `{'One','Two','Three'}`

Example: `[10 20 30 40]`

Font and Color

`FontName` — Font name
system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size
positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

`FontWeight` — Font weight
`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

`FontAngle` — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

`FontColor` — Font color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`BackgroundColor` — Background color

`[.96 .96 .96]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Interactivity

`Visible` — Drop-down component visibility

`'on'` (default) | `'off'`

Drop-down component visibility, specified as `'on'` or `'off'`. The `Visible` property determines whether the drop-down component is displayed on the screen. If the `Visible` property is set to `'off'`, then the drop-down component is hidden, but you can still specify and access its properties.

To make your app start faster, set the `Visible` property of all components that do not need to appear at startup to `'off'`.

`Editable` — Editable state of drop-down component
`'off'` (default) | `'on'`

Editable state of the drop-down component, specified as `'off'` or `'on'`.

If the `Enable` property value is `'off'`, then the app user cannot change the drop-down component text, even if the Editable property value is `'on'`.

`Enable` — Operational state of drop-down component
`'on'` (default) | `'off'`

Operational state of the drop-down component , specified as `'on'` or `'off'`.

- If you set this property value to `'on'`, then the appearance of the drop-down component indicates that the app user can change the drop-down component value.
- If you set this property value to `'off'`, then the appearance of the drop-down component appears dimmed. This appearance indicates that the app user cannot change the drop-down component value, and that the drop-down component will not trigger any callbacks.

`Tooltip` — Tooltip
`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size of drop-down component

`[100 100 100 22]` (default) | `[left bottom width height]`

Location and size of the drop-down component relative to the parent, specified as the vector `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the drop-down component
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the drop-down component
`width`	Distance between the right and left outer edges of the drop-down component
`height`	Distance between the top and bottom outer edges of the drop-down component

All measurements are in pixel units.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[100 100 100 22]`

`InnerPosition` — Inner location and size of drop-down component

`[100 100 100 22]` (default) | `[left bottom width height]`

Inner location and size of the drop-down component, specified as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to `Position` for drop-down components.

`OuterPosition` — Outer location and size of drop-down component

`[100 100 100 22]]` (default) | `[left bottom width height]`

This property is read-only.

Outer location and size of drop-down component returned as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to `Position` for drop-down components.

`Layout` — Layout options

empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a drop-down in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
dd = uidropdown(g);
dd.Layout.Row = 3;
dd.Layout.Column = 2;
```

To make the drop-down span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this drop-down spans columns `2` through `3`:

```
dd.Layout.Column = [2 3];
```

Callbacks

`ValueChangedFcn` — Value changed callback

`''` (default) | function handle | cell array | character vector

Value changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback function executes when the user selects a different option from the drop-down list. It does not execute if the `Value` property changes programmatically.

This callback function can access specific information about the user's interaction with the drop-down. MATLAB passes this information in a `ValueChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.PreviousValue` returns the previous value of the drop-down. The `ValueChangedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangedData` object.

Property	Value
`Value`	Drop-down component value after app user's most recent interaction with it.
`PreviousValue`	Drop-down component value before app user's most recent interaction with it.
`Edited`	Logical value (0 or 1) that indicates whether the callback was executed as a result of typing a value into the drop-down component. The `Edited` value is 1 when the app user typed in the drop-down component and 0 when the app user selected an option from the drop-down component.
`Source`	Component that executes the callback.
`EventName`	`'ValueChanged'`

For more information about writing callbacks, see "Write Callbacks in App Designer".

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

BusyAction — Callback queuing

'queue' (default) | 'cancel'

Callback queuing, specified as 'queue' or 'cancel'. The BusyAction property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The Interruptible property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the BusyAction property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the BusyAction property:

- 'queue' — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- 'cancel' — Does not execute the interrupting callback.

BeingDeleted — Deletion status

'off' | 'on'

This property is read-only.

Deletion status, returned as 'off' or 'on'. MATLAB sets the BeingDeleted property to 'on' when the DeleteFcn callback begins execution. The BeingDeleted property remains set to 'on' until the component object no longer exists.

Check the value of the BeingDeleted property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

Parent — Parent container

Figure object (default) | Panel object | Tab object | ButtonGroup object | GridLayout object

Parent container, specified as a Figure object created using the uifigure function, or one of its child containers: Tab, Panel, ButtonGroup, or GridLayout. If no container is specified, MATLAB calls the uifigure function to create a new Figure object that serves as the parent container.

HandleVisibility — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uidropdown'`

This property is read-only.

Type of graphics object, returned as `'uidropdown'`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
[ ] (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uidropdown` | `uifigure`

Introduced in R2016a

EditField Properties

Control edit field appearance and behavior

Description

Edit fields are UI components for entering text. Properties control the appearance and behavior of an edit field. Use dot notation to refer to a specific object and property.

```
uf = uifigure;
ef = uieditfield(uf);
ef.Value = 'New sample';
```

Properties

Text

`Value` — Text in edit field

`''` (default) | character vector | string scalar

Text in the edit field, specified as a character vector or string scalar. MATLAB displays the text as a single line. If you want to allow multiple lines of text, use a text area component instead of an edit field.

Example: `'Hello world'`

`HorizontalAlignment` — Horizontal alignment of text within edit field

`'left'` (default) | `'right'` | `'center'`

Alignment of text within the edit field, specified as `'left'`, `'right'`, or `'center'`. The alignment affects the display as the app user edits the edit field and how MATLAB displays the text in the app.

Font and Color

`FontName` — Font name

system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size
positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

`FontWeight` — Font weight
`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

`FontAngle` — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

`FontColor` — Font color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`BackgroundColor` — Background color

`[1 1 1]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Interactivity

`Visible` — Edit field visibility

`'on'` (default) | `'off'`

Edit field visibility, specified as `'on'` or `'off'`. The `Visible` property determines whether the edit field is displayed on the screen. If the `Visible` property is set to `'off'`, the edit field is hidden, but you can still specify and access its properties.

To make your app start faster, set the `Visible` property of all components that do not need to appear at startup to `'off'`.

`Editable` — Whether edit field is editable

`'on'` (default) | `'off'`

Whether the edit field is editable, specified as `'on'` or `'off'`.

Use this property in combination with the `Enable` property value to determine if and how the component responds to app user input:

- To make the edit field editable and the associated callback triggerable, set both the `Enable` property and the `Editable` property values to `'on'`.

- To make the edit field uneditable, but the text easy to read, set the `Enable` property value to `'on'` and the `Editable` property value to `'off'`.

- To make the edit field uneditable and the text dimmed, set both properties to `'off'`.

`Enable` — Operational state of edit field

`'on'` (default) | `'off'`

Operational state of the edit field, specified as `'on'` or `'off'`.

Use this property in combination with the `Editable` property value to determine if and how the component responds to app user input:

- To make the edit field editable and the associated callback triggerable, set both the `Enable` property and the `Editable` property values to `'on'`.

- To make the edit field uneditable, but the text easy to read, set the `Enable` property value to `'on'` and the `Editable` property value to `'off'`.

- To make the edit field uneditable and the text dimmed, set both properties to `'off'`.

`Tooltip` — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size of edit field

`[100 100 100 22]` (default) | `[left bottom width height]`

Location and size of the edit field relative to the parent, specified as the vector `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the edit field
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the edit field
`width`	Distance between the right and left outer edges of the edit field
`height`	Distance between the top and bottom outer edges of the edit field

All measurements are in pixel units.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[150 130 100 22]`

`InnerPosition` — Inner location and size of edit field
`[100 100 100 22]` (default) | `[left bottom width height]`

Inner location and size of the edit field, specified as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`OuterPosition` — Outer location and size of edit field
`[100 100 100 22]]` (default) | `[left bottom width height]`

This property is read-only.

Outer location and size of edit field returned as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then

this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a edit field in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
ef = uieditfield(g);
ef.Layout.Row = 3;
ef.Layout.Column = 2;
```

To make the edit field span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this edit field spans columns `2` through `3`:

```
ef.Layout.Column = [2 3];
```

Callbacks

`ValueChangedFcn` — Value changed callback

`''` (default) | function handle | cell array | character vector

Value changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

The callback executes when the user changes text in the edit field and either presses **Enter** or clicks outside the edit field. It does not execute if the edit field value changes programmatically.

This callback function can access specific information about the user's interaction with the edit field. MATLAB passes this information in a `ValueChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.PreviousValue` returns the previous value of the edit field. The `ValueChangedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangedData` object.

Property	Value
`Value`	Value of edit field after app user's most recent interaction with it
`PreviousValue`	Value of edit field before app user's most recent interaction with it
`Source`	Component that executes the callback
`EventName`	`'ValueChanged'`

For more information about writing callbacks, see "Write Callbacks in App Designer".

`ValueChangingFcn` — Value changing callback
`''` (default) | function handle | cell array | character vector

Value changing callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes as follows:

- As the user types in the edit field, the callback executes repeatedly.
- When the user presses **Enter**, the callback executes.

If the edit field value changes programmatically, then the callback does not execute.

This callback function can access specific information about the user's interaction with the edit field. MATLAB passes this information in a `ValueChangingData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.Value` is the value in the edit field that triggered the execution of the callback. The `ValueChangingData` object is not available to callback functions specified as character vectors.

Here are the properties of the `ValueChangingData` object:

Property	Description
`Value`	Value that triggered the execution of the callback
`Source`	Component that executes the callback
`EventName`	`'ValueChanging'`

The `Value` property of the `EditField` object is not updated until the user presses the **Enter** key. However, you can get the text that the user is typing before they press **Enter** by querying the `Value` property of the `ValueChangingData` object.

For more information about writing callbacks, see “Write Callbacks in App Designer”.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.

  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.

HandleVisibility Value	Description
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uieditfield'`

This property is read-only.

Type of graphics object, returned as `'uieditfield'`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uieditfield` | `uifigure`

Introduced in R2016a

Gauge Properties

Control gauge appearance and behavior

Description

Gauges are app components that represent measurement instruments. Properties control the appearance and behavior of a gauge. Use dot notation to refer to a particular object and property:

```
uf = uifigure;
g = uigauge(uf);
g.Value = 45;
```

Properties

Gauge

`Value` — Location of gauge needle

0 (default) | numeric

Location of the gauge needle, specified as any numeric value.

- If the value is less than the minimum `Limits` property value, then the needle points to a location immediately before the beginning of the scale.
- If the value is more than the maximum `Limits` property value, then the needle points to a location immediately after the end of the scale.

Changing the `Limits` property value has no effect on the `Value` property setting.

Example: `60`

`Limits` — Minimum and maximum gauge scale values

[0 100] (default) | two-element numeric array

Minimum and maximum gauge scale values, specified as a two-element numeric array. The first value in the array must be less than the second value.

If you change `Limits` such that the `Value` property is less than the new lower limit, or more than the new upper limit, then the gauge needle points to a location off the scale.

For example, suppose `Limits` is `[0 100]` and the `Value` property is 20. If the `Limits` changes to `[50 100]`, then the needle points to a location off the scale, slightly less than 50.

`ScaleDirection` — Direction of gauge scale

`'clockwise'` (default) | `'counterclockwise'`

Direction of the gauge scale, specified as one of the following:

- `'clockwise'` — The scale appears such that the scale tick values increase in a clockwise manner.
- `'counterclockwise'` — The scale appears such that the scale tick values increase in a counterclockwise manner.

`ScaleColors` — Scale colors

[ ] (default) | n-by-3 array of RGB triplets | cell array

Scale colors, specified one of the following arrays:

- An n-by-3 array of RGB triplets
- A cell array containing RGB triplets, any of the color options listed in the table below, or a combination of both.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`

Option	Description	Equivalent RGB Triplet
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`

Each color of the `ScaleColors` array corresponds to a colored section of the gauge. Set the `ScaleColorLimits` property to map the colors to specific sections of the gauge.

If you do not set the `ScaleColorLimits` property, MATLAB distributes the colors equally over the range of the gauge.

`ScaleColorLimits` — Scale color limits
[ ] (default) | n-by-2 array

Scale color limits, specified as a n-by-2 array of numeric values. For every row in the array, the first element must be less than the second element.

When applying colors to the gauge, MATLAB applies the colors starting with the first color in the `ScaleColors` array. Therefore, if two rows in `ScaleColorLimits` array overlap, then the color applied later takes precedence.

The gauge does not display any portion of the `ScaleColorLimits` that fall outside of the `Limits` property.

If the `ScaleColors` and `ScaleColorLimits` property values are different sizes, then the gauge shows only the colors that have matching limits. For example, if the `ScaleColors` array has three colors, but the `ScaleColorLimits` has only two rows, then the gauge displays the first two color/limit pairs only.

Ticks

`MajorTicks` — Major tick mark locations
`[0 20 40 60 80 100]` (default) | vector of numeric values | `[]`

Major tick mark locations, specified as a vector of numeric values or an empty vector. If you do not want to show major tick marks, specify this property as an empty vector.

Tick locations that are outside the range of the `Limits` property do not display.

MATLAB removes duplicate tick values. However, if a major tick falls on the same value as a minor tick, only the major tick displays.

Setting the `MajorTicks` property sets the `MajorTicksMode` property to `'manual'`.

`MajorTicksMode` — Major tick creation mode
`'auto'` (default) | `'manual'`

Major tick creation mode, specified as one of the following:

- `'auto'` — MATLAB determines the placement of major ticks.
- `'manual'` — You specify the `MajorTicks` value array.

`MajorTickLabels` — Major tick labels
`{'0','20','40','60','80','100'}` (default) | cell array of character vectors | string array | `{}` | ...

Major tick labels, specified as a cell array of character vectors, string array, or 1-D categorical array. If you do not want to show tick labels, specify this property as an empty cell array. If you want to remove a label from a specific tick mark, specify an empty character vector or empty string scalar for the corresponding element in the `MajorTickLabels` array. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

If the length of the `MajorTickLabels` array is different from the length of the `MajorTicks` vector, MATLAB ignores the extra entries of the longer array. If there are extra labels, they are ignored. If there are extra tick marks, they display without labels.

Setting `MajorTickLabels` changes the `MajorTickLabelsMode` value to `'manual'`.

`MajorTickLabelsMode` — Major tick labels mode
`'auto'` (default) | `'manual'`

Major tick labels mode, specified as one of the following:

- `'auto'` — MATLAB specifies the major tick labels.
- `'manual'` — You specify the major tick labels using the `MajorTickLabels` property.

`MinorTicks` — Minor tick mark locations
`[0 4 8 12 ... 100]` (default) | vector of numeric values | `[]`

Minor tick mark locations, specified as a vector of numeric values or an empty vector. If you do not want to show minor tick marks, specify this property as an empty vector.

Tick locations that are outside the range of the `Limits` property do not display.

MATLAB removes duplicate tick values. However, if a minor tick falls on the same value as a major tick, only the major tick displays.

Setting the `MinorTicks` property value sets the `MinorTicksMode` property value to `'manual'`.

`MinorTicksMode` — Minor tick creation mode
`'auto'` (default) | `'manual'`

Minor tick creation mode, specified as one of the following:

- `'auto'` — MATLAB determines the placement of minor ticks. MATLAB does not generate minor ticks for major ticks that are beyond scale limits. If the `Limits` property value changes, then MATLAB updates minor ticks to populate the full scale range (the `MinorTicks` property is updated accordingly).
- `'manual'` — You specify the `MinorTicks` property numeric array. The `MinorTicks` property value does not change size or content on its own.

Font and Color

`FontName` — Font name
system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size
positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

`FontWeight` — Font weight
`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

`FontAngle` — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

`FontColor` — Font color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`BackgroundColor` — Background color

`[1 1 1]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Interactivity

`Visible` — Visibility of gauge

`'on'` (default) | `'off'`

Visibility of the gauge, specified as `'on'` or `'off'`. The `Visible` property determines whether the gauge is displayed on the screen. If the `Visible` property is set to `'off'`, then the entire gauge is hidden, but you can still specify and access its properties.

To make your app start faster, set the `Visible` property of all components that do not need to appear at startup to `'off'`.

Enable — Operational state of gauge

`'on'` (default) | `'off'`

Operational state of gauge, specified as `'on'` or `'off'`.

- If you set this property to `'on'`, then the appearance of the gauge indicates that the gauge is operational.
- If you set this property to `'off'`, then the appearance of the gauge appears dimmed, indicating that the gauge is not operational.

Tooltip — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

Position — Location and size of gauge

`[100 100 120 120]` (default) | `[left bottom width height]`

Location and size of the gauge relative to the parent container, specified as the vector, `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of an imaginary box surrounding the gauge
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of an imaginary box surrounding the gauge
`width`	Distance between the right and left outer edges of the gauge

Element	Description
`height`	Distance between the top and bottom outer edges of the gauge

All measurements are in pixel units.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[200 120 120 120]`

`InnerPosition` — Inner location and size of gauge
`[100 100 120 120]` (default) | `[left bottom width height]`

Inner location and size of the gauge, specified as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`OuterPosition` — Outer location and size of gauge
`[100 100 120 120]]` (default) | `[left bottom width height]`

This property is read-only.

Outer location and size of the gauge returned as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a gauge in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
gauge = uigauge(g);
```

```
gauge.Layout.Row = 3;
gauge.Layout.Column = 2;
```

To make the gauge span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this gauge spans columns `2` through `3`:

```
gauge.Layout.Column = [2 3];
```

Callbacks

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.

  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.

- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.

HandleVisibility Value	Description
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uigauge'`

This property is read-only.

Type of graphics object, returned as `'uigauge'`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uifigure` | `uigauge`

Introduced in R2016a

Knob Properties

Control knob appearance and behavior

Description

Knobs are UI components representing instrument control knobs that the user can adjust to control a value. Properties control the appearance and behavior of a knob. Use dot notation to refer to a particular object and property:

```
uf = uifigure;
k = uiknob(uf);
k.Value = 45;
```

Properties

Knob

`Value` — Value of knob
0 (default) | numeric

Value of the knob, specified as a numeric. The numeric must be within the range specified for `Limits`.

`Limits` — Minimum and maximum knob values
[0 100] (default) | two-element numeric array

Minimum and maximum knob values, specified as a two-element numeric array. The first value in the array must be less than the second value.

If you change `Limits` such that the `Value` property is less than the new lower limit, then the `Value` property updates to the new lower limit. Similarly, if you change `Limits` such that the `Value` is greater than the new upper limit, then `Value` property updates to the new upper limit.

For example, suppose `Limits` is `[0 100]` and the `Value` property is `20`. If the `Limits` property changes to `[50 100]`, then the `Value` updates to `50`.

Ticks

`MajorTicks` — Major tick mark locations
`[0 10 20 ... 100]` (default) | vector of numeric values | `[]`

Major tick mark locations, specified as a vector of numeric values or an empty vector. If you do not want to show major tick marks, specify this property as an empty vector.

Tick locations that are outside the range of the `Limits` property do not display.

MATLAB removes duplicate tick values. However, if a major tick falls on the same value as a minor tick, only the major tick displays.

Setting the `MajorTicks` property sets the `MajorTicksMode` property to `'manual'`.

`MajorTicksMode` — Major tick creation mode
`'auto'` (default) | `'manual'`

Major tick creation mode, specified as one of the following:

- `'auto'` — MATLAB determines the placement of major ticks.
- `'manual'` — You specify the `MajorTicks` value array.

`MajorTickLabels` — Major tick labels
`{'0','10','20',...,'100'}` (default) | cell array of character vectors | string array | `{}` | ...

Major tick labels, specified as a cell array of character vectors, string array, or 1-D categorical array. If you do not want to show tick labels, specify this property as an empty cell array. If you want to remove a label from a specific tick mark, specify an empty character vector or empty string scalar for the corresponding element in the `MajorTickLabels` property. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

If the length of the `MajorTickLabels` array is different from the length of the `MajorTicks` vector, MATLAB ignores the extra entries of the longer array. If there are extra labels, they are ignored. If there are extra tick marks, they display without labels.

Setting `MajorTickLabels` changes the `MajorTickLabelsMode` value to `'manual'`.

`MajorTickLabelsMode` — Major tick labels mode
`'auto'` (default) | `'manual'`

Major tick labels mode, specified as one of the following:

- `'auto'` — MATLAB specifies the major tick labels.
- `'manual'` — You specify the major tick labels using the `MajorTickLabels` property.

`MinorTicks` — Minor tick mark locations
`[0 2.5 5 7.5 ... 100]` (default) | numeric array

Minor tick mark locations, specified as a vector of numeric values or an empty vector. If you do not want to show minor tick marks, specify this property as an empty vector.

Tick locations that are outside the range of the `Limits` property do not display.

MATLAB removes duplicate tick values. However, if a minor tick falls on the same value as a major tick, only the major tick displays.

Setting the `MinorTicks` property value sets the `MinorTicksMode` property value to `'manual'`.

`MinorTicksMode` — Minor tick creation mode
`'auto'` (default) | `'manual'`

Minor tick creation mode, specified as one of the following:

- `'auto'` — MATLAB determines the placement of minor ticks. MATLAB does not generate minor ticks for major ticks that are beyond scale limits. If the `Limits` property value changes, then MATLAB updates minor ticks to populate the full scale range (the `MinorTicks` property is updated accordingly).
- `'manual'` — You specify the `MinorTicks` property numeric array. The `MinorTicks` property value does not change size or content on its own.

Font

`FontName` — Font name
system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size
positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

`FontWeight` — Font weight
`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

`FontAngle` — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

`FontColor` — Font color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Interactivity

`Visible` — Visibility of knob

`'on'` (default) | `'off'`

Visibility of the knob, specified as `'on'` or `'off'`. The `Visible` property determines whether the knob is displayed on the screen. If the `Visible` property is set to `'off'`, the entire knob is hidden, but you can still specify and access its properties.

To make your app start faster, set the `Visible` property of all components that do not need to appear at startup to `'off'`.

Enable — Operational state of knob

`'on'` (default) | `'off'`

Operational state of the knob, specified as `'on'` or `'off'`.

- If you set this property to `'on'`, then the app user can turn the knob.
- If you set this property to `'off'`, then the appearance of the knob appears dimmed, indicating that the app user cannot turn the knob and it will not trigger a callback.

Tooltip — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

Position — Location and size of knob

`[100 100 60 60]` (default) | `[left bottom width height]`

Location and size of the knob, excluding tick marks and labels, specified as the vector, `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the knob
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the knob
`width`	Distance between the right and left outer edges of the knob, excluding tick marks and labels
`height`	Distance between the top and bottom outer edges of the knob, excluding tick marks and labels

All measurements are in pixel units. Due to aspect ratio constraints, you cannot change the knob height and width independently of one another. To increase the knob size, set width and height using `Position(3:4) = [width height]`.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[100 200 60 60`

`InnerPosition` — Inner location and size of knob
`[100 100 60 60]` (default) | `[left bottom width height]`

Inner location and size of the knob, excluding tick marks and tick labels, specified as the vector, `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`OuterPosition` — Outer location and size of knob
`[73 80 115 105]` (default) | `[left bottom width height]`

This property is read-only.

Outer location and size of knob, including tick marks and tick labels, specified as the vector, `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a knob in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
k = uiknob(g);
```

```
k.Layout.Row = 3;
k.Layout.Column = 2;
```

To make the knob span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this knob spans columns `2` through `3`:

```
k.Layout.Column = [2 3];
```

Callbacks

`ValueChangedFcn` — Value changed callback

`''` (default) | function handle | cell array | character vector

Value changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when the user changes the knob selector to point to a different value.

This callback function can access specific information about the user's interaction with the knob. MATLAB passes this information in a `ValueChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.PreviousValue` returns the previous value of the knob. The `ValueChangedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangedData` object.

Property	Value
`Value`	Value of knob after app user's most recent interaction with it.
`PreviousValue`	Value of knob before app user's most recent interaction with it.
`Source`	Component that executes the callback.
`EventName`	`'ValueChanged'`

For more information about writing callbacks, see “Write Callbacks in App Designer”.

`ValueChangingFcn` — Value changing callback
`''` (default) | function handle | cell array | character vector

Value changing callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes as the user turns the knob in the app. It does not execute if the `Value` property changes programmatically.

This callback function can access specific information about the user’s interaction with the knob. MATLAB passes this information in a `ValueChangingData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.Value` returns the current value of the knob. The `ValueChangingData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangingData` object.

Property	Value
`Value`	Current value of the knob as the app user is interacting with it.
`Source`	Component that executes the callback.
`EventName`	`'ValueChanging'`

The `Value` property of the `Knob` object is not updated until the app user releases the knob. Therefore, to get the knob values as the knob is being turned, your code must get the `Value` property of the `ValueChangingData` object.

The callback executes as follows:

- If the app user clicks the knob value, the callback executes once. For example, if the knob is on 1.0, and the app user single-clicks 1.1, the callback executes.

- If the app user clicks and drags the knob to a new position, the callback executes repeatedly. For example, if the knob value is 1.0, and the app user clicks, holds, and drags to value 10.0, the callback executes multiple times until the app user releases the knob.

For more information about writing callbacks, see “Write Callbacks in App Designer”.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.

  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.

- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.

HandleVisibility Value	Description
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uiknob'`

This property is read-only.

Type of graphics object, returned as `'uiknob'`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uifigure` | `uiknob`

Introduced in R2016a

Label Properties

Control label appearance

Description

Labels are UI components that contain static text for labelling parts of an app. Properties control the appearance and behavior of a label. Use dot notation to refer to a specific object and property.

```
uf = uifigure;
tlabel = uilabel(uf);
tlabel.Text = 'Options';
```

Properties

Text

`Text` — Label text

`'Label'` (default) | character vector | cell array of character vectors | string scalar | string array | ...

Label text, specified as a character vector, cell array of character vectors, string scalar, string array, or 1-D categorical array. Use a cell array of character vectors or a string array to specify multiple lines of text.

Alternatively, use the `sprintf` function to create formatted text containing line breaks and other special characters.

```
text = sprintf('%s\n%s','Line 1','Line 2');
label = uilabel('Text',text,'Position',[100 100 100 32]);
```

If you specify text as a character vector without using `sprintf`, MATLAB will not interpret control sequences such as `\n`.

If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Example: `'Threshold'`

Example: `{'Threshold' 'Value'}`

`HorizontalAlignment` — Horizontal alignment of text
`'left'` (default) | `'right'` | `'center'`

Horizontal alignment of the text, specified as:

- `'right'` — Text aligns on the right side of the area specified by the `Position` property.
- `'left'` — Text aligns on the left side of the area specified by the `Position` property.
- `'center'`— Text centers horizontally in the area specified by the `Position` property.

Aligning label text is useful when the text spans multiple lines.

`VerticalAlignment` — Vertical alignment of text
`'center'` (default) | `'top'` | `'bottom'`

Vertical alignment of the text, specified as one of the following:

- `'center'` — Text centers vertically in the area specified by the `Position` property.
- `'top'` — Text aligns on the top of the area specified by the `Position` property.
- `'bottom'` — Text aligns on the bottom of the area specified by the `Position` property.

Aligning label text is useful when the text spans multiple lines.

Font and Color

`FontName` — Font name
system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size
positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

`FontWeight` — Font weight
`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

`FontAngle` — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

`FontColor` — Font color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`BackgroundColor` — Background color

`'none'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Data Types: `double` | `char`

Interactivity

`Visible` — Label visibility

`'on'` (default) | `'off'`

Label visibility, specified as `'on'` or `'off'`. The `Visible` property determines whether the label is displayed on the screen. If the `Visible` property is set to `'off'`, the entire label is hidden, but you can still specify and access its properties.

To make your app start faster, set the `Visible` property of all components that do not need to appear at startup to `'off'`.

`Enable` — Visual appearance of label

`'on'` (default) | `'off'`

Visual appearance of the label, specified as one of the following:

- `'on'` — Label appears normal

  Label

- `'off'` — Label appears dimmed

  Label

`Tooltip` — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Label location and size

`[100 100 31 22]` (default) | `[left bottom width height]`

Label location and size, relative to the parent, specified as the vector `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the label
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the label
`width`	Distance between the right and left outer edges of the label
`height`	Distance between the top and bottom outer edges of the label

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

All measurements are in pixel units.

Example: `[100 100 100 20]`

`InnerPosition` — Inner location and size of label
`[100 100 31 22]` (default) | `[left bottom width height]`

Inner location and size of label, specified as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to `Position` property.

`OuterPosition` — Outer location and size of label
`[100 100 31 22]` (default) | `[left bottom width height]`

This property is read-only.

Outer location and size of label, returned as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to `Position`.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is

not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a label in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
tlabel = uilabel(g);
tlabel.Layout.Row = 3;
tlabel.Layout.Column = 2;
```

To make the label span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this label spans columns `2` through `3`:

```
tlabel.Layout.Column = [2 3];
```

Callbacks

`CreateFcn` — Creation function

`' '` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running

callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that

obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object
`'uilabel'`

This property is read-only.

Type of graphics object, returned as `'uilabel'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uifigure` | `uilabel`

Introduced in R2016a

Lamp Properties

Control lamp appearance

Description

Lamps are app components that indicate state using color. Properties control the appearance and behavior of a lamp. Use dot notation to refer to a specific object and property.

```
uf = uifigure;
mylamp = uilamp(uf);
mylamp.Color = 'red';
```

Properties

Color

`Color` — Color of lamp

`[0 1 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of the lamp, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Data Types: `double` | `char`

Interactivity

`Visible` — Visibility of lamp

`'on'` (default) | `'off'`

Visibility of the lamp, specified as `'on'` or `'off'`. The `Visible` property determines whether the lamp is displayed on the screen. If the `Visible` property of a lamp is set to `'off'`, the entire lamp is hidden, but you can still specify and access its properties.

To make your app start faster, set the `Visible` property of all components that do not need to appear at startup to `'off'`.

Enable — Operational state of the lamp

`'on'` (default) | `'off'`

Operational state of the lamp, specified as `'on'` or `'off'`.

- If you set this property to `'on'`, then the lamp appears bright, indicating that the lamp is operational.
- If you set this property to `'off'`, then the lamp appears dimmed.

Tooltip — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

Position — Location and size of lamp

`[100 100 20 20]` (default) | `[left bottom width height]`

Location and size of the lamp, specified as the vector `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the lamp
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the lamp
`width`	Distance between the right and left outer edges of the lamp
`height`	Distance between the top and bottom outer edges of the lamp

All measurements are in pixel units.

Due to aspect ratio constraints, you cannot change the lamp height and width independently of one another.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[100 200 20 20]`

`InnerPosition` — Inner location and size of lamp
`[100 100 20 20]` (default) | `[left bottom width height]`

Inner location and size of the lamp, specified as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`OuterPosition` — Outer location and size of lamp
`[100 100 20 20]` (default) | `[left bottom width height]`

This property is read-only.

Outer location and size of the lamp, returned as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a lamp in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
mylamp = uilamp(g);
mylamp.Layout.Row = 3;
mylamp.Layout.Column = 2;
```

To make the lamp span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this lamp spans columns `2` through `3`:

```
mylamp.Layout.Column = [2 3];
```

Callbacks

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.

- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.

HandleVisibility Value	Description
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uilamp'`

This property is read-only.

Type of graphics object, returned as `'uilamp'`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uifigure` | `uilamp`

Introduced in R2016a

LinearGauge Properties

Control linear gauge appearance and behavior

Description

A linear gauge is a UI component that represents a measurement instrument. Properties control the appearance and behavior of a linear gauge. Use dot notation to refer to a specific object and property.

```
uf = uifigure;
g = uigauge(uf,'linear');
g.Value = 45;
```

Properties

Gauge

`Value` — Location of gauge needle

0 (default) | numeric

Location of the gauge needle, specified as any numeric value.

- If the value is less than the minimum `Limits` property value, then the needle points to a location immediately before the beginning of the scale.
- If the value is more than the maximum `Limits` property value, then the needle points to a location immediately after the end of the scale.

Changing the `Limits` property value has no effect on the `Value` property setting.

Example: `60`

`Limits` — Minimum and maximum gauge scale values

[0 100] (default) | two-element numeric array

Minimum and maximum gauge scale values, specified as a two-element numeric array. The first value in the array must be less than the second value.

If you change `Limits` such that the `Value` property is less than the new lower limit, or more than the new upper limit, then the gauge needle points to a location off the scale.

For example, suppose `Limits` is `[0 100]` and the `Value` property is 20. If the `Limits` changes to `[50 100]`, then the needle points to a location off the scale, slightly less than 50.

`Orientation` — Orientation

`'horizontal'` (default) | `'vertical'`

Orientation of the gauge, specified as `'horizontal'` or `'vertical'`.

`ScaleColors` — Scale colors

[ ] (default) | n-by-3 array of RGB triplets | cell array

Scale colors, specified one of the following arrays:

- An n-by-3 array of RGB triplets
- A cell array containing RGB triplets, any of the color options listed in the table below, or a combination of both.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`

Each color of the `ScaleColors` array corresponds to a colored section of the gauge. Set the `ScaleColorLimits` property to map the colors to specific sections of the gauge.

If you do not set the `ScaleColorLimits` property, MATLAB distributes the colors equally over the range of the gauge.

`ScaleColorLimits` — Scale color limits

[ ] (default) | n-by-2 array

Scale color limits, specified as a n-by-2 array of numeric values. For every row in the array, the first element must be less than the second element.

When applying colors to the gauge, MATLAB applies the colors starting with the first color in the `ScaleColors` array. Therefore, if two rows in `ScaleColorLimits` array overlap, then the color applied later takes precedence.

The gauge does not display any portion of the `ScaleColorLimits` that fall outside of the `Limits` property.

If the `ScaleColors` and `ScaleColorLimits` property values are different sizes, then the gauge shows only the colors that have matching limits. For example, if the `ScaleColors` array has three colors, but the `ScaleColorLimits` has only two rows, then the gauge displays the first two color/limit pairs only.

Ticks

`MajorTicks` — Major tick mark locations

`[0 20 40 60 80 100]` (default) | vector of numeric values | `[]`

Major tick mark locations, specified as a vector of numeric values or an empty vector. If you do not want to show major tick marks, specify this property as an empty vector.

Tick locations that are outside the range of the `Limits` property do not display.

MATLAB removes duplicate tick values. However, if a major tick falls on the same value as a minor tick, only the major tick displays.

Setting the `MajorTicks` property sets the `MajorTicksMode` property to `'manual'`.

`MajorTicksMode` — Major tick creation mode

`'auto'` (default) | `'manual'`

Major tick creation mode, specified as one of the following:

- `'auto'` — MATLAB determines the placement of major ticks.

- `'manual'` — You specify the `MajorTicks` value array.

`MajorTickLabels` — Major tick labels
`{'0','20','40','60','80','100'}` (default) | cell array of character vectors | string array | `{}` | ...

Major tick labels, specified as a cell array of character vectors, string array, or 1-D categorical array. If you do not want to show tick labels, specify this property as an empty cell array. If you want to remove a label from a specific tick mark, specify an empty character vector or empty string scalar for the corresponding element in the `MajorTickLabels` array. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

If the length of the `MajorTickLabels` array is different from the length of the `MajorTicks` vector, MATLAB ignores the extra entries of the longer array. If there are extra labels, they are ignored. If there are extra tick marks, they display without labels.

Setting `MajorTickLabels` changes the `MajorTickLabelsMode` value to `'manual'`.

`MajorTickLabelsMode` — Major tick labels mode
`'auto'` (default) | `'manual'`

Major tick labels mode, specified as one of the following:

- `'auto'` — MATLAB specifies the major tick labels.
- `'manual'` — You specify the major tick labels using the `MajorTickLabels` property.

`MinorTicks` — Minor tick mark locations
`[0 4 8 12 ... 100]` (default) | vector of numeric values | `[]`

Minor tick mark locations, specified as a vector of numeric values or an empty vector. If you do not want to show minor tick marks, specify this property as an empty vector.

Tick locations that are outside the range of the `Limits` property do not display.

MATLAB removes duplicate tick values. However, if a minor tick falls on the same value as a major tick, only the major tick displays.

Setting the `MinorTicks` property value sets the `MinorTicksMode` property value to `'manual'`.

`MinorTicksMode` — Minor tick creation mode
`'auto'` (default) | `'manual'`

Minor tick creation mode, specified as one of the following:

- `'auto'` — MATLAB determines the placement of minor ticks. MATLAB does not generate minor ticks for major ticks that are beyond scale limits. If the `Limits` property value changes, then MATLAB updates minor ticks to populate the full scale range (the `MinorTicks` property is updated accordingly).
- `'manual'` — You specify the `MinorTicks` property numeric array. The `MinorTicks` property value does not change size or content on its own.

Font and Color

`FontName` — Font name
system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size
positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

`FontWeight` — Font weight
`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

`FontAngle` — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

`FontColor` — Font color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

BackgroundColor — Background color

[1 1 1] (default) | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Interactivity

`Visible` — Visibility of gauge

`'on'` (default) | `'off'`

Visibility of the gauge, specified as `'on'` or `'off'`. The `Visible` property determines whether the gauge is displayed on the screen. If the `Visible` property is set to `'off'`, then the entire gauge is hidden, but you can still specify and access its properties.

To make your app start faster, set the `Visible` property of all components that do not need to appear at startup to `'off'`.

`Enable` — Operational state of gauge

`'on'` (default) | `'off'`

Operational state of gauge, specified as `'on'` or `'off'`.

- If you set this property to `'on'`, then the appearance of the gauge indicates that the gauge is operational.

- If you set this property to `'off'`, then the appearance of the gauge appears dimmed, indicating that the gauge is not operational.

`Tooltip` — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Gauge location and size

`[100 100 120 40]` (default) | `[left bottom width height]`

Gauge location and size, specified as the vector, `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the gauge
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the gauge
`width`	Distance between the right and left outer edges of the gauge
`height`	Distance between the top and bottom outer edges of the gauge

All measurements are in pixel units.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[100 100 100 20]`

InnerPosition — Inner location and size of gauge

[100 100 120 40] (default) | [left bottom width height]

Inner location and size of the gauge, specified as [left bottom width height]. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the Position property.

OuterPosition — Outer location and size of gauge

[100 100 120 40]] (default) | [left bottom width height]

This property is read-only.

Outer location and size of the gauge, returned as [left bottom width height]. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the Position property.

Layout — Layout options

empty LayoutOptions array (default) | GridLayoutOptions object

Layout options, specified as a GridLayoutOptions object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the Row and Column properties on the GridLayoutOptions object.

For example, this code places a linear gauge in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
gauge = uigauge(g,'linear');
gauge.Layout.Row = 3;
gauge.Layout.Column = 2;
```

To make the gauge span multiple rows or columns, specify the Row or Column property as a two-element vector. For example, this gauge spans columns 2 through 3:

```
gauge.Layout.Column = [2 3];
```

Callbacks

CreateFcn — Creation function

'' (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.

- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

Parent — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

HandleVisibility — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uilineargauge'`

This property is read-only.

Type of graphics object, returned as `'uilineargauge'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uifigure` | `uigauge`

Introduced in R2016a

ListBox Properties

Control list box appearance and behavior

Description

List boxes are UI components for displaying items in a list. Properties control the appearance and behavior of a list box. Use dot notation to refer to a particular object and property:

```
uf = uifigure;
list = uilistbox(uf);
list.Items = {'Red','Green','Blue'};
```

Properties

List Box

`Value` — Value

element of `Items` | element of `ItemsData` | `{}`

Value, specified as an element of the `Items` array, `ItemsData` array, or an empty cell array. By default, `Value` is the first element in `Items`.

To specify no selection, set `Value` to an empty cell array.

Specifying `Value` as an element of `Items` selects the list item that matches that element. If `ItemsData` is not empty, then `Value` must be set to an element of `ItemsData`, and the list box will select the associated item in the list.

`Items` — List box items

`{'Item 1','Item 2', 'Item 3', 'Item 4'}` (default) | 1-by-n cell array of character vectors | string array | ...

List box items, specified as a cell array of character vectors, string array, or 1-D categorical array. Duplicate elements are allowed. The list box displays as many options as there are elements in the `Items` array. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

`ItemsData` — Data associated with each element of the `Items` property value
empty array (`[]`) (default) | 1-by-n numeric array | 1-by-n cell array

Data associated with each element of the `Items` property value, specified as a 1-by-n numeric array or a 1-by-n cell array. Duplicate elements are allowed.

For example, if you set the `Items` value to employee names, you might set the `ItemsData` value to corresponding employee ID numbers. The `ItemsData` value is not visible to the app user.

If the number of array elements in the `ItemsData` value and the `Items` value do not match, one of the following occurs:

- When the `ItemsData` value is empty, then all the elements of the `Items` value are presented to the app user.
- When the `ItemsData` value has more elements than the `Items` value, then all the elements of the `Items` value are presented to the app user. MATLAB ignores the extra `ItemsData` elements.
- When the `ItemsData` value is not empty, but has fewer elements than the `Items` value, the only elements of the `Items` value presented to the app user are those that have a corresponding element in the `ItemsData` value.

Example: `{'One','Two','Three'}`

Example: `[10 20 30 40]`

Font and Color

`FontName` — Font name
system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size
positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

`FontWeight` — Font weight
`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

`FontAngle` — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

`FontColor` — Font color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`BackgroundColor` — Background color

`[1 1 1]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Interactivity

`Visible` — List box visibility

`'on'` (default) | `'off'`

List box visibility, specified as `'on'` or `'off'`. The `Visible` property determines whether the list box is displayed on the screen. If the `Visible` property is set to `'off'`, the entire list box is hidden, but you can still specify and access its properties.

To make your app start faster, set the `Visible` property of all components that do not need to appear at startup to `'off'`.

`Multiselect` — Multiple node selection
`'off'` (default) | `'on'`

Multiple node selection, specified as `'off'` or `'on'`. Set this property to `'on'` to allow users to select multiple items simultaneously.

`Enable` — Operational state of list box
`'on'` (default) | `'off'`

Operational state of list box, specified as `'on'` or `'off'`. Set this property to `'off'` to make the list box appear dim, indicating that the user cannot interact with the list box.

`Tooltip` — Tooltip
`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size of list box
`[100 100 100 74]` (default) | `[left bottom width height]`

Location and size of the list box relative to the parent container, specified as the vector `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the list box
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the list box
`width`	Distance between the right and left outer edges of the list box
`height`	Distance between the top and bottom outer edges of the list box

All measurements are in pixel units.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[100 100 100 200]`

`InnerPosition` — Inner location and size of list box
`[100 100 77 78]` (default) | `[left bottom width height]`

Inner location and size of list box, specified as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`OuterPosition` — Outer location and size of list box
`[100 100 77 78]]` (default) | `[left bottom width height]`

This property is read-only.

Outer location and size of list box returned as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then

this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a list box in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
list = uilistbox(g);
list.Layout.Row = 3;
list.Layout.Column = 2;
```

To make the list box span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this list box spans columns `2` through `3`:

```
list.Layout.Column = [2 3];
```

Callbacks

`ValueChangedFcn` — Value changed function

`''` (default) | function handle | cell array | character vector

Value changed function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback function executes when the user selects a different item in the list box. It does not execute if the `Value` property setting changes programmatically.

This callback function can access specific information about the user's interaction with the list box. MATLAB passes this information in a `ValueChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.PreviousValue` returns the previous value of the list box. The `ValueChangedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangedData` object.

Property	Value
`Value`	Value of list box after app user's most recent interaction with it
`PreviousValue`	Value of list box before app user's most recent interaction with it
`Source`	Component that executes the callback
`EventName`	`'ValueChanged'`

For more information about writing callbacks, see "Write Callbacks in App Designer".

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see "Write Callbacks in App Designer".

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback.

MATLAB resumes executing the running callback when the interrupting callback completes.

- If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.

- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.

HandleVisibility Value	Description
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uilistbox'`

This property is read-only.

Type of graphics object, returned as `'uilistbox'`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`appdesigner` | `scroll` | `uifigure` | `uilistbox`

Introduced in R2016a

NinetyDegreeGauge Properties

Control ninety-degree gauge appearance and behavior

Description

A ninety-degree gauge is a UI component that represents a measurement instrument. Properties control the appearance and behavior of a ninety-degree gauge. Use dot notation to refer to a specific object and property.

```
uf = uifigure;
g = uigauge(uf,'ninetydegree');
g.Value = 45;
```

Properties

Gauge

`Value` — Location of gauge needle

0 (default) | numeric

Location of the gauge needle, specified as any numeric value.

- If the value is less than the minimum `Limits` property value, then the needle points to a location immediately before the beginning of the scale.
- If the value is more than the maximum `Limits` property value, then the needle points to a location immediately after the end of the scale.

Changing the `Limits` property value has no effect on the `Value` property setting.

Example: `60`

`Limits` — Minimum and maximum gauge scale values

[0 100] (default) | two-element numeric array

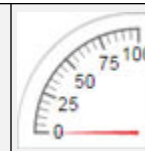
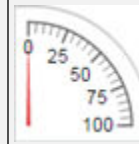
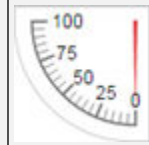
Minimum and maximum gauge scale values, specified as a two-element numeric array. The first value in the array must be less than the second value.

If you change `Limits` such that the `Value` property is less than the new lower limit, or more than the new upper limit, then the gauge needle points to a location off the scale.

For example, suppose `Limits` is `[0 100]` and the `Value` property is 20. If the `Limits` changes to `[50 100]`, then the needle points to a location off the scale, slightly less than 50.

`Orientation` — Orientation
`'northwest'` (default) | `'northeast'` | `'southwest'` | `'southeast'`

Orientation of the gauge, specified as one of the following positions:

`'northwest'`	
`'northeast'`	
`'southwest'`	
`'southeast'`	

`ScaleDirection` — Direction of gauge scale
`'clockwise'` (default) | `'counterclockwise'`

Direction of the gauge scale, specified as one of the following:

- `'clockwise'` — The scale appears such that the scale tick values increase in a clockwise manner.
- `'counterclockwise'` — The scale appears such that the scale tick values increase in a counterclockwise manner.

`ScaleColors` — Scale colors

[ ] (default) | n-by-3 array of RGB triplets | cell array

Scale colors, specified one of the following arrays:

- An n-by-3 array of RGB triplets
- A cell array containing RGB triplets, any of the color options listed in the table below, or a combination of both.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`

Each color of the `ScaleColors` array corresponds to a colored section of the gauge. Set the `ScaleColorLimits` property to map the colors to specific sections of the gauge.

If you do not set the `ScaleColorLimits` property, MATLAB distributes the colors equally over the range of the gauge.

`ScaleColorLimits` — Scale color limits

[ ] (default) | n-by-2 array

Scale color limits, specified as a n-by-2 array of numeric values. For every row in the array, the first element must be less than the second element.

When applying colors to the gauge, MATLAB applies the colors starting with the first color in the `ScaleColors` array. Therefore, if two rows in `ScaleColorLimits` array overlap, then the color applied later takes precedence.

The gauge does not display any portion of the `ScaleColorLimits` that fall outside of the `Limits` property.

If the `ScaleColors` and `ScaleColorLimits` property values are different sizes, then the gauge shows only the colors that have matching limits. For example, if the `ScaleColors` array has three colors, but the `ScaleColorLimits` has only two rows, then the gauge displays the first two color/limit pairs only.

Ticks

`MajorTicks` — Major tick mark locations
`[0 25 50 75 100]` (default) | vector of numeric values | `[]`

Major tick mark locations, specified as a vector of numeric values or an empty vector. If you do not want to show major tick marks, specify this property as an empty vector.

Tick locations that are outside the range of the `Limits` property do not display.

MATLAB removes duplicate tick values. However, if a major tick falls on the same value as a minor tick, only the major tick displays.

Setting the `MajorTicks` property sets the `MajorTicksMode` property to `'manual'`.

`MajorTicksMode` — Major tick creation mode
`'auto'` (default) | `'manual'`

Major tick creation mode, specified as one of the following:

- `'auto'` — MATLAB determines the placement of major ticks.
- `'manual'` — You specify the `MajorTicks` value array.

`MajorTickLabels` — Major tick labels
`{'0','25','50','75','100'}` (default) | cell array of character vectors | string array | 1-D categorical array | `{}`

Major tick labels, specified as a cell array of character vectors, string array, or 1-D categorical array. If you do not want to show tick labels, specify this property as an empty cell array. If you want to remove labels from a specific tick mark, specify an empty

character vector or empty string scalar for the corresponding element in the `MajorTickLabels` array. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

If the length of the `MajorTickLabels` array is different from the length of the `MajorTicks` vector, MATLAB ignores the extra entries of the longer array. If there are extra labels, they are ignored. If there are extra tick marks, they display without labels.

Setting `MajorTickLabels` changes the `MajorTickLabelsMode` value to `'manual'`.

`MajorTickLabelsMode` — Major tick labels mode
`'auto'` (default) | `'manual'`

Major tick labels mode, specified as one of the following:

- `'auto'` — MATLAB specifies the major tick labels.
- `'manual'` — You specify the major tick labels using the `MajorTickLabels` property.

`MinorTicks` — Minor tick locations
`[0 5 10 15 … 100]` (default) | vector of numeric values | `[]`

Minor tick mark locations, specified as a vector of numeric values or an empty vector. If you do not want to show minor tick marks, specify this property as an empty vector.

Tick locations that are outside the range of the `Limits` property do not display.

MATLAB removes duplicate tick values. However, if a minor tick falls on the same value as a major tick, only the major tick displays.

Setting the `MinorTicks` property value sets the `MinorTicksMode` property value to `'manual'`.

`MinorTicksMode` — Minor tick creation mode
`'auto'` (default) | `'manual'`

Minor tick creation mode, specified as one of the following:

- `'auto'` — MATLAB determines the placement of minor ticks. MATLAB does not generate minor ticks for major ticks that are beyond scale limits. If the `Limits` property value changes, then MATLAB updates minor ticks to populate the full scale range (the `MinorTicks` property is updated accordingly).

- `'manual'` — You specify the `MinorTicks` property numeric array. The `MinorTicks` property value does not change size or content on its own.

Font and Color

`FontName` — Font name
system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size
positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

`FontWeight` — Font weight
`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

`FontAngle` — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

`FontColor` — Font color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

BackgroundColor — Background color

[1 1 1] (default) | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Interactivity

`Visible` — Visibility of gauge

`'on'` (default) | `'off'`

Visibility of the gauge, specified as `'on'` or `'off'`. The `Visible` property determines whether the gauge is displayed on the screen. If the `Visible` property is set to `'off'`, then the entire gauge is hidden, but you can still specify and access its properties.

To make your app start faster, set the `Visible` property of all components that do not need to appear at startup to `'off'`.

`Enable` — Operational state of gauge

`'on'` (default) | `'off'`

Operational state of gauge, specified as `'on'` or `'off'`.

- If you set this property to `'on'`, then the appearance of the gauge indicates that the gauge is operational.
- If you set this property to `'off'`, then the appearance of the gauge appears dimmed, indicating that the gauge is not operational.

`Tooltip` — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a

string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size of gauge

`[100 100 90 90]` (default) | `[left bottom width height]`

Location and size of the gauge relative to the parent container, specified as the vector, `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of an imaginary box surrounding the gauge
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of an imaginary box surrounding the gauge
`width`	Distance between the right and left outer edges of the gauge
`height`	Distance between the top and bottom outer edges of the gauge

All measurements are in pixel units.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[100 100 180 180]`

`InnerPosition` — Inner location and size of gauge

`[100 100 90 90]` (default) | `[left bottom width height]`

Inner location and size of the gauge, specified as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`OuterPosition` — Outer location and size of gauge

`[100 100 90 90]` (default) | `[left bottom width height]`

This property is read-only.

Outer location and size of the gauge, returned as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`Layout` — Layout options

empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a ninety-degree gauge in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
gauge = uigauge(g,'ninetydegree');
gauge.Layout.Row = 3;
gauge.Layout.Column = 2;
```

To make the gauge span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this gauge spans columns `2` through `3`:

```
gauge.Layout.Column = [2 3];
```

Callbacks

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

BusyAction — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

BeingDeleted — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

Parent — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

HandleVisibility — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uininetydegreegauge'`

This property is read-only.

Type of graphics object, returned as `'uininetydegreegauge'`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
[ ] (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uifigure` | `uigauge`

Introduced in R2016a

NumericEditField Properties

Control numeric edit field appearance and behavior

Description

Numeric edit fields are UI components that allow users to type numeric values in an app. Properties control the appearance and behavior of a numeric edit field. Use dot notation to refer to a particular object and property:

```
uf = uifigure;
ef = uieditfield(uf,'numeric');
ef.Value = 20;
```

Properties

Value

`Value` — Value in edit field

0 (default) | double-precision number

Value in the edit field, specified as a double-precision number.

When the app user types or changes a value in the edit field, the value is a character vector. When the app user presses the **Enter** key or changes focus, MATLAB converts the app-user-entered value to a double-precision number.

MATLAB rejects the value if:

- It cannot convert the character vector to a scalar number.
- The value is NaN, blank, or a complex number.
- The value is a mathematical expression, such as `1 + 2`.
- The value is less than the `Limits` property lower limit or greater than the upper limit.

When MATLAB rejects the app-user-entered value, a tooltip appears describing the value requirements. The edit field text immediately reverts to the previous value and no `ValueChangedFcn` runs.

Example: 10

`Limits` — Minimum and maximum edit field values
[-Inf Inf] (default) | two-element numeric array

Minimum and maximum edit field values, specified as a two-element numeric array. The first value must be less than the second value. Set array elements to `-Inf` or `Inf` to specify no minimum or no maximum, respectively.

If you change `Limits` such that `Value` is less than the new lower limit, MATLAB sets `Value` to the lowest value within the new range. For example, suppose `Limits` is `[0 100]` and `Value` is 20. If `Limits` changes to `[50 100]`, inclusive, then MATLAB sets `Value` to 50.

Similarly, if you change `Limits` such that the `Value` is greater than the new upper limit, then MATLAB sets `Value` to the new upper limit (assuming the limits are inclusive).

Example: `[-Inf 200]`

Example: `[-100 Inf]`

Example: `[-100 200]`

`RoundFractionalValues` — Rounding of fractional values
`'off'` (default) | `'on'`

Rounding of fractional values entered by app users, specified as one of the following:

- `'on'` — MATLAB rounds the value if it results in a valid value and executes the `ValueChangedFcn` callback. If the resulting value is outside the lower or upper `Limits`, then MATLAB rounds to the nearest value that falls within the `Limits` and then executes the callback.
- `'off'` — MATLAB does not round a fractional value to a whole number.

If the `RoundFractionalValues` property value changes from `'off'` to `'on'` programmatically, then MATLAB applies these rules:

- If rounding the existing value yields an integer that lies inside the limit range specified by the `Limits` property, then MATLAB rounds up the existing value.
- If rounding the existing value yields an integer that is less than the lower limit, then MATLAB rounds up the existing value.
- If rounding the existing value yields an integer that is greater than the upper limit, then MATLAB rounds down the existing value.

- If the limits are configured such that there is no valid integer in the range, then MATLAB sets the `RoundFractionalValues` property value back to `'off'` and displays an error message.

`ValueDisplayFormat` — Value display format

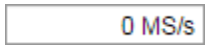
`'%11.4g'` (default) | character vector | string scalar

Value display format, specified as a character vector or string scalar. MATLAB uses `sprintf` to display the value using the specified format.

You can mix text with format operators. For example:

```
numfield = uieditfield('numeric','ValueDisplayFormat','%.0f MS/s');
```

The resulting edit field looks like this:



When the user clicks in the edit field in the running app, the field shows the value without the text.

For a complete list of supported format operators, see `sprintf`.

`LowerLimitInclusive` — Lower limit inclusiveness

`'on'` (default) | `'off'`

Lower limit inclusiveness, specified as one of the following:

- `'on'` — Value must be equal to or greater than the lower limit.
- `'off'` — Value must be greater than the lower limit.

`UpperLimitInclusive` — Upper limit inclusiveness

`'on'` (default) | `'off'`

Upper limit inclusiveness, specified as one of the following:

- `'on'` — Value must be equal to or less than the upper limit.
- `'off'` — Value must be less than the upper limit.

For example, if you want the numeric input to be between 0 and 1, excluding 0 and 1, do all of the following:

- Set the `Limits` property value to `[0 1]`.
- Set the `UpperLimitInclusive` property to `'off'`.
- Set the `LowerLimitInclusive` property to `'off'`.

Font and Color

`FontName` — Font name
system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size
positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

`FontWeight` — Font weight
`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

`FontAngle` — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

`FontColor` — Font color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

BackgroundColor — Background color

[1 1 1] (default) | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Interactivity

`Visible` — Edit field visibility

`'on'` (default) | `'off'`

Edit field visibility, specified as `'on'` or `'off'`. The `Visible` property determines whether the edit field is displayed on the screen. If the `Visible` property is set to `'off'`, the edit field is hidden, but you can still specify and access its properties.

To make your app start faster, set the `Visible` property of all components that do not need to appear at startup to `'off'`.

`Editable` — Editability of edit field

`'on'` (default) | `'off'`

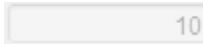
Editability of the edit field, specified as `'on'` or `'off'`.

Use this property in combination with the `Enable` property value to determine if and how the component responds to app user input:

- To make the edit field editable, set both the `Enable` property and the `Editable` property values to `'on'`.

- To make the edit field uneditable, but the text easy to read, set the `Enable` property value to `'on'` and the `Editable` property value to `'off'`.

- To make the edit field uneditable and the text dimmed, set both properties to `'off'`.



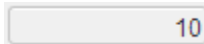
`Enable` — Operational state of edit field
`'on'` (default) | `'off'`

Operational state of the edit field, specified as `'on'` or `'off'`.

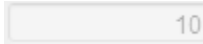
Use this property in combination with the `Editable` property value to determine if and how the component responds to app user input:

- To make the edit field editable and the associated callback triggerable, set both the `Enable` property and the `Editable` property values to `'on'`.

- To make the edit field uneditable, but the text easy to read, set the `Enable` property value to `'on'` and the `Editable` property value to `'off'`.



- To make the edit field uneditable and the text dimmed, set both properties to `'off'`.



`Tooltip` — Tooltip
`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size of edit field

`[100 100 100 22]` (default) | `[left bottom width height]`

Location and size of the edit field relative to the parent, specified as the vector `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the edit field
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the edit field
`width`	Distance between the right and left outer edges of the edit field
`height`	Distance between the top and bottom outer edges of the edit field

All measurements are in pixel units.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[150 130 100 22]`

`InnerPosition` — Inner location and size of edit field

`[100 100 100 22]` (default) | `[left bottom width height]`

Inner location and size of the edit field, specified as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`OuterPosition` — Outer location and size of edit field
`[100 100 100 22]` (default) | `[left bottom width height]`

This property is read-only.

Outer location and size of edit field returned as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`HorizontalAlignment` — Horizontal alignment of numbers
`'right'` (default) | `'left'` | `'center'`

Horizontal alignment of numbers within the edit field, specified as:

- `'right'` — Numbers align on the right side of the edit field.
- `'left'` — Numbers align on the left side of the edit field.
- `'center'`— Numbers align in the center of the edit field.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a numeric edit field in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
ef = uieditfield(g,'numeric');
ef.Layout.Row = 3;
ef.Layout.Column = 2;
```

To make the edit field span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this edit field spans columns `2` through `3`:

```
ef.Layout.Column = [2 3];
```

Callbacks

`ValueChangedFcn` — Value changed callback

`''` (default) | function handle | cell array | character vector

Value changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

The callback executes when the user changes text in the edit field and either presses **Enter** or clicks outside the edit field. It does not execute if the edit field value changes programmatically.

This callback function can access specific information about the user's interaction with the edit field. MATLAB passes this information in a `ValueChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.PreviousValue` returns the previous value of the edit field. The `ValueChangedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangedData` object.

Property	Value
`Value`	Value of edit field after app user's most recent interaction with it
`PreviousValue`	Value of edit field before app user's most recent interaction with it
`Source`	Component that executes the callback
`EventName`	`'ValueChanged'`

For more information about writing callbacks, see "Write Callbacks in App Designer".

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.

- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

Parent — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

HandleVisibility — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uinumericeditfield'`

This property is read-only.

Type of graphics object, returned as `'uinumericeditfield'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uieditfield` | `uifigure`

Introduced in R2016a

Panel Properties

Control panel appearance

Description

Panels are containers for grouping together UI components. Properties control the appearance and behavior of a panel. Use dot notation to refer to a particular object and property.

```
uf = uifigure;
p = uipanel('Parent', uf);
p.Title = 'Display Options';
```

The properties listed here are valid for panels in App Designer, or in apps created with the `uifigure` function. For panels used in GUIDE, or in apps created with the `figure` function, see Uipanel.

Properties

Title

`Title` — Title

character vector | string scalar | categorical array

Title, specified as a character vector, string scalar, or categorical array. If you specify this property as a categorical array, MATLAB displays only the first element in the array.

MATLAB does not interpret a vertical slash (`'|'`) character as a line break, it displays as a vertical slash in the title.

If you want to specify a Unicode character, pass the Unicode decimal code to the `char` function. For example, `['Multiples of ' char(960)]` displays as `Multiples of` π.

`TitlePosition` — Location of title

`'lefttop'` (default) | `'centertop'` | `'righttop'`

Location of the title, specified as `'lefttop'`, `'centertop'`, or `'righttop'`.

Color and Styling

`ForegroundColor` — Title color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Title color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`BackgroundColor` — Background color

`[.94 .94 .94]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

BorderType — Panel border
'line' (default) | 'none'

Panel border, specified as 'line' or 'none'.

Font

FontName — Font name
system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: 'Arial'

FontSize — Font size
positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

FontWeight — Font weight

`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

FontAngle — Font angle

`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

FontUnits — Font unit of measurement

`'pixels'` (default)

Font unit of measurement, specified as `'pixels'`. Distances in pixels are independent of your system resolution on Windows and Macintosh systems:

- On Windows systems, a pixel is 1/96th of an inch.
- On Macintosh systems, a pixel is 1/72nd of an inch.
- On Linux systems, the size of a pixel is determined by your system resolution.

Interactivity

Visible — Panel visibility

`'on'` (default) | `'off'`

Panel visibility, specified as `'on'` or `'off'`. The `Visible` property determines whether the panel is displayed on the screen. If the `Visible` property is set to `'off'`, the panel is hidden. However, you can still specify and access properties of the `Panel` object.

To make your app start faster, set the `Visible` property of all components that are not initially displayed to `'off'`.

Note Changing the `Visible` property does *not* change the values of the `Visible` properties of child components. This is true even though hiding the panel causes the child components to be hidden.

`Scrollable` — Ability to scroll

`'off'` (default) | `'on'`

Ability to scroll, specified as `'off'` or `'on'`. Setting this property to `'on'` enables scrolling within the container. However, there are additional requirements:

- The child components in the container must occupy a larger area than the container can display at one time.
- Components that do not fit in the container must be above or to the right of the container. You cannot scroll to components that are below or to the left of the container.

Certain types of charts and axes do not support scrollable containers. However, you can place the chart or axes in a nonscrollable panel, and then place the panel in the scrollable container. For more information, see “Displaying Graphics in App Designer”.

`Tooltip` — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size of panel, including borders and title

`[left bottom width height]`

Location and size of the panel, including borders and title, specified as a four-element vector of the form `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the panel
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the panel
`width`	Distance between the right and left outer edges of the panel
`height`	Distance between the top and bottom outer edges of the panel

All measurements are in units specified by the `Units` property.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

`InnerPosition` — Location and size of panel, excluding borders and title
`[left bottom width height]`

This property is read-only.

Location and size of the panel, excluding borders and title, returned as a four-element vector of the form `[left bottom width height]`. This table describes each element in the vector.

Value	Description
`left`	Distance from the inner left edge of the parent container to the inner left edge of the panel.
`bottom`	Distance from the inner bottom edge of the parent container to the inner bottom edge of the panel.
`width`	Distance between the inner edges of the panel’s right and left borders.
`height`	Distance between the inner edges of the panel’s top and bottom borders. This distance excludes the title, if it exists.

All measurements are in units specified by the `Units` property.

Note These are some important points to consider when using the `InnerPosition` property:

- `InnerPosition` values are affected by the presence (or absence) of a title, the font characteristics, and the `BorderType`.
- `InnerPosition` values are relative to the parent container's drawable area. The drawable area is the area inside the borders of the container and does not include the area occupied by the title.

`OuterPosition` — Location and size of panel, including borders and title
`[left bottom width height]`

Location and size of the panel, including borders and title, specified as a four-element vector of the form `[left bottom width height]`. All measurements are in units specified by the `Units` property.

This property value is identical to the `Position` property value.

`Units` — Units of measurement
`'pixels'` (default)

Units of measurement, specified as `'pixels'`.

`AutoResizeChildren` — Automatically resize children
`'on'` (default) | `'off'`

Automatically resize children, specified as `'on'` or `'off'`.

- `'on'` — Child components automatically resize when the container resizes.
- `'off'` — Child components do not resize.

The `AutoResizeChildren` property affects direct children of the container, not children inside nested containers.

To customize the resize behavior, set the `AutoResizeChildren` property to `'off'` and create a `SizeChangedFcn` callback for the container. For more information, see "Managing Resizable Apps in App Designer".

To disable resizing of an app, set the `Resize` property of the figure to `'off'`.

Layout — Layout options

empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a panel in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
p = uipanel(g);
p.Layout.Row = 3;
p.Layout.Column = 2;
```

To make the panel span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this panel spans columns `2` through `3`:

```
p.Layout.Column = [2 3];
```

Callbacks

SizeChangedFcn — Size change callback

`''` (default) | function handle | cell array | character vector

Size change callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

Define this callback to customize the app layout when the size of this container changes (e.g., when the user resizes the window).

Note The `SizeChangedFcn` callback will not execute unless the `AutoResizeChildren` property of this container is set to `'off'`. In App Designer, you can make the

`SizeChangedFcn` executable by clearing the **AutoResizeChildren** check box in the **UI Figure Properties** panel.

The `SizeChangedFcn` callback executes when:

- This container becomes visible for the first time.
- This container is visible while its size changes.
- This container becomes visible for the first time after its size changes. This situation occurs when the size changes while the container is invisible, and then it becomes visible later.

Other important points to consider when defining a `SizeChangedFcn` callback:

- Consider delaying the display of this container until after all the variables that the `SizeChangedFcn` uses are defined. This practice can prevent the `SizeChangedFcn` callback from returning an error. To delay the display of the container, set its `Visible` property to `'off'`. Then, set the `Visible` property to `'on'` after you define the variables that your `SizeChangedFcn` callback uses.
- If your app contains nested containers, they resize from the inside out.
- To access the container that is resizing from within the `SizeChangedFcn`, refer to the source object (the first input argument in the callback) or use the `gcbo` function.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`Children` — `Panel` children

empty `GraphicsPlaceholder` array (default) | 1-D array of component objects

`Panel` child objects, returned as an empty `GraphicsPlaceholder` or a 1-D array of component objects. The children of a `Panel` can be any component object, including another `Panel`.

You cannot add or remove child components using the `Children` property. Use this property to view the list of children or to reorder the children. The order of the child objects in this array reflects the front-to-back stacking order of the components on the screen.

To add a child object to this list, set the `Parent` property of the child object to the `Panel` object.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object
`'uipanel'`

This property is read-only.

Type of graphics object, returned as `'uipanel'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uifigure` | `uipanel`

Introduced in R2016a

RadioButton Properties

Control radio button appearance

Description

Radio buttons are typically presented as a set of options inside a button group. The user must choose one from the set. Properties control the appearance and behavior of a radio button. Use dot notation to refer to a specific object and property.

```
uf = uifigure;
bg = uibuttongroup(uf);
rb = uiradiobutton(bg);
rb.Text = 'One';
```

Properties

Button

`Value` — State of radio button

`1 | 0`

State of radio button, specified as `1` (true) or `0` (false). Within a button group, only one button can be selected at a time. The state of the first radio button added to a button group is `1`, by default. Subsequent buttons added to the same button group have a default state of `0`.

When the `Value` property of a `RadioButton` changes to `1`, the `Value` of the property for the previously selected `RadioButton` changes to `0`. In addition, the `SelectedObject` property value of the parent `ButtonGroup` is updated.

If you programmatically change the `Value` property of a `RadioButton` to `0`, MATLAB sets the `Value` property of the first `RadioButton` *added* to the `ButtonGroup` to `1`. If the first `RadioButton` added is the one for which you programmatically set the `Value` property to `0`, then MATLAB sets the `Value` property for the second `RadioButton` added to the `ButtonGroup` to `1`.

Note The first `RadioButton` *added* to a `ButtonGroup` is not necessarily the first `RadioButton` *listed* in the `Children` property of the `ButtonGroup`.

`Text` — Button label

`'Radio Button'` (default) | character vector | cell array of character vectors | string scalar | string array | ...

Button label, specified as a character vector, cell array of character vectors, string scalar, string array, or 1-D categorical array. Specify a character vector or string scalar to label the button with a single line of text. Use a cell array or string array to label the button with multiple lines of text. Each element in the array represents a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Font

`FontName` — Font name

system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size

positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

`FontWeight` — Font weight

`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

`FontAngle` — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

`FontColor` — Font color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Interactivity

`Visible` — Button visibility

`'on'` (default) | `'off'`

Button visibility, specified as `'on'` or `'off'`. The `Visible` property determines whether the button is displayed on the screen. If the `Visible` property of a button is set to `'off'`, the entire button is hidden, but you can still specify and access its properties.

To make your app start faster, set the `Visible` property of all components that do not need to appear at startup to `'off'`.

`Enable` — Operational state of button

`'on'` (default) | `'off'`

Operational state of the button, specified as `'on'` or `'off'`.

- If you set this property to `'on'`, then the appearance of the button indicates that the app user can interact with it.
- If you set this property to `'off'`, then the appearance of the button appears dimmed, indicating that the app user cannot interact with it.

`Tooltip` — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size of radio button

`[10 10 91 22]` (default) | `[left bottom width height]`

Location and size of radio button, specified as a vector of the form `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the button group to the outer left edge of the radio button
`bottom`	Distance from the inner bottom edge of the button group to the outer bottom edge of the radio button
`width`	Distance between the right and left outer edges of the radio button (including text)
`height`	Distance between the top and bottom outer edges of the radio button

The `Position` values are relative to the drawable area of the button group. The drawable area is the area inside the borders of the button group and does not include the area occupied by the title.

All measurements are in pixel units.

`InnerPosition` — Location and size of radio button

`[10 10 91 22]` (default) | `[left bottom width height]`

Location and size of radio button, specified as a four element vector of the form `[left bottom width height]`. All measurements are in pixel units. This property value is identical to the `Position` property.

`OuterPosition` — Location and size of radio button

`[10 10 91 22]` (default) | `[left bottom width height]`

This property is read-only.

Location and size of radio button, returned as a four element vector of the form `[left bottom width height]`. All measurements are in pixel units. This property value is identical to the `Position` property.

Callbacks

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.

  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container
`ButtonGroup` object

Parent container, specified as a `ButtonGroup` object. The `ButtonGroup` must be in a `Figure` created using the `uifigure` function.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.

HandleVisibility Value	Description
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uiradiobutton'`

This property is read-only.

Type of graphics object, returned as `'uiradiobutton'`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uibuttongroup` | `uifigure` | `uiradiobutton`

Introduced in R2016a

RockerSwitch Properties

Control rocker switch appearance and behavior

Description

Rocker switches are UI components that indicate a logical state. Properties control the appearance and behavior of a rocker switch. Use dot notation to refer to a particular object and property.

```
uf = uifigure;
rs = uiswitch(uf,'rocker');
rs.Items = {'Cold','Hot'};
```

Properties

Switch

`Value` — Value
element of `Items` | element of `ItemsData`

Value, specified as an element of the `Items` or `ItemsData` arrays. By default, `Value` is the first element in `Items`.

Specifying `Value` as an element of `Items` moves the switch to the position that matches that element. If `ItemsData` is not empty, then `Value` must be set to an element of `ItemsData`, and the switch will move to the associated position.

`Items` — Switch options
`{'Off','On'}` (default) | cell array of character vectors | string array | 1-by-2 categorical array

Switch options, specified as a cell array of character vectors, string array, or 1-by-2 categorical array. If you specify an array, it must have two elements. Duplicate elements are allowed. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

`ItemsData` — Data associated with each element of `Items`
empty array (`[]`) (default) | 1-by-2 numeric array | 1-by-2 cell array

Data associated with each element of the `Items` property value, specified as a 1-by-2 numeric array or a 1-by-2 cell array. Duplicate elements are allowed.

For example, if you set the `Items` value to `{'Freezing','Boiling'}`, then you might set the `ItemsData` value to corresponding temperatures in degrees Celsius, `[0,100]`. The `ItemsData` value is not visible to the app user.

Example: `{'One' 'Two'}`

Example: `[10 20]`

`Orientation` — Orientation
`'vertical'` (default) | `'horizontal'`

Orientation of the switch, specified as `'vertical'` or `'horizontal'`.

Font

`FontName` — Font name
system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size
positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

`FontWeight` — Font weight
`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

`FontAngle` — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

`FontColor` — Font color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Interactivity

`Visible` — Visibility of switch

`'on'` (default) | `'off'`

Visibility of the switch, specified as `'on'` or `'off'`. The `Visible` property determines whether the switch is displayed on the screen. If the `Visible` property is set to `'off'`, the entire switch is hidden, but you can still specify and access its properties.

To make your app start faster, set the `Visible` property of all components that do not need to appear at startup to `'off'`.

`Enable` — Operational state of switch

`'on'` (default) | `'off'`

Operational state of switch, specified as `'on'` or `'off'`.

If you set this property to `'on'`, the app user can slide the switch.

If you set this property to `'off'`, the switch appears dimmed, indicating that the app user cannot slide the switch and it will not trigger a callback.

`Tooltip` — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size of switch

`[100 100 20 45]` (default) | `[left bottom width height]`

Location and size of the switch, excluding state marks and labels, specified as the vector, `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the switch
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the switch
`width`	Distance between the right and left outer edges of the switch, excluding labels
`height`	Distance between the top and bottom outer edges of the switch, excluding labels

All measurements are in pixel units.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[100 100 40 90]`

InnerPosition — Inner location and size of switch

[100 100 20 45] (default) | [left bottom width height]

Inner location and size of the switch, excluding state marks and state labels, specified as the vector, [left bottom width height]. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the Position property.

OuterPosition — Outer location and size of switch

[100 80 20 85] (default) | [left bottom width height]

This property is read-only.

Outer location and size of the switch, including state marks and state labels, returned as the vector, [left bottom width height]. Position values are relative to the parent container. All measurements are in pixel units.

Layout — Layout options

empty LayoutOptions array (default) | GridLayoutOptions object

Layout options, specified as a GridLayoutOptions object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the Row and Column properties on the GridLayoutOptions object.

For example, this code places a rocker switch in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
rs = uiswitch(g,'rocker');
rs.Layout.Row = 3;
rs.Layout.Column = 2;
```

To make the switch span multiple rows or columns, specify the Row or Column property as a two-element vector. For example, this switch spans columns 2 through 3:

```
rs.Layout.Column = [2 3];
```

Callbacks

ValueChangedFcn — Value changed callback

'' (default) | function handle | cell array | character vector

Value changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when the state of the switch changes in the app. The switch changes state when the user does either of the following:

- Clicking and releasing the mouse button anywhere on the switch (including the state labels)
- Clicking on the switch, dragging, and then releasing the mouse button while still on the switch.

The callback does not execute if the `Value` property changes programmatically.

This callback function can access specific information about the user's interaction with the switch. MATLAB passes this information in a `ValueChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.PreviousValue` returns the previous value of the switch. The `ValueChangedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangedData` object.

Property	Value
`Value`	Value of switch after the app user's most recent interaction with it
`PreviousValue`	Value of switch before the app user's most recent interaction with it
`Source`	Component that executes the callback
`EventName`	`'ValueChanged'`

For more information about writing callbacks, see "Write Callbacks in App Designer".

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.

- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

Parent — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

HandleVisibility — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uirockerswitch'`

This property is read-only.

Type of graphics object, returned as `'uirockerswitch'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uifigure` | `uiswitch`

Introduced in R2016a

scroll

Package: `matlab.ui.container`

Scroll to location within container, list box, or tree

Syntax

```
scroll(component,location)
scroll(component,x,y)
```

Description

`scroll(component,location)` scrolls to the specified location within a component. Use this syntax to scroll within a figure created with the `uifigure` function, or any of these child components: grid layout, panel, button group, tab, tree, or list box.

`scroll(component,x,y)` scrolls to the specified (*x*, *y*) coordinates within a container. The coordinates are measured in pixels from the lower-left corner of the container. This syntax supports figures created with the `uifigure` function and any of these child containers: grid layout, panel, button group, or tab. For more information on when scrolling is possible, see “Conditions that Enable Scrolling” on page 1-17676.

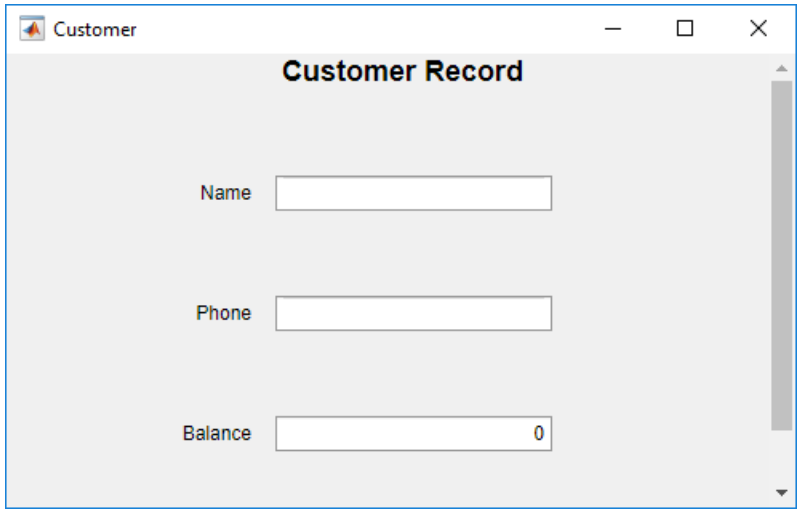
Examples

Scroll Within a Figure

Create a figure, and set the `Scrollable` property to `'on'`. Then add five child components. To ensure that the figure is scrollable, place the first component above the top edge of the figure. Then space the components far enough apart so that they cannot fit within the figure together.

```
% Create figure
fig = uifigure('Name','Customer','Scrollable','on');
fig.Position = [100 100 493 283];
```

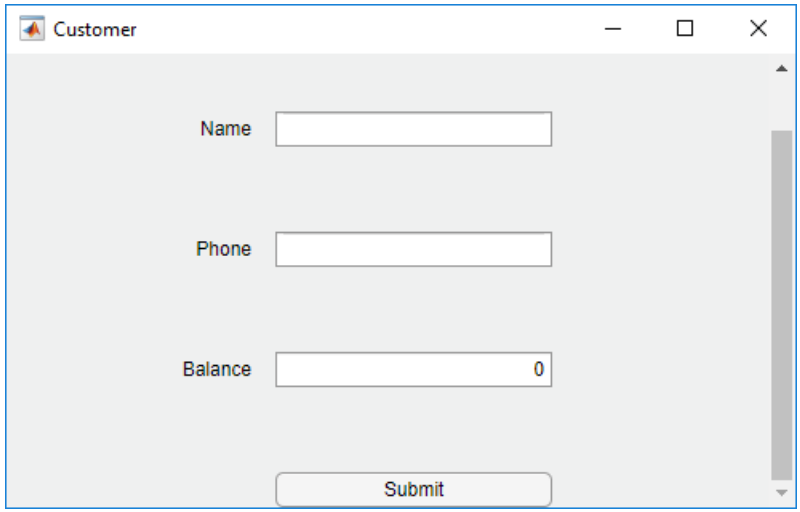
```
% Title label
title = uilabel(fig,'Text','Customer Record');
title.FontSize = 18;
title.FontWeight = 'bold';
title.Position = [173 315 157 22];

% Name
name = uieditfield(fig,'text');
name.Position = [169 239 173 22];
namelabel = uilabel(fig,'Text','Name','HorizontalAlignment','right');
namelabel.Position = [116 239 38 22];

% Phone
phone = uieditfield(fig,'text');
phone.Position = [169 164 173 22];
phonelabel = uilabel(fig,'Text','Phone','HorizontalAlignment','right');
phonelabel.Position = [114 164 40 22];

% Balance
balance = uieditfield(fig,'numeric');
balance.Position = [169 89 173 22];
balancelabel = uilabel(fig,'Text','Balance','HorizontalAlignment','right');
balancelabel.Position = [105 89 49 22];

% Submit button
button = uibutton(fig,'push','Text','Submit');
button.Position = [169 14 173 22];
```



By default, MATLAB scrolls to the upper-left corner of the area that encloses the child components.

Scroll to location `(1,1)`, which is the bottom of the figure.

```
scroll(fig,1,1);
```



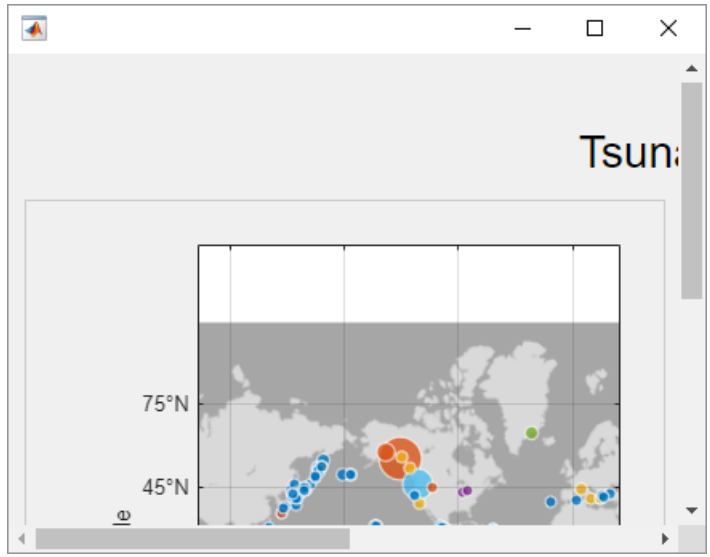
Scroll to Location in Grid

Bring child components of a scrollable grid layout into view by specifying pixel coordinates or a location name.

Create a 5-by-2 grid layout and set the `Scrollable` property of the grid to `'on'`. Then add a label, a table, and a panel to the grid. Set the `Scrollable` property of the panel to `'off'` and then add a chart to the panel.

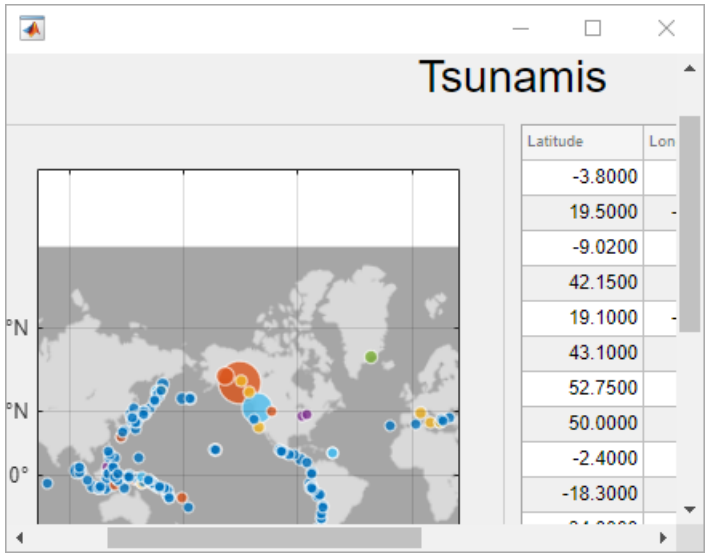
```
fig = uifigure('Position',[782 497 435 311]);
g = uigridlayout(fig,'Scrollable','on');
g.RowHeight = {22,40,22,22,400};
g.ColumnWidth = {400,400};

lbl = uilabel(g,'Text','Tsunamis');
lbl.Layout.Row = 2;
lbl.Layout.Column = [1,2];
lbl.HorizontalAlignment = 'center';
lbl.FontSize = 28;

tsunamis = readtable('tsunamis.xlsx');
tsunamis.Cause = categorical(tsunamis.Cause);
t = uitable(g,'Data',tsunamis);
t.Layout.Row = [3,5];
```

```
t.Layout.Column = 2;

p = uipanel(g);
p.Scrollable = 'off';
p.Layout.Row = [3,5];
p.Layout.Column = 1;
gb = geobubble(p,tsunamis.Latitude,tsunamis.Longitude,...
    tsunamis.MaxHeight,tsunamis.Cause);
```



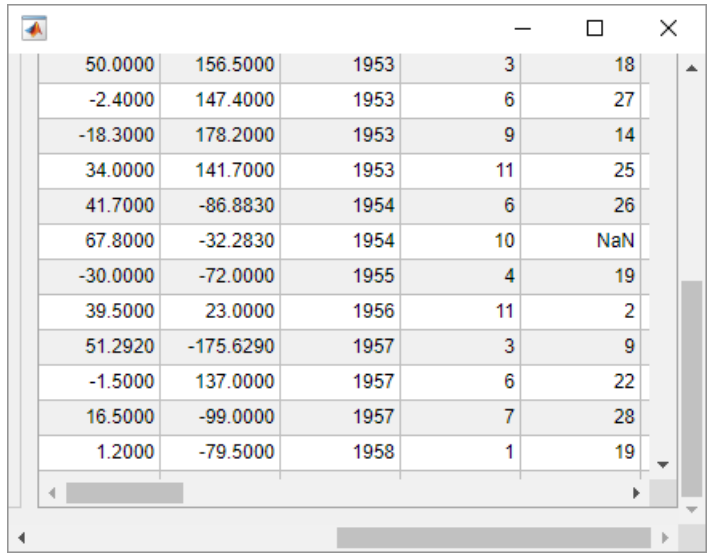
Scroll to a location in the grid.

```
scroll(g,100,-30);
```



Now use location names to scroll to the bottom-right corner of the grid.

```
scroll(g,'bottom','right');
```



50.0000	156.5000	1953	3	18
-2.4000	147.4000	1953	6	27
-18.3000	178.2000	1953	9	14
34.0000	141.7000	1953	11	25
41.7000	-86.8830	1954	6	26
67.8000	-32.2830	1954	10	NaN
-30.0000	-72.0000	1955	4	19
39.5000	23.0000	1956	11	2
51.2920	-175.6290	1957	3	9
-1.5000	137.0000	1957	6	22
16.5000	-99.0000	1957	7	28
1.2000	-79.5000	1958	1	19

Scroll to Component

Bring a component into view by specifying the scroll location as the first two position coordinates of the component you want to view.

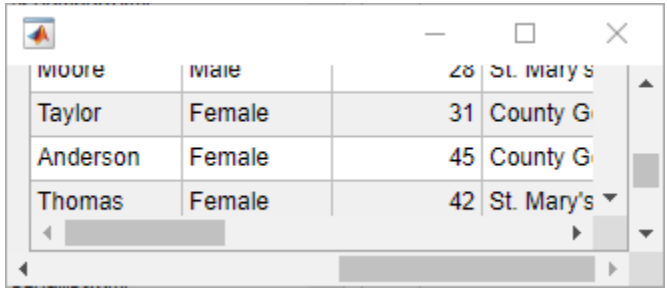
Create a figure with two drop-down components, a list box, and a table. Position the components so that they cannot all be displayed within the figure at one time.

```
fig = uifigure;
fig.Scrollable = 'on';
fig.Position = [100 300 328 110];

dd1 = uidropdown(fig);
dd1.Position = [20 360 120 22];

dd2 = uidropdown(fig);
dd2.Position = [20 200 120 22];

lb = uilistbox(fig);
lb.Position = [230 300 120 80];

t = readtable('patients.xls');
```

```
uit = uitable(fig,'Data',t);
uit.Position = [375 100 300 300];
```

Scroll to the table.

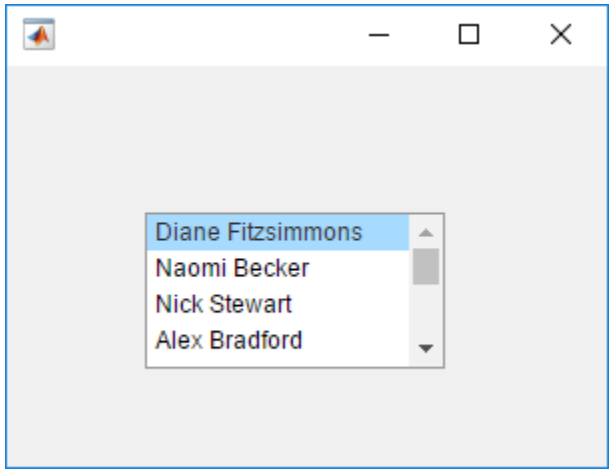
```
scroll(fig,uit.Position(1:2));
```



Scroll to List Box Item

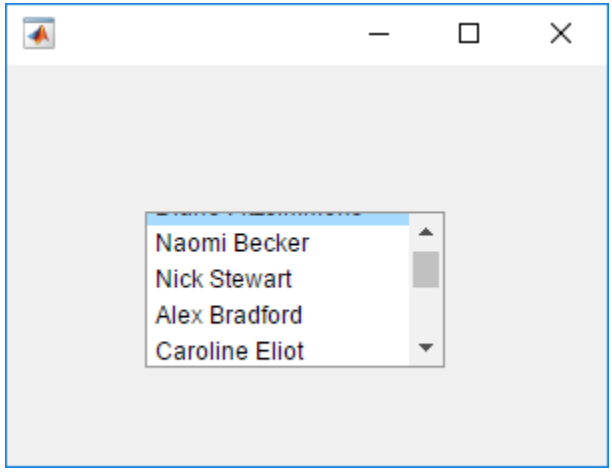
Create a list box containing a list of names with associated `ItemsData`.

```
fig = uifigure('Position',[680 678 300 200]);
list = uilistbox(fig, 'Position',[70 50, 150 78]);
list.Items = {'Diane Fitzsimmons', 'Naomi Becker', 'Nick Stewart',...
            'Alex Bradford', 'Caroline Eliot', 'Leslie Bond', ...
            'Aaron Silberlicht', 'Ramu Sadasiv', 'Joyce Wu',...
            'Ann Shanahan'};
list.ItemsData = [1 2 3 4 5 6 7 8 9 10];
```



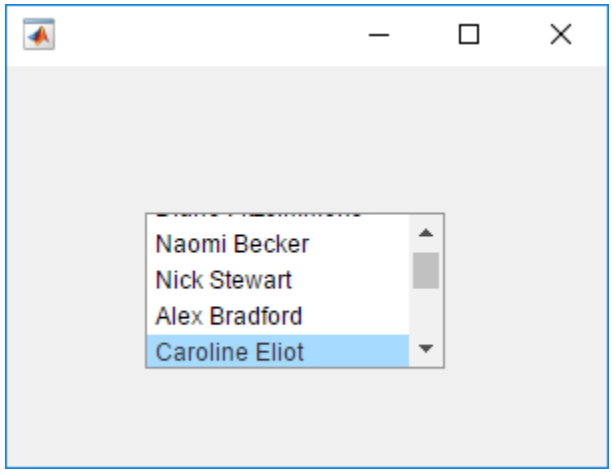
Scroll to `Caroline Eliot`.

```
scroll(list,'Caroline Eliot');
```



Select `Caroline Eliot` by setting the `Value` property to the corresponding element in the `ItemsData` property.

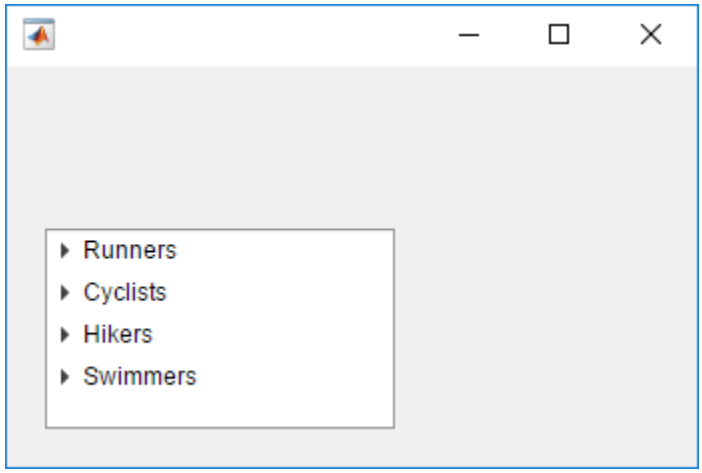
```
list.Value = 5;
```



Scroll to Tree Node

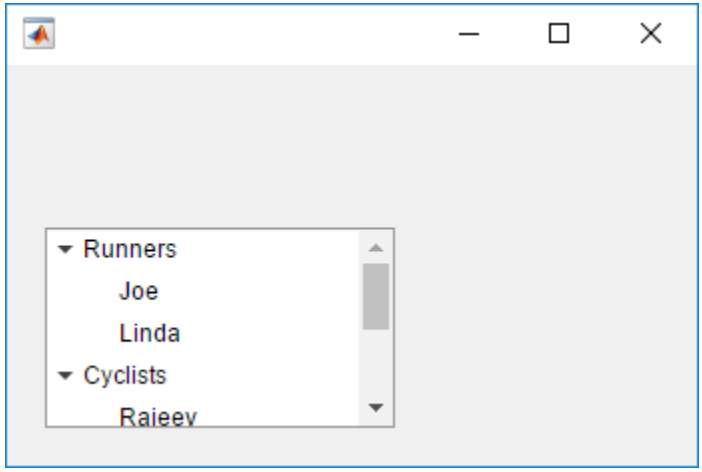
Create a tree containing four top-level nodes that each have child nodes.

```
fig = uifigure;
tree = uitree(fig,'Position',[20 20 175 100]);

% First level nodes
category1 = uitreenode(tree,'Text','Runners');
category2 = uitreenode(tree,'Text','Cyclists');
category3 = uitreenode(tree,'Text','Hikers');
category4 = uitreenode(tree,'Text','Swimmers');

% Second level nodes
r1 = uitreenode(category1,'Text','Joe');
r2 = uitreenode(category1,'Text','Linda');
c1 = uitreenode(category2,'Text','Rajeev');
h1 = uitreenode(category3,'Text','Jack');
s1 = uitreenode(category4,'Text','Logan');
```



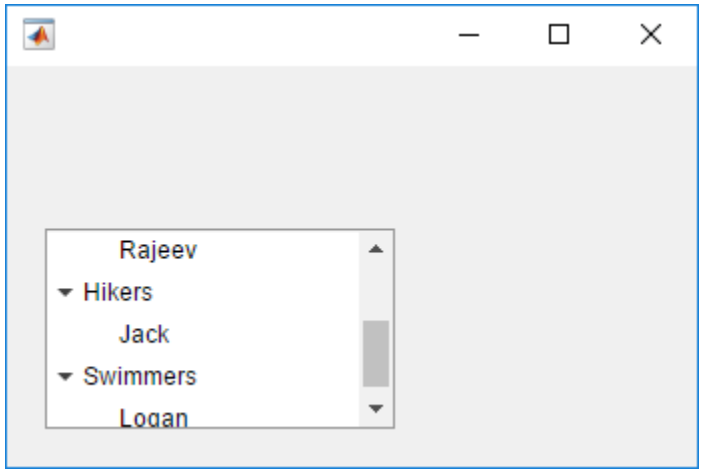
Expand the nodes, so that `Swimmers` scrolls out of view.

```
expand(tree);
```



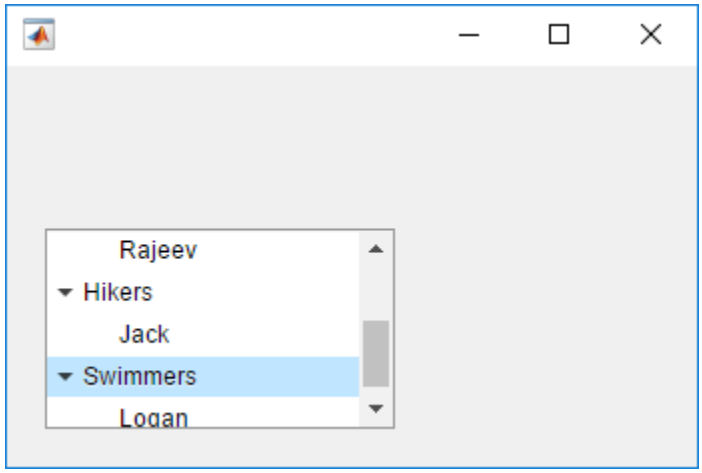
Scroll to the `Swimmers` node.

```
scroll(tree,category4)
```



Select the `Swimmers` node by setting the `SelectedNodes` property of the `Tree` object.

```
tree.SelectedNodes = category4;
```



Input Arguments

`component` — Scrollable component
`Figure` object | `GridLayout` object | `Panel` object | `ButtonGroup` object | `Tab` object | `ListBox` object | ...

Scrollable component, specified as a figure created with the `uifigure` function, or any of the following components within that figure: grid layout, panel, button group, tab, list box, or tree.

`location` — Scroll location
`'top'` | `'bottom'` | ...

Scroll location, specified as `'top'`, `'bottom'`, or a value listed in the table. The values `'top'` and `'bottom'` scroll to the top and bottom of the component. The other values are specific to the type of component. To scroll to the corner of a container, you can combine the `'top'` or `'bottom'` scroll locations with `'left'` or `'right'`. For example, `scroll(uf,'left','top')`.

Component	Values
Figure Panel Button group Tab Grid Layout	• `'left'` — Scrolls to the left edge of the container. • `'right'` — Scrolls to the right edge of the container.
Tree	• `TreeNode` object — Scrolls to the specified tree node.
List box	• Element of `Items` property — Scrolls to the specified item in the list box. • Element of `ItemsData` property — Scrolls to the corresponding item in the list box.

x — *x*-coordinate
number

x-coordinate, specified as a number in pixels from the left edge of the container. If the specified value exceeds the scrollable area of the container, the container scrolls as far as it can in the specified direction.

y — *y*-coordinate
number

y-coordinate, specified as a number in pixels from the bottom edge of the container. If the specified value exceeds the scrollable area of the container, the container scrolls as far as it can in the specified direction.

If a grid layout is taller than its parent container, you use negative *y*-coordinates to scroll to components within the grid that lie below the bottom edge of the parent container.

More About

Conditions that Enable Scrolling

To allow scrolling within a container, the `'Scrollable'` property of the container must be set to `'on'`. In addition, these container-specific conditions must also be true:

Grid layouts

- The sum of the values specified for the `'RowHeight'` property of the grid must be larger than the height of the parent container.
- The sum of the values specified for the `'ColumnWidth'` property of the grid must be larger than the width of the parent container.
- At least one row or column of the grid must be set to a fixed pixel height or width.
- The grid must contain components.

Containers other than grid layout

- The child components in the container must occupy a larger area than the container can display at one time.
- Components that do not fit in the container must be above or to the right of the container.

See Also

Functions
`uifigure` | `uigridlayout` | `uilistbox` | `uitree` | `uitreenode`

Properties
GridLayout Properties | ListBox | Tree | TreeNode | UI Figure

Topics
"Displaying Graphics in App Designer"

Introduced in R2016a

SemicircularGauge Properties

Control semicircular gauge appearance

Description

A semicircular gauge is a UI component that represents a measurement instrument. Properties control the appearance and behavior of a semicircular gauge. Use dot notation to refer to a particular object and property.

```
uf = uifigure;
g = uigauge(uf,'semicircular');
g.Value = 45;
```

Properties

Gauge

`Value` — Location of gauge needle

0 (default) | numeric

Location of the gauge needle, specified as any numeric value.

- If the value is less than the minimum `Limits` property value, then the needle points to a location immediately before the beginning of the scale.
- If the value is more than the maximum `Limits` property value, then the needle points to a location immediately after the end of the scale.

Changing the `Limits` property value has no effect on the `Value` property setting.

Example: `60`

`Limits` — Minimum and maximum gauge scale values

[0 100] (default) | two-element numeric array

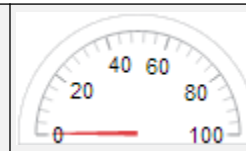
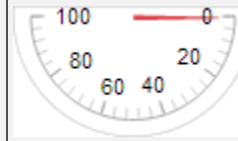
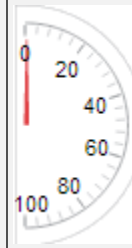
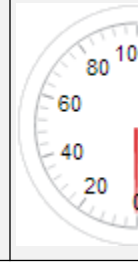
Minimum and maximum gauge scale values, specified as a two-element numeric array. The first value in the array must be less than the second value.

If you change `Limits` such that the `Value` property is less than the new lower limit, or more than the new upper limit, then the gauge needle points to a location off the scale.

For example, suppose `Limits` is `[0 100]` and the `Value` property is 20. If the `Limits` changes to `[50 100]`, then the needle points to a location off the scale, slightly less than 50.

`Orientation` — Orientation
`'north'` (default) | `'south'` | `'east'` | `'west'`

Orientation of the gauge, specified as one of the following:

`'north'`	
`'south'`	
`'east'`	
`'west'`	

`ScaleDirection` — Direction of gauge scale
`'clockwise'` (default) | `'counterclockwise'`

Direction of the gauge scale, specified as one of the following:

- `'clockwise'` — The scale appears such that the scale tick values increase in a clockwise manner.
- `'counterclockwise'` — The scale appears such that the scale tick values increase in a counterclockwise manner.

`ScaleColors` — Scale colors

[ ] (default) | n-by-3 array of RGB triplets | cell array

Scale colors, specified one of the following arrays:

- An n-by-3 array of RGB triplets
- A cell array containing RGB triplets, any of the color options listed in the table below, or a combination of both.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`

Each color of the `ScaleColors` array corresponds to a colored section of the gauge. Set the `ScaleColorLimits` property to map the colors to specific sections of the gauge.

If you do not set the `ScaleColorLimits` property, MATLAB distributes the colors equally over the range of the gauge.

`ScaleColorLimits` — Scale color limits

[ ] (default) | n-by-2 array

Scale color limits, specified as a n-by-2 array of numeric values. For every row in the array, the first element must be less than the second element.

When applying colors to the gauge, MATLAB applies the colors starting with the first color in the `ScaleColors` array. Therefore, if two rows in `ScaleColorLimits` array overlap, then the color applied later takes precedence.

The gauge does not display any portion of the `ScaleColorLimits` that fall outside of the `Limits` property.

If the `ScaleColors` and `ScaleColorLimits` property values are different sizes, then the gauge shows only the colors that have matching limits. For example, if the `ScaleColors` array has three colors, but the `ScaleColorLimits` has only two rows, then the gauge displays the first two color/limit pairs only.

Ticks

`MajorTicks` — Major tick mark locations

`[0 20 40 60 80 100]` (default) | vector of numeric values | `[]`

Major tick mark locations, specified as a vector of numeric values or an empty vector. If you do not want to show major tick marks, specify this property as an empty vector.

Tick locations that are outside the range of the `Limits` property do not display.

MATLAB removes duplicate tick values. However, if a major tick falls on the same value as a minor tick, only the major tick displays.

Setting the `MajorTicks` property sets the `MajorTicksMode` property to `'manual'`.

`MajorTicksMode` — Major tick creation mode

`'auto'` (default) | `'manual'`

Major tick creation mode, specified as one of the following:

- `'auto'` — MATLAB determines the placement of major ticks.
- `'manual'` — You specify the `MajorTicks` value array.

`MajorTickLabels` — Major tick labels
`{'0','20','40','60','80','100'}` (default) | cell array of character vectors | string array | `{}` | ...

Major tick labels, specified as a cell array of character vectors, string array, or 1-D categorical array. If you do not want to show tick labels, specify this property as an empty cell array. If you want to remove a label from a specific tick mark, specify an empty character vector or empty string scalar for the corresponding element in the `MajorTickLabels` array. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

If the length of the `MajorTickLabels` array is different from the length of the `MajorTicks` vector, MATLAB ignores the extra entries of the longer array. If there are extra labels, they are ignored. If there are extra tick marks, they display without labels.

Setting `MajorTickLabels` changes the `MajorTickLabelsMode` value to `'manual'`.

`MajorTickLabelsMode` — Major tick labels mode
`'auto'` (default) | `'manual'`

Major tick labels mode, specified as one of the following:

- `'auto'` — MATLAB specifies the major tick labels.
- `'manual'` — You specify the major tick labels using the `MajorTickLabels` property.

`MinorTicks` — Minor tick mark locations
`[0 4 8 12 ... 100]` (default) | vector of numeric values | `[]`

Minor tick mark locations, specified as a vector of numeric values or an empty vector. If you do not want to show minor tick marks, specify this property as an empty vector.

Tick locations that are outside the range of the `Limits` property do not display.

MATLAB removes duplicate tick values. However, if a minor tick falls on the same value as a major tick, only the major tick displays.

Setting the `MinorTicks` property value sets the `MinorTicksMode` property value to `'manual'`.

`MinorTicksMode` — Minor tick creation mode
`'auto'` (default) | `'manual'`

Minor tick creation mode, specified as one of the following:

- `'auto'` — MATLAB determines the placement of minor ticks. MATLAB does not generate minor ticks for major ticks that are beyond scale limits. If the `Limits` property value changes, then MATLAB updates minor ticks to populate the full scale range (the `MinorTicks` property is updated accordingly).
- `'manual'` — You specify the `MinorTicks` property numeric array. The `MinorTicks` property value does not change size or content on its own.

Font and Color

`FontName` — Font name
system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size
positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

`FontWeight` — Font weight
`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

`FontAngle` — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

`FontColor` — Font color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`BackgroundColor` — Background color

`[1 1 1]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Interactivity

`Visible` — Visibility of gauge

`'on'` (default) | `'off'`

Visibility of the gauge, specified as `'on'` or `'off'`. The `Visible` property determines whether the gauge is displayed on the screen. If the `Visible` property is set to `'off'`, then the entire gauge is hidden, but you can still specify and access its properties.

To make your app start faster, set the `Visible` property of all components that do not need to appear at startup to `'off'`.

`Enable` — Operational state of gauge

`'on'` (default) | `'off'`

Operational state of gauge, specified as `'on'` or `'off'`.

- If you set this property to `'on'`, then the appearance of the gauge indicates that the gauge is operational.

- If you set this property to `'off'`, then the appearance of the gauge appears dimmed, indicating that the gauge is not operational.

`Tooltip` — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size of gauge

`[100 100 120 65]` (default) | `[left bottom width height]`

Location and size of the gauge relative to the parent container, specified as the vector, `[left bottom width height]`.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of an imaginary box surrounding the gauge
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of an imaginary box surrounding the gauge
`width`	Distance between the right and left outer edges of the gauge
`height`	Distance between the top and bottom outer edges of the gauge

All measurements are in pixel units.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[100 100 100 20`

`InnerPosition` — Inner location and size of gauge
`[100 100 120 65]` (default) | `[left bottom width height]`

Inner location and size of the gauge, specified as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`OuterPosition` — Outer location and size of gauge
`[100 100 120 65]]` (default) | `[left bottom width height]`

This property is read-only.

Outer location and size of the gauge, returned as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to `Position` property.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a semicircular gauge in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
gauge = uigauge(g,'semicircular');
gauge.Layout.Row = 3;
gauge.Layout.Column = 2;
```

To make the gauge span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this gauge spans columns `2` through `3`:

```
gauge.Layout.Column = [2 3];
```

Callbacks

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.

HandleVisibility Value	Description
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object
`'uisemicirculargauge'`

This property is read-only.

Type of graphics object, returned as `'uisemicirculargauge'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uifigure` | `uigauge`

Introduced in R2016a

Slider Properties

Control slider appearance and behavior

Description

Sliders are UI components that allow the user to select a value along a continuum. Properties control the appearance and behavior of a slider. Use dot notation to refer to a particular object and property.

```
uf = uifigure;
s = uislider(uf);
s.Value = 20;
```

Properties

Slider

`Value` — Slider value

0 (default) | numeric value

Slider value, specified as a numeric value. The numeric value must be within the range specified by the `Limits` property value.

`Limits` — Minimum and maximum slider values

[0 100] (default) | two-element numeric array

Minimum and maximum slider values, specified as a two-element numeric array. The first value must be less than the second value.

If you change `Limits` such that `Value` property is less than the new lower limit, MATLAB sets the `Value` property to the new lower limit. For example, suppose the `Limits` property is `[0 100]` and `Value` is 20. If the `Limits` changes to `[50 100]`, then MATLAB sets the `Value` property to 50.

Similarly, if you change `Limits` such that the `Value` property is greater than the new upper limit, MATLAB sets the `Value` property to the new upper limit.

`Orientation` — Orientation

`'horizontal'` (default) | `'vertical'`

Orientation of slider, specified as `'horizontal'` or `'vertical'`.

Ticks

`MajorTicks` — Major tick mark locations

`[0 20 40 60 80 100]` (default) | vector of numeric values | `[]`

Major tick mark locations, specified as a vector of numeric values or an empty vector. If you do not want to show major tick marks, specify this property as an empty vector.

Tick locations that are outside the range of the `Limits` property do not display.

MATLAB removes duplicate tick values. However, if a major tick falls on the same value as a minor tick, only the major tick displays.

Setting the `MajorTicks` property sets the `MajorTicksMode` property to `'manual'`.

`MajorTicksMode` — Major tick creation mode

`'auto'` (default) | `'manual'`

Major tick creation mode, specified as one of the following:

- `'auto'` — MATLAB determines the placement of major ticks.
- `'manual'` — You specify the `MajorTicks` value array.

`MajorTickLabels` — Major tick labels

`{'0','20','40','60','80','100'}` (default) | cell array of character vectors | string array | `{}` | ...

Major tick labels, specified as a cell array of character vectors, string array, or 1-D categorical array. If you do not want to show tick labels, specify this property as an empty cell array. If you want to remove a label from a specific tick mark, specify an empty character vector or empty string scalar for the corresponding element in the `MajorTickLabels` array. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

If the length of the `MajorTickLabels` array is different from the length of the `MajorTicks` vector, MATLAB ignores the extra entries of the longer array. If there are extra labels, they are ignored. If there are extra tick marks, they display without labels.

Setting `MajorTickLabels` changes the `MajorTickLabelsMode` value to `'manual'`.

`MajorTickLabelsMode` — Major tick labels mode
`'auto'` (default) | `'manual'`

Major tick labels mode, specified as one of the following:

- `'auto'` — MATLAB specifies the major tick labels.
- `'manual'` — You specify the major tick labels using the `MajorTickLabels` property.

`MinorTicks` — Minor tick mark locations
`[0 4 8 12 ... 100]` (default) | vector of numeric values | `[]`

Minor tick mark locations, specified as a vector of numeric values or an empty vector. If you do not want to show minor tick marks, specify this property as an empty vector.

Tick locations that are outside the range of the `Limits` property do not display.

MATLAB removes duplicate tick values. However, if a minor tick falls on the same value as a major tick, only the major tick displays.

Setting the `MinorTicks` property value sets the `MinorTicksMode` property value to `'manual'`.

`MinorTicksMode` — Minor tick creation mode
`'auto'` (default) | `'manual'`

Minor tick creation mode, specified as `'auto'` or `'manual'`.

When `MinorTicksMode` is set to `'auto'`, MATLAB determines the placement of minor ticks.

Font

`FontName` — Font name
system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size
positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

`FontWeight` — Font weight
`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

`FontAngle` — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

`FontColor` — Font color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Interactivity

`Visible` — Slider visibility

`'on'` (default) | `'off'`

Slider visibility, specified as `'on'` or `'off'`. The `Visible` property determines whether the slider is displayed on the screen. If the `Visible` property is set to `'off'`, the entire slider is hidden, but you can still specify and access its properties.

To make your app start faster, set the `Visible` property of all components that do not need to appear at startup to `'off'`.

`Enable` — Operational state of slider

`'on'` (default) | `'off'`

Operational state of the slider, specified as `'on'` or `'off'`.

- If you set this property to `'on'`, then the app user can change the slider value.
- If you set this property to `'off'`, then the slider appears dimmed, indicating that the app user cannot change the slider value and the slider will not trigger a callback.

`Tooltip` — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size of slider

`[100 100 150 3]` (default) | `[left bottom width height]`

Location and size of the slider excluding tick marks and labels, specified as the vector `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the slider
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the slider
`width`	Distance between the right and left outer edges of the slider

Element	Description
`height`	Distance between the top and bottom outer edges of the slider

All measurements are in pixel units.

You cannot change the height of a slider when the `Orientation` property value is `'horizontal'`. Similarly, you cannot change the width of a slider when the `Orientation` property value is `'vertical'`.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[100 200 60 60]`

`InnerPosition` — Inner location and size of slider
`[100 100 150 3]` (default) | `[left bottom width height]`

Inner location and size of the slider, excluding tick marks and tick labels, specified as the vector, `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`OuterPosition` — Outer location and size of slider
`[94 70 166 39]` (default) | `[left bottom width height]`

This property is read-only.

Outer location and size of the slider, including state marks and state labels, specified as the vector, `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a slider in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
s = uislider(g);
s.Layout.Row = 3;
s.Layout.Column = 2;
```

To make the slider span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this slider spans columns `2` through `3`:

```
s.Layout.Column = [2 3];
```

Callbacks

`ValueChangedFcn` — Value changed callback

`''` (default) | function handle | cell array | character vector

Value changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when the user moves the thumb to a different position on the slider. The callback does not execute if the slider value changes programmatically.

This callback function can access specific information about the user's interaction with the slider. MATLAB passes this information in a `ValueChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.PreviousValue` returns the previous value of the slider. The `ValueChangedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangedData` object.

Property	Value
`Value`	Value of slider after app user's most recent interaction with it

Property	Value
`PreviousValue`	Value of slider before app user's most recent interaction with it
`Source`	Component that executes the callback
`EventName`	`'ValueChanged'`

For more information about writing callbacks, see "Write Callbacks in App Designer".

`ValueChangingFcn` — Value changing callback

`''` (default) | function handle | cell array | character vector

Value changing callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes as the user moves the thumb along the slider in the app. It does not execute if the `Value` property changes programmatically.

This callback can access specific information about the user's interaction with the slider. MATLAB passes this information in a `ValueChangingData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.Value` returns the current value of the slider. The `ValueChangingData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangingData` object.

Property	Value
`Value`	Current value of the slider as the app user is interacting with it
`Source`	Component that executes the callback
`EventName`	`'ValueChanging'`

The `Value` property of the `Slider` object is not updated until the user releases the slider thumb. Therefore, to get the value as the thumb is being moved, your code must get the `Value` property of the `ValueChangingData` object.

The `ValueChangingFcn` callback executes as follows:

- If the app user clicks the slider value once. then the callback executes a single time. For example, if the slider is on 1.0, and the app user single-clicks at 1.1, then the callback executes once.
- If the app user clicks and drags the slider to a new position, the callback executes repeatedly. For example, if the slider value is 1.0, and the app user clicks, holds, and drags the thump to value 10.0, then the callback executes multiple times until the app user releases the thumb.

For more information about writing callbacks, see “Write Callbacks in App Designer”.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If

you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uislider'`

This property is read-only.

Type of graphics object, returned as `'uislider'`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uifigure` | `uislider`

Introduced in R2016a

Spinner Properties

Control spinner appearance and behavior

Description

Spinners are UI components for selecting numeric values from a finite set. Properties control the appearance and behavior of a spinner. Use dot notation to refer to a particular object and property.

```
uf = uifigure;
s = uispinner(uf);
s.Value = 20;
```

Properties

Value

`Value` — Spinner value

0 (default) | double-precision number

Spinner value, specified as a double-precision number.

When the app user types a value in the spinner, the value is a character vector. When the app user presses the **Enter** key or changes focus, MATLAB converts the app-user-entered value to a double-precision number.

MATLAB rejects the value if:

- It cannot convert the character vector to a scalar number.
- The value is NaN, blank, or a complex number.
- The value is a mathematical expression, such as `1+2`.
- The value is less than or greater than the values specified by the `Limits` property.

When MATLAB rejects the app-user-entered value, a tooltip appears describing the value requirements. The spinner immediately reverts to its previous value and no `ValueChangedFcn` runs.

Example: 10

`Limits` — Minimum and maximum spinner values
[-Inf Inf] (default) | two-element numeric array

Minimum and maximum spinner values, specified as a two-element numeric array. The first value must be less than the second value. Set array elements to `-Inf` or `Inf` to specify no minimum or no maximum, respectively.

If you change `Limits` such that the `Value` property is outside the new limits, MATLAB sets the `Value` property to a value within the new limits. For example, suppose the `Limits` property is `[0 100]` and the `Value` property is 20. If the `Limits` property changes to `[50 100]`, then MATLAB sets the `Value` property to 50 (assuming the `LowerLimitInclusive` value is `'on'`.

Example: `[-Inf 200]`

Example: `[-100 Inf]`

Example: `[-100 200]`

`Step` — Quantity by which value is incremented or decremented
1 (default) | numeric scalar

Quantity by which the `Value` property increments or decrements when the app user presses the up and down arrows, respectively.

`RoundFractionalValues` — Rounding of fractional values
`'off'` (default) | `'on'`

Rounding of fractional values entered by app users, specified as one of the following:

- `'on'` — MATLAB rounds the value if it results in a valid value and executes the `ValueChangedFcn` callback. If the resulting value is outside the lower or upper `Limits`, then MATLAB rounds to the nearest value that falls within the `Limits` and then executes the callback.
- `'off'` — MATLAB does not round a fractional value to a whole number.

If the `RoundFractionalValues` property value changes from `'off'` to `'on'` programmatically, then MATLAB applies these rules:

- If rounding the existing value yields an integer that lies inside the limit range specified by the `Limits` property, then MATLAB rounds up the existing value.

- If rounding the existing value yields an integer that is less than the lower limit, then MATLAB rounds up the existing value.
- If rounding the existing value yields an integer that is greater than the upper limit, then MATLAB rounds down the existing value.
- If the limits are configured such that there is no valid integer in the range, then MATLAB sets the `RoundFractionalValues` property value back to `'off'` and displays an error message.

`ValueDisplayFormat` — Value display format
`'%11.4g'` (default) | character vector | string scalar

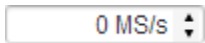
Value display format, specified as a character vector or string scalar.

MATLAB uses `sprintf` to display the value using the specified format.

You can mix text with format operators. For example:

```
spin = uispinner('ValueDisplayFormat','%.0f MS/s');
```

The resulting spinner component looks like this:



When the app user clicks in the spinner field, the field shows the value without the text.

For a complete list of supported format operators, see `sprintf`.

`LowerLimitInclusive` — Lower limit inclusiveness
`'on'` (default) | `'off'`

Lower limit inclusiveness, specified as one of the following:

- `'on'` — Value must be equal to or greater than the lower limit.
- `'off'` — Value must be greater than the lower limit.

`UpperLimitInclusive` — Upper limit inclusiveness
`'on'` (default) | `'off'`

Upper limit inclusiveness, specified as one of the following:

- `'on'` — Value must be equal to or less than the upper limit.
- `'off'` — Value must be less than the upper limit.

For example, if you want the numeric input to be between 0 and 1, excluding 0 and 1, do all of the following:

- Set the `Limits` property value to `[0 1]`.
- Set the `UpperLimitInclusive` property to `'off'`.
- Set the `LowerLimitInclusive` property to `'off'`.

Font and Color

`FontName` — Font name

system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size

positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

`FontWeight` — Font weight

`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

FontAngle — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

FontColor — Font color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`BackgroundColor` — Background color

`[1 1 1]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Interactivity

`Visible` — Spinner visibility

`'on'` (default) | `'off'`

Spinner visibility, specified as `'on'` or `'off'`. The `Visible` property determines whether the spinner is displayed on the screen. If the `Visible` property of a spinner is set to `'off'`, the entire spinner is hidden, but you can still specify and access its properties.

To make your app start faster, set the `Visible` property of all components that do not need to appear at startup to `'off'`.

`Editable` — Editability of spinner

`'on'` (default) | `'off'`

Editability of the spinner, specified as `'on'` or `'off'`.

Use this property in combination with the `Enable` property value to determine if and how the component responds to app user input:

- To make the spinner editable, the arrow buttons operational, and the associated callback triggerable, set both the `Enable` property and the `Editable` property values to `'on'`.
- To make the spinner uneditable, but the arrow buttons operational, set the `Editable` property to `'off'` and the `Enable` property to `'on'`.
- To make the spinner uneditable and the arrow buttons unoperational, set the `Editable` property to `'off'` and the `Enable` property to `'off'`.

`Enable` — Operational state of spinner

`'on'` (default) | `'off'`

Operational state of the spinner, specified as `'on'` or `'off'`. Use this property in combination with the `Editable` property value to determine if and how the component responds to app user input:

- To make the spinner editable, the arrow buttons operational, and the associated callback triggerable, set both the `Enable` property and the `Editable` property values to `'on'`
- To make the spinner uneditable, but the arrow buttons operational, set the `Editable` property to `'off'` and the `Enable` property to `'on'`.
- To make the spinner uneditable and the arrow buttons unoperational, set the `Editable` property to `'off'` and the `Enable` property to `'off'`.

`Tooltip` — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size of spinner

`[100 100 100 22]` (default) | `[left bottom width height]`

Location and size of spinner relative to the parent container, specified as the vector `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the spinner
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the spinner
`width`	Distance between the right and left outer edges of the spinner
`height`	Distance between the top and bottom outer edges of the spinner

All measurements are in pixel units.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[100 100 100 22]`

`InnerPosition` — Inner location and size of spinner
`[100 100 100 22]` (default) | `[left bottom width height]`

Inner location and size of the spinner, specified as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`OuterPosition` — Outer location and size of spinner
`[100 100 100 22]]` (default) | `[left bottom width height]`

This property is read-only.

Outer location and size of spinner, returned as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`HorizontalAlignment` — Horizontal alignment of numbers within spinner
`'right'` (default) | `'left'` | `'center'`

Horizontal alignment of numbers within the spinner, specified as:

- `'right'` — Numbers align on the right side of the spinner.
- `'left'` — Numbers align on the left side of the spinner.
- `'center'` — Numbers align in the center of the spinner.

Layout — Layout options

empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a spinner in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
s = uispinner(g);
s.Layout.Row = 3;
s.Layout.Column = 2;
```

To make the spinner span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this spinner spans columns `2` through `3`:

```
s.Layout.Column = [2 3];
```

Callbacks

ValueChangedFcn — Value changed callback

`''` (default) | function handle | cell array | character vector

Value changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when the user changes focus or presses the **Enter** key after changing the spinner value. It does not matter whether the user changes the spinner

value by typing or by pressing the arrow keys. The callback does not execute if the spinner value changes programmatically.

This callback function can access specific information about the user's interaction with the spinner. MATLAB passes this information in a `ValueChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.PreviousValue` returns the previous value of the spinner. The `ValueChangedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangedData` object.

Property	Value
`Value`	Value of spinner after app user's most recent interaction with it
`PreviousValue`	Value of spinner before app user's most recent interaction with it
`Source`	Component that executes the callback
`EventName`	`'ValueChanged'`

For more information about writing callbacks, see "Write Callbacks in App Designer".

`ValueChangingFcn` — Value changing callback
`''` (default) | function handle | cell array | character vector

Value changing callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback function executes as the user clicks and holds the up or down arrow on the spinner. It does not execute if the `Value` property changes programmatically.

This callback function can access specific information about the user's interaction with the spinner. MATLAB passes this information in a `ValueChangingData` object as the

second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.Value` returns the current value of the spinner. The `ValueChangingData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangingData` object.

Property	Value
`Value`	Current value of the spinner as the app user is interacting with it
`Source`	Component that executes the callback
`EventName`	`'ValueChanging'`

The `Value` property of the `Spinner` is not updated until the app user releases the arrow key. Therefore, to get the values while the arrow key is being pressed, your code must get the `Value` property of the `ValueChangingData` object.

The callback executes as follows:

- If the app user clicks a spinner up or down arrow, the callback executes once. For example, suppose that the spinner value is 2, and the `Step` value is 1. If the app user clicks the up arrow, the callback executes.
- If the app user presses and holds a spinner up or down arrow, the callback executes repeatedly. For example, if the app user clicks and holds the up arrow, the callback executes multiple times until the app user releases the up arrow.

For more information about writing callbacks, see “Write Callbacks in App Designer”.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

BusyAction — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

BeingDeleted — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

Parent — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

HandleVisibility — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uispinner'`

This property is read-only.

Type of graphics object, returned as `'uispinner'`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
[ ] (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uifigure` | `uispinner`

Introduced in R2016a

StateButton Properties

Control state button appearance and behavior

Description

State buttons are components that indicate a logical state. Properties control the appearance and behavior of a state button. Use dot notation to refer to a specific object and property.

```
uf = uifigure;
sb = uibutton(uf,'state');
sb.Value = true;
```

Properties

Button

`Value` — Pressed state of button

`0` (default) | `1`

Pressed state of button, specified as `0` (false) or `1` (true). When the value is 1, the state button appears pressed. When the value is 0, the state button appears unpressed.

`Text` — Button label

`'State Button'` (default) | character vector | cell array of character vectors | string scalar | string array | ...

Button label, specified as a character vector, cell array of character vectors, string scalar, string array, or 1-D categorical array. Specify a character vector or string scalar to label the button with a single line of text. Use a cell array or string array to label the button with multiple lines of text. Each element in the array represents a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

`Icon` — File name of button icon

`''` (default) | character vector | string scalar

File name of the button icon, specified as a character vector or string scalar.

The file name can be an image file name on the MATLAB path or a full path to an image file. If you plan to share an app with others, put the image file on the MATLAB path to facilitate app packaging.

The image file type must be JPEG, GIF, or PNG.

- If the button text takes up all the space specified by the `Position` property value, then MATLAB does not display the icon.
- If some room is available for the icon, then MATLAB scales down the image to fit, if necessary.

Example: `'icon.png'`

Example: `'C:\Documents\icon.png'`

Font and Color

`FontName` — Font name
system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size
positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

`FontWeight` — Font weight
`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

`FontAngle` — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

`FontColor` — Font color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`BackgroundColor` — Background color

`[.96 .96 .96]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Interactivity

`Visible` — Button visibility

`'on'` (default) | `'off'`

Button visibility, specified as `'on'` or `'off'`. The `Visible` property determines whether the button is displayed on the screen. If the `Visible` property is set to `'off'`, the entire button is hidden, but you can still specify and access its properties.

To make your app start faster, set the `Visible` property of all components that do not need to appear at startup to `'off'`.

Enable — Operational state of button

`'on'` (default) | `'off'`

Operational state of the button, specified as `'on'` or `'off'`.

- If you set this property to `'on'`, the app user can press the button.
- If you set this property to `'off'`, the button appears dimmed, indicating that the app user cannot press it, and that the button will not trigger a callback.

Tooltip — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

Position — Location and size of button

`[100 100 100 22]` (default) | `[left bottom width height]`

Location and size of button, specified as the vector of the form `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the button
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the button
`width`	Distance between the right and left outer edges of the button
`height`	Distance between the top and bottom outer edges of the button

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

All measurements are in pixel units.

`InnerPosition` — Location and size of button
`[100 100 100 22]` (default) | `[left bottom width height]`

Location and size of button, specified as a four element vector of the form `[left bottom width height]`. All measurements are in pixel units. This property value is identical to the `Position` property.

`OuterPosition` — Location and size of button
`[100 100 100 22]]` (default) | `[left bottom width height]`

This property is read-only.

Location and size of button, returned as a four element vector of the form `[left bottom width height]`. All measurements are in pixel units. This property value is identical to the `Position` property.

`HorizontalAlignment` — Horizontal alignment of icon and text
`'center'` (default) | `'left'` | `'right'`

Horizontal alignment of the icon and text, specified as `'center'`, `'left'`, or `'right'`. The horizontal alignment is relative to the area inside the borders of the button. Setting this property when the text spans the full width of the button has no apparent effect in the UI.

`VerticalAlignment` — Vertical alignment of icon and text
`'center'` (default) | `'top'` | `'bottom'`

Vertical alignment of the icon and the text, specified as `'center'`, `'top'`, or `'bottom'`. The vertical alignment is relative to the area inside the borders of the button. Setting this property when the text height fills the height of the button has no apparent effect in the UI.

`IconAlignment` — Location of icon relative to button text
`'left'` (default) | `'right'` | `'center'` | `'top'` | `'bottom'`

Location of icon relative to the button text, specified as `'left'`, `'right'`, `'top'`, or `'bottom'`. If the `Text` property is empty, then the icon uses the `HorizontalAlignment` and `VerticalAlignment` properties instead of the `IconAlignment` property.

`Layout` — Layout options

empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a state button in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
sb = uibutton(g,'state');
sb.Layout.Row = 3;
sb.Layout.Column = 2;
```

To make the button span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this button spans columns `2` through `3`:

```
sb.Layout.Column = [2 3];
```

Callbacks

`ValueChangedFcn` — Value changed callback

`''` (default) | function handle | cell array | character vector

Value changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when the user changes the button's state in the app. It does not execute if the state changes programmatically.

This callback function can access specific information about the user's interaction with the button. MATLAB passes this information in a `ValueChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.PreviousValue` returns the previous value of the button. The `ValueChangedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangedData` object.

Property	Value
`Value`	Value of state button after the app user's most recent interaction with it
`PreviousValue`	Value of state button before the app user's most recent interaction with it
`Source`	Component that executes the callback
`EventName`	`'ValueChanged'`

For more information about writing callbacks, see "Write Callbacks in App Designer".

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see "Write Callbacks in App Designer".

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running

callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that

obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uistatebutton'`

This property is read-only.

Type of graphics object, returned as `'uistatebutton'`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uibutton` | `uifigure`

Introduced in R2016a

Switch Properties

Control switch appearance and behavior

Description

Switches are UI components that indicate a logical state. Properties control the appearance and behavior of a switch. Use dot notation to refer to a particular object and property.

```
uf = uifigure;
s = uiswitch(uf);
s.Items = {'Cold','Hot'};
```

Properties

Switch

`Value` — Value
element of `Items` | element of `ItemsData`

Value, specified as an element of the `Items` or `ItemsData` arrays. By default, `Value` is the first element in `Items`.

Specifying `Value` as an element of `Items` moves the switch to the position that matches that element. If `ItemsData` is not empty, then `Value` must be set to an element of `ItemsData`, and the switch will move to the associated position.

`Items` — Switch options
`{'Off','On'}` (default) | cell array of character vectors | string array | 1-by-2 categorical array

Switch options, specified as a cell array of character vectors, string array, or 1-by-2 categorical array. If you specify an array, it must have two elements. Duplicate elements are allowed. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

`ItemsData` — Data associated with each element of `Items`
empty array (`[]`) (default) | 1-by-2 numeric array | 1-by-2 cell array

Data associated with each element of the `Items` property value, specified as a 1-by-2 numeric array or a 1-by-2 cell array. Duplicate elements are allowed.

For example, if you set the `Items` value to `{'Freezing','Boiling'}`, then you might set the `ItemsData` value to corresponding temperatures in degrees Celsius, `[0,100]`. The `ItemsData` value is not visible to the app user.

Example: `{'One' 'Two'}`

Example: `[10 20]`

`Orientation` — Orientation
`'horizontal'` (default) | `'vertical'`

Orientation of the switch, specified as `'horizontal'` or `'vertical'`.

Font

`FontName` — Font name
system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size
positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

`FontWeight` — Font weight
`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

`FontAngle` — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

`FontColor` — Font color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Interactivity

`Visible` — Visibility of switch

`'on'` (default) | `'off'`

Visibility of the switch, specified as `'on'` or `'off'`. The `Visible` property determines whether the switch is displayed on the screen. If the `Visible` property is set to `'off'`, the entire switch is hidden, but you can still specify and access its properties.

To make your app start faster, set the `Visible` property of all components that do not need to appear at startup to `'off'`.

`Enable` — Operational state of switch

`'on'` (default) | `'off'`

Operational state of switch, specified as `'on'` or `'off'`.

If you set this property to `'on'`, the app user can slide the switch.

If you set this property to `'off'`, the switch appears dimmed, indicating that the app user cannot slide the switch and it will not trigger a callback.

`Tooltip` — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size of switch

`[100 100 45 20]` (default) | `[left bottom width height]`

Switch location and size of the switch, excluding state marks and labels, specified as the vector, `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the switch
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the switch
`width`	Distance between the right and left outer edges of the switch, excluding labels
`height`	Distance between the top and bottom outer edges of the switch, excluding labels

All measurements are in pixel units.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[100 200 90 40`

InnerPosition — Inner location and size of switch

[100 100 45 20] (default) | [left bottom width height]

Inner location and size of the switch, excluding state marks and state labels, specified as the vector, [left bottom width height]. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the Position property.

OuterPosition — Outer location and size of switch

[79 100 88 20] (default) | [left bottom width height]

This property is read-only.

Outer location and size of the switch, including state marks and state labels, specified as the vector, [left bottom width height]. Position values are relative to the parent container. All measurements are in pixel units.

Layout — Layout options

empty LayoutOptions array (default) | GridLayoutOptions object

Layout options, specified as a GridLayoutOptions object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the Row and Column properties on the GridLayoutOptions object.

For example, this code places a switch in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
s = uiswitch(g);
s.Layout.Row = 3;
s.Layout.Column = 2;
```

To make the switch span multiple rows or columns, specify the Row or Column property as a two-element vector. For example, this switch spans columns 2 through 3:

```
s.Layout.Column = [2 3];
```

Callbacks

ValueChangedFcn — Value changed callback

'' (default) | function handle | cell array | character vector

Value changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when the state of the switch changes in the app. The switch changes state when the user does either of the following:

- Clicking and releasing the mouse button anywhere on the switch (including the state labels)
- Clicking on the switch, dragging, and then releasing the mouse button while still on the switch.

The callback does not execute if the `Value` property changes programmatically.

This callback function can access specific information about the user's interaction with the switch. MATLAB passes this information in a `ValueChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.PreviousValue` returns the previous value of the switch. The `ValueChangedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangedData` object.

Property	Value
`Value`	Value of switch after the app user's most recent interaction with it
`PreviousValue`	Value of switch before the app user's most recent interaction with it
`Source`	Component that executes the callback
`EventName`	`'ValueChanged'`

For more information about writing callbacks, see "Write Callbacks in App Designer".

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.

- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

Parent — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

HandleVisibility — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uiswitch'`

This property is read-only.

Type of graphics object, returned as `'uiswitch'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uifigure` | `uiswitch`

Introduced in R2016a

Tab Properties

Control tab appearance

Description

Tabs are containers for grouping for grouping UI components together with a tabbed label. Use the `uitab` function to create a tab inside a tab group. Properties control the appearance and behavior of a tab. Use dot notation to refer to a particular object and property:

```
uf = uifigure;
tg = uitabgroup(uf);
t = uitab(tg);
t.Title = 'Data';
```

The properties listed here are valid for tabs in App Designer, or in apps created with the `uifigure` function. For tabs used in GUIDE, or in apps created with the `figure` function, see Uitab.

Properties

Title and Color

`Title` — Title

character vector | string scalar | categorical array

Title, specified as a character vector, string scalar, or categorical array. If you specify this property as a categorical array, MATLAB displays only the first element in the array.

MATLAB does not interpret a vertical slash (`'|'`) character as a line break, it displays as a vertical slash in the title.

If you want to specify a Unicode character, pass the Unicode decimal code to the `char` function. For example, `['Multiples of ' char(960)]` displays as `Multiples of` π.

`ForegroundColor` — Title color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Title color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

BackgroundColor — Background color

[.94 .94 .94] (default) | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Interactivity

`Scrollable` — Ability to scroll

`'off'` (default) | `'on'`

Ability to scroll, specified as `'off'` or `'on'`. Setting this property to `'on'` enables scrolling within the container. However, there are additional requirements:

- The child components in the container must occupy a larger area than the container can display at one time.
- Components that do not fit in the container must be above or to the right of the container. You cannot scroll to components that are below or to the left of the container.

Certain types of charts and axes do not support scrollable containers. However, you can place the chart or axes in a nonscrollable panel, and then place the panel in the scrollable container. For more information, see “Displaying Graphics in App Designer”.

`Tooltip` — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size of tab

`[left bottom width height]`

This property is read-only.

Location and size of the tab, returned as a four-element vector of the form `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the tab group to the inner left edge of the tab
`bottom`	Distance from the inner bottom edge of the tab group to the inner bottom edge of the tab
`width`	Distance between the right and left inner edges of the tab
`height`	Distance between the top and bottom inner edges of the tab

All measurements are in units specified by the `Units` property.

Note These are some important points to consider when using the `Position` property:

- `Position` values are affected by the length of the tab title and the `TabLocation` property of the parent `TabGroup` object.
- `Position` values are relative to the drawable area of the parent `TabGroup`, which is the area inside its borders.

`InnerPosition` — Location and size of tab

`[left bottom width height]`

This property is read-only.

Location and size of the tab, returned as a four-element vector of the form `[left bottom width height]`. All measurements are in units specified by the `Units` property.

This property value is identical to the `Position` and `OuterPosition` property values.

`OuterPosition` — Location and size of tab
`[left bottom width height]`

This property is read-only.

Location and size of the tab, returned as a four-element vector of the form `[left bottom width height]`. All measurements are in units specified by the `Units` property.

This property value is identical to the `Position` and `InnerPosition` property values.

`Units` — Units of measurement
`'pixels'` (default)

Units of measurement, specified as `'pixels'`.

`AutoResizeChildren` — Automatically resize children
`'on'` (default) | `'off'`

Automatically resize children, specified as `'on'` or `'off'`.

- `'on'` — Child components automatically resize when the container resizes.
- `'off'` — Child components do not resize.

The `AutoResizeChildren` property affects direct children of the container, not children inside nested containers.

To customize the resize behavior, set the `AutoResizeChildren` property to `'off'` and create a `SizeChangedFcn` callback for the container. For more information, see "Managing Resizable Apps in App Designer".

To disable resizing of an app, set the `Resize` property of the figure to `'off'`.

Callbacks

`SizeChangedFcn` — Size change callback
`''` (default) | function handle | cell array | character vector

Size change callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

Define this callback to customize the app layout when the size of this container changes (e.g., when the user resizes the window).

Note The `SizeChangedFcn` callback will not execute unless the `AutoResizeChildren` property of this container is set to `'off'`. In App Designer, you can make the `SizeChangedFcn` executable by clearing the **AutoResizeChildren** check box in the **UI Figure Properties** panel.

The `SizeChangedFcn` callback executes when the parent tab group container:

- Becomes visible for the first time.
- Is visible while its size changes.
- Becomes visible for the first time after its size changes. This situation occurs when the size changes while the container is invisible, and then it becomes visible later.

Other important points to consider when defining a `SizeChangedFcn` callback:

- Consider delaying the display of this container until after all the variables that the `SizeChangedFcn` uses are defined. This practice can prevent the `SizeChangedFcn` callback from returning an error. To achieve this delay, you delay the display of the parent tab group container, by setting its `Visible` property to `'off'`. Then, set the `Visible` property to `'on'` after you define the variables that your `SizeChangedFcn` callback uses.
- If your app contains nested containers, they resize from the inside out.
- To access the container that is resizing from within the `SizeChangedFcn`, refer to the source object (the first input argument in the callback).

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.

- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent object

`TabGroup` object

Parent object, specified as a `TabGroup` object. The `TabGroup` must be the child of a `Figure` created using the `uifigure` function.

`Children` — Children of tab

empty `GraphicsPlaceholder` array (default) | 1-D array of component objects

Children of tab, returned as an empty `GraphicsPlaceholder` or a 1-D array of app component objects.

You cannot add or remove children using the `Children` property of the tab. Use this property to view the list of children or to reorder the children. The order of the children in this array reflects the order of the tabs displayed on the screen.

To add a child to this list, set the `Parent` property of the child component to be the `Tab` object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.

HandleVisibility Value	Description
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object
`'uitab'`

This property is read-only.

Type of graphics object, returned as `'uitab'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uifigure` | `uitab` | `uitabgroup`

Properties
TabGroup

Introduced in R2016a

TabGroup Properties

Control tab group appearance and behavior

Description

Tab groups are containers for grouping and managing tabs. Properties control the appearance and behavior of a tab group. Use dot notation to refer to a particular object and property.

```
uf = uifigure;
tg = uitabgroup(uf);
tg.Position = [20 20 200 200];
```

The properties listed here are valid for tab groups in App Designer, or in apps created with the `uifigure` function. For tab groups used in GUIDE, or in apps created with the `figure` function, see Uitabgroup.

Properties

Tabs

`TabLocation` — Tab label location

`'top'` (default) | `'bottom'` | `'left'` | `'right'`

Tab label location, specified as `'top'`, `'bottom'`, `'left'`, or `'right'`. This property specifies the location of the tab labels with respect to the tab group.

`SelectedTab` — Currently selected tab

`Tab` object

Currently selected tab, specified as a `Tab` object. Use this property to determine the currently selected tab within a tab group. You can also use this property to set the default tab selection. The default value of the `SelectedTab` property is the first `Tab` that you add to the `TabGroup`.

Interactivity

`Visible` — Tab group visibility

`'on'` (default) | `'off'`

Tab group visibility, specified as `'on'` or `'off'`. The `Visible` property determines whether the tab group is displayed on the screen. If the `Visible` property is set to `'off'`, the entire tab group is hidden. However, you can still specify and access properties of the `TabGroup` object.

To make your app start faster, set the `Visible` property of all components that are not initially displayed to `'off'`.

Note Changing the `Visible` property of a `TabGroup` object does *not* change the values of the `Visible` properties of its child components. This is true even though hiding the tab group causes the children to be hidden.

`Tooltip` — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or categorical array. Use this property to display a message when the user hovers the pointer over the tab group at run time. Tooltips for tab groups display only when there are no tabs in the tab group. To display multiple lines of text, specify a cell array of character vectors, string array, or categorical array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size of tab group

`[left bottom width height]`

Location and size of the tab group, specified as a four-element vector of the form `[left bottom width height]`.

This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the tab group
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the tab group
`width`	Distance between the right and left outer edges of the tab group
`height`	Distance between the top and bottom outer edges of the tab group

All measurements are in units specified by the `Units` property.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

`InnerPosition` — Location and size of tab group
`[left bottom width height]`

Location and size of the tab group, specified as a four-element vector of the form `[left bottom width height]`. All measurements are in units specified by the `Units` property.

This property value is identical to the `Position` and `OuterPosition` property values.

`OuterPosition` — Location and size of tab group
`[left bottom width height]`

Location and size of the tab group, specified as a four-element vector of the form `[left bottom width height]`. All measurements are in units specified by the `Units` property.

This property value is identical to the `Position` and `InnerPosition` property values.

`Units` — Units of measurement
`'pixels'` (default)

Units of measurement, specified as `'pixels'`.

`AutoResizeChildren` — Automatically resize children
`'on'` (default) | `'off'`

Automatically resize children, specified as `'on'` or `'off'`.

- `'on'` — Child components automatically resize when the container resizes.
- `'off'` — Child components do not resize.

The `AutoResizeChildren` property affects direct children of the container, not children inside nested containers.

To customize the resize behavior, set the `AutoResizeChildren` property to `'off'` and create a `SizeChangedFcn` callback for the container. For more information, see "Managing Resizable Apps in App Designer".

To disable resizing of an app, set the `Resize` property of the figure to `'off'`.

`Layout` — Layout options

empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a tab group in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
tg = uitabgroup(g);
tg.Layout.Row = 3;
tg.Layout.Column = 2;
```

To make the tab group span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this tab group spans columns `2` through `3`:

```
tg.Layout.Column = [2 3];
```

Callbacks

`SelectionChangedFcn` — Selection changed callback

`''` (default) | function handle | cell array | character vector

Selection changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback function executes when the user selects a different tab within the tab group.

This callback function can access specific information about the user's interaction with the tabs. MATLAB passes this information in a `SelectionChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.NewValue` returns the currently selected tab. The `SelectionChangedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `SelectionChangedData` object.

Property	Description
`OldValue`	Previously selected `Tab`, or `[]` if none was selected
`NewValue`	Currently selected `Tab`
`Source`	Component that executes the callback
`EventName`	`'SelectionChanged'`

For more information about writing callbacks, see "Write Callbacks in App Designer".

`SizeChangedFcn` — Size change callback
`''` (default) | function handle | cell array | character vector

Size change callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

Define this callback to customize the app layout when the size of this container changes (e.g., when the user resizes the window).

Note The `SizeChangedFcn` callback will not execute unless the `AutoResizeChildren` property of this container is set to `'off'`. In App Designer, you can make the `SizeChangedFcn` executable by clearing the **AutoResizeChildren** check box in the **UI Figure Properties** panel.

The `SizeChangedFcn` callback executes when:

- This container becomes visible for the first time.
- This container is visible while its size changes.
- This container becomes visible for the first time after its size changes. This situation occurs when the size changes while the container is invisible, and then it becomes visible later.

Other important points to consider when defining a `SizeChangedFcn` callback:

- Consider delaying the display of this container until after all the variables that the `SizeChangedFcn` uses are defined. This practice can prevent the `SizeChangedFcn` callback from returning an error. To delay the display of the container, set its `Visible` property to `'off'`. Then, set the `Visible` property to `'on'` after you define the variables that your `SizeChangedFcn` callback uses.
- If your app contains nested containers, they resize from the inside out.
- To access the container that is resizing from within the `SizeChangedFcn`, refer to the source object (the first input argument in the callback) or use the `gcbo` function.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you

do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`Children` — `TabGroup` children
empty `GraphicsPlaceholder` array (default) | 1-D array of `Tab` objects

`TabGroup` children, returned as an empty `GraphicsPlaceholder` or a one-dimensional array of `Tab` objects.

You cannot add or remove tabs using the `Children` property of the `TabGroup`. Use this property to view the list of tabs or to reorder the tabs. The order of the `Tab` objects in this array reflects the order of the tabs displayed on the screen.

To add a child to this list, set the `Parent` property of a `Tab` object to be the `TabGroup` object.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object
`'uitabgroup'`

This property is read-only.

Type of graphics object, returned as `'uitabgroup'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uifigure` | `uitab` | `uitabgroup`

Properties
Tab

Introduced in R2016a

TextArea Properties

Control text area appearance and behavior

Description

Text areas are UI components for entering multiple lines of text. Properties control the appearance and behavior of an text area. Use dot notation to refer to a specific object and property.

```
uf = uifigure;
tarea = uitextarea(uf);
tarea.Value = 'This sample is an outlier';
```

Properties

Text

`Value` — Value
`{ ' ' }` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Value, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. MATLAB can properly render formatted text, such as this:

```
cellArrayText{1} = sprintf('%s\n%s', 'Line 1', 'Line 2')
cellArrayText{2} = sprintf('%s\n%s', 'Line 3', 'Line 4')
textarea = uitextarea('Value',cellArrayText);
```

If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

If the text does not fit into the width of the text area, MATLAB wraps the text.

If there are too many rows to display in the text area, MATLAB adds a scroll bar.

Example: {'Joseph Welford'; 'Mary Reilly'; 'Roberta Silberlicht'}

`HorizontalAlignment` — Horizontal alignment of text within text area
`'left'` (default) | `'right'` | `'center'`

Alignment of text within the text area, specified as `'left'`, `'right'`, or `'center'`. The alignment affects the display as the app user edits the text area and how MATLAB displays the text in the app.

Font and Color

`FontName` — Font name
system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size
positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

`FontWeight` — Font weight
`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

`FontAngle` — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

`FontColor` — Font color

`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

BackgroundColor — Background color

[1 1 1] (default) | RGB triplet | hexadecimal color code | 'r' | 'g' | 'b' | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Interactivity

`Visible` — Text area visibility

`'on'` (default) | `'off'`

Text area visibility, specified as `'on'` or `'off'`. The `Visible` property determines whether the text area is displayed on the screen. If the `Visible` property is set to `'off'`, the entire text area is hidden, but you can still specify and access its properties.

To make your app start faster, set the `Visible` property of all components that do not need to appear at startup to `'off'`.

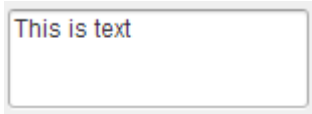
`Editable` — Editability of text area

`'on'` (default) | `'off'`

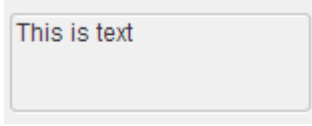
Editability of the text area, specified as `'on'` or `'off'`.

Use this property in combination with the `Enable` property value to determine if and how the component responds to app user input:

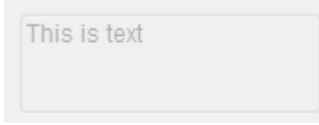
- To make the text area editable and the associated callback triggerable, set both the `Enable` property and the `Editable` property values to `'on'`.



- To make the text area uneditable, but the text easy to read, set the `Enable` property value to `'on'` and the `Editable` property value to `'off'`.



- To make the text area uneditable and the text dimmed, set both properties to `'off'`.



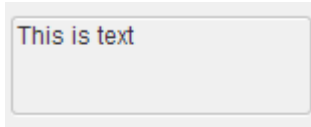
`Enable` — Operational state of text area
`'on'` (default) | `'off'`

Operational state of the text area, specified as `'on'` or `'off'`.

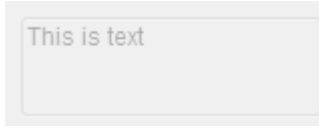
Use this property in combination with the `Enable` property value to determine if and how the component responds to app user input:

- To make the text area editable and the associated callback triggerable, set both the `Enable` property and the `Editable` property values to `'on'`.

- To make the text area uneditable, but the text easy to read, set the `Enable` property value to `'on'` and the `Editable` property value to `'off'`.



- To make the text area uneditable and the text dimmed, set both properties to `'off'`.



`Tooltip` — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size of text area

`[100 100 150 60]` (default) | `[left bottom width height]`

Location and size of the text area relative to the parent, specified as the vector `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the text area
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the text area
`width`	Distance between the right and left outer edges of the text area

Element	Description
`height`	Distance between the top and bottom outer edges of the text area

All measurements are in pixel units.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[100 100 100 90]`

`InnerPosition` — Inner location and size of text area
`[100 100 150 60]` (default) | `[left bottom width height]`

Inner location and size of the text area, specified as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`OuterPosition` — Outer location and size of text area
`[100 100 150 60]` (default) | `[left bottom width height]`

This property is read-only.

Outer location and size of the text area returned as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a text area in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
tarea = uitextarea(g);
```

```
tarea.Layout.Row = 3;
tarea.Layout.Column = 2;
```

To make the text area span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this text area spans columns `2` through `3`:

```
tarea.Layout.Column = [2 3];
```

Callbacks

`ValueChangedFcn` — Value changed callback

`''` (default) | function handle | cell array | character vector

Value changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

The callback executes when the user changes the text and either presses **Tab** or clicks outside the text area. It does not execute if the `Value` property changes programmatically.

This callback function can access specific information about the user's interaction with the text area. MATLAB passes this information in a `ValueChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.PreviousValue` returns the previous value of the text area. The `ValueChangedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangedData` object.

Property	Value
`Value`	Value of text area after app user's most recent interaction with it
`PreviousValue`	Value of text area before app user's most recent interaction with it

Property	Value
`Source`	Component that executes the callback
`EventName`	`'ValueChanged'`

For more information about writing callbacks, see “Write Callbacks in App Designer”.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.

  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.

- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.

HandleVisibility Value	Description
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object
`'uitextarea'`

This property is read-only.

Type of graphics object, returned as `'uitextarea'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uifigure` | `uitextarea`

Introduced in R2016a

ToggleButton Properties

Control toggle button appearance

Description

Toggle buttons are typically presented as a set of options inside a button group. The user must choose one from the set. Properties control the appearance and behavior of a toggle button. Use dot notation to refer to a specific object and property.

```
uf = uifigure;
bg = uibuttongroup(uf)
tbutton = uitogglebutton(bg);
tbutton.Text = 'One';
```

Properties

Button

`Value` — State of toggle button

1 | 0

State of the toggle button specified as `0` (unpressed) or `1` (depressed). Within a given button group, only one toggle button can be selected (depressed) at a time. When the `Value` property is set to `1`, the toggle button appears depressed. The state of the first button added to a button group is `1`, by default. Subsequent buttons added to the same button group have a default state of `0`.

When the `Value` property of a `ToggleButton` changes to `1`, the `Value` property of the previously selected `ToggleButton` changes to `0`. In addition, the `SelectedObject` property value of the `ButtonGroup` is updated.

If you programmatically change the `Value` property of a `ToggleButton` to `0`, MATLAB sets the `Value` property of the first `ToggleButton` *added* to the `ButtonGroup` to `1`. If the first `ToggleButton` added is the one for which you programmatically set the `Value` property to `0`, then MATLAB sets the `Value` property for the `ToggleButton` added to the `ButtonGroup` to `1`.

Note The first `ToggleButton` *added* to a `ButtonGroup` is not necessarily the first `ToggleButton` *listed* in the `Children` property of the `ButtonGroup`.

`Text` — Button label

`'Toggle Button'` (default) | character vector | cell array of character vectors | string scalar | string array | ...

Button label, specified as a character vector, cell array of character vectors, string scalar, string array, or 1-D categorical array. Specify a character vector or string scalar to label the button with a single line of text. Use a cell array or string array to label the button with multiple lines of text. Each element in the array represents a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

`Icon` — File name of button icon

`''` (default) | character vector | string scalar

File name of the button icon, specified as a character vector or string scalar.

The file name can be an image file name on the MATLAB path or a full path to an image file. If you plan to share an app with others, put the image file on the MATLAB path to facilitate app packaging.

The image file type must be JPEG, GIF, or PNG.

- If the button text takes up all the space specified by the `Position` property value, then MATLAB does not display the icon.
- If some room is available for the icon, then MATLAB scales down the image to fit, if necessary.

Example: `'icon.png'`

Example: `'C:\Documents\icon.png'`

Font and Color

`FontName` — Font name

system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size
positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

`FontWeight` — Font weight
`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

`FontAngle` — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

`FontColor` — Font color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to

F. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

`BackgroundColor` — Background color

`[.96 .96 .96]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Interactivity

`Visible` — Visibility of button

`'on'` (default) | `'off'`

Visibility of the button, specified as `'on'` or `'off'`. The `Visible` property determines whether the button is displayed on the screen. If the `Visible` property is set to `'off'`, the entire button is hidden, but you can still specify and access its properties.

To make your app start faster, set the `Visible` property of all components that do not need to appear at startup to `'off'`.

`Enable` — Operational state of button

`'on'` (default) | `'off'`

Operational state of the button, specified as `'on'` or `'off'`.

- If you set this property to `'on'`, then the appearance of the button indicates that the app user can interact with it.
- If you set this property to `'off'`, then the appearance of the button appears dimmed, indicating that the app user cannot interact with it.

`Tooltip` — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size of button

`[10 40 100 22]` (default) | `[left bottom width height]`

Location and size button, specified as a vector of the form `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the button group to the outer left edge of the button
`bottom`	Distance from the inner bottom edge of the button group to the outer bottom edge of the button
`width`	Distance between the right and left outer edges of the button
`height`	Distance between the top and bottom outer edges of the button

The `Position` values are relative to the drawable area of the button group. The drawable area is the area inside the borders of the button group and does not include the area occupied by the title.

All measurements are in pixel units.

`InnerPosition` — Location and size of button
`[10 10 100 22]` (default) | `[left bottom width height]`

Location and size of button, specified as a four element vector of the form `[left bottom width height]`. All measurements are in pixel units. This property value is identical to the `Position` property.

`OuterPosition` — Outer location and size of button
`[10 10 100 22]` (default) | `[left bottom width height]`

This property is read-only.

Location and size of button, returned as a four element vector of the form `[left bottom width height]`. All measurements are in pixel units. This property value is identical to the `Position` property.

`HorizontalAlignment` — Horizontal alignment of icon and text
`'center'` (default) | `'left'` | `'right'`

Horizontal alignment of the icon and text, specified as `'center'`, `'left'`, or `'right'`. The horizontal alignment is relative to the area inside the borders of the button. Setting this property when the text spans the full width of the button has no apparent effect in the UI.

`VerticalAlignment` — Vertical alignment of icon and text

`'center'` (default) | `'top'` | `'bottom'`

Vertical alignment of the icon and the text, specified as `'center'`, `'top'`, or `'bottom'`. The vertical alignment is relative to the area inside the borders of the button. Setting this property when the text height fills the height of the button has no apparent effect in the UI.

`IconAlignment` — Location of icon relative to button text

`'left'` (default) | `'right'` | `'center'` | `'top'` | `'bottom'`

Location of icon relative to the button text, specified as `'left'`, `'right'`, `'top'`, or `'bottom'`. If the `Text` property is empty, then the icon uses the `HorizontalAlignment` and `VerticalAlignment` properties instead of the `IconAlignment` property.

Callbacks

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container
`ButtonGroup` object

Parent container, specified as a `ButtonGroup` object. The `ButtonGroup` must be in a `Figure` created using the `uifigure` function.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uitogglebutton'`

This property is read-only.

Type of graphics object, returned as `'uitogglebutton'`.

Tag — Object identifier

`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data

`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uibuttongroup` | `uifigure` | `uitogglebutton`

Introduced in R2016a

ToggleSwitch Properties

Control toggle switch appearance and behavior

Description

Toggle switches are UI components that indicate a logical state. Properties control the appearance and behavior of a toggle switch. Use dot notation to refer to a particular object and property.

```
uf = uifigure;
s = uiswitch(uf,'toggle');
s.Items = {'Cold','Hot'};
```

Properties

Switch

`Value` — Value
element of `Items` | element of `ItemsData`

Value, specified as an element of the `Items` or `ItemsData` arrays. By default, `Value` is the first element in `Items`.

Specifying `Value` as an element of `Items` moves the switch to the position that matches that element. If `ItemsData` is not empty, then `Value` must be set to an element of `ItemsData`, and the switch will move to the associated position.

`Items` — Switch options
`{'Off','On'}` (default) | cell array of character vectors | string array | 1-by-2 categorical array

Switch options, specified as a cell array of character vectors, string array, or 1-by-2 categorical array. If you specify an array, it must have two elements. Duplicate elements are allowed. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

`ItemsData` — Data associated with each element of `Items`
empty array (`[]`) (default) | 1-by-2 numeric array | 1-by-2 cell array

Data associated with each element of the `Items` property value, specified as a 1-by-2 numeric array or a 1-by-2 cell array. Duplicate elements are allowed.

For example, if you set the `Items` value to `{'Freezing','Boiling'}`, then you might set the `ItemsData` value to corresponding temperatures in degrees Celsius, `[0,100]`. The `ItemsData` value is not visible to the app user.

Example: `{'One' 'Two'}`

Example: `[10 20]`

`Orientation` — Orientation
`'vertical'` (default) | `'horizontal'`

Orientation of the switch, specified as `'vertical'` or `'horizontal'`.

Font

`FontName` — Font name
system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size
positive number

Font size, specified as a positive number. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

Example: 14

`FontWeight` — Font weight
`'normal'` (default) | `'bold'`

Font weight, specified as one of these values:

- `'normal'` — Default weight as defined by the particular font
- `'bold'` — Thicker character outlines than `'normal'`

Not all fonts have a bold font weight. Therefore, specifying a bold font weight can result in the normal font weight.

`FontAngle` — Font angle
`'normal'` (default) | `'italic'`

Font angle, specified as `'normal'` or `'italic'`. Setting this property to `italic` selects a slanted version of the font, if it is available on the app user's system.

`FontColor` — Font color
`[0 0 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Font color, specified as an RGB triplet, a hexadecimal color code, or one of the options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Interactivity

`Visible` — Visibility of switch

`'on'` (default) | `'off'`

Visibility of the switch, specified as `'on'` or `'off'`. The `Visible` property determines whether the switch is displayed on the screen. If the `Visible` property is set to `'off'`, the entire switch is hidden, but you can still specify and access its properties.

To make your app start faster, set the `Visible` property of all components that do not need to appear at startup to `'off'`.

`Enable` — Operational state of switch

`'on'` (default) | `'off'`

Operational state of switch, specified as `'on'` or `'off'`.

If you set this property to `'on'`, the app user can slide the switch.

If you set this property to `'off'`, the switch appears dimmed, indicating that the app user cannot slide the switch and it will not trigger a callback.

`Tooltip` — Tooltip

`''` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Tooltip, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. Use this property to display a message when the user hovers the pointer over the component at run time. The tooltip displays even when the component is disabled. To display multiple lines of text, specify a cell array of character vectors or a string array. Each element in the array becomes a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Position

`Position` — Location and size of switch

`[100 100 20 45]` (default) | `[left bottom width height]`

Location and size of the switch, excluding state marks and labels, specified as the vector, `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the switch
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the switch
`width`	Distance between the right and left outer edges of the switch, excluding labels
`height`	Distance between the top and bottom outer edges of the switch, excluding labels

All measurements are in pixel units.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[100 200 40 90`

InnerPosition — Inner location and size of switch

`[100 100 20 45]` (default) | `[left bottom width height]`

Inner location and size of the switch, excluding state marks and labels, specified as the vector, `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

OuterPosition — Outer location and size of switch

`[100 80 20 85]` (default) | `[left bottom width height]`

This property is read-only.

Outer location and size of the switch, including state marks and labels, specified as the vector, `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units.

Layout — Layout options

empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a toggle switch in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
s = uiswitch(g,'toggle');
s.Layout.Row = 3;
s.Layout.Column = 2;
```

To make the switch span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this switch spans columns `2` through `3`:

```
s.Layout.Column = [2 3];
```

Callbacks

ValueChangedFcn — Value changed callback

`''` (default) | function handle | cell array | character vector

Value changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when the state of the switch changes in the app. The switch changes state when the user does either of the following:

- Clicking and releasing the mouse button anywhere on the switch (including the state labels)
- Clicking on the switch, dragging, and then releasing the mouse button while still on the switch.

The callback does not execute if the `Value` property changes programmatically.

This callback function can access specific information about the user's interaction with the switch. MATLAB passes this information in a `ValueChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.PreviousValue` returns the previous value of the switch. The `ValueChangedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangedData` object.

Property	Value
`Value`	Value of switch after the app user's most recent interaction with it
`PreviousValue`	Value of switch before the app user's most recent interaction with it
`Source`	Component that executes the callback
`EventName`	`'ValueChanged'`

For more information about writing callbacks, see "Write Callbacks in App Designer".

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.

- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

Parent — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

HandleVisibility — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object

`'uitoggleswitch'`

This property is read-only.

Type of graphics object, returned as `'uitoggleswitch'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

Functions
`appdesigner` | `uifigure` | `uiswitch`

Introduced in R2016a

uialert

Display alert dialog box

Syntax

```
uialert(f,message,title)
uialert( ___ ,Name,Value)
```

Description

`uialert(f,message,title)` displays a modal dialog box in front of the specified figure window. The figure must be created with the `uifigure` function. The dialog box displays the specified message and has a title bar with the specified title. By default, the dialog box also contains an error icon and an **OK** button.

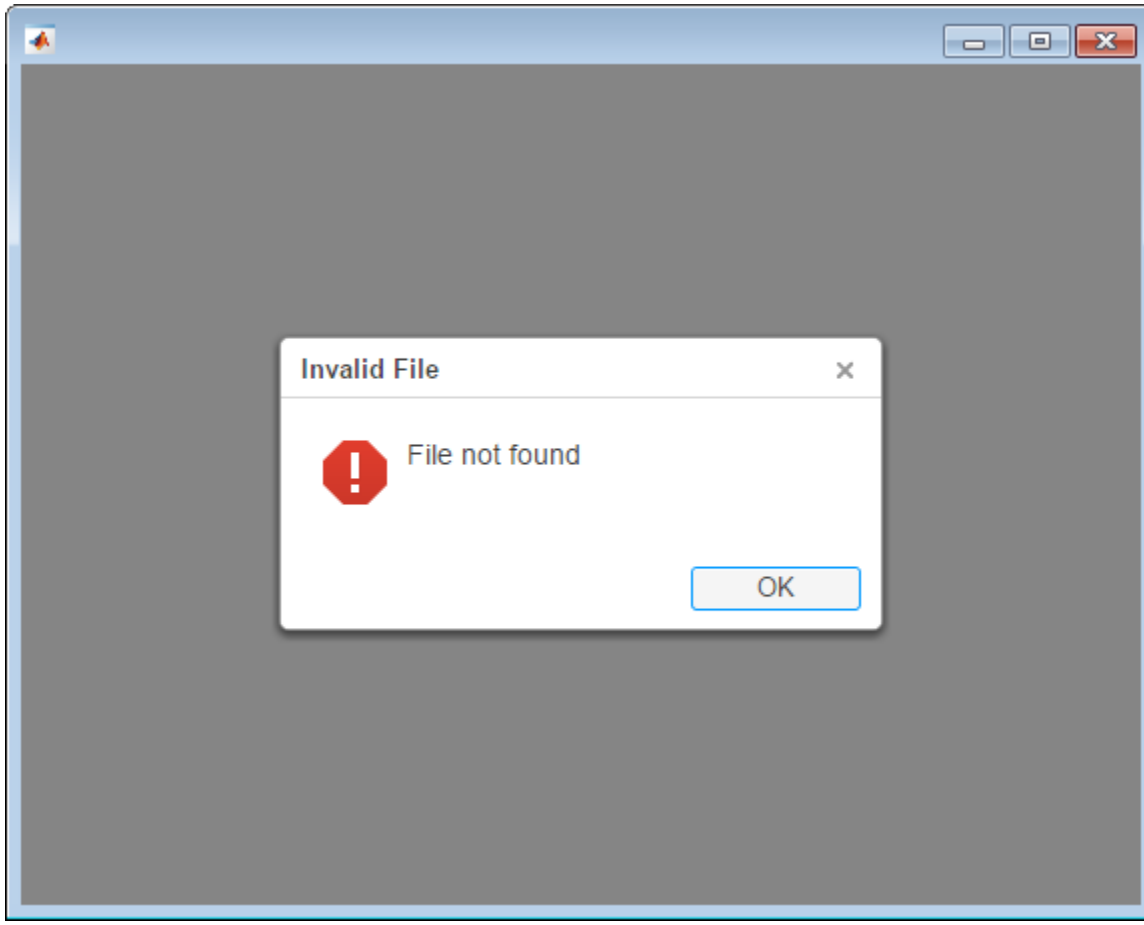
`uialert( ___ ,Name,Value)` displays a dialog box with properties specified by one or more `Name,Value` pair arguments.

Examples

Alert App User to Error

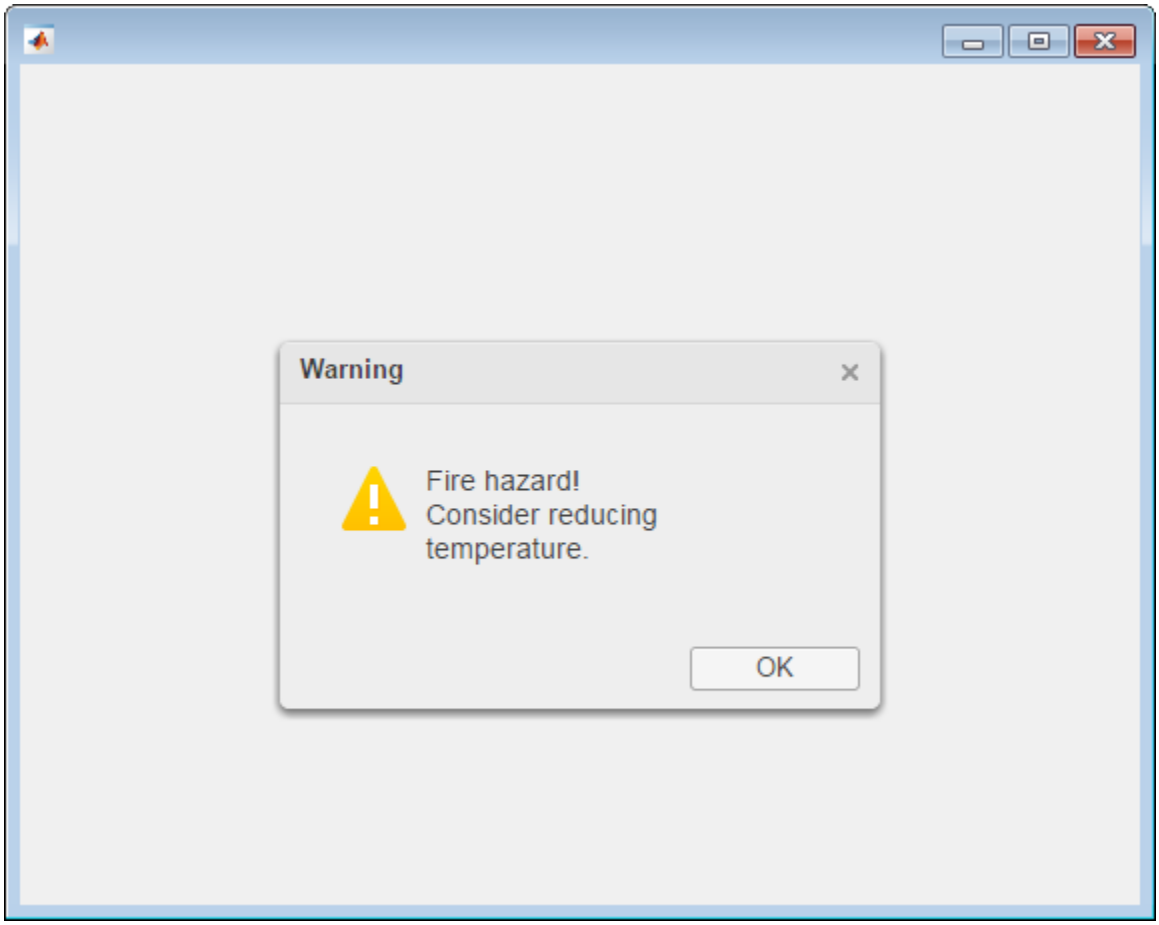
Create a modal alert dialog box. The figure behind it is inaccessible until you close the dialog box.

```
f = uifigure;
uialert(f,'File not found','Invalid File');
```



Create Multiline Modal Alert Dialog Box with Warning Icon

```
f = uifigure;
message = sprintf('Fire hazard! \n Consider reducing temperature.');
uialert(f,message,'Warning',...
'Icon','warning');
```



Input Arguments

f — Target figure
`Figure` object

Target figure, specified as a `Figure` object. The figure must be created with the `uifigure` function.

message — Message
character vector | cell array of character vectors | string array

Message, specified as a character vector, cell array of character vectors, or string array. The message can be any length.

Format characters such as the newline character (`\n`) are honored. To display multiple lines of text, use a cell array of character vectors or a string array.

Example: `'Invalid value'`

`title` — Title
`'Error'` (default) | character vector | string scalar

Title, specified as a character vector or string scalar. Long titles truncate and are followed by ellipses. The exact length at which a title is truncated depends on your system setup.

Spaces replace format characters, such as the newline (`\n`) character.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Icon','warning'`

`CloseFcn` — Alert dialog box close request function callback
`''` (default) | function handle | cell array | character vector

The alert dialog box close request callback function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

The `CloseFcn` callback executes when the app user dismisses the alert dialog box by clicking the close button in the title bar or clicking the **OK** button. The `CloseFcn` function also executes when the app user closes the figure window for which the alert dialog box displays from the MATLAB command line. You can close a figure window from the command line using the `close` function.

`Icon` — Icon

`'error'` (default) | predefined icon | custom icon

Icon, specified as a predefined icon or a custom icon.

Predefined Icon

This table lists the values for the predefined icons. For example, to show the check mark icon, specify the name-value pair `'Icon','success'`.

Value	Icon
`'error'` (default)	
`'warning'`	
`'info'`	
`'success'`	
`''`	No icon displays.

Custom Icon

Specify a custom icon as one of these values:

- A character vector that specifies the file name of an SVG, JPEG, GIF, or PNG image that is on the MATLAB path. Alternatively, you can specify a full path to the image file.
- A truecolor image array. See “Image Types” for more information.

`Modal` — Modal

`true` (default) | `false`

Modal, specified as `true` or `false`.

A modal dialog box is one that makes the figure inaccessible until the dialog box is closed. However, it does not prevent additional dialog boxes from appearing.

See Also

Functions
`errordlg` | `msgbox` | `uiconfirm` | `uifigure` | `uiprogressdlg` | `warndlg`

Introduced in R2016a

uiaxes

Create UI axes for plots in App Designer

Syntax

```
ax = uiaxes
ax = uiaxes(Name,Value)
ax = uiaxes(parent)
ax = uiaxes(parent,Name,Value)
```

Description

`ax = uiaxes` creates UI axes in a new figure window and returns the `UIAxes` object. MATLAB calls the `uifigure` function to create the figure.

`ax = uiaxes(Name,Value)` specifies `UIAxes` property values using one or more `Name,Value` pair arguments.

`ax = uiaxes(parent)` creates the UI axes in the specified parent container. The parent can be a `Figure` created using the `uifigure` function, or one of its child containers.

`ax = uiaxes(parent,Name,Value)` specifies `UIAxes` property values using one or more `Name,Value` arguments.

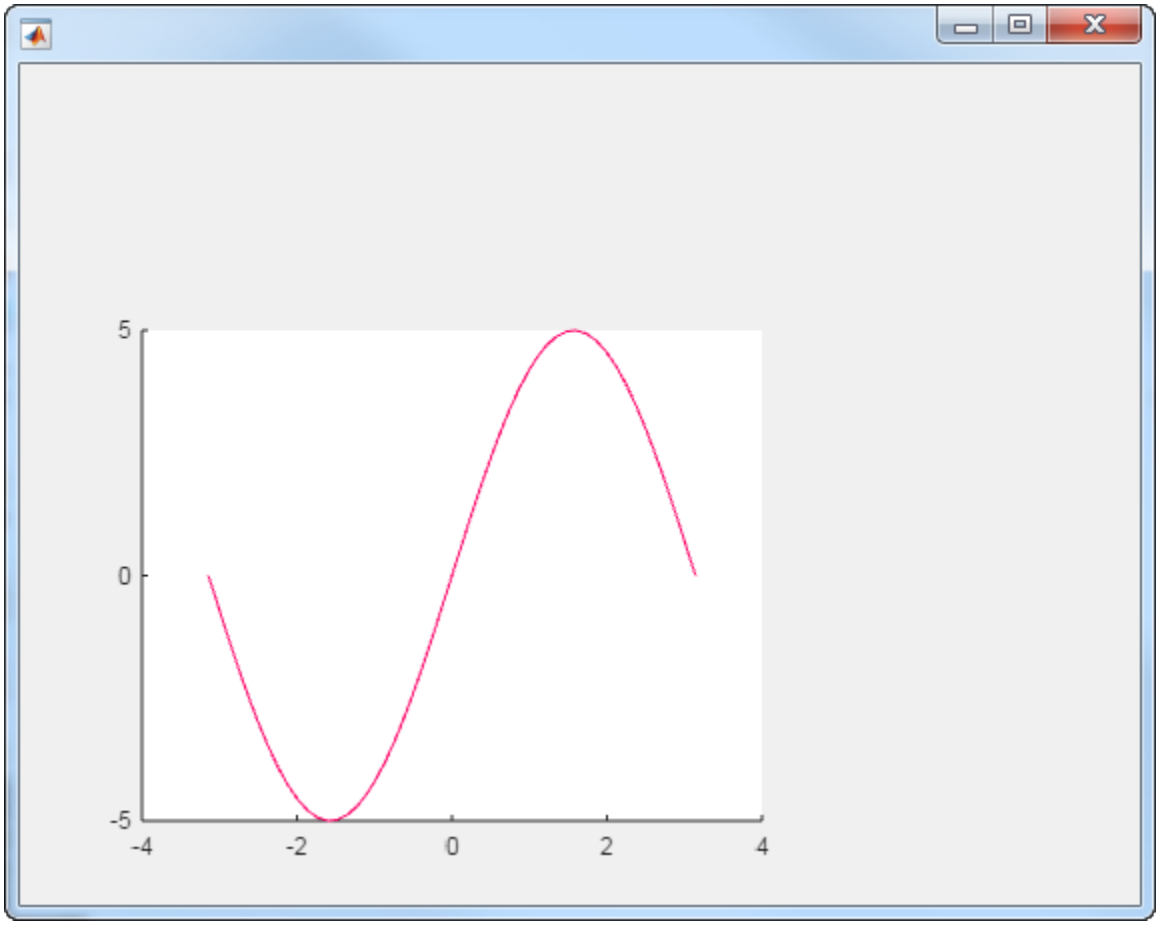
Examples

Create Plots in UI Axes

Create a line plot and a scatter plot in UI axes.

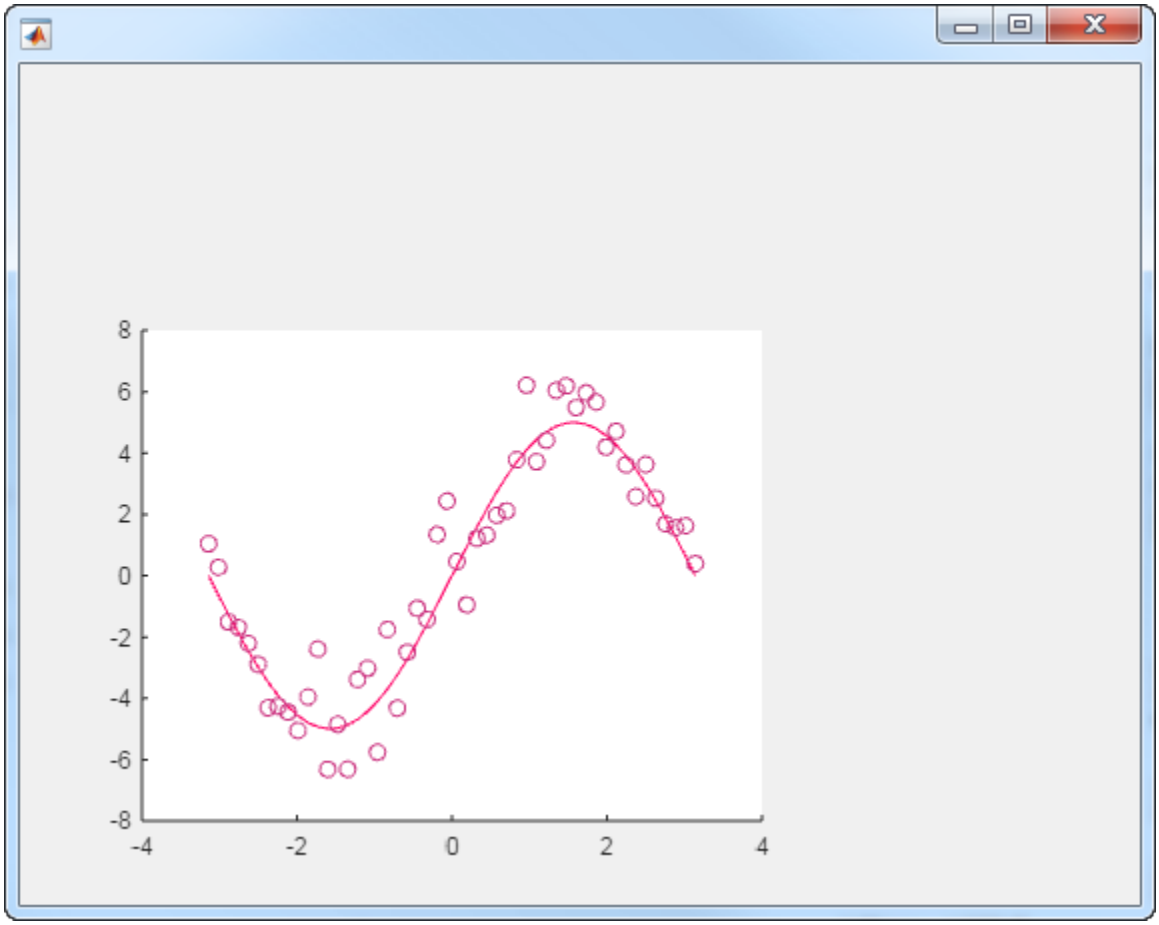
Create a figure window with UI axes and assign the `UIAxes` object to the variable `ax`. Add a line plot to the axes by specifying the `UIAxes` object as the first input argument for the `plot` function.

```
fig = uifigure;
ax = uiaxes(fig);
x = linspace(-pi,pi,50);
y = 5*sin(x);
plot(ax,x,y)
```



Set the hold state on and add a scatter plot. Specify the `UIAxes` object as the first input argument for the `hold` and `scatter` functions.

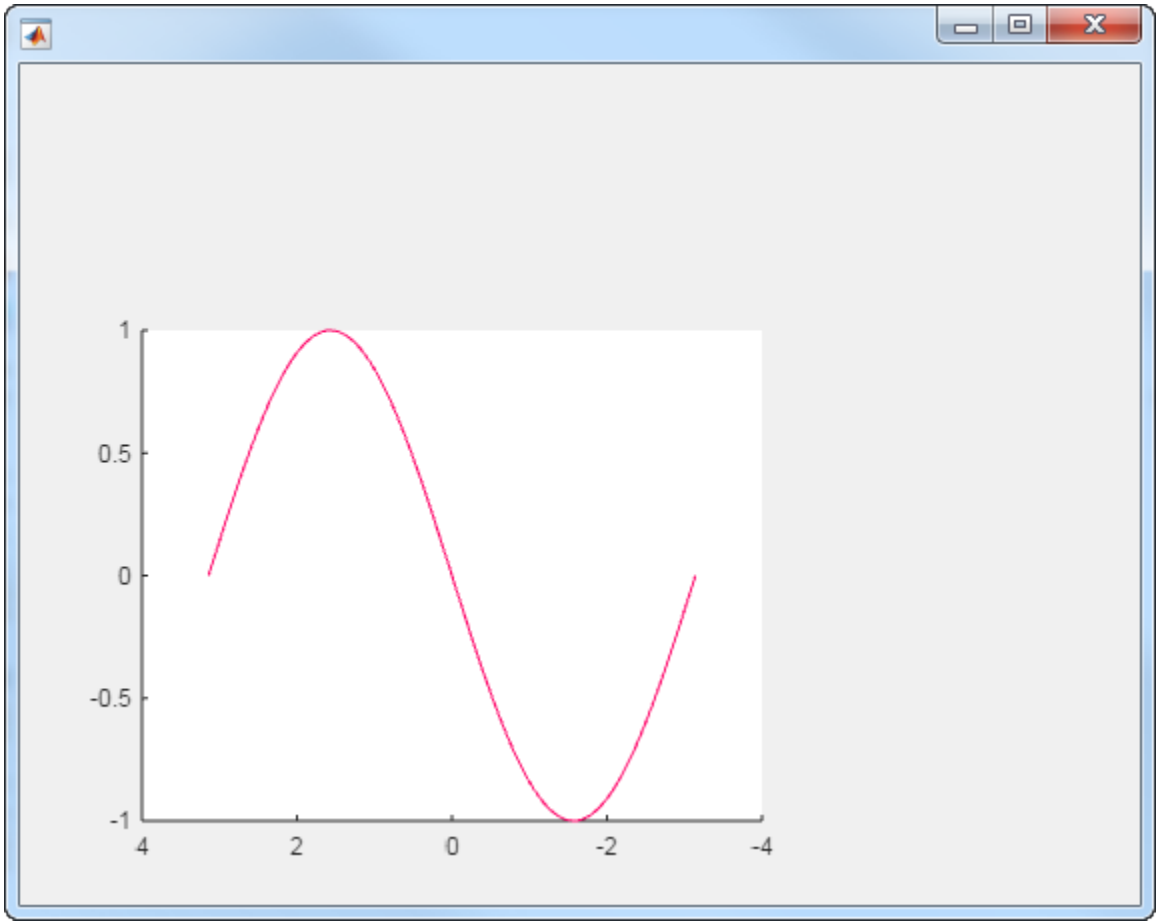
```
hold(ax,'on')
y2 = 5*sin(x) + randn(1,50);
scatter(ax,x,y2)
```



Specify UI Axes Properties

Modify the appearance of the UI axes by setting properties using name-value pair arguments. For example, reverse the *x*-axis direction using the `XDir` name-value pair.

```
fig = uifigure;
ax = uiaxes(fig,'XDir','reverse');
x = linspace(-pi,pi);
y = sin(x);
plot(ax,x,y)
```



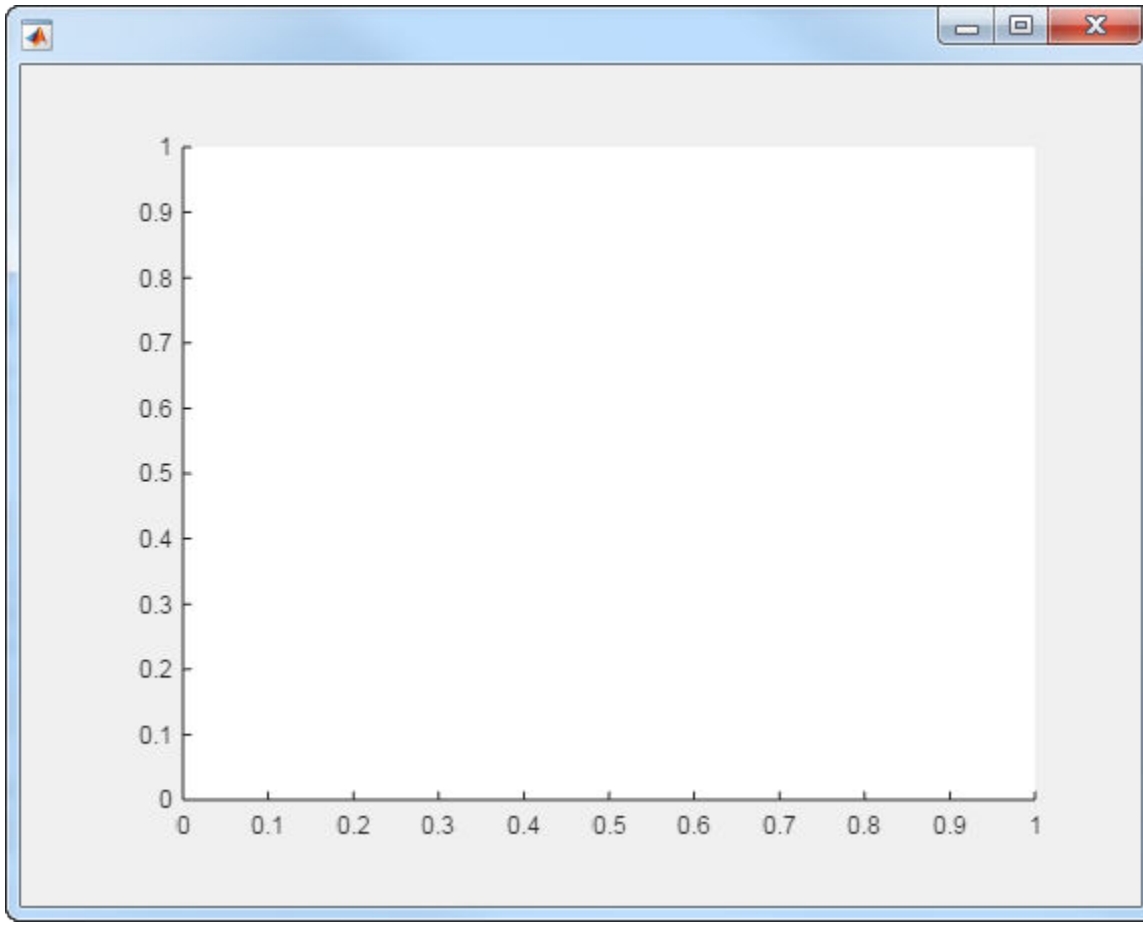
Alternatively, specify properties after the axes is created using dot notation. For example, reverse the *y*-axis direction using dot notation to access the `YDir` property.

```
ax.YDir = 'reverse';
```

Specify UI Axes Position

Specify the UI axes position by setting the `Position` property. Specify the position in pixels.

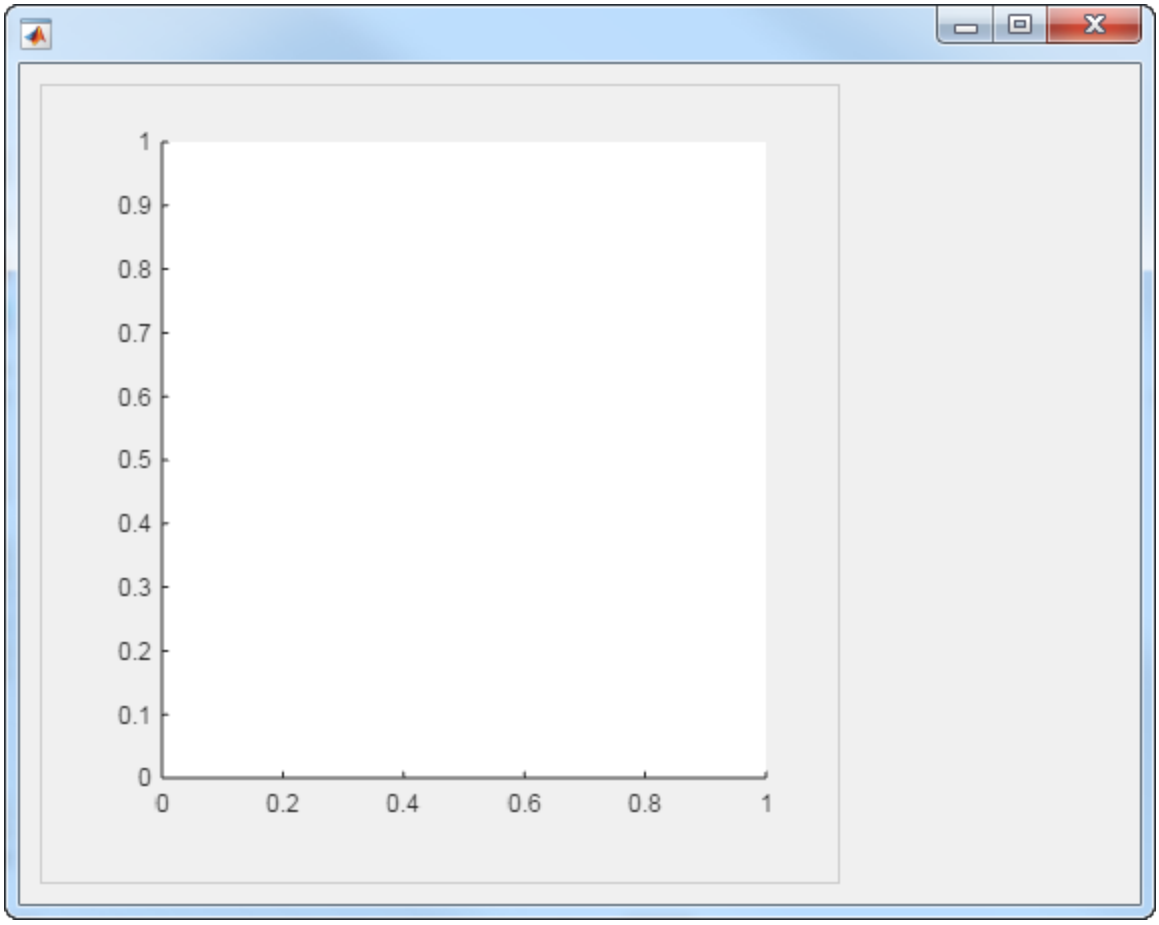
```
fig = uifigure;
ax = uiaxes(fig,'Position',[10 10 550 400]);
```



Add UI Axes to Panel

Add UI axes to a panel within a figure window. Specify the panel and axes positions in pixels.

```
fig = uifigure;
p = uipanel(fig,'Position',[10 10 400 400]);
ax = uiaxes(p,'Position',[10 10 390 390]);
```



Input Arguments

parent — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If you do not specify a parent container, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The properties list here are only a subset. For a full list, see UIAxes.

Example: `'Xscale','linear','YScale','log'`

`XLim`, `YLim`, `ZLim` — Minimum and maximum axis limits
`[0 1]` (default) | two-element vector of the form `[min max]`

Minimum and maximum limits, specified as a two-element vector of the form `[min max]`, where `max` is greater than `min`. You can specify the limits as numeric, categorical, datetime, or duration values. However, the type of values that you specify must match the type of values along the axis.

You can specify both limits or you can specify one limit and let the axes automatically calculate the other. For an automatically calculated minimum or maximum limit, use `-inf` or `inf`, respectively.

Example: `ax.XLim = [0 10]`

Example: `ax.YLim = [-inf 10]`

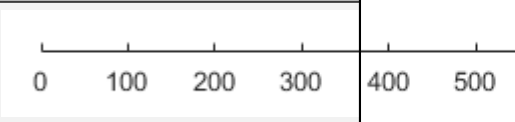
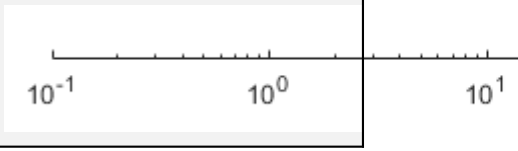
Example: `ax.ZLim = [0 inf]`

Alternatively, use the `xlim`, `ylim`, and `zlim` functions to set the limits. For an example, see “Specify Axis Limits”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `datetime` | `duration`

`XScale`, `YScale`, `ZScale` — Scale of values along axis
`'linear'` (default) | `'log'`

Axis scale, specified as one of these values.

Value	Description	Result
`'linear'`	Linear scale **Example:** `ax.XScale = 'linear'`	
`'log'`	Log scale **Example:** `ax.XScale = 'log'`	

`GridLineStyle` — Line style for grid lines

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style for grid lines, specified as one of the line styles in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

To display the grid lines, use the `grid on` command or set the `XGrid`, `YGrid`, or `ZGrid` property to `'on'`.

Example: `ax.GridLineStyle = '--'`

`Position` — Size and location of axes, including labels and margins

`[10 10 400 300]` (default) | four-element vector

Size and location of axes, including the labels and margins, specified as a four-element vector of the form `[left bottom width height]`. This vector defines a rectangle that encloses the outer bounds of the axes. The `left` and `bottom` elements define the position of the rectangle, measured from the lower left corner to the lower left corner of the parent container. The `width` and `height` define the size of the rectangle. The values are measured in units determined by the `Units` property. By default, the units are pixels.

Output Arguments

ax — UIAxes object
UIAxes object

UIAxes object. Use ax to set properties of the UIAxes after they are created.

Tips

- Zoom, pan, and rotate modes only support a subset of options for UI axes. zoom only supports the on, off, and zoom factor arguments. rotate3d and pan only support the on and off arguments. The recommended way to interact with charts in the UI axes is to use the axes toolbar or the built-in axes interactions. For more information, see “Control Chart Interactivity”.

See Also

Functions
appdesigner | uifigure

Properties
UIAxes

Introduced in R2016a

UIAxes Properties

UI axes appearance and behavior

Description

`UIAxes` properties control the appearance and behavior of a `UIAxes` object. By changing property values, you can modify certain aspects of the axes.

```
ax = uiaxes;
c = ax.Color;
ax.Color = 'blue';
```

The properties listed here are valid for axes in App Designer, or in figures created with the `uifigure` function. For axes used in GUIDE, or in apps created with the `figure` function, see Axes.

Properties

Font

`FontName` — Font name

system supported font name

Font name, specified as a system supported font name. The default font depends on the specific operating system and locale.

If the specified font is not available, then MATLAB uses the best match among the fonts available on the system where the app is running.

Example: `'Arial'`

`FontSize` — Font size

scalar numeric value

Font size, specified as a scalar numeric value. The font size affects the title, axis labels, and tick labels. It also affects any legends or colorbars associated with the axes. The units of measurement are pixels. The default font size depends on the specific operating system and locale.

MATLAB automatically scales some of the text to a percentage of the axes font size.

- Titles and axis labels — 110% of the axes font size by default. To control the scaling, use the `TitleFontSizeMultiplier` and `LabelFontSizeMultiplier` properties.
- Legends and colorbars — 90% of the axes font size by default. To specify a different font size, set the `FontSize` property for the `Legend` or `Colorbar` object instead.

Example: `ax.FontSize = 12`

`FontWeight` — Character thickness
`'normal'` (default) | `'bold'`

Character thickness, specified as `'normal'` or `'bold'`.

MATLAB uses the `FontWeight` property to select a font from those available on your system. Not all fonts have a bold weight. Therefore, specifying a bold font weight can still result in the normal font weight.

`FontAngle` — Character slant
`'normal'` (default) | `'italic'`

Character slant, specified as `'normal'` or `'italic'`.

Not all fonts have both font styles. Therefore, the italic font might look the same as the normal font.

`LabelFontSizeMultiplier` — Scale factor for label font size
`1.1` (default) | numeric value greater than 0

Scale factor for the label font size, specified as a numeric value greater than 0. The scale factor is applied to the value of the `FontSize` property to determine the font size for the *x*-axis, *y*-axis, and *z*-axis labels.

Example: `ax.LabelFontSizeMultiplier = 1.5`

`TitleFontSizeMultiplier` — Scale factor for title font size
`1.1` (default) | numeric value greater than 0

Scale factor for the title font size, specified as a numeric value greater than 0. The scale factor is applied to the value of the `FontSize` property to determine the font size for the title.

Example: `ax.TitleFontSizeMultiplier = 1.75`

`TitleFontWeight` — Title character thickness
`'bold'` (default) | `'normal'`

Title character thickness, specified as one of these values:

- `'bold'` — Thicker characters outlines than normal
- `'normal'` — Default weight as defined by the particular font

Example: `ax.TitleFontWeight = 'normal'`

`FontUnits` — Font size units
`'pixels'` (default) | `'inches'` | `'centimeters'` | `'normalized'` | `'points'`

Font size units, specified as one of the values in this table.

Units	Description
`'points'`	Points. One point equals 1/72 inch.
`'inches'`	Inches.
`'centimeters'`	Centimeters.
`'normalized'`	Interpret font size as a fraction of the axes height. If you resize the axes, the font size modifies accordingly. For example, if the `FontSize` is `0.1` in normalized units, then the text is 1/10 of the height value stored in the axes `Position` property.
`'pixels'`	Pixels. Starting in R2015b, distances in pixels are independent of your system resolution on Windows and Macintosh systems. • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. • On Linux systems, the size of a pixel is determined by your system resolution.

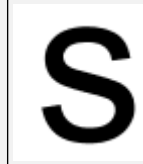
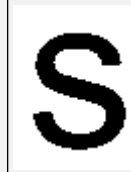
To set both the font size and the font units in a single function call, you first must set the `FontUnits` property so that the `UIAxes` object correctly interprets the specified font size.

`FontSmoothing` — Character smoothing

`'on'` (default) | `'off'`

This property is read-only.

Character smoothing, specified as `'on'` or `'off'`.

Value	Description	Result
`'on'`	Use antialiasing to make text appear smoother on the screen. **Example:** `ax.FontSmoothing = 'on'`	
`'off'`	Do not use antialiasing. Use this setting if the text seems blurry. **Example:** `ax.FontSmoothing = 'off'`	

Ticks

`XTick`, `YTick`, `ZTick` — Tick values

`[]` (default) | vector of increasing values

Tick values, specified as a vector of increasing values. If you do not want tick marks along the axis, then specify an empty vector `[]`. The tick values are the locations along the axis where the tick marks appear. The tick labels are the labels that you see next to each tick mark. Use the `XTickLabels`, `YTickLabels`, and `ZTickLabels` properties to specify the associated labels.

Example: `ax.XTick = [2 4 6 8 10]`

Example: `ax.YTick = 0:10:100`

Alternatively, use the `xticks`, `yticks`, and `zticks` functions to specify the tick values. For an example, see “Specify Axis Tick Values and Labels”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `categorical` | `datetime` | `duration`

`XTickMode`, `YTickMode`, `ZTickMode` — Selection mode for tick values
`'auto'` (default) | `'manual'`

Selection mode for the tick values, specified as one of these values:

- `'auto'` — Automatically select the tick values based on the range of data for the axis.
- `'manual'` — Manually specify the tick values. To specify the values, set the `XTick`, `YTick`, or `ZTick` property.

Example: `ax.XTickMode = 'auto'`

`XTickLabel`, `YTickLabel`, `ZTickLabel` — Tick labels
`''` (default) | cell array of character vectors | string array | categorical array

Tick labels, specified as a cell array of character vectors, string array, or categorical array. If you do not want tick labels to show, then specify an empty cell array `{}`. If you do not specify enough labels for all the ticks values, then the labels repeat.

Tick labels support TeX and LaTeX markup. See the `TickLabelInterpreter` property for more information.

If you specify this property as a categorical array, MATLAB uses the values in the array, not the categories.

As an alternative to setting this property, you can use the `xticklabels`, `yticklabels`, and `zticklabels` functions. For an example, see “Specify Axis Tick Values and Labels”.

Example: `ax.XTickLabel = {'Jan','Feb','Mar','Apr'}`

`XTickLabelMode`, `YTickLabelMode`, `ZTickLabelMode` — Selection mode for tick labels
`'auto'` (default) | `'manual'`

Selection mode for the tick labels, specified as one of these values:

- `'auto'` — Automatically select the tick labels.

- `'manual'` — Manually specify the tick labels. To specify the labels, set the `XTickLabel`, `YTickLabel`, or `ZTickLabel` property.

Example: `ax.XTickLabelMode = 'auto'`

`TickLabelInterpreter` — Tick label interpretation

`'tex'` (default) | `'latex'` | `'none'` | x

Tick label interpretation, specified as one of these values:

- `'tex'` — Interpret labels using a subset of TeX markup.
- `'latex'` — Interpret labels using a subset of LaTeX markup.
- `'none'` — Display literal characters.

TeX Markup

By default, MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the text type and color, and include special characters in the text.

The table that follows lists the supported modifiers when the `TickLabelInterpreter` property is set to `'tex'`, which is the default value. Modifiers remain in effect until the end of the text, except for superscripts and subscripts which only modify the next character or the text within the curly braces `{}`.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (rarely available)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{specifier}`	Set `specifier` as the name of a font family to change the font style. You can use this with other modifiers.	`'\fontname{Courier} text'`

Modifier	Description	Example
`\fontsize{specifier}`	Set `specifier` as a scalar numeric value to change the font size.	`'\fontsize{15} text'`
`\color{specifier}`	Set `specifer` as one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb] {specifier}`	Set `specifier` as a three-element RGB triplet to change the font color.	`'\color[rgb] {0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤
`\ast`	*	`\chi`	χ	`\infty`	∞
`\beta`	β	`\psi`	ψ	`\clubsuit`	♣
`\gamma`	γ	`\omega`	ω	`\diamondsuit`	♦
`\delta`	δ	`\Gamma`	Γ	`\heartsuit`	♥
`\epsilon`	ϵ	`\Delta`	Δ	`\spadesuit`	♠
`\zeta`	ζ	`\Theta`	Θ	`\leftrightarrow`	↔
`\eta`	η	`\Lambda`	Λ	`\leftarrow`	←
`\theta`	θ	`\Xi`	Ξ	`\Leftarrow`	⇐
`\vartheta`	ϑ	`\Pi`	Π	`\uparrow`	↑

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	μ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∋	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨	\langle	〈	\copyright	©

LaTeX Markup

To use LaTeX markup, set the `TickLabelInterpreter` property to `'latex'`. Use dollar symbols around the text, for example, use `'$\int_1^{20} x^2 dx$'` for inline mode or `'$$\int_1^{20} x^2 dx$$'` for display mode.

The displayed text uses the default LaTeX font style. To change the font style, use LaTeX markup within the text. The `FontName`, `FontWeight`, and `FontAngle` properties have no effect.

The maximum size of the text that you can use with the LaTeX interpreter is 1200 characters. For multiline text, this limit reduces by about 10 characters per line. For more information about the LaTeX system, see The LaTeX Project website at www.latex-project.org.

`XTickLabelRotation`, `YTickLabelRotation`, `ZTickLabelRotation` — Tick label rotation

`0` (default) | numeric value in degrees

Tick label rotation, specified as a numeric value in degrees. Positive values give counterclockwise rotation. Negative values give clockwise rotation.

Example: `ax.XTickLabelRotation = 45`

Example: `ax.YTickLabelRotation = 90`

Alternatively, use the `xtickangle`, `ytickangle`, and `ztickangle` functions.

`XMinorTick`, `YMinorTick`, `ZMinorTick` — Minor tick marks

`'off'` | `'on'`

Minor tick marks, specified as one of these values:

- `'off'` — Do not display minor tick marks. This value is the default for an axis with a linear scale.
- `'on'` — Display minor tick marks between the major tick marks on the axis. The space between the major tick marks determines the number of minor tick marks. This value is the default for an axis with a log scale.

Example: `ax.XMinorTick = 'on'`

`TickDir` — Tick mark direction

`'in'` (default) | `'out'` | `'both'`

Tick mark direction, specified as one of these values:

- `'in'` — Direct the tick marks inward from the axis lines. (Default for 2-D views)
- `'out'` — Direct the tick marks outward from the axis lines. (Default for 3-D views)
- `'both'` — Center the tick marks over the axis lines.

Example: `ax.TickDir = 'out'`

`TickDirMode` — Selection mode for `TickDir`
`'auto'` (default) | `'manual'`

Selection mode for the `TickDir` property, specified as one of these values:

- `'auto'` — Automatically select the tick direction based on the current view.
- `'manual'` — Manually specify the tick direction. To specify the tick direction, set the `TickDir` property.

Example: `ax.TickDirMode = 'auto'`

`TickLength` — Tick mark length
`[0.01 0.025]` (default) | two-element vector

Tick mark length, specified as a two-element vector of the form `[2Dlength 3Dlength]`. The first element is the tick mark length in 2-D views and the second element is the tick mark length in 3-D views. Specify the values in units normalized relative to the longest of the visible *x*-axis, *y*-axis, or *z*-axis lines.

Example: `ax.TickLength = [0.02 0.035]`

Rulers

`XLim`, `YLim`, `ZLim` — Minimum and maximum axis limits
`[0 1]` (default) | two-element vector of the form `[min max]`

Minimum and maximum limits, specified as a two-element vector of the form `[min max]`, where `max` is greater than `min`. You can specify the limits as numeric, categorical, datetime, or duration values. However, the type of values that you specify must match the type of values along the axis.

You can specify both limits or you can specify one limit and let the axes automatically calculate the other. For an automatically calculated minimum or maximum limit, use `-inf` or `inf`, respectively.

Example: `ax.XLim = [0 10]`

Example: `ax.YLim = [-inf 10]`

Example: `ax.ZLim = [0 inf]`

Alternatively, use the `xlim`, `ylim`, and `zlim` functions to set the limits. For an example, see “Specify Axis Limits”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `datetime` | `duration`

`XLimMode`, `YLimMode`, `ZLimMode` — Selection mode for axis limits
`'auto'` (default) | `'manual'`

Selection mode for the axis limits, specified as one of these values:

- `'auto'` — Automatically select the axis limits based on the data plotted, which is, the total span of the `XData`, `YData`, or `ZData` of all the objects displayed in the axes.
- `'manual'` — Manually specify the axis limits. To specify the axis limits, set the `XLim`, `YLim`, or `ZLim` property.

Example: `ax.XLimMode = 'auto'`

`XAxis`, `YAxis`, `ZAxis` — Axis ruler
ruler object

Axis ruler, returned as a ruler object. The ruler controls the appearance and behavior of the *x*-axis, *y*-axis, or *z*-axis. Modify the appearance and behavior of a particular axis by accessing the associated ruler and setting ruler properties. The type of ruler that MATLAB creates for each axis depends on the plotted data. For a list of ruler properties, see:

- NumericRuler
- DatetimeRuler
- DurationRuler
- CategoricalRuler

For example, access the ruler for the *x*-axis through the `XAxis` property. Then, change the `Color` property of the ruler, and thus the color of the *x*-axis, to red. Similarly, change the color of the *y*-axis to green.

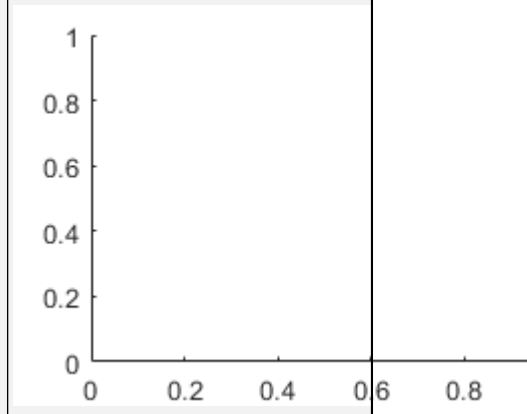
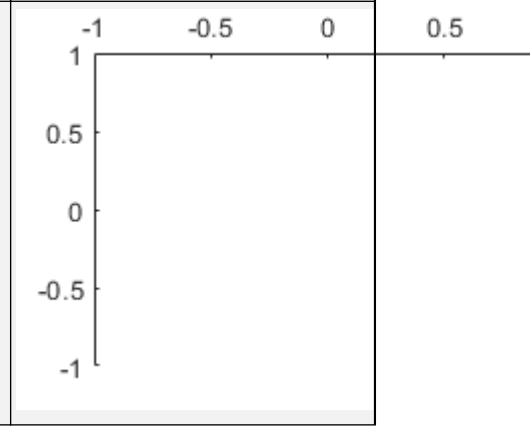
```
ax = gca;
ax.XAxis.Color = 'r';
ax.YAxis.Color = 'g';
```

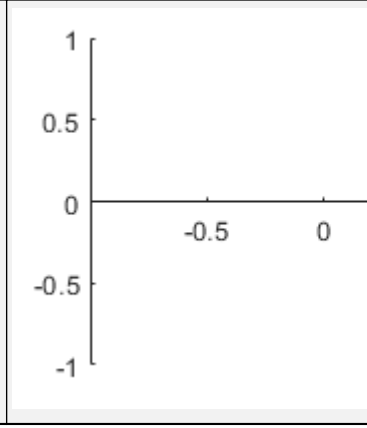
If the `Axes` object has two *y*-axes, then the `YAxis` property stores two ruler objects.

`XAxisLocation` — *x*-axis location

`'bottom'` (default) | `'top'` | `'origin'`

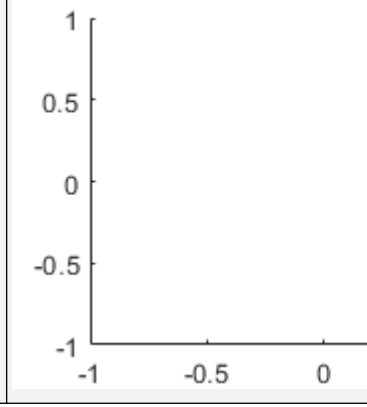
x-axis location, specified as one of the values in this table. This property applies only to 2-D views.

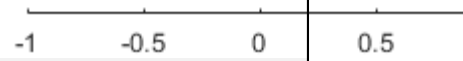
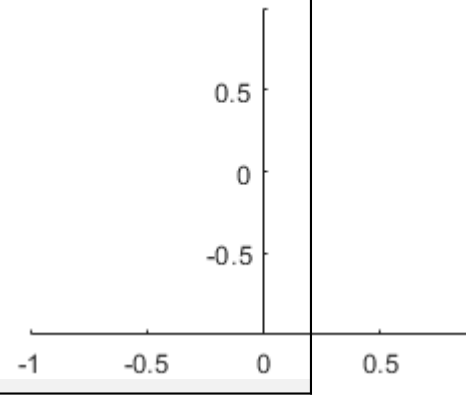
Value	Description	Result
`'bottom'`	Bottom of the axes. **Example:** `ax.XAxisLocation = 'bottom'`	
`'top'`	Top of the axes. **Example:** `ax.XAxisLocation = 'top'`	

Value	Description	Result
`'origin'`	Through the origin point (0,0). **Example:** `ax.XAxisLocation = 'origin'`	

`YAxisLocation` — *y*-axis location
`'left'` (default) | `'right'` | `'origin'`

y-axis location, specified as one of the values in this table. This property applies only to 2-D views.

Value	Description	Result
`'left'`	Left side of the axes. **Example:** `ax.YAxisLocation = 'left'`	

Value	Description	Result
`'right'`	Right side of the axes. **Example:** `ax.YAxisLocation = 'right'`	
`'origin'`	Through the origin point (0,0). **Example:** `ax.YAxisLocation = 'origin'`	

`XColor`, `YColor`, `ZColor` — Color of axis line, tick values, and labels
`[0.15 0.15 0.15]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of the axis line, tick values, and labels in the *x*, *y*, or *z* direction, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The color also affects the grid lines, unless you specify the grid line color using the `GridColor` or `MinorGridColor` property.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `ax.XColor = [1 1 0]`

Example: `ax.YColor = 'yellow'`

Example: `ax.ZColor = '#FFFF00'`

`XColorMode` — Property for setting *x*-axis grid color
`'auto'` (default) | `'manual'`

Property for setting the *x*-axis grid color, specified as `'auto'` or `'manual'`. The mode value only affects the *x*-axis grid color. The *x*-axis line, tick values, and labels always use the `XColor` value, regardless of the mode.

The *x*-axis grid color depends on both the `XColorMode` property and the `GridColorMode` property, as shown here.

XColorMode	GridColorMode	x-Axis Grid Color
`'auto'`	`'auto'`	`GridColor` property
	`'manual'`	`GridColor` property
`'manual'`	`'auto'`	`XColor` property
	`'manual'`	`GridColor` property

The *x*-axis minor grid color depends on both the `XColorMode` property and the `MinorGridColorMode` property, as shown here.

XColorMode	MinorGridColorMode	x-Axis Minor Grid Color
`'auto'`	`'auto'`	`MinorGridColor` property
	`'manual'`	`MinorGridColor` property
`'manual'`	`'auto'`	`XColor` property
	`'manual'`	`MinorGridColor` property

`YColorMode` — Property for setting *y*-axis grid color
`'auto'` (default) | `'manual'`

Property for setting the *y*-axis grid color, specified as `'auto'` or `'manual'`. The mode value only affects the *y*-axis grid color. The *y*-axis line, tick values, and labels always use the `YColor` value, regardless of the mode.

The *y*-axis grid color depends on both the `YColorMode` property and the `GridColorMode` property, as shown here.

YColorMode	GridColorMode	y-Axis Grid Color
`'auto'`	`'auto'`	`GridColor` property
	`'manual'`	`GridColor` property
`'manual'`	`'auto'`	`YColor` property
	`'manual'`	`GridColor` property

The *y*-axis minor grid color depends on both the `YColorMode` property and the `MinorGridColorMode` property, as shown here.

YColorMode	MinorGridColorMode	y-Axis Minor Grid Color
`'auto'`	`'auto'`	`MinorGridColor` property
	`'manual'`	`MinorGridColor` property
`'manual'`	`'auto'`	`YColor` property
	`'manual'`	`MinorGridColor` property

`ZColorMode` — Property for setting *z*-axis grid color
`'auto'` (default) | `'manual'`

Property for setting the *z*-axis grid color, specified as `'auto'` or `'manual'`. The mode value only affects the *z*-axis grid color. The *z*-axis line, tick values, and labels always use the `ZColor` value, regardless of the mode.

The *z*-axis grid color depends on both the `ZColorMode` property and the `GridColorMode` property, as shown here.

ZColorMode	GridColorMode	z-Axis Grid Color
`'auto'`	`'auto'`	`GridColor` property
	`'manual'`	`GridColor` property
`'manual'`	`'auto'`	`ZColor` property

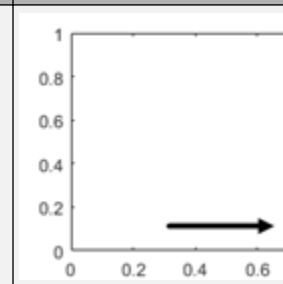
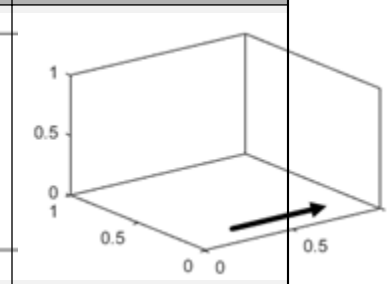
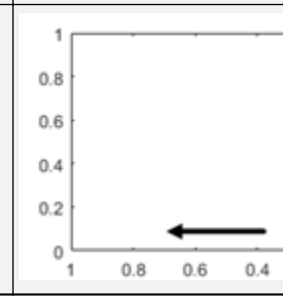
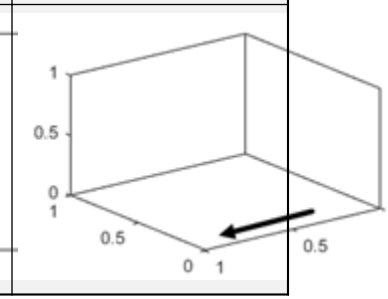
ZColorMode	GridColorMode	z-Axis Grid Color
	`'manual'`	`GridColor` property

The *z*-axis minor grid color depends on both the `ZColorMode` property and the `MinorGridColorMode` property, as shown here.

ZColorMode	MinorGridColorMode	z-Axis Minor Grid Color
`'auto'`	`'auto'`	`MinorGridColor` property
	`'manual'`	`MinorGridColor` property
`'manual'`	`'auto'`	`ZColor` property
	`'manual'`	`MinorGridColor` property

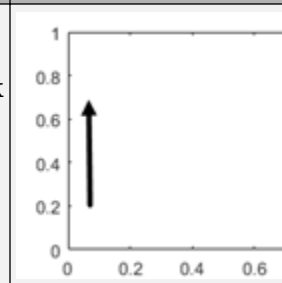
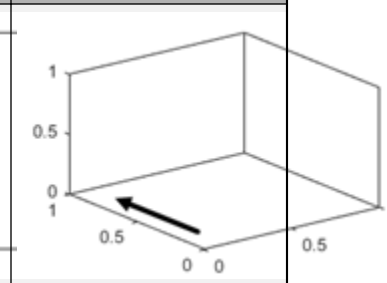
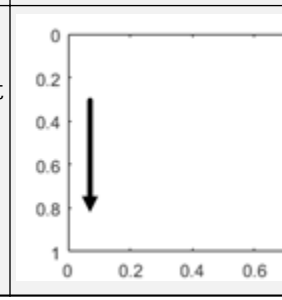
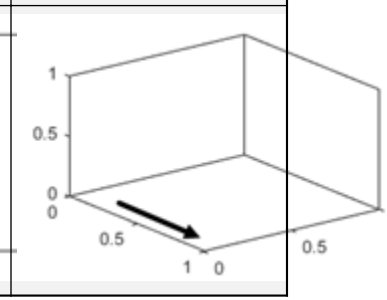
`XDir` — *x*-axis direction
`'normal'` (default) | `'reverse'`

x-axis direction, specified as one of these values.

Value	Description	Result in 2-D	Result in 3-D
`'normal'`	Values increase from left to right. **Example:** `ax.XDir = 'normal'`		
`'reverse'`	Values increase from right to left. **Example:** `ax.XDir = 'reverse'`		

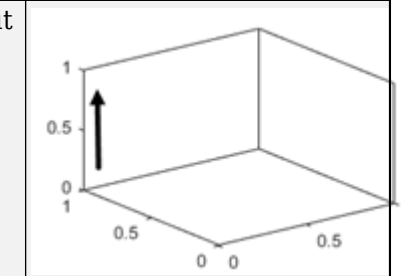
`YDir` — *y*-axis direction
`'normal'` (default) | `'reverse'`

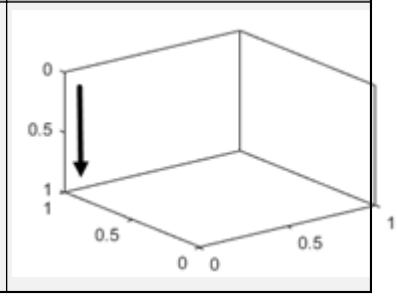
y-axis direction, specified as one of these values.

Value	Description	Result in 2-D	Result in 3-D
`'normal'`	Values increase from bottom to top (2-D view) or front to back (3-D view). **Example:** `ax.YDir = 'normal'`		
`'reverse'`	Values increase from top to bottom (2-D view) or back to front (3-D view). **Example:** `ax.YDir = 'reverse'`		

`ZDir` — *z*-axis direction
`'normal'` (default) | `'reverse'`

z-axis direction, specified as one of these values.

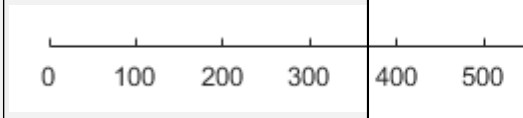
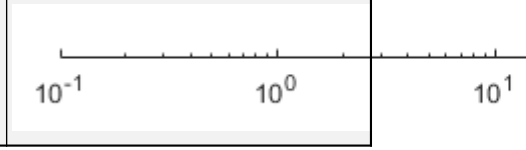
Value	Description	Result in 3-D
`'normal'`	Values increase pointing out of the screen (2-D view) or from bottom to top (3-D view). **Example:** `ax.ZDir = 'normal'`	

Value	Description	Result in 3-D
`'reverse'`	Values increase pointing into the screen (2-D view) or from top to bottom (3-D view). **Example:** `ax.ZDir = 'reverse'`	

`XScale`, `YScale`, `ZScale` — Scale of values along axis

`'linear'` (default) | `'log'`

Axis scale, specified as one of these values.

Value	Description	Result
`'linear'`	Linear scale **Example:** `ax.XScale = 'linear'`	
`'log'`	Log scale **Example:** `ax.XScale = 'log'`	

Grids

`XGrid`, `YGrid`, `ZGrid` — Grid lines

`'off'` (default) | `'on'`

Grid lines, specified as one of these values:

- `'off'` — Do not display the grid lines.
- `'on'` — Display grid lines perpendicular to the axis; for example, along lines of constant *x*, *y*, or *z* values.

Alternatively, use the `grid on` or `grid off` command to set all three properties to `'on'` or `'off'`, respectively. For more information, see `grid`.

Example: `ax.XGrid = 'on'`

Layer — Placement of grid lines and tick marks

`'bottom'` (default) | `'top'`

Placement of grid lines and tick marks in relation to graphic objects, specified as one of these values:

- `'bottom'` — Display tick marks and grid lines under graphics objects.
- `'top'` — Display tick marks and grid lines over graphics objects.

This property affects only 2-D views.

Example: `ax.Layer = 'top'`

GridLineStyle — Line style for grid lines

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style for grid lines, specified as one of the line styles in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

To display the grid lines, use the `grid on` command or set the `XGrid`, `YGrid`, or `ZGrid` property to `'on'`.

Example: `ax.GridLineStyle = '--'`

GridColor — Color of grid lines

`[0.15 0.15 0.15]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of grid lines, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

To set the colors for the axes box outline, use the `XColor`, `YColor`, and `ZColor` properties.

To display the grid lines, use the `grid on` command or set the `XGrid`, `YGrid`, or `ZGrid` property to `'on'`.

Example: `ax.GridColor = [0 0 1]`

Example: `ax.GridColor = 'blue'`

Example: `ax.GridColor = '#0000FF'`

`GridColorMode` — Property for setting grid color
`'auto'` (default) | `'manual'`

Property for setting the grid color, specified as one of these values:

- `'auto'` — Check the values of the `XColorMode`, `YColorMode`, and `ZColorMode` properties to determine the grid line colors for the *x*, *y*, and *z* directions.
- `'manual'` — Use `GridColor` to set the grid line color for all directions.

`GridAlpha` — Grid-line transparency
`0.15` (default) | value in the range `[0,1]`

Grid-line transparency, specified as a value in the range `[0,1]`. A value of `1` means opaque and a value of `0` means completely transparent.

Example: `ax.GridAlpha = 0.5`

`GridAlphaMode` — Selection mode for `GridAlpha`
`'auto'` (default) | `'manual'`

Selection mode for the `GridAlpha` property, specified as one of these values:

- `'auto'` — Default transparency value of `0.15`.
- `'manual'` — Manually specify the transparency value. To specify the value, set the `GridAlpha` property.

Example: `ax.GridAlphaMode = 'auto'`

XMinorGrid, YMinorGrid, ZMinorGrid — Minor grid lines

`'off'` (default) | `'on'`

Minor grid lines, specified as one of these values:

- `'off'` — Do not display grid lines.
- `'on'` — Display grid lines aligned with the minor tick marks of the axis. You do not need to enable minor ticks to display minor grid lines.

Alternatively, use the `grid minor` command to toggle the visibility of the minor grid lines.

Example: `ax.XMinorGrid = 'on'`

MinorGridLineStyle — Line style for minor grid lines

`':'` (default) | `'-'` | `'--'` | `'-.'` | `'none'`

Line style for minor grid lines, specified as one of the line styles shown in this table.

Line Style	Description	Resulting Line
`'-'`	Solid line	
`'--'`	Dashed line	
`':'`	Dotted line	
`'-.'`	Dash-dotted line	
`'none'`	No line	No line

To display minor grid lines, use the `grid minor` command or set the `XMinorGrid`, `YMinorGrid`, or `ZMinorGrid` property to `'on'`.

Example: `ax.MinorGridLineStyle = '-.'`

MinorGridColor — Color of minor grid lines

`[0.1 0.1 0.1]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of minor grid lines, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

To display minor grid lines, use the `grid minor` command or set the `XMinorGrid`, `YMinorGrid`, or `ZMinorGrid` property to `'on'`.

Example: `ax.MinorGridColor = [0 0 1]`

Example: `ax.MinorGridColor = 'blue'`

Example: `ax.MinorGridColor = '#0000FF'`

`MinorGridColorMode` — Property for setting minor grid color
`'auto'` (default) | `'manual'`

Property for setting the minor grid color, specified as one of these values:

- `'auto'` — Check the values of the `XColorMode`, `YColorMode`, and `ZColorMode` properties to determine the grid line colors for the *x*, *y*, and *z* directions.
- `'manual'` — Use `MinorGridColor` to set the minor grid line color for all directions.

`MinorGridAlpha` — Minor grid line transparency
`0.25` (default) | value in the range `[0,1]`

Minor grid line transparency, specified as a value in the range `[0,1]`. A value of `1` means opaque and a value of `0` means completely transparent.

Example: `ax.MinorGridAlpha = 0.5`

`MinorGridAlphaMode` — Selection mode for `MinorGridAlpha`
`'auto'` (default) | `'manual'`

Selection mode for the `MinorGridAlpha` property, specified as one of these values:

- `'auto'` — Default transparency value of `0.25`.
- `'manual'` — Manually specify the transparency value. To specify the value, set the `MinorGridAlpha` property.

Example: `ax.MinorGridAlphaMode = 'auto'`

Labels

`Title` — Text object for axes title
text object

Text object for axes title. To add a title, set the `String` property of the text object. To change the title appearance, such as the font style or color, set other properties. For a complete list, see Text.

```
ax = uiaxes;
ax.Title.String = 'My Graph Title';
ax.Title.FontWeight = 'normal';
```

Alternatively, use the `title` function to add a title and control the appearance.

```
title(ax,'My Title','FontWeight','normal')
```

`XLabel`, `YLabel`, `ZLabel` — Text object for axis label
text object

Text object for axis label. To add an axis label, set the `String` property of the text object. To change the label appearance, such as the font size, set other properties. For a complete list, see Text.

```
ax = uiaxes;
ax.YLabel.String = 'y-Axis Label';
ax.YLabel.FontSize = 12;
```

Alternatively, use the `xlabel`, `ylabel`, and `zlabel` functions to add an axis label and control the appearance.

```
ylabel(ax,'My y-Axis Label','FontSize',12)
```

Legend — Legend associated with axes
`empty GraphicsPlaceholder` (default) | `Legend` object

This property is read-only.

Legend associated with the `UIAxes` object, specified as a `Legend` object. To add a legend to the axes, use the `legend` function. Then, you can use this property to modify the legend. For a complete list of properties, see Legend.

```
ax = uiaxes;
ax.Legend.TextColor = 'red';
```

You also can use this property to determine if the axes has a legend.

```
ax = uiaxes;
lgd = ax.Legend
```

```
if ~isempty(lgd)
    disp('Legend Exists')
end
```

Multiple Plots

`ColorOrder` — Color order

seven predefined colors (default) | three-column matrix of RGB triplets

Color order, specified as a three-column matrix of RGB triplets. This property defines the palette of colors MATLAB uses to create plot objects such as `Line`, `Scatter`, and `Bar` objects. Each row of the array is an RGB triplet. An RGB triplet is a three-element vector whose elements specify the intensities of the red, green, and blue components of a color. The intensities must be in the range [0, 1]. This table lists the default colors.

Colors	`ColorOrder` Matrix

	`[ 0 0.4470 0.7410`
	`0.8500 0.3250 0.0980`
	`0.9290 0.6940 0.1250`
	`0.4940 0.1840 0.5560`
	`0.4660 0.6740 0.1880`
	`0.3010 0.7450 0.9330`
	`0.6350 0.0780 0.1840]`

MATLAB assigns colors to objects according to their order of creation. For example, when plotting lines, the first line uses the first color, the second line uses the second color, and so on. If there are more lines than colors, then the cycle repeats.

Changing the Color Order Before or After Plotting

Starting in R2019b, you can change the color order in either of the following ways:

- Call the `colororder` function to change the color order for all the axes in a figure. The colors of existing plots in the figure update immediately. If you place additional axes into the figure, those axes also use the new color order. If you continue to call plotting commands, those commands also use the new colors.
- Set the `ColorOrder` property on the axes, call the `hold` function to set the axes hold state to `'on'`, and then call the desired plotting functions. This is like calling the `colororder` function, but in this case you are setting the color order for the specific

axes, not the entire figure. Setting the `hold` state to `'on'` is necessary to ensure that subsequent plotting commands do not reset the axes to use the default color order.

If you are using R2019a or an earlier release, changing the color order matrix does not affect existing plots. To change colors in an existing plot, you must set the `ColorOrder` property, and then set the axes hold state to `'on'` before calling any plotting functions.

The behavior of this property changed in R2019b. For more information, see:

- "Changing ColorOrder or LineStyleOrder on the axes affects existing plots immediately" on page 1-17888
- "Indexing scheme for ColorOrder and LineStyleOrder might change plot colors and line styles" on page 1-17888

`ColorOrderIndex` — Color order index

1 (default) | positive integer

Color order index, specified as a positive integer. This property specifies the next color MATLAB selects from the axes `ColorOrder` property when it creates the next plot object such as a `Line`, `Scatter`, or `Bar` object. For example, if the color order index value is `1`, then the next object added to the axes uses the first color in the `ColorOrder` matrix. If the index value exceeds the number of colors in the `ColorOrder` matrix, then the index value modulo of the number of colors in the `ColorOrder` matrix determines the color of the next object.

When the `NextPlot` property of the axes is set to `'add'`, then the color order index value increases every time you add a new plot to the axes. To start again with first color, set the `ColorOrderIndex` property to `1`.

The behavior of this property changed in R2019b. For more information, see "Indexing scheme for ColorOrder and LineStyleOrder might change plot colors and line styles" on page 1-17888.

`LineStyleOrder` — Line style order

`'-'` solid line (default) | character vector | cell array of character vectors | string array

Line style order, specified as a character vector, a cell array of character vectors, or a string array. This property lists the line styles that MATLAB uses to display multiple plot lines in the axes. MATLAB assigns styles to lines according to their order of creation. It changes to the next line style only after cycling through all the colors in the `ColorOrder` property with the current line style. The default `LineStyleOrder` has only one line style, `'-'`.

To customize the line style order, create a cell array of character vectors or a string array. Specify each element of the array as a line specifier or marker specifier from the following tables. You can combine a line and a marker specifier into a single element, such as `'-*'`.

Line Specifier	Description
`'-'` (default)	Solid line
`'--'`	Dashed line
`':'`	Dotted line
`'-.'`	Dash-dotted line

Marker Specifier	Description
`'+'`	Plus sign markers
`'o'`	Circle markers
`'*'`	Star markers
`'.'`	Point markers
`'x'`	Cross markers
`'s'`	Square markers
`'d'`	Diamond markers
`'^'`	Upward-pointing triangle markers
`'v'`	Downward-pointing triangle markers
`'>'`	Right-pointing triangle markers
`'<'`	Left-pointing triangle markers
`'p'`	Five-pointed star (pentagram) markers
`'h'`	Six-pointed star (hexagram) markers

Changing Line Style Order Before or After Plotting

Starting in R2019b, you can change the line style order before or after plotting into the axes. When you set the `LineStyleOrder` property to a new value, MATLAB updates the styles of any lines that are in the axes. If you continue plotting into the axes, your plotting commands continue using the line styles from the updated list.

If you are using R2019a or an earlier release, you must change the line style order before plotting. Set the value of the `LineStyleOrder` property, and then call the `hold` function to set the axes hold state to `'on'` before calling any plotting functions.

The behavior of this property changed in R2019b. For more information, see:

- "Changing ColorOrder or LineStyleOrder on the axes affects existing plots immediately" on page 1-17888
- "Indexing scheme for ColorOrder and LineStyleOrder might change plot colors and line styles" on page 1-17888

`LineStyleOrderIndex` — Line style order index
1 (default) | positive integer

Line style order index, specified as a positive integer. This property specifies the next line style MATLAB selects from the axes `LineStyleOrder` property to create the next plot line. For example, if this property is set to `1`, then the next plot line you add to the axes uses the first item in the `LineStyleOrder` property. If the index value exceeds the number of line styles in the `LineStyleOrder` array, then the index value modulo of the number of elements in the `LineStyleOrder` array determines the style of the next line.

When the `NextPlot` property of the axes is set to `'add'`, MATLAB increments the index value after cycling through all the colors in the `ColorOrder` property with the current line style. To start again with first line style, set the `LineStyleOrderIndex` property to `1`.

The behavior of this property changed in R2019b. For more information, see "Indexing scheme for ColorOrder and LineStyleOrder might change plot colors and line styles" on page 1-17888.

`NextPlot` — Properties to reset
`'replacechildren'` (default) | `'add'` | `'replaceall'` | `'replace'`

Properties to reset when adding a new plot to the axes, specified as one of these values:

- `'add'` — Add new plots to the existing axes. Do not delete existing plots or reset axes properties before displaying the new plot.
- `'replacechildren'` — Delete existing plots before displaying the new plot. Reset the `ColorOrderIndex` and `LineStyleOrderIndex` properties to `1`, but do not reset other axes properties. The next plot added to the axes uses the first color and line style based on the `ColorOrder` and `LineStyle` order properties. This value is similar to using `cla` before every new plot.

- `'replace'` — Delete existing plots and reset axes properties, except `Position` and `Units`, to their default values before displaying the new plot.
- `'replaceall'` — Delete existing plots and reset axes properties, except `Position` and `Units`, to their default values before displaying the new plot. This value is similar to using `cla reset` before every new plot.

Note For `UIAxes` objects with only one *y*-axis, the `'replace'` and `'replaceall'` property values are equivalent. For `Axes` objects with two *y*-axes, the `'replace'` value affects only the active side while the `'replaceall'` value affects both sides.

Figures created with the `uifigure` function also have a `NextPlot` property. Alternatively, you can use the `newplot` function to prepare figures and axes for subsequent graphics commands.

`SortMethod` — Order for rendering objects

`'depth'` | `'childorder'`

Order for rendering objects, specified as one of these values:

- `'depth'` — Draw objects in back-to-front order based on the current view. Use this value to ensure that objects in front of other objects are drawn correctly.
- `'childorder'` — Draw objects in the order in which they are created by graphics functions, without considering the relationship of the objects in three dimensions. This value can result in faster rendering, particularly if the figure is very large, but also can result in improper depth sorting of the objects displayed.

Color and Transparency Maps

`Colormap` — Color map

parula (default) | `m`-by-`3` array of RGB triplets

Color map, specified as an `m`-by-`3` array of RGB (red, green, blue) triplets that define `m` individual colors.

Example: `ax.Colormap = [1 0 1; 0 0 1; 1 1 0]` sets the color map to three colors: magenta, blue, and yellow.

MATLAB accesses these colors by their row number.

Alternatively, use the `colormap` function to change the color map.

ColorScale — Scale for color mapping
`'linear'` (default) | `'log'`

Scale for color mapping, specified as one of these values:

- `'linear'` — Linear scale. The tick values along the colorbar also use a linear scale.
- `'log'` — Log scale. The tick values along the colorbar also use a log scale.

Example: `ax.ColorScale = 'log'`

CLim — Color limits
`[0 1]` (default) | two-element vector of the form `[cmin cmax]`

Color limits for objects in axes that use the colormap, specified as a two-element vector of the form `[cmin cmax]`. This property determines how data values map to the colors in the colormap where:

- `cmin` specifies the data value that maps to the first color in the colormap.
- `cmax` specifies the data value that maps to the last color in the colormap.

The `Axes` object interpolates data values between `cmin` and `cmax` across the colormap. Values outside this range use either the first or last color, whichever is closest.

CLimMode — Selection mode for CLim
`'auto'` (default) | `'manual'`

Selection mode for the `CLim` property, specified as one of these values:

- `'auto'` — Automatically select the limits based on the color data of the graphics objects contained in the axes.
- `'manual'` — Manually specify the values. To specify the values, set the `CLim` property. The values do not change when the limits of the axes children change.

Alphamap — Transparency map
array of 64 values from `0` to `1` (default) | array of finite alpha values from `0` to `1`

Transparency map, specified as an array of finite alpha values that progress linearly from `0` to `1`. The size of the array can be m-by-1 or 1-by-m. MATLAB accesses alpha values by their index in the array. Alphamaps can be any length.

AlphaScale — Scale for transparency mapping
`'linear'` (default) | `'log'`

Scale for transparency mapping, specified as one of these values:

- `'linear'` — Linear scale
- `'log'` — Log scale

Example: `ax.AlphaScale = 'log'`

`ALim` — Alpha limits

`[0 1]` (default) | two-element vector of the form `[amin amax]`

Alpha limits, specified as a two-element vector of the form `[amin amax]`. This property affects the `AlphaData` values of graphics objects, such as surface, image, and patch objects. This property determines how the `AlphaData` values map to the figure alpha map, where:

- `amin` specifies the data value that maps to the first alpha value in the figure alpha map.
- `amax` specifies the data value that maps to the last alpha value in the figure alpha map.

The `UIAxes` object interpolates data values between `amin` and `amax` across the figure alpha map. Values outside this range use either the first or last alpha map value, whichever is closest.

The `Alphamap` property of the figure contains the alpha map. For more information, see the `alpha` function.

`ALimMode` — Selection mode for `ALim`

`'auto'` (default) | `'manual'`

Selection mode for the `ALim` property, specified as one of these values:

- `'auto'` — Automatically select the limits based on the `AlphaData` values of the graphics objects contained in the axes.
- `'manual'` — Manually specify the alpha limits. To specify the alpha limits, set the `ALim` property.

Box Styling

`Color` — Color of plot area

`[1 1 1]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of plot area, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The color affects the area defined by the `InnerPosition` property value.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `ax.Color = [0 0 1]`

Example: `ax.Color = 'blue'`

Example: `ax.Color = '#0000FF'`

`BackgroundColor` — Color of margin around plot area
`[0.94 0.94 0.94]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of margin around plot area, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The color affects the margin between the areas defined by the `InnerPosition` and `OuterPosition` property values.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `ax.BackgroundColor = [0 0 1]`

Example: `ax.BackgroundColor = 'blue'`

Example: `ax.BackgroundColor = '#0000FF'`

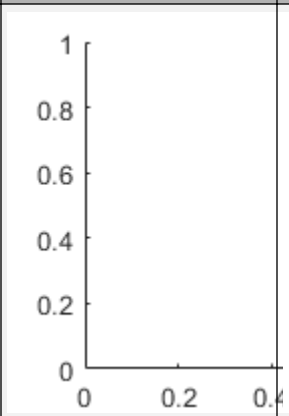
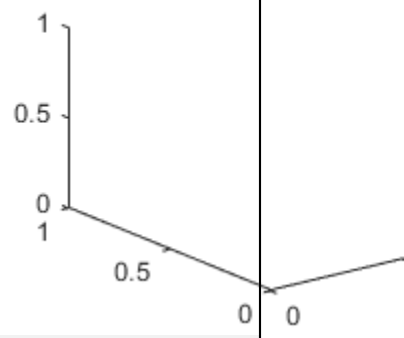
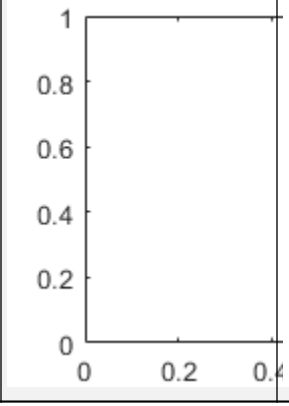
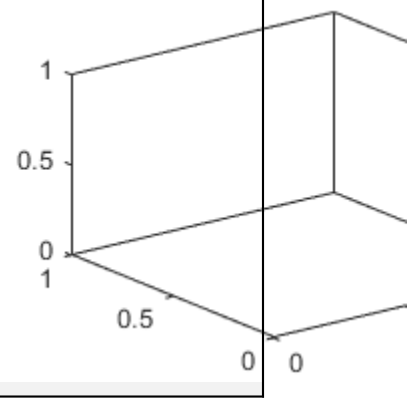
`LineWidth` — Line width
`0.5` (default) | positive numeric value

Line width of axes outline, tick marks, and grid lines, specified as a positive numeric value in point units. One point equals 1/72 inch.

Example: `ax.LineWidth = 1.5`

Box — Box outline
`'off'` (default) | `'on'`

Box outline, specified as `'off'` or `'on'`.

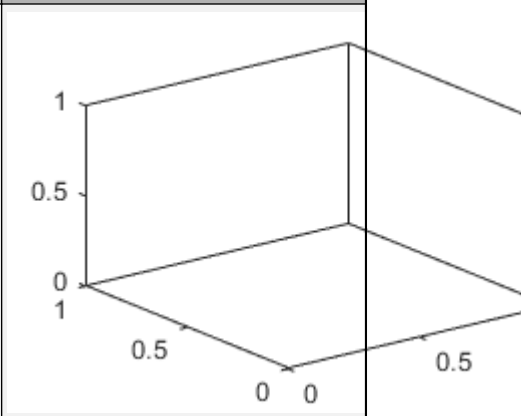
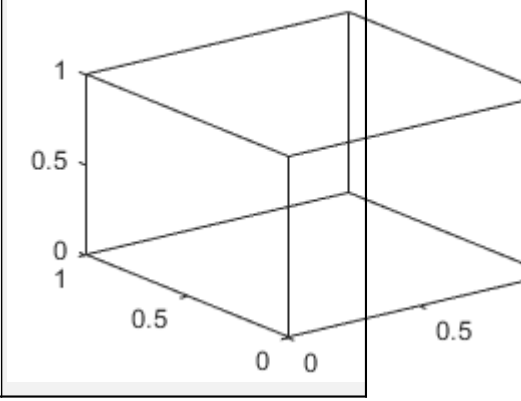
Value	Description	2-D Result	3-D Result
`'off'`	Do not display the box outline around the axes. **Example:** `ax.Box = 'off'`		
`'on'`	Display the box outline around the axes. For 3-D views, use the `BoxStyle` property to change extent of the outline. **Example:** `ax.Box = 'on'`		

The `XColor`, `YColor`, and `ZColor` properties control the color of the outline.

Example: `ax.Box = 'on'`

`BoxStyle` — Box outline style
`'back'` (default) | `'full'`

Box outline style, specified as `'back'` or `'full'`. This property affects only 3-D views.

Value	Description	Result
`'back'`	Outline the back planes of the 3-D box. **Example:** `ax.BoxStyle = 'back'`	
`'full'`	Outline the entire 3-D box. **Example:** `ax.BoxStyle = 'full'`	

`Clipping` — Clipping of objects to axes limits
`'on'` (default) | `'off'`

Clipping of objects to the axes limits, specified as either `'on'` or `'off'`. The clipping behavior of an object within the `Axes` object depends on both the `Clipping` property of the `Axes` object and the `Clipping` property of the individual object. The property value of the `Axes` object has these effects:

- `'on'` — Enable each individual object within the axes to control its own clipping behavior based on the `Clipping` property value for the object.
- `'off'` — Disable clipping for all objects within the axes, regardless of the `Clipping` property value for the individual objects. Parts of objects can appear outside of the

axes limits. For example, parts can appear outside the limits if you create a plot, use the `hold on` command, freeze the axis scaling, and then add a plot that is larger than the original plot.

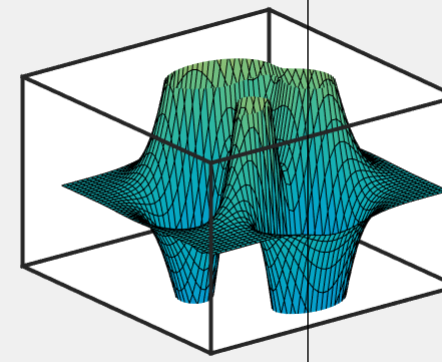
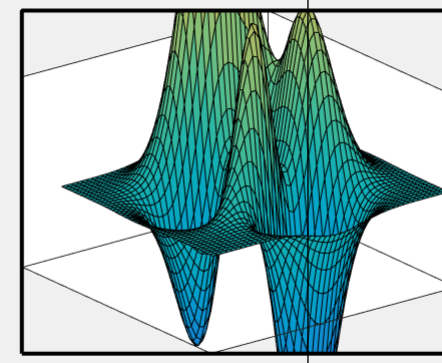
This table lists the results for different combinations of `Clipping` property values.

Clipping Property for Axes Object	Clipping Property for Individual Object	Result
`'on'`	`'on'`	Individual object is clipped. Others might or might not be.
`'on'`	`'off'`	Individual object is not clipped. Others might or might not be.
`'off'`	`'on'`	All objects are unclipped.
`'off'`	`'off'`	All objects are unclipped.

`ClippingStyle` — Clipping boundaries
`'3dbox'` (default) | `'rectangle'`

Clipping boundaries, specified as one of the values in this table. If a plot contains markers, then as long as the data point lies within the axes limits, MATLAB draws the entire marker.

The `ClippingStyle` property has no effect if the `Clipping` property is set to `'off'`.

Value	Descriptions	Illustration of Boundary Region
`'3dbox'`	Clip plotted objects to the six sides of the axes box defined by the axis limits. Thick lines might display outside the axes limits.	
`'rectangle'`	Clip plotted objects to a rectangular boundary enclosing the axes in any given view. Clip thick lines at the axes limits.	

`AmbientLightColor` — Background light color

`[1 1 1]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background light color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name. The background light is a directionless light that shines uniformly on all objects in the axes. To add light, use the `light` function.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	
`'none'`	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	

RGB Triplet	Hexadecimal Color Code	Appearance
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: `ax.AmbientLightColor = [1 0 1]`

Example: `ax.AmbientLightColor = 'magenta'`

Example: `ax.AmbientLightColor = '#FF00FF'`

Position

`Position` — Size and location of axes, including labels and margins
`[10 10 400 300]` (default) | four-element vector

Size and location of axes, including the labels and margins, specified as a four-element vector of the form `[left bottom width height]`. This vector defines a rectangle that encloses the outer bounds of the axes. The `left` and `bottom` elements define the position of the rectangle, measured from the lower left corner to the lower left corner of the parent container. The `width` and `height` define the size of the rectangle. The values are measured in units determined by the `Units` property. By default, the units are pixels.

`InnerPosition` — Size and position of inner axes, excluding labels and margins
four-element vector

This property is read-only.

Size and position of the inner axes, excluding labels and margins, returned as a four-element vector of the form `[left bottom width height]`. The `left` and `bottom` elements define the position of the rectangle, measured from the lower left corner to the lower left corner of the parent container. The `width` and `height` define the size of the rectangle. The values are measured in units determined by the `Units` property. By default, the units are pixels.

MATLAB automatically sets `InnerPosition` to the largest possible values that conform to all other properties. Other `UIAxes` properties that affect the axes size and shape include `Position`, `DataAspectRatio` and `PlotBoxAspectRatio`.

`OuterPosition` — Size and location of axes, including labels and margins
`[10 10 400 300]` (default) | four-element vector

Size and location of the axes, including the labels and margins, specified as a four-element vector of the form `[left bottom width height]`. Position values are relative to the parent container. By default, the values are measured in pixels.

This property value is identical to the `Position` property value.

`TightInset` — Margin for text labels
four-element vector of the form `[left bottom right top]`

This property is read-only.

Margin for text labels, specified as a four-element vector of the form `[left bottom right top]`. The elements define the distances between the bounds of the `InnerPosition` property and the extent of the axes text labels and title. By default, the values are measured in pixels. To change the units, set the `Units` property.

`Units` — Position units
`'pixels'` (default)

Position units, specified as `'pixels'`.

- On Windows systems, a pixel is 1/96th of an inch.
- On Macintosh systems, a pixel is 1/72nd of an inch.
- On Linux systems, the size of a pixel is determined by your system resolution.

`DataAspectRatio` — Relative length of data units
`[1 1 1]` (default) | three-element vector of the form `[dx dy dz]`

Relative length of data units along each axis, specified as a three-element vector of the form `[dx dy dz]`. This vector defines the relative *x*, *y*, and *z* data scale factors. For example, specifying this property as `[1 2 1]` sets the length of one unit of data in the *x*-direction to be the same length as two units of data in the *y*-direction and one unit of data in the *z*-direction.

Alternatively, use the `daspect` function to change the data aspect ratio.

Example: `ax.DataAspectRatio = [1 1 1]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`DataAspectRatioMode` — Data aspect ratio mode
`'auto'` (default) | `'manual'`

Data aspect ratio mode, specified as one of these values:

- `'auto'` — Automatically select values that make best use of the available space. If `PlotBoxAspectRatioMode` and `CameraViewAngleMode` are also set to `'auto'`, then enable "stretch-to-fill" behavior. Stretch the axes so that it fills the available space as defined by the `Position` property.
- `'manual'` — Disable the "stretch-to-fill" behavior and use the manually specified data aspect ratio. To specify the values, set the `DataAspectRatio` property.

`PlotBoxAspectRatio` — Relative length of each axis
three-element vector of the form `[px py pz]`

Relative length of each axis, specified as a three-element vector of the form `[px py pz]` defining the relative *x*-axis, *y*-axis, and *z*-axis scale factors. The plot box is a box enclosing the axes data region as defined by the axis limits.

Alternatively, use the `pbaspect` function to change the data aspect ratio.

If you specify the axis limits, data aspect ratio, and plot box aspect ratio, then MATLAB ignores the plot box aspect ratio. It adheres to the axis limits and data aspect ratio.

Example: `ax.PlotBoxAspectRatio = [1 0.75 0.75]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`PlotBoxAspectRatioMode` — Selection mode for `PlotBoxAspectRatio`
`'auto'` (default) | `'manual'`

Selection mode for the `PlotBoxAspectRatio` property, specified as one of these values:

- `'auto'` — Automatically select values that make best use of the available space. If `DataAspectRatioMode` and `CameraViewAngleMode` also are set to `'auto'`, then enable "stretch-to-fill" behavior. Stretch the `Axes` object so that it fills the available space as defined by the `Position` property.
- `'manual'` — Disable the "stretch-to-fill" behavior and use the manually specified plot box aspect ratio. To specify the values, set the `PlotBoxAspectRatio` property.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then

this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places a UI axes component in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
ax = uiaxes(g);
ax.Layout.Row = 3;
ax.Layout.Column = 2;
```

To make the axes span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this axes spans columns `2` through `3`:

```
ax.Layout.Column = [2 3];
```

View

`View` — Azimuth and elevation of view

`[0 90]` (default) | two-element vector of the form `[azimuth elevation]`

Azimuth and elevation of view, specified as a two-element vector of the form `[azimuth elevation]` defined in degree units. Alternatively, use the `view` function to set the view.

Example: `ax.View = [45 45]`

`Projection` — Type of projection onto 2-D screen

`'orthographic'` (default) | `'perspective'`

Type of projection onto a 2-D screen, specified as one of these values:

- `'orthographic'` — Maintain the correct relative dimensions of graphics objects regarding the distance of a given point from the viewer, and draw lines that are parallel in the data parallel on the screen.
- `'perspective'` — Incorporate foreshortening, which enables you to perceive depth in 2-D representations of 3-D objects. Perspective projection does not preserve the relative dimensions of objects. Instead, it displays a distant line segment smaller than a nearer line segment of the same length. Lines that are parallel in the data might not appear parallel on screen.

`CameraPosition` — Camera location

three-element vector of the form `[x y z]`

Camera location, or the viewpoint, specified as a three-element vector of the form `[x y z]`. This vector defines the axes coordinates of the camera location, which is the point from which you view the axes. The camera is oriented along the view axis, which is a straight line that connects the camera position and the camera target. For an illustration, see “Camera Graphics Terminology”.

If the `Projection` property is set to `'perspective'`, then as you change the `CameraPosition` setting, the amount of perspective also changes.

Alternatively, use the `campos` function to set the camera location.

Example: `ax.CameraPosition = [0.5 0.5 9]`

Data Types: `single` | `double`

`CameraPositionMode` — Selection mode for `CameraPosition`
`'auto'` (default) | `'manual'`

Selection mode for the `CameraPosition` property, specified as one of these values:

- `'auto'` — Automatically set `CameraPosition` along the view axis. Calculate the position so that the camera lies a fixed distance from the target along the azimuth and elevation specified by the current view, as returned by the `view` function. Functions like `rotate3d`, `zoom`, and `pan`, change this mode to `'auto'` to perform their actions.
- `'manual'` — Manually specify the value. To specify the value, set the `CameraPosition` property.

`CameraTarget` — Camera target point
three-element vector of the form `[x y z]`

Camera target point, specified as a three-element vector of the form `[x y z]`. This vector defines the axes coordinates of the point. The camera is oriented along the view axis, which is a straight line that connects the camera position and the camera target. For an illustration, see “Camera Graphics Terminology”.

Alternatively, use the `camtarget` function to set the camera target.

Example: `ax.CameraTarget = [0.5 0.5 0.5]`

Data Types: `single` | `double`

`CameraTargetMode` — Selection mode for `CameraTarget`
`'auto'` (default) | `'manual'`

Selection mode for the `CameraTarget` property, specified as one of these values:

- `'auto'` — Position the camera target at the centroid of the axes plot box.
- `'manual'` — Use the manually specified camera target value. To specify a value, set the `CameraTarget` property.

`CameraUpVector` — Vector defining upwards direction
three-element direction vector of the form `[x y z]`

Vector defining upwards direction, specified as a three-element direction vector of the form `[x y z]`. For 2-D views, the default value is `[0 1 0]`. For 3-D views, the default value is `[0 0 1]`. For an illustration, see "Camera Graphics Terminology".

Alternatively, use the `camup` function to set the upwards direction.

Example: `ax.CameraUpVector = [sin(45) cos(45) 1]`

`CameraUpVectorMode` — Selection mode for `CameraUpVector`
`'auto'` (default) | `'manual'`

Selection mode for the `CameraUpVector` property, specified as one of these values:

- `'auto'` — Automatically set the value to `[0 0 1]` for 3-D views so that the positive *z*-direction is up. Set the value to `[0 1 0]` for 2-D views so that the positive *y*-direction is up.
- `'manual'` — Manually specify the vector defining the upwards direction. To specify a value, set the `CameraUpVector` property.

`CameraViewAngle` — Field of view
`6.6086` (default) | scalar angle in range [0,180)

Field of view, specified as a scalar angle greater than 0 and less than or equal to 180. Changing the camera view angle affects the size of graphics objects displayed in the axes, but does not affect the degree of perspective distortion. The greater the angle, the larger the field of view and the smaller objects appear in the scene. For an illustration, see "Camera Graphics Terminology".

Example: `ax.CameraViewAngle = 15`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

CameraViewAngleMode — Selection mode for CameraViewAngle

'auto' (default) | 'manual'

Selection mode for the CameraViewAngle property, specified as one of these values:

- 'auto' — Automatically select the field of view as the minimum angle that captures the entire scene, up to 180 degrees.
- 'manual' — Manually specify the field of view. To specify a value, set the CameraViewAngle property.

Interactivity

Toolbar — Data exploration toolbar

AxesToolbar object (default)

Data exploration toolbar, which is an AxesToolbar object. The toolbar appears at the top-right corner of the UI axes when you hover over it.

The toolbar buttons depend on the contents of the UI axes, but typically include zooming, panning, rotating, exporting, and restoring the original view. You can customize the toolbar buttons using the axtoolbar and axtoolbarbtn functions.

If you do not want the toolbar to appear when you hover over the UI axes, set the Visible property of the AxesToolbar object to 'off'.

```
ax = uiaxes;
ax.Toolbar.Visible = 'off';
```

For more information, see AxesToolbar.

Interactions — Interactions

array of interactions | [ ]

Interactions, specified as an array of interaction objects or an empty array. The interactions you specify are available within your chart through gestures. You do not have to select any axes toolbar buttons to use them. For example, a panInteraction object enables dragging to pan within a chart. For a list of interaction objects, see “Control Chart Interactivity”.

The default set of interactions depends on the type of chart you are displaying. You can replace the default set with a new set of interactions, but you cannot access or modify any of the interactions in the default set. For example, this code replaces the default set of interactions with the `panInteraction` and `zoomInteraction` objects.

```
ax = uiaxes;
ax.Interactions = [panInteraction zoomInteraction];
```

To remove all interactions from the axes, set this property to an empty array. To temporarily disable the current set of interactions, call the `disableDefaultInteractivity` function. You can reenable them by calling the `enableDefaultInteractivity` function.

Note Interaction objects are not returned by `findobj` or `findall`, and they are not copied by `copyobj`.

`Visible` — State of visibility

`'on'` (default) | `'off'`

State of visibility, specified as one of these values:

- `'on'` — Display the object.
- `'off'` — Hide the object without deleting it. You still can access the properties of an invisible object.

`CurrentPoint` — Location of mouse pointer

2-by-3 array

Location of mouse pointer, specified as a 2-by-3 array. The `CurrentPoint` property contains the (*x*,*y*,*z*) coordinates of the mouse pointer with respect to the axes. The returned array is of the form:

```
[xfront yfront zfront
 xback  yback  zback]
```

The two points indicate the location of the last mouse click. However, if the figure has a `WindowButtonMotionFcn` callback defined, then the points indicate the last location of the mouse pointer. The figure also has a `CurrentPoint` property.

The values of the current point when using perspective projection can be different from the same point in orthographic projection because the shape of the axes volume can be different.

Orthogonal Projection

When using orthogonal projection, the values depend on whether the click is within the axes or outside the axes.

- If the click is inside the axes, the two points lie on the line that is perpendicular to the plane of the screen and that passes through the pointer. The coordinates are the points where this line intersects the front and back surfaces of the axes volume (which is defined by the axes x, y, and z limits). The first row is the point nearest to the camera position. The second row is the point farthest from the camera position. This is true for both 2-D and 3-D views.
- If the click is outside the axes, but within the figure, then the points lie on a line that passes through the pointer and is perpendicular to the camera target and camera position planes. The first row is the point in the camera position plane. The second row is the point in the plane of the camera target.

Perspective Projection

Clicking outside of the `UIAxes` object in perspective projection returns the front point as the current camera position. Only the back point updates with the coordinates of a point that lies on a line extending from the camera position through the pointer and intersecting the camera target at that point.

Callbacks

`SizeChangedFcn` — UI axes resize callback function

`''` (default) | function handle | cell array | character vector

UI axes resize callback function, specified as one of these values:

- Function handle
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector that is a valid MATLAB expression. MATLAB evaluates this expression in the base workspace.

Define this callback function to control the layout when the size of the UI axes change.

The `SizeChangedFcn` callback executes under these circumstances:

- The axes becomes visible for the first time.
- The axes is visible while its drawable area changes. The drawable area is the area inside the outer bounds of the axes.
- The axes becomes visible for the first time after its drawable area changes. This situation occurs when the drawable area changes while the axes is invisible, and then it becomes visible later.

These are some of the important characteristics of the `SizeChangedFcn` callback and some recommended best practices:

- Consider delaying the display of the figure until after all the variables that the callback uses are defined. This practice can prevent the `SizeChangedFcn` callback from returning an error. To delay the display of the figure, set its `Visible` property to `'off'`. Then, set the `Visible` property to `'on'` after you define the variables that your `SizeChangedFcn` callback uses.
- Use the `gcbo` function in your `SizeChangedFcn` code to get the `UIAxes` object that is resizing.

Example: `@myfun`

Example: `{@myfun,x}`

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you

do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`Children` — Children
empty `GraphicsPlaceholder` array | array of graphics objects

Children, returned as an array of graphics objects. Use this property to view a list of the children or to reorder the children by setting the property to a permutation of itself.

You cannot add or remove children using the `Children` property. To add a child to this list, set the `Parent` property of the child graphics object to the `UIAxes` object.

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Identifiers

Type — Type of graphics object
`'axes'`

This property is read-only.

Type of graphics object returned as `'axes'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object

elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

Compatibility Considerations

Changing ColorOrder or LineStyleOrder on the axes affects existing plots immediately

Behavior changed in R2019b

Starting in R2019b, if you change the axes `ColorOrder` or `LineStyleOrder` properties after plotting into the axes, the colors and line styles in your plot update immediately. In R2019a and previous releases, the new colors and line styles affect only subsequent plots, not the existing plots.

To preserve the original behavior, set the axes `ColorOrderIndex` or `LineStyleOrderIndex` property to any value (such as its current value) before changing the `ColorOrder` or `LineStyleOrder` property.

Indexing scheme for ColorOrder and LineStyleOrder might change plot colors and line styles

Behavior changed in R2019b

Starting in R2019b, there is a new indexing scheme that enables you to change the colors and line styles of existing plots by setting the `ColorOrder` or `LineStyleOrder` properties. MATLAB applies this indexing scheme to all objects that have a `ColorMode`, `FaceColorMode`, `MarkerFaceColorMode`, or `CDataMode`. As a result, your code might

produce plots that cycle though the colors and line styles differently than in previous releases.

In R2019a and earlier releases, MATLAB uses a different indexing scheme which does not allow you to change the colors of existing plots.

To preserve the way your plots cycle through colors and line styles, set the axes `ColorOrderIndex` or `LineStyleOrderIndex` property to any value (such as its current value) before plotting into the axes.

See Also

`uiaxes`

Topics

"Access Property Values"
"Graphics Object Properties"

Introduced in R2016a

uibutton

Create push button or state button component

Syntax

```
btn = uibutton
btn = uibutton(style)
btn = uibutton(parent)
btn = uibutton(parent,style)
btn = uibutton( ___ ,Name,Value)
```

Description

`btn = uibutton` creates a push button in a new figure and returns the `Button` object. MATLAB calls the `uifigure` function to create the figure.

`btn = uibutton(style)` creates a button of the specified style.

`btn = uibutton(parent)` creates the button in the specified parent container. The parent can be a `Figure` created using the `uifigure` function, or one of its child containers.

`btn = uibutton(parent,style)` creates a `Button` of the specified style in the specified parent container.

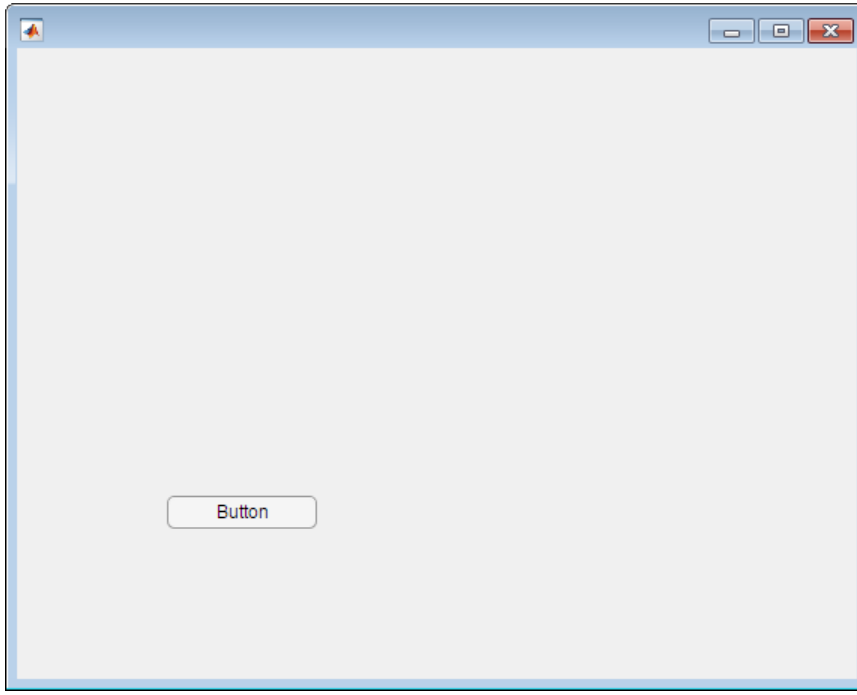
`btn = uibutton( ___ ,Name,Value)` creates a `Button` with properties specified by one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

Examples

Create a Button

Create a push button.

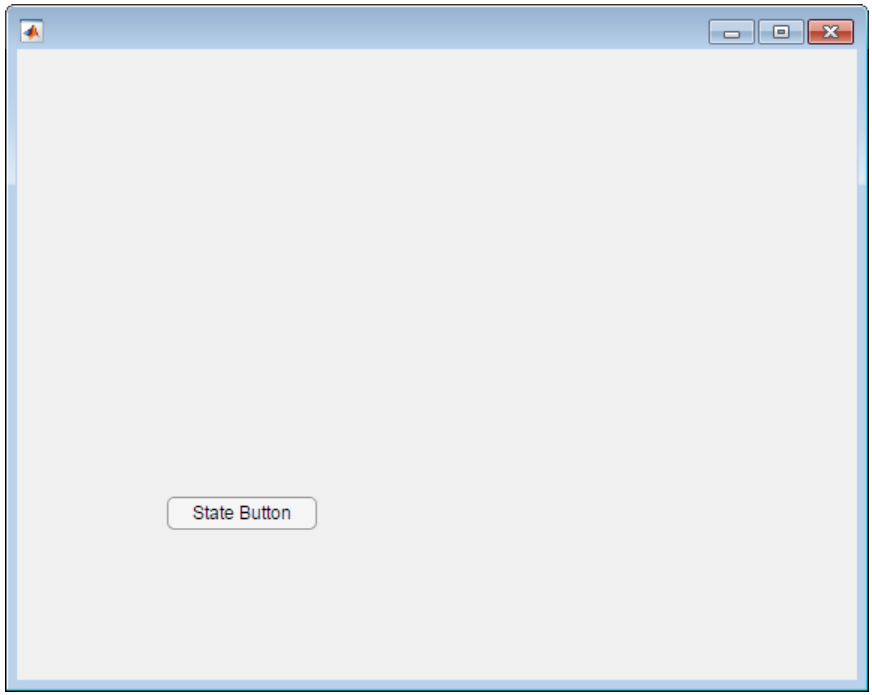
```
fig = uifigure;
btn = uibutton(fig);
```



Create a State Button

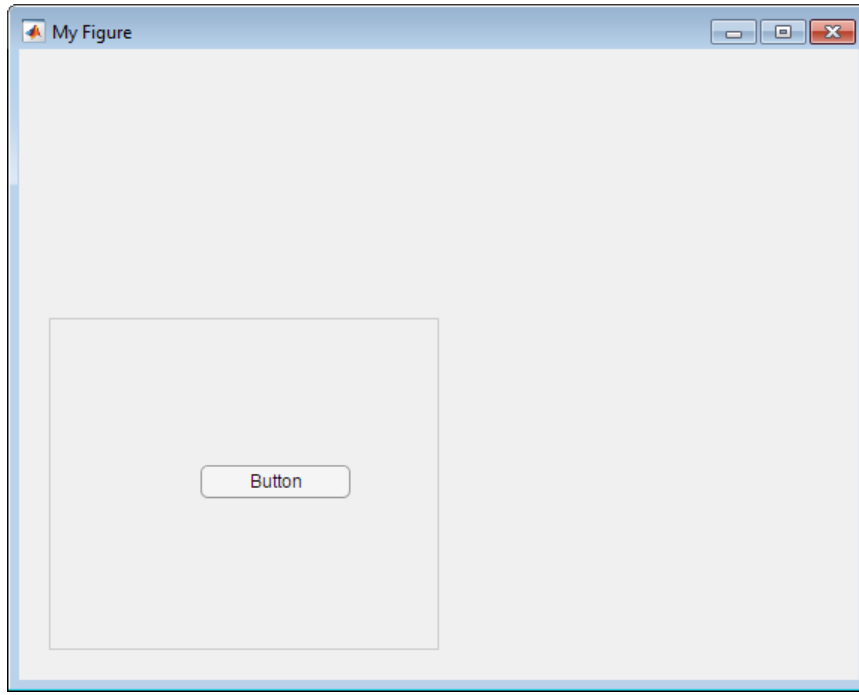
Create a state button by specifying the style as `'state'`.

```
fig = uifigure;
btn = uibutton(fig,'state');
```



Create a Button in a Panel

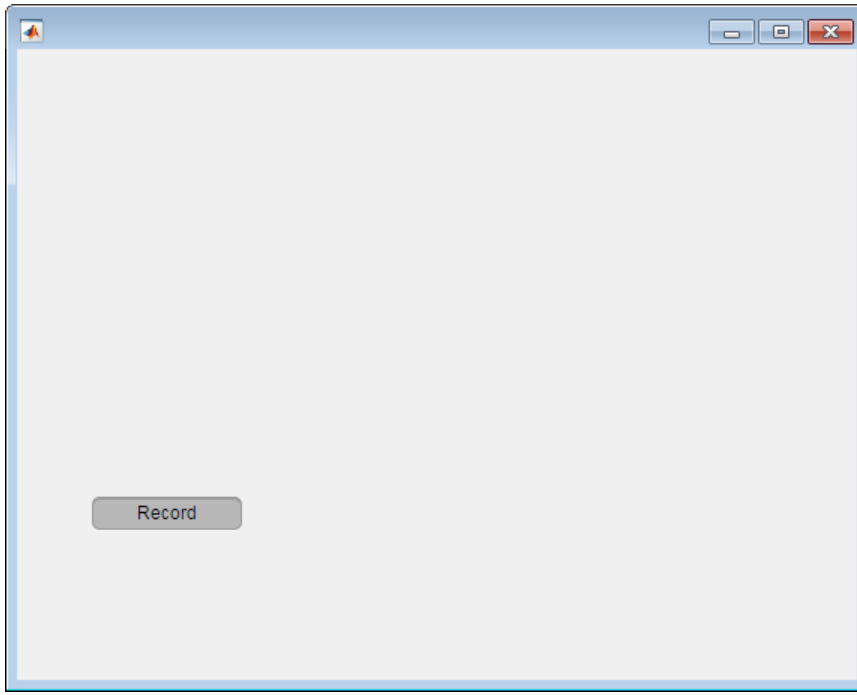
```
fig = uifigure('Name','My Figure');
pnl = uipanel(fig);
btn = uibutton(pnl);
```



Set and Access Button Property Values

Create a state button and specify property values.

```
fig = uifigure;
btn = uibutton(fig,'state',...
               'Text', 'Record',...
               'Value', true,...
               'Position',[50,100, 100, 22]);
```



Determine the font name of the state button text.

```
fname = btn.FontName
```

```
fname =

Helvetica
```

Change the font name of the button text.

```
btn.FontName = 'Arial Narrow';
```

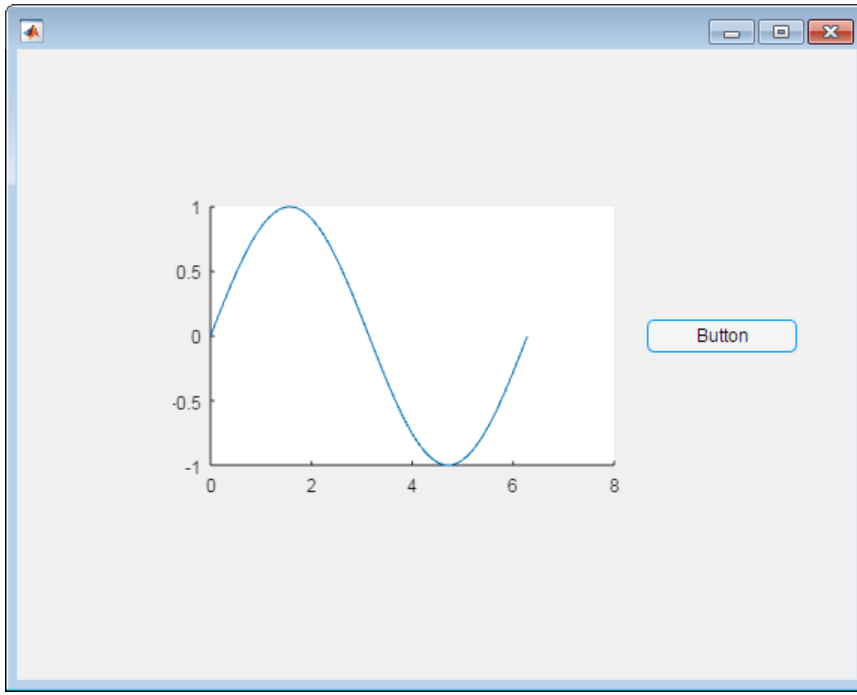
Code Response to Button Click

Create a button and a UI axes. When the app user presses the button, a graph is created.

Create `buttonPlot.m` on your MATLAB path. This code creates a window containing a button and a UI axes. When the app user clicks the button, the `ButtonPushedFcn` plots some data.

```
function buttonPlot
% Create a figure window
fig = uifigure;

% Create a UI axes
ax = uiaxes('Parent',fig,...
            'Units','pixels',...
            'Position', [104, 123, 300, 201]);

% Create a push button
btn = uibutton(fig,'push',...
               'Position',[420, 218, 100, 22],...
               'ButtonPushedFcn', @(btn,event) plotButtonPushed(btn,ax));
end

% Create the function for the ButtonPushedFcn callback
function plotButtonPushed(btn,ax)
        x = linspace(0,2*pi,100);
        y = sin(x);
        plot(ax,x,y)
end
```

Run `buttonPlot`, and then click the push button. MATLAB plots the data.



Input Arguments

`style` — Style of button
`'push'` (default) | `'state'`

Style of button, specified as one of the following:

- `'push'`

  When clicked once, the button appears to press and release.
- `'state'`

  When clicked once, the button remains in the pressed or released state until it is clicked again.

`parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If you do not specify a parent container, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Each type of `Button` object supports a different set of properties. For a full list of properties and descriptions for each type, see the associated property page.

- If `style` is the default, `'push'`, see Button
- If `style` is `'state'`, see StateButton

See Also

Functions
`appdesigner` | `uifigure`

Properties
Button | StateButton

Introduced in R2016a

uicheckbox

Create check box component

Syntax

```
cbx = uicheckbox
cbx = uicheckbox(parent)
cbx = uicheckbox( ___ ,Name,Value)
```

Description

`cbx = uicheckbox` creates a check box in a new figure window and returns the `CheckBox` object. MATLAB calls the `uifigure` function to create the figure.

`cbx = uicheckbox(parent)` creates the check box in the specified parent container. The parent can be a `Figure` created using the `uifigure` function, or one of its child containers.

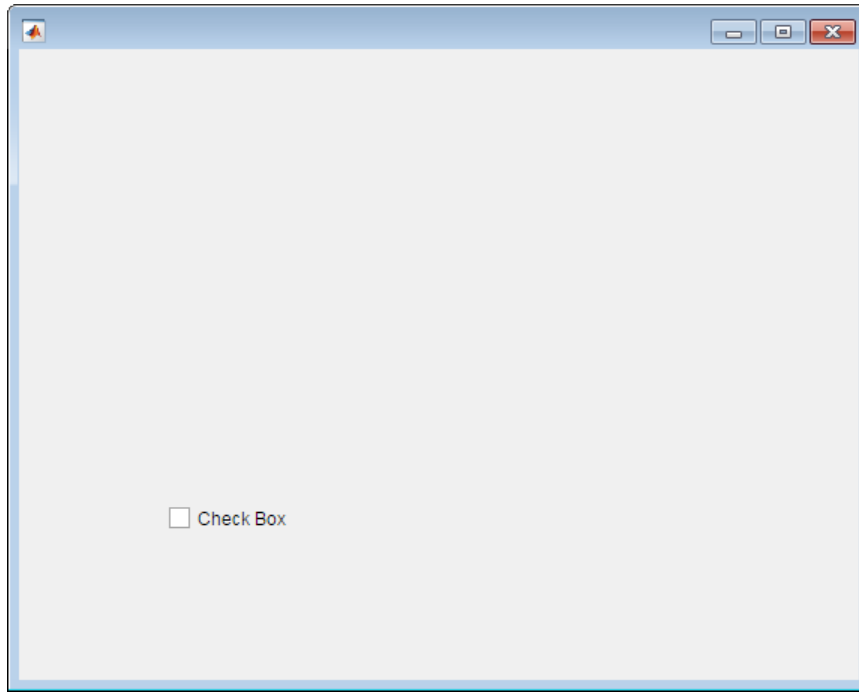
`cbx = uicheckbox( ___ ,Name,Value)` specifies `CheckBox` properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

Examples

Create Check Box

Create a check box.

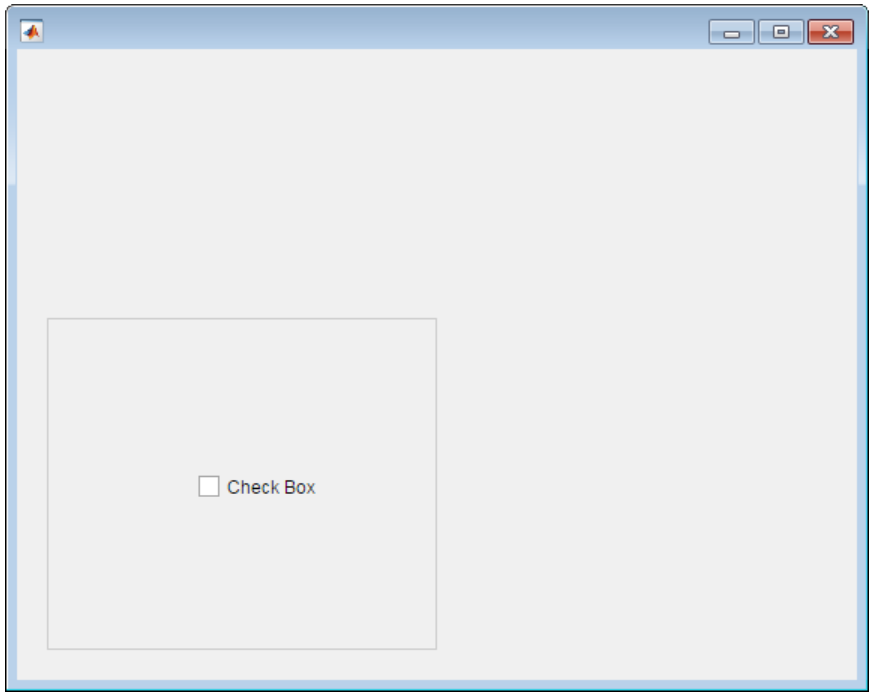
```
fig = uifigure;
cbx = uicheckbox(fig);
```



Create Check Box Within a Panel

Create a panel. Then create a check box within the panel.

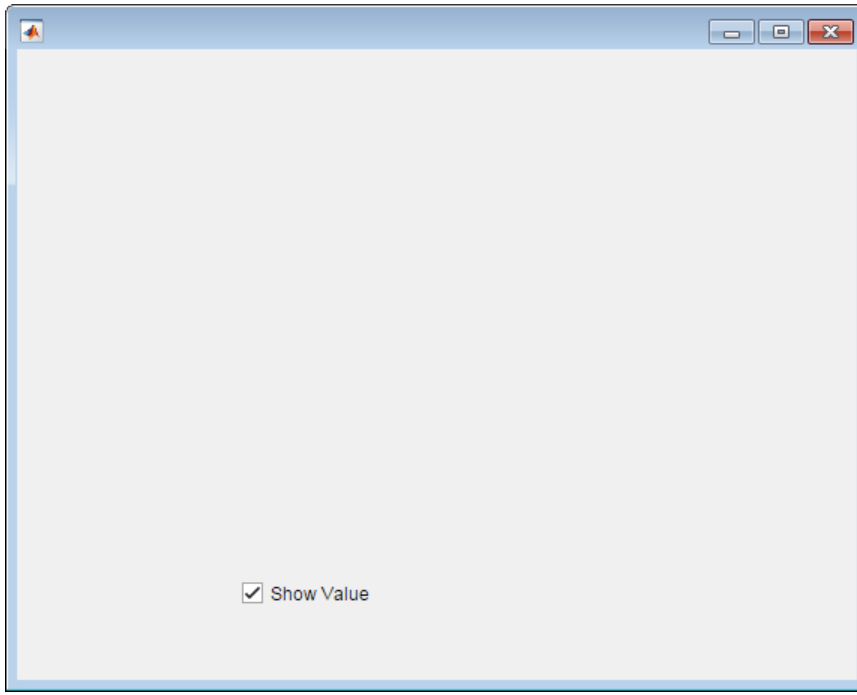
```
fig = uifigure;
pnl = uipanel(fig);
cbx = uicheckbox(pnl);
```



Set and Access Check Box Property Values

Create a check box and specify property values.

```
fig = uifigure;
cbx = uicheckbox(fig, 'Text','Show Value',...
                  'Value', 1,...
                  'Position',[150 50 102 15]);
```



Clear the check box.

```
cbx.Value = 0;
```

Determine the font size of the check box text.

```
fsize = cbx.FontSize
```

```
fsize =

    12
```

Code Response to Check Box Selection

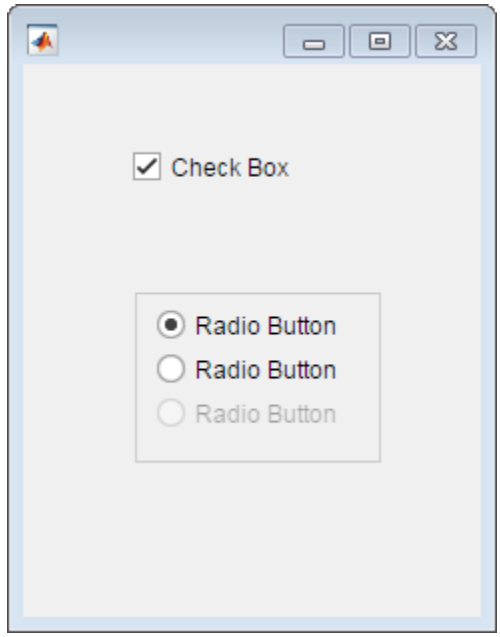
Create a radio button group and a check box. When an app user selects the check box, a radio button is disabled.

Save the following code to `disableRadioButton.m` on your MATLAB path.

This code creates a window containing a radio button group and a check box. When an app user clears the check box, the check box `ValueChangedFcn` disables the third radio button.

```
function disableRadioButton
% Create a figure window:
fig = uifigure('Position',[100 100 229 276]);

% Create a button group and radio buttons:
bg = uibuttongroup('Parent',fig,...
    'Position',[56 77 123 85]);
rb1 = uiradiobutton(bg,'Position',[10 60 91 15]);
rb2 = uiradiobutton(bg,'Position',[10 38 91 15]);
rb3 = uiradiobutton(bg,'Position',[10 16 91 15]);

% Create a check box:
cbx = uicheckbox(fig,'Position',[55 217 102 15],...
    'ValueChangedFcn',@(cbx,event) cBoxChanged(cbx,rb3));
end

% Create the function for the ValueChangedFcn callback:
function cBoxChanged(cbx,rb3)
val = cbx.Value;
if val
    rb3.Enable = 'off';
else
    rb3.Enable = 'on';
end
end
```

Run `disableRadioButton`, and then select the check box. The third radio button is disabled.



Input Arguments

`parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If you do not specify a parent container, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The properties listed here are a subset of the available properties. For the full list, see CheckBox.

Example: `'Value',1` specifies that the check box is displayed with a check mark.

`Value` — State of check box
0 (default) | 1

State of the check box, specified as `0` (false) or `1` (true). When the `Value` property is set to `1`, the check box is checked. When the `Value` property is set to `0`, the check box is not checked.

`ValueChangedFcn` — Value changed callback
`[]` (default) | function handle | cell array | character vector

Value changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when the user selects or clears the check box in the app. The callback does not execute if the check box value changes programmatically.

This callback function can access specific information about the user's interaction with the check box. MATLAB passes this information in a `ValueChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.PreviousValue` returns the previous value of the check box. The `ValueChangedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangedData` object.

Property	Value
`Value`	Value of check box after most recent app user interaction with it.

Property	Value
`PreviousValue`	Value of check box before most recent app user interaction with it.
`Source`	Component that executes the callback.
`EventName`	`'ValueChanged'`

For more information about writing callbacks, see “Write Callbacks in App Designer”.

`Position` — Location and size of check box
`[100 100 84 22]` (default) | `[left bottom width height]`

Location and size of the check box relative to the parent, specified as the vector `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the check box
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the check box
`width`	Distance between the right and left outer edges of the check box
`height`	Distance between the top and bottom outer edges of the check box

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

All measurements are in pixel units.

Example: `[200 200 102 15]`

See Also

Functions
`appdesigner` | `uifigure`

Properties
CheckBox

Introduced in R2016a

uidropdown

Create drop-down component

Syntax

```
dd = uidropdown
dd = uidropdown(parent)
dd = uidropdown( ___ ,Name,Value)
```

Description

`dd = uidropdown` creates a drop-down in a new figure window and returns the `DropDown` object. MATLAB calls the `uifigure` function to create the figure.

`dd = uidropdown(parent)` creates the drop-down in the specified parent container. The parent can be a `Figure` created using the `uifigure` function, or one of its child containers.

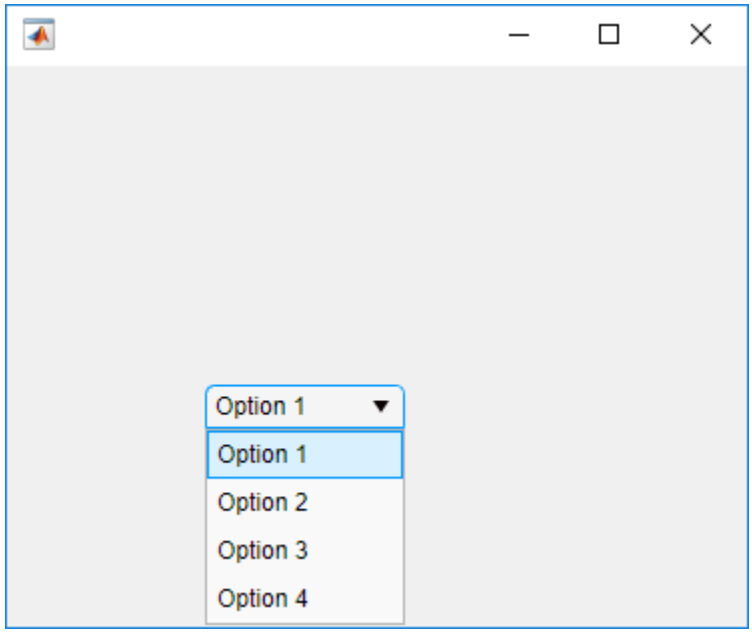
`dd = uidropdown( ___ ,Name,Value)` specifies object properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes. Use the `Name,Value` pair, `Editable,'on'` to specify a drop-down component that allows the app user to type text into the drop-down component or select a predefined option.

Examples

Create Drop-Down Component

Create a drop-down component with the default items.

```
fig = uifigure;
dd = uidropdown(fig);
```

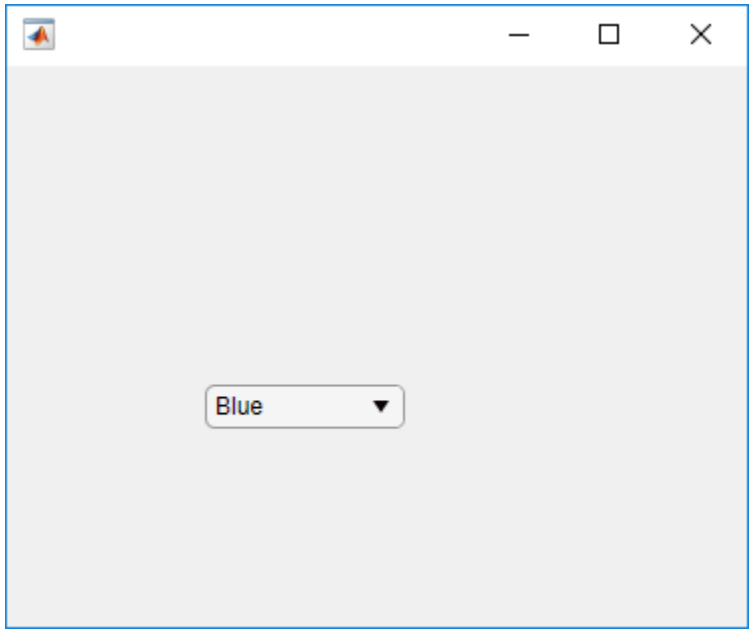


Clicking anywhere in the drop-down component causes it to open.

Set and Access Drop Down Properties

Create a drop-down component and specify the options.

```
fig = uifigure;
dd = uidropdown(fig,'Items',{'Red','Yellow','Blue','Green'},...
                    'Value','Blue');
```



Determine the value associated with the selected option.

```
value = dd.Value
```

```
value =

    'Blue'
```

By default, the `ItemsData` property is empty, so the drop-down component value corresponds to the element selected in the drop-down component.

Associate data values with each drop-down component item.

```
dd.ItemsData = [1 2 3 4];
```

Determine the value associated with the selected option.

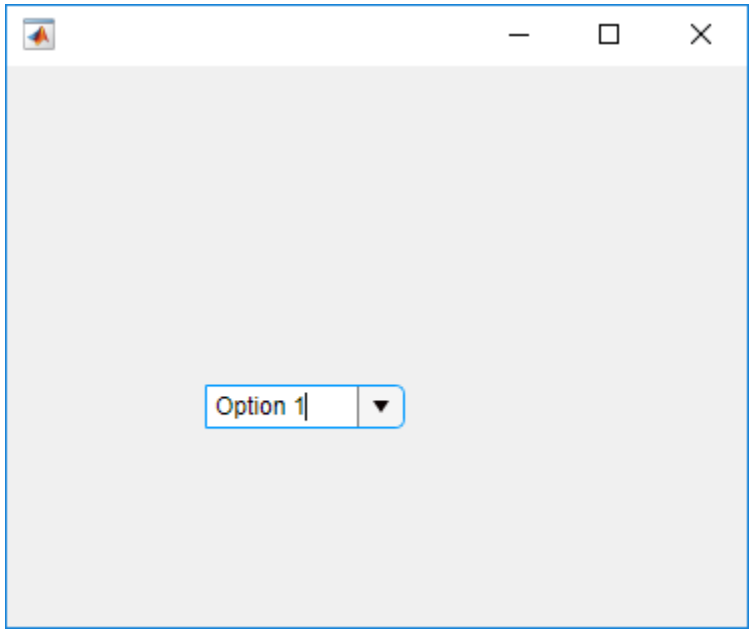
```
value = dd.Value
```

```
value =

3
```

Notice that when the `ItemsData` property value is not empty, the value of the drop-down component is the `ItemsData` value that corresponds to the selected `Items` value element.

Create an Editable Drop Down

```
fig = uifigure;
dd = uidropdown(fig,'Editable','on');
```



Clicking anywhere in the drop-down component, other than the down-arrow, inserts a caret, enabling the user to type text in the drop-down component.

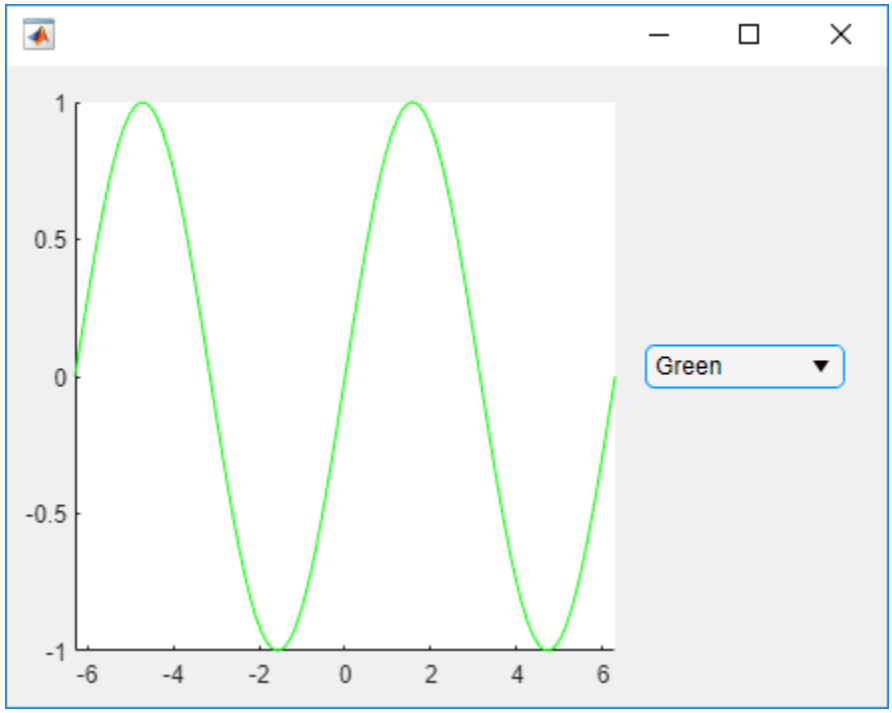
Code Response to Drop Down Selection

Create a plot and a drop-down component. When the app user makes a selection from the drop-down component, the plot changes color.

Save the following code to `plotOptions.m` on your MATLAB path. This code creates a window containing a plot and a drop-down component. When an app user changes the drop-down component selection, the `ValueChangedFcn` callback changes the plot color.

```
function plotOptions
fig = uifigure;
fig.Position(3:4) = [440 320];

ax = uiaxes('Parent',fig,...
    'Position',[10 10 300 300]);

x = linspace(-2*pi,2*pi);
y = sin(x);
p = plot(ax,x,y);
p.Color = 'Blue';

dd = uidropdown(fig,...
    'Position',[320 160 100 22],...
    'Items',{'Red','Yellow','Blue','Green'},...
    'Value','Blue',...
    'ValueChangedFcn',@(dd,event) selection(dd,p));
end

% Create ValueChangedFcn callback:
function selection(dd,p)
val = dd.Value;
p.Color = val;
end
```

Run `plotOptions`. Select **green** from the drop-down component to change the plot color to green.



Code Response to Drop-Down Component Entry or Selection

Create a drop-down component and a lamp. When the app user makes a selection from the drop-down component, the lamp size changes.

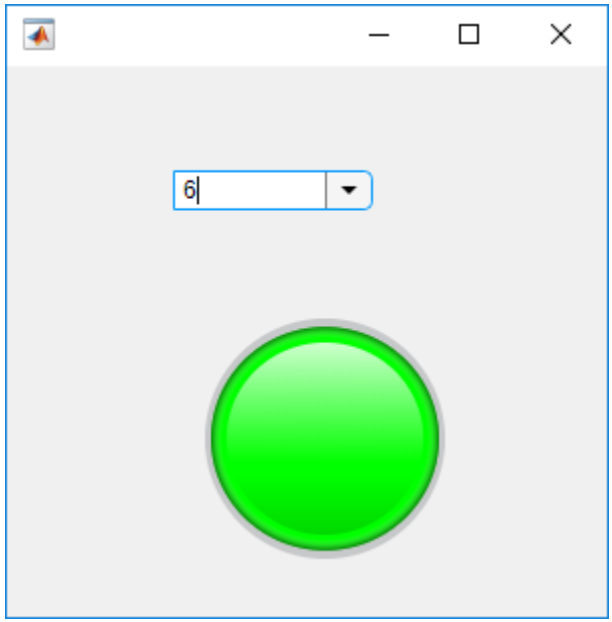
Save the following code to a `lampSize.m` on your MATLAB path. This code creates a figure window containing a drop-down component and a lamp. When an app user changes the drop-down component selection, the `ValueChangedFcn` callback changes the size of the lamp.

```
function lampSize
% Create figure and components

fig = uifigure('Position',[100 100 300 275]);

lmp = uilamp(fig,...
```

```
    'Position',[100 30 20 20]);

dd = uidropdown(fig,...
    'Editable','on',...
    'Position',[84 204 100 20],...
    'Items',{'Size x 1','Size x 2','Size x 3','Size x 4'},...
    'ItemsData',[1 2 3 4],...
    'Value',1,...
    'ValueChangedFcn',@(dd,event) optionSelected(dd,lmp));
end

% Create ValueChangedFcn callback
function optionSelected(dd,lmp)
val = dd.Value;
s = [20 20];
switch val
    case {1, 2, 3, 4}  % User selected a defined option
        size = val * s;
        lmp.Position(3:4) = size;
    otherwise % User typed a value
        m = str2num(val);
        size = m * s;
        lmp.Position(3:4) = size;
end
end
```

Run `lampSize` and select various options from the drop-down component.

Type a value in the drop-down component and press **Enter**. The lamp size changes. (If you type a large value, you might have to resize the figure to see the lamp.)



Input Arguments

parent — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If you do not specify a parent container, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The properties listed here are a subset of the available properties. For the full list, see .DropDown.

Example: `'Items',{'Red','Yellow','Blue'}` specifies the options presented in the drop-down component.

`Value` — Value
element of `Items` | element of `ItemsData`

Value, specified as an element of the `Items` or `ItemsData` arrays. By default, `Value` is the first element in `Items`.

Specifying `Value` as an element of `Items` selects the drop-down item that matches that element. If `ItemsData` is not empty, then `Value` must be set to an element of `ItemsData`, and the drop-down will select the associated item in the list.

`Items` — Drop-down items
`{'Option 1','Option 2','Option 3','Option 4'}` (default) | cell array of character vectors | string array | ...

Drop-down items, specified as a cell array of character vectors, string array, or 1-D categorical array. Duplicate elements are allowed. The drop-down component displays as many options as there are elements in the `Items` array. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Example: `{'Red','Yellow','Blue'}`

Example: `{'1','2','3'}`

`ItemsData` — Data associated with each element of the `Items` property value
empty array (`[]`) (default) | 1-by-n numeric array | 1-by-n cell array

Data associated with each element of the `Items` property value, specified as a 1-by-n numeric array or a 1-by-n cell array. Duplicate elements are allowed.

For example, if you set the `Items` value to employee names, you might set the `ItemsData` value to corresponding employee ID numbers. The `ItemsData` value is not visible to the app user.

If the number of array elements in the `ItemsData` value and the `Items` value do not match, one of the following occurs:

- When the `ItemsData` value is empty, then all the elements of the `Items` value are presented to the app user.

- When the `ItemsData` value has more elements than the `Items` value, then all the elements of the `Items` value are presented to the app user. MATLAB ignores the extra `ItemsData` elements.
- When the `ItemsData` value is not empty, but has fewer elements than the `Items` value, the only elements of the `Items` value presented to the app user are those that have a corresponding element in the `ItemsData` value.

Example: `{'One','Two','Three'}`

Example: `[10 20 30 40]`

`Editable` — Editable state of drop-down component
`'off'` (default) | `'on'`

Editable state of the drop-down component, specified as `'off'` or `'on'`.

If the `Enable` property value is `'off'`, then the app user cannot change the drop-down component text, even if the Editable property value is `'on'`.

`ValueChangedFcn` — Value changed callback
`''` (default) | function handle | cell array | character vector

Value changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback function executes when the user selects a different option from the drop-down list. It does not execute if the `Value` property changes programmatically.

This callback function can access specific information about the user's interaction with the drop-down. MATLAB passes this information in a `ValueChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.PreviousValue` returns the previous value of the drop-down. The `ValueChangedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangedData` object.

Property	Value
`Value`	Drop-down component value after app user's most recent interaction with it.
`PreviousValue`	Drop-down component value before app user's most recent interaction with it.
`Edited`	Logical value (0 or 1) that indicates whether the callback was executed as a result of typing a value into the drop-down component. The `Edited` value is 1 when the app user typed in the drop-down component and 0 when the app user selected an option from the drop-down component.
`Source`	Component that executes the callback.
`EventName`	`'ValueChanged'`

For more information about writing callbacks, see "Write Callbacks in App Designer".

`Position` — Location and size of drop-down component
`[100 100 100 22]` (default) | `[left bottom width height]`

Location and size of the drop-down component relative to the parent, specified as the vector `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the drop-down component
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the drop-down component
`width`	Distance between the right and left outer edges of the drop-down component
`height`	Distance between the top and bottom outer edges of the drop-down component

All measurements are in pixel units.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[100 100 100 22]`

See Also

Functions
`appdesigner` | `uifigure`

Properties
DropDown

Introduced in R2016a

uieditfield

Create text or numeric edit field component

Syntax

```
edt = uieditfield
edt = uieditfield(style)
edt = uieditfield(parent)
edt = uieditfield(parent,style)
edt = uieditfield( ___ ,Name,Value)
```

Description

`edt = uieditfield` creates a text edit field in a new figure window and returns the `EditField` object. MATLAB calls the `uifigure` function to create the figure.

`edt = uieditfield(style)` creates an edit field of the specified style.

`edt = uieditfield(parent)` creates the edit field in the specified parent container. The parent can be a `Figure` created using the `uifigure` function, or one of its child containers.

`edt = uieditfield(parent,style)` creates an edit field of the specified style in the specified parent container.

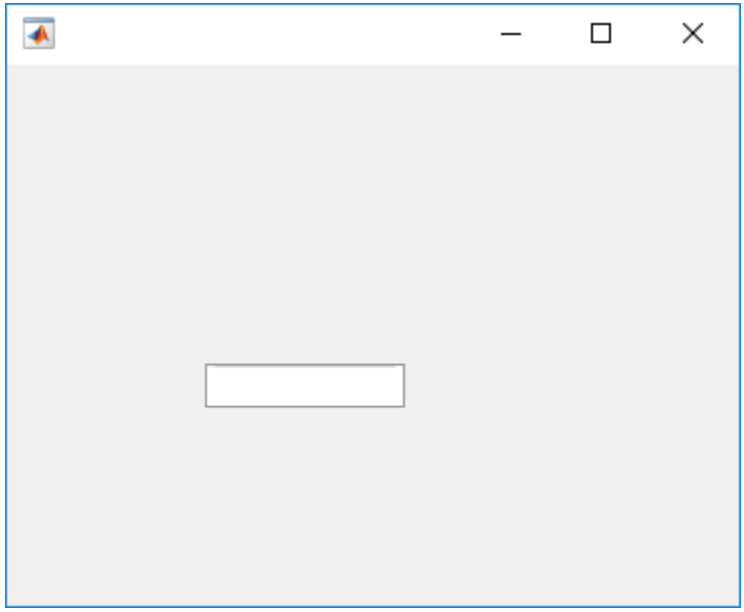
`edt = uieditfield( ___ ,Name,Value)` specifies object properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

Examples

Create Text Edit Field

Create a text edit field in a window.

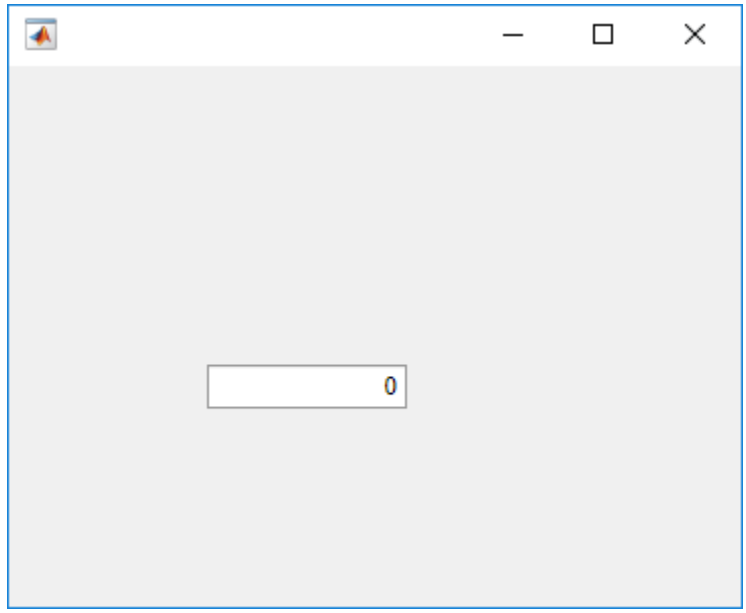
```
fig = uifigure;
edt = uieditfield(fig);
```



Create Numeric Edit Field

Create a numeric edit field by specifying the style as numeric.

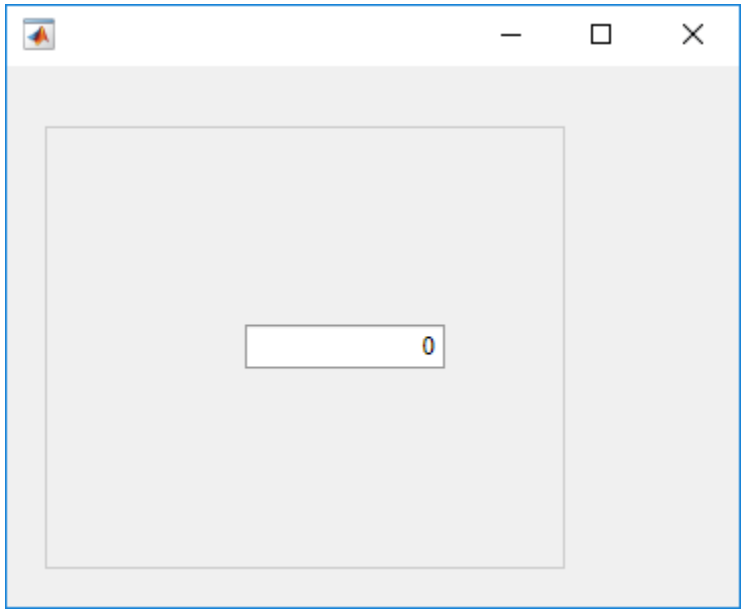
```
fig = uifigure;
edt = uieditfield(fig,'numeric');
```



Create Numeric Edit Field Within Panel

Specify a `Panel` as the parent object.

```
fig = uifigure;
pnl = uipanel(fig);
edt = uieditfield(pnl,'numeric');
```



Set and Access Numeric Edit Field Property Values

Create a numeric edit field with rounding on.

```
fig = uifigure;
edt = uieditfield(fig,'numeric',...
                    'RoundFractionalValues','on');
```

Determine the default limits.

```
limits = edt.Limits
```

```
limits =

  -Inf   Inf
```

The returned values indicate that there are no limits.

Change the limits to 0 through 100. (By default limits are inclusive.)

```
edt.Limits = [0 100];
```

Create Numeric Edit Field Specifying Limit Inclusiveness

Create a numeric edit field that allows the app user to enter a value greater than -5 and less than or equal to 10.

```
fig = uifigure;
edt = uieditfield(fig,'numeric',...
                      'Limits', [-5 10],...
                      'LowerLimitInclusive','off',...
                      'UpperLimitInclusive','on',...
                      'Value', 5);
```

If you enter a value in the numeric edit field that is outside the limits, MATLAB displays a message. The message indicates the problem and restores the value to the previous valid value.

Create Numeric Edit Field and Specify Display Format

Create a numeric edit field that allows the app user to enter any value, but always displays the value using exactly two decimal places and the specified units. MATLAB stores the exact value that the app user enters.

```
fig = uifigure;
edt = uieditfield(fig,'numeric',...
                      'ValueDisplayFormat', '%.2f Volts');
```

Type `5.5556` in the numeric edit field, and then click outside it. The edit field displays `5.56 Volts`.

MATLAB stores the value as `5.5556`. If you click in the edit field again, it displays `5.5556`. For a complete list of supported format display operators, see `sprintf`.

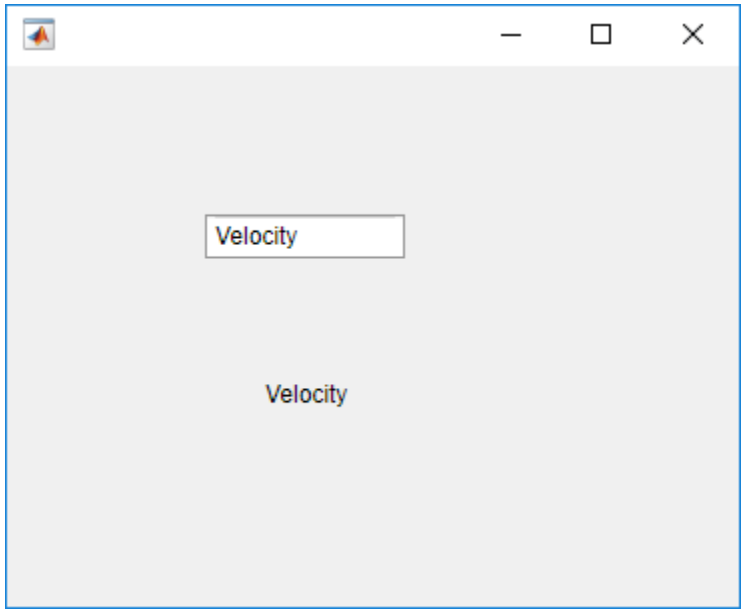
Code Response to Changed Edit Field Text

Code the `ValueChangedFcn` callback so that when the app user changes text in the edit field, a label is updated to match that text.

Save the following code to `textValue.m` on your MATLAB path.

```
function textValue
% Create figure and components.

fig = uifigure('Position',[100 100 366 270]);

lbl = uilabel(fig,...
      'Position',[130 100 100 15]);

txt = uieditfield(fig,...
      'Position',[100 175 100 22],...
      'ValueChangedFcn',@(txt,event) textChanged(txt,lbl));
end

% Code the callback function.
function textChanged(txt,lbl)
lbl.Text = txt.Value;
end
```

Run `textValue`, and type `Velocity` in the edit field. Click outside the edit field to trigger the callback.



Code Response to Changed Numeric Edit Field Value

Code the `ValueChangedFcn` callback such that when the app user changes the value in the edit field, a slider is updated to match that value.

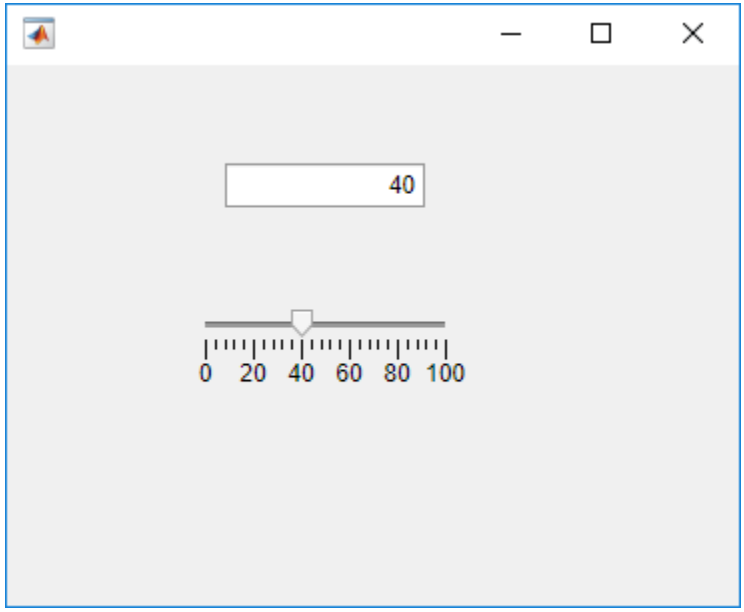
Save the following code to `numericEditFieldValue.m` on your MATLAB path.

```
function numericEditFieldValue
% Create figure and components

fig = uifigure('Position',[100 100 366 270]);

slider = uislider(fig,...
    'Position',[100 140 120 3]);

numfld = uieditfield(fig,'numeric',...
    'Position',[110 200 100 22],...
    'ValueChangedFcn',@(numfld,event) numberChanged(numfld,slider));

end
```

```
% Create ValueChangedFcn callback
function numberChanged(numfld,slider)
slider.Value = numfld.Value;
end
```

Run `numericEditFieldValue`.

Enter a value from 0 to 100 in the numeric edit field and click outside the field. The slider moves to indicate the numeric edit field value.



Use Event Data to Maintain a Log

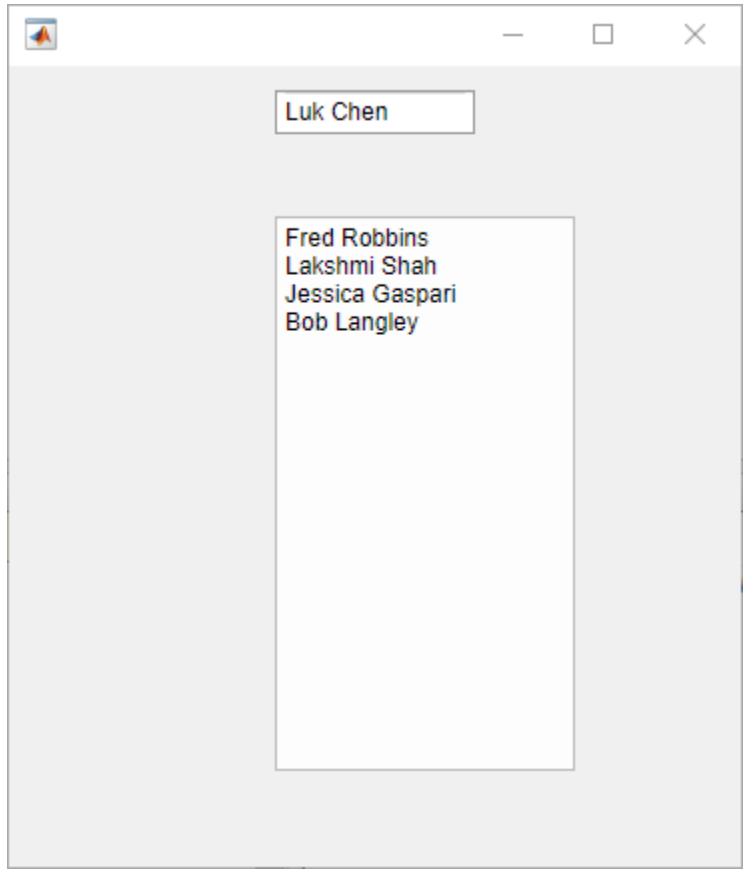
Code the `ValueChangedFcn` callback to maintain a log of values entered in a single session. When the app user changes the value in the edit field, the previous field value is added to a list maintained in a text area. The callback uses the `PreviousValue` property returned in the `event` argument to populate the text area.

Save the following code to `logNames.m` on your MATLAB path.

```
function logNames
% Create figure and components

fig = uifigure('Position',[100 100 366 400]);

loglist = uitextarea(fig,...
    'Position',[134 49 150 277],...
    'Editable','off');

namefld = uieditfield(fig,'text',...
  'Value', 'Bob Langley',...
  'Position',[134 367 100 22],...
  'ValueChangedFcn',@(namefld,event) nValChanged(namefld,event,loglist));
end

% Create ValueChangedFcn callback
function nValChanged(namefld,event,loglist)
newvalue = event.Value;
previousValue = event.PreviousValue;

loglist.Value = [previousValue; loglist.Value];

end
```

Run `logNames`.

Each time you enter a name in the text edit field and press enter, the name that was previously in the text edit field is added to the text area.



Input Arguments

`style` — Type of edit field
`'text'` (default) | `'numeric'`

Type of edit field, specified as one of the following:

- `'text'`

  By default, text edit fields are empty.

- `'numeric'`

  By default, numeric edit fields display the value 0. If the app user types a nonnumeric value in a numeric edit field, MATLAB opens an error tooltip and reverts the value to the last valid value.

`parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If you do not specify a parent container, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`EditField` and `NumericEditField` objects supports different sets of properties. For a full list of properties and descriptions for object, see the associated property page.

- EditField — If `style` is the default,`'text'`
- NumericEditField — If `style` is `'numeric'`

See Also

Functions
`appdesigner` | `uifigure`

Properties
EditField | NumericEditField

Introduced in R2016a

uifigure

Create figure for designing apps in App Designer

Syntax

```
f = uifigure
f = uifigure(Name,Value)
```

Description

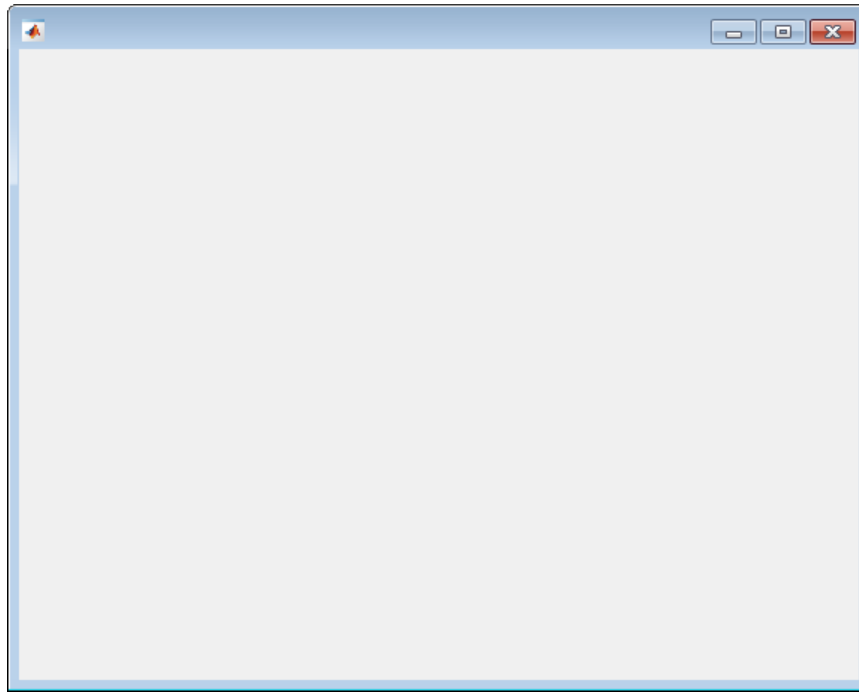
`f = uifigure` creates a figure for building a user interface and returns the `Figure` object. This is the type of figure that App Designer uses.

`f = uifigure(Name,Value)` specifies figure properties using one or more `Name,Value` pair arguments.

Examples

Create Default UI Figure

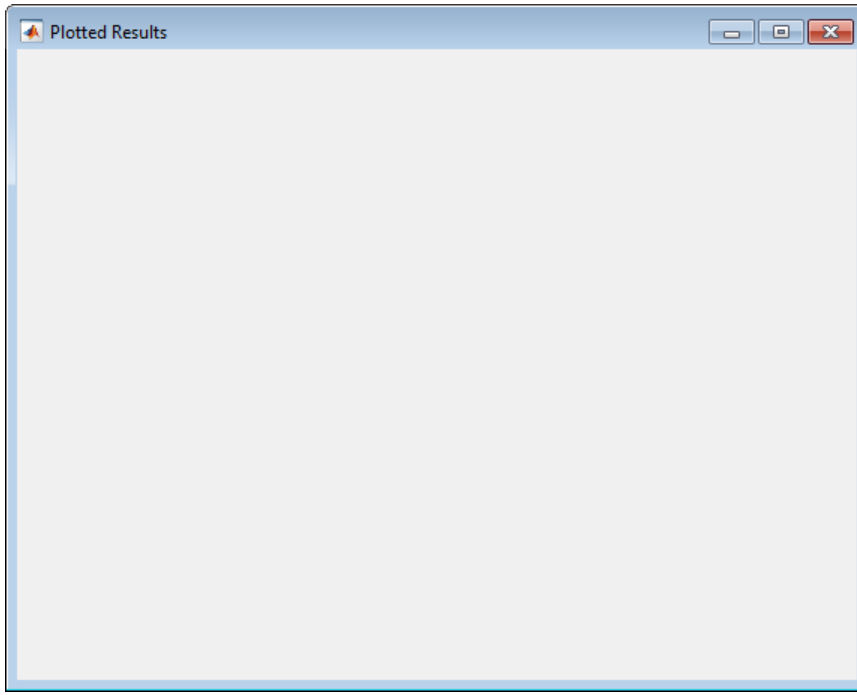
```
f = uifigure;
```



Set and Access Properties

Create a UI figure with a specific title.

```
f= uifigure('Name','Plotted Results');
```



Get the `Position` property value.

```
p = f.Position
```

```
ans =

   680   678   560   420
```

Code CloseRequestFcn to Confirm Closing UI Figure

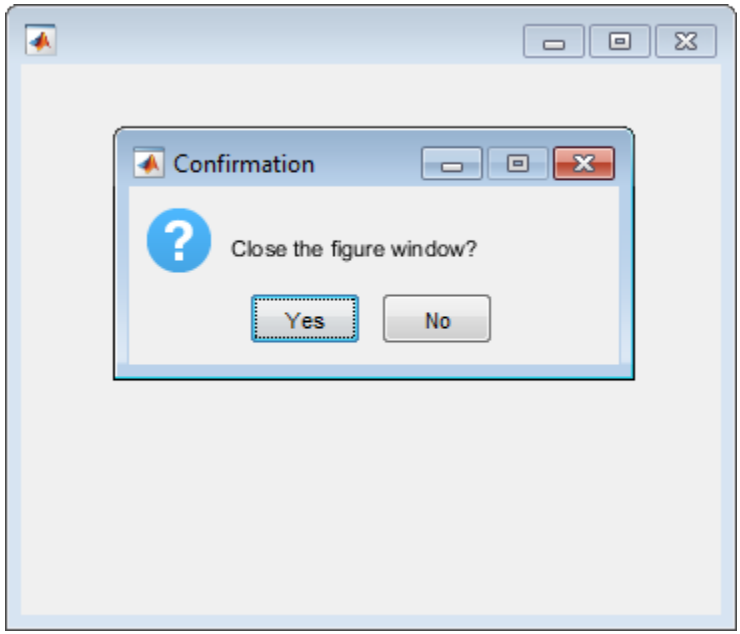
Code the `CloseRequestFcn` callback to open a modal Confirmation dialog box when the user tries to close the window.

Copy and paste this code into the MATLAB Editor, and then run `closeFig`.

```
function closeFig
f = uifigure('Position',[100 100 350 275],...
    'CloseRequestFcn',@(f, event)my_closereq(f));
```

```
end
function my_closereq(f)
selection = questdlg('Close the figure window?',...
    'Confirmation',...
    'Yes','No','Yes');
switch selection,
    case 'Yes',
        delete(f)
    case 'No'
        return
end
end
```

Click the figure close button. The Confirmation dialog box opens.



Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The properties listed here are a subset of the available properties. For the full list, see UI Figure.

Example: `'Name','My App'` specifies `My App` as the title of the UI figure.

Name — Name
`''` (default) | character vector | string scalar

Name of the figure, specified as a character vector or a string scalar.

Example: `uifigure('Name','Results')` sets the name of the figure to `'Results'`.

`Position` — Location and size of UI figure, excluding borders and title bar
`[left bottom width height]`

Location and size of the UI figure, excluding borders and title bar, specified as a four-element vector of the form `[left bottom width height]`.

This table describes each element in the vector.

Element	Description
`left`	Distance from the left edge of the primary display to the inner left edge of the UI figure window. This value can be negative on systems that have more than one monitor.
`bottom`	Distance from the bottom edge of the primary display to the inner bottom edge of the UI figure window. This value can be negative on systems that have more than one monitor.
`width`	Distance between the right and left inner edges of the UI figure.
`height`	Distance between the top and bottom inner edges of the UI figure.

Limitations

- Currently, you cannot pass a `Figure` object created with the `uifigure` function to the `print`, `rotate3d`, `pan`, or `zoom` functions. If you attempt to do so, MATLAB throws an error. To use the `pan`, `zoom`, and `rotate3d` interactions for plots you create in `uifigure`, specify the axes as the first argument, and specify `'on'`, `'off'`, or a scale factor (for `zoom`) as the second argument. For more information, see “Displaying Graphics in App Designer”.

See Also

Functions
`appdesigner` | `scroll` | `uibuttongroup` | `uipanel` | `uitab`

Properties
UI Figure

Introduced in R2016a

UI Figure Properties

Control UI figure appearance and behavior in App Designer

Description

UI figures are containers for creating apps in App Designer. Properties control the appearance and behavior of the UI figure. Use dot notation to refer to a particular object and property:

```
uf = uifigure;
name = uf.Name;
uf.Name = 'My App';
```

Properties

Window Appearance

`Color` — Background color
RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Background color, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (`#`) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

`WindowState` — Window state

`'normal'` (default) | `'minimized'` | `'maximized'` | `'fullscreen'`

Window state, specified as one of these values:

- `'normal'` — The window displays in a normal state.
- `'minimized'` — The window is collapsed, but you can still execute commands to get or set its properties, to add children, or to create plots in the window.

- `'maximized'` — The window fills the screen. The minimize, restore, and close buttons provided by the operating system appear in the corner of the window. If the window has menus in the normal state, they are present in this state.
- `'fullscreen'` — The window fills the screen. However, the minimize, restore, and close buttons provided by the operating system are hidden. If the window has menus in the normal state, they are present in this state.

Clicking the minimize, maximize, or restore button provided by the operating system sets the `WindowState` property accordingly. Pressing **Ctrl+F11** (Windows and Linux) or **Ctrl +Command+f** (Mac OS) toggles the `'fullscreen'` state. Setting this property on a docked figure or in MATLAB Online is not supported.

UI figures cannot be minimized directly from a full screen state.

Position

`Position` — Location and size of UI figure, excluding borders and title bar
`[left bottom width height]`

Location and size of the UI figure, excluding borders and title bar, specified as a four-element vector of the form `[left bottom width height]`.

This table describes each element in the vector.

Element	Description
`left`	Distance from the left edge of the primary display to the inner left edge of the UI figure window. This value can be negative on systems that have more than one monitor.
`bottom`	Distance from the bottom edge of the primary display to the inner bottom edge of the UI figure window. This value can be negative on systems that have more than one monitor.
`width`	Distance between the right and left inner edges of the UI figure.
`height`	Distance between the top and bottom inner edges of the UI figure.

`Units` — Units of measurement
`'pixels'` (default)

Units of measurement, specified as `'pixels'`.

`InnerPosition` — Location and size of UI figure, excluding borders and title bar
`[left bottom width height]`

Location and size of UI figure, excluding borders and title bar, specified as a four-element vector of the form `[left bottom width height]`. All measurements are in units specified by the `Units` property.

This property value is identical to the `Position` property value.

`Resize` — Resizable
`'on'` (default) | `'off'`

Resizable UI figure, specified as `'on'` or `'off'`. When this property is set to `'on'`, the UI figure is resizable. Otherwise, it is not resizable.

`AutoResizeChildren` — Automatically resize children
`'on'` (default) | `'off'`

Automatically resize children, specified as `'on'` or `'off'`.

- `'on'` — Child components automatically resize when the container resizes.
- `'off'` — Child components do not resize.

The `AutoResizeChildren` property affects direct children of the container, not children inside nested containers.

To customize the resize behavior, set the `AutoResizeChildren` property to `'off'` and create a `SizeChangedFcn` callback for the container. For more information, see "Managing Resizable Apps in App Designer".

To disable resizing of an app, set the `Resize` property of the figure to `'off'`.

Plotting

`Colormap` — Color map for `UIAxes` content
parula (default) | `m`-by-3 array of RGB triplets

Color map for `UIAxes` content, specified as an `m`-by-3 array of RGB (red, green, blue) triplets that define `m` individual colors.

Example: `uifigure('Colormap',[1 0 1; 0 0 1; 1 1 0])` sets the color map to three colors: magenta, blue, and yellow.

MATLAB accesses these colors by their row number.

`Alphamap` — Transparency map for axes content

vector of 64 values between `0` and `1` (default) | vector of values between `0` and `1`

Transparency map for axes content, specified as a vector of values between `0` and `1`, inclusive. The size of the vector can be m-by-1 or 1-by-m. MATLAB accesses alpha values by their index in the vector. Alphamaps can be any length.

Interactivity

`Visible` — Visibility of UI figure

`'on'` (default) | `'off'`

Visibility UI figure, specified as `'on'` or `'off'`. The `Visible` property determines whether the UI figure is displayed on the screen. If the `Visible` property is set to `'off'`, the entire UI figure is hidden. However, you can still specify and access properties.

To make your app start faster, set the `Visible` property of all components that are not displayed at startup to `'off'`.

Note Changing the `Visible` property does *not* change the value of the `Visible` property in a child component. This is true even though hiding the UI figure causes the children to be hidden.

`CurrentPoint` — Current point

two-element vector

Current point, returned as a two-element vector. The vector contains the (x, y) coordinates of the mouse pointer, measured in pixels from the lower-left corner of the UI figure. If the UI figure is scrollable, the coordinates are measured from the lower-left corner of the scrollable area. The scrollable area is the area that is bounded by the limits of the scroll bars.

The coordinates update when you do any of the following:

- Press the mouse button within the UI figure.
- Release the mouse button after pressing it within the UI figure.
- Press the mouse button within the UI figure, and then release it outside the UI figure.

- Rotate the scroll wheel within the UI figure.
- Move the mouse within the figure (without pressing any buttons), provided that the `WindowButtonMotionFcn` property is not empty.

If the UI figure has a callback that responds to mouse interactions, and you trigger that callback faster than the system can execute the code, the coordinates might not reflect the actual location of the pointer. Instead, they are the location when the callback began execution.

If you use the `CurrentPoint` property to plot points, the coordinate values might contain rounding error.

`CurrentCharacter` — Current character
`''` (default) | character

Current character, returned as the character of the last key pressed. This property updates when the UI figure has focus while the user presses a key.

`SelectionType` — Mouse selection type
`'normal'` (default) | `'extend'` | `'alt'` | `'open'`

Mouse selection type, returned as `'normal'`, `'extend'`, `'alt'`, or `'open'`. This property provides information about the last mouse button press that occurred in the UI figure.

This table lists the possible `SelectionType` values and the user actions that produce those values.

Value	Corresponding Action
`'normal'`	Click the left mouse button.
`'extend'`	Any of the following: • **Shift**-click the left mouse button. • Click the middle mouse button. • Click both left and right mouse buttons.
`'alt'`	Either of the following: • **Control**-click the left mouse button. • Click the right mouse button.

Value	Corresponding Action
`'open'`	Double-click any mouse button.

`Scrollable` — Ability to scroll

`'off'` (default) | `'on'`

Ability to scroll, specified as `'off'` or `'on'`. Setting this property to `'on'` enables scrolling within the container. However, there are additional requirements:

- The child components in the container must occupy a larger area than the container can display at one time.
- Components that do not fit in the container must be above or to the right of the container. You cannot scroll to components that are below or to the left of the container.

Certain types of charts and axes do not support scrollable containers. However, you can place the chart or axes in a nonscrollable panel, and then place the panel in the scrollable container. For more information, see “Displaying Graphics in App Designer”.

Common Callbacks

`ButtonDownFcn` — Button down callback

`''` (default) | function handle | cell array | character vector

Button down callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when the user clicks a blank area of the UI figure.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

Use the `SelectionType` property to determine whether the user pressed modifier keys.

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Keyboard Callbacks

`KeyPressFcn` — Key-press callback

`''` (default) | function handle | cell array | character vector

Key-press callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback function executes when the user presses a key while UI figure or a child component has focus. Pressing a key on a component that accepts keyboard focus (such as an edit field) does not execute the callback unless the component is disabled. You can disable a component by setting the `Enable` or `Editable` property to `'off'`.

If the user presses multiple keys at approximately the same time, MATLAB detects all the keys.

The `KeyPressFcn` callback can access specific information about the user's interaction with the keyboard. MATLAB passes this information in a `KeyData` object as the second argument to your callback function. This table lists the properties of the `KeyData` object.

Property	Description
`Character`	The character that appears as a result of pressing the key or keys. Pressing certain keys or modifying keys with the **Ctrl** key might return unprintable characters in this property. Pressing certain keys alone (such as **Ctrl**, **Alt**, **Shift**) does not generate `Character` data.
`Modifier`	Cell array containing the names of the modifier keys pressed, such as **control**, **alt**, or **shift**. If no modifier keys are pressed, the cell array is empty.

Property	Description
`Key`	Key pressed, identified by the lowercase label on the key, or a descriptive word such as `'space'`.
`Source`	Object that has focus when the key is pressed.
`EventName`	`'KeyPress'`

For more information about specifying callback property values and using callback arguments, see “Write Callbacks in App Designer”.

The `CurrentCharacter` property also returns character information.

`KeyReleaseFcn` — Key-release callback
`''` (default) | function handle | cell array | character vector

Key-release callback, specified as one of these values

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback function executes when the user releases a key while the UI figure or a child component has focus. Releasing a key on a component that accepts keyboard focus (such as an edit field) does not execute the callback unless the component is disabled. You can disable a component by setting the `Enable` or `Editable` property to `'off'`.

This callback function can access specific information about the user’s interaction with the keyboard. MATLAB passes this information in a `KeyData` object as the second argument to your callback function. This table lists the properties of the `KeyData` object.

Property	Description
`Character`	Character that corresponds to the key or keys that are released. Certain keys or key combinations might return unprintable characters in this property. Pressing and releasing certain keys alone (such as **Ctrl**, **Alt**, **Shift**) does not generate `Character` data.

Property	Description
`Modifier`	Cell array containing the names of the modifier keys released, such as **control**, **alt**, or **shift**. If no modifier keys are released, the cell array is empty.
`Key`	Key released, identified by the (lowercase) label on the key, or a descriptive word such as `'space'`.
`Source`	Object that has focus when the key is released.
`EventName`	`'KeyRelease'`

For more information about specifying callback property values and using callback arguments, see “Write Callbacks in App Designer”.

Window Callbacks

`CloseRequestFcn` — Close request callback

`'closereq'` (default) | function handle | cell array | character vector

Close request callback, specified as a function handle, cell array containing a function handle and additional arguments, or a character vector. MATLAB executes this callback when the app user attempts to close the app. For example, you can code the callback to display a dialog box that asks the app user to confirm or cancel the close operation.

Consider calling the `delete` function to close the `Figure` in the `CloseRequestFcn` callback:

- In App Designer, call `delete(app)`.
- For apps created programmatically, call `delete(f)`, where `f` is the `Figure` object.

If the `closeRequestFcn` callback does not contain a `delete` command, MATLAB never closes the window. You can always call the `delete` function to force the window to close.

`SizeChangedFcn` — Size change callback

`''` (default) | function handle | cell array | character vector

Size change callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

Define this callback to customize the app layout when the size of this container changes (e.g., when the user resizes the window).

Note The `SizeChangedFcn` callback will not execute unless the `AutoResizeChildren` property of this container is set to `'off'`. In App Designer, you can make the `SizeChangedFcn` executable by clearing the **AutoResizeChildren** check box in the **UI Figure Properties** panel.

The `SizeChangedFcn` callback executes when:

- This container becomes visible for the first time.
- This container is visible while its size changes.
- This container becomes visible for the first time after its size changes. This situation occurs when the size changes while the container is invisible, and then it becomes visible later.

Other important points to consider when defining a `SizeChangedFcn` callback:

- Consider delaying the display of this container until after all the variables that the `SizeChangedFcn` uses are defined. This practice can prevent the `SizeChangedFcn` callback from returning an error. To delay the display of the container, set its `Visible` property to `'off'`. Then, set the `Visible` property to `'on'` after you define the variables that your `SizeChangedFcn` callback uses.
- If your app contains nested containers, they resize from the inside out.
- To access the container that is resizing from within the `SizeChangedFcn`, refer to the source object (the first input argument in the callback) or use the `gcbo` function.

`WindowButtonDownFcn` — Window button down callback

`''` (default) | function handle | cell array | character vector

Window button down callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when the user clicks:

- A blank area of the UI figure
- A UI component within the UI figure that supports the `ButtonDownFcn` property
- A graphics object within the UI figure that supports the `ButtonDownFcn` property

For example, clicking an `axes` object triggers the `WindowButtonDownFcn` callback.

See the list of properties for a specific UI component or graphics object to determine if it supports the `ButtonDownFcn` property. See “Designing Apps in App Designer” for the properties of UI components. See “Graphics Object Properties” for the properties of graphics objects.

Note The callback executes when the user clicks a `uiaxes` component, even though `uiaxes` components do not support the `ButtonDownFcn` property.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

`WindowButtonMotionFcn` — Window button motion callback

`''` (default) | function handle | cell array | character vector

Window button motion callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

Typically, this callback function executes whenever the user moves the pointer within the UI figure. However, if the user clicks and holds the mouse button and moves the pointer outside the UI figure, the `WindowButtonMotionFcn` callback executes while the pointer is outside the UI figure.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

`WindowButtonUpFcn` — Window button up callback

`''` (default) | function handle | cell array | character vector

Window button up callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when the user:

- Releases the mouse button on a blank area of the UI figure
- Releases the mouse button on a UI component within the UI figure that supports the `ButtonDownFcn` property
- Releases the mouse button on a graphics object within the UI figure that supports the `ButtonDownFcn` property

For example, releasing the mouse button on an `axes` object triggers the `WindowButtonUpFcn` callback.

See the list of properties for a specific UI component or graphics object to determine if it supports the `ButtonDownFcn` property. See “Designing Apps in App Designer” for the properties of UI components. See “Graphics Object Properties” for the properties of graphics objects.

Button-up actions are associated with button-down actions in the UI figure. The mouse pointer must be inside the UI figure when the button-down action occurs, but it can be outside the figure when the button-up action occurs.

Note The callback executes when the user releases the mouse button on a `uiaxes` component, even though `uiaxes` components do not support the `ButtonDownFcn` property.

For more information about specifying a callback property value as a function handle, cell array, or character vector, see “Write Callbacks in App Designer”.

`WindowKeyPressFcn` — Window key-press callback

`''` (default) | function handle | cell array | character vector

Window key-press callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback function executes when the user presses a key while the UI figure or a child component has focus. If the user presses multiple keys at approximately the same time, MATLAB detects all the keys.

This callback function can access specific information about the user's interaction with the keyboard. MATLAB passes this information in a `KeyData` object as the second argument to your callback function. This table lists the properties of the `KeyData` object.

Property	Description
`Character`	The character that appears as a result of pressing the key or keys. Pressing certain keys or modifying keys with the **Ctrl** key might return unprintable characters in this property. Pressing certain keys alone (such as **Ctrl**, **Alt**, **Shift**) does not generate `Character` data.
`Modifier`	Cell array containing the names of the modifier keys pressed, such as **control**, **alt**, or **shift**. If no modifier keys are pressed, the cell array is empty.
`Key`	Key pressed, identified by the lowercase label on the key, or a descriptive word such as `'space'`.
`Source`	Object that has focus when the key is pressed.
`EventName`	`'KeyPress'`

For more information about specifying callback property values and using callback arguments, see "Write Callbacks in App Designer".

The `CurrentCharacter` property also returns character information.

`WindowKeyReleaseFcn` — Window key-release callback
`''` (default) | function handle | cell array | character vector

Window key-release callback, specified as one of these values

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback function executes when the user releases a key while the UI figure or a child component has focus.

This callback function can access specific information about the user's interaction with the keyboard. MATLAB passes this information in a `KeyData` object as the second argument to your callback function. This table lists the properties of the `KeyData` object.

Property	Description
`Character`	Character that corresponds to the key or keys that are released. Certain keys or key combinations might return unprintable characters in this property. Pressing and releasing certain keys alone (such as **Ctrl**, **Alt**, **Shift**) does not generate `Character` data.
`Modifier`	Cell array containing the names of the modifier keys released, such as **control**, **alt**, or **shift**. If no modifier keys are released, the cell array is empty.
`Key`	Key released, identified by the (lowercase) label on the key, or a descriptive word such as `'space'`.
`Source`	Object that has focus when the key is released.
`EventName`	`'KeyRelease'`

For more information about specifying callback property values and using callback arguments, see "Write Callbacks in App Designer".

`WindowScrollWheelFcn` — Window scroll wheel callback
`''` (default) | function handle | cell array | character vector

Window scroll wheel callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when the user moves the scroll wheel while the UI figure or any of its children have focus. However, other objects can capture scroll wheel movement and interfere with the execution of this callback.

This callback has no effect on the `CurrentObject` and `SelectionType` properties.

This callback can access specific information when the user rotates the scroll wheel. MATLAB passes this information in a `ScrollWheelData` object as the second argument to your callback function. This table lists the properties of the `ScrollWheelData` object.

Property	Contents
`VerticalScrollCount`	A positive or negative number that indicates the direction and number of scroll wheel clicks. Scrolling down returns a positive value. Scrolling up returns a negative value. The vertical scroll count is the sum of all scroll wheel clicks that occurred since the last time the callback executed. Typically, the value is `1` or `-1`. However, the scroll count can have a larger magnitude if the `WindowScrollWheelFcn` callback takes a long time to return, or if the user spins the scroll wheel very fast.

Property	Contents
VerticalScrollAmount	A constant value of 3.
Source	Object that executes the callback.

Contents Property
'WindowScrollWheel'. event Name

For more information about specifying callback property values and using callback arguments, see “Write Callbacks in App Designer”

Example: App for Changing x-Axis Limits

This example shows how to create an app that displays a plot. The user can scroll to change the limits of the *x*-axis. Copy and paste the following code into the editor and run it.

```
function scroll_wheel
% Shows how to use WindowScrollWheelFcn property
%
   f = uifigure('WindowScrollWheelFcn',@figScroll,'Name','Scroll Wheel Demo');
   x = 0:.1:40;
   y = 4.*cos(x)./(x+2);
   a = axes(f);
   h = plot(a,x,y);
   title(a,'Rotate the scroll wheel')

   function figScroll(~,event)
      if event.VerticalScrollCount > 0
         xd = h.XData;
         inc = xd(end)/20;
         x = [0:.1:xd(end)+inc];
         re_eval(x)
      elseif event.VerticalScrollCount < 0
         xd = h.XData;
         inc = xd(end)/20;
```

```
            % Don't let xd = 0
            x = [0:.1:xd(end)-inc+.1];
            re_eval(x)
        end
    end

    function re_eval(x)
        y = 4.*cos(x)./(x+2);
        h.YData = y;
        h.XData = x;
        a.XLim = [0 x(end)];
        drawnow
    end
end
```

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.

- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Children

`Parent` — Parent object

root object

This property is read-only.

Parent object, returned as a root object.

`Children` — Children of `Figure` object

empty `GraphicsPlaceholder` array (default) | 1-D array of component objects

Children of the `Figure` object, returned as an empty `GraphicsPlaceholder` or a one-dimensional array of component objects. The children can be any component, excluding another `Figure` object.

You cannot add or remove a child component using the `Children` property. Use this property to view the list of children or to reorder the children. The order of the children in this array reflects the front-to-back order (stacking order) of the components on the screen.

To add a child to this list, set the `Parent` property of the child component to the `Figure` object.

`HandleVisibility` — Visibility of object handle

`'off'`

This property is read-only.

Visibility of object handle, returned as `'off'`. This property provides information about the visibility of the `Figure` object in its parent's list of children. Because this property

value is always set to `'off'`, the `Figure` object is not visible in its parent's list of children and it is not returned by functions that access objects by searching the object hierarchy or querying object properties. These functions include `gcf`, `get`, `findobj`, `clf`, and `close`.

Objects are valid even if their `HandleVisibility` property is set to `'off'`. If you have access to an object, you can set and get its properties, and pass it to any function that operates on objects.

Identifiers

`Name` — Name
`''` (default) | character vector | string scalar

Name of the figure, specified as a character vector or a string scalar.

Example: `uifigure('Name','Results')` sets the name of the figure to `'Results'`.

`Number` — Number
integer | `[]`

This property is read-only.

Number of the figure, returned as an integer or empty array. You can refer to a figure using this value. For example, `figure(2)` makes the figure with a `Number` property value of `2` the current figure.

If the `IntegerHandle` property is set to `'off'`, the `Number` property is empty.

If `IntegerHandle` is `'on'`, the `Number` property is an integer. When you delete the figure, MATLAB reuses the number for the next figure.

`NumberTitle` — Use number title
`'on'` (default) | `'off'`

Number title, specified as `'on'` or `'off'`. When you set this property to `'on'`, the title of the UI figure includes the phrase `'Figure n'`, where `n` is a number.

Both the `NumberTitle` and `IntegerHandle` properties must be set to `'on'` to show to show the number in the title.

`IntegerHandle` — Use integer handle
`'off'` (default) | `'on'`

Use integer handle, specified as `'off'` or `'on'`.

If you set the `IntegerHandle` property to `'on'`, MATLAB finds the lowest integer value that is not used by an existing figure and sets the `Number` property to that value. If you delete a UI figure, MATLAB can reuse its number on a new figure.

If you set the `IntegerHandle` property to `'off'`, MATLAB does not assign an integer value to the UI figure, and it sets the `Number` property to an empty array.

To show the number in the title bar, both `IntegerHandle` and `NumberTitle` must be set to `'on'`.

Type — Type of graphics object
`'figure'`

This property is read-only.

Type of graphics object, returned as `'figure'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps”.

See Also

`appdesigner` | `uifigure`

Introduced in R2016a

uigauge

Create gauge component

Syntax

```
g = uigauge
g = uigauge(style)
g = uigauge(parent)
g = uigauge(parent,style)
g = uigauge( ___ ,Name,Value)
```

Description

`g = uigauge` creates a circular gauge in a new figure window and returns the `Gauge` object. MATLAB calls the `uifigure` function to create the figure.

`g = uigauge(style)` specifies the gauge style.

`g = uigauge(parent)` creates the gauge in the specified parent container. The parent can be a `Figure` created using the `uifigure` function, or one of its child containers.

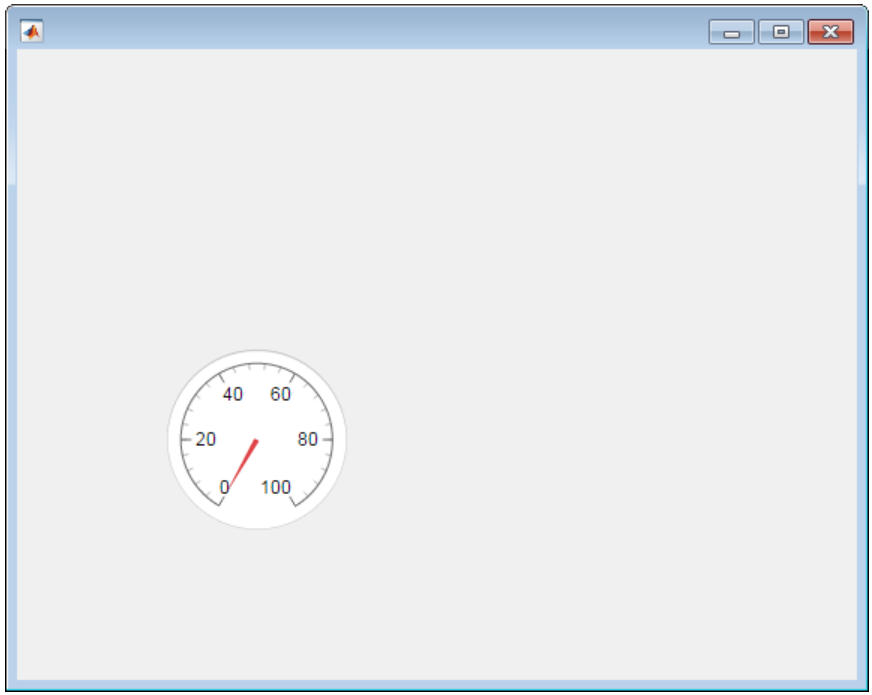
`g = uigauge(parent,style)` creates a gauge of the specified style in the specified parent container.

`g = uigauge( ___ ,Name,Value)` specifies object properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

Examples

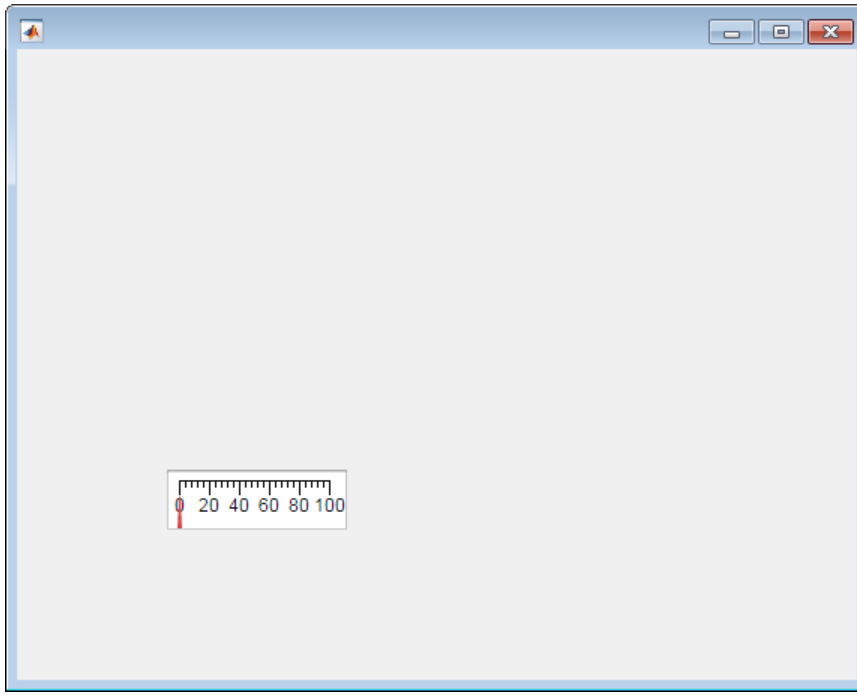
Create Circular Gauge

```
fig = uifigure;
cg = uigauge(fig);
```



Create Linear Gauge

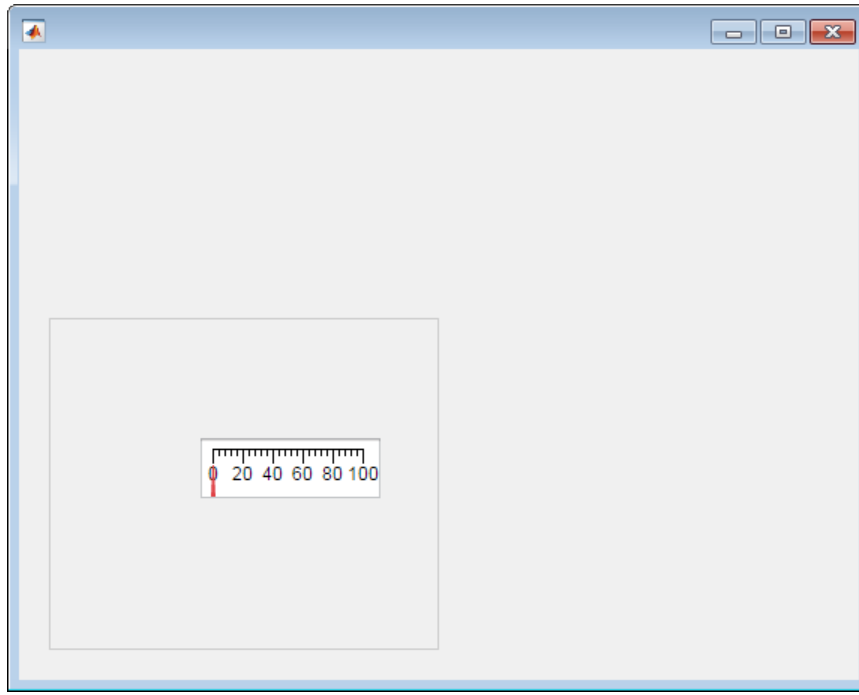
```
fig = uifigure;
lg  = uigauge(fig,'linear');
```



Create Linear Gauge in Panel

Specify a panel as the parent container for a linear gauge.

```
fig = uifigure;
pnl = uipanel(fig);
lg = uigauge(pnl,'linear');
```



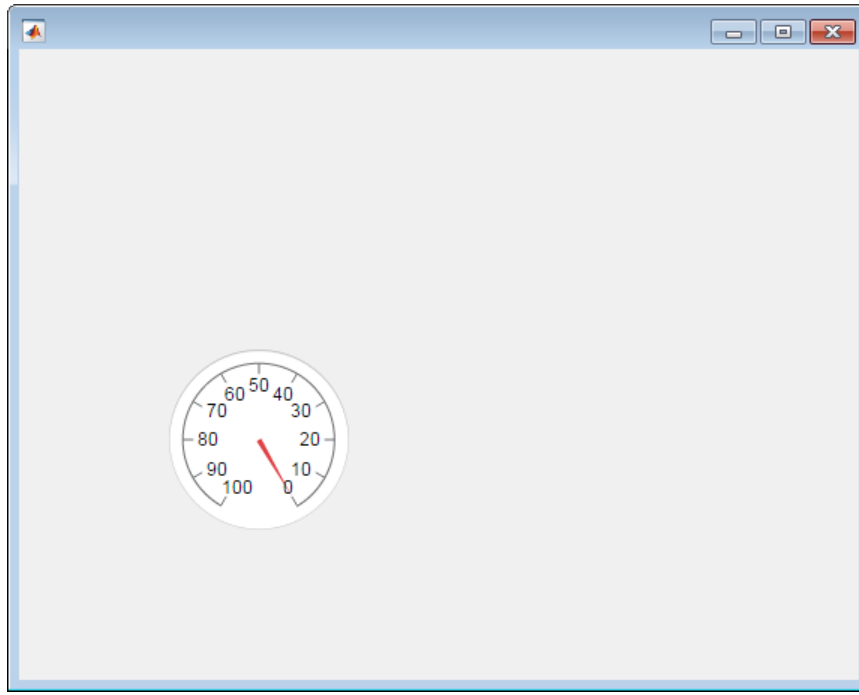
Set and Access Gauge Property Values

Create a circular gauge, and set the `ScaleDirection` property to specify that the needle moves counterclockwise.

```
fig = uifigure;
cg = uigauge(fig,'ScaleDirection','counterclockwise');
```

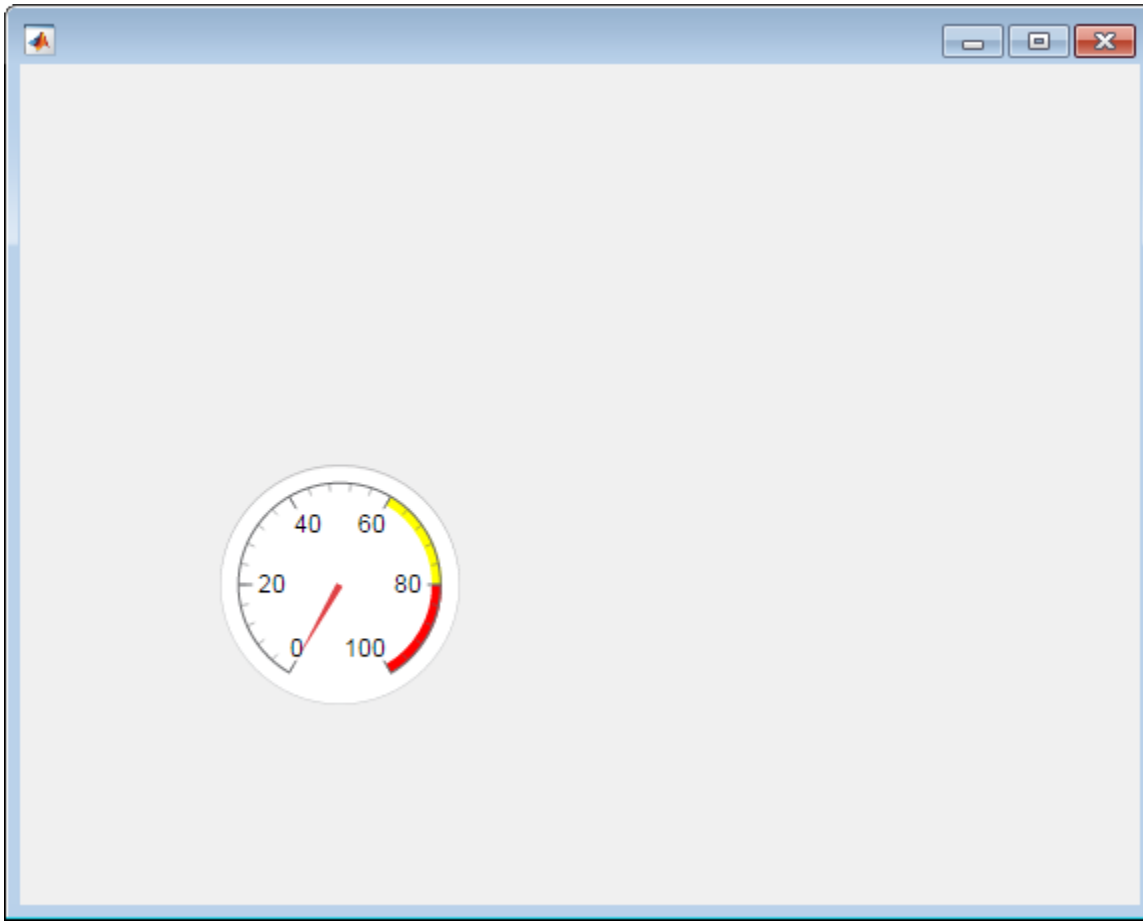
Change the number of major ticks, specify matching tick labels, and remove minor ticks.

```
cg.MajorTicks = [0:10:100];
cg.MajorTickLabels = {'0','10','20','30','40','50','60','70','80','90','100'};
cg.MinorTicks = [];
```



Specify Scale Colors and Color Limits

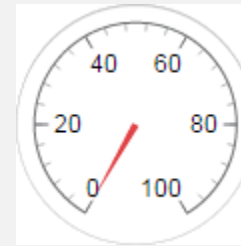
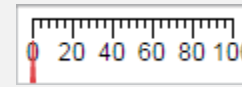
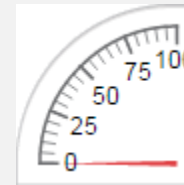
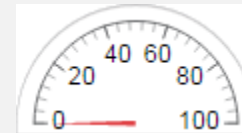
```
fig = uifigure;
cg = uigauge(fig,'ScaleColors',{'yellow','red'},...
                 'ScaleColorLimits', [60 80; 80 100]);
```



Input Arguments

`style` — Style of gauge
`'circular'` (default) | `'linear'` | `'ninetydegree'` | `'semicircular'`

Style of gauge, specified as any one of the following values:

Style	Appearance
`'circular'`	
`'linear'`	
`'ninetydegree'`	
`'semicircular'`	

`parent` — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If you do not specify a parent container, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Each type of gauge supports a different set of properties. For a full list of properties and descriptions for each type, see the associated property page.

- Gauge
- SemicircularGauge
- NinetyDegreeGauge
- LinearGauge

See Also

Functions
`appdesigner` | `uifigure`

Properties
Gauge | LinearGauge | NinetyDegreeGauge | SemicircularGauge

Introduced in R2016a

uiknob

Create knob component

Syntax

```
kb = uiknob
kb = uiknob(style)
kb = uiknob(parent)
kb = uiknob(parent,style)
kb = uiknob( ___ ,Name,Value)
```

Description

`kb = uiknob` creates a knob in a new figure window and returns the `Knob` object. MATLAB calls the `uifigure` function to create the figure.

`kb = uiknob(style)` specifies the knob style.

`kb = uiknob(parent)` creates the knob in the specified parent container. The parent can be a `Figure` created using the `uifigure` function, or one of its child containers.

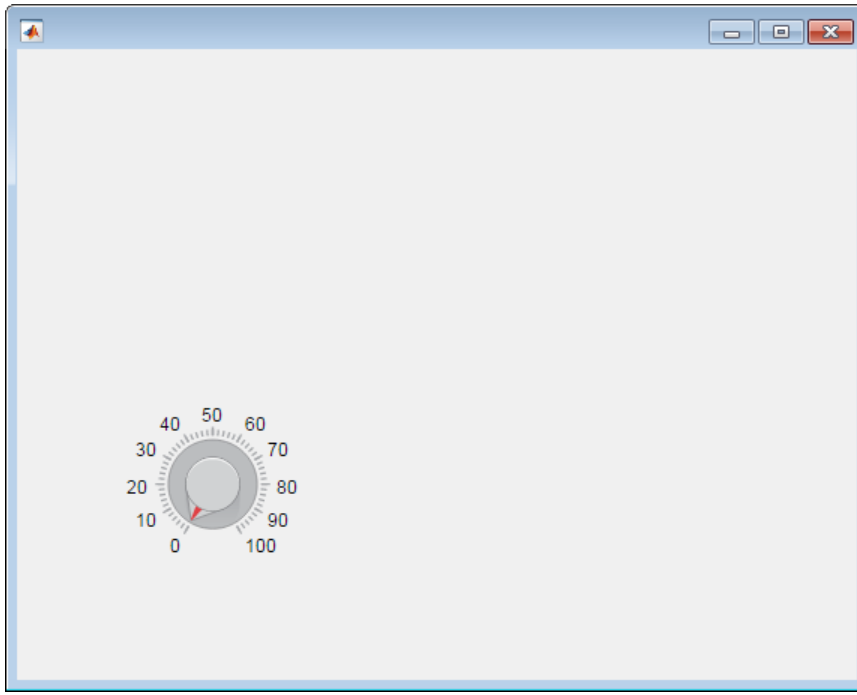
`kb = uiknob(parent,style)` creates a knob of the specified style in the specified parent container.

`kb = uiknob( ___ ,Name,Value)` specifies object properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

Examples

Create Continuous Knob

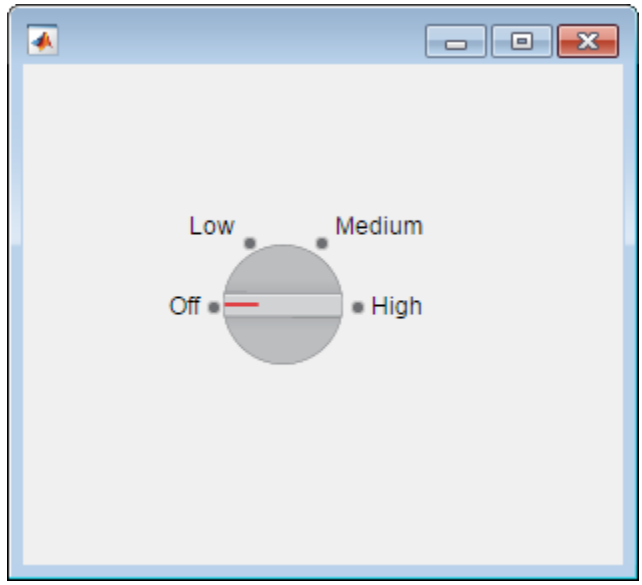
```
fig = uifigure;
kb = uiknob(fig);
```



Create Discrete Knob

Specify a `Figure` object as the parent container.

```
fig = uifigure('Position',[100 100 300 250]);
kb = uiknob(fig,'discrete');
```



Set and Access Continuous Knob Property Values

Create a continuous knob in a figure.

```
fig = uifigure;
kb = uiknob(fig);
```

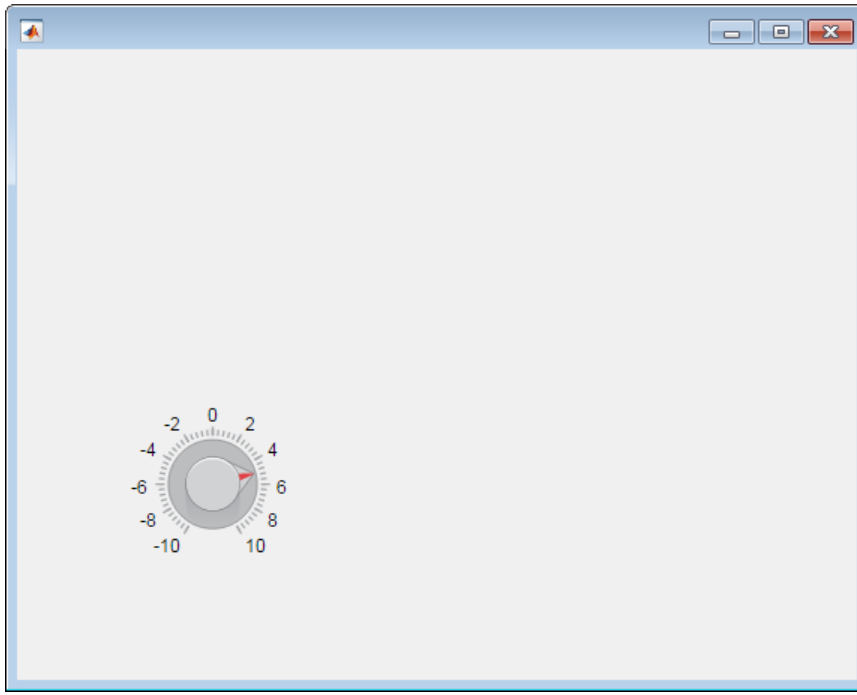
Determine the knob limits.

```
limits = kb.Limits
```

```
limits =

     0   100
```

Change the limits and the knob value.

```
kb.Limits = [-10 10];
kb.Value = 5;
```



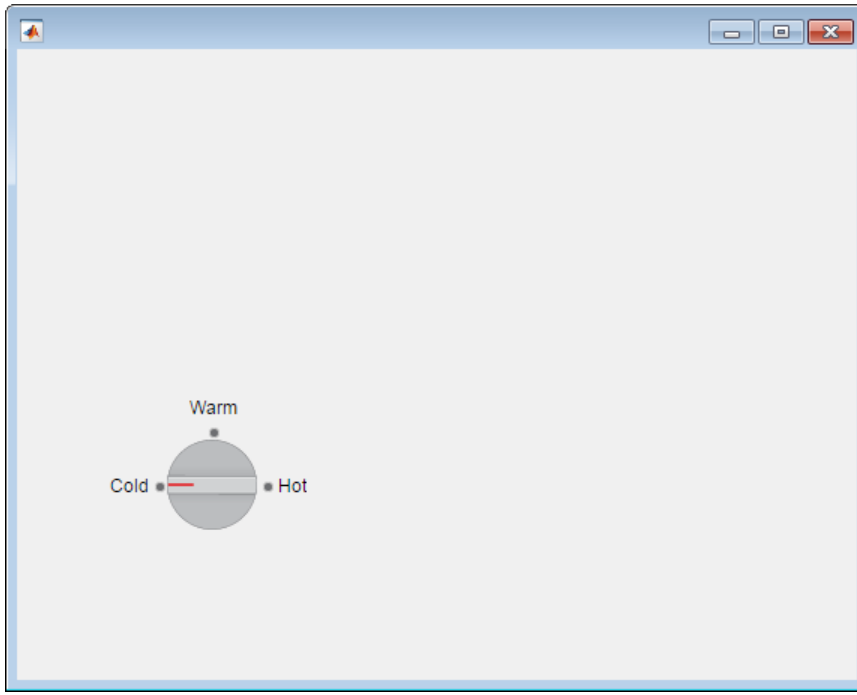
Set and Access Property Values

Create a discrete knob.

```
fig = uifigure;
kb = uiknob(fig,'discrete');
```

Change the knob states. Associate specific data with the knob states by configuring `ItemsData`. In this case, `ItemsData` reflects temperatures in degrees Fahrenheit.

```
kb.Items = {'Cold', 'Warm', 'Hot'};
kb.ItemsData = {32, 80, 212};
```



Get the temperature associated with the current knob value.

```
degrees = kb.Value
```

```
degrees =

    32
```

Code Response to Changed Discrete Knob Setting

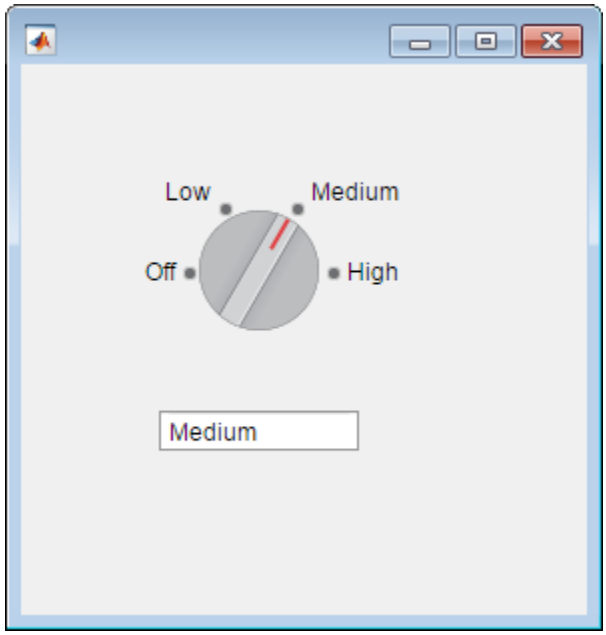
Create a discrete knob that performs an action after the app user turns it. Turning the knob updates the value of a text field to reflect the app user's choice.

Copy and paste the following code into a file named `displayknobvalue.m` on your MATLAB path. This code creates a window containing a discrete knob and a text field. It specifies a `ValueChangedFcn` callback to update the text field when the knob is turned.

```
function displayKnobValue
% Create figure window
```

```
fig = uifigure('Position',[100 100 283 275]);

% Create the text field
txt = uieditfield(fig,'text',...
    'Position', [69 82 100 22]);

% Create the knob
kb = uiknob(fig,'discrete',...
    'Position',[89 142 60 60],...
    'ValueChangedFcn',@(kb,event) knobTurned(kb,txt));
end

% Code the knob callback function
function knobTurned(knob,txt)
txt.Value = knob.Value;
end
```

Run `displayKnobValue`, and then turn the knob. When you release the mouse button, the edit field is updated to reflect the new knob value.



Code Response to Changed Continuous Knob Setting

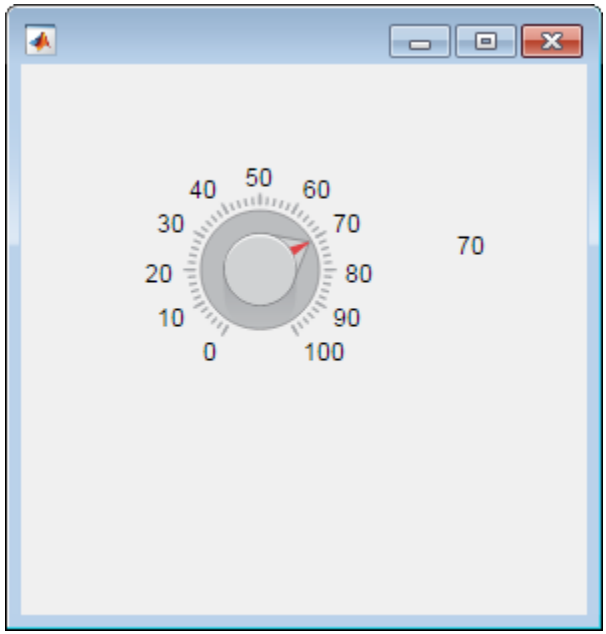
Create a continuous knob that performs an action after the user turns it. Turning the knob updates the value of a label to reflect the user's choice.

Copy and paste the following code into a file named `showknobvalue.m` on your MATLAB path. This code creates a window containing a continuous knob and a label field. It specifies a `ValueChangedFcn` callback to update the label when the knob is turned.

```
function showKnobValue
% Create figure window and components
fig = uifigure('Position',[100 100 283 275]);

% Create label
lbl = uilabel(fig,...
    'Position',[218 177 50 15],...
    'Text','0');

% Create knob
kb = uiknob(fig,...
```

```
    'Position',[89 142 60 60],...
    'ValueChangedFcn', @(kb,event) knobTurned(kb,lbl));
end

% Create ValueChangedFcn callback
function knobTurned(kb,lbl)
num = kb.Value;
lbl.Text = num2str(num);
end
```

Run `showKnobValue` and turn the knob. When you release the mouse button, the label is updated to reflect the new knob value.



Code Response to Changing Continuous Knob Setting

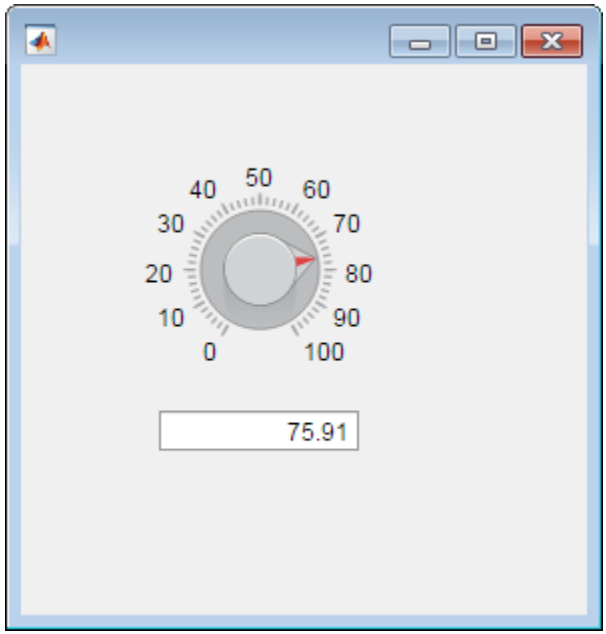
Create a continuous knob that repeatedly performs an action as the user is turning it. Instead of updating a label once when the user releases the mouse button, this knob updates the label as the knob is being turned.

Copy and paste the following code into a file named `showchangingvalue.m` on your MATLAB path. This code creates a window containing a continuous knob and a label field.

It specifies a `ValueChangingFcn` callback to keep updating the label as the knob is being turned.

```
function showChangingValue
% Create figure window
fig = uifigure('Position',[100 100 283 275]);

% Create numeric edit field
num = uieditfield(fig,'numeric',...
    'Position',[69 82 100 20]);

% Create knob
kb = uiknob(fig,...
    'Position',[89 142 60 60],...
    'ValueChangingFcn',@(kb,event) knobTurned(kb,event,num));
end

% Create ValueChangingFcn callback
function knobTurned(kb,event,num)
num.Value = event.Value;
end
```

Run `showChangingValue`, and turn the knob. As you do so, the numeric edit field is updated to show the changing knob values.



Code Response to Invalid Knob Setting

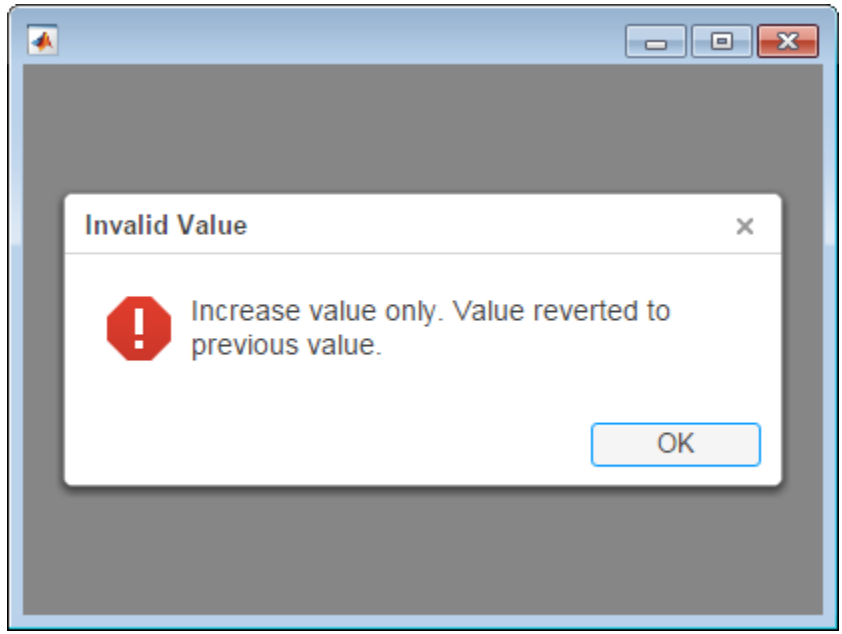
Create a continuous knob that performs an action after the user turns it. each turn of the knob causes MATLAB to perform a calculation using the current and previous knob values.

Copy and paste the following code into a file named `increaseOnly.m` on your MATLAB path. This code creates a window containing a continuous knob. It specifies a `ValueChangedFcn` callback for the knob to display an **Invalid Value** dialog box when the app user attempts to decrease the knob value.

```
function increaseOnly
% Create figure window
fig = uifigure('Position',[100 100 400 275]);

% Create knob
kb = uiknob(fig,...
    'Position',[150 125 60 60],...
    'ValueChangedFcn',@(kb,event) nValChanged(kb,event,fig));
end
```

```
% Create ValueChangedFcn callback
function nValChanged(kb,event,fig)
newvalue = event.Value;
previousvalue = event.PreviousValue;
if previousvalue >  newvalue
   uialert(fig, 'Increase value only. Value set to previous value.',...
        'Invalid Value');
   kb.Value = previousvalue;
end
end
```

Run `increaseOnly`, increase the knob value, and then try to decrease it. When you try to decrease the value, an error dialog box displays and the value is reverted to the previous valid value. You can only increase the knob value.

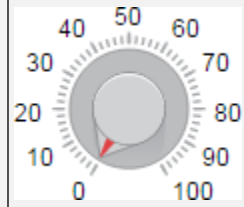
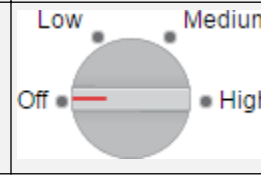


Input Arguments

`style` — Style of knob

`'continuous'` (default) | `'discrete'`

Style of knob, specified as one of the following values:

Style	Appearance
`'continuous'`	
`'discrete'`	

`parent` — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If you do not specify a parent container, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Each type of knob object supports a different set of properties. For a full list of properties and descriptions for each type, see the associated property page.

- Knob
- DiscreteKnob

See Also

Functions
`appdesigner` | `uifigure`

Properties
DiscreteKnob | Knob

Introduced in R2016a

uilabel

Create label component

Syntax

```
lbl = uilabel
lbl = uilabel(parent)
lbl = uilabel( ___ ,Name,Value)
```

Description

`lbl = uilabel` creates a label component (with the text `'Label'`) in a new in a new figure window and returns the `Label` object. MATLAB calls the `uifigure` function to create the figure.

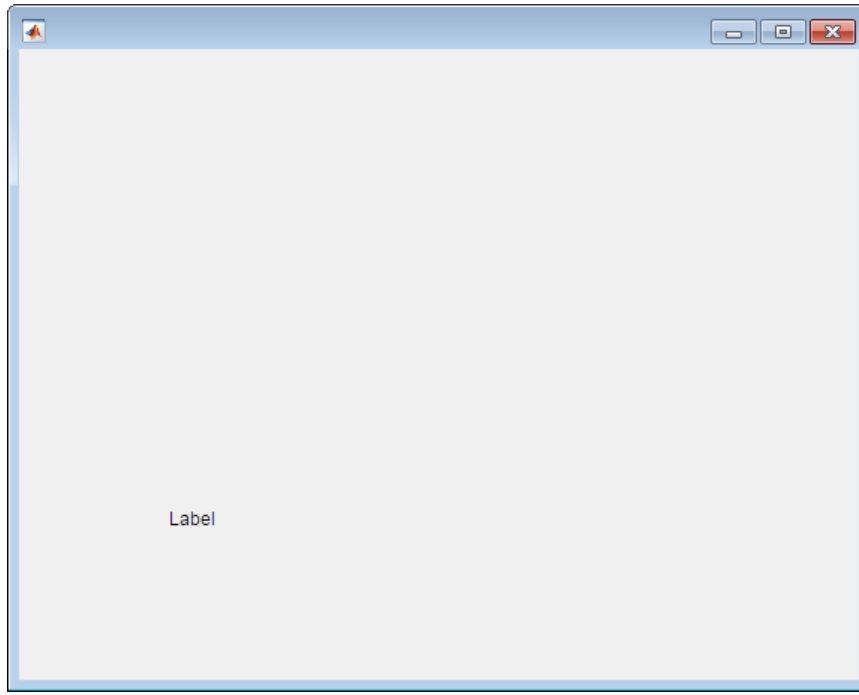
`lbl = uilabel(parent)` creates the label in the specified parent container. The parent can be a `Figure` created using the `uifigure` function, or one of its child containers.

`lbl = uilabel( ___ ,Name,Value)` specifies label properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

Examples

Create Label Component with Default Text

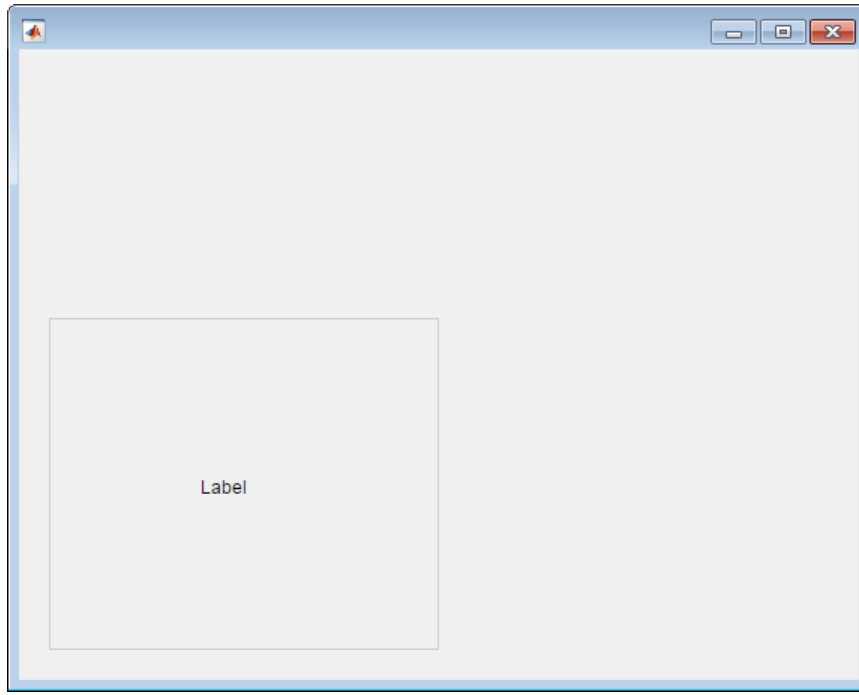
```
f = uifigure;
label = uilabel(f);
```



Create Label Component in Panel

Specify a `Panel` as the parent.

```
fig = uifigure;
pnl = uipanel(fig);
lbl = uilabel(pnl);
```



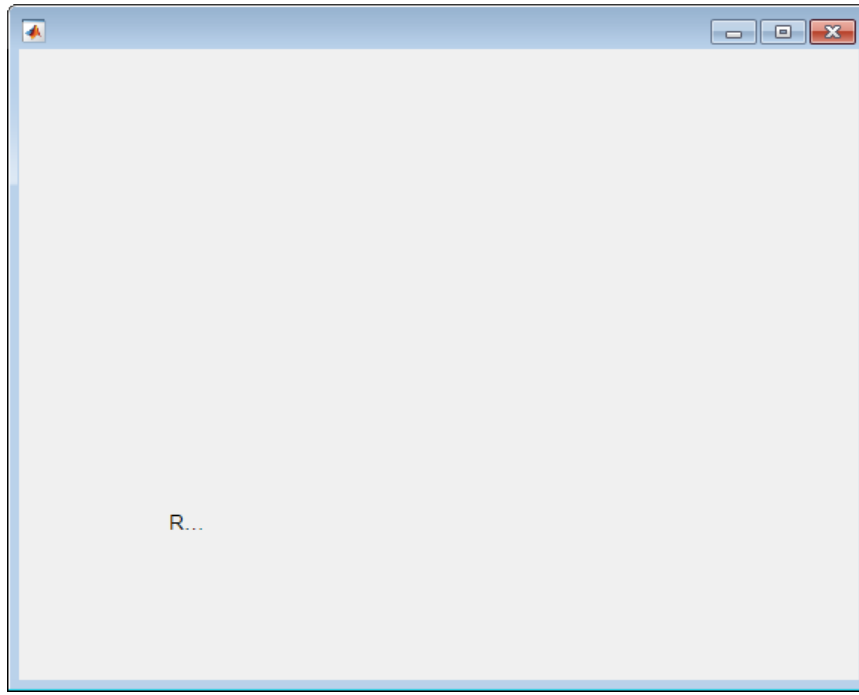
Set and Access Label Component Properties

Create a default label.

```
fig = uifigure;
lbl = uilabel(fig);
```

Change the label text and font size.

```
lbl.Text = 'Result';
lbl.FontSize = 14;
```



The label is clipped because the current label size is too small for the new text at the new font size.

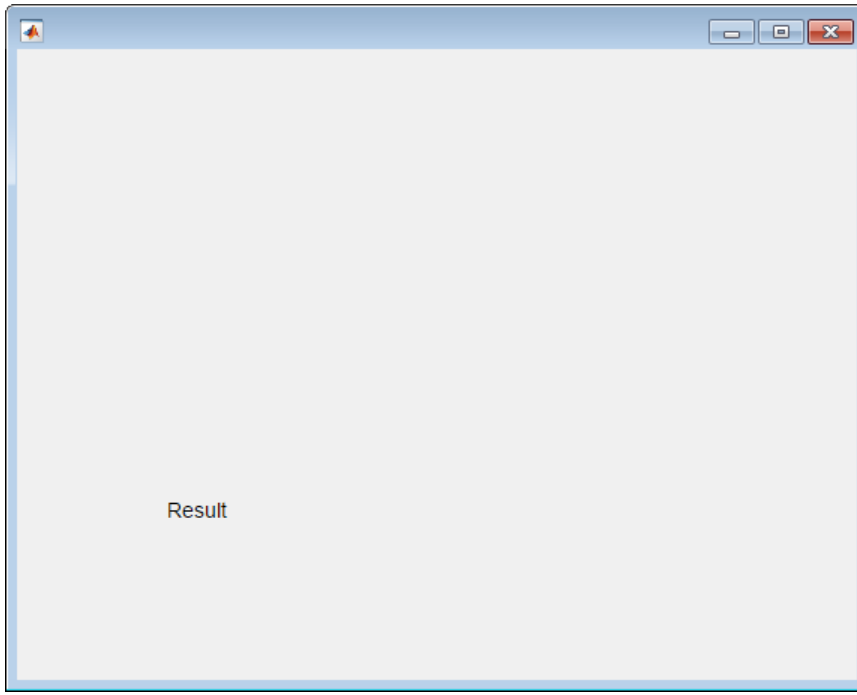
Determine the current label size by getting the third and fourth elements of the `Position` property value.

```
size = lbl.Position(3:4)
```

```
size =

    31    15
```

Change the label size to accommodate the new text.

```
lbl.Position(3:4) = [62 22];
```



Input Arguments

parent — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If you do not specify a parent container, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The properties listed here are a subset of the available properties. For the full list, see Label.

Example: `'Text','Sum:'` specifies the label displays the text `Sum:`.

`Text` — Label text
`'Label'` (default) | character vector | cell array of character vectors | string scalar | string array | ...

Label text, specified as a character vector, cell array of character vectors, string scalar, string array, or 1-D categorical array. Use a cell array of character vectors or a string array to specify multiple lines of text.

Alternatively, use the `sprintf` function to create formatted text containing line breaks and other special characters.

```
text = sprintf('%s\n%s','Line 1','Line 2');
label = uilabel('Text',text,'Position',[100 100 100 32]);
```

If you specify text as a character vector without using `sprintf`, MATLAB will not interpret control sequences such as `\n`.

If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

Example: `'Threshold'`

Example: `{'Threshold' 'Value'}`

`Position` — Label location and size
`[100 100 31 22]` (default) | `[left bottom width height]`

Label location and size, relative to the parent, specified as the vector `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the label
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the label
`width`	Distance between the right and left outer edges of the label
`height`	Distance between the top and bottom outer edges of the label

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

All measurements are in pixel units.

Example: `[100 100 100 20]`

See Also

Functions
`appdesigner` | `uifigure`

Properties
Label

Introduced in R2016a

uilamp

Create lamp component

Syntax

```
lmp = uilamp
lmp = uilamp(parent)
lmp = uilamp( ___ ,Name,Value)
```

Description

`lmp = uilamp` creates a lamp in a new figure window and returns the `Lamp` object. MATLAB calls the `uifigure` function to create the figure.

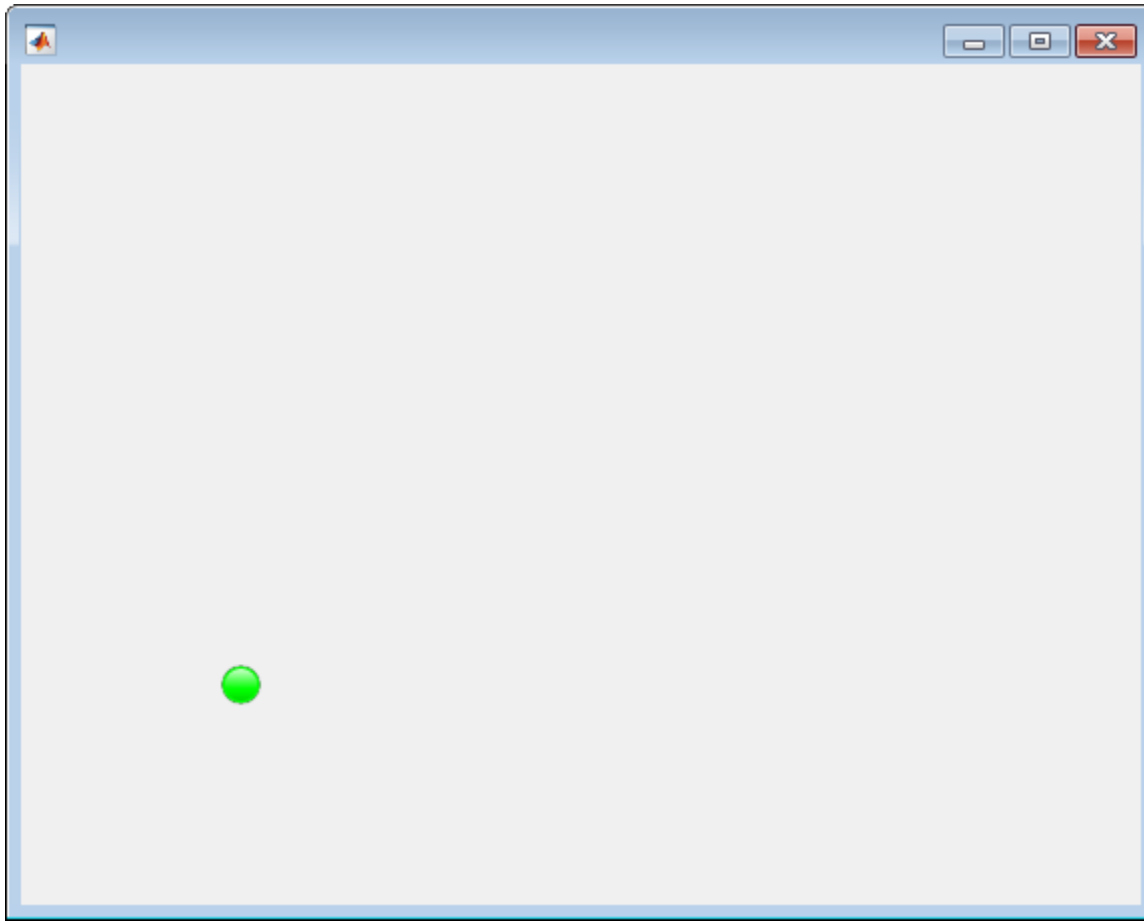
`lmp = uilamp(parent)` creates the lamp in the specified parent container. The parent can be a `Figure` created using the `uifigure` function, or one of its child containers.

`lmp = uilamp( ___ ,Name,Value)` specifies `Lamp` properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

Examples

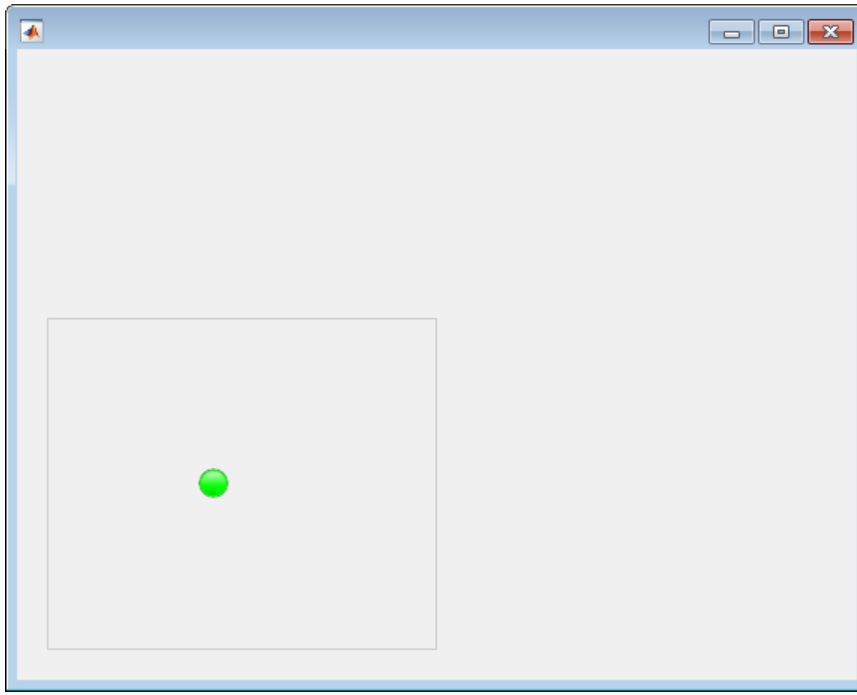
Create Lamp in Figure

```
fig = uifigure;
lmp = uilamp(fig);
```



Create Lamp in Panel

```
fig = uifigure;
pnl= uipanel(fig);
lmp = uilamp(pnl);
```



Set and Access Lamp Properties

Create a default lamp.

```
fig = uifigure;
lmp = uilamp(fig);
```

Determine the current color of the lamp.

```
color = lmp.Color
```

```
color =

     0     1     0
```

MATLAB returns the RGB value for green.

Change the lamp color to red by setting the `Color` property to the long name, `'red'`.

```
lmp.Color = 'red';
```

Change the lamp color to blue by setting the `Color` property to the RGB triplet for blue.

```
lmp.Color = [0 0 1];
```

Input Arguments

`parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If you do not specify a parent container, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The properties listed here are a subset of the available properties. For the full list, see Lamp.

Example: `'Color','red'` specifies that the lamp color is red.

`Color` — Color of lamp
`[0 1 0]` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Color of the lamp, specified as an RGB triplet, a hexadecimal color code, or one of the color options listed in the table.

RGB triplets and hexadecimal color codes are useful for specifying custom colors.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`.

- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `F`. The values are not case sensitive. Thus, the color codes `'#FF8800'`, `'#ff8800'`, `'#F80'`, and `'#f80'` are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`'red'`	`'r'`	`[1 0 0]`	`'#FF0000'`	
`'green'`	`'g'`	`[0 1 0]`	`'#00FF00'`	
`'blue'`	`'b'`	`[0 0 1]`	`'#0000FF'`	
`'cyan'`	`'c'`	`[0 1 1]`	`'#00FFFF'`	
`'magenta'`	`'m'`	`[1 0 1]`	`'#FF00FF'`	
`'yellow'`	`'y'`	`[1 1 0]`	`'#FFFF00'`	
`'black'`	`'k'`	`[0 0 0]`	`'#000000'`	
`'white'`	`'w'`	`[1 1 1]`	`'#FFFFFF'`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`'#0072BD'`	
`[0.8500 0.3250 0.0980]`	`'#D95319'`	
`[0.9290 0.6940 0.1250]`	`'#EDB120'`	
`[0.4940 0.1840 0.5560]`	`'#7E2F8E'`	
`[0.4660 0.6740 0.1880]`	`'#77AC30'`	
`[0.3010 0.7450 0.9330]`	`'#4DBEEE'`	
`[0.6350 0.0780 0.1840]`	`'#A2142F'`	

Data Types: `double` | `char`

Position — Location and size of lamp

`[100 100 20 20]` (default) | `[left bottom width height]`

Location and size of the lamp, specified as the vector `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the lamp
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the lamp
`width`	Distance between the right and left outer edges of the lamp
`height`	Distance between the top and bottom outer edges of the lamp

All measurements are in pixel units.

Due to aspect ratio constraints, you cannot change the lamp height and width independently of one another.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[100 200 20 20]`

See Also

Functions
`appdesigner` | `uifigure`

Properties
Lamp

Introduced in R2016a

uilistbox

Create list box component

Syntax

```
lb = uilistbox
lb = uilistbox(parent)
lb = uilistbox( ___ ,Name,Value)
```

Description

`lb = uilistbox` creates a list box in a new figure window and returns the `ListBox` object. MATLAB calls the `uifigure` function to create the figure.

`lb = uilistbox(parent)` creates the list box in the specified parent container. The parent can be a `Figure` created using the `uifigure` function, or one of its child containers.

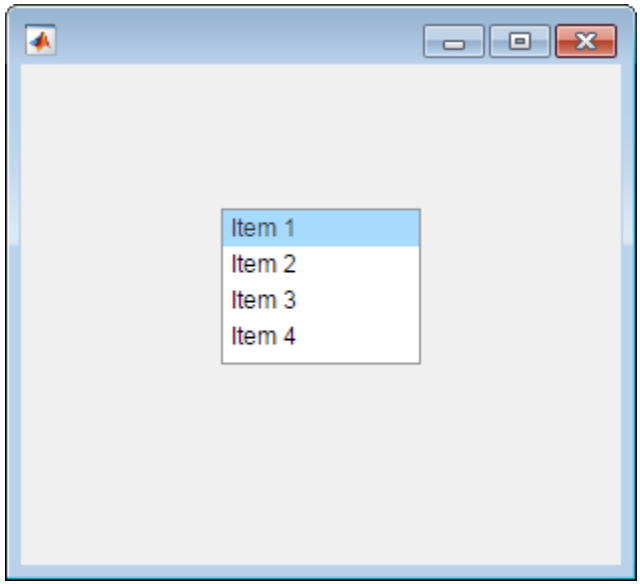
`lb = uilistbox( ___ ,Name,Value)` specifies `ListBox` properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

Examples

Create List Box

Create a list box in a figure window.

```
fig = uifigure('Position', [100 100 300 250]);
lbx = uilistbox(fig);
```



Set and Access List Box Property Values

Create a list box.

```
fig = uifigure;
lbx = uilistbox(fig);
```

Determine whether the list box allows multiple selections.

```
multi = lbx.Multiselect
```

```
multi =

     off
```

Enable multiselection.

```
lbx.Multiselect = 'on';
```

Display List Box Selection

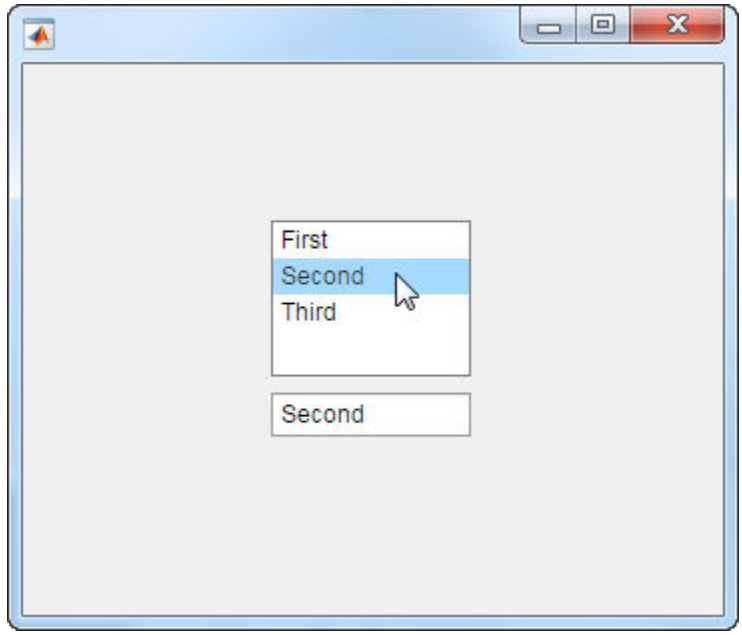
Create a list box that performs an action when the user selects an item in the list.

Save the following code as `selectlistbox.m` on your MATLAB path.

This code creates an app containing a list box and a text area. The `ValueChangedFcn` callback updates the text area to display the list box selection.

```
function selectlistbox

fig = uifigure('Position',[100 100 350 275]);

% Create text area
txt = uitextarea(fig,...
    'Position',[125 90 100 22],...
    'Value','First');

% Create list box
lbox = uilistbox(fig,...
    'Position',[125 120 100 78],...
    'Items',{'First','Second','Third'},...
    'ValueChangedFcn', @updateEditField);

% ValueChangedFcn callback
function updateEditField(src,event)
    txt.Value = src.Value;
end

end
```

Run `selectlistbox` and select an option from the list.



Display List Box Data

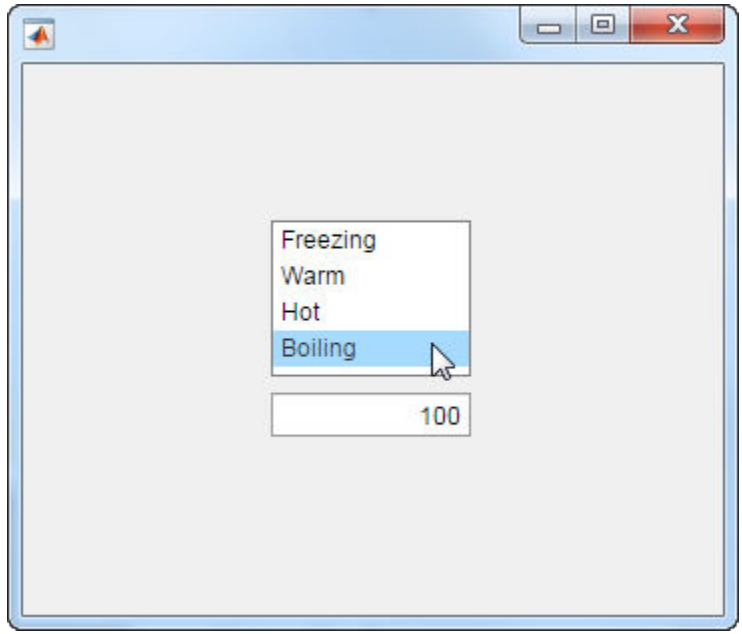
Create a list box that has a numeric value associated with each item. When the user selects an item in the list box, the edit field displays the associated numeric value.

Save the following code as `dataselection.m` on your MATLAB path. This code creates an app containing a list box and a numeric edit field. Each item in the list has a temperature associated with it. When the user selects an item in the list, the `ValueChangedFcn` callback displays the corresponding temperature in the edit field.

```
function dataselection
fig = uifigure('Position',[100 100 350 275]);

% Create Numeric Edit Field
ef = uieditfield(fig,'numeric',...
    'Position',[125 90 100 22]);

% Create List Box
lbox = uilistbox(fig,...
```

```
    'Items', {'Freezing', 'Warm', 'Hot', 'Boiling'},...
    'ItemsData', [0, 25, 40, 100],...
    'Position',[125 120 100 78],...
    'ValueChangedFcn', @selectionChanged);

% ValueChangedFcn callback
function selectionChanged(src,event)
    % Display list box data in edit field
    ef.Value = src.Value;
end

end
```

Run `dataselection` and select an item in the list. The numeric edit field updates to reflect the temperature associated with the selection.



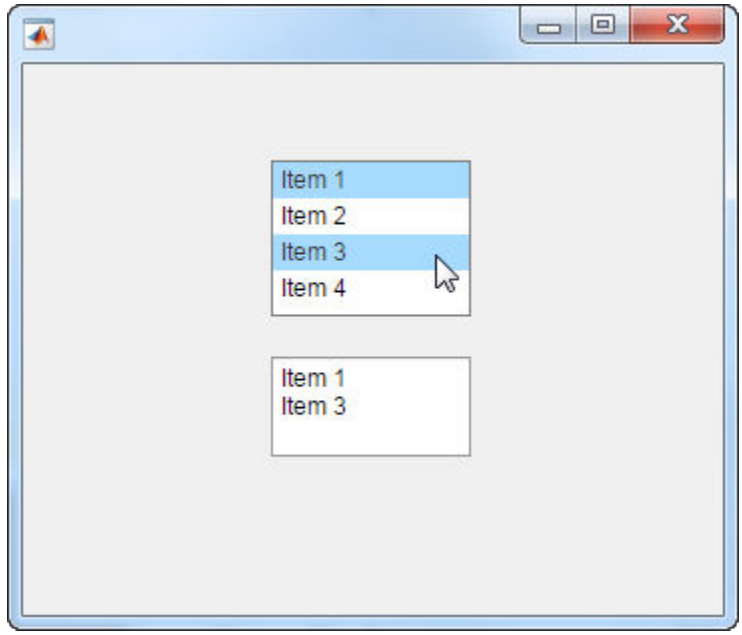
List Box That Allows Selection of Multiple Items

Create an app containing a list box that allows multiple items to be selected. Write the `ValueChangedFcn` callback to display the selected items in the text area below the list box.

Save the following code as `multiselect.m` on your MATLAB path.

```
function multiselect
fig = uifigure('Position',[100 100 350 275]);

% Create Text Area
txt = uitextarea(fig,...
    'Position',[125 80 100 50]);

% Create List Box
lbox = uilistbox(fig,...
    'Position',[125 150 100 78],...
    'Multiselect','on',...
    'ValueChangedFcn',@selectionChanged);

% ValueChangedFcn callback
function selectionChanged(src,event)
    txt.Value = src.Value;
end

end
```

Run `multiselect` and select items from the list. The text area displays your selection.



Input Arguments

`parent` — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If you do not specify a parent container, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The properties listed here are a subset of the available properties. For the full list, see ListBox.

Example: `'Items',{'Model 1','Model 2', 'Model 3', 'Model 4'}` specifies the list box options that the app user sees, from top to bottom.

`Value` — Value
element of `Items` | element of `ItemsData` | `{}`

Value, specified as an element of the `Items` array, `ItemsData` array, or an empty cell array. By default, `Value` is the first element in `Items`.

To specify no selection, set `Value` to an empty cell array.

Specifying `Value` as an element of `Items` selects the list item that matches that element. If `ItemsData` is not empty, then `Value` must be set to an element of `ItemsData`, and the list box will select the associated item in the list.

`Items` — List box items
`{'Item 1','Item 2', 'Item 3', 'Item 4'}` (default) | 1-by-n cell array of character vectors | string array | ...

List box items, specified as a cell array of character vectors, string array, or 1-D categorical array. Duplicate elements are allowed. The list box displays as many options as there are elements in the `Items` array. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

`ItemsData` — Data associated with each element of the `Items` property value
empty array (`[]`) (default) | 1-by-n numeric array | 1-by-n cell array

Data associated with each element of the `Items` property value, specified as a 1-by-n numeric array or a 1-by-n cell array. Duplicate elements are allowed.

For example, if you set the `Items` value to employee names, you might set the `ItemsData` value to corresponding employee ID numbers. The `ItemsData` value is not visible to the app user.

If the number of array elements in the `ItemsData` value and the `Items` value do not match, one of the following occurs:

- When the `ItemsData` value is empty, then all the elements of the `Items` value are presented to the app user.

- When the `ItemsData` value has more elements than the `Items` value, then all the elements of the `Items` value are presented to the app user. MATLAB ignores the extra `ItemsData` elements.
- When the `ItemsData` value is not empty, but has fewer elements than the `Items` value, the only elements of the `Items` value presented to the app user are those that have a corresponding element in the `ItemsData` value.

Example: `{'One','Two','Three'}`

Example: `[10 20 30 40]`

`Multiselect` — Multiple node selection
`'off'` (default) | `'on'`

Multiple node selection, specified as `'off'` or `'on'`. Set this property to `'on'` to allow users to select multiple items simultaneously.

`ValueChangedFcn` — Value changed function
`''` (default) | function handle | cell array | character vector

Value changed function, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback function executes when the user selects a different item in the list box. It does not execute if the `Value` property setting changes programmatically.

This callback function can access specific information about the user's interaction with the list box. MATLAB passes this information in a `ValueChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.PreviousValue` returns the previous value of the list box. The `ValueChangedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangedData` object.

Property	Value
`Value`	Value of list box after app user's most recent interaction with it
`PreviousValue`	Value of list box before app user's most recent interaction with it
`Source`	Component that executes the callback
`EventName`	`'ValueChanged'`

For more information about writing callbacks, see "Write Callbacks in App Designer".

`Position` — Location and size of list box

`[100 100 100 74]` (default) | `[left bottom width height]`

Location and size of the list box relative to the parent container, specified as the vector `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the list box
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the list box
`width`	Distance between the right and left outer edges of the list box
`height`	Distance between the top and bottom outer edges of the list box

All measurements are in pixel units.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[100 100 100 200]`

Tips

Use the `scroll` function to programmatically scroll a list box item or the top or bottom of the list into view.

See Also

Functions
`appdesigner` | `scroll` | `uifigure`

Properties
ListBox

Introduced in R2016a

uiradiobutton

Create radio button component

Syntax

```
rb = uiradiobutton
rb = uiradiobutton(parent)
rb = uiradiobutton( ___ ,Name,Value)
```

Description

`rb = uiradiobutton` creates a radio button within a button group and returns the `RadioButton` object. MATLAB calls the `uifigure` function to create the parent figure of the button group.

`rb = uiradiobutton(parent)` creates the radio button within the specified button group. The button group must be the child of a `Figure` created with the `uifigure` function, or must be parented to a child container of the figure: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`.

`rb = uiradiobutton( ___ ,Name,Value)` specifies `RadioButton` properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

Examples

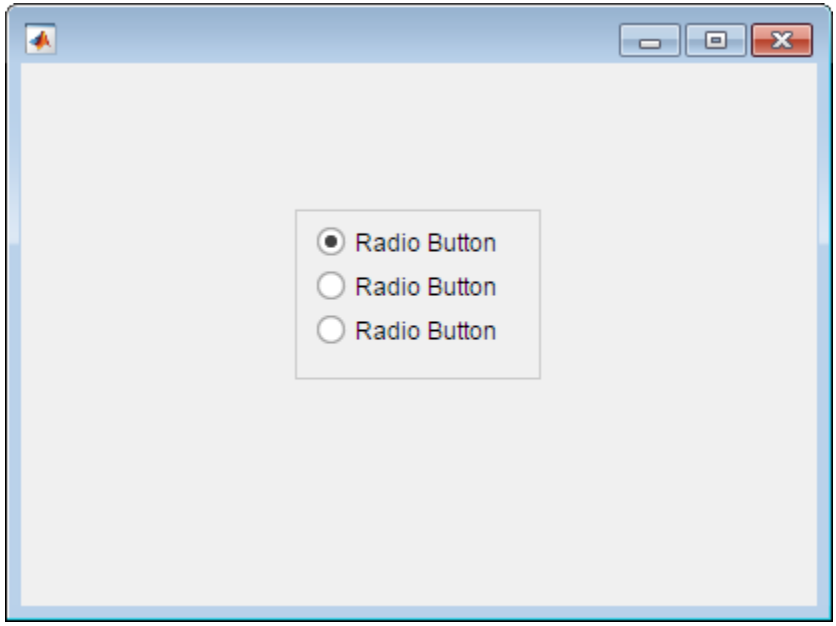
Create Radio Buttons Within a Button Group

Create a button group in a window.

```
fig = uifigure('Position',[680 678 398 271]);
bg = uibuttongroup(fig,'Position',[137 113 123 85]);
```

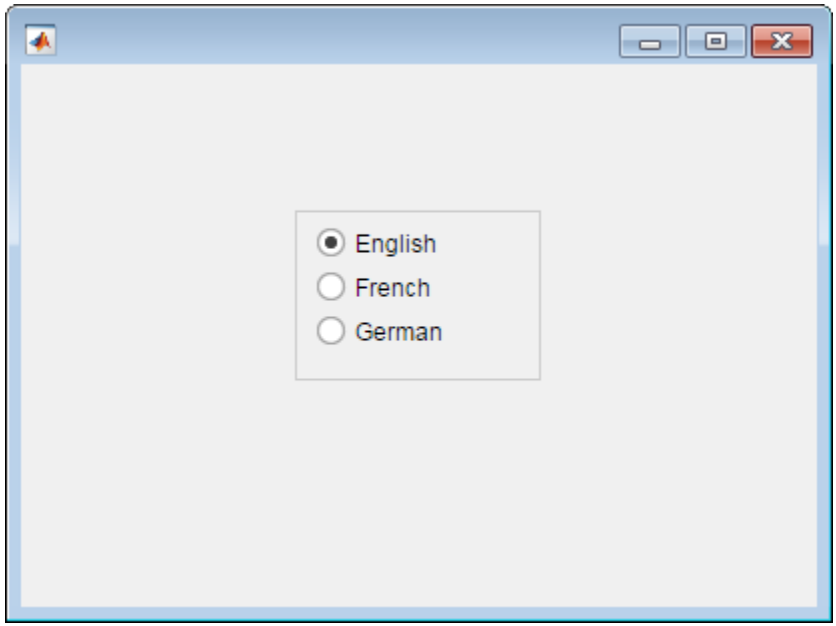
Create three `RadioButton` objects, and specify the parent and location of each.

```
rb1 = uiradiobutton(bg,'Position',[10 60 91 15]);
rb2 = uiradiobutton(bg,'Position',[10 38 91 15]);
rb3 = uiradiobutton(bg,'Position',[10 16 91 15]);
```



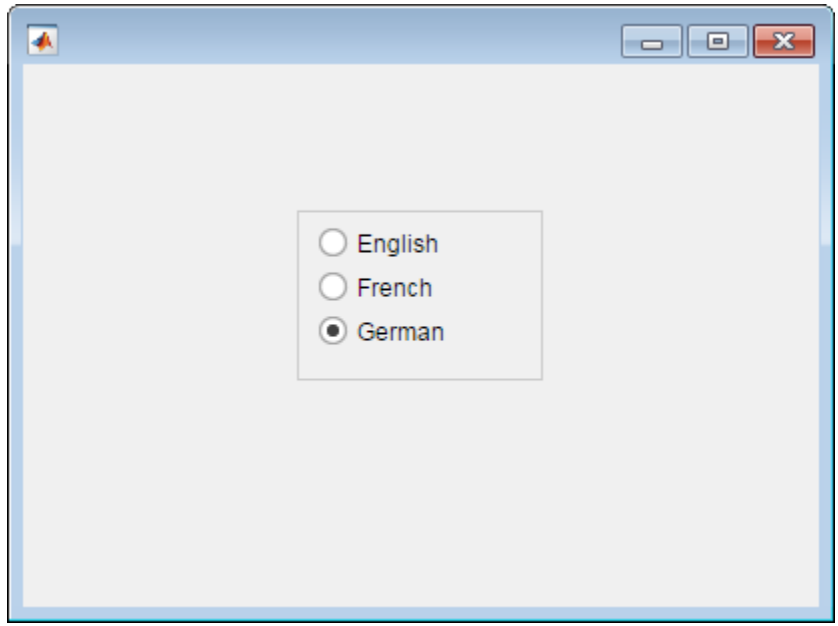
Change the text associated with each radio button.

```
rb1.Text = 'English';
rb2.Text = 'French';
rb3.Text = 'German';
```



Change the radio button selection to German.

```
rb3.Value = true;
```



Determine the font name of the German radio button text.

```
font = rb3.FontName
```

```
font =

Helvetica
```

Input Arguments

parent — Parent container
ButtonGroup object

Parent container, specified as a `ButtonGroup` object. The `ButtonGroup` must be parented to a `Figure` created using the `uifigure` function, or to a child container of a `uifigure`, such as: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The properties listed here are a subset of the available properties. For the full list, see RadioButton.

Example: `'Text', 'French'` specifies that the text “French” displays next to the radio button.

`Value` — State of radio button

`1` | `0`

State of radio button, specified as `1` (true) or `0` (false). Within a button group, only one button can be selected at a time. The state of the first radio button added to a button group is `1`, by default. Subsequent buttons added to the same button group have a default state of `0`.

When the `Value` property of a `RadioButton` changes to `1`, the `Value` of the property for the previously selected `RadioButton` changes to `0`. In addition, the `SelectedObject` property value of the parent `ButtonGroup` is updated.

If you programmatically change the `Value` property of a `RadioButton` to `0`, MATLAB sets the `Value` property of the first `RadioButton` *added* to the `ButtonGroup` to `1`. If the first `RadioButton` added is the one for which you programmatically set the `Value` property to `0`, then MATLAB sets the `Value` property for the second `RadioButton` added to the `ButtonGroup` to `1`.

Note The first `RadioButton` *added* to a `ButtonGroup` is not necessarily the first `RadioButton` *listed* in the `Children` property of the `ButtonGroup`.

`Text` — Button label

`'Radio Button'` (default) | character vector | cell array of character vectors | string scalar | string array | ...

Button label, specified as a character vector, cell array of character vectors, string scalar, string array, or 1-D categorical array. Specify a character vector or string scalar to label

the button with a single line of text. Use a cell array or string array to label the button with multiple lines of text. Each element in the array represents a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

`Position` — Location and size of radio button
`[10 10 91 22]` (default) | `[left bottom width height]`

Location and size of radio button, specified as a vector of the form `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the button group to the outer left edge of the radio button
`bottom`	Distance from the inner bottom edge of the button group to the outer bottom edge of the radio button
`width`	Distance between the right and left outer edges of the radio button (including text)
`height`	Distance between the top and bottom outer edges of the radio button

The `Position` values are relative to the drawable area of the button group. The drawable area is the area inside the borders of the button group and does not include the area occupied by the title.

All measurements are in pixel units.

Tips

- Button groups can contain any UI component type, but can only manage the selection of radio buttons and toggle buttons.
- To make your program respond when the user selects a radio button or toggle button that is inside a button group. define a `SelectionChangedFcn` callback function for the `ButtonGroup`. You cannot define callbacks for the individual buttons.
- To determine which radio button or toggle button is selected, query the `SelectedObject` property of the `ButtonGroup`. You can execute this query anywhere in your code.

- If you set the `Visible` property of a button group object to `'off'`, then any child objects it contains become invisible along with the parent `ButtonGroup`. However, the `Visible` *property value* of each child object remains unaffected.

See Also

Functions
`appdesigner` | `uibuttongroup` | `uifigure`

Properties
RadioButton

Introduced in R2016a

uislider

Create slider component

Syntax

```
sld = uislider
sld = uislider(parent)
sld = uislider( ___ ,Name,Value)
```

Description

`sld = uislider` creates a slider in a new figure window and returns the `Slider` object. MATLAB calls the `uifigure` function to create the figure.

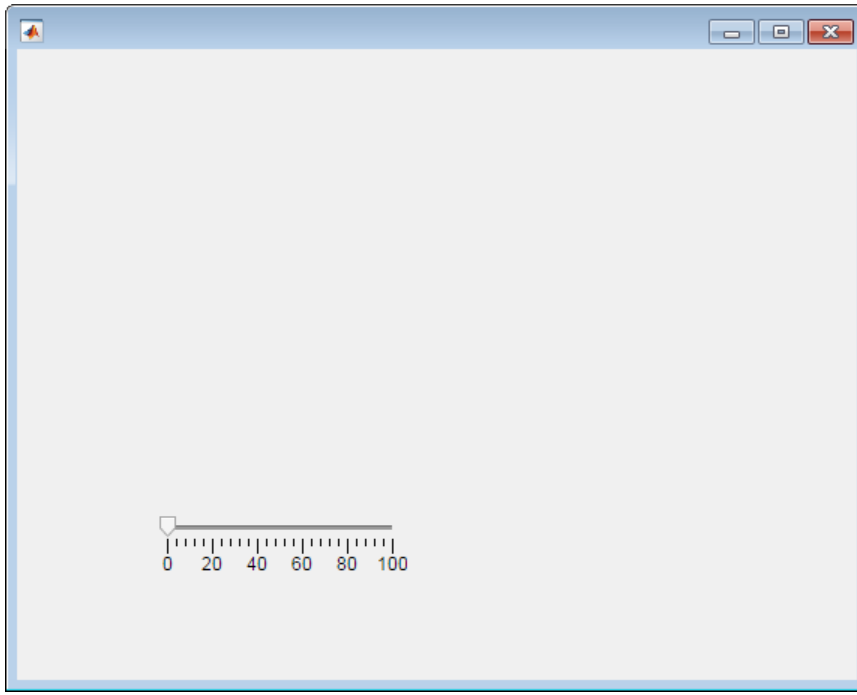
`sld = uislider(parent)` creates the slider in the specified parent container. The parent can be a `Figure` created using the `uifigure` function, or one of its child containers.

`sld = uislider( ___ ,Name,Value)` specifies `Slider` properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

Examples

Create Slider in Figure Window

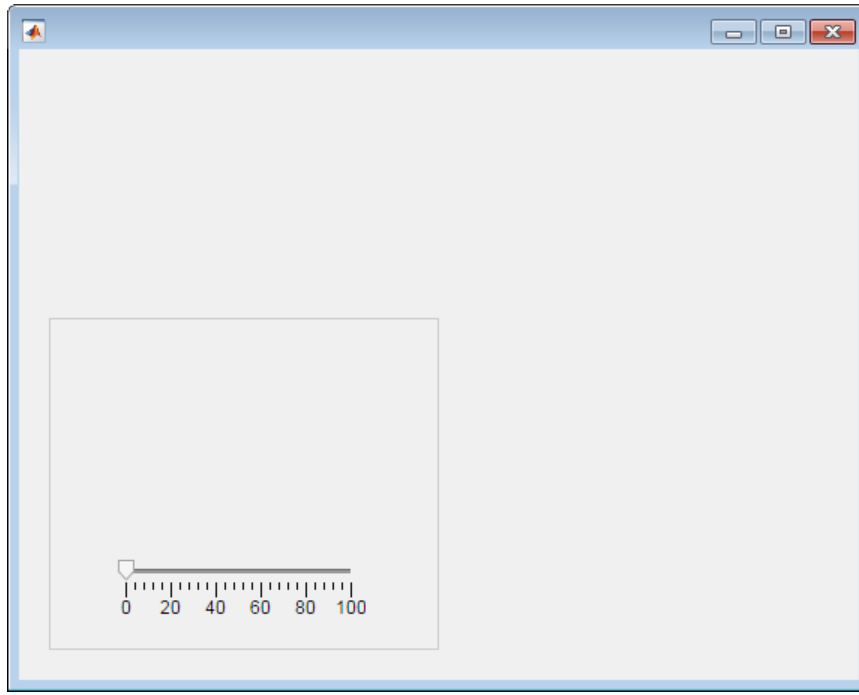
```
fig = uifigure;
sld = uislider(fig);
```



Create Slider in Panel

Create a uifigure window containing a panel. Create a slider and specify its position within the panel.

```
fig = uifigure;
pnl = uipanel(fig);
sld = uislider(pnl,'Position',[50 50 150 3]);
```



Set and Change Slider Property Values

Create a slider. Set the `Value` property to 50.

```
fig = uifigure;
sld = uislider(fig,'Value',50);
```

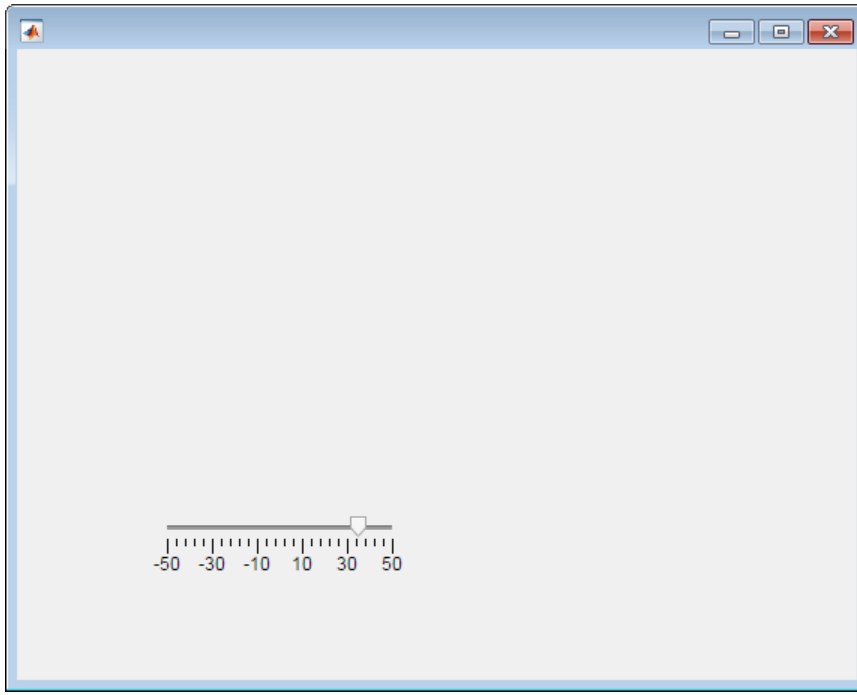
Determine the current slider limits.

```
limits = sld.Limits
```

```
limits =

     0   100
```

Change the slider limits and set the value to 35.

```
sld.Limits = [-50 50];
sld.Value = 35;
```



Code Response to Moved Slider Thumb

Create a slider and a gauge. When an app user moves the slider thumb and releases the mouse button, the needle of the gauge reflects the slider value.

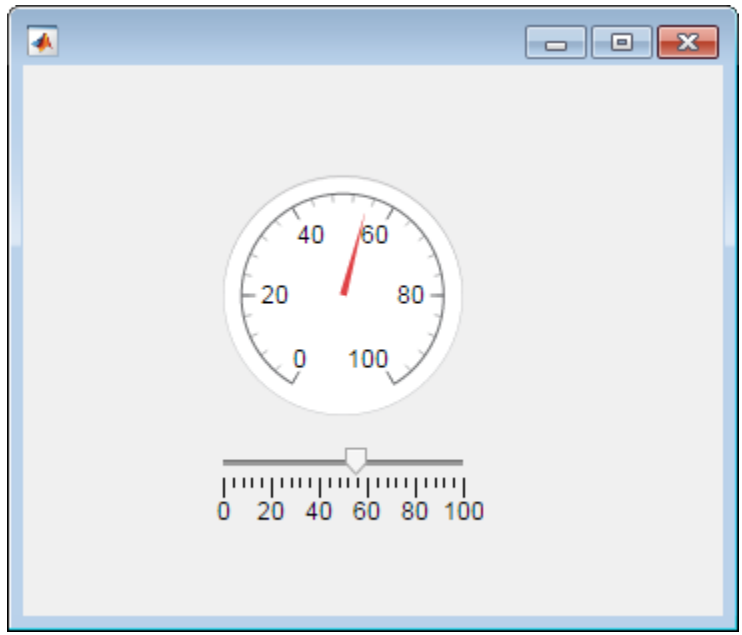
Save the following code to `sliderValue.m` on your MATLAB path.

This code creates a window containing a slider and a gauge. When an app user moves the slider thumb, the `ValueChangedFcn` callback updates the gauge to reflect the slider value.

```
function slidervalue
% Create figure window and components
```

```
fig = uifigure('Position',[100 100 350 275]);

cg = uigauge(fig,'Position',[100 100 120 120]);

sld = uislider(fig,...
    'Position',[100 75 120 3],...
    'ValueChangedFcn',@(sld,event) updateGauge(sld,cg));

end

% Create ValueChangedFcn callback
function updateGauge(sld,cg)
cg.Value = sld.Value;
end
```

Run `sliderValue`, and then move the slider thumb. When you release the mouse button, the circular gauge needle moves to the matching value on the gauge.



Code Response to Moving Slider

Create a slider and a gauge. As an app user moves the slider thumb, the needle of the gauge reflects the changing slider value.

This code creates a window containing a slider and a gauge. As an app user moves the slider thumb, the `ValueChangingFcn` callback updates the gauge to reflect the slider value.

Save the following code to `sliderChanging.m` on your MATLAB path.

```
function sliderchanging
% Create figure window and components

fig = uifigure('Position',[100 100 350 275]);

cg = uigauge(fig,'Position',[100 100 120 120]);

sld = uislider(fig,...
               'Position',[100 75 120 3],...
               'ValueChangingFcn',@(sld,event) sliderMoving(event,cg));

end

% Create ValueChangingFcn callback
function sliderMoving(event,cg)
cg.Value = event.Value;
end
```

Run `sliderChanging`, and then move the slider. As you move the slider, the circular gauge needle moves, reflecting the slider value.

Input Arguments

`parent` — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If you do not specify a parent container, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The properties listed here are a subset of the available properties. For the full list, see Slider.

Example: `'Limits',[0 50]` specifies the minimum slider value as `0` and the maximum slider value as `50`.

`Value` — Slider value
0 (default) | numeric value

Slider value, specified as a numeric value. The numeric value must be within the range specified by the `Limits` property value.

`Limits` — Minimum and maximum slider values
[0 100] (default) | two-element numeric array

Minimum and maximum slider values, specified as a two-element numeric array. The first value must be less than the second value.

If you change `Limits` such that `Value` property is less than the new lower limit, MATLAB sets the `Value` property to the new lower limit. For example, suppose the `Limits` property is `[0 100]` and `Value` is 20. If the `Limits` changes to `[50 100]`, then MATLAB sets the `Value` property to 50.

Similarly, if you change `Limits` such that the `Value` property is greater than the new upper limit, MATLAB sets the `Value` property to the new upper limit.

`MajorTicks` — Major tick mark locations
`[0 20 40 60 80 100]` (default) | vector of numeric values | `[]`

Major tick mark locations, specified as a vector of numeric values or an empty vector. If you do not want to show major tick marks, specify this property as an empty vector.

Tick locations that are outside the range of the `Limits` property do not display.

MATLAB removes duplicate tick values. However, if a major tick falls on the same value as a minor tick, only the major tick displays.

Setting the `MajorTicks` property sets the `MajorTicksMode` property to `'manual'`.

`MajorTickLabels` — Major tick labels
`{'0','20','40','60','80','100'}` (default) | cell array of character vectors | string array | `{}` | ...

Major tick labels, specified as a cell array of character vectors, string array, or 1-D categorical array. If you do not want to show tick labels, specify this property as an empty cell array. If you want to remove a label from a specific tick mark, specify an empty character vector or empty string scalar for the corresponding element in the `MajorTickLabels` array. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

If the length of the `MajorTickLabels` array is different from the length of the `MajorTicks` vector, MATLAB ignores the extra entries of the longer array. If there are extra labels, they are ignored. If there are extra tick marks, they display without labels.

Setting `MajorTickLabels` changes the `MajorTickLabelsMode` value to `'manual'`.

`ValueChangedFcn` — Value changed callback
`''` (default) | function handle | cell array | character vector

Value changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when the user moves the thumb to a different position on the slider. The callback does not execute if the slider value changes programmatically.

This callback function can access specific information about the user's interaction with the slider. MATLAB passes this information in a `ValueChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.PreviousValue` returns the previous value of the slider. The `ValueChangedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangedData` object.

Property	Value
`Value`	Value of slider after app user's most recent interaction with it
`PreviousValue`	Value of slider before app user's most recent interaction with it
`Source`	Component that executes the callback
`EventName`	`'ValueChanged'`

For more information about writing callbacks, see "Write Callbacks in App Designer".

`ValueChangingFcn` — Value changing callback
`''` (default) | function handle | cell array | character vector

Value changing callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes as the user moves the thumb along the slider in the app. It does not execute if the `Value` property changes programmatically.

This callback can access specific information about the user's interaction with the slider. MATLAB passes this information in a `ValueChangingData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.Value` returns the current value of the slider. The `ValueChangingData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangingData` object.

Property	Value
`Value`	Current value of the slider as the app user is interacting with it
`Source`	Component that executes the callback

Property	Value
`EventName`	`'ValueChanging'`

The `Value` property of the `Slider` object is not updated until the user releases the slider thumb. Therefore, to get the value as the thumb is being moved, your code must get the `Value` property of the `ValueChangingData` object.

The `ValueChangingFcn` callback executes as follows:

- If the app user clicks the slider value once. then the callback executes a single time. For example, if the slider is on 1.0, and the app user single-clicks at 1.1, then the callback executes once.
- If the app user clicks and drags the slider to a new position, the callback executes repeatedly. For example, if the slider value is 1.0, and the app user clicks, holds, and drags the thump to value 10.0, then the callback executes multiple times until the app user releases the thumb.

For more information about writing callbacks, see “Write Callbacks in App Designer”.

`Position` — Location and size of slider
`[100 100 150 3]` (default) | `[left bottom width height]`

Location and size of the slider excluding tick marks and labels, specified as the vector `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the slider
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the slider
`width`	Distance between the right and left outer edges of the slider
`height`	Distance between the top and bottom outer edges of the slider

All measurements are in pixel units.

You cannot change the height of a slider when the `Orientation` property value is `'horizontal'`. Similarly, you cannot change the width of a slider when the `Orientation` property value is `'vertical'`.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[100 200 60 60]`

See Also

Functions
`appdesigner` | `uifigure`

Properties
Slider

Introduced in R2016a

uispinner

Create spinner component

Syntax

```
spn = uispinner
spn = uispinner(parent)
spn = uispinner( ___ ,Name,Value)
```

Description

`spn = uispinner` creates a spinner in a new figure window and returns the `Spinner` object. MATLAB calls the `uifigure` function to create the figure.

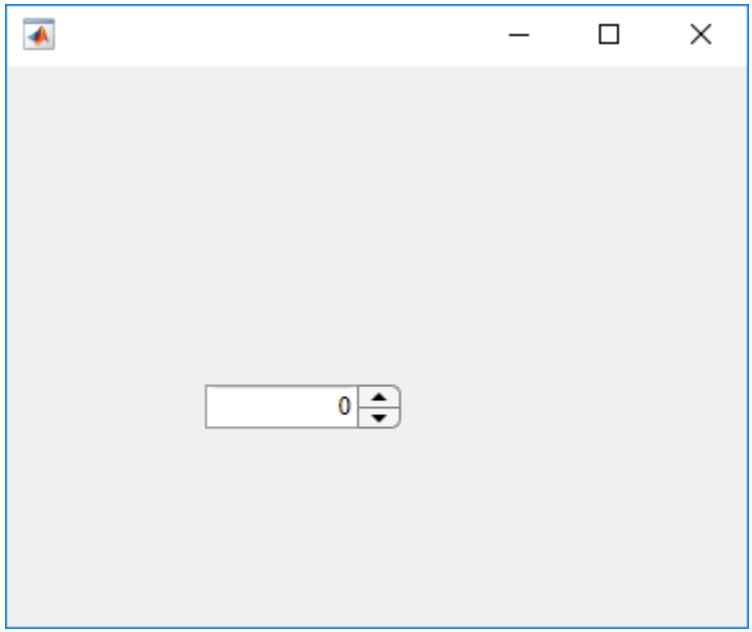
`spn = uispinner(parent)` creates the spinner in the specified parent container. The parent can be a `Figure` created using the `uifigure` function, or one of its child containers.

`spn = uispinner( ___ ,Name,Value)` specifies `Spinner` properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

Examples

Create Spinner in Figure Window

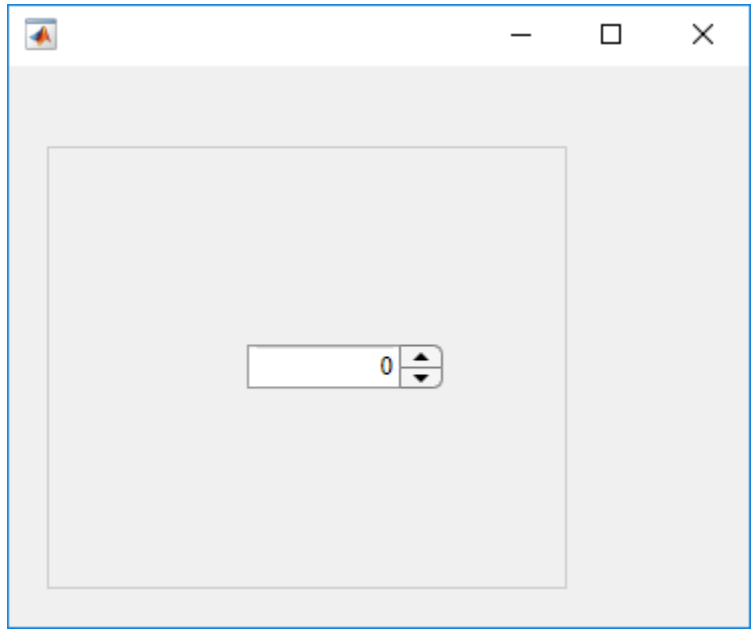
```
fig = uifigure;
spn = uispinner(fig);
```



Create Spinner in Panel

Create a spinner in a panel.

```
fig = uifigure;
pnl = uipanel(fig);
spn = uispinner(pnl);
```



Set and Access Spinner Property Values

Create a spinner that limits the values the app user can enter to between 0 and 100, inclusive.

Create a spinner.

```
fig = uifigure;
spn = uispinner(fig);
```

Determine the limits. The returned values indicate that the lower and upper limits are unlimited.

```
limits = spn.Limits
```

```
limits =

  -Inf   Inf
```

Set the limits to 0 and 100.

```
spn.Limits = [0 100];
```

Create Spinner and Specify Limit Inclusiveness

Create a spinner that allows the app user to enter a value greater than -5 and less than or equal to 10.

```
fig = uifigure;
spn = uispinner(fig,'Limits', [-5  10],...
            'LowerLimitInclusive','off',...
            'UpperLimitInclusive','on',...
            'Value', 5);
```

Run the code. If you enter a value in the spinner that is outside the limits, MATLAB automatically displays a message indicating the problem. MATLAB then restores the value to the previous valid value.

Create Spinner That Displays Values Using Exactly Two Decimals

Create a spinner that allows the app user to enter any value, but always displays the value using exactly two decimals. Be aware that MATLAB stores the exact value that the app user enters.

```
fig = uifigure;
spn = uispinner(fig,'ValueDisplayFormat', '%.2f');
```

Run the code, and then enter 5.555 in the spinner. Click outside the spinner. The spinner displays 5.55.

MATLAB stores the original value, 5.555.

Click in the spinner, it displays the value originally typed.

Code Response to Changed Spinner Value

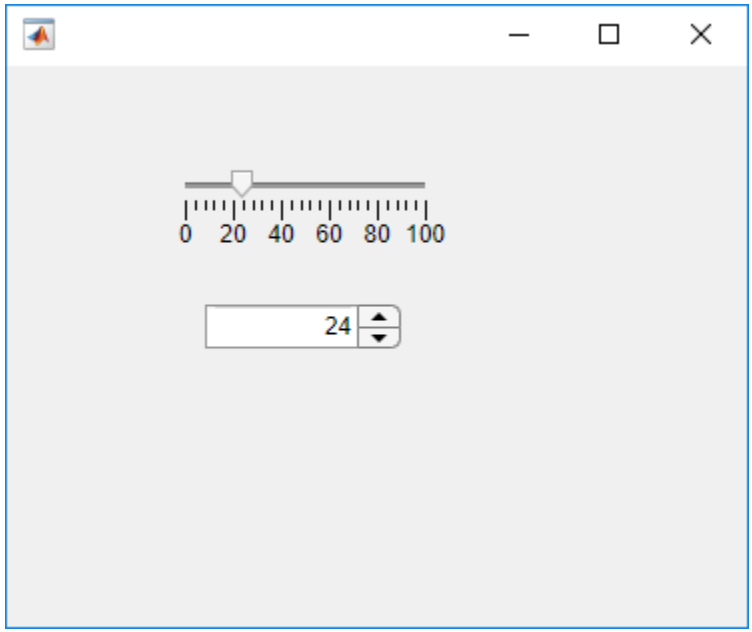
Create a spinner and a slider. When an app user changes the spinner value, the slider updates to match that value.

Save the following code to `spinnerValue.m` on your MATLAB path. This code creates a figure window containing a slider and a spinner. When an app user changes the spinner value, the `ValueChangedFcn` updates the spinner to reflect the slider value.

```
function spinnervalue
fig = uifigure('Position',[100 100 370 280]);

sld = uislider(fig,...
    'Position',[90 220 120 3]);

spn = uispinner(fig,...
    'Position',[100 140 100 22],...
    'ValueChangedFcn',@(spn,event) updateSlider(spn,sld));
end

% Create ValueChangedFcn callback
function updateSlider(spn,sld)
sld.Value = spn.Value;
end
```

Run `spinnerValue`.

Click and hold the up arrow in the spinner until the value reaches 24, and then release. The slider thumb moves to indicate the spinner value.



Code Response to Changing Spinner Value

Create a spinner and a slider. As an app user changes the spinner value, the slider repeatedly updates to match that value.

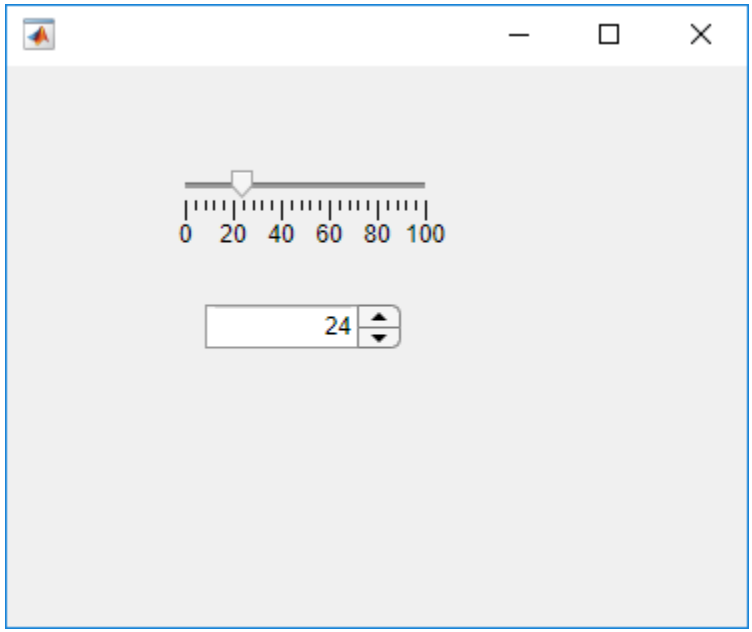
Save the following code to `showChangingValue.m` on your MATLAB path. This code creates a figure window containing a slider and a spinner. As an app user changes the spinner value, the `ValueChangingFcn` repeatedly updates the slider to reflect the spinner value as it changes.

```
function showChangingValue
fig = uifigure('Position',[100 100 370 280]);
sld = uislider(fig,...
    'Position',[90 220 120 3]);

spn = uispinner(fig,...
    'Position',[100 140 100 22],...
    'ValueChangingFcn',@(spn,event) spinnerChanging(event,sld));
end
```

```
% Create ValueChangingFcn callback
function spinnerChanging(event,sld)
sld.Value = event.Value;
end
```

Run `showChangingValue`.

Click, and hold the up arrow in the spinner until the value reaches 24, and then release. The slider moves as the spinner value changes.



Code Response to Calculate Changed Spinner Value

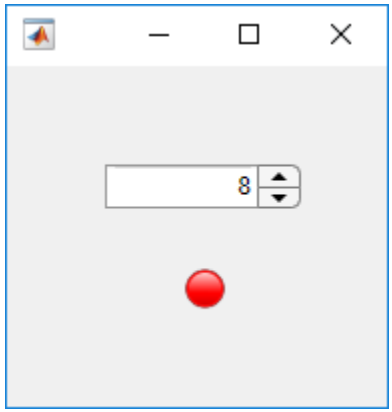
Code the `ValueChangedFcn` callback to determine if the value is rising or falling compared to the previous spinner value. Set lamp color to green when the value is increasing and to red when the value is decreasing

Save the following code to `upOrDown.m` on your MATLAB path.

```
function upOrDown
fig = uifigure(...
    'Position',[100 100 190 170]);

lmp = uilamp(fig,...
    'Position',[90 50 20 20],...
    'Color','green');

spn = uispinner(fig,...
    'Position',[50 100 100 22],...
    'ValueChangedFcn',@(spn,event) spinnerValueChanged(event,lmp));
end

% Create ValueChangedFcn that uses event data
function spinnerValueChanged(event,lmp)
newValue = event.Value;
previousValue = event.PreviousValue;
difference = newValue-previousValue;
if difference > 0
    lmp.Color = 'green';
else
    lmp.Color = 'red';
end
end
```

Run `upOrDown`.

Each time you change the spinner value, the `ValueChangedFcn` determines whether the value is increasing or decreasing and sets the lamp color accordingly.



Input Arguments

`parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If you do not specify a parent container, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The properties listed here are a subset of the available properties. For the full list, see Spinner.

Example: `'Value',150` specifies that the number 50 appears in the spinner.

`Value` — Spinner value
0 (default) | double-precision number

Spinner value, specified as a double-precision number.

When the app user types a value in the spinner, the value is a character vector. When the app user presses the **Enter** key or changes focus, MATLAB converts the app-user-entered value to a double-precision number.

MATLAB rejects the value if:

- It cannot convert the character vector to a scalar number.
- The value is NaN, blank, or a complex number.
- The value is a mathematical expression, such as `1+2`.
- The value is less than or greater than the values specified by the `Limits` property.

When MATLAB rejects the app-user-entered value, a tooltip appears describing the value requirements. The spinner immediately reverts to its previous value and no `ValueChangedFcn` runs.

Example: 10

`ValueDisplayFormat` — Value display format
`'%11.4g'` (default) | character vector | string scalar

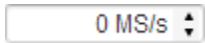
Value display format, specified as a character vector or string scalar.

MATLAB uses `sprintf` to display the value using the specified format.

You can mix text with format operators. For example:

```
spin = uispinner('ValueDisplayFormat','%.0f MS/s');
```

The resulting spinner component looks like this:



When the app user clicks in the spinner field, the field shows the value without the text.

For a complete list of supported format operators, see `sprintf`.

`RoundFractionalValues` — Rounding of fractional values
`'off'` (default) | `'on'`

Rounding of fractional values entered by app users, specified as one of the following:

- `'on'` — MATLAB rounds the value if it results in a valid value and executes the `ValueChangedFcn` callback. If the resulting value is outside the lower or upper `Limits`, then MATLAB rounds to the nearest value that falls within the `Limits` and then executes the callback.
- `'off'` — MATLAB does not round a fractional value to a whole number.

If the `RoundFractionalValues` property value changes from `'off'` to `'on'` programmatically, then MATLAB applies these rules:

- If rounding the existing value yields an integer that lies inside the limit range specified by the `Limits` property, then MATLAB rounds up the existing value.
- If rounding the existing value yields an integer that is less than the lower limit, then MATLAB rounds up the existing value.
- If rounding the existing value yields an integer that is greater than the upper limit, then MATLAB rounds down the existing value.
- If the limits are configured such that there is no valid integer in the range, then MATLAB sets the `RoundFractionalValues` property value back to `'off'` and displays an error message.

`Step` — Quantity by which value is incremented or decremented
1 (default) | numeric scalar

Quantity by which the `Value` property increments or decrements when the app user presses the up and down arrows, respectively.

`Limits` — Minimum and maximum spinner values
[-Inf Inf] (default) | two-element numeric array

Minimum and maximum spinner values, specified as a two-element numeric array. The first value must be less than the second value. Set array elements to `-Inf` or `Inf` to specify no minimum or no maximum, respectively.

If you change `Limits` such that the `Value` property is outside the new limits, MATLAB sets the `Value` property to a value within the new limits. For example, suppose the `Limits` property is `[0 100]` and the `Value` property is 20. If the `Limits` property changes to `[50 100]`, then MATLAB sets the `Value` property to 50 (assuming the `LowerLimitInclusive` value is `'on'`.

Example: `[-Inf 200]`

Example: `[-100 Inf]`

Example: `[-100 200]`

`LowerLimitInclusive` — Lower limit inclusiveness
`'on'` (default) | `'off'`

Lower limit inclusiveness, specified as one of the following:

- `'on'` — Value must be equal to or greater than the lower limit.
- `'off'` — Value must be greater than the lower limit.

`UpperLimitInclusive` — Upper limit inclusiveness
`'on'` (default) | `'off'`

Upper limit inclusiveness, specified as one of the following:

- `'on'` — Value must be equal to or less than the upper limit.
- `'off'` — Value must be less than the upper limit.

For example, if you want the numeric input to be between 0 and 1, excluding 0 and 1, do all of the following:

- Set the `Limits` property value to `[0 1]`.
- Set the `UpperLimitInclusive` property to `'off'`.
- Set the `LowerLimitInclusive` property to `'off'`.

`ValueChangedFcn` — Value changed callback
`''` (default) | function handle | cell array | character vector

Value changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback executes when the user changes focus or presses the **Enter** key after changing the spinner value. It does not matter whether the user changes the spinner value by typing or by pressing the arrow keys. The callback does not execute if the spinner value changes programmatically.

This callback function can access specific information about the user's interaction with the spinner. MATLAB passes this information in a `ValueChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.PreviousValue` returns the previous value of the spinner. The `ValueChangedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangedData` object.

Property	Value
`Value`	Value of spinner after app user's most recent interaction with it
`PreviousValue`	Value of spinner before app user's most recent interaction with it
`Source`	Component that executes the callback
`EventName`	`'ValueChanged'`

For more information about writing callbacks, see "Write Callbacks in App Designer".

`ValueChangingFcn` — Value changing callback
`''` (default) | function handle | cell array | character vector

Value changing callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

This callback function executes as the user clicks and holds the up or down arrow on the spinner. It does not execute if the `Value` property changes programmatically.

This callback function can access specific information about the user's interaction with the spinner. MATLAB passes this information in a `ValueChangingData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.Value` returns the current value of the spinner. The `ValueChangingData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangingData` object.

Property	Value
`Value`	Current value of the spinner as the app user is interacting with it
`Source`	Component that executes the callback
`EventName`	`'ValueChanging'`

The `Value` property of the `Spinner` is not updated until the app user releases the arrow key. Therefore, to get the values while the arrow key is being pressed, your code must get the `Value` property of the `ValueChangingData` object.

The callback executes as follows:

- If the app user clicks a spinner up or down arrow, the callback executes once. For example, suppose that the spinner value is 2, and the `Step` value is 1. If the app user clicks the up arrow, the callback executes.
- If the app user presses and holds a spinner up or down arrow, the callback executes repeatedly. For example, if the app user clicks and holds the up arrow, the callback executes multiple times until the app user releases the up arrow.

For more information about writing callbacks, see “Write Callbacks in App Designer”.

`Position` — Location and size of spinner
`[100 100 100 22]` (default) | `[left bottom width height]`

Location and size of spinner relative to the parent container, specified as the vector `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the spinner
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the spinner
`width`	Distance between the right and left outer edges of the spinner
`height`	Distance between the top and bottom outer edges of the spinner

All measurements are in pixel units.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[100 100 100 22]`

See Also

Functions
`appdesigner` | `uifigure`

Properties
Spinner

Introduced in R2016a

uiswitch

Create slider switch, rocker switch, or toggle switch component

Syntax

```
sw = uiswitch
sw = uiswitch(style)
sw = uiswitch(parent)
sw = uiswitch(parent,style)
sw = uiswitch( ___ ,Name,Value)
```

Description

`sw = uiswitch` creates a slider switch in a new figure window and returns the `Switch` object. MATLAB calls the `uifigure` function to create the figure.

`sw = uiswitch(style)` creates a switch of the specified style.

`sw = uiswitch(parent)` creates the switch in the specified parent container. The parent can be a `Figure` created using the `uifigure` function, or one of its child containers.

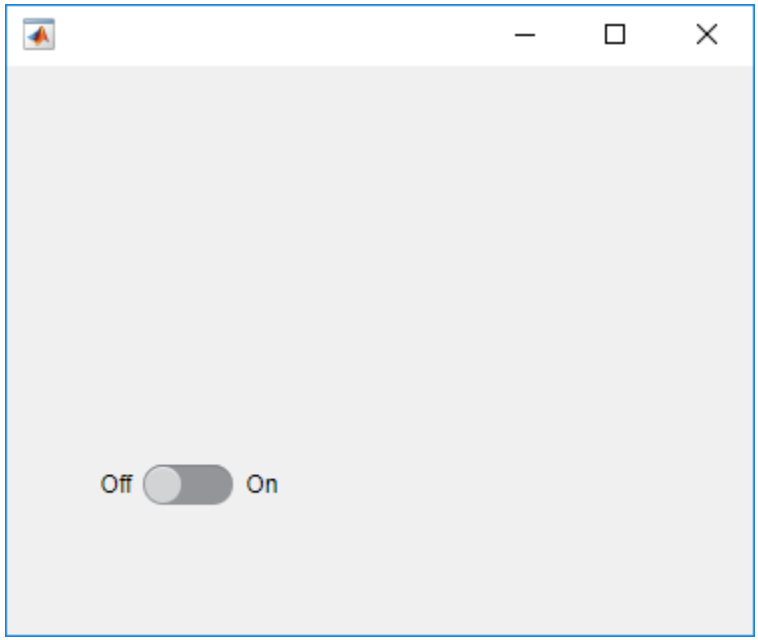
`sw = uiswitch(parent,style)` creates a switch of the specified style in the specified parent container.

`sw = uiswitch( ___ ,Name,Value)` specifies object properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

Examples

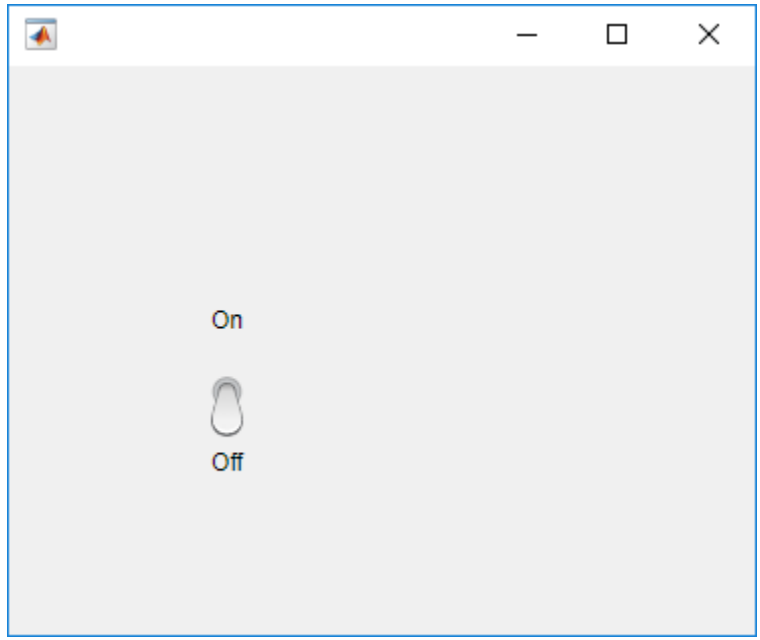
Create Slider Switch in Figure Window

```
fig = uifigure;
sliderswitch = uiswitch(fig);
```



Create Toggle Switch in Figure Window

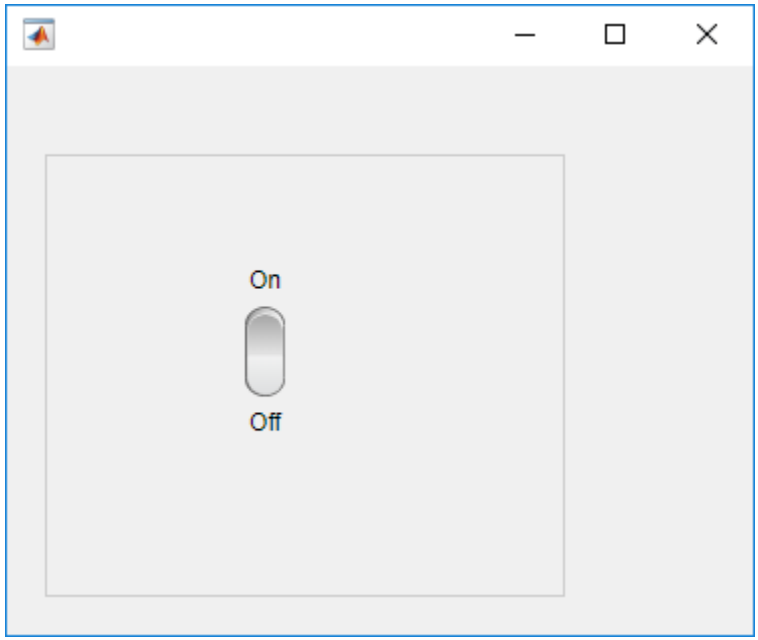
```
fig = uifigure;
toggleswitch = uiswitch(fig,'toggle');
```



Create a Rocker Switch in a Panel

Create a rocker switch in a panel.

```
fig = uifigure;
pnl = uipanel(fig);
rockerswitch = uiswitch(pnl,'rocker');
```



Set and Access Switch Property Values

Create a rocker switch.

```
fig = uifigure;
rockerswitch = uiswitch(fig,'rocker');
```

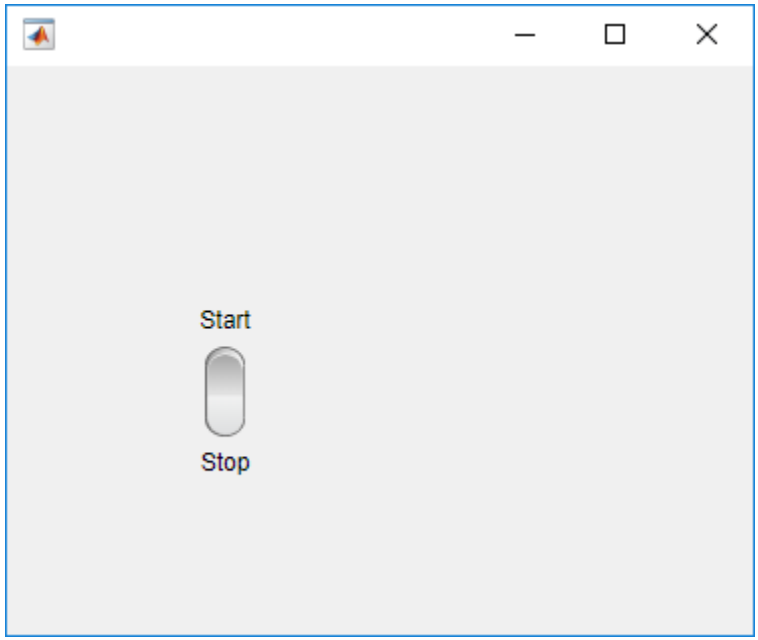
Change the switch text.

```
rockerswitch.Items = {'Stop','Start'};
```

Determine the current switch value.

```
val = rockerswitch.Value
```

```
val =

    'Stop'
```



Switch That Changes Lamp Color

Save the following code as `lampswitch.m` on your MATLAB path. This code creates an app containing a lamp and a rocker switch. When the user flips the switch, the `ValueChangedFcn` callback changes the lamp color.

```
function lampswitch
fig = uifigure('Position',[100 100 370 280]);


lmp = uilamp(fig,...
    'Position',[165 75 20 20],...
    'Color','green');


sw = uiswitch(fig,'toggle',...
    'Items',{'Go','Stop'},...
    'Position',[165 160 20 45],...
    'ValueChangedFcn',@switchMoved);
```

```
% ValueChangedFcn callback
function switchMoved(src,event)
    switch src.Value
        case 'Go'
            lmp.Color = 'green';
        case 'Stop'
            lmp.Color = 'red';
        end
    end
end
```

Run `lampswitch`, and click the switch to see the color change.

Input Arguments

`style` — Style of switch
`'slider'` (default) | `'rocker'` | `'toggle'`

Style of switch, specified as a value from the following table:

Style	Appearance
`'slider'`	
`'rocker'`	
`'toggle'`	

`parent` — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If you do not specify a parent container, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Text',{'0','1'}` specifies the two switch states are “0” and “1”.

Each type of switch supports a different set of properties. For a full list of properties and descriptions for each type, see the associated property page.

- Switch

- RockerSwitch
- ToggleSwitch

See Also

Functions
`appdesigner` | `uifigure`

Properties
RockerSwitch | Switch | ToggleSwitch

Introduced in R2016a

uitextarea

Create text area component

Syntax

```
txa = uitextarea
txa = uitextarea(parent)
txa = uitextarea( ___ ,Name,Value)
```

Description

`txa = uitextarea` creates a text area in a new figure window and returns the `TextArea` object. MATLAB calls the `uifigure` function to create the figure.

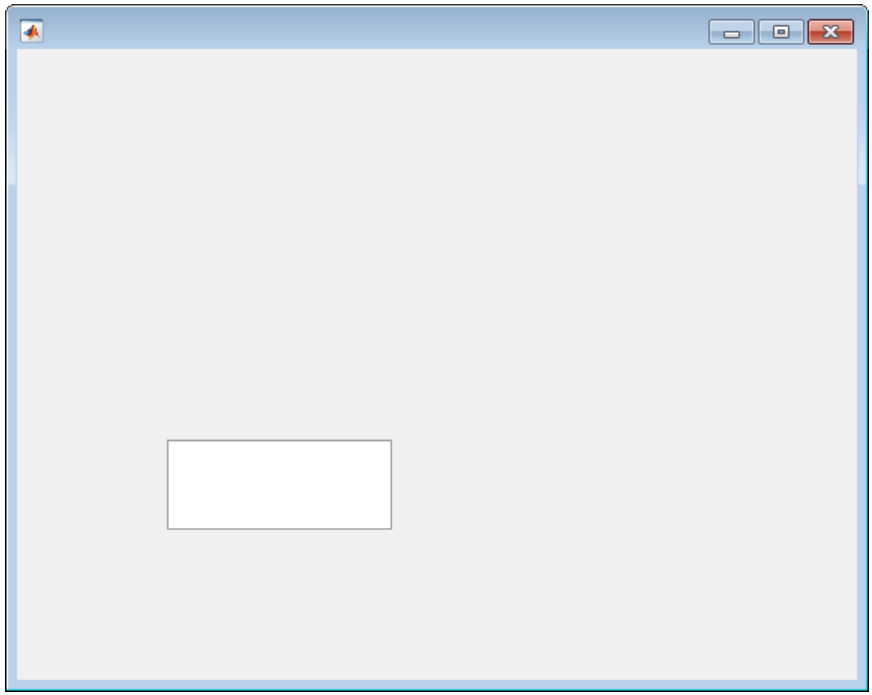
`txa = uitextarea(parent)` creates the text area in the specified parent container. The parent can be a `Figure` created using the `uifigure` function, or one of its child containers.

`txa = uitextarea( ___ ,Name,Value)` specifies `TextArea` properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

Examples

Create Text Area in Figure Window

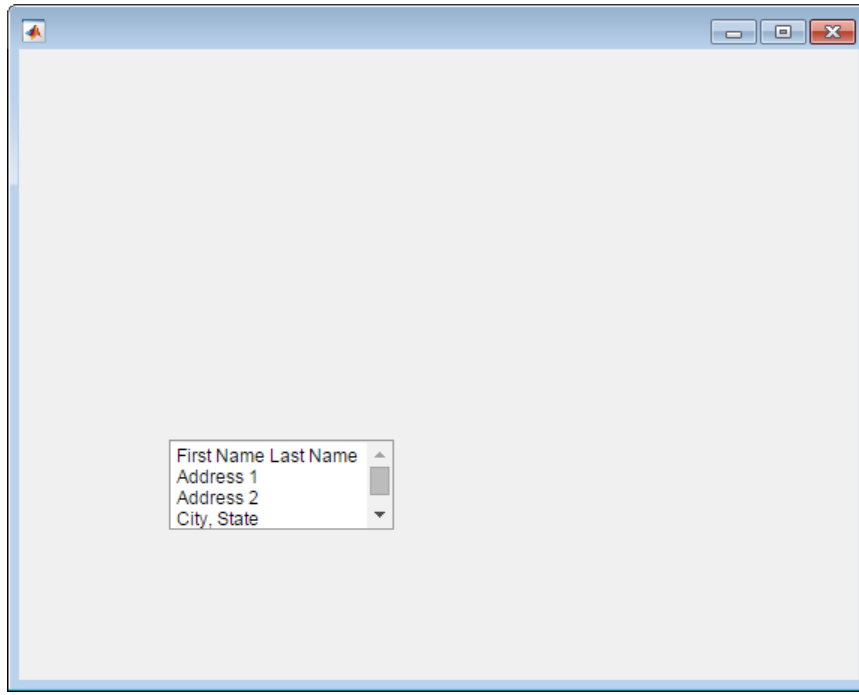
```
fig = uifigure;
txa = uitextarea(fig);
```



Set and Access Text Area Properties

Create a populated text area.

```
fig = uifigure;
txa = uitextarea(fig,...
                'Value', {'First Name Last Name';...
                'Address 1'; 'Address 2';'City, State'; 'Postal Code'});
```



Notice that the text area includes a scroll bar so that the app user can view the postal code.

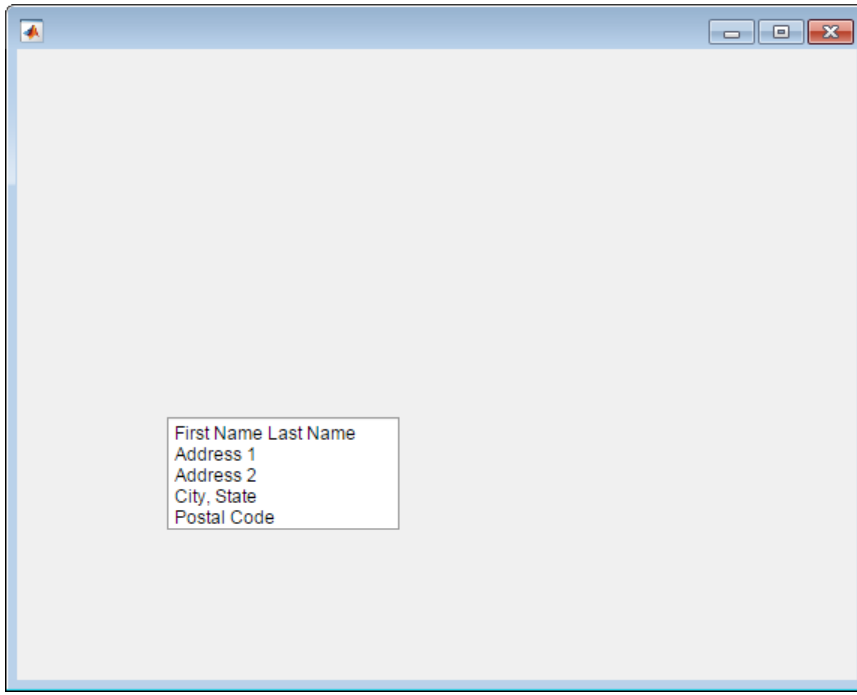
Determine the current size of the text area.

```
size = txa.Position(3:4)
```

```
size =

   150    60
```

Increase the text area size so that the postal code displays without the use of a scroll bar.

```
txa.Position(3:4) = [155 75];
```



Code Response to Changed Text Area Value

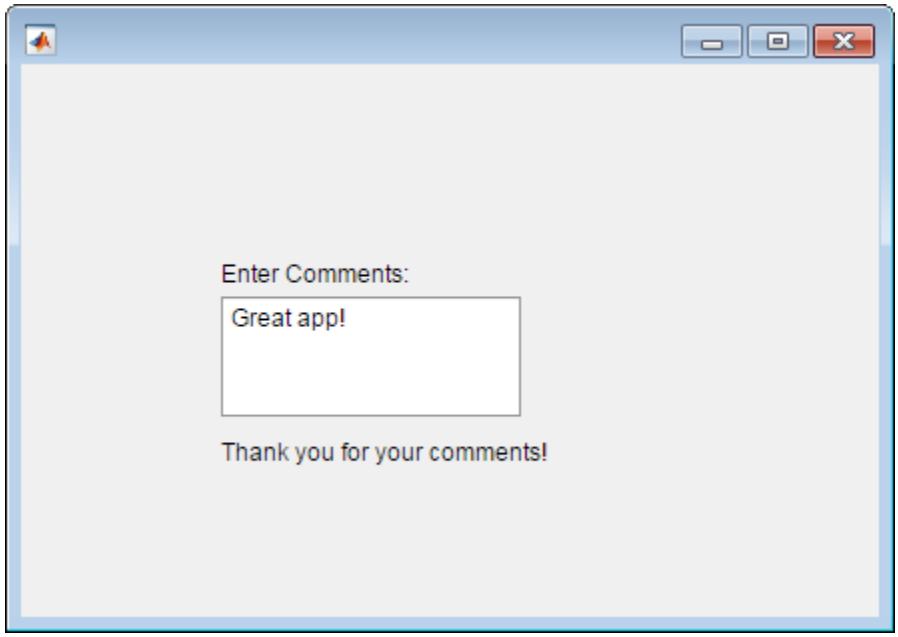
Create a text area and two labels. When an app user types text and clicks outside the text area, a label thanks the app user for the input. If the app user removes the text and clicks outside the text area, the thank you text is removed.

Save the following code to `comments.m` on your MATLAB path. This code creates a figure window containing two labels and a text area. When an app user types text and clicks outside the text area, the `ValueChangedFcn` callback updates one of the labels to thank the user.

```
function comments
% Create figure window and components

fig = uifigure('Position',[100 100 429 276]);

label1 = uilabel(fig,...
```

```
    'Position',[100 164 100 15],...
    'Text','Enter Comments:');

label2 = uilabel(fig,...
    'Position',[100 75 175 15],...
    'Text','');

textarea = uitextarea(fig,...
    'Position',[100 100 150 60],...
    'ValueChangedFcn',@(textarea,event) textEntered(textarea, label2));

% Create ValueChangedFcn callback
    function textEntered(textarea,label2)
        val = textarea.Value;
        label2.Text = '';
        % Check each element of text area cell array for text
        for k = 1:length(val)
            if(~isempty(val{k}))
                label2.Text = 'Thank you for your comments!';
                break;
            end
        end
    end
end
```

Run `comments`, and type text in the text area field. Click outside the text area to trigger the callback.



Input Arguments

parent — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If you do not specify a parent container, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The properties listed here are a subset of the available properties. For the full list, see .TextArea.

Example: `'Editable','off'` specifies that the app user cannot change the text area text.

`Value` — Value

`{''}` (default) | character vector | cell array of character vectors | string array | 1-D categorical array

Value, specified as a character vector, cell array of character vectors, string array, or 1-D categorical array. MATLAB can properly render formatted text, such as this:

```
cellArrayText{1} = sprintf('%s\n%s', 'Line 1', 'Line 2')
cellArrayText{2} = sprintf('%s\n%s', 'Line 3', 'Line 4')
textarea = uitextarea('Value',cellArrayText);
```

If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

If the text does not fit into the width of the text area, MATLAB wraps the text.

If there are too many rows to display in the text area, MATLAB adds a scroll bar.

Example: {'Joseph Welford'; 'Mary Reilly'; 'Roberta Silberlicht'}

`ValueChangedFcn` — Value changed callback

`''` (default) | function handle | cell array | character vector

Value changed callback, specified as one of these values:

- A function handle.
- A cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- A character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

The callback executes when the user changes the text and either presses **Tab** or clicks outside the text area. It does not execute if the `Value` property changes programmatically.

This callback function can access specific information about the user's interaction with the text area. MATLAB passes this information in a `ValueChangedData` object as the second argument to your callback function. In App Designer, the argument is called `event`. You can query the object properties using dot notation. For example, `event.PreviousValue` returns the previous value of the text area. The `ValueChangedData` object is not available to callback functions specified as character vectors.

The following table lists the properties of the `ValueChangedData` object.

Property	Value
`Value`	Value of text area after app user's most recent interaction with it
`PreviousValue`	Value of text area before app user's most recent interaction with it
`Source`	Component that executes the callback
`EventName`	`'ValueChanged'`

For more information about writing callbacks, see "Write Callbacks in App Designer".

`Position` — Location and size of text area
`[100 100 150 60]` (default) | `[left bottom width height]`

Location and size of the text area relative to the parent, specified as the vector `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of the text area
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of the text area
`width`	Distance between the right and left outer edges of the text area

Element	Description
`height`	Distance between the top and bottom outer edges of the text area

All measurements are in pixel units.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[100 100 100 90]`

See Also

Functions
`appdesigner` | `uifigure`

Properties
TextArea

Introduced in R2016a

uitogglebutton

Create toggle button component

Syntax

```
tb = uitogglebutton
tb = uitogglebutton(parent)
tb = uitogglebutton( ___ ,Name,Value)
```

Description

`tb = uitogglebutton` creates toggle button within a button group and returns the `ToggleButton` object. MATLAB calls the `uifigure` function to create the parent figure of the button group.

`tb = uitogglebutton(parent)` creates the toggle button within the specified button group. The button group must be the child of a `Figure` created with the `uifigure` function, or must be parented to a child container of the figure: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`.

`tb = uitogglebutton( ___ ,Name,Value)` specifies `ToggleButton` properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

Examples

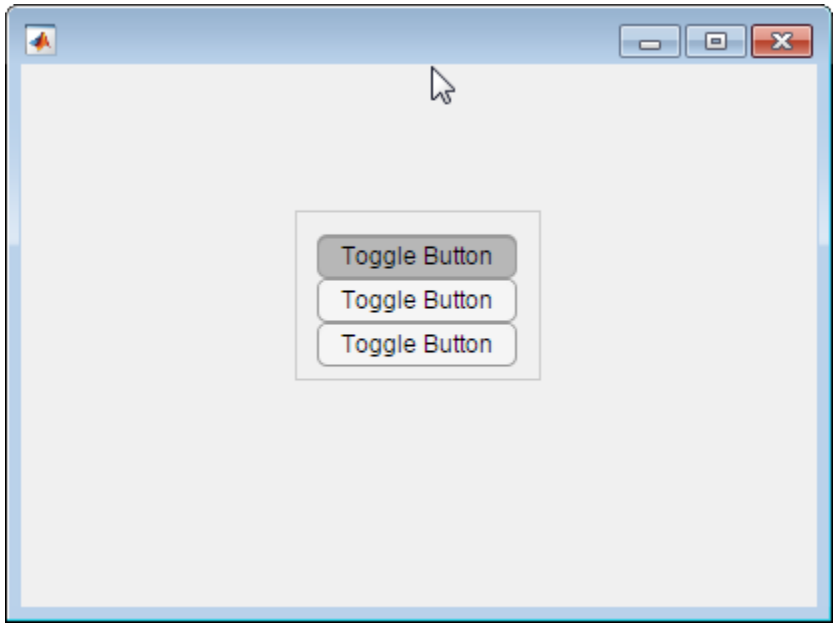
Create Toggle Buttons, Set and Access Property Values

Create toggle buttons by first creating a figure window and a button group to contain the buttons.

```
fig = uifigure('Position',[680 678 398 271]);
bg = uibuttongroup(fig,'Position',[137 113 123 85]);
```

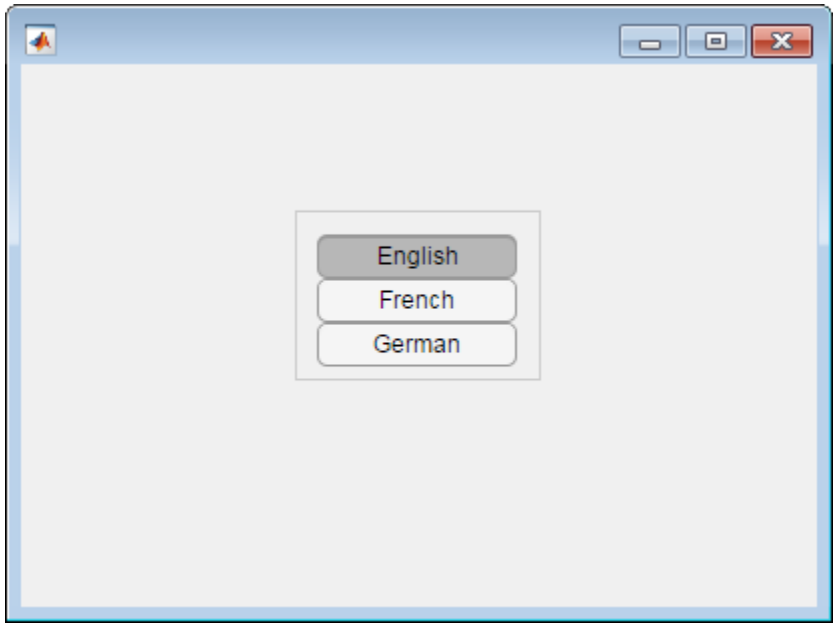
Create three toggle buttons and specify the location of each.

```
tb1 = uitogglebutton(bg,'Position',[10 50 100 22]);
tb2 = uitogglebutton(bg,'Position',[10 28 100 22]);
tb3 = uitogglebutton(bg,'Position',[10 6 100 22]);
```



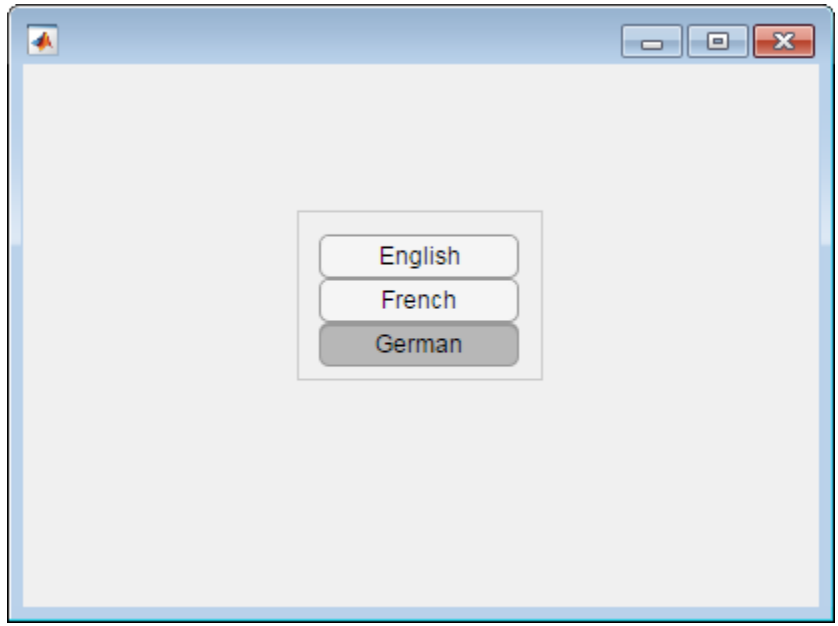
Change the text associated with each toggle button.

```
tb1.Text = 'English';
tb2.Text = 'French';
tb3.Text = 'German';
```



Change the toggle button selection to German programmatically.

```
tb3.Value = true;
```



Determine the font name of the German toggle button text.

```
font = tb3.FontName
```

```
font =

Helvetica
```

Input Arguments

parent — Parent container
ButtonGroup object

Parent container, specified as a `ButtonGroup` object. The `ButtonGroup` must be parented to a `Figure` created using the `uifigure` function, or to a child container of a `uifigure`, such as: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The properties listed here are a subset of the available properties. For the full list, see ToggleButton.

Example: `'Text', 'French'` specifies that the text “French” displays on the toggle button.

`Value` — State of toggle button
1 | 0

State of the toggle button specified as `0` (unpressed) or `1` (depressed). Within a given button group, only one toggle button can be selected (depressed) at a time. When the `Value` property is set to `1`, the toggle button appears depressed. The state of the first button added to a button group is `1`, by default. Subsequent buttons added to the same button group have a default state of `0`.

When the `Value` property of a `ToggleButton` changes to `1`, the `Value` property of the previously selected `ToggleButton` changes to `0`. In addition, the `SelectedObject` property value of the `ButtonGroup` is updated.

If you programmatically change the `Value` property of a `ToggleButton` to `0`, MATLAB sets the `Value` property of the first `ToggleButton` *added* to the `ButtonGroup` to `1`. If the first `ToggleButton` added is the one for which you programmatically set the `Value` property to `0`, then MATLAB sets the `Value` property for the `ToggleButton` added to the `ButtonGroup` to `1`.

Note The first `ToggleButton` *added* to a `ButtonGroup` is not necessarily the first `ToggleButton` *listed* in the `Children` property of the `ButtonGroup`.

`Text` — Button label
`'Toggle Button'` (default) | character vector | cell array of character vectors

Button label, specified as a character vector, cell array of character vectors, string scalar, string array, or 1-D categorical array. Specify a character vector or string scalar to label

the button with a single line of text. Use a cell array or string array to label the button with multiple lines of text. Each element in the array represents a separate line of text. If you specify this property as a categorical array, MATLAB uses the values in the array, not the full set of categories.

`Icon` — File name of button icon

`''` (default) | character vector | string scalar

File name of the button icon, specified as a character vector or string scalar.

The file name can be an image file name on the MATLAB path or a full path to an image file. If you plan to share an app with others, put the image file on the MATLAB path to facilitate app packaging.

The image file type must be JPEG, GIF, or PNG.

- If the button text takes up all the space specified by the `Position` property value, then MATLAB does not display the icon.
- If some room is available for the icon, then MATLAB scales down the image to fit, if necessary.

Example: `'icon.png'`

Example: `'C:\Documents\icon.png'`

`Position` — Location and size of button

`[10 40 100 22]` (default) | `[left bottom width height]`

Location and size button, specified as a vector of the form `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the button group to the outer left edge of the button
`bottom`	Distance from the inner bottom edge of the button group to the outer bottom edge of the button
`width`	Distance between the right and left outer edges of the button
`height`	Distance between the top and bottom outer edges of the button

The `Position` values are relative to the drawable area of the button group. The drawable area is the area inside the borders of the button group and does not include the area occupied by the title.

All measurements are in pixel units.

Tips

- Button groups can contain any UI component type, but can only manage the selection of radio buttons and toggle buttons.
- To make your program respond when the app user selects a radio button or toggle button that is inside a button group. define a `SelectionChangedFcn` callback function for the `ButtonGroup` object. You cannot define callbacks for the individual buttons.
- To determine which radio button or toggle button is selected, query the `SelectedObject` property of the `ButtonGroup` object. You can execute this query anywhere in your code.
- If you set the `Visible` property of a `ButtonGroup` object to `'off'`, then any child objects it contains become invisible along with the parent `ButtonGroup`. However, the `Visible` *property value* of each child object remains unaffected.

See Also

Functions
`appdesigner` | `uibuttongroup` | `uifigure`

Properties
ToggleButton

Introduced in R2016a

upgradePreviouslyInstalledSupportPackages

Get previously installed hardware support packages and optional features for the currently installed release

Syntax

```
upgradePreviouslyInstalledSupportPackages
```

Description

`upgradePreviouslyInstalledSupportPackages` checks for hardware support packages and optional features you installed in prior MATLAB releases. If upgraded versions are available, you can install them for this release.

Examples

Get Previously Installed Support Packages

```
upgradePreviouslyInstalledSupportPackages
```

If MATLAB doesn't already have your credentials, you are prompted to sign in to your MathWorks Account. This command then launches the Add-Ons Manager, where you can select all or only specific add-ons to upgrade.

Introduced in R2017b

matlab.System class

Base class for System objects

Description

`matlab.System` is the base class for System objects. In your class definition file, you must subclass your object from this base class (or from another class that derives from this base class). Subclassing allows you to use the implementation methods and service methods provided by this base class to build your object. Type this syntax as the first line of your class definition file to directly inherit from the `matlab.System` base class, where `ObjectName` is the name of your object:

```
classdef ObjectName < matlab.System
```

Note You must set `Access = protected` for each `matlab.System` method you use in your code.

The `matlab.System` class is a `handle` class.

Class Attributes

`Abstract`	`true`
`HandleCompatible`	`true`
`StrictDefaults`	`false`

For information on class attributes, see “Class Attributes”.

Methods

Specialized Operators and Functions

These methods specialize standard MATLAB operators and functions for objects of this class.

`nargin`	Number of input arguments
`nargout`	Number of output arguments

Protected Methods

<subgroup> Initialize, Run, Reset </subgroup>
<subgroup> Properties and States </subgroup>
<subgroup> Input and Output </subgroup>
<subgroup> Load and Save </subgroup>
<subgroup> MATLAB System Block Support </subgroup>
<subgroup> Simulink Nondirect Feedthrough </subgroup>
<subgroup> Simulink Propagation </subgroup>
<subgroup> MATLAB System Block Appearance Customization </subgroup>
<subgroup> Simulink Sample Time </subgroup>
<subgroup> Simulink Dataflow </subgroup>

Examples

Create Basic System Object

This example shows how to author a basic System object called `AddOne`.

In MATLAB, select **New > System object > Basic**. A new editor window opens with default syntax and comments for a new System object.

Rename the class `AddOne` and modify the default code so your class looks like this:

```
classdef AddOne < matlab.System
% ADDONE Compute an output value that increments the input by one

    methods (Access = protected)
       % Implement algorithm. Calculate y as a function of input x.
       function y = stepImpl(~,x)
          y = x + 1;
       end
    end
end
```

Use this object by creating an instance of `AddOne` and running the object with input.

```
adder = AddOne;
x = 1;
adder(x)
```

```
ans = 2
```

More About

Attributes

You can apply attributes to the System object™ class and properties. To learn more about attributes, see “Class Attributes” or “Property Attributes”.

Class Attribute

This table shows attributes that you can apply to the MATLAB System object class.

Attribute Name	Description
`StrictDefaults`	Control the defaults for the methods that restrict specification modifications changes: • `isInputSizeMutableImpl` • `isInputComplexityMutableImpl` • `isInputDataTypeMutableImpl` • `isTunablePropertyDataTypeMutableImpl` • `isDiscreteStateDataTypeMutableImpl` By default, these methods return `true`. When you add this class attribute, these methods return `false` by default. By making these methods return `false`, the specified aspects of the inputs, tunable properties, or discrete states cannot change. You can always implement these methods individually. Customized methods take precedence over the `StrictDefaults` attribute. For System objects used in Simulink, this attribute only restricts input size changes because Simulink already restricts complexity and data type for tunable properties, inputs, and states.

Specify the class attribute value in parentheses followed by the class name, for example:

```
classdef (StrictDefaults) MySystemObject < matlab.System
```

Property Attributes

You can apply the following attributes to any property of a custom System object.

`Nontunable`	Use `Nontunable` to prevent changes to a property value while the object is in use. By default, all properties are tunable. The `Nontunable` attribute is useful to lock down a property that has side effects when changed. This attribute is also useful for locking a property value assumed to be constant during processing. You should always specify properties that affect the number of input or output ports as `Nontunable`.
`DiscreteState`	Use `DiscreteState` to mark a property so it will display its state value when you use the `getDiscreteState` method.

See Also

`matlab.system.mixin.FiniteSource`

Topics

"Classes"
Class Attributes
Property Attributes
"Method Attributes"
"Define Basic System Objects"
"Define Property Attributes"

Introduced in R2011b

matlab.System Constructor

Construct System object with `Name,Value` pairs or value-only inputs

Syntax

```
function obj = ObjectName(varargin)
```

Description

`function obj = ObjectName(varargin)` constructs an *`ObjectName`* System object and sets properties from name-value pair inputs.

The System object constructor is a public method in the class file. The method name matches the class name. When you create a System object, the constructor is called to initialize properties to nondefault values. The constructor returns a new System object.

Constructor Body

Inside the constructor, call `setProperties` using one of the syntaxes in the table.

Input Arguments	Constructor Body	Example
`Name,Value` pairs	`setProperties(obj, nargin, varargin{:})`	“Constructor for Name-Value Pairs” on page 1-18071
`Name,Value` pairs and value-only arguments	`setProperties(obj, nargin, varargin{:}, '`*`Prop1`*`', ..., '`*`PropN`*`')`	“Constructor for Value-Only Property” on page 1-18071

When you call the System object, properties are specified as comma-separated pairs of `Name,Value` arguments or, if specified, `Value`-only arguments. `Name` is the property name and `Value` is the corresponding value. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Value-only arguments are useful for properties that are frequently set to nondefault values. For example, System objects that read a file often require the file name property to be reset.

Tip: Within the body of the constructor, do not assign property values. This practice can cause problems if you use the System object in multiple environments (such as in a System block, in a MATLAB script, and in generated code). Instead, use default property values or change values inside `setupImpl`.

Examples

Constructor for Name-Value Pairs

Define a System object constructor that allows name-value pair input arguments.

Define a constructor for name-value pair inputs.

```
function obj = Counter(varargin)
    % Support name-value pair arguments when constructing object
    setProperties(obj,nargin,varargin{:})
end
```

With this constructor body, create a `Counter` object using name-value pairs.

```
myObj = Counter('StartValue',0,'UseIncrement',true);
```

Constructor for Value-Only Property

Define a System object constructor with a value-only input property.

Define a constructor with `'StartValue'` as a value-only property input. This constructor also allows name-value inputs.

```
function obj = Counter(varargin)
    % Support value-only argument for StartValue when instantiating
    setProperties(obj,nargin,varargin{:},'StartValue');
end
```

With this constructor body, create a `Counter` object using a value-only argument for `StartValue` and name-value pairs for other properties.

```
myObj = Counter(0,'UseIncrement',true);
```

See Also

setProperties | setupImpl | nargin | varargin

Introduced in R2010a

allowModelReferenceDiscreteSampleTimeInheritanceImpl

Class: `matlab.System`

Model reference sample time inheritance status for discrete sample times

Syntax

```
flag = allowModelReferenceDiscreteSampleTimeInheritanceImpl(obj)
```

Description

`flag = allowModelReferenceDiscreteSampleTimeInheritanceImpl(obj)` specifies whether a System object in a referenced model is allowed to inherit the sample time of the parent model. Use this method only for System objects that use discrete sample time and are intended for inclusion in Simulink via the MATLAB System block.

Run-Time Details

During model compilation, Simulink sets the referenced model sample time inheritance before the System object `setupImpl` method is called.

Method Authoring Tips

You must set `Access = protected` for this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `allowModelReferenceDiscreteSampleTimeInheritanceImpl` method does not use the object, you can replace this input with ~.

Output Arguments

`flag` — discrete sample time inheritance flag
`true` | `false`

Flag indicating whether model reference discrete sample time inheritance is allowed for the MATLAB System block containing the System object, returned as a logical value.

The default value for this argument depends on the number of inputs to the System object. If you want to use the default value, you do not need to include this method in your System object class definition file.

Number of System object Inputs	Default Value and Override Effects
No inputs	**Default:** `false` — Model reference discrete sample time inheritance is not allowed. If your System object uses discrete sample time in its algorithm, override the default by returning `true` from `allowModelReferenceDiscreteSampleTimeInheritanceImpl`.
One or more inputs	**Default:** `true` — If no other Simulink constraint prevents it, model reference sample time inheritance is allowed.

Examples

Set Sample Time Inheritance for System Object

For a System object that has one or more inputs, to disallow model reference discrete sample time inheritance for that object, set the sample time inheritance to `false`. Include this code in your class definition file for the object.

```
methods (Access = protected)
   function flag = allowModelReferenceDiscreteSampleTimeInheritanceImpl(~)
      flag = false;
   end
end
```

See Also

`matlab.System`

Topics

"Set Model Reference Discrete Sample Time Inheritance" (Simulink)
"Model Reference Basics" (Simulink)
"Referenced Model Sample Times" (Simulink)

Introduced in R2016a

createSampleTime

Class: `matlab.System`

Create sample time specification object

Syntax

```
sts = createSampleTime(obj)
sts = createSampleTime(obj,'Type',Type)
sts = createSampleTime(obj,'Type',Type,Name,Value)
```

Description

`sts = createSampleTime(obj)` creates a sample time specification object for inherited sample time for the System object `obj`. Use this sample time specification object in the `getSampleTimeImpl` method of your System object. The sample time specification affects the simulation time when the System object is included in a MATLAB System block. If you do not include this method in your System object definition, the sample time is inherited.

`sts = createSampleTime(obj,'Type',Type)` creates a sample time specification object with the specified sample time type.

`sts = createSampleTime(obj,'Type',Type,Name,Value)` creates a sample time specification object with additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Dependencies

You can only call `createSampleTime` from the `getSampleTimeImpl` method of your System object.

Input Arguments

`obj` — System object
System object

System object that you want to specify the sample time.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Type','Fixed In Minor Step'`

Type — Sample time type
`'Inherited'` (default) | `'Controllable'` | `'Discrete'` | `'Fixed In Minor Step'`

Type of sample time you want the System object to use in Simulink. For descriptions of the different types of sample times, see:

- `'Inherited'` - “Inherited Sample Time” (Simulink)
- `'Controllable'` - “Controllable Sample Time” (Simulink)
- `'Discrete'` - “Discrete Sample Time” (Simulink)
- `'Fixed In Minor Step'` - “Fixed-in-Minor-Step” (Simulink)

Example: `createSampleTime('Type','Fixed In Minor Step')`

`AllowPropagation` — Allow the propagated sample time
`'Controllable'` | `'Constant'` | `'Continuous'` | cell array of character vectors | string array

For `Inherited` sample time only.

Specify one or more sample times you want to allow your object to use. When the sample time type is set to `Inherited` and the MATLAB System block receives the specified sample time, the block allows the sample time to propagate. You can specify multiple sample time types as a cell array or string array.

Example: `createSampleTime('Type','Inherited','AllowPropagation',{'Controllable','Continuous'})`

`AlternatePropagation` — Alter the propagated sample time
`'Controllable'` | `'Constant'` | `'Continuous'` | cell array of character vectors | string array

For `Inherited` sample time only.

Specify one or more sample times you do not want your object to use. When the sample time type is set to `Inherited` and you do not want the MATLAB System block to use the specified inherited sample time type, an alternative sample time is used by the block instead. You can specify multiple sample time types. For each sample time type, Simulink substitutes a different sample time type:

- `'Controllable'` — The base rate of the controlled sample is propagated instead. Use this option if your System object depends on having constant time between each sample-time hit.
- `'Constant'` — The base rate of the model is propagated instead, either the fastest discrete rate or fixed in minor step.
- `'Continuous'` — The fastest discrete rate is propagated instead.

Example: `createSampleTime('Type','Inherited','AlternatePropagation','Controllable')`

`ErrorOnPropagation` — Error on the propagated sample time
`'Controllable'` | `'Constant'` | `'Continuous'` | cell array of character vectors | string array

For `Inherited` sample time only.

Specify one or more sample times you do not want your object to use. When the sample time type is set to `Inherited` and the MATLAB System block receives the specified sample time, Simulink throws an error.

Example: `createSampleTime('Type','Inherited','ErrorOnPropagation','Controllable')`

`SampleTime` — Time between samples
1 (default) | positive number

For `Discrete` sample time only.

Specify the time between sample hits in Simulink.

Example: `createSampleTime('Type','Discrete','SampleTime',1)`

Data Types: `single` | `double`

`OffsetTime` — Offset from sample time
`0` (default) | nonnegative number less than `SampleTime`

For `Discrete` sample time only.

Specify the offset time for the sample hit. The offset is a time interval indicating an update delay. The block is updated later in the sample interval than other blocks operating at the same sample rate.

The offset time must be nonnegative and less than `SampleTime`.

Example:
`createSampleTime('Type','Discrete','SampleTime',2,'OffsetTime',1)`

Data Types: `single` | `double`

`TickTime` — Time between sample time hits
-1 (default) | positive scalar

Required for `Controllable` sample time only.

Specify the time between controllable sample time hits. The tick time must be a positive scalar.

Example:
`createSampleTime('Type','Controllable','TickTime',obj.TickTime)`

Data Types: `single` | `double`

Output Arguments

`sts` — Sample time specification object
sample time specification object

The sample time specification object. This object has the following properties:

- Type — Type of sample time
- SampleTime — Time between samples
- OffsetTime — Offset from sample time

Use this object as the return value of `getSampleTimeImpl`.

Examples

Create Inherited Sample Time Specification Object

Specify that the MATLAB System block inherits the sample from upstream blocks. Inherited sample time is the default, so no additional arguments are needed.

```
function sts = getSampleTimeImpl(obj)
    sts = createSampleTime(obj);
end
```

Create Discrete Sample Time Object

Specify a discrete sample time specification for the MATLAB System block, including offset time and the sample time.

```
function sts = getSampleTimeImpl(obj)
    sts = createSampleTime(obj,'Type','Discrete',...
      'SampleTime',10.2,'OffsetTime',0.5);
end
```

See Also

`getCurrentTime` | `getSampleTime` | `getSampleTimeImpl` | `setNumTicksUntilNextHit`

Topics

"Specify Sample Time for MATLAB System Block System Objects" (Simulink)

Introduced in R2017b

getCurrentTime

Class: `matlab.System`

Current simulation time in MATLAB System block

Syntax

```
time = getCurrentTime(obj)
```

Description

`time = getCurrentTime(obj)` returns the current simulation time in the MATLAB System block. Call this method in the `stepImpl` method of your System object.

Dependencies

If the MATLAB System block is operating in continuous sample time, `getCurrentTime` may return non-monotonic times due to solver operation.

Input Arguments

`obj` — System object
system object

System object included in a MATLAB System block that you want to query.

Output Arguments

`time` — Current simulation time
double

The current simulation time of the MATLAB System block that contains the System object.

Examples

Return Current Simulation Time

This example of `stepImpl` returns a count value `y` and the current simulation time `ct`. The simulation time is obtained by calling `getCurrentTime`.

```
function [y,ct] = stepImpl(obj,u)
    y = obj.Count + u;
    obj.Count = y;
    ct = getCurrentTime(obj);
end
```

For a complete class definition, see “Specify Sample Time for MATLAB System Block System Objects” (Simulink).

See Also

`createSampleTime` | `getSampleTime` | `getSampleTimeImpl`

Topics

“Specify Sample Time for MATLAB System Block System Objects” (Simulink)

Introduced in R2017b

getDiscreteStateImpl

Class: `matlab.System`

Discrete state property values

Syntax

```
s = getDiscreteStateImpl(obj)
```

Description

`s = getDiscreteStateImpl(obj)` returns a struct `s` of internal state value properties, which have the `DiscreteState` attribute. The field names of the struct are the object's `DiscreteState` property names. To restrict or change the values returned by `getDiscreteState` method, you can override this `getDiscreteStateImpl` method.

Run-Time Details

`getDiscreteStateImpl` is called by the `setup` method. For details, see

"Detailed Call Sequence".

Method Authoring Tips

- You must set `Access = protected` for this method.
- You cannot modify any properties in this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `getDiscreteStateImpl` method does not use the object, you can replace this input with ~.

Output Arguments

s — State values
struct

State values, returned as a struct.

Examples

Get Discrete State Values

Use the `getDiscreteStateImpl` method in your class definition file to get the discrete states of the object.

```
methods (Access = protected)
   function s = getDiscreteStateImpl(obj)
   end
end
```

See Also

`setupImpl`

Topics

"Define Property Attributes"

Introduced in R2012b

getGlobalNamesImpl

Class: `matlab.System`

Global variable names for MATLAB System block

Syntax

```
name = getGlobalNamesImpl(obj)
```

Description

`name = getGlobalNamesImpl(obj)` specifies the names of global variables that are declared in a System object for use in a Simulink P-code file. For P-code files, in addition to declaring your global variables in `stepImpl`, `outputImpl`, or `updateImpl`, you must include the `getGlobalNamesImpl` method. You declare global variables in a cell array in the `getGlobalNamesImpl` method. System objects that contain these global variables are included in Simulink using a MATLAB System block. To enable a global variable in Simulink, your model also must include a Data Store Memory block with a **Data Store Name** that matches the global variable name.

Run-Time Details

`getGlobalNamesImpl` is called by the MATLAB System block.

Method Authoring Tips

You must set `Access = protected` for this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `getGlobalNamesImpl` method does not use the object, you can replace this input with ~.

Output Arguments

name — Global variable names
cell array of character vectors

Cell array containing the global variable names. The elements of the cell array are character vectors.

Examples

Specify Global Names

Specify two global names in your class definition file.

```
methods(Access = protected)
    function glnames = getGlobalNamesImpl(obj)
      glnames = {'FEE','OTHERFEE'};
    end

    function y = stepImpl(obj,u)
      global FEE
      global OTHERFEE
      y = u - FEE * obj.lastData + OTHERFEE;
      obj.lastData = u;
    end
end
```

See Also

`outputImpl` | `stepImpl` | `updateImpl`

Topics

"System Object Global Variables in Simulink" (Simulink)

Introduced in R2016b

getHeaderImpl

Class: `matlab.System`

Header for System object display

Syntax

```
header = getHeaderImpl
```

Description

`header = getHeaderImpl` specifies the dialog header to display on the MATLAB System block dialog box. If you do not specify the `getHeaderImpl` method, no title or text appears for the header in the block dialog box.

Run-Time Details

`getHeaderImpl` is called by the MATLAB System block.

Method Authoring Tips

You must set `Access = protected` for this method.

Output Arguments

header — Header text
`matlab.system.display.Header` class

Header text specified as a `matlab.system.display.Header` class.

Examples

Define Header for System Block Dialog Box

Define a header in your class definition file for the `EnhancedCounter` System object.

```
methods (Static, Access = protected)
   function header = getHeaderImpl
      header = matlab.system.display.Header('EnhancedCounter',...
          'Title','Enhanced Counter');
   end
end
```

See Also

`getPropertyGroupsImpl`

Topics

"Add Header to MATLAB System Block" (Simulink)

Introduced in R2013b

getDiscreteStateSpecificationImpl

Class: `matlab.System`

Discrete state size, data type, and complexity

Syntax

```
[size,dataType,complexity] = getDiscreteStateSpecificationImpl(obj,
propertyName)
```

Description

`[size,dataType,complexity] = getDiscreteStateSpecificationImpl(obj, propertyName)` returns the size, data type, and complexity of the discrete state property. This property must be a discrete state property. You must define this method if your System object has discrete state properties and is used in the MATLAB System block.

Run-Time Details

`getDiscreteStateSpecificationImpl` is called by the MATLAB System block during Simulink model compilation.

Method Authoring Tips

- You must set `Access = protected` for this method.
- You cannot modify any properties in this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `getDiscreteStateSpecificationImpl` method does not use the object, you can replace this input with ~.

`propertyName` — Discrete state property name
property name

Name of discrete state property of the System object

Output Arguments

`size` — Property size
`[1 1]` (default) | numeric vector

Vector containing the length of each dimension of the property.

`dataType` — Property data type
`double` (default) | character vector

Data type of the property. For built-in data types, `dataType` is a character vector. For fixed-point data types, `dataType` is a numeric type object.

`complexity` — Complexity of the property
`false` (default) | `true`

Complexity of the property as a scalar, logical value:

- `true` — complex
- `false` — real

Examples

Specify Discrete State Property Size, Data Type, and Complexity

Specify in your class definition file the size, data type, and complexity of a discrete state property.

```
methods (Access = protected)
   function [sz,dt,cp] = getDiscreteStateSpecificationImpl(~,name)
```

```
        sz = [1 1];
        dt = "double";
        cp = false;
    end
end
```

See Also

Topics

"Set Discrete State Output Specification" (Simulink)

Introduced in R2013b

getOutputDataTypeImpl

Class: `matlab.System`

Data types of output ports

Syntax

```
[dt_1,dt_2,...,dt_n] = getOutputDataTypeImpl(obj)
```

Description

`[dt_1,dt_2,...,dt_n] = getOutputDataTypeImpl(obj)` returns the data type of each output port as a character vector for built-in data types or as a numeric object for fixed-point data types. The number of outputs must match the value returned from the `getNumOutputsImpl` method or the number of output arguments listed in the `stepImpl` method.

For System objects with one input and one output and where you want the input and output data types to be the same, you do not need to implement this method. In this case, `getOutputDataTypeImpl` assumes the input and output data types are the same and returns the data type of the input.

If your System object has more than one input or output, and you use propagation, you must set the output data types in the `getOutputDataTypeImpl` method. For Simulink, if the input and output data types are different, you might have to cast the output value to the data type of the appropriate `dt_n` output argument. You specify this casting in the `stepImpl` method. For bus output, you must specify the name of the output bus in `getOutputDataTypeImpl`.

If needed to determine the output data type, you can use `propagatedInputDataType` within the `getOutputDataTypeImpl` method to obtain the input type.

Run-Time Details

`getOutputDataTypeImpl` is called by the MATLAB System block.

Method Authoring Tips

- You must set `Access = protected` for this method.
- You cannot modify any properties in this method.
- If you are debugging your code and examine the data types before Simulink completes propagation, you might see outputs with empty, [ ], data types. This occurs because Simulink has not completed setting the output data types.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `getOutputDataTypeImpl` method does not use the object, you can replace this input with ~.

Output Arguments

`dt_1,dt_2,...` — Property data type
character vector | numeric type object

Data type of the property. For built-in data types, `dt` is a character vector. For fixed-point data types, `dt` is a numeric type object.

Examples

Specify Output Data Type

Specify, in your class definition file how to control the output data type from a MATLAB System block. This example shows how to use the `getOutputDataTypeImpl` method to change the output data type from single to double, or propagate the input as a double. It also shows how to cast the data type to change the output data type in the `stepImpl` method.

```
classdef DataTypeChange < matlab.System & ...
  matlab.system.mixin.Propagates

   properties(Nontunable)
      Quantize = false
   end

   methods(Access = protected)
      function y = stepImpl(obj,u)
         if obj.Quantize == true
            % Cast for output data type to differ from input.
            y = single(u);
         else
            % Propagate output data type.
            y = u;
         end
      end

      function out = getOutputDataTypeImpl(obj)
         if obj.Quantize == true
            out = "single";
         else
            out = propagatedInputDataType(obj,1);
         end
      end
   end
end
```

Specify Bus Output

Specify, in your class definition file, that the System object data type is a bus. You must also include a property to specify the bus name.

```
properties(Nontunable)
   OutputBusName = "myBus";
end

methods (Access = protected)
   function out = getOutputDataTypeImpl(obj)
      out = obj.OutputBusName;
```

```
    end
end
```

See Also

`propagatedInputDataType`

Topics

"Set Output Data Type" (Simulink)

Introduced in R2013b

getOutputSizeImpl

Class: `matlab.System`

Sizes of output ports

Syntax

```
[sz_1,sz_2,...,sz_n] = getOutputSizeImpl(obj)
```

Description

`[sz_1,sz_2,...,sz_n] = getOutputSizeImpl(obj)` returns the size of each output port. The number of outputs must match the value returned from the `getNumOutputs` method or the number of output arguments listed in the `stepImpl` method.

If your System object has only one input and one output and you want the input and output sizes to be the same, you do not need to implement this method. In this case `getOutputSizeImpl` assumes that the input and output sizes are the same and returns the size of the input. For variable-size inputs in MATLAB, the size varies each time you run your object. For variable-size inputs in Simulink, the output size is the maximum input size.

You must implement the `getOutputSizeImpl` method to define the output size, if:

- Your System object has more than one input or output
- You need the output and input sizes to be different.

  If the output size differs from the input size, you must also use the `propagatedInputSize` method

Run-Time Details

During Simulink model compilation and propagation, the MATLAB System block calls the `getOutputSizeImpl` method to determine the output size.

All inputs default to variable-size inputs For these inputs, the output size is the maximum input size.

Method Authoring Tips

- You must set `Access = protected` for this method.
- In this method, you cannot modify any properties.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `getOutputSizeImpl` method does not use the object, you can replace this input with ~.

Output Arguments

`sz_1,sz_2,...` — Output size
numeric vector

Vector containing the size of each output port.

Examples

Specify Output Size

Specify in your class definition file the size of a System object output.

```
methods (Access = protected)
   function sz_1 = getOutputSizeImpl(obj)
      sz_1 = [1 1];
   end
end
```

Specify Multiple Output Ports

Specify in your class definition file the sizes of multiple System object outputs.

```
methods (Access = protected)
   function [sz_1,sz_2] = getOutputSizeImpl(obj)
      sz_1 = propagatedInputSize(obj,1);
      sz_2 = [1 1];
   end
 end
```

Specify Output When Using Propagated Input Size

Specify in your class definition file the size of System object output when it depends on the propagated input size.

```
methods (Access = protected)
   function varargout = getOutputSizeImpl(obj)
      varargout{1} = propagatedInputSize(obj,1);
      if obj.HasSecondOutput
         varargout{2} = [1 1];
      end
   end
end
```

See Also

`propagatedInputSize`

Topics

“Set Output Size” (Simulink)

Introduced in R2013b

getIconImpl

Class: `matlab.System`

Name to display as block icon

Syntax

```
icon = getIconImpl(obj)
```

Description

`icon = getIconImpl(obj)` specifies the text or image to display on the block icon of the MATLAB System block. If you do not specify the `getIconImpl` method, the block displays the class name of the System object as the block icon. For example, if you specify `pkg.MyObject` in the MATLAB System block, the default icon is labeled `MyObject`

Run-Time Details

`getIconImpl` is called by the MATLAB System block during Simulink model compilation.

Method Authoring Tips

You must set `Access = protected` for this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `getIconImpl` method does not use the object, you can replace this input with ~.

Output Arguments

`icon` — Block icon
character vector | cell array of character vectors | string | string array | `matlab.system.display.Icon` object

The text or image to display as the block icon. Each cell is displayed as a separate line.

Examples

Add System Block Icon Name

Specify in your class definition file the name of the block icon as `'Enhanced Counter'` using two lines.

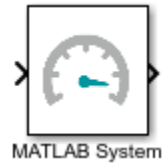
```
methods (Access = protected)
   function icon = getIconImpl(~)
      icon = {'Enhanced','Counter'};
   end
end
```

Add Image to MATLAB System Block

Define an image in your class definition file.

```
    methods(Access = protected)
        function icon = getIconImpl(~)
            % Define icon for System block
            icon = matlab.system.display.Icon('my_icon.png');
        end
    end
```

The image now appears on the System block icon.



See Also

`matlab.system.display.Icon` | `matlab.system.mixin.CustomIcon`

Topics

"Customize System Block Appearance" (Simulink)
"Customize MATLAB System Block Appearance" (Simulink)

Introduced in R2017a

getImpulseResponseLengthImpl

Class: `matlab.System`

Define length of input effects for dataflow subsystems

Syntax

```
impulseLength = getImpulseResponseLengthImpl(obj)
impulseLength = getImpulseResponseLengthImpl(obj,input1,
input2, ...)
```

Description

`impulseLength = getImpulseResponseLengthImpl(obj)` defines the impulse response length of a System object that is independent of input specifications.

`impulseLength = getImpulseResponseLengthImpl(obj,input1, input2, ...)` defines the impulse response length for `obj` as a function of one or more of its input specifications.

Method Authoring Tips

You must set `Access = protected` for this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `getImpulseResponseLengthImpl` method does not use the object, you can replace this input with ~.

`input1, input2, ...` — Inputs to the `stepImpl` method
inputs to the System object

List the inputs to the System object. The order of inputs must match the order of inputs defined in the `stepImpl` method. `stepImpl` passes the inputs into `setupImpl` to use the specifications, such as size and data types in the one-time calculations.

Output Arguments

`impulseLength` — Impulse response length
nonnegative scalar integer | inf

The impulse response length of an object, or number of input samples that affect the output of a block. The impulse response length is used by dataflow to optimize the automatic partitioning of a dataflow subsystem for multithreaded simulation and code generation.

Examples

Get Impulse Response Length

When the impulse response length of a system object is constant, specify the number of input samples that affect the output of the block.

```
function s = getImpulseResponseLengthImpl(obj)
    s = 50;
end
```

Get Impulse Response Length Based on System Object Input Arguments

When inputs to the system object affect the length of the impulse response, specify the inputs in the function signature.

```
function s = getImpulseResponseLengthImpl(obj, X, COEFF)
    if isequal(obj.NumeratorSource, ‘Property’)
        s = numel(obj.Numerator) - 1;
    else
        s = numel(COEFF) - 1;
```

```
    end
end
```

See Also

`getInputDimensionConstraintImpl` | `getOutputDimensionConstraintImpl`

Topics

"Dataflow Domain" (DSP System Toolbox)

Introduced in R2018b

getInputDimensionConstraintImpl

Class: `matlab.System`

Define input dimension constraints for dataflow subsystems

Syntax

```
constraints = getInputDimensionConstraintImpl(obj)
constraints = getInputDimensionConstraintImpl(obj,index)
```

Description

`constraints = getInputDimensionConstraintImpl(obj)` defines the input dimension constraint for the first input.

`constraints = getInputDimensionConstraintImpl(obj,index)` defines the input dimension constraint for `obj` for the `index`th input port.

Method Authoring Tips

- You must set `Access = protected` for this method.
- If you specify the `getInputDimensionConstraintImpl` method, you should also specify the `getOutputDimensionConstraintImpl` method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `getInputDimensionConstraintImpl` method does not use the object, you can replace this input with ~.

`index` — Index of input port
1 (default) | scalar integer

Index of input port, specified as a scalar integer.

Output Arguments

`constraints` — Dimension constraint
dimension constraint object

The input dimension constraint object, used by dataflow to optimize the automatic partitioning of a dataflow subsystem for multithreaded simulation and code generation.

Use the `inputDimensionConstraint` method to specify the following properties of the dimension constraint object.

- `'MinimumSize'` - Minimum number of samples needed to run, specified as a nonnegative scalar integer.
- `'Concatenable'` - Whether the object can accept multiples of the minimum size, specified as a boolean. The default value for `'Concatenable'` is false.

  If `'Concatenable'` is set to true, the software may buffer the inputs to the System object. If `'Concatenable'` is set to false, the software may execute the block several times to process all of the incoming data.

Examples

Specify Input Dimension Constraint

Specify the required input dimension of the first input of a system object.

```
function dc = getInputDimensionConstraintImpl(obj)
    dc = inputDimensionConstraint(obj, 'MinimumSize', obj.NumBins, 'Concatenable', true);
end
```

Specify Input Dimension Constraint for a Specified Port

To specify the input dimension for an input port other than the first input port, specify the port number in the constructor.

```
function dc = getInputDimensionConstraintImpl(obj, index)
    if index == 1
        dc = inputDimensionConstraint(obj, 'MinimumSize', obj.NumBins, 'Concatenable', true);
    else
        dc = inputDimensionConstraint(obj, 'MinimumSize', 1, 'Concatenable', true);
    end
end
```

Default Behavior

If this method is not overwritten, the MATLAB System block sets the input dimension constraint to unspecified. The unspecified setting allows the dataflow scheduler to make conservative assumptions for optimization.

See Also

`getImpulseResponseLengthImpl` | `getOutputDimensionConstraintImpl`

Topics

"Dataflow Domain" (DSP System Toolbox)

Introduced in R2018b

getInputNamesImpl

Class: `matlab.System`

Names of MATLAB System block input ports

Syntax

```
names = getInputNamesImpl(obj)
```

Description

`names = getInputNamesImpl(obj)` specifies the names of the input ports from System object, `obj` implemented in a MATLAB System block. The size of `names` matches the number of inputs returned by the `getNumInputs` method. If you change a property value that changes the number of inputs, the names of those inputs also change.

Class Information

This method is part of the `matlab.System` class.

Run-Time Details

`getInputNamesImpl` is called by the MATLAB System block.

Method Authoring Tips

You must set `Access = protected` for this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `getInputNamesImpl` method does not use the object, you can replace this input with ~.

Output Arguments

`names` — Names of inputs
empty string array (default) | string array

Names of the inputs for the specified object, returned as a string array whose length equals the number of inputs.

Examples

See Also

`getNumInputsImpl` | `getOutputNamesImpl`

Topics

"Specify Input and Output Names" (Simulink)

Introduced in R2013b

getNumInputsImpl

Class: `matlab.System`

Number of inputs to the System object

Syntax

```
num = getNumInputsImpl(obj)
```

Description

`num = getNumInputsImpl(obj)` returns the number of inputs expected by the System object.

If the signature of `stepImpl` or `updateImpl` does not include `varargin`, the System object can determine the number of outputs from the method signature. In this case, you do not need to include `getNumInputsImpl` in your class definition file.

If the signature of `stepImpl` or `updateImpl` does include `varargin`, you can implement the `getNumInputsImpl` method in your class definition file to determine the number of inputs. You can use `nargin` in the `stepImpl` method to get the number of inputs the object was called with.

Method Authoring Tips

- You must set `Access = protected` for this method.
- You cannot modify any properties in this method.
- If you set the return argument, `num`, from an object property, that object property must have the `Nontunable` attribute.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `getNumInputsImpl` method does not use the object, you can replace this input with ~.

Output Arguments

`num` — Number of inputs
1 (default) | integer

Number of inputs expected when running the object, returned as an integer.

Examples

Set Number of Inputs

Specify the number of inputs (two, in this case) expected by the object.

```
methods (Access = protected)
   function num = getNumInputsImpl(~)
      num = 2;
   end
end
```

Set Number of Inputs to Zero

Specify that the object does not accept any inputs.

```
methods (Access = protected)
   function num = getNumInputsImpl(~)
      num = 0;
   end
end
```

See Also

`getNumOutputsImpl` | `setupImpl` | `stepImpl`

Topics

"Change the Number of Inputs"

Introduced in R2011b

getNumOutputsImpl

Class: `matlab.System`

Number of outputs from System object

Syntax

```
num = getNumOutputsImpl(obj)
```

Description

`num = getNumOutputsImpl(obj)` returns the number of outputs expected from the System object.

If the signature of `stepImpl` or `outputImpl` does not include `varargout`, the System object *can* determine the number of outputs from the method signature. In this case, you do not need to implement the `getNumOutputsImpl` method.

If the signature of `stepImpl` or `outputImpl` does include `varargout`, you can implement the `getNumOutputsImpl` method in your class definition file to determine the number of outputs. You can use `nargout` in the `stepImpl` method to get the number of outputs the object was called with.

Method Authoring Tips

- You must set `Access = protected` for this method.
- You cannot modify any properties in this method.
- If you set the return argument, `num`, from an object property, that object property must have the `Nontunable` attribute.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `getNumOutputsImpl` method does not use the object, you can replace this input with ~.

Output Arguments

`num` — Number of outputs
integer

Number of outputs from the specified object, returned as an integer.

Examples

Set Number of Outputs

Specify the number of outputs (2, in this case) returned from the object.

```
methods (Access = protected)
   function num = getNumOutputsImpl(~)
      num = 2;
   end
end
```

Set Number of Outputs to Zero

Specify that the object does not return any outputs.

```
methods (Access = protected)
   function num = getNumOutputsImpl(~)
      num = 0;
   end
end
```

Use `nargout` for Variable Number of Outputs

Use `nargout` in the `stepImpl` method when you have a variable number of outputs and want to generate code.

```
methods (Access = protected)
   function varargout = stepImpl(~,varargin)
      for i = 1:nargout
         varargout{i} = varargin{i}+1;
      end
   end
end
```

See Also

`getNumInputsImpl` | `setupImpl` | `stepImpl`

Topics

"Change the Number of Inputs"

Introduced in R2011b

getOutputDimensionConstraintImpl

Class: `matlab.System`

Define output dimension constraints for dataflow subsystems

Syntax

```
constraints = getOutputDimensionConstraintImpl(obj)
constraints = getOutputDimensionConstraintImpl(obj,index)
```

Description

`constraints = getOutputDimensionConstraintImpl(obj)` defines the output dimension constraint for the first output.

`constraints = getOutputDimensionConstraintImpl(obj,index)` defines the output dimension constraint for the `index`th output port.

Method Authoring Tips

- You must set `Access = protected` for this method.
- If you specify the `getOutputDimensionConstraintImpl` method, you should also specify the `getInputDimensionConstraintImpl` method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `getOutputDimensionConstraintImpl` method does not use the object, you can replace this input with ~.

`index` — Index of output port
1 (default) | scalar integer

Index of output port, specified as a scalar integer.

Output Arguments

`constraints` — Output dimension constraint
dimension constraint object

The output dimension constraint object, used by dataflow to optimize the automatic partitioning of a dataflow subsystem for multithreaded simulation and code generation.

Use the `outputDimensionConstraint` method to specify the `'MinimumSize'` property of the dimension constraint object. `'MinimumSize'` defines the minimum number of samples needed to run, specified as a nonnegative scalar integer. This value is used for downstream automatic frame-size calculation in dataflow subsystems.

Examples

Specify Output Dimension Constraint

Specify the required output dimension of the first input of a system object.

```
function dc = getOutputDimensionConstraintImpl(obj)
    dc = outputDimensionConstraint(obj, 'MinimumSize', obj.NumBins);
end
```

Specify Output Dimension Constraint for a Specified Port

To specify the output dimension for an input port other than the first input port, specify the port index in the constructor.

```
function dc = getOutputDimensionConstraintImpl(obj, index)
    if index == 1
        dc = outputDimensionConstraint(obj, 'MinimumSize', obj.NumBins);
    else
```

```
        dc = outputDimensionConstraint(obj, 'MinimumSize', 1);
    end
end
```

Default Behavior

If this method is not overwritten, the MATLAB System block sets the output dimension constraint to unspecified. The unspecified setting allows the dataflow scheduler to make conservative assumptions for optimization.

See Also

`getImpulseResponseLengthImpl` | `getInputDimensionConstraintImpl`

Topics

"Dataflow Domain" (DSP System Toolbox)

Introduced in R2018b

getOutputNamesImpl

Class: `matlab.System`

Names of MATLAB System block output ports

Syntax

```
names = getOutputNamesImpl(obj)
```

Description

`names = getOutputNamesImpl(obj)` specifies the names of the output ports from System object, `obj` implemented in a MATLAB System block. The size of `names` matches the number of outputs returned by the `getNumOutputs` method. If you change a property value that affects the number of outputs, the names of those outputs also change.

Class Information

This method is part of the `matlab.System` class.

Run-Time Details

`getOutputNamesImpl` is called by the MATLAB System block.

Method Authoring Tips

You must set `Access = protected` for this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `getOutputNamesImpl` method does not use the object, you can replace this input with ~.

Output Arguments

`names` — Names of outputs
empty string array (default) | string array

Names of the outputs for the specified object, returned as a string array whose length equals the number of outputs.

Examples

See Also

`getInputNamesImpl` | `getNumOutputsImpl`

Topics

"Specify Input and Output Names" (Simulink)

Introduced in R2013b

getPropertyGroupsImpl

Class: `matlab.System`

Property groups for System object display

Syntax

```
group = getPropertyGroupsImpl
```

Description

`group = getPropertyGroupsImpl` specifies how to display properties in the block dialog of a MATLAB System block. You can specify:

- Sections — Arrange properties into sections (panel on the MATLAB System block dialog) with `matlab.system.display.Section`.
- Section Groups — Arrange properties and sections into groups (tabs on the MATLAB System block dialog) with `matlab.system.display.SectionGroup`.
- Order of properties — Specify the order of properties within sections/groups
- Order of sections — Specify the order of sections within groups.

If you create a section, but that section is not in a section group, its properties appear above the block dialog tab panels.

Run-Time Details

In Simulink, `getPropertyGroupsImpl` modifies the MATLAB System block dialog.

In MATLAB, `getPropertyGroupsImpl` modifies how properties appear when the System object is displayed at the MATLAB command line. `getPropertyGroupsImpl` defines the order and groupings of properties. If your `getPropertyGroupsImpl` defines multiple section groups, only properties from the first section group are displayed at the command line. To display properties in other sections, a link is provided at the end of a System object property display. Group titles are also displayed at the command line.

Method Authoring Tips

- You must set `Access = protected` for this method.
- If you include a `getPropertyGroupsImpl` method but do not list all properties, the excluded properties do not appear in the dialog box.
- To omit the `Main` title for the first group of properties, in `matlab.system.display.SectionGroup`, set `TitleSource` to `'Auto'`.
- By default if you do not customize this method in your System object, the MATLAB System block dialog displays all public properties in the order specified in the System object class file.
- To customize property names, add comments above each property. For an example, see “System Object to Block Dialog Box Default Mapping” (Simulink).

Output Arguments

`group` — Property sections and groups
array of `SectionGroup` or `Section` objects

Property groups and sections to pass to the MATLAB System block, specified as an array of `matlab.system.display.Section` and `matlab.system.display.SectionGroup` objects.

See Also

`matlab.system.display.Header` | `matlab.system.display.Section` | `matlab.system.display.SectionGroup`

Topics

“Add Property Groups to System Object and MATLAB System Block” (Simulink)
“Mapping System Object Code to MATLAB System Block Dialog Box” (Simulink)

Introduced in R2013b

getSampleTime

Class: `matlab.System`

Query sample time

Syntax

```
sts = getSampleTime(obj)
```

Description

`sts = getSampleTime(obj)` returns the sample time specification for the System object `obj` when the System object is included in a MATLAB System block. You can call `getSampleTime` in the `stepImpl` method to change the algorithm based on the sample time.

Before sample time has propagated throughout the MATLAB System block model, `getSampleTime` returns the `getSampleTimeImpl` sample time specification. If your system object does not override `getSampleTimeImpl`, the default `Inherited` sample time specification is returned.

After sample time has propagated, `getSampleTime` returns the sample time specification populated with the actual MATLAB System block sample time type, sample time, and offset time.

Input Arguments

`obj` — System object
System object

System object included in a MATLAB System block that you want to query.

Output Arguments

`sts` — Sample time specification object
sample time specification object

The sample time specification for the System object. For more details about sample time specification objects, see `createSampleTime`.

Examples

Return Sample Time

This example of `stepImpl` returns a count value `y`, the current simulation time `ct`, and the sample time `st`. The sample time is obtained by calling `getSampleTime`.

```
function [y,ct,st] = stepImpl(obj,u)
      y = obj.Count + u;
      obj.Count = y;
      ct = getCurrentTime(obj);
      sts = getSampleTime(obj);
      st = sts.SampleTime;
    end
```

For a complete class definition, see “Specify Sample Time for MATLAB System Block System Objects” (Simulink).

See Also

`createSampleTime` | `getCurrentTime` | `getSampleTimeImpl`

Topics

“Specify Sample Time for MATLAB System Block System Objects” (Simulink)

Introduced in R2017b

getSampleTimeImpl

Class: `matlab.System`

Specify sample time type, offset time, and sample time

Syntax

```
sts = getSampleTimeImpl(obj)
```

Description

`sts = getSampleTimeImpl(obj)` returns the sample time specification created within the method body, `sts`, for the System object `obj`. Specify the sample time specification within the body of `getSampleTimeImpl` by calling `createSampleTime`. The sample time specification affects the simulation time when the System object is included in a MATLAB System block.

Run-Time Details

getSampleTimeImpl is called during setup by `setupImpl`.

Method Authoring Tips

You must set `Access = protected` for this method.

Default Behavior

If you do not include this method in your System object definition, the sample time is inherited.

Input Arguments

`obj` — System object

System object

System object handle used to access properties, states, and methods specific to the object. If your `getSampleTimeImpl` method does not use the object, you can replace this input with ~.

Output Arguments

`sts` — Sample time specification object

sample time specification object

An object defining the sample time specification values. You create this object with the `createSampleTime` function.

Examples

Specify Inherited Sample Time

Specify that the MATLAB System block should inherit the sample from upstream blocks, except if the sample time type is controllable.

```
function sts = getSampleTimeImpl(obj)
    sts = createSampleTime(obj,'ErrorOnPropagation','Controllable');
end
```

Specify Discrete Sample Time

Specify a discrete sample time for the MATLAB System block.

```
function sts = getSampleTimeImpl(obj)
    sts = createSampleTime(obj,'Type','Discrete',...
```

```
      'SampleTime',10.2,'OffsetTime',0.5);
end
```

See Also

`createSampleTime` | `getCurrentTime` | `getSampleTime`

Topics

"Specify Sample Time for MATLAB System Block System Objects" (Simulink)

Introduced in R2017b

getSimulateUsingImpl

Class: `matlab.System`

Specify value for Simulate using parameter

Syntax

```
simmode = getSimulateUsingImpl
```

Description

`simmode = getSimulateUsingImpl` specifies the simulation mode of the System object implemented in a MATLAB System block. The simulation mode restricts your System object to simulation using either code generation or interpreted execution. The associated `showSimulateUsingImpl` method controls whether the **Simulate using** option is displayed on the dialog box.

Run-Time Details

`getSimulateUsingImpl` is called by the MATLAB System block.

Method Authoring Tips

You must set `Access = protected` and `Static` for this method.

Output Arguments

`simmode` — Simulation mode
`"Code generation"` | `"Interpreted execution"`

Simulation mode, returned as the string `"Code generation"` or `"Interpreted execution"`. If you do not include the `getSimulateUsingImpl` method in your class definition file, the simulation mode is unrestricted. Depending on the value returned by

the associated `showSimulateUsingImpl` method, the simulation mode is displayed on the dialog box as either a dropdown list or not at all.

Examples

Specify the Simulation Mode

In the class definition file of your System object, define the simulation mode to display in the MATLAB System block. To prevent **Simulate using** from displaying, see `showSimulateUsingImpl`.

```
methods (Static, Access = protected)
   function simMode = getSimulateUsingImpl
      simMode = "Interpreted execution";
   end
end
```

See Also

`showSimulateUsingImpl`

Topics

“Control Simulation Type in MATLAB System Block” (Simulink)

Introduced in R2015a

getSimulinkFunctionNamesImpl

Class: `matlab.System`

Register Simulink function names used in your System object

Syntax

```
names = getSimulinkFunctionNamesImpl(obj)
```

Description

`names = getSimulinkFunctionNamesImpl(obj)` specifies the Simulink function `names` used in the System object `obj`.

If you use a Simulink function in your System object, you can only call a Simulink function from the `stepImpl`, `updateImpl`, or `outputImpl` method.

Run-Time Details

`getSimulinkFunctionNamesImpl` is invoked by the MATLAB System during model compilation to obtain the list of Simulink functions being called from the System object. Simulink uses this information to resolve the function names to a Simulink function declared in a Simulink Function block. If `getSimulinkFunctionNamesImpl` does not return the name of a function being called in the System object methods, Simulink attempts to resolve the function to a function on the MATLAB path.

Method Authoring Tips

You must set `Access = protected` for this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `getSimulinkFunctionNamesImpl` method does not use the object, you can replace this input with ~.

Output Arguments

names — Simulink function names
`{''}` (default) | cell array of character vectors | string array

The names of the Simulink function you want to call from your System object.

Examples

Syntax for Registering a Simulink Function in a System object

Register three Simulink functions that you want to call from your System object. The Simulink function names must match the name registered in the Simulink Function block.

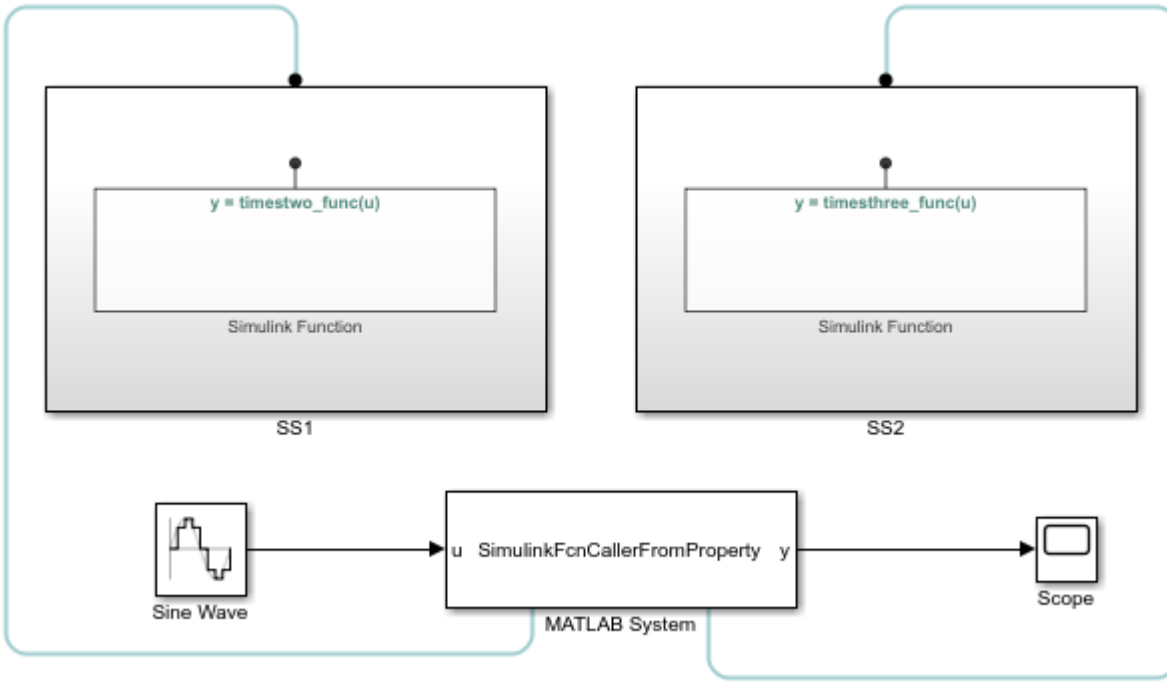
```
methods (Access = protected)
  function names = getSimulinkFunctionNamesImpl(obj)
    names = ["setPosition", "doProcessing", "timesTwo"];
  end
end
```

Call Simulink Functions from a MATLAB System Block

This example shows two `Simulink Functions` conditionally called by a `MATLAB System` block using the nontunable properties of the System object®.

The `MATLAB System` block calls one of the `Simulink Functions` inside two different subsystems, depending on the value of the signal coming from the `Sine Wave` block. If the value of the signal is less than 10, the `MATLAB System` block calls the `timestwo_func` `Simulink Function` inside the `SS1 Subsystem` block. If the value is larger than 10, it calls the `timesthree_func` in the `SS2 Subsystem` block.

Function names are defined as nontunable properties, are switched from string to functions using the `str2func` function. Then, these functions are declared as properties in the `getSimulinkFunctionNamesImpl` method.



See Also

Classes
`matlab.System`

Blocks
MATLAB System | Simulink Function

Topics

"Call Simulink Functions from MATLAB System Block" (Simulink)
"Simulink Functions Overview" (Simulink)

Introduced in R2019a

infoImpl

Class: `matlab.System`

Information about System object

Syntax

```
infoStruct = infoImpl(obj)
```

Description

`infoStruct = infoImpl(obj)` specifies information about the current configuration of a System object. This information is returned in a struct from the `info` object function. The default `infoImpl` method, which is used if you do not include `infoImpl` in your class definition file, returns an empty struct.

Run-Time Details

`infoImpl` is called by the `info` object function.

Method Authoring Tips

You must set `Access = protected` for this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `infoImpl` method does not use the object, you can replace this input with `~`.

Output Arguments

`infoStruct` — Information about the System object
struct

Information about the System object's current configuration returned as a struct.

Examples

Specify Current System object Information

Define the `infoImpl` method to return current count information.

```
methods (Access = protected)
   function s = infoImpl(obj)
      s = struct('Count',obj.Count);
   end
end
```

See Also

Topics
"Define System Object Information"

isInactivePropertyImpl

Class: `matlab.System`

Status of inactive property

Syntax

```
flag = isInactivePropertyImpl(obj,prop)
```

Description

`flag = isInactivePropertyImpl(obj,prop)` specifies whether a public, non-state property is inactive and not visible for the current object configuration. An inactive property is a property that is not relevant to the object because of the values of other properties. Inactive properties are not shown if you use the `disp` method to display object properties. If you attempt to use public access to directly access or use `get` or `set` on an inactive property, a warning occurs.

Run-Time Details

`isInactiveProperty` is called by `disp`, `get` and `set` methods.

Method Authoring Tips

You must set `Access = protected` for this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `isInactivePropertyImpl` method does not use the object, you can replace this input with ~.

`prop` — Public, non-state property name
property name

Public, non-state property name

Output Arguments

`flag` — Inactive status Indicator of the input property
`true` | `false`

Inactive status Indicator of the input property `prop` for the current object configuration.

Examples

Specify When a Property Is Inactive

Display the `InitialValue` property only when the `UseRandomInitialValue` property value is `false`.

```
methods (Access = protected)
  function flag = isInactivePropertyImpl(obj,propertyName)
    if strcmp(propertyName,'InitialValue')
      flag = obj.UseRandomInitialValue;
    else
      flag = false;
    end
  end
end
```

See Also
`setProperties`

Topics
"Hide Inactive Properties"

Introduced in R2012b

isInputSizeLockedImpl

Class: `matlab.System`

Status of locked input size

Syntax

```
flag = isInputSizeLockedImpl(obj,i)
```

Note This method will be removed in a future release. Use instead `isInputSizeMutableImpl`.

Description

`flag = isInputSizeLockedImpl(obj,i)` specifies whether the `i`th input to the System object cannot change its size during subsequent calls to run that object. If `flag` is `true`, the size is locked and inputs to the System object cannot change size while the object is locked. If `flag` is `false`, the input is variable size and is not locked. In the unlocked case, the size of inputs to the object can change while the object is running and locked.

`isInputSizeLockedImpl` executes once for each input during System object initialization.

Note You must set `Access = protected` for this method.

Input Arguments

`obj`

System object

`i`

System object input port number

Output Arguments

`flag`

Flag indicating whether the size of inputs to the specified port is locked, returned as a logical scalar value. If the value of `isInputSizeLockedImpl` is `true`, the size of the current input to that port is compared to the first input to that port. If the sizes do not match, an error occurs.

Default: `false`

Examples

Check If Input Size Is Locked

Specify in your class definition file to check whether the size of the System object input is locked.

```
methods (Access = protected)
   function flag = isInputSizeLockedImpl(~,index)
      flag = true;
   end
end
```

Compatibility Considerations

`isInputSizeLockedImpl` will be removed

Not recommended starting in R2018a

This method will be removed in a future release. Use instead `isInputSizeMutableImpl`.

See Also

`matlab.System`

Topics

"Handle Input Specification Changes"

isInputDirectFeedthroughImpl

Class: `matlab.System`

Direct feedthrough status of input

Syntax

```
[flag1,...,flagN] = isInputDirectFeedthroughImpl(obj)
[flag1,...,flagN] = isInputDirectFeedthroughImpl(obj,
input,input2,...)
```

Description

`[flag1,...,flagN] = isInputDirectFeedthroughImpl(obj)` specifies whether each input is a direct feedthrough input. If direct feedthrough is `true`, the output depends on the input at each time instant.

`[flag1,...,flagN] = isInputDirectFeedthroughImpl(obj, input,input2,...)` uses one or more of the System object input specifications to determine whether inputs have direct feedthrough.

If you do not include the `isInputDirectFeedthroughImpl` method in your System object class definition file, all inputs are assumed to be direct feedthrough.

Code Generation

The following cases describe when System objects in Simulink code generation use direct or nondirect feedthrough.

System object's code generation support	Uses a propagation `Impl` method	SimulinkCode Generation Result
Y	N	Simulink automatically infers the direct feedthrough settings from the System object code.
Y	Y	Simulink does not automatically infer the direct feedthrough settings. Instead, it uses the value returned by the `isInputDirectFeedthroughImpl` method.
N	—	Default `isInputDirectFeedthroughImpl` method returns false, indicating that direct feedthrough is not enabled. To override the default behavior, implement the `isInputDirectFeedthroughImpl` method in your class definition file.

Run-Time Details

`isInputDirectFeedthroughImpl` is called by the MATLAB System block.

Method Authoring Tips

- You must set `Access = protected` for this method.
- You cannot modify, implement, or access tunable properties in this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `isInputDirectFeedthroughImpl` method does not use the object, you can replace this input with ~.

`input1,input2,...` — Inputs to the System object
inputs to the System object algorithm

Inputs to the algorithm (`stepImpl`) of the System object. The inputs list must match the order of inputs in the `stepImpl` signature.

Output Arguments

`flag1,...,flagN` — Output flag for each input to the System object
logical

Logical value, either `true` or `false` indicating whether the input is direct feedthrough. The number of output flags must match the number of inputs to the System object (inputs to `stepImpl`, `outputImpl`, or `updateImpl`).

Examples

Specify Input as Nondirect Feedthrough

Use `isInputDirectFeedthroughImpl` in your class definition file for marking all inputs as nondirect feedthrough.

```
methods (Access = protected)
   function flag = isInputDirectFeedthroughImpl(~)
      flag = false;
   end
end
```

Complete Class Definition

```
classdef intDelaySysObj < matlab.System
   % intDelaySysObj Delay input by specified number of samples.

   properties
      InitialOutput = 0;
   end
   properties (Nontunable)
      NumDelays = 1;
   end
   properties (DiscreteState)
      PreviousInput;
   end
```

```
   methods (Access = protected)
      function validatePropertiesImpl(obj)
         if ((numel(obj.NumDelays)>1) || (obj.NumDelays <= 0))
            error('Number of delays must be > 0 scalar value.');
         end
         if (numel(obj.InitialOutput)>1)
            error('Initial Output must be scalar value.');
         end
      end

      function setupImpl(obj)
         obj.PreviousInput = ones(1,obj.NumDelays)*obj.InitialOutput;
      end

      function resetImpl(obj)
         obj.PreviousInput = ones(1,obj.NumDelays)*obj.InitialOutput;
      end

      function [y] = outputImpl(obj,~)
         y = obj.PreviousInput(end);
      end
      function updateImpl(obj, u)
         obj.PreviousInput = [u obj.PreviousInput(1:end-1)];
      end
      function flag = isInputDirectFeedthroughImpl(~)
         flag = false;
      end
   end
end
```

See Also

`outputImpl` | `updateImpl`

Topics

"Use Update and Output for Nondirect Feedthrough" (Simulink)

Introduced in R2013b

isOutputComplexImpl

Class: `matlab.System`

Complexity of output ports

Syntax

```
[cp_1,cp_2,...,cp_n] = isOutputComplexImpl(obj)
```

Description

`[cp_1,cp_2,...,cp_n] = isOutputComplexImpl(obj)` returns whether each output port has complex data. The number of outputs must match the value returned from the `getNumOutputs` method or the number of output arguments listed in the `stepImpl` method.

For System objects with one input and one output and where you want the input and output complexities to be the same, you do not need to implement this method. In this case `isOutputComplexImpl` assumes the input and output complexities are the same and returns the complexity of the input.

If your System object has more than one input or output or you need the output and input complexities to be different, you must implement the `isOutputComplexImpl` method to define the output complexity. You also must use the `propagatedInputComplexity` method if the output complexity differs from the input complexity.

Run-Time Details

During Simulink model compilation and propagation, the MATLAB System block calls the `isOutputComplex` method, which then calls the `isOutputComplexImpl` method to determine the output complexity.

Method Authoring Tips

- You must set `Access = protected` for this method.
- You cannot modify any properties in this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `isOutputComplexImpl` method does not use the object, you can replace this input with ~.

Output Arguments

`cp_1,cp_2,...` — Complexity flag
`true` | `false`

Logical, scalar value indicating whether the specific output port is complex (`true`) or real (`false`).

Examples

Specify Output as Real-Valued

Specify in your class definition file that the output from a System object is a real value.

```
methods (Access = protected)
   function c1 = isOutputComplexImpl(obj)
       c1 = false;
```

```
    end
end
```

See Also

`propagatedInputComplexity`

Topics

"Set Output Complexity" (Simulink)

Introduced in R2013b

isOutputFixedSizeImpl

Class: `matlab.System`

Fixed- or variable-size output ports

Syntax

```
[flag_1,flag_2,...flag_n] = isOutputFixedSizeImpl(obj)
```

Description

`[flag_1,flag_2,...flag_n] = isOutputFixedSizeImpl(obj)` returns whether each output port is fixed size. The number of outputs must match the value returned from the `getNumOutputs` method, which is the number of output arguments listed in the `stepImpl` method.

For System objects with one input and one output and where you want the input and output fixed sizes to be the same, you do not need to implement this method. In this case `isOutputFixedSizeImpl` assumes the input and output fixed sizes are the same and returns the fixed size of the input.

If your System object has more than one input or output or you need the output and input fixed sizes to be different, you must implement the `isOutputFixedSizeImpl` method to define the output fixed size. You also must use the `propagatedInputFixedSize` method if the output fixed size status differs from the input fixed size status.

Run-Time Details

During Simulink model compilation and propagation, the MATLAB System block calls the `isOutputFixedSize` method, which then calls the `isOutputFixedSizeImpl` method to determine the output fixed size.

All inputs default to variable-size inputs For these inputs, the output size is the maximum input size.

Method Authoring Tips

- You must set `Access = protected` for this method.
- You cannot modify any properties in this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `isOutputFixedSizeImpl` method does not use the object, you can replace this input with ~.

Output Arguments

`flag_1,flag2,...` — Fixed size flag
`true` | `false`

Logical, scalar value indicating whether the specific output port is fixed size (`true`) or variable size (`false`).

Examples

Specify Output as Fixed Size

Specify in your class definition file that the output from a System object is of fixed size.

```
methods (Access = protected)
   function c1 = isOutputFixedSizeImpl(obj)
      c1 = true;
```

```
    end
end
```

See Also

`propagatedInputFixedSize`

Topics

"Set Fixed- or Variable-Size Output" (Simulink)

Introduced in R2013b

isDiscreteStateSpecificationMutableImpl

Class: `matlab.System`

Control whether discrete states can change data type

Syntax

```
mutable = isDiscreteStateSpecificationMutableImpl(obj)
```

Description

`mutable = isDiscreteStateSpecificationMutableImpl(obj)` returns whether discrete states can change data type when the object is in use. If this method returns `false`, the discrete state must maintain the same data type while the object is in use.

Method Authoring Tips

You must set `Access = protected` for this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `isDiscreteStateSpecificationMutableImpl` method does not use the object, you can replace this input with ~.

Output Arguments

`mutable` — Whether state specification can change or not
`true` (default) | `false`

If you do not implement this method, discrete states can change data type unless the `StrictDefaults` on page 1-18068 class attribute is set. If you implement this method, returning `true` means discrete state data type can change and `false` means they cannot change.

Examples

Disallow Changes to Discrete States

Restrict changes to the data type of discrete states by adding the `isDiscreteStateSpecificationMutableImpl` method and returning `false`. By adding this method, users of the System object cannot change the data type of discrete states while the System object is in use.

```
function flag = isDiscreteStateSpecificationMutableImpl(obj)
    flag = false;
end
```

See Also

`getDiscreteStateImpl` | `getDiscreteStateSpecificationImpl` | `isTunablePropertyDataTypeMutableImpl`

Introduced in R2018a

isInputComplexityMutableImpl

Class: `matlab.System`

Set whether System object input complexity can change

Syntax

```
mutable = isInputComplexityMutableImpl(obj,index)
```

Description

`mutable = isInputComplexityMutableImpl(obj,index)` returns whether the `index`th input to the object can change complexity when the object is in use.

Method Authoring Tips

You must set `Access = protected` for this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `isInputComplexityMutableImpl` method does not use the object, you can replace this input with ~.

`index` — Ordinal index of input
positive integer

This argument specifies which input to `stepImpl` is checked for complexity mutability. The index number is the ordinal position of the input in the `stepImpl` signature.

Output Arguments

`mutable` — Whether input complexity can change or not
`true` | `false`

If you do not implement this method, inputs can change complexity unless the `StrictDefaults` on page 1-18068 class attribute is set. If you implement this method, returning `true` means that input complexity can change, and `false` means they cannot change.

Examples

Restrict Input Complexity Changes for Inputs

Restrict the complexity of all inputs by adding the `isInputComplexityMutableImpl` method and returning `false`. By adding this method, users of the System object cannot change the complexity of inputs while the System object is in use.

```
function flag = isInputComplexityMutableImpl(obj,~)
    flag = false;
end
```

To avoid a warning about unused variables, this example uses ~ as the second input argument. For more information about using ~ in place of arguments, see “Using ~ as an Input Argument in Method Definitions”.

Restrict Input Complexity Changes for One Input

This example shows how to write the `isInputComplexityMutableImpl` method to only restrict one input. `isInputComplexityMutableImpl` returns `true` for all inputs except input one.

```
methods (Access = protected)
    function flag = isInputComplexityMutableImpl(obj,index)
        flag = (index ~= 1);
```

```
    end
end
```

See Also

`isInputDataTypeMutableImpl` | `isInputSizeMutableImpl` | `matlab.System`

Topics

"Change Input Complexity, Dimensions, or Data Type"

Introduced in R2018a

isInputDataTypeMutableImpl

Class: `matlab.System`

Set whether System object input data type can change

Syntax

```
mutable = isInputDataTypeMutableImpl(obj,index)
```

Description

`mutable = isInputDataTypeMutableImpl(obj,index)` returns whether the `index`th input to the object can change data type when the object is in use.

Method Authoring Tips

You must set `Access = protected` for this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `isInputDataTypeMutableImpl` method does not use the object, you can replace this input with ~.

`index` — Ordinal index of input
positive integer

This argument specifies which input to `stepImpl` is checked for data type mutability. The index number is the ordinal position of the input in the `stepImpl` signature.

Output Arguments

`mutable` — Whether input data type can change or not
`true` (default) | `false`

If you do not implement this method, inputs can change data type unless the `StrictDefaults` on page 1-18068 class attribute is set. If you implement this method, returning `true` means that input data type can change, and `false` means they cannot change.

Examples

Restrict Input Data Type for Inputs

Restrict changes to the data type of all inputs by adding the `isInputDataTypeMutableImpl` method and returning `false`. By adding this method, users of the System object cannot change the data type of inputs while the System object is in use.

```
function flag = isInputDataTypeMutableImpl(obj,~)
    flag = false;
end
```

To avoid a warning about unused variables, this example uses ~ as the second input argument. For more information about using ~ in place of arguments, see “Using ~ as an Input Argument in Method Definitions”.

Restrict Input Data Type for One Input

This example shows how to write the `isInputDataTypeMutableImpl` method to only restrict one input. `isInputDataTypeMutableImpl` returns `true` for all inputs except input one.

```
methods (Access = protected)
    function flag = isInputDataTypeMutableImpl(obj,index)
        flag = (index ~= 1)
```

```
    end
end
```

See Also

isInputComplexityMutableImpl | isInputSizeMutableImpl | processInputSpecificationChangeImpl

Introduced in R2018a

isInputSizeMutableImpl

Class: `matlab.System`

Set whether System object input size can change

Syntax

```
mutable = isInputSizeMutableImpl(obj,index)
```

Description

`mutable = isInputSizeMutableImpl(obj,index)` returns whether the `index`th input to the object can change size when the object is in use.

Method Authoring Tips

You must set `Access = protected` for this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `isInputSizeMutableImpl` method does not use the object, you can replace this input with ~.

`index` — Ordinal index of input
positive integer

This argument specifies which input to `stepImpl` is checked for size mutability. The index number is the ordinal position of the input in the `stepImpl` signature.

Output Arguments

`mutable` — Whether input size can change or not
`true` (default) | `false`

If you do not implement this method, inputs can change size unless the `StrictDefaults` on page 1-18068 class attribute is set. If you implement this method, returning `true` means that input size can change, and `false` means they cannot change.

Examples

Restrict Input Size Changes for Inputs

Restrict the size of all inputs by adding the `isInputSizeMutableImpl` method and returning `false`. By adding this method, users of the System object cannot change the size of inputs while the System object is in use.

```
function flag = isInputSizeMutableImpl(obj,~)
    flag = false;
end
```

To avoid a warning about unused variables, this example uses ~ as the second input argument. For more information about using ~ in place of arguments, see “Using ~ as an Input Argument in Method Definitions”.

Restrict Input Size Changes for Single Input

This example shows how to write the `isInputSizeMutableImpl` method to only restrict the first input. `isInputSizeMutableImpl` returns `true` for all inputs except input the first input.

```
methods (Access = protected)
    function flag = isInputSizeMutableImpl(obj,index)
        flag = (index ~= 1);
```

```
    end
end
```

See Also

`isInputComplexityMutableImpl` | `isInputDataTypeMutableImpl` | `matlab.System` | `processInputSpecificationChangeImpl`

Topics

"Handle Input Specification Changes"

Introduced in R2018a

isTunablePropertyDataTypeMutableImpl

Class: `matlab.System`

Set whether tunable properties can change data type

Syntax

```
mutable = isTunablePropertyDataTypeMutableImpl(obj)
```

Description

`mutable = isTunablePropertyDataTypeMutableImpl(obj)` returns whether tunable properties can change data type when the object is in use.

Method Authoring Tips

You must set `Access = protected` for this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `isTunablePropertyDataTypeMutableImpl` method does not use the object, you can replace this input with ~.

Output Arguments

`mutable` — Whether tunable property data types can change
`true` (default) | `false`

If you do not implement this method, tunable properties can change data types unless the `StrictDefaults` on page 1-18068 class attribute is set. If you implement this method, returning `true` means tunable property data types can change and `false` means they cannot change.

Examples

Restrict Data Type of Tunable Properties

Restrict the data type of tunable properties by adding the `isTunablePropertyDataTypeMutableImpl` method and returning `false`. By adding this method, users of the System object cannot change the data type of tunable properties while the System object is in use.

```
function flag = isTunablePropertyDataTypeMutableImpl(obj)
    flag = false;
end
```

See Also

`isInputComplexityMutableImpl` | `isInputDataTypeMutableImpl` | `isInputDirectFeedthroughImpl` | `isInputSizeMutableImpl` | `processTunedPropertiesImpl`

Topics

"Validate Property and Input Values"

Introduced in R2018a

loadObjectImpl

Class: `matlab.System`

Load System object from MAT file

Syntax

```
loadObjectImpl(obj,s,wasInUse)
```

Description

`loadObjectImpl(obj,s,wasInUse)` implements the code to load a saved System object from a structure, `s`, or from a MAT file. If the object was in use when saved, the `wasInUse` input indicates that in-use state. Your `loadObjectImpl` method should correspond to your `saveObjectImpl` method to ensure that all saved properties and data are loaded.

Run-Time Details

End users can use `load`, which calls `loadObjectImpl` to load a saved System object into their workspace.

Method Authoring Tips

You must set `Access = protected` for this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `loadObjectImpl` method does not use the object, you can replace this input with ~.

s — Saved data
struct | MAT file

Saved data returned from `saveObjectImpl`.

`wasInUse` — In-use state
`true` | `false`

If the object was in use when saved, the `wasInUse` is `true`.

Examples

Load System object

Load a saved System object. In this example, the object contains:

- A child object
- Protected and private properties
- A discrete state

It also loads states if the object is in use and calls the `loadObjectImpl` method.

```
methods (Access = protected)
  function loadObjectImpl(obj,s,wasInUse)
    obj.child = matlab.System.loadObject(s.child);

    obj.protectedprop = s.protectedprop;
    obj.pdependentprop = s.pdependentprop;

    if wasInUse
      obj.state = s.state;
    end

    loadObjectImpl@matlab.System(obj,s,wasInUse);
  end
end
```

See Also

`saveObjectImpl`

Topics

"Save and Load System Object"

Introduced in R2012b

nargin

Number of input arguments for System object

Syntax

```
numInputs = nargin(obj)
nargin
```

Description

`numInputs = nargin(obj)` returns the number of input arguments that are required by the `obj` algorithm definition. This function is a System object extension of the general `nargin` function.

If your System object uses the `matlab.system.mixin.Nondirect` class, `nargin` returns the number of input arguments for the `updateImpl` and `outputImpl` methods.

`nargin` returns the number of input arguments specified in the call to the currently executing System object. Use this syntax in the body of a System object only.

Examples

Use `nargin` on System object with Variable-Size Input

This example shows the output from `nargin` when used on a System object `AddUp` with variable-sized input.

A `marker` System object is defined as follows:

```
classdef AddUp < matlab.System
   methods(Access = protected)
        function y = stepImpl(obj,u,varargin)
            % Implement algorithm.
            y = u + sum([varargin{:}]);
```

```
        end
    end
end
```

Create the object and call `nargin`.

```
total = AddUp();
nargin(total)
```

```
ans =
    -2
```

Input Arguments

`obj` — System object
System object

System object to query.

Output Arguments

`numInputs` — Number of inputs
integer

This argument represents the number of inputs needed to call the System object.

If the output is nonnegative, the number of inputs cannot change while the object is in use.

If the output is negative, the number of inputs can change from call to call. This situation occurs when the System object does not override `getNumInputsImpl` and the `stepImpl` signature contains `varargin`.

See Also

`getNumInputs` | `getNumInputsImpl` | `nargin` | `nargout`

Topics

"Change the Number of Inputs"

Introduced in R2018a

nargout

Number of output arguments for System object

Syntax

```
numOutputs = nargout(obj)
nargout
```

Description

`numOutputs = nargout(obj)` returns the number of output arguments that are required by the `obj` algorithm definition. This function is a System object extension of the general `nargout` function.

If your System object uses the `matlab.system.mixin.Nondirect` class, `nargin` returns the number of output arguments for the `updateImpl` and `outputImpl` methods.

`nargout` returns the number of output arguments specified in the call to the currently executing System object. Use this syntax in the body of a System object only.

Input Arguments

`obj` — System object
System object

System object to query.

Output Arguments

`numOutputs` — Number of outputs
integer

This argument represents the number of outputs needed to call the System object.

If the output is nonnegative, the number of outputs is cannot change while the object is in use.

If the output is negative, the number of outputs can change from call to call. This situation occurs when the System object does not override `getNumOutputsImpl` and the `stepImpl` signature contains `varargout`.

See Also

`getNumOutputs` | `getNumOutputsImpl` | `nargin` | `nargout`

Topics

"Change the Number of Inputs"

Introduced in R2018a

outputImpl

Class: `matlab.System`

Output calculation from input or internal state of System object

Syntax

```
[y1,y2,...,yN] = outputImpl(obj,u1,u2,...,uN)
```

Description

`[y1,y2,...,yN] = outputImpl(obj,u1,u2,...,uN)` specifies the algorithm to output the System object states. The output values are calculated from the states and property values. Any inputs that you set to nondirect feedthrough are ignored during output calculation.

Run-Time Details

`outputImpl` is called by the internal `output` method. It is also called before the `updateImpl` method. For sink objects, calling `updateImpl` before `outputImpl` locks the object. For all other types of objects, calling `updateImpl` before `outputImpl` causes an error.

Method Authoring Tips

- You must set `Access = protected` for this method.
- If the System object will be used in the Simulink MATLAB System block, you cannot modify any tunable properties in this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `outputImpl` method does not use the object, you can replace this input with ~.

`u1,u2,...uN` — Inputs to the algorithm
inputs

Inputs from the algorithm. The number of inputs must match the number of inputs returned by the `getNumInputs` method. Nondirect feedthrough inputs are ignored during normal execution of the System object. However, for code generation, you must provide these inputs even if they are empty.

Output Arguments

`y1,y2,...yN` — Outputs of the algorithm
outputs

Outputs calculated from the specified algorithm. The number of outputs must match the number of outputs returned by the `getNumOutputs` method.

Examples

Set Up Output that Does Not Depend on Input

Specify in your class definition file that the output does not directly depend on the current input with the `outputImpl` method. `PreviousInput` is a property of the `obj`.

```
methods (Access = protected)
   function [y] = outputImpl(obj, ~)
      y = obj.PreviousInput(end);
   end
end
```

See Also

`isInputDirectFeedthroughImpl` | `updateImpl`

Topics

"Use Update and Output for Nondirect Feedthrough" (Simulink)

Introduced in R2013b

propagatedInputComplexity

Class: `matlab.System`

Complexity of input during Simulink propagation

Syntax

```
flag = propagatedInputComplexity(obj,index)
```

Description

`flag = propagatedInputComplexity(obj,index)` returns `true` or `false` to indicate whether the input argument for the indicated System object is complex. `index` specifies the input for which to return the complexity flag.

You can use `propagatedInputComplexity` only from within the `isOutputComplexImpl` method in your class definition file. Use `isOutputComplexImpl` when:

- Your System object has more than one input or output.
- The input complexity determines the output complexity.
- The output complexity must differ from the input complexity.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `propagatedInputComplexity` method does not use the object, you can replace this input with ~.

`index` — Input index
positive integer

Index of the specified input. Do not count the `obj` in the `index`. The first input is always `obj`.

Output Arguments

`flag` — Complexity flag
`true` | `false`

Complexity of the specified input, returned as `true` or `false`

Examples

Match Input and Output Complexity

Get the complexity of the second input when you run the object and set the output to match it. Assume that the first input has no impact on the output complexity.

```
methods (Access = protected)
   function outcomplx = isOutputComplexImpl(obj)
      outcomplx = propagatedInputComplexity(obj,2);
   end
end
```

See Also
`isOutputComplexImpl`

Topics
"Set Output Complexity" (Simulink)

Introduced in R2014a

propagatedInputDataType

Class: `matlab.System`

Data type of input during Simulink propagation

Syntax

```
dt = propagatedInputDataType(obj,index)
```

Description

`dt = propagatedInputDataType(obj,index)` returns the data type of an input argument for a System object. `index` specifies the input for which to return the data type.

You can use `propagatedInputDataType` only from within `getOutputDataTypeImpl`. Use `getOutputDataTypeImpl` when:

- Your System object has more than one input or output.
- The input data type status determines the output data type.
- The output data type must differ from the input data type.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `propagatedInputDataType` method does not use the object, you can replace this input with ~.

`index` — Input index
positive integer

Index of the specified input. Do not count the `obj` in the `index`. The first input is always `obj`.

Output Arguments

dt — Input data type
character vector | numerictype object

Data type of the specified input, returned as a character vector for floating-point input or as a numerictype for fixed-point input.

Examples

Match Input and Output Data Type

Get the data type of the second input. If the second input data type is `double`, then the output data type is `int32`. For all other cases, the output data type matches the second input data type. Assume that the first input has no impact on the output.

```
methods (Access = protected)
   function dt = getOutputDataTypeImpl(obj)
       if strcmpi(propagatedInputDataType(obj,2),'double')
           dt = 'int32';
       else
           dt = propagatedInputDataType(obj,2);
       end
   end
end
```

See Also

`getOutputDataTypeImpl`

Topics

“Set Output Data Type” (Simulink)
“Data Type Propagation” (Simulink)

Introduced in R2014a

propagatedInputFixedSize

Class: `matlab.System`

Fixed-size status of input during Simulink propagation

Syntax

```
flag = propagatedInputFixedSize(obj,index)
```

Description

`flag = propagatedInputFixedSize(obj,index)` returns `true` or `false` to indicate whether an input argument of a System object is fixed size. `index` specifies the input for which to return the fixed-size flag.

You can use `propagatedInputFixedSize` only from within `isOutputFixedSizeImpl`. Use `isOutputFixedSizeImpl` when:

- Your System object has more than one input or output.
- The input fixed-size status determines the output fixed-size status.
- The output fixed-size status must differ from the input fixed-size status.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `propagatedInputFixedSize` method does not use the object, you can replace this input with ~.

`index` — Input index
positive integer

Index of the specified input. Do not count the `obj` in the `index`. The first input is always `obj`.

Output Arguments

`flag` — Fixed size flag
`true` | `false`

Fixed-size status of the specified input, returned as `true` or `false`.

Examples

Match Fixed-Size Status of Input and Output

Get the fixed-size status of the third input and set the output to match it. Assume that the first and second inputs have no impact on the output.

```
methods (Access = protected)
   function outtype = isOutputFixedSizeImpl(obj)
      outtype = propagatedInputFixedSize(obj,3)
   end
end
```

See Also

`isOutputFixedSizeImpl`

Topics

"Set Fixed- or Variable-Size Output" (Simulink)

Introduced in R2014a

propagatedInputSize

Class: `matlab.System`

Size of input during Simulink propagation

Syntax

```
size = propagatedInputSize(obj,index)
```

Description

`size = propagatedInputSize(obj,index)` returns, as a vector, the input size of the specified System object. The `index` specifies the input for which to return the size information. (Do not count the `obj` in the `index`. The first input is always `obj`.)

You can use `propagatedInputSize` only from within the `getOutputSizeImpl` method in your class definition file. Use `getOutputSizeImpl` when:

- Your System object has more than one input or output.
- The input size determines the output size.
- The output size must differ from the input size.

Note For variable-size inputs, the propagated input size from `propagatedInputSize` differs depending on the environment.

- MATLAB — `propagatedInputSize` returns the size of the inputs used when you run the object for the first time.
- Simulink — `propagatedInputSize` returns the upper bound of the input sizes.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `propagatedInputSize` method does not use the object, you can replace this input with ~.

`index` — Input index
integer

Index of the specified input

Output Arguments

`size` — Input size
numeric vector

Size of the specified input, returned as a vector

Examples

Match Size of Input and Output

Get the size of the second input. If the first dimension of the second input has a size greater than 1, then set the output size to a 1 x 2 vector. For all other cases, the output is a 2 x 1 matrix. Assume that the first input has no impact on the output size.

```
methods (Access = protected)
   function outsize = getOutputSizeImpl(obj)
       size = propagatedInputSize(obj,2);
       if size(1) == 1
           outsize = [1,2];
       else
           outsize = [2,1];
       end
   end
end
```

See Also

`getOutputSizeImpl`

Topics

"Set Output Size" (Simulink)

Introduced in R2014a

processInputSpecificationChangeImpl

Class: `matlab.System`

Perform actions when input size, complexity, or data type change

Syntax

```
processInputSpecificationChangeImpl(obj,input,input2, ...)
```

Description

`processInputSpecificationChangeImpl(obj,input,input2, ...)` implements specific actions when the input specification changes, such as data type, size, or complexity. You use this method when properties depend on the data type, size, or complexity of inputs.

Class Information

This method is part of the `matlab.System` class.

Run-Time Details

`processInputSpecificationChangeImpl` is called when running the System object using the object name or `step`. For details, see “Detailed Call Sequence”

Method Authoring Tips

You must set `Access = protected` for this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `processInputSpecificationChangeImpl` method does not use the object, you can replace this input with ~.

`input1,input2,...` — Inputs to the System object
inputs to the System object algorithm

Inputs to the algorithm (`stepImpl`) of the System object. The inputs list must match the order of inputs in the `stepImpl` signature.

Examples

Specify Actions When Input Size Changes

This example shows how to use `processInputSpecificationChangeImpl` to modify the `NumIterations` property when the size of `x` changes. Even though this method does not take action when the additional `varargin` inputs change, they are included in the method signature.

```
methods (Access = Protected)
    function y = stepImpl(obj,x,varargin)
        for n=1:obj.NumIterations
            y
        end
    end
    function processInputSpecificationChangeImpl(obj,x,varargin)
        obj.NumIterations = size(x,1);
    end
end
```

See Also

`isInputComplexityMutableImpl` | `isInputDataTypeMutableImpl` | `isInputSizeMutableImpl` | `processTunedPropertiesImpl` | `validateInputsImpl`

Topics

"Validate Property and Input Values"

Introduced in R2018a

processTunedPropertiesImpl

Class: `matlab.System`

Action when tunable properties change

Syntax

```
processTunedPropertiesImpl(obj)
```

Description

`processTunedPropertiesImpl(obj)` specifies the algorithm to perform when one or more tunable property values change. This method is called as part of the next call to the System object after a tunable property value changes. A property is tunable only if its `Nontunable` attribute is `false`, which is the default.

Run-Time Details

`processTunedPropertiesImpl` is called when you run the System object.

In MATLAB when multiple tunable properties are changed before running the System object, `processTunedPropertiesImpl` is called only once for all the changes. `isChangedProperty` returns `true` for all the changed properties.

In Simulink, when a parameter is changed in a MATLAB System block dialog, the next simulation step calls `processTunedPropertiesImpl` before calling `stepImpl`. All tunable parameters are considered changed and `processTunedPropertiesImpl` method is called for each of them. `isChangedProperty` returns `true` for all the dialog properties.

For details, see “Detailed Call Sequence”.

Method Authoring Tips

- You must set `Access = protected` for this method.
- Use this method when a tunable property affects the value of a different property.
- If the System object will be used in the Simulink MATLAB System block, you cannot modify any tunable properties in this method.
- To check if a property has changed since `stepImpl` was last called, use `isChangedProperty` within `processTunedPropertiesImpl`. See “Specify Action When Tunable Property Changes” on page 1-18190 for an example.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `processTunedPropertiesImpl` method does not use the object, you can replace this input with ~.

Examples

Specify Action When Tunable Property Changes

Use `processTunedPropertiesImpl` to recalculate the lookup table if the value of either the `NumNotes` or `MiddleC` property changes before the next call to the System object. `propChange` indicates if either property has changed.

```
methods (Access = protected)
   function processTunedPropertiesImpl(obj)
      propChange = isChangedProperty(obj,'NumNotes') ||...
         isChangedProperty(obj,'MiddleC')
      if propChange
         obj.pLookupTable = obj.MiddleC * (1+log(1:obj.NumNotes)/log(12));
      end
```

```
    end
end
```

See Also

`setProperties` | `validatePropertiesImpl`

Topics

"Process Tuned Properties"
"Validate Property and Input Values"
"Define Property Attributes"

Introduced in R2013b

releaseImpl

Class: `matlab.System`

Release resources

Syntax

```
releaseImpl(obj)
```

Description

`releaseImpl(obj)` releases any resources used by the System object, such as file handles or devices. This method also performs any necessary cleanup tasks. To release resources for a System object, you must use `releaseImpl` instead of a destructor.

Run-Time Details

`releaseImpl` is called by the `release` method. `releaseImpl` is also called when the object is deleted or cleared from memory, or when all references to the object have gone out of scope. For details, see “Detailed Call Sequence”.

Method Authoring Tips

You must set `Access = protected` for this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `releaseImpl` method does not use the object, you can replace this input with ~.

Examples

Close a File and Release Its Resources

Use the `releaseImpl` method to close a file opened by the System object.

```
methods (Access = protected)
   function releaseImpl(obj)
     fclose(obj.pFileID);
   end
end
```

See Also

`resetImpl`

Topics

"Release System Object Resources"

Introduced in R2011b

resetImpl

Class: `matlab.System`

Reset System object states

Syntax

```
resetImpl(obj)
```

Description

`resetImpl(obj)` specifies the algorithm that initializes or resets the states of a System object. Typically you reset the states to a set of initial values, which is useful for initialization at the start of simulation.

Run-Time Details

`resetImpl` is called by the `reset` method only if the object is in use. The object remains as "in-use" after it is reset. For details, see “Detailed Call Sequence”.

Method Authoring Tips

- You must set `Access = protected` for this method.
- If this System object will be used in the Simulink MATLAB System block, you cannot modify any tunable properties in this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `resetImpl` method does not use the object, you can replace this input with ~.

Examples

Reset Property Value

Use the `resetImpl` method to reset the state of the counter stored in the `Count` property to zero.

```
methods (Access = protected)
   function resetImpl(obj)
      obj.Count = 0;
   end
end
```

See Also

`releaseImpl`

Topics

"Reset Algorithm State"

Introduced in R2011b

saveObjectImpl

Class: `matlab.System`

Save System object in MAT file

Syntax

```
s = saveObjectImpl(obj)
```

Description

`s = saveObjectImpl(obj)` specifies the System object properties and state values to be saved in a structure or MAT file.

If you do not define a `saveObjectImpl` method for your System object class, only public properties and properties with the `DiscreteState` attribute are saved. To save any private or protected properties or state information, you must define a `saveObjectImpl` in your class definition file.

Your `loadObjectImpl` method should correspond to your `saveObjectImpl` method to ensure that all saved properties and data are loaded.

Run-Time Details

`save` calls `saveObject`, which then calls `saveObjectImpl`. To save a System object in generated code, the object must be unlocked and it cannot contain or be a child object.

End users can use `load`, which calls `loadObjectImpl` to load a saved System object into their workspace.

Method Authoring Tips

- You must set `Access = protected` for this method.
- Save the state of an object only if the object is in use. When the user loads that saved object, it loads in that usage state.

- To save child object information, use the associated `saveObject` method within the `saveObjectImpl` method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `saveObjectImpl` method does not use the object, you can replace this input with ~.

Output Arguments

`s` — Saved data
struct | MAT file

Saved data to load.

Examples

Define Property and State Values to Save

Define what is saved for the System object. Call the base class version of `saveObjectImpl` to save public properties. Then, save any child System objects and any protected and private properties. Finally, save the state if the object is in use.

```
methods (Access = protected)
  function s = saveObjectImpl(obj)
    s = saveObjectImpl@matlab.System(obj);
    s.child = matlab.System.saveObject(obj.child);
    s.protectedprop = obj.protectedprop;
    s.pdependentprop = obj.pdependentprop;
    if isLocked(obj)
      s.state = obj.state;
    end
```

```
    end
end
```

See Also

`loadObjectImpl`

Topics

"Save and Load System Object"

Introduced in R2012b

setProperties

Class: `matlab.System`

Set property values using name-value pairs when creating System object

Syntax

```
setProperties(obj,numArgs,name1,value1,name2,value2,...)
setProperties(obj,numArgs,arg1,...,argN,
propValName1,...propValNameN)
```

Description

`setProperties(obj,numArgs,name1,value1,name2,value2,...)` provides the name-value pair inputs to the System object constructor. Use this syntax if every input must specify both name and value.

`setProperties(obj,numArgs,arg1,...,argN, propValName1,...propValNameN)` provides the value-only inputs, which you can follow with the name-value pair inputs to the System object during object construction. Use this syntax if you want to allow users to specify one or more inputs by their values only.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `setProperties` method does not use the object, you can replace this input with ~.

`numArgs` — Number of input arguments to the constructor
integer

Number of inputs passed in by the object constructor

`name1` — Property names
character vector | string

Name of property to set.

`value1` — Property value
value allowed by property

Value of the property.

`arg1,...argN` — Value-only input to property
value allowed by property

Value of property for value-only input to the object constructor.

`propValName1,...propValNameN` — Name of value-only property
character vector | string

Name of the value-only property to be set by the constructor.

Examples

Setup Value-Only Inputs

Set up an `MyFile` object so that when you use the System object, you can specify value-only inputs for `VProp1`, `VProp2`, and other property values via name-value pairs when constructing the object.

The following code shows the syntax of the constructor method needed for this situation.

```
methods
   function obj = MyFile(varargin)
      setProperties(obj,nargin,varargin{:},'VProp1','VProp2');
```

```
    end
end
```

See Also

Topics

"Set Property Values at Construction Time"

Introduced in R2011b

setupImpl

Class: `matlab.System`

Initialize System object

Syntax

```
setupImpl(obj)
setupImpl(obj,input1,input2,...)
```

Description

`setupImpl(obj)` implements one-time tasks.

`setupImpl(obj,input1,input2,...)` sets up a System object using one or more of the `stepImpl` input specifications.

Run-Time Details

`setupImpl` is called via the `setup` method. Users never call the `setup` method directly. But, `setup` is called the first time a System object is run and after a System object has been released. For details, see “Detailed Call Sequence”

Method Authoring Tips

- If your System object does not require any setup tasks, you can omit this method from your class definition file.
- Use `setupImpl` to set private properties so they do not need to be calculated each time `stepImpl` method is called.
- To acquire resources for a System object, you must use `setupImpl` instead of a constructor.
- You must set `Access = protected` for this method.
- Do not use `setupImpl` to initialize or reset states. For states, use the `resetImpl` method.

- If the System object will be used in the Simulink MATLAB System block, you cannot modify any tunable properties in the `setupImpl` method
- Do not use the `setupImpl` method to set up input values.
- Do not include validation in `setupImpl`. To validate properties or inputs use the `validatePropertiesImpl`, `validateInputsImpl`, or `setProperties` methods.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `setupImpl` method does not use the object, you can replace this input with ~.

`input1,input2,...` — Inputs to the `stepImpl` method
inputs to the System object

List the inputs to the System object. The order of inputs must match the order of inputs defined in the `stepImpl` method. `stepImpl` passes the inputs into `setupImpl` to use the specifications, such as size and data types in the one-time calculations.

Examples

Setup a File for Writing

This example shows how to open a file for writing using the `setupImpl` method in your class definition file.

```
methods (Access = protected)
   function setupImpl(obj)
      obj.pFileID = fopen(obj.Filename,'wb');
      if obj.pFileID < 0
         error('Opening the file failed');
       end
```

```
   end
end
```

Initialize Properties Based on Object Inputs

This example shows how to use `setupImpl` to specify that running the object initializes the properties of an input. In this case, calls to run the object, which includes input `u`, initialize the object states in a matrix of size `u`.

```
methods (Access = protected)
   function setupImpl(obj, u)
    obj.State = zeros(size(u),'like', u);
  end
end
```

See Also

`matlab.System` `Constructor` | `setProperties` | `validateInputsImpl` | `validatePropertiesImpl`

Topics

"Initialize Properties and Setup One-Time Calculations"
"Set Property Values at Construction Time"

Introduced in R2011b

showFiSettingsImpl

Class: `matlab.System`

Fixed point data type tab visibility for System objects

Syntax

```
flag = showFiSettingsImpl
```

Description

`flag = showFiSettingsImpl` specifies whether the Data Types tab appears on the MATLAB System block dialog box. The Data Types tab includes parameters to control processing of fixed point data the MATLAB System block. You cannot specify which parameters appear on the tab. If you implement `showFiSettingsImpl`, the simulation mode is set to code generation.

The parameters that appear on the Data Types tab, which cannot be customized, are:

- **Saturate on integer overflow** — a check box to control the action to take on integer overflow for built-in integer types. The default is that the box is checked, which indicates to saturate. This is also the default for when **Same as MATLAB** is selected as the **MATLAB System fimath** option.
- **Treat these inherited Simulink signal types as fi objects** — a pull down that indicates which inherited data types to treat as fi data types. Valid options are:
  - `Fixed point` (default)
  - `Fixed point & integer`
- **MATLAB System fimath** has two radio button options:
  - **Same as MATLAB** (default) — uses the current MATLAB fixed-point math settings.
  - **Specify Other** — enables the edit box for specifying the desired fixed-point math settings.

  For information on setting fixed-point math, see `fimath`.

Run-Time Details

`showFiSettingsImpl` is called by the MATLAB System block. For more information, see "Simulink Engine Phases Mapped to System Object Methods" (Simulink).

Method Authoring Tips

- If you do not want to display the tab, you do not need to implement this method in your class definition file.
- You must set `Access = protected` for this method.

Output Arguments

`flag` — Display Data Types tab
`false` (default) | `true`

Flag indicating whether to display the Data Types tab on the MATLAB System block mask, returned as a logical scalar value. Returning a `true` value displays the tab. A `false` value does not display the tab.

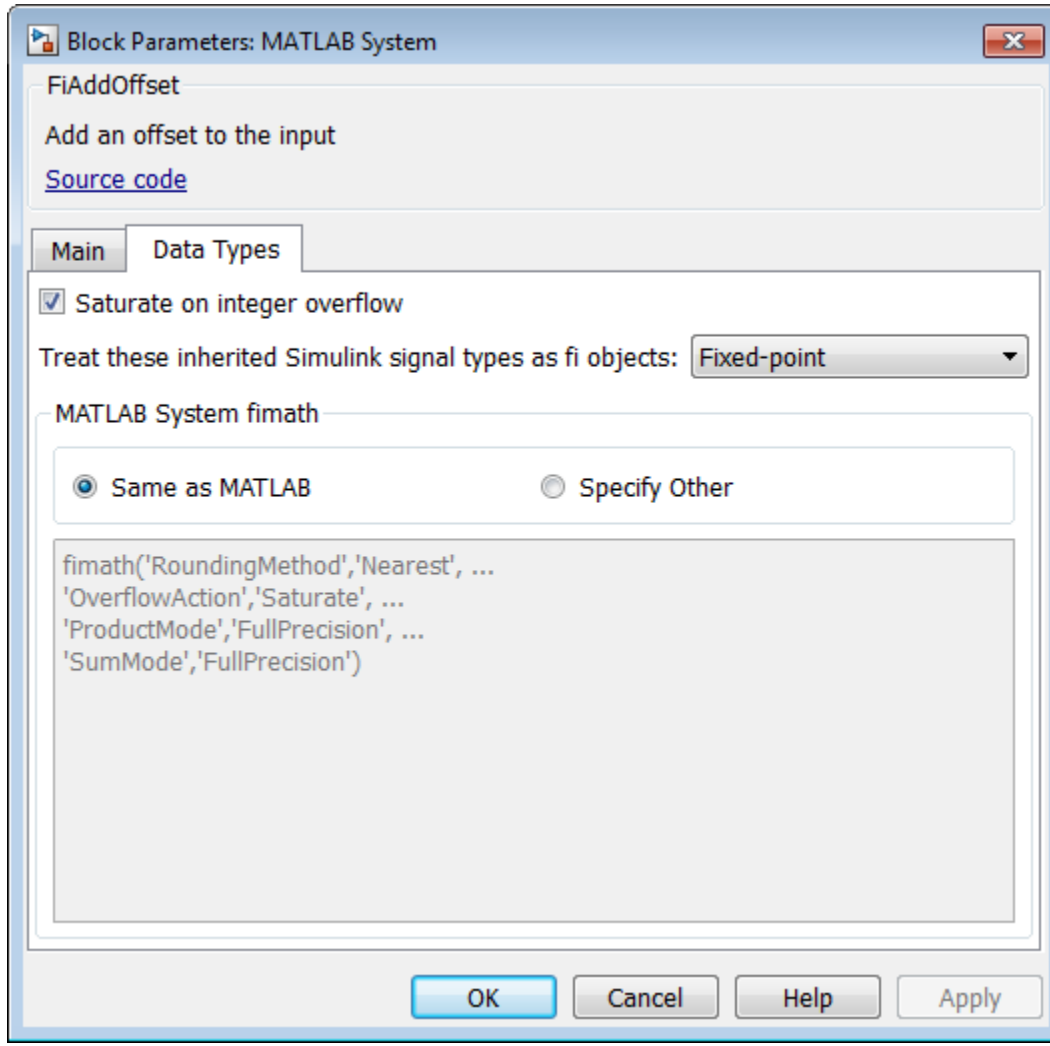
Examples

Show the Data Types Tab

Show the Data Types tab on the MATLAB System block dialog box.

```
methods (Static, Access = protected)
   function isVisible = showFiSettingsImpl
     isVisible = true;
   end
end
```

If you set the flag, `isVisible`, to `true`, the tab appears as follows when you add the object to Simulink with the MATLAB System block.



See Also

Topics

"Add Data Types Tab to MATLAB System Block" (Simulink)

Introduced in R2016a

showSimulateUsingImpl

Class: `matlab.System`

Visibility of Simulate using parameter

Syntax

```
flag = showSimulateUsingImpl
```

Description

`flag = showSimulateUsingImpl` specifies whether **Simulate using** appears on the MATLAB System block dialog box.

Run-Time Details

`showSimulateUsingImpl` is called by the MATLAB System block.

Method Authoring Tips

You must set `Access = protected` and `Static` for this method.

Output Arguments

`flag` — Display parameter on block dialog box
`true` (default) | `false`

Flag indicating whether to display the **Simulate using** parameter and dropdown list on the MATLAB System block mask. `true` displays the parameter and dropdown list. `false` hides the parameter and dropdown list.

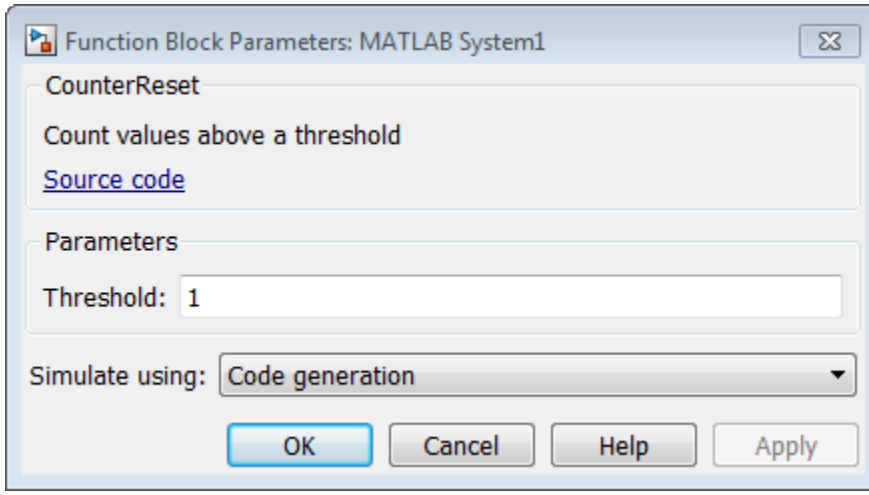
Examples

Hide the Simulate using Parameter

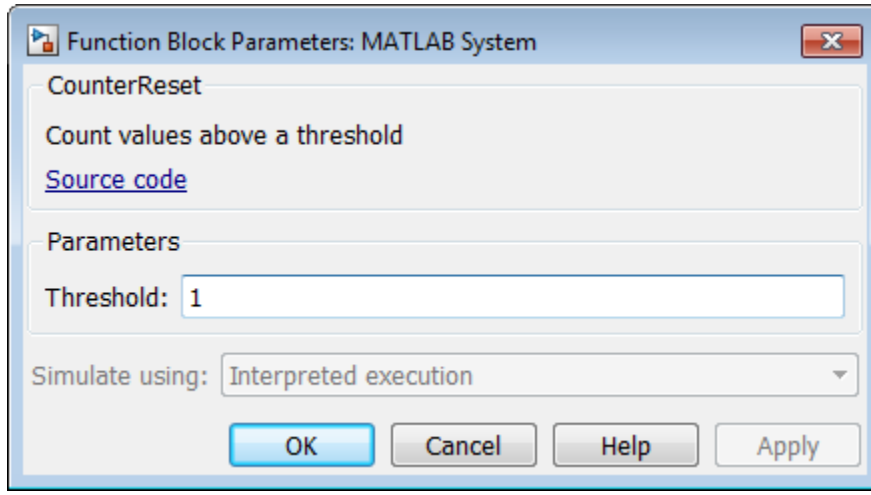
Hide the **Simulate using** parameter on the MATLAB System block dialog box.

```
methods (Static, Access = protected)
  function flag = showSimulateUsingImpl
    flag = false;
  end
end
```

If you set the flag to `true` or omit the `showSimulateUsingImpl` method, which defaults to `true`, the dialog appears as follows when you add the object to Simulink with the MATLAB System block.



If you also specify a single value for `getSimulateUsingImpl`, the dialog appears as follows when you add the object to Simulink with the MATLAB System block.



See Also

`getSimulateUsingImpl`

Topics

"Control Simulation Type in MATLAB System Block" (Simulink)

Introduced in R2015a

stepImpl

Class: `matlab.System`

System output and state update equations

Syntax

```
[output,output2,...] = stepImpl(obj,input1,input2,...)
```

Description

`[output,output2,...] = stepImpl(obj,input1,input2,...)` specifies the algorithm to execute when you run the System object. Running the object calculates the outputs and updates the object's state values using the inputs, properties, and state update equations. You can also run an object using function-like syntax instead of the `step` method. For example, if you define an FFT object using `txfourier = dsp.FFT`, you can run it simply by using `txfourier()`.

Run-Time Details

`stepImpl` is called via `step` when you run the System object. Users can also run the System object via the System object name, which calls `step`. For details, see "Detailed Call Sequence".

Method Authoring Tips

- You must set `Access = protected` for this method.
- The number of input arguments and output arguments must be greater than or equal to the numbers returned by the `getNumInputsImpl` and `getNumOutputsImpl` methods, respectively.
- Do not call `release` within the `stepImpl` method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `stepImpl` method does not use the object, you can replace this input with `~`.

`input1,input2,...` — Inputs to the System object
inputs to the System object

List the inputs to the System object. For examples of different input variations, see “Change the Number of Inputs”.

Output Arguments

`output` — Outputs returned from the System object
outputs

Output returned from the System object. For examples of different output variations, see “Change the Number of Inputs”.

Examples

Specify System Object Algorithm

Use the `stepImpl` method to increment two numbers.

```
methods (Access = protected)
   function [y1,y2] = stepImpl(obj,x1,x2)
      y1 = x1 + 1;
      y2 = x2 + 1;
```

```
    end
end
```

See Also

validateInputsImpl

Topics

"Define Basic System Objects"
"Change the Number of Inputs"

Introduced in R2011b

supportsMultipleInstanceImpl

Class: `matlab.System`

Support System object in Simulink For Each subsystem

Syntax

```
flag = supportsMultipleInstanceImpl(obj)
```

Description

`flag = supportsMultipleInstanceImpl(obj)` specifies whether the System object can be used in a Simulink For Each subsystem via the MATLAB System block. To enable For Each support, you must include the `supportsMultipleInstanceImpl` in your class definition file and have it return `true`. Do not enable For Each support if your System object allocates exclusive resources that may conflict with other System objects, such as allocating file handles, memory by address, or hardware resources.

Run-Time Details

During Simulink model compilation and propagation, the MATLAB System block calls the `supportMultipleInstance` method, which then calls the `supportsMultipleInstanceImpl` method to determine For Each support.

Method Authoring Tips

- You must set `Access = protected` for this method.
- You cannot modify any properties in this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `supportsMultipleInstanceImpl` method does not use the object, you can replace this input with ~.

Output Arguments

`flag` — Support for For Each subsystem flag
`false` (default) | `true`

Boolean value indicating whether the System object can be used in a For Each subsystem. The default value, if you do not include the `supportMultipleInstance` method, is `false`.

Examples

Enable For-Each Support for System Object

Specify in your class definition file that the System object can be used in a Simulink For Each subsystem.

```
methods (Access = protected)
   function flag = supportsMultipleInstanceImpl(obj)
       flag = true;
   end
end
```

See Also

`matlab.System`

Topics

"Enable For Each Subsystem Support" (Simulink)

Introduced in R2014b

updateImpl

Class: `matlab.System`

Update object states based on inputs

Syntax

```
updateImpl(obj,input1,input2,...)
```

Description

`updateImpl(obj,input1,input2,...)` specifies the algorithm to update the System object states. You implement this method when your algorithm outputs depend only on the object's internal state and internal properties.

Run-Time Details

`updateImpl` is called by the `update` method and after the `outputImpl` method.

For sink objects, calling `updateImpl` before `outputImpl` locks the object. For all other types of objects, calling `updateImpl` before `outputImpl` causes an error.

Method Authoring Tips

- Do not use this method to update the outputs from the inputs.
- You must set `Access = protected` for this method.
- If the System object will be used in the Simulink MATLAB System block, you cannot modify any tunable properties in this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `updateImpl` method does not use the object, you can replace this input with ~.

`input1,input2,...` — Inputs to the System object
inputs to the System object

List the inputs to the System object. The order of inputs must match the order of inputs defined in the `stepImpl` method.

Examples

Set Up Output that Does Not Depend on Current Input

Update the object with previous inputs. Use `updateImpl` in your class definition file. This example saves the `u` input and shifts the previous inputs.

```
methods (Access = protected)
   function updateImpl(obj,u)
      obj.PreviousInput = [u obj.PreviousInput(1:end-1)];
   end
end
```

See Also

`isInputDirectFeedthroughImpl` | `outputImpl`

Topics

"Use Update and Output for Nondirect Feedthrough" (Simulink)

Introduced in R2013b

validateInputsImpl

Class: `matlab.System`

Validate inputs to System object

Syntax

```
validateInputsImpl(obj,input1,input2,...)
```

Description

`validateInputsImpl(obj,input1,input2,...)` validates inputs to the System object the first time the object runs. Validation includes checking data types, complexity, cross-input validation, and validity of inputs controlled by a property value.

Run-Time Details

`validateInputImpl` is called when the object is run for the first time by `setup` and when the input size, data type, or complexity changes. For details, see “Detailed Call Sequence”.

Method Authoring Tips

- You must set `Access = protected` for this method.
- You cannot modify any properties in this method. Use the `processTunedPropertiesImpl` method or `setupImpl` method to modify properties.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `validateInputsImpl` method does not use the object, you can replace this input with ~.

`input1,input2,...` — Inputs to the System object
inputs to the System object

List the inputs to the System object. The order of inputs must match the order of inputs defined in the `stepImpl` method. Use the inputs to validate the data type, complexity, or size of inputs to the object.

Examples

Validate Input Type

Validate that the input is numeric.

```
methods (Access = protected)
   function validateInputsImpl(~,x)
      if ~isnumeric(x)
         error('Input must be numeric');
      end
   end
end
```

See Also

`setupImpl` | `validatePropertiesImpl`

Topics

"Validate Property and Input Values"

Introduced in R2012a

validatePropertiesImpl

Class: `matlab.System`

Validate property values of System object

Syntax

```
validatePropertiesImpl(obj)
```

Description

`validatePropertiesImpl(obj)` validates interdependent or interrelated property values the first time the System object runs.

Run-Time Details

`validatePropertiesImpl` is called by `setup` when the object is run for the first time and when any tunable property changes. For details, see “Detailed Call Sequence”.

Method Authoring Tips

- You must set `Access = protected` for this method.
- You cannot modify any properties in this method. Use the `processTunedPropertiesImpl` method or `setupImpl` method to modify properties.
- To check if a property has changed since `stepImpl` was last called, use `isChangedProperty(obj,property)` within `validatePropertiesImpl`.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `validatePropertiesImpl` method does not use the object, you can replace this input with ~.

Examples

Validate an Interdependent Property

To validate the value of `increment`, validate that the `useIncrement` property is `true` and that the value of the `increment` property is greater than zero.

```
methods (Access = protected)
   function validatePropertiesImpl(obj)
       if obj.useIncrement && obj.increment < 0
          error('The increment value must be positive');
       end
    end
end
```

Alternative Functionality

If your property does not depend on any other property, you can use MATLAB class property validators at the property definition. For more information, see “Validate Property and Input Values”.

See Also

`processTunedPropertiesImpl` | `setupImpl` | `validateInputsImpl`

Topics

“Validate Property and Input Values”

Introduced in R2012a

matlab.system.display.Action class

Package: `matlab.system.display`

Custom button

Syntax

```
matlab.system.display.Action(action)
matlab.system.display.Action(action,Name,Value)
```

Description

`matlab.system.display.Action(action)` specifies a button to display on the MATLAB System block. This button executes a function by launching a System object method or invoking any MATLAB function or code.

A typical button function launches a figure. The launched figure is decoupled from the block dialog box. Changes to the block are not synced to the displayed figure.

You define `matlab.system.display.Action` within the `getPropertyGroupsImpl` method in your class definition file. You can define multiple buttons using separate instances of `matlab.system.display.Action` in your class definition file.

`matlab.system.display.Action(action,Name,Value)` includes `Name,Value` pair arguments, which you can use to specify any properties.

Input Arguments

`action`

Action taken when the user presses the specified button on the MATLAB System block dialog. The action is defined as a function handle or as a MATLAB command. If you define the action as a function handle, the function definition must define two inputs. These inputs are a `matlab.system.display.ActionData` object and a System object instance, which can be used to invoke a method.

A `matlab.system.display.ActionData` object is the callback object for a display action. You use the `UserData` property of `matlab.system.display.ActionData` to store persistent data, such as a figure handle.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Properties

You specify these properties as part of the input using `Name,Value` pair arguments. Optionally, you can define them using `object.property` syntax.

- `ActionCalledFcn` — Action to take when the button is pressed. You cannot specify this property using a Name-Value pair argument.
- `Label` — Text to display on the button. The default value is an empty character vector.
- `Description` — Text for the button tooltip. The default value is an empty character vector.
- `Placement` — Character vector indicating where on a separate row in the property group to place the button. Valid values are `'first'`, `'last'`, or a property name. If you specify a property name, the button is placed above that property. The default value is `'last'`.
- `Alignment` — Character vector indicating how to align the button. Valid values are `'left'` and `'right'`. The default value is `'left'`.

Examples

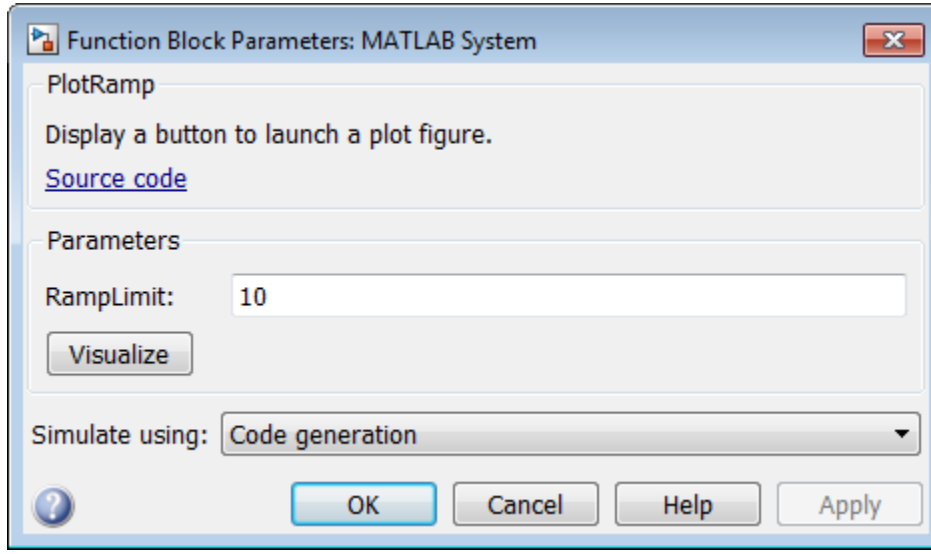
Define Button on MATLAB System Block

Define a **Visualize** button and its associated function to open a figure that plots a ramp using the parameter values in the block dialog.

```
methods(Static,Access = protected)
  function group = getPropertyGroupsImpl
```

```
  group = matlab.system.display.Section(mfilename('class'));
  group.Actions = matlab.system.display.Action(@(~,obj)...
      visualize(obj),'Label','Visualize');
  end
end

methods
  function obj = PlotRamp(varargin)
    setProperties(obj,nargin,varargin{:});
  end

  function visualize(obj)
    figure;
    d = 1:obj.RampLimit;
    plot(d);
  end
end
```

When you specify the System object in the MATLAB System block, the resulting block dialog box appears as follows.



To open the same figure, rather than multiple figures, when the button is pressed more than once, use this code instead.

```
methods(Static,Access = protected)
  function group = getPropertyGroupsImpl
    group = matlab.system.display.Section(mfilename('class'));
    group.Actions = matlab.system.display.Action(@(actionData,obj)...
       visualize(obj,actionData),'Label','Visualize');
  end
end

methods
  function obj = ActionDemo(varargin)
    setProperties(obj,nargin,varargin{:});
  end

  function visualize(obj,actionData)
    f = actionData.UserData;
    if isempty(f) || ~ishandle(f)
      f = figure;
      actionData.UserData = f;
    else
        figure(f); % Make figure current
    end

    d = 1:obj.RampLimit;
    plot(d);
  end
end
```

See Also

`getPropertyGroupsImpl` | `matlab.system.display.Section` | `matlab.system.display.SectionGroup`

Topics

"Classes"
Class Attributes
Property Attributes
"Add Button to MATLAB System Block" (Simulink)

matlab.system.display.Header class

Package: `matlab.system.display`

Header for System objects properties

Syntax

```
matlab.system.display.Header(N1,V1,...Nn,Vn)
matlab.system.display.Header(Obj,...)
```

Description

`matlab.system.display.Header(N1,V1,...Nn,Vn)` specifies a header for the System object, with the header properties defined in Name-Value (`N`,`V`) pairs. You use `matlab.system.display.Header` within the `getHeaderImpl` method. The available header properties are

- `Title` — Header title. The default value is an empty character vector.
- `Text` — Header description. The default value is an empty character vector.
- `ShowSourceLink` — Show link to source code for the object.

`matlab.system.display.Header(Obj,...)` creates a header for the specified System object (`Obj`) and sets the following property values:

- `Title` — Set to the `Obj` class name.
- `Text` — Set to help summary for `Obj`.
- `ShowSourceLink` — Set to `true` if `Obj` is MATLAB code. In this case, the **Source Code** link is displayed. If `Obj` is P-coded and the source code is not available, set this property to `false`.

You can use `mfilename('class')` from within this method to get the name of the System object. If you set any Name-Value pairs, those property values override the default settings.

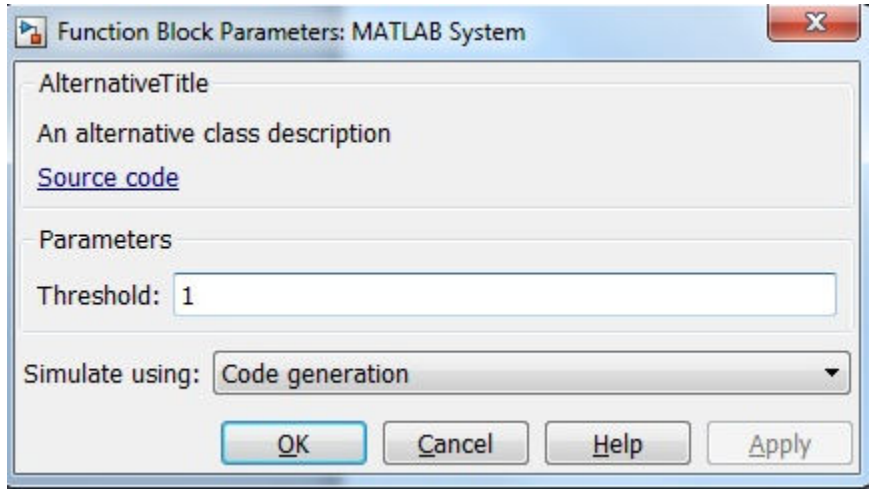
Examples

Define System Block Header

Define a header in your class definition file.

```
methods (Static, Access = protected)
   function header = getHeaderImpl
      header = matlab.system.display.Header(mfilename('class'), ...
          'Title','AlternativeTitle',...
          'Text','An alternative class description');
   end
end
```

The resulting output appears as follows. In this case, **Source code** appears because the `ShowSourceLink` property was set to `true`.



See Also

`getHeaderImpl` | `matlab.system.display.Section` | `matlab.system.display.SectionGroup`

Topics

"Classes"
"Add Header to MATLAB System Block" (Simulink)

matlab.system.display.Section class

Package: `matlab.system.display`

Property group section for System objects

Syntax

```
matlab.system.display.Section(N1,V1,...Nn,Vn)
matlab.system.display.Section(Obj,...)
```

Description

`matlab.system.display.Section(N1,V1,...Nn,Vn)` creates a property group section for displaying System object properties, which you define using property Name-Value pairs (`N`,`V`). You use `matlab.system.display.Section` to define property groups using the `getPropertyGroupsImpl` method. The available Section properties are

- `Title` — Section title. The default value is an empty character vector.
- `TitleSource` — Source of section title. Valid values are `'Property'` and `'Auto'`. The default value is `'Property'`, which uses the character vector from the `Title` property. If the `Obj` name is given, the default value is `Auto`, which uses the `Obj` name.
- `Description` — Section description. The default value is an empty character vector.
- `PropertyList` — Section property list as a cell array of property names. The default value is an empty array. If the `Obj` name is given, the default value is all eligible display properties.

Note Certain properties are not eligible for display either in a dialog box or in the System object summary on the command-line. Property types that cannot be displayed are: hidden, abstract, private or protected access, discrete state, and continuous state. Dependent properties do not display in a dialog box, but do display in the command-line summary.

`matlab.system.display.Section(Obj,...)` creates a property group section for the specified System object (`Obj`) and sets the following property values:

- `TitleSource` — Set to `'Auto'`, which uses the `Obj` name.
- `PropertyList` — Set to all publicly-available properties in the `Obj`.

You can use `mfilename('class')` from within this method to get the name of the System object. If you set any Name-Value pairs, those property values override the default settings.

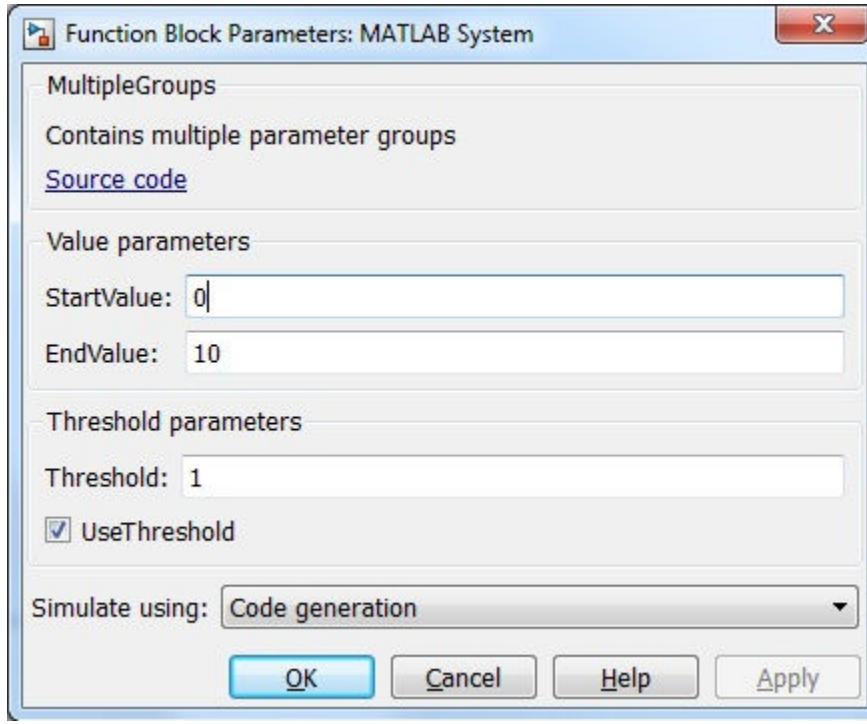
Examples

Define Property Groups

Define two property groups in your class definition file by specifying their titles and property lists.

```
methods (Static, Access = protected)
   function groups = getPropertyGroupsImpl
      valueGroup = matlab.system.display.Section(...
           'Title','Value parameters',...
           'PropertyList',{'StartValue','EndValue'});

      thresholdGroup = matlab.system.display.Section(...
           'Title','Threshold parameters',...
           'PropertyList',{'Threshold','UseThreshold'});
      groups = [valueGroup,thresholdGroup];
   end
end
```

When you specify the System object in the MATLAB System block, the resulting dialog box appears as follows.



See Also

`getPropertyGroupsImpl` | `matlab.system.display.Header` | `matlab.system.display.SectionGroup`

Topics

"Add Property Groups to System Object and MATLAB System Block" (Simulink)

matlab.system.display.SectionGroup class

Package: `matlab.system.display`

Section group for System objects

Syntax

```
matlab.system.display.SectionGroup(N1,V1,...Nn,Vn)
matlab.system.display.SectionGroup(Obj,...)
```

Description

`matlab.system.display.SectionGroup(N1,V1,...Nn,Vn)` creates a group for displaying System object properties and display sections created with `matlab.system.display.Section`. You define such sections or properties using property Name-Value pairs (`N`,`V`). A section group can contain both properties and sections. You use `matlab.system.display.SectionGroup` to define section groups using the `getPropertyGroupsImpl` method. Section groups display as separate tabs in the MATLAB System block. The available Section properties are

- `Title` — Group title. The default value is an empty character vector.
- `TitleSource` — Source of group title. Valid values are `'Property'` and `'Auto'`. The default value is `'Property'`, which uses the character vector from the `Title` property. If the `Obj` name is given, the default value is `Auto`, which uses the `Obj` name. In the System object property display at the MATLAB command line, you can omit the default "Main" title for the first group of properties by setting `TitleSource` to `'Auto'`.
- `Description` — Group or tab description that appears above any properties or panels. The default value is an empty character vector.
- `PropertyList` — Group or tab property list as a cell array of property names. The default value is an empty array. If the `Obj` name is given, the default value is all eligible display properties.
- `Sections` — Group sections as an array of section objects. If the `Obj` name is given, the default value is the default section for the `Obj`.

`matlab.system.display.SectionGroup(Obj,...)` creates a section group for the specified System object (`Obj`) and sets the following property values:

- `TitleSource` — Set to `'Auto'`.
- `Sections` — Set to `matlab.system.display.Section` object for `Obj`.

You can use `mfilename('class')` from within this method to get the name of the System object. If you set any Name-Value pairs, those property values override the default settings.

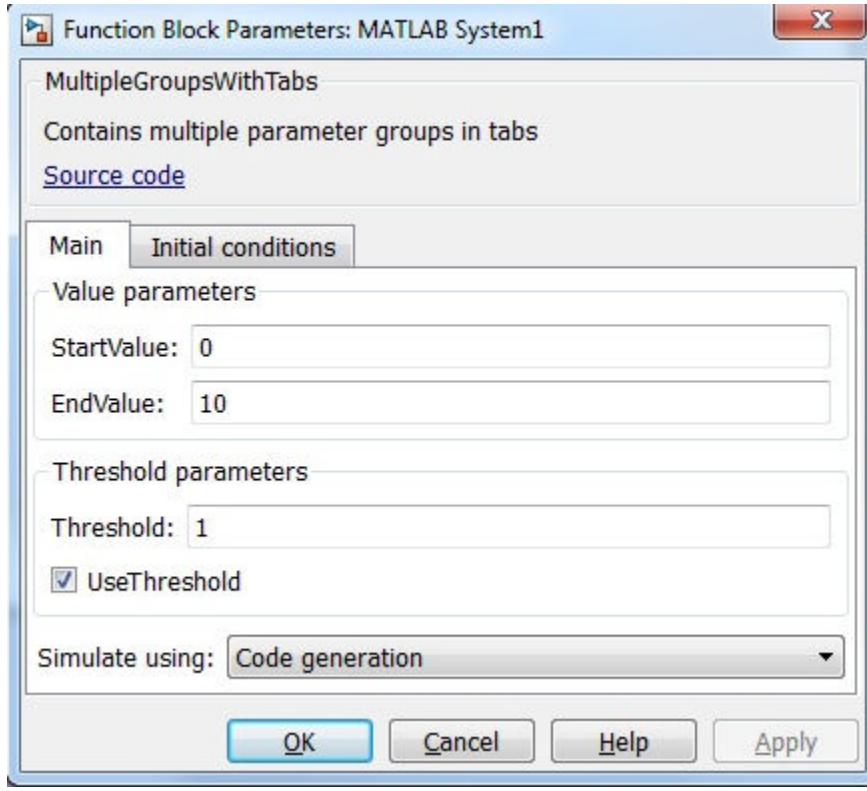
Examples

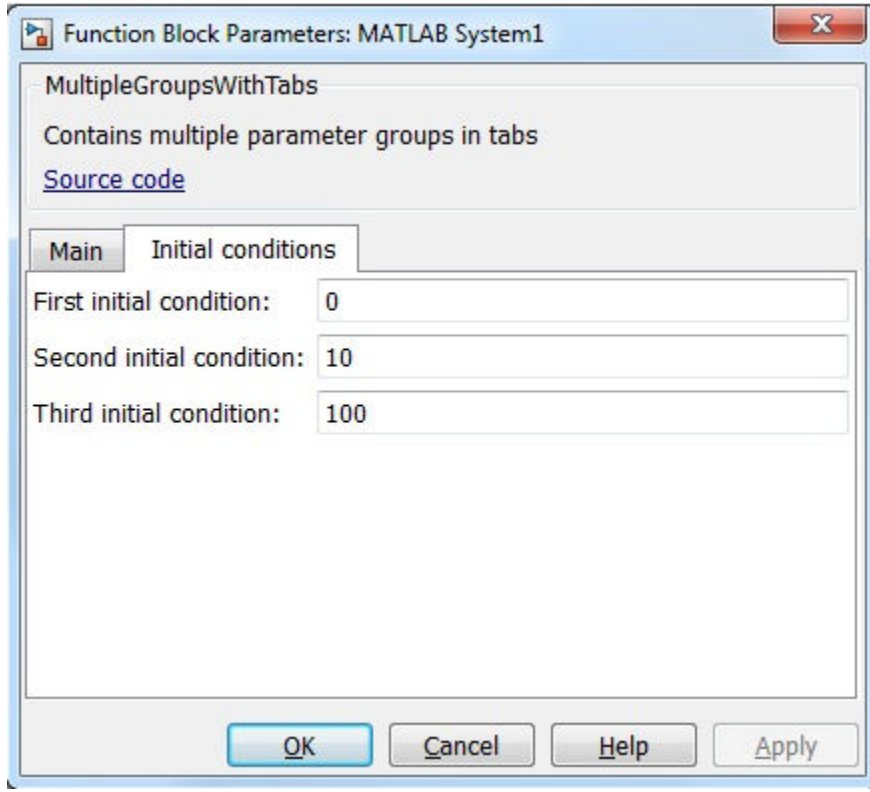
Define Block Dialog Tabs

Define in your class definition file two tabs, each containing specific properties. For this example, you use the `matlab.system.display.SectionGroup`, `matlab.system.display.Section`, and `getPropertyGroupsImpl` methods.

```
methods (Static, Access = protected)
   function groups = getPropertyGroupsImpl
      valueGroup = matlab.system.display.Section(...
           'Title','Value parameters',...
           'PropertyList',{'StartValue','EndValue'});

      thresholdGroup = matlab.system.display.Section(...
           'Title','Threshold parameters',...
           'PropertyList',{'Threshold','UseThreshold'});

      mainGroup = matlab.system.display.SectionGroup(...
           'Title','Main', ...
           'Sections',[valueGroup,thresholdGroup]);

      initGroup = matlab.system.display.SectionGroup(...
           'Title','Initial conditions', ...
           'PropertyList',{'IC1','IC2','IC3'});

      groups = [mainGroup,initGroup];
   end
end
```

The resulting dialog appears as follows when you add the object to Simulink with the MATLAB System block.





See Also

`getPropertyGroupsImpl` | `matlab.system.display.Header` | `matlab.system.display.Section`

Topics

"Add Property Groups to System Object and MATLAB System Block" (Simulink)

matlab.system.mixin.CustomIcon class

Package: `matlab.system.mixin`

Custom icon mixin class

Description

`matlab.system.mixin.CustomIcon` is a class that specifies the `getIcon` method. This method customizes the name of the icon used for the System object implemented through a MATLAB System block.

To use this method, you must subclass from this class in addition to the `matlab.System` base class. Type the following syntax as the first line of your class definition file, where `ObjectName` is the name of your object:

```
classdef ObjectName < matlab.system &...
    matlab.system.mixin.CustomIcon
```

The `matlab.system.mixin.CustomIcon` class is a `handle` class.

See Also

`matlab.System` | `matlab.system.display.Icon`

Topics

"Add Text to Block Icon" (Simulink)

Introduced in R2017a

matlab.system.mixin.FiniteSource class

Package: `matlab.system.mixin`

Finite source mixin class

Description

`matlab.system.mixin.FiniteSource` is a class that specifies the `isDone` method, which reports the state of a finite data source, such as an audio file.

To use this method, you must subclass from this class in addition to the `matlab.System` base class. Type the following syntax as the first line of your class definition file, where `ObjectName` is the name of your object:

```
classdef ObjectName < matlab.System &...
    matlab.system.mixin.FiniteSource
```

Methods

isDoneImpl	End-of-data flag

See Also

`matlab.System`

Topics

"Classes"
"Define Finite Source Objects"

isDoneImpl

Class: `matlab.system.mixin.FiniteSource`
Package: `matlab.system.mixin`

End-of-data flag

Syntax

```
status = isDoneImpl(obj)
```

Description

`status = isDoneImpl(obj)` specifies whether the end of the data has been reached. `isDoneImpl` should return `true` when data from a finite source has been exhausted, typically by having read and output all data from the source. You should also define the result of future reads from an exhausted source in the `isDoneImpl` method.

Run-Time Details

`isDoneImpl` is called by the `isDone` object function.

Method Authoring Tips

You must set `Access = protected` for this method.

Input Arguments

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `isDoneImpl` method does not use the object, you can replace this input with ~.

Output Arguments

`status` — End-of-data flag
`true` | `false`

Flag, `true` or `false`, that indicates if an end-of-data condition has occurred or not.

Examples

Check for End-of-Data

Set up the `isDoneImpl` method in your class definition file so the `isDone` method checks whether the object has completed eight iterations.

```
methods (Access = protected)
   function bdone = isDoneImpl(obj)
      bdone = obj.NumIters==8;
   end
end
```

See Also

`matlab.system.mixin.FiniteSource`

Topics

"Define Finite Source Objects"

matlab.system.mixin.Nondirect class

Package: `matlab.system.mixin`

Nondirect feedthrough mixin class

Description

`matlab.system.mixin.Nondirect` is a class that uses the `output` and `update` methods to process nondirect feedthrough data through a System object.

For System objects that use direct feedthrough, the object's input is needed to generate the output at that time. For these direct feedthrough objects, running the System object calculates the output and updates the state values. For nondirect feedthrough, however, the object's output depends only on the internal states at that time. The inputs are used to update the object states. For these objects, calculating the output with `outputImpl` is separated from updating the state values with `updateImpl`. If you use the `matlab.system.mixin.Nondirect` mixin and include the `stepImpl` method in your class definition file, an error occurs. In this case, you must include the `updateImpl` and `outputImpl` methods instead.

The following cases describe when System objects in Simulink use direct or nondirect feedthrough.

- System object supports code generation and does not inherit from the `Propagates` mixin — Simulink automatically infers the direct feedthrough settings from the System object code.
- System object supports code generation and inherits from the `Propagates` mixin — Simulink does not automatically infer the direct feedthrough settings. Instead, it uses the value returned by the `isInputDirectFeedthroughImpl` method.
- System object does not support code generation — Default `isInputDirectFeedthroughImpl` method returns false, indicating that direct feedthrough is not enabled. To override the default behavior, implement the `isInputDirectFeedthroughImpl` method in your class definition file.

Use the `Nondirect` mixin to allow a System object to be used in a Simulink feedback loop. A delay object is an example of a nondirect feedthrough object.

To use this mixin, you must subclass from this class in addition to subclassing from the `matlab.System` base class. Type the following syntax as the first line of your class definition file, where `ObjectName` is the name of your object:

```
classdef ObjectName < matlab.system & matlab.system.mixin.Nondirect
```

See Also

`matlab.System`

Topics

"Use Update and Output for Nondirect Feedthrough" (Simulink)

Introduced in R2013b

matlab.system.mixin.Propagates class

Package: `matlab.system.mixin`

Signal characteristics propagation mixin class

Description

`matlab.system.mixin.Propagates` specifies the output size, data type, and complexity of a System object. Use this mixin class and its methods when you will include your System object in Simulink via the MATLAB System block. This mixin is called by the MATLAB System block during Simulink model compilation.

Note If your System object has exactly one input and one output and no discrete property states, or if you do not need bus support, you do not have to implement any of these methods. The `matlab.system.mixin.Propagates` provides default values in these cases.

Implement the methods of this class when Simulink cannot infer the output specifications directly from the inputs or when you want bus support. If you do not include this mixin, Simulink cannot propagate the output or bus data type, an error occurs.

To use this mixin, subclass from this `matlab.system.mixin.Propagates` in addition to subclassing from the `matlab.System` base class. Type the following syntax as the first line of your class definition file. `ObjectName` is the name of your System object.

```
classdef ObjectName < matlab.System &...
    matlab.system.mixin.Propagates
```

The `matlab.system.mixin.Propagates` class is a `handle` class.

See Also

`matlab.System`

Topics

“Set Output Data Type” (Simulink)
“Set Output Size” (Simulink)
“Set Output Complexity” (Simulink)
“Set Fixed- or Variable-Size Output” (Simulink)
“Set Discrete State Output Specification” (Simulink)

Introduced in R2014a

matlab.system.StringSet class

Package: `matlab.system`

Set of valid character vector values

Note The `matlab.System.StringSet` class will be removed in a future release. To create properties with a finite list of values, use property validation with `mustBeMember` or enumerations instead. See “Limit Property Values to Finite List”.

Description

`matlab.system.StringSet` specifies a list of valid character vector values for a property. This class validates the character vector in the property and enables tab completion for the property value. A StringSet allows only predefined or customized character vectors as values for the property.

A `StringSet` uses two linked properties, which you must define in the same class. One is a public property that contains the current character vector value. This public property is displayed to the user. The other property is a hidden property that contains the list of all possible character vector values. This hidden property should also have the constant attribute so that the values cannot be changed by the user.

The following considerations apply when using `StringSets`:

- The property that holds the current character vector can have any name.
- The property that holds the `StringSet` must use the same name as the property with the suffix “Set” appended to it. This property is an instance of the `matlab.system.StringSet` class.
- Valid character vectors, defined in the `StringSet`, must be declared using a cell array. The cell array cannot be empty nor can it have any empty character vectors. Valid character vectors must be unique and are case-sensitive.
- The property must be set to a valid `StringSet` value.

The `matlab.system.StringSet` class is a `handle` class.

Examples

Set StringSet Property Values

Set the property, `Flavor`, and the `StringSet` property, `FlavorSet` in your class definition file.

```
properties
  Flavor = 'Chocolate'
end

properties (Hidden,Constant)
  FlavorSet = ...
    matlab.system.StringSet({'Vanilla','Chocolate'})
end
```

Compatibility Considerations

`matlab.System.StringSet` class will be removed

Not recommended starting in R2018a

The `matlab.System.StringSet` class will be removed in a future release. To create properties with a finite list of values, use property validation with `mustBeMember` or enumerations instead. See “Limit Property Values to Finite List”.

Remove matlab.system.StringSet from the `classdef` line of your System object.

See Also

`matlab.System` | `mustBeMember`

Topics

“Limit Property Values to Finite List”
“Properties”
“Enumerations”

Introduced in R2011b

matlab.system.display.Icon class

Package: `matlab.system.display`

Custom icon image

Syntax

```
icon = matlab.system.display.Icon(imageFile)
```

Description

`icon = matlab.system.display.Icon(imageFile)` sets the `imageFile` image as the MATLAB System block icon. To set the icon image, use the `icon` output argument from `getIconImpl`.

Input Arguments

`imageFile` — Image file
character array

Image file to display on the block icon, specified as a character array. If the image is not on the path, use the full path to your image file.

The image file must be in a file format supported for block masks. See “Draw Static Icon” (Simulink).

Example: `"image.png"`

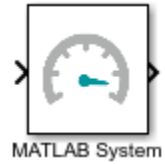
Examples

Add Image to MATLAB System Block

Define an image in your class definition file.

```
    methods(Access = protected)
        function icon = getIconImpl(~)
            % Define icon for MATLAB System block
            icon = matlab.system.display.Icon("my_icon.png");
        end
    end
```

The image now appears on the MATLAB System block icon.



See Also

`getIconImpl` | `matlab.system.mixin.CustomIcon`

Topics

Class Attributes
Property Attributes
“Customize System Block Appearance” (Simulink)

Introduced in R2017a

matlab.system.mixin.SampleTime class

Control sample time for System objects in Simulink

Description

`matlab.system.mixin.SampleTime` specifies the sample time specifications for a System object when it is included in a MATLAB System block. Inherit from this mixin class and use its methods to control the sample time of your System object in Simulink, via the MATLAB System block.

With this mixin, you can:

- Specify the sample time type
- Specify the sample time
- Customize the sample time with offsets and tick times
- Get the current simulation time

System objects that inherit from this mixin class must also inherit from `matlab.System`. For example:

```
classdef MySystemObject < matlab.System & matlab.system.mixin.SampleTime
```

The `matlab.system.mixin.SampleTime` class is a `handle` class.

Methods

See Also

Classes
`matlab.System`

Blocks
MATLAB System

Topics

"Specify Sample Time for MATLAB System Block System Objects" (Simulink)

Introduced in R2017b

clone

Create duplicate System object

Syntax

```
objClone = clone(obj)
```

Description

`objClone = clone(obj)` creates a copy of the System object `obj` that has the same property values and states as `obj`. If the object you clone is in use and you cannot change nontunable properties, the new object is also considered "in use".

Examples

Copy an Object

Create a `Counter` object and set a property.

```
obj = Counter;
obj.UseIncrement = false
```

```
obj = 
  Counter with properties:

    UseIncrement: false
    UseWrapValue: true
      StartValue: 1
       Increment: 1
       WrapValue: 10
```

Clone the object.

```
obj2 = clone(obj)
```

```
obj2 =
  Counter with properties:

    UseIncrement: false
    UseWrapValue: true
      StartValue: 1
       Increment: 1
       WrapValue: 10
```

Input Arguments

`obj` — System object to clone
System object

System object that you want to clone.

Output Arguments

`objClone` — Cloned System object
System object

Copy of the System object that has the same properties and state as the original object.

See Also

`isLocked`

Topics

"System Design in MATLAB Using System Objects"

Introduced in R2010a

getNumInputs

Number of inputs required to call the System object

Syntax

```
num = getNumInputs(obj)
```

Description

`num = getNumInputs(obj)` returns an integer, `num`, representing the number of inputs (not counting the object itself) that you must use when calling the object. This value changes when you alter properties that turn inputs on or off.

If the object is a source object, `num` equals zero.

Examples

Get Number of Inputs for an Object

Create a `Counter` object and set a property.

```
obj = Counter;
obj.UseIncrement = false
```

```
obj = 
  Counter with properties:

    UseIncrement: false
    UseWrapValue: true
      StartValue: 1
       Increment: 1
       WrapValue: 10
```

Query the number of inputs and outputs needed to call the object.

```
in = getNumInputs(obj)
```

```
in = 1
```

```
out = getNumOutputs(obj)
```

```
out = 1
```

Input Arguments

`obj` — System object to query
System object

System object to query for the number of inputs.

Output Arguments

`num` — Number of inputs
nonnegative integer

The number of inputs required to call the object, returned as an integer.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For details, see “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`getNumOutputs`

Topics

“System Design in MATLAB Using System Objects”

Introduced in R2010a

getNumOutputs

Number of outputs from calling the System object

Syntax

```
num = getNumOutputs(obj)
```

Description

`num = getNumOutputs(obj)` returns the number of outputs, `num`, from calling the object. This value changes when you alter properties that turn outputs on or off.

If the object is a sink object, `num` equals zero.

Examples

Get Number of Outputs for an Object

Create a `Counter` object and set a property.

```
obj = Counter;
obj.UseIncrement = false
```

```
obj = 
  Counter with properties:

    UseIncrement: false
    UseWrapValue: true
      StartValue: 1
       Increment: 1
       WrapValue: 10
```

Get the number of inputs and outputs needed to call the object.

```
in = getNumInputs(obj)
```

```
in = 1
```

```
out = getNumOutputs(obj)
```

```
out = 1
```

Input Arguments

`obj` — System object to query
System object handle

System object to query for the number of outputs.

Output Arguments

`num` — Number of outputs
integer

The number of outputs from calling the System object.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For details, see “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`getNumInputs`

Topics

“System Design in MATLAB Using System Objects”

Introduced in R2010a

isDone

End-of-data status

Syntax

```
tf = isDone(obj)
```

Description

`tf = isDone(obj)` returns `true` when the data from a finite source has been exhausted, typically because all data has been read. The System object `obj` is often a file reader, and `isDone` looks for the end of the file. Otherwise, `isDone` returns `false`.

Examples

Read MAT File

This example shows how to read a MAT file using the `dsp.MatFileReader` object.

Note This example requires DSP System Toolbox™.

```
filename = [tempname '.mat'];               % Create variable name
originalData = rand(10,2);
save(filename,'originalData','-v7.3');      % Write to MAT file

mfr = dsp.MatFileReader(filename,'VariableName',...
   'originalData','SamplesPerFrame', 4);
while ~isDone(mfr)            % Stream data into MATLAB
    isDone(mfr)               % Display zero if end-of-data not reached
    finalData = mfr()         % Display read frame
end
```

Input Arguments

`obj` — System object to query
System object

System object whose status you want to query.

Output Arguments

`tf` — True or false
`0` | `1`

True or false result, returned as:

- Logical `1` (done) — End-of-data has been reached.
- Logical `0` (not done) — End-of-data has not been reached.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

For details, see “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Topics
“Summary of Call Sequence”
“System Design in MATLAB Using System Objects”

Introduced in R2010a

isLocked

Determine if System object is in use

Syntax

```
tf = isLocked(obj)
```

Description

`tf = isLocked(obj)` returns logical `1` (`true`) if the System object is in use. Otherwise, `isLocked` returns logical `0` (`false`).

The object performs an internal initialization the first time that you call the object. After initialization, the object is considered in-use and you cannot change nontunable properties and other specifications determined by the System object author. While an object is in use, `isLocked` returns `1` (`true`). To allow changes to the object, call `release`.

Examples

Check If Object Is Being Used

Create a `Counter` object and set a property.

```
obj = Counter;
obj.UseIncrement = false
```

```
obj = 
  Counter with properties:

    UseIncrement: false
    UseWrapValue: true
      StartValue: 1
       Increment: 1
```

```
       WrapValue: 10
```

Call the object.

```
step(obj,1)
```

```
ans = 2
```

Check if the object is in use.

```
isLocked(obj)
```

```
ans = logical
   1
```

Release the object and check if the object is still locked.

```
reset(obj)
release(obj)
```

```
isLocked(obj)
```

```
ans = logical
   0
```

Input Arguments

`obj` — System object to query
System object

System object whose status you want to query.

Output Arguments

`tf` — True or false
`0` | `1`

True or false result, returned as logical `1` or `0` indicating the status of the System object as in use (locked) or not.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For details, see “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Topics

“Summary of Call Sequence”
“System Design in MATLAB Using System Objects”

Introduced in R2010a

release

Release resources and allow changes to System object property values and input characteristics

Syntax

```
release(obj)
```

Description

`release(obj)` releases system resources such as memory, file handles, or hardware connections, and allows you to change properties and input characteristics.

Examples

Release and Reset an Object

Create a `Counter` object and set a property.

```
obj = Counter;
obj.UseIncrement = false
```

```
obj = 
  Counter with properties:

    UseIncrement: false
    UseWrapValue: true
      StartValue: 1
       Increment: 1
       WrapValue: 10
```

Reset and release the object.

```
reset(obj)
release(obj)
```

Input Arguments

obj — System object to release
System object

System object you want to release.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

You can use `release` on a System object in code generated from MATLAB, but after you release its resources, you cannot use that System object again.

For details, see “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`reset`

Topics

“System Design in MATLAB Using System Objects”

Introduced in R2010a

reset

Reset internal states of System object

Syntax

```
reset(obj)
```

Description

`reset(obj)` resets the internal states and input properties of the System object to their initial values.

- If `obj` writes or reads a file, `reset` resets the object to the beginning of the file.
- If `obj` changes properties, `reset` resets the properties to their initial default values.
- If `obj` uses a random number generation seed, `reset` resets the seed property.

Examples

Reset Object

Create a `Counter` object and set a property.

```
obj = Counter;
obj.UseIncrement = false
```

```
obj = 
  Counter with properties:

    UseIncrement: false
    UseWrapValue: true
      StartValue: 1
       Increment: 1
       WrapValue: 10
```

Reset and release the object.

```
reset(obj)
release(obj)
```

Input Arguments

obj — System object to reset
System object

System object whose state you want to reset.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For details, see “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`release`

Topics

“System Design in MATLAB Using System Objects”

Introduced in R2010a

step

Run System object algorithm

Syntax

`step(obj, ___ )`

Description

`step(obj, ___ )` calls the System object and runs the algorithm. Depending on your System object, `step` can return output arguments.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

For details about the algorithm and input variables, see the individual System object reference pages.

Examples

Run an Object Using `step`

Create a `Counter` object and set a property.

```
obj = Counter;
obj.UseIncrement = false
```

```
obj = 
  Counter with properties:
```

```
    UseIncrement: false
    UseWrapValue: true
      StartValue: 1
       Increment: 1
       WrapValue: 10
```

Run the counter.

```
step(obj,1)
```

```
ans = 2
```

Input Arguments

obj — System object to run
System object

System object whose algorithm you want to run. Depending on the System object, additional input arguments might be required.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For details, see “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`reset`

Topics

“System Design in MATLAB Using System Objects”

Introduced in R2010a